С. В. Зибін

Асемблер для x86 і Pentium

Методичні вказівки та завдання
до виконання лабораторних робіт
з дисципліни "Системне програмування"

Київ – 2011

Міністерство освіти і науки України
Державний університет інформаційно-комунікаційних технологій

С.В. Зибін

Асемблер для x86 і Pentium

Методичні вказівки та завдання
до виконання лабораторних робіт
з дисципліни "Системне програмування"

спеціальності 7.091501 – "Комп’ютерні системи та мережі"

Київ – 2011

УДК 681.3.066
ББК 32.973-018.2
Г-60

Затверджено на засіданні вченої ради Навчально-наукового інституту телекомунікацій та інформатизації (протокол № ___ від "____" _____ 20__ р.)

Рецензент: доктор технічних наук, професор Самофалов Ю.Я.

Асемблер для x86 і Pentium: методичний посібник до лабораторних робіт з дисципліни "Системне програмування" / Укладач: Зибін С.В., - К.: ДУІКТ, 2011, 58 с.

Методичний посібник містить основні теоретичні відомості, вказівки та завдання до лабораторних робіт за дисципліною "Системне програмування". Присвячені вивченню особливостей програмування мовою асемблера (мікропроцесор ix86, Pentium).

Містять завдання, вказівки по підготовці до лабораторних робіт, вимоги до звіту.

Призначений для студентів денної форми навчання спеціальності 7.091501 – "Комп’ютерні системи та мережі".

ЗМІСТ

ВСТУП

Мова асемблера? Хіба це актуально? Хіба зараз, в час візуального програмування та інтелектуальних технологій побудови коду є сенс застосовувати мову асемблера в прикладному програмуванні?

Такі питання часто виникають не лише у студентів-початківців, але й у практикуючих програмістів, і, навіть у деяких викладачів програмування.

Адже дійсно, сучасне програмування активно прямує до позбавлення програміста рутинної необхідності написання коду. Розробляються і впроваджуються спеціальні потужні пакети візуального проектування, в яких робота програміста зводиться до графічного зображення схеми роботи системи, а перехід від них до будь-якої мови високого рівня є простим і безболісним.

Згадаємо, однак, про те, що якою б мовою не була реалізована програма, в решті решт вона перетворюється в послідовність машинних кодів за допомогою того чи іншого компілятора.

Найпростіший аналіз коду, згенерованого штучним компілятором, виявляє багато не оптимізованих ділянок, які, за певних умов, можуть суттєво знизити продуктивність обчислювального процесу. Саме тому навіть найновітніші компілятори не можуть застосовувати розширення сучасних процесорів при трансляції програм. І саме тому будь-який сучасний компілятор підтримує режим асемблерної вставки, що дозволяє програмістові у відповідальні моменти "брати справи у власні руки".

Для того, щоб приступити до вивчення мови асемблера, необхідно мати хоч би мінімум попередніх знань.

- Уміти працювати з числами в різних системах числення (двійковою, десятковою, шістнадцятковою). Уміти працювати з числами на рівні біт, оскільки без цього не можна зрозуміти, як працюють логічні операції та операції зміщення, як працює процесор з від'ємними числами.
- Потрібні деякі навички програмування, оскільки мова асемблера занадто деталізована, щоб починати програмувати з неї.

Мати навички роботи в середовищі операційних систем MS DOS і Windows – знання команд, структури файлової системи і т. і. Мати хоч би деяке уявлення про роботу із строковими компіляторами і про командний рядок.

Лабораторна робота № 1

ЕТАПИ ВИКОНАННЯ ПРОГРАМИ МОВОЮ АСЕМБЛЕРА

Мета роботи: вивчити процеси трансляції, компоновки та відлагодження програми, навчитись використовувати програми для створення об’єктних, виконуваних файлів і відлагодження асемблерного коду .

Теоретичні відомості

Процес створення програми мовою асемблера містить в собі наступні етапи (рис. 1.1):

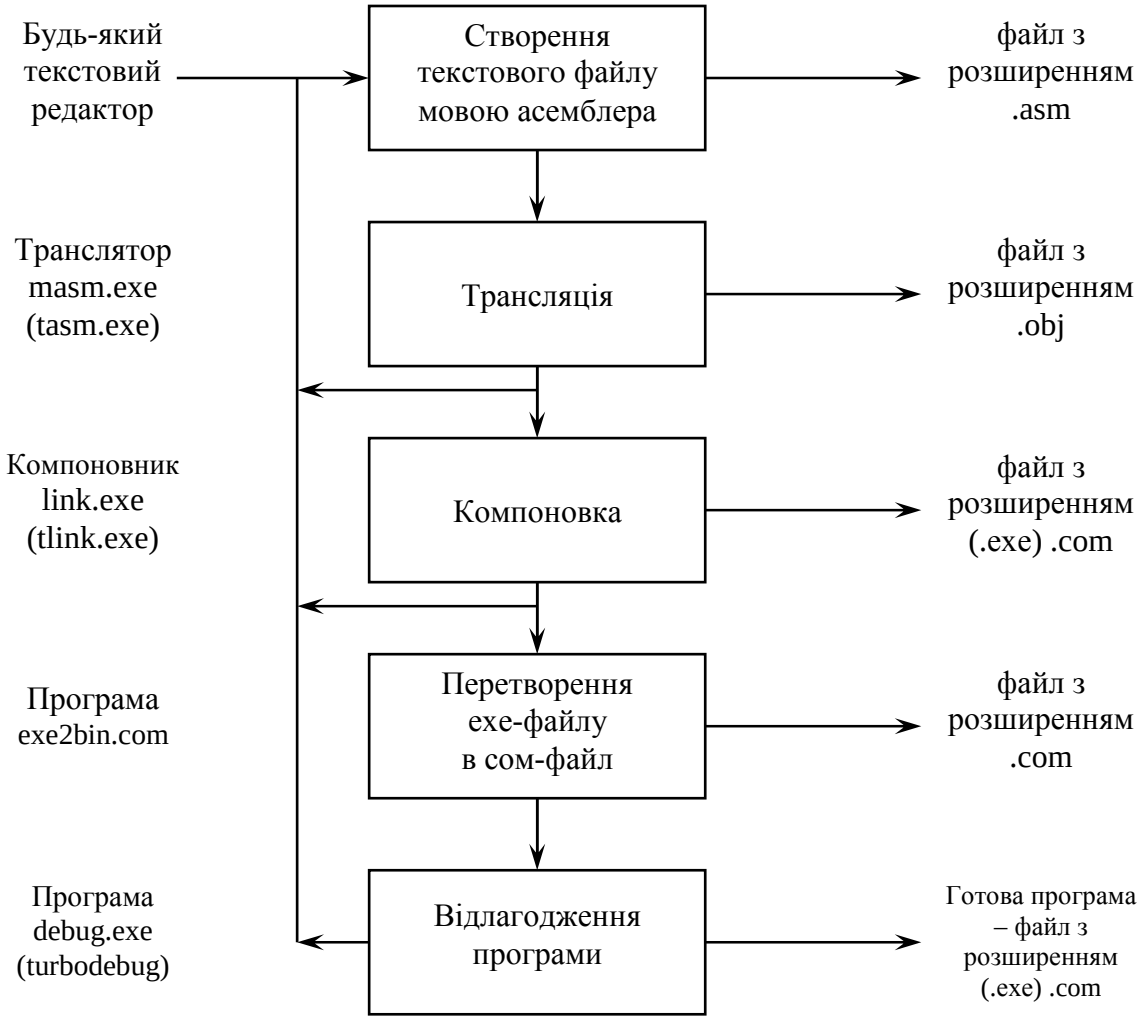


Рис. 1.1. Етапи розробки програми мовою асемблера

Першим етапом розробки програми є створення текстового файлу мовою асемблера (в подальшому – **ВИХІДНИЙ** текстовий файл). Цей етап виконується за допомогою будь-якого текстового редактору, який не зберігає керуючих символів. Для файлів, які створені за допомогою мови асемблера прийнято розширення .asm.

На другому етапі необхідно виконати **ТРАНСЛЯЦІЮ** текстового файлу мовою асемблера в об’єктний код. В процесі трансляції текст програми

перетворюється в машинний код, не прив’язаний (на цьому етапі) ще конкретним фізичним адресам. Об’єктний код програми після трансляції буде розміщено в OBJ-файл (файл з розширенням .obj). Для трансляції використовується програма-ТРАНСЛЯТОР masm.exe або tasm.exe. Знайдіть цю програму та запустіть її на виконання. На екрані з’явиться

source_filename [.asm]
object_filename [filename.obj]
source_listing [nul.lst]
cross-reference [nul.crf]

Курсор при цьому розміщується наприкінці першого рядку, де необхідно вказати ім'я файлу. Введіть ім'я вихідного текстового файлу. Розширенням за умовчанням є .asm. Якщо вихідний файл не знаходиться в тій ж теці, що й програма masm.exe (tasm.exe), то необхідно при написанні імені вихідного файлу вказати повний шлях до цього файлу.

В другому запиті додається таке ж ім'я файлу (за бажанням його можна змінити) з розширенням за умовчанням .obj. Якщо obj-файл знаходиться в іншій теці, необхідно явно вказати повний шлях та ім'я цього файлу.

Методичні вказівки

Завданням даної роботи є набуття навичок використання командного рядка і застосування програм для побудови об’єктних і виконуваних файлів, а також відлагодження виконуваних файлів.

Приклад виконання роботи

1. В будь-якому текстовому редакторі створити вихідний файл, який містить наступні рядки програми.

Наприклад: lab1.asm.

```
sgcode segment
assume      cs:sgcode, ds:sgcode, ss:sgcode, es:sgcode
org 100h
begin:
    jmp label1
    text    db    "It will be probably my first program. $"
label1:
    mov     ax, cs          ; рядки потрібний для правильної
    mov     ds, ax          ; роботи exe-програми
    mov     cx, 25
    lea     dx, text
    mov     ah, 09
    int     21h
    mov     ah, 4ch
    int     21h
sgcode ends
end begin
```

2. За допомогою транслятора tasm.exe створити об’єктний файл lab1.obj.

tasm.exe lab1.asm ,,,

Якщо транслятор знайшов помилки при створенні об’єктного файлу, їх необхідно усунути та повторити процедуру (п. 2). Перелік помилок, рядок виникнення і їх код відображується на екрані. Для того щоб побачити помилки тексті вихідного файлу необхідно створити файл з розширенням lab1.lst, який розміщується в тій ж самій теці що й вихідний файл.

3. Створити виконуваний файл за допомогою компоновника.

tlink lab1.obj

Варіанти завдань

1. Створити вихідний файл мовою асемблера, ім'я файлу повинно бути у вигляді lab1_FIO.asm, де FIO – прізвище, ім'я, по-батькові.
2. Створити об’єктний файл, файл листінг.
3. Усунути недоліки в вихідному файлі.
4. Створити виконуваний файл.
5. Виконати виконуваний файл.

Лабораторна робота № 2

АРХІТЕКТУРА ПРОЦЕСОРІВ СІМЕЙСТВА ix86

Мета роботи: вивчити архітектуру процесорів сімейства ix86 і закріпити отримані знання.

Теоретичні відомості

Формати команд

Мікропроцесор ix86 відноситься до класу однокристальних з фіксованою системою команд. Розглянемо програмну модель мікропроцесору, яка містить функціональні вузли (регістри). Загальні регістри розбито на дві групи: 1) група HL, яка складається з регістрів AX, BX, CX, DX, котрі використовуються для зберігання даних; 2) група PI, яка містить вказівні регістри BP, SP та індексні регістри SI, DI, в яких зберігається адресна інформація. Для загальних та сегментних регістрів вказано коди, які використовуються в форматах команд для їх адресації. Регістр прапорів F та вказівник команд IP адресуються в командах неявно.

Загальні регістри

		15		8	7		0	
HL	000	7	AH (100)	0	7	AL (000)	0	AX
	001	7	CH (101)	0	7	CL (001)	0	CX
	010	7	DH (110)	0	7	DL (010)	0	DX
	011	7	BH (111)	0	7	BL (011)	0	BX

		15	0
PI	100	SP	
	101	BP	
	110	SI	
	111	DI	

Регістр прапорів

11	10	9	8	7	6	4	2	0	
OF	DF	IF	TF	SF	ZF	AF	PF	CF	F

Вказівник команд

15	0
IP	

		15	0
Сегментні регістри	00	ES	
	01	CS	
	10	SS	
	11	DS	

Команди мікропроцесору можуть адресувати один чи два операнди, двооперандові команди симетричні, оскільки результат операції може бути спрямовано на місце будь-якого із операндів. Проте в подібних командах один із операндів повен обов’язково розташовуватись в регістрі, оскільки є команди типу регістр – регістр, регістр – пам'ять, пам'ять – регістр, але команди типу пам'ять – пам'ять відсутні. В загальному вигляді формат двоопернднної команди наведено на рис. 2.1,а, де штриховими лініями позначені необов’язкові байти команди. Перший байт команди містить код операції COP й два однобітових поля: напрямку d та слова w. При d=1 здійснюється передача операнда або результату операції в регістр, який визначається полем reg другого байту команди; при d=0 – передача із вказаного регістру. Поле w ідентифікує тип (розрядність) операндів: при w=1 команда оперує словом, при w=0 – байтом.

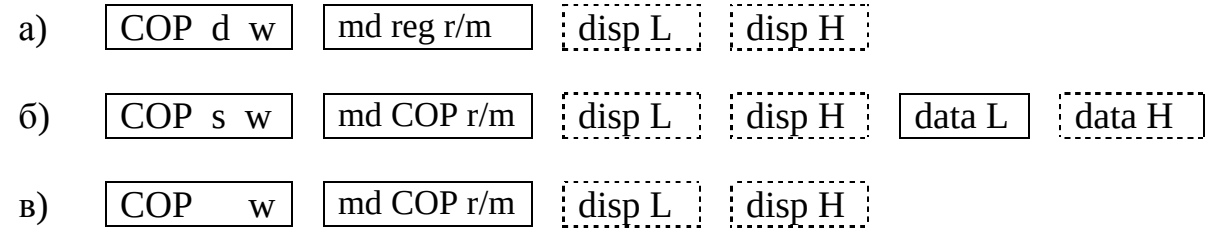


Рис. 2.1. Формати двооперандних команд

Другий байт називається постбайтом. Він визначає регістри, або регістр та комірку пам'яті, які беруть участь в операції. Постбайт складається з трьох полів: md – режим, reg – регістр, r/m – регістр/пам'ять. Поле reg визначає операнд, який обов’язково знаходиться в регістрі мікропроцесору й умовно вважається другим операндом. Поле r/m визначає операнд, який може знаходитись в регістрі або пам'яті й умовно вважається першим. Спосіб кодування внутрішніх регістрів мікропроцесору в полях reg й r/m наведено в табл. 2.1.

Таблиця 2.1

reg, r/m	w=0	w=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Поле reg використовується для вказівки регістру тільки в двооперандових командах. Якщо в команді один операнд, то він ідентифікується полем r/m, а поле reg використовується для розширення коду операції.

Поле md показує, як інтерпретуються поле r/m для розташування першого операнду: якщо md=11, то операнд розміщено в регістрі, в решті випадках – в пам'яті. Коли адресується пам'ять, поле md визначає варіант використання зсуву disp, який знаходиться в третьому й четвертому байтах команди:

$$md = \begin{cases} 00,\ disp = 0 \text{ - зсув відсутній;} \\ 01,\ disp = disp\ L \text{ - команда містить 8 - бітовий зсув, який розширюється з знаком до 16 біт;} \\ 10,\ disp = disp\ H, disp\ L \text{ - команда містить 16 - бітовий зсув.} \end{cases}$$

При md=11 реалізується непряма адресація пам'яті й поле r/m визначає правила формування ефективної адреси EA операнду у відповідності с табл.2.2, де disp означає зсув, який заданий в форматі команди.

Таблиця 2.2

Поле r/m	Ефективна адреса EA	Адресація
000	BX+SI+disp	Базово-індексна
001	BX+DI+ disp	
010	BP+SI+ disp	
011	BX+DI+ disp	
100	SI+ disp	Індексна
101	DI+ disp	
110	BP+ disp	Базова
111	BX+ disp	

Наведені в табл. 2.3 правила мають одне виключення, яке дозволяє реалізувати пряму (абсолютну) адресацію: якщо md=00 й r/m=110, то EA=disp H. disp L. Таким чином, існує три варіанти інтерпретації поля md й вісім варіантів інтерпретації поля r/m, що дає 24 варіанти обрахування ефективної адреси EA. Сумарні дані о постбайтових режимах адресації наведено в табл. 2.3 та рис. 2.2.

Таблиця 2.3

Поле r/m	Поле md				
	00	01	10	11	
				W=0	W=1
000	BX+SI	BX+SI+D8	BX+SI+D16	AL	AX
001	BX+DI	BX+DI+D8	BX+DI+D16	CL	CX
010	BP+SI	BP+SI+D8	BP+SI+D16	DL	DX
011	BP+DI	BP+DI+D8	BP+DI+D16	BL	BX
100	SI	SI+D8	SI+D16	AH	SP
101	DI	DI+D8	DI+D16	CH	BP
110	D16	D16+D8	D16+D16	DH	SI
111	BX	BX+D8	BX+D16	BH	DI

Примітка: D8=disp L (однобайтовий зсув); D16=disp H disp L (двобайтовий зсув).

Найбільш загальний формат двооперандової команди з безпосереднім операндом наведено на рис. 2.1,б. Необхідність адресації другого операнду відсутня, і поле reg використовується для розширення коду операції. Відсутній також біт напрямку d, тому що результат операції можна розмістити тільки на місце першого операнду. Поля s та w інтерпретуються наступним чином:

$$sw = \begin{cases} x0, \text{один байт даних data L;} \\ 01, \text{два байту даних data H, data L;} \\ 11, \text{один байт даних, який автоматично розширюється зі знаком до 16 біт.} \end{cases}$$

На рис. 2.1,в наведено типовий формат однооперандової команди.

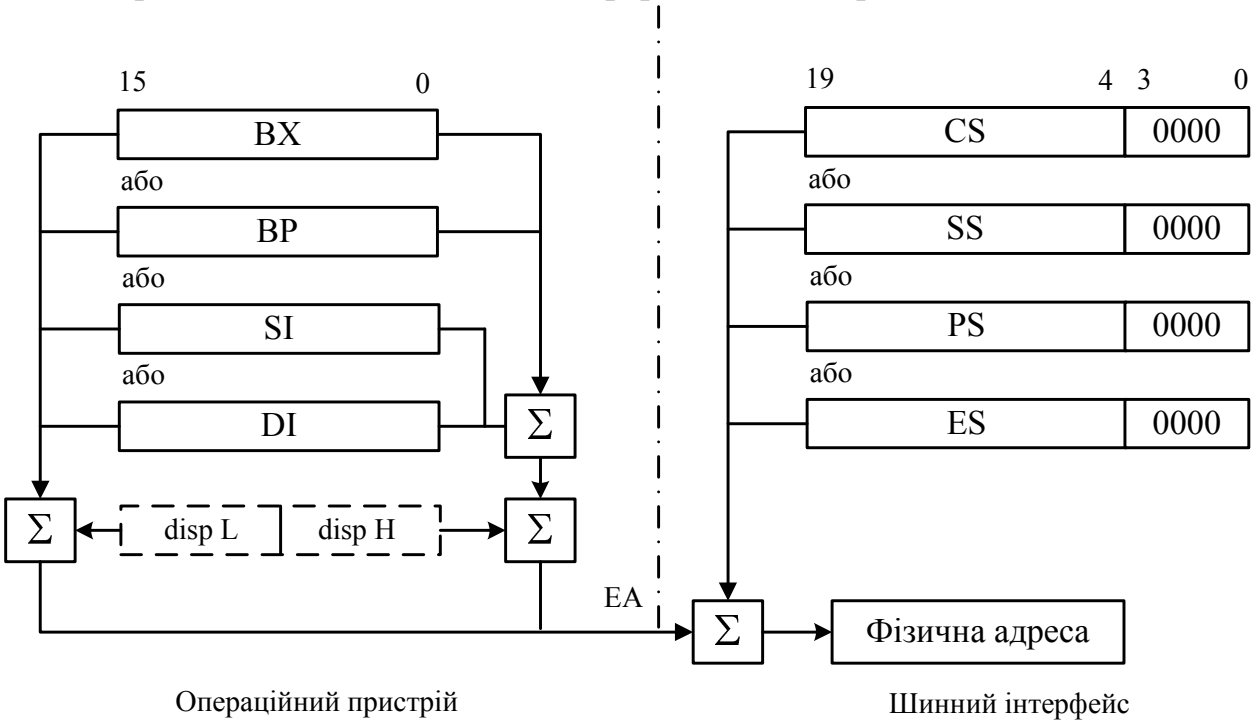


Рис. 2.2. Обчислення фізичної адреси пам'яті

Способи адресації

Команди мікропроцесору ix86 реалізують різноманітні способи адресації.

Регістрова адресація. Операнд знаходиться в одному з загальних регістрів або в одному із сегментних регістрів. Регістр може бути визначено в байті коду операції або в пост байті, в якому виділені 3-бітові поля reg та r/m (при md=11). Команди, які оперують вмістом регістрів, являються найбільш короткими та виконуються за найкоротшій час.

Безпосередня адресація. Безпосередніми операндами є константи довжиною 8 або 16 біт, які розташовуються в останніх байтах команди (молодший байт слідує першим).

Пряма адресація є простішим способом адресації комірки пристрою, що запам’ятовує, при якому ефективною адресою EA є вміст байтів зсуву disp команди. В командах перетворення даних цей спосіб реалізується при використанні пост байта з полями md=00 та r/m=110.

Непряма регістрова адресація. В командах перетворення даних ефективна адреса ЕА дорівнюється вмісту одного з регістрів SI, DI, BX, BP при відповідному кодуванні полів md та reg пост байту: md=00, r/m=100, 101, 111 та md=01, r/m=110; disp L=0. В командах безумовного переходу та виклику підпрограми з регістрової непрямою адресацією допускається вказівка будь-якого 16-бітового регістру (при md=11; r/m=000, …, 111).

Базова адресація. Ефективна адреса операнда ЕА обчислюється шляхом підсумовування вмісту базових регістрів BX або BP та зсуву disp (при md=01,10 та r/m=111, 110). При використанні BX відбувається звернення до операнду в поточному сегменті даних, а при використанні BP – в поточному сегменті стеку. Зсуви, які містяться в команді, можуть мати довжину 8 або 16 біт та інтерпретуються як знакові цілі, які представляються в додатковому коді.

Індексна адресація. Значення ЕА обчислюється як сума зміщення disp, який знаходиться в команді, та вмісту індексного регістру SI або DI (md=01, 10 та r/m=100, 101). Цей спосіб використовується для звертання до різних елементів одновимірного масиву (таблиці) даних, коли зсув визначає відомий при асемблируванні початкову адресу масиву, а індексний регістр, вміст якого може модифікуватися при виконанні програми, визначає елемент масиву.

Базова індексна адресація. Ефективна адреса ЕА дорівнюється сумі вмісту базового регістру BX або BP, індексного регістру SI або DI та зміщення disp, який знаходиться в команді. Цей спосіб реалізується при наступному кодуванні полів постбайту: md≠11; r/m=000, 001, 010, 011 та забезпечує найбільшу гнучкість адресації, оскільки два компоненти адреси можна визначити й варіювати при виконанні програми. Це зручно при звертанні до елементів матриць.

Відносна адресація реалізується тільки по відношенню до вказівника команд IP, сегментний зсув обчислюється як сума зсуву disp, який знаходиться в команді, та поточного значення IP. При цьому значення IP дорівнюється адресі байту, наступного за командою, яка в цей час виконується. Відносна адресація не використовується в командах, які оперують даними, а використовується тільки в командах умовних та безумовних переходів, викликів підпрограм та керування циклами.

Неявна адресація. Об’єкт, вмістом якого маніпулює команда, вказує за допомогою першого байту команди разом з кодом операції без виділення для цієї мети спеціального коду. Частіше всього цей спосіб адресації зустрічається в однобайтових командах, де об’єктом, який адресується, являється акумулятор, регістр прапорів або окремі прапорці.

Системи команд

За функціональною ознакою система команд розбивається на шість груп:

1) пересилка даних;
2) арифметичні операції;
3) логічні операції та зсуви;
4) передача керування;

5) обробка ланцюгів;
6) керування мікропроцесором.

Особливості виконання деяких команд

Команди пересилки даних складають чотири підгрупи: загальні, звернення до стеку, ввід/вивід та пересилка ланцюгів. Ці команди, за виключенням POPF та SAHF, не впливають на прапорці.

