Базові типи даних

В мові асемблера використовуються базові типи даних **відповідно до архітектури процесорів**.
Базовими типами даних для процесорів сімейства x86 є:

- **байт** (byte) який складається з 8 бітів;
- **слово** (word) складається з двох байтів і має розмір 16 біт;
- **подвійне слово** (doubleword) складається з двох слів і має розмір 32 біт;
- **квадрослово** (quadword) складається з двох подвійних слів і має розмір 64 біт. Цей тип було впроваджено у архітектурі починаючи з процесора Intel486;
- **подвійне квадрослово** (doublequadword) складається з двох квадрослів і має розмір 128 біт. Цей тип було впроваджено у процесорі Pentium III разом із розширенням SSE;
- **256-бітні слова**, які складаються з двох подвійних квадрослів. Цей тип даних з'явився у 2008 завдяки розширенню AVX.

Розташування даних у пам'яті

Порядок розташування бітів

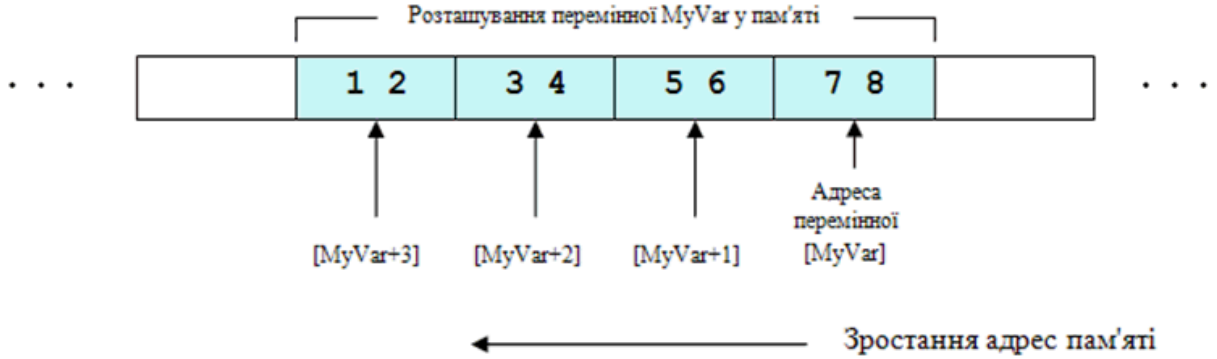
У архітектурі сімейства Intel x86 прийнято порядок розташування даних від молодших розрядів к старшим. Молодший байт (біти від 0 до 7) розташовуються по найменшій адресі – ця адреса буде адресою відповідного операнду, який має тип слова, подвійного слова, квадрослова або подвійного квадрослова.

У якості прикладу розглянемо, як розташовується у пам'яті 32-бітове число 12345678h, яке записано у перемінну MyVar:

```
.data
  MyVar dd 12345678h
.code

  xor eax, eax                      ;обнулюємо регістри
  xor ebx, ebx
  xor ecx, ecx
  xor edx, edx
  mov al, byte ptr [MyVar]          ;AL = 78h   (молодший байт числа)
  mov bl, byte ptr [MyVar+1]        ;BL = 56h
  mov cl, byte ptr [MyVar+2]        ;CL = 34h
  mov dl, byte ptr [MyVar+3]        ;DL = 12h   (старший байт числа)
```

Таким чином, адреса перемінної MyVar є адресою її молодшого байту, який містить значення 78h.



Таке розташування даних зветься "little_endian", або "little-end-first", або "low-end-first". На відміну цього, у процесорах інших архітектур, наприклад, Motorola порядок розташування даних протилежний – від старших бітів к молодшим ("big_endian", або "big-end-first", або "high-end-first").

Вирівнювання даних у пам'яті. Директива ALIGN

Загалом, слова, подвійні слова тощо можуть розташовуватися у пам'яті за будь-якими адресами. Вирівнювання означає розташування даних за адресами, кратними 2, 4, 8 або 16. Часто, наприклад, при запису даних у стек, необхідно розташовувати дані за адресами, кратними 4.

Ігнорування вимог вирівнювання призводить до зайвих тактів доступу до пам'яті.

Для деяких команд вирівнювання даних обов'язкове. Наприклад, команди, які обробляють подвійні квадрослова, потребують того, щоб ці операнди були записані по адресам, кратним 16 – інакше операція буде аварійно завершуватися.

У мові асемблеру є директива ALIGN, яка вказує потрібне вирівнювання при створенні даних. Наприклад:

```
.data
  var1 db 255       ;перемінна може мати таку адресу: 00404000h
  var2 db 255       ;для цієї перемінної така адреса: 00404001h
  var3 db 255       ;для цієї перемінної така адреса: 00404002h
  var4 db 255       ;для цієї перемінної така адреса: 00404003h
```

Вирівнювання по адресам, кратним 4:

```
.data
align 4
  var1 db 255       ;адреса: 00404000h     (вирівнювання є)
align 4
  var2 db 255       ;адреса: 00404004h     (вирівнювання є)
  var3 db 255       ;адреса: 00404005h     (без вирівнювання)
  var4 db 255       ;адреса: 00404006h     (без вирівнювання)
```

