Команди переходу

Команда переходу JMP

Ця команда означає перехід (***jump***) до виконання деякої іншої команди, а точніше кажучи – перехід до деякої адреси у пам'яті, де повинна розташовуватися потрібна команда. Команду JMP ще звуть командою безумовного переходу – вона подібна оператору GOTO мови програмування високого рівня.

Формат запису команди:

```
jmp [тип переходу] операнд
```

Квадратними дужками позначений необов'язковий елемент запису – тип переходу, який може бути:

- **short** (короткий перехід від -128 до +127 відносно поточного значення регістру EIP)
- **near** (перехід у межах поточного сегменту коду – це той сегмент, поточна адреса якого міститься у регістрі CS)
- **far** (перехід у інший сегмент. Операнд повинен містити значення нової сегментної адреси)

Зазвичай тип переходу вказувати не потрібно – компілятор визначить його сам.

Загалом у програмах на асемблері найуживанішим є такий формат запису команди JMP:

```
jmp мітка
```

Операндом команди JMP зазвичай прийнято вказувати мітку, яка позначає команду, на яку виконується перехід. Наприклад:

```
    mov eax, 2
    jmp @next
    add eax, 5
    inc ebx
@next:
    sub ecx, eax
```

Команди умовних переходів Jcc

Виконують перехід, якщо виконується деяка умова (*Jump if Condition Is Met*) . Узагальнений формат запису таких команд:

```
Jcc операнд
```

де cc – мнемоніка відношення. У вихідних текстах на асемблері операнд цієї команди зазвичай є міткою, проте у машинному коді буде записуватися у вигляді числового значення, яке при виконанні команди додаватиметься до вказівника команд.

Мнемоніка команди умовного переходи складається з 'J' плюс символи, які позначають умову відношення. Такими символами можуть бути:

E – equal (дорівнює)
L – less (менше для чисел зі знаком)
G – greater (більше для чисел зі знаком)
N – not (не)
A – above (більше для чисел без знаку)
B – below (менше для чисел без знаку)

а також:

C – carry (перенос)
S – sign (знак)
P – parity (парність)
O – overflow (переповнення)
Z – zero (нуль)

а також:

CXZ
ECXZ
RCXZ

Наприклад, команда JE означає перехід (jump), якщо дорівнює, JNE – jump якщо не дорівнює, JGE – jump якщо не більше, JL – jump якщо менше, JECXZ – jump якщо регістр ECX дорівнює 0.

З документації Intel:

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the Jcc instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16,* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of –128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each Jcc mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The Jcc instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the Jcc instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JRCXZ, JECXZ and JCXZ instructions differ from other Jcc instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as LOOPNE). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute 2^{64}, 2^{32} or 64K times (not zero times).

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

In 64-bit mode, operand size is fixed at 64 bits. JMP Short is RIP = RIP + 8-bit offset sign extended to 64 bits. JMP Near is RIP = RIP + 32-bit offset sign extended to 64-bits.

Приклад

Нехай у програмі на асемблері перемінна x оголошена як однобайтова перемінна

```
x db ?
```

Як запрограмувати, наприклад, такі дії, записані мовою високого рівня:

```
if (x > 17)
     y = 50;
else y = 10;
```

Рішення:

```
  cmp x, 17
  jle @else
  mov y, 50
  jmp @endif
@else:
  mov y, 10
@endif:
```

А як запрограмувати, наприклад, таке:

```
if (x > 200)
     y = 50;
else y = 10;
```

Якщо у тексті, записаному вище на асемблері, просто змінити 17 на 200:

```
  cmp x, 200          ;200 = 0C8h
  jle @else
  mov y, 50
  jmp @endif
@else:
  mov y, 10
@endif:
```

то програма буде працювати не так, як ми хотіли – наприклад, при x=100 отримаємо y=50, а не 10.
У чому причина? Оскільки перемінна x є однобайтовою, то порівняння із константою 200 = $C8_{16}$ буде у однобайтовому форматі, а однобайтовий код 0C8h буде сприйматися процесором як від'ємне число (-56).
Замість команди JLE треба скористатися командою JBE:

```
  cmp x, 200
  jbe @else           ;JBE, а не JLE
  mov y, 50
  jmp @endif
@else:
  mov y, 10
@endif:
```

