# Lý thuyết

## Câu 1:

Trình bày các nội dung sau trong kiến trúc Intel x86 32bit.

- Cấu trúc một chương trình hợp ngữ.

- Danh sách các thanh ghi được sử dụng trong chương trình.

- Danh sách các cờ

**Trả lời:**

**1. Cấu trúc chương trình**

.386

.model flat,stdcall

.stack 4096

;///// function and procedure

.data

; declare variables here

.code

main PROC

; /////write your code here

;//////INVOKE ExitProcess,0

main ENDP

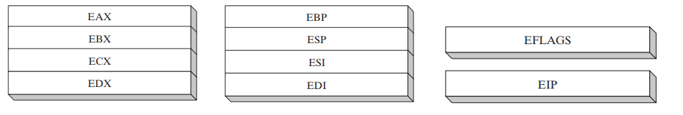
; //////(insert additional procedures here)

END main

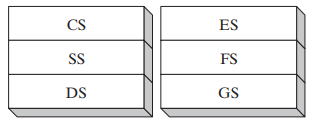
**2. Danh sách các thanh ghi được sử dụng trong chương trình.**

**+** Thanh ghi thực hiện chương trình cơ bản.

Các thanh ghi là các vị trí lưu trữ tốc độ cao trực tiếp bên trong CPU, được thiết kế để truy cập ở tốc độ cao hơn nhiều so với bộ nhớ thông thường. Khi một vòng lặp xử lý được tối ưu hóa về tốc độ, vd bộ đếm vòng lặp được giữ trong các thanh ghi thay vì các biến. Có 8 thanh ghi mục đích chung, 6 thanh ghi phân đoạn, thanh ghi cờ trạng thái bộ xử lý (EFLAGS) và một con trỏ lệnh (EIP).

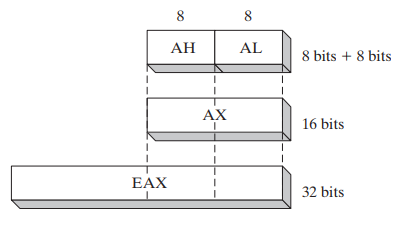


*Thanh ghi mục đích chung 32 bit*



*Thanh ghi phân đoạn 16 bit*

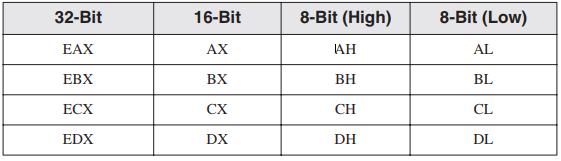
Các thanh ghi có mục đích chung chủ yếu được sử dụng cho số học và dữ liệu. Vd: 16 bit thấp hơn của thanh ghi EAX có thể được tham chiếu bằng tên AX.



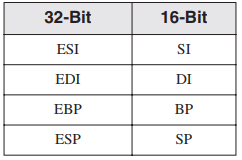
*Thanh ghi dùng cho mục đích chung*

Ví dụ: Ax = 9876h => Ah = 98h, Al = 76h

Các phần của một số thanh ghi có thể được xử lý dưới dạng giá trị 8 bit. Vd: thanh ghi AX có nửa trên 8 bit có tên AH và một nửa dưới 8 bit có tên AL. Mối quan hệ chồng chéo tương tự tồn tại cho các thanh ghi EAX, EBX, ECX và EDX:



Các thanh ghi mục đích chung còn lại chỉ có thể được truy cập bằng tên 32 bit hoặc 16 bit, như trong bảng sau:



+ Một số thanh ghi sử dụng chuyên dụng

***\*Một số thanh ghi mục đích chung có công dụng chuyên dụng:***

• EAX được tự động sử dụng theo chỉ thị nhân và chia.

• CPU tự động sử dụng ECX làm bộ đếm vòng lặp.

Ví dụ: Array PROC

Push ebp

Mov ebp, esp

Pushad

Mov esi, [ebp + 12]

Mov ecx, [ebp +8]

Cmp ecx, 0

Je L1

L1:

Mov eax, 10000h

Call RandomRange

Mov [esi], ax

Add esi, TYPE WORD

Loop L1

• ESP xử lý dữ liệu trên ngăn xếp (cấu trúc bộ nhớ hệ thống).

Ví dụ: AddTwo PROC

Push ebp

Mov ebp, esp

• ESI và EDI được sử dụng bởi các chỉ thị chuyển bộ nhớ tốc độ cao.

Ví dụ: xor buffer[esi],KEY

Inc esi

• EBP được sử dụng bởi các ngôn ngữ cấp cao để tham chiếu các tham số hàm và biến cục bộ trên ngăn xếp.

Ví dụ: push EBP

mov EBP,EDI

mov EAX,[EBP +16]

sub EAX,[EBP+12]

pop EBP

***+ Các thanh ghi đoạn***

Trong chế độ địa chỉ thực, các thanh ghi đoạn 16 bit cho biết các địa chỉ cơ sở của các vùng nhớ được gán trước. Trong chế độ được bảo vệ, các thanh ghi đoạn giữ con trỏ để phân đoạn bảng mô tả. Một số đoạn giữ các chương trình (mã), các đoạn khác giữ các biến (dữ liệu) và một đoạn khác có tên đoạn ngăn xếp chứa các biến cục bộ và tham số chức năng.

+Thanh ghi đoạn CS:

Ví dụ: pop CS

Pop IP ;CS – địa chỉ segment

;IP – đại chỉ offset

+ Thanh ghi đoạn DS:

Ví dụ: xau byte ‘helloword’

Mov ax, segment xau

Mov ds,ax

+ Thanh ghi đoạn SS:

Ví dụ: pop DI ; [SS:SP+1,SS:SP] ← DI

Pop CS ; [SS:SP+1,SS:SP] ← CS

Pop [SI] ; [SS:SP+1,SS:SP] ← [SI+1,SI]

***+ Con trỏ chỉ dẫn***

Con trỏ lệnh EIP hoặc con trỏ lệnh, thanh ghi chứa địa chỉ của lệnh tiếp theo sẽ được thực thi. Một số chỉ thị máy nhất định thao tác EIP, khiến chương trình rẽ nhánh đến một vị trí mới.

Ví dụ: CALL addsource ; Gọi procedure addsource giả sử chiếm 9 byte

; Địa chỉ hiện tại là 00000030, lưu vào EIP

MOV EAX,EBX ; Địa chỉ là 00000039

; Địa chỉ này được đẩy vào Stack khi đang gọi hàm addsource. Sau khi xử lý xong hàm mySub, EIP được load địa chỉ 00000039 để xử lý tiếp

addsource PROC:

; code

addsource ENDP

***+ Thanh ghi EFLAGS***

Thanh ghi EFLAGS (hoặc chỉ cờ) bao gồm các bit nhị phân riêng lẻ điều khiển hoạt động của CPU hoặc phản ánh kết quả của một số hoạt động của CPU.

