ЛАБОРАТОРНА РОБОТА №5

Масиви. Підпрограми

|  |
| --- |
|  |

***Мета:*** Вивчення операцій, пов’язаних з побудовою одновимірних та двовимірних масивів. Пошук елементів в двовимірному масиві. Сортування масива. Вивчити команди, що реалізують підпрограми на асемблері.

Короткі теоретичні відомості

Масиви

Дамо формальне визначення: **масив** *—* структурований тип даних, що складається з деякого числа елементів одного типу.

Опис та ініціалізація масиву в програмі

Спеціальних засобів опису масивів в програмах асемблера, звичайно, немає. При необхідності використовувати масив в програмі, його потрібно моделювати одним з наступних способів:

-перерахуванням елементів масиву в полі операндів однієї з директив опису даних. При перерахуванні елементи розділяються комами. Наприклад:

array db 2,4,3 ;масив з 3 елементів.

;Розмір кожного елемента 4 байти

- використовуючи оператор повторення dup. Наприклад:

array db 9 dup (0) ;масив и 9 нульовых елементів.

;розмір кожного елементу 2 байти

Такий спосіб визначення використовується для резервування пам'яті з метою розміщення і ініціалізації елементів масиву;

Використовуючи директиви label і rept. Пара цих директив може полегшити опис великих масивів в пам'яті і підвищити наочність такого опису. Директива rept відноситься до макрозасобів мови асемблера і викликає повторення вказане число разів рядків, розміщених між директивою і рядком endm. Наприклад, визначимо масив байт в області пам'яті, позначеної ідентифікатором mas\_b. В даному випадку директива label визначає символічне ім'я mas\_b аналогічно тому, як це роблять директиви резервування і ініціалізації пам'яті. Позитивна якість директиви label в тому, що вона не резервує пам'ять, а лише визначає характеристики об'єкту. В даному випадку об'єкт – це елемент пам'яті. Використовуючи декілька директив label, записаних одна за одною, можна присвоїти одній і тій же області пам'яті різні імена і типи, що і зроблено в наступному фрагменті:

n=0

...

array \_k label byte

array \_r label word

rept 4

dw 0f1f0h

endm

В результаті в пам'яті буде створена послідовність з чотирьох слів f1f0. Цю послідовність можна трактувати як масив байт або слів залежно від того, яке ім'я області ми використовуватимемо в програмі – mas\_b або mas\_w;

Використанням циклу для ініціалізації значеннями області пам'яті, яку можна буде згодом трактувати як масив. Подивимося на прикладі лістингу 1, яким чином це робиться.

Приклад 5.1.: Заповнення масиву числами.

.MODEL small

.STACK 256

.data

array db 9 dup (?) ;Резервуємо пам'ять під масив

.code

main:

mov ax,@data

mov ds,ax

mov es,ax

mov cx,9 ;Ініціалізуємо лічильник

lea di, array ;Завантажуємо адресу масиву в регістр

cld ;Очищуємо пропор напрямку

go:

mov al, cl ;Завантажуємо в al значення лічильника

stosb ;Додаємо значення в al до масиву

loop go ;Повторюємо цикл

mov cx,9 ;Ініціалізуємо лічильник

lea si, array ;Завантажуємо адресу масиву в регістр

mov ah,02h ;Функція DOS виведення символу

show:

lodsb ;Завантаження елемента в al

mov dl, al ;Завантажуємо в dl елемент масиву

add dl,30h ;Перетворюємо число в символ

int 21h ;Виводимо символ на екран

loop show

exit:

mov ax,4c00h

int 21h

end main

Доступ до елементів масиву

При роботі з масивами необхідно чітко уявляти собі, що всі елементи масиву розташовуються в пам'яті комп'ютера послідовно. Саме по собі таке розташування нічого не говорить про призначення і порядок використання цих елементів. І лише програміст за допомогою складеного ним алгоритму обробки визначає те, як потрібно трактувати цю послідовність байт, що складають масив. Так, одну і ту ж область пам'яті можна трактувати як одновимірний масив, і одночасно ті ж самі дані можуть трактуватися як двовимірний масив. Все залежить тільки від алгоритму обробки цих даних в конкретній програмі. Самі по собі дані не несуть ніякої інформації про свій «смисловий», або логічний тип. Пам'ятаймо про цей принциповий момент.

Ці ж міркування можна розповсюдити і на індекси елементів масиву. Асемблер не підозрює про їх існування і йому абсолютно все одно, які їх чисельні смислові значення. Для того, щоб локалізувати певний елемент масиву, до його імені потрібно додати індекс. Оскільки ми моделюємо масив, то повинні поклопотатися і про моделювання індексу. У мові асемблера індекси масивів — це звичайні адреси, але з ними працюють особливим чином. Іншими словами, коли при програмуванні на асемблері ми говоримо про індекс, то, швидше за все, маємо на увазі під цим не номер елементу в масиві, а деяку адресу. Давайте ще раз звернемося до опису масиву. Наприклад, в програмі статично визначена послідовність даних: array dw 0,6,2,8,4,9

Хай ця послідовність чисел трактується як одновимірний масив. Розмірність кожного елементу визначається директивою dw, тобто вона рівна двом байтам. Щоб дістати доступ до третього елементу, потрібно до адреси масиву додати 6. Нумерація елементів масиву в асемблері починається з нуля. Тобто в нашому випадку мова, фактично, йде про 4-й елемент масиву - 8, але про це знає тільки програміст; мікропроцесору в даному випадку все одно – йому потрібна тільки адреса. У загальному випадку для отримання адреси елементу в масиві необхідно початкову (базову) адресу масиву додати до добутку індексу (номер елементу мінус одиниця) цього елементу на розмір елементу масиву.

