Лабораторна робота №1

Тема: **ОЗНАЙОМЛЕННЯ З TURBO ASSEMBLER**

Мета роботи: ознайомитися з програмування на Turbo Assembler, створивши першу програму.

1 Теоретичні відомості

Після створення процесора 8086 фірма Intel розробила більш досконалі процесори об'єднані під назвою I 80x86, така назва означає, що всі команди мікропроцесора, які виконуються на молодших моделях обов'язково виконуються на старших, отже всі ПЗ, які розроблені для процесора 8086 успішно будуть працювати і на останніх моделях 80486 і Pentium. Ми будемо розглядати процесори з точки зору програміста. Не дивлячись на різноманітність моделей процесорів, найбільш важливим з точки зору біології програмування є 8086 як базова модель і 80386 , як перший процесор фірми Intel, який в повному об'ємі реалізував принцип багатозадачності.

1.1 Програмування на мові Асемблера

Програмування на мові асемблера вважається складною задачею, причини цього такі:

1. Мова Асемблера будь-якого процесора суттєво складніша будь-якої мови високого рівня. Щоб скористатись всіма можливостями мови асемблера, треба по крайній мірі знати команди мікропроцесора, а їх число з усіма можливими варіантами переважає 100, їх кількість значно перевищує кількість операторів і ключових слів інших мов високого рівня. Проблема ускладнюється ще тим, що зміни в асемблері виникають набагато швидше ніж в мовах високого рівня, це зв'язано з появою нових мікропроцесорів і відповідно нових команд.
2. Програміст, який використовує мови Асемблера повинен сам слідкувати за розподілом пам'яті та вмістом регістрів, щоб коректно розподіляти і оперувати пам'яттю, в мовах високого рівня це робиться автоматично при допомозі компілятора, але ця обставина має перевагу: можна оптимально

розташувати дані в пам'яті, забезпечити максимальну швидкість виконання та мінімальну довжину програми.

3. Програми на мові асемблера важче проектувати та підлагоджувати, треба весь час пам'ятати, що конкретно знаходиться в кожному регістрів в даній комірці пам'яті. Прийнято вважати, що розробка програми тільки на мові Асемблера, деякого процесора, навіть якщо він поширений не рекомендується. Зрозуміло, що будь-яку програму можна написати тільки з допомогою асемблера, але для цього треба використати набагато більшу кількість команд і час який піде на її виконання і відладку буде набагато більший ніж для мови високого рівня. Набагато вигідніше писати програми на мові високого рівня, а найбільш критичні частини на швидкодії писати на мові асемблера. Наприклад на Асемблері можна скласти процедури для реалізації вводу-виводу низького рівня, процедури обробки переривань та деякі інші.

Етапи створення програми представлены на Рис1.1.

Розробка програми на мові Асемблера включає кілька етапів.

1) Підготовка початкового тексту програми;

2) Асемблювання програми (отримання об'єктного коду);

3) Компоновка програми (отримання виконуваного файлу);

4) Відладка програми (знаходження помилок).

У програмуванні першою програмою традиційно є програма, що виводить на екран повідомлення "Привіт!". Не буде виключенням і наша програма, оскільки це є хорошою відправною крапкою.

Увійдіть в текстовий редактор (один з тих редакторів, які формують файли в коді ASCII, наприклад блокнот) і введіть наступні рядки програми під назвою hello.asm, текст програми наведений на Рис.1.2.

Після того, як ви введете цю програму, збережете її на диску.