Програмування циклів

Прості цикли

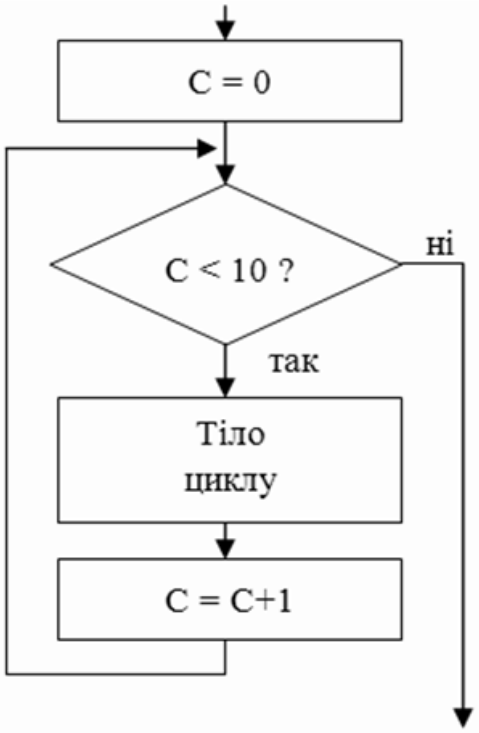
Нехай потрібно щось виконати 10 разів. Це "щось" будемо називати "тілом циклу". Тіло циклу може містити деяку множину рядків програмного коду на асемблері.

Можна виділити два основні різновиди циклів:

- **цикл з передумовою** – спочатку перевіряється умова, і якщо вона є дійсною, то виконується перша ітерація циклу;

- **цикл з постумовою** – спочатку виконується одна ітерація, а потім перевіряється умова продовження циклу.

Цикл з передумовою можна побудувати так. Використаємо перемінну-лічильник (C), яка буде зберігати індекс поточної ітерації. Спочатку лічильник обнулюємо, а після кожного виконання тіла циклу лічильник будемо збільшувати на одиницю. Умовою виконання циклу буде перевірка (C менше 10 ?). Це можна відобразити таким чином:



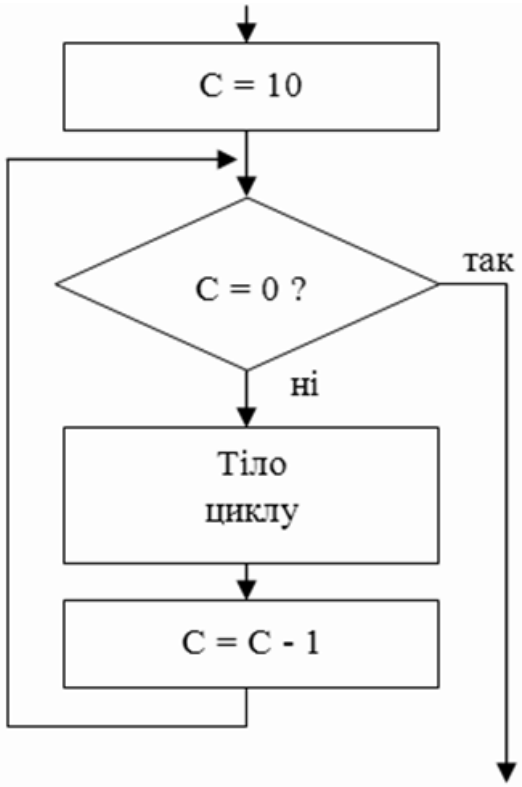

```
C = 0;
while (C < 10)
  {

  . . .      //деякий програмний код тіла циклу

  C = C + 1;
  }
```

