ЛАБОРАТОРНА РОБОТА №2

Робота з директивами визначення даних та командами переміщення даних.

|  |
| --- |
|  |

***Мета:*** використання директив визначення даних при розробці програм на Ассемблері; використання команд переміщення даних при розробці про-грам на Ассемблері.

План:

1. Дерективи визначення даних, DB, DW, DD.
2. Операнд ?. Оператори [], OFFSET.
3. Адресний вираз.
4. Команди пересилки.
5. Оператор PTR.

Короткі теоретичні відомості

Директиви визначення даних

Для опису змінних, з якими працює програма, використовуються директиви визначення даних.

Директива DB

По директиві DB (define byte, визначити байт) визначаються дані розміром у 1 байт. Її синтаксис такий:

**[<імя>] DB <операнд> {,<операнд>}**

Зустрічаючи таку директиву, ассемблер обчислює операнди і записує їх значення в послідовні байти пам’яті. Першому з цих байтів присвоюється зазначене ім'я, по якому на цей байт можна посилатися з інших місць програми.

Існує два основних способи задання операндів директиви DB:

• ? (знак невизначеного значення),

• константний вираз із значенням від -128 до 255.

Інші способи задання операндів похідні відцих двох.

Операнд ?

Х DB ?

За цією директивою описується змінна X. Для неї виділяється один байт пам'яті, у який нічого не записується. У цьому випадку говорять, що змінна не отримує початкового значення.

Зустрічаючи директиву DB, транслятор відводить під зазначену змінну перший із ще не зайнятих байтів пам'яті. Це варто враховувати і, наприклад, не треба вставляти директиву DB між командами.

Виділивши байт під змінну, ассемблер запам'ятовує його адресу. Коли він знову зустріне в тексті програми ім'я цієї змінної, то він замінить його на дану адресу (у цій заміні і полягає трансляція імен).

Адресу комірки, виділеної змінній з ім'ям X, прийнято називати значенням імені Х (не плутати з вмістом комірки по цій адресі!). Крім того, за описом змінної ассемблер запам'ятовує, скільки байт займає змінна в пам'яті. Цей розмір називається типом імені змінної. Значення (адреса) і тип (розмір) імені змінної однозначно визначають комірку, що позначається цим ім'ям. Нагадаємо, що з тієї ж самої адреси в ПК можуть починатися комірки різних розмірів - і байт, і слово, і подвійне слово, тому крім початкової адреси комірки треба знати і її розмір. У зв'язку з цим ассемблер запам'ятовує як адресу змінної, так і її розмір.

Оператор типу: type <ім'я>

Значенням цього оператора є розмір (у байтах) комірки, виділеної під змінну з зазначеним ім'ям. Якщо змінна описана по директиві DB (байтова змінна), то для її імені значення цього оператора дорівнює 1.

Відзначимо, що є стандартна константа з ім'ям BYTE і значенням 1, тому можна записати так:

TYPE Х = BYTE = 1

**Операнд - константний вираз із значенням від** *-128 до 255*

Ассемблердозволяє описувати і змінні з початковими значеннями. Для цього необхідно в якості операнда директиви DB вказати вираз, який ассемблер обрахує і значення якого запише в комірку, відведену під змінну. Це і є початкове значення змінної. Пізніше, при виконанні програми, його можна буде змінити, можна буде щось записати в цю комірку, але до початку виконання програми в цій комірці вже буде знаходитися дане значення.

У найпростішому і найбільш розповсюдженому випадку початкове значення байтової змінної задається у вигляді числа з величиною від -128 до 255. Наприклад:

А DB 254 ; OFEh

В DB -2 ; OFEh (=256-2=254)

С DB 17h ; 17h

По кожній з цих директив ассемблер відводить один байт під змінну і записує в цей байт зазначене число. Таким чином, до початку виконання програми змінна А буде мати значення 254, змінна В - значення -2, а змінна С - значення 17h.

Операнд-число переводиться ассемблером у двійкову систему. При цьому невід’ємні числа записуються в пам'ятьяк числа без знака, а від’ємні числа записуються в додатковому коді У зв'язку з цим в якості операндів можна вказувати числа від -128 до 255, тобто числа 254 і -2 будуть представлені в пам'яті тим самим байтом OFEh (це для нас дані числа різні, а для машини вони однакові, і їй байдуже, що позначає байт OFEh - число з чи без знака).