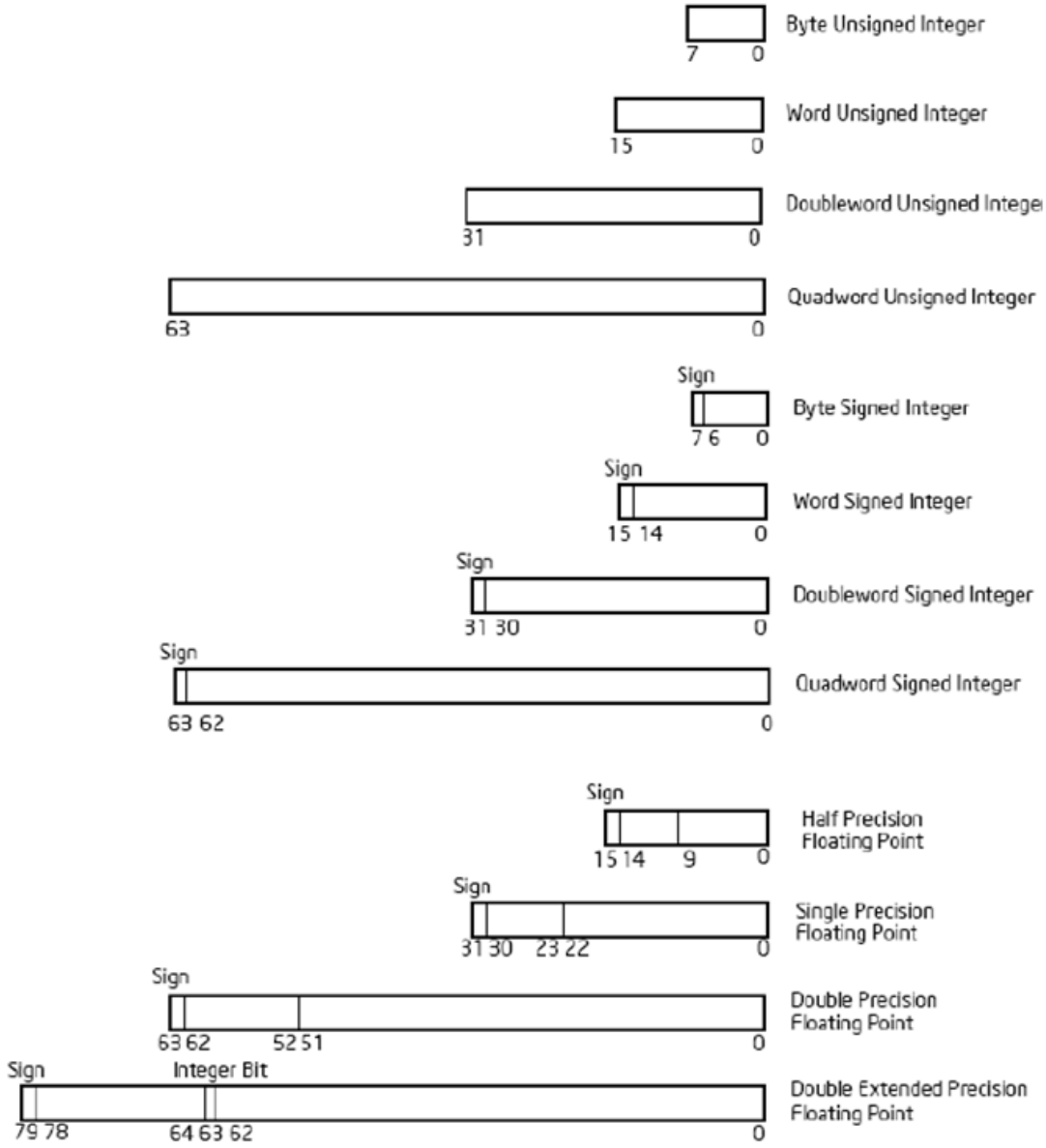
Ще приклад:

```
.data
align 4
  var1 db 255       ;адреса: 00404000h     (вирівнювання на 4)
align 16
  var2 db 255       ;адреса: 00404010h     (вирівнювання на 16)
align 4
  var3 db 255       ;адреса: 00404014h     (вирівнювання на 4)
  var4 db 255       ;адреса: 00404015h     (без вирівнювання)
```

Числові типи даних

Хоча байт, слово, подвійне слово, квадрослово та подвійне квадрослово є базовими типами даних, пр
виконанні кожної команди процесор інтерпретує ці порції бітів як кодовані числові значення у різн
форматах відповідно командам. Наприклад, подвійне слово (32 біти) може містити або ціле зі знаком, аб
ціле без знаку, або 32-бітове число у форматі з плаваючою точкою.

В архітектурі x86 прийнято такі типи числових даних:



З документації Intel

Типи для цілих чисел

В архітектурі x86 прийнято два типи, які можна використовувати для представлення цілих чисел – типи без знаку та типи зі знаком.

Цілі без знаку є звичайними двійковими числами, діапазон значень від 0 до максимального значення, яке можна представити у двійковому коді для певної кількості бітів. Надамо діапазони значень для цілочисельних типів без знаку:

- байтове ціле без знаку: від 0 до 255
- слово без знаку: від 0 до 65535
- подвійне слово без знаку: від 0 до 2^{32}-1
- квадрослово без знаку: від 0 до 2^{64}-1

Цілі числа зі знаком представляються у додатковому двійковому коді (two's complement binary). Діапазони значень для цілих типів зі знаком:

- байтове ціле зі знаком: від -128 до 127
- слово: від -32768 до 32767
- подвійне слово: від -2^{31} до 2^{31}-1
- квадрослово: від -2^{63} до 2^{63}-1

Типи з плаваючою точкою

В архітектурі x86 використовуються такі три основні типи (формати) з плаваючою точкою:

- 32-бітний одинарний формат (згідно стандарту IEEE754);
- 64-бітний подвійний формат (згідно стандарту IEEE754);
- 80-бітний розширений формат (внутрішній, робочий формат блоку x87 FPU).