Приклад реалізації такого циклу на асемблері наданий нижче.

```
  xor ecx,ecx          ; обнулюємо лічильник – регістр ECX
cycle:
  cmp ecx, 10          ; порівнюємо лічильник з потрібною кількістю повторень
  jge exit             ; якщо лічильник >= 10, то вихід з циклу

  . . .                ; деякий програмний код тіла циклу

  inc ecx              ; збільшуємо лічильник на 1
  jmp cycle            ; перехід на наступну ітерацію циклу

exit:                  ; ця мітка позначає деякий програмний код після цього циклу
```

Значення лічильника будемо зберігати у регістрі ECX. Перевірку значення лічильника і перехід по умові запрограмуємо командами CMP та JGE. Якщо лічильник більше або дорівнює 10, то цикл завершується – виконується перехід на мітку `exit` яка позначає те, що повинне виконуватися вже після циклу. Циклічне повторення програмного коду циклу забезпечується командою безумовного переходу JMP.

Іншим різновидом циклу із передумовою є такий – у лічильник спочатку записується потрібна кількість повторень, а на кожній ітерації лічильник зменшується на 1. Цикл повторюється, поки лічильник не нуль.




```
C = 10;          //кількість повторень = 10
while (C != 0)
  {

  . . .          //тіло циклу

  C = C - 1;
  }
```

Приклад реалізації такого циклу на асемблері наданий нижче.

```
  mov ecx, 10          ; у лічильник – регістр ECX записуємо кількість повторень
cycle:
  jecxz exit           ; якщо у регістрі ECX нуль, то вихід з циклу – перехід на мітку exit

  . . .                ; тіло циклу

  dec ecx              ; зменшуємо лічильник на 1
  jmp cycle

exit:                  ; ця мітка позначає деякий програмний код після цього циклу
```

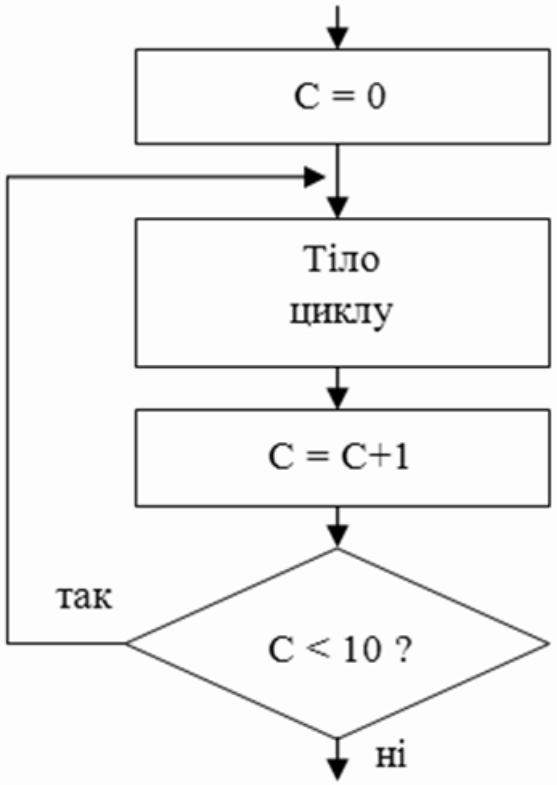
У цьому прикладі не випадково для зберігання значення лічильника використовується саме регістр ECX. У архітектурі x86 є спеціальна команда JECXZ, яка виконує перехід, якщо у регістрі ECX нуль.

Взагалі можна сказати, що регістр ECX традиційно використовується для лічильника циклу, і більше того, деякі команди (наприклад, обробки рядків даних) працюють саме з регістром ECX, у якому повинна бути задана кількість повторень.

