Лабораторна робота № 6

ОРГАНІЗАЦІЯ ЦИКЛІВ І РОБОТА З ЦІЛОЧИСЛЕНИМИ МАСИВАМИ

Мета заняття: ознайомитися з основними командами мови Assembler для організації циклів і роботи з цілочисельними масивами; набути практичних навичок в написанні програм з використанням циклів та масивів на мові Assembler.

Теоретичні відомості

Команди управління циклами LOOPx

Команди цього виду організують циклічні обчислення (LOOP - цикл), використовуючи регістр лічильника CX по своєму прямому призначенню. В регістр CX має бути попередньо занесена кількість повторень циклу. Ці команди реалізують класичний цикл з лічильником з постумовою.

1. Команда LOOP – перехід по лічильнику

Для організації цикла призначена команда LOOP. У цієї команди один операнд – ім'я мітки, на яку здійснюється перехід. В якості лічильника цикла використовується регістр CX. Команда LOOP виконує декремент CX, а потім перевіряє його значення. Якщо вміст CX не дорівнює нулю, то виконується перехід на мітку, інакше управління переходить до наступної після LOOP команди.

Вміст CX інтерпретується командою як число без знака. В CX потрібно поміщати число, яке дорівнює необхідній кількості повторень циклу. Зрозуміло, що максимально може бути 65535 повторень. Ще одне обмеження пов'язане з дальністю переходу. Мітка має знаходитись в діапазоні 127…+128 байт від команди LOOP.

Синтаксис команди:

LOOP коротка_мітка

Логіка роботи команди:

<CX> = <Counter>

Short_label: Виконання тіла циклу

<CX> = <CX> - 1

if (<CX> < > 0) goto short_label

Аналог реалізації команди LOOP на Асемблері:

```
        MOV CX, Counter
short_label:
        ; Виконання тіла циклу
        ;Перевірка умови ПРОДОВЖЕННЯ цикла
        DEC CX
        CMP CX, 0
        JNE short_label
```

Команда LOOP зменшує вміст регістра CX на 1. Потім передає управління мітці short_label, якщо вміст CX не дорівнює 0. Оскільки умова виходу із циклу перевіряється в кінці, при значенні Counter=0 цикл все одно виконається.

Щоб цього уникнути, зазвичай до початку циклу перевіряють вміст регістра CX на нуль. Таким чином, стандартна послідовність команд для організації цикла з лічильником має наступний вигляд:

```
    MOV CX, Counter
    JCXE ExitCicle; якщо <CX> = 0, цикл обійти
 short_label:
    ;Виконання тіла циклу
    LOOP short_label
```

Іноді потрібно організувати вкладений цикл, тобто цикл всередині іншого циклу. В цьому випадку необхідно зберегти значення CX перед початком вкладеного циклу і відновити після його закінчення (перед командою LOOP зовнішнього циклу). Зберегти значення можна в інший регістр, в тимчасову змінну або в стек.

2. Команда LOOPE (LOOPZ) перехід по лічильнику і якщо дорівнює

Дана команда має два рівнозначних мнемонічних імені (if Equal – якщо Дорівнює або if Zero – якщо Нуль). Мнемоніка команди LOOPZ передбачає знання того факту, що якщо два операнди однакові, прапорець нуля ZF=1 (встановлений).

Синтаксис команди:

LOOPE коротка_мітка
LOOPZ коротка_мітка

Логіка роботи команди:

```
        <CX> = <Counter>
Short_label: Виконання тіла циклу
        <CX> = <CX> - 1
        if (<CX> < > 0 and <ZF>=1) goto short_label
```

Все, що було сказане для команди LOOP, виконується і для команди LOOPE (LOOPZ), додається тільки перевірка прапорця ZF. Застосовується дана команда у випадку, якщо потрібно просто вийти з циклу, як тільки знаходиться перший елемент, відмінний від заданої величини.

3. Команда LOOPNE (LOOPNZ) перехід по лічильнику і якщо не дорівнює

Дана команда також має два рівнозначних мнемонічних імені (if Not Equal – якщо Не дорівнює або if Not Zero – якщо Не нуль). На відміну від попередньої команди перевіряється, чи скинутий прапорець нуля ZF=0.

Синтаксис команди:

LOOPNE коротка_мітка

LOOPNZ коротка_мітка

Логіка роботи команди:

<CX> = <Counter>

Short_label: Виконання тіла циклу

<CX> = <CX> - 1

if (<CX> < > 0 and <ZF>=0) goto short_label

Все, що було сказане для попередньої команди, виконується і для команди LOOPNE (LOOPNZ). Застосовується дана команда у випадку, якщо потрібно просто вийти з циклу, як тільки знаходиться перший елемент, рівний заданій величині.

Команда завантаження ефективної адреси (зсуву) джерела LEA (Load Effective Address)

Команда LEA схожа на команду mov, але відмінність lea від mov складається в тому, що використовується механізм блоку адресації процесора, а не арифметико-логічного блоку.