Архітектура мікропроцесора надає достатньо зручні програмно-апаратні засоби для роботи з масивами. До них відносяться базові і індексні регістри, адресації даних, що дозволяють реалізувати декілька режимів. Використовуючи дані режими адресації, можна організувати ефективну роботу з масивами в пам'яті. Пригадаймо ці режими:

1. індексна адресація зі зміщенням — режим адресації, при якому ефективна адреса формується з двох компонентів:
2. постійного (базового) - вказівкою прямої адреси масиву у вигляді імені ідентифікатора, що позначає початок масиву;
3. змінного (індексного) — вказівкою імені індексного регістра. Наприклад:

array dw 0,2,3,5,4,6

1. mov si,4 ;помістити 3-й элемент масиву array в регістр ax:

mov ax. array [si]

1. базова індексна адресація зі зміщенням — режим адресації, при якому ефективна адреса формується максимум з трьох компонентів:
2. постійного (необов'язкового компоненту), в якості якого може виступати пряма адреса масиву у вигляді імені ідентифікатора, що позначає початок масиву, або безпосереднє значення;
3. змінного (базового) — вказівкою імені базового регістра;
4. змінного (індексного) — вказівкою імені індексного регістра.

Цей вид адресації зручно використовувати при обробці двовимірних масивів. Приклад використання цієї адресації ми розглянемо нижче при вивченні особливостей роботи з двовимірними масивами.

Нагадаємо, що в якості базового регістра може використовуватися будь-який з восьми регістрів загального призначення. В якості індексного регістра також можна використовувати будь-який регістр загального призначення, за винятком esp/sp.

Мікропроцесор дозволяє **масштабувати індекс***.* Це означає, що якщо вказати після імені індексного регістра знак множення «\*» з подальшою цифрою 2, 4 або 8, то вміст індексного регістра буде множитися на 2, 4 або 8, тобто масштабуватися. Застосування масштабування полегшує роботу з масивами, які мають розмір елементів, рівний 2, 4 або 8 байтам, оскільки мікропроцесор сам проводить корекцію індексу для отримання адреси чергового елементу масиву. Нам потрібно лише завантажити в індексний регістр значення необхідного індексу (рахуючи від 0). До речі сказати, можливість масштабування з'явилася в мікропроцесорах Intel, починаючи з моделі i486. З цієї причини в прикладі програми, що розглядається нижче, розміщена директива .486. Її призначення, як і директиви .386, що раніше використовувалася, в тому, щоб дати асемблеру вказівку враховувати додаткові можливості нових мікропроцесорів при формуванні машинних команд.

Як приклад використання масштабування розглянемо лістинг 2, в якому проглядається масив, що складається із слів, і проводиться порівняння цих елементів з нулем. Виводиться відповідне повідомлення.

Приклад 5.2.: Проглядання масиву слів з використанням масштабування

MASM

MODEL small

STACK 256

.data ;початок сегменту даних

;тексти повідомлень:

message1 db 'не рівний 0!$',0ah,0dh

message2 db 'рівний 0!$',0ah,0dh

message3 db 0ah,0dh,'eлемент $'

array dw 1,1,2,3,5,8,0,1,0,0 ;початковий масив

.code

.486 ;це обов'язково

main:

mov ax,@data

mov ds,a ;зв’язок ds з сегментом даних

xor ax, ax ;онулення ax

prepare:

mov cx,10 ;значення лічильника циклу в cx

mov esi,0 ;індекс у esi

compare:

mov dx,array[esi\*2] ;перший елемент масиву в dx

cmp dx,0 ;порівняння dx з 0

je equal ;перехід, якщо рівно

not\_equal: ;не рівно

mov ah,09h ;вивід повідомлення message3 на екран

lea dx,message3

int 21h

mov ah,02h ;вивід номера елементу масиву на екран

mov dx,si

add dl,30h

int 21h

mov ah,09h

lea dx,message1

int 21h

inc esi ;на наступний елемент

dec cx ;умова для виходу з циклу

jcxz exit ;cx=0? Якщо так - на вихід

jmp compare ;ні - повторити цикл

equal ;рiвнe 0

mov ah,09h ;вивід повідомлення message3 на екран

lea dx,message3

int 21h

mov ah,02h

mov dx,si

add dl,30h

int 21h

mov ah,09h ;вивід повідомлення message2 на екран

lea dx,message2

int 21h

inc esi ;на наступний елемент

dec cx ;всі елементи опрацьовані?

Jcxz exit

Jmp compare

exit:

mov ax,4c00h ;стандартний вихід

int 21h

end main ;кінец програми

Ще декілька слів про умови:

1. Якщо для опису адреси використовується тільки один регістр, то мова йде про базову адресацію, і цей регістр розглядається як базовий:

mov al[ebx] ;переслати байт з області даних, адреса якої

;знаходиться в регістрі ebx

1. Якщо для завдання адреси в команді використовується пряма адресація (у вигляді ідентифікатора) у поєднанні з одним регістром, то мова йде про індексну адресацію. Регістр вважається індексним, і тому можна використовувати масштабування для отримання адреси потрібного елементу масиву:

add eax,mas[ebx\*2] ;додати вміст еах з подвійним словом за

;адресою mas + (ebx)\*2

1. Якщо для опису адреси використовуються два регістри, то мова йде про базово-індексну адресацію. Лівий регістр розглядається як базовий, а правий — як індексний. У загальному випадку це не принципово, але якщо ми використовуємо масштабування з одним з регістрів, то він завжди є індексним. Але краще дотримуватися певних угод. Пам'ятаймо, що застосування регістрів ebp/bp і esp/sp за замовчанням має на увазі, що сегментна складова адреси знаходиться в регістрі ss.

Відмітимо, що базово-індексну адресацію не забороняється поєднувати з прямою адресацією або вказівкою безпосереднього значення. Адреса тоді формуватиметься як сума всіх компонентів.

