ЛАБОРАТОРНА РОБОТА №4

Вивчення команд передачі керування

|  |
| --- |
|  |

***Мета:*** вивчити команди, що змінюють порядок виконання програми на асемблері. Операції-примітиви, префікси повторення, операції пересилання, порівняння і сканування ланцюжків, завантаження елемента ланцюжка в акумулятор, перенос елемента з акумулятора в ланцюжок, введення і висновок елемента ланцюжка

План:

1. Безумовні переходи.
2. Умовні переходи.
3. Цикли.
4. Ланцюжкові команди.

Короткі теоретичні відомості

Процесор 8086/8088 має 4 групи інструкцій передачі керування: інструкції безумовної та умовної передачі керування, ітерації і переривання. Тільки остання група впливає на стан прапорів процесора. Однак, виконання багатьох інструкцій передачі керування залежить від значень прапорів.

Безумовні переходи

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

Приклад 4.1.:Порівняння двох чисел

.model small

.stack 100h

.data

n1 dw 10 ;Оголошуємо числа для порівняння

n2 dw 13

msg\_l db 'lesser$'

msg\_e db 'equal$'

msg\_b db 'greater$'

.code

start:

mov ax, @data

mov ds, ax

mov ax, n1 ;Завантажуємо обидва числа в регістри

mov bx, n2

cmp ax, bx ;Порівнюємо числа в регістрах

jl t1 ;Переходимо до мітки відповідно до результату

je t2

jg t3

t1:

lea dx, msg\_l ;Завантажуємо повідомлення в регістр

jmp print ;Переходимо до мітки виведення результату

t2:

lea dx, msg\_e

jmp print

t3:

lea dx, msg\_b

print: ;Виводимо повідомлення про результат

mov ah,09h

int 21h

mov ax,4c00h

int 21h

end start

Іноді разом з інструкцією JMP використовується операція SHORT. Для вказівки на цільову мітку інструкція JMP звичайно використовує 16-бітове зміщення. Операція SHORT вказує Турбо Асемблеру, що потрібно використати не 16-бітове, а 8-бітове зміщення (що дозволяє заощадити в інструкції JMP один байт). Наприклад, останній фрагмент програми можна переписати так, що він стане на два байти коротше:

mov ax,1

jmp SHORT AddTwoToAX

AddOneToAx:

inc ax

jmp SHORT AXIsSet

AddTwoToAX:

inc ax

AXIsSet:

Недолік використання операції SHORT (короткий) полягає в тому, що короткі переходи можуть здійснювати передачу керування на мітки, що розміщені від інструкції JMP не далі, чим на 128 байтів, тому в деяких випадках Турбо Асемблер може повідомляти вам, що мітка недосяжна за допомогою короткого переходу.

Інструкцію JMP можна використати для переходу в інший сегмент коду, завантажуючи в одній інструкції регістри і CS, і IP.

Якщо ви хочете, щоб мітка вимушено інтерпретувалася, як мітка дальнього типу, можна використати операцію FAR PTR. Наприклад, у фрагменті програми:

.

*jmp FAR PTR NearLabel*

*nop*

*NearLabel:*

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

Нарешті, ви можете виконати перехід за адресою, записаною в регістрі або в змінній пам'яті. Наприклад:

mov ax,OFFSET TestLabel

jmp ax

.

.

TestLabel:

mov ah,1 ; функція DOS вводу с клавіатури

int 21h ; отримати наступну натиснуту клавішу

cmp al,'A' ; була натиснута буква "A"?

je AWasTyped ; так, опрацювати її

mov [TampByte], al ; ні, зберегти символ

.

.

AWasTyped:

push ax ; зберегти символ у стек

Умовні переходи

Інструкції умовного переходу організовують передачу керування задає інструкції, що при виконанні є специфічними для кожної інструкції цього типу умов. Ці умови визначаються поточними станами прапорів процесора. Кожна із цих 18 інструкцій (їм відповідають 30 мнемонічних кодів операцій) перевіряє певну комбінацію прапорів. Якщо умова виконується, відбувається перехід по зазначеній адресі; в іншому випадку керування передається на наступну інструкцію програми. Всі умовні переходи є короткими, тобто цільова інструкція повинна бути не далі, ніж на -128 або 127 байтів від першого байта наступної інструкції (JMP 00h забезпечує перехід до наступної інструкції). Оскільки перехід здійснюється додаванням до вмісту регістра IP відносного зміщення цілі, всі інструкції умовного переходу є самовідносними і тому можуть використовуватися при написанні адресно незалежних програм. Інструкції умовної передачі керування та перевіряючі при їхньому виконанні умови наведені в таблиці 6.1.

Таблиця 6.1 – Інструкції умовних переходів

|  |  |  |
| --- | --- | --- |
| Назва | Значення | Перевіряючі прапорці |
| JB/JNAE | Перейти, якщо менше / перейти, якщо не більше або дорівнює | CF = 1 |
| JAE/JNB | Перейти, якщо більше або дорівнює / перейти, якщо не менше | CF = 0 |
| JBE/JNA | Перейти, якщо менше або дорівнює / перейти, якщо не більше | CF = 1 або ZF = 1 |
| JA/JNBE | Перейти, якщо більше / перейти, якщо не менше або 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 менше ніж / перейти, якщо не більше чим або дорівнює | SF = OF |
| JGE/JNL | Перейти, якщо більше чим або дорівнює / перейти, якщо не менше ніж | SF = OF |
| JLE/JNLE | Перейти, якщо менше ніж або дорівнює / перейти, якщо не більше, ніж | ZF = 1 або SF = OF |
| JG/JNLE | Перейти, якщо більше чим / перейти, якщо не менше чим або дорівнює | ZF = 0 або SF = OF |
| JP/JPE | Перейти по парності | PF = 1 |
| JNP/JPO | Перейти по непарності | PF = 0 |
| JS | Перейти за знаком | 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 - прапор переносу, SF - прапор знаку, OF - прапор переповнення, ZF - прапор нуля, PF - прапор парності.

