Зміст лекції

1. Організація циклів на мові Assembler
2. Приклади циклів з використанням команд умовних переходів
3. Команди LOOPx
4. Робота зі стеком
5. Обробка масивів

У програмуванні на мові асемблера x86-32 цикл - це структура управління програмою, яка дозволяє виконувати певний блок інструкцій кілька разів. Як Ви знаєте з програмування є наступні основні типи циклів:

- Цикл з передумовою
- Цикл з постумовою
- Цикл з лічильником

При виконанні циклу з передумовою – відбувається перевірка якоїсь умови і за відповідності виконується тіло циклу.

При роботі з циклами з постумовою – спочатку виконується тіло циклу (хоча б один раз), а потім виконується перевірка умови для його повторення.

Отже ці цикли, з перед- та постумовою дозволяють нам переривати повторювані дії на певному етапі виконання. Наприклад за настання певної умови, або за настанні команди примусового виходу з циклу.

Також при роботі з цими циклами є важливим контроль умови виходу з циклу, щоб уникати нескінченного виконання циклу з якого програма зама не зможе вийти, наприклад перевірка умови, яка ніколи не виконується і відповідно туло циклу виконується нескінченну кількість разів і не переходить до інших елементів програми.

Цикл з лічильником – дозволяє виконати тіло циклу відому, заздалегідь визначену кількість разів. Тобто ми маємо якісь змінну, так званий лічильник, з якою ми виконуємо фіксовану кількість повторень.

На мовах високого рівня, ті же мові С передбачаються спеціальні команди для циклів, такі як Do-While, Until, а для циклів з лічильником – команда for.

В мові Assembler цикли організовуються трошки примітивніше, а саме наступним чином:

- з використанням команд умовних та безумовних переходів
- з використанням групи команд LOOPx

Зазвичай цикли з перед- та постумовою організовуються з використанням саме команд умовних та безумовних переходів, а цикли з лічильником використовують групу команд LOOPx. X у назві обозначає певні ознаки умовних переходів, які також можуть враховуватись для виходу з циклу. LOOPE, LOOPZ, LOOPNE, LOOPNZ з умовами рівності якомусь значенню або нулю, які вже відомі вам з попередніх лекцій. Більш детально ці команди ми розглянемо пізніше.

Отже, для розуміння роботи вищезазначених циклів розглянемо декілька прикладів:

Приклад циклу з використанням команд умовних переходів з використанням певного лічильника для наочності, але умова може бути будь-яка.

Цикл з постумовою

```
…
        MOV CX, n    ; n – кількість повторень циклу
begin:               ; початок тіла циклу
        …            ; тіло циклу
        DEC CX       ; перевірка умови продовження циклу
        CMP CX, 0    ; якщо CX = 0, то вийти з циклу
        JNE begin    ; якщо  CX <> 0, то перейти на початок
        …            ; наступні оператори програми
```

Цикл з передумовою

```
…
        MOV CX, n    ; n – кількість повторень циклу
begin:
        CMP CX, 0    ; перевірка умови продовження циклу
        JE end       ; якщо CX = 0, то вийти з циклу
        …            ; початок тіла циклу
        …            ; тіло циклу
        DEC CX
        JMP begin    ; перейти на початок
end:    …
        …            ; наступні оператори програми
```

Команди LOOPx

Команди реалізації класичного циклу з постумовою

LOOP, LOOPE, LOOPNE, LOOPZ, LOOPNZ

Всі команди LOOPx для збереження кількісті повторень тіла циклу використовують регістр CX.

Синтаксис команди LOOP

LOOP мітка

Логіка роботи команди

<CX> = кількість повторень

мітка:

… ;тіло циклу

<CX> = <CX> – 1

якщо <CX> <> 0 перейти на мітку

Тобто LOOP самостійно виконує декремент лічильника CX.

LOOPZ або LOOPE (Loop if Zero/Equal): Ця інструкція виконує наступну ітерацію циклу, якщо лічильник циклу не дорівнює нулю і прапорець Zero/Equal (ZF) встановлений. Значення лічильника зменшується на одиницю.

LOOPNE або LOOPNZ (Loop if Not Zero/Not Equal): Ця інструкція виконує наступну ітерацію циклу, якщо лічильник циклу не дорівнює нулю і прапорець Zero/Equal (ZF) не встановлений. Значення лічильника зменшується на одиницю.

Слід зазначити, що ці інструкції дещо застарілі, і їх використання може бути не найбільш ефективним способом реалізації циклів у сучасних програмах.

Приклад циклу з використанням команд LOOP.

Приклад 1. Розрахувати значення факторіалу

result = n! = 1 x 2 x 3 x … x n. Факторіал не може бути від'ємним.

```
__asm {; якщо p буде більше 16 бітів, розрахунок цієї програми буде некоректним
        MOV CX, n     ;  внесення кількість повторень
        MOV BX, 1     ;  BX регістр для множення
        MOV AX, BX    ;  <AX>=1
        JCXZ end      ; Перевірка чи <CX> = 0
begin:  MUL BX        ; <DX:AX> = <AX>x<BX>
        INC BX        ; Регістр BX = BX+1
        LOOP begin    ; перевірка чи <CX> = 0,
                      якщо не дорівнює, цикл повтрюється
end:    MOV result, AX
}
```

Приклад 2. Розрахувати значення факторіалу

p = n! = 1 x 2 x 3 x … x n. Факторіал не може бути від'ємним.

```
__asm {; якщо p буде більше 16 бітів, розрахунок цієї програми буде некоректним
        MOV CX, n     ;  внесення кількість повторень
        MOV AX, 1
        JCXZ end      ; Перевірка чи <CX> = 0
begin:  MUL CX        ; <DX:AX> = <AX>x<CX>
        LOOP begin    ; перевірка чи <CX> = 0,
                      якщо не дорівнює, цикл повтрюється
end:    MOV p, AX
}
```

