ЛАБОРАТОРНА РОБОТА №7

Макрозасоби.Файли.

|  |
| --- |
|  |

***Мета:*** Поняття про макрозасоби мови асемблера. Псевдооператори equ та =. Макрокоманди. Операціїз файлами.

План:

1. Поняття про сакрозасоби.
2. Макрокоманди.
3. Файли.
4. Читання і запис у файл.
5. Пошук файлів.

Короткі теоретичні відомості

Поняття про макрозасоби мови асемблера

Програми, що розглядалися до цих пір, були призначені для вирішення невеликих завдань, але навіть на прикладі цих маленьких за об'ємом програм стають очевидними деякі з нижчеперерахованих проблем:

1. погане розуміння початкового тексту програми, особливо після деякого часу після її написання;
2. обмеженість набору команд;
3. повторюваність деяких ідентичних або таких, що трохи відрізняються, ділянок програми;
4. необхідність включення в кожну програму ділянок коду, які вже були використані в інших програмах;
5. і т.д.

Якби ми писали програму на машинній мові, то дані проблеми були б принципово нерозв'язними. Але мова асемблера, будучи символічним аналогом машинної мови, надає для їх вирішення ряд засобів. Основною метою, яка при цьому переслідується, є підвищення зручності написання програм. У загальному випадку ця мета досягається по декількох напрямах за рахунок:

1. розширення набору директив;
2. введення деяких додаткових команд, що не мають аналогів в системі команд мікропроцесора.
3. введення складних типів даних.

Але це все глобальні напрями, по яких розвивається сам транслятор від версії до версії. Що ж робити програмістові для вирішення його локального завдання, для полегшення роботи в певній проблемній області? Для цього розробники компіляторів асемблера включають в мову і постійно удосконалюють апарат **макрозасобів***.* Цей апарат є дуже могутнім і важливим. У загальному випадку є сенс говорити про те, що транслятор асемблера складається з двох частин – безпосередньо транслятора, що формує об'єктний модуль, і **макроасемблера**. Якщо ви знайомі з мовами С або C++, то, звичайно, пам'ятаєте широко вживаний в них механізм препроцесорної обробки. Він є деяким аналогом механізму, закладеного в роботу макроасемблера. Основна його ідея – використання підстановок, які заміщають певним чином організовану символьну послідовність іншою символьною послідовністю. Створювана таким чином послідовність може бути як послідовністю, що описує дані, так і послідовністю програмних кодів. Головне тут те, що на вході макроасемблера може бути текст програми, вельми далекий по вигляду від програми на мові асемблера, а на виході обов'язково буде текст на чистому асемблері, що містить символічні аналоги команд системи машинних команд мікропроцесора. Таким чином, обробка програми на асемблері з використанням макрозасобів неявно здійснюється транслятором в дві фази (див. рис. 1). На першій фазі працює частина компілятора, звана макроасемблером, основні функції якого ми описали вище. На другій фазі трансляції бере участь безпосередньо асемблер, завданням якого є формування об'єктного коду, що містить текст початкової програми в машинному вигляді.

Псевдооператори equ і =

До простих макрозасобів мови асемблера можна віднести псевдооператори equ і «=» (рівно). Їх ми вже використовували раніше при написанні програм. Ці псевдооператори призначені для привласнення деякому виразу символічного імені або ідентифікатора. Згодом, коли в ході трансляції цей ідентифікатор зустрінеться в тілі програми, макроасемблер підставить замість нього відповідний вираз. В якості виразу можуть бути використані константи, імена міток, символічні імена і рядки в апострофах. Після привласнення цим конструкціям символічного імені його можна використовувати. скрізь, де потрібне розміщення даної конструкції. Синтаксис псевдооператора equ:

ім’я\_ідентифікатора equ рядок або числовий\_вираз

Синтаксис псевдооператора = :

ім’я\_ідентифікатора = числовий\_вираз

Не дивлячись на зовнішню і функціональну схожість, псевдооператори equ і = відрізняються наступним:

1. з синтаксичного опису видно, що за допомогою equ ідентифікатору можна ставити у відповідність як числові вирази, так і текстові рядки, а псевдооператор = може використовуватися тільки з числовими виразами;
2. ідентифікатори, визначені з допомогою =, можна перевизначати в початковому тексті програми, а визначені з використанням equ – не можна. Асемблер завжди намагається обчислити значення рядка, сприймаючи його як вираз. Для того, щоб рядок сприймався саме як текстовий, необхідно укласти його в кутові дужки: <рядок>. До речі сказати, кутові дужки є оператором асемблера, за допомогою якого транслятору повідомляється, що укладений в них рядок повинен трактуватися як текст, навіть якщо в нього входять службові слова асемблера або оператори. Хоча в режимі Ideal це не обов'язково, оскільки рядок для equ в нім завжди трактується як текстовий. Псевдооператори equ зручно використовувати для настроювання програми на конкретні умови виконання, заміни складних в позначенні об'єктів, багато разів використовуваних в програмі, простішими іменами і т.п. Наприклад:

masm

model small

stack 256

array\_size equ 15 ;розмірність массиву

akk equ ax ;переіменувати регістр

array\_elem equ array[bx][si] ;адресувати елемент масиву

.data

;описание массива из 15 байт:

arra db array­­\_size dup (0)

.code

mov akk, @data ;фактично mov ax, @data

mov ds, akk ;фактично mov ds, ax

- - -

mov al, array\_elem ;фактично mov al, array[bx][si]

Псевдооператор = зручно використовувати для визначення простих абсолютних (тобто не залежних від місця завантаження програми в пам'ять) математичних виразів. Головна умова те, щоб транслятор міг обчислити ці вирази під час трансляції. Наприклад:

.data

rda1 db 6 dup (0)

rda2 dw 0

len = 48

len = len+1 ;можна і так, через попереднє визначення

len = rda2– rda1

Як видно з прикладу, в правій частині псевдооператора = можна використовувати мітки і посилання на адреси – головне, щоб у результаті вийшов абсолютний вираз.

Компілятор TASM, починаючи з версії 3.00, містить директиви, значно розширяльні його можливості по роботі з текстовими макросами. Ці директиви аналогічні деяким функціям обробки рядків в мовах високого рівня. Під *рядками* тут розуміється текст, описаний за допомогою псевдооператора equ. Набір цих директив наступний:

1. директива злиття рядків catstr:

ідентифікатор catstr рядок1, рядок\_2...

Значенням цього макросу буде новий рядок, що складається із зчепленої зліва направо послідовності рядків **рядок\_1, рядок\_2** *.*

В якості зчіплюваних рядків можуть бути вказані імена раніше визначених макросів. Наприклад:

pre equ Привіт,

name equ < Юля>

privet catstr рге, nаmе ;privet= «Привіт, Юля»

1. директива виділення підрядка в рядку substr.

ідентифікатор substr рядок, номер\_позиції, розмір

Значенням даного макросу буде частина заданого **рядка***,* що починається з позиції з номером **номер\_позиції**і довжиною, вказаною в полі **розмір***.* Якщо потрібний тільки залишок рядка, починаючи з деякої позиції, то досить вказати тільки номер позиції без вказівки розміру. Наприклад:

;продовження попереднього фрагмента:

privet catstr рге, name ;privet = «Привіт, Юля»

name substr privet,7,3 ;nаmе = «Юля»

1. директива визначення входження одного рядка в інший instr:

ідентифікатор instr номер\_поч\_позиції, рядок\_1, рядок\_2

Після обробки даного макросу транслятором **ідентифікатору**буде привласнено числове значення, відповідне номеру (першої) позиції, з якою співпадають **рядок\_1 і рядок\_2***.* Якщо такого збігу немає, то **ідентифікатор**отримає значення 0;

1. директива визначення довжини рядка в текстовому макросі sizestr:

ідентифікатор sizestr рядок

В результаті обробки даного макросу значення *ідентифікатор* встановлюється рівним довжині рядка:

;як продовження попереднього фрагмента:

privet catstr рге, name ;privet = «Привіт, Юля»

len sizestr privet ;len=10

Ці директиви дуже зручно використовувати при розробці **макрокоманд***,* які є наступним макрозасобом, що надається компілятором асемблера.

