Съкратен справочник за асемблерните инструкции за архитектурата x86

С примери с вграден асемблерен код за MASM

Кирил Иванов

Юни 2019 година

Този кратък справочник е предназначен за практически упражнения при начално запознаване с програмирането на асемблерен език за архитектурата x86.

Описано е действието на инструкции, валидни в 32-разрядни потребителски приложения, т. е. в код, изпълняван от процесора в 32-разряден режим и с ограничени привилегии. Някои от тези инструкции са забранени, а някои имат малки промени в действието, в 64-разряден режим. Извън този текст остават 16-разрядният и виртуалният режими на работа на процесора, тъй като те са излезли от масова употреба.

Описани са само инструкциите за управление на изчислителния процес, за целочислена аритметика и за работа с код с плаваща запетая. При това са пропуснати малък брой детайли (например някои особености в работата с флагове). Избраният обхват е достатъчен за програмирането на асемблерно ниво на повечето алгоритми, използвани в традиционното обучение при въведение в програмирането.

Примерите и поясненията с асемблерен или подобен на него език са ориентирани към синтаксиса на MASM. Примерите с вграждане на асемблерен код в текст на езика C++ са ориентирани към синтаксиса на Visual C++.

Използвано е необичайно оцветяване за да се повиши нагледността на описанията (в бизнес програмирането се спазва качествено различно форматиране). Всеки текст с този фон е препратка, по която може да се премине с натискане на левия бутон на мишката.

Съдържание

Описания на инструкции по азбучен ред

→ Съдържание

aam – записва в регистъра **ax** стойност, получена от регистъра **al**, а именно:
в **al** записва остатъка от деленето на **al** с 10;
в **ah** записва цялата част от целочисленото делене на **al** с 10.
Тази инструкция предизвиква грешка при използване в 64-разряден код.

Например кодът

```
char n = 237, H, L;
__asm {
  push eax
    mov al, n
    aam
    mov H, ah
    mov L, al
  pop eax
}
cout << "H = " << (int)H << endl;
cout << "L = " << (int)L << endl;
```

извежда на екрана

```
H = 23
L = 7
```

adc op1, op2 – записва в ***op1*** сумата на ***op1***, ***op2*** и флага **CF**. Работи и с код без знак, и с допълнителен код.

Например кодът

```
int n = 800;
__asm {
  push edi
    mov edi, 30
    stc
    adc n, edi
  pop edi
}
cout << "n = " << n << endl;
```

извежда на екрана

```
n = 831
```

add op1, op2 – записва в ***op1*** сумата на ***op1*** и ***op2***. Работи и с код без знак, и с допълнителен код.

and op1, op2 – записва в ***op1*** поразрядната конюнкция (И; AND) на ***op1*** и ***op2***.

Например кодът

```
int x = 0xf0c030, y = 0xffff0f, res;
__asm {
  push eax
    mov eax, x
    and eax, y
    mov res, eax
  pop eax
}
```

```
cout << "res = " << hex << res << endl;
```

извежда на екрана

```
res = f0c000
```

andn op1, op2, op3 – записва в ***op1*** поразрядната конюнкция (поразрядното И, AND) на ***op2*** и ***op3***.

bsf op1, op2 – търси в ***op2*** единичен разряд *отдясно наляво* (от младшия към старшия разряд) и може да завърши по два начина:
Първо, ако ***op2*** е нула (няма единица в ***op2***), записва *единица* във флага **ZF** (за нулев резултат) и ***op1*** е *неопределено*.
Второ, ако в ***op2*** има единичен разряд, записва в ***op1*** номера (броенето започва от нула за младшия разряд) на най-дясната (младшата) единица от ***op2*** и *нулира* флага **ZF** (за нулев резултат).
Операндът ***op1*** трябва да бъде регистър.

Например кодът

```
int n = 0x30;
char z;
__asm {
  push esi
    mov esi, -2348
    bsf esi, n
    mov n, esi
    setz z
  pop esi
}
cout << "n = " << n << endl;
cout << "z = " << (int)z << endl;
```

извежда на екрана

```
n = 4
z = 0
```

Съответно кодът

```
int n;
char z;
__asm {
  push esi
  push ecx
    xor ecx, ecx
    mov esi, -2348
    bsf esi, ecx
    mov n, esi
    setz z
  pop ecx
  pop esi
}
cout << "n = " << n << endl;
cout << "z = " << (int)z << endl;
```

извежда на екрана

```
n = -2348
z = 1
```

bsr op1, op2 – търси в ***op2*** единичен разряд *отляво надясно* (от старшия към младшия разряд) и може да завърши по два начина:
Първо, ако ***op2*** е нула (няма единичен разряд в ***op2***), записва *единица* във флага **ZF** (за нулев резултат) и ***op1*** е *неопределено*.
Второ, ако в ***op2*** има единичен разряд, записва в ***op1*** номера (броенето започва от нула за младшия разряд) на най-лявата (старшата) единица и *нулира* флага **ZF** (за нулев резултат).
Операндът ***op1*** трябва да бъде регистър.

Например кодът

```
int n = 0x090;
char z;
__asm {
  push esi
    bsr esi, n
    mov n, esi
    setz z
  pop esi
}
cout << "n = " << n << endl;
cout << "z = " << (int)z << endl;
```

извежда на екрана

```
n = 7
z = 0
```

Съответно кодът

```
int n;
char z;
__asm {
  push esi
  push ecx
    xor ecx, ecx
    mov esi, -2348
    bsr esi, ecx
    mov n, esi
    setz z
  pop ecx
  pop esi
}
cout << "n = " << n << endl;
cout << "z = " << (int)z << endl;
```

извежда на екрана

```
n = -2348
z = 1
```

bswap op – пренарежда байтовете на ***op*** в точно обратен ред.
op трябва да бъде 32-разряден или 64-разряден регистър.

Например кодът

```
int x = 0x12345678;
cout << "x = " << hex << x << endl;
```

```
__asm {
  push eax
    mov eax, x
    bswap eax
    mov x, eax
  pop eax
}
cout << "x = " << hex << x << endl;
```

извежда на екрана

```
x = 12345678
x = 78563412
```

bt op1, op2 – записва във флага **CF** разряда от ***op1***, който има номер ***op2*** (номерирането започва от 0 за младшия разряд).

btc op1, op2 – записва във флага **CF** разряда от ***op1***, който има номер ***op2*** (номерирането започва от 0 за младшия разряд), и след това инвертира същия (записания) разряд в ***op1***.

Например кодът

```
int x = 0xf130;
short CFlag = 0;
__asm {
  btc x, 8
  setc byte ptr CFlag
}
cout << "x = " << hex << x << endl;
cout << "CFlag = " << CFlag << endl;
```

извежда на екрана

```
x = f030
CFlag = 1
```

btr op1, op2 – записва във флага **CF** разряда от ***op1***, който има номер ***op2*** (номерирането започва от 0 за младшия разряд), и след това нулира в ***op1*** същия (записания) разряд.

Например кодът

```
int number = 0xffff;
__asm {
  push ecx
      mov ecx, 7
    Cycle : btr number, ecx
            loop Cycle
  pop ecx
}
cout << "number = " << hex << number << endl;
```

извежда на екрана

```
number = ff01
```

bts op1, op2 – записва във флага **CF** разряда от ***op1***, който има номер ***op2*** (номерирането започва от 0 за младшия разряд), и после записва 1 в същия разряд.

Например кодът

```
int number = 0xf000;
```

```
__asm {
  push ecx
      mov ecx, 7
    Cycle : bts number, ecx
            loop Cycle
  pop ecx
}
cout << "number = " << hex << number << endl;
```

извежда на екрана

```
number = f0fe
```

call adr – записва в стека адреса за връщане от подпрограма (т. е. адреса на инструкцията, намираща се точно след ***call adr***) и предава управлението към адреса ***adr***.
Записваният адрес може да бъде в различни варианти. Обикновено разрядността и форматът му се определят автоматично от компилатора, но може да бъдат явно указани.

Например кодът

```
int number = -1234;
__asm {
  push eax
      jmp After
    SUBprog : inc dword ptr [eax]
              ret
    After :
      mov number, 700
      lea eax, number
      call SUBprog
  pop eax
}
cout << "number = " << number << endl;
```

извежда на екрана

```
number = 701
```

cbw – преобразува 8-разрядния допълнителен код в регистъра **al** в 16-разряден допълнителен код в регистъра **ax**.

cdq – преобразува допълнителния код от регистъра **eax** в **edx:eax**, т. е. в 64-разряден допълнителен код с младша половина в регистъра **eax** и старша половина в регистъра **edx**.

clc – нулира флага **CF** (carry flag; флаг за пренос наляво от разрядната решетка).

cld – нулира флага **DF** (direction flag; флаг за направление на обхождане на операндите при „низови“ инструкции, т. е. за избор на автоувеличение или автонамаление).

cmc – инвертира (complement) флага **CF** (carry flag; флаг за пренос наляво от разрядната решетка)

cmov... op1, op2 – премества в зависимост от условие (conditional move).
op1 трябва да бъде регистър.
op2 може да бъде регистър или операнд в оперативната памет.
Многоточието може да се замества със същите букви, които може да се пишат в инструкцията за условен преход ***j...*** след буквата ***j*** (описани са към инструкцията ***j...***).
Записва ***op2*** в ***op1***, точно когато е изпълнено условието, назовано с многоточието.

Например кодът

```
int x = 2, y = 58, z = -345;
cout << "z = " << z << endl;
```

```
__asm {
  push esi
    mov esi, x
    cmp esi, y
    cmovNGE esi, y // преместване, когато esi е по-малко от y
    mov z, esi
  pop esi
}
cout << "z = " << z << endl;
```

извежда на екрана

```
z = -345
z = 58
```

cmp op1, op2 – изчислява разликата ***op1-op2*** без да съхранява никъде резултата, обаче модифицира флаговете, точно както би ги променила инструкцията ***sub op1, op2*** (обикновено ***cmp*** предшествува инструкции за условен преход според условие някаква релация между ***op1*** и ***op2***).
op1 трябва да бъде регистър или операнд в оперативната памет.
op2 може да бъде регистър, операнд в оперативната памет или число (непосредствен операнд).
(***cmp*** е използвана в примера към инструкцията ***cmov...***)

cmpsb – изчислява разликата **byte ptr ds:[esi] - byte ptr es:[edi]** (или в 64-разряден режим съответно **byte ptr [rsi] - byte ptr [rdi]**), т. е. разликата на 8-разрядни операнди, без да съхранява получения резултат, но като модифицира флаговете, точно както биха ги променили инструкциите ***cmp*** или ***sub*** със същите умаляемо и умалител, и след това:
когато флагът **DF=0**, *прибавя* единица към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с единица **esi** и **edi** (или съответно **rsi** и **rdi**).
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например програмата

```
#include <iostream>
using namespace std;
const int Len = 100;
char s1[Len] = "abc123#", s2[Len] = "abc123";
bool s1GREATERs2 = true;
int main() {
  __asm { // сравнява s1 и s2
    pushad
    pushf
        cld
        mov ax, ds
        mov es, ax // този и горният ред са само за 32-разряден режим
        lea edi, s1
        mov ecx, -1
        xor al, al
      repne scasb // търси края на низа s1
        add ecx, 2
        neg ecx // ecx става равно на дължината на s1
        lea esi, s1
        lea edi, s2
      repe cmpsb
        seta s1GREATERs2
    popf
    popad
```

```
  }
  cout << '\"' << s1 << "\" > \"" << s2 << "\" => "
       << (s1GREATERs2 ? "true" : "false") << endl;
  system("pause");
}
```

извежда на екрана

```
"abc123#" > "abc123" => true
```

cmpsd – изчислява разликата **dword ptr ds:[esi] - dword ptr es:[edi]** (или в 64-разряден режим съответно **dword ptr [rsi] - dword ptr [rdi]**), т. е. разликата на 32-разрядни операнди, без да съхранява получения резултат, но като модифицира флаговете, точно както биха ги променили инструкциите ***cmp*** или ***sub*** със същите умаляемо и умалител, и след това:
когато флагът **DF=0**, *прибавя* четири към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с четири **esi** и **edi** (или съответно **rsi** и **rdi**).
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например програмата

```
#include <iostream>
using namespace std;
int ar1[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 },
    ar2[] = { 10, 20, -3, -4, 50, -6, -7, -8, 90 },
    rIndex = -321;
const int Len = sizeof(ar1) / sizeof(ar1[0]);
int main() {
  __asm {
    StartNext : // записва в rIndex индекса на най-дясната
                // двойка равни елементи с равни индекси
                // или -1, когато няма такива елементи
      pushad
      pushf
          mov ax, ds
          mov es, ax // този и горният ред са само за 32-разряден режим
          std // за автонамаление
          mov ecx, Len // максимален брой повторения
          inc ecx // отчита dec ecx по-надолу
          lea esi, ar1[ecx*4-8]
          lea edi, ar2[ecx*4-8]
        repne cmpsd // търси най-дясната двойка равни елементи
          dec ecx
          mov rIndex, ecx
      popf
      popad
  }
  cout << "rIndex = " << rIndex << endl;
  __asm { // продължава търсенето, докато има намерен индекс
    cmp rIndex, -1
    jng Stop
      push eax
        mov eax, rIndex
        inc dword ptr [ar1+eax*4] // същото, като ar1[rIndex]++
      pop eax
      jmp StartNext
```

```
    Stop :
  }
  system("pause");
}
```