Ця команда допускає індексацію операнда, що дозволяє зручно застосовувати її для визначення адреси параметра, що знаходиться в стеці. Наприклад, якщо в процедурі визначається локальний масив, то для роботи з ним часто необхідно завантажити його місце в регістр індексу (як раз це можна зробити командою LEA). Виконання команди LEA не впливає на прапори.

Синтаксис команди:

LEA Приймач, Джерело

Логіка роботи команди:

<Приймач> = <Значення зсуву джерела>

Роботи команди залежить від типу даних адресації (16 чи 32 біти). Якщо дані 16-бітні, то в регістр приймач завантажується 16-бітне значення зсуву джерела, якщо дані 32-бітні, то в регістр приймач завантажується 32-бітне значення зсуву джерела.

Команди завантаження даних (8/16/32)біт LODS/LODSB/LODSW/LODSD (LOad String Byte/Word/Double word operands)

Команди групи LODS завантажують елемент з послідовності (ланцюга) у регістр-акумулятор. Виконання команди LODS не впливає на прапори. Операнди їм не потрібні.

Синтаксис команди:

LODSB

LODSW

LODSD

Логіка роботи команди:

– завантажити елемент із комірки пам'яті, що адресується парою ds:esi/si, у регістр al/ax/eax. Розмір елемента визначається неявно (для команди LODS) чи явно відповідно до застосовуваної команди (для команд LODSB, LODSW, LODSD);

– змінити значення регістра si на величину, рівну довжині елемента ланцюга. Знак цієї величини залежить від стану прапора DF.

Якщо DF = 0 – значення позитивне, тобто перегляд від початку ланцюга до його кінця, якщо DF = 1 – значення негативне, тобто перегляд від кінця ланцюга до його початку.

Прапор напряму DF (10 біт у регістрі прапорів) управляє рядковими інструкціями (MOVS, CMPS, SCAS, LODS та STOS) – встановлення прапора в 1 змушує зменшувати адреси (обробляти рядки від старших адрес до молодших), обнулення змушує збільшувати адреси. Інструкція STD встановлює 1 в прапор DF а CLD відповідно обнуляє прапор DF.

Команди зсувів

1. Логічні зсуви

Команди логічних зсувів SHL і SHR зсувають біти операнда ліворуч або праворуч відповідно на один розряд і змінюють прапор переносу CF. При логічному зсуві всі біти рівноправні, а біти, що звільнилися, заповнюються нулями. Повторення відбувається стільки разів, скільки дорівнює значення <Джерела>. В якості <Джерела> – кількість зсувів ліворуч/праворуч.

Синтаксис команд:

SHL Приймач, Джерело

SHR Приймач, Джерело

Приклад команди shr (логічний зсув вправо):

	Біти								Висувається	CF
	7	6	5	4	3	2	1	0		
mov al, 91	0	1	0	1	1	0	1	1		
SHR al, 3				0	1	0	1	1	**011**	
Результат	**0**	**0**	**0**	0	1	0	1	1		**0**

Після виконання команди SHR відбувається зсув всіх бітів регістра al на 3 розряди вправо. Біти зліва заповнюються нулями, а біти праворуч висуваються. Останній висунутий біт стає значенням прапора переносу CF.

Команда SHL аналогічна SHR, але зсуває вліво.

Після виконання останні біти заповнилися нулями, прапор перенесення заповнюється значенням останнього висунутого біта.

2. Арифметичні зсуви

Команда SAR – зсуває біти операнда (реєстр/пам'ять) праворуч на один розряд, значення останнього виштовхнутого біта потрапляє у прапор переносу, а біти, що звільнилися, заповнюються знаковим бітом.

Команда SAL – зсуває біти операнда (реєстр/пам'ять) ліворуч на один розряд, значення останнього виштовхнутого біта потрапляє у прапор переносу, а біти, що звільнилися, заповнюються нулями, при цьому знаковий біт не рухається.

В якості <Джерела> – кількість зсувів ліворуч/праворуч.

Синтаксис команд:

SAL Приймач, Джерело

SAR Приймач, Джерело

Приклад команди sar (аріфметичні зсув вправо):

	Біти								Висувається	CF
	7	6	5	4	3	2	1	0		
mov al, 155	**1**	0	0	1	1	0	1	1		
SAR al, 3				1	0	0	1	1	**011**	
Результат	**1**	**1**	**1**	1	0	0	1	1		**0**

3. *Циклічні зсуви*

Циклічний зсув нагадує зміщення, біти, що висуваються, знову всуваються з іншого боку. В якості <Джерела> – кількість зсувів ліворуч/праворуч.

