Основна більшість цих команд НЕ розрізняє знакові і беззнакові дані - це прерогатива людини. Знак "відповідає" найстарший біт числа в його внутрішньому (машинному) поданні. Чи враховувати його як знак або як інформаційну частину числа вирішує людина.

Команди пересилання і обміну інформацією.

Це найпростіші і найпоширеніші команди. Насправді, команд пересилань в Асемблері набагато більше, але розглянемо базові команди.

Команди Ассемблера містять мнемокод, покликаний полегшити розуміння людиною даної команди.

1. Команда пересилки MOV

Мнемокод цієї команди отриманий в результаті скорочення такої пропозиції: MOVe operand to/from system registers – Пересилання операнда в/з системних регістрів.

Команда ця містить два операнда і має наступний формат (синтаксис):

MOV Destination, Source

Це ДУЖЕ поширений формат команд і, якщо команда містить два операнда, вони майже завжди інтерпретуються саме так:

перший операнд – приймач, а другий – джерело.

В даному конкретному випадку інформація з джерела пересилається (копіюється) в приймач. Ця команда в найпростішій своїй інтерпретації відповідає оператору присвоювання (вміст джерела <Джерело> копіюється в приймач):

Приймач: = <Джерело>

ДОВЖИНИ ОБОХ операндів повинні бути однаковими.

2. Команда обміну даними XCHG

Команда використовується для двунаправленої пересилки даних. Для цієї операції можна, звичайно, застосувати послідовність із декількох команд MOV, але через те, що операція обміну використовується досить часто, розробники системи команд процесора вирішили ввести окрему команду обміну XCHG.

Команда XCHG міняє місцями вміст двох операндів (eXCHanGe).

Синтаксис:

XCHG Приймач, Джерело

Існує три варіанта команди XCHG:

XCHG Register Register

XCHG Register Memory

XCHG Memory Register

Для операндів команди XCHG необхідно дотримуватися тих же правил і обмежень, що і для операндів команди MOV, за виключенням того, що операнди команди XCHG не можуть бути безпосередньо заданими числами.

Приклади використання команди XCHG:

1. Обмін вмісту 16-розрядних регістрів:

xchg AX, BX

2. Обмін вмісту 8-розрядних регістрів:

xchg AH, AL

3. Обмін вмісту 16-розрядного операнда в пам’яті і регістра BX:

xchg VAR1, BX

4. Обмін вмісту 32-розрядних регістрів:

xchg EAX, EBX

Арифметичні команди.

Ця група команд дозволяє розібратись, як IBM PC виконує арифметичні операції з цілочисельними даними довжиною 8 і 16 біт (частково і з даними довжиною 32 біти). Всі ці команди сигналізують про результат процесу обрахування, заносячи відповідні значення у відповідні біти регістра.

1. Команди додавання

Ці команди використовуються для додавання чисел в двійковій системі числення. Якщо в результаті додавання результат не помістився у відповідне місце, виробляється позначка переносу CF=1.

У цьому випадку говорять, що позначка CF встановилася. Є ще 4 позначки (табл.1):

Стан прапорців після виконання команд додавання

Прапорець	Призначення	Пояснення
CF=1	прапорець перенесення	Результат додавання не помістився в операнд-приймач
PF=1	прапорець парності	Результат додавання має парну кількість бітів із значенням 1
SF=1	прапорець знаку	Копіюється СТАРШИЙ (ЗНАКОВИЙ) біт результату додавання
ZF=1	прапорець нуля	Результат додавання дорівнює нулю
OF=1	прапорець переповнення	При додаванні двох чисел з ОДНАКОВИМ знаком (два додатні або два від'ємні) результат додавання вийшов БІЛЬШЕ ніж допустиме значення. В цьому випадку приймач МІНЯЄ ЗНАК.

1.1 Команда ADD

Найпростіша із команд додавання – ADD (ADDition - додавання).

Синтаксис:

ADD Приймач, Джерело

Логіка роботи команди:

<Приймач> = <Приймач> + <Джерело>

Можливі поєднання операндів для цієї команди аналогічні команді MOV.

Нехай у нас є два числа – 120 і 100. Щоб їх додати. Необхідно виконати наступні дії:

```
mov al, 120 ; al <=== 120
mov cl, 100 ; cl <=== 100
add al, cl  ; <al>:=<al>+<cl>
mov x, al  ; x <=== <al>
```

1.2 Команда ADC

Команда ADC (ADdition with Carry) – додавання з перенесенням, відрізняється від команди ADD використанням біта переносу CF при додаванні. Тому вона може використовуватися при складанні багатобайтних цілих чисел, довжина яких більша, ніж розрядність регістрів процесора який використовується.

