

TRƯỜNG ĐẠI HỌC CẦN THƠ
KHOA CÔNG NGHỆ
BỘ MÔN ĐIỆN TỬ - VIỄN THÔNG



Giáo trình

LẬP TRÌNH HỆ THỐNG

Biên soạn:

ThS. Nguyễn Hứa Duy Khang
Ks. Trần Hữu Danh

-ĐHCT-
Tháng 5-2008

NỘI DUNG

| | |
|---|-----------|
| Chương 1 - TỔ CHỨC BỘ XỬ LÝ INTEL-8086..... | 1 |
| 1.1. Bộ xử lý Intel-8086 (CPU-8086)..... | 1 |
| 1.1.1. Cấu trúc tổng quát | 1 |
| 1.1.2. Các thanh ghi của 8086 | 2 |
| 1.1.3. Trạng thái tràn: | 5 |
| 1.2. Bộ nhớ trong của Intel-80x86..... | 5 |
| 1.2.1. Tổ chức dữ liệu..... | 5 |
| 1.2.2. Sự phân đoạn bộ nhớ..... | 6 |
| 1.3. Địa chỉ các ngoại vi | 7 |
| 1.4. Các bộ xử lý Intel khác..... | 8 |
| 1.4.1. Bộ xử lý Intel-80386..... | 8 |
| 1.4.2. Tập thanh ghi của bộ xử lý Intel-80386: | 8 |
| 1.4.3. Các chế độ vận hành của bộ xử lý Intel-80386 | 9 |
| 1.4.4. Bộ xử lý Intel-80486: | 10 |
| 1.4.5. Bộ xử lý Intel PENTIUM: | 11 |
| BÀI TẬP CHƯƠNG 1 | 13 |
| Chương 2 - HỢP NGỮ'..... | 15 |
| 2.1. Ngôn ngữ máy và hợp ngữ | 15 |
| 2.2. Đặc tính tổng quát của hợp ngữ..... | 16 |
| 2.2.1. Cấu trúc của một dòng lệnh hợp ngữ. | 16 |
| 2.2.2. Macro..... | 17 |
| 2.2.3. Chương trình con..... | 17 |
| 2.2.4. Biến toàn cục (global), biến địa phương (local)..... | 18 |
| 2.2.5. Các bảng, thông báo: | 18 |
| 2.2.6. Hợp ngữ chéo (cross assembler) | 19 |
| 2.3. Hợp ngữ MASM (hợp ngữ của CPU-8086)..... | 19 |
| 2.3.1. Cấu trúc của một hàng lệnh..... | 19 |
| 2.3.2. Tên..... | 19 |
| 2.3.3. Từ gọi nhớ mã lệnh, lệnh giả..... | 20 |
| 2.3.4. Toán hạng và toán tử..... | 27 |
| 2.4. Cấu trúc của chương trình hợp ngữ MASM..... | 30 |
| 2.4.3. Tập tin thi hành dạng COM và dạng EXE | 31 |
| 2.4.4. Ví dụ | 32 |
| 2.5. Cách tạo chương trình hợp ngữ | 33 |

| | |
|--|-----------|
| Chương 3 - TẬP LỆNH CPU-8086 ĐƠN GIẢN và KIỂU ĐỊNH VỊ | 36 |
| 3.1. Tập lệnh của CPU-8086 | 36 |
| 3.1.1. Lệnh sao chép dữ liệu, địa chỉ: | 36 |
| 3.1.2. Lệnh tính toán số học. | 38 |
| 3.1.3. Nhóm lệnh logic và ghi dịch | 39 |
| 3.1.4. Nhóm lệnh vào ra ngoại vi. | 42 |
| 3.1.5. Nhóm lệnh hệ thống | 43 |
| 3.2. Kiểu định vị | 43 |
| 3.2.1. Định vị tức thì: | 44 |
| 3.2.2. Định vị thanh ghi | 44 |
| 3.1.3. Định vị trực tiếp (bộ nhớ): | 44 |
| 3.1.4. Định vị gián tiếp thanh ghi | 45 |
| 3.1.5. Định vị nền | 45 |
| 3.1.6. Định vị chỉ số | 46 |
| 3.1.7. Định vị chỉ số nền | 46 |
| 3.1.8. Định vị chuỗi | 46 |
| 3.1.9. Định vị cổng vào/ra | 47 |
| BÀI TẬP CHƯƠNG 3 | 48 |
| Chương 4 - HỆ THỐNG NGẮT MỀM | 50 |
| 4.1. Những cơ sở của ngắt mềm | 50 |
| 4.2. Sử dụng ngắt trong hợp ngữ | 50 |
| 4.3. Ngắt MS-DOS | 51 |
| 4.4 Các ví dụ | 56 |
| Chương 5 - LỆNH NHẢY VÀ VÒNG LẶP | 60 |
| 5.1. Lệnh nhảy (chuyển điều khiển) | 60 |
| 5.1.1. Lệnh nhảy không điều kiện | 60 |
| 5.1.2. Lệnh nhảy có điều kiện: | 61 |
| 5.2. Vòng lặp | 64 |
| BÀI TẬP CHƯƠNG 5 | 66 |
| Chương 6 - NGĂN XẾP VÀ CHƯƠNG TRÌNH CON | 68 |
| 6.1. Ngăn xếp | 68 |
| 6.1.1. Tổ chức và vận hành | 68 |
| 6.1.2. Truy xuất ngăn xếp | 69 |
| 6.2. Chương trình con | 70 |
| 6.2.1. Khai báo chương trình con (Thủ tục) | 70 |
| 6.2.2. Gọi thủ tục | 71 |
| 6.3. Các ví dụ | 71 |
| BÀI TẬP CHƯƠNG 6 | 75 |

| | |
|--|------------|
| Chương 7 - XỬ LÝ KÝ SỐ VÀ XỬ LÝ CHUỖI..... | 76 |
| 7.1. Xử lý ký tự..... | 76 |
| 7.1.1. Nhập xuất số nhị phân (Binary)..... | 76 |
| 7.1.2. Nhập xuất số thập lục phân (Hexa) | 77 |
| 7.2. Lệnh xử lý chuỗi..... | 78 |
| 7.2.1. Hướng xử lý chuỗi..... | 79 |
| 7.2.2. Các tiền tố lặp REP (Repeat)..... | 79 |
| 7.2.3. Lệnh Ghi vào chuỗi | 80 |
| 7.2.4. Lệnh Nạp từ chuỗi | 81 |
| 7.2.5. Lệnh di chuyển chuỗi | 81 |
| 7.2.6. Lệnh So sánh hai chuỗi..... | 83 |
| 7.2.7. Lệnh dò tìm trong chuỗi | 85 |
| BÀI TẬP CHƯƠNG 7 | 87 |
| Phụ lục 1 - Hướng Dẫn Sử Dụng Emu8086 | 88 |
| Phụ lục 2 – Tập Lệnh Intel-8086..... | 93 |
| Phụ lục 3 – Bảng mã ASCII..... | 117 |

GIỚI THIỆU MÔN HỌC

I. MỤC ĐÍCH YÊU CẦU

Môn Lập Trình Hệ Thống (CT143) cung cấp cho sinh viên những kiến thức cơ bản về lập trình hệ thống trên máy tính IBM/PC thông qua Hợp Ngữ (Assembly). Môn học này là nền tảng để tiếp thu hầu hết các môn học khác trong chương trình đào tạo. Mặt khác, nắm vững Hợp ngữ là cơ sở để phát triển các ứng dụng điều khiển thiết bị.

Học xong môn này, sinh viên phải nắm được các vấn đề sau:

- Tổ chức bộ xử lý Intel-8086
- Cấu trúc chương trình Hợp ngữ
- Tập lệnh của Intel-8086
- Hệ thống ngắt mềm trên máy tính IBM/PC
- Lệnh nhảy và vòng lặp trong Assembly
- Ngăn xếp và Thủ tục
- Xử lý số và Chuỗi

II. ĐỐI TƯỢNG MÔN HỌC

Môn học được dùng để giảng dạy cho các sinh viên sau:

- Sinh viên năm thứ 3 của các chuyên ngành Điện tử

III. NỘI DUNG CỐT LÕI

Giáo trình được cấu trúc thành 7 chương:

Chương 1: Tổ chức bộ xử lý Intel-8086

Chương 2: Hợp ngữ

Chương 3: Tập lệnh và Kiểu định vị

Chương 4: Hệ thống ngắt mềm

Chương 5: Lệnh nhảy và Vòng lặp

Chương 6: Ngăn xếp và Chương trình con

Chương 7: Xử lý số và Chuỗi

IV. KIẾN THỨC LIÊN QUAN

Để học tốt môn Lập trình Hệ thống, sinh viên cần phải có các kiến thức nền tảng sau:

- Kiến thức Kỹ thuật số.
- Kiến thức Kiến trúc máy tính
- Kiến thức Ngôn ngữ lập trình cấp cao: C, Pascal, Delphi ...
- Kỹ năng thao tác sử dụng máy tính.

V. DANH MỤC TÀI LIỆU THAM KHẢO

- [1] Nguyễn Văn Linh, Lâm Hoài Bảo, Dương Văn Hiếu, *Giáo trình Lập trình căn bản A*, Khoa Công Nghệ Thông Tin, Đại học Cần Thơ, 2005.
- [2] Nguyễn Đình Tê, Hoàng Đức Hải , *Giáo trình lý thuyết và bài tập ngôn ngữ C*; Nhà xuất bản Giáo dục, 1999.
- [3] Nguyễn Cẩn, *C – Tham khảo toàn diện*, Nhà xuất bản Đồng Nai, 1996.
- [4] Brain W. Kernighan & Dennis Ritchie, *The C Programming Language*, Prentice Hall Publisher, 1988.
- [5] Võ Văn Chín, *Bài giảng Ngôn ngữ hệ thống*, Khoa Công Nghệ Thông Tin, Đại học Cần Thơ, 1994.

TỔ CHỨC BỘ XỬ LÝ INTEL-8086

Mục đích:

- Cấu trúc bên trong CPU Intel-8086
- Tập thanh ghi
- Tổ chức bộ nhớ và dữ liệu
- Khái quát các bộ xử lý Intel khác như: 80386, 80486, Pentium

1.1. BỘ XỬ LÝ INTEL-8086 (CPU-8086)

1.1.1. Cấu trúc tổng quát

Intel-8086 là một CPU 16 bit (bus dữ liệu ngoại có 16 dây). Nó được dùng để chế tạo các máy vi tính PC-AT đầu tiên của hãng IBM vào năm 1981. Trong thực tế, hãng IBM đã dùng CPU 8088 (là một dạng của CPU 8086 với bus số liệu giao tiếp với ngoại vi là 8 bit) để chế tạo máy vi tính cá nhân đầu tiên gọi là PC-XT.

Cho đến nay CPU 8086 đã không ngừng cải tiến và đã trải qua các phiên bản 80186, 80286, 80386, 80486, Pentium (80586), Pentium Pro, Pentium MMX, Pentium II, III, 4. Các CPU trên tương thích từ trên xuống (downward compatible) nghĩa là tập lệnh của các CPU mới chế tạo gồm các tập lệnh của CPU chế tạo trước đó được bổ sung thêm nhiều lệnh mạnh khác.

Cấu trúc tổng quát của CPU-8086 có dạng như hình 1.1, gồm 2 bộ phận chính là: Bộ thực hiện lệnh và bộ phận giao tiếp bus.

1. Bộ phận thực hiện lệnh (EU):

Thi hành các tác vụ mà lệnh yêu cầu như: Kiểm soát các thanh ghi (đọc/ghi), giải mã và thi hành lệnh. Trong EU có bộ tính toán và luận lý (ALU) thực hiện được các phép toán số học và luận lý. Các thanh ghi đa dụng là các ô nhớ bên trong CPU chứa dữ liệu tương tự như ô nhớ trong bộ nhớ. Cờ cũng là một thanh ghi dùng để ghi lại trạng thái hoạt động của ALU. Thanh ghi lệnh chứa nội dung lệnh hiện tại mà CPU đang thực hiện.

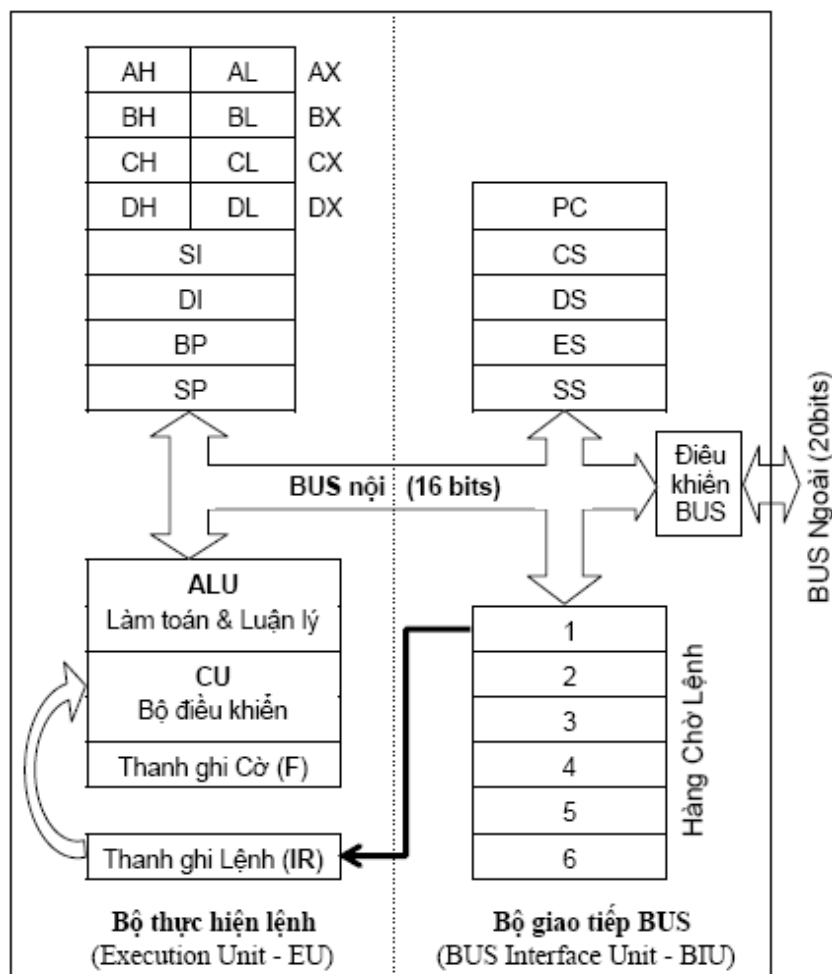
Các thanh ghi và bus trong EU đều là 16 bit. EU không kết nối trực tiếp với bus hệ thống bên ngoài. Nó lấy lệnh từ hàng chờ lệnh mà BIU cung cấp. Khi có yêu cầu truy xuất bộ nhớ hay ngoại vi thì EU yêu cầu BIU làm việc. BIU có thể tái định địa chỉ để cho phép EU truy xuất đầy đủ 1 MB (8086 có 20 đường địa chỉ ngoại).

2. Bộ giao tiếp bus (BIU):

BIU thực hiện chức năng giao tiếp giữa EU với bên ngoài (Bộ nhớ, thiết bị ngoại vi ...) thông qua hệ thống BUS ngoại (bus dữ liệu và bus địa chỉ). BIU thực hiện tất cả các tác vụ về bus mỗi khi EU có yêu cầu. Khi EU cần trao đổi dữ liệu với bên ngoài, BIU sẽ tính toán địa chỉ và truy xuất dữ liệu để phục vụ theo yêu cầu EU.

Trong BIU có 5 thanh ghi CS, DS, ES, SS và IP chứa địa chỉ. Thanh ghi IP chứa địa chỉ của lệnh sẽ được thi hành kế tiếp nên gọi là con trỏ lệnh.

EU và BIU liên lạc với nhau thông qua hệ thống bus nội. Trong khi EU đang thực hiện lệnh thì BIU lấy lệnh từ bộ nhớ nạp đầy vào hàng chờ lệnh (6 bytes). Do đó EU không phải đợi lấy lệnh từ bộ nhớ. Đây là một dạng đơn giản của cache để tăng tốc độ đọc lệnh.



Hình 1.1: Sơ đồ khối của CPU 8086

1.1.2. Các thanh ghi của 8086

Thanh ghi (register) là thành phần lưu trữ dữ liệu bên trong CPU, mỗi thanh ghi có độ dài nhất định (16 bit hoặc 8 bit) và được nhận biết bằng một tên riêng. Tùy vào độ dài và chức năng mà thanh ghi có công dụng chứa dữ liệu hoặc kết quả của phép toán, hoặc là các địa chỉ dùng để định vị bộ nhớ khi cần thiết.

Nội dung của thanh ghi được truy xuất thông qua tên riêng của nó, do đó tên thanh ghi là từ khóa quan trọng cần phải lưu ý trong lập trình.

CPU-8086 có 16 thanh ghi, mỗi thanh ghi là 16 bit, có thể chia 4 nhóm sau:

- Thanh ghi đoạn:** Gồm 4 thanh ghi 16 bit: CS, DS, ES, SS. Đây là những thanh ghi dùng để chứa địa chỉ đoạn của các ô nhớ khi cần truy xuất. Mỗi thanh ghi quản lý 1 đoạn tối đa 64K ô nhớ trong bộ nhớ trong. Người sử dụng chỉ được phép truy xuất ô nhớ dựa vào địa chỉ tương đối. CPU (cụ thể là BIU) có nhiệm vụ chuyển đổi địa chỉ tương đối thành địa chỉ tuyệt đối để truy xuất vào ô nhớ tuyệt đối tương ứng trong bộ nhớ. (Xem phần tổ chức bộ nhớ)

CS: Thanh ghi đoạn mã lệnh, lưu địa chỉ đoạn chứa mã lệnh chương trình của người sử dụng

DS: Thanh ghi đoạn dữ liệu, lưu địa chỉ đoạn chứa dữ liệu (các biến) trong chương trình.

ES: Thanh ghi đoạn dữ liệu thêm, lưu địa chỉ đoạn chứa dữ liệu thêm trong chương trình.

SS: Thanh ghi đoạn ngăn xếp, lưu địa chỉ đoạn của vùng ngăn xếp.

| | |
|-----------|--------------------|
| 15 | 0 |
| CS | Code Segment |
| DS | Data Segment |
| ES | Extra data Segment |
| SS | Stack Segment |

Thông thường bốn thanh ghi này có thể chứa những giá trị khác nhau, do đó chương trình có thể được truy cập trên bốn đoạn khác nhau và chương trình chỉ có thể truy cập cùng 1 lúc tối đa bốn đoạn. Mặc khác, đối với những chương trình nhỏ, chỉ sử dụng 1 đoạn duy nhất, khi đó cả bốn thanh ghi đều chứa cùng giá trị địa chỉ đoạn, gọi là đoạn chung.

- 2. Thanh ghi đa dụng (General Register):** Bao gồm bốn thanh ghi đa dụng 16 bit (AX, BX, CX, DX). Mỗi thanh ghi đa dụng có thể được sử dụng với nhiều mục đích khác nhau, tuy nhiên từng thanh ghi có công dụng riêng của nó.

| | | | |
|-----------|-----------|----------------------------|---|
| 15 | 8 | 7 | 0 |
| AH | AL | AX (Accumulator) | |
| BH | BL | BX (Base register) | |
| CH | CL | CX (Count register) | |
| DH | DL | DX (Data register) | |

AX : Là thanh ghi tích lũy cơ bản. Mọi tác vụ vào/ra đều dùng thanh ghi này, tác vụ dùng số liệu tức thời, một số tác vụ chuỗi ký tự và các lệnh tính toán đều dùng thanh AX.

BX: Thanh ghi nền dùng để tính toán địa chỉ ô nhớ.

CX: Là thanh ghi đếm, thường dùng để đếm số lần trong một lệnh vòng lặp hoặc lệnh xử lý chuỗi ký tự.

DX: Thanh ghi dữ liệu, thường chứa địa chỉ của một số lệnh vào/ra, lệnh tính toán số học (kể cả lệnh nhân và chia).

Mỗi thanh ghi 16 bit có thể chia đôi thành 2 thanh ghi 8 bit. Do đó, CPU-8086 có 8 thanh ghi 8 bit là: AH, AL; BH, BL; CH, CL; DH, DL (thanh ghi AH và AL tương ứng với byte cao và byte thấp của thanh ghi AX, tương tự cho các thanh ghi 8 bit còn lại).

Ví dụ: AX = 1234h => AH = 12h, AL = 34h

- 3. Thanh ghi con trỏ và chỉ số (Pointer & Index register):** Chức năng chung của nhóm thanh ghi này là chứa địa chỉ độ dời của ô nhớ trong vùng dữ liệu hay ngăn xếp.

SI : Thanh ghi chỉ số nguồn

DI : Thanh ghi chỉ số đích

BP: Thanh ghi con trỏ nền dùng để lấy số liệu từ ngăn xếp.

SP : Thanh ghi con trỏ ngăn xếp luôn chỉ vào đỉnh ngăn xếp.

| | |
|-----------|------------------------|
| 15 | 0 |
| SI | Source Index Reg. |
| DI | Destination Index Reg. |
| BP | Base Pointer Reg. |
| SP | Stack Pointer Reg. |

SI và DI chứa địa chỉ độ dời của ô nhớ tương ứng trong đoạn có địa chỉ chứa trong DS hoặc ES (*dữ liệu, còn gọi là Biến*). Còn BP và SP chứa địa chỉ độ dời của ô nhớ tương ứng trong đoạn có địa chỉ chứa trong SS, dùng để thâm nhập số liệu trong ngăn xếp.

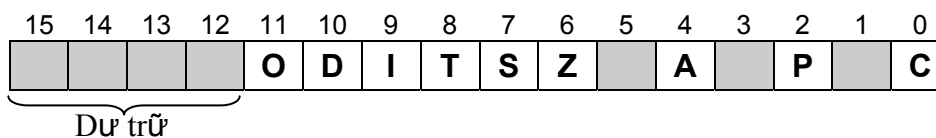
4. Thanh ghi Đếm chương trình và thanh ghi trạng thái (Cờ):

| | |
|-----------|-------------------------|
| 15 | 0 |
| F | Flag Register. |
| IP | Intruction Pointer Reg. |

- Thanh ghi con trỏ lệnh **IP** (*còn gọi là PC – đếm chương trình*) là thanh ghi 16 bit chứa địa chỉ của lệnh kế tiếp mà CPU sẽ thực hiện trong. Các lệnh của chương trình có địa chỉ đoạn trong CS.
- Thanh ghi Cờ (**F**) dài 16 bit, mỗi bit là một cờ. Mỗi cờ có giá trị 1 (*gọi là SET – Đặt*) hoặc 0 (*gọi là CLEAR – Xóa*). Hình 1.2 mô tả 9 bit trong số 16 bit tương ứng với 9 cờ trạng thái (các bit còn lại dùng cho dự trữ mở rộng khi thiết kế các CPU khác)

Thanh ghi cờ được chia thành hai nhóm:

- Nhóm cờ điều khiển (bảng 1.1) bao gồm các cờ dùng để điều khiển sự hoạt động của CPU và giá trị của cờ được thiết lập bằng các lệnh phần mềm.
- Nhóm cờ trạng thái (bảng 1.2) bao gồm các cờ phản ánh kết quả thực hiện lệnh cũng như trạng thái của CPU



Hình 1.2: Cấu trúc thanh ghi Cờ

| KÝ HIỆU | TÊN | Ý NGHĨA |
|---------------------------|-------------------|--|
| Nhóm cờ điều khiển | | |
| TF | Bẫy (Trap) | TF = 1: cho phép chương trình chạy từng bước |
| IF | Ngắt (Interrupt) | IF = 1: cho phép ngắt phần cứng |
| DF | Hướng (Direction) | DF = 1: thì SI và DI giảm 1 cho mỗi vòng lặp |

Nhóm cờ trạng thái

| | | |
|----|------------------|---|
| CF | Số giữ (Carry) | CF = 1: khi có số nhớ hoặc mượn từ MSB trong phép cộng hoặc trừ. (Có thể bị thay đổi theo lệnh ghi dịch và quay) |
| PF | Chẵn lẻ (Parity) | PF = 1: khi byte thấp của thanh ghi kết quả một phép tính có số lượng bit 1 chẵn |
| AF | Số giữ phụ (Haf) | AF = 1: khi có nhớ hoặc mượn từ bit 3 trong phép cộng hoặc trừ. Dùng trong các lệnh với số BCD |
| ZF | Zero | ZF = 1: khi kết quả của một phép tính bằng 0 |
| SF | Dấu (Sign) | SF = 1 khi kết quả phép tính là âm (MSB=1). SF = 0 khi kết quả dương (MSB=0) |
| OF | Tràn (Overflow) | OF = 1: nếu kết quả vượt quá khả năng tính toán của CPU |

Bảng 1.1: Ý nghĩa cờ

1.1.3. Trạng thái tràn:

Trạng thái tràn có thể không xảy ra (nếu không tràn) hoặc xảy ra (nếu tràn có dấu, tràn không dấu, đồng thời tràn có dấu và không dấu). Nói chung là có 2 trạng thái tràn: Tràn không dấu và Tràn có dấu.

Lưu ý: Nếu một giá trị có MSB=1 (bit dấu) thì CPU luôn luôn cho đó là số có dấu.

a. Tràn không dấu: CF=1

Ví dụ: ADD AX, BX ; với AX = 0FFFFh, BX = 1

- Nếu xem đây là các số không dấu thì AX không đủ chứa kết quả nên TRÀN không dấu, vậy CF = 1
- Nếu xem đây là các số có dấu thì kết quả sẽ là 0 (vì AX = -1) nên không tràn, do đó OF = 0

b. Tràn có dấu: OF = 1

Ví dụ: ADD AX, BX ; với AX = BX = 7FFFh = 32767

- Nếu xem đây là các số không dấu thì AX = 7FFFh + 7FFFh = 0FFFEh = 65534 nên không tràn.
- Nếu xem đây là các số có dấu thì tràn vì kết quả vượt quá phạm vi cho phép đối với số có dấu (cộng 2 số dương, kết quả lại là số âm). Thật sự trong trường hợp này, CPU sẽ làm cho OF = 1 theo qui tắc "Nhớ ra và vào MSB xảy ra không đồng thời" nghĩa là có nhớ vào MSB nhưng không có nhớ ra hoặc ngược lại thì tràn và không có hoặc có nhớ ra và vào MSB thì không tràn.

1.2. BỘ NHỚ TRONG CỬA INTEL-80x86

1.2.1. Tổ chức dữ liệu

Bộ nhớ trong được tổ chức thành mảng gồm các ô nhớ 8 bit liên tục nhau. Các dữ liệu có thể được ghi vào hoặc đọc ra (gọi là truy xuất) từ bất cứ vị trí ô nhớ nào. Mỗi ô nhớ 8 bit được phân cứng quản lý bằng một địa chỉ vật lý duy nhất. Việc truy xuất nội dung ô nhớ phải bằng địa chỉ vật lý này.

Dữ liệu 8 bit được lưu trữ bằng một ô nhớ và địa chỉ của ô nhớ chính là địa chỉ dùng để truy xuất dữ liệu. Dữ liệu nhiều hơn 8 bit được lưu trữ bởi nhiều ô

nhớ liên tục nhau. Theo quy ước Intel, byte dữ liệu cao được lưu ở ô nhớ có địa chỉ cao và byte dữ liệu thấp hơn lưu ở ô nhớ có địa chỉ thấp hơn. Khi đó, địa chỉ dùng để truy xuất dữ liệu là địa chỉ của ô nhớ thấp (*ô nhớ chứa byte thấp nhất của dữ liệu*)

Hình 1.3 mô tả việc tổ chức các dữ liệu có độ dài khác nhau trong bộ nhớ. Giá trị 5Fh (1 byte) được lưu trữ ở địa chỉ 0010h. Giá trị 0A0B1h (2 byte) được lưu trữ bởi 2 ô nhớ có địa chỉ 0015h và 0016h, địa chỉ để truy xuất giá trị này là 0015h. Còn giá trị 0A2B1C0h (3 byte) được lưu trữ bởi 3 ô nhớ 0012h, 0013h và 0014h, do đó địa chỉ truy xuất giá trị này là 0012h.

| | Bộ nhớ | Địa chỉ |
|---|--------|--------------|
| (A0h là byte cao, B1h là byte thấp) | A0h | 0016h |
| Giá trị: 0A0B1h → | B1h | 0015h |
| | A2h | 0014h |
| (C0h byte thấp nhất, A2h byte cao nhất) | B1h | 0013h |
| Giá trị: 0A2B1C0h → | C0h | 0012h |
| | | 0011h |
| Giá trị: 5Fh → | 5Fh | 0010h |

Hình 1.3: Tổ chức dữ liệu trong bộ nhớ

1.2.2. Sự phân đoạn bộ nhớ trong

CPU 8086 có không gian địa chỉ là 1MB (ứng với 20 bit địa chỉ). Vậy CPU 8086 có thể quản lý bộ nhớ trong là $2^{20} = 1\text{MB}$. Bộ nhớ 1 MB này được CPU-8086 quản lý bằng nhiều đoạn 64 KB. Các đoạn có thể tách rời hoặc có thể chồng lên nhau.

Mỗi đoạn có một địa chỉ đoạn 16 bit duy nhất, tùy vào mục đích sử dụng đoạn mà địa chỉ đoạn được lưu trữ trong thanh ghi đoạn tương ứng.

Đối với người lập trình, Địa chỉ của ô nhớ trong bộ nhớ được xác định bởi hai thông số 16 bit (gọi là địa chỉ logic): Địa chỉ Đoạn (segment) và địa chỉ độ dời (offset).

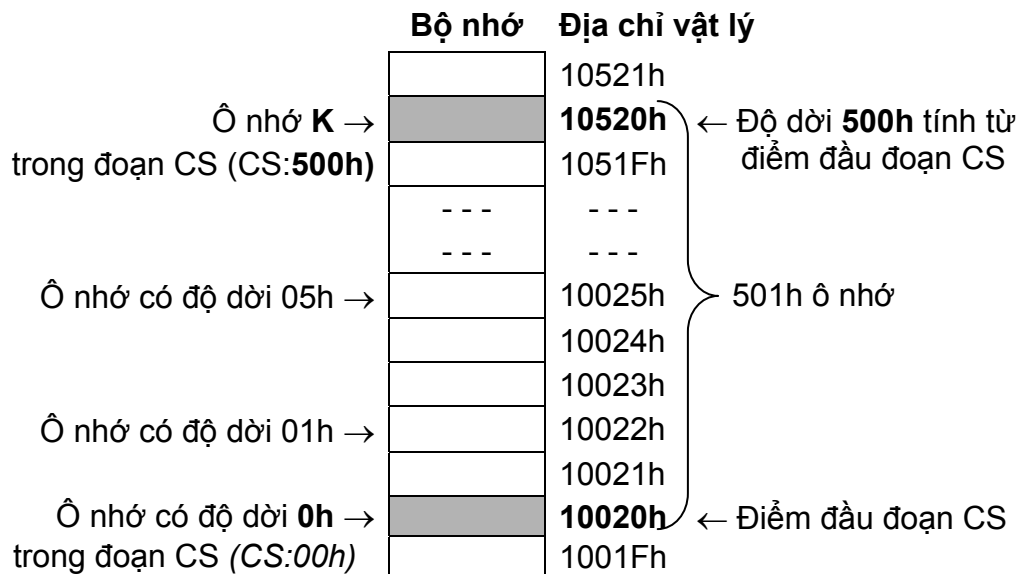
Cách viết: **Segment : Offset**

Địa chỉ vật lý của ô nhớ khi truy xuất sẽ được BIU tự động chuyển đổi từ địa chỉ logic bằng cách dịch trái thanh ghi đoạn bốn bit (*tức nhân nội dung của thanh ghi đoạn cho 16*) rồi cộng với địa chỉ độ dời. Vì vậy, người lập trình không cần địa chỉ vật lý của ô nhớ mà chỉ cần biết địa chỉ logic của ô nhớ.

Ví dụ: đoạn CS có giá trị là 1002h, địa chỉ độ dời của ô nhớ K trong đoạn CS là 500h (CS:0500h hoặc 1002h:0500h). Khi đó, địa chỉ vật lý của ô nhớ K được tính như sau:

$$\begin{array}{r}
 10020h \text{ (dịch trái địa chỉ đoạn 4 bit)} \\
 + 0500h \text{ (độ dời)} \\
 \hline
 10520h \text{ (địa chỉ vật lý)}
 \end{array}$$

Trong ví dụ trên, đoạn CS có điểm bắt đầu ở địa chỉ vật lý 10020h. Độ dời 500h là khoảng cách từ địa chỉ của điểm bắt đầu của đoạn CS đến ô nhớ K (xem hình 1.4)



Hình 1.4: Địa chỉ vật lý của ô nhớ K trong đoạn CS

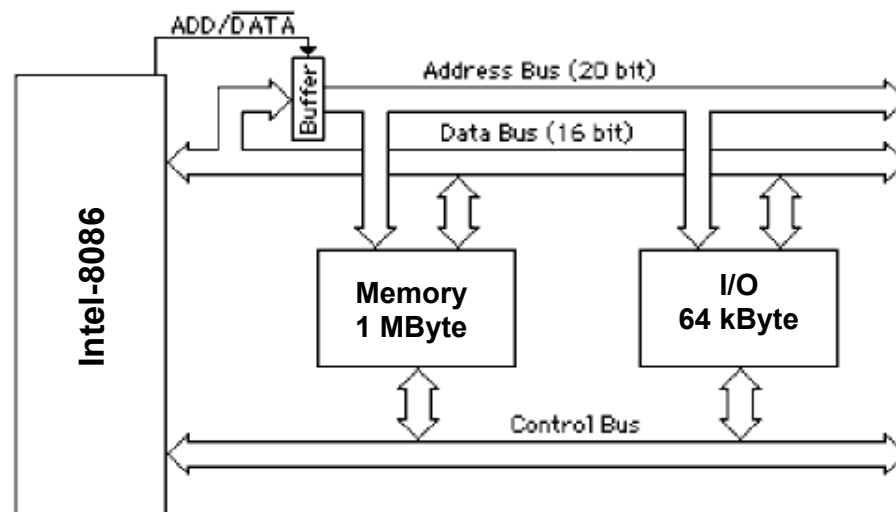
Khi truy xuất ô nhớ, BIU lấy sẽ sử dụng địa chỉ đoạn trong thanh ghi đoạn tương ứng với tính chất của ô nhớ cần truy xuất:

- Ô nhớ là Code (Mã lệnh) thì đoạn tương ứng là CS.
- Ô nhớ là Data (dữ liệu) thì đoạn tương ứng là DS.
- Ô nhớ nằm trên ngăn xếp thì dùng đoạn SS.
- Khi truy xuất chuỗi, DI và SI luôn chứa độ dài của ô nhớ trong đoạn DS hay ES

Khi khởi động, CPU 8086 nhảy đến địa chỉ vật lý cao nhất của bộ nhớ trong (đoạn CS = 0FFFFh và độ dài 0) để lấy lệnh. Địa chỉ này ứng với địa chỉ của ROM-BIOS của bộ nhớ trong nơi chứa chương trình khởi động máy tính.

1.3. Địa chỉ các ngoại vi

Các ngoại vi đều có địa chỉ riêng từ 0 đến 64K. CPU 8086 dùng các lệnh riêng biệt để truy xuất ngoại vi và bộ nhớ trong. Muốn truy xuất ngoại vi, BIU chỉ cần đưa địa chỉ của ngoại vi lên 16 bit thấp của bus địa chỉ (không có đoạn).