Синтаксис команд:

ROL Приймач, Джерело
ROR Приймач, Джерело
RCL Приймач, Джерело
RCR Приймач, Джерело

Приклад команди ror (циклічний зсув вправо):

	Біти								Висувається	CF
	7	6	5	4	3	2	1	0		
mov al, 155	1	0	0	1	1	0	1	1		
ROR al, 3				1	0	0	1	1	**011**	
Результат	**0**	**1**	**1**	1	0	0	1	1		**0**

З прикладу видно, що біти обертаються, тобто кожен біт, який виштовхується, знову вставляється з іншого боку. Прапор перенесення CF містить значення останнього висунутого біта.

ROL і ROR зсувають всі біти операнда вліво або вправо на один розряд, при цьому старший або молодший біт операнда всувається в операнд праворуч або ліворуч і стає значенням молодшого або старшого біта операнда; одночасно висувається біт стає значенням прапора перенесення CF. Зазначені дії повторюються кількість разів, що дорівнює значенню другого операнда.

RCL і RCR зсувають усі біти операнда вліво або вправо на один розряд, при цьому старший або молодший біт стає значенням прапора перенесення CF; одночасно старе значення прапора переносу CF всувається в операнд праворуч або ліворуч і стає значенням молодшого або старшого біта операнда. Зазначені дії повторюються кількість разів, що дорівнює значенню другого операнда.

Команда TEST

TEST (logical compare – логічне порівняння) – виконується як команда неруйнівного логічного множення, єдиним результатом якої є встановлення арифметичних прапорів для подальшого умовного переходу. Один з операндів містить величину, що перевіряється, а інший – маску, відповідну бітам, що перевіряються.

Данна команда впливає тільки на прапори (тобто перший операнд не змінюється).

Синтаксис команди:

TEST Приймач, Джерело

Залежно від результату можуть бути змінені прапори ZF, SF, PF. Інструкція TEST завжди скидає прапори OF і CF.

Існує декілька варіанта команди TEST:

TEST Register Register
TEST Register Memory
TEST Register Constant
TEST Memory Register
TEST Memory Constant

Команда TEST зазвичай використовується для перевірки бітів спільно з командами умовного переходу.

Крім того, за допомогою інструкції TEST можна визначити стан відразу декількох бітів числа.

Припустімо, ми хочемо дізнатися, чи скинуті нульовий і третій біти числа в регістрі AL. Тоді можна використовувати таку команду з бітовою маскою, де встановлені 3-й і 0-й біти:

TEST AL, 00001001b

А тепер кілька прикладів, які показують, як працює цей код.

0 0 1 0 0 0 0 1 1 0 1 – Початкове значення
0 0 0 0 0 0 1 0 0 0 1 – Бітова маска
0 0 0 0 0 0 0 0 0 0 1 – Результат: ZF = 0

0 0 0 1 0 0 1 0 0 0 0 – Початкове значення
0 0 0 0 0 0 1 0 0 0 1 – Бітова маска
0 0 0 0 0 0 1 0 0 0 0– Результат: ZF = 0

0 0 0 1 0 0 0 1 0 0 0 – Початкове значення
0 0 0 0 0 0 1 0 0 0 1 – Бітова маска
0 0 0 0 0 0 0 0 0 0 0 – Результат: ZF = 1

Тобто прапор нуля ZF буде встановлено тільки в тому разі, якщо обидва біти (0-й і 3-й) скинуто.

Приклад програми:

```
mov AX, 5
test AX, 1;      перевірка числа на парність
JNZ Odd1 ;     непарне, перехід на мітку Odd
JZ Even1;       парне, перехід на мітку Even

Odd:              якщо число в AX непарне
…
JMP Exit1
…
Even:            якщо число в AX парне
…
JMP Exit1
…
```

Тут ми перевіряємо, чи є число в регістрі AX парним або непарним. І залежно від результату переходимо до тієї чи іншої мітки.

Основні принципи організації і обробки масивів

Масив – структурований тип даних, який складається із деякої кількості елементів одного типу. Впорядкованість масива визначається набором цілих чисел, які називаються індексами. Індекси зв'язуються з кожним елементом масиву і однозначно конкретизують його розміщення серед інших елементів масиву. Локалізація конкретного елемента масиву – основна задача при розробці будь-яких алгоритмів, що працюють з масивами. Її складність напряму залежить від розмірності масиву.

1. Одномірні масиви

Одномірні масиви мають найпростіше представлення. Структура зберігання, яка їм відповідає – вектор. Вона однозначна, і представляє собою послідовне розміщення елементів в пам'яті. Щоб локалізувати потрібний елемент одномірного масиву, необхідно знати його індекс. Так як асемблер не має засобів для для роботи з масивом як з структурою даних, то для доступу до елементу масиву, необхідно визначити його адресу.

Нехай два масиви і вказівники на них на мові C/C++ описані наступним чином:

```
int ArrI[10],      *PtI;   PtI = ArrI;
float ArrF [5],    *PtF;   PtF = ArrF;
```

Зобразимо у вигляді таблиць основні характеристики цих масивів.