В іншому випадку як початкове значення перемінної вказується символ. Таке значення можна задати подвійно: або вказати числовий код цього символу, або вказати сам символ у лапках. Наприклад, у системі кодування ASCII код символу "\*" дорівнює 2Ah, тому випливає, що дві директиви еквівалентні:

Q DB 2Ah Q DB "\*"

В другому випадку ассемблер сам визначить код зазначеного символу і запише цей код у комірку пам'яті. Ясно, що цей варіант кращий - він наочніший і не вимагає знання кодів символів, тому його звичайно і використовують на практиці.

Ми розглянули два основних випадки задання початкового значення. У загальному ж випадку таке значення вказується будь-яким константним виразом зі значенням від -128 до 255. Якщо значення виходить за ці межі, то ассемблер зафіксує помилку.

Директива з декількома операндами

Ми розглянули випадки, коли в директиві DB вказується один операнд. Це зручно, коли треба описати скалярну змінну, але незручно, коли треба описати змінну-масив. Справді, якщо треба, приміром, описати масив з 4 байтів з деякими початковими значеннями, то це можна зробити так:

М db 2

DB -2

DB 7

DB '\*'

В масивах ім'я дається тільки його першому елементу, тому в прикладі ім'я зазначене лише в першій директиві. Якщо в директиві DB не зазначене ім'я, то байт у пам'яті виділяється, але він залишається безіменним.

Якщо в масиві багато елементів, то такий спосіб опису масиву громіздкий. Тому допускається спрощена форма опису масивів, коли він описується однією директивою, але з декількома операндами (кількість елементів в масиві). Наприклад, замість 4 директив можна виписати тільки одну:

М DB 2,-2,?,'\*'

По директиві DB з декількома операндами ассемблер виділяє в пам'яті сусідні байти пам'яті, по одному на кожен операнд, і записує в ці байти значення операндів (для операнда ? нічого не записує).

Операнд - рядок

Якщо в директиві DB декілька сусідніх операндів - символи, то їх можна об'єднати в один рядок. Наприклад, наступні дві директиви еквівалентні:

S DB 'a','b','c' *S* DB 'abc'

Відзначимо, що й у цьому випадку тип імені дорівнює 1 (TYPE S = BYTE), тому що кожна з цих директив є скороченням наступних трьох директив:

s db 'а'

DB 'b'

DB 'c',

а тут видно, що ім'я S позначає тільки перший байт.

Операнд - конструкція повторення DUP

Досить часто в директиві доводиться вказувати однакові операнди. Наприклад, якщо ми хочемо описати байтовий масив R з 8 елементів з початковим значенням 0 для кожного з них, то це можна записати так:

R DB 0,0,0,0,0,0,0,0 або R DB 8 DUP(0)

Тут у якості операнда використана конструкція повторення, у якій спочатку вказується коефіцієнт повторення, потім - службове слово DUP (duplicate, копіювати), а за ним у круглих дужках - повторювана величина.

У загальному випадку ця конструкція має наступний вигляд:

k DUP (p1,p1,...,p1),

де k - константний вираз з значенням ≥ 1, р, - будь-якої припустимий операнд директиви DB (зокрема, це може бути знову конструкція повторення). Даний запис є скороченням для k раз повтореної послідовності зазначених у дужках операндів.

Наприклад, директиви ліворуч еквівалентні директивам праворуч:

Х DB 2 DUP ('ab',?, 1) X DB 'ab',?,1,'ab',?,1

Y DB -7, 3 DUP(0, 2 DUP(?))

Y DB -7,0,?,?,0,?,?,0,?,?.

Директива DW

Директивою DW (define word, визначити слово) описуються змінні розміром у слово. Вона аналогічна директиві DB.

Операнд ?

Можливий приклад:

A DW ?

По цій директиві ассемблер відводить під змінну А слово пам'яті, у яке нічого нe записується. Тип змінної дорівнює 2, тому що вона займає два байти. У Ассемблері є стандартна константа з ім'ям WORD і значенням 2, тому можна записати :

TYPE A = WORD = 2.

