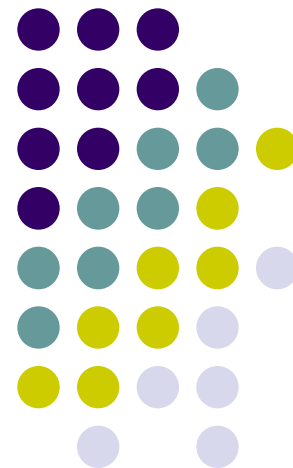
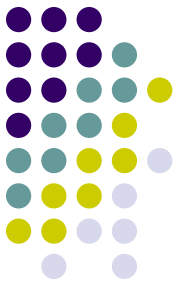


Тема 5

Команды управления



Команды управления программным потоком



Можно выделить следующие категории таких команд:

- переходы
 - безусловные
 - условные
- вызовы подпрограмм
- прерывания

Особенности работы команд управления программным потоком



- *безусловные переходы (JMP)*
содержимое регистров EIP и, возможно, CS изменяется в зависимости от информации, хранящейся в команде
(это справедливо для всех категорий)
- *условные переходы (Jxxx)*
переход выполняется только при выполнении определенного условия
- *вызовы подпрограмм (CALL)*
перед выполнением перехода в стеке сохраняются старые значения EIP и CS
- *выход из подпрограмм (RET)*
значения EIP и CS извлекаются из стека



Особенности работы команд управления программным потоком (продолжение)

- *прерывание (INT)*
перед выполнением перехода в стеке сохраняются старые значения регистра флагов, EIP и CS
- *выход из прерывания (IRET)*
значения регистра флагов, EIP и CS извлекаются из стека

Классификация безусловных переходов по расстоянию



- *короткие (JMP SHORT)*
можно выполнить переход только на (-128..127) байт относительно старого значения EIP
- *ближние (JMP NEAR)*
переход осуществляется в пределах одного сегмента, значение CS не изменяется
- *дальние (JMP FAR)*
осуществляется переход в другой сегмент, значение CS изменяется

Формат команды перехода (в общем случае)

`jmp адрес_перехода`

Классификация безусловных переходов по типу информации в команде



- *прямые* `jmp метка` ;Переход на метку
в команде хранится адрес, который необходимо занести в CS и EIP (в зависимости от расстояния). Иначе – *прямым* называется переход, в команде которого в явной форме указывается метка, на которую нужно перейти
- *косвенные* `jmp word[bx]`
адрес, который необходимо занести в CS и EIP, вычисляется по правилам косвенной адресации
- *регистровые* `jmp bx` ;Переход по адресу в BX
частный случай косвенной адресации – адрес хранится в регистре

Короткие переходы могут быть только прямыми!

Разновидности безусловных переходов



1) *прямой короткий переход:*

```
code segment
```

```
...
```

```
    jmp short go ;Код EB dd
```

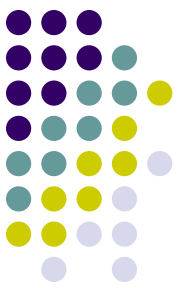
```
...
```

```
go:  ...
```

```
code ends
```

Метка должна присутствовать в том же программном сегменте, при этом помеченная команда может находиться как до, так и после команды JMP. Достоинство команды короткого перехода заключается в том, что она занимает лишь 2 байт памяти.

Разновидности безусловных переходов



2) *прямой ближний (внутрисегментный) переход:*

```
code segment
```

```
...
```

```
jmp [near ptr] go ;Код EB dddd
```

```
...
```

```
go: ...
```

```
code ends
```

Под смещение к точке перехода отводится целое слово. Это дает возможность осуществить переход в любую точку 64-кбайтного сегмента.

При вычислении адреса точки перехода смещение следует считать числом без знака, но при этом учитывать явление *оборачивания*.