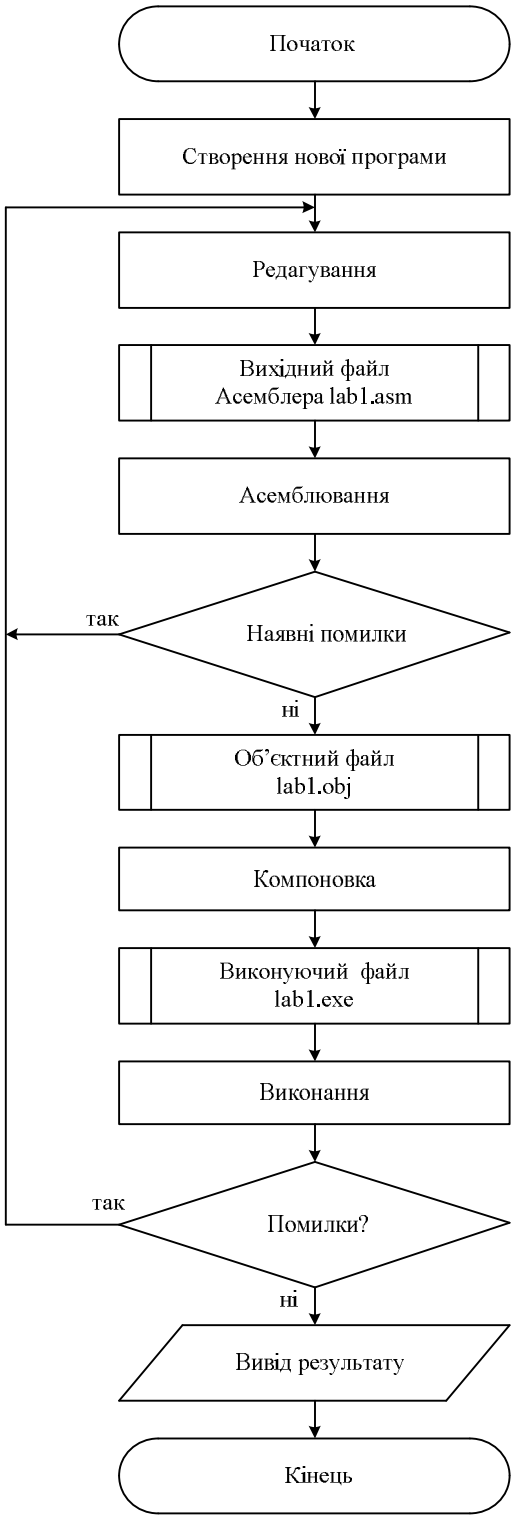

Рис.1.1 Редагування, асемблювання, компоновка і виконання

1.2 Асемблювання вашої першої програми

Після того, як ви зберегли файл HELLO.ASM, ви захочете запустити програму. Проте, перед тим, як ви зможете її запустити, потрібно буде перетворити програму у виконувчий вигляд. Як показано на Рис. 1.2, де

представлений повний цикл створення програми (редагування, асемблювання, компоновка і виконання), це потребує двох додаткових кроків – асемблювання ті компоновки.

На етапі асемблювання ваш вихідний код (текст програми) перетворюється на проміжну форму, яка називається *об'єктним модулем,* а на етапі компоновки один або декілька модулів комбінуються у виконувану програму.

Асемблювання і компоновку ви можете виконувати за допомогою командного рядка.

Для асемблювання файлу HELLO.ASM наберіть:

```
TASM hello
```

та натиснете клавішу Enter. Якщо ви не задали інше ім'я, файл HELLO.ASM буде асемблюватиме у файл HELLO.OBJ. (Відмітимо, що розширення імені файлу вводити не потрібно. Турбо Асемблер має на увазі в цьому випадку, що файл має розширення .ASM.) На екрані ви побачите наступне:

```
Turbo Assembler Version 3.0 Copyright (C) 1988,1991   (1)
                                       by Borland International Inc.
        Assembling file: HELLO.ASM      (2)
        Error messages: None            (3)
        Warning messages: None          (4)
        Passes: 1                       (5)
        Remaining memory: 266K          (6)
```

1 - Турбо Асемблер, версія 3.0; авторські права фірми Borland, 1991 г.; 2 - асемблює файл HELLO.ASM; 3 - повідомлення про помилки: немає; 4 - застережливі повідомлення: немає; 5 - число проходів: 1; 6 - залишається пам'яті: 266К.

Якщо ви введете файл HELLO.ASM у точності так, як показано, то ви не отримаєте жодних застережливих повідомлень або повідомлень про помилки. Якщо ви отримуєте такі повідомлення, вони з'являються на екрані поряд з номерами рядків, вказуючими рядки, де містяться помилки. При отриманні повідомлень про помилки перевірте вихідний код (текст)

програми і переконаєтеся, що він виглядає точно так, як вихідний код в нашому прикладі, а потім знову асемблюйте програму.

1.3 Компоновка програми

Після асемблювання файлу HELLO.ASM ви просунулися лише на один крок в процесі створення програми. Тепер, якщо ви скомпонуєте тільки що отриманий об'єктний код у виконуваний вигляд, ви зможете запустити програму.