извежда на екрана

```
rIndex = 8
rIndex = 4
rIndex = 1
rIndex = 0
rIndex = -1
```

(Това са индексите на всички двойки равни елементи с еднакви индекси от двата масива.)

cmpsq – изчислява разликата **qword ptr ds:[esi] - qword ptr es:[edi]** (или в 64-разряден режим съответно **qword ptr [rsi] - qword ptr [rdi]**), т. е. разликата на 64-разрядни операнди, без да съхранява получения резултат, но като модифицира флаговете, точно както биха ги променили инструкциите ***cmp*** или ***sub*** със същите умаляемо и умалител, и след това:
когато флагът **DF=0**, *прибавя* осем към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с осем **esi** и **edi** (или съответно **rsi** и **rdi**).
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

cmpsw – изчислява разликата **word ptr [esi] - word ptr [edi]** (или в 64-разряден режим съответно **word ptr [rsi] - word ptr [rdi]**), т. е. разликата на 16-разрядни операнди, без да съхранява получения резултат, но като модифицира флаговете, точно както биха ги променили инструкциите ***cmp*** или ***sub*** със същите умаляемо и умалител, и след това:
когато флагът **DF=0**, *прибавя* две към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с две **esi** и **edi** (или съответно **rsi** и от).
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например програмата

```
#include <iostream>
using namespace std;
short ar1[] = { -8, -2,  1,  3,  5,  7,  9, 11 },
      ar2[] = {  0,  0,  1,  3,  5,  7, -1, -1 };
const int Len = sizeof(ar1) / sizeof(ar1[0]);
int main() {
  short count = 0;
  for(short i=0; i<Len; ++i) if(ar1[i]==ar2[i]) ++count;
  cout << "cpp: count = " << count << endl;
  count = -1;
  __asm {
    pushad
    pushf
        mov count, 0
        cld // за автоувеличение
        mov ax, ds
        mov es, ax //този и горният ред са необходими в 32-разряден режим
        lea esi, ar1
        lea edi, ar2
        mov ecx, Len // брой повторения на цикъла
```

```
        Cycle : cmpsw
                jne Next
                  inc count
        Next : loop Cycle
      popf
      popad
    }
    cout << "asm: count = " << count << endl;
    system("pause");
  }
```

извежда на екрана

```
cpp: count = 4
asm: count = 4
```

(Това е броя на двойките равни елементи с равни индекси.)

cmpxchg op1, op2 – сравнява ***op1*** с онзи от регистрите **al**, **ax**, **eax** или **rax**, който има същата разрядност, като ***op1***. При това, модифицира флаговете, точно както инструкцията ***cmp***.
След сравнението копира ***op2*** на място, зависещо от равенството или различието на сравняваните стойности, а именно:
когато ***op1*** е *равно на* съответния от регистрите **al**, **ax**, **eax** или **rax**, тогава копира ***op2*** в ***op1***;
когато ***op1*** е *различно от* съответния от регистрите **al**, **ax**, **eax** или **rax**, тогава копира ***op1*** в съответния от регистрите **al**, **ax**, **eax** или **rax**.
op1 трябва да бъде регистър.
op2 може да бъде регистър или операнд в оперативната памет, но с разрядността на ***op1***.

Например кодът

```
short n1=-11, n2=-22, Reg=n1;
for(int i=1; i<=2; ++i) {
  cout << "------\n n1 = " << n1 << "\n n2 = " << n2
       << "\n Reg = " << Reg << "\n------\n";
  __asm {
    push ax
    push si
      mov si, 6789
      mov ax, Reg
      cmpxchg n1, si // сравнява n1 с ax
      mov Reg, ax
    pop si
    pop ax
  }
  cout << "After if(Reg==n1) n1=6789;\n      else Reg=n1; :\n"
       << " n1 = " << n1 << "\n n2 = " << n2
       << "\n Reg = " << Reg << "\n------\n";
  n1=-11, n2=-22, Reg=-33;
} // for end
```

извежда на екрана

```
------
 n1 = -11
 n2 = -22
 Reg = -11
------
```

```
After if(Reg==n1) n1=6789
      else Reg=n1; :
 n1 = 6789
 n2 = -22
 Reg = -11
------
------
 n1 = -11
 n2 = -22
 Reg = -33
------
After if(Reg==n1) n1=6789
      else Reg=n1; :
 n1 = -11
 n2 = -22
 Reg = -11
------
```

cwd – преобразува допълнителния код от регистъра **ax** в **dx:ax**, т. е. в 32-разряден допълнителен код с младша половина в регистъра **ax** и старша половина в регистъра **dx** .

cwde – преобразува 16-разрядния допълнителен код от регистъра **ax** в 32-разряден допълнителен код в регистъра **eax.**

dec op – намалява с единица стойността на ***op***. *Не променя* флага **CF** (за пренос наляво от разрядната решетка; carry flag), обаче променя **OF** (за препълване при работа с допълнителен код; overflow flag), **SF** (за знаков, старши разряд от резултата; sign flag), **ZF** (за нулев резултат; zero flag), **AF** (за „полупренос", наляво от разряд номер 3; auxiliary carry flag) и **PF** (за четност на броя на единиците в младшия байт на резултата; parity flag).

div op – дели 2n-разряден код без знак на делимо ***op*** и получава n-разрядни цяло частно и остатък също в код без знак.
Операндът ***op*** трябва да бъде в регистър или в паметта.
Вариантите са:

разрядност на ***op***	делимо	цяло частно	остатък
8	**ax**	**al**	**ah**
16	**dx:ax**, т. е. 32-разряден код без знак, старшите 16 разряда са в **dx**, а младшите 16 разряда са в **ax**	**ax**	**dx**
32	**edx:eax**, т. е. 64-разряден код без знак, старшите 32 разряда са в **edx**, а младшите 32 разряда са в **eax**	**eax**	**edx**
64	**rdx:rax**, т. е. 128-разряден код без знак, старшите 64 разряда са в **rdx**, а младшите 64 разряда са в **rax**	**rax**	**rdx**

Когато цялото частно не се побира в предвидения за него регистър, предизвиква прекъсване поради грешка при делене. (В този случай някои компилатори съобщават „делене с нула“.)

Например кодът

```
unsigned int dividend = 23, divisor = 5, quotient, remainder;
__asm {
  push edx
  push eax
    mov eax, dividend
    xor edx, edx // за кодове без знак edx:eax става равно на eax
    div divisor
    mov quotient, eax
    mov remainder, edx
  pop eax
  pop edx
}
cout << dividend << " = " << quotient << " * " << divisor
     << " + " << remainder << endl;
```

извежда на екрана

```
23 = 4 * 5 + 3
```

f2xm1 – записва в ***st(0)*** стойността на израза $2^{st(0)} - 1$.
st(0) трябва да бъде от -1.0 до +1.0, иначе резултатът е неопределен.

fabs – записва нула в знаковия разряд на ***st(0)***.

Например кодът

```
double n = -1.125;
__asm {
  fld n
  fabs
  fstp n
}
cout << "n = " << n << endl;
```

извежда на екрана

```
n = 1.125
```

fadd – еквивалентна на ***faddp st(1), st(0)***.

fadd op – замества върха ***st(0)*** на регистровия стек със сумата ***st(0)+op***, където ***op*** е *код с плаваща запетая* в паметта.

Например кодът

```
double n1 = -1.125, n2 = -200.3, s;
__asm {
  fld n1
  fadd n2
  fstp s
}
cout << n1 << " + " << n2 << " = " << s << endl;
```

извежда на екрана

```
-1.125 + -200.3 = -201.425
```

fadd st(i), st(j) – замества ***st(i)*** със сумата ***st(i)+st(j)***.
Единият от индексите ***i*** и ***j*** трябва да бъде ***0***.

faddp st(i), st(0) – замества ***st(i)*** със сумата ***st(i)+st(0)*** и премахва върха ***st(0)*** на регистровия стек в устройството за изчисления с плаваща запетая.

fchs – инвертира знаковия разряд на ***st(0)***.

Например кодът

```
float f = -1.625;
__asm {
  fld f
  fchs
  fstp f
}
cout << f << endl;
__asm {
  fld f
  fchs
  fstp f
}
cout << f << endl;
```

извежда на екрана

```
1.625
-1.625
```

fcom – изчислява разликата ***st(0)-st(1)*** без да я съхранява, но модифицира флагове ***C3***, ***C2***, ***C1*** и ***C0*** в думата на състоянието в устройството за изчисления с плаваща запетая.
Съответствието между флаговете и релацията, свързваща ***st(0)*** и ***st(1)*** е следното:

флагове в думата на състоянието на устройството за работа с плаваща запетая				релация
C3	***C2***	***C1***	***C0***	
0	0	0	0	***st(0)>st(1)***
0	0	0	1	***st(0)<st(1)***
1	0	0	0	***st(0)=st(1)***
1	1	0	1	несравними

В тази архитектура отсъствуват инструкции, които могат да разклоняват алгоритъма, според стойностите на флагове ***C3***, ***C2***, ***C1*** и ***C0***. За улесняване на такова разклонение може да се използват инструкциите ***fcomi*** или ***fcomip***. Друга обичайна практиката е чрез ***fstsw*** да се копира думата на състоянието в регистъра **ax** (или в паметта) и след това или да се проверяват разрядите на **ax** (или на паметта), или да се прехвърля стойността на **ax** в **RFLAGS** и да се работи с флаговете на **RFLAGS**.

fcom op – аналогично на ***fcom***, изчислява разликата на ***st(0)*** и ***op*** без да я съхранява, но модифицира флагове ***C3***, ***C2***, ***C1*** и ***C0*** в думата на състоянието в устройството за изчисления с плаваща запетая.
op трябва да бъде *код с плаваща запетая* и или да бъде регистър, или да има адрес в паметта.

fcomi st(0), st(i) – изчислява разликата на ***st(0)*** и ***st(i)*** без да я съхранява, но модифицира флагове ***ZF***, ***PF*** и ***CF*** в регистъра **FLAGS**.

Съответствието между флаговете и релацията, свързваща ***st(0)*** и ***st(i)*** е следното:

флагове в регистъра **FLAGS**			релация
PF	***ZF***	***CF***	
0	0	0	***st(0)>st(i)***
0	0	1	***st(0)<st(i)***
0	1	0	***st(0)=st(i)***
1	1	1	несравними

fcomip st(0), st(i) – прави същото, каквото и ***fcomi st(0), st(i)***, а след това премахва върха ***st(0)*** на регистровия стек.

fcomp – изпълнява същото, каквото и ***fcom***, а след това премахва върха на регистровия стек в устройството за работа с код с плаваща запетая.

fcomp op – аналогично на ***fcom***, изчислява разликата на ***st(0)*** и ***op*** без да я съхранява, но модифицира флагове ***C3***, ***C2***, ***C1*** и ***C0*** в думата на състоянието в устройството за изчисления с плаваща запетая, а след това премахва върха ***st(0)*** на регистровия стек.
op трябва да бъде *код с плаваща запетая* и или да бъде регистър, или да има адрес в паметта.

fcompp – изпълнява същото, каквото и ***fcom***, а след това *два пъти* премахва върха на регистровия стек в устройството за работа с код с плаваща запетая.

fcos – записва в ***st(0)*** косинуса от ***st(0)***, интерпретиран като радиани.

fdecstp – без да променя съдържанието на регистровия стек, намалява с единица номера на регистъра връх на регистровия стек. Когато номерът преди инструкцията е бил 0, той става 7. По този начин предишният връх ***st(0)*** след инструкцията става ***st(1)***.

fdiv op – дели ***st(0)*** на ***op*** и записва резултата в ***st(0)***.
op трябва да бъде *код с плаваща запетая* в паметта.

fdiv st(i), st(j) – дели ***st(i)*** на ***st(j)*** и записва резултата в ***st(i)***.
Единият от индексите ***i*** и ***j*** трябва да бъде ***0***.

fdivp – дели ***st(1)*** на ***st(0)*** и записва резултата в ***st(1)***, а след това премахва върха ***st(0)*** на регистровия стек.

fdivp st(i), st(0) – дели ***st(i)*** на ***st(0)*** и записва резултата в ***st(i)***, а след това премахва върха ***st(0)*** на регистровия стек.

fdivr op – дели ***op*** на ***st(0)*** и записва резултата в ***st(0)***.
op трябва да бъде *код с плаваща запетая* в паметта.

fdivr st(i), st(j) – дели ***st(j)*** на ***st(i)*** и записва резултата в ***st(i)***.
Единият от индексите ***i*** и ***j*** трябва да бъде ***0***.

fidivr op – дели ***op*** на ***st(0)*** и записва резултата в ***st(0)***.
op трябва да бъде *допълнителен код* в паметта.

fdivrp – дели ***st(0)*** на ***st(1)*** и записва резултата в ***st(1)***, а след това премахва върха ***st(0)*** на регистровия стек.

fdivrp st(i), st(0) – дели ***st(0)*** на ***st(i)*** и записва резултата в ***st(i)***, а след това премахва върха ***st(0)*** на регистровия стек.

fiadd op – замества върха ***st(0)*** на регистровия стек със сумата ***st(0)+op***, където ***op*** е *допълнителен код* в паметта.