100010 d w	md reg r/m	disp L	disp H	MOV r_1/m, r_2/m
				MOV r_1/m, r_2/m
1100011 w	md 000 r/m	disp L	disp H	data L data H
1011 w reg	data L	data H		MOV r,d

0 – MOV ac,m; 1 – MOV m,ac

101000 X w	disp L	disp H

0 – MOV r/m,sr; 1 – MOV sr,r/m

100011 X 0	md 0 sr r/m	disp L	disp H

Команда MOV здійснює пересилку вмісту джерела scr в одержувач dst та має узагальнене представлення: MOV dst, src.

Команда MOV r_1/m, r_2/m містить постбайт та забезпечує пересилки регістр – регістр/пам'ять та пам'ять – регістр при використанні будь-якого загального регістру та будь якого способу адресації пам'яті. Біт w визначає передачу байта або слова, а біт d – напрямок передачі. Команда MOV r/m,d дозволяє передавати безпосередні дані в загальний регістр або комірку пам'яті.

Команди MOV ac,m та MOV m,ac призначені для загрузки та запам’ятовування вмісту акумуляторів AL та AX при використанні прямої адресації.

Команди MOV sr,r/m та MOV r/m,sr здійснюють пересилки між сегментним регістром та регістром або пам’яттю. При цьому передаються тільки слова, а комірка пам'яті може бути визначена за допомогою будь-якого дозволеного способу адресації. Слід враховувати, що в команді MOV sr,r/m неможна вказувати сегментний регістр коду CS, тому що при цьому результат операції не визначено.

Команди LEA, LDS, LES відрізняються від інших команд пересилки тим, що при їх виконанні в регістр, в який адресується, передаються не дані, а їх адреси.

Команди передачі керування використовуються в циклічних програмах та програмах, які розгалужуються, а також при виклику підпрограм та поверненню із них.

Сегментна організація програмної пам'яті визначає два основних типу команд передачі керування: внутрішньосегментні NEAR (близькі) та міжсегментні FAR (далекі). При виконанні команд типу NEAR модифікується тільки регістр IP й адреса переходів представляється одним словом або навіть

байтом, якщо використовується короткий варіант переходу з обмеженим діапазоном адрес. При виконанні команд типу FAR змінюється вміст регістрів IP та CS й адреса переходу представляється двома словами (сегмент:зміщення), що дозволяє перейти в будь-яку точку адресного простору пам'яті.

В цю групу входять команди безумовних переходів, викликів, повернень, умовних переходів, керування циклами та переривань.

Команди безумовних переходів JMP (рис. 2.3) виконують модифікацію регістру IP або регістрів IP та CS без зберігання попередніх значень цих регістрів.

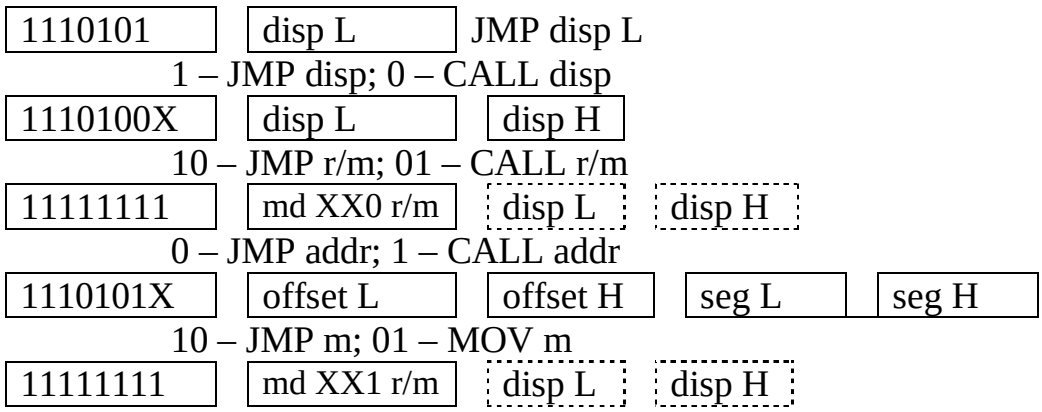


Рис. 2.3. Формати команд JMP та CALL

Існує три формату (рис. 2.4) команди JMP типу NEAR, які здійснюють перехід в межах поточного кодового сегменту (внутрішньосегмнтний, близький перехід) та два формату команди JMP типу FAR, які здійснюють перехід в будь-яку точку адресного простору (між сегментний, далекий перехід).

Двобайтова команда **JMP disp L** в другому байті містить зсув, який інтерпретується як знакове ціле. Цей зсув додається (з попереднім розширенням знаку до 16 біт) до вмісту IP, яке відповідає адресі команди, яка знаходиться після даної команди JMP.

Трьохбайтова команда **JMP disp** виконує таку ж дію як і попередня команда, але містить 16-бітовий зсув disp (асемблерна вказівка NEAR PTR).

Перші два формати команди JMP реалізують відносний спосіб адресації, який забезпечує позиційну незалежність програм. Наступні три формати визначають абсолютні адреси переходів та загружають регістр IP (при міжсегментних переходах й регістр CS) новими значеннями, які не залежать від попередніх.

Команда **JMP r/m** здійснює непрямий внутрішньосегментний перехід (асемблерний покажчик WORD PTR), при якому в регістр IP завантажується вміст регістру або комірки пам'яті у відповідності з постбайтовим режимом адресації.

Команда **JMP m** здійснює непрямий міжсегментний перехід (асемблерний покажчик DWORD PTR) через вміст двох комірок пам'яті: слово з комірки, яке адресується за допомогою пост байту, завантажується в IP, а слово з наступної комірки – в регістр CS.

Команда **JMP addr** здійснює прямий міжсегментний перехід (асемблерний покажчик FAR PTR) й містить 4 байти адреси переходу: два байти сегментного зсуву offset завантажуються в регістр IP, а два байти sr – в регістр CS.

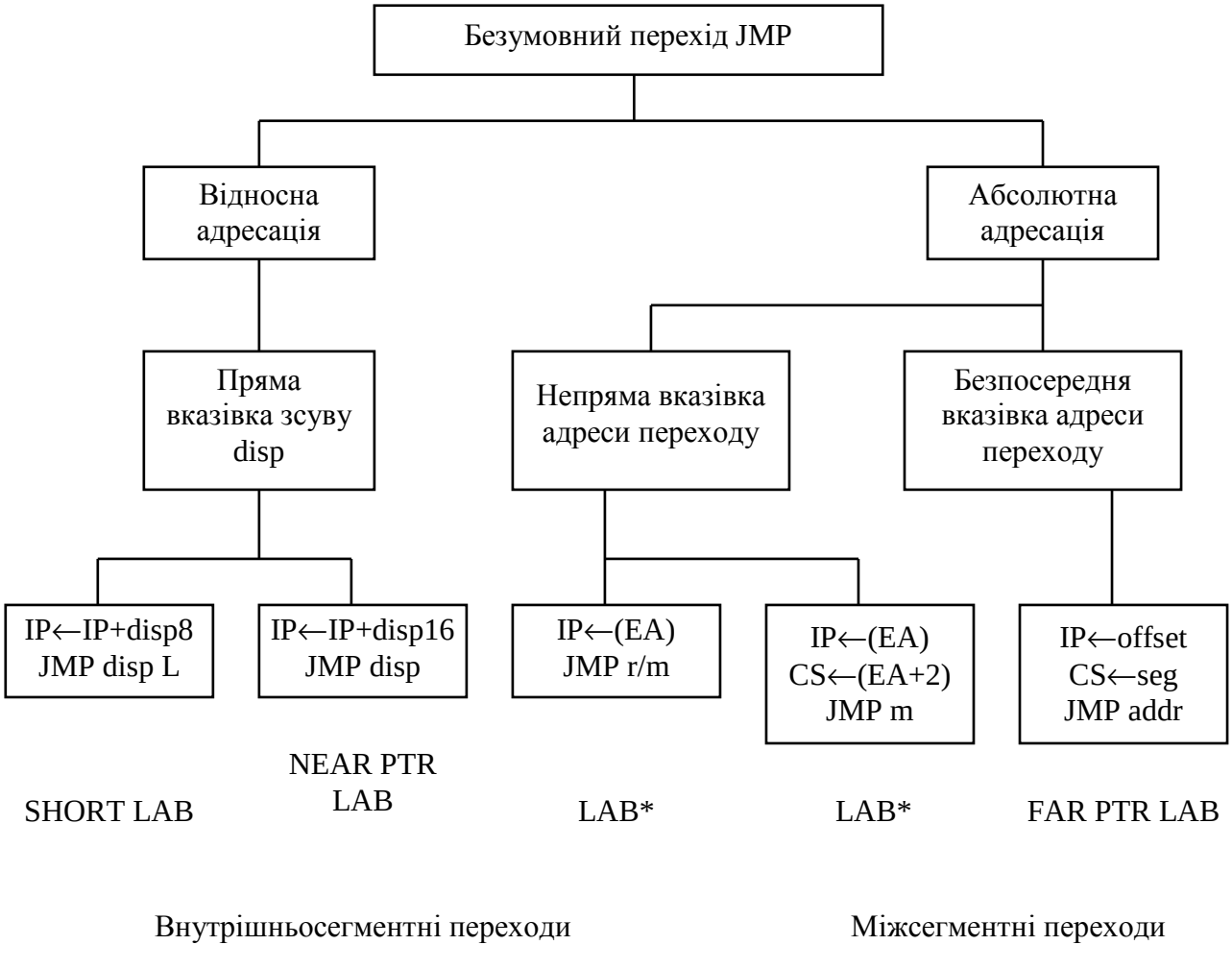


Рис. 2.4. Варіанти формування адрес переходів командою JMP

Команди виклику підпрограми CALL (рис. 2.3) мають такі ж формати, як й команди JMP (крім команди короткого переходу) та виконуються аналогічним способом, за виключенням того, що автоматично запам’ятовується адреса повернення (тобто адреса команди, наступної за командою CALL). З цією метою при внутрішньосегментних викликах в стеці запам’ятовується вміст IP, а при міжсегментних викликах – спочатку вміст IP, потім CS.

Команди повернення (із підпрограм) RET (рис. 2.5). Кожна підпрограма повинна містити хоча б одну команду повернення RET, яка повертає керування програмі, яка здійснила виклик.

Подібна передача керування здійснюється шляхом витягання із стеку адреси повернення, яка включена в нього командою виклику підпрограми. Тому команди повернення не містять ніякої адресної інформації й неявно адресують вершину стеку. Тип команди повернення (внутрішньосегментний

NEAR або міжсегментний FAR) вибирається у відповідності з типом команди CALL.

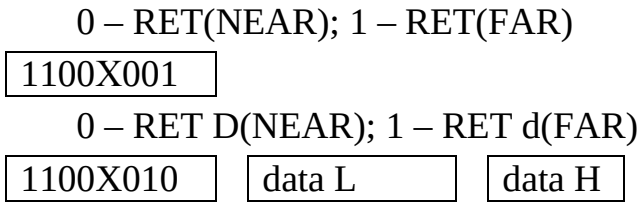


Рис. 2.5. Команди повернення із підпрограм

Методичні вказівки

Питання для перевірки

1. Визначити поняття "програмна модель".
2. Призначення елементів в програмній моделі AX, AH, AL, BX, DX.
3. Призначення елементів SP, SI, DI, BP.
4. В яких випадках використовуються регістри CS, SS, DS?
5. Призначення постбайту в команді (рис. 2,а).
6. Призначення полів d, w ,s.
7. Призначення полів md, reg, r/m.
8. Призначення поля disp.
9. Визначити поняття базової адресації.
10. Визначити поняття базової індексної адресації.
11. Визначити поняття відносної адресації.
12. Визначити поняття прямої адресації.
13. Визначити поняття регістрової адресації.
14. Визначити поняття непрямої регістрової адресації.
15. Визначити поняття безпосередньої адресації.
16. Визначити поняття неявної адресації.
17. Визначити поняття індексної адресації.
18. Коли можлива пряма адресація?
19. Сегментація пам’яті.
20. Визначити поняття логічної адреси.
21. Визначити поняття фізичної адресації.
22. Обчислення логічної адреси.
23. Призначення команди MOV, її модифікації.
24. Призначення команд CALL, RET.
25. Призначення команди JMP – ближній перехід.
26. Призначення команди JMP – дальній перехід.
27. Призначення команд умовного переходу.
28. Різниця між ближнім й дальнім переходами.

Лабораторна робота № 3

ОСОБЛИВОСТІ АРХІТЕКТУРИ ПРОЦЕСОРІВ INTEL PENTIUM 6

Мета роботи: вивчити особливості архітектури процесорів Intel Pentium і закріпити одержані знання.

Теоретичні відомості

Процесори сімейства Intel Pentium 6 мають ряд архітектурних та структурних особливостей в порівнянні з попередніми моделями.

Вони мають наступні характеристики:

- 32-розрядна внутрішня структура;
- використання системної шини з 36 розрядами адреси та 64 розрядами даних;
- роздільна внутрішня кеш-пам'ять 1-го рівня (L1) для команд та даних ємкістю по 16 Кбайт;
- підтримка загальної кеш-пам'яті команд та даних 2-го рівня (L2) ємкістю до 2 Мбайт;
- конвеєрне виконання команд з реалізацією 12 ступенів конвеєру;
- передбачення напрямку програмного розгалуження з високою точністю;
- прискорене виконання операцій з плаваючою крапкою;
- пріоритетне керування при звертанні до пам'яті (захищений режим);
- підтримка реалізації мультипроцесорних систем;
- наявність внутрішніх засобів, які забезпечують тестування, налагодження та моніторинг продуктивності.

Ці характеристики дозволяють процесорам Intel Pentium 6 ефективно працювати з різноманітним програмним забезпеченням під керуванням операційних систем MS-DOS, Windows, OS/2, UNIX, SVR4, Solaris 2.0, NextStep 486. Код, який виконується , для цих процесорів повністю сумісний з кодом попередніх моделей мікропроцесорів сімейства Intel 80x86 (8086, 8088, 80186, 80286, 80386, 80486, Pentium).

Процесори мають три основних режими функціонування:

- режим реальних адресів (реальний режим);
- режим захищених віртуальних адрес (захищений режим);
- режим системного керування.

В реальному режимі процесор працює як швидкий мікропроцесор 8086, який виконує обробку 16-розрядних операндів та який адресує 1 Мбайт оперативної пам'яті.

В захищеному режимі можуть одночасно виконуватися декілька окремих програм (задач), які захищені одна від одної й від операційної системи процесору. В цьому режимі процесор може також виконувати програми, які написано для мікропроцесору 8086, якщо реалізується модифікація захищеного режиму – режим віртуального 8086.

В захищеному режимі реалізується також багатозадачне функціонування.

Режим системного керування (SMM – System Management Mode) використовується для реалізації спеціальних системних функцій.

Процесор оперує з фізичною пам’яттю об’ємом до 64 Гбайт. Кожний байт пам'яті має свою фізичну адресу – від 000000000h до FFFFFFFFh. В пам'яті можуть зберігатися 8-розрядні байти, 16-розрядні слова, 32-розрядні подвійні слова та 64-розрядні зчетверені слова.

Процесор виконує звернення до пам'яті, яке використовує два способи організації – сегментація та розбиття на сторінки.

Сегментація пам’яті забезпечується при будь-якому режимі роботи процесору. В системі на основі процесору Intel Pentium 6 кожній задачі доступно до 8192 сегменти величиною до 4 Гбайт кожний. Для звернення до комірки сегментованої пам'яті використовується логічна адреса, яка складається із селектору, який задає базову адресу сегменту (початок) й відносної адреси комірки в сегменті. Арифметичне складання базової та відносної адреси дає фізичну адресу.

Сторінкова організація пам'яті забезпечується тільки в захищеному режимі. Для її реалізації необхідно за допомогою команди LMSW або MOV CR0 встановити в регістрі CR0 значення біту сторінкової адресації PG=1. При цьому сегменти розділяються на окремі сторінки ємністю 4 Кбайт або 4Мбайт.

Сторінкова трансляція дозволяє розширити об’єм пам'яті, яка адресується до 64 Гбайт, використовуючи 36-розрядну шину адреси.

Регістрова модель процесорів містить набір регістрів, які складають наступні групи:

Основні функціональні регістри:

- регістри загального призначення EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI;
- вказівник команд EIP;
- регістри прапорів EFLAGS;
- регістри сегментів CS, SS, DS, ES, FS, GS.

Регістри блоку обробки чисел з плаваючою точкою (регістри FPU):

- регістри даних R7-0 (ST7-0);
- регістр тегів TW;
- регістр стану FPSR;
- регістр керування FPCR;
- регістри-вказівники команди й операнду FIP, FDP.

Регістри блоку обробки пакетів чисел с плаваючою крапкою (регістри SSE):

- регістри пакетів даних XMM 7-0;
- регістри системних адрес GDTR, LDTR, IDTR, TR;
- регістри відлагодження DR 7-0.

Службові (модельно-специфічні) регістри.

Регістри двох перших груп використовуються при виконанні прикладних програм, регістри третьої групи – при виконанні системних програм й відладки, регістри четвертої групи – при тестуванні мікропроцесору й контролю

ефективності виконання програм. Системні й службові регістри доступні лише програмам з вищим рівнем привілеїв 0.

Основні функціональні регістри

Вісім 32-розрядних **регістрів загального призначення** EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP призначаються для зберігання даних й адрес. При операціях з байтами можна окремо звертатися до молодших (розряди 7-0) або старших (розряди 15-8) байтів регістрів.

Сегментні регістри CS, SS, DS, ES, FS, GS містять 16-розряднв значення селекторів сегментів, які визначають сегмент пам'яті, який адресується. В захищеному режимі вміст цих регістрів задає вибір дескриптору, який містить значення базової адреси та інші атрибути сегменту.

Селектор в регістрі CS забезпечує звернення до сегменту команд, селектор SS – до сегменту стека, селектори в DS, ES, FS, GS – до сегментів даних.

Вказівник команд EIP представляє собою 32-розрядний регістр, вміст якого використовується в якості зміщення при визначенні адреси наступної команди, яки повинна виконуватись.

Регістр прапорів EFLAGS містить ряд бітів, які мають різне призначення: признаки стану CF, PF, AF, ZF, SF, OF вказують характеристики результату, який отримано при виконанні команди; керуючий признак DF визначає порядок адресації операндів при виконанні послідовності команд обробки символів; системні признаки TF, IF, IOPL, NT, RF, VM, AC, VIF, VIP, ID задають режим процесору при обслуговуванні виключень й переривань, організації вводу/виводу даних, рішенні послідовності задач, які викликаються та реалізації.

Признаки стану в регістрі EFLAGS мають наступне значення:

- CF – признак переносу, приймає значення CF=1 при виникненні переносу із старшого розряду;
- PF – признак парності, виникає, коли молодші 8 розрядів результату містить парну кількість одиничних біт;
- AF – признак напівпереносу, якщо виникає перенос між молодшими тетродами;
- ZF – признак нулю, приймає значення 1 при отриманні нульового результату операції;
- SF – признак знаку, приймає значення старшого (знакового) розряду результату операції;
- OF – признак переповнення, приймає значення 1 у випадку переповнення розрядної сітки при обробці операндів зі знаком;
- DF – признак напрямку, при значенні 0 визиває автоматичне збільшення вмісту індексних регістрів ESI, EDI (SI, DI) після виконання команди обробки символу при DF=1 – зменшення вмісту цих регістрів.

Системні признаки (окрім NT) встановлюються операційною системою. За допомогою цих регістрів система задає режими виконання ряду процедур:

TF – признак трасування, при значенні 1 процесор переключається в режим покрокового виконання команд, з реалізацією після кожної команди відповідного переривання;
IF – признак дозволу переривання, встановлення значення 1 дозволяє обслуговування запиту переривання, яке поступило на внутрішній вхід INTR;

Формати команд та способи адресації

Набір команд забезпечує виконання операцій над операндами, які знаходяться в регістрі, пам'яті або безпосередньо в команді. В набір входять безадресні, одно- та двоадресні команди. Процесор реалізує наступні шість типів двоадресних команд:

- регістр-регістр;
- пам'ять-регістр;
- безпосередні дані – регістр;
- регістр – пам'ять;
- пам'ять – пам'ять;
- безпосередні дані – пам'ять.

Операнди можуть містити 8, 16 або 32 розряди. Для реалізації різних типів команд визначені формати, які задають порядок розміщення інформації та способах вибору операндів.

Загальний формат команди містить наступні поля:

OPC (1 або 2 байти)	MODR/M (0 або 1 байт)	SIB (0 або 1 байт)	DISP (0,1,2 або 4 байти)	IMM (0, 1, 2, 4 байти)

Рис. 3.1. Загальний формат команд

OPC – код операції (operation code);
MODR/M, SIB – байти адресації;
DISP – байти зміщення;
IMM – безпосередньо заданий операнд.

Перед кодом операції в ряді випадків вводяться один або декілька префікс них байтів, які модифікують команду.

Код операції **OPC** займає 1 або 2 байти. В багатьох командах пересилок, а також в логічних та арифметичних командах перший байт OPC містить біт w, значення якого визначає розрядність операндів: w=0 – операція с байтами; w=1 – операція словами (16 або 32 розряди).

Байт адресації **MODR/M** містить три поля (рис. 3.1). Поля MOD та R/M задають адресу одного із операндів, який може зберігатись в регістрі або комірці пам'яті. Кодування визначає спосіб адресації, який вибирається.

В одноадресних командах поле REG/OPC містить додаткові біти коду операції. В двоадресних командах поле REG містить код регістру, в якому зберігається другий із операндів. Тип команди визначається першим бітом OPC. При цьому в OPC міститься біт d, який задає вибір регістрів, які використовуються у якості джерела та приймальника інформації:

- d=0 – код джерела міститься в полі REG/OPC, код приймальника в полі R/M;
- d=1 – код джерела міститься в полі R/M код приймальника в полі REG/OPC.

	7 6	5 4 3	2 1 0
а)	MOD	REG/OPC	R/M
б)	SS	INDEX	BASE

Рис. 3.2. Формати байтів MODR/M (а) та SIB (б)

Кодування регістрів наведено в таблицях 3.1, 3.2.

Таблиця 3.1. Кодування регістрів загального призначення

Поле REG	Розрядність операндів		
	8	16	32
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Таблиця 3.2. Кодування сегментів регістрів

Поле sreg	Сегментний регістр
000 (00)	ES
001 (01)	CS
010 (10)	SS
011 (11)	DS
100	FS
101	GS

Для реалізації деяких способів адресації використовується байт SIB. Він місить 3-бітні поля INDEX та BASE, які визначають вибір регістрів, які використовуються в якості індексного та базового регістрів, та поле SS, яке задає масштабний коефіцієнт для модифікації індексу (рис. 3.2).

Якщо поле MOD байту MODR/M має значення 00 або 01, 10, то для формування адреси використовується 8-, 16- або 32-розрядне зміщення (табл. 3.3, 3.4)

При виконанні операцій з безпосередньою адресацією один із операндів *imm* задається в останніх байтах команди (поле imm на рис. 3.1). В цьому випадку OPC ряду команд містить біт s, який визначає спосіб використання

даних, які безпосередньо задаються. Якщо операція виконується над над байтами (w=0), то в якості операнду використовується один байт даних im8.

Таблиця 3.3. Вибір сегментних регістрів та відносної адреси

Тип звернення до пам’яті	Сегментний регістр	Відносна адреса
1. Вибірка команди	CS	EIP (IP)
2. Звернення до стеку	SS	ESP (SP)
3. Адресація операнду	DS(CS,SS,ES,FS,GS)	EA
4. Адресація елементу рядка-джерела	DS(CS,SS,ES,FS,GS)	ESI (SI)
5. Адресація елементу рядка-приймальника	ES	EDI (DI)
6. Адресація операнду з використанням в якості базового регістру EBP (BP) або ESP (SP)	SS(CS,DS,ES,FS,GS)	EA

Таблиця 3.4. Формування 16-розрядної адреси EA

Поле R/M	Поле MOD		
	00	01	10
000	DS:[BX+SI]	DS:[BX+SI+d8]	DS:[BX+SI+d16]
001	DS:[BX+DI]	DS:[BX+DI+d8]	DS:[BX+DI+d16]
010	SS:[BP+SI]	SS:[BP+SI+d8]	SS:[BP+SI+d16]
011	SS:[BP+DI]	SS:[BP+DI+d8]	SS:[BP+DI+d16]
100	DS:[SI]	DS:[SI+d8]	DS:[SI+d16]
101	DS:[DI]	DS:[DI+d8]	DS:[DI+d16]
110	DS:[d16]	SS:[BP+d8]	SS:[BP+d16]
111	DS:[BX]	DS:[BX+d8]	DS:[BX+d16]

Таблиця 3.5. Формування 32-розрядної адреси EA (байт SIB відсутній)

Поле R/M	Поле MOD		
	00	01	10
000	DS:[EAX]	DS:[EAX+d8]	DS:[EAX+d32]
001	DS:[ECX]	DS:[ECX+d8]	DS:[ESX+d32]
010	DS:[EDX]	DS:[EDX+d8]	DS:[EDX+d32]
011	DS:[EBX]	DS:[EBX+d8]	DS:[EBX+d32]
100	див. табл. 9	див. табл. 9	див. табл. 9
101	DS:[d32]	SS:[EBP+d8]	SS:[EBP+d32]
110	DS:[ESI]	DS:[ESI+d8]	DS:[ESI+d32]
111	DS:[EDI]	DS:[EDI+d8]	DS:[EDI+d32]

В залежності від типу звернення до пам'яті проводиться вибір сегментного регістру та способу визначення відносної адреси (табл. 3.3). Для деяких способів звернення можливі варіанти вибору сегментних регістрів, котрі

наведено в табл. 3.3 в скобках. Ці варіанти можуть бути вибрані за допомогою префіксу заміни сегменту SEG. В якості відносної адреси використовуються вміст регістрів EIP(IP), ESP (SP), ESI (SI), EDI (DI) або ефективна адреса EA, яка формується у відповідності з заданим способом адресації.