Hình 1.5: Cấu trúc đơn giản của máy tính

1.4. CÁC BỘ XỬ LÝ INTEL KHÁC

Ngoài CPU-8086, Intel đã cho ra đời thế hệ CPU mới hơn, nhiều tính năng hơn và mạnh hơn như: 80186, 80286, 80386, 80486, Celeron và Pentium (80586). Ngày nay, sức mạnh và tính năng của CPU Intel tăng vượt bậc nhờ công nghệ mới như: Centrino, Hyper Threading, Core Duo.

Bắt đầu từ CPU 80286, hãng Intel đã đưa vào một số cải tiến có ý nghĩa như tăng bus địa chỉ lên 24 bit và có thể vận hành với chế độ bảo vệ (protected mode). Chế độ này cho phép CPU 80286 vận hành trong một hệ điều hành đa nhiệm (multitasking).

1.4.1. Bộ xử lý 80386

Hãng Intel đã thành công lớn khi chế tạo CPU 80386. Đây vẫn còn là một CPU CISC thuần túy có bus địa chỉ 32 bit và bus số liệu 32 bit (ta gọi CPU 80386 là một CPU 32 bit).

Các thanh ghi của CPU 80386 đều là thanh ghi 32 bit (Hình 1.5). Một số thanh ghi đa dụng có thể chia thành thanh ghi 16 bit hoặc chia thành thanh ghi 8 bit để đảm bảo tính tương thích với các CPU chế tạo trước đó.

Với bus địa chỉ 32 bit, không gian địa chỉ của CPU 80386 là 4 GB tức 4096 MB. CPU 80386 có 64K cửa vào/ra 8 bit, 16 bit và 32 bit.

CPU 80386 cũng có thể hoạt động với bộ đồng xử lý toán học 80387 (math coprocessor). Bộ đồng xử lý toán học dùng xử lý các phép tính trên các số có dấu chấm động (số lẻ).

1.4.2. Tập thanh ghi của bộ xử lý 80386:

Hình 1.6 mô tả đầy đủ tập thanh ghi của CPU-80386.

Các thanh ghi đa dụng và thanh ghi con trở được mở rộng thành thanh ghi 32 bit được gọi tên là: EAX, EBX, ESP, EDI... . Tuy nhiên ta vẫn có thể sử dụng các thanh ghi 16 bit (AX, BX, ...) hoặc 8 bit (AH, AL, BH, BL ...) giống như các thanh ghi 16 bit hoặc 8 bit của bộ xử lý 8086.

Chiều dài các thanh ghi đoạn vẫn giữ nguyên 16 bit nhưng có thêm hai thanh ghi đoạn thêm là FS và GS. Các thanh ghi FS và GS được dùng giống như thanh ghi ES. Nghĩa là CPU-80386 quản lý được bốn đoạn dữ liệu.

Thanh ghi trạng thái SR (Status register) và thanh ghi đếm chương trình PC (program counter) cũng được nâng lên 32 bit. Ngoài các bit trạng thái đã thấy trong thanh ghi trạng thái của CPU-8086 (C, Z, S, ...) thanh ghi trạng thái của CPU-80386 còn có thêm các bit trạng thái như sau:

- IOP (Input/Output protection: bảo vệ vào/ra): Đây là hai bit trạng thái dùng trong chế độ bảo vệ để xác định mức ưu tiên mà một tiến trình phải có để có thể thâm nhập một vùng vào ra. Chỉ hệ điều hành mới có quyền dùng các bit này.
- N (Nested task: tiến trình lồng vào nhau): Trong chế độ bảo vệ, các hệ điều hành dùng bit này để biết có nhiều tiến trình đang vận hành và ít nhất có một tiến trình đang bị gián đoạn.
- R (Resume: tải trực): Bit này cho phép một tiến trình được tiếp tục vận hành lại sau khi bị gián đoạn.

- V (Virtual 8086 mode: chế độ 8086 ảo): Bit này cho phép 80386 đang vận hành ở chế độ bảo vệ, chuyển sang chế độ 8086 ảo.

| | 31 | 16 15 | 8 7 | 0 |
|-----|----|-------|-----|---|
| EAX | | AH | AL | |
| EBX | | BH | BL | |
| ECX | | CH | CL | |
| EDX | | DH | DL | |
| ESP | | SP | | |
| EBP | | BP | | |
| ESI | | SI | | |
| EDI | | DI | | |
| ESR | | SR | | |
| EPC | | PC | | |
| | | CS | | |
| | | DS | | |
| | | SS | | |
| | | ES | | |
| | | FS | | |
| | | GS | | |

Hình 1.6a: Thanh ghi đa dụng và thanh ghi con trỏ

| | 15 | 0 31 | 0 19 | 0 |
|------|------|------|------|---|
| TR | | | | |
| LDTR | | | | |
| | IDTR | | | |
| | GDTR | | | |

Hình 1.6b: Thanh ghi quản lý bộ nhớ

| | 31 | 16 15 | 0 |
|-----|----|-------|---|
| CR3 | | | |
| CR2 | | | |
| CR1 | | | |
| CR0 | | | |

Hình 1.6c: Thanh ghi điều khiển

| | 31 | 16 15 | 0 |
|-----|----|-------|---|
| TR7 | | | |
| TR6 | | | |

Hình 1.6d: Thanh ghi kiểm tra

| | 31 | 16 15 | 0 |
|-----|----|-------|---|
| DR7 | | | |
| DR6 | | | |
| DR5 | | | |
| DR4 | | | |
| DR3 | | | |
| DR2 | | | |
| DR1 | | | |
| DR0 | | | |

Hình 1.6e: Thanh ghi gỡ rối

Hình 1.6: Các thanh ghi của CPU 80386

1.4.3. Các chế độ vận hành của bộ xử lý 80386

CPU-80386 có thể vận hành theo một trong ba chế độ khác nhau: chế độ thực (real mode), chế độ bảo vệ (protected mode) và chế độ 8086 ảo (virtual 8086 mode). Chế độ vận hành của CPU phải được thiết lập trước bằng phần cứng.

- **Chế độ thực:** chế độ thực của bộ xử lý 80386 hoàn toàn tương thích với chế độ vận hành của bộ xử lý 8086. Trong chế độ này, không gian địa chỉ của 80386 bị giới hạn ở mức $2^{20} = 1\text{MB}$ giống như không gian địa chỉ của 8086 mặc dù bus địa chỉ của 80386 có 32 đường dây.
- **Chế độ bảo vệ:** (Còn gọi là chế độ đa nhiệm) chế độ bảo vệ đã được đầu tiên đưa vào bộ xử lý 80286. Chế độ này cho phép bộ xử lý 80386 dùng hết không gian địa chỉ của nó là $2^{32} = 4096\text{MB}$ và cho phép nó vận hành dưới một hệ điều hành đa nhiệm. Trong hệ điều hành đa nhiệm, nhiều tiến trình có thể chạy đồng thời và được bảo vệ chống lại các thâm nhập trái phép vào vùng ô nhớ bị cấm.
Trong chế độ bảo vệ, các thanh ghi đoạn không được xem như địa chỉ bắt đầu của đoạn mà là thanh ghi chọn (selector) gán các ưu tiên khác nhau cho các tiến trình. Phần ưu tiên khác nhau cho các tiến trình. Phần cốt lõi của hệ điều hành có ưu tiên cao nhất và người sử dụng có ưu tiên thấp nhất.
- **Chế độ 8086 ảo:** Chế độ này cho phép thiết lập một kiểu vận hành đa nhiệm trong đó các chương trình dùng trong chế độ thực, có thể chạy song song với các tiến trình khác.

1.4.4. Bộ xử lý Intel 80486:

CPU-80486DX được phát hành năm 1989. Đó là bộ xử lý 32bit chứa 1.2 triệu transistor. Khả năng quản lý bộ nhớ tối đa giống như 80386 nhưng tốc độ thi hành lệnh đạt được 26.9 MIPS (*Mega Instructions Per Second - triệu lệnh mỗi giây*) tại xung nhịp 33 MHz

Nếu bộ xử lý 80386 là bộ xử lý CISC thuần túy với bộ đồng xử lý toán học 80387 nằm bên ngoài bộ xử lý 80386, thì bộ xử lý 80486 là một bộ xử lý hỗn tạp CISC và RISC với bộ đồng xử lý toán học và với 8K cache nằm bên trong bộ xử lý 80486.

Trong bộ xử lý 80486, một số lệnh thường dùng, ví dụ như lệnh MOV, dùng mạch điện (kỹ thuật RISC) để thực hiện lệnh thay vì dùng vi chương trình như trong các CPU CISC thuần túy. Như thế thì các lệnh thường dùng này được thi hành với tốc độ nhanh hơn. Kỹ thuật ống dẫn cũng được đưa vào trong bộ xử lý 80486.

Với các kỹ thuật RISC được đưa vào, bộ xử lý 80486 nhanh hơn bộ xử lý 80386 đến 3 lần (nếu tốc độ xung nhịp là như nhau).

Bộ xử lý 80486 hoàn toàn tương thích với 2 bộ xử lý 80386 và 80387 cộng lại và như thế nó có các chế độ vận hành giống như 80386.

Bộ xử lý 80486 tỏ ra rất mạnh đối với các chương trình cần tính toán nhiều và các chương trình đồ họa, vì bộ đồng xử lý toán học nằm ngay trong bộ xử lý 80486. Hàng chờ lệnh của bộ xử lý 80486 là 32 byte.



Hình 1.7: CPU-80386



Hình 1.8: CPU-80486

1.4.5. Bộ xử lý Intel PENTIUM:

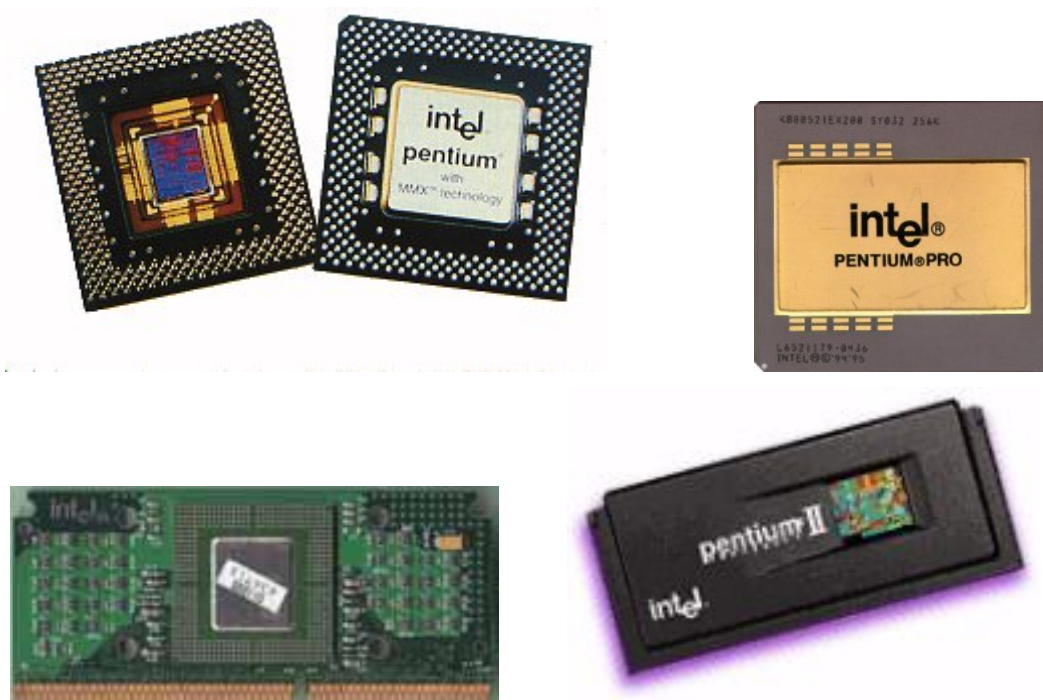
Được đưa ra thị trường vào giữa năm 1993, các bộ xử lý Pentium được chế tạo với hơn 3 triệu transistor, thụ hưởng các tính năng kỹ thuật của các bộ xử lý 80486 DX/2 và được có thêm nhiều tính năng mới.

Theo hãng sản xuất Intel, nếu hoạt động ở với xung nhịp 66 MHz thì tốc độ thi hành lệnh của Pentium là 112 MIPS thay vì 54 MIPS cho bộ xử lý 80486 DX/2 cùng xung nhịp.

Các tính năng nổi bật của bộ xử lý Pentium là :

- Hoàn toàn tương thích với các bộ xử lý được chế tạo trước đó (8086, 80286, 80386, 80486).
- Dùng kỹ thuật ống dẫn tốt hơn với 2 ống dẫn số nguyên độc lập và một ống dẫn số lẻ. Kỹ thuật siêu vô hướng và tiên đoán lệnh nhảy cũng được đưa vào.
- Dùng cache lệnh và cache dữ liệu riêng biệt.
- Bus số liệu là 64 bit với cách vận chuyển theo từng gói.
- Dùng cách quản lý hệ thống cho phép tiết kiệm năng lượng tiêu hao trong bộ xử lý Pentium.
- Có tối ưu hóa các chuỗi mã lệnh.

Pentium MMX: Các lệnh xử lý multimedia được đưa vào tập lệnh của CPU nên việc thi hành chương trình multimedia được cải thiện rất nhiều.



Pentium Celeron



Hình 1.9: Hình ảnh các loại bộ xử lý Intel

BÀI TẬP CHƯƠNG 1

1. Chọn các thanh ghi đa dụng để lưu trữ các dữ liệu sau đây, sao cho mỗi thanh ghi lưu trữ 1 giá trị và không trùng nhau (Giải thích việc chọn thanh ghi): 15h, 0AFh, 01234h, 230, 257, 'H', 8086.

Ghi chú: Số có tận cùng bằng h (hay H) là số thập lục phân (Hexa);
'H' : Ký tự H

2. Các thanh ghi đang lưu trữ giá trị như sau:

| | | |
|----------|-------------|-----------|
| AH = 11h | AL = 22h | CL = 15 |
| CH = 10 | BX = 0A1D4h | DX = 8086 |

Hãy cho biết giá trị thập lục phân của những thanh ghi sau và giải thích:

AX, CX, BH, BL, DH, DL

3. Mô tả các cách có thể sử dụng được để lưu trữ giá trị vào thanh ghi sau:

- | | |
|---------------------------|---------------------------|
| a. 1234h vào thanh ghi SI | b. 5678h vào thanh ghi AX |
| c. 100 vào thanh ghi DI | d. 100 vào thanh ghi DX |

4. Sử dụng mô hình bộ nhớ gồm 17 ô nhớ như hình A1 để ghi các dữ liệu sau đây vào bộ nhớ sao cho các giá trị không chồng lên nhau (sinh viên tự chọn địa chỉ ô nhớ để lưu trữ): 15h, 0AFh, 01234h, 230, 257, 'H', 8086, 3A4B5h, 0F1D2E3h

| Ô nhớ | Địa chỉ |
|-------|---------|
| | 00010h |
| | |
| | 00002h |
| | 00001h |
| | 00000h |

Hình A1: Mô hình bộ nhớ

5. Với mô hình bộ nhớ kết quả của câu 4, hãy cho biết giá trị dữ liệu 8/16/24/32 bit tại mỗi địa chỉ sau đây ở dạng thập lục phân:

- | | | | |
|-----------|-----------|-----------|-----------|
| a. 00004h | b. 00008h | c. 0000Bh | d. 0000Dh |
|-----------|-----------|-----------|-----------|

6. Đổi sang địa chỉ vật lý tương ứng với mỗi địa chỉ logic sau:

- | | |
|---------------|---------------|
| a. 0000:0001h | b. 0100:1234h |
| c. ABCD:3AFFh | d. AF70:00CFh |

7. Viết ra 4 địa chỉ logic khác nhau đối với mỗi địa chỉ vật lý sau:

- | | |
|-----------|------------|
| a. 40000h | b. 0D32FCh |
|-----------|------------|

8. Tìm tất cả các địa chỉ logic khác nhau có thể có của mỗi ô nhớ có địa chỉ vật lý sau đây:

- | | | | |
|-----------|-----------|-----------|-----------|
| a. 00000h | b. 0000Fh | c. 00010h | d. 0001Fh |
|-----------|-----------|-----------|-----------|

Có nhận xét gì về các kết quả trên?

9. Cho một chương trình bao gồm 100 byte lệnh (Code), 200 byte dữ liệu (Data) và 16 KB ngăn xếp (Stack). Vẽ hình mô tả tổ chức vùng nhớ của chương trình trên

trong bộ nhớ có mô hình như hình A2 theo hai cách sau (sinh viên tự gán địa chỉ vật lý và logic thích hợp với đầu và cuối cho từng vùng):

- Dùng chung 1 đoạn duy nhất cho cả 3 vùng Code, Data và Stack
- Dùng 3 đoạn riêng biệt không chồng nhau cho Code, Data, Stack

| Địa chỉ logic | Ô nhớ | Địa chỉ vật lý |
|---------------|-------|----------------|
| | | |
| | | |
| | | |
| | | |
| | | |

Hình A2: Mô hình tổ chức bộ nhớ

10. Bằng mô hình bộ nhớ kết quả của câu 9a, hãy thiết lập giá trị các thanh ghi sao cho CPU-8086 truy xuất được những ô nhớ trong mỗi trường hợp sau:

- Byte lệnh đầu tiên
- Byte lệnh thứ 20
- Byte dữ liệu đầu tiên
- Byte dữ liệu thứ 10

Chương 2

HỢP NGỮ

Mục Đích

- So sánh ngôn ngữ máy và hợp ngữ
- Lệnh giả của MASM
- Các cấu trúc chương trình Hợp ngữ
- Cách tạo chương trình hợp ngữ

2.1. NGÔN NGỮ MÁY VÀ HỢP NGỮ

Chương trình là một tập hợp các lệnh được đưa vào bộ nhớ trong của máy tính để bộ xử lý thực hiện. Các lệnh có thể được thể hiện ở những dạng (ngôn ngữ) khác nhau. Bộ xử lý hoạt động dựa trên kỹ thuật số nên chỉ hiểu được những giá trị nhị phân (*để đơn giản, giá trị nhị phân còn được viết thành giá trị thập lục phân*). Do đó, ngôn ngữ mà CPU hiểu được ấy gọi là **ngôn ngữ máy** (Machine language). Mỗi bộ xử lý có tập giá trị phân trong đó mỗi giá trị nhị phân sẽ điều khiển CPU thực hiện một tác vụ (gọi là mã lệnh), tập hợp các mã lệnh nhị phân ấy gọi là tập lệnh. Những bộ xử lý khác nhau sẽ có tập lệnh khác nhau, do đó ngôn ngữ máy của CPU nào thì chỉ thực hiện được trên CPU đó. Bảng 2.1 trình bày đoạn chương trình ngôn ngữ máy viết cho họ Intel-8086.

| Thứ tự lệnh | Giá trị thập lục phân | Mã lệnh nhị phân |
|-------------|-----------------------|----------------------------|
| 1 | B4 09 | 10110100 00001001 |
| 2 | BA 03 01 | 10111010 00000011 00000001 |
| 3 | 40 | 01000000 |

Bảng 2.1: Đoạn chương trình ngôn ngữ máy họ Intel-80x86

Đối với CPU có kiến trúc CISC, chiều dài các lệnh có thể khác nhau do đó trong đoạn chương trình trên gồm 3 lệnh, có chiều dài lần lượt là 2 byte, 3 byte và 1 byte. Byte đầu tiên của mỗi lệnh là Tác vụ (Op-Code) mà CPU phải thực hiện còn các byte còn lại là Tác tử (Operand) xác định dữ liệu hoặc nơi chứa dữ liệu mà lệnh tác động vào. Để có thể lập trình với ngôn ngữ máy này, người lập trình phải hiểu rõ tổ chức phần cứng của máy đang sử dụng.

Vì là ngôn ngữ riêng của CPU và được CPU thực hiện ngay khi đọc được lệnh nên chương trình viết bằng ngôn ngữ máy thực hiện rất nhanh và chiếm ít bộ nhớ trong. Tuy nhiên, dạng nhị phân của ngôn ngữ máy rất khó nhớ dễ nhầm lẫn nên khó viết.

Để khắc phục nhược điểm khó nhớ của dạng nhị phân, người ta dùng những từ ngữ dễ nhớ để thay thế cho những mã lệnh nhị phân (gọi là từ gợi nhớ mã lệnh – mnemonic). Chương trình viết bằng Từ gợi nhớ mã lệnh gọi là **Hợp Ngữ** (Assembly).

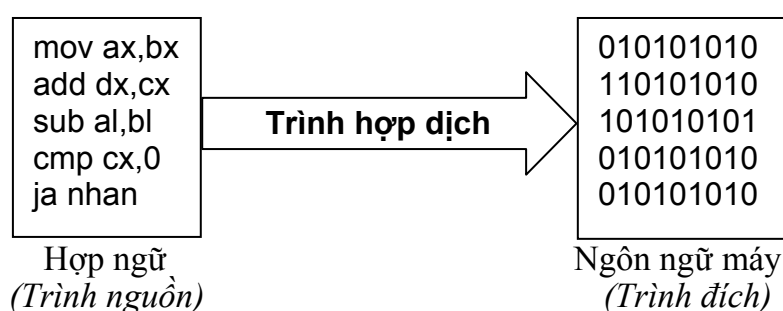
Mỗi mã lệnh nhị phân sẽ có một lệnh tương ứng trong Hợp ngữ và mỗi CPU khác nhau sẽ có Hợp ngữ khác nhau. Các tài liệu tham khảo về tập lệnh của CPU thường cung cấp cho lập trình viên tập lệnh của CPU ở dạng Hợp Ngữ.

Bảng 2.2 viết lại đoạn chương trình trong bảng 2.1 ở dạng Hợp ngữ. Qua đó người đọc có thể hiểu được phần nào chức năng của mỗi dòng lệnh trong đoạn chương trình đó thông qua lệnh Hợp ngữ.

| Thứ tự lệnh | Giá trị thập lục phân | Hợp Ngữ | Ý nghĩa |
|-------------|-----------------------|---------------|-------------|
| 1 | B4 09 | MOV AH, 09h | AH ← 09h |
| 2 | BA 03 01 | MOV DX, 0103h | DX ← 0103h |
| 3 | 40 | INC AX | AX ← AX + 1 |

Bảng 2.1: Đoạn chương trình Hợp ngữ cho họ Intel-80x86

Như vậy, Hợp ngữ sẽ làm cho lập trình viên dễ dàng hơn trong việc viết chương trình nhưng CPU thì không hiểu được những từ gọi nhớ mã lệnh đó. Do đó, từng lệnh trong chương trình viết bằng hợp ngữ phải được dịch sang ngôn ngữ máy tương ứng (hình 2.1). Chương trình làm nhiệm vụ này gọi là Trình hợp dịch (Assembler)



Hình 2.1: Hợp ngữ và Ngôn ngữ máy

Khi mới ra đời, các chương trình hợp ngữ chỉ làm việc đơn thuần là dịch những từ gọi nhớ mã lệnh sang mã máy nhưng dần dần các trình hợp dịch bổ sung thêm các lệnh riêng của trình hợp dịch cho phép dùng các nhãn, các ký hiệu, biến đổi dạng lệnh, khai báo biến, phân phối bộ nhớ, macro... nhằm giúp cho người lập trình viết chương trình hợp ngữ dễ dàng hơn, các lệnh riêng đó gọi là lệnh giả (pseudo code).

Các lệnh giả không có mã máy nhị phân tương ứng nên bản thân các lệnh giả không điều khiển CPU thực hiện tác vụ mà chỉ dùng để hướng dẫn trình hợp dịch trong quá trình dịch chương trình. Do đó, khi gặp từ gọi nhớ mã lệnh, trình hợp dịch sẽ dịch sang mã máy nhị phân; còn khi gặp lệnh giả, trình hợp dịch không dịch thành mã nhị phân.

2.2. ĐẶC TÍNH TỔNG QUÁT CỦA HỢP NGỮ

2.2.1. Cấu trúc của một dòng lệnh hợp ngữ.

Đối với hợp ngữ, trên một dòng văn bản chỉ cho phép viết một dòng lệnh duy nhất. Hình 2.2 trình bày bốn thành phần có thể có trong cấu trúc của dòng lệnh hợp ngữ.

| Tên | Mã lệnh | Toán hạng | Chú thích |
|-----|---------|-----------|-----------|
|-----|---------|-----------|-----------|

Hình 2.2: Cấu trúc dòng lệnh hợp ngữ

Bốn thành phần của dòng lệnh hợp ngữ được sắp xếp theo thứ tự trên một hàng lệnh và được phân cách nhau bằng một ký tự đặc biệt gọi là ký tự ngăn cách (delimiter). Các ký tự ngăn cách có thể gồm: Khoảng trống, hai chấm, chấm phẩy, dấu

phẩy, dấu hỏi... tùy theo hợp ngữ. Chính các ký tự ngăn cách này giúp cho trình hợp dịch phân biệt được các vùng khác nhau trong dòng lệnh. Vì vậy, khi khảo sát một hợp ngữ nào đó, chúng ta phải chú ý và sử dụng đúng các ký tự ngăn cách.

1. Tên (name)

Vùng tên cho phép gán tên cho một địa chỉ (nhãn) hay một dữ liệu (Biến, hằng). Khi đó ta có thể sử dụng tên này để thay thế địa chỉ hay dữ liệu khi tham khảo đến địa chỉ hay dữ liệu ấy. Mỗi tên chỉ được thay thế cho duy nhất một địa chỉ hoặc một trong chương trình và được đặt theo quy cách của trình hợp dịch.

2. Mã lệnh

Đây là vùng duy nhất không thể thiếu được của hàng lệnh, có thể chứa một từ gọi nhớ mã lệnh hoặc một lệnh giả.

- Từ gọi nhớ mã lệnh được trình bày ở chương 3 (Tập lệnh CPU-8086).
- Mỗi trình hợp dịch có thể xây dựng tập lệnh giả riêng, nên trong phần 2.3 sẽ trình bày các lệnh giả thường dùng của trình hợp dịch MASM.

3. Toán hạng: Chứa các toán hạng mà lệnh cần. Số toán hạng tùy vào mã lệnh

4. Chú thích: Dùng để ghi chú, giải thích cho dòng lệnh.

2.2.2. Macro

Macro là một nhóm lệnh nào đó được dùng 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.

2.2.3. 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 (linker) 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 gần xếp và làm chậm đi việc thực hiện chương trình chính.

Đối với macro, thì mỗi lần chương trình chính gọi macro để thực hiện thì toàn bộ nhóm lệnh trong macro được xen vào ngay điểm gọi, không cần dùng lệnh CALL hay JUMP, nên chương trình chính được thực hiện nhanh hơn.

Tuy nhiên mỗi lần gọi macro thì đoạn mã lệnh trong macro được xen vào chương trình chính làm cho chương trình chính ngày càng dài ra và chiếm nhiều ô nhớ trong hơn.

Chương trình con chỉ chiếm một vùng ô nhớ nhất định. Khi chương trình chính gọi nó thì chương trình chính lưu trữ địa chỉ trở về, nhảy đến địa chỉ bắt đầu của chương trình con để thực hiện chương trình này và khi thực hiện xong thì lấy lại địa chỉ trở về để tiếp tục thực hiện chương trình chính. Dùng chương trình con ít chiếm bộ nhớ trong.

| <i>Sử dụng Chương trình con</i> | | <i>Sử dụng Macro</i> | |
|---------------------------------|--------------------------------------|---|-------------------|
| CT con A → | Lệnh 1 Lệnh 2 Lệnh 3 Lệnh 4 | Lệnh 1 Lệnh 2 Lệnh 3 Lệnh 4 | ← Macro A |
| CT chính → | Lệnh i | Lệnh i | ← CT chính |
| Gọi CT con A → | CALL A | -Lệnh 1 -Lệnh 2 -Lệnh 3 -Lệnh 4 | ← Gọi MacroA |
| Gọi CTC A → | CALL A | Lệnh ii -Lệnh 1 -Lệnh 2 -Lệnh 3 -Lệnh 4 | ← Gọi MacroA |
| | Lệnh iii | Lệnh iii | Dài hơn!!! |

Hình 2.3: So sánh chương trình dùng Chương trình con và Macro

Trước đây, do bộ nhớ trong của máy tính còn hạn hẹp và tốc độ chưa cao nên việc dùng macro hay chương trình con được xem xét rất cẩn thận đảm bảo được chương trình ngắn và hoạt động không chậm. Ngày nay, dung lượng bộ nhớ trong khá lớn và với tốc độ nhanh của bộ xử lý đã làm cho người lập trình thoải mái hơn trong việc sử dụng macro hay chương trình con.

2.2.4. 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 có thể dùng cho chương trình chính, trong macro và các chương trình con.

Các biến được khai báo bên trong macro hay chương trình con được gọi là biến địa phương và chỉ được dùng cho nội bộ một macro hoặc chương trình con. Nói cách khác, nếu có 2 biến cùng tên được khai báo ở 2 chương trình con khác nhau là 2 biến hoàn toàn khác nhau.

Việc hiểu biết cặn kẽ về biến toàn cục và địa phương giúp cho người lập trình quản lý biến và sử dụng biến hiệu quả hơn, tránh trường hợp không bị trùng lặp khi sử dụng tên biến trong chương trình lớn.

2.2.5. Các bảng, thông báo:

Đa số các trình hợp ngữ khi tiến hành hợp dịch có thể cung cấp các bảng và thông báo cho người lập trình. Các bảng thông báo được cung cấp dưới dạng tập tin văn bản, bao gồm các bảng như sau:

- Liệt kê chương trình hợp ngữ và mã máy tương ứng.
- Liệt kê các lỗi trong chương trình gốc.
- Liệt kê các tên được dùng trong chương trình gốc.
- Danh sách các tham khảo ở chương trình khác, bên ngoài chương trình (chương trình con, biến dùng ở bên ngoài)
- Các macro, chương trình con và độ dài của chúng

2.2.6. 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 tính 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.

2.3. HỢP NGỮ MASM (HỢP NGỮ CỦA CPU-8086)

MASM (Microsoft Macro Assembler) là trình hợp ngữ do hãng phần mềm Microsoft phát triển cùng với phiên bản hệ điều hành DOS. Ngoài ra, hãng Borland – chuyên xây dựng các chương trình dịch – cũng phát hành chương trình dịch hợp ngữ gọi là TASM (Turbo Assembler). Các phiên bản này chỉ cho phép viết chương trình trên hệ điều hành DOS, cho nên không viết được chương trình cho Windows bằng các phiên bản này. Hiện nay, đã có các phiên bản cho phép viết chương trình trên Windows, như MASM32 (MASM 32 bits).

Ta dùng các từ gợi nhớ mã lệnh, các lệnh giả, các ký hiệu ... do trình hợp ngữ MASM qui định để viết ra một chương trình mà ta gọi là chương trình hợp ngữ nguồn (source file). Chương trình nguồn được lưu trữ trong đĩa từ dưới một tên tập tin có đuôi là ASM. MASM sẽ dịch chương trình nguồn thành chương trình đối tượng có đuôi OBJ (Object file). Chương trình đối tượng sẽ được liên kết (LINK) thành chương trình có thể chạy được có đuôi là EXE (Executive file).

Các loại hợp ngữ khác nhau đều có cú pháp giống nhau, nhưng cũng có những đặc điểm riêng thể hiện khả năng mạnh mẽ của nó. Sau đây chúng ta nghiên cứu các qui định, các cú pháp thường dùng cho việc viết một chương trình hợp ngữ.

2.3.1. Cấu trúc của một hàng lệnh :

Một hàng lệnh gồm có bốn thành phần:

| Tên | Từ gợi nhớ mã lệnh hoặc lệnh giả | Toán hạng hoặc biểu thức | Ghi chú |
|-----|----------------------------------|--------------------------|---------|
|-----|----------------------------------|--------------------------|---------|

Hình 2.4: Cấu trúc dòng lệnh hợp ngữ MASM

Ví dụ:

BDAU: MOV DX, 3F8h ; 3F8H là địa chỉ cổng vào ra nối tiếp

Trong đó: BDAU – Tên với vai trò là nhãn

: – (Dấu 2 chấm) phân cách Tên và Từ gợi nhớ

MOV – Từ gợi nhớ mã lệnh

DX, 3F8h – Hai toán hạng, phân cách bằng dấu phẩy

3F8H là địa chỉ cổng vào ra nối tiếp – Câu chú thích

; – (Dấu ;) Phân cách Toán hạng và Chú thích

2.3.2. Tên:

Tên có thể là nhãn, biến hay ký hiệu (hằng).

Tên có chiều dài tối đa là 31 ký tự và phải bắt đầu bằng một chữ cái. Các khoảng trống và ký hiệu toán học không được dùng để đặt tên.

- 1. NHÃN:** Nhãn dùng để đánh dấu một địa chỉ mà các lệnh như: lệnh nhảy, gọi chương trình con, lệnh lập cần đến. Nó cũng được dùng cho các lệnh giả LABEL hoặc PROC hoặc EXTRN

Ví dụ:

NH: MOV AX, DS ; NH là nhãn 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 nhãn ngoài chương trình gốc này

- 2. 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: **TWO DB 2** ; khai báo biến TWO có giá trị là 2

- 3. KÝ HIỆU:** Ký hiệu là một tên được định nghĩa để thay cho một biểu thức, một từ gọi nhớ lệnh. Ký hiệu có thể dùng làm toán hạng trong biểu thức, trong lệnh hay trong lệnh giả.

Ví dụ: **FOO EQU 7h** ; ký hiệu FOO thay thế giá trị 07h

TOTO = 0Fh ; ký hiệu TOTO thay thế giá trị 0Fh

2.3.3. Từ gọi nhớ mã lệnh, lệnh giả:

Từ gọi nhớ mã lệnh sẽ được trình bày trong các chương sau về tập lệnh của bộ xử lý 8086. Trong phần này sẽ trình bày các lệnh giả thường dùng của MASM.

Lệnh giả có thể được chia thành 5 nhóm: cấu trúc chương trình, khai báo dữ liệu, dịch có điều kiện, Macro và Liệt kê

A. Nhóm cấu trúc chương trình:

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 chẵn.

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 kết hợp phân đoạn này với phân đoạn khác:

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

Ví dụ: Khai báo hai đoạn có 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 một đoạn.

Cú pháp: ASSUME <SegReg>: <Tên1>, <SegReg>: <Tên2>

Ví dụ: Chỉ định đoạn có tên DATA là đoạn dữ liệu (DS) và đoạn có tên CODE là đoạn lệnh (CS):

```
ASSUME DS: DATA, CS: CODE
```

Lưu ý: Lệnh ASSUME chỉ được viết trong đoạn lệnh.

Ví dụ: Báo cho hợp ngữ biết là không có đoạn nào được chỉ định loại. Khi đó, mỗi lần liên hệ đến một nhân (biến) phải dùng cả địa chỉ đoạn của chúng:

```
ASSUME NOTHING
```

COMMENT: ghi chú chương trình. Có thể viết trên nhiều dòng

Cú pháp: COMMENT * <ghi chú> *

EVEN : Làm cho thanh ghi đếm chương trình PC là giá trị chẵn

EXTRN: Cho biết một tên hay một ký hiệu đã được định nghĩa bên ngoài ở một module khác được sử dụng ở module chương trình hiện tại.

