МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
імені ВОЛОДИМИРА ДАЛЯ

МЕТОДИЧНІ ВКАЗІВКИ

до практичних занять з дисципліни
«СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ»
Частина II

(для здобувачів вищої освіти 3 курсу денної та заочної форми навчання за спеціальністю 123 "Комп'ютерна інженерія")

ЗАТВЕРДЖЕНО
на засіданні кафедри
комп'ютерних наук та інженерії
Протокол №7 від 07.04.2021 р.

Сєвєродонецьк 2021

УДК 004.45

Методичні вказівки до практичних занять з дисципліни «Системне програмне забезпечення» Ч.ІІ (для здобувачів вищої освіти 3 курсу денної та заочної форми навчання за спеціальністю 123 "Комп’ютерна інженерія") / Уклад.: М.В.Деркач. – Сєверодонецьк: вид-во СНУ ім. В. Даля, 2021. – 70 с.

Методичні вказівки спрямовано на вивчення основ програмної роботи з обладнанням обчислювальної машини на низькому рівні, методів та алгоритмів взаємодії програм та пристроїв ПК. З огляду на це, в тематиці відображені ключові задачі, що вирішуються під час програмування на низькому рівні: структури даних, арифметичні та логічні методи обробки інформації, керуючі алгоритмічні конструкції.

Укладач: М.В. Деркач, доц.

Відповідальний за випуск: О.І. Рязанцев, проф.

Рецензент М.Є. Щербакова

ЗМІСТ

ВСТУП. ТЕХНОЛОГІЯ РОЗРОБКИ ПРОГРАМ МОВОЮ АСЕМБЛЕР 4
СТРУКТУРА ПРОГРАМИ НА МОВІ INTEL-ASSEMBLER ТА ОГОЛОШЕННІНЯ ДАНИХ 10
1.1 Методичні вказівки 10
1.2 Теоретичні відомості 11
1.3 Порядок виконання роботи 20
1.4 Контрольні запитання 20
2 СПОСОБИ АДРЕСУВАННЯ ТА КОМАНДИ ПЕРЕСИЛАННЯ ДАНИХ 21
2.1 Методичні вказівки 21
2.2 Теоретичні відомості 21
2.3 Порядок виконання завдання 33
2.4 Завдання для самостійного виконання 33
2.5 Контрольні питання 35
3 АРИФМЕТИЧНІ КОМАНДИ МОВИ ASSEMBLER ДЛЯ ДВІЙКОВИХ ДАНИХ 36
3.1 Методичні вказівки 36
3.2 Теоретичні відомості 36
3.3 Порядок виконання роботи 40
3.4 Завдання для самостійного виконання 40
3.5 Контрольні запитання 41
4 КОМАНДИ ПОРІВНЯННЯ, 42
УМОВНОГО ТА БЕЗУМОВНОГО ПЕРЕХОДІВ 42
4.1 Методичні вказівки 42
4.2 Теоретичні відомості 42
4.3 Порядок виконання роботи 46
4.4 Завдання для самостійного виконання 46
4.5 Контрольні запитання 47
5 КОМАНДИ ЛОГІКИ ТА ЗСУВУ. ПОРОЗРЯДНА ОБРОБКА І ФОРМУВАННЯ ДАНИХ 48
5.1 Методичні вказівки 48
5.2 Теоретичні відомості 48
5.3 Порядок виконання роботи 54
5.4 Завдання для самостійного виконання 55
5.5 Контрольні запитання 55
6 ПРОГРАМУВАННЯ ЦИКЛІВ 57
6.1 Методичні вказівки 57
6.2 Теоретичні відомості 57
6.3 Порядок виконання роботи 59
6.4 Завдання для самостійного виконання 59
6.5 Контрольні запитання 60
7 РОЗРОБКА ПРОГРАМ СТВОРЕННЯ ТА ОБРОБКИ МАСИВІВ 61
7.1 Методичні вказівки 61
7.2 Теоретичні відомості 61
7.3 Порядок виконання роботи 67
7.4 Завдання для самостійного виконання 67
7.5 Контрольні запитання 68
ПЕРЕЛІК ЛІТЕРАТУРИ……………………………………………………...69

ВСТУП. ТЕХНОЛОГІЯ РОЗРОБКИ ПРОГРАМ МОВОЮ АСЕМБЛЕР

Розробка програм (і не лише мовою Асемблер) відбувається у декілька етапів:

1. Постава задачі;
2. Вибір (розробка) алгоритму вирішення;
3. Введення тексту програми;
4. Трансляція та виправлення синтаксичних помилок;
5. Налагоджування та виправлення семантичних помилок;
6. Супровід програми.

Розглянемо докладніше кожен зі згаданих етапів з огляду на використання мови Асемблер.

Для створення ефективної програми, першочергове значення має правильна постава задачі. Вона полягає у точному усвідомленні того, що дано на початку роботи програми, і того, що потрібно отримати як результат її роботи. Правильна постанова задачі сприяє правильному вибору алгоритму її вирішення. Якість обраного алгоритму також має вирішальне значення з точки зору ефективності майбутньої програми. Наприклад, алгоритм пошуку за методом дихотомії (інші назви: двійковий пошук, метод половинного поділу) у масиві з тисячі елементів є приблизно у 50 разів ефективнішим за алгоритм лінійного пошуку. Таким чином, застосування більш ефективного алгоритму може призвести до такого ж результату як і запровадження нової, більш швидкої обчислювальної техніки, але коштуватиме значно дешевше.

Головні труднощі алгоритмізації (переходу від постанови задачі до детального алгоритму) полягають в тому, щоб подумки уявити і записати без помилок всю послідовність дій машини, які потрібні для отримання правильного рішення. Досвід свідчить, що зручним допоміжним прийомом під час переходу від постанови задачі до алгоритму є наочне схематичне зображення процесу перетворень (переміщень, переставлень, тощо) об'єктів, які входять до поданої у задачі ситуації. Так, наприклад, під час сортування масиву, можна за допомогою стрілок зобразити схему переміщення інформації в ньому. Таке наочне «роз'яснення для себе» дозволяє уникнути чисельних помилок, котрі завжди з'являються під час спроби алгоритмізації скільки-небудь складної задачі подумки.

Найбільш корисними, у цьому сенсі, є рисунки, на котрих зображуються: а) основні об'єкти (елементи) задачі; б) їх позначення у майбутній програмі; в) дії по змінюванню та переміщенню об'єктів; г) початковий та кінцевий стан (розташування) об'єктів.

Введення тексту програми відбувається за допомогою текстового процесора (редактора). Текстових редакторів існує чимало. Для зручного створення програм мовою Асемблер нас влаштує будь який з них, здатний забезпечити наступне:

- створення файлів у текстовому форматі (текст складається з символів ASCII, кожен рядок закінчується комбінацією керуючих символів з кодами 13 та 10);
- вирівнювати елементи тексту програми за допомогою символів табуляції;
- відображати позицію (номер рядка) в тексті.

Зазначені вимоги випливають з наступних практичних міркувань. По-перше, більшість трансляторів «не розуміють» файлів інших форматів окрім текстового. Особливо слід остерігатися при використанні редактора MS Word. Створювані за допомогою цього редактора файли, за замовчуванням, мають формат, що не розпізнається більшістю трансляторів.

По-друге, використання символу табуляції замість декількох пробілів значно пришвидшує роботу з текстом. Відомо, що програма, складена мовою асемблер, повинна мати певний формат де кожен рядок розподіляється на декілька зон. Акуратність оформлення тексту, чітке виділення структури програми значно спрощують роботу програміста і підвищують ефективність цієї роботи.

По-третє, дуже рідко щойно складена програма не містить помилок. Частіше – навпаки. Тоді, транслятор повідомляє яка помилка і у якому рядку програми має місце (йдеться лише про синтаксичні помилки). Тож, необхідно мати редактор, що дозволить знайти у програмі рядок із зазначеним номером. З огляду на це, стандартний редактор «Блокнот», присутній у OS Windows, не є зручним для роботи з асемблером.

Натомість, під час навчання, зручними можуть бути більшість редакторів вбудованих у операційні.

Невелика порада щодо вибору імені файлу з програмою мовою асемблер. Оскільки існує чимало трансляторів, котрі «не розуміють» кирилицю та довгих імен файлів, бажано щоб ім’я файлу складалося не більше ніж з восьми латинських літер. Розширення у файлу має бути – «.asm».

Коли текст програми введено і збережено на диску у вигляді файлу, його слід транслювати у завантажувальний модуль – програму складену машинною мовою. Така трансляція має бути виконана або за допомогою пакета assembler TASM, або пакета assembler MASM.

Пакет TASM

Трансляція виконується у два етапи.

На першому етапі трансляції слід отримати т. з. «об’єктний» модуль програми (файл з розширенням .obj). Ця операція виконується за допомогою програми «tasm.exe». Скористатися цією командою можна шляхом її введення у полі для введення команд, яке зазвичай міститься в нижній частині екрану. Команда, зазвичай має наступний вигляд:

```
tasm.exe /z /zi myprog.asm
```

Замість «myprog.asm» потрібно вказати ім’я власного файлу з програмою.

Параметр «/z» – не є обов’язковим, але його наявність змушує транслятор «tasm.exe», в разі знаходження синтаксичних помилок, виводити на екран не лише повідомлення з назвою помилки і номером рядка, в якому помилку знайдено, а й сам зазначений рядок; що є доволі зручно.

Параметр «/zi» – не є обов’язковим, але його наявність змушує транслятор tasm.exe, додавати до об’єктного файлу т. з. інформацію для налагодження. Це інформація, яка дозволяє встановлювати однозначну відповідність між частинами машинної програми і відповідної початкової програми. Наявність такої інформації робить процес налагодження значно зручнішим.

Програма «tasm.exe» може використовуватися і з іншими параметрами. Щоб отримати на екрані їх перелік з короткими коментарями слід запустити програму «tasm.exe» з єдиним параметром – «/?», або, – взагалі без параметрів.

Якщо програма, що транслюється, містить синтаксичні помилки, програма «tasm.exe» повідомляє про це і об’єктний файл не генерується. В такому разі слід повернутися до роботи з текстовим редактором, виправити всі помилки і повторити трансляцію. Процес повторюється допоки програма «tasm.exe» не повідомить, що помилок більше немає і відбулася успішна трансляція. Результатом успішної трансляції є файл з тією ж назвою, що і початковий, але з розширенням «.obj». (У наведеному прикладі буде створено файл myprog.obj).

На другому етапі трансляції на основі об’єктного файлу, отриманого на першому етапі трансляції, слід отримати т.з. «завантажувальний» модуль програми (файл з розширенням .exe). Ця операція виконується за допомогою програми редактора зв’язків – «tlink.exe». Скористатися цією командою можна шляхом її введення у полі для введення команд, яке зазвичай міститься в нижній частині екрану. Команда, зазвичай має наступний вигляд:

```
tlink.exe /v myprog.obj
```

Замість «myprog.obj» потрібно вказати ім’я власного об’єктного файлу.

Параметр «/v» – не є обов’язковим, але його наявність змушує редактор зв’язків «tlink.exe», додавати до завантажувального файлу т. з. інформацію для налагодження. Програма «tlink.exe» також може видавати повідомлення про помилки, але такі випадки трапляються вкрай рідко.

Наявність двох етапів трансляції має певний практичний сенс. Програма може проектуватися групою програмістів. Кожен з програмістів може автономно розробляти і транслювати власний фрагмент програми

(наприклад, підпрограму). На другому етапі, об'єктні файли, отримані від різних програмістів поєднуються у завантажувальний модуль за допомогою редактора зв'язків – «tlink.exe». Таким чином, програмісту зовсім не обов'язково розкривати тексти своїх програм.

Об'єктний код – це програма, за великим рахунком, вже перекладена машинною мовою. Але зв'язки між частинами програми ще визначено не остаточно. Наприклад, якщо підпрограма «A» здійснює виклик іншої підпрограми «B», котра знаходиться в іншому «.asm» файлі (а можливо, і на іншому комп'ютері), то програма «tasm.exe» просто фізично не спроможна визначити дійну адресу підпрограми «B» і змушена тільки означити місце виклику. Остаточне визначення таких посилань і виконує редактор зв'язків «tlink.exe», котрий збирає до купи всі частини програми і вже має в наявності інформацію про всі підпрограми, та глобальні змінні (котрі також може бути визначено в одному «.asm» файлі, а використано в іншому).

Оскільки трансляція – доволі складний процес, що потребує ретельності і уваги під час введення команд та їхніх параметрів, доцільним видається цей процес автоматизувати шляхом використання командних файлів.

Командний файл – текстовий файл з розширенням «.bat», що містить низку команд операційної системи. Опрацьовуючи цей файл, операційна система виконує почергово наявні команди. Для автоматичної трансляції програм, складених мовою Асемблер, може стати у пригоді командний файл наступного змісту:

```
@echo off
del %1.exe
del %1.obj

tasm.exe /z /zi %1.asm

if exist %1.obj tlink.exe /v %1.obj
if not exist %1.obj goto error

if exist %1.exe td.exe %1.exe
if not exist %1.exe goto error

goto end

:error
echo ***** П О М И Л К А !!! *****
pause

:end
```

```
del *.obj
del *.map
```

Щоб скористатися цим командним файлом достатньо його запустити (через командний рядок), передавши у якості параметра ім’я «.asm» файлу без розширення (!).

Наприклад, якщо командний файл зветься «trans.bat», а файл з програмою зветься «myprog.asm», то відповідний командний рядок виглядатиме наступним чином:

trans.bat myprog

Єдина вимога – щоб всі потрібні файли (tasm.exe, tlink.exe, td.exe, myprog.asm, trans.bat) було розташовано у поточному каталозі (папці).

Якщо трансляція програми була успішною і було отримано завантажувальний модуль («.exe»), це ще зовсім не означає, що розроблена програма не має помилок. Вона, швидше за все, не має синтаксичних помилок. Перевірка синтаксичної правильності складеної програми – справа формальна і транслятор може легко з нею впоратися. Інша річ – помилки семантичні. Транслятор ніяким чином не зреагує, якщо, наприклад, у арифметичному виразі замість символу «+» програміст використає символ «-». З огляду на синтаксис, використання обох символів є припустимим, а який саме символ має бути присутнім з огляду на алгоритм, що реалізується, знає лише сам програміст, і аж ніяк не транслятор. Таким чином, часто виникає ситуація, коли програма, що успішно пройшла трансляцію, не виконує покладені на неї дії, а виконує щось зовсім інше. Така програма потребує налагодження.

Налагодження програми полягає у послідовному, крок за кроком, її виконанні з метою виявити місце, в якому дії програми відмінні від дій, на які сподівався програміст. Це особливо актуально в разі, коли програма досить є великою за розміром і знайти помилку у інший спосіб проблематично.

Інструментальним засобом, для забезпечення процедури налагодження є спеціальна програма – налагоджував. Таких програм існує чимало.

Пакет MASM

Текст програми повинен бути підготовлений за допомогою редактора тексту на диску. Потім потрібно виконати асемблювання, видав директиву:

masm [ключі]<ім'я файлу програми>[,[<об'єктний файл>] [,[<файл_ лістингу>][,[<файл_ перехресних посилань >]]]]

Ключі можуть бути відсутніми. Найбільш часто застосовуються такі ключі:

/L– вказує на необхідність створення файлу лістингу;

/z – при виникненні помилок поряд з повідомленням про них виводити відповідні рядки;

/zd – включення в об'єктний файл інформацію про номери рядків для відладчика;

/zi – включити в об'єктний файл інформацію про ідентифікатори, необхідну для відладчика.