Таблиця 3.6. Формування 32-розрядної адреси EA (байт SIB присутній)

Поле R/M	Поле MOD		
	00	01	10
000	DS:[EAX+(IR*F)]	DS:[EAX+(IR*F)+d8]	DS:[EAX+(IR*F)+d32]
001	DS:[ECX+(IR*F)]	DS:[ECX+(IR*F)+d8]	DS:[ESX+(IR*F)+d32]
010	DS:[EDX+(IR*F)]	DS:[EDX+(IR*F)+d8]	DS:[EDX+(IR*F)+d32]
011	DS:[EBX+(IR*F)]	DS:[EBX+(IR*F)+d8]	DS:[EBX+(IR*F)+d32]
100	SS:[ESP+(IR*F)]	SS:[ESP +(IR*F)+d8]	SS:[ESP+(IR*F)+d32]
101	DS:[d32+(IR*F)]	SS:[EBP+(IR*F)+d8]	SS:[EBP+(IR*F)+d32]
110	DS:[ESI+(IR*F)]	DS:[ESI+(IR*F)+d8]	DS:[ESI+(IR*F)+d32]
111	DS:[EDI+(IR*F)]	DS:[EDI+(IR*F)+d8]	DS:[EDI+(IR*F)+d32]

Ефективна адреса операнду EA є 16- або 32-разрядною та формується в залежності від значення полів MOD та R/M в байті адресації MODR/M та вмісту байта SIB. В загальному випадку EA утворюється шляхом арифметичного складання трьох компонентів:

- вмісту базового регістру EBP (BP) або EBX (BX);
- вмісту індексного регістру (ESI (SI) або EDI (DI);
- 8-, 16- або 32-розрядного зміщення, яке задано в одно-, двох- або чотирьох байтах поля DISP команди рис. 3.1.

Методичні вказівки

1. Вивчити теоретичний матеріал.
2. Підготувати відповіді на контрольні питання.

Вимоги до звіту

В звіті відобразити лаконічні відповіді на контрольні питання.

Питання для перевірки

1. Характеристики процесорів сімейства Intel Pentium 6.
2. Режими функціонування процесорів.
3. Регістрова модель процесору.
4. Основні функціональні регістри.
5. Формати команд та способи адресації.

Лабораторна робота № 4

ПРОГРАМА МОВОЮ АСЕМБЛЕРА

Мета роботи: вивчити особливості побудови програм мовою асемблера і закріпити одержані знання.

Теоретичні відомості

Першим етапом створення програми є написання текстового файлу спеціальною мовою – асемблера. Подібний файл створюється будь-яким текстовим редактором.

В подальшому цей текстовий файл буде оброблений спеціальною програмою-транслятором для перетворення в машинний код.

В цій та наступних лабораторних роботах наведені правила написання програм мовою асемблера фірми Microsoft – MASM.

Програма, яка написана мовою асемблера складається із рядків. Максимальна довжина рядка – 132 символи. На практиці використовується лише 80, це ширина екрану монітору. Рядки відокремлюються одна від одної символом повернення каретки (Enter).

Основний формат рядка.

[мітка:] команда [операнд,] [операнд] [;коментар]

Мітка, якщо вона, команда та перший операнд відокремлюються одним чи більше пробілами чи символом табуляція.

МІТКА – це мнемонічне ім’я, яким відмічено даний рядок для наступного звернення. Мітка може містити до 31 символу. Допустимі символи:

- букви: від “A” до “Z”, від “a” до ”z”;
- цифри: від 0 до 9;
- спеціальні символи: знак питання (?); крапка (.) – тільки перший символ; ет (@); підкреслення (_); долар ($).

Першим символом мітки повинна бути літера або спеціальний символ. Асемблер не робить різниці між заголовними й рядковими буквами.

Імена регістрів зарезервовані. Їх допускається використовувати тільки для позначення регістрів.

КОМАНДА (оператор) – означає дію, яку повинен здійснити процесор або транслятор з мови асемблера. В якості команд можуть використовуватися команди процесору (команди) або директиви (псевдокоманди) асемблера.

ОПЕРАНД визначає елемент над яким проводить дію команда.

Команда може містити один чи два операнди або не мати їх зовсім.

```
ret              – команда без операнда;
inc   cx         – команда з одним операндом;
add   ax, 12     – команда з двома операндами.
```

Якщо команда має два операнди, вони відокремлюються комою (,). При використанні двох операндів першим вказується операнд-приймач, другим –

операнд-джерело. Таким чином, результат операції заноситься в перший операнд.

Приклади операндів:

```
mov   ax, bx      ; переслати вміст BX в AX
mov   ax, worda   ; переслати вміст комірки пам’яті під іменем worda в ax
mov   ax, [bx]    ; переслати вміст пам’яті за адресою, яку вказано в BX
                  ; в регістр AX
mov   ax, 25      ; переслати число 25 в регістр AX
mov   ax, [25]    ; переслати вміст комірки пам’яті зі зміщенням 25 в AX
```

КОМЕНТАРІ починаються з крапки з комою (;). Можуть займати увесь рядок або йти за командою в тому же рядку. Коментар з’являється тільки в листінгах і не приводять до генерацій машинних кодів. В коментарі допускаються любі символи основної клавіатури.

ДИРЕКТИВИ (або псевдокоманди).

Це ряд операторів, які дозволяють керувати процесом трансляції та формування листінга. Вони діють тільки в процесі трансляції і не генерують машинних кодів.

Директива **SEGMENT**. Використовується для опису сегментів, які використовуються в програмі. Будь-яка програма містить хоча би один сегмент – сегмент коду. Також може бути присутній сегмент стеку та сегмент даних, а також додатковий сегмент. Формат директиви:

```
Ім’я      Директива      Операнд
ім'я      SEGMENT        [параметри]
          .
          .
          .
ім'я      ENDS
```

Ім'я сегмента повинно обов’язково бути присутнім, бути унікальним та відповідати домовленостям для імен в асемблері. Директива **ENDS** означає кінець сегменту. Обидві директиви **SEGMENT** та **ENDS** повинні мати одинакові імена. Директива **SEGMENT** може вміщати три класи параметрів: вирівнювання, об'єднання, клас.

Вирівнювання. Цей параметр визначає границю початку сегменту. У випадку відсутності цього операнда за умовчанням приймається PARA, за яким сегмент встановлюється на границю параграфу.

Об'єднання. Цей параметр визначає, об’єднується даний сегмент с іншими сегментами в процесі компоновки чи ні. Можливі наступні типи об’єднань STACK, COMMON, PUBLIC, AT-вираз, MEMORY.

Якщо програма не повинна об’єднуватись з іншими програмами, то вказівка цих типів може бути пропущена.

Клас. Цей елемент, в апострофах, використовується для групування сегментів при компоновці.

Сегмент стека визначається наступним чином:

```
ім'я  SEGMENT     PARA     STACK     'Stack'
      DB 30 DUP (?)
ім'я  ENDS
```

Директива PROC. Сегмент коду містить команди програми, які виконуються. Він може містить в собі одну або декілька процедур, які визначаються директивою PROC. Сегмент, який містить тільки одну процедуру має наступний вигляд:

```
ім'я-сегменту     SEGMENT     PARA
ім'я-процедури    PROC        FAR; сегмент коду з однією процедурою
                  .
                  .
                  RET
ім'я-процедури    ENDP
ім'я-сегменту     ENDS
```

Ім'я процедури повинно бути обов'язково присутнім, бути унікальним та відповідати домовленостям по іменам в асемблері. Оператор FAR вказує завантажнику DOS, на те що початок процедури є точкою входу для виконання програми. Сегмент може мати декілька процедур. Кожна процедура повинна закінчуватись директивою RET.

Директива ASSUME. Процесор використовує регістр SS для адресації стеку, регістр DS – для адресації сегменту даних та регістр CS – для адресації сегменту коду, ES – для адресації екстра сегменту. Асемблеру необхідно повідомити призначення кожного сегменту.

```
Директива  Операнд
ASSUME     SS:ім_стеку, DS:ім_даних, CS:ім_код, ES:ім_ес
```

де ім_стеку, ім_даних, ім_код – імена відповідно сегменту стеку, даних, коду та екстра сегменту, які привласнені їм директивою SEGMENT.

Операнди можуть записуватись в любій послідовності. Регістр ES також може буди присутнім між операндів. Якщо програма не використовує регістр ES, то його можна пропустити або вказати ES:NOTHING.

Директива END. Повністю завершує всю програму.

```
Директива  Операнд
END        [ім'я процедури]
```

Операнд може бути опущений, якщо програма не передбачена для виконання (транслюється тільки для визначення даних або повинна бути скомпонована з іншим (головним) модулем). Для звичайної програми з одним (головним) модулем операнд містить ім'я, яке указано в директиві PROC, яке було позначено, як FAR.

ІНІЦІАЛІЗАЦІЯ ПРОГРАМИ

Існує два основних типу завантажувальних модулів EXE та COM. Розглянемо вимоги до EXE-програм. Система DOS має чотири вимоги для ініціалізації асемблерної EXE-програми.

1. Необхідно вказати асемблеру, які сегментні регістри відповідають сегментам (здійснюється директивою ASSUME).

2. Вимагається зберегти в стеці адресу, яка знаходиться в регістрі DS, коли програма починає виконуватись.

Завантажувальному модулю в оперативній пам'яті безпосередньо передує 256-байтова (100h) область, яка іменується префіксом програмного сегменту (PSP). Програма завантажника використовує регістр DS для встановлення адреси початкової точки PSP. Програма користувача повинна зберегти цю адресу, розмістивши її в стек.

3. Необхідно записати в стек нульову адресу.

4. Необхідно завантажити в DS адресу сегменту даних.

Завантажник DOS встановлює правильні адреси стеку в регістрі SS та сегменту кодів в регістрі CS. Оскільки програма завантажника використовує регістр DS для інших цілей, то необхідно провести ініціалізацію регістра DS.

Вихід із програми та повернення в DOS зводиться до використання команди RET.

Команда RET забезпечує вихід із програми користувача та повернення в DOS, для цього використовується адреса, яка записана в стек на початку програми із регістра DS. Іншим способом завершення програми, який часто використовується, є команда INT 20h.

Приклад виконання завдання

```
segst  segment     para  stack  'Stack'
               db     30 dup (?)
segst  ends
segdt  segment     para   'data'
               db     12 dup (?)
segdt  ends
segcod         segment para 'code'
               assume cs:segcod, ss:segst, ds:segdt
begin proc     far
        push   ds              ; зберегти вміст DS в стеку
        sub    ax, ax          ; обнулити регістр AX
        push   ax              ; занести вміст AX в стек
        mov    ax, segdt       ; занести адресу початку сегменту  даних в регістр AX
        mov    ds, ax          ; занести вміст AX в регістр DS
        sub    bx, bx          ; обнулити регістр BX
        add    bx,10           ; додати до вмісту BX 10
        mov    cx, 20          ; занести в CX 20
        add    bx, cx          ; додати CX до BX
        sub    bx, cx          ; відняти CX від BX
        ret
begin endp
segcod         ends
        end    begin
```

Питання для перевірки

1. Навести формат рядка мови асемблера.
2. Що буде знаходитись в регістрах AX та BX після виконання наступних команд:

```
MOV     AX, 20
MOV     BX, 10
ADD     AX, BX
```

3. Перерахуйте вимоги для ініціалізації EXE-програми.

Варіанти завдань

Написати програму, яка виконує наступні дії:

1. Очистити регістр CX.
2. Переслати число xxh в регістр AL.
3. Додати число xxxxh до регістру AX.
4. Переслати регістр AX в регістр BX.
5. Додати регістр BX до регістру AX.
6. Відняти регістр BX від регістру AX.

Де, xx – порядковий номер студента відповідно до списку в журналі; xxxx – дата народження, число та місяць.

Виконати трансляцію.

Створити листінг.

Створити exe-файл.

Виконати трасування програми з переглядом вмісту регістрів.

Лабораторна робота № 5

СТРУКТУРА COM-ФАЙЛІВ

Мета роботи: вивчити призначення та особливості COM-файлів, набути навичок створення асемблерних програм у вигляді COM-файлів.

Теоретичні відомості

Програму, яка виконується, можна створити не тільки у вигляді exe-файлу. Прикладом командного файлу може бути COMMAND.COM. Програми у вигляді exe-файлу та у вигляді com-файлу мають наступні відмінності.

1. **Розмір програми.** Програма у форматі exe може мати будь-який розмір. Com-програма обмежена розміром одного сегменту та не перевищує 64 Кбайт. Розмір com-файлу завжди менше розміру exe-файлу. Одна з причин цього – відсутність в com-файлі спеціального 512-байтового заголовку exe-файлу.

2. **Сегмент стеку**. В exe-програмі необхідно обумовити сегмент стеку. В COM-програмі стек генерується автоматично. В зв’язку з цим, при створенні асемблерної програми, яка буде перетворена в com-файл, сегмент стеку повинен бути пропущено. Якщо максимальний розмір сегменту (64К) достатній для програми, то MS DOS встановлює регістр SP на кінець сегменту (FFFE). Це є вершина стеку. Якщо програма займає увесь сегмент, місця для стеку недостатньо, то MS DOS встановлює стек в кінець пам'яті.

3. **Сегмент даних.** В exe-програмі визначається сегмент даних й регістр DS ініціалізується адресою цього сегменту. В com-програмі буде тільки один сегмент – сегмент кодів.

4. **Ініціалізація.** Спочатку exe-програми виконуються запис в стек значення регістру DS, нуля та ініціалізація регістра DS. Таким чином, сегменти стеку та даних в com-програмі не визначені, й ці кроки в ній відсутні. При завантаженні com-програми в оперативну пам'ять всі сегментні регістри містять адресу префіксу програмного сегменту (PSP) – 256-байтного (100h) блоку, який резервується MS-DOS в оперативній пам'яті перед exe- та com-програмою. Тому що регістр CS також може бути встановлений на початок PSP, необхідно встановити його на початок програми. Це може бути виконано директивою ORG 100h після директиви SEGMENT. Ця директива встановить значення вказівника команд (зсув відносно CS) 100h.

5. **Створення com-**файлу.

При використанні пакету MASM створіть exe-файл. При цьому компоновником буде видано повідомлення.

Warning: No STACK Segment (Сегмент стеку не визначено).

Це повідомлення можна проігнорувати.

Для перетворення exe-файлу в com-файл використовується програма EXE2BIN. Введіть командну строку у вигляді:

[шлях] exe2bin [шлях] ім’я_файлу.exe, [шлях] ім’я_файлу.com

Тому що перший файл завжди має розширення .exe, тому його можна не вказувати. Для другого файлу за умовчанням прийнято розширення .bin, тому вкажіть розширення .com, явно.

При роботі з TASM для створення com-файлу необхідно вказати опцію /t в процесі компоновки. Командний рядочок при компоновці буде мати вигляд:

[шлях] tlink/t objfiles, comfiles, mapfiles, libfiles, defiles.

При цьому, файл який виконується буде мати по умовчанню розширення .com.

Приклад виконання завдання:

```
segcod      segment     para   'code'
            assume      cs:segcod, ss:segcod, ds:segcod, es:segcod
            org         100h
begin:      jmp         main
            db          12 dup (?)
main        proc   near
            sub         bx, bx      ; обнулити регістр BX
            add         bx,10       ; додати до вмісту BX 10
            mov         cx, 20      ; занести в CX 20
            add         bx, cx      ; додати CX до BX
            sub         bx, cx      ; відняти CX від BX
            ret
main        endp
segcod      ends
            end    begin
```

Зверніть увагу на зміни, які було зроблено.

1. Сегмент стеку та сегмент даних відсутні.
2. Директива ASSUME ініціалізує всі сегментні регістри адресою початку сегменту кодів.
3. Директива ORG 100h встановлює вказівник команд на початок програми (після PSP).
4. Команда JMP (безумовний перехід) використовується для обходу області даних.
5. Після мітки точки входу до програми (BEGIN) знаходиться дві крапки. Його значення буде розглядатися далі (наступна лабораторна робота).
6. Процедура MAIN, яка задається параметром NEAR (внутрішня), тому що її початок вже не є точкою входу до програми.

Варіанти завдань

Перетворити програму, яку було зроблено в лабораторній роботі № 1 у вигляд, необхідний для створення com-програми.

Лабораторна робота № 6

РОБОЧІ ПОЛЯ ПРОГРАМИ

Мета роботи: набути навичок використання констант та робочих полів в програмі мовою асемблера.

Теоретичні відомості

В лабораторній роботі № 1, 4 необхідні дані задавалися безпосередньо в команді. Наприклад, mov cx, 25 або mov ax, 25.

В цьому випадку число 25 стає частиною машинного об’єктного коду.

Такий спосіб представлення даних доцільний, якщо задане число (в нашому випадку 25) зустрічається в програмі лише один раз. У випадку якщо задане число зустрічається в програмі багаторазово, то повторне завдання його в різних командах приведе до невиправданої витрати оперативної пам’яті. Пам'ять під число буде виділятися кожного разу, коли ви його вказуєте. Більш доцільним є запис числа в комірку оперативної пам'яті, присвоєнні цій комірці мнемонічного імені, надалі, звертатися к заданому числу за іменем. В цьому випадку пам'ять під число буде виділятися лише один раз.

В лабораторній роботі № 4 ми використовували для зберігання результатів обчислень (проміжних та результуючих) регістри загального призначення. Такий спосіб є найбільш швидкодіючим. Однак, в регістрах можна зберігати обмежений об’єм інформації.

Замість використання регістрів для зберігання результатів можна виділяти в оперативній пам'яті так звані робочі області – визначену кількість комірок оперативної пам'яті, котрим можна присвоїти імена та в командах звертатися до них по іменах.

В асемблері для опису вихідних даних та робочих областей в оперативній пам'яті служать директиви визначення даних.

За допомогою цих директив можна задавати константи (числа, значення яких визначаються в вихідному тексті програми) та змінні (робочі області визначеної довжини, значення яких може змінюватися в процесі роботи програми).

Основний формат визначення даних

[ім’я] Dn вираз

Ім’я елемента даних не обов’язкове, однак, якщо ви бажаєте в програмі звертатися до цього елементу, ви повинні це ім’я вказати. Ім’я задається за правилами, згідно за правилами написання міток (див. лабораторну роботу №4).

Директиви визначення даних (Dn) можуть використовуватися наступні: DB (байт), DW (слово), DD (подвійне слово), DQ (облікове слово), DT (десять байт).

В полі "вираз" може знаходитися:

1. Константа.

```
ABC  db     25
```

2. Послідовність констант.

```
BCD  db     1,2,3,4,5
```

Константи в послідовності повинні розділятись комами. Кількість констант в послідовності обмежено довжиною рядка. Ім’я (в нашому прикладі BCD) указує на першу константу із послідовності (в нашому прикладі – 1). До другої константи (2) можна звертатися по імені BCD+1, до третьої BCD+2 і так далі.

3. В якості виразу може стояти знак питання (?) для невизначеного значення змінної:

CDE db (?)

В цьому випадку значення змінній CDE не присвоюється.

4. Вираз допускає повторення константи або змінній в наступному вигляді

[ім’я] Dn число_повтореннь DUP (вираз)

наприклад,

DEF db 10 dup (?)

Оператор повторення (DUP) може бути вкладеним, наприклад

EFH db 10 dup (4 dup (2))

В цьому випадку будуть сгенеровані спочатку 4 копії константи 2, далі це значення повториться 10 раз. В результаті буде сформована послідовність із сорока 2.

Константа, яка задається в полі "вираз" може представляти собою символьний рядок або числову константу.

Символьний рядок використовується для представлення тексту. Вона береться в лапки або апостроф, наприклад

“Лабораторна робота № 5” або

‘Лабораторна робота № 5’

Асемблер переводить кожний символ символьного рядку у відповідний ASCII-код. Символьний рядок визначається директивою DB, в якій вказується більше двох символів в нормальній послідовності зліва направо.

NAME1 db ‘Лабораторна робота № 5’

Числові константи використовуються для арифметичних значень та для адрес пам'яті. Числова константа вказується безпосередньо в директиві визначення даних, без апострофів та лапок. Асемблер перетворює всі числові константи в двійкові та записує їх як послідовність байтів в зворотній послідовності – справа наліво.

Числові константи можуть задатися в наступних форматах.

Десятковий формат допускає десяткові цифри від “0” до “9” та визначається останньою буквою “D”. Десятковий формат приймається за умовчанням, тобто якщо ніяка буква після числа не вказана, то це десятковий формат. Десятковий формат перетворюється асемблером в двійковий, котрий в листінгу записується в шістнадцятковому форматі.

Наприклад,

NAME2 db 12d або

NAME2 db 12

Шістнадцятковий формат допускає цифри від “0” до “F” та визначається останньою буквою “H”. Оскільки асемблер вважає, що з букви

починаються імена, то першою цифрою шістнадцяткової константи повинна бути цифра від “0” до “9”. Якщо, наприклад, треба задати константу A0C1h, то її необхідно записати у вигляді 0A0C1h.

```
NAME3    db    0Ch
```

Двійковий формат допускає цифри від “0” до “1” та визначається останньою буквою “B”.

```
NAME4    db    11001010b
```

Вісімковий формат допускає цифри від “0” до “7” та визначається останньою буквою “Q” або “O”. В даний час вісімко вий формат використовується рідко.

```
NAME5    db    10q
```

Існує ще десятковий формат з плаваючою точкою, але він використовується в основному для завдання даних для співпроцесору.

При завданні символьних рядків та числових констант необхідно пам’ятати, що, наприклад ‘14’ та 14 представляють собою дві різні константи. Перша перетворюється в два байти ASCII-коду, яке відповідає символам “1” та “4”. Друга перетворюється в двійкове число 1110, в шістнадцятковому представленні – OE.

Директива визначення байту (DB) – визначити байт. Символьний вираз в директиві DB може містить в собі символьний рядок будь-якої довжини (обмежується тільки довжиною асемблерного рядку). Числова константа в директиві DB може містити одну або більше однобайтових констант. Один байт в листінгу представляється двома шістнадцятковими цифрами. Найбільше позитивне шістнадцяткове число в одному байті – 7F, всі більші числа від 80 до FF представляють негативні значення. В десятковому форматі ця межа виражається числами +127 та –128.

Приклад використовування директиви визначення байту:

```
NAME6    db    ?
NAME7    db    ‘Завдання № 5’
NAME8    db    12h
NAME9    db    112
NAME10   db    01101100b
NAME11   db    10,11,12,13,14
NAME12   db    2 dup (10)
```

Директива визначення слова (DW) визначає константи та змінні, які мають довжину в одне слово – два байту. Символьний вираз в DW обмежений двома символами, які асемблер представляє в зворотному порядку. Числовий вираз в DW може містити одну або більше двобайтових констант. Два байти представляються чотирма шістнадцятковими цифрами. Найбільше позитивне шістнадцяткове число в двох байтах – 7FFF, числа від 8000 до FFFF представляють негативні значення. В десятковому коді ця межа виражається числами +32767 та –32768. В полі ‘вираз’ директиви DW може бути підставлено ім’я константи або змінної, яка була визначна раніше, наприклад

```
NAME13   dw    NAME8
```

В цьому випадку під іменем NAME13 буде сгенерована константа, рівна адресі (зміщенню) константи NAME8.

Для директив DW, DD, DQ асемблер перетворює константи в шістнадцятковий об’єктний код, але записує їх в зворотному порядку проходження байтів. Таким чином, десятинне значення 12345 перетворюється в 3039H, але записується в об’єктному коді як 3930.