END: Điểm cuối chương trình nguồn và xác định điểm bắt đầu

Cú pháp: END <Nhãn>

Ví dụ: END begin ; điểm bắt đầu chạy CT là nhãn begin

GROUP: Nhóm các đoạn khác nhau có một tên nhóm dùng chung

Ví dụ: Nhóm 3 đoạn có tên DATA1, DATA2 và DATA3 có tên nhóm dùng chung là CGROUP:

```
CGROUP GROUP DATA1,DATA2,DATA3
```

INCLUDE : xen một tập tin hợp ngữ khác vào tập tin hiện hành

Ví dụ: Xen tập tin THEM.ASM trên ổ đĩa C:\ASM vào tập tin hiện hành ngay tại vị trí của lệnh giả INCLUDE:

INCLUDE C:\ASM\THEM.ASM

LABEL : đánh dấu một địa chỉ là địa chỉ của lệnh hay số liệu kế

Ví dụ:

NHF LABEL FAR ; nhả xa, đánh dấu vị trí NH của lệnh kế

NH: MOV AX, DATA

CH DW 100 DUP (0) ; chuỗi từng từ

NAME : đặt tên cho một module hợp ngữ

Ví dụ: NAME Cursor ; đặt tên cho modul là cursor

ORG: ấn định địa chỉ bắt đầu cho đoạn chương trình

Ví dụ: ORG 100h

MOV AX,code ; Lệnh này được đặt tại địa chỉ CS:100h

PROC và **ENDP**: Khai báo chương trình con

Cú pháp: <Tên CTC> **PROC** [Near/Far]

<Tên CTC> **ENDP**

Ví dụ:

CTCON PROC Near

MOV AX, 10 ; Bắt đầu chương trình con

ADD DX, AX

RET ; Trở về CT chính

CTCON ENDP

PUBLIC : khai báo các tên trong module hiện hành mà các module khác có thể sử dụng.

Ví dụ : PUBLIC FOO, NH, TOTO

B. Nhóm khai báo dữ liệu:

Tất cả dữ liệu trong chương trình hợp ngữ được trình hợp dịch chuyển sang dạng nhị phân. Do đó giá trị của dữ liệu có thể được viết ở các dạng như: ký tự, thập phân, nhị phân, bát phân hay thập lục phân.

B.1. Cách viết số:

- Hệ thập phân: 10, 150, 1234
- Hệ bát phân: kết thúc bằng ký tự *O* (hay *o*), như: 10o, 35O, 1230O
- Hệ thập lục phân: phải kết thúc bằng ký tự *H* (hoặc *h*) và bắt đầu là số, như: 10h, 13H, 2Fh, 0D4E1h

- Hệ nhị phân: kết thúc bằng ký tự B (hoặc b), như: 10b, 01100100B

B.2. Chuỗi ký tự

Ký tự hay chuỗi ký tự phải được kẹp giữa cặp dấu nháy đơn (‘) hoặc cặp dấu nháy kép (“), như: ‘A’, “b”, ‘Hello Assembly’, “This is a string”

Trình hợp dịch chuyển ký tự sang dạng nhị phân tương ứng với mã ASCII, do đó ‘A’, “A”, 41h hay 65 đều có nghĩa như nhau khi lưu trữ.

B.3. Định nghĩa dữ liệu (Khai báo biến)

Cú pháp chung: [Tên Biến] <Loại Biến> <Giá trị>

[Tên biến] đặt theo quy cách của Tên.

Tên biến (tên vùng nhớ), thực chất là địa chỉ tượng trưng của vùng nhớ và được chuyển thành địa chỉ thật sau khi dịch chương trình.

<Loại Biến> xác định kích thước của biến theo byte, bao gồm:

DB (Define Byte): khai báo biến 1 byte (1 ô nhớ)

DW (Define Word): khai báo biến 2 byte (1 từ máy tính)

DD (Define Double word): khai báo biến 4 byte (từ đôi)

DQ (Define Quad word): khai báo biến 8 byte (bốn từ).

DT (Define Ten byte): khai báo biến 10 byte.

<Giá trị> bao gồm các dạng:

- Số hay biểu thức: 01100b, 0A1D3h, 15 hay $(50+10h)*9$
- Ký tự hay chuỗi: ‘A’, ‘CHAO BAN’
- Rỗng (không gán trước giá trị): ?
- Mảng có n giá trị khác nhau: <trị 1>, <trị 2>, ..., <trị n>
- Mảng có X giá trị giống nhau: X DUP(<giá trị>)

Lưu ý: - Độ lớn của Trị không vượt quá khả năng lưu trữ của vùng nhớ đã định nghĩa bằng <Loại biến>

- Biến chuỗi ký tự phải khai báo bằng DB

Các ví dụ:

- Khai báo biến 1 byte, tên là SO và gán trước trị là 14:

SO DB 14

1 byte nhớ →

| |
|-----|
| SO |
| 0Eh |

 (Tên biến/ địa chỉ ô nhớ)
(Trị của Biến/ nội dung ô nhớ)

- Khai báo mảng M gồm 5 phần tử có trị lần lượt 1, 3, 5, 7, 9:

M DB 1, 3, 5, 7, 9

| M | M+1 | M+2 | M+3 | M+4 |
|-----|-----|-----|-----|-----|
| 01h | 03h | 05h | 07h | 09h |

- Khai báo biến 2 byte, lưu trữ giá trị 10, tên là WA:

WA DW 10 ; 10 = 000Ah

| WA | WA+1 | WB | WB+1 |
|-----|------|-----|------|
| 0Ah | 00h | 34h | 12h |

- Khai báo biến lưu trữ giá trị 1234h, đặt tên là WB:

WB DW 1234h

- Khai báo biến, lưu trữ chuỗi "Hello", tên là str:

str DB 'Hello' ; Lưu trong bộ nhớ bằng các mã ASCII

| str | str+1 | str+2 | str+3 | str+4 |
|-----|-------|-------|-------|-------|
| 48h | 65h | 6Ch | 6Ch | 6Fh |

- Khai báo biến mảng Str2 lưu trữ lần lượt các trị 'ABC', 0Ah, 0Dh, '\$':

Str2 DB 'ABC', 0Ah, 0Dh, '\$' ; Mảng 6 phần tử 1 byte

| Str2 | Str2 +1 | Str2 +2 | Str2 +3 | Str2 +4 | Str2 +5 |
|------|---------|---------|---------|---------|---------|
| 41h | 42h | 43h | 0Ah | 0Dh | 24h |

- Khai báo biến mảng 7 phần tử có cùng giá trị là '\$', tên là Str3:

Str3 DB 7 DUP('\$') ; Mảng 7 phần tử 1 byte

| Str3 | Str3+1 | Str3+2 | Str3+3 | Str3+4 | Str3+5 | Str3+6 |
|------|--------|--------|--------|--------|--------|--------|
| 24h | 24h | 24h | 24h | 24h | 24h | 24h |

Trong hợp ngữ, kiểu của biến được hiểu đơn giản hơn thông qua số lượng ô nhớ của biến và được thể hiện bằng <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 lần lượt từ ô nhớ có địa thấp đến cao. Trong một đoạn có nhiều biến được khai báo, biến được khai báo đầu tiên sẽ có địa chỉ độ dời trong đoạn dữ liệu bắt đầu là 0h và tiếp theo cho các biến được khai báo tiếp theo sau.

B.4. Dữ liệu có cấu trúc:

RECORD và **STRUC**: Khai báo biến kiểu có cấu trúc mẫu tin.

Cú pháp: [Tên Biến] <Kiểu cấu trúc> [trường: d] [...]

d: Số nguyên dương, xác định độ lớn của trường.

<Kiểu cấu trúc>:

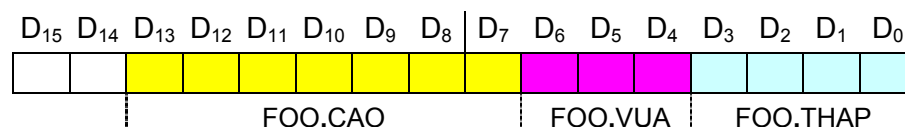
RECORD: d tính bằng bit. Cấu trúc dài tối đa là 16 bit

STRUC: d tính bằng byte.

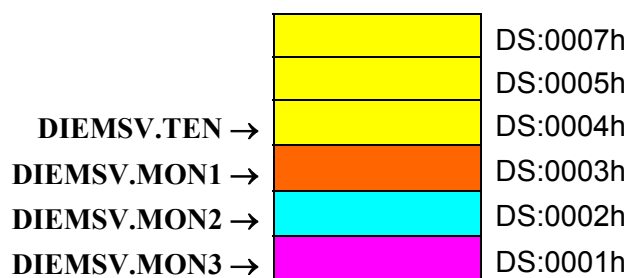
Ví dụ: Hình 2.5 mô tả quản lý vùng nhớ cho 2 biến có cấu trúc sau:

FOO RECORD CAO : 7, VUA : 3, THAP : 4

DIEMSV STRUC TEN: 3, MON1: 1, MON2: 1, MON3:1



Hình 2.5a: Mô hình quản lý biến FOO



Hình 2.5b: Mô hình quản lý biến DIEMSV

B.5. Khai báo hằng

EQU và **=** : gán trị cho 1 ký hiệu.

EQU để gán trị cho ký hiệu (hằng số) chỉ 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

C. Nhóm lệnh giả về dịch (compile) có điều kiện:

Các lệnh giả về dịch có điều kiện nhằm bảo 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)

Cú pháp chung: <Điều kiện> ; Điều kiện dịch chương trình

Nhóm lệnh A

[ELSE]

Nhóm lệnh B

ENDIF ; Hết điều kiện dịch

<Điều kiện> có thể có những hình thức sau:

IFE <B-Thức> : Nếu <B-Thức> = 0 thì nhóm lệnh A được dịch.

Nếu <B-Thức> ≠ 0 thì nhóm lệnh B được dịch (*nếu có lệnh giả ELSE*)

IF1 : Nếu đang dịch lần 1 thì nhóm lệnh A được dịch

IF2 : Nếu đang dịch lần 2 thì nhóm lệnh A được dịch

IFDEF <ký hiệu>: Nếu ký hiệu đã định nghĩa thì dịch nhóm lệnh A.

IFNDEF <ký hiệu>: Nếu ký hiệu không được định nghĩa thì dịch nhóm lệnh A

IFB <ĐốiSố> : Nếu đối số là khoảng trống hoặc không có đối số thì dịch nhóm lệnh A.

IFNB <ĐốiSố> : Nếu có đối số thì dịch nhóm lệnh A

IFIDN <ĐốiSố1>, <ĐốiSố2>: Nếu ĐốiSố1=ĐốiSố2 thì dịch nhóm lệnh A

IFDIF <ĐốiSố1>,<ĐốiSố2>: Nếu ĐốiSố1 ≠ ĐốiSố2 thì dịch nhóm lệnh A

D. Nhóm lệnh giả về MACRO

Cú pháp: <Tên> MACRO [tham số]
 ; đoạn chương trình
 ENDM ; chấm dứt Macro

Ví dụ : Viết MACRO tên **gen** với 3 tham số vào X, Y, Z như sau :

```
gen  MACRO    X, Y, Z
      MOV      AX, X      ; AX ← X
      ADD      AX, Y      ; AX ← AX + Y
      ADD      AX, Z      ; AX ← AX + Z
      ENDM
```

Trong chương trình, gọi macro **gen** bằng lệnh:

```
gen 10, 20, 30      ; X=10, Y=20, Z=30
```

Một đoạn chương trình sau đây được xen vào ngay lệnh gọi macro:

```
MOV      AX, 10      ; AX ← 10
ADD      AX, 20      ; AX ← AX + 20
ADD      AX, 30      ; AX ← AX + 30
```

Ngoài ra, trong Macro ta có thể dùng lệnh giả LOCAL, EXITM:

EXITM : Thoát ra khỏi MACRO trước khi gặp lệnh ENDM.

LOCAL: Định nghĩa các nhãn địa phương trong các MACRO khi chương trình muốn gọi macro nhiều lần.

```
Ví dụ:  WAIT      MACRO    count
              MOV      CX, count
      next:  ADD      AX, 1
              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 ; nhãn địa phương
              MOV      CX, count
      next:  ADD      AX, 1
              LOOP     next
              ENDM
```

E. Nhóm lệnh giả về liệt kê (listing)

Nhóm lệnh này dùng để điều khiển in ấn chương trình theo 1 định dạng văn bản ở đầu mỗi trang, như: Số trang, số cột, Tựa đề chương trình

PAGE số hạng, số cột

Ví dụ : PAGE 58, 60 ; mỗi trang liệt kê có 58 hàng, 60 cột.

TITLE (đề tựa) : cho phép đặt đề tựa chương trình

Ví dụ : TITLE chương trình hợp ngữ thứ nhất.

SUBTTL (Subtitle: đề tựa con): liệt kê tựa đề con ở mỗi đầu trang.

% OUT <văn bản> : văn bản được liệt kê khi hợp ngữ dịch chương trình. Bao gồm các chỉ thị loại văn bản sau:

.LIST : Liệt kê tất cả dòng lệnh với mã của nó (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 bảng đối chiếu chéo

.XCREF: Không liệt kê bảng đối chiếu chéo

2.3.4. Toán hạng và toán tử

A. Toán hạng:

Hợp ngữ phân biệt 3 loại toán hạng: tức thì, thanh ghi và ô nhớ. Để phân biệt, mỗi loại toán hạng có cách viết khác nhau hoàn toàn.

A.1. Toán hạng tức thì: có thể là một số, biểu thức hay một ký hiệu đã được định nghĩa (bằng lệnh EQU hay =).

Toán hạng tức thì có thể được sử dụng theo 4 dạng: thập phân, nhị phân, thập lục phân, bát phân và Ký tự

Cách viết: 10, 'B', 10b, 10h

Ký hiệu tổng quát đại diện cho toán hạng tức thì:

Immed : Toán hạng tức thì nói chung

Immed8 : Toán hạng tức thì 8 bit

Immed16 : Toán hạng tức thì 16 bit

Immed32 : Toán hạng tức thì 32 bit

A.2. Toán hạng thanh ghi: dùng để truy xuất đến nội dung thanh ghi.

Cách viết: AX, BX, CX, DX, AL, AH, SI, DS, ...

Ký hiệu tổng quát đại diện cho toán hạng thanh ghi:

Reg : Toán hạng thanh ghi đa dụng nói chung

Reg8 : Toán hạng thanh ghi đa dụng 8 bit

Reg16 : Toán hạng thanh ghi đa dụng 16 bit

SegReg : Toán hạng thanh ghi đoạn

A.3. Toán hạng ô nhớ: dùng để truy xuất đến nội dung ô nhớ.

Toán hạng ô nhớ thường được tượng trưng một địa chỉ của ô nhớ. Địa chỉ đó luôn luôn là địa chỉ độ dời trong một đoạn tương ứng.

Các cách viết cơ bản:

Tên_Biến : Truy xuất đến nội dung của Biến.

[Immed16] : Truy xuất nội dung của ô nhớ trong đoạn DS, ô nhớ đó có địa chỉ độ dời là *Immed16* . Còn gọi là trực tiếp ô nhớ.

[Reg16] : Truy xuất nội dung của ô nhớ trong đoạn DS, mà ô nhớ đó có địa chỉ độ dời lưu trong *Reg16*. Còn gọi là gián tiếp thanh ghi.

Ví dụ: `MOV AH, FOO` ; $AH \leftarrow$ nội dung biến *FOO*
 `MOV DH, [1234h]` ; $DH \leftarrow$ nội dung ô nhớ có địa chỉ là 1234h
 `MOV AL, [BX]` ; $AL \leftarrow$ nội dung ô nhớ có địa chỉ trong *BX*

Ký hiệu tổng quát đại diện cho toán hạng ô nhớ:

Mem : Toán hạng ô nhớ nói chung
Mem8 : Toán hạng ô nhớ 8 bit
Mem16 : Toán hạng ô nhớ 16 bit
Mem32 : Toán hạng ô nhớ 32 bit

Ví dụ các cách viết khác của toán hạng ô nhớ:

`FOO+5` , `FOO[5]` , `5[FOO]` : chỉ đến địa chỉ “FOO cộng 5” (*FOO* là tên của 1 biến)

| FOO | FOO+1 | FOO+2 | FOO+3 | FOO+4 | FOO+5 |
|-----|-------|-------|-------|-------|-------|
| | | | | | |

`5[BX][SI]` , `[BX+5][SI]` , `[BX]5[SI]` : tương đương với `[BX+SI+5]`.

B. Toán tử:

Có 4 loại toán tử: Thuộc Tính, Số Học, Quan Hệ và Logic.

Cú pháp chung: <Toán tử> <Toán hạng>

Toán tử tác động lên toán hạng để cho ra kết quả, kết quả này dùng làm toán hạng trong lệnh.

B.1. Toán tử thuộc tính (attribute):

PTR (pointer): dùng thay đổi kiểu của các địa chỉ, số liệu

Ví dụ: `CALL Word PTR [BX+SI]` ; Gọi CT Con

`[BX+SI]` mặc nhiên trở tới ô nhớ (dữ liệu) 1 byte, nhưng nếu muốn lấy dữ liệu 2 byte thì phải dùng *WORD PTR* để đổi.

: (Dấu hai chấm): dùng thay đổi đoạn mặc nhiên.

Ví dụ: `MOV AH, ES : [BX+SI]` ; $AH \leftarrow ES:[BX+SI]$

`[BX+SI]` mặc nhiên là địa chỉ độ dời của dữ liệu trong đoạn DS nhưng nếu dữ liệu trong đoạn ES thì phải viết *ES:[BX+SI]*

SHORT: dùng để thay đổi kiểu mặc nhiên là Near của lệnh *JMP* (lệnh nhảy) và báo cho hợp ngữ biết chỉ nhảy trong vòng từ -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` ⇔ `NH LABEL BYTE`

`SCH = THIS NEAR` ⇔ `SCH LABEL NEAR`

SEG : trả về địa chỉ đoạn của một nhãn hay biến

Ví dụ: lấy địa chỉ đoạn của biến *TENB* vào *AX*

`MOV AX, SEG TENB`

OFFSET: trả về địa chỉ *độ dời* của một nhãn hay biến

Ví dụ: MOV BX, OFFSET FOO ; độ dời của biến FOO

TYPE: Xác định kiểu của biến hay nhãn

TYPE <biến>: cho số byte ô nhớ mà loại biến đó được khai báo
(BYTE = 1, WORD = 2, DWORD = 4 ...)

TYPE <nhãn>: 0FFFFh nếu nhãn Near; 0FFFEh nếu nhãn Far

LENGTH : Trả về số phần tử của biến mảng khai báo bằng DUP()

SIZE : trả về tổng số byte mà một biến chiếm

Ví dụ: FOO DW 100 DUP (?)

MOV AX, type FOO ; AX ← 2

MOV CX, length FOO ; CX ← 100

MOV BX, size FOO ; BX ← 200

MOV DX, type FOO * length FOO ; DX ← 200

B.2. Toán tử số học:

Ngoài các toán tử số học thông dụng (+, -, *, /) còn có các toán tử:

MOD : chia lấy số dư

SHR : dịch sang trái

SHL : dịch sang phải

– : Dấu trừ đứng trước một số để chỉ đó là số âm: -5, -300

Ví dụ: MOV AX, 100 MOD 17 ; AX ← 15

MOV AX, 1100000b SHR 5 ; AX ← 11b

B.3. 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> <Toán Tử> <toán hạng 2>

<toán tử>:

EQ (Equal): trả về 1 (TRUE) nếu 2 toán hạng bằng nhau

NE (not equal): trả về TRUE nếu 2 toán hạng không bằng nhau

LT (less than): trả về TRUE nếu (toán hạng 1) < (toán hạng 2).

GT (greater than): trả về TRUE nếu (toán hạng 1) > (toán hạng 2)

LE (less than or equal): trả về TRUE nếu (t.hạng 1) ≤ (t.hạng 2)

GE (greater than or equal): trả về TRUE nếu (t.hạng 1) ≥ (t.hạng 2)

B.4. Toán tử logic: so sánh từng bit tương ứng giữa 2 toán hạng.

NOT : trả về TRUE nếu hai toán hạng logic 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.

Ví dụ: MOV AH, 10 ; $AH \leftarrow 10$
 MOV AL, 15 ; $AL \leftarrow 15$
 MOV DH, AH EQ AL ; $DH \leftarrow 0$ (vì $AH \neq AL$)
 MOV DL, AH EQ 10 ; $DL \leftarrow 1$ (vì $AH = 10$)
 MOV CH, AL LT AH ; $CH \leftarrow 0$ (vì $AL < AH$)
 MOV BH, AH AND 0Ah ; $BH \leftarrow 1$ (vì $AL < AH$)

2.4. CẤU TRÚC CỦA CHƯƠNG TRÌNH HỢP NGỮ MASM

Một chương trình hợp ngữ có thể gồm nhiều module. Các module có thể viết riêng lẻ bằng một chương trình xử lý văn bản và dịch riêng lẻ 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 tổng quát 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
        .... ; viết chương trình con
CTC      ENDP

CSEG     ENDS
        END BATDAU ; Chấm dứt CTrình và chọn địa chỉ bắt đầu
    
```

2.4.3. Tập tin thi hành dạng COM và dạng EXE

Hợp ngữ MASM cho phép tạo 2 loại tập tin thi hành. Tập tin dạng COM dùng cho các chương trình nhỏ và tập tin dạng EXE dùng để xây dựng các chương trình lớn. Mỗi loại có cấu trúc chương trình khác nhau.

1. Dạng COM:

- Chỉ dùng một đoạn duy nhất, các thanh ghi CS, DS, ES và SS đều có giá trị giống nhau.
- 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 vùng PSP (program segment prefix) 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 chương trình điển hình của tập tin dạng COM như sau:

```

..... ; Khai báo hằng số (Nếu có)
MNAME MACRO
    ..... ; Viết Macro (Nếu có)
    ENDM
CSEG SEGMENT
    ASSUME CS: CSEG, DS: CSEG
    ORG 100h
start: MOV AX, CSEG
    MOV DS, AX ; khởi động DS
    ... ; (Chương trình chính)
    INT 20h ; thoát
    ... ; Khai báo dữ liệu (Nếu có)
CT_CON PROC
    ..... ; Viết Thủ tục (Nếu có)
    RET
CT_CON ENDP
CSEG ENDS
END start
    
```

2.4.3.2. Dạng EXE:

- Chương trình lớn và nằm ở nhiều đoạn khác nhau.
- Có thể gọi các chương trình con 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.

Tập tin dạng EXE, không dùng lệnh ORG 100h ở đầu chương trình.

Cấu trúc chương trình điển hình của tập tin dạng EXE như sau:

```

.... ; Khai báo hằng số (nếu có)
MNAME      MACRO ; (Nếu có)
    ...      ; Viết lệnh Macro
    ENDM
DSEG SEGMENT
    ...      ; Khai báo dữ liệu
DSEG ENDS
CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
        MOV DS, AX      ; khởi động DS
        .....
        ..... ; (Chương trình chính)
        .....
        MOV AH, 4Ch      ; hay MOV AX, 4C00h
        INT 21h          ; thoát
CT_CON     PROC
            ..... ; Viết các Thủ tục (Nếu có)
            RET
CT_CON     ENDP
CSEG ENDS
        END start
    
```

2.4.4. Ví dụ:

Đoạn chương trình hợp ngữ sau thực hiện việc in ký tự 'B' lên màn hình (sử dụng hàm 02h của ngắt 21h – Ngắt này được trình bày cụ thể trong chương 4)

```

MOV AH, 02h      ; Sử dụng hàm in ký tự của ngắt 21h
MOV DL, 'B'      ; DL chứa ký tự cần in
INT 21h          ; Gọi ngắt để thực công việc
    