Окрім цих трьох стандартних форматів, в архітектурі x86 з'явився 16-бітовий формат з плаваючою точкою під назвою Half Precision. Цей формат підтримується тільки в командах розширення F16C (VCVTPH2PS, VCVTPS2PH).

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Half Precision	16	11	2^{-14} to 2^{15}	3.1×10^{-5} to 6.50×10^{4}
Single Precision	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}
Double Precision	64	53	2^{-1022} to 2^{1023}	2.23×10^{-308} to 1.79×10^{308}
Double Extended Precision	80	64	2^{-16382} to 2^{16383}	3.37×10^{-4932} to 1.18×10^{4932}

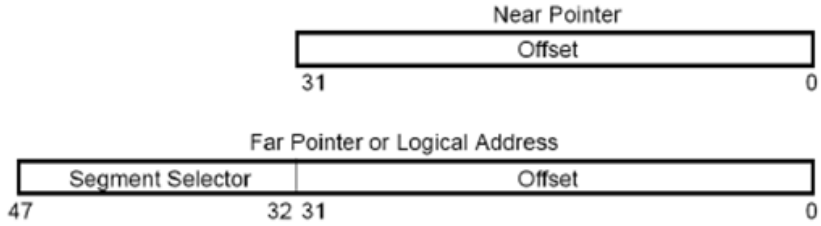
Діапазони типів з плаваючою точкою (дані з керівництва Intel Software Developers Manual)

Вказівник

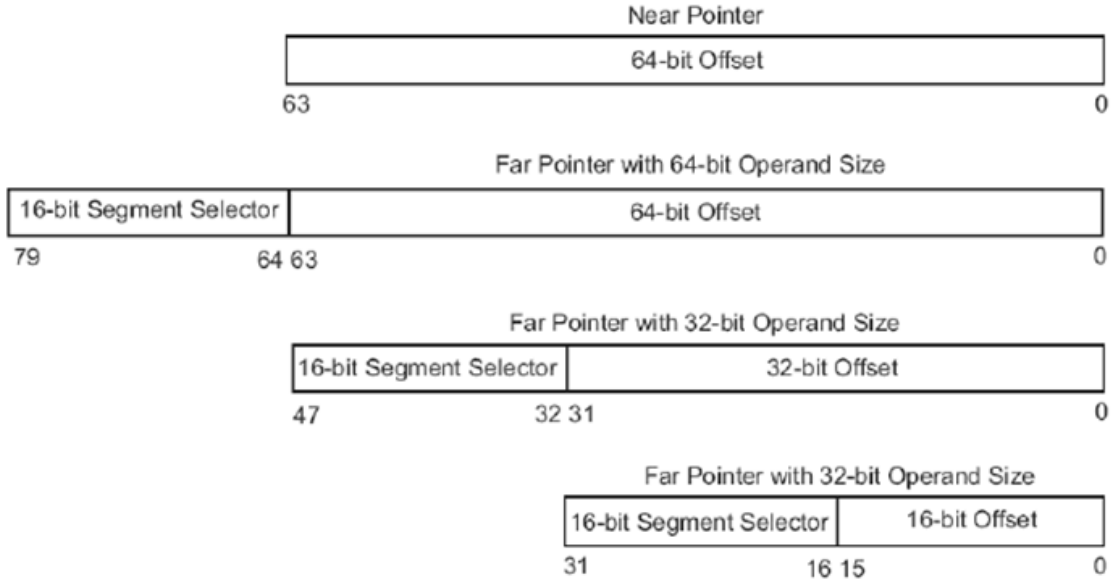
Вказівники (pointers) вказують розташування об'єктів у пам'яті.

Для 32- та 16-бітових режимів є два різновиди вказівників:

- **ближній вказівник** (near pointer). Це 32- або 16-бітовий зсув відносно поточного сегменту. Також ще зветься ефективною адресою. Ближні вказівники використовуються для адресації будь-яких об'єктів у flat моделі пам'яті. Для сегментованої моделі ближній вказівник означає зсув для поточного визначеного сегменту.
- **дальній вказівник** (far pointer). Складається з 16-бітового селектора сегменту та 32- або 16-бітового зсуву відносно початку цього сегменту. Також ще зветься логічною адресою.



Для 64-бітного режиму (sub-mode of IA-32e mode) ближній вказівник 64-бітний. Дальніх вказівників є 3 різновиди.



Огляд системи команд

У процесорів архітектури Intel 64 та IA-32 команди розподілені по наступним групам:

- General purpose
- x87 FPU
- x87 FPU and SIMD state management
- Intel® MMX technology
- SSE extensions
- SSE2 extensions
- SSE3 extensions
- SSSE3 extensions
- SSE4 extensions
- AESNI and PCLMULQDQ
- Intel® AVX extensions
- F16C, RDRAND, FS/GS base access
- FMA extensions
- Intel® AVX2 extensions
- Intel® Transactional Synchronization extensions
- System instructions
- IA-32e mode: 64-bit mode instructions
- VMX instructions
- SMX instructions

