О.І.Болдаков, Г.В.Красовська, В.Г.Голенков, К.М.Красовський

ОПЕРАЦІЙНІ СИСТЕМИ ТА СИСТЕМНЕ ПРОГРАМУВАННЯ

Мова програмування Асемблер. Конспект лекцій

ЗМІСТ

ВСТУП. ІСТОРІЯ ВИНИКНЕННЯ КОМП'ЮТЕРА 7

Принцип Фон-Неймана 7
Як виникли персональні комп'ютери (ПК) 9
Принцип відкритої архітектури (ВА) 10

ЕМА 1. АРХІТЕКТУРА ПЕРСОНАЛЬНОГО КОМП'ЮТЕРА СІМЕЙСТВА INTEL 11

ТЕМА 2. ОДИНИЦІ ВИМІРУ ІНФОРМАЦІЇ ТА СИСТЕМИ ЧИСЛЕННЯ 13

Перетворення чисел з однієї системи числення до іншої 14
Арифметичні операції над двійковими числами 16
Подання від'ємних чисел (додатковий код) 17
ASCII коди та двійково-кодовані десяткові (BCD) числа 18

ТЕМА 3. АРХІТЕКТУРА ПРОЦЕСОРА СІМЕЙСТВА INTEL 20

Регістри процесору І486(286). Регістри загального призначення та індексні 22
Сегментні регістри та адресація оперативної пам'яті 23
Регістри зміщення 27
Організація стеку [10] 27
Регістр прапорів [10] 28
Машинні мови і формати команд 30

ТЕМА 4. РЕЖИМИ АДРЕСАЦІЇ 31

1. Регістрова адресація 31
2. Безпосередня 31

3. Пряма 31
4. Непряма адресація (рус. - косвенная) 32
5. Непряма по базі (базова зі зсувом) 32
6. Пряма з індексуванням 33
7. Пряма з індексуванням та масштабуванням 33
8. Адресація по базі з індексуванням 33

ТЕМА 5. КОМАНДИ ПРОЦЕСОРІВ X86 34

1. КОМАНДИ ПЕРЕСИЛАННЯ ДАНИХ 34
1. Команда **MOV** 34
2. Команди обміну даними **XCHG** 35
2. Команда завантаження ефективної адреси **LEA** 35
3. Команда **XLAT** 35
4. Команда пересилання даних у стек PUSH 36
5. Команда пересилання даних зі стеку **POP** 36
6. Команда зчитування даних з порту **IN** 36
7. Команда запису даних в порт **OUT** 36
8. Команди конвертування 36
3. АРИФМЕТИЧНІ КОМАНДИ 37
1. Команди додавання 37
2. Команди віднімання 38
3. Команда порівняння 38
4. Команди множення 39
5. Команди ділення 39
4. КОМАНДИ ОБРОБКИ БІТІВ 40
5. Операції зсуву 41
6. Операції циклічного зсуву 41
7. Команди передачі управління 41
8. Команди умовного переходу 43
9. Команда організації циклу 44
10. Команди роботи з рядками символів 44
1. Команди копіювання рядків 45
2. Команди порівняння рядків 45
3. Команди сканування рядків 46
4. Команди читання з рядка 46
5. Команди запису в рядок 46
6. Команди зчитування з порту 46
7. Команди запису в порт 46

11. КОМАНДИ УПРАВЛІННЯ ПРАПОРАМИ 47

ТЕМА 6. ПРИЗНАЧЕННЯ ТА ВИКОРИСТАННЯ СПІВПРОЦЕСОРА 47

ЧИСЛА З ПЛАВАЮЧОЮ КРАПКОЮ ТА ТИПИ ДАНИХ FPU 47
ФОРМАТИ ДІЙСНИХ ЧИСЕЛ З ПЛАВАЮЧОЮ КРАПКОЮ, ВИКОРИСТОВУВАНІ В ПРОЦЕСОРАХ INTEL 49
АРХІТЕКТУРА СПІВПРОЦЕСОРА 51
КОМАНДИ FPU 53
1. ОСОБЛИВОСТІ ФОРМУВАННЯ КОМАНД 53
2. КОМАНДИ ПЕРЕСИЛКИ ДАНИХ FPU 54
3. АРИФМЕТИЧНІ ОПЕРАЦІЇ 55
4. ОПЕРАЦІЇ ПОРІВНЯННЯ 56
5. ТРАНСЦЕНДЕНТНІ ОПЕРАЦІЇ 57

ТЕМА 7. СТРУКТУРА ПРОГРАМИ МОВОЮ АСЕМБЛЕР 58

ОСНОВИ ПРОГРАМУВАННЯ В MS DOS 58
ПРОГРАМА ТИПУ EXE 59
СТРУКТУРА EXE ПРОГРАМИ (ПРИКЛАД 1) 60
ОПИС СЕГМЕНТІВ ТА КОМАНДА ASSUME 60
КІНЕЦЬ ПРОГРАМИ 61
КОМЕНТАРІ У ПРОГРАМАХ 62
МОДЕЛІ ПАМ'ЯТІ ТА СПРОЩЕНІ ДИРЕКТИВИ ОПИСУ СЕГМЕНТІВ 62
СТРУКТУРА EXE ПРОГРАМИ (ПРИКЛАД 2) 63
ПРОГРАМА ТИПУ COM 64
ДИРЕКТИВА УПРАВЛІННЯ ПРОГРАМНИМ РАХІВНИКОМ 64
СТРУКТУРА COM ПРОГРАМИ (ПРИКЛАД 1) 64
СТРУКТУРА COM ПРОГРАМИ (ПРИКЛАД) 65
ВИСНОВКИ ЗА ТЕМОЮ: 65

ТЕМА 8. ОРГАНІЗАЦІЯ ПІДПРОГРАМ У ПРОГРАМАХ МОВОЮ АСЕМБЛЕР 66

СИНТАКСИС ОПИСУ ПІДПРОГРАМИ 66
ПЕРЕДАЧА ПАРАМЕТРІВ У ПІДПРОГРАМУ 67
СТВОРЕННЯ ЛОКАЛЬНИХ ЗМІННИХ У ПІДПРОГРАМАХ 69
ГЛОБАЛЬНІ ОГОЛОШЕННЯ ТА ЗАГАЛЬНІ ДАНІ У ПРОГРАМАХ 69

СПИСОК ЛІТЕРАТУРИ 72

Вступ. Історія виникнення комп'ютера

Прямий переклад слова комп'ютер - обчислювач. У 1642 Блез Паскаль винайшов пристрій, що механічно виконує додавання чисел. Можна сказати, що це був прапрадід першої обчислювальної машини. У 1673 Лейбниц створив арифмометр - пристрій, що виконує вже чотири дії – додавання, віднімання, множення, ділення. У першій половині XIX сторіччя англійський математик Чарльз Бэббидж спробував побудувати універсальний обчислювальний пристрій – аналітичну машину, що виконує обчислення без участі людини. Вона повинна була вміти обробляти програми, що вводяться за допомогою перфокарт і мати “склад” для запам'ятовування даних і проміжних результатів (тобто пам'ять). Практично реалізувати свої ідеї Чарльз Беббидж не зміг - підвели технічні можливості його часу.

Тільки на базі техніки XX сторіччя ідеї Чарльза Беббиджа втілилися у життя. Використовуючи електромеханічні реле, у 1943 році американець Говард Эйкен розробив комп'ютер Марк-1.

Ще раніше ідеї Беббиджа були перевідкриті німецьким інженером Конрадом Цузе, що у 1941 році побудував аналогічну машину.

Починаючи з 1943 року група спеціалістів під керівництвом Джона Мочли і Преспера Экерта в США почала конструювати подібну машину вже на основі використання електронних ламп, а не реле. Їхня машина, названа ENIAC, працювала в тисячу разом швидше, ніж Марк-1, проте для завдання її програми приходилось на протязі декількох часів або навіть декількох днів під'єднувати необхідні дроти.

Принцип Фон-Неймана

У 1945 році до робіт по створенню комп'ютера був залучений знаменитий математик Джон фон Нейман. Його доповідь на цю тему одержала широку популярність, оскільки в ній фон Нейман ясно і просто сформулював загальні принципи функціонування універсальних обчислювальних пристроїв, тобто комп'ютерів.

Фон Нейман запропонував такі складові пристрої комп'ютера (рис.1):

- арифметико-логічний пристрій (АЛП), що виконує арифметичні і логічні операції;
- керуючий пристрій (КП), що організує процес виконання програми;
- пристрій, що запам'ятовує (ЗапП) або пам'ять для збереження програм і даних;
- зовнішній пристрій (ЗП) для введення/ виведення інформації.

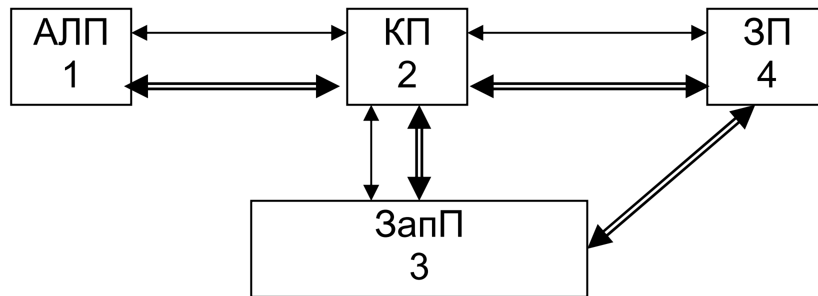

Рис.1. Машина Фон Неймана

Пам'ять комп'ютера повинна складатися з певної кількості пронумерованих комірок, у кожній із яких можуть знаходитися і оброблятися дані або інструкції програм. Всі комірки пам'яті повинні бути однаково доступні для інших пристроїв комп'ютера.

Далі фон Нейман запропонував зв'язки між пристроями комп'ютера, що відображені на рис. 1: одинарні лінії - керуючі зв'язки, подвійні - інформаційні.

За фон Нейманом принцип роботи комп'ютера такий: з ЗП в пам'ять вводиться програма. КП зчитує вміст пам'яті, де знаходиться перша інструкція (команда) програми і організує її виконання. Ця команда може задавати виконання арифметичних або логічних операцій, читання з пам'яті даних для виконання логічних або арифметичних операцій, запис результатів у пам'ять. Як правило, КП після виконання однієї команди починає зчитувати команду з комірки пам'яті, що знаходиться безпосередньо за тільки що виконаною командою, але порядок передачі керування може бути змінений за допомогою спеціальних команд, при цьому такий стрибок повинний виконуватися за певних умов (менше 0, дорівнює 0, більше 0), що дозволяє створити програми складної структури з розгалуженням, циклами та переходами. КП виконує інструкції програми автоматично, без утручання людини. Всі результати виконання програми повинні бути виведені на ЗП, після чого КП переходить в режим очікування яких-небудь сигналів зовнішніх пристроїв.

Схема устрою сучасних комп'ютерів декілька відрізняється від приведеної вище схеми устрою комп'ютера по фон Нейману. АЛП і КП об'єднані в один пристрій - центральний процесор (ЦП). Крім того, процес виконання програми може перериватися для невідкладних дій - переривань, пов'язаних із сигналами, що надійшли, від зовнішніх устроїв. Багато швидкодіючих комп'ютерів здійснюють обробку даних на декількох процесорах. Проте, схема роботи сучасних комп'ютерів відповідає

принципам роботи, сформульованим Фон Нейманом.

Як виникли персональні комп'ютери (ПК)

Комп'ютери в 40-х і 50-х роках були дуже дорогими і великими пристроями. Перший крок до зменшення розмірів комп'ютерів став можливий із виникненням у 1948 транзисторів - мініатюрних електронних приладів, що змогли замінити в комп'ютерах електронні лампи. У 50-х роках винайшли спосіб здешевлення виробництва транзисторів. В другій половині 50-х комп'ютери на транзисторах у сотні разів подешевіли. Але спочатку пам'ять на лампах не змогли замінити на транзистори, однак потім винайшли пам'ять на магнітних сердечниках. У середині 60-х з'явилися більш компактні зовнішні пристрої.

Наступний крок в історії створення ПК - інтегральні схеми (ИС). До появи інтегральних схем транзистори розроблялися відокремлено і потім збиралися в схеми. У 1958 році Джек Кілби придумав як на одній пластині напівпровідника одержати декілька транзисторів. У 1559 році Робер Нойс (майбутній фундатор фірми INTEL) удосконалив метод: на одній пластині помістив транзистори і всі необхідні елементи. Його винахід назвали інтегральними мікросхемами (ІМС) або чипами. У 1968 році фірма BURROUGHS випустила перший комп'ютер на ІМС. С 1970 року INTEL почав продавати інтегральні схеми пам'яті.

У 1970 Эдвард Хофф (INTEL) сконструював інтегральну схему, аналогічну по своїх функціях центральному процесору великої ЕОМ. Так у 1971 з'явився перший мікропроцесор INTEL 4004, який розпочав історію мікропроцесорів. Але він був малопотужним - опрацьовував тільки 4 бита інформації, міг адресувати 640 байт пам’яті, мав тактову частоту 108 КГц, що було значно гірше ніж у процесорів великих ЕОМ на той час.

В 1972 INTEL випустила 8-бітовий мікропроцесор INTEL8008, що адресував вже 16Кб пам’яті, а в 1974 - INTEL-8080, що адресував вже 64Кб пам’яті, мав тактову частоту 2МГц. Цей процесор вимагав в трьох джерел електроживлення. На початку 1975 року, на базі мікропроцесора INTEL-8080 з'явився перший персональний комп'ютер Альтаір (фірма MITS). У 1975 Поль Аллен, Білл Гейтс створили для комп'ютера Альтаір інтерпретатор мови Бейсік.

Далі з'явилися комерційні програми для редагування текстів WORDSTAR і табличний процесор VISICALC (відповідно 1978 і 1979 р.).

“Наступним етапом став процесор i8085 (5 МГц). Він зберіг популярну регістрову архітектуру 8080 і програмну сумісність, але в нього

додали порт послідовного інтерфейсу, видалили спеціальні ІМС підтримки (тактовий генератор та системний контролер) і трохи змінили зовнішній інтерфейс. Головним подарункам розробникам став перехід на єдину живлющу напругу.

Процесори Z80 фірми Zilog стали варіацією на тему 8080 і 8085. Зберігши програмну сумісність з 8080, в нього додали додаткові регістри, що надало можливість підвищити продуктивність. Результат виявився вражаючим - ще нещодавно популярні комп'ютери Sinclair, побудовані на Z80, демонстрували на іграх графіку, що не поступається PC на 16-розрядному процесорі 80286. ” [8]

У 1979 році головна компанія по виробництву великих комп'ютерів фірма IBM почала розробку ПК. При цьому фірма IBM, щоб не витрачати великі гроші, дозволила розпочати розробку не з нуля, а використовувати досвід і вже існуючі блоки розробок інших фірм. За основу взяли INTEL-8088 (16- розрядний) мікропроцесор, що дозволило працювати з 1М пам'яті. Програмне забезпечення було доручено розробити фірмі MicroSoft. І в серпні 1981 року були випущені перші ПК IBM PC, що на сьогоднішній день сумісні на 90% з усіма ПК виробленими у світі. Це був і80286, що знаменував новий етап архітектури ПК. Він адресував 16Мб фізичної пам'яті, підтримував захищений режим роботи і віртуальну пам'ять до 1Гб. Але ці новації не знайшли широкого застосування і новий і286 використовувався як поліпшений і8088.

Принцип відкритої архітектури (ВА)

Що ж дозволило і286 і донині бути сумісним з більшістю сучасних ПК сімейства INTEL? Фірма IBM зробила ПК не єдиним нероздільним пристроєм, а забезпечила можливість його зборки з незалежно виготовлених частин аналогічно дитячому конструктору. При цьому методи сполучень пристроїв комп'ютера IBM PC були доступні всім бажаючим. Цей принцип назвали принципом відкритої архітектури.

Фірма IBM запропонувала такий устрій ПК. На основній електронній платі комп'ютера (системної або материнської) розміщені тільки ті блоки який здійснюють опрацювання інформації (обчислення). Схеми, що управляють всіма іншими пристроями - монітором, дисками і т.д., реалізовані на окремих платах, що вставляють у стандартні роз’єми на системній платі - слоти. До цих електронних схем підводиться живлення з єдиного блока живлення. І для зручності все це розміщається в однім металевому ящику - системному блоці.

ЕМА 1. Архітектура персонального комп'ютера сімейства INTEL

Як вже було сказано архітектура (принцип побудови) персонального комп’ютера сімейства INTEL (рис.2) принципово не дуже відрізняється від принципу фон Неймана.