Một cờ được đặt khi nó bằng 1; chưa tác động (hoặc thiết lập lại), nó bằng 0.

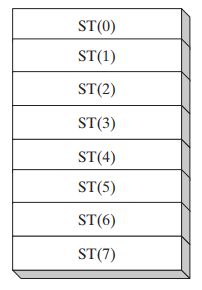
+*Thanh ghi MMX, XMM*

*\*Thanh ghi MMX*

Công nghệ MMX đã được Intel bổ sung vào bộ xử lý Pentium để cải thiện hiệu suất của các ứng dụng đa phương tiện và truyền thông tiên tiến. 8 thanh ghi MMX 64 bit hỗ trợ các hướng dẫn đặc biệt gọi là SIMD (Hướng dẫn đơn, Đa dữ liệu). Các lệnh MMX hoạt động song song trên các giá trị dữ liệu có trong các thanh ghi MMX. Mặc dù chúng có vẻ là các thanh ghi riêng biệt, nhưng các tên thanh ghi MMX trên thực tế là bí danh cho cùng các thanh ghi được sử dụng bởi đơn vị floating-point.

*\*Thanh ghi XMM*

Kiến trúc IA-32 cũng chứa 8 thanh ghi 128 bit được gọi là các thanh ghi XMM. Chúng được sử dụng bằng cách truyền các phần mở rộng SIMD vào tập lệnh.



**3. Danh sách các cờ**

*\*Cờ điều khiển*

Cờ điều khiển điều khiển hoạt động của CPU. Ví dụ, chúng có thể khiến CPU bị dừng sau mỗi lệnh thực thi, ngắt khi phát hiện tràn số, vào chế độ Virtual-8086 và vào chế độ được bảo vệ.

*\*Cờ trạng thái*

Các cờ Trạng thái phản ánh kết quả của các phép toán số học và logic được thực hiện bởi CPU. Chúng là cờ tràn(Over Flag),cờ dấu(Sign Flag),cờ Zero(Zero Flag), cờ phụ(Auxiliary Flag), cờ chẵn lẻ(Parity Flag) và Cờ nhớ(Carry Flag).

• Cờ Carry (CF) được đặt khi kết quả của phép toán số học không dấu quá lớn để phù hợp với đích.

Vd: mov eax, 00348765

mov ecx, 10000h

mul ecx ; Cờ Carry được bật CF = 1

• Cờ tràn (OF) được đặt khi kết quả của phép toán số học có dấu quá lớn hoặc quá nhỏ để phù hợp với đích.

Vd: mov al,48

mov bl,4

imul bl ;Cờ OF = 1

• Cờ ký hiệu (SF) được đặt khi kết quả của phép toán số học hoặc logic tạo ra kết quả âm.

Vd: mov AX,8000h

neg AX ; SF=1

• Cờ Zero (ZF) được đặt khi kết quả của phép toán số học hoặc logic tạo ra kết quả bằng không.

Vd: mov al,8

Cmp al,8 ; Cờ Zero được bật

• Cờ phụ (AC) được đặt khi hoạt động số học gây ra việc mang từ bit 3 đến bit 4 trong toán hạng 8 bit.

• Cờ chẵn lẻ (PF) được đặt nếu dãy byte có ý nghĩa trong kết quả chứa số chẵn bit 1. Nếu không, PF là clear. Nói chung, nó được sử dụng để kiểm tra lỗi khi có khả năng dữ liệu có thể bị thay đổi hoặc bị hỏng.

Ví dụ: mov al,80h

Mov bl,80h

Add AL,BL ; PF=1

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

8000h -0001h = 7FFFh => 0111 1111 1111 1111

SF = 0 vì MSB = 0

PF =1 vì có 8 bit 1 trong byte thấp của kết quả

ZF = 0 vì kết quả khác 0

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

OF = 1 vì trừ 1 số dương từ 1 số âm, cũng như cộng 2 số âm, mà kết quả ra dương

## Câu 2:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: MOV, INC, DEC, ADD, SUB, NEG.

***\* Lệnh MOV***

Gán giá trị nguồn vào giá trị đích

mov destination , source

Một số luật

• Cả hai toán hạng phải có cùng kích thước.

• Cả hai toán hạng không thể là toán hạng bộ nhớ.

• CS, EIP và IP không thể là toán hạng đích.

• Không thể chuyển giá trị ngay lập tức sang thanh ghi segment

Cách thanh ghi 32 bit chuyển đổi tùy mục đích người dùng

.data oneByte BYTE 78h

oneWord WORD 1234h

oneDword DWORD 12345678h

.code

mov eax,0 ; EAX = 00000000h

mov al,oneByte ; EAX = 00000078h

mov ax,oneWord ; EAX = 00001234h

mov eax,oneDword ; EAX = 12345678h

mov ax,0 ; EAX = 12340000h

Cách chuyển toán hạng 16 bit sang 32 bit

.data

count WORD 1

.code

mov ecx,0

mov cx,count (kết quả thu được ecx nhận giá trị count)

Trường hợp count là số âm (SWORD) cần mov ecx 0FFFFFFFFh

MOVZX (chuyển các toán hạng khác size vs các bit thừa điền 0 )

Các trường hợp :

MOVZX reg32,reg/mem8

MOVZX reg32,reg/mem16

MOVZX reg16,reg/mem8

VD:

mov bx,0A69Bh

movzx eax,bx ; EAX = 0000A69Bh

movzx edx,bl ; EDX = 0000009Bh

movzx cx,bl ; CX = 009Bh

MOVSX (chuyển các toán hạng khác size vs các bit thừa điền 1 )

VD:

mov bx,0A69Bh

movsx eax,bx ; EAX = FFFFA69Bh

movsx edx,bl ; EDX = FFFFFF9Bh

movsx cx,bl ; CX = FF9B

***\* INC and DEC Instructions( Cộng và trừ cho 1 )***

VD:

.data

myWord WORD 1000h

.code

inc myWord ; myWord = 1001h

mov bx,myWord

dec bx ; BX = 1000h

***\* ADD (Cộng 2 toán hạng)***

.data

var1 DWORD 10000h

var2 DWORD 20000h

.code

mov eax,var1 ; EAX = 10000h

add eax,var2 ; EAX = 30000h

( Cờ thay đổi phụ thuộc toán hạng đích )

***\* SUB (Trừ 2 toán hạng)***

VD:

.data

var1 DWORD 30000h

var2 DWORD 10000h

.code

mov eax,var1 ; EAX = 30000h

sub eax,var2 ; EAX = 20000h

***\* NEG (thêm dấu âm cho toán hạng )***

NEG reg

NEG mem

Cờ phụ thuộc toán hạng đích

## Câu 3:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: PUSH, PUSHAD, PUSHFD, POP, POPAD, POPFD, RET

## Câu 4:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: XCHG, JMP, LOOP, **truy cập Address (Địa chỉ trực tiếp và gián tiếp).**