Table 5-1. Instruction Groups in Intel 64 and IA-32 Processors

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
General Purpose	All Intel 64 and IA-32 processors
x87 FPU	Intel486, Pentium, Pentium with MMX Technology, Celeron, Pentium Pro, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
x87 FPU and SIMD State Management	Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
MMX Technology	Pentium with MMX Technology, Celeron, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
SSE Extensions	Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
SSE2 Extensions	Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
SSE3 Extensions	Pentium 4 supporting HT Technology (built on 90nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 3xxxx, 5xxx, 7xxx Series, Intel Atom processors
SSSE3 Extensions	Intel Xeon processor 3xxx, 5100, 5200, 5300, 5400, 5500, 5600, 7300, 7400, 7500 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors, Intel Atom processors
IA-32e mode: 64-bit mode instructions	Intel 64 processors
System Instructions	Intel 64 and IA-32 processors
VMX Instructions	Intel 64 and IA-32 processors supporting Intel Virtualization Technology
SMX Instructions	Intel Core 2 Duo processor E6x50, E8xxx; Intel Core 2 Quad processor Q9xxx

З документації Intel

Команда MOV

Ця команда часто використовується у програмах на асемблері. Вона виконує копіювання даних. Команда MOV має два операнди:

```
mov Куди, Джерело
```

Операнд **Джерело** повинен вказувати, звідки взяти інформацію. Перший операнд вказує, куди записати інформацію. Наприклад:

```
mov ecx, eax
```

означає скопіювати дані з регістру EAX у регістр ECX. У наступному рядку

```
mov ax, 5
```

запрограмований запис числа 5 у регістр AX. У якості другого операнду записане безпосередньо числове значення. Такі значення зберігаються у пам'яті, тому фактично виконується копіювання типу "пам'ять → регістр".

Певна кількість байтів джерела повинна записуватися у відповідне за розміром місце. Приклади помилок:

```
mov al, edx       ;помилка: з 32-бітового у 8-бітовий регістр
mov ax, ecx       ;помилка: з 32-бітового у 16-бітовий регістр
mov eax, cx       ;помилка: з 16-бітового у 32-бітовий регістр
```

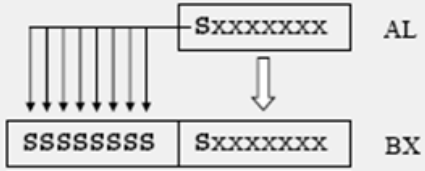
Див. також лаб 2

Команди MOVSX

Копіювання із розширенням знаку (*Move with Sign-Extension*). Розмножується біт знаку.

MOVSX r16, r/m8	Копіювання байту у слово із розширенням знаку
MOVSX r32, r/m8	Копіювання байту в подвійне слово із розширенням знаку
MOVSX r64, r/m8	Копіювання байту в квадрослово із розширенням знаку
MOVSX r32, r/m16	Копіювання слова у подвійне слово із розширенням знаку
MOVSX r64, r/m16	Копіювання слова у квадрослово із розширенням знаку

movsx bx, al



movsx edx, al

Наприклад:

```
.data
  ByteValuePositive db 5
  ByteValueNegative db -5
.code
  mov eax, 0FFFFFFFFh               ;EAX = FFFFFFFF
  mov ebx, 0FFFFFFFFh               ;EBX = FFFFFFFF
  mov edx, 0FFFFFFFFh               ;EDX = FFFFFFFF

  mov al, ByteValuePositive         ;EAX = FFFFFF05
  movsx bx, al                      ;EBX = FFFF0005
  movsx edx, al                     ;EDX = 00000005

  xor eax, eax                      ;EAX = 00000000
  xor ebx, ebx                      ;EBX = 00000000
  xor edx, edx                      ;EDX = 00000000

  mov al, ByteValueNegative         ;EAX = 000000FB
  movsx bx, al                      ;EBX = 0000FFFB
  movsx edx, al                     ;EDX = FFFFFFFB
```

Арифметичні команди

Команда ADD

ADD dest, src

Ця команда додає значення операнду **src** до операнду **dest** і записує результат в dest.

Наприклад

```
  mov eax, 5
  add eax, 12                    ;EAX = EAX + 12. Результат 17
```

```
.data
  value dd 5

.code
  add dword ptr [value], 12      ;до перемінної value додати 12
  add value, 12                  ;те саме – до перемінної value додати 12
```

Незважаючи на зовнішню відмінність, останні два рядка означають однакові дії, оскільки запис

```
add value, 12
```

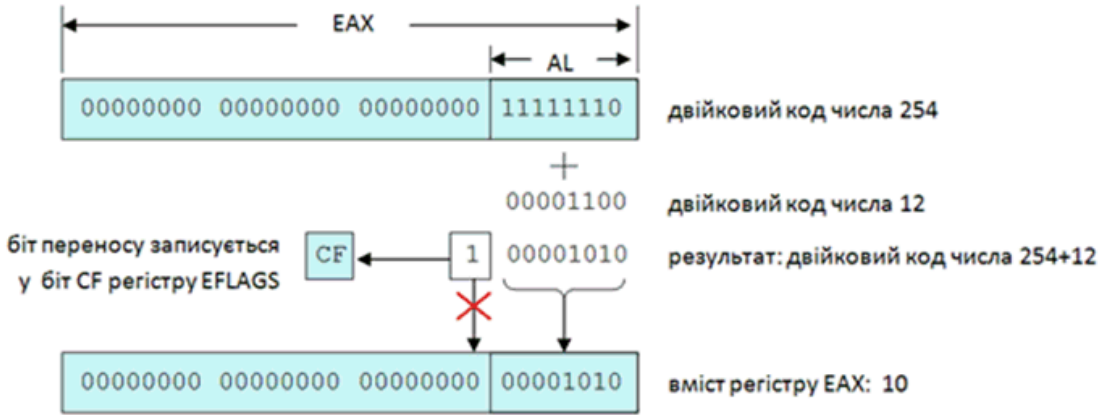
асемблер транслює у

```
add dword ptr [value], 12
```

Наступний приклад.