Таблиця 7.1

Основні характеристики цілочисельного масиву int ArrI[10] на мові C/C++

Байти в ОЗП	1	2	3	4	…	17	18	19	20
Елементи масиву	ArrI[0]		ArrI[1]		…	ArrI[8]		ArrI[9]	
Вказівники	ArrI		ArrI+1		…			ArrI+9	
	PtI		PtI+1		…	PtI+8		PtI+9	

PtF + 2 означає те ж саме, що і &ArrF[2] і ArrF+2 – це одна і та ж адреса масиву ArrF[2].

Таблиця 7.2

Основні характеристики дійсного масиву float ArrF[5]на мові C/C++

Байти в ОЗП	1	2	3	4	5	6	7	8	…	17	18	19	20
Елементи масиву	ArrF[0]				ArrF[1]				…	ArrF[4]			
Вказівники	ArrF				ArrF+1				…	ArrF+4			
	PtF				PtF+1				…	PtF+4			

Для того, щоб обробляти масив в Асемблері, потрібно знати, де він зберігається, і довжину його елемента. Ім'я масиву в Асемблері є також і його початковою адресою.

Режим адресації з індексацією вигляду *ім'я_масиву [регістр_індексу]* дозволяє обробляти кожний елемент масиву. В якості *регістру_індексу* можна використовувати будь-який допустимий для непрямої адресації регістр, наприклад, регістр BX.

Таблиця 7.3

Основні характеристики дійсного масиву float ArrF[5] в Асемблері

Байти в ОЗП	1	2	3	4	5	6	7	8	…	17	18	19	20
Індекс-змінна	0				1					4			
Елементи масиву	ArrF[0]				ArrF[1]				…	ArrF[4]			
Адреси (зміщення)	Arr F	Arr F+ 1	Arr F+ 2	Arr F+ 3	…				…	Arr F+ 16	Arr F+ 17	Arr F+ 18	Arr F+ 19
Індекс-регістр	BX_0				BX_0+4				…	BX_0+4*i			

В загальному випадку, якщо взяти в якості індекс-регістра регістр BX, доступ до будь-якого елемента одномірного масиву Array [i] довжини Larray підпорядковується в Асемблері наступній закономірності:

Array[i] → Array + BX_i = Array[BX_i], де

$BX_i = BX_0$ + Larray*I = BX_{i-1} + Larray;

BX_0 = 0; i=0,…,n-1; n – довжина масиву Array.

2. Двомірні масиви

Для двомірних масивів ідея та ж сама, тільки необхідно визначитися, як такий масив буде розміщуватися в оперативній пам'яті: по рядкам чи по стовбцям. Відповідно і індекс-регістрів також буде два. А також два цикли – внутрішній і зовнішній.

Наприклад, нехай є якась матриця Matr[M][N] і в пам'яті вона розміщується по рядкам: спочатку N елементів першого рядка, потім N елементів другого рядка і т.д. до рядка M. Довжину елемента цієї матриці також позначим через Larray.

Тоді адреса елемента Matr[i,j] буде дорівнювати Matr+N*i*Larray+j, де i=0,…, M-1; j=0,…,N-1. Виділимо в Асемблері для зберігання величини N*i*Larray регістр BX, а для j – регістр SI (або DI). Тоді Matr[BX] буде означати початкову адресу рядка i, а Matr[BX][SI] (Matr[BX+SI] або Matr+[BX+SI] – ці три записи рівнозначні) – адреса елемента j в цьому рядку, тобто Matr[i,j] → Matr[BX][SI].

Таблиця 7.4

Основні характеристики цілочисельної матриці A[10][5] в Асемблері

Байти	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…	97	98	99	100
Стовбці (j)	0		1		2		3		4		0		1		…	3		4	
Індекс-регістр SI	0		+2		+4		+6		+8		0		+2		…	+6		+8	
Рядки (i)	0										1				…	9			
Індекс-регістр (BX)	0										+(10*1)=+10				…	+(10*9)=+90			

Приклад 1: Написати програму для обробки одномірного масиву для початкових даних в знаковому форматі. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів. Тип даних SHOT INT.

Знайти, кількість елементів масиву A={a[i]} задовольняють умові: c<=a[i]<=d.

Розглянемо 2 варіанти:

1. *З використанням команд передачі управління та логічних зсувів.*

Лістинг програми:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#define N 10
int main()
{
   short int a[N];
   short int c = 0, d = 0;
   printf("c<=a[i]<=d\n\n");
   do {
      printf("Enter the values of the range [-32768...32767]:\n");
      printf("c = "); scanf_s("%hd", &c);
      printf("d = "); scanf_s("%hd", &d);
      if (c >= d)
      {
 printf("c can not be greater or equal d! Enter values again.\n\n");
      }
   } while (c >= d);
   int n = N;
   short int res = 0;
   srand(time(0)); // ел-ти масиву при кожній отладці будуть різними
   for (int i = 0; i < N; i++)
   {
      a[i] = rand() % 21-10; // елементи масиву від -10 до 10
      printf("A[%d] = %hd\n", i, a[i]);
   }
```

