МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Запорізький національний технічний університет

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт з дисципліни

“ Архітектура комп’ютера ”

для студентів напряму 6.050103

“Програмна інженерія”

денної форми навчання

2011

Методичні вказівки до лабораторних робіт з дисципліни “Архітектура комп’ютера” для студентів напряму 6.050103 “Програмна інженерія” денної форми навчання /Уклад.: Рисіков В.П., Степаненко О.О., Качан О.І. – Запоріжжя: ЗНТУ, 2011. – 51 с.

Укладачі: В.П. Рисіков, доцент, к.т.н.,
О.О. Степаненко, доцент, к.т.н.,
О.І. Качан, асистент

Рецензент: С.К. Корнієнко, доцент, к.т.н.

Відповідальний
за випуск: А.В. Притула, доцент, к.т.н.

Затверджено
на засіданні кафедри
"Програмні засоби"

Протокол № 1 від 29.08.2011 р.

ЗМІСТ

Турбо – Асемблер є могутнім асемблером, працюючим із командним рядком, який сприймає ваші початкові данні (файли із розширенням .ASM) та робить з них об’єктні модулі (файли із розширенням .OBJ). Після цього ви маєте можливість використовувати програму - компоновщик фірми Borland TLINK.EXE, яка відрізняється високою швидкістю компоновки, для компоновки отриманих об’єктних модулів та створення виконуємих файлів. (файлів з розширенням .EXE та .COM)

ЗАГАЛЬНІ ВІДОМОСТІ ПРО ТУРБО АСЕМБЛЕР

Турбо – Асемблер створений для роботи з процесорами серії 80x86 та 80x87 (детальніше набір інструкцій процесорів серії 80x86/80x87 описаний у інструкціях фірми Intel). У Турбо Асемблері є дуже могутній та гнучкий синтаксис командного рядка. Якщо ви запустите Турбо Асемблер, не задаючи ніяких параметрів, наприклад

TASM,

то на екран виведеться довідкова інформація, (англійською мовою), яка описує множину параметрів командного рядка та синтаксис для специфікації файлів які асемблюються.

TASM [параметри] вик_файл [,об’єкт_файл] [,лістинг] [пер_посилки]

/a,/s Упорядкування елементів по алфавітному порядку чи порядку початкового коду
/c Генерація у лістингу перехресних посилань
/dSYM[=VAL] Визначається SYM=0 чи SYM=VAL
/e/r Емулюємі чи дійсні інструкції з плаваючою точкою
/h/? Виводиться дана довідкова інформація
/lPATH Файли, які включаються, шукаються по маршруту визначеному PATH
/jCMD Визначає початкову директиву Асемблеру (наприклад IDEAL)
/kr#,/ks# Місткість хеш-таблиці #, місткість об’єму рядка #

/l/la Генерація лістингу: l-звичайний, la-розширений

/ml/mx/mu Розрізнимость у регістрі букв ідентифікаторів: ml=усі, mx=глобальні, mu=не розрізняються

/mv# Задає максимальну довжину ідентифікаторів.

/m# Дозволяє виконання декількох проходів для дозволення випереджаючих посилань

/h Вилучення у лістингах таблиці символів (ідентифікаторів)

/p Перевірка перекриття сегменту коду у захищеному режимі

/q Вилучення записів .OBJ не потрібних при компоновці

/t Вилучення повідомлень при успішному асемблюванні

/w0,/w1,/w2 Завдання рівня попередження w0=нема попереджень, w1=w2=є попередження

/w-xxx, /w+xxx Заборона чи дозвіл попередження типу xxx

/x Включення у листинги блоків умовного асемблювання

/zi, /zd Інформація про ідентифікатори для відлагодження: zi=повна, zd=тільки про номери рядків

ЛАБОРАТОРНА РОБОТА № 1

Вивчення відлагоджувача DEBUG

МЕТА РОБОТИ: Навчитися відлагоджувати програми за допомогою відлагоджувача DEBUG.

ТЕОРІЯ

Ключі до відлагоджувача DEBUG:

допомога	?
ввід команд асемблера	A[адреса]
порівняти	C діапазон адрес
отримати байт пам’яті	D[діапазон]
редагувати чарунки пам’яті	E адреса [список байт]
виконати зі вказаної адреси	G[=адреса][адреса]
скопіювати	M діапазон адрес
вихід	Q
зміст регістрів	R[регістр]
пошук	S
трасування	T[=адреса][число повторювань]
дізасемблер	U [діапазон адрес]

ЗАВДАННЯ

Самостійно вивчити відлагоджувач Debug.

Практично продемонструвати роботу з відлагоджувачем DEBUG викладачу.

ЛАБОРАТОРНА РОБОТА № 2

Вивчення режимів адресації i8086 та команд пересилання даних

МЕТА РОБОТИ: Вивчити режими адресації МП і 8086 та команди пересилання даних, а також навчитися застосовувати отримані знання при написанні програм на мові асемблер.

ТЕОРІЯ

Директива SEGMENT

Директива SEGMENT визначає початок сегменту. Мітка, яка вказується у даній директиві, визначає початок сегменту. Наприклад, директива:

Cseg SEGMENT

Визначає початок сегменту з ім’ям Cseg. Директива SEGMENT може також (необов’язково) визначати атрибути сегмента, включаючи вирівнювання у пам’яті на кордон байта, слова, подвійного слова, параграфа (16 байт) чи сторінки (256 байт). Інші атрибути включають в себе спосіб, за допомогою якого сегмент буде комбінуватися з іншими сегментами з тим же ім’ям та класом сегмента.

Директива ENDS

Директива ENDS визначає кінець сегменту. Наприклад:

Cseg ENDS

завершує сегмент з ім’ям Cseg, який починався по директиві SEGMENT. При використанні стандартних директив визначення сегментів, ви повинні явним чином завершити кожний сегмент.

Директива ASSUME

Директива ASSUME вказує Турбо Асемблеру, у значення якого сегменту встановлений даний сегментний регістр. Директиву ASSUME CS: потрібно указати у кожній програмі, в якої використовуються стандартні сегментні директиви, бо Турбо Асемблеру необхідно знати про сегмент коду для того, щоб встановити програму яка виконується. Крім того, звичайно використовуються директиви ASSUME DS: та ASSUME ES:, завдяки яким Турбо Асемблер знає до яких чарунок пам’яті ви можете адресуватись у даний момент. Директива ASSUME дозволяє Турбо Асемблеру перевірити допустимість кожного звернення до іменованої чарунки пам’яті з урахуванням значення поточного сегментного регістру. Роздивимось наступний приклад:

```
Data1                   SEGMENT WORD ‘DATA’
Var1                    DW 0
Data1                   ENDS
Data2                   SEGMENT WORD ‘DATA’
var2                    DW 0
Data2                   ENDS
Code                    SEGMENT WORD ‘CODE’
                        ASSUME CS:CODE
                        Program Start:
                        mov ax,Data1
                        mov ds,ax  ; установити DS у Data1
ASSUME DS:Data1
                        mov ax,[Var2]
                        ; спроба завантажити Var2 у AX
                        ; це призведе до помилки, бо
                        ; Var2 недоступна у сегменті
                        ; Data1
```

Турбо Асемблер відзначає у цій програмі помилку, бо у ній робиться спроба отримати доступ до змінної пам’яті Var2, коли регістр DS встановлений у значення сегменту Data1 (до Var2 неможливо адресуватись, допоки DS не буде встановлений у значення сегмента Data2).

Виділення даних

Тепер, коли відомо, як створювати сегменти, роздивимось, як можна заповнювати ці сегменти усвідомленими даними. Сегмент стеку проблему не являє: там знаходиться стек, а до стеку ви можете звертатися за допомогою інструкцій PUSH та POP та адресуватися через регістр BP. Сегмент коду заповнюється інструкціями які генеруються згідно з мнемонікою інструкцій вашої програми, тому проблеми тут також немає. Залишається сегмент даних. У Турбо Асемблері передбачена множина способів визначення змінних у сегменті даних, як ініціалізуємих деяким значенням, так і неініціалізованих. Щоб зрозуміти які данні дозволяє вам визначити Турбо Асемблер, ми повинні спочатку трохи розповісти вам про головні типи даних Асемблеру.

Ініціалізовані дані

Тепер ми готові до того, щоб роздивитися способи, за допомогою яких в Турбо Асемблері можна визначити змінні. Давайте спочатку роздивимось визначення ініціалізованих даних.

