CHƯƠNG 2
Lập trình hợp ngữ 8051

2.1 Bên trong 8051.

Trong phần này chúng ta nghiên cứu các thanh ghi chính của 8051 và trình bày cách sử dụng với các lệnh đơn giản MOV và ADD.

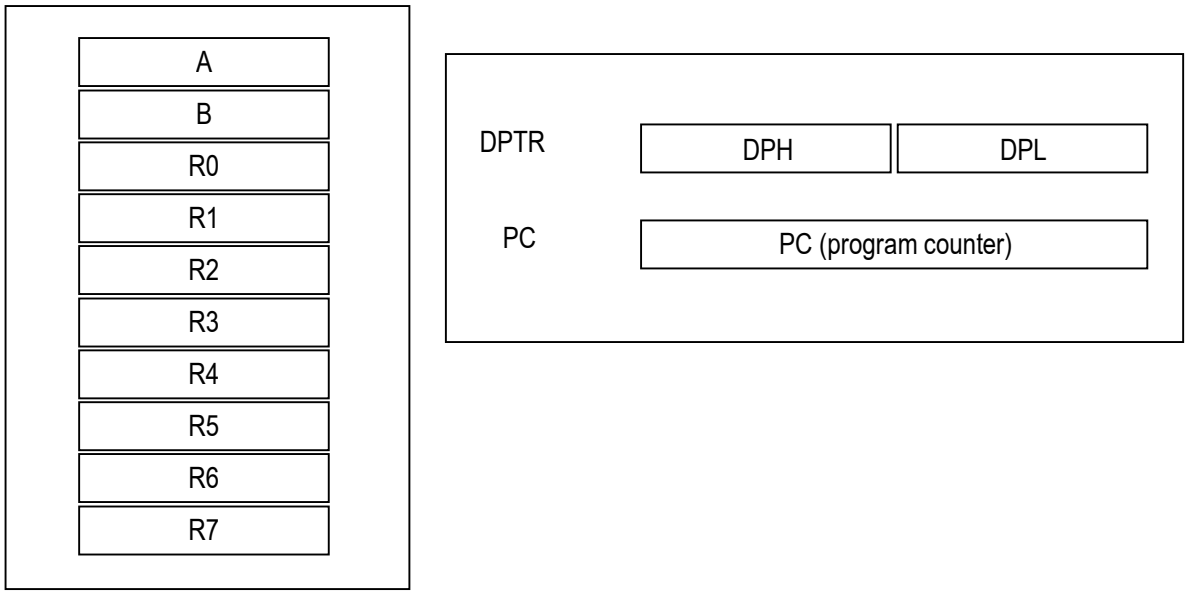
2.1.1 Các thanh ghi.

Trong CPU các thanh ghi được dùng để lưu cất thông tin tạm thời, những thông tin này có thể là một byte dữ liệu cần được sử lý hoặc là một địa chỉ đến dữ liệu cần được nạp. Phần lớn các thanh ghi của 8051 là các thanh ghi 8 bit. Trong 8051 chỉ có một kiểu dữ liệu: Loại 8 bit, 8 bit của một thanh ghi được trình bày như sau:

D7	D6	D5	D4	D3	D2	D1	D0

với MSB là bit có giá trị cao nhất D7 cho đến LSB là bit có giá trị thấp nhất D0. (MSB - Most Sigfican bit và LSB - Leart Significant Bit). Với một kiểu dữ liệu 8 bit thì bất kỳ dữ liệu nào lớn hơn 8 bit đều phải được chia thành các khúc 8 bit trước khi được xử lý. Vì có một số lượng lớn các thanh ghi trong 8051 ta sẽ tập trung vào một số thanh ghi công dụng chung đặc biệt trong các chương kế tiếp. Hãy tham khảo phụ lục Appendix A.3 để biết đầy đủ về các thanh ghi của 8051.

Hình 2.1: a) Một số thanh ghi 8 bit của 8051
b) Một số thanh ghi 16 bit của 8051



Các thanh ghi được sử dụng rộng rãi nhất của 8051 là A (thanh ghi tích luỹ), B, R0 - R7, DPTR (con trỏ dữ liệu) và PC (bộ đếm chương trình). Tất cả các dữ liệu trên đều là thanh g hi 8 bit trừ DPTR và PC là 16 bit. Thanh ghi tích luỹ A được sử dụng cho tất cả mọi phép toán số học và lô-gíc. Để hiểu sử dụng các thanh ghi này ta sẽ giới thiệu chúng trong các ví dụ với các lệnh đơn giản là ADD và MOV.

2.1.2 Lệnh chuyển MOV.

Nói một cách đơn giản, lệnh MOV sao chép dữ liệu từ một vị trí này đến một ví trí khác. Nó có cú pháp như sau:

```
MOV             ; Đích, nguồn; sao chép nguồn vào đích
```

Lệnh này nói CPU chuyển (trong thực tế là sao chép) toán hạng nguồn vào toán hạng đích. Ví dụ lệnh "MOV A, R0" sao chép nội dung thanh ghi R0 vào thanh ghi A. Sau khi lênh này được thực hiện thì thanh ghi A sẽ có giá trị giống như thanh ghi R0. Lệnh MOV không tác động toán hạng nguồn. Đoạn chương trình dưới đây đầu tiên là nạp thanh ghi A tới giá trị 55H 9là giá trị 55 ở dạng số Hex) và sau đó chuyển giá trị này qua các thanh ghi khác nhau bên trong CPU. Lưu ý rằng dấu "#" trong lệnh báo rằng đó là một giá trị. Tầm quan trọng của nó sẽ được trình bày ngay sau ví dụ này.

```
MOV   A, #55H;      ; Nạp trí trị 55H vào thanh ghi A (A = 55H)
MOV   R0, A         ; Sao chép nội dung A vào R0 (bây giờ R0=A)
MOV   R1, A         ; Sao chép nội dung A và R1 (bây giờ R1=R0=A)
MOV   R2, A         ; Sao chép nội dung A và R2 (bây giờ R2=R1=R0=A)
MOV   R3, #95H      ; Nạp giá trị 95H vào thanh ghi R3 (R3 = 95H)
MOV   A, R3         ; Sáo chép nội dung R3 vào A (bây giờ A = 95H)
```

Khi lập trình bộ vi điều khiển 8051 cần lưu ý các điểm sau:

1. Các giá trị có thể được nạp vào trực tiếp bất kỳ thanh ghi nào A, B, R0 - R7. Tuy nhiên, để thông báo đó là giá trị tức thời thì phải đặt trước nó một ký hiệu "#" như chỉ ra dưới đây.

```
MOV   A, #23H       ; Nạp giá trị 23H vào A (A = 23H)
MOV   R0, #12H      ; Nạp giá trị 12H vào R0 (R0 = 2BH)
MOV   R1, #1FH      ; Nạp giá trị 1FH vào R1 (R1 = 1FH)
MOV   R2, #2BH      ; Nạp giá trị 2BH vào R2 (R2 = 2BH)
MOV   B, # 3CH      ; Nạp giá trị 3CH vào B (B = 3CH)
MOV   R7, #9DH      ; Nạp giá trị 9DH vào R7 (R7 = 9DH)
MOV   R5, #0F9H     ; Nạp giá trị F9H vào R5 (R5 = F9H)
MOV   R6, #12       ;Nạp giá trị thập phân 12 = 0CH vào R6
                    (trong R6 có giá trị 0CH).
```

Để ý trong lệnh "MOV R5, #0F9H" thì phải có số 0 đứng trước F và sau dấu # báo rằng F là một số Hex chứ không phải là một ký tự. Hay nói cách khác "MOV R5, #F9H" sẽ gây ra lỗi.

2. Nếu các giá trị 0 đến F được chuyển vào một thanh ghi 8 bit thì các bit còn lại được coi là tất cả các số 0. Ví dụ, trong lệnh "MOV A,#5" kết quả là A=0.5, đó là A = 0000 0101 ở dạng nhị phân.
3. Việc chuyển một giá trị lớn hơn khả năng chứa của thanh ghi sẽ gây ra lỗi ví dụ:

```
MOV   A, #7F2H      ; Không hợp lệ vì 7F2H > FFH
MOV   R2, 456       ; Không hợp lệ vì 456 > 255 (FFH)
```

4. Để nạp một giá trị vào một thanh ghi thì phải gán dấu "#" trước giá trị đó. Nếu không có dấu thì nó hiểu rằng nạp từ một vị trí nhớ. Ví dụ "MOV A, 17H" có nghĩa là nạp giá trị trong ngăn nhớ có giá trị 17H vào thanh ghi A và tại địa chỉ đó dữ liệu có thể có bất kỳ giá trị nào từ 0 đến FFH. Còn để nạp giá trị là 17H vào thanh ghi A thì cần phải có dấu "#" trước 17H như thế này. "MOV A, #17H". Cần lưu ý rằng nếu thiếu dấu "#" trước một thì sẽ không gây lỗi vì hợp ngữ cho đó là một lệnh hợp

lệ. Tuy nhiên, kết quả sẽ không đúng như ý muốn của người lập trình. Đ ây sẽ là một lỗi thường hay gặp đối với lập trình viên mới.

2.1.3 Lệnh cộng ADD.

Lệnh cộng ADD có các phép như sau:

```
ADD a, nguồn          ; Cộng toán hạng nguồn vào thanh ghi A.
```

Lệnh cộng ADD nói CPU cộng byte nguồn vào thanh ghi A và đặt kết quả thanh ghi A. Để cộng hai số như 25H và 34H thì mỗi số có thể chuyển đến một thanh ghi và sau đó cộng lại với nhau như:

```
MOV   A, #25H       ; Nạp giá trị 25H vào A
MOV   R2, #34H      ; Nạp giá trị 34H vào R2
ADD   A, R2         ; Cộng R2 vào A và kết quả A = A + R2
```

Thực hiện chương trình trên ta được A = 59H (vì 25H + 34H = 59H) và R2 = 34H, chú ý là nội dụng R2 không thay đổi. Chương trình trên có thể viết theo nhiều cách phụ thuộc vào thanh ghi được sử dụng. Một trong cách viết khác có thể là:

```
MOV   R5, #25H      ; Nạp giá trị 25H vào thanh ghi R5
MOV   R7, #34H      ; Nạp giá trị 34H vào thanh ghi R7
MOV   A, #0         ; Xoá thanh ghi A (A = 0)
ADD   A, R5         ; Cộng nội dung R5 vào A (A = A + R5)
ADD   A, R7         ; Cộng nội dung R7 vào A (A = A + R7 = 25H + 34H)
```

Chương trình trên có kết quả trong A Là 59H, có rất nhiều cách để viết chương trình giống như vậy. Một câu hỏi có thể đặt ra sau khi xem đoạn chương trình trên là liệu có cần chuyển cả hai dữ liệu vào các thanh ghi trước khi cộng chúng với nhau không? Câu trả lời là không cần. Hãy xem đoạn chương trình dưới đây:

```
MOV   A, #25H       ; Nạp giá trị thứ nhất vào thanh ghi A (A = 25H)
ADD   A, #34H       ; Cộng giá trị thứ hai là 34H vào A (A = 59H)
```

Trong trường hợp trên đây, khi thanh ghi A đã chứa số thứ nhất thì giá trị thứ hai đi theo một toán hạng. Đây được gọi là toán hạng tức thời (trực tiếp).

Các ví dụ trước cho đến giờ thì lệnh ADD báo rằng toán hạng nguồn có thể hoặc là một thanh ghi hoặc là một dữ liệu trực tiếp (tức thời) nhưng thanh ghi đích luôn là thanh ghi A, thanh ghi tích luỹ. Hay nói cách khác là một lệnh như "ADD R2, #12H" là lệnh không hợp lệ vì mọi phép toán số học phải cần đến thanh ghi A và lệnh "ADD R4, A" cũng không hợp lệ vì A luôn là thanh ghi đích cho mọi phép số học. Nói một cách đơn giản là trong 8051 thì mọi phép toán số học đều cần đến thanh A với vai trò là toán hạng đích. Phần trình bày trên đây giải thích lý do vì sao thanh ghi A như là thanh thi tích luỹ. Cú pháp các lệnh hợp ngữ mô tả cách sử dụng chúng và liệt kê các kiểu toán hạng hợp lệ được cho trong phụ lục Appendix A.1.

Có hai thanh ghi 16 bit trong 8051 là bộ đếm chương trình PC và con trỏ dữ liệu APTR. Tầm quan trọng và cách sử dụng chúng được trình bày ở mục 2.3. Thanh ghi DPTR được sử dụng để truy cập dữ liệu và được làm kỹ ở chương 5 khi nói về các chế độ đánh địa chỉ.

2.2 Giới thiệu về lập trình hợp ngữ 8051.

Trong phần này chúng ta bàn về dạng thức của hợp ngữ và định nghĩa một số thuật ngữ sử dụng rộng rãi gắn liền với lập trình hợp ngữ.

CPU chỉ có thể làm việc với các số nhị phân và có thể chạy với tốc độ rất cao. Tuy nhiên, thật là ngán ngậm và chậm chạp đối với con người phải làm việc với các số 0 và 1 để lập trình cho máy tính. Một chương trình chứa các số 0 và 1 được gọi là ngôn ngữ máy.

Trong những ngày đầu của máy tính, các lập trình viên phải viết mã chương trình dưới dạng ngôn ngữ máy. Mặc dụ hệ thống thập lục phân (số Hex) đã được sử dụng như một cách hiệu quả hơn để biểu diễn các số nhị phân thì quá trình làm việc với mã máy vẫn còn là công việc cồng kềnh đối với con người. Cuối cùng, các nguồn ngữ hợp ngữ đã được phát, đã cung cấp các từ gợi nhớ cho các lệnh mã máy cộng với những đặc tính khác giúp cho việc lập trình nhanh hơn và ít mắc lỗi hơn. Thuật ngữ từ gợi nhớ (mnemonic) thường xuyên sử dụng trong tài liệu khoa học và kỹ thuật máy tính để tham chiếu cho các mã và từ rút gọn tương đối dễ nhớ, các chương trình hợp ngữ phải được dịch ra thanh mã máy bằng một chương trình được là trình hợp ngữ (hợp dịch). Hợp ngữ được coi như là một ngông ngữ bậc thấp vì nó giao tiếp trực tiếp với cấu trúc bên trong của CPU. Để lập trình trong hợp ngữ, lập trình viên phải biết tất cả các thanh ghi của CPU và kích thước của chúng cũng như các chi tiết khác.

Ngày nay, ta có thể sử dụng nhiều ngôn ngữ lập trình khác nhau, chẳng hạn như Basic, Pascal, C, C^{++}, Java và vô số ngôn ngữ khác. Các ngôn ngữ này được coi là nhưng ngôn ngữ bậc cao vì lập trình viên không cần phải tương tác với các chi tiết bên trong của CPU. Một trình hợp dịch được dùng để dịch chương trình hợp ngữ ra mã máy còn (còn đôi khi cũng còn được gọi mà đối tượng (Object Code) hay mã lệnh Opcode), còn các ngôn ngữ bậc cao được dịch thành các ngôn ngữ mã máy bằng một chương trình gọi là trình biên dịch. Ví dụ, để viết một chương trình trong C ta phải sử dụng một trình biên dịch C để dịch chương trình về dạng mã máy. Bây giờ ta xét dạng thức hợp ngữ của 8051 và sử dụng trình hợp dịch để tạo ra một chương trình sẵn sàng chạy ngay được.

2.2.1 Cấu trúc của hợp ngữ.

Một chương trình hợp ngữ bao gồm một chuỗi các dòng lệnh hợp ngữ. Một lệnh hợp ngữ có chứa một từ gợi nhớ (mnemonic) và tuy theo từng lệnh và sau nó có một hoặc hai toán hạng. Các toán hạng là các dữ liệu cần được thao tác và các từ gợi nhớ là các lệnh đối với CPU nói nó làm gì với các dữ liệu.

```
        ORG   0H          ; Bắt đầu (origin) tại ngăn nhớ 0
        MOV   R5, #25H    ; Nạp 25H vào R5
        MOV   R7, #34H    ; Nạp 34H vào R7
        MOV   A, #0       ; Nạp 0 vào thanh ghi A
        ADD   A, R5       ; Cộng nôi dụng R5 vào A (A = A + R5)
        ADD   A, R7       ; Cộng nội dung R7 vào A (A = A + R7)
        ADD   A, #121H    ; Cộng giá trị 12H vào A (A = A + 12H)
HERE: SJMP HERE           ; ở lại trong vòng lặp này
        END               ; Kết thúc tệp nguồn hợp ngữ
```

Chương trình 2.1: Ví dụ mẫu về một chương trình hợp ngữ.

Chương trình 2.1 cho trên đây là một chuỗi các câu lệnh hoặc các dòng lệnh được viết hoặc bằng các lệnh hợp ngữ như ADD và MOV hoặc bằng các câu lệnh được gọi là các chỉ dẫn. Trong khi các lệnh hợp ngữ thì nói CPU phải làm gì thì các chỉ lệnh

(hay còn gọi là giả lệnh) thì đưa ra các chỉ lệnh cho hợp ngữ. Ví dụ, trong chương trình 2.1 thì các lệnh ADD và MOV là các lệnh đến CPU, còn ORG và END là các chỉ lệnh đối với hợp ngữ. ORG nói hợp ngữ đặt mã lệnh tại ngăn nhớ 0 và END thì báo cho hợp ngữ biết kết thúc mã nguồn. Hay nói cách khác một chỉ lệnh để bắt đầu và chỉ lệnh thứ hai để kết thúc chương trình.

Cấu trúc của một lệnh hợp ngữ có 4 trường như sau:

[nhãn:] [từ gợi nhớ] [các toán hạng] [; chú giải]

Các trường trong dấu ngoặc vuông là tuỳ chọn và không phải dòng lệnh nào cũng có chúng. Các dấu ngoặc vuông không được viết vào. Với dạng thức trên đây cần lưu ý các điểm sau:

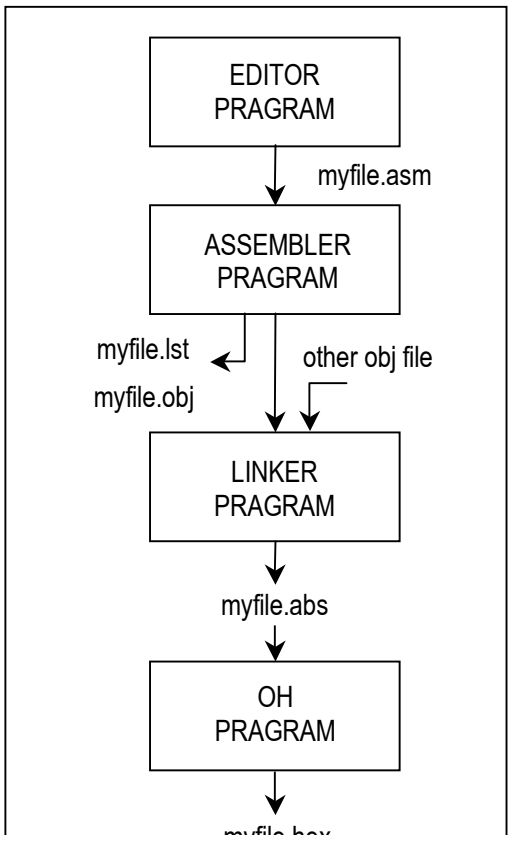
1. Trường nhãn cho phép chương trình tham chiếu đến một dòng lệnh bằng tên. Nó không được viết quá một số ký tự nhất định. Hãy kiểm tra quy định này của hợp ngữ mà ta sử dụng.
2. Từ gợi nhớ (lệnh) và các toán hạng là các trường kết hợp với nhau thực thi công việc thực tế của chương trình và hoàn thiện các nhiệm vụ mà chương trình được viết cho chúng. Trong hợp ngữ các câu lệnh như:

   " ADD A, B"
   "MOV A, #67H"

   thì ADD và MOV là những từ gợi nhớ tạo ra mã lệnh, còn "A, B" và "A, #67H" là những toán hạng thì hai trường có thể chứa các lệnh giả hoặc chỉ lệnh của hợp ngữ. Hãy nhớ rằng các chỉ lệnh không tạo ra mã lệnh nào (mã máy) và chúng chỉ dùng bởi hợp ngữ, ngược lại đối với các lệnh là chúng được dịch ra mã máy (mã lênh) cho CPU thực hiện. Trong chương trình 2.1 các lệnh ORG và END là các chỉ lệnh (một số hợp ngữ của 8051 sử dụng dạng .ORG và .END). Hãy đọc quy định cụ thể của hợp ngữ ta sử dụng.
3. Chương chú giải luôn phải bắt đầu bằng dấu chấm phẩy (;). Các chú giải có thể bắt đầu ở đầu dòng hoặc giữa dòng. Hợp ngữ bỏ qua (làm ngơ) các chú giải nhưng chúng lại rất cần thiết đối với lập trình viên. Mặc dù các chú giải là tuỳ chọn, không bắt buộc nhưng ta nên dùng chúng để mô tả chương trình để giúp cho người khác đọc và hiểu chương trình dễ dàng hơn.
4. Lưu ý đến nhãn HERE trong trường nhãn của chương trình 2.1. Một nhãn bất kỳ tham chiếu đến một lệnh phải có dấu hai chấm (:) đứng ở sau. Trong câu lệnh nhảy ngắn SJMP thì 8051 được ra lệnh ở lại trong vòng lặp này vô hạn. Nếu hệ thống của chúng ta có một chương trình giám sát thì takhông cần dòng lệnh này và nó có thể được xoá đi ra khỏi chương trình.

2.3 Hợp dịch và chạy một chương trình 8051.

Như vậy cấu trúc của một chương trình hợp ngữ ta đã được biết, câu hỏi đặt ra là chương



trình sẽ được tạo ra và hợp dịch như thế nào và làm thế nào để có thể chạy được? Các bước để tạo ra một chương trình hợp ngữ có thể chạy được là:

1. Trước hết ta sử dụng mộ trình soạn thảo để gỡ vào một chương trình giống như chương trình 2.1. Có nhiều trình soạn thảo tuyệt vời hoặc các bộ sử lý từ được sử dụng để tạo ra và/ hoặc để soạn thảo chương trình. Một trình soạn thảo được sử dụng rộng rãi là trình soạn thảo EDIT của MS-DOS (hoặc Noterad của Windows) đều chạy trên hệ điều hành Microsoft. Lưu ý rằng, trình soạn thảo phải có khả năng tạo ra tệp mã ASCII. Đối với nhiều trình hợp ngữ thì các tên tệp tuân theo các quy ước thường lệ củ DOS, nhưng phần mở rộng của các tệp nguồn phải là "asm" hay "src" tuỳ theo trình hợp ngữ mà ta sử dụng.
2. Tệp nguồn có phần mở rộng "asm" chứa mã chương trình được tạo ra ở bước 1 được nạp vào trình hợp dịch của 8051. Trình hợp dịch chuyển các lệnh ra mã máy. Trình hợp dịch sẽ tạo ra một tệp đối tượng và một tệp liệt kê với các thành phần mở rộng "obj" và "lst" tương ứng.
3. Các trình hợp dịch yêu cầu một bước thứ ba gọi là liên kết. Chương trình liên kết lấy một hoặc nhiều tệp đối tượng và tạo ra một tệp đối tượng tuyệt đối với thành phần mở rộng "abs". Tệp "abs" này được sử dụng bởi thùng chứa của 8051 có một chương trình giám sát.
4. Kế sau đó tệp "abs" được nạp vào một chương trình được gọi là "0H" (chuyển đối tượng object về dạng số Hex) để tạo ra một tệp với đuôi mở rộng "Hex" có thể nạp tốt vào trong ROM. Chương trình này có trong tất cả mọi trình hợp ngữ của 8051 các trình hợp ngữ dựa trên Windows hiện nay kết hợp các bước 2 đến 4 vào thành một bước.

   Hình 2.2: Các bước để tạo ra một chương trình.

2.3.1 Nói thêm về các tệp ".asm" và ".object".

Tệp ".asm" cũng được gọi là tệp nguồn và chính vì lý do này mà một số trình hợp ngữ đòi hỏi tệp này phải có một phần mở rộng "src" từ chữ "source" là nguồn. Hãy kiểm tra hợp ngữ 8051 mà ta sử dụng xem nó có đòi hỏi như vậy không? Như ta nói trước đây tệp này được tạo ra nhờ một trình biên tập chẳng hạn như Edit của DOS hoặc Notepad của Windows. Hợp ngữ của 8051 chuyển đổi các tệp hợp ngữ trong tệp .asm thành ngôn ngữ mã máy và cung cấp tệp đối tượng .object. Ngoài việc tạo ra tệp đối tượng trình hợp ngữ cũng cho ra tệp liệt kê "lst" (List file).

2.3.2 Tệp liệt kê ".lst".

Tệp liệt kê là một tuỳ chọn, nó rất hữu ích cho lập trình viên vì nó liệt kê tất cả mọi mã lệnh và địa chỉ cũng như tất cả các lỗi mà trình hợp ngữ phát hiện ra. Nhiều trình hợp ngữ giả thiết rằng, tệp liệt kê là không cần thiết trừ khi ta báo rằng ta muốn tạo ra nó. Tệp này có thể được truy cập bằng một trình biên dịch như Edit của DOS hoặc Notepad của Window và được hiển thị trên màn hình hoặc được gửi ra máy in. Lập trình viên sử dụng tệp liệt kê để tìm các lỗi cú pháp. Chỉ sau khi đã sửa hết các lỗi được đánh dấu trong tệp liệt kê thì tệp đối tượng mới sẵn sàng làm đầu vào cho chương trình liên kết.

```
1 0000              ORG   0H          ; Bắt đầu ở địa chỉ 0
2 0000  7D25        MOV   R5, #25H    ; Nạp giá trị 25H vào R5
3 0002 7F34         MOV   R7, #34H    ; Nạp giá trị 34H vào R7
4 0004  7400        MOV   A, #0       ; Nạp 0 vào A (xoá A)
5 0006 2D           ADD   A, R5       ; Cộng nội dung R5 vào A (A = A + R5)
6 0007 2F           ADD   A, R7       ; Cộng nội dung R7 vào A (A = A + R7)
7 0008 2412         ADD   A, #12H     ; Cộng giá trị 12H vào A (A = A + 12H)
```

```
8 00A  BCEF  HERE: SJMP HERE          ; ở lại vòng lặp này
9 000C             END                ; Kết thúc tệp .asm
```

Chương trình 2.2: Tệp liệt kê.

2.4 Bộ đếm chương trình và không gian ROM trong 8051.

2.4.1 Bộ đếm chương trình trong 8051.

Một thanh ghi quan trọng khác trong 8051 là bộ đếm chương trình . Bộ đếm chương trình chỉ đếm địa chỉ của lệnh kế tiếp cần được thực hiện. Khi CPU nạp mã lệnh từ bộ nhớ ROM chương trình thì bộ đếm chương trình tăng lên chỉ đếm lệnh kết tiếp. Bộ đếm chương trình trong 8051 có thể truy cập các địa chỉ chương trình trong 8051 rộng 16 bit. Điều này có nghĩa là 8051 có thể truy cập các địa chỉa chương trình từ 0000 đến FFFFH tổng cộng là 64k byte mã lệnh. Tuy nhiên, không phải tất cả mọi thành viên của 8051 đều có tất cả 64k byte ROM trên chíp được cài đặt. Vậy khi 8051 được bật nguồn thì nó đánh thức ở địa chỉa nào?

2.4.2 Địa chỉ bắt đầu khi 8051 được cấp nguồn.

Một câu hỏi mà ta phải hỏi về bộ vi điều khiển bất kỳ là thì nó được cấp nguồn thì nó bắt đầu từ địa chỉ nào? Mỗi bộ vi điều khiển đều khác nhau. Trong trường hợp họ 8051 thì mọi thành viên kể từ nhà sản xuất nào hay phiên bản nào thì bộ vi điều khiển đều bắt đầu từ địa chỉ 0000 khi nó được bật nguồn. Bật nguồn ở đây có nghĩa là ta cấp điện áp V_{cc} đến chân RESET như sẽ trình bày ở chương 4. Hay nói cách khác, khi 8051 được cấp nguồn thì bộ đếm chương trình có giá trị 0000. Điều này có nghĩa là nó chờ mã lệnh đầu tiên được lưu ở địa chỉa ROM 0000H. Vì lý do này mà trong vị trí nhớ 0000H của bộ nhở ROM chương trình vì đây là nơi mà nó tìm lệnh đầu tiên khi bật nguồn. Chúng ta đạt được điều này bằng câu lệnh ORG trong chương trình nguồn như đã trình bày trước đây. Dưới đây là hoạt động từng bước của bộ đếm chương trình trong qúa trình nạp và thực thi một chương trình mẫu.

2.4.3 Đặt mã vào ROM chương trình.

Để hiểu tốt hơn vai trò của bộ đếm chương trình trong quá trình nạp và thực thi một chương trình, ta khảo sát một hoạt động của bộ đếm chương trình khi mỗi lệnh được nạp và thực thi. Trước hết ta khảo sát một lần nữa tệp liệt kê của chương trình mẫu và cách đặt mã vào ROM chương trình 8051 như thế nào? Như ta có thể thấy, mã lệnh và toán hạng đối với mỗi lệnh được liệt kê ở bên trái của lệnh liệt kê.

Chương trình 2.1: Ví dụ mẫu về một chương trình hợp ngữ.

Chương trình 2.1 cho trên đây là một chuỗi các câu lệnh hoặc các dòng lệnh được viết hoặc bằng các lệnh hợp ngữ như ADD và MOV hoặc bằng các câu lệnh được gọi là các chỉ dẫn. Trong khi các lệnh hợp ngữ thì nói CPU phải làm gì thì các chỉ lệnh (hay còn gọi là giả lệnh) thì đưa ra các chỉ lệnh cho hợp ngữ. Ví dụ, trong chương trình 2.1 thì các lệnh ADD và MOV là các lệnh đến CPU, còn ORG và END là các chỉ lệnh đối với hợp ngữ. ORG nói hợp ngữ đặt mã lệnh tại ngăn nhớ 0 và END thì báo cho hợp ngữ biết kết thúc mã nguồn. Hay nói cách khác một chỉ lệnh để bắt đầu và chỉ lệnh thứ hai để kết thúc chương trình.

Cấu trúc của một lệnh hợp ngữ có 4 trường như sau:

[nhãn:] [từ gợi nhớ] [các toán hạng] [; chú giải]

Các trường trong dấu ngoặc vuông là tuỳ chọn và không phải dòng lệnh nào cũng có chúng. Các dấu ngoặc vuông không được viết vào. Với dạng thức trên đây cần lưu ý các điểm sau:

Trường nhãn cho phép chương trình tham chiếu đến một dòng lệnh bằng tên. Nó không được viết quá một số ký tự nhất định. Hãy kiểm tra quy định này của hợp ngữ mà ta sử dụng.
Từ gợi nhớ (lệnh) và các toán hạng là các trường kết hợp với nhau thực thi công việc thực tế của chương trình và hoàn thiện các nhiệm vụ mà chương trình được viết cho chúng. Trong hợp ngữ các câu lệnh như:

" ADD A, B"
"MOV A, #67H"

Thì ADD và MOV là những từ gợi nhớ tạo ra mã lệnh, còn "A, B" và "A, #67H" là những toán hạng thì hai trường có thể chứa các lệnh giả hoặc chỉ lệnh của hợp ngữ. Hãy nhớ rằng các chỉ lệnh không tạo ra mã lệnh nào (mã máy) và chúng chỉ dùng bởi hợp ngữ, ngược lại đối với các lệnh là chúng được dịch ra mã máy (mã lênh) cho CPU thực hiện. Trong chương trình 2.1 các lệnh ORG và END là các chỉ lệnh (một số hợp ngữ của 8051 sử dụng dạng .ORG và .END). Hãy đọc quy định cụ thể của hợp ngữ ta sử dụng.

Trương chú giải luôn phải bắt đầu bằng dấu chấm phẩy (;). Các chú giải có thể bắt đầu ở đầu dòng hoặc giữa dòng. Hợp ngữ bỏ qua (làm ngơ) các chú giải nhưng chúng lại rất cần thiết đối với lập trình viên. Mặc dù các chú giải là tuỳ chọn, không bắt buộc nhưng ta nên dùng chúng để mô tả chương trình để giúp cho người khác đọc và hiểu chương trình dễ dàng hơn.

Lưu ý đến nhãn HERE trong trường nhãn của chương trình 2.1. Một nhãn bất kỳ tham chiếu đến một lệnh phải có dấu hai chấm (:) đứng ở sau. Trong câu lệnh nhảy ngắn SJMP thì 8051 được ra lệnh ở lại trong vòng lặp này vô hạn. Nếu hệ thống của chúng ta có một chương trình giám sát thì takhông cần dòng lệnh này và nó có thể được xoá đi ra khỏi chương trình.

Địa chỉ	Mã lệnh
0000	7D
0001	25
0002	F7
0003	34
0004	74
0005	00
0006	2D
0007	2F
0008	24
0009	12
000A	80
000B	FE

Chương trình 2.1: Tệp liệt kê

Sau khi chương trình được đốt vào trong ROM của thành viên họ 8051 như 8751 hoặc AT 8951 hoặc DS 5000 thì mã lệnh và toán hạng được đưa vào các vị trí nhớ ROM bắt đầu từ địa chỉ 0000 như bảng liệt kê dưới đây.

Địa chỉ ROM	Ngôn ngữ máy	Hợp ngữ
0000	7D25	MOV R5, #25H
0002	7F34	MOV R7, #34H
0004	7400	MOV A, #0
0006	2D	ADD A, R5
0007	2F	ADD A, R7
0008	2412	ADD A, #12H
000A	80EF	HERE: SJMP HERE

Bảng nội dung ROM của chương trình 2.1.

Bảng liệt kê chỉ ra địa chỉ 0000 chứa mã 7D là mã lệnh để chuyển một giá trị vào thanh ghi R5 và địa chỉ 0001 chứa toán hạng (ở đây là giá trị 254) cần được chuyển vào R5. Do vậy, lệnh "MOV R5, #25H" có mã là "7D25" trong đó 7D là mã lệnh,

còng 25 là toán hạng. Tương tự như vậy, mã máy "7F34" được đặt trong các ngăn nhớ 0002 và 0003 và biểu diễn mã lệnh và toán hạng đối với lệnh "MOV R7, #34H". Theo cách như vậy, mã máy "7400" được đặt tại địa chỉ 0004 và 0005 và biểu diễn mã lệnh và toán hạng đối với lệnh "MOV A, #0". Ngăn nhớ 0006 có mã 2D là mã đối với lệnh "ADD A, R5" và ngăn nhớ 0007 có nội dung 2F là mã lệnh cho "ADD A, R7". Mã lệnh đối với lệnh "ADD A, #12H" được đặt ở ngăn nhớ 0008 và toán hạng 12H được đặt ở ngăn nhớ 0009. Ngăn nhớ 000A có mã lệnh của lệnh SJMP và địa chỉ đích của nó được đặt ở ngăn nhớ 000B. Lý do vì sao địa chỉ đích là FE được giải thích ở chương 3.

2.4.4 Thực hiện một chương trình theo từng byte.

Giả sử rằng chương trình trên được đốt vào ROM của chíp 8051 hoặc(8751, AT 8951 hoặc DS 5000) thì dưới đây là mô tả hoạt động theo từng bước của 8051 khi nó được cấp nguồn.

1. Khi 8051 được bật nguồn, bộ đếm chương trình PC có nội dung 0000 và bắt đầu nạp mã lệnh đầu tiên từ vị trí nhớ 0000 của ROM chương trình. Trong trường hợp của chương trình này là mã 7D để chuyển một toán hạng vào R5. Khi thực hiện mã lệnh CPU nạp giá trị 25 vào bộ đếm chương trình được tăng lên để chỉ đến 0002 (PC = 0002) có chứa mã lệnh 7F là mã của lệnh chuyển một toán hạng vào R7 "MOV R7, ...".
2. Khi thực hiện mã lệnh 7F thì giá trị 34H được chuyển vào R7 sau đó PC được tăng lên 0004.
3. Ngăn nhớ 0004 chứa mã lệnh của lệnh "MOV A, #0". Lệnh này được thực hiện và bây giờ PC = 0006. Lưu ý rằng tất cả các lệnh trên đều là những lệnh 2 byte, nghĩa là mỗi lệnh chiếm hai ngăn nhớ.
4. Bây giờ PC = 0006 chỉ đến lệnh kế tiếp là "ADD A, R5". Đây là lệnh một byte, sau khi thực hiện lệnh này PC = 0007.
5. Ngăn nhớ 0007 chứa mã 2F là mã lệnh của "ADD A, R7". Đây cũng là lệnh một byte, khi thực hiện lệnh này PC được tăng lên 0008. Qúa trình này cứ tiếp tục cho đến khi tất cả moi lệnh đều được nạp và thực hiện. Thực tế mà bộ đếm chương trình chỉ đến lệnh kế tiếp cần được thực hiện giải thích tại sao một số bộ vi xử lý (đáng nói là × 86) gọi bộ đếm là con trỏ lệnh (Instruction Pointer).

2.4.5 Bản đồ nhớ ROM trong họ 8051.

Như ta đã thấy ở chương trước, một số thành viên họ 8051 chỉ có 4k byte bộ nhớ ROM trên chíp (ví dụ 8751, AT 8951) và một số khác như AT 8951 có 8k byte ROM, DS 5000-32 của Dallas Semiconductor có 32k byte ROM trên chíp. Dallas Semiconductor cũng có motọ 8051 với ROM trên chíp là 64k byte. Điểm cần nhớ là không có thành viên nào của họ 8051 có thể truy cập được hơn 64k byte mã lệnh vì bộ đếm chương trình của 8051 là 16 bit (dải địa chỉ từ 0000 đến FFFFH). Cần phải ghi nhớ là lệnh đầu tiên của ROM chương trình đều đặt ở 0000, còn lệnh cuối cùng phụ thuộc vào dung lượng ROM trên chíp của mỗi thành viên họ 8051. Trong số các thành viên họ 8051 thì 8751 và AT 8951 có 4k byte ROM trên chíp. Bộ nhớ ROM trên chíp này có các địa chỉ từ 0000 đến 0FFFH. Do vậy, ngăn nhớ đầu tiên có địa chỉ 0000 và ngăn nhớ cuối cùng có địa chỉa 0FFFH. Hãy xét ví dụ 2.1.

VÝ dô 2.1:

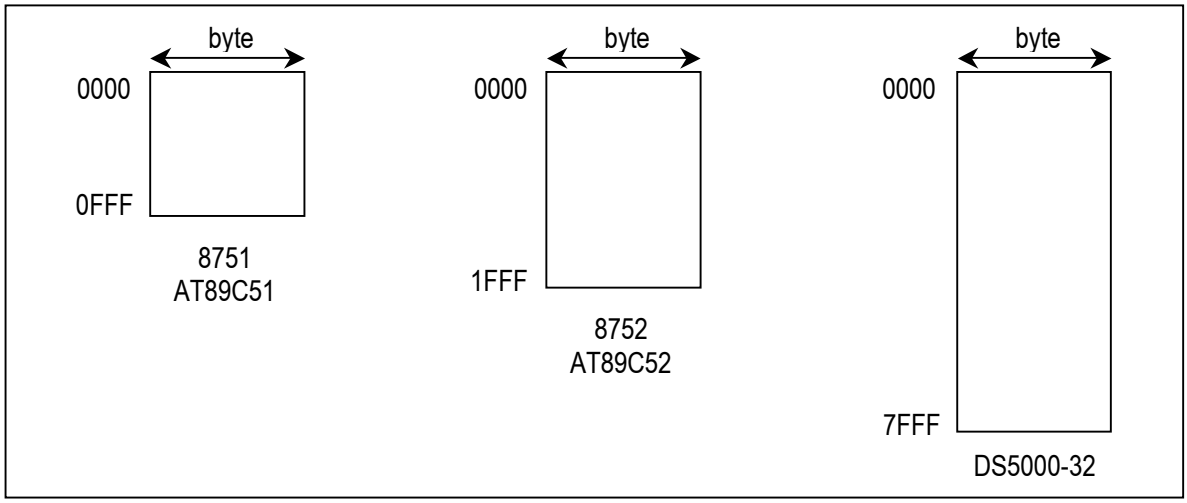
Tìm địa chỉ bộ nhớ ROM của mỗi thành viên họ 8051 sau đây.

a) AT 8951 (hoặc 8751) với 4k byte
b) DS 5000-32 với 32k byte

Lêi gi¶i:

a) Với 4k byte của không gian nhớ ROM trên chíp ta có 4096 byte bằng 1000H ở dạng Hex (4 × 1024 = 4096 hay 1000 ở dạng Hex). Bộ nhớ này được xắp xếp trong các ngăn nhớ từ 0000 đến 0FFFH. Lưu ý 0 luôn là ngăn nhớ đầu tiên.

b) Với 32k byte nhớ ta có 32.768 byte (32 × 1024). Chuyển đổi 32.768 về số Hex ta nhận được giá trị 8000H. Do vậy, không gian nhớ là dải từ 0000 đến 7FFFH.



Hình 2.3: Dải địa chỉ của ROM trên chíp một số thành viên họ 8051.

2.5 Các kiểu dữ liệu và các chỉ lệnh.

2.5.1 Kiểu dữ liệu và các chỉ lệnh của 8051.

Bộ vi điều khiển chỉ có một kiểu dữ liệu, nó là 8 bit và độ dài mỗi thanh ghi cũng là 8 bit. Công việc của lập trình viên là phân chia dữ liệu lớn hơn 8 bit ra thành từng khúc 8 bit (từ 00 đến FFH hay từ 0 đến 255) để CPU xử lý. Ví dụ về xử lý dữ liệu lớn hơn 8 bit được trình bày ở chương 6. Các dữ liệu được sử dụng bởi 8051 có thể là số âm hoặc số dương và về xử lý các số có dấu được bàn ở chương 6.

2.5.2 Chỉ lệnh DB (định nghĩa byte).

Chỉ lệnh DB là một chỉ lệnh dữ liệu được sử dụng rộng rãi nhất trong hợp ngữ. Nó được dùng để định nghĩa dữ liệu 8 bit. Khi DB được dùng để định nghĩa byte dữ liệu thì các số có thể ở dạng thập phân, nhị phân, Hex hoặc ở dạng thức ASII. Đối với dữ liệu thập phân thì cần đặt chữ "D" sau số thập phân, đối với số nhị phân thì đặt chữ "B" và đối với dữ liệu dạng Hex thì cần đặt chữ "H". Bất kể ta sử dụng số ở dạng thức nào thì hợp ngữ đều chuyển đổi chúng về thành dạng Hex. Để báo dạng thức ở dạng mã ASCII thì chỉ cần đơn giản đặt nó vào dấu nháy đơn 'như thế này'. Hợp ngữ sẽ gán mã ASCII cho các số hoặc các ký tự một cách tự động. Chỉ lệnh DB chỉ là chỉ lệnh mà có thể được sử dụng để định nghĩa các chuỗi ASCII lớn hơn 2 ký tự. Do vậy, nó có thể được sử dụng cho tất cả mọi định nghĩa dữ liệu ASCII. Dưới đây là một số ví dụ về DB:

```
        ORG    500H
DATA1:  DB     2B                  ; Số thập phân (1C ở dạng Hex)
DATA2:  DB     00110101B           ; Số nhị phân (35 ở dạng Hex)
DATA3:  DB     39H                 ; Số dạng Hex
               ORG    510H
DATA4:  DB     "2591"              ; Các số ASCII
               ORG    518H
DATA5   DB     "My name is Joe"    ; Các ký tự ASCII
```

Các chuỗi ASCII có thể sử dụng dấu nháy đơn 'như thế này' hoặc nháy kép "như thế này". Dùng dấu phẩy kép sẽ hữu ích hơn đối với trường hợp dấu nháy đơn được dùng sở hữu cách như thế này "Nhà O' Leary". Chỉ lệnh DB cũng được dùng để cấp phát bộ nhớ theo từng đoạn kích thước một byte.

2.5.3 Các chỉ lệnh của hợp ngữ.

1. Chỉ lệnh ORG: Chỉ lệnh ORG được dùng để báo bắt đầu của địa chỉ. Số đi sau ORG có kể ở dạng Hex hoặc thập phân. Nếu số này có kèm chữ H đằng sau thì là ở dạng Hex và nếu không có chữ H ở sau là số thập phân và hợp ngữ sẽ chuyển nó thành số Hex. Một số hợp ngữ sử dụng dấu chấm đứng trước "ORG" thay cho "ORG". Hãy đọc kỹ về trình hợp ngữ ta sử dụng.
2. Chỉ lệnh EQU: Được dùng để định nghĩa một hằng số mà không chiếm ngăn nhớ nào. Chỉ lệnh EQU không dành chỗ cất cho dữ liệu nhưng nó gắn một giá trị hằng số với nhãn dữ liệu sao cho khi nhãn xuất hiện trong chương trình giá trị hằng số của nó sẽ được thay thế đối với nhãn. Dưới đây sử dụng EQU cho hằng số bộ đếm và sau đó hằng số được dùng để nạp thanh ghi RS.

```
COUNT  EQU 25
MOV    R3, #count
```

   Khi thực hiện lện "MOV R3, #COUNT" thì thanh ghi R3 sẽ được nạp giá trị 25 (chú ý đến dấu #). Vậy ưu điểm của việc sử dụng EQU là gì? Giả sử có một hằng số (một giá trị cố định) được dùng trong nhiều chỗ khác nhau trong chương trình và lập trình viên muốn thay đổi giá trị của nó trong cả chương trình. Bằng việc sử dụng chỉ lệnh EQU ta có thể thay đổi một số lần và hợp ngữ sẽ thay đổi tất cả mọi lần xuất hiện của nó là tìm toàn bộ chương trình và gắng tìm mọi lần xuất hiện.
3. Chỉ lệnh END: Một lệnh quan trọng khác là chỉ lệnh END. Nó báo cho trình hợp ngữ kết thúc của tệp nguồn "asm" chỉ lệnh END là dòng cuối cùng của chương trình 8051 có nghĩa là trong mã nguồn thì mọi thứ sau chỉ lệnh END để bị trình hợp ngữ bỏ qua. Một số trình hợp ngữ sử dụng .END có dấu chấm đứng trước thay cho END.

2.5.4 Các quy định đố với nhãn trong hợp ngữ.

Bằng cách chọn các tên nhãn có nghĩa là một lập trình viên có thể làm cho chương trình dễ đọc và dễ bảo trì hơn, có một số quy định mà các tên nhãn phải tuân theo. Thứ nhất là mỗi tên nhãn phải thống nhất, các tên được sử dụng làm nhãn trong hợp ngữ gồm các chữ cái viết hoa và viết thường, các số từ 0 đến 9 và các dấu đặc biệt như: dấu hỏi (?), dấu (@), dấu gạch dưới (_), dấu đô là ($) và dấu chu kỳ (.). Ký tự đầu tiên của nhãn phải là một chữ cái. Hay nói cách khác là nó không thể là số Hex. Mỗi trình hợp ngữ có một số từ dự trữ là các từ gợi nhớ cho các lệnh mà không được dùng để làm nhãn trong chương trình. Ví dụ như "MOV" và "ADD". Bên cạnh các từ gợi nhớ còn có một số tự dự trữ khác, hãy kiểm tra bản liệt kê các từ dự phòng của hợp ngữ ta đang sử dụng.

2.6 Các bit cờ và thanh ghi đặc bệt PSW của 8051.

Cũng như các bộ vi xử lý khác, 8051 có một thanh ghi cờ để báo các điều kiện số học như bit nhớ. Thanh ghi cờ trong 8051 được gọi là thanh ghi từ trạng thái chương trình PSW. Trong phần này và đưa ra một số ví dụ về cách thay đổi chúng.

2.6.1 Thanh ghi từ trạng thái chương trình PSW.

Thanh ghi PSW là thanh ghi 8 bit. Nó cũng còn được coi như là thanh ghi cờ. Mặc dù thanh ghi PSW rộng 8 bit nhưng chỉ có 6 bit được 8051 sử dụng. Hai bit chưa dùng là các cờ ch người dùng định nghĩa. Bốn trong số các cờ được gọi là các cờ có điều kiện, có nghĩa là chúng báo một số điều kiện do kết quả của một lệnh vừa được thực hiện. Bốn cờ này là cờ nhớ CY (carry), cờ AC (auxiliary cary), cờ chẵn lẻ P (parity) và cờ tràn OV (overflow).

Như nhìn thấy từ hình 2.4 thì các bit PSW.3 và PSW.4 được gán như RS0 và RS1 và chúng được sử dụng để thay đổi các thanh ghi băng. Chúng sẽ được giải thích ở phần kế sau. Các bit PSW.5 và PSW.1 là các bit cờ trạng thái công dụng chung và lập trình viên có thể sử dụng cho bất kỳ mục đích nào.

CY	AC	F0	RS1	RS0	OV	–	P

CY PSW.7 ; Cờ nhớ
AC PSW.6 ; Cờ

- PSW.5 ; Dành cho người dùng sử dụng mục đích chung

RS1 PSW.4 ; Bit = 1 chọn băng thanh ghi
RS0 PSW.3 ; Bit = 0 chọn băng thanh ghi
OV PSW.2 ; Cờ bận

- PSW.1 ; Bit dành cho người dùng định nghĩa

P PSW.0 ; Cờ chẵn, lẻ. Thiết lập/ xoá bằng phần cứng mỗi chu kỳ lệnh báo tổng các số bit 1 trong thanh ghi A là chẵn/ lẻ.

RS1	RS0	Băng thanh ghi	Địa chỉ
0	0	0	00H - 07H
0	1	1	08H - 0FH
1	0	2	10H - 17H
1	1	3	18H - 1FH

H×nh 2.4: **C¸c bit cña thanh ghi PSW**

Dưới đây là giải thích ngắn gọn về 4 bit cờ của thanh ghi PSW.

1. Cờ nhớ CY: Cờ này được thiết lập mỗi khi có nhớ từ bit D7. Cờ này được tác động sau lệnh cộng hoặc trừ 8 bit. Nó cũng được thiết lập lên 1 hoặc xoá về 0 trực tiếp bằng lệnh "SETB C" và "CLR C" nghĩa là "thiết lập cờ nhớ" và "xoá cờ nhớ" tương ứng. Về các lệnh đánh địa chỉ theo bit được bàn kỹ ở chương 8.
2. Cờ AC: Cờ này báo có nhớ từ bit D3 sang D4 trong phép cộng ADD hoặc trừ SUB. Cờ này được dùng bởi các lệnh thực thi phép số học mã BCD (xem ở chương 6).
3. Cờ chẵn lẻ P: Cờ chẵn lẻ chỉ phản ánh số bit một trong thanh ghi A là chẵn hay lẻ. Nếu thanh ghi A chứa một số chẵn các bit một thì P = 0. Do vậy, P = 1 nếu A có một số lẻ các bit một.
4. Cờ chàn OV: Cờ này được thiết lập mỗi khi kết quả của một phép tính số có dấu quá lớn tạo ra bit bậc cao làm tràn bit dấu. Nhìn chung cờ nhớ được dùng để phát hiện lỗi trong các phép số học không dấu. Còn cờ tràn được dùng chỉ để phát hiện lỗi trong các phép số học có dấu và được bàn kỹ ở chương 6.

2.6.2 Lệnh ADD và PSW.

Bây giờ ta xét tác động của lệnh ADD lên các bit CY, AC và P của thanh ghi PSW. Một số ví dụ sẽ làm rõ trạng thái của chúng, mặc dù các bit cờ bị tác động bởi lệnh ADD là CY, P, AC và OV nhưng ta chỉ tập trung vào các cờ CY, AC và P, còn cờ OV sẽ được nói đến ở chương 6 vì nó liên quan đến phép tính số học số có dấu.

Các ví dụ 2.2 đến 2.4 sẽ phản ánh tác động của lệnh ADD lên các bit nói trên.

Bảng 2.1: Các lệnh tác động lên các bit cờ.

Instruction	CY	OV	AC
ADD	X	X	X
ADDC	X	X	X
SUBB	X	X	X
MUL	0	X	
DIV	0	X	
DA	X		
RRC	X		
RLC	X		
SETB C	1		
CLR C	0		
CPL C	X		
ANL C, bit	X		
ANL C,/ bit	X		
ORL C, bit	X		
ORL C,/bit	X		
MOV C, bit	X		
CJNE	X		

Ví dụ 2.2: Hãy trình bày trạng thái các bit cờ CY, AC và P sau lệnh cộng 38H với 2FH dưới đây:

```
MOV     A, #38H
ADD  A, #2FH              ; Sau khi cộng A = 67H, CY = 0
```

Lêi gi¶i:

```
     38        00111000
+    2F        00101111
-------------------------
     67        01100111
```

Cờ CY = 0 vì không có nhớ từ D7
Cờ AC = 1 vì có nhớ từ D3 sang D4
Cờ P = 1 vì thanh ghi A có 5 bit 1 (lẻ)

VÝ dô 2.3:

Hãy trình bày trạng thái các cờ CY, AC và P sau phép cộng 9CH với 64H.

Lêi gi¶i:

```
     9C        10011100
+    64        01100100
-------------------------
    100        00000000
```

Cờ CY = 1 vì có nhớ qua bit D7
Cờ AC = 1 vì có nhớ từ D3 sang D4
Cờ P = 0 vì thanh ghi A không có bit 1 nào (chẵn)

VÝ dô 2.4:

Hãy trình bày trạng thái các cờ CY, AC và P sau phép cộng 88H với 93H.

Lêi gi¶i:

```
     88        10001000
+    93        10010011
-------------------------
    11B        00011011
```

Cờ CY = 1 vì có nhớ từ bit D7
Cờ AC = 0 vì không có nhớ từ D3 sang D4
Cờ P = 0 vì số bit 1 trong A là 4 (chẵn)

2.7 Các băng thanh ghi và ngăn xếp của 8051.

Bộ vi điều khiển 8051 có tất cả 128 byte RAM. Trong mục này ta bàn vệ phân bố của 128 byte RAM này và khảo sát công dụng của chúng như các thanh ghi và ngăn xếp.

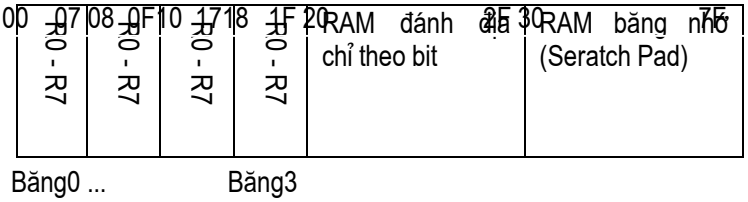
2.7.1 Phân bố không gian bộ nhớ RAM trong 8051.

Có 128 byte RAM trong 8051 (một số thành viên đang chú ý là 8052 có 256 byte RAM). 128 byte RAM bên trong 8051 được gán địa chỉ từ 00 đến 7FH. Như ta sẽ thấy ở chương 5, chúng có thể được truy cập trực tiếp như các ngăn nhớ 128 byte RAM này được phân chia thành từng nhóm như sau:

1. Tổng cộng 32 byte từ ngăn nhớ 00 đến 1FH được dành cho các thanh ghi và ngăn xếp.
2. Tổng cộng 16 byte từ ngăn nhớ 20H đến 2FH được dành cho bộ nhớ đọc/ ghi đánh địa chỉ được theo bit. Chương 8 sẽ bàn chi tiết về bộ nhớ và các lệnh đánh địa chỉ được theo bit.
3. Tổng cộng 80 byte từ ngăn nhớ 30H đến 7FH được dùng cho lưu đọc và ghi hay như vẫn thường gọi là bảng nháp (Serach pad). Những ngăn nhớ này (80 byte) của RAM được sử dụng rộng rãi cho mục đích lưu dữ liệu và tham số bởi các lập trình viên 8051. Chúng ta sẽ sử dụng chúng ở các chương sau để lưu dữ liệu nhận vào CPU qua các cổng vào-ra.

2.7.2 Các băng thanh ghi trong 8051.

Như đã nói ở trước, tổng cộng 32 byte RAM được dành riêng cho các băng thanh ghi và ngăn xếp. 32 byte này được chia ra thành 4 băng các thanh ghi trong đó mỗi băng có 8 thanh ghi từ R0 đến R7. Các ngăn nhớ RAM số 0, R1 là ngăn nhớ RAM số 1, R2 là ngăn nhớ RAM số 2 v.v... Băng thứ hai của các thanh ghi R0 đến R7 bắt đầu từ thanh nhớ RAM số 2 cho đến ngăn nhớ RAM số 0FH. Băng thứ ba bắt đầu từ ngăn nhớ 10H đến 17H và cuối cùng từ ngăn nhớ 18H đến 1FH là dùng cho băng các thanh ghi R0 đến R7 thứ tư.



Hình 2.5: Ngăn xếp các thanh nhớ RAM trong 8051.

Bank 0		Bank 1		Bank 2		Bank 3	
7	R7	F	R7	17	R7	1F	R7
6	R6	E	R6	16	R6	1E	R6
5	R5	D	R5	15	R5	1D	R5
4	R4	C	R4	14	R4	1C	R4
3	R3	B	R3	13	R3	1B	R3
2	R2	A	R2	12	R2	1A	R2
1	R1	9	R1	11	R1	19	R1
0	R0	8	R0	10	R0	18	R0

Hình 2.6: Các băng thanh ghi của 8051 và địa chỉ của chúng.

Như ta có thể nhìn thấy từ hình 2.5 băng 1 sử dụng cùng không gian RAM như ngăn xếp. Đây là một vấn đề chính trong lập trình 8051. Chúng ta phải hoặc là không sử dụng băng 1 hoặc là phải đánh một không gian khác của RAM cho ngăn xếp.

VÝ dô 2.5:

Hãy phát biểu các nội dung của các ngăn nhớ RAM sau đoạn chương trình sau:

```
MOV   R0, #99H          ; Nạp R0 giá trị 99H
MOV   R1, #85H          ; Nạp R1 giá trị 85H
MOV   R2, #3FH          ; Nạp R2 giá trị 3FH
MOV   R7, #63H          ; Nạp R7giá trị 63H
MOV   R5, #12H          ; Nạp R5 giá trị12H
```

Lêi gi¶i:

Sau khi thực hiện chương trình trên ta có:

Ngăn nhớ 0 của RAM có giá trị 99H

Ngăn nhớ 1 của RAM có giá trị 85H

Ngăn nhớ 2 của RAM có giá trị 3FH

Ngăn nhớ 7 của RAM có giá trị 63H

Ngăn nhớ 5 của RAM có giá trị 12H

2.6.3 Băng thanh ghi mặc định.

Nếu các ngăn nhớ 00 đến 1F được dành riêng cho bốn băng thanh ghi, vậy băng thanh ghi R0 đến R7 nào ta phải truy cập tới khi 8051 được cấp nguồn? Câu trả lời là các băng thanh ghi 0. Đó là các ngăn nhớ RAM số 0, 1, 2, 3, 4, 5, 6 và 7 được truy cập với tên R0, R1, R2, R3, R4, R5, R6 và R7 khi lập trình 8051. Nó dễ dàng hơn nhiều khi tham chiếu các ngăn nhớ RAM này ơví các tên R0, R1 v.v... hơn là số vị trí của các ngăn nhớ. Ví dụ 2.6 làm rõ khái niệm này.

VÝ dô 2.6:

Hãy viết lại chương trình ở ví dụ 2.5 sử dụng các địa chỉ RAM thay tên các thanh ghi.

Lêi gi¶i:

Đây được gọi là chế độ đánh địa chỉ trực tiếp và sử dụng địa chỉ các vị trí ngăn nhớ RAM đối với địa chỉ đích. Xem chi tiết ở chương 5 về chế độ đánh địa chỉ.

```
MOV   00, #99H          ; Nạp thanh ghi R0 giá trị 99H
MOV   01, #85H          ; Nạp thanh ghi R1 giá trị 85H
MOV   02, #3FH          ; Nạp thanh ghi R2 giá trị 3FH
MOV   07, #63H          ; Nạp thanh ghi R7giá trị 63H
MOV   05, #12H          ; Nạp thanh ghi R5 giá trị12H
```

2.6.4 Chuyển mạch các băng thanh ghi như thế nào?

Như đã nói ở trên, băng thanh ghi 0 là mặc định khi 8051 được cấp nguồn. Chúng ta có thể chuyển mạch sang các băng thanh ghi khác bằng cách sử dụng bit D3 và D4 của thanh ghi PSW như chỉ ra theo bảng 2.2.

Bảng 2.2: Bit lựa chọn các băng thanh ghi RS0 và RS1.

	RS1 (PSW.4)	RS0 (PSW.3)
Băng 0	0	0
Băng 1	0	1
Băng 2	1	0
Băng 3	1	1

Bit D3 và D4 của thanh ghi PSW thường được tham chiếu như là PSW.3 và PSW.4 vì chúng có thể được truy cập bằng các lệnh đánh địa chỉ theo bit như SETB và CLR. Ví dụ "SETB PSW.3" sẽ thiết lập PSW.3 và chọn băng thanh ghi 1. Xem ví dụ 2.7 dưới đây.

VÝ dô 2.7:

Hãy phát biểu nội dung các ngăn nhớ RAM sau đoạn chương trình dưới đây:

```
SETB    PSW.4           ; Chọn băng thanh ghi 4
MOV     R0, #99H        ; Nạp thanh ghi R0 giá trị 99H
MOV     R1, #85H        ; Nạp thanh ghi R1 giá trị 85H
MOV     R2, #3FH        ; Nạp thanh ghi R2 giá trị 3FH
MOV     R7, #63H        ; Nạp thanh ghi R7giá trị 63H
MOV     R5, #12H        ; Nạp thanh ghi R5 giá trị12H
```

Lêi gi¶i:

Theo mặc định PSW.3 = 0 và PSW.4 = 0. Do vậy, lệnh "SETB PSW.4" sẽ bật bit RS1 = 1 và RS0 = 0, bằng lệnh như vậy băng thanh ghi R0 đến R7 số 2 được chọn. Băng 2 sử dụng các ngăn nhớ từ 10H đến 17H. Nên sau khi thực hiện đoạn chương trình trên ta có nội dung các ngăn nhớ như sau:

Ngăn nhớ vị trí 10H có giá trị 99H
Ngăn nhớ vị trí 11H có giá trị 85H
Ngăn nhớ vị trí 12H có giá trị 3FH
Ngăn nhớ vị trí 17H có giá trị 63H
Ngăn nhớ vị trí 15H có giá trị 12H

2.6.5 Ngăn xếp trong 8051.

Ngăn xếp là một vùng bộ nhớ RAM được CPU sử dụng để lưu thông tin tạm thời. Thông tin này có thể là dự liệu, có thể là địa cỉ CPU cần không gian lưu trữ này vì số các thanh ghi bị hạn chế.

2.6.6 Cách truy cập các ngăn xếp trong 8051.

Nếu ngăn xếp là một vùng của bộ nhớ RAM thì phải có các thanh ghi trong CPU chỉ đến nó. Thanh được dùng để chỉ đến ngăn xếp được gọi là thanh ghi con trỏ ngăn xếp SP (Stack Pointer). Con trỏ ngăn xếp trong 8051 chỉ rộng 8 bit có nghĩa là nó chỉ có thể có thể được các địa chỉ từ 00 đến FFH.

Khi 8051 được cấp nguồn thì SP chứa giá trị 07 có nghĩa là ngăn nhớ 08 của RAM là ngăn nhớ đầu tiên được dùng cho ngăn xếp trong 8051. Việc lưu lại một thanh ghi

PCU trong ngăn xếp được gọi là một lần cất vào PUSH và việc nạp nội dung của ngăn xếp trở lại thanh ghi CPU được gọi là lấy ra POP. Hay nói cách khác là một thanh ghi được cất vào ngăn xếp để lưu cất và được lấy ra từ ngăn xếp để dùng tiếp công việc của SP là rất nghiêm ngặt mỗi khi thao tác cất vào (PUSH) và lấy ra (POP) được thực thi. Để biết ngăn xếp làm việc như thế nào hãy xét các lệnh PUSH và POP dưới đây.

2.6.7 Cất thanh ghi vào ngăn xếp.

Trong 8051 thì con trỏ ngăn xếp chỉ đến ngăn nhớ sử dụng cuối cùng của ngăn xếp. Khi ta cất dữ liệu vào ngăn xếp thì con trỏ ngăn xếp SP được tăng lên 1. Lưu ý rằng điều này đối với các bộ vi xử lý khác nhau là khác nhau, đáng chú ý là các bộ vi xử lý × 86 là SP giảm xuống khi cất dữ liệu vào ngăn xếp. Xét ví dụ 2.8 dưới đây, ta thấy rằng mỗi khi lệnh PUSH được thực hiện thì nội dung của thanh ghi được cất vào ngăn xếp và SP được tăng lên 1. Lưu ý là đối với mỗi byte của dữ liệu được cất vào ngăn xếp thì SP được tăng lên 1 lần. Cũng lưu ý rằng để cất các thanh ghi vào ngăn xếp ta phải sử dụng địa chỉ RAM của chúng. Ví dụ lệnh "PUSH 1" là cất thanh ghi R1 vào ngăn xếp.

VÝ dô 2.8:

Hãy biểu diễn ngăn xếp và con trỏ ngăn xếp đối với đoạn chương trình sau đây. Giả thiết vùng ngăn xếp là mặc định.

```
MOV     R6, #25H
MOV     R1, #12H
MOV     R4, #0F3H
PUSH    6
PUSH    1
PUSH    4
```

Lêi gi¶i:

	Sau PUSH 6	Sau PUSP 1	Sau PUSH 4
0B	0B	0B	0B
0A	0A	0A	0A F3
09	09	09 12	09 12
08	08 25	08 25	08 25
Bắt đầu SP = 07	SP = 08	SP = 09	SP = 0A

2.6.8 Lấy nôi dung thanh ghi ra từ ngăn xếp.

Việc lấy nội dung ra từ ngăn xếp trở lai thanh ghi đã cho là qúa trình ngược với các nội dung thanh ghi vào ngăn xếp. Với mỗi lần lấy ra thì byte trên đỉnh ngăn xếp được sao chép vào thanh ghi được xác định bởi lệnh và con trỏ ngăn xếp được giảm xuống 1. Ví dụ 2.9 minh hoạ lệnh lấy nội dung ra khỏi ngăn xếp.

VÝ dô 2.9:

Khảo sát ngăn xếp và hãy trình bày nội dung của các thanh ghi và SP sau khi thực hiện đoạn chương trình sau đây:

```
POP     3       ; Lấy ngăn xếp trở lại R3
POP     5       ; Lấy ngăn xếp trở lại R5
```

```
POP     2          ; Lấy ngăn xếp trở lại R2
```

Lêi gi¶i:

		Sau POP3		Sau POP 5		Sau POP 2	
0B	54	0B		0B		0B	
0A	F9	0A	F9	0A		0A	
09	76	09	76	09	76	09	
08	6C	08	6C	08	6C	08	6C
Bắt đầu SP = 0B		SP = 0A		SP = 09		SP = 08	

2.6.9 Giới hạn trên của ngăn xếp.

Như đã nói ở trên, các ngăn nhớ 08 đến 1FH của RAM trong 8051 có thể được dùng làm ngăn nhớ 20H đến 2FH của RAM được dự phòng cho bộ nhớ đánh địa chỉ được theo bit và không thể dùng trước cho ngăn xếp. Nếu trong một chương trình đã cho ta cần ngăn xếp nhiều hơn 24 byte (08 đến 1FH = 24 byte) thì ta có thể đổi SP chỉ đến các ngăn nhớ 30 đến 7FH. Điều này được thực hiển bởi lệnh "MOV SP, #XX".

2.6.10 Lệnh gọi CALL và ngăn xếp.

Ngoài việc sử dụng ngăn xếp để lưu cất các thanh ghi thì CPU cũng sử dụng ngăn xếp để lưu cất tam thời địa chỉ của lệnh đứng ngay dưới lệnh CALL. Điều này chính là để PCU biết chỗ nào để quay trở về thực hiện tiếp các lệnh sau khi chọn chương trình con. Chi tiết về lệnh gọi CALL được trình bỳ ở chương 3.