```
__asm
{
      mov ax, c     // <ax> = c
      mov bx, d     // <bx> = d

      mov ecx, n    // <ecx> = n
      dec ecx       // зменшуємо вміст регістру <ecx> на 1
                (перевірку масиву робимо з останнього елемента)
      mov dx, 0     // <dx> = 0 (В регістрі dx буде підрахову
           ватись кількість елементів, які відповідають умові)

      cycle :       // цикл cycle
        shl ecx, 1   // зсув вліво на 1 розряд
     (Перевірка масиву з останнього елемента, з права на ліво)
        mov si, a[ecx]   // <si> = a[ecx] (в регістр "індекс
       джерела" вносимо значення ел-та масиву яке розглядаємо)
        cmp si, ax  // порівнюємо значення регістрів <si> і <ax>
        jl exit1    // якщо менше - переходимо до циклу exit1
        cmp si, bx  // порівнюємо значення регістрів <si> і <bx>
        jg exit1    // якщо більше - переходимо до циклу exit1
        inc dx      // збільшуємо значення в регістрі <dx> на 1

      exit1 :       // цикл exit1
        shr ecx, 1  // зсув вправо на 1 розряд
        dec ecx     // зменшуємо значення в регістрі <ecx> на 1
                  (номер елемента матриці який ми переглядаємо)
        cmp ecx, 0  // порівнюємо значення регістра <ecx> з 0
        jnl cycle   // поки не менше - переходимо до циклу cycle
        mov res, dx       // res = <dx>
}
if (res > 32767 || res < -32768)
{
      printf("Overflow!\n");
}
else
{
      printf("Result = %hd\n", res);
}
_getch();
return 0;
}
```

Тестові перевірки роботи програми (рис.7.1)

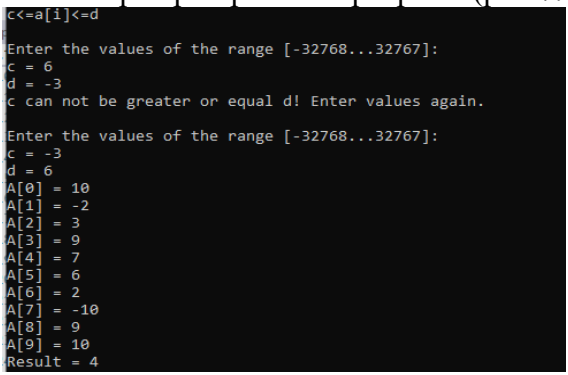

Рисунок 7.1 – Перевірка роботи програми

Значення регістрів при покроковому виконанні

Команда	**Значення регістра**				**EFLAGS/FLAGS (CF, OF)**
	ax	**bx**	**ecx**	**dx**	
mov ax, c	-3	н/в	н/в	н/в	–
mov bx, d	н/в	6	н/в	н/в	–
mov ecx, n	н/в	н/в	10	н/в	–
mov dx, 0	н/в	н/в	н/в	0	–
dec ecx	н/в	н/в	9	н/в	–
cycle:	Мітка циклу cycle (код буде виконуватись в цій мітці, якщо умова – істина)				
shl ecx, 1	Зсув біта операнда вліво на 1 розряд				
mov si, a[ecx]	Заносимо в регістр <si> елемент масиву з відповідним індексом				
cmp si, ax	Порівнюємо елемент масиву з нижньою границею – c				
jl exit1	Якщо менше – перейти до мітки циклу exit1				
cmp si, bx	Порівнюємо елемент масиву з верхньою границею – d				
jg exit1	Якщо більше – перейти до мітки циклу exit1				
inc dx	Збільшуємо значення в регістрі <dx> на 1				
exit1:	Мітка циклу exit1(код буде виконуватись в цій мітці, якщо умова – істина)				
shr ecx, 1	Зсув біта операнда вправо на 1 розряд				
dec ecx	**Зменшуємо значення в регістрі <ecx> на 1**				
cmp ecx, 0	**Порівнюємо значення регістра <ecx> з 0**				
jnl cycle	**Перейти на мітку циклу, якщо не менше**				
mov res, dx	н/в	н/в	н/в	4	–

2. *З використанням команд LOOP, LEA, LODSW*

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#define N 10
int main()
{
   short int a[N];
   short int c = 0, d = 0;
   printf("c<=a[i]<=d\n\n");
   do
   {
  printf("Enter the values of the range [-32768...32767]:\n");
      printf("c = "); scanf_s("%hd", &c);
      printf("d = "); scanf_s("%hd", &d);
      if (c >= d)
      {
printf("c can not be greater or equal d! Enter values again.\n\n");
      }
   }
   while (c >= d);
   int n = N;
   short int res = 0;
   srand(time(0));
   for (int i = 0; i < N; i++)
   {
      a[i] = rand() % 21-10; // елементи масиву від -10 до 10
      printf("A[%d] = %hd\n", i, a[i]);
   }
__asm
   {
      mov bx, c          // <bx> = c
      mov dx, d          // <dx> = d
      mov ecx, n         // <ecx> = n
      lea si, a          // завантаження зміщення масиву в
                  регістр SI(замінює shr ecx, 1 та shl ecx, 1)
      cycle :            // цикл cycle
         lodsw           // завантажуємо слово з пам'яті,
                              на який вказує регістр SI
         cmp ax, bx      // порівняння вмісту регістрві <ax> і <bx>
         jl exit1        // якщо менше - переходимо до циклу exit1
```

```
        cmp ax, dx    // порівняння вмісту регістрві <ax> і <dx>
        jg exit1      // якщо більше - переходимо до циклу exit1
        inc res        // збільшуємо значення res на 1

     exit1 :          // цикл exit1
        LOOP cycle    // зменшує значення в регістрі <ecx> на 1,
порівнює отримане значення з 0 і якщо не менше переходить до cycle
   }
   if (res > 32767 || res < -32768)
   {
      printf("Overflow!\n");
   }
   else
   {
      printf("Result = %hd\n", res);
   }
   _getch();
   return 0;
}
```

