МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«*Київський політехнічний інститут*»

Програмування мовою Асемблера

Лабораторний практикум з дисципліни «Системне програмування»

Ухвалено
Вченою радою ФПМ
НТУУ «КПІ»
Протокол № 6 від 28.01.2013 р.

Київ

НТУУ «КПІ»

2013

Програмування мовою Асемблера: лабораторний практикум з дисципліни «Системне програмування» для студентів напрямів підготовки «Комп'ютерна інженерія» та «Програмна інженерія» [Електронне видання] / О.К.Тесленко, І.П.Дробязко. – К. : НТУУ «КПІ», 2013. – 165 с.

Навчально-методичне видання

Програмування мовою Асемблера

Лабораторний практикум з дисципліни «Системне програмування»

Лабораторний практикум розроблено для ознайомлення студентів з різними аспектами створення програм мовою Асемблера та набуття ними практичного досвіду програмування мовою Асемблера, а також мовами високого рівня та Асемблера. Навчальне видання призначене для студентів, які навчаються за напрямами 6.050102 «Комп'ютерна інженерія» і 6.050103 «Програмна інженерія» факультету прикладної математики НТУУ «КПІ»

Укладачі *Тесленко Олександр Кирилович*, канд. техн. наук, доц.
Дробязко Ірина Павлівна, ст. викл.

Відповідальний за випуск *Тарасенко Володимир Петрович*, д-р техн. наук, проф.

Рецензент *Пустоваров Володимир Ілліч*, канд. техн. наук, доц.

ЗМІСТ

ВСТУП

Лабораторний практикум з дисципліни ”Системне програмування” призначений для студентів, які навчаються за освітніми напрямами 6.050102 «Комп’ютерна інженерія», 6.050103 «Програмна інженерія». Завданням циклу лабораторних робіт є закріплення отриманих студентами теоретичних знань і набуття ними практичного досвіду програмування мовою Асемблера.

Практикум складається з чотирьох лабораторних робіт, побудованих на єдиній методичній основі. Крім того, дві роботи мають альтернативні завдання (відповідно роботи №2-1 і №2-2 та №4-1 і №4-2). Лабораторні роботи відображають різні аспекти створення програм мовою Асемблера та присвячені вивченню методики реалізації основних програмних конструктів мовою Асемблера, подання мовою Асемблера типових структур даних та методів адресації елементів цих структур для їх обробки у програмах мовою Асемблера, вивченню правил організації взаємозв’язку між програмами мовою Асемблера та мовами високого рівня Паскаль і С++.

Кожна робота виконується на 2-х заняттях. Студент виконує індивідуальне завдання згідно свого варіанта, що узгоджується з викладачем.

Програмні та апаратні засоби для виконання лабораторних робіт

Для виконання лабораторних робіт необхідні наступні апаратні та програмні засоби:

- ПЕОМ з мікропроцесором 80x86 ... Pentium ;
- операційна система Windows, додаток Far;
- макроассемблер MASM (файл masm.exe), або TASM (файл tasm.exe);
- редактор зв’язків (компонувальник) LINK (файл link.exe) або TLINK (файл tlink.exe);
- налагоджувач AFD (файл afd.exe) або Turbo Debugger (файл td.exe);
- редактор текстів - додаток Notepad;

- Turbo Pascal v 6.0 (7.0) або Microsoft Visual Studio 2008 (2010).

План виконання лабораторних робіт

1. Ознайомлення з метою, завданням, теоретичними відомостями лабораторної роботи та матеріалами з відповідних розділів дисципліни.
2. Визначення варіанту завдання та послідовності виконання роботи.
3. Виконання завдання, використовуючи надані рекомендації та визначені засоби розробки.
4. Тестування програми та виправлення помилок за його результатами.
5. Підготовка відповідей на контрольні запитання.
6. Підготовка звіту з лабораторної роботи.
7. Демонстрація викладачеві результатів виконання роботи.

Загальні вимоги до звіту з лабораторної роботи

Результати виконання лабораторної роботи подаються у вигляді працюючої програми (файла .exe) на диску і протоколу, що містить: титульний аркуш, загальне завдання до лабораторної роботи, завдання за варіантом, лістинг програми мовою Асемблера (файла .lst). При виконанні завдань з використанням мови високого рівня – також текст програми мовою високого рівня.

Студент повинен бути готовим відповісти на будь-яке запитання щодо виконання роботи, змісту програми та на контрольні запитання.

ЛАБОРАТОРНА РОБОТА №1

Ознайомлення з типовою структурою програми та технологічними засобами створення програм мовою Асемблера

Мета роботи – ознайомлення з технологією програмування мовою Асемблера.

1.1. Зміст роботи

Робота виконується на двох заняттях. На першому занятті студенти, використовуючи програму lab1.asm, знайомляться з технологією програмування мовою Асемблера. На другому занятті – створюють у відповідності з завданням нескладну програму мовою Асемблера і перевіряють її працездатність за допомогою налагоджувача.

1.2. Теоретичні відомості

Технологія програмування мовою Асемблера складається з наступних аспектів:

- інструментальне середовище та його застосування;
- вимоги до структури програм;
- вимоги до оформлення програм (елементи стилю програмування).

Інструментальне середовище та його застосування

До складу інструментального середовища входять:

1) *Редактор текстів*, який використовується для створення і редагування початкових (входных, source) файлів з програмами мовою Асемблера. Рекомендується створювати початкові файли з розширенням .asm. Як редактор текстів може бути використаний, наприклад, додаток Notepad.
2) *Транслятор програм з мови Асемблера* - **MASM** або **TASM** (файли masm.exe або tasm.exe відповідно). Транслятор обробляє початковий файл

і генерує об’єктний файл (розширення .obj), файл лістингу (.lst) і файл перехресних посилань (.crf).

Об’єктний файл містить програму в кодах команд ЕОМ, а також дані для корекції адресних частин команд при об’єднанні декількох об’єктних файлів в одну програму.

Файл лістингу містить результати трансляції кожного рядка програми мовою Асемблера, власне рядок та діагностичні повідомлення транслятора. Наявність в файлі лістингу результатів трансляції полегшує вивчення мови Асемблера та системи команд ЕОМ. Транслятори програм мовами високого рівня (наприклад, мовою Паскаль) також можуть створювати файли лістингу, проте в них, як правило, відсутні результати трансляції рядків програми у машинні команди.

Файл перехресних посилань містить перелік рядків програми мовою Асемблера, в яких використовується той чи інший ідентифікатор. Цей файл особливо корисний при необхідності виправлення помилок під час розробки значних за розміром програм.

3) *Редактор з’язків* (компонувальник) **LINK** або **TLINK** (файли link.exe або tlink.exe відповідно). Вхідними файлами для редактора зв’язків є об’єктні файли, що можуть розташовуватися також у файлах бібліотек. Редактор створює завантажувальний файл (загрузочный файл, файл .exe) з розширенням .exe, а також файл розподілу пам’яті (файл .map).

Завантажувальний файл містить програму в кодах команд ЕОМ, а також дані для корекції адресних частин команд, які залежать від початкової адреси розміщення програми в пам’яті. Файл розподілу пам’яті містить дані про розміри програми в цілому та окремих її частин (сегментів).

4) *Налагоджувач* (отладчик, debugger) **AFD** або **TD** (файли afd.exe або td.exe відповідно). Оскільки реалізація виведення повідомлень (на екран або принтер) мовою Асемблера порівняно трудомістка (особливо для чисел), то налагоджувачі застосовуються значно інтенсивніше, ніж у випадку мов високого рівня.

Інструкції щодо запуску програм MASM і LINK та порядку роботи з налагоджувачем AFD надані у ДОДАТКАХ А і Б. Запуск програм TASM і TLINK, а також робота з налагоджувачем TD докладно описані в [2].

Налагоджувачі AFD та TD мають широкі можливості, насамперед необхідно вивчити наступні:

- призначення вікон налагоджувача та переключення з одного вікна на інше;
- завантаження програм у пам’ять;
- керування відображенням (скролінг) програм у вікні налагоджувача;
- заміна вмісту регістрів мікропроцесора;
- відображення та заміна вмісту будь-яких областей оперативної пам’яті;
- покроковий (покомандний) режим виконання програм;
- запуск програм на виконання в автоматичному режимі з завданням адрес зупинки.

Послідовність дій при створенні і налагодженні програм мовою Асемблера відповідає наступній ітераційній схемі (рис.1.1):

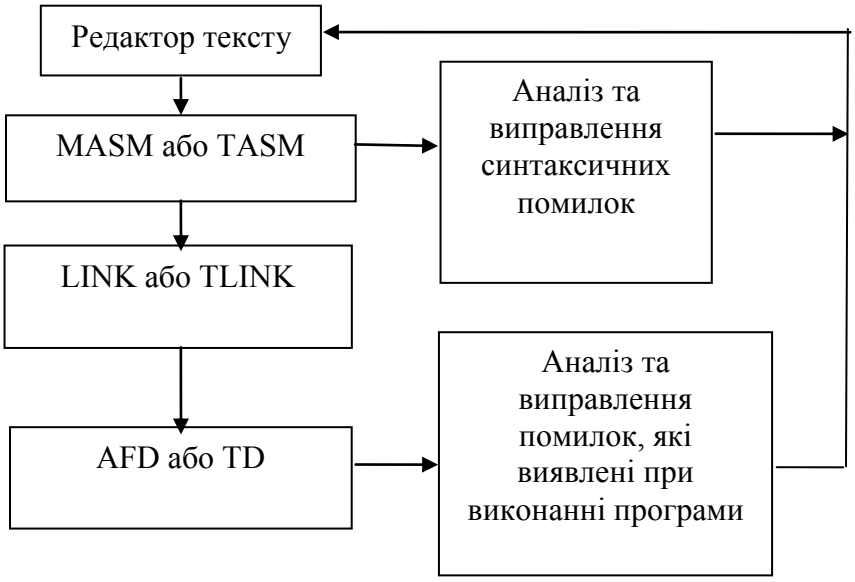

Рис.1.1. Послідовність розробки програми мовою Асемблера

Вимоги до структури програми

При складанні програми мовою Асемблера студент повинен враховувати вимоги синтаксису мови Асемблера, редактора зв'язків та операційної системи.

1) Структура програми мовою Асемблера

Програма мовою Асемблера складається з речень – рядків, які закінчуються символом CR (вводиться клавішею Enter). Рядки можна розділити на чотири типи:

- порожні рядки;
- рядки – коментарі;
- директиви Асемблера (їх називають також командами Асемблера або псевдокомандами);
- машинні інструкції – символічне зображення команд, які виконуються ЕОМ.

Порожні рядки можуть містити тільки символи пробілу чи табуляції. Вони потрібні для наглядної структуризації програм мовою Асемблера.

Рядки – коментарі можуть містити початкові пробіли або символи табуляції, далі символ ';', а за ним – довільні символи.

Директиви Асемблера служать для структуризації програм, резервування пам'яті, завдання даних та управління компілятором.

Машинні інструкції (або машинні команди) служать для символічного відображення команд ЕОМ.

Директиви Асемблера і машинні команди загалом складаються з чотирьох полів, а саме: імені, мнемокоду, операндів, коментаря:

--- *поле мітки* ---|--*поле мнемокоду*--|--*поле операндів*-- |-- *поле коментаря*

Поля розділяються між собою символами пробілу чи табуляції.

Поле імені може бути порожнім або містити ім'я чи мітку. Мітка може бути лише в машинних інструкціях і являє собою символічне позначення адреси команди ЕОМ. За давньою традицією після мітки обов'язково розміщують символ ':'. Імена в полі імені задаються лише в директивах

Асемблера і використовуються для позначення різних об’єктів програми, таких, наприклад, як сегменти, дані, процедури, макроси тощо. У більшості випадків імена програмних об’єктів використовуються у якості символічних позначень їх початкових адрес у пам’яті.

Поле мнемокоду містить символічне позначення машинної інструкції або директиви. У випадку машинних інструкцій, поле мнемокоду найбільше співставляється коду операції машинної команди, тобто в ньому стисло вказується суть команди.

В полі операндів вказуються операнди машинної інструкції, які розділяються комою. В якості операндів можуть бути регістри мікропроцесора, адреси даних у пам’яті (у сегменті) або константи. Операнди вказують на джерела даних для команди і на місце розміщення результату виконання команди. Структура операндів в директивах мови Асемблера суттєво залежить від директиви. Ознакою початку *поля коментаря* є символ ‘;’.

Для Асемблера мікропроцесорів фірми Intel (або їх аналогів) визначено, що операнд для розміщення результату виконання команди завжди задається в полі операндів першим.

Приклади машинних інструкцій:

```
continue:
        mov     ax, bx      ; ax:=bx
        sub     ax, dat1    ; ax:=ax-dat1
        inc     dat1        ; dat1:=dat1+1
        add     bx, 10h     ; bx:=bx+16
        clc                 ; ознака cf:=0
```

де ax, bx – регістри мікропроцесора, dat1 – символічне зображення адреси даних у сегменті даних.

Програма мовою Асемблера сучасних ПЕОМ на базі процесорів фірми Intel або їх аналогів складається з *логічних сегментів*. Типова програма найчастіше містить два логічні сегменти: *сегмент даних* і *сегмент кодів*. Деяким аналогом сегмента даних є декларативна частина програм, наприклад, мовою Паскаль, а сегмента кодів – їх виконавча частина. Початок логічного сегмента визначається директивою **SEGMENT**, а закінчення – директивою

ENDS. Поле імені цих директив містить ім’я логічного сегменту – оригінальний ідентифікатор, який задає програміст.

У початковому файлі логічні сегменти можуть створюватися або змінюватися програмістом згідно з вимогами до програми. Після трансляції, в об’єктному файлі, логічні сегменти мають фіксовані розміри і є атомарними об’єктами програми. Під час роботи редактора зв’язків із логічних сегментів формуються *фізичні сегменти* програми. Кожний фізичний сегмент формується із одного або декількох логічних сегментів. Фактичне розташування логічних сегментів у фізичному сегменті та розмір фізичних сегментів визначає редактор зв’язків при створенні завантажувального файлу. Фактичне розташування фізичних сегментів в адресному просторі ОЗП (прив’язка фізичних сегментів до *сегментів пам’яті*) визначає операційна система при завантаженні програми, тобто фізичні сегменти можуть бути розміщені в будь-якій області оперативної пам’яті.

Сегмент пам’яті – це блок комірок пам’яті з послідовно і безперервно зростаючими адресами. Таким чином, фізична адреса будь-якого об’єкту програми може бути обчислена шляхом додавання адреси об’єкта в сегменті до початкової (базової) адреси фізичного сегменту. Адресу об’єкта в сегменті називають *зміщенням у сегменті*. Початкові адреси сегментів розміщують в сегментних регістрах (**CS**, **DS**, **SS**, ES, GS та FS), а зміщення у сегменті задається в адресній частині команди шляхом використання одного з апаратно реалізованих *режимів адресації*. Вказане додавання початкової адреси фізичного сегменту та зміщення у сегменті виконується процесором автоматично. Очевидно, що у випадку переміщення програмних сегментів в пам’яті буде змінюватися лише початкова адреса сегменту, а зміщення у сегменті (адресні частини команд) можуть не змінюватися.

Для того, щоб операційна система могла виконувати необхідну корекцію вмісту сегментних регістрів, у програмі повинні бути команди завантаження цих регістрів. У випадку програм мовами високого рівня, відповідні команди генерує транслятор і програміст може особливо не турбуватися. У випадку

Асемблера, турботи про вміст сегментних регістрів покладаються на програміста. Детальніше ці питання будуть вивчатися в Лабораторній роботі №3. Програма Лабораторної роботи №1, що представлена нижче, містить наступне визначення вмісту сегментного регістру DS:

```
mov     ax, data    ; data – ім'я логічного сегменту даних
mov     ds, ax
```

Ці команди повинні розміщуватися на початку програми.

Виникає питання: чому б не скористаться командою *mov ds, data*? Відповідь проста: в процесорах 80x86 та Pentium така команда відсутня.

2) *Закінчення програми мовою Асемблера*

Після закінчення роботи програми на програміста покладається обов'язок організації повернення управління в операційну систему. Для цього необхідно наприкінці записати наступні машинні команди:

```
mov     ax, 4c00h   ; 4c00h – код для операційної системи
int     21h         ; виклик функції операційної системи
```

Програма мовою Асемблера закінчується директивою **END**. У полі операндів даної директиви може міститися ідентифікатор мітки першої виконуваної команди програми (точки входження в програму), що є визначенням основної програми. При завантаженні exe-файлу операційна система передає управління в точку входження, тобто у покажчик команд (регістр IP) буде завантажена адреса (зміщення), символічне позначення якої надавалось у директиві END.

Вимоги до стилю програм мовою Асемблера

1) Мова Асемблера не накладає вимог щодо прив'язки полів до конкретних позицій рядка. Все ж таки грамотним вважається дотримання одного й того ж розташування полів протягом всієї програми і використання достатньої кількості пробілів між ними для розділу.
2) Мітки машинних інструкцій доцільно розміщувати в окремих рядках. Крім наочності, це сприяє спрощенню редагування початкових програм.

3) Якщо наступна після команди передачі управління інструкція програми мітки не має, доцільно вставити перед нею порожній рядок. Це дозволяє наочно виділити частини програм з послідовним виконанням команд.
4) Мова Асемблера допускає довільне розміщення логічних сегментів у початковій програмі. Між тим рекомендується наступний (за аналогією з програмами мовами високого рівня) порядок: спочатку розміщуються сегменти даних, а потім – сегменти кодів.
5) Необхідно не лінуватися робити змістовні коментарі, що пояснюють реалізований алгоритм, оскільки зрозуміти алгоритм програми мовою Асемблера значно важче, ніж програми мовою високого рівня.

Докладніше щодо створення програм мовою Асемблера описано в [2– Урок 3. Разработка простой программы на ассемблере. Урок 4. Создание программы на ассемблере].

Огляд рекомендованих до використання операцій

Для виконання завдання до лабораторної роботи рекомендується використовувати логічні операції та операції зсувів, що розглядаються нижче.

Порозрядна логічна інверсія. Найуживаніше символічне позначення *NOT*. При виконанні цієї операції значення кожного розряду даних змінюється на протилежне, наприклад:

NOT

	0	0	1	0	0	1	1	0
=	1	1	0	1	1	0	0	1

Порозрядне логічне множення (AND). Операція виконується над двома операндами однакової довжини, часто її позначають символом $\wedge$ ($Z=X\wedge Y$). Визначення операції для довільного розряду і ($x_i \wedge y_i$) подано в Табл. 1.1.

Завдяки властивостям операції $x\wedge 0=0$, $x\wedge 1=x$, $x\wedge x=x$, вона застосовується в програмуванні для виділення та очищення (занесення 0-вих значень) окремих розрядів.

Таблиця 1.1

Операція AND

x_i \ y_i	**0**	**1**
0	**0**	**0**
1	**0**	**1**

Наприклад, виділення в 8-розрядних даних 5-го, 2-го і 1-го розрядів:

	7	6	5	4	3	2	1	0
	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
AND								
	0	0	1	0	0	1	1	0
=								
	0	0	x_5	0	0	x_2	x_1	0

У розглянутому прикладі очищені 7, 6, 4, 3 та 0-вий розряди.

Порозрядне логічне АБО (OR). Операція виконується над двома операндами однакової довжини, часто її позначають символом $\vee$ ($Z=X\vee Y$). Визначення операції для довільного розряду і ($x_i \vee y_i$) подано в Табл. 1.2.

Таблиця 1.2

Операція OR

x_i \ y_i	**0**	**1**
0	**0**	**1**
1	**1**	**1**

Завдяки властивостям операції $x\vee0=x$, $x\vee x=x$, $x\vee1=1$, вона застосовується для об’єднання розрядів та встановлення (занесення) 1 в окремих розрядах.

Приклад об’єднання розрядів

	y_7	0	y_5	0	0	0	y_1	y_0
OR								
	0	x_6	0	x_4	x_3	x_2	0	0
=								
	y_7	x_6	y_5	x_4	x_3	x_2	y_1	y_0

Приклад встановлення розрядів

	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
OR								
	0	0	1	0	0	1	1	0
=								
	x_7	x_6	1	x_4	x_3	1	1	x_0

Порозрядна сума за модулем два (XOR). Операція виконується над двома операндами однакової довжини, часто її позначають символом ⊕ (Z=X⊕Y). Визначення операції для довільного розряду і ($x_i \oplus y_i$) подано в Табл. 1.3.

Таблиця 1.3

Операція XOR

x_i \ y_i	**0**	**1**
0	**0**	**1**
1	**1**	**0**

Операція XOR має наступні властивості: x⊕0=x, x⊕x=0, x⊕1=NOT x.

Операція може бути використана для інверсії окремих розрядів, наприклад:

	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
XOR								
	0	0	0	1	1	1	0	0_0
=								
	x_7	x_6	x_5	NOT x_4	NOT x_3	NOT x_2	x_1	x_0

Властивість x⊕x=0 часто використовують для очищення (занесення значення 0 в усі розряди) регістра або комірок пам’яті. Наприклад, команди *xor eax, eax* та *mov eax, 0* обидві заносять в регістр EAX нульове значення, але команда *xor* займає в ОЗП дві комірки, а команда *mov* – п'ять комірок.

Лінійний зсув вліво (SHift Left - SHL) на один розряд. В цій операції значення розряду даних заміщується значенням попереднього молодшого розряду. При цьому значення самого старшого розряду даних втрачається (у відповідних командах ЕОМ значення старшого розряду записується в ознаку

переносу ***cf*** в регістрі ознак), а у наймолодший розряд записується 0, наприклад:

SHL

$$\begin{array}{r} x_7\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0 \\ \hline = \\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0\ 0 \end{array}$$

Лінійний зсув вправо (SHift Right - SHR). В цій операції значення розряду даних заміщується значенням попереднього старшого розряду. При цьому значення самого молодшого розряду даних втрачається (у відповідних командах ЕОМ значення самого молодшого розряду записується в ознаку переносу в регістрі ознак), а у найстарший розряд записується 0, наприклад:

SHR

$$\begin{array}{r} x_7\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0 \\ \hline\hline = \\ 0\ x_7\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1 \end{array}$$

Зсув на декілька розрядів можна розглядати як послідовність зсувів на один розряд.

1.3. Завдання на виконання роботи

Перше заняття

1) Переглянути текст програми lab1.asm, зміст якої представлений нижче. При відсутності відповідного файлу на диску, підготувати його, наприклад, за допомогою додатку Notepad.

Програма lab1.asm ілюструє основні елементи, що властиві програмам мовою Асемблера. Вона складається з двох логічних сегментів: *data* і *code*. Логічний сегмент *data* розміщується у фізичному сегменті даних, який адресується регістром **DS**. Логічний сегмент кодів *code* розташовується у фізичному сегменті кодів, який адресується сегментним регістром **CS**.

```
; програма lab1.asm
data    SEGMENT   BYTE
        val1      db    3
        val2      db    2
        result    db    ?
data    ENDS
code    SEGMENT
        ASSUME    cs:code, ds:data
; директива assume інформує Асемблер, що у сегментному регістрі cs міститься адреса
; сегмента code (її в регістр cs помістив завантажувач), а у сегментному регістрі ds –
; адреса сегмента data
begin:
        MOV       AX, DATA    ; запис адреси сегмента data в регістр ax
        MOV       DS, AX      ; запис адреси сегмента data в регістр ds
                              ; (це було "обіцяно" Асемблеру в директиві assume)
        mov       al, val1    ; запис в регістр al значення змінної за адресою val1
        add       al, val2    ; додавання значення змінної за адресою val2
                              ; до вмісту регістра al і запис результату в регістр al
        mov   result, al      ; пересилання вмісту регістра al у сегмент даних за
                              ; адресою result
; фактично реалізований оператор result := val1+val2;
; ОСОБЛИВІСТЬ АСЕМБЛЕРА:
; якщо мовою Паскаль під ідентифікаторами result, val1 та val2 звичайно розуміють
; значення змінних, то мовою Асемблера – адреси змінних у сегменті даних
        nop               ; холоста команда процесора, у даному випадку
                          ; використовується для зручності при роботі з налагоджувачем
        MOV     AX, 4C00H ; повернення в операційну систему
        INT     21H
code    ENDS
        end     begin
```

2) Відтранслювати програму з файлу lab1.asm за допомогою транслятора MASM. Створити об'єктний файл і файл лістингу.
3) Створити exe-файл за допомогою компонувальника LINK.

4) Запустити налагоджувач AFD. За допомогою команди L завантажити програму в середовище налагоджувача. Виконати програму покомандно (клавіша F1) до команди **nop**. Звернути увагу на зміну вмісту регістрів процесора та пам'яті після виконання кожної з команд.
5) Повернутися на початок програми. Виконати перші дві команди, далі здійснити заміну вмісту регістра **AL** мікропроцесора та заміну вмісту областей оперативної пам'яті **val1**, **val2** перед командами, що їх використовують. Пересвідчитись в отриманні нових результатів.

Друге заняття

1) Розробити програму, яка виконує перетворення вмісту довільного однобайтового числа шляхом переміщення (перестановок) його окремих бітів відповідно до варіанта (табл. 1.4). Наприклад, для варіанта 1: значення 7-го розряду вхідного операнда потрібно записати в 0-й та 2-й розряд результату, 2-го – у 1-й, 4-й та 7-й і т.д., 0-го – у 6-й. Вхідний операнд помістити в регістр DL, а результат записати в регістр DH.

Для виконання завдання необхідно ознайомитися з командами пересилання даних (**MOV**), порозрядної логічної обробки (**OR, AND**) та командами лінійного зсуву (**SHL, SHR**) процесорів Intel 80x86 і Pentium. Допускається використання будь-яких інших команд за вибором студента.

2) Перевірити коректність переміщень у налагоджувачі на різних даних.

Таблиця 1.4

Варіанти завдання

Номери розрядів початкових даних

№ варіанта	Номер розряду результату 7 6 5 4 3 2 1 0	№ варіанта	Номер розряду результату 7 6 5 4 3 2 1 0
1	2 0 1 2 3 7 2 7	16	3 4 5 6 1 3 2 5
2	7 7 2 3 2 6 7 5	17	5 2 7 3 1 3 0 5
3	1 6 4 3 6 2 0 6	18	0 5 4 0 6 7 4 1
4	7 2 5 4 7 7 0 4	19	3 0 5 5 2 6 2 7
5	6 4 7 6 5 1 5 1	20	5 0 6 6 1 2 4 4
6	4 7 7 5 7 0 1 3	21	5 4 1 6 5 1 1 1
7	1 0 6 7 2 2 2 6	22	0 6 7 6 4 3 0 5
8	7 3 2 0 6 3 5 3	23	6 3 3 3 7 0 6 1
9	6 3 0 5 3 5 1 1	24	7 7 7 5 5 2 0 5
10	0 0 5 2 2 5 6 1	25	0 3 4 5 4 1 0 4
11	6 1 2 5 6 0 5 7	26	4 3 6 1 1 3 7 6
12	0 0 7 5 3 4 0 4	27	7 0 5 1 7 6 7 1
13	4 5 2 4 4 1 1 4	28	1 7 2 3 3 6 4 5
14	4 7 7 5 5 5 5 7	29	0 7 4 1 5 0 3 2
15	4 3 7 6 4 4 3 6	30	6 1 5 5 1 6 2 2

1.4. Контрольні запитання

1. Яку структуру має програма мовою Асемблера?
2. Яке призначення мають програми MASM, LINK і AFD?
3. Назвіть основні опції програм MASM, LINK і AFD.
4. Як визначити основну програму мовою Асемблера?
5. Чим відрізняються директиви мови Асемблера від її машинних інструкцій?

ЛАБОРАТОРНА РОБОТА №2-1

Реалізація основних програмних конструктів мовою Асемблера. Використання асемблерних вставок у програмах мовою Паскаль

Мета роботи – вивчення методів реалізації мовою Асемблера основних виконавчих операторів мови Паскаль, ознайомлення з методикою включення текстів програм мовою Асемблера в програми мовою Паскаль.

2-1.1. Зміст роботи

Робота виконується на двох заняттях. На першому занятті на основі запропонованої програми мовою Паскаль студенти створюють файл, що містить результати трансляції кожного оператора Паскаль програми в інструкції команд Асемблера, та вивчають методи реалізації мовою Асемблера основних операторів мови Паскаль. На другому занятті створюють у Паскаль програмі асемблерну вставку, що оптимізує, якщо можливо, програму за обсягом і/або швидкодією.

2-1.2. Теоретичні відомості

Подання основних даних в ЕОМ та елементарні операції над ними

1) Розміри, типи, внутрішня логічна структура даних

Дані, які зберігаються в пам’яті і оброблюються командами, відрізняються своїм розміром (кількістю комірок пам’яті або байтів, які вони займають) та логічною структурою. Так, наприклад, компілятор TP мови Паскаль для типу integer відводить 2 байти, а Delphi та C++ – 4 байти для подання цілих зі знаком у доповняльному коді. Асемблер має синтаксичні правила, які дозволяють однозначно визначати розмір даних у байтах, з якими оперує окремо взята команда. У цьому є певна аналогія із завданням типів даних у мовах високого рівня. Саме тому в Асемблері *розмір даних в байтах*

називають типом даних, ігноруючи (на відміну від мов високого рівня) *внутрішню логічну структуру* даних, яка визначається та інтерпретується виключно операціями, що задаються в командах. При цьому в апаратурі ЕОМ і в Асемблері відсутні засоби контролю типів даних, які програмно реалізовані в трансляторах мов високого рівня. Мовою Асемблера задаються, а транслятором контролюються лише розміри даних. Тобто будь-яка команда буде читати з пам'яті стільки байтів, скільки їй потрібно, і буде обробляти їх так, як їй потрібно, незалежно від того, дані якого розміру та якої внутрішньої структури з точки зору програміста там розташовані. Контроль за відповідністю розміру та внутрішньої логічної структури даних командам ЕОМ у програмах мовою Асемблера повністю покладається на програміста. У випадку програм мовами високого рівня цю відповідність забезпечують транслятори, використовуючи описи типів даних, які є обов'язковими.

2) Адресація багатобайтних даних та порядок розташування байтів

В сучасних комп'ютерних системах стандартом фактично є побайтова організація ОЗП, тобто у комірці пам'яті, яка має свій оригінальний номер (фізичну адресу) міститься один байт. Це призвело до деяких ускладнень при організації обробки багатобайтних даних командами процесора.

В процесорах Intel принято, що *молодші байти багатобайтних даних розташовуються в ОЗП за молодшими адресами, а адресою багатобайтних даних є адреса їх молодшого байту.*

Нехай, наприклад, в пам'яті необхідно зберігати число 19498, яке в двійковій системі числення має вигляд 100110000101010, а у шістнадцятковій – 4c2a. Тоді за фізичною адресою Addr буде записано байт 2a (двійкове 00101010), а за фізичною адресою Addr+1 – байт 4c (двійкове 01001100). Фізичною адресою цього числа буде Addr. Як видно з прикладу, десяткова система числення при цьому досить незручна.

3) Цілі числа без знаку

В сучасних інформаційних технологіях цілі числа без знаку можуть мати розмір 1, 2, 4 або 8 байт. Мінімальним значенням цих чисел є 0, а максимальне

обчислюється за формулою 2^{k*8} -1, де k – розмір числа в байтах. Згідно з вищезазначеним, k вибирається із ряду 1,2,4,8. У табл. 2-1.1 надані діапазони значень цілих чисел без знаку в залежності від k.

Таблиця 2-1.1

Діапазони значень цілих чисел без знаку

k	1	2	4	8
Діапазон значень	0..255	0..65535	0..4294967295	0..18446744073709551615

Особливістю апаратної реалізації основних арифметичних операцій чисел без знаку є однакова розрядність операндів. У випадку відмінної розрядності чисел, розрядність числа з меншою розрядністю необхідно збільшити до розрядності числа з більшою розрядністю. Це досягається записом 0 у всі додаткові розряди. Починаючи з процесорів Pentium (відповідно у всіх сучасних процесорах), реалізується універсальна команда *MOVZX* для розширення розрядності беззнакового числа.

При додаванні чисел їх сума може перевищувати максимальне значення. В цьому випадку виникає перенесення із старшого розряду. Для фіксації наявності (або відсутності) перенесення використовується біт переносу ***cf*** (*carry flag*) у регістрі ознак. Вміст розряду переносу визначає, таким чином, наявність переповнення при додаванні беззнакових цілих чисел.

При відніманні беззнакових чисел також формується ознака переносу, яка фактично визначає позику із віртуального старшого розряду, який умовно знаходиться за старшим розрядом двійкового подання числа. Значення ознаки переносу дорівнює 1, якщо зменшуване менше за від'ємник, і 0 в інших випадках.

Таким чином, значення ознаки переносу, сформоване після операції віднімання беззнакових цілих чисел можна використовувати як результат їх порівняння. Тобто, якщо після виконання операції X-Y *cf=1*, то X<Y, а якщо *cf=0*, то X≥Y.

Для ідентифікації порівнянь X=Y, X>Y, та X≤Y необхідно використовувати додаткову ознаку – *ознаку нуля*. Ознака приймає значення 1, якщо результатом є нуль, в усіх інших випадках ознака нуля приймає значення 0. Ознака нуля позначається як ***zf*** (*zero flag*) і також міститься в регістрі ознак.

Команда процесора порівняння (мнемокод *CMP - CoMPare*) виконує операцію віднімання операндів, встановлюючи відповідні значення ознак, але результат віднімання нікуди не записує. Отже, після виконання операції X-Y для беззнакових чисел маємо:

- X>Y, якщо ((*cf*=0) і (*zf*=0)),
- X≥Y, якщо *cf*=0,
- X=Y, якщо *zf*=1,
- X≠ Y, якщо *zf*=0,
- X≤Y, якщо ((*cf*=1) і (*zf*=1)),
- X<Y, а якщо *cf*=1.

Сформовані ознаки використовуються в *командах передачі управління за умовою* процесорів 80x86 (Pentium) (табл. 2-1.2).

Таблиця 2-1.2

Команди передачі управління за умовою для операндів без знаку

Співвідношення між операндами	Значення ознак при відніманні (X - Y)	Найменування співвідношення	Мнемокод команди передачі управління за умовою
X > Y	(***cf***=0) і (***zf***=0)	Вище (або не нижче і не дорівнює)	**JA** (або **JNBE**)
X ≥ Y	***cf***=0	Вище чи дорівнює (або не нижче)	**JAE** (або **JNB**, або **JNC**)
X = Y	***zf***=1	Дорівнює	**JE** (або **JZ**)
X ≠ Y	***zf***=0	Не дорівнює	**JNE** (або **JNZ**)
X ≤ Y	***cf***=1 чи ***zf***=1	Нижче чи дорівнює (або не вище)	**JBE** (або **JNA**)
X < Y	***cf***=1	Нижче (або не вище і не дорівнює)	**JB** (або **JNAE**, або **JC**)

Як видно, одна й та сама команда передачі управління за умовою може мати декілька мнемокодів, що надає можливість програмістам краще відтворювати алгоритм програми і, тим самим, покращити його розуміння.