Макрокоманди

Ідейно макрокоманда є подальшим розвитком механізму заміни тексту. За допомогою макрокоманд в текст програми можна вставляти послідовності рядків (які логічно можуть бути даними або командами) і навіть більш того – прив'язувати їх до контексту місця вставки. Представимо ситуацію, коли необхідно виконати деякі дії, що повторюються. Структурно в ній повинні простежуватися повторювані ділянки коду. Їх можна оформити у вигляді макрокоманд і використовувати ці фрагменти, що повторюються, в різних програмах. Подальше наше обговорення буде присвячено тому, як це зробити. Визначимося з термінологією. **Макрокоманда** є рядком, що містить деяке символічне ім'я — **ім'я макрокоманди***,* призначене для того, щоб бути заміщеною однією або декількома іншими рядками. Ім'я макрокоманди може супроводжуватися параметрами. Зазвичай програміст сам відчуває момент, коли йому потрібно використовувати макрокоманди в своїй програмі. Якщо така необхідність виникає і немає готового, раніше розробленого варіанту потрібної макрокоманди, то спочатку необхідно задати її **шаблон-опис***,* який називають *макровизначенням.* Синтаксис макровизначення наступний:

ім’я\_макрокоманди macro список\_формальних\_аргументів

тіло макровизначення

endm

Де повинні розташовуватися макровизначення? Є три варіанти:

1. на початку початкового тексту програми до сегменту коду і даних з тим, щоб не погіршувати читабельність програми. Цей варіант слід застосовувати у випадках, якщо визначувані вами макрокоманди актуальні тільки в межах однієї цієї програми;
2. у окремому файлі. Цей варіант підходить при роботі над декількома програмами однієї проблемної області. Щоб зробити доступними ці макровизначення в конкретній програмі, необхідно на початку початкового тексту цієї програми записати директиву, наприклад:

include ім’я\_файлу

masm

model small

include show.inc ;в це місце буде вставлено текст файлу

;show.inc

1. в макробібліотеці. Якщо у вас є універсальні макрокоманди, які використовуються практично у всіх ваших програмах, то їх доцільно записати в так звану макробібліотеку. Зробитиактуальними макрокоманди з цієї бібліотеки можна за допомогою все тієї ж директиви include. Недолік цього і попереднього способів в тому, що в початковий текст програми включаються абсолютно всі макровизначення. Для виправлення ситуації можна використовувати директиву purge, в якості операндів якої через кому перераховуються імена макрокоманд, які не повинні включатися в текст програми. Наприклад:

іnclude iomac.inc

purge outstr,exit

В даному випадку в початковий текст програми перед початком компіляції TASM замість рядка include iomac.inc вставить рядки з файлу iomac.inc. Але вставлений текст відрізнятиметься від оригіналу тим, що в нім будуть відсутні макровизначення outstr і exit.

А зараз проаналізуємо текст наступної програми, виявимо ділянки, що повторюються, і складемо для них макровизначення (лістинг 1). Програма перетворить двозначне шістнадцяткове число в символьному вигляді в двійкове представлення.

У прикладі описані макровизначення. Їх призначення приведене відразу після заголовка в тілі кожного макровизначення. Всі ці макровизначення можна використовувати і при написанні інших програм. У модернізованому початковому тексті програми сегмент коду став зовні більш читабельним і осмисленим. Відкомпілюйте лістинг 1 і отримайте файл лістингу. Після цього порівняйте початковий текст програми і той, яким він став після його обробки асемблером. Ви побачите, що текст програми змінився, – після рядків програми, в яких були макрокоманди, з'явилися фрагменти тексту. Вид цих фрагментів залежить від того, чи є у макрокоманди параметри. Приділите аналізу файлу лістингу трохи часу.

Функціонально макровизначення схожі на процедури. Схожість їх в тому, що і ті, та інші достатньо один раз десь описати, а потім викликати їх спеціальним чином. На цьому їх схожість закінчується, і починаються відмінності, які залежно від цільової установки можна розглядати і як переваги, і як недоліки:

На відміну від процедури, текст якої незмінний, макровизначення в процесі макрогенерації може мінятися відповідно до набору фактичних параметрів. При цьому корекції можуть піддаватися як операнди команд, так і самі команди. Процедури в цьому відношенні менш гнучкі;

Приклад 7.1.: Макрос для виведення рядка на екран.

.model small

.stack 100h

.data

msg1 db 'Hello, programmers!$',0Ah,0Dh,'$' ;Рядки для виведення

msg2 db 'Welcome to the world of assembly programming!$'

write\_string macro var ;Макрос виведення рядка

mov ah, 09h ;Функція DOS виведення рядка на екран

mov dx, offset var ;Завантажуємо адресу змінної в регістр dx

int 21h ;Виводимо на екран

endm

init macro ;Макрос ініціалізації даних

mov ax, @data

mov ds, ax

endm

exit macro ;Макрос виходу програми

mov ax, 4c00h

int 21h

endm

.code

start:

init ;Виклик макроса ініціалізації даних

write\_string msg1 ;Виклик макроса виведення рядка

write\_string msg2

exit ;Виклик макроса виходу з програми

end start

При кожному виклику макрокоманди її текст у вигляді макророзширення вставляється в програму. При виклику процедури мікропроцесор здійснює передачу управління на початок процедури, що знаходиться в деякій області пам'яті в одному екземплярі. Код в цьому випадку виходить компактнішим, хоча швидкодія дещо знижується за рахунок необхідності здійснення переходів.

Приклад 7.2.: Макрос для виведення числа на екран.

.model small

.stack 100h

.data

num1 dw 5

num2 dw 9

write\_num macro var

mov ah, 02h

mov dx, var

add dx, 30h

int 21h

endm

init macro

mov ax, @data

mov ds, ax

endm

exit macro

mov ax, 4c00h

int 21h

endm

.code

start:

init

write\_num num1

write\_num num2

exit

end start

Макровизначення обробляється компілятором особливим чином. Для того, щоб використовувати описане макровизначення в потрібному місці програми, воно повинне бути активізоване за допомогою **макрокоманди**вказівкою наступної синтаксичної конструкції:

ім’я\_макрокоманди список\_фактичних\_аргументів

Результатом застосування даної синтаксичної конструкції в початковому тексті програми буде її заміщення рядками з конструкції тіла макровизначення. Але це не проста заміна. Зазвичай макрокоманда містить деякий список аргументів список\_фактичних\_аргументів, якими коректується макровизначення. Місця в тілі макровизначення, які заміщатимуться фактичними аргументами з макрокоманди, позначаються за допомогою так званих формальних аргументів. Таким чином, в результаті застосування макрокоманди в програмі формальні аргументи в макровизначенні заміщаються відповідними фактичними аргументами; у цьому і полягає врахування контексту. Процес такого заміщення називається макрогенерацією, а результатом цього процесу є макророзширення.

Приклад 7.3.: Макрос пошуку символу в рядку.

.model small

.stack 100h

.data

strlen dw 30 ;Довжина рядка для пошуку

fnd db 'Symbol found!$'

nochar db 'Symbol not found.$'

string db 'Test string to search symbol.$' ;Рядок для пошуку

write\_string macro var ;Макрос виведення рядка

mov ah, 09h

mov dx, offset var

int 21h

endm

init macro ;Макрос ініціалізації даних

mov ax, @data

mov ds, ax

mov es, ax

endm

exit macro ;Макрос виходу з програми

mov ax, 4c00h

int 21h

endm

find\_sym macro var\_str, var\_len, var\_sym

;Макрос пошуку символу

;(рядок, довжина рядка, шуканий символ)

mov al, var\_sym ;Завантажуємо в al символ

cld ;Очищуємо флаг напрямку

mov di, offset var\_str ;Завантажуємо адресу рядка

mov cx, var\_len ;Завантажуємо довжину рядка

repne scasb ;Виконуємо порівняння символів рядка з шуканим

je found ;Якщо знайшли, переходимо на мітку

failed:

write\_string nochar ;виводимо повідомлення про результат

jmp ext

found:

write\_string fnd

ext:

endm

.code

start:

init

find\_sym string, strlen, 't' ;Виклик макросу пошуку символу

exit

end start

Розглянемо найкоротше макровизначення в лістингу 1 – clear\_r. Як зазначено вище, результати роботи макроасемблера можна дізнатися, проглянувши файл лістингу після трансляції. Покажемо декілька його фрагментів, які демонструють, як був описаний текст макровизначення clear\_r, як був здійснений виклик макрокоманди clear\_r з фактичним параметром ax і як виглядає результат роботи макрогенератора, що сформував команду асемблера xor ax.ax :

clear\_r macro rg ;очистка регістра rg

xor rg,rg

endm

- - -

clear\_r ax

000Е 33 С0 xor аx,ax

Таким чином, у результаті ми отримали те, що і було потрібно, – команду очищення заданого регістра, в даному випадку ax. У іншому місці програми ви можете видати ту ж макрокоманду, але вже з іншим ім'ям регістра. Якщо у вас є бажання, то ви можете провести експерименти з цією і іншими макрокомандами. Кожен фактичний аргумент є рядком символів, для формування якого застосовуються наступні правила: рядок може складатись:

1. з послідовності символів без пропусків, крапок, ком, крапок з комою;
2. з послідовності будь-яких символів, взятих в кутові дужки: <...>.