Наприклад:

mov ax,mas[ebx][ecx\*4] ;адреса операнда рівна ;[mas+(ebx)+(ecx)\*4]

sub dx[ebx+4[ecx\*8] ;адреса операнда рівна ;[(ebx)+4+(ecx)\*8]

Але маймо на увазі, що масштабування ефективне лише тоді, коли розмірність елементів масиву рівна 2, 4 або 8 байтам. Якщо ж розмірність елементів інша, то організовувати звернення до елементів масиву потрібно звичайним способом, як описано вище.

Розглянемо приклад роботи з масивом з п'яти трьохбайтових елементів (лістинг 3). Молодший байт в кожному з цих елементів є якимсь лічильником, а старші два байти – щось ще, що не має для нас ніякого значення. Необхідно послідовно обробити елементи даного масиву, збільшивши значення лічильників на одиницю.

Приклад 5.3.Обробка масиву елементів з непарною довжиною

MASM

MODEL small ;модель пам'яті

STACK 256 ;розмір стеку

.data ;початок сегменту даних

N=6 ;кількість елементів масиву

array db 6 dup (4 dup (0))

.code ;сегмент коду

main: ;точка входу в програму

mov ax,@data

mov ds,ax

xor ax, ax ;обнулення ax

mov si,0 ;0 у si

mov cx,N ;N у cx

go:

mov dl,array[si] ;перший байт поля в dl

inc dl ;збільшення dl на 1 (по умові)

mov array[si],dl ;заслати назад в масив

add si,4 ;зсув на наступний елемент масиву

loop go ;повтор циклу

mov si,0 ;підготовка до виводу на екран

mov cx,N

show: ;вивід на екран вмісту перших байтів полів

mov dl,array[si]

add dl,30h

mov ah,02h

int 21h

loop show

exit:

mov ax,4c00h ;стандартный вихід

int 21h

end main ;кінець програми

Двовимірні масиви

З представленням одновимірних масивів в програмі на асемблері і організацією їх обробки все досить просто. А як бути, якщо програма повинна обробляти двовимірний масив? Всі проблеми виникають як і раніше через те, що спеціальних засобів для опису такого типу даних в асемблері немає. Двовимірний масив потрібно моделювати. На описі самих даних це майже ніяк не відбивається – пам'ять під масив виділяється за допомогою директив резервування і ініціалізації пам'яті. Безпосередньо моделювання обробки масиву проводиться в сегменті коду, де програміст, описуючи асемблеру алгоритм обробки, визначає, що деяку область пам'яті необхідно трактувати як двовимірний масив. При цьому ви вільні у виборі того, як розуміти розташування елементів двовимірного масиву в пам'яті: по рядках або по стовпцях.

Організувати адресацію двовимірного масиву логічно, використовуючи розглянуту нами раніше базово-індексну адресацію. При цьому можливі два основні варіанти вибору компонентів для формування ефективної адреси:

1. поєднання прямої адреси, як базового компоненту адреси, і двох індексних регістрів для зберігання індексів:

mov ax,mas[ebx][esi]

1. поєднання двох індексних регістрів, один з яких є і базовим, і індексним одночасно, а інший – тільки індексним:

mov ax[ebx][esi]

У програмі це виглядатиме приблизно так:

;Фрагмент програми вибірки елементу масиву mas(2.3) і його обнулення

.data

mas db 9,54,32,56,67,89,7,99,67,8,65,78,21,8,56,34

i = 2

j = 3

.code

**- - -**

.386

mov si,4\*1\*i

mov di,j

mov al,mas[si][di] ;в al елемент mas(2,3)

Як закінчений приклад розглянемо програму пошуку елементу в двовимірному масиві чисел. Елементи масиву задані статично.

Приклад 5.4. Пошук елементу в двовимірному масиві

MASM

MODEL small

STACK 256

.data

;матриця розміром 2x5 - якщо її не ініціалізувати

;то для наочності вона може бути описана так:

;array dw 2 DUP (5 DUP (?))

;але ми її ініціалізували: array dw 1,3,5,7,6,4,2,8,9,0

;логічно це виглядатиме так:

;array= {1 3}

; {5 7}

; {6 4}

; {2 8}

; {9 0}

elem dw 3 ;елемент для пошуку

failed db 0ah,0dh,'Немає такого елементу в масиві!','$'

success db 0ah,0dh,'Такий елемент в масиві присутній ','$'

found\_amount db ? ;кількість знайдених елементів

fnd db ' раз(а) ',0ah,0dh,'$'

.code

main:

mov ax,@data

mov ds,ax

xor ax, ax

mov si,0 ;si=стовпці у матриці

mov bx,0 ;bx=рядки у матриці

mov cx,5 ;число для зовнішнього циклу (по рядках)

external: ;зовнішній цикл по рядках

mov ax,array[bx][si] ;в ax перший елемент матриці

push cx ;збереження у стеку лічильника

;зовнішнього циклу

mov cx,2 ;число для внутрішнього циклу

;(по стовпцях)

mov si,0

internal: ;внутрішній цикл по рядках

inc si ;переміщення на наступний елемент в рядку

;порівнюємо вміст поточного елементу в ax з шуканим елементом:

cmp ax,elem

;якщо поточний співпав з шуканим, то перехід на here для обробки

;інакше цикл продовження пошуку

je here

;інакше - цикл по рядку cx=2 разів

loop internal

here:

jcxz move\_next ;продивились рядок ?

inc found\_amount ;інакше збільшуємо лічильник

;тих, що співпали

move\_next: ;переміщення у матриці

pop cx ;відновлюємо CX із стека (5)

add bx,1 ;пересуваємось на наступний рядок

loop external ;цикл (зовнішній)

cmp found\_amount,0h ;порівняння числа тих, що

;співпали з 0

ja eql ;якщо більше 0, то перехід

not\_equal: ;немає елементів, що співпали з шуканим

mov ah,09h ;вивід повідомлення на екран

mov dx,offset failed

int 21h

jmp exit ;на вихід

eql: ;є елементи, що співпали з шуканим

mov ah,09h ;вивід повідомлень на екран

mov dx,offset success

int 21h

mov ah,02h

mov dl,found\_amount

add dl,30h

int 21h

mov ah,09h

mov dx,offset fnd

int 21h

exit: ;вихід

mov ax,4c00h ;стандартне завершення програми

int 21h

end main ;кінець програми

При аналізі роботи програми не забувайте, що в мові асемблера прийнято елементи масиву нумерувати з 0. При пошуку певного елементу масив є видимим від початку до кінця. Програма зберігає в полі foundtime кількість входжень шуканого елементу в масив. В якості індексних регістрів використовуються si і bx.

