НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

КОНСПЕКТ ЛЕКЦІЙ

“ПРОГРАМУВАННЯ, ч.3

“СИСТЕМНЕ ПРОГРАМУВАННЯ”

Лекція 1

Вступ

Раніше ми розглядали мови програмування високого рівня. Переваги цих мов - наближення до користувача та зручність програмування складних логічних задач.

Недоліки – порівняно невисока ефективність машинних програм. Всі можливості ПЕОМ повністю не використовуються, тому для програмування системних задач, управління роботою обладнання такі мови - недостатньо ефективні.

Тут перевагу мають машинно-орієнтовані мови, однією з яких є асемблер. Що таке асемблер?

Після трансляції програма з мови високого рівня перетворюється на послідовність машинних команд. Одна первинна команда реалізується декількома машинними. Всі імена користувача (змінні, мітки) замінюються адресами. Наприклад, двохадресна команда має вигляд:

```
256 4725 0648
```

Перші три цифри – це код команди, далі адреса першого операнду й адреса другого операнду.

У мові асемблер коди команд й адреса замінюються їх символічними іменами. Наприклад:

```
MOV X,Y
```

Це значно полегшує роботу програміста, не потрібно пам’ятати машинні коди команд, самому займатися розподілом пам’яті. Окрім цього, асемблер має засоби модульного програмування: макрокоманди та процедури.

Одна команда в асемблері транслюється в одну машинну. Саме цьому програми на асемблері мають значно більший (щодо тексту) обсяг, ніж на мовах високого рівня. Не плутайте з кінцевим розміром файлу. За допомогою асемблера є можливість використовувати усі можливості ПЕОМ і ефективність виконання таких програм буде неймовірно високою. Зрозуміло, що програмувати на асемблері складні логічні програми буде дуже трудомісткою, а тому недоцільною справою. Призначення асемблера для сучасних машин – системне програмування для узгодження окремих програмних систем, управління роботою обладнання, системою передачі даних та ін.

ПЕОМ	Розрядність	Макс. Пам’ять, Мб
XT 8088/86	16	1
AT 80286	16	16
A 80386 DX	32	1024
80386 SX	16	16
80386 SL	16	16

Не випадково, програмна оболонка Norton Commander складається з 20 тис. рядків мовою С і 10 тис. на асемблері.

Якщо універсальні мови програмування незначною мірою залежать від типу ПЕОМ, то асемблер повністю пов’язаний з конкретним типом . Необхідно знати особливості архітектури певного класу ПЕОМ. Потрібно не лише вміти писати програми на асемблері, а й хоча б трохи знати ПЕОМ, з якою працюєте.

З іншого боку вивчення асемблеру дозволяє ознайомитись з усіма можливостями ПЕОМ, а також зрозуміти як в дійсності реалізується програма на машинному рівні.

Розділ 1

Особливості архітектури ПЕОМ на базі мікропроцесорів 8088/86

1.1 Стуктура ПЕОМ

Конструктивно ПЕОМ складається з декількох функціональних приладів, які реалізують арифметичні і логічні операції, управління, запам'ятовування, занесення/одержання даних.

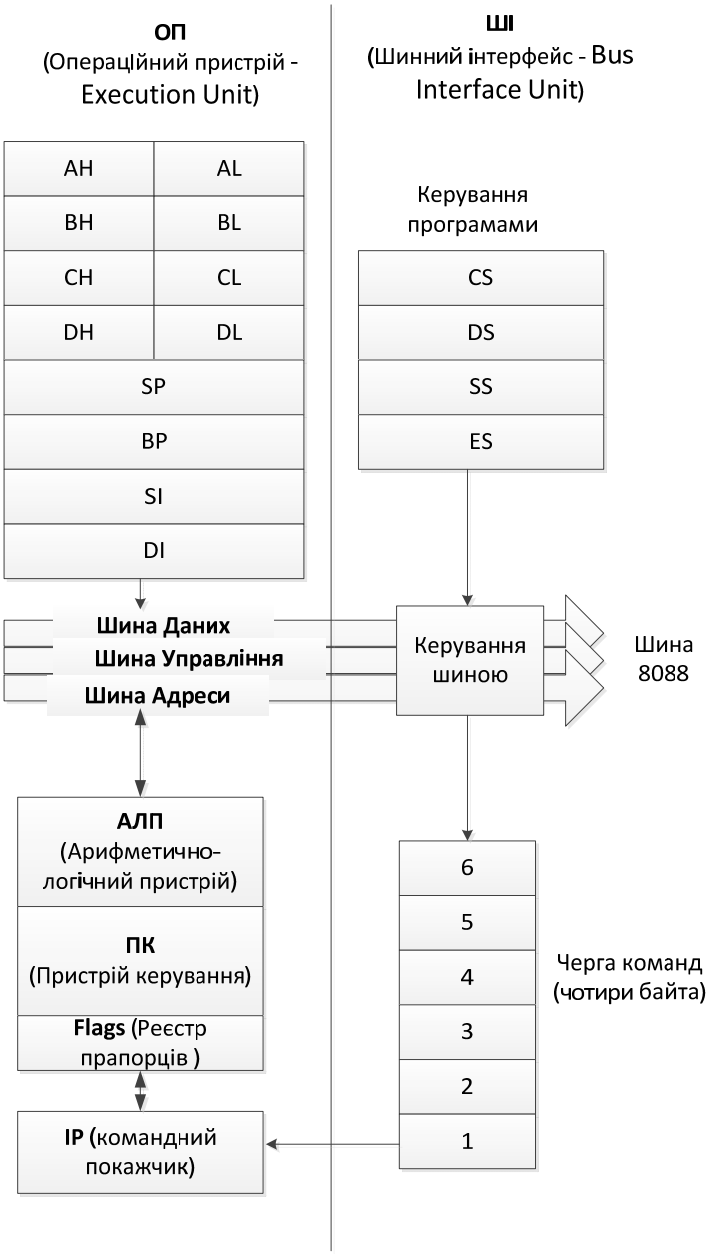


Рис. 1.1 Операційний пристрій і шинний інтерфейс

Як показано на рис.1.1, процесор розділено на дві частини: операційний пристрій (ОП) та шинний інтерфейс (ШІ). Призначення ОП – виконання команд, коли ШІ готує дані і команди для виконання.

ОП містить арифметично-логічний пристрій (АЛП), прилад управління (ПУ) і 14 регістрів. Ці пристрої забезпечують виконання команд, арифметичні розрахунки і логічні операції (порівняння на більше, менше, дорівнює). Програми операційної системи і програми, які виконуються, знаходяться в оперативній пам'яті (ОЗП) .

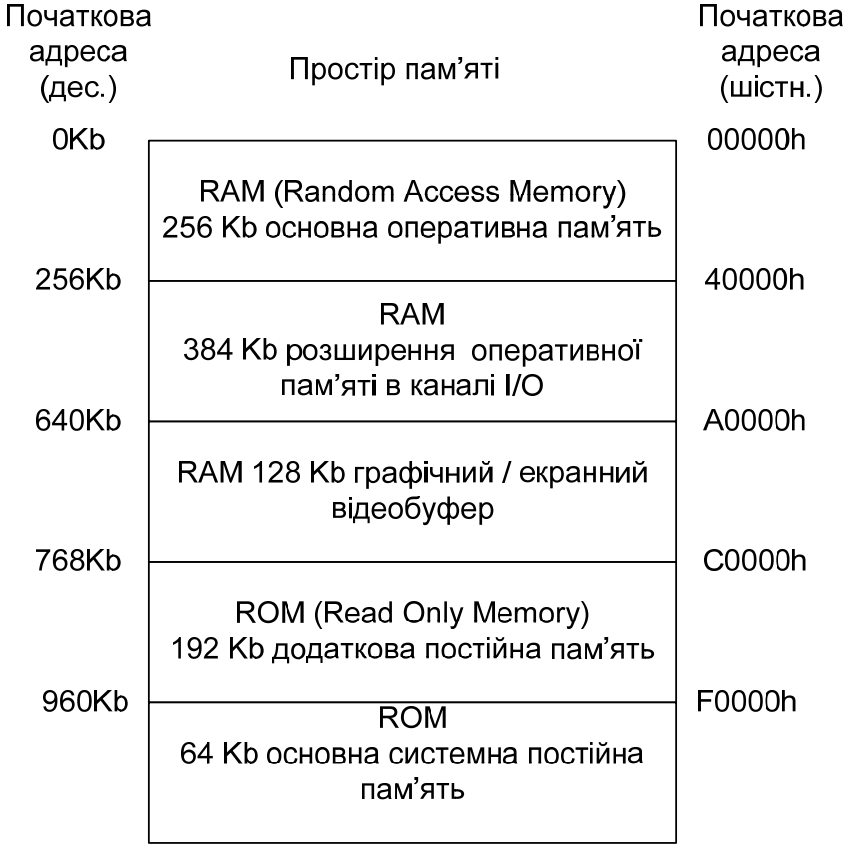


Рис. 1.2 Карта фізичної пам'яті

Пам'ять RAM містить перші *три чверті (3/4)* пам'яті, а ROM - останню чверть. Перші 256К пам'яті знаходяться на системній платі. Для програміста доступні перші 640К пам'яті.

1.1.1 Центральний процесор

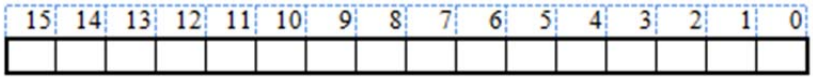
Як відомо, АЛП і ПУ об'єднуються в одному кремнієвому кристалі і утворюють центральний процесор (ЦП).

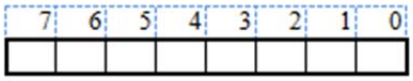
АЛП виконує основні арифметичні операції (додавання, віднімання) і інші логічні операції. Ці дії реалізуються спеціальними приладами, наприклад, суматорами. Якщо б між суматорами і ОЗП не було жодного додаткового пристрою, необхідно було б дуже багато пересилок. Для зменшення їх кількості в ПЕОМ застосовуються спеціальні проміжні ЗП – регістри.

1.1.2 Пам'ять і особливості її використання

Пам'ять складається з окремих комірок. Всередині ПЕОМ дані подаються в двійковій системі числення, тому комірка складається з ряду двійкових розрядів (бітів), в кожному з яких записано 0 або 1.

8 біт складають *байт*, який є найменшою адресованою одиницею даних. Комірка складається з одного *слова* або 2 байтів. Всі розряди слова або регістру нумеруються з права на ліво від 0 до 15. В байті розряди нумеруються від 0 до 7.





Операції можна виконувати як над словами так і над байтами. Кожна така комірка визначається місцем в пам’яті – адресою і даними, які містяться в ній, тобто змістом.

Так, як байт – найменша адресована одиниця даних, то для того, щоб дістатися до нього, потрібно мати його номер або адресу. Усі байти нумеруються від 0. У слові правий байт має меншу адресу, а лівий – більшу. Саме тому їх, відповідно, називають *молодшим і старшим*. Адреса слова співпадає з адресою його молодшого байта. З цього випливає, що байти спеціально нумеруються від 000, а слова тільки парною нумерацією. Тобто, слово не може мати непарної адреси.

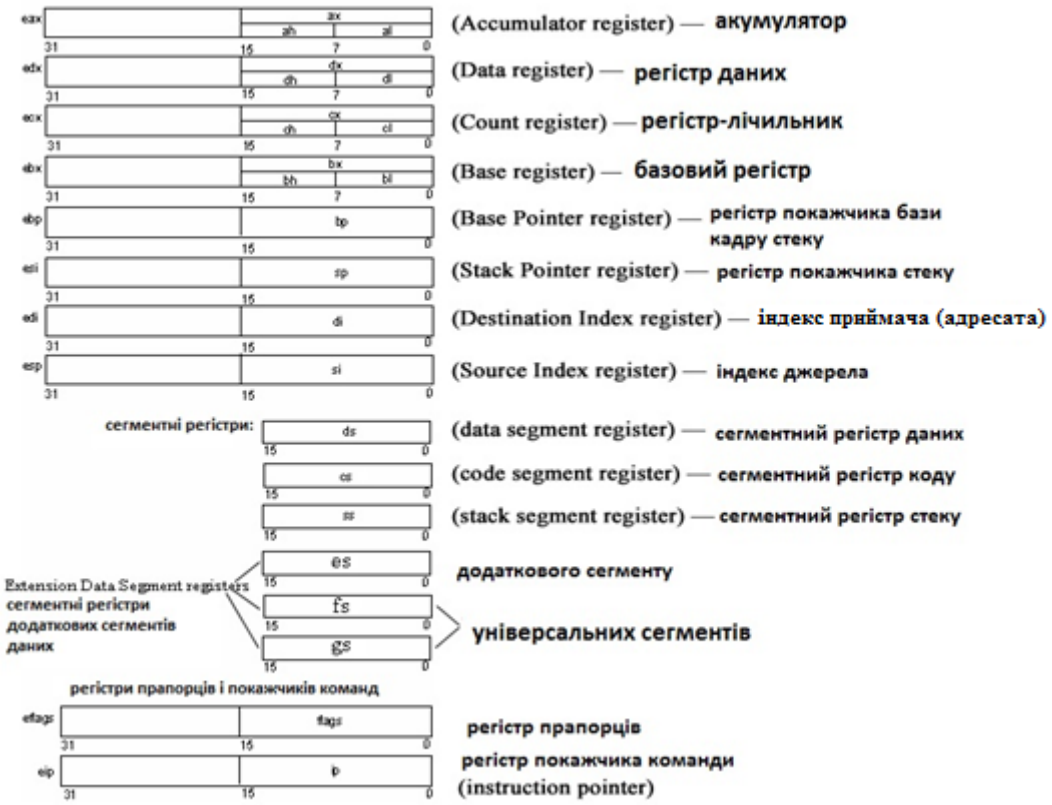
Найбільший номер, який можна записати до 16 біт - 2^{16}-1, а враховуючи нульовий номер - 2^{16} . З цього випливає, що з використанням 16-ти бітів можна пронумерувати $2^{16} = 2^6 * 2^{10}$ = 64 Кбайт. В 16й системі це буде найбільша адреса 0FFFFh.

Вся пам’ять розділяється на *сегменти* не більше, ніж 64 Кбайти. Одночасно можно використовувати не більш, ніж 4 сегменти. Кожна комірка буде визначатися своєю адресою відносно початку сегменту – зміщенням або *виконавчою адресою*. Адреса початку сегмента зберігається в *регістрах сегментів*. Адреса слова в пам’яті буде рівна сумі виконавчої адреси і адреси сегменту (IP + CS) і буде називатись *абсолютною адресою*. Адреса сегмента і виконавча адреса займають 16 біт, а абсолютна адреса – 20 біт. До цього можна прийти спеціальними перетвореннями.

1.1.3 Регістри ОВМ

Розглянемо особливості використання окремих регістрів.

Регістр - це проміжна пам'ять, місткістю одне машинне слово (16 біт). В ЦП існують багато регістрів, більшість з яких недосяжні для програміста .



За регістрами закріплені назви і імена, через які до них можна звертатися:

1. 4 регістри загального призначення - **AX, BX, CX, DX**.
2. 4 регістри покажчики - **SP, BP, SI, DI**.
3. 4 сегментних регістри - **CS, DS, SS, ES**.
4. 2 керуючих регістри - покажчика команд **IP** і регістру прапорців **F**.

Призначення окремих регістрів і особливості використання розглянемо далі.

Якщо програмі недостатньо одного сегмента даних, то вона має можливість використовувати ще три додаткових сегмента даних, але на відміну від основного сегмента даних, адреса якого міститься в сегментному регістрі **DS**, при використанні додаткових сегментів даних їх адреси потрібно вказувати явно, за допомогою спеціальних префіксів перевизначення сегментів в команді.

Адреси додаткових сегментів даних повинні міститися в регістрах **ES, GS, FS** (extension data segment registers).

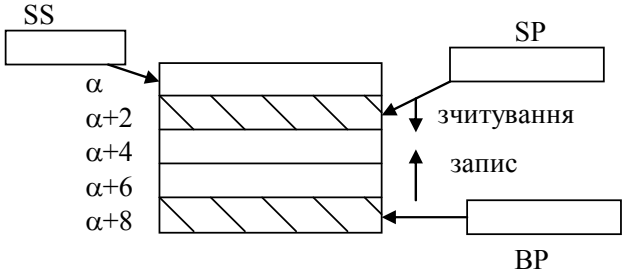


Рис. 1.3. Модель роботи стеку

В сучасних ПЕОМ широко використовуються *стеки*, наприклад, при роботі з підпрограмами, при обробці переривань, а також для запам'ятовування проміжних результатів.

Стек - це ділянка пам'яті, яка заповнюється в бік менших адрес, а звільняється в бік збільшення адрес. Тут реалізується метод "останній прийшов - першим обслуговується", тобто правило LIFO (Last Input First Output). Приклад стека - пачка аркушів паперу на столі. Стек характеризується своєю базовою адресою, яка записується в базовий регістр **SS**, і адресою вершини, яка записується в регістр покажчика стеку **SP** (stack pointer). В стек записуються слова (2 байти).+

Лекція 2

1.1.4. Регістри: стан і управління

Мікропроцесор складається з кількох регістрів (див. рис. 1.1.4), які постійно зберігають інформацію як про стан самого мікропроцесора, так і про програми, команди якої в даний момент завантажені на конвеєр. До цих регістрів відносяться:

1. Регістр прапорців eflags/flags;
2. Регістр покажчика команди eip/ip.

Використовуючи ці регістри можна отримати інформацію про результат виконання команди і впливати на стан самого мікропроцесора. Розглянемо детальніше призначення і вміст цих регістрів:

eflags/flags (flag register) — регістр прапорців. Розрядність eflags/flags — 32/16 біт. Окремі біти цього регістру мають певне функціональне призначення та називаються прапорцями. Молодша частина цього регістру повністю аналогічна регістру flags для i8086.

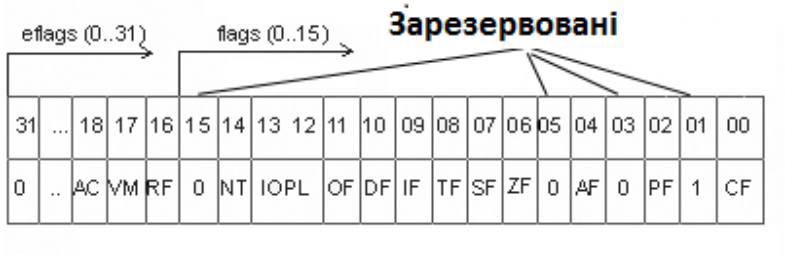


Рис. 1.1.4. Регістр прапорців

Таблиця

Призначення бітів регістру прапорців

Мнемоніка прапорця	Прапорець	№ біта в eflags	Вміст і призначення
CF	Прапорець переносу (Carry Flag)	0	1 — вказує на переповнення старшого біту при арифметичних командах. Старшим є 7, 15, 31-й біт в залежності від розмірності операнду
PF	Прапорець парності (Parity Flag)	2	1 — 8 молодших розрядів (тільки для 8 молодших розрядів) результату містять парну кількість одиничних бітів
AF	Прапорець корекції (Adjust Flag)	4	1- якщо арифметична операція виконує перенесення або займа в/із 3-й біт результату, інакше - скидається. Цей прапорець використовується в двійково-кодованій десятковій арифметиці(BCD - Binary-Coded Decimal).
ZF	Прапорець нуля (Zero Flag)	6	1 — результат операції нульовий; 0 — результат операції не нульовий
SF	Прапорець знаку (Sign Flag)	7	Відображає стан старшого біту результату (біти 7, 15, 31 для 8, 16, 32-розрядних операндів відповідно): 1 — старший біт результату дорівнює 1; 0 — старший біт результату дорівнює 0.
TF	Прапорець пастки (Trap Flag)	8	1 – процес використовує по-командне налагодження програми; 0 – програма виконується звичним чином.

IF	Прапорець дозволу переривань (Interrupt enable Flag)	9	1 – у відповідь на IRQ процесор генерується переривання; 0 – процесор не відповідає на них (але не ігнорує).
DF	Прапорець напрямку (Direction Flag)	10	0 – рядкові команди опрацьовують рядки даних, переходячи від менших адрес до більших (CLD); 1 – у зворотному напрямку (STD).
OF	Прапорець переповнення (Overflow Flag)	11	1 — в результаті операції виконується перенесення із старшого знакового біту результату (біти 7, 15 чи 31 для 8, 16 чи 32-розрядних операнд відповідно); Прапорці стану використовуються для цілочисельної арифметики трьох типів. При переповненні індикатором являється: 1. для знакової арифметики - прапорець **OF**, 2. для беззнакової арифметики - прапорець **CF**, 3. для BCD-арифметики - прапорець **AF**
IOPL	Рівень привілеїв введення/ виведення (Input/Output Privilege Level)	12, 13	Використовується в захищеному режимі роботи мікропроцесора для контролю доступу до команд введення/виведення в залежності від привілейованості задач.
NT	Прапорець вкладеної задачі (Nested Task flag)	14	1 – поточна задача викликана з попередньої; 0 – поточна задача ***НЕ*** являється викликаною з попередньої.
RF	Прапорець відновлення (Resume Flag)	16	Керує відповіддю процесора на виняток налагодження.
VM	Прапорець віртуального режиму 8086 (Virtual-8086 Mode flag)	17	1 – процесор переходить в віртуальний режим 8086; 0 – повертається в захищений режим.
AC	Прапорець перевірки вирівнювання (Alignment Check flag)	18	1- змушує процесор перевіряти вирівнювання під час доступу до пам’яті і в випадку невирівняного доступу генерувати виключення.
VIF	Прапорець віртуальних переривань (Virtual Interrupt)	19	Це віртуальний образ прапорця IF,використовується разом з прапорцем VIP при ввімкненому розширенні режиму віртуального 8086.
VIP	Прапорець очікування віртуального переривання (Virtual Interrupt Pending flag)	20	Встановлюється, коли виникає переривання. Процесором зчитується і використовується тільки разом з прапорцем VIF, змінюється тільки програмно.
ID	Прапорець	21	Якщо програма змогла встановити і скинути цей

	ідентифікації (IDentification flag)		прапорець, то це значить, що процесор може виконати команду CPUID.

1.2 Особливості виконання команд

Шина МП 8086 складається із трьох шин: інформаційної (16 бит), адресної (20 бит) та управляючої, якою передаються сигнали управління. Виконання програми складається з 5-ти етапів. Ці етапи виконуються послідовно.

1. Вибір службової машинної команди з пам’яті;
2. Розшифрування команди;
3. Зчитування операндів з пам’яті (якщо необхідно);
4. Виконання команди;
5. Запис операндів в пам’ять (якщо необхідно).

Для прискорення виконання наборів команд, вони виконуються двома пристроями: шинним інтерфейсом (Bus Interface Unit) та операційним пристроєм (Execution Unit) (рис. 1.2.1.). Перший пристрій зчитує команду і здійснює передачу даних. Другий – лише виконує команди. Шинний інтерфейс може вибирати наступну команду в той час, як операційний пристрій виконує вибрану раніше команду.

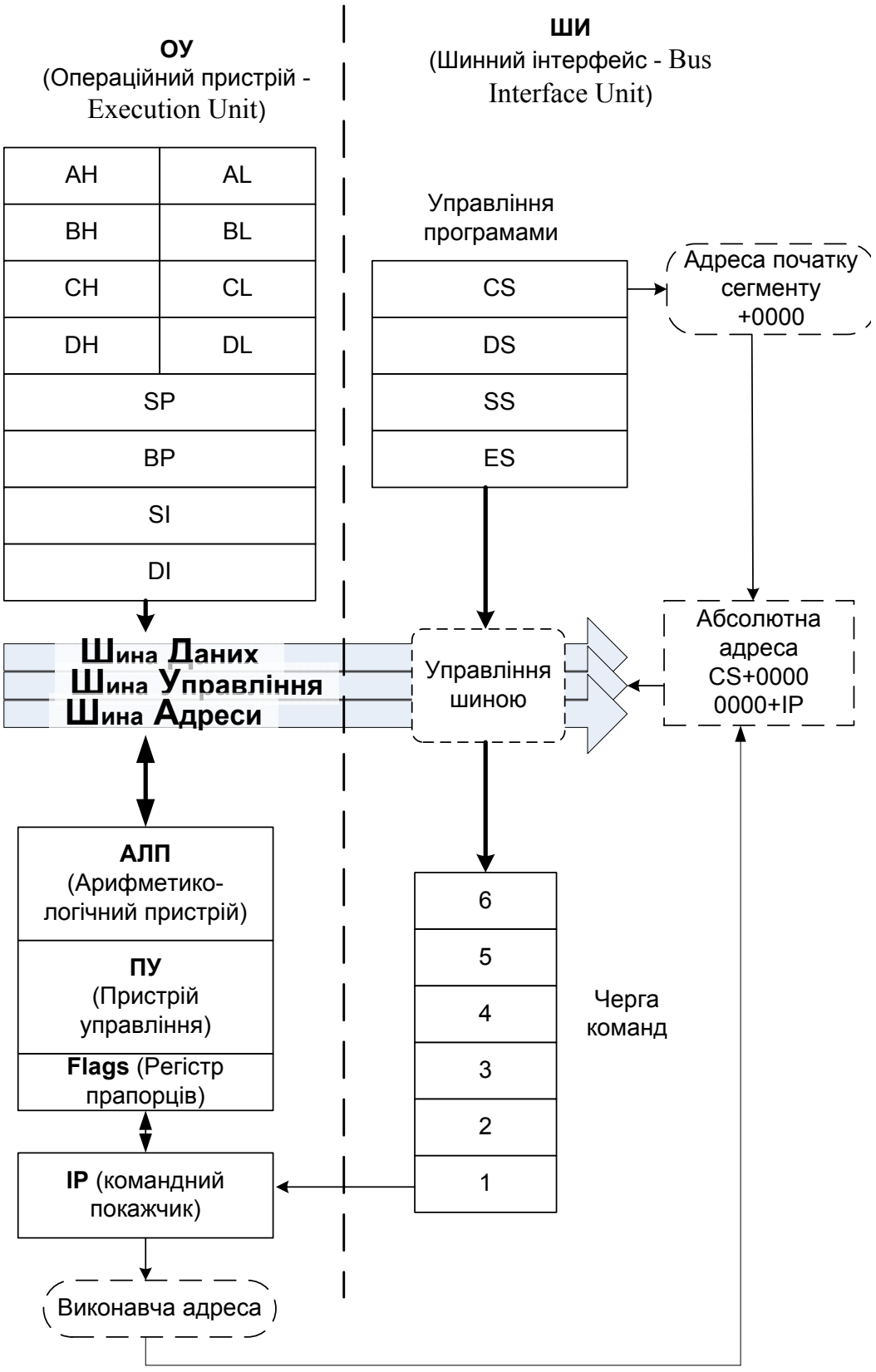


Рис. 1.2.1. Операційний пристрій і шинний інтерфейс.

Процесор має внутрішню пам’ять, яка називається *чергою команд*. Тут зберігається до 4 (в 8086 - до 6) попередньо вибраних із потоку команд-байт. Тобто, реалізується своєрідний конвеєр команд. Як же формується абсолютна адреса команди?

В регістрі покажчика команд IP записана виконавча адреса, а в регістрі сегменту коду CS - адреса початку сегменту. Абсолютна адреса дорівнює сумі CS+IP та записується в 20 біт, в той час як і CS і IP мають по 16 біт.

Прийнято, що сегмент повинен починатися не з будь-якої адреси, а з кратньої 16 біт. Область пам’яті 16 біт називається *параграфом*. Інакше кажучи, сегмент вирівнюється по межах параграфу. Отже, сегмент може починатися за адрес 16, 32, 48 . В двійковій системі це буде:

```
0000000
1000000
1100000
```

Як бачимо, при цьому останні 4 біти будуть нульовими. Для зберігання вони зайві і їх відкидають. Тобто, в 16-ти розрядних регістрах сегменту фактично зберігається 20-ти розрядна адреса, але без 4-ьох нулів праворуч.

При обчисленні абсолютної адреси до вмісту регістру CS дописують 4 нулі справа праворуч, а до вмісту IP - 4 нулі ліворуч і отримані коди додаються.

1.2.1. Реалізація переривань

Зчитування символу та запису до пам’яті триває кілька мікросекунд. Якщо ЦП буде лише приймати ці символи, то більшість часу він буде простоювати. Тому, закінчивши опрацювання символу, ЦП переходить до виконання іншої програми. Кожен раз, коли натискається клавіша, пристрій відправляє запит на переривання. ЦП перериває виконання програми і переходить до виконання процедури опрацювання переривання для клавіатури. Кожна така процедура являється визначеною програмою, яка записана в пам’яті.

Для того, щоб до неї перейти, треба знати її початкову адресу. Ця початкова адреса і записана в так званому *векторі переривань*. Кожне із можливих 256 переривань під своїм номером має вектор переривань в пам’яті. Вектор переривань складається з двох слів: **CS:IP**. Записані в початкових адресах від 0 до 03FFH (1024 байти).

Особливості 32-розрядних процесорів

В процесорі i80286 адресна шина складалася із 24 біт, що дає змогу адресувати до 16Mb пам'яті, але це можливо зробити в так званому *захищеному режимі*.

В процесорі i486 32-розрядне слово, яке дає можливість адресувати $2^{32} = 4*^{230} = 4$ Гбайт. При сегментній організації пам’яті, розмір сегменту і буде таким. Крім того, є ще організація пам’яті сторінками. Розмір сторінки - 4 Кб. Такий спосіб дозволяє використовувати віртуальну пам’ять, об’єм якої більший за фізичну, близько 4 Тбайт.

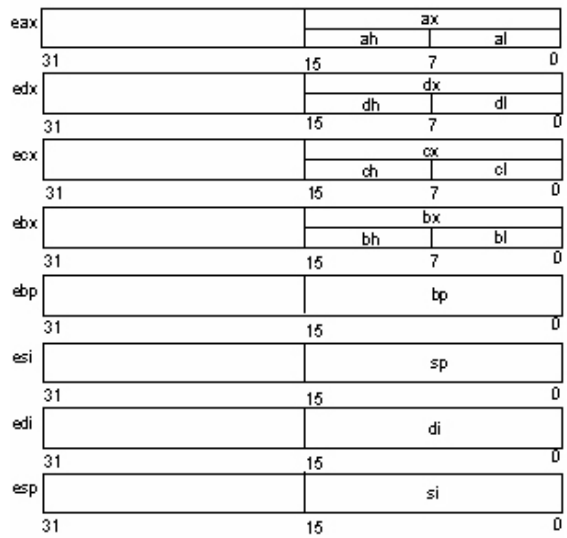
Ці процесори повинні мати змогу виконувати паралельні обчислення, тому їх структура складніша.

Для програміста процесор складається із 32-ох регістрів, 16 із яких являються системними, а інші - користувача.

Розглянемо особливості регістрів користувача.

Регістри загального призначення: 32-розрядні. Ім’я 32-розрядного починається літерою E (extended): **EAX, EBX, ECX, EDX**. Молодша половина просто **AX** . Молодша половина доступна і може ділитися навпіл. Старша половина окремо недоступна.

Регістри-покажчики : **ESP, EBP, ESI, EDI**.
Сегментні регістри : **CS, SS, DS, ES** і ще два додаткові -- **GS, FS**. Тобто, 6 регістрів.
Регістри управління: 32-розрядні -- **EFLAGS** и **EIP**, а молодша частина **- FLAGS** и **IP**.



(Accumulator register) — **акумулятор**

(Data register) — **регістр даних**

(Count register) — **регістр-лічильник**

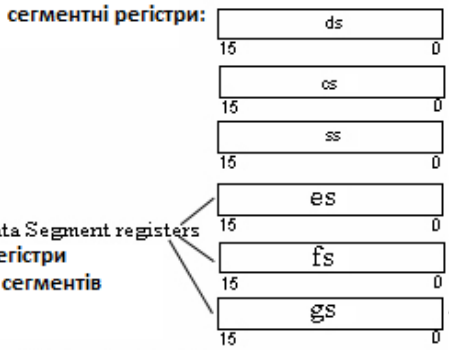
(Base register) — **базовий регістр**

(Base Pointer register) — **регістр покажчика бази кадру стеку**

(Stack Pointer register) — **регістр покажчика стеку**

(Destination Index register) — **індекс приймача(адресата)**

(Source Index register) — **індекс джерела**



(data segment register) — **сегментний регістр даних**

(code segment register) — **сегментний регістр коду**

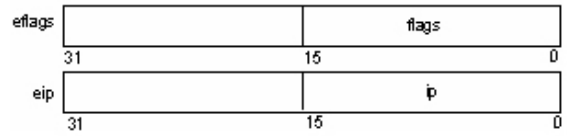
(stack segment register) — **сегментний регістр стеку**

додаткового сегменту

Extension Data Segment registers
сегментні регістри додаткових сегментів даних

універсальних сегментів

регістри прапорців і покажчиків команд



регістр прапорців

регістр покажчика команди (instruction pointer)

Лекція 3

1.2.2. Контролер переривань

1.2.2.1. Механізм переривань

Для обробки подій, що відбуваються асинхронно по відношенню до виконання програми, краще всього підходить механізм переривань. Переривання можна розглядати як деяку особливу подію в системі, що вимагає моментальної реакції. Наприклад, добре спроектовані системи підвищеної надійності використовують переривання по аварії в електромережі для виконання процедур запису вмісту регістрів і оперативної пам'яті на магнітний носій для того, щоб після відновлення живлення можна було продовжити з того ж місця.

Здається очевидним, що можливі найрізноманітніші переривання з самих різних причин. Тому переривання розглядається не просто як таке, з ним пов'язують число, а саме номертипу переривання або просто номер переривання. З кожним номером переривання зв'язується та чи інше подія. Система вміє розпізнавати, яке переривання, з яким номером сталося і запускає процедуру з відповідним номером.

Програми можуть самі викликати переривання з заданим номером. Для цього вони використовують команду INT. Це так звані програмні переривання. Програмні переривання не є асинхронними, так як викликаються з програми. Програмні переривання зручно використовувати для організації доступу до окремих, загальним для всіх програм модулів. Наприклад, програмні модулі операційної системи доступні прикладним програмам саме через переривання, і немає необхідності при виклику цих модулів знати їх поточну адресу в пам'яті. Прикладні програми можуть самі встановлювати свої обробники переривань для їх подальшого використання іншими програмами. Для цього вбудовані обробники переривань повинні бути резидентними в пам'яті. Апаратні переривання викликаються фізичними пристроями і приходять асинхронно. Ці переривання інформують систему про події, пов'язані з роботою пристроїв, наприклад про те, що нарешті завершилася друк символу на принтері і непогано було б видати наступний символ, або про те, що потрібний сектор диска вже прочитаний і його вміст доступно програмі.

Використання переривань при роботі з повільними зовнішніми пристроями дозволяють поєднати введення/виведення з обробкою даних в центральному процесорі і в результаті це підвищує загальну продуктивність системи.

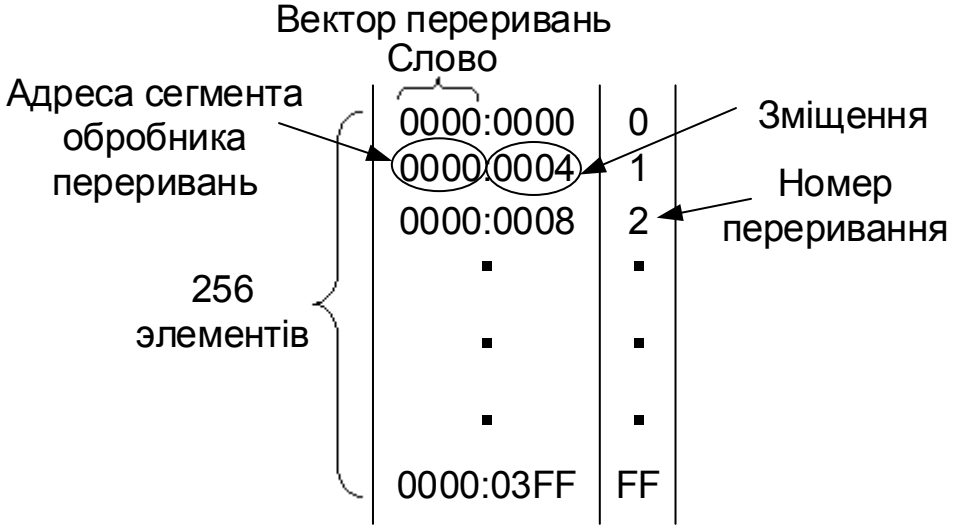
Деякі переривання (перші п'ять у порядку номерів) зарезервовані для використання самим центральним процесором на випадок якихось особливих подій на кшталт спроби ділення на нуль, переповнення і т.п. Іноді бажано зробити систему нечутливою до всіх або окремих перериваннь. Для цього використовують так зване маскування переривань, про яке ми ще будемо докладно говорити. Але деякі переривання замаскувати не можна, це немасковані переривання.

Зауважимо ще, що обробники переривань можуть самі викликати програмні переривання, наприклад, для отримання доступу до сервісу BIOS або DOS (сервіс BIOS також доступний через механізм програмних переривань).

1.2.2.2. Таблиця векторів переривань

Для того, щоб зв'язати адресу обробника переривання з номером переривання, використовується таблиця векторів переривань, що займає перший кілобайт оперативної пам'яті - адреси від 0000:0000 0000 до:03FF. Таблиця складається з 256 елементів - FAR-адреси обробників переривань. Ці елементи називаються векторами переривань. У першому слові

елемента таблиці записано зміщення, а в другому – адреса сегмента обробника переривання. Переривання з номером 0 відповідає адресу 0000:0000, переривання з номером 1 - 0000:0004 і т.д.



Ініціалізація таблиці відбувається частково BIOS після тестування апаратури і перед початком завантаження операційної системою, частково при завантаженні DOS. DOS може переключити на себе деякі переривання BIOS. Приведемо призначення деяких найбільш важливих векторів:

Таблиця

Призначення деяких найбільш важливих векторів:

Номер	Опис
0	**Помилка ділення.** Викликається автоматично після виконання команд DIV або IDIV, якщо в результаті ділення відбувається переповнення (наприклад, при діленні на 0). DOS зазвичай при обробці цього переривання виводить повідомлення про помилку і зупиняє виконання програми. Для процесора 8086 при цьому адреса повернення вказує на наступну після ділення команду, а в процесорі 80286 - на перший байт команди, що викликала переривання.
1	**Переривання покрокового режиму.** Виробляється після виконання кожної машинної команди, якщо в слові прапорців встановлений біт покрокового трасування TF. Використовується для налагодження програм. Це переривання не виробляється після виконання команди MOV в сегментні регістри або після завантаження сегментних регістрів командою POP, PUSH.
2	**Апаратне немасковане переривання.** Це переривання може використовуватися по-різному в різних машинах. Зазвичай виробляється при помилці парності в оперативній пам'яті і при запиті переривання від співпроцесора.
3	**Переривання для трасування.** Це переривання генерується при виконанні однобайтової машинної команди з кодом CCh і зазвичай використовується отладчиками для установки точки переривання.