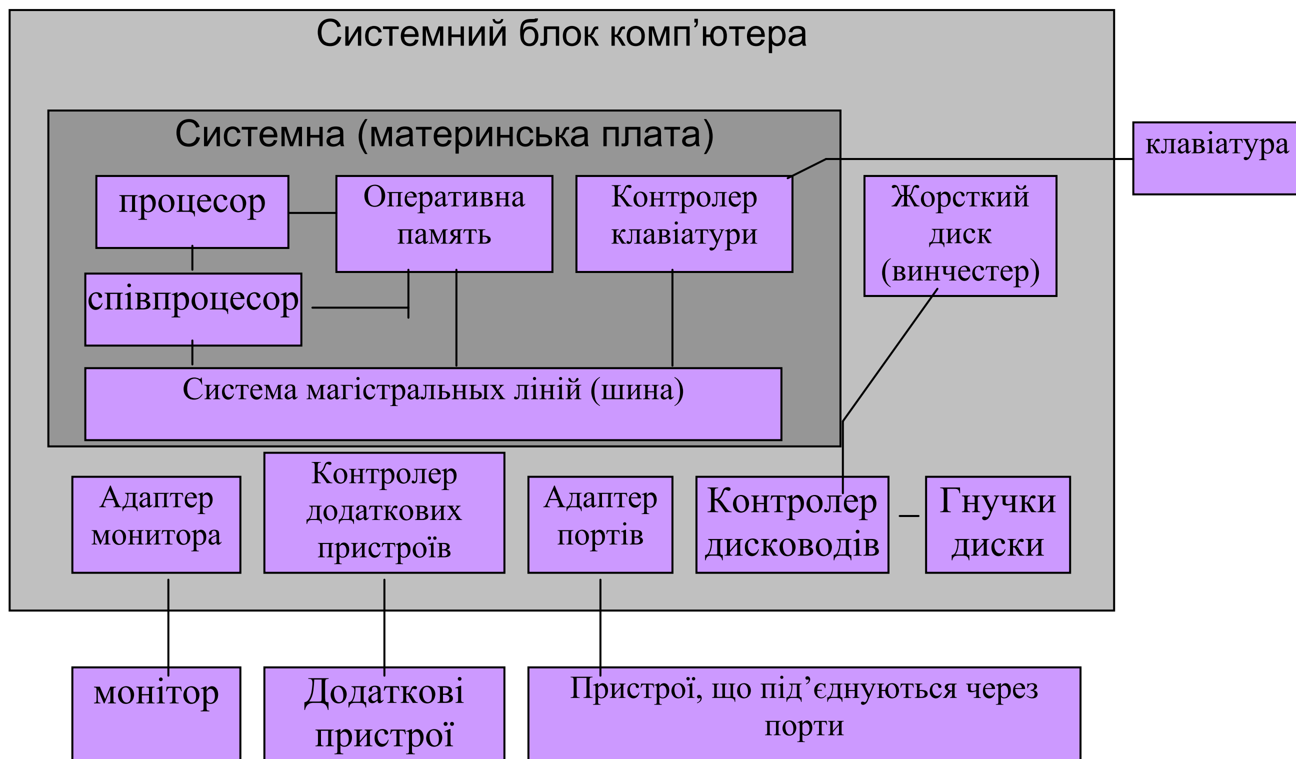

Рис. 2. Архітектура персонального комп’ютера сімейства INTEL

Центральний процесор (ЦП) - мозок комп'ютера – невеличка плата, що виконує всі аріфметико-логічні обчислення, обробляє сигнали від зовнішніх пристроїв та генерує управляючі імпульси. Мікропроцесор обробляє сотні різноманітних команд із швидкістю сотень мільйонів операцій в секунду.

Але ЦП виконує операції тільки над цілими числами, а для обробки дійсних чисел застосовується співпроцесор, який в молодших моделях процесорів реалізований як окремий пристрій, а в старших знаходиться на одному кристалі з ЦП.

Вся основна інформація, що поступає на обробку процесору зберігається в оперативній пам’яті або з оперативного запам’ятовуючого пристрою (ОЗП). Оперативна пам'ять (ОП) - важливий елемент

комп'ютера. З ОП мікропроцесор бере всі програми і вихідні дані в ОП і записує отримані результати, але при вимиканні комп'ютера інформації з ОП зникає, тобто ОП – тимчасове сховище інформації.

Для постійного зберігання інформації пристосовані постійний запам'ятовуючий пристій (ПЗП) та диски.

ПЗП – спеціальна мікросхема, на якій прошиті основні (базові) програми управління основними зовнішніми пристроями. Ця сукупність програм називається базовою системою введення-виведення (БСВВ або BIOS – base input-output system).

Всі розглянуті вище пристрої розташовані на материнській платі, на якій крім цього розташовані контролери переривань, контролер прямого доступу до пам'яті, генератор тактових імпульсів, системний таймер та інші.

Всі вхідні дані, що опрацьовуються потім ЦП надходять вони з різноманітних зовнішніх пристроїв комп'ютера - клавіатури, дисків, портів введення-виведення. Результати виконання програм так само виводяться на зовнішні пристрої – монітор, принтер і т.д. Таким чином весь час необхідний обмін інформацією між зовнішніми (ЗП) та внутрішніми пристроями (ВП). Для його реалізації між ЗП та ВП знаходяться дві проміжних ланки:

1. для кожного ЗП в комп'ютері є електронна схема, що їм управляє. Схема називається контролером або адаптером.

2. всі контролери й адаптери взаємодіють із мікропроцесором і ОП через системну магістраль передачі даних, що називається шиною.

Контролери знаходяться на окремих платах, що вставляються в спеціальні роз'єми (слоти) на материнській платі. Через ці слоти вони під'єднуються до шини. Любий постарілий адаптер легко замінити на новий.

Основними функціями контролерів є:

- управління роботою пристроїв;
- буферизація інформації. Не секрет, що ЗП працюють зі швидкістю набагато меншою ніж ВП, тому інформація, що поступає з ЗП спочатку накопичується, а потім на великій швидкості передається ВП і навпаки;
- забезпечення стандартного інтерфейсу між ЗП та ВП. Інтерфейс – це правила взаємодії двох об'єктів, де під об'єктами розуміємо пари: пристрій-пристрій, людина-програма і т.п. Контролер отримує сигнали від ЗП, перетворює їх в загальноприйняті коди і передає ВП і навпаки.

Як вже було сказано всі зовнішні пристрої пов'язані між собою сукупністю дротів – шиною. По шині передаються всі дані, управляючи сигнали та підводиться електроживлення. У відповідність з цим шини розподіляються на: шини даних, адресні шини, шини управління та шини живлення.

Найважливіша характеристика шини – це розрядність або об'єм інформації, яка передається по шині за один раз.

ТЕМА 2. Одиниці виміру інформації та системи числення

Комп'ютер опрацьовує тільки інформацію, подану в числовій формі, тобто звук, графіка, текст та інше кодується числами, а потім перетворюється на електричні сигнали. На сучасному рівні розвитку техніки існує тільки два сталих (достовірних) стана: увімкнений та вимкнений (є електрична напруга - 1 або немає – 0). Тому вся інформація подається в двійковій системі числення, тобто записується за допомогою нулів і одиниць.

Одиницею інформації є один біти, тобто двійковий розряд, що може приймати значення 0 або 1.

Як правило, інформація передається не окремими бітами, а групами послідовних бітів, що називаються байтами. В системах з 8-розрядною архітектурою один байт містить 8 бітів, а наприклад в телефонії використовується 5-розрядна архітектура, там один байт містить 5 бітів.

Більш значними одиницями інформації є:

кілобайт (Кб) = 1024 байтам ($1024 = 2^{10}$);

мегабайт (Мб) = 1024 Кб (2^{20});

гігабайт (Гб) = 1024 Мб (2^{30}).

Для роботи на ЕОМ застосовується чотири системи числення: десяткова, двійкова, вісімкова та шістнадцяткова.

Ці системи числення отримали свою назву від основи числення, тобто кількості знаків, що використовуються для запису числа. Найзручнішою для нас є десяткова система, у якій для запису використовується десять арабських цифр 0, 1, 2…9.

У двійковій системі для запису числа використовуються тільки дві цифри 1 та 0; у вісімковій – вісім 0, 1 … 7; у шістнадцятковій – шістнадцять: арабські цифри 0, 1, 2…9 та латинські букви A, B, C, D, E, F.

Всі ці системи є порядковими, тобто питома вага цифри при записі

числа залежить від її розташування у цьому числі. Наприклад, в десятковій системі в числі 33 перша від кінця трійка стоїть в розряді одиниць, а друга – в розряді десятків, тому питома вага першої – 3, а другої – 30.

Будь-яке число в порядковій системі числення можна подати у вигляді суми: $N = \sum_{i=0}^{k-1} m_i \cdot a^i$,

де N - подане число, k - кількість цифр в числі (кількість розрядів), m - кількісний еквівалент цифри, і - номер розряду цифри, *а* – основа системи числення.

Наприклад, число 25735 в десятковій системі можна подати так:

$25735 = 2*10^4+5*10^3+7*10^2+3*10^1+5*10^0$.

Розряди нумеруються з кінця числа, починаючи з 0:

4	3	2	1	0
2	5	7	3	5

Двійкове число 101 можна подати так:

$101 = 1*2^2+0*2^1+1*2^0 = 4+0+1 = 5$

При записі чисел в різних системах, щоби не виникало плутанини, використовують спеціальні позначення, які проставляються в кінці числа:

- латинська буква B або b – двійкова система. Наприклад, 101B.
- латинська буква H або h – шістнадцяткова. Наприклад, 78B3H .
- буква O або o – вісімкова.

Якщо після числа немає позначки, то вважається, що число записане в десятковій системі.

Треба сказати, що вісімкова система зараз використовується дуже рідко, тому надалі розглядатися не буде.

Перетворення чисел з однієї системи числення до іншої

Перетворення десяткових чисел у двійкові або шістнадцяткові може бути виконано методом віднімання або методом ділення.

Метод віднімання полягає у тому, що від десяткового числа віднімається найбільша можлива двійкова (шістнадцяткова) вага та у відповідну йому позицію записують 1. Від одержаного результату віднімають нову найбільшу можливу вагу та у відповідну йому позицію записують 1. Процес проводять доки, поки не буде отриманий нульовий результат, після чого у всі ті позиції бітів, ваги яких не віднімали, записують 0.

Наприклад, перевести 74 у двійкову систему числення.

_74
_64 /64=2^6, позиція біта 7=1
_10
_8 /8=2^3, позиція біта 4=1
_2
_2 /2=2^1, позиція біта 1=1
0

В усі інші біти (0,2,3,5,6) записують 0, отримуємо результат: 1001010B.

Для виконання зворотного перетворення з двійкового числа у десяткове необхідно значення кожного розряду помножити на його вагу та одержані результати просумувати. Наприклад, перетворимо у десяткове число одержане двійкове число 1001010B.

$$0*2^0+1*2^1+0*2^2+1*2^3+0*2^4+0*2^5+0*2^6=74$$

Метод ділення полягає у тому, що десяткове число ділиться на основу системи числення, в яку перетворюється число, поки частка не зробиться менше ніж дільник. Отримані остачі від ділення записуються у зворотному порядку і утворюють число у відповідній системі числення.

Слід пам'ятати, що першим (старшим) розрядом числа є остання отримана частка, другим - остання остача і так далі, останнім (молодшим розрядом) - перша остача. Наприклад, перевести 74 у двійкову систему числення.

74:2=37, остача 0
37:2=18, остача 1
18:2=9, остача 0
9:2=4, остача 1
4:2=2, остача 0
2:2=1, остача 0

Отримуємо результат 1001010B.

Метод ділення використовується також для перетворення десяткових чисел у шістнадцяткові. При цьому, якщо в остачі є числа 10,11,12,13,14,15, їх необхідно замінити символами A,B,C,D,E,F відповідно.

Наприклад, перетворити 74 у шістнадцяткове число

74:16=4, остача 10

Результат: 4AH

Перетворення з двійкового у шістнадцяткове число виконується шляхом виділення кожних чотирьох бітів – тетрад, починаючи з правого боку та перетворення їх відокремлено у шістнадцяткові розряди.

Наприклад, перетворити двійкове число 1001010В у шістнадцяткове.

Розбиваємо на тетради: 100 ‘ 1010 В.

Старша тетрада не повна: в ній три цифри, тому доповнюємо її зліва нулями. Отримуємо 0100 ‘ 1010 В.

Результат: 4АН.

Зворотна операція, тобто переклад шістнадцяткового числа у двійкове, потребує виконання дій у зворотному порядку.

Переклад шістнадцяткового числа у десяткове виконується за правилом перекладу двійкового у десяткове. При цьому вага кожного розряду змінюється так 16^0,16^1,16^2 і т.д.

Наприклад, перетворимо шістнадцяткове число 4АН у десяткове:

$$A*16^0+4*16^1=10*16^0+4*161=10+64=74$$

Арифметичні операції над двійковими числами

Операція додавання двійкових чисел проводиться аналогічно операції додавання десяткових чисел, тільки перенесення 1 у старший розряд виконується при наявності 1 в обох додаваємих розрядах, що і надано в таблиці:

Таблиця 1

Правила додавання двійкових чисел

	Початкові данні			Результат	
	Операнд 1	Операнд 2	Перенос	Сума	Перенесенні змінні
1	0	0	0	0	0
2	0	1	0	1	0
3	1	0	0	1	0
4	1	1	0	0	1
5	0	0	1	1	0
6	0	1	1	0	1
7	1	0	1	0	1
8	1	1	1	1	1

Приклад додавання двійкових чисел.

$_+1011$ $1*2^0+1*2^1+0*2^2+1*2^3=1+2+0+8=11$

$\underline{\quad 11}$ $1*2^0+1*2^1+=1+2=3$

1110 $0*2^0+1*2^1+1*2^2+1*2^3=0+2++4+8=14$

Операція віднімання чисел в ЕОМ замінюється на операцію додавання. Від'ємник замінюється додатковим кодом і до нього додається зменшуване.

Операція множення зводиться до операції зсуву ліворуч. Зсув на 1 біт ліворуч збільшує значення у 2 рази. У випадку множення на непарне число операція множення замінюється на операцію зсуву та додавання.

Операція ділення у загальному випадку замінюється на операції зсуву праворуч та віднімання (фактично додавання). Зсув на 1 біт праворуч зменшує значення у 2 рази. При зсуві ліворуч праві звільнені біти заповнюються 0. При зсуві праворуч чисел без знаку звільнений лівий біт заповнюється 0, а чисел зі знаком - звільнений біт заповнюється значенням знакового біту.

Подання від'ємних чисел (додатковий код)

Щоб отримати додатковий код від'ємного числа, необхідно узяти його додатну форму, інвертувати кожний біт, а потім додати одиницю до одержаного результату.

Наприклад, перетворити число -74 у додатковий код:

```
  01001010
  10110101  (результат інвертування)
 +       1  (додавання одиниці)
 ---------
 10110110   (додатковий код)  (-74)
```

Якщо отриманий код перетворити у десяткове число, то воно не буде дорівнювати 74. Проте сума двійкових кодів чисел +74 та -74 дають нульовий результат, що говорить о слушності даного підходу.

```
      01001010   (+74)
     +10110110   (-74) (додатковий код)
     ---------
 (1)  00000000
```

Вісім біт мають нульове значення, а перенесення одиничного біта ліворуч ігнорується.

Для визначення абсолютного значення від'ємного числа просто повторюють операції інвертування над додатковим кодом та додавання одиниці.

```
Наприклад:   10110110     -74
             01001001     результат інвертування
           +        1     додати одиницю
           ----------
             01001010      74
```

З цього виходить, що коли число знакове і його 7 або 15 біт має

одиницю (у залежності від того де воно розміщене - у байті або у слові), то комп'ютер це число сприймає як від'ємне, представлене у додатковому коді. У числах без знаку старший біт використовується під значення, а не під знак тому у байті розміщується число без знаку яке має значення від 0 до 255 та знакове число від -127 до +128.

ASCII коди та двійково-кодовані десяткові (BCD) числа

Дані, які вводяться у ЭОМ з клавіатури, записуються у пам'ять у вигляді ASCII символів. Аналогічно виведення інформації на екран здійснюється у кодах ASCII. ASCII-код кожного введеного/ виведеного символу записується в окремому байті. Наприклад, число 74 після вводу з клавіатури, у пам'яті буде зберігатися у двох байтах та мати такий ASCII-код у двійковому форматі:

0011 0111 0011 0100,

який у шістнадцятковому форматі має значення 3734.

Однак для виконання у ЕОМ арифметичних операцій над десятковим числом 74 використовується його двійкове значення (двійкове число):

0100 1010,

яке у шістнадцятковому форматі записується як 4A.

Тому слід не плутати представлення чисел у символьному форматі (ASCII-код) та у форматі арифметичних даних (двійкове число).

Двійкове число виходить з десяткового числа шляхом перетворення його методами ділення, віднімання, або іншими подібними.

ASCII-код десяткового числа може бути сформований з його упакованого BCD числа шляхом запису кожних його чотирьох бітів у окремому байті, у його правої частини, а у його лівої частини -двійкової величини 0011.

Можна сказати, що число зберігається у неупакованому форматі, якщо воно записано у вигляді ASCII-коді та в упакованому форматі, якщо - у вигляді двійкового значення числа, так як у першому випадку для зберігання одного розряду шістнадцяткового числа, відповідно вводимому десятковому числу, потрібно пам'яті у 1 байт, а у другому випадку - половину байта.

а)

0011	0111	0011	0100

б)

7	4

Формування ASCII-кодів для десяткового числа 74.

а/ у двійковому форматі

б/ у шістнадцятковому форматі.

Якщо кожну половину байта використовувати ні під один шістнадцятковий розряд, а під один десятковий, то у цьому випадку ми получимо нову форму представлення чисел, які називають упакованими двійково-кодованими десятковими числами(BCD).

Наприклад. Записати число 74 у вигляді упакованого BCD числа.

Для цього необхідно число 7 перетворити до двійкового 0111 та число 4 до двійкового 0100, потім помістити їх у байт пам'яті наступним чином:

01110100.

Одержаний результат є упаковане BCD число числа 74.

Неспаковані BCD числа виходять з упакованих BCD чисел шляхом запису кожних 4-х його бітів у окремому байті, у його правій частині.

Наприклад. Перетворити число 74 у неупаковане BCD число.

Упаковане BCD число 01110100, якому відповідає десяткове число 74, записується у неупакованому форматі так:

00000111	00000100

Упаковані BCD числа використовуються для зберігання великих чисел у RAM пам'яті комп'ютера та виконання над ними різних арифметичних операцій. Крім того перетворення таких величин у ASCII-коди здійснюється в кілька раз швидше, ніж двійкових чисел.