Приклади використання директиви визначення слова:

```
NAME14   dw   ?
NAME15   dw   23,64
NAME16   dw   ‘12’
NAME17   dw   NAME15
NAME18   dw   0110111100001010b
NAME19   dw   1F0Bh
```

Директива визначення подвійного слова (DD) визначає елементи, які мають в довжину два слова – чотири байти. Розмір числових констант, які використовуються в виразі, не повинне перевищувати чотирьох байт (вісім шістнадцяткових цифр). Найбільше позитивне шістнадцяткове число, яке може бути використане в директиві DD – 7FFFFFFF. Діапазон негативних чисел – від 80000000 до FFFFFFFF. В десятковому коді діапазон допустимих чисел в цій директиві – від +2147483647 до -2147483648. Асемблер перетворює всі числові константи в шістнадцяткове представлення, але записує об’єктний код в зворотній послідовності. Десяткове значення 12345 перетворюється в 00003039H, але записується в об’єктному коді як 39300000. Символьний вираз в директиві DD обмежений двома символами. Асемблер перетворює їх в ASCII-код та вирівнює їх зліва, як показано в наведеному рядку листінгу

```
0010  33 32 00 00        NAME20   dd   ‘23’
```

Приклади використовування директиви визначення подвійного слова:

```
NAME21   dd   1F00FFFFh
NAME22   dd   ‘AB’
NAME23   dd   ?
NAME24   dd   45,47,49
NAME25   dd   NAME23-NAME21
```

Директива визначення почетвереного слова (DQ) – визначає елементи, які мають довжину чотири слова (вісім байтів). Числові константи, які використовуються в директиві, не повинні перевищувати розмір вісім байтів. Асемблер перетворює всі числові константи в шістнадцятко вий вираз, але записує їх в об’єктний код в зворотній послідовності. Обробка асемблером символьних рядків в директиві DQ аналогічна обробці в директивах DW та DD.

Приклади використовування директиви визначення почетвереного слова:

```
NAME26   dq   245h
NAME27   dq   100
```

Директива визначення десяти байт (DT) – визначає елементи даних, які мають довжину десять байтів. Призначена в основному для визначення так званих "упакованих десяткових" числових значень. Генеруємий об’єктний код залежить від версії асемблера.

Приклади використання директиви визначення десяти байтів:

```
NAME28   dt    ?
NAME29   dt    ‘23’
```

Безпосередні операнди

Константа, яка задається безпосередньо в команді, називається безпосереднім операндом. Значення безпосереднього операнду входить в генеруємий для цієї команди об’єктний код. Довжина безпосереднього операнду залежить від довжини першого операнду в команді. Наприклад, в команді

```
mov   al, xx
```

безпосередній операнд, який ми позначили як XX, повинен мати довжину один байт, тому що регістр AL є однобайтовим регістром.

В команді

```
mov   ax,xx
```

безпосередній операнд може мати довжину слово (два байти).

Якщо це правило не буде дотримуватися, транслятор сгенерує повідомлення про помилку.

Директива EQU

Директива EQU не визначає елементи даних, але визначає значення, яке може бути використано в інших командах. Наприклад, якщо задамо

```
KOL  equ   10
```

то кожний раз, коли транслятор в тексті програми зустріне KOL, він замість нього підставить значення 10. Так транслятор перетворює директиву

```
NAME30   dw    KOL  dup (?)
```

в директиву

```
NAME30   dw    dup (?)
```

Ім’я, яке пов’язане з деяким значенням за допомогою директиви EQU, може використовуватися в якості операнда в команді. Наприклад,

```
mov   ax, KOL
```

Транслятор замінить ім’я KOL на значення 10, створює безпосередній операнд, неначебто було задано

```
mov   ax, 10
```

Контрольні питання

1. В чому різниця представлення в пам'яті двох констант

```
NAMEA    db    42
NAMEB    db    ‘42’
```

2. Яке значення підставиться замість імені NAMED у виразі

```
NAMEC    dw    20
NAMED    dw    NAMEC
```

3. Як будуть представлені в об’єктному коді константи

```
NAMEE    dd    2756
NAMEF    dd    ‘12’
```

Варіанти завдань

Створіть програму, яка визначає наступні в завданні поля даних та константи.

Виконайте трансляцію. Створіть листінг.

exe-файл не створювати!!! Програма не передбачена для виконання.

Модифікуйте програму із лабораторної роботи № 4 так, щоб опис констант було проведено в сегмент даних.

Варіант 1.

1. Визначення байту:
1.1. Неініціалізоване поле;
1.2. Символьний рядок, який містить Ваше прізвище, ім'я, ім'я по батькові;
1.3. Десяткова константа 54;
1.4. Число в символьній формі '54';
1.5. Двійкова константа, яка відповідає десятковому числу 67;
1.6. Шістнадяткова константа, яка відповідає десятковому числу 84;
1.7. 11 десяткових чисел 93;
2. Визначення слова:
2.1. Шістнадцяткова константа, яка відповідає десятковому числу 20389;
2.2. Двійкова константа, яка відповідає десятковому числу 28256;
2.3. Адресна константа, яка вказує на константу з п. 1.3;
2.4. Три послідовних константи, які представляють собою числа натурального ряду;
2.5. 3 десяткових чисела 4273;
3. Визначення двійкового слова:
3.1. Неініціалізоване поле;
3.2. Шістнадцяткова константа, яка відповідає десятковому числу 1036117869;
3.3. Різниця адресів констант із п.п. 2.1 та 1.5;
3.4. Дві десяткові константи: номер Вашого варіанту та номер лабораторної роботи;
3.5. Десяткова константа 1904188434;
4. Визначення учетвереного слова:
4.1. Неініціалізоване поле;
4.4. Десяткова константа 2,8735387390E+09;
5. Визначення десяти байтів:
5.1. Неініціалізоване поле;
5.2. Символьний рядок '5.2'.

Варіант 2

1. Визначення байту:
1.1. Неініціалізоване поле;
1.2. Символьний рядок, який містить Ваше прізвище, ім'я, ім'я по батькові;
1.3. Десяткова константа 27;
1.4. Число в символьній формі '27';

1.5. Двійкова константа, яка відповідає десятковому числу 168;
1.6. Шістнадяткова константа, яка відповідає десятковому числу 39;
1.7. 11 десяткових чисел 117;
2. Визначення слова:
2.1. Шістнадцяткова константа, яка відповідає десятковому числу 23384;
2.2. Двійкова константа, яка відповідає десятковому числу 42500;
2.3. Адресна константа, яка вказує на константу з п. 1.3;
2.4. 10 послідовних констант, які представляють собою числа натурального ряду;
2.5. 10 десяткових чисел 4273;
3. Визначення двійкового слова:
3.1. Неініціалізоване поле;
3.2. Шістнадцяткова константа, яка відповідає десятковому числу 143976745;
3.3. Різниця адресів констант із п.п. 2.1 та 1.5;
3.4. Дві десяткові константи: номер Вашого варіанту та номер лабораторної роботи;
3.5. Десяткова константа 2092352629;
4. Визначення учетвереного слова:
4.1. Неініціалізоване поле;
4.4. Десяткова константа 2,8735387390E+09;
5. Визначення десяти байтів:
5.1. Неініціалізоване поле;
5.2. Символьний рядок ‘5.2’.

Лабораторна робота № 7

ФУНКЦІЇ ВВОДУ/ВИВОДУ BIOS ТА MS DOS

Мета роботи: вивчити основні функції консольного вводу та виводу, які забезпечуються перериваннями BIOS та MS DOS, навчитись використовувати функції вводу/виводу в прикладних програмах мовою асемблера, закріпити знання.

Теоретичні відомості

Важливу роль при програмуванні мовою асемблера грають функції, які забезпечуються базовою системою вводу/виводу (BIOS) та операційною системою. Ці функції записуються в оперативну пам'ять ПК в процесі завантаження, прив’язуються до деякого переривання і можуть викликатися прикладними програмами. Для здійснення такого виклику програма повинна записати в регістр AH (або AX) умовний номер функції (його можна дізнатися з довідникової літератури), в інші регістри (якщо це передбачено специфікацією функції) – її аргументи і виконати команду INT X (виклик програмного переривання з номером X). Обробник переривання виконає передбачені дії та поверне керування назад до програми (точніше – до команди, наступної після виклику функції).

Використання готових функцій значно спрощує роботу програміста, так як ручна реалізація навіть примітивних процедур мовою асемблера є дуже трудомістким процесом. Далі приведені два програмні фрагменти, кожен з яких призначений для виводу рядка "Hell'o" на дисплей, але перший виконує це "самостійно", а другий – використовує функцію 9 переривання 21h DOS.

```
str db 'Hello'
…
lea si,str
mov ax,0xB800
mov es, ax
xor di,di
mov cx,5
cycle:
mov al,byte ptr [si]
mov ah,7
mov word pt res:[di],ax
add di,2
inc si
dec cx
jns cycle
…
```

Реалізація без використання функцій

```
str db 'Hello$'
…
mov ah,09h
lea dx,str
int 21h
```

Реалізація з використанням функції DOS

Базова система вводу/виводу надає програмістові широкий спектр функцій, призначених для зчитування даних з клавіатури та виводу на дисплей, які є доступними через програмні переривання 16h і 10h. Перелік основних функцій приведений в таблиці 7.1 та 7.2.

Таблиця 7.1. Основні функції вводу/виводу переривання 16h BIOS

Номер	Назва функції	Опис функції
0, 10h, 20h	Зчитування символу з очікуванням	Вхід: AH=0 (10h, 20h); Вихід: AL – ASCII-код символу, 0 або префікс скан-коду; AH – скан-код натиснутої клавіші або розширений ASCII-код. Якщо клавіші відповідає ASCII-символ, в регістрі AH повертається код цього символу, а в регістрі AL – скан-код клавіші. Якщо клавіші відповідає розширений ASCII-код, в регістрі AL повертається префікс скан-коду або 0, якщо його немає, а в регістрі AH – розширений ASCII-код. Функція 0 обробляє дані від 84-клавішної клавіатури; 10h – 105-клавішної; 20h – 122-клавішної
1, 11h, 21h	Перевірка наявності символу	Вхід: AH=1 (11h, 21h); Вихід: ZF=1, якщо буфер пустий, ZF=0, якщо в буфері є символ і AL – ASCII-код символу, 0 або префікс скан-коду; AL – ASCII-код символу, 0 або префікс скан-коду; AH – скан-код натиснутої клавіші або розширений ASCII-код. Функція 1 обробляє дані від 84-клавішної клавіатури; 10h – 105-клавішної; 20h – 122-клавішної
2, 12h, 22h	Зчитування стану клавіатури	Вхід: AH=2 (12h, 22h); Вихід: AL – байт стану клавіатури; AH – байт 2 стану клавіатури (для функцій 12h і 22h). Формати байтів стану приведені в таблиці 3
5	Поміщення символу в буфер клавіатури	Вхід: AH=5; CH – скан-код; CL – ASCII-код. Вихід: AL=0 – операція виконана успішно; AL=1 – буфер переповнений

Таблиця 7.2. Основні функції вводу/виводу переривання 10h BIOS

Номер	Назва функції	Опис функції
0	Встановлення відеорежиму	Вхід: AH=0 (10h, 20h); Вихід: AL – номер відео режиму. Стандартний текстовий відео режим 80x25, 16 кольорів має номер 3
2	Встановлення курсору	Вхід: AH=2; BH – номер відео сторінки; DH – рядок; DL – стовпчик. Відлік рядка і стовпчика ведеться від лівого верхнього кута екрану, який має координати (0;0). Дозволяється виводити текст на неактивну сторінку
3	Зчитування положення і розміру курсору	Вхід: AH=0; BH – номер відеосторінки; Вихід: DH – рядок; DL – стовпчик; CH – перший рядок курсору; CL – останній рядок курсору
8	Зчитування символу та його атрибуту на поточній позиції	Вхід: AH=8; BH – номер відеосторінки; Вихід: AH – атрибут символу; AL – ASCII –код символу
9	Виведення символу із заданим атрибутом на екран	Вхід: AH=9; BH – номер відеосторінки; AL – ASCII –код символу; BL – атрибут символу; CX – кількість повторень символу
0Ah	Виведення символу з поточним атрибутом на екран	Вхід: AH=0Ah; BH – номер відео сторінки: AL – ASCII –код символу; CX – кількість повторень символу
0Eh	Виведення символу в режимі телетайпу на екран	Вхід: AH=0Eh; BH – номер відеосторінки; AL – ASCII –код символу. Символи з кодами 0Dh, 0Ah,7 інтерпретуються як керуючі. Якщо текст виходить за межі останнього рядка, екран прокручується. В якості атрибуту використовується атрибут поточної позиції
13h	Виведення рядка із заданими атрибутами на екран	Вхід: AH=13h; AL – режим виведення: біт 0 – перемістити курсор в кінець рядка після виведення; біт 1 – рядок містить 2 байти на кожен символ (ASCII-код і атрибут); біти 2-7 – зарезервовані. CX – довжина рядка BL – атрибут (якщо рядок містить лише символи); DH,DL – рядок і стовпчик, з яких почне виводитися рядок; ES:BP – повна адреса рядка в пам'яті

Операційна система також забезпечує програміста набором функцій вводу/виводу, які є доступними через програмне переривання INT 21h. В таблиці 7.3 наведені деякі з них.

Таблиця 7.3. Основні функції вводу/виводу переривання 21h DOS

Номер	Назва функції	Опис функції
1	Зчитування символу з STDIN з луна-супроводженням та очікуванням	Вхід: AH=1; Вихід: AL – ASCII-код символу або 0. Якщо вміст AL дорівнює нулю, другий виклик цієї функції поверне в AL розширений номер ASCII-код символу. При використанні даної функції введений символ автоматично виводиться на дисплей ("луна"), тобто спрямовується в пристрій STDOUT
2	Виведення символу в STDOUT	Вхід: AH=2; DL – ASCII-код символу. При виведені відбувається інтерпретація керуючих символів (0Ah, 0Dh, 7 і т.і.)
6	Зчитування символу STDIN без луна-супроводження та очікування	Вхід: AH=6; DL – FFh. Вихід: ZF=0 і AL – ASCII-код символу, якщо було натиснуто клавішу; ZF=1 і AL=0, якщо не було натиснуто клавішу
7,8	Зчитування символу з STDIN без луна-супроводження та з очікуванням	Вихід: AH=7(8); Вихід: AL – ASCII-код символу
9	Виведення рядка в STDOUT	Вихід: AH=9; DS:DX – адреса рядка, який завершується символом "$"
0Ah	Зчитування рядка з STDIN в буфер	Вхід: AH=0Ah; DS:DX – адреса буфера. Перший байт буфера повинен містити максимальну кількість символів для вводу. В другий байт буфера автоматично записується кількість реально прочитаних символів (до натиснення клавіші Enter; код 0Dh записується також). Власне рядок розміщується з третього байту.
0Bh	Перевірка стану клавіатури	Вхід: AH=0Bh; Вихід: AL=0, якщо не було натиснуто клавішу; AL=FFh, якщо було натиснуто клавішу
0Ch	Очищення буфера і зчитування символу	Вхід: AH=0Ch; AL – номер функції DOS (1, 6, 7, 8, 0Ah) Особливістю функції є ігнорування натиснень клавіш, які були здійснені до її виклику (буфер клавіатури очищується). Вихідні значення залежать від номеру функції, заданого в регістрі AL
40h	Виведення рядка у файл або пристрій	Вхід: AH=40h; BX – ідентифікатор файлу або пристрою (1 – STDOUT, 2 – STDERR); DS:DX – адреса рядка; CX – кількість байтів (довжина рядка)

Порядок роботи з усіма переліченими функціями є простим і звичайним для роботи з функціями переривань: спочатку вхідні дані (якщо вони потрібні) записуються у відповідні регістри, після чого викликається відповідне програмне переривання. Після його виклику, в разі потреби, аналізується вміст вихідних регістрів.

Приклад:

```
…
msg   db    'Input string'
str   db    80,0
      db    80 dup (' ')
…
      ; функція 13h BIOS – виведення рядка на дисплей
mov   ah,13h
      ; режим виведення:
      ; біт 0=1 – перемістити курсор в кінець рядка після виводу
      ; біт 1=0 – рядок містить лише ASCII-коди символів (без атрибутів)
mov   al,1
      ; довжина рядка у байтах
mov   cx,13
      ; атрибут: 1Eh=00011110b – жовті символи на синьому фоні
mov   bl,1Eh
      ; рядок і стовпчик, з яких починається виведення – нуль
xor   dx,dx
      ; рядок зберігається у сегменті даних
      ; запис його сегментної адреси з регістра DS в регістр ES через стек
push ds
pop es
      ; запис в регістр BP ефективної адреси (зміщення) рядка
lea bp,msg
      ; виклик функції 13h переривання 10h BIOS
int 10h
      ; функція 0Ah DOS – зчитування рядка з клавіатури
mov ah,0Ah
      ; запис в регістр DX ефективної адреси буфера для прийому рядка
lea dx,str
      ; виклик функції 0Ah переривання 21h DOS
int 21h
…
```

Методичні вказівки

Завданням даної лабораторної роботи є розробка і реалізація мовою асемблера програм, які використовують у своїй роботі функції консольного вводу/виводу MS DOS і BIOS. При реалізації програм слід дотримуватися лінійного алгоритму. Для здійснення доступу до комірок пам'яті з даними рекомендується застосовувати різні методи адресації.

Кожну програму рекомендується оформлювати в окремому файлі, назва якого містить номер варіанту та номер завдання, наприклад: v11t07.asm (варіант – 11, завдання 7). Файли програм слід розміщувати в окремій директорії.

Приклад виконання роботи

Завдання:

Реалізувати програму, яка зчитує введений користувачем з клавіатури рядок і відразу ж виводить його на дисплей синім кольором на рожевому фоні.

Розв'язання:

1. для вводу рядка з клавіатури застосовуємо функцію 0Ah переривання DOS 21h, а для виводу на дисплей – функцію 13h переривання BIOS 10h.
2. текст програми для виконавчого модуля типа EXE:

```
.model small
.stack 100h
.data
        ; рядок-запрошення для вводу
inpmsg    db'Input string (no more than 80 characters):$'
        ; буфер для зберігання рядка розбито на дві частини
        ; перша частина – два байти: максимальна і реальна довжина
strbuf          db      80, ?
        ; друга частина – масив на 80 байтів: власне буфер для прийому рядка
strsrc          db      80 dup (?)
.code
        mov   ax,@data
        mov   ds,ax
        ; вивід рядка-запрошення на дисплей (функція 9 int 21h)
        mov   ah, 09h
        lea     dx, inpmsg
        int     21h
        ; зчитування рядка з клавіатури (функція 0Ah int 21h)
        mov   ah, 0Ah
        lea     dx, strbuf
        int     21h
        ; підготовка до виклику функції 13h
        mov   ah, 13h
        ; рядок містить лише символи (без атрибутів)
        mov   al, 1
        ; атрибут
        mov   bl, 01010001b
        ; рядок буде виведений в лівий верхній кут екрану
        xor     dx, dx
        ; запис в регістр CX реальної довжини рядка
        ; байт зі зміщенням 1 від strbuf
        ; використовується непряма адресація
```

```
        xor     cx, cx
        lea     bx, strbuf
        inc     bx
        mov     cl, byte ptr [bx]
        ; запис повної адреси рядка в пару регістрів ES:BP
        ; сегментна складова (через стек – з DS в ES)
        push    ds
        pop     es
        ; ефективна складова
        lea     bp, strscr
        int     10h
        ; виклик функції завершення програми
        mov     ah, 4Ch
        int     21h
end
```

Варіанти завдань

Завдання 1

Реалізувати програму, яка зчитує рядок з клавіатури та виводить на дисплей його [1]-й символ кольором [2], а [3]-й символ – кольором [4].

Завдання 2

Реалізувати програму, яка виводить на дисплей символ [5] кольором [6] на фоні [7]; очікує натиснення будь-якої клавіші користувачем (спосіб [8]); змінює кольори символу та фону на [9] і [10] відповідно.

Завдання 3

Реалізувати програму, яка зчитує рядок з клавіатури та виводить його на горизонталь [11] з вирівнюванням [12].

Завдання 4

Реалізувати програму, яка зчитує з клавіатури [13] символів і, після натиснення довільної клавіші виводить рядок з них на позицію екрану [14].

Значення параметрів за варіантами

№	Параметри													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	red	3	cyan	3	white	blue	зл	blue	white	1	л	2	1;10
2	2	green	1	yellow	2	brown	red	бл	red	brown	3	п	3	2;11
3	3	blue	2	magenta	1	magenta	green	зл	green	magenta	2	ц	4	3;12
4	4	red	1	brown	3	yellow	blue	бл	blue	yellow	4	л	2	30;6
5	5	green	2	white	2	cyan	red	зл	red	cyan	5	п	3	25;5
6	1	blue	3	cyan	1	white	green	бл	green	white	7	ц	4	20;4
7	2	red	1	yellow	3	brown	blue	зл	blue	brown	6	л	2	7;15
8	3	green	2	magenta	2	magenta	red	бл	red	magenta	8	п	3	8;16
9	4	blue	3	brown	1	yellow	green	зл	green	yellow	9	ц	4	9;17
10	5	red	1	white	3	cyan	blue	бл	blue	cyan	11	л	2	40;7
11	1	green	2	cyan	2	white	red	зл	red	white	10	п	3	45;8
12	2	blue	3	yellow	1	brown	green	бл	green	brown	12	ц	4	50;9
13	3	red	1	magenta	3	magenta	blue	зл	blue	magenta	13	л	2	4;18
14	4	green	2	brown	2	yellow	red	бл	red	yellow	15	п	3	5;19
15	5	blue	3	white	1	cyan	green	зл	green	cyan	14	ц	4	6;20

зл – з луна-супроводженням;
бл – без луна-супроводження;
л – ліворуч;
п – праворуч;
ц – від центру.

Лабораторна робота № 8

МЕХАНІЗМИ РОЗГАЛУЖУВАННЯ

Мета роботи: ознайомитися з механізмом розгалужування процесів та командами, які для цього використовуються, навчитись програмувати розгалужені алгоритми мовою асемблера, закріпити знання форматів операцій умовного і безумовного переходів.

Теоретичні відомості

В лабораторній роботі № 4 було створено програму, команди якої виконувались послідовно в тому порядку, в якому вони були записані. Така програма представляє собою лінійний процес. В теперішній програмі розглянемо, як організувати процес, який розгалужується. Це такий процес порядок виконання котрого залежить від результатів його виконання. Прикладом процесу, який розгалужується може бути найпростіший алгоритм:

1. Задамо два числа A й B;
2. Якщо A > B, обчислити C=A + B.
3. Інакше, якщо A ≤ B, обчислити C=B – A.

Для організації процесів, які розгалужуються необхідні команди, які б могли передавати керування за адресою команди, яка не знаходиться безпосередньо за командою, яка виконується. Передача керування може здійснюватися вперед для виконання нової групи команд або назад для повторення команд, які вже виконувалися.

Існують три типу переходів (передачі керування).

Перехід типу SHORT передає керування в межах змінення адреси на один байт (– 128 ÷ +127).

Перехід типу NEAR передає керування в межах одного сегменту.

Перехід типу FAR передає керування в інший сегмент.

При переходах SHORT та NEAR змінюється тільки вміст вказівника команд IP. При FAR переході змінюється вміст вказівника команд IP та кодового сегментного регістру CS.

Розглянемо види переходів, які використовуються для передачі керування в програмі.

Безумовний перехід

Безумовний перехід здійснює передачу керування за будь-яких обставинах.

Для виконання безумовного переходу використовується команда JMP.

Наприклад, наведена нижче програма здійснює складання чисел натурального ряду.

```
            mov   ax, 0         ; в AX накопичується сума
            mov   bx, 1         ; в BX формуються числа натурального ряду
label_1:    add   ax, bx
            add   bx, 1
            jmp   label_1
```

Команда JMP label_1 передає керування команді з міткою label_1. Зверніть увагу: при вказуванні мітки в команді JMP двокрапка після імені не ставиться.

Мітку можна записувати в одному рядку з командою

```
label_1:    add    ax, bx
```

або в окремому рядку

```
label_1:
            add    ax, bx
```

Заданий цикл не має виходу й призводить до нескінченного виконання – такі цикли звичайно не використовуються.

Розглянемо листінг заданої програми.

```
Turbo Assembler  Version 2.0        12/16/99    05:35:45                Page 1
exam6.asm

1     0000                          codesg      segment    para  'Code'
2                                   assume      cs:codesg,ss:codesg
3                                               org        100h
4     0199                          main        proc       near
5     0100  B8    0000             mov         ax, 0      ; в AX накопичується сума
6     0103  BB    0001             mov         bx, 1      ; в BX формуються числа натурального ряду
7     0106  03    C3               A20:        add        ax,bx
8     0108  83    C3    01                    add        bx,1
9     010B  EB    F9                          jmp        A20
10    010D                          main        endp
11    010D                          codesg      endp
12                                  end main
```

В нашому прикладі операнд команди JMP (мітка label_1) відповідає - 7 байт від команди, наступною за JMP, в чому можна переконатися по об'єктному коду команди EBF9. EB – машинний код для короткого переходу JMP, а F9 – від'ємне зміщення (– 7). Команда JMP додає F7 до командного вказівника (IP), який містить адресу команди, наступної за JMP (010D). В результаті складання виходить адреса переходу (0106). Операнд в команді JMP для переходу вперед буде мати позитивне значення.