Синтаксис:

ADC Приймач, Джерело

Логіка роботи:

<Приймач> = <Приймач> + <Джерело> + <CF>

На рис 1 показано додавання двох двійкових чисел командою ADD:

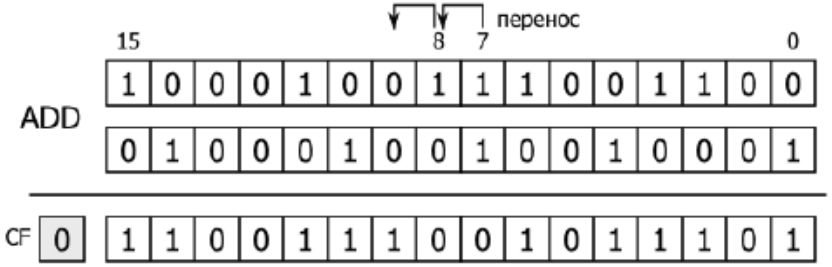

Рис. 1. Додавання двох 16-розрядних чисел 35276 + 17553 = 52829 командою ADD.

При додаванні відбувається перенесення з 7-го розряду в 8-й, саме на межі між байтами. Якщо ми будемо складати ці числа частинами командою ADD, то перенесений біт загубиться і в результаті ми отримаємо помилку. На щастя, перенесення зі старшого розряду завжди зберігається у прапорі CF. Щоб додати цей перенесений біт, достатньо застосувати команду ADC. Зазвичай ця команда працює у парі з командою ADD (складання молодших частин числа).

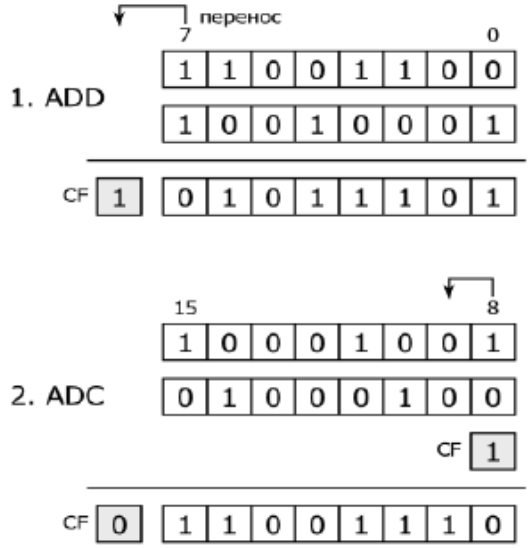

Рис. 2. – Додавання двох 16-розрядних чисел
35276 + 17553 = 52829 командою ADC.

1.3 Команда INC

Мнемокод цієї команди отриманий в результаті скорочення такого речення: INCrement operand by 1 – Збільшення значення операнда на 1. Ця команда містить один операнд і має наступний синтаксис:

INC Операнд

Логіка роботи команди:

<Операнд> = <Операнд> + 1

В якості операнда допустимі регістри і пам’ять: R8, R16, Mem8, Mem16.

Наприклад:

inc I ; I++

inc ax ; <ax> = <ax> + 1

inc cl ; <cl> = <cl> + 1

2 Команди віднімання

Ці команди обернені відповідним командам додавання. Вони мають ті ж операнди.

2.1 Команда SUB

Мнемокод цієї команди отриманий в результаті скорочення слова SUBstraction – віднімання. Її синтаксис:

SUB Приймач, Джерело

При виконанні віднімання треба бути особливо уважним до стану позначок, тому що коли від одного числа віднімається інше, можливість отримати від’ємне число в якості результата суттєво більша. Тому потрібно відслідковувати стан позначки SF і, якщо вона буде встановлена, то приймати необхідні міри, якщо в цьому буде необхідність.

Приклад використання команди:

mov al, 10

sub al, 7 ; al = 3; al – приймач, 7 – джерело.

2.2 Команда SBB

Команда SBB (SuBtract with Borrow) CF – віднімання із перенесенням (позикою) прапора CF. Прцює аналогічно команді ADC.