Типові операції з масивами

Для демонстрації основних прийомів роботи з масивами краще всього підходять програми пошуку і сортування. Розглянемо одну таку програму, що виконує сортування масиву за збільшенням.

Приклад 5.5. Сортування масиву

MASM

MODEL small

STACK 256

.data

message1 db 0ah,0dh,'Початковий масив - $',0ah,0dh

message2 db 0ah,0dh,'Відсортований масив - $',0ah,0dh

n equ 8 ;кількість елементів в масиві, рахуючи з 0

array dw 2,7,4,0,1,3,5,8 ;початковий масив

tmp dw 0 ;змінні для роботи з масивом

i dw 0

j dw 0

.code

main:

mov ax,@data

mov ds,ax

xor ax, ax ;вивід на екран початкового масиву

mov ah,09h

lea dx,message1

int 21h ;вивід повідомлення message1

mov cx,10

mov si,0

show\_primary: ;вивід значення елементів

;початкового масиву на екран

Mov dx,array[si]

Add dl,30h

Mov ah,02h

Int 21h

Add si,2

Loop show\_primary

;рядки 40-85 програми еквівалентні наступному коду на мові С:

;for (i=0;i<8;i++)

;for (j=8;j>i;j–)

;if (array[i]>array[j])

;tmp=array[i];

;array[i]=array[j];

;array[j]=tmp;}

Mov i,0 ;ініціалізація i

;внутрішній цикл по j

internal:

mov j,8 ;ініціалізація j

jmp cycl\_j ;перехід на тіло циклу

exchange:

mov bx,I ;bx=i

shl bx,1

mov ax,array[bx] ;ax=array[i]

mov bx,j ;bx=j

shl bx,1

cmp ax,array[bx] ;array[i] ? array[j] - порівняння елементів

jle lesser ;якщо array[i] менше, то обмін не потрібний і

;перехід на просування далі по масиву

;інакше tmp=array[i], array[i]=array[j],

;array[j]=tmp: tmp=array[i]

Mov bx,I ;bx=i

Shl bx,1 ;множимо на 2, оскільки елементи - слова

Mov tmp,ax ;tmp=array[i]

;array[i]=array[j]

Mov bx,j ;bx=j

Shl bx,1 ;множимо на 2, оскільки елементи - слова

Mov ax,array[bx] ;ax=array[j]

Mov bx,i ;bx=i

Shl bx,1 ;множимо на 2, оскільки елементи - слова

Mov array[bx],ax ;array[i]=array[j]

;array[j]=tmp

Mov bx,j ;bx=j

Shl bx,1 ;множимо на 2, оскільки елементи - слова

Mov ax,tmp ;ax=tmp

Mov array[bx],ax ;array[j]=tmp

lesser: ;просування далі по масиву у внутрішньому циклі

dec j ;j--

cycl\_j: ;тіло циклу по j

mov ax,j ;ax=j

cmp ax,I ; порівняти j ? i

jg exchange

;якщо j>i, то перехід на обмін

;інакше на зовнішній цикл по i

Inc i ;i++

Cmp i,n ;порівняти i ? n - пройшли до кінця масиву

jl internal

;якщо i<n – продовження обробки

;виведення відсортованого масиву

Mov ah,09h

Lea dx,message2

Int 21h

prepare:

mov cx,10

mov si,0

show: ;вивід значення елементу на екран

mov dx,array[si]

add dl,30h

mov ah,02h

int 21h

add si,2

loop show

exit:

mov ax,4c00h ;стандартний вихід

int 21h

end main ;кінець програми

У основі програми лежить алгоритм, схожий на метод бульбашкового сортування. Ця програма не претендує на безумовну оптимальність, оскільки існує ціла теорія, що стосується подібного типу сортувань. Перед нами інша мета — показати використання засобів асемблера для вирішення подібного роду завдань. У програмі два цикли. Зовнішній цикл визначає позицію в масиві чергового елементу, з яким проводиться попарне порівняння елементів правої частини масиву (щодо цього елементу). За кожну ітерацію зовнішнього циклу на місці цього чергового елементу з’являється менший елемент з правої частини масиву (якщо він є). У останньому програма достатньо

Підпрограми

Виконання підпрограм.

Основні моменти виконання підпрограми ілюструються на рис.6.1. Для виклику підпрограми у програмі виконується інструкція CALL, що заносить адресу наступної інструкції в стек і завантажує в регістр IP адреса відповідної підпрограми, здійснюючи в такий спосіб перехід на підпрограму. Після цього підпрограма виконується, як будь-який інший код. У підпрограмах можуть (часто це так і буває) утримуватися інструкції викликів інших підпрограм. Фактично, належним чином побудовані підпрограми можуть навіть викликати самі себе (це називається рекурсією).

Коли підпрограма закінчує роботу, вона викликає інструкцію RET, що витягає зі стека адресу, занесену туди відповідною інструкцією CALL, і заносить його в IP. Це приводить до того, що викликаюча програма відновить виконання з інструкції, що випливає за інструкції CALL.

