Лабораторна робота № 8
ПРОЦЕДУРИ

Мета роботи: ознайомитися з основами роботи з процедурами, видами адресацій процедур (NEAR, FAR); навчитися використовувати локальні змінні.

Теоретичні відомості

Процедура (підпрограма) – це основна функціональна одиниця декомпозиції (поділу на кілька частин) деякої задачі. Процедура є групою команд для розв'язання конкретної підзадачі і має засоби отримання керування з точки виклику задачі більш високого пріоритету й повернення керування в цю точку. У найпростішому випадку програма може складатися з однієї процедури. Процедуру можна визначити і як правильним чином оформлену сукупність команд, яка, будучи одноразово описаною, за необхідності може бути викликана в будь-якому місці програми.

Функція – процедура, здатна повертати деяке значення.

Процедури можна активізувати в будь-якому місці програми. Процедурам може буто передано деякі аргументи, що дає змогу, маючи одну копію коду в пам'яті змінювати її для кожного конкретного випадку використання, підставляючи необхідні значення аргументів.

Для опису послідовності команд у вигляді процедури в мові Асемблер використовуються дві директиви: `PROC` і `ENDP`.

Синтаксис опису процедури:

```
Ім'я Процедури PROC
; тіло процедури
Ім'я Процедури  ENDP
```

Процедура може розміщуватися в будь-якому місці програми, але так, щоб на неї випадковим чином не потрапило керування. Якщо процедуру просто вставити в загальний потік команд, то мікропроцесор буде сприймати команди процедури як частину цього потоку. Можна застосувати такі варіанти розміщення процедури в програмі:

− на початку (до першої виконуваної команди);

− у кінці (після команди, яка повертає керування операційній системі);

− проміжний варіант – тіло процедури розташовується всередині іншої процедури або основної програми;

− в іншому модулі .

При розміщенні процедури на початку сегмента коду

передбачається, що послідовність команд, обмежена парою директив PROC і ENDP, буде розміщено до мітки, що позначає першу команду, з якої починається виконання програми. Цю мітку необхідно вказати як параметр директиви END, що позначає кінець програми:

```
...
.code
myproc proc
ret
myproc endp
start proc
call myproc
...
start endp
end start
```

У цьому фрагменті після завантаження програми в пам'ять керування буде передано першій команді процедури з ім'ям start.

Оголошення імені процедури в програмі є рівнозначним оголошенню мітки, тому директиву PROC в окремому випадку можна розглядати як форму визначення мітки в програмі.

При розміщенні процедури в кінці програми передбачається, що послідовність команд, обмежену директивами PROC і ENDP, буде розміщено після команди, яка повертає керування операційній системі:

```
...
.code
start proc
call myproc
...
start endp
myproc proc
ret
myproc endp
end start
```

При проміжному варіанті передбачається розміщення тіла процедури всередині іншої процедури або основної програми. У цьому випадку потрібно передбачити обхід тіла процедури, обмежену директивами PROC і ENDP, з допомогою команди безумовного переходу jmp:

```
...
```

```
.code
start proc
jmp ml
myproc proc
ret
myproc endp
ml:
...
start endp
end start
```

При останньому варіанті розміщення описів процедур (в окремому модулі) припускається, що часто використовувані процедури виносяться в окремий файл. Файл з процедурами необхідно оформити як звичайний вихідний файл і підданий трансляції для отримання об'єктного коду. Згодом цей об'єктний файл на етапі компонування об'єднується з файлом, у якому ці процедури використовуються. При цьому способі передбачається наявність у вихідному тексті програми ще деяких елементів, пов'язаних з особливостями реалізації концепції модульного програмування мовою Асемблер. Варіант розташування процедур в окремому модулі використовується також при будуванні Windows-додатків на основі виклику API-функцій.

Виклик процедури і повернення з процедури

Виклик процедури – це по суті передання керування на першу команду процедури. Для передання керування можна використовувати команду безумовного переходу на мітку, що є ім'ям процедури. Можна навіть не використовувати директиви `PROC` і `ENDP`, а написати звичайну мітку з двокрапкою після виклику функції `ExitProcess`.