2.6.11 Xung đột ngăn xếp và băng thanh ghi số 1.

Như ta đa nói ở trên thì thanh ghi con trỏ ngăn xếp có thể chỉ đến vị trí RAM hiện thời dành cho ngăn xếp. Khi dữ liệu được lưu cất cào ngăn xếp thì SP được tăng lên và ngược lại khi dữ liệu được lấy ra từ ngăn xếp thì SP giảm xuống. Lý do là PS được tăng lên sau khi PUSH là phải biết lấy chắc chắn rằng ngăn xếp đang tăng lên đến vị trí ngăn nhớ 7FH của RAM từ địa chỉ thấp nhất đến địa chỉ cao nhất. Nếu con trỏ ngăn xếp đã được giảm sau các lệnh PUSH thì ta nên sử dụng các ngăn nhớ 7, 6, 5 v.v... của RAM thuộc các thanh ghi R7 đến R0 của băng 0, băng thanh ghi mặc định. Việc tăng này của con trỏ ngăn xếp đối với các lệnh PUSH cũng đảm bảo rằng ngăn xếp sẽ không với tới ngăn nhớ 0 của RAM (đáy của RAM) và do vậy sẽ nhảy ra khỏi không gian dành cho ngăn xếp. Tuy nhiên có vấn đề nảy sinh với thiết lập mặc định của ngăn xếp. Ví dụ SP = 07 khi 8051 được bật nguồn nên RAM và cũng thuộc về thanh ghi R0 củ băng thanh ghi số 1. Hay nói cách khác băng thanh ghi số 1 và ngăn xếp đang dùng chung một không gian của bộ nhớ RAM. Nếu chương trình đã cho cần sử dụng các băng thanh ghi số 1 và số 2 ta có thể đặt lại vùng nhớ RAM cho ngăn xếp. Ví dụ, ta có thể cấp vị trí ngăn nhớ 60H của RAM và cao hơn cho ngăn xếp trong ví dụ 2.10.

VÝ dô 2.10:

Biểu diễn ngăn xếp và con trỏ ngăn xếp đối với các lệnh sau:

```
MOV     SP, #5FH          ; Đặt ngăn nhớ từ 60H của RAM cho ngăn xếp
MOV     R2, #25H
MOV     R1, #12H
MOV     R4, #0F3H
PUSH    2
```

PUSH 1
PUSH 4

Lêi gi¶i:

	Sau PUSH 2	Sau PUSP 3	Sau PUSH 4
63	63	63	63
62	62	62	62 F3
61	61	61 12	61 12
60	60 25	60 25	60 25
Bắt đầuSP=5F	SP = 60	SP = 61	SP = 62

Các ngắt của hệ thống hỗ trợ cho lập trình ASSSEMBLY

Có 4 hàm hay dùng nhất:

Hàm 1: Chờ 1 ký tự từ bàn phím:

```
Mov ah, 1;AL chứa mã ASCII ký tự mã vào
Int 21h
```

Hàm 2: Đưa 1 ký tự dạng ASCII ra màn hình tại vị trí con trỏ đang đứng

Cách 1: Nhờ ngắt của BIOS

```
Mov al, mã ASCII của ký tự
Mov ah, oeh
Int 10h
```

Cách 2:

```
Mov dl, mã ASCII của ký tự
Mov ah, 2
Int 21h
```

Hàm 3: Hiện 1 xâu ký tự kết thúc bằng dấu $ ra màn hình

```
Mov dx, offset tên biến xâu
Mov ah, 9
Int 21h
```

Hàm 4: Trở về DOS

```
Mov ah, 4ch      ;[int 20h]
Int 21h
```

Các DIRECTIVE điều khiển SEGMENT dạng đơn giản: dễ viết, dễ liên kết nhưng chưa bao hết mọi tình huống về điều khiển SEGMENT

.Model thuê vùng nhớ RAM thích hợp cho chương trình

.Model kiểu_tiny code+data≤64KB
.Model kiểu_small code≤64KB;data≤64KB
.Model kiểu_compact code≤64KB;data≥64KB
.Model kiểu_medium code≥64KB;data≤64KB
.Model kiểu_large code≥64KB;data≥64KB song khi khai báo1 array không ≤64KB
.Model kiểu_large code≥64KB;data≥64KB song khi khai báo1 array không >64KB

.Stack Độ lớn (Byte) → Xác lập độ lớn stack cho chương trình

.Data Xác lập vùng nhớ cho dữ liệu của chương trình khai báo biến nằm ở segment này

.Data Khai báo biến (có 3 loại biến)

Với các loại biến số thì kiểu db có độ dài 1 byte
dw có độ dài 2 byte
dd có độ dài 4 byte
dp/df có độ dài 6 byte
dq có độ dài 8 byte
dt có độ dài 10 byte

Với các loại biến xâu có các cách khai báo như sau:

```
tên_biến_xâu  db  'các ký tự'
tên_biến_xâu  db  độ_lớn  dup('1 ký tự')
tên_biến_xâu  db  độ_lớn  dup(?)
```

Với các loại biến trường số (array) có các cách khai báo như sau:

```
tên_biến_trường  kiểu_thành_phần  (các số cách nhau bởi dấu ,)
tên_biến_trường  kiểu_thành_phần  độ_lớn  dup(giá trị 1 số)
tên_biến_trường  kiểu_thành_phần  độ_lớn  dup(?)
```

.Code Xác lập vùng nhớ truy xuất ngẫu nhiên dùng cho phần mã máy

```
.Code
    nhãn_chương_trình:
        mov ax, @data
        mov ds, ax
        ...
        thân chương trình
        ...
        mov ah, 4ch
        int 21h
        [các chương trình con]
    end nhãn_chương_trình
```

Các DIRECTIVE điều khiển SEGMENT dạng chuẩn: _SEGMENT; _GROUP; _ASSUME

_SEGMENT Xác lập các segment cho chương trình

```
tên_segment  SEGMENT  align  combine  use  'class'
{thân segment}
tên_segment  ENDS
```

trong đó:

tên_segment là 1 identifier (không chứa dấu cách, dấu \ ; : ...)

align là cách xác lập khoảng cách giữa segment đang khai báo với segment trước nó

```
align
byte khoảng cách 1 byte
word khoảng cách 2 byte
paka khoảng cách 16 byte
page khoảng cách 256 byte
```

combine có hai chức năng:

+cho phép đặt segment vào 1 địa chỉ mong muốn (theo yêu cầu) của bộ nhớ RAM

```
tên_segment  SEGMENT  at  địa_chỉ_dạng_vật_lý
{thân}
tên_segment  ENDS
```

+cho chương trình đa tệp: cách gộp các segment có cùng tên nằm ở các tệp khác nhau khi liên kết. Ví dụ: Tệp 1 có DATA SEGMENT; Tệp 2 có DATA SEGMENT.khác

_COMMON tức là độ dài của segment sau liên kết bằng độ dài segment lớn nhất

_PUBLIC tức là độ dài của segment sau liên kết bằng tổng độ dài cả 2 segment

_PRIVATE tức là độ dài của segment sau liên kết bằng độ dài của chính nó (không quan hệ với nhau, đây là chế độ default)

_STACK giống như _PUBLIC

_CLASS sắp xếp các segment lại gần nhau sau khi liên kết (Sau khi liên kết thì những segment nào cùng một nhóm thì ở gần nhau)

_GROUP gộp các segment có cùng kiểu lại với nhau cho dễ dàng qui chiếu

```
tên_ nhóm  GROUP  tên_các_segment (cách nhau bởi dấu , )
```

_ASSUME cho biết tên segment thuộc loại segment nào

```
_ASSUME tên_thanh_ghi  SEG : tên_seg
```

Cấu trúc một chương trình Assembly thường thấy: Khai báo MACRO, STRUCT, UNION, RECORD, SEGMENT

Dạng đơn giản

```
.Model  kiểu
.Stack  độ lớn
[.Data
    khai báo biến]
.Code
    nhãn_chương_trình:
        mov ax, @data
        mov ds, ax
        ...
        thân chương trình
        ...
        mov ah, 4ch
        int 21h
        [các  chương trình con]
    END nhãn_chương_trình
```

Dạng chuẩn

```
_Stack  segment
    db  độ_dài  dup(?)
_Stack  ends
Data  segment
    khai báo biến
Data  ends
Code  segment
        Assume  cs:code  ds:data  ss:stack nhãn_chương_trình:
        mov ax, data
        mov ds, ax
        ...
        thân chương trình
        ...
        mov ah, 4ch
        int 21h
        [các  chương trình con]
    ENDS
        END nhãn_chương_trình
```

***Chương trình con:**

1, Cơ chế khi 1 chương trình con bị gọi:

Bước 1: Tham số thực đưa vào STACK

Bước 2: Địa chỉ của lệnh tiếp theo được đưa vào STACK

Bước 3: Hệ điều hành quản lý địa chỉ đầu của chương trình con do vậy Hệ điều hành sẽ đưa địa chỉ đó vào CS:IP → rẽ nhánh vào chương trình con

Bước 4: Thực hiện thân chương trình con cho đến khi gặp lệnh RET thì vào STACK lấy địa chỉ lệnh tiếp theo (đã cất ở **Bước 2**) và cho vào CS:IP, rồi quay về chương trình đã gọi nó

Bước 5: Tiếp tục thực hiện chương trình đang đợi

2, Cú pháp của chương trình con Assembly:

```
Tên_chương_trình_con  PROC  [NEAR/FAR]
    Bảo vệ các thanh ghi sẽ bị phá vỡ ở thân chương trình
    Thân chương trình con
    Hồi phục các thanh ghi mà chương trình con phá vỡ
    RET
Tên_chương_trình_con  ENDP
```

a, Vấn đề NEAR – FAR:

NEAR chương trình con cùng nằm trên 1 segment với chương trình gọi nó → địa chỉ lệnh tiếp theo cất vào STACK (**Bước 2**) chỉ cần 2 byte offset
FAR chương trình con nằm khác segment với chương trình con gọi nó → địa chỉ lệnh tiếp theo cất vào STACK (**Bước 2**) cần đến 4 byte offset
***** Với khai báo segment dạng đơn giản thì directive model sẽ xác định hiện chương trình con NEAR hay FAR
. Model tiny → chương trình con là NEAR
. Model small → chương trình con là NEAR
. Model compact → chương trình con là NEAR
. Model medium→ chương trình con là FAR
. Model large → chương trình con là FAR
. Model huge → chương trình con là FAR
***** Với khai báo dạng chuẩn thì mặc định là NEAR

b, Chương trình con Assembly không có đối số → cơ chế kích hoạt chương trình con Assembly không có Bước 1

3, Chuyển giao tham số:

Cách 1: Nhờ thanh ghi

```
Chương_trình_chính          Chương_trình_con
......                      ......
mov ax, 10                  mov bx, ax
call Chương_trình_con       ......
......
(khi đó bx = 10)
```

Cách 2: Nhờ biến nhớ

```
Chương_trình_chính          Chương_trình_con
......                      ......
mov value, 20               mov bx, value
call Chương_trình_con       ......
......
(khi đó bx = 20)
```

Cách 3: Thông qua STACK (dùng khi liên kết với ngôn ngữ lập trình bậc cao)

4, Bảo vệ thanh ghi:

Khi thân chương trình con sử dụng các lệnh làm giá trị thanh ghi thay đổi như and, xor, ... thì phải bảo vệ các thanh ghi đó trước khi dùng
Cách 1: Dùng các lệnh POP, PUSH
Cách 2: Chuyển vào biến hoặc thanh ghi khác sau đó hồi phục

Tệp include

Tệp INCLUDE cho phép người lập trình viết gọn chương trình
- Thiết lập 1 tệp ngoài (gọi là tệp INCLUDE) mà trong tệp này chứa khối lệnh Assembly .ASM
- Sau đó dùng lệnh INCLUDE để chèn khối lệnh đó vào tệp chương trình đang viết. Cú pháp:

```
.............
INCULDE   X:\đường dẫn\tệp INCLUDE
.............
```

- Cơ chế của chương trình dịch Assembly khi gặp INCLUDE: chương trình dịch của Tubor Assember khi gặp INCLUDE thì thực hiện các bước:
***** Mở tệp INCLUDE theo sau Directive Include
***** Sao chép và chèn toàn bộ khối lệnh Assembly có trong tệp INCLUDE vào vị trí Directive Include đứng
***** Tiến hành dịch khối lệnh đó
- Cách tìm tệp INCLUDE để chèn
***** Nếu tệp đứng sau Directive Include có tên đĩa, đường dẫn thì chương trình dịch tìm đến, nếu không có thì đó là thư mục hiện hành, còn nếu không có nữa thì sai
***** Nếu sai thì chỉ có cách sửa lại chương trình nguồn. Khi dịch chương trình dùng thêm tham số TASM -i A:\...*.ASM
- Hạn chế của INCLUDE là không được phép có nhãn nhảy trong khối lệnh của tệp INCLUDE khi gọi nó 2 lần

macro và các vấn đề liên quan

1, Các lệnh lặp khối lệnh khi dịch chương trình:

REPT dịch khối lệnh theo số lần đi sau REPT

```
    REPT n
        Khối_lệnh
    ENDM
```

IRP dịch khối lệnh theo số lượng danh sách

```
    IRP tên_đối <danh sách>
        Khối_lệnh
    ENDM
```

- Các Directive điều khiển điều kiện khi dịch chương trình
Chức năng: Dịch khối lệnh khi điều kiện đúng TRUE

```
IF <điều kiện>          IF <điều kiện>
    Khối_lệnh               Khối_lệnh_1
ENDIF                   ELSE    Khối_lệnh_2
                        ENDIF
```

Lệnh IFE giống như lệnh IF nhưng ngược điều kiện
Chức năng: Dịch khối lệnh khi biểu thức <điều kiện> = 0

```
IFB <biểu thức>
    Khối_lệnh
ENDIF
```

Lệnh IFNB giống như lệnh IFB nhưng ngược điều kiện
Chức năng: Dịch khối lệnh khi <biểu thức 1> = <biểu thức 2>

```
IFIDN <biểu thức 1>, <biểu thức 2>
    Khối_lệnh
ENDIF
```

Lệnh IFDIF giống như lệnh IFIDN nhưng ngược điều kiện
Chức năng: Dịch khối lệnh khi nhãn theo sau đó đã được khai báo

```
IFDEF nhãn
    Khối_lệnh
ENDIF
```

Lệnh IFNDEF giống như lệnh IFDEF nhưng ngược điều kiện
***Macro** là 1 cơ chế giúp người lập trình tạo 1 lệnh mới trên cơ sở tập lệnh sẵn có của Assembly
- Trước khi được dùng thì phải khai báo
Cú pháp:

```
Tên_Macro  MACRO[đối]
    Bảo vệ các thanh ghi sẽ bị phá vỡ ở thân chương trình
    Thân MACRO
    Hồi phục các thanh ghi mà chương trình con phá vỡ
ENDM
```

- Cách dùng lệnh mới đã xác lập ở MACRO: Sau khi macro đã được xác lập thì tên của Macro trở thành một lệnh mới của ASM
- Cách dùng: Gọi tên macro và thay vì đối sẽ là tham số thực

Chương trình Macro hiện xâu ra màn hình:

```
    HIENSTRING   MACRO   XAU
    Push    ax, bx
    Lea     dx, XAU
    Mov     ah, 9
    Int  21h
    Pop dx, ax
    ENDM
```

Chương trình Macro xóa màn hình:

```
    CLRSCR   MACRO
    Push    ax
    Mov     ah, 0fh
    Int  10h
    Mov     ah, 0
    Int  10h
    Pop     ax
    ENDM
```

tính ưu việt của macro

a, So sánh Macro với chương trình con:

Tốc độ: Khi chạy chương trình thì Macro nhanh hơn vì không phải dùng lệnh CALL và RET
Tiết kiệm bộ nhớ: Chương trình con chiếm ít bộ nhớ hơn
Macro cho phép chuyển giao tham số thông qua đối và cho phép sử dụng các Directive lặp khidịch chương trình. Các Directive điều khiển điều kiện khi dịch chương trình.

b, So sánh Macro với tệp INCLUDE:

Cơ chế: Giống nhau khi dịch
Tốc độ: Khi chạy chương trình thì Macro nhanh hơn vì không phải mở đóng tệp
Macro cho phép có nhãn nhảy trong lệnh của Macro nhờ Directive Local. Trong thân Macro cho phép có các Macro khác

Chương trình dạng *.com và *.exe

Chương trình .EXE có 3 segment {code, data và stack}. Có thể không cùng nằm trên 1 segment
Chương trình .COM có 3 segment {code, data và stack} nằm cùng trên 1 segment. Khi chạy chương trình .COM cần 256 byte đầu của segment đó để nhảy. Do vậy, lệnh đầu của chương trình .COM sẽ đặt ở offset → người lập trình phải khai báo cho hệ điều hành Directive ORG
Khai báo chương trình dạng .COM có 1 segment và là Code Segment → biến cũng được khai báo ở Code Segment

```
.Code
    Nhãn_chương trình:
        Jmp   nhãn_khác
        [nếu có khai báo biến]
        Khai báo biến
        Nhãn_khác:
        ......
        mov   ah, 4ch
        int 21h
```

Dạng thường thấy của chương trình .COM thuần túy [Khai báo MACRO, STACK, UNION, RECORD]

Dạng đơn giản	**Dạng chuẩn**
.Model tiny	**.Code** segment
(hoặc small)	ORG 100h

.Code **assume** cs:code,ds:code,ss:code
ORG 100h Nhãn_chương_trình:
Nhãn_chương_trình: Jmp nhãn_khác
Jmp nhãn_khác Khai báo biến
Khai báo biến Nhãn_khác:
Nhãn_khác: .....
..... int 20h
int 20h [các chương trình con]
[các chương trình con] code ends
END nhãn_chương_trình END nhãn_chương_trình

Directive public

Chức năng: Báo cho chương trình dịch biết những nhãn ở Model này cho phép các tệp khác cũng có thể dùng
Cú pháp: Public tên_nhãn
Khai báo kiểu nhãn
.Với hằng: Public tên_hằng = hằng
Public Port
Port = 038h
.Với biến: Public tên_biến
Khai báo biến
.Với tên chương trình con:
Public tên_chương_trình_con
tên_chương_trình_con PROC
.....
RET
tên_chương_trình_con ENDP

Directive public

Chức năng: Báo cho chương trình dịch biết Module này xin phép được dùng các nhãn mà các Module khác đã cho phép
Cú pháp: Extrn tên_nhãn: kiểu
.Nhãn là tên hằng: Extrn tên_nhãn: ABS
Extrn Post kiểu
.Nhãn là biến nhớ: Extrn x: word (hoặc byte hoặc dword)
.Nhãn là chương trình con:
Extrn tên_chương_trình_con:PROC

Directive global

Chức năng: Không phải chương trình nào cũng có Directive này, nó thay cho Public và Extrn
Cú pháp: GLOBAL tên_nhãn: kiểu
Khai báo biến

Liên kết C với Assembly

INLINE ASM là chèn khối lệnh ASM vào chương trình được viết bằng C
Cú pháp: khối lệnh C
ASM lệnh ASM
.....
ASM lệnh ASM
khối lệnh C
Dịch và liên kết
TCC -ms :IC\TC\INCLUDE -LC
Hạn chế: Các lệnh ASM được chèn thì dịch nhờ bởi chương trình dịch của TC. Do đó 1 số lệnh khó của ASM dịch không đúng. Không cho phép có các nhãn nhảy trong ASM → khối lệnh chèn vào yếu (vì không có LOOP, nhảy có và không có điều kiện)

Viết tách biệt tệp cho c và tệp cho asm

Phải giải quyết 3 vấn đề:
1, Vấn đề đa tệp: (khai báo Public) với Module của C, bất kỳ khai báo nào của C đều là khai báo Public. Khai báo External ngôn ngữ C phải xin phép dùng các nhãn đã cho phép từ tệp ngoài. Với Module ASM giống như đa tệp thuần túy
2, Vấn đề dấu (-) (underscore) người viết chương trình ASM phải thêm dấu – vào trước trên các nhãn dùng chung với C và thêm ở mọi nơi mà tên đó xuất hiện
3, Vấn đề giá trị quay về của hàm ASM: qui định với giá trị 2 byte thì trước khi RET ax = bao nhiêu thì tên hàm ASM có giá trị bấy nhiêu. Với giá trị 4 byte trước khi RET dx:ax có giá trị bao nhiêu thì hàm ASM có giá trị bấy nhiêu

cơ chế khi một ngắt và chương trình con được kích hoạt

Chương trình con bình thường:
CALL
Bước 1: Tham số thực → STACK
Bước 2: Địa chỉ lệnh tiếp theo → STACK
Bước 3: Hệ điều hành quản lý địa chỉ đầu của chương trình con → Hệ điều hành đưa địa chỉ đầu của chương trình con → cs:ip → rẽ nhánh vào chương trình con
Bước 4: Thực hiện các lệnh của chương trình con → RET thì vào STACK lấy địa chỉ lệnh tiếp theo (đã cất ở bước 2) → cs:ip và trở về chương trình đang dở
Bước 5: Tiếp tục chương trình đang dở

Chương trình con phục vụ ngắt:
Int n (tác động linh kiện)
Bước 1: Flag → STACK;Tham số thực → STACK
Bước 2: Địa chỉ lệnh tiếp theo → STACK
Bước 3: Hệ điều hành quản lý địa chỉ đầu của chương trình con phục vụ ngắt. Song địa chỉ đầu của chương trình con phục vụ ngắt nằm trong ô nhớ tương ứng của bảng vectơ ngắt → máy tính vào vectơ ngắt lấy địa chỉ đầu của chương trình con phục vụ ngắt đưa vào cs:ip → rẽ nhánh vào chương trình con phục vụ ngắt
Bước 4: Thực hiện các lệnh của chương trình con cho đến khi gặp IRET thì vào STACK lấy địa chỉ lệnh tiếp theo (đã cất ở bước 2) → cs:ip và trở về chương trình đang dở
Bước 5: Trước khi tiếp tục chương trình đang dở thì vào STACK lấy cờ đã cất
Bảng vectơ ngắt: là vùng nhớ RAM chứa địa chỉ đầu của chương trình con phục vụ ngắt. Máy tính có 256 ngắt → có 256 chương trình con phục vụ ngắt. Địa chỉ ô bằng n * 4 (mỗi địa chỉ 4 byte)
Các bước để xác lập chương trình con phục vụ ngắt:
Bước 1: Viết chương trình con theo yêu cầu của thuật toán
Cú pháp: Tên_chtrình_con_pvụ_ngắt PROC [NEAR/FAR]
Bảo vệ các thanh ghi
Thân chương trình
Phục hồi các thanh ghi
IRET
Tên_chtrình_con_pvụ_ngắt ENDP
Bước 2: Sau khi viết xong chương trình con phục vụ ngắt thì tìm địa chỉ đầu của chương trình này đưa vào vị trí tương ứng của bảng vectơ ngắt
Khởi động máy tính với hệ điều hành DOS
Với máy tính của INTEL, khi bật máy thì thanh ghi CS = F000h; IP = FFF0h và sẽ nhảy vào thực hiện lệnh ở ô nhớ F000:FFF0. Lệnh này là lệnh jmp và nó nhảy đến chương trình khởi động máy tính đều nằm ở ROM-BIOS
ROM-BIOS là vùng nhớ chỉ đọc, không ghi được và chứa 2 loại chương trình khởi động máy và chương trình phục vụ ngắt của BIOS
Các chương trình khởi động máy tính:
Test CPU: kiểm tra các thanh ghi. Tống vào các giá trị 00, 55 và FF vào các thanh ghi và kiểm tra lại có bằng 00, 55 và FF không. Đồng thời kiểm tra một số lệnh ASM nếu có lỗi thì hiện FATA ERROR.
Kiểm tra ROM-BIOS: trong ROM có 1 byte CHECKSUM (tổng các byte của ROM) khi khởi động thì có 1 chương trình cộng các byte của ROM lại lưu kết quả vào 1 byte và so sánh byte này với CHECKSUM. Nếu bằng nhau thì tức là ROM tốt, ngược lại là tồi.
Kiểm tra một số linh kiện quan trọng của mainboard
8259 là chip phục vụ ngắt
8250 UART (COM)
8253 Timer
8237 DMA
Kiểm tra RAM (giống hệt CPU và thanh ghi) tức là cho toàn bộ các byte của RAM các giá trị 00, 55, FF liệu RAM có chấp nhận các giá trị này không
Xác lập bảng vec tơ ngắt của BIOS
Đưa mọi địa chỉ đầu của các chương trình con phục vụ ngắt vào bảng vec tơ ngắt
Đưa các thông số máy tính đang dùng vào vùng nhớ biến BIOS
Kiểm tra liệu có ROM mở rộng: với màn hình và ổ đĩa thì về phần cứng cho các Card điều khiển không giống nhau → không thể viết 1 driver chung và nạp vào ROM-BIOS chuẩn → thỏa hiệp của các hãng: Ai sản xuất phần cứng thì viết driver cho nó và nạp vào ROM và ROM đó sẽ được đặt trên Card đó
Int 19h: Lôi boot sector xuống RAM và trao quyền cho chương trình nằm trong boot sector
Trong boot sector là sector 512 byte chứa tham số đĩa và chứa chương trình mồi
Chương trình mồi lôi 2 tệp ẩn xuống RAM (hệ điều hành DOS)
Kiểm tra thiết bị ngoại vi
Lôi COMMAND.COM vào vùng nhớ RAM – là chương trình dịch các lệnh của DOS → Mã máy
CONFIG.SYS
AUTOEXEC.BAT
C:\>

Bài tập 1:
Hiện 1 xâu ký tự "Hello TASM!" ra màn hình
Cách 1:

```
.MODEL small
.STACK 100h
.DATA
    Message db 'Hello TASM!$'
.CODE
ProgramStart:
    Mov     AX,@DATA
    Mov     DS,AX
    Mov     DX,OFFSET Message
    Mov     AH,9
    Int     21h
    Mov     AH,4Ch
    Int     21h
    END     ProgramStart
```

Cách 2:

```
_STACK    segment    stack   'stack'
    db  100h   dup(?)
_STACKends
DATA      segment
    Message db 'Hello TASM!',0
DATA   ends
CODE      segment
    AssumeCS:CODE, DS:DATA, SS:_STACK
ProgramStart:
    Mov     AX,DATA
    Mov     DS,AX
    Mov     SL,OFFSET Message
    cld
L1:
    Lodsb
    AndAL,AL
    Jz      Stop
    Mov     AH,0eh
    Int     10h
    Jmp     L1
Stop:
    Mov     AH,1
    Int     21h
    Mov     AH,4Ch
    Int     21h
CODE      ends
    END     ProgramStart
```

Bài tập 2:
So sánh 2 số nguyên nhập từ bàn phím xem số nào bé hơn
Cách 1:

```
include C:\HTDAT\INCLUDE\lib1.asm
_stack segment stack 'stack'
     db 100h dup(?)
_stack ends
data segment
    m1 db 10,13,'Vao so thu nhat:$'
    m2 db 10,13,'Vao so thu hai:$'
    m3 db 10,13,'So be la:$'
    m4 db 10,13,'Co tiep tuc khong (c/k)?:$'
data ends
code segment
  assume cs:code,ds:data,ss:_stack
  ps:
    mov ax,data
    mov ds,ax
    clrscr
    hienstring m1
    call Vao_so_N
    mov bx,ax
    hienstring m2
    call Vao_so_N
    cmp ax,bx
    jl L1
    xchg ax,bx
 L1:
    hienstring m3
    call Hien_so_N
    hienstring m4
    mov ah,1
    int 21h
    cmp al,'c'
    je ps
    mov ah,4ch
    int 21h
include C:\HTDAT\INCLUDE\lib2.asm
code ends
end ps
```

So sánh 2 số nhập vào từ bàn phím xem số nào bé hơn
Cách 2:

```
hien_string MACRO xau
  push ax dx
   mov dx,offset xau
   mov ah,9
   int 21h
  pop dx ax
ENDM
;---------------------------------
.model small
.stack 100h
.data
    sohex   dw      ?
    temp    dw      ?
    m1   db     0ah,0dh,'Vao so thu1: $'
    m2   db     0ah,0dh,'Vao so thu2: $'
    m3   db     0ah,0dh,'So be la:  $'
.code
  ps:
     mov ax,@data
     mov ds,ax
     hien_string m1
     call VAOSO
     mov ax,sohex
     mov temp,ax
     hien_string m2
     call VAOSO
     mov bx,sohex
     hien_string m3
     cmp ax,bx
      jl L1
       xchg ax,bx
    L1:
     call HIENSO
     mov  ah,1
     int  21h
     mov ah,4ch
     int 21h
;-----------------------------------------------
   VAOSO PROC
     push ax bx cx dx
     mov bx,10
     xor cx,cx
     mov sohex,cx
  VS1:
     mov ah,1   ; Ham nhan 1 ki tu va --->al
     int 21h
     cmp al,0dh
      je VS2
     sub al,30h
     mov cl,al
     mov ax,sohex
     mul bx
     add ax,cx
     mov sohex,ax
   jmp VS1
  VS2:
     pop dx cx bx ax
     ret
  VAOSO ENDP
;-----------------------------------------------
  HIENSO PROC
    push ax bx cx dx
    mov bx,10
    xor cx,cx
 HS1:
   xor dx,dx
   div bx       ; tuc lay dx:ax chia cho bx kq thuong-->ax va du-->dx
```

```
  add dx,30h   ; de dua ra dang  ASCCI
  push dx      ; tong 1 chu vao stack
  inc cx
  cmp ax,0
  jnz HS1
 HS2:
  pop ax
  mov ah,0eh
  int 10h
 loop HS2
  pop dx cx bx ax
  ret
 HIENSO ENDP
end ps
```

Bài tập 3:

Tính trung bình cộng 2 só nguyên nhập từ bàn phím

```
INCLUDE C:\INCLUDE\LIB1.ASM
_STACKsegment
    db  100h    dup(?)
_STACKends
DATA        segment
    M1      db  'Hay vao so thu 1: $'
    M2      db  0ah,0dh,'Hay vao so thu 2: $'
    M3      db  0ah,0dh,'Trung binh cong cua 2 so nguyen la: $'
    M4      db  '-$'
    M5      db  '.5$'
    M6      db  0ah,0dh,' Co tiep tuc khong (c/k) ?: $'
DATA        ends
CODE        segment
    assume cs:code,ds:data,ss:_stack
ps:
    mov ax,data
    mov ds,ax
    clrscr
    HienString M1
    call VAO_SO_N
    mov bx,ax
    HienString M2
    call VAO_SO_N
    HienString M3
    Add ax,bx
    And ax,ax
    Jns L1
    HienString M4
    Neg ax
L1:
    Shr ax,1
    Pushf
    Call HIEN_SO_N
    Popf
    Inc L2
    HienString M5
L2:
    HienString M6
    Mov ah,1
    Int 21h
    Cmp al,'c'
    Je TT
    Mov ah,4ch
    Int 21h
TT:
    Jmp ps
INCLUDE C:\INCLUDE\LIB2.ASM
CODE   ends
    END ps
```

Bài tập 4:

Nhập một số nguyên dương n từ bàn phím và tìm giai thừa của nó

<u>Cách 1:</u>

```
include C:\HTDAT\INCLUDE\lib1.asm
_stack segment stack 'stack'
    db 100h dup(?)
_stack ends
data segment
  fv dw ?
  fac dw ?
  m1 db 10,13,'Vao so n:$'
  m2 db 10,13,'Giai thua cua $'
  m3 db ' la:$'
  m4 db 10,13,'Co tiep tuc khong(c/k)?: '
data ends
code segment
  assume cs:code,ds:data,ss:_stack
    ps:
      mov ax,data
      mov ds,ax
      clrscr
      hienstring m1
      call vao_so_N
      hienstring m2
      call Hien_so_N
      hienstring m3
      call S_N_T
      mov ax,fv
      call hien_so_N
      hienstring m4
      mov ah,1
      int 21h
      cmp al,'c'
      je ps
      mov ah,4ch
      int 21h
include C:\HTDAT\INCLUDE\lib3.asm
include C:\HTDAT\INCLUDE\lib2.asm
code ends
end ps
```

Chương trình tính giai thừa của một số n nhập từ bàn phím

<u>Cách 2:</u>

```
code segment
    assume cs:code,ds:code
    org 100h
start: jmp do
msg1 db 'nhap vao mot so:$'
msg2 db 'ket qua la:$'
giaithua  dw  1
so dw 0
m db 'ok $'
do :
     mov ah,09h
     mov dx,offset msg1
     int 21h
     call nhapso
     call cr_lf
     mov bx,1
     mov cx,ax
lap:
    mov ax,giaithua
    mul bx
    inc bx
    mov giaithua,ax
    loop lap
    mov ax,giaithua
    push ax
    push dx
    mov ah,09h
    mov dx,offset msg2
    int 21h
    pop dx
    pop ax
    call inra
    mov ah,01h
    int 21h
    int 20h
;---------------------------
cr_lf  proc near
    push ax
    push dx
    mov ah,02h
    mov dx,0dh
    int 21h
    mov dx,0ah
    int 21h
    pop dx
    pop ax
    ret
cr_lf endp
;---------------------------
nhapso proc near
    push dx
    push cx
```

```
    push bx
    xor dx,dx
    mov so,0
    mov cx,1
lap1:  call nhap
    cmp al,0dh
    je exit
    sub al,30h
    xor ah,ah
    xor dx,dx
    mov dx,ax
    mov ax,so
    cmp cx,1
    je nota
    mov bl,10
    mul bl
nota: add ax,dx
    mov so,ax
    inc cx
    jmp lap1
exit: mov ax,so
   pop bx
   pop cx
   pop dx
   ret
nhapso endp
;---------------------------
inra proc
    mov bx,10
    xor cx,cx
none_zero:
     xor dx,dx
     div bx
     push dx
     inc cx
     or ax,ax
     jnz none_zero
write: pop dx
     add dl,'0'
     mov ah,02
     int 21h
     loop write
     ret
inra endp

;---------------------------
public nhap
nhap proc near
sta :
    push dx
    mov ah,08
    int 21h
    cmp al,0dh
    je exit1
    cmp  al,30h
    jb sta
    cmp al,39h
    ja sta
     mov dl,al
;    xor ah,ah

    mov ah,02h
    int 21h
exit1:
    pop dx
    ret
nhap endp
;----------------------------
code ends
end start
```

Bài tập 5:

Tìm số nguyên tố nhỏ hơn hoặc bằng số giới hạn cho trước

```
INCLUDE C:\INCLUDE\LIB1.ASM
_STACKsegment
    db  100h    dup(?)
_STACKends
DATA        segment
    M1      db  'Hay vao so gioi han: $'
    M2      db  0ah,0dh,' Cac so nguyen to tu 2 den $'
    M3      db  'la: $'
    M4      db  0ah,0dh,' Co tiep tuc khong (c/k) ?: $'
    So      dw  dw
DATA        ends
CODE        segment
    Assume CS:CODE, DS:DATA, SS:_STACK
PS:
    Mov AX,DATA
    Mov DS,AX
    CLRSCR
    HienString M1
    Call VAO_SO_N
    HienString M2
    Call VAO_SO_N
    HienString M3
    Mov BX,AX
    Mov so,1
L1:
    Inc so
    Mov AX,so
    Cmp AX,BX
    Jg Stop
    Mov CX,AX
    Shr CX,1
L2:
    Cmp CX,1
    Jle L3
    Xor DX,DX
    Div CX
    And DX,DX
    Jz L1
    Mov AX,so
    Loop L1
L3:
    Call HIEN_SO_N
    HienString M4
    Jmp L1
Stop:
    HienString M5
    Mov AH,1
    Int 21h
    Cmp AL,'c'
    Je TT
    Mov AH,4Ch
    Int 21h
TT:
    Jmp PS
INCLUDE C:\INCLUDE\LIB2.ASM
CODE ends
    END PS
```

Bài tập 6:

Nhập 2 số vào từ bàn phím và in ra tích của chúng

```
EXTRN
    CR_LF:PROC,PRINT_CHAR:PROC,GET_IN_NUMBER:PROC,WRITE_C
HAR:PROC
;----------------------------------
DATA_SEG SEGMENT PUBLIC
DATA_1      DB     'ENTER TWO STRING:$'
DATA_2      DB     'NUMBER1:$'
DATA_3      DB     'NUMBER2:$'
PRODUCT     DB     'PRODUCT IS:$'
TEMP_VAR  DW0
TEMP     DW    0
NUMBER      DW    0
DATA_SEG  ENDS
;-------------------------------
STACK SEGMENT STACK
    DB 64 DUP('STACK')
STACK ENDS
;-----------------------------
CODE_SEG SEGMENT PUBLIC
    ASSUME     CS:CODE_SEG,DS:DATA_SEG,SS:STACK
START: MOV     AX,DATA_SEG        ;khoi tao thanh ghi DX
    MOV     DS,AX
    MOV     AH,09              ;yeu cau nhap
    MOV     DX,OFFSET DATA_1
    INT 21H
    CALL    CR_LF
    MOV     AH,09              ; so thu 1
    MOV     DX,OFFSET DATA_2
    INT 21H
```

```
    CALL    PRINT_CHAR
    CMP     AX,99
    JA  EXIT
    MOV     TEMP_VAR,AX
    CALL    CR_LF
    MOV     AH,09           ; so thu 2
    MOV     DX,OFFSET DATA_3
    INT 21H
    CALL    PRINT_CHAR
    CMP     AX,99
    JA  EXIT
    CALL    CR_LF
    MUL     NUMBER          ;AX:=AX*NUMBER
    PUSH    AX
    MOV     AH,09
    MOV     DX,OFFSET PRODUCT
    INT 21H
    POP     AX
    CALL    WRITE_CHAR
EXIT:   MOV     AH,4CH
    INT 21H
CODE_SEG ENDS
END START
;-----------------------------------
CR_LF PROC FAR
    PUSH    AX
    PUSH    DX
    MOV     AH,02
    MOV     DX,0AH
    INT 21H
    MOV     DX,0DH
    INT 21H
    POP     DX
    POP     AX
    RET
CR-LF ENDP
;-----------------------------------
GET_IN_NUMBER PROC
    PUSH    DX
NHAY:  MOV     AH,08
    INT 21H
    CMP     AL,0DH
    JE  EXIT_1
    CMP     AL,30H
    JB  NHAY
    CMP     AL,39H
    JA  NHAY
    MOV     DL,AL
    MOV     AH,02
    INT 21H
EXIT_1: POP     DX
    MOV     AX,4CH
    INT 21H
    RET
GET_IN_NUMBER ENDP
;-----------------------------------
PRINT_CHAR PROC NEAR
    PUSH    DX
    PUSH    BX
    MOV     TEMP,0
    MOV     CX,1
LOOP_1:     CALL    GET_IN_NUMBER
    CMP     AL,0DH
    JE  EXIT_2
    SUB     AL,30H
    MOV     DX,AX
    XOR     AH,AH
    MOV     AX,TEMP
    CMP     CX,2
    JB  NONE_ZERO
    MOV     BX,10
    MUL     BX
NONE_ZERO:
    ADD     AX,DX
    MOV     TEMP,AX
    INC CX
    CMP     CX,2
    JA  EXIT_2
    JMPLOOP_1
EXIT_2: MOV     AX,TAM
    POP     BX
    POP     DX
    RET
PRINT_CHAR ENDP
;----------------------------
WRITE_CHAR PROC NEAR
    PUSH    BX
    PUSH    CX
    XOR     DX,DX
    MOV     BX,10
    MOV     CX,1
LOOP_2:
    DIV BX
    PUSH    DX
    INC CX
    OR  AX,AX
    JNZ LOOP_2
    JE  PRINT
PRINT: POP      DX
    ADD     DX,30H
    MOV     AH,02H
    INT 21H
    LOOP    PRINT
    RET
WRITE_CHAR ENDP
;----------------------------
```

Bài tập 7:

Tính tổng hai số nhập từ bàn phím

```
CODE_SEG SEGMENT        BYTE  PUBLIC
ASSUME          CS:CODE_SEG,DS:CODE_SEG
        ORG  100H
START:      JMP      FIRST
MSG         DB 'NHAP VAO 2 SO DE CONG :$'
MSG1        DB  'SO THU NHAT :$'
MSG2        DB  'SO THU HAI :$'
MSG3        DB  'TONG CUA CHUNG LA :$'
NUMBER1         DB 0
NUMBER2         DB 0

soam db ?
dauso1 db ?
dauso2 db ?

FIRST:              MOV AH,09H
              MOV DX,OFFSET MSG
              INT 21H
                    call cr_lf
                    MOV AH,09H
                  MOV DX,OFFSET MSG1
                INT 21H

              mov soam,0
                    mov dauso1,0
                    CALL GET_AN_INT_NUM
              cmp soam,1
                    jne so1khongam
                    mov dauso1,1
          so1khongam:
                    CMP AX,255
              Jb  tieptuclam
                    int 20h

        tieptuclam:     MOV NUMBER1,AL
              CALL CR_LF

              MOV AH,09H
                MOV DX,OFFSET MSG2
                INT 21H

              mov soam,0h
                    mov dauso2,0
                    CALL GET_AN_INT_NUM
                    cmp soam,1
                    jne so2khongam
                     mov dauso2,1
          so2khongam:
              CMP AX,255
              JA EXIT
              MOV NUMBER2,AL
              CALL CR_LF
```

```
            MOV AH,09
            MOV DX,OFFSET MSG3
            INT 21H

;-------------------------------------------------------------
                mov    cl,dauso1
                add    cl,dauso2
                cmp    cl,1

                je     khacdau        ;HAI SO KHAC DAU

                XOR    AX,AX
                MOV    BL,NUMBER1
                ADD    AL,BL
                PUSH   AX
                cmp    dauso1,1
                jne    khongam
                call   indau
                jmp    khongam
;-------------------------------------------------------------
          khacdau:   mov    cl,number1
                cmp    cl,number2  ;SO1>SO2 ?
                je     writeZero
                ja     laydauso1
;-----------------------------------------------
                XOR    AX,AX
                MOV    BL,NUMBER1
                SUB    AL,BL
                PUSH AX

                cmp    dauso2,1
                jne    khongam
                CALL   INDAU
                JMP    KHONGAM

          laydauso1:  XOR   AX,AX
                MOV   AL,NUMBER1
                SUB   AL,NUMBER2
                PUSH  AX
                cmp    dauso1,1
                jne    khongam
                CALL   INDAU

          khongam:

              POP AX

                CALL WRITE_INT_NUMBER
                jmp exit
          writezero:   mov ax,0
                call write_int_number

      EXIT:
                INT 20
;-----------------------------------------------

;-----------------------------------------------
 indau proc
     push ax
     push dx
     mov ah,02
     mov dl,'-'
     int 21h
     pop dx
     pop ax
     ret
 indau endp
;-----------------------------------------------
GET_AN_INT_NUM  PROC
          JMP  $+4
TEMP_VAR    DW   0
          PUSH BX
          PUSH CX
          PUSH DX
          XOR  DX,DX
          MOV TEMP_VAR,0
          MOV CX,1
          mov soam,0h
LOOP_2:     CALL  GET_A_DEC_DIGIT
            CMP AL,0DH
            JE EXIT_2
          ;---------------------------------
                  cmp al,'-'
                   jne tieptuc
                    mov soam,1h
                   jmp  loop_2
;-----------------------------------------------------
           tieptuc:  SUB AL,30H
              XOR AH,AH
              MOV DX,AX
              MOV AX,TEMP_VAR
              CMP CX,1
              JE SUM_UP
              MOV BL,10
              PUSH DX
              MUL  BL
              POP DX
SUM_UP:         ADD AX,DX
              MOV TEMP_VAR,AX
              INC CX
              CMP CX,3
              JA   EXIT_2
              JMP  LOOP_2
EXIT_2:       MOV  AX,TEMP_VAR
              POP DX
              POP CX
              POP BX
              RET
GET_AN_INT_NUM   ENDP
;-----------------------------------------------
;
;-----------------------------------------------
GET_A_DEC_DIGIT  PROC
LOOP_1:
          PUSH  DX
          MOV  AH,08H
          INT  21H
             CMP AL,0DH
             JE  EXIT_1
           ;-------------------------
                CMP AL,'-'
                JNE TIEP
                JMP INSO
             ;-------------------------
         TIEP:     CMP AL,30H
             JB  LOOP_1
             CMP AL,39H
             JA  LOOP_1


        INSO:     MOV DL,AL
             MOV AH,02
             INT  21H
EXIT_1:    POP DX
             RET
GET_A_DEC_DIGIT ENDP
;-----------------------------------------------
;
;-----------------------------------------------
WRITE_INT_NUMBER  PROC NEAR
           MOV  BX,10
           XOR  CX,CX
NONE_ZERO:
           XOR  DX,DX
           DIV  BX
           PUSH  DX
           INC  CX
           OR  AX,AX
           JNZ  NONE_ZERO
WRITE_DIGIT_LOOP:
          POP DX
          ADD DL,48  ;=30H='0'
          MOV  AH,02
          INT 21H
          LOOP WRITE_DIGIT_LOOP
          RET
WRITE_INT_NUMBER   ENDP
;-----------------------------------
```

```
;
;---------------------------------
CR_LF   PROC NEAR
        PUSH  AX
        PUSH  DX
        MOV AH,02
        MOV DL,0DH
        INT 21h
        MOV DL,0AH
        INT 21H
        POP DX
        POP AX
        RET
CR_LF   ENDP
;---------------------------------
CODE_SEG   ENDS
END        START
```

Bài tập 8:

Chương Trình xác định số cổng COM và địa chỉ cổng COM

```
hien_string MACRO xau
       push ax dx
       mov dx,offset xau
       mov ah,9
       int 21h
       pop dx ax
ENDM
;---------------------------------
 _STACK SEGMENT  STACK 'STACK'
    db 100h dup(?)
_STACK ENDS
data segment
   m1   db         'Khong co cong COM. $'
   m2   db  0ah,0dh,'So luong cong COM la: $'
   m3   db  0ah,0dh,'Dia chi cong COM1 la: $'
data ends
code segment
  assume cs:code,ds:data,ss:_STACK
   ps:
      mov ax,data
      mov ds,ax

      mov ax,40h
      mov es,ax
      mov bx,11h
      mov al,es:[bx]
      and al,0eh ; lay 3 bit chua so luong cong COM (0 0 0 0 | x x x 0)
               ;                                  0       e
      jnz l1
       hien_string m1
      jmp stop
    l1:
     hien_string m2
     shr al,1
     add al,30h
     mov ah,0eh
     int 10h
     hien_string m3
     mov bx,2          ; cong COM 2
     mov ax,es:[bx]
     push ax
     mov al,ah
     call HIENHEX
     pop ax
     call HIENHEX
  stop:
     mov ah,1
     int 21h
     mov ah,4ch
     int 21h
; chuong trinh con HIENHEX va trong CTC nay lai chua CTC HIEN
 HIENHEX PROC
   push ax cx
   push ax
   mov cl,4
   shr al,cl
   call HIEN
   pop ax
   and al,0fh
   call HIEN
   pop cx ax
    ret
 HIENHEX ENDP
 HIEN PROC
  cmp al,10
  jl H
  add al,7
 H:
  add al,30h
  mov ah,0eh
  int 10h
  ret
 HIEN ENDP
  code ends
end ps
```

Bài tập 9:

Hiển thị tên ổ đĩa và thời gian đọc đĩa

```
COMMENT *
   PROGRAM DISKLITE
   chuong trinh se hien thi ten o dia va thoi gian doc dia
   moi khi co truy nhap dia
   Sudung:
      DISKLITE -> chay chuong trinh
      DISKLITE /U -> unload disklite*
CODE   SEGMENT
       ASSUME CS:CODE,DS:CODE
       ORG 100h
START:
       JMP    INIT    ;nhay toi thu tuc khoi tao
MAGIC_CODE  DB  'DISKLITE VERSION 1.0'
MAGIC_LEN   LABEL BYTE
NUM_IN      EQU 11 ;so chu so de in
DISPLAY_BASE DW 0B800h
OLD_CHARS      DB NUM_IN*2 DUP(?)
DISPLAY_DRV   DB 'A',70h,':',70h,' ',70h ;in ten o dia
DISPLAY_TM    DB '0',70h,'0',70h,':',70h,'0',70H,'0',70h,':',70h
              DB  2 DUP('0',70h)
NUM_FLOPPIES   DB ?
SECOND         DB 0
MINUTE         DB 0
HOUR           DB 0
TICKER         DB 0 ;so nhip dong ho
D_DISK         EQU  (80-NUM_IN-1)*2 ;offset de ghi ten o dia
D_TIME         EQU  (82-NUM_IN)*2  ;offset ghi thoi gian
;dia chi byte trang thai moto o mem
MOTOSTATUS  EQU 43Fh
;dia chi cong dia cung
HARDPORT EQU 1F7h
;dia chi co dia cung
HARDFLAGS EQU 48Ch   ;(Neu flags and 8)=8 thi dang roi
;dia chi co IN_DOS
DAPTR EQU  THIS DWORD
DAPTR_OFS  DW ?
DAPTR_SEG  DW ?
;cac thuc tuc ngat cu
OLDINT13_PTR EQU THIS DWORD
OLD_INT13   DW ? ;dia chi ngat 13H
            DW ?
OLDINT1C_PTR EQU THIS DWORD
OLD_INT1C   DW ? ;dia chi ngat 1C
            DW ?
INT13 PROC FAR
      ASSUME CS:CODE,DS:NOTHING
      PUSHF      ;luu thanh ghi co
      PUSH   AX
      PUSH   CX
      PUSH   DX
      PUSH   SI
      PUSH   DI
      PUSH   DS
      PUSH   ES
      CALL   GET_DISPLAY_BASE   ;tinh dia chi doan bo nho man hinh
      CALL   SAVE_SCREEN        ;Luu 11 ky tu
      CALL   DISPLAY_DRIVE
      CALL   DISPLAY_TIME
      POP    ES
      POP    DS
      POP    DI
      POP    SI
      POP    DX
      POP    CX
```

```
    POP   AX
    POPF
    PUSHF
    CALL  DWORD PTR CS:OLD_INT13
    PUSHF
    PUSH  AX
    PUSH  CX
    PUSH  SI
    PUSH  DI
    PUSH  DS
    PUSH  ES
    LEA   SI,OLD_CHARS
    MOV   DI,D_DISK
    MOV   CX,NUM_IN
    CALL  WRITE_S
    POP   ES
    POP   DS
    POP   DI
    POP   SI
    POP   CX
    POP   AX
    POPF
    RET   2
INT13  ENDP
INT1C  PROC FAR
    ASSUME CS:CODE,DS:NOTHING
    PUSH  AX
    ;PUSH  CX
    PUSH  DX
    PUSH  DI
    ;PUSH  SI
    PUSH  DS
    ;PUSH  ES
    ;LDS   DI,[DAPTR] ;nap IN_DOS vao DS:DI
    ;CMP   BYTE PTR DI,0 ;co DOS co ban khong
    XOR   AX,AX
    MOV   DS,AX
    MOV   AL,BYTE PTR DS:[MOTOSTATUS] ;co o dia nao quay khong
    AND   AL,3
    CMP   AL,0
    JNE   CONTINUE      ;neu ban thi tiep tuc
    ;MOV   DX,HARDPORT  ;cong dia cung chua byte so 7 la co bao
ban
    ;IN    AL,DX         ;kiem tra cong dia cung
    ;SHR   AL,7          ;kiem tra bit 7
    ;CMP   AL,1          ;neu ban
    MOV   AL,BYTE PTR DS:[HARDFLAGS]  ;kiem tra co dia cung
    AND   AL,8
    CMP   AL,8          ;neu co=8 la roi
    JNE   CONTINUE      ;khong thi tiep tuc
    XOR   AL,AL
    MOV   SECOND,AL
    MOV   MINUTE,AL
    MOV   HOUR,AL
    MOV   TICKER,AL
    JMP   NOT_INC
CONTINUE:
    XOR   DL,DL
    INC   TICKER
    MOV   AL,TICKER
    CMP   AL,18     ;so nhip 18.2 lan trong mot giay
    JB    NOT_INC   ;neu Chua bang thi in ra man hinh
    MOV   TICKER,DL ;neu qua thi dat lai ticker=0
    INC   SECOND    ;tang giay
    MOV   AL,SECOND
    CMP   AL,60     ;neu qua 60 giay thi tang phut
    JB    NOT_INC
    MOV   SECOND,DL
    INC   MINUTE    ;tang phut
    MOV   AL,MINUTE
    CMP   AL,60
    JB    NOT_INC
    MOV   MINUTE,DL
    INC   HOUR
NOT_INC:
    ;CALL  DISPLAY_TIME  ;thu
    ;POP   ES
    POP   DS
    ;POP   SI
    POP   DI
    POP   DX
    ;POP   CX
    POP   AX
    JMP   DWORD PTR CS:OLD_INT1C
INT1C  ENDP
;thu tuc get_display_base xac dinh doan bo nho man hinh
; thay doi AX
GET_DISPLAY_BASE PROC NEAR
    INT    11h  ;lay co thiet bi
    AND    AX,30h
    CMP    AX,30h  ;man hinh don sac
    MOV    AX,0B800h
    JNE    GET_BASE
    MOV    AX,0B000h
GET_BASE:
    MOV    DISPLAY_BASE,AX
    RET
GET_DISPLAY_BASE ENDP
;thu tuc savescreen luu manh hinh lai
;thay doi AX,si,di,ds,es,cx
SAVE_SCREEN PROC NEAR
   MOV     SI,D_DISK  ;lay dia chi bo nho man hinh
   MOV     DI,OFFSET OLD_CHARS
   MOV     AX,DISPLAY_BASE
   MOV     DS,AX         ;ky tu man hinh nam tai DS:SI
   MOV     AX,CS
   MOV     ES,AX         ;old_chars nam tai ES:DI
   MOV     CX,NUM_IN
   REP     MOVSW
   RET
SAVE_SCREEN ENDP
;thu tuc display_drive ghi ten o dia
;thay doi AX,SI,CX,DI
DISPLAY_DRIVE PROC NEAR
   MOV   AL,DL
   CMP   AL,80h    ;co phai o cung khong
   JB    DISPLAY   ;khong thi tiep tuc
   SUB   AL,80h    ;khong thi tru 80h
   ADD   AL,NUM_FLOPPIES ;cong voi so o dia
DISPLAY:
   ADD   AL,'A'
   LEA   SI,DISPLAY_DRV
   MOV   CS:[SI],AL
   MOV   CX,3
   MOV   DI,D_DISK   ;offset in ten dia
   CALL  WRITE_S
   RET
DISPLAY_DRIVE ENDP
;thu tuc display_time in so gio
;thay doi AX,CX,SI,DX
DISPLAY_TIME PROC NEAR
   LEA   SI,DISPLAY_TM ;dia chi cua gio de in
   MOV   DL,'0'
   MOV   CS:[SI],DL
   MOV   AL,HOUR
   XOR   AH,AH
   CMP   AX,10   ;gio co lon hon muoi khong
   JB    LESS_H
   MOV   CL,10
   DIV   CL       ;ket qua trong AL,so du trong AH
   ADD   AL,'0'
   MOV   CS:[SI],AL
   MOV   AL,AH    ;lay so du trong AH
LESS_H:
   INC   SI
   INC   SI
   ADD   AL,'0'
   MOV   CS:[SI],AL
   ADD   SI,4      ;dat SI vao offset cua minute
   MOV   CS:[SI],DL ;dat truoc hang chuc=0
   MOV   AL, MINUTE
   XOR   AH,AH
   CMP   AX,10   ; phut co lon hon 10 khong
   JB    LESS_M
   MOV   CL,10
   DIV   CL       ;ket qua trong AL,so du trong AH
   ADD   AL,'0'
   MOV   CS:[SI],AL
   MOV   AL,AH    ;lay so du trong AH
LESS_M:
```

```
   INC   SI
   INC   SI
   ADD   AL,'0'
   MOV   CS:[SI],AL
   ADD   SI,4      ;dat SI vao offset cua second
   MOV   CS:[SI],DL
   MOV   AL,SECOND
   XOR   AH,AH
   CMP   AX,10   ; giay co lon hon 10 khong
   JB    LESS_S
   MOV   CL,10
   DIV   CL       ;ket qua trong AL,so du trong AH
   ADD   AL,'0'
   MOV   CS:[SI],AL
   MOV   AL,AH    ;lay so du trong AH
LESS_S:
   INC   SI
   INC   SI
   ADD   AL,'0'
   MOV   CS:[SI],AL
   LEA   SI,DISPLAY_TM
   MOV   CX,NUM_IN-3
   MOV   DI, D_TIME
   CALL  WRITE_S
   RET
DISPLAY_TIME ENDP
;Thu tuc write_s in chuoi ra man hinh
;thay doi AX,ES,DS
WRITE_S  PROC NEAR
   MOV  AX,DISPLAY_BASE
   MOV  ES,AX   ;dia chi man hinh ES:DI
   MOV  AX,CS
   MOV  DS,AX   ;dia chi display_tm tai DS:SI
   REP  MOVSW
   RET
WRITE_S  ENDP
 ;bat dau khoi tao
INIT:        ;bat dau thuong tru
OLDPSP   DW ?
BLOCMCB EQU 'M'  ; bao hieu chua het MCB
LASTMCB EQU 'Z'  ; da het MCB
MCB        STRUC   ;cau truc cua MCB
 IDCODE     DB ?
 PSP        DW ?
 SIZE       DW ?
MCB         ENDS

        ;kiem tra xem da resident chua
        MOV    AH,51h
        INT    21h       ;lay PSP cua chuong trinh,gia tri cho trong BX
        MOV    DX,BX     ; giu gia tri  vao DX

        ;lay dia chi MCB dau
        MOV     AH,52h
        INT     21h    ;ES:BX-2 tro toi dia chi cua MCB dau tien
        SUB     BX,2
        MOV     AX,ES:[BX]
        MOV     ES,AX
        XOR     BX,BX ;ES:BX la MCB dau
        MOV    DI,OFFSET MAGIC_CODE ;ES:DI se chua Magic_code
neu da thuong tru
FIND_CODE:       ;Tim xem da thuong tru chua
        CMP    DX,ES:[BX].PSP       ;xem PSP(MCB) co bang PSP(PROG)
        JE     TIEP                 ;co thi nhay sang MCB khac
        MOV AX,DS ;dia chi cua chuong trinh thuong tru
        SUB    AX,DX
        ADD    AX,ES:[BX].PSP ;addr=PSP(MCB)+(segment(PROG)-
PSP(PROG))
        PUSH   ES            ;cat ES
        MOV    ES,AX         ;ES se chua doan chuong trinh thuong tru
neu da resident
        PUSH   DI                ;cat DI
        MOV    SI,DI             ;DS:SI se chua magic_code cua chuong
trinh
        MOV    CX,OFFSET MAGIC_LEN-OFFSET MAGIC_CODE  ;tinh
chieu dai chuoi
        REPE   CMPSB             ;kiem tra xem hai chuoi co bang nhau
        JCXZ   YET_INST          ;da co thuong tru ,ket thuc chuong
trinh
        POP    DI                ;tra lai DI
        POP    ES                ;tra lai ES
TIEP:
        CMP    ES:[BX].IDCODE,LASTMCB ;da het MCB chua
        JE     NOT_INST
        MOV    AX,ES             ;khong thi
        ADD    AX,ES:[BX].SIZE       ;cong ES voi SIZE+1 cua MCB
        INC    AX
        MOV    ES,AX             ;ES:BX la MCB ke tiep
        JMP    FIND_CODE

NOT_INST: ;neu chua TSR thi khoi tao
        CALL  GET_NUM_DISK

        ;xac dinh dia chi co in_dos
        MOV   AH,34h
        INT   21h
        MOV   CS:DAPTR_OFS,BX  ;offset cua co DOS
        MOV   CS:DAPTR_SEG,ES  ;segment cua co DOS

       ;giai phong khoi moi truong truoc khi TSR

        MOV   ES,DX ;ES doan cua PSP cua chuong trinh se thuong tru
        XOR  BX,BX     ;ES:BX  chua dia chi PSP
        MOV  AX,ES:[BX].ENVSEG  ;nap dia chi moi truong vao AX
        MOV  ES,AX
        MOV  AH,49h
        INT  21h
        ;lay int 13
        MOV   AX,3513h
        INT   21h
        MOV   CS:OLD_INT13,BX
        MOV   CS:OLD_INT13[2],ES
        ;dat int 13
        MOV   AX,2513h
        PUSH  CS
        POP   DS
        MOV   DX,OFFSET INT13
        INT   21h
        ;lay int 1C
        MOV   AX,351Ch
        INT   21h
        MOV   CS:OLD_INT1C,BX
        MOV   CS:OLD_INT1C[2],ES
        ;dat int 1C
        MOV   AX,251Ch
        PUSH  CS
        POP   DS
        MOV   DX,OFFSET INT1C
        INT   21h
        MOV   DX,OFFSET INIT_TSR
        CALL  WRITE_MSG
        ;ket thuc va noi tru
        MOV   DX,OFFSET INIT
        INT   27h

YET_INST: ;da TSR roi thi kiem tra va ket thuc chuong trinh
        POP    DI
        POP    ES
        MOV    AX,ES:[BX].PSP
        MOV    OLDPSP,AX   ;nap PSP cua TSR vao OLDPSP
        CALL   READPARAM
        INT    20h

GET_NUM_DISK   PROC NEAR
        PUSH  AX
        PUSH  CX
        INT   11h
        MOV   CL,6
        SHR   AX,CL
        AND   AL,3
        INC   AL
        CMP   AL,1   ;neu la 1 o dia
        JA    GET_F  ;thi ket thuc
        MOV   AL,2   ;co hai o
GET_F:
        MOV   NUM_FLOPPIES,AL
        POP   CX
        POP   AX
        RET
GET_NUM_DISK  ENDP
```

```
READPARAM       PROC NEAR
        MOV     SI,80h          ;Dia chi duoi dong lenh
        MOV     CL,[SI]         ;so do chieu dai duoi lenh
        JCXZ    NO_PARAM        ;neu khong co thi bao loi va ket thuc
READ_PARAM:                     ;khong thi lap voi CX=so tham so
        INC     SI              ;tham so dau tien
        MOV     BL,[SI]
        CMP     BL,' '
        JE      CON_READ        ;neu la ky tu trang thi doc tiep
        CMP     BL,'/'          ;co dung lenh khong
        JZ      CON_READ        ;dung thi tiep tuc
        CMP     BL,'U'          ;lenh vao la /U thi unload TSR
        JZ      REMOVE          ;dung thi loai TSR ra khoi bo nho
        CMP     BL,'u'
        JNZ     W_PARAM
REMOVE:
        CALL    UNLOAD          ;dung thi unload TSR
        RET
CON_READ:
        LOOP    READ_PARAM
NO_PARAM:
        MOV     DX,OFFSET YET_TSR
        CALL    WRITE_MSG       ;khong co tham so thi in thong bao
        RET
W_PARAM:
        MOV     DX,OFFSET WRONG_PARAM
        CALL    WRITE_MSG
        RET
READPARAM       ENDP

UNLOAD    PROC NEAR     ;thu tuc de loai TSR ra khoi bo nho
PREFIX       STRUC      ;cau truc cua PSP
  DUMMY  DB 2Ch DUP(?)
  ENVSEG DW ?           ;dia chi cua moi truong
PREFIX       ENDS
        ;lay dia chi cua khoi moi truong
        MOV  DX,OLDPSP ;nap OLDPSP vao DX
        MOV  AX,351Ch  ;kiem tra xem
        INT  21h       ;ngat 1CH co bi thay doi khong
        MOV  AX,ES     ;AX chua dia chi cua ngat 1CH
        CMP  AX,DX
        JNE  CHANGED   ;neu khong bang thi ket thuc

        MOV  AX,3513h  ;kiem tra xem
        INT  21h       ;ngat 13H co bi thay doi khong
        MOV  AX,ES     ;AX chua dia chi cua ngat 13H
        CMP  AX,DX
        JNE  CHANGED   ;neu khong bang thi ket thuc
BEGIN_UNLOAD:           ;neu khong thay doi thi loai TSR
        ;giai phong bo nho ,luc nay ES da chua segment PSP cua TSR
        ;PUSH ES
        MOV  AH,49h
        ;POP  ES        ;ES la dia chi doan cua TSR
        INT  21h
        ;dat lai ngat cu
        PUSH  DS        ;cat DS
        CLI
        MOV  AX,2513h   ;dat lai ngat 13
        LDS  DX,ES:OLDINT13_PTR
        INT  21h

        MOV  AX,251Ch   ;dat lai ngat 1c
        LDS  DX,ES:OLDINT1C_PTR
        INT  21h

        STI
        POP  DS                 ;tra lai DS cu
        MOV  DX,OFFSET SUCCESS  ;da loai ra khoi bo nho
        CALL WRITE_MSG
        RET
CHANGED:                ;vec to ngat da bi doi
        MOV  DX,OFFSET UN_SUCCESS
        CALL WRITE_MSG
        RET
UNLOAD    ENDP

WRITE_MSG   PROC NEAR
        PUSH AX
        MOV  AH,9
        INT  21h
        POP  AX
        RET
WRITE_MSG   ENDP
;khai  bao cac chuoi de thong bao
INIT_TSR    DB 'DISKLITE VERSION 1.0',13,10
            DB '   COPYRIGHT (C) LE ANH TUAN 1994$'
YET_TSR     DB 'CHUONG TRINH DA THUONG TRU$'
WRONG_PARAM DB 'SAI THAM SO$'
SUCCESS     DB 'CHUONG TRINH DA DUOC UNLOAD$'
UN_SUCCESS  DB 'KHONG THE UNLOAD DUOC CHUONG TRINH$'
CODE     ENDS
END      START
```

Bài tập 10:

Chương trình sửa bad track 0 của đĩa mềm

```
 .model tiny
 .code
org 100h
start:
jmp INIT
Oldint13h label dword
Int13hofs dw    ?
Int13hseg dw    ?
Make   db       1

NEWINT13H:
push es ds si di ax
push cs cs
pop ds es
cmp ax,0FEFEh
je ALREADY
lea di,Make
cmp ax,0BABAh
je EQUAL0
cmp ax,0ABABh
je EQUAL1
jmp CONTINUE

ALREADY:
pop ax di si ds es
mov ax,0EFEFh
iret
EQUAL0:
xor al,al
stosb
pop ax di si ds es
iret
EQUAL1:
mov al,1
stosb
pop ax di si ds es
iret

CONTINUE:
lea si, Make
lodsb
cmp al,0
pop ax di si ds es
je PASSBY
cmp ah,2
je READ
cmp ah,3
je READ
PASSBY:
jmp cs:Oldint13h

READ:
cmp dl,0
je GOON
cmp dl,1
jne PASSBY
GOON:
cmp ch,0
je TRACK0
cmp ch,79
je TRACK79
jmp PASSBY

TRACK0:
```

```
add ch,79
jmp PASSBY

TRACK79:
cmp ah,3
jne COMEBACK
mov ah,1
stc
COMEBACK:
Iret

INIT:
mov si,80h
lodsb
push ax
mov ax,0FEFEh
int 13h
cmp ax,0EFEFh
pop ax
jne INSTALL
cmp al,1
jbe CHANGETRACK
mov si,82h
lodsw
cmp ax,0752Fh
je NORMAL
cmp ax,0552Fh
je NORMAL
lea dx,Error
call WRITE
ret
CHANGETRACK:
mov ax,0ABABh
int 13h
lea dx,Mess2
call WRITE
ret
NORMAL:
mov ax,0BABAh
int 13h
lea dx,Mess1
call WRITE
ret

INSTALL:
mov ax,3513h
int 21h
mov int13hofs,bx
mov int13hseg,es
lea dx,newint13h
mov ax,2513h
int 21h
lea dx,Mess
call WRITE
lea dx,Mess2
Call WRITE
lea dx,INIT
int 27h

WRITE:
mov ah,9
int 21h
ret

Mess db 'Program repares bad track0 disk.',13,10,'Written by Hoang
Tuan Dat.',13,10,'Finish install.',13,10,'$'
Mess1 db 13,10,'Now you can not read bad track0 disk',13,10,'$'
Mess2 db 13,10,'Now you only can read bad track0 disk',13,10,'$'
Error db 'Input valid',13,10,'$'
end Start
```

Chương 2: LẬP TRÌNH HỢP NGỮ TRÊN VI ĐIỀU KHIỂN MCS-51

Chương này giới thiệu cách thức lập trình trên MCS-51 cũng như giải thích hoạt động của các lệnh sử dụng cho họ MCS-51.