Оборачивание в прямых ближних переходах



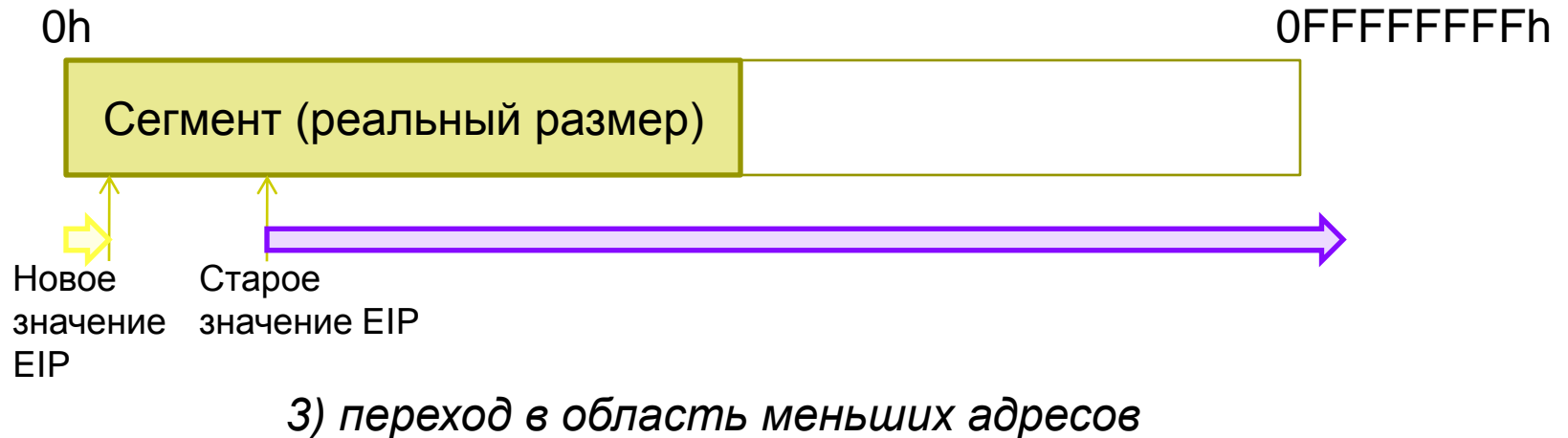
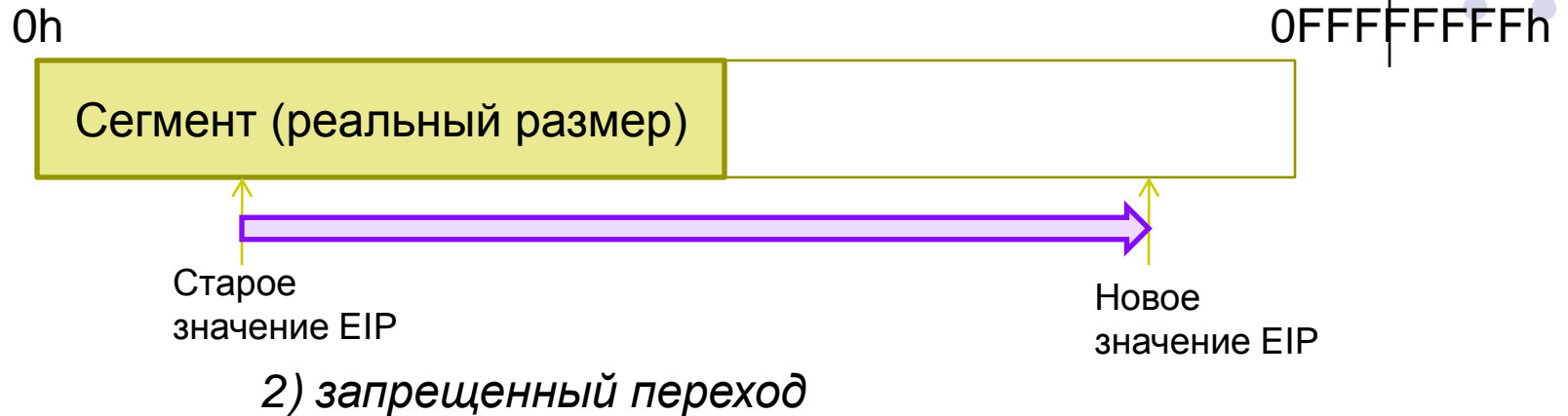
Адрес в командах ближнего перехода интерпретируется как беззнаковое смещение относительно адреса команды. Однако, если при сложении достигается максимальный размер сегмента, выполняется переход в начало сегмента – явление оборачивания, суть которого можно кратко выразить такими соотношениями:

$$\text{FFFFh} + 0001\text{h} = 0000\text{h} \quad 0000\text{h} - 0001\text{h} = \text{FFFFh}$$



1) переход в область БОльших адресов

Оборачивание в прямых ближних переходах (продолжение)



Разновидности безусловных переходов



3) *прямой дальний (межсегментный) переход* –
в другой сегмент команд

```
code1 segment
```

```
    assume CS:code1    ;Сообщим транслятору,  
    ...                ;что это сегмент команд  
    jmp far ptr go      ;Код EA dddd ssss
```

```
    ...
```

```
code1 ends
```

```
code2 segment
```

```
    assume CS:code2
```

```
    ...
```

```
go: ...
```

```
    ...
```

```
code2 ends
```

Метка go находится в другом сегменте команд.