Повний перелік ключів можна отримати за командою masm.exe (без параметрів).

За першою директорії текст програми з результатами асемблювання виводиться на екран, а по другій - на друк.

Якщо програма пройшла успішне асемблювання - то виконати компонування програми, видав директиву:

link [ключі]< ім'я файлу програми >.OBJ

Ключі можуть бути відсутніми. Найбільш часто застосовуються такі ключі:

/co– додати інформацію про ідентифікатори, необхідну для відладчика.

Для перевірки і відладки роботи програми необхідно використовувати відладчик Microsoft CodeView Debuger – cv.exe

Виконання програми за допомогою відладчика

Ввести команду:

cv [ключі]< виконавчий файл >

/R - дозволяється використовувати налагоджувальні регістри 80386/80486

1 СТРУКТУРА ПРОГРАМИ НА МОВІ INTEL-ASSEMBLER ТА ОГОЛОШЕНННЯ ДАНИХ

Мета роботи: Опанування технологією розробки програм мовою асемблер, побудови асемблерної програми, оголошення та визначення даних.

1.1 Методичні вказівки

Щоб успішно впоратись з завданням слід ретельно вивчити призначення та особливості використання регістрів. Вивчити формати команд резервування даних, способи запису констант.

Команди визначення – резервування пам'яті задають тип змінної і кількість байт для розташування значення змінної відповідно до таблиці 1.1.

Під час практичного засвоєння матеріалу, студенти мають ввести наведену програму, виконати її трансляцію, знайти та виправити можливі помилки, а також, дослідити, крок за кроком, її виконання за допомогою налагоджувальної програми. Це дозволить відчути особливості роботи процесора, здобути навички програмування на низькому рівні.

Таблиця 1.1 - Команди визначення пам'яті

Команда	Тип	Розмір(в байтах)
DB	BYTE	1
DW	WORD	2
DD	DWORD	4
DQ	QWORD	8
DT	TBYTE	10

Формат команд визначення пам'яті*:*

< ім'я змінної ><команда визначення пам'яті >< список виразів >

Ім'я змінної і список виразів повинні відділятися від команди хоча б одним пробілом або знаком табуляції. Вирази в списку відокремлюються один від одного комами. У таблиці 1.2 наведені приклади усіх можливих видів виразів.

У разі резервування пам'яті для великих сукупностей даних таких, як масиви, можуть використовуватися вирази, структура яких наведена нижче:

< кількість > *DUP* (<вираз, що визначає початкове значення>)

Приклади:

MAS DW 52 DUP(?); резервується 104 байти з невизначеними значеннями

MMM DD 100 DUP(0); резервується 400 байт із значенням 0

При написанні програми необхідно знати формат і призначення регістрів.

Таблиця 1.2- Приклади опису

Поле мітки	Команда	Поле операндів
A1	**DB**	? ; резервується 1 байт з невизначеним значенням
BB23	**DW**	-12 ; резервується 2 байти з занесенням початкового значення -12
DLL	**DB**	?,?,? ; резервується 3 байти з невизначеним значенням, причому ім'я DLL буде відноситися до першого з них.
STR2	**DB**	'Table'; резервується 5 байтів, в кожен міститься по одному символу, а ім'я STR2 буде відноситися до першого з них.

1.2 Теоретичні відомості

Створення тексту програми

Текст асемблерної програми створюється у вигляді одного або декількох файлів з розширенням *.asm. Для цього можна використати будь-який екранний редактор текстів. У Windows для створення програми можна використати програми NotePad.exe або WordPad.exe. При необхідності один файл можна включити в інший за допомогою директиви: INCLUDE назва_файлу.

Компіляція програми

Після створення програми здійснюється її перетворення у машинний код. Для цього використовується один із компіляторів, наприклад tasm.exe або masm.exe . Задачею компілятора є виявлення синтаксичних помилок та створення машинного коду – файлу із розширенням *.obj (об’єктного файлу). Об’єктний файл створюється тільки при відсутності синтаксичних помилок у програмі. Компілятор з мови асемблера є двохпрохідним, він переглядає текст програми два рази. На першому проході компілятор будує таблицю імен та їх відносну адресу. На другому етапі формується об’єктний код. При необхідності компілятор може згенерувати файл лістингу *.lst та файл перехресних посилань *.crf . Файл лістингу, крім асемблерних команд містить їх машинні коди та можливі помилки. Файл перехресних посилань визначає які команди посилаються на які дані.

Загальний формат команди tasm:

TASM [ключи] имя_исходнрго_файла [,имя_объектного_файла] [,имя_файла_листинга] [,имя_файла_перекрестных_ссылок]
TASM [опції] *.asm [,*.obj] [,*.lst] [,*.crf]

Замість символу '*' вказується ім'я файлу. Квадратні дужки позначають необов'язкові параметри команди. Для запуску компілятора використовується ряд опцій, значення яких можна вивчити, запустивши команду tasm.exe з командного рядка. Наприклад, для включення в об'єктний файл інформації для програми відлагодження необхідно використати опцію /zi :

TASM /zi *.asm

Для отримання файлу лістингу додатково можна використати опцію /la:

TASM /zi /la *.asm

Об'єктний файл – це ще не готова до виконання машинна програма, оскільки може містити неініціалізовані адреси зовнішніх посилань.

Редагування зв'язків програми

Для перетворення об'єктного файлу у виконуваний файл *.exe використовується редактор зв'язків:

TLINK *.obj[, *.exe][, *.map][, *.lib][, *.def]

де *.obj – один або декілька об'єктних файлів;
*.exe – виконуваний файл;
*.map – карта розподілу пам'яті;
*.lib – бібліотечні файли;
*.def – файл визначення модуля програми.

Обов'язковим є тільки перший параметр команди, що визначає ім'я об'єктного файлу.

Якщо ім'я exe-файлу не вказано, то воно буде таким як ім'я obj-файлу.

Команда може містити ряд опцій, значення яких можна вивчити, запустивши команду з командного рядка програму: tlink.exe.

Наприклад:
TLINK /v *.obj - включити відлагоджувальну інформацію в exe-файл;
TLINK /t *.obj - створити com-файл.
Замість символу '*' необхідно вказати ім'я файлу.

Виконання програми

Після створення виконуваного коду у вигляді *.exe або *.com – програми можна її виконати. Запуск програми здійснюється з командного рядка або з програмної оболонки для роботи з файлами.

Використання програми відлагодження програми

При необхідності для перегляду результатів або для вивчення помилок у роботі програми можна використати програму відлагодження, наприклад td.exe . Для цього у командному рядку необхідно набрати:

TD.EXE *.EXE

Замість символу ‘*’ необхідно вказати ім’я файлу.

1.Створити файл *tasm.bat* для компіляції програми:

```
del %1.exe
tasm  /zi %1.asm tlink  /v %1.obj
del %1.obj del %1.map
```

2. Набрати подані нижче програми, зберегти їх у файлах з розширенням “*.ASM* ”.

3. Відкомпілювати набрані програми за допомогою командного рядка:

tasm.bat <u>*назва файлу програми без розширення*</u>

4. Вивчити режими запуску компілятора та редактора зв’язків.

5. Отримати файли лістингу *.lst, розглянути та вивчити загальну структуру програм.

6. Запустити одержані exe-файли на виконання.

7. Запустити програму для відлагодження td.exe та вивчити режими її роботи.

Опції транслятора TASM та редактора зв’язків TLINK

Таблиця 1.3 - Опції транслятора TASM

Опція	Призначення опції
/a, /s	**/a** - сегменти в об'єктному файлі повинні бути розміщені за абеткою; **/s** - сегменти в об'єктному файлі розташовуються у порядку їхнього опису в програмі
/c	вказівка на включення у файл лістинга з інформацією про перехресні посилання

/e, /r	**/e** — генерація інструкцій емуляції операцій із плаваючою крапкою; **/r** — дозвіл трансляції дійсних інструкцій із плаваючою крапкою, які повинні виконуватися реальним арифметичним співпроцесором
/h, /?	вивід на екран довідкової інформації. Це еквівалентно запуску TASM без параметрів
/i<шлях>	задає шлях до файлу, який включається за директивою INCLUDE. Синтаксис аргументу "шлях" такий же, як для команди PATH файлу autoexec.bat
/jдиректива_TASM	визначає директиви, які будуть транслюватися перед початком трансляції вихідного файлу програми на Асемблері. У директиві не повинно бути аргументів
/khn	задає максимальну кількість ідентифікаторів, що може містити вихідна програма, фактично задається розмір таблиці символів транслятора. За замовчуванням програма може містити їх до 16384. Це значення можна збільшити (не більш ніж до 32 768) або зменшити до n. Сигналом до того, що необхідно використати даний параметр, служить поява повідомлення "Out of hash space" ("Буферний простір вичерпаний")
/l, /la	**/l** - указує на необхідність створення файлу лістинга, навіть якщо він не "замовляється" у командному рядку; **/la** - показати у лістингу код, який вставляється транслятором для організації інтерфейсу з мовою високого рівня по директиві MODEL
/ml, /mx, /mu	**/ml** — розрізняти у всіх ідентифікаторах прописні й малі літери; **/mx** - розрізняти малі й прописні символи в зовнішніх і загальних ідентифікаторах. Це важливо при компонуванні із програмами на тих мовах високого рівня, у яких малі й прописні символи в ідентифікаторах розрізняються; **/mu** - сприймати всі символи ідентифікаторів як прописні
/mvn	визначення максимальної довжини ідентифікаторів. Мінімальне значення n дорівнює 12
/mn	установка кількості (n) проходів транслятора TASM. За замовчуванням транслятор виконує один прохід. Максимально при необхідності можна задати виконання до 5 проходів
/os, /o, /op, /oi	генерація оверлейного коду
/p	перевіряти наявність коду з побічними ефектами при роботі в захищеному режимі
/q	видалення з об'єктної програми зайвої інформації,

	непотрібної на етапі компонування
/t	придушення виводу всіх повідомлень при умовній трансляції, крім повідомлень про помилки (тобто тестування програми на предмет виявлення синтаксичних помилок)
/z	при виникненні помилок разом з повідомленням про них виводити відповідні рядки тексту
/zi, /zd, /zn	/zi — включити в об'єктний файл інформацію для налагодження; /zd - помістити в об'єктний файл інформацію про номери рядків, які необхідні для роботи налагоджувальника на рівні вихідного тексту програми; /zn - заборонити додавання в об'єктний файл налагоджувальної інформації.

Таблиця 1.4 - Опції редактора зв'язків TLINK

Опція	Призначення опції
/x	Не створювати файл карти (map)
/m	Створити файл карти
/s	Те саме, що **/m**, але додатково у файл карти включається інформація про сегменти (адреса, довжина в байтах, клас, ім'я сегмента й т. ін.)
/l	Створити розділ у файлі карти з номерами рядків
/n	Ігнорувати бібліотеки, які вказуються іншими компіляторами
/c	Розрізняти малі й прописні букви в ідентифікаторах (у тому числі й зовнішніх)
/v	Включити налагоджувальну інформацію у завантажувальний файл
/d	Попереджати про дублювання символів у бібліотеках, які компонуються
/t	Створити файл типу **.com** (за замовчуванням **.exe**)
/3	підтримка 32-розрядного коду (тільки tlink.exe).
/ax	тип додатку (тільки tlink32.exe): /aa — віконний Windows-додаток; /ap — консольний Windows-додаток.

Для запобігання несумісності слід використовувати програми TASM і TLINK однієї версії.

Опис регістрів

Для програміста мікропроцесор представляється основним адресним простором, адресним простором зовнішніх пристроїв і програмно-доступними регістрами.

Регістри за функціональним призначенням поділяються на:
– регістри загального призначення;
– індексні регістри і регістри бази;
– сегментні регістри;
– регістр прапорів;
– вказівник команд.

Схематичне позначення регістрів мікропроцесора 8086 наведено на рисунку 1.1, а мікропроцесора ix86 наведено на рисунку 1.2.

Використання регістрів загального призначення, а також індексних регістрів і регістрів-вказівників пояснюється в описі команд.

Рисунок 1.1 - Схематичне позначення регістрів мікропроцесора i8086

		AX	
EAX		AH	AL
EBX		BH	BL
ECX		CH	CL
EDX		DH	DL

	15	Регістри-індекси та вказівники	0
SP	Вказівник стека		
BP	Вказівник бази		
SI	Індексний регістр джерела		
DI	Індексний регістр призначення		

	31	Регістри-індекси та вказівники	0
ESI		**SI**	Індекс джерела
EDI		**Di**	Індекс приймача
EBP		**BP**	Вказівник бази
ESP		**SP**	Вказівник стека

	15 Сегментні регістри 0
CS	Регістр сегмента коду
DS	Регістр сегмента даних
SS	Регістр сегмента стека
ES	Регістр додаткового сегмента
FS	Регістр додаткового сегмента
GS	Регістр додаткового сегмента

	31	15 0
EIP		**IP**
	EFLAGS	

Рисунок 1. 2 - Схематичне позначення регістрів мікропроцесора ix86

Як видно з рисунка 1.2, регістри загального призначення і регістри-вказівники відрізняються від аналогічних регістрів i86 тим, що вони є 32-розрядними. Відповідно, до їх мнемонічним позначенням добавлена літера E (від extended, розширений). Для збереження сумісності з ранніми моделями процесорів допускається звернення до молодших половин всіх регістрів, які мають ті ж мнемонічні позначення, що і в i86 (AX, BX, CX, DX, SI, DI, BP и SP). Звичайно, збережена

можливість роботи з молодшими (AL, BL, CL и DL) и старшими (AH, BH, CH и DH) половинками регістрів мікропроцесора (МП) i86. Проте старші половини 32-розрядних регістрів не мають мнемонічних позначень і безпосередньо недоступні. Для того, щоб прочитати, наприклад, вміст старшої половини регістра EAX (біти 31...16) доведеться зрушити весь вміст EAX на 16 біт вправо (в регістр AX) і прочитати потім вміст AX. Все регістри загального призначення і вказівники програміст може використовувати на свій розсуд для тимчасового збереження адрес і даних розміром від байту до подвійного слова. Так, наприклад, можливе використання наступних команд:

```
mov EAX,0FFFFFFFFh      ;Робота з подвійним словом (32 біт)
mov AX,0FFFFh           ;Робота зі словом (16 біт)
mov AL, 0FFh            ;Робота з байтом (8 біт)
```

Всі сегментні регістри, як і в i86, є 16-розрядними. До їх складу включено ще два регістри - FS і GS, які можуть використовуватися для збереження сегментних адрес двох додаткових сегментів даних. Таким чином, при використанні розширених можливостей сучасних процесорів програмі одночасно доступні чотири сегменти даних, а не два, як в МП86.