Директиви визначення даних DB, DW, DD, DF, DP, DQ та DT дозволяють нам визначити змінні у пам’яті різного розміру:

DB 1 байт
DW 2 байта=1 слово
DD 4 байта=1 подвійне слово
DF, DP 6 байт=1 показник дальнього типу (386)
DQ 8 байт=одне четвертне слово
DT 10 байт

Наприклад

ByteVar DB ‘Z’ ; 1 байт
WordVar DW 101b ; 2 байта (1 слово)
DwordVar DD 2BFh ; 4 байта (1 подвійне слово)
QwordVar DQ 307o ;8 байт (одне четвертне слово)
TwordVar DT 100 ; 10 байт

Ініціалізація масивів

В одній директиві визначення даних може вказуватися декілька значень. Наприклад директива:

```
SampleArray        DW 0, 1, 2, 3, 4
                   DD 25, 36, 49, 64, 81
                   DD 100, 121, 144, 169, 196
```

створює масив з ім’ям SampleArray, елементи якого мають розмір в слово. В директивах визначення даних можна використовувати будь-яке число значень.

Турбо Асемблер дозволяє вам визначити блок пам’яті, ініціалізованої вказаним значенням, за допомогою операції DUP. Наприклад:

```
BlankArray DW 100h DUP (0)
```

Тут створюється масив BlankArray, який складається з 255 (десят.) слів, ініціалізованих значенням 0. Аналогічно директива:

```
ArrayofA DB 92 DUP (‘A’)
```

створює масив з 92 байт, кожний з яких ініціалізований символом A.

Ініціалізація рядків символів

Роздивимось тепер створення рядків символів. Символи являють собою допустимі операнди директив визначення даних, тому рядок символів можна визначити наступним чином:

```
String DB ‘A’, ‘B’, ‘C’, ‘D’
```

В Турбо Асемблері в цьому випадку передбачена також зручна скорочена форма:

```
String DB ‘ABCD”
```

Режими адресації пам’яті

Як при використанні операнда у пам’яті задати ту чарунку пам’яті з якою ви бажаєте працювати? Очевидна відповідь полягає в тому, щоб привласнити потрібній змінній у пам’яті ім’я, як ми це робили у останньому розділі. За допомогою, наприклад, наступних операторів ви можете вирахувати змінну пам’яті Debts (борги) зі змінної пам’яті Assets (майно):

```
Assets DW ?
Debts DW ?
        .       .
mov ax,[Debts]
sub [Assets],ax
```

В дійсності мова Асемблера забезпечує декілька різних способів адресації до рядків символів, масивів та буферів даних.

Найбільш простіший спосіб полягає в тому, щоб зчитати дев’ятий по рахунку символ строки CharString:

```
CharString DB ‘ABCDEFGHIJKLM’
                                .
                                .
mov ax,@Data
mov ds,ax
mov al,[CharString+8]
```

В даному випадку це теж саме, що:

```
mov al,[100+8]
```

так як CharString починається зі зміщенням 100. Все, що замкнено у квадратні дужки, інтерпретується Турбо Асемблером, як адреса, тому зміщення Charstring та 8 складаються та використовується в якості адреси пам’яті. Інструкція приймає вигляд:

```
mov al,[108]
```

Такий тип адресації, коли чарунка пам’яті створюється її ім’ям, плюс деяка константа, зветься безпосередньою (прямою) адресацією. Хоча безпосередня адресація – це добрий метод, вона не відрізняється достатньою гнучкістю, тому що звернення виконується кожний раз в одній й тій же адресі пам’яті. Тому давайте роздивимось другий, більш гнучкий шлях адресації пам’яті.

Розглянемо наступний фрагмент програми, де у регістр AL також завантажується дев’ятий символ CharString:

```
mov bx, OFFSET CharString+8
mov al,[bx]
```

У даному прикладі для посилання на дев’ятий символ використовується регістр bx. Перша інструкція завантажує у регістр bx зміщення CharString (згадайте про те, що операція OFFSET повертає зміщення мітки у пам’яті), плюс 8. (Обчислення OFFSET та складання для цього виразу виконується Турбо Асемблером підчас асемблювання.) Друга інструкція визначає, що AL потрібно скласти зі змістом по зміщенню в пам’яті, на яке вказує регістр BX

```
mov AL,[108]
```

Квадратні дужки вказують, що в якості операнда – джерела повинна бути використовуватися чарунка, на яку вказує регістр BX, а не сам регістр BX. Не забувайте вказувати квадратні дужки при використанні BX в якості показника пам’яті. Наприклад:

```
mov ax,[bx]          ; завантажити AX з чарунки пам’яті,
                     ; на яку вказує BX
та
mov ax,bx            ; завантажити у AX зміст
                     ; регістра BX
```

це дві цілком різні інструкції.

BX – це не єдиний регістр, котрий можна використовувати для посилання на пам’ять. Допускається також використовувати разом з необов’язковим значенням-константою або міткою регістри BP, SI та DI. Загальний вигляд операндів у пам’яті виглядає наступним чином:

[базовий регістр + індексний регістр + зміщення],

де базовий регістр – це BX, індексний регістр – це SI чи DI, а зміщення – будь-яка 16-бітова константа, включно мітки та вирази. Кожний раз, коли виконується інструкція, яка використовує операнд в пам'яті, процесором 8086 ці три компоненти складаються. Кожна з трьох частин операнда у пам'яті є необов'язковою, хоча (це очевидно) ви повинні використовувати один з трьох елементів, інакше ви не отримаєте адресу у пам'яті. Ось як елементи операнда у пам'яті виглядають у іншому форматі:

BX		SI
чи+	чи	+Зміщення
BP		DI
(база)		(індекс)

Існують 16 способів завдання адреси у пам'яті:

[зміщення]	[bp+зміщення]
[bx]	[bx+зміщення]
[si]	[si+зміщення]
[di]	[di+зміщення]
[bx+si]	[bx+si+зміщення]
[bx+di]	[bx+di+зміщення]
[bp+si]	[bp+si+зміщення]
[bp+di]	[bp+di+зміщення]

Де зміщення – це те що можна привести до 16-бітового постійного значення.

Може здаватися, що 16 режимів адресації – це дуже багато, але якщо ще раз подивитесь на цей список, то побачите, що усі режими адресації отримують всього з декількох елементів, які комбінуються різними шляхами. Ось ще декілька способів, за допомогою яких можна, використовуючи різні режими адресації, завантажити дев'ятий символ рядка CharString у регістр AL.

```
CharString DB ‘ABCDEFGHIJKLM’,0
mov ax,@Data
mov ds,ax
mov si,OFFSET CharString+8
mov al,[si]

mov bx,8
mov al,[CharString+bx]

mov bx,OFFSET CharString
mov al,[bx+8]

mov si,8
mov al,[CharString+si]

mov bx,OFFSET CharString
mov di,8
mov al,[bx+di]

mov si, OFFSET CharString
mov bx,8
mov al,[si+bx]

mov bx,OFFSET CharString
mov si,7
mov al,[bx+si+1]

mov bx,3
mov si,5
mov al,[bx+CharString+si]
```

Усі ці інструкції посилаються на одну й ту ж чарунку пам’яті [CharString]+8.

У даному прикладі можна знайти декілька цікавих моментів. По-перше, повинні розуміти, що знак плюс (+), який використовується всередині квадратних дужок, має спеціальне значення.

ПОРЯДОК ВИКОНАННЯ РОБОТИ

Зарезервувати місце в пам’яті у сегменті даних Dseg_1 для масиву байт у 10 елементів MasB.

Зарезервувати місце у пам’яті у другому сегменті даних Dseg_2 для масиву слів у 30 елементів з ім’ям MasW.

Зарезервувати місце у додатковому сегменті Eseg для таблиці, складеної з 10 записів по 4 елемента TablB.

Зарезервувати місце у додатковому сегменті Eseg для таблиці подвійних слів з ім’ям TablAdr 40 елементів.

Використовуючи пряму адресацію з індексуванням заповнити масив MasB.

Використовуючи базову адресацію зі зміщенням заповнити MasW.

Використовуючи базову адресацію з індексуванням заповнити TablB наступним чином: в якості змісту 1-го елементу кожного запису береться елемент MasB з тим же номером, що й номер запису. В якості змісту решти полів береться перший байт елементів MasW по порядку їх слідування у масиві.

Використовуючи навперемінно, якщо це можливо чи команду завантаження абсолютної адреси, чи псевдооператори SEG, OFFSET заповнити таблицю TablAdr так, щоб кожний елемент таблиці удавав собою адресу відповідного елементу таблиці TablB

Використовуючи операції зі стеком спочатку залишаємо в TablB порядок елементів у записах незмінними, змінити в зворотному порядку розташування записів, а потім не змінюючи розташування записів змінити на зворотній порядок елементів.

ЗМІСТ ЗВІТУ

Постановка задачі.
Текст програми, вихідні дані.
Тести та результати відлогодження.
Блок-схема програми.
Результати рішення на ЕОМ.

КОНТРОЛЬНІ ПИТАННЯ

Директива SEGMENT та її параметри.
Директива ENDS.
Директива ASSUME.
Заголовок EXE програми.
Визначення даних.
Способи адресації.
Команда MOV, призначення, обмеження при використанні.
Стек, команди роботи зі стеком.

ЛАБОРАТОРНА РОБОТА № 3

Арифметичні операції мікропроцесора i8086

МЕТА РОБОТИ: Вивчити набір арифметичних операцій i8086.