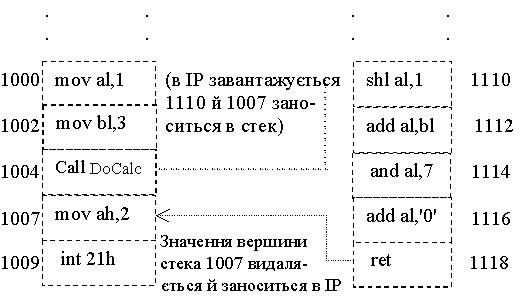
****

Рис. 6.1 – Виконання підпрограми.

Наприклад, програма виводить на екран три рядки:

Привіт

Приклад рядка

Ще один рядок

Для виводу рядків викликається підпрограма PrintString:

DOSSEG

.MODEL SMALL

.STACK 200h

.DATA

Message1 DB 'Привіт',0dh,0ah,0

Message2 DB 'Приклад рядка',0dh,0ah,0

Message3 DB 'Ще один рядок',0dh,0ah,0

.CODE

ProgramStart PROC NEAR

mov ax,@Data

mov ds,ax

mov bx,OFFSET Message1

call PrintString ; вивести рядок "Привіт"

mov bx,OFFSET Message2

call PrintString ; вивести рядок "Приклад рядка"

mov bx,OFFSET Message3

call PrintString ; вивести рядок "Ще один рядок"

mov ax,4ch ; функція DOS завершення програми

int 21h ; завершити програму

ProgramStart ENDP

;

; Підпрограма виводу на екран рядка, що завершується нульовим символом

;

; Вхідні дані:

; DS:BX - покажчик на виведений рядок.

;

PrintString PROC NEAR

PrintStringLoop:

mov al,[bx] ; одержати наступний символ рядка

and al,30 ; значення символу дорівнює 0?

jz EndPrintString ;якщо це так, то виведення рядка завершений

inc bx ; посилання на наступний символ

mov ah,2 ; функція DOS виводу символу

int 21h ; викликати DOS для виводу символу

jmp PrintStringLoop ;вивести наступний символ, якщо він є

EndPrintString:

ret ; повернення в програму

PrintString ENDP

END ProgramStart

Тут варто відзначити два моменти. По-перше, підпрограма PrintString не настроєна на виведення певного рядка. Вона може виводити будь-який рядок, на який за допомогою BX вкаже програма. По-друге, для виділення підпрограми використані дві нових директиви - PROC та ENDP. Директива PROC використовується для того, щоб відзначити початок процедури. Мітка, зазначена в директиві PROC (у цьому випадку - PrintString), являє собою ім'я процедури, так якби виконалось*:*

PrintString LABEL PROC.

Однак директива PROC робить більше. Вона визначає, яку інструкцію RET (повернення керування) - ближню або далеку - варто використати в даній процедурі. Давайте розглянемо останній оператор трохи докладніше. Якщо інструкція CALL посилається на далеку мітку, то завантажуються і CS, і IP (як і при переході). От чому при дальньому виклику в стек заносяться регістри і CS, і IP.

Директива ENDP використовується для того, щоб позначити кінець підпрограми, початої за допомогою директиви PROC. Директива ENDP відзначає кінець підпрограми, що починається з директиви PROC з тією же міткою. Наприклад, директиви:

.

TestSub PROC NEAR

. .

TestSub ENDP

.

відзначають початок і кінець підпрограми TestSub.

Директиви ENDP та PROC не генерують виконуваного коду, адже це директиви, а не інструкції. Вся їхня дія полягає у керуванні типом інструкції RET даної підпрограми. Якщо операндом директиви PROC є NEAR (ближній), то всі інструкції RET між директивою PROC і відповідною директивою ENDP асемблюються як повернення керування ближнього типу. Якщо ж, з іншого боку, операндом директиви PROC є FAR (дальній), то всі інструкції RET у даній процедурі асемблюються як повернення керування дальнього типу. Тому, щоб, наприклад, змінити тип всіх інструкцій RET в TestSub, зміните директиву PROC у такий спосіб: TestSub PROC FAR.

Приклад 5.6. Створення простої підпрограми з директивою proc

.model small

.stack 100h

.data

example\_string DB 'Hello proc subprogram!',10,13,'$'

.code

;Назва підпрогами

subprogram proc

mov dx, offset example\_string

mov ah, 09h

int 21h

;Повернення з підпрограми

ret

;Завершення опису підпрограми

endp

start:

mov ax,@data

mov ds,ax

;Виклик підпрограми

call subprogram

mov ax,4c00h

int 21h

end start

Передача параметрів.

Із програм, що викликають підпрограми, останнім часто передається інформація. Наприклад, у прикладі програми попереднього розділу для передачі в підпрограму PrintString використовувався регістр BX. Ця дія називається передачею параметрів. При цьому параметри вказують подпрограмі, що потрібно зробити. Існує два загальноприйнятих способи передачі параметрів: у регістрах та у стеку. Передача параметрів через регістри часто використовується в чистому коді Асемблера, а передача через стек використовується в більшості мов високого рівня, включаючи Паскаль і С, і в підпрограмах на Асемблері, викликуваних із цих мов. Передача параметрів у регістрах дуже проста. Для цього потрібно просто помістити значення-параметри у відповідні регістри та викликати підпрограму. Кожна підпрограма може мати свої власні потреби в параметрах, хоча ви імовірно зрозумієте, що щоб уникнути плутанини, краще виробити деякі угоди та дотримуватися їх. Наприклад, ви можете додержуватися правила, відповідно до якого перший параметр-покажчик завжди передається в регістрі BX, другий - в SI і т.д. Якщо ви використаєте для передачі параметрів регістри, акуратно коментуйте кожну підпрограму - які параметри вона одержує та у яких регістрах вони повинні перебувати. Передача параметрів у стеку більш складна, ніж передача їх через регістри. Якщо ви вирішили використати передачу параметрів через стек, ви, імовірно, будете використовувати угоди, прийнятої вами мови високого рівня. Це дозволить легко компонувати підпрограми на Асемблері із програмами, написаними даною мовою.