Регістр вказівника команд також є 32-розрядним і зазвичай при описі процесора його називають EIP. Молодші шістнадцять розрядів цього регістра відповідають регістру IP процесора МП i86. Весь регістр EIP використовується тільки в 32-розрядних додатках; в 16-розрядних програмах адреси можуть бути тільки 16-розрядними і, відповідно, для адресації в програмному сегменті використовується молодша половина регістра EIP. Регістр прапорів прийнято називати EFLAGS (від extended flags, розширені прапори). Хоча він має довжину 32 біт, тільки молодші 18 біт (та й то не всі) містять значущу інформацію. Додатково до шести прапорів стану (CF, PF, AF, ZF, SF і OF) і трьом прапорам управління станом процесора (TF, IF і DF), він включає нові прапори завдання, рестарту і віртуального режиму, а також двохбайтне поле привілеїв вводу-виводу. Всі ці біти використовуються тільки в захищеному режимі і тут розглядатися не будуть. Програма мовою Асемблер структурно складається з частин, що звуться – сегментами. Призначення одних сегментів – зберігання даних. Інші сегменти містять команди програми. Зазвичай, прості програми складаються з трьох сегментів. А саме: сегмента даних, сегмента команд (кодів) і сегмента стека. Наведемо текст заготовки простої програми з подальшими коментарями:

```
Data        SEGMENT
; місце для визначення змінних
   Message  DB 'HELLO!',13,10,'$'
Data        ENDS
```

```
Code    SEGMENT
        ASSUME      DS:Data,CS:Code,SS:STK
;початок програми
Start:
;ініціалізація сегментних регістрів
        mov     ax,Data
        mov     ds,ax
        mov     ax,STK
        mov     ss,ax
;місце для визначення послідовності команд
        mov   ah,9              ; функція DOS виводу рядка
        mov   dx,OFFSET Message    ; адреса повідомлення
        int   21h               ; вивести "HELLO!" на екран
;вихід з програми
        mov     ah,4ch
        mov     al,0
        int     21h
Code ENDS
STK     SEGMENT     stack
        db      512     dup(0)
STK     ENDS
        END     Start
```

В наведеному тексті програми визначено три сегменти з назвами: Data, Code та STK. Сегмент даних, поки що, – порожній. Сегмент кодів містить лише ініціалізацію сегментних регістрів і код, пов’язаний із завершенням програми. Справа в тім, що під час завантаження програми на виконання, операційна система автоматично виконує ініціалізацію лише регістра кодового сегмента (CS), що побіжно і призводить до передачі управління. Натомість, значення інших сегментних регістри програміст має визначити самостійно, таким чином, щоб регістри DS і SS містили адреси пам’яті, починаючи з яких в ній розташовано сегменти даних і стеку відповідно. Вихід з програми означає повернення управління операційній системі. Один із способів це зробити – виклик системної функції 4ch (див. лістинг вище). При цьому, через регістр AL програма передає операційній системі т. з. код завершення, за допомогою якого програма сповіщає про результат своєї роботи. Цей результат може бути проаналізовано командним процесором операційної системи, наприклад, під час виконання командного файлу, за допомогою команди IF. Зазвичай, коли програма закінчується успішно, то код завершення дорівнює нулеві. В разі виникнення певних проблем, програма повертає код помилки. Так, програма-пакувальник RAR, в разі успішного розпакування архівного файлу, повертає нуль, а в разі неуспішного – відмінне від нуля значення (пошкоджений архів, неправильний пароль). Цим можна користатися для

складання командних файлів, що виконують автоматичну архівацію/розпаковування даних.

Особливу увагу слід звернути на вираз:

ASSUME DS:Data, CS:Code, SS:STK

Цей вираз міститься у кодовому сегменті і є псевдокомандою (директивою). На відміну від команд, котрі виконує процесор, псевдокоманди є вказівками до транслятору щодо його налаштовування та правильного трактування окремих сентенцій програми. Зокрема, згадана директива ASSUME вказує транслятору, що, коли не сказано інше, адреси всіх змінних слід вираховувати як зміщення відносно початку сегмента Data, а адреси переходів – відносно початку сегмента Code. Нагадаємо, що символічні імена вічка пам’яті (змінні) мають лише умовно, щоб програмісту не довелося мати справу з їх номерами. Після трансляції всі імена змінних замінюються на відносні адреси, а директива ASSUME уточнює порядок обчислення цих адрес. Вилучення директиви ASSUME призведе до необхідності щоразу перед іменем змінної явним чином вказувати відповідний сегментний регістр. Наприклад:

```
MOV     DS:SUMMA, 0
```

1.3 Порядок виконання роботи

1. Вивчити структуру програми, правила оголошень сегментів даних, стеку, коду програми.

2. Вивчити правила оголошення констант та змінних в асемблерній програмі.

3.Набрати програму та зберегти її у файлі з розширенням “*.ASM* ”.

4. Відкомпілювати набрану програму за допомогою командного рядка, використовуючи пакетний файл tasm.bat:

tasm.bat <u>*назва файлу програми без розширення*</u>

5. Запустити програму для відлагодження td.exe та прочитати результат роботи програми у сегменті даних.

6. Оформити звіт по роботі.

1.4 Контрольні запитання

1. Послідовність перетворення асемблерної програми для утворення виконуваного коду.

2. Призначення кроку компіляції асемблерної програми. Режими роботи компілятора.

3. Призначення кроку редагування зв’язків програми. Режими роботи редактора зв’язків.

4. Режими роботи програми відлагодження.

5. Типи констант, оголошення констант та змінних.
6. Запис числових констант у різних системах числення.

2 СПОСОБИ АДРЕСУВАННЯ ТА КОМАНДИ ПЕРЕСИЛАННЯ ДАНИХ

Мета: Вивчення способів адресування та набуття практичних навичок їх використання.

2.1 Методичні вказівки

З метою практичного засвоєння матеріалу слід зрозуміти що таке режими адресації, які способи адресації бувають.

Під час практичного засвоєння матеріалу, студенти мають ввести наведену програму, виконати її трансляцію, знайти та виправити можливі помилки, а також, дослідити, крок за кроком, її виконання за допомогою налагоджувальної програми. Це дозволить відчути особливості роботи процесора, здобути навички програмування на низькому рівні.

2.2 Теоретичні відомості

Способи адресації

Способом, або режимом адресації називають процедуру знаходження операнду для виконуваної команди. Якщо команда використовує два операнди, то для кожного з них має бути заданий спосіб адресації, причому режими адресації першого і другого операнду можуть як збігатися, так і розрізнятися. Операнди команди можуть знаходитися в різних місцях: безпосередньо у складі коди команди, в якому-небудь регістрі, в елементі пам'яті; у останньому випадку існує декілька можливостей вказівки його адреси. Строго кажучи, способи адресації є елементом архітектури процесора, відображаючи закладені в нім можливості пошуку операндів. З іншого боку, різні способи адресації певним чином позначаються в мові асемблера і в цьому сенсі є розділом мови.

Слід зазначити неоднозначність терміну "операнд" стосовно програм, написаних на мові асемблера. Для машинної команди операндами є ті дані (по суті, двійкові числа), з якими вона має справу. Ці дані можуть, як вже наголошувалося, знаходитися в регістрах або в пам'яті. Якщо ж розглядати команду мови асемблера, то для неї операндами (або, краще сказати, параметрами) є ті позначення, які дозволяють спочатку транслятору, а потім процесору визначити місцезнаходження операндів машинної команди. Так, для команди асемблера

```
mov mem, AX
```

як операнди використовується позначення елементу пам'яті mem, а також позначення регістра AX. В той же час, для відповідної машинної команди операндами є вміст елементу пам'яті й вміст регістра. Було б правильніше говорити про операнди машинних команд і про параметри, або аргументи команд мови асемблера.

По відношенню до команд асемблера було б правильніше використовувати термін "параметри", залишивши за терміном "операнд" позначення тих фізичних об'єктів, з якими має справу процесор при виконанні машинної команди, проте зазвичай ці тонкощі не враховують, і кажучи про операнди команд мови, розуміють насправді операнди машинних команд.

Регістрова адресація. Операнд (байт або слово) знаходиться в регістрі. Цей спосіб адресації застосовний до всіх регістрів процесора, що програмно-адресуються.

```
inc CH          ; Плюс 1 до вмісту CH
push DS         ; DS зберігається в стеку
xchg Bx, вр     ; BX и BP обмінюються змістом
mov ES, AX      ; Зміст AX пересилається в ES
```

Безпосередня адресація. Операнд (байт або слово) указується в команді і після трансляції поступає в код команди; він може мати будь-який сенс (число, адреса, код ASCII), а також бути представлений у вигляді символічного позначення.

```
mov AH, 40h     ;Число 40h завантажується в AH
mov Al,'*'      ;Код ASCII символу "*' завантажується в AL
int 21h         ;Команда переривання з аргументом 21h
limit = 528     ;Число 528 отримує позначення limit
mov Cx,limit    ;Число, позначене limit, завантажується в CX
```

Команда mov, використана в останній пропозиції, має два операнди; перший операнд визначається за допомогою регістрової адресації, другої, - за допомогою безпосередньої.

Важливим застосуванням безпосередньої адресації є пересилка відносних адрес (зсувів). Щоб вказати, що мова йде про відносній адресі даного осередку, а не про її вміст, використовується описувач offset (зсув):

```
; Сегмент даних
mes db "Урок 1'              ;Рядок символів
;Сегмент команд
 mov Dx,offset mes ;Адреса рядка засилається  в DX
```

У приведеному прикладі відносна адреса рядка mes, тобто номер першого байту рядка від початку сегменту, в якому вона знаходиться, заноситься в регістр DX.

Пряма адресація пам'яті. Адресується пам'ять; адреса елементу пам'яті (слова або байту) указується в команді (зазвичай в символічній формі) і поступає в код команди:

```
;Сегмент даних
meml dw 0          ;Слово пам'яті містить 0
mem2 db 230        ;Байт пам'яті містить 230
;Сегмент команд
inc meml  ;Зміст  слова meml збільшується на 1
mov DX, meml       ; Зміст слова з ім'ям menu завантажується в DX
mov Al,mem2        ; Зміст байту з ім'ям mem2 завантажується в Al
```

Порівнюючи цей приклад з попереднім, ми бачимо, що вказівка в команді імені елементу пам'яті означає, що операндом є зміст цього осередку; вказівка імені осередку з описувачем offset - що операндом є адреса осередку.

Пряма адресація пам'яті на перший погляд здається простим і наглядним. Якщо ми хочемо звернутися, наприклад, до осередку meml, ми просто указуємо її ім'я в програмі. Насправді, проте, справа йде складніше. Пригадаємо, що адреса будь-якого осередку складається з двох компонентів: сегментної адреси і зсуву. Позначення meml і mem2 в попередньому прикладі, очевидно, є зсувами. Сегментні ж адреси зберігаються в сегментних регістрах. Проте сегментних регістрів чотири: DS, ES, CS і SS. Яким чином процесор дізнається, з якого регістра узяти сегментну адресу, і як повідомити його про це в програмі?

Процесор розрізняє групу кодів, що носять назву префіксів. Є декілька груп префіксів: повторення, розміру адреси, розміру операнду, заміни сегменту. Тут нас цікавитимуть префікси заміни сегменту.

Команди процесора, що звертаються до пам'яті, можуть як перший байт своєї коди містити префікс заміни сегменту, за допомогою якого процесор визначає, з якого сегментного регістра узяти сегментну адресу. Для сегментного регістра ES код префіксу складає 26h, для SS - 361i, для CS - 2eh. Якщо префікс відсутній, сегментна адреса береться з регістра DS (хоча для нього теж передбачений свій префікс).

Якщо на початку програми за допомогою директиви assume вказана відповідність сегменту даних сегментного регістра DS:

```
assume Ds:data
```

то команди звернення до пам'яті транслюються без якого-небудь префіксу, а процесор при виконанні цих команд бере сегментну адресу з регістра DS.

Якщо в директиві assume вказана відповідність сегменту даних регістру ES:

```
assume Es:data
```

(в цьому випадку сегмент даних повинен розташовуватися перед сегментом команд), то команди звернення до полів цього сегменту транслюються з додаванням префіксу заміни для сегменту ES. При цьому пропозиції програми виглядають звичайним способом; у них як і раніше просто указуються імена полів даних, до яких проводиться звернення.

Проте у ряді випадків префікс заміни сегменту повинен указуватися в програмі в явній формі. Така ситуація виникає, наприклад, якщо дані розташовані в сегменті команд, що типово для резидентних обробників переривань. Для звернення до таких даних можна, звичайно, використовувати регістр DS, якщо заздалегідь набудувати його на сегмент команд, але простіше виконати адресацію через регістр CS, який і так вже настроєний належним чином. Якщо в сегменті команд міститься поле даних з ім'ям mem, то команда читання з цього поля виглядатиме таким чином:

```
mov Ax,cs:mem
```

В цьому випадку транслятор додає в код команди префікс заміни для сегменту CS. Інші приклади команд із заміною сегменту будуть приведені нижче. Часто буває потрібно звернутися до пам'яті поза межами програми: до векторів переривань, системних таблиць, відеобуферу і так далі Зрозуміло, таке звернення можливе тільки якщо ми знаємо абсолютну адресу осередку. В цьому випадку необхідно спочатку встановити один з сегментних регістрів на початок області, після чого можна адресуватися до осередків по їх зсувах.

Хай потрібно вивести в лівий верхній кут екрану декілька символів, наприклад, два знаки окликів. Цю операцію можна реалізувати за допомогою наступних команд:

```
mov Ax,0b800h      ; сегментна адреса відеобуфера
mov Es,ax          ; занесемо його в ES
mov byte ptr Es:0,  ' ! ' ; Відправимо символ на 1-е знакомісце екрану
mov byte ptr Es:2,  ' ! ' ; Відправимо символ на 2-е знакомісце екрану
```

У приведеному прикладі використовується атрибутивний оператор byte ptr, який дозволяє в явній формі задати розмір операнду. Проте якщо раніше цей оператор використовувався, щоб витягувати байт з даного, оголошеного, як слово, то тут його призначення інше. Транслятор, обробляючи команду

```
mov byte ptr Es:0,  ' ! '
```

не має можливості визначити розмір операнду-приймача. Зрозуміло, відеобуфер, як і будь-яка пам'ять, складається з байтів, проте чи

треба розглядати цю пам'ять, як послідовність байтів або слів?

Варто ще відзначити, що вказівка в команді описувача word ptr

```
mov word ptr Es:0'!'
```

не викликає помилки трансляції, але приведе до неприємних результатів. В цьому випадку у відеобуфер буде записано слово 002lh, яке заповнить байт 0 відеобуфера кодом 21h, а байт 1 кодом 00h. Проте атрибут 00h позначає чорний колір на чорному фоні, і символ на екрані видний не буде (хоча і буде записаний у відеобуфер). За бажання можна позбавитися від необхідності вводити описувач розміру операнду. Для цього треба пересилати не безпосереднє дане, а вміст регістра:

```
mov AL'!'
mov Es:0,al
```

Тут операндом-джерелом служить регістр AL, розмір якого (1 байт) відомий, і розмір операнду-приймача визначати не треба. Зрозуміло, команда

```
mov Es:0,ax ;заповнить у відеобуфері не байт, а слово.
```

Регістрова непряма (базова і індексна). Адресується пам'ять (байт або слово). Відносна адреса елементу пам'яті знаходиться в регістрі, позначення якого полягає в прямі дужки. При використанні регістрів BX або BP адресацію називають базовою, при використанні регістрів SI або DI - індексною.

```
mov Ax,0b800h ;сегментна адреса
mov Es,ax ; відео буферу  в ES
mov Bx,2000 ; зміщення до середини екрану
mov byte ptr ES:[BX], ' ! ' ;
```

Якщо непряма адресація здійснюється через один з регістрів BX, SI або DI, то мається на увазі сегмент, що адресується через DS, тому при адресації через цей регістр позначення DS: можна опустити:

```
mov Ax,0b800h ; сегментна адреса
mov Ds,ax ;відеобуферу у DS
mov Bx,2000 ; зміщення до середини екрану
mov byte ptr [BX] ' ! ' ;Символ на екран
```

Цей фрагмент ефективніший за попереднє в сенсі витрачання пам'яті. Через відсутність в коді останньої команди префіксу заміни

сегменту він займає на 1 байт менше місця.