Повернення з процедури – дещо складніший процес, оскільки звертатися до процедури можна з різних місць основної програми, а тому й повернення з процедури має здійснюватися в різні місця. Сама процедура не знає, куди треба повернути керування, зате це знає основна програма. Тому при зверненні до процедури основна програма має повідомити їй адресу повернення, тобто адресу тієї команди, на яку процедура має зробити перехід після закінчення своєї роботи. Оскільки при різних зверненнях до процедури будуть указуватися різні адреси повернення, то й повернення керування буде здійснюватися в різні місця програми. Адресу повернення прийнято передавати через стек.

Приклад:

```
.686
```

```
.model flat, stdcall
option casemap: none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib

.code
program:
 push L
 jmp Procedure
L: nop

 push 0
 call ExitProcess

Procedure:
 pop eax
 jmp eax
end program
```

Однак так зазвичай не роблять – система команд мови Асемблер містить спеціальні команди для виклику процедури й повернення з процедури.

```
CALL <ім'я процедури> ; виклик процедури
RET                   ; повернення з процедури
```

Команда `CALL` записує адресу наступної команди в стек і здійснює перехід на першу команду зазначеної процедури. Команда `RET` зчитує з вершини стека адресу й виконує перехід по ньому.

```
.686
.model flat, stdcall
option casemap: none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib

.code
program:
 call Procedure

 push 0
 call ExitProcess
```

```
Procedure proc
 ret
Procedure endp

end program
```

Передання параметрів процедури

Існують кілька способів передання параметрів у процедуру.

1. Параметри можна передавати через регістри.

Якщо процедура отримує невелику кількість параметрів, то ідеальним місцем для їх передання будуть регістри. Існують угоди про виклики, у яких передбачається передання параметрів через регістри `ECX` і `EDX`. Цей метод є найшвидшим, але його зручно застосовувати тільки для процедур з невеликою кількістю параметрів.

2. Параметри можна передавати в глобальних змінних.

Параметри процедури можна записати в глобальні змінні, до яких потім буде звертатися процедура. Однак цей метод є неефективним, і його використання може призвести до того, що рекурсія і повторне входження стануть неможливими.

3. Параметри можна передавати в блоці параметрів.

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

4. Параметри можна передавати через стек.

Передання параметрів через стек – найбільш поширених спосіб. Саме його використовують мови високого рівня, такі, як C ++. Параметри поміщають у стек безпосередньо перед викликом процедури.

При уважному аналізі цього методу передання параметрів постають відразу два питання: що буде видаляти параметри зі стеку – процедура чи програма, яка її викликає, і в якому порядку поміщати параметри в стек. В обох випадках виявляється, що обидва варіанти мають свої переваги й недоліки. Якщо процедура звільняє стек, то код програми буде меншим, а якщо за звільнення стеку від параметрів відповідає програма, то стає можливим викликати кілька функцій з одними й тими самими параметрами просто послідовними командами `CALL`. Перший спосіб є більш строгим і використовується при реалізації процедур у мові Паскаль, а другий, що дає більше можливостей для оптимізації, – у мові C ++.

В основній угоді про виклики мови Паскаль передбачається, що параметри поміщають у стек у прямому порядку. В угодах про виклики мови C ++, у тому числі в одній із основних угод про виклики ОС Windows `stdcall`, припускається, що параметри поміщають у стек у зворотному

порядку. Це уможливлює реалізацію функцій зі змінною кількістю параметрів (як, наприклад, `printf`). При цьому перший параметр визначає кількість інших параметрів:

```
push <параметр n>
...
push <параметр 1>
call Procedure
```

У наведеній вище ділянці коду в стек кладуться кілька параметрів і потім викликається процедура. Слід пам'ятати, що команда *CALL* також кладе в стек адресу повернення. Таким чином, перед виконанням першої команди процедури стек буде мати вигляд, показаний на рис. 8.1.