Цикл із постумовою

Умова продовження циклу після виконання першої ітерації

```
C = 0;
do
  {

  . . .     //тіло циклу

  C = C + 1;
  }
while (C < 10);
```

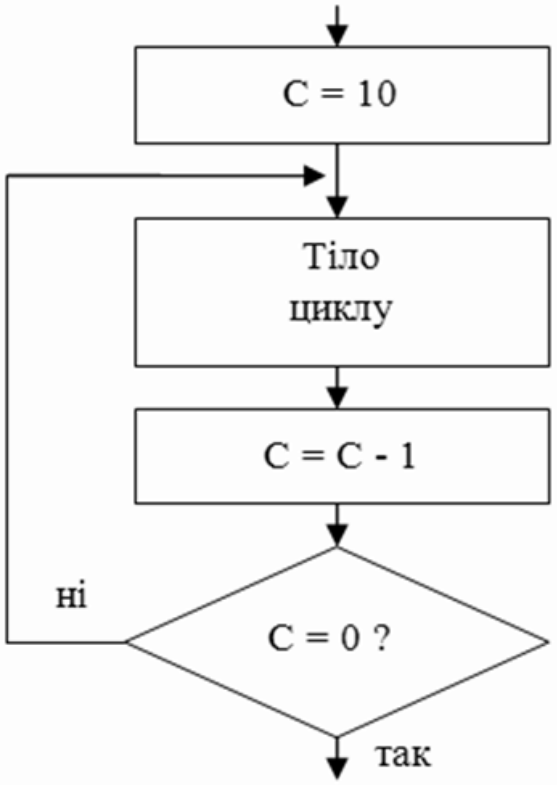


Розглянемо програмний код на асемблері.

```
  xor ecx,ecx          ; обнулюємо лічильник – регістр ECX
cycle:

  . . .                ; тіло циклу

  inc ecx              ; збільшуємо лічильник на 1
  cmp ecx, 10          ; порівнюємо лічильник з 10
  jl cycle             ; якщо лічильник менше – продовжуємо виконання циклу
```

Можна дещо спростити реалізацію циклу, якщо у лічильник записати кількість ітерацій, а після кожної ітерації зменшувати лічильник. Порівнювати лічильник з нулем краще не командою CMP, а командою переходу JNZ. Команда JNZ аналізує прапорець ZF – біт 6 регістру EFLAGS. Якщо цей біт дорівнює 1, то це означає, що результат останньої операції є нуль.

Крім того, такий варіант менш критичний у сенсі запобігання спотворенню корисної інформації на противагу варіанту на основі команди CMP, яка може змінювати значення бітів регістру EFLAGS. Це може бути суттєвим, наприклад, при програмуванні арифметичних операцій підвищеної розрядності.



Приклад програмного коду циклу із постумовою:

```
  mov ecx,10          ; у лічильник – регістр ECX записуємо кількість повторень
cycle:

  . . .               ; тіло циклу

  dec ecx             ; зменшуємо лічильник на 1
  jnz cycle           ; якщо лічильник не 0 – перехід на виконання тіла циклу
```

Використання команд LOOP, LOOPE, LOOPNE

Ці команди забезпечують циклічне виконання фрагменту коду, використовуючи у якості лічильника регістр RCX, або ECX або CX (у залежності від розміру адрес 64 біт, або 32 біт або 16 біт). Після кожної ітерації лічильник зменшується на одиницю. Якщо лічильник 0, то цикл припиняється і програма переходить до виконання наступної за LOOP команди. Якщо лічильник не 0, виконується ближній (near) перехід на те, що вказує операнд команди LOOP.

Команда	Що виконується
LOOP операнд	Декремент лічильника. Короткий перехід, якщо лічильник ≠ 0
LOOPE операнд	Декремент лічильника. Короткий перехід, якщо лічильник ≠ 0 та ZF = 1
LOOPNE операнд	Декремент лічильника. Короткий перехід, якщо лічильник ≠ 0 та ZF = 0

Операндом цих команд є відносний зсув – зсув відносно поточного значення регістру IP/EIP/RIP. У тексті на асемблері цей операнд зазвичай записується як мітка, проте на рівні машинного коду це декодується у 8-бітове числове значення зі знаком, яке додається до вказівника інструкцій (IP). Значення зсуву можуть бути від -128 до +127.

```
  mov ecx,10          ;у лічильник – регістр ECX записуємо кількість повторень
cycle:

  . . .               ;тіло циклу

  loop cycle          ;перехід на cycle, якщо ECX ≠ 0
```

Різновиди команди LOOP – команди LOOPE та LOOPNE використовують у якості умови виконання/завершення циклу окрім значення лічильника також значення біту ZF регістру EFLAGS. Значення ZF формується в результаті виконання операції у тілі циклу.