***\* XCHG (trao đổi giá trị của 2 toán hạng cùng size)***

VD:

xchg ax,bx ; exchange 16-bit regs

xchg ah,al ; exchange 8-bit regs

xchg var1,bx ; exchange 16-bit mem op with BX

xchg eax,ebx ; exchange 32-bit regs

Trao đổi 2 giá trị ô nhớ

mov ax,val1

xchg ax,val2

mov val1,ax

***\* JMP***

Lệnh JMP gây ra chuyển vô điều kiện đến đích, được xác định bởi nhãn mã

được dịch bởi trình biên dịch thành một phần bù. Cú pháp là

JMP destination

Khi CPU thực hiện chuyển giao vô điều kiện, phần bù của đích được chuyển vào

con trỏ lệnh, khiến việc thực thi tiếp tục tại vị trí mới.

Tạo một vòng lặp Lệnh JMP cung cấp một cách dễ dàng để tạo LOOP bằng cách chuyển đến một lable ở đầu vòng lặp:

top:

.

.

Jmp top ; lặp vô tận

***\* LOOP***

Lệnh LOOP, lặp lại một khối với số lần cụ thể. ECX được tự động sử dụng làm bộ đếm và giảm dần mỗi khi vòng lặp lặp lại. Cú pháp của nó l

LOOP destination

Thực hiện lệnh LOOP bao gồm hai bước:

Đầu tiên, nó trừ 1 từ ECX. Tiếp theo, nó so sánh ECX bằng không. Nếu ECX không bằng 0, một bước nhảy được đưa đến nhãn được xác định bởi đích. Mặt khác, nếu ECX bằng 0, không có bước nhảy nào xảy ra và điều khiển chuyển đến lệnh theo vòng lặp

VD: Sử dụng 2 lệnh loop chồng nhau

.data

count DWORD ?

.code

mov ecx,100

L1:

mov count,ecx

mov ecx,20

L2:

.

.

Loop L2

mov ecx,count

loop L1

## Câu 5:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: OFFSET, PTR, TYPE, SIZEOF, LENGTHOF

***\* OFFSET (Trả về vị trí của của các biến trong ô nhớ)***

VD:

.data

bVal BYTE ?

wVal WORD ?

dVal DWORD ?

dVal2 DWORD ?

( Địa chỉ của bVal là 00404000 )

mov esi,OFFSET bVal ; ESI = 00404000

mov esi,OFFSET wVal ; ESI = 00404001

mov esi,OFFSET dVal ; ESI = 00404003

mov esi,OFFSET dVal2 ; ESI = 00404007

***\*PTR (Cắt ghép toán hạng trong trường hợp chuyển khác size)***

VD : Lấy 1 phần

.data

myDouble DWORD 12345678h

.code

mov ax , myDouble ; error ( Lỗi do khác size )

Có thể dùng lệnh mov kết hợp ptr

Mov ax , WORD PTR myDouble ( Cắt myDouble thành WORD)

Tổ chức bộ nhớ lưu biến DWORD

VD: Ghép

.data

wordList WORD 5678h,1234h

.code

Mov eax ,DWORD PTR wordList ; EAX = 12345678h (Thứ tự ghi vào eax ngược lại)

***\* TYPE (Biểu diễn số byte của 1 toán hạng)***

VD :

.data

var1 BYTE ?

var2 WORD ?

var3 DWORD ?

var4 QWORD ?

Có

TYPE var1 = 1

TYPE var2 = 2

TYPE var3 = 4

TYPE var4 = 8

***\* SIZEOF (trả về kích thước của 1 biến hoặc mảng tính bằng byte)***

VD:

.data

intArray WORD 32 DUP(0)

.code mov eax,SIZEOF intArray ; EAX = 64

***\* LENGTHOF (Xác định số phần tử trong một mảng được xác định bởi giá trị)***

VD:

.data

byte1 BYTE 10,20,30

array1 WORD 30 DUP(?),0,0

array2 WORD 5 DUP(3 DUP(?))

array3 DWORD 1,2,3,4

digitStr BYTE "12345678",0

Kết quả

LENGTHOF byte1 3

LENGTHOF array1 30 2

LENGTHOF array2 5 \* 3

LENGTHOF array3 4

LENGTHOF digitStr 9

## Câu 6:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: AND, OR, XOR, NOT, TEST, CMP

**\*Câu lệnh AND:**

**- Cú pháp: AND Destination , Source**

- Công dụng:

Lệnh này thực hiện phép AND giữa 2 toán hạng, kết quả cuối cùng chứa trong toán hạng đích.

Dùng để xoá các bit nhất định của toán hạng đích giữ nguyên các bit còn lại.

- Muốn vậy ta dùng 1 mẫu bit gọi là mặt nạ bit(Mask), các bit mặt nạ được chọn để sao cho các bit tương ứng của đích được thay đổi như mong muốn.



VD1. xoá bit dấu của AL, giữ nguyên các bit còn lại : dùng AND với 01111111b làm mặt nạ.

AL : **1**1111111b

AND **0**1111111

**0**1111111b

VD2. Đổi chữ thường thành chữ hoa

Lệnh AND cung cấp một cách dễ dàng để chuyểnchuyển một chữ cái từ chữ thường sang chữ hoa. Nếu chúng ta so sánh mã ASCII của chữ A và chữ thường a, thì rõ ràng chỉ có bit 5 là khác nhau:

0 1 **1** 0 0 0 0 1 = 61h ('a')

AND

1 1 **0** 1 1 1 1 1

1. 1 **0** 0 0 0 0 1 = 41h ('A')

**\*Câu lệnh OR:**

**- Cú pháp: OR Destination , Source**

- Công dụng: dùng để bật lên 1 bit và giữ nguyên các bit khác.



VD. OR AL, 10000001b ;Bật bit cao nhất và bit thấp nhất trong thanh ghi AL lên 1.

**\*Câu lệnh XOR:**

**- Cú pháp: XOR Destination, Source**

- Công dụng: Dùng để tạo đồ hoạ màu tốc độ cao.



- XOR có thể đảo ngược này của XOR làm cho nó trở thành một công cụ lý tưởng cho một hình thức mã hóa đối xứng đơn giản.

VD. Bật bit cao của AL 2 lần:

MOV AL , 00111011b

XOR AL, 11111111b ; AL = **11**000**1**00b

XOR AL, 11111111b ; AL = **00**111**0**11b

**\*Câu lệnh NOT:**

**- Cú pháp: NOT Destination**

- Công dụng: Thực hiện thao tác Boolean NOT trên một toán hạng đích.