ТЕОРІЯ

Переміщення даних – це звичайно, важлива функція, тому що комп'ютер витрачає істотну частину свого часу на переміщення даних з одного місці до другого. Однак в рівній мірі важливо мати можливість маніпулювати даними, виконуючи над ними арифметичні та логічні операції, які підтримуються процесором i8086.

Арифметичні операції

Навіть якщо ваш комп'ютер PC і не витрачає часу на роботу з числами та обчислювальні операції, ви знаєте, що він може це зробити, якщо вам це буде потрібно. Крім того, на комп'ютері PC може працювати множиною електронних таблиць, програм баз даних та інженерних пакетів. Якщо прийняти все це до уваги, то стає достатньо очевидним, що комп'ютер IBM PC повинен володіти міцними обчислювальними здібностями. Крім того, у процесорі 8086 не передбачено арифметичних та логічних інструкцій, котрі можуть безпосередньо працювати з операндами, розмір яких перевищує 16 біт.

Складання та віднімання

В багатьох примірниках програм ми вже зустрічалися з операндами ADD (складання) та SUB (віднімання). Їх дія відповідає назві. Інструкція ADD виконує складання операнда джерела (правого операнда) зі змістом операнда-приймача. Інструкція SUB робить навпаки - відніме операнд – джерело з операнда-приймача.

Наприклад, інструкції

```
BaseVal DW 99
Adjust DW 10
```

```
mov dx,[baseval]
add dx,11
sub dx,[Adjust]
```

спочатку завантажують значення, записане у BaseVal, в регістр DX, потім додають до нього константу 11 (в результаті у DX виходить значення 110) та, зрештою, віднімають з DX значення 10, записане у змінній Adjust. Отримане в результаті значення 100 зберігається у регістрі DX.

32-разрядні операції

Операції ADD та SUB працюють з 8-ми розрядними операндами. Якщо ви, наприклад, бажаєте скласти чи відняти 32-розрядні операції, ви повинні розбити операцію на ряд операцій зі значеннями розміром у слово та використовувати інструкції ADC та SBB.

Коли ви складаєте два операнда, процесор 8086 записує стан у флаг переносу (біт C у регістрі флагів), котре показує, чи був виконаний перенос з приймача. Ви знайомі з принципом переносу в десятковій арифметиці: якщо ви складаєте 90 та 10, то виходить перенос в третю цифру (розряд).

Інструкція ADC аналогічна інструкції ADD, але в ній не враховується флаг переносу (спочатку встановлений попереднім складанням). Всякий раз коли ви складаєте два значення, які перевищують по розміру слово, то молодші (менш значущі) слова треба скласти за допомогою інструкції ADD, а решта слів цих значень - за допомогою однієї чи декількох інструкцій ADC, останніми складаємо самі значущі слова. Наприклад, наступні інструкції складають значення у регістрах CX:BX, розміром у подвійне слово, зі значеннями, записаними у регістрах DX:AX:

```
add ax,bx
adc dx,cx
```

а в наступній групі інструкцій виконується складання четверного слова у змінній DuobleLong1 з четвертним словом у змінній DoubleLong2:

```
mov ax,[Doblelong1]
add [DoubleLong2],ax
mov ax,[DobleLong1+2]
adc [DoubleLong2+2],ax
mov ax,[DoubleLong1+4]
adc [DobleLong1+4],ax
mov ax,[DoubleLong1+6]
adc [DoubleLong2+6],ax
```

Інструкція SBB працює по тому ж принципу, що й інструкція ADC. Коли інструкція SBB виконує віднімання, в ній ураховується позика, яка відбулася у попередньому відніманні. Наприклад, наступні інструкції віднімають значення, записане у регістрах CX:BX, зі значення розміром в подвійне слово, записаного у регістрах DX:AX:

```
sub ax,bx
sbb dx,cx
```

При роботі з інструкціями ADC та SBB ви повинні упевнитися, що флаг переносу не змінився з моменту виконання останнього складання чи віднімання, інакше стан позики/переносу, який зберігається у флагу переносу, буде втрачено. Наприклад, у наступному фрагменті програми складення CX:BX з DX:AX виконується некоректно.

```
add ax,bx        ; скласти молодші слова
sub si,si        ; очистити SI (флаг переносу
                 ;скидається в 0)
adc dx,cx        ; скласти старші слова
                 ; так це буде працювати некоректно
                 ; так як з моменту останньої
                 ; операції складення зміст
                 ; флаг переносу втрачено
```

Збільшення та зменшення

Іноді у програмі на Асемблері буває потрібно виконати складання, яке складається просто у додатку до операнду значення 1.

Така операція зветься збільшенням (інкрементацією). Аналогічно, зі змісту регістрів та змінних у пам'яті іноді потрібно відняти значення 1. Така операція зветься зменшенням (декрементацією). Для таких операцій, як зміна змісту лічильника чи просування регістрів-вказівників по пам'яті чи операнди складання та віднімання можна виконувати за допомогою збільшення чи зменшення. Для виконання таких часто необхідних дій у наборі інструкцій процесорів 8086 передбачені дві інструкції – INC (збільшити) та DEC (зменшити). Інструкція INC додає до регістру чи змінної у пам'яті 1, а інструкція DEC віднімає з регістрів чи змінної 1.

Наприклад, наступна програма заповнює 10-байтовий масив TempArray числами 0, 1, 2, 3, 4, 5, 6, 7, 8, 9:

```
TempArray  DB  10 DUP (?)
FillCount DW ?

mov al,0                    ; перше значення,
                            ; записується у TempArray
mov bx,OFFSET TempArray     ; BX вказує на TempArray
mov [FillCount],10          ; число елементів якими потрібно
                            ; заповнити масив
FillTempArrayLoop:
mov [bx],al                 ; установити поточний елемент
                            ; TempArray
inc bx                      ; посилка на наступний елемент
                            ; масиву TempArray
inc al                      ; наступне значення, яке
                            ; записується
dec [FillCount]             ; зменшити лічильник числа
                            ; елементів, які заповнюються
jnz FillTempArray           ; обробити наступний елемент,
                            ; якщо ми ще не заповнили всі
елементи.
```

Чому переважно використовувати інструкцію:

```
inc bx
```

а не інструкцію:

```
add bx,1
```

вони ж роблять одне й теж? Це так, але в той же час, як інструкція ADD займає 3 байта, інструкція INC займає тільки 1 байт та виконується швидше. Фактично, більш економно виконати дві операції INC, ніж додати до регістра розміром у слово значення 2 (збільшення чи зменшення регістрів та змінних у пам'яті розміром у байт займає 2 байта, що також коротше, ніж складання та віднімання).

Коротше кажучи, інструкції INC та DEC – це найбільш ефективні інструкції, за допомогою яких можна збільшувати та зменшувати значення змінних у пам'яті та регістрів. Їх слід використовувати там, де це можливо.

Множення та ділення

Процесор 8086 може виконувати окремі типи операцій множення та ділення, Це одна з сильних сторін процесору 8086, тому що в багатьох мікропроцесорах взагалі відсутня безпосередня підтримка операції множення та ділення, а ці операції досить складно виконати програмним шляхом.

Інструкція MUL перемножує 8- чи 16- бітові беззнакові співмножники, створюючи 16- чи 32-бітовий добуток. Давайте спочатку роздивимось множення 8-бітових співмножників.

При 8-бітному (8-розрядному) множенні один з операндів повинен зберігатися у регістрі AL, а інший може являти собою будь-який 8-бітовий загальний регістр чи змінну пам'яті відповідного розміру. Інструкція MUL завжди зберігає 16-бітовий добуток у регістрі AX. Наприклад, в фрагменту програми:

```
mov al,25
mov dh,40
mul dh
```

AL множиться на DH, а результат (1000) розміщується у регістрі AX. Зауважимо, що в інструкції MUL потрібно указувати тільки один

операнд, другий співмножник завжди зберігається у регістрі AL (чи в регістрі AX у випадку перемноження 16-бітових множників).

Інструкція перемноження 16-бітових співмножників працює аналогічно. Один зі співмножників повинен зберігатися у регістрі AX, а другий може знаходитися у 16-розрядному загальному регістрі чи у змінній пам'яті. 32-бітовий добуток інструкція MUL розміщує в цьому випадку в регістрі DX:AX, при цьому молодші (менш значущі) 16 бітів додатку записуються у регістр AX, а старші (більш значущі) 16 біт – у регістр DX. Наприклад, інструкції:

```
mov ax,1000
mul ax
```

завантажують у регістр AX 1000, а потім підносять його у квадрат, розміщуючи результат (значення 1000000) у регістри DX:AX.