Константний вираз із значенням від -32768 до 65535

Наприклад :

В DW 1234h

С DW -2

Відповідно до цих директив під змінні В и С виділяється по слову пам'яті,а у відповідні комірки записуються зазначені числа, що стають початковими значеннями цих змінних.

Як і у випадку директиви DB, невід’ємні числа записуються в пам'ять як числа без знака, а від’ємні числа - у додатковому коді. Тому числа, що можуть бути задані як операнди директиви DW, повинні належати відрізку [-215, 216-1].

Відмінність від директиви DB полягає в тому, що в ПК числа розміром у слово зберігаються в пам'яті в "переверненому" вигляді, а "перекиданням" їх займається сам ассемблер, тому згідно двох директив у прикладі пам'ять заповниться таким чином:

34 12 FE FF

B C

Адресний вираз

У якості операнда директиви DW може бути зазначений адресний вираз, тобто вираз, значенням якого є адреса. Основний випадок адресного виразу- це імя змінної чи мітка. Наприклад:

с db ?

D DW С

У цьому випадку ассемблер записує в слово, виділене під змінну D, адресу змінної С, що стає початковим значенням змінної D.

У правій частині директиви DW можна вказати будь-яке число операндів, а також конструкцію повторення. Наприклад:

Е DW 40000, 3 DUP(?)

Директива DD

По директиві DD (define double word, визначити подвійне слово) описуються змінні, під які відводяться подвійні слова. Тому імена цих змінних мають тип 4 чи DWORD. В іншому ця директива схожа на дві попередні.

Приклад:

А DD ?

Під перемінну А виділяється подвійне слово і змінна А не отримує початкового значення.

**Ціле число із значення від** **-231 до 231-1**

Приклад:

В DD 123456h

У даному випадку змінна В отримує початкове значення, і це значення ассемблер записує в пам'ять у "переверненому" вигляді:

56 34 12 00

Команди пересилки

Команда MOV

Машинні команди пересилання байта чи слова (переслати однією командою подвійне слово не можна). Величина, що пересилається, береться з команди чи регістра комірки пам'яті, а записується в регістр чи комірку памяті.

MOV opl, op2

По команді MOV на місце першого операнда пересилається значення другого операнда. Прапори команда не змінює.

Приклади:

MOV АХ, 500 ;А:=500

MOV BL, DH ;BL:=DH

Слід враховувати, що забороняється пересилання з однієї комірки пам'яті в іншу, з одного сегментного регістра в інший, запис безпосереднього операнда в сегментний регістр. Це обумовлено тим, що немає таких машинних команд. Якщо по алгоритму все-таки потрібна така дія, то вона реалізується в двох командах, пересиланням через який-небудь несегментний регістр. Наприклад, записати число 100 у сегментний регістр DS можна так:

MOV АХ, 100

MOV DS, AX ;DS:=100

Відзначимо також, що командою MOV не можна змінювати вміст сегментного регістра CS. Це зв'язано з тим, що регістри CS і IP визначають адресу тієї команди програми, що повинна бути виконана наступною, тому зміна кожного з цих регістрів є ніщо інше, як операція переходу, а не пересилання.

Числа розміром у слово зберігаються в пам'яті в "переверненому" виді, а в регістрах - у нормальному, непереверненому. Команда MOV враховує це і при пересиланні слів між пам'яттю і регістрами сама "перевертає" їх:

Q DM 1234h ;Q: 34h, Q+1:12h

MOV AX, Q ;AH=12h, AL=34h

Командою MOV можна переслати як байт, так і слово.

Відзначимо, що при пересиланнях ніяких перетворень байта в чи слово слова в байт не виробляється.

Приклад 2.1. Завантаження регістрів даними.AL=12H BL=56H, AH=34H BH=78H.