-VD:

MOV AL, 11110000b  
 NOT AL ; AL = 00001111b

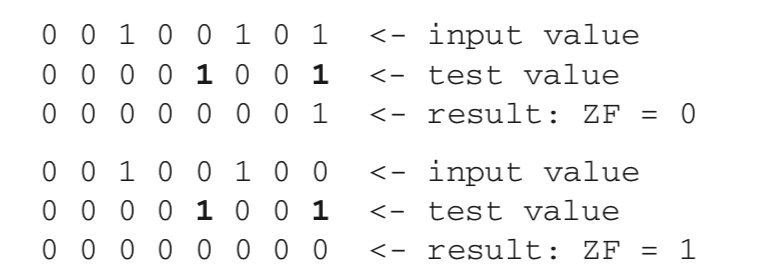
**\*Câu lệnh TEST:**

**- Cú pháp: TEST Destination, Source**

- Công dụng: Dùng để khảo sát trị của từng bit hay nhóm bit.

TEST thực hiện giống AND nhưng không làm thay đổi toán hạng đích.

TEST đặc biệt có giá trị để tìm hiểu xem các bit riêng lẻ trong toán hạng có được đặt hay không.



VD. Kiểm tra bit 13 trong DX là 0 hay là 1

Để kiểm tra 1 bit nào đó chỉ cần đặt bit 1 vào đúng vị trí bit cần kiểm tra và khảo sát cờ ZF.

(Nếu bit kiểm tra là 1 thì ZF sẽ xoá, ngược lại ZF được lập)

TEST DX, 2000h

JZ BitIs0

BitIs1 : bit 13 là 1.

BitIs0: bit 13 là 0.

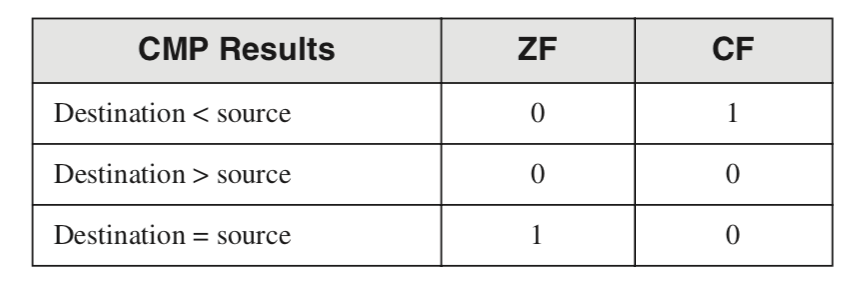
**\*Câu lệnh CMP:**

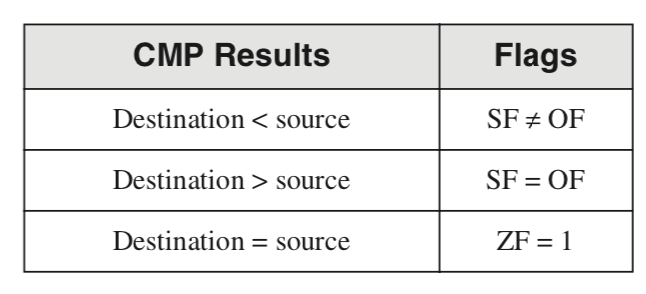
**- Cú pháp: CMP Destination, Source**

- Công dụng: So sánh toán hạng đích với toán hạng nguồn bằng cách lấy:

Toán hạng đích – Toán hạng nguồn.

- Hoạt động: Dùng phép toán trừ nhưng không có toán hạng nào bị thay đổi. Mà chỉ có tác động lên cờ. Các toán hạng của lệnh CMP không thể cùng là các ô nhớ. Lệnh CMP giống hệt lệnh SUB trừ việc toán hạng đích không thay đổi.



- Khi hai toán hạng đã ký được so sánh, các cờ Sign, Zero và Overflow chỉ ra các mối quan hệ sau đây giữa các toán hạng: 

VD:

MOV AX, 5

MOV AX, 10 ; ZF = 0, CF = 1

------------------------------------------------------------------------

MOV AX, 1000

MOV CX, 1000

CMP CX, AX ; ZF = 1, CF = 0

------------------------------------------------------------------------

MOV SI, 100

CMP SI, 0 ; ZF = 0, CF = 0

## Câu 7 + Câu 8:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: JE, JNE, JC, JZ, JNC, JP, LOOPZ, LOOPNZ, LOOPE, LOOPNE

**\*Câu lệnh nhảy có điều kiện.**

**- Cú pháp: Jconditional destination**

- Công dụng: Nhờ các lệnh nhảy có điều kiện , ta mới mô phỏng được các phát biểu có cấu trúc của các ngôn ngữ cấp cao bằng Assembly

- Phạm vi:

* Chỉ nhảy đến nhãn có khoảng cách từ -128 đến +127 byte so với vị trí hiện hành
* Dùng các trạng thái cờ để quyết định có nhảy hay không?

-Hoạt động:

* Để thực hiện một lệnh nhảy CPU nhìn vào các thanh ghi cờ.
* Nếu điều kiện của lệnh nhảy thoả mãn, CPU điều chỉnh IP trỏ đến nhãn đích các lệnh sau nhãn này sẽ được thực hiện.

**+Nhảy dựa trên các cờ:**

|  |  |  |
| --- | --- | --- |
| Lệnh nhảy | Miêu tả | Cờ |
| JZ | Nhảy nếu bằng 0 | ZF = 1 |
| JNZ | Nhảy nếu khác 0 | ZF = 0 |
| JC | Nhảy nếu có Carry | CF = 1 |
| JNC | Nhảy nếu không có Carry | CF = 0 |
| JO | Nhảy nếu tràn trị | OF = 1 |
| JNO | Nhảy nếu không tràn trị | OF = 0 |
| JS | Nhảy nếu SF = 1 | SF = 1 |
| JNS | Nhảy nếu SF = 0 | SF = 0 |
| JP | Nhảy nếu PF = 1 | PF = 1 |
| JNP | Nhảy nếu PF = 0 | PF = 0 |

**+Nhảy dựa trên sự so sánh bằng nhau:**

|  |  |
| --- | --- |
| Lệnh nhảy | Miêu tả |
| JE | Nhảy nếu 2 toán hạng bằng nhau |
| JNE | Nhảy nếu kết quả so sánh là khác nhau |
| JCXZ | Nhảy nếu CX = 0 |
| JCXZ | Nhảy nếu ECX = 0 |

**+Nhảy dựa trên các toán hạng không dấu:**

|  |  |
| --- | --- |
| Lệnh nhảy | Miêu tả |
| JA | Nhảy nếu Toán hạng trái > Toán hạng phải |
| JNBE | Nhảy nếu Toán hạng trái > Toán hạng phải |
| JAE | Nhảy nếu Toán hạng trái >= Toán hạng phải |
| JNB | Nhảy nếu Toán hạng trái >= Toán hạng phải |
| JB | Nhảy nếu Toán hạng trái < Toán hạng phải |
| JNAE | Nhảy nếu Toán hạng trái < Toán hạng phải |
| JBE | Nhảy nếu Toán hạng trái <= Toán hạng phải |
| JNA | Nhảy nếu Toán hạng trái <= Toán hạng phải |