Для компоновки програми використовується програма TLINK, що представляє собою поставлений разом з Турбо Асемблером компонувальник. Введіть командний рядок:

```
TLINK HELLO
```

Тут знову не потрібно вводити розширення імені файлу. Компонувальник TLINK за умовчанням передбачає, що цим розширенням є розширення .OBJ. Коли компоновка завершиться (саме більше через декілька секунд), компонувальник автоматично привласнить файлу з розширенням .EXE ім'я, співпадаюче з ім'ям вашого об'єктного файлу (якщо ви не визначили інше ім'я). При успішній компоновці на екрані з'являється повідомлення:

```
Turbo Linker Version 3.0 Copyright (c) 1988, 1991 by Borland
International Inc.
```

В процесі компоновки можуть виникнути помилки (у даній програмі це мало вірогідно). Якщо ви отримали повідомлення про помилки компоновки (вони виводяться на екран), змініте вихідний код програми так, щоб він в точності відповідав тексту програми в наведеному вище прикладі, а потім знову виконаєте асемблювання і компоновку.

1.4 Запуск першої програми

Тепер програму можна запустити на виконання. Для цього у відповідь на підказку операційної системи DOS введіть hello. На екран виведеться повідомлення:

Привіт!

1.5 Що відбувається?

Тепер, коли ви отримали і виконали програму HELLO.ASM, давайте повернемося назад і розглянемо детально, що відбувається з моменту введення тексту програми до її виконання.

Коли ви асемблюєте файл HELLO.ASM, Турбо Асемблер перетворює текст інструкцій в цьому файлі в їх двійковий еквівалент в об'єктному файлі HELLO.OBJ. Цей файл є проміжним файлом (проміжною ланкою в процесі переходу від текстового до виконуваного файлу). Файл HELLO.OBJ містить всю інформацію, необхідну для створення виконуваної коди з інструкцій, що містяться у файлі HELLO.ASM, але вона записана у вигляді, який дозволяє комбінувати її з іншими об'єктними файлами для створення однієї програми.

При компоновці файлу HELLO.OBJ TLINK перетворить його у виконуваний файл HELLO.EXE, який ви запускаєте, ввівши hello у відповідь на підказку DOS.

Тепер введіть:

*dir hello.**

При цьому буде виведений список файлів HELLO на диску. Це будуть файли HELLO.ASM, HELLO.OBJ, HELLO.EXE і HELLO.MAP.

1.6 Елементи та структура програми на мові Turbo Assembler

Тепер, коли ви краще розумієте, що являє собою програма на мові асемблера процесора 8086, ви готові до того, щоб почати писати на Асемблері програми. Давайте розглянемо, які мінімальні вимоги до працюючої програмі на Асемблері. Навіть проста програма на Асемблері включає в себе деяку кількість рядків. Розглянемо, наприклад, программу представлену на рис.1.2. Дана програма містить спрощені директиви визначення сегментів DOSSEG .MODEL, .STACK, .DATA і .CODE, а також директиву END.

```
 TITLE Перша програма        ; заголовок
 DOSSEG                      ; директива впорядкування сегментів
.MODEL SMALL
.STACK 200h
.DATA
Message  DB 'Привет!',13,10,'$'
.CODE
mov    ax,@Data
mov ds,ax                    ; встановити регістр DS  таким
                             ; чином, щоб він вказував
mov    ah,9                  ; функция DOS вывода строки
mov    dx,OFFSET Message     ; ссылка на сообщение "Привет!"
int    21h                   ; вывести "Привет!" на экран
mov    ah,4ch                ; функция DOS завершения  программы

int    21h                   ; завершить программу
END
```

Рис 1.2 Текст програми hello.asm

У кожній програмі на Асемблері, щоб забезпечити визначення сегментів і управління ними, необхідні директиви визначення сегментів (спрощені або стандартні), а завершувати програму на Асемблері завжди повинна директива END.

Директиви є лише "рамками" програми на Асемблері. У самій програмі необхідні також рядки вихідної коди, що виконують які-небудь дії, наприклад:

mov ah,9

Цими рядками є мнемоніки інструкцій, відповідні набору інструкцій процесора 8086, список яких приведений в Додатку 1.

1.7 Директиви DOSSEG, .MODEL, .STACK, .DATA, .CODE

Директива *.DOSSEG* приводить до того, що сегменти в програмі Асемблера будуть згруповані відповідно до угод по впорядковуванню сегментів фірми Microsoft. У даним момент вам не слід вникати в сенс того, що це означає. Запам'ятаєте просто, що майже всі автономні програми на

Асемблері прекрасно працюватимуть, якщо ви почнете їх з директиви *DOSSEG*.