Разновидности безусловных переходов



4) косвенный ближний (внутрисегментный) переход
`code segment`

...

`jmp DS:go_addr` ; Код FF 26 dddd

...

`go:` ... ; Точка перехода

`code ends`

`data segment`

...

`go_addr dw go` ; Адрес перехода (слово)

...

`data ends`

Точка перехода `go` может находиться в любом месте сегмента команд.

Разновидности безусловных переходов



5) *Косвенный дальний (межсегментный) переход*

```
code1 segment
```

```
    assume CS:code1,DS:data
```

```
    ...
```

```
    jmp DS:go_addr ;Код FF 2E dddd
```

```
    ...
```

```
code1 ends
```

```
code2 segment
```

```
    assume CS:code2
```

```
    ...
```

```
go: ...                ;Точка перехода в другом  
                        ;сегменте команд
```

```
    ...
```

```
code2 ends
```

Разновидности безусловных переходов



5) Косвенный дальний (межсегментный) переход

```
data segment
```

```
...
```

```
go_addr dd go          ; Двухсловный адрес точки  
                        ; перехода
```

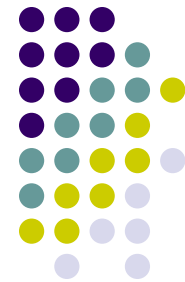
```
...
```

```
data ends
```

Точка перехода `go` находится в другом сегменте команд этой двухсегментной программы.

Ячейка `go_addr` объявляется директивой `dd` и содержит в первом слове содержится смещение `go` в сегменте команд `code1`, во втором слове сегментный адрес `code1`.

Условные переходы



Формат команды:

Jxxx адрес_перехода

Решение о переходе принимается в зависимости от выполнения определенных условий:

- пустоты регистра CX/ECX;
- состояния флагов ZF, OF, CF, SF;
- результатов сравнения целых чисел со знаком;
- результатов сравнения беззнаковых целых чисел.

Команды условного перехода относятся к ближнему прямому типу перехода (для 16-битного режима – к короткому прямому типу).

Команды, анализирующие результаты сравнения



<i>Код операции</i>	<i>Проверяемое условие</i>	<i>Флаги</i>	<i>Тип операндов</i>
JE	операнд1 == операнд2	ZF=1	любые
JNE	операнд1 != операнд2	ZF=0	
JG / JNLE	операнд1 > операнд2	ZF=0 и SF=OF	знаковые
JGE / JNL	операнд1 >= операнд2	SF=OF	
JL / JNGE	операнд1 < операнд2	SF≠OF	
JLE / JNG	операнд1 <= операнд2	ZF=1 и SF≠OF	
JA / JNBE	операнд1 > операнд2	CF=0 и ZF=0	беззнаковые
JAE / JNB	операнд1 >= операнд2	CF=0	
JB / JNAE	операнд1 < операнд2	CF=1	
JBE / JNA	операнд1 <= операнд2	CF=1 и ZF=1	

Команды, анализирующие содержимое флагов



<i>Флаг</i>	<i>Проверка на 1</i>	<i>Проверка на 0</i>
Флаг переноса CF	JC	JNC
Флаг переполнения OF	JO	JNO
Флаг нуля ZF	JZ (JE)	JNZ (JNE)
Флаг четности PF	JP	JNP
Флаг знака SF	JS (JL)	JNS (JNL)

Команды, анализирующие содержимое регистра CX/ECX

<i>Состояние регистра</i>	<i>Команда</i>
CX == 0	JCXZ
ECX == 0	JECXZ

Пример команд условного перехода



```
// найти минимум из 3-х чисел
#include <iostream>
using namespace std;
void main() {
    int a, b, c, min;
    cout << "Enter three integer numbers:\n";
    cin >> a >> b >> c;
    _asm {
        mov     eax, a
        cmp     eax, b
        jle     m1
        mov     eax, b
m1:  cmp     eax, c
        jle     m2
        mov     eax, c
m2:  mov     min, eax    }
    cout << "min is: " << min << "\n"; }
```



Команды условной пересылки

- **CMOVxxx оп1, оп2**

Эти команды появились в процессорах PentiumPro и Pentium II. Они копируют содержимое источника в приемник, если удовлетворяется то или иное условие. Коды условий аналогичны кодам в командах условного перехода.

