# ЛАБОРАТОРНА РОБОТА № 6.
# Використання макрокоманд та процедур.
# Ввід даних з клавіатури та вивід результату на екран.

**Мета:** набути навиків написання макрокоманд та процедур на Асемблері, освоїти способи передачі параметрів. Реалізувати ввід даних з клавіатури та вивід даних на екран.

## Процедури та оператор CALL

Процедура – це частина програми, яка може бути описана в довільному місці і містити дії над довільними даними. Процедура починається директивою PROC та завершується директивою ENDP. Кодовий сегмент може містити будь-яку кількість процедур, що розділяються директивами PROC і ENDP. Виклик процедури здійснюється командою CALL. Для повернення з процедури використовується команда RET.

Опис процедури мовою асемблера виглядає так:

```
<ім'я процедури> PROC
 <тіло процедури>
<ім'я процедури> ENDP
```

Незважаючи на те, що після імені процедури не ставиться двокрапка, це ім'я є міткою, яка позначає першу команду процедури.

У мові асемблера імена та мітки, описані в процедурі, не локалізуються всередині неї, тому вони мають бути унікальними.

Розміщувати процедуру в програмі мовою асемблера слід таким чином, щоб команди процедури виконувались не самі по собі, а лише тоді, коли відбувається звернення до процедури. Зазвичай процедури розміщують або в кінці секції коду після виклику функції ExitProcess, або на початку секції коду, відразу після директиви .code.

Виклик процедури – це, власне, передача управління на першу команду процедури. Для передачі керування можна використовувати команду безумовного переходу на мітку, яка є ім'ям процедури. Можна навіть не використовувати директиви proc та endp, а написати звичайну мітку з двокрапкою після виклику функції ExitProcess.

З поверненням із процедури справа складніша. Справа в тому, що звертатися до процедури можна з різних місць основної програми, а тому повернення з процедури має здійснюватися в різні місця. Сама процедура не знає, куди треба повернути управління, зате знає основна програма. Тому при зверненні до процедури основна програма має повідомити адресу повернення, тобто. адресу тієї команди, яку процедура має зробити перехід по закінченні своєї роботи. Оскільки при різних зверненнях до процедури будуть вказуватися різні адреси повернення, то повернення управління здійснюватиметься в різні місця програми. Система команд мови асемблера включає спеціальні команди для виклику процедури та повернення з процедури.

```
CALL <ім'я процедури> ; Виклик процедури
RET ; Повернення з процедури
```

Команда CALL записує адресу наступної команди у стек і здійснює перехід на першу команду зазначеної процедури.

Команда RET зчитує з вершини стека адресу повернення та виконує перехід по ній.

```
.code
program:
   call Procedure

   push 0
   call ExitProcess
```

```
Procedure proc
   ret
Procedure endp

end program
```

**Існують кілька способів передачі параметрів у процедуру.**

Параметри можна надсилати через регістри.

Якщо процедура отримує невелику кількість параметрів, ідеальним місцем їх передачі виявляються регістри. Існують угоди про виклики, що передбачають передачу параметрів через регістри ECX та EDX. Цей метод найшвидший, але він зручний лише для процедур із невеликою кількістю параметрів.

Параметри можна передавати через глобальні змінні.

Параметри процедури можна записати до глобальних змінних, до яких потім буде звертатися процедура. Однак цей метод є неефективним, і його використання може призвести до того, що рекурсія та повторний вхід у процедуру стануть неможливими.

Параметри можна надсилати в блоці параметрів.

Блок параметрів – це ділянка пам'яті, що містить параметри і розташована зазвичай в сегменті даних. Процедура отримує адресу початку цього блоку за допомогою будь-якого методу передачі параметрів (у регістрі, змінній, стеку, коді або навіть в іншому блоці параметрів).

Параметри можна надсилати через стек.

Передача параметрів через стек – найпоширеніший спосіб. Саме його використовують мови високого рівня. Параметри розміщуються у стек безпосередньо перед викликом процедури.

При уважному аналізі цього методу передачі параметрів виникає відразу два питання: хто повинен видаляти параметри зі стеку, процедура чи програма, що викликає її, і в якому порядку поміщати параметри в стек. В обох випадках виявляється, що обидва варіанти мають свої «за» та «проти». Якщо стек звільняє процедура, то код програми виходить меншим, а якщо за звільнення стека від параметрів відповідає програма, що викликає, то стає можливим викликати кілька функцій з одними і тими ж параметрами просто послідовними командами CALL. Перший спосіб, суворіший, використовується при реалізації процедур у мові Паскаль, а другий, що дає більше можливостей для оптимізації, - в мові C++.