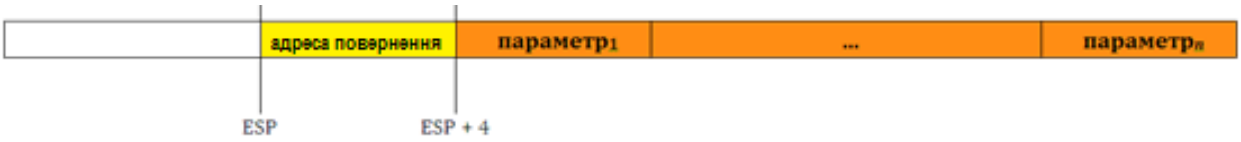

Рис. 8.1. Вміст стеку перед виконанням першої команди процедури

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

Після завершення роботи процедури необхідно звільнити стек. Якщо використовується угода про виклики `stdcall` (або будь-яка інша, коли припускається, що стек звільняється процедурою), то в команді `RET` слід указати сумарний розмір у байтах усіх параметрів процедури. Тоді команда `RET` після вилучення адреси повернення додасть до регістра `ESP` указане значення, звільнивши, таким чином, стек. Якщо ж використовується угода про виклики `cdecl` (або будь-яка інша, коли припускається, що стек звільняється програмою), то після команди `CALL` слід помістити команду, яка додасть до регістра `ESP` необхідне значення.

```
;Передання параметрів і повернення з процедури
;з використанням угоди про виклики stdcall
```

```
    .686
    .model flat, stdcall
    option casemap: none

    include \masm32\include\windows.inc
    include \masm32\include\kernel32.inc
    includelib \masm32\lib\kernel32.lib

    .data
     x dd 0
     y dd 4

    .code
    program:
     push y       ;Кладемо в стек два параметри розміром
по                ;чотири байти
     push x
     call Procedure

     push 0
     call ExitProcess

    Procedure proc
     ret 8        ;У команді повернення вказуємо, що треба
                  ;звільнити вісім байтів стеку
    Procedure endp

    end program
```

;Передання параметрів і повернення з процедури
;з використанням угоди про виклики *cdecl*

```
    .686
    .model flat, c
    option casemap: none

    include \masm32\include\windows.inc
    include \masm32\include\kernel32.inc
    includelib \masm32\lib\kernel32.lib

    .data
     x dd 0
     y dd 4

    .code
    program:
```

```
    push y      ;Кладемо в стек два параметри розміром
по
                ;чотири байти
    push x
    call Procedure
    add esp, 8      ; Звільняємо вісім байтів стеку

    push 0
    call ExitProcess

   Procedure proc
    ret         ;Використовуємо  команду  повернення
                ;без параметрів
   Procedure endp

   end program
```

5. Параметри можна передавати в потоці коду.

У цьому незвичайному методі дані, що передаються процедурі, розміщуються прямо в коді програми, відразу після команди `CALL`. Щоб прочитати параметр, процедура повинна використовувати його адресу, яка автоматично передається в стеку як адреса повернення з процедури. Зрозуміло, процедура повинна буде змінити адресу повернення на перший байт після передання параметрів перед виконанням команди `RET`:

```
   .686
   .model flat, stdcall
   option casemap: none

   include \masm32\include\windows.inc
   include \masm32\include\kernel32.inc
   includelib \masm32\lib\kernel32.lib

   .code
   program:
    call Procedure  ;Команда CALL кладе в стек адресу
                    ;наступної команди
    db 'string',0   ;У  цьому  випадку  –  адресу
                    ;початку рядка

    push 0
    call ExitProcess
   Procedure proc
```

```
    pop esi          ;Виймаємо зі стеку адресу початку
                     ;рядка
    xor eax, eax     ;Обнуляємо EAX, у ньому буде
                     ;зберігатися кількість символів
   L1: mov bl, [esi]     ;Заносимо в регістр BL байт,
                         ;що зберігається за адресою ESI
    inc esi          ;Збільшуємо значення в регістрі ESI
                     ;на одиницю
    inc eax          Збільшуємо значення в регістрі EAX
                     ;на одиницю
    cmp bl, 0        ;Порівнюємо прочитаний символ з
                     ;нулем
    jne L1           ; Якщо не нуль, то переходимо до
                     ;початку циклу
    push esi         ;Кладемо в стек адресу байта, що є
                     ;наступним за рядком
    ret              ;Повернення з процедури
   Procedure endp

   end program
```