Приклад 4.2.: Перевірка парності числа

.model small

.stack 100h

.data

num db 4 ;Число для перевірки

msg\_z db 'Number is zero$'

msg\_p db 'Number is even$'

msg\_o db 'Number is odd$'

.code

start:

mov ax, @data

mov ds, ax

mov al, num ;Завантажуємо число в регістр

cmp al, 0 ;та порівнюємо з 0

jz t1 ;Переходимо на t1, якщо число = 0

jp t2 ;Переходимо на t2, якщо число парне

jmp t3

t1:

lea dx, msg\_z

jmp print ;Перехід на мітку виведення

t2:

lea dx, msg\_p

jmp print

t3:

lea dx, msg\_o

print:

mov ah,09h ;Виводимо повідомлення результату

int 21h

mov ax,4c00h

int 21h

end start ;Кінець програми

Приклад 4.3.: Перевірка, чи є символ числом.

.model small

.stack 100h

.data

num db '0' ;Символ для перевірки

msg\_not db 'Symbol is not number$'

msg\_yes db 'Symbol is number$'

.code

start:

mov ax, @data

mov ds, ax

mov al, num ;Завантажуємо число в регістр

cmp al, '0' ;Порівнюємо з нижньою межею

jl no ;Якщо код символу менший,

;переходимо на no

cmp al, '9' ;Порівнюємо з верхньою межею

jg no ;Якщо код символу більший,

;переходимо на no

lea dx, msg\_yes ;Виводимо повідомлення про

;результат

jmp print

no:

lea dx, msg\_not

print:

mov ah,09h

int 21h

mov ax,4c00h

int 21h

end start

Цикли

Одним з видом конструкцій у програмі, які можна побудувати за допомогою умовних переходів, є цикли. Цикл - це просто блок коду, що завершується умовним переходом, завдяки чому блок даних може виконуватися повторно до досягнення умови завершення.

Цикли - це основний засіб, що використовується для виконання повторюваних дій. Тому використовуються вони досить часто, настільки часто, що в наборі інструкцій процессора 8086 передбачено фактично кілька інструкцій циклів: LOOP, LOOPNE, LOOPE та JCXZ.

Давайте розглянемо спочатку інструкцію LOOP.

Приклад 4.4.: Виведення ASCII-символів.

.model small

.stack 100h

codeseg

start:

mov ah, 02h ; Функція DOS виводу символу в регістрі DX

mov dl, 0 ; Задаємо початковий символ

mov cx, 255 ; Ініціалізуємо лічильник циклу

loopsym: ; Мітка початку циклу

int 21h ;Виводимо символ на екран

inc dx ;Збільшуємо код символу

dec cx ;Зменшуємо лічильник

jnz loopsym ;Повторюємо цикл поки лічильник

;не дорівнює 0

mov ax, 4c00h

int 21h

end start

Є, однак, кращий спосіб.

loop PrintStringLoop робить те ж, що й інструкції:

dec cx

jnz PrintStringLoop

Приклад 4.5.: Сума чисел від 1 до 10.

.model small

.stack 100h

codeseg

start:

xor ax, ax ;Онулюємо регістр ax (для

;збереження суми)

mov cx, 10 ;Ініціалізуємо лічильник

;значенням 10

sum: ;Мітка початку циклу

add ax, cx ;Додаємо до суми число

dec cx ;Зменшуємо лічильник

jnz sum ;Повторюємо цикл поки лічильник

;не дорівнює 0

mov ax, 4c00h

int 21h

end start

Однак виконується вона швидше та займає на один байт менше. Щораз, коли вам потрібно організувати цикл, поки значення лічильника не стане рівним 0, запишіть початкове значення лічильника в регістр CX і використайте інструкцію LOOP.

Приклад 4.6.: Вивести числа від 0 до 9.

.model small

.stack 100h

.code

start:

mov ah, 02h ;Функція DOS виведення символу в

;dl на екран

mov dl, '0' ;Ініціалізуємо початковий символ

mov cx, 10 ;Ініціалізуємо лічильник циклу

num: ;Мітка початку циклу

int 21h ;Виведення символу на екран

inc dx ;Збільшення коду символу

loop num ;Повторення циклу num поки cx не

;дорівнює 0

mov ax, 4c00h

int 21h

end start

Інструкція LOOPE працює також, як інструкція LOOP, тільки цикл при її виконанні буде завершуватися (тобто перестануть виконуватися переходи), якщо регістр CX прийме значення 0 або прапор нуля буде встановлений у значення 1 (потрібно пам'ятати про те, що прапор нуля встановлюється в значення 1, якщо результат останньої арифметичної операції був нульовим або два операнда в останній операції порівняння не збігалися). Аналогічно, інструкція LOOPNE завершує виконання циклу, якщо регістр CX прийняв значення 0 або прапор нуля скинутий (має нульове значення). Інструкція LOOPE відома 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 регістра CX дорівнює 0. Це дає зручний спосіб перевіряти регістр CX перед початком циклу.

Щодо інструкцій циклів, можна зробити декілька цікавих зауважень. По-перше, потрібно пам'ятати про те, що інструкції циклів, як і інструкції умовних переходів, можуть виконати перехід тільки на мітку, що розміщена від інструкції циклу не більше, ніж на 128 байт у той чи інший бік. Цикли, що перевищують 128 байт, вимагають використання умовних переходів за допомогою безумовних переходів (цей метод описаний у попередньому розділі "Умовні переходи"). По-друге, важливо розуміти, що жодна з інструкцій циклів не впливає на стан прапорів.

