Лабораторна робота № 6
РЯДКОВІ ІНСТРУКЦІЇ І МАСИВИ

Мета роботи: вивчити команди переміщення даних, команди повторення строкових інструкцій; вивчити організацію масивів мовою Асемблер.

Теоретичні відомості

Команди роботи з рядками

Рядок – це ланцюжок байтів, для якого є відомим алгоритм визначення його довжини. У мовах програмування прописують процедуру визначення довжини рядка й уводять обмеження на те, які байти можуть міститися в рядку, а також у якому кодуванні мають інтерпретуватися рядки: скільки байтів кодує символ і яка таблиця буде використовуватися для зіставлення байтів і символів. У мові С під рядком розуміється послідовність байтів, що закінчується байтом зі значенням 0.

Рядкові операції працюють з одним елементом рядка: байтом, словом, подвійним словом. Робота з рядками може виконуватися з допомогою команд роботи з рядками (які виконують тільки одну ітерацію для кожної операції) спільно з іншими можливостями команд процесора, такими, як префікси повторення. Команди роботи з рядками працюють з великими структурами даних у пам'яті, такими, як алфавітно-цифрові рядки символів:

- **MOVS** – переслати один байт або одне слово з однієї області пам'яті в іншу;
- **LODS** – завантажити з пам'яті один байт у регістр `AL` або одне слово в регістр `AX`;
- **STOS** – записати вміст регістра `AL` або `AX` у пам'ять;
- **CMPS** – порівняти вміст двох областей пам'яті розміром один байт або одне слово;
- **SCAS** – порівняти вміст регістра `AL` або `AX` з вмістом пам'яті.

Ці команди можна «налаштувати» на оброблення байта або слова. Для цього після службового слова (назви команди) необхідно додати букву `B` (байт) або W (слово), наприклад: `MOVSB` або `MOVSW`. Якщо для команди `MOVS` наявність операндів є обов'язковою, то для команд `MOVSB` і `MOVSW` їх може не бути. При застосуванні цих команд припускається, що регістри `DI` і `SI` містять відносні адреси, які вказують на необхідні області пам'яті (для їх завантаження можна використовувати команду `LEA`). Регістр `SI` зазвичай зв'язаний з регістром сегмента даних – `DS:SI`. Регістр `DI` завжди зв'язаний з

регістром додаткового сегмента – ES:DI. Отже, команди MOVS, STOS, CMPS і SCAS потребують ініціалізації регістра ES.

Кожна з перелічених команд може мати префікс REP, який дає змогу цим командам обробляти рядки будь-якої довжини. Префікс кодується безпосередньо перед ланцюжковою командою, наприклад:

```
REP MOVSB
```

Для використання префікса REP необхідно встановити кількість повторень у регістрі CX. При виконанні ланцюжкової команди з префіксом REP відбувається зменшення на одне значення в регістрі CX до нуля. Таким чином, можна обробляти рядки будь-якої довжини.

Прапорець напрямку визначає напрямок повторюваної операції:

– для напрямку зліва направо необхідно з допомогою команди CLD установити прапорець DF у 0;

– для напрямку справа наліво необхідно з допомогою команди STD установити прапорець DF у 1.

У наступному прикладі виконується пересилання 10 байтів з області START в область END_S. Припустимо, що обидва регістри DS і ES ініціалізовано адресою сегмента даних:

```
. . .
START DB 10 DUP('*')
END_S DB 10 DUP(' ')
. . .
CLD ;Скидання флагу DF.
MOV CX,10 ;Лічильник на 10 байт.
LEA DI,END_S ;Адреса області "куди".
LEA SI,START ;Адреса області "звідки".
REP MOVSB ;Переслати дані.
. . .
```

Для області, яка приймає рядок, сегментним є регістр ES, а регістр DI містить відносну адресу області, яка приймає рядок. Для області, що передає рядок, сегментним є регістр DS, а регістр SI містить відносну адресу. Таким чином, на початку програми перед виконанням команди MOVS необхідно ініціалізувати регістр ES разом з регістром DS, а також завантажити необхідні відносні адреси полів у регістри DI і SI. Залежно від стану прапорця DF команда MOVS виробляє збільшення або зменшення на один байт або на два байти (для слова) вмісту регістрів DI і SI.