Директива *.MODEL* визначає модель пам'яті в модулі Асемблера, де використовуються спрощені директиви визначення сегментів. Відмітимо, що в "ближньому" коді переходи здійснюються за допомогою завантаження одного регістра IP, а в "далекому" коді - шляхом завантаження регістрів CS і IP. Аналогічно, до "ближніх" даним звернення виконується лише по зсуву, а до "далеких" - за допомогою повної адреси "сегмент:зсув". Коротко, термін "далекий" (FAR) означає використання повної 32-розрядної адреси ("сегмент:зсув"), а "ближній" (NEAR) означає використання 16-розрядних зсувів. Існують наступні моделі пам’яті: зверхмала, мала, середня, компактна, велика, зверхвелика.

Директиви визначення сегментів *.STACK*, *.CODE* і *.DATA* визначають, відповідно, сегмент стека, сегмент коду і сегмент даних. Наприклад, директива:

```
.STACK 100h
```

Визначає стек розміром 200h (512) байт. Що стосується стеку, то це все що ви зможете зробити. Необхідно переконатися, що у вашій програмі є директива .STACK, і Турбо Асемблер виділить для вас стек. Для звичайних програм сповна досить стек розміром 200h, хоча в програмах, що інтенсивно використовують стек (наприклад, в програмах, що містять рекурсивні виклики) може потрібно стек більшого розміру.

Директива *.CODE* відзначає початок сегменту коди. Ви можете порахувати, що для Турбо Асемблера досить очевидний, що всі ваші інструкції відносяться до сегменту коди. Насправді Турбо Асемблер дозволяє вам (за допомогою стандартних директив визначення сегментів) використовувати декілька сегментів коди, а директива *.CODE* вказує Турбо Асемблеру, в який саме сегмент треба помістити ваші інструкції. Визначення сегменту коди ще простіше, ніж визначення сегменту стека, оскільки аргументи для директиви *.CODE* вказувати не потрібно. Наприклад:

```
mov    ax,@Data
mov ds,ax          ; установить регистр DS  таким
                   ; образом, чтобы он указывал
                   ; на сегмент данных
```

Директива *.DATA* декілька складніша. Як можна зрозуміти, директива *.DATA* відзначає початок сегменту даних. У цьому сегменті слід розміщувати ваші змінні пам'яті. Наприклад:

```
Message  DB 'Привет!',13,10,'$'
```

Це досить просто. Вся "складність" директиви *.DATA* полягає в тому, що до того, як ви звертатиметеся до елементів пам'яті в сегменті, визначеному за допомогою директиви *.DATA*, потрібно явно завантажувати сегментний регістр DS ідентифікатором *@data*. Оскільки сегментний регістр можна завантажити з регістра загального назначення або елементи пам'яті, але в нього не можна завантажити константу, регістр DS зазвичай завантажується за допомогою послідовності з двох інструкцій представлені вище на прикладі директиви *.CODE*.

(Замість регістра AX можна використовувати будь-який загальний регістр.) Дана послідовність інструкцій встановлює DS так, щоб він вказував на сегмент даних, який починається по директиві *.DATA*.

Без двох інструкцій, які встановлюють регістр DS в значення сегменту, визначеного за допомогою директиви *.DATA*, функція друку рядка правильно не працюватиме. Рядок *Message* знаходиться в сегменті даних і недоступний, поки регістр DS не буде встановлений в значення цього сегменту. Це можна розглядати таким чином: коли ви викликаєте операційну систему DOS для друку рядка, в парі регістрів DS:DX ви передаєте повну адресу у форматі "сегмент:зсув". Повний покажчик вигляду "сегмент:зсув" ви отримаєте лише після того, як в регістр DS буде завантажений сегмент даних, а в DX - зсув *Message*.

У вас може виникнути питання, чому потрібно завантажувати регістр DS, а не CS або SS (або ES)?

По-перше, регістр CS ніколи не завантажується явно, оскільки при запуску програми за вас це робить операційна система DOS. Крім того, якби регістр CS вже не був встановлений, коли прийшов час виконати першу інструкцію програми, процесор 8086 не знав би, де знайти цю інструкцію, і програма не стала б працювати. Поки для вас це може бути недостатньо очевидно, але повірте нам на слово: регістр CS завантажується при запуску програми автоматично, і у вас немає необхідності завантажувати його явно.

Аналогічно, при запуску програми DOS завантажує регістр SS, значення якого при виконанні програми зазвичай залишається незмінним. Хоча змінювати вміст регістра SS можна, це рідко виявляється бажаним, і визначено це не варто робити, якщо ви не знаєте точно, що ви робите. Таким чином, регістр SS аналогічно регістру CS встановлюється при початку виконання програми автоматично, і далі його чіпати не потрібно.