4	**Переповнення.** Генерується машинною командою INTO, якщо встановлений прапор OF. Якщо прапорець не встановлений, то команда INTO виконується як NOP. Це переривання використовується для обробки помилок при виконанні арифметичних операцій.
5	**Друк копії екрану**. Генерується при натисканні на клавіатурі клавіші PrtScr. Зазвичай використовується для друку образу екрана. Для процесора 80286 генерується при виконанні машинної команди BOUND, якщо значення, що перевіряється, вийшло за межі заданого діапазону.
6	Невизначений код операції або довжина команди більше 10 байт (для процесора 80286).
7	Особливий випадок відсутності математичного співпроцесора (процесор 80286).
8	***IRQ0*** - перериванняінтервального **таймеру**, виникає 18,2 рази за секунду.
9	***IRQ1*** - переривання від**клавіатури**. Генерується при натисканні і при відпусканні клавіші. Використовується для читання даних з клавіатури.
A	***IRQ2***- використовується для каскадування апаратних переривань в машинах класу AT.
B	***IRQ3*** - переривання асинхронного порту **COM2.**
C	***IRQ4*** - переривання асинхронного порту **COM1.**
D	***IRQ5*** - переривання від контролера жорсткого диску для XT.
E	***IRQ6*** - переривання генерується контролером флоппі-диска після завершення операції.
F	***IRQ7*** - переривання принтера. Генерується принтером, коли він готовий до виконання чергової операції. Багато адаптерів принтера не використовують це переривання.
10	Обслуговування **видеоадаптера.**
11	**Визначення конфігурації** пристроїв в системі.
12	**Визначення разміру оперативної пам'яті** в системі.
13	Обслуговування **дискової** системи.
14	Послідовне **введення/виведення.**
15	Розширений сервіс для AT-компьютерів.
16	Обслуговування клавіатури.
17	Обслуговування принтера.
18	Запуск BASIC в ПЗУ, якщо він є.
19	Завантаження операційної системи.

1A	Обслуговування годинника.
1B	Обробник переривань Ctrl-Break.
1C	Переривання виникає 18.2 рази в секунду, викликається програмно обробником переривання таймера.
1D	**Адреса відео таблиці** для контролера відеоадаптера 6845.
1E	Покажчик на таблицю параметрів дискети.
1F	Покажчик на графічну таблицю для символів с кодами ASCII 128-255.
20-5F	Використовується DOS чи зарезервовано для DOS.
60-67	Переривання, зарезервовані для користувача.
68-6F	Не використовуються.
70	***IRQ8***–переривання від годинника реального часу.
71	***IRQ9*** - переривання від контролера EGA.
72	***IRQ10***– зарезервовано.
73	***IRQ11***–зарезервовано.
74	***IRQ12***–зарезервовано.
75	***IRQ13*** - переривання від математичного співпроцесора.
76	***IRQ14*** - переривання від контролера жорсткого диску.
77	***IRQ15*** – зарезервовано.
78 - 7F	Не використовуються.
80-85	Зарезервовані для BASIC.
86-F0	Використовуються інтерпретатором BASIC.
F1-FF	Не використовуються.

IRQ0-IRQ15 - це апаратні переривання. Про них буде сказано пізніше.

1.2.2.3. Маскування переривань

Часто при виконанні критичних ділянок програм для того, щоб гарантувати виконання певної послідовності команд, доводиться забороняти переривання. Це можна зробити командою CLI. Її потрібно помістити в початок критичної послідовності команд, а в кінці розташувати команду STI, що дозволяє процесору сприймати переривання. Команда CLI забороняє тільки масковані переривання, немасковані завжди обробляються процесором.

1.2.2.4. Зміна таблиці векторів переривань

Вашій програмі може знадобитися організування обробки деяких переривань. Для цього програма повинна змінити вектор на свій обробник. Це можна зробити, змінивши вміст відповідного елемента таблиці векторів переривань. Дуже важливо не забути перед завершенням роботи відновити вміст змінених в таблиці векторів переривань, тому що після завершення роботи програми пам'ять, яка була їй виділена, вважається вільною і може бути використана для завантаження іншої програми. Якщо ви забули відновити вектор і відбулося переривання, то система може зруйнуватися - вектор тепер вказує на область, яка може містити що завгодно. Тому послідовність дій для нерезидентних програм, які бажають обробляти переривання, повинна бути такою:

1. Прочитати вміст елемента таблиці векторів переривань для вектора з потрібним вам номером;

2. Запам'ятати цей вміст (адреса старого обробника переривання) в області даних програми;

3. Встановити нову адресу в таблиці векторів переривань так, щоб воно відповідало початку Вашої програми обробки переривання;

4. Перед завершенням роботи програми прочитати з області даних адресу старого обробника переривання і записати його в таблицю векторів переривань.
Для полегшення роботи по заміні переривання DOS надає у ваше розпорядження спеціальні функції для читання елемента таблиці векторів переривання і для запису в неї нової адреси. Якщо ви будете використовувати ці функції, DOS гарантує, що операція по заміні вектора буде виконана правильно. Вам не треба турбуватися про безперервність процесу заміни вектора переривання. Для читання вектора використовуйте функцію 35h переривання 21h. Перед її викликом регістр AL повинен містити номер вектора в таблиці. Після виконання функції в регістрах ES:BX буде шукана адреса обробника переривання. Функція 25h переривання 21h встановлює для вектора з номером, який знаходиться в AL, обробник переривання DS:DX. Зрозуміло, ви можете безпосередньо звертатися до таблиці векторів переривань, але тоді при записі необхідно замаскувати переривання командою CLI, не забувши потім дозволити їх обробку командою STI.

1.2.2.4. Зміна таблиці векторів переривань

Апаратні переривання виробляються пристроями комп'ютера, коли виникає необхідність їх обслуговування. Наприклад, при перериванні таймера відповідний обробник переривання збільшує вміст комірок пам'яті, використовуваних для зберігання часу. На відміну від програмних переривань, що викликаються заплановано самою прикладною програмою, апаратні переривання завжди відбуваються асинхронно по відношенню до тих, що виконуються програмами. Крім того, може виникнути одночасно відразу декілька переривань! Для того, щоб система не «розгубилася», вирішуючи яке переривання обслуговувати в першу чергу, існує спеціальна схема пріоритетів. Кожному переривання призначається свій унікальний пріоритет. Якщо відбувається одночасно декілька переривань, то система віддає перевагу самому высокоприоритетному, відкладаючи на певний час опрацювання інших переривань.

Таблиця

Апаратні переривання, розташовані у порядку зменшення пріоритету

Номер	Опис
8	IRQ0 переривання интервального таймеру, виникає 18,2 рази за секунду.
9	IRQ1 переривання від клавіатури. Генерується при натисканні та при відпусканні клавіші. Використовується для зчитування даних з клавіатури.
A	IRQ2 використовується для каскадування апаратних переривань у машинах класу AT.
70	IRQ8 переривання від годинника реального часу.
71	IRQ9 переривання від контролера EGA.
72	IRQ10 зарезервовано.
73	IRQ11 зарезервовано.
74	IRQ12 зарезервовано.
75	IRQ13 переривання від математичного співпроцесора.
76	IRQ14 переривання від контролера жорсткого диску.
77	IRQ15 зарезервовано.
B	IRQ3 перериванняасинхронного порта COM2.
C	IRQ4 перериванняасинхронного порта COM1.
D	IRQ5 перериваннявід контролера жорсткого дискудля XT.
E	IRQ6 п переривання генерується контролером флоппі диска після завершення операції.
F	IRQ7 перериванняпринтера. Генерується принтером, коли він готовий до виконаннячергової операції. Багато адаптерів принтера не використовують це переривання.

З таблиці видно, що найвищий пріоритет у переривань від інтервального таймера, потім йде переривання від клавіатури. Для управління схемами пріоритетів необхідно знати внутрішню структуру контролера переривань. Переривання, які виникають, запам'ятовуються в регістрі запиту на переривання IRR. Кожних з восьми біт в цьому регістрі відповідає якомусь перериванню. Після перевірки на обробку зараз іншого переривання запитується інформація з реєстру обслуговування ISR. Перед видачею запиту на переривання в процесор перевіряється вміст восьмибітового регістру маски переривань IMR. Якщо переривання даного рівня не замасковане, то видається запит на переривання.

1.3 Представлення даних в ЕОМ

Дані, що обробляються МП 8086, можна розділити на три види:
1. фізичні і логічні коди;

2. числа з фіксованою крапкою;

3. числа з плаваючою точкою.

З точки зору розмірності (фізична інтерпретація) мікропроцесор апаратно підтримує наступні основні типи даних.

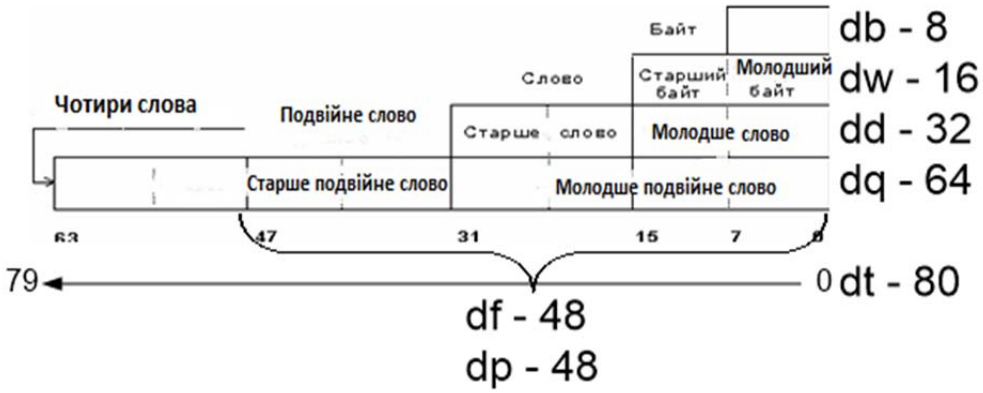
1.3.1 Типи даних

байт - вісім послідовно розташованих бітів, пронумерованих від 0 до 7, при цьому біт 0 є самим молодшим значущим бітом;

слово - послідовність з двох байт, що мають послідовні адреси. Розмір слова 16 біт; біти в слові нумеруються від 0 до 15.

подвійне слово - послідовність з чотирьох байт (32 біта), розташованих за послідовним адресами. Нумерація цих бітів проводиться від 0 до 31. Адресою подвійного слова вважається адреса його молодшого слова. Адреса старшого слова може бути використана для доступу до старшої половини подвійного слова.

чотири слова - послідовність з восьми байт (64 біта), розташованих за послідовними адресами. Адресою чотирьох слів слова вважається адреса його молодшого подвійного слова. Адреса старшого подвійного слова може бути використана для доступу до старшої половини чотирьох слів.



З точки зору їх розрядності, мікропроцесор на рівні команд підтримує логічну інтерпретацію:

1.3.2. Логічні коди

Логічними кодами подаються: символи, числа без знака, бітові величини. Усі пристрої введення/виведення обмінюються даними з ЕОМ символами. Будь-який текст або число при цьому розглядаються як послідовність символів. Кожен символ кодується кодом ASCII (КОІ -- 7) та в пам'яті ЕОМ займає один байт. Основна частина символів (латинські літери, цифри, розділові знаки) - 128 символів кодується 7 бітами, 8-й завжди дорівнює нулю. Інші символи (кириличні букви і ін.) кодуються розширеною частиною коду. (8-ма бітами). Наприклад, велика латинська буква А має код -65_{10}, літера З - 67_{10}. Тому АС буде мати вигляд:

128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	1
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
C								A							

Цифра 0 має код 48_{10} = 30h:

7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	0

Адреса комірки пам'яті або байта є числом без знака, яке подається так само, як і число з фіксованою крапкою, тільки без знака. При роботі з зовнішніми пристроями, при перетворенні даних доведеться мати справу з окремими бітами слова. В цьому випадку вміст комірки розглядається як визначений код або бітова величина.

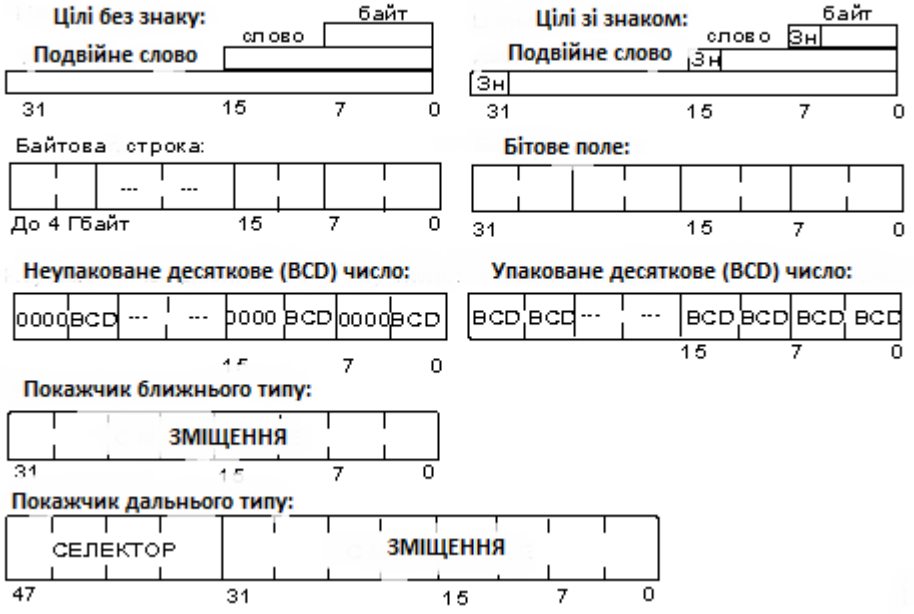
1.3.3. Числа з фіксованою крапкою

Цілий тип зі знаком - двійкове значення зі знаком, розміром 8, 16 або 32 біта. Знак у цьому двійковому числі міститься в 7, 15 або 31-му біті відповідно. Нуль в цих бітах в операндах відповідає додатньому числу, а одиниця - від'ємного. Від'ємні числа подаються в додатковому коді. Числові діапазони для цього типу даних наступні:
8-розрядне ціле - від -128 до +127;
16-розрядне ціле - від -32 768 до +32 767;
32-розрядне ціле - від -231 до +231-1.

Додатковий код від'ємного числа формується таким чином:
1. представити у відповідній системі абсолютну величину числа N;
2. для кожного розряду взяти його доповнення (якщо в двійковій системі, то просто замінити 1 на 0, і навпаки;);
3. додати одиницю, нехтуючи при цьому можливим переносом зі старшого розряду. Наприклад, число - 1607_{10} матиме такий вигляд: - 1607_{10}
1. |N| = 0000011001000111;
2. = 1111100110111000;
3. = 1111100110111001.

Цілий тип без знаку - двійкове значення без знаку, розміром 8, 16 або 32 біта. Числовий діапазон для цього типу наступний: байт - від 0 до 255; слово - від 0 до 65 535; подвійне слово - від 0 до 2^{32}–1. Ланцюжок (байтовий рядок)- представляє собою певний безперервний набір байтів, слів і подвійних слів максимальної довжини до 4 Гбайт. Бітове поле являє собою безперервну послідовність біт, в якій кожен біт є незалежним і може розглядатися як окрема змінна. Бітове поле може починатися з будь-якого біта будь-якого байта і містити до 32 біт. Неупакований двійково-десятковий тип BCD- байтове подання десяткової цифри від 0 до 9. Неупаковані десяткові числа зберігаються як байтові значення без знака по одній цифрі в кожному байті. Значення цифри визначається молодшим півбайтом. Упакований двійково-десятковий BCD тип являє собою упаковане подання двох десяткових цифр від 0 до 9 в одному байті. Кожна цифра зберігається у своєму півбайті. Цифра в старшому

полубайті (біти 4-7) є старшою. Покажчик на пам'ять двох типів: близького типу - 32-розрядна логічна адреса, що представляє собою відносне зміщення в байтах від початку сегмента. Ці покажчики можуть також використовуватися в суцільний (плоскої) моделі пам'яті, де сегментні складові однакові; далекого типу - 48-розрядна логічна адреса, що складається з двох частин: 16-розрядної сегментної частини - селектора, і 32-розрядного зміщення.



У пам'яті ПЕОМ ціле число представляється в двійковій системі числення. Для зручності реалізації арифметичних операцій цілі числа всередині ПЕОМ представляються в додатковому двійковому коді. Додатні числа мають звичайне двійкове подання і можуть займати одне слово або байт. Тому максимальним звичайним числом буде $215\text{-}1=32767_{10}$. В байт можна записати не більше, ніж $27 - 1 = 127_{10}$.

Ясно, що в старшому розряді від'ємного числа завжди буде 1, а додатнього - 0. Тому цей розряд називається знаковим.У додатковому двійковому коді +0 -0 представляються однаково.

Звичайні цілі числа в пам'яті ПЕОМ можуть змінюватися в діапазоні

$$-32768_{10} \le N \le 32767_{10}$$

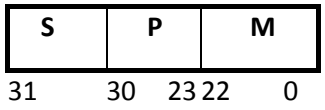
Використання додаткового двійкового коду дозволяє замінити операцію віднімання операцією додавання. Крім зазначеного подання цілих чисел використовується спеціальне представлення десяткових чисел для двійково-десяткової арифметики. При цьому одна десяткова цифра записується в 8 (НЕупакована форма) або 4 біта (упакована форма) в двійковій формі.

1.3.4. Числа з плаваючою крапкою

У МП 8086 безпосередньо не передбачається дій з плаваючою крапкою. Звичайні десяткові числа можна записувати як число з порядком, наприклад: $1234.5 = 123.45*10 = 12.345*10^2 = 1.2345*10^3$

Така форма виходить неоднозначною. Однак, якщо число зображувати так, щоб воно було менше одиниці і перша цифра за десятковою точкою була значущою (тобто не нуль), то таке подання буде однозначне і буде називатися нормалізованим. Нормалізоване число в пам'яті подається мантисою і порядком.
Для МП 8086 нормалізованим в двійковій системі вважається число, ціла частина мантиси якого дорівнює одиниці. Така форма зручна для компактного представлення дуже малих або дуже великих чисел. Якщо ж число має багато розрядів, то зберегти їх усі не вийде. Тому числа з плаваючою крапкою представляються в ПЕОМ наближено, а з фіксованою - точно.

В ПЕОМ звичайне дійсне число представляються у формі з плаваючою крапкою і займає два слова. Старший біт першого слова є знаковим (S). Порядок (P) займає байт (розряди 30 - 23). Мантиса (M) розташована в 23 бітах.



Відзначимо особливості подання порядку і мантиси числа. Щоб не відводити один розряд під знак порядку, в ці розряди записується не справжній порядок, а зі зміщенням $+127_{10}$.

Оскільки мантиса числа записується у нормалізованому вигляді, то у двійковому представленні ціла її частина буде завжди дорівнювати одиниці. Цю одиницю в пам'яті НЕ записують (зміщують мантиссу на один розряд вліво).

Наприклад:

Число $15.375_{10} = 1111.011_2$.

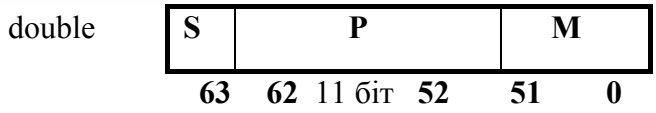
В нормалізованій формівоно буде мати вигляд:

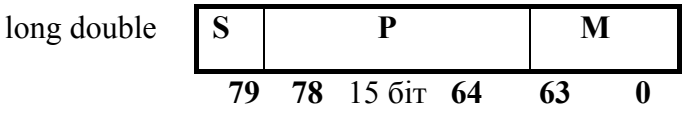
$1.111011*2^{11}{}_2$

Внутрішнє представлення числа:

S = 0; P = 3 + 127 = $130_{10} = 10000010_2$; M = 1110110…0.

На відміну від чисел з фіксованою крапкою від'ємні дійсні числа відрізняються від додатніх лише знаком. Порядок числа визначає його діапазон, а мантиса - точність (кількість значущих розрядів). Крім звичайних чисел можна використовувати подвійні (double) і довгі подвійні (long double), які мають структуру:





У довгих подвійних числах мантиса записується повністю, а в просто подвійних перша одиниця мантиси зберігається неявно. Властивості чисел з плаваючою крапкою:

Тип	Розмір (біт)	Діапазон		Точність десяткових розрядів
		Мінімальний	Максимальний	
Float	32	$3.4*10^{-38}$	$3.4*10^{38}$	7
Double	64	$1.7*10^{-308}$	$1.7*10^{308}$	15
Long double	80	$3.4*10^{-4932}$	$3.4*10^{4932}$	19

У самому МП 8086 дій над числами з плаваючою крапкою не передбачено. Вони виконуються в співроцесорі 8087, 8187, 8287, або моделюються програмно (емулюються).

Лекція 4

1.3.5. Десяткові числа

Десяткові числа - спеціальний вид представлення числової інформації, в основу якого покладено принцип кодування кожної десяткової цифри числа групою з чотирьох біт. При цьому кожен байт числа містить одну або дві десяткові цифри в, так званому, двійково-десятковому коді (BCD - Binary-Coded Decimal).

Мікропроцесор зберігає BCD-числа у двох форматах (рис. 1.3.1):

упакованому форматі - в цьому форматі кожен байт містить дві десяткові цифри. Десяткова цифра являє собою двійкове значення в діапазоні від 0 до 9 розміром 4 біта. При цьому код старшої цифри числа займає старші 4 біта. З цього випливає,що діапазон представлення десяткового упакованого числа в одному байті становить від 00 до 99;

не упакованому форматі - в цьому форматі кожен байт містить одну десяткову цифру в чотирьох молодших бітах. Старші чотири біти мають нульове значення. Це ,так звана, "зона". Отже, діапазон представлення десяткового не упакованого числа в одному байті складає від 0 до 9.

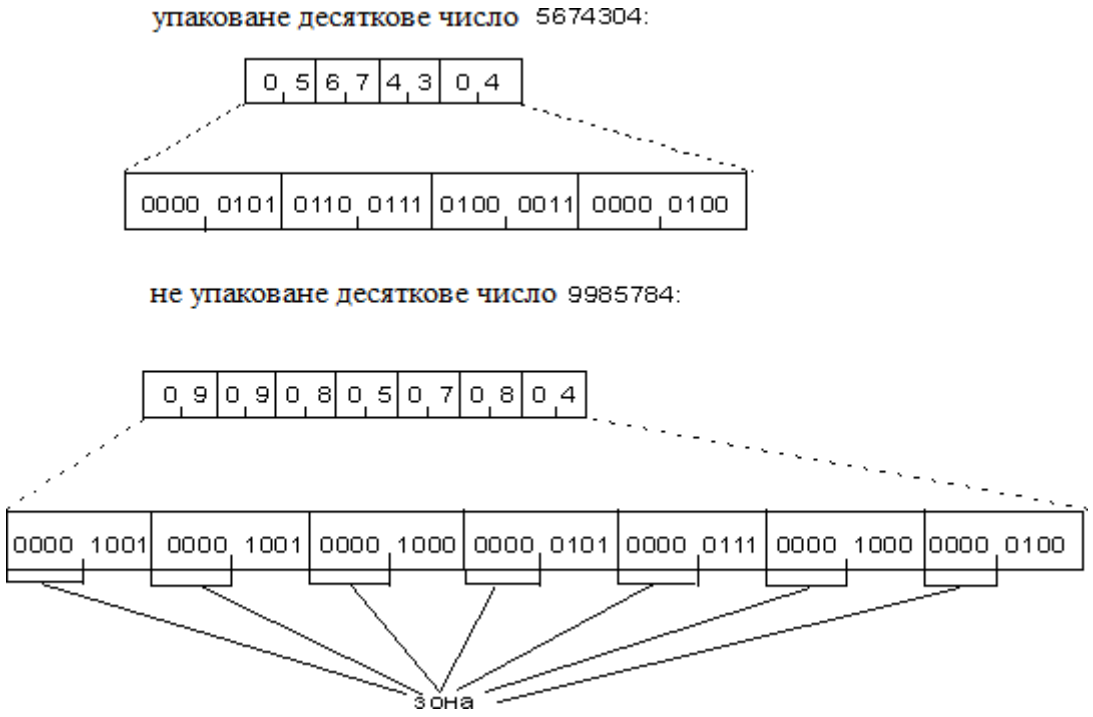


Рис. 1.3.1. Представлення числової інформації у двійково-десятковому коді (BCD - Binary-Coded Decimal)

Для цього можна використовувати лише дві директиви опису та ініціалізації даних - **db** і **dt**. Можливість застосування лише цих директив для опису BCD-чисел обумовлена тим, що до таких чисел також можна застосувати принцип "молодший байт за молодшою адресою", що, як ми побачимо далі, дуже зручно для їх обробки. І взагалі, при використанні такого типу даних як BCD-числа, порядок опису цих чисел в програмі і алгоритм їх обробки - це справа смаку і особистих пристрастей програміста. Це стане ясно після того, як ми нижче розглянемо основи роботи з BCD-числами. Наприклад, у сегменті

даних лістингу, наведена послідовність описів BCD-чисел буде мати вигляд у пам'яті так, як показано на рис. 1.3.2.

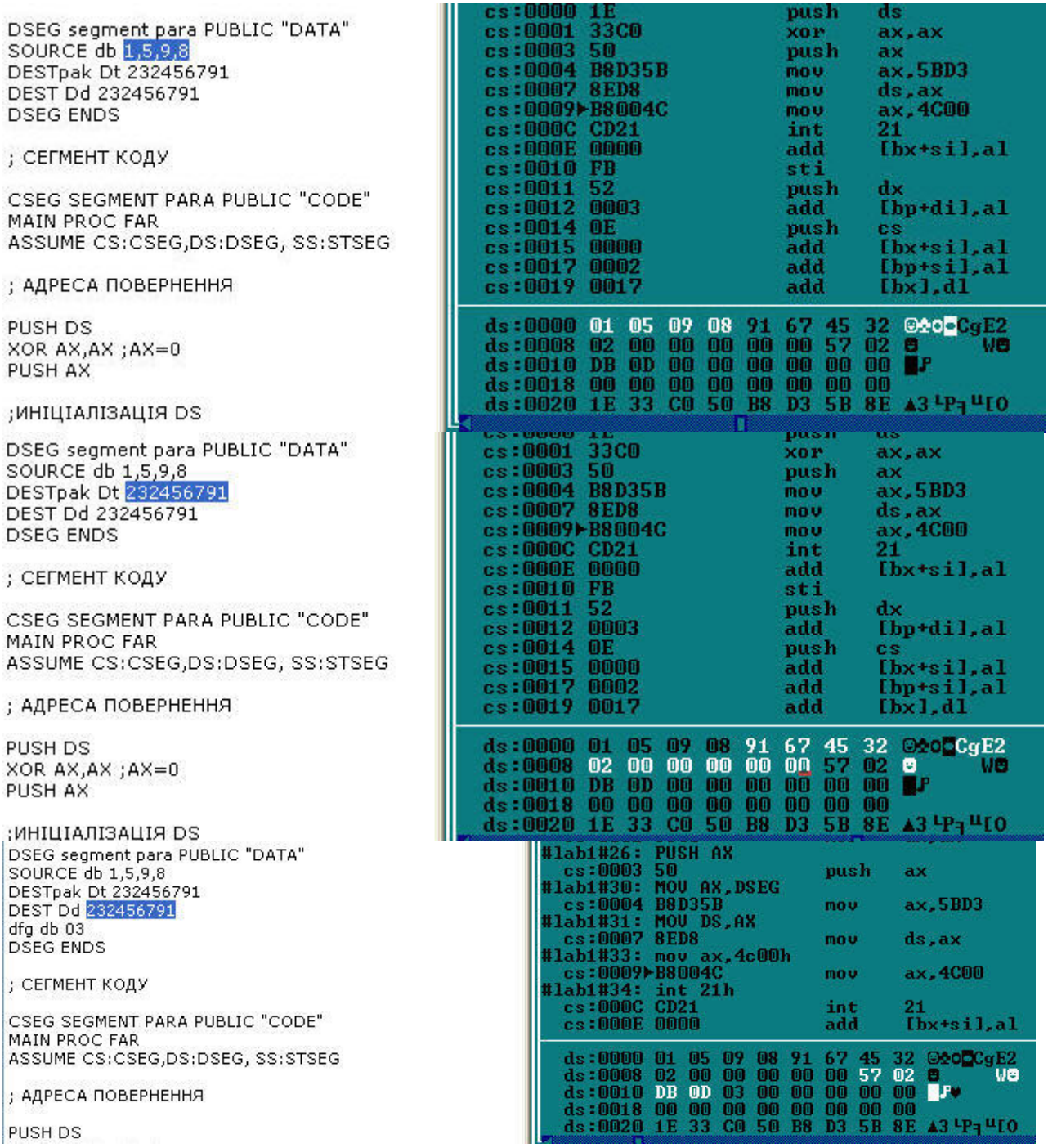


Рис. 1.3.2. Лістинги описів BCD-чисел

1.4. Режим адресації

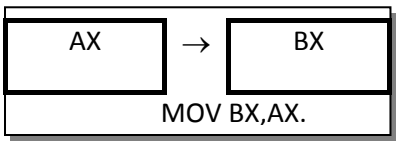
При виконанні програми процесор звертається до пам'яті, де зберігаються команди і дані. В командах перетворення даних визначаються адреси, де зберігається відповідна інформація, а у командах передачі управління визначається адреса команди, на яку необхідно перейти, тобто, адреси переходів. Спосіб або метод визначення у команді адреси операнда або адреси переходу, називається режимом адресації, чи просто адресацією. Можна у командах відзначати адреси операндів, тобто, використовувати пряму адресацію. Але операнди зберігаються не тільки у комірках пам'яті. Вони можуть знаходитися у регістрах загального призначення, сегментних регістрах. Крім того, операндами можуть бути константи, або операнди можуть перебувати у портах вводу/ виводу. В таких умовах використання тільки прямої адресації призведе до неефективних програм. Саме тому, у сучасних ЕОМ використовують багато інших режимів адресації, що дозволяє отримувати високоефективні програми для різних додатків.

Режими адресації МП 8086 можна розділити на 7 груп:

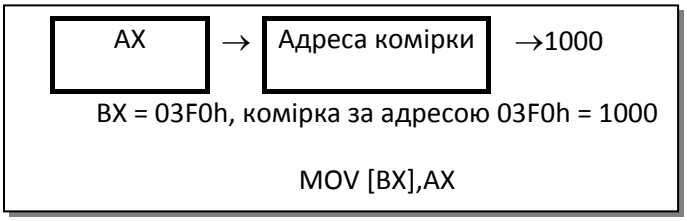
1. регістрова адресація;
2. безпосередня адресація;
3. пряма;
4. непряма регістрова;
5. адресація за базою;
6. пряма адресація з індексуванням;
7. адресація за базою з індексуванням.

У подальшому будемо вважати, що коли у команді відзначено ім'я регістра, то операндом буде його зміст.

Наприклад:



Якщо ж ім'я регістру укладено у прямокутні дужки, то значить, що операндом є вміст комірки, адреса якого зберігається у регістрі.



У першому випадку - пряма адресація, а в другому - непряма.

1.4.1. Регістрова адресація

Операнд (байт або слово) знаходиться у регістрі. Цей спосіб адресації можна застосувати до всіх програмно-адресованих регістрів процесора.

```
MOV AX,CX
```

В цьому випадку операндом є вміст певного регістру.

16-бітове слово, яке зберігається у лічильнику CX, переписується у акумулятор AX. CX залишиться незмінним, а AX зміниться.

Аналогічно для байтів MOV AL, BH.

Тобто, сам операнд визначає свою довжину - слово або байт. Цей метод не потребує звернення до пам'яті, сама команда займає мало місця. Тому виконується дуже швидко і є дуже ефективною.

```
inc CH ;Плюс 1 до того, що міститься CH

push DS ;DS зберігається  в стеку

xchg BX,BP ;BX и BP обмінюються вмістом

mov ES, AX ;Вміст AX пересилається в ES
```

1.4.2. Безпосередня адресація

В цьому випадку замість операнда джерела використовується безпосередньо константа:

```
MOV AX, 60
```

В акумулятор заноситься число 60. Ця константа розміщується не у пам'яті, а у самій машинній команді, тобто, у черзі команд. Тому буде виконуватися досить швидко. Можна:

```
MOV CL, -50
```

Константа може представлятися літералом, а може бути і іменованою. Ім'я константі присвоюється спеціальним оператором: EQU:

```
L EQU 256
……
MOV CX,L
```

1.4.3. Пряма адресація

Як зазначалося, у покажчику команд IP зберігається відносна адреса команди у сегменті, тобто, кількість байт щодо його початку, або виконавча адреса. Для прямої адресації виконавча адреса відзначається безпосередньо у команді. Якщо це адреса даних, то МП додає її до змісту значення регістру даних DS, який зсувається на 4 біта і отримує 20-бітну абсолютну адресу. Використовувати конкретні числові значення адреси незручно. Тому адреса частіше задається міткою, щоб розмістити якесь число за цією адресою, використовуються оператори DB, DW або DD (Define Byte, Define Word, Define Double Word).

```
TABLE DW 1560;          в TABLE (2 байти)  записано 1560
INDEX DB -126;          в байт INDEX записано -126
```

Тоді можна записати:

```
MOV AX,TABLE;       переслати зміст TABLE в аккумулятор
```

Відзначимо особливість такої пересилки. Коли у пам'яті було записано:

X	TABLE
Y	TABLE+1

то молодший байт X буде пересланий у молодший байт регістру, а старший - у старший. І отримаємо:

Y	X
AH	AL

Тобто, байти ніби помінялися місцями (зміщення записується до команди).

1.4.4. Непряма регістрова адресація

Цей спосіб адресації використовує базовий регістр **BX**, покажчик **BP** і індексні регістри **SI**, **BI**, де записана адреса операнда:

```
MOV AX,[BX]
```

Для вибору операнда-джерела відбувається звернення до регістру **BX**, де зберігається його адреса. Якщо там записано 2000, то зміст комірки 2000 пересилається у акумулятор. А як у регістр **BX** занести адресу комірки, наприклад, TABLE? Це можна зробити за допомогою операції **OFFSET** (зміщення).

Наприклад:

```
MOV BX, OFFSET TABLE
Порівняйте: MOV BX, TABLE
```

1.4.5. Адресація за базою

Якщо необхідно отримати доступ до однієї комірки, то їй потрібно надати ім'я і використовувати пряму адресацію:

```
MOV AX,TABLE
```

При роботі з масивом даних позначати своїм ім'ям кожен елемент масиву недоцільно, а досить запам'ятати адресу початку масиву, наприклад, у регістрі **BX** або **BP**.

Тоді адресу будь-якого елементу масиву можна визначити як суми базової адреси і цілої константи **[BP] + N**, де N - кількість байт від початку масиву (зміщення). Якщо початковий адресу масиву записати у регістр **BP**, то другий елемент масиву можна переслати у акумулятор так:

```
MOV AX, [BP] + 2
```

Виконавча адреса буде визначена як сума вмісту **BP** і певного зміщення. Такий метод адресації називається *адресацією за базою.*

Вище наведена запис має й інші еквівалентні форми:

```
MOV AX, 2[BP]
MOV AX, [BP + 2]
```

Ясно, що зміщення може бути і від’ємним.

Тобто, вибравши необхідне зміщення, можна довільно адресувати елементи масиву.

1.4.6. Пряма адресація з індексуванням

Якщо зафіксувати базову адресу елементів даних певною міткою, тоді дістатися до інших елементів даних можна за допомогою індексних регістрів SI, DI.

Наприклад:

```
MOV DI, 2
MOV AX, TABLE
```

Виконавча адреса визначиться як сума адреси TABLE і регістру DI. В даному випадку у акумулятор AX буде переслано друге слово після TABLE.

Це пряма адресація з індексуванням. Цей метод адресації зручно використовувати для регулярної обробки масивів. Якщо другу команду розмістити у циклі і там же додавати 2 до DI, то можна обробляти всі елементи масиву TABLE.

1.4.7. Адресація за базою з індексуванням

Для обробки двовимірних масивів зручно використовувати адресацію за базою з індексуванням, коли виконавча адреса дорівнює сумі значень базового регістру, індексного регістру і зміщення.

```
MOV AX, VALUE [BX] [DI]
```

Тут VALUE - іменована константа, а не адреса комірки. Замість імені змінної можна задавати адресну константу.

Наприклад:

```
MOV AX, 2[BP] [SI]
```

Операнди у дужках можна записувати по-різному:

```
MOV AX, [BP + 2 + SI];
MOV AX, [SI + BP + 2];
MOV AX. [BP] [SI + 2].
```

Ясно, що зміщення може не бути. Такі основні методи адресації. Проте, це далеко не всі методи адресації.

Лекція 5

1.5 Особливості формування машинних команд

1.5.1. Формат машинної команди

За мнемонікою команди транслятор створює машинну команду. Вона може займати від 1 до 6 байт в залежності від того, які режими адресації застосовуються. Зміщення і власне дані також записуються в команду. Якщо вони в межах -128 +127, то займають байт, інакше - слово.

Найкоротші команди – один байт, для тих команд, для яких операндів визначається самою командою (команди для роботи з бітами регістра прапорців чи **CLC** – очистити біт прапорця CF).

Найдовші команди, коли в них використовується 16-бітове зміщення (DISP) або безпосередньо 16-бітові дані (DATA).

Тому для різних команд і формати команд будуть різними, тобто які коди і де вони розміщуються.

Розглянемо формат двоопернадної команди. В першому байті записується код операції, в другому – режим адресації. Інші байти виділяються під зміщення (DISP) чи безпосередньо дані (DATA).

7 2	1	0
КОП	**S**	**W**

MOD	**REG**	**R/M**

В першому байті крім коду операції є два однобітові індикатори: **W** – визначає, над якою одиницею даних виконується команда. **W** = 1 – над словом, **W** = 0 - над байтом.

Бітів **S** показує чим являється даний регістр – операндом-джерелом (**S** = 0) чи приймачем (**S** = 1).

Регістри кодуються таким чином:

КОП \ W	W = 1	W = 0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

Сегментні регістри кодуються так:

00	DS
01	CS
10	SS
11	ES

Режим адресації другого операнда визначається кодами в полях **MOD** і **R/M** (register-memory).

Поле **MOD** визначає байт, що саме закодовано в полі слово **R/M**. Якщо **MOD** = 00 – зміщення нема; **MOD** = 01 – зміщення, **MOD** = 10 – зміщення. Коли ж **MOD** = 11, це значить, що в поле **R/M** записано код другого регістру.