Команда JMP для переходу в межах від –128 до +127 байт має тип SHORT. Асемблер генерує в цьому випадку однобайтовий операнд в межах від 00 до FF. Команда JMP, що перевищує ці межі, отримує тип FAR, для котрого генерується інший машинний код та двобайтовий операнд. Транслятор в першому перегляді вихідної програми визначає довжину кожної команди. Якщо до моменту перегляду команди JMP транслятор вже обчислив значення операнда (при переході назад)

```
A50:        …
            …
            jmp A50
```

то він генерує двобайтову команду. Якщо транслятор ще не обчислив значення операнда (при переході вперед)

```
        jmp A90
        …
A90:    …
```

то він не знає типу переходу NEAR або FAR та автоматично генерує трьохбайтову команду. Для того, щоб вказати транслятору на необхідність генерації двобайтової команди, необхідно використати оператор SHORT:

```
        jmp short   A90
        …
A90:    …
```

Умовний перехід

Умовний перехід представляє собою двоступінчатий процес: спочатку перевіряється умова, а потім здійснюється перехід, якщо умова виконується, або продовження роботи, якщо вимога не виконується.

Команди умовного переходу здійснюють передачу керування в залежності від результату виконання попередньої команди.

Команди умовного переходу використовують єдиний операнд, який містить адресу (мітку), на яку повинен буди здійснено перехід. Відстань від команди переходу до заданої адреси повинно бути менш 128 байт (перехід типу SHORT).

Команди умовного переходу використовують стан одного або декілька прапорців (вміст прапорцевого регістру) в якості умов переходу. Таким чином, будь-яка команда, яка встановлює прапорець за певною умовою, може бути командою перевірки умови. Частіше для цієї мети використовуються команди CMP та TEST. Командою переходу може бути будь-яка з 31 команди умовного переходу.

Прапорцевий регістр зберігає у вигляді бітових полів результати виконання операцій (ознаки нульового, негативного результату, переповнення розрядної сітки і т.і.). Прапорці зберігають своє значення до тих пір, доки інша команда не змінить їх.

Формат прапорцевого регістру

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X
				F	F	F	F	F	F		F		F	

X відмічені невживані біти.

CF (Carry Flag) – прапорець переносу. Містить значення переносів (0 або 1) із старшого розряду при арифметичних операціях та деяких операціях зсуву та циклічного зсуву.

PF (Parity Flag) – прапорець парності. Перевіряє молодші 8 бітів операцій над даними. Непарне число бітів приводить до установки цього прапору в “0”, парне – в “1”. Не слід плутати цей прапорець з бітом контролю парності.

AF (Auxiliary Carry Flag) – допоміжний прапорець переносу. Встановлюється в 1, якщо арифметична операція призводить до переносу четвертого справа біту (біт номер 3) в регістровій однобайтній команді. Цей прапорець має відношення до арифметичних операцій над символами коду ASCII та до десятковим упакованим полям.

ZF (Zero Flag) – прапорець нуля. Встановлюється в якості результату арифметичних команд та команд порівняння. При нульовому результаті – 1, при ненульовому – 0.

SF (Sign Flag) – прапорець знаку. Встановлюється в залежності від знаку результату (старшого біту) після арифметичних операцій. При позитивному результаті – 0, при негативному – 1.

TF (Trap Flag) – прапорець трасування. Якщо цей прапорець встановлено в одиничний стан, то процесор переходить в режим покрокового виконання команд.

IF (Interrupt Flag) – прапорець переривання. При нульовому стані цього прапорецьу переривання заборонені, при одиничному – дозволені.

DF (Direction Flag) – використовується в рядкових операціях для визначення напрямку передачі даних.

OF (Overflow Flag) – прапорець переповнення. Фіксує арифметичне переповнення, тобто перенесення в (із) старшого (знакового) розряду при знакових арифметичних операціях.

Наприклад, команда CMP порівнює два операнди. Операнди в процесі виконання команди не змінюються. Команда використовується для перевірки співвідношень рівно, або не рівно, більше, менше, більше або рівно, менше або рівно. Впливає на прапорці AF, CF, OF, PF, SF, ZF. За суттю команда CMP співпадає з командою SUB, за виключенням того, що не змінює операнд-приймальник. Відсутня необхідність перевіряти всі прапорці, які встановлюються командою CMP окремо. В наступному прикладі перевіряється, містить чи ні регістр BX нульове значення:

```
        cmp     bx, 00      ; порівняння BX з 0
        jz      B50         ; перехід на B50, якщо 0
        …
        …
B50:    …                   ; точка переходу при BX=0
```

Якщо BX містить нульове значення, команда CMP встановлює прапорець нуля ZF в одиничний стан та, можливо змінює (або ні) інші прапорці. Команда JZ (перехід, якщо нуль) перевіряє тільки прапорець ZF. При одиничному значенні ZF, що означає нульовий результат, команда передає керування на адресу, яка вказана в її операнді, тобто на мітку B50.

Необхідно пояснити характер використання команд умовного переходу. Типи даних над котрими здійснюються арифметичні операції та операції порівняння, визначають, якими командами слід користуватися: знаковими або беззнаковими. Беззнакові дані використовують всі біти як біти даних. Характерним прикладом є символьні рядки та натуральні числа. В знакових

даних самий лівий біт представляє собою знак, причому, якщо його значення дорівнюється нулю, то число позитивне, а якщо дорівнюється одиниці – негативне.

В якості прикладу припустимо, що регістр AL містить 11000110, а BL – 00010110. Команда

```
cmp al, bl
```

порівнює вміст регістрів AL та BL. Якщо дані розглядаються як знакові, то значення в BL більше, якщо як без знакові, то значення в AL більше.

Команди переходу для беззнакових даних

Мнемоніка	Опис	Прапорці, які перевіряються
JE/JZ	Перехід, якщо дорівнює/нуль	ZF
JNE/JNZ	Перехід, якщо не дорівнює /не нуль	ZF
JA/JNBE	Перехід, якщо вище/не нижче або дорівнює	ZF,CF
JAE/JNB	Перехід, якщо вище або дорівнює /не нижче	CF
JB/JNAE	Перехід, якщо нижче/не вище або дорівнює	CF
JBE/JNA	Перехід, якщо нижче або дорівнює /не вище	CF, AF

Будь-яку перевірку можна кодувати одним з двох мнемонічних кодів. Наприклад, JB та JNAE генерує один і той же об’єктний код, хоча позитивну перевірку JB легше зрозуміти, ніж негативну JNAE.

Команди переходу для знакових даних

Мнемоніка	Опис	Прапорці, які перевіряються
JE/JZ	Перехід, якщо дорівнює /нуль	ZF
JNE/JNZ	Перехід, якщо не дорівнює /не нуль	ZF
JG/JNLE	Перехід, якщо більше/не менше або дорівнює	ZF,CF, OF
JGE/JNL	Перехід, якщо більше або дорівнює /не менше	CF, OF
JL/JNGE	Перехід, якщо менше/не більше або дорівнює	CF, OF
JLE/JNG	Перехід, якщо менше або дорівнює /не більше	CF, AF

Команди переходу для умови рівно або нуль (JE/JZ) та рівно або не нуль (JNE/JNZ) присутні в обох списках для знакових та без знакових даних. Стан рівно/нуль відбувається поза залежністю від присутності знаку.

Якщо використовувати команди умовного переходу, можна організувати розгалужений процес, який реалізує алгоритм, який наведено на початку лабораторної роботи.

```
 1 0000                         codesg      segment     para   'code'
 2                                          assume cs:codesg,ds:codesg,ss:codesg
 3                                          org 100h
 4 0100      EB 07 90           main :      jmp    begin
 5 0103      006                a           dw 6
 6 0105      005                b           dw 5
 7 0107      ???                c           dw ?
 8                              ;нехай змінні a,b,c знаходяться відповідно в
 9                              ;змінних в пам'яті A, B, C
10 0109                         begin proc  near
11 0109      A1     0103r      mov  ax, A          ;заносимо A в регістр AX
12 010C      3B 06 0105r       cmp  ax, B          ;порівняємо AX з B
13 0110      7F 09             jg L1               ;якщо AX більше B
14                                               ;переходимо на L1
15 0112      29 06 0105r       sub  B, ax          ;інакше віднімемо від B AX
16 0116      A1 0105r          mov  ax, B
17 0119      EB 04             jmp  short L2       ;перейдемо на L2
18 011B      03 06 0105r       L1:  add   ax, B    ;додаємо B до AX
19 011F      A3 0107r          L2:  mov   C, ax    ;результат заносимо в C
20 0122      C3                     ret
21 0123                         begin endp
22 0123                         codesg ends
23                              end main
```

Програму можна мінімізувати, наприклад, якщо замість операції порівняння відразу виконати операцію віднімання, яка встановлює ті ж самі прапорці.

Спеціальні арифметичні перевірки

Мнемоніка	Опис	Прапорці, які перевіряються
JS	Перехід, якщо присутній знак (негативно)	SF
JNS	Перехід, якщо нема знаку (позитивне)	SF
JC	Перехід, якщо присутній переніс (аналогічно JB)	CF
JNC	Перехід, якщо нема переносу	CF
JO	Перехід, якщо присутнє переповнення	OF
JNO	Перехід, якщо відсутнє переповнення	OF
JP/JPE	Перехід, якщо паритет парний	PF
JNP/JPO	Перехід, якщо паритет непарний	PF

Методичні вказівки

Завданням даної лабораторної роботи є розробка і реалізація мовою асемблера програм, які працюють за розгалуженими алгоритмами.

Перед виконанням роботи слід уважно вивчити завдання і дати відповіді на наступні запитання:

1. На яку максимальну кількість байт можна здійснити перехід командами JMP та командами умовного переходу.
2. В чому різниця організації розгалуження процесів для знакових та без знакових даних.
3. На які прапорці впливають наступні події:
   - відбувся перенос;
   - результат негативний;
   - відбулося переповнення;
   - результат нульовий.
4. Які функції вводу і виводу будуть використовуватись в програмах.
5. Які операції умовного переходу найбільш зручно використовувати в програмі для найбільш компактної організації розгалуженого коду.

В програмі, яка реалізується слід обов’язково передбачити контроль правильності вхідних даних.

Приклад виконання роботи

Завдання:

Реалізувати програму, яка зчитує з клавіатури три одно розрядних додатних цілих числа та виводить на дисплей найбільше з них.

Розв'язання:

```
.model      tiny
.code
        org     100h
start:
        jmp begin
        msg1        db      0Ah, 0Dh, 'Input first number (1 digit):$'
        msg2        db      0Ah, 0Dh, 'Input second number (1 digit):$'
        msg3        db      0Ah, 0Dh, 'Input third number (1 digit):$'
        msg4        db      0Ah, 0Dh, 'Maximal number is:$'
        errmsg      db      0Ah, 0Dh, 'Incorrect number: must be [0..9]:$'
        maxnum      db      ?
begin:
        mov   ah, 09h       ; вивід запрошення до вводу першого числа
        lea   dx, msg1
        int 21h
        mov   ah, 01h       ; функція вводу символу з клавіатури
        int   21h
        cmp   al, '0'       ; перевірка коректності символу
        jb    not_a_number
        cmp   al, '9'
        ja    not_a_number
        mov   maxnum, al ; запис першого числа в змінну
        mov   ah, 09h       ; вивід запрошення до вводу другого числа
```

```
        lea     dx, msg2
        int     21h
        mov     ah, 01h         ; функція вводу символу з клавіатури
        int     21h
        cmp     al, '0'         ; перевірка коректності символу
        jb      not_a_number
        cmp     al, '9'
        ja      not_a_number
        cmp     al, maxnum ; порівняння другого числа з першим, збереженим раніше
        jb      next1
        mov     maxnum, al ; якщо друге число більше першого – замістити ним перше
next1:
        mov     ah, 09h         ; вивід запрошення до вводу третього числа
        lea     dx, msg3
        int     21h
        mov     ah, 01h         ; функція вводу символу з клавіатури
        int     21h
        cmp     al, '0'         ; перевірка коректності символу
        jb      not_a_number
        cmp     al, '9'
        ja      not_a_number
        cmp     al, maxnum ; порівняння третього числа з найбільшим, збереженим раніше
        jb      next2           ; якщо третє число більше найбільшого – замістити ним найбільше
        mov     maxnum, al
next2:
        mov     ah, 09h
        lea     dx, msg4
        int     21h
        mov     ah, 02h         ; вивід найбільшого числа на дисплей
        mov     dl, maxnum
        int     21h
        jmp     finish
not_a_number:
        mov     ah, 09h
        lea     dx, errmsg
        int     21h
finish:
        ret
end start
```

Варіанти завдань

1. Реалізувати програму, яка додає до порядкового номера студента згідно списку журналу значення поточної дати, якщо день народження студента більше ніж 15 інакше додає до порядкового номера студента згідно списку журналу значення поточного часу.
2. Реалізувати програму, яка зчитує з клавіатури два рядки і виводить на дисплей повідомлення "First", якщо кількість символів у першому рядку більша за кількість символів у другому і останній символ у першому рядку має код, більший за код останнього символу у другому рядку, і повідомлення "Second", якщо жодна з перелічених умов не виконується.
3. Реалізувати програму, яка зчитує з клавіатури два трьохзначних десяткових числа і виводить на дисплей те з них, сума цифр якого найбільша.
4. Реалізувати програму, яка зчитує з клавіатури рядок з чотирьох символів, перевіряє, читається він однаково зліва направо і навпаки, чи ні (рядок-паліндром) і виводить на екран відповідне повідомлення.
5. Реалізувати програму, яка зчитує з клавіатури десяткове число в діапазоні від 0 до 15 і виводить його на дисплей у 16-й системі числення.
6. Реалізувати програму, яка виводить на дисплей назву поточного дня тижня, номер тижня в місяці, назву місяця та номер кварталу.
7. Реалізувати програму, яка виводить на дисплей поточний час доби (ранок, день, …) і номер чверті години.
8. Реалізувати програму, яка зчитує з клавіатури трьохзначне ціле число і виводить його на екран у текстовому вигляді (113 – "сто тринадцять").
9. Реалізувати програму, яка "згадує" три одно розрядні цілі числа і дає користувачу змогу вгадати їх; на екран виводиться повідомлення про кількість вгаданих чисел та результат гри (перемога, поразка).

Лабораторна робота № 9

ЗСУВ ТА ЦИКЛІЧНИЙ ЗСУВ РОЗРЯДІВ

Мета роботи: вивчити механізм організації зсуву та циклічного зсуву розрядів, навчитись використовувати зсув в алгоритмах мовою асемблера.

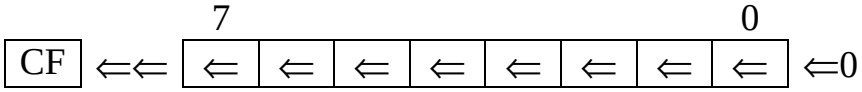
Теоретичні відомості

Асемблер має команди для зсуву та циклічного зсуву розрядів в байті або слові. Розряди можуть зсуватися вправо (в сторону молодших розрядів) або вліво (в сторону старших розрядів). Значення розряду, який біло висунуто за кінець оперну попадає в прапорець переносу.

Команди зсуву та циклічного зсуву розділяються на дві групи. Логічні команди зсувають операнд та не враховують його знак, вони використовуються для дій над числами без знаку або над нечисловими значеннями. Арифметичні команди зберігають старший знаковий біт оперну, вони використовуються для дій над числами з знаком.

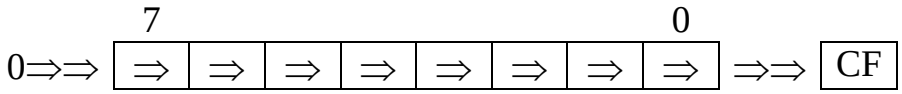
На наступному рисунку показано дію 8 команд зсуву-циклічного зсуву для 8-розрядних операндів.

SHL (логічний беззнаковий зсув вліво)



SHL:	CF	значення операндів	заповнення
до операції		10110111	0
після операції	1	01101110	

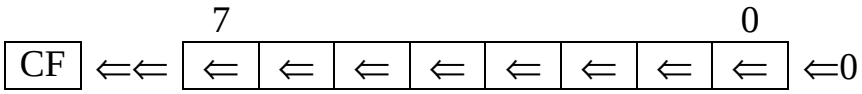
SHR (логічний беззнаковий зсув вправо)



SHR:	заповнення	значення операндів	CF
до операції	0	10110111	
після операції		01011011	1

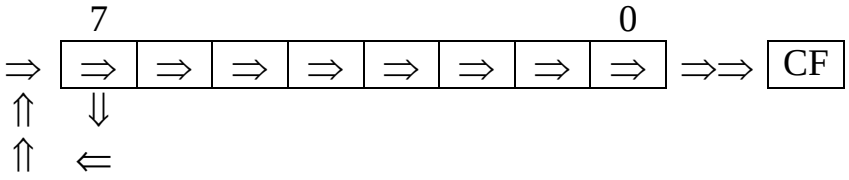
При будь-якому логічному зсуві на місце, що звільнилося, завжди записується 0. Прапорець переповнювання (OF) виставляється, якщо в результаті операції операнд змінив знак.

SAL (арифметичний зсув вліво)



SAL:	CF	значення операндів	заповнення
до операції		10110111	0
після операції	1	01101110	

SAR (арифметичний зсув вправо)



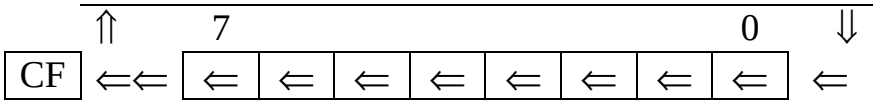
SAR:	заповнення	значення операндів	CF
до операції	sign(1)	10110111	
після операції		11011011	1

Операція арифметичного зсуву вліво SAL діє абсолютно аналогічно операції SHL (також на порожнє місце справа зсувається 0). Операція арифметичного зсуву вправо SAR діє інакше: у позицію, що звільняється, зліва копіюється знаковий біт початкового операнда.

При виконанні операцій циклічного зсуву прапорець перенесення (CF) завжди містить значення останнього висунутого біта. Існують наступні види операцій циклічного зсуву:

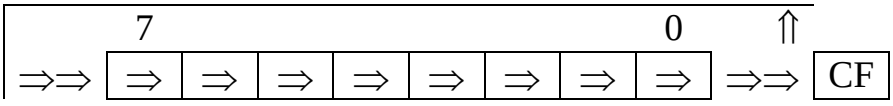
- ROL – циклічний зсув вліво на один біт.
- ROR – циклічний зсув вправо на один біт.
- RCL – циклічний зсув вліво на один біт за участю прапора перенесення.
- RCR – циклічний зсув вправо на один біт за участю прапора перенесення.

ROL – циклічний зсув вліво на один біт.



ROL:	CF	значення операндів	заповнення
до операції		10110111	sign(1)
після операції	1	01101111	

ROR – циклічний зсув вправо на один біт.

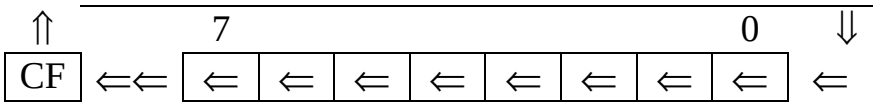


ROR:	заповнення	значення операндів	CF
до операції	right bit(1)	10110111	
після операції		11011011	1

Операції ROL і ROR (аналогічно з командам зсуву SAL, SAR, SHL, SHR) діють на прапорець переповнювання (OF) контролюючи зміну знаку операнда. Так само встановлюються прапорці парності (PF), знаку (SF) і нуля (ZF). Прапорець перенесення з молодшої тетради (AF) не визначається і просто скидається в 0.

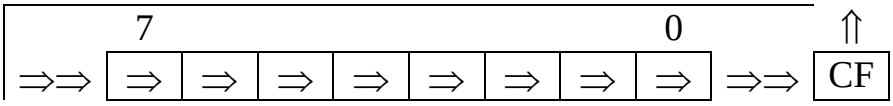
У операціях RCL і RCL в зсуві бере участь прапорець перенесення (CF). Біт що висувається з операнду, заноситься в прапорець перенесення (CF) після того, як значення (попереднє) CF поступить в позицію, що звільнилася.

RCL – циклічний зсув вліво на один біт за участю прапора перенесення.



RCL:	CF	значення операндів	заповнення
до операції	0	10110111	CF(0)
після операції	1	01101110	

RCR – циклічний зсув вправо на один біт за участю прапора перенесення.



RCR:	заповнення	значення операндів	CF
до операції	CF(0)	10110111	0
після операції		01011011	1

Команди SHL та SHR абсолютно однакові.

В операнді-приймальнику команд зсуву знаходиться значення, яке підлягає зсуву. Після виконання команди в ньому знаходиться результат. В операнді-джерелі повинно знаходитися число, на яке необхідно зсунути розряди. Дане число може бути задане як безпосереднє значення “1” або як вміст регістру CL.

Команда SAR зберігає знак операнда, репродукуючи його при виконанні зсуву. Команда SAL не зберігає знак, але заносить “1” в прапорець переповнення OF у випадку зміни знака операнда.

Нехай AL містить 0B4h, а CF=1

```
                ;       AL=1011   0100b
sal     al, 1   ;       AL=0110   1000        CF=1
sar     al, 1   ;       AL=1101   1010        CF=0
shl     al, 1   ;       AL=0110   1000        CF=1
shr     al, 1   ;       AL=0101   1010        CF=0
rol     al, 1   ;       AL=0110   1001        CF=1
ror     al, 1   ;       AL=0101   1010        CF=0
rcl     al, 1   ;       AL=0110   1001        CF=1
rcr     al, 1   ;       AL=1101   1010        CF=0
```

Зсув вправо на 1 еквівалентний діленню на 2, зсув вправо на 1 – множенню на 2. Дана особливість може бути використана для множення та

ділення констант, які часто зустрічаються. Наприклад, 2^n. При подвійному зсуві вліво відбувається множення на 4, при трійному – на 8 і т.д.

SHR використовується для ділення беззнакових чисел, SAR – для знакових. Множення за допомогою зсуву є однаковим для знакових та беззнакових чисел, тому можна використовувати як SAL, так і SHL.

```
shl     ax, cl ; помножити число без знаку на 4
sal     ax, cl ; помножити число зі знаком на 4
shr     ax, cl ; поділити число без знаку на 4
sar     ax, cl ; поділити число зі знаком на 4
```

Кожна з цих команд з урахуванням завантаження CL виконується в 6-8 раз скоріше відповідної команди множення або ділення.

В разі необхідності зсуву значення, яке занадто велике для регістру можна зсувати кожну частину окремо. Розряди, які зсуваються передаються через прапорець переносу. Для передачі переносу із першого регістру в другий повинні використовуватись команди RCR або RCL.

Наприклад:

```
shl     ax, 1  ; 32-розрядне число в регістрах AX та DX
rcl     dx, 1  ; множимо на 2
```

Для реалізації завдання вам необхідно навчитися умножати і ділити константу на ступінь числа 2. Виконувати подібні операції доцільно с з використанням зсуву.

Варіанти завдань

1. Використовуючи команди SHL, SAL, ROL, RCL здійснити зсув регістра загального призначення (відповідно до таблиці) на кількість розрядів, яке відповідає молодшому розряду номера студента згідно списку в журналі.
2. Використовуючи команди SHR, SAR, ROR, RCR здійснити зсув регістра загального призначення (відповідно до таблиці) на кількість розрядів, яке відповідає молодшому розряду номера студента згідно списку в журналі.
3. Порівняти результати. Зробити висновки.

Таблиця 9.1. Регістри, зсув вмісту яких необхідно здійснити

№ з/п	Регістр
1.	AX
2.	BX
3.	CX
4.	DX

Лабораторна робота № 10

ЦИКЛІЧНІ ПРОЦЕСИ МОВОЮ АСЕМБЛЕРА

Мета роботи: вивчити механізм організації циклічних процесів та команди асемблера, які використовуються з цією метою, навчитись програмувати циклічні алгоритми мовою асемблера.

Теоретичні відомості

При створенні програми може з’явитися необхідність повторити деяку послідовність команд задану кількість разів. Такий процес називається циклічним процесом.

Відомо, що організація будь-якого циклу з кінцевою, але не обов’язково заздалегідь відомою кількістю кроків в програмі здійснюється у два етапи:

1) аналіз деякого логічного виразу – умови роботи циклу;
2) умовний перехід на початок тіла циклу залежно від значення логічного виразу.