Các ký hiệu cần chú ý:

Rn : các thanh ghi từ R0 – R7 (bank thanh ghi hiện hành)
Ri : các thanh ghi từ R0 – R1 (bank thanh ghi hiện hành)
@Rn : định địa chỉ gián tiếp 8 bit dùng thanh ghi Rn
@DPTR : định địa chỉ gián tiếp 16 bit dùng thanh ghi DPTR
direct : định địa chỉ trực tiếp RAM nội (00h – 7Fh) hay SFR (80h – FFh)
(direct) : nội dung của bộ nhớ tại địa chỉ direct
#data8 : giá trị tức thời 8 bit
#data16 : giá trị tức thời 16 bit
bit : địa chỉ bit của các ô nhớ có thể định địa chỉ bit (00h – 7Fh đối với địa chỉ bit và 20h – 2Fh đối với địa chỉ byte)

1. Các phương pháp định địa chỉ

❖ Định địa chỉ trực tiếp

Định địa chỉ trực tiếp chỉ dùng cho các thanh ghi chức năng đặc biệt và RAM nội của 8951. Giá trị địa chỉ trực tiếp 8 bit được thêm vào phía sau mã lệnh. Nếu địa chỉ trực tiếp từ 00h – 7Fh thì đó là RAM nội của 8951 (128 byte), còn địa chỉ từ 80h – FFh là địa chỉ các thanh ghi chức năng đặc biệt (xem bảng 1.2, chương 1).

Các lệnh sau có kiểu định địa chỉ trực tiếp:

```
MOV A, P0
MOV A, 30h
```

Lệnh đầu tiên chuyển nội dung từ Port 0 vào thanh ghi A. Khi biên dịch, chương trình sẽ thay thế từ gợi nhớ P0 bằng địa chỉ trực tiếp của Port 0 (80h) và đưa vào byte 2 của mã lệnh. Lệnh thứ hai chuyển nội dung của RAM nội có địa chỉ 30h vào thanh ghi A.

❖ Định địa chỉ gián tiếp

Định địa chỉ gián tiếp có thể dùng cho cả RAM nội và RAM ngoại. Trong chế độ này, địa chỉ của RAM xác định thông qua một thanh ghi (R0, R1, SP cho địa chỉ 8 bit và DPTR cho địa chỉ 16 bit). Các lệnh sau có kiểu địa chỉ gián tiếp:

```
MOV A, @R0
```

```
MOVX A, @DPTR
```

Lệnh đầu tiên chuyển nội dung cúa RAM nội có địa chỉ chứa trong thanh ghi R0 vào thanh ghi A (giả sử R0 = 30h thì chuyển nội dung của ô nhớ 30h). Lệnh thứ hai chuyển nội dung RAM ngoại vào thanh ghi A (địa chỉ RAM chứa trong DPTR).

❖ **Định địa chỉ thanh ghi**

Các thanh ghi từ R0 – R7 có thể truy xuất bằng cách định địa chỉ trực tiếp hay gián tiếp như trên. Ngoài ra, các thanh ghi này còn có thể truy xuất bằng cách dùng 3 bit trong mã lệnh để chọn 1 trong 8 thanh ghi (8 thanh ghi này có địa chỉ trực tiếp thay đổi tuỳ theo bank thanh ghi đang sử dụng).

❖ **Định địa chỉ tức thời**

Giá trị của một hằng số có thể đưa trực tiếp vào mã lệnh của chương trình. Trong hợp ngữ, hằng số được xác định bằng cách sử dụng dấu #.

Lệnh:

```
MOV A, #10h
```

có chế độ địa chỉ tức thời.

❖ **Định địa chỉ chỉ số**

Quá trình định địa chỉ chỉ số chỉ có thể dùng cho bộ nhớ chương trình, được dùng để đọc dữ liệu trong các bảng tìm kiếm. Chế độ này thường dùng một thanh ghi nền 16 bit (PC hay DPTR) để chỉ vị trí của bảng và thanh ghi A chỉ vị trí của các phần tử trong bảng.

2. Các vấn đề liên quan khi lập trình hợp ngữ

2.1. Cú pháp lệnh

Một lệnh trong chương trình hợp ngữ có dạng như sau:

Nhãn	**Lệnh**	**Toán hạng**	**Chú thích**
A:	MOV	A, #10h	; Đưa giá trị 10h vào thanh ghi A
LED	EQU	30h	; Định nghĩa ô nhớ chứa mã led
On_Led	BIT	00h	; Cờ trạng thái led

Trường nhãn định nghĩa các ký hiệu (có thể là địa chỉ trong chương trình, các hằng dữ liệu, tên đoạn hay các cấu trúc lập trình). Trường nhãn không bắt đầu bằng số và không trùng với các từ khoá có sẵn.

Trường lệnh chứa các từ gợi nhớ cho các lệnh của MCS-51 hay các lệnh giả dùng cho chương trình dịch.

Trường toán hạng chứa các thông số liên quan đến lệnh đang sử dụng.

Trường chú thích dùng để ghi chú trong chương trình hợp ngữ. Trường này phải được bắt đầu bằng dấu **;** và chương trình dịch sẽ bỏ qua các từ đặt sau dấu **;**.

Lưu ý rằng các chương trình dịch không phân biệt chữ hoa và chữ thường.

2.2. Khai báo dữ liệu

- Khi khai báo hằng số, chữ **h** cuối cùng xác định hằng số là số thập lục phân; chữ **b** cuối cùng xác định số nhị phân và chữ **d** cuối (hay không có) xác định số thập phân. Lưu ý rằng đối với số thập lục phân, khi bắt đầu bằng chữ A $\rightarrow$ F thì phải thêm số 0 vào phía trước.

Ví dụ:

1010b ; Số nhị phân

1010h ; Số thập lục phân

1010 ; Số thập phân

0F0h ; Số thập lục phân nhưng bắt đầu bằng chữ F nên phải thêm vào phía trước số 0.

- Khi dùng dấu # phía trước một con số, đó chính là dữ liệu tức thời còn nếu không dùng dấu # thì đó là địa chỉ của ô nhớ. Lưu ý rằng khi dùng RAM nội thì chỉ dùng địa chỉ từ 00 – 7Fh còn vùng địa chỉ từ 80h – 0FFh dùng cho các thanh ghi chức năng đặc biệt. Đối với họ 89x52, RAM nội có 256 byte thì các byte địa chỉ cao (từ 80h – 0FFh) không thể truy xuất trực tiếp mà phải truy xuất gián tiếp.

Ví dụ:

```
MOV A,30h      ; Chuyển nội dung ô nhớ 30h vào A
MOV A,#30h     ; Chuyển giá trị 30h vào A
MOV A,80h      ; Chuyển nội dung Port 0 vào A (80h là
               ; địa chỉ Port 0
MOV R0,#80h    ; Chuyển nội dung ô nhớ 80h vào A (chỉ
MOV A,@R0      ; dùng cho họ 89x52)
```

- Để định nghĩa trước một vùng nhớ trong bộ nhớ chương trình, có thể dùng các chỉ dẫn **DB** (define byte – định nghĩa 1 byte) hay **DW** (define word – định nghĩa 2 byte).

Ví dụ: Định nghĩa trước dữ liệu cho led như sau:

```
Led: DB   01h,02h,04h,08h,10h,20h,40h,80h
```

Đoạn chương trình này xác định tại nhãn Led có chứa các giá trị lần lượt từ 01h đến 80h. Nếu nhãn Led đặt tại địa chỉ 100h thì giá trị tương ứng như sau:

Địa chỉ	Giá trị
100h	01h
101h	02h
102h	04h
103h	08h
104h	10h
105h	20h
106h	40h
107h	80h

- Để dễ nhớ và dễ hiểu khi lập trình, các chương trình dịch cho phép dùng các ký tự thay thế cho các ô nhớ bằng các lệnh giả EQU, BIT.

Ví dụ:

```
LED  EQU  30h
ON_LED   BIT  00h
```

Giả sử chương trình hợp ngữ có các lệnh sau:

```
MOV  A,LED
SETB ON_LED
```

Khi biên dịch, chương trình dịch sẽ tự động chuyển thành dạng lệnh sau:

```
MOV  A,30h
SETB 00h
```

2.3. Các toán tử

❖ Các toán tử số học:

Bao gồm các toán tử +, -, *, /, **mod**.

Ví dụ: Các lệnh sau tương đương:

```
MOV A,#12h          MOV A,#10h + 2h
MOV A,#21 mod 2     MOV A,#1
MOV A,#12/4         MOV A,#3
```

❖ **Các toán tử logic:**

Bao gồm các toán tử: **OR**, **AND**, **NOT**, **XOR**.

Ví dụ: Các lệnh sau tương đương:

```
MOV A,#01h          MOV A,#03h AND 91h
MOV A,#-5           MOV A,#NOT 5
MOV A,#24h          MOV A,#20h OR 04h
```

❖ **Các toán tử quan hệ:**

Bao gồm các toán tử: **EQ** (=), **NE** (<>), **LT** (<), **LE** (<=), **GT** (>), **GE** (>=). Lưu ý rằng khí sử dụng các toán tử quan hệ, chỉ có 2 kết quả: sai (= 0) hay đúng (= FFh hay FFFFh tuỳ theo kết quả là 8 bit hay 16 bit).

Ví dụ: Các lệnh sau tương đương:

```
MOV A,#00h          MOV A,#5 EQ 6
MOV A,#0FFh         MOV A,#7 < 9
MOV DPTR,#0FFFFh    MOV DPTR,#5 NE 6
```

❖ **Các toán tử khác:**

Bao gồm các toán tử: **SHR** (dịch phải), **SHL** (dịch trái), **HIGH** (byte cao), **LOW** (byte thấp), **(**, **)**.

Ví dụ: Các lệnh sau tương đương:

```
MOV A,#06h          MOV A,#03h SHL 1
MOV A,#01h          MOV A,#HIGH 0123h
MOV A,#02h          MOV A,#LOW 0102h
```

2.4. Cấu trúc chương trình

- Cấu trúc chương trình hợp ngữ cơ bản mô tả như sau:

```
ORG 0000h      ; Đặt lệnh LJMP main tại địa chỉ
LJMP main      ; 0000h (địa chỉ bắt đầu khi
               ; reset AT89C51)
ORG 0030h      ; Vùng địa chỉ 0003h – 002Fh
Main:          ; dùng để chứa các chương trình
               ; phục vụ ngắt
```

```
…
CALL Subname
…
;---------------
Subname:
…
…
RET
END       ; kết thúc chương trình
```

Các lệnh giả ORG cho biết lệnh phía sau đặt tại vị trí nào trong chương trình. Lưu ý rằng khi khởi động, chương trình trong AT89C51 sẽ được thực thi tại địa chỉ 0000h nên thông thường tại địa chỉ này sẽ có lệnh ***LJMP main*** để xác định chương trình chính sẽ bắt đầu tại nhãn main.

Các dấu **;** xác định đây là một chú thích, chương trình dịch sẽ bỏ qua tất cả các phần nằm sau dấu **;**.

Các địa chỉ từ 0003h – 002Fh phục vụ cho mục đích xử lý ngắt nên không sử dụng. Tuy nhiên, nếu chương trình không cần xử lý ngắt thì cũng có thể sử dụng luôn vùng địa chỉ này.

- Khi thực hiện soạn thảo chương trình hợp ngữ, có thể dùng bất kỳ chương trình soạn thảo không định dạng (như NotePad, Norton Commander, …) và thường lưu file với phần mở rộng .asm, .a51 (tuỳ theo chương trình dịch).
- Sau khi soạn thảo, dùng một chương trình dịch để chuyển từ file văn bản thành file .hex (có thể dùng sim51.exe, oh.exe). Ngoài ra, có nhiều chương trình soạn thảo bao gồm cả chương trình dịch bên trong (xem thêm phần phụ lục).
- Khi dịch ra file .hex, dùng một mạch nạp để nạp file .hex vào AT89C51 (xem thêm phụ lục).

3. Tập lệnh

3.1. Nhóm lệnh chuyển dữ liệu

3.1.1. RAM nội

Các lệnh trong nhóm lệnh chuyển dữ liệu trong RAM nội mô tả như bảng sau:

Bảng 2.1 – Các lệnh chuyển dữ liệu trong RAM nội

Lệnh	Hoạt động	Chế độ địa chỉ				Chu kỳ thực thi
		Tức thời	Trực tiếp	Gián tiếp	Thanh ghi	
MOV A,(byte)	A = (byte)	x	x	x	x	1
MOV (byte),A	(byte) = A		x	x	x	1
MOV (byte1),(byte2)	(byte1) = (byte2)	x	x	x	x	2
MOV DPTR,#data16	DPTR = data16	x				2
PUSH (byte)	SP = SP + 1 [SP] = (byte)		x			2
POP (byte)	(byte) = [SP] SP = SP – 1		x			2
XCH A,(byte)	Chuyển đổi dữ liệu giữa ACC và (byte)		x	x	x	1
XCHD A,@Ri	Chuyển đổi 4 bit thấp giữa ACC và @Ri			x		1

❖ **Lệnh MOV (Move):**

Di chuyển dữ liệu giữa các thanh ghi và bộ nhớ trong đó 128 byte RAM có địa chỉ từ 80h – FFh (chỉ có trong 8x52) chỉ có thể truy xuất bằng cách định địa chỉ gián tiếp. Các dạng của lệnh MOV như sau:

```
MOV A, Rn       ; Chuyển nội dung thanh ghi Rn vào thanh ghi A
MOV Rn, A       ; Chuyển nội dung thanh ghi A vào thanh ghi Rn
MOV A, direct   ; Chuyển nội dung ô nhớ trực tiếp vào thanh ghi A
MOV direct, A   ; Chuyển nội dung thanh ghi A vào ô nhớ trực tiếp
MOV A,@Ri       ; Chuyển nội dung của ô nhớ có địa chỉ chứa trong Ri vào A
MOV @Ri,A       ; Chuyển nội dung củaA vào ô nhớ có địa chỉ chứa trong Ri
```

MOV A, #data8 ; Chuyển giá trị 8 bit vào A
MOV Rn, direct; Chuyển nội dung ô nhớ trực tiếp vào thanh ghi Rn
MOV direct, Rn ; Chuyển nội dung thanh ghi Rn vào ô nhớ trực tiếp
MOV Rn, #data8; Chuyển giá trị 8 bit vào Rn
MOV direct, direct; Chuyển nội dung giữa 2 ô nhớ trực tiếp
MOV direct, @Ri; Chuyển nội dung của ô nhớ có địa chỉ chứa trong Ri vào ô nhớ trực tiếp
MOV @Ri, direct; Chuyển nội dung của ô nhớ trực tiếp vào ô nhớ có địa chỉ chứa trong Ri
MOV direct, #data8; Chuyển giá trị 8 bit vào ô nhớ trực tiếp
MOV @Ri, #data8; Chuyển giá trị 8 bit vào ô nhớ có địa chỉ chứa trong Ri
MOV C, bit ; Chuyển giá trị 1 bit vào cờ C
MOV bit, C ; Chuyển giá trị cờ C vào 1 bit
MOV DPTR, #data16 ; Chuyển giá trị tức thời 16 bit vào thanh ghi DPTR

Trong lệnh MOV, khi sử dụng địa chỉ trực tiếp từ 80h – FFh thì có thể thay bằng các từ gợi nhớ của các thanh ghi chức năng đặc biệt.

Ví dụ: lệnh MOV A, 80h có thể thay thế bằng lệnh MOV A, P0 (xem thêm bảng 1.2, chương 1).

Khi lệnh MOV thực hiện truy xuất bit, các bit có thể là địa chỉ trực tiếp (từ 00h – 7Fh) hay các từ gợi nhớ đã được định nghĩa. Các bit được định nghĩa trước mô tả như sau:

Bảng 2.2 – Các bit được định nghĩa trước trong 8951

Thanh ghi	Từ gợi nhớ	Địa chỉ bit	Thanh ghi	Từ gợi nhớ	Địa chỉ bit
A	ACC.0 – ACC.7	E0h – E7h	B	B.0 – B.7	F0h – F7h
PSW	CY hay C	D7h	SCON	SM0	9Fh
	AC	D6h		SM1	9Eh
	F0	D5h		SM2	9Dh
	RS1	D4h		REN	9Ch
	RS0	D3h		TB8	9Bh
	OV	D2h		RB8	9Ah
	P	D0h		TI	99h
				RI	98h
Các thanh ghi Port	P0.0 – P0.7	80h – 87h	IP	PS	BCh
	P1.0 – P1.7	90h – 97h		PX1	BBh
	P2.0 – P2.7	A0h – A7h		PT1	BAh
	P3.0 – P3.7	B0h – B7h		PX0	B9h
				PT0	B8h

IE	EA	AFh	TCON	TF1	8Fh
	ES	ACh		TR0	8Eh
	EX1	ABh		TF0	8Dh
	ET1	AAh		TR0	8Ch
	EX0	A9h		IE1	8Bh
	ET0	A8h		IT1	8Ah
				IE0	89h
				IT0	88h

Ví dụ: Lệnh MOV C, P0.0 có thể thay bằng lệnh MOV C, 80h.

❖ **Lệnh PUSH / POP:**

Các lệnh này cho phép cất hay lấy nội dung của stack. Khi thực hiện lệnh PUSH, nội dung thanh ghi SP tăng lên 1 và cất byte vào stack. Khi thực hiện lệnh POP, byte được lấy ra từ stack và sau đó giảm SP 1 giá trị. Lưu ý rằng khi sử dụng 8951, do bộ nhớ nội chỉ có 128 byte (00h – 7Fh) nên giá trị của SP không được vượt quá 7Fh (nếu vượt qua thì dữ liệu sẽ bị mất khi dùng lệnh PUSH và dữ liệu không xác định khi dùng lệnh POP). Còn đối với 8x52, do RAM nội là 256 byte nên không có hiện tượng này.

Các dạng của lệnh PUSH / POP:

PUSH direct ; Cất vào stack
POP direct ; Lấy dữ liệu từ stack

Lưu ý rằng lệnh PUSH và POP chỉ dùng cho địa chỉ trực tiếp nên không thể thực hiện lệnh PUSH Rn do thanh ghi Rn có 4 địa chỉ khác nhau tuỳ theo bank thanh ghi sử dụng.

Xét thanh ghi R0: 4 địa chỉ của R0 ứng với 4 bank là 00h, 08h, 10h, 18h. Mặc định khi reset, bank 0 được sử dụng nên các thanh ghi Rn có địa chỉ từ 00h – 07h. Khi đó thay vì dùng lệnh PUSH R0, ta có thể thay bằng lệnh PUSH 00h.

❖ **Lệnh XCH / XCHD (Exchange / Exchange Digit):**

Lệnh XCH / XCHD dùng để hoán chuyển 8 bit / 4 bit thấp của thanh ghi A với các thanh ghi khác hay bộ nhớ (lệnh XCHD chỉ dùng cho bộ nhớ nội định địa chỉ gián tiếp). Các dạng lệnh như sau:

XCH A,(byte) ; Hoán chuyển 8 bit
XCHD A,@Ri ; Hoán chuyển 4 bit thấp

Ví dụ: Xét đoạn lệnh:

```
MOV A, #30h   ; A = 30h
```

```
MOV R0, #54h  ; R0 = 54h
MOV 30h, #20h ;  Ô  nhớ  30h  chứa  giá  trị  20h  hay
              ;(30h) = 20h
XCH A, R0     ; Hoán chuyển giữa A và R0 → A = 54h
              ; và R0 = 30h
XCHD A, @R0   ; Chuyển 4 bit thấp giữa A và ô nhớ
              ; R0 = 30h → @R0: nội dung ô nhớ 30h → 20h
              ;Chuyển 4 bit thấp→ A = 50h và (30h) = 24h
```

3.1.2. RAM ngoại

Các lệnh trong nhóm lệnh chuyển dữ liệu trong RAM ngoại mô tả như sau:

Bảng 2.3 – Các lệnh chuyển dữ liệu trong RAM ngoại

Lệnh	Hoạt động	Chu kỳ thực thi
MOVX A, @Ri	Đọc nội dung từ RAM ngoại tại địa chỉ Ri	2
MOVX @Ri, A	Ghi vào RAM ngoại tại địa chỉ Ri	2
MOVX A, @DPTR	Đọc nội dung từ RAM ngoại tại địa chỉ DPTR	2
MOVX @DPTR, A	Ghi vào RAM ngoại tại địa chỉ DPTR	2

(MOVX : Move eXternal)

Đối với các lệnh đọc / ghi dữ liệu của RAM ngoại, chỉ cho phép thực hiện định địa chỉ gián tiếp. Khi địa chỉ RAM là 8 bit thì dùng thanh ghi R0 hay R1 còn nếu là địa chỉ 16 bit thì phải dùng thanh ghi DPTR. Lưu ý rằng khi dùng địa chỉ 8 bit thì các bit địa chỉ cao không sử dụng nên Port 2 có thể sử dụng cho mục đích khác nhưng nếu dùng địa chỉ 16 bit thì Port 2 chỉ có nhiệm vụ là xuất 8 bit địa chỉ cao.

Khi thực hiện lệnh đọc từ RAM ngoại, chân $\overline{RD}$ sẽ xuống mức thấp còn khi thực hiện lệnh ghi, chân $\overline{WR}$ xuống mức thấp.

3.1.3. Bảng tìm kiếm

Các lệnh trong nhóm lệnh tìm kiếm dữ liệu trong bảng mô tả như sau:

Bảng 2.4 – Các lệnh tìm kiếm dữ liệu

Lệnh	Hoạt động	Chu kỳ thực thi
MOVC A, @A + DPTR	Đọc nội dung bộ nhớ chương trình tại địa chỉ A + DPTR	2
MOVC A, @A +PC	Đọc nội dung bộ nhớ chương trình tại địa chỉ A + PC	2

(MOVC: Move Code)

Các lệnh này cho phép tìm kiếm dữ liệu đã định nghĩa sẵn trong bộ nhớ chương trình (nếu bộ nhớ chương trình là ROM ngoại thì tín hiệu đọc là $\overline{\text{PSEN}}$). Các thanh ghi DPTR hay PC (Program Counter: bộ đếm chương trình – xác định địa chỉ của lệnh kế tiếp sẽ thực hiện) chứa vị trí nền của các bảng tìm kiếm còn thanh ghi A chứa vị trí của phần tử (thông thường kích thước 1 phần tử trong bảng tìm kiếm là 1 byte).

Ví dụ: Lấy phần tử thứ 2 trong bảng LED_7S:

```
MOV A, #2          ; Phần tử thứ 2
MOV DPTR, #LED_7S  ; Địa chỉ nền của bảng tìm kiếm
MOVC A, @A + DPTR  ; Đọc nội dung phần tử
……
LED_7S: DB data8, data8, data8, data8, … ; Nội dung bảng tìm kiếm có thể đặt tuỳ ý trong bộ nhớ chương trình
```

Để sử dụng thanh ghi PC tìm kiếm dữ liệu, quá trinh tìm kiếm phải thực hiện thông qua chương trình con và bảng phải được đặt ngay sau chương trình con.

Ví dụ: Lấy phần tử thứ 2 trong bảng LED_7S:

```
MOV A, #2          ; Phần tử thứ 2
CALL Read_Led7s
……
Read_Led7s:
MOVC A, @A+PC
RET
LED_7S: DB 0, data8, data8, data8, data8, … ; Nội dung bảng tìm kiếm
```

Lưu ý rằng trong đoạn lệnh trên, khi thực hiện lệnh MOVC, thanh ghi PC sẽ chỉ đến lệnh kế tiếp là lệnh RET chứ không phải bảng LED_7S. Do đó, bảng tìm kiếm trong trường hợp này sẽ không có phần tử 0 mà bắt đầu tại phần tử 1. Để chương trình giống như cách thực hiện dùng DPTR, cần phải thay đổi chương trình con như sau:

Ví dụ: Lấy phần tử thứ 2 trong bảng LED_7S:

```
MOV A, #2          ; Phần tử thứ 2
CALL Read_Led7s
……
Read_Led7s:
INC A                  ; Tăng nội dung A lên 1 để hiệu chỉnh vị trí bảng
MOVC A, @A+PC
RET
```

```
LED_7S: DB data8, data8, data8, data8, … ; Nội dung bảng tìm kiếm
```

3.2. Nhóm lệnh xử lý bit

Họ MCS-51 chứa một bộ xử lý bit hoàn chỉnh. RAM nội có 128 bit có thể xử lý bit và các thanh ghi chức năng đặc biệt có thể hỗ trợ lên tới 128 bit (các bit trong SFR xem tại bảng 2.2). Các địa chỉ bit từ 00h – 7Fh nằm trong RAM nội còn các địa chỉ từ 80h – FFh nằm trong SFR.

Các lệnh trong nhóm lệnh logic mô tả như trong bảng sau:

Bảng 2.5 – Các lệnh logic

Lệnh	Hoạt động	Chu kỳ thực thi
ANL C,bit	C = C AND bit	2
ANL C,/bit	C = C AND (NOT bit)	2
ORL C,bit	C = C OR bit	2
ORL C,/bit	C = C OR (NOT bit)	2
MOV C,bit	C = bit	1
MOV bit,C	Bit = C	2
CLR C	C = 0	1
CLR bit	Bit = 0	1
SETB C	C = 1	1
SETB bit	Bit = 1	1
CPL C	C = NOT C	1
CPL bit	Bit = NOT bit	1
JC rel	Nhảy đến nhãn rel nếu C = 1	2
JNC rel	Nhảy đến nhãn rel nếu C = 0	2
JB bit,rel	Nhảy đến nhãn rel nếu bit = 1	2
JNB bit,rel	Nhảy đến nhãn rel nếu bit = 0	2
JBC bit,rel	Nhảy đến nhãn rel nếu bit = 1 và sau đó xoá bit	2

ANL: And logic; ORL: Or logic; CLR: Clear; CPL: Complement

Bit: các bit trong RAM nội từ 00h – 7Fh hay trong SFR theo bảng 2.2

Rel: địa chỉ tương đối (cho phép trong vùng từ -128 ÷ 127 byte trong bộ nhớ chương trình)

Ví dụ: Chuyển từ bit 00h vào P1.0

```
MOV C, 00h    ; Chuyển bit 00h vào cờ Carry
MOV P1.0, C   ; Chuyển cờ Carry vào P1.0
```

Lưu ý rằng trong tập lệnh logic không có lệnh XOR mà phải thực hiện bằng phần mềm, cụ thể như sau:

Thực hiện lệnh C = C XRL bit:

```
JNB bit, next
CPL C
Next:
```

Ngoài ra, các lệnh nhảy trên đều dùng địa chỉ tương đối, nghĩa là chỉ cho phép trong vùng từ -128 ÷ 127 byte. Nếu cần nhảy đến dịa chỉ xa hơn thì phải dùng các lệnh nhảy khác, như mô tả trong phần sau.

3.3. Nhóm lệnh chuyển điều khiển

Nhóm lệnh chuyển điều khiển bao gồm các lệnh nhảy, các lệnh liên quan đến chương trình con, mô tả như sau:

Bảng **2.6** – Các lệnh chuyển điều khiển

Lệnh	Hoạt động	Chu kỳ thực thi
JMP addr	Nhảy tới nhãn addr	2
JMP @A+DPTR	Nhảy tới địa chỉ A + DPTR	2
CALL addr	Gọi chương trình con tại địa chỉ addr	2
RET	Trở về từ chương trình con	2
RETI	Trở về từ chương trình con phục vụ ngắt	2
NOP	Không làm gì cả	1

JMP: Jump

RET: Return

RETI: Return from Interrupt

NOP: No Operation

Lệnh	Hoạt động	Chế độ địa chỉ				Chu kỳ thực thi
		Tức thời	Trực tiếp	Gián tiếp	Thanh ghi	
JZ rel	Nhảy đến nhãn rel nếu A = 0	Chỉ dùng cho thanh ghi A				2
JNZ rel	Nhảy đến nhãn rel nếu A ≠ 0	Chỉ dùng cho thanh ghi A				2
DJNZ (byte),rel	(byte) = (byte) - 1 Nếu (byte) ≠ 0 thì nhảy đến nhãn rel		x		x	2
CJNE A,(byte),rel	Nhảy đến nhãn rel nếu A ≠ (byte)	x	x			2
CJNE (byte), #data8,rel	Nhảy đến nhãn rel nếu (byte) ≠ data8			x	x	2

JZ: Jump if Zero; JNZ: Jump if Not Zero

DJNZ: Decrement and Jump if Not Zero

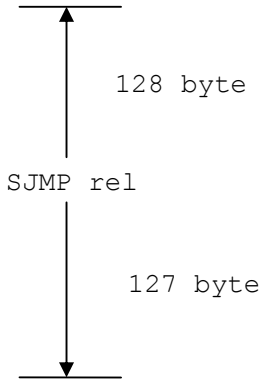
CJNE: Compare and Jump if Not Equal

❖ **Lệnh JMP (Jump):**

Lệnh JMP bao gồm 3 lệnh: LJMP (Long jump), AJMP (Absolute jump) và SJMP (Short jump) cho phép nhảy đến một vị trí bất kỳ trong chương trình.

Lệnh LJMP có kích thước 3 byte trong đó 1 byte mã lệnh và 2 byte chứa địa chỉ nhãn nên phạm vi biểu diễn địa chỉ là 64K (2 byte = 16 bit → phạm vi biểu diễn $2^{16} = 2^6 \times 2^{10} = 64K$). Do đó lệnh LJMP có thể thực hiện nhảy đến bất kỳ vị trí nào trong chương trình và địa chỉ sử dụng trong lệnh LJMP là địa chỉ tuyệt đối.

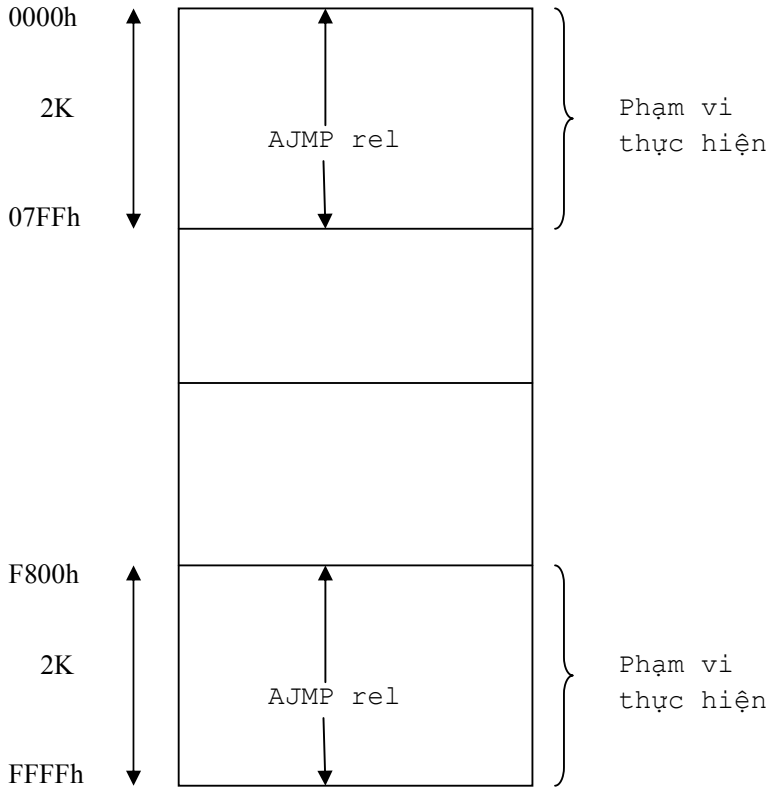
Lệnh SJMP có kích thước 2 byte trong đó có 1 byte mã lệnh và 1 byte địa chỉ nên phạm vi biểu diễn địa chỉ là 256 byte. Trong lệnh này, địa chỉ sử dụng không phải là địa chỉ tuyệt đối mà là địa chỉ tương đối (khoảng nhảy tính từ vị trí bắt đầu lệnh). Do byte địa chỉ sử dụng phương pháp bù 2 nên phạm vi biểu diễn từ -128 ÷ + 127, nghĩa là phạm vi nhảy của lệnh SJMP chỉ trong phạm vi từ - 128 đến 127 byte. Phạm vi thực hiện mô tả như hình vẽ.



Hình 2.1 – Phạm vi thực hiện của lệnh SJMP

Lệnh AJMP có kích thước 2 byte trong đó địa chỉ chứa trong 11 bit nên phạm vi biểu diễn địa chỉ là 2^{11} (2K). Trong khi đó, vùng địa chỉ tối đa của MCS-51 là 64K nên khi thực hiện lệnh AJMP, 64K chương trình phải chia thành từng vùng 2K (tổng cộng 32 vùng) và lệnh AJMP chỉ có thể thực hiện trong một vùng.

Tuy nhiên, khi lập trình cho MCS-51, thông thường các chương trình dịch đều cho phép sử dụng lệnh **JMP** thay thế cho 3 lệnh trên. Khi biên dịch, chương trình dịch sẽ tự động thay thế bằng các lệnh thích hợp.



Hình 2.2 – Phạm vi thực hiện của lệnh AJMP

Lệnh JMP @A + DPTR cho phép chọn các vị trí nhảy khác nhau tuỳ theo giá trị trong thanh ghi A. Địa chỉ nhảy đến chính là tổng giá trị của thanh ghi A và DPTR.

Ví dụ:

```
MOV DPTR, # JUMP_TABLE ; Địa chỉ bảng nhảy
MOV A, INDEX_NUMBER    ; Vị trí nhảy
MOV B, #3              ; x3 do lệnh LJMP
MUL AB                 ; có kích thước 3
JMP @ A + DPTR
……
JUMP_TABLE:
LJMP LABEL0            ; Vị trí nhảy 0
LJMP LABEL1            ; Vị trí nhảy 1
LJMP LABEL2            ; Vị trí nhảy 2
LJMP LABEL3            ; Vị trí nhảy 3
LJMP LABEL4            ; Vị trí nhảy 4
```

❖ **Lệnh CALL, RET, RETI:**

Lệnh CALL dùng để gọi chương trình con, bao gồm 2 lệnh: ACALL (Absolute Call) và LCALL (Long Call). Vị trí có thể gọi lệnh CALL giống như đã xét trong lệnh JMP. Khi lập trình, thông thường các chương trình dịch cũng cho phép thay thế duy nhất bằng lệnh CALL và khi biên dịch, lệnh CALL sẽ được thay thế bằng lệnh ACALL hay LCALL tuỳ theo vị trí gọi lệnh. Lưu ý rằng khi thực hiện lệnh CALL thì trong chương trình con phải kết thúc bằng lệnh RET.

Ngoài ra, khi sử dụng các chương trình con phục vụ ngắt, khi kết thúc phải dùng lệnh RETI. Lệnh RETI và lệnh RET chỉ khác nhau ở chỗ lệnh RETI báo cho hệ thống điều khiển ngắt biết rằng quá trình xử lý ngắt đã thực hiện xong.

❖ **Lệnh JZ, JNZ:**

Lệnh JZ và JNZ dùng để kiểm tra nội dung của thanh ghi A. Lệnh JZ nhảy khi A = 0 và JNZ nhảy khi A ≠ 0. Lưu ý rằng phạm vi nhảy chỉ cho phép trong khoảng từ -128 ÷ 127 byte (giống như khi sử dụng lệnh SJMP).

❖ **Lệnh DJNZ:**

Lệnh DJNZ thường được dùng để tạo vòng lặp. Số lần lặp được chuyển vào thanh ghi đếm ở đầu vòng lặp (thanh ghi đếm có thể dùng bất kỳ thanh ghi nào hay là bộ nhớ).

Ví dụ:

```
     MOV R7, #10 ; Lặp 10 lần
LOOP:
     ……
     ……
     DJNZ R7, LOOP
```

❖ **Lệnh CJNE:**

Lệnh CJNE dùng để so sánh 2 giá trị với nhau, khi 2 giá trị này khác nhau thì sẽ thực hiện lệnh nhảy. Lưu ý rằng trong tập lệnh của MCS-51 không có lệnh lớn hơn hay nhỏ hơn nên chỉ có thể thực hiện các lệnh này bằng cách kết hợp lệnh CJNE và nội dung của cờ Carry.

Trong lệnh CJNE, nếu byte đầu tiên nhỏ hơn byte thứ hai thì CF = 1. Ngược lại (byte đầu tiên lớn hơn hay bằng byte thứ hai) thì CF = 0.

Ví dụ: Kiểm tra nội dung của thanh ghi A, nếu A nhỏ hơn 10 thì xuất giá trị trong thanh ghi A ra Port 1. Ngược lại thì xuất giá trị 10 ra Port 1.

```
     CJNE A,#10,Khacnhau; So sánh A với 10
     JMP  Xuat10        ; Nếu A = 10 thì xuất giá trị 10
Khacnhau:
     JC   XuatA         ; Nếu CF = 1 (A < 10) thì xuất nội
Xuat10:                 ; dung trong A ra P1
     MOV  P1,#10
     SJMP Tiep
XuatA:
     MOV  P1,A
Tiep:
```

3.4. Nhóm lệnh logic

Nhóm lệnh logic bao gồm các lệnh liên quan đến xử lý logic theo từng byte, mô tả như sau:

Bảng 2.7 – Các lệnh logic

Lệnh	**Hoạt động**	**Chế độ địa chỉ**				**Chu kỳ thực thi**
		Tức thời	Trực tiếp	Gián tiếp	Thanh ghi	
ANL A,(byte)	A = A AND (byte)	x	x	x	x	1
ANL (byte),A	(byte)=(byte) AND A		x			1
ANL (byte),#data8	(byte)=(byte)AND data8		x			2
ORL A,(byte)	A = A OR (byte)	x	x	x	x	1
ORL (byte),A	(byte)=(byte) OR A		x			1
ORL (byte),#data8	(byte)=(byte) OR data8		x			2
XRL A,(byte)	A = A XOR (byte)	x	x	x	x	1
XRL (byte),A	(byte)=(byte) XOR A		x			1
XRL (byte),#data8	(byte)=(byte) XOR data8		x			2
CLR A	A = 0	Chỉ dùng cho thanh ghi A				1
CPL A	A = NOT A	Chỉ dùng cho thanh ghi A				1
RR A	Quay phải thanh ghi A 1 bit	Chỉ dùng cho thanh ghi A				1
RLC A	Quay phải thanh ghi A và CF 1 bit	Chỉ dùng cho thanh ghi A				1
RL A	Quay trái thanh ghi A 1 bit	Chỉ dùng cho thanh ghi A				1

RLC A	Quay trái thanh ghi A và CF 1 bit	Chỉ dùng cho thanh ghi A	1
SWAP A	Đổi vị trí nibble cao và thấp của ACC	Chỉ dùng cho thanh ghi A	1

RL: Rotate Left, RLC: Rotate Left through Carry

RR: Rotate Right; RRC: Rotate Right through Carry

❖ **Lệnh ANL, ORL, XRL:**

Các lệnh logic này thực hiện giống như trong các lệnh xử lý bit nhưng thực hiện trên 8 bit của các thanh ghi hay bộ nhớ. Lệnh XRL còn được dùng để đảo tất cả các bit như sau:

```
XRL P0, #0FFh
```

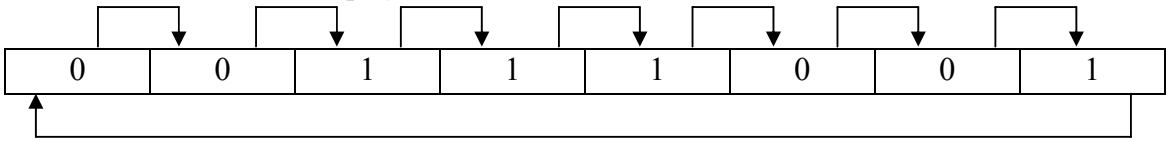
❖ **Lệnh RR, RRC, RL, RLC:**

Các lệnh này dùng để quay phải hay quay trái thanh ghi A 1 bit.

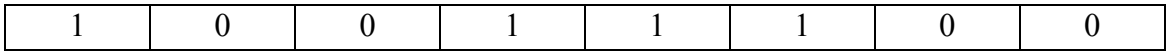
Ví dụ: Giả sử thanh ghi A = 39h (0011 1001b), CF = 1. Nội dung thanh ghi A sau khi thực hiện các lệnh quay tương ứng như sau:

RR A:

Trước khi quay:



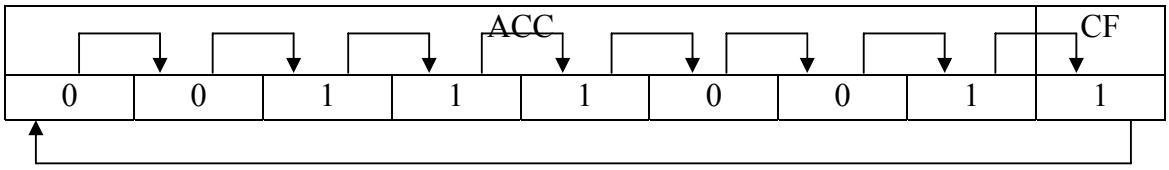
Sau khi quay:



RL A: A = 0111 0010b (72h)

RRC A:

Trước khi quay:



Sau khi quay:

ACC								CF
1	0	0	1	1	1	0	0	1

RLC A: A = 0111 0011b (73h); CF = 0

❖ **Lệnh SWAP:**

Lệnh SWAP A dùng để hoán chuyển nội dung 2 nibble trong thanh ghi A.

VÍ Dụ: Nếu nội dung thanh ghi A = 39h thì sau khi thực hiện lệnh SWAP A, nội dung thanh ghi A là 93h.

3.5. Nhóm lệnh số học

Các lệnh trong nhóm lệnh số học mô tả như trong bảng sau:

Bảng 2.8 – Các lệnh số học

Lệnh	**Hoạt động**	**Chế độ địa chỉ**				**Chu kỳ thực thi**
		Tức thời	Trực tiếp	Gián tiếp	Thanh ghi	
ADD A,(byte)	A = A + (byte)	x	x	x	x	1
ADDC A,(byte)	A = A + (byte) + C	x	x	x	x	1
SUBB A,(byte)	A = A - (byte) - C	x	x	x	x	1
INC A	A = A + 1	Chỉ dùng cho thanh ghi tích luỹ ACC				1
INC (byte)	(byte) = (byte) + 1		x	x	x	1
INC DPTR	DPTR = DPTR + 1	Chỉ dùng cho thanh ghi con trỏ lệnh DPTR				2
DEC A	A = A - 1	Chỉ dùng cho thanh ghi tích luỹ ACC				1
DEC (byte)	(byte) = (byte) - 1		x	x	x	1
MUL AB	B_A = B x A	Chỉ dùng cho thanh ghi tích luỹ ACC và thanh ghi B				4
DIV AB	A = A div B B = A mod B	Chỉ dùng cho thanh ghi tích luỹ ACC và thanh ghi B				4
DA A	Hiệu chỉnh trên số BCD	Chỉ dùng cho thanh ghi tích luỹ ACC				1

❖ **Lệnh ADD:**

Thực hiện cộng giữa thanh ghi tích luỹ A và một toán hạng khác. Lệnh ADD ảnh hưởng đến các cờ Carry (C), Overflow (OV) và Auxiliary (AC).

Lệnh ADD có 4 chế độ địa chỉ khác nhau:

- `ADD A, #30h` ; định địa chỉ tức thời (A = A + 30h)
- `ADD A, 30h` ; định địa chỉ trực tiếp (A = A + [30h] trong đó [30h] là giá trị của RAM nội có địa chỉ 30h)
- `ADD A, @R0` ; định địa chỉ gián tiếp (A = A + [R0] trong đó [30h] là giá trị của RAM nội có địa chỉ chứa trong thanh ghi R0)

```
MOV R0,#30h    ; R0 = 30h
```

```
ADD A,@R0      ; A = A + [R0] = A + [30h] (cộng nội dung của thanh ghi ACC với RAM nội có địa chỉ 30h)
```

- `ADD A,R0` ; định địa chỉ thanh ghi (A = A + R0)

❖ **Lệnh ADDC, SUBB:**

Thực hiện cộng hay trừ nội dung của thanh ghi A với một toán hạng khác trong đó có dùng thêm cờ Carry. Lệnh ADDC và SUBB ảnh hưởng đến các cờ C, OV và AC.

❖ **Lệnh MUL:**

Nhân nội dung của thanh ghi A với thanh ghi B. Lệnh MUL ảnh hưởng đến cờ OV và xoá cờ C (C = 0).

Ví dụ:

```
MOV  A,#50     ; 50 x 25 = 1250 → 04E2h
MOV  B,#25     ; byte cao = 04h, byte thấp = E2h
MUL  AB        ; B = 04h, A = E2h
```

❖ **Lệnh DIV:**

Chia nội dung của thanh ghi A cho thanh ghi B. Lệnh DIV ảnh hưởng đến cờ OV và xoá cờ C (C = 0).

Ví dụ:

```
MOV  A,#250    ; 250 / 40 = 6 dư 10
MOV  B,#40     ;
DIV  AB        ; B = 0Ah (10), A = 06h
```

❖ **Lệnh DA A:**

Hiệu chỉnh nội dung thanh ghi A sau khi thực hiện các phép toán liên quan đến số BCD. Quá trình thực hiện lệnh DA A mô tả như sau:

- Nếu A[3-0] > 9 hay AC = 1 thì A[3-0] = A[3-0] + 6
- Nếu A[7-4] > 9 hay C = 1 thì A[7-4] = A[7-4] + 6

Lệnh DA A cũng ảnh hưởng đến cờ C.

BÀI TẬP CHƯƠNG 2

1. Xác định giá trị của các biểu thức sau:

a. (10 SHL 2) OR (1000 1000b)

b. (5*2 – 10 SHR 1) AND (11h)

c. HIGH(10000)

d. LOW(-30000)

2. Viết đoạn chương trình đọc nội dung của ô nhớ 30h. Nếu giá trị đọc lớn hơn hay bằng 10 thì xuất 10 ra P0, ngược lại thì xuất giá trị vừa đọc ra P0.

3. Viết đoạn chương trình xuất các giá trị trong ô nhớ 30h – 3Fh ra P1 (giữa các lần xuất có thời gian trì hoãn).

4. Viết đoạn chương trình theo yêu cầu sau:

   - Đọc dữ liệu từ P1 (10 lần) và lưu giá trị đọc mỗi lần vào ô nhớ 30h – 39h (mỗi lần đọc có trì hoãn một khoảng thời gian).
   - Tìm giá trị lớn nhất trong các ô nhớ 30h – 39h, lưu vào ô nhớ 3Ah và xuất giá trị này ra P2.
   - Kiểm tra nội dung ô nhớ 3Ah, nếu = 0 thì quay lại đầu chương trình, ngược lại thì xuất giá trị này ra P3.

5. Viết đoạn chương trình theo yêu cầu:

   - B1: Kiểm tra bit P3.0:

P3.0	**Thực hiện**
= 0	Đến bước 2
= 1	Đến bước 3

   - B2: Đọc dữ liệu từ P2, đảo tất cả các bit và xuất ra P0. Sau đó quay lại bước 1.
   - B3: xuất nội dung tại ô nhớ 30h ra P1 và quay lại bước 1

CHƯƠNG 3: LẬP TRÌNH HỢP NGỮ

1. Các tập tin .EXE và .COM

DOS chỉ có thể thi hành được các tập tin dạng .COM và .EXE. Tập tin .COM thường dùng để xây dựng cho các chương trình nhỏ còn .EXE dùng cho các chương trình lớn.

1.1. Tập tin .COM

- Tập tin .COM chỉ có một đoạn nên kích thước tối đa của một tập tin loại này là 64 KB.
- Tập tin .COM được nạp vào bộ nhớ và thực thi nhanh hơn tập tin .EXE nhưng chỉ áp dụng được cho các chương trình nhỏ.
- Chỉ có thể gọi các chương trình con dạng near.

Khi thực hiện tập tin .COM, DOS định vị bộ nhớ và tạo vùng nhớ dài 256 byte ở vị trí 0000h, vùng này gọi là PSP (Program Segment Prefix), nó sẽ chứa các thông tin cần thiết cho DOS. Sau đó, các mã lệnh trong tập tin sẽ được nạp vào sau PSP ở vị trí 100h và đưa giá trị 0 vào stack. Như vậy, kích thước tối đa thực sự của tập tin .COM là 64 KB – 256 byte PSP – 2 byte stack.

Tất cả các thanh ghi đoạn đều chỉ đến PSP và thanh ghi con trỏ lệnh IP chỉ đến 100h, thanh ghi SP có giá trị 0FFFEh.

1.2. Tập tin .EXE

- Nằm trong nhiều đoạn khác nhau, kích thước thông thường lớn hơn 64 KB.
- Có thể gọi được các chương trình con dạng near hay far.
- Tập tin .EXE chứa một header ở đầu tập tin để chứa các thông tin điều khiển cho tập tin.

2. Khung của một chương trình hợp ngữ

Khung của một chương trình hợp ngữ có dạng như sau:

```
TITLE       Chương trình hợp ngữ
.MODEL      Kiểu kích thước bộ nhớ     ; Khai báo quy mô sử
                                       ; dụng bộ nhớ
.STACK      Kích thước                 ; Khai báo dung lượng
                                       ; đoạn stack
.DATA                                  ; Khai báo đoạn dữ  liệu
msg   DB    'Hello$'
.CODE                                  ; Khai báo đoạn mã
main  PROC
…
CALL        Subname                    ; Gọi chương trình con
…
main  ENDP

Subname     PROC                       ; Định nghĩa chương
                                       ; trình con
…
```

```
RET
Subname     ENDP
END   main
```

❖ **Quy mô sử dụng bộ nhớ:**

Bảng 3.1:

Loại	Mô tả
Tiny	Mã lệnh và dữ liệu nằm trong một đoạn
Small	Mã lệnh trong một đoạn, dữ liệu trong một đoạn
Medium	Mã lệnh không nằm trong một đoạn, dữ liệu trong một đoạn
Compact	Mã lệnh trong một đoạn, dữ liệu không nằm trong một đoạn
Large	Mã lệnh không nằm trong một đoạn, dữ liệu không nằm trong một đoạn và không có mảng nào lớn hơn 64KB
Huge	Mã lệnh không nằm trong một đoạn, dữ liệu không nằm trong một đoạn và các mảng có thể lớn hơn 64KB

Thông thường, các ứng dụng đơn giản chỉ đòi hỏi mã chương trình không quá 64 KB và dữ liệu cũng không lớn hơn 64 KB nên ta sử dụng ở dạng Small:

```
.MODEL      SMALL
```

❖ **Khai báo kích thước stack:**

Khai báo stack dùng để dành ra một vùng nhớ dùng làm stack (chủ yếu phục vụ cho chương trình con), thông thường ta chọn khoảng 256 byte là đủ để sử dụng (nếu không khai báo thì chương trình dịch tự động cho kích thước stack là 1 KB):

```
.STACK      256
```

❖ **Khai báo đoạn dữ liệu:**

Đoạn dữ liệu dùng để chứa các biến và hằng sử dụng trong chương trình.

❖ **Khai báo đoạn mã:**

Đoạn mã dùng chứa các mã lệnh của chương trình. Đoạn mã bắt đầu bằng một chương trình chính và có thể có các lệnh gọi chương trình con (CALL).

Một chương trình chính hay chương trình con bắt đầu bằng lệnh PROC và kết thúc bằng lệnh ENDP (đây là các lệnh giả của chương trình dịch). Trong chương trình con, ta sử dụng thêm lệnh RET để trả về địa chỉ lệnh trước khi gọi chương trình con.

3. Cú pháp của các lệnh trong chương trình hợp ngữ

Một dòng lệnh trong chương trình hợp ngữ gồm có các trường (field) sau (không nhất thiết phải đầy đủ tất cả các trường):

```
Tên   Lệnh    Toán hạng   Chú thích
A:    MOV     AH,10h      ; Đưa giá trị 10h vào thanh ghi AH
Main  PROC
```

Trường tên chứa nhãn, tên biến hay tên thủ tục. Các tên nhãn có thể chứa tối đa 31 ký tự, không chứa ký tự trắng (space) và không được bắt đầu bằng số (A: hay Main:). Các nhãn được kết thúc bằng dấu '**:**'.

Trường lệnh chứa các lệnh sẽ thực hiện. Các lệnh này có thể là các lệnh thật (MOV) hay các lệnh giả (PROC). Các lệnh thật sẽ được dịch ra mã máy.

Trường toán hạng chứa các toán hạng cần thiết cho lệnh (AH,10h).

Trường chú thích phải được bắt đầu bằng dấu '**;**'. Trường này chỉ dùng cho người lập trình để ghi các lời giải thích cho chương trình. Chương trình dịch sẽ bỏ qua các lệnh nằm phía sau dấu **;**.

3.1. Khai báo dữ liệu

Khi khai báo dữ liệu trong chương trình, nếu sử dụng số nhị phân, ta phải dùng thêm chữ **B** ở cuối, nếu sử dụng số thập lục phân thì phải dùng chữ **H** ở cuối. ***Chú ý rằng đối với số thập lục phân, nếu bắt đầu bằng chữ A..F thì phải thêm vào số 0 ở phía trước.***

Ví dụ:

```
1011b       ; Số nhị phân
1111        ; Số thập phân
1011h       ; Số thập lục phân
```

3.2. Khai báo biến

Khai báo biến nhằm để chương trình dịch cung cấp một địa chỉ xác định trong bộ nhớ. Ta dùng các lệnh giả sau để định nghĩa các biến ứng với các kiểu dữ liệu khác nhau: DB (define byte), DW (define word) và DD (define double word).

VD:

```
A1    DB    1       ; Định nghĩa biến A1 dài 1 byte (chương
                    ; trình dịch sẽ dùng 1 byte trong bộ nhớ để
                    ; lưu trữ A1), trị ban đầu A1 = 1
A2    DB    ?       ; Biến A2 kiểu byte, không có giá trị gán
                    ; ban đầu
A3    DB    'A'     ; Biến kiểu ký tự
A4    DW    1       ; Định nghĩa biến A4 dài 2 byte, giá trị ban
                    ; đầu A4 = 1, ta cũng có thể dùng dấu ? để
                    ; xác định biến không cần khởi tạo giá trị ban đầu
A5    DD    1       ; Biến A5 dài 4 byte
A6    DB    1,2,3   ; Định nghĩa biến mảng (array) gồm có 3
                    ; phần tử, mỗi phần tử dài 1 byte (nghĩa là
                    ; sẽ dùng 3 byte lưu trữ) với các giá trị ban
                    ; đầu của các phần tử lần lượt là 1,2,3
A7    DB    10      DUP(0)
                    ; Khai báo biến mảng gồm 10 phần tử, mỗi
                    ; phần tử có chiều dài 1 byte với giá trị gán
                    ; ban đầu là 0
```

```
A8    DB    10    DUP(?)
                  ; Khai báo biến mảng gồm 10 phần tử, mỗi
                  ; phần tử có chiều dài 1 byte, không cần
                  ; gán giá trị ban đầu
```

Ngoài ra ta có thể dùng các toán tử DUP lồng vào nhau khi khai báo biến mảng. Giả sử ta cần khai báo mảng A9 có các giá trị gán ban đầu 1,2,3,1,1,3,2,2,1,1,3,2,2. Ta có thể thực hiện như sau:

```
      A9    DB    1,2,3,1,1,3,2,2,1,1,3,2,2
Hay:  A9    DB    1,2,3,2 DUP(1,1,3,2,2)
Hay:  A9    DB    1,2,3,2 DUP(2 DUP(1),3,2 DUP(2))
```

Đối với các biến có nhiều hơn 1 byte, byte thấp sẽ chứa ở ô nhớ có địa chỉ thấp và byte cao sẽ chứa ở ô nhớ có địa chỉ cao.

VD:

```
A10   DW    1234h
```

Biến A10 giả sử bắt đầu lưu tại địa chỉ 1000h thì ô nhớ 1000h chứa giá trị 34h còn ô nhớ 1001h chứa giá trị 12h.

Đối với biến kiểu chuỗi (string), thực chất là một mảng các ký tự, ta có thể khai báo như sau:

```
      A11   DB    'ABCD'
Hay   A11   DB    65h,66h,67h,68h
```

Sau lệnh khai báo này thì ô nhớ 1000h (giả sử biến A11 lưu trữ tại địa chỉ 1000h) chứa 'A', 1001h chứa 'B', 1002h chứa 'C' và 1003h chứa 'D'.

3.3. Khai báo hằng

Các hằng khai báo trong chương trình hợp ngữ bằng lệnh giả EQU để chương trình dễ hiểu hơn. Hằng có thể ở dạng số, ký tự hay chuỗi.

VD:

```
A12   EQU   10
A13   EQU   'AAA'
```

Sau khi sử dụng khai báo này, nếu ta dùng lệnh:

```
MOV  AH,A12
```

thì AH = 10h

```
A14   DB    'B',A13
```

thì khai báo chuỗi A14 với giá trị gán ban đầu là 'BAAA'.

4. Các toán tử trong hợp ngữ

❖ **Toán tử số học:**

Bảng 3.2:

Toán tử	Cú pháp	Mô tả
+	+bt	Số dương
-	-bt	Số âm
*	bt1*bt2	Nhân
/	bt1/bt2	Chia
mod	bt1 mod bt2	Lấy phần dư
+	bt1 + bt2	Cộng
-	bt1 – bt2	Trừ
shl	bt shl n	Dịch trái n bit
shr	bt shr n	Dịch phải n bit

Trong đó bt, bt1, bt2 là các biểu thức hằng, n là số nguyên.

```
VD:  MOV  AH,(8+1)*7/3          ; AH ← 21
     MOV  AH, 00010001b shr 2   ; AH ← 0000 0100b
     MOV  AH,00010001b shl 2    ; AH ← 0100 0100b
     MOV  AH,100 mod 3          ; AH ← 1
```

❖ **Toán tử logic:**

Bao gồm các toán tử AND, OR, NOT, XOR

```
VD:  MOV  AH,10 OR 4 AND 2      ; AH = 10
     MOV  AH, 0F0h AND 7Fh      ; AH = 70h
```

❖ **Toán tử quan hệ:**

Các toán tử quan hệ so sánh 2 biểu thức, cho giá trị true (-1) nếu điều kiện thoả và false (0) nếu không thoả.

Bảng 3.3:

Toán tử	Cú pháp	Mô tả
EQ	bt1 EQ bt2	Bằng
NE	bt1 NE bt2	Không bằng
LT	bt1 LT bt2	Nhỏ hơn
LE	bt1 LE bt2	Nhỏ hơn hay bằng
GT	bt1 GT bt2	Lớn hơn
GE	bt1 GE bt2	Lớn hơn hay bằng

❖ **Các toán tử cung cấp thông tin:**

➢ **Toán tử SEG:**

SEG bt

Toán tử SEG xác định địa chỉ đoạn của biểu thức ***bt***. ***bt*** có thể là biến, nhãn, hay các toán hạng bộ nhớ.

➢ **Toán tử OFFSET:**

OFFSET bt

Toán tử OFFSET xác định địa chỉ offset của biểu thức ***bt***. ***bt*** có thể là biến, nhãn, hay các toán hạng bộ nhớ.

```
VD:   MOV  AX,SEG A        ; Nạp địa chỉ đoạn và địa chỉ offset
      MOV  DS,AX           ; của biến A vào cặp thanh ghi
      MOV  AX,OFFSET A     ; DS:AX
```

➢ **Toán tử chỉ số [ ]: (index operator)**

Toán tử chỉ số thường dùng với toán hạng trực tiếp và gián tiếp.

➢ **Toán tử (:) (segment override operator)**

Segment:bt

Toán tử **:** quy định cách tính địa chỉ đối với segment được chỉ. ***Segment*** là các thanh ghi đoạn CS, DS, ES, SS.

Chú ý rằng khi sử dụng toán tử : kết hợp với toán tử [ ] thì ***segment:*** phải đặt ngoài toán tử [ ].

VD: Cách viết [CS:BX] là sai, ta phải viết CS:[BX]

➢ **Toán tử TYPE:**

TYPE bt

Trả về giá trị biểu thị dạng của biểu thức ***bt***.

- Nếu ***bt*** là biến thì sẽ trả về 1 nếu biến có kiểu byte, 2 nếu biến có kiểu word, 4 nếu biến có kiểu double word.
- Nếu ***bt*** là nhãn thì trả về 0FFFFh nếu bt là near và 0FFFEh nếu bt là far.
- Nếu ***bt*** là hằng thì trả về 0.

➢ **Toán tử LENGTH:**

LENGTH bt

Trả về số các đơn vị cấp cho biến ***bt***

➢ **Toán tử SIZE:**

SIZE bt

Trả về tổng số các byte cung cấp cho biến ***bt***

```
VD:   A      DD    100    DUP(?)
      MOV  AX,LENGTH A     ; AX = 100
      MOV  AX,SIZE A       ; AX = 400
```

❖ **Các toán tử thuộc tính:**

➢ **Toán tử PTR:**

Loai PTR bt

Toán tử này cho phép thay đổi dạng của biểu thức ***bt***.

- Nếu ***bt*** là biến hay toán hạng bộ nhớ thì ***Loai*** là byte, word hay dword.
- Nếu ***bt*** là nhãn thì ***Loai*** là near hay far.

```
VD:  A     DW   100   DUP(?)
     B     DD   ?
     MOV  AH,BYTE PTR A        ; Đưa byte đầu tiên trong mảng A
                               ; vào thanh ghi AH
     MOV  AX,WORD PTR B        ; Đưa 2 byte thấp trong biến B
                               ; vào thanh ghi AX
```

➢ **Toán tử HIGH, LOW:**

HIGH bt

LOW bt

Cho giá trị của byte cao và thấp của biểu thức ***bt***, ***bt*** phải là một hằng.

```
VD:  A     EQU  1234h
     MOV  AH,HIGH A            ; AH ← 12h
     MOV  AH,LOW A             ; AH ← 34h
```

5. Các cách định địa chỉ trong hợp ngữ

❖ **Toán hạng trực tiếp:**

Toán hạng trực tiếp là một biểu thức hằng xác định. Các hằng số có thể ở dạng thập phân (có dấu và không dấu), nhị phân, thập lục phân, các hằng số định nghĩa bằng lệnh EQU, …

```
VD:  MOV  AH,10
     MOV  AH,1010b
     MOV  AH,0Ah
     MOV  AH,A12
     MOV  AX,OFFSET msg
     MOV  AX,SEG msg
```

❖ **Toán hạng thanh ghi:**

Các thanh ghi có thể sử dụng trong phép định địa chỉ thanh ghi là AH, BH, CH, DH, AL, BL, CL, DL, AX, BX, CX, DX, SP, BP, SI, DI, CS, DS, ES, SS.

❖ **Toán hạng bộ nhớ:**

➢ **Trực tiếp:**

Toán hạng này xác định dữ liệu lưu trong bộ nhớ tại một địa chỉ xác định khi dịch, địa chỉ này là một biểu thức hằng (có thể kết hợp với toán tử chỉ số [ ] hay toán tử +, -, :). Thanh ghi đoạn mặc định là thanh ghi DS nhưng ta có thể dùng toán tử : để chỉ thanh ghi đoạn khác.

VD:
```
A     DW   1000h
B     DB   100   DUP(0)
MOV  AX,A              ; Chuyển nội dung của biến A vào
MOV  AX,[A]            ; thanh ghi AX
MOV  AH,B              ; Truy xuất phần tử đầu tiên của
MOV  AH,B[0]           ; mảng B
MOV  AH,B + 1          ; Truy xuất phần tử thứ hai của
MOV  AH,B[1]           ; mảng B
MOV  AH,B  + 5         ; Truy xuất phần tử thứ 6 của
MOV  AH,B[5]           ; mảng B
```

Chú ý rằng lệnh MOV AX,[1000h] sẽ chuyển giá trị 1000h vào thanh ghi AX. Nếu muốn chuyển nội dung tại ô nhớ 1000h vào thanh ghi AX thì phải dùng lệnh MOV AX,DS:[1000h] hay MOV AX,DS:1000h

- **Gián tiếp:**

Toán hạng bộ nhớ gián tiếp cho phép dùng các thanh ghi BX, BP, SI, DI để chỉ các giá trị trong bộ nhớ.

VD:
```
MOV  BX,2
MOV  SI,3
MOV  AH,B[BX]          ; Chuyển phần tử thứ 3 của mảng B
                       ; vào thanh ghi AH
MOV  AH,B[BX+1]        ; Chuyển phần tử thứ 4 của mảng B
MOV  AH,B[BX]+1        ; vào thanh ghi AH (BX + 1 = 3)
MOV  AH,B[BX+SI]       ; Chuyển phần tử thứ 6 của mảng B
MOV  AH,B[BX][SI]      ; vào thanh ghi AH
MOV  AH,[B+BX+SI]      ; BX + SI = 5
MOV  AH,[B][BX][SI]
MOV  AH,B[BX+SI+5]     ; Chuyển phần tử thứ 11 của mảng B
MOV  AH,B[BX][SI]+5    ; vào thanh ghi AH
MOV  AH,[B+BX+SI+5]    ; BX + SI + 5 = 10
```

6. Tạo và thực thi chương trình hợp ngữ

Ta có thể tạo và thực thi một chương trình hợp ngữ trên một máy PC theo các bước sau:

- Dùng một chương trình soạn thảo văn bản ***không định dạng*** (như NC) tạo một tập tin chứa chương trình hợp ngữ (gán phần mở rộng của tập tin này là .ASM, giả sử là TEMP.ASM).
- - Dùng chương trình TASM.EXE (Turbo Assembler) để dịch ra mã máy dạng .OBJ: **TASM TEMP**
- Sau khi dịch xong, ta sẽ được file TEMP.OBJ chứa các mã máy của chương trình. Để chuyển thành file thực thi, ta dùng chương trình TLINK.EXE để chuyển thành tập tin .EXE: **TLINK TEMP**
- - Nếu tập tin thực thi ở dạng .COM thì ta dùng thêm chương trình EXE2BIN.EXE: **EXE2BIN TEMP TEMP.COM**

7. Tập lệnh hợp ngữ

7.1. Nhóm lệnh chuyển dữ liệu

7.1.1. Nhóm lệnh chuyển dữ liệu đa dụng

❖ Lệnh **MOV dst,src**: chuyển nội dung toán hạng src vào toán hạng dst. Toán hạng nguồn src có thể là thanh ghi (reg), bộ nhớ (mem) hay giá trị tức thời (immed); toán hạng đích dst có thể là reg hay mem.

Lệnh MOV có thể có các trường hợp sau:

Reg8 ← reg8	MOV AL,AH
Reg16 ← reg16	MOV AX,BX
Mem8 ← reg8	MOV [BX],AL
Reg8 ← mem8	MOV AL,[BX]
Mem16 ← reg16	MOV [BX],AX
Reg16 ← mem16	MOV AX,[BX]
Reg8 ← immed8	MOV AL,04h
Mem8 ← immed8	MOV mem[BX],01h
Reg16 ← immed16	MOV AL,0F104h
Mem16 ← immed16	MOV mem[BX],0101h
SegReg ← reg16	MOV DS,AX
SegReg ← mem16	MOV DS,mem
Reg16 ← segreg	MOV AX,DS
Mem16 ← segreg	MOV [BX],DS

- Lệnh MOV không ảnh hưởng đến các cờ.
- Không thể chuyển trực tiếp dữ liệu giữa hai ô nhớ mà phải thông qua một thanh ghi

```
MOV  AX,mem1
MOV  mem2,AX
```

- Không thể chuyển giá trị trực tiếp vào thanh ghi đoạn

```
MOV  AX,1010h
MOV  DS,AX
```

- Không thể chuyển trực tiếp giữa 2 thanh ghi đoạn
- Không thể dùng thanh ghi CS làm toán hạng đích

❖ Lệnh **XCHG dst,src**: (Exchange) hoán chuyển nội dung 2 toán hạng. Toán hạng chỉ có thể là reg hay mem.