.model small ;модель пам'яті ближнього типу

.code ;початок сегменту коду

org 100h ;задаємо стек розміром 100h

Start: ;мітка початку програми

mov ax,@data ;формування адреси сегмента даних

mov ds,ax

mov al,12h ;завантаження в регістр al даних 12h

mov ah,34h ;завантаження в регістр ah даних 34h

mov bl,56h

mov bh,78h

exit:

mov ax, 4c00h ;функція DOS завершення програми

int 21h

END Start ;кінець програми

Оператор вказування типу (PTR)

Оператор вказівки типу PTR (від pointer, покажчик), що записується так :

<тип> PTR<вираз>

де <тип> - це BYTE, WORD чи DWORD , а вираз може бути константним чи адресним.

Якщо зазначено константний вираз, то оператор вказує, що значення цього виразу (число) повинне розглядатися асемблером, як величина зазначеного типу (розміру); наприклад, BYTE PTR 0 - це нуль як байт, a WORD PTR 0 - це нуль як слово (запис BYTE PTR 300 помилковий, тому що число 300 не може бути байтом). Відзначимо, що в цьому випадку оператор PTR відноситься до константних виразів.

Якщо ж у PTR зазначено адресний вираз, то оператор вказує, що адреса, яка є значенням виразу, повинна сприйматися асемблером, як адреса комірки зазначеного типу (розміру); наприклад: WORD PTR A - адреса А позначає слово (байти з адресами А і А+1). У даному випадку оператор PTR відноситься до адресних виразів.

Оператор PTR корисний ще в ситуації, коли треба не уточнити тип операнда, а змінити його. Нехай, приміром, Z - змінна розміром у слово:

Z DM 1234h ;Z: 34h, Z+1:12h,

і треба записати нуль не в слово, а тільки в його перший бант - у той, де знаходиться величина 34h.

MOV BYTE PTR Z,0 ;Z: OOh, Z+1:12h

Тут вказано ігнорувати тип імені Z, і вважати, що ім'я Z позначає байт. Така зміна типу локальна, вона діє тільки в даній команді.

Отже, оператор PTR використовується в наступних ситуаціях: коли типи операндів команд невідомі і тому треба вказати явно тип одного з операндів, і коли не влаштовує тип, приписаний імені при його описі, і тому необхідно вказати потрібний тип.

Приклад 2.2: Використання оператора PTR

.data  
string db "1234567890:", 0  
.code  
start:  
mov eax, dword ptr [string] ;помістимо подвійне слово (4 байти)  
mov ax, word ptr [string] ;помістимо слово

;(2 байти)  
mov ah, byte ptr [string] ;помістимо байт  
end start

Команда XCHG

У машинних програмах доводиться досить часто переставляти місцями дві величини, і хоча таку перестановку можна реалізувати за допомогою команди MOV, введена спеціальна команда для цього:

Перестановка (exchange): XCHG op1,op2

Ця команда змінює місцями значення своїх операндів (вони повинні бути або байтами, або словами. Прапори при цьому не змінюються. Приклад:

MOV AX, 62 ;АХ=62

MOV SI, 135 ;SI=135

XCHG AX, SI;AX=135, SI=62

Ще допускається перестановка вмісту двох комірок пам'яті. Якщо все-таки треба зробити таку перестановку, то це реалізується через регістр. Наприклад, поміняти місцями значення байтових змінних Х и Y можна так:

MOV AL, X ;AL=X

XCHG AL, Y ;AL=Y, Y=X

MOV X, AL ;X=Y (вихідне значення)

Робота з адресами і покажчиками

Для роботи з адресами операндів, що знаходяться в пам'яті, передбачена група команд:

lea призначення, джерело — завантаження ефективної адреси;

lds призначення, джерело — завантаження покажчика в регістр сегмента даних ds;

les призначення, джерело — завантаження покажчика в регістр додаткового сегмента даних es;

lgs призначення, джерело — завантаження покажчика в регістр додаткового сегмента даних gs;

lfs призначення, джерело — завантаження покажчика в регістр додаткового сегмента даних fs;

lss призначення, джерело — завантаження покажчика в регістр сегмента стека ss.