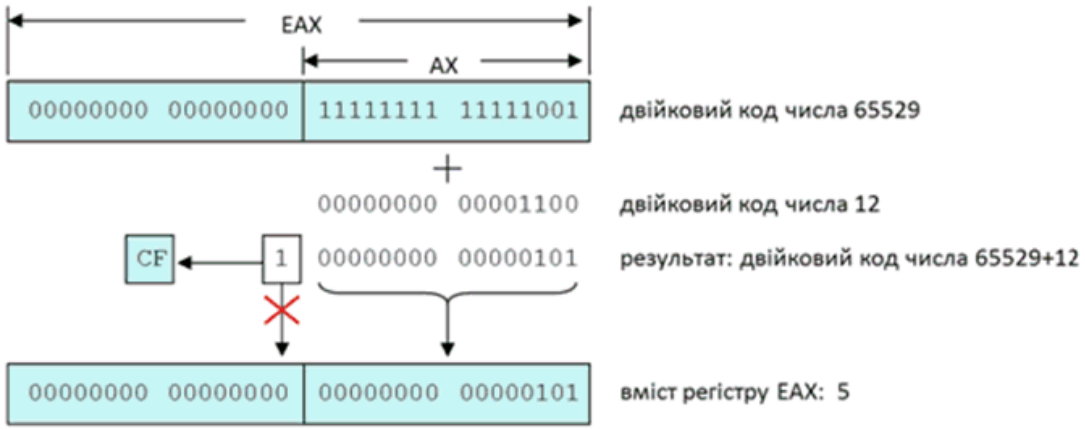
```
mov eax, 254
add al, 12
```

Не зважаючи на те, що AL є молодшою частиною регістру EAX, після додавання до AL числа 12 у регістрі EAX буде значення 10, а не 254+12 = 266. Тобто, AL тут є немов би окремим регістром і розрядна сітка операції додавання фактично обмежується вісьма бітами. Біт переносу буде записано не у дев'ятий біт регістру EAX, а у біт **CF (*carry flag*)** регістру EFLAGS.



Подібна ситуація буде і при вказуванні для команди ADD у якості операнду призначення регістру AX – тоді буде виконуватися **16-бітове додавання**. Наприклад:

```
mov eax, 65529
add ax, 12
```



Команда ADC

Див. також лаб 4

ADC dest, src

Команда додавання, враховуючи перенос (Add with Carry).
Ця команда подібна до команди ADD, но обчислює суму вже трьох значень: dest, src та біту CF регістру EFLAGS:

dest = dest + src + CF

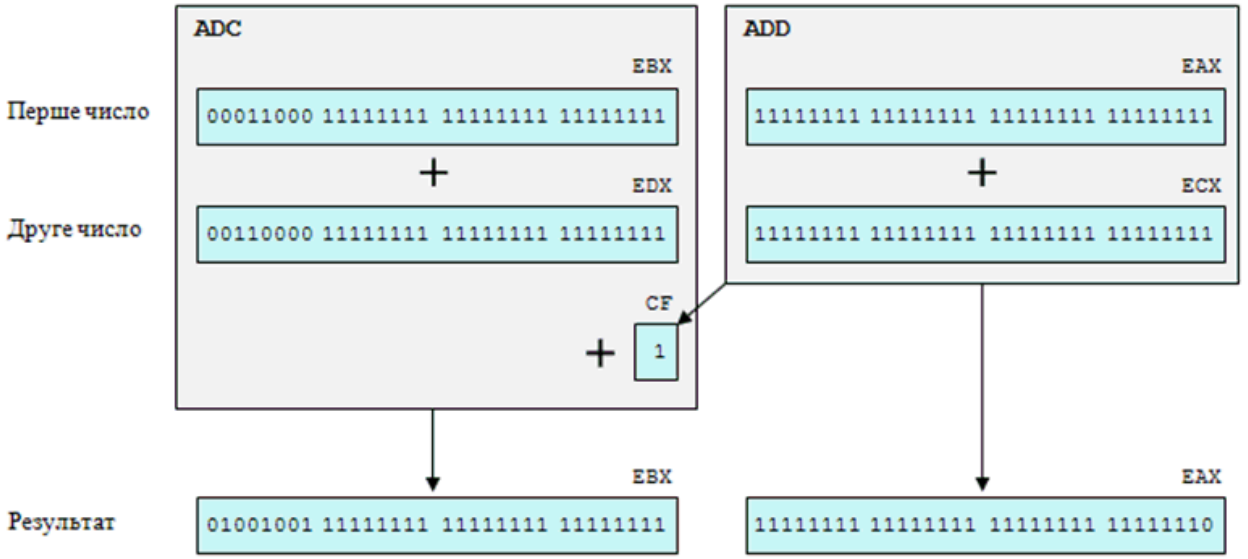
Це може бути використано для програмування обчислень операцій підвищеної розрядності. Надамо приклад обчислення суми 64-бітових цілих чисел у 32-бітовому процесорі. Нехай одне 64-бітове число записане у парі регістрів EBX:EAX, а інше 64-бітове число – у парі регістрів EDX:ECX.

```
;-одне 64-бітове число:  18ffffff ffffffff
  mov eax, 0ffffffffh           ;молодші 32 біти
  mov ebx,  08ffffffh           ;старші 32 біти

;-друге 64-бітове число: 30ffffff ffffffff
  mov ecx, 0ffffffffh           ;молодші 32 біти
  mov edx,  30ffffffh           ;старші 32 біти

;-додавання 64-бітових чисел
  add eax, ecx                  ;EAX = fffffffe, CF=1
  adc ebx, edx                  ;EBX = 49ffffff
```