Інтерпретація результатів операції зсуву для беззнакових чисел. Лінійний зсув вліво (команда *SHL – SHift Left*) на один розряд інтерпретується як операція множення на 2. Вміст ознаки перенесення, як і при додаванні, визначає факт переносу. *Лінійний зсув вправо* (команда *SHR – SHift Right*) на один розряд інтерпретується як операція ділення цілого додатного числа на 2. В результаті зсуву на один розряд вправо формується ціле значення частки від ділення на 2, а ознака перенесення містить значення залишку. При багаторозрядних зсувах ознака переносу формується за результатом останнього зсуву.

4) Цілі числа зі знаком

В процесорах 80x86 (Pentium) цілі числа зі знаком подаються в доповняльному коді. Крім того, вся адресна арифметика (формування ефективної адреси у випадку багатокомпонентної адреси, формування нового значення регістра IP при відносній адресації, формування фізичної адреси із логічної) в процесорах 80x86 (Pentium) виконується в доповняльному коді.

Позначимо через X_m доповняльний код числа X. Тоді

$$X_m = \begin{cases} X, \text{ якщо } X \geq 0 \\ 2^{k*8} - |X|, \text{ якщо } X < 0 \end{cases}$$

Діапазон значень доповняльного коду:

$$-2^{k*8-1} \leq X \leq +2^{k*8-1} - 1$$

Старший біт доповняльного коду визначає знак числа. Операція зміни знаку виконується за два кроки: на першому кроці виконується операція порозрядної логічної інверсії, а на другому – до результату додається 1.

Для ручної зміни знаку числа в доповняльному коді доречним є наступне правило: в коді числа справа (з молодших розрядів) пропустити всі 0 та першу 1, а в решті розрядів замінити 0 на 1, а 1 на 0.

У випадку відмінної розрядності чисел при виконанні операцій, розрядність числа з меншою розрядністю необхідно збільшити до розрядності числа з більшою розрядністю, не змінюючи ні знак, ні значення модуля числа. У випадку додатних чисел, додаткові старші розряди заповнюються значенням 0. У випадку від'ємних чисел, вони заповнюються значенням 1, тобто розширення полягає в заповненні старших розрядів значенням знакового розряду (команди *CBW - Convert Byte to Word*, *CWDE - Convert Word to DoubleWord*). Починаючи з процесорів Pentium (відповідно у всіх сучасних процесорах) реалізується універсальна команда знакового розширення *MOVSX*.

Операція додавання чисел зі знаком. Для додавання чисел із знаком в доповняльному коді можна використати операцію беззнакового додавання і, таким чином, одну й ту ж саму команду процесора (*ADD – ADDition*) як для додавання беззнакових чисел, так і для додавання чисел зі знаком. Але при виконанні операції додавання для чисел зі знаком необхідно враховувати додаткову ознаку - *ознаку переповнення* ***of*** (overflow flag) із регістра ознак, яка приймає значення 1, коли результат додавання виходить за межі діапазону та 0, коли результат додавання не виходить за межі діапазону (у випадку беззнакових чисел цю функцію виконує *ознака перенесення* ***cf***). Ознака переповнення записується в регістр ознак і далі може використовуватися в командах передачі управління за умовою. Крім того, ознака переповнення може бути джерелом внутрішнього переривання (виключення) для інформування, при необхідності, про помилку в обчисленнях.

При додаванні чисел зі знаком часто виникає потреба аналізувати знак результату. Для спрощення аналізу знаку в регістр ознак розміщують *ознаку знаку* ***sf*** *(sign flag)*, значення якої є копією значення старшого розряду результату. Ця ознака використовується в *командах передачі управління за умовою.* Команди передачі управління за умовою (див. далі) дозволяють проаналізувати ознаки, встановлені арифметичними (та деякими іншими) командами, та організувати правильний процес обробки у програмі.

Операція віднімання чисел зі знаком. Для віднімання чисел зі знаком у доповняльному коді можна використати операцію беззнакового віднімання і, таким чином, одну й ту ж саму команду процесора (*SUB – SUBtract***)** як для віднімання беззнакових чисел, так і для віднімання чисел зі знаком. Операція віднімання може бути використана для порівняння чисел. Якщо для порівняння беззнакових чисел можна використовувати ознаку перенесення (наявність позики) та інколи додатково ознаку нуля, то для порівняння чисел зі знаком необхідно аналізувати співвідношення значень ознак переповнення та знаку (інколи додатково ознаку ***zf***). Сформовані (після виконання операції для чисел у доповняльному коді) ознаки використовуються в *командах передачі управління за умовою* (табл. 2-1.3).

Таблиця 2-1.3

Команди передачі управління за умовою для операндів зі знаком

Співвідношення між операндами	Значення ознак при відніманні (X - Y)	Найменування співвідношення	Мнемокод команди передачі управління за умовою
X > Y	(***of***=***sf***) і (***zf***=0)	Більше (або не менше і не дорівнює)	**JG** (або **JNLE**)
X ≥ Y	***of***=***sf***	Більше чи дорівнює (або не менше)	**JGE** (або **JNL**)
X = Y	***zf***=1	Дорівнює	**JE** (або **JZ**)
X ≠ Y	***zf***=0	Не дорівнює	**JNE** (або **JNZ**)
X ≤ Y	(***of***≠***sf***) чи (***zf***=1)	Менше чи дорівнює (або не більше)	**JLE** (або **JNG**)
X < Y	***of***≠***sf***	Менше (або не більше і не дорівнює)	**JL** (або **JNGE**, або **JC**)

Таким чином, тільки для співвідношень "дорівнює" та "не дорівнює" для чисел зі знаком в доповняльному коді і чисел без знаку команди передачі

управління за умовою співпадають. Для решти співвідношень операндів команди передачі за умовою відрізняються. Програміст і тільки програміст шляхом використання відповідних команд передачі управління за умовою визначає, які саме дані порівнювались. Це часто буває джерелом помилок при використанні у програмах команд передачі управління *JG\JNLE, JGE\JNL, JLE\JNG, JL\JNGE після порівняння чисел без знаку та команд JA\JNBE, JAE\JNB, JBE\JNA, JB\JNAE після порівняння чисел зі знаком.*

Операції множення і ділення. Операції відрізняються для чисел без знаку і чисел зі знаком у доповняльному коді**,** оскільки у другому випадку необхідно враховувати знаки операндів і формувати знак результату (команди множення для чисел без знаку *MUL - MULtiple* та для чисел зі знаком – *IMUL*; відповідно *DIV - DIVide* та *IDIV*).

Операції зсуву для чисел зі знаком. Для чисел зі знаком використовуються арифметичні зсуви.

Арифметичний зсув вліво (*SAL- Shift Arithmetic Left*). У випадку доповняльного коду арифметичний зсув вліво співпадає з лінійним зсувом вліво, тобто команда *SAL* еквівалентна команді *SHL*. І якщо програміст інтерпретує вміст бітового поля як число зі знаком, тоді після виконання операції зсуву на один розряд вліво потрібно аналізувати ознаку переповнення, а у випадку беззнакових чисел – ознаку перенесення. Необхідно мати на увазі, що у випадку багаторозрядних зсувів вліво ознака переповнення не визначена.

Арифметичний зсув вправо (*SAR - Shift Arithmetic Right*). При арифметичному зсуві вправо на один розряд у вивільнений старший розряд записується його попереднє (до зсуву) значення. Тим самим виконується операція ділення числа зі знаком на 2 з округленням в сторону -∞. При цьому ознака переповнення скидається в 0. Арифметичні зсуви на 2 і більше розрядів вправо визначаються як послідовність зсувів на один розряд. При багаторозрядному арифметичному зсуві вправо ознака переповнення вважається невизначеною.

Завдання операндів машинних команд

Можливі наступні варіанти завдання в машинних командах значень та місця розташування даних:

- операнди *за замовчуванням*;
- значення задаються *безпосередньо в команді*;
- операнди знаходяться *в регістрах*;
- операнди розташовуються *в пам'яті*.

Більшість команд процесорів 80x86 мають у своїй структурі адресну частину, яка у загальному випадку містить *байти режиму адресації modr/m, sib* та *зміщення в команді*.

Для операндів, які розташовані в пам'яті**,** адреса формується як сума двох складових: зсунутого на 4 біти вліво вмісту сегментного регістра та *зміщення у сегменті – ефективної адреси*, що у загальному випадку є сумою трьох компонент: *зміщення в команді* (задається безпосередньо), *бази* та *індексу*. База та індекс містяться в регістрах загального призначення **EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP**, які використовуються як адресні регістри**.** Індекс може мати множник 2,4 або 8, який визначає, на яке число необхідно помножити вміст 32-розрядного індексного регістра перед формуванням ефективної адреси (обмеження – регістр **ESP** не може задаватись із множником). Множник ефективно використовується для доступу до елементів масивів. Наприклад:

```
mov   eax, Ar[ebx + edi*4]
mov   edx, [ecx + edi]
mov   ebx, [eax + esi*2] + 8
```

Для 16-розрядних регістрів множник не задається, і як адресні регістри використовуються регістри **BX, SI, DI, BP.** Для формування ефективної адреси можуть використовуватись лише ці регістри, а також лише наступні їх пари: **BX+SI**, **BX+DI**, **BP+SI**, **BP+DI**. Наприклад:

```
mov   ax, [X + di]
mov   ax, [A + si - 4]
```

Докладніше про способи адресації див. у Лабораторній роботі №3.

2-1.3. Рекомендації до виконання роботи

Знання мови Паскаль може допомогти у вивченні мови Асемблера. Дійсно, знаючи оператори мови Паскаль, а також маючи результати трансляції Паскаль операторів на мову Асемблера, не важко зрозуміти, ЩО САМЕ виконують окремі команди невеликого фрагмента програми мовою Асемблера, який реалізує окремий Паскаль оператор.

Перша проблема, що необхідно вирішити, – це створити файл, в якому після кожного Паскаль оператора містились би машинні інструкції – результати трансляції Паскаль оператора на мову Асемблера. Більшість компіляторів мов високого рівня мають стандартний режим генерації такого файла, але він відсутній в інтегрованому середовищі ТурбоПаскаль. Існує багато варіантів створення вказаного файлу. Всі вони грунтуються на перетворенні програми в кодах команд процесора в програму мовою Асемблера. В даній лабораторній роботі для цього рекомендується використовувати відповідну команду налагоджувача AFD. Розглянемо процес створення такого файлу та його подальшого аналізу на прикладі нескладної програми мовою Паскаль.

Програма мовою Паскаль містить:

- невелику кількість операторів;
- дані типу integer, byte, word та масиви даних цих типів;
- найуживаніші оператори мови Паскаль (присвоєння, **if..then..else, for , while, repeat..until**);
- контрольне виведення даних на екран;

Приклад файлу test0000.pas

```
{1}   program test0000;
{2}   var
{3}     c,d: byte;
{4}     k: integer;
{5}     A: array [3..10] of integer;
{6}
{7}   begin
{8}   k:=6;
{9}   d:=0;
```

```
{10}  for c:=3 to 10 do
{11}  begin
{12}      if k<d then
{13}          d:=d+k
{14}      else d:=k-d;
{15}      A[c]:=d;
{16}      k:=k+1;
{17}  end;
{18}   for c:=3 to 10 do
{19}   write (A[c]:4);writeln;
{20}   end.
```

Настроювання інтегрованого середовища ТурбоПаскаль

Необхідно виконати наступне настроювання інтегрованого середовища ТурбоПаскаль:

- Меню **Compile**. Опцію **Destination** встановити у стан **Disk**.
- Меню **Options**. У підменю **Memory Size**, опцію **High heap limit** встановити в 0 (цим вивільняємо пам'ять, щоб уможливити завантаження програми у налагоджувач AFD).
- У підменю **Linker** опцію **Map file** встановити в стан **Detailed**, опцію **Link bufer** у стан **Disk** (цим забезпечується формування файлу карти пам'яті *.map*, який містить розподіл об'єктів програми в пам'яті).

Файл test0.map
(перелік сегментів)

Start	Stop	Length	Name	Class
00000H	000AEH	000AFH	test0000	CODE
000B0H	00950H	008A1H	System	CODE
00960H	00C11H	002B2H	DATA	DATA
00C20H	04C1FH	04000H	STACK	STACK
04C20H	04C20H	00000H	HEAP	HEAP

(структура сегмента даних)

Address	**Publics by Value**
0000:0000	@
0096:0002	OvrCodeList
0096:0004	OvrHeapSize
0096:0006	OvrDebugPtr

0096:000A OvrHeapOrg
0096:000C OvrHeapPtr
0096:000E OvrHeapEnd
0096:0010 OvrLoadList
0096:0012 OvrDosHandle
0096:0014 OvrEmsHandle
0096:0016 HeapOrg
0096:001A HeapPtr
0096:001E HeapEnd
0096:0022 FreeList
0096:0026 FreeZero
0096:002A HeapError
0096:002E ExitProc
0096:0032 ExitCode
0096:0034 ErrorAddr
0096:0038 PrefixSeg
0096:003A StackLimit
0096:003C InOutRes
0096:003E RandSeed
0096:0042 SelectorInc
0096:0044 Seg0040
0096:0046 SegA000
0096:0048 SegB000
0096:004A SegB800
0096:004C Test8086
0096:004D Test8087
0096:004E FileMode
0096:0052 c
0096:0053 d
0096:0054 k
0096:0056 A
0096:0066 Input
0096:0166 Output
0096:0266 SaveInt00
0096:026A SaveInt02
0096:026E SaveInt1B
0096:0272 SaveInt21
0096:0276 SaveInt23
0096:027A SaveInt24
0096:027E SaveInt34
0096:0282 SaveInt35
0096:0286 SaveInt36
0096:028A SaveInt37
0096:028E SaveInt38
0096:0292 SaveInt39
0096:0296 SaveInt3A
0096:029A SaveInt3B
0096:029E SaveInt3C
0096:02A2 SaveInt3D
0096:02A6 SaveInt3E
0096:02AA SaveInt3F
0096:02AE SaveInt75

(структура сегмента кодів)

Line numbers for test0000(TEST0000.PAS) segment test0000

7 0000:0000	8 0000:000F	9 0000:0015	10 0000:001A
12 0000:0025	13 0000:0030	14 0000:003E	15 0000:004D
16 0000:0061	17 0000:0068	18 0000:006F	19 0000:007A
20 0000:00A7			

Program entry point at 0000:0000

end of test0000.map

Файл карти пам'яті складається із трьох частин: *таблиці сегментів*, *структури сегмента даних* і *структури сегмента кодів*. З *таблиці сегментів* видно, що компілятор мови Паскаль формує додатковий сегмент кодів System, який значно об'ємніший за основний. Цей сегмент містить стандартні процедури компілятора, навіть якщо вони не всі потрібні для конкретної програми. Із *структури сегмента даних* також зрозуміло, що значна частина сегменту заповнена системними даними. Очевидно, що більшість із них в даній програмі також не використовується. У структурі сегменту даних жирним шрифтом відмічені дані, що визначені в програмі. Кожний ідентифікатор із структури є символічним зображенням адреси (зміщення) в сегменті, яка вказана ліворуч від ідентифікатора та праворуч від символу ":" Ліворуч від символу ":" позначається одна із можливих адрес сегменту.

У *структурі сегменту кодів* подані адреси (зміщення) у сегменті кодів послідовностей команд ПЕОМ, які згенеровані транслятором із рядків Паскаль програми з відповідними порядковими номерами. Наприклад, 11-ий рядок з ключовим словом *Begin*, який започатковує тіло циклу *for*, не має реалізації в кодах команд, а 17-ий рядок з ключовим *End*, який вказує на закінчення тіла циклу *for*, – має .

Робота з налагоджувачем AFD

Звичайно, попередньо програму testxxxx.exe необхідно завантажити у налагоджувач за допомогою команди **L** (**L**oad). Налагоджувач AFD дозволяє створювати файли, що містять результати перетворення програми в кодах

команд ЕОМ у програму мовою Асемблера (*процес зворотної трансляції*). Для цього використовується команда **PD**, що має 3 параметри:

- початкова адреса ділянки пам’яті;
- довжина ділянки;
- ім’я файлу результата.

Перший із них – початкова адреса – дорівнює молодшому (правому від символу "**:**") значенню *Program entry point*, показаному у файлі testxxxx.map (це компонента зміщення у сегменті логічної адреси точки входу в програму – Offset Program entry point). Для визначення довжини L ділянки пам’яті рекомендується використати наступну формулу:

L = Addr_end +8 - Offset_Program_entry_point

де Addr_end – зміщення у сегменті кодів останнього рядка Паскаль програми (*end*.), 8 – сукупна кількість байтів команд ПЕОМ, які реалізують цей рядок.

ЗАУВАЖЕННЯ. Другий параметр команди PD (довжина) також задається у 16-ковому форматі.

Таким чином, із файла test0000.map маємо:

L = 0A7h+8-0 = 0**Af**h,

а команда PD матиме вигляд: **PD 0, 0af, test0000.prn**

Приклад файлу test0000.prn

```
>> AFD-Pro print out                    9-9-2003    21:53
9717:0000    9A00002297      CALL        9722:0000
9717:0005    55              PUSH        BP
9717:0006    89E5            MOV         BP,SP
9717:0008    31C0            XOR         AX,AX
9717:000A    9ACD022297      CALL        9722:02CD
9717:000F    C70654000600    MOV         [0054],0006
9717:0015    C606530000      MOV         [0053],00
9717:001A    C606520003      MOV         [0052],03
9717:001F    EB04            JMP         0025
9717:0021    FE065200        INC         B/[0052]
9717:0025    A05300          MOV         AL,[0053]
9717:0028    30E4            XOR         AH,AH
9717:002A    3B065400        CMP         AX,[0054]
9717:002E    7E0E            JNG         003E
9717:0030    A05300          MOV         AL,[0053]
9717:0033    30E4            XOR         AH,AH
```

```
9717:0035  03065400      ADD   AX,[0054]
9717:0039  A25300        MOV   [0053],AL
9717:003C  EB0F          JMP   004D
9717:003E  A05300        MOV   AL,[0053]
9717:0041  30E4          XOR   AH,AH
9717:0043  8BD0          MOV   DX,AX
9717:0045  A15400        MOV   AX,[0054]
9717:0048  2BC2          SUB   AX,DX
9717:004A  A25300        MOV   [0053],AL
9717:004D  A05300        MOV   AL,[0053]
9717:0050  30E4          XOR   AH,AH
9717:0052  8BD0          MOV   DX,AX
9717:0054  A05200        MOV   AL,[0052]
9717:0057  30E4          XOR   AH,AH
9717:0059  8BF8          MOV   DI,AX
9717:005B  D1E7          SHL   DI,1
9717:005D  89955000      MOV   [0050+DI],DX
9717:0061  A15400        MOV   AX,[0054]
9717:0064  40            INC   AX
9717:0065  A35400        MOV   [0054],AX
9717:0068  803E52000A    CMP   [0052],0A
9717:006D  75B2          JNZ   0021
9717:006F  C606520003    MOV   [0052],03
9717:0074  EB04          JMP   007A
9717:0076  FE065200      INC   B/[0052]
9717:007A  BF6601        MOV   DI,0166
9717:007D  1E            PUSH  DS
9717:007E  57            PUSH  DI
9717:007F  A05200        MOV   AL,[0052]
9717:0082  30E4          XOR   AH,AH
9717:0084  8BF8          MOV   DI,AX
9717:0086  D1E7          SHL   DI,1
9717:0088  8B855000      MOV   AX,[0050+DI]
9717:008C  99            CWD
9717:008D  52            PUSH  DX
9717:008E  50            PUSH  AX
9717:008F  6A00          PUSH  0000
9717:0091  9A91062297    CALL  9722:0691
9717:0096  9ADD052297    CALL  9722:05DD
9717:009B  9A91022297    CALL  9722:0291
9717:00A0  803E52000A    CMP   [0052],0A
9717:00A5  75CF          JNZ   0076
9717:00A7  C9            LEAVE
9717:00A8  31C0          XOR   AX,AX
9717:00AA  9A16012297    CALL  9722:0116
```

end of test0.prn

Оброблення файлу testxxxx.prn

Виконати ці пункти зручніше або при використанні роздруківок файлів testxxxx.pas, testxxxx.map і testxxxx.prn, або ж при розміщенні, наприклад, за допомогою додатку Notepad, вказаних 3-х відкритих файлів на екрані наступним чином:

файл testxxxx.prn	файл testxxxx.pas
файл testxxxx.map	

Далі виконується почергове перенесення рядків Паскаль програми testxxxx.pas у файл testxxxx.prn і запис їх перед командами з адресами рядків, вказаними у файлі testxxxx.map. Таким чином встановлюється відповідність між Паскаль операторами та їх реалізацією командами Асемблера.

Заміна числових адресних посилань на символічні виконується за допомогою інформації з файлу testxxxx.map. Наприклад, **0052** змінюється на логічне ім'я **c**, **0053** на **d** і т.д. В командах мовою Асемблера, у деяких випадках, необхідно вказувати довжину в байтах змінної, яка адресується. Відлагоджувач AFD у цих випадках генерує символи W/ або B/. Їх необхідно замінити відповідно на *word ptr* або *byte ptr*. В результаті проведених замін файл test0000.prn буде мати наступний вигляд:

```
>> AFD-Pro print out                  9-9-2003      21:53
  begin
9717:0000    9A00002297          CALL        9722:0000
9717:0005    55                  PUSH        BP
9717:0006    89E5                MOV         BP,SP
9717:0008    31C0                XOR         AX,AX
9717:000A    9ACD022297          CALL        9722:02CD

    k:=6;
9717:000F    C70654000600        MOV         [k],0006
    d:=0;
9717:0015    C606530000          MOV         [d],00
    for c:=3 to 10 do
9717:001A    C606520003          MOV         [c],03
9717:001F    EB04                JMP         0025
9717:0021    FE065200            INC         Byte ptr [c]
        begin
          if k<d then
9717:0025    A05300              MOV         AL,[d]
```

```
9717:0028    30E4            XOR     AH,AH
9717:002A    3B065400        CMP     AX,[k]
9717:002E    7E0E            JNG     003E
        d:=d+k
9717:0030    A05300          MOV     AL,[d]
9717:0033    30E4            XOR     AH,AH
9717:0035    03065400        ADD     AX,[k]
9717:0039    A25300          MOV     [d],AL
9717:003C    EB0F            JMP     004D
        else d:=k-d;
9717:003E    A05300          MOV     AL,[d]
9717:0041    30E4            XOR     AH,AH
9717:0043    8BD0            MOV     DX,AX
9717:0045    A15400          MOV     AX,[k]
9717:0048    2BC2            SUB     AX,DX
9717:004A    A25300          MOV     [d],AL
        A[c]:=d;
9717:004D    A05300          MOV     AL,[d]
9717:0050    30E4            XOR     AH,AH
9717:0052    8BD0            MOV     DX,AX
9717:0054    A05200          MOV     AL,[c]
9717:0057    30E4            XOR     AH,AH
9717:0059    8BF8            MOV     DI,AX
9717:005B    D1E7            SHL     DI,1
9717:005D    89955000        MOV     [A+DI-6],DX
        k:=k+1;
9717:0061    A15400          MOV     AX,[k]
9717:0064    40              INC     AX
9717:0065    A35400          MOV     [k],AX
      end;
9717:0068    803E52000A      CMP     [c],0A
9717:006D    75B2            JNZ     0021
  for c:=3 to 10 do
9717:006     C606520003      MOV     [c],03
9717:0074    EB04            JMP     007A
9717:0076    FE065200        INC     byte ptr [c]
   write (A[c]:4);
   writeln;
9717:007A    BF6601          MOV     DI,0166
9717:007D    1E              PUSH    DS
9717:007E    57              PUSH    DI
9717:007F    A05200          MOV     AL,[c]
9717:0082    30E4            XOR     AH,AH
9717:0084    8BF8            MOV     DI,AX
9717:0086    D1E7            SHL     DI,1
9717:0088    8B855000        MOV     AX,[A+DI-6]
9717:008C    99              CWD
9717:008D    52              PUSH    DX
9717:008E    50              PUSH    AX
9717:008F    6A00            PUSH    0000
9717:0091    9A91062297      CALL    9722:0691
9717:0096    9ADD052297      CALL    9722:05DD
```

```
9717:009B   9A91022297      CALL     9722:0291
9717:00A0   803E52000A      CMP      [c],0A
9717:00A5   75CF            JNZ      0076
  end.
9717:00A7   C9              LEAVE
9717:00A8   31C0            XOR      AX,AX
9717:00AA   9A16012297      CALL     9722:0116
                  end of test0000.prn
```

Ознайомлення з реалізацією Паскаль операторів мовою Асемблера

Цей пункт найважливіший в даній лабораторній роботі. За допомогою довідника з системи команд мікропроцесора (наприклад, див. [2]) необхідно ознайомитись з командами, які реалізують відповідний оператор Паскаль програми, та зрозуміти (вивчити) методику реалізації цих операторів.

Асемблерна вставка

Асемблерна вставка у Паскаль програмі починається оператором **asm** і закінчується оператором **end;**. Між ними записуються машинні інструкції мови Асемблера і директиви визначення пам'яті. Всі мітки повинні починатися символом **@**. У директивах визначення пам'яті поле мітки може містити мітку, а не ім'я. В адресних частинах команд як адресні посилання можуть використовуватися імена змінних і процедур, котрі видимі (доступні) в даній точці програми. Необхідно використовувати покажчики довжини змінних (*word ptr* або *byte ptr*). Константи задаються за правилами як Паскаля, так і Асемблера. Коментарі задаються за правилами Паскаля.

Уважне вивчення команд ПЕОМ дозволяє в ряді випадків зробити більш ефективну реалізацію окремих операторів Паскаля порівняно з результатами роботи транслятора:

```
{1}  program test0000;
{2}  var
{3}    c,d: byte;
{4}    k: integer;
{5}    A: array [3..10] of integer;
{6}
{7}  begin
{8}  k:=6;
{9}  d:=0;
```

```
{10}   for c:=3 to 10 do
{11}   begin
       asm
{12     if k<d then}
       mov    al,d
       xor      ah,ah
       cmp     ax,k
       jng      @20
{13     d:=d+k}
       mov  al,byte ptr k
       add   byte ptr d,al
       jmp    @30
{14     else d:=k-d;}
@20:
       mov   al,d
       mov dl,byte ptr k
       sub   dl,al
       mov   byte ptr d,dl
{15     A[c]:=d;}
@30:
       mov   al,d
       xor     ah,ah
       mov    bl,c
       xor     bh,bh
       shl     bx,1
       mov    word ptr A[bx-6],ax
{16     k:=k+1;}
       inc    word ptr k
       end; {asm}
{17}   end;
{18}   for c:=3 to 10 do
{19}   write (A[c]:4);writeln;
{20}   end.
```

Навіть у цій дуже простій програмі можна досягти деякої економії пам’яті та підвищення швидкодії при реалізації рядків 13, 15 та 16.

2-1.4. Завдання на виконання роботи

Перше заняття

1) Скопіювати програму мовою Паскаль testxxxx.pas (табл.2-1.4), де xxxx – номер варіанту. Перевірити працездатність програми, відкомпілювавши та запустивши її у середовищі Турбо Паскаль.

2) Створити файли testxxxx.exe і testxxxx.map, здійснивши відповідне настроювання середовища Турбо Паскаль. Для виконання даного та

наступних пунктів необхідно використовувати надані вище Рекомендації до виконання роботи та приклади.

3) За допомогою команди PD налагоджувача AFD створити файл testxxxx.prn.

4) У файлі testxxxx.prn перед групою команд ПЕОМ, які сформовані транслятором на основі чергового рядка Паскаль програми testxxxx.pas, записати цей Паскаль-рядок. Інформація про початкові адреси рядків береться із структури сегмента кодів, що у файлі testxxxx.map.

5) За інформацією з файла testxxxx.map замінити у файлі testxxxx.prn числові адресні посилання на символічні.

Друге заняття

1) Ознайомитись з реалізацією операторів Паскаль програми командами мови Асемблера.

2) Вибрати декілька Паскаль операторов (наприклад, тіло циклу) і замінити їх у програмі testxxxx.pas асемблерною вставкою, намагаючись досягти, якщо можливо, економії часу виконання і/або пам'яті.

3) Переконатись у правильності функціонування модифікованої програми testxxxx.pas шляхом виведення на екран результатів та їх порівняння з результатами немодифікованої програми.

Таблиця 2-1.4

Варіанти завдання

1.	**2.**

Program test001;
Var
  i,j,k: integer;
  A:array[3..11] of integer;
Begin
  k:=1;
  i:=3;
  repeat
  j:=i+k*2;
  if j>11
  then A[i]:=j
  else
  begin
	Program test002;
	Var
	  i,j,k: integer;
	  A:array[2..11] of integer;
	Begin
	  k:=1;
	  i:=2;
	  while i<=11 do
	  begin
	  j:=k+i*2;
	  if j>19
	  then A[i]:=j
	  else

Продовження табл. 2-1.4

```
      k:=k+1;
      A[i]:=k;
    end;
  inc(i);
 until i>11;
 for i:=2 to 10 do write(A[i]:4);
 writeln;
end.
```

```
      begin
        k:=k+3;
        A[i]:=k;
      end;
    inc(i);
  end;
 for i:=2 to 11 do
 write(A[i]:4);writeln;
end.
```

3.

```
Program test003;
Var
 i,j,k: integer;
 A:array[3..11] of integer;
Begin
 k:=1;
 for i:=3 to 11 do
   begin
     j:=k+i*3;
     if j>18
       then A[i]:=j
     else
       begin
         k:=k+2;
         A[i]:=k;
       end;
   end;
 for i:=3 to 11 do
   write(A[i]:4);writeln;
end.
```

4.

```
Program test004;
Var i,d:integer;
 A:array [4..11] of integer;
Begin
 d:=1;
 i:=11;
 repeat
   d:= i mod 2;
    if d=0 then
      A[i]:=i
    else A[i]:=-i;
     dec(i);
 until i<2;
 for i:=2 to 7 do
   write(A[i]:4);writeln;
End.
```

5.

```
Program test005;
Var i,d:integer;
  A:array [5..11] of integer;
Begin
  d:=1;
   i:=5;
 while i<=11 do
   begin
     A[i]:=i or d;
     d:=d + A[i];
      if d>10 then A[i]:=127-i;
        inc(i);
   end;
 for i:=5 to 11 do
    write(A[i]:4);writeln;
 End.
```

6.

```
Program test006;
Var
 p,k:integer;
A:array[6..13] of integer;
Begin
   k:=1;
   p:=6;
 repeat
     if p< 10 then
        k:=k or p
     else k:=k and p;
        A[p]:=k;
        p:=p+1;
until p>=14;
    for p:=6 to 13 do
    write(A[p]:4);writeln;
 End.
```

7.

```
Program test007;
Var  a1,c:integer;
       b:byte;
       A:array[7..13] of integer;
Begin
  a1:=3;
  c:=0;
  for  b:=7 to 13 do
   begin
     c:=c+3;
      if c<9 then
          a1:=5*c+b+1
       else a1:=b shl 2;
          A[b]:=a1;
   End;
for b:=7 to 13 do
     write(A[b]:4);writeln;
End.
```

8.

```
Program test008;
 Var  a1,c:integer;
       b:byte;
      A:array[8..20] of integer;
Begin
  A1:=2;
  b:=20;
  while  b>=8 do
   begin
     if b<15 then
       c:=a1*2
     else c:=c-a1;
     A[b]:=c;
    A1:=a1+b;
   dec(b);
  end;
for b:=8 to 20 do
     write(A[b]:4);writeln;
End.
```

9.

```
Program test009;
 Var  a,c:integer;
       b:byte;
       A1:array[9..20] of integer;
Begin
  a:=12;
  b:=9;
  repeat
     if a>b then  c:=a*3-b
     else c:= a*2+b;
      A1[b]:=c;
      inc(b);
  until b=21;
 for b:=9 to 20 do
     write(A1[b]:4);writeln;
 End.
```

10.

```
Program test0010;
Var  l,k:integer;
       j:byte;
     A:array[0..10] of integer;
Begin
  l:=2;
  k:=0;
   for  j:=0 to 10 do
  begin
       A[j]:=k;
      if  (j<2) or (j=4) then   k:=l+k
      else k:=l-k;
   end;
    for l:=0 to 10 do
     write(A[l]:4);writeln;
 End.
```

11.

```
Program test0011;
Var  k:integer;
       j:byte;
      A:array[1..11] of integer;
Begin
  k:=3;  j:=1;
  while j<=11 do
   begin
     if  j<>7 then
```

12.

```
Program test0012;
Var  l,k:integer;
       j:byte;
      A:array[2..12] of integer;
Begin
  l:=2;   k:=64;
  j:=12;
  repeat
     if  (j mod 4)=0  then
```

```
    begin
      k:=k+j;
    end;
    A[j]:=k;
    j:=j+1;
 end;
 for j:=1 to 11 do
    write(A[j]:4);writeln;
End.
```

```
      k:=k div l
    else k:=k+l;
      A[j]:=k;
      dec(j);
 until j<2;
for j:=2 to 12 do
    write(A[j]:4);writeln;
End.
```

13.

```
Program test0013;
Var p,j:integer;
      A: array [3..13] of integer;
Begin
  for p:=3 to 13 do
  begin
     A[p]:=4*p;
     If A[p]<9 then A[p]:=A[p]+1;
  End;
for p:=3 to 13 do
    write(A[p]:4);writeln;
End.
```

14.

```
Program  test0014;
Var i,d:byte;
  A:array [2..5] of integer;
Begin
  d:=1;
  for i:=2 to 5 do
   begin
     A[i]:=d*I+1;
     d:= A[i]-d+1;
       if d>3 then  d:=3;
   end;
for i:=2 to 5 do
    write(A[i]:4);writeln;
End.
```

15.

```
Program test0015;
Var  j:integer;
      A:array [5..15] of byte;
Begin
     j:=7 ;
    A[6]:=1;    A[5]:=1;
    repeat
      A[j]:=A[j-1]+A[j-2];
        If A[j]>127
       Then A[j]:=0;
      inc(j);
   until j>15;
  for j:=5 to 15 do
     write(A[j]:4);writeln;
End.
```

16.

```
Program  test0016;
Var i,s:integer;
  A:array [6..16] of integer;
Begin
 s:=0;
 for i:=6 to 16 do
    begin
      A[i]:=2*i;
      s:=s+A[i];
       if s>10 then A[i]:=s;
    end;
for i:=6 to 16 do
     write(A[i]:4);writeln;
End.
```

17.

```
Program test0017;
Var i,s:integer;
  A:array [7..17]of integer;
Begin
 s:=0; i:=17
```

18.

```
Program  test0018;
Var i,s:integer;
  A:array [8..18] of integer;
Begin
 s:=0;
```

```
while i>=7 do
    begin
      A[i]:=2*i;
      s:=s+A[i];
        if s>15 then A[i]:=s;
      dec(i);
    end;
 for i:=7 to 17 do
     write(A[i]:4);writeln;
End.
```

```
i:=8;
  repeat
      A[i]:=2*i;
      s:=s+A[i];
         If s>80 then
      A[i]:=s;
      inc(i);
  until i>18;
 for i:=8 to 18 do
     write(A[i]:4);writeln;
End.
```

19.

```
Program  test0019;
Var i,m:byte;
   A:array [9..19] of byte;
Begin
   m:=1;
   for i:=9 to 19 do
   begin
       m:=(m xor i) and 1;
       if m=1 then
          A[i]:=2*i
     Else A[i]:=2*i+1;
    end;
 for i:=9 to 19 do
     write(A[i]:4);writeln;
End.
```

20.

```
Program test0020;
Var i,m:integer;
   A:array [10..20] of integer;
Begin
   m:=1;
   i:=10;
 while i<=20 do
    begin
      A[i]:=3*i;
      m:=m+A[i];
         if m>100 then A[i]:=2*i;
      inc(i);
    end;
 for i:=10 to 20 do
     write(A[i]:4);writeln;
End.
```

21.

```
Program test0021;
Var    i,m:integer;
   A:array [13..21] of integer;
Begin
   m:=0;
   i:=21;
   repeat
      m:=m+i;
        if m<50 then
          A[i]:=5*i
       Else A[i]:=m;
     dec(i)
    until i<13;
for i:=13 to 21 do
     write(A[i]:4);writeln;
End.
```

22.

```
Program  test0022;
Var i,r:integer;
   A:array [13..22] of integer;
Begin
  r:=0;
  for i:=13 to 22 do
  begin
      r:=r+i*2;
       if r<100 then
         A[i]:=r
      Else
         A[i]:=r and i;
  end;
 for i:=13 to 22 do
     write(A[i]:4);writeln;
End.
```

23.

```
Program test0023;
Var i,r:integer;
   A:array [13..23] of integer;
Begin
  r:=100;
  i:=23;
  while i>=13 do
  begin
       A[i]:=2*i;
       r:=r-A[i];
         if r<0 then
           A[i]:=-A[i]+1;
       dec(i);
  end;
 for i:=13 to 23 do
      write(A[i]:4);writeln;
End.
```

24.

```
Program  test0024;
Var i,r:integer;
   A:array [14..24] of integer;
Begin
  r:=0; i:=15;
  A[14]:=4;
  repeat
       A[i]:=A[i-1]+r;
       r:=r or i;
          if r>21 then
             A[i]:=r;
       inc(i);
  until i>24;
for i:=14 to 24 do
      write(A[i]:4);writeln;
End.
```

25.

```
Program test0025;
Var  j:integer;
       A:array [4..14] of integer;
Begin
     j:=5;
    A[4]:=0;
    while  j<=14 do
    begin
       if A[j-1]<5 then
         A[j]:=A[j-1]+j;
       inc(j);
     end;
for j:=4 to 14 do
      write(A[j]:4);writeln;
   end.
```

26.

```
Program test0026;
Var
 p,i:integer;
A:array[16..26] of byte;
Begin
   p:=16;
   i:=0;
  while p<26 do
  begin
      if p>=20 then
           i:=i or p
      else i:=i and p;
      p:=p+1;
      A[p]:=i;
  end;
    for i:=5 to 10 do
      write(A[i]:4);writeln;
End.
```

27.

```
Program test0027;
Var
p,i:integer;
A:array[2..7] of word;
Begin
  p:=6;
  for i:=7 downto 2 do
  begin
     if i<6 then
        p:=p+1
```

28.

```
Program test0028;
Var
i,j:integer;
A:array[2..8] of integer;
Begin
  i:=2;
  j:=3;
  repeat
      if j<5 then
         i:=i+2*j
```

Продовження табл. 2-1.4

else p:=p-1;
A[i]:=p;
end;
for i:=2 to 7 do
write(A[i]:4);writeln;
End.
	else i:= i+j;
	A[j]:=i;
	j:=j+1;
	until j>8;
	for i:=2 to 8 do
	write(A[i]:4);writeln;
	End.