Приклад 2.3. Отримання адреси за допомогою команди lea

.model small

.stack 100h

.data

example\_string DB 'Example string!',10,13,'$'

.code

start:

mov ax,@data

mov ds,ax

lea dx, example\_string ;завантаження в dx адреси початку рядка

mov ah, 09h ;функція виведення рядку на екран

int 21h

lea dx, example\_string[3] ;завантаження в dx адреси третього символу рядка

mov ah, 09h

int 21h

mov ax,4c00h

int 21h

end start

Перетворення даних

З усієї сукупності команд мікропроцесора безпосередньо до команд перетворення даних можна віднести одну команду:

хlat *[адреса\_ табпиці\_перекодування]*

Її дія полягає в тому, що вона заміщає значення в регістрі аl іншим байтом з таблиці в пам'яті, розташованої за адресою, зазначеною операндом:

*адреса\_таблиці\_перекодування.*

Адреса байта в рядку, яким буде заміщатися вміст регістра аl, визначається сумою (bx) + (а1), тобто вміст аl виконує роль індексу в байтовому масиві.

Приклад 2.4. Програма перетворення двозначного шістнадцяткового числа в двійкове подання з використанням команди xlat. Вхід: вихідне шістнадцяткове число; вводиться з клавіатури. Вихід: результат перетворення в регістрі al.

.model small  
.data ;сегмент даних  
message db "Введіть дві шістнадцяткові цифри, $"  
tabl db 48 dup (0), 6, 1,2,3, 4,5,6,7,8,9 7 dup (0)  
db 0ah, 0bh, 0ch, 0dh, 0eh, 0fh, 26 dup (0)  
db 0ah, 0bh, 0ch, 0dh, 0eh, 0fh, 152 dup (0)  
.stack 256 ;сегмент стека  
.code ;початок сегмента коду  
main: ; початок процедури main  
mov ax, @ data ;фізичну адресу сегмента даних в регістр ах  
mov ds, ax ;ax записуємо в ds  
lea bx, tabl ;завантаження адреси рядка байт в регістр bх  
mov ah, 9  
mov dx, offset message  
int 21h ;вивести запрошення до вводу  
хог ах, ах ;очистити регістр ах  
mov ah, 1h ;значення 1h в регістр ah  
int 21h ;вводимо першу цифру в al  
xlat ;перекодування першого введеного символу в al  
mov dl, al  
shl dl, 4 ;вліво для звільнення місця для молодшої цифри  
int 21h ;введення другого символу в al  
xlat ;перекодування другого введеного символу в al  
add al, dl ;додаємо для отримання результату  
mov ах,4c00h ;пересилка 4c00h в регістр ах  
 int 21h ;завершення програми  
end main ;кінець програми з точкою входу main

Оператор []

Індексний оператор []. Дужки - теж оператор, і транслятор їх наявність сприймає як вказівку скласти значення вираз\_1 за цими дужками із вираз\_2, укладеними в дужки. Наявність індексного оператора вказує транслятору, що необхідно отримати значення за обчисленою адресою.

Приклад 2.5.: Отримання даних, що зберігаються за адресою.

.model small

.stack 100h

.data

exit\_code DW 4c00h

.code

start:

mov ax,@data

mov ds,ax

lea bx, exit\_code ;завантажуємо в bx адресу змінної exit\_code

mov ax, [bx] ;отримання значення, що ;знаходиться за адресою, ;розміщеною в регістрі bx

int 21h

end start

Оператор OFFSET

Оператор одержання зсуву виразу **offset** дозволяє отримати значення зсуву виразу в байтах відносно початку того сегмента, в якому вираз визначено.

Директива **offset:** компілятор, знайшовши цю директиву, вставляє в зібрану програму покажчик на змінну (тобто адресу тієї змінної, ім'я якої стоїть після offset).

Приклад 2.6.: Отримання адреси за допомогою директиви offset

.model small

.stack 100h

.data

example\_string DB 'Example string!',10,13,'$'

.code

start:

mov ax,@data

mov ds,ax

mov dx, offset example\_string ;завантаження в dx адреси початку рядка

mov ah, 09h ;функція виведення рядку на екран

int 21h

mov dx, offset example\_string+3 ;завантаження в dx ;адреси рядка + 3 байти(символи)

int 21h ;вміст ax не змінювався, mov ah, ;09h можна не писати

mov ax,4c00h

int 21h

end start

Робота зі стеком

**Стек** *—* це область пам'яті, спеціально виділена для тимчасового збереження даних програми. Важливість стека визначається тим, що для нього в структурі програми передбачений окремий сегмент. Для роботи з стеком призначені три регістри:

1. ss — сегментний регістр стека;
2. sp/esp — регістр покажчика стека;
3. bp/ebp — регістр покажчика бази кадру стека.