**+Nhảy dựa trên các toán hạng có dấu:**

|  |  |
| --- | --- |
| Lệnh nhảy | Miêu tả |
| JG | Nhảy nếu Toán hạng trái > Toán hạng phải |
| JNLE | Nhảy nếu Toán hạng trái > Toán hạng phải |
| JGE | Nhảy nếu Toán hạng trái >= Toán hạng phải |
| JNL | Nhảy nếu Toán hạng trái >= Toán hạng phải |
| JL | Nhảy nếu Toán hạng trái < Toán hạng phải |
| JNGE | Nhảy nếu Toán hạng trái < Toán hạng phải |
| JLE | Nhảy nếu Toán hạng trái <= Toán hạng phải |
| JNG | Nhảy nếu Toán hạng trái <= Toán hạng phải |

**+Các VD:**

VD1: Nhập vào bàn phím một số, hiển thị số đó lên màn hình, thoát khỏi vòng lặp khi nhấn 0.

.386

.model flat,stdcall

.stack 4096

ExitProcess proto,dwExitCode:dword

INCLUDE irvine32.inc

.data

.code

main proc

L1:

MOV EAX, 10 ;CREATE 10ms DELAY

CALL Delay

CALL ReadInt

CALL writeInt

CALL Crlf

CMP Al, 0

JNE L1

invoke ExitProcess,0

main endp

end main

**\*LOOPE và LOOPZ**

**- Cú pháp:**

* **LOOPE Destination**
* **LOOPZ Destination**

- Hoạt động:

* ECX ECX – 1
* Nếu ECX > 0 và ZF = 1: Nhảy đến đích

- Công dụng: Hữu ích khi quét một mảng cho phần tử đầu tiên, điều đó không phù hợp với một gía trị nhất định.

**\*LOOPNE và LOOPNZ**

**- Cú pháp:**

* **LOOPNE Destination**
* **LOOPNZ Destination**

- Hoạt động:

* ECX ECX – 1
* Nếu ECX > 0 và ZF = 0: Nhảy đến đích

- Công dụng: Hữu ích khi quét một mảng cho phần tử đầu tiên, điều đó phù hợp với một gía trị nhất định.

**+VD LOOPNZ:**

VD1: Tìm vị trí của số dương đầu tiên trong một mảng

.386

.model flat,stdcall

.stack 4096

ExitProcess proto,dwExitCode:dword

INCLUDE irvine32.inc

.data

arr SWORD -3, -6, -1, -10, 23, 10, 98

sentinel SWORD 0

.code

main proc

MOV ESI, OFFSET arr

MOV ECX, LENGTHOF arr

NEXT:

TEST WORD PTR [ESI], 8000H

PUSHFD ; PUSH FLAGS ON STACK

ADD ESI, TYPE arr

POPFD ; POP FLAGS ON STACK

LOOPNZ NEXT

JNZ QUIT

SUB ESI, TYPE arr

QUIT:

MOV EAX, LENGTHOF arr

SUB EAX, ECX

CALL WriteInt

invoke ExitProcess,0

main endp

end main

## Câu 8 (Bao gồm trong câu 7):

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: JG, JL, JNG, JA, JB, JNA, LOOPZ, LOOPNZ, LOOPE, LOOPNE

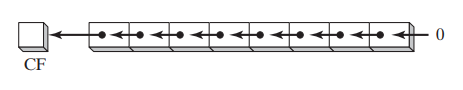
## Câu 9:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR

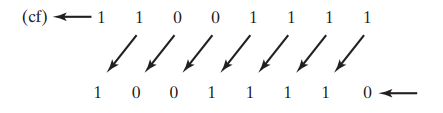
**\*SHL Instruction**

Lệnh SHL (shift trái) thực hiện dịch chuyển trái logic trên toán hạng đích, điền vào

bit thấp nhất bằng 0. Bit cao nhất được chuyển sang cờ Carry và bit nằm trong cờ Carry bị loại bỏ:



Nếu bạn thay đổi 11001111 còn lại 1 bit, nó sẽ trở thành 10011110:



Toán hạng đầu tiên trong SHL là đích và thứ hai là số ca:

**SHL *destination,count***

Sau đây liệt kê các loại toán hạng được cho phép theo hướng dẫn này:

**SHL reg,imm8**

**SHL mem,imm8**

**SHL reg,CL**

**SHL mem,CL**

Bộ xử lý x86 cho phép im8 là bất kỳ số nguyên nào trong khoảng từ 0 đến 255. Ngoài ra, thanh ghi CL có thể chứa một số lượng ca. Các định dạng hiển thị ở đây cũng áp dụng cho SHR, SAL, SAR, ROR, ROL, Hướng dẫn RCR và RCL.

Ví dụ Trong các hướng dẫn sau, BL được dịch chuyển một lần sang trái. Bit cao nhất được sao chép vào cờ Carry và vị trí bit thấp nhất được gán 0:

**mov bl,8Fh ;bl=10001111**

**shl bl,1 ;CF=1, bl=00011110b**

Khi một giá trị được dịch chuyển sang trái nhiều lần, cờ Carry chứa bit cuối cùng được dịch chuyểntrong số bit đáng kể nhất (MSB). Trong ví dụ sau, bit 7 không kết thúc trong cờ Carry vì nó được thay thế bằng bit 6 (một số không)

**Mov al,10000000b**

**Shl al,2 ;CF= 0, AL=00000000b**

Tương tự, khi một giá trị được dịch chuyển sang phải nhiều lần, cờ Carry chứa bit cuối cùng để được chuyển ra khỏi bit có ý nghĩa nhỏ nhất (LSB).