У цій послідовності можна вказувати як пропуски, так і крапки, коми, крапки з комами. Не забувайте про те, що кутові дужки < > – це теж оператор асемблера. Ми згадували про них при обговоренні директиви equ;

Для того, щоб вказати, що деякий символ усередині рядка, що представляє фактичний параметр, є власне символом, а не чимось іншим, наприклад, деяким роздільником або обмежуючою дужкою, застосовується спеціальний оператор «**!**». Цей оператор ставиться безпосередньо перед описаним вище символом, і його дія еквівалентна взяттю даного символу в кутові дужки (див. попередній пункт);

Якщо потрібне обчислення в рядку деякого константного виразу, то на початку цього виразу потрібно поставити знак *%:*

% константний\_вираз

Значення % константний\_вираз обчислюється і підставляється в текстовому вигляді відповідно до поточної системи числення.

Під поточною системою числення розуміється те, як інтерпретуються транслятором числа або рядки з фіксованим числовим значенням – як двійкові, десяткові або шістнадцяткові числа. За умовчанням транслятор трактує їх як десяткові. Асемблер має спеціальну директиву .radix, яка дає можливість змінити поточну систему числення. В якості операнда директива .radix має значення 2, 10 або 16, що означає вибір, відповідно, двійкової, десяткової або шістнадцяткової системи числення.

Приклад 7.4.: Макрос знаходження більшого числа.

.model small

.stack 100h

write\_num macro var ;Макрос виведення числа на екран

mov ah, 02h

mov dx, var

add dx, 30h

int 21h

endm

exit macro ;Макрос виходу з програми

mov ax, 4c00h

int 21h

endm

max macro n1,n2 ;Макрос порівняння 2 чисел

mov ax, n1 ;Завантажуємо числа в регістри

mov bx, n2

cmp ax, bx ;Порівнюємо значення

jle b ;Якщо перше число менше, ;переходимо на b

write\_num n1 ;інакше виводимо перше число

jmp ext

b:

write\_num n2 ;виводимо друге число

ext:

endm

.code

start:

max 4, 7

exit

end start

Приклад 7.5.: Макрос додавання двох чисел

.model small

.stack 100h

exit macro ;Макрос виходу з програми

mov ax, 4c00h

int 21h

endm

sum macro n1,n2 ;Макрос додавання 2 чисел

mov ax, n1 ;Завантажуємо числа в регістри

mov bx, n2

add ax, bx ;Додаємо числа, результат в ax

endm

.code

start:

sum 4, 7 ;Виклик макросу додавання

exit

end start

Файли

ДОС підтримує тільки одну файлову систему - FAT, і починаючи з версії 7.0 (Windows 95) її модифікацію VFAT с довгими іменами файлів. Набір функцій для роботи з файлами, запропонований у MS DOS 1.0, виявився незручним: кожен відкритий файл описувався 37-байтною структурою FCB (блок керування файлом), адреса якої була потрібна для всіх файлових операцій, а передача даних здійснювалася через структуру даних DTA (область передачі даних). Вже в MS DOS 2.0, разом з удосконаленням FAT (наприклад, появою вкладених директорій), з'явився набір UNIX-подібних функцій роботи з файлами, що використовують для опису файлу всього одне 16-бітне число, ідентифікатор чи файлу пристрою. Всі інші функції роботи з файлами використовують потім тільки це число. Перші п'ять ідентифікаторів ініціалізуються системою наступним чином:

0: STDIN - стандартний пристрій введення (звичайно клавіатура);

1: STDOUT - стандартний пристрій виведення (звичайно екран);

2: STDERR - пристрій виведення повідомлень про помилки (завжди екран);

3: AUX - послідовний порт (звичайно СОМ1);

4: PRN - паралельний порт (звичайно LPT1);

так що функції читання/запису (а також скидання буферів на диск) файлів можна застосовувати і до пристроїв.

Створення і відкриття файлів

Функція DOS 3Ch: Створити файл

Вхід: АН-3Ch

СХ = атрибут файлу

біт 7: файл можна відкривати різним процесам у Novell Netware

біт 6: не використовується

біт 5: архівний біт (1, якщо файл не зберігався)

біт 4: директорія (повинен бути 0 для функції 3Ch)

біт 3: мітка тому (ігнорується функцією 3Ch)

біт 2: системний файл

біт 1: схований файл

біт 0: файл тільки для читання

DS:DX адреса ASCIZ-рядка з повним ім'ям файлу (ASCIZ-рядок ASCII-символів, що закінчується нулем)

Вихід: CF = 0 і АХ - ідентифікатор файлу, якщо не було помилок

CF = 1 і АХ =03h, якщо шлях не знайдений

CF = 1 і АХ = 04h, якщо занадто багато відкритих файлів

CF = 1 і АХ = 05h, якщо доступ заборонено

Якщо файл вже існує, функція 3Ch все одно відкриває його, даючи йому нульову довжину. Щоб цього не відбулося, варто користатися функцією 5Bh.

Приклад 7.6.: Створення нового файлу.

.model small

.stack 100h

.data

fname db 'file.txt',0 ;Назва створюваного файлу

msg\_ok db 'ok$' ;Повідомлення про результат

msg\_err db 'error$'

.code

start:

mov ax,@data

mov ds,ax

mov ax, 3c00h ;Функція DOS створення файлу

lea dx, fname ;Завантажуємо адресу імені файлу

xor cx, cx ;Обнуляємо cx (атрибути файлу)

int 21h ;Створюємо файл

jc Error ;Якщо флаг CF встановлений, ;відбулась помилка

lea dx, msg\_ok

jmp write

Error:

lea dx, msg\_err

Write:

mov ah, 9 ;Виводимо повідомлення про ;результат

int 21h

mov ax, 4c00h

int 21h

end start

Функція DOS 3Dh: Відкрити існуючий файл

Вхід: AH=3Dh

AL = режим доступу

біт 0: відкрити для читання

біт 1: відкрити для запису

біти 2-3: зарезервовані (0)

біти 6-4: режим доступу для інших процесів:

000: режим сумісності (інші процеси також повинні відкривати цей файл у режимі сумісності)

001: всі операції заборонені

010: запис заборонений

011: читання заборонене

100: заборон немає

біт 7: файл не успадковується породжуваними процесами

DS:DX = адреса ASCIZ-рядка з повним ім'ям файлу

CL - маска атрибутів файлів

Вихід: CF = 0 і АХ = ідентифікатор файлу, якщо не було помилок

CF = 1 і АХ - код помилки (02h - файл не знайдений, 03h - шлях не знайдений, 04h - занадто багато відкритих файлів, 05h - доступ заборонений, ОСh - неправильний режим доступу)

Приклад 7.7.: Відкриття файлу.