На відміну від складання та віднімання, у операції множення не враховується, чи являються співмножники операндами зі знаком чи без знака, тому є друга інструкція множення IMUL для множення 8-ми та 16-бітових співмножників зі знаком. Якщо не приймати до уваги, що перемножуються значення зі знаком, інструкція IMUL працює аналогічно інструкції MUL. Наприклад, при виконанні інструкцій:

```
mov al,-2
mov ah,10
imul ah
```

у регістрі AX буде записано значення –20. Процесор 8086 дозволяє нам з певними обмеженнями розділити 32-бітове значення на 16-бітове, чи 16-бітове значення на 8-ми бітове. Давайте спочатку розглянемо ділення 16-бітового значення на 8-ми бітове.

При беззнаковому діленні 16-бітового значення на 8-ми бітове ділене повинно бути записано у регістр AX. 8-ми бітовий дільник може зберігатися у любому 8-бітовому загальному регістрі чи змінній у пам'яті відповідного розміру. Інструкція DIV завжди записує 8-бітову частину у регістр AL, а 8-бітовий залишок – в AH. Наприклад, в результаті виконання інструкцій:

```
mov ax,51
mov dl,10
```

```
div dl
```

результат 5 (51/10) буде записаний у регістр AL, а залишок 1 (залишок від ділення 51/10) – у регістр AH.

Зауважимо, що частка являє собою 8-бітове значення. Це означає, що результат ділення 16-бітового операнда на 8-бітовий операнд повинен перевищити 255. Якщо частка занадто велика, то генерується переривання 0 (переривання по діленню на 0).

Інструкції:

```
mov ax,0fffh
mov bl,1
div bl
```

генерують переривання по діленню на 0 (як можна очікувати, переривання по діленню на 0 генерується також, якщо 0 використовується в якості дільника).

При діленні 32-бітового операнда на 16-бітовий операнд ділимо повинно записуватися у регістрах DX:AX. 16-бітовий дільник може знаходитися у будь-якому з 16-бітових регістрів загального призначення чи у змінній пам’яті відповідного розміру. Наприклад, у результаті виконання інструкцій:

```
mov ax,2
mov dx,1                    ; загрузити у DX:AX 10002h
mov bx,10h
div bx
```

вчасне 1000h (результат ділення 100002h на 10h) буде записано у регістрі AX, а 2 (залишок від ділення) у регістрі DX.

Як при множенні, при діленні має значення, використовуються операнди зі знаком чи без знака. Для ділення беззнакових операндів використовується операція DIV, а для ділення операндів зі знаком – IDIV. Наприклад, операції:

```
TestDivisor DW 100
```

```
mov ax,-667
cwd                    ; встановити DX:AX у значення –667
idiv [TestDivisor]
```

зберігають значення –6 у регістрі AX і значення –67 у регістрі DX.

Логічні операції

Турбо Асемблер підтримує повний набір інструкцій для виконання логічних операцій, включно інструкцій для виконання логічних операцій , включно інструкції AND (і), OR (чи), XOR (виключне чи) та NOT (ні). Ці інструкції можуть виявитися дуже корисними при роботі з окремими бітами слова чи байта, а також для виконування операцій булевої алгебри.

Результати виконання логічних операцій показані у таблиці. Логічна інструкція виконує порозрядні операції над бітами вихідних операндів. Наприклад, інструкція:

```
and ax,dx
```

виконує логічну операцію AND з бітом 0 регістра AX ті бітом 0 регістра DX, потім ту ж саму операцію з бітами 1, 2 і т.п. до біта 15.

Виконання логічних інструкцій процесора 8086 AND, OR та XOR наведено в табл. 3.1

Таблиця 3.1 – Виконання логічних інструкцій процесора 8086 AND, OR та XOR

Вихідний біт A	**Вихідний біт B**	**A AND B**	**A OR B**	**A XOR B**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Інструкція AND комбінує два операнди у відповідності з правилами, які показані у таблиці, встановлює кожний біт результату (операнда-приймача) в 1 тільки у тому випадку, якщо обидва біта операнда-джерела дорівнюють 1. Інструкція AND дозволяє нам виділити окремий біт чи примусово встановити його у значення 0. Наприклад, інструкції:

```
mov dx,3dah
in al,dx
and al,1
```

виділяють біт 0 байта стану кольорового графічного адаптеру (CGA). Ці інструкції залишають регістр AL встановлений у значення 1, якщо відеопам'ять адаптера CGA можна змінити, не викликаючи перешкод на екрані ("сніг"), та встановлює його у нульове значення у противному разі.

Інструкція OR теж комбінує два операнди у відповідності з правилами, які приведені у таблиці, встановлює кожний біт операнда-приймача у значення 1, якщо будь-який з відповідних бітів операнда-джерела дорівнює 1. Інструкція OR дозволяє вам примусово встановити окремі біти у значення 1. Наприклад, інструкції:

```
mov ax,40h
mov ds,ax
mov bx,10h
or WORD PTR [bx],0030h
```

встановлює біти 5 та 4 слова флагів апаратури базової системи вводу-виводу BIOS у значення 1. При цьому BIOS буде підтримувати монохромний дисплейний адаптер.

Інструкція XOR також комбінує два операнди у відповідності з правилами, які приведені у таблиці, встановлюючи кожний біт операнда-приймача у значення 1, тільки у тому випадку якщо один з відповідних бітів операнда-джерела дорівнює 0, та у значення 1 у протилежному випадку. Інструкція XOR дозволяє нам "перемикати" значення окремих бітів у байті. Наприклад, інструкції:

```
mov al,01010101b
xor al,11110000b
```

встановлюють регістр AL у значення 10100100b чи A5h. Коли для регістра AL виконується операція XOR зі значенням 11110000b (0F0h), біти зі значенням 0 залишають відповідні біти AL незмінними.

До речі інструкція XOR дає зручний спосіб обнуління змісту регістру. Наприклад, наступна інструкція встановлює зміст регістра AX у значення 0:

```
xor ax,ax
```

Нарешті, інструкція NOT просто змінює значення кожного біту операнда на протилежне (як якщо б над початковим операндом була виконана операція XOR зі значенням 0FFh). Наприклад:

```
mov bl,10110001b
not bl                  ; переключити BL у 01001110b
xor bl,0ffh             ; переключити BL зворотно у значення
                        ; 10110001b
```

Зсуви та циклічні зсуви

У процесорах 8086 існує множина способів, за допомогою яких можна зсувати біти регістра чи змінної у пам’яті вліво чи вправо. Найпростішим з них є логічний зсув.

Інструкція SHL (зсув ліворуч, синонім – SAL) переміщує кожний біт операнда-приймача на один розряд вліво, у напрямку до самого значущого біту. Наприклад значення 10010110b (96h чи 150 у десятковій уяві) записане у AL, зсувається вліво за допомогою інструкції SHL AL,1. В результаті виходить значення 00101100b (24Ch чи 44 у десятковій уяві), яке записується назад у регістр AL. Флаг переносу встановлюється в значення 1.

Інструкція SAR (арифметичний зсув праворуч) аналогічна інструкції SHR, тільки при її виконанні найбільш значущий біт операнда зсувається праворуч у наступний біт, а потім копіюється назад. Наприклад значення 10010110b (96h чи –106 у десятковій уяві зі знаком), записане у регістрі AL зсувається праворуч за допомогою інструкції SAR AL,1. В результаті виходить значення 11001011b

(0CBh чи –53 у десятковій уяві зі знаком), яке записується назад у регістр AL. Флаг переносу встановлюється у значення 0.

Операція ROL має дію, зворотну дії операції ROR. Операнд також зсувається циклічно, але зсув виконується ліворуч. При цьому найбільш значущий біт зсувається у найменш значущий. Інструкції ROR та ROL корисно використовувати для пере упорядковування біт у байті чи у слові. Наприклад, в результаті виконання інструкцій:

```
mov si,49F1h
mov cl,4
ror si,cl
```

у регістр SI буде записано значення 149Fh: біти 3-0 перемістяться в біти 15-12, біти 7-4 – в біти 3-0 і т.д.

Інструкції RCR та RCL працюють трохи по-іншому. Інструкція RCR аналогічно інструкції зсуву праворуч, при цьому найбільш значущий біт зсувається з флага переносу. Наприклад значення 10010110b (96h чи 150 у десятковій уяві), записане у регістрі AL, циклічно зсувається праворуч через флаг переносу, початкове значення якого дорівнює 1, за допомогою інструкції RCR AL,1. В результаті отримуємо значення 1100101b (0CBh чи 203 у десятковій уяві), яке записується назад у регістр AL. Флаг переносу встановлюється у значення 0.

Інструкція RCL аналогічно, відповідно, лівому зсуву. При виконанні цієї інструкції найменш значущий біт зсувається з флага переносу. Інструкції RCR та RCL корисно використовувати для зсуву операнда, який складається з декількох слів. Наприклад, наступні інструкції виконують множення значення у DX:AX, розміром у подвійне слово на 4:

```
shl ax,1        ; біт 15 регістру AX зсувається у флаг переносу
rcl dx,1        ; флаг переносу зсувається у біт 0 регістру DX
shl ax,1        ; біт 15 регістру AX зсувається у флаг переносу
rcl dx,1        ; флаг переносу зсувається у біт 0 регістру DX.
```

Інструкції циклічного зсуву, аналогічно інструкціям зсуву, можуть зсовувати операнд на 1 біт чи на число біт, заданих регістром CL.