Ланцюжкові команди

Ці команди також називають командами обробки рядків символів. Відмінність у тому, що під рядком символів тут розуміється послідовність байт, а ланцюжок — це більш загальна назва для випадків, коли елементи послідовності мають розмір більше байта — слово або подвійне слово. Таким чином, ланцюжкові команди дозволяють проводити дії над блоками пам'яті, що представляють собою послідовності елементів наступного розміру:

8 біт — байт; 16 біт — слово; 32 біта — подвійне слово.

Вміст цих блоків для мікропроцесора не має ніякого значення. Це можуть бути символи, числа і все що завгодно. Головне, щоб розмірність елементів збігалася з однією з перерахованих і ці елементи знаходилися в сусідніх комірках пам'яті.

Операції-примітиви, префікси повторення

Усього в системі команд мікропроцесора є сім операцій-примітивів обробки ланцюжків. Кожна з них реалізується в мікропроцесорі трьома командами, у свою чергу, кожна з цих команд працює з відповідним розміром елемента — байтом, словом або подвійним словом.

Особливість усіх ланцюжкових команд у тому, що вони, крім обробки поточного елемента ланцюжка, здійснюють ще й автоматичне просування до наступного елемента даного ланцюжка. У таблиці 7.1 перераховані операції-примітиви і команди, за допомогою яких вони реалізуються. Логічно до цих команд потрібно віднести і так звані префікси повторення. Ці префікси повторення вказуються перед потрібною ланцюжковою командою в полі мітки. Ланцюжкова команда без префікса виконується один раз. Розміщення префікса перед ланцюжковою командою змушує її виконуватися в циклі. Відмінності приведених префіксів у тому, на якій підставі приймається рішення про циклічне виконання ланцюжкової команди: за станом регістра ecx/cx або за прапором нуля zf:

1. префікс повторення rep (REPeat). Цей префікс використовується з командами, що реалізують операції-примітиви пересилання і збереження елементів ланцюжків — відповідно, movs і stos. Префікс rep змушує дані команди виконуватися, поки вміст у ecx/cx не стане рівним 0. При цьому ланцюжкова команда, перед якою стоїть префікс, автоматично зменшує вміст ecx/cx на одиницю. Та ж команда, але без префікса, цього не робить;
2. префікси повторення repe або repz (REPeat while Equal or Zero). Ці префікси є абсолютними синонімами. Вони змушують ланцюжкову команду виконуватися доти, поки вміст ecx/cx не дорівнює нулю або прапор zf дорівнює 1. Як тільки одне з цих умов порушується, керування передається наступній команді програми. Завдяки можливості аналізу прапора zf, найбільше ефективно ці префікси можна використовувати з командами cmps і scas для пошуку елементів ланцюжків, що відрізняються;
3. префікси повторення repne або repnz (REPeat while Not Equal or Zero). Ці префікси також є абсолютними синонімами. Їхня дія на ланцюжкову команду трохи відрізняється від дій префіксів repe/repz. Префікси repne/repnz змушують ланцюжкову команду циклічно виконуватися доти, поки вміст ecx/cx не дорівнює нулю або прапор zf дорівнює нулеві. При невиконанні однієї з цих умов робота команди припиняється.

Дані префікси також можна використовувати з командами cmps і scas, але для пошуку співпадаючих елементів ланцюжків.

Наступний важливий момент, пов'язаний з ланцюжковими командами, полягає в особливостях формування фізичної адреси операндів адреса\_джерела та адреса\_приймача.

Приклад 4.7.: Пошук символу в рядку.

.MODEL small

.STACK 256

.data

fnd db 0ah,0dh,'Symbol found!$'

nochar db 0ah,0dh,'Symbol not found.$'

string db 'Test string to search symbol.$',0ah,0dh,'$' ;Рядок для пошуку

strlen dw 30 ;Довжина рядка

.code

main:

mov ax,@data

mov ds,ax

mov es,ax

mov ah,09h ;Функція DOS виведення рядка

lea dx,string ;Завантажуємо адресу рядка

int 21h ;Виводимо рядок на екран

mov al,'i' ;Завантажуємо в al шуканий символ

cld ;Очищуємо прапор напрямку для

;порівняння в порядку зростання адрес

lea di,string ;Завантажуємо адресу рядка в di

mov cx,strlen ;Ініціалізуємо лічильник циклу

repne scasb ;Повторюємо сканування символів рядка

je found ;поки не знайдемо шуканий (в al)

;Переходимо на мітку found якщо знайшли символ

failed:

mov ah,09h ;Виведення повідомлення про невдачу

lea dx,nochar

int 21h

jmp exit

found:

mov ah,09h ;Виведення повідомлення про успіх

lea dx,fnd

int 21h

exit:

mov ax,4c00h

int 21h

end main

Ланцюжок-джерело, що адресується операндом адреса\_джерела, може знаходитися в поточному сегменті даних, обумовленому регістром ds.

Ланцюжок-приймач, що адресується операндом адреса\_приймача, повинен бути в додатковому сегменті даних, що адресується сегментним регістром es. Важливо відзначити, що допускається заміна (за допомогою префікса заміни сегмента) тільки регістра ds, регістр es підмінювати не можна. Другі частини адрес – зсув у ланцюжках — також знаходяться в строго визначених місцях. Для ланцюжка-джерела це регістр esi/si (Source Index register — індексний регістр джерела). Для ланцюжка-приймача це регістр edi/di (Destination Index register - індексний регістр приймача). Таким чином, повні фізичні адреси для операндів ланцюжкових команд наступні:

адреса\_джерела — пари ds:esi/si; адреса\_приймача — пари es:edi/di.