```

1. Muốn đoạn chương trình này thực thi trên máy tính ở dạng COM thì phải dùng cấu trúc chương trình dạng COM để viết, như sau:

CSEG SEGMENT

ASSUME CS: CSEG

ORG 100h

Begin: MOV AH, 02h ; Sử dụng hàm in ký tự của ngắt 21h

MOV DL, 'B' ; DL chứa ký tự cần in

INT 21h ; Gọi ngắt để thực công việc

INT 20h

CSEG ENDS

END Begin

2. Muốn đoạn chương trình này thực thi trên máy tính ở dạng EXE thì phải dùng cấu trúc chương trình dạng EXE để viết, như sau:

DSEG SEGMENT

; Vì không có biến để khai báo nên đoạn DSEG có thể bỏ đi

DSEG ENDS

CSEG SEGMENT

ASSUME CS: CSEG, DS: DSEG

Begin: MOV AX, DSEG

MOV DS, AX

MOV AH, 02h ; Sử dụng hàm in 1 ký tự của ngắt 21h

MOV DL, 'B' ; DL chứa ký tự cần in

INT 21h ; Gọi ngắt để thực công việc

MOV AH, 4Ch

INT 21h

CSEG ENDS

END Begin

3. Viết lại chương trình trên nhưng không khai báo đoạn DSEG, khi đó chương trình chỉ có 1 đoạn duy nhất, như sau:

CSEG SEGMENT

ASSUME CS: CSEG, DS: CSEG

Begin: MOV AX, CSEG

MOV DS, AX

MOV AH, 02h ; Sử dụng hàm in ký tự của ngắt 21h

MOV DL, 'B' ; DL chứa ký tự cần in

INT 21h ; Gọi ngắt để thực công việc

MOV AH, 4Ch

INT 21h

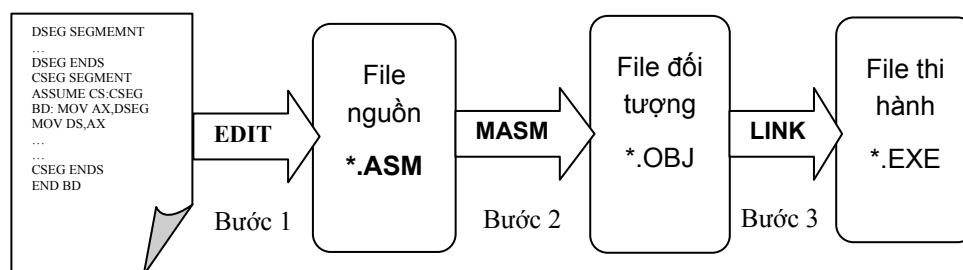
CSEG ENDS

END Begin

II.5. CÁCH TẠO CHƯƠNG TRÌNH HỢP NGỮ:

Để tạo một chương trình viết bằng hợp ngữ chạy trên máy tính PC, bạn cần phải có bộ trình hợp dịch MASM (hay TASM). Trình hợp dịch MASM được cung cấp gồm các 2 tập tin cơ bản: MASM.EXE (hợp dịch) và LINK.EXE (liên kết). Ngoài ra còn một số tập tin khác như EXE2BIN.COM dùng để chuyển sang dạng COM và DEBUG.COM dùng để gỡ rối chương trình. Các tập tin này được thực thi dưới MS-DOS.

Cần ba hoặc bốn bước để tạo chương trình hợp ngữ chạy được trên máy tính PC. Hình 2.6 và 2.7 mô tả các bước và lưu đồ thực hiện qui trình này.



Hình 2.6: Các bước tạo chương trình hợp ngữ

BƯỚC 1: Thảo chương, tạo tập tin nguồn (*.ASM)

Dùng bất kỳ trình soạn thảo văn bản dạng TEXT quen thuộc để viết chương trình nguồn hợp ngữ, lưu thành file có phần mở rộng là ***.ASM**, Ví dụ: **HELLO.ASM**

BƯỚC 2: Hợp dịch file ASM thành file đối tượng (HELLO.OBJ)

C:\> MASM <filename>;>

Trình hợp dịch kiểm tra lỗi cú pháp trên từng dòng lệnh. Nếu có lỗi thì phải quay về bước 1 để sửa chữa và dịch lại cho đến hết lỗi.

BƯỚC 3: Liên kết một hay nhiều file OBJ thành file thi hành (*.EXE)

C:\> LINK <filename1>;> [<filename2>;> ...] [<filename.exe>]

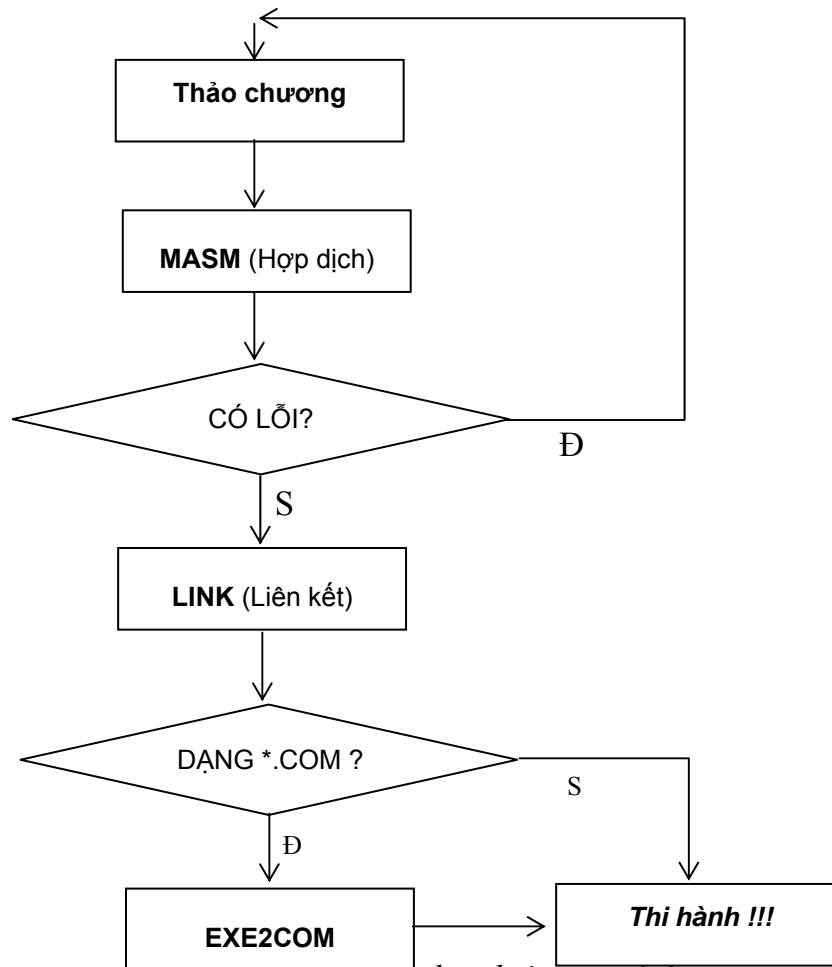
BƯỚC 4: Đổi từ EXE sang COM (Dành cho chương trình viết dạng COM)

C:\> EXE2BIN <filename.EXE> <filename.COM>

Ví dụ: Dịch file nguồn HELLO.ASM thành file thực thi HELLO.EXE, từng bước thực hiện các lệnh như sau

C:\>MASM HELLO;

C:\>LINK HELLO;



Hình 2.1: Lưu đồ thực hiện qui trình

Ngày nay, hầu hết các máy tính đều sử dụng hệ điều hành Windows, nên các lập trình viên đã phát triển các phần mềm chạy trên Windows 98/2000/XP để làm môi trường phát triển Hợp ngữ, như RadASM Assembly IDE (<http://www.radasm.com>). Khi đó, việc sử dụng các phần mềm này đơn giản hơn rất nhiều thông qua giao diện đồ họa, nó cung cấp sẵn giao diện soạn thảo chương trình nguồn mạnh và các bước hợp dịch, liên kết đều được thực hiện trên thanh công cụ hay bằng phím tắt.

Chương 3

TẬP LỆNH INTEL-8086 và KIỂU ĐỊNH VỊ

3.1. TẬP LỆNH CỦA CPU-8086

Bộ xử lý 8086 có tập lệnh gồm 111 lệnh với chiều dài của lệnh từ 1 byte đến vài byte. Tập lệnh của họ các bộ xử lý Intel 80x86 càng ngày càng có nhiều lệnh mạnh và phức tạp (CPU-80386 có 206 lệnh, Pentium có hơn 400 lệnh).

Lệnh của CPU-8086 có tối đa 2 toán hạng, mỗi lệnh có số toán hạng và loại toán hạng đã được xác định trước. Cú pháp tổng quát như sau:

<Tác Vụ Lệnh> TH1, TH2

Tác vụ lệnh cho biết công việc mà CPU sẽ thực hiện (là thành phần bắt buộc), các toán hạng (TH1 và TH2) phải cách nhau bởi dấu phẩy (,).

Đối với lệnh có hai toán hạng, thông thường TH2 sẽ là toán hạng nguồn còn TH1 là toán hạng đích và cũng là toán hạng nguồn còn lại. Kích thước của hai toán hạng phải bằng nhau:

- Nếu TH1 là 8 bit thì TH2 cũng phải là 8 bit và ngược lại.
- Nếu TH2 là 16 bit thì TH1 cũng phải là 16 bit và ngược lại.

Tập lệnh CPU-8086 được sắp xếp theo các nhóm như sau:

- Sao chép dữ liệu và địa chỉ
- Tính toán số học
- Tính toán logic, ghi dịch và quay
- Truy xuất ngoại vi
- Lệnh hệ thống
- Rẽ nhánh và vòng lặp
- Chương trình con
- Xử lý chuỗi

Chương này chỉ trình bày các nhóm lệnh *đơn giản và thường dùng*. Các nhóm lệnh phức tạp như: rẽ nhánh, vòng lặp, chương trình con và xử lý chuỗi) sẽ được trình bày trong những chương sau (chương 5, 6 và 7). Tập lệnh đầy đủ của bộ xử lý Intel-8086 sẽ được giới thiệu trong phụ lục 1.

3.1.1. Lệnh sao chép dữ liệu, địa chỉ:

1. MOV - Di chuyển dữ liệu

Cú pháp: MOV Đích, Nguồn
 MOV SegReg, Reg16
 MOV Reg16, SegReg

Nguồn (toán hạng nguồn) : Reg, Mem, Immed.

Đích (toán hạng đích) : Reg, Mem.

Ý nghĩa: Sao chép dữ liệu (xác định bởi toán hạng nguồn) vào vùng nhớ hoặc thanh ghi (xác định bởi toán hạng đích). Sau khi thực hiện xong thì giá trị toán hạng nguồn và toán hạng đích bằng nhau.

Lưu ý: - Lệnh MOV không ảnh hưởng đến thanh ghi trạng thái.

- Một trong hai toán hạng phải là thanh ghi.

Ví dụ: MOV CX, BX ; Sao chép nội dung thanh ghi BX vào thanh ghi CX. Sau khi thực hiện xong, BX=CX.

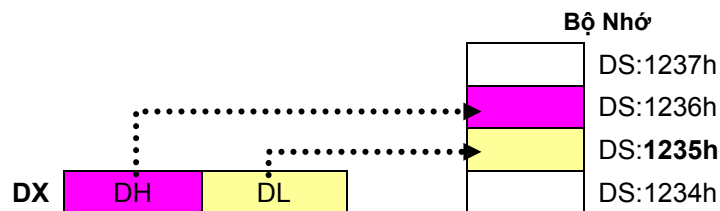
MOV DX, 1234h ; DL ← 34h, DH ← 12h

MOV DS, AX ; DS ← AX

MOV AH, [1234h] ; AH ← M[DS:1234h]

MOV [1235h], DX ; M[DS:1235h] ← DL, M[DS:1236h] ← DH

MOV AH, CX ; **SAI** vì AH có 8 bit trong khi CX là 16 bit.



Hình 3.1: Mô tả lệnh MOV [1235h], DX

Hình 3.1 mô tả cách sao chép dữ liệu 16 bit giữa thanh ghi và ô nhớ. Byte thấp của dữ liệu trong DX (là DL) đưa vào ô nhớ địa chỉ thấp (DS:1235h), còn byte cao trong DX (là DH) đưa vào ô nhớ có địa chỉ cao hơn (DS:1236h)

2. XCHG – Hoán chuyển dữ liệu

Cú pháp: XCHG Đích, Nguồn ; Đích ↔ Nguồn

Nguồn, đích: Reg, Mem.

Ý nghĩa: Lệnh XCHG hoán chuyển dữ liệu giữa nguồn và đích. Cả hai toán hạng không đồng thời là Mem.

Ví dụ: XCHG AL, BH ; AL ↔ BH

XCHG AX, [1235h]; AL ↔ M[DS:1235h], AH ↔ M[DS:1236h]

XCHG bienA, bienB ; **SAI** vì bienA và bienB đều là vùng nhớ.

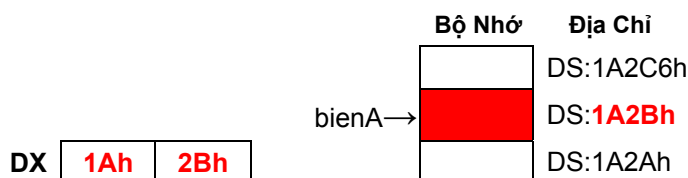
3. LEA - Lấy địa chỉ độ dời

Cú pháp: LEA Reg16, Mem16 ; Reg16 ← Địa chỉ độ dời

Ý nghĩa: Lấy địa chỉ độ dời của biến hay nhãn (mem16) đưa vào thanh ghi đích Reg16.

Ví dụ: LEA DX, bienA ; Lấy độ dời của biến bienA lưu vào DX

Theo hình 3.1, bienA được cấp phát tại vùng nhớ có địa chỉ độ dời là 1A2Bh, sau khi thực hiện lệnh thì thanh ghi DX = 1A2Bh.



Hình 3.2: Mô tả lệnh LEA DX, bienA

3.1.2. Lệnh tính toán số học.

1. ADD - Cộng hai số nguyên

Cú pháp: ADD đích, nguồn ; đích \leftarrow đích + nguồn

Nguồn : Reg, Mem, Immed.

Đích : Reg, Mem.

Ý nghĩa: Lấy toán hạng nguồn cộng toán hạng đích và lưu kết quả lưu toán hạng đích.

2. SUB - Trừ hai số nguyên

Cú pháp: SUB đích, nguồn ; đích \leftarrow đích – nguồn

Nguồn : Reg, Mem, Immed.

Đích : Reg, Mem.

Ý nghĩa: Lấy toán hạng đích (số bị trừ) trừ toán hạng nguồn (số trừ) và lưu kết quả ở toán hạng đích.

Ví dụ: ADD AL, 15 ; AL \leftarrow AL + 15
 ADD AX, DX ; AX \leftarrow AX + DX
 SUB BX, 15h ; BX \leftarrow BX – 15h
 SUB DX, CL ; **SAI** vì DX là 16bit trong khi CL chỉ có 8bit
 ADD 15h, AL ; **SAI** vì đích là Immed

3. INC - Tăng 1 đơn vị (Increment)

Cú pháp: INC đích ; đích \leftarrow đích + 1

Đích : Reg, Mem.

Ý nghĩa: Tăng nội dung của toán hạng đích thêm 1 đơn vị.

4. DEC - Giảm 1 đơn vị (Decrmemt)

Cú pháp: DEC đích ; đích \leftarrow đích – 1

Đích : Reg, Mem.

Ý nghĩa: Giảm nội dung của đích đi 1 đơn vị.

Ví dụ: INC AL ; AL \leftarrow AL + 1
 DEC AX ; AX \leftarrow AX - 1
 INC bienC ; bienC \leftarrow bienC + 1

5. MUL - Nhân hai số nguyên

Cú pháp: MUL nguồn

Nguồn: Reg, Mem

Ý nghĩa: Nhân thanh ghi tích lũy với toán hạng nguồn. Tùy vào kích thước của toán hạng nguồn mà CPU thực hiện phép nhân 8 hay 16 bit.

- **Nhân 8 bit:** dành cho toán hạng nguồn là 8 bit. Khi đó, CPU sẽ lấy thanh ghi AL nhân với toán hạng nguồn và lưu kết quả vào AX.
- **Nhân 16 bit:** dành cho toán hạng nguồn là 16 bit. Khi đó, CPU sẽ nhân thanh ghi AX với toán hạng nguồn và lưu kết quả vào cặp thanh ghi DX:AX (*nghĩa là kết quả có 32 bit, 16 bit cao lưu vào DX, còn 16 bit thấp lưu vào AX*)

Ví dụ: MUL BL ; AX \leftarrow AL * BL (*nhân 8 bit*)
MUL BX ; DX:AX \leftarrow AX * BX (*nhân 16 bit*)
MUL [1235h] ; AX \leftarrow AL * M[DS:1235h] (*nhân 8 bit*)

6. DIV - Chia hai số nguyên

Cú pháp: DIV nguồn

Nguồn: Reg, Mem

Ý nghĩa: lấy thanh ghi tích lũy (số bị chia) chia cho nguồn (số chia). Tùy vào kích thước của toán hạng nguồn mà CPU thực hiện phép chia 8 hay 16 bit.

- **Chia 8 bit:** dành cho toán hạng nguồn là 8 bit. Khi đó, CPU sẽ lấy thanh ghi AX chia cho toán hạng nguồn. Thương số của kết quả được lưu vào AL còn phần dư lưu vào AH.
- **Chia 16 bit:** dành cho toán hạng nguồn là 16 bit. Khi đó, CPU lấy DX:AX (số 32 bit) chia cho toán hạng nguồn. Thương số của kết quả được lưu vào AX còn phần dư thì lưu vào DX.

Ví dụ: DIV DL ; AL, AH \leftarrow AX \div DL
DIV BX ; AX, DX \leftarrow DX:AX \div BX

3.1.3. Nhóm lệnh logic và ghi dịch

1. AND, OR, XOR, TEST: Và, Hoặc, Cộng không nhớ, Kiểm tra bit

Cú pháp: AND đích, nguồn ; đích \leftarrow đích \wedge nguồn
OR đích, nguồn ; đích \leftarrow đích \vee nguồn
XOR đích, nguồn ; đích \leftarrow đích \oplus nguồn
TEST đích, nguồn ; Cờ \leftarrow đích \wedge nguồn

Nguồn: Reg, Mem hay Immed.

Đích: Reg, Mem.

Ý nghĩa: Thực hiện phép toán logic (and, or, xor) theo từng bit tương ứng giữa toán hạng đích và toán hạng nguồn, lưu giữ kết quả ở toán hạng đích.

- Lệnh TEST thực hiện kiểm tra bit, giống lệnh AND nhưng không lưu giữ lại kết quả mà kết quả của phép toán chỉ ảnh hưởng đến các cờ trạng thái.

Ví dụ: MOV AL, 01101110b ; AL \leftarrow 01101110b
AND AL, 00110110b ; AL \leftarrow AL \wedge 01101110b (*AL=00100110b*)
XOR AX, AX ; AX \leftarrow AX \oplus AX (*Kết quả: AX = 0*)

AND DL, 11000011b ; Xóa 4 bit giữa trong thanh ghi DL

TEST DH, 00010000b ; Kiểm tra giá trị của bit 4 trong thanh ghi DH. Kết quả kiểm tra sẽ ảnh hưởng lên cờ Zero (ZF)

2. NOT – Đảo bit (Lấy Bù 1)

Cú pháp: NOT đích ; đích $\leftarrow \overline{\text{đích}}$

Đích: Reg, Mem

Ý nghĩa: Đảo ngược từng bit (hay lấy bù 1) toán hạng đích.

Ví dụ: MOV AH, 0Fh ; AH \leftarrow 0Fh (00001111b)

NOT AH ; AH = 0F0h (11110000b)

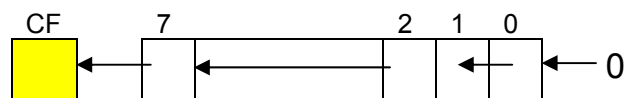
3. SHL – Dịch trái logic (logical SHift Left)

Cú pháp: SHL đích, 1 ; dịch toán hạng đích sang trái 1 bit.

SHL đích, CL ; dịch sang trái CL bit

Đích: Reg, Mem

Ý nghĩa: Dịch toán hạng đích sang trái, có thể dịch 1 bit hay nhiều bit hơn (CL chứa số bit dịch). LSB được nạp vào logic 0, còn MSB được dịch sang cờ Carry (CF).



Hình 3.3: Dịch toán hạng đích 8 bit sang trái 1 bit

Ví dụ: MOV AL, 01101101b ; AL = 01101101b

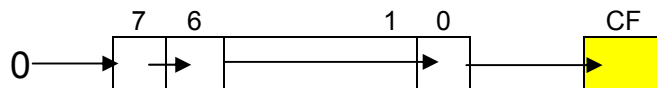
SHL AL, 1 ; AL = 11011010b và CF=0

4. SHR (logical shift right) dịch phải logic.

Cú pháp: SHR đích, 1 ; dịch phải toán hạng đích 1 bit.

SHR đích, CL ; dịch phải đích với số bit trong CL

Ý nghĩa: Giống lệnh SHL nhưng bây giờ dịch toán hạng đích sang phải



Hình 3.4: Dịch toán hạng đích 8 bit sang phải 1 bit

Ví dụ: MOV AL, 01101101b ; AL = 01101101b

SHR AL, 1 ; AL = 00110110b và CF=1

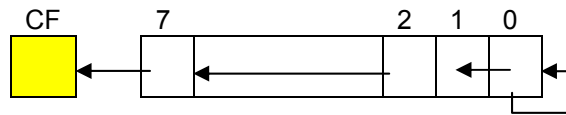
5. SAL (Shift arithmetic left): dịch trái số học

Cú pháp: SAL đích, 1 ; dịch toán hạng đích sang trái 1 bit.

SAL đích, CL ; dịch sang trái CL bit

Đích: Reg, Mem

Ý nghĩa: Lệnh này giống SHL nhưng LSB (bit thấp nhất) vừa được dịch lên bit 1 và giữ nguyên (bảo toàn bit 0).



Hình 3.5: Dịch toán hạng đích 8 bit sang trái 1 bit

Ví dụ: MOV AL, 01101101b ; AL = 01101101b
 SHL AL, 1 ; AL = 11011011b và CF=0

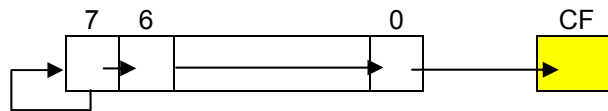
6. SAR (Shift arithmetic right) : dịch phải số học

Cú pháp: SAR đích, 1 ; dịch toán hạng đích sang trái 1 bit.

 SAR đích, CL ; dịch sang trái CL bit

Đích: Reg, Mem

Ý nghĩa: Lệnh này giống SHR nhưng MSB (bit cao nhất) vừa được dịch xuống bit thấp hơn và giữ nguyên (bảo toàn bit MSB – Bit dấu).



Hình 3.6: Dịch toán hạng đích 8 bit sang phải 1 bit

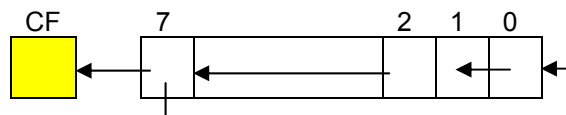
Ví dụ: MOV AL, 01101101b ; AL = 01101101b
 SAR AL, 1 ; AL = 00110110b và CF=1

7. ROL (Rotate left) : quay vòng sang trái

Cú pháp: ROL đích, 1 ; quay vòng toán hạng đích sang trái 1 bit.

 ROL đích, CL ; quay vòng sang trái số lần quay bằng CL.

Ý nghĩa: Giống SHL, nhưng thực hiện việc quay vòng, MSB dịch vào LSB



Hình 3.7: Quay toán hạng đích 8 bit sang trái 1 bit

Ví dụ: MOV AL, 01101101b ; AL = 01101101b
 ROL AL, 1 ; AL = 11011010b và CF=0

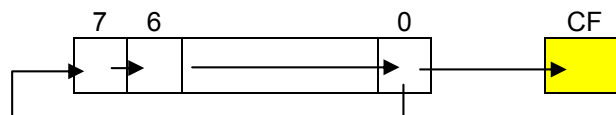
8. ROR (Rotate right) : Quay vòng sang phải

Cú pháp: ROR đích, 1 ; Quay vòng toán hạng đích sang phải 1 bit.

 ROR đích, CL ; Quay vòng với số lần quay bằng CL

Đích: Reg, Mem

Ý nghĩa: Lệnh này giống ROL nhưng quay vòng sang phải, nghĩa là LSB (bit thấp nhất) vừa được dịch đến MSB.



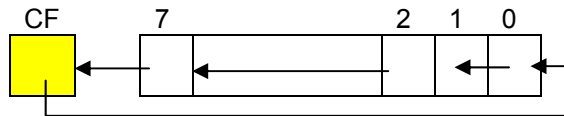
Hình 3.8: Quay toán hạng đích 8 bit sang phải 1 bit

Ví dụ: `MOV AL, 01101101b` ; `AL = 01101101b`
 `SAR AL, 1` ; `AL = 00110110b` và `CF=1`

9. RCL (Rotate through carry left): Quay trái qua cờ Carry

Cú pháp: `RCL đích, 1` ; quay vòng TH. đích qua cờ sang trái 1 bit.
 `RCL đích, CL` ; quay vòng qua cờ sang trái số lần bằng CL.

Ý nghĩa: Giống như lệnh ROL nhưng có sự tham gia của cờ carry (CF). Xem hình 3.9. Như vậy CF cũng là 1 bit tham gia vào vòng quay



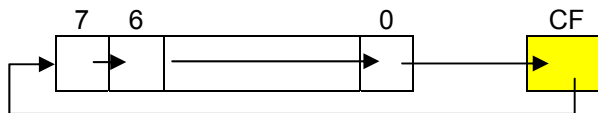
Hình 3.9: Quay toán hạng đích 8 bit qua cờ sang trái 1 bit

10. RCR (Rotate through carry right): Quay vòng qua cờ sang phải

Cú pháp: `RCR đích, 1` ; Quay vòng TH. đích qua cờ sang phải 1 bit.
 `RCR đích, CL` ; số lần quay bằng CL

Đích: *Reg, Mem*

Ý nghĩa: Lệnh này giống RCL nhưng quay vòng sang phải.



Hình 3.10: Quay toán hạng đích 8 bit qua cờ sang phải 1 bit

3.1.4. Nhóm lệnh vào ra ngoại vi.

1. IN - Lấy số liệu từ ngoại vi.

Cú pháp: `IN AL, địa chỉ cổng 8 bit`
 `IN AL, DX` ; `DX` chứa địa chỉ cổng 16 bit

Ý nghĩa: Lấy dữ liệu ở ngoại vi có địa chỉ 8 bit đưa vào thanh ghi AL. Nếu địa chỉ của ngoại vi là 16 bit thì phải dùng DX chứa địa chỉ này.

Ví dụ : - Đọc dữ liệu ở ngoại vi 3Fh

`IN AL, 3Fh` ; `AL ← port(3Fh)` (*3Fh là địa chỉ cổng 8 bit*)

- Đọc dữ liệu ở ngoại vi 3F8h

`MOV DX, 3F8h` ; `DX ← 3F8h` (*3F8h là địa chỉ cổng 16 bit*)

`IN AL, DX` ; `AL ← port(DX)`

2. OUT - Đưa số liệu ra ngoại vi.

Cú pháp: `OUT địa chỉ cổng 8 bit, AL` ; `port(8 bit) ← AL`
 `OUT DX, AL` ; `port(DX) ← AL`

Ý nghĩa: Giống lệnh IN nhưng là xuất dữ liệu trong AL ra ngoại vi.

Ví dụ: `MOV DX, 04A5h` ; `DX ← 04A5h`
 `OUT DX, AL` ; `port(DX) ← AL`

1.3.5. Nhóm lệnh hệ thống

Nhóm lệnh hệ thống bao gồm các lệnh điều khiển hoạt động của CPU hay thiết lập giá trị các cờ. Các lệnh này không có toán hạng do đó cú pháp của lệnh rất đơn giản chỉ gồm các tác vụ lệnh sau:

- CLC ; Xóa cờ CF ($CF = 0$) – *Clear Carry flag*
- CMC ; Đảo ngược cờ CF – *Complement Carry flag*
- CLD ; Xóa cờ hướng ($DF=0$) – *Clear Direction flag*
- STD ; Đặt cờ hướng ($DF=1$) – *Set Direction flag*
- CLI ; Xóa cờ ngắt ($IF=0$) – *Clear Interrupt flag*
- STI ; Đặt cờ ngắt ($IF=1$) – *Set Interrupt flag*
- HLT ; Dừng mọi hoạt động của CPU (treo máy) – *Halt*
- INT ; Gọi ngắt phần mềm – *Interrupt*
- IRET ; Trở về chương trình chính từ chương trình phục vụ ngắt – *Return from Interrupt*
- LOCK ; Khóa Bus ngoài. CPU sẽ không giao tiếp với BUS
- NOP ; Không có tác vụ – *No operation*
- WAIT ; Tạm dừng CPU cho đến khi có tín hiệu điện (mức thấp) ở chân TEST của CPU-8086 thì hoạt động tiếp.

3.2. KIỂU ĐỊNH VỊ

Kiểu định vị (Addressing mode) là cách thức xác định vị trí của dữ liệu được mô tả trong câu lệnh. Thông qua kiểu định vị, CPU biết được vị trí lưu trữ toán hạng để truy xuất đến đúng giá trị mà lệnh cần xử lý.

Tùy vào loại toán hạng, CPU-8086 cung cấp các kiểu định vị sau:

- Định vị thanh ghi (register addressing)
- Định vị tức thì (immediate addressing)
- Định vị trực tiếp (direct addressing)
- Định vị gián tiếp thanh ghi (register indirect addressing)
- Định vị nền (based offset addressing)
- Định vị chỉ số (index offset addressing)
- Định vị chỉ số nền (based index offset addressing)
- Định vị chuỗi (string addressing)
- Định vị vào ra (port in/out addressing).

Trong các kiểu định vị trên thì kiểu định vị thanh ghi và tức thì có tốc độ truy xuất nhanh nhất. Đối với toán hạng tức thì, dữ liệu nằm ngay trong lệnh và giá trị của nó được chèn trực tiếp vào trong mã máy; còn đối với kiểu định vị thanh ghi, do thanh ghi là một bộ phận bên trong của CPU, nên CPU không mất thời gian truy xuất bộ nhớ trong.

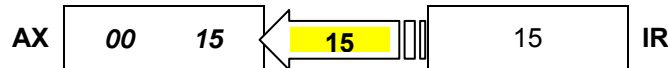
3.2.1. Định vị tức thì:

Là kiểu định vị trong đó phải có toán hạng nguồn là toán hạng tức thì và toán hạng đích phải là toán hạng thanh ghi.

Ví dụ: MOV AX, 15 ; $AX \leftarrow 15$ ($AH=0; AL=15$)
 ADD AH, 'A' ; $AH \leftarrow AH + 41h$ ('A' = $41h$)
 MOV DX, 3*1Ah ; $DX \leftarrow 4Eh$ ($DH=0; DL=4Eh$)
 IN AL, 0Fh ; Không phải kiểu định vị tức thì

Các giá trị tức thì là: 15, 'A' và 004Eh

Toán hạng tức thì là dữ liệu nằm ngay trong mã lệnh. Khi CPU thực hiện lệnh, mã lệnh được đọc vào thanh ghi lệnh (IR) do đó giá trị tức thì cũng tồn tại trong CPU khi lệnh được thi hành (Xem hình 3.11)



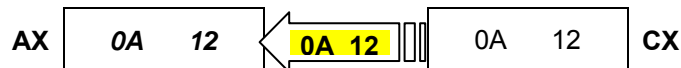
Hình 3.11: Vị trí toán hạng tức thì khi CPU thi hành lệnh

3.2.2. Định vị thanh ghi

Định vị thanh ghi là kiểu định vị mà tất cả các toán hạng đều phải là toán hạng thanh ghi. Kiểu định vị này được sử dụng phổ biến vì có tốc độ nhanh và ít chiếm bộ nhớ.

Ví dụ: MOV AX, CX ; $AX \leftarrow CX$
 INC AX ; $AX \leftarrow AX + 1$
 OUT DX, AL ; Không phải kiểu định vị thanh ghi

Trong đó AX và CX đều là 2 toán hạng thanh ghi.



Hình 3.12: Hoạt động của lệnh MOV AX, CX

3.1.3. Định vị trực tiếp (bộ nhớ):

Địa chỉ hiệu dụng (EA - Effective Address): Là địa chỉ độ dời (Offset) mà EU tính toán cho một toán hạng bộ nhớ để có thể truy xuất nội dung toán hạng bộ nhớ đó. EU có thể tính địa chỉ hiệu dụng theo nhiều cách. Tùy theo cách tính mà ta có các kiểu định vị bộ nhớ khác nhau.

Định vị trực tiếp là kiểu định vị trong đó chỉ có một toán hạng là địa chỉ độ dời của ô nhớ chứa dữ liệu, toán hạng kia là toán hạng thanh ghi. Địa chỉ hiệu dụng nằm ngay trong lệnh. Địa chỉ hiệu dụng EA kết hợp với nội dung hiện tại của thanh ghi DS trong BIU để tính toán ra địa chỉ vật lý của ô nhớ theo công thức : $PA = Segment * 10h + EA$

Ví dụ: MOV AH, 12h
 MOV FOO, AH ; $FOO \leftarrow AH (=12h)$
 MOV BX, [1234h] ; $BL \leftarrow M[DS:1234h], BH \leftarrow [DS:1235h]$

Giả sử thanh ghi đoạn dữ liệu có địa chỉ segment là DS=0200h. Vậy địa chỉ vật lý (Physical Address) của toán hạng nguồn được tính như sau:

$$PA = DS * 10h + EA = 200h * 10h + 1234h = 03234h$$

Chú ý không có lệnh nào cho phép cả hai toán hạng đều tham chiếu đến vị trí bộ nhớ. Muốn làm được điều này chúng ta phải dùng kiểu định vị gián tiếp thanh ghi.

3.1.4. Định vị gián tiếp thanh ghi

Địa chỉ hiệu dụng là nội dung của một trong các thanh ghi BX, BP, SI, DI. Định vị gián tiếp thanh ghi cũng tương tự như định vị trực tiếp, địa chỉ hiệu dụng EA kết hợp với nội dung của thanh ghi phân đoạn DS để hình thành một địa chỉ vật lý của toán hạng. Tuy nhiên cũng có sự khác nhau về cách tính độ dời. Địa chỉ hiệu dụng EA là nội dung trong các thanh ghi BP, BX, SI và DI bên trong CPU 8086.

Ví dụ: MOV AL, [BX] ; $AL \leftarrow M[DS:BX]$
ADD CX, [BP] ; $CX \leftarrow CX + M[DS:BP]$
MOV [SI], DX ; $M[DS:SI] \leftarrow DL$, $M[DS:SI+1] \leftarrow DH$

Giả sử SI = 1234h và DS = 0300h, địa chỉ vật lý được tính là:

$$PA = DS * 10h + EA = 0300h * 10h + 1234h = 04234h$$

(với $EA = SI = 1234h$)

3.1.5. Định vị nền

Giống như định vị gián tiếp thanh ghi, nhưng địa chỉ hiệu dụng EA trong trường hợp này là tổng của nội dung của thanh ghi BX hoặc BP với độ dời (giá trị 16 bit có dấu):

$$EA = BX + \text{Độ Dời} \quad \text{hay} \quad EA = BP + \text{Độ Dời}$$

Địa chỉ vật lý là tổng giá trị độ dời đã dịch chuyển trực tiếp hoặc gián tiếp nội dung của thanh ghi BX hoặc BP với giá trị chứa trong thanh ghi phân đoạn DS hoặc SS tương ứng.

Ví dụ: MOV AL, [BX + 20h] ; $AL \leftarrow M[DS:BX+20h]$
MOV [BX].beta, CX ; $M[DS:BX+beta] \leftarrow CL$
; $M[DS:BX+beta+1] \leftarrow CH$

Giả sử DS = 0200h, BX = 1000h còn beta = 0FA34h. Vậy địa chỉ vật lý của toán hạng đích trong trường hợp này là:

$$PA = 0200h * 10h + 1000h + 0FA34h = 10C34h$$

Nếu thanh ghi BP được sử dụng thay vì sử dụng thanh ghi BX thì việc tính toán địa chỉ vật lý của toán hạng đích khi đó sẽ phải sử dụng thanh ghi phân đoạn SS thay vì là DS như trên. Điều này cho phép truy xuất đến dữ liệu trong phân đoạn ngăn xếp của bộ nhớ.

3.1.6. Định vị chỉ số

Địa chỉ hiệu dụng là tổng của độ dời 16 bit và thanh ghi SI hoặc DI

$$EA = SI + \text{Độ dời} \quad \text{hoặc} \quad EA = DI + \text{Độ dời}$$

Cũng tương tự như kiểu định vị nền, địa chỉ vật lý là tổng độ dịch chuyển trực tiếp hoặc gián tiếp nội dung của thanh ghi SI hoặc DI với giá trị chứa trong thanh ghi phân đoạn DS hoặc ES tương ứng.

Ví dụ: MOV DL, [SI + 25h] ; $DL \leftarrow M[DS:SI+25h]$

$\text{MOV AL, ARRAY[SI] ; AL} \leftarrow \text{M[DS:SI+ARRAY]}$

Giả sử DS = 0200h, SI = 2000h còn ARRAY là độ dời có giá trị là 1234h. Vậy địa chỉ vật lý của toán hạng đích trong trường hợp này là:

$$\text{EA} = \text{SI} + \text{ARRAY} = 2000\text{h} + 1234\text{h} = 3234\text{h}$$

$$\text{PA} = 0200\text{h} * 10\text{h} + \text{EA} = 2000\text{h} + 3234\text{h} = 05234\text{h}$$

Nếu thanh ghi DI được sử dụng thay vì sử dụng thanh ghi SI thì việc tính toán địa chỉ vật lý của toán hạng đích khi đó sẽ phải sử dụng thanh ghi phân đoạn ES thay vì là DS như trên.

Ví dụ: $\text{MOV AL, [DI+1Fh] ; AL} \leftarrow \text{M[ES:DI+1Fh]}$

$$\text{PA} = \text{ES} * 10\text{h} + \text{DI} + 100\text{h}$$

3.1.7. Định vị chỉ số nền

Đây là kiểu định vị kết hợp từ kiểu định vị nền và định vị chỉ số. Địa chỉ hiệu dụng là tổng của thanh ghi nền (BX hay BP), thanh ghi chỉ số (SI hay DI) và độ dời 16 bit.

$$\text{EA} = \text{BX} + \text{SI} + \text{Độ Dời} \Rightarrow \text{PA} = \text{DS} * 10\text{h} + \text{EA}$$

$$\text{EA} = \text{BX} + \text{DI} + \text{Độ Dời} \Rightarrow \text{PA} = \text{ES} * 10\text{h} + \text{EA}$$

Ví dụ: $\text{MOV DL, [BX+SI+100] ; PA} = \text{DS} * 10\text{h} + \text{BX} + \text{SI} + 100$

$\text{MOV AL, [BX+DI+200] ; PA} = \text{ES} * 10\text{h} + \text{BX} + \text{DI} + 200$

3.1.8. Định vị chuỗi

Các lệnh xử lý chuỗi không dùng kiểu định vị bộ nhớ thông thường mà dùng 2 thanh ghi chỉ số SI và DI để chứa địa chỉ độ dời của chuỗi nguồn và chuỗi đích. Các lệnh xử lý chuỗi trong 8086 tự động dùng các thanh ghi chỉ mục nguồn và đích để tính toán địa chỉ của toán hạng nguồn và đích tương ứng. Kiểu định vị này gọi là định vị chuỗi và chỉ áp dụng cho các lệnh xử lý chuỗi.

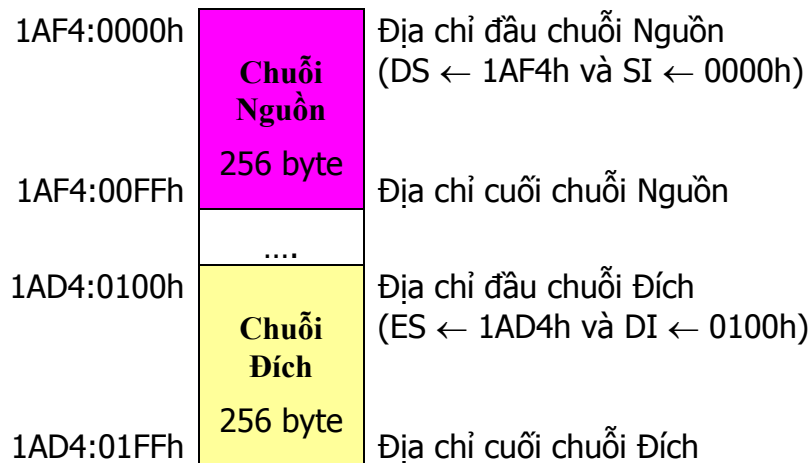
Các lệnh xử lý chuỗi của Intel-8086 được trình bày kỹ trong chương 6, bao gồm: MOVSB/W, SCASB/W, CMPSB/W, STOSB/W, LODSB/W.

Lệnh sẽ xử lý 1 lần 1 Byte hoặc 2 byte (Word) - còn gọi là *phần tử*. Sau khi thực hiện xong lệnh, SI và DI tự động điều chỉnh tăng/giảm 1 đơn vị để chỉ đến phần tử kế tiếp cần xử lý.

SI và DI tự động cùng tăng hay cùng giảm được quyết định bởi cờ Hướng (DF). Nếu DF = 0 thì xử lý theo chiều tăng địa chỉ, còn DF = 1 thì xử lý theo chiều giảm địa chỉ. Tùy vào cách xử lý chuỗi mà người lập trình chọn hướng xử lý thích hợp. Khi đó:

- DS:SI phải chỉ đến byte đầu tiên cần xử lý của chuỗi nguồn.
- ES:DI phải chỉ đến byte đầu tiên cần xử lý của chuỗi đích.

Hình 3.13 mô tả vị trí 2 chuỗi Nguồn và Đích (mỗi chuỗi 256 byte) trong bộ nhớ với chiều xử lý tăng địa chỉ.



Hình 3.13: Mô hình bộ nhớ chứa chuỗi Nguồn và Đích

3.1.9. Định vị cổng vào/ra

Định vị cổng vào/ra chỉ áp dụng các lệnh vào ra ngoại vi trong tập lệnh của 8086 (IN và OUT) để truy xuất đến các port vào/ra của thiết bị ngoại vi. Có 2 cách truy xuất đến không gian địa chỉ của Port vào ra:

- **Trực tiếp:** Chỉ dùng cho cổng có địa chỉ 8 bit. Địa chỉ cổng được cung cấp trực tiếp trên dòng lệnh.

Ví dụ: IN AL, 0F8h ; Đọc dữ liệu tại cổng 0F8h đưa vào AL
 OUT 01h, AL ; Xuất giá trị AL ra cổng có địa chỉ 01h

- **Gián tiếp:** Đối với những cổng có địa chỉ 16 bit thì không sử dụng cách trực tiếp để truy xuất được, khi đó thanh ghi DX được dùng để chứa địa chỉ của thiết bị ngoại vi.

Ví dụ: IN AL, DX ; AL ← port(DX)
 OUT DX, AL ; port(DX) ← AL

BÀI TẬP CHƯƠNG 3

3.1. Cho biết nội dung của các toán hạng đích và giá trị các cờ CF, SF, ZF, PF, OF sau khi thực hiện mỗi lệnh sau:

- a. ADD AX, BX ; Với AX = 7FFFh, BX = 1
- b. SUB AL, BL ; Với AL = 1, BL = 0FFh
- c. DEC AL ; Với AL = 0
- d. NEG AL ; Với AL = 7Fh
- e. XCHG AX, BX ; Với AX = 1ABCh, BX = 712Ah
- f. ADD AL, BL ; Với AL = 80h, BL = 0FFh
- g. SUB AX, BX ; Với AX = 0, BX = 8000h
- h. NEG AX ; Với AX = 1

3.2. Cho biết kết quả sau khi thực hiện lệnh ADD AX, DX và trạng thái cờ tràn (OF) trong các trường hợp sau:

- a. AX = 512Ch , DX = 4185h
- b. AX = 0FE12h, DX = 1ACBh
- c. AX = 0E1E4h, DX = 0DAB3h
- d. AX = 7132h , DX = 7000h
- e. AX = 6389h , DX = 1176h

3.3. Cho biết kết quả sau khi thực hiện lệnh SUB CX, BX và trạng thái cờ tràn (OF) trong các trường hợp sau:

- a. CX = 2143h , BX = 1986h
- b. CX = 81FEh , BX = 1986h
- c. CX = 19BCh , BX = 81FEh
- d. CX = 02h , BX = FE0Fh
- e. CX = 8BCDh , BX = 71ABh

3.4. Cho AL = 0CBh và CF = 1. Cho biết giá trị AL sau khi thực hiện lệnh:

- a. SHL AL, 1
- b. SHR AL, 1
- c. ROL AL, CL ; Với CL = 2
- d. ROR AL, CL ; Với CL = 3
- e. SAR AL, CL ; Với CL = 2
- f. RCL AL, 1
- g. RCR AL, CL ; Với CL = 3

3.5. Dùng lệnh TEST để thiết lập giá trị các cờ sau:

- a. ZF = 1 nếu AX = 0
- b. ZF = 0 nếu AX lẻ
- c. SF = 1 nếu DX < 0
- d. ZF = 1 nếu DX >= 0
- e. PF = 1 nếu BL chẵn.

3.6. Viết đoạn lệnh thực hiện từng yêu cầu sau:

- a. Xóa các bit ở vị trí chẵn của AX, giữ nguyên các bit khác.
- b. Đặt LSB và MSB của BL, giữ nguyên các bit khác.
- c. Đảo MSB của AL, giữ nguyên các bit khác.
- d. Xóa bốn bit cao của DL, giữ nguyên các bit khác.

3.7. Cho biết nội dung DX, AX và các cờ OF, CF sau khi thực hiện lệnh:

- a. MUL BX ; với AX = 08h, BX = 03h
- b. MUL BX ; với AX = 0FFFFh, BX = 1000h
- c. DIV BX ; với AX = 07h, BX = 02h, DX = 0h
- d. DIV CX ; với AX = 0FFFEh, CX = 10h, DX = 0h

3.8. Cho biết nội dung AX và các cờ OF, CF sau khi thực hiện lệnh:

- MUL BL ; với AL = 0ABh, BL = 10h
- MUL AH ; với AL = 0ABh, AH = 01h
- DIV BL ; với AX = 0Dh, BL = 03h
- DIV DL ; với AX = 0FEh, DL = 10h

3.9. Xem mô hình bộ nhớ trong hình C1. Cho AX = 500h, BX = 1000h, SI = 1500h, DI = 2000h, BETA là biến kiểu word nằm ở địa chỉ 1000h. Trong các lệnh sau đây, nếu hợp lệ thì hãy cho biết địa chỉ của các toán hạng nguồn hoặc thanh ghi và kết quả lưu trong toán hạng đích. Nếu không hợp lệ thì giải thích tại sao?

- MOV DL, SI
- MOV DX, [DI]
- ADD AX, [SI]
- SUB BX, [DI]
- LEA BX, BETA[BX]
- ADD [SI], [DI]
- ADD BH, [BL]
- ADD AH, [SI]
- MOV AX, [BX+DI+BETA]

| Ô nhớ | Địa chỉ |
|-------|----------|
| 03h | DS:3001h |
| 00h | DS:3000h |
| 02h | DS:2501h |
| 50h | DS:2500h |
| 02h | DS:2001h |
| 00h | DS:2000h |
| 01h | DS:1501h |
| 50h | DS:1500h |
| 01h | DS:1001h |
| 00h | DS:1000h |

Hình C1: Mô hình bộ nhớ

3.10. Với khai báo dữ liệu:

```

A      DB  1, 2, 3
B      DW  4, 5, 6
C      LABEL WORD ; C chỉ là 1 nhãn kiểu word
                        ; (không chiếm vùng nhớ)

MSG    DB  'ABC'
```

và giả sử BX chứa địa chỉ của C. Hãy cho biết lệnh nào hợp lệ (nếu không thì giải thích) và trị của các toán hạng đích sau khi thi hành lệnh trong các trường hợp sau:

- MOV AH, BYTE PTR A
- MOV AX, WORD PTR B
- MOV AX, C
- MOV AX, MSG
- MOV AH, BYTE PTR C

3.11. Dùng thanh ghi BP và kiểu định vị nền để viết các lệnh thực hiện yêu cầu sau (không được dùng các lệnh PUSH và POP)

- Đưa 0 vào hai word ở đỉnh ngăn xếp.
- Sao chép 5 word ở đỉnh ngăn xếp vào mảng ARR lần lượt theo thứ tự (nghĩa là phần tử ở đỉnh ngăn xếp vào phần tử 1 của ARR, phần tử kế trong ngăn xếp vào phần tử thứ 2 của ARR, ...)

Chương 4

HỆ THỐNG NGẮT MỀM

4.1. NHỮNG CƠ SỞ CỦA NGẮT MỀM

Một bí quyết để lập trình thành công và hiệu quả cao trên máy tính kiểu IBM-PC là việc sử dụng có hiệu quả các chương trình phục vụ của BIOS (Basic Input Output System) được cài đặt sẵn ngay trong vùng nhớ ROM-BIOS của máy tính hay các chương trình phục vụ của hệ điều hành DOS. Ngắt mềm (hay Ngắt) là cách gọi tên ngắn gọn của các chương trình phục vụ này.

Ngắt mềm là chương trình mã máy nhị phân (chính xác hơn là chương trình con) được thiết kế sẵn và tồn tại trong bộ nhớ của máy tính nhằm thực hiện các chức năng điều khiển phần cứng máy tính mỗi khi được gọi thực hiện. Khi đó, để điều khiển phần cứng nào thì người lập trình chỉ cần gọi các ngắt tương ứng để thực hiện công việc mà không cần phải biết đến cấu trúc của phần cứng là như thế nào, nguyên lý vận hành của phần cứng đó ra sao... đây là công việc rất khó khăn đối với người làm phần mềm không quen thuộc với phần cứng là những mạch điện tử.

Các ngắt mềm xem như là cầu nối giữa phần cứng và các ngôn ngữ lập trình (kể cả hệ điều hành). Chúng làm việc trực tiếp với phần cứng và các thiết bị ngoại vi, thực hiện những chức năng cơ bản nhất của hệ thống như đọc/ghi từng byte dữ liệu ở bàn phím, ra màn hình hoặc ổ đĩa từ...

Hình 4.1 trình bày các cách giao tiếp với phần cứng của ngôn ngữ lập trình thông qua các ngắt hay trực tiếp. Giao tiếp trực tiếp sẽ khó khăn hơn rất nhiều so với giao tiếp thông qua các Ngắt.

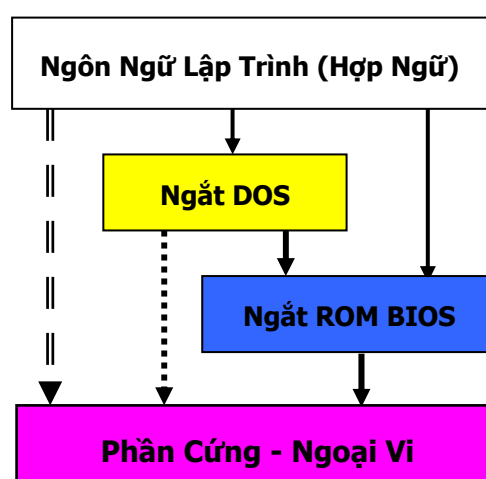
Các ngắt của BIOS thì luôn sẵn có trong máy tính IBM-PC, còn các ngắt DOS chỉ tồn tại đối với hệ điều hành DOS. Nghĩ là, nếu máy tính IBM-PC được cài đặt hệ điều hành UNIX thì trên máy đó không tồn tại các ngắt DOS. Hệ điều hành Windows được phát triển trên cơ sở của hệ điều hành DOS nên các ngắt DOS vẫn sử dụng được trên hệ điều hành Windows. Tất nhiên có một vài ngắt DOS không được phép sử dụng vì đảm bảo độ an toàn cho vận hành của hệ điều hành Windows.

Toàn bộ tài liệu về các ngắt mềm được nhiều sách về lập trình hợp ngữ nêu lên đầy đủ. Giáo trình này chỉ giới thiệu những ngắt rất cơ bản và thường dùng.

4.2. SỬ DỤNG NGẮT TRONG HỢP NGỮ

Khi sử dụng ngắt 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: Thanh ghi hay bộ nhớ do hàm qui định.
- Tham số ra: Thanh ghi hay bộ nhớ do hàm qui định.



Hình 4.1: Giao tiếp phần cứng

Lệnh gọi 1 ngắt hay 1 hàm:

MOV AH, <tên hàm>

.... ; Khai báo tham số vào (nếu có)

INT <tên ngắt>

.... ; Lưu trữ tham số ra (nếu có)

Ví dụ: Hàm 02h của ngắt 21h thực hiện việc in ký tự trong thanh ghi DL ra màn hình. Như vậy, khi muốn hiển thị ký tự B trên màn hình, ta viết như sau đoạn lệnh như sau:

MOV AH, 02h ; Chọn hàm 02h

MOV DL, 'B' ; DL chứa ký tự cần in 'B'

INT 21h ; Gọi ngắt thực hiện để in ký tự

Lưu ý: Việc sử dụng thanh ghi DL để chứa ký tự cần in là do hàm 02h qui định, không thể sử dụng thanh ghi khác để thay thế.

4.3. NGẮT MS-DOS

Các ngắt của DOS được đánh số từ 20h đến 0FFh. Ngắt quan trọng nhất là ngắt 21h – gồm nhiều chức năng (hàm) của DOS.

INT 20h: Chấm dứt chương trình và trở về DOS từ một chương trình có đuôi COM.

INT 23h: Chặn tổ hợp phím Ctrl- Break

Người sử dụng có thể chấm dứt sớm một chương trình bằng cách bấm tổ hợp phím Ctrl-Break.

Tổ hợp phím này làm CPU nhảy tới chương trình phục vụ ngắt 23h và sau đó đến một thủ tục của DOS để kết thúc chương trình đang chạy và trở về DOS.

Nếu muốn không cho phép chương trình kết thúc sớm bằng Ctrl-Break, ta có thể cho ngắt 23h trở đến một chương trình đặt biệt nào đó để thông báo lên màn hình rằng không cho chương trình đang chạy kết thúc sớm.

INT 27h: Kết thúc và đặt thường trú

Ngắt này cho phép để trong bộ nhớ trong một chương trình nào đó cho đến khi tắt máy. Để sử dụng ngắt 27h, ta cho chạy một chương trình gồm 2 phần. Một phần sẽ phụ trách việc thường trú cho phần kia.

INT 21h: Các chức năng của DOS

| | |
|----------------|---|
| Hàm 01h | Đợi đọc 1 ký tự từ bàn phím (có hiện trên màn hình) |
| Vào | Không |
| Ra | AL : Mã ASCII của phím nhận được |

Đợi đọc một ký tự từ bàn phím (phím nhận được sẽ hiển thị trên 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 02h | Hiển thị 1 ký tự tại vị trí con trỏ trên màn hình |
| <i>Vào</i> | DL ← Mã ASCII của ký tự cần hiển thị |
| <i>Ra</i> | Không |
| Hàm 03h | Đọc dữ liệu vào từ cổng nối tiếp chuẩn (COM1) |
| <i>Vào</i> | Không |
| <i>Ra</i> | AL : byte dữ liệu đọc được |
| Hàm 04h | Xuất ký tự ra cổng nối tiếp chuẩn (COM1) |
| <i>Vào</i> | DL ← ký tự cần xuất |
| <i>Ra</i> | Không |
| Hàm 05h | Xuất byte dữ liệu ra cổng máy in chuẩn (LPT1) |
| <i>Vào</i> | DL : Byte dữ liệu cần xuất |
| <i>Ra</i> | Không |
| Hàm 06h | Nhập / Xuất từ thiết bị chuẩn (không đợi) |
| <i>Vào</i> | DL ← Mã ASCII ký tự cần xuất lên màn hình Nếu DL ← 0FFh : Thực hiện chức năng nhập ký tự |
| <i>Ra</i> | Nếu ZF = 0 thì AL chứa Mã ASCII ký tự nhận được Nếu ZF = 1 thì không nhận được ký tự |
| Hàm 07h | Giống hàm 1, nhưng không hiện ký tự ra màn hình |
| <i>Vào</i> | Không |
| <i>Ra</i> | AL : Mã ASCII ký tự nhận được |
| Hàm 08h | Giống hàm 1, nhưng không hiện ký tự ra màn hình |
| <i>Vào</i> | Không |
| <i>Ra</i> | AL : Mã ASCII ký tự nhận được |
| Lưu ý: Ctrl-Break không ảnh hưởng đến hoạt động của hàm 07h và 08h | |
| Hàm 09h | Xuất 1 chuỗi ký tự lên màn hình |
| <i>Vào</i> | DS:DX ← địa chỉ logic của chuỗi ký tự cần xuất |
| <i>Ra</i> | Không |

Lưu ý: DS ← địa chỉ segment của chuỗi trong bộ nhớ

DX ← địa chỉ offset của chuỗi trong bộ nhớ

Chuỗi ký tự phải chấm dứt bằng ký tự '\$'

Ví dụ: In chuỗi "Hello Assembly" ra màn hình.

Khai báo biến chứa chuỗi trong đoạn dữ liệu DS:

chuoi DB 'Hello Assembly\$'

Viết chương trình trong đoạn lệnh:

MOV AH, 09h ; Gọi hàm 09h

LEA DX, **chuoi** ; DX ← Offset(chuoi)

INT 21h ; Thực hiện hàm

| | |
|----------------|--|
| Hàm 0Ah | Đợi đọc 1 chuỗi ký tự từ bàn phím, kết thúc bằng Enter |
| <i>Vào</i> | DS:DX ← địa chỉ của vùng đệm bàn phím trong bộ nhớ |
| <i>Ra</i> | LEN: Tổng số ký tự nhận được BUFF: Mã ASCII của các ký tự nhận được |

Hàm 0Ah nhận 1 chuỗi ký tự từ bàn phím, kết thúc bằng phím Enter (Mã ASCII là 0Dh). Hàm trả về chuỗi nhận được và chiều dài chuỗi trong vùng đệm bàn phím.

Vùng đệm bàn phím là 1 biến mảng trong bộ nhớ gồm 3 phần MAX, LEN và BUFF có cấu trúc như sau:

| | | | | | | | |
|-----|-----|------|--------|--------|-------|-------|--------|
| MAX | LEN | BUFF | BUFF+1 | BUFF+2 | | | BUFF+n |
| | | | | | | | |

MAX: Phần tử chứa số ký tự tối đa có thể nhận được gán khi khai báo.

LEN: Phần tử chứa tổng số ký tự của chuỗi nhận được, do hàm trả về sau khi nhận được chuỗi.

BUFF: Phần tử chứa mã ASCII của từng ký tự trong chuỗi nhận được (*mỗi ký tự chiếm 1 byte nhớ*). Như vậy, số byte của BUFF phải bằng hay lớn hơn giá trị của MAX.

Ví dụ: Nhận chuỗi tối đa 30 ký tự từ bàn phím

Khai báo vùng đệm trong đoạn dữ liệu:

MAX DB 30 ; 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 ; DX ← Đ/c vùng đệm bàn phím

INT 21h

Cách khai báo biến vùng đệm bàn phím khác:

BANPHIM DB 30, ?, 31 DUP(?)

| | |
|----------------|--|
| Hàm 0Bh | Cho trạng thái bàn phím |
| <i>Vào</i> | DS:DX ← địa chỉ logic của vùng đệm bàn phím trong bộ nhớ |
| <i>Ra</i> | LEN: Tổng số ký tự nhận được BUFF: Mã ASCII của các ký tự nhận được |

| | |
|----------------|--|
| Hàm 3Ch | Tạo tập tin mới |
| <i>Vào</i> | DS:DX ← Địa chỉ chuỗi tên của tập tin mới. CX ← thuộc tính của tập tin (0: bình thường, 1: chỉ đọc, 2: ẩn, 4: tập tin hệ thống) |
| <i>Ra</i> | AX : Thẻ tập tin |

| | |
|----------------|--|
| Hàm 3Dh | Mở tập tin trên đĩa |
| <i>Vào</i> | DS:DX ← Địa chỉ chuỗi tên của tập tin mới. AL ← kiểu truy xuất tập tin (0: chỉ đọc, 1: chỉ ghi, 2: đọc/ghi) |
| <i>Ra</i> | AX : Thẻ tập tin |

Khai báo tên tập tin và biến chứa thẻ file:

```
Tenf    DB    "C:\DULIEU\DATA.TXT",0
Thef    DW    ?
```

Chuỗi Tên tập tin có chứa cả tên đường dẫn của file. Nếu không có tên đường dẫn thì sẽ truy xuất trong thư mục hiện hành.

Thẻ tập tin là thành phần quan trọng trong việc truy xuất file nên sau khi mở 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 bytes.

```
MOV Thef,AX    ;Lưu trữ thẻ file vào biến.
MOV BX,Thef    ;Đưa thẻ file vào BX để truy xuất file
```

Có thể mở hay tạo nhiều tập tin khác nhau trong cùng chương trình để xử lý. Khi đó, mỗi thẻ tập tin phải được lưu trữ trong mỗi biến khác nhau.

| | |
|----------------|-----------------------|
| Hàm 3Eh | Đóng tập tin đang mở. |
| <i>Vào</i> | BX ← Thẻ tập tin |
| <i>Ra</i> | AX : Mã lỗi (nếu có) |

Sau khi xử lý xong tập tin và trước khi thoát khỏi chương trình, tập tin phải được đóng lại để cập nhật thông tin hoặc nội dung mới của tập tin.

| | |
|----------------|--|
| Hàm 3Fh | Đọc nội dung tập tin vào bộ nhớ |
| <i>Vào</i> | DS:DX ← Địa chỉ vùng nhớ chứa dữ liệu đọc được BX ← Thẻ tập tin cần đọc CX ← số byte cần đọc |
| <i>Ra</i> | AX : Số byte đọc được |

| | |
|----------------|---|
| Hàm 40h | Ghi dữ liệu trong bộ nhớ vào tập tin |
| <i>Vào</i> | DS:DX ← Địa chỉ vùng nhớ chứa dữ liệu cần ghi BX ← Thẻ tập tin cần đọc CX ← số byte cần ghi |
| <i>Ra</i> | AX : Số byte ghi được |

Các hàm 3Fh và 40h cũng có thể được sử dụng đọc hoặc ghi dữ liệu cho ngoại vi, vì DOS quản lý việc truy xuất ngoại vi bằng thẻ

| | |
|----------------|-----------------------------------|
| Hàm 41h | Xóa tập tin trên đĩa |
| <i>Vào</i> | DS:DX ← Địa chỉ chuỗi tên tập tin |
| <i>Ra</i> | AX : Mã lỗi (nếu có) |

| | |
|----------------|---|
| Hàm 42h | Đòi con trỏ tập tin hiện hành |
| <i>Vào</i> | AL ← Hướng đòi CX:DX ← Cụ ly đòi = (CX*65536) + DX BX ← Thẻ tập tin |
| <i>Ra</i> | DX:AX : Vị trí con trỏ mới = (DX*65536)+AX |

Cấu trúc dữ liệu trong tập tin:

| | | | | | | | | | | | | | |
|-----|-----|-----|---------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
| 'C' | 'h' | 'a' | 'o' | ' ' | 'C' | 'a' | 'c' | ' ' | 'B' | 'a' | 'n' | EOF | |
| | | | ↑ | | | | | | | | | | |
| | | | Con trỏ tập tin hiện hành | | | | | | | | | | |

| | |
|----------------|---|
| Hàm 43h | Đọc hoặc thay đổi thuộc tính tập tin trên đĩa |
| <i>Vào</i> | DS:DX ← Địa chỉ chuỗi tên tập tin <i>Nếu Đọc thuộc tính:</i> AL ← 0 <i>Nếu Thay đổi thuộc tính:</i> AL ← 1 và CX ← thuộc tính mới |
| <i>Ra</i> | <i>Nếu Đọc thuộc tính:</i> CX : chứa thuộc tính tập tin |

| | |
|----------------|-------------------------------------|
| Hàm 4Ch | Kết thúc chương trình và trở về DOS |
| <i>Vào</i> | Không |
| <i>Ra</i> | Không |

| | |
|----------------|---|
| Hàm 56h | Đổi tên tập tin |
| <i>Vào</i> | DS:DX ← Địa chỉ chuỗi tên tập tin cũ ES:DI ← Địa chỉ chuỗi tên tập tin mới |
| <i>Ra</i> | Không |

4.4 CÁC VÍ DỤ:

4.4.1. Chương trình nhận 1 ký tự từ bàn phím, sau đó cất ký tự nhận được vào 1 biến.

DSEG SEGMENT

```
msg DB 'Hay nhap 1 ky tu tu ban phim: $'
kytu DB ? ; Biến chứa ký tự nhận được
```

DSEG ENDS

CSEG SEGMENT

```
ASSUME CS: CSEG, DS: DSEG
```

```
start: MOV AX, DSEG
```

```
MOV DS, AX
```

```
MOV AH, 09 ; Hàm 9 in chuỗi ký tự ra màn hình
```

```
LEA DX, msg ; DX chứa địa chỉ chuỗi cần in
```

```
INT 21h ; gọi ngắt thực hiện
```

```
MOV AH, 01 ; Hàm nhập 1 ký tự, AL chứa Ký tự
```

```
INT 21h
```

```
MOV kytu, AL ; Cất Ký tự vào biến
```

```
MOV AH, 4Ch ; thoát khỏi chương trình
```

```
INT 21h
```

CSEG ENDS

```
END start
```

4.4.2. Chương trình nhận 1 ký tự từ bàn phím, sau đó in ra màn hình ký tự kế sau của ký tự nhận được.

DSEG SEGMENT

```
msg1 DB 'Hay nhap 1 ky tu tu ban phim: $'
```

```
msg2 DB 10,13,'Ky tu ke sau ky tu nhan duoc la: $'
```

```
kytu DB ?
```

DSEG ENDS

CSEG SEGMENT

```
ASSUME CS: CSEG, DS: DSEG
```

```
start: MOV AX, DSEG
```

```
MOV DS, AX
```

```
MOV AH, 09 ; Hàm 9 in chuỗi ký tự ra màn hình
```

```
LEA DX, msg1 ; DX chứa địa chỉ chuỗi msg1
```

```
INT 21h ; gọi ngắt thực hiện
```

```
MOV AH, 01 ; Hàm nhập 1 ký tự, AL chứa Ký tự
```

```
INT 21h
```

```
MOV kytu, AL ; cất ký tự vào biến
```

```
MOV AH, 09 ; Hàm 9 in chuỗi ký tự ra màn hình
```

```
LEA DX, msg2 ; DX chứa địa chỉ chuỗi msg2
```

```
INT 21h ; gọi ngắt thực hiện
```

```
MOV DL, kytu
ADD DL, 1          ; Ký tự kế sau
MOV AH, 02         ; In ký tự
INT 21h
MOV AH, 4Ch        ; thoát khỏi chương trình
INT 21h
CSEG ENDS
END start
```

4.4.3. Chương trình nhận chuỗi A và B từ bàn phím, sau đó in ra màn hình B trước, chuỗi A sau.

```
DSEG SEGMENT
    msg1 DB 'Hay nhap vao chuoi A: $'
    msg2 DB 10,13, 'Hay nhap vao chuoi B: $'
    msg3 DB 10,13, 'Cac chuoi nhan duoc la: $'
    max1 DB 30
    len1 DB ?
    strA DB 31 DUP('$')
    max2 DB 30
    len2 DB ?
    strB DB 31 DUP('$')
DSEG ENDS
CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
       MOV DS, AX
       MOV AH, 09          ; Hàm 9 in chuỗi ký tự ra màn hình
       LEA DX, msg1        ; DX chứa địa chỉ chuỗi msg1
       INT 21h             ; gọi ngắt thực hiện
       MOV AH, 0Ah         ; Nhập chuỗi A
       LEA DX, max1
       INT 21h
       MOV AH, 09          ; Hàm 9 in chuỗi ký tự ra màn hình
       LEA DX, msg2        ; DX chứa địa chỉ chuỗi msg2
       INT 21h             ; gọi ngắt thực hiện
       MOV AH, 0Ah         ; Nhập chuỗi B
       LEA DX, max2
       INT 21h
       MOV AH, 09          ; Hàm 9 in chuỗi ký tự ra màn hình
       LEA DX, msg3        ; DX chứa địa chỉ chuỗi msg3
       INT 21h             ; gọi ngắt thực hiện
       MOV AH, 09          ; Hàm 9 in chuỗi ký tự ra màn hình
```

```
    LEA DX, strB      ; DX chứa địa chỉ chuỗi B
    INT 21h           ; gọi ngắt thực hiện
    MOV AH, 09        ; Hàm 9 in chuỗi ký tự ra màn hình
    LEA DX, strA      ; DX chứa địa chỉ chuỗi A
    INT 21h           ; gọi ngắt thực hiện
    MOV AH, 4Ch       ; thoát khỏi chương trình
    INT 21h
CSEG ENDS
    END start
```

4.4.4. Chương trình ghi chuỗi “Welcome to Assembly!” vào tập tin mới có tên là W2A.TXT.

```
DSEG SEGMENT
    msg1 DB 'Da ghi xong file.$'
    msg2 DB 'Welcome to Assembly!'
    tenfile DB 'D:\W2A.TXT',0
    thefile DW ?
DSEG ENDS
CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
    MOV DS, AX
    MOV AH, 3Ch      ; Tạo file mới
    LEA DX, tenfile
    MOV CX, 0        ; thuộc tính bình thường
    INT 21h
    MOV thefile, BX ; cất thẻ file
    MOV AH, 40h      ; ghi file
    MOV BX, thefile
    MOV CX, 20        ; số byte cần ghi
    LEA DX, msg2      ; chuỗi cần ghi
    INT 21h
    MOV AH, 09        ; Hàm 9 in chuỗi ký tự ra màn hình
    LEA DX, msg1      ; DX chứa địa chỉ chuỗi msg1
    INT 21h          ; gọi ngắt thực hiện
    MOV AH, 3Eh       ; Đóng file
    MOV BX, thefile
    INT 21h
    MOV AH, 4Ch       ; thoát khỏi chương trình
    INT 21h
CSEG ENDS
    END start
```


4.4.5. Chương trình xóa tập tin trên đĩa. Tên file được nhập từ bàn phím khi chạy chương trình.

```
DSEG SEGMENT
    msg1 DB 'Ten file can xoa: $'
    msg2 DB 'Da xoa xong file.$'
    max   DB 30
    len   DB ?
    tenf  DB 31 DUP(?)
DSEG ENDS

CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
       MOV DS, AX
       MOV AH, 09      ; Hàm 9 in chuỗi ký tự ra màn hình
       LEA DX, msg1    ; DX chứa địa chỉ chuỗi msg1
       INT 21h
       MOV AH, 0Ah     ; Đọc tên file
       LEA DX, max
       INT 21h
       XOR CX, CX      ; đoạn chèn giá trị 0 vào cuối tên
       MOV CL, len     ; tập tin
       LEA SI, tenf
       ADD SI, CX
       MOV [SI], CH    ; CH = 0
       MOV AH, 41h     ; Tạo file mới
       LEA DX, tenfile
       INT 21h
       MOV AH, 09      ; Hàm 9 in chuỗi ký tự ra màn hình
       LEA DX, msg1    ; DX chứa địa chỉ chuỗi msg1
       INT 21h         ; gọi ngắt thực hiện
       MOV AH, 4Ch     ; thoát khỏi chương trình
       INT 21h
CSEG ENDS

END start
```

Chương 5

LỆNH NHẢY VÀ VÒNG LẶP

5.1. LỆNH NHẢY (CHUYỂN ĐIỀU KHIỂN)

Cách thực hiện các lệnh trong chương trình của bộ xử lý là tuần tự thực hiện các lệnh từ trên xuống, hết lệnh này đến lệnh khác. Lệnh nhảy là lệnh làm thay đổi thứ tự thực hiện lệnh của bộ xử lý. Đây cũng là cơ sở của việc xây dựng các cấu trúc rẽ nhánh trong hợp ngữ.

Có 2 loại lệnh nhảy: nhảy không có điều kiện và nhảy có điều kiện.

5.1.1. Lệnh nhảy không điều kiện

Cú pháp: **JMP** Đích : IP ← Đích

Đích: *Immed16, Reg16, Mem16*

Ý nghĩa: Luôn luôn nhảy đến thực hiện lệnh có địa chỉ là Đích. Phạm vi nhảy là trong đoạn (near jump).

Ví dụ: Xem đoạn chương trình sau

1. **MOV** AX, 1
2. **JMP** cong2 ; nhảy đến lệnh có nhãn cong2
3. cong1: **INC** AX
4. **JMP** setAX ; nhảy đến lệnh có nhãn setAX
5. cong2: **ADD** AX, 2
6. setAX:

Trong đoạn chương trình trên, các lệnh luôn được thi hành là 1, 2, 5 và 6. Còn các lệnh 3 và 4 không bao giờ được thi hành.

Các dạng của lệnh **JMP**:

- **JMP SHORT** Đích ; (short jump). Nhảy trong phạm vi từ -128 đến +127 byte so với vị trí hiện tại.

Ví dụ: **JMP SHORT** Calculate

- **JMP FAR PTR** Đích ; (far jump). Nhảy đến bất kì chỗ nào, có thể khác đoạn.

Ví dụ: **JMP FAR PTR** Calculate

- **JMP <con trỏ 2 byte>** ; (near indirect jump). Khi thực hiện, thanh ghi PC sẽ được gán bằng giá trị lưu tại địa chỉ này. Có thể kết hợp dùng với định vị chỉ số.

Ví dụ: có khai báo biến myPointer như sau:

myPointer **DW** Prepare, Calculate, Check, Output

Xem đoạn lệnh:

MOV BX, 2 ; chỉ số trong mảng con trỏ
SHL BX, 1 ; nhân đôi

```

                                JMP  myPointer[BX] ; nhảy đến Check
                                ...
Prepare:  .... ; công việc 0
                                ....
Calculate: .... ; công việc 1
                                ....
Check:  .... ; công việc 2 – nơi cần nhảy đến
                                .... ;
Output:   .... ; công việc 3
    
```

- JMP <con trỏ 4 byte> ; (far indirect jump). Tương tự trường hợp trên, nhưng con trỏ gồm cả segment và offset. Chỉ khác ở khai báo con trỏ
- JMP <thanh ghi 2 byte> ; (indirect jump via regs). Nhảy đến địa chỉ lưu trong thanh ghi AX.

Ví dụ: MOV AX, offset Calculate

JMP AX ; IP \leftarrow AX (nhảy đến lệnh có địa chỉ trong AX)

5.1.2. Lệnh nhảy có điều kiện:

Lệnh nhảy có điều kiện sẽ thực hiện việc nhảy đến nhãn hay không tùy thuộc vào điều kiện của lệnh nhảy. Lệnh nhảy có điều kiện chỉ nhảy trong phạm vi 128 byte. Muốn nhảy xa hơn, phải kết hợp với lệnh JMP.

Cú pháp tổng quát: Jxx Đích ; xx là các ký tự mô tả điều kiện

Đích: Immed, Reg, Mem

Ý nghĩa: Nếu điều kiện đúng thì nhảy tới thực hiện lệnh do toán hạng Đích, ngược lại thì tiếp tục thi hành lệnh ngay sau lệnh nhảy.

Các lệnh nhảy có điều kiện được tạo ra bằng cách ghép chữ J (JUMP) với các chữ cái mô tả điều kiện để nhảy. Các điều kiện này thường được thiết lập thông qua trạng thái thanh ghi cờ. Bảng 6.1 trình bày các lệnh nhảy thường dùng.

Ví dụ: Lệnh JNE (Jump if Not Zero), nhảy nếu không bằng, điều kiện này đúng khi ZF = 0 (gọi là NZ).

Như vậy, giá trị các cờ trạng thái hiện tại là điều kiện để lệnh nhảy xem xét khi thi hành. Thông thường, lệnh so sánh (CMP) hay lệnh TEST được sử dụng để thiết lập điều kiện lệnh nhảy.

Lệnh so sánh:

Cú pháp: CMP Trái, Phải ; Cờ \leftarrow Trái – Phải

Nếu Trái > Phải \Rightarrow Trái - Phải > 0 : CF = 0 và ZF = 0

Nếu Trái < Phải \Rightarrow Trái - Phải < 0 : CF = 1 và ZF = 0

Nếu Trái = Phải \Rightarrow Trái - Phải = 0 : CF = 0 và ZF = 1

Trái, Phải: Immed, Reg, Mem

Bản chất của lệnh CMP là lệnh SUB (thực hiện phép tính Đích – Nguồn) nhưng kết quả của phép tính không được lưu vào Đích như trong lệnh SUB mà tính chất của kết quả được thể hiện thông qua cờ

Ví dụ: so sánh hai số nguyên dương

MOV AH, 1 ; AH ← 1

MOV AL, 2 ; AL ← 2

CMP AH, AL ; CF ← 1, ZF ← 0 vì AH < AL

Sau khi thực hiện các lệnh trên, cờ Carry bật (CF=1) nghĩa là AH < AL

Ví dụ: kiểm tra hai bit cuối cùng của AL

TEST AL, 3 ; 3h = 00000011b

Nếu cờ Zero bật (ZF=1), có nghĩa là hai bit 0 và 1 của AL đều bằng 0

| Lệnh | Ý Nghĩa | Điều Kiện |
|-------------------|--|-------------------|
| JB JNAE JC | Nhảy nếu nhỏ hơn (Jump if Below) Nhảy nếu không lớn hơn hoặc bằng Nhảy nếu có nhớ | CF = 1 |
| JAЕ JNB JNC | Nhảy nếu lớn hơn hoặc bằng (Jump if Above or Equal) Nhảy nếu không nhỏ hơn Nhảy nếu không có nhớ | CF = 0 |
| JBE JNA | Nhảy nếu nhỏ hơn hoặc bằng (Jump if Below or Equal) Nhảy nếu không lớn hơn | CF = 1 hay ZF = 1 |
| JA JNBE | Nhảy nếu lớn hơn (Jump if Above) Nhảy nếu không nhỏ hơn hoặc bằng | CF = 0 và ZF = 0 |
| JE JZ | Nhảy nếu bằng (Jump if Equal) Nhảy nếu bằng (Jump if Zero) | ZF = 1 |
| JNE JNZ | Nhảy nếu không bằng (Jump if Not Equal) Nhảy nếu không bằng (Jump if Not Zero) | ZF = 0 |
| JL JNGE | Nhảy nếu nhỏ hơn Nhảy nếu không lớn hơn hoặc bằng | SF ≠ OF |
| JGE JNL | Nhảy nếu lớn hơn Nhảy nếu không nhỏ hơn | SF = OF |
| JLE JNG | Nhảy nếu nhỏ hơn hay bằng Nhảy nếu không lớn hơn | ZF=1 và SF ≠ OF |
| JG JNLE | Nhảy nếu lớn hơn Nhảy nếu không nhỏ hơn hay bằng | ZF=0 và SF = OF |
| JP JPE | Nhảy nếu có bit kiểm chẵn lẻ Nhảy nếu kiểm tra chẵn | PF = 1 |
| JNP JPO | Nhảy nếu không có bit kiểm chẵn lẻ Nhảy nếu kiểm tra lẻ | PF = 0 |
| JS JNS | Nhảy nếu có dấu Nhảy nếu không có dấu | SF = 1 SF = 0 |
| JO JNO | Nhảy nếu có tràn Nhảy nếu không có tràn | OF = 1 OF = 0 |
| JCXZ | Nhảy nếu CX = 0 | CX = 0 |

Bảng 5.1: Các lệnh nhảy có điều kiện

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 \neq 200h thì thi hành các lệnh khác

NH: ADD AX, BX ; Vị trí nhảy đến được gắn nhãn NH

Khi sử dụng lệnh nhảy có điều kiện sau khi thực hiện phép so sánh, phải đặc biệt lưu ý toán hạng trong phép so sánh là số có dấu (signed) hay không có dấu (unsigned) để lựa chọn lệnh cho phù hợp.

Ví dụ: MOV AH, AL ; Giả sử AL = 128

CMP AH, 1

JGE Greater ; AH > 1 nhưng không nhảy ????

...

Greater:

Ví dụ: nếu AL là số nguyên không dấu thì đoạn chương trình ở trên phải sửa lại như sau:

MOV AH,AL

CMP AH,1

JAE Greater

.....

Greater:

Ví dụ 5-1: Viết chương trình nhận 1 ký tự từ bàn phím, sau đó:

- Nếu ký tự nhận được là 'C' thì in ra: "Welcome to C!"
- Nếu ký tự nhận được là 'A' thì in ra: "Welcome to Assembly!"
- Nếu là ký tự khác thì thoát khỏi chương trình.

DSEG SEGMENT

msg DB 10, 13, "Hay nhap 1 ky tu: \$"

msgC DB 10, 13, "Welcome to C!\$"

msgA DB 10, 13, "Welcome to Assembly!\$"

DSEG ENDS

CSEG SEGMENT

ASSUME CS: CSEG, DS: DSEG

start: MOV AX, DSEG

MOV DS, AX

MOV AH, 09h

LEA DX, msg

INT 21h

MOV AH, 01h

INT 21h

CMP AL, 'C'

JE in_C

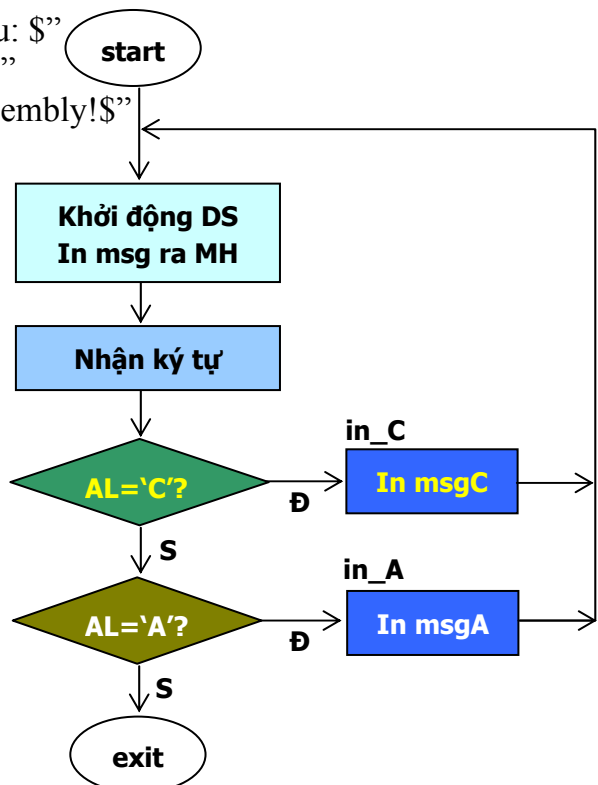
CMP AL, 'A'

JE in_A

JMP exit

in_C: MOV AH, 09h

LEA DX, msgC



Hình 6.1: Lưu đồ ví dụ 5-1