Безумовні переходи

Головною інструкцією переходу у наборі інструкцій процесора 8086 є інструкція JMP. Ця інструкція вказує процесору 8086, що у якості наступної за JMP інструкцією потрібно виконати інструкцію цільової мітки. Наприклад, після завершення виконання фрагменту програми:

```
mov ax,1
jmp AddTwoToAX
AddOneToAX
inc ax
jmp AxslSet
AddTwoToAX:
inc ax
AxlsSet:
```

регістр AX ,буде містити значення 3, а інструкція ADD та JMP, слідуючи за міткою AddOneToAX, ніколи виконані не будуть. Тут інструкція:

```
AddTwoToAX
```

вказує процесору 8086, що треба встановити покажчик інструкцій IP у значення зміщення мітки AddTwoToAx, тому наступною виконуємою інструкцією буде інструкція:

```
add ax,2
```

Іноді спільно з інструкцією JMP використовується операція SHORT. Для вказівки не цільові мітку інструкція JMP звичайно використовує 16-бітове зміщення. Операція SHORT вказує Турбо Асемблеру, що треба використовувати не 16-бітове, а 8-бітове зміщення (що дозволяє зекономити в інструкції JMP один байт).

Наприклад, останній фрагмент програми можна переписати так, що він стане на два байта коротшим:

```
mov ax,1
jmp SHORT AddTwoToAX
AddOneToAX:
inc ax
jmp SHORT AxsSet
AddTwoToAX:
inc ax
AxsSet:
```

Недолік використання операції SHORT (короткий) полягає у тому, що короткі переходи можуть здійснювати передачу керування на мітки, які на віддалі від інструкції JMP не далі, ніж на 128 байтів, тому у деяких випадках Турбо Асемблер може повідомляти вам, що мітка недосяжна за допомогою короткого переходу. До також операцію SHORT має сенс використовувати для посилок вперед, тому що для переходів назад (на попередні мітки) Турбо Асемблер автоматично використовує короткі переходи, якщо на мітку можна перейти за допомогою короткого переходу, та довгі у протилежному випадку.

Інструкцію JMP можна також використовувати для переходу в інший сегмент коду, завантажуючи в одній інструкції і регістр CS, і регістр IP. Наприклад, у програмі:

```
Cseg1 SEGMENT
ASSUME CS:Cseg1
FarTarget LABEL FAR
Cseg1 ENDS
Cseg2 SEGMENT
ASSUME CS:Cseg2
jmp FarTarget
Cseg2 ENDS
```

виконується перехід дальнього типу.

Якщо ви бажаєте, щоб мітка, примусово інтерпретувалася, як мітка дальнього тупу, можна використовувати операцію FAR PTR. Наприклад, у фрагменту програми:

```
jmp FAR PTR NearLabel
nop
NearLabel:
```

виконується перехід дальнього типу на мітку NearLabel, хоча ця мітка знаходиться у тім же сегменті коду, що й інструкція JMP.

Нарешті, ви можете виконати перехід по адресі, записаній у регістрі чи в змінній пам’яті. Наприклад:

```
mov ax,OFFSET TestLabel
jmp ax
.
.
TestLabel:
```

Тут виконується перехід на мітку TestLabel, так як і у наступному фрагменту:

```
JumpTarget DW TaestLabel

jmp [JumpTarget]
.
.
TestLabel:
```

Умовні переходи

Описані у попередньому розділі інструкції переходів – це тільки частина того, що вам потребується для написання корисних програм. В дійсності необхідна можливість писати такі програми, які можуть приймати рішення. Саме це можливо зробити за допомогою операцій умовних переходів.

Інструкція умовного переходу може здійснювати (чи ні) перехід на цільову (вказану в ній) мітку, в залежності від стану регістру флагів. Роздивимося наступний приклад:

```
mov ah,1                ; функція DOS вводу з клавіатури
int 21h                 ; отримати наступну
                        ; натиснуту клавішу
cmp al,'A'              ; чи була натиснута буква 'A'?
je AwasTyped            ; якщо так то обробити її
mov [TampByte],al       ; ні, зберегти символ
.
.
AwasTyped:
push ax                 ; зберегти символ у стекі
```

Спочатку у даній програмі за допомогою функції операційної системи DOS сприймається натиснута клавіша. Потім для зрівняння введеного символу з символом “A” використовується інструкція CMP. Ця інструкція аналогічна інструкції SUB, тільки її виконання ні на що не впливає, тому що призначення даної інструкції полягає у тому, щоб можна було порівняти два операнди, встановивши флаг також, як це робиться у інструкції SUB. Тому у попередньому прикладі флаг нуля встановлюється у значення 1 тільки у тому випадку, якщо регістр AL містить символ A.

Тепер ми підійшли до основного моменту. Інструкція JE представляє собою інструкцію умовного переходу, яка здійснює передачу керування тільки у тому випадку, якщо флаг нуля дорівнює 1. У протилежному випадку виконується інструкція, безпосередньо слідуюча за інструкцією JE (у даному випадку – інструкція MOV). Флаг нуля у даному прикладі буде встановлено тільки у випадку натиснення “A”, і тільки при цьому випадку процесор 8086 перейде до виконання інструкції з міткою AwasTyped, тобто інструкції PUSH.

Набір інструкцій процесора 8086 передбачає велику різноманітність інструкцій умовних переходів, що дозволяє вам здійснювати перехід майже по будь-якому флагу чи їх комбінації. Можна здійснювати умовний перехід по стану нуля, переносу, по знаку чи флагу переповнення і по комбінації флагів, вказуючи результати операцій зі знаками.

Перелік інструкцій умовних переходів приводиться у табл. 3.2.

Таблиця 3.2 – Інструкції умовних переходів

Назва	**Значення**	**Флажки, що перевіряються**
JNB/JNAE	Перейти, якщо менш/ перейти, якщо не більш чи дорівнює	CF=1
JAE/JNB	Перейти, якщо більш чи дорівнює/ перейти, якщо не менш	CF=0
JBE/JNA	Перейти, якщо менш чи дорівнює/перейти, якщо не більш	CF=1 чи ZF=1
JA/JNBE	Перейти, якщо більш/ перейти якщо не менш чи дорівнює	CF=0 та ZF=0
JE/JZ	Перейти, якщо дорівнює	ZF=1
JNE/JNZ	Перейти, якщо не дорівнює	ZF=0
JL/JNGE	Перейти, якщо менш ніж/ перейти, якщо не більш ніж чи дорівнює	SF=OF
JGE/JNL	Перейти, якщо більш ніж чи дорівнює/ перейти, якщо не більш ніж	SF=OF
JLE/JNLE	Перейти, якщо менш ніж чи дорівнює/ перейти, якщо не більш ніж	ZF=1 чи SF=OF
JG/JNLE	Перейти, якщо більш ніж/ перейти, якщо не більш ніж чи дорівнює	ZF=0 чи SF=OF
JP/JPPE	Перейти по парності	PF=1
JNP/JPO	Перейти по непарності	PF=0
JS	Перейти по знаку	SF=1
JNS	Перейти, якщо знак не встановлений	SF=0
JC	Перейти при наявності переносу	CF=1
JNC	Перейти при відсутності переносу	CF=0
JO	Перейти по переповненню	OF=1
JNO	Перейти при відсутності переповнення	OF=0

CF – флаг переносу, SF – флаг знаку, OF – флаг переповнення, ZF – флаг нуля, PF – флаг парності

Цикли

Для чого використовуються цикли? Вони служать для роботи з масивами, перевірки стану портів вводу-виводу до отримання певного стану, очистки блоків пам’яті, читання рядків з клавіатури та виводу їх на екран і т.д. Цикли – це головний засіб, який використовується для виконання дій які повторюються. Тому використовуються вони досить часто, настільки часто, що у наборі інструкцій процесора 8086 передбачено фактично декілька інструкцій циклів: LOOP, LOOPNE, LOOPE та JSXZ.

Давайте роздивимось спочатку інструкцію LOOP. Припустимо, ми хочемо вивести 17 символів текстової строки TestString. Це можна зробити наступним чином:

```
TestString DB ‘Це перевірка…’

mov cx,17
mov bx,OFFSET TestString
PrintStringLoop:
mov dl,[bx]                ; отримати наступний
                           ; символ
inc bx                     ; посилка на наступний
                           ; символ
mov ah,2                   ; функція DOS виводу на екран
int 21h                    ; визвати DOS для виводу
                           ; символу
dec cx                     ; зменшити лічильник довжини
                           ; рядка
jnz PrintStringLoop        ; обробити наступний символ, якщо
він є.
```

Але ж є кращій спосіб. Можливо ви пам’ятаєте, що раніше ми говорили о том, що регістр CX корисно буває використовувати для організації циклів. Інструкція:

loop PrintStringLoop

робить теж саме, що й інструкції:

dec cx
jnz PrintStringLoop

однак виконується вона скоріше та займає на один байт менше.