- Lệnh XCHG không ảnh hưởng đến các cờ
- Không thể dùng cho các thanh ghi đoạn

❖ Lệnh **PUSH src**: cất nội dung một thanh ghi vào stack. Toán hạng là reg16

❖ Lệnh **POP dst**: lấy dữ liệu 16 bit từ stack đưa vào toán hạng dst.

Ta có thể dùng nhiều lệnh PUSH để cất dữ liệu vào stack nhưng khi dùng lệnh POP để lấy dữ liệu ra thì phải dùng theo thứ tự ngược lại.

```
PUSH    AX
PUSH    BX
```

```
PUSH    CX
…
POP     CX
POP     BX
POP     AX
```

- Lệnh **XLAT [src]**: chuyển nội dung của ô nhớ 8 bit vào thanh ghi AL. Địa chỉ ô nhớ xác định bằng cặp thanh ghi DS:BX (nếu không chỉ ra src) hay src, địa chỉ offset chứa trong thanh ghi AL.

Lệnh XLAT tương đương với các lệnh:

```
MOV  AH,0
MOV  SI,AX
MOV  AL,[BX+SI]
```

7.1.2. Nhóm lệnh chuyển địa chỉ

- Lệnh **LEA reg16,mem16**: (Load Effective Address) chuyển địa chỉ offset của toán hạng bộ nhớ vào thanh ghi reg16.

Lệnh này sẽ tương đương với **MOV reg16, OFFSET mem16**

- Lệnh **LDS reg16,mem32**: (Load pointer using DS) chuyển nội dung bộ nhớ toán hạng mem32 vào cặp thanh ghi DS:reg16.

Lệnh LDS AX,mem tương đương với:

```
MOV  AX,mem
MOV  BX,mem+2
MOV  DS,BX
```

- Lệnh **LES reg16,mem32**: (Load pointer using ES) giống như lệnh LDS nhưng dùng cho thanh ghi ES

7.1.3. Nhóm lệnh chuyển cờ hiệu

- Lệnh **LAHF**: (Load AH from flag) chuyển các cờ SF, ZF, AF, PF và CF vào các bit 7,6,4,2 và 0 của thanh ghi AH (3 bit còn lại không đổi)

- Lệnh **SAHF**: (Store AH into flag) chuyển các bit 7,6,4,2 và 0 của thanh ghi AH vào các cờ SF, ZF, AF, PF và CF.

- Lệnh **PUSHF**: chuyển thanh ghi cờ vào stack

- Lệnh **POPF**: lấy dữ liệu từ stack chuyển vào thanh ghi cờ

7.1.4. Nhóm lệnh chuyển dữ liệu qua cổng

Mỗi I/O port giao tiếp với CPU sẽ có một địa chỉ 16 bit cho nó. CPU gởi hay nhận dữ liệu từ cổng bằng cách chỉ đến địa chỉ cổng đó. Tuỳ theo chức năng mà cổng có thể: chỉ đọc dữ liệu (input port), chỉ ghi dữ liệu (output port) hay có thể đọc và ghi dữ liệu (input/output port).

❖ Lệnh **IN**: đọc dữ liệu từ cổng và đưa vào thanh ghi AL

```
IN      AL,port8
IN      AL,DX
```

Nếu địa chỉ port chỉ có 8 bit thì có thể đưa giá trị trực tiếp vào, nếu là 16 bit thì phải thông qua thanh ghi AX.

❖ Lệnh **OUT**: ghi dữ liệu trong thanh ghi AL ra cổng

```
OUT   port8,AL
OUT   DX,AL
```

VD:
```
MOV  AL,3
OUT  61h,AL          ; Gởi giá trị 03h ra cổng 61h
MOV  AL,1
MOV  DX,03F8h        ; Xuất ra cổng máy in
OUT  DX,AL
MOV  DX,03F8h
IN   AL,DX           ; Đọc dữ liệu từ cổng máy in
```

7.2. Nhóm lệnh chuyển điều khiển

7.2.1. Lệnh nhảy không điều kiện JMP

JMP label
JMP reg/mem

Lệnh JMP dùng để chuyển điều khiển chương trình từ vị trí này sang vị trí khác (thay đổi nội dung cặp thanh ghi CS:IP).

7.2.2. Lệnh nhảy có điều kiện

Lệnh nhảy có điều kiện chỉ sử dụng cho các nhãn nằm trong khoảng từ –127 đến 128 byte so với vị trí của lệnh.

❖ Lệnh **JA label**: (Jump if Above)
Nếu CF = 0 và ZF = 0 thì JMP label

❖ Lệnh **JAE label**: (Jump if Above or Equal)
Nếu CF = 0 thì JMP label

❖ Lệnh **JB label**: (Jump if Below)
Nếu CF = 1 thì JMP label

❖ Lệnh **JBE label**: (Jump if Below or Equal)
Nếu CF = 1 hoặc ZF = 1 thì JMP label

❖ Lệnh **JNA label**: (Jump if Not Above)
Giống lệnh JBE

❖ Lệnh **JNAE label**: (Jump if Not Above or Equal)
Giống lệnh JB

❖ Lệnh **JNB label**: (Jump if Not Below)
Giống lệnh JAE

❖ Lệnh **JNBE label**: (Jump if Not Below or Equal)
Giống lệnh JA

❖ Lệnh **JG label**: (Jump if Greater)
Nếu SF = OF và ZF = 0 thì JMP label

❖ Lệnh **JGE label**: (Jump if Greater or Equal)
Nếu SF = OF thì JMP label

❖ Lệnh **JL label**: (Jump if Less)
Nếu SF <> OF thì JMP label

❖ Lệnh **JLE label**: (Jump if Less or Equal)
Nếu CF <> OF hoặc ZF = 1 thì JMP label

❖ Lệnh **JNG label**: (Jump if Not Greater)
Giống lệnh JLE

❖ Lệnh **JNGE label**: (Jump if Not Greater or Equal)
Giống lệnh JL

❖ Lệnh **JNL label**: (Jump if Not Less)
Giống lệnh JGE
❖ Lệnh **JNLE label**: (Jump if Not Less or Equal)
Giống lệnh JG

❖ Lệnh **JC label**: (Jump if Carry)
Giống lệnh JB

❖ Lệnh **JNC label**: (Jump if Not Carry)
Giống lệnh JNB

❖ Lệnh **JZ label**: (Jump if Zero)
Nếu ZF = 1 thì JMP label

❖ Lệnh **JE label**: (Jump if Equal)
Giống lệnh JZ

❖ Lệnh **JNZ label**: (Jump if Not Zero)
Nếu ZF = 0 thì JMP label

❖ Lệnh **JNE label**: (Jump if Equal)
Giống lệnh JNZ

❖ Lệnh **JS label**: (Jump on Sign)
Nếu SF = 1 thì JMP label

❖ Lệnh **JNS label**: (Jump if No Sign)
Nếu SF = 0 thì JMP label

❖ Lệnh **JO label**: (Jump on Overflow)
Nếu OF = 1 thì JMP label

❖ Lệnh **JNO label**: (Jump if No Overflow)
Nếu OF = 0 thì JMP label

❖ Lệnh **JP label**: (Jump on Parity)
Nếu PF = 1 thì JMP label

❖ Lệnh **JNP label**: (Jump if No Parity)
Nếu PF = 0 thì JMP label

❖ Lệnh **JCXZ label**: (Jump if CX Zero)
Nếu CX = 1 thì JMP label

7.2.3. Lệnh so sánh

CMP left(reg/mem), right(reg/mem/immed)

Lệnh CMP dùng để so sánh nội dung 2 toán hạng, kết quả chứa vào thanh ghi cờ và không làm thay đổi nội dung các toán hạng.

VD: Đoạn chương trình so sánh 2 số A và B: A >B thì nhảy đến label1, A = B thì nhảy đến label2, A < B thì nhảy đến label3.

```
MOV  AX,A
CMP  AX,B
JG   label1
JL   label2
JMP  label3
```

7.2.4. Các lệnh vòng lặp

❖ Lệnh **LOOP**:
LOOP label
Mô tả:
CX = CX – 1
Nếu CX <> 0 thì JMP label

❖ Lệnh **LOOPE**:
LOOPE label
Mô tả:
CX = CX – 1
Nếu (ZF = 1) và (CX <> 0) thì JMP label

❖ Lệnh **LOOPZ**:
Giống lệnh LOOPE

❖ Lệnh **LOOPNE**:
LOOPNE label
Mô tả:
CX = CX – 1
Nếu (ZF = 0) và (CX <> 0) thì JMP label

❖ Lệnh **LOOPNZ**:
Giống lệnh LOOPNE

7.2.5. Lệnh liên quan đến chương trình con

❖ Lệnh **CALL**:
Lệnh CALL dùng để gọi một chương trình con, có thể là near hay far.

```
CALL    label     ; Gọi chương trình con tại vị trí xác định
                  ; bởi nhãn label
CALL    reg/mem   ; Gọi chương trình con tại vị trí xác định
                  ; trong reg/mem
```

❖ Lệnh **RET**: (return)
RET [n]
RETN [n]
RETF [n]
Lệnh RET dùng để kết thúc chương trình con, điều khiển sẽ được đưa về địa chỉ trước khi gọi chương trình con. RETN để kết thúc chương trình con dạng near và RETF dùng để kết thúc chương trình con dạng far.

Trong trường hợp lệnh RET có hằng số n theo sau thì sẽ cộng với thanh ghi SP giá trị n (n phải là số chẵn). Lệnh này dùng để loại bỏ một số tham số chương trình con sử dụng ra khỏi stack.

7.3. Nhóm lệnh xử lý số học

7.3.1. Xử lý phép cộng

❖ Lệnh **ADD dst,src**:
dst ← dst + src
Toán hạng src có thể là reg, mem hay immed còn toán hạng dst là reg hay mem.

- Không thể cộng trực tiếp 2 thanh ghi đoạn
- Lệnh ADD ảnh hưởng đến các cờ sau:
  + Cờ CF: = 1 khi kết quả phép cộng có nhớ hay có mượn
  + Cờ AF: = 1 khi kết quả phép cộng có nhớ hay có mượn đối với 4 bit thấp
  + Cờ PF: = 1 khi kết quả phép cộng có tổng 8 bit thấp là một số chẵn.
  + Cờ ZF: = 1 khi kết quả phép cộng là 0.
  + Cờ SF: = 1 nếu kết quả phép cộng là một số âm
  + Cờ OF: = 1 nếu kết quả phép cộng bị sai dấu, nghĩa là vượt ra ngoài phạm vi lớn nhất hay nhỏ nhất mà số có dấu có thể chứa trong toán hạng dst.

❖ Lệnh **ADC dst, src**: (Add with Carry)

dst ← dst + src + CF

Lệnh ADC thường dùng để cộng các số lớn hơn 16 bit.

❖ Lệnh **INC dst**: (Increment)

dst ← dst + 1

Dst có thể là reg hay mem.

❖ Lệnh **AAA**: (ASCII Adjust for Addition)

Hiệu chỉnh kết quả phép cộng 2 số BCD dạng không nén (mỗi chữ số BCD lưu bằng 1 byte).

```
VD:   MOV  AX,9
      MOV  BX,3
      ADD  AL,BL          ; Kết quả là AX = 0Ch
      AAA                 ; AX = 0102h (AH = 1, AL = 2)
```

Lệnh AAA chỉ ảnh hưởng đến các cờ AF và CF, không ảnh hưởng đến các cờ còn lại.

❖ Lệnh **DAA**: (Decimal Adjust for Addition)

Hiệu chỉnh kết quả phép cộng 2 số BCD dạng nén (mỗi chữ số BCD lưu bằng 4 bit, nghĩa là 1 byte biểu diễn được các số nguyên từ 0 đến 99).

```
VD:   MOV  AX,4338h
      ADD  AL,AH        ; AX ← 437Bh
      DAA               ; AX ← 4381h (43 + 38 = 81)
```

Lệnh DAA chỉ ảnh hưởng đến các cờ AF, CF, PF, SF, ZF và không ảnh hưởng đến thanh ghi AH.

7.3.2. Xử lý phép trừ

❖ Lệnh **SUB dst,src**:

dst ← dst - src

Toán hạng src có thể là reg, mem hay immed còn toán hạng dst chỉ có thể là reg hay mem.

- Không thể trừ trực tiếp thanh ghi đoạn
- Ảnh hưởng đến các cờ AF, CF, OF, PF, SF và ZF.

❖ Lệnh **SBB dst,src**:

dst ← dst – src – CF

Lệnh ADC thường dùng để trừ các số lớn hơn 16 bit.

❖ Lệnh **DEC dst**: (decrement)

dst ← dst – 1

dst là reg hay mem. Lệnh DEC ảnh hưởng đến các cờ AF, OF, PF, SF, ZF.

❖ Lệnh **NEG dst**:

dst ← - *dst*

dst là reg hay mem.

Lệnh NEG ảnh hưởng đến các cờ:

CF = 1 nếu nội dung kết quả là số khác 0.

SF = 1 nếu nội dung kết quả là số âm khác 0.

PF = 1 nếu tổng 8 bit thấp là một số chẵn.

ZF = 1 nếu nội dung kết quả là 0.

OF = 1 nếu nội dung toán hạng dst là 80h (dạng byte) hay 8000h (dạng word).

VD: Nếu muốn thực hiện phép toán 100 – AH, ta không thể cùng lệnh:

```
SUB   100,AH
```

mà phải dùng lệnh:

```
SUB   AH,100
NEG   AH
```

❖ Lệnh **AAS**: (Ascii Adjust for Substract)

Hiệu chỉnh kết quả phép trừ 2 số BCD dạng không nén (mỗi chữ số BCD lưu bằng 1 byte). Lệnh AAS chỉ ảnh hưởng cờ AF và CF.

❖ Lệnh **DAS**: (Decimal Adjust for Substract)

Hiệu chỉnh kết quả phép trừ 2 số BCD dạng nén (mỗi chữ số BCD lưu bằng 4 bit). Lệnh AAS chỉ ảnh hưởng cờ AF và CF.

7.3.3. Xử lý phép nhân

❖ Lệnh **MUL src**:

Nếu src là reg hay mem 8 bit: AX ← AL*src

Nếu src là reg hay mem 16 bit: DX:AX ← AX*src

Lệnh MUL chỉ ảnh hưởng đến cờ CF và OF.

❖ Lệnh **IMUL src**:

Giống như lệnh MUL nhưng kết quả là số có dấu.

❖ Lệnh **AAM**: (Ascii Adjust for Multiple)

Hiệu chỉnh kết quả phép nhân 2 số BCD dạng không nén, lệnh AAM thực hiện chia AL cho 10, lưu phần thương vào AL và phần dư vào AH. Lệnh AAM ảnh hưởng đến các cờ PF, SF và ZF.

7.3.4. Xử lý phép chia

❖ Lệnh **DIV src**:

Nếu src là reg/mem 8 bit: AL ← AX DIV src và AH ← AX MOD src

Nếu src là reg/mem 16 bit: AX ← DX:AX DIV src và DX ← DX:AX MOD src

Lệnh DIV không ảnh hưởng đến các cờ nhưng xảy ra tràn trong các trường hợp sau:

- Chia cho 0

- Thương lớn hơn 256 đối với dạng 8 bit.
- Thương lớn hơn 65536 đối với dạng 16 bit.

❖ Lệnh **IDIV src**:

Giống như lệnh DIV nhưng kết quả là số có dấu. Các trường hợp tràn:

- Chia cho 0
- Thương nằm ngoài khoảng (-128,127) đối với dạng 8 bit.
- Thương nằm ngoài khoảng (-32767,32768) đối với dạng 16 bit.

❖ Lệnh **AAD**: (Ascii Adjust for Division)

Hiệu chỉnh kết quả phép chia 2 số BCD dạng không nén. Lưu ý rằng lệnh AAD phải được thực hiện trước lệnh chia. Sau khi thực hiện chia thì phải hiệu chỉnh lại dạng BCD bằng cách dùng lệnh AAM.

❖ Lệnh **CBW**: (Convert Byte to Word)

Nếu AL < 80h thì AH = 0, ngược lại AH = 0FFh

Lệnh CBW dùng để chuyển số nhị phân có dấu 8 bit thành số nhị phân có dấu 16 bit.

❖ Lệnh **CWD**: (Convert Word to Double word)

Nếu AX < 8000h thì DX = 0, ngược lại DX = 0FFFFh

Lệnh CWD dùng để chuyển số nhị phân có dấu 16 bit thành số nhị phân có dấu 32 bit chứa trong DX:AX.

7.3.5. Dịch chuyển và quay

❖ Lệnh **SHL**: (Shift Logical Left)

SHL dst,1
SHL dst,CL

Dịch trái 1 bit hay CL bit.

CF ← dst7 ← dst6 … ← dst0 ← 0

❖ Lệnh **SHR**: (Shift Logical Right)

SHR dst,1
SHR dst,CL

Dịch phải 1 bit hay CL bit.

0 → dst7 → dst6 … → dst0 → CF

❖ Lệnh **SAL**: giống SHL

❖ Lệnh **SAR**:

Giống như lệnh SHR nhưng giá trị bit dst7 không thay đổi, nghĩa là

dst7 → dst7 → dst6 … → dst0 → CF

❖ Lệnh **ROL**: (Rotate Left)

ROL dst,1
ROL dst,CL

Quay trái 1 bit hay CL bit.

CF ← dst7 ← dst6 … ← dst0 ← dst7

❖ Lệnh **ROR**: (Rotate Right)
ROR dst,1
ROR dst,CL
Quay phải 1 bit hay CL bit.
dst0 → dst7 → dst6 … → dst0 → CF

❖ Lệnh **RCL**: (Rotate though Carry Left)
RCL dst,1
RCL dst,CL
Quay trái 1 bit hay CL bit.
CF ← dst7 ← dst6 … ← dst0 ← CF

❖ Lệnh **RCR**: (Rotate though Carry Right)
RCR dst,1
RCR dst,CL
Quay phải 1 bit hay CL bit.
CF → dst7 → dst6 … → dst0 → CF

7.3.6. Các lệnh logic

❖ Lệnh **AND**:
AND dst,src
dst ← dst AND src
CF ← 0, OF ← 0
Src là reg, mem hay immed còn dst là reg, mem.

❖ Lệnh **OR**:
OR dst,src
dst ← dst OR src
CF ← 0, OF ← 0

❖ Lệnh **XOR**:
XOR dst,src
dst ← dst XOR src
CF ← 0, OF ← 0

❖ Lệnh **NOT**:
NOT dst
dst ← NOT dst
Lệnh NOT không ảnh hưởng đến các cờ.

❖ Lệnh **TEST**:
TEST dst,src
Lệnh TEST thực hiện phép toán AND 2 toán hạng nhưng chỉ ảnh hưởng đến các cờ và không ảnh hưởng đến toán tử.

7.4. Nhóm lệnh xử lý chuỗi

Bao gồm các lệnh sau:

- Lệnh **MOVS**: chuyển dữ liệu từ vùng nhớ này sang vùng nhớ khác.

+ MOVSB: chuyển 1 byte từ vị trí chỉ đến bởi SI đến vị trí chỉ bởi DI. Nếu DF = 0 thì SI ← SI + 1, DI ← DI + 1 còn nếu DF = 1 thì SI ← SI - 1, DI ← DI - 1.

+ MOVSW: chuyển 1 word từ vị trí chỉ đến bởi SI đến vị trí chỉ bởi DI. Nếu DF = 0 thì SI ← SI + 2, DI ← DI + 2 còn nếu DF ← 1 thì SI ← SI - 2, DI ← DI - 2.

- Lệnh **CMPS**: so sánh nội dung 2 vùng nhớ

+ CMPSB: so sánh 1 byte tại vị trí chỉ đến bởi SI và tại vị trí chỉ bởi DI. Nếu DF = 0 thì SI ← SI + 1, DI ← DI + 1 còn nếu DF ← 1 thì SI ← SI - 1, DI ← DI - 1.

+ CMPSW: so sánh 1 word tại vị trí chỉ đến bởi SI và tại vị trí chỉ bởi DI. Nếu DF = 0 thì SI ← SI + 2, DI ← DI + 2 còn nếu DF = 1 thì SI ← SI - 2, DI ← DI - 2.

- Lệnh **SCAS**: tìm một phần tử trong vùng nhớ, địa chỉ vùng nhớ xác định bằng cặp thanh ghi ES:DI, giá trị cần tìm đặt trong thanh ghi AL, nếu tìm thấy thì ZF = 1. Giá trị của DI và SI thay đổi giống như trên.

- Lệnh **LODS**: đưa một byte hay word có địa chỉ xác định bởi cặp thanh ghi DS:SI vào thanh ghi AL hay AX. Giá trị của DI và SI thay đổi giống như trên.

- Lệnh **STOS**: chuyển nội dung của AL hay AX vào vùng nhớ xác định bởi cặp thanh ghi ES:DI. Giá trị của DI và SI thay đổi giống như trên.

8. Các cấu trúc cơ bản trong lập trình hợp ngữ

8.1. Cấu trúc tuần tự

Cấu trúc tuần tự là cấu trúc đơn giản nhất. Trong cấu trúc tuần tự, các lệnh được sắp xếp tuần tự, lệnh này tiếp theo lệnh kia.

Lệnh 1
Lệnh 2
…
Lệnh n

VD: Cộng 2 giá trị của thanh ghi BX và CX, rồi nhân đôi kết quả, kết quả cuối cùng chứa trong AX

```
MOV  AX,BX
ADD  AX,CX      ; Cộng BX với CX
SHL  AX,1       ; Nhân đôi
```

8.2. Cấu trúc IF – THEN, IF – THEN – ELSE

IF Điều kiện THEN Công việc
IF Điều kiện THEN Công việc1 ELSE Công việc2
VD: Gán BX = |AX|

```
            CMP  AX,0        ; AX > 0?
            JNL  DUONG       ; AX dương
            NEG  AX          ; Nếu AX < 0 thì đảo dấu
DUONG:      MOV  BX,AX
NEXT:
```

VD: Gán CL giá trị bit dấu của AX

```
            CMP  AX,0        ; AX > 0?
            JNS  AM          ; AX âm
            MOV  CL,1        ; CL = 1 (AX dương)
            JMP  NEXT
        AM: MOV  CL,0        ; CL = 0 (AX âm)
        NEXT:
```

8.3. Cấu trúc CASE

CASE Biểu thức
Giá trị 1: Công việc 1
Giá trị 2: Công việc 2
…
Giá trị n: Công việc n
END

VD: Nếu AX > 0 thì BH = 0, nếu AX < 0 thì BH = 1. Ngược lại BH = 2

```
        CMP  AX,0
             JL   AM
        JE   KHONG
             JG   DUONG
DUONG:  MOV  BH,0
        JMP  NEXT
AM:     MOV  BH,1
        JMP  NEXT
KHONG:  MOV  BH,2
NEXT:
```

8.4. Cấu trúc FOR

FOR Số lần lặp DO Công việc

VD: Cho vùng nhớ M dài 200 bytes trong đoạn dữ liệu, chương trình đếm số chữ A trong vùng nhớ M như sau:

```
            MOV  CX,200          ; Đếm 200 bytes
            MOV  BX,OFFSET M     ; Lấy địa chỉ vùng nhớ
            XOR  AX,AX           ; AX = 0
```

```
NEXT:   CMP  BYTE PTR [BX],'A'; So sánh với chữ A
        JNZ  ChuA              ; Nếu không phải là chữ A thì tiếp
        INC  AX                ; tục, ngược lại thì tăng AX
ChuA:   INC  BX
        LOOP      NEXT
```

8.5. Cấu trúc lặp WHILE

WHILE Điều kiện DO Công việc

VD: Chương trình đọc vùng nhớ bắt đầu tại địa chỉ 1000h vào thanh ghi AH, đến khi gặp ký tự '$' thì thoát:

```
        MOV  BX,1000h
CONT:   CMP  AH,'$'
        JZ   NEXT
        MOV  AH,DS:[BX]
        JMP  CONT
NEXT:
```

8.6. Cấu trúc lặp REPEAT

REPEAT Công việc UNTIL Điều kiện

VD: Chương trình đọc vùng nhớ bắt đầu tại địa chỉ 1000h vào thanh ghi AH, đến khi gặp ký tự '$' thì thoát:

```
        MOV  BX,1000h
CONT:   MOV  AH,DS:[BX]
        CMP  AH,'$'
        JZ   NEXT
        JMP  CONT
NEXT:
```

9. Các ngắt của 8086

Bảng 3.4:

Vector ngắt	Công dụng
00h	CPU: tác động khi chia cho 0
01h	CPU: chương trình thực thi từng bước
02h	CPU: ngắt không che được
03h	CPU: tạo điểm dừng chochương trình
04h	CPU: tác động khi kết quả số học tràn
05h	Tác động khi nhấn Print Screen
06h - 07h	Dành riêng
08h	Tác động bởi nhịp đồng hồ (18.2 lần/s)
09h	Tác động khi có phím nhấn
0Ah	Dành riêng
0Bh - 0Ch	Tác động phần cứng liên lạc nối tiếp

0Dh	Đĩa cứng
0Eh	Đĩa mềm
0Fh	Máy in
10h	BIOS: màn hình
11h	BIOS: xác định cấu hình máy tính
12h	BIOS: thông báo kích thước RAM
13h	BIOS: gọi các phục vụ đĩa cứng/mềm
14h	BIOS: giao tiếp nối tiếp
15h	BIOS: truy xuất cassette hay mở rộng ngắt
16h	BIOS: xuất / nhập bàn phím
17h	BIOS: máy in
18h	Xâm nhập ROM basic
19h	BIOS: khởi động máy tính
1Ah	BIOS: ngày / giờ hệ thống
1Bh	Lấy điều khiển từ ngắt bàn phím
1Ch	Lấy điều khiển từ ngắt đồng hồ (sau int 08h)
1Dh	Địa chỉ bảng tham số màn hình
1Eh	Địa chỉ bảng tham số đĩa
1Fh	Địa chỉ bộ mã ký tự
20h	DOS: kết thúc chương trình
21h	DOS: các chức năng DOS
22h	Địa chỉ cần chuyển khi kết thúc chương trình
23h	Địa chỉ cần chuyển khi gặp Ctrl – Break
24h	Địa chỉ cần chuyển khi gặp lỗi
25h	DOS: đọc đĩa cứng / mềm
26h	DOS: ghi đĩa cứng / mềm
27h	DOS: chấm dứt chương trình và thường trú
28h – 3Fh	Dành riêng cho DOS
40h	BIOS: các chức năng đĩa mềm
41h	Bảng thông số đĩa cứng thứ nhất
42h – 45h	Dành riêng
46h	Bảng thông số đĩa cứng thứ hai
47h – 49h	Định nghĩa do người sử dụng
4Ah	Giờ báo hiệu (chỉ trong AT)
4Bh – 67h	Định nghĩa do người sử dụng
68h – 6Fh	Không sử dụng
70h	Đồng hồ thời gian thực (chỉ trong AT)
71h – 7Fh	Dành riêng
80h – 85h	Dành riêng
86h – F0h	Sử dụng bởi chương trình thông dịch BASIC
F1h – FFh	Không sử dụng

9.1. Ngắt 21h

❖ **Hàm 01h**: nhập một ký tự từ bàn phím và hiện ký tự nhập ra màn hình. Nếu không có ký tự nhập, hàm 01h sẽ đợi cho đến khi nhập.

- Gọi: AH = 01h
- Trả về: AL chứa mã ASCII của ký tự nhập

```
MOV  AH,01h
INT  21h          ; AL chứa mã ASCII của ký tự nhập
```

❖ **Hàm 02h**: xuất một ký tự trong thanh ghi DL ra màn hình tại vị trí con trỏ hiện hành

- Gọi AH = 02h, DL = mã ASCII của ký tự
- Trả về: không có

```
MOV  AH,02h
MOV  DL,'A'
INT  21h
```

❖ **Hàm 08h**: giống hàm 01h nhưng không hiển thị ký tự ra màn hình

❖ **Hàm 09h**: xuất một chuỗi ký tự ra màn hình tại vị trí con trỏ hiện hành, địa chỉ chuỗi được chứa trong DS:DX và phải được kết thúc bằng ký tự $

- Gọi AH = 09h, DS:DX = địa chỉ chuỗi
- Trả về: không có

```
.DATA
Msg  DB   'Hello$'
…
MOV  AH,09h
LEA  DX,Msg
INT  21h
```

❖ **Hàm 0Ah**: nhập một chuỗi ký tự từ bàn phím (tối đa 255 ký tự), dùng phím ENTER kết thúc chuỗi

- Gọi AH = 0Ah, DS:DX = địa chỉ lưu chuỗi
- Trả về: không có

Chuỗi phải có dạng sau:

- Byte 0: Số byte tối đa cần đọc (kể cả ký tự Enter)
- Byte 1: số byte đã đọc
- Byte 2: lưu các ký tự đọc

```
.DATA
Msg  DB   101          ; Đọc tối đa 100 ký tự
     DB   ?
     DB   101  DUP(?)
     …
     MOV  AH,0Ah
     LEA  DX,Msg
     INT  21h
```

❖ **Hàm 4Ch**: kết thúc chương trình

```
MOV  AH,4Ch
INT  21h
```

9.2. Ngắt 10h

❖ **Xoá màn hình:**

- Gọi AX = 02h
- Trả về: không có

```
MOV  AX,02h
INT  10h
```

❖ **Chuyển toạ độ con trỏ:**

- Gọi AH = 02h, DH = dòng, DL = cột

```
MOV  AH,02h
MOV  DX,0F15h
INT  10h
```

10. Truyền tham số giữa các chương trình

Trong lập trình, một vấn đề ta cần quan tâm là truyền tham số giữa chương trình chính và chương trình con. Để thực hiện truyền tham số, ta có thể dùng các cách sau đây:

- Truyền tham số qua thanh ghi
- Truyền tham số qua ô nhớ (biến)
- Truyền tham số qua ô nhớ do thanh ghi chỉ đến
- Truyền tham số qua stack

10.1. Truyền tham số qua thanh ghi

Ta thực hiện truyền tham số qua thanh ghi bằng cách: một chương trình con sẽ đưa giá trị vào thanh ghi và chương trình con khác sẽ xử lý giá trị trên thanh ghi đó.

VD: Cộng giá trị tại 2 ô nhớ 1000h và 1001h, kết quả chứa trong 1002h (bye cao) và 1003h (byte thấp).

```
.MODEL    SMALL
.STACK    100h
.CODE
main  PROC
      MOV       AX,@DATA
      MOV       DS,AX
      MOV       BYTE PTR DS:[1000h],10h      ; Đưa giá trị vào
      MOV       BYTE PTR DS:[1001h],0FFh     ; các ô nhớ
      CALL      Read
      CALL      Sum
      Mov       AH,4Ch
      INT       21h
main  ENDP
Read  PROC                    ; Đọc dữ liệu vào thanh ghi AX
      MOV       AH,DS:[1000h]
      MOV       AL,DS:[1001h]
      RET
Read  ENDP                    ; Xử lý dữ liệu tại thanh ghi AX
```

```
Sum   PROC
      ADD       AH,AL
      JZ        next
      MOV       DS:[1003h],1
next: MOV       DS:[1002h],AH
RET
Sum   ENDP
END   main
```

10.2. Truyền tham số qua ô nhớ (biến)

Quá trình truyền tham số cũng giống như trên nhưng thay vì thực hiện thông qua thanh ghi, ta sẽ thực hiện thông qua các ô nhớ.

VD: Cộng giá trị tại 2 ô nhớ m1 và m2, kết quả chứa trong m3 (bye cao) và m4 (byte thấp).

```
.MODEL    SMALL
.STACK    100h
.DATA
      m1    db    ?
      m2    db    ?
      m3    db    ?
      m4    db    ?
.CODE
main  PROC
      MOV       AX,@data
      MOV       DS,AX
      MOV       m1,10h      ; Đưa giá trị vào
      MOV       m2,0FFh     ; các ô nhớ
      CALL      Sum
      MOV       AH,4Ch
      INT       21h
main  ENDP
Sum   PROC
      MOV       m4,0
      MOV       AH,m1
      ADD       AH,m2
      JNC       next
      MOV       m4,1
next: MOV       m3,AH
RET
Sum   ENDP
END   main
```

10.3. Truyền tham số qua ô nhớ do thanh ghi chỉ đến

Trong cách truyền tham số này, ta dùng các thanh ghi SI, DI, BX để chỉ địa chỉ offset của các tham số còn thanh ghi đoạn mặc định là DS.

VD: Cộng giá trị tại 2 ô nhớ m1 và m2, kết quả chứa trong m3 (bye cao) và m4 (byte thấp).

```
.MODEL    SMALL
.STACK    100h
.DATA
        m1    db    ?
        m2    db    ?
        m3    db    ?
        m4    db    ?
.CODE
main    PROC
        MOV       AX,@data
        MOV       DS,AX
        LEA       SI,m1
        LEA       DI,m2
        LEA       BX,m3
        MOV       [SI],10h      ; Đưa giá trị vào
        MOV       [DI],0FFh     ; các ô nhớ
        CALL      Sum
        MOV       AH,4Ch
        INT       21h
main    ENDP
Sum     PROC
        MOV       AL,[SI]
        ADD       AL,[DI]
        JZ        next
        MOV       [BX+1],1
next:   MOV       [BX],AL
RET
Sum     ENDP
END     main
```

10.4. Truyền tham số qua stack

Trong phương pháp truyền tham số này, ta dùng stack làm nơi chứa các tham số cần truyền thông qua các tác vụ PUSH và POP.

VD: Cộng giá trị tại 2 ô nhớ m1 và m2, kết quả chứa trong m3 (byte cao) và m4 (byte thấp).

```
.MODEL    SMALL
.STACK    100h
.DATA
        m1    dw    ?
        m2    dw    ?
        m3    dw    ?
        m4    dw    ?
.CODE
main    PROC
        MOV       AX,@data
```

```
        MOV     DS,AX
        LEA     SI,m1
        LEA     DI,m2
        MOV     [SI],1234h          ; Đưa giá trị vào
        MOV     [DI],0FEDCh         ; các ô nhớ
        PUSH    m1                  ; Đưa vào stack
        PUSH    m2
        CALL    Sum
        POP     m3                  ; Lấy kết quả đưa vào stack
        POP     m4
        MOV     AH,4Ch
        INT     21h
main    ENDP
Sum     PROC
        POP     DX    ; Lưu lại địa chỉ trả về của lệnh CALL
        POP     AX    ; Lấy dữ liệu từ stack
        POP     BX
        ADD     AX,BX
        JNC     next
        PUSH    1
next:   PUSH    AX
        PUSH    DX    ; Trả lại địa chỉ trở về của lệnh CALL
RET
Sum     ENDP
END     main
```

11. Các ví dụ minh hoạ

11.1. In chuỗi ký tự ra màn hình

```
.MODEL  SMALL
.STACK  100h
.DATA
        msg     DB      'Hello$'
.CODE
main    PROC
        MOV     AX,@DATA        ; Khởi động thanh ghi DS
        MOV     DS,AX
        MOV     AX,02h          ; Xoá màn hình
        INT     10h
        MOV     AH,02h          ; Chuyển toạ độ con trỏ
        MOV     DX,0C15h        ; đến dòng 12 (0Ch) và cột 21 (15h)
        INT     10h
        LEA     DX,msg          ; Địa chỉ thông điệp
        MOV     AH,09h          ; In thông điệp ra màn hình
        INT     21h
        MOV     AH,4Ch          ; Kết thúc chương trình
        INT     21h
main    ENDP
```

```
END   main
```

11.2. In chuỗi ký tự ra màn hình tại toạ độ nhập vào

```
.MODEL      SMALL
.STACK      100h
.DATA
        msg     DB      'Hello$'
        msg1    DB      'Nhap vao toa do:$'
        Crlf    DB      0Dh,0Ah,'$'
        Td      DB      3
                DB      ?
                DB      3       DUP(?)
.CODE
main    PROC
        MOV     AX,@DATA
        MOV     DS,AX               ; Khởi động thanh ghi DS
        MOV     AX,02h
        INT     10h                 ; Xóa màn hình
        LEA     DX,msg1
        MOV     AH,09h              ; In thông điệp
        INT     21h
        CALL    Nhap                ; Nhập dòng
        MOV     CL,AL
        LEA     DX,Crlf             ; Xuống dòng
        MOV     AH,09h
        INT     21h
        CALL    Nhap                ; Nhập cột
        MOV     CH,AL
        MOV     AH,02h              ; Chuyển tọa độ con trỏ
        MOV     DX,CX
        INT     10h
        LEA     DX,msg
        MOV     AH,09h              ; In ra màn hình
        INT     21h
        MOV     AH,4Ch              ; Kết thúc chương trình
        INT     21h
main    ENDP
Nhap    PROC
        MOV     AH,0Ah              ; Nhập vào
        LEA     DX,Td
        INT     21h
        LEA     BX,Td               ; Lấy chữ số hàng chục
        MOV     AL,DS:[BX+2]
        SUB     AL,'0'              ; Chuyển từ dạng ký tự sang dạng số
        MOV     BL,10
        MUL     BL                  ; Nhân số hàng chục với 10
        PUSH    AX
        LEA     BX,Td               ; Lấy chữ số hàng dơn vị
```

```
        MOV  AL,DS:[BX+3]
        SUB  AL,'0'
        POP  BX
        ADD  AL,BL
        RET
Nhap    ENDP
END     main
```

11.3. Cộng 2 số nhị phân dài 5 byte

```
.MODEL      SMALL
.STACK      100h
.DATA
        m1      DB      00h,08h,10h,13h,24h,00h
        m2      DB      0FFh,0FCh,0FAh,0F0h,0F1h,00h;
        m3      DB      6       DUP(0)
.CODE
main    PROC
        MOV  AX,@DATA
        MOV  DS,AX              ; Khởi động thanh ghi DS
        LEA  SI,m1
        LEA  DI,m2
        LEA  BX,m3
        MOV  CX,6
        XOR  AL,AL
next:   MOV  AL,[SI]
        ADC  AL,[DI]
        MOV  [BX],AL
        INC  BX
        INC  SI
        INC  DI
        LOOP next
        MOV  AH,4Ch
        INT  21h
main    ENDP
END     main
```

11.4. Nhập một chuỗi ký tự và chuyển chữ thường thành chữ hoa

```
.MODEL      SMALL
.STACK      100h
.DATA
        m1      DB      81
                DB      ?
                DB      81      DUP(?)
        m2      DB      'Chuoi da doi:$'
.CODE
main    PROC
        MOV  AX,@DATA
```

```
        MOV  DS,AX          ; Khởi động thanh ghi DS
        MOV  ES,AX
        LEA  DX,m1
        MOV  AH,0Ah         ; Nhập chuỗi
        INT  21h
        LEA  SI,m1          ; Lấy địa chỉ chuỗi
        ADD  SI,2
        MOV  DI,SI          ; Chuỗi nguồn và đích trùng nhau
Next:   LODSB               ; Lấy ký tự
        CMP  AL,0Dh         ; Nếu là ký tự Enter thì kết thúc
        JE   quit
        CMP  AL,'a'         ; Nếu ký tự nhập không phải là ký tự
        JB   cont           ; thường từ 'a' đến 'z' thì bỏ qua
        CMP  AL,'z'
        JA   cont
        SUB  AL,20h         ; Chuyển ký tự thường thành ký tự hoa
        STOSB               ; Lưu ký tự vừa chuyển
        DEC  DI             ; Nếu là ký tự thường thì dùng lệnh STOSB
                            ; nên DI tăng lên 1, ta phải giảm DI
cont:   INC  DI             ; Tăng lên ký tự kế
        JMP  next
quit:   MOV  AL,'$'
        STOSB
        MOV  AX,02h         ; Xóa màn hình
        INT  10h
        LEA  DX,m2
        MOV  AH,09h
        INT  21h
        LEA  DX,m1+2
        MOV  AH,09h
        INT  21h
        MOV  AH,4Ch
        INT  21h
main    ENDP
END     main
```

Giáo trình Hợp ngữ

Chương 1 : CƠ BẢN VỀ HỢP NGỮ

Trong chương này sẽ giới thiệu những nguyên tắc chung để tạo ra , dịch và chạy một chương trình hợp ngữ trên máy tính .

Cấu trúc ngữ pháp của lệnh hợp ngữ trong giáo trình này được trình bày theo Macro Assembler (MASM) dựa trên CPU 8086 .

1.1 Cú pháp lệnh hợp ngữ

Một chương trình hợp ngữ bao gồm một loạt các mệnh đề (statement) được viết liên tiếp nhau , mỗi mệnh đề được viết trên 1 dòng .

Một mệnh đề có thể là :

- Một lệnh (instruction) : được trình biên dịch (Assembler =ASM) chuyển thành mã máy.
- Một chỉ dẫn của Assembler (Assembler directive) : ASM không chuyển thành mã máy

Các mệnh đề của ASM gồm 4 trường :

Name Operation Operand(s) Comment

các trường cách nhau ít nhất là một ký tự trống hoặc một ký tự TAB

ví dụ lệnh đề sau :

START : MOV CX,5 ; khơi tạo thanh ghi CX

Sau đây là một chỉ dẫn của ASM :

MAIN PROC ; tạo một thủ tục có tên là MAIN

1.1.1 Trường Tên (Name Field)

Trường tên được dùng cho nhãn lệnh , tên thủ tục và tên biến . ASM sẽ chuyển tên thành địa chỉ bộ nhớ .

Tên có thể dài từ 1 đến 31 ký tự . Trong tên chứa các ký tự từ a-z , các số và các ký tự đặc biệt sau : ? ,@ , _ , $ và dấu . Không được phép có ký tự trống trong phần tên .

Nếu trong tên có ký tự . thì nó phải là ký tự đầu tiên . Tên không được bắt đầu bằng một số . ASM không phân biệt giữa ký tự viết thường và viết hoa . Sau đây là các ví dụ về tên hợp lệ và không hợp lệ trong ASM .

Tên hộp lệ	Tên không hợp lệ
COUNTER1	TWO WORDS
@CHARACTER	2ABC
SUM_OF_DIGITS	A45.28
DONE?	YOU&ME
.TEST	ADD-REPEAT

1.1.2 Trường toán tử (operation field)

Đối với 1 lệnh trường toán tử chứa ký hiệu (sumbol) 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 mô tả chức năng của phép toán , ví dụ ADD , SUB , INC , DEC , INT ...

Đối với chỉ dẫn của ASM , trường toán tử chứa một opcode giả (pseudo operation code = pseudo-op) . ASM không chuyển pseudo-op thành mã máy mà hướng dẫn ASM thực hiện một việc gì đó ví dụ tạo ra một thủ tục , định nghĩa các biến ...

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

Trong một lệnh trường toán hạng chỉ ra 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 thường làthanh ghi hoặc vị trí nhớ dùng để lưu trữ kết qua . Toán hạng thứ hai là toán hạng nguồn . Toán hạng nguồn thường không bị thay đổi sau khi thực hiện lệnh .

Đối với một chỉ dẫn của ASM , trường toán hạng chứa một hoặc nhiều thông tin mà ASM dùng để thực thi chỉ dẫn .

1.1.4 Trường chú thích (comment field)

Trường chú thích là một tuỳ chọn của mệnh đề trong ngôn ngữ ASM . Lập trình viên dùng trường chú thích để thuyết minh về câu lệnh . Điều này là cần thiết vì ngôn ngữ ASM là ngôn ngữ cấp thấp (low level) vì vậy sẽ rất khó hiểu chương trình nếu nó không được chú thích một cách đầy đủ và rỏ ràng . Tuy nhiên không nên có chú thích đối với mọi dòng của chương trình , kể cả nnhững lệnh mà ý nghĩa của nó đã rất rỏ ràng như :

```
NOP             ; không làm chi cả
```

Người 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 cuả chương trình ,ví dụ :

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

1.2 Các kiểu số liệu trong chương trình hợp ngữ

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 hợp ngữ cho phép biểu diễn số liệu dưới dạng nhị phân, thập phân hoặc thập lục phân và thậm chí là cả ký tự nửa .

1.2.1 Các số

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

Một số thập phân là một dãy các chữ só 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 hợp lệ và không hợp lệ trong ASM :

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

1.2.2 Các ký tự

Ký tự và một chuỗi các ký tự phải được đóng giữa hai dấu ngoặc đơn hoặc hai dấu ngoặc kép . Ví dụ ‘A’ và “HELLO” . Các ký tự đều được 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 như trong ngôn ngữ cấp cao . Mỗi biến có một loại dữ liệu và nó được gán một địa chỉ bộ nhớ sau khi dịch chương trình . Bảng sau đây liệt kê các toán tử giả dùng để định nghĩa các loại số liệu .

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

1.3.1. Biến byte

Chỉ dẫn của ASM để định nghĩa biến byte có dạng như sau :

NAME DB initial_value

Ví dụ :

ALPHA DB 4

Chỉ dẫn này sẽ gán tên ALPHA cho một byte nhớ trong bộ nhớ mà giá trị ban đầu của nó là 4 . Nếu giá trị của byte là không xác định thì đặt dấu chấm hỏi (?) vào giá trị ban đầu . Ví dụ :

BYT DB ?

Đối với biến byte vùng giá trị khả dĩ mà nó lưu trữ được là -128 đến 127 đối với số có dấu và 0 đến 255 đối với số không dấu .

1.3.2 Biến từ

Chỉ dẫn của ASM để định nghĩa một biến từ như 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 đối với số có dấu và 0 đến 56535 đối với số không dấu.

1.3.3 Mảng (arrays)

Trong ASM một mảng là một loạt các byte nhớ hoặc từ nhớ liên tiếp nhau . Ví dụ để 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 được gán cho byte đầu tiên

B_ARRAY+1 là tên của byte thứ hai

B_ARRAY+2 là tên của byte thứ ba

Nếu ASM gán địa chỉ offset là 0200h cho mảng B_ARRAY thì nội dung bộ nhớ sẽ như sau :

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

Chỉ dẫn sau đây sẽ định nghĩa một mảng 4 phần tử có tên là W_ARRAY:

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

Giả sử mảng bắt đầu tại 0300h thì bộ nhớ sẽ như 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

Đôi khi chúng ta cần truy xuất tới byte thấp và byte cao của một biến từ . Giả sử chúng ta định nghĩa :

```
WORD1       DW      1234h
```

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

Chuỗi các ký tự (character strings)

Một mảng các mã ASCII có thể được định nghĩa bằng một chuỗi các ký tự

Ví dụ :

```
LETTERS     DW      41h,42h,43h
```

tương đương với

```
LETTERS     DW      'ABC '
```

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

Trong ASM cũng có thể tổ hợp các ký tự và các số trong một định nghĩa . Ví dụ:

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

tương đương với

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

1.4 Các hằng (constants)

Trong một chương trình các hằng có thể được đặt tên nhờ chỉ dẫn EQU (equates) . Cú pháp của EQU là :

```
NAME    EQU     constant
```

ví dụ :

```
LF      EQU     0AH
```

sau khi có khai báo trên thì LF được dùng thay cho 0Ah trong chương trình. Vì vậy ASM sẽ chuyển các lệnh :

```
MOV   DL,0Ah
```

và

```
MOV   DL,LF
```

thành cùng một mã máy .

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

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

Sau khi có 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 đơn giản của 8086 mà chúng thường được dùng với các thao tác di chuyển số liệu và thực hiện các phép toán số học .

Trong phần sau đây , 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 di 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 đổi

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
MemoryLocation	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 như lệnh MOV có một số hạn chế đối với lệnh XCHG như 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 được dùng để cộng và trừ nội dung của 2 thanh ghi , của một thanh ghi và một vị trí nhớ , hoặc cộng (trừ) một số với (khỏi) một thanh ghi hoặc một vị trí nhớ . Cú pháp là :

ADD Destination , Source

SUB Destination , Source

Ví dụ :

```
ADD WORD1, AX
ADD BL , 5
SUB AX,DX ; AX=AX-DX
```

Vì lý do kỹ thuật , lệnh ADD và SUB cũng bị một số hạn chế như bảng sau:

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

Việc cộng hoặc trừ trực tiếp giữa 2 vị trí nhớ là không được phép . Để giải quyết vấn đề này người ta phải di chuyển byte (từ) nhớ đến một thanh ghi sau đó mới cộng hoặc trừ thanh ghi này với một byte (từ) nhớ khác . Ví dụ:

```
MOV AL, BYTE2
ADD BYTE1, AL
```

Lệnh INC (incremrent) để cộng thêm 1 vào nội dung của một thanh ghi hoặc một vị trí nhớ . Lệnh DEC (decrement) để giảm bớt 1 khỏi một thanh ghi hoặc 1 vị trí nhớ . Cú pháp của chúng là :

```
INC Destination
DEC Destination
```

Ví dụ :

```
INC WORD1
INC AX
DEC BL
```

1.5.3 Lệnh NEG (negative)

Lệnh NEG để 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 đây 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 ; đưa A vào AX
MOV B,AX ; đưa AX vào B
```

1.6.2 Mệnh đề A=5-A

```
MOV AX,5 ; đưa 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 bộ nhớ . Mỗi một đoạn chương trình là được chuyển thành một đoạn bộ nhớ bởi ASM .

1.7.1 Các kiểu bộ nhớ (memory models)

Độ lớn của mã và số liệu trong một chương trình được quy định bởi chỉ dẫn MODEL nhằm xác định kiểu bộ nhớ dùng với chương trình . Cú pháp của chỉ dẫn MODEL như sau :

```
.MODEL    memory_model
```

Bảng sau cho thấy các kiểu bộ nhớ :

MODEL	DESCRITION
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 dayta lớn hơn 1 đoạn , array không qúa 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 ... Để bắt đầu đoạn số liệu chúng ta dùng chỉ dẫn DATA với cú pháp như sau :

```
.DATA
;khai báo tên các biến , hằng và mãng
```

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 satck) 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 chỉ dẫn CODE như sau :

```
.CODE
```

Bên trong đoạn mã các lệnh thường được tổ chức thành thủ tục (procedure) mà cấu trúc của một thủ tục như sau :

```
name PROC
; body of the procedure
name ENDP
```

Sau đây là câú trúc của một chương trình hợp ngữ mà phần CODE là thủ tục có tên là MAIN

```
.MODEL SMALL
.STACK 100h
.DATA
; định nghĩa số liệu tại đây
.CODE
MAIN PROC
;thân 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 địa chỉ cổng thay đổi tuỳ theo loại máy tính
- có thể lập trình cho các IO dễ dàng hơn nhờ các chương trình con (routine) được 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 các hàm (function) của DOS . Tuỳ 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

FUNTION 1 : Single key input

Input : AH=1

Output: AL= ASCII code if character key is pressed

AL=0 if non character key is pressed

Để 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
```

FUNTION 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 điều khiển .Nếu DL chứa ký tự điều khiển thì khi gọi INT 21h , ký tự điều khiển sẽ được thực hiện .
Các ký tự điều khiển thường dùng là :

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 đầu tiên

Chúng ta sẽ viết một chương trình hợp ngữ nhằm đọc một ký tự từ bàn phím và in nó trên đầu dòng mới .

```
TITLE PGM1: ECHO PROGRAM
        .MODEL SMALL
        .STACK 100H
        .CODE
        MAIN PROC
        ; display dấu nhắc
                MOV AH,2
                MOV DL,'?'
                INT 21H
        ; nhập 1 ký tự
                MOV AH,1 ; hàm đọc ký tự
                INT 21H ; ký tự được đưa vào AL
                MOV BL,AL ; cất ký tự trong BL
        ; nhảy đến dòng mới
                MOV AH,2 ; hàm xuất 1 ký tự
                MOV DL,0DH ; ký tự carriage return
                INT 21H , thực hiện carriage return
                MOV DL,0AH ; ký tự line feed
                INT 21H ; thực hiện line feed
        ; xuất ký tự
                MOV DL,BL ; đưa ký tự vào DL
                INT 21H ; xuất ký tự
        ; trở về DOS
                MOV AH,4CH ; hàm thoát về DOS
                INT 21H ; exit to DOS
        MAIN ENDP
                END MAIN
```

1.10 Tạo ra và chạy một chương trình hợp ngữ

Có 4 bước để tạo ra và chạy một chương trình hợp ngữ là :

- Dùng một trình soạn thảo văn bản để tạo ra tập tin chương trình nguồn (source program file) .
- Dùng một trình biên dịch (Assembler) để tạo ra tập tin đối tượng (object file) ngôn ngữ máy
- Dùng trình LINK để liên kết một hoặc nhiều tập tin đối tượng rồi tạo ra file thực thi được .
- Cho thực hiện tập tin EXE hoặc COM .

Bước 1 : Tạo ra chương trình nguồn

Dùng một trình soạn thảo văn bản (NC chẳng hạn) để tạo ra chương trình nguồn .Ví dụ lất tên là PGM1.ASM. Phần mở rộng ASM là phần mở rộng quy ước 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 đối tượng ngôn ngữ máy goị 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 inra số dòng bị lỗi và một mộ tả ngắn về lỗi đó .

Nếu không có lỗi thì MASM sẽ chuyển PGM1.ASM thành tậo tin đối tượng ngôn ngữ máy gọi là PGM1.OBJ .

Dấu chấm phẩy sau lệnh MASM PGM1 có nghĩa là chúng ta không muốn tạo ra một tập tin đối tượng có tên khác với PGM1 . Nếu không có dấu chấm phẩy 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 nhiên là NUL có nghĩa là không tạo ra file tương ứng trừ khi lập trình viên gõ vào tên tập tin .

Tập tin danh sách nguồn (source listing file) : là một tập tin Text có đánh số dòng , trong đó mã hợp ngữ và mã nguồn nằm cạnh nhau . Tập tin này thường dùng 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 đuợc dùng để tìm các biến và nhãn trong một chương trình lớn .

Bước 3 : Liên kết chương trình

Tập tin đối tượng tạo ra ở bước 2 là một tập tin ngôn ngữ máy nhưng nó không chạy được vì chưa có dạng thích hợp của 1 file chạy . Hơn nữa nó chưa biết chương trình được nạp vào vị trí nào trên bộ nhớ để chạy . Một số địa chỉ dưới dạng mã máy có thể bị thiếu .

Trình LINK sẽ liên kết một hoặc nhiều file đói tượng thành một file chạy duy nhất (*.EXE) .Tập tin này có thể được nạp vào bộ nhớ và thi hành .

Để liên kết chương trình ta gõ :

LINK PGM1;

Nếu không có dấu chấm phẩy ASM sẽ yêu câù chúng ta gõ vào tên tập tin thực thi .

Bước 4 : Chạy chương trình

Từ dấu nhắc lệnh có thể chạy chương trình bằng cách gõ tên nó rồi nhấn ENTER.

1.11 Xuất một chuỗi ký tự

Trong chương trình PGM1 trên đây chúng ta đã dùng INT 21H hàm 2 và 4 để đọc và xuất một ký tự . Hàm 9 ngắt 21H có thể 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 điều khiển thì chức năng điều khiển tương ứng 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 destnation , source

Ngắt 21h , hàm số 9 sẽ xuất một chuỗi ký tự ra màn hình với điều kiện địa chỉ hiệu dụng của biến chuỗi phải ở trên DX . Có thể thực hiện điều 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 chuá 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 phân đoạn của nó (PSP) trong cả DS và ES trước khi thực thi chương trình . Kết qủa là thanh ghi DS không chứa số đoạn 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 đây :

MOV AX,@DATA

MOV DS,AX

Ở đây @DATA là tên của đoạn số liệu được định nghĩa bởi DATA . Assembler sẽ chuyển @DATA thành số đoạn .

Sau đây là chương trình hoàn chỉnh để 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 đổi chữ thường sang chữ hoa

Chúng ta sẽ viết 1 chương trình yêu cầu người dùng gõ vào một ký tự bằng chữ thường . Chương trình sẽ đổi nó sang dạng chữ hoa rồi in ra ở 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:$'
MSG2 DB 0DH,0AH,'IN UPPER CASE IT IS :'
CHAR DB ?,'$' ; định nghĩa biến CHAR có giá trị ban đầu chưa
;xác định
.CODE
MAIN PROC
; INITIALIZE DS
        MOV AX,@DATA
        MOV DS,AX
;PRINT PROMPT USER
        LEA DX,MSG1 ; lấy thông điệp số 1
        MOV AH,9
        INT 21H ; xuất nó ra màn hình
;nhập vào một ký tự thường và đổi nó thành ký tự hoa
        MOV AH,1 ; nhập vào 1 ký tự
        INT 21H ; cất nó trong AL
        SUB AL,20H ; đổi thành chữ hoa và cất nó trong AL
        MOV CHAR, AL ; cất ký tự trong biến CHAR
; xuất ký tự trên dòng tiếp theo
        LEA DX, MSG2 ; lấy thông điệp thứ 2
        MOV AH,9
        INT 21H ; xuất chuỗi ký tự thứ hai , vì MSG2 không kết
;thúc bởi ký tự $ nên nó tiếp tục xuất ký tự có trong biến CHAR
;dos exit
        MOV AH,4CH
        INT 21H ; dos exit
MAIN ENDP
        END MAIN
```

Chương 2 : TRẠNG THÁI CỦA VXL & CÁC THANH GHI CỜ

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

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

2.1 Các thanh ghi cờ (Flags register)

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

Bảng 2.1 cho thấy thanh ghi cờ 16 bit của 8086

11	10	9	8	7	6	5	4	3	2	1	0
OF	DF	IF	TF	SF	ZF		AF		PF		CF

Bảng 2.1 :Thanh ghi cờ của 8086

Mục đích của các thanh ghi cờ là chỉ ra trạng thái của CPU .Có hai loại cờ là cờ trạng thái (status flags) và cờ điều khiển (control flags) . Cờ trạng thái phản ánh các kết qủa thực hiện lệnh của CPU . Bảng 2.2 chỉ ra tên và ký hiệu các thanh ghi cờ trong 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	Interrrupt flag	IF
10	Direction flag	DF

Bảng 2.2 : Các cờ của 8086

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

Các cờ 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 qủa của phép trừ là zero . Cờ nhớ (Carry Flag - CF) : CF=1 nếu xuất hiện bit nhớ (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 chẳn lẻ (Parity Flag - PF) : PF=1 nếu byte thấp của kết qủa có tổng số con số 1 là một số chẳn (even parity). PF=0 nếu byte thấp là chẳn lẻ lẻ (old parity). Ví dụ nếu kết qủa là FFFEh thì PF=0

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

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

Cờ dấu (Sign Flag - SF) : SF=1 nếu MSB của kết qủa là 1 (kết qủa 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 đây chúng ta sẽ phân tích các trường hợp xảy ra tràn trong khi thực hiện tính toán . Hiện tượng 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 . Đối 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 qủa của một phép toán vượt ra ngoài dãi số có thể biểu diễn thì xảy ra sự tràn số . Khi có sự tràn số kết qủa thu được sẽ bị 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 học chẳng hạn phép cộng , sẽ xảy ra 4 khả năng sau đây :

1) không tràn

2) chỉ tràn dấu

3) chỉ tràn không dấu

4) tràn cả dấu và không dấu

Ví dụ của tràn không dấu là phép cộng ADD AX,BX với AX=0FFFFh , BX=0001h .Kết qủa dưới dạng nhị phân là :

1111 1111 1111 1111
0000 0000 0000 0001
10000 0000 0000 0000

Nếu diễn giải kết qủa dưới dạng không dấu thì kết qủa là đúng (10000h=65536) . Nhưng kết qủa đã vượt quá độ lớn của từ nhớ . Bit 1 (bit nhớ từ vị trí MSB) đã xảy ra và kết qủa trên AX =0000h là sai . Sự tràn như thế là tràn không dấu . Nếu xem rằng phép cộng trên đây là phép cộng hai số có dấu thì kết qủa trên AX = 0000h là đúng , vì FFFFh = -1 , còn 0001h = +1 , do đó kết qủa phép cộng là 0 . Vậy trong trường hợp này sự tràn dấu không xảy ra .

Ví dụ về sự tràn dấu : giả sử AX = BX = 7FFFh ,

lệnh ADD AX,BX sẽ cho kết qủa như 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 . Như vậy là đối với phép cộng có dấu cũng như không dấu thì kết qủa vẫn là 32767 + 32767 = 65534 . Số này(65534) đã vượt ngoài dãi giá trị mà 1 số 16 bit có dấu có thể biểu diễn . Hơn nửa FFFEh = -2 . Do vậy sự tràn dấu đã xảy ra .

Trong trường hợp xảy ra tràn , CPU sẽ biểu thị sự tràn như 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 được thực hiện để sửa sai kết qủa ngay lập tức . Các lập trình viên sẽ chỉ phải quan tâm tới cờ OF hoặc CF nếu biểu diễn số của họ là có dấu hay không dấu một cách tương ứng .

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

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

a) Khi cộng hai số cùng dấu , sự 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 qủa là FFFFh (số âm)

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

Vậy làm thế nào để CPU chỉ 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 nhớ (carry) hoặc mượn (borrow) từ MSB
- Tràn dấu xảy ra khi cộng hai số cùng dấu (hoặc trừ 2 số khác dấu) mà kết qủa với dấu khác với dấu mong đợi . Phép cộng hai số có dấu khác nhau không thể xảy ra sự tràn . Trên thực tế CPU dùng phương pháp sau : cờ OF=1 nếu số nhớ vào và số nhớ ra từ MSB là không phù hợp : nghĩa là có nhớ vào nhưng không có nhớ ra hoặc có nhớ ra nhưng không có nhớ vào .

Cờ điều khiển (control flags)

Có 3 cở điều khiển trong CPU , đó là :

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

Các cờ điều khiển được dùng để điều khiển hoạt động của CPU

Cờ hướng (DF) được dùng trong các lệnh xử lý chuỗi của CPU . Mục đích của DF là dùng để điều khiển hướng mà một chuỗi được xử lý . Trong các lệnh xử lý chuỗi hai thanh ghi DI và SI được dùng để địa chỉ bộ nhớ chứa chuỗi . Nếu DF=0 thì lệnh xử lý chuỗi sẽ tăng địa chỉ bộ nhớ sao cho chuỗi được xử lý từ trái sang phải Nếu DF=1 thì địa chỉ bộ nhớ sẽ được xử lý theo hướng từ phải sang trái .

2.3 Các lệnh ảnh hưởng đế cờ như thế nào

Tại một thời điểm , CPU thực hiện 1 lệnh , các cờ lần lượt phản ánh kết qủa thực hiện lệnh . Dĩ nhiên 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 chỉ xét ảnh hưởng của các lệnh (đã nghiên cứu ở chương trước) lên các cờ như thế nào .

Bảng sau đây cho thấy ảnh hưởng của các lệnh đến các cờ :

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

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

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

FFFFh
+ FFFFh
1FFFEh

Kết qủa chứa 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 qủa

ZF=0 vì kết qủa khác 0

CF=1 vì có nhớ 1 từ MSB

OF=0 vì dấu của kết qủa giống như dấu của 2 số hạng ban đầu .

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

80h
+ 80h
100h

Kết qủa trên AL = 00h

SF=0 vì MSB=0

PF=1 vì tất cả các bit đều bằng 0

ZF=1 vì kết qủa bằng 0

CF=1 vì có nhớ 1 từ MSB

OF=1 vì cả 2 toán hạng là số âm nhưng kết qủa là

số dương (có nhớ ra từ MSB nhưng không có nhớ vào) .

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

8000h
- 0001h

7FFFFh = 0111 1111 1111 1111

SF=0 vì MSB=0

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

ZF=0 vì kết qủa khác 0

CF=0 vì không có mượn

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

Ví dụ 4 : INC AL trong đó AL=FFh

Kết qủa trên AL=00h = 0000 0000

SF=0 vì MSB=0

PF=1

ZF=1 vì kết qủa bằng 0

CF không bị ảnh hưởng bởi lệnh INC mặc dù có nhớ 1 từ MSB

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

Ví dụ 5: MOV AX,-5

Kết quả trên BX = -5 = FFFBh

Không có cờ nào ảnh hưởng bởi lệnh MOV

Ví dụ 6: NEG AX trong đó AX=8000h

8000h =1000 0000 0000 0000

bù 1 =0111 1111 1111 1111

+1

1000 0000 0000 0000 = 8000h

Kết qủa trên AX=8000h

SF=1 vì MSB=1

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

ZF=0 vì kết qủa khác 0

CF=1 vì lệnh NEG làm cho CF=1 trừ khi kết qủa bằng 0

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

2.4 Chương trình DEBUG.EXE

Debug là một chương trình của DOS cho phép chạy thử các chương trình hợp ngữ . Người dùng có thể cho chạy chương trình từng lệnh 1 từ đầu đến cuối ,trong quá trình đó có thể thấy nội dung các thanh ghi thay đổi như thế nào . Debug cho phép nhập vào một mã hợp ngữ trực tiếp sau đó 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 đây chúng ta sẽ dùng DEBUG để mô tả cách thức mà các lệnh ảnh hưởng đến các cờ như thế nào .

Giả sử chúng ta có chương trình hợp ngữ 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 ; HÀM THOÁT VỀ DOS
        INT 21H ; EXIT TO DOS
END
MAIN ENDP
        END MAIN
```

Sau khi dịch chương trình , giả sử file chạy là CHECKFL.EXE trên đường dẫn C:\ASM . Để chạy debug chúng ta gõ lệnh sau :

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

Từ lúc này trở đi dấu nhắc làcủa debug (dấu "_") , người sử dụng có thể đưa vào các lệnh debug từ dấu nhắc này . Trước hết có thể xem nội dung các thanh ghi bằng lệnh

R(Register) , màn hình sẽ có nội dung như 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 và nội dung của chúng (dưới dạng HEX) trên 3 dòng đầu .

Dòng thứ 4 là trạng thái các thanh ghi theo cách biểu thị của debug.

Bảng 2-3 là cách mà Debug biểu thị trạng thái của các thanh ghi cờ của CPU .