Регістри BX, SI і DI в даному застосуванні абсолютно рівнозначні, і з однаковим успіхом можна скористатися будь-яким з них:

```
mov D1,2000 ;Зсув до середини екрану
mov byte ptr [DI] ' ! ' ;Символ на екран
```

Не так йде справа з регістром BP. Цей регістр спеціально призначений для роботи із стеком, і при адресації через цей регістр в режимах непрямої адресації мається на увазі сегмент стека; іншими словами, як сегментний регістр за умовчанням використовується регістр SS. Зазвичай непряма адресація до стека використовується в тих випадках, коли необхідно звернутися до даних, що містяться в стеку, без вилучення їх звідти (наприклад, якщо до ці дані доводиться прочитувати неодноразово).

Порівнюючи приведені вище фрагменти програм, можна відмітити, що використання базової адресації, на перший погляд, знижує ефективність програми, оскільки вимагає додаткової операції - завантаження в базовий регістр необхідної адреси. Дійсно, базова адресація в нашому прикладі не виправдана - у разі прямого звернення до пам'яті замість двох команд:

```
mov Bx,2000 ;зміщення до середини екрану
mov byte ptr ES: [BX], ' ! ' ;Символ на екран
;можна використовувати одну:
mov byte ptr Es:2000, '!' ;
```

Проте команда з базовою адресацією займає менше місця в пам'яті (оскільки в неї не входить адреса осередку) і виконується швидше за команду з прямою адресацією (через те, що команда коротша, процесору вимагається менше часу на її прочитування з пам'яті). Тому базова адресація ефективна в тих випадках, коли за заданою адресою доводиться звертатися багато разів, особливо, в циклі. Виграш виявляється тим більше, чим більше число разів відбувається звернення за вказаною адресою. З іншого боку, можливості цього режиму адресації невеликі, і на практиці частіше використовують складніші способи, які будуть розглянуті нижче.

Регістрова непряма адресація із зсувом (базова і індексна). Адресується пам'ять (байт або слово). Відносна адреса операнду визначається, як сума вмісту регістра BX, BP, SI або DI і вказаної в команді константи, іноді званої зсувом. Зсув може бути числом або адресою. Так само, як і у разі базової адресації, при використанні регістрів BX, SI і DI мається на увазі сегмент, що адресується через DS, а при використанні BP мається на увазі сегмент стека і, відповідно, регістр SS.

Приклад застосування непрямої адресації із зсувом на прикладі прямого виводу у відеобуфер.

```
mov Ax,0b800h ;Сегментна адреса
mov Es,ax ;відеобуферу у ES
mov DI, 80*2*24 ;зсув до нижнього рядка екрану
mov byte ptr ES: [DI],'o' ;Символ на екран
mov byte ptr Es:2 [DI],'к' ;Напишемо символ в наступну позицію
mov byte ptr Es:4[DI] є! ' ;Напишемо символ в наступну позицію
```

В даному прикладі як базовий вибраний регістр DI; у нього заноситься базова відносна адреса пам'яті, в даному випадку зсув у відеобуфері на початок останнього рядка екрану. Модифікація цієї адреси з метою отримати зсув по рядку екрану здійснюється за допомогою констант 2 і 4, які при обчисленні процесором виконавської адреси додаються до вмісту базового регістра DI.

Іноді можна зустрітися з альтернативними позначеннями того ж способу адресації, які допускає асемблер. Замість, наприклад, 4[BX] можна з таким же успіхом написати [Bx+4], 4+[BX] або [BX]+4. Така неоднозначність мови нічого, окрім плутанини, не приносить, проте її треба мати на увазі, оскільки з цими позначеннями можна зіткнутися, наприклад, розглядаючи текст деасембльованої програми.

Приклад використання базової адресації із зсувом при зверненні до стека:

```
;Основна програма
push DS    ;В стек завантажуються значення
push ES            ;трьох регістрів
push SI            ;передаваємих підпрограмі
call mysub         ;Вклик підпрограми mysub
;що використовує ці параметри
;Підпрограма mysub
mov Bp,sp          ;  у BP поточна адреса вершини стека
mov Ax,2[BP]       ;  у AX останній параметр (SI)
mov Bx,4[BP]       ; читаємо у BX попередній параметр (ES)
mov Cx,6[BP];Читаємо у CX перший параметр (DS)
```

Продемонстрований класичний прийом читання вмісту стека без витягання з нього цього вмісту. Після того, як основна, програма зберегла в стеку три параметри, які буде потрібно підпрограмі, командою call викликається підпрограма mysub. Ця команда зберігає в стеку адресу повернення (адреса наступного за call пропозиції основної програми) і здійснює перехід на підпрограму. Стан стека при вході в підпрограму приведений на рисунку 2.1.

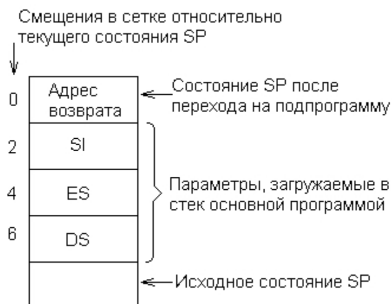


Рисунок 2.1- Стан стека після завантаження в нього трьох параметрів і переходу на підпрограму

Якби підпрограма просто зняла із стека параметри, що знаходяться там, вона насамперед вилучила б із стека адресу повернення, і позбавила б себе можливості повернутися в основну програму. Тому в даному випадку замість команд pop зручніше скористатися командами mov.

Приклад. Хай треба заповнити масив з 10000 слів натуральним рядом чисел. Зарезервуємо в сегменті даних місце під цей масив, а в сегменті команд організовуємо цикл занесення в послідовні слова масиву ряду наростаючих чисел. Нам доведеться скористатися декількома новим. Команди inc, add і loop надалі будуть розглянуті детальніше.

```
;Сегмент даних
array       dw 10000 Dup(?)
;Сегмент команд
mov SI, 0 ; Початкове значення індексу елемента в масиві
mov AX, 0           ; Перше число -заповнювач
mov Cx,10000        ;Число кроків в циклі (завжди в CX)
fill:       mov array[SI],ax ; Занесення числа в елемент масиву
inc AX              ; інкремент числа -заповнювача
add Si,2            ; Зсув у масиві до наступного слова
loop fill           ; Повернення на мітку fill (CX разів)
```

Цикл починається з команди, поміченою міткою fill (правила утворення імен влучний такі ж, як і для імен полів даних). У цій команді вміст AX, спочатку рівне 0, переноситься в елемент пам'яті, адреса якої обчислюється, як сума адреси масиву array і вмісту індексного регістра SI, в якому в першому кроці циклу теж 0. В результаті в перше слово масиву заноситься 0. Далі вміст регістра AX збільшується на 1, вміст регістра SI - на 2 (через те, що масив складається із слів), і командою loop здійснюється перехід на мітку fill, після чого тіло циклу повторюється при нових значеннях регістрів AX і SI. Число кроків в циклі, відлічуване командою

loop, визначається початковим вмістом регістра CX.

Базово-індексна адресація. Адресується пам'ять (байт або слово). Відносна адреса операнду визначається, як сума вмісту наступних пар регістрів:

[BX] [SI] (мається на увазі DS:[BX][SI])
[BX][DI] (мається на увазі DS:[BX][DI])
[BP] [SI] (мається на увазі SS:[BP][SI])
[BP] [DI] (мається на увазі SS:[BP][DI])

Це надзвичайно поширений спосіб адресації, особливо, при роботі з масивами. У нім використовуються два регістри, при цьому одним з них має бути базовий (BX або BP), а іншим - індексний (SI або DI). Як правило, в одному з регістрів знаходиться адреса масиву, а в іншому - індекс в нім, при цьому абсолютно байдуже, в якому що. Трансформуємо попередній приклад, ввівши в нього ефективнішу базово-індексну адресацію.

```
;Сегмент даних
Array        dw 10000 dup(?)
;Сегмент команд
mov Bx,offset array           ;Базова адреса масиву в
;базовому регістрі
mov SI, 0             ;Початкове значення індексу
;елементу в масиві
mov AX, 0                     ; Перше число -заповнювач
mov Cx,10000                  ;Число кроків в циклі
fill:        mov [BX][SI],ax  ; Занесення числа в масив
inc AX                        ; інкремент числа -заповнювача
add SI, 2             ; Зсув у масиві до наступного слова
loop fill                     ; Повернення на мітку fill (CX разів)
```

Підвищення ефективності досягається за рахунок того, що команда занесення числа в елемент масиву виявляється коротшою (оскільки в неї не входить адреса масиву) і виконується швидше, оскільки цю адресу не треба кожного разу прочитувати з пам'яті.

Базово-індексна адресація із зсувом. Адресується пам'ять (байт або слово). Відносна адреса операнду визначається як сума вмісту двох регістрів і зсуву.

Це спосіб адресації є розвитком попереднього. У нім використовуються ті ж пари регістрів, але отриману з їх допомогою результуючу адресу можна ще змістити на значення вказаної в команді константи. Як і у разі базово-індексної адресації, константа може бути індексом (і тоді в одному з регістрів повинна міститися базова адреса

пам'яті), але може бути і базовою адресою. У останньому випадку регістри можуть використовуватися для зберігання складових індексу.

Приклад даного режиму адресації. В сегменті даних визначений масив з 24 байтів, в якому записані коди латинських і російських символів верхнього ряду клавіатури:

```
 sims       db "Qwertyuiop{}'

   db "йцукенпшщзх"
;Послідовність команд:
mov Bx,12           ;Число байтів в рядку
mov SI, 6
mov Dl,syms[BX][SI]
```

;завантажить в регістр DL елемент з індексом 6 з другого ряду, тобто ;код ASCII букви Г.

Той же результат можна отримати, завантаживши в один з регістрів не індекс, а адресу масиву:

```
mov BX, offset sym
mov Si,6
mov DL, 12 [BX] [SI]
```

Режими непрямої адресації пам'яті, що надаються 32-розрядними процесорами

Режими непрямої адресації пам'яті, що надаються 32-розрядними процесорами при використанні 32-розрядних регістрів, зображені на рисунку 2.2.

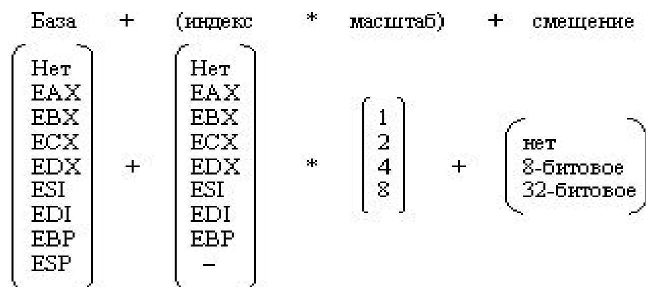


Рисунок 2.2 - Режими непрямої адресації з використанням 32-розрядних регістрів

В якості базового можна використовувати всі регістри загального

призначення, включаючи навіть покажчик стека ESP. При цьому, якщо в якості базового виступає один з регістрів ESP або EBP, то за замовчуванням адресація здійснюється через сегментний регістр SS, хоча можлива заміна сегменту. У всіх інших випадках адресація за умовчанням здійснюється через сегментний регістр DS. Використання регістра EBP як індексний не адреса нас до стека: адресація здійснюється за допомогою регістра **DS.**

Команди завантаження та пересилання даних

Призначені для передачі даних між регістрами та оперативною пам'яттю.

В командах з двома операндами вимагається їх однакова довжина – 1, 2 або 4 байти.

Команда	Призначення	Варіанти використання	Приклади

Mov	Пересилає дані між регістрами, між регістром і пам'яттю, а також передає безпосереднє значення у регістр чи пам'ять.
	Формат команди:
	Mov отримувач, джерело
		Mov r,r
		Mov r,m
		Mov m,r
		Mov r,i
		Mov m,i
			Mov ax, bx
			Mov bx, N

Lea	Завантажує ефективну адресу змінної (зміщення) у регістр. Формат команди: Lea регістр, ідентифікатор Еквівалентна команді: Mov регістр, OFFSET ідентифікатор	Lea r,m	Lea bx, MATRIX
Lds	Завантажує у задану регістрову пару повну логічну адресу (сегмент: зміщення), яка зберігається у пам'яті. Сегментна частина адреси записується у регістр DS, а зміщення – у загальний регістр, який задається першим операндом команди Lds. Формат команди: Lds регістр, адреса_пам'яті		Video DW 0h DW 0B800h Lds bx, DWORD\ PTR Video

Les	Ініціалізує регістр ES та регістр, вказаний першим операндом цієї команди, повною адресою змінної для забезпечення доступу до цієї змінної. Формат команди: Les регістр, адреса_пам'яті		Video DW 0h DW 0B800h Les bx, DWORD\ PTR Video
Xchg	Обмінює значення двох регістрів, або регістра та пам'яті. Не обмінює сегментних регістрів.	Xchg r,r Xchg r,m Xchg m,r	Xchg ax, bx
Xlat	Перетворює байти в інший формат, наприклад нижній регістр у верхній. Для виконання команди необхідно визначити таблицю перетворення байтів та завантажити її адресу у регістр BX. В регістр AL заноситься число, що є зміщенням до потрібного символу у таблиці. За цим зміщенням команда вибирає байт з таблиці і розміщує його у регістр AL. Формат команди: Xlat назва_таблиці	Xlat m	Mov al,10 Lea bx,Table Xlat Table
Push	Заносить слово, задане операндом, у стек. Операнд може бути регістром загального призначення, сегментним регістром або словом у пам'яті.	Push r16 Push m16	Push ax
Pop	Забирає слово зі стеку та записує його у регістр або пам'ять. Регістр може бути загального призначення або сегментним. Формат команди: Pop приймач	Pop r16 Pop m16	Pop ax
Pusha	Заносить у стек усі загальні регістри: AX, CX, DX, BX, SP, BP, SI, DI.	Pusha	Pusha
Popa	Відновлює зі стеку усі загальні регістри: DI, SI, BP, SP, BX, DX, CX, AX.	Popa	Popa

Lahf	Завантажує регістр прапорців Flags у регістр AH, який приймає значення прапорців: SZ-A-P-C (тире позначає незаповнені біти)	Lahf	Lahf
Pushf	Зберігає регістр прапорців Flags у стеку	Pushf	Pushf
Popf	Відновлює регістр прапорців Flags зі стеку	Popf	Popf
In	Вводить із порту байт у регістр AL, або слово у регістр AH. Порт задається цілим числом або значенням регістра DX. Формат команди: In регістр, порт	In AL, port In AX, port In AX, DX	In ax, 60h
Out	Виводить значення байту або слова у порт. Байт задається у регістрі AL, а слово – у регістрі AX. Порт задається цілим числом або значенням регістра DX. Формат команди: In порт, регістр	Out port, AL Out port, AX Out DX, AX	Out 61h, ax

Примітка: r – регістр; m – змінна в оперативній пам’яті; i – безпосереднє значення (константа).

2.3 Порядок виконання завдання

1. Вивчити режими адресування та команди пересилання даних.
2. Написати та набрати програму, що відповідає виданому завданню. Зберегти програму у файлі з розширенням “.ASM ”.
3. Відкомпілювати та відлагодити набрану програму у режимі командного рядка:

   Tasm /zi *.asm

   Tlink /v *.obj

   Примітка: можна скористатися командним файлом tasm.bat
4. Запустити програму для відлагодження td.exe та прочитати результат роботи програми у відповідному регістрі, у стеку або в оперативній пам’яті:

   Td.exe *.exe
5. Перевести результат у 10-ву систему числення.
6. Оформити звіт по роботі.