Например кодът

```
double d = 0.5, res;
long long L = 14;
__asm {
  fld d
  fiadd L
  fstp res
}
cout << d << " + " << L << " = " << res << endl;
```

извежда на екрана

```
0.5 + 14 = 14.5
```

ficom op – изчислява разликата на ***st(0)*** и ***op*** без да я съхранява, но модифицира флагове ***C3***, ***C2***, ***C1*** и ***C0*** в думата на състоянието в блока за изчисления с плаваща запетая.
op трябва да бъде *допълнителен код* в паметта.
Съответствието между флаговете и релацията, свързваща ***st(0)*** и ***st(1)*** е следното:

флагове в думата на състоянието на устройството за работа с плаваща запетая				релация
C3	***C2***	***C1***	***C0***	
0	0	0	0	***st(0)>st(1)***
0	0	0	1	***st(0)<st(1)***
1	0	0	0	***st(0)=st(1)***
1	1	0	1	несравними

ficomp op – прави същото, каквото и ***ficom op***, а след това премахва върха ***st(0)*** на регистровия стек.

fidiv op – дели ***st(0)*** на ***op*** и записва резултата в ***st(0)***.
op трябва да бъде *допълнителен код* в паметта.

fild op – включва ***op*** като нов връх на регистровия стек.
op трябва да бъде *допълнителен код* в паметта.

fimul op – умножава ***st(0)*** с ***op*** и записва произведението в ***st(0)***.
op трябва да бъде *допълнителен код* в паметта.

fist op – записва ***st(0)*** в допълнителния код ***op***.

fincstp op – без да променя съдържанието на регистровия стек, увеличава с единица номера на регистъра връх на регистровия стек. Когато номерът преди инструкцията е бил 7, той става 0. По този начин предишният връх ***st(1)*** след инструкцията става ***st(0)***.

fistp op – записва ***st(0)*** в допълнителния код ***op***, а след това премахва върха ***st(0)*** на регистровия стек. (Начинът на закръгляне се определя от ***RC***.)

fisttp op – отсича дробната част от ***st(0)*** и записва полученото цяло число в допълнителния код ***op***, а след това премахва върха ***st(0)*** на регистровия стек. (Начинът на закръгляне игнорира ***RC***.)

fisub op – записва в ***st(0)*** разликата ***st(0)-op***.
op трябва да бъде *допълнителен код* в паметта.

fisubr op – записва в **st(0)** разликата **op-st(0)**.
op трябва да бъде *допълнителен код* в паметта.

fld op – включва **op** като нов връх на регистровия стек.

fld1 op – включва стойността +1.0 като нов връх на регистровия стек.

fldl2e op – включва стойността $\log_2 e$ като нов връх на регистровия стек.

fldl2t op – включва стойността $\log_2 10$ като нов връх на регистровия стек.

fldlg2 op – включва стойността $\log_{10} 2$ като нов връх на регистровия стек.

fldln2 op – включва стойността $\log_e 2$ като нов връх на регистровия стек.

fldpi op – включва числото π като нов връх на регистровия стек.

fldz op – включва числото **+0.0** като нов връх на регистровия стек.

fmul op – умножава **st(0)** с **op** и записва произведението в **st(0)**.
op трябва да бъде *код с плаваща запетая* в паметта.

fmul st(i), st(j) – умножава **st(i)** с **st(j)** и записва произведението в **st(i)**.
Единият от индексите **i** и **j** трябва да бъде **0**.

fmulp – умножава **st(1)** с **st(0)** и записва произведението в **st(1)**, а след това премахва върха **st(0)** на регистровия стек.

fmulp st(i), st(0) – умножава **st(i)** с **st(0)** и записва произведението в **st(i)**, а след това премахва върха **st(0)** на регистровия стек.

fprem – записва в **st(0)** остатъка (в интерпретация за реални числа) от деленето на **st(0)** с **st(1)**.

fprem1 – както и **fprem**, записва в **st(0)** остатъка (в интерпретация за реални числа) от деленето на **st(0)** с **st(1)**, но го изчислява според стандарта IEEE Standard 754.

fptan – записва в **st(0)** тангенса от **st(0)** и включва (push) в регистровия стек числото +1.0.
st(0) трябва да бъде в радиани от 2^{-63} до 2^{+63}.

Например кодът

```
double r=3.1415/4.0, t;
__asm {
  fld r
  fptan
  fincstp
  fstp t
}
cout.precision(18);
cout << "tan( " << r << " rad ) = " << t << endl;
```

извежда на екрана (тангенса от приблизително 45 градуса)

```
tan( 0.78537500000000005 rad ) = 0.99995367427815629
```

frndint – закръгля **st(0)** до цяло число. (Начинът на закръгляне се определя от **RC**.)

fscale – записва в **st(0)** стойността на израза $st(0).2^{frndint(st(1))}$.

fsin – записва в **st(0)** синуса от **st(0)**, интерпретиран като радиани.

fsincos – изчислява синуса и косинуса от **st(0)** и записва синуса в **st(0)**, а след това включва косинуса като нов връх на регистровия стек.

fsqrt – записва в **st(0)** квадратния корен от **st(0)**.

fst op – записва **st(0)** в **op**.
op трябва да бъде *код с плаваща запетая* в регистър или в паметта.

fstp op – записва **st(0)** в **op**, а след това премахва върха **st(0)** на регистровия стек.
op трябва да бъде *код с плаваща запетая* в регистър или в паметта.

fstsw ax – записва в регистъра **ax** думата на състоянието на блока за изчисления с плаваща запетая, като преди това завършва провежданото в момента изчисление с плаваща запетая.

fsub op – записва в **st(0)** разликата **st(0)-op**.
op трябва да бъде *код с плаваща запетая* в паметта.

fsub st(i), st(j) – записва в **st(i)** разликата **st(i)-st(j)**.
Единият от индексите **i** и **j** трябва да бъде **0**.

fsubp – записва в **st(1)** разликата **st(1)-st(0)**, а след това премахва върха **st(0)** на регистровия стек.

fsubp st(i), st(0) –записва в **st(i)** разликата **st(i)-st(0)**, а след това премахва върха **st(0)** на регистровия стек.

fsubr op – записва в **st(0)** разликата **op-st(0)**.
op трябва да бъде *код с плаваща запетая* в паметта.

fsubr st(i), st(j) – записва в **st(i)** разликата **st(j)-st(i)**.
Единият от индексите **i** и **j** трябва да бъде **0**.

fsubrp – записва в **st(1)** разликата **st(0)-st(1)**, а след това премахва върха **st(0)** на регистровия стек.

fsubrp st(i), st(0) –записва в **st(i)** разликата **st(0)-st(i)**, а след това премахва върха **st(0)** на регистровия стек.

ftst – сравнява **st(0)** с нула и модифицира флагове **C3**, **C2**, **C1** и **C0** в думата на състоянието в устройството за изчисления с плаваща запетая.
Съответствието между флаговете и релацията е следното:

флагове в думата на състоянието на устройството за работа с плаваща запетая				релация
C3	**C2**	**C1**	**C0**	
0	0	0	0	**st(0)>0.0**
0	0	0	1	**st(0)<0.0**
1	0	0	0	**st(0)=0.0**
1	1	0	1	несравними

fxch – разменя стойностите на **st(0)** и **st(1)**.

fxch st(i) – разменя стойностите на **st(0)** и **st(i)**.

fyl2x – записва в **st(1)** стойността на израза $st(1).\log_2(st(0))$ и премахва върха **st(0)** на стека.

Тъй като $\log_x(y)=\frac{\log_2(y)}{\log_2(x)}$, тази инструкция може да се използва за намиране на $\log_x(y)$.

Например кодът

```
for(double x=2.0, y=8.0, L; y<9.5; x==2.0 ? x+=1.0 : (x=2.0, y+=0.5)) {
  __asm {
    fld1
    fld x
    fyl2x
    fld1
    fdivrp st(1), st(0)
    fld y
    fyl2x
    fstp L
  }
  cout << "log" << x << "(" << setw(3) << y << ") = " << L << endl;
}
```

извежда на екрана

```
log2(  8) = 3
log3(  8) = 1.89279
log2(8.5) = 3.08746
log3(8.5) = 1.94797
log2(  9) = 3.16993
log3(  9) = 2
```

fyl2xp1 st(i) – записва в ***st(1)*** стойността на израза $st(1).\log_2(st(0)+1)$.

st(1) трябва да бъде в интервала $\left(0;1-\frac{1}{\sqrt{2}}\right)$.

idiv op – аналогична на ***div***, но работи с допълнителен код, т. е. дели 2n-разряден допълнителен код с делимо ***op*** и получава n-разрядни цяло частно и остатък също в допълнителен код.
Операндът ***op*** трябва да бъде в регистър или в паметта.
Вариантите са:

разрядност на ***op***	делимо	цяло частно	остатък
8	**ax**	**al**	**ah**
16	**dx:ax**, т. е. 32-разряден допълнителен код, старшите 16 разряда са в **dx**, а младшите 16 разряда са в **ax**	**ax**	**dx**
32	**edx:eax**, т. е. 64-разряден допълнителен код, старшите 32 разряда са в **edx**, а младшите 32 разряда са в **eax**	**eax**	**edx**
64	**rdx:rax**, т. е. 128-разряден допълнителен код, старшите 64 разряда са в **rdx**, а младшите 64 разряда са в **rax**	**rax**	**rdx**

Когато цялото частно не се побира в предвидения за него регистър, предизвиква прекъсване поради грешка при делене. (В този случай някои компилатори съобщават „делене с нула“.)

Например кодът

```
int dividend = -23, divisor = 5, quotient, remainder;
__asm {
  push edx
  push eax
    mov eax, dividend
    cdq // разширява знаково eax до edx:eax
    idiv divisor
    mov quotient, eax
    mov remainder, edx
  pop eax
  pop edx
}
cout << dividend << " = " << quotient << " * " << divisor
     << " + " << remainder << endl;
```

извежда на екрана

```
-23 = -4 * 5 + -3
```

imul op – умножава n-разрядни допълнителни кодове и получава 2n-разрядно произведение също в допълнителен код.
Операндът ***op*** е втори множител и трябва да бъде в регистър или в паметта.
Вариантите са:

разрядност на ***op***	първи множител	произведение
8	**al**	**ax**
16	**ax**	**dx:ax**, т. е. 32-разряден допълнителен код, старшите 16 разряда са в **dx**, а младшите 16 разряда са в **ax**
32	**eax**	**edx:eax**, т. е. 64-разряден допълнителен код, старшите 32 разряда са в **edx**, а младшите 32 разряда са в **eax**
64	**rax**	**rdx:rax**, т. е. 128-разряден допълнителен код, старшите 64 разряда са в **rdx**, а младшите 64 разряда са в **rax**

Например кодът

```
short op1 = -12, op2 = 5;
int product;
__asm {
  push edx
  push eax
```

```
    mov ax, op1
    imul op2
    mov word ptr product, ax
    mov word ptr product + 2, dx
  pop eax
  pop edx
}
cout << op1 << " * " << op2 << " = " << product << endl;
```

извежда на екрана

```
-12 * 5 = -60
```

imul op1, op2 – умножава допълнителни кодове и записва произведението им на мястото на ***op1***.
Операндите трябва да бъдат в регистри или в паметта.

Например кодът

```
int op1 = -3, op2 = 4;
int product;
__asm {
  push esi
    mov esi, op1
    imul esi, op2
    mov product, esi
  pop esi
}
cout << op1 << " * " << op2 << " = " << product << endl;
```

извежда на екрана

```
-3 * 4 = -12
```

imul op1, op2, op3 – записва в регистъра ***op1*** произведението на ***op1***, ***op2*** и числото ***op3***.
Операндите и резултатът са в допълнителен код.
op1 и ***op2*** трябва да бъдат с еднаква разрядност.

Например кодът

```
int op1 = -3;
int product;
__asm {
  push esi
    mov esi, op1
    imul esi, esi,  21
    mov product, esi
  pop esi
}
cout << op1 << " * " << 21 << " = " << product << endl;
```

извежда на екрана

```
-3 * 21 = -63
```

inc op – увеличава с единица стойността на ***op***;
Не променя флага **CF**, но модифицира флагове **ZF**, **OF**, **SF** и **PF**.

int op – предизвиква прекъсване с номер ***op***.

j... adr – условен преход към адреса ***adr***, като условието е назовано с буквите, заместващи многоточието.
Когато в названието на инструкцията, сред буквите, заместващи многоточието, присъствува буквата ***n***, тя винаги означава отрицание на условието, назовано с останалите букви.
Вариантите на условните преход са следните:

Условен преход, според регистър със стойност нула

За такъв преход ***adr*** трябва да се получава с прибавяне на 8-разрядно знаково изместване към адреса на самата команда ***j...cxz*** за условен преход.

jcxz adr – условен преход към адреса ***adr***, точно когато регистърът **cx** е нула;

jecxz adr – условен преход към адреса ***adr***, точно когато регистърът **ecx** е нула;

jrcxz adr – условен преход към адреса ***adr***, точно когато регистърът **rcx** е нула.