Тема 3. Архітектура процесора сімейства Intel

Як вже було сказано процесор це пристрій, що виконує команди. Процесор під'єднаний до пам'яті шиною, по якій за один раз передається тільки одне число (рис. 3). Тому однією з суттєвих характеристик процесору є розрядність шини, бо це обмежує ємність інформації, що

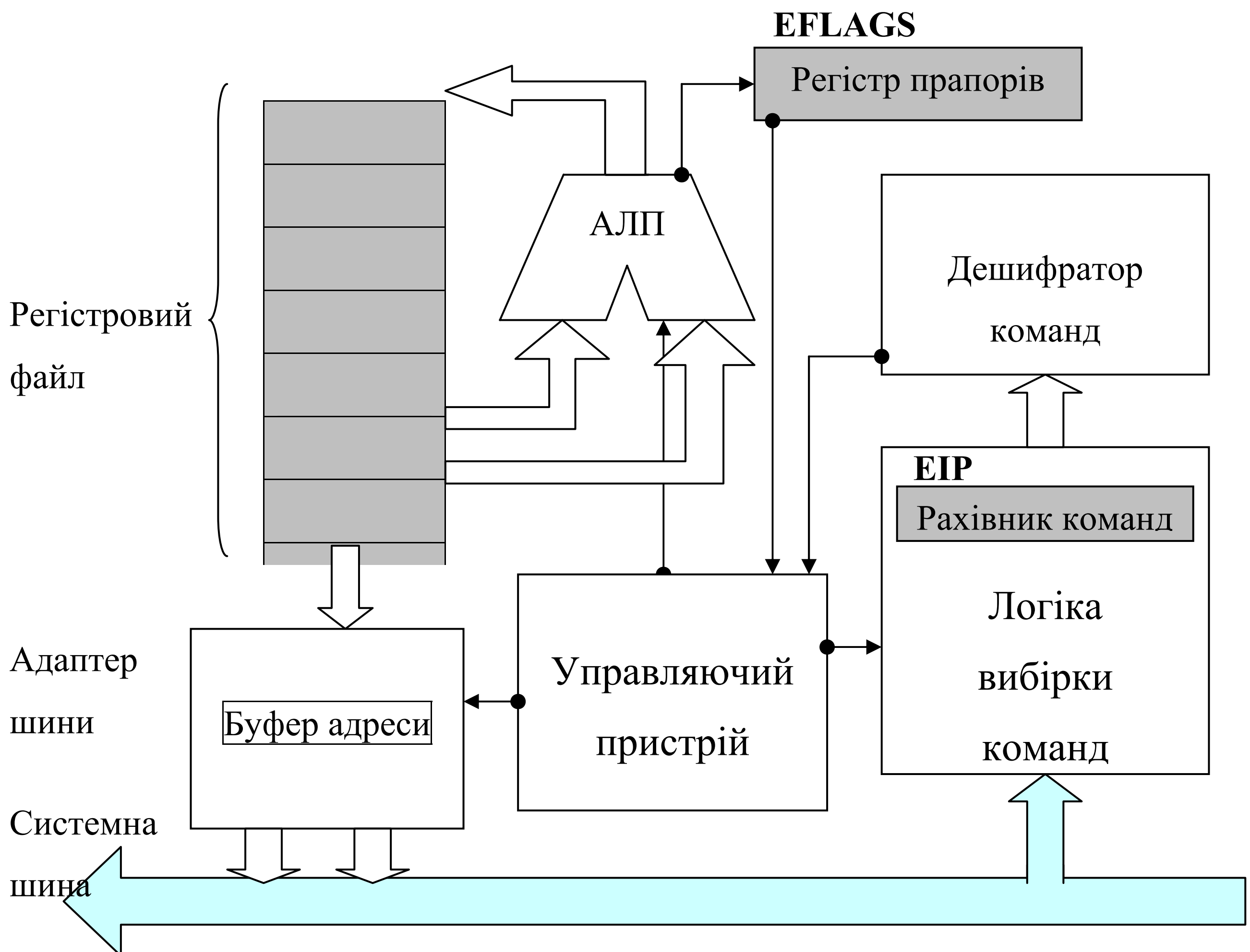

Рис. 3. Типова структура мікропроцесора [11]

передається за один такт. Наприклад, якщо розрядність шини 16 біт, а розрядність процесора 32 біти, то за один раз по шині буде передаватися тільки 16-розрядні дані. Для того, щоб передати 32-розрядні дані необхідно два такти шини, що значно зменшує швидкість. Для спілкування з шиною в процесорах існує шинний інтерфейс (ШІ) або адаптер шини. Функції шинного інтерфейсу такі:

- вибірка команд і даних для обробки в АЛУ; команди виструюються у чергу, дані буферизуються (накопичуються).
- управління передачею даних;

- управління адресацією пам'яті. Всі програмні команди знаходяться в пам'яті. ШІ повинно мати доступ до них для того щоб здійснити вибірку їх в чергу команд.

Розмір черги команд обмежений. Тому ШІ повинний здійснювати вибірку "з випередженням" та вибирати команди так, щоб існувала непуста черга команд, готових для виконання.

Операційний пристрій (ОП) виконую всі логічні та арифметичні операції, операції порівняння та команди.

ОП та ШІ працюють паралельно, при чому ШІ опереджає на один крок ОП. ОП повідомляє про необхідність доступу до даних в пам'яті або на пристроях вводу/ виводу. ОП запрошує машинні команди з черги вводу команд. Доки ОП виконує команди з черги, ШІ вибирає наступну команду з пам'яті. Ця вибірка здійснюється під час виконання, що підвищує швидкість роботи.

Арифметико-логічний пристрій (АЛП) процесору виконує всі арифметичні та логічні команди. Зазвичай він не може оперувати даним, що розташовані в оперативній пам'яті, тому що для виконання команди необхідний доступ до трьох комірок пам'яті, що зберігають операнди і результат. Для вирішення цієї проблеми будь-який процесор має декілька регістрів – спеціалізованих запам'ятовуючих пристроїв, що вміщують ціле число або адресу. Всі процесори мають як мінімум 6 регістрів:

- регістр для адреси поточної команди (рахівник команд **EIP**);
- регістр прапорців (**EFLAGS**), де зберігаються коди умов і багато іншої службової інформації. Часто цей регістр називають словом стану процесора;
- три буферні регістри АЛП;
- буферний регістр, в якому зберігається поточна команда.

Зі всіх цих регістрів програмісту доступні тільки рахівник команд та слово стану процесора (FLAGS) (під доступністю розуміємо можливість вказувати регістри як явні та неявні операнди команд).

Для доступу до регістрів процесору не треба займати зовнішню шину даних, цикл доступу до них дуже короткий і співпадає з циклом роботи АЛУ. Тому, чим більше у процесора регістрів, тим швидше він працює з оперативними даними.

Раніше кількість регістрів обмежувалася електротехнічними характеристиками апаратури. Кількість регістрів сучасного процесора вимірюється тисячами. Наприклад, сучасні процесори замість буфера команд мають так звану чергу попередньої вибірки команд. У мікроконтролерів PIC і Atmel те, що в специфікаціях зветься ОЗП,

фактично являють собою регістри.

Регістри процесору І486. Регістри загального призначення та індексні

Регістри, що доступні програмісту для зберігання даних називаються регістрами загального призначення (рис. 4).

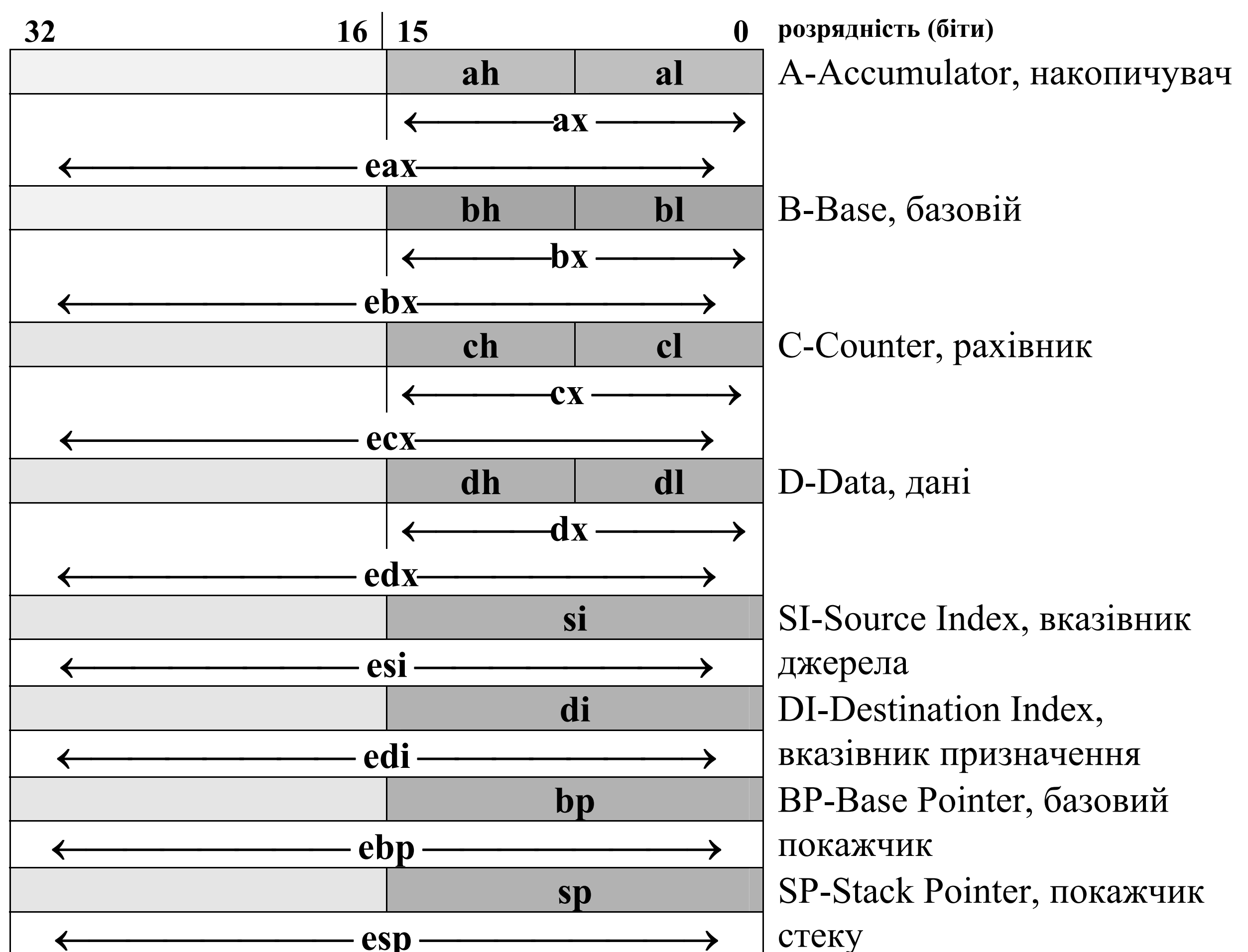

Рис. 4. Регістри загального призначення в системі команд x86

EAX – головний суматор, який може застосовуватися для всіх арифметичних операцій, операцій над строками та операцій вводу/ виводу.

EBX – базовий регістр. Регістр загального призначення, який може виконуватись при обчисленнях та в якості індексу для розширеної

адресації.

ECX – лічильник, може виконуватись при обчисленнях, необхідний для управління числом ітераційних циклів і для операції зсуву.

EDX – регістр даних. Застосовуються для арифметичних операцій (особливо при діленні великих чисел), при виконанні вводу/ виводу.

Регістри EAX, EBX, ECX, EDX 32-розрядні. У процесорах ***молодших за 486*** існували тільки ***16-розрядні*** регістри AX, BX, CX, DX, які зараз можуть адресуватися всіма новими процесорами. Крім того можливе використання їх складових: старшої (high) та молодшої (low) 8-розрядних частин – ah, al, bh, bl, dh, dl, ch, cl.

Сегментні регістри та адресація оперативної пам'яті

З точки зору процесору оперативна пам'ять це масив комірок, що пронумеровані. Номер кожної комірки називається її ***адресою***. Розрядність цієї адреси є однією з найважливіших характеристик процесора та системи його команд, бо це обумовлює розмір (об'єм) ***адресного простору***. Системи з 16-розрядною адресою адресують 64Кбайт пам'яті (2^{16}=65536=64К), з 32-розрядною – 4Гбайта. В теперішній час адресний простір в 4Гбайт вважається дуже малим і потребується 64-розрядна адресація.

Враховуючи те, що процесору потребується виконання арифметичних операцій над адресами, тому розрядність адреси зазвичай співпадає з розрядністю АЛУ.

Одиницею адресації в сучасних комп'ютерах є байт – комірка пам'яті розміром 8 біт. Крім того процесори можуть адресувати 2 байти – слово, 4 байти – подвійне слово (якщо це дозволяє розрядність) за один такт шини (або одну операцію пересилки даних).

Якщо адресація (нумерація) комірок пам'яті фіксована, тобто кожна комірка має незмінний адрес, то така адресація називається *фізичною*. Тоді адреса складається з полів, які безпосередньо використовуються як номер фізичної мікросхеми в пам'яті і номерів рядків і стовпців в цій мікросхемі.

Більшість сучасних процесорів загального призначення використовують *віртуальну адресацію*, коли номер конкретної комірки залежить не від фізичного розташування, а від контексту (режиму), в якому відбувається адресація.

В процесорах гарвардської архітектури данні і код зберігаються окремо (на відміну від принстонської, коли дані і код зберігаються разом). Тому для програмного застосування виділяється декілька блоків пам'яті

для зберігання коду, даних і організації стеку. Такий блок безпосередньо адресованої пам'яті називається ***сегментом***. В залежності від функціонального призначення бувають сегменти коду, сегменти даних та сегменти стеку.

Для формування (зберігання) адреси цих сегментів процесор використовує 16-розрядні адресні регістри (рис. 5).

16 0	
CS	CS – Code Segment
DS	DS – Data Segment
SS	SS – Stack Segment
ES	ES – Extra Segment
FS	FS – Data* Segment
GS	GS – Data* Segment

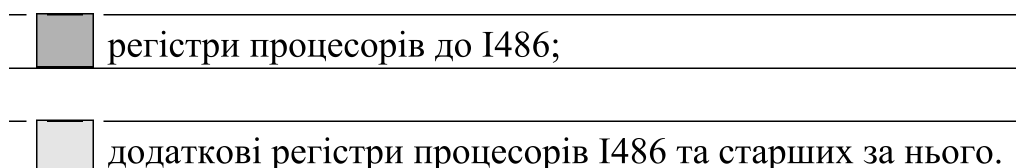

Рис.5. Сегментні регістри

CS – адреса сегменту коду виконуваної програми.

DS – адреса сегменту даних.

SS – адреса сегменту стека – області даних для тимчасового зберігання параметрів підпрограм, локальних змінних та адрес повернення в точки виклику.

ES, FS, GS - додаткові регістри, що дозволяють програмі працювати більш ніж 64К пам'яті одночасно.

В реальному режимі роботи процесора пам'ять розбивається на ***параграфи***. Границя параграфа – це адреса, що кратна 16. Таким чином, перший параграф має адресу 0000h, другий - 0010h, третій - 0020h. Якщо добре згадати шістнадцяткову систему обчислення, то всі числа, що кратні 16 закінчуються на 0.

Сегмент прийнято починати на границі параграфа, тобто адреса кожного сегмента закінчується на 0. Програмісти як люди розумні, а системні програмісти ще й дуже економні, вирішили, що для процесора

---

* ці додаткові сегменти процесорів старших поколінь використовуються для адресування додаткових сегментів даних.

зберігання зайвого нуля це розкіш. Тому в сегментних регістрах зберігається “усічена” адреса сегментів.

Виникає законне запитання: і що це йому дало? Давайте поміркуємо.

Сегментний регістр 16-розрядний. 2^{16}=65636, тому адресуємий простір не може перевищувати 64 Кбайт (“Маловато бедет!!!”, рис. 6).

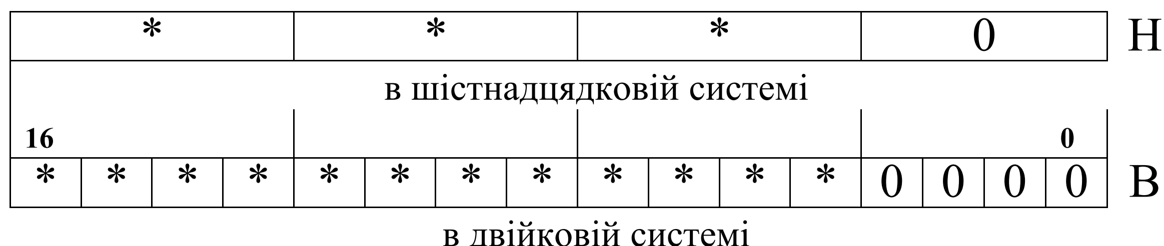

Рис. 6. Подання адреси сегменту

Якщо не зберігати кінцевий нуль, тим самим адресний простір розширюється до 2^{20}=1Мбайт (рис. 7).

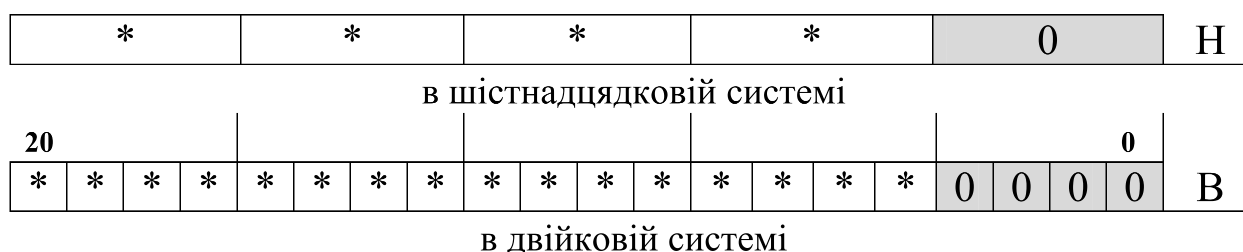

Рис. 7. Подання адреси сегменту в “усіченому” варіанті

Сегмент коду зберігає машинні команди, які будуть виконуватись. Звичайно перша команда має адресу, що співпадає з адресою сегмента і операційна система передає управління по адресу цього сегменту, який зберігається в CS. В середині програми всі адреси команд відраховуються відносно початку сегменту (рис. 8).