Flags	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 interrupts)	DI (disable interrupts)

Bảng 2.3 : Biểu thị trạng trạng các cờ của DEBUG

Dòng cuối cùng cho biết giá trị hiện hành của PC (địa chỉ của lệnh sẽ được thực hiện dưới dạng địa chỉ logic) mã máy của lệnh và nội dung của lệnh tương ứng . Khi chạy chương trình này trên 1 máy tính khác có thể sẽ thấy một địa chỉ đoạn 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 qủa 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
```

Cờ OF=0(NV) nhưng CF=1(CY) vì có mượn từ MSB
Cờ PF=0(PO) vì byte thấp chỉ có 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 qủa 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 cộng 2 số dương mà kết quả là 1 số âm
CF=1(CY) vì lệnh INC không ảnh hưởng tới cờ này .
Để thực hiện toàn bộ chương trình chúng ta gõ G(Go)

```
-G
```

Program terminated normally

Để thoát khoỉ debug gõ Q(Quit)

-Q

C:\>

Bảng sau đây cho biết một số lệnh debug thường dùng , các tham số để trong ngoặc là tuỳ 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 đã
	được hiển thi
G(=start) (addr1 addr2...addrn)	Chạy (go) lệnh từ vị trí Start với các điểm dừng tại addr1,addr2,addrn
G	Thực thi lệnh từ CS:IP đến hết
G=100	Thực thi lệnh từ CS:100h đến hết
G=100 150	Thực thi lệnh tại CS:100h dừng tại CS:150h
Q	Quit debug and return to DOS
R(register)	Xem/ thay đổi nội dung của thanh ghi
R	Xem nội dung tất cả các thnah ghi và cờ
R AX	Xem và thay đổi nội dung của thanh ghi AX
T(=start)(value)	Quét "value" lệnh từ vị trí start
T	Trace lệnh tại CS:IP
T=100	Trace lệnh tại CS:100h
T=100 5	Trace 5 lệnh bắt đầu từ CS:100h
T 4	Trace 4 lệnh bắt đầu từ CS:IP
U(start)(value)	Unassemble vùng địa chỉ thành lệnh asm
U CS:100 110	Unassemble từ CS:100h đến CS:110h
U 200 L 20	Unassemble 20 lệnh từ CS:200h
U	Unassemble 32 bytes từ bytes cuối cùng được hiển thi
A(start)	Đưa vào mã hợp ngữ cho 1 địa chỉ hoặc 1
	vùng địa chỉ
A	
A CS:100h	Đưa vào mã hợp ngữ tại CS:IP
	Đưa vào mã hợp ngữ tại CS:100h

Chương 3 : CÁC LỆNH ĐIỀU KHIỂN

Một chương trình thông thường sẽ thực hiện lần lượt các lệnh theo thứ thự mà 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

Để hình dung được lệnh nhảy làm việc như 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 đến 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 đoạn chương trình xuất ký tự có nhãn địa chỉ bộ nhớ là PRINT_LOOP

3.2 Nhảy có điều kiện

Lệnh JNZ là một lệnh nhảy có điều kiện .Cú 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. Đối với lệnh JNZ thì điều kiện là kết qủa của lệnh trước nó phải bằng 0 .

Phạm vi của lệnh nhảy có điều kiện .

Cấu trúc mã máy của lệnh nhảy có điều kiện yêu cầu destination-label đến (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 điều chỉnh IP đến 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 khám xét các cờ ZF . Nếu ZF=0 điều khiển được 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 đối với kết quả)
- Nhảy không dấu (dùng cho các diễn dịch không dấu đối với kết quả)
- Nhảy một cờ (dùng cho các thao tác chỉ ảnh hưởng lên 1 cờ)

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

Nhảy có dấu

SYMBOL	DESCRITION	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 jupm if not less or equal to	SF=OF
JL/JNGE	jump if lees 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	DESCRITION	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 cờ

SYMBOL	DESCRITION	CONDITION FOR JUMPS
JE/JZ	jump if equal	ZF=1
	jump if equal to zero	
JNE/JNZ	jump if not equal	ZF=0
	jump if not zero	
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
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 thường lấy kết qủa của lệnh Compare như là 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 Destinaition - Source . Kết qủa sẽ không được cất giữ . Như vậy là lệnh CMP giống như lệnh SUB , chỉ khác là trong lệnh CMP toán hạng đích không thay đổi .

Giả sử chương trình chứa các lệnh sau :

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

Kết qủa của lệnh CMP AX,BX là 7FFEh . Lệnh JG được thỏa mãn vì ZF=0=SF=OF do đó điều khiển được chuyển đến nhãn BELOW.

Diễn dịch lệnh nhảy có điều kiện

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

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

có nghĩa là nếu AX>BX thì nhảy đến nhãn BELOW

Mặc dù lệnh CMP được thiết kế cho các lệnh nhảy . Nhưng lệnh nhảy có thể đứng trước 1 lệnh khác , chẳng hạn :

```
DEC AX
JL THERE
```

có nghĩa là nếu AX trong diễn dịch có dấu < 0 thì điều khiển được chuyển cho THERE .

Nhảy có dấu so với nhảy không dấu

Một lệnh nhảy có dấu tương ứng với 1 nhảy không dấu . Ví dụ lệnh nhảy có dấu JG và lệnh nhảy không dấu JA . Việc sử dụng JG hay JA là tuỳ thuộc vào diễn dịch 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 dụnh lệnh nhảy không hợp lý sẽ tạo ra kết quả sai .

Giả sử rằng chúng ta diễn dịch có dấu .Nếu AX=7FFFh và BX=8000h , các lệnh :

```
CMP AX,BX
JA below
```

sẽ cho kết qủa sai mặc dù 7FFFh > 8000h (lệnh JA không thực hiện được vì 7FFFFh < 8000h trong diễn dịch không dấu)

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

Ví dụ : Giả sử rằng AX và BX chưá các số có dấu . Viết đoạn ct để đặt 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 khắc phục hạn chế 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:
        ; thân vòng lặp
        DEC CX
        JNZ TOP ; nếu CX>0 tiếp tục lặp
        MOV AX,BX
```

giả sử thân vòng lặp chứa nhiều lệnh mà nó vượt khỏi 126 bytes trước lệnh JNZ TOP . Có thể giải quyết tình trạng này bằng các lệnh sau :

```
TOP:
        ; thân 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 nhảy để thực hiện các cấu trúc tương tự như 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 đến những đoạn khác nhau tuỳ 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 tuyết đối của nó

Thuật toán như sau :

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

Có thể mã hoá 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 hoá 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 thanh ghi hay, biến nào đo hay một biểu thức mà giá trị cụ thể nằn 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 hoá 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 :
```

Rẻ nhánh với một tổ hợp các điều kiện

Đôi khi tình trạng rẻ 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 : Đọc 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 ( charater <= ‘Z’)
THEN
display character
END_IF
```

Sau đây là code

```
;read a character
        MOV AH,1
        INT 21H                 ; character in AL
; IF ( ‘A’<= character ) AND ( charater <= ‘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 : Đọc một ký tự , nếu ký tự đó là ‘Y’ hoặc ‘y’ thì in nó lên màn hình , ngược lại thì kết thúc chương trình .

Thuật toán

```
Read a charcter ( into AL)
IF ( character =‘Y’) OR ( character=‘y’)
THEN
        dispplay it
ELSE
        terminate the program
END_IF
```

Code

```
;read a character
        MOV AH,1
        INT 21H             ; character in AL
; IF ( character =‘y’ ) OR ( charater = ‘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 được lặp lại , số lần lặp phụ thuộc điều kiện .

a) Vòng FOR

Lệnh LOOP có thể dùng để thực hiện vòng FOR .Cú 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 chưa 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 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 chưá số lần lặp
        MOV AH,2        ; hàm xuất ký tự
        MOV DL,'*'      ;DL chưá ký tự '*'
TOP:
        INT 21h         ; in dấu '*'
        LOOP TOP        ; lặp 80 lần
```

Lưu ý rằng vòng FOR cũng như 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 . Để tránh 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 điều khiển được chuyển cho destination_label . Các lệnh sau đây sẽ đảm bảo 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 đếm số ký tự được nhập vào trên cùng một hàng .

```
        MOV DX,0            ; DX để đếm 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 thi hành đồng thời điều kiện được kiểm tra. Nếu điều kiện đúng thì vòng lặp kết thúc .

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

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

Lưu ý : việc sử dụng REPEAT hay WHILE là tuỳ theo chủ quan của mỗi người . Tuy nhiên có thể thấy rằng REPEAT phải tiến hành ít nhất lần , trong khi đó WHILE có thể không tiến hành lần nào cả nếu ngay từ đầu điều kiện đã bị 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 thường , máy sẽ thông báo 'No capitals'

Kết qủa chạy chương trình sẽ như sau :

```
Type a line of text :
TRUONG DAi HOC DALAT
First capital = A
Last capital = U
```

Để 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 như sau :

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

Bước 1: Hiện dấu nhắc .

Bước này có thể mã hoá như sau :

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

Dấu nhắc có thể mã hoá như sau trong đoạn số liệu .

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

Bước 2 : Đọc và xử lý một dòng văn bản

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 carrige 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 đó dòng (*) có nghĩa là điều kiện để ký tự là hoa là điều kiện AND

```
IF ( ‘A’<= character ) AND ( character <= ‘Z’)
```

Bước 2 có thể mã hoá 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 đến
END_IF
        CMP AL,’Z’          ; char <= ‘Z’?
        JNLE END_IF         ; không phải ký tự hoa thì nhảy đến
END_IF
; thì
; nếu ký tự nằm trước biến FIRST ( giá trị ban đầu là‘[‘ : ký tự sau Z )
        CMP AL,FISRT ; 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 :
; đọc 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 số liệu :

```
FIRST DB '[ $' ; '[' là ký tự sau Z
LAST DB '@ $ ' ; '@' là ký tự trước A
```

Bước 3 : In kết qủa

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 được định nghĩa như sau trong đoạn số liệu .

```
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 hoá như sau :

```
;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 qủa
;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 và xử lý 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 đến END_IF
        CMP AL,'Z'           ; char <= 'Z'?
        JNLE END_IF ; không phải ký tự hoa thì nhảy đến END_IF
; thì
; nếu ký tự nằm trước biến FIRST
        CMP AL,FISRT        ; 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
        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 qủa
;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, DỊCH 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 đến từng bit thường là không có trong các ngôn ngữ cấp cao (trừ C) và đây là lý do giải thích tại sao hợp ngữ vẫn đóng 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 được áp dụng cho toán hạng 8 hoặc 16 bit thì có thể áp dụng phép toán logic đó trên từng bit để thu được kết qủa 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 :

1. 1010 1010
   AND 1111 0000
   = 1010 0000

2. 1010 1010
   OR 1111 0000
   = 1111 1010

3. 1010 1010
   XOR 1111 0000
   = 0101 1010

4. NOT 10101010
   = 01010101

4.1.1 Lệnh AND,OR và XOR

Lệnh AND,OR và XOR thực hiện các chức năng đúng như tên gọi của nó .

Cú pháp của chúng là :

AND destination , source

OR destination , source

XOR destination , source

Kết qủa của lệnh được lưu trữ trong toán hạng đích do đó chúng phải là thanh ghi hoặc vị trí nhớ . Toán hạng nguồn là có thể là hằng số , thanh ghi hoặc vị trí nhớ. Dĩ nhiên hai toán hạng đều là vị trí nhớ là không được phép .

Ảnh hưởng đến các cờ :

Các cờ SF,ZF và PF phản ánh kết qủa

AF không xác định

CF=OF=0

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

b AND 1 = b	b OR 0 = b	b XOR 0 = b
b AND 0 = 0	b OR 1 = 1	b XOR 1 = not b

- Lệnh AND có thể dùng để xóa (clear) toán hạng đích nếu mặt nạ bằng 0
- Lệnh OR có thể dùng để đặt (set) 1 cho toán hạng đích nếu mặt nạ bằng 1
- Lệnh XOR có thể dùng để lấy đảo toán hạng 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 : Xoá bit dấu của AL trong khi các bit khác không thay đổi

Giải : Dùng lệnh AND với mặt nạ 0111111=7Fh

```
AND AL,7Fh          ; xóa bit dấu ( dấu + ) 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 dấu của DX

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

```
XOR DX,8000h
```

Các lệnh logic là đặc biệt có ích khi thực hiện các nhiệm vụ sau :

Đổi một số dưới dạng ASCII thành một số

Giả sử rằng chúng ta đọc một ký tự từ bàn phím bằng hàm 1 ngắt 21h . Khi đó AL chứa mã ASCII của ký tự . Điều này cũng đúng nếu ký tự đó là một số (digital character) . Ví dụ nếu chúng ta gỏ số 5 thì AL = 35h (ASCII code for 5)

Để chứa 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 đổi mọi số ASCII ra thập phân .

Chương trình hợp ngữ đổi một số thập phân thành mã ASCII của chúng được xem như 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 . Giả sử DL chuá ký tự thường , để đổi nó thành chữ hoa ta dùng lệnh :

```
SUB DL,20h
```

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

Character	**Code**	**Character**	**Code**
a(61h)	01100001	A (41h)	01000001
b (62h)	01100010 B	(42h) 0	1000010
.			
.			
z (7Ah)	01111010	Z (5Ah)	01011010

Có thể xóa bit thứ 5 của DL bằng cách dùng lệnh AND với mặt nạ 11011111= DF h

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

Xóa một thanh ghi

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

```
MOV AX,0
```

hoặc

```
SUB AX,AX
XOR AX,AX
```

Lệnh thứ nhất cần 3 bytes trong khi đó 2 lệnh sau chỉ cần 2 bytes . Nhưng lệnh MOV phải được dùng để xoá 1 vị trí nhớ .

Thay cho lệnh

```
CMP AX,0
```

Người 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) toán hạng đích . 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 toán hạng đích và toán hạng nguồn nhưng không làm thay đổi toán hạng đích . 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 qủa

AF không xác định

CF=OF=0

Lệnh TEST có thể dùng để khám 1 bit trên toán hạng . Mặt nạ phải chứa bit 1 tại vị trí cần khám , các bit khác thì bằng 0 . Kết quả của lệnh :

TEST destination,mask

sẽ là 1 tại bit cần test nếu như toán hạng đích chứa 1 tại bit test . Nếu toán hạng đích 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 ; đúng , nhảy đến BELOW
```

4.2 Lệnh SHIFT

Lệnh dịch và quay sẽ dịch các bit trên trên toán hạng đích một hoặc nhiều vị trí sang trái hoặc sang phải . Khác nhau của lệnh dịch và lệnh quay là ở chỗ : các bit bị dịch ra (trong lệnh dịch) sẽ bị mất .Trong khi đó đối với lệnh quay , các bit bị dịch ra từ một đầu của toán hạng sẽ được đưa trở lại đầu kia của nó . Có 2 khả năng viết đối với lệnh dịch và quay :

OPCODE destination,1

OPCODE destination,CL

trong cách viết thứ hai thanh ghi CL chuá N là số lần dịch hay quay . Toán hạng đích có thể là một thanh ghi 8 hoặc 16 bit , hoặc một vị trí nhớ .

Các lệnh dịch và quay thường dùng để nhân và chia các sô nhị phân. Chúng cũng được dùng cho các hoạt động nhập xuất nhị phân và hex .

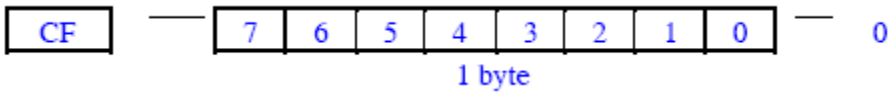
4.2.1 Lệnh dịch trái (left shift)

Lệnh SHL dịch toán hạng đích 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)

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



Các cờ bị ảnh hưởng :

SF,PF,ZF phản ánh kết qủa

AF không xác định

CF= bit cuối cùng được dịch ra

OF= 1 nếu kết qủa 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 qủa DH=01010000=50h , CF=0

Nhân bằng lệnh SHL

Chúng ta hãy xét số 235decimal . Nếu dịch trái 235 một bit và thêm 0 vào bên phải chúng ta sẽ có 2350 . Noí cách khác , khi dịch trái 1 bit chúng ta đã nhân 10. Đối với số nhị phân, dịch trái 1 bit có nghĩa là nhân nó 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 nhấn mạnh đến tính chất số học của phép toán thì lệnh SAL sẽ được dùng thay cho SHL . Cả 2 lệnh đều tạo ra cùng một mã máy .

Một số âm cũng có thể được nhân 2 bằng cách dịch trái . Ví dụ : Nếu AX=FFFFh= -1 thì sau khi dịch trái 3 lần AX=FFF8h = -8

Tràn

Khi chúng ta dùng lệnh dịch trái để nhân thì có thể xảy ra sự tràn . Đối với lệnh dịch trái 1 lần , CF và OF phản ánh chính xác sự tràn dấu và tràn không dấu . Tuy nhiên các cờ sẽ không phản ánh chính xác kết qủa 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=80h , CL=2 thì lệnh

SHL BL,CL

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

Ví dụ : viết đoạn mã nhân AX với 8 . Giả sử rằng không có tràn .

```
MOV CL,3 ; CL=3
SHL AX,CL ; AX*8
```

4.2.2 Lệnh dịch phải (Right Shift)

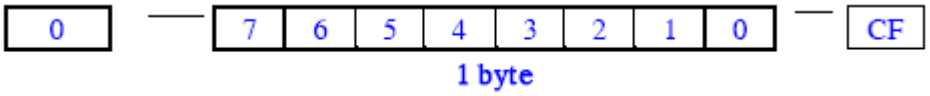
Lệnh SHR dịch phải toán hạng đích 1 hoặc N lần .

SHR destination,1

SHR destination,CL

Cứ mỗi lần dịch phải , một số 0 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 qủa trên DH=22h , CF=1

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

Chia bằng lệnh dịch phải

Lệnh dịch phải sẽ chia 2 giá trị của toán hạng đích . Điều này đúng đối với số chẵn . Đối với số lẻ , lệnh dịch phải sẽ chia 2 và làm tròn xuống số nguyên gần nó nhất . Ví dụ, nếu BL = 00000101=5 thì khi dịch phải BL=00000010 =2 .

Chia có dấu và không dấu

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

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

```
MOV AX,65134
MOV CL,2
SHR AX,CL
```

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

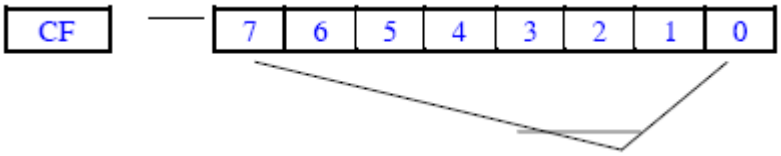
SAR AL,1 được thực hiện

Giải : AL= -15 = 11110001b

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

4.3 Lệnh quay (Rotate)

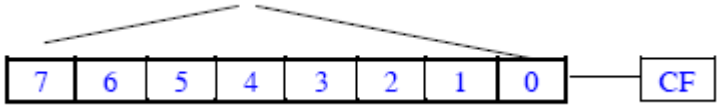
Quay trái (rotate left) = ROL sẽ quay các bit sang trái , LSB sẽ được thay bằng MSB . Còn CF=MSB



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 đây cho thấy cách để khám các bit trên một byte hoặc 1 từ mà không làm thay đổi nội dung của nó .

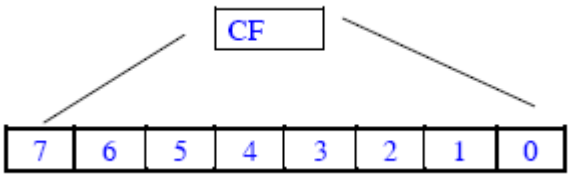
Ví dụ : Dùng ROL để đếm số bit 1 trên BX mà không thay đổi nội dung của nó. Kết qủa cất trên AX .

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

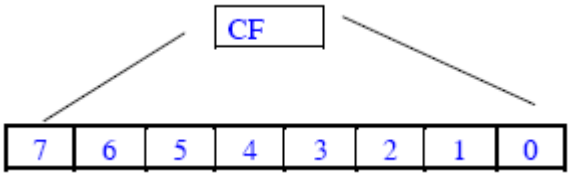


Cú pháp của của lệnh RCL như sau :

RCL destination,1

RCL destination,CL

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



Cú pháp của 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

Anh hưởng của lệnh quay lên các cờ

SF,PF và ZF phản ảnh kết quả

CF-bit cuối cùng được dịch ra

OF=1 nếu kết qủa 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 .

Đoạn mã để 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 thường được sử dụng trong các hoạt động xuất nhập số nhị phân và số hex.

4.4.1 Nhập số nhị phân

Giả sử 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 luỹ chúng trong 1 thanh ghi . Thuật toán sau đây sẽ đọc một số nhị phân từ bàn phím và cất nó 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
```

Đoạn mã thực hiện thuật toán trên như sau :

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

```
END_WHILE:
```

4.4.2 Xuất số nhị phân

Giả sử 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
```

Đoạn mã để xuất số nhị phân có thể xem như bài tập .

4.4.3 Nhập số HEX

Nhập số hex bao gồm các số từ 0 đến 9 và các ký tự A đến F . Kết qủa chứa trong BX. Để cho đơn giản chúng ta giả sử rằng :

- ✓ chỉ có ký tự hoa được dùng
- ✓ người dùng nhập vào không qúa 4 ký tự hex

Thuật toán như 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
```

Đoạn mã có thể viết như 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

Để xuất số hex trên BX (16 bit = 4 digit hex) có thể 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

Đoạn ngăn xếp (stack segment) trong chương trình được dùng để cất giử tạm thời số liệu và địa chỉ . Trong chương này chúng ta sẽ xem xét cách tổ chức stack và sử dụng nó để thực hiện các thủ tục (procedure) .

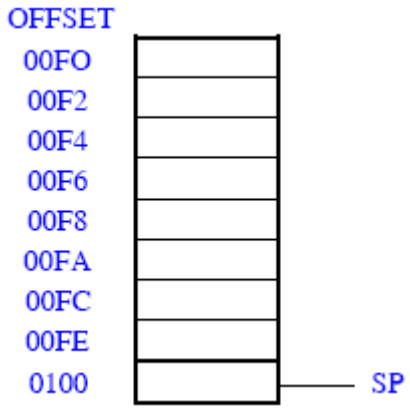
5.1 Ngăn xếp

Ngăn xếp là cấu trúc dữ liệu 1 chiều . Điều đó có nghĩa là số 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 đó số liệu được đưa vào hay lấy ra gọi là đỉnh của ngăn xếp (top of stack) .Có thể hình dung satck như 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 chỉ dẫn

.STACK 100h

để khai báo kích thước vùng stack là 256 bytes .

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



AX=1234 BX=5678 SP=0100

Hình 4.1 : STACK EMPTY

Lệnh PUSH và PUSHF

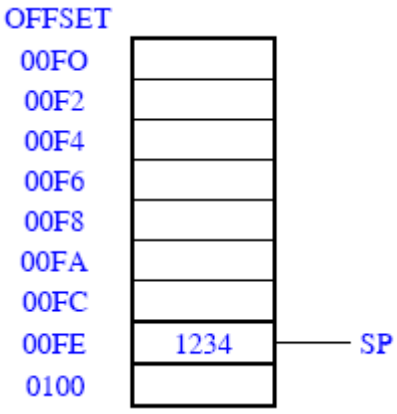
Để thêm một từ mới vào stack chúng ta dùng lệnh :

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

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

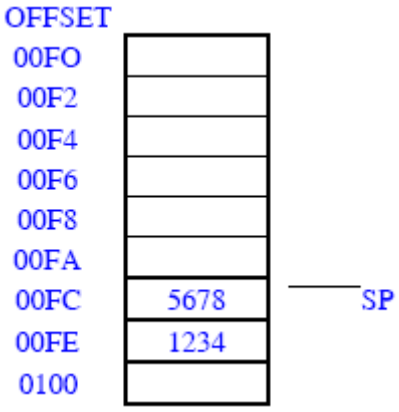
- SP giảm đi 2
- một bản copy của toán hạng nguồn được chuyển đến địa chỉ SS:SP còn toán hạng nguồn không thay đổi .

Lệnh PUSHF không có toán hạng .Nó dùng để đẩy nội dung thanh ghi cờ 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 .



AX=1234 BX=5678 SP=00FE

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



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

Lệnh POP và POPF

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

POP destination ; lấy số liệu tại đỉnh stack ra destination

Destination có thể là 1 thanh ghi hoặc từ nhớ 16 bit . Ví dụ :

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

Khi thực hiện lệnh POP :

- nội dung của đỉnh stack (địa chỉ SS:SP) được di chuyển đến đích .
- SP tăng 2

Lệnh POPF sẽ lấy đỉnh stack đưa vào thanh ghi cờ .

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 như :

PUSH DL ; lệnh không hợp lệ

PUSH 2 ; lệnh không hợp lệ

Ngoài chức năng lưu trữ số liệu và địa chỉ của chương trình do người sử dụng viết, tack còn được dùng bởi hệ điều hành để lưu trữ trạng thái của chương trình chính khi có ngắt.

5.2 Ứng dụng của stack

Bởi vì nguyên tắc làm việc của stack là LIFO nên các đối tượng được lấy ra khỏi stack có trật tự ngược lại với trật tự mà chúng được đưa vào stack. Chương trình sau đây sẽ đọc một chuỗi ký tự rồi in chúng trên dòng mới với trật tự ngược lại .

Thuật toán cho chương trình như sau :

```
Display a '? '
Initialize count to 0
Read a character
        WHILE character is not CR DO
                PUSH chracter onto stack
                Incremet 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 đếm 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 đếm
        PUSH AX ; đẩy 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 được 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 giữ trên AL nhưng phải đẩy 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 Thủ tục (Procedure)

Trong chương 3 chúng ta đã đề cập đến ý tưởng lập trình top-down. Ý tưởng này có nghĩa là một bài toán nguyên thuỷ được chia thành các bài toán con mà chúng dễ giải quyết hơn bài toán nguyên thuỷ . Trong các ngôn ngữ cấp cao người ta dùng thủ tục để giải các bài toán con , và chúng ta cũng làm như vậy trong hợp ngữ. Như vậy là một chương trình hợp ngữ có thể được xây dựng bằng các thủ tục .

Một thủ tục gọi là thủ tục chính sẽ chứa nội dung chủ yếu của chương trình. Để thực hiện một công việc nào đó , thủ tục chính gọi (CALL) một thủ tục con. Thủ tục con cũng có thể gọi một thủ tục con khác .

Khi một thủ tục gọi một thủ tục khác , điều khiển được chuyển tới (control transfer) thủ tục được gọi và các lệnh của thủ tục được gọi sẽ được thi hành . Sau khi thi hành hết các lệnh trong nó , thủ tục được gọi sẽ trả điều khiển (return control) cho thủ tục gọi nó . Trong ngôn ngữ cấp cao , lập trình viên không biết và không thể biết cơ cấu của việc chuyển và trả điều khiển giữa thủ tục chính và thủ tục con.

Nhưng trong hợp ngữ có thể thấy rỏ cơ cấu này (xem phần 5.4) .

Khai báo thủ tục

Cú pháp của lệnh tạo một thủ tục như sau :

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

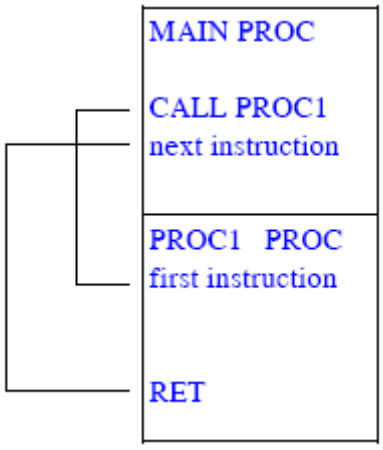
Name do người dùng định nghĩa là tên của thủ tục .

Type có thể là NEAR (có thể không khai báo) hoặc FAR . NEAR có nghĩa là thủ tục được gọi nằm cùng một đoạn với thủ tục gọi . FAR có nghĩa là thủ tục được gọi và thủ tục gọi nằm khác đoạn . Trong phần này chúng ta sẽ chỉ mô tả thủ tục NEAR .

Lệnh RET trả điều khiển cho thủ tục gọi . Tất cả các thủ tục phải kết thúc bởi RET trừ thủ tục chính .

Chú thích cho thủ tục : Để người đọc dễ hiểu thủ tục người ta thường sử dụng 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 qủa 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 được dùng 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 được thi hành thì :

- Địa chỉ quay về của thủ tục gọi được cất vào stack . Địa chỉ này chính là offset của lệnh tiếp theo sau lệnh CALL .
- IP lấy địa chỉ offset của lệnh đầu tiên trên thủ tục được gọi , có nghĩa là điều khiển được chuyển đến thủ tục .

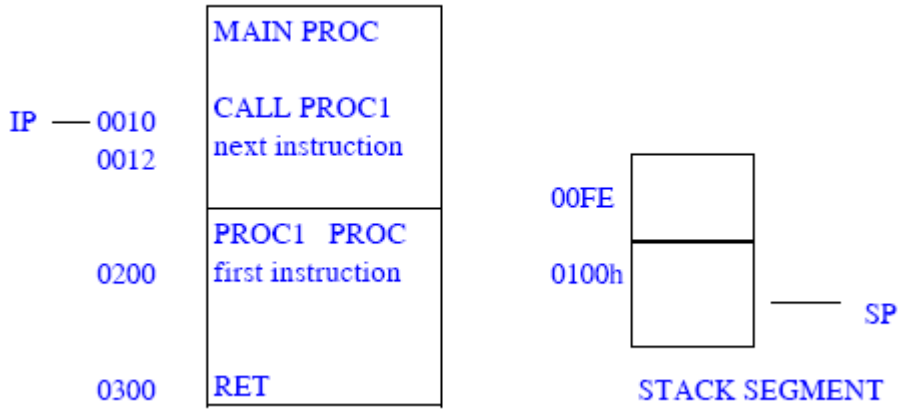
Để trả điều khiển cho thủ tục chính , lệnh

RET pop-value

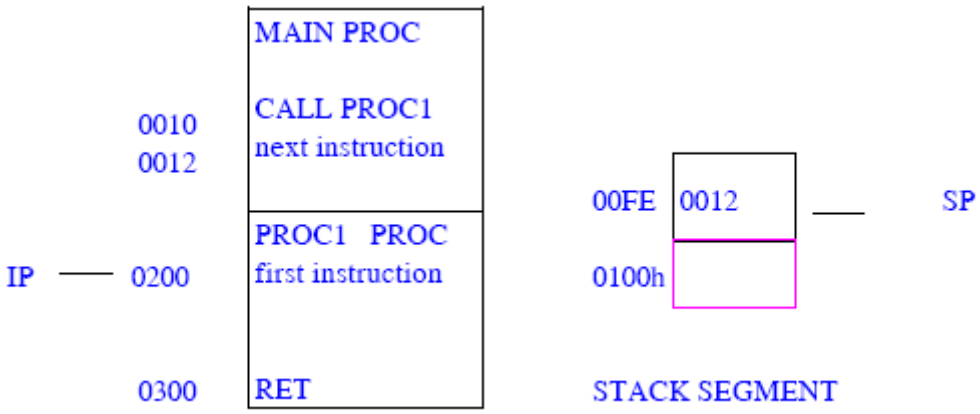
được sử dụng . Pop-value (một số nguyên N) là tùy chọn . Đối với thủ tục NEAR, lệnh RET sẽ lấy giá trị trong SP đưa vào IP . Nếu pop-value là ra một số N thì

IP=SP+N

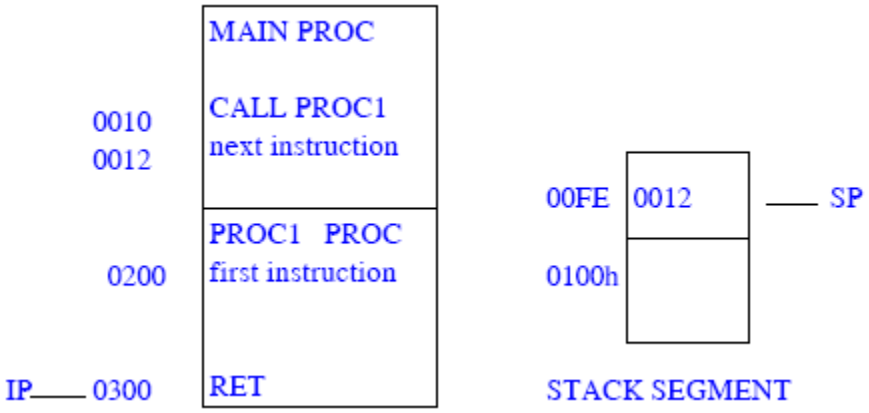
Trong cả 2 trường hợp thì CS:IP chứa địa chỉ trở về chương trình gọi và điều khiển được trả cho chương trình gọi (xem hình 5-2)



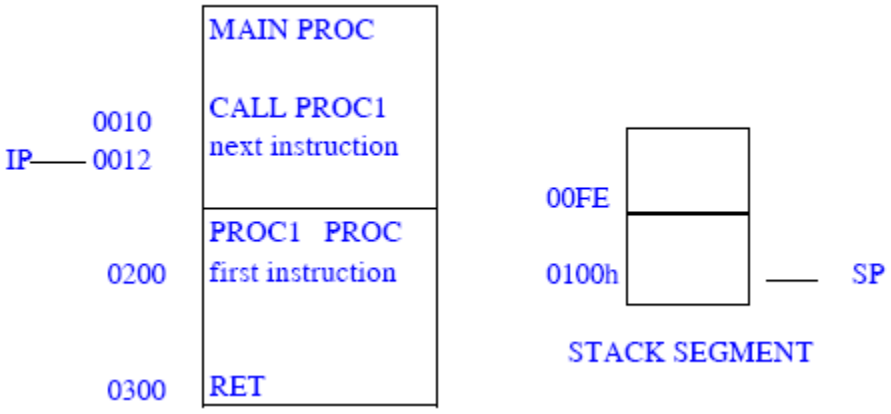
Hình 5-2 a : Trước khi CALL



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 thủ tục

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

Thuật toán như 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 đây chúng ta sẽ mã hoá thủ tục nhân với tên là MULTIPLY. Chương trình chính không có 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 qủa
        MOV AH,4CH
        INT 21H
        MAIN ENDP
        MULTIPY PROC
; input : AX=A , BX=B , AX và BX có giá trị trong khoảng 0...FFH
; output : DX= kết qủa
        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 chưa bằng 0 thì lặp
                POP BX ; lấy lại BX
                POP AX ; lấy lại AX
        RET ; trả điều khiển cho chương trình chính
MULTIPLY ENDP
        END MAIN
```

Sau khi dịch chương trình , có thể dùng DEBUG để chạy thử nó bằng cách cung cấp giá trị ban đầu cho AX và BX .

Dùng lệnh U(unassembler) để xem nội dung của bộ nhớ tương ứng với các lệnh hợp ngữ .

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 từng 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 đặc biệt đến IP để xem cách chuyển và trả điều khiển 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 thường dùng nhất của lệnh nhân và chia là thực hiện các thao tác nhập xuất thập phân . Trong chương này chúng ta sẽ viết thủ tục cho nhập xuất thập phân mà chúng được sử dụng nhiều trong các hoạt động xuất nhập từ ngoại vi

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 qủa 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 qủa là một từ 16 bit .Nếu 2 toán hạng từ được nhân với nhau thì kết qủa 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 nhưng không được là một hằng

Phép nhân kiểu byte

Đối với phép nhân mà toán hạng là kiểu byte thì

AX=AL*SOURCE ;

Phép nhân kiểu từ

Đối với phép nhân mà toán hạng là kiểu từ thì

DX:AX=AX*SOURCE

Ảnh 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 qủa(DX) bằng 0

=1 trong các trường hợp khác

sau lệnh IMUL CF/OF = 0 nếu nửa trên của kết qủa có bit dấu giống như 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	00000	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 đơn giản của lệnh MUL và IMUL

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

Ví dụ 1 : Chuyển đoạn chương trình sau trong ngôn ngữ cấp cao thành mã hợp ngữ : A = 5xA -12xB . Giả sử rằng A và B là 2 biến từ và không xảy ra sự tràn .

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 thủ tục FACTORIAL để tính N! cho một số nguyên dương . Thủ tục phải chứa N trên CX và trả về N! trên AX . Giả sử không có tràn .

Giải : Định nghiã của N! là

N! = 1 nếu N=1

= N x (N-1)x (N-2) x...x 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 như lệnh nhân , có 2 lệnh chia DIV và IDIV cho số không dấu và cho số có dấu . Cú pháp của chúng là :

DIV divisor

IDIV divisor

Toán hạng byte

Lệnh chia toán hạng byte sẽ chia số bị 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 . Thương số ở trên AL còn số dư trên AH .

Toán hạng từ

Lệnh chia toán hạng từ sẽ chia số bị 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 . Thương 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 qủa cóthể không chứa hết trên AL hoặc AX nếu số chia bé hơn rất nhiều so với số bị 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 2 : 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 Mở rộng dấu của số bị chia

Phép chia với toán hạng từ

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

- ✓ Đối với lệnh DIV , DX phải bị xoá
- ✓ Đối với lệnh IDIV , DX phải được mở rộng dấu 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               ; mở rộng dấu của AX vào DX
MOV BX,7 ; BX=7
IDIV BX           ; chia DX:AX cho BX , kết qủa trên AX , số dư
                  ; trên DX
```

Phép chia với toán hạng byte

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

- ➢ Đối với lệnh DIV , AH phải bị xoá
- ➢ Đối với lệnh IDIV , AH phải được mở rộng dấu 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               ; mở rộng dấu của AL vào AH
MOV BL,-7         ; BX= -7
IDIV BL           ; chia AX cho BL , kết qủa trên AL , số dư
                  ; trên AH
```

Không có cờ nào bị ảnh hưởng bởi lệnh CWD và CBW .

6.5 Thủ tục nhập xuất số thập phân

Mặc dù trong PC tất cả số liệu 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 thủ tục nhập xuất số thập phân .

Khi nhập số liệu , nếu chúng ta gõ 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 nhị phân tương đương của 21543 . Ngược lại khi xuất số liệu , nội dung nhị phân của thanh ghi hoặc vị trí nhớ phải được biến đổi thành một chuỗi ký tự biểu diễn một số thập phân trước khi chúng được in ra .

Xuất số thập phân (Decimal Output)

Chúng ta sẽ viết một thủ tục OUTDEC để in nội dung của thanh ghi AX như là 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 (đổi thànb số dương) rồi in số dương này sau dấu trừ (-). Như vậy là trong cả 2 trường hợp , OUTDEC sẽ in giá trị thập phân tương đương 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 .

Để hiểu chi tiết bước 6 cần phải làm việc gì , chúng ta giả sử rằng 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 10d theo thủ tục như sau :

Divide 24618 by 10 . Qoutient = 2461 , remainder = 8
Divide 2461 by 10 . Qoutient = 246 , remainder = 1
Divide 246 by 10 . Qoutient = 24 , remainder = 6
Divide 24 by 10 . Qoutient = 2 , remainder = 4
Divide 2 by 10 . Qoutient = 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 thủ tục 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 = qoutient , DX= remainder
        PUSH DX ; push remainder onto stack
        INC CX ; increment count
;until
        OR AX,AX ; qoutient = 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 giả INCLUDE

Chúng ta có thể thay đổi OUTDEC bằng cách đặt nó bên trong một chương trình ngắn và chạy chương trình trong DEBUG . Để đưa thủ tục OUTDEC vào trong chương trình mà không cần gõ nó , chúng ta dùng toán tử giả INCLUDE với cú pháp như sau :

INCLUDE filespec

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

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

INCLUDE A:\PGM6_1.ASM

Sau đây là chương trình để test thủ tục 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)

Để nhập số thập phân chúng ta cần biến đổi một chuỗi các digits ASCII thành biểu diễn nhị phân của một số nguyên thập phân . Chúng ta sẽ viết thủ tục INDEC để làm việc này .

Trong thủ tục OUTDEC chúng ta chia lặp cho 10d . Trong thủ tục INDEC chúng ta sẽ nhân lặp 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 đây chúng ta sẽ xây dựng thủ tục 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 “?” để nhắc người dùng gõ vào dấu + hoặc - , theo sau đólà một chuỗi các digit và kết thúc là ký tự CR . Nếu người dùng gõ vào một ký tự không phải là 0 đến 9 thì thủ tục sẽ nhảy xuống dòng mới và bắt đầu lại từ đầu . Với những yêu cầu như trên đây thủ tục 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
IND_IF
read acharacter
UNTIL character is a carriage return
IF negative = true
        then
total = - total
END_IF
```

Thủ tục có thể mã hoá như 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 equvalent of number
        PUSH BX ; Save regiter
        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 NDEC cho nhập thập phân và OUTDEC cho xuất thập phân như sau :

```
TITLE PGM6_4.ASM
.MODEL SMALL
.STACK 100h
.CODE
MAIN PROC
; input a number
```

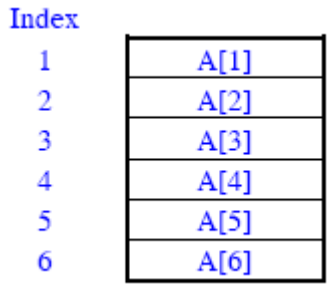
```
        CALL INDEC
        PUSH AX ; save number
; move cursor to a new line
        MOV AH,2
        MOV DL,0DH
        INT 21h
        MOV DL,0Ah
        INT 21H
;output a number
        POP AX
        CALL OUTDEC
; dos exit
        MOV AH,4CH
        INT 21H
        MAIN ENDP
        INCLUDE A:\PGM6_1.ASM ; include outdec
        INCLUDE A:\PGM6-2.ASM ; include indec
END MAIN
```

Chương 7: MẢNG VÀ CÁC CHẾ ĐỘ ĐỊA CHỈ

Trong chương này chúng ta sẽ đề cập đến mảng một chiều và các kỹ thuật xử lý mảng trong Assembly . Phần còn lại củachương này sẽ trình bày các chế độ địa chỉ.

7.1 Mảng một chiều

Mảng một chiều là một danh sách các phần tử cùng loại và có trật tự . Có trật tự có nghĩa là có phần tử thứ nhất , phần tử thứ hai , phần tử thứ ba ... Trong toán học , nếu A là một mảng thì các phần tử của mảng được định nghĩa làA[1},
A[2] , A[3} ... Hình vẽ là dưới đây là mảng A có 6 phần tử .



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

```
MSG   DB    'abcde'
```

hoặc một mảng từ W gồm 6 số nguyên mà giá trị ban đâù 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 định nghĩa một mảng mà 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 1  00 DUP (0) ; tạo một mảng 100 từ mà giá trị ban đâù 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 đương 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 chỉ ra số byte của một phần tử của mảng (S=1 đối với mảng byte và S=2 đối 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+9x2=W+18

Phần tử thứ 25 là W[25] có địa chỉ là W+24x2=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 chỉ ra toán hạng trong lệnh gọi là chế độ địa chỉ . Các chế độ địa chỉ thường dùng là :

- Chế độ địa chỉ bằng thanh ghi (register mode) : toán hạng là thanh ghi
- Chế độ địa chỉ tức thời (immediate mode) : toán hạng là hằng số
- Chế độ địa chỉ 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 địa chỉ khác là :

- Chế độ địa chỉ gián tiếp bằng thanh ghi (register indirect mode)
- Chế độ địa chỉ cơ sở (based mode)
- Chế độ địa chỉ chỉ số (indexed mode)
- Chế độ địa chỉ chỉ số sơ sở (based indexed mode)

7.2.1 Chế độ địa chỉ gián tiếp bằng thanh ghi

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

Ví dụ : giả sử rằng SI = 100h và từ nhớ 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 tương ứng 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 hợp lý , offset nguồn và kết qủa của các lệnh hợp lý .

a. MOV BX,[BX]

b. MOV CX,[SI]

c. MOV BX,[AX]

d. ADD [SI],[DI]

e. INC [DI]

Chương 7 : Mảng và các chế độ địa chỉ 83

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 định nghĩa như sau :

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

Giải :

```
        XOR AX,AX ; xoá AX
        LEA SI,W ; SI trỏ tới địa chỉ cơ sở ( base) của mảmg W .
        MOV CX,10 ; CX chứa số phần tử của mảng
ADDITION:
        ADD AX,[SI] ; AX=AX + phần tử thứ nhất
        ADD SI,2 ; tăng con trỏ lên 2
```

```
        LOOP ADDITION ; lặp
```

Ví dụ 3 : Viết thủ tục để đảo ngược một mảng n từ . Điều này có nghĩa là phần tử thứ nhất sẽ đổi thành phần tử thứ n , phần tử thứ hai sẽ thành phần tử thứ n-1 ... Chúng ta sẽ dùng SI như là con trỏ của mảng còn BX chứa số phần tử của mảng (n từ)

Giải : Số lần trao đổi là N/2 lần . Nhớ rằng phần tử thứ N 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 chỉ tới phần tử thứ n
        MOV DI,SI ; DI trỏ tới từ thứ nhất
        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
; thứ n
        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] ; đưa nó lên 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 thứ n-1
        LOOP XCHG_LOOP
        POP DI
        POP SI
        POP CX
        POP BX
        POP AX
        RET
REVERSE ENDP
```

7.2.2 Chế độ địa chỉ chỉ số và cơ sở

Trong các chế độ địa chỉ này , địa chỉ offset của toán hạng có được bằng cách cộng một số gọi là displacement với nội dung của một thanh ghi .

Displacement có thể là :

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

Cú pháp của một toán hạng có thể là một trong các kiểu tương đương sau :

[register + displacement]
[displacement + register]
[register]+ displacement
[displacement]+ register
displacement[register]

Các thanh ghi phải là BX , SI , DI (địa chỉ đoạn phải là thanh ghi DS) và BP (thanh ghi SS chứa địa chỉ đoạn)

Chế độ địa chỉ được gọi là cơ sở (based) nếu thanh ghi BX(base register) hoặc BP (base pointer) được dùng .

Chế độ địa chỉ được gọi là chỉ số (indexed) nếu thanh ghi SI(source index) hoặc DI (destination index) được dùng .

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

```
MOV AX,W[BX}
```

displacement là địa chỉ offset của biến W . Lệnh này sẽ di chuyển phần tử có địa chỉ W+4 vào thanh ghi AX . Lệnh này cũng có thể viết dưới các dạng tương đương 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 địa chỉ 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 địa chỉ cơ sở có thể viết lại code cho bài toán tính tổng 10 phần tử của mảng như sau :

```
XOR AX,AX ; xoá AX
XOR BX,BX ; xoá BX ( thanh ghi cơ sở )
```

```
        MOV CX,10 ; CX= số phần tử =10
ADDITION:
        ADD AX,W[BX} ; sum=sum+element
        ADD BX,2 ; trỏ tới phần tử thứ hai
        LOOP ADDITION
```

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

```
ALPHA    DW    0123h,0456h,0789h,0ADCDH
```

trong đoạn được địa chỉ bởi DS và giả sử rằng :

BX =2 [0002]= 1084h

SI=4 [0004]= 2BACh

DI=1

Chỉ ra các lệnh nào sau đây là hợp lệ, địa chỉ offset nguồn và số được chuyển .

a. MOV AX,[ALPHA+BX]

b. MOV BX,[BX+2]

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 đây cho thấy một mảng được xử lý như thế nào bởi chế độ địa chỉ chỉ số và cơ sở .

Ví dụ : Đổi các ký tự viết thường 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 chỉ số cho ký tự
TOP:
        CMP MSG[SI], ‘ ’ ; blank?
        JE NEXT ; yes , skip
        AND MSG[SI],0DFH ; đổi thành chữ hoa
NEXT:
```

INC SI ; chỉ 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 tương ứng với toán hạng kia . Ví dụ , ASM sẽ thực hiện lệnh MOV AX,1 như là lệnh toán hạng từ . Tương tự , ASM sẽ thực hiện lệnh MOV BH,5 như là lệnh byte . Tuy nhiên , lệnh

MOV [BX],1 là không hợp lệ vì ASM không biết toán hạng chỉ bởi thanh ghi BX là toán hạng byte hay toán hạng từ . Có thể khắc phục điều này bằng toán tử PTR như 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 tự t thành T trong chuỗi được định nghĩa bởi :

MSG DB ‘this is a message’

Cách 1: Dùng chế độ địa chỉ gián tiếp thanh ghi :

LEA SI,MSG ; SI trỏ tới MSG

MOV BYTE PTR [SI],’T’ ; thay t bằng T

Cách 2 : Dùng chế độ địa chỉ chỉ số :

XOR SI,SI ; xoá SI

MOV MSG[SI],’T ’ ; thay t bởi T

Ở đây không cần dùng PTR vì MSG là biến byte .

Nói chung toán tử PTR được dùng để khai báo loại (type) của toán hạng . Cú pháp chung của nó như sau:

Type PTR address_expression

Trong đó Type : byte , word , Dword

Addres_expression : là các biến đã được 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 chỉ bằng một lệnh MOV duy nhất . Có thể dùng lệnh sau :

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

Toán tử giả LABEL

Có một cách khác để giải quyết vấn đề xung đột về loại toán hạng như trên bằng cách dùng toán tử giả LABEL như sau đây :

MONEY LABEL WORD

DOLLARS DB 1AH

CENTS DB 52H

Các lệnh trên đây khai báo biến MONEY là biến từ với 2 thành phần là DOLLARS và CENTS . Trong đó DOLLRAS có cùng địa chỉ với MONEY . Lệnh

MOV AX, MONEY

Tương đương với 2 lệnh :

MOV AL, DOLLARS

MOV AH, CENTS

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

```
.DATA
        A       DW      1234h
        B       LABEL   BYTE
        D       W       5678h
        C       LABEL   WORD
        C1      DB      9Ah
        C2      DB      0bch
```

Hãy cho biết các lệnh nào sau đây là hợp lệ và kết qủa của lệnh .

a. MOV AX,B

b. MOV AH,B

c. MOV CX,C

d. MOV BX,WORD PTR B

e. MOV DL,WORD PTR C

f. MOV AX, WORD PTR C1

Giải :

a. không hợp lệ

b. hợp lệ , 78h

c. hợp lệ , 0BC9Ah

d. hợp lệ , 5678h

e. hợp lệ , 9Ah

f. hợp lệ , 0BC9Ah

7.2.4 Chiếm đoạn (segment override)

Trong chế độ địa chỉ gián tiếp bằng thanh ghi , các thanh ghi con trỏ BX,SI hoặc DI chỉ ra địa chỉ offset còn thanh ghi đoạn là DS . Cũng có thể chỉ ra 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 toán hạng nguồn là ES:0100h

Việc chiếm đoạn cũng có thể dùng với chế độ địa chỉ chỉ số và chế độ địa chỉ cơ sở .

7.2.5 Truy xuất đoạn stack

Như chúng ta đã nói trên đây khi BP chỉ ra một địa chỉ offset trong chế độ địa chỉ gián tiếp bằng thanh ghi , SS sẽ cung cấp số đoạn . Điều này có nghĩa là có thể dùng 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 như 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 só lớn nhất trong 2 só A[1] và A[2}. Gán só lớn nhất cho A[2}

Ví dụ : giả sử rằng 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 đây là chương trình để sort các phần trong mộ mảng . Chúng ta sẽ dùng thủ tục SELECT để chọn phần tử trên mảng . Thủ tục SELECT sẽ goị thủ tục 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 vàthủ 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 ; cất địa chỉ offfset 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 tới phần tử thứ nhất
        MOV AL,[DI] ; AL chứa phần tử thứ 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 thứ nhất
        ING 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
; đổi chỗ 2 phần tử của mảng
; input : SI= phần tử thứ nhất
; DI = phần tử thứ hai
; 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 qủa .

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 bộ nhớ là 1 chiều vì vậy các phần tử của mảng 2 chiều phải được lưu trữ trên bộ nhớ theo kiểu lần lượt . Có 2 cách được dùng :

- Cách 1 là lưu trữ theo thứ tự dòng : 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 cột : 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
chứá 90,100,110,120 trên dòng 3

Theo trật tự hàng chúng được lưu trữ như sau :

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

Theo trật tự cột chúng 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 biên dịch mảng 2chiều theo trật tự dòng .

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

Nếu các thành phần của một hàng được xử lý lần lượt thì cách lưu trữ theo trật tự hàng được dùng . Ngược lại thì dùng cách lưu trữ theo trật tự cột .

Xác định một phần tử trên mảng 2 chiều :

Giả sử rằng mảng A gồm MxN phần tử lưu trữ theo trật tự dòng . Goị S là độ lớn của một phần tử : S=1 nếu phần tử là byte , S=2 nếu phần tử là từ . Để tìm phần tử thứ A[i,j] thì cần tìm : hàng i và tìm phần tử thứ j trên hàng này . Như 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)xSxN .

Bước 2: Phần tử thứ 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 được lưu trữ theo trật tự hàng thì phần tử A[i,j] có địa chỉ được xác định như sau :

A+((i-1)xN + (j-1))x S (1)

Tương tự nếu lưu trữ theo trật tự cột thì phần tử A[i,j] có địa chỉ như sau :

A+(i-1)+(j--)xM)xS (2)

Ví dụ : Giả sử A là mảng MxN phần tử kiểu từ (S=2) được lưu trữ theo kiểu trật tự hàng . Hỏi :

Hàng i bắt đầu tại địa chỉ nào ?

Cột j bắt đầu tại địa chỉ nào ?

Hai phần tử trên một cốt cách nhau bao nhiêu bytes

Giải :

Hàng i bắt đầu tại A[i,1] theo công thức (1) thì nó có địa chỉ là : A+(i-1)xNx2

Cột j bắt đầu tại A[1,j] theo công thức (1) thì nó có địa chỉ : A+(j-1)x2

Vì có N cột nên 2 phần tử trên cùng một cột cách nhau 2xN byte .

7.5 Chế độ địa chỉ chỉ số cơ sở

Trong chế độ 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 (tuỳ chọn)
4. một hằng âm hoặc dương (tuỳ chọn)

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

1. variable[base_register][index_register]
2. [base_register + index_register + variable + constant]
3. variable [base_register + index_register + constant]
4. constant [base _ register + index_register + variable]

Trật tự của các thành phần trong dấu ngoặc là tuỳ ý .

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

MOV AX, W[BX][SI]

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

Lệnh này cũng có thể viết theo 2 cách sau :

MOV AX,[W+BX+SI]

MOV AX,W[BX+SI]

Chế độ địa chỉ chỉ số cơ sở thường được dùng để xử lý mảng 2 chiều như ví dụ sau : Giả sử rằng A là mảng 5x7 từ được lưu trữ theo trật tự dòng . Viết đoạn mã dùng chế độ địa chỉ chỉ số để :

1) xóa dòng 3

2) xoá cột 4

Giải :

1) Dòng i bắt đầu tại A+(i-1)xNx2 . Như vậy dòng 3 bắt đầu tại A+(2-1)x7x2 = A + 28 . Có thể xóa dòng 3 như sau :

```
MOV BX,28 ; BX chỉ đến đầu dòng 3
XOR SI,SI ; SI sẽ chỉ mục cột
MOV CX,7 ; CX= số phần tử của một hàng
CLEAR:
MOV A[BX][SI],0 ; xoá A[3,1]
ADD SI,2 ; đến cột tiếp theo
LOOP CLEAR
```

2) Cột j bắt đầu tại địa chỉ A + (j-1)x2 . Vậy cột 4 bắt đầu tại địa chỉ A+(4- 1)x2 = A+ 6 . Hai phần tử trên một cột cách nhau Nx2 byte , ở đây N=7 , vậy 2 phần tử cách nhau 14 byte . Có thể xóa cột 4 như sau :

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

7.6 Ưng dụng để tính trung bình

Giả sử một lớp gồm 5 sinh viên và có 4 môn thi . Kết qủa 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 viết1 chương trình tính điểm trung bình cho mỗi bài thi . Để làm điều này 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 đó bước 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 chỉ đến cột thứ 4
       XOR BX,BX ; BX chỉ hàng thứ nhất
       XOR AX,AX ; AX chứa tổng theo cột
; Tổng điểm trên cột j
FOR:
       ADD AX , SCORES[BX+SI]
       ADD BX,8 ; BX chỉ đến hàng thứ 2
       LOOP FOR
; end_for
; tính trung bình cột j
       XOR DX,DX ; xoá phần cao của số bị chia (DX:AX)
       DIV FIVE ; AX = AX/5
```

```
        MOV AVG[SI],AX ; cất kết qủa trên mảng AVG
        SUB SI,2 ; đến cột tiếp
; un til j=0
        JNL REPEAT
;DOS EXIT
        MOV AH,4CH
        INT 21H
MAIN ENDP
        END MAIN
```

Sau khi biên dịch chương teình có thể dùng DEBUG để chạy và xem kết qủa 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 nhưng IBM Mainframes dùng EBCDIC (Extended Binary Coded Decimal Interchange Code) . Để chuyển một chuỗi ký tự đã được mã hoá 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) được dùng để đổi 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 tạo 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 nó bằng mã ASCII của số hex tương đương nó , 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 , để đổi 0Ch thành “C” , chúng ta thực hiện các lệnh sau :

```
        MOV AL,0Ch ; số cần biến đổi
        LEA BX,TABLE ; BX chuá 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 khỏang 0 đến 15 thì XLAT sẽ cho một giá trị sai .

Ví dụ : Mã hoá và giải mã một thông điệp mật

Chương trình này sẽ :

Nhắc nhở người dùng nhập vào một thông điệp

Mã hoá nó dưới dạng không nhận biết được ,

In chúng ra ở dòng tiếp theo

Dịch chúng trở lại dạng ban đầu rồi in chúng ở dòng tiếp theo

Khi chạy ct màn hình 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 anew 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 ( ' '), '
XQPOGHZBCADEIJUVFMNKLRSTWY'
DB 37 DUP (' ') ; 128 ký tự của bảng mã ASCII
CODED DB 80 dup ('$') ; 80 ký tự được gõ vào
DECODE_KEY DB 65 DUP (' '),
'JHIKLQEFMNTURSDCBVWXOPYAZG'
DB 37 DUP (' ')
PROMPT DB 'ENTER A MESSAGE :',0DH,0AH,'$'
CRLF DB 0DH,0AH,'$'
.CODE
MAIN PROC
    MOV AX,@DATA
    MOV DS, AX
; in dấu nhắc
MOV AH,9
    LEA DX,PROMPT
    INT 21H
; đọc và mã hoá ký tự
    MOV AH,1
    LEA BX,CODE_KEY ; BX chỉ tới CODE_KEY
```

```
        LEA DI, CODED ; DI chỉ tới thông điệp đã mã hoá
WHILE_:
        INT 21h ; đọc ký tự vào AL
        CMP AL,0DH ; có phải là ký tự CR
        JE ENDWHILE ; đúng , đến phần in thông điệp đã mã hoá
        XLAT ; mã hoá ký tự
        MOV [DI],AL ; cất ký tự trong CODE
        JMP WHILE_ ; xử lý ký tự tiếp theo
; xuống hàng
        MOV AH,9
        LEA DX,CRLF
        INT 21H
; in thông điệp đã mã hoá
        LEA DX,CODED
        INT 21H
; xuống hàng
        LEA DX,CRLF
        INT 21H
; giải mã thông điệp và in nó
        MOV AH,2
        LEA BX,DECODE_KEY ; BX chứa địa chỉ bảng giải mã
        LEA SI,CODED ; SI chỉ tới thông điệp đã mã hoá
WHILE1:
        MOV AL,[SI] ; lấy ký tự từ thông điệp đã mã hoá
        CMP AL.'$' ; có phải cuối thông điệp
        JE ENDWHILE1 ; kết thúc
        XLAT ; giải mã
        MOV DL,AL ;đặt ký tự vào DL
        INT 21H ; in ký tự
        INC SI ; SI=SI+1
        JMP WHILE1 ; tiếp tục
        ENDWHILE1:
        MOV AH,4CH
        INT 21H
MAIN ENDP
        END MAIN
```

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

```
; ALPHABET ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Cho biết bảng chữa cái tiếng Anh

CODE_KEY DB 65 DUP (’ ‘), ‘ XQPOGHZBCADEIJUVFMNKLRSTWY’
DB 37 DUP (‘ ‘)

Khai báo 128 ký tự của bảng mã ASCII , trong đó thứ tự các ký tự hoa là tuỳ ý .

CODED DB 80 dup (‘$’) 80 ký tự được gõ vào , giá trị ban đầu là $ để có thể in bằng hàm 9 ngắt 21h

DECODE_KEY DB 65 DUP (‘ ‘), ‘JHIKLQEFMNTURSDCBVWXOPYAZG’
DB 37 DUP (‘ ‘)

Bảng giải mã được thiết lập theo cách mã hoá , nghĩa là trong phần mã hoá chúng ta đã mã hoá ‘A’ thành ‘X’ vì vậy khi giải mã ‘X’ phải giải mã thành ‘A’ ... Các ký tự gõ vào không phải là ký tự hoa đếu được chuyển thành ký tự trống.

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

Nội dung môn học

1. Giới thiệu chung về hệ vi xử lý
2. Bộ vi xử lý Intel 8088/8086
3. Lập trình hợp ngữ cho 8086
4. Tổ chức vào ra dữ liệu
5. Ngắt và xử lý ngắt
6. Truy cập bộ nhớ trực tiếp DMA
7. Các bộ vi xử lý trên thực tế

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

3.1 Giới thiệu khung của chương trình hợp ngữ

3.2 Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC

3.3 Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ

3.4 Một số chương trình cụ thể

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

- 3.1 Giới thiệu khung của chương trình hợp ngữ
  - 3.1.1 Cú pháp của chương trình hợp ngữ
  - 3.1.2 Dữ liệu cho chương trình
  - 3.1.3 Biến và hằng
  - 3.1.4 Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

- 3.1 Giới thiệu khung của chương trình hợp ngữ
  - ☑ 3.1.1 Cú pháp của chương trình hợp ngữ
  - ❑ Dữ liệu cho chương trình
  - ❑ Biến và hằng
  - ❑ Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.1 Cú pháp của chương trình hợp ngữ

```
1.  .Model Small
2.  .Stack 100
3.  .Data
4.          Tbao DB 'Chuoi da sap xep:', 10, 13
5.          MGB DB 'a', 'Y', 'G', 'T', 'y', 'Z', 'U', 'B', 'D', 'E',
6.               DB '$'
7.  .Code
8.  MAIN Proc
9.          MOV AX, @Data                       ;khoi dau DS
10.         MOV DS, AX
11.         MOV BX, 10                          ;BX: so phan tu cua mang
12.         LEA         DX, MGB                 ;DX chi vao dau mang byte
13.         DEC         BX                      ;so  vong so sanh phai lam
14. LAP:    MOV SI, DX                          ; SI chi vao dau mang
15.         MOV         CX, BX                  ; CX so lan so cua vong so
16.         MOV         DI, SI                  ;gia su ptu dau la max
17.         MOV         AL, [DI]                ;AL chua phan tu max
18. TIMMAX:
19.         INC SI                              ;chi vao phan tu ben canh
20.         CMP         [SI], AL                ; phan tu moi > max?
21.         JNG         TIEP                    ;khong, tim max
22.         MOV         DI, SI                  ; dung, DI chi vao max
23.         MOV         AL, [DI]                            ;AL chua phan tu max
24. TIEP:  LOOP TIMMAX                          ;tim max cua mot vong so
25.         CALL  DOICHO                        ;doi cho max voi so moi
26.         DEC         BX                      ;so vong so con lai
27.         JNZ         LAP                     ;lam tiep vong so moi
28.         MOV         AH, 9                               ; hien thi chuoi da sap xep
29.         LEA         DX, Tbao
30.         INT         21H
31.         MOV         AH, 4CH                 ;ve DOS
32.         INT         21H
33. MAIN    Endp
34. DOICHO Proc
35.         PUSH AX
36.         MOV         AL, [SI]
37.         XCHG        AL, [DI]
38.         MOV         [SI], AL
39.         POP         AX
40.         RET
41. DOICHO Endp
42.         END MAIN
```

khai báo kiểu kích thước bộ nhớ

khai báo đoạn ngăn xếp

khai báo đoạn dữ liệu

khai báo đoạn mã lệnh

bắt đầu chương trình chính

chú thích bắt đầu bằng dấu ;

kết thúc chương trình chính

bắt đầu chương trình con

kết thúc đoạn mã

© DHBK 2005

3.1.1 Cú pháp của chương trình hợp ngữ

- Tên Mã lệnh Các toán hạng ; chú giải
- Chương trình dịch không phân biệt chữ hoa, chữ thường
- Trường tên:
  - chứa các nhãn, tên biến, tên thủ tục
  - độ dài: 1 đến 31 ký tự
  - tên không được có dấu cách, không bắt đầu bằng số
  - được dùng các ký tự đặc biệt: ? . @ _ $ %
  - dấu . phải được đặt ở vị trí đầu tiên nếu sử dụng
  - Nhãn kết thúc bằng dấu :
  - Ví dụ:

```
TWO_WORD
?1
two-word
.@?
1word
Let's_go
```

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

- 3.1 Giới thiệu khung của chương trình hợp ngữ
  - ❑ Cú pháp của chương trình hợp ngữ
  - ☑ 3.1.2 Dữ liệu cho chương trình
  - ❑ Biến và hằng
  - ❑ Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.2 Dữ liệu cho chương trình

- Dữ liệu:
  - các số hệ số 2: 0011B
  - hệ số 10: 1234
  - hệ số 16: 1EF1H, 0ABBAH
  - Ký tự, chuỗi ký tự: ‘A’, “abcd”

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

- 3.1 Giới thiệu khung của chương trình hợp ngữ
  - ❑ Cú pháp của chương trình hợp ngữ
  - ❑ Dữ liệu cho chương trình
  - ☑ 3.1.3 Biến và hằng
  - ❑ Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.3 Biến và hằng

- DB (Define Byte): định nghĩa biến kiểu byte
- DW (Define Word): định nghĩa biến kiểu từ - 2 byte
- DD (Define Double word): định nghĩa biến kiểu từ kép - 4 byte

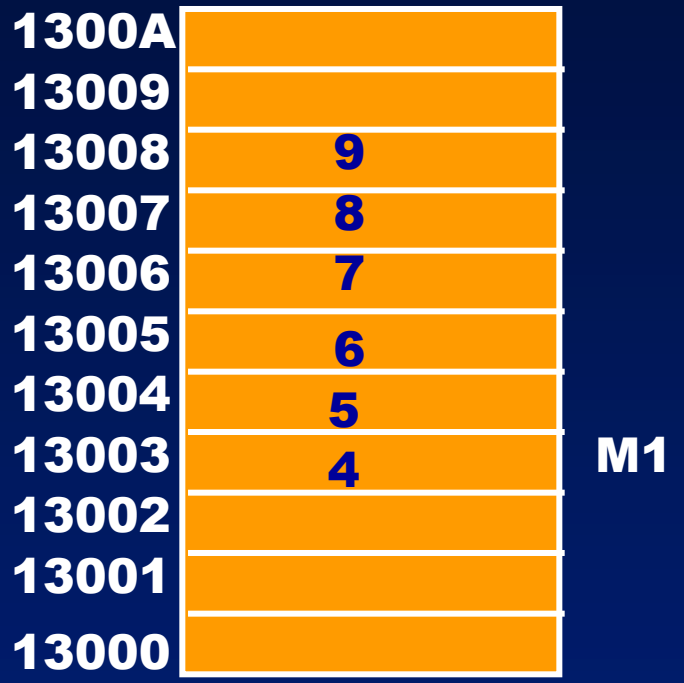
- Biến byte:
  - Tên DB gia_trị_khởi đầu
  - Ví dụ:
    - ⇨ B1 DB 4
    - ⇨ B1 DB ?
    - ⇨ C1 DB '$'
    - ⇨ C1 DB 34

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.3 Biến và hằng

- **Biến từ:**
  - Tên DW gia_trị_khởi đầu
  - Ví dụ:
    - ⇨ W1 DW 4
    - ⇨ W2 DW ?
- **Biến mảng:**
  - M1 DB 4, 5, 6, 7, 8, 9
  - M2 DB 100 DUP(0)
  - M3 DB 100 DUP(?)
  - M4 DB 4, 3, 2, 2 DUP (1, 2 DUP(5), 6)
  - M4 DB 4, 3, 2, 1, 5, 5, 6, 1, 5, 5, 6



© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.3 Biến và hằng

- **Biến mảng 2 chiều:**

$$\begin{pmatrix} 1 & 6 & 3 \\ 4 & 2 & 5 \end{pmatrix}$$

- M1 DB 1, 6, 3
  DB 4, 2, 5

- M2 DB 1, 4
  DB 6, 2
  DB 3, 5

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.3 Biến và hằng

- Biến kiểu xâu ký tự
  - STR1 DB 'string'
  - STR2 DB 73h, 74h, 72h, 69h, 6Eh, 67h
  - STR3 DB 73h, 74h, 'r', 'i', 6Eh, 67h

- Hằng có tên
  - Có thể khai báo hằng ở trong chương trình
  - Thường được khai báo ở đoạn dữ liệu
  - Ví dụ:
    - ⇨ CR EQU 0Dh ; là carriage return
    - ⇨ LF EQU 0Ah ; LF là line feed
    - ⇨ CHAO EQU 'CR Hello'
    - ⇨ MSG DB CHAO, '$'

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

- 3.1 Giới thiệu khung của chương trình hợp ngữ
  - ❑ Cú pháp của chương trình hợp ngữ
  - ❑ Dữ liệu cho chương trình
  - ❑ Biến và hằng
  - ☑ 3.1.4 Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.4 Khung của chương trình hợp ngữ

- **Khai báo quy mô sử dụng bộ nhớ**
  - ❑ **.MODEL Kiểu kích thước bộ nhớ**
  - ❑ **Ví dụ: .Model Small**

Kiểu	Mô tả
Tiny (hẹp)	mã lệnh và dữ liệu gói gọn trong một đoạn
Small (nhỏ)	mã lệnh nằm trong 1 đoạn, dữ liệu 1 đoạn
Medium (trung bình)	mã lệnh nằm trong nhiều đoạn, dữ liệu 1 đoạn
Compact (gọn)	mã lệnh nằm trong 1 đoạn, dữ liệu trong nhiểu đoạn
Large (lớn)	mã lệnh nằm trong nhiều đoạn, dữ liệu trong nhiều đoạn, không có mảng nào lớn hơn 64 K
Huge (đồ sộ)	mã lệnh nằm trong nhiều đoạn, dữ liệu trong nhiều đoạn, các mảng có thể lớn hơn 64 K

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.4 Khung của chương trình hợp ngữ

- Khai báo đoạn ngăn xếp
  - .Stack kích thước (bytes)
  - Ví dụ:
    - ⇨ .Stack 100 ; khai báo stack có kích thước 100 bytes
  - Giá trị ngầm định 1 KB
- Khai báo đoạn dữ liệu:
  - .Data
  - Khai báo các biến và hằng
- Khai báo đoạn mã
  - .Code

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.4 Khung của chương trình hợp ngữ

- Khung của chương trình hợp ngữ để dịch ra file .EXE