Результат – у регістрах EBX:EAX. Проілюструємо цей приклад наступним чином



Див. також лаб 4

Команда SUB

SUB dest, src

Ця команда зменшує значення операнду **dest** на величину **src**. Наприклад:

```
mov eax, 12
sub eax, 7          ;EAX = EAX - 7. Результат 00000005h  (+5)
```

Ще приклад:

```
mov eax, 12
sub eax, -7          ;EAX = EAX - (-7). Результат 00000013h  (+19)
```

Для наступного прикладу результат повинен бути від'ємним:

```
mov eax, 7
sub eax, 12         ;EAX = EAX - 12. Результат FFFFFFFBh (-5)
```

Шістнадцятковий запис FFFFFFFB означає двійковий додатковий код числа (-5).

Як ви вважаєте, яким буде результат у наступному прикладі:

```
mov eax, 7         ;EAX = 00000007h
sub al, 12         ;Виконується AL - 12. EAX = ?
```

Якщо вважати результатом вміст регістру EAX, то це буде 000000FBh = 251 **(??)**
Якщо вважати, що байтова операція представляє результат у восьмибітовій розрядній сітці, то результатом є вміст регістру AL: значення FBh (у двійковому коді це 11111011) представляє 8-бітний двійковий додатковий код числа (-5). Крім того, ця команда SUB записує одиницю у біт CF регістру EFLAGS.

Команда SBB

Див. також лаб 4

SBB dest, src

Означає цілочисельне віднімання, враховуючи позичання (integer Subtraction with Borrow).
Ця команда від першого операнду (dest) віднімає два значення: src та біту CF регістру EFLAGS:

dest = dest - src - CF

Це використовується для програмування обчислень операцій підвищеної розрядності. Надамо приклад віднімання двох 64-бітових цілих чисел у 32-бітовому процесорі. Нехай одне 64-бітове число записане у парі регістрів EBX:EAX, а інше 64-бітове число – у парі регістрів EDX:ECX.

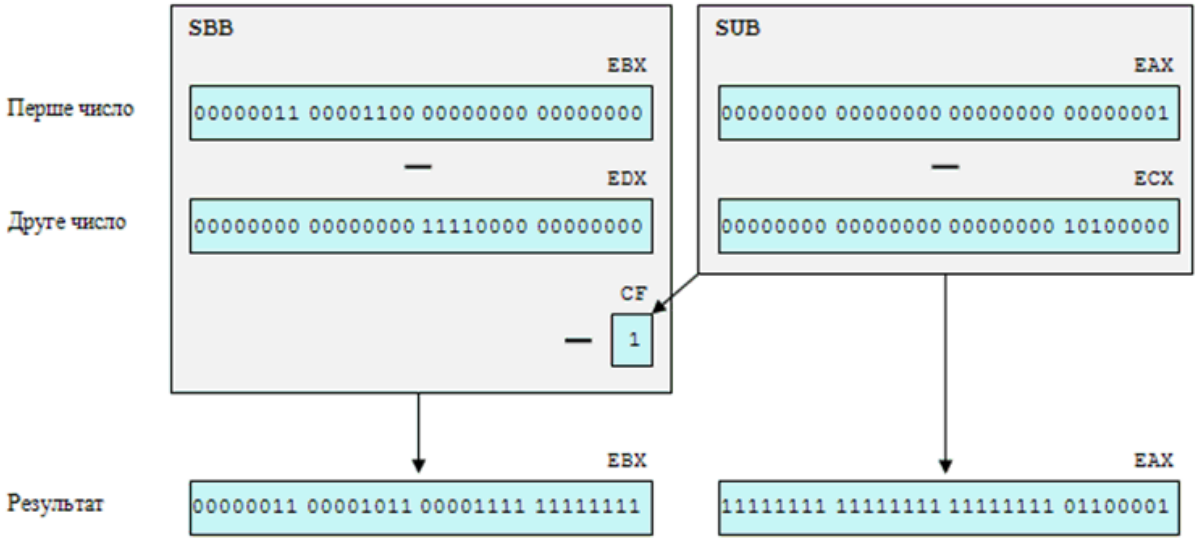
```
;-одне 64-бітове число:   030c0000 00000001
  mov eax, 00000001h
  mov ebx, 030c0000h

;-друге 64-бітове число:  0000f000 000000a0
  mov ecx, 000000a0h
  mov edx, 0000f000h

  sub eax, ecx  ;                                   00000000 00000000 00000000 00000001
                ;                                 - 00000000 00000000 00000000 10100000
                ;
                ;результат FFFFFF61                 11111111 11111111 11111111 01100001
                ;                             CF = 1

  sbb ebx, edx  ;  00000011 00001100 00000000 00000000
                ;- 00000000 00000000 11110000 00000000
                ;
                ;результат 030B0FFF
                ;  00000011 00001011 00001111 11111111
```