Таким чином для того щоб визначити фізичну адресу (ФА) певної команди в реальному режимі роботи процесора необхідно до адреси початку сегмента (АПС) додати зміщення команди (у байтах). Враховуючи те, що в сегментних регістрах процесора адреса зберігається в усіченому вигляді, спочатку її необхідно помножити на 16 (або у шістнадцятковій системі на 10Н) :

ФА = АПС*10Н + зміщення,

Наприклад, припустимо, що в регістрі CS зберігається 0045Н, а зміщення команди 0023Н, тоді ФА буде:

0045H*10H + 0023H = 00450H + 0023H=00473H

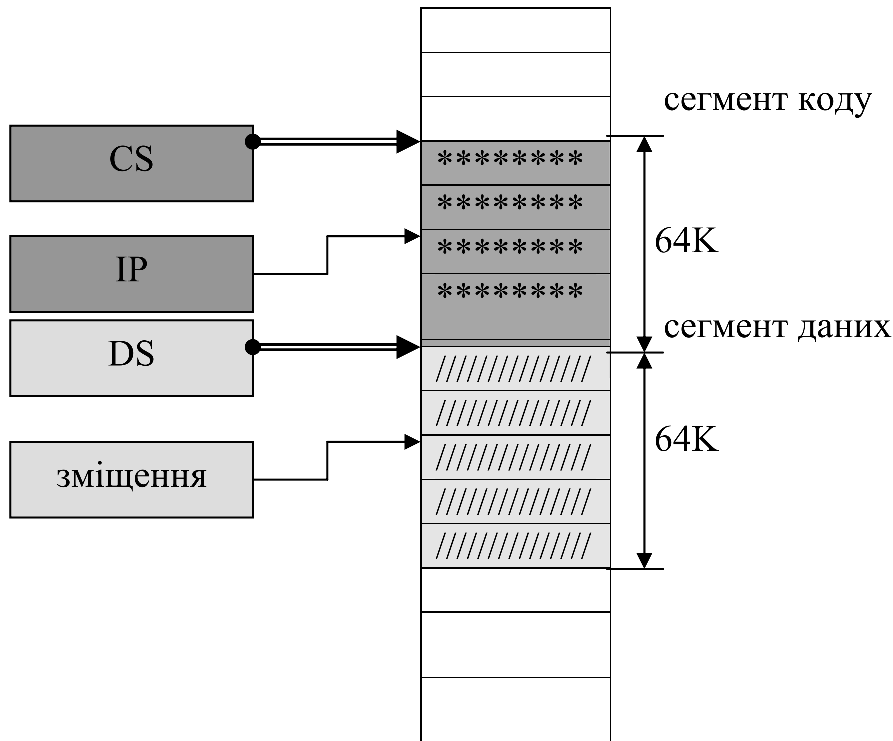

Рис. 8. Формування фізичної адреси в реальному режимі роботи

Найчастіше адресу записують у вигляді пари:

< сегмент > : < зміщення >

З наведеного приклада адреса буде 0045H:0023H

Як вже було сказано фізична адреса пам'яті, яка передається на шину –адреса одного байту. Тому розрядність шини визначає розмір адресного простору N

$$\boldsymbol{N = 2^p,}$$

де p – розрядність шини.

Розмір адресного простору для процесорів різних поколінь в реальному режимі роботи такий:

8086	p=20	N =2^{20} =1M
80286	p=24	N =2^{24} =16M
80386/486	p=32	N =2^{32} =4Г

Якщо шина даних 16 або 32-розрядна за одне звернення до пам'яті передається слово або подвійне слово. Адресою складених одиниць даних приймається молодша адреса, тих байт, які утворюють цю одиницю. Через це слово (або подвійне слово) розміщується в пам'яті в зворотному

порядку. Наприклад, якщо в регістрі EAX було записане 0456 2341H, то при пересилці в пам'ять молодша частина (2341H) буде записана в комірку з меншою адресою, а старша (0456H) – у комірки з більшою адресою (рис.9)

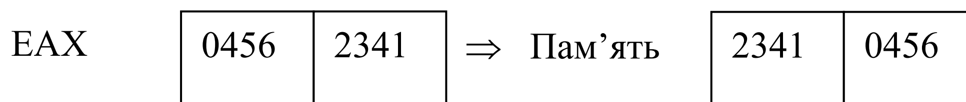

Рис. 9. Порядок розташування складених даних при передачі у пам'ять

Регістри зміщення

Регістри зміщення служать для виказання точної адреси байта (слова) відносно початку сегмента.

IP – визначає зміщення наступної інструкції, яка виконується в сегменті коду. Тому в процесорі адреса команди визначається парою регістрів CS:IP.

Адреса байту в сегменті даних визначається в залежності від використаного режиму адресації (див. відповідний розділ), але адресу сегменту визначатимуть регістри DS або ES:

DS : < зміщення > ,
ES : < зміщення >

Адреса байту в сегменті стеку визначається:
SS:SP (вершина),
SS:BP (початок стеку) – вказівки зміщення в стеці.

Організація стеку [10]

Стек – це спеціальним чином організована область пам'яті, що використовується для тимчасового зберігання змінних, для передачі параметрів підпрограм і для збереження адреси повернення підпрограм та переривань. Стек організований за принципом LIFO (рис. 10) – останній прийшов, перший пішов (last input, first output).

Таким чином, якщо записати в стек числа 1, 2, 3, тоді при зчитуванні вони будуть отримані в зворотному порядку – 3, 2, 1. Стек розташований у сегменті пам'яті, адреса якого зберігається в сегментному регістрі SS, а поточне зміщення вершини стека описується регістром ESP. При цьому стек росте вниз, тобто при запису у стек якогось значення його зміщення

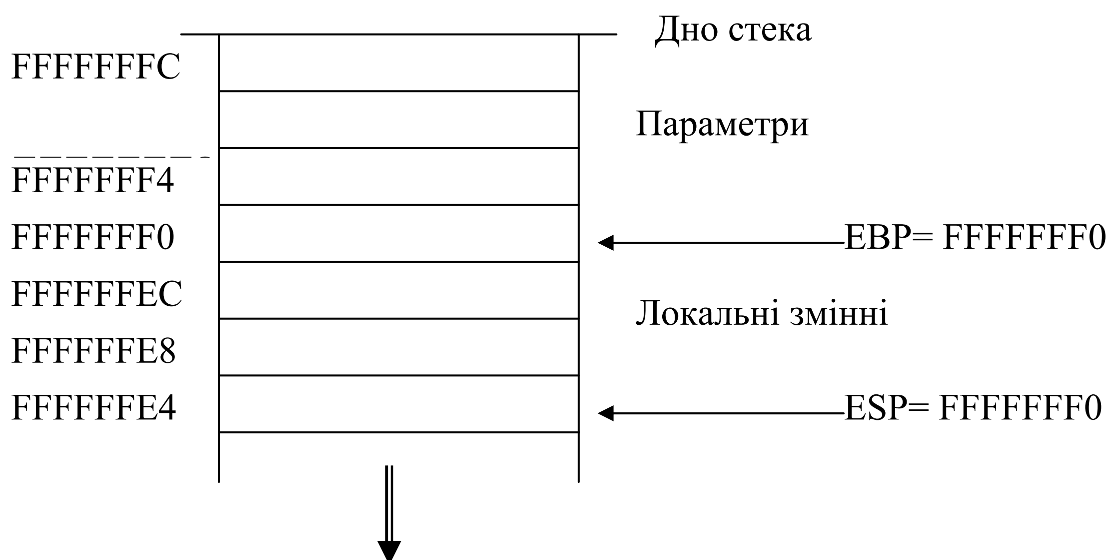

Рис. 10. Організація стеку

зменшується. Таким чином стек росте вниз до мінімально можливої адреси.

Таке розташування стеку “до гори ногами” може бути необхідним, наприклад при моделі організації пам’яті, коли всі сегменти враховуючи сегмент стеку і сегмент коду займають одну область пам’яті. Тоді програма виконується у нижній області пам’яті, в області молодшої адреси, і EIP зростає, а стек розташований в верхній області пам’яті, і ESP зменшується.

Під час виклику процедур в більшості випадків до стеку заносяться параметри, а до EBP записують поточне значення ESP. Тоді, якщо підпрограма використовує стек для зберігання локальних змінних, ESP змінюється, а EBP використовується для того щоб зчитати значення параметрів напряму зі стека. Більш детальну інформацію дивися у розділі “Передача параметрів в підпрограми”.

Регістр прапорів [10]

Регістр прапорів один з найважливіших регістрів, що використовується при виконанні переважної більшості команд. регістр прапорів EFLAGS містить набір статусних управляючих бітів, що називають прапорами. У процесорах, що молодші за 386 регістр прапорів 16- розрядний (FLAGS), у старших процесорів - 32-розрядний (рис.11).

Всі прапори в старшому слові регістра EFLAGS використовуються в захищеному режимі роботи процесора, тому розглянемо тільки регістр FLAGS.

CF – прапор переносу. Дорівнює 1, якщо при виконанні операції відбувся арифметичний перенос розряду, тобто результат операції не вмістився в приймачі або необхідний заєм (при відніманні). Інакше встановлюється в 0. Наприклад, при додаванні 0FFFFh та 1, якщо регістр, в який треба помістити результат – слово, в нього буде записано 0000h і прапор CF=1.

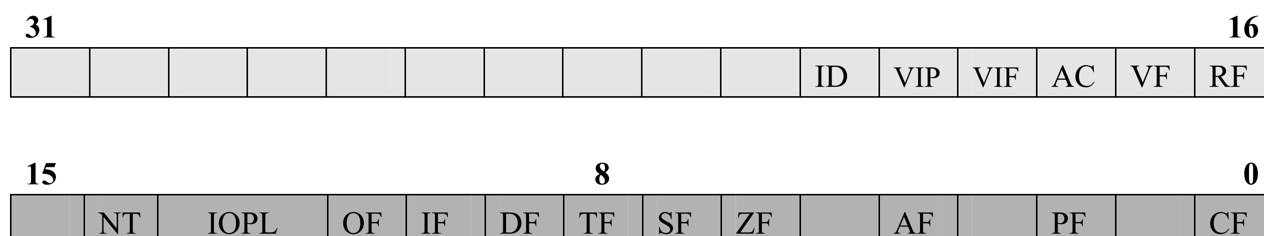

Рис. 11. Регістр прапорів

PF – прапор парності (або паритету). Встановлюється в 1, якщо молодший байт результату попередньої команди містить парне число біт, що дорівнюють 1 (PF =1 - парність). Не плутати з парністю числа, тобто ділимістю на 2! Число буде парним, якщо молодший біт числа буде дорівнювати 0.

AF – допоміжний прапор переносу (або заєму) з 3 в 4 розряди, для 8-бітних даних.

ZF – прапор нуля. ZF=1, якщо отриманий нульовий результат арифметичної операції або при порівнянні рівність двох операндів.

SF – прапор знаку. SF=1, якщо отриманий від'ємний результат арифметичної операції.

TF – прапор пастка. Цей прапор передбачений для програм-налагоджувальнику. Якщо TF=1, то встановлюється режим покрокової роботи процесора, тобто після виконання кожної команди управління передається налагоджувальнику.

IF – прапор переривань. якщо дорівнює 0, процесор перестає обробляти переривання від зовнішніх пристроїв.

DF – прапор напрямку. цей прапор контролює поведінку команд обробки рядків – коли він встановлений в 1, рядки обробляються в бік зменшення адреси (тобто зліва на право).

OF – прапор переповнення. Встановлюється в 1, якщо при виконанні операції зі знаковими цілими отриманий результат, що виходить за припустимі межі. Наприклад, якщо при додаванні двох додатних чисел, було отримане число зі старшим бітом, що дорівнює 1, тобто від'ємне, і

навпаки.

Прапори IOPL (рівень привілеїв введення/ виведення) та NT (вкладена задача) використовується у захищеному режимі.

Машинні мови і формати команд

Центральний процесор сучасного комп'ютера – це пристрій, що виконує команди. Повний набір команд, що виконується конкретним процесором називається машинною мовою або системою команд.

Різні процесори мають однакову або слабо відрізняючуся систему команд – наприклад, процесори Intel386, 486, Pentium, Pentium II, AMD K6, Athlon і т.д. – надалі всі ці процесори будемо називати x86.

Процесори, які вміють виконувати програми на одній і тій же машинній мові, називаються бінарно-сумісними. Співвідношення бінарної сумісності не завжди симетричне: наприклад, більш новий процесор може мати додаткові команди – тоді він буде сумісним з більш старим процесором, але не навпаки. Не рідко буває і так, що нові процесори мають зовсім іншу систему команд, але вміє виконувати програми на машинній мові старого процесора. Наприклад, всі процесори сімейства x86 в режимі сумісності виконують програми Intel 8086 і 80286.

Ще більша кількість процесорів сумісна між собою за мовою асемблера. Це означає, що кожна команда одного процесора має відповідну в системі команд іншого. Це дає можливість автоматизувати перетворення програм з однієї машинної мови на іншу.

Набори команд різних процесорів відрізняються великим розмаїттям, але існують операції, які в тій чи іншій формі виконуються всіма сумісними процесорами.

По-перше, це арифметичні операції над цілими числами в двійковому поданні, команди умовного і безумовного переходу.

Команда центрального процесора складається з коду операції і одного або декількох операндів (об'єктів, над якими виконується дія). В залежності від кількості операндів команди розділяються на безадресні (що не мають операндів або мають неявну вказівку на них), одноадресні (що виконують операцію над одним об'єктом або одним явно і одним або декількома неявно вказаними), двох- і триадресними.
Кількість адрес іноді використовують і для характеристики всієї системи команд. Двохадресною називають систему команд, в якій команди мають не більше двох операндів, триадресною – максимум три. Більшість сучасних процесорів мають дво- або триадресну систему команд.

ТЕМА 4. РЕЖИМИ АДРЕСАЦІЇ

Адресація – спосіб завдання адреси зберігання операндів певної команди.

1. Регістрова адресація

Операнди можуть бути розташованими в будь-яких регістрах загального призначення або в сегментних регістрах. В цьому випадку як операнди команди вказуються назви регістрів. Наприклад, для того щоб скопіювати вміст регістру BX в регістр AX , команду пересилання даних можна записати так:

```
MOV  AX, BX
```

2. Безпосередня

Деякі команди (всі арифметичні, окрім ділення) дозволяють вказувати значення одного з операндів безпосередньо в тексті програми, наприклад команда

```
 MOV  AX, 2
```

записує в регістр AX значення 2.

3. Пряма

Якщо відома безпосередня адреса певного операнда, то її можна вказувати в тексті програми:

```
MOV AX, ES:0001h
```

Ця команда запише в регістр AX значення, що зберігається в сегменті, на який вказує регістр ES, зі зміщенням від початку сегмента 0001h. Посилання на 16-розрядний регістр AX передбачає, що операнд розміром в слово (тобто теж 16 розрядів).

В реальному житті дуже рідко знадобляється звертатися до безпосередньої адреси. Найчастіше при прямій адресації вказують ім’я змінної:

```
MOV  AX, M
```

де M – ім’я змінної, що розташована у пам’яті в сегменті на який посилається регістр DS. Якщо, наприклад, ця змінна розташована в іншому сегменті, адреса якого зберігається в регістрі ES, тоді:

```
MOV  AX, ES:M
```

Але при використанні прямої адресації є деякі обмеження. ***Заборонено*** пересилати прямою адресацією значення ***в сегментні регістри та в стек***. У цих випадках припустима тільки ***регістрова адресація***.

4. Непряма адресація (рус. - косвенная)

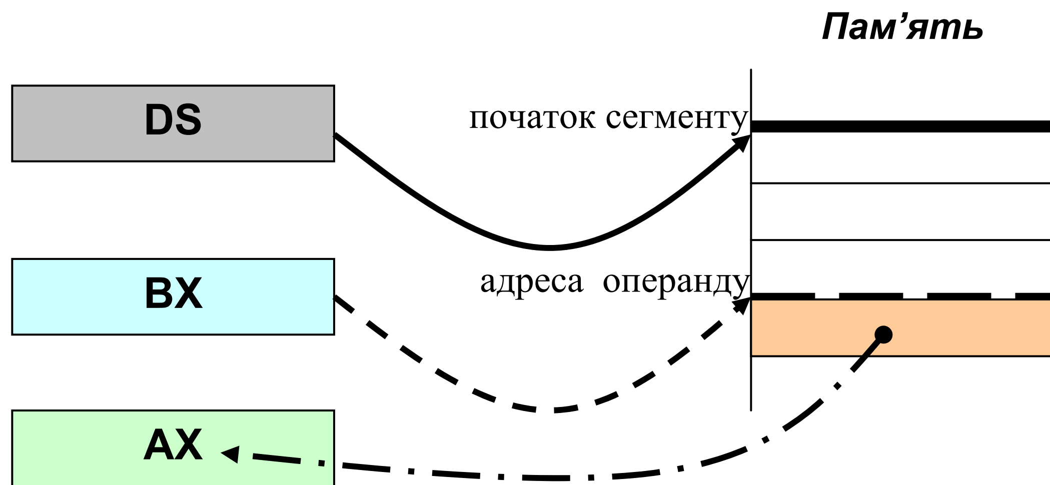

*Рис.*12. Непряма адресація

В режимі непрямої адресацій адреса операнда зберігається в регістрі. Команда

MOV AX, DS:[BX]

зчитує з пам'яті значення, що зберігається за адресою, що знаходиться в регістрі BX. Непряма адресація можлива і без вказування сегментного регістра:

MOV AX, [BX]

До 386 процесора для непрямої адресації для зберігання адреси могли вказуватися тільки регістри BX, SI, DI, BP. В старших моделях ці обмеження були зняті для регістрів EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP (але не можна використовувати AX, CX, DX, SP).

Якщо сегментний регістр явно не вказаний, то вважається , що це DS, але не у випадку, коли зміщення знаходиться в регістрах EBP, ESP або BP. Тоді як сегментний береться регістр SS.

5. Непряма по базі (базова зі зсувом)

MOV AX, [BX+2]

Вказана команда помістить в регістр AX слово, що знаходиться у сегменті, вказаному в DS зі зміщенням на 2 більшим ніж адреса, що знаходиться в BX.

Така форма адресації використовується у тих випадках, коли у BX знаходиться адреса початку структури даних або масиву, а доступ треба здійснити до певного елемента.

Другий поширений випадок використання цього режиму адресації –

доступ з підпрограми до параметрів, що знаходяться у стеку з використанням регістру BP (EBP).