Значення, які повертають підпрограми.

Підпрограми часто повертають значення в викликаючу програму. У програмах на асемблері, які передбачається викликати із програми мовою високого рівня, для повернення значень ви повинні додержуватися угод даної мови. Наприклад, функції, що викликаються в мові С, повинні повертати 8- або 16-бітові значення (значення символьного, цілого типу та ближні покажчики) у регістрі AX, а 32-бітові значення (довгі цілі та далекі покажчики) - у парі регістрів DX:AX. У програмах, де використовується тільки мова асемблера, у відношенні що повертають значень допускається повна воля: ви можете поміщати їх у той регістр, який захочете. Фактично, у регістрі прапорів підпрограми можуть навіть повертати інформацію про стан (у вигляді установки прапора переносу або прапора нуля). Однак, краще встановити деякі угоди і їх дотримуватися. Корисною угодою може служити повернення 8-бітових значень у регістрі AL та 16-бітових значень у регістрі AX. Основна проблема при використанні в асемблері повертаємих підпрограмами значень полягає в тому, що при поверненні інформації підпрограми можуть зруйнувати важливу для викликаючої програми інформацію. В асемблері легше писати звернення до підпрограми, не пам'ятаючи про те, що підпрограма повертає значення, скажемо в SI (або що дана підпрограма просто змінює SI), але при цьому ви одержите програму, у якій важко буде виявляти помилки. Із цієї причини краще зводити до мінімуму число значень, що повертаються у регістри (найкраще до одного) і повертати додаткові значення, зберігаючи їх у комірках пам'яті, на які посилаються передані покажчики (як це робиться в Паскалі та С).

Збереження регістрів.

Відповідне збереження регістрів при звертанні до підпрограми - це істотний момент у програмуванні на асемблері. У сучасних мовах високого рівня підпрограма звичайно не може змінити значення змінні викликаючої програми, якщо викликаюча програма цього явно не допускає. В асемблері це не так: змінні викликаючої програми зберігаються часто в тих же регістрах, що й регістри, використовувані підпрограмою. Наприклад, якщо підпрограма змінює регістр, значення якого викликаюча програма встановила перед викликом підпрограми, але який вона використає після звертання до неї, ви одержите помилку. Одне з рішень цієї проблеми полягає в тому, щоб при вході в кожну підпрограму заносить у стек всі використовувані нею регістри та відновлювати їх зі стека перед поверненням у викликаючу програму. На жаль, для цього потрібне істотний час і великий обсяг коду. Інша можливість полягає в тім, щоб ввести правило, що викликаюча програма ніколи не розраховує на збереження регістрів підпрограмою та всі функції по їхньому збереженню виконує сама. Але це не привабливо, оскільки істотним доводом на користь застосування асемблера є воля ефективного використання регістрів. Якщо говорити коротко, то в мові асемблера існує конфлікт між швидкістю та простотою програмування. Якщо ви збираєтеся використати асемблер, ви зможете писати швидкі та компактні програми, а це означає, що потрібно розумно ставитися до збереження регістрів і забезпечити, щоб при виклику кожної підпрограми через регістри не виникало конфліктів. Найкращим підходом є акуратне коментування кожної підпрограми і вказівка, які регістри вона використає, і звертання до даних коментарями при кожному виклику підпрограми. Потрібно приділяти увагу як відстеженню збереження регістрів, так і максимально ефективному їхньому використанню. У програмуванні на асемблері це однаково важливо. Мови високого рівня виконують за вас цю роботу, але вони, як ми вже згадували, не дозволяють одержувати такі швидкі та компактні програми, які можна одержати з допомогою асемблера.

Опис команд

CALL (виклик процедури).

Інструкція CALL активізує зазначену процедуру, зберігаючи в стеці адресу наступної за CALL інструкції. Асемблер генерує інструкцію CALL різних типів залежно від того, з яким описувачем оголошене процедура-процедури-імені-процедури: NEAR або FAR. Для коректного повернення із процедури тип інструкції CALL повинен відповідати типу інструкції RET, виконуючий це повернення. Потенційна можливість невідповідності має місце, коли інструкція CALL і сама процедура перебувають у роздільно трансльованих частинах програми. Звичайно асемблер сам стежить за відповідністю викликів і повернень. Різні форми CALL дозволяють одержувати адресу процедури із самої інструкції (пряма CALL) або з області пам'яті або регістра, на які посилається CALL (непряма CALL). Варто пам'ятати, що процесор автоматично встановлює регістр IP указує на наступну інструкцію до збереження його в стеці. Для внутрісегментній прямій CALL вміст регістра SP зменшується на 2, і в стек записується вміст регістра IP. Потім до вмісту IP додається відносне зміщення адреси процедури, що змінюється в межах від -32До до +32ДО. Ця форма CALL є самовідносною та динамічно переміщуваною (CALL і процедура перебувають в одному сегменті та переміщаються разом) і тому може використовуватися в адресно незалежних програмах. Внутрісегментна непряма CALL може функціонувати через пам'ять або через регістр. Вміст SP зменшується на 2, і в стек міститься вміст IP. Зміщення процедури витягається зі специфікованих в інструкції слова пам'яті або 16-бітового регістра та заноситься в IP. Для міжсегментно прямої CALL вміст SP зменшується на 2, і в стек записується вміст регістра CS. В CS заноситься отримане з інструкції слово сегмента. SP знову зменшується на 2, і в стек записується вміст IP, а в сам IP заноситься слово зміщення з інструкції. У випадку міжсегментно непрямого CALL, що може функціонувати тільки через пам'ять, вміст SP зменшується на 2, у стек заноситься регістр CS, у який потім копіюється 2-е слово з адресуємого інструкцією 4-байтного покажчика. SP знову зменшується на 2, і в стек записується вміст регістра IP, куди потім заноситься 1-і слово покажчика.