.model small

.stack 100h

.data

fname db 'input.txt',0 ;Ім'я вхідного файлу

msg\_ok db 'ok$' ;повідомлення про результат

msg\_err db 'error$'

.code

start:

mov ax,@data

mov ds,ax

mov ax, 3d00h ;Функція DOS відкриття файлу

lea dx, fname ;Завантажуємо адресу імені файлу

int 21h ;Відкриваємо файл

jc Error ;Якщо флаг CF встановлений, відбулась ;помилка

lea dx, msg\_ok

jmp write

Error:

lea dx, msg\_err

Write:

mov ah, 9 ;Виводимо повідомлення про результат

int 21h

mov ax, 4c00h

int 21h

end start

Функція DOS 5Bh: Створити і відкрити новий файл

Вхід: АН = 5Bh

СХ = атрибут файлу

DS:DX - адреса ASCIZ-рядка з повним ім'ям файлу

Вихід: CF = 0 і АХ = ідентифікатор файлу, відкритого для читання/запису в режимі сумісності, якщо не було помилок

CF = 1 і АХ = код помилки (03h - шлях не знайдений, 04h - занадто багато відкритих файлів, 05h - доступ заборонений, 50h - файл вже існує)

Функція DOS 5Ah: Створити і відкрити тимчасовий файл

Вхід: АН=5Ah

СХ = атрибут файлу

DS:DX - адреса ASCIZ-рядка з шляхом, що закінчується символом \,і тринадцятьма нульовими байтами вкінці.

Вихід:CF = 0 і АХ = ідентифікатор файлу, відкритого для читання/запису в режимі сумісності, якщо не було помилки (у рядок за адресою DS:DX дописується ім'я файлу)

CF = 1 і АХ = код помилки (03h - шлях не знайдений, 04h - занадто багато відкритих файлів, 05h - доступ заборонений)

Функція 5Ah створює файл з унікальним ім'ям, що не є насправді тимчасовим.

В усіх випадках рядок з повним ім'ям файлу має вид типу

filespec db ‘c:\data\filename.ext’,0

причому, якщо диск чи шлях опущені, використовуються їхній поточні значення.

Для роботи з довгими іменами файлів у DOS 7.0 (Windows 95) і старше використовуються додаткові функції, що викликаються так само, як функція DOS 71 h.

Функція LFN 6Ch: Створити чи відкрити файл із довгим ім'ям

Вхід: AX=716Ch

ВХ = режим доступу Windows 95

біти 2-0: доступ

000 - тільки для читання

001 - тільки для запису

010 - для читання і запису

100 - тільки для читання, не змінювати час останнього звертання до файлу

біти 6-4: доступ для інших процесів (див. функцію 3Dh)

біт 7: файл не успадковується породжуваними процесами

біт 8: дані не буферуються

біт 9: не архівувати файл, якщо використовується архівування файлової системи (DoubleSpace)

біт 10: використовувати число в DI для запису наприкінці короткого імені файлу

біт 13: не викликати переривання 24h при критичних помилках

біт 14: скидати буфера на диск після кожного запису у файл

СХ = атрибут файлу :

DX = дія

біт 0: відкрити файл (помилка, якщо файл існує)

біт 1: замінити файл (помилка, якщо файл не існує)

біт 4: створити файл (помилка, якщо файл існує)

DS:SI = адреса ASCIZ-рядка з ім'ям файлу

DI = число, що буде записано наприкінці короткого варіанта імені файлу

Вихід:CF = 0

АХ = ідентифікатор файлу

СХ = 1, якщо файл відкритий

СХ = 2, якщо файл створений

СХ = 3, якщо файл замінений

CF = 1, якщо відбулася помилка

АХ = код помилки (7100h, якщо функція не підтримується)

Якщо функції відкриття файлів повертають помилку «занадто багато відкритих файлів» (АХ = 4), варто збільшити число припустимих ідентифікаторів за допомогою функції 67h.

Функція DOS 67h: Змінити максимальне число ідентифікаторів файлів

Вхід: АН=67h

ВХ - нове максимальне число ідентифікаторів (20 - 65535)

Вихід: CF = 0, якщо не було помилки

CF= 1 і АХ = код помилки, якщо відбулася помилка (наприклад: 04h, якщо задане число менше, ніж кількість уже відкритих файлів, чи 08h, якщо DOS не вистачає пам'яті для нової таблиці ідентифікаторів)

Читання і запис у файл

Функція DOS 3Fh: Читання з файлу чи пристрою

Вхід: АН = 3Fh

ВХ = ідентифікатор

СХ = число байтів

DS:DX = адреса буфера для прийому даних

Вихід: CF = 0 і АХ = число зчитаних байтів, якщо не було помилки

CF = 1 і AX = 05h, якщо доступ заборонений, 06h, якщо неправильний ідентифікатор

Якщо при читанні з файлу число фактично зчитаних байтів в АХ менше, ніж замовлене число в СХ, то був досягнутий кінець файлу. Кожна наступна операція читання, так само як і запису, починається не з початку файлу, а з того байта, на якому зупинилася попередня операція читання/запису. Якщо потрібно прочитати (чи записати) довільну ділянка файлу, використовують функцію 42h

Приклад 7.8.: Читання даних з файла.

.model small

.stack 100h

.data

fname db 'file.txt',0 ;Ім'я файлу для читання

buffer db 21 dup ('$') ;Резервування пам'яті під зчитані ;дані

bufsize dw 20 ;Розмір буфера під зчитані дані

msg\_ok db 'ok$' ;Повідомлення про результат

msg\_err db 'error$'

handle dw 1 ;Дескриптор відкритого файлу

error\_msg macro ;Макрос виведення повідомлення ;про помилку

mov ah,9

lea dx, msg\_err

int 21h

jmp close ;Переходимо до закриття файлу

endm

.code

start:

mov ax, @data

mov ds, ax

mov ax,3D00h ;Функція DOS відкриття файлу

lea dx,fname ;Завантажуємо адресу імені файлу

xor cx,cx ;Обнуляємо cx (атрибути файлу)

int 21h ;Відкриваємо файл

jnc read ;Якщо CF=0 (нема помилки), ;переходимо до читання

error\_msg ;інакше виводимо повідомлення і виходимо

jmp exit

read: mov [handle],ax ;Зберігаємо дескриптор файлу з ax в змінну handle

mov bx,ax ;Завантажуємо дескриптор в bx

mov ah,3Fh ;Функція DOS читання з файлу

lea dx,buffer ;Завантажуємо адресу буфера під зчитані дані

mov cx,[bufsize] ;Ініціалізуємо розмір зчитаних даних

int 21h ;Читаємо дані з файлу

jnc print ;Якщо CF=0 (нема помилки), переходимо до виведення

error\_msg ;інакше виводимо повідомлення і виходимо

print:

mov ah, 9 ;Виводимо зчитані дані на екран

lea dx, buffer

int 21h

close:

mov ah,3Eh ;Функція DOS закриття файлу

mov bx,[handle] ;Завантажуємо дескриптор файлу

int 21h ;Закриваємо файл

jnc exit

error\_msg

exit:

mov ax,4C00h

int 21h

end start

Функція DOS 42h: Перемістити покажчик читання/запису

Вхід: АН = 42h

ВХ = ідентифікатор

CX:DX - відстань, на яку треба перемістити покажчик (із знаком)

AL = переміщення:

0 - від початку файлу

1 - від поточної позиції

2 - від кінця файлу

Вихід: CF = 0 і CX:DX = нове значення покажчика (у байтах від початку файлу),

якщо не було помилки

CF = 1 і АХ = 06h, якщо неправильний ідентифікатор

Покажчик можна установити за реальними межами файлу:

1. у від’ємне число, тоді наступна операція читання/запису викликає помилку;
2. у додатне число, більше довжини файлу, тоді чергова операція запису збільшить розмір файлу.

Ця функція також часто використовується для визначення довжини файлу - досить викликати її з СХ = 0, DX = 0, AL= 2, і в CX:DX буде повернута довжина файлу в байтах.

Функція DOS 40h: Запис у файл чи пристрій

Вхід: АН=40h

ВХ = ідентифікатор

СХ = число байтів

DS:DX - адреса буфера з даними

Вихід: CF = 0 і АХ = число записаних байтів, якщо не було помилки

CF = 1 і АХ = 05h, якщо доступ заборонений; 06h, якщо неправильний ідентифікатор

Якщо при записі у файл вказати СХ = 0, він буде обрізаний за поточним значенням покажчика. Насправді відбувається запис у буфер DOS, дані з якого скидаються на диск під час закриття файлу, або коли якщо їхня кількість перевищує розмір сектора диска. Для негайного очищення буфера можна використовувати функцію 68h.