Условен преход, според стойността на флаг

За такъв преход може да се използват само флагове **ZF**, **CF**, **OF**, **SF** и **PF**.

jz – преход при **ZF**=1;

jnz – преход при **ZF**=0;

jc – преход при **CF**=1;

jnc – преход при **CF**=0;

jo – преход при **OF**=1;

jno – преход при **OF**=0;

js – преход при **SF**=1;

jns – преход при **SF**=0;

jp – преход при **PF**=1;

jnp – преход при **PF**=0.

Условен преход, според релацията между две числа

Обикновено такъв преход се прави според флаговете, модифицирани от предхождаща инструкция **cmp op1, op2** или **sub op1, op2**. Затова мнемониката съответствува на релацията между **op1** и **op2** непосредствено след инструкцията **cmp op1, op2**. Обаче флаговете, използвани за определяне на релацията са различни при работа с код без знак и с допълнителен код. Заради това има два различни набора от инструкции:

Условен преход според релацията между числа, представени чрез ***код без знак:***

je – преход при $op1 = op2$;

jne – преход при $op1 \neq op2$;

jb или **jnae** – преход при $op1 < op2$;

jbe или **jna** – преход при $op1 \leq op2$;

ja или **jnbe** – преход при $op1 > op2$;

jae или **jnb** – преход при $op1 \geq op2$.

Условен преход според релацията между числа, представени чрез ***допълнителен код:***

je – преход при $op1 = op2$;

jne – преход при $op1 \neq op2$;

jl или **jnge** – преход при $op1 < op2$;

jle или **jng** – преход при $op1 \leq op2$;

jg или **jnle** – преход при $op1 > op2$;

jge или **jnl** – преход при $op1 \geq op2$.

Например кодът

```
char n = 125;
while( n > 0 ) {
  cout << (int)n << " + 1";
  bool overflow;
  __asm {
      mov overflow, 1
      inc n
      jo Exist
        mov overflow, 0
    Exist :
  }
  cout << (overflow ? " != " : " == ") << (int)n << endl;
}
```

извежда на екрана

```
125 + 1 == 126
126 + 1 == 127
127 + 1 != -128
```

Съответно кодът

```
short n1 = -3, n2 = 2, min;
__asm {
  push ecx
      mov cx, n1
      cmp cx, n2
      jng RESULT
        mov cx, n2
    RESULT :
      mov min, cx
  pop ecx
}
cout << "min(" << n1 << ";" << n2 << ") = " << min << endl;
```

извежда на екрана

```
min(-3;2) = -3
```

jmp adr – безусловно предава управлението към адреса ***adr*** (обикновено ***adr*** е етикет).
Тази инструкция може да направи преход към всеки адрес, достъпен за програмата. Съответно тя може да се използва за „заобикаляне“ на ограниченията за адресите при други преходи.

Пример за такова „заобикаляне“ е следният код (намиращ сумата на целите числа от n до 1):

```
int n = 4, r = 0;
__asm {
  push ecx
      mov ecx, n
    adr :       add r, ecx
                // тук може да има произволно дълъг код
            loop Next // този преход е ограничен до 8-разрядно изместване
            jmp Stop
    Next :  jmp adr // този скок може да бъде произволно далеч
    Stop :
  pop ecx
}
cout << "r = " << r << endl;
```

Горният код извежда на екрана:

```
r = 10
```

lahf – записва младшия байт на регистъра с флагове **RFLAGS** в регистъра **ah**.

lea op1, op2 – записва в ***op1*** адреса на ***op2*** (записва в ***op1*** само отместването от пълния логически адрес на ***op2***) без да прави достъп до ***op2*** в паметта.
Обикновено след тази инструкция ***op1*** се използва като базов регистър за адресиране на операнда ***op2*** или на съседни с него данни.
op1 трябва да бъде регистър (с общо предназначение) със същата разрядност, каквато има изпълняваният в момента код.
op2 трябва да бъде в оперативната памет (да има адрес).

Например кодът

```
int ar[] = { 1, 2, 3 };
cout << ar[0] << ',' << ar[1] << ',' << ar[2] << " -> ";
__asm {
  push eax
  push esi
    lea esi, ar
    mov eax, [esi]
```

```
      xchg eax, [esi+8]
      mov [esi], eax
    pop esi
    pop eax
  }
  cout << ar[0] << ',' << ar[1] << ',' << ar[2] << endl;
```

извежда на екрана

```
1,2,3 -> 3,2,1
```

lodsb – записва в регистъра **al** един байт от адрес **byte ptr ds:[esi]** (или в 64-разряден режим съответно от адрес **byte ptr [rsi]**) и след това:
когато флагът **DF=0**, *прибавя* едно към **esi** (или съответно към **rsi**);
когато флагът **DF=1**, *намалява* с едно **esi** (или съответно **rsi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например кодът

```
const int Len = 20;
char s[Len] = "abcdefgh";
for(int Num = 8; 0<Num; --Num) {
  char cNum;
  __asm {
    push ax
    push ecx
    push esi
    pushf
        cld
        mov ecx, Num
        lea esi, s
      rep lodsb
        mov cNum, al
    popf
    pop esi
    pop ecx
    pop ax
  }
  cout << "s[" << Num-1 << "] = \'" << cNum << "\'\n";
}
```

извежда на екрана

```
s[7] = 'h'
s[6] = 'g'
s[5] = 'f'
s[4] = 'e'
s[3] = 'd'
s[2] = 'c'
s[1] = 'b'
s[0] = 'a'
```

lodsd – записва в регистъра **eax** четири байта от адрес **byte ptr ds:[esi]** (или в 64-разряден режим съответно от адрес **byte ptr [rsi]**) и след това:
когато флагът **DF=0**, *прибавя* четири към **esi** (или съответно към **rsi**);
когато флагът **DF=1**, *намалява* с четири **esi** (или съответно **rsi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

lodsq – записва в регистъра **rax** осем байта от адрес **byte ptr ds:[esi]** (или в 64-разряден режим съответно от адрес **byte ptr [rsi]**) и след това:
когато флагът **DF=0**, *прибавя* осем към **esi** (или съответно към **rsi**);
когато флагът **DF=1**, *намалява* с осем **esi** (или съответно **rsi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

lodsw – записва в регистъра **ax** два байта от адрес **byte ptr ds:[esi]** (или в 64-разряден режим съответно от адрес **byte ptr [rsi]**) и след това:
когато флагът **DF=0**, *прибавя* две към **esi** (или съответно към **rsi**);
когато флагът **DF=1**, *намалява* с две **esi** (или съответно **rsi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например кодът

```
short ar[] = { -1, 22, -3, 44, -5, 66, -7, 88 };
const int Len = sizeof(ar) / sizeof(ar[0]);
for(int i = Len-1; 0<=i; --i) {
  short elm;
  __asm {
    push ax
    push ecx
    push esi
    pushf
        std
        mov esi, Len
        lea esi, ar[esi*2-2]
        mov ecx, Len
        sub ecx, i
      rep lodsw
        mov elm, ax
    popf
    pop esi
    pop ecx
    pop ax
  }
  cout << "ar[" << i << "] = " << elm << endl;
}
```

извежда на екрана

```
ar[7] = 88
ar[6] = -7
ar[5] = 66
```

```
ar[4] = -5
ar[3] = 44
ar[2] = -3
ar[1] = 22
ar[0] = -1
```

Loop adr – намалява с единица регистъра **ecx**, а в 64-разряден режим съответно **rcx**, без да променя флагове и след това, само когато **ecx** все още е различно от нула, прави преход към адреса ***adr***. Адресът ***adr*** трябва да може да се получи от адреса на самата инструкция ***Loop adr*** чрез прибавяне на 8-разрядно знаково изместване.

Например кодът

```
int n = 4, r = 0;
short elm;
__asm {
  push ecx
      mov ecx, n
    Label : inc r
            loop Label
  pop ecx
}
cout << "r = " << r << endl;
```

извежда на екрана

```
r = 4
```

Loope adr – намалява с единица регистъра **ecx**, а в 64-разряден режим съответно **rcx**, без да променя флагове (включително без да променя и флага **ZF**) и след това, само когато **ecx** все още е различно от нула и *едновременно* с това флагът **ZF==1**, прави преход към адреса ***adr***. Адресът ***adr*** трябва да може да се получи от адреса на самата инструкция ***Loop adr*** чрез прибавяне на 8-разрядно знаково изместване.

Loopne adr – намалява с единица регистъра **ecx**, а в 64-разряден режим съответно **rcx**, без да променя флагове (включително без да променя и флага **ZF**) и след това, само когато **ecx** все още е различно от нула и *едновременно* с това флагът **ZF==0**, прави преход към адреса ***adr***. Адресът ***adr*** трябва да може да се получи от адреса на самата инструкция ***Loop adr*** чрез прибавяне на 8-разрядно знаково изместване.

Например кодът

```
char ar[] = "abcde", *s = ar;
short Len;
do {
  __asm {
    push ecx
    push esi
        xor ecx, ecx // за цикъл, който се повтаря до 2^32 пъти
        mov esi, s // записва в esi адреса на низа ar
        dec esi
      Cycle : inc esi
              cmp byte ptr [esi], 0
        loopne Cycle
        sub esi, s
        mov Len, si
    pop esi
    pop ecx
  }
  cout << "Length OF \"" << s << "\" = " << Len << endl;
```

```
    if( Len ) s[Len-1] = 0;
  } while( Len );
```

извежда на екрана

```
Length OF "abcde" = 5
Length OF "abcd" = 4
Length OF "abc" = 3
Length OF "ab" = 2
Length OF "a" = 1
Length OF "" = 0
```

loopnz *е еквивалентна на* ***loopne***.

loopz *е еквивалентна на* ***loope***.

mov op1, op2 – записва в ***op1*** операнда ***op2***.
Двата операнда трябва да бъдат с равни разрядности.

movsb – премества байта **byte ptr ds:[esi]** (в 64-разряден режим съответно **byte ptr [rsi]**) в **byte ptr es:[edi]** (в 64-разряден режим съответно в **byte ptr [rdi]**) и след това:
когато флагът **DF=0**, *прибавя* единица към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с единица **esi** и **edi** (или съответно **rsi** и **rdi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например кодът (показващ копиране на низ)

```
char ja[20] = "abcd", s2[20] = "123456789";
__asm {
  push esi
  push edi
  pushf
      //mov si, ds // само за 32-разряден режим
      //mov es, si
      lea esi, s1
      lea edi, s2
      cld
    Cycle : Cycle : cmp byte ptr [esi], 0
            movsb
        jne Cycle
  popf
  pop edi
  pop esi
}
cout << "s2 = \"" << s2 << "\"\n";
```

извежда на екрана

```
s2 = "abcd"
```

movsd – премества четворка байтове от адрес **dword ptr ds:[esi]** (или в 64-разряден режим съответно от адрес **dword ptr [rsi]**) на адрес **dword ptr es:[edi]** (или в 64-разряден режим съответно на адрес **dword ptr [rdi]**) и след това:
когато флагът **DF=0**, *прибавя* четири към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с четири **esi** и **edi** (или съответно **rsi** и **rdi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например кодът (показващ копиране на низ)

```
int ar1[] = {  1,  2,  3,  4,  5,  6,  7,  8,  9 },
    ar2[] = { -1, -2, -3, -4, -5, -6, -7, -8, -9 };
int const L = sizeof(ar1) / sizeof(ar1[0]);
__asm {
  push esi
  push edi
  pushf
  push ecx
      //mov si, ds // само за 32-разряден режим
      //mov es, si
      mov ecx, L
      inc ecx
      sar ecx, 1 // целочислено разделя ecx на 2
      lea esi, ar1
      lea edi, ar2
      cld
    Cycle : movsd
            add esi, 4 // прескача един копиран елемент
            add edi, 4 // прескача едно място за копиране
        loop Cycle
  pop ecx
  popf
  pop edi
  pop esi
}
for(int i=0; i<L; ++i) cout << ar2[i] << ' ';
cout << endl;
```

извежда на екрана

```
1 -2 3 -4 5 -6 7 -8 9
```

movsq – премества осем байта от адрес **qword ptr ds:[esi]** (или в 64-разряден режим съответно от адрес **qword ptr [rsi]**) на адрес **qword ptr es:[edi]** (или в 64-разряден режим съответно на адрес **qword ptr [rdi]**) и след това:
когато флагът **DF=0**, *прибавя* осем към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с осем **esi** и **edi** (или съответно **rsi** и **rdi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

movsw – премества четири байта от адрес `word ptr ds:[esi]` (или в 64-разряден режим съответно от адрес `word ptr [rsi]`) на адрес `word ptr es:[edi]` (или в 64-разряден режим съответно на адрес `word ptr [rdi]`) и след това:
когато флагът **DF=0**, *прибавя* две към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с две **esi** и **edi** (или съответно **rsi** и **rdi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

movsx op1, op2 – разширява знаково допълнителния код ***op2*** от 8 или 16 разряда до допълнителен код с разрядността на операнда ***op1*** и записва полученото в ***op1***.
op1 трябва да бъде регистър с разрядност, по-голяма от тази на ***op2***.
op2 трябва да бъде регистър или памет с разрядност 8 или 16.