**-Fast Multiplication**

Fast Multipcation được thực hiện khi bạn dịch chuyển một số bit bit trong một

hướng trái (về phía MSB). Ví dụ, SHL có thể thực hiện phép nhân với lũy thừa bằng 2. Dịch chuyển bất kỳ toán hạng nào còn lại bởi n bit sẽ nhân toán hạng 2^n. Ví dụ: dịch chuyển số nguyên 5 còn lại 1 bit mang lại sản phẩm của 5\* 2^1 =10:

**mov dl, 5 ; 000000101 =5**

**shl dl,1 ;000001010=10**

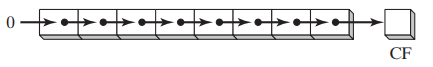
Nếu nhị phân 00001010 (thập phân 10) được dịch chuyển sang trái bởi hai bit, kết quả sẽ giống như nhân 10 với 2^2

**Mov dl, 10 ;00001010=10**

**Shl dl, 2 ;00101000=40**

**\*SHR Instruction**

Lệnh SHR (shift phải) thực hiện dịch chuyển phải logic trên toán hạng đích, thay thế bit cao nhất bằng 0. bit thấp nhất được sao chép vào cờ Carry và bit đó trước đây trong cờ Carry bị mất:



SHR sử dụng các định dạng hướng dẫn tương tự như SHL. Trong ví dụ sau, 0 từ bit thấp nhất trong AL được sao chép vào cờ Carry và bit cao nhất trong AL được điền bằng 0:

**mov al,0D0h ; AL = 11010000b**

**shr al,1 ; AL = 01101000b, CF = 0**

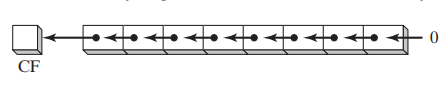
Trong hoạt động nhiều ca, bit cuối cùng được chuyển ra khỏi vị trí 0 (LSB) kết thúc ở Cờ carry:

**mov al,00000010b**

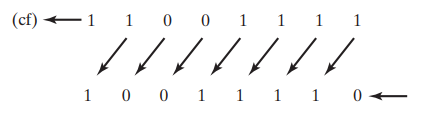
**shr al,2 ; AL = 00000000b, CF = 1**

**\*SAL and SAR Instructions**

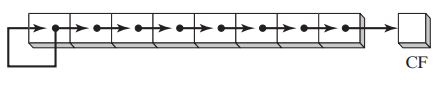
Lệnh SAL (số học trái số) hoạt động giống như lệnh SHL. Đối với mỗi ca làm việc, SAL chuyển từng bit trong toán hạng đích sang vị trí bit cao nhất tiếp theo. Bit thấp nhất được chỉ định 0. Bit cao nhất được chuyển đến cờ Carry và bit trong cờ Carry bị loại bỏ:



Nếu bạn dịch chuyển nhị phân 11001111 sang trái một bit, nó sẽ trở thành 10011110:



Lệnh SAR (quyền số học thay đổi) thực hiện dịch chuyển số học đúng về đích của nó toán hạng:



Các toán hạng cho SAL và SAR giống hệt với toán hạng cho SHL và SHR. Sự thay đổi có thể là lặp đi lặp lại, dựa trên bộ đếm trong toán hạng thứ hai:

SAR destination,count

Ví dụ sau đây cho thấy cách SAR nhân đôi bit dấu. AL là âm trước và sau khi nó được chuyển sang bên phải:

**mov al,0F0h ; AL = 11110000b (-16)**

**sar al,1 ; AL = 11111000b (-8), CF = 0**

**Ví dụ tổng quát**

**Mov al,6Bh**

**Shr al,1 ;al=35h**

**Shl al,3 ;al=A8h**

**Mov al,8Ch**

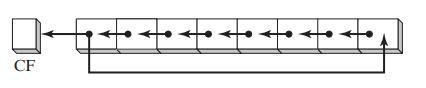
**Sar al,1 ;al=C6h**

**Sar al,3 ;al=F8h**

**\*ROL Instruction**

Xoay bitwise xảy ra khi bạn di chuyển các bit theo kiểu vòng tròn. Trong một số phiên bản, bit để lại một đầu của số ngay lập tức được sao chép vào đầu kia. Một kiểu quay khác sử dụng cờ Carry làm điểm trung gian cho các bit được dịch chuyển.

Lệnh ROL (xoay trái) dịch chuyển từng bit sang trái. Bit cao nhất được sao chép vào cờ Carry và vị trí bit thấp nhất. Định dạng hướng dẫn giống như đối với SHL:



Xoay bit không mất bit. Một bit được quay ở một đầu của một số xuất hiện lại ở đầu kia kết thúc. Lưu ý trong ví dụ sau về cách bit cao được sao chép vào cả cờ Carry và bit vị trí 0:

**mov al,40h ; AL = 01000000b**

**rol al,1 ; AL = 10000000b, CF = 0**

**rol al,1 ; AL = 00000001b, CF = 1**

**rol al,1 ; AL = 00000010b, CF = 0**

**ví dụ:**

**mov al,1111000b**

**rol al,1 ;AL=11100001b**

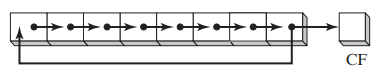
**mov dl,3Fh**

**rol dl,4 ;dl=F3h**

**\*ROR Instruction**

Lệnh ROR (xoay phải) dịch chuyển từng bit sang phải và sao chép bit thấp nhất

vào cờ Carry và vị trí bit cao nhất. Định dạng hướng dẫn giống như cho SHL:



Trong các ví dụ sau, lưu ý cách bit thấp nhất được sao chép vào cả cờ Carry và

vị trí bit cao nhất của kết quả:

**mov al,01h ; AL = 00000001b**

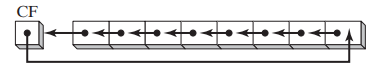
**ror al,1 ; AL = 10000000b, CF = 1**

**ror al,1 ; AL = 01000000b, CF = 0**

**\*RCL and RCR Instructions**

Lệnh RCL (xoay mang trái) dịch chuyển từng bit sang trái, sao chép cờ Carry sang

LSB và sao chép MSB vào cờ Carry



Nếu chúng ta tưởng tượng cờ Carry là một bit bổ sung được thêm vào cao cấp của toán hạng, RCL trông giống như một thao tác xoay trái. Trong ví dụ sau, lệnh CLC xóa cờ Carry. Các lệnh RCL đầu tiên di chuyển bit BL cao vào cờ Carry và dịch chuyển các bit khác sang trái.

Lệnh RCL thứ hai di chuyển cờ Carry vào vị trí bit thấp nhất và dịch chuyển cái khác bit còn lại:

**clc ; CF = 0**

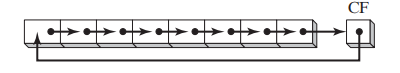
**mov bl,88h ; CF,BL = 0 10001000b**

**rcl bl,1 ; CF,BL = 1 00010000b**

**rcl bl,1 ; CF,BL = 0 00100001b**

-Lệnh RCR (xoay mang bên phải) dịch chuyển từng bit sang phải, sao chép

cờ Carry vào MSB và sao chép LSB vào cờ Carry



Như trong trường hợp RCL, nó giúp trực quan hóa số nguyên trong hình này dưới dạng giá trị 9 bit, với cờ Carry bên phải của LSB.