```

        INT    21h
        JMP    start
in_A:   MOV    AH, 09h
        LEA    DX, msgA
        INT    21h
        JMP    start
exit:   MOV    AH, 4Ch
        INT    21h
CSEG    ENDS
        END    start
    
```

5.2. VÒNG LẶP

Vòng lặp là một đoạn chương trình được thực hiện nhiều lần cho đến khi thỏa một điều kiện nào đó thì dừng lại, do đó vòng lặp thường kết thúc bằng một lệnh nhảy có điều kiện. Tuy nhiên, ngoài các lệnh nhảy đã biết, Intel-8086 còn cung cấp thêm các lệnh vòng lặp như LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ... các lệnh này đều có cú pháp giống nhau.

Cú pháp: LOOP *Đích* ; *Đích: Immed, Reg, Mem*

Ý nghĩa: tự động giảm CX một đơn vị, nếu CX \neq 0 thì nhảy đến *Đích*, ngược lại nếu CX = 0 thì không nhảy đến *Đích* mà thực hiện lệnh ngay sau LOOP. Nói cách khác, vòng lặp LOOP dừng lại khi CX=0. Đây là vòng lặp *for* có số lần lặp lưu trữ trong CX.

Cấu trúc vòng lặp for viết bằng LOOP:

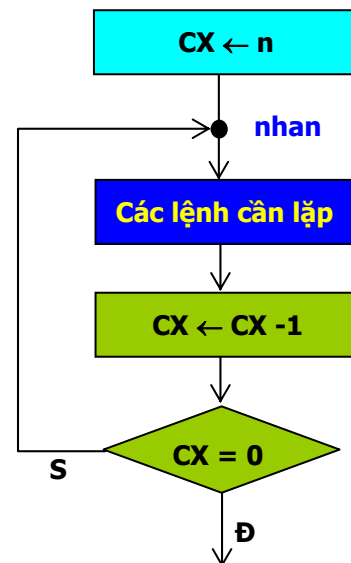
```

        MOV    CX, n ; n là số lần lặp
nhan:   ..... ; Các lệnh cần lặp lại
        .....
        LOOP   nhan
    
```

Ví dụ: Đoạn lệnh in ra màn hình các ký tự từ A đến Z

```

        MOV    DL, 'A' ; DL ← 'A'
        MOV    CX, 26 ; A→Z: 26 ký tự
inkytu: MOV    AH, 02h
        INT    21h
        INC    DL ; DL tăng lên 1 để có ký tự kế
        LOOP   inkytu
    
```



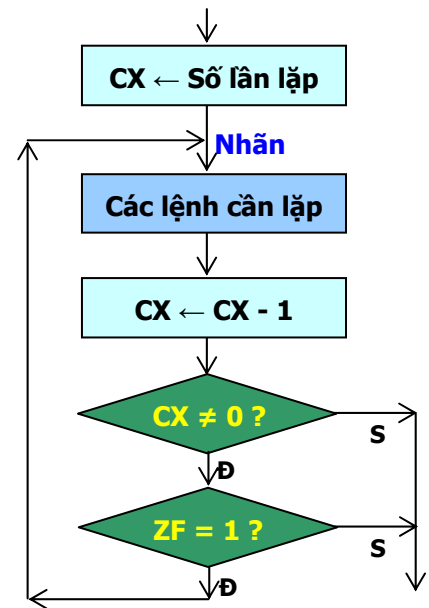
Hình 6.2: Lưu đồ LOOP

LOOPZ/LOOPE:

Tự động giảm CX một đơn vị, nếu $CX \neq 0$ và $ZF = 1$ thì nhảy đến Đích để thực hiện lệnh. Ngược lại, nếu $CX = 0$ hay $ZF = 0$ thì không nhảy, khi đó lệnh viết sau lệnh nhảy được thực hiện.

Cấu trúc vòng lặp do ... while viết bằng LOOPZ:

```
MOV CX, n
nhan: <Các lệnh cần lặp>
      LOOPZ nhan
```



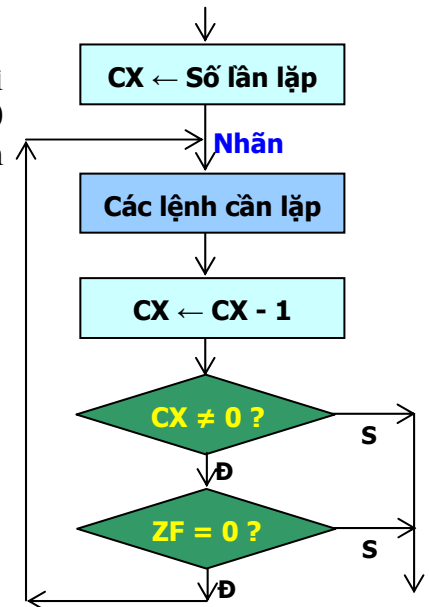
Hình 6.3: Lưu đồ LOOPE/LOOPZ

LOOPNZ/LOOPNE:

Tự động giảm CX một đơn vị, nếu $CX \neq 0$ và $ZF = 0$ thì nhảy đến Đích để thực hiện lệnh. Ngược lại, nếu $CX = 0$ hay $ZF = 1$ thì không nhảy, khi đó lệnh viết sau lệnh nhảy được thực hiện.

Cấu trúc vòng lặp do ... while viết bằng LOOPNZ:

```
MOV CX, n
nhan: <Các lệnh cần lặp>
      LOOPNZ nhan
```



Hình 6.4: Lưu đồ LOOPNE/LOOPNZ

Lưu ý: Khi sử dụng các lệnh vòng lặp cần phải chú ý đến giá trị của CX.

- Nếu $CX=0$, vì LOOP giảm CX trước khi kiểm tra nên khi thực hiện lệnh LOOP thì $CX = CX-1 = 0-1 = -1 = 0FFFFh$. Như vậy LOOP sẽ thực hiện thêm 65535 lần nữa.
- Lệnh JCXZ (xem trong bảng 5.1) nhảy khi $CX = 0$ thường được dùng để kiểm tra giá trị CX trước khi thực hiện vòng lặp.

Ví dụ: Nhập mảng A gồm 10 ký tự, dừng lại nếu gặp phím Enter

```
MOV SI, 0 ; chỉ số mảng
MOV CX, 10 ; số lần lặp
LAP: MOV AH, 1 ; nhập ký tự
      INT 21H
      MOV A[SI], AL
```

```
INC SI
CMP AL, 0Dh
LOOPNE LAP
```

Ví dụ 5-2: Viết chương trình sử dụng hàm 01/21h để nhập chuỗi ký tự dài tối đa 128 ký tự hoặc kết thúc bằng phím Enter

```
DSEG SEGMENT
    chuoi DB 128 DUP(?)
DSEG ENDS
CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
        MOV DS, AX
        LEA SI, chuoi ; SI ← địa chỉ chuỗi
        MOV CX, 128 ; Chiều dài chuỗi tối đa (số vòng lặp)
key_in: MOV AH, 01h ; Hàm nhập 1 ký tự
        INT 21h
        MOV [SI], AL ; Gán ký tự vào biến chuoi
        INC SI
        CMP AL, 0Dh ; Ký tự vừa nhập là Enter? nếu không phải Enter
        LOOPNE key_in ; hay chưa đủ 128 ký tự thì nhập tiếp
        MOV AH, 4Ch
        INT 21h
CSEG ENDS
END start
```

BÀI TẬP CHƯƠNG 5

5.1. Viết các lệnh để thực hiện các cấu trúc rẽ nhánh sau:

- IF (AX < 0) THEN BX = BX – 1
ENDIF
- IF (DL >= 'A') and (DL <= 'Z') THEN (In DL ra màn hình)
ENDIF
- IF (AX < BX) or (BX < CX) THEN DX = 0
ELSE DX = 1
ENDIF
- IF (AX < BX)
If (BX < CX) THEN AX = 0
Else BX = 0
EndIf
ENDIF

5.2. Viết chương trình đọc 1 ký tự từ bàn phím.

- Nếu ký tự nhận được là 'A' thì chuyển con trỏ về đầu dòng.
- Nếu ký tự nhận được là 'B' thì chuyển con trỏ xuống dòng.
- Nếu nhận được ký tự khác thì thoát khỏi chương trình.

5.3. Viết đoạn lệnh để tính:

a. $AX = 1 + 4 + 7 + \dots + 148 + 151$

b. $AX = 100 + 95 + 90 + \dots + 10 + 5$

5.4. Không sử dụng lệnh DIV, viết đoạn lệnh để thực hiện AX chia cho BX (nếu $BX \neq 0$), phần thương chứa trong CX, số dư chứa trong AX. Giải thuật như sau:

CX = 0

WHILE (AX >= BX)

 CX = CX + 1

 AX = AX - BX

ENDWHILE

5.5. Không sử dụng lệnh MUL, viết đoạn lệnh để thực hiện AX nhân với BX (nếu $BX \neq 0$), kết quả lưu trong CX. Giải thuật như sau:

CX = 0

DO

 CX = CX + AX

 BX = BX - 1

WHILE BX > 0

5.6. Sử dụng hàm 08h, ngắt 21h để viết chương trình nhận 1 chuỗi đầy đủ 30 ký tự từ bàn phím.

- Nếu ký tự nhận được là HOA thì hiển thị ký tự đó lên màn hình.
- Nếu ký tự nhận được là thường thì hiển thị dấu '*' lên màn hình.

5.7. Viết chương trình nhập 1 chuỗi tối đa 256 ký tự từ bàn phím. Sau đó in đảo ngược chuỗi nhận được ra màn hình.

5.8. Viết chương trình nhận 1 chuỗi ký tự thường từ bàn phím. Sau đó đổi chuỗi nhận được thành chuỗi ký tự HOA và in ra màn hình.

5.9. Viết chương trình nhận 1 chuỗi ký tự từ bàn phím. Sau đó đếm số CHỮ có trong chuỗi nhận được và in ra màn hình số đếm được.

Chương 6

NGĂN XẾP VÀ CHƯƠNG TRÌNH CON

Ngăn xếp là vùng nhớ lưu trữ tạm thời dữ liệu cho chương trình hoặc lưu trữ địa chỉ trở về từ chương trình con (còn gọi là thủ tục).

6.1. NGĂN XẾP

6.1.1. Tổ chức và vận hành

Ngăn xếp là vùng nhớ đặc biệt trong bộ nhớ có cách truy xuất đặc biệt khác hẳn với việc truy xuất ngẫu nhiên các ô nhớ trong bộ nhớ. Hệ điều hành sẽ cấp phát vùng nhớ cho ngăn xếp là vùng nhớ có địa chỉ cao nhất trong bộ nhớ, mỗi chương trình khi thi hành trên máy tính sẽ có ngăn xếp riêng cho chương trình đó.

Việc truy xuất nội dung ngăn xếp theo cơ chế “Vào sau, ra trước” (Last In First Out – LIFO) nghĩa là dữ liệu nào được đưa vào sau cùng sẽ được lấy ra trước.

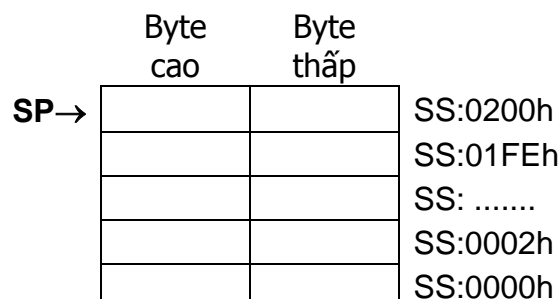
Ngăn xếp được tổ chức thành mảng nhiều phần tử, mỗi phần tử là 2 byte (word). Kích thước của ngăn xếp phụ thuộc vào chương trình và do người viết chương trình xác định bằng cách khai báo đoạn ngăn xếp.

Ví dụ: Khai báo ngăn xếp 256 phần tử có tên là SSEG

```
SSEG SEGMENT STACK 'STACK'
      DW 256 DUP(?)
SSEG SEGMENT
```

Địa chỉ logic của đỉnh ngăn xếp trong bộ nhớ được xác định bằng con trỏ ngăn xếp SS:SP (SS chứa địa chỉ đoạn ngăn xếp. SP chứa địa chỉ độ dài của đỉnh ngăn xếp). Đỉnh của ngăn xếp khi mới khởi tạo (gọi là đáy của ngăn xếp) luôn luôn là phần tử có địa chỉ cao nhất trong đoạn ngăn xếp.

Hình 6.1 mô tả vùng nhớ đoạn ngăn xếp trong ví dụ trên và giá trị của con trỏ ngăn xếp khi mới khởi tạo là phần tử cao nhất trong đoạn ngăn xếp (SP = 0200h).



Hình 6.1: Mô hình Đoạn Stack gồm 256 phần tử (256 word)

Khi cất dữ liệu vào ngăn xếp, SP giảm 2 trước khi lưu trữ dữ liệu vào. Khi lấy dữ liệu ra khỏi ngăn xếp, thì dữ liệu được đọc ra trước sau đó SP mới tăng lên 2 để chỉ phần tử kế tiếp trong ngăn xếp.

6.1.2. Truy xuất ngăn xếp

Cú pháp:

| | | |
|-------|-------|--|
| PUSH | Nguồn | ; $SP \leftarrow SP - 2, \text{Mem}[SP] \leftarrow \text{Nguồn}$ |
| PUSHF | | ; $SP \leftarrow SP - 2, \text{Mem}[SP] \leftarrow \text{Flag}$ |
| POP | Đích | ; $\text{Đích} \leftarrow \text{Mem}[SP], SP \leftarrow SP + 2$ |
| POPF | | ; $\text{Flag} \leftarrow \text{Mem}[SP], SP \leftarrow SP + 2$ |

Nguồn, Đích: Reg16 hay Mem16.

Ý nghĩa: - Lệnh PUSH giảm con trỏ ngăn xếp (SP) xuống 2, sau đó lưu trữ toán hạng nguồn vào ngăn xếp.

- Lệnh PUSHF lưu giữ thanh ghi cờ vào ngăn xếp.

- Lệnh POP lấy 2 byte dữ liệu tại đỉnh ngăn đưa vào toán hạng đích, sau đó tăng SP lên 2.

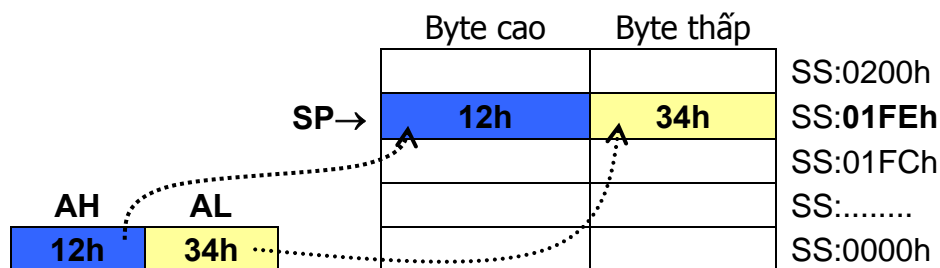
- Lệnh POPF lấy 2 byte từ ngăn xếp đưa vào thanh ghi cờ.

Ví dụ:

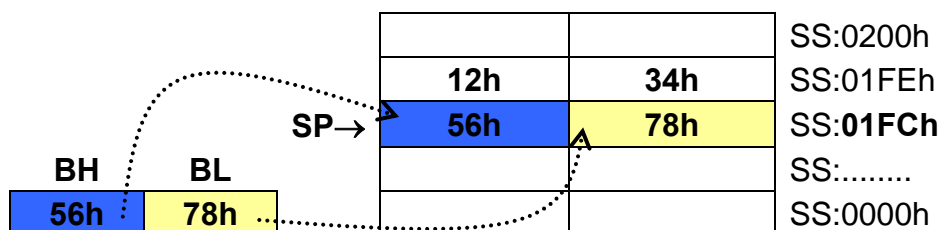
```

MOV AX, 1234h
MOV BX, 5678h
PUSH    AX    ;  $SP \leftarrow SP - 2, \text{Mem}[SP] \leftarrow AX$ 
PUSH BX    ;  $SP \leftarrow SP - 2, \text{Mem}[SP] \leftarrow BX$ 
POP     DX    ;  $DX \leftarrow \text{Mem}[SP], SP \leftarrow SP + 2$ 
    
```

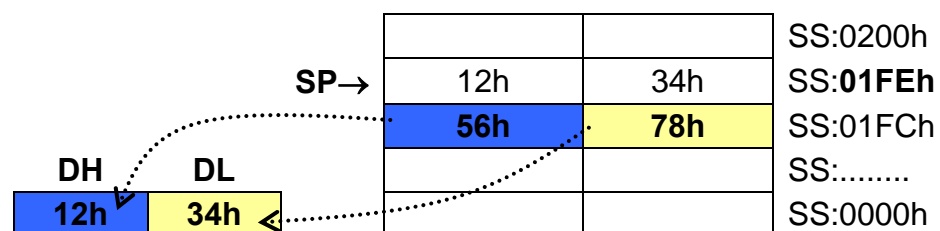
Hình 6.2 lần lượt mô tả hoạt động của ngăn xếp ứng với 3 lệnh truy xuất ngăn xếp trong ví dụ trên.



Hình 6.2a: PUSH AX (với AX=1234h)



Hình 6.2b: PUSH BX (với BX=5678h)



Hình 6.2c: POP DX (DX ← 5678h)

Ghi chú: Qua hai lệnh PUSH và POP, ta thấy ngăn xếp đi từ ô nhớ có địa chỉ cao đến ô nhớ có địa chỉ thấp, nghĩa là số liệu đưa vào ngăn xếp trước thì ở địa chỉ cao và số liệu đưa vào ngăn xếp sau thì ở địa chỉ thấp hơn.

Ví dụ 6-1: Chương trình nhập 1 chuỗi từ bàn phím, sau đó in đảo ngược chuỗi nhận được ra màn hình sử dụng giải thuật của ngăn xếp (*Ký tự nhập vào sau cùng được in ra trước*)

```
DSEG SEGMENT
    msg1 DB    "Hay nhap chuoi ky tu, ket thuc bang Enter: $"
    msg2 DB    10, 13, "Chuoi dao nguoc la: $"
DSEG ENDS
SSEG SEGMENT STACK 'STACK'
    DW 256 DUP(?)
SSEG ENDS
CSEG SEGMENT
    ASSUME     CS: CSEG, DS: DSEG, SS: SSEG
start: MOV AX, DSEG
      MOV DS, AX
      MOV AH, 09h
      LEA DX, msg1
      INT 21h
      XOR CX, CX
nhap:  MOV AH, 01
      INT 21h
      CMP AL, 0Dh    ; Có phải phím Enter không?
      JZ  inra        ; phải thì dừng, không phải thì nhập tiếp
      PUSHAX          ; Cất ký tự trong AL vào ngăn xếp
      INC CX          ; đếm số ký tự nhập
      JMP  nhap        ; nhập tiếp ký tự
inra:  MOV AH, 09h
      LEA DX, msg2
      INT 21h
intiep: MOV AH, 02
      POP DX          ; Lấy ký tự trong ngăn xếp ra DL để in
      INT 21h
      LOOP intiep
      MOV AH, 4Ch
      INT 21h
CSEG ENDS
      END start
```

6.2. CHƯƠNG TRÌNH CON

6.2.1. Khai báo chương trình con (Thủ tục)

Thủ tục là 1 đoạn chương trình có nhiệm vụ tương đối độc lập được sử dụng nhiều nơi trong chương trình chính. Thực chất, chương trình con hay thủ tục chỉ là 1 phần lệnh được viết riêng, giúp cho chương trình dễ đọc, linh hoạt và dễ bảo trì. Thủ tục thường được viết ở cuối chương trình, trong đoạn lệnh và có cấu trúc như sau:

TênThủTục **PROC** [kiểu]

..... ; Các lệnh trong thủ tục

RET ; chấm dứt thủ tục và trở về nơi gọi thủ tục.

TênThủTục **ENDP**

TênThủTục là một nhãn được người lập trình đặt theo qui cách đặt tên trong hợp ngữ. [kiểu] có thể là NEAR hay FAR dùng để xác định phạm vi của lệnh gọi thủ tục cùng hay khác đoạn với thủ tục. Nếu không khai báo rõ kiểu, thì mặc nhiên là NEAR.

RET (Return) là lệnh kết thúc thủ tục và trở về nơi gọi thủ tục để tiếp tục thi hành các lệnh sau lệnh gọi thủ tục trong chương trình. Lệnh RET sẽ lấy 2 byte địa chỉ trở về đang lưu trữ trong ngăn xếp để nạp vào thanh ghi IP (tương đương lệnh POP).

$$\text{RET} \Leftrightarrow \begin{cases} \text{IP} \leftarrow \text{M}[\text{SS}:\text{SP}] \\ \text{SP} \leftarrow \text{SP} + 2 \end{cases}$$

6.2.3. Gọi thủ tục

Cú pháp: CALL *Đích*; $\Leftrightarrow \begin{cases} \text{SP} \leftarrow \text{SP} - 2 \\ \text{M}[\text{SS}:\text{SP}] \leftarrow \text{IP} \\ \text{IP} \leftarrow \text{Đích} \end{cases}$

Đích: tên thủ tục hay địa chỉ thủ tục

Khi thực hiện lệnh CALL, địa chỉ của lệnh ngay sau lệnh CALL (IP đang chứa địa chỉ này) được cất vào ngăn xếp, sau đó địa chỉ của thủ tục được nạp vào IP, do đó lệnh thi hành sau lệnh CALL sẽ là lệnh đầu tiên trong thủ tục được gọi. Như vậy, lệnh CALL cũng thực hiện thao tác nhảy gần giống như lệnh nhảy.

Thủ tục không có cơ chế truyền tham số vào trực tiếp trên dòng lệnh gọi thủ tục. Vì thế việc truyền tham số vào cho thủ tục phải thông qua các thanh ghi hay biến xác định trước khi gọi thủ tục. Đó đó, khi viết thủ tục, phải tự chọn thanh ghi hay biến làm tham số vào.

6.3. CÁC VÍ DỤ

6.3.1. Viết lại ví dụ 5-1, trong đó việc in chuỗi ra màn hình được thực hiện bằng thủ tục inchuoi.

DSEG SEGMENT

msg DB "Hay nhập 1 ky tu: \$"

msgC DB "Welcome to C!\$"

msgA DB "Welcome to Assembly!\$"

DSEG ENDS

CSEG SEGMENT

ASSUME CS: CSEG, DS: DSEG

start: MOV AX, DSEG

MOV DS, AX

LEA DX, msg

CALL inchuoi

MOV AH, 01h

```

        INT    21h
        CMP    AL, 'C'
        JE     in_C
        CMP    AL, 'A'
        JE     in_A
        JMP    exit
in_C:   LEA     DX, msgC
        CALL    inchuoi
        JMP     start
in_A:   LEA     DX, msgA
        CALL    inchuoi
        JMP     start
exit:   MOV     AH, 4Ch
        INT     21h
inchuoi PROC
        PUSH    DX
        MOV     AH, 02h    ; Xuống dòng trước khi in chuỗi
        MOV     DL, 13
        INT     21h
        MOV     DL, 10
        INT     21h
        POP     DX        ; DX chứa địa chỉ chuỗi cần in và
        MOV     AH, 09h    ; chính là tham số vào của thủ tục.
        INT     21h
        RET
inchuoi ENDP
CSEG ENDS
        END     start

```

6.3.2. Viết lại ví dụ 5-1, trong đó việc in chuỗi ra màn hình được thực hiện bằng macro `in Chuoi` có tham số vào là biến chuỗi cần in. (cú pháp *MACRO* được trình bày trong chương 2)

```

in Chuoi MACRO    Chuoi
        MOV     AH, 02h    ; Xuống dòng trước khi in chuỗi
        MOV     DL, 13
        INT     21h
        MOV     DL, 10
        INT     21h        ; DX chứa địa chỉ
        LEA     DX, Chuoi ; biến chuỗi cần in là tham số vào Macro
        MOV     AH, 09h
        INT     21h
        ENDM

```

```

DSEG SEGMENT
    msg DB "Hay nhap 1 ky tu: $"
    msgC DB "Welcome to C!$"
    msgA DB "Welcome to Assembly!$"
DSEG ENDS
CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
        MOV DS, AX
        inchoi msg
        MOV AH, 01h
        INT 21h
        CMP AL, 'C'
        JE in_C
        CMP AL, 'A'
        JE in_A
        JMP exit
in_C: inchoi msgC
        JMP start
in_A: inchoi msgA
        JMP start
exit: MOV AH, 4Ch
        INT 21h
CSEG ENDS
        END start
    
```

6.3.3. Viết lại ví dụ 6-1, trong đó việc nhập chuỗi ký tự và xuất đảo ngược chuỗi ký tự được viết bằng thủ tục. In chuỗi bằng Macro

```

inchoi MACRO chuoi
    MOV AH, 02h ; Xuống dòng trước khi in chuỗi
    MOV DL, 13
    INT 21h
    MOV DL, 10
    INT 21h ; DX chứa địa chỉ
    LEA DX, chuoi ; biến chuỗi cần in là tham số vào Macro
    MOV AH, 09h
    INT 21h
ENDM

DSEG SEGMENT
    msg1 DB "Hay nhap chuoi ky tu, ket thuc bang Enter: $"
    msg2 DB "Chuoi dao nguoc la: $"
    sokt DW ?
DSEG ENDS
SSEG SEGMENT STACK 'STACK'
    
```

```
DW 256 DUP(?)
SSEG ENDS
CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG, SS: SSEG
start: MOV AX, DSEG
        MOV DS, AX
        inchoi msg1
        CALL nhapchoi
        inchoi msg2
        CALL daochoi
        MOV AH, 4Ch
        INT 21h
nhapchoi PROC
        POP BX
        XOR CX, CX
        nhap: MOV AH, 01
                INT 21h
                CMP AL, 0Dh ; Có phải phím Enter không?
                JZ stop ; phải thì dừng, không phải thì nhập tiếp
                PUSH AX ; Cất ký tự trong AL vào ngăn xếp
                INC CX ; đếm số ký tự nhập
                JMP nhap ; nhập tiếp ký tự
        stop: MOV sokt, CX ; cất số ký tự đã nhập
                PUSH BX
                RET
nhapchoi ENDP
daochoi PROC
        POP BX
        MOV CX, sokt
        intiep: MOV AH, 02
                POP DX ; Lấy ký tự trong ngăn xếp ra DL để in
                INT 21h
                LOOP intiep
                PUSH BX
                RET
daochoi ENDP
CSEG ENDS
        END start
```


BÀI TẬP CHƯƠNG 6

6.1. Cho $AX = 1234h$, $BX = 5678h$, $CX = 9ABCh$ và $SP = 1000h$. Hãy cho biết nội dung AX , BX , CX và SP sau khi thực hiện xong mỗi lệnh sau đây và vẽ mô hình ngăn xếp để minh họa quá trình thay đổi dữ liệu trong ngăn xếp.

```
PUSH AX
PUSH BX
XCHG AX, CX
POP CX
PUSH AX
POP BX
```

6.2. Vẽ mô hình ngăn xếp minh họa quá trình thay đổi dữ liệu trong ngăn xếp cho chương trình ví dụ 6-1.

6.3. Giả sử $SP = 0200h$ và nội dung đỉnh ngăn xếp là $012Ah$, Hãy cho biết trị của IP và SP sau khi thực hiện xong lệnh RET .

6.4. Vẽ mô hình ngăn xếp minh họa quá trình thay đổi dữ liệu trong ngăn xếp cho chương trình ví dụ 6.3.1

6.5. Với 2 lệnh sau đây, và giả định MOV nằm ở địa chỉ $08FD:0203h$, $PROC_1$ là thủ tục $NEAR$ tại địa chỉ $08FD:0300h$, $SP = 010AH$. Hãy cho biết nội dung IP và SP sau mỗi lệnh.

```
CALL PROC_1
MOV AX, BX
```

6.6. Viết chương trình nhập từ bàn phím một biểu thức đại số có chứa các dấu ngoặc tròn $()$ hay $[]$. Sau đó kiểm tra biểu thức nhận được là hợp lệ hay không hợp lệ và in kết quả ra màn hình.

Ví dụ: $[a + (b - [c * (d - e)]) + f]$ là hợp lệ

$[a + (b - [c * (d - e)) + f)]$ là không hợp lệ.

Hướng dẫn: dùng ngăn xếp để $PUSH$ các dấu ngoặc trái ‘(’, ‘[’ vào ngăn xếp. Nếu gặp dấu ngoặc phải ‘)’, ‘]’ thì POP từ stack ra để so sánh. Nếu không POP được, hoặc POP ra không đúng loại với dấu ngoặc phải là không hợp lệ. Ngược lại là hợp lệ.

Chương 7

XỬ LÝ KÝ SỐ VÀ XỬ LÝ CHUỖI

7.1. XỬ LÝ KÝ TỰ

Như đã biết, việc xuất nhập trong hợp ngữ chỉ là xuất nhập 1 ký tự hay 1 chuỗi ký tự thông qua mã ASCII của nó và đó cũng chính là cơ chế hoạt động của bàn phím và màn hình. Do đó, khi cần nhập hay xuất các dạng số (nhị phân, thập phân, thập lục phân) thì phải xử lý các ký tự số (còn gọi là ký số) thành số sau khi nhập và xử lý số thành ký số để xuất ra màn hình.

7.1.1. Nhập xuất số nhị phân (Binary)

Số nhị phân nhập từ bàn phím là 1 chuỗi ký tự bao gồm các ký số '0' và '1'. Mỗi ký số đó được đưa vào máy tính ở dạng mã ASCII (8 bit), phải được xử lý lại thành dạng 1 bit. Như vậy:

- Ký số '0', mã ASCII là 30h, phải xử lý thành 1 bit 0.
- Ký số '1', mã ASCII là 31h, phải xử lý thành 1 bit 1.

Trước khi xuất số nhị phân ra màn hình thì phải xử lý ngược lại, nghĩa là lấy từng bit để đổi thành ký số tương ứng:

- Bit 0 phải xử lý thành mã ASCII là 30h (Ký số '0').
- Bit 1 phải xử lý thành mã ASCII là 31h (Ký số '1').

Ví dụ: Đoạn chương trình nhập 1 số nhị phân 8 bit từ bàn phím, lưu trữ trong thanh ghi BL. Sử dụng hàm 01/21h để nhập từng ký số.

```
MOV BL, 0          ; Xóa BL
MOV CX, 8          ; nhập đủ 8 bit thì dừng
nhap: MOV AH, 01h   ; Hàm nhập ký tự
      INT 21h
      CMP AL, 0Dh   ; nếu là phím Enter thì thôi nhập
      JZ  exit      ; không phải Enter thì đổi sang bit
      SHL BL, 1     ; Dịch trái BL 1 bit
      SUB AL, 30h    ; Ký số - 30h = số
      ADD BL, AL     ; Chuyển bit từ AL sang BL lưu trữ
      LOOP nhap
exit:  .....
```

Ví dụ: Đoạn chương trình xuất số nhị phân 8 bit trong BL ra màn hình. Sử dụng hàm 02/21h để xuất từng ký số.