Розмір стека залежить від режиму роботи мікропроцесора й обмежується 64 Кбайт (чи 4 Гбайт у захищеному режимі). У кожен момент часу доступний тільки один стек, адреса сегмента якого міститься в регістрі ss. Цей стек називається **поточним***.* Для того щоб звернутися до іншого стеку («переключити стек»), необхідно завантажити в регістр ss іншу адресу. Регістр ss автоматично використовується процесором для виконання всіх команд, що працюють з стеком.

Особливості роботи з стеком:

Запис і читання даних у стеку здійснюються відповідно до принципу LIFO (Last In First Out — «останнім прийшов, першим пішов»).

В міру запису даних у стек останній росте в бік молодших адрес.

При використанні регістрів esp/sp і ebp/bp для адресації пам'яті ассемблер автоматично вважає, що значення, які містяться в ньому, являють собою зміщення щодо сегментного регістра ss.

Регістри ss, esp/sp і ebp/bp використовуються комплексно, і кожний з них має своє функціональне призначення. Регістр esp/sp завжди вказує на вершину стека, тобто містить зміщення, по якому в стек був занесений останній елемент.

Якщо необхідно одержати доступ до елементів не на вершині, а усередині стека то для цього застосовують регістр ebp. Регістр ebp — регістр покажчика бази кадру

1. Для організації роботи зі стеком існують команди запису і читання.
2. push **джерело** *—* запис значення *джерела* у вершину стека.
3. Алгоритм роботи цієї команди, що включає наступні дії:
4. (sp) = (sp) - 2: значення sp зменшується на 2;
5. значення з **джерела** записується за адресою, що вказується парою ss:sp.
6. pop **призначення** *—* запис значення з вершини стека по місцю, зазначеному операндом **призначення***.* Значення при цьому «знімається» з вершини стека.
7. Алгоритм роботи команди pop зворотний алгоритму команди push:
8. запис вмісту вершини стека по місцю, зазначеному операндом **призначення***;*
9. (sp) = (sp) + 2; збільшення значення sp.
10. pusha — команда групового запису в стек. По цій команді в стек послідовно записуються регістри ах, сх, dx, bx, sp, bp, s1, d1.
11. pushaw — майже синонім команди pusha. Різниця полягає при роботі команд з pusha і pushaw з атрибутами розрядності use16 та use32
12. use16 — алгоритм роботи pushaw аналогічний алгоритму pusha;
13. use32 — pushaw не змінюється (тобто вона нечуттєва до розрядності сегмента і завжди працює з регістрами розміром у слово — ax, ex, dx, bx, sp, bp, si, dl). Команда pusha чуттєва до встановленої розрядності сегмента і при вказівці 32-розрядного сегмента працює з відповідними 32-розрядними регістрами, тобто еах, есх, edx, ebx, esp, ebp, esi, edl.
14. pushad — виконується аналогічно команді pusha

Приклад 2.7. Використання стеку за допомогою команд push та pop

.model small

.stack 100h

.data

first\_string DB 'First string!',10,13,'$'

second\_string DB 'Second string!',10,13,'$'

third\_string DB 'Third string!',10,13,'$'

.code

start:

mov ax,@data

mov ds,ax

lea ax, first\_string ;завантаження в ax адреси ;початку першого рядка

push ax ;додавання на вершину ;стеку вмісту регістру ax

lea ax, second\_string ;завантаження в ax адреси ;початку другого рядка

push ax ;додавання на вершину ;стеку вмісту регістру ax

lea ax, third\_string ;завантаження в ax адреси початку третього рядка

push ax ;додавання на вершину ;стеку вмісту регістру ax

mov ah, 09h ;функція виведення рядку ;на екран

pop dx ;вилучення вершини стеку в ;dx(адреса третього рядка)

int 21h

pop dx ;вилучення вершини стеку в dx ;(адреса другого рядка)

int 21h

pop dx ;вилучення вершини стеку в dx ;(адреса першого рядка)

int 21h

mov ax,4c00h

int 21h

end start

Наступні три команди виконують дії, зворотні вищеописаним командам:

1. рора;
2. popaw;
3. popad.