Синтаксис:

SBB Приймач, Джерело

Логіка роботи:

<Приймач> = <Приймач> – <Джерело> – <CF>

2.3 Команда DEC

Команда DEC зменшує на одиницю вміст приймача. Її синтаксис:

DEC Операнд

Вона еквівалентна команді:

SUB Приймач, 1

Логіка роботи команди:

<Операнд> = <Операнд> - 1

Приклад використання команди:

mov al, 15

dec al; al = 14

2.3 Команда NEG

Команда NEG (NETate operating) – зміна знака операнда. Її синтаксис:

NEG Операнд

Логіка роботи команди:

<Операнд> = - <Операнд>

В якості операнда допустимі регістри R8, R16, Mem8, Mem16. Наприклад, для обчислення <AL> = =50 - <AL> можна використати такий код:

neg al

add al, 50

Як бачимо, операцію віднімання замінили додаванням. Це пов’язано з тим, що операція віднімання повільніша, ніж операція додавання.

Розрахунок 16 бітних виразів.

Приклад. Розрахувати значення виразу

x = (a + b +10) – (b+c+d-e) + 20, де x, a, b, c, d, e – 16-бітні знакові.

Потрібно звернути увагу, що у нас обмежені ресурси, і ми маємо лише чотири 16-бітні регістри (AX, BX, CX, DX).

Розв’язок.

```
int main ()
{ short int a, b, c, d, e, x_c, x_asm;
 printf("a = "); scanf_s("%d", &a);
 printf("b = "); scanf_s("%d", &b);
 printf("c = "); scanf_s("%d", &c);
 printf("d = "); scanf_s("%d", &d);
 printf("e = "); scanf_s("%d", &e);
 x_c = (a + b +10) – (b+c+d-e) + 20;
 printf("Result C x_c = %d\n", x_c);
__asm {
        ; Варіант 1
        MOV AX, a           ; <AX>=a
        MOV BX, b           ; <BX>=b
        ADD AX, BX          ; <AX>=a+b
        MOV BX, 10          ; <BX>=10
        ADD AX, BX          ; <AX> = a+b+10
        MOV BX, b           ; <BX>=b
        MOV CX, c           ; <CX>=c
        ADD BX, CX          ; <BX>= b+c
        MOV CX, d           ; <CX> = d
        ADD BX, CX          ; <BX> = b+c+d
        MOV CX, e           ; <CX> = e
        SUB BX,CX           ; <BX> = b+c+d-e;
        SUB AX, BX          ; <AX> = (a+b+10)-(b+c+d-e)
        MOV BX, 20          ; <BX> = 20
        ADD AX, BX          ; <AX> = (a+b+10)-(b+c+d-e) + 20
        MOV x_asm, AX       ; <AX> = (a+b+10)-(b+c+d-e) + 20
}
 printf("Result ASM x_asm = %d\n", x_asm);
```

```
 system("Pause");
 return 0;
}

; Варіант 2
        MOV AX, a           ; <AX>=a
        MOV BX, b           ;<BX>=b
        ADD AX, BX          ; <AX>=a+b
        ADD AX, 10          ; <AX>=a+b+10
        ADD BX, c           ; <BX>=b+c
        ADD BX, d           ; <BX> = b+c+d
        SUB BX, e           ; <BX> = b+c+d-e;
        SUB AX, BX          ; <AX> = (a+b+10)-(b+c+d-e)
        ADD AX, 20          ; <AX> = (a+b+10)-(b+c+d-e) + 20
        MOV x_asm, AX       ; <AX> = (a+b+10)-(b+c+d-e) + 20
```

Оптимізація самих виразів

Початковий вираз

x = (a + b +10) – (b+c+d-e) + 20,

Оптимізований вираз

x = **a + b +10 – b – c – d + e + 20 = a – c – d + e +30**

Код програми після оптимізації

```
        …
        MOV AX, a           ; <AX>=a
        SUB AX, c           ; <AX>=a-c
        SUB AX, d           ; <AX>=a-c-d
        ADD AX, e           ; <AX>=a-c-d+e
        ADD AX, 30          ; <AX> = a-c-d+e+30
        MOV x_asm, AX       ; x_asm = (a+b+10)-(b+c+d-e) + 20 = a-c-d+e+30
```

При виконанні операції множення операнди повинні мати однакову довжину, а результат довжину в два рази більшу:

8 біт * 8 бітів = 16 бітів
16 бітів * 16 бітів = 32 бітів
32 біти * 32 біти = 64 біти
64 біти * 64 біти = 128 біти

При виконанні операції ділення ділене повинне мати довжину в два рази більше ніж дільник та частка:

16 бітів : 8 бітів = 8 бітів
32 біти : 16 бітів = 16 бітів
64 біти : 32 біти = 32 біти
128 бітів : 64 біти = 64 біти