29.
Program test0029;
Var
i,j:integer;
A:array[2..10] of integer;
Begin
i:=0;
j:=2;
while i<=20 do
begin
i:=i+2;
j:=j+1;
if i<17 then
A[j]:=i
else A[j]:=3;
end;
for i:=2 to 10 do
write(A[i]:4);writeln;
End.
	30.
	Program test0030;
	Var
	i,k:integer;
	A:array[1..9] of word;
	Begin
	k:=0;
	for i:=1 to 9 do
	begin
	k:=k+i;
	if k<9 then
	A[i]:=i
	else A[i]:=k;
	end;
	for l:=1 to 9 do
	write(A[i]:4);writeln;
	End.

2-1.5. Контрольні запитання

1. Що відображає мовою Асемблера ідентифікатор змінної мови Паскаль?

2. Як реалізуються мовою Асемблера оператори присвоєння?

3. Як реалізуються мовою Асемблера оператори **if...then** ?

4. Як реалізуються мовою Асемблера оператори **for**?

5. Як реалізуються мовою Асемблера оператори **while**?

6. Як реалізуються мовою Асемблера оператори **repeat..until**?

7. Які вимоги ставлять до асемблерних вставок у програмах мовою Паскаль?

ЛАБОРАТОРНА РОБОТА №2-2

Реалізація основних програмних конструктів мовою Асемблера. Використання асемблерних вставок у програмах мовою С++

Мета роботи – вивчення методів реалізації мовою Асемблера основних виконавчих операторів мови С++, ознайомлення з методикою включення текстів програм мовою Асемблера в програми мовою С++.

2-2.1. Зміст роботи

Робота виконується на двох заняттях. На першому занятті на основі програми мовою С++ студенти створюють файл, що містить результати трансляції кожного С++ оператора на мову Асемблера, вивчають методи реалізації на Асемблері найуживаніших операторів мови С++. На другому занятті оформлюють у програмі мовою С++ асемблерну вставку, що оптимізує, якщо можливо, С++ програму за обсягом і/або швидкодією.

2-2.2. Теоретичні відомості

Основні відомості щодо подання даних в ЕОМ та елементарних операцій над ними, операндів машинних команд та способів їх адресації викладені у теоретичних відомостях Лабораторної роботи №2-1.

2-2.3. Рекомендації до виконання роботи

Знання мови С++ може допомогти у вивченні мови Асемблера. Знаючи оператори мови С++, а також маючи результати трансляції С++ операторів на мову Асемблера, не важко зрозуміти, ЯКИМИ САМЕ машинними командами реалізуються окремі С++ оператори.

Для цього, по-перше, потрібно створити файл, у якому після кожного С++ оператора містились би машинні інструкції – результати трансляції С++ оператора на мову Асемблера. На відміну, зокрема, від інтегрованого

середовища ТурбоПаскаль, середовище розробки Microsoft Visual Studio (компілятор C++) має режим генерації такого файлу. Розглянемо процес його створення на прикладі нескладної програми мовою C++.

Приклад програми test2011.cpp

```
/*1*/  #include <stdio.h>
/*2*/  int vec[10];
/*3*/  int s;
/*4*/  int main(){
/*5*/    int i;
/*6*/    s=0;
/*7*/    for(i=0;i<10;i++){
/*8*/              vec[i]=2*i;
/*9*/              s+=vec[i];
/*10*/   if(s>10)
/*11*/   vec[i]=s;
/*12*/ }
/*13*/
/*14*/  for(i=0;i<10;i++)
/*15*/  printf("%d ",vec[i]);
/*16*/  printf("\n");
/*17*/  return 0;
/*18*/  }
```

І почнемо зі створенння проекту C++ програми.

Створення проекту у середовищі Visual Studio

Для створення нового проекту необхідно у головному вікні Visual Studio вибрати та перейти на меню **File -> New ->Project**, далі вибрати **Win32 Console Application**, а в полі **Name** ввести назву проекту, наприклад, test2011. Після натискання **ОК** з'являється нове вікно, в якому треба перейти на вкладку **Application Settings** і відмітити галочкою поле **Empty project**, після чого натиснути кнопку **Finish**. Новий проект буде створений.

Далі необхідно додати наш файл test2011.cpp до проекту. Для цього потрібно перейти в **Solution Explorer** і зробити правий клік на вкладці **Source Files**. У випадаючому меню вибрати **Add->New Item…**, далі – **C++ File (.cpp)** і у полі **Name** ввести назву файлу test2011.cpp . Натиснувши кнопку Add, додамо файл до проекту.

Запустимо програму на виконання (**Ctrl + F5**). Отримаємо в консолі наступний результат виконання нашої програми:

0 2 4 12 20 30 42 56 72 90

Результат нам далі знадобиться для перевірки коректності роботи програми з асемблерною вставкою

Створення файлу з результатами трансляції програми

Для створення файла лістингу, що містить асемблерний, машинний та оригінальний код C++ програми (*файлу з асемблерним кодом*) необхідно перейти на вкладку **Solution Explorer** і клікнути правою клавішею миші на назві проекту. У випадаючому списку перейти на пункт **Properties**. З'явиться віконце, в якому треба вибрати випадаючий список **C/C++** і перейти на пункт **Output Files**. Праворуч вибрати пункт **Assembler Output** і з випадаючого списку – опцію **Assembly, Machine Code and Source (/FAcs),** а потім – натиснути кнопку **ОК**.

Далі запустимо програму на виконання (**Ctrl + F5**). Відкривши папку проекту, перейдемо в папку, назва якої співпадає з назвою проекту, а після цього – в папку **Debug**. У разі, якщо компіляція та запуск програми були виконані успішно, в даній папці буде створений файл **<name>.cod** , де <name> – ім'я ***.cpp** файлу, котрий містить код C++ програми.

За замовчуванням, файли проектів зберігаються в папці

C:\Users\<user_name>\Documents\Visual Studio 2008\Projects

де <user_name> – імя облікового запису користувача в системі.

Тоді шуканий файл буде знаходитись в наступній папці: C:\Users\<user_name>\Documents\VisualStudio2008\Projects\test2011\test2011\Debug\test2011.cod

Відкривши файл test2011.cod будь-яким текстовим редактором, наприклад, Nodepad, можемо прочитати асемблерний код. Для нашого прикладу C++ програми він буде мати наведений нижче вигляд:

Файл test2011.cod

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 15.00.21022.08

TITLE
c:\Users\<user_name>\Documents\VisualStudio2008\Projects\test2011\test2011\test2011.cpp
.686P
.XMM
include listing.inc
.model flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC      ?vec@@3PAHA                                   ; vec
PUBLIC      ?s@@3HA                                       ; s
_BSS    SEGMENT
?vec@@3PAHA DD  0aH DUP (?)                               ; vec
?s@@3HA     DD      01H DUP (?)                           ; s
_BSS    ENDS
PUBLIC      ??_C@_01EEMJAFIK@?6?$AA@                      ; `string'
PUBLIC      ??_C@_03JDANDILB@?$CFd?5?$AA@                 ; `string'
PUBLIC      _main
EXTRN       __imp__printf:PROC
EXTRN       __RTC_CheckEsp:PROC
EXTRN       __RTC_Shutdown:PROC
EXTRN       __RTC_InitBase:PROC
;COMDAT
??_C@_01EEMJAFIK@?6?$AA@
;File c:\users\<user_name>\documents\visual studio 2008\projects\test2011\test2011\test2011.cpp
CONST       SEGMENT
??_C@_01EEMJAFIK@?6?$AA@ DB 0aH, 00H                      ; `string'
CONST       ENDS
;COMDAT ??_C@_03JDANDILB@?$CFd?5?$AA@
CONST       SEGMENT
??_C@_03JDANDILB@?$CFd?5?$AA@ DB '%d ', 00H               ; `string'
CONST       ENDS
;COMDAT rtc$TMZ
rtc$TMZ     SEGMENT
__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ     ENDS
;COMDAT rtc$IMZ
rtc$IMZ     SEGMENT
__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
; Function compile flags: /Odtp /RTCsu /ZI
rtc$IMZ     ENDS
;COMDAT _main
_TEXT       SEGMENT
_i$ = -8                                                  ; size = 4
_main  PROC                                               ; COMDAT
; 5    : {       int i;
00000  55            push    ebp
00001  8b ec         mov     ebp, esp
```

```
00003  81 ec cc 00 00 00 sub  esp, 204                          ; 000000ccH
00009  53                 push   ebx
0000a  56                 push   esi
0000b  57                 push   edi
0000c  8d bd 34 ff ff ff lea    edi, DWORD PTR [ebp-204]
00012  b9 33 00 00 00     mov    ecx, 51                         ; 00000033H
00017  b8 cc cc cc cc  mov    eax, -858993460                  ; ccccccccH
0001c  f3 ab              rep stosd
; 6    :  s=0;
0001e          c7 05 00 00 00
00 00 00 00 00          mov    DWORD PTR ?s@@3HA, 0             ; s
; 7    :  for(i=0;i<10;i++){
00028  c7 45 f8 00 0000 00    mov    DWORD PTR _i$[ebp], 0
0002f  eb 09              jmp    SHORT $LN7@main
$LN6@main:
00031  8b 45 f8          mov    eax, DWORD PTR _i$[ebp]
00034  83 c0 01          add    eax, 1
00037  89 45 f8          mov    DWORD PTR _i$[ebp], eax
$LN7@main:
0003a  83 7d f8 0a        cmp    DWORD PTR _i$[ebp], 10           ; 0000000aH
0003e  7d 40              jge    SHORT $LN5@main
; 8    :          vec[i]=2*i;
00040  8b 45 f8          mov    eax, DWORD PTR _i$[ebp]
00043  d1 e0              shl    eax, 1
00045  8b 4d f8          mov    ecx, DWORD PTR _i$[ebp]
00048  89 04 8d 00 0000 00  mov    DWORD PTR ?vec@@3PAHA[ecx*4], eax
; 9    :          s+=vec[i];
0004f  8b 45 f8          mov    eax, DWORD PTR _i$[ebp]
00052  8b 0d 00 00 00 00 mov ecx, DWORD PTR ?s@@3HA                ; s
00058  03 0c 85 00 00 00 00 add ecx, DWORD PTR ?vec@@3PAHA[eax*4]
0005f  89 0d 00 00 00 00 mov DWORD PTR ?s@@3HA, ecx                ; s
; 10   :          if(s>10)
00065  83 3d 00 00 00 00 0a cmp DWORD PTR ?s@@3HA, 10              ; s, 0000000aH
0006c  7e 10              jle    SHORT $LN4@main
; 11   : vec[i]=s;
0006e  8b 45 f8          mov    eax, DWORD PTR _i$[ebp]
00071  8b 0d 00 00 00 00 mov ecx, DWORD PTR ?s@@3HA                ; s
00077  89 0c 85 00 00 00 00  mov    DWORD PTR ?vec@@3PAHA[eax*4], ecx
$LN4@main:
; 12   : }
0007e          eb b1              jmp    SHORT $LN6@main
$LN5@main:
; 13   :
; 14   : for(i=0;i<10;i++)
00080  c7 45 f8 00 00
00 00             mov    DWORD PTR _i$[ebp], 0
00087  eb 09              jmp    SHORT $LN3@main
$LN2@main:
00089  8b 45 f8          mov    eax, DWORD PTR _i$[ebp]
0008c          83 c0 01           add    eax, 1
0008f          89 45 f8           mov    DWORD PTR _i$[ebp], eax
$LN3@main:
```

```
00092  83 7d f8 0a       cmp    DWORD PTR _i$[ebp], 10                    ; 0000000aH
00096  7d 24             jge    SHORT $LN1@main
; 15   : printf("%d ",vec[i]);
00098  8b f4             mov    esi, esp
0009a  8b 45 f8          mov    eax, DWORD PTR _i$[ebp]
0009d  8b 0c 85 00 00 00 00  mov     ecx, DWORD PTR ?vec@@3PAHA[eax*4]
000a4  51                push   ecx
000a5  68 00 00 00 00 push   OFFSET ??_C@_03JDANDILB@?$CFd?5?$AA@
000aa  ff 15 00 00 00 00 call  DWORD PTR __imp__printf
000b0  83 c4 08          add    esp, 8
000b3  3b f4             cmp    esi, esp
000b5  e8 00 00 00 00 call     __RTC_CheckEsp
000ba  eb cd             jmp    SHORT $LN2@main
$LN1@main:
; 16   : printf("\n");
000bc  8b f4             mov    esi, esp
000be  68 00 00 00 00 push   OFFSET ??_C@_01EEMJAFIK@?6?$AA@
000c3  ff 15 00 00 00 00call   DWORD PTR __imp__printf
000c9  83 c4 04          add    esp, 4
000cc  3b f4             cmp    esi, esp
000ce  e8 00 00 00 00 call     __RTC_CheckEsp
; 17   : return 0;
000d3  33 c0             xor    eax, eax
; 18   : }
000d5  5f                pop    edi
000d6  5e                pop    esi
000d7  5b                pop    ebx
000d8  81 c4 cc 00 00 00 add  esp, 204                                   ; 000000ccH
000de  3b ec             cmp    ebp, esp
000e0  e8 00 00 00 00 call     __RTC_CheckEsp
000e5  8b e5             mov    esp, ebp
000e7  5d                pop    ebp
000e8  c3                ret    0
_main  ENDP
_TEXT          ENDS
END
```

Ознайомлення з реалізацією C++ операторів мовою Асемблера

Проаналізуємо отриманий лістинг. Для кращого розуміння, він містить вказівки номерів та змісту рядків – операторів C++ програми. Нижче зліва направо можемо прочитати відносні адреси, машинний код (16-кове подання) і команди мовою Асемблера, що реалізують вказаний оператор.

ПРИМІТКА. ***Жирним*** *шрифтом виділено ті рядки лістингу, які стосуються визначення основних змінних програми, а також рядки, що являють собою реалізацію операторів основного* for *циклу програми. Ці рядки*

слід особливо уважно проаналізувати для подальшого використання при створенні асемблерної вставки.

Тепер за допомогою довідника з системи команд мікропроцесора (наприклад, з опису комад в [2], необхідно ознайомитись з командами Асемблера, які реалізують відповідний оператор С++ програми, а також зрозуміти (вивчити) методику реалізації цих операторів.

Асемблерна вставка

Уважно розглянувши лістинг та розібравшись зі способами реалізації конструкцій та операторів мови С++ мовою Асемблера, необхідно вибрати фрагмент програми мовою С++ для заміни його асемблерним кодом – асемблерною вставкою. Виберемо з лістингу відповідний фрагмент асемблерного коду та використаємо його як вміст асемблерної вставки.

Для внесення в текст С++ програми фрагмента, написаного мовою Асемблера (асемблерної вставки), необхідно цей асемблерний код помістити в блок:

```
_asm {
<assembler code>
}
```

Крім того, слід врахувати, що використання деяких символів заборонене в мові С++ (наприклад, символ @ за замовчуванням використовується в мітках асемблерного коду, але не розпізнається в асемблерній вставці програми мовою С++).

Для роботи з користувацькими змінними в асемблерній вставці слід залишити ті ж самі ідентифікатори, як і в оригінальному коді С++ програми (тобто, наприклад, замість ідентифікатора змінної ?s@@3HA, що використовується в асемблерному коді сформованого компілятором лістингу, в асемблерній вставці слід записати ідентифікатор s).

Виконавши вищезазначені дії, отримаємо наступну асемблерну вставку:

```
#include <stdio.h>
int vec[10];
int s;
int main(){
int i;
s=0;
for(i=0;i<10;i++){
__asm{
; 10  :          vec[i]=2*i;
mov     EAX, i                                  ; i
shl     EAX, 1
mov     ECX, i                                  ; i
mov     vec[ECX*4], EAX
; 11  :          s+=vec[i];
mov     EAX, i                                  ; i
mov     ECX, s                                  ; s
add     ECX, vec[EAX*4]
mov     s, ECX                                  ; s
; 12  :          if(s>10)
cmp     s, 10
jle     SHORT LN4main
; 13  :                  vec[i]=s;
mov     EAX, i                                  ; i
mov     ECX, s                                  ; s
mov     vec[EAX*4], ECX
LN4main:
}
}
for(i=0;i<10;i++)
printf("%d ",vec[i]);
printf("\n");
return 0;
}
```

Модифікувавши програму, необхідно знову запустити її на виконання та впевнитись в тому, що програма працює так само, як і до заміни частини оригінального коду асемблерною вставкою. Уважно прочитавши та проаналізувавши операції, котрі виконує вставка, можемо помітити, що певні моменти доцільно оптимізувати з метою пришвидшення роботи програми та економії пам’яті.

В результаті можемо отримати наступний код:

```
#include <stdio.h>
int vec[10];
int s;
int main(){
int i,s;
s=0;
for(i=0;i<10;i++){
```

```
_asm{
// vec[i]=2*i;
mov    EAX, i
mov   ECX,EAX
shl      EAX, 1
shl      ECX, 2
mov    vec[ECX], EAX
// 10   :                  s+=vec[i];
mov    EAX, s
add     EAX, vec[ECX]
mov    s, EAX
// 11   :                  if(s>10)
cmp    s, 10
jle      SHORT esc_if
// 12   :                  vec[i]=s;
mov    vec[ECX], EAX
esc_if:
}
}
for(i=0;i<10;i++)
printf("%d ",vec[i]);
printf("\n");
return 0;
}
```

Навіть на прикладі невеликої програми можна помітити, що, завдяки видаленню в асемблерній вставці «зайвих» операцій або їх заміни на швидкіші аналоги, досягається певна економія пам’яті та пришвидшення програми.

2-2.4. Приклад реалізації різних конструкцій циклів та розгалужень мовою Асемблера

Розглянемо приклади реалізації різних конструкцій C++ циклів та розгалужень мовою Асемблера.

ПРИМІТКА. В прикладах виконується лише створення асемблерної вставки на основі коду з лістингу без подальшої її оптимізації.

Для прикладу використаємо наступний C++ код:

```
/*1*/   #include <stdio.h>
/*2*/
/*3*/   char i1,i3,s4;
/*4*/   int A1[15], A2[8], A4[10];
/*5*/
/*6*/   int main(){
/*7*/             int j1,i2,j3,i4;
/*8*/             char j2,A3[9];
```

```
/*9*/
/*10*/                                          //example #1
/*11*/      i1=7;
/*12*/      j1=0;
/*13*/      while(j1<15){
/*14*/            if((j1>4) && (j1<11))
/*15*/                  i1=4*j1;
/*16*/            A1[j1]=i1;
/*17*/            j1++;
/*18*/      }
/*19*/
/*20*/                                          //example #2
/*21*/      i2=0;
/*22*/      j2=7;
/*23*/      do{
/*24*/            if(i2>=3)
/*25*/                  j2=3*i2-5;
/*26*/            else
/*27*/                  j2=j2+i2+2;
/*28*/            A2[i2]=j2;
/*29*/            i2++;
/*30*/      }while(i2<8);
/*31*/
/*32*/                                          //example #3
/*33*/      for(j3=0;j3<9;j3++){
/*34*/            i3=2*j3;
/*35*/            switch(j3){
/*36*/                  case 3:i3+=10; break;
/*37*/                  case 5: i3*=2; break;
/*38*/                  case 7: i3-=4; break;
/*39*/                  default: i3++;
/*40*/            }
/*41*/            A3[j3]=i3;
/*42*/      }
/*43*/
/*44*/                                          //example #4
/*45*/      s4=0;
/*46*/      for(i4=0;i4<10;i4++){
/*47*/            A4[i4]=2*i4;
/*48*/            s4+=A4[i4];
/*49*/            if(s4>10)
/*50*/                  A4[i4]=s4;
/*51*/            else
/*52*/                  A4[i4]=i4;
/*53*/      }
/*54*/
/*55*/                                          //output
/*56*/      for(j1=0;j1<15;j1++)
/*57*/            printf("%d ",A1[j1]);
/*58*/      printf("\n\n");
/*59*/
/*60*/      for(i2=0;i2<8;i2++)
```