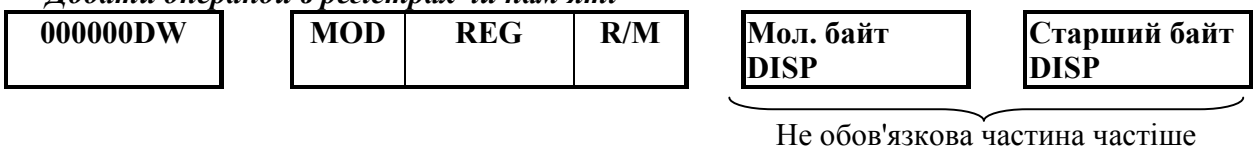
Таблиця

Коди режимів адресації

R/M \ MOD	00	01	10
000	[BX] + [SI]	[BX + SI] + DISP 8	[BX + SI] + DISP 16
001	[BX] + [DI]	[BX + DI] + DISP 8	[BX + DI] + DISP 16
010	[BP + SI]	[BP + SI] + DISP 8	[BP + SI] + DISP 16
011	[BP + DI]	[BP + DI] + DISP 8	[BP + DI] + DISP 16
100	[SI]	[SI] + DISP 8	[SI] + DISP 16
101	[DI]	[DI] + DISP 8	[DI] + DISP 16
110	disp 16	[BP] + DISP 8	[BP] + DISP 16
111	[BX]	[BX] + DISP 8	[BX] + DISP 16

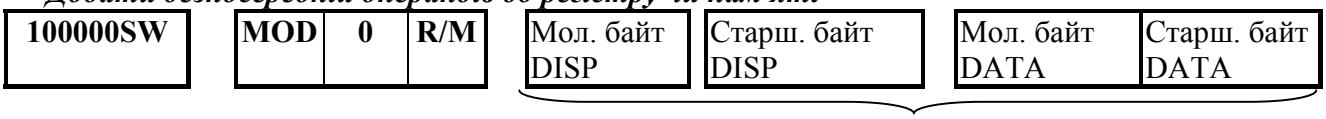
Особливістю МП є те, що для однієї команди може бути кілька кодів та форматів в залежності від режиму адресації чи операндів, які там використовуються. Наприклад, для команди **ADD** (додати):

Додати операнди в регістрах чи пам'яті



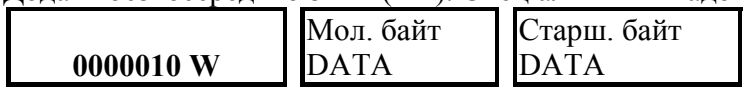
Не обов'язкова частина частіше залежить від поля **MOD**

Додати безпосередній операндів до регістру чи пам'яті



Не обов'язкова частина частіше залежить від поля

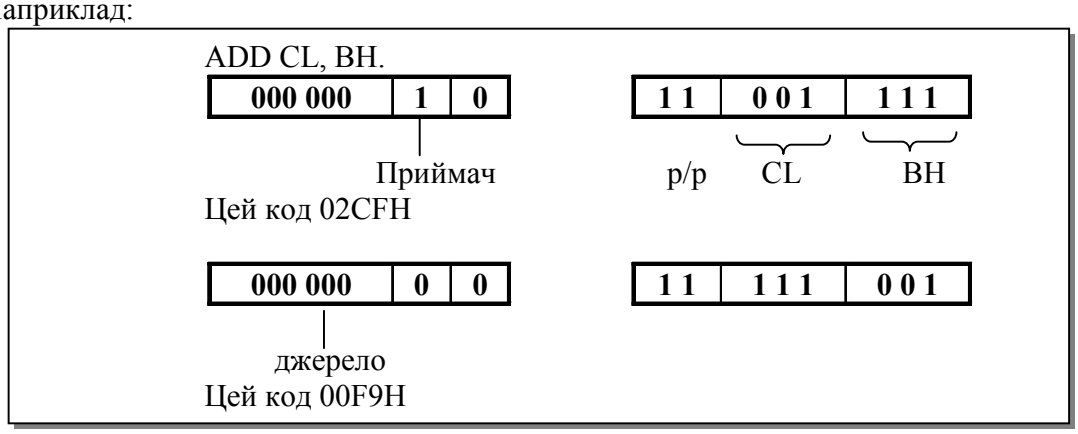
Додати безпосередньо з **AX** (**AL**). Спеціальний випадок для акумулятора:



Тобто, команда **ADD** має 3 варіанти коду і різні формати. Як видно, в форматі є лише одне поле для кодування комірки пам'яті. Тому в двоадресних командах може бути лише одна операнда – комірка пам'яті, а не дві. Отже, додати вміст однієї команди до іншої за одну команду не можна.

Інколи, одна і та ж команда може бути закодована транслятором по-різному.

Наприклад:



Поля із 6-ти бітів для кодування всіх команд не достатньо. Тому деякі команди об'єднуються в групу і в першому байті кодується група команд, а команда утворюється в другому байті. Наприклад, формати команд, з безпосередньою адресацією.

Регістр – безпосередній операнд

КОП	S	W

1 1	КОП	REG

Не обов'язкова частина

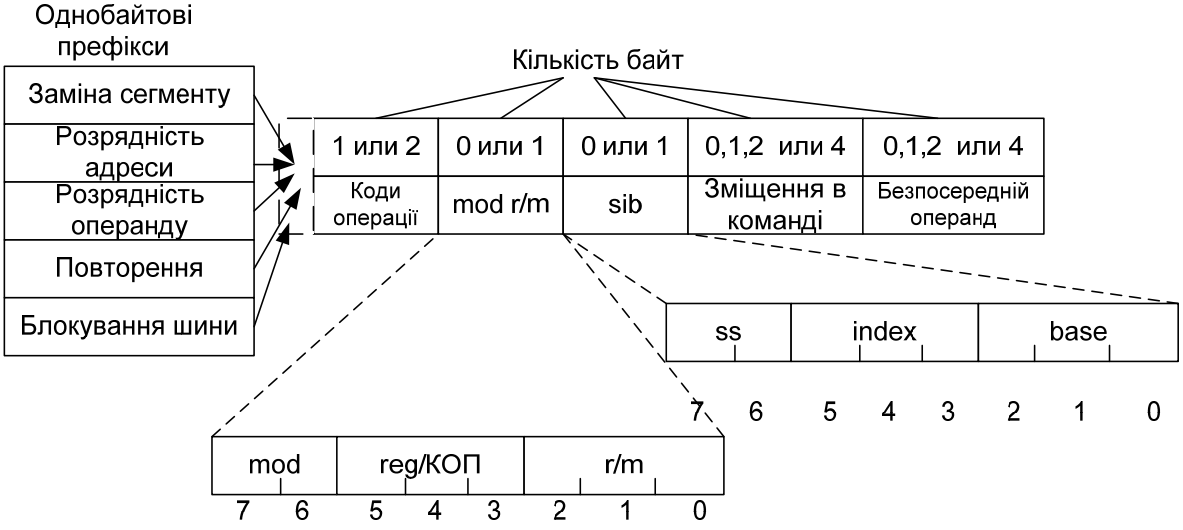
Пам'ять – безпосередній операнд

КОП	S	W

1 1	КОП	MEM

Крім відмічених частин може бути кілька префіксів, що може збільшити розмір від 1 до 5 байт (рис. 1.5.1).

Формат машинної команди



Розрядність адреси:
уточнює розрядність адреси (32 чи 16-розрядна)

Розрядність операнду:
66h — 16-розрядний;
67h — 32-розрядний

Префікс повторення:
- ***безумовні*** (rep — 0f3h), змушують повторюватися ланцюгову команду певну кількість разів;
- ***умовні*** (repe/repz — 0f3h, repne/repnz — 0f2h), при зацикленні перевіряють деякі прапорці і в результаті перевірки можливий достроковий вихід із циклу

Код операції (КОП)

Обов'язковий елемент, який описує операцію, яку виконує команда. Багатьом командам відповідає декілька кодів операцій, кожен з яких визначає нюанси виконання операції.

Байт mod r/m - режим адресації

Значення цього байту визначає використосовану форму адреси операндів:
mod = 00 – поле зміщення в команді відсутнє;
mod = 01 – поле зміщення в команді присутнє;
mod = 11 – операндів в пам'яті немає, вони знаходяться в регістрах;
reg/КОП - визначає регістр, що знаходиться в команді на місці першого операнду, або можливе розширення коду операції
r/m *використовується спільно з полем* ***mod*** *визначає один регіст, або базові та індексні регістри.*

Байт масштаб-індекс-база (байт sib)

Розширює можливості адресації операндів:
ss – в ньому розміщується масштабний множник для індексного компоненту **index**
index — використовується для збереження індексного регістру;
base — використовується для збереження базового регістру.

Поле зміщення в команді

8, 16 чи 32-розрядне ціле число зі знаком, що являє собою, повністю або частково, значеннння ефективної адреси операнду.

Поле безпосереднього операнду

Необов'язкове поле являє собою 8, 16 чи 32-розрядний безпосередній операнд. Наявність цього поля, відображається на значенні байту **mod r/m.**

Рис. 1.5.1. Формат машинної команди

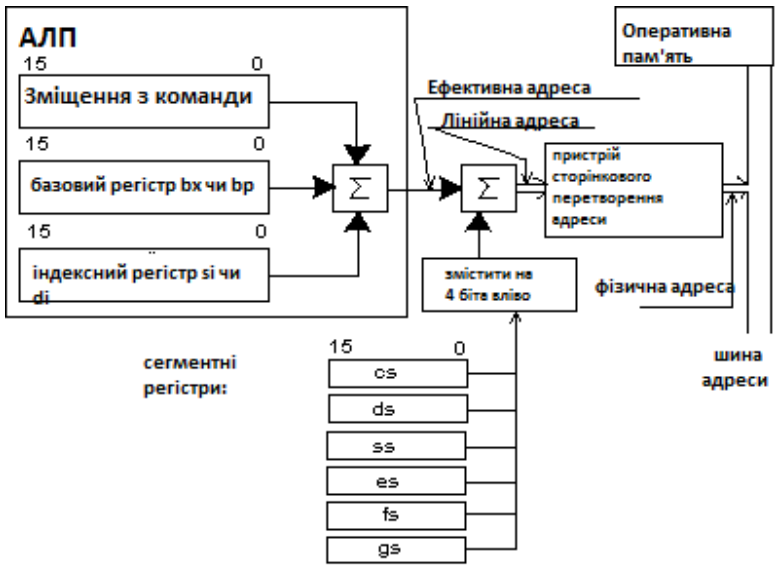


Рис. 1.5.2. Механізм формування фізичної адреси в реальному часі

1.5.2. Оператори

Програма на асемблері складається з окремих рядків-операторів, які описують виконувані операції. Оператором може бути *команда* чи псевдооператор (директива).

Команда – це умовне (символьне) позначення машинних команд. Тобто, команда каже ЕОМ, що їй треба виконати. Команди після трансляції перетворюються в машинні коди.

На відмінну від команди, *псевдооператори* чи *директиви* мають допоміжний характер. Вони повідомляють транслятору, що робити з командою. Як правило, вони не перетворюються в машинні команди. Команди виконуються під час обчислень, а псевдооператори – під час трансляції.

Кожна команда може мати 4 поля:

[Мітка:] Мнемокод [Операнд] [, Операнд] [; Коментар]

Із них обов'язкове є лише поле мнемокоду, всі інші частково або повністю можуть бути відсутніми.

Наприклад:

```
COUNT: MOV AX,DI; відправити DI в акумулятор
```

Мітка присвоює ім'я команді. На цю мітку можна передавати управління з інших місць програми, тобто, на неї можуть посилатися інші команди.

Масив може складатися не більше ніж з 31-го символу і закінчується “:”. Сюди можуть входити великі та маленькі латинські букви від A до Z, чи від a до z (великі і малі букви не розрізняються), цифри від 0 до 9 чи спеціальні знаки ? . @ _ $. Мітка може починатися будь-яким символом крім цифри. Не можна включати розділові знаки (пропуски), виняток нижнє підкреслення.

Поле мнемокоду – включає мнемокод чи мнемоніку команди.

Операнди, як зазначалося, мають свої назви: *приймач і джерело*. Після виконання команди змінюється приймач, а джерело залишається незмінним. Операнди розділяються комою.

Коментарі – все, що стоїть справа від «**;**». Можуть бути в рядку, а також займати цілий рядок. Поля відділяються пропусками.

Псевдооператори – керують роботою транслятора, а не мікропроцесора. З їх допомогою визначають сегменти і процедури (підпрограми), дають імена командам та елементам даних, резервують робочі місця в пам'яті.

Псевдооператори можуть мати 4 поля:

[Ідентифікатор] Псевдооператор [операнд] [; коментар]

Поля в дужках можуть бути відсутніми. В Макроасемблері нараховують близько 60 псевдооператорів. Розглянемо найпоширеніші.

1.5.3. Псевдооператори даних

Їх можна розділити на 5 груп:

1. Визначення ідентифікаторів

Дозволяють присвоїти символічне ім'я виразу, константі, адресі, іншому символічному імені. Після цього можна використовувати це ім'я.

Наприклад:

```
K          EQU   1024; константа
TABLE      EQU   DS: [BP] [SI]; адреса
SPEED      EQU   RATE ; синонім
COUNT      EQU   CX; ім'я регістру
```

Операндом в цьому псевдооператорі в загальному випадку може бути вираз:

```
DBL  SPEED EQU 2*SPEED
```

Тобто, вираз може включати деякі простіші арифметичні та логічні дії. Якщо вираз включає константу, то за замовчуванням вона рахується десятковою.

Шіснадцяткові константи – справа літера H (2FH). Коли така константа починається з літери, то ліворуч треба поставити 0 - 0FH, а не FH;

Вісімкові константи – з літерою Q: 256Q;

Двійкові – з літерою B - 01101B.

Очевидно, що на відмінну від мов високого рівня, тут вираз виконується під час трансляції.

Крім псевдооператора EQU можна використати знак “ = ”.

```
CONST = 59;
CONST = 98;
CONST = CONST + 2.
```

Таке ім'я може задавати лише числову константу, а не символьну чи якусь іншу. Це ім'я можна перевизначити, а визначену через **EQU** – НЕ МОЖНА.

2. Визначення даних

Коли комірка використовується для збереження даних, їй можна присвоїти ім'я за допомогою псевдооператорів **DB**, **DW**, **DD** (Define Byte, Word, Double Word).

Загальна структура:

[ім'я] псевдооператор вираз [,]

[,] – один чи кілька виразів через кому.

```
MAX DB 57
WU_MAX DW 5692
A_TABLE DW 25 -592, 643, 954 - масив
```

Коли значення змінних завчасно невідоме, але буде використане для результатів, то в полі виразу треба поставити «**?**».

```
LAMBDA DW ?
```

Для тексту можна використовувати псевдооператор **DB**.

```
POLITE DB ‘Введіть данні знову’
```

Якщо записуються однакові дані, то можна використовувати команду **DUP** (duplicate) - повторити.

```
BETA DW 15 DUP(0)     чи     GAMA DW 3 DUP (4DUP(0))
```

Зрозуміло, що цю операцію можна використовувати для резервування пам'яті:

```
ALPHA DW 20 DUP (?)
```

Змінні можна використовувати для збереження не тільки даних, але й адрес. Якщо певна змінна має назву THERE, тоді псевдооператори

```
THERE DB 123
……
NEAR_THERE DW THERE
FAR_THERE DD THERE
```

під іменем NEAR_THERE записують 16-бітівну адресу THERE, тобто, її зміщення, а під іменем FAR_THERE – 32-бітову адресу THERE, тобто, сегмент + зміщення.

3. Псевдооператори визначення сегменту і процедури

Як зазначалося, програма може складатися з декількох сегментів: коду, даних, стеку, додаткового сегменту.

Для поділу програми на сегменти використовуються псевдооператори **SEGMENT** та **ENDS**. Їх структура:

Ім'я SEGMENT [атрибути]

……

Ім'я ENDS

Наприклад:

```
DATASEG SEGMENT
  A DW 500
  B DW -258
  SQUARE DB 54, 61, 95, 17
DATASEG ENDS
```

В сегменті даних визначаються імена даних та резурвується пам'ять для результатів.

У псевдооператора **SEGMENT** можуть бути 3 атрибути:

Имя SEGMENT [вирівнювання] [об'єднання][клас] [розмір сегменту]

Вирівнювання визначає, з яких адрес можна розміщувати сегмент. Часто це атрибут **PARA**, який вказує, що сегмент треба розмістити на межі параграфу, тобто, початкова адреса кратна 16. За замовчуванням приймається PARA. Але можуть бути й інші:

BYTE	x 1
WORD	x 2
DWORD	x 4
PARA	x 16
PAGE	x 256
MEMPAGE	x 1024

Об'єднання визначає спосіб опрацювання сегменту при компонуванні.

PRIVATE – за замовчуванням, сегмент повинен бути відділений від інших сегментів.

PUBLIC – всі сегменти з однаковим іменем та класом завантажуються в суміжні області. Всі вони будуть мати *одну початкову адресу*.

STACK – для компонувальника аналогічно PUBLIC. В будь-якій програмі повинен бути сегмент з атрибутом STACK.

Клас – цей атрибут може мати будь-яке коректне ім'я, не обмежене апострофами “ ”. Атрибут використовується компонувальником для обробки сегментів з однаковими іменами та класами. Часто використовуються імена ‘CODE’ та ’STACK’.

```
STACK_SEG SEGMENT PARA STACK ‘STACK’
    MAS DW 20 DUP (?)
STACK_SEG ENDS
```

Як зазначалося, процесор використовує регістр **CS** для адресації сегменту коду, **SS** – сегменту стеку, **DS** - даних, **ES** - додатковий.

Оскільки, транслятор не розуміє тексту, то йому необхідно повідомити призначення кожного сегменту. Для цього існує псевдооператор **ASSUME**, який має вигляд:

ASSUME SS:ім'я стеку, DS:ім'я даних, CS:ім'я коду.

Наприклад:

```
ASSUME SS:STACK  SEG, DS:DATA  SEG, CS:CODE  SEG.
```

Якщо певний сегмент не використовується, то можна його упустити або записати:

DS:NOTHING

Ця директива лише повідомляє *транслятору* що з чим зв'язувати. Але не завантажує відповідні адреси в регістри. Це повинен зробити сам програміст в програмі.

Директива реалізується в сегменті коду спочатку.

Сегмент коду може вміщати одну або кілька процедур. Окрема процедура почиається псевдоопертором PROC:

ім'я PROC [атрибут]

ім'я ENDP і закінчується

Якщо процедура передбачає повернення до точки виклику, то перед **ENDP** повинна стояти директива **RET** (Return From Procedure). Тоді процедура стає підпрограмою.

Атрибутом процедури може бути **NEAR** (близький) и **FAR** (далекий).

```
CALC PROC
.......
        RET
CALC ENDP
```

NEAR-процедура може бути викликана лише з цього сегменту.

Процедуру з атрибутом **FAR** можна викликати з будь-якого сегменту команд. Основна процедура повинна мати атрибут **FAR**, для створення .exe файлу.

1.5.4. Підпрограмні сегменти

В програмі може бути кілька сегментів з однаковим іменем. Рахується, що це один сегмент, який з певних причин записаний частинами.

Параметри директиви **SEGMENT** потрібні для великих програм, які складаються з декількох файлів. Для невеликої програми, яка складається з одного файлу, ці параметри не потрібні, крім деяких випадків.

Наприклад, маємо такий ряд сегментів:

```
A SEGMENT
   A1 DB 400h DUP(?)
   A2 DW 8
A ENDS
;
B SEGMENT
   B1 DW A2
   B2 DD A2
B ENDS
;
C SEGMENT
ASSUME ES:A, DS:B, CS:C
L: MOV AX, A2
   MOV BX, B2
……….
C ENDS
```

Сегменти, розміщені на межі параграфу, тобто, адреса кратна 16. Якщо A розміщено 1000h, то він займе місце до 1402h. Наступна адреса - 1403h не кратна 16, тому сегмент B розміститься з адреси 1410h. Під сегмент B буде відведено 6 байт, а сегмент C розміститься з адреси 1420h.

Значення імені сегменту

Значення імені сегменту являється номером, який відповідає сегменту пам'яті, тобто, перші 16 бітів початкової адреси заданого сегменту. Тобто, **A** набуде значення 1000h, **B** - 1410h, **C** - 1420h. Зберігаючи в тексті ім'я сегменту, асемблер буде заміняти його на відповідну величину. Наприклад:

```
MOV BX, B     відповідає     MOV BX, 1410h
```

Порівняйте MOV BX, A2.

Отже, в мові асемблер імена сегментів – це константні вирази, а не адресні. Тому, команда запише до BX адресу 1410h, а не вміст слова за цією адресою.

Початкове завантаженння сегментних регістрів

Директива **ASSUME** відмічає, з якими сегметними регістрами пов'язувати сегменти. Регістри **DS** і **ES** повинен завантажити початковими адресами сам програміст.

Нехай, регістр **DS** потрібно встановити на початок сегменту **B**. Оскільки, ім'я сегменту – це константа, то безпосередньо в **DS** її відправити не можна, а треба через регістр загального значення:

```
MOV AX, B
MOV DS, AX
```

Аналогічно завантажується і регістр **ES**.

Регістр **CS** завантажувати не потрібно. Це зробить сама операційна система перед тим, як передати управління програмі. Відносно регістру **SS** існує дві можливості:

- По-перше, це можна зробити так як для регістрів **DS** і **ES**. Але тут потрібно записати початкову адресу в регістр **SS**, а в покажчик стеку **SP** – кількість байт під стек.
- По-друге – це можна поручити операційній системі. Для того в відповідній директиві **SEGMENT** треба відмітити атрибут **STACK**.

1.6. Структура програми

Взаємне розміщення сегментів програми може бути довільним. Якщо сегменти даних розміщені після сегменту коду, то в сегменті коду є посилання на адресу сегменту даних. Тобто, відразу не можна визначити зміщення цих адрес. Отже будемо мати посилання вперед.

Тому для зменшення кількості посилань рекомендується сегменти даних і стеку розміщувати перед сегментом коду.

Відносно сегменту стеку **SS**, навіть якщо програма і не використовує його, то створити такий сегмент в програмі необхідно. Тому що, стек програми використовується операційною системою при опрацюванні переривань, наприклад, при натисканні кнопок. Рекомендований розмір стеку - 128 байт. Тому, програма на асемблері має таку структуру:

```
Title EXAMPLE   ;заголовок
; сегмент даних
dat segment
mas dw 1,3,56,91
res dw 10 dup(?)
dat ends
; сегмент стеку
st segment stack ‘stack’
db 128 dup(?)
st ends
; сегмент коду
cod segment ‘code’
      ASSUME DS:dat, SS:st, CS:cod
beg proc far
      <оператори>
     ret
beg endp
cod ends
      end beg
```

В загальному випадку в сегменті даних можна розміщувати і команди, а в сегменті коду - дані. Але краще цього не робити, бо виникнуть проблеми з сегментацією.

END – кінець програми. Якщо програма з одного файлу, то в цьому рядку додається ім'я початкової виконуваної адреси - beg. Якщо з кількох файлів, то тільки в одній програмі відмічається ця адреса. В інших - лише end.

1.6.1. Скорочений опис деяких елементів

Є два види трансляторів асемблеру - **MASM** фірми Microsoft та **TASM** фірми Borland. Останній може працювати в режимі MASM і IDEAL.

Для простих програм, які складаються з одного сегменту даних, одного сегменту стеку та одного сегменту коду є можливість спростити директиви опису сегменту. Для цього спочатку наводиться директива **masm** чи **ideal** режиму роботи транслятора **TASM**. Далі відмічається модель пам'яті **model small**, яка частково виконує функції директиви **ASSUME**.

Наприклад:

```
Masm ; режим роботи транслятора TASM
model small ; модель пам'яті
.data ; заголовок сегменту даних
mas dw 10 DUP (?)
.stack ; заголовок сегменту стеку
db 256 dup (?)
.code ; заголовок сегменту коду
main proc
      mov ax, @data ; адреса сегменту стеку до ax
      mov ds, ax
; текст програми
      mov ax, 4c00h    ; те саме, що і RET
       int 21h
main endp
end main
```

Таблиця

Спрощені директиви визначення сегменту

Режим MASM	Режим IDEAL	Описание
.code [ім’я]	Codeseg[ім’я]	Початок чи продовження сегменту коду
.data	Dataseg	Початок чи продовження сегменту ініціалізації даних
.const	Const	Початок чи продовження сегменту констант
.data?	Udataseg	Початок чи продовження сегменту неініціалізованих даних
.stack [розмір]	Stack[розмір]	Початок сегменту стеку
.fardata[ім’я]	Fardata[ім’я]	Ініціалізовані дані типу FAR
.fardata?[ім’я]	Ufardata[ім’я]	Неініціалізовані дані типу FAR

Ідентифікатори, які створює директива **MODEL**:

@ code – фізична адреса (зміщення) сегменту коду
@ data – фізична адреса (зміщення) сегменту даних
@ fardata – фізична адреса (зміщення) сегменту даних типу **far**
@ fardata? – фізична адреса (зміщення) сегменту ініціалізованих даних типу **far**
@ curseg – фізична адреса (зміщення) сегменту неініціалізованих даних типу **far**
@ stack – фізична адреса (зміщення) сегменту стеку

Моделі пам'яті

Модель	**Тип коду**	**Тип даних**	**Призначення моделі**
TINY	Near	Near	Код і данні в одній групі DGROUP для створення .com-програм
SMALL	Near	Near	1 сегмент коду. Дані в одну групу DGROUP
MEDIUM	Far	Near	Код займає ***n*** сегментів, по одному в кожному модулі. Всі передачі управління типу **far**. Дані в одній групі; всі посилання на них типу **near**
COMPACT	Near	Far	Код в 1 сегменті; посилання на данні типу **far**
LARGE	Far	Far	Код в ***n*** сегментах, по одному на кожен об'єднаний модуль

Модифікатор директиви **model** дозволяє визначити деякі особливості вибраної моделі пам'яті:

Use 16 - 16-бітові сегменти
Use 32 - 32-бітові сегменти
DOS - програма в MS DOS
Повний і спрощений опис не виключає один одного, але в повному більше можливостей.

Лекція 6
Розділ 2. Команди МП 8088/86

2.1. Склад команд

Мікропроцесор має 92 команди, які можна поділити на 7 груп:

1. Команди пересилання даних між регістрами, осередками і портами введення/виведення;

2. Арифметичні команди;

3. Команди над бітами, які здійснюють зміщення і логічні операції;

4. Команди передачі керування, виклику процедур і повернення з процедури;

5. Команди обробки рядків;

6. Команди переривання для обробки специфічних подій;

7. Команди управління процесором - встановлення і скидання прапорців стану, зміни режиму функціонування МП.

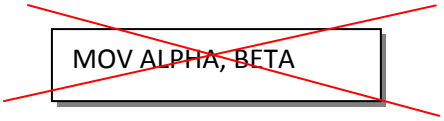
2.1.1. Команди пересилання даних

Команда **MOV** – найчастіше вживається в програмі. Її можна застосовувати для пересилання:

```
MOV AX, CX ; з регістра в регістр
MOV AX, TABLE ;з пам'яті в регістр
MOV TABLE, AX ; з регістра в пам'ять
MOV DS, AX ;з регістрав регістр сегменту
MOV AH, AL ;пересилка байт
MOV AX, -40 ; константу в регистр
MOV BETA, 2Fh ; константу в память
```

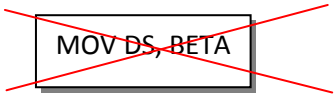
НЕ МОЖНА здійснювати такі пересилання:

З пам'яті в пам'ять. Як зазначалося, в двухадресних командах не можна використовувати пряму адресацію в двох операндах. Тому пересилку пам'ять-пам'ять можна здійснити двома командами:



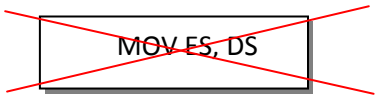

```
MOV AX, BETA
MOV ALPHA, AX
```

Вміст комірок НЕ МОЖНА пересилати безпосередньо в регістр сегмента, а тільки через регістр загального призначення:



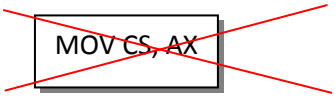

```
MOV AX, BETA
```

НЕ МОЖНА пересилати дані з одного регістра сегмента в інший, а тільки через регістр загального призначення:




```
MOV AX, DS
MOV ES, AX
```

НЕ МОЖНА використовувати регістр CS як приймач, тобто, **НЕ МОЖНА**:



Стек автоматично створюється для роботи з підпрограмами. Але оскільки в МП лише 4 регістра загального призначення, то часто доводиться запам'ятовувати вміст регістрів, щоб звільнити регістри і виконати інші дії. Саме для цього використовуються команди:

PUSH джерело; заслати в стек

POP приймач ; зчитати зі стека

Наприклад:

```
PUSH SI
PUSH DS
PUSH CX
PUSH ALPHA
PUSH DELTA [BI + SI]
```

Як зазначалося, дані засилаються на вершину стека, тому при їх зчитуванні необхідно додержуватись відповідної послідовності. Наприклад:

```
PUSH AX
PUSH BX
```

	DS	**SS:02F8**
	BS	**SS:02FA**
SP - 2	**AX**	**SS:02FC**
SP -		**SS:02FE**

Оскільки зверху знаходиться **DS**, а нижче - **BX** і **AX**, то відновлення регістрів потрібно здійснити в протилежному порядку

```
POP DS
POP BX
POP AX
```

Команди PUSH-POP можна використовувати для обміну між сегментними регістрами:

```
PUSH DS
POP ES
```

При цьому не використовуються регістри загального призначення. Але виконання команд буде довшим: пара PUSH-POP реалізується за 26 тактів, а дві команди MOV за 4 такти.

2.1.2. Команда обміну XCHG

Назва походить від англійського слова exchange - обміняти. Використовується

```
XCHG BX, AX
XCHG AH, BL
XCHG AX, TABLE
```

для обміну вмістом двох регістрів або регістра і пам'яті.

~~XCHG CS, DS~~

~~Не можна~~ використовувати для обміну між *сегментними регістрами*.

2.1.3. Командиобміну з портами

IN акумулятор, порт

OUT порт, акумулятор

Акумулятор – це регістр **AX** при обміні **словами** і **AL** при обміні **байтами**. Порт визначається своїм номером від 0 до 256. Можна безпосередньо відзначати номер порту або дати йому ім'я.

```
PORT_NUM EQU 210
IN AX, 200
IN AL, PORT_NUM
OUT DX, AX
OUT 200, AL
```

Також номер порту можна записати в регістр **DX**.

2.1.4. Команда LEA – завантаження ефективної адреси

LEA(load effective address – завантажити виконавчу адресу) пересилає зміщення комірки пам'яті в:

1. певний 16-бітовий регістр ЗП

2. регістр покажчика

3. індексний регістр

LEA регістр 16, пам'ять 16

На відміну від команди MOV з операцією OFFSET, операнд пам'ять 16 може бути індексованим, що забезпечує гнучкість адресації.

Наприклад:

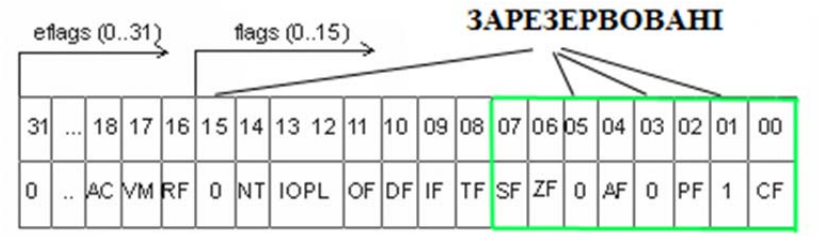
```
LEA BP, TABLE [DI]
```

Якщо в **DI** міститься 8, то в **BP** буде заслана адреса TABLE + 8.

2.1.5. Команди пересилання прапорців

Можна пересилати в регістр **AH** молодший байт регістра прапорців **F** командою **LAHF**;

LAHF; F→AH



Зворотнє пересилання з **AH** в молодший байт регістра F - SAHF:

SAHF; AH→F

Вміст регістру прапорців можна також пересилати в стек командою **PUSHF**, а в зворотному напрямку - **POPF**.

Це потрібно для захисту регістра прапорів від зміни при виклику процедур. Якщо немає впевненості, що процедура не змінить регістр прапорців, то його потрібно захистити (зберегти).

Приклад:

```
PUSH AX
PUSH DI
PUSHF
CALL SORT
POPF
```

Лекція 7

2.2 Арифметичні команди

Цілочисельний обчислювальний пристрій підтримує трохи більше десятка арифметичних команд.

В МП 8088/86 є група команд для реалізації 4 арифметичних дій (додавання, віднімання, множення, ділення) над цілими числами. При цьому число може бути записано як в байт, так і в слово. Зміст байта або слова можна розглядати як число зі знаком, або без нього. Деякі різновиди цих команд дозволяють реалізувати арифметичні дії для чисел підвищеної точності. Крім цього, за допомогою звичайних арифметичних операцій можна реалізувати дії над числами в ВСД-форматі. Ясно, що результат буде правильним. Проте, його можна скорегувати. Тому передбачено для такого випадку команди корекції результатів. Їх розглядати ми не будемо. Нагадаємо, що більшість двохоперандних команд діють так, що змінюється операнд-приймач, а операнд-джерело залишається незмінним.

1 - після виконання команди прапорець встановлюється (дорівнює 1);
0 - після виконання команди прапорець скидається (дорівнює 0);
r - значення прапорця залежить від результату роботи команди;
? - Після виконання команди прапорець не визначений;
пробіл - після виконання команди прапорець не змінюється;

2.2.1 Команди додавання

Існують 3 команди:
ADD (add -- додати);
ADC (add with carry) – додати з переносом;
INC (increment) – додати одиницю.

ADD операнд_1, операнд_2 — команда додавання з принципом дії: ***операнд_1 = операнд_1 + операнд_2***

```
ADD AX, CX          AX + CX → AX
```

У цій команді можуть бути наведені 2 регістра (загального призначення), один операнд може бути словом (2 байти), з усіма можливими режимами адресації – приймач і безпосередні дані - як джерело.

```
ADD AX, ALPHA
ADD ALPHA, AX
ADD AH, 20
ADD ALPHA, 30
```

Ця команда впливає на 6 прапорців.

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
r	r	r	r	r	r

ADC операнд_1, операнд_2 - команда додавання з урахуванням прапора переносу **cf.**

Принцип дії команди:
операнд_1 = операнд_1 + операнд_2 + значення_cf

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
r	r	r	r	r	r

INC операнд - збільшення значення операнда в пам'яті або в регістрі на 1

11	07	06	04	02
OF	SF	ZF	AF	PF
r	r	r	r	r

2.2.2. Команди віднімання

Аналогічно командам додавання, існують дві команди віднімання
SUB (substract) - відняти;
SBB (substract with borrow) - відняти з займом;
DEC (DECrement) - відняти одиницю.

SUB операнд_1, операнд_2 - цілочисельне віднімання
операнд_1 = операнд_1-операнд_2

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
r	r	r	r	r	r

SBB операнд_1, операнд_2 - цілочисельне віднімання з урахуванням результату попереднього віднімання командами **SBB** і **SUB** (станом прапора переносу **CF):**

1. виконати додавання операнд_2 = операнд_2 + **(CF);**
2. виконати віднімання операнд_1 = операнд_1-операнд_2**;**

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
r	r	r	r	r	r

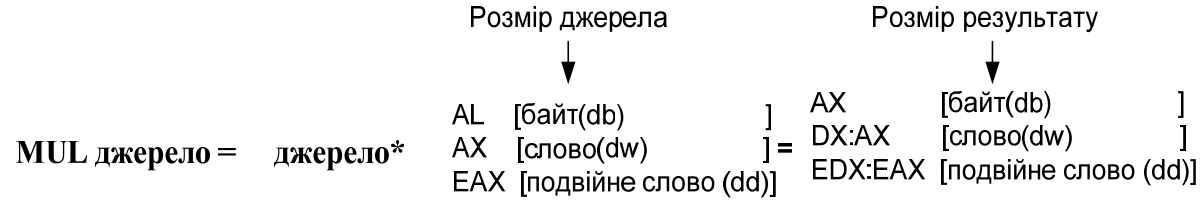
DEC операнд - зменшення значення операнда в пам'яті або в регістрі на 1

11	07	06	04	02
OF	SF	ZF	AF	PF
r	r	r	r	r

2.2.3. Команди множення

Існують дві команди:
MUL (multiply) - множення без знаку;
IMUL (integer multiply) – множення зі знаком.

MUL джерело



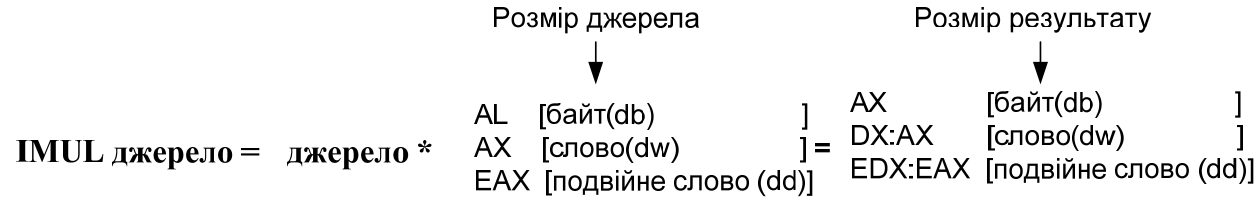
Стан прапорців після виконання команди
(якщо ***старша половина результату – нульова AH=0;DX=0;EDX=0):***

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
1	?	?	?	?	0

Стан прапорців після виконання команди
(якщо ***старша половина результату – ненульова AH≠0;DX≠0;EDX≠0):***

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
1	?	?	?	?	1

IMUL джерело



IMUL множ_1, множ_2 **(** IMUL AX, 5 AX = AX*5 **)**
IMUL рез-т, множ_1, множ_2 **(** IMUL DI, AX, 5 DI = AX*5 **)**

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
r	?	?	?	?	r

2.2.4 Команди ділення

Зазначимо, що ділення виконується як ділення цілих чисел, тобто, отримаємо частку (ціле) і залишок (ціле). Отже, 19: 5 = 3 і 4 - залишок. Ніяких дійсних чисел не отримаємо.
DIV (divide) - ділення чисел без знаку;
IDIV (integer divide) - ділення цілих чисел (зі знаком).

DIV дільник

Розмір діленого

ділене / **дільник**	Байт (db)	Слово (dw)	Подвійне слово (dd)
ділене	AX	DX:AX	EDX:EAX
частка	AL	AX	EAX
залишок	AH	DX	EDX

Результати команд ділення на **прапорці не впливає**.
Коли ж частку повністю не можна розмістити в відведеному їй слові або в байті, то здійснюється переривання типу 0 - ділення на 0.

IDIV дільник
Залишок завжди має знак діленого. Знак частки залежить від стану знакових бітів (старших розрядів) діленого і дільника.

2.2.5. Команди зміни знаку

NEG (NEGate operand) - змінити знак операнда

NEG джерело
Стан прапорців після виконання команди (якщо результат нульовий):

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
r	r	r	r	r	0

Стан прапорців після виконання команди (якщо результат ненульовий):

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
r	r	r	r	r	1

2.2.6. Команди розширення знаку

Щоб правильно подати код числа, яке записано в молодшому байті, при зчитуванні його з цілого слова і код числа, записаного в молодше слово, при зчитуванні його з молодшого слова, потрібно розширити знак числа відповідно на старший байт і старше слово.

Для цього існують дві команди:
CBW (convert byte to word) - перетворення байта в слово;
CWD (convert word to double word) - перетворення слова в подвійне слово;

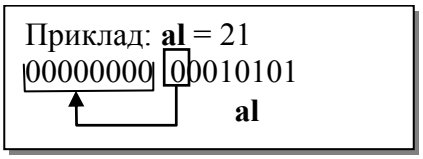
CWDE (convert word to double word Extended) - перетворення слова в подвійне слово;

CDQ (Convert Double word to Quad word) - перетворення подвійного слова в два слова (чотири байти) .