Приклад 5.7.: Створення безпечної підпрограми

.model small

.stack 100h

.data

new\_line DB 10,13,'$'

.code

;Оголошення підпрограми toString що перетворює число в рядок

toString:

lea si, new\_line

symobol:

mov dx, 0

div bx ;Ділимо число на базу

add dx, 48 ;Додаємо до остачі код символа 0

dec si ;Декрементуємо лічильник джерела

mov [si], dl ;Поміщаємо отриманий символ в рядок

cmp ax, 0 ;Якщо резульатат ділення дорівнює 0

jz exit ;Виходимо з циклу

jmp symobol ;Інакше повторюємо кроки для частки

exit:

;Записуємо адресу початку отриманого рядку в dx

mov dx, si

;Вихід з підпрограми

ret

;Підпрограма safe\_toString

safe\_toString:

;Завантаження в стек регістрів що можуть бути змінені в ході підпрограми

push bp

push bx

push ax

push si

call toString

;Повернення значень регістрів до почакових

pop si

pop ax

pop bx

pop bp

ret

print:

push ax

mov ah, 9

int 21h

pop ax

ret

start:

mov ax,@data

mov ds,ax

;Завантажуємо аргуметри в відповідні регістри

mov ax, 512

mov bx, 10

;Виклик звичайної toString

call toString

call print

;Повторний виклик toString. AX було змінено попереднім викликом,

;тому буде виведено 0 а не 512

call toString

call print

;Повторно завантажуємо аргуметри в відповідні регістри

mov ax, 512

mov bx, 10

;Виклик безпечної toString

call safe\_toString

call print

;Повторний виклик safe\_toString. Отриманий результат тепер 512

call safe\_toString

call print

mov ax,4c00h

int 21h

end start

RET число (повернення з процедури).

Інструкція RET передає керування із процедури на інструкцію, безпосередньо наступну за інструкцією, що викликала процедуру, CALL. Асемблер генерує внутрісегментну RET, якщо програміст визначив NEAR-процедуру, і міжсегментну RET у випадку FAR-процедури. Інструкція RET витягає з адресуємого регістром SP вершини стека слово, поміщаючи його в регістр IP, і збільшує вміст SP на 2. У випадку між сегментну RET з нової вершини стека витягається та міститься в регістр CS ще одне слово, а вміст SP збільшується ще на 2. Якщо в інструкції RET зазначене число (воно може бути відсутнім), це значення додається в SP. Ця можливість може бути використана для видалення поміщених у стек параметрів процедури.

Переривання.

Інструкції цієї групи дозволяють використовувати в програмах сервісні функції, забезпечення через апарат переривань операційної системи. Із цією метою генерується програмне переривання, ефект якого подібний до ефекту апаратних переривань. Однак, процесор не виконує цикл передачі підтвердження переривання, якщо переривання ініційоване програмою або пов'язане з NMI. Інструкції переривань кожна по-своєму впливають на стани прапорів процесора.

INTO.

Переривання, якщо є переповнення. По інструкції INTO генерується програмне переривання, якщо встановлено в 1 прапор OF; у противному випадку керування передається наступній інструкції без активізації процедури обробки переривання. Інструкція INTO звертається до необхідної процедури (номер переривання - 4) через покажчик переривання, розташований за 16-ковою адресою 10, встановлює в 0 прапори TF та IF та в іншому працює також, як INT. Інструкція INTO може використатися після арифметичних або логічних операцій для обробки можливих переповнень.

INT3.

Точка виходу. Виконання цієї інструкції еквівалентно перериванню з номером 3 ("досягнута точка виходу"). Асемблер генерує коротку, 1-байтную машинну інструкцію.

IRET.

Повернення з переривання. По інструкції IRET керування повертається в точку переривання шляхом відновлення зі стека вмісту регістрів IP та CS і значень прапорів, поміщених туди при виникненні переривання. Ця інструкція використовується для виходу як із програмних, так і з апаратних переривань.

**Контрольні запитання**

1. Назвіть складні типи даних, котрі підтримує TASM.
2. Дайте формальне визначення масиву.
3. Які способи визначення масивів ви знаєте?
4. Яким чином у мові асемблера використовуються індекси масивів?
5. Назвіть режими адресації для формування ефективної адреси елементів масиву.
6. Які регістри можуть використовуватися в якості базового і в якості індексного регістрів?
7. У який спосіб мікропроцесор дозволяє масштабувати індекс?

**Завдання до лабораторної роботи №5**

**Перший рівень**

1.1.Набрати **програму 5.1**.

1.2.Завантажити програму в відладчик

1.3. Виконати програмупо кроках.

Програма 5.1.

MASM

MODEL small ;модель пам'яті

STACK 256 ;розмір стеку

.data ;початок сегменту даних

N=6 ;кількість елементів масиву

array db 6 dup (4 dup (0))

.code ;сегмент коду

main: ;точка входу в програму

mov ax,@data

mov ds,ax

xor ax, ax ;обнулення ax

mov si,0 ;0 у si

mov cx,N ;N у cx

go:

mov dl,array[si] ;перший байт поля в dl

inc dl ;збільшення dl на 1 (по умові)

mov array[si],dl ;заслати назад в масив

add si,4 ;зсув на наступний елемент масиву

loop go ;повтор циклу

mov si,0 ;підготовка до виводу на екран

mov cx,N

show: ;вивід на екран вмісту перших байтів полів

mov dl,array[si]

add dl,30h

mov ah,02h

int 21h

loop show

exit:

mov ax,4c00h ;стандартный вихід

int 21h

end main ;кінець програми

**Другий рівень**

1. Реалізувати **завдання 1** мовою асемблер.
2. Забезпечити обробку даних масивів, у вигляді окремої процедури.
3. Застосувати оператори CALL, RET

**Третій рівень**

1. Реалізувати **завдання 1 та 2** мовою асемблер
2. Забезпечити обробку даних масивів, введення даних з клавіатури, та виведення результатів у вигляді окремих процедур.
3. В процедурах продемонтрувати використання операндів NEAR та FAR.
4. Застосувати оператори CALL, RET.