Розглянемо приклад циклу, який розрахований на максимальну кількість ітерацій 10, проте може бути перерваний у випадку, якщо у регістрі EDX буде значення 7. Для цього використаємо команду LOOPNE.

```
  mov ecx,10            ; у лічильник – регістр ECX записуємо кількість повторень
cycle:

  . . .                 ; тіло циклу

  cmp edx, 7
  loopne cycle          ; перехід на cycle, якщо (ECX ≠ 0) та (EDX ≠ 7)
```

Вкладені цикли

Взагалі, для будь-якого циклу бажано у якості лічильника використовувати регістр, наприклад, регістр ECX. Оскільки у 32-бітних процесорів x86 регістрів загального призначення небагато, то при програмуванні вкладених циклів може виникнути проблема нестачі регістрів для лічильників циклів. Тому, може знадобитися зберігання лічильників у пам'яті. Щоб, по-можливості, зменшити втрати швидкодії рекомендується лічильник внутрішнього циклу зберігати у регістрі, а у пам'яті зберігати лише лічильники зовнішніх циклів.

Розглянемо приклад трьох вкладених циклів:

```
x = 30;
do
  {
  y = 100;
  do
    {
    . . .
    z = 20;
    do
      {
      . . .                    ; тіло внутрішнього циклу
      z = z-1;
      }
    while (z>0);
    . . .
    y = y-1;
    }
  while(y>0);
  . . .
  x = x-1;
  }
while (x>0);
```

Один з варіантів реалізації на асемблері. Лічильник внутрішнього циклу у регістрі ECX, а лічильники зовнішніх циклів у вигляді перемінних (x, y).

```
.data
   x dd ?                          ; ці перемінні будуть лічильниками циклів
   y dd ?

.code
mov x, 30
@cycle1:
   . . .                           ; команди циклу1
   mov y, 100
   @cycle2:
      . . .                        ; команди циклу2
      mov ecx, 20
      @cycle3:
         . . .                     ; тіло внутрішнього циклу3
      loop @cycle3
      . . .                        ; команди циклу2
   dec y
   jnz @cycle2
   . . .                           ; команди циклу1
dec x
jnz @cycle1
```

Далі про лічильники вкладених циклів у стеку: варіант push-pop

```
mov ecx, 30
@cycle1:                                    ; зовнішній цикл
push ecx
   . . .
   mov ecx, 100
   @cycle2:
   push ecx
      . . .
      mov ecx, 20
      @cycle3:
         . . .                              ; тіло внутрішнього циклу
      loop @cycle3
      . . .
   pop ecx
   loop @cycle2
   . . .
pop ecx
loop @cycle1
```

Тут можна звернути увагу на доцільність використання у вихідному тексті відступів зліва, щоб зробити асемблерний текст зрозумілішим та гарнішим.

Лічильники циклу як локальні перемінні. Використання директиви LOCAL

Оголосимо процедуру, яка має три параметри, які передаються через стек, та одну локальну перемінну.

```
MyProc proc
  local counter:DWORD     ;створення 4-байтової локальної перемінної counter

  . . .                   ;програмний код тіла процедури

  ret 12                  ;вилучаємо 3 параметри по 4 байт
MyProc endp
```

Локальна перемінна створюється у стеку.

У вікні дизасемблера можна подивитися, який насправді компілятор створює код:

```
MyProc proc
00401055  push        ebp
00401056  mov         ebp,esp
00401058  add         esp,0FFFFFFFCh
  local counter:DWORD


  . . .                   ;програмний код тіла процедури


  ret 12                  ;вилучаємо 3 параметри по 4 байт
00401075  leave
00401076  ret         0Ch
MyProc endp
```