Рекурсивні процедури

Рекурсія – ресурсомісткий спосіб реалізації алгоритмів, який потребує багато місця для зберігання локальних даних на кожному кроці рекурсії. Рекурсивні процедури зазвичай виконуються не дуже швидко. Тому мові Асемблер, призначеній для написання швидких програм, рекурсія зазвичай не є властивою, однак її можна написати. Принципи реалізації рекурсивної процедури мовою Асемблер такі самі, як і іншими мовами. У процедурі має бути термінальна вітка, у якій немає рекурсивного виклику, і робоча вітка.

При реалізації рекурсивних процедур стає особливо важливим використання стеку для передання параметрів і адреси повернення, що дає змогу зберігати дані, які належать до різних рівнів рекурсивних викликів у різних областях пам'яті.

Для прикладу розглянемо рекурсивну процедуру обчислення факторіала цілого беззнакового числа. Процедура отримує параметр через стек і повертає результат через регістр `EAX`:

```
   factorial proc

    mov eax, [esp + 4]   ;Заносимо   в   регістр   EAX
параметр
                         ;процедури
    test eax, eax    ; Перевіряємо значення в регістрі
EAX
```

```
    jz END              ;Якщо EAX = 0, то рекурсивна
                        ;вітка є необхідною
    dec eax             ;Зменшуємо значення в регістрі EAX
на                      ;одиницю
    push eax            ;Кладемо в стек параметр для
                        ;наступного рекурсивного виклику
    call factorial          ;Викликаємо процедуру
    add esp, 4              ; Очищуємо стек, тому що
процедура
                            ;використовує RET без
параметрів
    mul dword ptr [esp + 4]  ;Множимо EAX, який
зберігає
                             ;результат попереднього
                             ;виклику, на параметр
                             ;поточного виклику
процедури
    ret                 ;Повернення з процедури (без
                        ;параметрів)
  END: inc eax          ;Якщо EAX дорівнював нулю, то
                        ;записуємо в EAX одиницю
    ret                 ;Повернення з процедури (без
                        ;параметрів)

  factorial endp
```

Організація інтерфейсу з процедурою

Аргумент – це посилання на деякі дані, які потрібні для виконання функцій модуля і які розміщено поза цим модулем. За аналогією з макрокомандами розглядають поняття формального й фактичного аргументів. Виходячи з цього, формальний аргумент можна розглядати не як безпосередні дані або їх адресу, а як місцетримач для дійсних даних, які буде підставлено в нього з допомогою фактичного аргументу. **Формальний аргумент** можна розглядати як елемент інтерфейсу модуля, а **фактичний аргумент** – як такий, що фактично передається на місце формального аргументу.

Змінні – це дані, розміщені в регістрі або комірці пам'яті, які можна в подальшому змінювати.

Константи – це дані, значення яких не можна змінювати.

Сигнатура процедури (функції) – це ім'я функції, тип значення й список аргументів із зазначенням порядку їх прямування й типів.

Семантика процедури (функції) – це опис того, що певна функція робить. Семантика функції містить опис того, що є результатом обчислення функції, як і від чого цей результат залежить. Зазвичай результат виконання залежить тільки від значень аргументів функції,

але в деяких модулях є поняття стану. Тоді результат функції може залежати від цього стану, і, крім того, результатом може стати зміна стану. Логіка цих залежностей і змін належить до семантики функції. Повним описом семантики функцій є виконуваний код функції або її математичне визначення.

Якщо змінна знаходиться за межами модуля (процедури) і її необхідно передати в нього, то для модуля ця змінна є формальним аргументом. Значення змінної передається в модуль для заміщення відповідної дії з допомогою фактичного аргументу.