**Таблиця 4.1 – Операції-примітиви і команди обробки ланцюжків**

|  |  |
| --- | --- |
| Операції-  примітиви | Команди |
| пересилання ланцюжка | [movs](#movs) адреса\_приймача,адреса\_джерела [movsb](../../Text/Command.htm#movs)  [movsw](../../Text/Command.htm#movs)  [movsd](../../Text/Command.htm#movs) |
| порівняння ланцюжків | [cmps](#cmps) адреса\_приймача,адреса\_джерела  [cmpsb](../../Text/Command.htm#cmps)  [cmpsw](../../Text/Command.htm#cmps)  [cmpsd](../../Text/Command.htm#cmps) |
| сканування  ланцюжка | [scas](#scas) адреса\_приймача  [scasb](../../Text/Command.htm#scas)  [scasw](../../Text/Command.htm#scas)  [scasd](../../Text/Command.htm#scas) |
| завантаження  елемента  з ланцюжка | [lods](#lods) адреса\_джерела  [lodsb](../../Text/Command.htm#lods)  [lodsw](../../Text/Command.htm#lods)  [lodsd](../../Text/Command.htm#lods) |
| збереження  елемента у ланцюжку | [stos](#stos) адреса\_приймача  [stosb](../../Text/Command.htm#stos)  [stosw](../../Text/Command.htm#stos)  [stosd](../../Text/Command.htm#stos) |
| одержання елементів ланцюжка  з порту  вводу/виводу | [ins](#ins) адреса\_приймача,номер\_порту  [insb](../../Text/Command.htm#ins)  [insw](../../Text/Command.htm#ins)  [insd](../../Text/Command.htm#ins) |
| виведення елементів ланцюжка у порт вводу/виводу | [outs](#outs) номер\_порту,адреса\_джерела  [outsb](../../Text/Command.htm#outs)  [outsw](../../Text/Command.htm#outs)  [outsd](../../Text/Command.htm#outs) |

Останній важливий момент, що стосується всіх ланцюжкових команд, — це напрямок обробки ланцюжка. Є дві можливості:

1. від початку ланцюжка до його кінця, тобто в напрямку зростання адрес;
2. від кінця ланцюжка до початку, тобто в напрямку спадання адрес.

Як ми побачимо нижче, ланцюжкові команди самі виконують модифікацію регістрів, що адресують операнди, забезпечуючи тим самим автоматичне просування по ланцюжку. Кількість байт, на які ця модифікація здійснюється, визначається кодом команди. А от знак цієї модифікації визначається значенням прапорця напрямку df (Direction Flag) у регістрі eflags/flags:

1. якщо df **= 0**, то значення індексних регістрів esi/si і edi/di буде автоматично збільшуватися (операція інкремента) ланцюжковими командами, тобто обробка буде здійснюватися в напрямку зростання адрес;
2. якщо df **= 1**, то значення індексних регістрів esi/si і edi/di буде автоматично зменшуватися (операція декремента) ланцюжковими командами, тобто обробка буде йти в напрямку спадання адрес.

Станом прапора df можна керувати за допомогою двох команд, що не мають операндів: [**cld**](../../Text/Command.htm#cld) (Clear Direction Flag) – очистити прапор напрямку. Команда скидає прапор напрямку df у 0. [**std**](../../Text/Command.htm#std) (Set Direction Flag) – встановити прапор напрямку. Команда встановлює прапор напрямку df у 1.

Операція пересилання ланцюжків.

Команди, що реалізують цю операцію-примітив, роблять копіювання елементів з однієї області пам'яті (ланцюжка) в іншу. Розмір елемента визначається застосовуваною командою. Система команд TASM надає програмісту чотири команди, що працюють з різними розмірами елементів ланцюжка:

*movs адреса\_приймача,адреса\_джерела* (MOVe String) — переслати ланцюжок;   
 *movsb* (MOVe String Byte) — переслати ланцюжок байт;   
 *movsw* (MOVe String Word) — переслати ланцюжок слів;   
 *movsd* (MOVe String Double word) — переслати ланцюжок подвійних слів.

Команда movs: movs адреса\_приймача, адреса\_джерела. Команда копіює байт, слово або подвійне слово з ланцюжка, що адресується операндом адреси\_джерела, у ланцюжок, що адресується операндом адреси\_приймача. Розмір елементів, що пересилаються, ассемблер визначає, виходячи з атрибутів ідентифікаторів, що вказують на області пам'яті приймача і джерела. Перелічимо набір дій, які потрібно виконати в програмі для того, щоб виконати пересилання послідовності елементів з однієї області пам'яті в іншу за допомогою команди movs. У загальному випадку цей набір дій можна розглядати як типовий для виконання будь-якої ланцюжкової команди:

1. встановити значення прапорця df у залежності від того, у якому напрямку будуть оброблятися елементи ланцюжка — у напрямку зростання або спадання адрес;
2. завантажити покажчики на адреси ланцюжків у пам'яті в пари регістрів ds:(e)si і es: (e)di;
3. завантажити в регістр ecx/cx кількість елементів, що підлягають обробці;
4. видати команду movs із префіксом rep.

На прикладі розглянемо, як ці дії реалізуються програмно. У цій програмі виробляється пересилання символів з одного рядка в інший. Рядки знаходяться в одному сегменті пам'яті. Для пересилання використовується команда-примітив movs із префіксом повторення rep.

Приклад 4.8.: Пересилання символів з одного рядка в інший

MODEL small

STACK 256

.data

source db 'Тестуємий рядок','$' ;рядок-джерело

dest db 19 DUP (' ') ;рядок-приймач

.code

assume ds:@data,es:@data

main: ;крапка входу в програму

mov ax,@data ;завантаження сегментних регістрів

mov ds,ax ;налашьування регістрів DS і ES

mov es,ax ;на адресу сегмента даних

cld ;скидання прапора DF — обробка рядка ;від початку до кінця

lea si,source ;завантаження в si зміщенняу рядка-

;джерела

lea di,dest ;завантаження в DS зміщенняу рядка-

;приймача

mov cx,20 ;для префікса rep — лічильник повторень

;(довжина рядка)

rep movs dest,source ;пересилання рядка

lea dx,dest

mov ah,09h ;виведення на екран рядка-приймача

int 21h

exit:

mov ax,4c00h

int 21h

end main

Приклад 4.9.: Порівняти два рядки на рівність.