Примеры команд:

CMOVC – переслать, если установлен флаг CF;

CMOVL – переслать, если после предыдущего сравнения знаковых чисел первый операнд меньше или равен второму.

Пример использования команд условной пересылки



```
#include <iostream>
using namespace std;
int main() {
    int a, b, c, min;
    cout << "Enter three integer numbers:\n";
    cin >> a >> b >> c;
    _asm {
        mov     eax, a
        cmp     eax, b
        cmovg   eax, b
        cmp     eax, c
        cmovg   eax, c
        mov     min, eax
    }
    cout << "min is: " << min << "\n";
    return 0;
}
```

Организация циклов



while (условие)

команды тела цикла

В Ассемблере организован следующим образом:

метка:

CMR оп1, оп2

Jxx метка_выхода

; команды тела цикла

JMP метка

метка_выхода: ...

Организация циклов



do

команды тела цикла

while (условие)

В Ассемблере организован следующим образом:

метка :

; команды тела цикла

CMR оп1, оп2

Jxx метка

Организация цикла со счетчиком



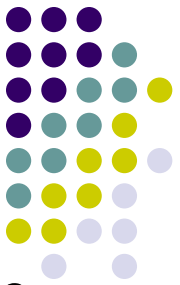
- **LOOP адрес**

Команда работает следующим образом:

- регистр CX (ECX в 32-разрядном режиме) уменьшается на единицу;
- если новое значение этого регистра не равно нулю, выполняется переход по указанному адресу, в противном случае выполняется следующая команда.

Команда LOOP относится к ближнему прямому типу перехода (для 16-битного режима – к короткому прямому типу).

Использование команды LOOP



Простейший цикл, который должен выполняться N раз, можно записать следующим образом:

```
        mov     ecx, N
cycle1:
;   тело цикла
        loop    cycle1
```

Более надежный вариант , который сработает и при N=0:

```
        mov     ecx, N
        jecxz   end_cycle1
cycle1:
;   тело цикла
        loop    cycle1
end_cycle1:
```


Команды LOOPE и LOOPNE



Эти команды позволяют при завершении итерации дополнительно анализировать содержимое флага ZF.

Первая из этих команд запускает следующую итерацию, когда $ZF=1$, а вторая – в случае $ZF=0$.

Пример

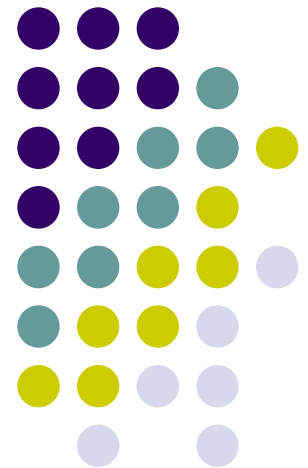


Найти сумму $1 + 2 + 3 + \dots + x$

```
void main() {  
    int x=15, sum;  
    _asm {  
        mov     eax, 0        // сумма  
        mov     ecx, x  
beg:  
        add     eax, ecx  
        loop    beg  
        mov     sum, eax  
    }  
    cout <<  sum <<  "\n";  
}
```

Тема 6

Команды модификации данных



Команды умножения



- **MUL оп1**
умножение беззнаковых чисел
- **IMUL оп1**
умножение чисел со знаком

Второй сомножитель и результат умножения находятся в регистрах, в зависимости от длины операнда:

Длина операнда	Второй операнд	Результат
байт	AL	AX
слово	AX	DX, AX
двойное слово	EAX	EDX, EAX

Если размер результата превышает размер множителей, устанавливаются флаги CF и OF

Пример работы команд умножения



Пусть нам необходимо определить, для какого максимального k величина A^k будет помещаться в СЛОВО:

```
short A;
int res=0;
cout << "Enter integer number:\n";
cin >> A;
_asm {
    mov     cx, 1
    mov     ax, A
    mov     bx, A
m1:    imul  bx
    jo      ex
    inc     cx
    jmp     m1
ex:    mov     word ptr res, ax
}
cout << "result is: " << res << "\n";
```

Команды целочисленного деления



- **DIV op1**

деление беззнаковых чисел (операнд – делитель)

- **IDIV op1**

деление чисел со знаком (операнд – делитель)

Длина делителя	Делимое	Частное	Остаток
байт	AX	AL	AH
слово	DX, AX	AX	DX
двойное слово	EDX, EAX	EAX	EDX

Команды целочисленного деления (продолжение)



Флаги не определены. Если частное не помещается в выделенное место, возникает прерывание с номером 0 («деление на ноль»).