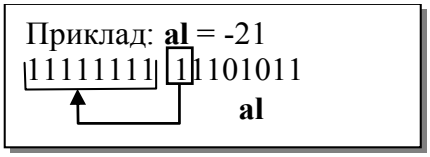
Алгоритм роботи:

CBW — при роботі команда використовує лише регістри **al** и **AX**:

аналіз знакового біту регістру **al**:
якщо знаковий біт **al**=0, то **ah**=00h;



аналіз знакового біту регістру **al**:
якщо знаковий біт **al**=1, то **ah**=0ffh



CWD - при роботі команда використовує лише регістри **AX** і **DX**:
аналіз знакового біта регістра **al**:
якщо знаковий біт **AX** = 0, то **DX** = 00h;
якщо знаковий біт **AX** = 1, то **DX** = 0ffh.
CWDE - при роботі команда використовує лише регістри **AX** і **EAX**:
аналіз знакового біта регістра **AX**:
якщо знаковий біт **AX** = 0, то встановити старше слово **EAX** = 0000h;
якщо знаковий біт **AX** = 1, то встановити старше слово **EAX** = 0ffffh.
виконання команди ***не впливає на прапорці***
CDQ - при роботі команда використовує лише регістри **EAX** і **EDX**:
аналіз знакового біта регістра **EAX**:
якщо знаковий біт **EAX** = 0, то встановити старше слово **EDX** = 0000h;
якщо знаковий біт **EAX** = 1, то встановити старше слово **EDX** = 0ffffh.
виконання команди ***НЕ впливає на прапорці***

Приклад:

Дані команди використовуються для приведення операндів до потрібної розмірності з урахуванням знаку. Така необхідність може, зокрема, виникнути при програмуванні арифметичних операцій.

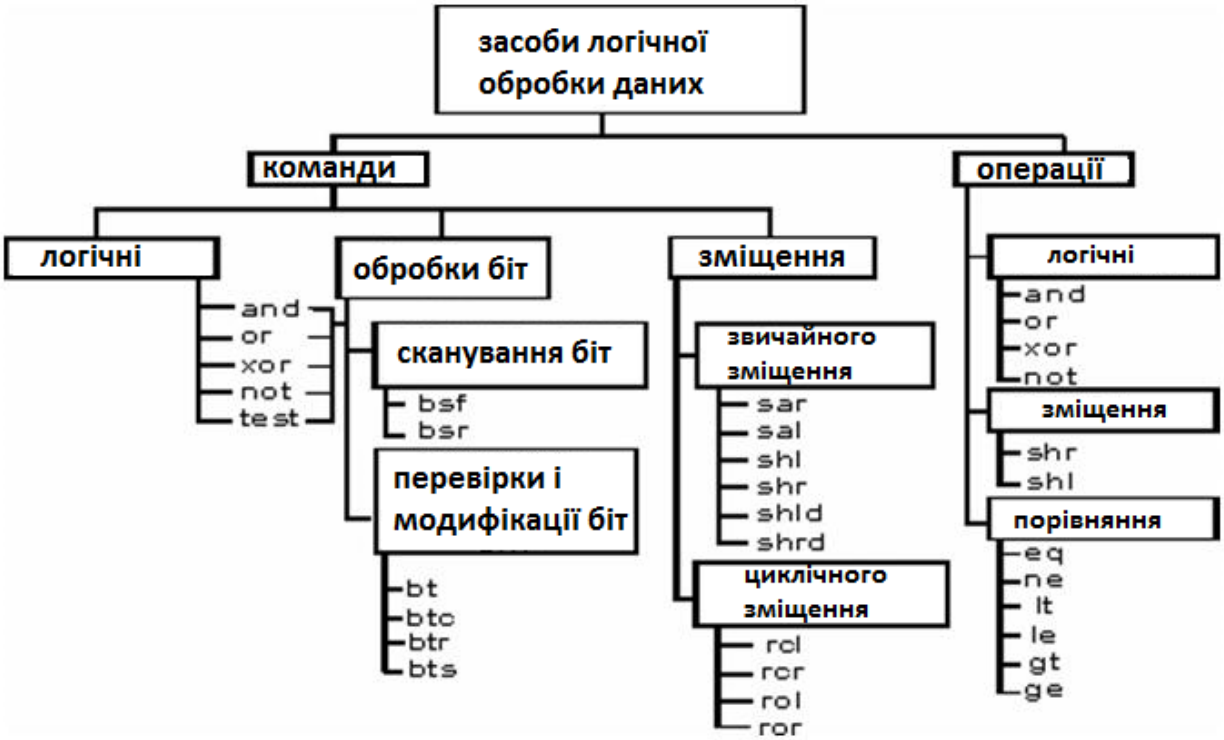
```
.386    ; лише для cwde, cwd була для i8086
        mov     ebx,10fecd23h
        mov     ax,-3   ;ax=1111 1111 1111 1101
        cwde    ;eax=1111 1111 1111 1111 1111 1111 1111 1101
        add     eax,ebx
```

Лекція 8

2.3. Команди логічної обробки даних

Команди діляться на 4 групи:

1. Логічні команди;
2. Команди зміщення;
3. Команди циклічного зміщення;
4. Команди обробки бітів даних.



2.3.1. Логічні команди (AND, OR, XOR, NOT, TEST)

Ці команди реалізують порозрядні операції, тобто, і-ий розряд результату – залежить від і-го розряду операнду. Операції виконуються паралельно над усіма розрядами. “true” – коли буде один, хоча б в одному розряді результату і “false” – коли в усіх бітах результату нулі.

Команди змінюють всі прапорці умов, але частіше звертаємо увагу на прапорець **ZF**. Якщо результат - “true”, то ZF = 0, а якщо “ false”, то ZF = 1.

Операндами логічних операцій можуть бути слова або байти, але **НЕ** одночасно.

Команда побітове ”AND”

AND приймач, джерело - операція логічного множення. Команда виконує порозрядно логічну операцію “І” (кон’юнкцію) над бітами операндів приймач і джерело. Результат записується на місце **приймача.**

Наприклад:

```
AND AX, BX; AX*BX - результат записати в AX
```

В біті приймача встановлюється 1 тоді, коли в відповідних бітах джерела і приймача були 1. Якщо в біті джерела був 0, то у відповідному біті приймача встановиться 0, не залежно від того, що там було. Тому, команда **AND** використовується для селективного встановлення 0 в тих бітах приймача, яким відповідає 0 в джерелі. Підбираючи відповідні біти джерела, впливаємо на визначені біти приймача. Такі дії часто виконуються на бітами для управління пристроями обміну. При цьому оператор-джерело називається маскою, а сама операція - *маскуванням*.

```
; AX=95h, BX=5Bh
AND AX, BX; AX = 11h
```

Наприклад:

```
AND  10010101 – приймач
     01011011 – джерело (маска)
     00010001
     ↑   ↑
  Стан не змінився
```

Операндами команди **AND** можуть бути **байти** чи **слова**. Можуть використовуватися два регістри, регістр зі словом (байтом пам'яті) і безпосереднє значення:

```
AND AL, M_BYTE
AND M_BYTE, AL
AND TABLE [BX], MASK
AND BL, 1101B
```

Відповідно, команда **AND** змінює приймач. Оскільки, біт змінюється лише самим пристроєм, то можна використовувати команду **AND** для перевірки стану пристрою.

Наприклад, порт, 200 з'єднаний з 16-бітовим регістром зовнішнього пристрою і 6-й біт показує ввімкнений (1) чи вимкнений (0) даний пристрій.

```
CHECK: IN AX, 200
       AND AX, 1 000 000B
       JZ CHECK ; НЕ ввімкнено
```

Програма може працювати далі лише тоді, коли пристрій ввімкнений:

Якщо, пристрій вимкнений, то в 6-му біті - 0, і результат команди **AND** - 0, відповідно, коли **ZF** = 1 виконується JZ.

Як тільки в біті буде 1, **ZF** = 0 - команда JZ НЕ виконується

11	07	06	02	00
OF	SF	ZF	PF	CF
0	r	r	r	0

Команда побітове "OR "

OR приймач, джерело — операція логічного додавання.

Команда виконує порозрядно логічну операцію АБО (диз'юнкція) над бітами операндів приймач і джерело. Результат записується на місце приймача:

```
OR AX, BX ;AX+BX - результат записати в AX
```

OR **10**01**0**0**0**1 – приймач
01011011 – джерело (маска)
1**1**011**0**1**1**1
↑ ↑ ↑
стан змінився

Отже, команду **OR** використовують для селективного встановлення 1 в приймачі.

```
; AX=91h, BX=5Bh
OR AX, BX; AX = DBh
```

Команда побітове ”XOR ”

XOR приймач, джерело — операція побітового виключаючого додавання.

Команда порозрядно виконує операцію виключаючого АБО над бітами операндів приймач і джерело. Результат записується на місце приймача.

Встановлює 1 в ті біти приймача, які відрізняються від бітів джерела.

Приклад:

```
XOR AX, TEST_PAR
JZ ALPHA
```

Якщо, хоча б в одному біті, не співпадають коди, то результат команд **XOR**=1 і **ZF**=0, отже, команда **JZ** не буде виконана. Вона буде виконуватися лише тоді, коли коди повністю співпадають. Команда **XOR** змінює приймач.

XOR 1**1**01**0**01**1** – приймач
01001001 – джерело (маска)
1**0**011**0**1**0**
↑ ↑ ↑
стан змінився

```
; AX=93h, BX=49h
XOR AX, BX; AX = 9Ah
```

11	07	06	04	02	00
OF	SF	ZF	AF	PF	CF
0	r	r	?	r	0

Команда побитовое ” NOT ”

NOT операнд — операція логічного заперечення.

Результат записується на місце операнда. Команда **NOT** змінює всі біти на протилежні:

Приклад:

```
flag    db      0ffh ; значення прапорця - true
...
cycl:
...
        CMP     flag,0
        JE      m1
...
m1:     NOT     flag    ;встановити прапорець true
```

Виконання команди ***НЕ впливає на прапорці***

```
; AX=10011010b
NOT AX; AX = 01100101b
```

Команда побітове ” TEST ”

TEST приймач, джерело — операція “перевірити” (способом логічного множення).

Команда виконує порозрядно ***операцію І*** над бітами операндів приймача і джерела. Стан операндів залишається сталим, змінюються тільки прапорці ***ZF, SF і PF***, що дає можливість аналізувати стан окремих бітів операндів без зміни їх стану.

Якщо, хоча б одна пара бітів дорівнює 1, то результат команди дорівнює 1, отже, **ZF** = 0.

Наприклад:

```
TEST   al,01h
JNZ    m1      ;перехід, якщо нульовий біт al дорівнює 1
```

```
TEST 01101101 – приймач
     01100100 – джерело (маска)
     01100100
       ↑↑  ↑
   Перевірені біти
```

11	07	06	02	00
OF	SF	ZF	PF	CF
0	r	r	r	0

2.3.2. Команди зміщення
(SHL, SHR, SAL, SAR, SHLD, SHRD, ROL, ROR, RCL, RCR)

До цієї групи належить 10 команд.

1. **6** зміщують операнд (**SHL, SHR, SAL, SAR, SHLD, SHRD**)
2. **4** крутять чи циклічно зміщують операнду (**ROL, ROR, RCL, RCR**).

SHL, SHR, SAL, SAR, SHLD, SHRD

Алгоритм:

1. Черговий зміщуваний біт встановлює прапорець **CF;**
2. Біт, що вводитья з іншого кінця, дорівнює 0;
3. При зміщенні чергового біта він переходить в прапорець **CF**, при цьому значення попереднього зміщеного біта губиться!

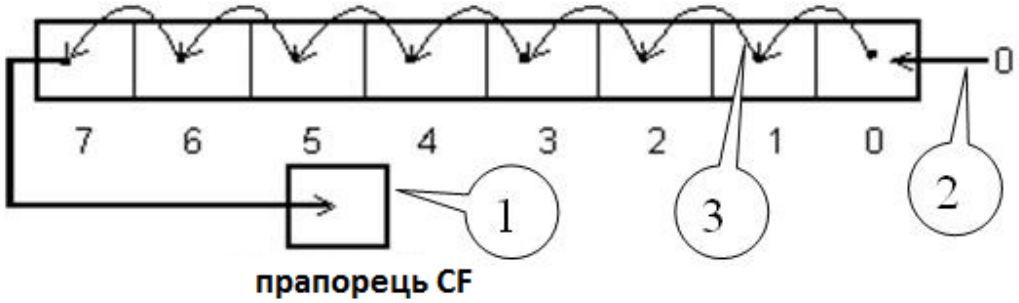
SHL операнд, лічильник зміщень (Shift Logical Left) – логічне зміщення ***ліворуч***.

Вміст операнди зміщується ліворуч на кількість бітів, що визначається значенням лічильника зміщень. Праворуч (в позицію меншого біту) записуються нулі;

SHL AX, CL -- помножити AX без знаку на 2^{CL}

SHR операнд, лічильник зміщень (Shift Logical Right) — логічне зміщення ***праворуч***.

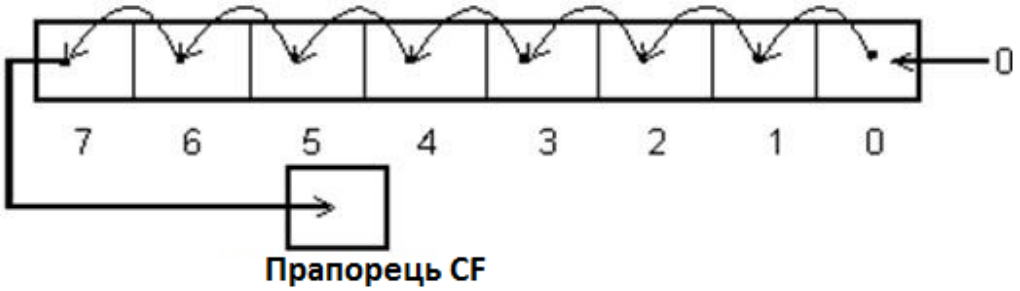
Вміст операндів зміщується праворуч на кількість бітів, яка визначається значенням лічильника зміщень. Ліворуч (в позицію більшого знакового біта) записуються нулі. На рис. показано принцип роботи цих команд.



SHR AX, CL - поділити AX без знаку на 2^{CL}

SAL операнд, лічильник зміщень (Shift Arithmetic Left) —арифметичне зміщення ***ліворуч***.

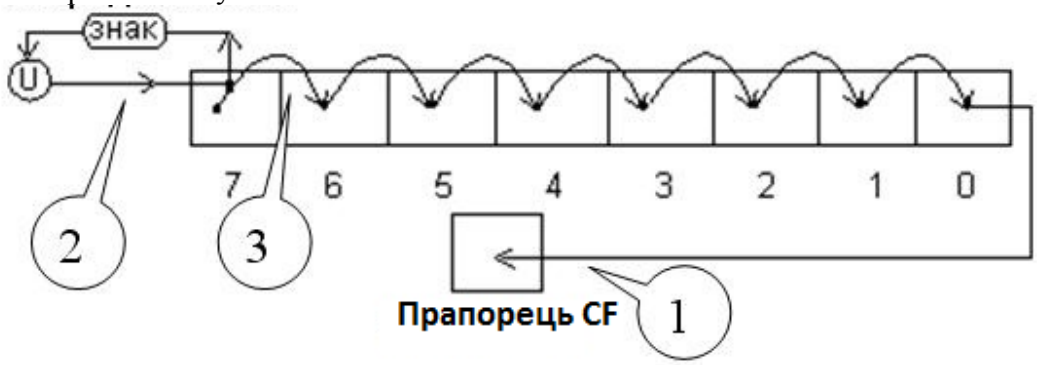
Вміст операнду зміщується ліворуч на кількість бітів, яка визначається лічильником зміщень. Праворуч (в позицію меншого біта) записуються нулі. Команда **SAL** не зберігає знак, але встановлює прапорець ***CF*** у випадку зміни знаку черговим зміщуваним бітом. У іншому випадку команда **SAL** повністю аналогічна команді **SHL**;



SAL AX, CL - помножити AX зі знаком на 2^{CL}

SAR операнд, лічильник зміщень (Shift Arithmetic Right) — арифметичне зміщення ***праворуч***.

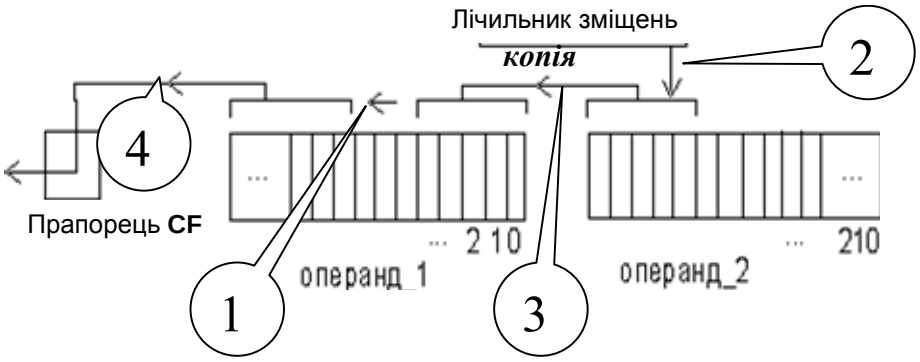
Вміст операнду зміщується праворуч на кількість бітів, яка визначається значенням лічильника зміщень. Команда **SAR** зберігає знак, встановлюючи його після зміщення кожного чергового біту.



SAR AX, CL - поділити AX зі знаком на 2^{CL}.

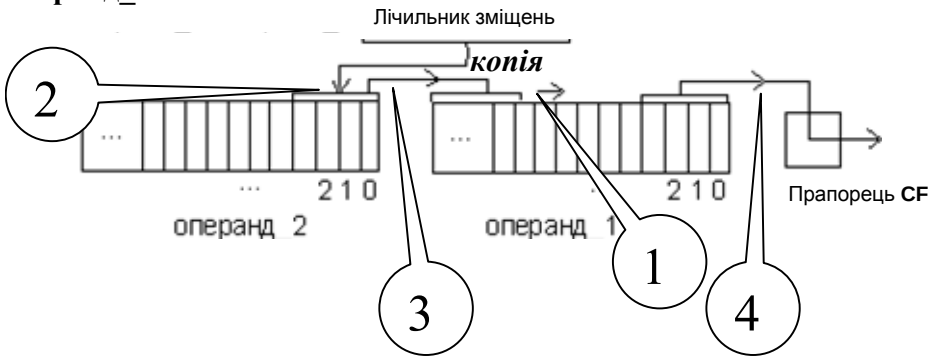
SHLD операнд_1, операнд_2, лічильник зміщень — зміщення ***ліворуч*** подвійної точності.

Команда **SHLD** виконує заміну шляхом зміщення бітів **операнд_1** ліворуч, заповнюючи його біти праворуч бітами, які витісняються із **операнд_2** згідно схемі на рис. Кількість зміщуваних бітів визначається лічильником зміщень. Значення лежить в діапазоні 0...31. Це значення може задаватися операндом або зберігатися в регістрі **CL**. Значення **операнд_2** НЕ змінюється.



SHRD операнд_1,операнд_2, лічильник зміщень — зміщення ***праворуч*** подвійної точності.

Команда **SHLD** виконує заміну шляхом зміщення бітів **операнд_1** праворуч, заповнюючи його біти ліворуч бітами, які витісняються із **операнд_2** згідно схемі на рис. Кількість зміщуваних бітів визначається лічильником зміщень. Значення лежить в діапазоні 0...31. Це значення може задаватися операндом або зберігатися в регістрі **CL**. Значення **операнд_2** НЕ змінюється.



2.3.3. Команди циклічного зміщення

1. Команди простого циклічного зміщення

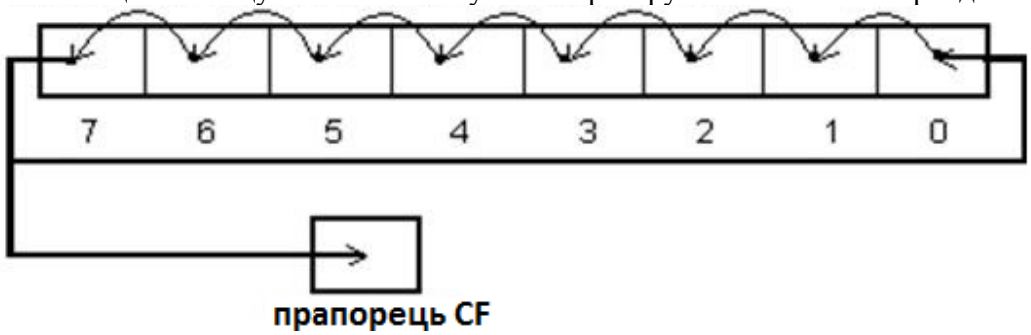
ROL, ROR

Алгоритм:

1. Зміщення всіх бітів операнду ***ліворуч*** на один розряд, при цьому більший передається в операнд праворуч і стає значенням меншого біту операнду;
2. Одночасно бітів, що висувався стає значенням прапорця переносу **CF**;
3. Вказані вище дії повторюються кількість раз, яке дорівнює значенню лічильника зміщень;

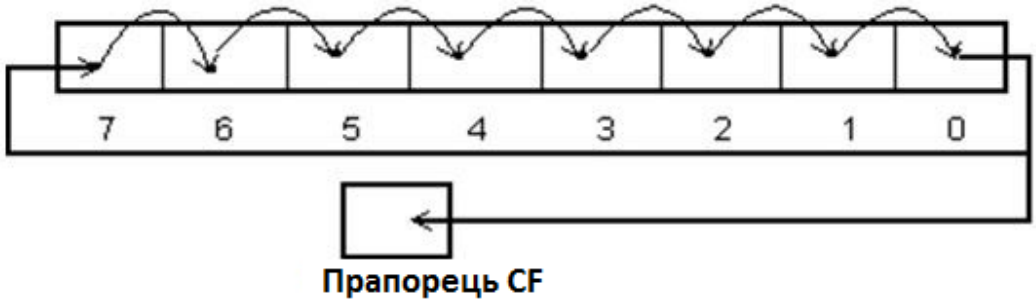
ROL операнд, лічильник зміщень (Rotate Left) — циклічне зміщення ліворуч.

Вміст операнда зміщується ліворуч на кількість бітів, що визначається операндом лічильник зміщень. Зміщувані біти записуються праворуч в той самий операнд.



ROR операнд, лічильник зміщень (Rotate Right) — циклічне зміщення праворуч.

Вміст операндів зміщується праворуч на кількість бітів, що визначається операндом лічильник зміщень. Зміщувані біти записуються ліворуч в той самий операнд.



2. Команди циклічного зміщення через прапорець перенесення CF

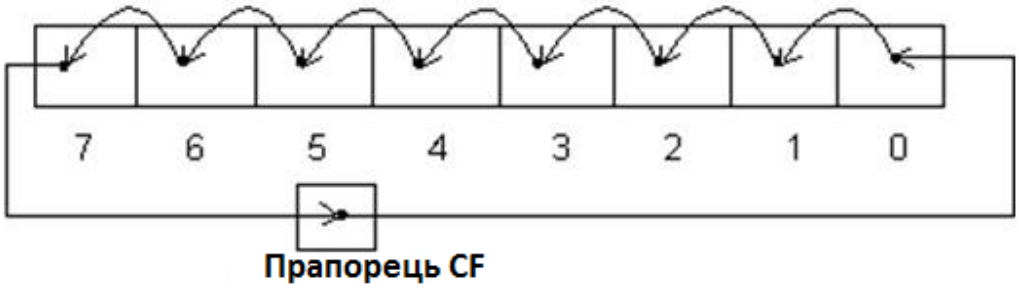
RCL, RCR

Алгоритм:

1. Зміщення всіх бітів операнда на один розряд, при цьому більший біт операнди стає значенням прапорця перенесення **CF**;
2. Одночасно старе значення прапорця перенесення **CF** зміщується в операнд праворуч і набуває значення меншого біту операнду;
3. Вказані вище дії повторюються ту кількість разів, що вказана в лічильнику зміщень.

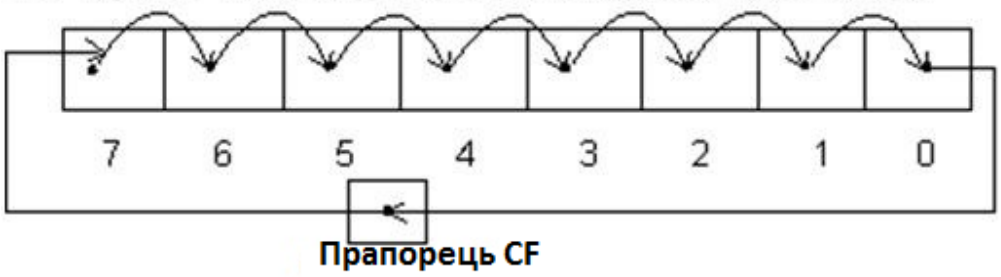
RCL операнд, лічильник зміщень (Rotate through Carry Left) — циклічне зміщення ліворуч через перенесення.

Вміст операнду зміщується ліворуч на кількість бітів, що визначається операндом лічильника зміщень. Зміщувані біти почергово стають значеннями прапорця перенесення **CF**.



RCR операнд, лічильник зміщень (Rotate through Carry Right) — циклічне зміщення праворуч через перенесення.

Вміст операнду зміщується праворуч на кількість бітів, що визначається операндом лічильника зміщень. Зміщувані біти почергово стають значеннями прапорця перенесення **CF**.



Ці команди виконуються швидше ніж **MUL** та **DIV**

Лекція 9

2.4. Команди передачі керування

Команди передачі керування поділяються на:

1. ***безумовні*** - в даному місці необхідно передати управління не тій команді, яка йде наступною, а інший, яка знаходиться на деякому віддаленні від поточної команди;

2. ***умовні*** - рішення про те, яка команда буде виконуватися наступною, приймається на основі аналізу деяких умов або даних;

3. ***управління циклом***;

4. ***виклик процедур***.

2.4.1. Команда безумовного переходу JMP

JMP [модифікатор] адрес_переходу - безумовний перехід без збереження інформації про місце повернення.

Модифікатор може приймати такі значення:

• ***near ptr*** - прямий перехід на мітку всередині поточного сегмента коду. Модифікується тільки регістр **ip /eip** (залежно від типу сегмента коду use16 / 32) на основі зазначеного в команді адреси (мітки) або вирази;

```
JMP pt ;Перехід на мітку pt в межах поточного сегмента
JMP near ptr pt ; те ж саме
```

• ***far ptr*** - прямий перехід на мітку в іншому сегменті коду. Адреса переходу задається у вигляді безпосереднього операнда або адреси (мітки) і складається з 16-бітного селектора cs і 16/32-бітного зсуву **ip / eip**;

```
JMP far ptr far pt ;Перехід на мітку farpt в іншому програмному сегменті
JMP farpt ;Перехід на мітку farpt в іншому
;програмному сегменті,  якщо farpt
;оголошена дальньою міткою директивою farpt label far
```

• ***word ptr*** - непрямий перехід на мітку всередині поточного сегмента коду. Модифікується (значенням зміщення в пам'яті за вказаною в команді адресою, або з регістра) тільки **ip /eip**. Розмір зміщення 16 або 32 біт;

```
;В полях даних:
addr dw pt ;Комірка з адресою місця переходу
;В програмному сегменті:
jmpDS:addr ;Перехід в місце pt
JMP word pt raddr ; Те ж саме
```

• ***dword ptr*** - непрямий перехід на мітку в іншому сегменті коду. Модифікуються (значенням з пам'яті, **з регістра не можна**) обидва регістра, **cs** і **ip /eip**. Перше слово / подвійне слово цієї адреси представляє зміщення і завантажується в **ip / eip**; друге / третє слово завантажується в **cs**.

```
; В полях даних:
addr dd pt ;Поле з адресою в два слова
;В програмному сегменті:
jmp DS:addr ;Перехід в місце pt
JMP dword ptr addr ; те ж саме
```

Крім прямого переходу, можна здійснити непрямий перехід. При цьому в команді **JMP** відзначається ім'я слова, де зберігається адреса переходу, або ім'я регістра, де записується адреса переходу.

Приклади непрямих ближніх переходів

Приклад 1

```
mov BX,offset pt ;BX=адреса точки переходу
jmp BX ;Перехід в місце pt
```

Приклад 2

```
; В полях даних:
addr dw pt ;Комірка з адресою місця переходу
;В програмному сегменті:
mov DI,offset addr ;B1 – адреса комірки з адресою
;місця переходу
jmp [DI] ;Перехід в місце pt
```

Приклад 3

```
;В полях даних:
tb ldw ptl ;Комірка з адресою 1
dw pt2 ;Комірка з адресою 2
dw pt3 ;Комірка з адресою 3
;В програмному сегменті:
mov BX,offset bl ;BX – адреса таблиці адресів переходів
mov SI, 4 ;SI - зміщення до адреси pt3
call [BX][SI] ;Перехід в точку pt3
```

2.4.2. Команди умовного переходу

Всього передбачається **17 команд** умовного переходу. Але деякі команди мають кілька синонімів (мнемонік). Тому іноді говорять про **31 команду**.

Ці команди дозволяють перевірити:

1. відношення між операндами зі знаком ("більше - менше");
2. відношення між операндами без знака ("вище - нижче");
3. стану арифметичних прапорців **zf**, **sf**, **cf, of**, **pf** (але НЕ **af**).

Команди умовного переходу мають однаковий синтаксис:

JCC мітка_переходу

Як видно, мнемокод всіх команд починається з "**J**" - від слова jump (стрибок), **CC** - визначає конкретну умову, проаналізованупевними командами.

Що стосується операнда **мітка_переходу**, то ця мітка може знаходитися тільки в межах поточного сегмента коду, міжсегментна передача управління в умовних переходах НЕ допускається.

Команда порівняння **CMP** має цікавий принцип роботи. Він абсолютно такий же, як і у команди віднімання.

CMP операнд_1, операнд_2 (compare) - порівнює два операнда, віднімаючи **операнд_2** від **операнд_1**, не змінюючи їх і по результату встановлює прапори.

Таблиця

Значення абревіатур в назві команди JCC

Мнемонічне значення	Англійська	Українська	Тип операндов
E чи e	equal	Рівно	Будь які
N чи n	not	НІ	Будь які
G чи g	greater	Більше	Числа зі знаком
L чи l	less	Менше	Числа зі знаком
A чи a	above	Вишче, в понятті “більше”	Числа без знака
B чи b	below	Нижче, в понятті “менше”	Числа без знака

Команди реакції на арифметичні порівняння зі знаком

Для таких порівнянь використовуються слова “менше” (less) та “більше” (greater). Зрозуміло, що можно перевірити 6 умов:

Типи операндів	Мнемокод команди умовного переходу	Критерій умовного переходу	Значення прапорців для здійснення переходу
Будь які	JE	операнд_1 = операнд_2	zf = 1
Будь які	JNE	операнд_1<>операнд_2	zf = 0
Зі знаком	JL / JNGE	операнд_1 < операнд_2	Sf <> of
Зі знаком	JLE / JNG	операнд_1 <= операнд_2	sf <> of або zf = 1
Зі знаком	JG / JNLE	операнд_1 > операнд_2	sf = of та zf = 0
Зі знаком	JGE / JNL	операнд_1 => операнд_2	sf = of

Наприклад:

```
CMP AX, BX
JL LABEL2; перехід, якщоAX<BX
```

Команди реакції на арифметичні порівняння без знаку

Типи операндів	Мнемокод команди умовного переходу	Критерій умовного переходу	Значення прапорців для здійснення переходу
Будь які	JE	операнд_1 = операнд_2	zf = 1
Будь які	JNE	операнд_1<>операнд_2	zf = 0
Зі знаком	JB / JNAE	операнд_1 < операнд_2	cf = 1
Зі знаком	JBE / JNA	операнд_1 <= операнд_2	cf = 1 або zf=1
Зі знаком	JA / JNBE	операнд_1 > операнд_2	cf = 0 та zf = 0
Зі знаком	JAE / JNB	операнд_1 => операнд_2	cf = 0

Для таких порівнянь використовуються слова "вище" (above) і "нижче" (below), після порівняння (**CMP**) адрес:

Команди перевірки окремих прапорців і регістрів

Мнемонічне позначення деяких команд умовного переходу відображає назва прапорця, з яким вони працюють, і має наступну структуру: першим йде символ "**J**" (jump, перехід), другим - або позначення прапорця, або символ заперечення "**N**", після якого стоїть назва прапорця.

Назва прапорця	№ біта в eflags/flag	Команда умовного переходу	Значення прапорця для здійснення переходу
Прапорець переносу cf	1	JC	cf = 1
Прапорець парності pf	2	JP	pf = 1
Прапорець нуля zf	6	JZ	zf = 1
Прапорець знака sf	7	JS	sf = 1
Прапорець переповнення of	11	JO	of = 1
Прапорець переносу cf	1	JNC	cf = 0
Прапорець парностіpf	2	JNP	pf = 0
Прапорець нуля zf	6	JNZ	zf = 0
Прапорець знака sf	7	JNS	sf = 0
Прапорець переполвення of	11	JNO	of = 0

Типи операндів	Команда умовного переходу	Критерій умовного перехода	Зн.регістрів для переходу
Будь які	JCXZ	Jump if **cx** is Zero	cx = 0

Будь які	JECXZ	Jump Equal **ecx** Zero	ecx = 0

Наприклад:

```
CMP AX, BX
JE cycl
JCXZ   m1  ;обійти цикл, якщосх=0
cycl:;деякий цикл
LOOP  cycl
m1:    ...
```

2.4.3. Команди управління циклами

Оскільки в програмах часто доводиться реалізовувати циклічні дії, то для цього існує кілька команд.

LOOP мітка - зменшує регістр **CX** на 1 і здійснює перехід на певну мітку, якщо зміст **CX** ≠ **0**, якщо **CX** = 0, здійснюється перехід на наступну за **LOOP** команду.

Ця команда використовує регістр **CX** як лічильник циклу.

Отже, **LOOP** повинна завершувати цикл. На початку циклу в **CX** необхідно занести кількість повторень. Тобто, фрагмент може мати такий вигляд:

```
MOV CX, LOOP_COUNT
BEGIN_LOOP:
<тіло циклу>
LOOP BEGIN_LOOP
```

Здійсни , що в тілі циклу регістр **CX** додатково змінювати **НЕ МОЖНА**.

Оскільки при реалізації циклів перевіряється вміст регістра **CX**, то для цього існує спеціальна команда умовного переходу **JCXZ** (if CX is zero), коли зміст **CX** дорівнює нулю.

Що буде, коли в регістр **CX** потрапить не якесь число, а нуль? В цьому випадку буде 65536 повторень. Щоб цього не сталося, доцільно використовувати спеціальну перевірку **CX** на нуль:

```
MOV CX, LOOP_COUNT
JCXZ END_LOOP
BEGIN LOOP: <тіло>
LOOP BEGIN_LOOP
END_LOOP: <продовження>
```

Існує д и.

LOOPE / LOOPZ мітка - команда продовжується до тих пір, поки **CX≠0** або поки **ZF=1**.

Таким чином, цю команду зручно застосовувати для пошуку першого ненульового елемента в масиві без перебору всього масиву до кінця.

LOOPNE / LOOPNZ мітка - команда продовжує цикл поки CX ≠ 0 або поки **ZF=0**.

Ця команда зручна для пошуку першого нульового елемента масиву.

Наприклад:

```
; початкова адресамасива у BX
; кінцева адресамасива у DI
SUB DI, BX; різниця адрес
INC DI; кількість байт в масиві
MOV CX, DI; лічильник цикла
DEC BX
NEXT: INC BX; до наступногоелемента
      CMP BYTE PTR [BX], 0; порівняти з нулем
       LOOPE NEXT;якщо 0 -продовжуємо
       JNZ NZ_FOUND; знайденоненульовийелемент
NZ_FOUND:
```

У наве ризначення - локально **EAR**, **FAR**. Вживається я для того, щоб адресу

Відзначимо, що команди циклу, як і інші команди умовного переходу, діють в межах -128, 127 байт. Якщо обсяг команд більший, то тоді потрібно інакше організувати фрагмент з використанням команди **JMP**.

```
model small
.stack 100h
.data
masdb 1,0,9,8,0,7,8,0,2,0
        db 1,0,9,8,0,7,8,0,2,0
        db 1,0,9,8,0,7,8,?,2,0
        db 1,0,9,8,0,7,6,?,3,0
        db 1,0,9,8,0,7,8,0,2,0
. code
start:
movax,@data
mov ds.ax
xor ax, ax
leabx, mas
movcx, 5
cycl_l:
pushcx
xorsi , si
mov cx, 10
cycl_2:
cmpbyte ptr [bx+si],0
jneno_zero
movbyte ptr [bx+si],0ffh
no_zero:
incsi
loop cycl_2
pop cx
addbx, 10
loop cycl_1
exit:
movax,4c00h
int 21h
endstart
```

```
TITLE   CALLPROC (EXE) Викликпроцедур
0000                    STACKSG SEGMENT PARA STACK 'Stack'
0000    20 [ ???? ]             DW      32 DUP(?)
0040                    STACKG  ENDS

0000                    CODESG  SEGMENT PARA 'Code'
0000BEGIN   PROC    FAR
ASSUME  CS:CODESG,SS:STACKSG
0000  1E                        PUSH    DS
0001  2B C0                     SUB     AX,AX
0003  50                        PUSH    AX
0004  E8 0008 R                 CALL    B10     ;Викликати B10
;       ...
0007  CB      RET               ;Завершитипрограму
0008                    BEGIN   ENDP
;-------------------------------------
0008                    B10     PROC
0008  E8000C R                CALL    C10     ;Викликати C10
;       ...
000B  C3                        RET             ;Повернутися у програму,
000CB10     ENDP  ; що викликається
                        ;-----------------------------------------------
000CC10     PROC
                        ;       ...
000C  C3                        RET             ;Повернутися в
прогаму,000D                    C10     ENDP            ; що
викликається
                        ;-----------------------------------------------
000D                    CODESG  ENDS
                                END     BEGIN
```

Лекція 10

2.5. Команди обробки рядків (ланцюжків) символів

Під *рядком* символів тут розуміється послідовність байт, а *ланцюжок* - це більш загальна назва для випадків, коли елементи послідовності мають розмір більше байта - слово або подвійне слово.

Таким чином, ланцюгові команди дозволяють проводити дії над блоками пам'яті, що являють собою послідовності елементів наступного розміру:

• 8 біт - байт;
• 16 біт - слово;
• 32 біта - подвійне слово.

Вміст цих блоків для мікропроцесора не має ніякого значення. Це можуть бути символи, числа і все що завгодно. Головне, щоб розмірність елементів збігалася з однією з перерахованих і ці елементи знаходилися у сусідніх комірках пам'яті.

Передбачено 7 основних операцій (примітивів):

1. пересилання елементів;
2. порівняння елементів;
3. сканування елементів;
4. завантаження елементів;
5. зберігання елементів;
6. отримання елементів ланцюжка з порту вводу-виводу;
7. виведення елементів ланцюжка у порт вводу-виводу

2.5.1. Пересилання

Команда **MOVS**:
MOVS адреса_приймача, адреса_джерела (MOVe String) - переслати ланцюжок;
MOVSB (MOVe String Byte) - переслати ланцюжок байтів;
MOVSW (MOVe String Word) - переслати ланцюжок слів;
MOVSD (MOVe String Double word) - переслати ланцюжок подвійних слів.

Команда **MOVS** копіює байт, слово або подвійне слово з ланцюжка, що адресується операндом **адреса_джерела**, в ланцюжок, що адресується операндом **адреса_приймача**.