MOV AX, [BP+2]

або звернення до локальних змінних:

MOV AX, [BP-2]

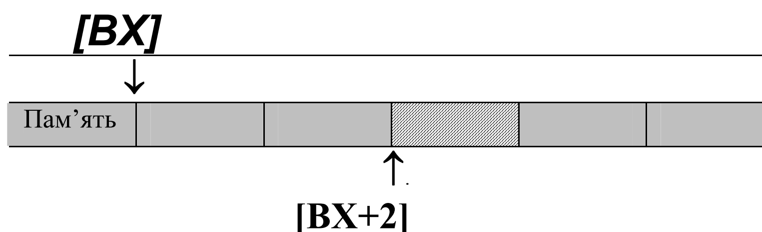

Рис. 13. Непряма по базі

Докладнішу інформацію дивися у розділі "Передача параметрів в підпрограми".

6. Пряма з індексуванням

Для доступу до елементів масиву зручніше використовувати інший режим адресації: прямий з індексуванням. Наприклад команда

MOV AX, A[SI]

записує в регістр AX значення елементу масиву A з номером індексу, що зберігається в регістрі DI.

7. Пряма з індексуванням та масштабуванням

Цей спосіб адресації повністю аналогічний попередньому, але за його допомогою можна прочитати елемент масиву слів, подвійних слів або зчетверених слів.

MOV AX, A[SI*n]

при цьому множник n може приймати значення – 1, 2, 4, 8, що відповідає розміру елемента – байт, слово, подвійне слово, зчетверене слово.

8. Адресація по базі з індексуванням

В цьому методі адресації зміщення операнду визначається як сума чисел, що знаходяться у регістрах, і зміщення, якщо воно вказане. Всі наступні команди – це різні форми запису одної і тої ж дії.

```
MOV AX, [BX + SI + 2]
MOV AX, [BX][SI] + 2
MOV AX, [BX + 2][SI]
MOV AX, [BX][SI + 2]
MOV AX, 2[BX][SI]
```

Цей режим зручний при роботі з двовимірними масивами. Якщо заданий двовимірний масив D (10, 10), регістр BX містить число 20, а SI = 7, то команда

MOV AH, D[BX][SI]

помістить в регістр AH елемент масиву D[2, 7]

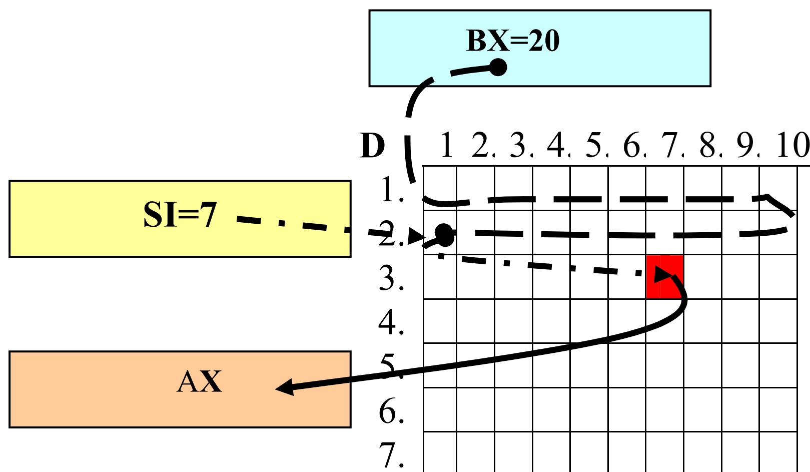

Рис. 14. Формування адреси елементу масиву D

ТЕМА 5. КОМАНДИ ПРОЦЕСОРІВ x86

1. КОМАНДИ ПЕРЕСИЛАННЯ ДАНИХ

1. Команда ***MOV***

Копією вміст джерела в приймач, діє як команда присвоєння в мовах високого рівня. Має два операнда: 1-й – приймач; 2-й – джерело. Наприклад,

MOV ax, a

В наведеному прикладі регістр **ax** є приймачем даних, в нього записується значення змінної **a**, в свою чергу а буде джерелом даних.

Як джерело може використовуватися регістри загального призначення, сегментні регістри, змінні у пам’яті, безпосереднє значення (чисельна або символьна константа).

Як приймач використовуються регістри загального призначення, сегментні регістри (окрім CS) або змінні.

Не можна виконувати пересилання даних за допомогою команди ***MOV***:

- з однієї змінної до іншої;
- з одного сегментного регістра в інший;
- пересилати в сегментний регістр безпосереднє значення. Ця дія має виконуватися в два етапи: перший – переслати значення до регістру загального призначення, друге – переслати з регістра

загального призначення до сегментного регістра.

*2. Команди обміну даними **XCHG***

Виконує обмін значень операндів проміж собою, тобто вміст операнду 1 копіює в операнд 2, а вміст операнду 2 – в операнд 1.

XCHG AH, AL
XCHG BX, AX
XCHG CX, OldCount

Як операнди можуть бути два регістри або регістр і змінна.

*2. Команда завантаження ефективної адреси **LEA***

Визначає адресу джерела (змінної) та завантажує її в приймач (регістр).

LEA BX, Value

За допомогою ***LEA*** можна обчислювати адресу змінної, яка описується складним методом адресації. Наприклад команда

LEA DI, A +2

занесе в регістр адресу третього байту змінної A.

*3. Команда **XLAT***

заносить до регістру AL байт з масиву, адреса якого формується парою регістрів ES:BX, зі зміщенням, що дорівнює AL (рис. 12). Для пересилання в регістр AL п’ятого елемента масиву можна задати таку послідовність команд:

```
MOV ES, DS ; пересилка в ES адреси сегменту даних
LEA BX, X    ; завантаження в BX адреси масиву X
MOV AL, 4    ; завантаження в AL зміщення п’ятого елемента
             ;(за умови, що всі елементи розміром в байт)
XLAT         ; пересилання даних
```

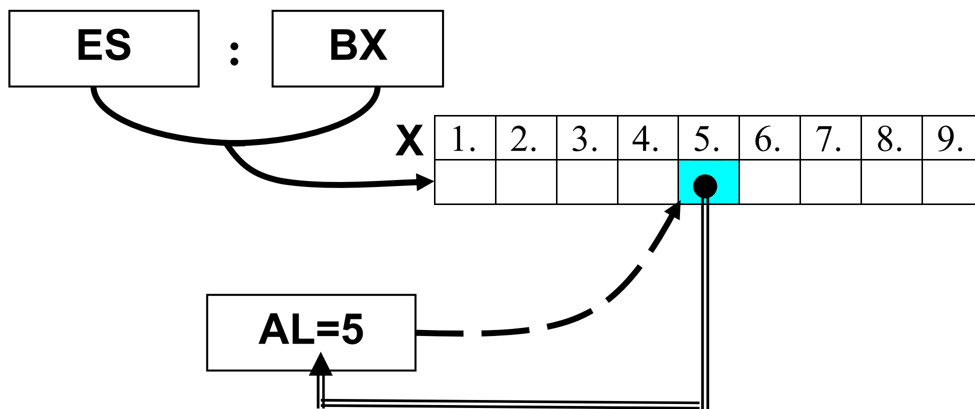

Рис 15. Схема пересилання даних командою XLAT

4. Команда пересилання даних у стек PUSH

Записує вміст джерела у стек. Джерелом може бути регістр, безпосередній операнд або змінна. Фактично ця команда копіює вміст джерела у пам’ять за адресою SS:[SP] та зменшує SP на розмір джерела в байтах (2 або 4). Наприклад,

PUSH ax

зменшить SP на 2 байти, а

PUSH ebx

на 4 байти.

5. Команда пересилання даних зі стеку ***POP***

Ця команда обернена до попередньої. Вона виконує виштовхування зі стеку байта або слова (в залежності від розмірів приймача) у приймач. При виштовхуванні зі стеку значення SP збільшується на розмір джерела в байтах (2 або 4).

6. Команда зчитування даних з порту ***IN***

IN <приймач>, <джерело>

Має два операнди. Копіює число з порту, номер якого вказаний в другому операнді в перший. Першим операндом (приймачем) може бути тільки один з регістрів AL, AX, EAX. Другим (джерело) – або безпосередньо номер порту, або регістр DX.

7. Команда запису даних в порт ***OUT***

OUT <приймач>, <джерело>

Має два операнди. Записує в порт, номер якого вказаний в першому операнді значення другого операнду. Другим операндом (приймачем) може бути тільки один з регістрів AL, AX, EAX. Першим (джерело) – або безпосередньо номер порту, або регістр DX.

Програмування портів введення-виведення розглянуто докладніше у розділі “Програмування на рівні портів введення-виведення”.

8. Команди конвертування

- Конвертування байту в слово ***CBW***;
- Конвертування слова в подвійне слово ***CWD***, ***CWDE*** ;
- Конвертування подвійного слова в зчетверене слово ***CDQ***.

Команди не мають операндів. Вважається, що операнд один з регістрів AL, AX, EAX, а результат теж заноситься відповідно до AX, EAX, DX (дивися таблицю 2).

Таблиця 2 **Команди конвертування**			
Команда	Процесор	Операнд	Результат
CBW	8086	AL	AX
CWD	8086	AX	DX:AX
CWDE	80386	AX	EAX
CDQ	80386	EAX	EDX:EAX

Розширення виконуються за рахунок розмноження старшого байту операнду на старшу частину результату.

3. АРИФМЕТИЧНІ КОМАНДИ

Всі арифметичні команди, крім команд ділення та множення, впливають на прапори CF, AF, SF, ZF, OF, PF.

1. Команди додавання

1.1 ADD op1, op2

Додавання двох цілочисельних операндів. Результат заноситься в перший, тобто обчислюється вираз:

op1= op1+op2

Операндами можуть бути змінні у пам’яті та регістри. Розмірність операндів повинна співпадати. Не можна, наприклад, додати до 8-розрядного операнда 16-розрядний.

1.2 ADC op1, op2

Двійкове додавання з використанням прапору CF, який додається до молодшого біту результату. Результат заноситься до першого операнду:

op1= op1+op2+CF

Команда ADC застосовується при додаванні багатобайтних (багатословних) операндів або коли додаються операнди різні за розміром. Наприклад, якщо необхідно додати до змінної A розміром в слово змінну B розміром в байт, додавання виконується в декілька етапів:

1. змінна B розширяється до розміру слова

```
MOV AL, B ; AL = B
CBW       ; AL →AX
MOV BX, AX
```

2. Завантаження змінної A до регістрів:

```
LEA BX, A
MOV AX, [BX]
MOV DX, [BX+2]
```

3. Додавання:

ADD AX, BX

ADC DX, 0

Якщо при додаванні молодших слів (команда ADD) відбувся арифметичний перенос розряду з молодшої частини до старшої (прапор CF=1), тоді він буде врахований наступною командою ADC (рис. 16).

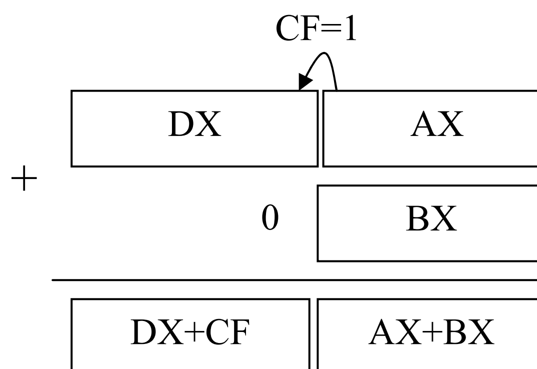

Рис. 16. Схема додавання з переносом

1.3 INC операнд

Збільшує операнд (регістр або змінну) на 1. Але ця команда на відміну від ADD не впливає на прапор CF.

2. Команди віднімання

2.1 SUB op1, op2

Віднімає op1 від op2 результат записує в op1.

op1= op1-op2

Операндами можуть бути змінні у пам’яті та регістри, але обидва операнди не можуть бути одночасно змінними. Другим операндом може бути безпосередньо число. Розмірність операндів повинна співпадати.

2.2 SBB op1, op2

Двійкове віднімання з використанням прапору CF, який віднімається від першого операнду. Результат заноситься до першого операнду:

op1= op1-op2-CF

Умови застосування аналогічні команді ADC.

2.3 DEC операнд

Зменшує операнд (регістр або змінну) на 1, не впливає на прапор CF.

2.4 NEG операнд

Змінює знак операнду (регістр або змінна) шляхом віднімання його значення від 0.

3. Команда порівняння

CMP op1, op2

Порівняння двох операндів. При порівнянні від першого операнду віднімається другий, але результат не записується. В залежності від результату віднімання змінюються прапори CF, OF, SF, ZF, AF, PF. Якщо не вдаватися у зайві подробиці, то в залежності від співвідношення операндів встановлюються такі прапори:

op1>=op2 CF=0, ZF=0

op1<=op2 CF=1, SF=1
op1=op2 ZF=1

Якщо порівнюються різні за розміром операнди, то перед виконанням операції порівняння значення меншого необхідно розширити до більшого.

4. Команди множення

4.1 MUL операнд

Добуток беззнакових цілих. Операндом може бути регістр або змінна. Робота цієї команди залежить від розміру операнда:

√ операнд розміром в *байт* множиться на вміст регістру AL. Результат розміщується в AX. Прапори CF і OF стають в 0, якщо AH=0;

√ операнд розміром в *слово* множиться на AX. Результат записується в пару DX:AX або EAX. Прапори CF, OF мають значення 0, якщо DX=0;

√ операнд розміром в подвійне слово [*] множиться на EAX. Результат записується в EDX:EAX.

4.2 IMUL операнд

Множення цілих зі знаком. Операнда (регістр/ змінна) множиться на регістр AL, AX, EAX, вибір якого визначається розміром першого операнду. Результат заноситься в регістр AX, EAX, DX:AX, EDX:EAX (дивися команду ***MUL***).

5. Команди ділення

5.1 DIV операнд

Цілочисельне знакове ділення. Як дільник задається операнд, ділене – регістр, вибір якого залежить від розміру дільника. Залишок ділення завжди менший від дільника.

Таблиця 3

Використання регістрів процесора під час ділення

Розмір дільника	***Ділене***	***Частка***	***Остача***
Байт	AX	AL	AH
Слово	DX:AX	AX	DX
Слово [*]	EAX	AX	DX
Подвоєне слово [*]	EDX:EAX	EAX	EDX

[*] - використання 32-розрядних регістрів можливе тільки для процесорів, починаючи з 486.

5.2 IDIV операнд

Цілочисельне ділення зі знаком. Робота цієї команди аналогічна DIV. Залишок має той же знак, що і ділене, а абсолютне значення його менше за дільник.

4. КОМАНДИ ОБРОБКИ БІТІВ

Команди обробки бітів можуть змінювати прапори CF, OF, ZF, PF, SF, значення AF невизначене.

AND op1, op2

Команда виконує логічне "І" над першим операндом (регістр або змінна) та другим операндом (число, регістр або змінна). Треба мати на увазі, що перший та другий операнди не можуть бути одночасно змінними. Результат операції записується у перший операнд. Найчастіше команда ***AND*** застосовується для обнулення окремих бітів, наприклад, команда:

AND AL, 00001111h

обнулить старші чотири біти регістру AL та збереже незмінними молодші чотири біти.

OR op1, op2

Команда виконує логічне "АБО" над першим операндом (регістр або змінна) та другим операндом (число, регістр або змінна). Перший та другий операнди не можуть бути одночасно змінними. Результат операції записується у перший операнд. Найчастіше команда ***OR*** застосовується для вибіркового встановлення окремих бітів, наприклад, команда:

OR AL, 00001111h

призведе до того, що молодші чотири біти регістру AL будуть встановлені в 1.

XOR op1, op2

Команда виконує логічне "виключаюче АБО" над першим операндом (регістр або змінна) та другим операндом (число, регістр або змінна). Перший та другий операнди не можуть бути одночасно змінними. Результат операції записується у перший операнд. Будь який біт результату дорівнює 1, якщо відповідні біти операндів різняться, та 0, якщо однакові, наприклад, команда:

XOR AL, AL

призведе до того, що вміст регістру AL обнуляться.

NOT op1

кожен біт операнда (регістр, змінна), що дорівнює 0, встановлюється в 1, і навпаки кожен біт, що дорівнює 1, встановлюється в 0.

5. Операції зсуву

SAL op1, op2
SHL op1, op2

SAL (SHL) – зсув бітів операнда вліво, старший біт переміщується в CF, молодший очищується.

SAR op1, op2
SHR op1, op2

SAR (SHR) – зсув бітів операнда вправо, молодший біт в CF, старший біт розмножується в порожні.

Зсув повторюється стільки разів, скільки задає другий операнд, який може задаватися безпосередньо в команді або знаходитися в регістрі CL.

Команди зсуву впливають на прапори OF, CF, SF, ZF, PF, значення AF не визначене.

6. Операції циклічного зсуву

RCL, RCR, ROL, RCR

За своєю дією команди циклічного зсуву аналогічні попереднім командам, але біти, що виходять за межі операнду не загублюються, а циклічно пересуваються в кінець (при зсуві вправо) або на початок при зсуві вліво (рис. 17). CF отримує копію біта, який був пересунутий. Зсув повторюється скільки разів скільки задається в другому операнді.

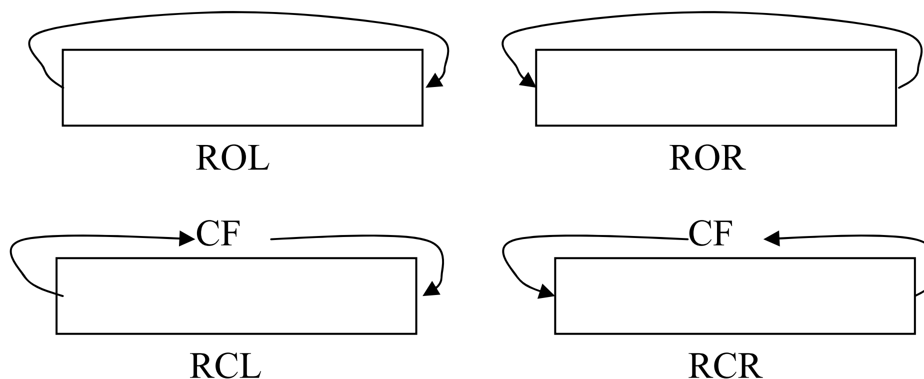

Рис. 17. Команди циклічного зсуву

7. Команди передачі управління

JMP операнд

Команда безумовного переходу JMP передає управління в іншу точку програми без збереження інформації для повернення. Операндом

може бути безпосередньо адреса переходу, ім’я мітки, регістр або змінна, що містить адресу.

```
A1:
ADD AX, 01
ADD AX, BX
        SHL CX, 1
        JMP A1
```

Якщо перехід здійснюється в межах –128…127 байт, то JMP – 1-байтна команда типу SHORT (короткий перехід). Якщо перехід вище зазначених меж, але в межах сегменту, то JMP 2-байтна команда (ближній перехід). Якщо адреса переходу в іншому сегменті, то JMP 3-байтна, тобто здійснюється дальній перехід.