Як і мови високого рівня, мова асемблера дозволяє організувати циклічні алгоритми двох видів: цикли "ДО" і цикли "ПОКИ". Нагадаємо, що в циклах першого виду аналіз умови роботи здійснюється після виконання, а в циклах другого виду – перед виконанням тіла циклу.

Найбільш простим та ефективним способом організації циклів у програмах мовою асемблера є застосування операції порівняння та операцій умовного (і безумовного) переходів. Далі наведені два відповідних "шаблони" циклів, реалізованих мовою асемблера.

```
…
cycle_label:
…
; тіло циклу
…
; аналіз умови роботи циклу
cmp …
; перехід на початок циклу
; або вихід з циклу
; залежно від результатів
jxx cycle_label
…
```

Цикл "ДО"

```
…
cycle_label:
; аналіз умови роботи циклу
cmp …
; виконання тіла циклу
; або вихід з циклу
; залежно від результатів
jxx end_of_cycle
…
; тіло циклу
…
jmp cycle_label
end_of_cycle:
```

Цикл "ПОКИ"

Якщо на момент перевірки циклової умови прапорці встановлені так, як це потрібно для здійснення перевірки, операцію порівняння CMP можна не виконувати.

Приклад: заданий масив однобайтових чисел у кількості 7. Збільшити кожне число на одиницю.

```
…
array  db      12, 33, 14, 25 ,16, 17, 19
…
lea     bx, array
mov   cx, 7
cycle:
inc     byte ptr [bx]
inc     bx
dec     cx
; після виконання операції dec прапори встановлюються автоматично
; тому порівняння CMP CX,0 є цілком зайвим
jnc     cycle
```

Для організації циклів із заздалегідь відомою кількістю кроків в якості циклової змінної звичайно прийнято використовувати регістр CX (Counter). Специфіка цього регістра дозволяє використовувати спеціальні операції процесора, призначені для більш легкої реалізації циклів (таблиця 10.1).

Таблиця 10.1. Циклові операції процесора i80386

Позначення і формат інструкції		Призначення
jcxz jecxz	label	Умовний перехід на мітку label, якщо вміст регістра CX/ECX – нуль
loop	label	Зменшення ECX на 1 і близький перехід на вказану мітку, якщо ECX не дорівнює нулю
loope loopz loopne loopnz	label	Зменшення ECX на 1 близький перехід на вказану мітку, якщо: - прапорець ZF встановлений (loope/loopz); - прапорець ZF скинутий (loopne/loopnz)

При використанні команди LOOP кількість циклів (кількість повторень ділянки програми) задається в регістрі CX. В кожному циклі команда LOOP автоматично зменшує вміст CX на 1. Доки значення CX не буде рівним 0, керування передається за адресою, яка вказана в операнді команди LOOP. Коли CX стане рівним нулю, керування передається на наступну за LOOP команду.

Приклад:

```
 1 0000                 codesg      segment     para   'code'
 2                                  assume cs:codesg, ds:codesg, ss:codesg
 3                                  org 100h
 4 0100                 begin       proc near
 5 0100      B8 0001                mov   ax, 1
```

```
 6 0103    BB 0001              mov   bx, 1
 7 0106    B9 000A              mov   cx, 10
 8 0109                cycle_label:
 9 0109    40                   inc   ax      ; ax+1
10 010A    03 D8                add   bx, ax
11 010C    E2FB                 loop  cycle_label
12 010E    C3                   ret
13 010F                begin     endp
14 010F                codesg    ends
15                     end begin
```

Програма, яку наведено в прикладі, виконує ділянку програми, яка знаходиться між міткою A20 та командою LOOP десять разів (число 10 задається **ДО** початку першого циклу в регістрі CX).

Відстань від команди LOOP до мітки, яка задається в якості її операнда, не повинна перевищувати –128 ÷ +127 байт (перехід типу SHORT).

Додатково існують два різновиди команди LOOP – LOOPE (LOOPZ) та LOOPNE (LOOPNZ). Обидві команди також зменшують значення CX на 1. Команда LOOPE (LOOPZ) передає керування за адресою операнда, якщо регістр має ненульове значення та прапорець нуля встановлено (ZF=1). Команда LOOPNE (LOOPNZ) передає керування за адресою операнда, якщо регістр має ненульове значення та прапорець нуля скинутий (ZF=0).

Приклад:

```
 1 0000                codesg    segment  para  ‘code’
 2                               assume cs:codesg, ds:codesg, ss:codesg
 3                               org 100h
 4 0100                begin     proc near
 5 0100    B8 0001              mov   ax, 1
 6 0103    BB 0001              mov   bx, 1
 7 0106    B9 000A              mov   dx,10
 8 0109                cycle_label:
 9 0109    40                   inc   ax      ; ax+1
10 010A    03 D8                add   bx, ax
11 010C    4A                   dec   dx      ; dx-1
12 010D    75 FA                jnz   cycle_label
13 010F    C3                   ret
14 0110                begin     endp
15 0110                codesg    ends
16                              end begin
```

Методичні вказівки

Завданням даної лабораторної роботи є розробка і реалізація мовою асемблера програм, які працюють за циклічними алгоритмами.

При проектуванні алгоритмів необхідно вибирати такі циклічні схеми, які забезпечуватимуть максимальну швидкодію коду; широко застосовувати

регістри загального призначення в якості параметрів циклу; мінімізувати кількість обслуговуючих цикл операцій. Слід пам’ятати, що циклічні алгоритми яскраво "висвітлюють" неоптимізовані ділянки коду тіла циклу через накопичення затримок їх виконання.

В розв’язаннях задач повинно бути передбачене:

- ініціалізація елементів масивів/матриць випадковими елементами;
- вивід вихідного стану масивів/матриць на екран;
- вивід результатів роботи програми на екран.

Рекомендується використовувати одно байтові масиви/матриці, а усі операції над ними виконувати засобами цілочисельної арифметики.

Для адресації елементів матриць необхідно використовувати подвійне індексування, а перетворення індексів здійснювати окремо.

Рекомендується широко застосовувати різноманітні методи адресації.

Контрольні питання

1. Який тип переходу допускає команда LOOP?
2. Яким чином можна організувати вкладені цикли?
3. Де буде знаходитись лічильник циклів при використанні команд умовного переходу для організації циклів?

Приклад виконання роботи

Завдання:

поділити на два кожен парний елемент масиву A(N), а до кожного непарного елементу цього масиву додати одиницю.

Розв'язання:

```
.model      tiny
.stack      100h
.data
        ; довжина масиву масив
        N       dw      8
        ; масив
        A       db      2, 3, 4, 8, 9, 7, 1, 5
        msg1    db      'Source array:', 0Ah, 0Dh, '$'
        msg2    db      'Result array:', 0Ah, 0Dh, '$'
        msg3    db      0Ah, 0Dh, '$'
.code
        mov     ax, @data
        mov     ds, ax
        mov     ah, 09h         ; вивід першого повідомлення на екран
        lea     dx, msg1
        int     21h
        mov     cx, word ptr N      ; завантаження лічильника кількістю елементів
        mov     bx, offset A ; в регістр BX – зміщення масиву (його нульового елементу)
cycle1:
        mov     ah, 02h         ; вивід чергового елементу масиву
```

```
        mov   dl, byte ptr [bx]
        add   dl, 30h
        int   21h
        mov   ah, 02h       ; вивід проміжку
        mov   dl, ' '
        int   21h
        inc   bx
        loop cycle1
        mov   cx, word ptr N
        mov   bx, offset A
cycle2:
        mov   al, byte ptr [bx]
        mov   dl, al
        shr   al, 1   ; перевірка, чи є черговий елемент парним, якщо при зсуві
        jc    not_par       ; при зсуві праворуч "вилетів" біт – елемент непарний
        mov   byte ptr [bx], al     ; елемент парний – запис частки назад
        jmp   next_byte
not_par:
        dec   dl      ; елемент непарний – зменшення на одиницю і запис назад
        mov   byte ptr [bx], dl
next_byte:
        inc   bx
        loop  cycle2
        mov   ah, 09h
        lea   dx, msg3
        int   21h
        mov   cx, word ptr N      ; вивід результуючого масиву
        mov   bx, offset A
cycle3:
        mov   ah, 02h
        mov   dl, byte ptr [bx]
        add   dl, 30h
        int   21h
        mov   ah, 02h
        mov   dl, ' '
        int   21h
        inc   bx
        loop  cycle3
        mov   ah, 4Ch
        int   21h
end
```

Варіанти завдань

1.1. Заданий масив D (N). Знайти найбільший і найменший елементи в ньому і поміняти їх місцями.

1.2. Обчислити і записати в одновимірний масив суму додатних елементів кожного стовпчика матриці A (M, N).

2.1. Переписати в масив Y додатні елементи з масиву X (Y).

2.2. Обчислити суму і кількість елементів матриці B (N, N), що знаходяться під головною діагоналлю.

3.1. Переписати в масив X додатні, а в масив Y – від'ємні елементи з масиву A (N).

3.2. Записати на місце додатних елементів матриці D (K, K) нулі.

4.1. Обчислити середнє арифметичне елементів масиву Y (N).

4.2. Кожен нульовий елемент матриці P (K, K) замінити на діагональний елемент, що знаходиться у тому ж стовпчику.

5.1. Розташувати в масиві R спочатку додатні, а потім – від'ємні елементи масиву Z (N).

5.2. Обчислити і записати в одномірний масив суми елементів кожного рядка матриці A (M, N).

6.1. Визначити суму елементів масиву K (N), що є парними.

6.2. В матриці A (K, L), де L – парне число, поміняти місцями стовпчики: перший з останнім, другий – з передостаннім і т. д.

7.1. Обчислити суму з кожного другого елемента масиву X (N).

7.2. Додати до елементів останнього рядка матриці Z (M, M) відповідні елементи всіх інших рядків.

8.1. Визначити найменший елемент масиву C (N) та його індекс.

8.2. Обчислити суму і кількість елементів матриці W (N, N), що знаходяться над головною діагоналлю.

9.1. Визначити найбільший елемент масиву D (N) та його індекс.

9.2. Обчислити суму та кількість від'ємних діагональних елементів матриці R (M, M).

10.1. В масиві X (N) кожен елемент, починаючи з третього замінити на напівсуху двох попередніх.

10.2. Записати на місце від'ємних елементів матриці A (M, N) нулі.

11.1. Обчислити кількість нульових елементів масиву B (K).

11.2. Знайти найбільший та найменший елементи і матриці C (M, N) та поміняти їх місцями.

12.1. Обчислити суму додатних елементів масиву P (N).

12.2. Транспонувати матрицю F (M, M).

13.1. Обчислити суму та кількість елементів масиву X (K), які менші за його перший елемент.

13.2. Для матриці Z (M, N) знайти і записати в одномірний масив кількості нульових елементів у кожному рядку.

14.1. Переписати елементи масиву X (N), які стоять на позиціях з парними номерами в масив A, а ті, що стоять на непарних позиціях – в масив B.

14.2. Знайти в кожному стовпчику матриці P (N, N) найбільший елемент і поміняти його місцями з відповідним діагональним елементом.

15.1. В масиві C (N) кожен нульовий елемент замінити на найближчий до нього ліворуч (останній – для першого, якщо той є нульовим).

15.2. Знайти найбільший і найменший елементи матриці A (M, N) і поміняти їх місцями.

Лабораторна робота № 11

ВИКОРИСТАННЯ ОПЕРАЦІЙ ЦІЛОЧИСЕЛЬНОЇ АРИФМЕТИКИ ТА ЛОГІЧНИХ ОПЕРАЦІЙ

Мета роботи: навчитись програмувати цілочисельні розрахунки мовою асемблера, закріпити знання арифметичних і логічних операцій.

Теоретичні відомості

Базова система вводу/виводу надає програмістові широкий спектр функцій, призначених для зчитування даних з клавіатури та виводу на дисплей, які є доступними через програмні переривання 16h і 10h. Перелік основних функцій приведений в таблиці 11.1 та 11.2.

Таблиця 11.1. Арифметичні та логічні операції

<table>
<tr><th colspan="4">Позначення і формат інструкції</th><th>Призначення</th></tr>
<tr><td rowspan="2">add
adc</td><td colspan="2">r8, 16, 32</td><td>op8, 16, 32
r8, 16, 32
mem8, 16, 32</td><td rowspan="2">Складання двох операндів (ADD). Складання двох операндів з урахуванням переносу із старшого розряду – CF (ADC). Перший і другий операнд є доданками; результат поміщується в перший операнд.</td></tr>
<tr><td colspan="2">mem8, 16, 32</td><td>op8, 16, 32
r8, 16, 32</td></tr>
<tr><td rowspan="2">sub
sbb</td><td colspan="2">r8, 16, 32</td><td>op8, 16, 32
r8, 16, 32
mem8, 16, 32</td><td rowspan="2">Віднімання двох операндів (SUB). Віднімання двох операндів з урахуванням займу – CF (SBB). Перший операнд виступає в ролі зменшуваного, другий – від’ємника. Результат поміщується в перший операнд.</td></tr>
<tr><td colspan="2">mem8, 16, 32</td><td>op8, 16, 32
r8, 16, 32</td></tr>
<tr><td rowspan="9">imul</td><td colspan="3">r8, 16, 32
mem8, 16, 32</td><td rowspan="9">Цілочисельне множення із знаком.
1. Однооперандний формат:
• якщо операнд-співмножник, заданий явно має розмір 1 байт, другий співмножник повинен розміщуватись в регістрі AL;
• якщо операнд-співмножник має розмір 2 байти, в регістрі AX;
• якщо операнд-співмножник має розмір 4 байти, в регістрі EAX;
Розміщення результату виконання залежить від його розміру:
• якщо множники – байти, результат розміщується в регістрі AX (слово);
• якщо множники – слова – в парі регістрів DX:AX (подвійне слово);
• якщо множники – подвійні слова – в парі регістрів EDX:EAX (слово, збільшене в 4 рази).
2. Двохоперандний формат: перший операнд визначає перший множник і результат водночас; другий операнд є другим множником.
3. Трьохоперандний формат: перший операнд – результат, другий і третій – множники.</td></tr>
<tr><td colspan="2">r16</td><td>r16
mem16</td></tr>
<tr><td colspan="2">r16</td><td>op8
op16</td></tr>
<tr><td>r16</td><td>r16
mem16</td><td>op8,
16</td></tr>
<tr><td colspan="2">r32</td><td>r32
mem32</td></tr>
<tr><td>r32</td><td>r16
mem16</td><td>op32</td></tr>
<tr><td>r32</td><td>r16
mem16</td><td>op8</td></tr>
<tr><td>r32</td><td>op32</td><td>op32</td></tr>
</table>

Продовження таблиці 11.1

idiv	r8, 16, 32 mem8, 16, 32		Цілочисельне ділення із знаком. Ділене вказується неявно (розміщується в акумуляторі): • якщо дільник має розмір 1 байт, ділене повинне розміщуватись в регістрі AX; після виконання операції частка поміщується в регістр AL, а залишок – в регістр AH; • якщо дільник має розмір 2 байти, ділене повинне розміщуватися в парі регістрів DX:AX; після виконання операції частка поміщується в регістр AX, а залишок – в регістр DX; • якщо дільник має розмір 4 байти, ділене повинне розміщуватися в парі регістрів EDX:EAX; після виконання операції частка поміщується в регістр EAX, а залишок – в регістр EDX.
mul	r8, 16, 32 mem8, 16, 32		Цілочисельне множення без урахування знаку. Вимоги до розміщення операндів співпадають з одноопераднім форматом операції IMUL.
div	r8, 16, 32 mem8, 16, 32		Цілочисельне ділення без урахування знаку. Вимоги до розміщення операндів співпадають з операцією IDIV.
inc	r8, 16, 32 mem8, 16, 32		Збільшення операнд на 1.
dec	r8, 16, 32 mem8, 16, 32		Зменшення операнд на 1.
neg	r8, 16, 32 mem8, 16, 32		Зміна знаку операнд.
and	r8, 16, 32	op8, 16, 32 r8, 16, 32 mem8, 16, 32	Побітове логічне "ТА"
	mem8, 16, 32	op8, 16, 32 r8, 16, 32	
or	r8, 16, 32	op8, 16, 32 r8, 16, 32 mem8, 16, 32	Побітове логічне "АБО"
	mem8, 16, 32	op8, 16, 32 r8, 16, 32	
xor	r8, 16, 32	op8, 16, 32 r8, 16, 32 mem8, 16, 32	Побітове логічне додавання за модулем 2 ("АБО", яке виключає нерівнозначність)
	mem8, 16, 32	op8, 16, 32 r8, 16, 32	
not	r8, 16, 32 mem8, 16, 32		Побітове логічне "НІ" (заперечення, інверсія)

Методичні вказівки

Завданням даної лабораторної роботи є розробка і реалізація мовою асемблера програми, призначеної для виконання цілочисельних обчислень за допомогою відповідних інструкцій процесора.

При виконанні роботи дозволяється не реалізовувати підпрограми для вводу вихідних чисел з клавіатури, а задавати їх значення безпосередньо в тексті програми.

Приклад виконання роботи

Завдання:

Реалізувати програму, яка виконує множення двох 16-бітових чисел, заданих в упакованому форматі BCD і виводить на екран результат у двійковій системі числення.

Розв'язання:

```
.model      tiny
.start
; 1) трансляція числа з BCD у двійковий формат
; CX – десятковий множник (1, 10, 100, 1000)
        mov   cx, 1
        xor   di, di              ; DI – накопичена сума
        mov   bx, word ptr [a]    ; bx – перше BCD-число
c1:
; виділення чергової цифри
        mov   ax, bx
        and   ax, 000Fh
        mul   cx                  ; отримання одиниць / десятків
        add   di, ax              ; накопичення суми
        cmp   cx, 1000
        je    stop
; збільшення десяткового множника в 10 разів за формулою
; CX * 10 = CX * 8 + CX * 2
        mov   ax, cx
        shl   cx, 3
        shl   ax, 1
        add   cx, ax
        shr   bx, 4               ; зсув наступної цифри
        jmp   c1
stop:
; 2) порозрядне множення
        mov   cx, 1
        mov   bx, word ptr [b]
c2:
; отримання чергової десяткової цифри другого множника, множення її на
; перший множник і на десятковий коефіцієнт
        mov   ax, bx
        and   ax, 000Fh
        mul   cx
        mul   di
        add   word ptr [c]        ;накопичення результату (молодше, старше слово)
```

```
        jnc     nocarry
        inc     word ptr [c + 2]
nocarry:
        add     word ptr [c + 2], dx
        cmp     cx, 1000
        je      finish
        mov     ax, cx
        shl     cx, 3
        shl     ax, 1
        add     cx, ax
        shr     bx, 4
        jmp     c2
finish:
; вивід старших 16 біт результату
        mov     cx, 16
        mov     bx, word ptr [c + 2]
        shl     bx, 1           ; зсув ліворуч старшого слова результату
        jc      isbit1          ; і перевірка прапору переносу
; вивід символу нуля на екран, якщо переносу не було
        mov     ah, 02h
        mov     dl, 30h
        int     21h
        jmp     step1
isbit1:
; вивід символу одиниці на екран, якщо перенос був
        mov     ah, 02h
        mov     dl, 31h
        int     21h
step1:
        loop    cycle1
; вивід молодших 16 біт результату
        mov     cx, 16
        mov     bx, word ptr [c]
cycle2:
        shl     bx, 1
        jc      isbit2
        mov     ah, 02h
        mov     dl, 30h
        int     21h
        jmp     step2
isbit2:
        mov     ah, 02h
        mov     dl, 31h
        int     21h
step2:
```

```
        loop   cycle2
        ret
a       dw     7259h
b       dw     3886h
c       dd     0
        end start
```

Варіанти завдань

Виконати команди з відповідними параметрами згідно таблиці.

№ з/п	add		sub		mul	div	inc	dec	neg	and		or		xor		not
	op1	op2	op1	op2						op1	op2	op1	op2	op1	op2	
1.	r8	r8			r8		r8			r8	r8					
2.			r8	r8		r8		r8				r8	r8			
3.	r16	r16			r16		r16			r16	r16					
4.			r16	r16		r16		r16				r16	r16			
5.	r8	r8				r8			r8					r8	r8	
6.			r8	r8			r8		r8							r8
7.	r8	op8			r8		mem8			r8	op8					
8.			r8	op8		r8		r8				r8	op8			
9.	r16	op16			r16		op16			r8	op8					
10.			r16	op16		r16		r16				r16	op16			
11.	r16	r16				r8			r8					r8	r8	
12.			r16	r16			r16		r16							r16
13.	r16	op16				r8			r8					r16	op8	
14.			r16	op16			r16		r16							r16
15.	r8	op8			r8		mem8					r8	op8			
16.			r8	op8		r8		r8		r8	op8					
17.	r16	r16			r16		r16							r16	r16	
18.			r16	r16		r16		r16						r16	r16	
19.	r8	r16			r8		r8			r8	r16					
20.			r8	r16		r8		r8				r8	r16			
21.	r8	op16				r8			r8					r8	op8	
22.			r16	r16			r16					r16	r16			r16
23.	r16	op16				r16		op16		r8	op8					
24.			r16	op16	r16		r16					r16	op16			
25.	r8	r8			r8				r8							r8

Лабораторна робота № 12

ЗАСТОСУВАННЯ ЛАНЦЮГОВИХ ОПЕРАЦІЙ ДЛЯ ОБРОБКИ МАСИВІВ ДАНИХ

Мета роботи: засвоїти ланцюгові операції, які підтримуються процесором, вивчити формати ланцюгових операцій, навчитись застосовувати ланцюгові операції для обробки масивів даних.

Теоретичні відомості

Ланцюгові операції дозволяють виконувати дії над блоками даних у пам'яті, що являють собою послідовності елементів з розміром 8, 16 або 32 біт. Особливістю ланцюгових операцій є те, що кожна з них, окрім обробки поточного елемента ланцюга, здійснює ще й автоматичне просування до наступного елемента даного ланцюга.

До складу системи операцій процесора входять сім операцій-примітивів обробки ланцюгів. Кожна з них реалізована в процесорі трьома командами; кожна з цих команд, в свою чергу, працює з елементом відповідного розміру – байтом, словом або подвійним словом.

В таблиці 12.1 перелічені операції-примітиви та команди, за допомогою яких вони реалізуються.

Таблиця 12.1. Ланцюгові операції

Позначення і формат інструкції			Призначення
movs movsb movsw movsd	dmem	smem	Пересилання ланцюга (b – байтів, w – слів, d – подвійних слів); dmem – адреса приймача даних (повинна розміщуватись у парі регістрів ES:EDI (DI); smem – адреса джерела даних (повинна розміщуватись у парі регістрів DS:ESI (SI)
cmps cmpsb cmpsw cmpsd	dmem	smem	Порівняння ланцюгів. Операція послідовно порівнює (віднімає) відповідні елементи двох ланцюгів та встановлює прапори ZF, SF і OF у відповідності з результатами. Вимоги до розміщення ланцюгів співпадають попередньою операцією.
scass scasb scasw scasd	dmem		Сканування ланцюгів. Операція послідовно порівнює (віднімає) вміст акумулятора (AL, AX або EAX) та черговий елемент ланцюга і встановлює прапори ZF, SF і OF у відповідності з результатами. Адреса елемента (dmem) зберігається в парі регістрів ES:EDI (DI).
lods lodsb lodsw lodsd	smem		Завантаження елемента з ланцюга. Операція завантажує черговий елемент ланцюга, розташований за адресою (smem), що зберігається в парі регістрів DS:ESI (SI), в акумулятор (AL, AX або EAX).

Продовження таблиці 12.1

stos stosb stosw stosd	dmem		Запис елемента в ланцюг. Операція записує вміст акумулятора (AL, AX або EAX) до чергового елемента ланцюга, розташованого за адресою (dmem), що зберігається в парі регістрів ES:EDI (DI).
ins insb insw insd	dmem	port	Завантаження елемента з порту вводу/виводу. Операція записує до чергового елемента ланцюга, адреса якого (dmem) зберігається в парі регістрів ES:EDI (DI), вміст порту вводу/виводу (port), номер якого записаний в регістрі DX.
outs outsb outsw outsd	port	smem	Запис елемента в порту вводу/виводу. Операція записує в порт вводу/виводу (port) , номер якого записаний в регістрі DX, черговий елемент ланцюга, адреса якого (smem) зберігається в парі регістрів ES:ESI (SI).

Будь-яка операція з перелічених в таблиці 1 записується в програмі без явних операндів.

```
; завантаження адрес джерела і приймача до пар регістрів DS:SI і ES:DI відповідно
; ланцюг-джерело і ланцюг-приймач знаходиться у сегменті даних
mov   ax, @data
mov   ds, ax
mov   es, ax
; завантаження ефективних адрес
lea     si, source
lea     di, destination
; переписати 1 байт у напрямку DS:SI → ES:DI
movsb
```

При цьому операція виконуватиметься один раз.