Пример возникновения прерывания:

```
mov    ax, 10000
mov    bl, 10
div    bl      ; integer overflow
```

Должно быть:

```
xor    dx, dx      ; нужно добавить
mov    ax, 10000
xor    bx, bx      ; нужно добавить
mov    bl, 10
div    bx
```

Команды деления (пример)



Задача: поместить в строку S длиной 5 байт десятичное представление числа A

```
int A;  
char *S = new char[5];  
short res=0;  
cout << "Enter integer number:\n";  
cin >> A;  
_asm  
{  
    mov     ecx, 5  
    mov     ebx, S
```


Команды деления (продолжение примера)



```
m1:    mov    byte ptr [ebx], ' '  
        inc    ebx  
        loop  m1  
        dec    ebx  
        mov    eax, A  
        mov    esi, 10  
m2:    xor     edx, edx  
        div    esi  
        add    dl, '0'  
        mov    [ebx], dl  
        dec    ebx  
        cmp    eax, 0  
        jne    m2  
  
}  
S[5]=0;  
cout << "result is: " <<S << "\n";
```

Команды для работы с битами



- **AND оп1, оп2**

- **OR оп1, оп2**

- **XOR оп1, оп2**

выполняет соответствующие логические операции над битами оп2 и оп1. Результат помещается в оп1.

- **NOT оп1**

инвертирует биты оп1. Результат помещается в оп1.

- **TEST оп1, оп2**

вычисляет оп1 AND оп2, но не сохраняет результат (подобно CMP).

Все команды устанавливают флаг ZF, если результат будет содержать только нули.

Примеры работы побитовых команд



Проверить, установлен ли третий бит справа в регистре AX:

```
test ax, 100b
```

```
je    m1    ; переход, если он равен нулю
```

Проверить несколько бит:

```
test ax, 1101b
```

```
je    m1    ; переход, если все они равны нулю
```

Установить третий бит справа в регистре AX:

```
or    ax, 100b
```

Инвертировать третий бит справа в регистре AX:

```
xor    ax, 100b
```

Сбросить третий бит справа в регистре AX:

```
and    ax, not (100b)
```

Команды сдвига



Формат

коп операнд, счетчик_сдвигов

Счетчик_сдвигов может задаваться двумя способами:

- *статически* — непосредственно во втором операнде;
- *динамически* — в регистре CL перед выполнением команды сдвига.

По принципу действия команды сдвига можно разделить на два типа:

- команды ***линейного сдвига***;
- команды ***циклического сдвига***.

Линейный сдвиг



Алгоритм:

1. Очередной «выдвигаемый» бит устанавливает флаг CF (сдвигается во флаг CF).
2. Бит, появившийся с другого конца операнда, имеет значение 0.
3. При сдвиге очередного бита он переходит во флаг CF, при этом значение предыдущего сдвинутого бита *теряется*.

Команды линейного сдвига делятся на два подтипа:

- команды **логического** линейного сдвига;
- команды **арифметического** линейного сдвига.

Команды линейного сдвига



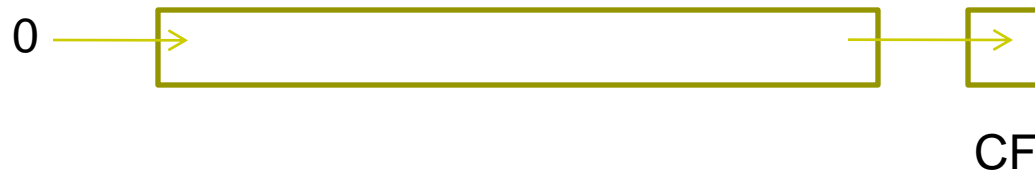
- **Линейный/арифметический сдвиг ВЛЕВО (Shift Logical /Arithmetic Left)**

SHL (SAL) оп1, число бит



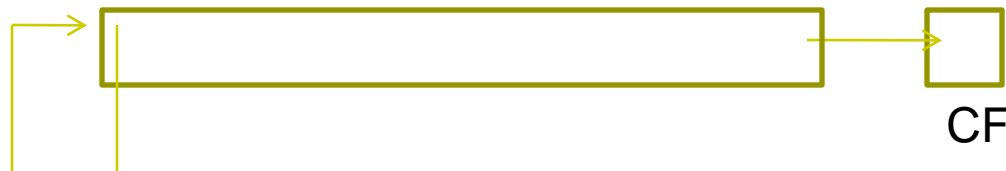
- **Линейный сдвиг ВПРАВО (Shift Logical Right)**

SHR оп1, число бит



- **Арифметический сдвиг ВПРАВО (Shift Arithmetic Right)**

SAR оп1, число бит



Команды линейного сдвига



Команды арифметического сдвига позволяют выполнить «быстрое» умножение и деление операнда на степени двойки:

75 01001011

150 10010110

Второе число является сдвинутым **влево** на один разряд первым числом.