Результат – у регістрах EBX:EAX. Проілюструємо цей приклад наступним чином

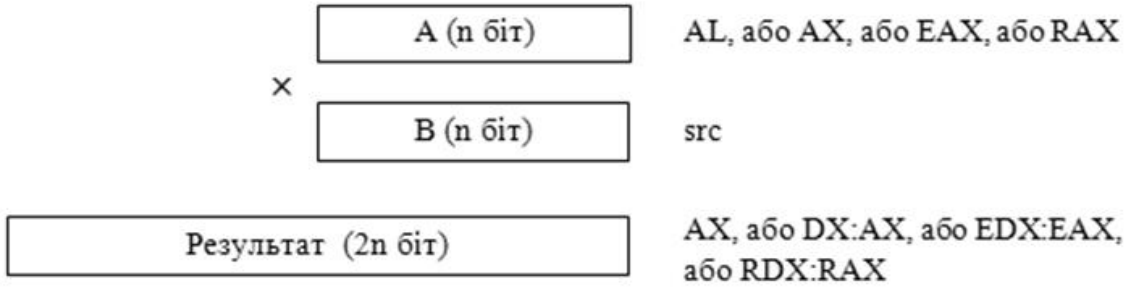


Команда MUL

Команда MUL означає множення цілих чисел без знаку.

MUL src

Ця команда виконує множення операнду src на значення у регістрі AL, або AX, або EAX, або RAX у залежності від розрядності операнду src. Результат записується відповідно у регістр AX, або регістри AX:DX, або у регістри EAX:EDX, або у регістри RAX:RDX.



Таким чином, якщо представити множення як A×B, то перед виконанням команди MUL програміст повинен забезпечити наявність множника A у регістрі A.

Операнд src може бути регістром, або посиланням на пам'ять. Безпосередньо числове значення (*immediate operand*) у якості операнду для команди MUL не допускається.

Можливі варіанти форматів інструкцій на основі команди MUL надані у таблиці.

Інструкція	Розрядність операнду	Що виконується	Розрядність результату
MUL r/m8	8	AX ← AL * r/m8	16
MUL r/m16	16	DX:AX ← AX * r/m16	32
MUL r/m32	32	EDX:EAX ← EAX * r/m32	64
MUL r/m64	64	RDX:RAX ← RAX * r/m64	128

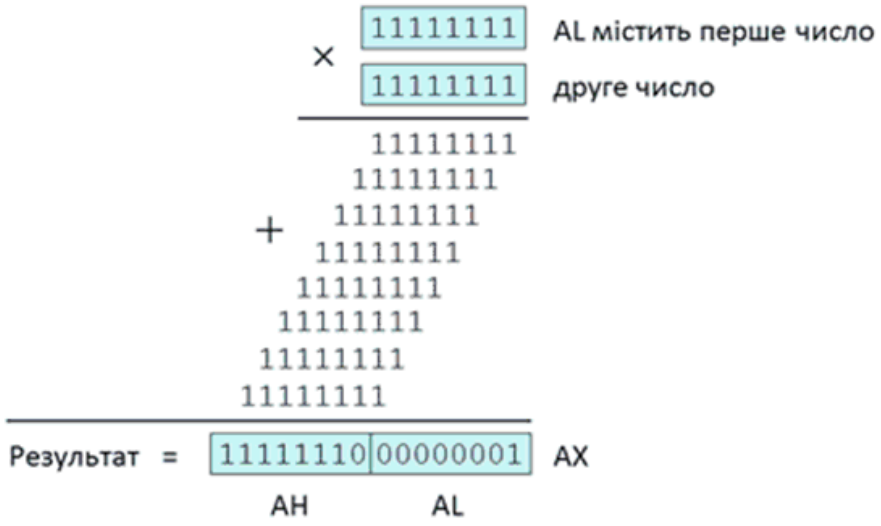
Приклади перемноження 8-бітових значень

```
.data
  bvalue db 255

.code
  mov eax, 0FFh         ;беззнакове 255
  mov ebx, 0FFh
  mul bl                ;AX = AL*BL. Результат FE01h = 65025 = 255*255

  mov eax, 0FFh
  mul bvalue            ;AX = AL*bvalue. Результат FE01h = 65025 = 255*255
```

При множенні двох 8-бітових чисел без знаку результат операції може бути у діапазоні від 0 = 0×0 до 65025 (FE01h) = 255×255. Тобто, для представлення результату потрібно 16 біт, які записуються у регістр AX.



Команда IMUL

Команда IMUL означає множення цілих чисел зі знаком. Ця команда має три форми, які відрізняються кількістю операндів:

```
IMUL src                ; один операнд
IMUL dest, src          ; два операнди
IMUL dest, src1, src2   ; три операнди
```

Одноопераднда форма IMUL по формату (проте не по результату) ідентична MUL.