2.4 Завдання для самостійного виконання

№	Завдання
1.	Оголосити та визначити одновимірний масив з 10 цілих чисел. Записати 3-й елемент масиву у регістр AX, використовуючи режим адресування за базою.
2.	Оголосити та визначити одновимірний масив з 10 цілих чисел. Записати 5-й елемент масиву у регістр BX, використовуючи режим непрямого адресування за допомогою одного із індексних регістрів.
3.	Оголосити та визначити одновимірний масив з 10 цілих чисел. Записати 8-й елемент масиву у стек, використовуючи режим прямого адресування з індексуванням. Переписати елемент зі стеку у змінну оперативної пам'яті X.
4.	Оголосити та визначити одновимірний масив з 10 цілих чисел. Записати його 5-й елемент у змінну Five=5.
5.	Оголосити та визначити одновимірний масив з 10 цілих чисел. Поміняти місцями 3-й та 7-й елементи.
6.	Оголосити та визначити одновимірний масив з 10 цілих чисел. Поміняти місцями 5-й елемент масиву зі значенням змінної TWO=2.
7.	Оголосити та визначити двовимірний масив з 4 x 5 цілих чисел. Записати елемент, що знаходиться на перетині 2-го рядка та 3-го стовпчика, у регістр DX, використовуючи режим адресування за базою з індексуванням.
8.	Оголосити та визначити двовимірний масив з 4 x 5 цілих чисел. Записати елемент, що знаходиться на перетині 3-го рядка та 2-го стовпчика, у змінну оперативної пам'яті X, використовуючи режим прямого адресування за базою з індексуванням.
9.	Оголосити та визначити двовимірний масив з 4 x 5 цілих чисел. Записати елемент, що знаходиться на перетині 3-го рядка та 3-го стовпчика, у стек, використовуючи режим адресування за базою з індексуванням. Переписати елемент зі стеку у регістр BX.
10.	Оголосити та визначити двовимірний масив з 4 x 5 цілих чисел. Поміняти місцями елемент, що знаходяться на перетині 1-го рядка та 4-го стовпчика, з елементом на перетині 3-го рядка та 2-го стовпчика, використовуючи режим прямого адресування за базою з індексуванням.
11.	Оголосити та визначити двовимірний масив з 4 x 5 цілих чисел. Обміняти значення елемента, що знаходиться на перетині 3-го рядка та 4-го стовпчика, зі значенням змінної оперативної пам'яті TEN=10, використовуючи режим адресування за базою з

	індексуванням.
12.	Оголосити та визначити одновимірний масив з 10 цілих чисел. Поміняти місцями другий та п'ятий елементи, використовуючи режим адресування за базою.
13.	Оголосити та визначити двовимірний масив з 4 х 5 цілих чисел. Записати елемент, що знаходиться на перетині 2-го рядка та 4-го стовпчика, у змінну оперативної пам'яті X, використовуючи режим адресування за базою з індексуванням.
14.	Оголосити та визначити одновимірний масив з 10 цілих чисел. Записати його 8-й елемент у змінну X, використовуючи режим прямого адресування з індексуванням.
15.	Оголосити та визначити двовимірний масив з 4 х 5 цілих чисел. Обміняти значення елемента, що знаходиться на перетині 4-го рядка та 2-го стовпчика, зі значенням регістра CX=25, використовуючи режим адресування за базою з індексуванням.

2.5 Контрольні питання

1.Наведіть класифікацію програмно-доступних регістрів.

2.Поясніть спосіб непрямої адресації і його застосування для організації циклічної обробки елементів масиву.

3.Поясніть принцип адресації з використанням індексних регістрів і його застосування для організації циклічної обробки елементів масиву.

4.Дайте характеристику команд пересилання даних.

3 АРИФМЕТИЧНІ КОМАНДИ МОВИ ASSEMBLER ДЛЯ ДВІЙКОВИХ ДАНИХ

Мета роботи: Опанування технологією розробки програм мовою асемблер. Вивчення прийомів виконання елементарних арифметичних розрахунків.

3.1 Методичні вказівки

Щоб успішно впоратись з програмуванням арифметичних виразів слід ретельно вивчити призначення, формати, правила використання наступних команд процесора: MOV, ADD, SUB, INC, DEC, DIV, IDIV, MUL, IMUL, CBW, CWD

Під час практичного засвоєння матеріалу, здобувачі вищої освіти мають ввести наведену програму, виконати її трансляцію, знайти та виправити можливі помилки, а також, дослідити, крок за кроком, її виконання за допомогою налагоджувальної програми. Це дозволить відчути особливості роботи процесора, здобути навички програмування на низькому рівні. Отримані знання та навички слід закріпити шляхом виконання наступних завдань.

3.2 Теоретичні відомості

Лінійні обчислювальні процеси – це ті, що не містять циклічних та розгалужених частин. Навіть складні програми з циклами та розгалуженнями містять лінійні ділянки. Крім того, лінійні програми є найбільш простими. Тому, вивчення програмування на низькому рівні доцільно розпочати саме з них. Найкращім прикладом лінійного обчислювального процесу є програмування обчислень арифметичних виразів.

Формат команд з двома операндами**:**

<Команда> <приймач>, <джерело>

Операнди в командах можуть бути задані різними способами. Можливі наступні способи поєднання операндів:

– "регістр → регістр";

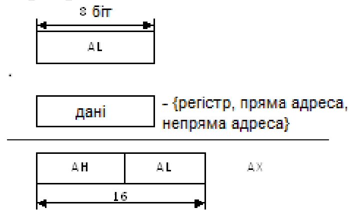
– "регістр → пам’ять";

– "пам’ять → регістр";

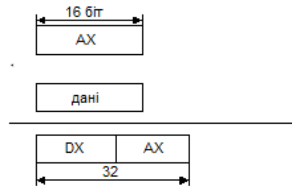
– "безпосередньо задані в команді дані переслати в регістр або пам'ять".

Команди множення і ділення - одноадресні команди і використовують регістр акумулятор для значення одного операнду. Тому, при складанні програми обчислень за формулою, звернути особливу увагу на команди ділення і множення (при множенні - один із співмножників повинен міститися в регістрі акумуляторі: **EAX, Ax** або **AL**; при поділі - ділене повинно міститися в регістри: **EDx, Eax,** або **Dx, Ax** або **Ah, AL**) .

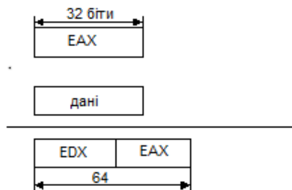
1) множення 8-розрядних даних:



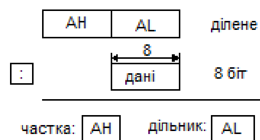
2) множення 16-розрядних даних:



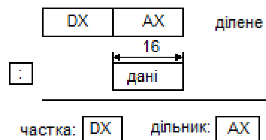
3) множення 32-розрядних даних:



4) ділення 8-розрядних даних:



5) ділення 16-розрядних даних:



6) ділення 32-розрядних даних:

EDX EAX ділене

32

: дані

частка: EDX дільник: EAX

Якщо ділене і дільник одного формату, то при підготовці до поділу необхідно виконати перетворення діленого 8 біт -> 16 біт (якщо фактичне ділене має формат байт) або 16 біт -> 32 біта (якщо фактичне ділене має формат слово), 32 біт -> 64 біти (якщо фактичне ділене має формат подвійне слово). Для цього служать команди перетворення *CBW, CWD або CDQ*.

Формат: CBW ; (ділене в AL) - розширення до (AX).
CWD ; (AX) - розширення до (DX),(AX).
CDQ ; (EAX) - розширення до (EDX),(EAX).

У таблиці 3.3 наведені типові набори команд, які застосовуються при виконанні операцій ділення.

Таблиця 3.1- Набор команд для виконання операції ділення

Формат	**Знакові дані**	**Беззнакові дані**

8біт/8біт	MOV AL,<ділене>
	CBW
	IDIV <дільник>
		MOV AL,<ділене>
		MOV AH,0
		DIV<дільник>

16біт/16 біт	MOV AX,<ділене>
	CWD
	IDIV<дільник>
		MOV AX,<ділене>
		MOV DX,0
		DIV<дільник>

32біта/32 біта	MOV EAX,<ділене>
	CDQ
	IDIV<дільник>
		MOV EAX,<ділене>
		MOV EDX,0
		DIV<дільник>

Більша частина арифметичних команд відносяться до типу:

$$X \# Y \rightarrow X,$$

тобто отриманий результат замінює значення першого операнду команди (# - символ будь-який з двохоперандних команд).

У таблиці 3.2 дано перелік арифметичних команд із зазначенням того, який вид адресації допускається для операндів.

Приклад. Обчислити значення виразу, що визначається формулою:

;Y=$(K+3*k-25)^2/(3*n^2)*2$”

```
;Опис змінних (формат  даних -16 – розрядні, знакові).
data            segment
y               dw ?
k               dw ?
n               dw ?
```

```
tri                     dw 3
dwa                     dw 2
data                    ends
; основна частина програми - сегмент коду
code        segment
                        assume cs: code, ds:data
start:      mov ax, data
            mov ds, ax
            ; ax=k *3
            mov ax,3   ;ax=3
            mul  k     ;ax=3*k
            jo .exit
            add ax,k     ; ax=k+3*k
            sub  ax, 25;ax=k+3*k-25
            mov bx,ax ; bx= k+3*k-25
            ;  n2
            mov ax,n
            mul ax     ;ax=n2
            jo .exit
            mul tri     ;ax=3*n2
            jo .exit
            mul dwa     ;ax=(3*n2)*2
            jo .exit     ;
            xchg ax,bx  ;ax= k+3*k-25   bx=(3*n2)*2
            mov dx,0
            div bx     ;ax=(k+3*k-25)2/(3*n2)*2
            mov y,ax   ;  запам'ятовуємо результат в y
.exit:      mov ax,4c00h ;завершити програму і передати управління
ОС
           int 21h
code    ends
             end  start
```

Таблиця 3.2- Перелік арифметичних команд

Команда	**Функції**	**Використання**			
		Рег.	**Беспос. дан.**	**Прям. адр.**	**Вик. слова**
ADD	Додавання	+	+	+	+
ADC	Додавання з переносом	+	+	+	+
SUB	Віднімання	+	+	+	+
SBB	Віднімання з переносом	+	+	+	+
CMP	Порівняння	+	+	+	+

MUL	Множення без знаку	+	-	+	+
IMUL	Множення цілих із знаком	+	-	+	+
DIV	Ділення без знаку	+	-	+	+
IDIV	Ділення із знаком	+	-	+	+
CBW	Перетворення байту в слово	-	-	-	-
CWD	Перетворення байту в подвійне слово	-	-	-	-
INC	Нарощування на одиницю	+	-	+	+
DEC	Зменшення на одиницю	+	-	+	+
NEG	Зміна знаку	+	-	+	+

3.3 Порядок виконання роботи

1 Вивчити основні команди, які використовуються при організації лінійних процесів на мові Асемблер.

2 Написати та набрати програму, що відповідає виданому завданню. Прокоментувати команди програми. Зберегти програму у файлі з розширенням “*.ASM* ”.

3 Відкомпілювати та відлагодити набрану програму у режимі командного рядка:

```
Tasm /zi *.asm
Tlink /v *.obj
```

Примітка: можна скористатися командним файлом tasm.bat

4 Запустити програму для відлагодження td.exe та прочитати результат роботи програми у відповідному регістрі, у стеку або в оперативній пам’яті:

```
Td.exe *.exe
```

5 Перевести результат у 10-ву систему числення.

6 Оформити звіт по роботі.

3.4 Завдання для самостійного виконання

Розробити програму обчислення виразу, що визначається формулою мовою Assembler. При виконанні операцій ділення залишком знехтувати. Всі змінні - цілого типу.

Таблиця 3.3 - Завдання для самостійного виконання

1	$\frac{x^2-2xy}{3y}\cdot\frac{y}{x^2-4y^2}$	integer
2	$\frac{2p}{x^2-p^2}-\frac{1}{x-p}$	word
3	$\frac{1-q}{x^2-y^2}:\frac{1-q}{x-y}$	integer

4	$\frac{q^2-6q+9}{q-3}$	word
5	$\frac{c-1}{c}\cdot\frac{c}{2c-2}$	integer
6	$\frac{5p^2x-5p^2y}{20p(y-x)}$	word
7	$\frac{ax-bx}{a}\cdot(a-b)^{-1}$	integer
8	$\frac{2x^2-2y^2}{x^2+xc}:\frac{8x-8y}{x+c}$	word
9	$\frac{(x+1)(x^2-1)}{x}\cdot(x+1)^{-2}$	integer
10	$\frac{3a^2-3b^2}{a^2+ac}$	word

3.5 Контрольні запитання

1.Скільки байтів пам'яті зарезервує наступна послідовність операторів:

A DB 3, 4 DUP(3), ?
M1 DW 5 DUP(?),20
B DB 20 DUP(?)

2.Яке значення буде поміщено в змінну:

C DB ?

3.Якими псевдооператор відзначається початок програми?

4.Пояснити на прикладі в чому полягає помилка використання команди CWD для беззнакових змінних.

5.Перелічити способи адресації, пояснити на прикладах.

6.Пояснити сенс псевдокоманди Assume.

7.Пояснити особливості використання регістрів з командами множення та ділення.

8.Що означають числа, які зберігаються у сегментних регістрах?

9.Перелічити та пояснити етапи розробки програми мовою Ассемблер.

10. Перелічити основні регістри процесора та пояснити призначення та особливості використання кожного з них.

11. Пояснити, які комбінації параметрів команди MOV є не припустимими і чому.

4 КОМАНДИ ПОРІВНЯННЯ, УМОВНОГО ТА БЕЗУМОВНОГО ПЕРЕХОДІВ

Мета роботи: Вивчення прийомів використання команд умовних переходів при програмуванні алгоритмів, що розгалужуються.

4.1 Методичні вказівки

З метою практичного засвоєння матеріалу здобувачам вищої освіти слід ретельно вивчити призначення та особливості використання команд CMP, JMP, JZ, JNZ та інших команд умовного переходу, а також регістру прапорців. Під час практичного засвоєння матеріалу, здобувачі вищої освіти мають ввести наведену програму, виконати її трансляцію, знайти та виправити можливі помилки, а також, дослідити, крок за кроком, її виконання за допомогою налагоджувальної програми. Це дозволить відчути особливості роботи процесора, здобути навички програмування на низькому рівні. Отримані знання та навички слід закріпити шляхом виконання наступних завдань.

4.2 Теоретичні відомості

Команди умовного переходу застосовуються для зміни порядку виконання команд у програмі, використовуючи ознаки результату раніше виконаних команд. Ознаки результату формуються автоматично у вигляді значень бітів регістра прапорів як при виконанні арифметичних, логічних команд, так і в результаті виконання команди порівняння та зсувів.

Призначення бітів регістра прапорів наведено в таблиці 4.1.

Таблиця 4.1 - Призначення бітів регістра прапорів

N біта	Умов. позначення	Призначення
0	CF	Перенесення: =1 - якщо було перенесення з старшого біта або при позику "1" в старший біт
2	AF	Допоміжне перенесення: =1 – при перенесенні з молодшої тетради в старшу чи при позику "1" із старшої тетради
4	PF	Парність: =1 - в результаті операції парне число одиниць.
6	ZF	Дорівнює: =1 - результат = 0
7	SF	Знак: =1 - знаковий розряд = 1 (<0
8	TF	Дозвіл / заборона покрокового режиму виконання програми
9	IF	Дозвіл / заборона переривань
10	DF	Напрямок обробки ланцюжків байтів: =1 – зменшення індексу =0 – збільшення індексу.
11	OF	Переповнення

Ознаки результату формуються як при виконанні арифметичних, логічних команд, так і в результаті виконання команди порівняння та зсувів.