Значення регістрів при покроковому виконанні

Команда	**Значення регістра**				**EFLAGS/FLAGS (CF, OF)**
	ax	**bx**	**ecx**	**dx**	
mov bx, c	н/в	-3	н/в	н/в	–
mov dx, d	н/в	н/в	н/в	6	–
mov ecx, n	н/в	н/в	10	н/в	–
lea si, a	Заносимо в регістр <si> адресу першого елемента масиву				
cycle:	Мітка циклу cycle (код буде виконуватись в цій мітці, якщо умова – істина)				
lodsw	Завантажуємо слово з пам'яті, на який вказує регістр SI				
cmp ax, bx	Порівнюємо елемент масиву з нижньою границею – c				
jl exit1	Якщо менше – перейти до мітки циклу exit1				
cmp ax, dx	Порівнюємо елемент масиву з верхньою границею – d				
jg exit1	Якщо більше – перейти до мітки циклу exit1				
inc res	Збільшуємо значення res на 1				
exit1:	Мітка циклу exit1(код буде виконуватись в цій мітці, якщо умова – істина)				
LOOP cycle	**Зменшує значення в регістрі <ecx> на 1**				
	Порівнює значення регістра <ecx> з 0				
	Передає управління на мітку цикл				

Приклад 2: Написати програму для обробки двовимірного масиву для початкових даних в знаковому форматі. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Знайти, скільки елементів масиву A={a[i][j]} задовольняють умові: c<=a[i][j]<=d. Тип даних SHOT INT.

Лістинг програми:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#define N 5
int main()
{
   short int a[N][N];
   short int c = 0, d = 0;
   do {
      printf("c<=a[i]<=d\n\n");
      printf("Enter the values c and d:\n");
      printf("c = "); scanf_s("%hd", &c);
      printf("d = "); scanf_s("%hd", &d);
      printf("\n");
      if (c >= d)
      {
 printf("c can not be greater or equal d! Enter values again.\n");
      }
   } while (c >= d);
   int n = N * N;        // загальна кількість елементів масиву
   short int res = 0;
   srand(time(0));
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
      {
            a[i][j] = rand() % 19-10;
            printf("%hd ", a[i][j]);
      }
      printf("\n");
   }
```

```
    __asm
    {
        mov bx, c      // <bx> = c
        mov dx, d      // <dx> = d
        mov ecx, n     // <ecx> = n
        lea si, a      // завантаження зміщення масиву в регістр
                            SI (замінює shr ecx, 1 та shl ecx, 1)

        cycle :       // цикл cycle
            lodsw     // завантажуємо слово з пам'яті,
                                  на який вказує регістр SI
            cmp ax, bx    // порівнюємо з нижньою границею - c
            jl exit1      // якщо менше - перейти до циклу exit1
            cmp ax, dx    // порівнюємо з верхньою границею - d
            jg exit1      // якщо більше - перейти до циклу exit1
            inc res       // збільшити результат на 1