Ресурси

Операції множення та ділення «прив’язані» до регістрів:

8 бітне множення та ділення – регістри AL, AX

16 бітне множення та ділення – регістри AX, <DX:AX> (EAX)

32 бітне множення та ділення – регістри EAX, <EDX:EAX> (RAX)

Команди перетворення типів

cbw – розширює регістр AL до регістра AX

cwd – розширює регістр AX до регістра комбінованого регістра <DX:AX> (EAX)

cdq – розширює регістр EAX до комбінованого регістра <EDX:EAX> (RAX)

Команди «розширення» регістрів

Беззнакові числа займають всю комірку пам'яті, поняття знак для них не існує – вони вважаються додатніми. Тому при діленні беззнакових чисел в старшу частину діленого треба занести нуль. Це можна зробити уже відомими нам командами: mov ah, 0 або mov dx, 0.

Але це не ефективно, тому ми використовуємо рідну для комп'ютера команду додавання по модулю 2 – xor ah, ah або xor dx, dx.

Для знакових даних існують дві команди розширення знака. CBW (Convert Byte to Word – перетворити байт, який знаходиться в регістрі AL, в слово – регістр AX) і CWD (Convert Word to Double word – перетворити слово, яке знаходиться в регістрі AX в подвійне слово – регістри <DX:AX>). Операнди їм не потрібні.

Синтаксис:

CBW

CWD

CBW (AL -> AX)

CWD (AX -> DX:AX)

CWDE (AX -> EAX)

CDQ (EAX -> EDX:EAX)

Наприклад. Перетворення 8-бітного значення в 16-бітне +/- 7

Для додатнього «7»

AL: 00000111

AX: AH:00000000 AL:00000111

Для від'ємного «-7» (зберігається в додатковому коді)

AL: 11111001 – 8 бітів знакове

AX: AH: 11111111 AL:11111001 – 16 бітів знакове

Команди множення та ділення

Команди множення MUL (MULtiply – беззнакове множення) та IMUL (Integer MULtiply – знакове цілочисельне множення) містять неявні операнди. Як видно із визначення команд, вони враховують або не враховують наявність знака.

Беззнакове множення

MUL Операнд 2 (Джерело)

Знакове множення

IMUL Операнд 2 (Джерело)

Неявні операнди команд MUL, IMUL

Довжина джерела	Множник	Добуток
Byte	AL AX	(<AH:AL>)
Word	AX	<DX:AX>
DWord	EAX	<EDX:EAX>

Логіка роботи команди

Добуток = Множене x Операнд 2 (Джерело)

Довжина Добутку завжди в два рази більша, ніж у Множника.

Причому старша частина Добутку знаходиться або в регістрі AH, або в DX або в EDX.

Ці команди також виробляють прапорці CF=1 та OF=1. Але вони встановлюються тільки тоді, коли результат занадто великий для відведених йому регістрів призначення.

Множене – знаходиться в регістрі AL (8 бітів), AX (16 бітів) EAX (32 біти)…

Добуток – зберігається в регістрі AX (пара регістрів AH:AL для 8-бітного множення), DX:AX (16 бітне множення)

Операнд 2 – або регістр, або пам'ять (змінна), константи (числа) не допускаються.

Команди ділення DIV (DIVide – беззнакове ділення), IDIV(Integer DIVide – знакове ділення цілих цисел) – також містять неявні операнди та, відповідно, невраховують, або враховують знак.

Беззнакове ділення

DIV Операнд 2

Знакове ділення

IDIV Операнд 2

Неявні операнди команд DIV, IDIV

Довжина джерела (Дільника)	Ділене	Добуток	
		Частка	Залишок
Byte	AX(<AH:AL>)	AL	AH
Word	<DX:AX>	AX	DX
DWord	<EDX:EAX>	EAX	EDX

Логіка роботи команди

<Частка:Залишок>=<Ділене> / Операнд 2 (Джерело_Дільник)

Ділене – знаходиться в регістрі AX (8-бітне ділення), DX:AX (16 бітне ділення …

Результат – зберігається в регістрі AL (8-бітне ділення), AX (16 бітне ділення) …

Операнд 2 – або регістр, або пам’ять (змінна), константи (числа) не допускаються.