Щоб скоротити розмір команди при ближньому переході вперед явно вказується тип переходу:

```
 JMP SHORT A2  . .
A2:    . . .
```

МІТКА включає в себе:

- √ літери A…Z, a…z;
- √ цифри;
- √ спеціальні символи “?”,”.”(тільки перший символ), “@”,”_”,”$”.

Перший символ ідентифікатора мітки – літера або спеціальний символ. Максимальна довжина 31 символ. Імена регістрів та зарезервованих слів не можна використовувати для міток.

CALL ***операнд***

Команда виклику процедури. Зберігає поточну адресу в стеку і передає управління за адресою, що вказана в операнді. Операндом може бути безпосередньо значення адреси, мітка, ім’я процедури, регістр або змінна, що містить адресу. Перед викликом процедури процесор поміщаю в стек значення регістру EIP, що відповідає наступній за ***CALL*** команді.

RET число

Команда повернення з процедури. Виконує дії зворотні до виклику процедури командою ***CALL***.

INT число

Команда виклику переривання. Операндом команди є номер переривання. Команда ***INT*** записує в стек регістр прапорів, CS, EIP, після чого передає управління програмі – обробнику переривання.

IRET число

Команда повернення з процедури обробки перериваня або виключення. Вивантажує зі стеку всі регістри, що були збережені перед цим командою ***INT***.

8. *Команди умовного переходу*

Jxx мітка

Цей набір команд, кожна з котрих виконує перехід типу near або short, якщо виконується відповідна умова. Найчастіше команди умовного переходу виконуються сумісно (безпосередньо після) команди CMP, яка встановлює певний стан прапорів.

Якщо необхідно виконати дальній перехід, то ***Jxx*** використовуються сумісно з командою ***JMP***.

Таблиця 4

Варіанти команди ***Jxx***

Команда	**Реальна умова**	**Умова для CMP**
JA JBE	CF=0, ZF=0	якщо вище якщо не нижче або дорівнює
JAE JNB JNC	CF=0	якщо вище або дорівнює якщо не нижче якщо немає переносу
JB JNAE JC	CF=1	якщо нижче якщо не вище або дорівнює якщо перенос
JBE JNA	CF=1, ZF=1	якщо нижче або дорівнює якщо не вище
JE JZ	ZF=1	якщо дорівнює якщо нуль
JG JNLE	ZF=0, SF=OF	якщо більше якщо не менше або дорівнює
JGE JNL	SF=OF	якщо більше або дорівнює якщо не менше
JL JNGE	SF<>OF	якщо менше якщо більше або дорівнює
JLE JNG	ZF=1, SF<>OF	якщо не менше або дорівнює якщо не більше
JNE JNZ	ZF=0	якщо не дорівнює якщо не нуль

Команда	Реальна умова	Умова для СМР
JNO	OF=0	якщо немає переповнення
JO	OF=1	якщо є переповнення
JNP JPO	PF=0	якщо немає парності якщо непарне
JP JPE	PF=1	якщо є парність якщо парне
JNS	SF=0	якщо немає знаку
JS	SF=1	якщо є знак

Для тог щоб легше було зрозуміти логіку побудови мнемоніки команд ***Jxx*** та правила їх застосування необхідно запам'ятати таке правило:

При побудові мнемоніки команди при роботі з ***беззнаковими цілими*** використовується позначення

√ A (above)– вище;

√ B (below)– нижче;

для знакових:

√ L (less) – менше;

√ G (greater) – більше.

Крім того позначення Z (zero) - нуль, P (parity) - парність, O (overflow) – переповнення.

9. Команда організації циклу

mov cx, < кількість ітерацій >

< мітка >:

< тіло циклу >

LOOP < мітка >

При виконанні команди ***LOOP*** перевіряється на рівність 0 вміст CX, якщо CX≠0, зменшує його на 1 та передає управління на мітку (виконує перехід типу short). Якщо CX=0, то робота циклу припиняється.

Команда ***LOOP*** не впливає а прапори.

10. Команди роботи з рядками символів

Всі команди обробки рядків вважають, що рядок-джерело знаходиться за адресою DS:SI (або DS:ESI), тобто в сегменті пам'яті, що вказаний в DS зі зміщенням SI, а рядок –приймач – відповідно в ES:DI (ES:EDI). Крім того, всі команди обробляють тільки один елемент рядка

(байт, слово, подвійне слово) за один раз. Для того щоб команда виконувалася декілька разів використовуються командні префікси повторення операції:

REP – повторювати;

REPE – повторювати доки дорівнює;

REPNE – повторювати доки не дорівнює;

REPZ – повторювати доки нуль;

REPNZ – повторювати доки не нуль.

Всі ці команди – префікси для операцій над рядками. Кожен з цих префіксів повторює команду, яка йде за ним стільки разів скільки вказано в регістрі ECX (або CX), зменшуючи його кожен раз на 1 при кожному виконанні команди. Крім того префікси ***REPZ, REPE*** припиняють повторення команди, якщо прапор ZF=0, а префікси ***REPNZ, REPNE*** припиняють повторення, якщо ZF=1.

Префікс ***REP*** зазвичай використовуються командами INS, OUTS, MOVS, LODS, STOS, а префікси ***REPZ, REPE, REPNZ, REPNE*** – командами CMPS, SCAS.

Поведінка префіксів не з командами обробки рядків не визначена.

1. Команди копіювання рядків

MOVS приймач, джерело

MOVSB приймач, джерело

MOVSW приймач, джерело

MOVSD приймач, джерело

Копіюють один ***MOVSB*** байт, ***MOVSW*** слово, ***MOVSD*** подвійне слово (використовується для процесорів старших од 386) із пам'яті за адресою DS:ESI в пам'ять за адресою ES:EDI. При використанні команди ***MOVS*** процесор сам визначає з типу вказаних операндів (прийнято вказувати імена рядків, що копіюються, але можна використовувати будь-які операнди, що підходять за типом), яку з трьох форм цієї команди (***MOVS***, ***MOVSW***, ***MOVSD***) використовувати у кожному випадку. Після виконання команди регістри ESI, EDI збільшуються на 1, 2, 4 байти відповідно до типу команди, якщо прапор DF=0, і зменшуються якщо DF=1. При використанні префікса ***REP*** команда ***MOVS*** копіює рядки довжиною в ECX (або CX) байтів, слів або подвійних слів.

2. Команди порівняння рядків

CMPS приймач, джерело

CMPSB приймач, джерело

CMPSW приймач, джерело

CMPSD приймач, джерело

Правила використання цих команд аналогічні до попередніх. Кожна з них порівнює елемент рядка з приймача з елементом рядка з джерела та встановлює прапори, як і команда CMP.

3. Команди сканування рядків

SCAS приймач

SCASB

SCASW

SCASD

Порівнює вміст регістру AL (***SCASB***), AX (***SCASW***), EAX (***SCASD***) з байтом, словом, подвійним словом, що знаходиться за адресою ES:EDI (ES:DI) і встановлює прапори аналогічно команді CMP. При виконанні команди регістр EDI (DI) зменшується або збільшується (відповідно до значення прапору DF) на 1, 2, 4 байти, що в свою чергу залежить від розміру операндів.

4. Команди читання з рядка

LODS джерело

LODSB

LODSW

LODSD

Копіює один байт, слово або подвійне слово із пам’яті за адресою DS:SI (або DS:ESI) в регістр AL, AX або EAX відповідно. Правила використання аналогічні до попередніх команд.

5. Команди запису в рядок

STOS приймач

STOSB

STOSW

STOSD

Копіює вміст регістру AL, AX або EAX в пам’ять за адресою ES:EDI (ES:DI). Правила використання аналогічні до попередніх команд.

6. Команди зчитування з порту

INS джерело, DX

INSB

INSW

INSD

Зчитує з порту введення/ виведення, номер якого вказаний в регістр DX, байт слово або подвійне слово в пам’ять за адресою ES:EDI (ES:DI). Правила використання аналогічні до попередніх команд.

7. Команди запису в порт

OUTS DX, приймач

OUTSB
OUTSW
OUTSD
Записує в порт введення/ виведення, номер якого вказаний в регістр DX, байт слово або подвійне слово з пам’яті за адресою DS:SI (або DS:ESI). Правила використання аналогічні до попередніх команд.

11. Команди управління прапорами

Опис команд управління прапорами наведений у таблиці 5.

Таблиця 5

Команди управління прапорами

Команда	Призначення
STC	Встановлює прапор CF=1.
CLC	Скидає прапор переносу.
CMC	Інвертує прапор переносу.
STD	Встановлює прапор напрямку (DF=1).
CLD	Скидає прапор напрямку.
LAHF	Завантажити прапори в AH.
SAHF	Вивантажити прапори станів з AH.
PUSHF	Записати прапори FLAGS в стек.
PUSHFD	Записати прапори EFLAGS в стек.
POPF	Вивантажити прапори зі стеку в регістр FLAGS.
POPFD	Вивантажити прапори зі стеку в регістр EFLAGS.
CLI	Заборонити переривання.
STI	Дозволити переривання.

ТЕМА 6. Призначення та використання співпроцесора

Числа з плаваючою крапкою та типи даних FPU

У процесорах Intel всі операції з плаваючою крапкою виконує спеціальний пристрій FPU (Floating Point Unit), що поставлявся спочатку у виді співпроцесора (8087, 80287, 80387, 80487), а починаючи з 80486DX - вбудований в основний процесор.

Типи даних, що обробляються співпроцесором представлені у таблиці 6.

Таблиця 6

Типи даних співпроцесора

Тип даних	Кількість біт	Кількість значущих цифр	Діапазон подання чисел
Ціле слово	16	4	-32768 … 32767
Коротке ціле	32	9	$-2*10^{9}$ … $2*10^{9}$
Довге ціле	64	18	$-9*10^{18}$ …$9*10^{18}$
Спаковане десяткове	80	18	-99..99 …+99..99 (18 цифр)
Коротке дійсне	32	7	$1.18 *10^{-38}$ … $3.40*10^{38}$
Довге дійсне	64	15-16	$2.23 *10^{-308}$…$1.79*10^{308}$
Розширене дійсне	80	19	$3.37 *10^{-4932}$ …$1.18*10^{4932}$

Дійсні числа зберігаються у формі двійкових чисел. Двійковий запис числа з плаваючою крапкою аналогічний десятковому, тільки позиції зліва від коми визначаються відніманням 2 у відповідному від'ємному степені.

Приклад 1

Перетворити число 0.625 у двійкову систему числення.

1. 0.625 – 0.5 (або 1/2)= 0.125 - результат операції додатний, записуємо 1.
2. 0.125 – 0.25(або 1/4)= - 0.125 – результат від'ємний, записуємо 0.
3. 0.125 – 0.125(або 1/8)= 0 – записуємо 1.

В результаті перетворення отримуємо двійкове дійсне число 0.101B.

Неодмінною вимогою є запис числа у нормалізованій формі: перед десятковою крапкою має стояти одиниця. Таким чином наше число має такий вигляд: 1.01B *2-1

Приклад 2

Перетворити число 3.25 у двійкову систему обчислення.

1. Ціла частина числа 3.25 – 3. У двійковій системі обчислення вона записується як 11B.
2. Дробова частина 0.25 – 01B.
3. Загальний вигляд числа 3.25 у двійковій системі обчислення 11,01B :

Номер розряду	1	0	-1	-2
Значення	1	1	0	1

Для зворотного перетворення необхідно обчислити суму:

$1*2^{1}+1*2^{0}+0*2^{-1}+1*2^{-2}$= 2+1+0.25=3.25.

Формати дійсних чисел з плаваючою крапкою

Числа з плаваючою крапкою складаються з двох частин: мантиси та порядку (експоненти) і записуються у виді :

$N = M \cdot S^p$,

де N - число з плаваючою крапкою;

M - мантиса числа;

S - основа системи обчислення;

p - порядок числа.

Приклад 3

- $5.729 = 0.5729 \cdot 10^1$, де 0.5729 – мантиса, 10 – основа системи обчислення, 1 – порядок числа;
- $1.101*2^{-1}$, де 1.101 мантиса двійкового числа, -1 – порядок.

Для того щоб не зберігати від'ємний порядок числа, він подається у так званому зміщеному вигляді, тобто збільшеним на певну константу T: $p' = p + T$. Наприклад, для короткого дійсного на $T = 2^6 = 64$, довгого дійсного – 1023, розширеного дійсного – 16 383. Тобто для короткого дійсного замість $-64 \le p \le 63$, значення порядку буде $0 \le p' \le 127$, де $p' < 64$ - від'ємний порядок, $p' \ge 64$ - додатний порядок. Це спрощує дії над числами та полегшує виконання операцій порівняння.

Крім того, число треба записувати у нормалізованій формі: перед крапкою має бути одна значуща цифра (див. приклад 1).

Розміри порядку (експоненти) та мантиси числа приведені у таблиці 7, формат запису двійкового числа на рис. 18, де M – кількість розрядів під мантису (відповідно до типу числа); P – кількість розрядів під експоненту; 1 розряд під знак числа (1 – від'ємне, 0 - додатне).

Таблиця 7

Формати чисел з плаваючою крапкою

Тип даних	Кількість біт			Кількість значущих цифр
	Загальна	Мантиса	Порядок (експонента)	
Коротке дійсне	32	23	8	7
Довге дійсне	64	52	11	15 –16
Розширене дійсне	80	64	15	19

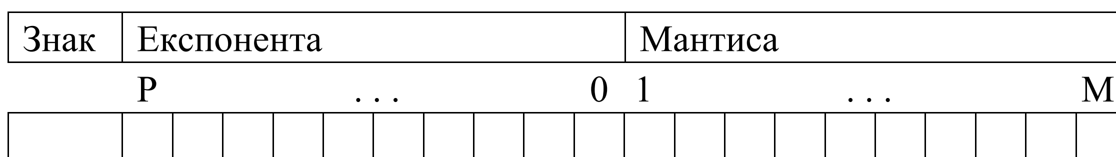

Рис. 18. Формат двійкового числа з плаваючою крапкою

Такий формат представлення даних відповідає міжнародному стандарту IEEE854. Його перевагою є зокрема те, що операції порівняння дійсних чисел відбуваються за тим же самим алгоритмом, що і порівняння цілих чисел.

FPU виконує всі обчислення в 80-бітному розширеному форматі, а 32- і 64-бітні числа використовуються для обміну даними з основним процесором та і пам'яттю.

У 32- і 64-бітному форматах, на відміну від 80-бітного, одиниця зліва від коми не зберігається.

Приклад 4

Десяткове значення -247.375 у двійковій системі обчислення має вигляд:

Знак
1 | 11110111,011B.

У нормалізованій формі:

Знак
1 | 1,1110111011B*27.

Таблиця 8

Формат дійсного числа у двійковій системі числення

Знак	Порядок	Мантиса	Тип	Примітка
1	10000110	11101110110…0	Коротке дійсне	Порядок збільшений на 127
1	1000000110	11101110110…0	Довге дійсне	Порядок збільшений на 1023
1	100000000000110	111101110110…0	Розширене дійсне	Порядок збільшений на 16383

Крім звичайних чисел формат FPU передбачає декілька спеціальних типів даних, що можуть утворюватися в результаті математичних операцій і над якими можна виконувати деякі операції:

Позитивний нуль: усі біти числа встановлені в нуль;

Негативний нуль: знаковий біт - 1, всі інші біти - нулі;

Позитивна нескінченість: знаковий біт - 0 , усі біти мантиси - 0 , усі біти експоненти - 1;

Негативна нескінченість: знаковий біт - 1 , усе біти мантиси - 0, усе біти експоненти - 1;

Денормалізоване число: усе біти експоненти - 0 (використовуються для роботи з дуже малими числами);

Невизначеність: знаковий біт - 1 , перший біт мантиси (перші два для 80-бітних чисел) - 1 ,а інші - 0 , усе біти експоненти - 1;

Нечисло типу SNAN (сигнальне): усі біти експоненти - 1 , перший біт мантиси - 0 (для 80-бітних чисел перших два бита мантиси -10) , а серед інших біт є одиниці;

Нечисло типу QNAN (тихе): усі біти експоненти - 1 , перший біт мантиси (перші два для 80-бітних чисел) - 1 , серед інших є одиниця. Невизначеність - один із варіантів QNAN;

Невизначене число: всі інші ситуації.

Інші формати даних FPU також припускають невизначеність - одиниця в старшому біті і нулі в інших для цілих чисел, і старші 16 біт - одиниці для спакованих десяткових чисел.

Архітектура співпроцесора

FPU надає вісім регістрів для збереження даних (рис. 18) і п'ять допоміжних регістрів (рис. 18).

Регістри даних FPU R0-R7 - це 80 – розрядні регістри. Їх робота організована за принципом стеку LIFO. Доступ до кожного регістра здійснюється за логічним ім’ям ST(0) – ST(7). Вершиною стеку вважається регістр ST(0).

Коли стек пустий, покажчик вершини стеку TOP = 7 і регістр R7 має логічне ім’я ST(0). Коли до стеку записується число покажчик вершини стеку TOP зменшується на 1, вершина стеку переміщується вище. Стек має кільцеву організацію. Коли він заповнюється, наступне значення записується поверх того, що було записане першим. При цьому буде згенеровано особливі ситуації: порушення стеку та неіснуюча операція (див. регістр станів).