Ví dụ mã sau sử dụng STC để đặt cờ Carry; Sau đó, nó thực hiện một vòng quay

hoạt động đúng trên thanh ghi AH:

**stc ; CF = 1**

**mov ah,10h ; AH, CF = 00010000 1**

**rcr ah,1 ; AH, CF = 10001000 0**

## Câu 10:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: MUL, IMUL, DIV, IDIV, CBW, CWD, CDQ

**\*MUL Instruction**

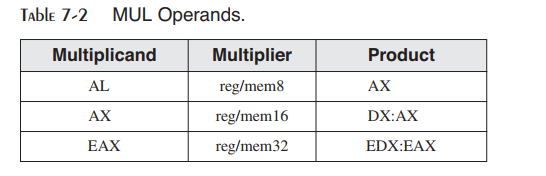
Trong chế độ 32 bit, lệnh MUL (bội số không dấu) có ba phiên bản: Phiên bản đầu tiên phiên bản nhân một toán hạng 8 bit bằng thanh ghi AL. Phiên bản thứ hai nhân 16 bit toán hạng bằng thanh ghi AX và phiên bản thứ ba nhân toán hạng 32 bit bằng EAX Đăng ký. Số nhân và bội số phải luôn có cùng kích thước và sản phẩm là gấp đôi kích thước của chúng. Ba định dạng chấp nhận thanh ghi và toán hạng bộ nhớ, nhưng không phải ngay lập tức Toán hạng:

**MUL reg/mem8**

**MUL reg/mem16**

**MUL reg/mem32**

Toán hạng đơn trong lệnh MUL là cấp số nhân. Bảng 7-2 hiển thị bội số và sản phẩm mặc định, tùy thuộc vào kích thước của cấp số nhân. Bởi vì toán hạng đích là gấp đôi kích thước của bội số và bội số, không thể xảy ra tràn. MUL thiết lập Carry và Cờ tràn nếu nửa trên của sản phẩm không bằng không. Cờ Carry là thông thường được sử dụng cho số học không dấu, vì vậy chúng tôi sẽ tập trung vào nó ở đây. Ví dụ, khi AX được nhân với toán hạng 16 bit, sản phẩm được lưu trữ trong các thanh ghi DX và AX kết hợp. Đó là, 16 bit cao của sản phẩm được lưu trữ trong DX và 16 bit thấp được lưu trữ trong AX. Cờ Carry được đặt nếu DX không bằng 0, điều đó cho chúng tôi biết rằng sản phẩm sẽ không phù hợp với nửa dưới của toán hạng đích ngụ ý.



**Ví dụ:**

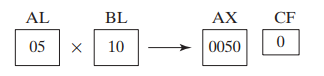
Các câu lệnh sau nhân AL với BL, lưu trữ sản phẩm trong AX. Cờ Carry rõ ràng

(CF = 0) vì AH (nửa trên của sản phẩm) bằng 0:

**mov al,5h**

**mov bl,10h**

**mul bl ; AX = 0050h, CF = 0**



Các câu lệnh sau nhân giá trị 16 bit 2000h với 0100h. Cờ Carry được đặt bởi vì phần trên của sản phẩm (nằm trong DX) không bằng không:

**.data**

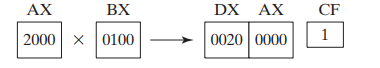
**val1 WORD 2000h**

**val2 WORD 0100h**

**.code**

**mov ax,val1 ; AX = 2000h**

**mul val2 ; DX:AX = 00200000h, CF = 1**



**\*IMUL Instruction**

Lệnh IMUL (bội số đã ký) thực hiện phép nhân số nguyên đã ký. Không giống như Hướng dẫn MUL, IMUL bảo quản dấu hiệu của sản phẩm. Nó làm điều này bằng cách ký mở rộng bit cao nhất của nửa dưới của sản phẩm vào các bit trên của sản phẩm. Hướng dẫn x86 set hỗ trợ ba định dạng cho lệnh IMUL: một toán hạng, hai toán hạng và ba toán hạng. Trong định dạng một toán hạng, số nhân và bội số có cùng kích thước và sản phẩm có kích thước gấp đôi chúng.

**IMUL reg/mem8 ; AX = AL \* reg/mem8**

**IMUL reg/mem16 ; DX:AX = AX \* reg/mem16**

**IMUL reg/mem32 ; EDX:EAX = EAX \* reg/mem32**

**-Using IMUL in 64-Bit Mode**

Trong chế độ 64 bit, bạn có thể sử dụng toán hạng 64 bit với lệnh MUL. Trong hai toán hạng định dạng, thanh ghi 64 bit hoặc toán hạng bộ nhớ được nhân với RDX, tạo ra 128 bit sản phẩm mở rộng đăng nhập trong RDX: RAX. Trong ví dụ tiếp theo, RBX được nhân với RAX, tạo ra sản phẩm 128 bit là 16.

**mov rax,-4**

**mov rbx,4**

**imul rb ; RDX = 0FFFFFFFFFFFFFFFFh, RAX = -16**

Nói cách khác, số thập phân 16 được biểu thị dưới dạng thập lục phân FFFFFFFFFFF0 trong RAX và RDX chỉ chứa một phần mở rộng của bit thứ tự cao RAX, còn được gọi là bit dấu của nó.

Định dạng ba toán hạng cũng có sẵn trong chế độ 64 bit. Trong ví dụ tiếp theo, chúng tôi nhân lên bội số (16) với 4, tạo 64 trong thanh ghi RAX:

**.data**

**multiplicand QWORD -16**

**.code**

**imul rax, multiplicand, 4 ; RAX = FFFFFFFFFFFFFFC0 (-64)**

**Ví dụ:**

Các hướng dẫn sau đây nhân 48 với 4, tạo 192 trong AX. Mặc dù sản phẩm là

đúng, AH không phải là phần mở rộng dấu hiệu của AL, vì vậy cờ tràn được đặt:

**mov al,48**

**mov bl,4**

**imul bl ; AX = 00C0h, OF = 1**

**\*DIV Instruction**

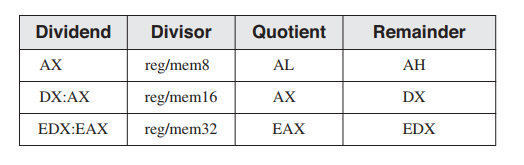
Trong chế độ 32 bit, lệnh DIV (chia không dấu) thực hiện 8 bit, 16 bit và 32 bit phép chia số nguyên không dấu. Thanh ghi đơn hoặc toán hạng bộ nhớ là ước số. Các định dạng là

**DIV reg/mem8**

**DIV reg/mem16**

**DIV reg/mem32**

Bảng sau đây cho thấy mối quan hệ giữa số chia, ước số, thương số và phần còn lại:



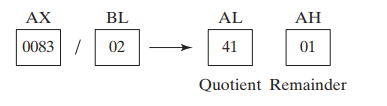
Trong chế độ 64 bit, lệnh DIV sử dụng RDX: RAX làm số chia và cho phép ước số là một thanh ghi 64 bit hoặc toán hạng bộ nhớ. Thương số được lưu trữ trong RAX và phần còn lại trong RDX.