**Варіанти ЗАВДАННЯ1:**

* 1. З елементів цілочисельного вектора С(п), п<12, кратних заданому цілому числу Т, сформувати вектор В. Виконати зсув отриманого масиву вправо на одну позицію.
  2. Задано вектор Х(п), п<12. Сформувати вектор В з Хі, що перевищують величину М = (minх + maxх)/2, де minх і maxх - найменше й найбільше значення елементів Хі.
  3. Задано цілий вектор А(п), п<14. Сформувати вектор Xі тих елементів Аі, значення яких є прості числа.
  4. Задано цілий вектор Х(п), п<16. Сформувати масив К з тих елементів вектораX, які є ступенем числа 2.
  5. Задано вектор А(n), п < 12. Сформувати вектор X з елементів вектора А, значення яких не перевищують середнє арифметичне масиву А. Виконати зсув отриманого масиву вправо на дві позиції.
  6. Задано цілий масив А(п), п<12. Сформувати вектор М з елементів А[і], значення яких кратні 5 і не кратні 7. Виконати зсув отриманого масиву вліво на одну позицію.
  7. Задано цілий вектор М(п), п<12. Сформувати вектор А, елементи якого А [і] дорівнюють кількості цифр у десятковому записі елемента М[і].
  8. Задано вектор М(п), п < 13. Сформувати вектор С із елементів М[і], значення яких належать інтервалу [А, В]. Виконати зсув отриманого масиву вправо на одну позицію.
  9. Задано вектор М(п), п<9. Сформувати вектор С із елементів М[і], значення яких не рівні А і В. Знайти кількість нулів у новому масиві. Виконати зсув отриманого масиву вліво на одну позицію.
  10. Якщо сума додатних елементів вектора С(n), п < 12, дорівнює числу В, то переслати додатні елементи С[і] у новий масив X, у іншому випадку переслати в масив X елементи, менші числа В.
  11. Задано вектор Т(n), п < 12. Сформувати вектор X з елементів Т[і], що передують найбільшому елементу вектора Т. Виконати зсув отриманого масиву вправо на одну позицію.
  12. Задано цілий вектор Т(n), *п<*12. Сформувати вектор *X* з тих елементів масиву Т, сума цифр яких кратна 3. Виконати зсув отриманого масиву вправо на одну позицію.
  13. Задано масив А(n), *п <* 15, цілого типу. Сформувати вектор *М* з елементів А[і], які більше своїх сусідів*1.* Виконати зсув отриманого масиву вліво на одну позицію.
  14. Задано масив А(n), *п* < 12, цілого типу. Одержати вектор *М,* елементи якого дорівнюють непарним елементам масиву А. Знайти середнє арифметичне елементів нового масиву.
  15. Задано масив А(n), *п* < 16, цілого типу. Одержати вектор *М,* розташувавши в ньому спочатку всі додатні елементи масиву А, потім - всі від'ємні.

**Варіанти ЗАВДАННЯ2:**

1. Задано матрицю A(m, n), 1<m, n<10. У кожному стовпці знайти номер і значення максимального елемента. Переставити рядки матриці за зростанням суми додатних елементів у рядках.
2. Для матриці А(m, n), 1<m, n<10, підрахувати кількість елементів, рівних нулю. Переставити рядки за спаданням значень максимальних елементів рядків.
3. Для матриці В(m, n), 1<m, n<10, знайти суму всіх елементів. Переставити стовпці за зростанням кількості нулів у стовпцях.
4. Для матриці С(m, n), 1<m, n<10, знайти суму елементів, більших 3,2. Переставити стовпці за зростанням кількості від'ємних елементів у стовпцях.
5. Для матриці Р(m, n), 1<m, n<10, надрукувати номери стовпців з кількістю нулів більше 1. Переставити рядка за зростанням значень мінімальних елементів рядків.
6. Для матриці O(m, n), 1<m, n<10, надрукувати індекси ма­ксимального елемента масиву і його значення. Переставити рядки за спаданням значень сум елементів рядків.
7. Для матриці С(m, n), 1 < m, n < 10, знайти значення мінімального елемента. Переставити стовпці за спаданням кількості додатних елементів у стовпцях.
8. Для матриці К(m, n), 1 < m, n< 10, знайти в кожному стовпці номер першого від'ємного елемента. Переставити рядка за спаданням значень перших елементів рядків.
9. У матриці Т(m, n), 1<m, n <10, знайти суму абсолютних значень всіх елементів масиву. Переставити рядка за спаданням зна­чень останніх елементів рядків.
10. Для матриці A(m, n), 1 < m, n <10, визначити рядок з найбільшою кількістю від'ємних елементів. Переставити стовпці за зростанням суми додатних елементів стовпців.
11. Для матриці A(m, n), 1< m, n <10, надрукувати номери рядків, у яких є хоча б один нуль. Переставити рядки за спаданням значень максимальних елементів рядків.
12. Для матриці К(m, n), 1 < m, n<10, знайти кількість рядків, де є хоча б один нуль. Переставити рядки за зростанням значень максимальних елементів рядків.
13. Для матриці K(m, n), 1<m,n<10, надрукувати номери стовпців, що містять більше двох нулів. Переставити рядки за спадан­ням значень суми елементів рядків.