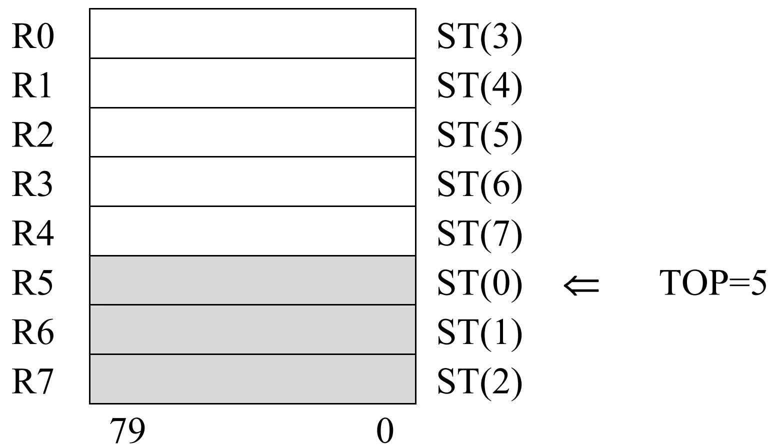

Рис. 18. Регістри даних FPU

Регістр станів SR містить слово стану FPU (рис. 19):
B – зайнятість FPU;
C3 – прапорець умови 3;
TOP – покажчик вершини стеку регістрів даних;
C2 – прапорець умови 2;
C1 – прапорець умови 1;
C0 – прапорець умови 0;
ES – загальний прапорець помилки, дорівнює 1, якщо виникла особлива ситуація.
SF – помилка стеку.
PE – прапорець неточного результату.
UE – прапорець антипереповнення;
OE – прапорець переповнення;
ZE – прапорець ділення на 0;
DE – прапорець денормалізованого операнда;
IE – прапорець неприпустимої операції.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	TOP			C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

Рис. 19. Регістр станів SR

Біти C3-C0 відповідають бітам регістра прапорців основного процесора і (C0– CF, C1 – не має, C2 – PF, C3 - ZF) використовуються при виконанні арифметичних операцій, операцій порівняння та умовних переходів.

Регістр управління CR, біти якого визначають спосіб округлення результатів, точність результатів деяких арифметичних команд та маскують відповідні виключні ситуації. Вміст регістру може змінюватися програмно.

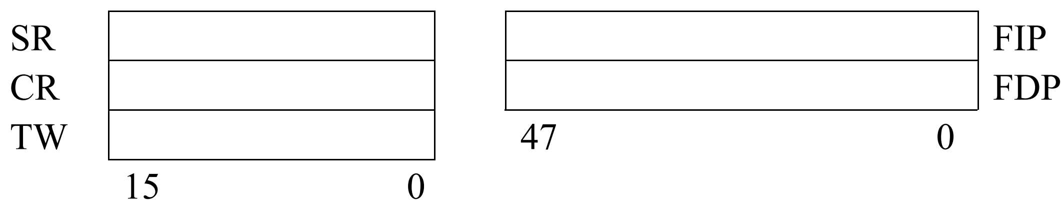

Рис. 20. Регістри управління FPU

Регістр тегів TW містить вісім пар бітів, що описують зміст кожного регістра даних. Біти 15 –14 описують регістр R7, 13—12 –R6 і т.д. Якщо пара біт дорівнює 11, відповідний регістр пустий, 00 – регістр містить число, 01 – ноль,10 – не число, нескінченність, денормалізоване число, непідтримуване число.

Регістри FIP і FDP містять відповідну адресу останньої виконаної команди і адресу її операндів.

Команди FPU

1. Особливості формування команд

Бінарні операції, наприклад додавання, припускають декілька форм.

- Вказані два операнди:

  < команда > <операнд1>,<операнд2>.

  Тут <операнд1>, <операнд2> - регістр даних FPU. При цьому один з них має бути ST(0), а другим - ST(i).

Після виконання операції покажчик стеку збільшується на 1, результат записується до першого операнда, другий операнд зі стеку видаляється.

- Вказаний один операнд:

  <команда > <операнд >.

<Операнд > - регістр даних, змінна у пам’яті, чисельна константа.

Операція виконується над вершиною стеку та вказаним операндом. Результат записується до вершини замість попереднього значення. Покажчик стеку не змінюється.

- Команда вказується без операндів.

Операція здійснюється над регістрами ST(0) та ST(1). Вміст ST(0)

виштовхується зі стеку, покажчик стеку зменшується на 1. Результат записується до нової вершини стеку – колишнього ST(1) (рис. 21).

До виконання операції

ST(7)	
ST(0)	
ST(1)	

Після виконання операції

. . .	
ST(7)	
ST(0)	Результат

Рис. 21. Використання регістрів при виконанні унарних операцій

Унарні операції виконуються над операндом або над операндом та вершиною стеку. Запис результату залежить від реалізації.

2. Команди пересилання даних FPU

Команди пересилання даних FPU унарні та мають два напрямки: завантаження до стеку та зчитування (копіювання) зі стеку. Вони еквівалентні командам завантаження до стеку PUSH та виштовхування зі стеку POP. При завантаженні виконується:

- √ зменшення покажчика стеку TOP на 1, таким чином створюється нова вершина стеку;
- √ передача операнда в нову вершину стеку.
- √ При зчитуванні (виштовхуванні) зі стеку виконується:
- √ передача значення ST(0) до операнда;
- √ звільнення регістра ST(0), тобто збільшення покажчика стеку TOP на 1.

При копіюванні зі стеку значення ST(0) передаються до операнда, вершина стеку не змінюється.

Таблиця 6

Команди пересилання даних

	Завантажити до стеку	Скопіювати у приймач	Виштовхнути зі стеку у приймач
Дійсне число	FLD	FST	FSTP
Ціле	FILD	FIST	FISTP
Спаковане десяткове	FBLD	-	FBSTP

Команди, що вказані у таблиці 3 мають 1 операнд. В залежності від напрямку операції операндами можуть бути:

- ➢ запис у стек - імена змінних, константи, ST(n);
- ➢ зчитування (копіювання) до змінної або ST(n).

Приклад 5

FLD Variable ; завантажити довге ціле
FLD ST(2); завантажити значення з ST(2) до вершини стеку
FILD M ; завантажити цілочисельне M
FST ST(5) ; скопіювати ST(0) до регістра стеку
FST R ; скопіювати ST(0) до змінної R
Команда: ***FXCH*** джерело
Призначення: поміняти місцями два регістри стеку
Обмін місцями вмісту регістра ST(0) і джерела (регістр ST(n)). Якщо операнд не зазначений, обмінюється вміст ST(0) і ST(1).

Команда: ***FCMOVcc*** приймач, джерело
Призначення: умовна пересилка даних
В залежності від виконання (або не виконання) певної умови копіює вміст джерела (регістр ST(n)) у приймач (тільки ST(0)), якщо виконується відповідна умова.

Таблиця 10

Команди умовної пересилки даних

Команда	Значення прапорців	Дія після FCOM
FCMOVE	ZF=1	Якщо дорівнює
FCMOVNE	ZF=0	Якщо не дорівнює
FCMOVB	CF=1	Якщо менше
FCMOVBE	CF=1 ZF=1	Якщо менше або дорівнює
FCMOVNB	CF=0	Якщо не менше
FCMOVNBE	CF=0 ZF=0	Якщо не менше або дорівнює
FCMOVU	PF=1	Якщо не порівнянні
FCMOVNU	PF=0	Якщо порівнянні

3. Арифметичні операції

Таблиця 11

Команди арифметичних операцій

	Дійсне число	Дійсне з виштовхув. зі стеку	Ціле число
Додавання	FADD	FSDDP	FIADD
Віднімання	FSUB	FSUBP	FISUB
Добуток	FMUL	FMULP	FIMULP
Ділення	FDIV	FDIVP	FIDIV

Кожна команда має два операнда. Результат виконання операції зберігається у першому операнді.

Арифметичні команди з виштовхуванням зі стеку звільнюють ST(0) та збільшують TOP на 1.

Команда: ***FABS***

Призначення: знайти абсолютне значення ST(0)

Якщо ST(0) був від'ємним числом, переводить його в додатне.

Команда: ***FCHS***

Призначення: змінити знак ST(0)

Змінити знак ST(0), перетворює від'ємне число в додатне і навпаки.

Команда: ***FRNDINT***

Призначення: округлити до цілого ST(0)

Округлити значення ST(0) до цілого числа відповідно до режиму округлення, заданим бітами RC.

Команда: ***FSQRT***

Призначення: добути квадратний корінь

Приклад 6

```
FADD ST(i), ST ; ST(i) ← ST(i) + ST
        FADD ST(i) ; ST← ST+ ST(i)
        FADD D   ; ST ← ST+ D, D  - дійсна змінна у пам'яті
```

4. Операції порівняння

Таблиця 12

Операцій порівняння

Типи операнду	Унарні операція		Без операндів
	Порівняння	Порівняння з виштовхуванням зі стеку	Порівняння з виштовх. двох чисел зі стеку
Дійсне число	FCOM	FCOMP	FCOMPP
Дійсне без врахування порядку	FUCOM	FUCOMP	FUCOMPP
Ціле число	FICOM	FICOMP	FICOMPP
Ціле з встановл.	FCOMI	FCOMIP	

Типи операнду	Унарні операція		Без операндів
	Порівняння	Порівняння з виштовхуванням зі стеку	Порівняння з виштовх. двох чисел зі стеку
прапорців EFLAGS			
Ціле без врахув. Порядку	FUCOMI	FUCOMIP	

Команди виконують порівняння вмісту регістра ST(0) з вказаним операндом (32- або 64- бітна змінна або регістр ST(n), якщо операнд не визначений – ST(1)), і встановлюються прапорці C0, C2 і C3 відповідно до таблиці 7.

Таблиця 13

Прапорці порівняння FPU

Умова	C3	C2	C0
ST(0)>джерело	0	0	0
ST(0)<джерело	0	0	1
ST(0)=джерело	1	0	0
Не порівнянні	1	1	1

Якщо один з операндів – нечисло або число, що не підтримується, виникає особлива ситуація "неприпустима операція".

Після виконання команд FSTSW AX (завантажити регістр SR в AX) і SAHF (переслати AH в FLAGS) значення C3, C2 і C0 записуються у відповідні біти регістра прапорців ZF, PF і CF, після чого всі умовні команди (Icc, CMOVcc, SETcc) можуть використовувати результат порівняння, як після команди CMD.

Команда FCOMP після виконання порівняння виштовхує зі стеку ST(0) (помічає його як порожній і збільшує TOP на 1), а команда FCOMPP виштовхує зі стеку і ST(0), і ST(1).

Команда: ***FTST***

Призначення: перевірити, чи не містить SP(0) нуль

Порівнює вміст ST(0) з нулем і виставляє прапорці C3, C2 і C0 аналогічно іншим командам порівняння.

5. Трансцендентні операції

Команда: ***FSIN***

Обчислює синус числа в ST(0), зберігає результат в тому ж самому

регістрі. Операнд вважається заданим в радіанах.

Команда: ***FCOS***

Обчислює косинус числа в ST(0), зберігає результат в тому ж самому регістрі. Операнд вважається заданим в радіанах.

Команда: ***FSINCOS***

Обчислює синус та косинус числа в ST(0), зберігає результат у стеку та зменшує покажчик стеку на 1. Операнд вважається заданим в радіанах.

Команда: ***FPTAN***

Обчислює тангенс числа в ST(0), зберігає результат в тому ж самому регістрі. Потім дописує в стек 1 так, що ST(0) містить 1, а ST(1) – результат операції. Операнд вважається заданим в радіанах.

ТЕМА 7. Структура програми мовою Асемблер

Основи програмування в MS DOS

Програма, що написана мовою Асемблер, також як і програма, що написана будь-якою іншою мовою програмування, виконується під управлінням операційної системи. Операційна система виділяє область пам’яті для програми, завантажує її, передає їй управління і забезпечує взаємодію програми з пристроями введення /виведення, файловою системою і іншими програмами. .способи взаємодії програми з зовнішнім середовищем різняться для різних операційних систем, таким чином програма, що була написана під управлінням Windows, не буде робити під управлінням DOS, а програма для Linux Solaris/ x86, хоча всі ці системи можуть функціонувати на одному і тому ж комп’ютері.

Найпростішою та найпоширенішою системою до недавнього часу для комп’ютерів на базі Intel була DOS (Дискова Операційна Система). DOS надає програмам повну свободу дії, ніяк не обмежуючи доступ до пам’яті і зовнішнім пристроям, дозволяючи їм самим управляти процесором і розподіленням пам’яті. Тому DOS як найкраще підходить для того щоб близько познайомитися з будовою комп’ютера і можливостями, які може мати програма на Асемблері, які часто криються компіляторами з мов високого рівня і більш досконалими операційними системами.

Таким чином, для того щоб програма виконувалася будь-якою операційною системою, вона має бути скомпільована в виконуємий файл. Основні два формати виконуємих файлів в DOS – COM та EXE.

Файли типу COM містять тільки скомпільований код програми без

будь-якої додаткової інформацій про саму програму. Всі дані, код і стек такої програми розташовані в одному сегменті і не можуть перевищувати 64 Кбайта (область пам'яті для стека розташована безпосередньо після закінчення програми, тому початково регістр SP містить адресу кінця програми. Це можна використати, наприклад, для визначення у програмі її реального розміру).

Файли типу EXE містять заголовок – ***префікс програмного сегмента PSP***, в якому описується розмір файлу, потребуємий розмір пам'яті, список команд у програмі, що використовують абсолютні адреси, які залежать від розташування програми у пам'яті і т.д. EXE-файл може мати будь-який розмір, тому що його код, дані та стек можуть розміщуватися у декількох сегментах.

При завантаженні COM програми структура даних PSP створюється на початку блоку пам'яті, що для неї відведений, також розміром до 256 байтів.

Крім звичайних виконуємих файлів операційна система MS DOS може завантажувати драйвери пристроїв – спеціальні програми, що використовуються для спрощення доступу до зовнішніх пристроїв і управління ними.

Програма типу EXE

Як вже було сказано, існують два типи виконуємих програм: EXE та COM програми.

Операційна система MS DOS висуває 4 вимоги до напису EXE-програм:

1. у програмі необхідно вказати, які сегментні регістри зберігають адреси сегментів програми;
2. на початку роботи програми необхідно зберегти в стеці адресу, що містить регістр DS;
3. записати в стек 0;
4. записати в DS адресу сегмента даних.

Кожній програмі типу EXE у пам'яті передує область розміром 256 байт, яка називається префіксом програмного сегмента PSP. Програма завантажувач (спеціальна програма операційної системи, яка завантажує текст програм в оперативну пам'ять) використовує регістр DS для збереження адреси початку PSP. Кожна програма типу EXE повинна зберегти цю адресу в стеці, для того щоб при завершенні роботи можна било визначити точку повернення в DOS. За правилами роботи DOS після

адреси повернення до стеку заноситься 0.

Структура EXE програми (приклад 1)

```
<ім'я_сегменту_кода> segment ; початок сегменту
assume cs:<ім'я_сегменту_кода>, ds:<ім'я_сегменту_даних>,
        ss:<ім'я_сегменту_стека>, es:<ім'я_сегменту_додатк.сегм.>
<мітка_початку_програми>:
;обов'язкова частина для кожної EXE-програми
        push ds
        sub ax,ax
        push ax
        mov ax, <ім'я_сегменту_даних>
        mov ds,ax

;текст програми

        ret          ;команда повернення з програми
<ім'я_сегменту_кода> ends

<ім'я_сегменту_даних> segment
; вміст сегменту даних
<ім'я_сегменту_даних> ends

<ім'я_сегменту_стека> segment
; вміст сегменту стека
<ім'я_сегменту_стека> ends

end  <мітка_початку_програми>          ;кінець програми
```

Опис сегментів та команда ASSUME

Як вже було сказано кожна EXE-програма може включати декілька сегментів. Зазвичай область пам'яті, у якій знаходиться текст програми називається сегментом коду, область, що містить дані – сегментом даних, а область, що відведена для стеку – сегментом стеку. Зрозуміло, що Асемблер дозволяє змінювати устрій програми як завгодно – розташовувати дані в сегменті коду, розподіляти код на декілька сегментів, поміщати стек в один сегмент з даними або зовсім використовувати один сегмент для всього.

Сегменти програми описуються директивою SEGMENT та ENDS:

<ім’я_сегменту> **segment** [<атрибут> <вирівнювання> <тип> <розрядність> ‘клас’]
. . .
<ім’я_сегменту> **ends**
<ім’я_сегменту> - мітка, яка буде використовуватися для отримання сегментної адреси.
Наступні 5 операндів директиви SEGMENT не є обов’язковими:
<атрибут> може приймати такі значення:

- readonly – тільки для зчитування;
- readwrite – зчитування і запис;
- execonly – тільки для виконання;
- execread – виконання та зчитування.

<вирівнювання> - вказує асемблеру, з якої адреси може починатися сегмент. Значення цього операнду:

- BYTE – з будь-якої адреси;
- WORD – з парної адреси;
- DWORD – з адреси, кратної 4;
- PARA – з адреси, кратної 16 (границі параграфа), що приймається за замовчанням;
- PAGE – з адреси, кратної 256.

<тип > - задає один з можливих типів комбінування сегментів.
<розрядність > - задає розрядність операндів сегменту (16- або 32-розрядні)
‘ клас ’ - будь-яка мітка, що взята в лапки. Сегменти з однаковим класом будуть розташовані в виконуємому файлі підряд.

Директива ASSUME вказує асемблеру, з яким сегментом (або групою сегментів) пов’язаний той або інший сегментний регістр.

assume <регістр >: <зв’язок > …
<регістр > - будь-який сегментний регістр;
<зв’язок > - мітка сегменту, групи сегментів.
Кількість сегментних регістрів, що вказуються в ASSUME жорстко не обмежується, але всі сегментні регістри, які використовуються в програмі мають бути визначені.

Кінець програми

end <мітка_початку_програми>

Цією директивою закінчується будь-яка програма мовою Асемблер. В ролі операнду тут постає мітка, з якої починається виконання програми.

Якщо програма складається з декількох модулів, то тільки один файл може містити початкову адресу.

Коментарі у програмах

Коментарі у програмах мовою Асемблер починаються з символу ; (крапка з комою) і закінчуються на кінці рядка. Коментарі можуть включати будь-які символи .

Для визначення великих блоків коментарів може використовуватися директива

```
comment  @
            текст коментаріїв
@
```

Операндом для comment може бути будь-який символ, який буде вважатися кінцем коментарію. Весь текст, між цими символами асемблером повністю ігнорується.