**Ví dụ:**

**mov ax,0083h ; dividend**

**mov bl,2 ; divisor**

**div bl ; AL = 41h, AH = 01h**



**\*The IDIV Instruction**

Lệnh IDIV (phân chia có dấu) thực hiện phân chia số nguyên có dấu, sử dụng cùng các toán hạng như DIV. Trước khi thực hiện phân chia 8 bit, số chia (AX) phải được gia hạn hoàn toàn. Phần còn lại luôn có cùng dấu hiệu với số chia.

**Ví dụ:**

-Chia -48 cho 5 bằng 16bit

**Mov ax,-48**

**Cwd ;extend AX into DX**

**Mov bx,5**

**Idiv bx ;AX= -9,DX= -3**

-Chia -48 cho 5 bằng 32 bit

**Mov eax, -48**

**Cdq ;extend EAX into EDX**

**Mov ebx,5**

**Idiv ebx ;EAX = -9, EDX = -3**

**Unsigned Arithmetic Expressions**

**Ví dụ:** var4 = (var1 +var2) \*var3

**Mov eax, var1**

**Add eax,var2 ;EAX = var1+ var2**

**Mul var3 ; EAX =EAX \*var3**

**Jc TooBig ; check for carry**

**Mov var4,eax ;save product**

**-Sign Arithmetic Expressions**

**Ví dụ:** eax =(-var1\*var2)+var3

**Mov eax,var1**

**Neg eax**

**Imul var2**

**Jo TooBig ;check for overflow**

**Add eax,var3**

**Jo TooBig ; check for overflow**

**\* CBW, CWD, CDQ**

Intel cung cấp ba hướng dẫn mở rộng ký hiệu: CBW, CWD và CDQ. Hướng dẫn CBW (chuyển đổi byte thành từ) mở rộng bit dấu của AL thành AH, giữ nguyên dấu hiệu Số. Ở phần tiếp theo ví dụ, 9Bh (tính theo AL) và FF9Bh (tính theo AX) đều bằng nhau − 101 thập phân:

**.data**

**byteVal SBYTE -101 ; 9Bh**

**.code**

**mov al,byteVal ; AL = 9Bh**

**cbw ; AX = FF9Bh**

Lệnh CWD (chuyển đổi từ thành từ kép) mở rộng bit dấu của AX thành DX:

**.data**

**wordVal SWORD -101 ; FF9Bh**

**.code**

**mov ax,wordVal ; AX = FF9Bh**

**cwd ; DX:AX = FFFFFF9Bh**

Lệnh CDQ (chuyển đổi từ kép thành tứ giác) mở rộng bit dấu của EAX thành EDX:

**.data**

**dwordVal SDWORD -101 ; FFFFFF9Bh**

**.code**

**mov eax,dwordVal**

**cdq ; EDX:EAX = FFFFFFFFFFFFFF9Bh**

## Câu 11:

- Giải thích hoạt động, tác động vào các cờ, cho ví dụ minh hoạ của các lệnh sau: COORD, **DUP,** PROC, MACRO

**+ Cấu trúc COORD:** được định nghĩa trong các tọa độ màn hình của X và Y của Windows API. Trường X có độ lệch bằng 0 so với phần tử đầu của cấu trúc, trường Y có độ lệch bằng 2

COORD STRUCT

X WORD ? ; offset 00

Y WORD ? ; offset 02

COORD ENDS

+ Lệnh **PROC:**

- Khởi tạo một procedure (có thể bao gồm danh sách tham số của procedure đó).

Cú pháp chung:

ProcedureName PROC

parameter1 : TYPE (BYTE, WORD,…)

parameter1 : TYPE (BYTE, WORD,…)

....

**Ví dụ:**

main PROC

; Get the Console standard output handle:

INVOKE GetStdHandle,STD\_OUTPUT\_HANDLE

mov outHandle,eax

; Set the colors of adjacent cells:

INVOKE WriteConsoleOutputAttribute,

outHandle, ADDR attributes,

BufSize, xyPos, ADDR cellsWritten

; Write character codes 1 through 20:

INVOKE WriteConsoleOutputCharacter,

outHandle, ADDR buffer, BufSize,

xyPos, ADDR cellsWritten

INVOKE ExitProcess,0 ; end program

main ENDP

END main

**+ MACROS:**

- Macro là một khối được đặt tên của ngôn ngữ assembly.

- Macro được định nghĩa một lần, nó có thể được gọi lại rất nhiều lần trong chương trình như mong muốn của người lập trình. Khi gọi lại (invoke) một thủ tục macros, một bản copy của đoạn code đó được chèn vào vị trí mà nó đã được gọi. Loại chèn code tự động này chỉ cũng được biết đến như mở rộng nội tuyến.

- Một macro được định nghĩa sử dụng chỉ thị MACRO và ENDM. Cấu trúc như sau:

[macro name] MACRO parameter-1, parameter-2, …

Statement-list

ENDM

- Nên đặt tên 1 macro bắt đầu bởi ký tự m. Ví dụ: mNewLine

- Parameter: tham số của marco được đặt tên cho các đối số văn bản được truyền cho lời gọi. đối số này có thể là số nguyên, tên biến, hay là các giá tri khác nhưng tiền xử lý coi chúng là văn bản. parameter không chứa thông tin loại, vì vậy tiền xử lý sẽ không kiểm tra xem chúng có đúng không. Nếu một lỗi xảy ra, nó sẽ bị bắt bởi trình biên dịch.

- Một macro được invoke bằng cách chèn tên của nó trong chương trình, theo sau bằng các đối số. Cú pháp

[macro name] argument-1, argument-2, ….

- Trong đó macro name là tên của macro đã được định nghĩa trước khi gọi trong mã nguồn. Mỗi đối số là giá trị văn bản thay thế một parameter của macro. Trật tự của các đối số phải tương ứng với parameter, nhưng số lượng đối số không cần phải tương ứng với số lượng parameter. Nếu có quá nhiều đối số được truyền vào, assembler sẽ đưa ra cảnh báo, nếu có ít hơn đối số được truyền vào, parameter tương ứng sẽ được bỏ trống.

- Tốc độ của macro sẽ nhanh hơn procedure vì các procedure sẽ có thêm chi phí hướng dẫn call và ret, tuy nhiên khi sử dụng macro nhiều lần sẽ làm tăng kích thước của chương trình.

- Debugging một chương trình sử dụng macro có sự khác biệt. Sau khi assembling chương trình, kiểm tra các file liên kết của nó để chắc rằng mỗi macro được mở rộng đúng cách, và lựa chọn Go to Disassembly từ popup menu. Mỗi lời gọi macro theo sau bởi code được sinh ra từ macro.