.MODEL small

.STACK 256

.data

match db 'Strings match.','$'

failed db 'Strings mismatch','$'

string1 db '0123456789',10,13,'$' ;Рядки для порівняння

string2 db '0123456789','$'

.code

main:

mov ax,@data ;Завантажуємо дані в сегментні регістри

mov ds,ax

mov es,ax

cld ;Очищуємо прапор напрямку для порівняння

;символів в порядку зростання адрес

lea si,string1 ;Завантажуємо адреси рядків для порівняння

lea di,string2 ;в спеціальні індексні регістри si, di

mov cx,10 ;Завантажуємо в cx кількість ітерацій

repe cmpsb ;Повторюємо порівняння, поки символи рівні

jcxz equal ;Якщо лічильник cx=0, усі символи рівні

jne mismatch ;переходимо на відповідну мітку

equal:

lea dx,match ;Завантажуємо повідомлення про результат

mov ah,09h ;Функція DOS виведення рядка

int 21h

jmp exit

mismatch:

lea dx,failed

mov ah,09h

int 21h

exit:

mov ax,4c00h

int 21h

end main

Операція порівняння ланцюжків.

Команди, що реалізують цю операцію-примітив, роблять порівняння елементів ланцюжка-джерела з елементами ланцюжка-приймача. Тут ситуація з набором команд і методами роботи з ними аналогічна операції-примітиву пересилання ланцюжків. TASM надає програмісту чотири команди порівняння ланцюжків, що працюють з різними розмірами елементів ланцюжка:

1. cmps адреса\_приймача, адреса\_джерела (CoMPare String) — порівняти рядка;
2. cmpsb (CoMPare String Byte) — порівняти рядок байт;
3. cmpsw (CoMPare String Word) — порівняти рядок слів;
4. cmpsd (CoMPare String Double word) — порівняти рядок подвійних слів.

Команда cmps адреса\_приймача, адреса\_джерела:

1. адреса\_джерела визначає **ланцюжок-джерело** в сегменті даних. Адреса ланцюжка повинна бути заздалегідь завантажена у пари ds:esi/si;
2. адреса\_приймача визначає **ланцюжок-приймач**. Ланцюжок повинен знаходитися в додатковому сегменті, і його адреса повинна бути заздалегідь завантажена у пари es:edi/di.

Алгоритм роботи команди cmps полягає в послідовному виконанні операції віднімання (елемент ланцюжка-джерела — елемент ланцюжка-одержувача) над черговими елементами обох ланцюжків. Принцип виконання віднімання командою cmps аналогічний команді порівняння cmp. Вона, так само, як і cmp, виконує віднімання елементів, не записуючи при цьому результат, і встановлює прапори zf, sf і of. Після виконання віднімання чергових елементів ланцюжків командою cmps, індексні регістри esi/si і edi/di автоматично змінюються у відповідності зі значенням прапора df на значення, рівне розміру елемента порівнюваних ланцюжків.

Щоб змусити команду cmps виконуватися кілька разів, тобто робити послідовне порівняння елементів ланцюжків, необхідно перед командою cmps визначити префікс повторення. Вибір префікса визначається причиною, що приводить до виходу з циклу. Таких причин може бути дві для кожного з префіксів. Для визначення конкретної причини найбільш підходящим є спосіб, що використовує команду умовного переходу jcxz. Її робота полягає в аналізі вмісту регістра ecx/cx, і якщо він рівний нулеві, то керування передається на мітку, зазначену в якості операнда jcxz. Тому що в регістрі ecx/cx утримується лічильник повторень для ланцюжкової команди, що має кожний з префіксів повторення, то, аналізуючи ecx/cx, можна визначити причину виходу з зациклення ланцюжкової команди. Якщо значення в ecx/cx не дорівнює нулеві, то це означає, що вихід відбувся через збіг або розбіжність чергових елементів ланцюжків. Існує можливість ще більше конкретизувати інформацію про причину, що призвела до закінчення операції порівняння. Зробити це можна за допомогою команд умовної передачі керування (табл. 7.2), (приклад 4.9)

**Таблиця 7.2 – Сполучення команд умовної передачі керування з результатами команди cmps**

|  |  |  |  |
| --- | --- | --- | --- |
| для чисел без знака | | для чисел зі знаком | |
| Причина припинення операції порівняння | Команда умовного переходу, що реалізує перехід з цієї причини | Причина припинення операції порівняння | Команда умовного переходу, що реалізує перехід з цієї причини |
| операнд\_джерело > операнд\_приймач | ja | операнд\_джерело > операнд\_приймач | jg |
| операнд\_джерело = операнд\_приймач | je | операнд\_джерело = операнд\_приймач | je |
| операнд\_джерело <> операнд\_приймач | jne | операнд\_джерело <> операнд\_приймач | jne |
| операнд\_джерело < операнд\_приймач | jb | операнд\_джерело < операнд\_приймач | jl |
| операнд\_джерело <= операнд\_приймач | jbe | операнд\_джерело <= операнд\_приймач | jle |
| операнд\_джерело >= операнд\_приймач | jae | операнд\_джерело >= операнд\_приймач | jge |

Операція сканування ланцюжків.

Команди, що реалізують цю операцію-примітив, роблять пошук деякого значення в області пам'яті. Логічно ця область пам'яті розглядається як послідовність (ланцюжок) елементів фіксованої довжини розміром 8, 16 або 32 біт. Шукане значення попередньо повинно бути записане в регістр al/ax/eax. Вибір конкретного регістра з цих трьох повинен бути погоджений з розміром елементів ланцюжка, у якому здійснюється пошук. Система команд мікропроцесора надає програмісту чотири команди сканування ланцюжка. Вибір конкретної команди визначається розміром елемента:

1. *scas адреса\_приймача* (SCAning String) — сканувати ланцюжок;
2. *scasb* (SCAning String Byte) — сканувати ланцюжок байт;
3. *scasw* (SCAning String Word) — сканувати ланцюжок слів;
4. *scasd* (SCAning String Double Word) — сканувати ланцюжок подвійних слів.