movsxd op1, op2 – разширява знаково, т. е. като допълнителен код, ***op2*** от 32 разряда до 64 разряда и записва полученото в ***op1***.
op1 трябва да бъде 64-разряден регистър.
op2 трябва да бъде 32-разряден или в регистър, или в паметта.

movzx op1, op2 – разширява беззнаково, т. е. като код без знак, ***op2*** от 8 или 16 разряда до разрядността на операнда ***op2*** и записва полученото в ***op1***.
op1 трябва да бъде регистър с разрядност, по-голяма от тази на ***op2***.
op2 трябва да бъде регистър или памет с разрядност 8 или 16.

mul op – умножава n-разрядни кодове без знак и получава 2n-разрядно произведение също в код без знак.
Операндът ***op*** е втори множител и трябва да бъде в регистър или в паметта.
Вариантите са:

разрядност на ***op***	първи множител	произведение
8	**al**	**ax**
16	**ax**	**dx:ax**, т. е. 32-разряден код без знак, старшите 16 разряда са в **dx**, а младшите 16 разряда са в **ax**
32	**eax**	**edx:eax**, т. е. 64-разряден код без знак, старшите 32 разряда са в **edx**, а младшите 32 разряда са в **eax**
64	**rax**	**rdx:rax**, т. е. 128-разряден код без знак, старшите 64 разряда са в **rdx**, а младшите 64 разряда са в **rax**

Например кодът

```
unsigned short op1 = 12, op2 = 5;
unsigned int product;
__asm {
```

```
        push edx
        push eax
          mov ax, op1
          imul op2
          mov word ptr product, ax
          mov word ptr product + 2, dx
        pop eax
        pop edx
      }
      cout << op1 << " * " << op2 << " = " << product << endl;
```

извежда на екрана

```
12 * 5 = 60
```

neg op – променя (обръща) знака на ***op***, интерпретиран като допълнителен код. (Т. е. прилага операцията допълнение до две към ***op***.)

not op – записва в ***op*** поразрядното отрицание на ***op***

or op1, op2 – записва в ***op1*** поразрядната дизюнкция (поразрядното Или; XOR) на ***op1*** и ***op2***

pop op – записва в ***op*** върха на хардуерно поддържания стек и го премахва;
актуализира регистъра **esp** (или **rsp** съответно в 64-разряден режим).
Връх на стека е **ss:[esp]** в 32-разряден режим или **ss:[rsp]** в 64-разряден режим.

popa – възстановява от хардуерно поддържания стек регистрите **ax**, **bx**, **cx**, **dx**, **si**, **di**, **sp**, **bp** като премахва от стека съответните стойности.

popad – възстановява от хардуерно поддържания стек регистрите **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **esp**, **ebp** като премахва от стека съответните стойности.

popf – премахва 16-разрядна данна от върха на хардуерно поддържания стек, като записва премахнатото в регистър **FLAGS**.
Връх на стека е **ss:[esp]** в 32-разряден режим или **ss:[rsp]** в 64-разряден режим.

popfd – премахва 32-разрядна данна от върха на хардуерно поддържания стек и записва премахнатото във флаговия регистър **EFLAGS**.
Връх на стека е **ss:[esp]** в 32-разряден режим или **ss:[rsp]** в 64-разряден режим.
(В потребителски режим системните флагове са недостъпни, а отделно има особеност за **VM** и **RF**.)

popfq – премахва 64-разрядна данна от върха на хардуерно поддържания стек и записва премахнатото във флаговия регистър **RFLAGS**.
Връх на стека е **ss:[esp]** в 32-разряден режим или **ss:[rsp]** в 64-разряден режим.
(В потребителски режим системните флагове са недостъпни, а отделно има особеност за **VM** и **RF**.)

push op – записва ***op*** в хардуерно поддържания стек,
т. е. намалява според разрядността на операнда **esi** в 32-разряден режим или **rsi** в 64-разряден режим и след това записва операнда на адрес **ss:[esp]** в 32-разряден режим или **ss:[rsp]** в 64-разряден режим.
op трябва да бъде:
или 2-, 4- или 8-байтов регистър с общо предназначение или операнд в паметта;
или число (непосредствен операнд), но то ще бъде знаково разширено до 2, 4 или 8 байта (според правилата на транслатора);
или, но само в 32-разряден режим, сегментен регистър (който и да било от **cs**, **ss**, **ds**, **es**, **fs**, **gs**).

pusha – записва в хардуерно поддържания стек регистрите **ax**, **bx**, **cx**, **dx**, **si**, **di**, **sp**, **bp**.

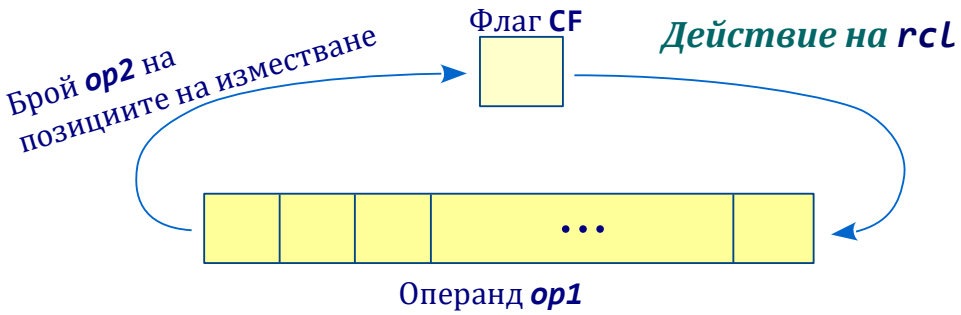
pushad – записва в хардуерно поддържания стек регистрите **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **ebp**, **esp**.

pushf – добавя (съдържанието на) регистъра **FLAGS** (младшите 16 разряда на флаговия регистър **RFLAGS**) като нов връх на хардуерно поддържания стек.

pushfd – добавя (съдържанието на) регистъра **EFLAGS** (младшите 32 разряда на флаговия регистър **RFLAGS**) като нов връх на хардуерно поддържания стек.

pushfq – добавя (съдържанието на) регистъра **RFLAGS** като нов връх на хардуерно поддържания стек. (В потребителски режим системните флагове са недостъпни, а отделно има особеност за **VM** и **RF**.)

rcl op1, op2 – циклично ротира (измества циклично; шифтва с въртене) разрядите на ***op1*** *наляво* през флага **CF** на ***op2*** позиции.
op2 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число.
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Следващата схема илюстрира действието:



При ***op2=0*** не променя флагове.
Въздейства на флага **OF** по следния начин:
Когато ***op2=1***, тогава записва във флага **OF** резултата от разделителното или над новите (след ротацията) стойности на флага **CF** и старшия бит на **op1**.
Когато ***op2>1***, тогава флагът **OF** е неопределен.

Например кодът

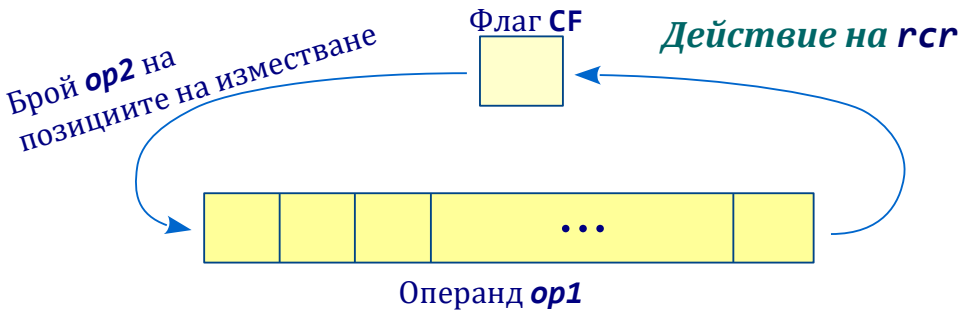
```
unsigned short i, j = i = 0x1f11, iCF, jCF = iCF = 0;
cout <<   " i= " << hex << i << " => stc ; rcl i, 1"
     << "\n j= " << hex << j << " => stc ; rcl j, 5" << endl;
__asm {
  stc
  rcl i, 1
  setc byte ptr iCF
  stc
  rcl j, 5
  setc byte ptr jCF
}
cout <<   " i= " << hex << i << " <-> CF = " << iCF
     << "\n j= " << hex << j << " <-> CF = " << jCF << endl;
```

извежда на екрана

```
 i= 1f11 => stc ; rcl i, 1
 j= 1f11 => stc ; rcl j, 5
```

```
i= 3e23 <-> CF = 0
j= e231 <-> CF = 1
```

rcr op1, op2 – циклично ротира (измества циклично; шифтва с въртене) разрядите на ***op1*** *надясно* през флага **CF** на ***op2*** позиции.
op2 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число.
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Следващата схема илюстрира действието:



При ***op2=0*** не променя флагове.
Въздейства на флага **OF** по следния начин:
Когато ***op2=1***, тогава записва във флага **OF** резултата от разделителното или над новите (след ротацията) стойности на старшите два бита на **op1**.
Когато ***op2>1***, тогава флагът **OF** е неопределен.

Например кодът

```
unsigned short i = 0x0301;
unsigned char CFlag;
cout <<   "i = " << hex << i  << " => stc ; rcr i, 9"<< endl;
__asm {
  stc
  rcr i, 9
  setc byte ptr CFlag
}
cout <<   "i = " << hex << i << " <-> CF = " << (int)CFlag << endl;
```

извежда на екрана

```
i = 301 => stc ; rcr i, 9
i = 181 <-> CF = 1
```

rep – префикс (не е самостоятелна инструкция), който може да стои пред така наричаните „низови“ инструкции и предизвиква тяхното зацикляне според **ecx** в 32-разряден режим или според **rcx** в 64-разряден режим. По-точно действието е следното:
Когато **ecx==0** (респективно **rcx==0**), тогава *не се изпълнява* инструкцията след ***rep***.
Когато **ecx!=0** (респективно **rcx!=0**), тогава *първо се изпълнява* инструкцията след ***rep***, *след това се намалява с единица* регистърът **ecx** (респективно **rcx**) без да се актуализират флагове, а *накрая се променя* **ecx** (респективно **rcx**) според заварения флаг **DF** по следния начин:
при **DF=0**, *се прибавя* броя байтове, заемани от операнда на инструкцията, съответно към **esi**, към **edi** или и към двата(респективно към **rsi**, към **rdi** или и към двата), като конкретната инструкция определя, кой точно от двата регистъра се модифицира (това действие се нарича

автоувеличение);
при **DF=1**, *се намалява* с броя байтове, заемани от операнда на инструкцията, съответно **esi**, **edi** или и двата (респективно **rsi**, **rdi** или и двата), като конкретната инструкция определя, кой точно от двата регистъра се модифицира (това действие се нарича автонамаление).
Във всички случаи се модифицират точно регистрите, участвуващи в конкретната инструкция.

Например кодът (показващ преместване на част от масива на ново място в същия масив)

```
int ar[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
const int L = sizeof(ar) / sizeof(ar[0]);
__asm {
  push ecx
  push esi
  pushf
      mov ecx, L
      lea edi, ar[ecx*4-8]
      sub ecx, 5
      lea esi, ar[ecx*4-4]
      std
    rep movsd
  popf
  pop esi
  pop ecx
}
for(int i=0; i<L; ++i) cout << ar[i] << ' ';
cout << endl;
```

извежда на екрана (на такъв фон е преместената част)

```
1 2 3 4 1 2 3 4 5 6 7 8 9 10 15
```

repe – префикс, който работи почти както ***rep***, но с единствената разлика, че за изпълнението на инструкцията, освен **ecx!=0** (респективно **rcx!=0**), е необходимо едновременно и флагът **ZF=1**.

repne – префикс, който работи почти както ***rep***, но с единствената разлика, че за изпълнението на инструкцията, освен **ecx!=0** (респективно **rcx!=0**), е необходимо едновременно и флагът **ZF=0**.

repnz – префикс (не е самостоятелна инструкция), еквивалентен на ***repne***.

repz – префикс (не е самостоятелна инструкция), еквивалентен на ***repe***.

ret num – извлича от стека (и премахва) адреса за връщане от процедура (от подпрограма) и прави преход към извлечения адрес.
Когато има операнд, което не е задължително, премахва ***num*** байта от върха на стека след извличането на адреса.