Існує відома специфіка застосування команд умовного переходу в залежності від типів порівнюваних даних. У таблиці 4.2 наведено рекомендовані поєднання команд умовного переходу (КУП) з командою порівняння CMP. Якщо використовується команда TEST, то після неї рекомендується застосовувати команди JZ або JNZ, JP або JNP.

Команда порівняння

CMP (операнд-призначення),(операнд-джерело)

Команда CMP (порівняння) віднімає операнд-джерело з операнду-призначення, не змінюючи при цьому значення операндів. Операнди можуть бути байтовими, двобайтовими або чотирьохбайтовими числами. Хоча значення операндів на змінюються, значення прапорів оновлюються, що може бути враховано в наступних командах умовного переходу. Команда CMP впливає на прапори AF, CF, OF, PF, SF і ZF. При збігу значень операндів зводиться прапор ZF. Прапор переносу зводиться, якщо операнд призначення менше операнду-джерела.

Синтаксис команди:

Таблиця 4.2 - Рекомендовані поєднання КУП з командою CMP

Умови переходу			Наступна за CMP команда	
			Порівняння без-знакових даних	Порівняння даних зі знаком
<приймач>	> = <> < <= >=	<джерело>	Ja,jnbe Je,jz Jne,jnz Jb,jnae Jbe,jna Jae,jnb	Jg, jnle Je, jz Jne, jnz Jl, jnge Jle, jng Jge, jnl

```
Data     Segment   ...
X        db ?
Y        db ?
Data     Ends
Code     Segment
         ASSUME Cs:Code, Ds:data
Start:   Mov Ax,Data
Mov Ds,Ax
Mov AL,x
Test AL,3    ; маска для аналізу значень 0,1 розрядів
JZ  M1       ; обидва розряда дорівнюють нулю y=0
JP  M2       ; обидва розряда дорівнюють 1 y:=2
     ;    інакше - один з аналізованих розрядів дорівнює одиниці
y:=1
         mov y, 1
         jmp exit
m2:      mov  y, 2
         jmp exit
m1:      mov  y, 0
exit:    mov  ax,4c00h
         int 21h
Code     ends
         End Start
```

Команди арифметики, логіки, або команд зсуву по результату операцій встановлюють значення прапорів: CF, ZF, SF, OF, PF. Тому після цієї групи команд використовуються команди умовного переходу, перелік яких наведено в таблиці 4.3.

Таблиця 4.3 - Рекомендовані поєднання КУП з командами арифметики, логіки, зсувів

Умова	Команда
Cf=0	Jnc - немає переносу
Cf=1	Jc - є перенос
Zf=1	Je - «=» Jz - «=0»
Sf=0	Jnc - перехід по «+»
Sf=1	Js-перехід по «-»
Of=0 Of<>1	Jno - немає переповнювання Jo - є переповнювання
Pf=0	Jnp - немає парності(к-сть 1 непарна) Jpo - -/-/-/-/-/-/-/-/-/-/-/
Pf=1	Jp - є парність Jpe - -/-/-/-/-/-/-/-/-/-/-/-/

Приклад 3. Розробити програму обчислення виразу, що визначається формулою мовою Assembler.

$$A = \begin{cases} x^2 + 3x + 1, \text{если} \quad a < x < b \\ (x^2 + 3 * x + 1)^2, \text{ если } x = a, \text{или } x = b \\ \dfrac{1}{x^2 - 3 * x + 1}, \text{иначе} \end{cases}$$

```
Data        Segmet
Xdw          ?
Bdw          ?
Adw          ?
.3dw        3
 Data       ends
Code    Segment
   Assume Cs:Code,Ds: data
Start:     Mov ax, data
 Mov ds,ax
 Mov ax, x
 Imul ax
 Jc @exit
 Mov bx, ax
 Mov ax, 3
 Imul x
  Jc @exit
 Add ax, bx
 Add ax,1
 Mov bx, x
 Cmp bx, a
 Jl  .m3
 Je . m2
 Cmp bx, b
 Jg . m3
 Je .m2
 Mov A, ax
 Jmp @exit
.M2:      imul ax
            jc @exit
          mov A, Ax
          jmp @exit
.M3:     Mov bx,1
```

```
              xchg ax, bx
               mov dx, 0
              idiv bx
              mov a, ax
@exit:        mov ax, 4c00h
               int 21h
Code      ends
End Start
```

4.3 Порядок виконання роботи

1.Вивчити призначення та формати команд умовних та безумовного переходів.

2.Написати та набрати програму, що відповідає виданому завданню. Прокоментувати команди програми. Зберегти програму у файлі з розширенням "*.ASM*".

3.Відкомпілювати та відлагодити набрану програму у режимі командного рядка:

```
Tasm /zi *.asm
Tlink /v *.obj
```

Примітка: можна скористатися командним файлом tasm.bat

4.Запустити програму для відлагодження td.exe та прочитати результат роботи програми у відповідному регістрі, у стеку або в оперативній пам'яті:

```
Td.exe *.exe
```

5.Перевести результат у 10-ву систему числення.

6.Оформити звіт по роботі.

4.4 Завдання для самостійного виконання

Обчислити результат, який визначається формулою. При виконанні операцій ділення залишком знехтувати. Для контролю правильності роботи підготувати тестові набори даних.

Варіант	Формула	Тип даних
1	$y=\begin{cases} x^2+3x+4, & если\ \ a<x<b \\ 10x, & если\ \ b<=x<c \\ (x^2+3x+4)^2, & если\ \ c<=x \end{cases}.$	integer
2	$y=\begin{cases} d(1-\frac{b^2}{x^2}), & если\ \ 0<x<b \\ d/(b(x-1)*c, & если\ \ b<=x<20 \\ x-a+c, & если,\ \ x>=20 \end{cases}$	integer

3	$F=\begin{cases}(y\cdot z+1)^2, \text{ если } x<y<z\\ -x(4-z)^2, \text{ иначе}\end{cases}$	integer
4	$y=\begin{cases}d+c\cdot x^2, & k=1\\ d+x^2, & k=2\\ d\cdot e, & k=4\\ a-d, & k=6\\ 0, & \text{â î ñò àëüí û õ ñëó÷àÿõ}\end{cases}$	word
5	$y=(k+\frac{l}{m})(m+1)^2$ $A=\begin{cases}15, & \text{если в} Y \quad 5 \text{ и } 10 \text{раз.}=1\\ 10, & \text{если в} Y \quad 5 \text{ или } 10 \text{раз.}=1\\ 0, & \text{если в} Y \quad 5 \text{ и } 10 \text{раз.}=0\end{cases}$	integer
6	$y=\begin{cases}k*m^2, & \text{если } s1='E', s2='N', s3='D'\\ k\frac{m^2}{2} & \text{если } s1='F', s2='O', s3='R'\\ 0 & \text{,в ост. сл.}\end{cases}$	integer
7	$D=\begin{cases}(y^2\cdot z-a^2+1)^2, \text{ если } a<y<z\\ -a^2x(4-z)^2, \text{ если } a<0 \text{ и } x<0\\ 0, \text{ иначе}\end{cases}$	word
8	$d=\begin{cases}\frac{a^2}{2}(x-2)^2, & \text{если } A>x \text{ и } x>2\\ x^2(a-1), & \text{если } A>1 \text{ и } x<0\end{cases}$	integer
9	$Z=\begin{cases}x^2+2x-5, \text{ если } x<0 \text{ è нечетнь}\\ \frac{x^3}{3}, \text{ если } x>10 \text{ и } x^3 \text{ кратен } 3\\ (x-y)^2+\frac{x+y}{2}, \text{ если } 10<x\le 20\end{cases}$	integer

4.5 Контрольні запитання

1.Можливо чи ні неправильне виконання програми за відсутності узгодження типу порівнюваних даних та виду команди умовного переходу?

2.У яких випадках вигідніше застосовувати команду TEST замість CMP?

3.Навести приклад програми, в якій використовуються команди умовного переходу, в залежності від значень прапорів. Передбачити в

програмі три варіанти, на кожен з яких управління передається по одній з умов: x> 0, x <0, x = 0, команду CMP не використовувати.

4.В чому різниця порівняння знакових чисел і чисел без знаку?

5.Перерахувати прапорці в регістрі Flags та пояснити їх призначення та використання.

5 КОМАНДИ ЛОГІКИ ТА ЗСУВУ. ПОРОЗРЯДНА ОБРОБКА І ФОРМУВАННЯ ДАНИХ

Мета роботи: Опанування технологією роботи з окремими розрядами даних за допомогою порозрядних логічних команд та команд зсувів.

5.1 Методичні вказівки

З метою практичного засвоєння матеріалу здобувачам вищої освіти слід ретельно вивчити призначення та особливості використання команд: AND, TEST, OR, XOR, NOT, BT, SHL, SHR, ROR, ROL, SAR, SAL та їхні можливі модифікації.

Під час практичного заняття, здобувачі вищої освіти мають ввести наведену програму, виконати її трансляцію, знайти та виправити можливі помилки, а також, дослідити, крок за кроком, її виконання за допомогою налагоджувальної програми. Це дозволить відчути особливості роботи процесора, здобути навички програмування на низькому рівні. Отримані знання та навички слід закріпити шляхом виконання наступних завдань.

5.2 Теоретичні відомості

Логічні операції є важливим елементом у проектуванні мікросхем і мають багато загального в логіку програмування. Команди AND, OR, XOR і TEST - є командами логічних операцій. Ці команди використовуються для скидання й установки біт і для арифметичних операцій у коді ASCII. Одне з головних призначень системних програм – взаємодія та керування роботою апаратних пристроїв, що входять до складу обчислювальної системи. Під час роботи системних програм часто виникає необхідність оперувати окремими розрядами інформації. Наприклад, більшість пристроїв, що входять до складу ПК тримають інформацію про свій стан у спеціальному регістрі, в котрому кожен біт є ознакою, що дозволяє діагностувати певну ситуацію. Керування пристроями здійснюється шляхом надсилання їм певних команд – байтів або слів, в котрих кожен з розрядів має бути відповідним чином визначено. Тож, виникає потреба мати доступ (можливість аналізувати, змінювати) до окремих розрядів інформації. Інструментальним засобом виконання операцій над окремими розрядами є порозрядні логічні операції. А методика застосування порозрядних логічних операцій з метою зміни або аналізу окремих бітів зветься накладанням масок (маскуванням).

Головна ідея маскування полягає в наступному. Існує число А (наприклад, байт або слово), в якому потрібно вирізнити значення певного розряду. Для цього, треба таким чином підібрати число В (маску) і побітову логічну операцію, щоб число С, отримане у результаті виконання обраної побітової операції над числами А і В, містило нулі у всіх розрядах, крім того, котрий нас цікавить; він лишається незмінним. В результаті, якщо С дорівнює нулеві, то і розряд, що нас цікавить у числі А є нульовим, і навпаки.

Наприклад, припустимо, що нам потрібно дізнатися значення третього розряду числа A. (Нульовим вважається молодший розряд). Для цього обираємо B=00001000 і використовуємо логічну порозрядну операцію AND.

```
A = XXXXXXXX
                AND
B = 0 0 0 0 1 0 0 0
-----------------------
C = 0 0 0 0 X 0 0 0
```

Як бачимо, число C складається з нулів і лише третій розряд повторює те ж значення, що і в числі A. На Асемблері ті ж самі дії виглядатимуть наступним чином:

```
mov     AX, A
and     AX, 00001000b
jz      m1
```

У наведеному прикладі також показано, як, в залежності від значення досліджуваного розряду числа A, здійснюється (чи не здійснюється) перехід на позначку m1. В разі, коли треба встановити значення певного розряду в одиницю, а значення решти розрядів залишити як було, то це виконується за наступною схемою:

```
A = XXXXXXXX
                OR
B = 0 0 0 0 1 0 0 0
-----------------------
C = XXXX1XXX
```

В разі, коли треба встановити значення певного розряду в нуль, а значення решти розрядів залишити як було, то це виконується за наступною схемою:

```
A = XXXXXXXX
                AND
B = 1 1 1 1 0 1 1 1
-----------------------
C = XXXX0XXX
```

На практиці можливі варіанти коли знадобиться встановити значення не одного, а декількох розрядів. Це робиться так само, як у наведених прикладах, лише маска буде складніша.

Для наступних незв'язаних прикладів, припустимо, що AL містить 1100 0101b, а BH містить 0101 1100b:

```
1. AND AL,BH ;Встановлює в AL 0100 0100b
2. OR BH,AL ;Встановлює в BH 1101 1101b
3. XOR AL,AL ;Встановлює в AL 0000 0000
4. AND AL,00 ;Встановлює в AL 0000 0000b
5. AND AL,0FH ;Встановлює в AL 0000 0101b
6. OR CL,CL ;Установлює прапори SF і ZFb
```

Приклади 3 і 4 демонструють спосіб очищення регістра. У прикладі 5 обнуляються ліві чотири біти регістра AL. Хоча команди порівняння CMP можуть бути зрозуміліше, можна застосувати команду OR для наступних цілей:

```
1. OR CX,CX ;Перевірка CX на нуль
    JZ              ;Перехід, якщо нуль
2. OR CX,CX ;Перевірка знаку в CX
   JS ...    ;Перехід, якщо негативно
```

Команда TEST діє аналогічно команді AND, але встановлює тільки прапори, а операнд не змінюється. Нижче наведемо кілька прикладів:

```
1. TEST BL,11110000B      ;Любий з лівих біт у BL
    JNZ ...                ; дорівнює одиниці?
2. TEST AL,00000001B      ;Регістр AL містить
   JNZ ...                ; непарне значення?
3. TEST DX,OFFH           ;Регістр DX містить
   JZ ...                 ; нульове значення?
```

Ще одна логічна команда NOT встановлює обернене значення біт у чи байті в слові, у чи регістрі в пам'яті: нулі стають одиницями, а одиниці - нулями. Якщо, наприклад, регістр AL містить 1100 0101, то команда NOT AL змінює це значення на 0011 1010. Прапори не міняються. Команда NOT не еквівалентна команді NEG, що змінює значення з позитивного на негативне і навпаки, за допомогою заміни біт на протилежне значення і додатки одиниці.

```
mydata      segment
x:              db          0
y:              dw
mydata          ends
logical         segment
                assume cs: logical,ds:mydata
```

```
andor:   and  ah, 15
        ; за допомогою команди xor можна організовувати
            xor  x, 1; x=1
            xor  x,1 ; x=0;  перемикач  (x)
            and     ax, 255
            and     bx, 1023  ; логічні множення розрядів bx на
; розряди числової константи 1023
; обнуління 5,6 розрядів
            and     byte ptr x, 110011111b
            and     ax, bx
            and     cx, word ptr [di]
            or      ax, 15h         ;установка в 1 0, 2, 4 розрядів
            xor     di, word ptr[si+bx+255]
            not     byte ptr [di+256]  ; інвертувати розряди байту,
;заданого  адресою di+256
            xor  cx,cx                      ; обнулити cx
            not bx                          ; інвертувати розряди bx
logical             ends
                    end andor
```