```
/*61*/                  printf("%d ",A2[i2]);
/*62*/          printf("\n\n");
/*63*/
/*64*/          for(j3=0;j3<9;j3++)
/*65*/                  printf("%d ",A3[j3]);
/*66*/          printf("\n\n");
/*67*/
/*68*/          for(i4=0;i4<10;i4++)
/*69*/                  printf("%d ",A4[i4]);
/*70*/          printf("\n\n");
/*70*/          return 0;
/*71*/  }
```

Виконаємо описані вище дії для створення файлу лістингу. Він матиме наступний вигляд:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 15.00.21022.08

TITLE c:\Users\NRG\Documents\Visual Studio 2008\Projects\1312\1312\main.cpp
.686P
.XMM
include listing.inc
.model flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC      ?s4@@3DA                                ; s4
PUBLIC      ?A1@@3PAHA                              ; A1
PUBLIC      ?i1@@3DA                                ; i1
PUBLIC      ?A2@@3PAHA                              ; A2
PUBLIC      ?A4@@3PAHA                              ; A4
PUBLIC      ?i3@@3DA                                ; i3
_BSS   SEGMENT
?s4@@3DA DB       01H DUP (?)                       ; s4
ALIGN       4

?A1@@3PAHA DD  0fH DUP (?)                          ; A1
?i1@@3DA DB       01H DUP (?)                       ; i1
ALIGN       4

?A2@@3PAHA DD  08H DUP (?)                          ; A2
?A4@@3PAHA DD  0aH DUP (?)                          ; A4
?i3@@3DA DB       01H DUP (?)                       ; i3
_BSS   ENDS
PUBLIC      ??_C@_02PHMGELLB@?6?6?$AA@              ; `string'
PUBLIC      ??_C@_03JDANDILB@?$CFd?5?$AA@           ; `string'
PUBLIC      __$ArrayPad$
PUBLIC      _main
EXTRN       __imp__printf:PROC
EXTRN       ___security_cookie:DWORD
```

```
EXTRN	@__security_check_cookie@4:PROC
EXTRN	@_RTC_CheckStackVars@8:PROC
EXTRN	__RTC_CheckEsp:PROC
EXTRN	__RTC_Shutdown:PROC
EXTRN	__RTC_InitBase:PROC
;COMDAT
??_C@_02PHMGELLB@?6?6?$AA@
;File c:\users\nrg\documents\visual studio 2008\projects\1312\1312\main.cpp
CONST	SEGMENT
??_C@_02PHMGELLB@?6?6?$AA@ DB 0aH, 0aH, 00H		; `string'
CONST	ENDS
;COMDAT ??_C@_03JDANDILB@?$CFd?5?$AA@
CONST	SEGMENT
??_C@_03JDANDILB@?$CFd?5?$AA@ DB '%d ', 00H		; `string'
CONST	ENDS
;COMDAT rtc$TMZ
rtc$TMZ	SEGMENT
__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ	ENDS
;COMDAT rtc$IMZ
rtc$IMZ	SEGMENT
__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
; Function compile flags: /Odtp /RTCsu /ZI
rtc$IMZ	ENDS
;COMDAT _main
_TEXT	SEGMENT
tv90 = -280                                             ; size = 4
_A3$ = -80                                              ; size = 9
_j2$ = -57                                              ; size = 1
_i4$ = -48                                              ; size = 4
_j3$ = -36                                              ; size = 4
_i2$ = -24                                              ; size = 4
_j1$ = -12                                              ; size = 4
__$ArrayPad$ = -4                                       ; size = 4
_main	PROC                                            ; COMDAT
; 6	: /*6*/	int main(){
00000	55		push	ebp
00001	8b ec		mov	ebp, esp
00003	81 ec 18 01 00 00	sub	esp, 280                        ; 00000118H
00009	53		push	ebx
0000a	56		push	esi
0000b	57		push	edi
0000c	8d bd e8 fe ff ff	lea	edi, DWORD PTR [ebp-280]
00012	b9 46 00 00 00	mov	ecx, 70                         ; 00000046H
00017	b8 cc cc cc cc	mov	eax, -858993460                 ; ccccccccH
0001c	f3 ab		rep stosd
0001e	a1 00 00 00 00	mov	eax, DWORD PTR ___security_cookie
00023	33 c5		xor	eax, ebp
00025	89 45 fc	mov	DWORD PTR __$ArrayPad$[ebp], eax
; 7	: /*7*/	int	j1,i2,j3,i4;
; 8	: /*8*/ char j2,A3[9];
; 9	: /*9*/
```

```
; 10   : /*10*/   //example #1
; 11   : /*11*/            i1=7;
00028  c6 05 00 00 00
00 07             mov     BYTE PTR ?i1@@3DA, 7                    ; i1
; 12   : /*12*/            j1=0;
0002f  c7 45 f4 00 00 00 00   mov     DWORD PTR _j1$[ebp], 0
$LN34@main:
; 13   : /*13*/  while(j1<15){
00036  83 7d f4 0f      cmp     DWORD PTR _j1$[ebp], 15         ; 0000000fH
0003a  7d 33            jge     SHORT $LN33@main
; 14   : /*14*/   if((j1>4)&& (j1<11))
0003c  83 7d f4 04      cmp     DWORD PTR _j1$[ebp], 4
00040  7e 11            jle     SHORT $LN32@main
00042  83 7d f4 0b      cmp     DWORD PTR _j1$[ebp], 11         ; 0000000bH
00046  7d 0b            jge     SHORT $LN32@main
; 15   : /*15*/  i1=4*j1;
00048  8b 45 f4         mov     eax, DWORD PTR _j1$[ebp]
0004b  c1 e0 02         shl     eax, 2
0004e  a2 00 00 00 00 mov       BYTE PTR ?i1@@3DA, al           ; i1
$LN32@main:
; 16   : /*16*/  A1[j1]=i1;
00053  0f be 05 00 00 00 00   movsx           eax, BYTE PTR ?i1@@3DA        ; i1
0005a  8b 4d f4         mov     ecx, DWORD PTR _j1$[ebp]
0005d  89 04 8d 00 00 00 00   mov     DWORD PTR ?A1@@3PAHA[ecx*4], eax
; 17   : /*17*/            j1++;
00064  8b 45 f4         mov     eax, DWORD PTR _j1$[ebp]
00067  83 c0 01         add     eax, 1
0006a  89 45 f4         mov     DWORD PTR _j1$[ebp], eax
; 18   : /*18*/            }
0006d  eb c7            jmp     SHORT $LN34@main
$LN33@main:
; 19   : /*19*/
; 20   : /*20*/  //example #2
; 21   : /*21*/            i2=0;
0006f  c7 45 e8 00 00
00 00             mov     DWORD PTR _i2$[ebp], 0
; 22   : /*22*/            j2=7;
00076  c6 45 c7 07      mov     BYTE PTR _j2$[ebp], 7
$LN31@main:
; 23   : /*23*/            do{
; 24   : /*24*/            if(i2>=3)
0007a  83 7d e8 03      cmp     DWORD PTR _i2$[ebp], 3
0007e  7c 0e            jl      SHORT $LN28@main
; 25   : /*25*/            j2=3*i2-5;
00080  8b 45 e8         mov     eax, DWORD PTR _i2$[ebp]
00083  6b c0 03         imul    eax, 3
00086  83 e8 05         sub     eax, 5
00089  88 45 c7         mov     BYTE PTR _j2$[ebp], al
; 26   : /*26*/                    else
0008c  eb 0e            jmp     SHORT $LN27@main
$LN28@main:
; 27   : /*27*/  j2=j2+i2+2;
```

```
0008e  0f be 45 c7      movsx   eax, BYTE PTR _j2$[ebp]
00092  8b 4d e8         mov     ecx, DWORD PTR _i2$[ebp]
00095  8d 54 08 02      lea     edx, DWORD PTR [eax+ecx+2]
00099  88 55 c7         mov     BYTE PTR _j2$[ebp], dl
$LN27@main:
; 28   : /*28*/          A2[i2]=j2;
0009c  0f be 45 c7      movsx   eax, BYTE PTR _j2$[ebp]
000a0  8b 4d e8         mov     ecx, DWORD PTR _i2$[ebp]
000a3  89 04 8d 00 00 00 00  mov    DWORD PTR ?A2@@3PAHA[ecx*4], eax
; 29   : /*29*/          i2++;
000aa  8b 45 e8         mov     eax, DWORD PTR _i2$[ebp]
000ad  83 c0 01         add     eax, 1
000b0  89 45 e8         mov     DWORD PTR _i2$[ebp], eax
; 30   : /*30*/  }while(i2<8);
000b3  83 7d e8 08      cmp     DWORD PTR _i2$[ebp], 8
000b7  7c c1            jl      SHORT $LN31@main
; 31   : /*31*/
; 32   : /*32*/  //example #3
; 33   : /*33*/          for(j3=0;j3<9;j3++){
000b9  c7 45 dc 00 00 00 00  mov    DWORD PTR _j3$[ebp], 0
000c0  eb 09            jmp     SHORT $LN26@main
$LN25@main:
000c2  8b 45 dc         mov     eax, DWORD PTR _j3$[ebp]
000c5  83 c0 01         add     eax, 1
000c8  89 45 dc         mov     DWORD PTR _j3$[ebp], eax
$LN26@main:
000cb  83 7d dc 09      cmp     DWORD PTR _j3$[ebp], 9
000cf  0f 8d 80 00 00 00 jge    $LN24@main
; 34   : /*34*/          i3=2*j3;
000d5  8b 45 dc         mov     eax, DWORD PTR _j3$[ebp]
000d8  d1 e0            shl     eax, 1
000da  a2 00 00 00 00   mov     BYTE PTR ?i3@@3DA, al           ; i3
; 35   : /*35*/          switch(j3){
000df  8b 45 dc         mov     eax, DWORD PTR _j3$[ebp]
000e2  89 85 e8 fe ff ff mov    DWORD PTR tv90[ebp], eax
000e8  83 bd e8 fe ff ff 03 cmp         DWORD PTR tv90[ebp], 3
000ef  74 14            je      SHORT $LN21@main
000f1  83 bd e8 fe ff ff 05 cmp         DWORD PTR tv90[ebp], 5
000f8  74 1c            je      SHORT $LN20@main
000fa  83 bd e8 fe ff ff 07 cmp         DWORD PTR tv90[ebp], 7
00101  74 23            je      SHORT $LN19@main
00103  eb 32            jmp     SHORT $LN18@main
$LN21@main:
; 36   : /*36*/  case 3:i3+=10; break;
00105  0f be 05 00 00 00 00 movsx   eax, BYTE PTR ?i3@@3DA       ; i3
0010c  83 c0 0a         add     eax, 10                         ; 0000000aH
0010f  a2 00 00 00 00   mov     BYTE PTR ?i3@@3DA, al           ; i3
00114  eb 2d            jmp     SHORT $LN22@main
$LN20@main:
; 37   : /*37*/  case 5: i3*=2;break;
00116  0f be 05 00 00 00 00 movsx   eax, BYTE PTR ?i3@@3DA       ; i3
0011d  d1 e0            shl     eax, 1
```

```
0011f  a2 00 00 00 00 mov    BYTE PTR ?i3@@3DA, al          ; i3
00124  eb 1d          jmp    SHORT $LN22@main
$LN19@main:
; 38   : /*38*/                    case 7: i3-=4; break;
00126  0f be 05 00 00 00 00 movsx  eax, BYTE PTR ?i3@@3DA       ; i3
0012d  83 e8 04       sub    eax, 4
00130  a2 00 00 00 00 mov    BYTE PTR ?i3@@3DA, al          ; i3
00135  eb 0c          jmp    SHORT $LN22@main
$LN18@main:
; 39   : /*39*/  default: i3++;
00137  a0 00 00 00 00 mov    al, BYTE PTR ?i3@@3DA          ; i3
0013c  04 01          add    al, 1
0013e  a2 00 00 00 00       mov    BYTE PTR ?i3@@3DA, al          ; i3
$LN22@main:
; 40   : /*40*/                }
; 41   : /*41*/           A3[j3]=i3;
00143  8b 45 dc       mov    eax, DWORD PTR _j3$[ebp]
00146  8a 0d 00 00 00 00 mov       cl, BYTE PTR ?i3@@3DA; i3
0014c  88 4c 05 b0     mov    BYTE PTR _A3$[ebp+eax], cl
; 42   : /*42*/           }
00150  e9 6d ff ff ff  jmp    $LN25@main
$LN24@main:
; 43   : /*43*/
; 44   : /*44*/  //example #4
; 45   : /*45*/           s4=0;
00155  c6 05 00 00 00 00 00 mov     BYTE PTR ?s4@@3DA, 0                ; s4
; 46   : /*46*/           for(i4=0;i4<10;i4++){
0015c  c7 45 d0 00 00 00 00 mov    DWORD PTR _i4$[ebp], 0
00163  eb 09          jmp    SHORT $LN17@main
$LN16@main:
00165  8b 45 d0       mov    eax, DWORD PTR _i4$[ebp]
00168  83 c0 01       add    eax, 1
0016b  89 45 d0       mov    DWORD PTR _i4$[ebp], eax
$LN17@main:
0016e  83 7d d0 0a     cmp    DWORD PTR _i4$[ebp], 10        ; 0000000aH
00172  7d 53          jge    SHORT $LN15@main
; 47   : /*47*/                A4[i4]=2*i4;
00174  8b 45 d0       mov    eax, DWORD PTR _i4$[ebp]
00177  d1 e0          shl    eax, 1
00179  8b 4d d0       mov    ecx, DWORD PTR _i4$[ebp]
0017c  89 04 8d 00 00 00 00 mov    DWORD PTR ?A4@@3PAHA[ecx*4], eax
; 48   : /*48*/                s4+=A4[i4];
00183  0f be 05 00 00 00 00 movsx       eax, BYTE PTR ?s4@@3DA       ; s4
0018a  8b 4d d0       mov    ecx, DWORD PTR _i4$[ebp]
0018d  03 04 8d 00 00 0 00 add    eax, DWORD PTR ?A4@@3PAHA[ecx*4]
00194  a2 00 00 00 00       mov    BYTE PTR ?s4@@3DA, al          ; s4
; 49   : /*49*/                if(s4>10)
00199  0f be 05 00 00 00 00 movsx       eax, BYTE PTR ?s4@@3DA       ; s4
001a0  83 f8 0a        cmp    eax, 10                  ; 0000000aH
001a3  7e 13          jle    SHORT $LN14@main
; 50   : /*50*/                    A4[i4]=s4;
001a5  0f be 05 00 00 00 00 movsx       eax, BYTE PTR ?s4@@3DA       ; s4
```

```
001ac  8b 4d d0          mov    ecx, DWORD PTR _i4$[ebp]
001af  89 04 8d 00 00 00 00  mov    DWORD PTR ?A4@@3PAHA[ecx*4], eax
; 51   : /*51*/                    else
001b6  eb 0d             jmp    SHORT $LN13@main
$LN14@main:
; 52   : /*52*/                            A4[i4]=i4;
001b8  8b 45 d0          mov    eax, DWORD PTR _i4$[ebp]
001bb  8b 4d d0          mov    ecx, DWORD PTR _i4$[ebp]
001be  89 0c 85 00 00 00 00  mov    DWORD PTR ?A4@@3PAHA[eax*4], ecx
$LN13@main:
; 53   : /*53*/            }
001c5  eb 9e             jmp    SHORT $LN16@main
$LN15@main:
; 54   : /*54*/
; 55   : /*55*/            //output
; 56   : /*56*/            for(j1=0;j1<15;j1++)
001c7  c7 45 f4 00 00 00 00  mov    DWORD PTR _j1$[ebp], 0
001ce  eb 09             jmp    SHORT $LN12@main
$LN11@main:
001d0  8b 45 f4          mov    eax, DWORD PTR _j1$[ebp]
001d3  83 c0 01          add    eax, 1
001d6  89 45 f4          mov    DWORD PTR _j1$[ebp], eax
$LN12@main:
001d9  83 7d f4 0f       cmp    DWORD PTR _j1$[ebp], 15          ; 0000000fH
001dd  7d 24             jge    SHORT $LN10@main
; 57   : /*57*/                    printf("%d ",A1[j1]);
001df  8b f4             mov    esi, esp
001e1  8b 45 f4          mov    eax, DWORD PTR _j1$[ebp]
001e4  8b 0c 85 00 00 00 00  mov    ecx, DWORD PTR ?A1@@3PAHA[eax*4]
001eb  51                push   ecx
001ec  68 00 00 00 00    push   OFFSET ??_C@_03JDANDILB@?$CFd?5?$AA@
001f1  ff 15 00 00 00 00 call   DWORD PTR __imp__printf
001f7  83 c4 08          add    esp, 8
001fa  3b f4             cmp    esi, esp
001fc  e8 00 00 00 00 call   __RTC_CheckEsp
00201  eb cd             jmp    SHORT $LN11@main
$LN10@main:
; 58   : /*58*/            printf("\n\n");
00203  8b f4             mov    esi, esp
00205  68 00 00 00 00    push   OFFSET ??_C@_02PHMGELLB@?6?6?$AA@
0020a  ff 15 00 00 00 00 call   DWORD PTR __imp__printf
00210  83 c4 04          add    esp, 4
00213  3b f4             cmp    esi, esp
00215  e8 00 00 00 00 call   __RTC_CheckEsp
; 59   : /*59*/
; 60   : /*60*/            for(i2=0;i2<8;i2++)
0021a  c7 45 e8 00 00
00 00    mov    DWORD PTR _i2$[ebp], 0
00221  eb 09             jmp    SHORT $LN9@main
$LN8@main:
00223  8b 45 e8          mov    eax, DWORD PTR _i2$[ebp]
00226  83 c0 01          add    eax, 1
```

```
00229  89 45 e8          mov    DWORD PTR _i2$[ebp], eax
$LN9@main:
0022c  83 7d e8 08       cmp    DWORD PTR _i2$[ebp], 8
00230  7d 24             jge    SHORT $LN7@main
; 61   : /*61*/                  printf("%d ",A2[i2]);
00232  8b f4             mov    esi, esp
00234  8b 45 e8          mov    eax, DWORD PTR _i2$[ebp]
00237  8b 0c 85 00 00 00 00  mov    ecx, DWORD PTR ?A2@@3PAHA[eax*4]
0023e  51                push   ecx
0023f  68 00 00 00 00 push   OFFSET ??_C@_03JDANDILB@?$CFd?5?$AA@
00244  ff 15 00 00 00 00       call     DWORD PTR __imp__printf
0024a  83 c4 08          add    esp, 8
0024d  3b f4             cmp    esi, esp
0024f  e8 00 00 00 00 call     __RTC_CheckEsp
00254  eb cd             jmp    SHORT $LN8@main
$LN7@main:
; 62   : /*62*/           printf("\n\n");
00256  8b f4             mov    esi, esp
00258  68 00 00 00 00          push   OFFSET ??_C@_02PHMGELLB@?6?6?$AA@
0025d  ff 15 00 00 00 00       call     DWORD PTR __imp__printf
00263  83 c4 04          add    esp, 4
00266  3b f4             cmp    esi, esp
00268  e8 00 00 00 00 call     __RTC_CheckEsp
; 63   : /*63*/
; 64   : /*64*/           for(j3=0;j3<9;j3++)
0026d  c7 45 dc 00 00 00 00  mov    DWORD PTR _j3$[ebp], 0
00274  eb 09             jmp    SHORT $LN6@main
$LN5@main:
00276  8b 45 dc          mov    eax, DWORD PTR _j3$[ebp]
00279  83 c0 01          add    eax, 1
0027c  89 45 dc          mov    DWORD PTR _j3$[ebp], eax
$LN6@main:
0027f  83 7d dc 09       cmp    DWORD PTR _j3$[ebp], 9
00283  7d 22             jge    SHORT $LN4@main
; 65   : /*65*/                  printf("%d ",A3[j3]);
00285  8b 45 dc          mov    eax, DWORD PTR _j3$[ebp]
00288  0f be 4c 05 b0  movsx ecx, BYTE PTR _A3$[ebp+eax]
0028d  8b f4             mov    esi, esp
0028f  51                push   ecx
00290  68 00 00 00 00 push   OFFSET ??_C@_03JDANDILB@?$CFd?5?$AA@
00295  ff 15 00 00 00 00 call  DWORD PTR __imp__printf
0029b  83 c4 08          add    esp, 8
0029e  3b f4             cmp    esi, esp
002a0  e8 00 00 00 00 call     __RTC_CheckEsp
002a5  eb cf             jmp    SHORT $LN5@main
$LN4@main:
; 66   : /*66*/           printf("\n\n");
002a7  8b f4             mov    esi, esp
002a9  68 00 00 00 00 push   OFFSET ??_C@_02PHMGELLB@?6?6?$AA@
002ae  ff 15 00 00 00 00 call  DWORD PTR __imp__printf
002b4  83 c4 04          add    esp, 4
002b7  3b f4             cmp    esi, esp
```

```
002b9  e8 00 00 00 00 call     __RTC_CheckEsp
; 67   : /*67*/
; 68   : /*68*/            for(i4=0;i4<10;i4++)
002be  c7 45 d0 00 00 00 00  mov    DWORD PTR _i4$[ebp], 0
002c5  eb 09               jmp     SHORT $LN3@main
$LN2@main:
002c7  8b 45 d0            mov     eax, DWORD PTR _i4$[ebp]
002ca  83 c0 01            add     eax, 1
002cd  89 45 d0            mov     DWORD PTR _i4$[ebp], eax
$LN3@main:
002d0  83 7d d0 0a         cmp     DWORD PTR _i4$[ebp], 10          ; 0000000aH
002d4  7d 24               jge     SHORT $LN1@main
; 69   : /*69*/                    printf("%d ",A4[i4]);
002d6  8b f4               mov     esi, esp
002d8  8b 45 d0            mov     eax, DWORD PTR _i4$[ebp]
002db  8b 0c 85 00 00 00 00  mov    ecx, DWORD PTR ?A4@@3PAHA[eax*4]
002e2  51                  push    ecx
002e3  68 00 00 00 00 push   OFFSET ??_C@_03JDANDILB@?$CFd?5?$AA@
002e8  ff 15 00 00 00 00 call  DWORD PTR __imp__printf
002ee  83 c4 08            add     esp, 8
002f1  3b f4               cmp     esi, esp
002f3  e8 00 00 00 00 call     __RTC_CheckEsp
002f8  eb cd               jmp     SHORT $LN2@main
$LN1@main:
; 70   : /*70*/            printf("\n\n");
002fa  8b f4               mov     esi, esp
002fc  68 00 00 00 00 push   OFFSET ??_C@_02PHMGELLB@?6?6?$AA@
00301  ff 15 00 00 00 00 call  DWORD PTR __imp__printf
00307  83 c4 04            add     esp, 4
0030a  3b f4               cmp     esi, esp
0030c  e8 00 00 00 00 call     __RTC_CheckEsp
; 71   : /*70*/            return 0;
00311  33 c0               xor     eax, eax
; 72   : /*71*/  }
00313  52                  push    edx
00314  8b cd               mov     ecx, ebp
00316  50                  push    eax
00317  8d 15 00 00 00 00 lea  edx, DWORD PTR $LN39@main
0031d  e8 00 00 00 00 call     @_RTC_CheckStackVars@8
00322  58                  pop     eax
00323  5a                  pop     edx
00324  5f                  pop     edi
00325  5e                  pop     esi
00326  5b                  pop     ebx
00327  8b 4d fc            mov     ecx, DWORD PTR __$ArrayPad$[ebp]
0032a  33 cd               xor     ecx, ebp
0032c  e8 00 00 00 00 call     @__security_check_cookie@4
00331  81 c4 18 01 00 00 add esp, 280                                   ; 00000118H
00337  3b ec               cmp     ebp, esp
00339  e8 00 00 00 00 call     __RTC_CheckEsp
0033e  8b e5               mov     esp, ebp
00340  5d                  pop     ebp
```

```
00341 c3              ret     0
00342 8b ff           npad    2
$LN39@main:
00344 01 00 00 00     DD      1
00348 00 00 00 00     DD      $LN38@main
$LN38@main:
0034c b0 ff ff ff     DD      -80                             ; fffffb0H
00350 09 00 00 00     DD      9
00354 00 00 00 00     DD      $LN37@main
$LN37@main:
00358 41              DB      65                              ; 00000041H
00359 33              DB      51                              ; 00000033H
0035a 00              DB      0
_main ENDP
_TEXT         ENDS
END
```

*ПРИМІТКА. Основні фрагменти лістингу, що необхідні для створення асемблерної вставки, виділені **жирним** шрифтом.*

Використавши отриманий код, замінимо частини оригінального коду C++ програми асемблерними вставками. Матимемо наступний вигляд програми:

```
#include <stdio.h>
#define eax EAX
#define ecx ECX
char i1,i3,s4;
int A1[15], A2[8], A4[10];
int main(){
int j1,i2,j3,i4;
char j2,A3[9];
        //example #1
i1=7;
j1=0;
__asm{
LN34main:
cmp    DWORD PTR j1, 15
jge    SHORT LN33main
; 14   : /*14*/          if((j1>4) && (j1<11))
cmp    DWORD PTR j1, 4
jle    SHORT LN32main
cmp    DWORD PTR j1, 11
jge    SHORT LN32main
; 15   : /*15*/          i1=4*j1;
mov    eax, DWORD PTR j1
shl    eax, 2
mov    BYTE PTR i1, al                          ; i1
LN32main:
; 16   : /*16*/          A1[j1]=i1;
movsx eax, BYTE PTR i1                          ; i1
mov    ecx, DWORD PTR j1
```

```
mov    DWORD PTR A1[ecx*4], eax
; 17   : /*17*/          j1++;
mov    eax, DWORD PTR j1
add    eax, 1
mov    DWORD PTR j1, eax
; 18   : /*18*/          }
jmp    SHORT LN34main
LN33main:
}
        //example #2
i2=0;
j2=7;
__asm{
LN31main:
; 23   : /*23*/          do{
; 24   : /*24*/          if(i2>=3)
cmp    DWORD PTR i2, 3
jl     SHORT LN28main
; 25   : /*25*/          j2=3*i2-5;
mov    eax, DWORD PTR i2
imul   eax, 3
sub    eax, 5
mov    BYTE PTR j2, al
; 26   : /*26*/          else
jmp    SHORT LN27main
LN28main:
; 27   : /*27*/          j2=j2+i2+2;
movsx eax, BYTE PTR j2
mov    ecx, DWORD PTR i2
lea    edx, DWORD PTR [eax+ecx+2]
mov    BYTE PTR j2, dl
LN27main:
; 28   : /*28*/          A2[i2]=j2;
movsx eax, BYTE PTR j2
mov    ecx, DWORD PTR i2
mov    DWORD PTR A2[ecx*4], eax
; 29   : /*29*/          i2++;
mov    eax, DWORD PTR i2
add    eax, 1
mov    DWORD PTR i2, eax
; 30   : /*30*/          }while(i2<8);
cmp    DWORD PTR i2, 8
jl     SHORT LN31main
}
        //example #3
__asm{
mov    DWORD PTR j3, 0
jmp    SHORT LN26main
LN25main:
mov    eax, DWORD PTR j3
add    eax, 1
mov    DWORD PTR j3, eax
```

```
LN26main:
cmp    DWORD PTR j3, 9
jge    LN24main
; 34   : /*34*/          i3=2*j3;
mov    eax, DWORD PTR j3
shl    eax, 1
mov    BYTE PTR i3, al

; 35   : /*35*/          switch(j3){
mov    eax, DWORD PTR j3
mov    ebx, eax
cmp    ebx, 3
je     SHORT LN21main
cmp    ebx, 5
je     SHORT LN20main
cmp    ebx, 7
je     SHORT LN19main
jmp    SHORT LN18main
LN21main:
; 36   : /*36*/          case 3:i3+=10; break;
movsx eax, BYTE PTR i3
add    eax, 10
mov    BYTE PTR i3, al
jmp    SHORT LN22main
LN20main:
; 37   : /*37*/          case 5: i3*=2;break;
movsx eax, BYTE PTR i3
shl    eax, 1
mov    BYTE PTR i3, al
jmp    SHORT LN22main
LN19main:
; 38   : /*38*/          case 7: i3-=4; break;
movsx eax, BYTE PTR i3
sub    eax, 4
mov    BYTE PTR i3, al
jmp    SHORT LN22main
LN18main:
; 39   : /*39*/          default: i3++;
mov    al, BYTE PTR i3
add    al, 1
mov    BYTE PTR i3, al
LN22main:
; 40   : /*40*/          }
; 41   : /*41*/          A3[j3]=i3;
mov    eax, DWORD PTR j3
mov    cl, BYTE PTR i3
mov    BYTE PTR A3[eax], cl
; 42   : /*42*/          }
jmp    LN25main
LN24main:
}
        //example #4
```

```
s4=0;
__asm{
mov    DWORD PTR i4, 0
jmp    SHORT LN17main
LN16main:
mov    eax, DWORD PTR i4
add    eax, 1
mov    DWORD PTR i4, eax
LN17main:
cmp    DWORD PTR i4, 10
jge    SHORT LN15main
; 47   : /*47*/          A4[i4]=2*i4;
mov    eax, DWORD PTR i4
shl    eax, 1
mov    ecx, DWORD PTR i4
mov    DWORD PTR A4[ecx*4], eax
; 48   : /*48*/          s4+=A4[i4];
movsx eax, BYTE PTR s4
mov    ecx, DWORD PTR i4
add    eax, DWORD PTR A4[ecx*4]
mov    BYTE PTR s4, al                    ; s4
; 49   : /*49*/          if(s4>10)
movsx eax, BYTE PTR s4
cmp    eax, 10
jle    SHORT LN14main
; 50   : /*50*/          A4[i4]=s4;
movsx eax, BYTE PTR s4
mov    ecx, DWORD PTR i4
mov    DWORD PTR A4[ecx*4], eax
; 51   : /*51*/          else
jmp    SHORT LN13main
LN14main:
; 52   : /*52*/          A4[i4]=i4;
mov    eax, DWORD PTR i4
mov    ecx, DWORD PTR A4[eax*4]
add    ecx, 4
mov    edx, DWORD PTR i4
mov    DWORD PTR A4[edx*4], ecx
LN13main:
; 53   : /*53*/          }
jmp    SHORT LN16main
LN15main:
}
//output
for(j1=0;j1<15;j1++)
printf("%d ",A1[j1]);
printf("\n\n");
for(i2=0;i2<8;i2++)
printf("%d ",A2[i2]);
printf("\n\n");
for(j3=0;j3<9;j3++)
printf("%d ",A3[j3]);
```

```
printf("\n\n");
for(i4=0;i4<10;i4++)
printf("%d ",A4[i4]);
printf("\n\n");
return 0;
}
```

Уважно проаналізуйте реалізацію C++ операторів мовою Асемблера.

2-2.5. Завдання на виконання роботи

Перше заняття

1) У середовищі розробки Microsoft Visual Studio створити власний проект з C++ програмою згідно варіанта завдання (табл. 2-2.1). При створенні проекту, а також виконанні інших пунктів завдання слід використовувати надані вище Рекомендації щодо виконання роботи.
2) Запустити програму на виконання і впевнитись, що програма працює коректно.
3) Створити файл лістингу з асемблерним кодом своєї програми.
4) Ознайомитись з реалізацією C++ операторів мовою Асемблера.

Друге заняття

1) Вибрати декілька C++ операторов (основний цикл або тільки його тіло) і виконати їх заміну у програмі на асемблерну вставку.
2) Переконатись в правильності функціонування модифікованої програми шляхом порівняння результатів її роботи з результатами початкової програми мовою C++.
3) Спробувати оптимізувати зміст асемблерної вставки, намагаючись, якщо можливо, досягти економії часу виконання і/або пам’яті.
4) Переконатись в правильності функціонування оптимізованої програми шляхом порівняння результатів її роботи з попередніми результатами.

Таблиця 2-2.1

Варіанти завдання

1.

```
#include <stdio.h>
int A[8];
int k;
int main(){
int i,j;
k=1;
i=0;
do{
  j=i+k*2;
  if(j>11)
    A[i]=j;
  else{
    k+=1;
  A[i]=k;
  }
i++;
}while(i<8);
for(i=0;i<8;i++)
  printf("%d ",A[i]);
printf("\n");
return 0;
}
```

2.

```
#include <stdio.h>
int A[9];
int i,j;
int main(){
i=0;
while(i<9){
  j=2*i;
  switch(i){
    case 5:j+=10; break;
    case 6: j*=2; break;
    case 7: j-=4; break;
    case 8: j=7; break;
    default: j++;
  }
  A[i]=j;
  i++;
}
for(i=0;i<9;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

3.

```
#include <stdio.h>
int i,k;
int main(){
int j;
int A[9];
k=1;
for(i=0;i<=8;i++){
  j=k+i*3;
  if(j>18)
    A[i]=j;
  else{
    k+=2;
  A[i]=k;
  }
}
for(i=0;i<=8;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

4.

```
#include <stdio.h>
int A[8];
int main(){
int       i,d;
d=1;
i=7;
do{
  d=i%2;
  if(!d)
    A[i]=i;
  else
    A[i]=-i;
  i--;
}while(i>=0);
for(i=0;i<8;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

5.

```
#include <stdio.h>
int A[7];
int d;
int main(){
int i;
d=1;
i=0;
while(i<=6){
  A[i]=i | d;
  d+=A[i];
  if(d>10)
    A[i]=127-i;
  i++;
  }
for(i=0;i<7;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

6.

```
#include <stdio.h>
int A[8];
int main(){
int p,k;
k=1;
p=0;
do{
  k=p;
  k++;
  switch(p){
    case 0: k=99; break;
    case 2: k+=10; break;
    case 4: k*=2; break;
    case 6: k-=4; break;
    default: k*=4;
  }
  A[p]=k;
  p++;
  }while(p<8);
for(p=0;p<8;p++)
printf("%d ",A[p]);
printf("\n");
return 0;
}
```

7.

```
#include <stdio.h>
char b;
int c;
int main(){
int       a1;
int A[7];
a1=3;
c=0;
for(b=0;b<7;b++){
  c+=3;
  if(c<9)
    a1=5*c+b+1;
  else
    a1=b<<2;
  A[b]=a1;
  }
for(b=0;b<7;b++)
printf("%d ",A[b]);
printf("\n");
return 0;
}
```

8.

```
#include <stdio.h>
char a1,c;
int A[13];
int main(){
char b;
a1=2;
b=12;
while(b>=0){
  if(b<10)
    c=a1*2;
  else
    c-=a1;
  A[b]=c;
  a1+=b;
  b--;
 }
for(b=0;b<13;b++)
printf("%d ",A[b]);
printf("\n");
return 0;
}
```

9.

```
#include <stdio.h>
int c;
int A1[12];
int main(){
int a;
char b;
a=6;
b=0;
do{
  if(a>b)
    c=a*3-b;
  else
    c=a*2+b;
  A1[b]=c;
  b++;
  }while(b!=12);
for(b=0;b<12;b++)
printf("%d ",A1[b]);
printf("\n");
return 0;
}
```

10.

```
#include <stdio.h>
int A[11];
int main(){
int l,k;
char j;
l=2;
k=0;
for(j=0;j<11;j++){
  k=j;
  l+=8;
  k++;
  switch(j){
    case 1: k=0; break;
    case 3: k+=l; break;
    case 5: k-=l; break;
    case 6: k=2*l-k; break;
    default: k--;
  }
  A[j]=k;
  }
for(j=0;j<11;j++)
printf("%d ",A[j]);
printf("\n");
return 0;
}
```

11.

```
#include <stdio.h>
int k;
int main(){
int A[10];
char j;
k=3;
j=0;
while(j<10){
  if(j!=7)
  k+=j;
  A[j]=k;
  j++;
  }
for(j=0;j<10;j++)
printf("%d ",A[j]);
printf("\n");
return 0;
}
```

12.

```
#include <stdio.h>
int l,k;
int A[11];
int main(){
char j;
l=2;
k=64;
j=10;
do{
  if(j%4==0)
    k/=l;
  else
    k+=l;
  A[j]=k;
  j--;
  }while(j>=0);
for(j=0;j<11;j++)
printf("%d ",A[j]);
printf("\n");
return 0;
}
```

13.

```
#include <stdio.h>
int p,j;
int main(){
int A[11];
for(p=0;p<11;p++){
  j=32-2*p;
  switch(p){
    case 9: j*=2; break;
    case 7: j+=15; break;
    case 4: j=-5; break;
    case 2: j-=8; break;
    case 0: j=0; break;
    default: j=77;
    }
  A[p]=j;
  }
for(j=0;j<11;j++)
printf("%d ",A[j]);
printf("\n");
return 0;
}
```

14.

```
#include <stdio.h>
int A[11];
int main(){
int j;
j=0;
A[4]=0;
while(j<11){
  if(j>5)
    A[j]=A[j-1]+j;
  else
    A[j] = j*j;
  j++;
  }
for(j=0;j<11;j++)
printf("%d ",A[j]);
printf("\n");
return 0;
}
```

15.

```
#include <stdio.h>
int j;
int main(){
int A[11];
j=2;
A[1]=1;
A[0]=1;
do{
  A[j]=A[j-1]+A[j-2];
  if(A[j]>62)
    A[j]=0;
  j++;
  }while(j<11);
for(j=0;j<11;j++)
printf("%d ",A[j]);
printf("\n");
return 0;
}
```

16.

```
#include <stdio.h>
int A[11];
int s;
int main(){
int i;
s=0;
for(i=0;i<11;i++){
  A[i]=2*i;
  s+=A[i];
  if(s>10)
    A[i]=s;
  }
for(i=0;i<11;i++)
printf("%d ",A[i]);
printf("\n");
return 0; }
```

17.

```
#include <stdio.h>
int i,s;
int main(){
int A[11];
s=0;
i=10;
```

18.

```
#include <stdio.h>
int i;
int A[11];
int main(){
int s;
s=0;
```

```
while(i>=0){
  s=i*4;
  switch(i){
    case 6: s=17; break;
    case 3: s*=4; break;
    case 4: s+=8; break;
    default: s-=4;
  }
  A[i]=s;
  i--;
}
for(i=0;i<11;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

```
i=0;
do{
  A[i]=2*i;
  s+=A[i];
  if(s>80)
    A[i]=s;
  i++;
  }while(i<11);
for(i=0;i<11;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

19.

```
#include <stdio.h>
int i;
int main(){
int m;
int A[11];
m=1;
for(i=0;i<11;i++){
  m= (m^i) && 1;
  if(m==1)
    A[i]=2*i;
  else
    A[i]=2*i+1;
  }
for(i=0;i<11;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

20.

```
#include <stdio.h>
int A[11];
int m;
int main(){
int i;
m=1;
i=0;
while(i<=10){
  A[i]=3*i;
  m+=A[i];
  if(m>100)
    A[i]=2*i;
    i++;
  }
for(i=0;i<11;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

21.

```
#include <stdio.h>
int i,m;
int main(){
int A[9];
m=0;
i=8;
do{
  m=8*i;
  switch(i){
    case 2: m+=4; break;
    case 0: m=17; break;
    case 7: m-=4; break;
```

22.

```
#include <stdio.h>
int r;
int A[10];
int main(){
int i;
r=0;
for(i=0;i<10;i++){
  r+=i*2;
  if(r<32)
    A[i]=r;
  else
    A[i]=r & i;
```

```
   case 1: m=4; break;
   default: m++;
  }
  A[i]=m;
  i--;
  }while(i>-1);
for(i=0;i<9;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

```
}
for(i=0;i<10;i++)
printf("%d ",A[i]);
printf("\n");
return 0; }
```

23.

```
#include <stdio.h>
int i,r;
int main(){
int A[11];
r=100;
i=10;
while(i>=0){
  A[i]=2*i;
  r-=A[i];
  if(r<0)
    A[i]=-A[i]+1;
  i--;
  }
for(i=0;i<11;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

24.

```
#include <stdio.h>
int A[11];
int main(){
int i,r;
r=0;
i=1;
A[0]=4;
do{
  A[i]=A[i-1]+r;
  r=r | i;
  if(r>9)
    A[i]=r;
  i++;
  }while(i<11);
for(i=0;i<11;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

25.

```
#include <stdio.h>
char i;
int main(){
char d;
int A[4];
d=1;
for(i=0;i<4;i++){
  d=16*i;
  switch(i){
    case 0: d+=14; break;
    case 1: d*=4; break;
    case 3: d-=7; break;
    default: d+=10;
  }
  A[i]=d;
}
```

26.

```
#include <stdio.h>
char p;
char A[11];
int main(){
int i;
p=0;
i=0;
while(p<11){
  if(p>=7)
    i=i ^ p;
  else
    i=i | p;
  A[p]=i;
  p++;
  }
for(i=0;i<11;i++)
```

```
for(i=0;i<4;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

```
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

27.

```
#include <stdio.h>
unsigned int A[6];
int main(){
int i,p;
p=6;
for(i=5;i>=0;i--){
  if(i<3)
    p++;
  else
    p--;
  A[i]=p;
  }
for(i=0;i<6;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

28.

```
#include <stdio.h>
int j;
int main(){
int i;
int A[7];
i=3;
j=0;
do{
  if(j<5)
    i+=2*j;
  else
    i+=j;
  A[j]=i;
  j++;
  }while(j<7);
for(i=0;i<7;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

29.

```
#include <stdio.h>
int i;
int A[9];
int main(){
int j;
i=0;
j=0;
while(i<=8){
  j=i+10;
  switch(i){
    case 2: j=0; break;
    case 4: j=13; break;
    case 6: j*=4; break;
    case 8: j+=9; break;
    default: j--;
  }
  A[i]=j;
  i++;
}
for(i=0;i<9;i++)
printf("%d ",A[i]);
printf("\n"); return 0;}
```

30.