Моделі пам'яті та спрощені директиви опису сегментів

Моделі пам'яті задаються директивою .MODEL:

.model < модель > , < мова >, < модифікатор >

де < модель > - одне з наступних слів:

- ♦ TINY – код,, дані та стек розміщуються в одному сегменті розміром до 64 Кбайт. Ця модель пам'яті використовується при створення COM-програм.
- ♦ SMALL – код розташований в одному сегменті, а дані та стек в іншому. Ця модель пам'яті використовується для створення EXE-програм.
- ♦ COMPACT – код розташований в одному сегменті, а для збереження даних можуть використовуватися декілька сегментів.
- ♦ MEDIUM – код розташований в декількох сегментах, а дані в одному.
- ♦ LARGE та HUGE – і код, і дані можуть бути розташовані в декількох сегментах.
- ♦ FLAT – аналогічно TINY, але використовується в захищеному режимі роботи процесора.

Наступні два операнди не є обов'язковими.

< мова > приймає значення C, PASCAL, BASIC, FORTRAN, SYSCALL, STDCALL. Якщо він вказаний та асемблер вважає, що процедури призначені для виклику з відповідної мови високого рівня.

< модифікатор > приймає значення NEARSTACK, FARSTACK. В другому випадку сегмент стека не буде об'єднуватися з сегментом даних.

Після того як модель пам’яті визначена, вступає до сили спрощені директиви опису сегментів, які поєднують у собі директиви SEGMENT та ASSUME. Крім того сегменти, що оголошені через спрощені директиви не потребують використання ENDS. Вони завершуються автоматично, як тільки асемблер зустрічає нову директиву опису сегменту або кінець програми.

Розглянемо деякі директиви опису сегментів:

.CODE < ім’я_сегменту >

описує сегмент коду. Операнд < ім’я_сегменту > не є обов’язковим. Цей опис еквівалентний

< ім’я_сегменту > segment word public ‘CODE’

.STACK < розмір >

описує сегмент стеку і еквівалентна директиві:

STACK segment para publick ‘stack’

Необов’язковим є параметр < розмір >. По замовчанню він дорівнює 1 Кбайту.

.DATA

описує звичайний сегмент даних і аналогічний до директиви:

< ім’я_сегменту > segment word public ‘DATA’

.CONST

описує сегмент незмінних даних і аналогічний до директиви:

CONST segment word public ‘CONST’

Структура EXE програми (приклад 2)

```
.model small
.stack 100h
.code
<мітка_початку>:
        ;обов’язкова частина для EXE-програми
        mov ax, @data
        mov ds,ax
        . . .
        ;текст програми
        . . .
        ;закінчення програми замість ret
        mov ax, 4C00h
        int 21h
.data
;вміст сегменту даних
end <мітка_початку>
```

В цьому прикладі визначаються три сегменти: сегмент стека розміром 256 байт (100h), сегмент коду та сегмент даних. Команда mov ax, @data завантажує в регістр AX адресу сегмента даних. Ідентифікатор @data – зарезервована мітка, що містить адресу початку сегмента даних.

Для закінчення програми використовується не команда ret (вона призначена для виходу з процедури), а системний виклик DOS 4Ch, після чого викликається переривання 21h.

Програма типу COM

Структура COM-програми набагато простіша, тому що код і дані мають бути описані в одному сегменті, розмір якого не перевищує 64Кб. Стек для цих програм генерується автоматично. Крім того COM-програми не містять префіксу програмного сегмента, але при завантаженні програми в пам'ять цей префікс формується. Тому на початку COM програми требу зарезервувати під нього місце.

Директива управління програмним рахівником

Програмний рахівник – це внутрішня змінна асемблеру, що дорівнює зміщенню цієї команди або даних від початку сегмента. Для перетворення міток в адреси використовується значення цього рахівника. Значенням рахівника можна управляти за допомогою директиви

ORG < вираз >,

яка встановлює значення програмного рахівника.

Директива ***ORG*** з операндом 100h обов'язково використовується при написі COM-програм.

Структура COM програми (приклад 1)

<ім'я_сегменту_кода> **segment** ; *початок сегменту*

assume **cs**:<ім'я_сегменту_кода>, **ds**:<ім'я_сегменту_кода>, **ss**:<ім'я_сегменту_кода>, **es**:<ім'я_сегменту_кода>

;обов'язкова команда для кожної COM-програми

ORG 100h

<мітка_початку_програми>:

;текст програми

ret *;команда повернення з програми*

;опис даних

<ім'я_сегменту_кода> **ends**

end <мітка_початку_програми> *;кінець програми*

Структура COM програми (приклад)

```
.model tiny
.code
;обов’язкова команда для кожної COM-програми
        ORG  100h
<мітка_початку>:
        . . .
        ;текст програми
        . . .
        ret          ;команда повернення з програми
;опис даних
end  <мітка_початку_програми>          ;кінець програми
```

В наведених прикладах описується програма типу COM, яка складається з одного сегмента, дані в програмі описуються безпосередньо після завершення тексту програми. Вихід з програми в обох прикладах (на відміну від EXE програми) виконується командою RET.

Висновки за темою:

1. Існують три типи виконуємих програм:

√ COM програми;
√ EXE програми;
√ драйвери зовнішніх пристроїв.

2. Ці програми відрізняються за своєю будовою:

- EXE програми містять префікс програмного сегмента PSP; можуть складатися з декількох сегментів, завдяки чому розмір EXE програми не обмежений.
- COM програми не містять PSP, але на початку роботи резервують для нього місце; складаються з одного сегмента (дані описуються в сегменті коду); розмір обмежений 64 Кб (розмір одного сегменту в MS DOS).

3. При написі програм використовуються різні способи опису сегментів та завдання моделей пам’яті.

Тема 8. Організація підпрограм у програмах мовою Асемблер

Синтаксис опису підпрограми

У програмах мовою Асемблер не розрізняються різні типи підпрограм (процедури, функції), тому всі підпрограми називаються процедурами. Для оформлення у програмі процедури використовуються такі директиви:

<мітка> proc <мова> <тип> USES регістри

<мітка> - ім'я процедури, що складається за правилами формування ідентифікаторів мови Асемблер;

<мова> - визначає взаємодію процедури з мовою програмування високого рівня. Приймає значення C, PASCAL, BASIC, FORTRAN, SYSCALL, STDCALL.

<тип> - може приймати значення NEAR або FAR. За замовчанням вважається NEAR в моделях пам'яті SMALL, COMPACT, TINY. Тип FAR – указує завантажнику в ОС (операційна система), що початок даної процедури є точкою входу для виконання програми.

USES регістри – перелік регістрів, які буде змінювати процедура. Якщо вони вказані, то асемблер автоматично згенерує на початку процедури збереження вмісту вказаних регістрів у стеку (набір команд POP), а перед закінченням виштовхне їх зі стеку (набір команд PUSH).

Команда RET закінчує виконання процедури і повертає управління процедурі, з якої була викликана поточна процедура.

Директива ENDP - визначає кінець процедури і помічається міткою, що вказана в заголовку перед директивою PROC.

Наприклад, сегмент, що містить лише одну процедуру має вигляд:

```
Ім'я_сегмента       SEGMENT
Ім'я_процедури      PROC FAR ; сегмент коду з однією процедурою
                           …
; тіло процедури
                           …
                    RET
Ім'я_процедури   ENDP
Ім'я_сегмента       ENDS
```

END Ім’я_процедури

Якщо в сегменті коду описано декілька процедур, то тільки одна з них може мати тип FAR.

Виклик процедури здійснюється командою:

CALL ім’я_процедури

Наприклад, у сегменті коду описані три процедури: головна – BEGIN, B10 та C10.

```
        CODESG SEGMENT PARA
BEGIN PROC FAR
                ----------------------------------------------
                ;тіло головної процедури BEGIN
                ----------------------------------------------
        CALL B10
        CALL C10
        RET
BEGIN ENDP
B10 PROC NEAR
                ----------------------------------------------
                ; тіло процедури B10
                ----------------------------------------------
        RET
B10 ENDP
C10 PROC NEAR
                ----------------------------------------------
                ; тіло процедури C10
                ----------------------------------------------
        RET
C10 ENDP
CODESG ENDS
END BEGIN
```

Передача параметрів у підпрограму

До процедури параметри передаються :

- за значенням – процедурі передається значення параметру, яке копіюється, використовується у процедурі і не може модифікувати вхідне значення параметру;
- за посиланням – процедурі передається адреса параметра, по якій процедура може як зчитати його значення так і змінити його.

Параметри у процедуру передаються через:

- глобальні змінні – процедура використовує змінні, що були описані в головній програмі;
- регістри – перед викликом процедури до певних регістрів записуються значення параметр, які потім використовуються у процедурі;
- стек – перед викликом процедури до стеку записуються необхідні значення або посилання (адреси змінних) і потім зчитуються процедурою.

Саме передачу параметрів через стек використовують мови високого рівня Сі та Паскаль з тією різницею, що при записі у стек у програмах мовою Сі параметри заносяться з кінця (спочатку останній, потім передостанній і так далі), а у Паскалі параметри заносяться, починаючи з першого.

Розглянемо приклад, коли до стеку заносяться два параметри – значення.

```
        push param1
        push param2
        call myproc
        . . .
myproc      proc near
            push bp
            mov bp, sp
            mov ax, [bp+4]
            mov bx, [bp+6]
            . . .
            pop bp
            ret 4
myproc endp
```

Рис. 22 Стековий кадр

Перед викликом процедури до стеку заносяться два параметри. На початку роботи процедури у стек заноситься старе значення регістру BP, а потім до BP записується нове значення вершини стеку.

“Параметри в стеці, адреса повернення і старе значення BP разом називається ***активаційним записом*** функції” [10] або ***стековим кадром*** (рис. 22).

Виникає запитання, хто несе відповідальність за очищення стеку, тобто видалення параметрів після завершення роботи процедури? Відповідь проста – програміст. Зверніть увагу, що в наведеному прикладі перед закінченням вказується ret 4, що видалить зі стеку 4 байти, тобто два

параметри, що були туди записані.

Створення локальних змінних у підпрограмах

Часто у процедурах використовуються локальні змінні, в які після закінчення роботи процедури немає потреби. Найпростіше зберігати проміжні результати обчислень процедури в регістрах, але їх (регістрів) може бути замало. Найчастіше локальні змінні зберігаються у стеку після збереженого значення BP (рис.23). Модифікуємо попередній приклад і додамо в процедурі дві локальні змінні: одну для збереження суми переданих параметрів, а другу для різниці. Процедура набуде такий вигляд:

```
myproc      proc near
            push bp
            mov bp, sp
            ; зменшення sp на 4 байти –
        резервування місця для
            ;локальних змінних
            sub sp, 4
            mov ax, [bp+4]
            mov cx, ax
            mov bx, [bp+6]
            add ax, bx
            sub cx, bx
            mov [bp-2], ax
            mov [bp-4], cx
            . . .
            pop bp
            ret 4
myproc endp
```

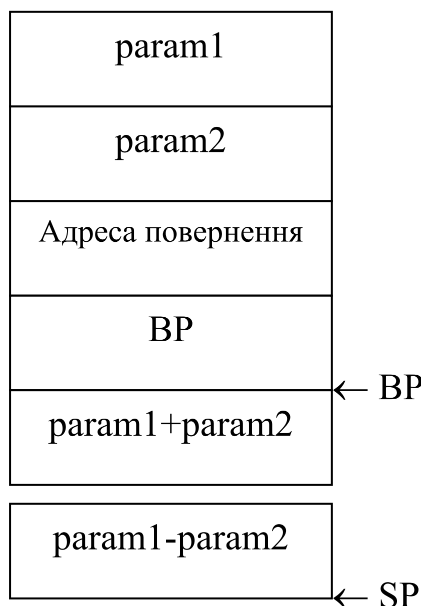

Рис. 23 Стек для процедури

Глобальні оголошення та загальні дані у програмах

Для оголошення змінних, констант, процедур та міток, що будуть використовуватися з іншого програмного модуля використовується директива PUBLIC:

PUBLIC < мова > < ім'я_мітки >

Якщо в одному модулі (unit1) необхідно використати процедуру, що була описана в іншому модулі (unit2) з директивою PUBLIC, то в unit1 її оголошують з директивою EXTERN.

Загальний опис директиви EXTRN має такий формат:

EXTERN <мова> <ім’я> : <тип> [,…]

<мова> - необов’язкова частина, опис якої аналогічний до опису мови у процедурі;

<ім’я> - ідентифікатор змінної, процедури і т.ін.

<тип> - може приймати значення BYTE, WORD, DWORD, FWORD, QWORD, TBYTE, FAR, NEAR, ABS або ім’я структури даних.

За допомогою директиви EXTRN можна визначити асемблеру, що посилання на SUBPROG має атрибут FAR, тобто вона визначена в іншому асемблерному модулі:

```
;оголошеняя в unit1
EXTERN SUBPROC: FAR
MAINPROG PROC
                …
               CALL SUBPROG
               …
               RET
MAINPROG ENDP
               …
```

```
; unit2
PUBLIC SUBPROG
       …
SUBPROG PROC
               …
               RET
SUBPROG ENDP
               …
```

Підпрограма SUBPROG описується з директивою PUBLIC, яка вказує асемблеру і компанувальнику, що інший модуль повинний знати адресу SUBPROG.

Компанувальник ставить відповідність між адресами EXTRN в одному об'єктному модулі з адресою PUBLIC в іншому.

Як приклад розглянемо процедуру, що обчислює вартість придбаного товару. Ціна товару зберігається в змінній PRICE, а кількість – QTY.

Основна програма.

```
PAGE 60,132
TITLE “CALLMUL1.EXE – виклик підпрограми множення”
EXTERN SUBMUL:FAR
;----------------------------------
STACKSG SEGMENT PARA STACK ‘stack’
DW 64 DUP (?)
STACKSG ENDS
;----------------------------------
DATASG SEGMEN PARA ‘DATA’
```

```
QTV    DW 0140H
PRICE DW 2500H
DATASG ENDS
;----------------------------------
CODESG SEGMENT PARA ‘Code’
BEGIN PROC FAR
   ASSUME CS:CODESG, DS:DATASG, SS:STACKSG

   PUSH DS
   SUB AX,AX
   PUSH AX
   MOV AX,DATASG
   MOV DS,AX
   MOV AX,PRICE        ;  в AX переслати ціну
   MOV BX,QTY          ;  в BX переслати кількість
   CALL SUBMUL         ;виклик підпрограми
   RET
BEGIN ENDP
CODESG ENDS
END BEGIN
```

Підпрограма.

```
PAGE 60,132
TITLE “SUBMUL – підпрограма для множення”
CODESG SEGMEN PARA ‘Code’
    SUBMUL PROC FAR
    ASSUME SG:CODESG
    PUBLIC SUBMUL
    MUL BX          ;  в AX ціна
                    ;  в BX кількість
                    ; добуток в DX:AX
    RET
SUBMUL ENDP
CODESG ENDS
END SUBMUL
```

Зверніть увагу, що в наведеному прикладі параметри для процедури передаються за значенням через регістри.

Наявність спільних даних передбачує можливість обробки в одному асемблер-модулі даних, що визначені в іншому ассемблер-модулі.

Змінимо попередній приклад так: змінні QTY и PRICE також визначаються в основній програмі, але завантаження значень цих змінних у регістри AX і BX виконуються в підпрограмі. До попередньої програми необхідно внести такі зміни:

1. У основній програмі ім'я QTY і PRICE визначаються як PUBLIC. Сегмент даних так само визначений з атрибутами PUBLIC.

```
…
EXTERN SUBMUL: FAR
PUBLIC QTY, PRICE
…
DATASG SEGMENT PARA ‘DATA’
… ; все інше залишається без змін
```

2. У підпрограмі імена QTY і PRICE визначені як EXTERN і WORD. Таке визначення вказує Ассемблеру на довжину цих полів у 2 байта.

Підпрограма.

```
    PAGE 60,132
    TITLE “SUBMUL  - підпрограма для добутку”
    EXTERN QTY:WORD, RPICE:WORD
    CODESG SEGMEN PARA PUBLIC ‘CODE’
    SUBMUL PROC FAR
       ASSUME CS:CODESG
       PUBLIC SUBMUL
       MOV AX,PRICE
       MOV BX,QTY
       MUL BX          ; добуток в DX:AX
       RET
    SUBMUL ENDP
    CODESG ENDS
END SUBMUL
```

Список літератури

4. Абель П. Язык Ассемблера для IBM PC и программирования: Пер. с англ. – М.: Высш. Шк., 1992. – 447 с.:ил.
5. Анпілогов П.І., Дехтярюк Д.Е., Діомідов О.О. Операційні системи та системне програмування: Навч. посібник. – К.:ІСДО, 1993. – 392 с.
6. Григорьев В.Л. Архитектура и программирование арифметического сопроцессора. – М. : Энергоатомиздат, 1991. – 208 с.: ил.

7. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование (в 4-х книгах). Книга 2. Аппаратная архитектура. - М., ГРАНАЛ, 1993.- с.346, ил. 87.
8. Гук М. Процессоры Pentium II, Pentium Pro и просто Pentium – СПб:ЗАО “Издательство “Питер”, 1999. – 288 с.: ил.
9. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT, AT: Пер. с англ. – М. Финансы и статистика, 1992. – 544 с.
10. Зубков С.В. Assembler. Для DOS, Windows и Unix. – М.:ДМК, 1999. – 640 с., ил.
11. Иртегов Д.В., Введение в операционные системы: Учебное пособие – СПб: БХВ – Петербург, 2002 г. – 624 с.
12. Использование Turbo Assembler при разработке программ – Киев: “Диалектика”, 1994. – 288 с., ил.
13. Методичні вказівки до виконання індивідуальних завдань по курсу “Системне програмування та операційні системи”/ Уклад. Красовська А.В. – К.: КНТУБА – 2001.
14. Методичні вказівки до лабораторних робіт по курсу “Системне програмування та операційні системи”/ Уклад. Демідов П.Г. – К.:КДТУБА – 1998.
15. Сван, Том . Освоение Turbo Assembler.- К.:Диалектика, 1996.-544 с.