```
MOV CX, 8          ; Xuất 8 bit
xuat: MOV DL, 0
      SHL BL, 1     ; CF chứa MSB, xuất ra màn hình
      RCL DL, 1     ; đưa CF vào LSB của DL
      ADD DL, 30h   ; Số + 30h = Ký số
```

MOV AH, 02h ; In ra màn hình
INT 21h
LOOP xuất

7.1.2. Nhập xuất số thập lục phân (Hexa)

Giải thuật nhập/xuất số thập lục phân cũng gần giống như số nhị phân. Cần lưu ý rằng:

- Các chữ số thập lục phân bao gồm: '0' ... '9' và 'A' ... 'F'. Khi đó 'A' chuyển thành 0Ah, 'F' chuyển thành 0Fh. Còn '0' đến '9' thì giống như trường hợp nhị phân.
- Nhưng mỗi số thập lục phân là 4 bit nhị phân.

Ví dụ: Đoạn chương trình nhập từ bàn phím số thập lục phân 16 bit (4 chữ số thập lục phân) vào thanh ghi BX. Sử dụng hàm 01/21h để nhập.

Giải thuật nhập:

BX ← 0
lap: Nhập ký tự
Nếu ký tự là ký số thập lục phân:
 Đổi thành số tương ứng
 Dịch trái BX 4 bit
 Đưa trị đã đổi vào 4 bit thấp của BX
Nhảy về **lap** cho đến khi ký tự nhập là Enter

Đoạn chương trình thể hiện giải thuật trên:

```
MOV CL, 4
XOR BX, BX
nhap: MOV AH, 01
      INT 21h
      CMP AL, 0Dh
      JZ exit
      CMP AL, 39h ; Đổi ký số thành số tương ứng
      JA kytu
      SUB AL, 30h
      JMP save
kytu: SUB AL, 37h
save: SHL BX, CL
      ADD BL, AL
      JMP nhap
exit: .....
```

Ví dụ: Đoạn chương trình xuất giá trị BX ra màn hình ở dạng số thập lục phân (4 chữ số thập lục phân). Sử dụng hàm 02/21h để xuất.

Giải thuật xuất:

Lặp 4 lần: DL ← BH
 Dịch phải DL 4 bit
 Nếu DL < 10
 Đổi thành ký số '0' ... '9' tương ứng
 Nếu DL ≥ 10

Đổi thành ký tự 'A' 'F' tương ứng
In ra màn hình ký tự trong DL
Quay trái BX 4 bit

Đoạn chương trình thể hiện giải thuật trên:

```

MOV CX, 4
xuất: PUSH CX
      MOV CL, 4
      MOV DL, BH
      SHR DL, CL
      CMP DL, 09h
      JA kytu
      ADD DL, 30h ; Đổi thành ký số '0' ... '9' tương ứng
      JMP inra
kytu: ADD DL, 37h ; Đổi thành ký tự 'A' .... 'F' tương ứng
inra:  MOV AH, 02h ; In ra màn hình ký tự đã đổi
      INT 21h
      SHL BX, CL ; Quay trái BX 4 bit
      POP CX
      LOOP xuất
    
```

7.2. XỬ LÝ CHUỖI

7.2. LỆNH XỬ LÝ CHUỖI

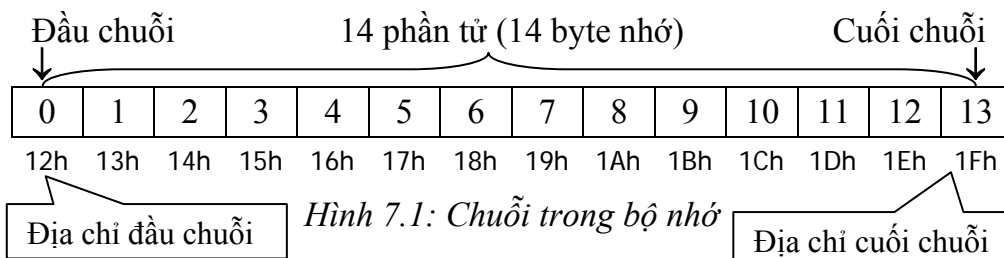
Khái niệm chuỗi trong máy tính không giới hạn ở chuỗi ký tự, mà là khái niệm mẫng gồm nhiều phần tử, kiểu dữ liệu của phần tử là byte hay word. Các phần tử có thể chứa ký tự hay số liệu. Do đó, các lệnh thao tác trên chuỗi cho phép thao tác trên các mẫng hay bất kỳ vùng đệm dữ liệu nào.

Chuỗi lưu trữ trong bộ nhớ có địa chỉ đầu và địa chỉ cuối chính là địa chỉ của phần tử đầu tiên và phần tử cuối trong chuỗi. Như vậy, thông số của 1 chuỗi trong bộ nhớ bao gồm: Địa chỉ đầu, địa chỉ cuối, số phần tử của chuỗi phải thỏa mãn công thức sau:

$$(\text{Số byte của phần tử} \times \text{số phần tử}) = \text{ĐC cuối} - \text{ĐC đầu} + 1$$

Trong đó: $(\text{Số byte của phần tử} \times \text{số phần tử}) = \text{Số byte của chuỗi}$

Hình 7.1 mô tả chuỗi gồm 14 phần tử, mỗi phần tử là 1 byte được lưu trữ trong bộ nhớ bắt đầu tại địa chỉ 12h.



Hình 7.1: Chuỗi trong bộ nhớ

Intel-8086 cung cấp nhiều lệnh xử lý chuỗi để thực hiện các thao tác như: chuyển chuỗi, so sánh chuỗi, dò tìm trong chuỗi Khi sử dụng những lệnh này thì việc viết chương trình sẽ ngắn hơn và thi hành nhanh hơn là sử dụng các lệnh MOV, CMP ... trong các thao tác chuỗi. Các lệnh xử lý chuỗi gồm 3 nhóm trong bảng 7.1.

Các lệnh xử lý chuỗi không có toán hạng trên dòng lệnh, nên việc sử dụng các toán hạng mặc nhiên phải tuân thủ qui định của từng lệnh.

| LỆNH | Ý NGHĨA |
|--|---|
| Nhóm di chuyển chuỗi | |
| MOVSB | Di chuyển chuỗi từng byte (Move String Byte) |
| MOVSW | Di chuyển chuỗi từng word (Move String Word) |
| LODSB | Nạp chuỗi từng byte (Load String Byte) |
| LODSW | Nạp chuỗi từng word (Load String Word) |
| STOSB | Ghi chuỗi từng byte (Store String Byte) |
| STOSW | Ghi chuỗi từng word (Store String Word) |
| Nhóm so sánh chuỗi | |
| CMPSB | So sánh chuỗi từng byte (Compare String Byte) |
| CMPSW | So sánh chuỗi từng word (Compare String Word) |
| Nhóm dò tìm giá trị trong chuỗi | |
| SCASB | Do tìm trong chuỗi từng byte (Scan String Byte) |
| SCASW | Do tìm trong chuỗi từng word (Scan String Word) |

Bảng 7.1: Lệnh xử lý chuỗi

7.2.1. Hướng xử lý chuỗi

Khi xử lý 1 chuỗi có nghĩa là xử lý lần lượt các phần tử trong chuỗi, hết phần tử này đến phần tử khác cho đến khi hết chuỗi. Tùy vào đặc điểm của chuỗi, có hai hướng xử lý:

- Hướng tăng (từ trái qua phải) là xử lý phần tử đầu tiên trước và lần lượt đến phần tử cuối. Khi đó địa chỉ các phần tử sẽ *tăng* dần từ địa chỉ đầu cho đến địa chỉ cuối.
- Hướng giảm (từ phải qua trái) là xử lý phần tử cuối trước và lần lượt sau đó mới đến các phần tử đầu. Khi đó địa chỉ các phần tử sẽ *giảm* dần từ địa chỉ cuối cho đến địa chỉ đầu.

Thái thái cờ Hướng (DF) dùng để chọn hướng xử lý của chuỗi, do đó trước khi thực hiện lệnh xử lý chuỗi, phải chọn hướng xử lý chuỗi thích hợp bằng các lệnh thiết lập trạng thái DF như sau:

CLD ; DF = 0 : Hướng tăng

STD ; DF = 1 : Hướng giảm

7.2.2. Các tiền tố lặp REP (Repeat)

Tiền tố REP có thể đặt trước các lệnh xử lý chuỗi như sau:

REP <Lệnh xử lý chuỗi>

Khi đó các lệnh xử lý chuỗi sẽ được lập lại với số lần lập xác định trong CX và sau mỗi lần lập, CX tự động giảm 1 cho đến khi CX = 0 thì kết thúc vòng lặp (Trong trường hợp này cũng có thể sử dụng lệnh vòng lặp LOOP, nhưng lệnh sẽ dài dòng hơn).

Ví dụ: Đoạn lệnh thực hiện MOVSB lập lại 10 lần, dùng REP và LOOP như sau:

REP
MOV CX, 10
REP MOVSB

LOOP
MOV CX, 10
lap: MOVSB
LOOP lap

Ngoài ra còn có các tiền tố lặp tương tự như REP (thường dùng cho các lệnh SCASB, SCASW, CMPSB, CMPSW) như bảng 7.2 sau:

| Lệnh | Ý Nghĩa |
|----------------|--|
| REPZ REPE | Lặp lại lệnh theo sau nó nếu CX \neq 0 và ZF = 1. Khi CX = 0 hay ZF = 0 sẽ không thực hiện vòng lặp. |
| REPNZ REPNE | Lặp lại lệnh theo sau nó nếu CX \neq 0 và ZF = 0. Khi CX = 0 hay ZF = 1 sẽ không thực hiện vòng lặp. |

Bảng 7.2: Ý nghĩa các tiền tố lặp

7.2.3. Lệnh Ghi vào chuỗi

STOSB sẽ ghi nội dung AL (1 byte) vào 1 phần tử trong chuỗi đích có địa chỉ xác định bởi ES:DI.

STOSW sẽ ghi nội dung AX (2 byte) vào 1 phần tử trong chuỗi đích có địa chỉ xác định bởi ES:DI.

| STOSB | STOSW |
|---|---|
| Mem[ES:DI] \leftarrow AL IF (DF = 0) DI \leftarrow DI + 1 ELSE DI \leftarrow DI - 1 | Mem[ES:DI] \leftarrow AX IF (DF = 0) DI \leftarrow DI + 2 ELSE DI \leftarrow DI - 2 |

Ví dụ 7-1: Viết lại chương trình ở ví dụ 5-2, trong đó sử dụng lệnh STOSB để lưu các ký tự nhận được từ bàn phím vào biến chuỗi.

```
DSEG  SEGMENT
        chuoi  DB   128 DUP(?)
DSEG  ENDS
CSEG  SEGMENT
        ASSUME CS: CSEG, DS: DSEG, ES: DSEG
start:  MOV  AX, DSEG
        MOV  DS, AX
        MOV  ES, AX
        LEA  DI, chuoi      ; SI  $\leftarrow$  địa chỉ chuỗi
        MOV  CX, 128        ; Chiều dài chuỗi tối đa (số vòng lặp)
key_in: MOV  AH, 01h        ; Hàm nhập 1 ký tự
        INT  21h
        STOSB               ; Gán ký tự vào biến chuoi
        CMP  AL, 0Dh        ; Ký tự vừa nhập là Enter? nếu không phải Enter
                        ; hay chưa đủ 128 ký tự thì nhập tiếp
        LOOPNE key_in
        MOV  AH, 4Ch
        INT  21h
```

CSEG ENDS
END start

7.2.4. Lệnh Nạp từ chuỗi

LODSB sẽ nạp 1 byte trong chuỗi nguồn có địa chỉ xác định bởi DS:SI vào thanh ghi AL.

LODSW sẽ nạp 2 byte trong chuỗi nguồn có địa chỉ xác định bởi DS:SI vào thanh ghi AX.

| LODSB | LODSW |
|---|---|
| AL \leftarrow Mem[DS:SI] IF (DF = 0) SI \leftarrow SI + 1 ELSE SI \leftarrow SI - 1 | AX \leftarrow Mem[DS:SI] IF (DF = 0) SI \leftarrow SI + 2 ELSE SI \leftarrow SI - 2 |

Ví dụ 7-2: Viết chương trình dùng hàm 02/21h để in chuỗi “Welcome to Assembly” ra màn hình, trong đó sử dụng lệnh LODSB để nạp từng ký tự trong chuỗi vào AL

```
DSEG SEGMENT
    chuỗi DB    “Welcome to Assembly”    ; chuỗi gồm 19 ký tự
DSEG ENDS
CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
       MOV DS, AX
       LEA SI, chuỗi    ; SI  $\leftarrow$  địa chỉ chuỗi
       MOV CX, 19       ; Chiều dài chuỗi
key_in: LODSB           ; nạp từng byte trong chuỗi vào AL
       MOV DL, AL
       MOV AH, 02h      ; Hàm nhập in 1 ký tự
       INT 21h
       LOOP key_out ;
       MOV AH, 4Ch
       INT 21h
CSEG ENDS
END start
```

7.2.5. Lệnh di chuyển chuỗi

MOVS sẽ chuyển 1 byte trong chuỗi nguồn có địa chỉ xác định bởi DS:SI đến 1 phần tử trong chuỗi đích có địa chỉ xác định bởi ES:DI.

MOVSW sẽ chuyển 2 byte trong chuỗi nguồn có địa chỉ xác định bởi DS:SI đến 2 byte nhớ trong chuỗi đích có địa chỉ xác định bởi ES:DI.

| MOVSB | MOVSW |
|--|--|
| Mem[ES:DI] ← Mem[DS:SI] IF (DF = 0) SI ← SI + 1 DI ← DI + 1 ELSE SI ← SI - 1 DI ← DI - 1 | Mem[ES:DI] ← Mem[DS:SI] IF (DF = 0) SI ← SI + 2 DI ← DI + 2 ELSE SI ← SI - 2 DI ← DI - 2 |

Ví dụ 7-3: Viết chương trình nhập 2 chuỗi từ bàn phím bằng hàm 0Ah, trong đó chỉ được khai báo 1 vùng đệm bàn phím.

(Hướng dẫn: Sau khi nhận được 1 chuỗi thì chuyển chuỗi nhận được sang biến chuỗi khác)

```

in Chuoi    MACRO    Chuoi
            MOV AH, 09h
            LEA DX, Chuoi
            INT 21h
            ENDM

NhapChuoi  MACRO    VungDem
            MOV AH, 0Ah    ; Nhập chuỗi bằng hàm 0Ah/21h
            LEA DX, VungDem
            INT 21h
            ENDM

ChuyenChuoi MACRO    Dich    ; Chuyển chuỗi từ vùng đệm buff đến chuỗi đích
            XOR CX, CX    ;
            MOV CL, Len    ; Số ký tự của chuỗi nhận được
            CLD            ; Chiều tăng
            LEA SI, Buff    ; địa chỉ chuỗi nguồn
            LEA DI, Dich    ; địa chỉ chuỗi đích
            REP MOVSB
            ENDM

DSEG SEGMENT
    msg1 DB    "Hay nhap Chuoi thu nhat: $"
    msg2 DB    "Hay nhap Chuoi thu hai: $"
    xdong DB    10, 13, '$'    ; Ký tự xuống dòng
    Chuoi1 DB    30    DUP (?)    ; nơi chứa chuỗi thứ nhất
    Chuoi2 DB    30    DUP (?)    ; nơi chứa chuỗi thứ hai
    max DB    30
    len DB    ?
    buff DB    31 DUP(?)
DSEG ENDS
CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG

```



```

MOV DS, AX
MOV ES, AX      ; Vì chuỗi nguồn và chuỗi đích cùng đoạn DS
inchuoi      msg1
nhapchuoi    max  ; Nhập chuỗi thứ nhất
chuyenchuoi chuoi1; Chuyển vào chuoi1
inchuoi      xdong
inchuoi      msg2
nhapchuoi    max  ; Nhập chuỗi thứ hai
chuyenchuoi chuoi2; Chuyển vào chuoi2
MOV AH, 4Ch
INT  21h
CSEG ENDS
END start

```

7.2.6. Lệnh So sánh hai chuỗi

CMPSB sẽ so sánh 1 byte trong chuỗi nguồn có địa chỉ xác định bởi DS:SI với 1 byte trong chuỗi đích có địa chỉ xác định bởi ES:DI.

CMPSW sẽ so sánh 2 byte trong chuỗi nguồn có địa chỉ xác định bởi DS:SI với 2 byte trong chuỗi đích có địa chỉ xác định bởi ES:DI.

| CMPSB | CMPSW |
|---|---|
| Cờ ← Mem[DS:SI] – Mem[ES:DI] IF (DF = 0) SI ← SI + 1 DI ← DI + 1 ELSE SI ← SI – 1 DI ← DI – 1 | Cờ ← Mem[DS:SI] – Mem[ES:DI] IF (DF = 0) SI ← SI + 2 DI ← DI + 2 ELSE SI ← SI – 2 DI ← DI – 2 |

Ví dụ 7-4: Viết tiếp ví dụ 7-3, sau khi đã nhập 2 chuỗi thì so sánh 2 chuỗi đã nhập được. Nếu giống nhau thì in ra màn hình “Hai chuỗi giống nhau”. Ngược lại thì in ra màn hình “Hai chuỗi khác nhau”

```

inchuoi      MACRO      chuoi
MOV AH, 09h
LEA  DX, chuoi
INT  21h
ENDM

nhapchuoi    MACRO      vungdem
MOV AH, 0Ah  ; Nhập chuỗi
LEA  DX, vungdem
INT  21h
ENDM

chuyenchuoi  MACRO      dich  ; Chuyển chuỗi từ vùng đệm bàn
XOR  CX, CX      ;          phím đến chuỗi đích
MOV  CL, len      ; Số ký tự của chuỗi nhận được

```

```
CLD ; Chiều tăng
LEA SI, buff ; địa chỉ chuỗi nguồn
LEA DI, dich ; địa chỉ chuỗi đích
REP MOVSB
ENDM

DSEG SEGMENT
msg1 DB "Hay nhap chuoai thu nhat: $"
msg2 DB "Hay nhap chuoai thu hai: $"
msg3 DB "Sau khi so sanh, ket qua: $"
msg4 DB "Hai chuoai giống nhau$"
msg5 DB "Hai chuoai khác nhau$"
msg
xdong DB 10, 13, '$' ; Ký tự xuống dòng
chuoai1 DB 30 DUP('$') ; nơi chứa chuỗi thứ nhất
chuoai2 DB 30 DUP('$') ; nơi chứa chuỗi thứ hai
max DB 30
len DB ?
buff DB 31 DUP('$')
DSEG ENDS
CSEG SEGMENT
ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
MOV DS, AX
MOV ES, AX ; Vì chuỗi nguồn và chuỗi đích cùng đoạn DS
inchoai msg1
nhapchuoai max ; Nhập chuỗi thứ nhất
chuyenchuoai chuoai1; Chuyển vào chuoai1
inchoai xdong
inchoai msg2
nhapchuoai max ; Nhập chuỗi thứ hai
chuyenchuoai chuoai2; Chuyển vào chuoai2
; ----- So sánh 2 chuỗi: chuoai1 và chuoai2 -----
MOV CX, 30 ; 30 ký tự
LEA SI, chuoai1 ; chuoai1 là chuỗi nguồn
LEA DI, chuoai2 ; chuoai2 là chuỗi đích
REPZ CMPSB ; nếu so sánh bằng thì tiếp tục so sánh tiếp
PUSHF
inchoai xdong
inchoai msg3
POPF
JE giống
inchoai msg5
JMP exit
giống: inchoai msg4
exit: MOV AH, 4Ch
```

```
INT 21h
CSEG ENDS
END start
```

7.2.7. Lệnh dò tìm trong chuỗi

SCASB sẽ so sánh nội dung AL (1 byte) với 1 byte trong chuỗi đích có địa chỉ xác định bởi ES:DI.

SCASW sẽ so sánh nội dung AX (2 byte) với 2 byte trong chuỗi đích có địa chỉ xác định bởi ES:DI.

| SCASB | SCASW |
|--|--|
| Cờ \leftarrow AL – Mem[ES:DI] IF (DF = 0) DI \leftarrow DI + 1 ELSE DI \leftarrow DI – 1 | Cờ \leftarrow AX – Mem[ES:DI] IF (DF = 0) DI \leftarrow DI + 2 ELSE DI \leftarrow DI – 2 |

Ví dụ: Viết chương trình nhập từ bàn phím 1 chuỗi, sau đó kiểm tra trong chuỗi nhận được có ký tự ‘A’ không. In kết quả ra màn hình.

```
inchuoi    MACRO    chuoi
            MOV AH, 09h
            LEA DX, chuoi
            INT 21h
            ENDM

nhapchuoi   MACRO    vungdem
            MOV AH, 0Ah    ; Nhập chuỗi
            LEA DX, vungdem
            INT 21h
            ENDM

DSEG SEGMENT
    msg1 DB    “Hay nhap chuoi (toi da 50 ky tu): $”
    msg2 DB    “Dang tim ky tu A trong chuoi.....$”
    msg3 DB    “Da tim gap ky tu A.$”
    msg4 DB    “Khong tim gap ky tu A.$”
    xdong DB    10, 13, ‘$’    ; Ký tự xuống dòng
    max DB    30
    len DB    ?
    buff DB    31    DUP(‘$’)
DSEG ENDS

CSEG SEGMENT
    ASSUME CS: CSEG, DS: DSEG
start: MOV AX, DSEG
        MOV DS, AX
        MOV ES, AX    ; Vì chuỗi nguồn và chuỗi đích cùng đoạn DS
        inchuoi    msg1
        nhapchuoi    max    ; Nhập chuỗi thứ nhất
```