При трансляції в залежності від типу операндів транслятор перетворює її в одну з трьох машинних команд: **MOVSB**, **MOVSW** або **MOVSD**.

Якщо перед командою написати префікс **REP**, то однією командою можна переслати до 64 Кбайт даних (якщо розмір адреси в сегменті 16 біт - **use16**) або до 4 Гбайт даних (якщо розмір адреси в сегменті 32 біти - **use32**).

Алгоритм:

1. Встановити значення прапора **DF** залежно від того, в якому напрямку будуть оброблятися елементи ланцюжка - в напрямку зростання **(DF = 0)** або зменшення адрес **((DF = 1));**
2. Завантажити покажчики на адреси ланцюжків в пам'яті в пари регістрів **DS: (E) SI і ES: (E) DI;**
3. Завантажити в регістр **ECX / CX** кількість елементів, що підлягають обробці;
4. Видати команду **MOVS** з префіксом **REP**.

Отже, перед виконанням команд обробки рядків потрібно відповідно встановити стан прапора **DF** за допомогою команди:

STD (set direction flag) - DF = 1 - в напрямку зменшення адрес.

CLD (clear direction flag) (DF = 0) - в напрямку зростання адрес.

Префікси повторення

Замість команд **LOOP** тут вживаються спеціальні префікси повторення. Кількість повторень попередньо записується в регістр **CX**.

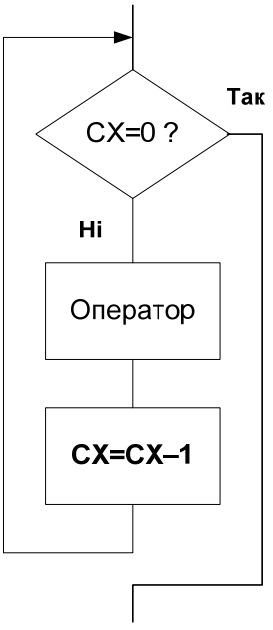
Наприклад:

```
MOV CX, 50
REP MOVS DEST, SOURCE
```

REP використовується перед строковими командами і їх короткими еквівалентами: **movs**, **stos**, **ins**, **outs**

Алгоритм:

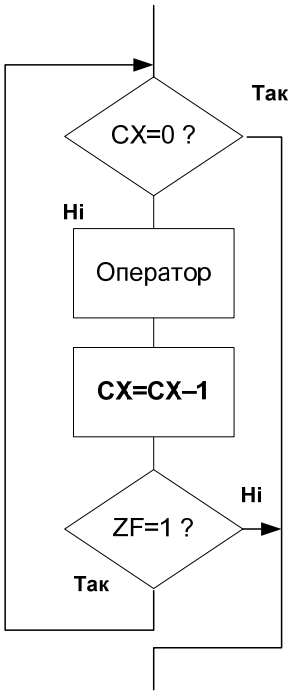
1. аналіз вмісту **CX**;
2. якщо **CX <> 0**, то виконати строковую команду, наступну за даними префіксом і перейти до кроку 4;
3. якщо **CX = 0**, то передати управління команді, наступної за даною строковою командою (вийти з циклу по **REP**);
4. зменшити значення **CX = CX-1** і повернутися до кроку 1.



REPE і **REPZ** використовуються перед наступними ланцюжковими командами і їх короткими еквівалентами: **CMPS**, **SCAS**.

Алгоритм:

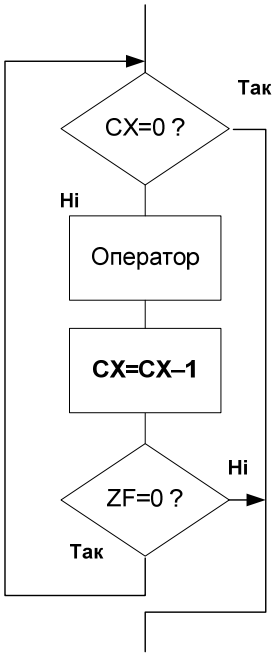
1. аналіз вмісту **CX**;
2. якщо **CX <> 0**, то виконати ланцюжкову команду, наступну за даними префіксом, і перейти до кроку 4;
3. якщо **CX = 0** або **ZF = 0**, то передати управління команді, наступної за даною ланцюговою командою і перейти до кроку 6;
4. зменшити значення **CX = CX -1;**
5. якщо **ZF = 1** повернутися до кроку 1;
6. вийти з циклу по **REP**.



REPNE і **REPNZ** також мають один код операції і можуть бути використаними перед еквівалентами: **cmps**, **scas**.

Алгоритм:

1. аналіз вмісту **CX;**
2. якщо **CX <> 0**, то виконати ланцюжкову команду, наступну за даними префіксом, і перейти до кроку 4;
3. якщо **CX = 0** або **ZF = 0**, то передати управління команді, наступної за даною ланцюжковою командою і перейти до кроку 6;
4. зменшити значення **CX = CX -1**
5. якщо **ZF = 0** повернутися до кроку 1
6. вийти з циклу по **rep**.



Фрагмент Пересилання рядків командою **MOVS** буде мати вигляд:

```
MASM
MODEL small
STACK 256
.data
source db 'Тестуємо рядок', '$'
; рядок-джерело
dest db 19 DUP ('); рядок-приймач
.code
assume ds: @ data, es:data
main:; точка входу в програму
mov ax, @ data; завантаження сегментних регістрів
mov ds, ax; настройка регістрів DS і ES; на адресу сегмента даних
mov es, ax
cld; скидання прапора DF - обробка рядка від початку до кінця
lea si, source; завантаження в si зміщення рядка-джерела
lea di, dest; завантаження в DS зміщення рядка-приймача
mov cx, 20; для префікса rep - лічильник повторень (довжина рядка)
rep movs dest, source; пересилання рядка
lea dx, dest
mov ah, 09h; висновок на екран рядка-приймача
int 21h
exit:
mov ax, 4c00h; теж саме, що і RET
int 21h
end main
```

2.5.2. Порівняння рядків

CMPS адреса_приймача, адреса_джерела - порівнює байти або слова
CMPSB
CMPSW
CMPSD

Адреса_джерела знаходиться в сегменті даних і адресується за допомогою регістрів **DS і SI. Адреса_приймача** - в додатковому сегменті і адресується за допомогою регістрів **ES і DI**.

Виконується як віднімання, але на відміну від **CMP**, **від джерела віднімається приймач**. Це необхідно враховувати для наступних команд умовного переходу.

Алгоритм:

1. Завантажити адресу джерела в декілька регістрів **DS: ESI / SI**;
2 Завантажити адресу призначення в декілька регістрів **ES: EDI / DI**;
3 Виконати віднімання елементів (джерело - приймач);
4 Залежно від стану прапора **DF** змінити значення регістрів **ESI / SI і EDI / DI:**
5 Якщо **DF** = 0, то збільшити вміст цих регістрів на довжину елементу послідовності;
6 Якщо **DF** = 1, то зменшити вміст цих регістрів на довжину елементу послідовності;
7 Залежно від результату віднімання встановити прапорці:
8 Якщо чергові елементи ланцюжків не рівні, то **CF** = 1, **ZF** = 0;
9 Якщо чергові елементи ланцюжків або ланцюжка загалом рівні,то **CF** = 0, **ZF** = 1;
10. При наявності префікса виконати обумовлені ним дії (див. Команди **REPE** / **REPNE**).

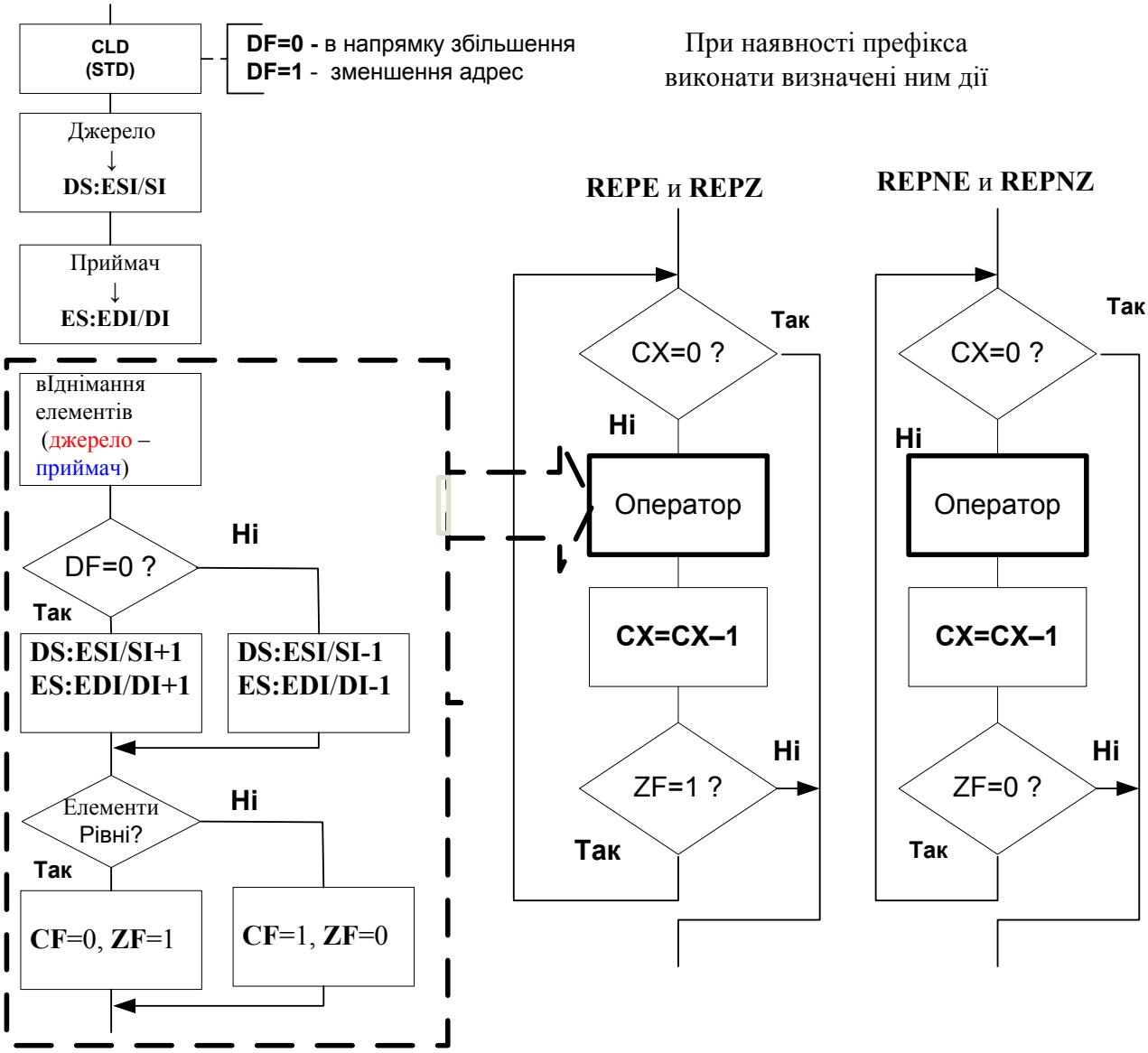
Частіше з даною командою для повторення вживається префікс **REPE** (до першого відмінності) або **REPNE** (до першого співпадання).

```
CLD
MOV CX,50
REPE CMPS DEST, SOURCE; до першої відмінності
```

Вихід з циклу відбувається з двох причин - масив переглянутий повністю, або є відмінності. Тому **CMPS** необхідно реагувати на відповідну причину, використовуючи команди умовного переходу. Отже, після наведеного фрагмента потрібно поставити перехід на мітку:

```
JNE FOUND; відмінність знайдено
……
FOUND:      ; так, продовжити опрацювання
```

Команда **CMPS** перетворюється транслятором: на **CMPSB** або на **CMPSW**, аналогічно команді **MOVS**.



Наприклад, можна визначити кількість різних пар в рядках:

```
MOV AX, 0; кількість різних пар
CLD
LEA SI, SOURCE
LEA DI, ES:DEST
MOV CX, N
COMP: REPE CMPSB
JE FIN; перехід на FIN, якщо рядки рівні
INC AX; наступна пара, що не співпадає
CMP CX, 0; чи залишились символи?
JNE COMP; продовжити, якщо рядки не порожні
FIN:
```

2.5.3 Сканування рядків

SCAS адреса_приймача -дозволяє визначити значення (байт чи слово) в рядку-приймачеві, який знаходиться в додатковому сегменті, початок якого фіксує регістр **ES**.

SCASB

SCASW

SCASD

При цьому задане значение повинно знаходитись в регістрі **AX** (під час пошуку слова) або **AL** (під час пошуку символу).

Тобто, це фактично команди порівняння з вмістом акумулятора.

Наприклад, знайти першу крапку '.' і замінити на '*':

```
CLD
LEA DI, ES: STRING
MOV AL, '.'
MOV CX, 50
REPNE SCAS STRING ; или SCAS B
JNE FIN ; крапки немає
MOV BYTE PTR ES:[DI -1], '*'
FIN:
```

Це буде пошук у рядку STRING першої крапки. При виході з циклу в регістрі DI буде адреса наступного за крапкою '.' байта.

2.5.4. Завантаження рядка

Після пошуку слова або символу за допомогою сканування з ним щось потрібно зробити.

LODS адреса_джерела - команда завантаження рядка

LODSB

LODSW

LODSD

Пересилається операнд **рядок_джерело**, який адресований регістром **SI** з сегменту даних в регістр **AL** (при пересиланні байту) або в регістр **AX** (пересилання слова), а потім змінює регістр **SI** так, щоб він показував на наступний елемент рядка. Тобто, регістр **SI** збільшується на 1 або 2, якщо **DF** = 0 і зменшується, якщо **DF** = 1.

Отже, команди **LODS** еквівалентні двом:

```
MOV AL, [SI]
INC SI
```

Наприклад:

```
        CLD
        LEA DI, ES DEST ; зміщення адреси DEST
        LEA SI, SOURCE ; зміщення адреси SOURCE
        MOV CX, 100
        REPE CMPSB; перевірка скасуванная
                JCXZ MATCH; чи є розбіжності?
                DEC SI; скорегувати SI
                LODS SOURCE; завантажити інший елемент в
регістр AL
        MATCH; розбіжностей немає
```

2.5.5 Команда зберігання рядка

STOS адреса_приймача - протилежна до **LODS,** пересилає байт з **AL** або слово з **AX** в **рядок-приймач**, який знаходиться в додатковому сегменті й має адресу регістра **DI**. Після цього змінює регістр **DI** так,що він показує на наступний елемент рядку.

STOSB

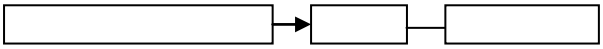
STOSW

STOSD

Командою зручно заповнювати рядки певними значеннями. Наприклад, в рядку MY_STRING здійснюється пошук серед 100 слів першого ненульового елемента. Якщо такий елемент знайдено, то наступні 5 слів, що йдуть після нього, заповнюються нулями.

```
        CLD
        LEA DI, ES: MY_STRING ; адреса рядка
        MOV AX, 0;
        MOV CX, 100  ; лічильник пошуку
        REPNE SCASW ; сканування рядка
        JCXZ NOT_SCAN  ; чи знайдено ненульове слово?
        MOV CX, 5  ; так!
        REP STOS MY_STRING
        NOT SCAN
```

Отже, ці дві команди для зручності, якщо знайшли щось, то:



2.5.6 Отримання елементів ланцюжка з порту введення-виведення

INS адреса_приймача, номер_порту - ввести елементи з порта введення-виведення в ланцюжок.

INSB

INSW

INSD

Ця команда вводить елементи з порту, номер якого знаходиться в регістрі **dx**, в елемент ланцюжка, адреса якого визначається операндом **адреса_приймача**.

Наприклад:

Введемо 10 байт з порту 5000h в область пам'яті pole.

```
.data
pole    db      10 dup (' ')
.code
...
        push    ds
        pop     es      ;налаштування es на ds
        mov     dx, 5000h
        lea     di, pole
        mov     cx,10
rep     insb
...
```

2.5.7 Виведення елементу ланцюжка в порт введення-виведення

OUTS номер_порту, адреса_джерела (Output String) — вивести елементи з ланцюжка в порт введення-виведення .

OUTSB

OUTSW

OUTSD

Ця команда виводить елемент ланцюжка в порт, номер якого знаходиться в регістрі **DX**. Адреса елемента ланцюжка визначається операндою **адреса_джерела**. Незважаючи на те, що ланцюжок, з якого виводиться елемент, адресується вказівкою цього операнда, значення адреси повинно бути явно сформовано в парі регістрів **DS: ESI / SI**. Як приклад розглянемо фрагмент програми, яка виводить послідовність символів в порт введення-виведення, відповідного принтеру **(номер 378 (lpt1)).**

```
.data
str_pech        db      'Текст для друку'
.code
...
        mov     dx,378h
        lea     di,str_pech
        mov     cx,16
rep     outsb
...
```

Команди завантаження адресних пар в регістри

Використання рядкових команд потребує певної кількості інсталяційних команд. Їх можна скоротити за допомогою двох команд, які встановлюють пари регістрів **DS: SI і ES: SI** на опрацювання рядків.

Це команди: **LDS, LES, LFS, LGS, LSS**

LDS приймач, джерело - отримання повного покажчика у вигляді сегментної складової і зміщення.

Приймач - ім'я регістра, а **джерело** - пам'ять - адреса подвійного слова пам'яті, яке задає абсолютну адресу. Команда записує в регістр зміщення **ofs**, а в регістр **DS** - номер цього сегмента.

Алгоритм:

Алгоритм роботи команди залежить від діючого режиму адресації (**use16** або **use32**):

• якщо **use16**, то завантажити перші **два** байти з комірки пам'яті джерело в 16-розрядний регістр, вказаний операндом **приймач**. Наступні два байта в області **джерело** повинні містити сегментну складову деякої адреси; Вони завантажуються в регістр **DS / ES / FS / GS / SS** - залежно від команди;

• якщо **use32**, то завантажити перші **чотири** байти з комірки пам'яті джерело в 32-розрядний регістр, вказаний операндом **приймач**. Наступні два байта в області джерело повинні містити сегментну складову, або селектор, деякої адреси; Вони завантажуються в регістр **DS / ES / FS / GS / SS**.

Приклад:

```
ADR DD ALPHA; [ADR] → ofs [ADR + 2]:SEG
LDS SI, ADR;  SI - ofs, DS → SEG
```

Друга команда:

LES приймач, джерело - (Load pointer using ES) аналогічна до першої, тільки номер сегмента завантажується в регістр **ES**.

Наприклад:

```
DATA1 SEGMENT
        S1 DB 200DUP(?)
        AS DD S2
DATA1 ENDS
DATA2 SEGMENT
        S2 DB 200 DUP(?)
DATA2 ENDS
; скопіювати масив S1 в S2
CODE SEGMENT
         ASSUME CS: CODE, DS:DATA1
; нехай в цей момент  DS = DATA1
        CLD
         LEA SI, S1 ; DS:SI = початок S1
         LES DI, AS ; ES:DI = початок S2
        MOV CX, 200
        REP MOVSB; скопіювати S1 в S2
```

2.6 Команди управління мікропроцесором

1. Команди управління прапорцями;
2. Команди зовнішньої синхронізації;
3. Команди холостого ходу.

2.6.1. Команди управління прапорцями

Дозволяють впливати на стан трьох прапорців регістра прапорців: **CF** (carry flag) - перенесення, **DF** (direction) - спрямування та переривання **IF** (interupt F).

Структура команд дуже проста: встановити біт в одиницю (Set), скинути в 0 (Clear). Відповідні дві літери - початкові для команди. Третя літера визначає, який саме біт потрібно опрацювати. Тому, маємо 6 команд:

STC	**STD**	**STI**
CLC	**CLD**	**CLI**

І сьома команда – змінити значення біту переносу на протилежне:

CMC (CoMpliment Carr Flag).

Команди управління бітом Carry вживаються перед виконанням команд циклічного зміщення з переносом **RCR (RLC)** - операція циклічного зміщення операнда вправо (вліво) через прапорець перенесення **CF**.

Команди управління бітом напрямку, як бачимо, використовуються перед командами опрацювання рядків і задають напрямок модифікації індексних регістрів **DI** і **SI** (**DF** = 0 - в сторону збільшення, **DF** = 1 - в сторону зменшення).

Команди управління бітом **IF** використовуються, наприклад, при обробці замаскованих переривань. Якщо під час роботи програми жодне замасковане переривання не дозволяється, то потрібно встановити **IF** в 0. Нагадаємо, що при цьому незамасковані переривання дозволяються.

2.6.2 Команди зовнішньої синхронізації

Використовуються для синхронізації роботи МП із зовнішніми подіями.

Команда **HLT** (halt - зупинити) - зупиняє роботу мікропроцесора, він переходить на холостий хід і не виконує ніяких команд. З цього стану його можна вивести сигналами під час входу **RESET, NMI, INTR**. Якщо для відновлення роботи мікропроцесора використовується переривання, то збережене значення пари **CS: EIP / IP** вказує на команду, наступну за **hlt**. В мікропроцесорі не передбачено спеціальних засобів для подібного перемикання. Скидання мікропроцесора можна ініціювати, якщо вивести байт із значенням **0feh** в порт клавіатури **64h**. Після цього мікропроцесор переходить в реальний режим і управління отримує програма **BIOS**, яка аналізує байт відключення в **CMOS**-пам'яті за адресою **0fh**. Для нас інтерес становлять два значення цього байта - **5h і 0ah**:

5h - скидання мікропроцесора ініціює ініціалізацію програмованого контролера переривань на значення базового вектора **08h**. Далі управління передається за адресою, яка знаходиться в комірці області даних **BIOS 0040: 0067;**

0ah - скидання мікропроцесора ініціює безпосередньо передачу управління за адресою в комірці області даних **BIOS 0040: 0067** (тобто без перепрограмування контроллера переривань).

Таким чином, якщо ви не використовуєте переривання, то досить встановити байт **0fh** в **CMOS**-пам'яті в **0ah**. Попередньо, звичайно, ви повинні ініціалізувати комірку області даних **BIOS 0040: 0067** значенням адреси, за якою необхідно передати управління після скидання. Для програмування **CMOS**-пам'яті використовуються номера портів **070h і 071h**. Спочатку в порт **070h** заноситься необхідний номер комірки **CMOS**-пам'яті, а потім в порт **071h** - нове значення цієї комірки.

Якщо переривання заблоковані під час зупинки, ПЕОМ повністю "завмирає". У цій ситуації єдина можливість запустити ЕОМ заново - вимкнути живлення і включити його знову. Однак, якщо переривання були дозволені в момент зупинки мікропроцесора, вони продовжують сприйматися і керування буде передаватись обробнику переривань. Після виконання команди **IRET** в обробнику програма продовжує виконання з комірки, наступної за командою **HLT**. Команду **HLT** можна використовувати в мультизадачних системах, щоб завершити поточну активну задачу, але це не завжди найкращий спосіб такого завершення. Розробники персональної ПЕОМ використовують команду зупинки тільки тоді, коли виникає катастрофічна помилка обладнання та подальша робота безглузда.

Наприклад:

```
;працюємо в реальному режимі, готуємося до переходу в захищений режим:
      push   es
      mov    ax,40h
      mov    es,ax
      mov    word ptr es:[67h],offset ret_real
;ret_real — мітка в програмі з якої повинно початись виконання програми після скидання
      mov    es:[69h],cs
             mov    al,0fh  ;будемо звертатись до комірки 0fh в CMOS
      out    70h,al
      jmp    $+2     ;трохи затримаємось, щоб апаратура опрацювала
;скидання без перепрограмування контролера
      mov    al,0ah
      out    71h,al
;переходимо в захищений режим встановленням  біта 0 cr0 в 1
;працюємо в захищеному режимі, готуємося перейти назад, в реальний режим
      mov    al,01fch
      out    64h,al  ;скидання мікропроцесора  hlt
;зупинка до фізичного завершення процесу скидання
      ret_real:      ...     ;мітка, на яку буде передано управління після скидання
```

Після цього МП починає виконувати команду, наступну за **HLT**.

Команда **WAIT** (wait - чекати) - переводить МП на холостий хід, але при цьому, через кожні 5 тактів перевіряється активність вхідної лінії **TEST**. Якщо цей висновок активний під час виконання **WAIT**, то зупинка **НЕ** відбувається.

Команда **ECS** (escape - втеча) - забезпечує передачу команди МП 88/86 зовнішнім процесорам, наприклад, арифметичному співпроцесору 8087. Сам же МП 88 нічого не робить, тільки читає якісь дані і відкидає.

Команди **WAIT** і **ESC** використовуються для роботи з співпроцесором 8087.

Префікс **LOCK** - це командний префікс, подібно **REP**-префіксу, він призначений для роботи в мультипроцесорних системах, коли декілька процесорів працюють з однією і тією ж ділянкою пам'яті. Префікс **LOCK** примушує захопити лінії управління і тим самим отримати виключне право доступу до пам'яті під час час обробки команди з префіксом.

```
MOV AL,1

LOCK XCHG AL,FLAG_BYTE
```

У загальній області FLAG_BYTE встановлено 1, якщо в ній працює інший МП, і 0, якщо ніякої МП не працює. Коли там є 1, то МП очікуватиме, поки область не звільниться.

Порожня команда **NOP** - нічого не виконується 2 такта, реалізується як **XCHG AX, AX.**

Цю команду можна використовувати:

- в програмах реального часу, коли потрібно витримати точну тривалість фрагментів;
- при налагодженні програми, коли окремі фрагменти можуть змінюватися.

Наприклад:

```
JZ ONE
....
ONE: MOV AX, SUM[BX]
```

Краще:

```
JZ ONE
.....
ONE: NOP
MOV AX, SUM[BX]
```

Лекція 11

Розділ 3. Процедури та макрокоманди

3.1 Процедури та особивості їх використання

Процедурою (підпрограмою) називається фрагмент команди, до якого можна перейти і з якого можна повернутися на місце, з якого здійснювався перехід.

Перехід до процедури називається викликом, а перехід назад - поверненням. Повернення здійснюється до команди, наступної після виклику.

Ці функції виконують дві команди:

CALL (call a procedure) і

RET (return from procedure – повернутися із процедури).

Команда CALL має формат:

CALL ім'я

Де ім'я – це ім'я процедури, яка викликається.

Сама ж процедура, має вигляд:

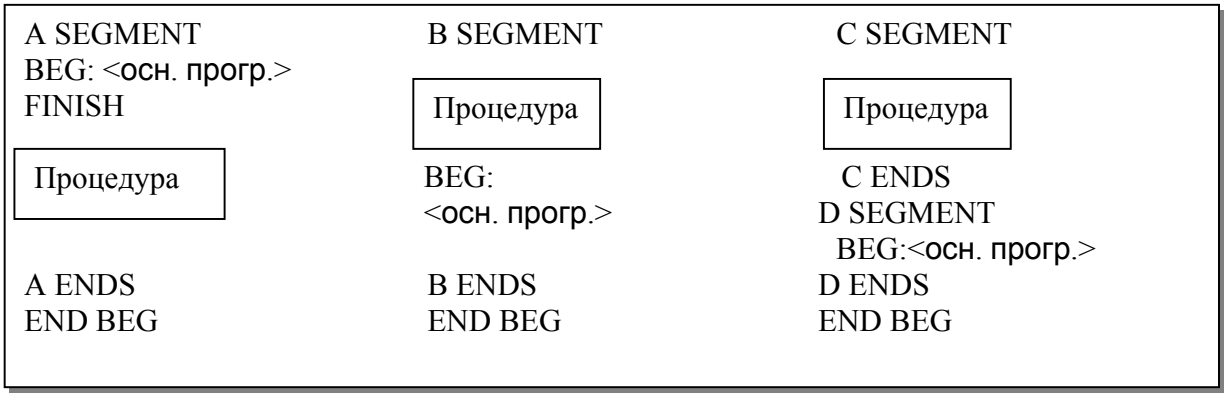
```
NAME PROC
……
RET
NAME ENDP
```

Ім'я процедури вважається міткою, яка належить першій команді процедури. Тому ім'я процедури можна відмічати в командах переходу, яке означає перехід до першої команди процедури.

На відмінну від мов високого рівня, імена в процедурах (змінні, мітки) не локалізуються всередині, відповідно повинні бути унікальними для відповідного сегменту.

Де розміщувати процедуру? Де завгодно. Але так, щоб процедура не виконувалася якщо до неї не звертаються. Тому можна рекомендувати 3 варіанти:

1. В одному сегменті після основної команди;
2. В одному сегменті перед точкою входу;
3. В окремому сегменті, тоді можна в кінці основної прогами поставити команду FINISH.



Якщо процедура не передбачена для використання в інших сегментах, то вона має атрибут дистанції **FAR**. Головна процедура також повинна мати атрибут **FAR**. За замовчуванням, процедура має атрибут **NEAR**.

При виконанні команди **CALL *NAME*** спочатку треба занести адресу повернення в стек, а потім зробити перехід на ім'я ***NAME***.

Для **NEAR**-процедури в стек достатньо занести лише адресу зміщення.

Отже, в цьому випадку команда **CALL NAME** *еквівалентна*

```
PUSH IP
JMP NAME
```

Для **FAR**-процедури адреса повернення являється абсолютною і складається з двох слів **CS:IP**. Тому команда **CALL *NAME*** буде еквівалентна трьом командам:

```
PUSH CS
PUSH IP
JMP NAME
```

Іншими словами словами:

1. Спочатку в стек заноситься адреса початку сегменту коду, яка міститься в регістрі **CS**
2. Потім заноситься вміст регістру покажчика команд **IP** - зміщення.

Команда **RET** повертає управління до точки виклику процедури. Еквівалентна двом командам:

```
POP IP
POP CS
```

Якщо процедура має атрибут **NEAR** – виконується тільки перша команда. Тобто, особливості виконання команди **RET** визначають заголовком процедури – наявністю атрибуту **NEAR** чи **FAR**.

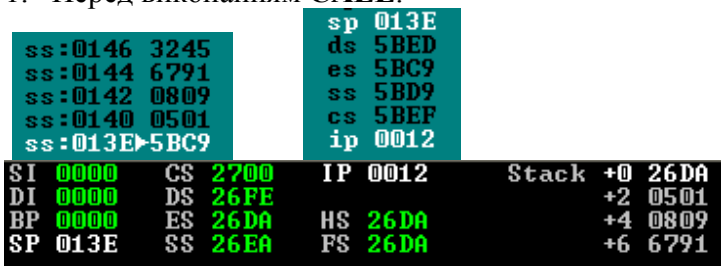
Приклад: перший фрагмент команди буде транслюватися за такими адресами:

```
0012  E8 0007        CALL MY_PROC; виклик процедури
0015  8B C3          MOV AX, BX; повернутися сюди із процедури
0017  B8 4C00        MOV AX,4c00h
001A  CD 21          INT 21h
001C                 MAIN ENDP
                     ; опис процедури
001C                 MY_PROC PROC; початок процедури
001C  B1 0A           MOV CL, 10; перша команда процедури
001E  C3              RET; повернутися до точки виклику
001F                  MY_PROC ENDP
```

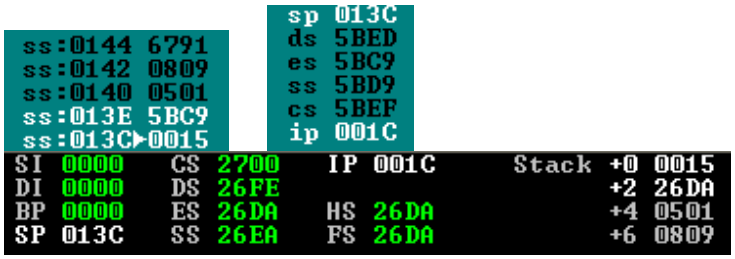
За замовчуванням MY_PROC має атрибут **NEAR**.

Вплив процедури на стек:

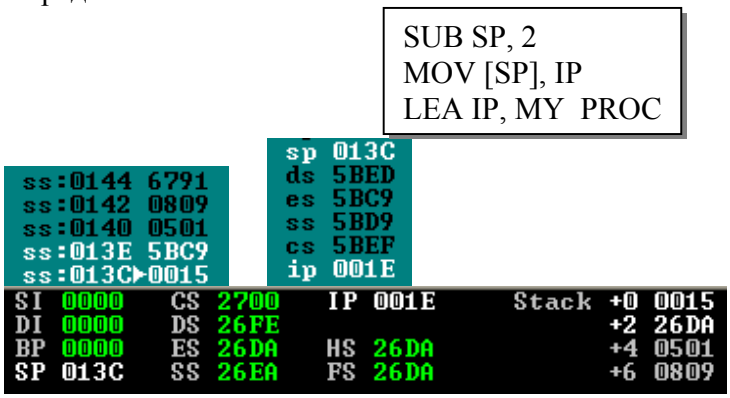
1. Перед виконанням **CALL**:



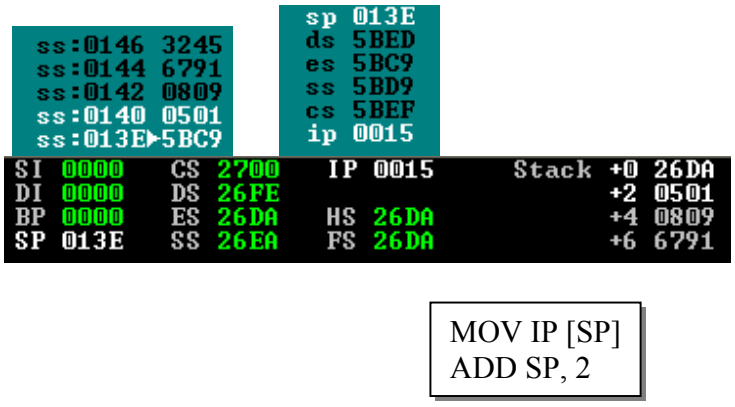
Після виконання CALL:



Перед виконанням **RET**:



Після виконання **RET**:



Продовжиться виконання, починаючи з команди NEXT:

Оскільки, значення IP змінилося, то почнеться виконання процедури починаючи з команди MOV CL, 10 і до команди **RET**.

Процедура може розміщуватися в тому ж сегменті, звідки здійснювалося звернення до неї чи в іншому сегменті. В першому випадку атрибутом процедури може бути **NEAR** і звернення також типу **NEAR**. В другому випадку атрибутом процедури буде **FAR** і звернення також буде **FAR**.

Наприклад:

```
SEGX SEGMENT
.........
SUBT PROC FAR
.........
RET
SUBT ENDP
........
CALL FAR PTR SUBT
...........
SEGX ENDS
SEGY SEGMENT
.........
CALL FAR PTR SUBT
...........
SEGY ENDS
```

Тобто, в зверненні до процедури можуть бути атрибути **FAR PTR** чи **NEAR PTR**. В даному випадку процедура визначена її викликом, тобто, здійснюється виклик “назад”. Тому якщо не написати **FAR PTR**, то асемблер все-одно правильно реалізує і виклик, і повернення.

В даному випадку їх можна залишити для ясності, але, коли процедура реалізується після виклику, тобто, здійснюється виклик “вперед”, то ці атрибути обов’язкові.

Однак таких дій замало. Для виклику процедури із сегменту SEGY в ньому обов’язково потрібно пояснити, що ім’я SUBT являється зовнішнім. Тобто, вставити директиву EXTRN SUBT: FAR.

В сегменті SEGX потрібно пояснити, що до імені SUBT може бути доступ із зовні, тобто, потрібно встановити директиву PUBLIC SUBT.

Відмітимо, що ім’я SUBT знаходиться в тому ж сегменті, що і директива PUBLIC. Тому в ній відмічати тип імені не потрібно (транслятор розбереться). В директиві EXTRN відмітити тип потрібно. За аналогією з командою **JMP** в процедурі можливі звернення:

1. прямі;
2. непрямі;
3. близькі;
4. далекі.

Не дозволяються короткі виклики **SHORT**, тому що рахується, що процедура не розміщується поряд з командою виклику. Можливі такі варіанти виклику:

```
CALL P; близький перехід
CALL FAR PTR Q; далекий перехід
CALL Q; близький перехід
CALL NEA; близький непрямий виклик
CALL FA; далекий непрямий виклик
LEA BX,Q
CALL [BX]; близький виклик процедури Q
CALL DWORD PTR [BX]; далекий непрямий виклик
Приклад:
NEAR DW P
FA DD Q
P PROC
P1: ....
P ENDP
Q PROC FAR
Q1: .....
Q ENDP
```

Лекція 12

3.2. Передача параметрів в процедуру

Як бачимо, аналогії з передачею параметрів в процедуру як в мовах високого рівня немає. Отже, немає списку параметрів. Тому програміст сам повинен подбати про таку передачу як в процедуру, так і з неї.

Можна використовувати декілька варіантів:

1. Через регістри;

2. Через стек;

3. Через загальні області пам'яті.

3.2.1. Передача параметрів через регістри

Основна программа розміщує параметри в певних регістрах, а процедура їх звідти бере. Процедура записує результати в регістри, а программа їх потім зчитує.

Наприклад:

```
визначити c = max(a,b); a запишемо в AX, b - в BX, а результат c - в AX.
основнапрограма
MOV AX, A
MOV BX, B
CALLMAX
MOVC, AX; скопіювати AX в змінну C.
……
MAX PROC FAR
CMP AX, BX  ;BX-AX
JGE MAX1  ; якщо BX ≥ AX
MOVAX, BX  ;скопіювати BX у AX
MAX1: RET
MAX ENDP
```

Захист регістрів в процедурах

Регістрів в МП мало, тому процедура і програма використовують одні й ті ж регістри. Щоб процедура не змінювала змісту регістрів в головній програмі, потрібно їх захистити. Тобто, переслати вміст регістрів в стек, а потім обновити в зворотньому порядку. Це рекомендується робити навіть тоді, коли програма і процедура використовують різні регістри. З часом основна программа може змінитися, і тоді процедура буде "заважати".

Для полегшення таких дій в МП, починаючи з 80186, впроваджені команди **PUSHA** і **POPA**, які записують і зчитують з стека вміст відразу 8 регістрів**: AX, BX, CX, DX, DI, SI, SP, BP**.

Передача параметрів складних типів

Крім параметрів значень, в процедуру можна передавати адреси. Коли параметром є складний тип - структура або масив, то на машинному рівні передають не дані, а початкову адресу. Знаючи кількість елементів і початкову адресу або поле, можна отримати доступ до кожного елементу.

Наприклад:

```
для двох масивів чисел без знаку
X DB 100 DUP(?)
Y DB 50 DUP(?)
Обчислити   max (X[i])+max(Y[i]).
```

Ясно, що краще зробити процедуру визначення максимального елемента. У неї буде два параметри: початкова адреса і кількість елементів.

```
; процедура MAX: AL = макс. елемент
MAX PROC
      PUSH BX ;початкова адреса
      PUSH CX
MOV AL,0
MAX1: CMP [BX],AL
         JLE MAX2
         MOV AL[BX]
MAX2: INC BX; до наступного ел-та
        LOOP MAX1
         POP CX
         POP BX
RET
MAX ENDP
; Основная программа
LEA BX,X
MOV CX,100
CALL MAX
MOV DL,AL; защита AL
LEA BX,Y
MOV CX,50
CALL MAX
ADD DL,AL
```

Захист регістрів

Якщо немає гарантії, що процедура може змінити зміст регістрів, які повинні зберегтися після виходу з процедури такими, як безпосередньо перед викликом процедури, то їх потрібно на час виконання заслати в стек.