```
.Model   Small
.Stack   100
.Data
         ;các định nghĩa cho biến và hằng
.Code
MAIN     Proc
         ;khới đầu cho DS
         MOV     AX, @data
         MOV     DS, AX
         ;các lệnh của chương trình

         ;trở về DOS dùng hàm 4CH của INT 21H
         MOV     AH, 4CH
         INT     21H
MAIN     Endp
         ;các chương trình con nếu có
END MAIN
```

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.4 Khung của chương trình hợp ngữ

- Chương trình Hello.EXE

```
.Model      Small
.Stack      100
.Data
            CRLF      DB        13,10,'$'
            MSG       DB        'Hello! $'
.Code
MAIN        Proc
            ;khởi đầu cho DS
            MOV       AX, @data lay dia chi segment cua data

            MOV       DS, AX
            ;về đầu dòng mới dùng hàm 9 của INT 21H
            MOV       AH,9
            LEA       DX, CRLF
            INT       21H
            ;Hiển thị lời chào dùng hàm 9 của INT 21H
            MOV       AH,9
            LEA       DX, MSG
            INT       21H
            ;về đầu dòng mới dùng hàm 9 của INT 21H
            MOV       AH,9
            LEA       DX, CRLF
            INT       21H
            ;trở về DOS dùng hàm 4CH của INT 21H
            MOV       AH, 4CH
            INT       21H
MAIN        Endp
```

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

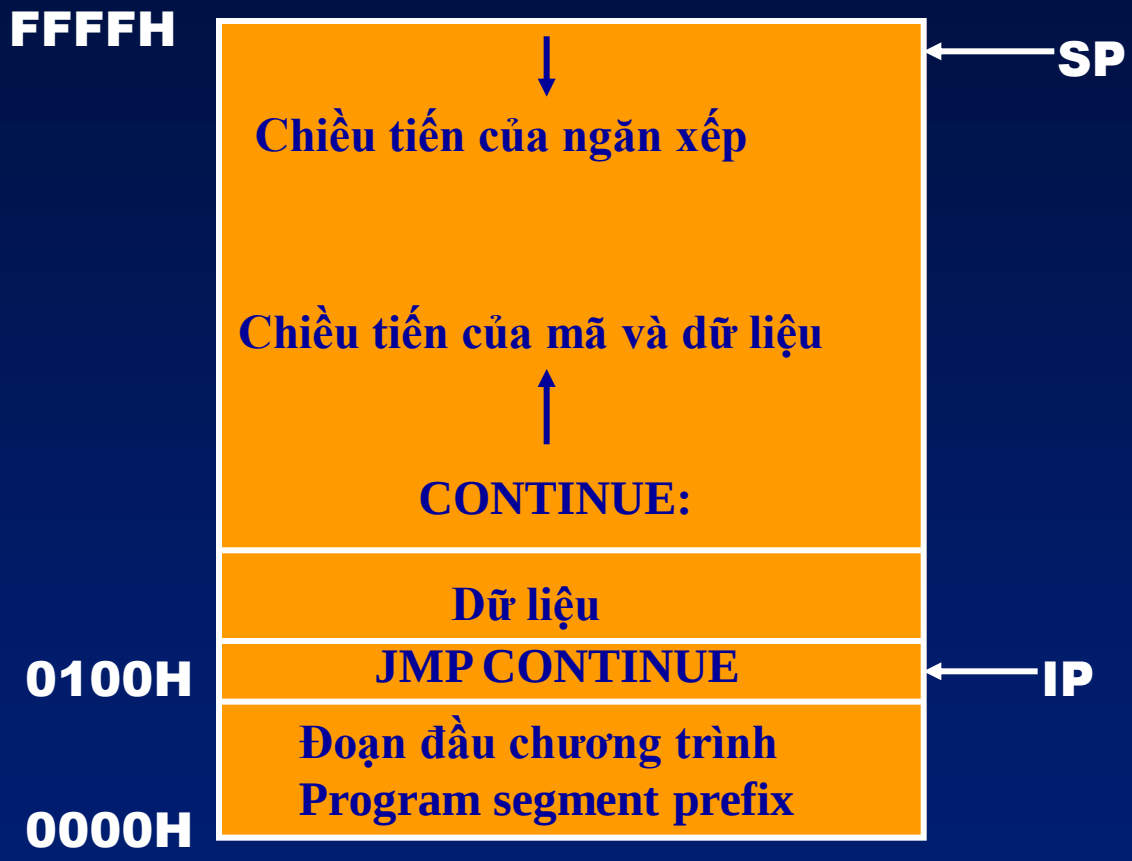
3.1.4 Khung của chương trình hợp ngữ

- Khung của chương trình hợp ngữ để dịch ra file .COM
  - Chỉ có 1 đoạn cho Code,Data,Stack
  - Trở về DOS bằng INT 20H

```
.Model    Tiny
.Code
          ORG      100h
START: JMP         CONTINUE
          ;các định nghĩa cho biến và hằng
CONTINUE:
MAIN      Proc
          ;các lệnh của chương trình
          INT      20H       ;trở về DOS
MAIN      Endp
          ;các chương trình con nếu có
END START
```

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.4 Khung của chương trình hợp ngữ



Khoa ĐIỆN TỬ - VIỄN THÔNG

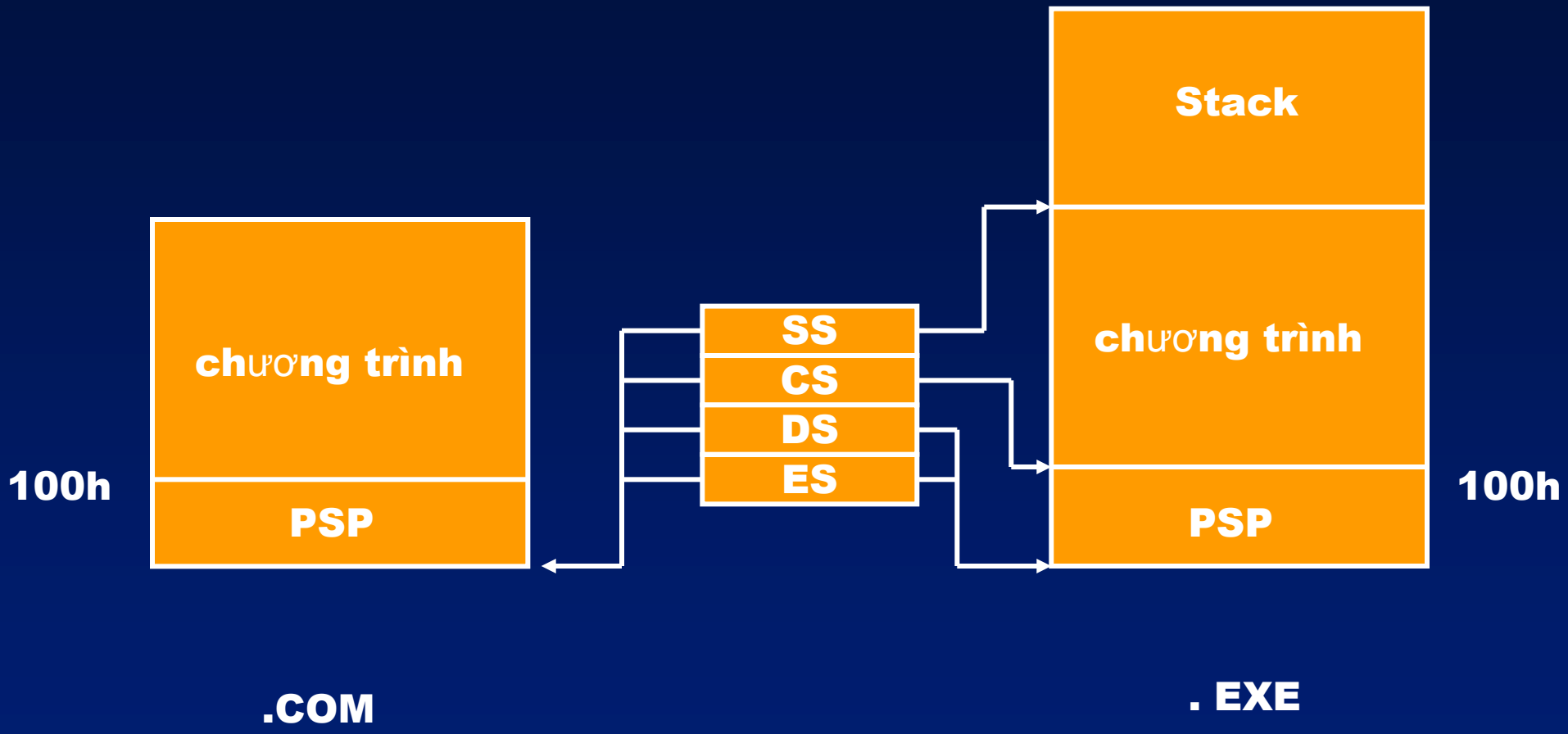
3.1.4 Khung của chương trình hợp ngữ

- Chương trình Hello.COM

```
.Model      Tiny
.Code
            ORG       100H
START: JMP CONTINUE
            CRLF      DB        13,10,'$'
            MSG       DB        'Hello! $'
CONTINUE:
MAIN        Proc
            ;về đầu dòng mới dùng hàm 9 của INT 21H
            MOV       AH,9
            LEA       DX, CRLF
            INT       21H
            ;Hiển thị lời chào dùng hàm 9 của INT 21H
            MOV       AH,9
            LEA       DX, MSG
            INT       21H
            ;về đầu dòng mới dùng hàm 9 của INT 21H
            MOV       AH,9
            LEA       DX, CRLF
            INT       21H
            ;trở về DOS
            INT       20H
MAIN        Endp
            END START
```

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.1.4 Khung của chương trình hợp ngữ



© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

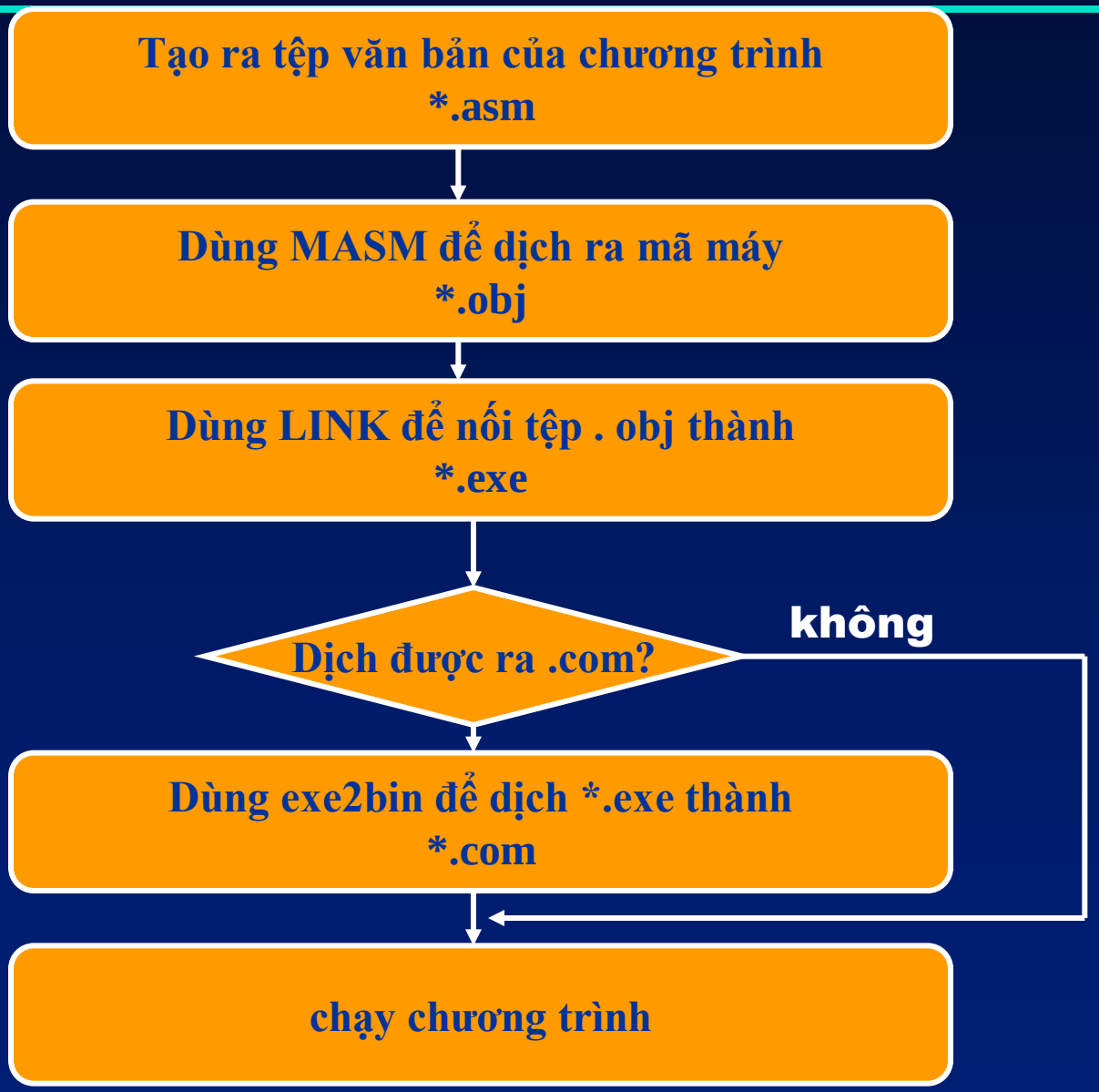
Giới thiệu khung của chương trình hợp ngữ

3.2 Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC

Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ

Một số chương trình cụ thể

3.2 Cách tạo một chương trình hợp ngữ



© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

© DHBK 2005

3.3.1 Cấu trúc lựa chọn If-then

- If điều_kiện then công_việc
- Ví dụ: Gán cho BX giá trị tuyệt đối của AX

```
                                    ; If AX<0
          CMP     AX, 0             ; AX<0 ?
          JNL     End_if  ; không, thoát ra
                          ; then
          NEG     AX      ; đúng, đảo dấu
End_if:   MOV     BX, AX            ;gán
```

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.3.1 Cấu trúc lựa chọn If-then-else

- If điều_kiện then công_việc1 else công_việc2
- Ví dụ: if AX<BX then CX=0 else CX=1

```
        ; if AX<BX
        CMP         AX, BX          ; AX<BX ?
        JL          Then_   ; đúng, CX=0
        ;else
        MOV         CX, 1   ; sai, CX=1
        JMP         End_if
Then_: MOV      CX, 0;
End_if:
```

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.3.1 Cấu trúc lựa chọn Case

- **Case Biểu thức**

  Giá trị 1: công việc 1

  Giá trị 2: công việc 2

  ...

  Giá trị N: công việc N

  END CASE

- Ví dụ:

Nếu AX<0 thì CX=-1
Nếu AX=0 thì CX=0
Nếu AX>0 thì CX=1

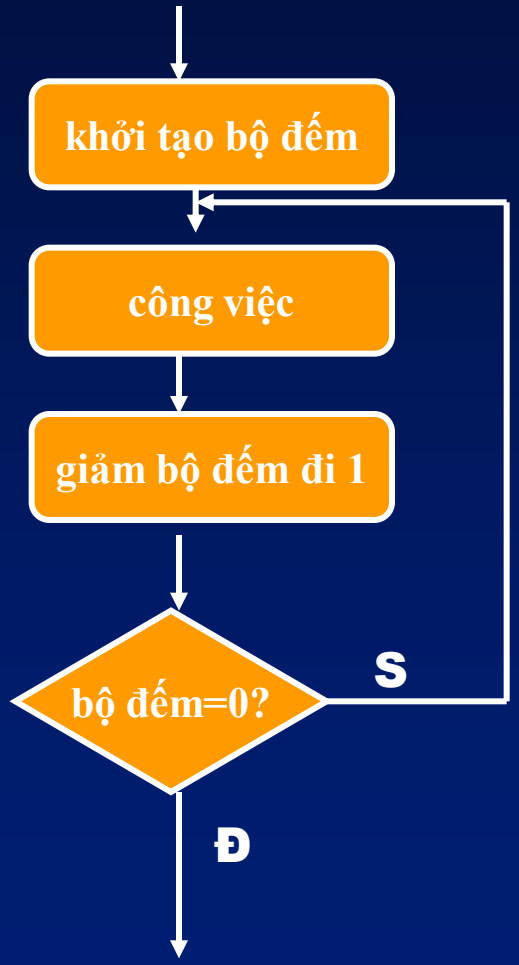
```
      CMP       AX, 0    ;
      JL        AM       ; AX<0
      JE        Khong    ; AX=0
      JG        DUONG; AX>1
AM:   MOV       CX, -1
      JMP       End_case
Khong: MOV      CX, 0
      JMP       End_case

DUONG: MOV      CX, 1
End_case:
```

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.3.2 Cấu trúc lặp For-Do

- For số lần lặp Do công việc



ví dụ: Hiển thị một dòng ký tự $ trên màn hình

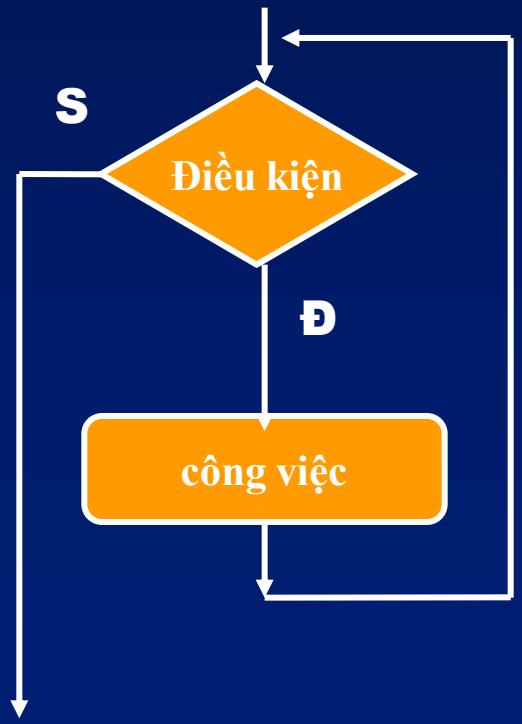
```
        MOV CX, 80              ;số lần lặp
        MOV  AH,2               ;hàm hiển thị
        MOV   DL,'$';DL chứa ký tự cần hiển thị
HIEN: INT 21H                   ; Hiển thị
      LOOP   HIEN
;End_for
```

© DHBK 2005

3.3.2 Cấu trúc lặp While-Do

- **While** điều kiện **Do** công việc

ví dụ:
Khởi tạo AX=0, BX=0
Trong khi AX<>10 thì
BX=BX+1 và AX=AX+2



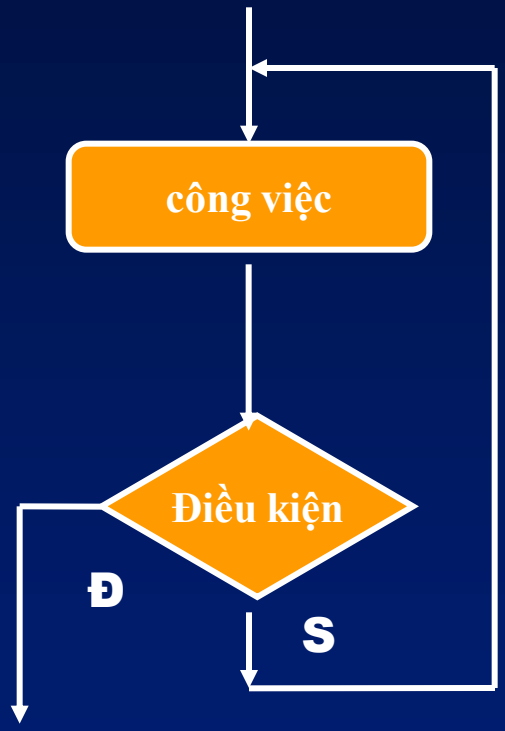

```
      XOR AX, AX          ;AX=0
      XOR BX, BX          ;BX=0
TIEP:
            CMP AX,10          ;so sánh AX với 10
            JE End_while    ;kết thúc nếu AX=10
            INC BX             ;BX=BX+1
            ADD AX,2           ;AX=AX+2
            jmp tiep
End_while:
```

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

3.3.2 Cấu trúc lặp Repeat-until

- **Repeat** công việc **until** điều kiện



ví dụ: đọc từ bàn phím cho tới khi gặp ký tự CR thì thôi

```
        MOV  AH,1              ;hàm đọc ký tự từ bàn phím
TIEP:   INT 21H                ; đọc một ký tự vào AL
        CMP  AL, 13            ; đọc CR?
        JNE TIEP               ; chưa, đọc tiếp
End_:
```

© DHBK 2005

Khoa ĐIỆN TỬ - VIỄN THÔNG

Chương 3 Lập trình hợp ngữ với 8086

3.1 Giới thiệu khung của chương trình hợp ngữ

3.2 Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC

3.3 Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ

3.4 Một số chương trình cụ thể

3.4.1 Xuất nhập dữ liệu

- 2 cách:
  - ❑ Dùng lệnh IN, OUT để trao đổi với các thiết bị ngoại vi
    - ⇨ phức tạp vì phải biết địa chỉ cổng ghép nối thiết bị
    - ⇨ Các hệ thống khác nhau có địa chỉ khác nhau
  - ❑ Dùng các chương trình con phục vụ ngắt của DOS và BIOS
    - ⇨ đơn giản, dễ sử dụng
    - ⇨ không phụ thuộc vào hệ thống
- Ngắt 21h của DOS:
  - ❑ Hàm 1: đọc 1 ký tự từ bàn phím
    - ⇨ Vào: AH=1
    - ⇨ Ra: AL=mã ASCII của ký tự, AL=0 khi ký tự là phím chức năng
  - ❑ Hàm 2: hiện 1 ký tự lên màn hình
    - ⇨ Vào: AH=2
      DL=mã ASCII của ký tự cần hiển thị
  - ❑ Hàm 9: hiện chuỗi ký tự với $ ở cuối lên màn hình
    - ⇨ Vào: AH=9
      DX=địa chỉ lệch của chuỗi ký tự cần hiển thị
  - ❑ Hàm 4CH: kết thúc chương trình loại .exe
    - ⇨ Vào: AH=4CH

3.4.2 Một số chương trình cụ thể

- Ví dụ 1: Lập chương trình yêu cầu người sử dụng gõ vào một chữ cái thường và hiển thị dạng chữ hoa của chữ cái đó lên màn hình
  - ❑ Ví dụ:
    - ⇨ Hay nhap vao mot chu cai thuong: a
    - ⇨ A
- Ví dụ 2: Đọc từ bàn phím một số hệ hai (dài nhất là 16 bit), kết quả đọc được để tại thanh ghi BX. Sau đó hiện nội dung thanh ghi BX ra màn hình.
- Ví dụ 3: Nhập một dãy số 8 bit ở dạng thập phân, các số cách nhau bằng 1 dấu cách và kết thúc bằng phím Enter. Sắp xếp dãy số theo thứ tự tăng dần và in dãy số đã sắp xếp ra màn hình.

3.4.2 Một số chương trình cụ thể

- **Ví dụ 4: Viết chương trình cho phép nhập vào kích thước M*N và các phần tử của một mảng 2 chiều gồm các số thập phân 8 bit.**
  - ⇨ Tìm số lớn nhất và nhỏ nhất của mảng, in ra màn hình
  - ⇨ Tính tổng các phần tử của mảng và in ra màn hình
  - ⇨ Chuyển thành mảng N*M và in mảng mới ra màn hình

❑ **Giải:**

```
Hãy nhập giá trị M=
Hãy nhập giá trị N=
Nhập phần tử [1,1]=
Nhập phần tử [1,2]
.....
Số lớn nhất là phần tử [3,4]=15
Số nhỏ nhất là phần tử [1,2]=2
Tổng=256
...
```

LẬP TRÌNH HỢP NGỮ MIPS

Mục đích

- Làm quen với hợp ngữ MIPS.
- Biết cách viết, biên dịch và chạy chương trình hợp ngữ MIPS với công cụ MARS.

Tóm tắt lý thuyết

Hợp ngữ (Assembly) là ngôn ngữ lập trình bậc thấp, nó gồm tập các từ khóa và từ gợi nhớ rất gần với ngôn ngữ máy (machine code).

Mỗi kiến trúc vi xử lý đều có tập lệnh (instruction set) riêng, do đó sẽ có hợp ngữ riêng dành cho kiến trúc đó. Ở đây, ta tập trung nghiên cứu về hợp ngữ dành cho kiến trúc MIPS. Môi trường lập trình được sử dụng là chương trình MARS. MARS là môi trường lập trình giả lập giúp ta viết, biên dịch và chạy hợp ngữ MIPS trên các máy x86.

- **Cấu trúc của một chương trình hợp ngữ MIPS**

```
        .data           # khai báo biến sau chỉ thị này
...
        .text           # viết các lệnh sau chỉ thị này
main:                   # điểm bắt đầu của chương trình
…
```

- **Cách khai báo biến**

tên_biến: kiểu_lưu_trữ giá_trị

Các kiểu lưu trữ hỗ trợ: .word, .byte, .ascii, .asciiz, .space

Lưu ý: tên_biến (nhãn) phải theo sau bởi dấu hai chấm (:)

Ví dụ:

```
var1:   .word       3       # số nguyên 4-byte có giá trị khởi tạo là 3
var2:   .byte       'a','b' # mảng 2 phần tử, khởi tạo là a và b
var3:   .space      40      # cấp 40-byte bộ nhớ, chưa được khởi tạo
char_array:  .byte  'A':10  # mảng 10 ký tự được khởi tạo là 'A', có thể thay 'A' bằng 65
int_array:   .word  0:30    # mảng 30 số nguyên được khởi tạo là 0
```

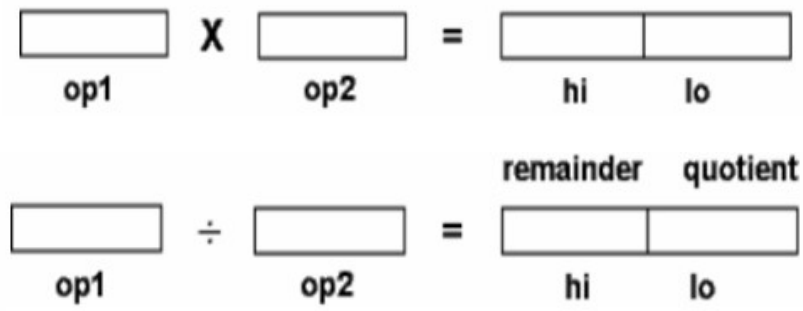
- **Các thanh ghi trong MIPS**

Thanh ghi đa năng

Số	Tên	Ý nghĩa
$0	$zero	Hằng số 0
$1	$at	Assembler Temporary
$2-$3	$v0-$v1	Giá trị trả về của hàm hoặc biểu thức
$4-$7	$a0-$a3	Các tham số của hàm
$8-$15	$t0-$t7	Thanh ghi tạm (không giữ giá trị trong quá trình gọi hàm)
$16-$23	$s0-$s7	Thanh ghi lưu trữ (giữ giá trị trong suốt quá trình gọi hàm)
$24-$25	$t8-$t9	Thanh ghi tạm
$26-27	$k0-$k1	Dự trữ cho nhân HĐH
$28	$gp	Con trỏ toàn cục (global pointer)
$29	$sp	Con trỏ stack
$30	$fp	Con trỏ frame
$31	$ra	Địa chỉ trả về

Thanh ghi HI và LO

Thao tác nhân của MIPS có kết quả chứa trong 2 thanh ghi HI và LO. Bit 0-31 thuộc LO và 32-63 thuộc HI.



Thanh ghi dấu phẩy động

MIPS sử dụng 32 thanh ghi dấu phẩy động để biểu diễn độ chính xác đơn của số thực. Các thanh ghi này có tên là : **$f0 – $f31.**

Để biểu diễn độ chính xác kép (double precision) thì MIPS sử dụng sự ghép đôi của 2 thanh ghi có độ chính xác đơn.

➢ Cú pháp tổng quát lệnh MIPS

<tên-lệnh> <r1>, <r2>, <r3>

- r1: thanh ghi chứa kết quả
- r2: thanh ghi
- r3: thanh ghi hoặc hằng số

➢ Một số lệnh MIPS cơ bản

Ghi chú:

- Rd: thanh ghi đích, Rs, Rt: thanh ghi nguồn.
- các lệnh màu xanh là các lệnh giả (pseudo instructions).

Lệnh Load / Store

Đây là các lệnh duy nhất được phép truy xuất bộ nhớ RAM trong tập lệnh của MIPS.

Cú pháp		Ý nghĩa
lw	Rd, RAM_src	Chép 1 word (4 byte) tại vị trí trong bộ nhớ RAM vào thanh ghi
lb	Rd, RAM_src	Chép 1 byte tại vị trí trong bộ nhớ RAM vào byte thấp của thanh ghi
sw	Rs, RAM_dest	Lưu 1 word trong thanh ghi vào vị trí trong bộ nhớ RAM
sb	Rs, RAM_dest	Lưu 1 byte thấp trong thanh ghi vào vị trí trong bộ nhớ RAM
li	Rd, value	Khởi tạo thanh ghi với giá trị
la	Rd, label	Khởi tạo thanh ghi với địa chỉ của nhãn

Nhóm lệnh số học:

Cú pháp		Ý nghĩa
add	Rd, Rs, Rt	Rd = Rs + Rt (kết quả có dấu)
addi	Rd, Rs, imm	Rd = Rs + imm
addu	Rd, Rs, Rt	Rd = Rs + Rt (kết quả không dấu)
sub	Rd, Rs, Rt	Rd = Rs - Rt
subu	Rd, Rs, Rt	Rd = Rs - Rt (kết quả không dấu)
mult	Rs, Rt	(Hi,Lo) = Rs * Rt
div	Rs, Rt	Lo = Rs / Rt (thương), Hi = Rs % Rt (số dư)
mfhi	Rd	Rd = Hi
mflo	Rd	Rd = Lo
move	Rd, Rs	Rd = Rs

Nhóm lệnh nhảy

Cú pháp	Ý nghĩa
j label	Nhảy không điều kiện đến nhãn 'label'
jal label	Lưu địa chỉ trở về vào $ra và nhảy đến nhãn 'label' (dùng khi gọi hàm)
jr Rs	Nhảy đến địa chỉ trong thanh ghi Rs (dùng để trở về từ lời gọi hàm)
bgez Rs, label	Nhảy đến nhãn 'label' nếu Rs >= 0
bgtz Rs, label	Nhảy đến nhãn 'label' nếu Rs > 0
blez Rs, label	Nhảy đến nhãn 'label' nếu Rs <= 0
bltz Rs, label	Nhảy đến nhãn 'label' nếu Rs < 0
beq Rs, Rt, label	Nhảy đến nhãn 'label' nếu Rs = Rt
bne Rs, Rt, label	Nhảy đến nhãn 'label' nếu Rs != Rt

System Call:

Lệnh syscall làm treo sự thực thi của chương trình và chuyển quyền điều khiển cho HĐH (được giả lập bởi MARS). Sau đó, HĐH sẽ xem giá trị thanh ghi $v0 để xác định xem chương trình muốn nó làm việc gì.

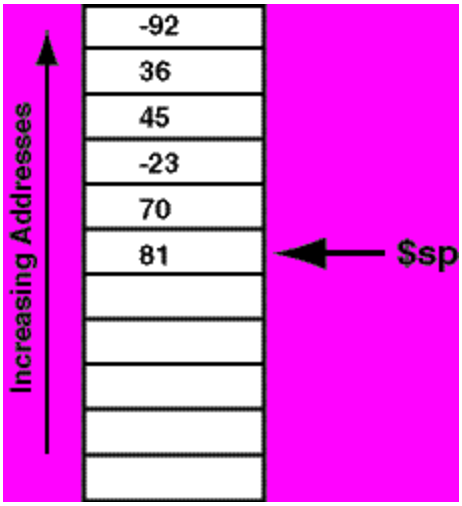
Bảng các system call

Dịch vụ	Giá trị trong $v0	Đối số	Kết quả
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (trong $v0)
read_float	6		float (trong $f0)
read_double	7		double (trong $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (trong $v0)
exit	10		
print_character	11	$a0 = char	
read_character	12		char (trong $v0)

Ví dụ:

```
        .data           # khai báo data segment
```

```
str:    .asciiz “hello world”
        .text
        .globl main
main:  # nhãn main cho vi xử lý biết nơi thực thi lệnh đầu tiên
        la $a0, str     # tải địa chỉ của nhãn str vào thanh ghi $a0
        addi $v0, $zero, 4   # đưa giá trị 4 vào thanh ghi $v0
        syscall
        addi   $v0, $zero, 10
        syscall
```



➢ Stack

Stack (ngăn xếp) là vùng nhớ đặc biệt được truy cập theo cơ chế “vào trước ra sau” (LIFO – Last In First Out), nghĩa là dữ liệu nào đưa vào sau sẽ được lấy ra trước.

Hình bên là cấu trúc stack trong bộ nhớ, mỗi phần tử có kích thước một word (32-bit).

Thanh ghi $sp đóng vai trò là con trỏ ngăn xếp (stack pointer), luôn chỉ đến đỉnh của stack. Stack phát triển theo chiều giảm của địa chỉ vùng nhớ (đỉnh của stack luôn có địa chỉ thấp).

Hai thao tác cơ bản trong stack là push (đưa một phần tử vào stack) và pop (lấy một phần tử ra khỏi stack). Cơ chế như sau:

- push: giảm $sp đi 4, lưu giá trị vào ô nhớ mà $sp chỉ đến.

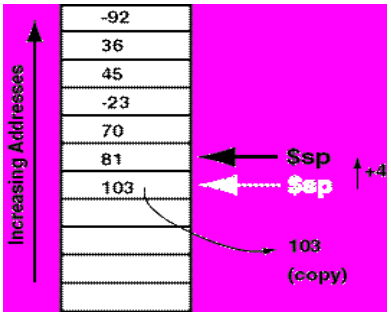
  Ví dụ: push vào stack giá trị trong $t0

  ```
  subu   $sp, $sp, 4
  sw     $t0, ($sp)
  ```

- pop: copy giá trị trong vùng nhớ được chỉ đến bởi $sp, cộng 4 vào $sp.

  Ví dụ: pop từ stack ra $t0

  ```
  lw     $t0, ($sp)
  addu   $sp, $sp, 4
  ```



➢ Thủ tục

MIPS hỗ trợ một số thanh ghi để lưu trữ các dữ liệu phục vụ cho thủ tục:

- Đối số $a0, $a1, $a2, $a3
- Kết quả trả về $v0, $v1
- Biến cục bộ $s0, $s1, ... , $s7
- Địa chỉ quay về $ra

Cấu trúc của một thủ tục:

Đầu thủ tục

```
entry_label:
addi $sp,$sp, -framesize        # khai báo kích thước cho stack
sw $ra, framesize-4($sp)        # cất địa chỉ trở về của thủ tục trong $ra vào ngăn xếp
                                (dùng khi gọi hàm lồng nhau)
Lưu tạm các thanh ghi khác (nếu cần)
```

Thân thủ tục ...

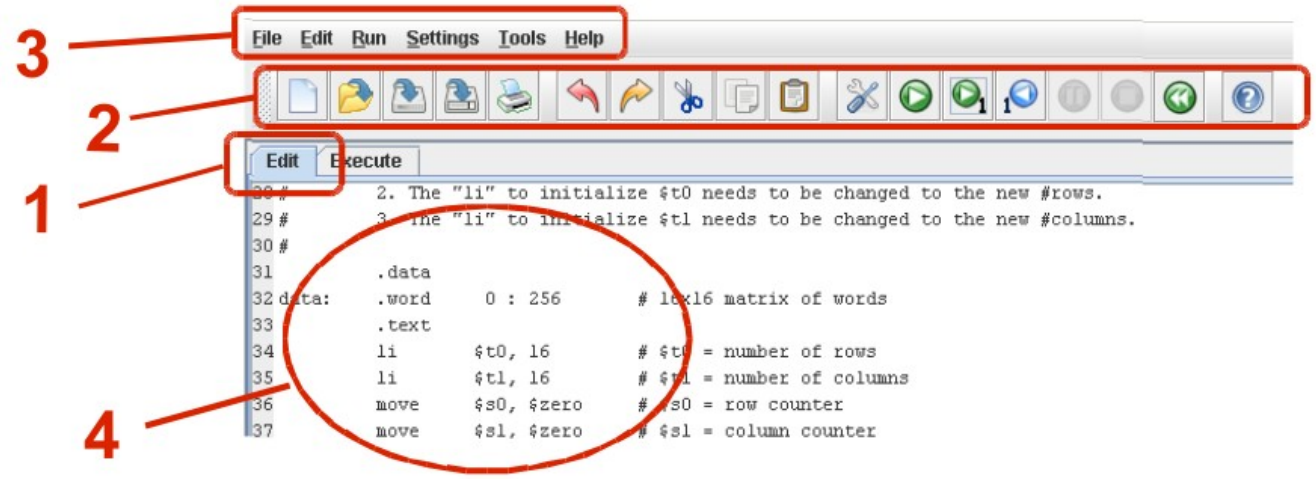
```
(có thể gọi các thủ tục khác...)
```

Cuối thủ tục

```
Phục hồi các thanh ghi khác (nếu cần)
lw $ra, framesize-4($sp)        # lấy địa chỉ trở về ra $ra
addi $sp,$sp, framesize
jr $ra
```

Gọi thủ tục: jal entry_label

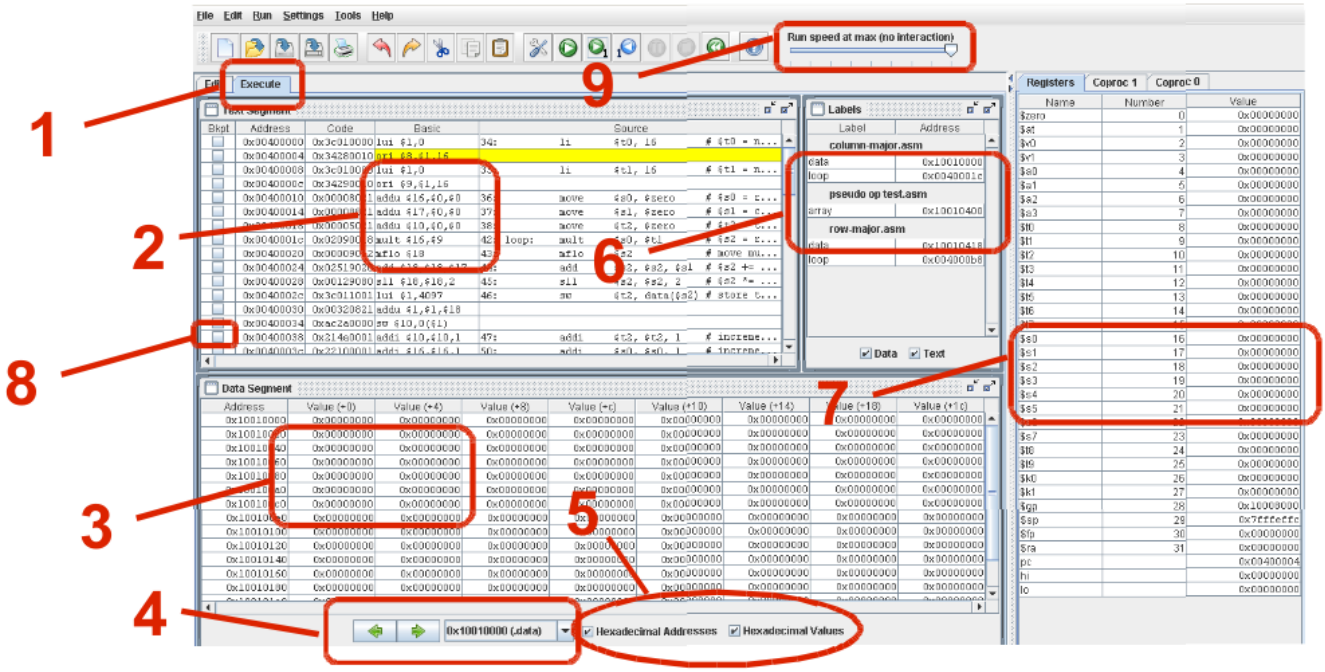
- **Giới thiệu chương trình MARS**



1. Cho biết ta đang ở chế độ soạn thảo

2,3. Thanh menu và thanh công cụ hỗ trợ các chức năng của chương trình.

4. Nơi soạn thảo chương trình hợp ngữ MIPS



1. Cho biết ta đang ở chế độ thực thi
2. Khung thực thi cho ta biết địa chỉ lệnh (Address), mã máy (Code), lệnh hợp ngữ MIPS (Basic), dòng lệnh trong file source tương ứng (Source).
3. Các giá trị trong bộ nhớ, có thể chỉnh sửa được.
4. Cho phép ta duyệt bộ nhớ (2 nút mũi tên) và đi đến các phân đoạn bộ nhớ thông dụng.

5. Bật, tắt việc xem địa chỉ và giá trị ô nhớ ở dạng thập phân (decimal) hay thập lục phân (hexa).
6. Địa chỉ của các khai báo nhãn và dữ liệu.
7. Các giá trị trong thanh ghi, có thể chỉnh sửa được.
8. Điểm đặt breakpoint dùng cho việc debug chương trình.
9. Điều chỉnh tốc độ chạy chương trình, cho phép người dùng có thể xem những gì diễn ra thay vì chương trình kết thúc ngay.

Tài liệu tham khảo

[1] http://chortle.ccsu.edu/AssemblyTutorial/index.html - Programmed Introduction to MIPS Assembly Language, *Bradley Kjell*

[2] http://www.scribd.com/doc/3577342/MIPS-Assembly-Language-Programming - MIPS Assembly Language Programming, *Robert Britton*.

[3] http://dkrizanc.web.wesleyan.edu/courses/231/07/mips-spim.pdf - MIPS Assembly Language Programming, *Daniel J. Ellard*.

[4] http://logos.cs.uic.edu/366/notes/MIPS%20Quick%20Tutorial.htm - MIPS Architecture and Assembly Language Overview

[5] http://www.cs.cornell.edu/~tomf/notes/cps104/mips.html - MIPS Examples

Bài tập

Hãy viết chương trình hợp ngữ MIPS (không dùng lệnh giả) để giải quyết các bài toán sau:

1. Nhập vào một chuỗi, xuất lại chuỗi đó ra màn hình (echo).

Ví dụ:
Nhap mot chuoi: Hello
Chuoi da nhap: Hello

2. Nhập vào một ký tự, xuất ra ký tự liền trước và liền sau.

Ví dụ:
Nhap mot ky tu: b
Ky tu lien truoc: a
Ky tu lien sau: c

3. Nhập vào một ký tự hoa, in ra ký tự thường.

Ví dụ:
Nhap mot ky tu: A
Ky tu thuong: a

4. Nhập từ bàn phím 2 số nguyên, tính tổng, hiệu, tích, thương của 2 số.

Ví dụ:
Nhap so thu nhat: 7

Nhap so thu hai: 4
Tong: 11
Hieu: 3
Tich: 28
Thuong: 1 du 3

5. Nhập vào 2 số nguyên, xuất ra số lớn hơn.
Ví dụ:
Nhap so thu nhat: 6
Nhap so thu hai: 9
So lon hon la: 9

6. Nhập một ký tự từ bàn phím. Nếu ký tự vừa nhập thuộc [0-9], [a-z], [A-Z] thì xuất ra màn hình ký tự đó và loại của ký tự đó (số, chữ thường, chữ hoa).
Ví dụ:
Nhập vào một ký tự: 5
Ký tự vừa nhập: 5 là số
Nhập vào một ký tự : f
Ký tự vừa nhập : f là chữ thường
Nhập vào một ký tự : D
Ký tự vừa nhập : D là chữ hoa

7. Nhập một mảng các số nguyên n phần tử, xuất mảng đó ra màn hình.
Ví dụ:
Nhap n: 5
[0] = 4
[1] = 2
[2] = 7
[3] = 9
[4] = 3
Mang vua nhap: 4 2 7 9 3

8. Nhập vào một số nguyên n, tính tổng từ 1 đến n.
Ví dụ:
Nhap mot so: 4
Tong tu 1 den 4 la: 10

9. Nhập vào một chuỗi. Tính chiều dài của chuỗi.
Ví dụ:
Nhap mot chuoi: HCMUS
Chieu dai cua chuoi: 5

10. Nhập vào mảng 1 chiều n số nguyên. Xuất giá trị lớn nhất và nhỏ nhất.
Ví dụ:

Nhap n: 3
[0] = 4
[1] = 2
[2] = 7
Gia tri nho nhat: 2
Gia tri lon nhat: 7

Các bài tập sau dùng stack để làm:

11. Nhập vào một chuỗi, xuất ra chuỗi ngược.
Ví dụ:
Nhap vao mot chuoi: hello
Chuoi nguoc la: olleh

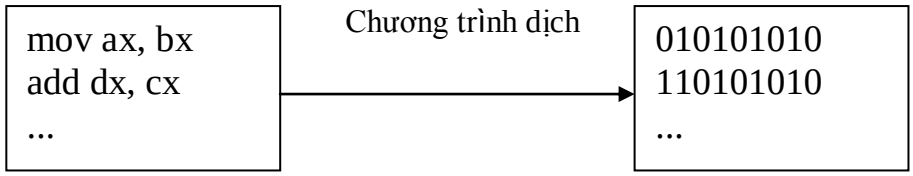
12. Viết lại bài 1 dưới dạng thủ tục.
Hướng dẫn: Hàm xuất chuỗi có dạng sau PRINT(&buf), &buf là địa chỉ của vùng nhớ chứa chuỗi.

13. Viết lại bài 4 dưới dạng thủ tục.
Hướng dẫn: Hàm tính tổng có dạng sau SUM(X, Y, Z), trong đó X, Y là 2 số nguyên, Z là tổng của 2 số.

14. Viết lại bài số 10 dưới dạng thủ tục.
Hướng dẫn: viết hàm MinMax(&X, N, Min, Max), trong đó
&X: địa chỉ bắt đầu của mảng
N: số phần tử của mảng
Min, Max: giá trị nhỏ nhất, lớn nhất

TỔNG QUAN VỀ LẬP TRÌNH HỢP NGỮ

CPU 8086 là CPU 16 bít (bus số liệu ngoại có 16 dây). Nó được dùng để chế tạo các máy vi tính đầu tiên của hảng IBM vào năm 1981. Cho đến nay, dòng CPU này không ngừng phát triển, chảy qua nhiều thế hệ như: 80X86, Pentium X



Từ gợi nhớ mã lệnh Hợp ngữ Mã máy nhị phân

1. ĐẶC TÍNH TỔNG QUÁT CỦA HỢP NGỮ :

1.1. Cấu trúc của một dòng lệnh hợp ngữ

- Trên một dòng văn bản chỉ cho phép viết duy nhất một dòng lệnh
- Một lệnh của hợp ngữ gồm các phần:

Tên: **Từ gợi nhớ mã lệnh** **Toán hạn** Chú thích

Ví dụ:

LENH: MOV DX, 3F8H ; 3F8H là địa chỉ cổng vào ra nối tiếp

- Tên

- Từ gợi nhớ mã lệnh

Một số lệnh giả thường gặp:

ORG (origin: điểm gốc): Chỉ định điểm bắt đầu của một đoạn chương trình hay dữ liệu

EQU (equate: bằng)

DEFINE (định nghĩa): cho phép đặt tên cho một dữ liệu nào đó

DS (define storage: định nghĩa vùng ô nhớ lưu trữ số liệu)

RM (reserve memory: để dành vùng ô nhớ): dùng khai báo biến

DATA, DB (define byte), **DW** (define word) cũng để khai báo biến

END: Xác định điểm chấm dứt chương trình

- Toán hạng

Chứa các toán hạng mà lệnh cần

- Chú thích

Dùng để ghi chú giải thích cho dòng lệnh

1.2 Macro

Macro là một nhóm lệnh nào đó được dung nhiều lần trong cùng một chương trình nên ta gán cho nó một tên. Mỗi khi sử dụng nhóm lệnh ấy chỉ cần gọi tên đã gán cho nhóm lệnh đó.

1.3 Chương trình con

Hợp ngữ thường cho phép dịch riêng biệt các chương trình con. Nó sẽ đánh dấu các tham khảo đến chương trình con trong chương trình chính và chương trình liên kết sẽ gán các địa chỉ của các chương trình con.

Một số hợp ngữ còn cho phép làm một thư viện chương trình con để sử dụng chung cho nhiều chương trình khác nhau.

Muốn sử dụng chương trình con thì phải dùng lệnh CALL hay lệnh JUMP để chuyển điều khiển đến chương trình con đó. Do đó phải lưu địa chỉ trở về chương trình chính ở ngăn xếp và làm chậm đi việc thực hiện chương trình chính.

1.4 Biến toàn cục (global), biến địa phương (local)

Các biến được khai báo trong chương trình chính được gọi là biến toàn cục, các biến này được dùng cho chương trình chính, chương trình con và macro.

Các biến được khai báo bên trong macro hay chương trình con gọi là biến địa phương, các biến này chỉ được dùng cho nội bộ tại macro hay chương trình con có khai báo biến đó.

1.5 Các bảng, thông báo

Đa số các chương trình hợp ngữ khi tiến hành hợp dịch có thể cung cấp các bảng và các thông báo cho người lập trình. Các bảng thông báo được cung cấp dưới dạng tập tin văn bản bao gồm:

- Bảng liệt kê chương trình hợp ngữ và mã máy tương ứng
- Bảng liệt kê các lỗi trong chương trình gốc

- Bảng các tên được dùng trong chương trình gốc

- Danh sách các tham khảo ở chương trình khác, bên ngoài chương trình (các chương trình con, các biến dùng bên ngoài)

- Danh sách các macro, chương trình con và độ dài của chúng

1.6 Hợp ngữ chéo (cross assembler)

Một hợp ngữ chạy trên một máy tính nào đó để dịch ra mã máy cho chương trình viết phục vụ một CPU khác chủng loại với CPU của máy mà hợp ngữ đó đang dùng thì được gọi là hợp ngữ chéo.

Ví dụ: Hiện nay rất khó tìm được một máy tính dùng CPU – Z80. Muốn dịch một chương trình hợp ngữ P1 thành chương trình mã máy P2 dùng cho CPU – Z80, người ta phải dùng hợp ngữ chéo chạy trên máy IBM-PC (có CPU 8086) thông dụng. Chương trình P2 không chạy được trên máy IBM-PC mà chỉ chạy được trên máy tính hoặc hệ vi xử lý do CPU – Z80 điều khiển.

2. HỢP NGỮ MASM (HỢP NGỮ CỦA CPU-8086)

MASM (MicroSoft Macro Assembler) là trình hợp ngữ do hãng phần mềm Microsoft phát hành cùng với phiên bản hệ điều hành DOS.

2.1. Cấu trúc của một hàng lệnh

Một hàng lệnh gồm có 4 vùng:

Tên	Từ gợi nhớ mã lệnh hoặc lệnh giả	Toán hạng hoặc biểu thức	Ghi chú

2.2 . Tên

Tên có thể là **nhản**, **biến** hay **ký hiệu** Tên có chiều dài tối đa là 31 ký tự và phải bắt đầu bằng một chữ cái. Các khoảng trống và các ký hiệu toán học không được dùng để đặt tên

- Nhản: Nhản dùng để đánh dấu một địa chỉ mà các lệnh như JUMP, CALL, LOOP cần đến. Nó cũng được dùng cho các lệnh giả LABEL hoặc PROC hoặc EXTRN

Ví dụ:

```
NH:   MOV      AX ; DS ; NH là nhản đánh dấu một địa chỉ ô nhớ
FOO   LABEL    Near  ; đặt tên cho địa chỉ ô nhớ sau lệnh giả này
```

CTCON PROC FAR ; địa chỉ bắt đầu của chương trình con

EXTRN NH Near ; NH là nhản ngoài chương trình gốc này

- Biến: Biến dùng làm toán hạng cho lệnh hoặc biểu thức. Biến tượng trưng cho một địa chỉ nơi đó có giá trị mà ta cần.

Ví dụ:

T DB 2 ; Biến T có giá trị là 2

- Ký hiệu: Ký hiệu là một tên được định nghĩa để thay cho một biểu thức, một từ gợi nhớ lệnh. Ký hiệu có thể dùng làm toán hạng trong biểu thức, trong lệnh hay trong lệnh giả.

Ví dụ:

FOO EQU 7h

TOTO = 0Fh

2.3 Từ gợi nhớ mã lệnh, lệnh giả

Từ gợi nhớ mã lệnh đã được trình bài ở chương I về tập lệnh của bộ xử lý 8086.

Lệnh giả được chia thành 4 nhóm sau:

2.3.1 Nhóm liên quan đến bộ nhớ

- SEGMENT và ENDS : Khai báo đoạn

Cú pháp:

<Tên đoạn> SEGMENT [align] [combine] ['class']

.....; nội dung của đoạn

<Tên đoạn> ENDS

[align] xác định nơi bắt đầu của đoạn như sau, gồm các giá trị:

Byte : Đoạn có thể bắt đầu ở địa chỉ bất kỳ

Word : Đoạn phải bắt đầu ở địa chỉ chẳn

Para : Đoạn phải bắt đầu ở địa chỉ là bội số của 16

Page : Đoạn phải bắt đầu ở địa chỉ là bội số của 256

[combine] xác định cách phân đoạn này kết hợp với các phân đoạn khác như sau:

Public : các đoạn cùng tên và cùng class được ghép nối tiếp nhau khi liên kết

Common : các đoạn cùng tên và cùng class được ghép phủ lấp lên nhau khi liên kết

AT <biểu thức> : Đoạn được đặt tại một địa chỉ là bội số của 16 và được ghi trong biểu thức

Stack : Giống như Public, tuy nhiên con trỏ ngăn xếp SP chỉ vào địa chỉ đầu tiên của ngăn xếp đầu tiên

Private : các đoạn cùng tên và cùng class không được ghép vào nhau

4. CÁC THANH GHI CỦA BỘ XỬ LÝ 8086 :

- Thanh ghi đa dụng: CPU 8086 có 4 thanh ghi đa dụng 16 bít có thể chia đôi thành 8 thanh, mỗi thanh 8 bít

+ AX (accumulator

+ BX (base register)

+ CX (count register)

+ DX (data register)

- Thanh ghi con trỏ: Dùng để thâm nhập số liệu trên ngăn xếp

+ SP (Stack pointer): Thanh ghi con trỏ ngăn xếp

+ BP (bace pionter): Thanh ghi con trỏ nền dùng để lấy số liệu từ ngăn xếp.

- Thanh ghi chỉ số:

+ SI (source index): Thanh ghi chỉ số nguồn.

+ DI (destination index): Thanh ghi chỉ số đích.

- Thanh ghi đoạn: Được dùng trong mọi tính toán địa chỉ ô nhớ. Mỗi thanh ghi đoạn xác định 64K ô nhớ trong bộ nhớ trong.

+ CS (code segment)

+ DS (data segment)

+ ES (extra segment)

+ SS (stack segment)

- Thanh ghi cờ: Phản ánh kết quả của phép tính số học và luận lý, xác định trạng thái hoạt động của CPU. Các bít của thanh ghi cờ có ý nghĩa như sau:

+ CF: thể hiện có số giữ thoát ra từ bít cao nhất của thanh ghi kết quả sau một phép tính.

+ OF: thể hiện việc tính toán vượt quá khả năng của CPU

+ SF: bằng bít cao nhất của thanh ghi kết quả sau một phép tính

+ AF: thể hiện số giữ thoát ra từ bít thứ 4 của thanh ghi kết quả.

+ PF: bằng 1 nếu 8 bít thấp của thanh ghi kết quả một phép tính có số con số 1 chẵn.

+ ZF: bằng 1 khi kết quả của 1 phép tính bằng 0

+ DF: bằng 1 thì SI và DI giảm 1 cho mỗi vòng lặp

+ IF: bằng 1 khi cho phép ngắt

+ TF: bằng 1 khi cho phép chương trình chạy từng bước để phục vụ sửa sai một chương trình

5. CÁC LỆNH THƯỜNG DÙNG CỦA BỘ XỬ LÝ 8086

Bộ xử lý 8086 có tập lệnh gồm 111 lệnh với chiều dài của lệnh từ 1 byte đến vài byte

5.1. Nhóm lệnh di chuyển số liệu

- MOV

MOV ĐÍCH, NGUỒN ;

Ví dụ:

MOV CX, BX ; chuyển nội dung thanh ghi BX vào thanh ghi CX. Nội dung thanh ghi BX vẫn giữ nguyên. Sau khi thực hiện lệnh này thì BX và CX có cùng nội dung.

- PUSH: đưa số liệu vào ngăn xếp

PUSH NGUỒN ;

Ví dụ:

PUSH AX ; Đưa nội dung thanh ghi AX vào ngăn xếp. Con trỏ ngăn xếp SP giảm đi 2 đơn vị

- POP: lấy số liệu ra từ ngăn xếp

POP ĐÍCH ;

Ví dụ:

POP BX ; Lấy nội dung của 2 byte ô nhớ mà SP trỏ tới để đưa vào BX, byte có địa chỉ thấp đưa vào BL và byte có địa chỉ cao đưa vào BH. Con trỏ ngăn xếp SP tăng lên 2 đơn vị

5.2. Nhóm lệnh chuyển địa chỉ

- **LEA** (load effective address): Nạp địa chỉ hiệu dụng

LEA reg 16, Mem 16 ;

5.3. Nhóm lệnh chuyển cờ hiệu

- **PUSHF** (push flag: lưu giữ cờ): Đây là lệnh lưu giữ thanh ghi trạng thái vào ngăn xếp.

- **POPF** (pop flag: lấy cờ ra): Đây là lệnh lấy 2 byte từ ngăn xếp đưa vào thanh ghi cờ trạng thái (thanh ghi trạng thái)

5.4. Nhóm lệnh vào ra ngoại vi

IN AL, địa chỉ cổng 8 bít ;

- **IN** (in: vào): Lấy số liệu từ ngoại vi

Ví dụ:

```
IN  AL, 3FH  ; 3FH là địa chỉ cổng 8 bít
MOV  DX, 3F8H  ; 3F8H là địa chỉ cổng 16 bít
IN  AL, DX
```

- **OUT** (out: ra): Đưa số liệu ra ngoại vi

OUT địa chỉ cổng 8 bít, AL

Ví dụ:

```
MOV  DX, địa chỉ cổng 16 bít
OUT  DX, AL
```

5.5. Nhóm lệnh chuyển điều khiển (các lệnh nhảy)

- Lệnh nhảy vô điều kiện

JMP : nhảy đến một địa chỉ khác

Cú pháp:

JMP đích ;

Tuỳ theo khoảng cách của đích đến lệnh JMP hiện tại mà ta có 3 kiểu:

+ **JMP near đích :** nhảy đến địa chỉ đích nằm trong đoạn CS hiện tại

+ **JMP short đích :** nhảy đến địa chỉ đích trong khoảng từ -128 đến +128 tính từ vị trí lệnh JMP hiện tại

+ **JMP far đích :** nhảy đến vị trí đích nằm ngoài đoạn CS hiện tại

- Lệnh nhảy có điều kiện

Cú pháp chung : lệnh đích

+ JA (jump if above: nhảy nếu lớn hơn). Nếu cờ CF=ZF=0 thì nhảy

+ JB (jump if below: nhảy nếu nhỏ hơn). Nếu cờ CF=1 thì nhảy

+ JZ (jump if zero: nhảy nếu bằng không). Nếu cờ ZF=1, nghĩa là kết quả của phép tính (hoặc so sánh) trước đó bằng 0 (hoặc 2 số được so sánh bằng nhau) thì nhảy.

+ JNZ (jump if not zero: nhảy nếu khác 0). Nếu cờ ZF=0, nghĩa là kết quả phép tính trước đó khác 0 thì nhảy.

- Lệnh so sánh CMP (compare)

Cú pháp:

CMP trái, phải ;

Nếu trái > phải thì ZF=0 và CF=0
Nếu trái = phải thì ZF=1 và CF=0
Nếu trái < phải thì ZF=0 và CF=1

Ví dụ:

```
    MOV  AX, 1000h  ; AX=1000h
    CMP  AX, 200h   ; so sánh AX với 200h
    JZ  NH              ; Nếu AX=200h thì nhảy đến nhãn NH
    MOV  CX, BX     ; Nếu AX <> 200h thì lệnh được thi hành
    …                   ; Các lệnh khác
NH :    ADD  AX, BX     ; Vị trí nhảy đến được gán nhãn NH
```

5.6. Nhóm lệnh vòng lặp

LOOP : nhảy vòng

Cú pháp:

LOOP nhãn ;

Khi lệnh LOOP được thực hiện xong, thanh ghi CX được giảm xuống 1 và nếu CX <> 0 thì nhảy tới một nhãn (trong vòng từ -128 đến +128 kể từ lệnh LOOP) .

Ví dụ:

```
        MOV  CX, 10  ; nhảy 10 vòng
NH :    ADD  AX,  CX
        MOV  BL, AL
        LOOP  NH
        …
```

+ LOOPZ (loop if zero): nhảy vòng nếu cờ ZF=1

+ LOOPNZ (loop if not zero): nhảy vòng nếu cờ ZF=0

5.7. Nhóm lệnh gọi chương trình con

- CALL : lệnh gọi chương trình con

Cú pháp:

```
CALL  nhản ;
```

- RET (return): Kết thúc chương trình con và trở về

Khi gặp lệnh RET thì 8086 lấy địa chỉ trở về ở ngăn xếp để trở về tiếp tục thi hành chương trình chính.

5.8. Nhóm lệnh tính toán số học

- ADD : cộng số nguyên

Cú pháp:

```
ADD  đích, nguồn  ;
```

Ví dụ:

```
ADD  AL, 15  ; cộng 15 vào nội dung AL, kết quả được lưu trong AL
```

- INC (Increment): tăng lên một đơn vị

Cú pháp:

```
INC  đích  ;
```

- **SUB** (subtract): trừ 2 số nguyên

Cú pháp:

```
SUB đích, nguồn  ;
```

- **DEC** (Decrment): Giảm xuống 1 đơn vị

Cú pháp:

```
DEC đích  ;
```

- MUL (mutiplication): nhân số nguyên
Cú pháp:

```
MUL nguồn  ;
```

- DIV (division): chia số nguyên
Cú pháp:

```
DIV  nguồn  ;
```

5.9. Nhóm lệnh dịch chuyển và quay

- SHL (logical shift left) : dịch trái logic
Cú pháp:

```
SHL  đích, 1        ;
```

Dịch trái toán hạng đích 1 bít

```
SHL   đích, CL      ;
```

- SHR (logical shift right): dịch phải logic
Cú pháp:

```
SHR  đích, 1        ;
```

Dịch phải toán hạng đích 1 bit

```
SHR  đích,  CL
```

Dịch phải toán hạng đích một số lần bằng nội dung của CL

- SAL (Shift arithmetic left): dịch trái số học

Lệnh này giống SHL nhưng bít thứ 0 được giữ nguyên

- SAR (Shift arithmetic right): dịch phải số học

Giống như SHR nhưng bit cao nhất của toán hạng đích được giữ nguyên

- ROL (Rotate left): quay vòng sang trái 1 bít
Cú pháp:

ROL đích, 1 ;

Quay vòng sang trái 1 bít

ROL đích, CL ;

Quay vòng sang trái một số lần bằng nội dung trong CL

- ROR (Rotate right): giống như ROL nhưng là quay sang phải

- RCL (Rotate through carry left): Giống như lệnh ROL nhưng có sự tham gia của bít số giữ CF

- RCR (Rotate through carry right): giống như RCL nhưng quay sang phải

5.10. Nhóm lệnh Logic

- AND : Giao

Cú pháp: **AND đích, nguồn ;**

Ví dụ:

```
MOV AL, 01101110B
AND  AL, 00110110B ; Kết quả là số 00100110B chứa trong AL
```

- OR : Hoặc

Cú pháp:

OR đích, nguồn ;

- XOR : Hoặc loại

Cú pháp: **XOR đích, nguồn ;**

- NOT : Đảo ngược

Cú pháp: **NOT đích ;**

- TEST : Kiểm tra

Cú pháp: **TESTđích, nguồn ;**

5.11. Nhóm lệnh xử lý chuỗi

- **MOVSB** (move string byte: Di chuyển chuỗi từng byte một)
- **MOVSW** (move string word: Di chuyển chuỗi từng từ 16 bit)
- **CMPSB** (compare string byte: So sánh chuỗi từng byte một)
- **CMPSW** (compare string word: So sánh chuỗi từng từ 16 bit)
- **SCASB** (scan string byte: Quét chuỗi từng byte một)
- **SCASW** (scan string word: Quét chuỗi từng từ 16 bit một)
- **LODSB** (load string byte: Nạp chuỗi từng byte một)
- **LODSW** (load string word: Nạp chuỗi từng từ 16 bit một)
- **STOSB** (store string byte: Lưu giữ chuỗi từng byte một)
- **STOSW** (store string word: Lưu giữ chuỗi từng từ 16 bit một)

5.12. Các lệnh khác

- **CLC** (clear carry flag): Xóa cờ CF
- **CLD** (clear diretion flag): Xóa cờ hướng
- **CLI** (clear interrup flag): Xóa cờ ngắt
- **CMC** (complement carry flag): Đảo ngược cờ CF
- **HLT** (halt): CPU ngưng hoạt động
- **INT** (interrupt): Gọi ngắt
- **IRET** (return from interrupt): Trở về chương trình chính từ chương trình phục vụ ngắt
- **LOCK**: Khóa hệ thống BUS
- **NOP** (no operation): không có tác vụ
- **WAIT**: đợi cho đến khi có xung ở chân test của CPU 8086

Ví dụ: Khai báo các đoạn tên là DSEG và CSEG: không phủ lấp lên nhau