Кожний раз, коли вам потрібно організувати цикл, доки значення лічильника не буде дорівнювати 0, запишіть початкове значення лічильника у CX та використовуйте інструкцію LOOP.

Як же будуються цикли з більш складною умовою завершення, ніж зворотній відлік значення лічильника? Для таких випадків передбачені інструкції LOOP та LOOPNE.

Інструкція LOOPE працює також, як інструкція LOOP, тільки цикл при її виконанні буде завершуватися (тобто перестануть виконуватися переходи), якщо регістр CX прийме значення 0 чи флаг нуля буде встановлений у значення 1 (треба пам'ятати про те, що флаг нуля встановлюється у значення 1, якщо результат останньої арифметичної операції був нульовим або два операнди у останній операції зрівняння не збігались). Аналогічно, інструкція LOOPNE завершує виконання циклу, якщо регістр CX прийняв значення 0 або флаг нуля скинутий (має нульове значення).

Інструкція LOOPE звісна також, як інструкція LOOPZ, інструкція LOOPNE – як інструкція LOOPNZ, також як інструкції JE еквівалентна інструкція JZ (це інструкції – синоніми).

Є ще одна інструкція циклу. Це інструкція JCXZ. Інструкція JCXZ здійснює перехід тільки в тому випадку, якщо значення регістру CX дорівнює 0. Це дає зручний спосіб перевіряти регістр CX перед початком циклу. Наприклад, у наступному фрагменту програми, при зверненні до якого регістр BX вказує на байт, які потрібно обнулити, інструкція JCXZ використовується для пропуску тіла циклу у тому разі, якщо регістр CX має значення 0:

jcxz SkipLoop ; якщо CX має значення 0, ;то нічого робити не треба
ClearLoop:

```
mov BYTE PTR [si],0        ; встановити наступний байту
                           ; значення 0
inc si                     ; посилка на наступний байт, що
                           ; очищується
SkipLoop:
```

Чому бажано пропустити виконання циклу, якщо значення регістру CX дорівнює 0? Тому що у протилежному випадку значення CX буде зменшено до величини 0FFFFh та інструкція LOOP здійснить перехід на вказану мітку. Після цього цикл буде виконуватися 65535 разів. Ви ж бажали, щоб значення регістру CX, рівне 0, вказувало, що треба обнулити 0 байт, а не 65536. Інструкція JCXZ дозволяє вам у цьому випадку швидко виконати потрібну перевірку.

ПОРЯДОК ВИКОНАННЯ РОБОТИ

Отримати індивідуальне завдання у викладача (варіанти наведені у додатку А).

Розрахувати корені квадратного рівняння.

Загальні поняття.

Рівняння виду $ax^2+bx+c=0$, де x – невідоме, а коефіцієнти *a, b* и *c* – дані числа, називається квадратним рівнянням. В квадратному рівнянні $a \neq 0$, так як в іншому випадку воно було б лінійним рівнянням: $bx+c=0$. В той же час a може бути і позитивний і від'ємним. Якщо $a<0$, то помножив обидві частини на -1, отримаємо рівняння з позитивний коефіцієнтом при x^2. Коефіцієнт c називається свободний членом, ax^2 – старшим членом, bx – членом, що містить першу степінь невідомого.

Якщо $b\neq0$ і $c\neq0$, то рівняння $ax^2+bx+c=0$ називається повним квадратним рівнянням загального виду. Розділив всі члени його на a($a \neq 0$), отримаємо

$$x+b/a\ x+c/a=0 \qquad (3.1)$$

Допустимо $\frac{b}{a}=p$, $\frac{c}{a}=q$, маємо рівняння $x^2+px+q=0$, яке називається квадратним рівнянням приведеного виду або приведеним квадратним рівнянням. Якщо хоча б один із коефіцієнтів b або c дорівнює нулю,

то квадратне рівняння називається неповним. Неповні квадратні рівняння бувають трьох видів:

якщо b=0, c≠0, то $ax^2+c=0$;

якщо b ≠0, c =0, то $ax^2+bx=0$;

якщо b=0, c=0, то $ax^2=0$.

Квадратне рівняння виду $ax^2+bx+c=0$ можна розв'язати по формулі коренів приведеного рівняння, якщо дане рівняння попередньо розділить на *a* (a ≠0). Проте, можна користуватися і спеціальною формулою:

$$x1 = \frac{-b + \sqrt{D}}{2a} \tag{3.2}$$

$$x2 = \frac{-b - \sqrt{D}}{2a} \tag{3.3}$$

Вираз D, вхідні в цю формулу під радикалом, називають дискримінантом квадратного рівняння загального виду.

Якщо D >0, рівняння має два різних дійсних кореня.

Якщо D=0, то рівняння має два однакових кореня:

$$x_1=x_2=\frac{-b}{2a}. \tag{3.4}$$

Якщо D<0, то рівняння не має кореня (дійсних).

Програма повинна розраховувати значення коефіцієнтів нормативних значень із ASCII в двійковий формат, виконувати множення, округлення і зміщення.

ЗМІСТ ЗВІТУ

Постановка задачі.

Текст програми, вихідні дані.

Тести та результати відладки.

Блок схема програми.

Результати рішення на ЕОМ.

КОНТРОЛЬНІ ПИТАННЯ

Загальна характеристика арифметичних команд.

Команди складання та віднімання.

Відмінності в роботі команд ADD та ADC, SUB та SBB.

Нарощування та зменшення на 1 (INC та DEC).

Команди ділення та множення (зі знаком та без).

Відмінності при роботі усіх арифметичних команд з 8-бітними, 16-бітним та 32-бітними операндами.

Логічні операції AND, OR, XOR.

Команди SHL, SHR, RAR та їх функціонування.

Команди ROR, ROL, RCR, RCL та їх функціонування.

Різниця між командами SHL, SHR, SAR, SAL та ROR, ROL, RCR, RCL.

Команда JMP та типи переходів

Інструкція CMP та її застосування.

Команди умовного переходу.

Інструкції циклів: LOOP, LOOPNE, LOOPE, JCXZ.

Алгоритм ділення без залишку.

ЛАБОРАТОРНА РОБОТА № 4
Рядкові команди МП i8086

МЕТА РОБОТИ: Вивчити набір рядкових команд МП i8086.

ТЕОРІЯ

Рядкові інструкції

Тепер ми підійшли до розгляду найбільш потужних та незвичайних інструкцій процесора 8086 – інструкцій для роботи з рядками. Рядкові інструкції відрізняються від інших інструкцій процесора 8086. Вони можуть (у однієї інструкції) звертатися до пам'яті та збільшувати чи зменшувати регістр-показчик. Одна рядкова інструкція може звертатися до пам'яті 130000 разів. Як ясно з їх назви, рядкові інструкції особливо корисні при роботі з текстовими рядками. Їх можна також використовувати при роботі з масивами, буферами даних та будь-якими типами рядків байт чи слів. Рядковими інструкціями слід користуватися, там де тільки це можливо, оскільки вони, як правило, коротші та працюють скоріше, ніж еквівалентна їм комбінація звичайних інструкцій процесора 8086, таких, як MOV, INC та LOOP.

Ми роздивимось дві різні по функціональному призначенню групи рядкових інструкцій: рядкові інструкції для переміщення даних (LODS, STOS та MOVS) та строкові інструкції, які використовуються для пошуку та порівняння даних (SCAS та CMPS).

Строкові інструкції для переміщення даних

Строкові інструкції переміщення даних в цілому аналогічні інструкції MOV, але можуть виконувати більше функцій, ніж інструкції MOV та працюють швидше. Ми роздивимось спочатку інструкцію LODS. Зауважимо, що у всіх рядкових інструкціях флаг вказівки напрямку задає напрямок, в якому змінюються регістри-показчики.

Інструкція LODS

Інструкція LODS, яка завантажує файл чи слово з пам'яті у акумулятор (накопичувач), підрозділяється на дві інструкції – LODSB та LODSW. LODSB завантажує байт, який адресується за допомогою пари регістрів DS:SI, у регістр AL та зменшує чи збільшує регістр SI (у залежності від стану флага напрямку). Якщо флаг напрямку встановлений у 0 (встановлений за допомогою інструкції CLD), то регістр SI збільшується, а якщо флаг напрямку дорівнює 1 (встановлений за допомогою інструкції STD), то регістр SI зменшується. І це вірно не тільки для інструкцій LODSB, флаг напрямку керує напрямком, у якому змінюються усі регістри-показники рядкових інструкцій.