Наприклад:

```
всі регістри загального призначения:
PUSH AX
PUSH BX
PUSH CX
```

Перед виходом з процедури (**RET**)потрібно ці регістри оновити:

```
POP DX
POP CX
POP BX
POP AX
```

Це робимо в зворотному порядку. Це ж стосується індексних регістрів і молодшого байта регістра прапорців.

3.2.2. Передача параметрів через стек

Якщо в процедурі багато параметрів, то може не вистачити регістрів. Тому в цьому випадку можна використовувати стек. Перед викликом процедури записуємо параметри в стек в порядку, який визначає програміст.

При цьому застосовується своєрідна методика роботи зі стеком не за допомогою команд **push** і **pop**, а за допомогою команд **mov** з непрямою адресацією через регістр **BP**, який архітектурно призначений саме для адресації в стеку.

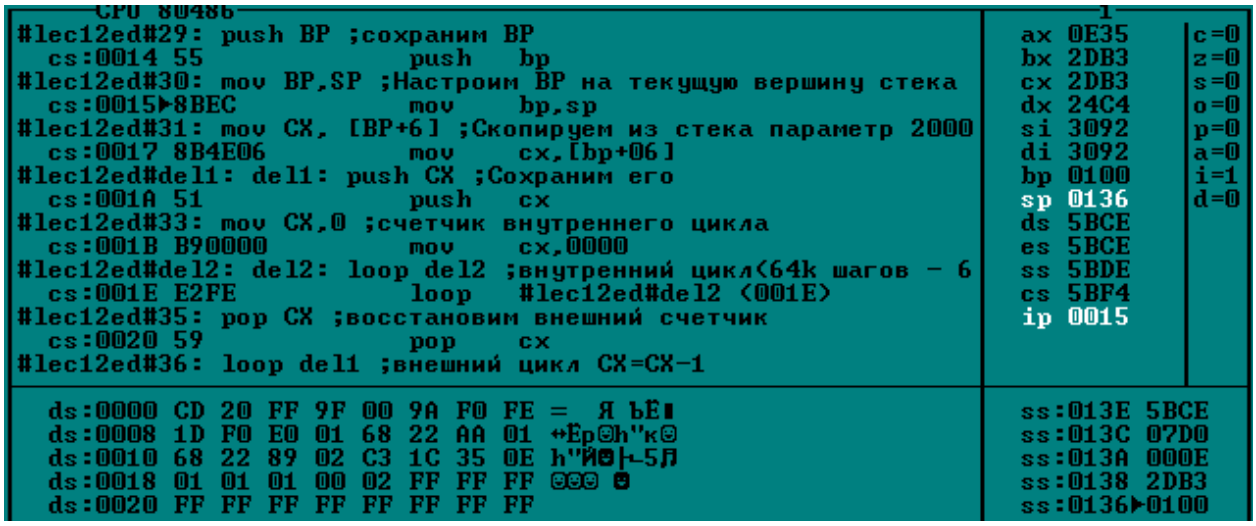
Приклад:

Потрібно, щоб параметр (умовна величина затримки) передавався в процедуру через стек. Виклик процедури delay виконується наступним чином:

```
0001  50 55 8BECC7 4602+  push 2000  ;Записуємо в стек значення 2000
000BE8 0005    calldelay          ;Викликаєм підпрограму delay
000E  B8 4C00      mov ax,4c00h
0011  CD 21                int 21h
0013                       MAINENDP
0013                       delayproc                   ;Процедура-підпрограма
0013  51                   pushCX                      ;Збережемо CX основної процедури
0014  55                   pushBP              ;ЗбережемоBP
0015  8BEC                 movBP,SP                    ;Налаштуємо BP       на поточну вершину стека
0017  8B 4E 06 movCX, [BP+6]              ;Скопіюємозі стека параметр
001A  51                   del1:  pushCX      ;Збережемо його
001BB9 0000    movCX,0  ;Лічильник внутрішнього цикла
001EE2 FE                  del2:  loopdel2     ;Внутрішній цикл(64Ккроків)
```

Команда call, передаючи управління процедурі, зберігає в стеку адресу повернення в основну програму. Процедура зберігає в стеку ще два 16-розрядних регістра. В результаті стек має наступний стан:

Після збереження в стеку початкового вмісту регістра **BP** (в основній процедурі нашого прикладу цей регістр не використовується, однак в загальному випадку це може бути і не так), в регістр **BP** копіюється вміст покажчика стека, після чого в **BP** є зміщення вершини стека. Далі командою mov в регістр **CX** заноситься вміст комірки стека, на 6 байтів нижче поточної вершини. У цьому місці стека якраз знаходиться параметр, який передається в процедуру параметрів, як це показано в лівому стовпчику рис.



Конкретну величину зміщення щодо вершини стека треба для кожної процедури визначати індивідуально, виходячи з того, скільки слів збережено нею в стеку до цього моменту. Нагадаємо, що при використанні непрямої адресації з регістром **BP** як базовий, за замовчуванням адресується стек, що в данному випадку і потрібно.

Виконавши покладене на неї завдання, процедура відновлює збережені раніше регістри і здійснює повернення в основнупрограму за допомогою команди ret, як аргумент якої вказується число байтів, займаних в стеку відправленими туди перед викликом процедури параметрами. У нашому випадку, коли параметр займає 2 байта. Якщо тут використовувати звичайну команду ret без аргументу, то після повернення в основну програму параметр залишиться в стеку, і його треба буде звідти витягувати (між іншим, не дуже зрозуміло, куди саме, адже всі регістри у нас можуть бути зайняті). Команда ж з аргументом, здійснивши повернення в процедуру, що викликається, збільшує вміст покажчика стека на значення її аргументу, тим самим здійснюючи логічне зняття параметра. Фізично цей параметр, як, втім, і всі інші дані, поміщені в стек, залишається в стеку і буде затертий при подальших зверненнях до стека.

Зрозуміло, в стек можна було помістити не один, а скільки завгодно параметрів. Тоді для їх читання треба було використовувати кілька команд mov із значеннями зсуву **BP + 6**, **BP + 8**, **BP + 0Ah** і т.д.

Розглянута методика може бути використана і при дальних викликах підпрограм, але в цьому випадку необхідно враховувати, що дальня команда call зберігає в стеку не одне, а два слова, що вплине на величину, яка розраховує зміщення щодо вершини стека.

3.2.3. Передача параметрів через загальні області пам'яті

Якщо процедура і виклик до неї розміщені в різних сегментах коду, то дані можна розмістити на самому початку і використовувати їх імена з директивами **PUBLIC** і **EXTRN**.

Директива **EXTRN** призначена для оголошення деякого імені зовнішнім по відношенню до даного модулю. Це ім'я в іншому модулі має бути оголошено в директиві **PUBLIC**.

Директива **PUBLIC** призначена для оголошення деякого імені, визначеного в цьому модулі і видимого в інших модулях.

Синтаксис цих директив наступний:

extrn ім'я: тип, ім'я: тип

public ім'я, ..., ім'я

Тут ім'я - ідентифікатор, визначений в іншому модулі. Як ідентифікатор можуть виступати:

• імена змінних, визначених директивами типу **DB**, **DW** і т. д.;

• імена процедур;

• імена констант, визначених операторами «=» і **EQU**.

Аргумент «тип» визначає тип ідентифікатора. Вказувати тип необхідно для того, щоб транслятор правильно сформував відповідну машинну команду. Дійсні адреси обчислюються на етапі редагування, коли будуть формуватись зовнішні посилання. Можливі значення типу визначаються допустимими типами об'єктів для цих директив:

• ім'я змінної - тип може приймати значення **DB**, **DW** і т. д.;

• ім'я процедури - тип може приймати значення near або far;

• ім'я константи.

```
;Модуль 1
masm
.model small
.stack 256
.data
.code
my_proc_l proc
my_proc_l endp
my_proc_2 proc
my_proc_2 endp
public my_proc_l ;оголошуємо процедуру my_proc_l видимої ззовні


start:
movax,@data
end start
```

Покажемо принцип використання директив **EXTRN** і **PUBLIC** на прикладі модулів 1 і 2.

```
;Модуль 2
masm
.model small
.stack 256
.data
. code
extrnmy_proc_l ;оголошуємо процедуру roy_proc_lззовні
```

Якщо в процедурі, описаному в одному сегменті є опис змінних, то в головній процедурі, де здійснюється виклик цієї процедури, можна використовувати й інші імена цих елементів даних.

Наприклад:

```
Додавання елементів масива:
DATA SEGMENT COMMON
ARY DW 100 DUP(?)
COUNT DW ?
SUM DW 0
DATA ENDS
```

Наприклад:

NUM замість ARY,
N замість COUNT A
TOTAL замість SUM.
Для цього цей сегмент потрібно визначити так:

```
DATA SEGMENT COMMON
NUM DW 100 DUP(?)
N DW ?
TOTAL DW 0
DATA ENDS
```

Під час редагування зв'язків сегменти DATA суміщаються і перші 100 слів (200 байт) DATA матимуть в різних сегментах однакові адреси, тобто адреси змінних ARYi NUM будуть однакові, та адреси COUNT і TOTAL теж.

Лекція 13

3.3 Макрокоманди

Для коротких фрагментів програм їх організація у вигляді підпрограм (процедур) може бути неефективною. Наприклад, додавання вмісту двох слів і результат записується в третє слово ,що можна реалізувати таким чином:

```
ADD_WORDS PROC
MOV AX, BX
ADD AX, CX
RET
ADD_WORDS ENDP
```

До цього додаються три команди передачі параметрів - дві до виклику процедури і одна після:

```
MOV BX, ARG1
MOV CX, ARG2
CALL ADD_WORDS
MOV SUM, AX
```

Як бачимо, на дві команди потрібно ще додаткові 5 команд, з яких виклик процедури реалізується як 2 команди **PUSH** і **JMP**. Отже, така реалізація неефективна. Краще просто записати 5 команд в певному місці.

В асемблері існує, для цього випадку, ще один засіб - макрокоманди. Макрокоманди дозволяють звертатися до групи команд як до однієї.

Для цього вони, попередньо, повинні бути записані у вигляді макровизначення.

Макровизначення складається з:

• Заголовка;

• Тіла;

• Заключного рядка.

Назва складається з імен, ключового слова MACRO і списку формальних параметрів.

Ім'я MACRO список параметрів

Тіло - це послідовність команд макроасемблера.

Список формальних параметрів може бути відсутнім. Якщо формальні параметри відмічені, то тіло макровизначення їх використовує.

Заключний рядок макроозначення має вигляд:

ENDM; без вказівки імені макровизначення.

Отже, наведений вище фрагмент може бути реалізований у вигляді макровизначення:

```
ADD_WORDS MACRO ARG1, ARG2, SUM
MOV AX, ARG1
ADD AX, ARG2
MOV SUM, AX
ENDM
```

Макровизначення розміщуються в будь-якому місці програми, але до виклику макрокоманди.

Звернення до макрокоманд має вигляд:

Ім'я список фактичних параметрів

У нашому випадку:

```
ADD_WORDS TERM1, TERM2, COST
```

Ясно, що таких звернень в програмі може бути декілька. Як виконується макрокоманда? На місце виклику макрокоманди вставляється тіло макровизначення з заміною формалізованих параметрів фактичними.

Тобто, замість одного рядка звернення буде розміщено 3 рядки:

```
MOV AX, TERM1
ADD AX, TERM2
MOV COST, AX
```

В результаті цього транслятор створює макророзширення в кожному місці, де був виклик макрокоманди. Тобто, програма ніби розширюється, збільшується. Після цього макровизначення не потрібно і може бути вилучено. Далі скомпонована програма виконується послідовно без переходів до інших частин. Тобто, швидкість виконання буде високою, але використання макрокоманд призводить до збільшення обсягу програми.

Таким чином бачимо, що виклик макрокоманди це **НЕ** вказівка микропроцессору, а директива транслятору.

Макрокоманди динамічні: за рахунок зміни параметрів можна змінювати і об'єкти, і самі дії. Для процедур можна змінювати лише об'єкти.

Макрокоманди виконуються швидше процедур, немає потреби в переходах і поверненнях. Але використання макрокоманд збільшує обсяг пам'яті: в тілі програми макророзширення дублюються стільки раз, скільки були викликані. Процедура ж в пам'яті записується один раз.

Макровизначення можна записати в бібліотеку і використовувати при розробці нових програм.

3.3.1. Псевдооператори макроасемблера

Розділяють на 4 групи:

1. Загального призначення;
2. Повторення;
3. Умовні;
4. Управління лістингом.

3.3.1.1. Псевдооператори загального призначення

Директива **LOCAL**

Нехай є макровизначення, де якесь слово зменшується до 0, чим реалізується певна затримка.

```
DELAY_1 MACRO COUNT
LOCAL NEXT
.....
PUSH CX
MOV CX, COUNT
NEXT: LOOP NEXT
POP CX
ENDM
```

Якщо до цієї макрокоманді звертаються в програмі кілька разів, то в кожному макророзширенні з'явиться мітка NEXT, що неприпустимо тому, що кожна мітка повинна унікальною.

Для того, щоб у кожному розширенні створювалися унікальні мітки потрібно застосувати директиву **LOCAL**.

Після цього перший раз буде створена мітка ??0000, другий - ??0001 третій - ??0002 і т.д. Директива **LOCAL** розміщується одразу за заголовком макророзширення.

3.3.1.2. Псевдооператори повторення

Призначені для повторення кількох команд, які починаються заголовком і закінчуються словом **ENDM**.

Існує 4 види таких псевдооператорів:

1. **WHILE;**
2. **REPT;**
3. **IRP;**
4. **IRPC**

Директиви **WHILE** и **REPT** застосовують для повторення певної кількості разів деякої послідовності рядків.

WHILE константний_вираз
послідовність_рядків
ENDM

При використанні директиви **WHILE** макрогенератор транслятора повторюватиме **послідовність_рядків** доти, доки значення **константного_виразу** не буде дорівнювати нулю. Це значення обчислюється щоразу перед черговою ітерацією циклу повторення (тобто, значення **константного_виразу** повинно змінюватись всередині **послідовність_рядків** в процесі макрогенерації).

REPT константний_вираз
послідовність_рядків
ENDM

Директива **REPT** подібна директиві **WHILE** повторює **послідовність_рядків** стільки разів, скільки це визначено значенням **константного_виразу** і автоматично зменшує на одиницю значення **константного_виразу** після кожної ітерації.

Наприклад:

```
;Використання директив повторення
;prg_13_3.asm
def_sto_1 macro id_table,ln:=<5>
;макрос резервування пам’яті розміру len.
;Використовується WHILE
 id_table label byte
 len=ln
 while len
 db 0
 len=len-1
 endm
def_sto_2 macro id_table,len
; макрос резервування пам’яті розміру len.
 id_table label byte
 rept len
 db 0
 endm
data segment para public 'data'
def_sto_1 tab_1, 10
def_sto_2 tab_2, 10
data ends
end
```

```
    21   0000                    data segment para public 'data'
    22                            def_sto_1    tab_1, 10
1   23   0000                     tab_1 label byte
2   24   0000  00                 db 0
2   25   0001  00                 db 0
2   26   0002  00                 db 0
2   27   0003  00                 db 0
2   28   0004  00                 db 0
2   29   0005  00                 db 0
2   30   0006  00                 db 0
2   31   0007  00                 db 0
2   32   0008  00                 db 0
2   33   0009  00                 db 0
    34                            def_sto_2    tab_2, 10
1   35   000A                     tab_2 label byte
2   36   000A  00                 db 0
2   37   000B  00                 db 0
2   38   000C  00                 db 0
2   39   000D  00                 db 0
2   40   000E  00                 db 0
2   41   000F  00                 db 0
2   42   0010  00                 db 0
2   43   0011  00                 db 0
2   44   0012  00                 db 0
2   45   0013  00                 db 0
    46                            data ends
```

IRP формальний_аргумент, <рядок_символів_1, ..., рядок_символів_N>
Послідовність_рядків
ENDM

Дія даної директиви полягає в тому, що вона повторює **послідовність_рядків N** раз, тобто, стільки разів, скільки **рядків_символів** укладено в кутові дужки в другому операнді директиви **IRP**. Повторення **послідовності_рядків** супроводжується заміною в ній **формального_аргумента** рядком символів з другого операнди.

Так, при першій генерації **послідовності_рядків формальний_аргумент** в них замінюється на **рядок_символів_1.**

Якщо є **рядок_символів_2**, то це призводить до генерації другої копії **послідовності_рядків**, в якій **формальний_аргумент** замінюється на **рядок_символів_2**. Ці дії тривають **до рядка_символів_N** включно.

Наприклад:

```
IRP ini,<1,2,3,4,5>
 db ini
endm
```

Макрогенератором буде згенеровано наступні макророзширення:

```
db 1
db 2
db 3
db 4
```

IRPC формальний_аргумент, рядок_символів
послідовність рядків
ENDM

Дія даної директиви подібно до **IRP**, але відрізняється тим, що вона на кожній черговій ітерації замінює **формальний_аргумент** черговим символом зі **рядка_символів**.

Зрозуміло, що кількість повторень **послідовності_рядків** визначатиметься кількістю символів в **рядку_символів.**

Наприклад:

```
irpc char,HELLO
    db char
endm
```

В процесі макрогенерації ця директива розгорнеться в наступну послідовність рядків:

```
DB 'H'
DB 'E'
DB 'L'
DB 'L'
DB 'O'
```

```
DB H
DB E
DB L
DB L
DB O
```

Потрібно відзначити, що псевдооператор повторення дещо відрізняються від команди **LOOP** і префікса **REP**. **LOOP** виконується подібно до процедури, тобто управління передається на початок циклу. Директиви повторення виконуються подібно макрокомандам: дублюється фрагмент відповідне число раз.

3.3.1.3. Умовні псевдооператори

Макроасемблер підтримує кілька умовних директив, які корисні всередині макровизначень. Кожна директива **IF** повинна мати її відповідну директиву **ENDIF** для завершення умовного блоку і можливо всередині **ELSE**.

Тобто структура умовного блоку:

IFxxx логічний_вираз_чи_аргументи
фрагмент_програми_1
ELSE
фрагмент_програми_2
ENDIF

Усього є **10** типів умовних директив компіляції. Їх логічно попарно об'єднати в чотири групи:

1. Директиви **IF** і **IFE** - по результату обчислення логічного виразу;
2. Директиви **IFDEF** і **IFNDEF** - по факту визначення символічного імені;
3. Директиви **IFB** та **IFNB** - по факту визначення фактичного аргументу при виклику макрокоманди;
4. Директиви **IFIDN, IFIDNI, IFDIF** і **IFDIFI** - по результату порівняння рядків символів.

IF(E) логічний_вираз
фрагмент_програми_1
ELSE
фрагмент_програми_2
ENDIF

Обробка цих директив макроасемблером полягає в обчисленні **логічного_виразу** і включенні в об'єктний модуль **фрагмент_програми_1** або **фрагмент_програми_2** залежно від того, в якій директиві **IF** або **IFE** цей вираз зустрівся:

• якщо в директиві **IF** логічний вираз істинний, то в об'єктний модуль вміщується **фрагмент_програми_1.**

• якщо логічний вираз помилковий, то при наявності директиви **ELSE** в об'єктний код вміщується **фрагмент_програми_2**. Якщо ж директиви **ELSE** немає то вся частина програми між директивами **IF** і **ENDIF** ігнорується і в об'єктний модуль нічого не включається. До речі поняття істинності і хибності значення **логічного_виразу** вельми умовно. Хибним воно вважатиметься якщо його значення дорівнює нулю, а істинним - при будь-якому значенні відмінному від нуля.

• директива **IFE** аналогічно директиві **IF** аналізує значення **логічного_виразу**. Але тепер для включення **фрагмент_програми_1** в об'єктний модуль необхыдно щоб **логічний_вираз** мав значення "неправда".

Директиви **IF** і **IFE** дуже зручно використовувати при необхідності зміни тексту програми в залежності від деяких умов.

```
<1>...
<2>debug equ 1
<3>...
<4>.code
<5>...
<6>if debug
<7> ;будь які команди і директиви асемблера
<8> ;(вивід на друк чи екран)
<9>endif
```

IF(N)DEF символічне_ім'я
фрагмент_програми_1
ELSE
фрагмент_програми_2
ENDIF

Дані директиви дозволяють управляти трансляцією фрагментів програми в залежності від того, визначено чи ні в програмі деяке **символічне_ім'я**. Директива **IFDEF** перевіряє, описано чи ні в програмі **символічне_ім'я**, і, якщо це так, то в об'єктний модуль вміщується **фрагмент_програми_1.** В іншому випадку, за наявності директиви **ELSE**, в об'єктний код вміщується **фрагмент_програми_2.**

Якщо ж директиви **ELSE** немає (і символічне_ім'я в програмі не описано), то вся частина програми між директивами **IF** і **ENDIF** ігнорується і в об'єктний модуль не включається.

Дія **IFNDEF** назад **IFDEF**. Якщо **символічного_ім'я** в програмі немає, то транслюється **фрагмент_програми_1**. Якщо воно присутнє, то при наявності **ELSE** транслюється **фрагмент_програми_2**. Якщо **ELSE** відсутній, а символічне_ім'я в програмі визначено, то частина програми, укладена між **IFNDEF** і **ENDIF**, ігнорується.

Приклад:

Розглянемо ситуацію, коли в об'єктний модуль програми потрібно включити один з трьох фрагментів коду. Який з трьох фрагментів буде включений в об'єктний модуль, залежить від значення деякого ідентифікатора sw:

1. якщо sw = 0, то згенерувати фрагмент для обчислення виразу y = x * 2 ^ n;
2. якщо sw = 1, то згенерувати фрагмент для обчислення виразу y = x / 2 ^ n;

3. якщо sw не визначено, то нічого не генерувати.

```
IFNDEF sw   ;якщо sw не визначено, то вийти з макросу
 EXITM
else         ;інакше — на обрахування
 movcl,n
 ife sw
  sal x,cl   ;множення на степінь 2 із зміщенням вліво
 else
  sar x,cl ;ділення на степінь 2 із зміщенням вправо
 endif
ENDIF
```

IF(N)B аргумент
фрагмент_програми_1
ELSE
фрагмент_програми_2
ENDIF

Дані директиви використовуються для перевірки фактичних параметрів, переданих в макрос. При виклику макрокоманди вони аналізують значення аргументу, і, залежно від того, чи дорівнює воно пробілу чи ні, транслюється або **фрагмент_програми_1**, або **фрагмент_програми_2**. Який саме фрагмент буде обраний, залежить від коду директиви:

• директива **IFB** перевіряє *рівність* аргументу *пробілу*. Як аргумент можуть виступати ім'я або число;

• якщо його значення дорівнює пробілу (тобто фактичний аргумент при виклику макрокоманди не було наведено), то транслюється і поміщається в об'єктний модуль **фрагмент_программи_1;**

• в іншому випадку, за наявності директиви **ELSE**, в об'єктний код поміщається **фрагмент_програми_2.** Якщо ж директиви **ELSE** немає, то при рівності аргументу пробілу вся частина програми між директивами **IFB** та **ENDIF** ігнорується і в об'єктний модуль **НЕ** включається.

Дія **IFNB** назад **IFB**. Якщо значення аргументу в програмі НЕ ДОРІВНЮЄ ПРОБІЛУ, то транслюється **фрагмент_програми_1.**

Приклад: рядки в макровизначенні, які перевірятимуть, чи вказується фактичний аргумент при виклику відповідної макрокоманди:

```
Show macro reg
IFB
display 'не заданий регістр'
exitm
ENDIF
...
endm
```

Якщо тепер в сегменті коду викликати макрос *show* без аргументів, то буде виведено повідомлення про те, що регістр не задано і генерація макророзширення буде припинена директивою **exitm**.

IFIDN(I) аргумент_1, аргумент_2
фрагмент_програми_1
ELSE
фрагмент_програми_2
ENDIF

У цих директивах перевіряються **аргумент_1** і **аргумент_2** як рядки символів. Який саме код - **фрагмент_програми_1** або **фрагмент_програми_2** - транслюватиметься за результатами порівняння, залежить від коду директиви.

Парність цих директив пояснюється тим, що вони дозволяють враховувати, або не враховувати відмінність малих і великих літер. Так, директиви **IFIDNI** і **IFDIFI** ігнорують цю відмінність, а **IFIDN** і **IFDIF** - враховують.

Директива **IFIDN (I)** порівнює символьні значення **аргумент_1** і **аргумент_2.**

Якщо результат порівняння додатній (>0), то **фрагмент_програми_1** транслюється і розміщується в об'єктний модуль. В іншому випадку, за наявності директиви **ELSE**, в об'єктний код поміщається **фрагмент_програми_2.**

Якщо ж директиви **ELSE** немає, то вся частина програми між директивами **IFIDN (I)** і **ENDIF** ігнорується і в об'єктний модуль не включається.

IFDIF(I) аргумент_1, аргумент_2
фрагмент_програми_1
ELSE
фрагмент_програми_2
ENDIF

Дія **IFDIF (I)** обернено **IFIDN (I).**

Якщо результат порівняння від'ємний (<0) (***рядки не збігаються***), транслюється **фрагмент_програми_1**. В іншому випадку все відбувається аналогічно розглянутим раніше директивам.

Як ми вже згадували, ці директиви зручно застосовувати для перевірки фактичних аргументів макрокоманд.

Приклад: Перевіримо, який з регістрів - **AL** або **AH** - переданий в макрос як параметр (перевірка проводиться без урахування відмінності малих і великих літер):

```
make_signed_word   macro   signed_byte
IFDIFI   <al>, <signed_byte>        ; впевнитись, що операндом НЕ є AL
        mov   al, signed_byte
ENDIF
        cwb
        endm
```

3.3.1.4. Операції в макровизначеннях

Існує **4** операції в макровизначеннях.
1.;; - Коментар, який не включається в лістинг;
2. & - конкатенація;
3. перевизначення;
4. скасування.
& - Конкатенація,дозволяє задавати модифіковані мітки і оператори;

Приклад:

```
DEF_TABLE MACRO SUFFIX, LENGTH
TABLE& SUFFIX DB LENGTH DUP(?)
ENDM
```

```
DEF_TABLE A,5
……..
TABLE A DB 5 DUP(?)
```

Якщо відзначити виклик DEF_TABLE A, 5, то утвориться розширення
TABLE A DB 5 DUP (?)

Перевизначення. Якщо в програмі **описати** макрокоманду **з тим же ім'ям**, яке було раніше у певній макрокоманди, то попереднє її визначення вже **НЕ** діє:

```
A MACRO Y
 INC Y
 INC BX
ENDM
A BX
A MACRO X, Z
 CMP X, 0
 CMP BH, 0
 JE Z
 JE EL
ENDM
A BM, EL
```

Скасування. Макровизначення можна знищити директивою
PURGE <ім'я макросу>
З цього моменту звертатися до зазначеної макрокоманди не можна.

3.3.1.5. Використання бібліотек макрокоманд

Крім макрокоманд, визначених у програмі, можна використовувати макрокоманди з бібліотеки MACRO.LIB, або створити свої власні і записати туди Тоді не потрібно наводити визначення, а викликати їх з бібліотеки за допомогою директиви:

INCLUDE <ім'я файлу>

Краще це зробити в умовній директиві оскільки макровизначення зчитується на першому проході транслятора. Якщо якісь макроозначення непотрібні, то їх можна

```
IF1
 INCLUDE MACRO.LIB
ENDIF
```

```
PURGE MAC1, MAC2, MAC3
```

видалити:

Всього є приблизно **40** стандартних макрокоманд у вигляді .ASM файлу. Вони виконують багато функцій, аналогічних функціям DOS або BIOS.

Директиву **INCLUDE** можна використовувати і для включення інших файлів
Замість цієї директиви асемблер підставить весь текст файлу
INCLUDE A: MACROS.TXT

Лекція 14

Розділ 4. Структури і записи

4.1. Записи

4.1.1. Опис типу запису

Записи з бітовими полями – це тип записів, в яких кожне поле має записану в бітах довжину і ініціалізовану деяким значенням. Розмір запису з бітовими полями визначається сумою розмірів його полів.

Перевагою даних такого типу є можливість збереження даних в найкомпактнішому вигляді. Наприклад, при необхідності збереження 16-ти прапорців, кожен з яких має два стани, традиційно використовується область пам'яті із 16-ти байтів чи слів, а при використанні записів з бітовими полями потрібно всього 16 біт.

Оголошення запису в режимі **IDEAL** має наступний синтаксис:

Ім'я RECORD [поле[,поле...]]

Кожне "***поле***" має наступний синтаксис:

Ім'я_поля : вираз_для_розміру[=значення]

Параметр "***ім'я_поля***" – це символьне ім'я поля запису.

TASM для розміщення цього поля видає область пам'яті розміром, який отриманий в результаті обчислення "***вираз_для_розміру***". "***Значення***" описує дані, якими ініціалізується поле (це значення буде присвоюватися полю при кожному створенні екземпляру запису такого типу). "***Значення***" и "***вираз_для_розміру***" не можуть бути відносними адресами.

Наприклад:

```
REC RECORD Am:3, Bm:3 = 7
DATE RECORD Yea:7, Mon:4, Day:5
```

Імена полів запису являються глобальними ідентифікаторами і не можуть перевизначатися. Значення імені поля при використанні в виразах являється лічильником зміщення для пересилки поля в праве крайнє положення.

"***Ім'я***" являється символьним іменем даних типу запис, за яким до нього можна звертатися в тексті програми. Крім того, ці імена можна використовувати для створення змінних і розміщення їх в пам'яті.

Данні типу запис (не окремі поля !!!) являються переобумовленими, тобто в тексті програми можна кілька раз оголошувати дані такого типу з одним і тим самим іменем.

Для визначення даних типу запис **TASM** підтримує багаторядковий синтаксис.

4.1.2. Опис змінних записів

При створенні екземпляру даних типу запис в якості директиви оголошення і розподілення даних використовується ім'я даних типу запис.

Синтаксис опису змінних має вигляд:

Ім'я_змінної ім'я_типу <поч_зн[; поч._зн]>

де **поч_зн** – це:

- Константний вираз;
- чи «?»;
- чи порожньо.

```
R1 Rec <5,6> ; Am=5, Bm = 6
R2 Rec <,?>
VIC DATE <49,7,8> Yea=49, Mon=7, Day=8
```

Якщо **порожньо**, то береться початкове значення із опису типу. Передбачені і масиви записів:

```
X DATE 100 DUP (<>) ; масив
```

Наприклад, якщо в програмі визначено тип:

```
MyRec RECORD Val:3=4,Mode:2,Asize:4=15
MTest MyRec ?
```

то оператором MTest MyRec ? буде створений екземпляр запису типу MyRec, визначений змінною MTest. Екземпляр запису завжди має розмір байту, слова чи подвійного слова, в залежності від числа бітів у визначенні запису.

4.1.3. Робота з полями запису

Якщо сума значень ширини полів <= 8, то розмір екземпляру записів буде 1 байт, більше за 8 і <=16 - слово, більше за 16 и <=32 - подвійне слово. Перше описане поле поміщається в старші значущі біти запису, наступні поля поміщаються в наступні біти, які розташовані праворуч. Якщо описані поля не займають повні 8, 16 чи 32 біта, повний запис зміщується направо так, що останній біт поля стає молодшим бітом запису. Не використані біти в старшій частині ініціалізуються нулями.

MTest

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1
							Val			Mode		Asize			

Якщо при ініціалізації екземпляру даних типу запис якісь значення ініціалізації полів записів упущені, то TASM заміняє їх значенням 0. Найпростіший метод ініціалізації екземпляру запису заключається в присвоєнні полям екземпляру значень, вказаних у визначенні типу запису, наприклад:

```
MyRec{}
```

еквівалентне

```
DW (4 SHL 6) + (0 SHL 4) + (15 SHL 0)
```

4.1.4. Значення імені поля

Пусті фігурні дужки являють собою нульове значення ініціалізації запису. Значення, вказані в ініціалізаторі, визначають, які із значень ініціалізації полів будуть перекриватися і з якими значеннями. При цьому використовується синтаксис:

{[ім'я_поля=вираз[, ім'я_поля=вираз..]]}

Тут "***ім'я_поля***" – ім'я поля в записі, "**вираз**" - значення, яке повинно бути призначене вказаному полю в екземплярі запису. Значення «?» еквівалентне 0. Всі значення, не вказані в ініціалізаторі, встановлюються TASM рівними значеннями, вказаним в визначенні запису.

Наприклад:

```
MyRec {Val=2,ASize=?}
```

еквівалентне

```
DW (2 SHL 6) + (0 SHL 4) + (0 SHL 0)
```

Іншим методом ініціалізації екземпляру запису являється застосування ініціалізатора з кутовими дужками (<>). Значення, вказані в ініціалізаторі, не мають назв, але вони повинні бути задані в тому ж порядку, що і відповідні поля відповідного запису.

Синтаксис ініціалізатора такого типу:

<[вираз[, вираз..]]>

де "***вираз***" - значення, що задається для відповідного поля у визначенні запису. Пропущене значення вказує, що буде ініціалізуватися значення, яке прийняте за замовчуванням (із визначення запису). Ключове слово ? вказує, що значення ініціалізації 0.

Наприклад:

```
MyRec <,2,?> ; Mode=2,Asize=0
```

еквівалентне

```
DW (4 SHL 6) + (2 SHL 4) + (0 SHL 0)
```

Якщо в ініціалізаторі вказано менше значень, ніж необхідно для ініціалізації всіх полів екземпляру, TASM для всіх інших полів використовує значення, прийняте за замовчуванням.

Якщо ім'я поля структури має значення зміщення поля в байтах від початку структури, то імені поля запису асемблер присвоює таке ж визначене число. Воно дорівнює кількості бітів, на яку поле потрібно змістити праворуч, щоб воно виявилося притиснутим до правого краю слова.

Зустрічаючи ім'я поля запису в тексті програми, асемблер заміняє його своїми значеннями. Це потрібно для підготовки виконання команд зміщення після виділення поля командою **AND**. В попередньому фрагменті поле ***Day*** являється крайнім і щоб перевірити його зміст, не потрібно його зміщувати праворуч.

Очевидно, що, якщо потрібно перевірити поле ***Mon***, то його попередньо потрібно змістити праворуч на стільки бітів, скільки займає поле ***Day*** чи на значення Mon = 5.

Отже, щоб перевірити, поле Mon = 7?, маємо наступний код:

```
MOV AX, VIC
AND AX, MASK Mon; AX: 0M0
MOV CL, Mon; константа 5 до CL
SHR AX, CL; AX: 00M
CMP AX,7
JE Yes
No:
```

Оператор **MASK** повертає *бітову маску поля*. Оператор **WIDTH** повертає *довжину поля* в бітах.

Оператор MASK

Синтаксис

MASK ім'я_поля_запису

чи

MASK ім'я_запису

Оператор **MASK ім'я_поля_запису** повертає бітову маску, яка рівна 1, для бітових позицій, зайнятих заданим полем запису, і рівну 0 для інших бітових позицій.

Оператор **MASK ім'я_запису** повертає бітову маску, в якій біти, зарезервовані для бітових полів, встановлені в 1, інші – в 0.

Приклад:

```
.DATA
COLOR    RECORD BLINK:1,BACK:3,INTENSE:1,FORE:3
MESSAGE  COLOR <0,5,1,1>
.CODE
MOV AH,MESSAGE ;завантаження первинного      0101 1001
AND AH,NOT MASK BACK ;закрити маску BACK     AND 1000 1111
; 0000 1001
OR  AH,MASK BLINK ;відкрити маску BLINK      OR   1000 0000
; 1000 1001
XOR AH,MASK INTENSE ; XOR 0000 1000
; 1000 0001
```

Оператор WIDTH

Синтаксис

WIDTH ім'я_запису – повертає загальну кількість бітів, які зарезервовані в описі запису;

WIDTH ім'я_поля_запису – повертає загальну кількість бітів, які зарезервовані для поля в описі запису;

Крім того, як уже зазначалося, ім'я поля, при використанні у виразі являється лічильником зміщення для пересилання поля в крайнє праве положення.

Приклад

```
.DATA
COLOR    RECORD BLINK:1,BACK:3,INTENSE:1,FORE:3
WBLINK   EQU WIDTH BLINK; "WBLINK"=1 "BLINK"=7
;(показати звідки)
WBACK    EQU WIDTH BACK ; "WBACK"=3 "BACK"=4
WINTENSE EQU WIDTH INTENSE; "WINTENSE"=1 "INTENST"=3
WFORE    EQU WIDTH FORE         ; "WFORE"=3  "FORE"=0
WCOLOR   EQU WIDTH COLOR     ; "WCOLOR"=8
PROMPT   COLOR <1,5,1,1>
.CODE
.....
IF (WIDTH COLOR) GE 8       ;якщо COLOR=16 біт, завантажити
MOV AX,PROMPT               ;в 16-бітний регістр
ELSE                        ;інакше
MOV AL,PROMPT                ;завантажити в 8-бітний регістр
XOR AH,AH                   ;очистити старший 8-бітний регістр
END IF                      ;регістр
```

Ще один приклад використання:

```
.DATA
COLOR    RECORD BLINK:1,BACK:3,INTENSE:1,FORE:3
CURSOR   COLOR <1,5,1,1>
.CODE
;збільшити на 1 "BACK" в "cursor" без зміни інших
;полів
MOV AL,CURSOR ;завантажити значення з пам’яті
MOV AH,AL ;створити робочу копію   1101 1001   AH/AL
AND AL,NOT MASK BACK ; AND  1000 1111    MASK
; 1000 1001
;замаскувати старші біти для маскування старого CURSOR
MOV CL,BACK ;завантажити позицію біту
SHR AH,CL ;змістити праворуч 0000 0101 AH
INC AH ;збільшити на 1 0000 0110    AH
SHL AH,CL ;змістити назад ліворуч  0110 0000  AH
AND AH,MASK BACK ; AND 0111 0000 MASK
; 0110 0000    AH
OR AH,AL ; OR      1000 1001     AL
;  1110 1001    AH
MOV CURSOR,AH
```

Додаткові можливості опрацювання

Розуміючи важливість типу даних «запис», для ефективного програмування, розробники транслятора TASM, починаючи з версії 3.0, включили в систему його команд дві додаткові команди на правах директив. Останнє означає, що ці команди зовні мають формат звичайних команд асемблера, але після трансляції вони зводяться до однієї чи кількох машинних команд. Введення цих команд в мову TASM підвищує наочність роботи з записами, оптимізує код та зменшує розмір програми. Ці команди дозволяють приховати від програміста дії з виділення і встановлення окремих полів запису. Для встановлення значення і вибірки значення деякого поля запису використовуються команди **SETFIELD** і **GETFIELD**

SETFIELD імя_елементу_запису приймач, регістр_джерело – встановлює за зміщенням **імя_елементу_запису приймача** значення, яке дорівнює **регістр_джерело.**

Робота команди **SETFIELD** заключається в наступному. Положення запису визначається операндою приймач, який може являти собою ім'я регістру чи адресу пам'яті. Операнда **ім'я_елементу_запису** визначає елемент запису, з яким ведеться робота (по суті, якщо ви були уважні, він визначає зміщення елементу в записі відносно меншого розряду). Нове значення, в яке необхідно встановити вказаний елемент запису, повинно міститися в операнді **регістр_джерело**. Обробляючи дану команду, транслятор генерує послідовність команд, які виконують наступні дії.