Команда scas адрес\_приймача

Команда має один операнд, що позначає місцезнаходження ланцюжка в додатковому сегменті (адреса ланцюжка повинна бути заздалегідь сформована у es:edi/di). Транслятор аналізує тип ідентифікатора адреси\_приймача, що позначає ланцюжок у сегменті даних, і формує одну з трьох машинних команд scasb, scasw або scasd. Умова пошуку для кожної з цих трьох команд знаходиться в строго визначеному місці. Так, якщо ланцюжок описаний за допомогою директиви db, то шуканий елемент повинен бути байтом і знаходитися в al, а сканування ланцюжка здійснюється командою scasb; якщо ланцюжок описаний за допомогою директиви dw, то це — слово в ax, і пошук ведеться командою scasw; якщо ланцюжок описаний за допомогою директиви dd, то це — подвійне слово в eax, і пошук ведеться командою scasd. Принцип пошуку той же, що й у команді порівняння cmps, тобто послідовне виконання віднімання (вміст\_регістра\_акумулятора - вміст\_чергового\_елемента\_ланцюжка). У залежності від результатів віднімання виробляється встановлення прапорів, при цьому самі операнди не змінюються.

Приклад 4.10.: Пошук символу в рядку

.MODEL small

.STACK 256

.data

fnd db 0ah,0dh,'Symbol found!$'

nochar db 0ah,0dh,'Symbol not found.$'

string db 'Test string to search symbol.$',0ah,0dh,'$' ;Рядок для пошуку

strlen dw 30 ;Довжина рядка

.code

main:

mov ax,@data

mov ds,ax

mov es,ax

mov ah,09h ;Функція DOS виведення рядка

lea dx,string ;Завантажуємо адресу рядка

int 21h ;Виводимо рядок на екран

mov al,'i' ;Завантажуємо в al шуканий символ

cld ;Очищуємо флаг напрямку для

; порівняння в порядку зростання адрес

lea di,string ;Завантажуємо адресу рядка в di

mov cx,strlen ;Ініціалізуємо лічильник циклу

repne scasb ;Повторюємо сканування символів рядка

je found ; поки не знайдемо шуканий (в al)

;Переходимо на мітку found якщо знайшли символ

failed:

mov ah,09h ;Виведення повідомлення про невдачу

lea dx,nochar

int 21h

jmp exit

found:

mov ah,09h ;Виведення повідомлення про успіх

lea dx,fnd

int 21h

exit:

mov ax,4c00h

int 21h

end main

Завантаження елемента ланцюжка в акумулятор.

Ця операція-примітив дозволяє витягти елемент ланцюжка і помістити його в регістр-акумулятор al, ax або eax. Цю операцію зручно використовувати разом з пошуком (скануванням) для того, щоб, знайшовши потрібний елемент, витягти його (наприклад, для зміни). Можливий розмір елемента, що витягається, визначається застосовуваною командою. Програміст може використовувати чотири команди завантаження елемента ланцюжка в акумулятор, що працюють з елементами різного розміру:

1. *lods адреса\_джерела*(LOa String) - завантажити елемент із ланцюжка в регістр-акумулятор al/ax/eax;
2. *lodsb* (LOa String Byte) — завантажити байт із ланцюжка в регістр al;
3. *lodsw* (LOa String Word) — завантажити слово з ланцюжка в регістр ax;
4. *lodsd*  (LOa String Double Word) - завантажити подвійне слово з ланцюжка в регістр eax.

Команда lods адреса\_джерела (LOaD String) — завантажити елемент із ланцюжка в акумулятор al/ax/eax. Команда має один операнд, що позначає рядок в основному сегменті даних. Робота команди полягає в тому, щоб витягти елемент із ланцюжка за адресою, що відповідає вмісту пари регістрів ds:esi/si, і помістити його в регістр eax/ax/al. При цьому вміст esi/si піддається інкременту або декременту (у залежності від стану прапора df) на значення, рівне розміру елемента. Цю команду зручно використовувати після команди scas, що локалізує місце розташування шуканого елемента в ланцюжку. Префікс повторення в цій команді може і не знадобитися – усеі залежить від логіки програми.

Приклад 4.11.: Знайти довжину рядка.

.model small

.stack 100h

.data

str1 db 'String$' ;Рядок для визначення довжини

strlen dw 50 ;Максимальна довжина рядка

.code

Begin:

mov ax, @data

mov ds, ax

mov cx, strlen ;Ініціалізуємо лічильник циклу

lea si, str1 ;Завантажуємо адресу рядка в регістр

mov dl, '$' ;Завантажуємо символ кінця рядка

cld ;Очищуємо прапорець напрямку

LP:

lodsb ;Завантажуємо наступний символ в al

cmp al, dl ;Порівнюємо символ з символом кінця рядка

je exit ; якщо вони рівні, виходимо з циклу

loop LP ;Повторюємо цикл, поки cx не дорівнює 0

exit:

mov dx, strlen ;Знаходимо довжину рядка, віднявши сх

sub dx, cx ;від значення максимальної довжини

mov ax, 4c00h

int 21h

End Begin

Перенос елемента з акумулятора в ланцюжок.