Група наступних команд дозволяє зберегти в стеку регістр прапорців і записати слово чи подвійне слово в стек. Це єдині команди, що дозволяють одержати доступ (і які мають потребу в цьому доступі) до усього вмісту регістра прапорців.

Приклад 2.8.: Використання операторів pusha і popa.

.286 ;дозвіл на виконання непривілейованих інструкцій процесора 80286

.model small

.stack 100h

.data

example\_string DB 'Example string!',10,13,'$'

.code

start:

mov ax,@data

mov ds,ax

lea dx, example\_string ;завантажуємо адресу ;рядкa в dx

mov ah, 09h ;завантажуємо номер

;ф-ії виведення рядку в al

;Розміщення в стеку регістрів загального призначення в наступній послідовності: ax, cx, dx, bx, sp, bp, si, di.

pusha

;Змінимо вміст певних регістрів загального призначення

mov dx, 0h

mov ax, 0h

;Вилучення зі стеку регістрів загального призначення

;di, si, bp, sp, bx, dx, cx, ax.

popa

;Значення повернулися до попередніх, отже рядок може бути виведений

int 21h

mov ax,4c00h

int 21h

end start

1. pushf — зберігає регістр прапорів у стек. Робота цієї команди залежить від атрибута розміру сегмента:
2. usel6 — у стек записується регістр flags розміром 2 байти;
3. use32 — у стек записується регістр eflags розміром 4 байти;
4. pushfw — зберігає в стек регістр прапорів розміром у слово. Завжди працює як pushf з атрибутом usel6;
5. pushfd — зберігає в стек регістр прапорів flags чи eflags, у залежності від атрибута розрядності сегмента (тобто те ж, що і pushf).
6. Аналогічно випливають три команди, що виконують дії, зворотні розглянутим вище операціям:
7. popf;
8. popfw;
9. popfd.

Основні види операцій, коли використання стека практично обов'язкове:

1. виклик підпрограм;
2. тимчасове збереження значень регістрів;
3. визначення локальних змінних.

Приклад 2.9.: Використання стеку за допомогою покажчика стеку.

.model small

.stack 100h

.data

first\_string DB 'First string!',10,13,'$'

second\_string DB 'Second string!',10,13,'$'

third\_string DB 'Third string!',10,13,'$'

.code

start:

mov ax,@data

mov ds,ax

lea ax, first\_string ;Завантаження в ax адреси

;початку першого рядка

push ax ;Додавання на вершину

;стеку вмісту регістру ax

lea ax, second\_string ;Завантаження в ax адреси ;початку другого рядка

push ax ;Додавання на вершину ;стеку вмісту регістру ax

lea ax, third\_string ;Завантаження в ax адреси ;початку третього рядка

push ax ;Додавання на вершину стеку вмісту регістру ax

;Завантаження покажчика на вершину стеку в bp

;bp спеціальний регістровий покажчик, регістр

;загального призначення не може бути використаний

;у даному випадку

mov bp, sp

mov ah, 09h

;Завантаження в dx даних, що знаходяться на дві по-;зиції нижче вершини стеку (адреса першого рядку).

mov dx, [bp+4] ;Використовується + 4 оскільки на кожний покажчик ;відведено 2 байти

int 21h

mov dx, [bp+2] ;Завантаження в dx даних, що знаходяться на одну по-;зицію нижче вершини стеку (адреса другого рядку)

int 21h

mov dx, [bp] ;Завантаження в dx даних, що знаходяться на вершині

;стеку (адреса третього рядку).

int 21h

pop dx ;Вилучаємо вершину стеку в dx (все ще адреса третього рядка)

int 21h

;Оскільки стек змінився, важливо оновити покажчик bp

;інакше можна отримати неочікувану поведінку

mov bp, sp

mov dx, [bp] ;тепер вершина стеку вказує на адресу другого рядка

int 21h

mov ax,4c00h

int 21h

end start

**Завдання до лабораторної роботи №2**

**Перший рівень**

Виконати завдання 1.