Існує, однак, можливість організувати автоматичне багаторазове виконання ланцюгової операції за допомогою так званих префіксів повторення, можливі позначення яких приведені в таблиці 2.

Таблиця 12.2. Префікси повторення для ланцюгових операцій

Позначення префікса	Призначення
rep	Виконання ланцюгової операції до тих пір, поки вміст регістра ECX (CX) не стане дорівнювати нулю.
repe repz	Виконання ланцюгової операції до тих пір, поки вміст регістра ECX (CX) не стане дорівнювати нулю і прапор ZF дорівнює одиниці. Найчастіше використовуються разом з операціями CMPS і SCAS для пошуку елементів ланцюгів, які відрізняються.
repne repnz	Виконання ланцюгової операції до тих пір, поки вміст регістра ECX (CX) не стане дорівнювати нулю і прапор ZF дорівнює нулю. Найчастіше використовуються разом з операціями CMPS і SCAS для пошуку елементів ланцюгів, які співпадають.

Отже кількість повторень операції, які треба здійснити попередньо записується в регістр ECX (CX). При записі з префіксом повторення кожна ланцюгова операція автоматично зменшує вміст регістра ECX(CX) на одиницю.

Ланцюгові команди автоматично модифікують вміст регістрів, які містіть адресу джерела і/або приймача даних, однак тип цієї модифікації можна задавати за допомогою прапора DF. Якщо вміст цього прапора дорівнює нулю, то значення індексних регістрів ESI (SI) та EDI (DI) будуть автоматично збільшуватись ланцюговими операціями на розмір операнда. Якщо вміст прапора DF – одиниця, значення індексних регістрів будуть автоматично зменшуватись. Таким чином здійснюється регулювання напрямку обробки даних (у бік збільшення або в бік зменшення адрес). Керування вмістом прапора DF здійснюється за допомогою наступних операцій:

- CLD (Clear Direction Flag) – очистити прапор напрямку – скидання прапору напрямку в нуль;
- STD (Set Direction Flag) – встановити прапор напрямку – встановлення прапору напрямку в одиницю.

Приклад:

```
str     db      'Test string for testing of SCAS instruction'
strlen db       43      ; довжина рядка
sym     db      'f'     ; символ для пошуку
…
mov     ax, @data
mov     es, ax
lea     di, str
mov     cx, byte ptr [strlen]
mov     al, byte ptr [sym]
cld             ; очищення прапора напрямку – адреси збільшуватимуться
repne scas      ; пошук в рядку, вихід при першому спів падінні
je      found ; якщо символ знайдено
```

Методичні вказівки

Завданням даної лабораторної роботи є розробка і реалізація мовою асемблера програми з використанням ланцюгових операцій.

Приклад виконання роботи

Завдання:

задані два масиви слів, розміром 10кб кожний. Реалізувати програму, яка обчислює кількість співпадаючих елементів у них і виводить на екран повідомлення "Більше половини", "Менше половини" і "Половина" в залежності від відношення цієї кількості до загальної довжини масивів.

Розв'язання:

```
.model      small
.stack      100h
.data
    mas1  dw    5120 dup (?)
```

```
        mas2  dw     5120 dup (?)
        count dw     0
        above_msg  db      'More than half', '$'
        below_msg  db      'Less than half', '$'
        equate_msg db      'Half', '$'
.code
        mov   ax, @data
        mov   ds, ax
        push  ds              ; підготовка сегментних ефективних адрес масивів
        push  es
        lea   si, mas1        ; підготовка ефективних адрес масивів
        lea   di, mas2
        mov   cx, 5120
        cld                   ; напрямок за збільшенням адреси
continue:
        repne cmpsw           ; цикл по елементного порівняння масивів до першого співпадіння
        jcxz  stop_cycle
        inc   word ptr [count]
        jnp   continue
stop_cycle:
        mov   ax, word ptr [count]
        cmp   ax, 2560
        jbe   below_or_equate
        mov   ah, 09h
        lea   dx, above_msg
        int   21h
        jmp   finish
below_or_equate:
        mov   ax, word ptr [count]
        cmp   ax, 2560
        jb    below
        mov   ah, 09h
        lea   dx, eqate_msg
        int   21h
        jmp   finish
below:
        mov   ah, 09h
        lea   dx, below_msg
        int   21h
finish:
        mov   ah, 4Ch
        int   21h
end
```

Варіанти завдань

1. Реалізувати програму, яка зчитує з клавіатури рядок і окремий символ, а потім обчислює кількість входжень цього символу в рядок.
2. Реалізувати програму, яка обчислює і виводить на дисплей кількість непустих знакомісць на екрані в текстовому режимі.
3. Реалізувати програму, яка обчислює кількість не співпадаючих елементів у двох рядках, які вводяться користувачем з клавіатури.
4. Реалізувати програму, яка динамічно виділяє сегмент оперативної п’ясті та заповнює його символ, який вводить користувач з клавіатури.
5. Реалізувати програму, яка збільшує на одиницю кожен елемент одновимірного масиву та створює копію результату в пам'яті.
6. Реалізувати програму, яка в одновимірному масиві кожен парний елемент перетворює на непарний шляхом віднімання одиниці від нього.
7. Реалізувати програму, яка в одновимірному масиві кожен нульовий елемент замінює на символ, заданий користувачем.
8. Реалізувати програму, яка в заданому рядку підраховує кількість символів – цифр.
9. Реалізувати програму, яка переписує з одного масиву в інший виключно непарні числа.
10. Реалізувати програму, яка виводить на екран номери позицій проміжків в рядку символів.
11. Реалізувати програму, яка переписує з одного рядка в інший лише ті символи, що не співпадають із заданими.
12. Реалізувати програму, яка підраховує кількість розміщень символу, заданого користувачем на екрані в текстовому режимі.
13. Реалізувати програму, яка в заданому одновимірному масиві кожен від'ємний елемент замінює на нуль, а кожен елемент, що перевищує 255 – на 255.
14. Реалізувати програму, яка зчитує з клавіатури рядок і два символи, а потім всі символи в рядку, коди яких лежать між заданими, замінює на проміжки.
15. Реалізувати програму, яка перетворює масив цифрових символів на масив одно розрядних чисел (30h – 0, 31h – 1, …)

Лабораторна робота № 13

МЕХАНІЗМИ ПЕРЕРИВАНЬ В МОВІ АСЕМБЛЕРА

Мета роботи: вивчити основні принципи механізму переривань в мові асемблера.

Теоретичні відомості

Переривання – це події, які заставляють центральний процесор (ЦП) переривати виконання поточної роботи та перейти на виконання програми, яка називається обробник переривань. Цей перехід відбувається за малий час за допомогою спеціально розроблених апаратних засобів.

Обробник переривання визначає причину переривання, виконує заплановані дії, після чого повертає керування перерваній програмі.

Звичайно переривання викликаються подіями внутрішніми по відношенню до ЦП, які потребують негайних дій. Наприклад:

- завершення операції вводу/виводу;
- виявлення апаратного збою;
- відмови живлення.

Більшість сучасних процесорів підтримають механізм типів і рівнів переривань. Кожному типу звичайно відповідає комірка в пам'яті, яка називається вектором переривання, яка визначає місцезнаходження програми обробника переривання даного типу. Концепція типів переривання дозволяє назначити переривання пріоритети. Переривання групуються за рівнями, які мають однаковий пріоритет.

ЦП комп’ютерів, які підтримують переривання, повинні мати засоби блокування переривань на час виконання критичних ділянок програми (вибіркового або глобального). При обробці переривання ЦП блокує всі переривання цього або більш низьких рівнів та дозволяє – більш високих.

Процесори сімейства іх86 підтримують 256 типів переривань, які викликаються подіями трьох груп:

- внутрішні апаратні переривання;
- зовнішні апаратні переривання;
- програмне переривання.

Кожному типу переривання відповідає свій номер.

Внутрішні переривання або відмови генеруються окремими подіями, які виникають в процесі виконання програми (ділення на 0, неправильний код операції). Закріплення номерів переривань за визначеними причинами "зашито" в процесорі.

Зовнішні переривання ініціалізуються контролерами периферійного устаткування або співпроцесорами.

Програмне переривання. Будь-яка програма може ініціалізувати синхронне програмне переривання шляхом виконання команди INT з вказівкою номеру переривання.

Розподілення номерів програмних переривань умовно ще не закріплено апаратно.

Нижні 1024 байтів системної пам'яті носять назву таблиці векторів переривань. Вектор займає 4 байта: сегмент та зсув відповідного обробника переривання.

У випадку отримання сигналу або команди переривання ЦП розміщує в стек вміст прапорцевого регістру, очищує прапорці TF та IP, заносить в стек вміст регістрів CS та IP й блокує систему переривань. Далі, за допомогою 8-бітового числа – номера переривання, яке встановлене на внутрішній шині, витягує із таблиці векторів адреси CS та IP та відновлює роботу з цієї адреси.

Звичайно обробник розблоковує систему, зберігає регістри, які будуть їх використовуватися й обробляє переривання. В кінці обробника повинна знаходитись команда IRET – повернення із переривання. Ця команда відновлює первинне значення прапорцевого регістру, CS та IP.

В якості прикладу розглянемо, як використовувати командні переривання для виводу інформації на екран дисплея.

Всі необхідні екранні операції можна здійснити за допомогою команди Int 10h, яка передає керування безпосередньо в BIOS. Перед викликом переривання Int 10h необхідно задати в регістрі AH номер функції виводу. В інших регістрах необхідно задати параметри, які вимагаються цією функцією. Номера підфункцій, їх параметри, а також регістри, в які їх необхідно заносити наведено в довідковій інформації по BIOS.

Екран можна представити у вигляді двомірного простору з позиціями, які адресуються, в будь-яку з якої може бути встановлено курсор або виведено символ. Монітор, наприклад, може мати 25 рядків (нумеруються від 0 до 24) та 80 стовпчиків (нумеруються від 0 до 79). Нумерація рядків відбувається з лівого верхнього кута екрану донизу, нумерація стовпчиків – також с лівого верхнього кута екрану вправо.

Очищення екрану.

Запити та команди залишаються на екрані дисплею доти, доки не будуть зсунуті в результаті прокручування або не переписані на тому ж місці іншими запитами або командами. Коли програма починає своє виконання, екран може бути очищено. Область екрану, яка очищується, може починатися з будь-якої позиції екрану та закінчуватися в будь-якій іншій позиції з більшим номером.

Початкове значення рядка та стовпчика заноситься в регістр CX, кінцеве – в DX, значення 07 – в регістр BH, 0600h – AX.

Наступний приклад виконує очищення всього екрану.

```
mov   ax,   0600h      ; AH=06 (прокрутка) AL=00 (весь екран)
mov   bh,   07         ; нормальний атрибут (чорно/білий)
mov   cx,   0000       ; верхня ліва позиція (CH=00 – номер рядка,
                       ; CL=00 – номер стовпчика
mov   dx,   184Fh      ; нижня права позиція (DH=18 – номер рядка,
                       ; DL=4F – номер стовпчика
int   10h              ; передача керування в BIOS
```

Розміщення курсору. Команда Int 10h включає в собі розміщення курсору в будь-яку позицію екрану. Нижче наведено приклад встановлення на 5-й рядок й 12-й стовпчик.

```
mov  ah,   02     ; підфункія установки курсору
mov  bh,   00     ; екранна сторінка 0
mov  dh,   05     ; рядок 05
mov  dl,   12     ; стовпчик 12
int 10h           ; передача керування в BIOS
```

Вивід на екран в базовій версій DOS.

Вивід на екран в базовій версії DOS потребує визначення текстового повідомлення в сегменті даних, установки в регістрі AH значення 09 (виклик функції DOS) й вказівки команди Int 21h. В процесі виконання операції кінець повідомлення визначається по обмежувачу (знак долару ‘$’), як наведено нижче

```
nampr     db    ‘ДАНЕ ПОВІДОМЛЕННЯ’,’$’
          …
          mov   ah, 09        ; підфункція відображення рядку на екран
          lea   dx, nampr     ; завантаження адреси повідомлення в регістр DX
          int 21h
```

Знак обмеження ‘$’ можна записувати безпосередньо після символьного рядку, як це наведено у прикладі, в середині рядка ‘ДАНЕ ПОВІДОМЛЕННЯ$’ або в наступному операторі DB ’$’. Використовуючи цю операцію неможливо вивести на екран символ долара. Якщо символ долара буде відсутній наприкінці повідомлення, яке виводиться, то на екран будуть виводитися всі наступні символи, доки код знаку ’$’ не зустрінеться в пам'яті.

Команда LEA завантажує адресу області пам'яті NAMPR в регістр DX для передачі в DOS адреси інформації, яки виводиться.

Контрольні питання

1. Що таке переривання?
2. Що таке вектор переривання?
3. На які групи розділяються типи переривань процесорів x86?
4. Що представляє собою таблиця векторів переривань?
5. Яка послідовність обробки переривань?
6. Яка команда повинна знаходитись в кінці обробника переривань?
7. Який номер програмного переривання для виводу на екран засобами BIOS?

Варіанти завдань

1. Виконати виведення на екран ПІБ використовуючи int 10h у позицію – номер стовпчика відповідає порядковому номеру відповідно до номера студента за журналом; номер рядка відповідає останнім двом цифрам залікової книжки.

Лабораторна робота № 14

ПРОЦЕДУРИ ТА МАКРОКОМАНДИ

Мета роботи: ознайомитись з поняттями процедури та макрокоманди.

Теоретичні відомості

При створенні програми може з’явитись необхідність в різним місцях програми виконати одні й ті самі дії, можливо з різними даними. Ділянки програми, які повторюються, можна оформити у вигляді ПРОЦЕДУРИ. В цьому випадку послідовність кодів, яка повторюється, буде описана в програмі один раз, звичайно в кінці основної програми. В тексті основної програми в тих місцях, де повинен знаходитись код процедури буде знаходитись оператор виклику процедури CALL. При описі процедури оператором команди описі буде це ім’я.

Для опису процедури використовується директива PROC. Формат цієї директиви наведено в лабораторній роботі №1. Нагадаємо, що в цій лабораторній роботі програма містила одну процедуру, яка була оформлена у вигляді:

```
begin       proc  far
;команди, які виконувалися
            ret
begin       endp
```

Оператор FAR інформує систему, що дана адреса є точкою входу для виконання. Директива ENDP визначає кінець процедури.

Кодовий сегмент може містити будь-яку кількість процедур. Типову організація багато процедурної програми наведено нижче.

```
codesg      segment     para
begin       proc        far
            …
            call        b10
            call        c10
            ret
begin       endp
b10         proc        near
            …
            ret
b10         endp
c10         proc        near
            …
            ret
c10         endp
codesg      ends
end         begin
```

Директиви PROC, які використовуються для опису процедур з іменами B10 та C10 мають тип NEAR, що вказує на те, що дані процедури знаходяться в поточному кодовому сегменті. В подальшому тип NEAR може бути опущено, тому що цей тип приймається за умовчанням.

Кожній процедурі повинно бути присвоєно унікальне ім’я, яке знаходиться в полі імені процедури (begin, b10, c10). В результаті виконання команди CALL B10 керування буде передано процедурі b10, виконуються команди, які містяться в описі процедури та по команді RET керування буде передано в основну програму BEGIN на команду, наступну за CALL B10. Команда CALL B10 діє аналогічно.

Команда CALL при своєму виконанні автоматично заносить в стек адресу команди, наступної за CALL, при цьому зменшується вказівник стеку (значення регістру SP). Команда RET витягує цю адресу із стеку та використовує його для повернення в програму, яка його викликала. При цьому автоматично збільшується вказівник стеку. Для правильного повернення із процедури до моменту виконання команди RET вказівник стеку повинен знаходитись в тому ж стані, що й відразу після виконання команди CALL. Запам’ятайте, що команди PUSH, PUSHF, CALL, INT, INTO заносять інформацію в стек й зменшують вказівник стеку. Команди POP, POPF, RET, IRET витягує інформацію із стеку й збільшує вказівник стеку. Наприклад, якщо в процедурі використовували декілька команд PUSH, бажано виконати таку ж саму кількість команд POP для відновлення вказівника стеку або відновити вказівник стеку іншим способом.

Приклад 1. Наведено текст програми, яка показує вплив програми, яка виконується, на стек.

```
 1    0000                stacksg    segment    para  stack 'Stack'
 2    0000  20*(????)                dw    32 dup (?)
 3    0040                stacksg    ends
 4
 5    0000                codesg     segment    para  'Code'
 6    0000                begin proc  far
 7                                assume      cs:codesg, ss:stacksg
 8    0000  1E                    push  ds
 9    0001  2B CO                 sub   ax, ax
10    0003  50                    push  ax
11    0004  E8 0001               call  B10   ;викликати B10
12                                ;------------------
13    0007  CB                    ret                ;завершення
14    0008                begin endp
15                                ;------------------
16    0008                B10         proc
17    0008  E8 0001                   call  C10   ;викликати C10
18                                ;------------------
19    000B C3                     ret                ;повернутися в програму
```

```
20    000C              B10   endp
21                            ;-----------------
22    000C              C10   proc
23                            ;-----------------
24    000C C3                 ret                 ;повернутися в програму
25    000D              C10   endp
26                            ;-----------------
27    000D              codesg      ends
28                            end   begin
```

Якщо ви виконуєте трасування даної програми, то побачите наступне. Поточною доступною коміркою стеку для занесення або витягання слова є вершина стеку. Перша команда PUSH зменшує значення SP на 2 та заносить вміст регістру DS (в даному прикладі 049F) в вершину стеку, тобто за адресою 4B00+3E. Друга команда PUSH також зменшує SP на 2 та записує вміст регістру AX (0000) за адресою 4B00+3C. Команда CALL B10 зменшує значення SP та записує відносну адресу наступної команди (0007) в стек за адресою 4B00+3A. Команда CALL B10 зменшує значення SP та записує відносну адресу наступної команди (000B) в стек за адресою 4B00+38. При поверненні із процедури команда C10 команда RET витягує із стеку (4B00+38), розміщує його в вказівник команд IP та збільшує SP на 2. При цьому відбувається автоматичне повернення за відносною адресою 000B в кодовому сегменті, тобто повернення в процедуру B10. Команда RET в кінці процедури B10 витягує адресу 0007 із стеку (4B00+3A), розміщує його в IP та збільшує значення на 2. При цьому відбувається автоматичне повернення за відносною адресою 0007 в кодовому сегменті. Команда RET за адресою 0007 завершує виконання програми та здійснює повернення типу FAR.

Нижче наведено вплив на стек при виконанні кожної команди. Наведено тільки вміст пам'яті за адресами 0034-003F та вміст регістру SP.

Команда	Стек	SP
Початкове значення:	xxxx xxxx xxxx xxxx xxxx xxxx	0040
push ds (запис 049F)	xxxx xxxx xxxx xxxx xxxx 049F	003E
push ax (запис 0000)	xxxx xxxx xxxx xxxx 0000 049F	003C
call B10 (запис 0007)	xxxx xxxx xxxx 0007 0000 049F	003A
call C10 (запис 000B)	xxxx xxxx 0B00 0007 0000 049F	0038
ret (витяг 000B)	xxxx xxxx xxxx 0007 0000 049F	003A
ret (витяг 0007)	xxxx xxxx xxxx xxxx 0000 049F	003C
	⇑ ⇑ ⇑ ⇑ ⇑ ⇑	
Зсув в стеку	0034 0036 0038 003A 003C 003E	

Зверніть увагу. По перше, слова в пам'яті вістять байти в зворотній послідовності. 0007 записується у вигляді 0700. По друге, відладчик DEBUG при використанні його для перегляду стеку заносить в стек інші значення, в тому числі вміст IP для своїх потреб.

Макрокоманди подібні процедурам та представляють собою "міні-програми", які можна вставляти в результатні програми, вказуючи їх імена.

Макрокоманди представляють собою послідовність операторів мовою асемблера (команд та директив мовою асемблера), які можуть декілька разів повторюватися в програмі. Подібно процедурам макрокоманди мають імена. Після того, як макрокоманда задана, її ім’я можна використовувати в результуючій програмі замість послідовності програм.

Між макрокомандами та процедурами існують розбіжності. Коди команд процедури входять в об’єктний код одноразово та макропроцесор а процесі виконання програми передає їм керування (тобто викликає їх командою CALL) за необхідністю. Коди команд макрокоманди будуть включатись в об’єктний код стільки разів, скільки разів буде вказуватись його ім’я. Транслятор замінює кожне ім’я макрокоманди на ті команди, які вона представляє. Говорять, що транслятор "розширює" макрокоманду. Отже, при виконанні програми мікропроцесор виконує команди макрокоманди безпосередньо, тобто не передає керування в інше місце пам'яті як в процедурі. Таким чином, імя макрокоманди представляє собою директиву транслятора; воно служить командою транслятору, а не мікропроцесору.

В порівнянні з процедурами макрокоманди мають три переваги:

1. Макрокоманди динамічні. За рахунок зміни вхідних параметрів макрокоманди можна змінювати не тільки об’єкти, якими маніпулює, але й дії, над якими виконує дії. У випадку процедури, можна змінювати тільки дані, які передаються, що робить процедури менш гнучкими.
2. Використання макрокоманд замість процедур прискорює використання програми, оскільки мікропроцесору не треба виконувати команди виклику процедури й повернення із неї.
3. Макрокоманди можна ввести в бібліотеку, із якої програміст може витягувати їх при створенні інших програм.

Макрокоманди мають основний недолік, якого позбавлені процедури: при їх використанні об’єктні коди програми становляться довше (в порівнянні з процедурами), отже макрокоманди розширюються при кожній їх появі та пам'ять заповнюється командами, які повторюються.

Склад макрокоманд (макросів)

Кожній макрос складається з трьох частин:

1. Заголовок: директива MACRO, в полі імені якої вказано І’МЯ МАКРОСУ, а в полі операнда – СПИСОК ПАРАМЕТРІВ;
2. Тіло – послідовність операторів асемблера (команд та директив), які задають дії, які виконуються макрокомандою.
3. Кінцевик – директива ENDM, яка відзначає кінець макроса.

Загальний вигляд запису макрокоманди:

```
ім’я MACRO [список_параметрів]
;тіло макрокоманди
ENDM
```

Макрокоманда повинна знаходитись до визначення сегменту. Розглянемо приклад простого макросу на ім’я INIT1, який ініціалізує сегментні регістри для EXE-програми.

```
init1     macro                                  ;заголовок
assume    cs:csseg, ds:dseg, ss:stack, es:dseg   ;тіло
          push  ds
          sub   ax, ax
          push  ax
          mov   ax, dseg
          mov   ds, ax
          mov   es, ax
          endm
```

Імена на які маються посилання в макрокоманді (CSEG, DSEG, STACK) повинні буди визначені будь-де в іншому місті програми. Макрокоманду INIT1 можна вказувати в тому місті програми, де необхідно ініціалізувати регістри. Відповідний асемблерний рядок буде виглядіти – INIT1.

Нижче наведено листінг програми, яка використовує макрос INIT1. В листінгу макророзширення кожна команда, яка помічена “1”, є результатом генерації макрокоманди. В макророзширенні відсутня директива ASSUME, оскільки вона не генерує об’єктний код.

```
  1 init1                    macro                                ;заголовок
  2 assume                   cs:cseg,ds:dseg,ss:stseg,es:dseg     ;тіло
  3                          push  ds
  4                          sub   ax, ax
  5                          push  ax
  6                          mov   ax, dseg
  7                          mov   ds, ax
  8                          mov   es, ax
  9                          endm
 10 0000                     stseg  segment para stack 'Stack'
 11 0000  20*(????)          dw 32 DUP (?)
 12 0040                     stseg  ends
 13 0000                     dseg   segment para 'DATA'
 14 0000  54 65 73 74 20 6F 66+ messeg db 'Test of Macro IN'
 15       20 4D 61 63 72 6F 20+
 16       49 4E
 17 0010                     dseg   ends
 18 0000                     cseg   segment para 'CODE'
 19 0000                     begin proc far
 20                          init1
1 21 0000  1E                push ds
1 22 0001  2B  C0            sub   ax, ax
1 23 0003  50                push  ax
1 24 0004  B8  0000s         mov   ax, dseg
```

```
1 25 0007   8E    D8          mov   ds, ax
1 26 0009   8E    C0          mov   es, ax
  27 000B   1E                push ds
  28 000C   2B    C0          sub ax, ax
  29 000E   50                push ax
  30 000F   B8    0000s       mov ax, dseg
  31 0012   8E    D8          mov ds, ax
  32 0014   8E    C0          mov es, ax
  33 0016   B8    0040        mov   ax, 40h
  34 0019   BB    0001        mov   bx, 01
  35 001C   B9    001A        mov   cx, 26
  36 001F   BA    0000r       lea     dx, messeg
  37 0022   CD    21          int 21h
  38 0024   CB                ret
  39 0025                     begin endp
  40 0025                     cseg   ends
  41                          end begin
```

Варіанти завдань

1. Вставити в текст програми до лабораторної роботи № 8 спочатку та в кінці програми вивід на екран повідомлення, яке містить номер вашої групи, П.І.Б. Виконайте це за допомогою процедури.
2. Виконайте п.1 за допомогою макрокоманди.
3. Виконайте трансляцію обох програм, створіть листінги, порівняйте їх. Скомпонуйте й виконайте обидві програми.