Приклад 7.9.: Запис даних в файл.

.model small

.stack 100h

.data

fname db 'file.txt',0 ;Назва файлу

buffer db 'Text to write into file' ;Дані для запису

bufsize dw 23 ;Розмір записуваних даних

msg\_ok db 'ok$' ;Повідомлення про ;результат

msg\_err db 'error$'

handle dw 1 ;Дескриптор файлу

error\_msg macro ;Макрос виведення ;повідомлення про помилку

mov ah,9

lea dx, msg\_err

int 21h

endm

.code

start:

mov ax, @data

mov ds, ax

mov ax,3C00h ;Функція DOS створення файлу

lea dx,fname ;Завантаження адреси імені файлу

xor cx,cx ;Обнуляємо cx (атрибути файлу)

int 21h ;Створення файлу

jnc write ;Якщо CF=0 (нема помилки), переходимо до запису

error\_msg ;інакше виводимо повідомлення і виходимо

jmp exit

write: mov [handle],ax ;Зберігаємо дескриптор з ax в змінну handle

mov bx,ax ;Завантажуємо дескриптор в bx

mov ah,40h ;Функція DOS запису в файл

lea dx,buffer ;Завантаження адреси записуваних даних

mov cx,[bufsize] ;Ініціадізація розміру записуваних даних

int 21h ;Запис даних в файл

jnc close\_file ;Якщо CF=0 (нема помилки), закриваємо файл

error\_msg ;інакше виводимо повідомлення і виходимо

close\_file:

mov ah,3Eh ;Функція DOS закриття файлу

mov bx,[handle] ;Завантажуємо дескриптор файлу

int 21h ;Закриваємо файл

jnc exit ;Якщо відбулась помилка, виводимо повідомлення

error\_msg

exit:

mov ax,4C00h

int 21h

end start

Функція DOS 68h: Скидання файлових буферів DOS на диск

Вхід: АН = 68h

ВХ = ідентифікатор

Вихід: CF = 0, якщо операція виконана

CF = 1, якщо відбулася помилка (AX = код помилки)

Для критичних ділянок програм краще використовувати більш ефективну функцію ODh.

Функція DOS ODh: Скидання усіх файлових буферів на диск

Вхід: АН = ODh

Вихід: Ніякого

Закриття і видалення файлу

Функція DOS 3Eh: Закрити файл

Вхід: АН = 3Eh

ВХ = ідентифікатор

Вихід: CF = 0, якщо не було помилки

CF = 1 і АХ = 6, якщо неправильний ідентифікатор

Якщо файл був відкритий для запису, усі файлові буфери скидаються на диск, встановлюється час модифікації файлу і записується його нова довжина.

Функція DOS 41h: Видалення файлу

Вхід: АН=41h

DS:DX = адреса ASCIZ-рядка з повним ім'ям файлу

Вихід: CF = 0, якщо файл було видалено

CF = 1 і АН = 02h, якщо файл не знайдений; 03h, якщо шлях не знайдений; 05h, якщо доступ заборонений

Видалити файл можна тільки після того, як він буде закритий, інакше DOS продовжить виконання запису в неіснуючий файл, що може привести до руйнування файлової системи. Функція 41h не дозволяє використовуватимаски(символи \* і ? в імені файлу) для видалення відразу декількох файлів, хоча цього можна домогтися, викликаючи її через недокументовану функцію 5D00h. Але, починаючи з DOS 7.0 (Windows 95), офіційна функція видалення файлу здатна працювати відразу з декількома файлами.

Функція LFN41h: Видалення файлів з довгим ім'ям

Вхід: AX=7141h

DS:DX - адреса ASCIZ-рядка з довгим ім'ям файлу

SI = 0000h: маски не дозволені й атрибути в СХ ігноруються

SI = 0001h: маски в імені файлу й атрибути в СХ дозволені:

CL = атрибути, що файли можуть мати

СН = атрибути, що файли повинні мати

Вихід: CF = 0, якщо файл чи файли видалені

CF = 1 і АХ - код помилки, якщо відбулася помилка. Код 7100h означає, що функція не підтримується

Пошук файлів

Знайти потрібний файл на диску набагато складніше, ніж просто відкрити його, - для цього вимагаються дві функції при роботі з короткими іменами (знайти перший файл і знайти наступний файл) і три - при роботі з довгими іменами в DOS 7.0 (знайти перший файл, знайти наступний файл, припинити пошук).

Функція DOS 4Eh: Знайти перший файл

Вхід: АН=4Eh

AL використовується при звертанні до функції APPEND

СХ = атрибути, що повинен мати файл (біти 0 (тільки для читання) і 5 (архівний біт) ігноруються. Якщо біт 3 (мітка тому) встановлений, всі інші біти ігноруються)

DS:DX - адреса ASCIZ-рядка з ім'ям файлу, що може включати шлях і маски для пошуку (символи \* і ?)

Вихід: CF = 0 і область DTA заповнюється даними, якщо файл знайдений

CF = 1 і АХ = 02h, якщо файл не знайдений; 03h, якщо шлях не знайдений; 12h, якщо неправильний режим доступу.

Виклик цієї функції заповнює даними область пам'яті DTA (область передачі даних), що починається із зміщення 0080h від початку блоку даних PSP (при запуску СОМ- і ЕХЕ-програм сегменти DS і ES містять сегментну адресу початку РSР), але її можна перевизначити за допомогою функції lAh.

Функція DOS 1Ah: Встановити область DTA

Вхід: АН - 1Ah

DS:DX - адреса початку DTA (128-байтний буфер)

Функції пошуку файлів заповнюють DTA таким чином:

+00h: байт - біти 0-6: ASCII-код букви диска біт 7: диск мережний

+01h: 11 байт-маска пошуку (без шляху)

+0Ch: байт - атрибути для пошуку

+0Dh: слово -порядковий номер файлу в директорії

+0Fh: слово - номер кластера початку зовнішньої директорії

+11h: 4 байти – зарезервовано

+15h: байт - атрибут знайденого файлу

+16h: слово - час створення файлу у форматі DOS:

біти 15-11: година (0-23)

біти 10-5: хвилина

біти 4-0: номер секунди, розділений на 2 (0-30)

+18h: слово - дата створення файлу у форматі DOS:

біти 15-9: рік, починаючи з 1980

біти 8-5: місяць

біти 4-0: день

+1Ah: 4 байти - розмір файлу

+lEh: 13 байт - ASCIZ-ім'я знайденого файлу з розширенням

Після того як DTA заповнена даними, для продовження пошуку варто викликати функцію 4Fh, поки не буде повернута помилка.

Функція DOS 4Fh: Знайти наступний файл

Вхід: АН = 4Fh

DTA - містить дані від попереднього виклику функції 4Eh чи 4Fh

Вихід: CF = 0 і DTA містить дані про наступний знайденому файлі, якщо не відбулася помилка

CF = 1 і АХ = код помилки, якщо відбулася помилка

У випадку з довгими іменами файлів (LFN) застосовується набір із трьох підфункцій функції DOS 71h, які можна використовувати, тільки якщо запущений IFSmgr (завжди запускається при звичайній установці Windows 95, але не запускається, наприклад, із завантажувальної дискети MS DOS 7.0).