Наведемо команди, що є еквівалентними ланцюжковій команді REP MOVSB:

```
LABEL1: JCXZ LABEL2 ;Перехід, якщо CX=0.
MOV AL,[SI]
MOV [DI],AL
INC/DEC DI ; Інкремент або декремент.
INC/DEC SI ; Інкремент або декремент.
LOOP LABEL1
LABEL2: . . .
```

При використанні **команд MOVSB** або **MOVSW** Асемблер припускає наявність коректної довжини рядкових даних і не потребує кодування операндів у команді. Для команди MOVS довжину має бути закодовано в операндах. Наприклад, якщо поля FLDA і FLDB визначено як байтові (DB), то команда

```
REP MOVS FLDA,FLDB
```

передбачає повторюване пересилання байтів з поля FLDB у поле FLDA. Цю команду можна також записати в такому вигляді:

```
REP MOVS ES:BYTE PTR[DI], DS:[SI].
```

Однак завантаження регістрів DI і SI адресами FLDA і FLDB є обов'язковим у будь-якому випадку.

Команда LODS завантажує з пам'яті в регістр AL один байт або в регістр AX одне слово. Адреса пам'яті визначається регістрами DS:SI. Залежно від значення прапорця DF відбувається збільшення або зменшення значення регістра SI.

Команду LODS зручно використовувати в тому випадку, коли потрібно просуватися уздовж рядка (по байту або по слову), перевіряючи завантаження регістра на конкретне значення.

Команди, що є еквівалентними команді LODSB:

```
MOV AL,[SI]
INC SI
```

Команда STOS записує (зберігає) вміст регістра AL або AX у байті або в слові пам'яті. Адреса пам'яті завжди подається регістрами ES:DI. Залежно від прапорця DF команда STOS також збільшує або зменшує адресу в регістрі DI на один байт або на два байти (для слова).

Цю команду з префіксом REP можна використовувати для ініціалізації області даних конкретним значенням. Довжина області (один байт або одне слово) завантажується в регістр AX.

Команди, що є еквівалентними команді REP STOSB:

```
LABEL1: JCXZ LABEL2
MOV [DI],AL
INC/DEC DI ; Інкремент або декремент.
LOOP LABEL1
LABEL2: . . .
```

Команда STOS є корисною для установлення в деякій області певних значень байтів і слів. Для дублювання зразка, довжина якого перевищує розмір слова, можна використовувати команду MOVS з невеликою модифікацією. Припустимо, що необхідно сформувати рядок, у якому багато разів повторюються символи: '11 ** '. Спочатку визначимо перші чотири байти, які розташуємо безпосередньо перед оброблюваним рядком таким чином:

```
PATTERN DB '11**'
DISAREA DB 44 DUP(?)
. . .
CLD
MOV CX,22
LEA DI,DISAREA
LEA SI,PATTERN
REP MOVSW
. . .
```

Під час виконання команда MOVSW спочатку пересилає перше слово (символи '11') зі зразка PATTERN у перше слово області DISAREA, потім – друге слово (символи '**'). До цього моменту регістр DI буде містити адресу DISAREA+4, а регістр SI – адресу PATTERN+4, яка також є адресою DISAREA. Потім команда MOVSW автоматично дублює зразок, пересилаючи перше слово з DISAREA в DISAREA+4, з DISAREA+2 в DISAREA+6, з DISAREA+4 в DISAREA+8 і т. д. Унаслідок цього зразок буде повністю продубльованим по всій області DISAREA.

Цій спосіб можна використовувати для дублювання в області пам'яті будь-якого зразка будь-якої довжини, тільки його необхідно розташувати безпосередньо перед приймальною областю.

Команда CMPS порівнює вміст однієї області пам'яті (адресується регістрами DS:SI) з вмістом іншої області (адресується як ES:DI). Залежно від прапорця DF команда CMPS також збільшує або зменшує адреси в регістрах SI і DI на один байт або на два байти (для слова). Команда CMPS установлює прапорці AF, CF, OF, PF, SF і ZF. При використанні префікса REP в регістрі CX має знаходитися довжина порівнюваних полів. Команда CMPS може порівнювати будь-яку кількість байтів або слів.

При виконанні команд CMPS і SCAS можливим є установлення прапорців стану так, щоб операція могла припинитися відразу після виявлення необхідної умови. Наведемо модифікації префікса REP для цих цілей:

– REP **–** повторювати операцію, поки CX не дорівнює нулю;

– REPZ або REPE – повторювати операцію, поки прапор ZF показує «дорівнює або нуль»; припинити операцію, коли прапорець ZF буде показувати «не дорівнює або не нуль» або при CX = 0;

– REPNE або REPNZ **–** повторювати операцію, поки прапорець ZF показує «не дорівнює або не нуль»; припинити операцію, коли прапорець ZF буде показувати «дорівнює або нуль» або при CX = 0.