Зазвичай один і той самий модуль можна використовувати багато разів для різних наборів значень формальних аргументів. Для передання аргументів мовою Асемблер існують такі способи:

- через регістри;
- через загальну область пам'яті;
- через стек;
- з допомогою директив `extern` і `public`.

Передання аргументів через регістри є найбільш простим у реалізації способом. Дані, передані таким способом, стають доступними негайно після передання керування процедурі. Цей спосіб часто застосовують при переданні невеликого обсягу даних.

Обмеження на спосіб передання аргументів через регістри:

- невелика кількість доступних для користувача регістрів;
- необхідність постійно пам'ятати про те, яка інформація в якому регістрі знаходиться;
- обмеження розміру переданих даних розмірами регістра: якщо розмір даних перевищує 8, 16 або 32 біти, то передання даних з допомогою регістрів зробити не можна, у цьому випадку передавати потрібно не самі дані, а покажчики на них.

Передання аргументів через спільну область пам'яті – припускається, що програма, яку викликають, і програма, яка викликається, використовують деяку область пам'яті як спільну. Для організації такої області пам'яті використовується атрибут комбінування сегментів `common`. Наявність цього атрибуту вказує компоновнику, як потрібно комбінувати сегменти, що мають одне ім'я: усі сегменти, які мають однакове ім'я в об'єднаних модулях, компоновник розташовує, починаючи з однієї адреси оперативної пам'яті. Це означає, що вони будуть перекриватися в пам'яті і, отже, спільно використовувати виділену пам'ять. Дані в сегментах `common` можуть мати однакові імена. Головне – структура загальних сегментів, яка має бути ідентичною в усіх модулях, що використовують обмін даними через спільну пам'ять.

Недоліком цього способу в реальному режимі роботи мікропроцесора є відсутність засобів захисту даних від руйнування, оскільки не можна проконтролювати дотримання правил доступу до цих даних.

Передання аргументів через стек найбільш часто

використовується при викликанні процедур. Суть цього способу полягає в тому, що процедура, яка викликає, самостійно заносить у стек дані, що передаються, після чого передає керування процедурі, яку викликають. При переданні керування процедурі мікропроцесор автоматично записує у вершину стеку чотири байти. Ці байти є адресою повернення до викличної програми. Якщо перед переданням керування процедурі командою `call` у стек було записано передані процедурі дані або покажчики на них, то їх можна знайти під адресою повернення.

Стек обслуговується трьома регістрами:

а) `ESS` – покажчик дна стеку (початку сегмента стеку);

б) `ESP` – покажчик вершини стеку;

в) `EBP` – покажчик бази.

Мікропроцесор автоматично працює з регістрами `ESS` і `ESP` у припущенні, що вони завжди вказують на дно й вершину стеку відповідно. З цієї причини їх вміст змінювати не рекомендується. Для здійснення вільного доступу до даних у стеку архітектура мікропроцесора має спеціальний регістр `EBP`. Так само, як і для регістра `ESP`, при використанні `EBP` автоматично передбачається робота із сегментом стеку.

Перед використанням цього регістра для доступу до даних стеку його вміст необхідно правильно форматувати, тобто передбачається формування в ньому адреси, яка би вказувала безпосередньо на передані дані. Для цього в початок процедури рекомендується включити додатковий фрагмент коду – **пролог процедури**. Код прологу складається всього з двох команд:

```
push ebp
mov ebp, esp
```

Перша команда зберігає вміст `EBP` у стеку для того, щоб виключити псування значення (яке знаходиться в ньому) у процедурі, що викликається. Друга команда прологу налаштовує `EBP` на вершину стеку. Після цього можна не хвилюватися про те, що вміст `ESP` перестане бути актуальним і здійснювати прямий доступ до вмісту стеку.