Функція LFN 4Eh: Знайти перший файл із довгим ім'ям

Вхід: АХ=714Еh

CL = атрибути, що файл може мати (біти 0 і 5 ігноруються)

СН = атрибути, що файл повинен мати

SI = 0: використовувати Windows-формат дати/часу

SI = 1: використовувати DOS-формат дати/часу

DS:DX = адреса ASCIZ-рядка з маскою для пошуку (може включати \* і ?. Для сумісності маска \*.\* шукає всі файли в директорії)

ES:DI - адреса 318-байтного буфера для інформації про файл

Вихід: CF = 0

АХ - пошуковий ідентифікатор

СХ = Unicode-флаг:

біт 0: довге ім'я містить підкреслення замість незмінюваних Unicode-символів

біт 1: коротке ім'я містить підкреслення замість незмінюваних Unicode-символів

CF = 1, AX = код помилки, якщо була помилка (7100h - функція не підтримується)

Якщо файл, що підходить під маску і атрибути пошуку, знайдений, область даних за адресою ES:DI заповнюється таким чином:

+00h: 4 байти - атрибути файлу

біти 0-6: атрибути файлу DOS

біт 8: тимчасовий файл

+04h: 8 байт - час створення файлу

+0Ch: 8 байт - час останнього доступу до файлу

+14h: 8 байт - час останньої модифікації файлу

+1Сh: 4 байти - старше подвійне слово довжини файлу

+20h: 4 байти - молодше подвійне слово довжини файлу

+24h: 8 байт - зарезервований

+2Сh: 260 байт - ASCIZ-ім'я файлу довге

+130h:14 байт - ASCIZ-ім'я файлу коротке

Причому дати створення/доступу/модифікації записуються в одному з двох форматів, у відповідності зі значенням SI при виклику функції. Windows-формат - 64-бітне число 100-наносекундних інтервалів з 1 січня 1601 року; якщо використовується DOS-формат - у старше подвійне слово записується DOS-дата, а в молодше - DOS-час.

Функція LFN 4Fh: Знайти наступний файл

Вхід: AX=714Fh

ВХ = пошуковий ідентифікатор (від функції 4Еh)

SI - формат дати/часу

ES:DI = адреса буфера для інформації про файл

Вихід: CF = 0 і СХ - Unicode-флаг, якщо наступний файл знайдений

CF = 1, AX - код помилки, якщо була помилка (7100h - функція не підтримується)

Функція LFN A1h: Закінчити пошук файлу

Вхід: AX=71A1h

ВХ - пошуковий ідентифікатор

Вихід: CF = 0, якщо операція виконана

CF = 1 і АХ= код помилки, якщо була помилка (7100h - функція не підтримується)

Керування файловою системою

Починаючи з MS DOS 2.0 файлова система організована у вигляді директорій. Пошук файлів виконується тільки в межах поточної директорії, а створення і видалення файлів неприйнятні до директорій, хоча на найнижчому рівні директорія - той же файл, в атрибуті якого біт 4 встановлений у 1 і який містить список імен вкладених файлів, їхніх атрибутів і фізичних адрес на диску.

Функція DOS 39h: Створити директорію

Вхід: АН = 39h

DS:DX - адреса ASCIZ-рядка зі шляхом, у якому всі директорії, крім останньої, існують.

Вихід: CF = 0, якщо директорія створена

CF = 1 і АХ = 3, якщо шлях не знайдений; 5, якщо доступ заборонений

Функція LFN 39h: Створити директорію з довгим ім'ям

Вхід: AX=7139h

DS:DX - адреса ASCIZ-рядка з шляхом

Вихід: CF = 0, якщо директорія створена

CF = 1 і АХ = код помилки (7100h, якщо функція не підтримується)

Функція DOS 3Ah: Видалити директорію

Вхід: АН = 3Ah

DS:DX = адреса ASCIZ-рядка зі шляхом, де остання директорія буде вилучена (якщо вона порожня)

Вихід: CF = 0, якщо директорія вилучена

CF = 1 і АХ = 3, якщо шлях не знайдений; 5, якщо доступ заборонений; 10h, якщо директорія, що видаляється - поточна

Функція LFN 3Ah: Видалити директорію з довгим ім'ям

Вхід: AX=713Ah

DS:DX - адреса рядка з шляхом

Вихід: CF = 0, якщо директорія вилучена, інакше CF = 1 і АХ = код помилки

Функція DOS 47h: Визначити поточну директорію

Вхід: AH=47h

DL = номер диска (00h - поточний, 01h = А и т. д.)

DS:SI - 64-байтний буфер для поточного шляху (ASCIZ-рядок без імені диска, першого й останнього символу \)

Вихід: CF = 0 і АХ = 0100h, якщо операція виконана

CF = 1 і АХ = 0Fh, якщо зазначено неіснуючий диск

Функція LFN47h: Визначити поточну директорію з довгим ім'ям

Вхід: АХ - 7147h

DL - номер диска

DS:SI - буфер для шляху (ASCIZ-рядок без імені диска, першого й останнього символу \. Необов'язково містить лише довгі імена - повертається той шлях, що використовувався при останній зміні поточної директорії.)

Вихід: CF = 0, якщо директорія визначена, інакше CF = 1 і АХ = код помилки

*Функція DOS 3Bh:* Змінити директорію

Вхід: АН = 3Bh

DS:DX = адреса 64-байтного ASCIZ-буфера зі шляхом, що стане поточною директорією

Вихід: CF = 0, якщо директорія змінена, інакше CF =1 і АХ = 3 (шлях не знайдений)

*Функція LFN3B:* Змінити директорію з довгим ім'ям

Вхід: AX=713Bh

DS:DX = адреса ASCIZ - буфера зі шляхом

Вихід: CF = 0, якщо директорія змінена, інакше CF = 1 і АХ = код помилки

Перед роботою з будь-якими функціями LFN необхідно один раз викликати підфункцію 0A0h, щоб визначити розміри буферів для імен файлів і шляхів.

*Функція LFN OAOh:* Одержати інформацію про розділ файлової системи VFAT

Вхід: AX=71A0h

DS:DX = адреса ASCIZ-рядка з ім'ям розділу (наприклад: db "C:\",0)

ES:DI = адреса буфера для імені файлової системи (FAT, NTFS, CDFS)

СХ = розмір буфера в ES:DI (звичайно 32 байта)

Вихід: СХ = 0, АХ = 0000h або 0200h

ВХ = флаги файлової системи:

біт 0: функції пошуку враховують регістр символів

біт 1: регістр символів зберігається для імен директорій

біт 2: використовуються символи Unicode

біт 14: підтримуються функції LFN

біт 15: включено-стискування розділу (DoubleSpace)

СХ = максимальна довжина імені файлу (звичайно 255)

DX = максимальна довжина шляху (звичайно 260) у Windows 95 повертає 0000h для CD-ROM

CF = 1 і АХ = код помилки, якщо відбулася помилка (7100h, якщо функція не підтримується)

Крім того, при виклику будь-якої функції LFN варто встановлювати CF у 1 для сумісності з ранніми версіями DOS. Старі версії DOS не змінювали CF, так що в результаті, якщо функція не підтримується, CF залишиться рівним 1.

Завдання до лабораторної роботи №7

**Перший рівень**

ЗАВДАННЯ

1.1.Наберіть програму 4.1.

1.2.Завантажте програму в відладчик. Виконайте **програму 7.1** по крокам.

**Програма 7.1**

.model small

.stack 100h

.data

fname db 'file.txt',0 ;Назва файлу

buffer db 'Text to write into file' ;Дані для запису

bufsize dw 23 ;Розмір записуваних даних

msg\_ok db 'ok$' ;Повідомлення про ;результат

msg\_err db 'error$'

handle dw 1 ;Дескриптор файлу

error\_msg macro ;Макрос виведення ;повідомлення про помилку

mov ah,9

lea dx, msg\_err

int 21h

endm

.code

start:

mov ax, @data

mov ds, ax

mov ax,3C00h ;Функція DOS створення файлу

lea dx,fname ;Завантаження адреси імені файлу

xor cx,cx ;Обнуляємо cx (атрибути файлу)

int 21h ;Створення файлу

jnc write ;Якщо CF=0 (нема помилки), переходимо до запису

error\_msg ;інакше виводимо повідомлення і виходимо

jmp exit

write: mov [handle],ax

mov bx,ax ;Завантажуємо дескриптор в bx

mov ah,40h ;Функція DOS запису в файл

lea dx,buffer ;Завантаження адреси записуваних даних

mov cx,[bufsize] ;Ініціадізація розміру записуваних даних

int 21h ;Запис даних в файл

jnc close\_file ;Якщо CF=0 (нема помилки), закриваємо файл

error\_msg ;інакше виводимо повідомлення і виходимо

close\_file:

mov ah,3Eh ;Функція DOS закриття файлу

mov bx,[handle] ;Завантажуємо дескриптор файлу

int 21h ;Закриваємо файл

jnc exit ;Якщо відбулась помилка, виводимо повідомлення

error\_msg

exit:

mov ax,4C00h

int 21h

end start

**Другий рівень**

Виконати пункт 1 із завдання згідно варіанту з лабораторної роботи №6. Результати записати у файл.