        exit1 :       // цикл exit1
        LOOP cycle    // зменшує значення в регістрі <ecx> на 1, по-
рівнює отримане значення з 0 і якщо не менше переходить до exit1
    }
    if (res > 32767 || res < -32768)
    {
        printf("Overflow!\n");
    }
    else
    {
        printf("\n");
        printf("Result = %hd\n", res);
    }
    _getch();
    return 0;
}
```

Проведемо тестову перевірку роботи програми(рис.7.2)

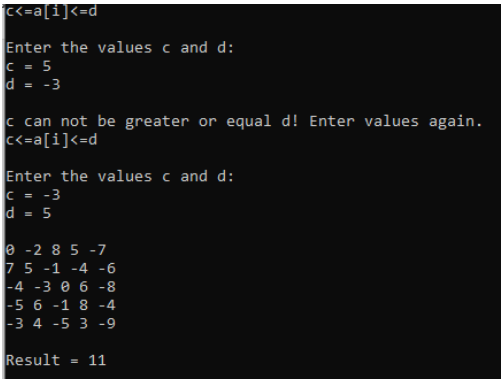

Рисунок 7.2 – Перевірка роботи програми

Значення регістрів при покроковому виконанні

Команда	**EFLAGS/FLAGS (CF, OF)**			
	bx	**dx**	**ecx**	–
mov bx, c	-3	н/в	н/в	–
mov dx, d	н/в	5	н/в	–
mov ecx, n	н/в	н/в	25	–
lea si, a	Заносимо в регістр <si> адресу першого елемента масиву			
cycle:	Мітка циклу cycle (код буде виконуватись в цій мітці, якщо умова – істина)			
lodsw	Завантажуємо слово з пам'яті, на який вказує регістр SI			
cmp ax, bx	Порівнюємо елемент масиву з нижньою границею – c			
jl exit1	Якщо менше – перейти до мітки циклу exit1			
cmp ax, dx	Порівнюємо елемент масиву з верхньою границею – d			
jg exit1	Якщо більше – перейти до мітки циклу exit1			
inc res	Збільшуємо результат res на 1			
exit1:	Мітка циклу exit1 (код буде виконуватись в цій мітці, якщо умова – істина)			
LOOP	Зменшує значення в регістрі <ecx> на 1			
	Порівнює значення регістра <ecx> з 0			
	Поки індекс елементу масиву не менше 0 – перейти до циклу cycle			

Завдання на лабораторну роботу

1. Написати програму для обробки одномірного масиву для початкових даних типу **SHORT INT** (табл.7.5) в знаковому форматі. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Таблиця 7.5

Дані для розрахунку

№ Варіанту	Вираз
1	Знайти суму **min** і **max** елементів масиву **A={a[i]}**.
2	Знайти середнє арифметичне від'ємних елементів масиву **A={a[i]}**.
3	Знайти добуток числа **5** та **max** додатного елемента масиву **A={a[i]}**.
4	Знайти частку від **max** і **min** елементів масиву **A={a[i]}**.
5	Знайти різницю елементів масиву **A={a[i]}**, які задовольняють умові **c<=a[i]<=d**.
6	Знайти різницю елементів масиву **A={a[i]}**, значення яких більше за значення **1**-го елемента.
7	Знайти середнє арифметичне парних елементів масиву **A={a[i]}**.
8	Обчислити суму по модулю всіх від'ємних елементів масиву **A={a[i]}**.
9	Знайти частку від **1**-го і **k**-го елементів масиву **A={a[i]}**.
10	Знайти середнє арифметичне всіх елементів масиву **A={a[i]}**.
11	Знайти частку від додатних **max** і **min** не нульового елементів масиву **A={a[i]}**.
12	Знайти різницю від'ємних елементів масиву **A={a[i]}**.
13	Знайти суму елементів масиву **A={a[i]}**, значення яких більше значення **1**-го елемента.
14	Знайти середнє арифметичне непарних елементів масиву **A={a[i]}**.
15	Знайти добуток числа **3** та **min** не нульового додатного елемента масиву **A={a[i]}**.

Продовження табл.7.5

Дані для розрахунку

16	Знайти суму додатних **max** і **min** елементів масиву **A={a[i]}** після зміни їх знаку на протилежний.
17	Знайти суму всіх елементів масиву **A={a[i]}**, які задовольняють умові **c<=a[i]<=d**.
18	Знайти різницю додатних **max** і **min** елементів масиву **A={a[i]}**.
19	Знайти суму непарних елементів масиву **A={a[i]}**.
20	Знайти добуток числа **2** та **max** від'ємного елемента масиву **A={a[i]}**.
21	Знайти різницю **max** і **min** елементів масиву **A={a[i]}**.
22	Знайти добуток числа **2** і суми парних елементів масиву **A={a[i]}**.
23	Знайти добуток **k**-го і 5 елементів масиву **A={a[i]}**.
24	Знайти різницю від'ємних **max** і **min** елементів масиву **A={a[i]}**.
25	Знайти добуток першого і **k**-го елементів масиву **A={a[i]}**.
26	Знайти добуток числа **4** та **min** від'ємного елемента масиву **A={a[i]}**.
27	Знайти суму всіх додатних елементів масиву **A={a[i]}**.
28	Знайти добуток **min** і **max** елементів масиву **A={a[i]}**.
29	Знайти суму додатних елементів масиву **A={a[i]}**.
30	Знайти суму елементів масиву **A={a[i]}**, значення яких менше значення першого елемента.
31	Знайти суму парних елементів масиву **A={a[i]}**.
32	Знайти добуток числа **5** та **max** додатного елемента масиву **A={a[i]}**.