1. Зміщення вмісту операнди **регістр_джерело** ліворуч на кількість розрядів, яка відповідає положенню **елементу_запису**.

2. Виконання логічної операції **OR** над операндами **приймач** і **регістр_джерело**. Результат операції розміщується в операнді **приймач**.

Важливо відмітити, що **SETFIELD** не виконує попередньої очистки елементу, в результаті, після логічного додавання командою **OR** можливе накладання старого вмісту елементу і нового встановлюваного значення. Тому потрібно попередньо підготувати поле в записі шляхом присвоєння йому значення 0.

GETFIELD ім'я_елементу_запису регістр_приймач, джерело – витягує значення за зміщенням **ім'я_елементу_запису,** знайдене в джерелі чи за **адресою пам'яті,** і встановлює в це значення відповідну за зміщенням **ім'я_елементу_запису** частину **регістру_приймача.**

Дія команди **GETFIELD** обернена дії **SETFIELD**. В якості операнда **джерело** може бути вказаний регістр чи адреса пам'яті. В регістр, вказаний операндою **регістр_приймач** поміщається результат роботи команди – значення елементу запису. Цікава особливість пов'язана з операндом **регістр_приймач.** Команда **GETFIELD** завжди використовує 16-розрядний регістр, навіть, якщо ви вкажете в цій команді ім'я 8-розрядного регістру.

Обробляючи дану команду, транслятор генерує послідовність команд, які виконують наступні дії.

1. Пересилання значень **джерела** в **ім'я_елементу_запису регістр_приймач.**

2. Виконання логічної операції **AND** над операндами **регістр_приймач** і **маскою** (одиничні біти за зміщенням **ім'я_елементу_запису**). Результат операції розміщується в операнді **регістр_приймач.**

В якості прикладу застосування команд **SETFIELD** і **GETFIELD** розглянемо приклад.

```
masm
model small
stack 256
iotest record i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
.data
flag   iotest <>
.code
main:
        mov   ax,@data
        mov   ds,ax
        mov   al,flag  ;al=108=06ch=01101100
        mov   bl,3
        setfield i5 al,bl ;al=111=06fh=01101111
        mov   al,110
        xor   bl,bl
        getfield        i5 bl,al ;i5=00  bl=10=2h=2  al=06eh=01101110
        mov   bl,1
        setfield i4 al,bl ;al=110=06eh=01101110 bl=100=4h=4
        setfield i5 al,bl ;al=108=06eh=01101110 bl=100=4h=4
exit:
        mov   ax,4c00h      ; стандартний вихід
        int   21h
end main                    ;кінець програми
```

4.3. Команди переривань

Їх виконання має багато спільного з командою виклику процедури. Тут також відбувається перехід до групи команд, яка починається з визначеної адреси і називається програмою обробки переривань.

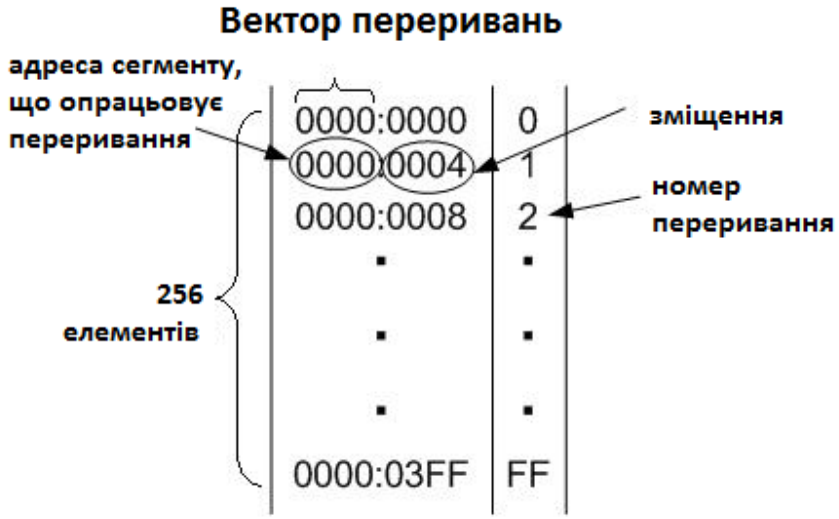
Якщо під час виклику процедури можна використовувати різні режими адресації, щоб визначити адресу процедури, то перехід до обробки переривань є непрямим і здійснюється через вектор переривань. Вектор переривань займає два слова і зберігає значення регістру коду **CS** і покажчика команд **IP**, з яких починається відповідна програма обробки переривань. Всього може бути 256 переривань, кожне з яких, характеризується своїм номером, наприклад, 21H. Адреса вектора переривань визначається як номер, помножений на 4: 4*21H.

Де знаходяться вектори переривань в пам’яті?

Вони знаходяться на основній системній платі пам’яті RAM 64Кбайт. Найперший 1Кбайт займають вектори переривань.

Головною виконавчою програмою ПЕОМ являється базова система введення/виведення (Basic Input/Output System) BIOS.

Вона запам’ятовує символи, які набирають на клавіатурі, відображає символи на екрані, здійснює обмін даними між пристроями ПЕОМ: дисководами, принтером та іншими пристроями.



Таблиця значення деяких, найважливіших, переривань

Номер	Опис
0	**Помилка ділення.** Викликається автоматично після виконання команд DIV чи IDIV, якщо, в результаті ділення, відбувається переповнення (наприклад, при діленні на 0). DOS зазвичай при опрацюванні цього переривання виводить повідомлення про помилку і зупиняє виконання програми. Для процесора 8086 при цьому адреса повернення вказує на наступну після команди ділення команду, а в процесорі 80286 – на перший байт команди, що викликала переривання.
1	**Переривання покрокового режиму.** Здійснюється після виконання кожної машинної команди, якщо в слові прапорців встановлений біт покрокового трасування TF. Використовується для налагодження програм. Це переривання не здійснюється після виконання команди MOV в сегментні регістри чи після завантаження сегментних регістрів командою POP
2	**Апаратне не замасковане переривання.** Це переривання може використовуватися по-різному в різних машинах. Зазвичай виробляються при помилці парності в оперативній пам’яті і при запиті переривання від співпроцесора.
3	**Переривання для трасування** Це переривання генерується при виконанні однобайтової машинної команди з кодом CCh і зазвичай використовується налагоджувачами для встановлення точки переривань.
4	**Переповнення** Генерується машинною командою INTO, якщо встановлений прапорець OF. Якщо прапорець не встановлений, то команда INTO виконується як NOP. Це переривання використовується для обробки помилок при виконанні арифметичних операцій.
5	**Друк копії екрану.** Генерується при натисканні та клавіатурі клавіші PrtScr. Зазвичай використовується для друку образу екрану. Для процесора 80286 генерується при виконанні машинної команди BOUND, якщо значення, що перевіряється вийшло за межі заданого діапазону.
6	Не визначений код операції чи довжина команди більше 10 байт (для процесора 80286).

7	Особливий випадок відсутності математичного співпроцесора (процесор 80286).
8	***IRQ0*** – переривання інтервального таймера, виникає 18,2 рази в секунду.
9	***IRQ1*** – переривання від клавіатури.
A	***IRQ2*** – використовується для каскадування апаратних переривань.
B	***IRQ3*** – переривання асинхронного порту **COM2.**
C	***IRQ4*** – переривання асинхронного порту **COM1.**
D	***IRQ5*** – переривання від контролера жорсткого диску для XT.
E	***IRQ6*** – переривання генерується контролером флоппі-диску.
F	***IRQ7*** – переривання принтера.
10	Обслуговування відеоадаптера.
11	Визначення конфігурації пристроїв в системі.
12	Визначення розміру оперативної пам’яті в системі.
13	Обслуговування дискової системи.
14	Послідовне введення/виведення.
15	Розширений сервіс для AT-комп’ютера.
16	Обслуговування клавіатури.
17	Обслуговування принтера.
18	Запуск BASIC в ПЗП, якщо він є.
19	Завантаження операційної системи.
1A	Обслуговування годинника.
1B	Оброблювач переривань Ctrl-Break.
1C	Переривання виникає 18.2 рази в секунду.
1D	Адреса відео таблиці для контролера відеоадаптера 6845.
1E	Покажчик на таблицю параметрів дискети.
1F	Покажчик на графічну таблицю для символів з кодами ASCII 128-255.
20-5F	Використовується DOS чи зарезервовано для DOS.
60-67	Переривання, зарезервовані для користувача.
68-6F	Не використовуються.
70	***IRQ8*** – переривання від годинника реального часу.
71	***IRQ9*** – переривання від контролера EGA.
72	***IRQ10*** – зарезервовано.

73	***IRQ11*** – зарезервовано.
74	***IRQ12*** – зарезервовано.
75	***IRQ13*** – переривання від математичного співпроцесора.
76	***IRQ14*** – переривання від контролера жорсткого диску.
77	***IRQ15*** – зарезервовано.
78 - 7F	Не використовуються.
80-85	Використовуються інтерпретатором BASIC.
86-F0	Зарезервовані для BASIC.
F1-FF	Не використовуються.

Таблиця резервування груп переривань

00000	Вектори переривань BIOS (0H - 1FH).
00080h	Вектори переривань DOS (20H - 3FH).
0180h	Векторів переривань користувача (60H - 7FH).
0100h	Зарезервовані вектори переривань (40H - 5FH).
0400h	Області даних BIOS.
0200h	Векторів переривань Basic (80H - FFH).
0600h	62,5K Область пам’яті доступна для читання і запису.
0500h	Області даних Basic та DOS.

Як і виклик підпрограм, переривання може мати дистанцію **NEAR** чи **FAR**.

Під час виконання програми опрацювання переривань забороняють замасковані переривання і трасування програми. Тому прапорці **IF** і **TF** обнуляються. Отже, програми опрацювання переривань змінюють значення регістрів прапорців. Тому перед початком опрацювання переривань в стек відправляється регістр прапорців **F**. Тому, на відмінну від виклику підпрограми, тут, в стеку, буде три слова:

a	IP
a+2	CS
a+4	F

INT (interrupt) - команда умовного переривання

Перериває опрацювання програми, передає управління в **DOS** чи **BIOS** для визначеної дії і потім повертає управління в перервану програму для продовження опрацювання. Найчастіше переривання використовується для виконання операції введення чи виведення. Для виходу з програми на опрацювання переривань і для подальшого повернення команди **INT** виконує наступні дії:

1. Зменшується покажчик стеку на 2 і заносить в вершину стеку вміст регістру прапорців;
2. Очищує прапорці **TF** і **IF**;
3. Зменшує покажчик стеку на 2 і заносить вміст регістру **CS** в стек;
4. Зменшує покажчик стеку на 2 і заносить в стек значення командного покажчика
5. Забезпечує виконання необхідних дій;
6. Відновлює із стеку значення регістру і повертає управління в перервану програму на команду, наступну після **INT**.

Цей процес виконується повністю автоматично. Необхідно лише визначити сегмент стеку достатньо великим для запису в нього значень регістрів.

Розглянемо два типи переривань: команду **BIOS INT 10H** і команду **DOS INT 21H** для виведення на екран і введення з клавіатури. В наступних прикладах, в залежності від потреб використовують як **INT 10H** так і **INT 21H**.

Наприклад:

MOV AH, номер_функції

<параметри>

INT 21H; переривання DOS

Існує три команди переривань:

1. INT;
2. INTO;
3. IRET.

INTO (interrupt if overflow) – переривання при переповненні. Призводить до переривання при виникненні переповнення (прапорець **OF** встановлений в 1) і виконує команду **IRET 4**. Адреса підпрограми опрацювання переривань (вектор переривання) знаходиться за адресою 10H.

IRET (interrupt return) – команда повернення після переривання. Забезпечує повернення із підпрограми опрацювання переривання. Команда **IRET** виконує наступне:

1) розміщує слово із вершини стеку в регістр **IP** і збільшує значення **SP** на 2;

2) розміщує слово із вершини стеку в регістр **CS** і збільшує значення **SP** на 2;

3) розміщує слово із вершини стеку в регістр прапорців і збільшує значення **SP** на 2.

ПЕРЕРИВАННЯ BIOS

INT 05H Друк екрану **–** виконує виведення вмісту екрану на друкуючий пристрій. Команда INT 05H виконує дану операцію з програми, а натискання кнопок Ctrl/PrtSc - з клавіатури. Операція забороняє переривання і зберігає позицію курсору.

INT 10H Управління дисплеєм **–** забезпечує екранні і клавіатурні операції.

Переривання INT 10H забезпечує управління всім екраном. В регістрі AH встановлюється код, що визначає функцію переривання. Команда зберігає вміст регістрів **BX, CX, DX, SI** та **BP**.

Приклад:

```
MOV  AH,02        ;встановити положення курсору
MOV  BH,00        ;сторінка 0
MOV  DH, 5        ;рядок
MOV  DL, 1        ;стовбець
INT  10H          ;викликати BIOS
```

чи

```
MOV  AX,0600H  ;повна прокрутка вверх всього екрану, очищуючи його пробілами
MOV  AX,0601H  ;прокрутити на один рядок вверх
MOV  BH,07     ;Атрибут: нормальний, чорно-білий
MOV  CX,0000   ;Координати від 00,00
MOV  DX,184FH  ;до 24,79 (повний екран)
INT  10H       ;викликати BIOS
```

INT 11H запит списку приєднаного обладнання **–** визначає наявність різних пристроїв в системі, результуюче значення повертає в регістр AX. При ввімкненні комп'ютера система виконує цю операцію і зберігає вміст AX в пам'яті за адресою 410h, значення бітів в регістрі AX:

Біт	**Пристрій**
15,14	Число підключених принтерів.
13	Послідовний принтер.
12	Ігровий адаптер.

11-9	Число послідовних адаптерів роз'єму RS232.
7,6	Число дискретних дисководів, при біті 0=1: 00=1, 01=2, 10=3 и 11=4.
5,4	Початковий відео-режим: 00 = не використовується, 01 = 40x25 плюс колір, 10 = 80x25 плюс колір, 11 = 80x25 чорно-білий режим.
1	Значення 1 говорить про наявність співпроцесора.
0	Значення 1 говорить про наявність одного чи кількох дискових пристроїв і завантаження операційної системи повинне здійснюватися з диску.

INT 12H Запит розміру фізичної пам'яті **–**повертає в регістр AX розмір пам'яті в кілобайтах, наприклад, 200_{16} відповідає в пам'яті 512К. Дана операція корисна для вирівнювання розміру програми у відповідності з доступною пам'яттю.

INT 13H Дискові операції введення/виведення **–** забезпечує операції введення/виведення для дискет і вінчестера.

INT 14H Управління комунікаційним адаптером **–** забезпечує послідовне введення/виведення через комунікаційний порт RS232. Регістр DX повинен містити номер (0 чи 1) адаптера роз'єму RS232. Чотири типи операції, що визначаються регістром AH, виконують прийом і передачу символів та повертають в регістр AX байт стану комунікаційного порту.

INT 15H Касетні операції введення/виведення чи спеціальні функції для комп'ютерів AT **–** забезпечує операції введення/виведення для касетного магнітофону, а також розширені операції для комп'ютерів AT.

INT 16H Введення з клавіатури **–** забезпечує три типи команд введення з клавіатури.

INT 17H Виведення на принтер **–** забезпечує виведення даних на друкуючий пристрій.

INT 18H Звертається до BASIC, вбудованого в ROM **–** викликає BASIC-інтерпретатор, що знаходиться в постійній пам'яті ROM.

INT 19H Перезапуск системи **–** дана операція при доступному диску зчитує сектор 1 з доріжки 0 в область початкового завантаження в пам'ять (сегмент 0, зміщення 7C00) і передає управління за цією адресою. Якщо дисковод не доступний, то операція передає управління через INT 18H в ROM BASIC. Дана операція не очищує екран і не ініціалізує дані в ROM BASIC, тому її можна використовувати із програми.

INT 1AH Запит і встановлення поточного часу і дати – зчитується і записується показання годинника у відповідності із значенням в регістрі AH. Для визначення тривалості виконання програми можна перед початком виконання встановити годинник в 0, а потім зчитати поточний час. Відлік часу йде приблизно 18,2 рази в секунду. Значення в регістрі AH відповідає наступним операціям:

AH=00 Запит часу. В регістрі CX встановлюється більша частина значення, а в регістрі DX - менша. Якщо після останнього запиту пройшло 24 години, то в регістрі AL буде не нульове значення.

AH=01 Установка часу. Час встановлюється по регістру CX (більша частина значення) і DX (менша частина значення)**.**

Коди 02 і 06 керують часом і датою для AT.

INT 1FH Адреса таблиці графічних символів **-** в графічному режимі має доступ до символів з кодами 128-255 в 1K таблиці, що містить по 8 байт кожен символ. Прямий доступ в графічному режимі забезпечується тільки до перших 128 ASCII-символам (від 0 до 127).

Переривання DOS

Під час своєї роботи BIOS використовує два модулі DOS: IBMBIO.COM та IBMDOS.COM, так як модулі DOS забезпечують велику кількість різних додаткових перевірок, то операції DOS простіші в використанні і менш машиннозалежні, ніж їх BIOS аналоги.

Модуль IBMBIO.COM забезпечує інтерфейс з BIOS низького рівня. Ця програма виконує управління введенням/виведенням при зчитуванні даних із зовнішніх пристроїв в пам’ять і запис з пам’яті на зовнішні пристрої.

Модуль IBM DOS.COM містить засоби управління файлами і ряд сервісних функцій, таких як блокування і деблокування записів. Коли користувацька програма видає запит INT 21H, то в програму IBM DOS через регістри передається визначена інформація. Потім програма IBMDOS транслює цю інформацію в один чи кілька викликів IBMBIO, яка в свою чергу викликає BIOS. Вказані зв’язки наведені в наступній схемі:

Рівень користувача	**Вищий рівень**	**Нижчий рівень**	**ROM**	**Зовнішній рівень**
Програмний запит в/в	DOS IBMDOS.COM	DOS IBMBIO.COM	BIOS	Пристрій

Переривання від 020h до 062h зарезервовані для операцій DOS. Нижче наведені найосновніші з них:

INT 20H Закінчення програми – запит закінчує виконання програми і передає управління в DOS. Даний запит зазвичай знаходиться в основній процедурі.

INT 21H Запит функцій DOS – основна операція DOS, що викликає визначену функцію у відповідності з кодом в регістрі AH.

INT 22H Адреса підпрограми опрацювання завершення задачі (див. **INT 24H**).

INT 23H Адреса підпрограми реакції на Ctrl/Break (див. **INT 24H**).

INT 24H Адреса підпрограми реакції на фатальну помилку - в цьому елементі і в двох попередніх містяться адреси, які ініціалізуються системою в префіксі програмного сегменту і які можна змінити для своїх цілей.

INT 25H Абсолютне зчитування з диску

INT 26H Абсолютний запис на диск

INT 27H Завершення програми, що залишає її резидентною – дозволяє зберегти COM-програму в пам'яті.

Функції переривання DOS INT 21H

Нижче наведені базові функції для переривання **DOS INT 21H**. Код функції встановлюється в регістрі **AH**:

Таблиця

00 Завершення програми (аналогічно INT 20H).

01 Введення символу з клавіатури з ехом на екран.

02 Виведення символу на екран.

03 Введення символу із асинхронного комунікаційного каналу.

04 Виведення символу на асинхронний комунікаційний канал.

05 Виведення символу на друк.

06 Пряме введення з клавіатури і виведення на екран.

07 Введення з клавіатури без еха і без перевірки Ctrl/Break.

08 Введення з клавіатури без еха і з перевіркою Ctrl/Break.

09 Виведення рядка символів на екран.

0A Введення з клавіатури з буферизацією.

0B Перевірка наявності введення з клавіатури.

0C Очистка буфера, введення з клавіатури і запит на введення.

0D Висунути диск.

0E Встановлення поточного дисководу.

0F Відкриття файлу через FCB.

10 Закриття файлу через FCB.

11 Початковий пошук файлу за шаблоном.

12 Пошук наступного файлу за шаблоном.

13 Видалення файлу з диску.

14 Послідовне зчитування з файлу.

15 Послідовний запис в файл.

16 Створення файлу.

17 Переіменування файлу.

18 Внутрішня операція DOS.

19 Визначення поточного дисководу.

1A Встановлення області передачі даних (DTA).

1B Отримання таблиці FAT для поточного дисководу.

1C Отримання FAT для будь-якого дисководу.

21 Читання із диску з прямим доступом.

22 Запис на диск з прямим доступом.

23 Визначення розміру файлу.

24 Встановлення номеру запису для прямого доступу.

25 Встановлення вектора переривань.

26 Створення програмного сегменту.

27 Читання блоку записів з прямим доступом.

28 Запис блоку з прямим доступом.

29 Перетворення імені файлу у внутрішній параметр.

2A Отримання дати (CX-рік,DH-місяць,DL-день).

2B Встановлення дати.

2C Отримання часу (CH-години,CL-хвилини,DH-секунди,DL-1/100c).

2D Встановлення часу.

2E Встановлення/відміна верифікації запису на диск.

Лекція 15

Розділ 4 Структури і записи

4.2. Визначення структур та об'єднань

Розглянуті нами раніше масиви являють собою сукупність однотипних елементів. Але часто в додатках виникає необхідність розглядати деяку сукупність даних різного типу як деякий єдиний тип. Це дуже актуально, наприклад, для програм баз даних, де з одним об'єктом може асоціюватися сукупність даних різного типу.

Структури і об'єднання - це спеціальні конструкції асемблера, що дозволяють змішувати і поєднувати дані різних типів. Структура являє тип даних, в якому міститься один або більше елементів даних, званих членом структури.

Структура відрізняється від запису тим, що члени структури завжди представлені певною кількістю байтів. Розмір структури визначається сумарним розміром всіх елементів даних, що входять до неї.

Об'єднання подібні структурам, за винятком того, що всі члени об'єднання займають одну і тусаму ділянку пам'яті. Розмір об'єднання таким чином визначається розміром найбільшого його члена. Об'єднання застосовуються в тих випадках, коли блок пам'яті повинен представляти один з декількох можливих типів даних.

Асемблер дозволяє застосовувати вкладені структури та об'єднання, хоча зловживання цим значно ускладнює сприйняття програми.

З метою підвищити зручність використання мови асемблера в нього також був введений такий тип даних.

За визначенням структура - це тип даних, що складається з фіксованого числа елементів різного типу.

Для використання структур в програмі необхідно виконати три дії.

1. Задати шаблон структури. За змістом це означає визначення нового типу даних, який згодом можна використовувати для визначення змінних цього типу.

2. Визначити примірник структури. Цей етап має на увазі ініціалізацію конкретної змінної з наперед визначеною (за допомогою шаблону) структурою.

3. Організувати звернення до елементів структури.

Дуже важливо добре розуміти різницю між описом структури в програмі та її визначенням. Опис структури в програмі означає лише вказівку компілятору її схеми, чи шаблона; пам'ять при цьому не виділяється. Компілятор отримує з цього шаблону інформацію про розташування полів структури і їх значеннях за замовчуванням. Визначення структури означає вказівка транслятору на виділення пам'яті і привласнення цієї області пам'яті символічного імені. Описати структуру в програмі можна тільки один раз, а визначити - будь-яку кількість разів. Після того як структура визначена, тобто її ім'я пов'язане з ім'ям змінної, можливе звернення до полів структури за їх іменами.

Опис шаблону структури

Опис шаблону структури має наступний синтаксис:

ім’я_структури STRUC

<опис полей>

ім’я _структури ENDS

Тут <опис полів> представляє собою послідовність директив опису даних **DB, DW, DD, DQ і DT**. Їх операнди визначають розмір полів і при необхідності - початкові значення. Цими значеннями будуть, можливо, ініціалізуватися відповідні поля при визначенні структури.

Як ми вже зазначили, при описі шаблона пам'ять не виділяється, так як це всього лише інформація для транслятора. Розташування шаблону в програмі може бути довільним, але, слідуючи логіці роботи однопрохідного транслятора, шаблон повинен бути описаний раніше, ніж визначається змінна з типом даних структури. Тобто при описі в сегменті даних змінної з типом деякій структури її шаблон необхідно описати на початку сегмента даних або перед ним.

Розглянемо роботу зі структурами на прикладі моделювання бази даних про співробітників деякого відділу. Для простоти, щоб піти від проблем перетворення інформації при введенні, домовимося, що всі поля символьні.

Визначимо структуру запису цієї бази даних наступним шаблоном:

```
workerstruc ;інформація про співробітника
namdb 30 dup (" ")          ;прізвище, і;мя, по-батькові
sex db " "                  ;стать
position db 30 dup (" ")    ;посада
age db 2 dup (" ")          ;вік
standing db 2 dup (" ")     ;стаж
salarydb 4 dup (" ")        ;з/пврублях
birthdate db 8 dup (" ")    ;дата нарождення
worker ends
```

Створення примірника структури

Для використання описаної за допомогою шаблону структури в програмі необхідно визначити змінну з типом даних структури. Для цього використовується наступна синтаксична конструкція:

[ім'я змінної] імя_структури <[список значень]>

Тут: ім'я змінної - ідентифікатор змінної даного структурного типу. Завдання імені змінної необов'язково. Якщо його не вказати, буде просто виділена область пам'яті розміром в суму довжин усіх елементів структури;

список значень - узятий в кутові дужки список початкових значень елементів структури, розділених комами, але його задання необов'язково ..

Якщо список зазначений не повністю, то всі поля структури для даної змінної ініціалізуються значеннями із шаблону. Допускається ініціалізація окремих полів, але в цьому випадку пропущені поля, які будуть ініціалізовані значеннями із шаблону структури, повинні відділятися комами. Якщо при визначенні нової змінної з типом даних структури ми згодні з усіма значеннями полів в її шаблоні (тобто заданими за замовчуванням), то потрібно просто написати кутові дужки.

Наприклад:

```
victor worker <>
```

Для прикладу визначимо кілька змінних з типом описаної раніше структури:

```
data segment
sotrl worker <"Гурко Андрій Вячеславович",,'художник','33' , '15' , '1800' , '26.01.64 >
sotr2worker<"Михайлова Наталя Геннадіївна",'ж','програміст', '30','10'.'1680'.'27.10.58>
sotr3worker<"Степанов Юрій Лонгиінович",,'художник','38' , '20','1750','01.01.58'>
sotr4worker<"Юрова Олена Олександрівна",'ж','програміст','32' , '2' , , '09.01.66>
sotr5worker<> ;тут всі значення за замовчуванням
data ends
```

Методи роботи зі структурами

Сенс введення структурного типу даних в будь-яку мову програмування полягає в об'єднанні різнотипних змінних в один об'єкт. У мові повинні бути засоби доступу до цих змінних всередині конкретного екземпляра структури. Для того щоб звернутися в команді на поле деякої структури, використовується спеціальний оператор. (крапка):

адресний_вираз.імя_поля_структури

тут:

• адресний_вираз - ідентифікатор змінної деякого структурного типу або вираз в дужках відповідно до зазначених раніше синтаксичних правил;

• імя_поля_структури - ім'я поля з шаблону структури (це насправді теж адреса, а точніше, зміщення поля від початку структури).

Оператор «.» (Крапка) обчислює вираз (адресний_вираз) + (імя_поля_структури).

Продемонструємо за допомогою певної нами структури worker деякі прийоми роботи зі структурами. Наприклад, потрібно витягти в регістр **AX** значення поля з віком. Так як навряд чи вік працездатної людини може бути більше 99 років, то після приміщення вмісту цього

символьного поля в регістр **AX** його буде зручно перетворити в двійкове подання командою **AAD**. Будьте уважні, тому що через принцип зберігання даних «молодший байт за молодшу адресу» старша цифра віку буде поміщена в **AL**, а молодша - в **AH**. Для коригування досить використовувати команду **xchg ah, al**:

```
mov ax,word ptr sotrl.age ;в al вік sotrl
xchgah, al
;аналогічно:
lea bx,sotrl
mov ax,word ptr [bx].age
xchgah, al
aad;перетворення BCD в двійкове подання
```

Давайте уявимо, що співробітників не четверо, а набагато більше, і до того ж їх число і інформація про них постійно змінюється. В цьому випадку втрачається сенс явного визначення змінних з типом worker для конкретних особистостей.

Асемблер дозволяє визначати не тільки окрему змінну з типом структури, а й масив структур.

Наприклад, визначимо масив з 10 структур типу worker:

```
mas_sotr worker 10 dup (<>)
```

Подальша робота з масивом структур проводиться так само, як і з одновимірним масивом. Тут виникає кілька запитань. Як бути з розміром і як організувати індексацію елементів масиву? Аналогічно іншим ідентифікаторам, визначеним у програмі, транслятор призначає імені типу структури та імені змінної з типом структури атрибут типу. Значенням цього атрибута є розмір в байтах, займаний полями структури. Витягти це значення можна за допомогою оператора:

TYPE імя_типу_структури

Після того як стає відомим розмір екземпляра структури, організація індексації в масиві структур не представляє особливої складності.

Приклад:

```
worker struc
……..
worker ends
……
mas_sotr worker 10 dup (<>)
……..
movbx, type worker ;bx=77=4Dh
leadi,mas_sotr
;витягти і винести на екран стать всіх співробітників:
mov cx,10
cycl:
mov dl,[di].sex
;вивід на екран вмісту
;поля  sex структури worker
adddi.bx ;до наступної структури в масиві mas_sort
loop cycl
```

Як виконати копіювання поля з однієї структури в відповідне поле іншої структури? Або як виконати копіювання всієї структури?

Приклад: виконаємо копіювання поля nam третього співробітника в поле nam п'ятого працівника

```
worker struc
……..
worker ends
……..
mas_sotr worker 10 dup (<>)
……..
mov bx,offset mas_sotr
mov si,(type worker)*2 ;si=77*2
add si,bx
mov di,(type worker)*4 ;si=77*4
add di,bx
movcx,30
repmovsb ; пересилка елементів послідовності (ds:esi/si) в (es:edi/di) пам’яті
```

Об'єднання

Уявімо ситуацію, коли ми використовуємо певну область пам'яті для розміщення того чи іншого об'єкта програми (змінної, масиву або структури). Раптом після деякого етапу роботи у нас відпадає потреба в цих даних. У звичайному випадку пам'ять залишається зайнятою до кінця роботи програми.

Звичайно, її можна було б задіяти для зберігання інших змінних, але без прийняття спеціальних заходів не можна змінити тип та ім'я даних. Непогано було б мати можливість перевизначити цю область пам'яті для об'єкта з іншими типом і ім'ям. Асемблер надає таку можливість у вигляді спеціального типу даних - об'єднанням.

Об'єднання - тип даних, що дозволяє трактувати одну й ту ж область пам'яті як дані, що мають різні типи та імена.

Опис об'єднань в програмі нагадує опис структур, тобто спочатку вказується шаблон, в якому за допомогою директив опису даних перераховуються імена і типи полів:

ім'я_об'єднання UNION

<опис полів>

ім'я_об'єднання ENDS

Відмінність об'єднань від структур полягає, зокрема, в тому, що при визначенні змінної типу об'єднання пам'ять виділяється відповідно до розміру максимального елемента. Звернення до елементів об'єднання відбувається за їх іменами, але при цьому потрібно, звичайно, пам'ятати, що

всі поля в об'єднанні накладаються один на одного. Одночасна робота з елементами об'єднання виключена. Як елементи об'єднання можна використовувати і структури.

Приклад, який ми зараз розглянемо, примітний тим, що крім демонстрації власне типу даних об'єднання, в ньому показується можливість взаємного вкладення структур та об'єднань. Постарайтеся уважно поставитися до аналізу цієї програми. Основна ідея тут в тому, що покажчик на пам'ять, формований програмою, може бути представлений у вигляді:

• 16-розрядного зміщення;

• 32-розрядного зміщення;

• пари з 16-розрядного зсуву і 16-розрядної сегментної складової адреси;

• пари з 32-розрядного зсуву і 16-розрядного селектора.

Які з цих покажчиків можна застосовувати в конкретній ситуації, залежить від режиму адресації (use16 або use32) і режиму роботи процесора. Шаблон об'єднання, описаний у прикладі, дозволяє спростити формування і використання покажчиків різних типів.

Вкладені об'єднання

Асемблер дозволяє визначати вкладені директиви **UNION** і **ENDS** всередині відкритого опису даних типу об'єднання.

У структурі кожен елемент даних починається відразу за закінченням попереднього. В об'єднанні кожен елемент починається з того ж зміщення, що і попередній. Дозволивши одному елементу даних містити ціле об'єднання, можна отримати великі можливості гнучкого управління даними.

Приклад:

```
CUNION   STRUC
CTYPEDB ?
 UNION;Початок об’єднання
  STRUC
CT0PAR1      DW 1
CT0PAR2      DB 2
    ENDS
  STRUC
CT1PAR1      DB 3
CT1PAR2      DD 4
  ENDS
 ENDS
ENDS
```

Члени об’єднань цього прикладу мають наступні параметри:

I’мя	Тип	Зміщення	Значення по замовчуванню
CTYPE	BYTE	0	?
CT0PAR1	WORD	1	1
CT0PAR2	BYTE	3	2
CT1PAR1	BYTE	1	3
CT1PAR2	DWORD	2	4

1	2	3	4	5	6
CTYPE	CT0PAR1 CT1PAR1	CT0PAR1 CT1PAR2	CT0PAR2 CT1PAR2	CT1PAR2	CT1PAR2

Об'єднання відрізняються від структур тим, що їх члени перекривають один одного. При ініціалізації даних типу об'єднання необхідно проявляти обережність, оскільки Асемблер дозволяє тільки одному члену примірника об'єднання мати ініціалізоване значення.

Наприклад:

```
BUNION {}
```

є допустимим, оскільки всі три члена об'єднання в оголошенні даних типу об'єднання були неініціалізованих. Цей оператор еквівалентний

```
DB 4 DUP(?)
```

В даному прикладі резервується 4 байта, так як розмір об'єднання визначається розміром максимального його члена (в даному випадку - подвійне слово).

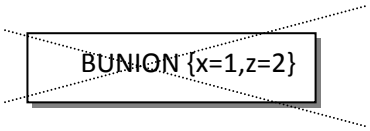
Наприклад:

```
BUNION {z=1}
```

еквівалентно

```
DB 1
DB 3 DUP(?)
```

Конструкція



Недопустима

Альтернативним методом ініціалізації примірника об'єднання є використання ініціалізатора з кутовими дужками (<>). Значення в Ініціалізатор такого виду не мають імен, тому вони повинні обов'язково слідувати в тому ж порядку, в якому описані відповідні члени об'єднання у визначенні структур або об'єднань. Ключове символ «?» вказує, що даний член структури буде неініціалізованим.

Наприклад:

```
ASTRUC <'abc',,?>
```

є еквівалентом

```
DB 'abc'
DW 1
DD ?
```

При вказанні ініціалізатора примірника об'єднання меншої кількості значень ініціалізації Ассемблер використовує для всіх членів, що залишилися примірника об'єднання ті значення, які були вказані при визначенні типу цієї об'єднання. Таким чином, замість **ASTRUC <'abc',,>** можна вказати **ASTRUC <'abc'>**.

Якщо застосовується ініціалізатор з кутовими дужками, то при ініціалізації вкладених об'єднань використовуються спеціальні угоди. Для кожного вкладеного рівня необхідно вказати додаткову пару кутових дужок. Наприклад, для типу даних **CUNION**:

```
CUNION <0,<<2,>,?>>
```

еквівалентно

```
DB 0
DW 2
DB 2
DB 2 DUP(?) ;щобзберегти правильний розмір об’єднання
```

Приклад: використання об'єднання

```
masm
model   small
stack   256
.586P
pntstruc        ;структура pnt, яка має вкладене об’єднання
        union           ;опис вкладеного в структуру об’єднання
                offs_16 dw      ?
                offs_32 dd      ?
        ends            ;кінець опису об’єднання
        segm    dw      ?
ends            ;кінець опису структури
.data
point   union   ;визначення  об’єднання, яке має вкладену структуру
        off_16  dw      ?
        off_32  dd      ?
        point_16        pnt     <>
        point_32        pnt     <>
point   ends
tst     db      “Рядок для тестування"
adr_data        point   <>      ;визначення екземпляра об'єднання
.code
main:
        mov     ax,@data
        mov     ds,ax
        mov     ax,segtst
;записати адресу сегмента рядка tst в поле структури adr_data
        mov     adr_data.point_16.segm,ax
;коли знадобиться, можно дістати значення з цього поля знов у регістр bx:
        mov     bx,adr_data.point_16.segm
;зформуємо зміщення в поле структури adr_data
        mov     ax,offsettst    ;зміщення рядка в ax
        mov     adr_data.point_16.offs_16,ax
;аналогічно, коли знадобиться, єзмога дістати значення з цього поля:
        mov     bx,adr_data.point_16.offs_16
        exit:
        mov     ax,4c00h
        int     21h
end     main
```

Як буде виглядати рядок, якщо необхідно segm = 18?

Коли ви будете працювати в захищеному режимі процесора і використовувати 32-розрядні адреси, то аналогічним методом можете заповнити та задіяти описане раніше об'єднання.

Лекція 16

Розділ 5. Багатомодульні програми

Велику програму недоцільно робити монолітною. Краще розділити її на кілька частин, або модулів і доручити їх виконання окремим програмістам.

Модуль - це частина програми, яка виконує певну підзадачі і більш-менш незалежна від інших частин. Окремий випадок модуля - *процедура*. Але до модуля можна включати і дані, типи та інше.

Традиційна схема - об'єднання модулів перед трансляцією. Вона зручна для невеликих програм. Внесення змін в окремий модуль тягне за собою перетрансляцію всієї програми. Тому для великих програм краще транслювати модулі окремо, а далі об'єднувати їх компонувальником в одну програму. Зміни в окремому модулі послужать причиною перетрансляції лише цього модуля.

Трансляція модуля здійснюється незалежно, тому адреси визначаються відносно початку цього модуля. Ясно, що всі локальні змінні отримують значення адреси і всі асемблерні команди з такими змінними перетворюються в машинні команди.

Але в модулі є ще й зовнішні імена, значення яких ще невідомі. Тому, створити машинні команди з такими іменами не можна. Отже, після трансляції модуля одержуємо, так званий, об'єктний файл (типу obj), де є і машинні команди і символьні.

Об'єднання, або компоновка модулів

Програма-компоновщик (linker) об'єднує модулі. При цьому вона розташовує модулі відповідно до послідовності викликів і перевіряє зовнішні імена. Отже, всі команди з зовнішніми іменами перетворюються в машинні. Головний модуль починається з нульової адреси, а інші - відповідно їх розташуванню. Адреси в допоміжних модулях перевизначаються в процесі компоновки. Отримуємо виконавчий файл типу EXE. Програма майже готова до виконання, але не прив'язана до конкретного місця в пам'яті.

При запуску програми викликається програма-завантажувач. Залежно від вільної пам'яті, вона завантажує програму в оперативну пам'ять (ОП), тобто, де треба перевизначає адреси в командах і передає управління програмі. Коли програма виконана, управління повертається операційній системі.

