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):

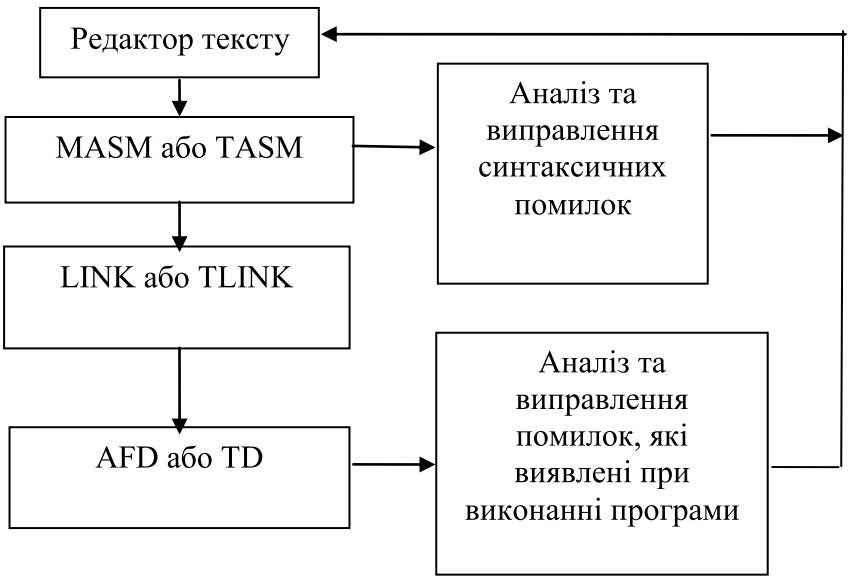

Рис.1.1. Послідовність розробки програми мовою Асемблера

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

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

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

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

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**.