2. **Приклад.** Обчислити C і, залежно від значень вказаних розрядів присвоїти Y значення відповідного виразу: $C = d(1 - \frac{b^2}{x^2})$

$$Y = \begin{cases} C^2 + 2, & \text{если в C 0 и 15разряды} \langle\rangle 1 \\ C^2, & \text{если в C 0 и 5разряды} \langle\rangle 1 \\ \frac{1}{C^2}, & \text{если в C 5разряд или 0разряд} \langle\rangle 1 \end{cases}$$

```
data            segmet
x     dw     ?
c     dw     ?
d     dw     ?
b     dw    ?
data           ends
code         segment
      assume cs:code, ds: data
start:          mov ax, data
    mov ds,ax
    mox ax,x
    imul ax
    jc  @exit
    mov cx, ax
```

```
     mov ax,b
     imul  ax
     jc @exit
     cwd
     idiv  cx
     neg ax
     inc  ax
      imul d
     jc @exit
      mov c, ax
               mov bx, ax
               imul ax
               jc @exit   ; ax= c2
               and bx, 8001h
               jz v2
               cmp bx,8001h
               jne v3
               inc ax
               inc ax
               mov y,ax
               jmp @exit
v2:            mov y, ax
               jmp @exit
v3:            mov bx ,1
               mov dx,0
               xchg bx, ax
               idiv bx
               mov y,ax
@exit:         mov ax, 4c00h
               int 21h
code           ends
     end start
```

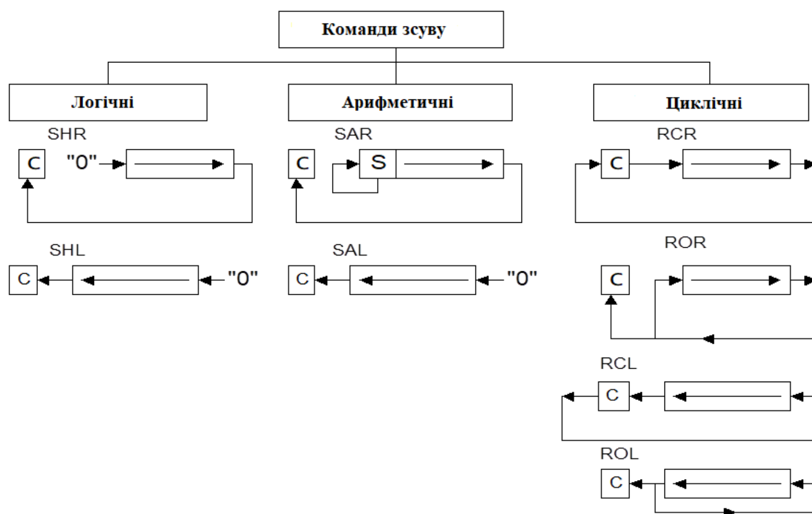
Команди зсуву

Команди зсуву виконують переміщення бітового представлення числа вліво або вправо на задане число бітів. При цьому (якщо не застосовується циклічне зрушення) біти, що виходять за межі розрядів байту (або слова) втрачаються. Місце, що звільнилося заповнюється зазвичай нулями (виняток становить арифметичне зрушення вправо). У виконанні команд зсуву бере участь прапор CF, який зберігає значення біта, що вийшов за межі розрядної сітки біта. На наступній діаграмі дано розподіл команд по групах і відображена схема виконання операції. На діаграмі застосовані такі позначення: C - прапор переносу CF; S - знаковий розряд числа.

Загальний формат команд зсуву:

<код команди> <операнд, що зрушується>, <кількість розрядів>

Результат виконання команди замінює попереднє значення операнду. Кількість розрядів, на яке потрібно зрушити значення операнду задається або безпосереднім значенням, рівним 1, або якщо воно більше 1, то попередньо поміщається в регістр CL, який вказується в команді другим операндом. Особливістю команди арифметичного зсуву вправо є те, що знаковий розряд не зсувається, а його значення використовується для заповнення звільнених розрядів значення.



Приклади різних варіантів запису команд зсуву

```
mydata      segment
x           db   ?
 y          dw  ?
 mydata     ends
 shift         segment
      assume cs:shift,  ds:mydata
begin:      mov ah,5   ; ah=5
            shl ah,1  ; після зсуву ah=10
            mov al,8    ; al=8
            mov  cl,2
            shr  al, cl  ; після зсуву al=2
            sal  bx,1
```

```
                sar  bx,cl
                rol  x,1
                rol  y,cl
                rcl  byte ptr [si],1
                rcr  word ptr [bx+si+255],cl
shift           ends
                end  begin
```

Приклад: Дано 16- розрядне число. Визначити кількість одиничних розрядів в двійковому коді заданого числа.

Розв'язання:

```
data            segment
x               dw   ?        ;   початкове число
n                   db ?            ;   змінна для кількості одиничних
розрядів
data            ends
pr              segment
                assume cs:pr, ds:data
 start:         mov ax, data
                mov ds, ax
                mov    bl, 16     ; кількість повторень
                mov bh.0          ; лічильник ненульових розрядів
                mov    ax, x
cikl:          shr ax,1           ;  зрушити на один розряд
                     jnc    met          ;   останній висунутий розряд
поміщається ;в cf  якщо  ноль  - не рахувати
; розряд = 1, наростити значення лічильника
               inc    bh
 met:                dec     bl            ; зменшити лічильник кількості
повторень
               jnz    cikl
               mov    n, bh       ; запам'ятати результат
               mov    ah,4ch   ;  завершити програму
               mov    al,0
               int    21h
pr             ends
end           start
```

5.3 Порядок виконання роботи

1 Вивчити призначення та формати логічних команд і команд зсуву.

2 Написати та набрати програму, що відповідає виданому завданню. Прокоментувати команди програми. Зберегти програму у файлі з розширенням “*.ASM*”.

3 Відкомпілювати та відлагодити набрану програму у режимі командного рядка:

```
Tasm /zi *.asm
Tlink /v *.obj
```

Примітка: можна скористатися командним файлом tasm.bat

4 Запустити програму для відлагодження td.exe та прочитати результат роботи програми у відповідному регістрі, у стеку або в оперативній пам'яті:

```
Td.exe *.exe
```

5 Перевести результат у 10-ву систему числення.

6 Оформити звіт по роботі.

5.4 Завдання для самостійного виконання

№	Умова	Тип даних
1	Дан x, порядок розрядів замінити на зворотний, результат запам'ятати в y.	Byte
2	Використовуючи логічні операції виконати швидке ділення цілого числа X на 10.	Byte
3	Використовуючи логічні операції та зсуву виконати швидке множення цілого числа X на 10.	Word
4	Дани X, Y. Сформувати Z з 4 та 2 тетрад X і 1 та 3 тетрад Y. Нумерація тетрад справа.	Word
5	Скласти програму, яка змінює порядок розташування на зворотній бітів у тетрадах змінної X (x- 16 біт.)	BYTE
6	Написати програму на мові Assembler, яка підраховує кількість нульових бітів у заданому слові.	Word
7	$y = (k + \frac{l}{m})(m+1)^2$ $A = \begin{cases} 15, & если \quad вY \quad 5 \quad и \quad 10 раз. = 1 \\ 10, & если \quad вY \quad 5 \quad или \quad 10 раз. = 1 \\ 0, & если \quad вY \quad 5 \quad и \quad 10 раз. = 0 \end{cases}$	Word

5.5 Контрольні запитання

1 У чому різниця команд TEST и AND?

2 В якому випадку використовуються регістр CL у командах зсуву?

3 Як змінюється прапор Cf після команд зсуву?

4 Пояснити призначення команд OR, XOR, AND,TEST на прикладах.

5 Наведіть класифікацію команд зсуву. У чому різниця?

6 Чим відрізняється арифметичний зрушення від логічного зсуву.

7 У заданій змінній x встановити в 1 7 і 3 розряди і обнулити 4, 15.

8 Навести кілька варіантів фрагмента коду, в якому б мінялися місцями байти заданого подвійного слова. Дати порівняльну характеристику.

9 Привести фрагмент коду, в якому б в заданій змінній мінялися місцями старша і молодший тетради заданого слова.

10 Пояснити на прикладі команду bt, дати порівняльну характеристику команд bt і test.

11 Пояснити сенс команд RCR, RCL. Наведіть приклад з їх використанням.

12 Наведіть приклади аналізу значення будь-якого розряду за допомогою команд зсуву, команди bt. Дайте порівняльну характеристику.

6 ПРОГРАМУВАННЯ ЦИКЛІВ

Мета роботи: Вивчення прийомів використання команд організації циклів при програмуванні циклічних алгоритмів.

6.1 Методичні вказівки

З метою практичного засвоєння матеріалу здобувачам вищої освіти слід звернути увагу на формат і призначення команд циклів. При використанні цієї групи команд слід пам'ятати, що за регістром CX закріплюється постійна функція - лічильник циклів. При виконанні кожного циклу команда зменшує вміст регістра CX на 1, і виконує перевірку його значення на рівність "0". Якщо CX не дорівнює 0, то передається управління на ближню мітку (наступний цикл), а інакше - виконується наступна по порядку команда (завершення циклу).

Під час практичного заняття, здобувачі вищої освіти мають ввести наведену програму, виконати її трансляцію, знайти та виправити можливі помилки, а також, дослідити, крок за кроком, її виконання за допомогою налагоджувальної програми. Це дозволить відчути особливості роботи процесора, здобути навички програмування на низькому рівні. Отримані знання та навички слід закріпити шляхом виконання наступних завдань.

6.2 Теоретичні відомості

Програми, що входять до складу системного програмного забезпечення (та й прикладні програми також) часто мусять опрацьовувати значні за обсягом об'єми даних, що мають регулярний характер. В таких випадках заведено використовувати масиви різної вимірності, як зручний спосіб зберігання таких даних. Зберігання одноманітних даних у вигляді масивів дозволяє значно спростити їх опрацювання та аналіз і використати для цього алгоритми, що містять цикли та розгалуження. Для організації програмних циклів і розгалужень сучасні процесори мають спеціальні команди: LOOP, CMP, JMP, JZ, JNZ та інші подібні.

Загадані вище команди є універсальними і можуть з успіхом використовуватися для роботи з масивами довільної вимірності. Зазначимо лише, що оскільки пам'ять класичних ПК має лінійну одновимірну структуру, то роботу з багатовимірними масивами програми можуть лише моделювати. Мови високого рівня роблять це прозоро (непомітно для нас), а, під час програмування на низькому рівні, моделювання багатовимірних масивів доводиться виконувати самостійно. Тобто, рядки матриць розташовуються у пам'яті послідовно один за одним і щоб звернутися до певного елемента масиву з використанням декількох індексів, слід мати процедуру обчислення лінійного індексу.

Втім, найчастіше, під час програмування доводиться мати справу з одновимірними масивами (ланцюжками) даних.

Основна команда цієї групи LOOP - "повторювати поки лічильник не дорівнює 0".

Формат:	Сенс:
LOOP <ближня позначка>	CX:=CX-1
	if CX<>0 then goto<ближня позначка> else<наступна команда>

Інші команди цієї групи включають додаткові умови припинення циклу:

Формат:	Сенс:
LOOPE <ближня позначка> **LOOPZ** **LOOPNE** <ближня позначка> **LOOPNZ**	CX:=CX-1 if CX<>0 &&ZF=1 then goto <ближня позначка>

Структура фрагмента програми, що використовує оператор циклу:

```
        ....
        Mov CX,<кількість повторень циклу>
        ....
<позначка початку циклу>:
            <послідовність команд, що виконується в циклі>
        LOOP <позначка початку циклу>
```

Приклад. Обчислити ступінь числа $y=x^n$. Результат може бути 64 – розрядним.

```
data        segment
x           db  ?       ;   початкове число
n           db  ?       ;   початкове число
y      dd ?,?      ;   результат
data        ends
pr          segment
            assume cs:pr, ds:data
start:      mov ax, data
            mov ds, ax
    mov cx,n
    mov al, x
    cbw
    cwde;   початковий байт в eax
    mov ebx,eax
cikl:       imul ebx
```

```
            jo stop; результат перевищує  64 розряду
            loop cikl
stop:       cmp cx,1; це було останнє множення?
            ja perepoln; ні
            mov y,eax; так
            mov y+4,edx
            mov   ah,4ch   ;  завершити програму
            mov   al,0
            int   21h
pr          ends
end         start
```

6.3 Порядок виконання роботи

1. Вивчити команди організації циклів на мові Intel Assembler.

2. Написати та набрати програму, що відповідає виданому завданню. Прокоментувати команди програми. Зберегти програму у файлі з розширенням “*.ASM*”.

3. Відкомпілювати та відлагодити набрану програму у режимі командного рядка:

```
Tasm /zi *.asm
Tlink /v *.obj
```

Примітка: можна скористатися командним файлом tasm.bat.

4. Запустити програму для відлагодження td.exe та прочитати результат роботи програми у відповідному регістрі, у стеку або в оперативній пам’яті:

```
Td.exe *.exe
```

Перевести результат у 10-ву систему числення.

5. Оформити звіт по роботі.

6.4 Завдання для самостійного виконання

№	Завдання
1.	Обчислити суму парних чисел з діапазону [21, 99].
2.	Обчислити факторіал числа n=5.
3.	Знайти суму десяткових цифр натурального числа n=369.
4.	Обчислити кількість цифр у десятковому записі натурального числа n.
5.	Знайти добуток цифр у десятковому записі натурального числа n.
6.	Обчислити кількість цілих чисел з діапазону [-125, 348], у десятковому записі яких є цифра 3.
7.	Обчислити кількість цілих чисел з діапазону [40, 280], які діляться на 3 та на 5.

8.	Перевірити, чи задане натуральне число n є простим.
9.	Обчислити суму дільників натурального числа n.
10.	Обчислити кількість цілих чисел з діапазону [-12, 245], у десятковому записі яких є цифри 2 та 4.
11.	Обчислити суму непарних чисел з діапазону [23, 125].
12.	Змінній t присвоїти значення 1 або 0 в залежності від того, чи натуральне число k є степенем числа 3.
13.	Знайти кількість простих чисел з діапазону [2, n], n>2.
14.	Визначити, чи задане натуральне число n є досконалим, тобто чи воно дорівнює сумі усіх своїх додатних дільників, крім самого себе.
15.	Обчислити K - кількість точок з цілочисельними координатами, які потрапляють в коло радіуса R>0 з центром у початку координат.
16.	Обчислити k-й член (k>0) послідовності $xn = nxn-1 + 1/n$, $n = 1,2,3,...$, $x0 = 1$.

6.5 Контрольні запитання

1. Яким чином зберігаються багатовимірні масиви інформації у оперативній пам’яті ЕОМ, якщо остання має лінійну структуру?

2. Навести групу команд, що виконують дії тотожні до дій команди LOOP.

3. Пояснити роль регістра CX під час організації циклічних алгоритмів.

4. Яким чином можна організувати роботу вкладених циклів з параметром?

5. Наведіть приклад з використанням команд LOOPE і LOOPNE.

7 РОЗРОБКА ПРОГРАМ СТВОРЕННЯ ТА ОБРОБКИ МАСИВІВ

Мета роботи: оволодіння технологією обробки масивів даних і програмування циклічних алгоритмів на мові асемблера.

7.1 Методичні вказівки

З метою практичного засвоєння матеріалу здобувачам вищої освіти слід вивчити правила опису масивів даних, зрозуміти як нумеруються (індексуються) його елементи, як залежить адреса елемента від індексу цього елемента. Повторити матеріал про режими адресації, формати та привила використання команд роботи зі стеком, команд циклу.