Ініціалізація масивів

В одній директиві визначення даних може вказуватися кілька значень. Наприклад, директива

```
SampleArray DW 0, 1, 2, 3, 4
```

створює масив з п'яти елементів з ім'ям SampleArray, елементи якого мають розмір одне слово (рис. 7.1). У директивах визначення даних можна використовувати будь-яку кількість значень, що уміщаються на рядку.

Для визначення масиву, який є занадто великим і не може вміститися на одному рядку, потрібно просто додати кілька рядків. Мітку в директиві визначення даних вказувати не обов'язково. Наприклад, за директивами

```
SquareArray DD 0, 1, 4, 9, 16
DD 25, 36, 49, 64, 81
DD 100, 121, 144, 169, 196
```

створюється масив елементів розміром одне подвійне слово з ім'ям SquareArray, що складається з квадратів перших 15 цілих чисел.

Турбоасемблер дає змогу визначити блок пам'яті, ініціалізований певним значенням, з допомогою операції DUP. Наприклад:

```
BlankArray DW 100h DUP (0)
```

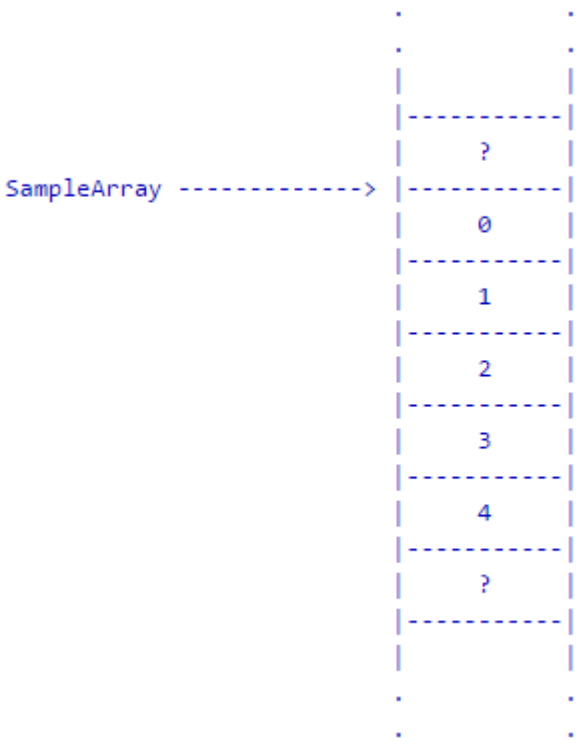

Рис. 7.1. Приклад масиву з п'яти елементів

Тут створюється масив `BlankArray`, що складається з 255 (десяткових) слів, ініціалізованих значенням 0. Аналогічно директива

```
ArrayOfA DB 92 DUP ('A')
```

створює масив з 92 байтів, кожен з яких ініціалізовано символом «`A`».

Оброблення масивів

Нехай є масив `x` і змінна `n`, що зберігає кількість елементів цього масиву:

```
x dd 100 dup(?)
n dd ?
```

Для оброблення масиву можна використовувати кілька способів.

1. У регістрі можна зберігати зміщення елемента масиву:

```
 mov eax, 0
 mov ecx, n
 mov ebx, 0
L: add eax, x[ebx]
 add ebx, type x
```

```
 dec ecx
 cmp ecx, 0
 jne L
```

2. У регістрі можна зберігати номер елемента масиву й множити його на розмір елемента:

```
 mov eax, 0
 mov ecx, n
L: dec ecx
 add eax, x[ecx * type x]
 cmp ecx, 0
 jne L
```

3. У регістрі можна зберігати адресу елемента масиву. Адресу початку масиву можна записати в регістр з допомогою команди `LEA`:

```
 mov eax, 0
 mov ecx, n
 lea ebx, x
L: add eax, [ebx]
 add ebx, type x
 dec ecx
 cmp ecx, 0
 jne L
```

4. За необхідності можна в один регістр записати адресу початку масиву, а в іншій – номер або зміщення елемента масиву:

```
 mov eax, 0
 mov ecx, n
 lea ebx, x
L: dec ecx
 add eax, [ebx + ecx * type x]
 cmp ecx, 0
 jne L
```

Модифікувати адресу можна також за двома з регістрів:

```
x [ebx][esi]
```

Це може бути зручно при роботі зі структурами даних, які розглядаються як матриці.