Довжина діленого завжди в два рази більша, ніж у Дільника. Причому старша частина Діленого знаходиться в регістрі AH, або в DX, або в EDX. Ці команди також виробляють прапорці CF=1 та OF=1. Але вони встановлюються коли частка не поміщається в регістри AL, AX або EAX. Тут, на відміну від команд множення, завжди генерується переривання «Ділення на нуль», хоча на нуль і не ділимо.

Додаткові команди

SHL (Logical Shift Left), SAL (Arithmetic Shift Left), SHR, SAR – команди логічного та арифметичного зсуву ліворуч та праворуч на певну кількість бітів (можуть застосовуватися для беззнакових чисел для виконання множення/ділення на 2^x(2, 4, 8).

$00000100_2 = 4_{10}$

Операція SAL AL, 2

$00010000_2 = 16_{10}$

$00000100_2 = 4_{10}$

Операція SAR AL, 2

$00000001_2 = 1_2$

Для знакових чисел в командах логічного та арифметичного зсуву вправо є різниця в збереженні знаку.

$11111001_2 = -7_{10}$ (в додатковому коді) (або 249_{10} в прямому)

SHR AL, 1; сприймає дані як беззнакове число в прямому коді

$01111100_2 = 124_{10}$

SAR AL, 1; сприймає дані як від’ємне число в додатковому коді

10000111_2 –> $10000011_2 = -4_{10}$ –> 11111101_2

При зміщенні додатнього знакового числа вліво ми отримуємо не коректний результат, оскільки втрачаємо старший біт

Приклад 1.

Розрахувати значення виразу **x = (a+5)/b + c/d**, всі числа 8-бітні знакові

x = (16 бітів)/8 бітів + (16 бітів)/(8 бітів) = 8 бітів

```
__asm {
        MOV AL, A ; <AL> = a
        ADD AL, 5 ; <AL> = a + 5
        CBW  ; <AX> = a + 5
        MOV BL, b ; <BL> = b
        IDIV BL ; <AL> = (a+5)/b
        MOV CL, AL ; <CL> = (a+5)/b, <AL> = (a+5)/b, <BL> = b
        MOV AL, c ; <AL> = c, <BL> = b, <CL> = (a+5)/b
        CBW  ; <AX> = c, <BL> = b, <CL> = (a+5)/b
        MOV BH, d ; <AX> = c, <BL> = b, <BH> = d, <CL> = (a+5)/b
        IDIV BH ; <AL> = c/d, <BL> = b, <BH> = d, <CL> = (a+5)/b
        ADD CL, AL ; <CL> = (a+5)/b + c/d, <AL> = c/d, <BL> = b, <BH> = d
        MOV x, CL ; x = (a+5)/b + c/d
        }
```

Приклад 2.

Розрахувати значення виразу **x = (a+d+5)/b + a/(d*2)**, всі числа 16-бітні знакові

x = (16 бітів -> 32 бітів)/16 бітів + (16 -> 32 бітів)/((16 бітів *16 бітів = 32 біти) / 16 бітну одиницю) = 8 бітів

```
__asm {
        MOV AX, a ; <AX> = a
        MOV CX, d ; <CX> = d
        ADD AX, CX ; <AX> = a + d, <CX> = d
        ADD AX, 5 ; <AX> = a + d + 5, <CX> = d
        CWD  ; <DX:AX> = a + d + 5, <CX> = d
        MOV BX, b ; <DX:AX> = a + d + 5, <CX> = d, <BX>=b
        IDIV BX ; <AX> = (a + d + 5)/b , <CX> = d, <BX>=b
        XCHG AX, CX ; <AX> = d, <CX> = (a + d + 5)/b, <BX>=b
        MOV BX, 2 ; <AX> = d, <CX> = (a + d + 5)/b, <BX>=2
        IMUL BX  ; <DX:AX> = d*2, <CX> = (a + d + 5)/b, <BX>=2
        DEC BX  ; <DX:AX> = d*2, <CX> = (a + d + 5)/b, <BX>=1
        IDIV BX  ; <AX> = d*2, <CX> = (a + d + 5)/b, <BX>=1
        MOV BX, a ; <AX> = d*2, <CX> = (a + d + 5)/b, <BX>=a
        XCHG AX, BX ; <AX> = a, <CX> = (a + d + 5)/b, <BX>= d*2
        CWD  ; <DX:AX> = a, <CX> = (a + d + 5)/b, <BX>= d*2
        IDIV BX  ; <AX> = a/(d*2), <CX> = (a + d + 5)/b, <BX>= d*2
        ADD CX, AX ; <CX> = (a + d + 5)/b + a/(d*2) …
        MOV x, CX
        }
```