```
#include <stdio.h>
unsigned int A[9];
int main(){
int i,k;
k=0;
for(i=0;i<9;i++){
  k+=i;
  if(k<9)
    A[i]=i;
  else
    A[i]=k;
  }
for(i=0;i<9;i++)
printf("%d ",A[i]);
printf("\n");
return 0;
}
```

2-2.6. Контрольні запитання

1. Що відображає мовою Асемблера ідентифікатор змінної мови С++?
2. Як реалізуються мовою Асемблера оператори присвоєння?
3. Як реалізуються мовою Асемблера оператори **if** та **if...else**?
4. Як реалізуються мовою Асемблера оператори **for**?
5. Як реалізуються мовою Асемблера оператори **while**?
6. Як реалізуються мовою Асемблера оператори **do…while**?
7. Як реалізується мовою Асемблера оператор **switch**?
8. Які вимоги ставлять до асемблерних вставок у програмах мовою С++?

ЛАБОРАТОРНА РОБОТА №3

Ознайомлення з методами адресації даних

Мета роботи – вивчення типових структур даних та методів адресації їх елементів у мікропроцесорах 80x86 (Pentium) в реальному режимі.

3.1. Зміст роботи

При виконанні роботи студенти вивчають особливості створення та використання в мові Асемблера багатовимірних масивів, структур та масивів структур. Для доступу до елементів складних структур даних вивчаються режими адресації та команди обробки адрес. З навчальною метою мовою Асемблера створюються програми, які складаються з багатьох логічних сегментів. На конкретних прикладах вивчаються проблеми, які виникають при об'єднанні логічних сегментів в один фізичний сегмент.

3.2. Теоретичні відомості

Створення багатовимірних масивів, структур та масивів структур у програмах мовою Асемблера

1) Директиви визначення простих даних

Асемблер надає дуже широкий набір засобів подання й обробки даних, які за своїми можливостями не гірші від аналогічних засобів багатьох мов високого рівня. Спочатку розглянемо правила опису простих типів даних, які є базовими для більш складних типів. *Для опису використовують спеціальні директиви резервування й ініціалізації даних*. Якщо проводити аналогію з мовами високого рівня, то ці директиви являють собою визначення змінних. Машинного еквіваленту цим директивам не має, просто транслятор, оброблюючи кожну таку директиву, виділяє необхідну кількість комірок пам'яті та, при необхідності, ініціалізує цю область деяким значенням. Формат

директив резервування й ініціалізації даних простих типів показаний на рис. 3.1.

- **db** – резервування пам’яті для даних розміром 1 байт;
- **dw** - резервування пам’яті для даних розміром 2 байти;
- **dd** - резервування пам’яті для даних розміром 4 байти;
- **df** - резервування пам’яті для даних розміром 6 байтів;
- **dp** - резервування пам’яті для даних розміром 6 байтів;
- **dq** - резервування пам’яті для даних розміром 8 байтів;
- **dt** - резервування пам’яті для даних розміром 10 байтів.

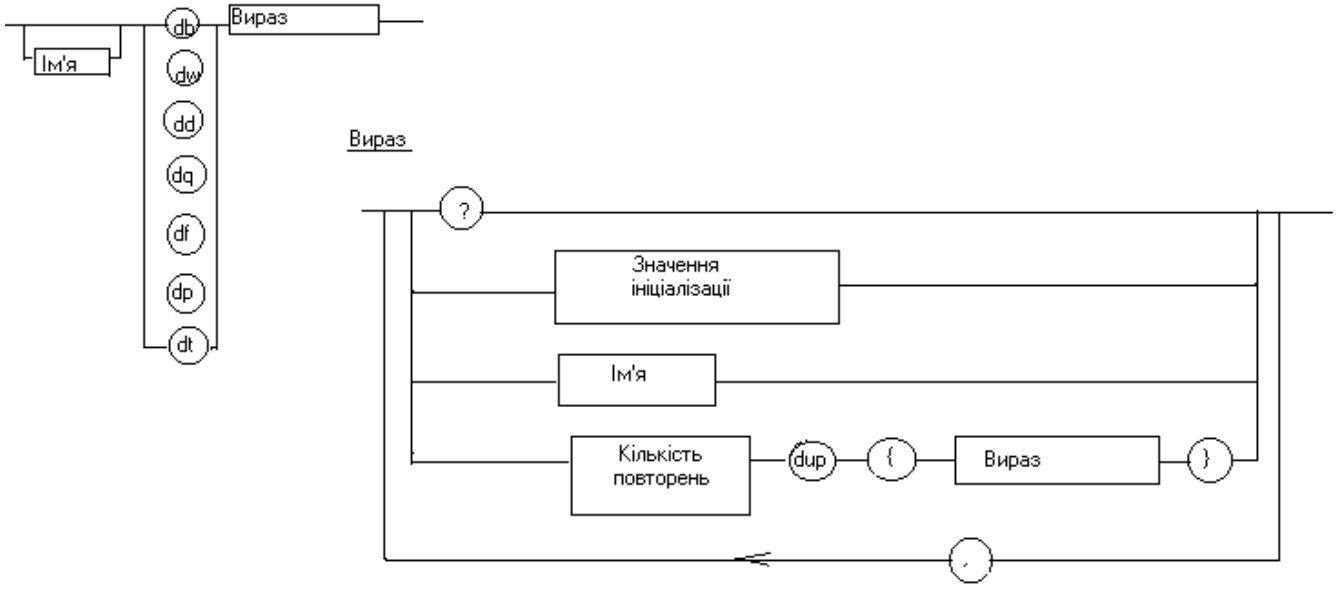

Рис. 3.1. Формат директив резервування й ініціалізації даних

Транслятор *masm*, залежно від версії, може не підтримувати директиви **df** та **dp**. На рис.3.1 використані наступні позначення:

- **?** показує, що вміст поля не визначено, тобто *вміст виділеної частини фізичної пам’яті при завантаженні програми змінюватися не буде*. Фактично створюється неініціалізована змінна. Операційні системи в багатьох випадках попередньо обнуляють область пам’яті для завантаження програми, але розраховувати на це не слід.

- значення ініціалізації – значення елемента даних, котре буде занесене в пам’ять після завантаження програми. Фактично створюється ініціалізована змінна; в якості ініціалізатора можуть виступати константи, рядки символів, константні й адресні вирази в залежності від типу даних.
- вираз – ітеративна конструкція із синтаксисом, представленим на рисунку.
- ім’я – ідентифікатор, який репрезентує адресу або константу і визначений у програмі, наприклад:

```
Value1          db    ?
….
P1          dw    Value1
P2          dd    Value1
P3          dw    @10
P4          dd    @10
```

За адресою P1 транслятором згенерується зміщення у сегменті зарезервованої комірки пам’яті, а за адресою P2 – повна логічна адреса комірки. Аналогічно у випадку мітки @10.

Дуже важливо уяснити собі порядок розміщення байтів багатобайтних даних у пам’яті. Він зумовлений логікою роботи мікропроцесора з даними. Для мікропроцесорів 80x86 молодший байт знаходиться за молодшою адресою.

2) Визначення масивів

Згідно з рис.3.1 для завдання одновимірних масивів можна використати методи, які пояснюються наступними прикладами:

```
Array1      db    1,2,3,4,2,3
Array2      db    1500 dup (?)
Array3      db    2000 dup (56h)
```

У першому випадку, кожний елемент масиву Array1 ініціалізується окремо. У другому випадку, масив Array2 не ініціалізується. У третьому випадку, всі елементи масиву ініціалізуються значенням 56h. При цьому кількість елементів в масиві дорівнює кількості повторень (*dup*).

Багатовимірний масив задається шляхом використання вкладених повторень, наприклад:

```
Ar1   db     4 dup (3 dup (2 dup (?)))
```

Мовою Паскаль це еквівалентно наступному оператору:

```
Ar1: array[0..3,0..2,0..1] of byte;
```

Звідси можна вважати, що мовою Асемблера всі індекси у масивах розпочинають свої значення з 0. Оскільки пам’ять є насправді одновимірним масивом комірок, то можуть виникати розбіжності у розміщені елементів багатовимірних масивів в одновимірному масиві комірок. При програмуванні мовами високого рівня формування адрес комірок покладається на компілятор, а програмісту необхідно слідкувати лише за порядком індексів при зверненні до елементів багатовимірного масиву. Тому порядок розміщення елементів багатовимірних масивів в одновимірному масиві комірок не має суттєвого значення. При програмуванні мовою Асемблера програміст змушений сам формувати адреси елементів багатовимірних масивів. Тому вказаний порядок має першочергове значення. У програмуванні, як правило, прийнятий такий порядок розміщення елементів багатовимірного масиву в одновимірному масиві комірок як у прикладі (табл. 3.1), де перший рядок містить адресу (зміщення) елемента масиву відносно початку масива, а другий рядок – значення індексів.

Табл. 3.1

Порядок розміщення елементів багатовимірного масиву в пам’яті

Ar1+0	Ar1+1	Ar1+2	Ar1+3	Ar1+4	Ar1+5	…..	Ar1+22	Ar1+23
0,0,0	0,0,1	0,1,0	0,1,1	0,2,0	0,2,1	…..	3,2,0	3,2,1

У загальному випадку, зміщення елемента масиву відносно його початку, згідно із вищевказаним порядком, обчислюється за наступною формулою:

$$O=(I_1*(P_2)+ I_2*(P_3)+ I_3*(P_4)+ \ldots+I_{n-1}*(P_n)+ I_n)*T, \qquad (3.1)$$

де I_1, I_2,…, I_n – індекси n-вимірного масиву (вважається, що індексація починається з 0);

P_j (j=2,3,…,n) – добуток розмірів вимірів масиву від j до n включно;

T – тип (кількість байтів) елемента масиву.

Для масиву Ar1 ця формула матиме наступний вигляд:

$$O=(I_1*6+ I_2*2 + I_3)$$

Окрім формули (3.1) у програмуванні часто використовується так звана схема Горнера:

$$O=((...((I_1 *D_2 + I_2)*D_3 + I_3)*D_4+...)*D_n+ I_n)*T \qquad (3.2)$$

Для масиву Ar1 ця формула матиме наступний вигляд:

$$O=(I_1 3+ I_2)*2+ I_3$$

Схема Горнера може бути ефективнішою ніж (3.1), особливо при великій кількості вимірів. Із табл. 3.1 та формул (3.1), (3.2) випливає, що для послідовного, за розташуванням у пам’яті, перегляду елементів масиву зміна індексів розпочинається з I_n у напрямку до I_1 .

3) Визначення структур та масивів структур

Недоліком масивів при вирішенні, зокрема, задач системного програмування є однотипність усіх елементів. Якщо в складних структурах даних елементи є різнотипними, тоді використовуються структури та масиви структур. У мові Асемблера перед використанням структури попередньо повинен бути заданий шаблон структури, який має наступний формат:

```
Name_st     struc
<Директиви визначення простих даних або масивів>
Name_st     ends
```

де Name_st – оригінальний ідентифікатор користувача (ім’я структури). Наприклад:

```
Instr32     struc
Opcode      dw    ?
Modrm       db    ?
Sib         db    ?
Disp        dd    ?
Instr32     ends
```

Сама структура задається у форматі директив визначення даних, де в полі мнемокоду задається ім’я структури, наприклад:

```
In1    instr32      <>
```

або

```
Min1 instr32      5 dup(<>)
```

У прикладах задається лише резервування пам’яті без початкової ініціалізації. Детальніше зі структурами для мови Асемблера можна познайомитись, наприклад, в [3, урок 12, Сложные структуры данных].

В мові Асемблера існують спеціальні оператори часу трансляції, що використовуються для визначення кількісних характеристик масивів. До таких операторів належать:

- LENGTH – визначає кількість елементів даних у масиві
- SIZE – визначає кількість байтів, які займає масив
- TYPE – визначає кількість байтів, які займає елемент масиву

Наприклад:

```
table       dw    0,1,2,3,4,5,6,7          ;масив
mov         ax, length table               ;ax=8
mov         ax, size table                 ;ax=16
mov         ax, type table                 ;ax:=2
```

або

```
mov         ax, type instr32               ;ax:=8
mov         ax, type Min1                  ;ax:=8
mov         ax, length Min1                ;ax=5
mov         ax, size table                 ;ax=40
```

Отже SIZE <array>= LENGTH <array> * TYPE <array>

Засоби адресації простих даних та елементів складних даних

Переважна більшість команд процесорів 80x86 мають адресну частину, яка у загальному випадку містить байти *modr/m*, *sib* та *зміщення* в команді. На основі цих даних процесор формує зміщення в сегменті, яке у даному випадку назвали *ефективною адресою*. У загальному випадку *ефективна адреса є сумою трьох компонент – зміщення в команді, бази та індексу*. База та індекс

містяться в регістрах загального призначення, які використовуються як адресні регістри. Індекс може мати множник 2,4 або 8. Він визначає, на яку величину необхідно помножити вміст 32-розрядного індексного регістра перед формуванням ефективної адреси (*для 16-розрядних регістрів множник не задається!*). Будь-яка з компонент в адресному виразі може бути відсутня, що визначає наступні можливі режими адресації (табл.3.2):

Табл.3.2

Режими адресації даних

Зміщення в команді	База	Індекс	Режим адресації	Приклад
-	-	+	Посередня регістрова	[si], [eax] [esp]
-	+	+	Базова індексна	[bx+si], [ecx+edx] [ebx+esi*4]
-	+	-	Посередня регістрова	[bx], [ecx]
+	-	+	Індексна	Dat1[si] Dat1[ecx] Dat1[edi*8]
+	-	-	Пряма	Dat1
+	+	+	Базова індексна зі зміщенням	Dat1[bx+di] Dat1[ebx][edx] Dat1[edx+esi*2]
+	+	-	Базова	[bp+4], [bp-6] [ecx+7]

ПРИМІТКА 1. Для 16-розрядних регістрів при формуванні ефективної адреси можуть використовуватись лише регістри BP, BX, SI і DI , а також лише наступні їх пари: BX+SI, BX+DI, BP+SI та BP+DI. Для 32-розрядних регістрів загального призначення таке обмеження на їх використання

відсутнє, за виключенням регістра ESP – він не може задаватись із множником.

*ПРИМІТКА 2. Мовою Асемблера можна задати посередню регістрову адресацію із множником, наприклад, Add eax,[edx*4], але в процесорі такі команди відсутні. Асемблер сформує машинну команду, в якій зміщення в команді буде мати нульове значення.*

При програмуванні мовою Асемблера розглянуті вище режими адресації доцільно використовувати наступним чином:

- *пряму* – для адресації простих (скалярних) даних, адреси яких при виконанні програми не змінюються;
- *посередню регістрову* - для адресації скалярних даних, адреси яких змінюються при виконанні програми;
- *індексну* – для адресації елементів масивів. Особливо ефективно можна використовувати індексну адресацію для доступу до елементів одновимірних масивів байтів, масивів слів, подвійних та квадро слів;
- *базову* – для адресації елементів структур, відносні адреси яких при виконанні програми не змінюються;
- *базову індексну* – для адресації елементів структур, відносні адреси яких змінюються при виконанні програми;
- *базову індексну зі зміщенням* – для адресації елементів масивів структур або для адресації елементів багатовимірних масивів.

Обчислення адрес елементів складних структур даних, таких як багатовимірні масиви, списки, графи, дерева і т.п. у загальному випадку може бути достатньо трудомістким. Тому в склад команд процесора включені команди *LEA* та багатоопераднa команда *IMUL*. Команда *LEA* має наступний формат:

LEA *приймач*, *джерело*

Операнд *приймач* – 16-розрядний регістр (звичайно адресний), або 32-розрядний регістр. Операнд *джерело* – адресний операнд мови Асемблера.

Алгоритм роботи команди наступний:

1) якщо 16-розрядний приймач та 16-розрядна адресація, то в регістр *приймач* завантажується 16-бітне значення ефективної адреси;
2) якщо 32- розрядний приймач та 16-розрядна адресація, то в молодші 16 розрядів регістра *приймач* завантажується 16-бітне значення ефективної адреси, а в старші 16 розрядів записується 0.
3) якщо 16-розрядні дані та 32-розрядна адресація, то в регістр *приймач* завантажуються молодші 16-біт значення ефективної адреси;
4) якщо 32-розрядні дані та 32-розрядна адресація, то в регістр *приймач* завантажується 32-бітне значення ефективної адреси.

Багатоопернадна команда *IMUL* має наступні формати:

IMUL *множ_1, множ_2*

IMUL *рез-т, множ_1, множ_2*

або детальніше в табл. 3.3.

Табл. 3.3

Опис команди IMUL

Структура команди	Приклад
IMUL r16,r/m16	IMUL bx, word ptr[si]
IMUL r32,r/m32	IMUL eax, dword ptr[edx*8]
IMUL r16,r/m16, imm8	IMUL si, [bx], 6
IMUL r32,r/m32, imm8	IMUL ecx, esi, 11
IMUL r16,imm8	IMUL si, 6
IMUL r32,imm8	IMUL edi, 23
IMUL r16,r/m16, imm16	IMUL bx, dx, 166h
IMUL r32,r/m32, imm32	IMUL ecx, esi, 112233h
IMUL r16,imm16	IMUL si, 166h
IMUL r32,imm32	IMUL edx, 666777h

Команди з двома та трьома операндами однозначно визначають розташування результату і співмножників у такий спосіб:

1) У команді з двома операндами перший операнд визначає місце розташування першого співмножника. На його місце згодом буде

записаний результат. Другий операнд визначає місце розташування другого співмножника: *множ_1 = множ_1 * множ_2* ;

2) У команді з трьома операндами перший операнд визначає місце розташування результату, другий операнд – місце розташування першого співмножника, третій операнд може бути лише безпосередньо заданим значенням розміром байт, слово чи подвійне слово:
*множ_1 = множ_2 * множ_3*

Команда *IMUL* встановлює в нуль ознаки ***of*** і ***cf***, якщо розмір результату відповідає розміру регістра призначення. Відмінність цих ознак від нуля означає, що результат занадто великий для відведеного йому регістра призначення, і тоді старші розряди добутку ігноруються. Проте при обчисленні адрес така невідповідність малоймовірна, а в реальному режимі призведе лише до звернення до менших зміщень в сегменті (за кільцем).

Розглянемо приклади використання команди *LEA* та багатооперандної команди *IMUL*.

```
; завантаження в регістр ax елемента mas[i1,i2,i3]
.data
  mas    dw     10 dup (12 dup (14 dup (? )))
  i1     dw     ?
  i2     dw     ?
  i3     dw     ?
.code
  imul   bx, i1, 12*14*2
  imul   ecx, i2, 14*2
  xor    esi, esi
  mov    si, i3
  lea    di, [ecx+esi*2]
  mov    ax, mas[bx+di]          ;ax:=mas[i1,i2,i3]
```

Команду IMUL із трьома операндами доцільно застосовувати для визначення адрес структур у масиві структур, наприклад:

```
person      struc
name        db        30 dup (?)
grup        db        6 dup (?)
zalic       db        ?
person      ends
base_person           person 50 dup (<>)
mov         cx, 50
@12:
```

```
imul    si, cx, size person
mov     byte ptr base_person[si-size person].zalic, '+'
loop    @12
```

Група команд обробки одновимірних масивів (команди обробки рядків, ланцюгові команди)

Усі команди цієї групи є однобайтними, і вони не мають адресної частини. Адреси операндів команд задаються неявно (фактично кодом операції). Кожна із цих команд має два операнди: або обидва в пам’яті, або один в акумуляторі, а другий в пам’яті (табл. 3.4).

Табл. 3.4

Команди обробки рядків

Машинний код команди	Мнемоніка	Приймач	Джерело	Призначення
0A4h	MOVSB	пам’ять	пам’ять	Пересилання елементів масивів
0A5h	MOVSW або MOVSD	пам’ять	пам’ять	Пересилання елементів масивів
0A6h	CMPSB	пам’ять	пам’ять	Порівняння елементів масивів
0A7h	CMPSW або CMPSD	пам’ять	пам’ять	Порівняння елементів масивів
0AEh	SCASB	пам’ять	акумулятор	Порівняння елемента масиву з даними в акумуляторі
0AFh	SCASW або SCASD	пам’ять	акумулятор	Порівняння елемента масиву з даними в акумуляторі
0AAh	STOSB	пам’ять	акумулятор	Занесення в елемент масиву даних з акумулятора
0ABh	STOSW або STOSD	пам’ять	акумулятор	Занесення в елемент масиву даних з акумулятора
0ACh	LODSB	акумулятор	пам’ять	Занесення в акумулятор елемента масиву
0ADh	LODSW або LODSD	акумулятор	пам’ять	Занесення в акумулятор елемента масиву

Особливість цих команд полягає в наступному:

1) Якщо операнд розташований у пам'яті, то у випадку джерела його логічна адреса – **DS:SI**, а у випадку приймача – **ES:DI**.
2) Після виконання команди вміст використаних регістрів (SI та/або DI) залежно від типу (*b – byte, w – word, d – dword*) автоматично змінюється (збільшується або зменшується) на 1, 2 або 4.
3) Збільшення відбувається, коли ознака ***df*** (*direction flag*) в регістрі ознак встановлена в 0; якщо ознака *df=1*, тоді відбувається автоматичне зменшення.
4) Перед командами може використовуватись префікс повторення для організації апаратних циклів.

Префікси повторення в мові Асемблера задаються тільки явно як окрема машинна інструкція або безпосередньо в полі мнемокоду ланцюгової команди. Мнемоніки префіксів повторення: *REP*, *REPZ* (або *REPE*), *REPNZ* (або *REPNE*). Префікс *REP* задається перед ланцюговими командами, які не виконують порівняння. Алгоритм префікса *REP* наступний:

1) якщо CX=0 (або ECX=0), тоді вийти з циклу;
2) виконати наступну ланцюгову команду;
3) виконати декремент регістра CX (ECX) і перейти до п.1)

Префікси *REPZ* (або *REPE*), *REPNZ* (або *REPNE*) задаються перед ланцюговими командами, які виконують порівняння. Алгоритм їхньої роботи наступний:

1) якщо CX=0 (або ECX=0), тоді вийти з циклу;
2) виконати наступну ланцюгову команду;
3) виконати декремент регістра CX (ECX);
4) якщо *zf=1* для REPZ (REPE) або *zf=0* для REPNZ (REPNE), тоді перейти до п.1)

Особливості трансляції ланцюгових команд на прикладі команд *MOVS* полягають в наступному:

При трансляції команд *MOVSW* та *MOVSD* Асемблер генерує префікс заміни розрядності даних (код 66h), якщо розрядність даних за замовчуванням не відповідає мнемокоду команди. Розрядність адрес для команд *MOVSB*, *MOVSW* та *MOVSD* (відповідно використовуються регістри **SI** та **DI** чи **ESI** та **EDI**) встановлюється тільки за замовчуванням. При трансляції команди *MOVS* Асемблер може згенерувати префікс заміни розрядності даних, префікс заміни розрядності адрес і один із префіксів заміни сегменту. Нехай за замовчуванням встановлені 16-розрядні дані та адреси (*атрибут USE16 в директиві SEGMENT*), тоді в результаті трансляції машинної інструкції

movs dword ptr [edi], gs:dword ptr [esi]

асемблер згенерує наступні 4 байти:

67h| 66h| 65h: 0A5h

де 67h – префікс заміни розрядності адрес, 66h – префікс заміни розрядності даних, 65h – префікс заміни сегменту, 0A7h – код команди MOVSW. Таким чином, поле операндів в машинній інструкції MOVS фактично використовується для генерування префіксів та уточнення коду операції.

ЗАУВАЖЕННЯ. У реальному режимі при використанні регістра EDI старші 16 розрядів 32-розрядного зміщення обнулюються.

Команду *MOVS* (*MOVSB, MOVSW* та *MOVSD*) застосовують, наприклад, для пересилання рядків (послідовностей, ланцюжків, масивів) у пам’яті. Для цього необхідно використовувати префікс *REP*. Він змушує циклічно виконувати команди пересилання доти, поки вміст регістра **CX (ECX)** не дорівнюватиме нулю. При накладанні рядків приймача та джерела необхідно визначати порядок пересилання – з початку чи з кінця рядка.

```
str1        db      'str1 копіюється в str2'
len_        str1=$-str1
a_str1  dd  str1
str2        db      len_str1 dup (' ')
a_str2  dd  str2
mov         cx,len_str1
lds         si,str1
les         di,str2
cld
rep         movsb
```

3.3. Завдання на виконання роботи

Перше заняття

1) Ознайомитися з варіантом завдання (табл. 3.5): початковими даними та завданням на обробку масиву структур.
2) Ознайомитися з можливостями створення та використання в програмах мовою Асемблера багатовимірних масивів, структур та масивів структур. Вивчити режими адресації елементів складних структур даних та команди їх обробки.
3) Створити програму, що реалізує п.1 індивідуального завдання. Відтранслювати її та створити виконуваний файл програми.
4) Перевірити правильність функціонування програми у налагоджувачі AFD шляхом аналізу значень комірок пам’яті, де розміщений масив структур та індекси (використовувати вікна пам’яті налагоджувача, зупинення програми у визначених точках).

Друге заняття

1) Доповнити програму визначеннями та командами, що реалізують п.2 і 3 індивідуального завдання.
2) Відтранслювати програму та створити виконуваний файл.
3) Перевірити правильність функціонування програми у відлагоджувачі AFD шляхом аналізу значень комірок пам’яті, де розміщені масиви структур та індекси (використовувати вікна пам’яті налагоджувача, зупинення програми у визначених точках).

Таблиця 3.5

Варіанти завдання

1.

Заданий логічний сегмент

```
Data1 segment
I1      db      ?
I2      db      ?
I3      db      ?
A1      dw      5 dup (6 dup (0fh,4 dup (0)))
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: всі слова трьохвимірного масиву A1із значеннями 0fh замінити на суму наступних (за зміщенням у сегменті) 5-ти слів шляхом використання індексів I1,I2,I3 (значення, що містяться за адресами I1,I2,I3).
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 dw 5 dup (6 dup (0fh,4 dup (0)))**
3. Створити ще один логічний сегмент кодів (наприклад з ім'ям Code2), в якому завдання п.1 виконується з використанням ланцюгових команд для пошуку значення 0fh. Після підрахунку порівняти значення в масивах A1 та A2. Забезпечити безумовну передачу управління із сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

2.

Заданий логічний сегмент

```
Data1 segment
I1      db      ?
I2      db      ?
I3      db      ?
A1      dw      6 dup (8 dup (7 dup (0)))
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: в кожний елемент масиву записати добуток індексів, значення яких містяться в байтах за адресами I1,I2,I3.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 dw 6 dup (8 dup (7 dup (0)))**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд знайти всі слова масиву A1, значення яких дорівнює 36, а адреси (зміщення в сегменті) цих слів записати в елементи масиву A2 із тими самими індексами. Забезпечити безумовну передачу управління із сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

3.

Заданий шаблон структури

```
Tabl1           struc
namex           db      8 dup (?)
field2          dw      ?
Tabl1           ENDS
```

Заданий логічний сегмент

```
Data1 segment
I1      db      0
```

A1 Tabl1 6 dup (<>)
Data1 ends

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: в кожне поле field2 структур із масиву структур A1 записати зміщення в сегменті поля namex структури з тим самим індексом у масиві. Початкове значення індексу міститься в байті за адресою I1.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 Tabl1 12 dup (<>)**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати весь масив структур A1 в масив структур A2, починаючи з 3-ої структури A2. Забезпечити безумовну передачу управління із сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

4.

Заданий шаблон структури

```
Tabl1     struc
namex     db    8 dup (10h)
field1    db    ?
field2    dw    ?
field8    dq    ?
Tabl1     ENDS
```

Заданий логічний сегмент

```
Data1 segment
I1   db   1
A1   Tabl1 6 dup (<>)
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: в кожне поле field2 структур із масиву структур A1 записати суму байтів поля namex цієї ж структури плюс індекс структури в масиві (індекси змінюються від 0 до 5). Значення індексу зберігається в байті за адресою I1.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 Tabl1 6 dup (<>)**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати 8 байтів кожного поля namex масиву структур A1 в поле field8 масиву A2 із тим самим індексом. Забезпечить безумовну передачу управління із сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

5.

Заданий шаблон структури

```
Tabl1     struc
namex     db    4 dup (?)
field1    dw    4 dup (?)
Tabl1     ENDS
```

Заданий логічний сегмент

```
Data1 segment
I_struc          db    ?
I_namex          db    ?
A1   Tabl1 6 dup (<>)
Data1 ends
```

Продовження табл. 3.5

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: в кожний елемент масиву поля field1 (з індексом I_namex) структури із масиву структур A1 (з індексом I_struc) записати зміщення в сегменті елемента масиву поля namex (з індексом I_namex) структури з тим самим індексом у масиві структур.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 Tabl1 13 dup (<>)**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати весь масив структур A1 в масив структур A2, починаючи з 5-ої структури A2. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

6.

Заданий логічний сегмент

```
Data1 segment
Sump        dw    0
sumnp dw    0
A1    dw    6 dup (8 dup (7 dup (102h)))
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: накопичити в слові за адресою sump суму байтів масиву A1, які мають парне значення зміщення в сегменті, а за адресою sumnp - непарні значення зміщення в сегменті.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиви

```
A2    dw    3 dup (8 dup (2 dup (0)))
A3    dw    2 dup (4 dup (4 dup (0)))
```

3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд заповнити масив A2 значенням суми із Sump, а в масив A3 - із sumnp. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

7.

Заданий логічний сегмент Data1

```
Data1 segment
I1          db    0
I2          db    0
I3          db    0
I4          db    0
A1    dw    5 dup (6 dup (6 dup (5 dup (?))))
Data1 ends
```

Заданий логічний сегмент Data2

```
Data2 segment
A2    dd    5 dup (6 dup (6 dup (5 dup (0abcd0123h))))
Data2 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: пересилання двох "середніх" байтів (0CD01H) кожного чотирьохбайтного елемента масиву A2 у відповідний (тобто, з тими самими значеннями індексів) елемент масиву A1, використовуючи значення індексів масивів за адресами I1,I2,I3,I4.

2. Створити ще один логічний сегмент кодів (наприклад, з ім’ям Code2), в якому за допомогою ланцюгових команд підраховувалась би кількість байт логічного сегменту Data2, значення яких дорівнює 0abh. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

8.

Заданий логічний сегмент Data1

```
Data1 segment
I1          db    0
I2          db    0
I3          db    0
I4          db    0
A1    dw    2 dup (6 dup (7 dup (5 dup (?))))
Data1 ends
```

Заданий логічний сегмент Data2

```
Data2 segment
A2    dd    2 dup (6 dup (7 dup (5 dup (?))))
Data2 ends
```

1. Створити сегмент кодів (наприклад, з ім’ям Code1), в якому реалізувати наступне завдання: завантаження логічної адреси кожного елементу масиву A1 у відповідний (тобто, з тими самими значеннями індексів) елемент масиву A2. Для значення індексів використовувати байти за адресами I1,I2,I3,I4.
2. Створити ще один логічний сегмент кодів (наприклад, з ім’ям Code2), в якому за допомогою ланцюгових команд завантажити у всі елементи масиву A1 зміщення в сегменті елементу A2[0,1,2,3] масиву A2. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

9.

Заданий логічний сегмент

```
Data1 segment
I1    db    ?
I2    db    ?
I3    db    ?
A1    dw    5 dup (8 dup (2 dup (0)))
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім’ям Code1), в якому б програма реалізувала наступне завдання: якщо всі індекси за адресами I1,I2,I3 парні, тоді в елемент масиву A1[i1,i2,i3] записується зміщення в сегменті цього елементу, а якщо хоча б один з індексів непарний, тоді в елемент масиву A1[i1,i2,i3] записується сегментна частина логічної адреси цього елементу.
2. Створити ще один логічний сегмент даних (наприклад, з ім’ям Data2), в якому задати директиву **A2 dw   5 dup (8 dup (2 dup (0)))**
3. Створити ще один логічний сегмент кодів (наприклад, з ім’ям Code2), в якому за допомогою ланцюгових команд переписати масив A1 в A2 та підрахувати в масиві A2 кількість елементів, які містять сегментні частини логічних адрес. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

Продовження табл. 3.5

10.

Заданий шаблон структури

```
Node        struc
namex       db      5 dup (?)
field1      dw      5 dup (?)
Node        ENDS
```

Заданий логічний сегмент

```
Data1 segment
I_struc             db      ?
I_namex             db      ?
A1      Node    6 dup (<>)
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: в елементи масиву поля field1 (з індексом I_namex) структури із масиву структур A1 (з індексом I_struc) записати зміщення в сегменті інших структур із масиву структур A1 (формування повного орієнтованого графа, в якому вершини - це поля namex, а дуги - елементи масиву поля field1). Для доступу до слів масиву в полі field1 використовувати індекс I_namex. Для доступу до структур в масиві A1 використовувати індекс I_struc.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **Db 'node1','node2', ,'node3','node4','node5','node6'**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати імена вершин графа в поля namex структур масиву структур A1. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

11.

Заданий шаблон структури

```
Node        struc
namex       db      4 dup (0ffh)
field1      dw      2 dup (0)
Node        ENDS
```

Заданий логічний сегмент

```
Data1 segment
I_struc             db      ?
A0      Node    <>
A1      Node    2 dup (<>)
A2      Node    4 dup (<>)
A3      Node    8 dup (<>)
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: створити дерево, де в 2-х елементний масив поля field1 структури A0 записуються зміщення в сегменті 2-х структур із масиву структур A1. В 2-х елементний масив поля field1 структури A1[0] заносяться зміщення в сегменті структур A2[0] та A2[1] і т.д. При створенні програми для індексації масивів структур використовувати байт за адресоюі I_struc.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати наступні директиви:

```
B0      Node    <>
B1      Node    2 dup (<>)
B2      Node    4 dup (<>)
```

B3 Node 8 dup (<>)

3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати масиви структур A0-A4 в масиви структур B0-B4 та відкоригувати значення полів field1 так, щоб масиви структур B0-B4 також репрезентували дерево. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

12.

Заданий шаблон структури

```
Node        struc
namex       db      4 dup (?)
dst         dw      ?
Node        ENDS
```

Заданий логічний сегмент

```
Data1 segment
I_struc             db          ?
Order       db              0,4,2,1,3
A1          Node            5 dup (<>)
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: на базі масиву структур A1 створити кільцевий список, де в поле dst структури A1[0] записується зміщення в сегменті структури A1[4], в поле dst структури A1[4] записується зміщення в сегменті структури A1[2] і т.д. згідно з порядком, який заданий в масиві Order. В поле dst структури A1[3] записується зміщення в сегменті структури A1[0]. При створенні програми для індексації масивів структур використовувати байт за адресою I_struc.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 Node 5 dup (<>)**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати масив A1 в масив A2 та відкоригувати значення полів dst таким чином, щоб масив A2 також являв собою кільцевий список. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

13.

Заданий шаблон структури

```
Node        struc
namex       db      5 dup (?)
field1      dw       0
Node        ENDS
```

Заданий логічний сегмент

```
Data1 segment
I_struc             db      ?
A0      Node    <>
A1      Node    2 dup (<>)
A2      Node    4 dup (<>)
A3      Node    8 dup (<>)
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: створити дерево, де в поля field1 структур A3[0] та A3[1] записуються зміщення в сегменті структури A2[0], в поля field1 структур A3[2] та A3[3]

записуються зміщення в сегменті структури A2[1] і т.д., в поля field1 структур A3[6] та A3[7] записуються зміщення в сегменті структури A2[3]. Аналогічно заповнюються поля field1 в структурах із масивів A2 та A1. При створенні програми для індексації масивів структур використовувати байт за адресою I_struc.

2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати наступні директиви:

Db 'node0','node10', ,'node11,'node20','node21','node22', 'node23'
Db 'node30','node31','node32', 'node33','node34','node35','node36', 'node37'

3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати імена вершин дерева в поля namex структур, які формують дерево (структура A0, масиви структур A1-A3). Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

14.

Заданий шаблон структури

```
Node    struc
namex   db   'парне'
field1  dw   ?
Node    ENDS
```

Заданий логічний сегмент

```
Data1 segment
I1      db   ?
I2      db   ?
A1   Node  6 dup (4 dup<>)
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: в поле field1 структури із масиву структур A1 записати зміщення в сегменті наступної структури із масиву структур A1. Для доступу до структур в масиві A1 використовувати індекси за адресами I1 та I2. Наступною структурою для структури A1[5,3] вважати структуру A1[0,0]. Якщо значення в полі field1 виявилось непарним, тоді в поле namex записати 'непар'
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 Dw 24 dup (?)**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд послідовно переписати в масив A2 всі парні значення полів fild1 масиву структур A1. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

15.

Заданий логічний сегмент Data1

```
Data1 segment
I1      db   0
I2      db   0
I3      db   0
I4      db   0
A1   db   3 dup (2 dup (7 dup (5 dup (?))))
Data1 ends
```

Заданий логічний сегмент Data2

```
Data2 segment
II1     db   0
```

II2 db 0
II3 db 0
A2 db 7 dup (5 dup (2 dup (?)))
Data2 ends

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: занести в кожний елемент масиву A1 значення його зміщення відносно початку масиву, а потім переслати, використовуючи значення індексів масивів за адресами I1,I2,I3,I4 та II1,II2,II3, 17 елементів масиву A1, починаючи з елемента A1[1,0,3,2] в масив A2, починаючи з елемента A2[3,4,0].
2. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд виконати пересилання вказаних елементів масиву A1 в масив A2, починаючи з елемента A2[0,3,1]. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

16.