Приклад підрахунку кількості рядків матриць з позитивною сумою елементів:

```
mov esi, 0        ;Початкове зміщення рядка
mov ebx, 0        ;EBX буде містити кількість рядків,
```

```
                    ;які відповідають умові
     mov ecx, m     ;Завантажуємо в ECX кількість
рядків
    L1: mov edi, 0  ;Початкове зміщення елемента в
рядку
    mov eax, 0      ;EAX буде містити суму елементів
                    ;рядка
    mov edx, n      ;Завантажуємо в EDX кількість
                    ;елементів у рядку
    L2: add eax, y[esi][edi]  ;Додаємо до EAX елемент
                              ;масиву
    add edi, type y ;Додаємо до зміщення елемента в
рядку
                    ;розмір елемента
    dec edx         ;Зменшуємо на одиницю лічильник
                    ;внутрішнього циклу
    cmp edx, 0      ;Порівнюємо EDX з нулем
    jne L2          ;Якщо EDX не дорівнює нулю, то
                    ;переходимо до початку циклу
    cmp eax, 0      ;Після циклу порівнюємо суму
                    ;елементів рядка з нулем
    jle L3          ;Якщо сума менше або дорівнює нулю,
                    ;то обходимо збільшення EBX
     inc ebx        ;Якщо ж сума більше нуля,
                    ;то збільшуємо EBX
    L3: mov eax, n  ;Завантажуємо в EAX кількість
                    ;елементів в рядку
    imul eax, type y ;Множимо кількість елементів у
рядку
                    ;на розмір елемента
    add esi, eax    ;Додаємо до зміщення отриманий
розмір
                    ;рядка
    dec ecx         ;Зменшуємо на одиницю лічильник
                    ;зовнішнього циклу
    cmp ecx, 0      ;Порівнюємо ECX з нулем
    jne L1          ;Якщо ECX не дорівнює нулю,
                    ;то переходимо до початку циклу
```

Завдання

1. Прочитайте завдання, яке виконує наведена програма, напишіть цю програму з ім'ям `Lab7.asm`.

```
;Задано два масиви ArrayA, ArrayB, що складаються з
```

```
    ;10 елементів кожен.
    ;Порівняти ці масиви поелементно, і якщо елементи є
    ;однаковими, то записати у відповідний елемент
    ;третього масиву (Difference) 'Y', якщо різними - то
'N'
    ;Знайти суму й кількість всіх однакових і різних
    ;елементів двох масивів (ArrayA, ArrayB)

    .model tiny
    .stack 100h
    .data
     ArrayA db 05,10,06,44,20,32,05,11,46,0
     ArrayB db 35,10,15,44,20,02,65,10,46,0
     Difference db 10 dup (0)
     NumOfDiff dw 0
     NumOfEqual dw 0
     SumOfDiff dd 0
     SumOfEqual dd 0
    .code
    start:
         mov ax,@data
         mov ds,ax
         push ds
         pop es
         mov di,offset Difference
         mov cx,10
         mov al,'Y'
         cld
         rep stosb
         mov si,offset ArrayA
         mov di,offset ArrayB
         mov bx,offset Difference
         mov cx,10
         cld
    findDE:
         cmpsb
         jne NotEqual
         inc NumOfEqual
         inc bx
         dec di
         dec si
         mov al,byte ptr ds:[si]
         cbw
         add SumOfEqual, ax
         mov al,byte ptr ds:[di]
```

```
        cbw
        add SumOfEqual, ax
        inc si
        inc di
        jmp NextElement
    NotEqual:
        inc NumOfDiff
        mov byte ptr ds:[bx],'N'
        inc bx
        dec di
        dec si
        mov al,byte ptr ds:[si]
        cbw
        add SumOfDiff, ax
        mov al,byte ptr ds:[di]
        cbw
        add SumOfDiff, ax
        inc si
        inc di
    NextElement:
        loop findDE
        mov ax,4c00h
        int 21h
    end start
```

2. Створіть виконуваний файл і простежте за його роботою в Турбоналагоджувачі.

3. Перепишіть програму у звіт і заповніть коментарями про зміст регістрів усі рядки в сегменті коду.