Основна угода про виклики мови Паскаль передбачає, що параметри поміщають у стек у прямому порядку. Угоди про виклики мови C++, у тому числі одна з основних угод про виклики Windows stdcall, припускають, що параметри поміщають у стек у зворотному порядку. Це робить можливим реалізацію функцій зі змінним числом параметрів (як, наприклад, printf). У цьому перший параметр визначає кількість інших параметрів.

```
push <параметр>
...
push <параметр1>
call Procedure
```

У наведеному вище ділянці коду в стек кладуться кілька параметрів і потім викликається процедура. Слід пам'ятати, що команда CALL також поміщає адресу повернення в стек.

Адреса повернення знаходиться у стеку поверх параметрів. Однак оскільки в рамках своєї ділянки стека процедура може звертатися без обмежень до будь-якої ділянки пам'яті, немає необхідності перекладати кудись адресу повернення, а потім повертати її назад у стек. Для звернення до першого параметру використовують адресу [ESP + 4] (додаємо 4, тому що на архітектурі Win32 адреса має розмір 32 біти), для звернення до другого параметра – адреса [ESP + 8] і т.д.

Після завершення роботи процедури необхідно звільнити стек. Якщо використовується угода про виклики stdcall (або будь-яке інша, що передбачає що стек звільняється процедурою), то в команді RET слід вказати сумарний розмір у байтах усіх параметрів процедури. Тоді команда RET після отримання адреси повернення додасть до регістру ESP вказане значення, звільнивши таким чином стек. Якщо ж використовується угода про виклики cdecl (або будь-яке інша, що передбачає, що стек звільняється програмою, що викликає), то після команди CALL слід помістити команду, яка додасть до регістру ESP потрібне значення.

Для передачі результату процедури використовується регістр EAX. Цей спосіб використовується у програмах мовою асемблера, а й у програмах мовою C++. Об'єкти, що мають розмір 8 байт, можуть передаватися через регістрову пару EDX:EAX. Якщо ці способи не підходять, то слід передати як параметр адресу комірки пам'яті, куди буде записаний результат.

Майже будь-які дії в мові асемблера вимагають використання регістрів. Однак регістрів дуже мало і навіть у невеликій програмі неможливо буде розділити регістри між частинами програми, тобто. домовитись, що основна програма використовує, наприклад, регістри EAX, ECX, EBP, ESP, а процедура – регістри EBX, EDX, ESI, EDI. У принципі зробити так можна, але сенсу в цьому немає, програмувати буде вкрай незручно, доведеться переміщати дані з регістрів в оперативну пам'ять і назад, що уповільнить виконання програми. Крім того, існують правила, які змінити не можна – у регістрі ESP зберігається адреса вершини стека, а команди множення та ділення завжди використовують регістри EAX та EDX. Тому виходить, що основна програма і процедура змушені використовувати одні і ті ж регістри, причому обчислення в основній програмі перериваються для того, щоб виконати обчислення процедури. Таким чином, щоб основна програма могла продовжити обчислення, процедура повинна при виході відновити значення регістрів, які були до початку виконання процедури. Звичайно, для цього процедурі доведеться попередньо зберегти значення регістрів. Все вищесказане стосується також випадку, коли одна процедура викликає іншу процедуру.

Особливо уважно слід ставитись до регістрів ESI, EDI, EBP та EBX. Windows використовує ці регістри для своїх цілей і не очікує, що ви зміните їх значення.

Якщо ви пишите всю програму повністю, то, в принципі, можете досягти того, що після виклику процедури в основній програмі необхідні регістри будуть правильно проініціалізовані. Якщо ж ви пишете окремі процедури, які потім будуть використовуватися в іншій програмі, то жодних гарантій немає, і збереження та відновлення регістрів стає життєво необхідною операцією.

Де можна зберегти значення регістрів? Звісно ж, у стеку. Можна зберегти регістри по одному за допомогою команди PUSH, або всі відразу за допомогою команди PUSHAD. У першому випадку наприкінці процедури потрібно буде відновити значення збережених регістрів за допомогою команди POP у зворотному порядку. У другому випадку для відновлення значень регістрів використовується команда POPAD.