Аналогичная ситуация — с операцией деления на степени двойки 2, 4, 8 и т. д. — сдвиг **вправо**. При делении пополам нечетных чисел результатом становятся значения, округленные в меньшую сторону: 5-2, 7 -3 и флаг CF=1.

Циклический сдвиг



К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых битов.

Есть два типа команд циклического сдвига:

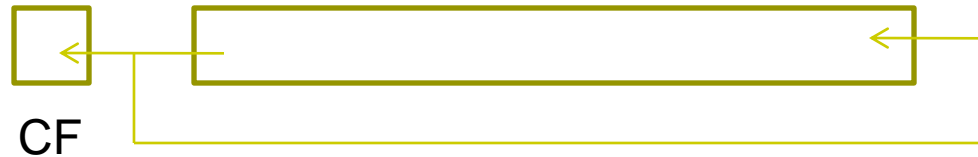
- команды **простого** циклического сдвига;
- команды циклического сдвига **через флаг переноса** CF.

Команды циклического сдвига



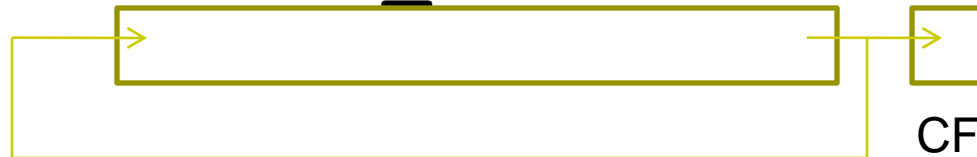
- Циклический сдвиг влево (Rotate Left)

ROL оп1, число_бит



- Циклический сдвиг вправо (Rotate Right)

ROR оп1, число_бит



Пример:

обменять содержимое двух половинок регистра EAX:

```
mov eax, ffff0000h
```

```
mov cl, 16 ; динамическое задание
```

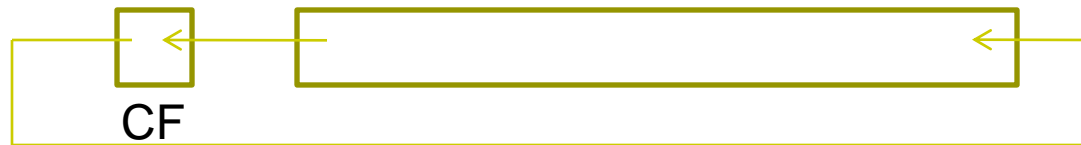
```
rol eax, cl
```

Команды циклического сдвига



- Циклический сдвиг влево через перенос (Rotate through Carry Left)

RCL оп1, число_бит



- Циклический сдвиг вправо через перенос (Rotate through Carry Right)

RCR оп1, число_бит



Команды циклического сдвига (пример)



Пример: переписать в регистр ВХ старшую половину регистра ЕАХ с одновременным ее обнулением в регистре ЕАХ:

```
mov cx,16          ;кол-во сдвигов для еах
```

```
ml:
```

```
clc                ;сброс флага CF в 0
```

```
rcl eax,1          ;сдвиг крайнего левого бита  
                  ; из еах в CF
```

```
rcl bx,1           ;перемещение бита из CF  
                  ;справа в bx
```

```
loop ml            ;цикл 16 раз
```

```
rol eax,16         ;восстановить правую часть еах
```

Команды сдвига (пример)



Задача: определить количество единиц в двоичном представлении числа, хранящегося в регистре AX

```
short A, res=0;
cout << "Enter integer number:\n";
cin >> A;
_asm {
    mov     ax, A
    xor     dx, dx
    mov     cx, 16
m1:        shr     ax, 1
           jnc     m2
           inc     dx
m2:        loop    m1    ; число единиц хранится в dx
           mov     res, dx
}
cout << "result is: " << res << "\n";
```



Спасибо за
внимание!