7.2 Теоретичні відомості

Обробка таблиць (масивів) є складовою частиною кожної системної програми, включаючи компілятори, завантажувачі, файлові системи та операційні системи. Наприклад, асемблери та компілятори з алгоритмічних мов використовують таблиці символічних імен та літералів, таблиці машинних операцій і т.п. Від правильного вибору алгоритму роботи з таблицями залежить, в кінцевому рахунку, показники ефективності роботи системної програми. Тому системний програміст повинен володіти базовими алгоритмами роботи з таблицями та вміти оцінювати ефективність цих алгоритмів в залежності від технічних вимог, що пред'являються до програми.

Елементи таблиць можуть мати різну структуру. Елементами масивів можуть бути дані типу char (db), integer (dw), dword (dd). Цикли, що дозволяють виконати деяку ділянку програми багаторазово, в будь-якій мові є однією з найбільш уживаних конструкцій. В системі команд мікропроцесора (МП) цикли реалізуються, головним чином, за допомогою команди loop (петля), хоча є й інші способи організації циклів. У всіх випадках число кроків в циклі визначається вмістом регістра CX, тому максимальне число кроків складає 64 К. У наступних версіях процесорів, лічильник циклу - 32-розрядний заноситься в ECX.

При описі масиву вказується кількість елементів у ньому та їх тип:

x dw 30 dup(?),

проте не вказується як нумеруються (індексуються) його елементи.

Описанню може відповідати масив:

x [0..29]
x [1..30]
x [k..29+k]

Якщо нумерація строго не визначена, тоді краще почати нумерацію з нуля.

Як залежить адреса елемента від індексу цього елемента:

x dw 30 dup(?); x [k..29+k] =>
адреси (x[i])=x+2*(i-k)
адреси (x[i])=x+(type x)*(i-k)
при k =0
адреси (x[i])=x+(type x)*i

Тому зазвичай вважають, що елементи в асемблері нумеруються з 0:

x dw 30 dup(?); x [0..29]

Для багатомірних масивів ситуація аналогічна:

A [N * M]
N,M - const
N - кількість рядків
M - кількість стовбців

Припустимо, що елементи матриці розміщені в пам'яті по рядках (можна розміщувати й по стовбцям):

адреси (A [i,j])=A+M*(type A)*(i-k*M)+(type A)*(i-k*N)

Найбільш доступний вигляд ця залежність приймає при нумерації з 0.

Як здійснюється доступ до елементів масиву?

До цього моменту ми працювали з командами, у яких для операндів вказувалися їх точні адреси.

Приклад 1. Пошук в масиві першого від початку ненульового значення.

Розв'язання:

```
  Data segment
  mas dw 100 dup (?)   ; початковий масив з 100 елементів
  n db    ?            ; змінна для номера ненульового елемента
  n_zero dw ?
  data ends
  pr segment
              assume cs: pr, ds:data
 start: sub    bx, bx    ; номер елемента масиву (починаючи з нуля)
              mov   cx, 100        ; кількість повторень
 cikl:        mov   ax, mas [bx]
              cmp    ax, 0  ; порівняти черговий елемент з нулем
              jne    met   ; якщо не нуль - вийти з циклу
              inc    bx
              inc    bx
              loop cikl
; запам'ятати номер першого ненульового елемента
  met:        mov  n, bl
; запам'ятати значення ненульового елемента
              mov    n_zero, ax
```

```
                mov  ah, 4ch        ; завершити програму
                mov  al, 0
                int    21h
pr              ends
                end   start
```

;Другий варіант. Розв'язання з використанням непрямої адресації:

```
data segment
mas  dw 100 dup(?)          ; початковий масив з 100 елементів
off  dw    ?    ; змінна для адреси  ненульового елемента
n_zero dw  ?        ; для значення ненулевого елементу масиву
data                ends
pr                  segment
        assume cs:pr, ds:data........
start:              lea   bx,,mas        ; в  bx  занести адресу масиву
                    mov   cx,100
                    dec   bx
                    dec  bx
cikl:               inc   bx
                    inc bx
                    mov ax, [bx]
                    cmp   ax, 0
                    loope cikl
; по команді loop цикл переривається, коли буде
; встановлений прапор нуля (коли операнди рівні) або cx =0
met:                mov  off,  bx    ;     зберегти  адресу  ненульового
елемента
                    mov n_zero, ax
                    mov    ah,4ch
                    mov    al,0
                    int    21h
pr                  ends
                    end    start
```

Приклад 2. Для матриці A(10,10) знайти середнє арифметичне парних чисел, які розташовано на головній діагоналі.

Розв'язання:

```
.386         ; будемо використовувати команду 386 процесора  bt
data                segment use16
; матриця 7*7
a dw        100       dup       (?)
s dw        ?         ; середнє арифметичне
```

```
data                ends
code                segment use16     ; 16-розрядний додаток
          assume     cs: code, ds:data
start:              mov       ax, data
          mov       ds, ax
          mov       di,0        ; лічильних парних
          mov       ax,0        ; для суми парних
          mov       cx,10       ; для циклу по елементах діагоналі
          mov       bx,0        ; номер байту першого елемента
                           ; поточного рядка
          mov       si,0        ; номер байту, що визначає черговий
                           ; стовпчик
  ; ax - сума парних чисел головної діагоналі
cikl1:
          mov       dx, a[bx+si]
          bt        dx, 0       ; перевіримо значення нульового
                           ;розряду
          jc        next        ; cf=1 - непарне число
          add       ax, dx              ; додаємо парне до суми
          inc       di          ; рахуємо кількість парних
  ; перехід до наступного елемента по діагоналі
  ; в кожному рядку 10 елементів (20 байт)
next:
          add       bx, 20
          inc       si          ; номер байту наступного стовпчика в
          inc       si          ; рядку
  ; виконати цикл для всіх елементів діагоналі
          loop      cikl1
          cwd       ; знаходимо середнє арифметичне
          idiv      di
          mov       s, ax       ; запам'ятовуємо
          mov ax, 4c00h
          int 21h
code      ends
          end       start
```

Приклад 3. Для заданої матриці K (5,7) знайти суму елементів у кожному рядку. Отриману суму записати в одновимірний масив.

```
data      segment
k         dw 5 dup(7 dup(?))            ; опис матриці
mas2      dw 5 dup(?)         ; опис одновимірного масиву
data      ends
zadan2    segment
```

```
            assume cs:zadan2, ds:data
start:      mov ax, data
          mov ds, ax
            mov cx,5                          ; 5  рядків
            mov bx,offset k    ; адреса першого рядка
; адреса результуючого масиву
            mov di,offset mas2
cikl1:    push cx    ; зберігаємо лічильник зовнішнього ;циклу
; ініціалізіруємо лічильник внутрішнього циклу - по стовпцях
            mov cx,7
   ; номер байту чергового елемента в рядку
            mov si,0
  mov ax,0; для суми в кожному рядку
; додаємо черговий елемент рядка
cikl0:      add ax,bx[si]
; перехід до наступного елементу в рядку
            inc si
            inc si
   ; виконати цикл для всіх елементів рядка
            loop cikl0
            mov [di],ax  ; занести суму в одновимірний ;масив
  ; просунути адресу для наступного результату
            inc di
            inc di
; встановити адресу на наступний рядок
            add bx,si
            pop cx              ; витягаємо cx з стека
            loop cikl1         ; і виконуємо цикл для всіх ;рядків
            mov ax, 4c00h
            int 21h
zadan2      ends
            end  start
```

Приклад 4. Розрахунок суми елементів масиву.

```
prog        segment
            assume  cs:prog, ds:prog
_array   dw  10 dup(?)
sum      dw  ?
start:
           mov     ax,cs
            mov     ds,ax
            xor     ax,ax
            mov     bx,ax
```

```
            mov    cx,10
      next:
            add    ax, _array [bx]
            add    bx,2
            loop   next
            mov    sum,ax
            ret
     prog   ends
     end    start
```

Проте команда може працювати й з виконавчою (ефективною) адресою:

Авик = (A+[рег]) mod 2^16, де [рег] - зміст регістра

Спершу ніж виконати команду, ЦП додає до адреси А (вказаній у команді) поточний зміст регістра - отримає нову адресу й буде працювати з операндом за цією адресою.

Команда LEA. При використанні регістрів-модифікаторів часто доводиться записувати в них ті чи інші адреси. Нехай, наприклад, нам необхідно занести в регістр ВХ адресі змінної Х:

```
X DW 88
```

Щоб зробити це, можна завести ще одну змінну, значенням якої буде адреса Х:

```
y dw x
```

а потім передати значення цієї змінної в регістр ВХ:

```
mov bx,y
```

Алі зрозуміло, що це дещо штучний спосіб завантаження адреси в регістр. Тому в систему команд ПК уведена спеціальна команда такого завантаження:

Завантаження виконуючої адреси (load effective address): LEA r16,A

Ця команда розраховує виконуючу адресу іншого операнду й заносить її в регістр r16: r16:=Авик. Прапорці команда не змінює. У якості першого операнду може бути вказаний будь-який регістр загального призначення, а в якості іншого - будь-який адресний вираз (з модифікаторами чи без них).

Одним з прикладів, де корисна команда LEA, є вивід рядка по операції OUTSTR. Ця операція, нагадаємо, потребує, щоб початкова адреса рядка, що буде виведена знаходилася в регістрі DX. Це доцільно робити по команді LEA:

```
s dв 'a+ь=c','$'
…
lea dx,s ; dx:=адреси S
Mov ah,9 ; буде виведено: a+ь=c
Int 21h
```

Розглянемо особливості команди LEA. При цьому будемо вважати,

що в програмі є наступні описання:

q dw -45
r dw -8

По-перше відмітимо, що команда LEA дуже схожа на команду MOV, але між ними є й принципова відмінність: якщо LEA заносить в регістр саму адресу, вказану в команді, то MOV заносить вміст комірки з цією адресою:

lea bx,q; bx:=адреси q
mov bx,q; bx:=вміст q (-45)

У команді LEA другий операнд може бути будь-якою адресою - і адресою байту, і адресою слова й т.п. Проте в якості цього операнду неможна вказати константний вираз чи ім'я регістра:

lea cx.88 ; помилка lea cx,bx ; помилка

Якщо в якості іншого операнду вказано модифікатор адреси, то спочатку розраховується виконуюча адреса й лише потім відбувається завантаження в регістр: mov si,2

lea ax,q[si]; ax:=Авик-q+[si]=адреси(q+2)=адреси(R)

Цим можна скористатися для передачі в який-небудь регістр значення регістра-модифікатора, збільшеного чи зменшеного на деяке число:

```
mov bx,50
lea cx,[bx+2]   ;cx:=[bx]+2-50+2=52
lea di,[di-3]   ;di:=di-3 (sub di,3)
```

7.3 Порядок виконання роботи

1. Вивчити команди організації циклів та способи адресування даних для роботи з масивами цілих чисел.
2. Написати та набрати програму, що відповідає виданому завданню. Прокоментувати команди програми. Зберегти програму у файлі з розширенням “*.ASM*”.
3. Відкомпілювати та відлагодити набрану програму у режимі командного рядка:

Tasm /zi *.asm
Tlink /v *.obj

Примітка: можна скористатися командним файлом tasm.bat.

4. Запустити програму для відлагодження td.exe та прочитати результат роботи програми у відповідному регістрі, у стеку або в оперативній пам’яті:

Td.exe *.exe

Перевести результат у 10-ву систему числення.

5. Оформити звіт по роботі.

7.4 Завдання для самостійного виконання

№	Завдання
1	Дана матриця A [10,5]. Знайти суму елементів парних рядків.
2	Дана матриця A [8,8]. Знайти мінімальний елемент верхнього трикутника.
3	Дана матриця A [7,7]. Знайти мінімальний елемент над головною діагоналлю.
4	Дана матриця A [5,5.] Знайти максимальний елемент над побічною діагоналлю.
5	Дана матриця A [3,7]. Знайти номери рядків елементів A[i,j]=C.
6	Дана матриця A [10]. Знайти номер мінімального елемента.
7	Дана матриця A [6,6]. Знайти мінімальний елемент під головною діагоналлю.
8	Дана матриця A [5,5]. Знайти максимальний елемент під побічною діагоналлю.
9	Дана матриця A [3,7]. Знайти номери рядків елементів A[i,j]=A[1,1].
10	Дана матриця A [8,8]. Знайти максимальний елемент верхнього трикутника й визначити номер стовпця, у якому він знаходиться.
11	Дана матриця A [5,5]. Знайти максимальний і мінімальний елементи .
12	Дана матриця A [7,20]. Для заданої матриці знайти суму парних елементів у кожному рядку. Отриману суму записати в одновимірний масив.
13	Дана матриця A [3,7]. Знайти номери рядків елементів A[i,j]=min.
14	Дана матриця A [5,3]. Знайти номер рядка, у якому знаходиться мінімальний елемент.
15	Дана матриця A [6,6]. Знайти мінімальний елемент у непарних стовбцях.
16	Дана матриця A [5,5]. Знайти максимальний елемент серед елементів < A[2,3], якщо таких елементів немає - вивести повідомлення.
17	Дана матриця A [10,5]. Знайти суму елементів непарних рядків, які більше 2.
18	Дана матриця A [8,8]. Для заданої матриці знайти мінімальний елемент у кожному стовпчику. Результат записати в одновимірний масив.
19	Дана матриця A [8,8]. Знайти мінімальний елемент над головною діагоналлю й номер рядка, у якому він знаходиться.
20	Дана матриця A [6,6]. Знайти максимальний елемент над

	побічною діагоналлю й замінити його на min.

7.5 Контрольні запитання

1. Як нумеруються елементи масиву?
2. Як залежить адреса елемента масиву від індексу цього елемента?
3. Як здійснюється доступ до елементів масиву?

ПЕРЕЛІК ЛІТЕРАТУРИ

1.Таненбаум Э. Архітектура комп'ютера. – СПб.: Питер, 2002. – 704 с.

2.Юров В.И. Assembler: Підручник для вузів. 2-е вид. – СПб.: Пітер, 2004. – 637 с.

3.Юров В.И. Assembler: Практикум. – СПб.: Пітер, 2003. – 400 с.

4.Зубков С.В. Assembler для DOS, Windows и UNIX. 3-е вид. – М.: ДМК Прес; СПб.: Пітер, 2004. – 608 с.

5.Столлінгс, Вільям, Операційні системи, 4-е видання.: Пер. з англ. - М.: Видавничий будинок “Вільямс”, 2002. -848с.: мул. – Парал. тит. Англ.

6.Солдатов В. П. Програмування драйверів Windows. Вид. 2-е, перераб. і доп. – М.: ООО “Біном – Прес”, 2004г.-480с.: мул.

7.Рязанцев О.І., Ларгіна А. М. Організація обчислювальних процесів в комп’ютерних системах: Навчальний посібник.- Луганськ: Вид-во СНУ ім В. Даля , 2006. –608с.

Навчальне видання

МЕТОДИЧНІ ВКАЗІВКИ
до практичних занять з дисципліни
«СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ»
(для здобувачів вищої освіти 3 курсу денної та заочної форми навчання за спеціальністю 123 "Комп'ютерна інженерія")
Частина ІІ

Укладач:
М.В. ДЕРКАЧ

Оригінал-макет *М.В. Деркач*

Підписано до друку __________
Формат 60х84 1/16. Папір типогр. Гарнітура Times.
Друк офсетний. Умов. друк. арк. ___. Обл.-вид. арк.___.
Тираж ___ екз. Вид. № ___. Замов. №   . Ціна договірна.

Видавництво Східноукраїнського національного університету імені Володимира Даля

Свідоцтво про реєстрацію: серія __ № ____ від ________ р.
Адреса університету: просп. Центральний 59-А
м. Сєвєродонецьк, 93400, Україна
e-mail: vidavnictvoSNU.ua@gmail.com