Команди **LOOPE (LOOPZ)** перехід по лічильнику та якщо дорівнює. Аналізують ознаку регістра EFLAGS ZF = 1

Синтаксис команди LOOPE (LOOPZ)

LOOPE (LOOPZ) мітка

Логіка роботи команди

<CX> = кількість повторень

мітка:

… ;тіло циклу

<CX> = <CX> – 1

якщо (<CX> <> 0 та <ZF> = 1) перейти на мітку

Команди **LOOPNE (LOOPNZ)** перехід по лічильнику та якщо НЕ дорівнює. Аналізують ознаку регістра EFLAGS ZF = 0

Синтаксис команди LOOPNE (LOOPNZ)

LOOPNE (LOOPNZ) мітка

Логіка роботи команди

<CX> = кількість повторень

мітка:

… ;тіло циклу

<CX> = <CX> – 1

якщо (<CX> <> 0 та <ZF> = 0) перейти на мітку

Приклад. Знайти значення суми чисел S = 1+2+3+…+n. Обчислення припинити якщо S=K або перебрані всі n чисел.

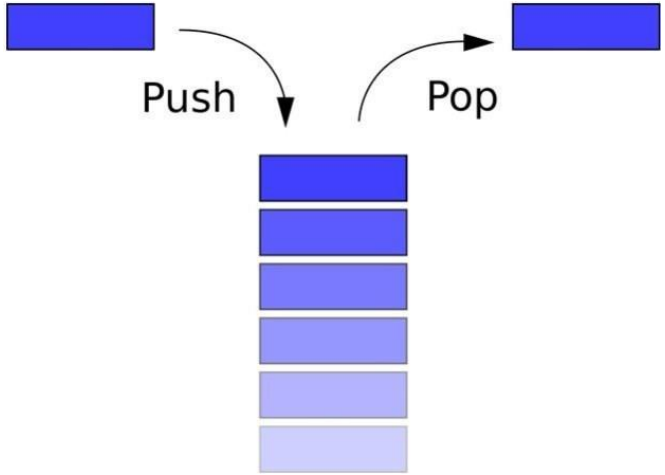
```
__asm {
MOV CX, n
XOR AX, AX ; очистити регістр, фактично <AX>=0
XOR BX, BX    ; очистити регістр, фактично <BX>=0
JCXZ end
    begin:  INC BX
            ADD AX, BX
            CMP AX, k
            LOOPNE begin
     end:  MOV S, AX
}
```

Розглянемо ще роботу зі стеком. Ми його згадували раніше у лекціях, але, певен, більшість з Вас не дуже зрозуміла, що це і як ним користуються.

Stack в перекладі з англійської перекладається як «стопка». Відповідно він працює за принципом LIFO – Last In First Out, тобто останній зайшов – перший вийшов. Приклад – колода карт, або магазин з патронами.

Відповідно для запису «проштовхування» в стек використовується команда push %sourse%, наприклад push ax. Відповідно значення з регістра AX записується у верхню вільну комірку стека.

Для читання зі стеку доступна тільки саме ця верхня комірка стеку. Прочес отримання називається «Виштовхування» і має синтаксис pop %destination%, наприклад pop bx. Значення з верхівки стеку переписуєтсья в регістр BX і теперь доступною стає наступна комірка.



Розміри комірок відповідають розмірності даних які ми проштовхуємо:

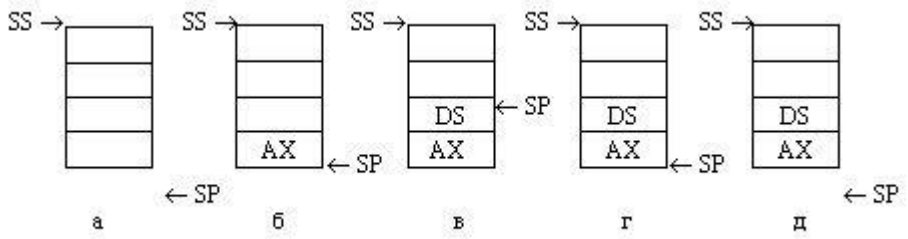
Push al = char – 1 байт

Push ax = short int – 2 байти

Push eax = int – 4 байти.

Початкова точка запису для конкретного фрейму програми (потоку) вказується в регістрі BP – base pointer. Він використовується коли викликається якась функція і відповідно для неї записи починаються з пустої комірки, яка не завжди може бути першою в основному фреймі програми. При цьому по завершенню функції дані записані в стек теж мають бути доступні.

На конкретну комірку для запису вказує регістр SP – stack pointer. І перед кожним записом він збільшується на розмір комірки у яку ми записували дані. Автоматично для команд push та pop.



Звертаю Вашу увагу, що у вихідному положенні показник стеку SP вказує на комірку, що лежить під дном стеку і не входить в нього.

Відповідно використання стеку зручно нам, коли ми маємо працювати з декількома лічильниками, наприклад для багатовимірних масивів.

Наприклад для двовимірного масива, зазвичай кількість стовпців, або комірок в кожному рядку, однакова для кожного рядка. Відповідно перейшовши до наступного рядка ми можемо зберегти його в стек і призначити лічильнику CX значення у кількість стовпців (комірок) кожного рядка, а по завершенню – виштовхнути лей лічильник назад зі стека в CX.