Ця операція-примітив дозволяє зробити дію, зворотню команді lods, тобто зберегти значення з регістра-акумулятора в елемент ланцюжка. Цю операцію зручно використовувати разом з операцією пошуку (сканування) scans і завантаження lods, для того, щоб, знайшовши потрібний елемент, витягти його в регістр і записати на його місце нове значення. Команди, що підтримують цю операцію-примітив, можуть працювати з елементами розміром 8, 16 або 32 біт. TASM надає програмісту чотири команди збереження елемента ланцюжка з регістра-акумулятора, що працюють з елементами різного розміру:

1. *stos адреса\_приймача* (STOre String) — зберегти елемент із регістра-акумулятора al/ax/eax у ланцюжку;
2. *stosb* (STOre String Byte) — зберегти байт із регістра al у ланцюжку;
3. *stosw* (STOre String Word)— зберегти слово з регістра ax у ланцюжку;
4. *stosd* (STOre String Double Word) - зберегти подвійне слово з регістра eax у ланцюжку.

Команда має один операнд адреса\_приймача, що адресує ланцюжок у додатковому сегменті даних. Робота команди полягає в тому, що вона пересилає елемент з акумулятора (регістра eax/ax/al) в елемент ланцюжка за адресою, що відповідає вмісту пари регістрів es:edi/di. При цьому вміст edi/di піддається інкременту або декременту (у залежності від стану прапора df) на значення, рівне розмірові елемента ланцюжка. Префікс повторення в цій команді може і не знадобитися — усе залежить від логіки програми. Наприклад, якщо використовувати префікс повторення rep, то можна застосувати команду для ініціалізації області пам'яті деяким фіксованим значенням.

Введення елемента ланцюжка з порту вводу/виводу.

Дана операція дозволяє зробити введення ланцюжка елементів з порту вводу/виводу і реалізується командою ins, що має наступний формат:

*ins адреса\_приймача,номер\_порту* (Input String)- ввести елементи з порту вводу/виводу в ланцюжок.

Ця команда вводить елемент із порту, номер якого знаходиться в регістрі dx, в елемент ланцюжка, адреса якого визначається операндом адреса\_приймача. Незважаючи на те, що ланцюжок, у який вводиться елемент, адресується вказівкою цього операнда, його адреса повинна бути явно сформована у парі регістрів es:edi/di. Розмір елементів ланцюжка повинен бути погоджений з розмірністю порту — він визначається директивою резервування пам'яті, за допомогою якої виділяється пам'ять для розміщення елементів ланцюжка. Після пересилання команда ins робить корекцію вмісту edi/di на величину, рівну розміру елемента, що брав участь в операції пересилання. Зазвичай, при роботі ланцюжкових команд враховується стан прапора df. Подібно командам, що реалізують розглянуті вище ланцюжкові операції-примітиви, транслятор перетворить команду ins в одну з трьох машинних команд без операндів, що працюють з ланцюжками елементів визначеного розміру:

1. *insb* (INput String Byte) — ввести з порту ланцюжок байт;
2. *insw* (INput String Word) — ввести з порту ланцюжок слів;
3. *insd* (INput String Double Word)— ввести з порту ланцюжок подвійних слів.

Наприклад, виведемо 10 байт з області пам'яті pole у порт 5000h.

.data

pole db 10 dup (‘ ‘)

.code

...

push ds

pop es ;налаштування es на ds

mov dx,5000h

lea di,pole

mov cx,10

rep insb

Виведення елемента ланцюжка в порт вводу/виводу.

Дана операція дозволяє зробити виведення елементів ланцюжка в порт вводу/виводу. Вона реалізується командою outs, що має наступний формат: outs номер\_порту,адреса\_джерела (Output String) — вивести елементи з ланцюжка в порт вводу/виводу. Ця команда виводить елемент ланцюжка в порт, номер якого знаходиться в регістрі dx. Адреса елемента ланцюжка визначається операндом адреса\_джерела. Незважаючи на те, що ланцюжок, з якого виводиться елемент, адресується вказівкою цього операнда, значення адреси повинне бути явно сформоване в парі регістрів ds:esi/si. Розмір структурних елементів ланцюжка повинен бути погоджений з розмірністю порту. Він визначається директивою резервування пам'яті, за допомогою якої виділяється пам'ять для розміщення елементів ланцюжка. Після пересилання команда outs робить корекцію вмісту esi/si на величину, рівну розміру елемента ланцюжка, що брав участь в операції пересилання. При цьому, як звичайно, враховується стан прапора df. Подібно команді ins, транслятор перетворить команду outs в одну з трьох машинних команд без операндів, що працюють з ланцюжками елементів визначеного розміру:

1. outsb (OUTput String Byte) — вивести ланцюжок байт у порт вводу/виводу;
2. outsw (OUTtput String Word) — вивести ланцюжок слів у порт вводу/виводу;
3. outsd (OUTput String Double Word) — вивести ланцюжок подвійних слів у порт вводу/виводу.

Як приклад розглянемо фрагмент програми, що виводить послідовність символів у порт вводу/виводу, що відповідає принтерові (номер 378 (lpt1)).

.data

str\_pech db 'Текст для друку'

.code

...

mov dx,378h

lea di,str\_pech

mov cx,16

rep outsb

**Контрольні запитання**

1. Який регістр є неявним параметром у командах циклу?

2. Для чого служить регістр ознак? Які існують ознаки?

3. Які є види умов у командах пере

ходу? Які є види безумовних переходів?

4. Для чого потрібно різні типи безумовних переходів? Чим вони відрізняються?

5. Який тип безумовного переходу використовує асемблер по замовчуванню?

6. Скільки байт займають команди безумовного переходу?

7. Чому умовні переходи лише короткі?

**Завдання до лабораторної роботи №4**

**Перший рівень**

ЗАВДАННЯ 1

1.1.Наберіть програму 4.1.

1.2.Завантажте програму в відладчик

1.3. Виконайте **програму 4.1** по крокам.