```
    inhuoi    xdong
    inhuoi    msg2
    XOR  CX, CX
    MOV  CL, len        ; số ký tự của chuỗi
    MOV  AL, 'A'        ; ký tự cần tìm
    LEA  DI, buff        ; địa chỉ chuỗi cần tìm
    REPNE SCASB        ; khi gặp thì dừng
    PUSHF                ; Cất trạng thái cờ sau khi thực hiện SCASB
    inhuoi    xdong
    POPF                ; Lấy lại trạng thái cờ đã cất
    JE    gapA
    inhuoi    msg4
    JMP  exit
gapA: inhuoi    msg3
exit:  MOV  AH, 4Ch
      INT  21h
CSEG ENDS
      END start
```

BÀI TẬP CHƯƠNG 7

7.1. Viết chương trình nhập từ bàn phím 1 ký tự, sau đó in ra màn hình mã ASCII của ký tự nhận được ở dạng nhị phân.

Ví dụ: Hãy nhập 1 ký tự: **K**

Mã ASCII dạng nhị phân là: **01110101**

7.2. Viết chương trình nhập từ bàn phím 1 ký tự, sau đó in ra màn hình mã ASCII của ký tự nhận được ở thập lục phân.

Ví dụ: Hãy nhập 1 ký tự: **K**

Mã ASCII dạng thập lục phân là: **75**

7.3. Viết chương trình nhập từ bàn phím 1 số nhị phân 8 bit, sau đó in ra màn hình giá trị nhận được ở dạng số thập lục phân.

Ví dụ: Hãy nhập số nhị phân 8 bit: **10110101**

Dạng thập lục phân là: **B5**

7.4. Viết chương trình nhập 2 chuỗi A và B từ bàn phím, sau đó ghép chuỗi A với chuỗi B để tạo thành chuỗi C. In ra màn hình chuỗi C.

Ví dụ: Hãy nhập chuỗi A: *Chao cac ban*

 Hãy nhập chuỗi B: *Sinh vien CNTT*

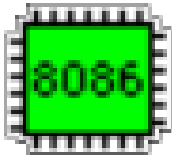
Chuỗi ghép C là: *Chao cac ban Sinh vien CNTT*

7.5. Viết chương trình nhập 1 chuỗi tối đa 256 ký tự. Sau đó tìm xem có các ký tự 'A', 'a', 'B', 'b' có trong chuỗi hay không. In ra màn hình các kết quả tìm được.

Ví dụ: Hãy nhập chuỗi: *Chao cac ban Sinh vien CNTT*

Kết quả:

- Không có 'A'.
- Có 'a'.
- Không có 'B'.
- Có 'b'.




Phụ lục 1

Hướng Dẫn Sử Dụng Emu8086

Emu8086 là công cụ mạnh của người lập trình Hợp ngữ. Bao gồm nhiều chức năng như: thực thi chương trình dạng EXE (EXE Template), thực thi chương trình dạng COM (COM Template), Thực thi đoạn lệnh hợp ngữ (BIN Template), tạo đoạn Boot máy (BOOT Template). Trong phần hướng dẫn này chỉ hướng dẫn cho sinh viên cách sử dụng công cụ Emu8086 để khảo sát lệnh Intel-8086, BIN Template (*các chức năng khác, sinh viên tự tìm hiểu thêm thông qua phần Help của công cụ*). Công cụ này giúp sinh viên thấy được sự thay đổi giá trị các thanh ghi, cờ, ngăn xếp, vùng nhớ ... sau khi thực hiện lệnh. Qua đó sinh viên hiểu rõ hơn hoạt động của lệnh hay sự tác động lên giá trị thanh ghi/cờ/ngăn xếp/bộ nhớ khi thực hiện lệnh.

1. Khởi động Emu8086:

Nhấp đúp biểu tượng  trên desktop, màn hình khởi động như hình 1 xuất hiện.

- **Code Samples** : Chọn file chương trình mẫu để thực hiện.
- **Quick Start Tutor**: Truy cập trang web hướng dẫn (phải có kết nối Internet).
- **Recent Files**: Chọn file trong danh sách file thường dùng.
- **Continue ...**: Tiếp tục vào màn hình làm việc.



Hình 1: Màn hình khởi động Emu8086

Khi bấm **Continue**, màn hình làm việc xuất hiện như hình 2 với file chương trình mẫu “Hello World” mặc nhiên xuất hiện trong vùng soạn thảo.

Hình 3 là công cụ **Number Convertor** (Bấm vào nút **Convertor** trên thanh công cụ) rất hữu dụng khi muốn chuyển đổi giá trị giữa các hệ thống số với nhau.

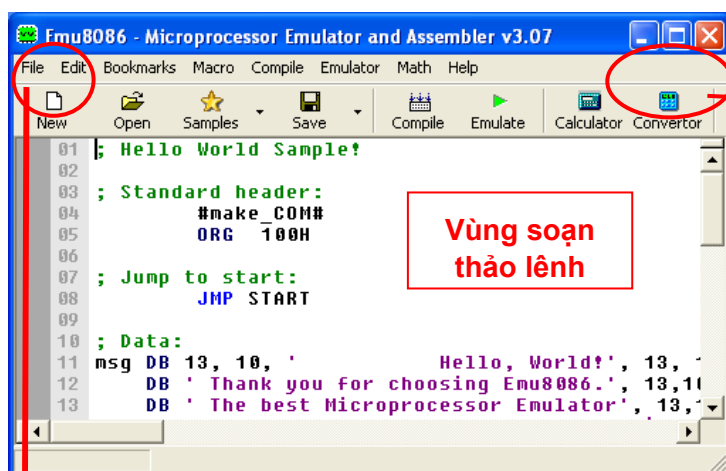
2. Soạn thảo lệnh hợp ngữ để khảo sát:

Để mở vùng làm việc mới chọn **NEW**, xuất hiện hình 4 để chọn Template

Để khảo sát lệnh Intel-8086 thì chọn chức năng thực thi lệnh (BIN Template). Vùng làm việc BIN Template xuất hiện như hình 5.

Trong BIN Template, quan trọng nhất là dòng đầu tiên **#make_bin#** dùng để xác định chế độ dịch lệnh của Emu8086. **Tuyệt đối không được thay đổi dòng lệnh giả này.**

Các dòng còn lại dùng để khởi động các giá trị ban đầu cho các thanh ghi và thông số địa chỉ segment/offset cho chương trình. Các dòng này không quan trọng, có thể xóa bỏ được hoặc thay đổi giá trị khởi động khác. Khi các dòng này bị xóa bỏ thì các thông số và thanh ghi sẽ được khởi động theo giá trị mặc nhiên. Để đơn giản, nên xóa bỏ từ dòng 2 đến hết.

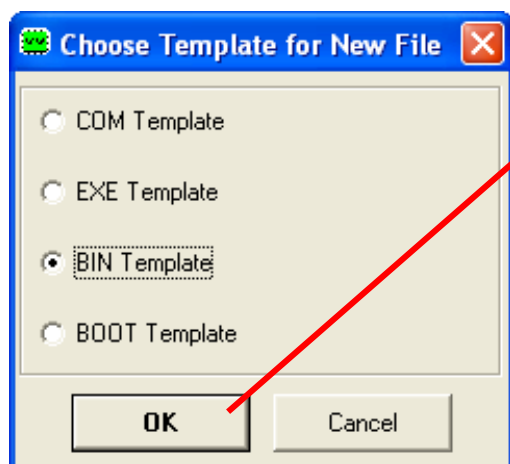


Hình 2: Màn hình Emu8086

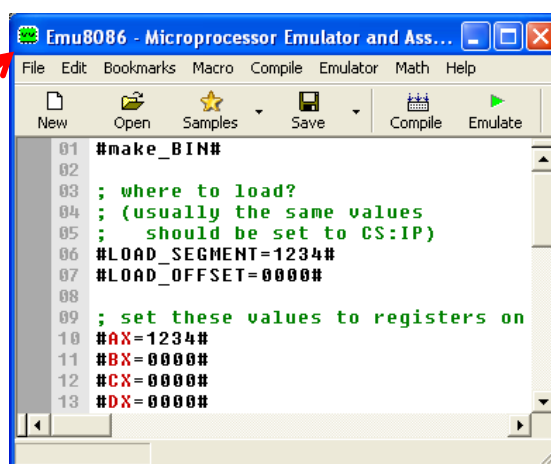
Converter



Hình 3: Chức năng

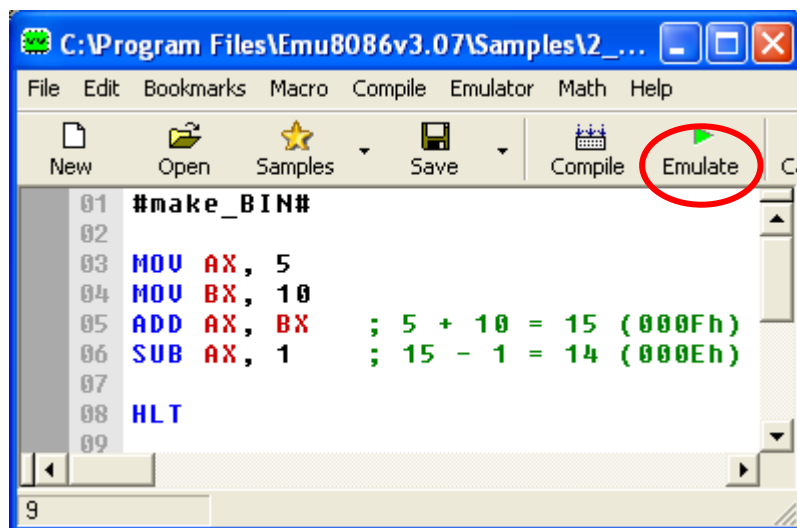


Hình 4: Chọn template



Hình 5: BIN Template

Ví dụ: Soạn đoạn lệnh như trong hình 6 sau để khảo sát:

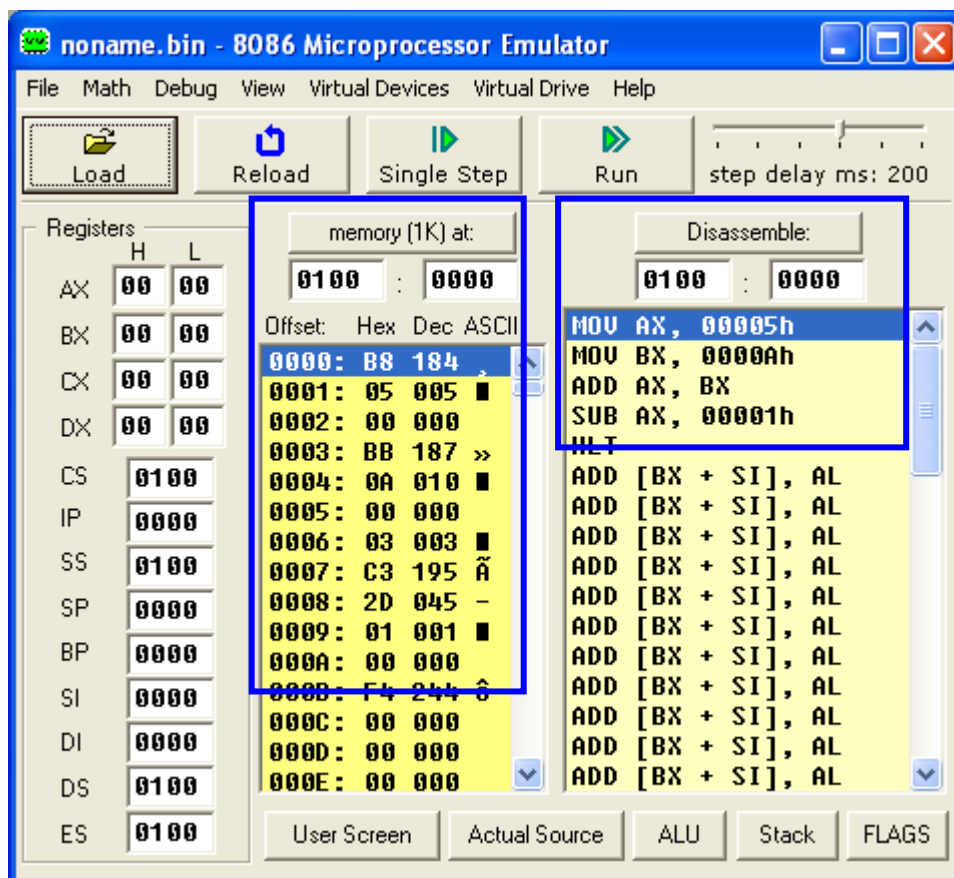


Hình 6: Ví dụ

3. Khảo sát lệnh (Giả lập - Emulate)

Để khảo sát lệnh bằng cách giả lập, chọn **Emulate**, khi đó màn hình giả lập xuất hiện như hình 7. Màn hình giả lập gồm 3 vùng: Thanh ghi (**Registers**), Bộ nhớ 1KB (**Memory**) chứa mã máy nhị phân và vùng hiển thị lệnh hợp ngữ tương ứng với mã máy nhị phân (**Disassemble**)

Giá trị các thanh ghi được trình bày ở dạng số Hex. Vùng bộ nhớ trình bày Hex – Dec – ASCII đối với từng ô nhớ (địa chỉ offset)

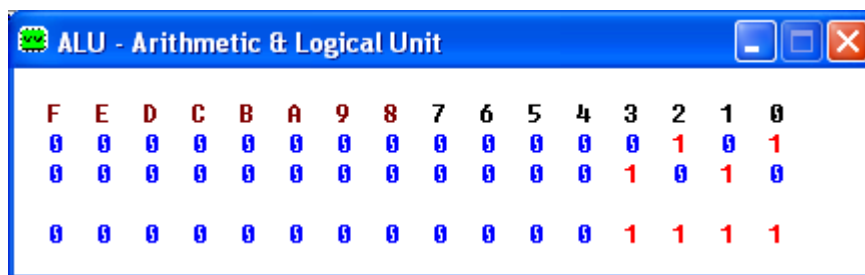


Hình 7: Màn hình giả lập

- **Reload:** Nạp lại đoạn lệnh
- **Run:** Chạy cả đoạn lệnh từ đầu cho đến khi gặp lệnh HLT (dừng)
- **Single Step:** Mỗi khi Single Step được bấm thì CPU chỉ chạy 1 lệnh hiện hành duy nhất (xác định bằng vệt sáng màu xanh) và dừng lại chờ cho đến khi Single Step được bấm tiếp. Như vậy, việc khảo sát lệnh có thể thực hiện thông qua Single Step.

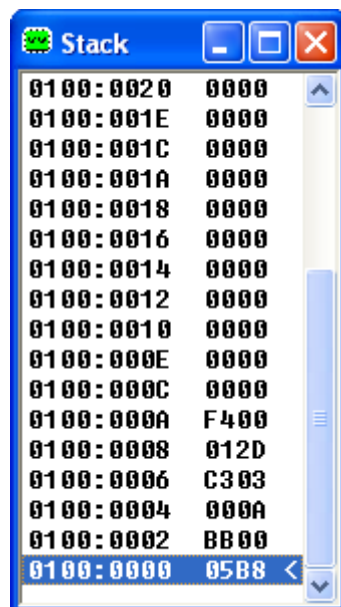
Các thành phần khác còn có thể xem được trạng thái khi CPU thực hiện lệnh trong chế độ giả lập như ALU, Stack và FLAGS (thanh ghi Cờ) bằng cách bấm vào các nút tương ứng

- Hình 8 cho biết trạng thái ALU khi thực hiện các phép toán (giá trị ở dạng nhị phân 16 bit). Dòng đầu tiên là thứ tự bit, dòng thứ 2 là giá trị toán hạng nguồn 1, dòng thứ 3 là giá trị toán hạng nguồn 2 và dòng cuối là giá trị kết quả sau khi thực hiện phép toán
- Hình 9 trình bày nội dung ngăn xếp ở dạng Hex 2 byte
- Hình 10 thể hiện trạng thái các cờ sau khi thực hiện phép toán



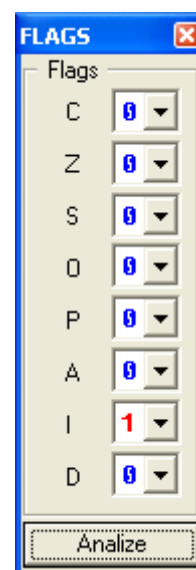
| F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Hình 8: Trạng thái ALU



| Address | Value |
|-----------|-------|
| 0100:0020 | 0000 |
| 0100:001E | 0000 |
| 0100:001C | 0000 |
| 0100:001A | 0000 |
| 0100:0018 | 0000 |
| 0100:0016 | 0000 |
| 0100:0014 | 0000 |
| 0100:0012 | 0000 |
| 0100:0010 | 0000 |
| 0100:000E | 0000 |
| 0100:000C | 0000 |
| 0100:000A | F400 |
| 0100:0008 | 012D |
| 0100:0006 | C303 |
| 0100:0004 | 000A |
| 0100:0002 | BB00 |
| 0100:0000 | 05B8 |

Hình 9: Stack



| Flag | Value |
|------|-------|
| C | 0 |
| Z | 0 |
| S | 0 |
| O | 0 |
| P | 0 |
| A | 0 |
| I | 1 |
| D | 0 |

Analyze

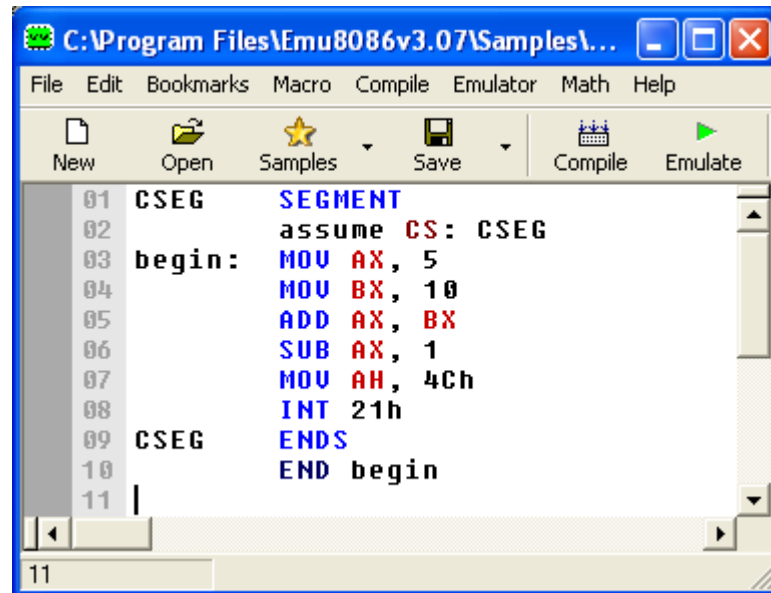
Hình 10: Thanh ghi Cờ

4. Thực thi chương trình dạng EXE hay COM

Emu8086 có thể thực thi chương trình Hợp ngữ viết theo cấu trúc dạng EXE hay COM.

Khi đó trong vùng soạn thảo, **không có dòng lệnh giả #make_bin#** và soạn thảo chương trình theo đúng cấu trúc dạng chương trình tương ứng.

Ví dụ: Chương trình dạng EXE như hình 11



Hình 11: Ví dụ chương trình dạng EXE

Phụ lục 2

Complete 8086 instruction set

http://www.emu8086.com/assembly_language_tutorial_assembler_reference/8086_instruction_set.html

Quick reference:

| | | | | | | | |
|----------------------|-----------------------|----------------------|----------------------|------------------------|-----------------------|-----------------------|-----------------------|
| AAA | CMPSB | JAE | JNBE | JPO | MOV | RCR | SCASB |
| AAD | CMPSW | JB | JNC | JS | MOVS | REP | SCASW |
| AAM | CWD | JBE | JNE | JZ | MOVSW | REPE | SHL |
| AAS | DAA | JC | JNG | LAHF | MUL | REPNE | SHR |
| ADC | DAS | JCXZ | JNGE | LDS | NEG | REPZ | STC |
| ADD | DEC | JE | JNL | LEA | NOP | REPZ | STD |
| AND | DIV | JG | JNLE | LES | NOT | RET | STI |
| CALL | HLT | JGE | JNO | LODSB | OR | RETF | STOSB |
| CBW | IDIV | JL | JNP | LODSW | OUT | ROL | STOSW |
| CLC | IMUL | JLE | JNS | LOOP | POP | ROR | SUB |
| CLD | IN | JMP | JNZ | LOOPE | POPA | SAHF | TEST |
| CLI | INC | JNA | JO | LOOPNE | POPF | SAL | XCHG |
| CMC | INT | JNAE | JP | LOOPNZ | PUSH | SAR | XLATB |
| CMP | INTO | JNB | JPE | LOOPZ | PUSHA | SBB | XOR |
| | IRET | | | | PUSHF | | |
| | JA | | | | RCL | | |

Operand types:

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

SREG: DS, ES, SS, and only as second operand: CS.

memory: [BX], [BX+SI+7], variable, etc....

immediate: 5, -24, 3Fh, 10001101b, etc...

Notes:

- When two operands are required for an instruction they are separated by comma. For example:
REG, memory
- When there are two operands, both operands must have the same size (except shift and rotate instructions). For example:
AL, DL
DX, AX
m1 DB ?
AL, m1
m2 DW ?
AX, m2

- Some instructions allow several operand combinations. For example:
memory, immediate
REG, immediate

memory, REG
REG, SREG

These marks are used to show the state of the flags:

- 1** - instruction sets this flag to **1**.
- 0** - instruction sets this flag to **0**.
- r** - flag value depends on result of the instruction.
- ?** - flag value is undefined (maybe **1** or **0**).

Instructions in alphabetical order:

| Instruction | Operands | Description | | | | | | | | | | | | |
|-------------|-------------|--|---|---|---|---|---|---|---|---|---|---|---|---|
| AAA | No operands | <p>ASCII Adjust after Addition. Corrects result in AH and AL after addition when working with BCD values.</p> <p>It works according to the following Algorithm: if low nibble of AL > 9 or AF = 1 then:</p> <p style="padding-left: 40px;">AL = AL + 6 AH = AH + 1 AF = 1 CF = 1</p> <p>else</p> <p style="padding-left: 40px;">AF = 0 CF = 0</p> <p>in both cases: clear the high nibble of AL.</p> | | | | | | | | | | | | |
| AAD | Nooperands | <p>ASCII Adjust before Division. Prepares two BCD values for division. Algorithm:</p> <p style="padding-left: 40px;">AL = (AH * 10) + AL AH = 0</p> <table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>?</td><td>r</td><td>r</td><td>?</td><td>r</td><td>?</td></tr></table> | C | Z | S | O | P | A | ? | r | r | ? | r | ? |
| C | Z | S | O | P | A | | | | | | | | | |
| ? | r | r | ? | r | ? | | | | | | | | | |
| AAM | No operands | <p>ASCII Adjust after Multiplication. Corrects the result of multiplication of two BCD values. Algorithm:</p> <p style="padding-left: 40px;">AH = AL / 10 AL = remainder</p> <table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>?</td><td>r</td><td>r</td><td>?</td><td>r</td><td>?</td></tr></table> | C | Z | S | O | P | A | ? | r | r | ? | r | ? |
| C | Z | S | O | P | A | | | | | | | | | |
| ? | r | r | ? | r | ? | | | | | | | | | |

| | | | | | | | | | | | | | | |
|------|--|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AAS | No operands | ASCII Adjust after Subtraction. Corrects result in AH and AL after subtraction when working with BCD values. Algorithm: if low nibble of AL > 9 or AF = 1 then: AL = AL - 6 AH = AH - 1 AF = 1 CF = 1 else AF = 0 CF = 0 in both cases: clear the high nibble of AL. <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>?</td><td>?</td><td>?</td><td>?</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | ? | ? | ? | ? | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | ? | ? | ? | ? | r | | | | | | | | | |
| ADC | REG,memory memory, REG REG, REG memory,immediate REG, immediate | Add with Carry. Algorithm: operand1 = operand1 + operand2 + CF <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| ADD | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Add. Algorithm: operand1 = operand1 + operand2 <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| AND | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Logical AND between all bits of two operands. Result is stored in operand1. These rules apply: 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0 <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td></tr></table></div> | C | Z | S | O | P | 0 | r | r | 0 | r | | |
| C | Z | S | O | P | | | | | | | | | | |
| 0 | r | r | 0 | r | | | | | | | | | | |
| CALL | procedure name label 4-byte address | Transfers control to procedure, return address is (IP) is pushed to stack. 4-byte address may be entered in this form: 1234h:5678h, first value is a segment second value is an offset (this is a far call, so CS is also pushed to stack). | | | | | | | | | | | | |
| CBW | No operands | Convert byte into word. Algorithm: if high bit of AL = 1 then AH = 255 (0FFh) | | | | | | | | | | | | |

| | | | | | | | | | | | | | | |
|-------|--|--|---|---|---|---|---|---|---|---|---|---|---|---|
| | | else AH = 0 | | | | | | | | | | | | |
| CLC | No operands | Clear Carry flag. Algorithm: CF = 0 | | | | | | | | | | | | |
| CLD | No operands | Clear Direction flag. SI and DI will be incremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVS, MOVSW, STOSB, STOSW. Algorithm: DF = 0 | | | | | | | | | | | | |
| CLI | No operands | Clear Interrupt enable flag. This disables hardware interrupts Algorithm: IF = 0 | | | | | | | | | | | | |
| CMC | No operands | Complement Carry flag. Inverts value of CF. Algorithm: if CF = 1 then CF = 0 if CF = 0 then CF = 1 | | | | | | | | | | | | |
| CMP | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Compare. Algorithm: operand1 - operand2 result is not stored anywhere, flags are set (OF, SF, ZF, AF, PF, CF) according to result. <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| CMPSB | No operands | Compare bytes: ES:[DI] from DS:[SI]. Algorithm: DS:[SI] - ES:[DI] set flags according to result: OF, SF, ZF, AF, PF, CF if DF = 0 then SI = SI + 1 DI = DI + 1 else SI = SI - 1 DI = DI - 1 <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>R</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | R | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | R | r | r | | | | | | | | | |
| CMPSW | No operands | Compare words: ES:[DI] from DS:[SI]. Algorithm: DS:[SI] - ES:[DI] set flags according to result: OF, SF, ZF, AF, PF, CF if DF = 0 then SI = SI + 2 DI = DI + 2 else SI = SI - 2 DI = DI - 2 <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |

| | | | | | | | | | | | | | | |
|-----|---------------|--|---|---|---|---|---|---|---|---|---|---|---|---|
| CWD | No operands | Convert Word to Double word. Algorithm: if high bit of AX = 1 then: DX = 65535 (0FFFFh) else DX = 0 | | | | | | | | | | | | |
| DAA | No operands | Decimal adjust After Addition. Corrects the result of addition of two packed BCD values. Algorithm: if low nibble of AL > 9 or AF = 1 then: AL = AL + 6 AF = 1 if AL > 9Fh or CF = 1 then: AL = AL + 60h CF = 1 <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| DAS | No operands | Decimal adjust After Subtraction. Corrects the result of subtraction of two packed BCD values. Algorithm: if low nibble of AL > 9 or AF = 1 then: AL = AL - 6 AF = 1 if AL > 9Fh or CF = 1 then: AL = AL - 60h CF = 1 <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| DEC | REG memory | Decrement. Algorithm: operand = operand - 1 <table><tr><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> <i>CF - unchanged!</i> | Z | S | O | P | A | r | r | r | r | r | | |
| Z | S | O | P | A | | | | | | | | | | |
| r | r | r | r | r | | | | | | | | | | |
| DIV | REG memory | Unsigned divide. Algorithm: when operand is a byte : AL = AX / operand AH = remainder (modulus) when operand is a word : AX = (DX AX) / operand DX = remainder (modulus) <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table> | C | Z | S | O | P | A | ? | ? | ? | ? | ? | ? |
| C | Z | S | O | P | A | | | | | | | | | |
| ? | ? | ? | ? | ? | ? | | | | | | | | | |
| HLT | No operands | Halt the System. | | | | | | | | | | | | |

| | | | | | | | | | | | | | | |
|--------|--|---|---|---|---|---|---|---|--------|---|---|---|---|---|
| IDIV | REG memory | Signed divide. Algorithm: when operand is a byte : $AL = AX / \text{operand}$ $AH = \text{remainder (modulus)}$ when operand is a word : $AX = (DX\ AX) / \text{operand}$ $DX = \text{remainder (modulus)}$ <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table></div> | C | Z | S | O | P | A | ? | ? | ? | ? | ? | ? |
| C | Z | S | O | P | A | | | | | | | | | |
| ? | ? | ? | ? | ? | ? | | | | | | | | | |
| IMUL | REG memory | Signed multiply. Algorithm: when operand is a byte : $AX = AL * \text{operand}$. when operand is a word : $(DX\ AX) = AX * \text{operand}$. <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>?</td><td>?</td><td>r</td><td>?</td><td>?</td></tr></table></div> <p><i>CF=OF=0 when result fits into operand of IMUL.</i></p> | C | Z | S | O | P | A | r | ? | ? | r | ? | ? |
| C | Z | S | O | P | A | | | | | | | | | |
| r | ? | ? | r | ? | ? | | | | | | | | | |
| IN | AL, im.byte AL, DX AX, im.byte AX, DX | Input from port into AL or AX . Second operand is a port number. If required to access port number over 255 - DX register should be used. | | | | | | | | | | | | |
| INC | REG memory | Increment. Algorithm: operand = operand + 1 <div><table><tr><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table><p><i>CF - unchanged!</i></p></div> | Z | S | O | P | A | r | r | r | r | r | | |
| Z | S | O | P | A | | | | | | | | | | |
| r | r | r | r | r | | | | | | | | | | |
| INT | immediate byte | Interrupt numbered by immediate byte (0..255). Algorithm: Push to stack: flags register, CS , IP IF = 0 Transfer control to interrupt procedure | | | | | | | | | | | | |
| INTO | No operands | Interrupt 4 if Overflow flag is 1. Algorithm: if OF = 1 then INT 4 | | | | | | | | | | | | |
| IRET | No operands | Interrupt Return. Algorithm: Pop from stack: IP , CS , flags register <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">popped</td></tr></table></div> | C | Z | S | O | P | A | popped | | | | | |
| C | Z | S | O | P | A | | | | | | | | | |
| popped | | | | | | | | | | | | | | |
| JA | label | Short Jump if first operand is Above second operand (as set by CMP instruction). Unsigned. Algorithm: if (CF = 0) and (ZF = 0) then jump | | | | | | | | | | | | |
| JAE | label | Short Jump if first operand is Above or Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 0 then jump | | | | | | | | | | | | |

| | | |
|------|-------------------------|---|
| JB | label | Short Jump if first operand is Below second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 then jump |
| JBE | label | Short Jump if first operand is Below or Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 or ZF = 1 then jump |
| JC | label | Short Jump if Carry flag is set to 1. Algorithm: if CF = 1 then jump |
| JCXZ | label | Short Jump if CX register is 0. Algorithm: if CX = 0 then jump |
| JE | label | Short Jump if first operand is Equal to second operand (as set by CMP instruction). Signed/Unsigned. Algorithm: if ZF = 1 then jump |
| JG | label | Short Jump if first operand is Greater than second operand (as set by CMP instruction). Signed. Algorithm: if (ZF = 0) and (SF = OF) then jump |
| JGE | label | Short Jump if first operand is Greater or Equal to second operand (as set by CMP instruction). Signed. Algorithm: if SF = OF then jump |
| JL | label | Short Jump if first operand is Less than second operand (as set by CMP instruction). Signed. Algorithm: if SF \neq OF then jump |
| JLE | label | Short Jump if first operand is Less or Equal to second operand (as set by CMP instruction). Signed. Algorithm: if SF \neq OF or ZF = 1 then jump |
| JMP | label 4-byte address | Unconditional Jump. Transfers control to another part of the program. 4-byte address may be entered in this form: 1234h:5678h, first value is a segment second value is an offset Algorithm: always jump |
| JNA | label | Short Jump if first operand is Not Above second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 or ZF = 1 then jump |
| JNAE | label | Short Jump if first operand is Not Above and Not Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 then jump |
| JNB | label | Short Jump if first operand is Not Below second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 0 then jump |
| JNBE | label | Short Jump if first operand is Not Below and Not Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if (CF = 0) and (ZF = 0) then jump |
| JNC | label | Short Jump if Carry flag is set to 0. Algorithm: if CF = 0 then jump |

| | | |
|------|-------|--|
| JNE | label | Short Jump if first operand is Not Equal to second operand (as set by CMP instruction). Signed/Unsigned. Algorithm: if ZF = 0 then jump |
| JNG | label | Short Jump if first operand is Not Greater then second operand (as set by CMP instruction). Signed. Algorithm: if (ZF = 1) and (SF \neq OF) then jump |
| JNGE | label | Short Jump if first operand is Not Greater and Not Equal to second operand (as set by CMP instruction). Signed. Algorithm: if SF \neq OF then jump |
| JNL | label | Short Jump if first operand is Not Less then second operand (as set by CMP instruction). Signed. Algorithm: if SF = OF then jump |
| JNLE | label | Short Jump if first operand is Not Less and Not Equal to second operand (as set by CMP instruction). Signed. Algorithm: if (SF = OF) and (ZF = 0) then jump |
| JNO | label | Short Jump if Not Overflow. Algorithm: if OF = 0 then jump |
| JNP | label | Short Jump if No Parity (odd). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 0 then jump |
| JNS | label | Short Jump if Not Signed (if positive). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if SF = 0 then jump |
| JNZ | label | Short Jump if Not Zero (not equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if ZF = 0 then jump |
| JO | label | Short Jump if Overflow. Algorithm: if OF = 1 then jump |
| JP | label | Short Jump if Parity (even). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 1 then jump |
| JPE | label | Short Jump if Parity Even. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 1 then jump |
| JPO | label | Short Jump if Parity Odd. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 0 then jump |

| | | |
|-------|-------------|---|
| JS | label | Short Jump if Signed (if negative). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if SF = 1 then jump |
| JZ | label | Short Jump if Zero (equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if ZF = 1 then jump |
| LAHF | No operands | Load AH from 8 low bits of Flags register. Algorithm: AH = flags register AH bit: 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 are reserved. |
| LDS | REG, memory | Load memory double word into word register and DS. Algorithm: REG = first word DS = second word |
| LEA | REG, memory | Load Effective Address. Algorithm: REG = address of memory (offset) |
| LES | REG, memory | Load memory double word into word register and ES. Algorithm: REG = first word ES = second word |
| LODSB | No operands | Load byte at DS:[SI] into AL. Update SI. Algorithm: AL = DS:[SI] if DF = 0 then SI = SI + 1 else SI = SI - 1 |
| LODSW | No operands | Load word at DS:[SI] into AX. Update SI. Algorithm: AX = DS:[SI] if DF = 0 then SI = SI + 2 else SI = SI - 2 |
| LOOP | label | Decrease CX, jump to label if CX not zero. Algorithm: CX = CX - 1 if CX > 0 then jump else no jump, continue |
| LOOPE | label | Decrease CX, jump to label if CX not zero and Equal (ZF=1) Algorithm: CX = CX - 1 if (CX > 0) and (ZF = 1) then jump else no jump, continue |

| | | |
|--------|--|--|
| LOOPNE | label | Decrease CX, jump to label if CX not zero and Not Equal (ZF = 0). Algorithm: CX = CX - 1 if (CX > 0) and (ZF = 0) then jump else no jump, continue |
| LOOPNZ | label | Decrease CX, jump to label if CX not zero and ZF = 0. Algorithm: CX = CX - 1 if (CX > 0) and (ZF = 0) then jump else no jump, continue |
| LOOPZ | label | Decrease CX, jump to label if CX not zero and ZF = 1. Algorithm: CX = CX - 1 if (CX > 0) and (ZF = 1) then jump else no jump, continue |
| MOV | REG, memory memory, REG REG, REG memory, immediate REG, immediate SREG, memory memory, SREG REG, SREG SREG, REG | Copy operand2 to operand1. The MOV instruction <u>cannot</u> : set the value of the CS and IP registers. copy value of one segment register to another segment register (should copy to general register first). copy immediate value to segment register (should copy to general register first). Algorithm: operand1 = operand2 |
| MOVS | No operands | Copy byte at DS:[SI] to ES:[DI]. Update SI and DI. Algorithm: ES:[DI] = DS:[SI] if DF = 0 then SI = SI + 1 DI = DI + 1 else SI = SI - 1 DI = DI - 1 |
| MOVSW | No operands | Copy word at DS:[SI] to ES:[DI]. Update SI and DI. Algorithm: ES:[DI] = DS:[SI] if DF = 0 then SI = SI + 2 DI = DI + 2 else SI = SI - 2 DI = DI - 2 |

| | | | | | | | | | | | | | | |
|------|--|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL | REG memory | Unsigned multiply. Algorithm: when operand is a byte : $AX = AL * \text{operand}$. when operand is a word : $(DX\ AX) = AX * \text{operand}$. <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>?</td><td>?</td><td>r</td><td>?</td><td>?</td></tr></table> <i>CF=OF=0 when high section of the result is zero.</i> | C | Z | S | O | P | A | r | ? | ? | r | ? | ? |
| C | Z | S | O | P | A | | | | | | | | | |
| r | ? | ? | r | ? | ? | | | | | | | | | |
| NEG | REG memory | Negate. Makes operand negative (two's complement). Algorithm: Invert all bits of the operand Add 1 to inverted operand <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| NOP | No operands | No Operation. Algorithm: Do nothing | | | | | | | | | | | | |
| NOT | REG memory | Invert each bit of the operand. Algorithm: if bit is 1 turn it to 0. if bit is 0 turn it to 1. | | | | | | | | | | | | |
| OR | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Logical OR between all bits of two operands. Result is stored in first operand. <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td><td>?</td></tr></table> | C | Z | S | O | P | A | 0 | r | r | 0 | r | ? |
| C | Z | S | O | P | A | | | | | | | | | |
| 0 | r | r | 0 | r | ? | | | | | | | | | |
| OUT | im.byte, AL im.byte, AX DX, AL DX, AX | Output from AL or AX to port. First operand is a port number. If required to access port number over 255 - DX register should be used. | | | | | | | | | | | | |
| POP | REG SREG memory | Get 16 bit value from the stack. Algorithm: operand = SS:[SP] (top of the stack) SP = SP + 2 | | | | | | | | | | | | |
| POPA | No operands | Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack. SP value is ignored, it is Popped but not set to SP register). <i>Note: this instruction works only on 80186 CPU and later!</i> Algorithm: POP DI POP SI | | | | | | | | | | | | |

| | | |
|-------|---|---|
| | | POP BP POP xx (SP value ignored) POP BX POP DX POP CX POP AX |
| POPF | No operands | Get flags register from the stack. Algorithm: flags = SS:[SP] (top of the stack) SP = SP + 2 <div><div>CZSOPA</div><div>popped</div></div> |
| PUSH | REG SREG memory immediate | Store 16 bit value in the stack. <i>Note: PUSH immediate works only on 80186 CPU and later!</i> Algorithm: SP = SP - 2 SS:[SP] (top of the stack) = operand |
| PUSHA | No operands | Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack. Original value of SP register (before PUSHA) is used. <i>Note: this instruction works only on 80186 CPU and later!</i> Algorithm: PUSH AX PUSH CX PUSH DX PUSH BX PUSH SP PUSH BP PUSH SI PUSH DI |
| PUSHF | No operands | Store flags register in the stack. Algorithm: SP = SP - 2 SS:[SP] (top of the stack) = flags |
| RCL | memory, immediate REG, immediate memory, CL REG, CL | Rotate operand1 left through Carry Flag. The number of rotates is set by operand2. When immediate is greater than 1, assembler generates several RCL xx, 1 instructions because 8086 has machine code only for this instruction (the same principle works for all other shift/rotate instructions). Algorithm: shift all bits left, the bit that goes off is set to CF and previous value of CF is inserted to the right-most position. <div><div>C</div><div>O</div><div>r</div><div>r</div></div> <p>OF=0 if first operand keeps original sign.</p> |

| | | | | | | |
|-------|---|---|---|---|---|---|
| RCR | <div>memory, immediate REG, immediate</div> <div>memory, CL REG, CL</div> | <div>Rotate operand1 right through Carry Flag. The number of rotates is set by operand2.</div> <div>Algorithm:</div> <div>shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left-most position.</div> <div><table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table><div>OF=0 if first operand keeps original sign.</div></div> | C | O | r | r |
| C | O | | | | | |
| r | r | | | | | |
| REP | chain instruction | <div>Repeat following MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times.</div> <div>Algorithm:</div> <div>check_cx:</div> <div>if CX <> 0 then</div> <div>do following chain instruction</div> <div>CX = CX - 1</div> <div>go back to check_cx</div> <div>else</div> <div>exit from REP cycle</div> <div><table><tr><td>Z</td></tr><tr><td>r</td></tr></table></div> | Z | r | | |
| Z | | | | | | |
| r | | | | | | |
| REPE | chain instruction | <div>Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal), maximum CX times.</div> <div>Algorithm:</div> <div>check_cx:</div> <div>if CX <> 0 then</div> <div>do following chain instruction</div> <div>CX = CX - 1</div> <div>if ZF = 1 then go back to check_cx</div> <div>else exit from REPE cycle</div> <div>else exit from REPE cycle</div> <div><table><tr><td>Z</td></tr><tr><td>r</td></tr></table></div> | Z | r | | |
| Z | | | | | | |
| r | | | | | | |
| REPNE | chain instruction | <div>Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Equal), maximum CX times.</div> <div>Algorithm:</div> <div>check_cx:</div> <div>if CX <> 0 then</div> <div>do following chain instruction</div> <div>CX = CX - 1</div> <div>if ZF = 0 then go back to check_cx</div> <div>else exit from REPNE cycle</div> <div>else exit from REPNE cycle</div> <div><table><tr><td>Z</td></tr><tr><td>r</td></tr></table></div> | Z | r | | |
| Z | | | | | | |
| r | | | | | | |

| | | |
|------|---|---|
| REPZ | chain instruction | <p>Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Zero), maximum CX times.</p> <p>Algorithm:</p> <p>check_cx</p> <p>if CX <> 0 then</p> <p>do following <u>chain instruction</u></p> <p>CX = CX - 1</p> <p>if ZF = 0 then go back to check_cx</p> <p>else exit from REPZ cycle</p> <p>else exit from REPZ cycle</p> <div><div>Z</div><div>r</div></div> |
| REPZ | chain instruction | <p>Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Zero), maximum CX times.</p> <p>Algorithm:</p> <p>check_cx</p> <p>if CX <> 0 then</p> <p>do following <u>chain instruction</u></p> <p>CX = CX - 1</p> <p>if ZF = 1 then go back to check_cx</p> <p>else exit from REPZ cycle</p> <p>else exit from REPZ cycle</p> <div><div>Z</div><div>r</div></div> |
| RET | No operands or even immediate | <p>Return from near procedure.</p> <p>Algorithm: Pop from stack → IP</p> <p>if <u>immediate</u> operand is present: SP = SP + operand</p> |
| RETF | No operands or even immediate | <p>Return from Far procedure.</p> <p>Algorithm: Pop from stack → CS:IP</p> <p>if <u>immediate</u> operand is present: SP = SP + operand</p> |
| ROL | memory, immediate REG, immediate memory, CL REG, CL | <p>Rotate operand1 left. The number of rotates is set by operand2.</p> <p>Algorithm:</p> <p>shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position.</p> <div><div>C</div><div>O</div><div>r</div><div>r</div></div> <p><i>OF=0 if first operand keeps original sign.</i></p> |
| ROR | memory, immediate REG, immediate memory, CL REG, CL | <p>Rotate operand1 right. The number of rotates is set by operand2.</p> <p>Algorithm:</p> <p>shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position.</p> <div><div>C</div><div>O</div><div>r</div><div>r</div></div> <p><i>OF=0 if first operand keeps original sign.</i></p> |

| | | | | | | | | | | | | | | |
|-------|---|--|---|---|---|---|---|---|---|---|---|---|---|---|
| SAHF | No operands | Store AH register into low 8 bits of Flags register. Algorithm: flags register = AH AH bit: 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 are reserved. <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| SAL | memory, immediate REG, immediate memory, CL REG, CL | Shift Arithmetic operand1 Left. The number of shifts is set by operand2. Algorithm: Shift all bits left, the bit that goes off is set to CF. Zero bit is inserted to the right-most position. <div><table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <i>OF=0 if first operand keeps original sign.</i></div> | C | O | r | r | | | | | | | | |
| C | O | | | | | | | | | | | | | |
| r | r | | | | | | | | | | | | | |
| SAR | memory, immediate REG, immediate memory, CL REG, CL | Shift Arithmetic operand1 Right. The number of shifts is set by operand2. Algorithm: Shift all bits right, the bit that goes off is set to CF. The sign bit that is inserted to the left-most position has the same value as before shift. <div><table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <i>OF=0 if first operand keeps original sign.</i></div> | C | O | r | r | | | | | | | | |
| C | O | | | | | | | | | | | | | |
| r | r | | | | | | | | | | | | | |
| SBB | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Subtract with Borrow. Algorithm: operand1 = operand1 - operand2 - CF <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| SCASB | No operands | Compare bytes: AL from ES:[DI]. Algorithm: ES:[DI] - AL set flags according to result: OF, SF, ZF, AF, PF, CF if DF = 0 then DI = DI + 1 else DI = DI - 1 <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |
| SCASW | No operands | Compare words: AX from ES:[DI]. Algorithm: ES:[DI] - AX set flags according to result: OF, SF, ZF, AF, PF, CF if DF = 0 then DI = DI + 2 else DI = DI - 2 <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |

| | | | | | | | | | | | | | | |
|-------|---|--|---|---|---|---|---|---|---|---|---|---|---|---|
| SHL | memory, immediate REG, immediate memory, CL REG, CL | Shift operand1 Left. The number of shifts is set by operand2. Algorithm: Shift all bits left, the bit that goes off is set to CF. Zero bit is inserted to the right-most position. <div><table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <i>OF=0 if first operand keeps original sign.</i></div> | C | O | r | r | | | | | | | | |
| C | O | | | | | | | | | | | | | |
| r | r | | | | | | | | | | | | | |
| SHR | memory, immediate REG, immediate memory, CL REG, CL | Shift operand1 Right. The number of shifts is set by operand2. Algorithm: Shift all bits right, the bit that goes off is set to CF. Zero bit is inserted to the left-most position. <div><table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <i>OF=0 if first operand keeps original sign.</i></div> | C | O | r | r | | | | | | | | |
| C | O | | | | | | | | | | | | | |
| r | r | | | | | | | | | | | | | |
| STC | No operands | Set Carry flag. Algorithm: CF = 1 | | | | | | | | | | | | |
| STD | No operands | Set Direction flag. SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: DF = 1 | | | | | | | | | | | | |
| STI | No operands | Set Interrupt enable flag. This enables hardware interrupts. Algorithm: IF = 1 | | | | | | | | | | | | |
| STOSB | No operands | Store byte in AL into ES:[DI]. Update DI. Algorithm: ES:[DI] = AL if DF = 0 then DI = DI + 1 else DI = DI - 1 | | | | | | | | | | | | |
| STOSW | No operands | Store word in AX into ES:[DI]. Update DI. Algorithm: ES:[DI] = AX if DF = 0 then DI = DI + 2 else DI = DI - 2 | | | | | | | | | | | | |
| SUB | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Subtract. Algorithm: operand1 = operand1 - operand2 <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> | C | Z | S | O | P | A | r | r | r | r | r | r |
| C | Z | S | O | P | A | | | | | | | | | |
| r | r | r | r | r | r | | | | | | | | | |

| | | | | | | | | | | | | | | |
|-------|--|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TEST | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Logical AND between all bits of two operands for flags only. These flags are effected: ZF, SF, PF . Result is not stored anywhere. <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td></tr></table></div> | C | Z | S | O | P | 0 | r | r | 0 | r | | |
| C | Z | S | O | P | | | | | | | | | | |
| 0 | r | r | 0 | r | | | | | | | | | | |
| XCHG | REG, memory memory, REG REG, REG | Exchange values of two operands. Algorithm: operand1 < - > operand2 | | | | | | | | | | | | |
| XLATB | No operands | Translate byte from table. Copy value of memory byte at DS:[BX + unsigned AL] to AL register. Algorithm: AL = DS:[BX + unsigned AL] | | | | | | | | | | | | |
| XOR | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand. <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td><td>?</td></tr></table></div> | C | Z | S | O | P | A | 0 | r | r | 0 | r | ? |
| C | Z | S | O | P | A | | | | | | | | | |
| 0 | r | r | 0 | r | ? | | | | | | | | | |

Phụ Lục 3

BẢNG MÃ ASCII

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|------------------|-----|-----|-------|-----|-----|------|-----|-----|------|
| 0 | 00 | Null | 32 | 20 | Space | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 01 | Start of heading | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 02 | Start of text | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | End of text | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | End of transmit | 36 | 24 | \$ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | Enquiry | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | Acknowledge | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | Audible bell | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | Backspace | 40 | 28 | (| 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | Horizontal tab | 41 | 29 |) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | Line feed | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | Vertical tab | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | Form feed | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | Carriage return | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | Shift out | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | Shift in | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | Data link escape | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | Device control 1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | Device control 2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | Device control 3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | Device control 4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | Neg. acknowledge | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | Synchronous idle | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | End trans. block | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | Cancel | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | End of medium | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | Substitution | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | Escape | 59 | 3B | ; | 91 | 5B | [| 123 | 7B | { |
| 28 | 1C | File separator | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | |
| 29 | 1D | Group separator | 61 | 3D | = | 93 | 5D |] | 125 | 7D | } |
| 30 | 1E | Record separator | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | Unit separator | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | □ |

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|------|-----|-----|------|-----|-----|------|-----|-----|------|
| 128 | 80 | Ç | 160 | A0 | á | 192 | C0 | Ł | 224 | E0 | α |
| 129 | 81 | ù | 161 | A1 | í | 193 | C1 | ł | 225 | E1 | β |
| 130 | 82 | é | 162 | A2 | ó | 194 | C2 | Ṭ | 226 | E2 | Γ |
| 131 | 83 | â | 163 | A3 | ú | 195 | C3 | ṭ | 227 | E3 | π |
| 132 | 84 | ä | 164 | A4 | ñ | 196 | C4 | — | 228 | E4 | Σ |
| 133 | 85 | à | 165 | A5 | Ñ | 197 | C5 | † | 229 | E5 | σ |
| 134 | 86 | ă | 166 | A6 | ² | 198 | C6 | ‡ | 230 | E6 | μ |
| 135 | 87 | ç | 167 | A7 | ° | 199 | C7 | ‡ | 231 | E7 | ι |
| 136 | 88 | ê | 168 | A8 | ¿ | 200 | C8 | ℔ | 232 | E8 | Φ |
| 137 | 89 | ë | 169 | A9 | ƒ | 201 | C9 | ℥ | 233 | E9 | Θ |
| 138 | 8A | è | 170 | AA | ¬ | 202 | CA | ℥ | 234 | EA | Ω |
| 139 | 8B | ï | 171 | AB | ½ | 203 | CB | ℥ | 235 | EB | δ |
| 140 | 8C | î | 172 | AC | ¼ | 204 | CC | ‡ | 236 | EC | ∞ |
| 141 | 8D | ì | 173 | AD | ¡ | 205 | CD | = | 237 | ED | ∞ |
| 142 | 8E | Ä | 174 | AE | « | 206 | CE | ‡ | 238 | EE | ε |
| 143 | 8F | Å | 175 | AF | » | 207 | CF | ℥ | 239 | EF | ∩ |
| 144 | 90 | É | 176 | B0 | ⋮ | 208 | D0 | ℥ | 240 | FO | ≡ |
| 145 | 91 | æ | 177 | B1 | ⋮ | 209 | D1 | ℥ | 241 | F1 | ± |
| 146 | 92 | Æ | 178 | B2 | ⋮ | 210 | D2 | π | 242 | F2 | ≥ |
| 147 | 93 | ô | 179 | B3 | | 211 | D3 | ℥ | 243 | F3 | ≤ |
| 148 | 94 | ö | 180 | B4 | † | 212 | D4 | ℥ | 244 | F4 | [|
| 149 | 95 | ò | 181 | B5 | ‡ | 213 | D5 | ℥ | 245 | F5 |] |
| 150 | 96 | û | 182 | B6 | ‡ | 214 | D6 | π | 246 | F6 | ÷ |
| 151 | 97 | ù | 183 | B7 | π | 215 | D7 | ‡ | 247 | F7 | ≈ |
| 152 | 98 | ÿ | 184 | B8 | ¶ | 216 | D8 | ‡ | 248 | F8 | ° |
| 153 | 99 | Ö | 185 | B9 | ‡ | 217 | D9 | ┘ | 249 | F9 | ▪ |
| 154 | 9A | Ü | 186 | BA | | 218 | DA | ƒ | 250 | FA | · |
| 155 | 9B | ◊ | 187 | BB | ¶ | 219 | DB | ■ | 251 | FB | √ |
| 156 | 9C | £ | 188 | BC | ¶ | 220 | DC | ■ | 252 | FC | ² |
| 157 | 9D | ¥ | 189 | BD | ¶ | 221 | DD | ■ | 253 | FD | ² |
| 158 | 9E | ℔ | 190 | BE | ¶ | 222 | DE | ■ | 254 | FE | ■ |
| 159 | 9F | f | 191 | BF | ¶ | 223 | DF | ■ | 255 | FF | □ |