SS - на стек, регістр DS вказує на дані. Програми не працюють безпосередньо з інструкціями або стеком, проте з даними вони працюють постійно. Крім того, програми можуть у будь-який момент побажати отримати данні в декількох різних сегментах. Потрібно пам'ятати про те, що процесор 8086 у будь-який час дозволяє вам дістати доступ до будь-якого елементу пам'яті в межах 1 Мбайта, але лише в блоках по 64К (відносно сегментного регістра).

Ви можете захотіти завантажити регістр DS одним сегментом, посвітити доступ до даних в цьому сегменті, а потім завантажити DS іншим сегментом, щоб звернутися до іншого блоку даних. У маленьких і середніх програмах (таких, як в наведених нами прикладах) вам не потрібно буде використовувати більш за один сегмент даних, але у великих програмах декілька сегментів даних використовується часто. Крім того, вам потрібно буде завантажувати регістр DS різним значеннями, якщо ви хочете дістати доступ до системних областей пам'яті, наприклад, до елементів пам'яті, використовуваних базовою системою введення-виводу (BIOS).

Зі всього цього можна зробити наступний короткий вивід: Турбо Асемблер дозволяє вам у будь-який момент встановити регістр DS в значення будь-якого сегменту. За цю гнучкість доводиться розплачуватися тим, що ви повинні явно встановлювати регістр DS в значення потрібного сегменту (зазвичай @data), що еквівалентно сегменту, який починається з директиви *.DATA*. Після цього ви зможете дістати доступ до елементів пам'яті цього сегменту.

Сегментний регістр ES завантажується аналогічно регістру DS. Найчастіше вам не потрібно буде використовувати регістр ES, але коли з'явиться необхідність дістати доступ до елементу пам'яті в сегменті, на який вказує регістр ES, ви повинні спочатку завантажити регістр ES значенням цього сегменту. Наприклад, наступна програма завантажує регістр ES значенням сегменту *.DATA*, а потім завантажує через ES символ, який потрібно надрукувати з цього сегменту:

```
        DOSSEG
       .MODEL  small
       .STACK  200h
      .DATA
OutputChar     DB     'B'
       .CODE
ProgramStart:
        mov     dx,@data
        mov     es,dx                       ;встановити ES в значення
                                            ; сегменту .DATA
        mov   bx,OFFSET OutputChar          ; BX указує на
                                            ; зміщення OutputChar
        mov   al,es:[bx]                    ; отримати виведений символ
                                            ; з сегменту, на який
                                            ; вказує регистр ES
        mov   ah,2                          ; функція DOS виводу символа
int   21h                                   ; викликати DOS для виводу
                                            ; символа на екран
END   ProgramStart
```

Звернете увагу, що регістр ES (як і регістр DS раніше) завантажується послідовністю з двох інструкцій:

```
        .
        .
```

```
.
mov   dx,@Data
mov   es,dx
.
.
.
```

Зрозуміло, в даному прикладі немає конкретної причини використовувати замість DS регістр ES. Фактично, використання регістра ES означає, що ми застосували префікс перевизначення сегменту ES. Проте у багатьох випадках надзвичайно зручно, коли регістр DS вказує на один сегмент, а регістр ES - на іншій (особливо це стосується використання строкових інструкцій).

2 Порядок виконання роботи

2.1 Створити файл в текстовому редакторі, який формує файл в коді ASCII hello.asm згідно Рис.1.2, але замінивши на своє повідомлення в строчці:

```
Message  DB 'Привет!',13,10,'$'
```

2.2 Зберегти свій файл в папку обов’язково з розширенням .asm, наприклад Lab1.asm.

2.3 Переконатися, що в папці є асемблювальник TASM та компонувальник TLINK.

2.4 В командній строчці набрати TASM lab1, отримати об’єктний файл lab1.obj. В випадку неможливості отримати lab1.obj перевірити на помилки, переглянути їх можна натиснувши CTRL+O.

2.5 В командній строчці набрати TLINK lab1, отримати виконуваний файл lab1.exe. В випадку неможливості отримати lab1.exe перевірити на помилки.

2.6 Запустити виконуючий файл.

3 Зміст звіту

3.1 Назва та мета роботи

3.2 Блок-схема алгоритму створення програми на Turbo Assembler

3.3 Текс програми

3.4 Результат виконання програми

3.5 Висновки по роботі

4 Контрольні питання

4.1 Назвіть повний етап створення програми на Turbo Assembler?

4.2 Як виконати асемблювання та компоновку?

4.3 Для чого потрібен файл TLINK.EXE?

4.4 Для чого потрібен файл TASM.EXE?

4.5 Як виконується компоновка файлу?

4.6 Як виконується асемблювання файлу?