5.1. Структура модулів. Локалізація імен

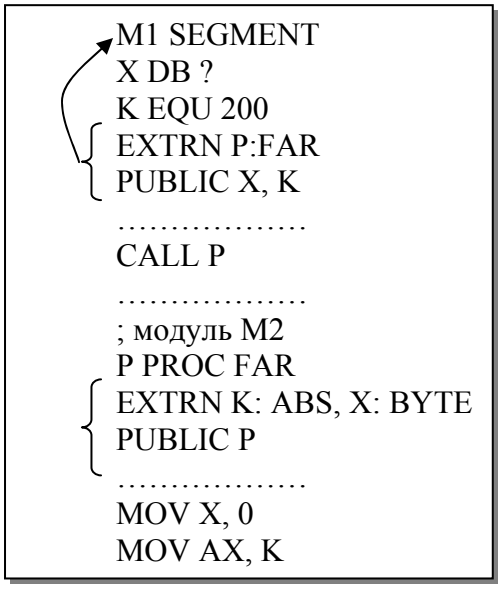
Кожен модуль закінчується директивою **END**. Серед них лише один головний, який закінчується **END [точка входу].**

Точка входу - це мітка команди, з якої починається виконання. З головного модуля починається компонування. Якщо ні в одному модулі не вказана точка входу, то компонування починається з першого за порядком модуля.

Всі імена в модулях мають *локальний* характер.

5.1.1. Зовнішні і загальні імена. Директиви EXTRN і PUBLIC

При взаємодії модулів виникають деякі проблеми. Нехай, є два модуля - M1 і M2:



Модулі транслюються окремо, тому при трансляції М1 нічого не відомо про процедуру Р. А при трансляції М2 нічого не відомо про іменах Х і К. Тому, транслятор виявить помилку: ***незнайдені імена***.

Отже, треба додатково повідомити транслятору, що зазначені імена знаходяться в інших модулях. Для цього існує директива:

EXTRN <ім'я>: тип, ... <ім'я>: тип

де тип - це слова **BYTE WORD, DWORD, ABS, NEAR, FAR.**

Три перших слова до даних, **ABS** - до констант, **NEAR**, **FAR** - до процедур.

Отже, в М1 треба включити:

```
EXTRN P:FAR
```

а в М2:

```
EXTRN K:ABS, X:BYTE
```

Проте, цього недостатньо. У модулях, де описуються зовнішні імена, необхідно вказати, що на ці імена будуть посилатися ззовні. Для цього існує директива:

PUBLIC <ім'я>, <ім'я>, ...

Типи відзначати не треба. Тому, в модулі М1 додається директива PUBLIC X, K, а в М2 - PUBLIC P.

Якщо цього не зробити, наприклад, не поставити в М2 директиву PUBLIC P, то хоча в модулі М1 буде EXTRN P, але після трансляції модуля М2 всі імені зникнуть і будуть лише адреси. Встановити, яку саме адресу мали на увазі буде неможливим. Отже, кожне ім'я, на яке посилаються з іншого модуля, має бути описано двічі: у своєму модулі в директиві PUBLIC, а в іншому - в директиві **EXTRN**.

5.1.2. Сегментування зовнішніх імен

У директиві **EXTRN** не відзначаються префікси сегментування регістрів. Але асемблер повинен знати за яким регістром сегментувати ім'я.

Наприклад:

```
MOV X, 0
; можна сприймати як MOV DS:X, 0 або MOV ES:X,0.
```

Для цього існує кілька угод.

Ясно що імена констант НЕ сегментуються. Для імені процедури дистанція визначається по її заголовку. Тобто в попередньому вигляді **CALL P** трактується як **CALL FAR PTR P.**

Для зовнішніх імен змінних діють такі правила:

1. якщо директива **EXTRN** розміщена поза сегментом то для кожної команди визначається регістр по замовчуванню: для даних **DS,** а при індексуванні за допомогою регістра **BP** визначається сегментний регістр **SS**.

2. якщо директива **EXTRN** розташована в програмному сегменті, то по замовчуванню імені сегментуються за тим же регістром який і всі імена з даного сегмента.

Наприклад:

```
 EXTRN X:WORD
A SEGMENT
EXTRN Y:WORD
INC X; INC DS:X
INC X [BP]; INC SS:X[BP]
INC Y; INC ES:Y
INC Y[BP]; INC ES:Y[BP]
A ENDS
B SEGMENT
EXTRN Z:WORD
INC Z; INC DS:Z
INC Z[BP]; INC DS:Z[BP]
B ENDS
ASSUME ES:A, DS:B
```

Хоча директиву **EXTRN** можна розміщувати довільно в програмі, від цього не залежить як будуть сегментуватися відповідні зовнішні імена.

5.1.3. Доступ до зовнішніх імен

Визначити, за яким сегментним регістром буде сегментуватися зовнішнє ім'я ще не все. Щоб команда з цим ім'ям була виконана правильно, треба, щоб до її виконання було правильно визначено в відповідний сегментний регістр.

Наприклад:

```
; M1
A SEGMENT
EXTRN X:WORD
Y DW 0
A ENDS
ASSUME DS:A
; M2
PUBLIC X
B SEGMENT
X DW ?
B ENDS
```

Припустимо, що регістр **DS** встановлено на початок сегмента A, і в модулі M1 необхідно виконати присвоєння X: = Y.

Це виконують команди:

```
MOV AX, Y
MOV X, AX
```

Перша команда виконається правильно **MOV AX, DS: Y**. Але друга - НЕВІРНО, так як транслятор буде трактувати її як **MOV DS: X, AX** і невідомо за якою адресою буде здійснена пересилання, так як **DS** не встановлено на сегмент B. Тому, щоб не змінювати регістр **DS**, можна використовувати регістр **ES** для сегментації сегмента **B**.

Отже, треба додати:

```
MOV BX, SEG X
MOV ES, BX
……
MOV AX, Y
MOV ES:X, AX
```

Оператор **SEG** визначає номер сегмента, де розташоване X, тобто його початкову адресу.

5.*2. Параметри директиви SEGMENT*

В команді-компонувальнику **LINK** відзначаються об'єктні модулі, які необхідно об'єднати. Компоновщик розміщує в пам'яті модулі, починаючи з головного, тобто, де є END START. А всі інші - в порядку їх перераховування.

Сегменти модулів розміщуються в послідовному порядку, один за одним. Але іноді такий порядок треба змінити. Для цього існують атрибути директиви SEGMENT:

Ім'я **SEGMENT [<вирівнювання>] [<об'єднання>] [<'клас'>]**

Ці атрибути можуть повністю або частково бути відсутніми, але порядок їх розташування змінювати не можна.

5.2.1. Атрибут 'клас'

Тут в апострофах відзначається довільне ім'я, яке є ім'ям класу, до якого відноситься сегмент. Що це дає?

Сегменти одного і того ж класу з різних модулів розміщуються поруч. Якщо сегмент відсутній атрибут класу, то за умовчанням цей клас вважається "порожнім" ім'ям.

Наприклад, є два модуля:

```
; M1
A SEGMENT 'DATA'
……
B SEGMENT
; M2
C SEGMENT 'DATA'
……
D SEGMENT 'CODE'
……
E SEGMENT
; программа
A 'DATA'
C 'DATA'
B
E
D
```

Спочатку розміщуються сегменти модуля, M1, а потім M2. Оскільки A - сегмент має клас 'DATA', то сегмент C буде розташовано після A. Оскільки сегмент B має порожній клас і сегмент E також, то сегмент E буде розташований після B, а вже потім - сегмент D.

5.2.2. Параметр об'єднання

За замовчуванням - **PRIVATE**. Атрибут об'єднання може мати значення **PUBLIC**, **STACK, AT, COMMON.**

Значення ***PUBLIC***

Як зазначалося, сегменти одного класу розміщуються поруч. Але при цьому вони незалежні, тобто, при використанні даних з певного сегменту на нього попередньо треба встановити відповідний сегментний регістр. Це потребує додаткових команд.

Але це можна спростити, об'єднавши сегменти одного класу і з однаковим ім'ям до одного. Тоді регістр сегменту треба встановити лише один раз, який забезпечить доступ до всіх даних сегмента.

Для цього треба використовувати значення **PUBLIC** атрибуту об'єднання. Але об'єднання здійсниться лише за трьох умов: всі сегменти мають одне ім'я, відносяться до одного класу і мають атрибут об'єднання **PUBLIC**.

Якщо хоча одна умова не виконується, то об'єднання не буде.

Приклад:

```
; M1
A SEGMENT PUBLIC 'R'
……….
B SEGMENT
; M2
C SEGMENT PUBLIC 'R'
……….
A SEGMENT PUBLIC 'R'
; M3
A SEGMENT 'R'
```

Відзначимо різницю понять: розмістити поруч і об'єднати. У першому випадку сегменти розміщуються поруч, але вважаються незалежними і на кожен сегмент треба встановлювати сегментний регістр. У другому це ніби окремі частини одного сегмента в різних модулях. При цьому сегментний регістр встановлюється один раз.

В одному і в іншому випадку сегменти розміщуються в пам'яті не щільно, байт за байтом, а з урахуванням вирівнювання. Тобто, наступна частина розміщується по замовчуванню на кордон параграфа, починаючи з адреси, кратної 16 (10h). Незалежно від того, де закінчилася попередня частина.

Значення ***STACK***

Використовується лише для сегментів стеку. Кілька сегментів стеку об'єднуються до одного, якщо мають одне ім'я, один клас і параметр об'єднання **STACK**. Такий об'єднаний сегмент розглядається як один стек і на нього автоматично встановлюються регістри **SS** і **SP**. Як зазначалося, значення **PUBLIC**, не пов'язане з автоматичною ініціалізацією регістра сегмента **DS** або **ES**.

Значення ***AT***

Має форму: **AT <конст_вираз>**

До значення константного виразу визначає номер сегмента пам'яті, тобто, абсолютну адресу без останніх чотирьох біт. Програмний сегмент з параметром **AT** ні з чим не об'єднується, а розміщується за цією адресою.

Наприклад:

```
VIDEO SEGMENT AT 0B800h
DW 25*80 DUP(?)
VIDEO ENDS
```

Сегмент довжиною 2000 слів буде розташований, починаючи з абсолютної адреси, 0B800h, тобто, накладений на відеопам'ять. За допомогою **AT**-сегментів можна отримати доступ до фіксованих ділянкам пам'яті (вектори переривань, відеопам'яті). При цьому змінювати вміст цих ділянок в **AT**-сегменті не можна, тобто, в цих сегментах не може бути команд і директив **DB**, **DW** з певними значеннями, а лише з (?).

Значення **COMMON**

Всі сегменти, які мають однакове ім'я, належать до одного класу і в директиві **SEGMENT** яких зазначений параметр **COMMON**, розміщуються компонувальником з однієї і тої ж початкової адреси. Це адреса, за якою розміщено перший сегмент. Отже, при цьому відбувається накладення сегментів і остаточний сегмент матиме розмір найбільшого.

Такі сегменти можна використовувати тоді, коли однакові осередки в різних модулях мають різні імена.

Наприклад:

```
; M1
S SEGMENT COMMON
A DB 1
B DD 0
S ENDS
; M2
S SEGMENT COMMON
X DB 3
Y DB ?
S ENDS
```

На обидва сегменти відводиться 5 байтів, перший байт має ім'я A у модулі M1, і ім'я X у модулі M2.

Параметр вирівнювання

Як зазначалося, по замовчуванню сегменти розміщуються з початку параграфа, тобто, з адреси, кратної 16. Щоб змінити цей порядок, можна використовувати такі значення:

BYTE - з найближчої вільної адреси;

WORD - з найближчого слова;

PARA - з початку параграфа;

PAGE - з найближчого початку сторінки, тобто, з адреси, кратної 256.

Отже, щоб розмістити сегменти щільно, треба використовувати значення BYTE.

Наприклад:

```
; M1
A SEGMENT PUBLIC 'Q'
X DW ?
Y DB ?
A ENDS
; M2
A SEGMENT BYTE PUBLIC 'Q'
Z DW ?
A ENDS
```

Буде створений загальний сегмент шляхом об’єднання сегмента A з модуля M2 без пропусків.

```
A SEGMENT
X DW ?
Y DB ?
Z DW ?
A ENDS
```

5.2.3. Додаткові директиви. Групи сегментів

Директива:

<ім'я_групи> GROUP <ім'я_сегмента> [, <ім'я_сегмента>] ...

Об'єднує зазначені сегменти в одну групу. Це означає, що всі імена цих сегментів будуть сегментуватися за одним і тим же регістром. За яким саме, визначить директива **ASSUME**, якщо в ній є операнд

SR: <ім'я_группи>

де **SR** - сегментний регістр.

Тоді кожне ім'я групи асемблер буде замінятися адресною парою **SR**: OFS, де OFS - змщення від початку групи. Така заміна здійснюється навіть тоді, коли для якогось сегменту вказано "свій" регістр. Відзначимо, що регістр SR треба прив'язувати до початку групи в програмі.

Об'єднання сегментів до групи означає лише їх однакове сегментування, а не безперервне розміщення в пам'яті. Розташування визначає параметр класу. Отже, сегменти однієї групи можуть бути розміщені нещільно, але відстань до останнього байта не може перевищувати 64 Кбайт.

Ім'я групи повинне бути унікальним, а сегменти можуть описуватися як перед, так і після директиви **GROUP**. Ім'я групи є константним виразом подібно імені сегмента. Наприклад:

```
GR GROUP S1, S2
S1 SEGMENT
A DB 20h DUP (0)
S1 ENDS
S2 SEGMENT
B DW 1
S2 ENDS
CODE SEGMENT
ASSUME CS:CODE, ES:GR
MOV AX, GR
MOV ES, AX
MOV DW, A; MOV DW, ...
MOV CX, B; MOV CX, ES
```

Існує деяка послідовність при використанні груп. У операторі **OFFSET** визначається зміщення не від початку групи, а від початку сегмента. Те ж саме маємо і при використанні адресних констант:

```
N_NEA DW N
N_FA DDN
```

Якщо треба зміщення від початку групи, то:

```
N_NEA DW GR:N
N_FA DD GR:N
```

5.3. Зміна поточної адреси

Під час трансляції асемблер визначає адресу кожної пропозиції програми. За цим стежить лічильник розміщення. Його символом в програмі є ім'я $.

Існують директиви зміни значення поточної адреси. Це директива **EVEN**, після якої наступну пропозицію буде розміщуватися обов'язково з першої парної адреси. Тобто, якби це мав бути непарний байт, то він пропускається і до нього записується команда **NOP**.

Директива **ORG <вираз>**

У цьому виразі може бути і знак поточної адреси $.
Наприклад:

```
ORG 100h; пропустить 256 байт часто використовується на початку COM-файлів.
ORG $+30; пропустить 30 байт
```

5.4. Директива LABEL

<ім'я> LABEL <тип>

Визначає ім'я, яке приписується поточною адресою ($), і зазначений тип. Ім'я повинно бути унікальним. Значення типів **BYTE, WORD, NEAR, FAR.**

Використовується для того, щоб на наступну за директивою команду або змінну можна було б посилатись за новим ім'ям з іншим типом, ніж це витікає з контексту програми.

Наприклад:

```
FL LABEL FAR
NL: MOV AX, 0
```

FL і NL - це одна і та сама адреса, тільки **NL - NEAR, а FL - FAR.**

5.5. Зміна системи обчислення констант

Вище зазначалося, що локально можна задавати системи обчислення констант 1011B, 0F23h, 15D, 2560. За замовчуванням константи вважаються десятковими. Але можна змінювати системи обчислення констант і на цілий фрагмент програми за допомогою директиви.

RADIX <константний_вираз>

Наприклад:

```
RADIX 8
```

Встановлюється вісімкова система обчислення для констант литералів. Область дії - до наступної директиви **RADIX**, або до кінця програми. У директиві можна вказати константи 2, 8, 10, 16.

5.6. Використання команд інших процесорів

За замовчуванням макроасемблер MASM дозволяє використовувати команди процесора 8086. Але в кожному подальшому з типів процесорів і співпроцесорів існують свої додаткові команди. Тому, для можливості їх використання попередньо треба навести попередню директиву .186; команди 80186 .2860; додаткові команди 286 непривілейованого режиму; .286P - теж непривілейованого режиму; .8087 - співпроцесора 8087 .287 - співпроцесора 80287.

Лекція 17

5.2 Особливості роботи із зовнішніми пристроями

Внутрішніми пристроями ПЕОМ є центральний процесор та оперативна пам'ять. Всі інші являються зовнішніми. Тому під введенням/виведенням в широкому розумінні мають на увазі обмін інформацією між ЦП та будь-яким зовнішнім пристроєм.

Як зазначалося передача інформації між ЦП та пристроями відбувається через порти. Порт – це регістр розміром з байт чи слово, що знаходиться поза ЦП. Порти нумеруються від 0 до 0FFFFh. Адже може бути 65536 портів, тим не менше, реально їх 256.

З кожним пристроєм зв'язані один чи кілька портів, їх номери відомі. Команди обміну з ЦП:

IN AL, n

OUT n, AL

ЦП обмінюється з портами не лише інформацією, а і керуючими сигналами. Тому порти бувають двох типів:

1. Порти, з яких передаються дані – інформаційні порти
2. Порти, з яких поступають сигнали керування – керуючі порти

Пристрої бувають різні.

Повільнодіючі: клавіатура, монітор, мишка.

Швидкодіючі: гнучкі й жорсткі диски. Послідовність обміну сигналами та інформацією дуже відрізняється.

Типовою схемою обміну може бути така. Перед початком обміну з зовнішнім пристроєм ЦП записує в порт управління цього пристрою певну комбінацію біт, яка допомагає встановити зв'язок з пристроєм. Якщо пристрій в робочому стані і не зайнятий, він записує іншу кобінацію-відповідь, яка вказує на готовність до обміну. Після деякої паузи ЦП зчитує комбінацію-відповідь пристрою і якщо, там комбінація про встановлення зв'язку, то починається обмін.

Якщо ЦП хоче щось вивести на зовнішній пристрій, тоді до порту управління записує команду виведення , а до інформаційного порту – код символу. Зовнішній пристрій зчитує цю інформацію, встановлює сигнал «зайнято» і виконує команду.

Якщо під час виконання програми щось зіпсувалося, то пристрій встановлює сигнал «зіпсовано», інакше після вдалого виконання встановлюється сигнал «вільно». Далі все повторюється знову. Обмін з зовнішніми пристроями виконується в двох режимах:

1. очікування;
2. з перериванням.

В першому режимі під час виконання обміну процесор нічого не виконує, а лише перевіряє закінчення команди обміну. В другому – вдає команду на початок обміну і переходить до виконання іншої програми, дозволивши переривання зовнішньому пристрою. Ефективність кожного режиму залежить від частоти переривань.

Тож, програми обміну досить складні, їх написання потребує знань особливостей роботи пристроїв, структуру портів обміну. Тому для стандартних пристроїв ці програми написані у вигляді процедур і їх можна використовувати в програмах. Доцільно писати програми обміну лише для унікальних пристроїв.

Переривання. Функції DOS

Переривання використовують для спостереження за зовнішніми подіями, наприклад, при натисканні на кнопки. Їх не можна використовувати для спостережння за внутрішніми подіями, наприклад, ділення на 0. Переривання використовують для реалізації обміну з зовнішніми пристроями.

Процедури обміну написані та включені до ОС. Можна було б застосувати команду CALL і вказати адресу. Але адреса цих процедур змінюється в залежності від ОС. Тому окремі номери переривань призначені для виклику процедур ОС. Для відповідних векторів переривань записані початкові адреси процедур. Тому достатньо знати не адресу процедури, а номер їх вектора переривань, який однаковий для всіх версій ОС.

В склад ОС входить багато процедур і для всіх не вистачає допустимих номерів переривань. Тому процедури об'єднали в групи, які викликаються за одним номером переривання.

Процедури однієї групи називають функціями відповідного переривання. Щоб їх розрізнити, перед виконанням команди **INT** до регістру **AH** необхідно записати номер потрібної функції.

MOV AH, <номер функції>
INT 21H

Для виконання деяких функцій потрібні початкові параметри, наприклад, адреса, з якої здійснюється виведення. Ці параметри передаються через відповідні регістри. Для кожної функції є свої параметри.

Вектори переривань DOS

20H – закінчення програми;
21H – виклик функції;
22H – передача управління за адресою PSP DWORD PTR [0ah];
23H – адреса CTRL-Break;
24H – опрацювання непоправної помилки;
25H – режим зчитування із диску;
26H – режим запису на диск;
27H – закінчення програми без звільнення пам'яті.
Вектори 28H - 3FH зарезервовані для DOS.

Вектори 22h, 23h, 24h є покажчиками програм реакції на відповідні ситуації – кінець програми, натиснути кнопки Ctrl-Break виникнення непоправної помилки. Звична реакція на комбінацію Ctrl-Break – це зупинка. При виникненні непоправної помилки здійснюється виведення причини такої ситуації. Якщо ж необхідно інакше реагувати на ці події, то треба написати програму та змінити значення потрібних векторів переривань.

Переривання 25h здійснює зчитування, а 26h – запис інформації на диск з абсолютною адресацією. При цьому забезпечується доступ до визначених ділянок диску, а не до компонентів у файлі.

Переривання 20h і 27h повертають керування програмою до DOS. Перше відповідає нормальному закінченню програми зі звільненням пам'яті, а друге – без звільнення. Програма залишається в пам'яті до перезавантаження системи або виключення ПЕОМ.

Введення з клавіатури

При натисненні кнопки код символу записується до спеціального системного буферу, розмір якого - 15 символів. З цього буферу функції введення зчитують символи з початку буфера.

Є декілька фукцій введення. Розгляемо одну із них. 0Ah (10) переривання 21h, з допомогою якого можна ввести рядок і відредагувати текст.

Адреса для запису рядка повинна бути в регістрах **DS:DX**

MOV AH, 0Ah

INT 21h

Функція виводить символи і записує їх у буфер, поки не натиснуто кнопку ENTER. При цьому символи висвічуються на екрані (“exo”). Поки не натиснута кнопка Enter, текст можна редагувати з допомогою кнопок BackSpace – знищує останній символ та ESC – знищує весь набраний текст.

Буфер, в який записується текст, повинен мати структуру:

Max	**n**	**s1**	**s2**	**…**	**sn**	**CR**
1	2	3			n+2	max+2

В першому байті повинна записуватися максимальна кількість символів в рядку max. Більше за цю кількість в рядок записати не можна. В другому байті функція сама запише поточну кількість символів n <= max. Після останнього символу функція запише символ кінця рядка CR (код 13), який відповідає кнопці Enter, і який не рахується до загальної кількості символів рядка.

Наприклад:

```
BUF DB 10, ?, 10 DUP (' ')     сегмент данних
……….....
LEA DX, BUF
MOV AH, 0Ah
INT 21h
```

Якщо ввести символи 'ABCD', то в буфері буде:

BUF[1] = 10
BUF[2] = 4
BUF[3] = 65
BUF[4] = 66
BUF[5] = 67
BUF[6] = 68
BUF[7] = 13

На основі цієї функції можна реалізувати функцію введення цілого числа.

Заміна векторів переривань

За необхідності вектори переривань можна замінити.
Послідовність дій буде такою:

1. написати свою процедуру обробки переривань;
2. запам'ятати початкове значеня вектора переривань в області даних;
3. встановити в таблиці повний вектор переривань;
4. перед закінченням програми виконати початковий вектор переривань.

Для роботи з векторами переривань існують функції переривань 21h:
35h – отримати значення вектора переривань;
25h – встановити нове значення вектора переривань.

Функція 35h

Номер вектора переривань, значення якого хочемо отримати, треба занести до регістру **AL**. Функція передає значення вектору переривань в регістрах **ES:BX**.

Тож, виклик функції може бути таким:

```
PUSH ES; захистити вектор переривань від зміни функцією 35h
MOV AL, 20h
MOV AH, 35h
INT 21h
; пересилання значення виразу
MOV OLD_CS, ES
MOV OLD_IP, BX
POP ES
```

Функція 25h

Значення нового вектору переривань повинне записуватися в пари регістрів **DS:DX**. Значення вектора, яке ми хочемо отримати потрібно записати до **AL**.

Тож, якщо хочемо встановити нове значення вектору переривань, то послідовність може бути такою.

Наприклад, процедура опрацювання переривань називається OUR_INT:

```
PUSH DS
MOV DX, offset our_int
MOV AX, seg our_int
MOV DS, AX
MOV AL, 20h; нове значення вектору 20h
MOV AH, 25h
INT 21h
POP DS
```

Тож, структура програми зі зміною вектору переривань:

```
;заміняємо вектор 20h
Data_seg segment
……
old_cs dw 0
old_ip dw 0
data_seg ends
cod_seg segment 'code'
; запам'ятовуємо старий вектор
PUSH ES
MOV AL, 20h
MOV AH, 35h
INT 21h
MOV old_cs, ES
MOV old_ip, BX
POP ES
; встановлюємо новий вектор
CLI; заборона маскування переривань
```

```
PUSH DS; захист DS
MOV DX, offset our_int
MOV AX, seg our_int
MOV DS, AX
MOV AL, 20h; номер вектора, який заміняємо
MOV AH, 25h
INT 21h
POP DS
STI; дозволити маскування переривань
; програма
; відновлення старого вектора
 CLI; заборона переривань
PUSH DS
MOV DX, old_ip
MOV AX, old_cs
MOV DS, AX
MOV AL, 20h
MOV AH, 25h
INT 21h
POP DS
STI
…….....
our_int proc far
<тіло>
IRET
our_int endp
cod_seg ends
end <точка входу>
VST_VEC MACRO offs, segm numvec
CLI
PUSH DS
MOV DX, offs
MOV AX, segm
MOV DS, AX
MOV AL, numvec
MOV AH, 25h
INT 21h
POP DS
STI
ENDM
```

Аналогічно можна доповнити існування процедури DOS'у опрацювання переривань. Для цього значення існуючого вектора переривань треба перемістити на місце вектора користувача, наприклад, 60h.

Написати процедуру доповнення, адреси, яка записана на місце старого вектору переривань. Із процедури доповнення викликати стару процедуру int 60h під новим номером.

Послідовність дій буде такою. Наприклад, хочемо доповнити процедуру обробки переривань 16h.

1. Написати процедуру додаткових дій;
2. За допомогою функції 35h переривання 21h отримати значення старого вектора переривання old_cs і old_ip.

3. За допомогою функції 25h встановити нове значення в новий вектор переривань, наприклад, 60h. Це переривання треба викликати перед **IRET** додаткової процедури.
4. За допомогою функції 25h встановити нове значення вектора переривань 16h.
5. Нову процедуру викликати через вектор переривань 16h.
6. Після закінчення програми функцією 25h відновити старі значення вектора переривань 16h.

Лекція 18

5.3. Особливості структури та виконання .COM і .EXE програм

До цього часу ми виконували програми з багатосегментною структурою, з яких створювалися .EXE - файли.

Але короткі програми можна реалізовувати в односегментної вигляді, тобто, як .COM - файли (компактні).

У цих файлах є лише один сегмент - коду, в якому розміщуються також дані. Сегмента стеку немає. Ясно, що обсяг такої програми не може перевищувати 64Кбайт.

Структура .COM-програми може бути такою:

```
CODE SEGMENT
ASSUME CS: CODE, DS: CODE, SS: CODE, ES: CODE
ORG 100h; резерв
START: JMP Begin; обійти опис даних
A DW 15
B DB 'оголошення'
MAS DB 20 DUP (?)
S DD?
Begin Proc Far
<тіло програми>
RET
Begin ENDP
CODE ENDS
END START
```

У вигляді .COM - файлів виконуються програми драйверів (управління пристроями), та для обробки переривань.

Програмний префікс

З точки зору користувача, програма виконує деякі обчислення, а з точки зору операційної системи - це деякий процес (робота), яку треба виконати. Тому для реалізації програми її треба доповнити додатковою інформацією для виконання конкурентного процесу. Ця інформація зосереджується в програмному префіксі (Program Segment Prefics) PSP, який займає 256 байт (100h).

В PSP записано:

Зміщення			**байти**
0H			2
2H			2
4H			6
0AH			4
0EH			4
12H			4
16H			22
20H			2
2EH			46
50H			16
60H			16
80H			128

INT 20h, розмір доступної пам'яті. Адреси початку обробки переривання **22h, 23h, 24h** зарезервовано.

Завантаження .COM-файлів до пам'яті

Як бачимо, .COM-файл майже повністю готовий до виконання, тільки в ньому не вистачає програмного префікса PSP.

Тому при завантаженні .COM-файлу з диску виконуються такі дії:

1. створюється **PSP**-префікс програмного сегменту, який і записується в перші 256 байт (100h);

2. відразу ж за **PSP** повністю копіюється програма;

3. всі чотири сегментні регістри (**CS**, **DS**, **ES** і **SS**) встановлюються на початок **PSP**;

4. регістр **IP** встановлюється на 100h-початок .COM-програми, регістр **SP** на кінець сегменту на адресу FFFE, куди записується нуль-адреса першого слова **PSP** - адреса повернення;

5. починається виконання програми.

Структура .EXE-програми

Як зазначалося, ця структура використовується для багатосегментніх програм. Після компоновки окремі сегменти створені, але всі адреси відраховуються там від 0-умовного адреси. Коли ж сегмент отримає іншу початкову адресу, то треба перерахувати і ці адреси. Отже, під час завантаження відбувається переміщення. Машинні команди, які треба перерахувати, позначаються в файлі лістингу буквою r (relocation). Наприклад, всі команди з прямою адресацією **MOV AX, TABLE**.

Для того, щоб здійснити такі переміщення, компоновник створює таблицю переміщенньь (relocation table). Ця таблиця розміщується в заголовку .EXE-файла. Заголовок займає не менше 512 байт.

Оскільки .EXE-файл не текстовий, то безпосередньо прочитати заголовок не можна. Фірма Microsoft створила спеціальну програму EXEMOD, яка дає можливість прочитати деяку інформацію з заголовку.

EXEMOD WRITE STR	hex	du
.EXE size (bytes)	290	656
Minimum local size (bytes)	90	144
Overlay number	0	0
Initial CS:IP	0000.0000	
Initial SS:SP	0004.0050	80
Minimum allocation	0	0
Maximum allocation	FFFF	05535
Header size	20	32
Relocation table offset	1E	30
Relocation entries	1	1

При завантаженні .EXE-файлу до пам'яті DOS виконує такі дії:

1. створює **PSP**;

2. переглядає заголовок і визначає, де закінчується і починається програма; починає завантажувати програму до пам'яті;

3. використовуючи інформацію заголовку, DOS знаходить і виправляє все переміщувані посилання;

4. встановлює регістри **ES** і **DS** на початок **PSP**.

Тому якщо в програмі є свій сегмент даних і додатковий, то в програмі треба встановити **ES** і **DS** на відповідні сегменти.

5. встановлює регістр **CS** на початок сегменту коду, а **IP** - на точку входу.

Пара регістрів **SS**: **SP** встановлює відповідно інформації в заголовку.

Наприклад, з попередньої таблиці SS буде встановлено 4 параграфа після **PSP**, а **SP** в 0050.

В.EXE-програмах є певні проблеми з використанням переривання 20h - закінчення програми. Це переривання потребує, щоб при його використанні регістр **CS** було встановлено

на початок **PSP**. В COM-програмах це виконується, а у .EXE-програмах **CS** встановлено на сегмент коду. Тому встановлення **CS** на початок **PSP** робиться в програмі, створюється в стеку адреса повернення 0000: 0000. Оскільки **DS** встановлюється на початок **PSP**, то перед його встановленням на сегмент даних засилаємо значення **DS** до стека:

```
PUSH DS
XOR AX, AX
PUSH AX
```

А потім засилаємо до стека нуль. При виконанні команди RET до **CS** і буде переписано адресою повернення 0000: 0000.

Системні засоби розподілу пам'яті

Використовуючи системні засоби, можна з однієї програми викликати іншу - програму-нащадок. Наприклад, в Turbo Pascal є директива Exec (Path, CmdLine: string), де Path - це скорочений або повний шлях до виконуваного файлу і його імені; CmdLine - параметр, за допомогою якого можна задавати до викликаної програмі командна рядок.

Але для того, щоб таке можна було зробити, треба врахувати особливості розподілу пам'яті в MS DOS. Як відомо, по замовчуванню встановлені такі параметри директиви:

{$ M 16384, 0, 0},

тобто, всі 640К динамічної пам'яті відведені основній програмі. Тому для програми-нащадка пам'ять зовсім не виділена.

Отже, якщо хочемо викликати інші програми, то треба встановити директиву:

{$ M 1024, 0, 0}.

Для динамічного управління пам'яттю існують спеціальні функції MS DOS.

48H - виділити пам'ять;

49H - звільнити пам'ять;

4AH - змінити розмір пам'яті.

Функція 48h - до регістру **BX** треба занести обсяг потрібної пам'яті в параграфах (кратне 16). А до **AH** - номер функції. Якщо пам'ять є і вона виділена, то **CF** = 0 і до **AX** записується сегментна адреса початку потрібної ділянки пам'яті.

Інакше **CF** = 1 - помилка, а до **AX** запишеться код помилки:

7 - коли зруйнований блок управління пам'яттю;

8 - немає вільного об'єму пам'яті, при цьому до **BX** запишеться обсяг найбільшого вільної ділянки пам'яті в параграфах.

Приклад виклику функції 48h.

Нехай, в **BX** зформовано обсяг потрібної пам'яті в байтах. тоді:

```
MOV CL, 4
SHR BX, CL
INC BX; додаємо 1 $ ддя закруглення, оскільки мл. байт неповний $
MOV AH, 48H
INT 21H
```

Функція 49h - звільнення динамічної пам'яті, яка була попередньо виділена функцією 48h.

Параметри функції: в **AH**-номер, в **ES** - сегментна адреса пам'яті, яку треба звільнити.

Якщо звільнення відбулося правильно, то **CF** = 0.

Інакше - **CF** = 1, в регістрі **AH** встановлюється код помилки:

7 - зруйнований блок управління пам'яттю;

9 - неправильний сегментна адреса, тобто пам'ять функцією 48h не виділялася.

Функція **4AH** - зміна розміру пам'яті.

Параметри: **AH** - номер функції, **ES** - сегментна адреса тієї пам'яті, розмір якої хочемо змінити, **BX** - повний обсяг пам'яттю в параграфах. Якщо пам'ять змінена, то **CF** = 0.

Інакше **CF** = 1 і в регістрі **AX** встановлюється код помилки:

7 - зруйнований блок управління пам'яттю;

8 - немає пам'яті потрібного обсягу, до **BX** - наявний розмір пам'яті;

9 - неправильний сегментна адреса.

Функцію **4AH** можна застосовувати до зменшення обсягу пам'яті під програму до мінімально необхідного.

Для того, щоб скористатися функцією **4AH** треба визначити розмір пам'яті під програму.

Це робиться по-різному для .EXE і .COM - файлів.

Визначення розміру .EXE-програми

Програма складається з декількох сегментів: стеку, даних, коду програмного префікса, довжиною 100h байтів. З попереднього знаємо, що виконавча система встановлює регістри **DS** і **ES** на початок програмного префікса **PSP**. Отже, номер сегменту початку програми знаходиться в **ES**.

Для того, щоб визначити сегментна адреса кінця програми перед заключним рядком end точка_входу розмістимо порожній сегмент zzz.

```
Data_seg segment
<дані>
data_seg ends
st_seg segment stack
dw 128 dup (?)
st_seg ends
cod_segment 'code'
assume ds: data_seg, cs: cod_seg, ss: stseg
main proc far
; команди зміни обсягу пам'яті під програму
mov ax, zzz
mov bx, es
sub ax, bx
mov bx, ax
mov ah, 4ah
int 21h
.............
ret
main endp
cod_seg ends
zzz segment
zzz ends
end main
```

Визначення довжини .COM-програми

Така програма складається лише з одного сегменту. Тут використовується лише зміщення, сегментні адреси (номери сегменту) не потрібні. Тому довжина програми буде в байтах. Початкова адреса - це точка входу. Кінцеву адресу зафіксуємо:

Last equ $

Як відомо, покажчик стеку встановлюється на кінець сегменту. Тому при зменшенні програми треба зарезервувати пам'ять під стек, і в програмі встановити значення **SP**.

```
cod segment 'code'
assume cs cod, ss: cod, ds: cod
org 100h
main proc
jmp eb
<дані>
  eb: ... ... ....
mov sp, offset last
<фрагмент зменшення пам'яті>
main endp
dw 128 dup (?)
last equ $
cod ends
end main
```

Як визначити потрібний розмір пам'яті під програму в байтах? Це буде кінцева адреса - початкова + 100h + 15. Потрібна кількість параграфів В/16.

Отже, програма буде мати структуру:

Фрагмент зменшення пам'яті:

```
MOV BX, (last - main + 10FH) / 16
MOV AH, 4AH
```

До цього часу дані в програмі задавалися статично за допомогою директив **DW**, **DB**. Але, використовуючи функції 48h і 49h можна використовувати динамічну пам'ять. Наприклад, нехай в регістрі **AX** записана потрібна кількість байт. Тоді виділимо під них пам'ять:

```
MOV BX, AX
MOV CL, 4
SHR BX, CL
INC BX; потрібна кількість параграфів
MOV AH, 48H
INT 21H; в AX - сегментна адреса виділеної пам'яті
Доступ до виділеної пам'яті можна реалізувати за допомогою індексного регістра, наприклад, [SI]
PUSH ES
MOV ES, AX
..... .....
MOV ES: [SI], DATA; в SI створюємо відповідне змінне
Звільнення динамічної пам'яті
; в ES сегментна адреса
MOV AH, 49H
INT 21H
POP ES
```

Програми, резидентні в пам'яті

Це такі програми, які знаходяться в пам'яті постійно і негайно реагують на запити, або на певні події в системі.

Резидентними є більшість функцій ОС, драйвери і процедури обробки переривань.

Але користувач може сам создать резидентні програми як у вигляді .EXE, так і .COM - файла. Найчастіше - у вигляді .COM-файла.

Така програма повинна складатися їз двох частин:

1. резидентної;

2. ініціалізації.

Під час першого запуску програма управління передається ініціалізуючій частини, яка налаштовує відповідним чином резидентну частину. Після цього виконання програми закінчується, а резидентная частина залишається в пам'яті.

Ініцалізуюча частина закінчує роботу викликом функції **31h** переривання **21h**. Перед викликом цієї функції в **DX** треба занести обсяг резидентної частини в параграфах.

Структура .COM-програми:

```
cod segment 'code'
assume cs: cod, ds: cod, ss: cod
org 100h
start jmp init
; резидентна частина
; дані резидентної частини
...... dw ......
...... db .......
entry proc far
; оператори резідентної частини
entry endp
memsize equ $; кінцева адреса резидентної частини
; частина ініціалізації
init proc
; оператори налаштування резидентної частини
mov dx, (memsize - start + 10FH) / 16
mov al, 0
mov ah, 31h; функція завершена
int 21h
init endp
cod ends
end start
```

Приклад: резидентну частину можна активізувати як процедуру обробки переривання. Для цього можна використовувати вільні вектори переривання, наприклад, **60h**.

В ініціалізуючій частини треба записати початкові адреси резидентної частини за допомогою функції **25h**. Оператори налаштовування резидентної частини будуть такі:

```
MOV AL, 60h
Lea DX, entry; регістр DS містить сегментну адресу
MOV AH, 25H
INT 21h; встановлення вектора 60h на резидентну частину
```

Зрозуміло, що резидентна частина повинна завершуватися **IRET**. Виклик резидентної частини з будь-якої програми за допомогою команди **INT 60h**.