Заданий логічний сегмент Data1

Data1 segment
I1 db 0
I2 db 0
I3 db 0
I4 db 0
A1 dw 4 dup (7 dup (5 dup (2 dup (4567h))))
Data1 ends

Заданий логічний сегмент Data2

Data2 segment
A2 dd 4 dup (7 dup (5 dup (2 dup (23456789h))))
Data2 ends

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: пересилання слова 4567h кожного елемену масиву A1 в старше слово відповідного елементу масиву A2, використовуючи значення індексів масивів за адресами I1,I2,I3,I4.
2. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд підраховувалась би кількість байтів логічного сегменту Data2, значення яких дорівнює 67h. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

17.

Заданий логічний сегмент

Data1 segment
I1 db ?
I2 db ?
I3 db ?
A1 dw 6 dup (8 dup (7 dup (0)))
Data1 ends

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: в кожний елемент масиву записати суму індексів, значення яких містяться в байтах за адресами I1,I2,I3.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 dd 6 dup (8 dup (7 dup (0)))**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд знайти всі слова масиву A1, значення яких

дорівнює 5, а логічні адреси цих слів записати в елементи масиву A2 з тими самими індексами. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2.

18.

Заданий логічний сегмент

```
Data1 segment
Sumc4 dw    0
Sumnc4      dw    0
A1    dd    3 dup (5 dup (9 dup (1020201h)))
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: накопичити в слові за адресою sumc4 суму байтів масиву A1, в яких значення зміщення в сегменті кратне 4 (ділиться на 4), а за адресою sumnc4 - не кратне 4 (не ділиться на 4).
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиви:

```
A2    dw    2 dup (8 dup (4 dup (0)))
A3    dw    4 dup (4 dup (4 dup (0)))
```

3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд заповнити масив A2 значенням суми із Sumc4, а в масив A3 - із Sumnc4. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

19.

Заданий логічний сегмент Data1

```
Data1 segment
I1          db    0
I2          db    0
I3          db    0
I4          db    0
A1    dw    3 dup (4 dup (2 dup (5 dup (?))))
Data1 ends
```

Заданий логічний сегмент Data2

```
Data2 segment
A2    dd    3 dup (4 dup (2 dup (5dup (?))))
Data2 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: завантаження зміщення в сегменті старшого слова кожного елементу масиву A2 у відповідний (тобто, з тими самими значеннями індексів) елемент масиву A1. Для значення індексів використовувати байти за адресами I1,I2,I3,I4.
2. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд завантажити в усі елементи масиву A2 логічну адресу елемента A1[2,1,1,3] масиву A1. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

20.

Заданий шаблон структури

```
Tabl1       struc
namex       db    7 dup (?)
field1      db    ?
field2      dw    ?
field4      dd    ?
```

Tabl1 ENDS

Заданий логічний сегмент

Data1 segment
I1 db 0
A1 Tabl1 6 dup (<>)
Data1 ends

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: в кожне поле field2 структур із масиву структур A1 записати зміщення в сегменті поля field1 структури з тим самим індексом у масиві. Початкове значення індексу міститься в байті за адресою I1.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 Tabl1 11 dup (<>)**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати весь масив структур A1 в масив структур A2, починаючи з 2-ої структури. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

21.

Заданий логічний сегмент

Data1 segment
I1 db ?
I2 db ?
I3 db ?
A1 dw 4 dup (5 dup (6 dup (0)))
Data1 ends

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: якщо сума індексів за адресами I1,I2,I3 парна, тоді в елемент масиву A1[i1,i2,i3] записується сегментна частина логічної адреси цього елемента, а якщо хоча б один з індексів непарний, тоді в елемент масиву A1[i1,i2,i3] записується зміщення в сегменті цього елемента.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 dw 4 dup (5 dup (6 dup (0)))**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати масив A1 в A2 та підрахувати в масиві A2 кількість елементів, які не містять сегментні частини логічних адрес. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

22.

Заданий шаблон структури

Tabl1 struc
namex db 5 dup (1h)
field1 db ?
field2 dw ?
field8 db 'not error'
Tabl1 ENDS

Заданий логічний сегмент

Data1 segment
I1 db 1
A1 Tabl1 6 dup (<>)
Data1 ends

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне

Продовження табл. 3.5

завдання: в кожне поле field2 структур із масиву структур A1 записати суму байтів поля namex плюс зміщення в сегменті поля field1. Значення індексу масиву структур зберігається в байті за адресою I1 та змінюється від 0 до 5.

2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 Tabl1 6 dup (<>)**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати старші 6 байтів кожного поля field8 масиву структур A1 в поле field8 масиву A2 із тим самим індексом. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

23.

Заданий логічний сегмент

```
Data1 segment
I1      db      ?
I2      db      ?
I3      db      ?
A1      dd      4 dup (7 dup (3 dup (0),1234h))
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: усі слова трьохвимірного масиву A1 із значеннями 1234h замінити на суму наступних за зміщенням в сегменті 4-х слів шляхом використання індексів I1,I2,I3 (значення яких містяться в байтах за адресами I1,I2,I3).
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **A2 dd 4 dup (7 dup (4 dup (0)))**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому завдання п.1 виконується з використанням ланцюгових команд для пошуку значення 1234h. Після підрахунку порівняти значення в масивах A1 та A2. Забезпечити безумовну передачу управління із сегменту Code1 в сегмент Code2 з використанням непрямої адресації.

24.

Заданий шаблон структури

```
Node        struc
namex       db      5 dup (?)
left        dw      ?
right       dw      ?
Node        ENDS
```

Заданий логічний сегмент

```
Data1 segment
I_struc     db          ?
I_order     db          ?
Order       db          0,3,1,2,4
A1          Node        5 dup (<>)
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: на базі масиву структур A1 створити кільцевий двонаправлений список, для чого в поле left структури A1[0] записується зміщення в сегменті структури A1[4], а в поле right структури A1[0] – зміщення в сегменті структури A1[3]; в поле left структури A1[3] записується зміщення в сегменті структури A1[0], а в поле right структури A1[3] – зміщення в сегменті структури A1[1], і т.д. згідно з порядком, який заданий в масиві Order. При створенні програми для індексації масивів структур використовувати байт за адресою I_struc, а для доступу до елементів

Продовження табл. 3.5

масиву Order – байт за адресою I_order.

2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву **Db 'Null ','One ','Two ',Three','Four ''**
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд переписати імена елементів списку в поля namex структур, які формують список. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2 за допомогою непрямої міжсегментної передачі управління.

25.

Заданий логічний сегмент

```
Data1 segment
I1      db      ?
I2      db      ?
I3      db      ?
I4      db      ?
A1      dd      6 dup (5 dup (4 dup (3 dup (?))))
Data1 ends
```

1. Створити сегмент кодів (наприклад, з ім'ям Code1), в якому реалізувати наступне завдання: записати у всі елементи масиву A1 значення індексів згідно правила:

   A1[I1,I2,I3,I4]:= (I1 shl 24) or (I2 shl 16) or (i3 shl 8) or I4;

   Вважається, що індекс за адресою I1 змінюється від 0 до 2, за адресою I2 – від 0 до 3 і т.д.
2. Створити ще один логічний сегмент даних (наприклад, з ім'ям Data2), в якому задати директиву A2 dd 180 dup (0)
3. Створити ще один логічний сегмент кодів (наприклад, з ім'ям Code2), в якому за допомогою ланцюгових команд знайти та послідовно переслати в масив A2 всі 4-х байтні елементи масиву A1, що містять хоча б один байт зі значенням 2. Забезпечити безумовну передачу управління з сегменту Code1 в сегмент Code2.

Ускладнене завдання

1) Створити, налагодити та перевірити працездатність простого завдання згідно варіанту.
2) Скопіювати початковий файл простого завдання.
3) У копії додати новий логічний сегмент даних (наприклад, Date3) та новий логічний сегмент кодів (наприклад Code3), в якому б програма реалізувала наступне завдання:

- скопіювати сегмент даних Date2, а потім Date1 у сегмент Date3;
- скопіювати сегмент кодів Code2, а потім Code1 у сегмент Code3;
- відкоригувати адресні посилання у скопійованих програмах;
- виконати просте завдання у спільному сегменті Code3 з використанням спільного сегменту Date3.

3.4. Контрольні запитання

1. Як у програмі прочитати команди програми?
2. Що таке багатокомпонентна адреса і для чого вона використовується?
3. Як зміниться результат трансляції програми (порівняти лістинги), якщо задати наступну директиву ASSUME:

   ASSUME CS:CODE,DS: NOTHING,ES:DATA,SS:NOTHING

   Перевірити правильність виконання програми та пояснити результат перевірки.
4. Як треба змінити програму, щоб вона виконувалась без помилок при наступному операторі ASSUME:

   ASSUME CS:CODE, DS:NOTHING, ES:DATA, SS:NOTHING

ЛАБОРАТОРНА РОБОТА №4-1

Організація взаємозв'язку програм мовою Асемблера з програмами мовою Паскаль

Мета роботи – ознайомлення з організацією взаємозв'язку програм мовою Ассемблера з програмами мовою Паскаль, вивчення методів опрацювання складних структур даних

4-1.1. Зміст роботи

Робота виконується на двох заняттях. На першому занятті студенти, використовуючи приклади програм у файлах lab4.pas та BigShow.asm, вивчають правила взаємозв'язку окремо скомпільованих програм мовою Асемблера та мовою Паскаль. Шляхом модифікації Паскаль програми ознайомлюються з машинною структурою різних типів даних, також з індивідуальним варіантом завдання щодо реалізаціїї елементарних операцій з надвеликими цілими числами, які застосовуються в сучасній комп'ютерній криптографії.

На другому занятті складають власну підпрограму мовою Асемблера для виконання елементарної операції з надвеликими числами згідно варіанта, складають мовою Паскаль тестову програму з тестовими даними для виклику асемблерної процедури, використовують процедуру BigShow для відображення тестових даних і результатів виконання відповідної операції.

4-1.2. Теоретичні відомості

Взаємозв'язок програм мовою Асемблера і мовою Паскаль

Взаємозв’язок програм мовою високого рівня і програм мовою Асемблера здійснюється двома наступними методами:

- вставками асемблерного тексту в текст програми мовою високого рівня;

- використання окремо скомпільованих асемблерних процедур, які на мові високого рівня об’являються як зовнішні (*external*).

З першим методом студенти ознайомлювались у Лабораторній роботі №2. Використання же окремо скомпільованих асемблерних процедур пов’язане з виконанням додаткових спеціальних вимог до програм як мовою Паскаль, так і мовою Асемблера.

Додаткові вимоги до програм мовою Паскаль в інтегрованому середовищі Турбо Паскаль

1) В програмі повинна бути директива компілятора **L**, яка забезпечить підключення об'єктного файла, сформованого Асемблером (Tasm або Masm). Наприклад, *{$L BigShow.obj},* при цьому файл BigShow.obj повинен знаходитись у тому ж каталозі, що і компілятор Паскаля. Якщо файл BigShow.obj знаходиться в іншому каталозі, тоді необхідно вказати повний шлях деревом каталогу, наприклад, *{$L / ..шлях.. /BigShow.obj}*.
2) В розділі опису підпрограм і функцій Паскаль програми за правилами мови Паскаль повинні бути описані асемблерні процедури. Додатковою вимогою є доповнення опису кожної процедури ключовим словом *External*, наприклад:

```
Procedure BigShow(Var m; len:word); external;
Function Xyz(x,y:integer):integer; external;
```

3) Для виклику підпрограм використовується внутрішньосегментна або міжсегментна команда процесора *CALL*. Компілятор Паскаля сам вирішує, яку і коли команду використовувати відповідно до контексту програми і ключів компіляції інтегрованового середовища. Тип процедури (*near* чи *far*) мовою Асемблера компілятором при цьому не враховується. Тому, для запобігання неузгодженості і досягнення максимального універсалізму, рекомендується всі процедури мовою Асемблера оформляти за типом *far*, а в Паскаль-програмі перед описом асемблерних процедур помістити директиву far-компіляції *{$F+}***.** Після

опису асемблерних процедур рекомендується помістити директиву "ближньої" адресації *{$F-}*, наприклад:

```
{$F+}
Procedure BigShow(Var m; len:word); external;
Function Xyz(x,y:integer):integer; external;
{$F-}
```

Додаткові вимоги до програм мовою Асемблера

1) Поле операндів директиви END

Програма мовою Асемблера не повинна бути основною, а це означає, що поле операндів директиви *END* має бути порожнім.

2) Імена логічних сегментів

Повинні використовуватися певні імена логічних сегментів:

- _Text – для сегменту кодів;
- _Data – для сегменту даних, початкові значення яких визначаються під час завантаження;
- _Bss – для сегменту даних, значення яких під час завантаження не визначені.

Примітка. Не рекомендується використовувати інші назви логічних сегментів, навіть якщо це дозволяє інтегроване середовище Турбо Паскаль, оскільки вищезазначені назви є стандартними для всіх компіляторів, у тому числі і для інших мов програмування. Крім того, можуть виникнути проблеми з початковими значеннями даних.

3) Операнди директив SEGMENT

Тип об’єднання для всіх сегментів – *Public*.

Класи сегментів:

- для сегментів _Text – *'Code'*
- для сегментів _Data – *'Data'*
- для сегментів _Bss – *'Bss'*

Розрядність: за умови використання директиви .386 або будь-якої іншої директиви 32-розрядного мікропроцесора, обов’язково задається операнд *Use16*. Приклад подання директиви опису логічного сегмента кодів:

```
_Text    Segment    Word    Public    'Code'    Use16
```

4) Доступ до параметрів процедур

Параметри перед викликом процедури записуються в стек в порядку їх слідування (конвенція Паскаля): спочатку перший, потім другий і т.д., а після останнього параметра командою CALL в стек записується адреса повернення (рис.4-1.1).

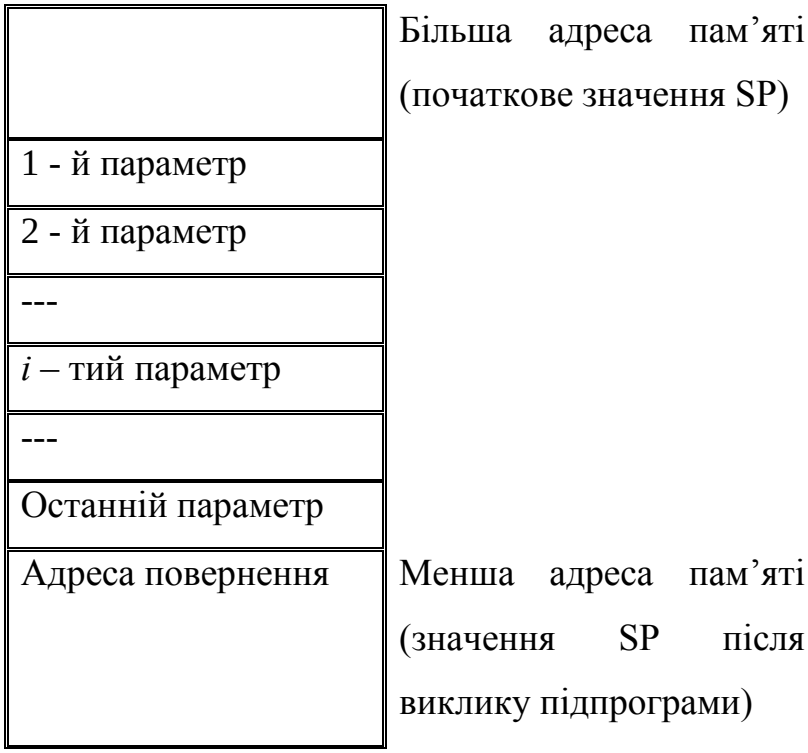

Рис.4-1.1. Послідовність запису параметрів в стек при виклику процедури

Можливі два варіанти запису параметра в стек:

- в стек записується повна логічна адреса параметра (сегментна складова та зміщення в сегменті) – 2 слова;
- в стек записується значення параметра – не більше 2-х слів (4-х байтів).

Перший варіант (повна логічна адреса) використовується у випадках:

- коли параметр передається за посиланням;

- коли параметр передається за значенням, але займає в пам’яті більше 4-х байтів.

Якщо параметр передається за значенням і займає в пам’яті один байт, тоді в стек записуються 2 байти. Таким чином, будь-який параметр займає в стеку одне або два слова.

Для доступу до параметрів процедур використовується базовий регістр стеку *BP*. Оскільки в програмах широко використовується виклик одних процедур з інших, то необхідно зберегти попередній вміст цього регістра, а потім записати в нього вміст покажчика стеку (*SP*):

```
push    bp
mov     bp, sp
```

В результаті, для адресації останнього аргументу можна використати адресний вираз [BP+6] (4 байти – адреса повернення і ще два байти – попередній вміст регістра BP). Для адресації решти аргументів (пробігаючи в зворотному напрямку від останнього до першого) необхідно кожний раз збільшувати константу на 2 або 4, в залежності від кількості байтів попереднього аргументу. Нехай, наприклад, маємо наступний опис зовнішньої асемблерної процедури в програмі мовою Паскаль:

```
Procedure pp(var arg1, arg2; arg3:char, arg4:integer); external;
```

Тоді адреса аргументу arg4 в стеку буде [BP+6], аргументу arg3 – [BP+8], аргументу arg2 – [BP+10], аргументу arg1 – [BP+14]. При багаторазових звертаннях до аргументів в асемблерній програмі доцільно задати наступну послідовність директив *EQU*:

```
Arg4    EQU     [BP+6]
Arg3    EQU     [BP+8]
Arg2    EQU     [BP+10]
Arg1    EQU     [BP+14]
```

або

```
Arg4    EQU   word ptr [BP+6]
Arg3    EQU   byte ptr [BP+8]
Arg2    EQU   dword ptr [BP+10]
Arg1    EQU   dword ptr [BP+14]
```

За умови повного розуміння механізму передачі та доступу до параметрів, можна використати спеціальну директиву Асемблера *ARG*, яка заміняє дії вищезазначених директив EQU:

```
ARG   Arg4:Word, Arg3:Byte, Arg2:Dword, Arg1:Dword
```

5) Вміст регістрів процесора

Вміст усіх сегментних регістрів, а також регістра BP(EBP) повинен не змінюватись або бути відновленим при поверненні з підпрограми.

6) Доступ до змінних програми мовою Паскаль

За умови незмінності вмісту регістра DS, ідентифікатори змінних, визначених в розділі опису змінних глобального блоку мовою Паскаль, можна використовувати в асемблерній програмі як символічні адреси (зміщення в сегменті даних). Для цього в асемблерній програмі такі ідентифікатори необхідно визначити в директиві *Extrn*, що має наступний формат:

```
Extrn   ім'я:тип, ...,  ім'я:тип
```

Тут мається на увазі асемблерний тип: byte, word, dword (необхідно знати кількість байтів, що відводиться для змінних Паскаля).

7) Доступ до процедур і функцій програми мовою Паскаль

Для забезпечення доступу ідентифікатори необхідних паскальних процедур і функцій необхідно визначити в директиві Extrn з типом *near* або *far*. Тип *far* задається для процедур і функцій, описаних в секції Interface, або при явному використанні режиму far-компіляції (наприклад, шляхом завдання директиви **{$F+}**).

Для надійного визначення типу рекомендується за допомогою методики, викладеній в Лабораторній роботі №2-1, визначити код команди *RET*, яка міститься в скомпільованій Паскаль процедурі, а потім по коду визначити, чи вона є міжсегментною, чи внутрішньосегментною.

8) Організація повернення з підпрограми

У відповідності до конвенції Паскаля, на підпрограму покладаються обов'язки щодо забезпечення початкового значення SP. Тому підпрограма повинна закінчуватись командами:

```
pop   bp
ret   const
```

Зауважимо, що команда *ret const* після виконання дії повернення додає до вмісту регістра SP значення const. Значення const обчислюється шляхом додавання кількості байтів, які відведені в стеку на кожний параметр. Наприклад, для раніше розглянутої процедури

Procedure pp(var arg1,arg2;arg3:char,arg4:integer); External;

значення const буде дорівнювати 12 (4 байти на аргумент arg1, 4 байти на аргумент arg2, 2 байти на аргумент arg3 і 2 байти на аргумент arg4), а закінчення асемблерної процедури буде мати вигляд:

```
pop   bp
ret   12
```

Директива *ARG* дозволяє доручити обчислення const Асемблеру. Для цього необхідно закінчити директиву виразом *=ідентифікатор,* наприклад:

```
ARG   Arg4:Word,Arg3:Byte,Arg2:Dword,Arg1:Dword=arg_size
```

Тоді закінчення асемблерної процедури буде таким:

```
pop   bp
ret   arg_size
```

9) Повернення значень функцій

Якщо мовою Асемблера необхідно реалізувати процедуру-функцію, яка використовується в Паскаль програмі, тоді перед поверненням із асемблерної процедури значення функції необхідно розмістити:

- в регістрі *AL*, якщо на паскальний тип даних відводиться один байт;
- в регістрі *AX*, якщо на паскальний тип даних відводиться одно слово;
- в парі регістрів *DX* і *AX*, якщо на паскальний тип даних відводиться подвійне слово, при цьому в регістрі *DX* розміщується старша частина даних або сегментна частина логічної адреси.

4-1.3. Приклад організації взаємодії програми мовою Паскаль і програми на Асемблері

Як приклад організації взаємодії програм мовою Паскаль і мовою Асемблера пропонується програма lab4.pas і відповідно програма BigShow.asm. Програма lab4.pas містить визначення мовою Паскаль двох цілих беззнакових чисел великої розрядності – байтових масивів x, y – та їх початкове заповнення, а також виклики процедури BigShow для відображення на екрані значень цих чисел у 16-ковому форматі.

```
Program lab4(input,output);
var
    i    : word;
    x    :array [1..2000] of byte;
    y    :array [1..1000] of word;
{$L bigshow.obj}
{$F+}
 Procedure BigShow(var p1;p2:word);external;
{$f-}
 begin  {Main program}
 for i:=1 to 300 do
 begin
  x[i]:=i;
  y[i]:=i;
 end;
 for i:=1 to 30 do
 begin
   writeln('x= ');
   BigShow(x,301-i);
   writeln('y= ');
   BigShow(y,301-i);
   readln;
 end;
  end.
```

В оперативному запам’ятовуючому пристрої дані цілого беззнакового типу великої розрядності займають *k* комірок, де *k* - довільне значення. Нехай A – адреса даних такого типу. Тоді адреси комірок пам’яті та нумерацію двійкових розрядів надвеликого числа можна подати наступним чином:

A+k-1	...	A+i-1	...	A+1	A
b_{k*8-1} $b_{(k-1)*8}$	...	b_{i*8-1} $b_{(i-1)*8}$	...	b_{15} b_8	b_7 b_0

Значення B такого числа визначається стандартним чином:

$$B = \sum_{J=0}^{k*8-1} b_i * 2^j$$

Мова Паскаль не підтримує такий тип даних. Для подання мовою Паскаль даних надвеликого цілого беззнакового типу доцільно використовувати байтові масиви. Тобто, один байтовий масив використовується для вмісту ОДНОГО надвеликого цілого беззнакового числа.

Процедура BigShow реалізована як асемблерна процедура в окремому програмному модулі BigShow.asm. Вона працює з надвеликими цілими додатними числами, які розміщуються у байтових масивах. Процедура призначена для перевірки правильності результатів виконання завдання.

Процедура BigShow має два параметри: перший із них – повна логічна адреса байтового масиву, другий параметр передається за значенням і задає кількість байтів у масиві. Програма виводить байти масиву на екран у шістнадцятковому форматі. Байти групуються при відображенні у подвійні слова (8 шістнадцяткових символів для подвійного слова). Байт з найменшою адресою (заданою першим параметром) завжди виводиться в крайній правій позиції останнього рядка, що зручно для зорового порівняння двох масивів.

Для виведення на екран використовується функція MS-DOS 02h. Для виклику функції використовується команда програмного переривання Int 21h. Параметром виклику є символ ASCII, який необхідно записати в регістр DL. Номер функції (02h) розміщують в регістрі AH.

```
; програмний модуль BigShow.asm
.386
_text    segment word public 'text' use16
        assume  cs:_text
;
;****************************************
; п/п виведення на екран в hex-форматі даних із регістра ebx:
; якщо di=28 – виводяться усі 4 байти
; якщо di=20 – виводяться 3 молодших байти
; якщо di=12 – виводяться 2 молодших байти
; якщо di=4 – виводиться один молодший байт
 show_bt        proc
        pushad
        mov       cx,di
```

```
        mov     ah,2
bt0:
        mov     edx,ebx
        shr     edx,cl
        and     dl,00001111b
        cmp     dl,10
        jl      bt1
        add     dl,7
bt1:
        add     dl,30h
        int     21h
        sub     cl,4
        jnc     bt0
        popad
        ret
show_bt     endp

BigShow     proc    far                 ; procedure BigShow(var mas, len:word)
      public    BigShow
; mas - адреса байтового масиву
@mas        equ     [bp+8]              ; адреса адреси
; len - кількість байт масива, які необхідно вивести на екран
@len        equ     [bp+6]              ; адреса кількості

        push    bp
        mov     bp,sp                   ; базова адреса фактичних параметрів
;  перехід на новий рядок екрану
        mov     ah,2
        mov     dl,13
        int     21h
        mov     dl,10
        int     21h
; обчислення кількості пробілів у першому рядку
        mov     ax,@len
        test    ax,00000011b
        pushf
        shr     ax,2
        popf
        jz      @1
        inc     ax
@1:
        xor     cx,cx
        mov     di,28
        and     ax,00000111b
        jz      @2
; формування пробілів на відсутніх подвійних словах
        mov     ah,8
        sub     ah,al
        mov     al,ah
        xor     ah,ah
        imul    ax,8+1
        mov     cx,ax
```

```
@2:
        mov     dx,@len
        and     dx,00000011b
        jz      l000
; формування початкового значення кількості зсувів
        mov     di,dx           ;di -  1  2   3
        dec     di              ;di -  0  1   2
        shl     di,3            ;di -  0  8  16
        add     di,4            ;di -  4  12 20
;  формування пробілів на відсутніх байтах у подвійному слові
        mov     dh,4
        xchg    dh,dl           ;dh -  1  2  3
        sub     dl,dh           ;dl -  3  2  1
        shl     dl,1            ;dl -  6  4  2
        xor     dh,dh           ;dx -  6  4  2
        add     cx,dx
l000:
        jcxz    l002
; виведення початкових пробілів у першому рядку
l001:
        mov     ah,2
        mov     dl," "
        int     21h
        loop    l001
l002:
        mov     cx,@len
        shr     cx,2
        cmp     di,28
        jz      @3
        inc     cx
@3:
        xor     esi,esi
        lds     si,@mas
        lea     esi,[esi+ecx*4]-4
        std
; виведення  масиву
l004:
        lodsd
        mov     ebx , eax
        call    show_bt
        mov     di,28
        mov     ah , 2
        mov     dl,20h
        int     21h
        dec             ecx
        test    ecx,7
        jne     l005
; перехід на новий рядок
        mov      ah,2
        mov      dl,13
        int     21h
        mov     dl,10
```

```
        int     21h
l005:
        jcxz    L006
        jmp     l004
l006:
        mov     ah,2
        mov     dl,13
        int     21h
        mov     dl,10
        int     21h
 ; mov   ah,1
 ; int   21h
        pop     bp
        ret     6
BigShow endp
_text   ends
        end
```

Програма lab4.pas містить всі необхідні елементи для забезпечення зв'язку з асемблерною процедурою BigShow. Вона демонструє незалежність процедури BigShow від паскального типу даних. Це означає, що процедура BigShow (або подібні їй процедури) можна використовувати також для аналізу машинного формату типів даних мови Паскаль. Наприклад, за допомогою процедури BigShow легко визначається формат логічних значень *True* та *False*.

4-1.4. Завдання на виконання роботи

Перше заняття

1) Скопіювати програми lab4.pas та BigShow.asm в окремий робочий каталог. Ознайомитись з їх призначенням і вмістом.
2) Протранслювати за допомогою tasm (або masm) програму мовою Асемблера BigShow.asm. В директиві L програми мовою Паскаль lab4.pas відкоригувати (при необхідності) шлях до файлу BigShow.obj. Відкомпілювати Паскаль-програму (разом з підключеним файлом BigShow.obj) та перевірити її працездатність.
3) Вивчити правила взаємозв'язку окремо скомпільованих програм мовою Асемблера та мовою Паскаль (див. Теоретичні відомості);

4) Розробити алгоритм реалізації операції з надвеликими цілими додатними числами згідно варіанта завдання (табл. 4-1.1).

Друге заняття

Розробити програму мовою Паскаль і програму мовою Асемблера згідно варіанта завдання та наступних вимог:

1) Програма мовою Паскаль повинна:

- відповідати вимогам зв'язку з асемблерними процедурами;
- містити визначення байтових масивів та їх початкове заповнення;
- містити виклики процедури BigShow для відображення початкових даних,
- містити виклик розробленої асемблерної процедури з відповідними параметрами;
- містити виклики процедури BigShow для відображення результатів;
- перед викликом процедури BigShow у Паскаль програмі забезпечити виведення на екран текстових повідомлень (коментарів).

2) Програма мовою Асемблера повинна:

- розміщуватися в початковому асемблерному модулі;
- містити процедуру відображення BigShow.asm і власну асемблерну процедуру, що виконує ту чи іншу елементарну операцію (згідно варіанта) з надвеликими цілими додатними числами, які розміщуються у байтових масивах. Тобто, один байтовий масив Паскаль програми використовується для вмісту ОДНОГО надвеликого цілого беззнакового числа.
- щоб уможливити виклик власної асемблерної процедури з програми мовою Паскаль, вона повинна відповідати спеціальним вимогам (див. Теоретичні відомості);
- на початку модуля мовою Асемблера розмістіти директиву Title із зазначенням групи та прізвища студента;

3) *Використати процедуру BigShow.asm* для відображення і перевірки коректності роботи розробленої програми на різних тестових наборах значень байтових масивів.

4) Додаткові експерименти:

- визначити, чи може програмний модуль мовою Асемблера, який об’єднується з Паскаль програмою, мати додаткові логічні сегменти з довільними іменами;
- визначити порядок передачі до функції значення типу **String.**

Таблиця 4-1.1

Варіанти завдання

1.	Розробити процедуру **Big2sAdd(var M1,M2;len:word)**, де M1,M2 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M1+M2. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використать команди для 8 - розрядних даних.. Вважати, що M1 і M2 знаходяться в різних сегментах.
2.	Розробити процедуру **Big2Add(var M1,M2,Carry;len:word)**, де M1,M2 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M1+M2. Змінній байтового типу Carry присвоюється значення 1 в разі переповненя і 0 при його відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1,M2 знаходяться в одному сегменті.
3.	Розробити функцію **FBig2Add(var M1,M2;len:word):Boolean**, де M1,M2 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M1+M2. Функції **FBig2Add** присвоюється значення False в разі переповненя і True при його відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1,M2 і Carry знаходяться в одному сегменті
4.	Розробити процедуру **Big3sAdd(var M1,M2,M3;len:word)**, де M1,M2,M3 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M2+M3. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1,M2 і M3 знаходяться в різних сегментах.

Продовження табл. 4-1.1

5. Розробити процедуру **Big3Add(var M1,M2,M3,Carry;len:word)**, де M1,M2,M3 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M2+M3. Змінній байтового типу Carry присвоюється значення 1 в разі переповненя і 0 при його відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1,M2,M3 і Carry знаходяться в одному сегменті.
6. Розробити функцію **FBig3Add(var M1,M2,M3;len:word):Boolean**, де M1,M2,M3 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M2+M3. Функції FBig3Add присвоюється значення False в разі переповненя і True при його відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1,M2,M3 знаходяться в одному сегменті.
7. Розробити процедуру **Big2sSub(var M1,M2;len:word)**, де M1,M2 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M1-M2. . Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використать команди для 8 - розрядних даних.. Вважати, що M1 і M2 знаходяться в різних сегментах.
8. Розробити процедуру **Big2Sub(var M1,M2,Carry;len:word)**, де M1,M2 надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M1-M2. Змінній байтового типу Carry присвоюється значення 1 при наявності позики і 0 при її відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1, M2 і Carry знаходяться в одному сегменті.
9. Розробити функцію **FBig2Sub(var M1,M2;len:word):Boolean**, де M1,M2 надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M1-M2. Функції Fbig2Sub присвоюється значення False в разі наявності позики і True при її відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1, M2 знаходяться в одному сегменті.
10. Розробити процедуру **Big3sSub(var M1,M2,M3;len:word)**, де M1,M2,M3 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M2-M3. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використать команди для 8 - розрядних даних.