Например кодът

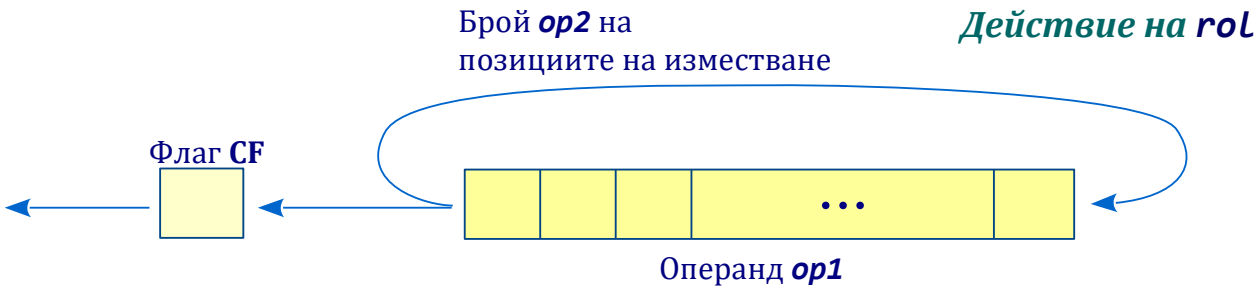
```
int n1 = -123, n2 = -456, n3 = -789, r = 89;
__asm {
    jmp Lab2
  Lab1 :
    ret 8
  Lab2 :
    push n1
    push n2
    push n3
    call Lab1
    pop r
}
```

```
cout << "r = " << r << endl;
```

извежда на екрана

```
r = -123
```

rol op1, op2 – циклично ротира (измества циклично; шифтва с въртене) разрядите на ***op1*** *наляво* на ***op2*** позиции.
op1 трябва да бъде регистър или да има адрес в паметта.
op2 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число.
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Следващата схема илюстрира действието:



Въздейства на флагове по следния начин:
При ***op2=0*** не променя флагове.
При ***op2>0*** записва в **CF** последния преместен разряд.
Когато ***op2=1***, тогава записва във флага **OF** резултата от разделителното или над новите (след ротацията) стойности на флага **CF** и старшия бит на **op1**.
Когато ***op2>1***, тогава флагът **OF** е неопределен.

Например кодът

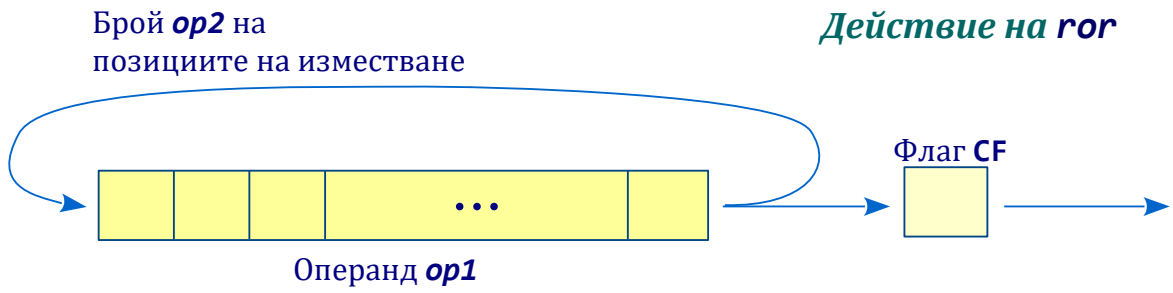
```
unsigned short n = 0x3212;
bool Cflag, Oflag;
for(int i=1; i<=4; ++i) {
  cout << " value: " << hex << n << endl;
  __asm {
    rol n, 1
    setc Cflag
    seto Oflag
  }
  cout << "result: " << hex << n << endl;
  cout << "CF = " << Cflag << endl;
  cout << "OF = " << Oflag << endl;
}
```

извежда на екрана

```
 value: 3212
result: 6424
CF = 0
OF = 0
 value: 6424
result: c848
CF = 0
OF = 1
```

```
 value: c848
result: 9091
CF = 1
OF = 0
 value: 9091
result: 2123
CF = 1
OF = 1
```

ror op1, op2 – циклично ротира (измества циклично; шифтва с въртене) разрядите на ***op1*** *надясно* на ***op2*** позиции.
op1 трябва да бъде регистър или да има адрес в паметта.
op2 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число (непосредствен операнд).
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Следващата схема илюстрира действието:



Въздейства на флагове по следния начин:
При ***op2=0*** не променя флагове.
При ***op2>0*** записва в **CF** последния преместен разряд.
Когато ***op2=1***, тогава записва във флага **OF** резултата от разделителното или над новите (след ротацията) стойности на старшите два бита на **op1**.
Когато ***op2>1***, тогава флагът **OF** е неопределен.

Например кодът

```
unsigned short n = 0x2123;
bool Cflag, Oflag;
for(int i=1; i<=4; ++i) {
  cout << " value: " << hex << n << endl;
  __asm {
    ror n, 1
    setc Cflag
    seto Oflag
  }
  cout << "result: " << hex << n << endl;
  cout << "CF = " << Cflag << endl;
  cout << "OF = " << Oflag << endl;
}
```

извежда на екрана

```
 value: 2123
result: 9091
CF = 1
OF = 1
```

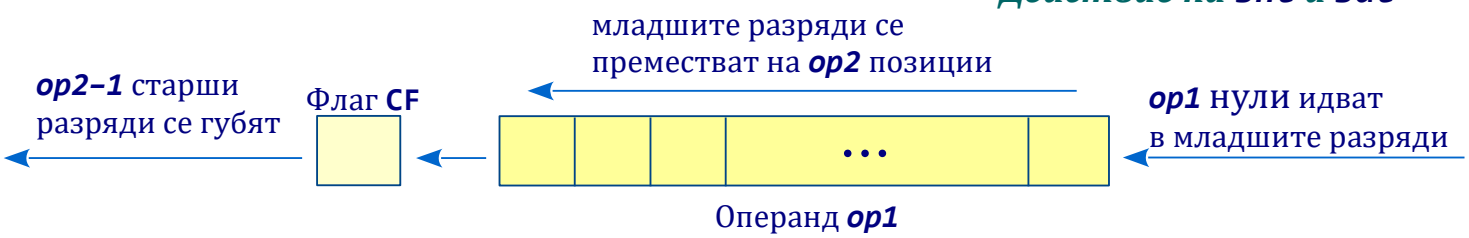
```
 value: 9091
result: c848
CF = 1
OF = 0
 value: c848
result: 6424
CF = 0
OF = 1
 value: 6424
result: 3212
CF = 0
OF = 0
```

sahf – записва регистъра **ah** в младшия байт на **FLAGS**, но игнорира разрядите на **ah** с номера 1, 3 и 5. В съответните разряди на **FLAGS** записва 1, 0 и 0. По този начин влияе на флаговете **SF** (разряд номер 7), **ZF** (разряд номер 6), **AF** (разряд номер 4), **PF** (разряд номер 2) и **CF** (разряд номер 0).

sal op1, op2 – еквивалентно на ***shl*** (аритметично изместване *наляво* през флага **CF**).

sal и за код без знак, и за допълнителен код е еквивалентна на умножение на ***op1*** с 2^{op2} и записване на резултата пак в ***op1***.
Както и при другите измествания и тук при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Въздействия на флаговете, точно както ***shl***.
Действието на инструкцията илюстрира следната схема (същата, както за ***shl***):

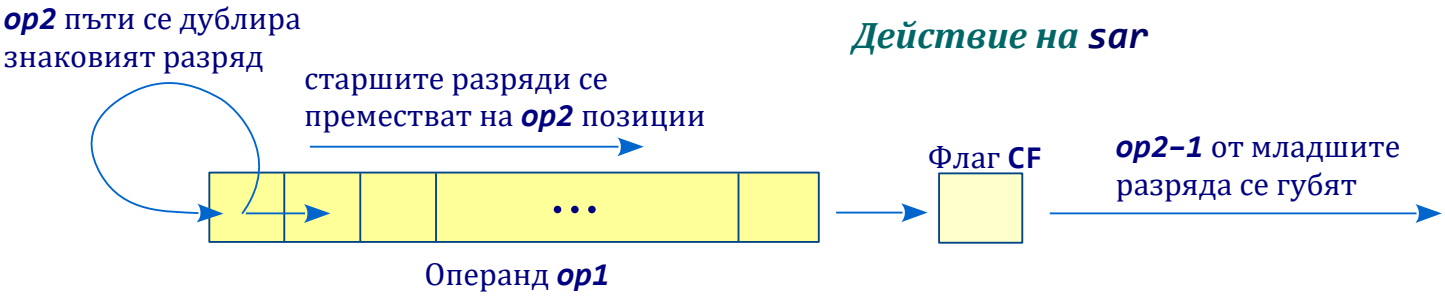
Действие на shl и sal



sar op1, op2 – аритметично изместване *надясно* през флага **CF**.

Както за код без знак, така и за допълнителен код, е еквивалентно на делене на ***op1*** с 2^{op2} със закръгляне на резултата надолу (т. е. към най-близкото цяло число, равно на или по-малко от ***op1***) и записване на резултата в ***op1***.
sar се различава от ***shr*** по това, че при ***sar*** знаковият разряд се дублира (запазва си стойността), докато при ***shr*** отляво се дописват нули.
op1 трябва да бъде регистър или да има адрес в паметта.
op2 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число.
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.

Следващата схема илюстрира действието на ***sar***:



sar влияе на флагове по следния начин:
При ***op2=0*** не променя флагове.
При ***op2>0*** записва в **CF** последния разряд, излизащ извън ***op1***.
Когато ***op2=1***, тогава записва нула във флага **OF**.
Когато ***op2>1***, тогава флагът **OF** е неопределен.
По обичайните критерии се модифицират флаговете **ZF**, **SF** и **PF**.
Във всички варианти флагът **AF** е неопределен.

Например кодът

```
for(short n, r = n = -15; r<7; n=r+=3) {
  cout << n << " -> ";
  __asm sar n, 1
  cout << n << endl;
}
```

извежда на екрана

```
-15 -> -8
-12 -> -6
-9 -> -5
-6 -> -3
-3 -> -2
0 -> 0
3 -> 1
6 -> 3
```

sbb op1, op2 – изважда от ***op1*** първо ***op1***, а после и флага **CF**, и записва резултата в ***op1***.

Например кодът

```
int n1 = 15, n2 = 5, res;
__asm {
  push edi
    stc
    mov edi, n1
    sbb edi, n2
    mov res, edi
  pop edi
}
cout << "res = " << res << endl;
```

извежда на екрана

```
res = 9
```

scasb – изчислява разликата **al - byte ptr es:[edi]** (или в 64-разряден режим съответно **al - byte ptr [rdi]**), т. е. разликата на 8-разрядни операнди, без да съхранява получения резултат, но като модифицира флаговете, точно както биха ги променили инструкциите ***cmp*** или ***sub*** със същите умаляемо и умалител, и след това:
когато флагът **DF=0**, *прибавя* едно към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с единица **esi** и **edi** (или съответно **rsi** и **rdi**).
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

scasd – изчислява разликата **eax - dword ptr es:[edi]** (или в 64-разряден режим съответно **eax - dword ptr [rdi]**), т. е. разликата на 32-разрядни операнди, без да съхранява получения резултат, но като модифицира флаговете, точно както биха ги променили инструкциите ***cmp*** или ***sub*** със същите умаляемо и умалител, и след това:
когато флагът **DF=0**, *прибавя* четири към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с четири **esi** и **edi** (или съответно **rsi** и **rdi**).
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например кодът (показваш намиране индекса на последния елемент, равен на дадено число)

```
int ar[10] = { -1, -2, -3, -4, -1, -2, -3, -4, -1, -2 };
             //  0   1   2   3   4   5   6   7   8   9
for(int n=-5, pos; n<=1; ++n) {
  __asm {
    pushad
    pushf
          mov ax, ds
          mov es, ax
          std
          mov ecx, 10
          lea edi, ar + 9*4
          mov eax, n
        repne scasd
          jz Exist
            mov pos, -1
            jmp Ready
      Exist :
            mov pos, ecx
      Ready :
    popf
    popad
  }
  if( pos<0 ) cout<<"ar[...] != "<<n<<endl;
  else cout<<"ar["<<pos<<"] = "<<ar[pos]<<" = "<<n<<endl;
}
```

извежда на екрана

```
ar[...] != -5
ar[7] = -4 = -4
ar[6] = -3 = -3
ar[9] = -2 = -2
ar[8] = -1 = -1
ar[...] != 0
ar[...] != 1
```

scasq – изчислява разликата **rax - qword ptr es:[edi]** (или в 64-разряден режим съответно **rax - qword ptr [rdi]**), т. е. разликата на 64-разрядни операнди, без да съхранява получения резултат, но като модифицира флаговете, точно както биха ги променили инструкциите ***cmp*** или ***sub*** със същите умаляемо и умалител, и след това:
когато флагът **DF=0**, *прибавя* осем към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с осем **esi** и **edi** (или съответно **rsi** и **rdi**).
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

scasw – изчислява разликата **ax - word ptr es:[edi]** (или в 64-разряден режим съответно **ax - word ptr [rdi]**), т. е. разликата на 16-разрядни операнди, без да съхранява получения резултат, но като модифицира флаговете, точно както биха ги променили инструкциите ***cmp*** или ***sub*** със същите умаляемо и умалител, и след това:
когато флагът **DF=0**, *прибавя* две към **esi** и към **edi** (или съответно към **rsi** и към **rdi**);
когато флагът **DF=1**, *намалява* с две **esi** и **edi** (или съответно **rsi** и **rdi**).
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

set... op – записва в ***op*** или 1 (в смисъл на истина), или 0 (в смисъл на неистина) резултата от проверката на логическото условие, назовано с буквите, заместващи многоточието.
Възможни са названията на всички условия, зависещи от флагове, които могат да заместват многоточието в инструкцията ***j...*** за условен преход.
op трябва да бъде или 8-разряден регистър, или байт в паметта.