Закінчення процедури також має містити дії, що забезпечують коректне повернення з процедури. Фрагмент коду, що виконує такі дії, має свою назву – **епілог процедури**. Код епілогу повинен відновити контекст програми в точці виклику процедури з програми, що її викликає. При цьому, зокрема, потрібно відкоригувати вміст стеку, прибравши з нього аргументи, що стали непотрібними і які передавалися в процедуру. Це можна зробити кількома способами:

- використовуючи послідовність з `n` команд `pop xx;` найкраще це робити у викличній програмі відразу після повернення керування з

процедури;

– коригуючи регістр покажчика стека ESP на величину 4 * n, наприклад, командою

```
add esp, NN
```

де NN = 4 * n, (n – кількість аргументів); це також краще робити після повернення керування процедурі, що викликається;

– використовуючи машинну команду ret n як останню виконувану команду в процедурі, де n – кількість байтів, на яку потрібно збільшити вміст регістра ESP після того, як зі стека буде знято складові адреси повернення; цей спосіб є аналогічним попередньому, але виконується автоматично мікропроцесором.

Програма, яка містить виклик процедури з переданням аргументів через стек:

```
.586
.model flat, stdcall
.stack 4096
.data
.code
proc_1 proc         ;Початок процедури
    push ebp            ;Пролог: збереження EBP
    mov ebp, esp        ;Пролог: ініціалізація EBP
    mov eax, [ebp+8]    ;Доступ до аргументу 4
    mov ebx, [ebp+12] ;Доступ до аргументу 3
    mov ecx, [ebp+16] ;Доступ до аргументу 2
    pop ebp             ;Епілог: відновлення EBP
    ret 12
proc_1 endp
main proc
    push 2
    push 3
    push 4
    call proc_1
    ret
main endp
end main
```

Для доступу до аргументу 4 досить зміститися від вмісту EBP на вісім байтів (чотири байти зберігають адресу повернення до викличної процедури, ще чотири байти зберігають значення регістра EBP,

поміщене в стек цією процедурою), для аргументу 3 – на 12 байтів і т. д.

Пролог і епілог процедури можна також замінити командами підтримки мов високого рівня:

а) команда `enter` готує стек для звернення до аргументів, має два операнди:

- перший операнд визначає кількість байтів у стеку, що використовуються для зберігання локальних ідентифікаторів процедури;
- другий операнд визначає рівень вкладеності процедури.

б) команда `leave` готує стек до повернення з процедури і не має операндів:

```
proc_1 proc
enter 0,0
mov eax, [ebp+8]
mov ebx, [ebp+12]
mov ecx, [ebp+16]
leave
ret 12
proc_1 endp
```

Передання аргументів з допомогою директив `extern` і `public` використовується в таких випадках:

- обидва модуля використовують сегмент даних програми, що викликається;
- у кожного модуля є свій власний сегмент даних;
- модулі використовують атрибут комбінування сегментів `public` у директиві сегментації `segment`.

Розглянемо приклад виведення на екран двох символів, описаних у програмі, що викликається. Два модулі використовують тільки сегмент даних програми, що викликається. У цьому випадку не потрібно перевизначення сегмента даних у процедурі, що викликається:

```
;Модуль 1
.586
.model flat, stdcall
.data
STR1    DB "Програма",0
STR2    db "Менє звуть Лена",0
extern p1@0:near
public STR1, STR2
.code
start proc
call p1@0
```

```
;Модуль 2
.586
.model flat, stdcall
public p1
extern          STR1:BYTE,
STR2:BYTE
EXTERN MessageBoxA@16:NEAR
.code
p1 proc
PUSH     0
```

```
ret
start endp
end start
```

```
PUSH      OFFSET STR1
PUSH      OFFSET STR2
PUSH      0
CALL      MessageBoxA@16
ret
p1 endp
end
```

Способи передання аргументів у процедуру

У процедуру можуть передаватися або дані, або їх адреси (покажчики на дані). У мові високого рівня це називають переданням за значенням і за адресою відповідно.