**Третій рівень**

Виконати пункти 1 та 2 із завдання згідно варіанту з лабораторної роботи №6. Результати пункту 1 записати у файл rezult1.txt. Результат пункту 2 записати у файл result 2.txt

Контрольні запитання

1. Дати визначення макрозасобам.
2. З яких частин складається транслятор асемблера?
3. Які псевдо оператори можна віднести до простих макрозасобів мови асемблер?
4. Яким являється синтаксис псевдо оператора?
5. Що собою представляє термінологія макрокоманди?
6. Відомості про файл. Створення та відкриття файлів.
7. Як відбувається читання і запис у файл?
8. Як відбувається закриття і видалення файлу?

Література:

[6, c. 95; 8, c.156; 11, c.183]

ЛАБОРАТОРНА РОБОТА №8

Зв'язок асемблера і мов високого рівня. На прикладі Visual C++.

|  |
| --- |
|  |

***Мета:*** навчитися створювати проекти, що складаються з модулів, написаних на асемблері та на С++.

План:

1. Форми комбінування програм.
2. Зв'язок С – Асемблер.
3. Трасування (відлагодження) проектів у Microsoft Visual C++.

Короткі теоретичні відомості

Форми комбінування програм

Існують наступні форми комбінування програм на мовах високого рівня з асемблером:

- використання операторів типу inline і асемблерних вставок. Ця форма сильно залежить від синтаксису мови високого рівня і конкретного компілятора. Вона припускає, що асемблерні коди у виді команд асемблера, чи безпосередньо в машинних командах вставляються в текст програми мови високого рівня. Компілятор мови розпізнає їх як команди асемблера (машинні коди) і без змін включає у формований ним об'єктний код. Ця форма зручна, якщо треба вставити невеликий фрагмент.

- використання зовнішніх процедур і функцій. Це більш універсальна форма комбінування.

- використання оператора inline і асемблерних ввставок сильно залежить від синтаксису конкретної мови.

Згадаємо синтаксис директиви PROC:

назва\_процедури PROC [[модифікатор\_мови] мова] [відстань]

Один з операндів - мова. Він служить для того, щоб компілятор міг правильно організувати інтерфейс (зв'язок даних) між процедурою на асемблері і програмою на мові високого рівня. Необхідність такої указівки виникає внаслідок того, що способи передачі аргументів при виклику процедур різні для різних мов високого рівня. TASM і MASM підтримують декілька значень операнду «мова». У таблиці 2.1 для деяких з них приведені характерні риси передачі аргументів і угоди про те, яка процедура очищає стек – та, яка викликає чи та, яку викликають. Під напрямком передачі аргументів розуміється порядок, у якому аргументи включаються в стек, у порівнянні з порядком їх проходження у виклику процедури. Так, наприклад, для мови PASCAL характерний прямий порядок включення аргументів у стек: першим у стек записується перший переданий аргумент з оператора виклику процедури, другим – другий аргумент і т.д. На вершині стека після запису всіх переданих аргументів виявляється останній аргумент. Для мови С, навпаки, характерний зворотний порядок передачі аргументів. Відповідно до нього в стек спочатку включається останній аргумент з оператора виклику процедури (функції), потім передостанній і т.д. В остаточному підсумку на вершині стека виявляється перший аргумент. Що ж стосується очищення стека, то зрозуміло, що повинні бути визначені домовленості про це. У мові PASCAL цю операцію завжди робить процедура, яку викликають, у мові С – та, що викликає.

Таблиця 2.1 – Передача аргументів у мовах високого рівня

|  |  |  |  |
| --- | --- | --- | --- |
| Операнд “мова” | Мова | Напрямок передачі | Яка процедура очищає стек |
| NOLANGUAGE | Асемблер | зліва на право | яка викликається |
| BASIC | Basic | зліва на право | яка викликається |
| PROLOG | Prolog | з права наліво | яка викликає |
| FORTRAN | Fortran | зліва на право | яка викликається |
| C | C | з права наліво | яка викликає |
| C++ (CPP) | C++ | з права наліво | яка викликає |
| PASCAL | Pascal | зліва на право | яка викликається |
| STDCALL | – | з права наліво | яка викликається |
| SYSCALL | C++ | з права наліво | яка викликає |

Зв'язок С – асемблер.

Спочатку завжди необхідно зберігати – і перед виходом із процедури відновлювати – вміст регістрів EBP, ESP, CS, DS, SS, ES. Це робиться перед викликом процедури. Інші регістри потрібно зберігати згідно необхідності, але, гарним тоном є збереження і наступне відновлення всіх регістрів, що піддаються зміні усередині процедури.

Передача аргументів у процедуру на асемблері з програми на С здійснюється через стек. Порядок їх розміщення в стеці зворотний запису прототипу процедури мовою С, тобто з права наліво. Процедури на асемблері одержують доступ до аргументів, переданим у стеці, за допомогою регістру EBP. Попередньо у початок процедури асемблера необхідно вставити код прологу:

push ebp

mov ebp, esp

Після цього доступ до аргументів у стеці здійснюється за допомогою зсуву щодо вмісту регістру EBP, наприклад:

mov eax, [ebp+08h] ; переписати значення першого аргументу зі стека в eax

mov ebx, [ebp+0ch] ; переписати значення другого аргументу зі стека в ebx

При організації зв'язку С–асемблер також можна використовувати директиву ARG. Це позбавить від необхідності підраховувати зсув у стеці для доступу до аргументів і дозволить звертатися до них за іменами.

arg chr:byte, x:word, y:dword, kol:word

push ebp

mov ebp,esp

...

mov al, [chr] ; переписати значення chr зі стека в al

mov ebx, [y] ; значення y – у регістр ebx

Приклад 8.1.: Зв'язок асемблера і C++.

#include <iostream>

#include <cstdint>

using namespace std;

int main()

{

//Шаблонні типи з cstdint дозволяють бути певними

//щодо розміру змінних в бітах

uint8\_t x = 185;

uint16\_t rezult;

//ASM вставка може бути як однорядковою

\_\_asm mov al, x;

//ASM так і багаторядковою

\_\_asm

{

//Можна використовувати оголошені змінні як для вхідних даних

mul x;

//Так і для вихідних даних

mov rezult, ax;

}

cout << (int)x << " \* " << (int)x << " = " << rezult << endl;

cin.get();

return 0;

}

Що стосується передачі аргументів С–асемблер, то тут усе досить прозоро. У приведеному вище прикладі використовується директива MODEL з операндами FLAT і STDCALL і директива PROC з указівкою мови С. Це повідомляє компілятору самому сформувати коди прологу й епілогу, а також організувати звертання до змінних у стеці за їх іменами. Але при використанні конкретних програмних засобів організація такого зв'язку виглядає набагато складнішою. Не в останню чергу це зв'язано з тим, що компілятори С/С++ розробляють безліч фірм. Ця обставина – основна причина складності зв'язку С–асемблер, тому що кожна фірма реалізує її по-своєму (хоча суть залишається практично незмінної). Так, наприклад, для компіляторів фірми Microsoft DOS платформи, компіляторів фірми Borland і інших фірм при оголошенні процедури її назві передує символ підкреслення “\_asmproc proc c”, а для компіляторів фірми Microsoft (MASM32) Windows платформи символ підкреслення в оголошенні процедури відсутній. Тому перед організацією зв'язку С-асемблер необхідно звернутися до опису компілятора конкретної фірми для уточнення таких особливостей.

Для того щоб повернути результат у програму на С із процедури на асемблері існують стандартні умови.

Перед поверненням керування в програму на С в програмі на асемблері необхідно помістити результат чи сформувати покажчик у зазначених регістрах:

Приклад 8.2.: Перевірка скільки тіків процесора займає бульбашкове та швидке сортування.