При збереженні регістрів покажчик стека зміниться на деяке значення, що залежить від кількості збережених регістрів. Це потрібно буде враховувати при обчисленні адрес параметрів процедури, що передаються через стек.

## Вивід на екран.

Для виводу даних на екран можна скористатись API функцією `WriteConsoleA`, яка записує рядок символів у буфер екрану консолі, починаючи з поточного положення курсору.

```
BOOL WINAPI WriteConsole(
  _In_             HANDLE  hConsoleOutput,
  _In_       const VOID    *lpBuffer,
  _In_             DWORD   nNumberOfCharsToWrite,
  _Out_opt_        LPDWORD lpNumberOfCharsWritten,
  _Reserved_       LPVOID  lpReserved
);
```

Параметри:

hConsoleOutput - Дескриптор буфера екрану консолі.

lpBuffer - Вказівник на буфер, що містить символи для запису в буфер екрану консолі. Передбачається, що це є масив char.

nNumberOfCharsToWrite - Кількість символів, що записуються.

lpNumberOfCharsWritten - Вказівник на змінну, яка отримує кількість фактично записаних символів.

lpReserved - Зарезервовано; має бути значення NULL.

Дескриптор буфера екрану консолі можна отримати API функцією `GetStdHandle`.

```
HANDLE WINAPI GetStdHandle(
  _In_ DWORD nStdHandle
);
```

Параметр nStdHandle - Стандартний пристрій. Цей параметр може приймати одне з наведених нижче значень:

STD_INPUT_HANDLE((DWORD)-10) Стандартний пристрій введення. Спочатку це вхідний буфер консолі.

STD_OUTPUT_HANDLE((DWORD)-11) Стандартний вихідний пристрій. Спочатку це активний буфер екрану консолі.

STD_ERROR_HANDLE((DWORD)-12) Визначення стандартних помилок. Спочатку це активний буфер екрану консолі.

Для переведення числа у рядок можна скористатись функцією `wsprintfA`, яка записує відформатовані дані у вказаний буфер. Будь-які аргументи перетворюються та копіюються у вихідний буфер відповідно до відповідної специфікації формату в рядку форматування. Функція додає кінцевий нульовий символ до символів, які вона записує, але повернуте значення не включає кінцевий нульовий символ у кількість символів.

.

```
int WINAPIV wsprintfA(
  [out] LPSTR  unnamedParam1,
  [in]  LPCSTR unnamedParam2,
        ...
);
```

## Ввід з клавіатури

Для вводу даних з клавіатури можна скористатись API функцією `ReadConsoleA`, яка зчитує вхідні символи з буфера введення консолі та видаляє їх з буфера.

```
BOOL WINAPI ReadConsole(
  _In_     HANDLE  hConsoleInput,
  _Out_    LPVOID  lpBuffer,
  _In_     DWORD   nNumberOfCharsToRead,
  _Out_    LPDWORD lpNumberOfCharsRead,
  _In_opt_ LPVOID  pInputControl
);
```

Параметри схожі на параметри функції `WriteConsoleA`, `lpBuffer` - вказівник на буфер, який отримує дані, зчитані з вхідного буфера консолі. Якщо зчитуємо число, то функція збереже його у вигляді рядка. Для перетворення рядка на ціле число можна скористатись функцією `crt_atoi`.

---

## Контрольні запитання:

1. Що в Асемблері роблять команди CALL і RET?
2. Як описати процедуру?
3. Які способи передачі параметрів у процедуру можна використовувати?
4. Як правильно передавати параметри через стек?
5. Де процедура залишає результат своєї роботи?

---

Література:

1.Р.Джордейн.Справочник програмиста персональных компъютеров типа IBM PC XT и AT. - М."Финансы и статистика",1992,стор.13-31.
2.Л.О.Березко,В.В.Троценко. Особливості програмування в турбо-асемблері. -Київ,НМК ВО,1992.
3.Л.Дао. Программирование микропроцессора 8088.Пер.с англ.-М."Мир",1988.
4.П.Абель.Язык ассемблера для IBM PC и программирования. Пер. з англ.-М.,"Высшая школа",1992.

## ЗАВДАННЯ:

1. Створити *.exe програму, яка реалізовує обчислення, задані варіантом над даними, введеними з клавіатури і результат виводить на екран.

Програма повинна складатися з трьох основних підпрограм:
процедура вводу – забезпечує ввід даних з клавіатури;
процедура безпосередніх обчислень – здійснює всі необхідні арифметичні дії;
процедура виводу – забезпечує вивід на екран результату.