**Другий рівень**

Виконати завдання 1,2.

**Третій рівень**

Виконати завдання 1-4.

**ЗАВДАННЯ 1.**

1.1 Написати програму, що виконує наступні дії:

1) завантаження регістрів даними

AL=12H BL=56H CX=89H

AH=34H BH=78H DX=0ABCH

2) обмін даними

AX <---> BX

CX <---> DX

3) пересилання даних

AX ---> SI

BX ---> DI

CX ---> BP

1.2 Створити файл із будь-яким ім'ям, який виконує дану програму, використовуючи стандартне оформлення програми.

1.3 Одержати файл із розширенням .EXE

1.4 Одержати файл із розширенням .COM

1.5 Виконати трасування програми по кроках, продивившись на кожному кроці зміну стану регістрів.

**ЗАВДАННЯ 2.** Модифікуйте пpогpаму із **завдання 1** переоформивши програму у спрощеному вигляді (використовуючи спрощені директиви, .MODEL SMALL, .STACK 100H, .CODE):

2.1. Визначите початок сегмента даних .DATA;

2.2. Визначте дані в сегменті даних довжиною 1 байт з ім'ям SRC і значеннями 1,2,3,4,5,6,7,8; SRC DB 1,2,3,4,5,6,7,8

2.3. Заpезеpвувати в області пам'яті даних DST по 1 байту 8 порожніх байти - DST DB 8 DUP (?).

2.4. Заpезеpвувати в області пам'яті даних ABC по 1 байту 3 порожніх байти - ABC DB 3 DUP (?).

2.5. Отримайте файл із розширенням .EXE Завантажите вашу програму в відладчик. Виконаєте пpогpаму по кроках, продивляючись на кожному кроці зміну стану pегістpів.

**ЗАВДАННЯ 3** Модифікувати пpогpаму із **завдання 1** переоформивши програму у спрощеному вигляді (використовуючи спрощені директиви, .MODEL SMALL, .STACK 100H, .CODE):

2.1. Визначите початок сегмента даних .DATA;

2.2. Визначте дані в сегменті даних довжиною 1 байт з ім'ям SRC і значеннями 1,2,3,4,5,6,7,8; SRC DB 1,2,3,4,5,6,7,8

2.3. Заpезеpвувати в області пам'яті даних DST по 1 байту 8 порожніх байти - DST DB 8 DUP (?).

2.4. Заpезеpвувати в області пам'яті даних ABC по 1 байту 3 порожніх байти - ABC DB 3 DUP (?).

2.5.Доповнити програму наступними операціями:

1. Сформувати адресу сегмента даних
2. Виконати пересилання даних з регістрів у пам'ять, використовуючи команди з прямою адресацією:

AH ---> ABC MOV ABC,AH

BH ---> ABC+1 MOV ABC+1,BH

CH ---> ABC+2 MOV ABC+2,CH

3) Виконати пересилання вмісту області пам'яті с початковою адресою SRC (джерело) в іншу початковою адресою DST (приймач) із залученням стеку.



2.6. Отримати файл із розширенням .EXE Завантажите вашу програму в відладчик. Виконаєте трасування пpогpами по кроках, продивляючись на кожному кроці зміну стану pегістpів.

**ЗАВДАННЯ 4.** Реалізувати програму кодування (перетворення) двійкового числа в двозначне шістнадцяткове число. Отримати файл із розширенням .EXE Завантажите вашу програму в відладчик. Виконаєте трасування пpогpами по кроках, продивляючись на кожному кроці зміну стану pегістpів.

**Контрольні питання**

1. Приведіть загальний формат команди визначення даних для програм на Ассемблері.

2. Перелічите директиви визначення даних мови ассемблер.

3. Яке призначення команди MOV?

4. Яке призначення команди LEA?

5. Яке призначення команди XLAT?

6. Яке призначення команди INC?

7. Якою директивою даних визначається символьний рядок. Приведіть приклад.

8. Призначення команди PUSH, приведіть загальний формат команди, приведіть приклад використання цієї команди.

9. Призначення команди POP, приведіть загальний формат команди, приведіть приклад використання цієї команди.

10. Призначення команди XCHG, приведіть приклад використання цієї команди.