4. Запишіть наведену нижче програму й простежте за її роботою. У звіт занесіть результати роботи програми.

```
    .
    // string.cpp: визначає точку входу для консольного
    //застосування

    #include "stdafx.h"
    #include <windows.h> //необхідний для роботи DWORD
    #include <stdio.h> // необхідний для роботи printf
    #include <conio.h> // необхідний для роботи _getch()
    /* Оголошення функції */
    DWORD lens (char *); // Функція визначення довжини
рядка
    void cats(char*, char*,char*); // функція злиття двох
                                   //рядків у третій
    /* Оголошення змінних */
```

```
char a[]="fdhfjliop";
char b[]="12345";
char c[]="4";

int _tmain(int argc, _TCHAR* argv[])
{
   __asm{
```
/* необхідно викликати процедуру cats, яка за своїї параметри має три покажчика на рядок типу char (у порядку a, b, c). Щоб процедура cats могла отримати параметри, що передаються за значенням, необхідно помістити їх у стек (у порядку c, b, a), а потім вирівняти стек */
```
    LEA EAX, c; // поміщаємо у регістр EAX адресу першого
                // елемента рядка c
    PUSH EAX; // поміщаємо вміст EAX у стек
    LEA EAX, b; // поміщаємо в регістр EAX адресу першого
                // елемента рядка b
    PUSH EAX; // поміщаємо вміст EAX у стек
    LEA EAX, a; // поміщаємо в регістр EAX адресу першого
                // елемента рядка a
    PUSH EAX; // поміщаємо вміст EAX у стек
    CALL cats; // викликаємо процедуру
    ADD ESP,12; // вирівнюємо стек
    };
    printf("%s\n", c); // виводимо сумарний рядок
    _getch();
    return 0;
}

// функція визначення довжини рядка
DWORD lens (char * s)
{
    DWORD l=0;
__asm
    {
```
CLD; /* задаємо напрямок сканування (скидаємо прапорець напрямку DF). Цей прапорець ураховується в рядкових операціях. Якщо прапорець дорівнює одиниці, то в рядкових операціях адреса автоматично зменшується. Щоб адреса автоматично збільшувалася, цей прапорець необхідно скинути*/
```
    MOV EDI, s; // поміщаємо адресу початку рядка в
                // регістр EDI
    MOV ESI,EDI; // зберігаємо адресу рядка в регістрі ESI
```

```
    MOV ECX, 0ffffffffh; // поміщаємо в регістр-
лічильник                // циклу максимальне 32-бітове
ціле число
    XOR AL,AL; // запускаємо нескінченний цикл – операція
            // XOR від двох однакових елементів повертає
0
    // таким чином, ця команда являє собою "поки 0 ..."
    REPNE SCASB; // скануємо рядок байтів, поки
                 // не трапиться 0
    SUB EDI,ESI; // віднімаємо з поточної адреси, яка
                 // зберігається в EDI, адресу початку
                 // рядка (що зберігається в ESI), таким
                 // чином, знаходимо довжину рядка разом
                 // з термінальним символом
    DEC EDI; // віднімаємо з отриманої довжини рядка 1
            // виключаємо термінальний символ
    MOV l, EDI; // поміщаємо в змінну l довжину рядка
   }
   return l; // повертаємо довжину рядка
   }

   // Функція злиття рядків s1+s2->s3
   void cats (char* s1, char* s2,char* s3)
   {
   __asm{
    /* визначаємо довжину рядка s1*/
    CLD;         // задаємо напрямок копіювання
    MOV ESI,s1; // поміщаємо в регістр ESI покажчик
                 // на початок рядка  s1
    PUSH ESI; // поміщаємо покажчик на s1 у стек
    CALL lens; // викликаємо функцію визначення довжини
              // рядка
    ADD ESP,4; // вирівнюємо стек

    /* копіюємо вміст рядка s1 у рядок s3*/
    /* функція lens повертає результат у регістр EAX */
    MOV ECX, EAX;   // поміщаємо довжину рядка s1 у
                    // регістр ECX
    MOV ESI,s1;  // поміщаємо в регістр ESI покажчик
                  // на рядок  s1
    MOV EDI,s3;   // поміщаємо в регістр EDI покажчик
                  // на рядок  s3
    REP MOVSB;  /* повторюємо, поки вміст ECX не
повернувся в 0, побайтове копіювання рядка, адреса
початку якого зберігається в ESI, в рядок, адреса початку
```

якого зберігається в EDI (копіюємо вміст рядка s1 у рядок s3) */

/* копіюємо вміст рядка s2 у рядок s3, починаючи з позиції після s1 */

```
MOV ESI,s2; // поміщаємо в ESI покажчик на перший
            // елемент рядка  s2
PUSH ESI; // поміщаємо покажчик на s2 у стек
CALL lens;  // викликаємо функцію визначення довжини
            // рядка
ADD ESP,4;  // вирівнюємо стек
/* функція lens повертає результат у регістр EAX */
MOV ECX,EAX;     // поміщаємо довжину рядка s21 у
                 // регістр ECX
```

REP MOVSB; /* повторюємо, поки вміст ECX не повернувся в 0, побайтове копіювання рядка, адреса початку якого зберігається в ESI (адреса початку рядка s2), у рядок, адреса початку якого зберігається в EDI (адреса кінця рядка s1 + 1) (копіюємо вміст рядка s2 в рядок s3) */

```
MOV BYTE PTR [EDI],0; // фіксуємо кінець рядка
}
}
```