Контрольні питання

1. Як транслятор оброблює макрокоманду?
2. Що виконується скоріше, процедура чи макрокоманда?

Лабораторна робота № 15

ПРОГРАМУВАННЯ МАТЕМАТИЧНИХ РОЗРАХУНКІВ З ВИКОРИСТАННЯМ ІНСТРУКЦІЙ МАТЕМАТИЧНОГО СПІВПРОЦЕСОРА

Мета роботи: засвоїти основні операції, які підтримуються математичним співпроцесором (FPU) і80387; вивчити операції FPU; навчитись застосовувати операції FPU для обробки даних з плаваючою точкою.

Теоретичні відомості

В ПК на базі процесорів Intel усі операції з плаваючою точкою виконує спеціальний пристрій – FPU (Float Point Unit), представлений спочатку у вигляді співпроцесора (8087, 80287, 80387, 80487), а з моделі 80486DX – вбудований в основний процесор.

Математичний співпроцесор може виконувати операції над типами даних, представленими в таблиці 15.1, з яких 4 типи є цілими, а 3 – з плаваючою точкою.

Таблиця 15.1. Типи даних FPU

Назва типу	Розрядність, біт	Кількість значущих цифр	Границі
Ціле слово	16	4	– 32768…32767
Коротке ціле	32	9	$-2^{31}+1 \ldots 2^{31}-1$
Довге ціле	64	18	$-2^{63}+1 \ldots 2^{63}-1$
Упаковане десяткове	80	18	– 99…9… + 99…9 (18 цифр)
Коротке речове	32	7	$1{,}18 \times 10^{-38} \ldots 3{,}40 \times 10^{38}$
Довге речове	64	15 – 16	$2{,}23 \times 10^{-308} \ldots 1{,}79 \times 10^{308}$
Розширене речове	80	19	$3{,}37 \times 10^{-4932} \ldots 1{,}18 \times 10^{4932}$

Речові формати, що використовуються FPU мають наступну структуру, представлену в таблиці 15.2.

Таблиця 15.2. Речові формати FPU

Назва формату	Знак мантиси (біта)	Експонент (біти)	Мантиса (біти)
Коротке речове	31	23 – 30 (8)	0 – 22 (23)
Довге речове	63	52 – 62 (11)	0 – 51 (52)
Розширене речове	79	64 – 78 (15)	0 – 63 (64)

Окрім звичайних чисел передбачені декілька спеціальних випадків, які можуть виникати в результаті математичних операцій:

- додатній нуль: всі біти числа – нульові;
- від'ємний нуль: знаковий біт – одиниця, всі інші біти – нульові;
- додатна нескінченність: знаковий біт – нуль, всі біти мантиси – нулі, всі біти експоненти – одиниці;
- від'ємна нескінченність: знаковий біт – одиниця, всі біти мантиси – нулі, всі біти експоненти – одиниці;
- ненормалізовані числа: всі біти експоненти – нулі;
- невизначеність: знаковий біт – одиниця, перший біт мантиси (перші два – для 80-бітових чисел) – одиниця, всі інші – нулі, всі біти експоненти – одиниці;
- не-число типу SNAN (сигнальне): всі біти експоненти – одиниці, перший біт мантиси (перші два – для 80-бітових чисел – один і нуль) – нуль, серед інших бітів є одиниці;
- не-число типу QNAN (тихе): всі біти експоненти – одиниці, перший біт мантиси (перші два – для 80-бітових чисел) – одиниця, серед інших бітів є одиниці. Невизначеність є одним з варіантів QNAN.

До складу FPU входять 8 регістрів даних і п’ять допоміжних регістрів.

Регістри даних недоступні за назвами, а організують стек, вершина якого називається ST (0), а більш глибокі елементи – ST (1), ST (2), …, ST (7). Вершина стеку може змінювати своє розміщення.

Регістр стану SR – містить слово стану FPU. Його структура приведена в таблиці 15.3.

Таблиця 15.3. Структура регістра стану SR

Біт	Назва	Призначення
0	IE	Прапор недопустимої операції: відбулася помилка стеку (SF) або виконана недопустима операція.
1	DE	Прапор ненормалізованого результату: виконана операція над ненормалізованим числом.
2	ZE	Прапор ділення на нуль; виконано ділення на нуль.
3	OE	Прапор переповнення: результат надто великий.
4	UE	Прапор антипереповнення: результат надто малий.
5	PE	Прапор неточного результату: результат не може бути представлений точно
6	SF	Помилка стеку: якщо C1 = 1, відбулось переповнення (спроба запису в непусту комірку стеку), якщо C1 = 0 – відбулось антипереповнення (спроба зчитування з пустої позиції стеку).
7	ES	Загальний прапор помилки: будь-яке немасковане виключення.
8	C0	Умовний прапор 0.
9	C1	Умовний прапор 1.

Продовження таблиці 15.3

10	C2	Умовний прапор 2
11	TOP	Число від 0 до 7: показує який з регістрів даних (ST (0) – ST (7)) в даний момент є вершиною стеку.
12		
13		
14	C3	Умовний прапор 3.
15	B	Зайнятість FPU: прапор призначений для сумісності з i8087, його значення співпадає з ES.

У мовні прапори C0 – C3 використовуються як і біти стану основного процесора: їхні значення відбивають результат виконання попередньої команди і використовуються для здійснення умовних переходів:

C0 – наявність переносу;

C1 – втрачається;

C2 – прапор паритету;

C3 – прапор нуля.

Регістр керування CR – призначений для встановлення режимів роботи FPU. Його структура представлена в таблиці 15.4.

Таблиця 15.4. Структура регістра стану SR

Біт	Назва	Призначення
0	IM	Маска недійсної операції.
1	DM	Маска ненормалізованого результату.
2	ZM	Маска ділення на нуль.
3	OM	Маска переповнення.
4	UM	Маска антипереповнення.
5	PM	Маска неточного результату.
6, 7	–	Зарезервовані.
8, 9	PC	Управління точністю.
10, 11	RC	Управління округленням.
12	IC	Управління нескінченністю (для сумісності з i80287).
13 - 15	–	Зарезервовані.

Біти 0 – 5 регістра керування масують відповідні виключення: якщо біт встановлений, виключення не відбувається.

Біти управління округленням RC визначають спосіб округлення результатів операцій FPU:

- 0 – до найближчого числа;
- 1 – до від'ємної нескінченності;

- 2 – до доданої нескінченності;
- 3 – нуля.

Біти управління точністю PC визначають точність результатів арифметичних операцій FPU:

- 0 – одинарна точність (32-бітові числа);
- 1 – зарезервовано;
- 2 – подвійна точність (64-бітові числа);
- 3 – розширена точність (80-бітові числа).

Регістрів тегів TW містить вісім пар бітів, що характеризують вміст кожного регістра даних: біти 15, 14 – регістра ST (7), біти 13, 12 – регістра ST (6), і т.д. Якщо пара бітів (тег) дорівнює 11, відповідний регістр даних – пустий; якщо 00 – регістр містить число; 01 – нуль; 10 – не число, нескінченність, ненормалізоване число і т.п.

При виконанні операцій FPU можуть виникати шість типів особливих ситуацій, які називаються виключенням. При виникненні виключення відповідний прапор в регістрі SR встановлюється в 1, і якщо маска цього виключення в регістрі CR не встановлена, викликається звичайне переривання IRQ 10 (якщо біт NE в регістрі CPU – CR0 встановлений) або IRQ 13, обробник якого може прочитати регістр SR, щоб визначити тип виключення та команду, яка його викликала, а потім спробувати виправити ситуацію.

За умовчуванням виконуються наступні дії:

- неточний результат: результат округляється у відповідності з бітами RC;
- антипереповнення: результат перетворюється в нескінченність відповідного знаку;
- ділення на нуль: результат перетворюється в нескінченність відповідного знаку (з урахуванням знаку нуля);
- ненормалізований операнд: обчислення триває звичайним шляхом;
- недійсна операція: результат залежить від операції.

В таблиці 15.5 приведені деякі операції математичного співпроцесора i80387.

Таблиця 15.5. Основні операції математичного співпроцесора i80387

Позначення і формат інструкції		Призначення
fld	st (x) mem32, 64, 80	Завантаження речового числа у стек співпроцесора.
fst	st (x) mem32, 64	Копіювання вмісту ST (0) у приймач.
fstp	st (x) mem32, 64, 80	Виштовхування вмісту ST (0) у приймач. Регістр ST (0) відмічається як пустий.
fild	mem16, 32, 64	Перетворення цілого числа у речовий формат і завантаження його у стек співпроцесора.
fist	mem16, 32, 64	Перетворення числа в ST (0) в ціле і копіювання його у приймач.

Продовження таблиці 15.5

<table>
<tr><td>fistp</td><td colspan="2">mem16, 32, 64</td><td>Перетворення числа в ST (0) в ціле і виштовхування його у приймач. Регістр ST (0) відмічається як пустий.</td></tr>
<tr><td>fbld</td><td colspan="2">mem80</td><td>Завантаження BCD-числа у стек співпроцесора.</td></tr>
<tr><td>fbstp</td><td colspan="2">mem80</td><td>Перетворення ST (0) у 80-бітове упаковане і виштовхування його у приймач. Регістр ST (0) відмічається як пустий.</td></tr>
<tr><td>fxch</td><td colspan="2">st (x)</td><td>Обмін місцями вмісту ST (0) і джерела. Якщо операнд не вказаний, обмінюються вмісти ST (0) і ST (1).</td></tr>
<tr><td rowspan="3">fadd
fsub
fmul
fdiv</td><td colspan="2">mem32, 64</td><td rowspan="3">Складання/віднімання/множення/ділення операндів. Одноопераднийй формат працює з ST (0). Результат поміщується у перший операнд. Безоперандний формат еквівалентний FADD ST (0), ST (1).</td></tr>
<tr><td>st (0)</td><td>st (x)</td></tr>
<tr><td>st (x)</td><td>st (0)</td></tr>
<tr><td>faddp
fsubp
fmulp
fdivp</td><td>st (x)</td><td>st (0)</td><td>Складання/віднімання/множення/ділення операндів і виштовхування ST (0). Результат поміщується у перший операнд. Безоперандний формат еквівалентний FADD ST (1), ST (0).</td></tr>
<tr><td>fiadd
fisub
fimul
fidiv</td><td colspan="2">mem16, 32</td><td>Складання/віднімання/множення/ділення операндів. Явний операнд – цілочисельна змінна, неявний – ST (0). Результат поміщується в ST (0).</td></tr>
<tr><td>fsubr
fsubrp
fisubr</td><td colspan="2">еквівалентні fsub/fsubp/fisub</td><td>Зворотне віднімання операндів. Операції еквівалентні fsub/fsubp/fisub, але здійснюють віднімання приймача з джерела, а не навпаки.</td></tr>
<tr><td>fprem
fprem1</td><td colspan="2">–</td><td>Обчислення часткового залишку від ділення ST (0) на ST (1). Вміст ST (1) послідовно віднімається від вмісту ST (0), поки ST (0) > ST (1), але не більше 54 разів. Результат залишається в ST (0). Якщо було отримано точний залишок, прапор C2 скидається, інакше – встановлюється. Операції відрізняються лише напрямком округлення.</td></tr>
<tr><td>fabs</td><td colspan="2">–</td><td>Визначення модуля (абсолютного значення) вмісту ST (0).</td></tr>
<tr><td>fchs</td><td colspan="2">–</td><td>Заміна знаку ST (0) на протилежний.</td></tr>
<tr><td>frndint</td><td colspan="2">–</td><td>Округлення вмісту ST (0) відповідно до поточного режиму.</td></tr>
<tr><td>fscale</td><td colspan="2">–</td><td>Множення ST (0) на 2 у степеню ST (1) і запис результату в ST (0).</td></tr>
<tr><td>fxtract</td><td colspan="2">–</td><td>Видобування експоненти і мантиси. Вміст ST (0) розділяється на мантису і експоненту, після чого мантиса потрапляє в ST (0), а експонента – в ST (1).</td></tr>
<tr><td>fsqrt</td><td colspan="2">–</td><td>Видобування квадратного кореня. Операнд і результат – ST (0).</td></tr>
</table>

Продовження таблиці 15.5

<table>
<tr><td>fsin
fcos</td><td>–</td><td>Обчислення синуса/косинуса від вмісту ST (0).
Результат заміщує ST (0).</td></tr>
<tr><td>fsincos</td><td>–</td><td>Обчислення синуса/косинуса від вмісту ST (0). Синус поміщується в ST (1), косинус – ST (0).</td></tr>
<tr><td>fptan</td><td>–</td><td>Обчислення тангенса від вмісту ST (0). Результат поміщується в ST (1), а в ST (0) записується 1.</td></tr>
<tr><td>fpatan</td><td>–</td><td>Обчислення арктангенса від частки ST (1) / ST (0).
Результат записується в ST (1), а ST (0) виштовхується 1</td></tr>
<tr><td>f2xmi</td><td>–</td><td>Обчислення $2^{ST\ (0)} - 1$. Значення операнду повинно лежати в межах від – 1 до 1. Результат заміщує ST (0).</td></tr>
<tr><td>fyl2x</td><td>–</td><td>Обчислення ST (1) × $\log_2$ST (0). Результат записується ST (1), а ST (0) виштовхується.</td></tr>
<tr><td>fyl2xp1</td><td>–</td><td>Обчислення ST (1) × $\log_2$ (ST (0) + 1). Результат записується ST (1), а ST (0) виштовхується.</td></tr>
<tr><td rowspan="3">fcom
fcomp
fcompp</td><td rowspan="3">st (x)
mem32, 64</td><td>Порівняння речових чисел/порівняння і виштовхування. Операції порівнюють вміст ST (0) з явним операндом і встановлюють прапори C0, C2 і C3 наступним чином:
<table>
<tr><td></td><td>C3</td><td>C2</td><td>C0</td></tr>
<tr><td>ST (0) > операнд
ST (0) < операнд
ST (0) = операнд
не можна порівняти</td><td>0
0
1
1</td><td>0
0
0
1</td><td>0
1
0
1</td></tr>
</table>
</td></tr>
<tr><td>Операція fcompp не має операндів, порівнює ST (0) з ST(1) і виштовхує їх.</td></tr>
<tr><td></td></tr>
<tr><td>ficom
ficomp</td><td>mem16, 32</td><td>Порівняння цілих чисел/порівняння і виштовхування. Результат еквівалентний fcom/fcomp.</td></tr>
<tr><td rowspan="3">fcomi
fcomip</td><td rowspan="3">st (x)</td><td>Порівняння речових чисел/порівняння і виштовхування та встановлення прапорів регістра EFLAGS. Операції порівнюють вміст ST (0) з явним операндом:
<table>
<tr><td></td><td>ZF</td><td>PF</td><td>CF</td></tr>
<tr><td>ST (0) > операнд
ST (0) < операнд
ST (0) = операнд
не можна порівняти</td><td>0
0
1
1</td><td>0
0
0
1</td><td>0
1
0
1</td></tr>
</table>
</td></tr>
<tr><td></td></tr>
<tr><td></td></tr>
<tr><td>ftst</td><td>–</td><td>Перевірка вмісту ST (0) на нуль. Після перевірки C3, C2 і C0 встановлюються аналогічно до інших операцій.</td></tr>
</table>

Продовження таблиці 15.5

<table>
<tr><td rowspan="10">fxam</td><td rowspan="10">–</td><td colspan="4">Аналіз вмісту ST (0) і встановлення прапорів за схемою:</td></tr>
<tr><td>тип числа</td><td>C3</td><td>C2</td><td>C0</td></tr>
<tr><td>не підтримується</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>не число</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>нормальне число</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>нескінченність</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>нуль</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>пустий регістр</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>денормалізоване</td><td colspan="3"></td></tr>
<tr><td>fincstp</td><td>–</td><td colspan="4">Збільшення покажчика вершини стека. Якщо покажчик дорівнював сім, він обнуляється. Операція не є еквівалентною виштовхуванню ST (0).</td></tr>
<tr><td>fdecstp</td><td>–</td><td colspan="4">Зменшення покажчика вершини стека. Якщо покажчик дорівнював нулю, він стає рівним сім.</td></tr>
<tr><td>Ffree</td><td>st (x)</td><td colspan="4">Помітка операнд як пустого в регістрі тегів TW. Вміст регістра та покажчик стека не змінюється.</td></tr>
<tr><td>fnop</td><td>–</td><td colspan="4">Відсутня операція.</td></tr>
</table>

Окрім операцій, перелічених в таблиці 5, математичний співпроцесор підтримує декілька операцій, призначених для завантаження у стек деяких часто використовуваних констант:

fld1 – помістити у стек 1,0;
fldz – помістити у стек +0,0;
fldpi – помістити у стек число π;
fldl2e – помістити у стек $\log_2 e$;
fldl2t – помістити у стек $\log_2 10$;
fldln2 – помістити у стек ln 2;
fldlg2 – помістити у стек lg 2.

Методичні вказівки

Завданням даної лабораторної роботи є розробка і реалізація мовою асемблера програми, призначеної для обчислення заданого математичного виразу з використанням інструкцій математичного співпроцесора.

Вихідні дані для обчислення рекомендується задавати в тілі програми за допомогою директиви оголошення змінних – DD (Define Double word – 32 біта, float), DW (Define Quadruple word – 64 біта, double) або DT (Define Ten bytes – 80 біт, long double). Застосування перелічених директив позбавляє програміста необхідності штучного формування розрядної сітки речового числа і дозволяє задавати його у звичайному вигляді:

```
…
float_number        dt      314.15926e-2
…
```

Після виконання обчислення результат слід переписати з вихідного регістра FPU в змінну. Аналіз результату можна виконати двома способами:

- запустити програму в середовищі де багера (наприклад, debug.exe або afd.exe), який дозволяє переглядати вміст оперативної памяті;
- передбачити у програмі запис результату в двійковий файл, після чого реалізувати мовою C/C++ найпростішу програму читання вмісту файлу та виводу його на дисплей. Для запису даних у двійковий файл можна скористатися наступними функціями переривання 21h DOS:

1) створення і відкриття нового файлу:
   - AH=5Bh;
   - CX – атрибут файлу (нуль);
   - DS:DX – адреса ASCII-рядка з повною назвою файлу;
   - на виході:AX – ідентифікатор файлу;
2) запис у файл або пристрій:
   - AH=40h;
   - BX – ідентифікатор файлу;
   - CX – кількість байтів для запису;
   - DS:DX – адреса буфера з даними для запису;
3) закриття файлу:
   - AH=3Eh;
   - BX – ідентифікатор файлу.

Фактично, функція 40h знайшла відображення в мові C у вигляді функції fwrite:

fwrite ((void*) buf, size_t size, size_t num, FILE* file);

Приклад виконання роботи

Завдання:

реалізувати програму, яка обчислює найближче значення числа π за наступною формулою: $p = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right)$ з точністю до 10^{-4}.

Розв'язання:

```
.model small
.286
.stack    100h
.data
          precision      dq     0.0001
          result         dq     0.0
          x              dq     1.0
          sign           dq     1.0
          koef           dq     4.0
          filename       db     'result.bin', 0
.code
; обчислення 1/x
          fld1
```

```
        fdiv    x
; порівняння результату із заданою точністю
        fcom    precision
; запис регістра SR FPU в регістр AX
        fstsw   ax
; копіювання AH у молодший байт FLAGS
        sahf
; задана точність досягнута – вихід з циклу
        jb      stop
; встановлення знаку
        fmul    sign
        fadd    result
        fstp    result
; збільшення x на 2 (x + 1 + 1)
        fld     x
        fld1
        fadd
        fld1
        fadd
        fstp    x
; зміна знаку "знакової" змінної
        fld     sign
        fchs
        fstp    sign
; продовження обчислення
        jmp     next
stop:
; множення накопиченої суми на 4
        fld     result
        fmul    koef
        fstp    result
; створення і відкриття файлу
        mov     ah, 5Bh
        xor     cx, cx
        lea     dx, filename
        int     21h
; запис файлового ідентифікатора в стек для подальших операцій
        push    ax
        mov     bx, ax
; запис результату в файл
        mov     ah, 40h
        mov     cx, 8
        lea     dx, result
        int     21h
; закриття файлу
```

```
        mov     ah, 3Eh
        pop     bx
        int     21h
; завершення роботи програми
        mov     ah, 4Ch
        int     21h
end
```

Текст програми мовою С для виводу вмісту файлу на екран:

```
# include <stdio.h>
FILE *fpufile;
double y;
voide main ()
{
        fpufile = fopen ("result.bin", "rb");
        fread ((void*) &y, sizeof (double), 1, fpufile);
        fclose (fpufile);
        printf ("y = %lf\n", y);
}
```

Варіанти завдань

Реалізувати програму, яка обчислює наближену суму елементів рядку [1] з точністю [2] (тобто запис – 3 слід розуміти як 10^{-3}).

Значення параметрів за варіантами

№ вар	Параметри	
	1	2
1	$1+\frac{x}{1!}+\frac{x^2}{2!}+\cdots$	– 3
2	$1+\frac{x\cdot ln\ a}{1!}+\frac{(x\cdot ln\ a)^2}{2!}+\cdots$	– 6
3	$\sin x\ +\ \frac{\sin\sin x}{2}+\ \frac{\sin\sin\sin x}{2}+\ \ldots$	– 9
4	$1+\frac{2}{1!}+\frac{4}{2!}+\frac{8}{3!}+\cdots$	– 3
5	$\frac{2}{3}\sin 2x-\frac{3}{8}\sin 3x+\frac{4}{15}\sin 4x-\cdots$	– 6
6	$\frac{1}{x}-\ \frac{1}{3x^3}+\frac{1}{5x^5}-\cdots$	– 9
7	$\cos x+\ \frac{\cos 2x}{4}+\frac{\cos 3x}{9}+\cdots$	– 3
8	$1-\ \frac{(p/6)^2}{2!}+\ \frac{(p/6)^4}{4!}-\cdots$	– 6
9	$x\cos\ \frac{p}{3}+\ \frac{x^2}{2}\cos\frac{2p}{3}+\ \frac{x^3}{3}\cos p+$	– 9
10	$1+\frac{x^2}{2!}-\ \frac{3x^4}{4!}+\frac{5x^6}{6!}\ldots$	– 3
11	$1+\ \frac{\cos x}{1!}+\frac{\cos 2x}{2!}+\cdots$	– 6
12	$\frac{x^3}{3}-\ \frac{x^5}{15}+\frac{x^7}{35}-\cdots$	– 9
13	$1+\frac{x}{1!}\cos\frac{p}{4}+\ \frac{x^2}{2!}\cos\frac{p}{2}+\ \frac{x^3}{3!}\cos\frac{3p}{4}+\cdots$	– 3
14	$\frac{(2x)^2}{2!}+\ \frac{(2x)^4}{4!}+\ \frac{(2x)^6}{6!}+\ \ldots$	– 6
15	$\frac{\cos 2x}{1\cdot 3}+\ \frac{\cos 4x}{3\cdot 5}+\ \frac{\cos 6x}{5\cdot 7}$	– 9

Перелік рекомендованої літератури

1. Лю Ю-Чжен, Гибсон Г. Микропроцессоры семейства 8086/8088. Архитектура, программирование и проектирование микрокомпьютерных систем: Пер. с англ. – М.: – Радиосвязь, 1987. – 512 с.; ил.

2. Пирогов В. Ю. Ассемблер. Учебный курс. – СПб.: БХВ-Петербург, 2003. – 1056 с. ил.

3. Юров В. Assembler: учебный курс – СПб: Питер Ком, 1999. – 672 с.

Навчально-методичне видання

Зибін Сергій Вікторович

Асемблер для x86 і Pentium

Методичні вказівки та завдання до виконання лабораторних робіт з дисципліни "Системне програмування"

студентів спеціальності 7.091501 "Комп'ютерні системи та мережі"

Технічний редактор Чирков Д.В.
Коректор Капустян М.В.

Підписано до друку 01.09.2011 р. Формат 64×84/16, папір офсетний
Друк офсетний
Умовн. друк. арк. 1,5. Обл. вид. арк. 1,2
Наклад 300 прим. Замовлення № 09/11

Видавництво ДУІКТ
03110, м. Київ, вул. Солом'янська, 7.
Надруковано видавництвом ДУІКТ.
03110, м. Київ, вул. Солом'янська, 7.
Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру
Серія ДК № 2539 від 26.06.2006 р.