Передача параметрів може здійснюватися довільним чином. Кожна з перерахованих процедур може містити довільну кількість додаткових підпрограм.

2. Переконатися у правильності роботи кожної процедури зокрема та програми загалом.
3. Скласти звіт про виконану роботу з приведенням тексту програми та коментарів до неї.
4. Дати відповідь на контрольні запитання.

ВАРІАНТИ ЗАВДАННЯ – ДИВ. ЛАБОРАТОРНУ РОБОТУ №5.

## Приклад виконання.

*Завдання:* Написати програму, яка знаходить суму чисел заданого масиву і результат записує в пам’ять.

Згідно з завданням, програма має складатись з трьох процедур. Відповідно, структура програми буде мати такий вигляд.

```
.data ;опис необхідних даних

.code

start:

call Input ;виклик процедури вводу
call Calculation ;виклик основної процедури
call Output ;виклик процедури виводу

invoke ExitProcess, 0

Input proc ;тіло процедури вводу
ret
Input endp

Calculation proc ;тіло основної процедури
ret
Calculation endp

Output proc ;тіло процедури виводу
ret
Output endp

end start
```

Програма, що реалізує поставлене завдання, приведена в лабораторній роботі №5. Тепер, слід оформити цю програму у вигляді процедури та написати відповідні підпрограми, щоб забезпечити коректний ввід і вивід результату на екран. Вхідні дані для обчислень – масив X і кількість елементів

N будемо передавати через стек. Результат обчислень процедура залишить у регістрі EAX. Вхідні параметри для процедур вводу і виводу даних будемо передавати через глобальні змінні.

Підпрограма вводу. – INPUT, обчислень – CALCULATION, виводу – OUTPUT.

Нижче приведено текст програми, що реалізовує завдання.

```
.686
.model flat, stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\msvcrt.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\msvcrt.lib

.data

X dd 5 dup(0)                   ;масив
N equ ($-X) / type X      ;розмір масиву
Res dd ?                        ;результат

hConsoleInput dd 0        ;дескриптор консолі для вводу
hConsoleOutput dd 0       ;дескриптор консолі для виводу
NumberOfChars dd 0        ;кількість введених/виведених символів
ReadBuf db 32 dup(0)      ;будеф для введених символів
Message db 'Input elements of array:', 10, 13
NumberOfChToWMessage dd $-Message
Format db 'Result = %d', 0
Result db     32 dup (0), 10, 13
NumberOfChToWResult dd $-Result

.code
start:
call Input

push dword ptr N    ;другий параметр - кількість елементів у масиві
push offset X             ;перший параметр - адреса масиву
call Calculation
mov Res, eax

call Output

invoke ExitProcess, 0

;процедура вводу даних - параметри передаються через глобальні змінні
Input proc
       invoke GetStdHandle, -11
       mov hConsoleOutput, eax
       invoke WriteConsoleA, hConsoleOutput, addr Message, NumberOfChToWMessage, addr
NumberOfChars, 0
       invoke GetStdHandle, -10
       mov hConsoleInput, eax
       mov ecx, dword ptr N
       lea ebx, X
       mov edi, 0
       L_input:
              push ecx
              invoke ReadConsoleA, hConsoleInput, addr ReadBuf, 32, addr NumberOfChars, 0
              invoke crt_atoi, addr ReadBuf
              pop ecx
              mov [ebx][edi], eax
              add edi, 4
              loop L_input
```

```
	ret
Input endp

;процедура обчислення суми в циклі - параметри передаються через стек, результат в регістрі eax
Calculation proc
	push ebp
	mov ebp, esp
	mov ebx, [ebp + 8] ;перший параметр - адреса масиву
	mov ecx, [ebp + 12];другий параметр - кількість елементів у масиві
	mov edx, 0
	L:
		mov eax, [ebx + ecx * type X - type X]
		add edx, eax
		loop L
	mov eax, edx	;результат в регістрі eax
	pop ebp
	ret 8
Calculation endp

;процедура виводу даних - параметри передаються через глобальні змінні
Output proc
	invoke wsprintfA, addr Result, addr Format, Res
	invoke GetStdHandle, -11
	mov hConsoleOutput, eax
	invoke WriteConsoleA, hConsoleOutput, addr Result, NumberOfChToWResult, addr
NumberOfChars, 0
	ret
Output endp

end start
```