Наприклад, у наступному фрагменту програми:

```
cld
mov si,0
lodsb
```

інструкція LODSB завантажує регістр AL змістом байта зі зміщенням 0 у сегменті даних та збільшує значення регістру SI на 1. Це еквівалентно виконанню наступних інструкцій:

```
mov si,0
mov al,[si]
inc si
```

однак інструкція LODSB працює суттєво швидше (та займає на два байта мнеш), ніж інструкції:

```
mov al,[si]
inc si
```

Інструкція LODSW аналогічна інструкції LODSB. Вона зберігає у регістрі AX слово, яке адресується парою регістрів DS:SI, а значення регістра SI зменшується чи збільшується на 2, а не 1. Наприклад, інструкції:

```
std
mov si,0
lodsw
```

завантажують слово зі зміщенням 10 у сегменті даних у регістр RU, а потім значення SI зменшується на 2.

Інструкція STOS

Інструкція STOS – це доповнення інструкції LODS. Вона записує значення розміром у байт чи слово з акумулятора у чарунку пам’яті, на яку вказує пара регістрів ES:DI, а потім зменшує чи збільшує DI. Інструкція STOSB записує байт, який міститься у регістрі AL, у чарунку пам’яті по адресі ES:DI, а потім збільшує чи зменшує регістр DI, в залежності від флагу напрямку. Наприклад, інструкції:

```
std
mov di,0ffffh
mov al,55h
stosb
```

записують значення 55h у байт зі зміщенням 0FFFFh у сегменті, на який вказує регістр ES, а потім зменшує DI до значення 0FFFEh.

Інструкція STOSW працює аналогічно, записуючи значення розміром у слово, яке міститься у регістрі AX, по адресі ES:DI, а потім збільшує чи зменшує значення регістра DI на 2. Наприклад, інструкції:

```
cld
mov di,0ffeh
mov al,102h
stosw
```

записують значення 102h розміром у слово, записане у регістрі AX, по зміщенню 0FFEh у сегменті, на який вказує регістр ES, а потім значення регістра збільшується до 1000h.

Інструкції LODS та STOS можна чудово використовувати разом для копіювання буферів. Однак для переміщення байта чи слова з одного місця на інше є ще кращий спосіб. Це інструкція MOVS.

Інструкція MOVS

Інструкція MOVS аналогічна інструкціям LODS та STOS, якщо їх поєднати у одну інструкцію. Ця інструкція зчитує байт чи слово, записане по адресі DS:SI, а потім записує це значення по адресі, яка визначається парою регістрів ES:DI. Слово чи байт не передається при цьому через регістри, тому зміст регістра AX не змінюється. Інструкція MOVSB має мінімально можливу для інструкції довжину. Вона займає тільки один байт, а працює ще швидше, ніж комбінація інструкцій LODS та STOS.

```
mov cx,ARRAY_LENGTH_IN_WORDS
mov si,OFFSET SourceArray
mov ax,SEG SourceArray
mov ds,ax
mov di,OFFSET DestArray
mov ax,SEG DestArray
mov es,ax
cld
CopyLoop:
movsw
loop CopyLoop
```

Повторення рядкової інструкції

Хоча у останньому прикладі код виглядає досить ефективним, непогано було б позбавитися від інструкції LOOP та переміщувати весь масив за допомогою однієї інструкції. Інструкції процесора 8086 являють таку можливість. Це форма рядкових інструкцій з префіксом REP.

Префікс повторення REP – це не інструкція, а префікс інструкції. Префікс інструкції змінює роботу подальшої інструкції. Префікс REP робить наступне: він вказує, що наступну інструкцію треба повторювати доки зміст регістра CX не буде дорівнювати 0. (якщо регістр CX дорівнює 0 у початку виконання інструкції, то інструкція виконується 0 разів, іншими словами, ніяких дій не здійснюється.

Використовуючи префікс REP, можна замінити у останньому прикладі інструкції:

```
CopyLoop:
mosw
loop CopyLoop
```

на інструкцію:

```
rep movsb
```

Ця інструкція буде переміщувати блок з 65535 слів (0FFFh) з пам’яті, яка починається з адреси DS:SI у пам’ять, яка починається з адреси, яка визначається регістрами ES:DI.

Звичайно, для виконання інструкції 65535 разів потрібно значно більше часу, ніж для виконання інструкції оди раз, тому що для звертання до усій цієї пам’яті потребується час. Однак кожне повторення (з допомогою префікса) рядкової інструкції виконується скоріше, ніж виконання однієї рядкової інструкції. Це дозволяє отримати дуже швидкий спосіб читання з пам’яті, запису у пам’ять та копіювання.

Префікс REP можна використовувати не тільки з інструкцією MOVS, але також з інструкціями LODS та STOS (та інструкціями SCAS та CMPS – це ми розглянемо пізніше). Інструкцію STOS можна з успіхом повторювати для очистки чи заповнення блоків пам’яті, наприклад:

```
cld
mov ax,SEG WordArray
mov es,ax
mov di,OFFSET WordArray
sub ax,ax
mov cx,WORD_ARRAY_LENGTH
rep stosw
```

тут масив WordArray заповнюється нулями.

Префікс REP викликає повторення тільки рядкової інструкції. інструкція типу:

rep mov ax,[bx]

не має сенсу. У цьому випадку префікс REP ігнорується та виконується інструкція:

mov ax,[bx]

Інструкція SCAS

Інструкція SCAS використовується для перегляду пам'яті та пошуку збігів чи незбігів з конкретним значенням розміром у байт чи слово. Як і всі рядкові інструкції, інструкції SCAS має дві форми – SCASB та SCASW.

Інструкція SCASB порівнює зміст регістру AL з байтовим значенням по адресу ES:DI, встановлюючи при цьому флаги, які відображують результат порівняння (як при виконанні інструкції CMP). Як і при виконанні інструкції STOSB, при виконанні інструкції SCASB збільшується чи зменшується значення регістру DI.

Рядкові інструкції ніколи не встановлюють флаги таким чином, щоб вони відображали зміни значень регістрів SI, DI і/чи CX. Інструкції STOS та MOVS взагалі не змінюють ніяких флагів, а інструкції SCAS та CMPS змінюють флаги тільки у відповідності з результатом виконаного їм порівняння.

Префікс REPE (який також має назву префікс REPZ) також вказує процесору 8086, що інструкцію SCAS (чи CMPS) треба повторювати доколи регістр CX не буде дорівнювати нулю, чи допоки не станеться незбіг. Префікс REPE можна розглядати, як префікс, який означає "повторювати, допоки дорівнює". Аналогічно, префікс REPNE (REPNZ) вказує процесору 8086, що інструкцію SCAS (CMPS) треба повторювати, допоки CX не буде дорівнювати 0 чи допоки не станеться збіг. Префікс REPNE можна розглядати, як префікс "повторювати допоки не дорівнює".

Як і всі рядкові інструкції, інструкція SCAS збільшує регістр-показчик DI, якщо флаг напрямку дорівнює 0 (очищений за допомогою інструкції STD).

Інструкція SCASW – це форма інструкції SCASB для роботи зі словом. Вона зрівнює зміст регістру AX зі змістом пам'яті по адресі

ES:DI та збільшує чи зменшує значення регістру DI у кінці кожного виконання на 2, а ні на 1.

Інструкція CMPS

Інструкція CMPS дозволяє виконувати зрівняння двох байт чи слів. При одному виконанні інструкції CMPS зрівнюються дві чарунки пам'яті, а потім збільшуються регістри SI та DI. Інструкцію CMPS можна розглядати, як аналог інструкції MOVS, який замість копіювання однієї чарунки пам'яті у іншу зрівнює дві чарунки пам'яті.

Інструкція CMPSB порівнює байт по адресі DS:SI з байтом по адресі ES:DI, встановлюючи відповідним чином флаги та збільшуючи чи зменшуючи регістри SI та DI (у відповідності від флагу напрямку). Регістр AX при цьому не змінюється.

Як і всі рядкові інструкції, інструкція CMPS має дві форми (для роботи з байтами та для роботи зі словами), може збільшувати чи зменшувати регістри SI та DI та буде повторюватися при наявності префікса REP.

ПОРЯДОК ВИКОНАННЯ РОБОТИ

Помістити у сегменти (DATA1, DATA2) даних два тексти відповідно.

Очистити екран засобами рядкових команд.

Вивести на екран обидва тексти засобами рядкових команд.

У першому тексті підрахувати кількість слів, які починаються з букви "М".

У другому тексті підрахувати кількість слів, які починаються з букви "Н".

Вивести слова яких більше.

Знайти та вивести однакові слова.

Замінити усі точки (".") на знаки "!".

Вивести на екран обидва тексти.

ЗМІСТ ЗВІТУ

Постановка задачі.
Текст програми, вихідні дані.
Тести та результати відладки.
Блок схема програми.
Результати рішення на ЕОМ.

КОНТРОЛЬНІ ЗАПИТАННЯ

Пояснити роботу команд:

STOS, STOSB, STOSW;
LODS, LODSB, LODSW;
MOVS, MOVSB, MOVSW;
SCAS, SCAB, SCAW;
CMPS, CMPSB, CMPSW.