33	Знайти суму додатних **max** і **min** елементів масиву **A={a[i]}**.
34	Обчислити добуток суму по модулю всіх від'ємних елементів масиву **A={a[i]}** і числа **5**.
35	Знайти частку від суми непарних елементів масиву **A={a[i]}** і числа **10**.

2. Написати програму для обробки двовимірного масиву для початкових даних типу **SHORT INT** (табл.7.6) в знаковому форматі. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Таблиця 7.6

Дані для розрахунку

№ Варіанту	Вираз
1	Знайти суму **min** і **max** елементів масиву **A={a[i][j]}**.
2	Знайти середнє арифметичне від'ємних елементів масиву **A={a[i][j]}**.
3	Знайти добуток числа **5** та **max** додатного елемента масиву **A={a[i][j]}**.
4	Знайти частку від **max** і **min** елементів масиву **A={a[i][j]}**.
5	Знайти різницю елементів массиву **A={a[i][j]}**, які задовольняють умові **c<=a[i][j]<=d**.
6	Знайти різницю елементів масиву **A={a[i][j]}**, значення яких більше за значення **1**-го елемента.
7	Знайти середнє арифметичне парних елементів масиву **A={a[i][j]}**.
8	Обчислити суму по модулю всіх від'ємних елементів масиву **A={a[i][j]}**.
9	Знайти частку від **1**-го і **k**-го елементів масиву **A={a[i][j]}**.
10	Знайти середнє арифметичне всіх елементів масиву **A={a[i][j]}**.
11	Знайти частку від додатних **max** і **min** не нульового елементів масиву **A={a[i][j]}**.
12	Знайти різницю від'ємних елементів масиву **A={a[i][j]}**.
13	Знайти суму елементів масиву **A={a[i][j]}**, значення яких більше значення **1**-го елемента.
14	Знайти середнє арифметичне непарних елементів масиву **A={a[i][j]}**.
15	Знайти добуток числа **3** та **min** не нульового додатного елемента масиву **A={a[i][j]}**.

Продовження табл.7.6

Дані для розрахунку

16	Знайти суму додатних **max** і **min** елементів масиву **A={a[i][j]}** після зміни їх знаку на протилежний.
17	Знайти суму всіх елементів массиву **A={a[i][j]}**, які задовольняють умові **c<=a[i][j]<=d**.
18	Знайти різницю додатних **max** і **min** елементів масиву **A={a[i][j]}**.
19	Знайти суму непарних елементів масиву **A={a[i][j]}**.
20	Знайти добуток числа **2** та **max** від'ємного елемента масиву **A={a[i][j]}**.
21	Знайти різницю **max** і **min** елементів масиву **A={a[i][j]}**.
22	Знайти добуток числа **2** і суми парних елементів масиву **A={a[i][j]}**
23	Знайти добуток **k**-го і **5** елементів масиву **A={a[i][j]}**.
24	Знайти різницю від'ємних **max** і **min** елементів масиву **A={a[i][j]}**.
25	Знайти добуток **1**-го і **k**-го елементів масиву **A={a[i][j]}**.
26	Знайти добуток числа **4** та мінімального від'ємного елемента масиву **A={a[i][j]}**.
27	Знайти суму всіх додатних елементів масиву **A={a[i][j]}**.
28	Знайти добуток **min** і **max** елементів масиву **A={a[i][j]}**.
29	Знайти суму додатних елементів масиву **A={a[i][j]}**.
30	Знайти суму елементів масиву **A={a[i][j]}**, значення яких менше значення **1**-го елемента.
31	Знайти суму парних елементів масиву **A={a[i][j]}**.
32	Знайти добуток числа **5** та **max** додатного елемента масиву **A={a[i]}**.
33	Знайти суму додатних **max** і **min** елементів масиву **A={a[i][j]}**.
34	Обчислити добуток суму по модулю всіх від'ємних елементів масиву **A={a[i][j]}** і числа **5**.
35	Знайти частку від суми непарних елементів масиву **A={a[i][j]}** і числа **10**.

Контрольні питання

1. Організація циклів на мові Ассемблер з використанням команд умовних та безумовних переходів.
2. Організація циклів на мові Ассемблер з використанням команд LOOPx.
3. Команда організації циклів LOOP, LOOPE, LOOPZ та LOOPNE. Призначення та синтаксис.
4. Особливості застосуванняк команди JCXZ при організації циклів на мові Ассемблер.
5. Типова послідовність команд для організації циклу в мові Ассемблер.
6. Команди логічних зсувів SHL і SHR. Призначення та синтаксис.
7. Команди арифметичних зсувів SAR і SAL. Призначення та синтаксис.
8. Команди циклічних зсувів ROL, ROR, RCL і RCR. Призначення та синтаксис.
9. Команди завантаження даних LODS, LODSB, LODSW, LODSD. Призначення та синтаксис.
10. Команда завантаження ефективної адреси джерела LEA. Призначення та синтаксис.