Например кодът

```
const int Len = 6;
int ar[Len] = { -2, 3, 3, 4, -3, -3 };
for(int i=1; i<Len; ++i) {
  bool notLess;
  __asm {
    push esi
    push eax
      mov esi, i
      mov eax, ar[esi*4-4]
      cmp eax, ar[esi*4]
      setge notLess
    pop eax
    pop esi
  }
  cout << ar[i-1] << (notLess ? " >= " : " < " ) << ar[i] << endl;
}
```

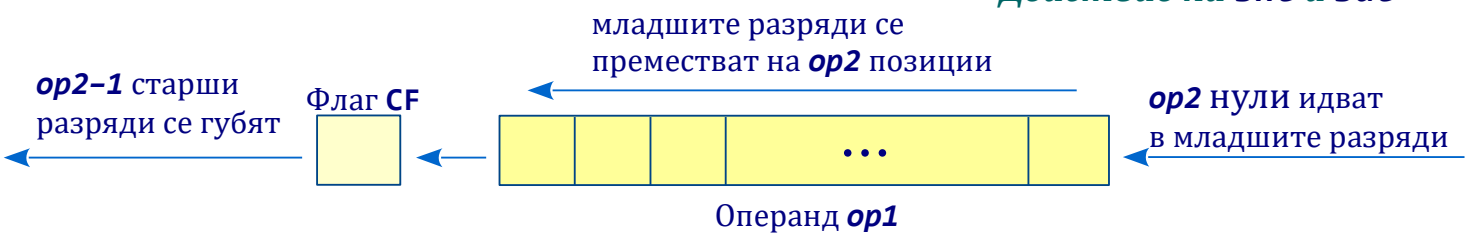
извежда на екрана

```
-2 < 3
3 >= 3
3 < 4
4 >= -3
-3 >= -3
```

shl op1, op2 – логическо изместване на ***op1*** *наляво* на ***op2*** позиции през флага **CF** (при което старшите ***op2–1*** разряда се губят, а отдясно се появяват ***op2*** нули).
op1 трябва да бъде регистър или да има адрес в паметта.
op2 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число.
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при

32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Следващата схема илюстрира действието на ***shl***:

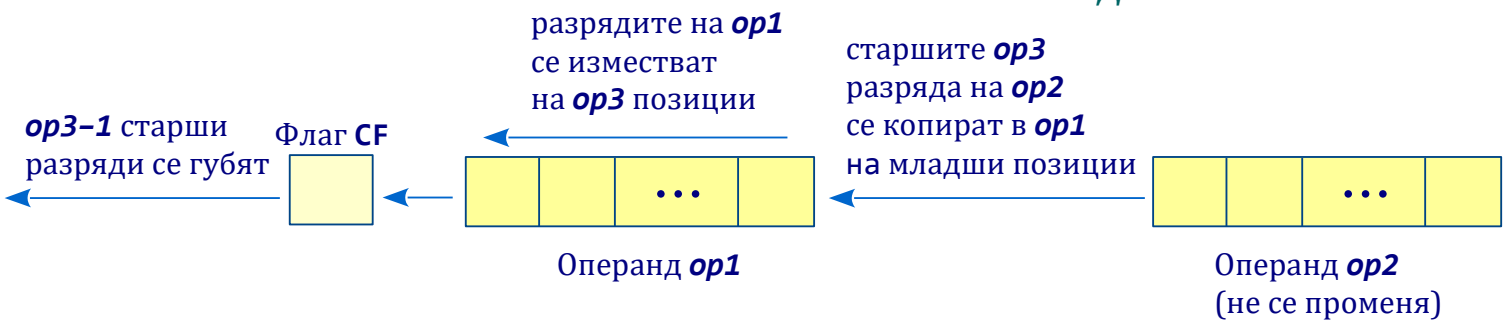
Действие на shl и sal



Инструкцията влияе на флагове по следния начин:
При ***op2=0*** не променя флагове.
При ***op2>0*** записва в **CF** последния преместен разряд.
Когато ***op2=1***, тогава записва във флага **OF** резултата от разделителното или над новите (след изместването) стойности на флага **CF** и старшия бит на **op1**.
Когато ***op2>1***, тогава флагът **OF** е неопределен.
По обичайните критерии се модифицират флаговете **ZF**, **SF** и **PF**.
Във всички варианти флагът **AF** е неопределен.

shld op1, op2, op3 – двойно изместване *наляво* (shift left double) на ***op3*** позиции през флага **CF**, при което старшите ***op3–1*** разряда на ***op1*** се губят, а отдясно в ***op1*** се копират старшите разряди на ***op2***.
op1 трябва да бъде регистър или да има адрес в паметта.
op2 трябва да бъде регистър.
op3 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число.
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Следващата схема илюстрира действието на ***shld***:

Действие на shld



Инструкцията влияе на флагове по следния начин:
При ***op2=0*** не променя флагове.
При ***op2>0*** записва в **CF** последния разряд, излизащ от **op1**.
Когато ***op2=1***, тогава записва единица във флага **OF**, точно при промяна на знака **op1**.
Когато ***op2>1***, тогава флагът **OF** е неопределен.

Например кодът

```
short n1 = 0x1234, n2 = 0x5678;
bool Cflag;
cout << hex << n1 << ' ' << hex << n2 << "  <<-- 12\n";
```

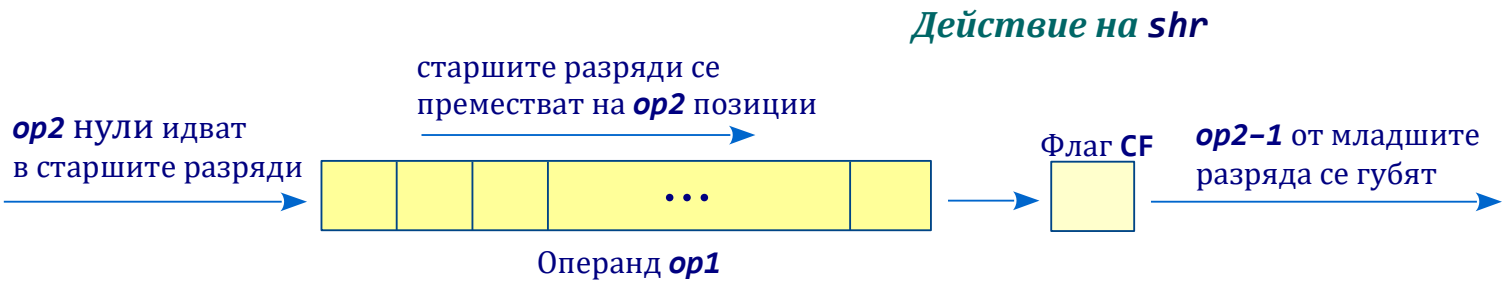
```
__asm {
  push si
    mov si, n2
    shld n1, si, 12
    setc Cflag
  pop si
}
cout << hex << n1 << ' ' << hex << n2 << " // CF = " << Cflag << endl;
```

извежда на екрана

```
1234 5678  <<-- 12
4567 5678 // CF = 1
```

shr op1, op2 – логическо изместване *надясно* през флага **CF**.

Само за код без знак е еквивалентно на делене на ***op1*** с 2^{op2} със закръгляне на резултата надолу (т. е. към най-близкото цяло число, равно на или по-малко от ***op1***) и записване на резултата в ***op1***.
shr се различава от ***sar*** по това, че при ***shr*** отляво се дописват нули, докато при ***sar*** знаковият разряд се дублира (запазва си стойността).
op1 трябва да бъде регистър или да има адрес в паметта.
op2 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число.
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Следващата схема илюстрира действието на ***shr***:



shr влияе на флагове по следния начин:
При ***op2=0*** не променя флагове.
При ***op2>0*** записва в **CF** последния разряд, излизащ извън ***op1***.
Когато ***op2=1***, тогава записва във флага **OF** старшия разряд от стойността на ***op1*** преди изместването.
Когато ***op2>1***, тогава флагът **OF** е неопределен.
По обичайните критерии се модифицират флаговете **ZF**, **SF** и **PF**.
Във всички варианти флагът **AF** е неопределен.

Например кодът

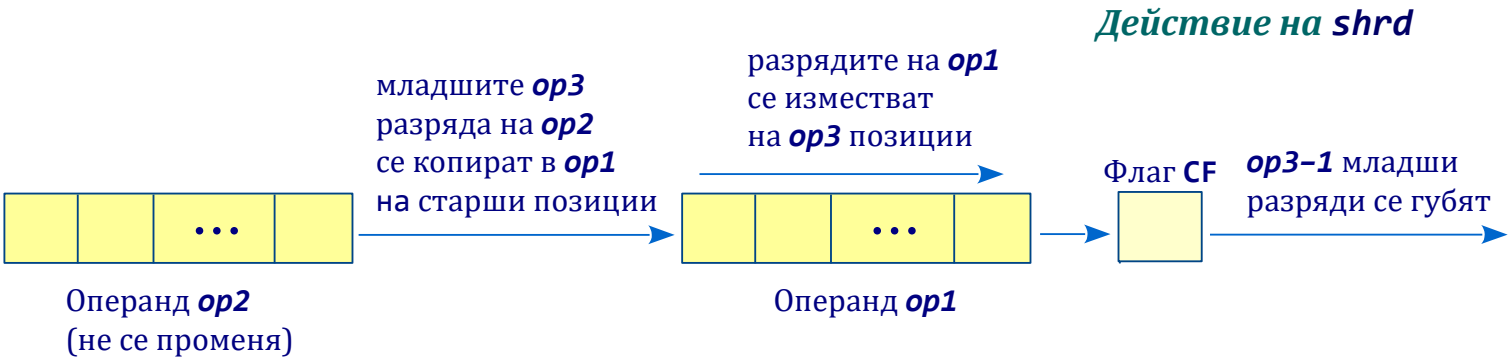
```
short n = 0x8423;
bool Cflag, Oflag;
cout<<hex<<n<<"  ->  1\n";
__asm {
  push si
    mov si, n
    shr n, 1
    setc Cflag
    seto Oflag
  pop si
```

```
}
cout<<hex<<n<<"  //  CF = "<<Cflag<<" , OF = "<<Oflag<<endl;
```

извежда на екрана

```
8423  ->  1
4211  //  CF = 1 , OF = 1
```

shrd op1, op2, op3 – двойно изместване *надясно* (shift right double) на ***op3*** позиции през флага **CF**, при което старшите ***op3–1*** разряда на ***op1*** се губят, а отляво в ***op1*** се копират младшите разряди на ***op2***.
op1 трябва да бъде регистър или да има адрес в паметта.
op2 трябва да бъде регистър.
op3 се интерпретира като код без знак и трябва да бъде или регистъра **cl**, или число.
Всъщност при 64-разряден режим процесорът отчита *само* младшите 6 бита на ***op2***, а при 32-разряден режим отчита *само* младшите 5 разряда на ***op2***. Т. е. изместването винаги е или от 0 до 63, или от 0 до 31 позиции в съответствие с режима на процесора.
Следващата схема илюстрира действието на ***shrd***:



Инструкцията влияе на флагове по следния начин:
При ***op2=0*** не променя флагове.
При ***op2>0*** записва в **CF** последния разряд, излизащ от **op1**.
Когато ***op2=1***, тогава записва единица във флага **OF**, точно при промяна на знака **op1**.
Когато ***op2>1***, тогава флагът **OF** е неопределен.
По обичайните критерии се модифицират флаговете **ZF**, **SF** и **PF**.
Във всички варианти флагът **AF** е неопределен.