***Програма 4.1***. В програмі порівнюються два числа й виводиться результат порівняння у вигляді текстового повідомлення. В програмі використовуються функція виводу на екран (**DOS 09h**) та функція вводу з клавіатури з відображенням (**DOS 01h**).

title команди переходу

model small

.stack 256

.data ;сегмент даних

var1 db 0

var2 db 0

mess1 db 'Введіть перше число: ','$'

mess2 db 'Введіть друге число: ','$'

mess\_G db 'Перше > другого',0Dh,0Ah,'$'

mess\_L db 'Перше < другого',0Dh,0Ah,'$'

mess\_E db 'Перше = другому',0Dh,0Ah,'$'

CRLF db 0Dh,0Ah,'$'

.code ;сегмент коду

begin:

mov ax,@data

mov ds,ax

mov dx,offset mess1

mov ah,09

int 21h

;Введення першого числа в регістр AL

mov ah,01h

int 21h

;зберегти перше число в var1

mov var1,al

;повернення каретки

mov dx,offset CRLF

mov ah,9

int 21h

mov dx,offset mess2

mov ah,9

int 21h

;Введення другого числа в регістр AL

mov ah,01h

int 21h

;зберегти друге число в var2

mov var2,al

mov dx,offset CRLF

mov ah,9

int 21h

mov al,var1

;порівняння чисел

cmp al,var2

jg msgr

jl msle

mov dx,offset mess\_E

jmp prn

msgr:

mov dx,offset mess\_G

jmp prn

msle:

mov dx,offset mess\_L

prn:

mov ah,9

int 21h

mov ax,4c00h ;стандартний вихід

int 21h

end begin

Користуючись **відладником**, перевірити всі переходи в програмі **lab4\_1n.asm**, записати до звіту значення чисел, що було введено та стани ознак, які зазначають перехід, наприклад:

Перше число (op1): 5

Друге число (op2): 4

Результат команди **cmp** op1,op2: **zf**=0, **sf**=0, **cf**=0, **of**=0

Команда **JGE** виконала перехід на мітку m1 за умови **sf**=**of**.

**Другий рівень**

Виконати завдання 1

**Третій рівень**

Виконати завдання 1 та 2

ЗАВДАННЯ 1

1. Використати та застосувати команди, вказані в варіанті **таблиця 4.1.** написати програму мовою асемблера (TASM) згідно варіанту.
2. Організувати цикл за допомогою команди **JCXZ**
3. Організувати цикл за допомогою команди **LOOP**
4. Користуючись **відладником**, перевірити всі переходи в програмі
5. Завантажити програму в відладчик і відобразити у звіті зміну вмісту регістрів.

**Таблиця 4.1 – Варіанти завдання**

|  |  |
| --- | --- |
| Номер варіанта | Команди |
| 1 | **JAE**, **JLE** |
| 2 | **JBE**, **JA** |
| 3 | **JGE**, **JZ** |
| 4 | **JB**, **JLE** |
| 5 | **JNE**, **JA** |
| 6 | **JG**, **JC** |
| 7 | **JL**, **JBE** |
| 8 | **JA**, **JS** |
| 9 | **JE**, **JBE** |
| 10 | **JB**, **JLE** |

ЗАВДАННЯ 2

2.1. Написати програму мовою асемблера (TASM) згідно варіанту

2.2. Завантажити програму в відладчик і відобразити у звіті зміну вмісту регістрів.

**Варіанти ЗАВДАННЯ 1:**

1. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та виводить їх на екран у зворотному порядку.
2. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та виводить їх на принтер.
3. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та заміняє у ньому усі букви S на 0.
4. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та виводить на екран кожну третю букву.
5. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та рахує кількість букв А у ньому.
6. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та виводить кількість слів у ньому (слова розділені пробілами).
7. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та заміняє у ньому пробіли на крапки.
8. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та замінює у ньому усі букви S на 0.
9. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та повертає його довжину.
10. Використовуючи ланцюжкові команди написати програму, яка зчитує рядок символів з клавіатури та виводить на принтер його довжину.

**Варіанти ЗАВДАННЯ 2:**

1. Заданий текст, у якому є хоча б одна крапка. Перетворити текст, вилучивши з нього всі коми, що передують першій крапці, і замінивши всі пробіли в тексті на символ “+”.
2. Даний рядок символів s1, s2, …, sn (n<80). Видалити з неї групи символів між дужками “(“ і “)”. Самі дужки теж повинні бути видалені. Передбачається, що усередині кожної пари дужок немає інших дужок.
3. Заданий текст. Групи символів, розділені пробілами (одним або декількома), які вважатимемо словами. У цих словах, які мають закінчення “ing”, зробити заміну “ing” на “ed”.
4. Даний рядок символів s1, s2, …, sn (n<70). Видалити із заданого рядка символи, що є цифрами 1,2,3,4,5, і замінити кожну букву “a” символом “ про”.
5. Даний текст, кожний символ якого може бути буквою, цифрою або знаком “+”, “-”, “\*”. Знайти найдовшу групу цифр. Вивести результат на екран.
6. Заданий текст, що полягає зі слів, розділених не менш ніж одним пробілом. Видалити із цього тексту слова з непарними номерами й записати букви в словах з парними номерами у зворотному порядку.
7. Задано два речення. Знайти саме коротке слово першого речення, якого немає в другому реченні.
8. Даний рядок символів s1, s2, …, sn (n<50). Знайти перше натуральне i, для якого кожний із символів sj і sj+1 збігається із заданою буквою а. Якщо такої пари символів немає в послідовності, то відповіддю повинне бути число 0.
9. Із заданого тексту вибрати ті слова, які зустрічаються в ньому один раз ( у тому порядку, у якому вони зустрічаються в тексті).
10. Заданий текст, у якому слова відділені пробілами. Знайти число входжень заданого слова в текст і видалити з тексту слова, що збігаються із заданим словом.

Література:

[2, c. 137; 4, c.170; 8, c.190]