Інструкція	Розрядність операнду	Що виконується	Розрядність результату
IMUL r/m8	8	AX ← AL * r/m8	16
IMUL r/m16	16	DX:AX ← AX * r/m16	32
IMUL r/m32	32	EDX:EAX ← EAX * r/m32	64
IMUL r/m64	64	RDX:RAX ← RAX * r/m64	128

Приклади одноопераднодного 8-бітового множення командою IMUL

```
mov eax, 0FFh           ;AL = -1 (8 бітове число зі знаком)
mov ebx, 0FFh           ;BL = -1
imul bl                 ;AX = AL*BL. Результат 0001h = 1 = (-1)*(-1)
```

```
mov al, 7Fh       ;AL = 127 (максимальне позитивне 8 бітове число зі знаком)
mov bl, 7Fh       ;BL = 127
imul bl           ;AX = AL*BL. Результат 3F01h = 16129 = 127*127
```

```
mov al, 7Fh       ;AL =  127 (максимальне позитивне 8 бітове число зі знаком)
mov bl, 80h       ;BL = -128 (мінімальне від'ємне 8 бітове число зі знаком)
imul bl           ;AX = AL*BL. Результат C080h = -16256 = 127*(-128)
```

Приклади одноопераднього 16-бітового множення

```
mov ax, 0FFFFh          ;AX = -1 (16 бітове число зі знаком)
mov bx, 0FFFFh          ;BX = -1
imul bx                 ;DX:AX = AX*BX. Результат 0000h:0001h = 1 = (-1)*(-1)
```

```
mov ax, 7FFFh    ;AX = 32767 (максимальне позитивне 16 бітове число зі знаком)
mov bx, 7FFFh    ;BX = 32767
imul bx          ;DX:AX = AX*BX. Результат 3FFFh:0001h = 1073676289 = 32767*32767
```

```
mov ax, 7FFFh     ;AX =  32767 (максимальне позитивне 16 бітове число зі знаком)
mov bx, 8000h     ;BX = -32768 (мінімальне від'ємне 16 бітове число зі знаком)
imul bx           ;DX:AX = AX*BX. Результат C000h:8000h = -1073709056 = 32767*(-32768)
```

Двохоперандна форма IMUL

IMUL dest, src

Перший операнд (операнд призначення dest) помножується на другий операнд (src).

Перший операнд може бути тільки регістром загального призначення, другий операнд може бути регістром загального призначення або посиланням на пам'ять.

Інструкція	Розрядність операндів і результату	Що виконується
IMUL r16, r/m16	16	WORD регістр ← WORD регістр * r/m16
IMUL r32, r/m32	32	DWORD регістр ← DWORD регістр * r/m32
IMUL r64, r/m64	64	QWORD регістр ← QWORD регістр * r/m64

На відміну від одноопернадної форми, тут немає вимоги обов'язкового використання регістрів EAX та EDX. Проілюструємо це наступним прикладом, у якому для операндів використовуються регістри EBX та ECX.

```
mov ebx, 10
mov ecx, 5
imul ebx, ecx       ;EBX = EBX*ECX. Результат 50.
```

Переповнення розрядної сітки при множенні

Спробуємо 32-бітове позитивне число 7FFFFFFF (2147483647) помножити на 8. Результатом повинно бути 2147483647*8 = 17179869176. Проте, такий результат не може вміститися у 32-бітовий регістр.

```
mov eax, 7FFFFFFFh
mov ebx, 8
                    ;до виконання команди IMUL регістр EFLAGS = 00000246
imul eax, ebx
                    ;результат EAX = FFFFFFF8 (помилка)
                    ;регістр EFLAGS = 00000A47 (записуються біти CF=1, OF=1)
```

У результаті регістр міститме EAX = FFFFFFF8 (якщо розглядати це як 32-бітовий додатковий код, то означатиме -8, що неправильно). При такому переповненні розрядної сітки у біти CF та OF регістру EFLAGS записуються 1.

У наступному прикладі 32-бітове позитивне число 0FFFFFFF (268 435 455) множимо на 8 і отримуємо правильний результат 7FFFFFF8 (2147483640) – переповнення немає.

```
mov eax, 0FFFFFFFh
mov ebx, 8
                   ;до виконання команди IMUL регістр EFLAGS = 00000246
imul eax, ebx
                   ;результат EAX = 7FFFFFF8 (правильно)
                   ;регістр EFLAGS = 00000246 (не змінилося жодного біта)
```

Про безпосередні числові значення для одного з операндів

У документації Intel для двохоперандного IMUL записано, що другий операнд (src) може також бути безпосереднім числом (*immediate value*), проте у таблиці кодів інструкцій серед двохоперандних IMUL для другого операнду *imm* не вказано. Спробуємо розібратися. Розглянемо приклад двохоперандного 32-бітового множення, у якому для команди IMUL запишемо другий операнд у вигляді безпосередньо числового значення.

```
  mov eax, 5
  imul eax, -200      ;у результаті EAX = -1000    (32-бітний код FFFFFC18)
```

Компілятор MASM Visual Studio C++ не видає помилку і програма коректно виконується. Проте, якщо при налагодженні програми звернути увагу на зміст вікна дизасемблера, то там бачимо, зокрема, наступне

```
  mov eax, 5
00401005  mov         eax,5
  imul eax, -200                              ;записано у вихідному тексті програми
0040100A  imul        eax,eax,0FFFFFF38h    ;те, що насправді виконується
```

Таким чином, інструкцію двохоперандної IMUL вихідного тексту компілятор сам перетворив у **трьохоперандну** форму.

Трьохоперандна форма IMUL

Інструкція	Розрядність результату	Що виконується
IMUL r16, r/m16, imm8	16	WORD регістр ← r/m16 * sign-extended imm8
IMUL r32, r/m32, imm8	32	DWORD регістр ← r/m32 * sign-extended imm8
IMUL r64, r/m64, imm8	64	QWORD регістр ← r/m64 * sign-extended imm8
IMUL r16, r/m16, imm16	16	WORD регістр ← r/m16 * imm16
IMUL r32, r/m32, imm32	32	DWORD регістр ← r/m32 * imm32
IMUL r64, r/m64, imm32	64	QWORD регістр ← r/m64 * imm32