```
DSEG        SEGMENT
            ; khai báo dữ liệu
DSEG        ENDS
CSEG        SEGMENT
```

; các lệnh trong đoạn

CSEG ENDS

- ASSUME: Chỉ định loại của đoạn

Cú pháp:
ASSUME <SegRes> : <Tên_1> , ...

Ví dụ:

ASSUME DS : DATA, CS : CODE

Chỉ định đoạn có tên DATA là đoạn dữ liệu DS và đoạn có tên CODE là đoạn lệnh CS

Ví dụ:

ASSUME NOTHING

Báo cho hợp ngữ biết là không có tên đoạn nào được cho biết loại và như vậy mỗi lần liên hệ đến một nhản (biến) thì ta phải dùng cả địa chỉ đoạn của chúng.

- COMMENT: ghi chú chương trình. Có thể viết trên nhiều dòng

Cú pháp:	
COMMENT	****<ghi chú>****

Ví dụ:

COMMENT* Đây là vùng ghi chú *

- EQU và = : gán trị cho một ký hiệu

EQU để gán trị cho một ký hiệu (hằng số) một lần khi khai báo. Muốn gán lại giá trị nhiều lần, ta dùng lệnh "="

Ví dụ:

FOO EQU 2*10 ; gán trị 20 vào FOO

FOO = 2*10 ; gán trị 20 vào FOO

- EVEN: Làm cho thanh ghi đếm chương trình PC có nội dung là một số chẳn

- EXTRN: Cho biết một tên hay một ký hiệu đã được định nghĩa bên ngoài ở một module khác được sử dụng ở một module chương trình hiện tại.

Ví dụ:

EXTRN Tagn: NEAR, So: WORD

Tagn là nhản gần (2 bytes) và So là từ biến 2 bytes nằm ngoài module hiện tại.

- Các lệnh giả khai báo biến, khởi động giá trị cho biến:

Cú pháp: ***<Tên biến> <Loại biến><giá trị>***

<Loại biến> bao gồm các lệnh giả như sau:

DB (Define Byte): Dành vùng ô nhớ trong để chứa từng byte

DW (Define Word): Dành vùng ô nhớ trong chứa từng từ (2 bytes)

DD (Define Double Word): Dành vùng ô nhớ trong chứa từ đôi (4 bytes)

DQ (Define Quad word): Dành vùng ô nhớ trong chứa từng 8 bytes

DT (Define Ten byte) : Dành vùng ô nhớ trong chứa từng 10 bytes

<Giá trị>

+ Số hay biểu thức: 01100b, 0A1D3h, 15 hay (50+10h)*9

+ Ký tự hay chuỗi: 'A', 'Hello'

+ Rỗng (không gán được giá trị): ?

Ví dụ:

```
KYTU    DB   'A' ; dành 1 byte nhớ chứa mã ASCII (65)
MOTSO   DB   14  ; dành 1 byte nhớ chứa giá trị 14
CHUOI   DB   'HELLO ASM' ; dành 9 bytes nhớ chứa chuỗi
BUF     DB   5 DUP ('B') ; dành 10 ô nhớ và không gán giá trị trước
ARRAYW  DW   1023, 01A5h, 15, 1010101010101011b
BUFFER  DB   'A', 14h, 5 DUP(?), 'HELLO', ?, 5*10, 10 DUP(2*2)
```

Ta thấy, trong hợp ngữ, kiểu của biến được hiểu đơn giản hơn thông qua kích thước ô nhớ của biến và được thể hiện trong <Loại biến>.

Các biến phải được khai báo trong đoạn dữ liệu (DS), một vài trường hợp đặt biệt vẫn được khai báo trong đoạn lệnh, nhưng phải lưu ý đến tổ chức chương trình sao cho CPU không xem các biến như là lệnh. Các biến sẽ được phân phối bộ nhớ theo thứ tự được khai báo. Biến khai báo đầu tiên sẽ có địa chỉ độ dời trong đoạn dữ liệu bắt đầu là 0h.

- RECORD và STRUC: Khai báo biến kiểu có cấu trúc (mẫu tin và cấu trúc).

Cú pháp: ***[Tên biến] <Kiểu cấu trúc> [trường: d] [, ...]***

d: số nguyên dương, xác định độ lớn của trường

<Kiểu cấu trúc>:

RECORD: d tính bằng bit. Cấu trúc dài tối đa là 16 bit

STRUC: d tính bằng byte.

Ví dụ:

```
FOO  RECORD    CAO: 7, VUA: 3, THAP: 4
DIEMSV    STRUC      TEN: 7, MON1: 1, MON2: 1, MON3: 1
```

- END: chấm dứt chương trình nguồn

- GROUP: gôm các đoạn khác nhau thành nhóm có một tên mới dùng chung.

Ví dụ:

```
CGROUP    GROUP      DATA1, DATA2, DATA3
```

DATA1, DATA2, DATA3 là tên của 3 đoạn khác nhau, bây giờ chúng được gôm chung lại thành 1 nhóm và có tên nhóm là CGROUP

- INCLUDE: xen thêm một tập tin hợp ngữ khác vào tập tin hợp ngữ hiện hành

Ví dụ:

```
INCLUDE   C:\ ASM\THEM.ASM;
```

tập tin THEM.ASM trên ổ đĩa C:\ ASM được xen vào tập tin hiện hành ngay tại vị trí của lệnh giả INCLUDE.

- LABEL: đánh dấu một địa chỉ là địa chỉ của lệnh hay số liệu kế đó.

Ví dụ:

```
NHF  LABEL   FAR ; nhản xa, đánh dấu vị trí NH của lệnh kế
NH:  MOV AX, DATA
     MOV DS, AX
CHB  LABEL   Byte ; đánh dấu vị trí CH mà ta có thể lấy từng byte
CH   DW   100 DUP (0) ; chuỗi từng từ
```

- NAME: đặt tên cho một module hợp ngữ

Ví dụ:

```
NAME    Cursor ; đặt tên cho module là cursor
```

- ORG: ấn định địa chỉ cho đoạn chương trình viết sau lệnh giả ORG

Ví dụ:

```
ORG  100h
MOV AX, code ; lệnh này sẽ đặt tại địa chỉ độ dời 100h
```

- PROC và ENDP: Khai báo chương trình con

Cú pháp: <Tên CTC> PROC[Near/Far]

Ví dụ:

```
CTCON   PROC          Near
        MOV AX, code  ; bắt đầu chương trình con
        ...
        RET
CTCON   ENDP
```

- PUBLIC: dùng khai báo các tên trong module hiện hành mà các module khác có thể sử dụng

Ví dụ:

```
PUBLIC   FOO, NH, TOTO
```

6. Nhóm lệnh giả về dịch (compile) có điều kiện

Các lệnh giả về dịch có điều kiện nhằm báo cho hợp ngữ tiến hành dịch một nhóm lệnh nếu một điều kiện được thỏa mãn (đúng). Ngược lại sẽ không dịch khi điều kiện không thỏa mãn (sai)

IF xxxx [đối số] có thể có những hình thức như sau:

Cú pháp ***IF xxxx [đối số] ;*** đặt điều kiện dịch chương trình

... ; Nhóm lệnh
[ELSE]
... ; Nhóm lệnh

IFE <biểu thức>: Nếu <biểu thức> = 0 thì đoạn chương trình sau IFE được dịch. Nếu <biểu thức> <>0 thì đoạn chương trình sau ELSE được dịch nếu có lệnh giả ELSE.

IF1: Nếu một hợp ngữ đang dịch lần 1 thì đoạn chương trình sau IF1 được dịch.

IF2: Nếu một hợp ngữ đang dịch lần 2 thì đoạn chương trình sau IF2 được dịch.

IFDEF <ký hiệu>: Nếu ký hiệu đã được định nghĩa thì dịch đoạn chương trình sau IFDEF.

IFNDEF <ký hiệu>: Nếu ký hiệu không được định nghĩa thì dịch đoạn chương trình sau IFNDEF.

IFB <đối số>: Nếu đối số là khoảng trống hoặc không có đối số thì dịch đoạn chương trình sau IFB.

IFNB <đối số>: Nếu có đối số thì dịch đoạn chương trình sau IFNB

IFIDN <đối số 1>, <đối số 2>: Nếu <đối số 1> = <đối số 2> thì dịch đoạn chương trình sau IFIDN.

IFDIF <đối số 1>, <đối số 2>: Nếu <đối số 1> <> <đối số 2> thì dịch đoạn chương trình sau IFDIF.

7. Nhóm lệnh giả về Macro

Lệnh giả Macro giúp ta viết một đoạn chương trình mà ta có thể xen vào ở bất cứ nơi nào trong chương trình hợp ngữ bằng cách viết ra tên của Macro mà ta muốn gọi. Ta định nghĩa một Macro như sau:

```
<Tên>       MACRO    [tham số]
;           (đoạn chương trình)
ENDM ;      (chấm dứt Macro)
```

Ví dụ:

```
Gan   MACRO    X, Y, Z
      MOV      AX, X
      MOV      BX, Y
      MOV      CX, Z
      ENDM
```

Trong một chương trình nào đó ta có thể gọi MACRO Gen như sau:

```
      Gen   10, 20, 30  ; X=10, Y=20, Z=30
```

Như vậy một đoạn chương trình sau đây sẽ được xen vào:

```
      MOV AX, 10
      MOV BX, 20
      MOV CX, 30
```

Ngoài ra, bên trong Macro ta có thể dùng lệnh giả LOCAL, EXITM

EXITM dùng để thoát khỏi Macro trước khi có lệnh chấm dứt Macro là ENDM

LOCAL dùng để định nghĩa các nhãn địa phương trong các Macro mà ta muốn gọi nhiều lần

Ví dụ:

```
Wait  MACRO    count
      MOV CX, count
Next: LOOP             next
      ENDM
```

Macro này chỉ được phép gọi một lần vì nhãn Next chỉ có thể xuất hiện trong chương trình một lần

Nếu muốn gọi Macro Wait nhiều lần thì ta phải thêm lệnh giả LOCAL như sau:

```
Wait  MACRO    count
```

```
        LOCAL     Next ;  Next là nhãn địa phương
        MOV CX, count
Next    LOOP              Next
        ENDM
```

8. Nhóm lệnh giả về liệt kê

Nhóm lệnh này dùng để điều khiển in ấn chương trình theo một định dạng văn bản ở đầu mỗi trang, như: số trang, số cột, tựa đề chương trình.

- PAGE số hàng, số cột

Ví dụ:

```
PAGE              24, 15 ; mỗi trang liệt kê có 24 hàng và 15 cột
```

- TITLE (đề tựa) : cho phép đặt đề tựa chương trình

Ví dụ:

```
TITLE  chuong trinh hop ngu
```

- SUBTTL (đề tựa con): cho phải liệt kê một đề tựa con ở mỗi đầu trang

- % OUT <văn bản> : Văn bản được liệt ke khi hợp ngữ dịch chương trình. Bao gồm các chỉ thị loại văn bản như sau:

+ LIST : cho liệt kê tất cả các hàng lệnh với mã của nó (điều kiện mặc nhiên)

+ XLIST : không cho liệt kê

+ XALL : liệt kê mã do MACRO tạo nên

+ LALL : liệt kê toàn bộ MACRO

+ SALL : không liệt kê MACRO

+ CREF : liệt kê bản đối chiếu chéo

+ XCREF : không liệt kê bản đối chiếu chéo

9. Toán hạng và toán tử

9.1 Toán hạng

Bao gồm toán hạng tức thì, thanh ghi và ô nhớ

- Toán hạng tức thì: có thể là một số hoặc một ký hiệu đã được gán một số bằng các lệnh giả EQU và dấu "=". Toán hạng tức thì có thể được sử dụng theo 4 dạng: Thập phân, Nhị phân, Thập lục phân và Ký tự

Ví dụ:

```
SL      EQU  15 ; thập phân
MOV AL, SL
ADD  AL, 20h  ; thập lục phân
MOV DL, 'A'  ;  ký tự
ADD  DL, 01010101b   ;  Nhị phân
```

- Toán hạng thanh ghi: dùng các ký hiệu tên thanh ghi như: AX, BX, CX, DX, AL, AH, SI, ...

- Toán hạng ô nhớ: tượng trưng một địa chỉ của ô nhớ. Nó luôn luôn là độ dời đối với một địa chỉ bắt đầu của đoạn tương ứng.

Ví dụ:

```
FOO  DW    0FEFEh
MOV AX, FOO  ; FOO là địa chỉ của một số liệu
MOV FOO, AX
```

Các cách khác viết toán hạng ô nhớ:

FOO+5, FOO[5], 5[FOO] là các cách viết tương đương nhau và đều chỉ đến địa chỉ FOO cộng 5

5[BX][SI], [BX +5] [SI] , [BX] 5 [SI] là tương đương, với [BX+SI+5] trong định vị nền + chỉ số

9.2 Toán tử

Có 4 loại toán tử: thuộc tính, số học, quan hệ và logic

- Toán tử thuộc tính (attribute):

+ PTR (pointer): dùng thay đổi kiểu của các địa chỉ, số liệu

Ví dụ:

```
CALL          Word          PTR [BX + SI]
```

[BX + SI] trỏ tới 1 byte trong ô nhớ, nhưng ta muốn lấy 2 byte địa chỉ bắt đầu của chương trình con nên ta dùng từ Word PTR

+ DẤU HAI CHẤM (:) dùng thay đổi đoạn mặc nhiên

Ví dụ:

MOV AX, ES: [BX + SI]

Mặc nhiên thì [BX + SI] trỏ tới số liệu trong đoạn DS nhưng ta muốn lấy trong đoạn ES nên viết ES: [BX + SI]

+ SHORT: dùng thay đổi kiểu mặc nhiên là Near của lệnh JMP và báo cho hợp ngữ biết chỉ nhảy trong vòng –128 đến +127 so với vị trí lệnh JMP

+ THIS: tạo một toán hạng có giá trị tùy thuộc vào tham số của THIS

Ví dụ:

NH EQU THIS BYTE tương đương NH LABEL BYTE

SCH = THIS NEAR tương đương SCH LABEL NEAR

+ SEG: cho ta giá trị địa chỉ đoạn của một nhản hay một biến

Ví dụ:

MOV AX, SEG TENB ; đưa địa chỉ đoạn của TENB vào AX

+ OFFSET: Tương tự SEG nhưng cho địa chỉ độ dời

Ví dụ:

MOV BX, OFFSET FOO ; độ dời của biến FOO

+ TYPE: Xác định kiểu của Tên

TYPE <biến>: cho số byte ô nhớ mà loại biến đó được khai báo (byte= 1, word= 2, dword= 4, ...)

TYPE <nhản>: 0FFFFh nếu nhản Near và 0FFFEh nếu nhản Far

Ví dụ:

MOV AX, TYPE NH ; với NH là một nhản Far -> AX=0FFFEh

+ LENGTH: cho số phần tử của một biến

Ví dụ:

FOO DW 100 DUP(?)

MOV CX, LENGTH FOO ; CX chứa 100

+ SIZE: cho số byte mà một biến chiếm

Ví dụ:

FOO DW 100 DUP(?)

MOV BX, SIZE FOO ; BX chứa 200

- Toán tử số học

Ngoài các toán tử số học thông dụng như +, -, *, / ta còn có các toán tử:

+ MOD: chia lấy số dư

Ví dụ:

MOV AX, 100 MOD 17 ; AX chứa 15

+ SHR: dịch sang trái

Ví dụ:

MOV AX, 1100000b SHR 5 ; AX chứa 11B

+ SHL : dịch sang phải

+ "-" : dấu trừ đứng trước một số để chỉ đó là số âm: -5, -20

- Toán tử quan hệ

Toán tử quan hệ dùng so sánh 2 toán hạng và thường được dùng trong việc dịch chương trình có điều kiện

Cú pháp:
<toán hạng 1> EQ <toán hạng 2>

Nếu toán hạng 1 bằng toán hạng 2 thì biểu thức đúng (true)

Một số toán tử quan hệ khác:

+ NE (not equal): trả về TRUE nếu 2 toán hạng không bằng nhau

+ LT (less than): trả về TRUE nếu toán hạng bên trái nhỏ hơn toán hạng bên phải

+ GT (greater than): trả về TRUE nếu toán hạng bên trái lớn hơn toán hạng bên phải.

+ LE (less than or equal): trả về TRUE nếu toán hạng bên trái nhỏ hơn hoặc bằng toán hạng bên phải.

+ GE (greater than or equal): trả về TRUE nếu toán hạng bên trái lớn hơn hoặc bằng toán hạng bên phải.

- Toán tử logic

Toán tử logic so sánh hai toán hạng từng bít một

+ NOT: trả về TRUE nếu hai toán hạng bên trái và phải khác nhau

+ AND: trả về TRUE nếu cả hai toán hạng logic đều là TRUE

+ OR: trả về TRUE nếu một trong hai toán hạng là TRUE

+ XOR: trả về TRUE nếu hai toán hạng khác nhau

10. LẬP TRÌNH DÙNG HỢP NGỮ MASM

Trong lập trình dùng hợp ngữ MASM, ta thường sử dụng các ngắt có sẵn trong Rom-Bios (Basic input output system: hệ thống vào ra cơ bản) hoặc trong DOS.

Khi sử dụng các ngắt ta cần lưu ý một số yếu tố:

+ Chức năng của ngắt/ hàm

+ Tên ngắt/ hàm: Dùng số hex từ 0h đến 0FFh

+ Tham số vào/ ra : Dùng thanh ghi hay bộ nhớ

+ Lệnh gọi ngắt/ hàm:

```
MOV  AH,  <tên hàm>
INT   <tên ngắt>
```

Ví dụ:

Hàm 02h của ngắt 21h cho phép in ký tự trong thanh ghi DL ra màn hình. Như vậy, khi muốn hiển thị lên màn hình ký tự 'B' ta viết như sau:

```
MOV  AH, 02h      ; chỉ định hàm được sử dụng
MOV  DL, 'B'      ; hay MOV  DL, 42h
INT   21h         ; Gọi ngắt thực hiện
```

10.1 Các ngắt thường dùng trong DOS

Các ngắt của Dos được đánh số từ 20H đến 27H. Ngắt quan trọng nhất là ngắt 21H gồm nhiều chức năng của Dos

- **INT 20h**: Chấm dứt chương trình và trở về Dos từ một chương trình có đuôi COM

- **INT 23h**: Chận tổ hợp phím Ctrl-Break

Ta có thể chấm dứt sớm một chương trình bằng cách ấn tổ hợp phím Ctrl+Break

- **INT 21h**: Ngắt 21h có nhiều hàm chức năng của Dos như sau:

+ Hàm 1: Đợi đọc một ký tự từ bàn phím và cho in ra màn hình. Khi một phím được ấn thì ký tự tương ứng với phím đó được lưu trong thanh ghi AL. Nếu phím được ấn là một phím đặt biệt thì AL=0. Tổ hợp phím Ctrl+Break kết thúc công việc của hàm này.

+ **Hàm 2**: Đưa một ký tự ra màn hình. Ký tự cần đưa ra màn hình được chứa trong DL

+ **Hàm 3**: Đọc vào từ cổng nối tiếp. Ký tự được chứa trong AL

+ **Hàm 4**: Đưa ký tự chứa trong DL ra cổng nối tiếp

+ **Hàm 5**: Đưa ký tự chứa trong DL ra cổng máy in

+ **Hàm 6**: Có thể thực hiện cả vào lẫn ra

Muốn nhập ta cho DL=0FFh. Sau khi thực hiện hàm 6h nếu cờ ZF=0 thì AL chứa ký tự mới nhập vào, nếu cờ FZ=1 thì không có ký tự nào được nhập vào.

Nếu DL chứa một số khác 0FFh thì ký tự chứa trong DL được đưa ra màn hình.

+ **Hàm 7**: Giống hàm 1h chỉ khác ở chỗ nó không in ra màn hình. Ctrl+Break không kết thúc công việc.

+ **Hàm 8**: Giống hàm 1h, nhưng không in ký tự vừa đọc ra màn hình.

+ **Hàm 9**: Đưa ra màn hình một xâu ký tự được đặt trong DS:DX (nghĩa là chuỗi ký tự đó nằm trong đoạn DS và có độ dời nằm trong thanh ghi DX), và xâu ký tự phải chấm dứt bằng dấu $.

Ví dụ:

In chuỗi "Toi yeu Viet Nam " ra màn hình

```
chuoi  DB  'Toi yeu Viet Nam'  ; Khai báo biến chuỗi trong đoạn dữ liệu DS
MOV  AH, 09h                   ; Gọi hàm 09h
LEA   DX, chuoi                ; hay MOV  DX, OFFSET chuoi
INT   21h                      ; Thực hiện lệnh
```

+ **Hàm 0Ah**: Đọc vào vùng đệm bàn phím

Trước khi dùng hàm này ta phải tạo ra một vùng đệm trong bộ nhớ. Byte đầu tiên của vùng này xác định số ký tự tối đa có thể đưa vào. Byte thứ hai chính DOS cho biết số ký tự đã thực sự đua vào. Địa chỉ vùng đệm được đặt trong DS:DX

Cấu trúc vùng đệm bàn phím:

MAX:1byte	LEN: 1 byte	BUFF: 1 byte		...			

MAX: Biến chứa số ký tự tối đa có thể nhận, do người dùng qui định

LEN: Biến chứa số ký tự nhận được, do hàm trả về

BUFF: Biến chứa mã ASCII của từng ký tự trong chuỗi nhận được. Số byte của BUFF phải bằng hay lớn hơn giá trị MAX.

Ví dụ:

Nhận chuỗi tối đa 30 ký tự từ bàn phím

Khai báo vùng đệm trong đoạn dữ liệu

```
MAX  DB  31       ; tạo biến MAX
LEN    DB  ?              ; tạo biến LEN
BUFF  DB   31  DUP(?) ; vùng nhớ lưu trữ ký tự
```

Viết chương trình trong đoạn lệnh

```
MOV   AH, 0Ah
LEA    DX, max   ; hay MOV  DX, OFFSET  max
INT     21h
```

+ **Hàm 0Bh**: Cho trạng thái bàn phím. Sau khi thực hiện hàm nếu AL=0 thì không có ký tự nào đang đợi. Nếu AL chứa một số khác không thì có một ký tự đang đợi CPU đọc vào.

Tiếp theo hàm 0Bh là một nhóm hàm thực hiện các thao tác trên FCB (file control block) đã lỗi thời nên ta sẽ không nói đến. Thay vào đó ta dùng nhóm hàm tương đương dùng thẻ tập tin (file handle)

+ **Hàm 25h**: Đổi vecto ngắt. DS:DX chứa giá trị mới của địa chỉ vecto ngắt và trong AL là số của ngắt

+ **Hàm 30h**: Lấy ấn bản (version) của Dos. Sau khi thực hiện, AL chứa phần chính và AH chứa phần phụ của ấn bản

+ **Hàm 31h**: Kết thúc và ở lại thường trú. Hàm này giống ngắt 27h. Ta đặt trong thanh ghi DX số paragraph (16 bytes) của phần tập tin ta muốn giữ thường trú

+ **Hàm 35h**: Lấy giá trị của vecto ngắt, ta để trong AL số của ngắt. Sau khi thực hiện hàm này thì ES:BX chứa địa chỉ của vecto ngắt

+ **Hàm 3Ch**: Tạo tập tin mới. Cặp thanh ghi DS:DX trỏ tới tên của tập tin mới. Cuối chuỗi ta phải có số 0. CX chứa thuộc tính của tập tin (0: bình thường; 1: chỉ đọc; 2: ẩn; 4: tập tin hệ thống). Sau khi thực hiện hàm thì thẻ tập tin được lưu giữ trong AX. Ta phải giữ nó lại trong bộ nhớ vì mỗi lần truy xuất tập tin vừa tạo đó ta cần dùng thẻ tập tin.

+ **Hàm 3Dh**: Mở một tập tin đã có trên đĩa. Cặp thanh ghi DS:DX trỏ tới chuỗi tên tập tin (có giá trị 0 ở cuối chuỗi). Thanh ghi AL chứa kiểu truy cập tập tin (kiểu 0: chỉ đọc; 1: chỉ ghi; 2: đọc và ghi). Thẻ tập tin nằm trong AX.

Khai báo tên tập tin và thẻ file:

```
Tenfile     DB  'C:\dulieu\data.txt',0
Thefile     DW  ?
```

Chuỗi tên file có chứa cả tên đường dẫn của file. Nếu không có tên đường dẫn thì sẽ truy xuất trong thư mục hiện hành.

Thẻ file là thành phần quan trọng trong việc truy xuất file nên sau khi mở file hay tạo file phải lưu trữ lại để sử dụng trong việc đọc ghi file. Mỗi file phải có một thẻ file, biến thẻ file phải là 2 byte.

```
MOV  thefile, AX  ; lưu trữ thẻ file vào biến
MOV  BX, thefile  ; Đưa thẻ file vào BX để truy xuất file
```

+ **Hàm 3Eh**: Đóng tập tin. Thẻ tập tin cần đóng được chứa trong BX

+ **Hàm 3Fh**: Đọc từ tập tin hoặc thiết bị. Cặp thanh ghi DS:DX trỏ đến vùng bộ nhớ mà dữ liệu sẽ chuyển tới, BX chứa thẻ tập tin, trước khi đọc phải mở tập tin và CX

chứa số byte cần đọc vào. Sau khi thực hiện hàm thì AX chứa số byte vừa đọc được.

Ta cũng có thể dùng hàm này để đọc từ ngoại vi, vì Dos cũng gán thẻ cho ngoại vi.

+ **Hàm 40h**: Ghi lên tập tin hoặc thiết bị

DS:DX trỏ đến vùng chứa dữ liệu trong bộ nhớ

BX: chứa thẻ tập tin

CX: chứa số byte cần ghi lên đĩa

Sau khi ghi xong, thanh ghi AX chứa số byte ghi được, có thể dùng hàm này để gởi dữ liệu ra ngoại vi

+ **Hàm 41h**: Xóa tập tin trên đĩa

DS:DX trỏ đến tập tin muốn xóa, ta không thể xóa tập tin đang mở hoặc có thuộc tính chỉ đọc.

+ **Hàm 43h**: Đọc hoặc thay đổi thuộc tính tập tin

Đọc thuộc tính:

Đặt AL=0

DS:DX trỏ đến tên tập tin

Sau khi thực hiện hàm thì thuộc tính nằm trong CX

Tay đổi thuộc tính:

Đặt AL=1

Cho CX= thuộc tính mới

DS:DX trỏ đến tên tập tin mới

+ **Hàm 4CH**: Kết thúc chương trình và trở về DOS (cho cả hai loại tập tin có đuôi COM và EXE)

+ **Hàm 56H**: Đổi tên tập tin

DS:DX chứa địa chỉ chuỗi tên file cũ

ES:DI chứa địa chỉ chuỗi tên file mới

10.2 Cấu trúc của chương trình hợp ngữ dùng MASM

Một chương trình hợp ngữ có thể dùng nhiều module. Các module có thể viết riêng lẽ bằng một chương trình xử lý văn bản và dịch riêng lẽ bằng MASM để cho các chương trình đích. Các chương trình đích sẽ được chương trình liên kết (link) nối lại với nhau thành một chương trình chạy được có đuôi EXE.

Mỗi module có thể viết theo một cấu trúc điển hình như sau:

```
PAGE    60, 132
TITLE           ; chọn tiền đề cho module
        ; các phát biểu EXTRN hay PUBLIC (nếu có)
MCR     MACRO   ; (nếu có)
        ; (viết macro)
```

```
                ENDM
STACK       SEGMENT  PARA  STACK  'STACK'
                DB  64  DUP(?)
STACK       ENDS
DSEG        SEGMENT  PARA  PUBLIC  'DATA'
            ; khai báo dữ liệu
DSEG        ENDS
CSEG        SEGMENT  PARA  PUBLIC  'CODE'
                ASSUME  CS:CSEG, DS:DSEG, SS:STACK
BATDAU:     MOV  AX, DSEG
            MOV  DS, AX       ; khởi động DS
                :
            ; (chương trình chính)
                :
CTC         PROC  NEAR  ; (nếu có)
            ; (chương trình con)
CTC         ENDP
CSEG        ENDS
        END BATDAU  ; chấm dứt chương trình và chọn địa chỉ bắt đầu
```

10.3 Tập tin đuôi COM và tập tin đuôi .EXE

Hợp ngữ MASM cho phép tại hai loại tập tin có đuôi COM dùng cho các chương trình nhỏ còn tập tin có đuôi EXE dùng để xây dựng các chương trình lớn.

10.3.1 Đặc điểm tập tin có đuôi .COM

- Chỉ dùng một đoạn duy nhất, các thanh ghi CS, DS, ES và SS đều có chung một trị số.

- Kích thước chương trình tối đa là 64K.

- Tập tin COM được nạp vào bộ nhớ trong và thực hiện nhanh hơn tập tin EXE.

Khi DOS thực hiện tập tin COM, nó tạo một vùng ô nhớ từ độ dời 0 đến 0FFh để chứa thông tin cần thiết cho DOS. Vùng này gọi là vung PSP (program segment prefit) và tất cả các thanh ghi đoạn đều phải chỉ tới vùng PSP này. Vì thế địa chỉ bắt đầu của tập tin COM phải có địa chỉ độ dời là 100h.

Cấu trúc điển hình của tập tin COM như sau:

```
; Khai báo hằng số (nếu có)
MNAME   MACRO    ;(nếu có)
        ; Viết Macro
        ENDM
CSEG    SEGMENT
        ASSUME  CS:CSEG, DS:CSEG
        ORG  100H
BD:  MOV  AX, CSEG
        MOV  DS, AX      ; khởi động DS
        ...
        ; chương trình chính
        ...
        INT   20H        ; thoát
; Khai báo dữ liệu (nếu có)
;
THUTUC  PROC             ;(nếu có)
        ; viết thủ tục(nếu có)
        RET
THUTUC  ENDP
CSEG    ENDS
        END  BD
```

Ví dụ:

Viết chương trình dùng hàm 9h ngắt 21 để in ra màn hình chuỗi "Toi yeu Viet Nam" theo dạng COM

```
CSEG    SEGMENT
        ASSUME   CS:CSEG, DS:CSEG
        ORG   100H; Đặt địa chỉ đầu chương trình là 100h
BEGIN:  MOV   AX, CSEG
        MOV   DS, AX      ; khởi động DS
        MOV   AH, 9               ; hàm in ra màn hình
        MOV   DX,  OFFSET MESS
        INT   21H
        INT   20H         ; thoát
MESS    DB    'Toi yeu Viet Nam$'      ; Khai báo dữ liệu
CSEG    ENDS
        END  BEGIN
```

10.3.2 Tập tin có đuôi EXE có các đặc điểm

- Chương trình lớn và nằm ở nhiều đoạn khác nhau
- Có thể gọi các chương trình con ở dạng FAR
- Kích thước tập tin tùy ý
- Có header ở đầu tập tin để chứa các thông tin điều khiển cần thiết
- Thi hành chậm hơn tập tin dạng COM

Cho tập tin dạng EXE ta không dùng lệnh ORG 100H ở đầu chương trình.

Cấu trúc điển hình của tập tin EXE như sau:

```
; Khai báo hằng số
MNAME    MACRO     ; nếu có
```

```
                ; Viết các Macro
                ENDM
DSEG    SEGMENT
                ; Khai báo dữ liệu
DSEG    ENDS
CSEG    SEGMENT
                ASSUME  CS:CSEG, DS:DSEG
BD:             MOV     AX, DSEG
                MOV     DS, AX      ; Khởi động DS
                ...
                ; Chương trình chính
                ...
                MOV  AH, 4CH    ; hay MOV  AX, 4C00H
                INT   21H       ; Thoát
THUTUC  PROC
                ; Viết thủ tục (nếu có)
                RET
THUTUC  ENDP
CSEG    ENDS
                END  BD
```

Ví dụ:

Viết chương trình dùng hàm 9h ngắt 21h để in ra màn hình chuỗi "Toi yeu Viet Nam" theo dạng EXE

```
DSEG    SEGMENT
                Mess  DB   'Toi yeu Viet Nam$' ; khai báo dữ liệu
DSEG    ENDS
CSEG    SEGMENT
                ASSUME  CS:CSEG, DS:DSEG
```

```
BD:     MOV AX, DSEG
        MOV DS, AX      ; khởi động DS
        MOV AH, 9H      ; Hàm in ra màn hình
        LEA DX, Mess    ; DX chứa địa chỉ chuỗi
        INT 21H
        MOV AH, 4CH     ; hay MOV AX, 4C00H
        INT 21H         ; thoát
CSEG    ENDS
```

CÂU HỎI ÔN TẬP VÀ BÀI TẬP CHƯƠNG VI

*****

Sinh viên hãy dùng hợp ngữ viết các chương trình với các yêu cầu sau:

Bài 1

In một kí tự bất kì ra màn hình.

Bài 2

In ra màn hình một chuỗi kí tự.

Bài 3

Nhận một kí tự từ bàn phím.

Bài 4

Nhận một chuỗi kí tự từ bàn phím.

Bài 5

Viết một chương trình nhập từ bàn phím tên của một người, sau đó in ra màn hình chuỗi có dạng như sau:

Xin chào <tên_đã_nhập>

Bài 6

Viết một chương trình nhận một kí tự. Nếu kí tự hoa in ra màn hình "Kí tự hoa", ngược lại thì in "Kí tự thường".

Bài 7

Dùng Macro để in ra một chuỗi kí tự.

Bài 8

Hãy viết một chương trình phân biệt được ba loại kí tự nhập từ bàn phím: “Kí tự hoa”, “Kí tự thường”, “Kí tự khác”.

Bài 9

In ra các kí tự từ A đến Z.

Bài 10

Viết một chương trình nhập từ bàn phím một kí tự thường. Sau đó in ra màn hình lần lượt các kí tự từ kí tự nhận được đến ‘**z**’ sao cho giữa các kí tự có một khoảng trống.

Bài 11

Nhập vào một kí tự thường sau đó đổi hoa kí tự vừa nhập và in ra màn hình.

PHỤC LỤC : PHẦN MỀM MÔ PHỎNG 8086 MICROPROCESSOR EMULATOR (EMU8086)

□ MỤC ĐÍCH

Giúp sinh viên khảo sát các vấn đề sau:

□ Sử dụng phần mềm Emu8086 để mô phỏng hoạt động của vi xử lý 8086.

□ THIẾT BỊ SỬ DỤNG

□ Máy vi tính.

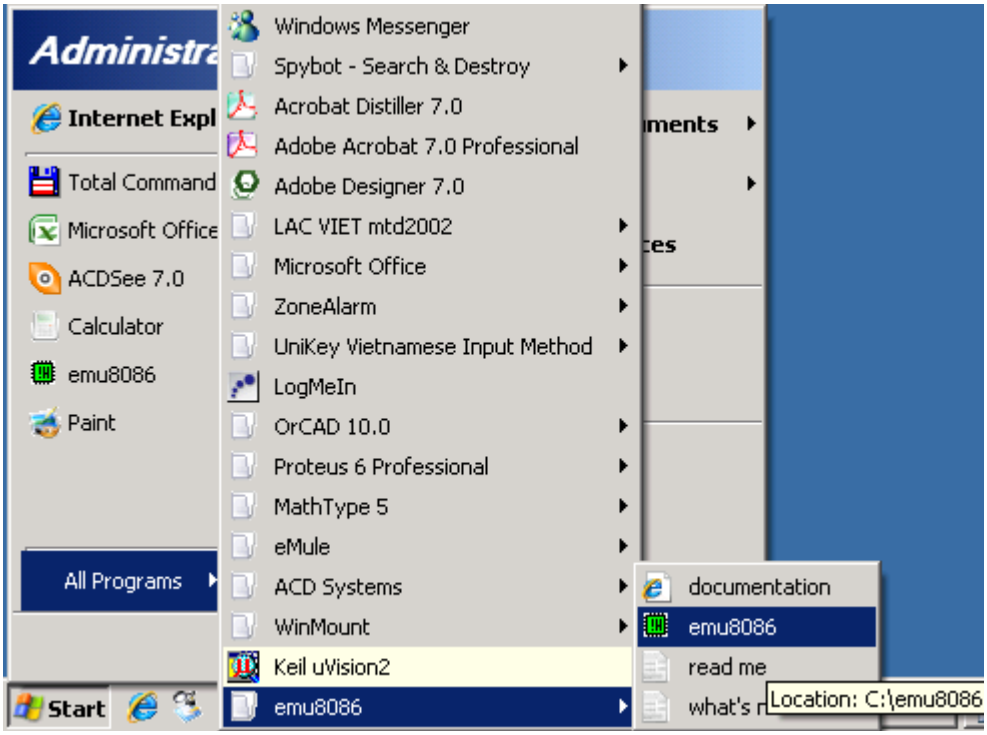
□ Phần mềm Emu8086

1. Giới thiệu.

Phần mềm Emu8086 là phần mềm cho phép mô phỏng hoạt động của vi xử lý 8086 bao gồm các câu lệnh cơ bản của 8086, xử lý ngắt mềm, giao tiếp với
thiết bị ngoại vi, …

□ **Khởi động chương trình**

□ Start > All Program > emu8086 > emu8086



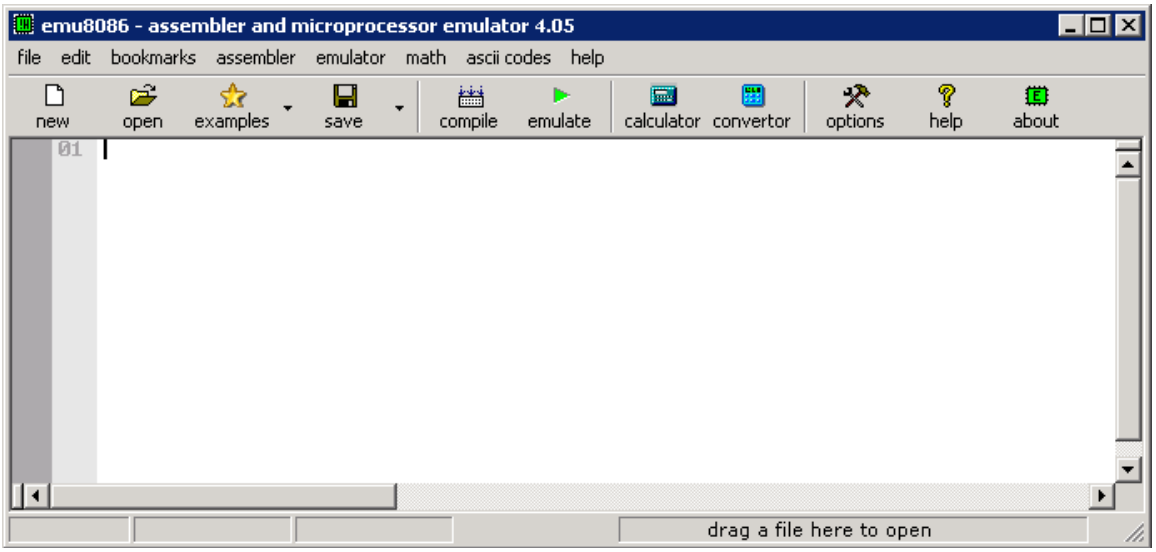
□ Cửa sổ chương trình sau khi khởi động:

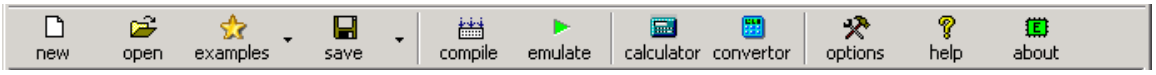
□ **Sử dụng thanh công cụ chuẩn:**

Các thao tác trên thanh công cụ chuẩn cũng có thể thực hiện thông qua menu File và menu Emulator.

□ **Tạo và thực thi chương trình:**

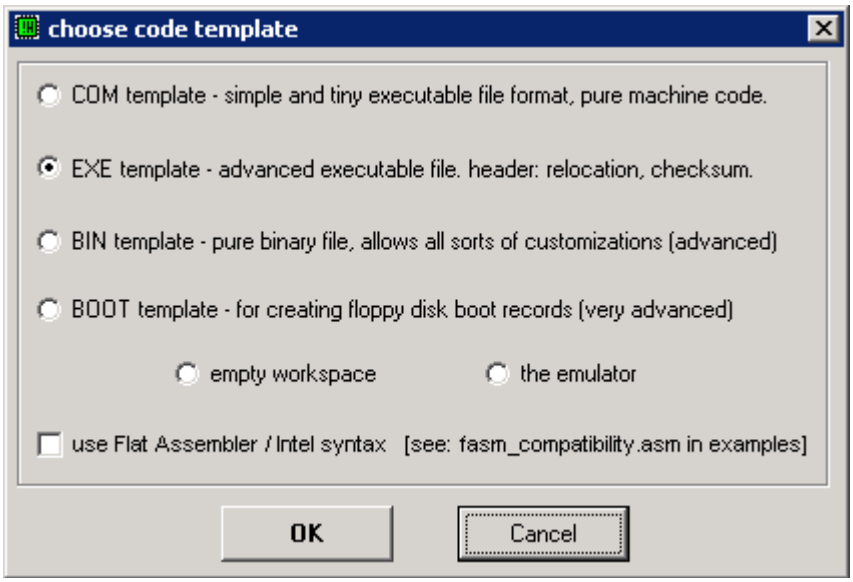
- Nhấn **New** trên thanh công cụ sẽ xuất hiện cửa sổ chọn loại file:
Compile: biên
dịch file
Open: mở file có
sẵn
New: tạo file mới Save: lưu trữ file
Emulate: biên dịch
và thực hiện mô
phỏng
Vùng không gian soạn thảo
Thanh công cụ chuẩn chương
trình





Sử dụng thanh công cụ chuẩn:
Các thao tác trên thanh công cụ chuẩn cũng có thể thực hiện thông qua menu File và menu Emulator.

- **Tạo và thực thi chương trình:**
  - Nhấn **New** trên thanh công cụ sẽ xuất hiện cửa sổ chọn loại file:



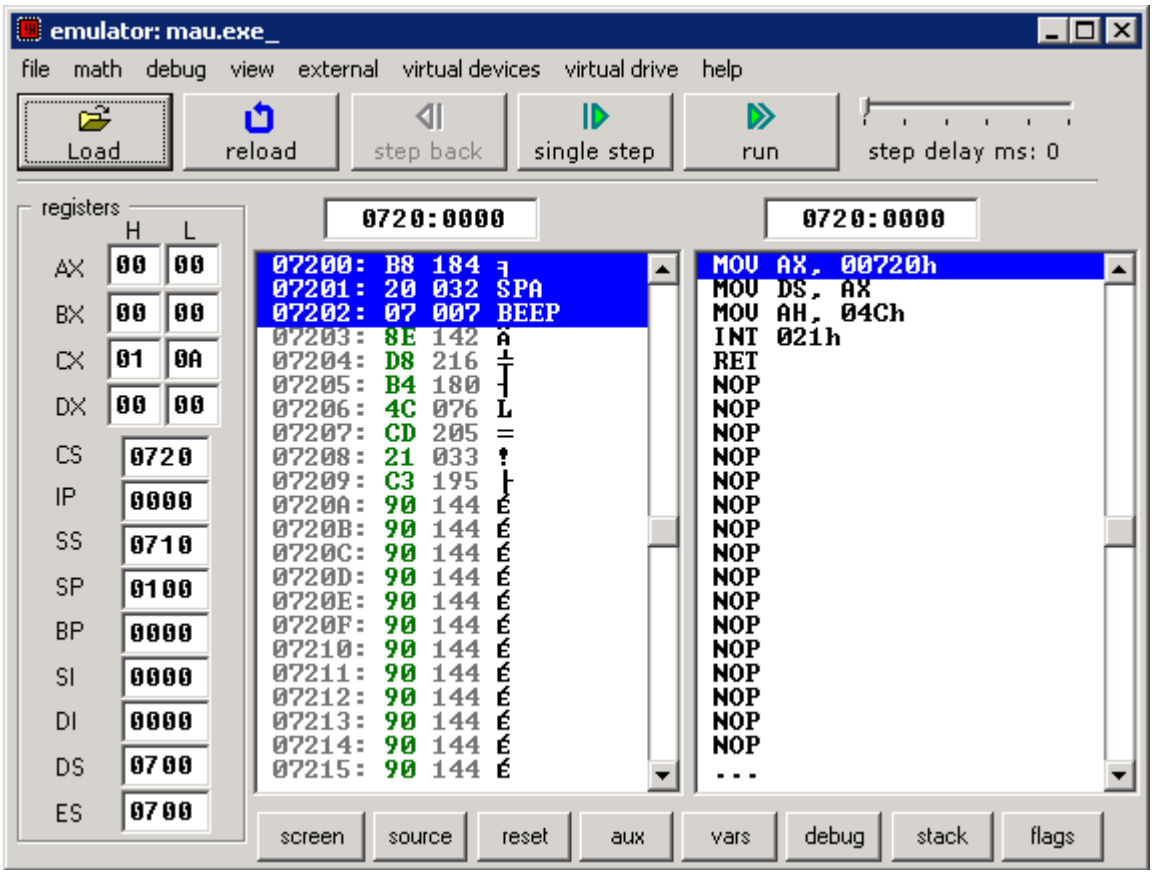
Nhấn Cancel để bỏ qua, cửa sổ soạn thảo của chương trình sẽ xuất hiện.

- Soạn chương trình hợp ngữ và nhấn vào nút Emulate để biên dịch và mô phỏng chương trình.

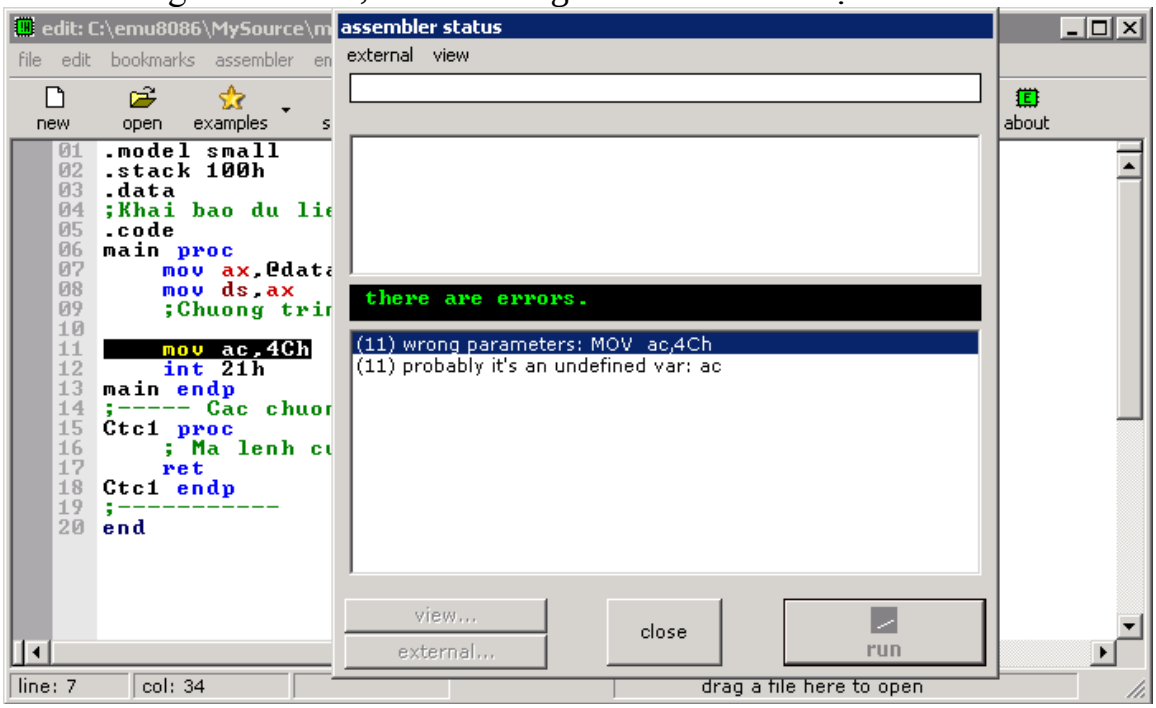
Sau khi biên dịch thành công (không có lỗi trong chương trình), Emu8086 sẽ mở thêm 2 cửa sổ: cửa sổ chương trình gốc và cửa sổ mô phỏng.

Chương trình gốc:

Cửa sổ mô phỏng:



Nếu chương trình có lỗi, cửa sổ thông báo lỗi sẽ xuất hiện:



Nhấn Close để quay lại cửa sổ soạn thảo.

 Cấu trúc chương trình 8086:

```
.model small
.stack 100h
.data
;Khai bao du lieu
.code
main proc
mov ax,@data
mov ds,ax
;Chuong trinh hop ngu
mov ah,4Ch
int 21h
main endp
;----- Cac chuong trinh con ---
Ctc1 proc
; Ma lenh cua chuong trinh con
ret
Ctc1 endp
;-----------
End
```

2. Cơ sở lý thuyết

2.1. Ngắt 21h

 Hàm 01h: nhập một ký tự từ bàn phím và hiện ký tự nhập ra màn hình. Nếu
không có ký tự nhập, hàm 01h sẽ đợi cho đến khi nhập.

- Gọi: AH = 01h
- Trả về: AL chứa mã ASCII của ký tự nhập

```
MOV AH,01h
INT 21h ; AL chứa mã ASCII của ký tự nhập
```

 Hàm 02h: xuất một ký tự trong thanh ghi DL ra màn hình tại vị trí con trỏ hiện hành

- Gọi AH = 02h, DL = mã ASCII của ký tự
- Trả về: không có

```
MOV AH,02h
MOV DL,'A'
INT 21h
```

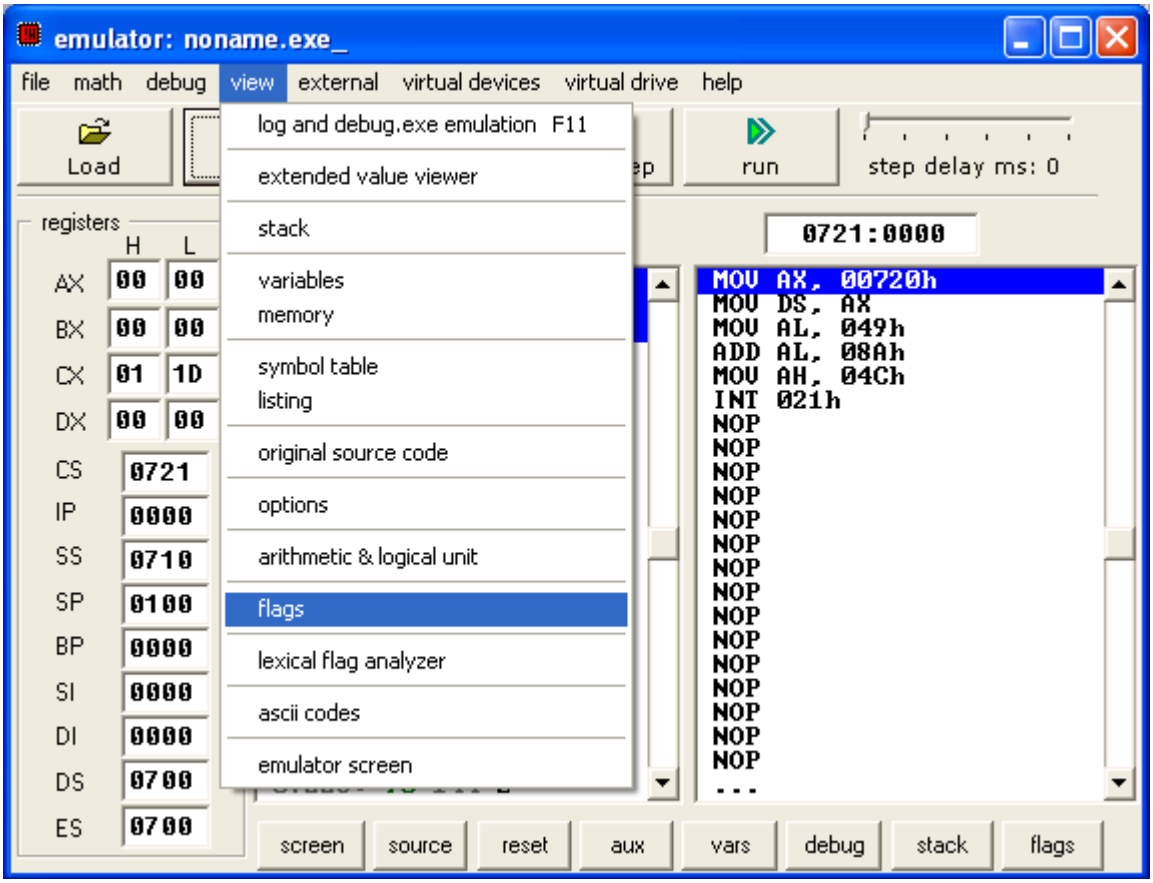
 Hàm 08h: giống hàm 01h nhưng không hiển thị ký tự ra màn hình

 Hàm 09h: xuất một chuỗi ký tự ra màn hình tại vị trí con trỏ hiện hành, địa

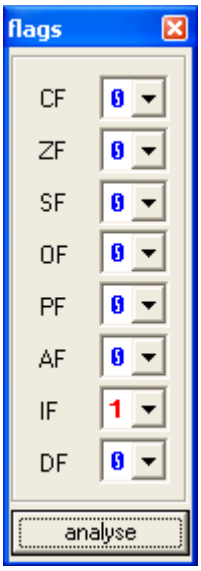
chỉ chuỗi được chứa trong DS:DX và phải được kết thúc bằng ký tự $
- Gọi AH = 09h, DS:DX = địa chỉ chuỗi
- Trả về: không có
.DATA
Msg DB 'Hello$'
…
MOV AH,09h
LEA DX,Msg
INT 21h
 Hàm 0Ah: nhập một chuỗi ký tự từ bàn phím (tối đa 255 ký tự), dùng phím
ENTER kết thúc chuỗi
- Gọi AH = 0Ah, DS:DX = địa chỉ lưu chuỗi
- Trả về: không có
Chuỗi phải có dạng sau:
- Byte 0: Số byte tối đa cần đọc (kể cả ký tự Enter)
- Byte 1: số byte đã đọc
- Byte 2: lưu các ký tự đọc
.DATA
Msg DB 101 ; Đọc tối đa 100 ký tự
DB ?
DB 101 DUP(?)
…
MOV AH,0Ah
LEA DX,Msg
INT 21h
 Hàm 0Bh: kiểm tra phím nhấn trên bàn phím
Gọi: AH = 0Bh
Trả về: AL = 0FFh nếu có nhấn phím, AL = 0 nếu không nhấn phím
 Hàm 4Ch: kết thúc chương trình
MOV AH,4Ch
INT 21h
2.2. Ngắt 10h
 Hàm 02h: Gọi AH = 02h, DH = dòng, DL = cột
MOV AH,02h
MOV DX,0F15h
INT 10h
3. Tiến trình thực hiện
3.1. Các lệnh cơ bản
- Bài 1.1: Thực hiện chương trình sau (cộng 49h với 8Ah):

```
.model small
.stack 100h
.data
msg db 'Hello$'
.code
main proc
mov ax,@data
mov ds,ax
mov al,49h
add al,8Ah
mov ah,4Ch
int 21h
main endp
;----------
End
```

- Nhấn vào nút Emulate để thực hiện mô phỏng.
- Tại cửa sổ mô phỏng, chọn menu **View** > **Flags** để hiển thị nội dung các cờ.



Cửa sổ flags cho phép xem nội dung của các cờ:



- Nhấn nút **Run** thực thi chương trình và quan sát nội dung các cờ. Giải
thích.
- Bài 1.2: Thực hiện chương trình cộng 2 số và kiểm tra nội dung các
cờ: CF, ZF, SF, OF, PF, AF. Từ đó rút ra kết luận về mục đích của các
cờ này.
- 0FFh + 01h
- 0FFh + 10h
- 40h + 55h
- 22h + 8Fh
- 99h + 7Ch
- Bài 1.3: Thực hiện lệnh nhân giữa 15h và 41h, 154h và 289Ah.
- Bài 1.4: Thực hiện lệnh chia giữa 5623h và 14h, 3219154h và 9Ah.
- Bài 1.5: Thực hiện lệnh dịch trái 1 bit, 2 bit, 3 bit giá trị 25h.
- Bài 1.6: Thực hiện lệnh quay trái 1 bit, 2 bit, 3 bit giá trị 25h.
- Bài 1.7: Thực hiện lệnh quay trái kết hợp với cờ carry 1 bit, 2 bit, 3 bit giá trị 25h.
- Bài 1.8: Thực hiện lệnh dịch phải 1 bit, 2 bit, 3 bit giá trị 25h.
- Bài 1.9: Thực hiện lệnh quay phải 1 bit, 2 bit, 3 bit giá trị 25h.
- Bài 1.10: Thực hiện lệnh quay phải kết hợp với cờ carry 1 bit, 2 bit, 3 bit giá trị 25h.
- Bài 1.11: Sử dụng lệnh IN để nhập dữ liệu và lệnh OUT để xuất dữ liệu ra thiết bị ngoại vi có địa chỉ **110**. Chú ý thêm vào chuỗi **#start=simple.exe#** ở đầu chương trình để kiểm tra kết quả.

3.2. Sử dụng ngắt 21h và ngắt 10h

 Bài 1.12: Dùng hàm 09h xuất chuỗi ra màn hình:

```
.model small
.stack 100h
.data
msg db 'Hello$'
.code
main proc
mov ax,@data
mov ds,ax
mov ah,09h ; Xuất chuỗi ra màn hình
lea dx,msg
```

```
int 21h
exit:
mov ah,4Ch
int 21h
main endp
```

CHƯƠNG III : LẬP TRÌNH HỢP NGỮ VÀ TÓM TẮT TẬP LỆNH

I. LẬP TRÌNH HỢP NGỮ :

Hợp ngữ (assembly language) là ngôn ngữ của máy tính có vị trí ở giữa ngôn ngữ máy và ngôn ngữ cấp cao.Các ngôn ngữ cấp cao như Pascal,C sử dụng các từ và các phát biểu dễ hiểu hơn. Ngôn ngữ máy (machine language) là ngôn ngữ ở dạng số nhị phân của máy tính. Một chương trình viết bằng ngôn ngữ máy là một chuỗi các byte nhị phân biểu diễn các lệnh mà máy tính thực thi được.

Hợp ngữ thay thế các mã nhị phân của ngôn ngữ máy bằng các mã gợi nhớ giúp ta dễ nhớ và dễ lập trình hơn.

Ex : lệnh cộng có mã nhị phân là "10110011" được hợp ngữ thay thế bằng mã gợi nhớ ADD

Một chương trình viết bằng hợp ngữ không thể được thực thi trực tiếp .Sau khi được viết xong chương trình này phải được dịch thành ngôn ngữ máy.

Một chương trình viết bằng hợp ngữ là chương trình viết dưới dạng các ký hiệu, các mã gợi nhớ … trong đó mỗi phát biểu tương ứng với một lệnh của ngôn ngữ máy.

Mỗi dòng lệnh được chia thành các trường cách biệt nhau bởi khoãng trắng hoặc Tab. Khuôn dạng tổng quát của mỗi dòng lệnh như sau:

[label :] mnemonic [operand][,operand][, …] [;comment]

Label: nhãn
Mnemonic : mã gợi nhớ
Operand : toán hạng
Comment : chú thích

a. Trường nhãn :

Nhãn là một loại ký hiệu và được nhận dạng bằng dấu "**:**" (kết thúc nhãn). Nhãn phải được bắt đầu bằng một ký tự chữ, dấu hỏi "?",dấu nối dưới "_" và tiếp theo phải là các ký tự chữ,các số, dấu "?",dấu "_". Nhãn có thể dài tối đa là 31 ký tự ở dạng chữ thường hoặc chữ in. Nhãn không được trùng với các từ khóa (các mã gợi nhớ, các chỉ dẫn, các toán tử hoặc các ký hiệu tiền định nghĩa).

b. Trường mã gợi nhớ :

Mã gợi nhớ là các ký hiệu biểu diễn cho các lệnh . Trường mã gợi nhớ của lệnh theo sau trường nhãn .

Ex : MOV,ANL,SETB ….

c. Trường toán hạng :

Trường toán hạng theo sau trường mã gợi nhớ. Trường này chứa địa chỉ hoặc dữ liệu mà lệnh sẽ sử dụng. Một nhãn có thể được dùng để biểu thị địa chỉ của dữ liệu .Các khả năng của trường toán hạng phụ thuộc vào thao tác. Có thao tác không có toán hạng (ex: lệnh RET,NOP..) trong khi các thao tác khác cho phép nhiều toán hạng cách nhau bởi dấu phẩy.

d. Trường chú thích :

Các ghi chú dùng để làm rõ chương trình được đặt trong trường chú thích ở cuối dòng lệnh. Các chú thích được bắt đầu bằng dấu " **;** ". Các chú thích có thể chiếm nhiều dòng riêng và cũng phải bắt đầu bằng dấu " **;** " . Các chương trình con và các phần có kích thước lớn của chương trình thường bắt đầu bởi một khối chú thích bao gồm nhiều dòng chú thích để giải thích các đặt trưng của chương trình.

II. TÓM TẮT TẬP LỆNH :

Cũng như các bộ vi xử lý 8 bit các lệnh của 8051 có các opcode 8 bit , do vậy số lệnh có thể lên đến 256 lệnh (thực tế có 255 lệnh ,1 lệnh không được định nghĩa). Ngoài opcode một số lệnh còn có thêm 1 hoặc 2 byte nữa cho dữ liệu hoặc địa chỉ . Tập lệnh có 139 lệnh 1 byte , 92 lệnh 2 byte và 24 lệnh 3 byte .

1. CÁC KIỂU ĐỊNH ĐỊA CHỈ :

Các kiểu định địa chỉ là phần cần thiết cho toàn bộ tập lệnh của mỗi một bộ vi xử lý , bộ vi điều khiển . Các kiểu định địa chỉ cho phép ta xác định rõ nguồn và đích của dữ liệu theo nhiều cách khác nhau phụ thuộc vào tình huống lập trình ,có 8 kiểu định địa chỉ :

- Thanh ghi (register)
- Trực tiếp (direct)
- Gián tiếp (indirect)
- Tức thời (immediate)
- Tương đối (relative)
- Tuyệt đối (absolute)
- Dài (long)
- Chỉ số (index)

1.1. ĐỊNH ĐỊA CHỈ THANH GHI :

Kiểu định địa chỉ thanh ghi đực ký hiệu là Rn, trong đó n có giá trị từ 0-7, A, DPTR, PC, C và cặp thanh ghi AB.

Ex : ADD A,R7

Có 4 dãy thanh ghi nhưng ở một thời điểm chỉ có một dãy tích cực. Các dãy thanh ghi chiếm 32 byte đầu tiên của RAM dữ liệu trên chip (00H-1FH). Để chọn dãy thanh ghi tích cực ta tác động lên các bit RS1,RS0 của từ trạng thái chương trình PSW. Khi hệ thống được reset thì dãy thanh ghi 0 mặc định được tích cực.

1.2. ĐỊNH ĐỊA CHỈ TRỰC TIẾP :

Kiểu định địa chỉ trực tiếp được sử dụng để truy xuất các biến nhớ hoặc các thanh ghi trên chip.

```
Ex :  MOV A,55H
      MOV P1,A        ( P1 có địa chỉ 90H )
```

1.3. ĐỊNH ĐỊA CHỈ GIÁN TIẾP :

Kiểu định địa chỉ gián tiếp được nhận biết nhờ vào ký tự @ đặt trước R0 hoặc R1. Các thanh ghi R0 và R1 có thể hoạt động như là các con trỏ và nội dung của chúng chỉ ra địa chỉ trong RAM nơi mà dữ liệu được đọc hay ghi.Ta cần đến kiểu định địa chỉ gián tiếp khi ta duyệt các vị trí liên tiếp trong bộ nhớ.

```
Ex :  Thực hiện việc xóa tuần tự RAM nội từ địa chỉ 60H-7FH
      MOV R0,#60H
LOOP: MOV @R0,#0
      INC R0
      CJNE R0,#80H,LOOP
      ( tiếp tục )
```

1.4. ĐỊNH ĐỊA CHỈ TỨC THỜI :

Khi toán hạng nguồn là một hằng số thay vì là một biến ,hằng số này có thể đưa vào lệnh và đây là byte dữ liệu tức thời. Các toán hạng tức thời được nhận biết nhờ vào ký tự # đặt trước chúng. Toán hạng này có thể là một hằng số học,một biến hoặc một biểu thức số học sử dụng các hằng số .

```
Ex :  MOV A,#12
      MOV DPTR,#8000H
```

1.5. ĐỊNH ĐỊA CHỈ TƯƠNG ĐỐI :

Kiểu định địa chỉ tương đối chỉ được sử dụng cho các lệnh nhảy .Một địa chỉ tương đối(còn được gọi là offset) là một giá trị 8 bit có dấu .Giá trị này được cộng với bộ đếm chương trình để tạo ra địa chỉ của lệnh tiếp theo cần được thực thi.

```
Ex :  SJMP THERE
      DJNE …
      CJNE …
```

Định địa chỉ tương đối có điểm lợi là cung cấp cho chúng ta mã không phụ thuộc vào vị trí , nhưng lại có điểm bất lợi là các đích nhảy bị giới hạn trong tầm -128 byte đến 127 byte.

1.6. ĐỊNH ĐỊA CHỈ TUYỆT ĐỐI :

Kiểu định địa chỉ tuyệt đối chỉ được sử dụng với các lệnh ACALL và AJMP. Đây là các lệnh 2 byte cho phép rẽ nhánh chương trình trong trang 2K hiện hành của bộ nhớ chương trình . Để không bị giới hạn ta có thể sử dụng lệnh LCALL, LJMP.

```
Ex :  LOOP:
      LCALL GIAI_MA
      LJMP LOOP
```

1.7. ĐỊNH ĐỊA CHỈ DÀI :

Kiểu định địa chỉ dài chỉ được dùng cho các lệnh LCALL và LJMP các lệnh 3 byte này chứa địa chỉ đích 16 bit . Lợi ích của lệnh này là sử dụng hết toàn bộ không gian nhớ chương trình 64K, nhưng lại có điểm bất lợi là lệnh dài đến 3 byte.

1.8. ĐỊNH ĐỊA CHỈ CHỈ SỐ :

Kiểu định địa chỉ chỉ số sử dụng một thanh ghi nền (hoặc bộ đếm chương trình hoặc con trỏ dữ liệu) và một offset (thanh chứa A) tạo thành dạng địa chỉ hiệu dụng cho lệnh JMP hoặc lệnh MOVC.