Описати роботу префіксів:

REP, REPZ, REPNZ, REPE, REPNE.

ЛАБОРАТОРНА РОБОТА № 5

Робота з клавіатурою та дисплеєм через BIOS

Мета роботи: Вивчити роботу клавіатури та відео-адаптера.

ТЕОРІЯ

Розширений Графічний Адаптер (Enhanced Graphics Adapter – EGA) фірми IBM являє собою графічний контролер, який забезпечує можливість роботи у різних відеорежимах сумісно з кольоровими чи монохромними моніторами з цифровими входами. Крім цього, адаптер забезпечує можливість роботи зі світовим пером. Адаптер може функціонувати у декількох графічних режимах (використовуються 4 бітові площини) та володіє можливістю завантаження у відеопам'ять шрифтів у алфавітно-цифрових режимах.

Адаптер містить у собі 64Кбайт пам'яті, оформленої у вигляді 4 бітових площин по 16Кбайт. Крім того, забезпечується можливість розширення пам'яті адаптера до 128 чи 256 Кбайт.

Відеобуфер (Display Buffer)

Розмір відео буфера (який має назву також відеопам'ять чи пам'ять адаптера) дорівнює 64 Кб. Відеобуфер доступний з боку процесора як на читання так і на запис та складається з 4 бітових площин по16 Кб. Існує можливість розширення відеобуфера до 128 Кб. На платі розширення встановлені слоти для підключення ще 128 Кб пам'яті, що дозволяє довести розмір відеобуферу до 256 Кб. При цьому у кожну бітову площину додається два додаткових банка пам'яті по 16 Кб. З метою сумісності з моделями відеоадаптерів, які існували раніше, адреси відеобуфера можуть змінюватися. Можливі 4 варіанта. Відеобуфер може бути встановлено довжиною 128 Кб та починатись з сегментної адреси A0000, довжиною 64 Кб та починатись з адреси A0000, довжиною 32 Кб та починатись з адреси B0000 чи довжиною 32 Кб з початком по адресі B8000.

Базова система вводу/виводу (BIOS)

Базова система вводу/виводу відеоадаптера знаходиться у пам'яті спеціального ПЗУ встановленого на платі адаптера. Відео BIOS об'єднується з системною базовою системою вводу/виводу. Тут розташовуються шрифти, які використовуються для генерації символів та керуючі програми відеоадаптера. Розмір ПЗУ – 16 Кб, початкова адреса – C0000.

У приведеній нижче табл. 5.1 подано список та характеристики доступних режимів при використанні стандартного монітору IBM.

Таблиця 5.1 – Список та характеристики доступних режимів при використанні стандартного монітору IBM

Режим	**Тип**	**Кольо-ри**	**Тексто-вий формат**	**Початок буферу**	**Розмір знако-місця**	**Макс-но сторінок**	**Розріше ння**
0	А/Ц	16	40x25	B800	8x8	8	320x200
1	А/Ц	16	40x25	B800	8x8	8	320x200
2	А/Ц	16	80x25	B800	8x8	8	640x200
3	А/Ц	16	80x25	B800	8x8	8	640x200
4	ГР	4	40x25	B800	8x8	1	320x200
5	ГР	4	40x25	B800	8x8	1	320x200
6	ГР	2	80x25	B800	8x8	1	640x200
D	ГР	16	40x25	A000	8x8	2/4/8	320x200
E	ГР	16	80x25	A000	8x8	1/2/4	640x200

Режими з 0 по 6 відповідають режимам кольорового графічного адаптера (CGA).

Режими 0, 2 та 5 ідентичні режимам 1, 3 та 4 відповідно по вихідному інтерфейсу.

Поля “Максимальна кількість сторінок для режимів D та E вказують кількість сторінок, які підтримуються при наявності графічної пам’яті розміром у 64 Кб, 128 Кб чи 256 Кб відповідно.

При використанні удосконаленого кольорового монітору зберігається сумісність з усіма раніше переліченими відеорежимами. Крім того, припускається використання додаткових відеорежимів, табл. 5.2, які підтримуються програмами базової системи вводу/виводу EGA.

Таблиця 5.2 – Список та характеристики додаткових відеорежимів, які підтримуються програмами базової системи вводу/виводу EGA

Режим	**Тип**	**Кольори**	**Тексто-вий формат**	**Початок буфера**	**Розмір знако-місця**	**Макс-но сторінок**	**Розрі-шення**
0*	А/Ц	16/64	40x25	B800	8x14	8	320x350
1*	А/Ц	16/64	40x25	B800	8x14	8	320x350
2*	А/Ц	16/64	80x25	B800	8x14	8	640x350
3*	А/Ц	16/64	80x25	B800	8x14	8	640x350
10*	ГР	4/16	80x25	A000	8x14	½	640x350

Режими 0, 1, 2 та 3 приведені також і у таблиці режимів для кольорового монітора IBM. Відео-BIOS забезпечує підвищену якість виводу при використанні удосконаленого кольорового монітору.

У полі “Кольори” визначається кількість одночасно поданих на екрані кольорів у тому чи іншому відеорежимі та кількість кольорів у палітрі. Для 10h відеорежиму у полі “Кольори” та полі кількість сторінок приведено два варіанта значень, які відповідають об’єму встановленої пам’яті – 64 Кб чи більш 64 Кб.

ЗАВДАННЯ

Написати програму яка виконує наступні дії:

При натисненні клавіші “T”:

Перейти у текстовий режим 80x25 16 кольорів;

Очистити екран, фон синього кольору;

Вивести своє ім’я у лівому верхньому куту красним кольором на зеленому фоні з атрибутом мигання символів;

Виведений напис плавно переміщувати у правий нижній кут.

При натисненні клавіші “G”:

Перейти у графічний режим 320x200 16 кольорів;

Очистити екран, фон чорного кольору;

Вивести малюнок PCX формату на екран;

При натисненні будь-якої клавіш крім “G” та “T” змінювати окремі кольори малюнка.

При повторному натисненні клавши “G” чи “T” виконувати відповідно пункти 1, 2.

ЗМІСТ ЗВІТУ

Постановка задачі.

Текст програми, вихідні дані.

Тести та результати відладки.

Блок схема програми.

Результати рішення на ЕОМ.

КОНТРОЛЬНІ ПИТАННЯ

Які типи відеоадаптерів ви знаєте, опишіть їх роботу?

Режими які підтримуються цими адаптерами.

Організація відеопам’яті у текстовому та графічному режимах адаптера EGA (Enhanced Graphics Adapter)/

Три режиму роботи клавіатури та їх різниця.

Функції BIOS для роботи з клавіатурою (16h, 15h, 9h, 8h).

Функції BIOS для роботи з відеоадаптером (10h, 15h).

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна

1. Абель П. Язык Ассемблера для IBM PC и программирования. /Пер. с англ. –М.:Высш.шк., 1992. - 447с.

2. Ларионов А.М., Майоров С.А., Новиков Г.И. Вычислительные комплексы, системы и сети: Учебник для вузов. – Л.: Энергоиздат, Ленингр. отд-ние, 1987.

3. Ивенса Д.И. Система параллейной обработки: Пер.с англ. – М.: Под ред. Мир, 1985.

4. Івенса д.і. Система параллейной обробки: Пер.с англ. – М.: Під ред. Мир, 1985.

5. Моотока Т, Томита О., Танака Х. и др. Компьтеры на СБИС: в 2-х кн.: Пер. с япон. – М.: Мир, 1988.

6. Фути К. Сутзуки Н. Языки программирования и схемотехника СБИС: Пер. с япон. – М.: Мир, 1988.

7. Гамкрелидзе С.А., Завьялов А.В. и др. Цифровая обработка информации на основе быстродействующих БИС. – М.: Энергоатомиздат, 1988.

8. Вайцер Б. Микроанализ производительности вычислительных систем. – М.: Радио и связь, 1983.

Додаткова

1. Головкин Б.А. Параллейные вычислительные системы. – М.: Наука, 1980.

2. Ферради Д. Оценка производительности вычислительных систем. – М.: Мир, 1981.

3. Хорошевский В.Г. Инженерный анализ функционирования вычислительных машин и систем. – М.: Радио и связь

Додаток А

Завдання на лабораторну роботу №3

№ варіанту	a	b	c
1	2	4	2
2	4	16	8
3	2	7	1
4	3	6	3
5	4	5	2
6	4	8	4
7	5	11	5
8	6	12	6
9	3	6	2
10	3	10	8
11	1	2	1
12	3	11	10
13	4	9	5
14	2	10	9
15	1	4	4
16	3	13	12
17	2	8	8
18	1	7	8
19	2	12	15
20	5	10	5
21	5	8	3
22	4	12	9
23	3	9	6
24	2	8	7
25	7	14	7
26	6	11	5
27	11	19	8
28	5	12	7
29	4	15	13