Например кодът

```
short n2 = 0x8421, n1 = 0x2421;
bool Cflag, Oflag, Zflag, Sflag, Pflag;
cout << hex << n2 << ' ' << hex << n1 << "  -->> 1\n";
__asm {
  push si
    mov si, n2
    shrd n1, si, 1
    setc Cflag
    seto Oflag
    setz Zflag
    sets Sflag
    setp Pflag
  pop si
}
```

```
cout << hex << n2 << ' ' << hex << n1
     << " // CF = " << Cflag << " ; OF = " << Oflag
     << " ; ZF = " << Zflag << " ; SF = " << Sflag
     << " ; PF = " << Pflag << endl;
```

извежда на екрана

```
8421 2421  -->> 1
8421 9210 // CF = 1 ; OF = 1 ; ZF = 0 ; SF = 1 ; PF = 0
```

stc – записва 1 във флага **CF**.

std – записва 1 във флага **DF**.

stosb – записва регистъра **al** в байта с адрес **byte ptr es:[edi]** (или в 64-разряден режим съответно с адрес **byte ptr [rdi]**) и след това:
когато флагът **DF=0**, *прибавя* едно към **edi** (или съответно към **rdi**);
когато флагът **DF=1**, *намалява* с едно **edi** (или съответно **rdi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например кодът

```
char str[] = "abcdefgh";
cout << "str = \"" << str << "\"\n";
__asm {
  push ax
  push ecx
  push edi
  pushf
      mov ax, ds
      mov es, ax
      cld
      lea edi, str
      mov al, '#'
    Cycle : cmp byte ptr [edi], 0
            je Stop
            stosb
            jmp Cycle
    Stop :
  popf
  pop edi
  pop ecx
  pop ax
}
cout << "str = \"" << str << "\"\n";
```

извежда на екрана

```
str = "abcdefgh"
str = "########"
```

stosd – записва регистъра **eax** в паметта на адрес **dword ptr es:[edi]** (или в 64-разряден режим съответно на адрес **dword ptr [rdi]**) и след това:
когато флагът **DF=0**, *прибавя* четири към **edi** (или съответно към **rdi**);
когато флагът **DF=1**, *намалява* с четири **edi** (или съответно **rdi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

Например кодът

```
int ar1[] = { -1, 1, -2, 2, -5, 5, 9876 };
const int Len = sizeof(ar1) / sizeof(ar1[0]);
for(int i = 0; i < Len; ++i) cout << ar1[i] << ' ';
cout << endl;
int ar2[Len] = { 0 };
short elm;
__asm {
  push eax
  push ecx
  push esi
  push edi
  pushf
      mov ax, ds
      mov es, ax
      cld
      mov ecx, Len
      lea esi, ar1
      lea edi, ar2
    Cycle : lodsd
            stosd
        loop Cycle
  popf
  pop edi
  pop esi
  pop ecx
  pop eax
}
for(int i = 0; i < Len; ++i) cout << ar2[i] << ' ';
cout << endl;
```

извежда на екрана

```
-1 1 -2 2 -5 5 9876
-1 1 -2 2 -5 5 9876
```

stosq – записва регистъра **rax** в паметта на адрес **qword ptr es:[edi]** (или в 64-разряден режим съответно на адрес **qword ptr [rdi]**) и след това:
когато флагът **DF=0**, *прибавя* осем към **edi** (или съответно към **rdi**);
когато флагът **DF=1**, *намалява* с осем **edi** (или съответно **rdi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

stosw – записва регистъра **ax** в паметта на адрес **word ptr es:[edi]** (или в 64-разряден режим съответно на адрес **word ptr [rdi]**) и след това:
когато флагът **DF=0**, *прибавя* две към **edi** (или съответно към **rdi**);
когато флагът **DF=1**, *намалява* с две **edi** (или съответно **rdi**).
Флаговете запазват стойностите си при изпълнението на тази инструкция.
Може да се зацикля чрез префиксите ***rep***, ***repz***, ***repe***, ***repnz*** и ***repne***. При наличие на такъв префикс, изпълнението зависи от регистъра **ecx** (или съответно **rcx**) и той се модифицира съответно.

sub op1, op2 – записва в ***op1*** разлика с умаляемо ***op1*** и умалител ***op2***.

test op1, op2 – изчислява поразрядната конюнкция (поразрядното И, AND) на ***op1*** и ***op2*** без да го съхранява, но модифицира флаговете, точно както би ги променила инструкцията ***and op1, op2***.
op1 трябва да бъде регистър или да има адрес в паметта.
op2 може да бъде и число.
Обикновено се използва за сравняване с шаблон от битове.

xadd op1, op2 – записва ***op1*** в ***op2***, а в ***op1*** записва сумата ***op2-op1***.
Например кодът

```
int n1 = -50, n2 = 90;
cout << "n1 = " << n1 << " ; n2 = " << n2 << endl;
__asm {
  push ecx
  push esi
    mov ecx, n1
    mov esi, n2
    xadd ecx, esi
    mov n1, ecx
    mov n2, esi
  pop esi
  pop ecx
}
cout << "n1 = " << n1 << " ; n2 = " << n2 << endl;
```

извежда на екрана

```
n1 = -50 ; n2 = 90
n1 = 40 ; n2 = -50
```

xchg op1, op2 – разменя стойностите на ***op1*** и ***op2***.
Например кодът

```
int n1 = -8, n2 = 29;
cout << "n1 = " << n1 << " ; n2 = " << n2 << endl;
__asm {
  push esi
    mov esi, n1
    xchg esi, n2
    mov n1, esi
  pop esi
}
cout << "n1 = " << n1 << " ; n2 = " << n2 << endl;
```

извежда на екрана

```
n1 = -8 ; n2 = 29
n1 = 29 ; n2 = -8
```

xor op1, op2 – записва в ***op1*** поразрядното разделително или (XOR) на ***op1*** и ***op2***.

Списък на инструкциите по групи според предназначението

→ Съдържание

Аритметични инструкции за допълнителен код и код без знак

- `aam` (делене с 10)
- `adc` (сумиране заедно с CF)
- `add` (сумиране)
- `cmp` (сравняване чрез изваждане)
- `cmpxchg` (размяна според сравняване чрез изваждане)
- `dec` (намаляване с 1)
- `div` (делене с код без знак)
- `idiv` (делене с допълнителен код)
- `imul` (умножение с допълнителен код)
- `inc` (увеличаване с 1)
- `mul` (умножение с код без знак)
- `neg` (смяна на знака с допълнителен код)
- `sbb` (изваждане заедно с CF)
- `sub` (изваждане)
- `xadd` (сумиране с обратен ред)

Логически инструкции (поразрядни)

- `and` (конюнкция; и)
- `andn` (конюнкция; и)
- `not` (отрицание)
- `or` (дизюнкция)
- `test` (сравняване чрез конюнкция)
- `xor` (разделително или)

Инструкции за работа със стека

- `lahf` (записване на младшия байт на FLAGS в *ax*)
- `pop` (извличане от стека)
- `popa` (извличане от стека на *ax*, *bx*, *cx*, *dx*, *si*, *di*, *sp*, *bp*)
- `popad` (извличане от стека на *eax*, *ebx*, *ecx*, *edx*, *esi*, *edi*, *esp*, *ebp*)
- `popf` (извличане от стека на FLAGS)
- `popfd` (извличане от стека на EFLAGS)
- `popfq` (извличане от стека на RFLAGS)
- `push` (включване в стека)
- `pusha` (включване в стека на *ax*, *bx*, *cx*, *dx*, *si*, *di*, *sp*, *bp*)
- `pushad` (включване в стека на *eax*, *ebx*, *ecx*, *edx*, *esi*, *edi*, *esp*, *ebp*)
- `pushf` (включване в стека на FLAGS)
- `pushfd` (включване в стека на EFLAGS)
- `pushfq` (включване в стека на RFLAGS)
- `sahf` (записване на *ax* в младшия байт на FLAGS)

Инструкции за изместванията

- `rcl` (ротация наляво през CF)
- `rcr` (ротация надясно през CF)
- `rol` (ротация наляво)
- `ror` (ротация надясно)
- `sal` (аритметично изместване наляво)
- `sar` (аритметично изместване надясно)
- `shl` (логическо изместване наляво)
- `shld` (двойно изместване наляво)
- `shr` (логическо изместване надясно)
- `shrd` (двойно изместване надясно)

Инструкции за работа с отделни разряди

- `bsf` (търсене на единица отдясно наляво)
- `bsr` (търсене на единица отляво надясно)
- `bt` (извличане на бит в CF)
- `btc` (извличане на бит в CF и инвертиране)
- `btr` (извличане на бит в CF и нулиране)
- `bts` (извличане на бит в CF и запис на единица)

Инструкции за работа с флагове

- `clc` (нулиране на CF)
- `cld` (нулиране на DF)
- `cmc` (инвертиране на CF)
- `lahf` (записване на младшия байт на FLAGS в *ax*)
- `popf` (извличане от стека на FLAGS)
- `popfd` (извличане от стека на EFLAGS)
- `popfq` (извличане от стека на RFLAGS)
- `pushf` (включване в стека на FLAGS)
- `pushfd` (включване в стека на EFLAGS)
- `pushfq` (включване в стека на RFLAGS)
- `sahf` (записване на *ax* в младшия байт на FLAGS)
- `set` (записване в байт на условие според флагове)
- `stc` (записване на единица във CF)
- `std` (записване на единица във DF)

Инструкции за предаване на управлението (преходи)

- `call` (извикване на процедура)
- `int` (предизвикване на препълване)
- `j`... (условен преход)
- `jmp` (безусловен преход)
- `loop` (намаляване на *ecx* и преход при *ecx*!=0)
- `loope` (намаляване на *ecx* и преход при *ecx*!=0 и ZF=1)
- `loopne` (намаляване на *ecx* и преход при *ecx*!=0 и ZF=0)
- `loopnz` (намаляване на *ecx* и преход при *ecx*!=0 и ZF=0)
- `loopz` (намаляване на *ecx* и преход при *ecx*!=0 и ZF=1)

`ret` (връщане от процедура с възможно изчистване на стека)

Инструкции за преобразувания и прехвърляния

`bswap` (пренареждане на байтове в обратен ред)
`cbw` (разширяване на *al* до *ax*)
`cdq` (разширяване на *eax* до *edx:eax*)
`cwd` (разширяване на *ax* до *dx:ax*)
`cwde` (разширяване на *ax* до *eax*)
`lea` (записване на адрес в регистър)
`mov` (преместване)
`movsx` (преместване на допълнителен код от 8 или 16 разряда с разширяване)
`movsxd` (преместване на допълнителен код от 32 разряда с разширяване)
`movzx` (преместване на код без знак от 8 или 16 разряда с разширяване)
`xchg` (размяна на стойности)

Низови инструкции (за работа с вектори от елементи по 1, 2, 4 или 8 байта)

`cmpsb` (сравняване чрез изваждане на два байта)
`cmpsd` (сравняване чрез изваждане на две двойни думи)
`cmpsq` (сравняване чрез изваждане на две четворни думи)
`cmpsw` (сравняване чрез изваждане на две думи)
`lodsb` (записване на байт в *al*)
`lodsd` (записване на двойна дума в *eax*)
`lodsq` (записване на четворна дума в *rax*)
`lodsw` (записване на дума в *ax*)
`movsb` (прехвърляне на байт)
`movsd` (прехвърляне на двойна дума)
`movsq` (прехвърляне на четворна дума)
`movsw` (прехвърляне на дума)
`rep` (повторение при *ecx*!=0 с намаляване на *ecx*)
`repe` (повторение при *ecx*!=0 и ZF=1 с намаляване на *ecx*)
`repne` (повторение при *ecx*!=0 и ZF=0 с намаляване на *ecx*)
`repnz` (повторение при *ecx*!=0 и ZF=0 с намаляване на *ecx*)
`repz` (повторение при *ecx*!=0 и ZF=1 с намаляване на *ecx*)
`scasb` (сравняване на байтове чрез изваждане от *al*)
`scasd` (сравняване на двойни думи чрез изваждане от *eax*)
`scasq` (сравняване на четворни думи чрез изваждане от *rax*)
`scasw` (сравняване на думи чрез изваждане от *ax*)
`stosb` (записване на *al* в байт)
`stosd` (записване на *eax* в двойна дума)
`stosq` (записване на *rax* в четворна дума)
`stosw` (записване на *ax* в дума)

Инструкции за работа с кодове с плаваща запетая

`f2xm1` (2^y-1)
`fabs` (абсолютна стойност)
`fadd` (събиране)
`faddp` (събиране)
`fchs` (инвертиране на знака)
`fcom` (сравняване)
`fcomi` (сравняване)
`fcomip` (сравняване)
`fcomp` (сравняване)
`fcompp` (сравняване)
`fcos` (косинус)
`fdecstp` (st(0) става st(1)))
`fdiv` (делене)
`fdivp` (делене)
`fdivr` (делене)
`fdivrp` (делене)
`fiadd` (сумиране)
`ficom` (сравняване)
`ficomp` (сравняване)
`fidiv` (делене)
`fidivr` (делене)
`fild` (включване в стека)
`fimul` (умножение)
`fincstp` (st(1) става st(0)))
`fist` (запис на стойност)
`fistp` (запис на стойност)
`fisttp` (закръгляне към цяло)
`fisub` (изваждане)
`fisubr` (изваждане)
`fld` (включване в стека)
`fld1` (включване на 1.0 в стека)
`fldl2e` (включване на $\log_2(e)$)
`fldl2t` (включване на $\log_2(10)$)
`fldlg2` (включване на $\log_2(e)$)
`fldln2` (включване на $\log_e(2)$)
`fldpi` (включване числото пи)
`fldz` (включване на 0.0 в стека)
`fmul` (умножение)
`fmulp` (умножение)
`fprem` (остатък от делене)
`fprem1` (остатък от делене)
`fptan` (тангенс)
`frndint` (закръгляне до цяло)
`fscale` ($st(0)*2^{frndint(st(1))}$)
`fsin` (синус)
`fsincos` (синус и косинус)
`fsqrt` (квадратен корен)
`fst` (запис на стойност)
`fstp` (запис на стойност)
`fstsw` (записване думата на състоянието)
`fsub` (изваждане)
`fsubp` (изваждане)
`fsubr` (изваждане)
`fsubrp` (изваждане)
`ftst` (сравняване)
`fxch` (размяна стойности)
`fyl2x` ($st(1)*\log_2(st(0))$)
`fyl2xp1` ($st(1)*\log_2(st(0)+1)$)

Азбучен указател

→ Съдържание