Продовження табл. 4-1.1

Вважати, що M1,M2 і M3 знаходяться в різних сегментах.
11. Розробити процедуру **Big3Sub(var M1,M2,M3,Carry;len:word)**, де M1,M2,M3 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M2-M3. Змінній байтового типу Carry присвоюється значення 1 при наявності позики і 0 при її відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1,M2,M3 і Carry знаходяться в одному сегменті.
12. Розробити процедуру **FBig3Sub(var M1,M2,M3;len:word)**:Boolean, де M1,M2,M3 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - M1=M2-M3 Функції **Fbig3Sub** присвоюється значення False в разі наявності позики і True при її відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використать команди для 8 - розрядних даних. Вважати, що M1,M2,M3 знаходяться в одному сегменті.
13. Розробити функцію **Biggr(var M1,M2;len:word):Boolean**, де M1,M2 - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - якщо M1 > M2 то значення Biggr - True, інакше -False . Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використать команди для 8 - розрядних даних. Вважати, що M1 і M2 знаходяться в одному сегменті.
14. Розробити функцію **Biggreq (var M1,M2;len:word):Boolean**, де M1,M2 - - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - якщо M1 ≥ M2 то значення Biggreq - True, інакше - False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використать команди для 8 - розрядних даних. Вважати, що M1 і M2 знаходяться в одному сегменті.
15. Розробити функцію **Bigeq (var M1,M2;len:word):Boolean**, де M1,M2 - - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - якщо M1 = M2 то значення Bigeq - True, інакше - False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використать команди для 8 - розрядних даних. Вважати, що M1 і M2 знаходяться в одному сегменті.
16. Розробити функцію **Bigne (var M1,M2;len:word):Boolean**, де M1,M2 - - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - якщо M1 ≠ M2 то значення Bigne - True, інакше - False. Повинні використовуватись команди для 32-розрядних

Продовження табл. 4-1.1

даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використать команди для 8 - розрядних даних. Вважати, що M1 і M2 знаходяться в одному сегменті.
17. Розробити функцію **Bigles (var M1,M2;len:word):Boolean**, де M1,M2 - - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - якщо M1 < M2 то значення Bigles - True, інакше - False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використать команди для 8 - розрядних даних. Вважати, що M1 і M2 знаходяться в одному сегменті.
18. Розробити функцію **Bigleseq (var M1,M2;len:word):Boolean**, де M1,M2 - - надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - якщо M1 ≤ M2 то значення Bigleseq - True, інакше - False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використать команди для 8 - розрядних даних. Вважати, що M1 і M2 знаходяться в одному сегменті.
19. Розробити процедуру **BigShlCount(var M1;len,count: word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len), count - кількість розрядів зсуву. Операція - лінійний зсув вліво (в сторону старших розрядів) на кількість двійкових розрядів, яка задана параметром count. При цьому count старших розрядів втрачаються, а в count молодших розрядів заноситься 0. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для останніх байт використать команди для 8 - розрядних даних.
20. Розробити процедуру **BigShrCount(var M1;len,count: word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len), count - кількість розрядів зсуву. Операція - лінійний зсув вправо (в сторону молодших розрядів) на кількість двійкових розрядів, яка задана параметром count. При цьому count молодших розрядів втрачаються, а в count старших розрядів заноситься 0. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для останніх байт використать команди для 8 - розрядних даних.
21. Розробити процедуру **BigRolCount(var M1;len,count: word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len), count - кількість розрядів зсуву. Операція - циклічний зсув вліво (в сторону старших розрядів) на кількість двійкових розрядів, яка задана параметром count. При цьому count старших розрядів поступають на місце молодших розрядів. Повинні використовуватись команди для 32-розрядних

Продовження табл. 4-1.1

даних. Якщо значення len не кратно 4, то при необхідності для останніх байт використать команди для 8 - розрядних даних.
22. Розробити процедуру **BigRorCount(var M1;len,count: word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len), count - кількість розрядів зсуву. Операція - циклічний зсув вправо (в сторону молодших розрядів) на кількість двійкових розрядів, яка задана параметром count. При цьому count молодших розрядів поступають на місце старших розрядів. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для останніх байт використать команди для 8 - розрядних даних.
23. Розробити процедуру **BigZeroShl(var M1,cnt;len: word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len), cnt - кількість розрядів зсуву - змінна типу word. Операція - лінійний зсув вліво (в сторону старших розрядів) до тих пір поки в len*8-1 розряді не з'явиться одиничка. Кількість зсувів записується в параметр cnt. Якщо в початковому значенню числа M1 розряд len*8-1 дорівнює 1, то зсуви не виконуються, а в параметр cnt записується нуль.
24. Розробити процедуру **BigZeroShr(var M1,cnt;len: word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len), cnt - кількість розрядів зсуву. Операція - лінійний зсув вправо (в сторону молодших розрядів) до тих пір поки в молодшому розряді числа не з'явиться одиничка. Кількість зсувів записується в параметр cnt. Якщо в початковому значенню числа M1 молодший розряд дорівнює 1, то зсуви не виконуються, а в параметр cnt записується нуль.
25. Розробити процедуру **BigShl(var M1,Carry;len:word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len), Carry - змінна типу byte. Операція - лінійний зсув вліво (в сторону старших розрядів) на один розряд. При цьому в змінну Carry заноситься значення len*8-1 розряду числа M1, а в молодший розряд числа M1 заноситься 0.
26. Розробити процедуру **BigShr(var M1,Carry;len:word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len), Carry - змінна типу byte. Операція - лінійний зсув вправо (в сторону молодших розрядів) на один розряд. При цьому в змінну Carry заноситься значення молодшого розряду числа M1, а в старший розряд числа M1 заноситься 0.
27. Розробити функцію **FcBigShl(var M1;len:word):Byte**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len). Операція - лінійний зсув вліво (в сторону

Продовження табл. 4-1.1

старших розрядів) на один розряд. При цьому функція **FcBigShl** приймає значення len*8-1 розряду числа M1, а в молодший розряд числа M1 заноситься 0.
28. Розробити функцію **FBigShl(var M1;len:word):Boolean**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len). Операція - лінійний зсув вліво (в сторону старших розрядів) на один розряд. При цьому функція FBigShl приймає значення False, якщо len*8-1 розряд числа M1 до зсуву дорівнює 1 і True в протилежному випадку. В молодший розряд числа M1 при зсуві заноситься 0.
29. Розробити процедури **BigSetBit(var M1;len,number: word)** та **BigClrBit(var M1;len,number: word)**, де M1 - надвелике ціле додатнє число (байтовий масив довжиною len)., number - номер двійкового розряду числа M1, починаючи з 0. Операція - записать одиничку в розряд number для процедури BigSetBit і 0 для процедури BigClrBit.
30. Розробити процедуру **Extract(var M1,M2;len,ibeg,iend:word)**, надвеликі цілі додатні числа (байтові масиви довжиною len), ibeg,iend - номера розрядів, такі, що len*8-1 ≥ iend ≥ ibeg. Операція - виділить із числа M1 розряди з ibeg по iend включно та одержане таким чином число присвоїть M2. В старші розряди числа M2 занести 0.

4-1.5. Контрольні запитання

1. Чому поле операндів директиви END в модулі мовою Асемблера повинно бути порожнім?
2. Які імена логічних сегментів повинна мати програма мовою Асемблера для забезпечення зв’язку з програмами мовами високого рівня?
3. За якою адресою оперативної пам’яті (більшою чи меншою) буде розташований перший фактичний параметр для процедури, яка визивається Паскаль програмою?
4. Чи можливий в програмі мовою Асемблера виклик процедур мовою Паскаль?
5. До яких змінних програми мовою Паскаль можливий доступ в програмі мовою Асемблера і як він забезпечується?
6. Яка програма (та що викликає чи та яку викликають) відповідає в Паскалі за відновлення вмісту покажчика стеку – вмісту регістра SP?

ЛАБОРАТОРНА РОБОТА №4-2

Організація взаємозв'язку програм мовою Асемблера з С++ програмами

Мета роботи – ознайомлення з організацією взаємозв'язку програм мовою Асемблера з програмами мовою С++, вивчення методів опрацювання складних структур даних.

4-2.1. Зміст роботи

Робота виконується на двох заняттях. На першому занятті студенти, використовуючи приклади програм lab4.cpp та BigShowN.asm, вивчають правила взаємозв'язку програм мовою С++ і мовою Асемблера. Шляхом модифікації програми lab4.cpp ознайомлюються з машинною структурою різних типів даних, також з індивідуальним варіантом завдання по реалізації елементарних операцій з надвеликими цілими числами, які застосовуються у сучасній комп'ютерній криптографії для електронного цифрового підпису та при створенні віртуальних захищених каналів у відкритих комп'ютерних мережах.

На другому занятті студенти складають мовою С++ програму, що містить виклик самостійно реалізованої мовою Асемблера підпрограми для виконання елементарної операції з надвеликими числами, а також виклик асемблерної процедури BigShowN для відображення тестових даних і результатів виконання операції.

4-2.2 Теоретичні відомості

Правила взаємозв'язку програм мовою С++ і мовою Асемблера

Взаємозв'язок програм мовою високого рівня і програм мовою Асемблера здійснюється двома методами:

- вставками асемблерного тексту в текст програми мовою високого рівня;
- використанням окремо скомпільованих асемблерних процедур, які мовою високого рівня об’являються як зовнішні.

З першим методом студенти знайомились у Лабораторній роботі №2-2. Завданням даної лабораторної роботи є створення зовнішніх для С++ програми асемблерних процедур. Для його вирішення необхідно перш за все з’ясувати наступні питання:

- як передати параметри з С++ програми в асемблерну процедуру;
- як звернутися до цих параметрів в асемблерній процедурі;
- як повернути результат роботи асемблерної процедури у програму мовою С++.

Крім того, необхідно знати, яким спеціальним додатковим вимогам мають вдовольняти програми як мовою С++, так і мовою Асемблера.

Загальні принципи організації зв’язку С++ – Асемблер нагадують розглянуті у Лабораторній роботі №4-1 правила Паскаль – Асемблер, проте є і відмінності, і додаткові можливості. Стандартні можливості зв’язку надаються при організації зв’язку *в стилі С*, решта може залежати від компілятора С++ в обраному середовищі програмування. Дана лабораторна робота передбачає роботу в інтегрованому середовищі Visual Studio та відповідно з компілятором Visual C++.

Розглянемо основні принципи організації зв’язку С++ – Асемблер.

1) Передавання параметрів з С++ програми в асемблерну процедуру

Передавання параметрів в асемблерну процедуру здійснюється через стек (як і у випадку мови Паскаль). Проте, на відміну від Паскаля, параметри С++ функцій передаються в стек *у зворотному порядку*. Тобто першим в стек записується останній параметр, відповідно останнім – перший параметр функції. Передаватися може значення параметра чи його адреса. Далі в стек записується адреса повернення в С++ програму та здійснюється сам виклик функції.

2) Доступ до переданих з C++ програми параметрів в асемблерній процедурі

Для доступу до параметрів, що знаходяться у стеку, в асемблерній процедурі використовують базовий регістр стеку **EBP**. Оскільки в програмах широко використовується виклик одних процедур з інших, попередній вміст цього регістра необхідно зберегти, а потім записати в нього вміст покажчика стеку регістра **ESP** (настроївши, таким чином, його на останні записані в стек дані):

```
push    EBP
mov     EBP, ESP
```

Після цього доступ до параметрів у стеку здійснюється за зміщенням відносно вмісту регістра **EBP**. Тобто, для адресації будь-якого аргументу можна використати адресний вираз **[EBP+зміщення]**, який визначає місце розташування аргументу в стеку.

Збережений вміст регістра **EBP** має відновлюватися перед виходом з асемблерної процедури. Тому наприкінці процедури повинна бути команда

```
pop     EBP
```

Після повернення з асемблерної процедури вміст покажчика стеку ESP повинен також відновлюватися – приймати значення, яке було перед записом цих параметрів в стек та викликом функції. Таке очищення стеку (вивільнення його від параметрів) здійснюється C++ функцією, а не асемблерною процедурою. (Інші (не C) конвенції можуть вимагати очищення стеку саме від асемблерної процедури. Тоді закінчення асемблерної процедури виглядатиме наступним чином:

```
pop     EBP
ret     const
```

де const – розмір параметрів процедури у байтах.)

3) Повернення результату з асемблерної процедури у C++ програму

Повернення результату асемблерної процедури (тобто значення C++ функції) здійснюється через регістр-акумулятор, куди перед поверненням з процедури необхідно помістити отриманий результат.

Вимоги до програми мовою C++

1) Оголошення зовнішніх асемблерних процедур

Прототипи асемблерних процедур (а для C++ це є функції) повинні бути описані за правилами мови C/C++. Крім того, для забезпечення зв'язку *в стилі C* як стандартному, потрібно доповнити їх опис ключовим словом *extern* з додатком *"C"*. Наприклад:

```
extern "C"
{
    void BigShowN(byte* p1, short p2);
    byte* AsmFunc(byte* m1, byte* m2, short sz);
}
```

Тоді функції будуть відповідним чином сформовані і зможуть легко під'єднатися до асемблерної програми. Додаткові можливості зв'язку в стилі C++ можуть бути розглянуті студентами самостійно.

2) Опис глобальних змінних

Глобальні змінні, що можуть використовуватися в асемблерній програмі, потрібно також описати в *extern* з додатком *"C"*, наприклад:

```
extern "C"{
    short number = 255;
    bool flag;
}
```

3) Приклад передавання параметрів з C++ програми в асемблерну процедуру

Згідно з вищезазначеними принципами зв'язку програм C++ – Асемблер, перед безпосереднім викликом функції її параметри заносяться в стек, починаючи з останнього параметра та закінчуючи першим. Потім в стек записується адреса повернення в C++ програму. Наприклад, маємо наступний прототип функції мовою C++:

```
void pp(char* arg1, char arg2, short arg3, short arg4);
```

Тоді загальний вигляд її виклику:

```
pp(&arg1, arg2, arg3, arg4);
```

Першим в стек записується значення параметра arg4, далі значення arg3, значення arg2 і останнім – адреса arg1.

Мовою Асемблера реалізація виклику мала би наступний вигляд:

```
push arg4                ; запис в стек значення arg4
push arg3                ; запис в стек значення arg3
push arg2                ; запис в стек значення arg2
push offset arg1         ; запис в стек адреси arg1
call pp          ; запис в стек адреси повернення та виклик процедури pp
add sp, const ; вивільнення стеку від параметрів, const=розмір параметрів
```

Вимоги до програми мовою Асемблера

1) Директива завдання набору допустимих команд процесора

У Visual Studio для асемблерних програм за замовчуванням встановлена директива *.686*, яка визначає набір інструкцій процесора (аж до 686), які можуть бути використані у вихідному коді програми. Проте, для виконання цієї лабораторної роботи, де мова йдеться про звичайну користувацьку програму, достатньо вказати директиву *.386*.

2) Директива public

Звичайно для забезпечення беззаперечної видимості (загальнодоступності) асемблерної процедури у програмі мовою C++ потрібно в асемблерній програмі використати директиву *public* з ім'ям процедури, наприклад:

```
public pp
```

Проте, Visual Studio забезпечує абсолютну видимість асемблерних процедур, що входять до складу проекту, і без використання директиви *public*, тобто, у даному випадку, її використання не є обов'язковим.

3) Модель пам'яті

Для 32-розрядних програм використовується тільки пласка модель пам'яті *flat*, яка підтримується компілятором C++. *Модель flat* передбачає організацію пам'яті у вигляді неперервного лінійного (несегментованого) адресного простору. Є один великий (з максимальним розміром 4 Гбайт)

сегмент пам’яті, а в якості адрес використовується *тільки 32-бітні зміщення* відносно базової адреси. Таким чином, будь-яка адреса займає 4 байти.

Програма мовою Асемблера може мати наступні складові (хоча усі логічні сегменти, крім *.code*, є необов’язковими):

```
        .data
; ініціалізовані дані
        .data?
; неініціалізовані дані
        .const
; константи
        .code
; код програми
```

Сегменти автоматично об’єднуються в групу з ім’ям *flat*, а сегментні регістри CS, DS, SS відповідно настроюються на усю цю групу.

Щоб застосувати модель пам’яті *flat* з урахуванням угод мови високого рівня щодо виклику процедур, потрібно на початку програми записати директиву *model* з модифікатором мови C:

```
.model flat, C
```

4) Доступ до переданих C++ параметрів в асемблерній процедурі

Згідно з вищезазначеними правилами передавання параметрів і виклику асемблерної процедури з C++ програми, стек при входженні в асемблерну процедуру буде мати наступний стан (рис.4-2.1):

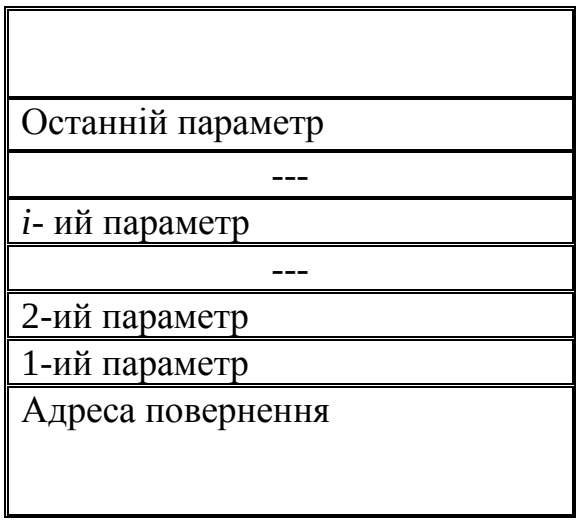

Рис.4-2.1. Стан стеку після виклику процедури

За меншою адресою, на яку вказує покажчик стеку ESP, знаходитиметься адреса повернення в С++ програму, потім перший параметр, далі другий і т.д. Адреса повернення займатиме 4 байти.

Можливі два варіанти запису параметрів в стек:

- в стек записується адреса параметра (зміщення в сегменті);
- в стек записується значення параметра.

Перший варіант (адреса параметра) використовується, якщо параметр передається за посиланням.

Якщо ж параметр передається за значенням і займає в пам'яті один або два байти, то в стек записуються 2 байти. Якщо розмір параметру більший за два байти, в стек записується кількість байтів, що кратна 4.

Для доступу до параметрів процедур використовується базовий регістр стеку *EBP*, який настроюється на область стеку. Для цього попередньо поточний вміст EBP зберігають, а потім записують в нього вміст покажчика стеку – регістра ESP:

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

Визначити позиції аргументів у стеку нескладно, оскільки відомий порядок їх запису в стек. Звідси *для адресації першого аргументу* можна використати адресний вираз [EBP+8] (4 байти – адреса повернення і ще 4 байти – попередній вміст регістра EBP). Для адресації наступних аргументів (пробігаючи від першого до останнього) необхідно кожний раз збільшувати константу на 2 або 4 (у загальному випадку), в залежності від розмірності попереднього аргументу.

Наприклад, вищезазначена С++ функція pp має чотири аргументи. Аргументи arg2, arg3, arg4 передаються за значенням (по 2 байти кожен), arg1 – за адресою (4 байти). Тоді адреса у стеку для аргумента arg1 буде визначатися виразом [EBP+8], аргумента arg2 – [EBP+12], аргумента arg3 – [EBP+14], аргумента arg4 – [EBP+16].

При багаторазових звертаннях до аргументів в асемблерній програмі доцільно задати наступну послідовність константних адресних виразів за допомогою директиви *EQU*:

```
Arg1   EQU    [EBP+8]
Arg2   EQU    [EBP+12]
Arg3   EQU    [EBP+14]
Arg4   EQU    [EBP+16]
```

або

```
Arg1   EQU    dword ptr [EBP+8]
Arg2   EQU    byte ptr [EBP+12]
Arg3   EQU    word ptr [EBP+14]
Arg4   EQU    word ptr [EBP+16]
```

Існує ще одна можливість роботи з аргументами процедури завдяки використанню директиви *model*. Вона дозволяє описати аргументи безпосередньо в директиві *proc*: після ключового слова *proc* додаються параметри з асемблерним типом. Наприклад, для процедури pp:

```
pp   proc @Arg1: dword, Arg2:byte, Arg3: word Arg2: word
       …
pp   endp
```

А для представленої нижче процедури **BigShowN**:

```
BigShowN proc @mas:dword, @len:word
       …
ret
BigShowN endp
```

Тоді параметри @mas та @len не потрібно вираховувати, як рознайменовані адреси у стеку (хоча вони так само будуть зберігатися в стеку за адресами [EBP+8], [EBP+12]), і з програми відповідно можна виключити рядки:

```
@mas       equ      [ebp+8]
@len       equ      [ebp+12]
```

Вивільняти параметри із стеку наприкінці процедури за допомогою *ret const* (де *const* – загальний розмір цих параметрів у байтах) також непотрібно.

Середовище Visual Studio дозволяє передавати параметри без явного використання стеку в асемблерній процедурі (див. процедуру **BigShowN**). Крім

того, при такому описі взагалі не потрібні ніякі дії з регістрами EBP, ESP. Команди на початку

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

та наприкінці процедури

```
pop         ebp
```

виконуватимуться автоматично.

5) Доступ до змінних С++ програми

Глобальні змінні програми мовою С++ можна використовувати в асемблерній програмі. Для цього в асемблерній програмі необхідно визначити ідентифікатори цих змінних у директиві *Extrn C* з наступним форматом:

Extrn C ім’я:тип, ..., ім’я:тип

Звичайно тип – це асемблерний тип (byte, word, dword), тому необхідно знати кількість байтів, що займають змінні С++.

6) Локальні параметри

У процедурах мовою Асемблера можна використовувати локальні параметри, що існують лише під час виконання процедури. Вони описуються за допомогою ключового слова *local* з асемблерними типами (byte, word, dword):

local ідентифікатор:тип, …, ідентифікатор:тип

Простір для локальних параметрів також виділяється в стеку шляхом зменшення вмісту *ESP* на загальний розмір цих параметрів. Таким чином, враховуючи встановлений перед цим вміст *EBP*, локальні параметри зберігатимуться за адресами вигляду *[EBP-зміщення]* (рис.4-2.2), причому загальна кількість зайнятих кожним з них байтів округлюється до більшого кратного 4 значення. Наприклад:

local TmpAddr:dword, number:word, symbol:byte

Адреси цих параметрів у стеку становлять відповідно:

TmpAddr – [EBP-4]
number – [EBP-6]
symbol – [EBP-8]

Ці локальні параметри займатимуть 8 байт у стеку (7 округлюється до 8).

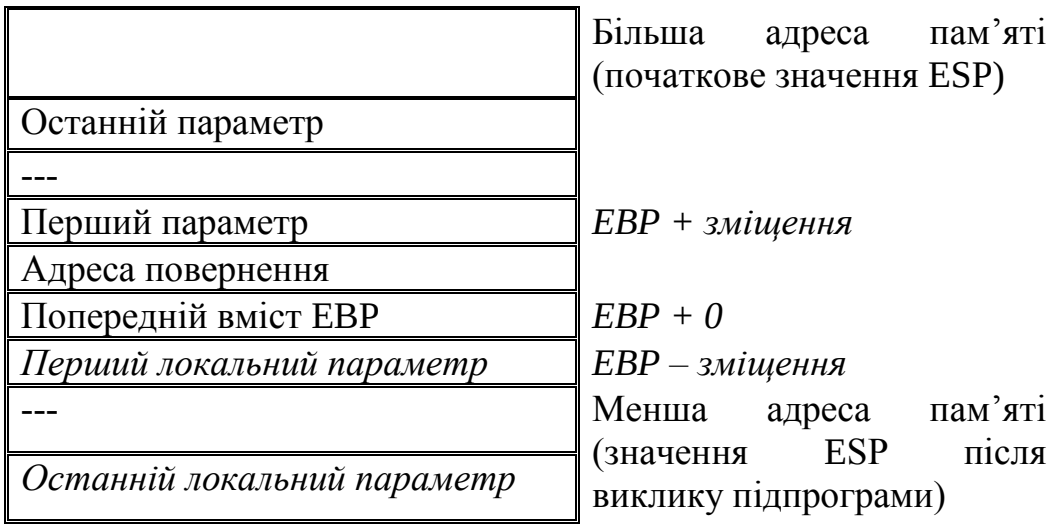

Рис.4-2.2. Стан стеку при наявності локальних параметрів

Потрібно пам’ятати, що значення локальних параметрів в стеку поки що не визначені, тому підпрограма має їх ініціалізувати. Перед виходом з підпрограми стек потрібно вивільнити від них, збільшивши відповідним чином покажчик стеку ESP.

7) Повернення значень з асемблерної процедури

Отримане значення функції перед поверненням із асемблерної процедури необхідно розмістити:

- в регістрі *AL*, якщо на тип даних у C++ відводиться один байт;
- в регістрі *AX*, якщо на тип даних у C++ відводиться одне слово;
- в регістрі *EAX*, якщо на тип даних у C++ відводиться одне подвійне слово.

Результат булевої функції зберігається у регістрі AL.

8) Організація повернення з підпрограми

Збережений на початку асемблерної процедури вміст регістра EBP має відновлюватися по закінченню її роботи. У відповідності до конвенції C, обов’язки щодо забезпечення початкового значення ESP покладаються на C/C++ програму. Тому підпрограма повинна закінчуватись командами:

```
pop  ebp
ret
```

9) Поле операндів директиви END

Програма мовою Асемблера не повинна бути основною, а це означає, що поле операндів директиви END повинно бути порожнім.

10) Виклик C++ функції з асемблерної програми

Для виклику в асемблерній програмі функцій C++ програми потрібно ідентифікатори цих функцій описати в асемблерній програмі за допомогою директиви *PROTO* (та вказівкою асемблерних типів) наступним чином:

Ім’я PROTO параметр:тип, …, параметр:тип

Слід зауважити, що в C++ програмі прототипи таких функцій повинні бути описані у тілі *extern "C"*.

Для виклику функції також може використовуватися макровизначення *invoke*. Воно полегшує виклик функції й автоматично передає параметри та очищує стек після закінчення роботи функції.

Наприклад, викликаємо функцію, котрій в якості параметрів передаємо вміст регістрів *EAX* та *DX*. Тобто функція має 2 параметри розміром відповідно 4 та 2 байти:

```
oddfunc PROTO first:WORD, second:DWORD
```

Виклик функції

```
invoke oddfunc, dx, eax
```

відповідає наступним командам:

```
push eax
push dx
call oddfunc
add esp,6                ; dx та eax займали 6 байтів
```

Налаштування середовища Visual Studio для роботи з асемблерним модулем

1) Створення проекту

У Visual Studio звичайно програма створюється у вигляді проекту, що складається з декількох програмних модулів. Що стосується програм, які написані різними мовами, то їх зв’язок забезпечує вбудований компонувальник

середовища. Програми, а точніше їх об'єктні файли, зв'язуються та об'єднуються в єдину програму.

Для створення нового проекта у Visual Studio, як вже відомо з виконання Лабораторної роботи №2-2, необхідно у головному вікні перейти на меню **File-> New ->Project**, далі для типу проекту вибрати **Visual C++** і **Win32 Console Application** (проекту мовою Асемблера немає), а в полі **Name** ввести назву проекту. Після натискання **ОК** з'являється нове вікно, в якому треба перейти на вкладку **Application Settings** і у вікні налаштувань відмітити поле **Empty project**, після чого натиснути кнопку **Finish.** Новий проект буде створений.

(Для розробки і компіляції C++ програми доцільно використовувати середовище Microsoft Visual C++, оскільки в ньому компіляція C++ програм відбувається за допомогою переведення програми мовою Асемблера з синтаксисом MASM, після чого подальша обробка здійснюється засобами трансляції та компонування з пакету Microsoft Assembler відповідної версії (для Visual Studio 2005 це MASM v8.0).

Звичайно Visual Studio не розпізнає файли мовою Асемблера. Для підтримки мови Асемблера треба включити у проекті умови побудови для файлів *.asm. Для цього обираємо пункт в меню **Custom Build Rules....** У новому вікні слід увімкнути готове правило для *.asm файлів, поставивши галочку навпроти правила «Microsoft Macro Assembler».

Далі необхідно додати файл *.cpp до проекту. Для цього потрібно перейти в **Solution Explorer** і зробити правий клік на вкладці **Source Files**. У випадаючому меню вибрати **Add->New Item…**, далі – **C++ File (.cpp)** і в полі **Name** ввести назву файлу, наприклад, **test2012.cpp**. Натиснувши кнопку **Add**, додамо цей файл до проекту. Далі в нього можна скопіювати код C++ програми.

*2) Створення *.asm файлу*

Асемблерний файл можливо скомпілювати окремо, а потім додати у C/C++ проект лише його файл. Для цього окремо скомпільований об'єктний файл асемблерної програми потрібно записати в папку проекту

ProjectName\ProjectName\Debug. Потім додати цей файл до проекту за допомогою меню проекту **Add > Existing Item…**

Проте, можливо і доцільно внести до проекту, крім **C++** файлів, саме початковий асемблерний файл (з розширенням .asm**).**

Створити такий файл можна наступними способами:

1. Відкрити меню **Add > New Item…** У вікні, що відкриється, написати назву файлу з розширенням **.asm**. Далі в цей файл внести чи скопіювати текст асемблерної підпрограми.
2. Створити окремо файл з розширенням **.asm** за допомогою, наприклад, додатку **Блокнот**. Відкрити меню папки проекту **Add > Existing Item…** У вікні, що відкриється, знайти та обрати створений **.asm** файл. Якщо правило компіляції *.asm файлів досі не було обрано, вікно **Custom Build Rules...** відкриється саме.

3) Налагодження програми

Середовище Visual Studio надає зручні засоби для налагодження програми як мовою C++, так і мовою Асемблера. По-перше, можна поставити *breakpoint* у будь-якій частині програми та відслідковувати покроково за виконанням програми. По-друге, можна легко подивитись вміст регістру, навівши курсор мишки на його ідентифікатор у програмі. І, по-третє, існують спеціальні вікна стану регістрів та пам’яті під час налаштування. Їх можна увімкнути у меню

Debug > Windows > Registers (Alt+5)

та

Debug > Windows > Memory > Memory 1 (Alt+6)

під час виконання програми. Значення в цих вікнах подаються у 16-ковому форматі.

4-2.3. Приклад організації взаємодії програми мовою C++ і програми на Асемблері

Як приклад організації взаємодії програм мовою C++ і мовою Асемблера пропонується програма **lab4.cpp** і відповідно програма **BigShowN.asm**. Програма lab4.cpp містить визначення мовою C++ двох байтових масивів x, y для представлення двох цілих беззнакових чисел великої розрядності та їх початкове заповнення, а також виклики процедури BigShowN для відображення на екрані значень цих чисел у 16-ковому форматі. lab4.cpp містить усі необхідні елементи для забезпечення зв'язку з асемблерною процедурою BigShowN.

```
// Program lab4.cpp
#include <stdio.h>
#define n 255                    // кількість байтів у надвеликому числі
typedef unsigned char byte;   // для роботи з байтами використовується тип char
extern "C" void BigShowN(byte* p1, int p2); //функція реалізована мовою Асемблера
int main()
{
        byte x[n], y[n];          //надвеликі числа
        for (int i=0; i<n; i++)
        {
                x[i]=i;
                y[i]=0;
        }
        printf("x=");
        BigShowN(x, n);
        printf("y=");
        BigShowN(y, n);
        return 0;
}
```

Представлення цілих беззнакового типу великої розрядності за допомогою байтових масивів зумовлено наступним. Звичайно такі цілі займають *k* комірок в оперативному запам'ятовуючому пристрої, де *k* – довільне значення. Нехай A – адреса даних такого типу. Тоді адреси комірок пам'яті та нумерацію двійкових розрядів надвеликого числа можна подати наступним чином:

A+k-1	...	A+i-1	...	A+1	A
b_{k*8-1} $b_{(k-1)*8}$	...	b_{i*8-1} $b_{(i-1)*8}$	...	b_{15} b_8	b_7 b_0

Значення B такого числа визначається стандартним чином:

$$B = \sum_{J=0}^{k*8-1} b_i * 2^j$$

Мова C++ (як і Паскаль) не підтримує такий тип даних. Для їх подання мовою C++ доцільно використовувати байтові масиви, причому один байтовий масив – для вмісту ОДНОГО надвеликого цілого беззнакового числа. Перший елемент масиву представляє молодший розряд числа, останній – відповідно старший розряд.

Представлена нижче процедура мовою Асемблера BigShowN призначена для виведення на екран байтів масиву у шістнадцятковому форматі і тим самим перевірки правильності виконання завдань лабораторної роботи. Процедура має два параметри: перший із них – адреса байтового масиву, другий параметр передається за значенням і задає кількість байтів масиву. При відображенні байти групуються у подвійні слова. Байт з найменшою адресою (задається першим параметром) завжди виводиться у найправішій позиції останнього рядка, що зручно для зорового порівняння двох масивів.

Безпосередньо для виведення на екран у BigShowN використовується функція C *printf*. Їй передається зміщення рядка, котрий треба вивести, та список параметрів для виведення, якщо вони потрібні.

```
.686                    ; можна використовувати .386
.model flat,C           ; модель пам`яті та передача параметрів за правилами C

public  BigShowN        ; глобальна видимість процедури, не обов`язково

.const                  ; опис констант

NewLine db 10,13,0  ;10 – перехід на новий рядок, 13 - перехід на початок рядка,
                        ; 0 – термінальній нуль
Space db 32,0           ; 32 – пробіл, 0 – термінальній нуль
Symbol db '%c',0        ; рядок для друку символа, заданого параметром (dl)

.code                   ; розділ коду програми

printf PROTO arg1:Ptr Byte, printlist: VARARG    ; прототип функції виведення
; Увага! printf змінює значення регістрів edx, ecx та eax
```

```
;****************************************
; п/п виведення на екран в hex-форматі
; даних із регістра esi:
; якщо di=28, то виводяться всі 4 байти
; якщо di=20, то виводяться 3 молодші байти
; якщо di=12, то виводяться 2 молодші байти
; якщо di=4, то виводиться один молодший байт

show_bt     proc
            pushad
            mov   bx,di
bt0:
            mov   edx,esi
            mov   cl,bl
            shr   edx,cl
            and   dl,00001111b
            cmp   dl,10
            jl    bt1
            add   dl,7
bt1:
            add   dl,30h
            invoke printf, offset Symbol, dl      ; написати цифру у 16-ковому форматі
            sub   bl,4
            jnc   bt0
            invoke printf, offset Space           ; записати один пробіл
            popad
            ret
show_bt     endp

; void BigShowN(byte* p1, int p2)

BigShowN    proc

; mas -  адреса байтового масиву
@mas        equ   [ebp+8]       ; місцезнаходження адреси масиву
; len - кількість байтів масиву, які необхідно вивести на екран
@len        equ   [ebp+12]      ; місцезнаходження кількості

            push  ebp
            mov   ebp,esp       ; базова адреса фактичних параметрів
; перехід на новий рядок без збереження ecx
            invoke printf, offset NewLine         ; перехід на новий рядок
;---------------------------------------------
; обчислення кількості пробілів у першому рядку
;---------------------------------------------
            mov   ax,@len
            test  ax,00000011b
            pushf
            shr   ax,2
            popf
            jz    @1
```

```
                inc     ax
@1:
                xor     bx,bx
                mov     di,28
                and     ax,00000111b
                jz      @2

; формування пробілів по відсутніх подвійних словах
                mov     ah,8
                sub     ah,al
                mov     al,ah
                xor     ah,ah
                imul    ax,9            ;8+1
                mov     bx,ax
@2:
                mov     dx,@len
                and     dx,00000011b
                jz      l000

; формування початкового значення кількості зсувів
                mov     di,dx           ;di -  1  2   3
                dec     di              ;di -  0  1   2
                shl     di,3            ;di -  0  8  16
                add     di,4            ;di -  4  12 20

; формування пробілів по відсутніх байтах у подвійному слові
                mov     dh,4
                xchg    dh,dl           ;dh -  1  2  3
                sub     dl,dh           ;dl -  3  2  1
                shl     dl,1            ;dl -  6  4  2
                xor     dh,dh           ;dx -  6  4  2
                add     bx,dx
l000:
                jcxz    l002

        ; виведення початковіх пробілів у першому рядку (ecx не зберігається)
l001:
                invoke printf, offset Space     ; вивести один пробіл
                dec     bx
                cmp bx,0
                jne     l001

l002:
                xor             ecx,ecx
                mov     cx,@len
                shr     cx,2
                cmp     di,28
                jz      @3

                inc      cx
@3:
                mov     ebx,@mas                ; записуємо в ebx адресу масиву
```

```
                lea     ebx,[ebx+ecx*4]-4   ; записуємо в ebx адресу останнього
                                            ; (найстаршого) елемента масиву

; виведення масиву (з найстаршого елементу до наймолодшого)
l004:
                mov     esi, dword ptr [ebx]    ; зберігаємо 4 байти масиву для виведення
                sub     ebx,4                   ; переміщуємо ebx на молодші 4 байти
                call    show_bt                 ; виклик функції виведення
                mov     di,28
                dec     cx
                test    cx,7                    ; 7 = 0111b
                jne     l005

; перехід на новий рядок зі збереженням ecx
                push    ecx                     ; printf змінює ecx, тому треба його зберегти
                invoke printf, offset NewLine       ; перехід на новий рядок
                pop     ecx
l005:
                jcxz    l006
                jmp     l004
l006:
                invoke printf, offset NewLine       ; перехід на новий рядок
                pop     ebp
                ret
BigShowN        endp
end
```

Програма lab4.cpp демонструє незалежність процедури BigShowN від типу даних мови C++. Це означає, що процедуру BigShowN можна використовувати також і для аналізу машинного формату типів даних мови C++. Наприклад, за допомогою процедури BigShowN легко визначається формат логічних значень *True* та *False*.