```
Ex:   MOVC A,@A+<base reg>
      JMP @A+DPTR
```

2. CÁC LOẠI LỆNH :

Các lệnh của 8051 được chia làm 5 nhóm:

- Nhóm lệnh số học
- Nhóm lệnh logic
- Nhóm lệnh di chuyển dữ liệu
- Nhóm lệnh xử lý bit
- Nhóm lệnh rẽ nhánh

Ghi chú :

```
Rn          : địa chỉ thanh ghi R0-R7
direct      : địa chỉ 8 bit trong RAM nội ( 00H-0FH )
@Ri         : địa chỉ gián tiếp sử dụng thanh ghi R0 hoặc R1
source            : toán hạng nguồn - có thể là Rn,direct hoặc @Ri
destination : toán hạng đích – có thể là Rn,direct hoặc @Ri
#data       : hằng số 8 bit ( Binary,Decimal,Hexa)
```

#data 16 : hằng số 16 bit
bit : địa chỉ trực tiếp của một bit
rel : địa chỉ tương đối (offset) 8 bit ex : nhãn
addr11 : địa chỉ 11 bit trong trang hiện hành
addr16 : địa chỉ 16 bit

2.1. CÁC LỆNH SỐ HỌC :

ADD A,source : cộng toán hạng nguồn với A
ADD A,#data
ADDC A,source : cộng toán hạng nguồn với A và cờ nhớ
ADDC A,#data
SUBB A,source : trừ bớt A bởi toán hạng nguồn và số mượn (cờ nhớ)
SUBB A,#data
INC A : tăng thanh ghi A một đơn vị
INC source
DEC A : giảm thanh ghi A một đơn vị
INC DPTR : tăng thanh ghi DPTR một đơn vị
MUL AB : nhân A với B
DIV AB : chia A bởi B (A chứa thương số,B chứa số dư)
DA A : hiệu chỉnh thập phân thanh ghi A

2.2. CÁC LỆNH LOGIC :

ANL A,source : AND
ANL A,#data
ANL direct,A
ANL direct,#data
ORL A,source : OR
ORL A,#data
ORL direct,A
ORL direct,#data
XRL A,source : XOR
XRL A,#data
XRL direct,A
XRL direct,#data
CLR A : xóa thanh ghi A
CPL A : lấy bù A
RL A : quay trái A (MSB=>LSB)
RLC A : quay trái A với cờ C
RR A : quay phải A (LSB=>MSB)

RRC A : quay phải A với cờ C
SWAP A : hoán đổi hai nibble (hai nữa 4 bit)

2.3. CÁC LỆNH DI CHUYỂN DỮ LIỆU :

MOV A,source : di chuyển toán hạng nguồn đến toán hạng đích
MOV A,#data
MOV dest,A
MOV dest,source
MOV dest,#data
MOV DPTR,#data16
MOVC A,@A+DPTR : di chuyển từ bộ nhớ chương trình
MOVC A,@A+PC
MOVX A,@Ri : di chuyển từ bộ nhớ dữ liệu
MOVX A,@DPTR
MOVX @Ri,A
MOVX @DPTR,A
PUSH direct : cất vào stack
POP direct : lấy ra từ stack
XCH A,source : trao đổi các byte
XCHD A,@Ri : trao đổi các digit thấp

2.4. CÁC LỆNH THAO TÁC TRÊN BIT :

CLR C : xóa bit
CLR bit : set bit bằng 1
SETB C
SETB bit
CPL C : lấy bù bit
CPL bit
ANL C,bit : AND bit với C
ANL C,/bit : AND NOT bit với C
ORL C,bit : OR bit với C
ORL C,/bit : OR NOT bit với C
MOV C,bit : di chuyển bit đến bit
MOV bit,C
JC rel : nhảy nếu C bằng 1
JNC rel : nhảy nếu C bằng 0
JB bit,rel : nhảy nếu bit bằng 1
JNB bit,rel : nhảy nếu bit bằng 0
JBC bit,rel : nhảy nếu bit bằng 1 rồi xóa bit

2.5. CÁC LỆNH RẼ NHÁNH :

ACALL addr11	: gọi chương trình con
LCALL addr16	
RET	: quay về từ chương trình con
RETI	: quay về từ trình phục vụ ngắt
AJMP addr11	: nhảy
LJMP addr16	
SJMP rel	
JMP @A+DPTR	
JZ rel	: nhảy nếu A bằng 0
JNZ rel	: nhảy nếu A khác 0
CJNE A,direct,rel	: so sánh và nhảy
CJNE A,#data,rel	
CJNE Rn,#data,rel	
DJNZ Rn,rel	: giảm và nhảy nếu khác không
DJNZ direct,rel	
NOP	: không làm gì

III. CẤU TRÚC CHƯƠNG TRÌNH :

1. TỔ CHỨC CHƯƠNG TRÌNH :

Các phần của chương trình được sắp xếp theo trình tự sau :

- Các phép gán
- Các lệnh khởi động (thiết lập timer, ngắt , nạp các giá trị ban đầu …)
- Thân chính của chương trình
- Các chương trình con
- Các định nghĩa hằng dữ liệu (DB và DW)

Gán :

Việc định nghĩa các hằng số bằng phát biểu gán làm cho chương trình dễ đọc và bảo trì hơn. Các hằng số được dùng trong suốt chương trình bằng cách thay thế các giá trị bằng các ký hiệu đã được gán. Khi chương trình được dịch thì các giá trị tương ứng được thay thế cho các ký hiệu.

Chương trình con :

Khi các chương trình trở nên lớn, ta phải chia nhỏ các thao tác lớn và phức tạp thành các thao nhỏ và đơn giản. Các thao tác nhỏ và đơn giản này được lập trình thành các chương trình con. Các chương trình con được bắt đầu bằng một nhãn và kết thúc bằng lệnh RET hoặc RETI.

Các chỉ dẫn :

Các chỉ dẫn là các lệnh đối với trình dịch hợp ngữ.

- ORG : chỉ dẫn ORG dùng để thiết lập một gốc mới của chương trình cho các phát biểu theo sau.

```
Ex :  ORG 0000H
      ORG 0030H
```

- END : là chỉ dẫn thông báo cho trình dịch hợp ngữ biết đã kết thúc chương trình nguồn. END là phát biểu cuối cùng của chương trình.
- EQU (equate) : chỉ dẫn EQU gán giá trị số cho tên của ký hiệu được định nghĩa.

```
Ex :  N27     EQU   27
      HERE    EQU   $
      DATA    EQU   50H
```

- BIT : chỉ dẫn BIT gán giá trị bit vào tên của ký hiệu.

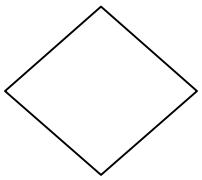
```
Ex :  MOTOR   BIT   P0.0
```

- DB : chỉ dẫn DB dùng để gán một hằng số vào bộ nhớ chương trình, các biểu thức theo sau có thể là một chuỗi của một hay nhiều giá trị byte. Chỉ dẫn DB cho phép chuỗi ký tự (đặt trong hai dấu nháy đơn) dài hơn hai ký tự, mỗi ký tự trong chuỗi được biến đổi thành mã ASCII . Nếu có một nhãn được nhãn được dùng, nhãn được gán địa chỉ của byte đầu tiên.

```
Ex :  SQUARES:    DB    0,1,4,9,16,25
      MESSAGE:    ‘ LAC HONG UNIVERSITY ‘,0
                  ; chuỗi ký tự kế thúc bởi 0
```

2. LƯU ĐỒ THUẬT GIẢI :

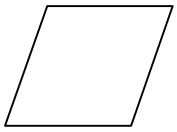
Lưu đồ thuật giải là các cộng cụ thường dùng cho các giai đoạn khởi đầu của lập trình hợp ngữ. Lưu đồ thuật giải là công cụ trực quan giúp ta dễ dàng trình bày và hiểu chương trình hợp ngữ một cách hệ thống. Lưu đồ thuật giải cho phép một yêu cầu được mô tả dưới dạng “điều gì phải được thực hiện “ hơn là “thực hiện điều đó bằng cách nào”. Các ký hiệu thường dùng nhất cho việc lập lưu đồ bao gồm :



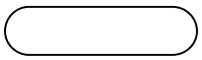
: khối quyết định ,khối này thường đặt ra các câu hỏi cho các câu trả lời có (Yes) hay không (No).

: mũi tên chỉ đường đi của chương trình

: khối xử lý

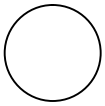


: khối xuất/nhập



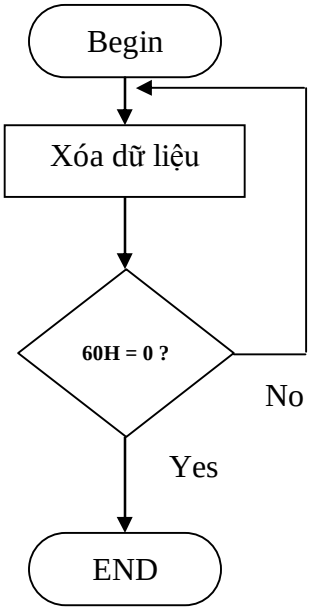
: điểm bắt đầu hoặc kết thúc chương trình

: chương trình con



: kết nối qua trang

Ex : Hãy viết một chương trình xóa các thanh ghi trong RAM nội từ địa chỉ 50H-60H.



Chương trình :

```
        MOV R0,#50H
        MOV A,#0
LOOP:
        MOV @R0,A
        INC R0
        CJNE R0,#61H,LOOP
        END
```

Ex : Hãy viết chương trình kiểm tra một nút nhấn S liên tục , nếu nút nhấn được tác động thì cho phép đèn D sáng và kết thúc chương trình.

Ex : Hãy viết chương trình cho một ổ khóa mật mã gồm 8 công tắc được kết nối vào port 0, chương trình sẽ kiểm tra nếu mã nhập vào giống với mã mặc định (00001111) thì cho phép mở khóa K và kết thúc chương trình, khóa K được kết nối vào chân P1.0 (quy ước 0: khóa , 1: mở khóa) .

CHƯƠNG 3: LẬP TRÌNH HỢP NGỮ

1. Các tập tin .EXE và .COM

DOS chỉ có thể thi hành được các tập tin dạng .COM và .EXE. Tập tin .COM thường dùng để xây dựng cho các chương trình nhỏ còn .EXE dùng cho các chương trình lớn.

1.1. Tập tin .COM

- Tập tin .COM chỉ có một đoạn nên kích thước tối đa của một tập tin loại này là 64 KB.
- Tập tin .COM được nạp vào bộ nhớ và thực thi nhanh hơn tập tin .EXE nhưng chỉ áp dụng được cho các chương trình nhỏ.
- Chỉ có thể gọi các chương trình con dạng near.

Khi thực hiện tập tin .COM, DOS định vị bộ nhớ và tạo vùng nhớ dài 256 byte ở vị trí 0000h, vùng này gọi là PSP (Program Segment Prefix), nó sẽ chứa các thông tin cần thiết cho DOS. Sau đó, các mã lệnh trong tập tin sẽ được nạp vào sau PSP ở vị trí 100h và đưa giá trị 0 vào stack. Như vậy, kích thước tối đa thực sự của tập tin .COM là 64 KB – 256 byte PSP – 2 byte stack.

Tất cả các thanh ghi đoạn đều chỉ đến PSP và thanh ghi con trỏ lệnh IP chỉ đến 100h, thanh ghi SP có giá trị 0FFFEh.

1.2. Tập tin .EXE

- Nằm trong nhiều đoạn khác nhau, kích thước thông thường lớn hơn 64 KB.
- Có thể gọi được các chương trình con dạng near hay far.
- Tập tin .EXE chứa một header ở đầu tập tin để chứa các thông tin điều khiển cho tập tin.

2. Khung của một chương trình hợp ngữ

Khung của một chương trình hợp ngữ có dạng như sau:

```
TITLE       Chương trình hợp ngữ
.MODEL      Kiểu kích thước bộ nhớ     ; Khai báo quy mô sử
                                       ; dụng bộ nhớ
.STACK      Kích thước                 ; Khai báo dung lượng
                                       ; đoạn stack
.DATA                                  ; Khai báo đoạn dữ  liệu
msg   DB    'Hello$'
.CODE                                  ; Khai báo đoạn mã
main  PROC
…
CALL        Subname                    ; Gọi chương trình con
…
main  ENDP

Subname     PROC                       ; Định nghĩa chương
                                       ; trình con
…
```


```
RET
Subname     ENDP
END   main
```

❖ **Quy mô sử dụng bộ nhớ:**

Bảng 3.1:

Loại	Mô tả
Tiny	Mã lệnh và dữ liệu nằm trong một đoạn
Small	Mã lệnh trong một đoạn, dữ liệu trong một đoạn
Medium	Mã lệnh không nằm trong một đoạn, dữ liệu trong một đoạn
Compact	Mã lệnh trong một đoạn, dữ liệu không nằm trong một đoạn
Large	Mã lệnh không nằm trong một đoạn, dữ liệu không nằm trong một đoạn và không có mảng nào lớn hơn 64KB
Huge	Mã lệnh không nằm trong một đoạn, dữ liệu không nằm trong một đoạn và các mảng có thể lớn hơn 64KB

Thông thường, các ứng dụng đơn giản chỉ đòi hỏi mã chương trình không quá 64 KB và dữ liệu cũng không lớn hơn 64 KB nên ta sử dụng ở dạng Small:

```
.MODEL      SMALL
```

❖ **Khai báo kích thước stack:**

Khai báo stack dùng để dành ra một vùng nhớ dùng làm stack (chủ yếu phục vụ cho chương trình con), thông thường ta chọn khoảng 256 byte là đủ để sử dụng (nếu không khai báo thì chương trình dịch tự động cho kích thước stack là 1 KB):

```
.STACK      256
```

❖ **Khai báo đoạn dữ liệu:**

Đoạn dữ liệu dùng để chứa các biến và hằng sử dụng trong chương trình.

❖ **Khai báo đoạn mã:**

Đoạn mã dùng chứa các mã lệnh của chương trình. Đoạn mã bắt đầu bằng một chương trình chính và có thể có các lệnh gọi chương trình con (CALL).

Một chương trình chính hay chương trình con bắt đầu bằng lệnh PROC và kết thúc bằng lệnh ENDP (đây là các lệnh giả của chương trình dịch). Trong chương trình con, ta sử dụng thêm lệnh RET để trả về địa chỉ lệnh trước khi gọi chương trình con.

3. Cú pháp của các lệnh trong chương trình hợp ngữ

Một dòng lệnh trong chương trình hợp ngữ gồm có các trường (field) sau (không nhất thiết phải đầy đủ tất cả các trường):

Tên	Lệnh	Toán hạng	Chú thích
A:	MOV	AH,10h	; Đưa giá trị 10h vào thanh ghi AH
Main	PROC		

Trường tên chứa nhãn, tên biến hay tên thủ tục. Các tên nhãn có thể chứa tối đa 31 ký tự, không chứa ký tự trắng (space) và không được bắt đầu bằng số (A: hay Main:). Các nhãn được kết thúc bằng dấu '**:**'.

Trường lệnh chứa các lệnh sẽ thực hiện. Các lệnh này có thể là các lệnh thật (MOV) hay các lệnh giả (PROC). Các lệnh thật sẽ được dịch ra mã máy.

Trường toán hạng chứa các toán hạng cần thiết cho lệnh (AH,10h).

Trường chú thích phải được bắt đầu bằng dấu '**;**'. Trường này chỉ dùng cho người lập trình để ghi các lời giải thích cho chương trình. Chương trình dịch sẽ bỏ qua các lệnh nằm phía sau dấu **;**.

3.1. Khai báo dữ liệu

Khi khai báo dữ liệu trong chương trình, nếu sử dụng số nhị phân, ta phải dùng thêm chữ **B** ở cuối, nếu sử dụng số thập lục phân thì phải dùng chữ **H** ở cuối. ***Chú ý rằng đối với số thập lục phân, nếu bắt đầu bằng chữ A..F thì phải thêm vào số 0 ở phía trước.***

Ví dụ:

```
1011b       ; Số nhị phân
1111        ; Số thập phân
1011h       ; Số thập lục phân
```

3.2. Khai báo biến

Khai báo biến nhằm để chương trình dịch cung cấp một địa chỉ xác định trong bộ nhớ. Ta dùng các lệnh giả sau để định nghĩa các biến ứng với các kiểu dữ liệu khác nhau: DB (define byte), DW (define word) và DD (define double word).

VD:

```
A1      DB      1       ; Định nghĩa biến A1 dài 1 byte (chương
                        ; trình dịch sẽ dùng 1 byte trong bộ nhớ để
                        ; lưu trữ A1), trị ban đầu A1 = 1
A2      DB      ?       ; Biến A2 kiểu byte, không có giá trị gán
                        ; ban đầu
A3      DB      'A'     ; Biến kiểu ký tự
A4      DW      1       ; Định nghĩa biến A4 dài 2 byte, giá trị ban
                        ; đầu A4 = 1, ta cũng có thể dùng dấu ? để
                        ; xác định biến không cần khởi tạo giá trị ban đầu
A5      DD      1       ; Biến A5 dài 4 byte
A6      DB      1,2,3   ; Định nghĩa biến mảng (array) gồm có 3
                        ; phần tử, mỗi phần tử dài 1 byte (nghĩa là
                        ; sẽ dùng 3 byte lưu trữ) với các giá trị ban
                        ; đầu của các phần tử lần lượt là 1,2,3
A7      DB      10      DUP(0)
                        ; Khai báo biến mảng gồm 10 phần tử, mỗi
                        ; phần tử có chiều dài 1 byte với giá trị gán
                        ; ban đầu là 0
```


```
A8    DB    10    DUP(?)
                  ; Khai báo biến mảng gồm 10 phần tử, mỗi
                  ; phần tử có chiều dài 1 byte, không cần
                  ; gán giá trị ban đầu
```

Ngoài ra ta có thể dùng các toán tử DUP lồng vào nhau khi khai báo biến mảng. Giả sử ta cần khai báo mảng A9 có các giá trị gán ban đầu 1,2,3,1,1,3,2,2,1,1,3,2,2. Ta có thể thực hiện như sau:

```
      A9    DB    1,2,3,1,1,3,2,2,1,1,3,2,2
Hay:  A9    DB    1,2,3,2 DUP(1,1,3,2,2)
Hay:  A9    DB    1,2,3,2 DUP(2 DUP(1),3,2 DUP(2))
```

Đối với các biến có nhiều hơn 1 byte, byte thấp sẽ chứa ở ô nhớ có địa chỉ thấp và byte cao sẽ chứa ở ô nhớ có địa chỉ cao.

VD:

```
A10   DW    1234h
```

Biến A10 giả sử bắt đầu lưu tại địa chỉ 1000h thì ô nhớ 1000h chứa giá trị 34h còn ô nhớ 1001h chứa giá trị 12h.

Đối với biến kiểu chuỗi (string), thực chất là một mảng các ký tự, ta có thể khai báo như sau:

```
      A11   DB    'ABCD'
Hay   A11   DB    65h,66h,67h,68h
```

Sau lệnh khai báo này thì ô nhớ 1000h (giả sử biến A11 lưu trữ tại địa chỉ 1000h) chứa 'A', 1001h chứa 'B', 1002h chứa 'C' và 1003h chứa 'D'.

3.3. Khai báo hằng

Các hằng khai báo trong chương trình hợp ngữ bằng lệnh giả EQU để chương trình dễ hiểu hơn. Hằng có thể ở dạng số, ký tự hay chuỗi.

VD:

```
A12   EQU   10
A13   EQU   'AAA'
```

Sau khi sử dụng khai báo này, nếu ta dùng lệnh:

```
MOV  AH,A12
```

thì AH = 10h

```
A14   DB    'B',A13
```

thì khai báo chuỗi A14 với giá trị gán ban đầu là 'BAAA'.

4. Các toán tử trong hợp ngữ

❖ **Toán tử số học:**

Bảng 3.2:

Toán tử	Cú pháp	Mô tả
+	+bt	Số dương
-	-bt	Số âm
*	bt1*bt2	Nhân
/	bt1/bt2	Chia
mod	bt1 mod bt2	Lấy phần dư
+	bt1 + bt2	Cộng
-	bt1 – bt2	Trừ
shl	bt shl n	Dịch trái n bit
shr	bt shr n	Dịch phải n bit

Trong đó bt, bt1, bt2 là các biểu thức hằng, n là số nguyên.

```
VD:  MOV  AH,(8+1)*7/3          ; AH ← 21
     MOV  AH, 00010001b shr 2   ; AH ← 0000 0100b
     MOV  AH,00010001b shl 2    ; AH ← 0100 0100b
     MOV  AH,100 mod 3          ; AH ← 1
```

❖ **Toán tử logic:**

Bao gồm các toán tử AND, OR, NOT, XOR

```
VD:  MOV  AH,10 OR 4 AND 2      ; AH = 10
     MOV  AH, 0F0h AND 7Fh      ; AH = 70h
```

❖ **Toán tử quan hệ:**

Các toán tử quan hệ so sánh 2 biểu thức, cho giá trị true (-1) nếu điều kiện thoả và false (0) nếu không thoả.

Bảng 3.3:

Toán tử	Cú pháp	Mô tả
EQ	bt1 EQ bt2	Bằng
NE	bt1 NE bt2	Không bằng
LT	bt1 LT bt2	Nhỏ hơn
LE	bt1 LE bt2	Nhỏ hơn hay bằng
GT	bt1 GT bt2	Lớn hơn
GE	bt1 GE bt2	Lớn hơn hay bằng

❖ **Các toán tử cung cấp thông tin:**

➢ **Toán tử SEG:**

SEG bt

Toán tử SEG xác định địa chỉ đoạn của biểu thức ***bt***. ***bt*** có thể là biến, nhãn, hay các toán hạng bộ nhớ.

➢ **Toán tử OFFSET:**

OFFSET bt

Toán tử OFFSET xác định địa chỉ offset của biểu thức ***bt***. ***bt*** có thể là biến, nhãn, hay các toán hạng bộ nhớ.

```
VD:   MOV  AX,SEG A          ; Nạp địa chỉ đoạn và địa chỉ offset
      MOV  DS,AX             ; của biến A vào cặp thanh ghi
      MOV  AX,OFFSET A       ; DS:AX
```

➢ **Toán tử chỉ số [ ]: (index operator)**

Toán tử chỉ số thường dùng với toán hạng trực tiếp và gián tiếp.

➢ **Toán tử (:) (segment override operator)**

Segment:bt

Toán tử **:** quy định cách tính địa chỉ đối với segment được chỉ. ***Segment*** là các thanh ghi đoạn CS, DS, ES, SS.

Chú ý rằng khi sử dụng toán tử : kết hợp với toán tử [ ] thì ***segment:*** phải đặt ngoài toán tử [ ].

VD: Cách viết [CS:BX] là sai, ta phải viết CS:[BX]

➢ **Toán tử TYPE:**

TYPE bt

Trả về giá trị biểu thị dạng của biểu thức ***bt***.

- Nếu ***bt*** là biến thì sẽ trả về 1 nếu biến có kiểu byte, 2 nếu biến có kiểu word, 4 nếu biến có kiểu double word.
- Nếu ***bt*** là nhãn thì trả về 0FFFFh nếu bt là near và 0FFFEh nếu bt là far.
- Nếu ***bt*** là hằng thì trả về 0.

➢ **Toán tử LENGTH:**

LENGTH bt

Trả về số các đơn vị cấp cho biến ***bt***

➢ **Toán tử SIZE:**

SIZE bt

Trả về tổng số các byte cung cấp cho biến ***bt***

```
VD:   A      DD    100    DUP(?)
      MOV  AX,LENGTH A     ; AX = 100
      MOV  AX,SIZE A       ; AX = 400
```


❖ **Các toán tử thuộc tính:**

➢ **Toán tử PTR:**

Loai PTR bt

Toán tử này cho phép thay đổi dạng của biểu thức ***bt***.

- Nếu ***bt*** là biến hay toán hạng bộ nhớ thì ***Loai*** là byte, word hay dword.
- Nếu ***bt*** là nhãn thì ***Loai*** là near hay far.

```
VD:  A     DW   100   DUP(?)
     B     DD   ?
     MOV  AH,BYTE PTR A        ; Đưa byte đầu tiên trong mảng A
                               ; vào thanh ghi AH
     MOV  AX,WORD PTR B        ; Đưa 2 byte thấp trong biến B
                               ; vào thanh ghi AX
```

➢ **Toán tử HIGH, LOW:**

HIGH bt

LOW bt

Cho giá trị của byte cao và thấp của biểu thức ***bt***, ***bt*** phải là một hằng.

```
VD:  A     EQU  1234h
     MOV  AH,HIGH A            ; AH ← 12h
     MOV  AH,LOW A             ; AH ← 34h
```

5. Các cách định địa chỉ trong hợp ngữ

❖ **Toán hạng trực tiếp:**

Toán hạng trực tiếp là một biểu thức hằng xác định. Các hằng số có thể ở dạng thập phân (có dấu và không dấu), nhị phân, thập lục phân, các hằng số định nghĩa bằng lệnh EQU, …

```
VD:  MOV  AH,10
     MOV  AH,1010b
     MOV  AH,0Ah
     MOV  AH,A12
     MOV  AX,OFFSET msg
     MOV  AX,SEG msg
```

❖ **Toán hạng thanh ghi:**

Các thanh ghi có thể sử dụng trong phép định địa chỉ thanh ghi là AH, BH, CH, DH, AL, BL, CL, DL, AX, BX, CX, DX, SP, BP, SI, DI, CS, DS, ES, SS.

❖ **Toán hạng bộ nhớ:**

➢ **Trực tiếp:**

Toán hạng này xác định dữ liệu lưu trong bộ nhớ tại một địa chỉ xác định khi dịch, địa chỉ này là một biểu thức hằng (có thể kết hợp với toán tử chỉ số [ ] hay toán tử +, -, :). Thanh ghi đoạn mặc định là thanh ghi DS nhưng ta có thể dùng toán tử : để chỉ thanh ghi đoạn khác.

VD:
```
A       DW    1000h
B       DB    100   DUP(0)
MOV  AX,A              ; Chuyển nội dung của biến A vào
MOV  AX,[A]            ; thanh ghi AX
MOV  AH,B              ; Truy xuất phần tử đầu tiên của
MOV  AH,B[0]           ; mảng B
MOV  AH,B + 1          ; Truy xuất phần tử thứ hai của
MOV  AH,B[1]           ; mảng B
MOV  AH,B  + 5         ; Truy xuất phần tử thứ 6 của
MOV  AH,B[5]           ; mảng B
```

Chú ý rằng lệnh MOV AX,[1000h] sẽ chuyển giá trị 1000h vào thanh ghi AX. Nếu muốn chuyển nội dung tại ô nhớ 1000h vào thanh ghi AX thì phải dùng lệnh MOV AX,DS:[1000h] hay MOV AX,DS:1000h

- **Gián tiếp:**

Toán hạng bộ nhớ gián tiếp cho phép dùng các thanh ghi BX, BP, SI, DI để chỉ các giá trị trong bộ nhớ.

VD:
```
MOV  BX,2
MOV  SI,3
MOV  AH,B[BX]          ; Chuyển phần tử thứ 3 của mảng B
                       ; vào thanh ghi AH
MOV  AH,B[BX+1]        ; Chuyển phần tử thứ 4 của mảng B
MOV  AH,B[BX]+1        ; vào thanh ghi AH (BX + 1 = 3)
MOV  AH,B[BX+SI]       ; Chuyển phần tử thứ 6 của mảng B
MOV  AH,B[BX][SI]      ; vào thanh ghi AH
MOV  AH,[B+BX+SI]      ; BX + SI = 5
MOV  AH,[B][BX][SI]
MOV  AH,B[BX+SI+5]     ; Chuyển phần tử thứ 11 của mảng B
MOV  AH,B[BX][SI]+5    ; vào thanh ghi AH
MOV  AH,[B+BX+SI+5]    ; BX + SI + 5 = 10
```

6. Tạo và thực thi chương trình hợp ngữ

Ta có thể tạo và thực thi một chương trình hợp ngữ trên một máy PC theo các bước sau:

- Dùng một chương trình soạn thảo văn bản ***không định dạng*** (như NC) tạo một tập tin chứa chương trình hợp ngữ (gán phần mở rộng của tập tin này là .ASM, giả sử là TEMP.ASM).
- - Dùng chương trình TASM.EXE (Turbo Assembler) để dịch ra mã máy dạng .OBJ: **TASM TEMP**
- Sau khi dịch xong, ta sẽ được file TEMP.OBJ chứa các mã máy của chương trình. Để chuyển thành file thực thi, ta dùng chương trình TLINK.EXE để chuyển thành tập tin .EXE: **TLINK TEMP**
- - Nếu tập tin thực thi ở dạng .COM thì ta dùng thêm chương trình EXE2BIN.EXE: **EXE2BIN TEMP TEMP.COM**

7. Tập lệnh hợp ngữ

7.1. Nhóm lệnh chuyển dữ liệu

7.1.1. Nhóm lệnh chuyển dữ liệu đa dụng

❖ Lệnh **MOV dst,src**: chuyển nội dung toán hạng src vào toán hạng dst. Toán hạng nguồn src có thể là thanh ghi (reg), bộ nhớ (mem) hay giá trị tức thời (immed); toán hạng đích dst có thể là reg hay mem.

Lệnh MOV có thể có các trường hợp sau:

Reg8 ← reg8	MOV AL,AH
Reg16 ← reg16	MOV AX,BX
Mem8 ← reg8	MOV [BX],AL
Reg8 ← mem8	MOV AL,[BX]
Mem16 ← reg16	MOV [BX],AX
Reg16 ← mem16	MOV AX,[BX]
Reg8 ← immed8	MOV AL,04h
Mem8 ← immed8	MOV mem[BX],01h
Reg16 ← immed16	MOV AL,0F104h
Mem16 ← immed16	MOV mem[BX],0101h
SegReg ← reg16	MOV DS,AX
SegReg ← mem16	MOV DS,mem
Reg16 ← segreg	MOV AX,DS
Mem16 ← segreg	MOV [BX],DS

- Lệnh MOV không ảnh hưởng đến các cờ.
- Không thể chuyển trực tiếp dữ liệu giữa hai ô nhớ mà phải thông qua một thanh ghi

```
MOV  AX,mem1
MOV  mem2,AX
```

- Không thể chuyển giá trị trực tiếp vào thanh ghi đoạn

```
MOV  AX,1010h
MOV  DS,AX
```

- Không thể chuyển trực tiếp giữa 2 thanh ghi đoạn
- Không thể dùng thanh ghi CS làm toán hạng đích

❖ Lệnh **XCHG dst,src**: (Exchange) hoán chuyển nội dung 2 toán hạng. Toán hạng chỉ có thể là reg hay mem.

- Lệnh XCHG không ảnh hưởng đến các cờ
- Không thể dùng cho các thanh ghi đoạn

❖ Lệnh **PUSH src**: cất nội dung một thanh ghi vào stack. Toán hạng là reg16

❖ Lệnh **POP dst**: lấy dữ liệu 16 bit từ stack đưa vào toán hạng dst.

Ta có thể dùng nhiều lệnh PUSH để cất dữ liệu vào stack nhưng khi dùng lệnh POP để lấy dữ liệu ra thì phải dùng theo thứ tự ngược lại.

```
PUSH    AX
PUSH    BX
```


```
PUSH    CX
…
POP     CX
POP     BX
POP     AX
```

- Lệnh **XLAT [src]**: chuyển nội dung của ô nhớ 8 bit vào thanh ghi AL. Địa chỉ ô nhớ xác định bằng cặp thanh ghi DS:BX (nếu không chỉ ra src) hay src, địa chỉ offset chứa trong thanh ghi AL.

Lệnh XLAT tương đương với các lệnh:

```
MOV  AH,0
MOV  SI,AX
MOV  AL,[BX+SI]
```

7.1.2. Nhóm lệnh chuyển địa chỉ

- Lệnh **LEA reg16,mem16**: (Load Effective Address) chuyển địa chỉ offset của toán hạng bộ nhớ vào thanh ghi reg16.

Lệnh này sẽ tương đương với **MOV reg16, OFFSET mem16**

- Lệnh **LDS reg16,mem32**: (Load pointer using DS) chuyển nội dung bộ nhớ toán hạng mem32 vào cặp thanh ghi DS:reg16.

Lệnh LDS AX,mem tương đương với:

```
MOV  AX,mem
MOV  BX,mem+2
MOV  DS,BX
```

- Lệnh **LES reg16,mem32**: (Load pointer using ES) giống như lệnh LDS nhưng dùng cho thanh ghi ES

7.1.3. Nhóm lệnh chuyển cờ hiệu

- Lệnh **LAHF**: (Load AH from flag) chuyển các cờ SF, ZF, AF, PF và CF vào các bit 7,6,4,2 và 0 của thanh ghi AH (3 bit còn lại không đổi)

- Lệnh **SAHF**: (Store AH into flag) chuyển các bit 7,6,4,2 và 0 của thanh ghi AH vào các cờ SF, ZF, AF, PF và CF.

- Lệnh **PUSHF**: chuyển thanh ghi cờ vào stack

- Lệnh **POPF**: lấy dữ liệu từ stack chuyển vào thanh ghi cờ

7.1.4. Nhóm lệnh chuyển dữ liệu qua cổng

Mỗi I/O port giao tiếp với CPU sẽ có một địa chỉ 16 bit cho nó. CPU gởi hay nhận dữ liệu từ cổng bằng cách chỉ đến địa chỉ cổng đó. Tuỳ theo chức năng mà cổng có thể: chỉ đọc dữ liệu (input port), chỉ ghi dữ liệu (output port) hay có thể đọc và ghi dữ liệu (input/output port).

❖ Lệnh **IN**: đọc dữ liệu từ cổng và đưa vào thanh ghi AL

```
IN      AL,port8
IN      AL,DX
```

Nếu địa chỉ port chỉ có 8 bit thì có thể đưa giá trị trực tiếp vào, nếu là 16 bit thì phải thông qua thanh ghi AX.

❖ Lệnh **OUT**: ghi dữ liệu trong thanh ghi AL ra cổng

```
OUT   port8,AL
OUT   DX,AL
```

```
VD:   MOV  AL,3
      OUT  61h,AL          ; Gởi giá trị 03h ra cổng 61h
      MOV  AL,1
      MOV  DX,03F8h        ; Xuất ra cổng máy in
      OUT  DX,AL
      MOV  DX,03F8h
      IN   AL,DX           ; Đọc dữ liệu từ cổng máy in
```

7.2. Nhóm lệnh chuyển điều khiển

7.2.1. Lệnh nhảy không điều kiện JMP

JMP label
JMP reg/mem

Lệnh JMP dùng để chuyển điều khiển chương trình từ vị trí này sang vị trí khác (thay đổi nội dung cặp thanh ghi CS:IP).

7.2.2. Lệnh nhảy có điều kiện

Lệnh nhảy có điều kiện chỉ sử dụng cho các nhãn nằm trong khoảng từ –127 đến 128 byte so với vị trí của lệnh.

❖ Lệnh **JA label**: (Jump if Above)
Nếu CF = 0 và ZF = 0 thì JMP label

❖ Lệnh **JAE label**: (Jump if Above or Equal)
Nếu CF = 0 thì JMP label

❖ Lệnh **JB label**: (Jump if Below)
Nếu CF = 1 thì JMP label

❖ Lệnh **JBE label**: (Jump if Below or Equal)
Nếu CF = 1 hoặc ZF = 1 thì JMP label

❖ Lệnh **JNA label**: (Jump if Not Above)
Giống lệnh JBE

❖ Lệnh **JNAE label**: (Jump if Not Above or Equal)
Giống lệnh JB

❖ Lệnh **JNB label**: (Jump if Not Below)
Giống lệnh JAE

❖ Lệnh **JNBE label**: (Jump if Not Below or Equal)
Giống lệnh JA

❖ Lệnh **JG label**: (Jump if Greater)
Nếu SF = OF và ZF = 0 thì JMP label

❖ Lệnh **JGE label**: (Jump if Greater or Equal)
Nếu SF = OF thì JMP label

❖ Lệnh **JL label**: (Jump if Less)
Nếu SF <> OF thì JMP label

❖ Lệnh **JLE label**: (Jump if Less or Equal)
Nếu CF <> OF hoặc ZF = 1 thì JMP label

❖ Lệnh **JNG label**: (Jump if Not Greater)
Giống lệnh JLE

❖ Lệnh **JNGE label**: (Jump if Not Greater or Equal)
Giống lệnh JL

❖ Lệnh **JNL label**: (Jump if Not Less)
Giống lệnh JGE
❖ Lệnh **JNLE label**: (Jump if Not Less or Equal)
Giống lệnh JG

❖ Lệnh **JC label**: (Jump if Carry)
Giống lệnh JB

❖ Lệnh **JNC label**: (Jump if Not Carry)
Giống lệnh JNB

❖ Lệnh **JZ label**: (Jump if Zero)
Nếu ZF = 1 thì JMP label

❖ Lệnh **JE label**: (Jump if Equal)
Giống lệnh JZ

❖ Lệnh **JNZ label**: (Jump if Not Zero)
Nếu ZF = 0 thì JMP label

❖ Lệnh **JNE label**: (Jump if Equal)
Giống lệnh JNZ

❖ Lệnh **JS label**: (Jump on Sign)
Nếu SF = 1 thì JMP label

❖ Lệnh **JNS label**: (Jump if No Sign)
Nếu SF = 0 thì JMP label

❖ Lệnh **JO label**: (Jump on Overflow)
Nếu OF = 1 thì JMP label

❖ Lệnh **JNO label**: (Jump if No Overflow)
Nếu OF = 0 thì JMP label

❖ Lệnh **JP label**: (Jump on Parity)
Nếu PF = 1 thì JMP label

❖ Lệnh **JNP label**: (Jump if No Parity)
Nếu PF = 0 thì JMP label

❖ Lệnh **JCXZ label**: (Jump if CX Zero)
Nếu CX = 1 thì JMP label

7.2.3. Lệnh so sánh

CMP left(reg/mem), right(reg/mem/immed)

Lệnh CMP dùng để so sánh nội dung 2 toán hạng, kết quả chứa vào thanh ghi cờ và không làm thay đổi nội dung các toán hạng.

VD: Đoạn chương trình so sánh 2 số A và B: A >B thì nhảy đến label1, A = B thì nhảy đến label2, A < B thì nhảy đến label3.

```
MOV  AX,A
CMP  AX,B
JG   label1
JL   label2
JMP  label3
```

7.2.4. Các lệnh vòng lặp

❖ Lệnh **LOOP**:
LOOP label
Mô tả:
CX = CX – 1
Nếu CX <> 0 thì JMP label

❖ Lệnh **LOOPE**:
LOOPE label
Mô tả:
CX = CX – 1
Nếu (ZF = 1) và (CX <> 0) thì JMP label

❖ Lệnh **LOOPZ**:
Giống lệnh LOOPE

❖ Lệnh **LOOPNE**:
LOOPNE label
Mô tả:
CX = CX – 1
Nếu (ZF = 0) và (CX <> 0) thì JMP label

❖ Lệnh **LOOPNZ**:
Giống lệnh LOOPNE

7.2.5. Lệnh liên quan đến chương trình con

❖ Lệnh **CALL**:
Lệnh CALL dùng để gọi một chương trình con, có thể là near hay far.

```
CALL    label       ; Gọi chương trình con tại vị trí xác định
                    ; bởi nhãn label
CALL    reg/mem     ; Gọi chương trình con tại vị trí xác định
                    ; trong reg/mem
```

❖ Lệnh **RET**: (return)
RET [n]
RETN [n]
RETF [n]
Lệnh RET dùng để kết thúc chương trình con, điều khiển sẽ được đưa về địa chỉ trước khi gọi chương trình con. RETN để kết thúc chương trình con dạng near và RETF dùng để kết thúc chương trình con dạng far.

Trong trường hợp lệnh RET có hằng số n theo sau thì sẽ cộng với thanh ghi SP giá trị n (n phải là số chẵn). Lệnh này dùng để loại bỏ một số tham số chương trình con sử dụng ra khỏi stack.

7.3. Nhóm lệnh xử lý số học

7.3.1. Xử lý phép cộng

❖ Lệnh **ADD dst,src**:
dst ← dst + src
Toán hạng src có thể là reg, mem hay immed còn toán hạng dst là reg hay mem.

- Không thể cộng trực tiếp 2 thanh ghi đoạn
- Lệnh ADD ảnh hưởng đến các cờ sau:
  + Cờ CF: = 1 khi kết quả phép cộng có nhớ hay có mượn
  + Cờ AF: = 1 khi kết quả phép cộng có nhớ hay có mượn đối với 4 bit thấp
  + Cờ PF: = 1 khi kết quả phép cộng có tổng 8 bit thấp là một số chẵn.
  + Cờ ZF: = 1 khi kết quả phép cộng là 0.
  + Cờ SF: = 1 nếu kết quả phép cộng là một số âm
  + Cờ OF: = 1 nếu kết quả phép cộng bị sai dấu, nghĩa là vượt ra ngoài phạm vi lớn nhất hay nhỏ nhất mà số có dấu có thể chứa trong toán hạng dst.

❖ Lệnh **ADC dst, src**: (Add with Carry)

dst ← dst + src + CF

Lệnh ADC thường dùng để cộng các số lớn hơn 16 bit.

❖ Lệnh **INC dst**: (Increment)

dst ← dst + 1

Dst có thể là reg hay mem.

❖ Lệnh **AAA**: (ASCII Adjust for Addition)

Hiệu chỉnh kết quả phép cộng 2 số BCD dạng không nén (mỗi chữ số BCD lưu bằng 1 byte).

```
VD:   MOV  AX,9
      MOV  BX,3
      ADD  AL,BL          ; Kết quả là AX = 0Ch
      AAA                 ; AX = 0102h (AH = 1, AL = 2)
```

Lệnh AAA chỉ ảnh hưởng đến các cờ AF và CF, không ảnh hưởng đến các cờ còn lại.

❖ Lệnh **DAA**: (Decimal Adjust for Addition)

Hiệu chỉnh kết quả phép cộng 2 số BCD dạng nén (mỗi chữ số BCD lưu bằng 4 bit, nghĩa là 1 byte biểu diễn được các số nguyên từ 0 đến 99).

```
VD:   MOV  AX,4338h
      ADD  AL,AH        ; AX ← 437Bh
      DAA               ; AX ← 4381h (43 + 38 = 81)
```

Lệnh DAA chỉ ảnh hưởng đến các cờ AF, CF, PF, SF, ZF và không ảnh hưởng đến thanh ghi AH.

7.3.2. Xử lý phép trừ

❖ Lệnh **SUB dst,src**:

dst ← dst - src

Toán hạng src có thể là reg, mem hay immed còn toán hạng dst chỉ có thể là reg hay mem.

- Không thể trừ trực tiếp thanh ghi đoạn
- Ảnh hưởng đến các cờ AF, CF, OF, PF, SF và ZF.

❖ Lệnh **SBB dst,src**:

dst ← dst – src – CF

Lệnh ADC thường dùng để trừ các số lớn hơn 16 bit.

❖ Lệnh **DEC dst**: (decrement)

dst ← dst – 1

dst là reg hay mem. Lệnh DEC ảnh hưởng đến các cờ AF, OF, PF, SF, ZF.

❖ Lệnh **NEG dst**:

dst ← - *dst*

dst là reg hay mem.

Lệnh NEG ảnh hưởng đến các cờ:

CF = 1 nếu nội dung kết quả là số khác 0.

SF = 1 nếu nội dung kết quả là số âm khác 0.

PF = 1 nếu tổng 8 bit thấp là một số chẵn.

ZF = 1 nếu nội dung kết quả là 0.

OF = 1 nếu nội dung toán hạng dst là 80h (dạng byte) hay 8000h (dạng word).

VD: Nếu muốn thực hiện phép toán 100 – AH, ta không thể cùng lệnh:

```
SUB   100,AH
```

mà phải dùng lệnh:

```
SUB   AH,100
NEG   AH
```

❖ Lệnh **AAS**: (Ascii Adjust for Substract)

Hiệu chỉnh kết quả phép trừ 2 số BCD dạng không nén (mỗi chữ số BCD lưu bằng 1 byte). Lệnh AAS chỉ ảnh hưởng cờ AF và CF.

❖ Lệnh **DAS**: (Decimal Adjust for Substract)

Hiệu chỉnh kết quả phép trừ 2 số BCD dạng nén (mỗi chữ số BCD lưu bằng 4 bit). Lệnh AAS chỉ ảnh hưởng cờ AF và CF.

7.3.3. Xử lý phép nhân

❖ Lệnh **MUL src**:

Nếu src là reg hay mem 8 bit: AX ← AL*src

Nếu src là reg hay mem 16 bit: DX:AX ← AX*src

Lệnh MUL chỉ ảnh hưởng đến cờ CF và OF.

❖ Lệnh **IMUL src**:

Giống như lệnh MUL nhưng kết quả là số có dấu.

❖ Lệnh **AAM**: (Ascii Adjust for Multiple)

Hiệu chỉnh kết quả phép nhân 2 số BCD dạng không nén, lệnh AAM thực hiện chia AL cho 10, lưu phần thương vào AL và phần dư vào AH. Lệnh AAM ảnh hưởng đến các cờ PF, SF và ZF.

7.3.4. Xử lý phép chia

❖ Lệnh **DIV src**:

Nếu src là reg/mem 8 bit: AL ← AX DIV src và AH ← AX MOD src

Nếu src là reg/mem 16 bit: AX ← DX:AX DIV src và DX ← DX:AX MOD src

Lệnh DIV không ảnh hưởng đến các cờ nhưng xảy ra tràn trong các trường hợp sau:

- Chia cho 0

- Thương lớn hơn 256 đối với dạng 8 bit.
- Thương lớn hơn 65536 đối với dạng 16 bit.

❖ Lệnh **IDIV src**:

Giống như lệnh DIV nhưng kết quả là số có dấu. Các trường hợp tràn:

- Chia cho 0
- Thương nằm ngoài khoảng (-128,127) đối với dạng 8 bit.
- Thương nằm ngoài khoảng (-32767,32768) đối với dạng 16 bit.

❖ Lệnh **AAD**: (Ascii Adjust for Division)

Hiệu chỉnh kết quả phép chia 2 số BCD dạng không nén. Lưu ý rằng lệnh AAD phải được thực hiện trước lệnh chia. Sau khi thực hiện chia thì phải hiệu chỉnh lại dạng BCD bằng cách dùng lệnh AAM.

❖ Lệnh **CBW**: (Convert Byte to Word)

Nếu AL < 80h thì AH = 0, ngược lại AH = 0FFh

Lệnh CBW dùng để chuyển số nhị phân có dấu 8 bit thành số nhị phân có dấu 16 bit.

❖ Lệnh **CWD**: (Convert Word to Double word)

Nếu AX < 8000h thì DX = 0, ngược lại DX = 0FFFFh

Lệnh CWD dùng để chuyển số nhị phân có dấu 16 bit thành số nhị phân có dấu 32 bit chứa trong DX:AX.

7.3.5. Dịch chuyển và quay

❖ Lệnh **SHL**: (Shift Logical Left)

SHL dst,1
SHL dst,CL

Dịch trái 1 bit hay CL bit.

CF ← dst7 ← dst6 … ← dst0 ← 0

❖ Lệnh **SHR**: (Shift Logical Right)

SHR dst,1
SHR dst,CL

Dịch phải 1 bit hay CL bit.

0 → dst7 → dst6 … → dst0 → CF

❖ Lệnh **SAL**: giống SHL

❖ Lệnh **SAR**:

Giống như lệnh SHR nhưng giá trị bit dst7 không thay đổi, nghĩa là

dst7 → dst7 → dst6 … → dst0 → CF

❖ Lệnh **ROL**: (Rotate Left)

ROL dst,1
ROL dst,CL

Quay trái 1 bit hay CL bit.

CF ← dst7 ← dst6 … ← dst0 ← dst7

❖ Lệnh **ROR**: (Rotate Right)
ROR dst,1
ROR dst,CL
Quay phải 1 bit hay CL bit.
dst0 → dst7 → dst6 … → dst0 → CF

❖ Lệnh **RCL**: (Rotate though Carry Left)
RCL dst,1
RCL dst,CL
Quay trái 1 bit hay CL bit.
CF ← dst7 ← dst6 … ← dst0 ← CF

❖ Lệnh **RCR**: (Rotate though Carry Right)
RCR dst,1
RCR dst,CL
Quay phải 1 bit hay CL bit.
CF → dst7 → dst6 … → dst0 → CF

7.3.6. Các lệnh logic

❖ Lệnh **AND**:
AND dst,src
dst ← dst AND src
CF ← 0, OF ← 0
Src là reg, mem hay immed còn dst là reg, mem.

❖ Lệnh **OR**:
OR dst,src
dst ← dst OR src
CF ← 0, OF ← 0

❖ Lệnh **XOR**:
XOR dst,src
dst ← dst XOR src
CF ← 0, OF ← 0

❖ Lệnh **NOT**:
NOT dst
dst ← NOT dst
Lệnh NOT không ảnh hưởng đến các cờ.

❖ Lệnh **TEST**:
TEST dst,src
Lệnh TEST thực hiện phép toán AND 2 toán hạng nhưng chỉ ảnh hưởng đến các cờ và không ảnh hưởng đến toán tử.

7.4. Nhóm lệnh xử lý chuỗi

Bao gồm các lệnh sau:

- Lệnh **MOVS**: chuyển dữ liệu từ vùng nhớ này sang vùng nhớ khác.

+ MOVSB: chuyển 1 byte từ vị trí chỉ đến bởi SI đến vị trí chỉ bởi DI. Nếu DF = 0 thì SI ← SI + 1, DI ← DI + 1 còn nếu DF = 1 thì SI ← SI - 1, DI ← DI - 1.

+ MOVSW: chuyển 1 word từ vị trí chỉ đến bởi SI đến vị trí chỉ bởi DI. Nếu DF = 0 thì SI ← SI + 2, DI ← DI + 2 còn nếu DF ← 1 thì SI ← SI - 2, DI ← DI - 2.

- Lệnh **CMPS**: so sánh nội dung 2 vùng nhớ

+ CMPSB: so sánh 1 byte tại vị trí chỉ đến bởi SI và tại vị trí chỉ bởi DI. Nếu DF = 0 thì SI ← SI + 1, DI ← DI + 1 còn nếu DF ← 1 thì SI ← SI - 1, DI ← DI - 1.

+ CMPSW: so sánh 1 word tại vị trí chỉ đến bởi SI và tại vị trí chỉ bởi DI. Nếu DF = 0 thì SI ← SI + 2, DI ← DI + 2 còn nếu DF = 1 thì SI ← SI - 2, DI ← DI - 2.

- Lệnh **SCAS**: tìm một phần tử trong vùng nhớ, địa chỉ vùng nhớ xác định bằng cặp thanh ghi ES:DI, giá trị cần tìm đặt trong thanh ghi AL, nếu tìm thấy thì ZF = 1. Giá trị của DI và SI thay đổi giống như trên.

- Lệnh **LODS**: đưa một byte hay word có địa chỉ xác định bởi cặp thanh ghi DS:SI vào thanh ghi AL hay AX. Giá trị của DI và SI thay đổi giống như trên.

- Lệnh **STOS**: chuyển nội dung của AL hay AX vào vùng nhớ xác định bởi cặp thanh ghi ES:DI. Giá trị của DI và SI thay đổi giống như trên.

8. Các cấu trúc cơ bản trong lập trình hợp ngữ

8.1. Cấu trúc tuần tự

Cấu trúc tuần tự là cấu trúc đơn giản nhất. Trong cấu trúc tuần tự, các lệnh được sắp xếp tuần tự, lệnh này tiếp theo lệnh kia.

Lệnh 1
Lệnh 2
…
Lệnh n

VD: Cộng 2 giá trị của thanh ghi BX và CX, rồi nhân đôi kết quả, kết quả cuối cùng chứa trong AX

```
MOV  AX,BX
ADD  AX,CX      ; Cộng BX với CX
SHL  AX,1       ; Nhân đôi
```


8.2. Cấu trúc IF – THEN, IF – THEN – ELSE

IF Điều kiện THEN Công việc
IF Điều kiện THEN Công việc1 ELSE Công việc2
VD: Gán BX = |AX|

```
            CMP  AX,0        ; AX > 0?
            JNL  DUONG       ; AX dương
            NEG  AX          ; Nếu AX < 0 thì đảo dấu
DUONG:      MOV  BX,AX
NEXT:
```

VD: Gán CL giá trị bit dấu của AX

```
            CMP  AX,0        ; AX > 0?
            JNS  AM          ; AX âm
            MOV  CL,1        ; CL = 1 (AX dương)
            JMP  NEXT
        AM: MOV  CL,0        ; CL = 0 (AX âm)
        NEXT:
```

8.3. Cấu trúc CASE

CASE Biểu thức
Giá trị 1: Công việc 1
Giá trị 2: Công việc 2
…
Giá trị n: Công việc n
END

VD: Nếu AX > 0 thì BH = 0, nếu AX < 0 thì BH = 1. Ngược lại BH = 2

```
        CMP  AX,0
             JL   AM
        JE   KHONG
             JG   DUONG
DUONG:  MOV  BH,0
        JMP  NEXT
AM:     MOV  BH,1
        JMP  NEXT
KHONG:  MOV  BH,2
NEXT:
```

8.4. Cấu trúc FOR

FOR Số lần lặp DO Công việc

VD: Cho vùng nhớ M dài 200 bytes trong đoạn dữ liệu, chương trình đếm số chữ A trong vùng nhớ M như sau:

```
            MOV  CX,200          ; Đếm 200 bytes
            MOV  BX,OFFSET M     ; Lấy địa chỉ vùng nhớ
            XOR  AX,AX           ; AX = 0
```


```
NEXT:   CMP  BYTE PTR [BX],'A'; So sánh với chữ A
        JNZ  ChuA              ; Nếu không phải là chữ A thì tiếp
        INC  AX                ; tục, ngược lại thì tăng AX
ChuA:   INC  BX
        LOOP       NEXT
```

8.5. Cấu trúc lặp WHILE

WHILE Điều kiện DO Công việc

VD: Chương trình đọc vùng nhớ bắt đầu tại địa chỉ 1000h vào thanh ghi AH, đến khi gặp ký tự '$' thì thoát:

```
        MOV  BX,1000h
CONT:   CMP  AH,'$'
        JZ   NEXT
        MOV  AH,DS:[BX]
        JMP  CONT
NEXT:
```

8.6. Cấu trúc lặp REPEAT

REPEAT Công việc UNTIL Điều kiện

VD: Chương trình đọc vùng nhớ bắt đầu tại địa chỉ 1000h vào thanh ghi AH, đến khi gặp ký tự '$' thì thoát:

```
        MOV  BX,1000h
CONT:   MOV  AH,DS:[BX]
        CMP  AH,'$'
        JZ   NEXT
        JMP  CONT
NEXT:
```

9. Các ngắt của 8086

Bảng 3.4:

Vector ngắt	Công dụng
00h	CPU: tác động khi chia cho 0
01h	CPU: chương trình thực thi từng bước
02h	CPU: ngắt không che được
03h	CPU: tạo điểm dừng chochương trình
04h	CPU: tác động khi kết quả số học tràn
05h	Tác động khi nhấn Print Screen
06h - 07h	Dành riêng
08h	Tác động bởi nhịp đồng hồ (18.2 lần/s)
09h	Tác động khi có phím nhấn
0Ah	Dành riêng
0Bh - 0Ch	Tác động phần cứng liên lạc nối tiếp

0Dh	Đĩa cứng
0Eh	Đĩa mềm
0Fh	Máy in
10h	BIOS: màn hình
11h	BIOS: xác định cấu hình máy tính
12h	BIOS: thông báo kích thước RAM
13h	BIOS: gọi các phục vụ đĩa cứng/mềm
14h	BIOS: giao tiếp nối tiếp
15h	BIOS: truy xuất cassette hay mở rộng ngắt
16h	BIOS: xuất / nhập bàn phím
17h	BIOS: máy in
18h	Xâm nhập ROM basic
19h	BIOS: khởi động máy tính
1Ah	BIOS: ngày / giờ hệ thống
1Bh	Lấy điều khiển từ ngắt bàn phím
1Ch	Lấy điều khiển từ ngắt đồng hồ (sau int 08h)
1Dh	Địa chỉ bảng tham số màn hình
1Eh	Địa chỉ bảng tham số đĩa
1Fh	Địa chỉ bộ mã ký tự
20h	DOS: kết thúc chương trình
21h	DOS: các chức năng DOS
22h	Địa chỉ cần chuyển khi kết thúc chương trình
23h	Địa chỉ cần chuyển khi gặp Ctrl – Break
24h	Địa chỉ cần chuyển khi gặp lỗi
25h	DOS: đọc đĩa cứng / mềm
26h	DOS: ghi đĩa cứng / mềm
27h	DOS: chấm dứt chương trình và thường trú
28h – 3Fh	Dành riêng cho DOS
40h	BIOS: các chức năng đĩa mềm
41h	Bảng thông số đĩa cứng thứ nhất
42h – 45h	Dành riêng
46h	Bảng thông số đĩa cứng thứ hai
47h – 49h	Định nghĩa do người sử dụng
4Ah	Giờ báo hiệu (chỉ trong AT)
4Bh – 67h	Định nghĩa do người sử dụng
68h – 6Fh	Không sử dụng
70h	Đồng hồ thời gian thực (chỉ trong AT)
71h – 7Fh	Dành riêng
80h – 85h	Dành riêng
86h – F0h	Sử dụng bởi chương trình thông dịch BASIC
F1h – FFh	Không sử dụng

9.1. Ngắt 21h

❖ **Hàm 01h**: nhập một ký tự từ bàn phím và hiện ký tự nhập ra màn hình. Nếu không có ký tự nhập, hàm 01h sẽ đợi cho đến khi nhập.

- Gọi: AH = 01h
- Trả về: AL chứa mã ASCII của ký tự nhập

```
MOV  AH,01h
INT  21h          ; AL chứa mã ASCII của ký tự nhập
```

❖ **Hàm 02h**: xuất một ký tự trong thanh ghi DL ra màn hình tại vị trí con trỏ hiện hành

- Gọi AH = 02h, DL = mã ASCII của ký tự
- Trả về: không có

```
MOV  AH,02h
MOV  DL,'A'
INT  21h
```

❖ **Hàm 08h**: giống hàm 01h nhưng không hiển thị ký tự ra màn hình

❖ **Hàm 09h**: xuất một chuỗi ký tự ra màn hình tại vị trí con trỏ hiện hành, địa chỉ chuỗi được chứa trong DS:DX và phải được kết thúc bằng ký tự $

- Gọi AH = 09h, DS:DX = địa chỉ chuỗi
- Trả về: không có

```
.DATA
Msg  DB   'Hello$'
…
MOV  AH,09h
LEA  DX,Msg
INT  21h
```

❖ **Hàm 0Ah**: nhập một chuỗi ký tự từ bàn phím (tối đa 255 ký tự), dùng phím ENTER kết thúc chuỗi

- Gọi AH = 0Ah, DS:DX = địa chỉ lưu chuỗi
- Trả về: không có

Chuỗi phải có dạng sau:

- Byte 0: Số byte tối đa cần đọc (kể cả ký tự Enter)
- Byte 1: số byte đã đọc
- Byte 2: lưu các ký tự đọc

```
.DATA
Msg  DB   101          ; Đọc tối đa 100 ký tự
     DB   ?
     DB   101  DUP(?)
     …
     MOV  AH,0Ah
     LEA  DX,Msg
     INT  21h
```

❖ **Hàm 4Ch**: kết thúc chương trình

```
MOV  AH,4Ch
INT  21h
```


9.2. Ngắt 10h

❖ **Xoá màn hình:**

- Gọi AX = 02h
- Trả về: không có

```
MOV  AX,02h
INT  10h
```

❖ **Chuyển toạ độ con trỏ:**

- Gọi AH = 02h, DH = dòng, DL = cột

```
MOV  AH,02h
MOV  DX,0F15h
INT  10h
```

10. Truyền tham số giữa các chương trình

Trong lập trình, một vấn đề ta cần quan tâm là truyền tham số giữa chương trình chính và chương trình con. Để thực hiện truyền tham số, ta có thể dùng các cách sau đây:

- Truyền tham số qua thanh ghi
- Truyền tham số qua ô nhớ (biến)
- Truyền tham số qua ô nhớ do thanh ghi chỉ đến
- Truyền tham số qua stack

10.1. Truyền tham số qua thanh ghi

Ta thực hiện truyền tham số qua thanh ghi bằng cách: một chương trình con sẽ đưa giá trị vào thanh ghi và chương trình con khác sẽ xử lý giá trị trên thanh ghi đó.

VD: Cộng giá trị tại 2 ô nhớ 1000h và 1001h, kết quả chứa trong 1002h (bye cao) và 1003h (byte thấp).

```
.MODEL    SMALL
.STACK    100h
.CODE
main  PROC
      MOV       AX,@DATA
      MOV       DS,AX
      MOV       BYTE PTR DS:[1000h],10h      ; Đưa giá trị vào
      MOV       BYTE PTR DS:[1001h],0FFh     ; các ô nhớ
      CALL      Read
      CALL      Sum
      Mov       AH,4Ch
      INT       21h
main  ENDP
Read  PROC                    ; Đọc dữ liệu vào thanh ghi AX
      MOV       AH,DS:[1000h]
      MOV       AL,DS:[1001h]
      RET
Read  ENDP                    ; Xử lý dữ liệu tại thanh ghi AX
```


```
Sum   PROC
      ADD       AH,AL
      JZ        next
      MOV       DS:[1003h],1
next: MOV       DS:[1002h],AH
RET
Sum   ENDP
END   main
```

10.2. Truyền tham số qua ô nhớ (biến)

Quá trình truyền tham số cũng giống như trên nhưng thay vì thực hiện thông qua thanh ghi, ta sẽ thực hiện thông qua các ô nhớ.

VD: Cộng giá trị tại 2 ô nhớ m1 và m2, kết quả chứa trong m3 (bye cao) và m4 (byte thấp).

```
.MODEL    SMALL
.STACK    100h
.DATA
      m1    db    ?
      m2    db    ?
      m3    db    ?
      m4    db    ?
.CODE
main  PROC
      MOV       AX,@data
      MOV       DS,AX
      MOV       m1,10h      ; Đưa giá trị vào
      MOV       m2,0FFh     ; các ô nhớ
      CALL      Sum
      MOV       AH,4Ch
      INT       21h
main  ENDP
Sum   PROC
      MOV       m4,0
      MOV       AH,m1
      ADD       AH,m2
      JNC       next
      MOV       m4,1
next: MOV       m3,AH
RET
Sum   ENDP
END   main
```

10.3. Truyền tham số qua ô nhớ do thanh ghi chỉ đến

Trong cách truyền tham số này, ta dùng các thanh ghi SI, DI, BX để chỉ địa chỉ offset của các tham số còn thanh ghi đoạn mặc định là DS.

VD: Cộng giá trị tại 2 ô nhớ m1 và m2, kết quả chứa trong m3 (bye cao) và m4 (byte thấp).

```
.MODEL    SMALL
.STACK    100h
.DATA
        m1    db    ?
        m2    db    ?
        m3    db    ?
        m4    db    ?
.CODE
main    PROC
        MOV       AX,@data
        MOV       DS,AX
        LEA       SI,m1
        LEA       DI,m2
        LEA       BX,m3
        MOV       [SI],10h      ; Đưa giá trị vào
        MOV       [DI],0FFh     ; các ô nhớ
        CALL      Sum
        MOV       AH,4Ch
        INT       21h
main    ENDP
Sum     PROC
        MOV       AL,[SI]
        ADD       AL,[DI]
        JZ        next
        MOV       [BX+1],1
next:   MOV       [BX],AL
RET
Sum     ENDP
END     main
```

10.4. Truyền tham số qua stack

Trong phương pháp truyền tham số này, ta dùng stack làm nơi chứa các tham số cần truyền thông qua các tác vụ PUSH và POP.

VD: Cộng giá trị tại 2 ô nhớ m1 và m2, kết quả chứa trong m3 (byte cao) và m4 (byte thấp).

```
.MODEL    SMALL
.STACK    100h
.DATA
        m1    dw    ?
        m2    dw    ?
        m3    dw    ?
        m4    dw    ?
.CODE
main    PROC
        MOV       AX,@data
```


```
        MOV     DS,AX
        LEA     SI,m1
        LEA     DI,m2
        MOV     [SI],1234h          ; Đưa giá trị vào
        MOV     [DI],0FEDCh         ; các ô nhớ
        PUSH    m1                  ; Đưa vào stack
        PUSH    m2
        CALL    Sum
        POP     m3                  ; Lấy kết quả đưa vào stack
        POP     m4
        MOV     AH,4Ch
        INT     21h
main    ENDP
Sum     PROC
        POP     DX    ; Lưu lại địa chỉ trả về của lệnh CALL
        POP     AX    ; Lấy dữ liệu từ stack
        POP     BX
        ADD     AX,BX
        JNC     next
        PUSH    1
next:   PUSH    AX
        PUSH    DX    ; Trả lại địa chỉ trở về của lệnh CALL
RET
Sum     ENDP
END     main
```

11. Các ví dụ minh hoạ

11.1. In chuỗi ký tự ra màn hình

```
.MODEL  SMALL
.STACK  100h
.DATA
        msg     DB      'Hello$'
.CODE
main    PROC
        MOV     AX,@DATA        ; Khởi động thanh ghi DS
        MOV     DS,AX
        MOV     AX,02h          ; Xoá màn hình
        INT     10h
        MOV     AH,02h          ; Chuyển toạ độ con trỏ
        MOV     DX,0C15h        ; đến dòng 12 (0Ch) và cột 21 (15h)
        INT     10h
        LEA     DX,msg          ; Địa chỉ thông điệp
        MOV     AH,09h          ; In thông điệp ra màn hình
        INT     21h
        MOV     AH,4Ch          ; Kết thúc chương trình
        INT     21h
main    ENDP
```


```
END   main
```

11.2. In chuỗi ký tự ra màn hình tại toạ độ nhập vào

```
.MODEL      SMALL
.STACK      100h
.DATA
        msg     DB      'Hello$'
        msg1    DB      'Nhap vao toa do:$'
        Crlf    DB      0Dh,0Ah,'$'
        Td      DB      3
                DB      ?
                DB      3       DUP(?)
.CODE
main    PROC
        MOV     AX,@DATA
        MOV     DS,AX               ; Khởi động thanh ghi DS
        MOV     AX,02h
        INT     10h                 ; Xóa màn hình
        LEA     DX,msg1
        MOV     AH,09h              ; In thông điệp
        INT     21h
        CALL    Nhap                ; Nhập dòng
        MOV     CL,AL
        LEA     DX,Crlf             ; Xuống dòng
        MOV     AH,09h
        INT     21h
        CALL    Nhap                ; Nhập cột
        MOV     CH,AL
        MOV     AH,02h              ; Chuyển tọa độ con trỏ
        MOV     DX,CX
        INT     10h
        LEA     DX,msg
        MOV     AH,09h              ; In ra màn hình
        INT     21h
        MOV     AH,4Ch              ; Kết thúc chương trình
        INT     21h
main    ENDP
Nhap    PROC
        MOV     AH,0Ah              ; Nhập vào
        LEA     DX,Td
        INT     21h
        LEA     BX,Td               ; Lấy chữ số hàng chục
        MOV     AL,DS:[BX+2]
        SUB     AL,'0'              ; Chuyển từ dạng ký tự sang dạng số
        MOV     BL,10
        MUL     BL                  ; Nhân số hàng chục với 10
        PUSH    AX
        LEA     BX,Td               ; Lấy chữ số hàng đơn vị
```


```
        MOV  AL,DS:[BX+3]
        SUB  AL,'0'
        POP  BX
        ADD  AL,BL
        RET
Nhap    ENDP
END     main
```

11.3. Cộng 2 số nhị phân dài 5 byte

```
.MODEL      SMALL
.STACK      100h
.DATA
        m1      DB      00h,08h,10h,13h,24h,00h
        m2      DB      0FFh,0FCh,0FAh,0F0h,0F1h,00h;
        m3      DB      6       DUP(0)
.CODE
main    PROC
        MOV  AX,@DATA
        MOV  DS,AX              ; Khởi động thanh ghi DS
        LEA  SI,m1
        LEA  DI,m2
        LEA  BX,m3
        MOV  CX,6
        XOR  AL,AL
next:   MOV  AL,[SI]
        ADC  AL,[DI]
        MOV  [BX],AL
        INC  BX
        INC  SI
        INC  DI
        LOOP next
        MOV  AH,4Ch
        INT  21h
main    ENDP
END     main
```

11.4. Nhập một chuỗi ký tự và chuyển chữ thường thành chữ hoa

```
.MODEL      SMALL
.STACK      100h
.DATA
        m1      DB      81
                DB      ?
                DB      81      DUP(?)
        m2      DB      'Chuoi da doi:$'
.CODE
main    PROC
        MOV  AX,@DATA
```


```
        MOV  DS,AX          ; Khởi động thanh ghi DS
        MOV  ES,AX
        LEA  DX,m1
        MOV  AH,0Ah         ; Nhập chuỗi
        INT  21h
        LEA  SI,m1          ; Lấy địa chỉ chuỗi
        ADD  SI,2
        MOV  DI,SI          ; Chuỗi nguồn và đích trùng nhau
Next:   LODSB               ; Lấy ký tự
        CMP  AL,0Dh         ; Nếu là ký tự Enter thì kết thúc
        JE   quit
        CMP  AL,'a'         ; Nếu ký tự nhập không phải là ký tự
        JB   cont           ; thường từ 'a' đến 'z' thì bỏ qua
        CMP  AL,'z'
        JA   cont
        SUB  AL,20h         ; Chuyển ký tự thường thành ký tự hoa
        STOSB               ; Lưu ký tự vừa chuyển
        DEC  DI             ; Nếu là ký tự thường thì dùng lệnh STOSB
                            ; nên DI tăng lên 1, ta phải giảm DI
cont:   INC  DI             ; Tăng lên ký tự kế
        JMP  next
quit:   MOV  AL,'$'
        STOSB
        MOV  AX,02h         ; Xóa màn hình
        INT  10h
        LEA  DX,m2
        MOV  AH,09h
        INT  21h
        LEA  DX,m1+2
        MOV  AH,09h
        INT  21h
        MOV  AH,4Ch
        INT  21h
main    ENDP
END     main
```