Найбільш простий спосіб передання аргументів у процедуру – **передання за значенням**. При цьому способі передбачається, що передаються самі дані, тобто їх значення. Програма, що викликається, отримує значення аргументу через регістр або стек. При переданні змінних через регістр або стек на їх розмірність накладаються обмеження, пов'язані з розмірністю використовуваних регістрів або стеку. При переданні аргументів за значенням у процедурі, що викликається, обробляються їх копії, тому значення змінних у процедурі, що викликається, не змінюються.

При **переданні** аргументів **за адресою** передбачається, що процедура, яка викликається, одержує не самі дані, а їх адреси. У процедурі потрібно витягти ці адреси тим самим методом, як це робилося для даних, і завантажити їх у відповідні регістри. Після цього, використовуючи адреси в регістрах, слід виконати необхідні операції над самими даними. На відміну від способу передання даних за значенням, при переданні їх за адресою у процедурі, що викликається, обробляється не копія, а оригінал переданих даних. Тому при зміненні даних у процедурі, що викликається, вони автоматично змінюються й у викличній програмі, оскільки зміни стосуються однієї області пам'яті.

Повернення результату з процедури

У загальному випадку програміст має три варіанти повернення значень з процедури.

З використанням регістрів. Обмеження тут такі самі, що й при переданні даних, – невелика кількість доступних регістрів і їх фіксований розмір. Цей спосіб є найбільш швидким, тому його слід використовувати для організації критичних за часом викликів процедур.

З використанням загальної області пам'яті. Цей спосіб є зручним при поверненні великої кількості даних, але потребує уважності при визначенні областей даних і докладного документування для усунення неоднозначностей.

З використанням стеку. Тут, як і при переданні аргументів через стек, потрібно використовувати регістр `EBP`, при цьому можливими є такі варіанти:

- використання для аргументів, що повертаються, тих самих комірок у стеку, які застосовувалися для передання аргументів на процедуру, тобто передбачається заміщення вхідних аргументів, що стали непотрібними, вихідними даними;
- попереднє переміщення в стек нарівні з переданими фіктивних аргументів з метою резервування місця для значення, що повертається; при використанні цього варіанта процедура, звичайно ж, не повинна намагатися очистити стек командою `ret`, цю операцію доведеться робити у викличній програмі, наприклад, командою `pop`.

Завдання

1. Прочитайте завдання, яке виконує наведена нижче програма, напишіть цю програму з ім'ям `Lab8.asm`.
2. Зробіть виконуваний файл і простежте за його роботою в Турбоналагоджувачі з різними даними.
3. Напишіть коментар до кожного рядка програми з вмістом регістрів.

```
;Ця програма переводить вміст регістра AX у
;шістнадцяткове подання й записує результат у рядок,
;зміщення якого зберігається в регістрі BX

.model tiny
.stack 100h
.data
 outStr db '0000$' ;Вихідний рядок
.code
;Вхідні дані для процедури translByte AL – байт, який
;потрібно перевести
;Вихідні дані BX – зміщення рядка, у перші два байти
;якого буде записано результат
translByte proc
     push ax
     push ax
     shr al,4
     cmp al,9
     ja greater10
     mov byte ptr [bx],'0'
     add [bx],al
     jmp next4Bit
greater10:
```

```
        mov byte ptr [bx],'A'
        sub al,10
        add [bx],al
    next4Bit:
        pop ax
        and al,0Fh
        cmp al,9
        ja _greater10
        mov byte ptr [bx+1],'0'
        add [bx+1],al
        jmp exitByteProc
    _greater10:
        mov byte ptr [bx+1],'A'
        sub al,10
        add [bx+1],al
    exitByteProc:
    pop ax
    ret
    translByte endp

    translWord proc
        push ax
        push ax
        shr ax,8
        call translByte
        pop ax
        and ax,00FFh
        add bx,2
        call translByte
        sub bx,2
        pop ax
        ret
    translWord endp
    start:
        mov ax,@data
        mov ds,ax
        mov bx,OFFSET outStr
        mov ax,60000
        call translWord
        mov ah,9
        mov dx,OFFSET outStr
        int 21h
        mov ax,4c00h
        int 21h
end start
```