4-2.4. Завдання на виконання роботи

Перше заняття

1) Розглянути приклад організації взаємодії програм мовою C++ і Асемблера, представлений у п. 4-2.3.
2) Скопіювати програми **lab4.cpp** та **BigShowN.asm** в окремі файли робочого каталогу.

3) Використовуючи методичні вказівки п.4-2.2, створити у середовищі Visual Studio проект на основі **lab4.cpp** і **BigShowN.asm** (BigShowN.obj). Відкомпілювати програму та перевірити її працездатність.
4) Вивчити правила взаємозв'язку програм мовою C++ та мовою Асемблера (див. п.4-2.2).
5) Розробити алгоритм реалізації операції з надвеликими цілими додатними числами згідно варіанта завдання (табл. 4-1.1).

Друге заняття

У середовищі Visual Studio створити новий проект, який складається з програми мовою C++ і власної програми мовою Асемблера для виконання елементарної операції над даними цілого беззнакового типу великої розрядності згідно варіанта завдання та наступних вимог:

1) Програма мовою C++ повинна:

- відповідати вимогам зв'язку з асемблерними процедурами;
- містити визначення байтових масивів та їх початкове заповнення;
- містити виклики процедури BigShowN для відображення початкових даних,
- містити виклик розробленої асемблерної процедури з відповідними параметрами;
- містити виклики процедури BigShowN для відображення результатів;
- забезпечити виведення на екран текстових повідомлень (коментарів) перед викликом процедури BigShowN.

2) Програма мовою Асемблера повинна:

- розміщуватися або в початковому модулі разом з процедурою BigShowN, або в окремому файлі зі структурою, аналогічною файлу BigShowN.asm;
- виконувати ту чи іншу елементарну операцію (згідно варіанта) з надвеликими цілими додатними числами, які розміщуються у байтових масивах.

- відповідати спеціальним вимогам, щоб уможливити її виклик з програми мовою C++ (див. п.п.4-2.2);
- на початку модуля мовою Асемблера мати директиву Title із зазначенням групи та прізвища студента;

3)*Протестувати створений проект* на різних наборах значень байтових масивів, використавши для відображення і перевірки коректності роботи програми процедуру BigShowN.asm.

Таблиця 4-2.1

Варіанти завдання

1.	Розробити функцію **bool FBig2Add(byte* M1, byte* M2, short len)**, де M1,M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M1+M2. Функції FBig2Add присвоюється значення False в разі переповненя і True при його відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використовувати команди для 8-розрядних даних.
2.	Розробити функцію **void Extract(byte* M1, byte* M2, short len, short ibeg, short iend)**, де M1, M2 –надвеликі цілі додатні числа (байтові масиви довжиною len), ibeg, iend – номера двійкових розрядів, такі, що len*8-1 ≥ iend ≥ ibeg. Операція – виділити із числа M1 розряди з ibeg по iend включно та одержане таким чином число присвоїти M2. В старші розряди числа M2 занести 0.
3.	Розробити функцію **bool FBig3Add(byte* M1, byte* M2, byte* M3, short len)**, де M1, M2, M3 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M2+M3. Функції FBig3Add присвоюється значення False в разі переповненя і True при його відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використовувати команди для 8-розрядних даних.
4.	Розробити функцію **void Big2Sub(byte* M1,byte* M2,byte* Carry, short len),** де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M1-M2. Змінній байтового типу Carry присвоюється значення 1 при наявності позики і 0 при її відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використовувати команди для 8-розрядних даних.

Продовження табл. 4-2.1

5. Розробити функцію **void Big3sSub(byte* M1, byte* M2, byte* M3, byte* Carry, short len)**, де M1, M2, M3 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M2-M3. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використовувати команди для 8-розрядних даних.
6. Розробити функцію **bool FBig3Sub(byte* M1, byte* M2, byte* M3, short len)**, де M1, M2, M3 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M2-M3. Функції Fbig3Sub присвоюється значення False в разі наявності позики і True при її відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використовувати команди для 8-розрядних даних.
7. Розробити функцію **bool Biggreq(byte* M1, byte* M2, short len)**, де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – якщо M1 ≥ M2, то значення Biggreq – True, інакше – False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використовувати команди для 8-розрядних даних.
8. Розробити функцію **bool Bigne(byte* M1, byte* M2, short len)**, де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція - якщо M1 ≠ M2, то значення Bigne – True, інакше – False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використовувати команди для 8-розрядних даних.
9. Розробити функцію **bool Bigleseq(byte* M1, byte* M2, short len)**, де M1,M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – якщо M1 ≤ M2, то значення Bigleseq – True, інакше – False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використовувати команди для 8-розрядних даних.
10. Розробити функцію **void BigShrCount(byte* M1, short len, short count)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), count – кількість розрядів зсуву. Операція – лінійний зсув вправо (в сторону молодших розрядів) на кількість двійкових розрядів, яка задана параметром count. При цьому count молодших розрядів втрачаються, а в count старших розрядів заноситься 0. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для останніх байт використовувати команди для 8-розрядних даних.

Продовження табл. 4-2.1

11. Розробити функцію **void BigRorCount(byte* M1, short len, short count)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), count – кількість розрядів зсуву. Операція – циклічний зсув вправо (в сторону молодших розрядів) на кількість двійкових розрядів, яка задана параметром count. При цьому count молодших розрядів поступають на місце старших розрядів. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для останніх байт використовувати команди для 8-розрядних даних.
12. Розробити функцію **void BigZeroShr(byte* M1, short* cnt, short len)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), cnt – кількість розрядів зсуву. Операція – лінійний зсув вправо (в сторону молодших розрядів) до тих пір, поки в молодшому розряді числа не з'явиться одиничка. Кількість зсувів записується в параметр cnt. Якщо в початковому значенню числа M1 молодший розряд дорівнює 1, то зсуви не виконуються, а в параметр cnt записується нуль.
13. Розробити функцію **void BigShr(byte* M1, byte* Carry, short len)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), Carry – адреса змінної типу byte. Операція – лінійний зсув вправо (в сторону молодших розрядів) на один розряд. При цьому в змінну Carry заноситься значення молодшого розряду числа M1, а в старший розряд числа M1 заноситься 0.
14. Розробити функцію **bool FBigShl(byte* M1, short len)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len). Операція – лінійний зсув вліво (в сторону старших розрядів) на один розряд. При цьому функція FBigShl приймає значення False, якщо len*8-1 розряд числа M1 до зсуву дорівнює 1 і True в протилежному випадку. В молодший розряд числа M1 при зсуві заноситься 0.
15. Розробити функцію **void Big2sAdd(byte* M1, byte* M2, short len)**, де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M1+M2. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використовувати команди для 8-розрядних даних.
16. Розробити функцію **void Big3sAdd(byte* M1, byte* M2, byte* M3, short len)**, де M1, M2, M3 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M2+M3. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використовувати команди для 8-розрядних даних.

Продовження табл. 4-2.1

17. Розробити функцію **void Big3Add(byte* M1, byte* M2, byte* M3, byte* Carry, short len)**, де M1, M2, M3 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M2+M3. Змінній байтового типу Carry присвоюється значення 1 в разі переповненя і 0 при його відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використовувати команди для 8-розрядних даних.
18. Розробити функцію **void Big2sSub(byte* M1, byte* M2, short len)**, де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M1-M2. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використовувати команди для 8-розрядних даних.
19. Розробити функцію **bool FBig2Sub(byte* M1, byte* M2, short len)**, де M1,M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M1-M2. Функції Fbig2Sub присвоюється значення False в разі наявності позики і True при її відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використовувати команди для 8-розрядних даних.
20. Розробити функцію **void Big3Sub(byte* M1, byte* M2, byte* M3, byte* Carry, short len)**, де M1, M2, M3 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M2-M3. Змінній байтового типу Carry присвоюється значення 1 при наявності позики і 0 при її відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для віднімання останніх байт використовувати команди для 8-розрядних даних.
21. Розробити функцію **bool Biggr(byte* M1, byte* M2, short len)**, де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – якщо M1 > M2, то значення Biggr – True, інакше – False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використовувати команди для 8-розрядних даних.
22. Розробити функцію **bool Bigeq(byte* M1, byte* M2, short len)**, де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – якщо M1 = M2, то значення Bigeq – True, інакше – False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використовувати команди для 8-розрядних даних.

Продовження табл. 4-2.1

23. Розробити функцію **bool Bigles(byte* M1, byte* M2, short len)**, де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – якщо M1 < M2, то значення Bigles – True, інакше – False. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для порівняння останніх байт використовувати команди для 8-розрядних даних.
24. Розробити функцію **void BigShlCount(byte* M1, short len, short count)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), count – кількість розрядів зсуву. Операція – лінійний зсув вліво (в сторону старших розрядів) на кількість двійкових розрядів, яка задана параметром count. При цьому count старших розрядів втрачаються, а в count молодших розрядів заноситься 0. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для останніх байт використовувати команди для 8-розрядних даних.
25. Розробити функцію **void BigRolCount(byte* M1, short len, short count)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), count – кількість розрядів зсуву. Операція – циклічний зсув вліво (в сторону старших розрядів) на кількість двійкових розрядів, яка задана параметром count. При цьому count старших розрядів поступають на місце молодших розрядів. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то при необхідності для останніх байт використовувати команди для 8-розрядних даних.
26. Розробити функцію **void BigZeroShl(byte* M1, short* cnt, short len)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), cnt – кількість розрядів зсуву. Операція – лінійний зсув вліво (в сторону старших розрядів) до тих пір, поки у len*8-1 розряді не з'явиться одиничка. Кількість зсувів записується в параметр cnt. Якщо в початковому значенню числа M1 розряд len*8-1 дорівнює 1, то зсуви не виконуються, а в параметр cnt записується нуль.
27. Розробити функцію **void BigShl(byte* M1, byte* Carry, short len)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), Carry – адреса змінної типу byte. Операція – лінійний зсув вліво (в сторону старших розрядів) на один розряд. При цьому в змінну Carry заноситься значення len*8-1 розряду числа M1, а в молодший розряд числа M1 заноситься 0.
28. Розробити функцію **byte FcBigShl(byte* M1, short len)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len). Операція – лінійний зсув вліво (в сторону старших розрядів) на один розряд. При цьому функція FcBigShl приймає значення

Продовження табл. 4-2.1

len*8-1 розряду числа M1, а в молодший розряд числа M1 заноситься 0.
29. Розробити функції **void BigSetBit(byte* M1, short len, short number)** та **void BigClrBit(byte* M1, short len, short number)**, де M1 – надвелике ціле додатнє число (байтовий масив довжиною len), number – номер двійкового розряду числа M1, починаючи з 0. Операція – записати одиничку в розряд number для процедури BigSetBit і 0 для процедури BigClrBit.
30. Розробити функцію **void Big2Add(byte* M1, byte* M2, byte* Carry, short len)**, де M1, M2 – надвеликі цілі додатні числа (байтові масиви довжиною len). Операція – M1=M1+M2. Змінній байтового типу Carry присвоюється значення 1 в разі переповненя і 0 при його відсутності. Повинні використовуватись команди для 32-розрядних даних. Якщо значення len не кратно 4, то для додавання останніх байт використовувати команди для 8-розрядних даних.

Додаткові експерименти

1. Передати параметри у власну асемблерну функцію без явного використання стеку.
2. Викликати функцію через асемблерну вставку.
3. Викликати функцію мови C++ з окремого початкового файлу мовою Асемблера.
4. Провести дослідження передачі в асемблерну процедуру в якості параметрів динамічних масивів.

4-2.5. Контрольні запитання

1. Чому поле операндів директиви END в асемблерному модулі повинно бути порожнім?
2. За якою адресою оперативної пам’яті (більшою чи меншою) буде розташований перший фактичний параметр по відношенню до останнього для асемблерної процедури з викликом з C++ програми?

3. Чи можливий у програмі мовою Асемблера виклик C++ функцій?
4. До яких змінних C++ програми можливий доступ в програмі мовою Асемблера та як він забезпечується?
5. Яка програма (та, що викликає, чи та, яку викликають) відповідає в C++ за відновлення вмісту покажчика стека ESP?

Рекомендована література

1. Пирогов В. Assembler. Учебный курс. [Текст] / Пирогов В. — М.: Издатель Молгачаева С.В., Издательство Нолидж, 2001. — 848 с.
2. Юров В. Assembler. Учебник для ВУЗов, 2-е изд. [Текст] / Юров В.— СПб : Питер, 2003. — 637 с.
3. Юров В. Assembler: учебный курс. [Текст] / Юров В., Хорошенко С. — СПб : Питер Ком, 1999. — 672 с.

ДОДАТОК А

Довідник з макроасемблера MASM і редактора зв'язків LINK

А.1. Запуск макроасемблера

Запуск макроасемблера може виконуватися у двох режимах: з використанням підказок або за допомогою командного рядка.

Для запуску макроасемблера з використанням підказок необхідно ввести командний рядок, що містить тільки ім'я *MASM* (або стандартним запуском програм у додатку FAR чи в іншому командному процесі). MASM перейде в діалоговий режим і серією підказок запросить у користувача інформацію про наступні файли (відповідь полягає в наборі потрібних символів і натисканні клавіші ENTER):

- ім'я початкового файлу. Якщо у відповіді не зазначене розширення, передбачається розширення *.asm*;
- ім'я об'єктного файлу. Якщо у відповіді не зазначене розширення, передбачається *.obj*. Базове ім'я об'єктного файлу за замовчуванням збігається з базовим ім'ям початкового файлу.
- ім'я файлу лістингу. Якщо у відповіді не зазначене розширення, передбачається *.lst*. Базове ім'я файлу лістингу за замовчуванням – NUL (тобто файл лістингу не створюється);
- ім'я файлу перехресних посилань. Якщо у відповіді не зазначене розширення, передбачається *.crf*. Базове ім'я файлу лістингу за замовчуванням – NUL (тобто файл перехресних посилань не створюється).

Наприкінці будь-якої відповіді після символу / можуть бути задані опції макроасемблера, що описані нижче. Якщо в якійсь відповіді вказуватиметься специфікований символ *;* (крапка з комою), MASM вийде з діалогового режиму й установить решту імен за замовчуванням із наступного списку:

<ім’я початкового файлу>.obj

NUL.LST

NUL.CRF

У будь-якій відповіді також можуть бути задані відповіді на декілька наступних підказок. У цьому випадку, відповіді відокремлюється одна від іншої комою.

Для запуску MASM за допомогою командного рядка необхідно ввести командний рядок наступного вигляду:

MASM <ім’я початкового файлу>[,[< ім’я об'єктного файлу>] [,[<ім’я файлу лістингу>][,[< ім’я файлу посилань>]]]] [<опції>][;]

Символ *;* може бути специфікований у будь-якому місці командного рядка до визначення усіх файлів. У цьому випадку, імена решти невизначених файлів приймаються за замовчуванням із наведеного вище списку. З цього ж списку приймаються за замовчуванням імена файлів, специфікація яких у командному рядку опущена. Якщо в командному рядку виявлена помилка, про це повідомляється через консоль, і MASM переходить у діалоговий режим. Опції MASM можуть розташовуватися у будь-якому місці командного рядка.

Ім’я кожного файлу може супроводжуватися інформацією про шлях до каталогу, що містить цей файл, інакше пошук початкового файлу; створення результуючих файлів буде здійснюватися в поточному каталозі.

Робота MASM може бути в будь-який момент припинена натисканням клавіш *CONTROL-C.*

Приклади командних рядків запуску MASM:

masm lab1.asm, lab1.obj, lab1.lst;

masm lab1.asm, lab1.obj, lab1.lst

masm lab1;

masm lab1, , lab1;

А.2. Опції MASM

Опції MASM дозволяють керувати роботою макроасемблера незалежно від початкової програми. Кожна опція позначається попереднім символом / і може кодуватися як малими, так і великими буквами. Опції можуть розташовуватися у будь-якому місці командного рядка чи відповіді на підказку. Нижче подається список найуживаніших опцій MASM з описом виконуваних ними функцій.

/HELP – Вивести на екран повний перелік опцій MASM. Транслятор MASM постійно удосконалюється, існує значна кількість його версій з можливими змінами у переліку опцій. Опція */HELP* дозволяє уточнювати перелік опцій для конкретної версії MASM.

/A – Сегменти в об'єктному файлі розташовуються за абеткою. При відсутності опції розташування сегментів відповідає порядку у початковому файлі.

/D – Діагностичні повідомлення після 1-го перегляду помістити в лістинг програми. Багато помилок 1-го перегляду виправляються на 2-ому перегляді, та, якщо не заданий */D*, у лістинг вони не потрапляють. Вказівка цієї опції дає більш глибоку діагностику початкового тексту. Помилки як 1-го, так і 2-го проходів видаються на консоль, навіть коли файл лістингу не створюється. Порівняння лістингу 1-го та 2-го переглядів дозволяє ефективніше локалізувати помилку *Phase error*.

/D<ім'я> = [value] – Визначити ім'я. Зазначене ім'я приймає абсолютне значення *value* і вводиться в початкову програму аналогічно використанню директиви *EQU*. Ця опція дозволяє створювати різні версії програм з одного й того ж початкового файлу, наприклад, шляхом використання визначеного опцією імені у директивах умовного асемблювання.

/I<шлях> – Завдання шляху пошуку файлів, що підключаються у початковий директивою *INCLUDE* без явної вказівки шляху. Вказівка шляху в *INCLUDE* більш пріоритетна, ніж в опції */I*.

/ML – Установити відмінність малих та великих букв в ідентифікаторах. При відсутності цієї опції, малі літери автоматично перетворяться на великі. Опція може знадобитися для сумісності з програмами, які чуттєві до регістрів мови.

/MX – Установити різницю між великими та малими буквами в загальних і зовнішніх іменах. Опція подібна */ML*, але її дія поширюється лише на імена, які задані у директивах *PUBLIC* або *EXTRN*.

/N – Заборонити формування у файлі лістингу таблиць макроструктур, записів, сегментів і імен.

/R – Генерація коду для математичного співпроцесора. Застосовується при відсутності в початковому файлі директив завдання допустимих команд для процесорів, починаючи з i80386.

/V – Включити в діагностику, що виводиться на екран, інформацію про кількість оброблених рядків та ідентифікаторів користувача. При відсутності цієї опції на екран видається лише інформація про кількість помилок і об’єм пам’яті.

/X – Виводити в лістинг тіла блоків *IF (IF, IFE, IF1, IF2, IFDEF, IFNDEF, IFB, IFNB, IFIDN і IFDIF)*, для яких умови асемблювання не виконуються, і коди команд за цією причиною не генеруються.

/Z – Виводити на екран рядки початкового файлу, що містять помилки. При відсутності цієї опції на консоль видаються тільки повідомлення про помилку і номер рядка.

А.3. Запуск редактора зв'язків LINK

Запуск LINK може здійснюватись одним із наступних способів:

- з використанням підказок;
- за допомогою командного рядка;
- з використанням файлу відповіді.

Для запуску LINK з використанням підказок необхідно ввести командний рядок, що містить тільки ім’я LINK. Редактор зв’язків LINK

перейде в діалоговий режим і серією підказок запросить у користувача інформацію про наступні файли (відповідь полягає в наборі необхідних символів і натисканні клавіші ENTER):

- ім’я об'єктного файлу. Якщо у відповіді не зазначене розширення, передбачається *.obj*. Якщо потрібно визначити кілька файлів, їхні імена розділяються символом + . Якщо усі імена не містяться на одному рядку, введення імен можна продовжити, поставивши символ + в останню позицію поточного рядка. У цьому випадку LINK повторить запит для введення додаткових імен.
- ім’я виконавчого файлу. Якщо у відповіді не зазначене розширення, передбачається *.exe*. Базове ім’я виконавчого файлу за замовчуванням збігається з базовим і ім’ям об’єктного файлу.
- ім’я файлу карти пам’яті. Карта пам’яті містить перелік створених фізичних сегментів та інші дані. Якщо при відповіді не зазначене розширення, передбачається MAP. Базове ім’я за замовчуванням – NUL (файл не створюється).
- ім’я бібліотеки. Якщо у відповіді не зазначене розширення, передбачається LIB. Можна задавати кілька імен бібліотек за аналогією з OBJ-файлами. Якщо відразу натиснути ENTER без введення імені, бібліотеки використовуватися не будуть.

У кожній відповіді після символу / можуть бути задані опції LINK, що описані нижче. Якщо в якійсь відповіді є специфікований символ ; (крапка з комою), LINK вийде з діалогового режиму й установить решту імен за замовчуванням із наступного списку:

< ім’я об'єктного файлу>.EXE

NUL.MAP

Бібліотеки не використовуються.

У будь-якій відповіді також можуть бути задані відповіді на декілька наступних підказок. У цьому випадку, відповідь відокремлюється одна від іншої комою.

Для запуску LINK за допомогою командного рядка, необхідно ввести командний рядок наступного вигляду:

LINK < ім'я об'єктного файлу>[,[< ім'я виконавчого файлу>][,[< ім'я файлу карти пам'яті >] [,[<ім'я бібліотеки>]]]][<опції>][;]

Символ *;* (крапка з комою) може бути специфікований у будь-якому місці командного рядка до визначення усіх файлів. Тоді імена решти невизначених файлів приймаються за замовчуванням із приведеного вище списку. З цього ж списку приймаються за замовчуванням імена файлів, специфікація яких у командному рядку опущена (за допомогою зайвої коми). Про виявлення помилки у командному рядку повідомляється через консоль, і LINK переходить у діалоговий режим.

Ім'я кожного файлу може супроводжуватися інформацією про каталог (шлях), що містить цей файл, інакше пошук вхідних файлів або створення результуючих буде здійснюватися в поточному каталозі.

Робота LINK може бути в будь-який момент припинена натисканням клавіш *CONTROL-C.*

А.4. Опції LINK

Всі опції LINK позначаються попереднім символом / і можуть бути скорочені довільним чином, але так, щоб код залишався унікальним серед опцій. Нижче приведені описи всіх опцій LINK (у дужках зазначені мінімальні скорочення):

/HELP (HE) – Видати список діючих опцій. Цю опцію не можна використовувати разом з ім'ям файлу.

/PAUSE (P) – Пауза перед записом модуля в EXE-файл (і після запису MAP-файл, якщо це передбачено). Під час цієї паузи можна при необхідності переставити дискети.

/EXEPACK (E) – Установити компактний запис послідовностей однакових бітів. Такий EXE-файл має менший обсяг і швидше завантажується в пам’ять, але його не можна налагоджувати за допомогою налагоджувачів. Опція дає ефект, якщо програма містить довгі потоки ідентичних бітів.

/MAP (M) – Формувати MAP-файл. Файл формується навіть без специфікації при запуску LINK і має в цьому випадку ім’я за замовчуванням.

/NOIGNORECASE (NOI) – Установити відмінність малих та великих букв. Відмінність може бути встановлена також опціями */ML* і */MX* MASM.

/STACK:<число> (ST) – Установити розмір стеку (в байтах). Інформація про розмір стеку, що міститься в об'єктному модулі, ігнорується. Розмір стеку може бути заданий в межах від 1 до 65535.

/HIGH (H) – Встановити адресу початку програми на найвищу можливу адресу вільної пам'яті. Без цієї опції установка здійснюється на мінімально можливу адресу.

/DSALLOCATE (D) – Обробити групу з ім’ям DGROUP. Звичайно LINK привласнює молодшому байту групи зміщення 0000h. При завданні цієї опції старшому байту групи з ім’ям DGROUP привласнюється зміщення FFFFh. У результаті дані будуть розміщатися в областях програми з максимально великими адресами. Опція /D звичайно застосовується разом з опцією /H для більш ефективного використання незайнятої пам’яті до старту програми.

/SEGMENTS:<число> (SE) – Установити максимальне число сегментів, яке може обробити LINK. Число може бути задане в межах від 1 до 1024. При відсутності опції – 128. Пам’ять виділяється з урахуванням цього максимального числа сегментів.

/DOSSEG (DO) – Упорядкувати сегменти в EXE-файлі. При специфікації цієї опції сегменти розташовуються в наступній послідовності:

- сегменти з класом CODE;
- інші сегменти, що не входять у групу DGROUP;
- сегменти, що входять у групу DGROUP.

ДОДАТОК Б

Довідник з налагоджувача AFD

Б.1. Загальна характеристика налагоджувача

Повноекранний налагоджувач AFD забезпечує налагоджування програм лише для процесорів i8086 (i286) в операційній системі DOS. Під час роботи AFD на екрані відображується вміст усіх регістрів процесора, 4-х верхніх елементів стеку та до дев'яти рядків дизасембльованого коду програми. Окрім того, є два незалежних вікна пам'яті, які дозволяють відображати вміст комірок пам'яті у шістнадцятковому форматі та в коді ASCII. Дизасембльований код, а також вміст пам'яті у шістнадцятковому вигляді чи в коді ASCII можуть бути виведені на друк чи у файл.

Функції налагоджувача задаються командами в полі (вікні) командного рядка. Функції, що використовуються найчастіше, можна задати за допомогою функціональних клавіш. Команди мають довжину від одного до двох символів. Для перевірки їх синтаксису можливо, не перериваючи роботи, викликати виведення на екран довідкової інформації. Крок виконання програми може бути виконаний натисканням однієї клавіші. Навіть процедури, які викликаються за допомогою CALL чи INT можуть бути виконані натисканням однієї клавіші.

Будь-яка помилка введення команди викликає повідомлення про помилку, і курсор встановлюється на символ, де під час розбору команди була знайдена помилка.

Код завантаженої програми видається на екран у дизасембльованому вигляді, при цьому підтримується мнемоніка усіх команд лише мікропроцесорів I8086 та I80286.

Машинні інструкції в коді програми можуть бути легко змінені за допомогою вбудованого в AFD асемблера. Якщо прикладні програми використовують виведення даних на екран, тоді користувач може вибрати

режим альтернативного екрану для розділення даних, виведених AFD та прикладною програмою.

Б.2. Запуск AFD

Налогоджувач AFD може міститись в файлах *afd.exe*, *afdp.exe* або *afdpro.exe*. Запуск може виконуватись двома способами: інтерактивним або пакетним. У першому випадку, AFD запускається засобами операційної системи як і будь-яка інша програма без параметрів. *При цьому на екран виводиться лише рекламна картинка. Для продовження роботи необхідно натиснути будь-яку клавішу, після чого з'являється основний екран AFD.*

У другому випадку, необхідно ввести наступний командний рядок DOS:

{пристрій:} AFD {специфікація файлу}

У фігурних скобках вказані необов'язкові параметри командного рядка. При всі відсутності параметрів виконується інтерактивний запуск AFD.

Параметр *пристрій:* задає диск та шлях до каталогу, де знаходиться AFD.EXE, якщо активним (поточним) є інший каталог. Необов'язковий параметр *специфікація файлу* містить шлях та ім'я файлу програми, що буде налагоджуватись (вивчатись). Рекомендується розміщати AFD та програму в одному каталозі, що дає змогу не вказувати шлях.

При вказівці у команді запуску AFD імені файлу виконується завантаження саме цього файлу.

Після запуску AFD курсор встановлюється в полі командного рядка, відміченого знаком └─>. Далі ми будемо посилатися на цю частину екрану як на *командний рядок AFD*. Поле над командним рядком вказує вміст регістрів процесора та 4-х верхніх елементів стеку. Цю частину екрану далі будемо називати *областю регістрів*.

AFD програмним шляхом моделює два додаткові сегментні регістри: HS та FS, які не реалізовані в процесорах i8086 (i286) апаратно. Ці 2 регістри можуть слугувати як допоміжні. Вони використовуються також для ідентифікації сегменту після виконання команд пошуку чи порівняння. Регістр

HS використовується в командах пошуку та порівняння. FS використовується тільки в команді порівняння, якщо для адресації не можна використати вміст іншого сегментного регістру. Крім того, ці регістри можуть використовуватися при визначенні значення регістру сегмента для адресації вікна в пам'яті, яке не повинно змінюватись навіть при зміні вмісту регістрів сегментів програмою. Користувач може звертатися до них так, як і до інших регістрів.

Область нижче від командного рядка містить текст програми у дизасембльованому вигляді, починаючи з комірки пам'яті за адресою CS:IP. Крайнє ліве поле кожного рядка цього тексту містить зміщення в сегменті кодів, а наступні шістнадцяткові значення визначають код кожної команди процесора. Дизасембльований текст виглядає аналогічно тексту програми мовою Асемблера, за винятком того, що символічні імена адрес не використовуються.

Один із рядків дизасембльованної області показаний на екрані з інверсним фоном. Інструкція в цьому рядку є поточною і виконуватиметься далі за командою *G* чи *Крок* (функціональна клавіша *F1*) незалежно від вмісту IP. При виконанні команди *Крок* виконана інструкція зсувається на один рядок вгору. Рядок вище називається рядком попередньої команди. Цей верхній рядок дизасембльованої області (рядок попередньої команди) перекривається повідомленнями про стан системи та повідомленнями про помилки, якщо вони з'являються. Тому на цей рядок (в залежності від представленої там інформації) посилаються також, як на *рядок стану*. Дизасембльована область може бути зсунута вгору та вниз за допомогою клавіш *'курсор вгору'* та *'курсор вниз'*, клавіш *PgUp* та *PgDn*.

У випадку посилання поточної інструкції на комірку пам'яті, вміст цієї комірки показується в інверсному фоні наприкінці рядка, розташованого після командного рядка. В залежності від типу інструкції може бути показано значення слова чи байта.

Область праворуч від командного рядка – *вікно пам'яті номер 1*, а область під дизасембльованою областю – *вікно пам'яті номер 2*.

Кожне вікно у першій колонці має логічну адресу, а вміст пам’яті виводиться у шістнадцятковому форматі. В кожному рядку вікна 1 у шістнадцятковому форматі показано 8 байтів. Вікно 2 розділено на дві області, які використовують одну й ту ж саму адресу. Ліва частина відображає вміст пам’яті як і вікно 1, але по 16 байтів у рядку. Права частина показує вміст тієї ж області пам’яті, що й ліва, але в коді ASCII. Адреси у вікнах можуть відрізнятися одна від одної.

Усі символи, що виводяться з підвищеною яркістю, можуть бути змінені користувачем шляхом введення на їх місце нових значень. Курсор може пересуватися у кожному вікні. Для переходу з одного вікна в інше необхідно використовувати клавіші *F7-F10*.

Б.3. Опис основних процедур

До основних функцій AFD користувач може звернутися за допомогою функціональних клавіш та команд у командному рядку.

Нижче описуються найуживаніші команди AFD (у фігурних дужках вказуються необов’язкові параметри).

1) Завантаження програми для надагоджування

Формат команди:

L <file>

Команда використовується для завантаження у пам’ять файлів будь-якого типу. Якщо розширення імені файлу не вказано, тоді за замовчуванням використовується розширення .EXE.

2) Визначення початкової адреси дизасемблювання

Формат команди:

D адр

Наприклад:

D 120

D *

D FS:100

D 123:AX+SI

D IP

Команда D використовується для установки початкової адреси дизасембльованої області. Адреса *адр* може бути задана як сегмент та зміщення. Для завдання зміщення можна використовувати арифметичний вираз. При відсутності вказівки сегмента використовується вміст регістра сегмента коду, що відображується в даний момент. При вказівці сегменту, це значення буде використано для запису в регістр CS, вміст якого показаний в області регістрів.

3) Команда G (виконати)

Формат команди:

G {поч.адр.}, {адр.зуп.}

Наприклад:

G

G *

G CS:100

G 123, 1100

G , 345a:1200

Команда G (виконати) використовується для запуску програми, що налагоджується. Адресні параметри мають посилалися на комірки, що містять перші байти доступних машинних команд. В іншому випадку результат виконання команди G не передбачуваний. Початкова адреса 'поч.адр.' може бути вказана як зміщення, або як логічна адреса. При відсутності сегментної частини використовується значення сегментного регістра коду CS, що відображується в даний момент. Адреса зупинки {адр.зуп.} може бути як логічною адресою, так і зміщенням у поточному сегменті кодів. Її завдання в команді призводить до закінчення виконання програми за умови, що вміст регістрів CS:IP співпадатиме з заданою адресою.

4) Формування файлу дизасембльованого коду

Формат команди:

PD адреса, довжина {,фспец}

Наприклад:

PD 0, 100, c:\subject\lab2\test0060.prn

PD DS:1103, CX, A:TST.PRN

За допомогою цієї команди дизасембльований код може виводитись в заданий файл. При відсутності вказівки у параметрі *адреса* сегментної складової використовується вміст регістра CS. Параметр *адреса* задає початкову адресу пам'яті, а параметр *довжина* визначає кількість байтів програми, які необхідно дизасемблювати. Параметр *довжина* має задаватися у шістнадцятковому форматі.

5) Використання вбудованого асемблера

Формат команди:

A {адреса}

Приклад:

A

A 200

A CS:30

За допомогою цієї команди можна вводити (редагувати) програми у вікні дизасембльованого коду. *Редагування відбувається лише шляхом заміни, а не вставки.* При відсутності параметра {адреса} редагування виконується з поточної адреси. Асемблерний текст вводиться з прийнятими при дизасемблюванні особливостями по відношенню до стандартних синтаксичних конструкцій Асемблера. Завершення редагування чергової інструкції відбувається при натисканні клавіші "Enter". При цьому AFD переходить на наступний рядок. Для закінчення введення (редагування) необхідне одночасне натискання клавіш *Ctrl/Enter*.

6) Використання функціональних клавіш

F1 – виконання однієї команди програми;

F2 – виконання процедури;

F3 – вилучення останньої команди із стеку команд;

F4 – виведення довідкової інформації;

F5 – вхід до меню визначення місць зупинки;

F6 – перехід на альтернативний екран та зворотньо;

F7 – курсор до вікон вгору;

F8 – курсор до вікон вниз;

F9 – курсор до вікон вліво;

F10 – курсор до вікон вправо.

7) Завершення роботи та повернення в операційну систему

Формат команди:

QUIT