#include <iostream>

#include <cstdint>

#include <random>

#include <vector>

#include <algorithm>

using namespace std;

unsigned \_\_int64 inline GetRDTSC()

{

\_\_asm

{

;Онулюємо eax

XOR eax, eax

; запобігання "out-of-order" виконання інструкцій

CPUID

; Отримуємо кількість тіків в edx:eax

RDTSC

}

}

template <typename RandomAccessIterator>

void bubble\_sort(RandomAccessIterator begin, RandomAccessIterator end)

{

for(auto i = begin; i != end - 1; ++i)

for (auto j = i + 1; j != end; ++j)

{

if (\*i > \*j)

iter\_swap(i, j);

} }

int main() {

const int max = 10000;

const int min = 0;

const int count = 1000;

//Гереруємо масив випадкових чисел

default\_random\_engine generator;

uniform\_int\_distribution<int> distribution(min, max);

vector<int> randomArray;

for(int i = 0; i < count; i++)

{

randomArray.push\_back(distribution(generator));

}

vector<int> arrayCopy = randomArray;

unsigned \_\_int64 start;

unsigned \_\_int64 finish;

//Перевірка скільки тіків процесора займає бульбашкове сортування

start = GetRDTSC();

bubble\_sort(randomArray.begin(), randomArray.end());

finish = GetRDTSC();

cout<<"Bubble sort tooks"<<finish-start<<"processor ticks"<<endl;

//Перевірка скільки тіків процесора займає швидке сортування

start = GetRDTSC();

sort(arrayCopy.begin(), arrayCopy.end());

finish = GetRDTSC();

cout <<"Quick sort tooks" <<finish-start <<"processor ticks"<<endl;

cin.get();

return 0;

}

Таблиця 2.2 Повернення аргументів із процедури на асемблері в програму на С/С++

|  |  |  |
| --- | --- | --- |
| Тип значення, що повертається, (С++) | Результат DOS | Результат Win32 |
| unsigned char | al | al |
| char | al | al |
| enum | ax | ax |
| unsigned short | ax | ax |
| short | ax | ax |
| unsigned int | ax | eax |
| int | ax | eax |
| double long | tbyte ptr ST(0) стек спiвпроцесора | qword ptr ST(0) стек спiвпроцесора |
| unsigned long | eax | eax |
| long | eax | eax |
| покажчик near | dx:ax | eax |
| покажчик far | dx:ax | eax |

Приклад 8.3. У програмі реалізований зв'язок С–асемблер для виведення повідомлення.

#include <iostream>

#include <stdio.h>

using namespace std;

int power2( int num, int power )

{

\_\_asm

{

mov eax, num ; Get first argument

mov ecx, power ; Get second argument

shl eax, cl ; EAX = EAX \* ( 2 to the power of CL )

}

}

int main()

{

char hi[6] = "Hello";

char earth[] = "World";

char text[] ="%s %s";

\_\_asm

{

lea eax,earth // eax = address of earth

push eax // put eax at the top of the stack

lea eax,hi // eax = address of hi

push eax

lea eax,text

push eax

call DWORD ptr printf

// or the indierct call

// mov eax, printf

// call eax

pop ebx // clean up the stack

pop ebx

pop ebx

}

int a=power2(2,3);

cout<<a<<endl;

cin.get();

}

Приклад 8.4.: У програмі реалізований зв'язок С–асемблер для обчислення суми елементів масиву

#include "stdafx.h"

extern "C" int sum\_asm(int massiv[], int count);

int main(int argc, char\* argv[])

{

int mas[5]={1,2,3,4,5};

int len = 5;

int sum=0;

sum=sum\_asm(mas,len);

printf("Sum = %d\n",sum);

return 0;

}

; lab6.asm

.386

model flat, stdcall

option casemap :none ; case sensitive

public sum\_asm

.code

sum\_asm proc C adr\_mas:dword,len\_mas:dword

xor eax,eax

mov ecx,len\_mas ;довжину масиву - у ecx

mov esi,adr\_mas ;адресу масиву - у esi

cycl:

add eax,[esi] ; додавання акумулятора з елементом масиву

add esi,4 ; адресувати наступний елемент масиву

loop cycl

ret ; повернення

Приклад 8.5:Функції які викликаються з ASM файлу

ASM – файл:

.386 ;Дозвіл на виконання непривілейованих інструкцій процесора 80386

.model flat

;Визначення функції, присутньої в іншому модулі

extern \_LogResults : proc

;Функція ASM\_Sum доступна в інших модулях

public ASM\_Sum

.code

\_ASM\_Square proc

;Переміщення в eax вершини стеку в якому знаходиться

;аргумент переданий з с++

mov eax, [esp+4]

imul eax

;Збереження в стеку отриманого результату

;для його подальшого використання в с++ функції

push eax

;Викликаємо функцію LogResults в сpp файлі

push 0; ResultCode = Success

push 0; OprationCode = ShowSquare

call \_LogResults

;Очищення стеку від 8 байтів (2х верхніх параметрів)

add esp, 8

;Оскільки всередині фукції \_LogResults значення

;регістрів змінювалось, відновлюємо результат множення зі стеку

pop eax

ret

\_ASM\_Square endp

;Визначення параметрів функції

ASM\_Sum proc C array\_adr:dword, arr\_len:dword

xor eax, eax ; онулення регістру eax

mov ecx, arr\_len ;Завантаження довжини масиву зі arr\_len у ecx

mov esi, array\_adr ;Завантаження адреси масиву зі array\_adr у esi

cycl:

add eax, [esi] ; додавання акумулятора з елементом масиву

add esi,4 ; адресувати наступний елемент масиву

loop cycl

push eax

push 0 ;ResultCode = Success

push 1 ;OprationCode = ShowSum

call \_LogResults ;Виклик функції з срр файлу

add esp, 8 ; Очищення стеку від 8 байтів (2х верхніх параметрів)

pop eax

ret ; повернення з функцiї, результат - у eax

ASM\_Sum endp

End

СРР – файл:

#include <iostream>

using namespace std;

enum OprationCode { ShowSquare, ShowSum };

enum ResultCode { Success, Failure };

//Попереднє оголошення функцій описаних в стилі С

//Функція в стилі С не містить в своєму описі місця

//для адреси об'єкта, що викликається.

//Це лише заголовки. Функції визначені в ASM файлі

extern "C" int ASM\_Sum(int \_array[], int length);

extern "C" long ASM\_Square(long);

int main(int argc, char\* argv[])

{

int arrayLen = 5;

int mas[5] = {1, 2, 3, 4, 5};

//Виклик функції ASM\_Sum з ASM файлу

long sum = ASM\_Sum(mas, arrayLen);

//Виведення результату функції з ASM файлу

cout << "Returned sum is: " << sum << endl;

int number;

long square;

cout << "Enter number to square" << endl;

cin >> number;

//Виклик фунцкіїї ASM\_Square з ASM файлу

square = ASM\_Square(number);

//Виведення результату функції з ASM файлу

cout << "Returned square is: " << square << endl;

cin.get();

cin.get();

return 0;

}

extern "C"

void LogResults(OprationCode operation, ResultCode result, long value)

{

//Дана функція оголошена як функція в стилі С

//І викликається лише з ASM файлу для логування результатів

switch (operation)

{

case ShowSquare:

switch (Success)

{

case ShowSquare:

cout << "Logged square is: " << value << endl;

break;

default:

cout << "Error calculating square" << endl;

break;

}

break;

case ShowSum:

switch (Success)

{

case ShowSquare:

cout << "Logged sum is: " << value << endl;

break;

default:

cout << "Error calculating sum" << endl;

break;

}

break;

} }

**Завдання до лабораторної роботи №8**

**Перший рівень**

Здійснити компіляцію та запуск прикладу 1. Отримати результати.

**Другий рівень**

У програмі реалізувати зв'язок С–асемблер для здійснення операції згідно варіанту у наступний спосіб: обчислення/операції оформляються у вигляді асемблерної функції з параметрами і дана функція викликається в С++ коді.

Реалізація здійснюється як проект VisualStudio, що містить два файли lab\_8.cpp та lab\_8.asm.

**Третій рівень**

У програмі реалізувати зв'язок С–асемблер для здійснення операції згідно варіанту у два способи:

1. операції оформляються у вигляді асемблерної функції з параметрами і дана функція викликається в С++ коді.
2. операції оформляються у вигляді С++-функції з параметрами і дана функція викликається в асемблерному коді.

Реалізація здійснюється як проект VisualStudio, що містить два файли lab\_8.cpp та lab\_8.asm. Варіанти беруться з **лабораторної роботи №5**.