МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Запорізький національний технічний університет

Н.В. Щербак, М.М. Хохлов

ТЕКСТИ (конспект) лекцій з дисципліни

Програмування на мові Асемблер
для студентів спеціальності 123 "Комп'ютерна інженерія"
усіх форм навчання

2018

Тексти (конспект) лекцій з дисципліни “Програмування на мові Асемблер” для студентів спеціальності 123 “Комп’ютерна інженерія” усіх форм навчання /Укл.: Н.В. Щербак, М.М. Хохлов. - Запоріжжя: ЗНТУ, 2018. – 82 с.

Укладачі: Н.В. Щербак, ст. викладач
М.М. Хохлов, ст. викладач

Рецензент: С.Ю. Скрупський, доцент, к.т.н.

Відповідальний
за випуск: Н.В. Щербак, ст. викладач

Затверджено
на засіданні кафедри
“Комп’ютерні системи
та мережі ”
Протокол № 9
від “17” травня 2018.

Рекомендовано до видання
НМК факультету комп’ютерних
наук і технологій
Протокол № 10
від “28” травня 2018.

ЗМІСТ

ТЕМА 1. Загальні відомості мови Асемблер	5
1.1 Поняття програмування низького рівня	5
1.2 Оперативна пам'ять. Регістри процесора	7
1.3 Подання даних	13
ТЕМА 2. Структура програми на мові Асемблер	17
2.1 Сегментація в асемблері	17
2.2 Директиви асемблера. Моделі пам'яті	19
2.3 Опис структури програми	26
ТЕМА 3. Пересилання даних	28
3.1 Команди пересилання даних	28
3.2 Оператор зазначення типу	31
3.3 Команди роботи зі стеком	32
3.4 Виведення і введення інформації	32
ТЕМА 4. Арифметичні операції	34
4.1 Команди додавання	34
4.2 Команди віднімання	37
4.3 Команди множення	39
4.4 Команди ділення	41
4.5 Переповнення	42
4.6 Команди перетворення даних	44
ТЕМА 5. Порозрядні операції	46
5.1 Логічні команди	46
5.2 Команди операцій над бітами	47
5.3 Команди зсуву	48
ТЕМА 6. Команди передачі управління. Цикли	51
6.1 Команди безумовної передачі керування	52
6.2 Команди передачі управління за умовою	53
6.3 Команди організації циклів	57
6.4 Обробка масивів	60
ТЕМА 7. Макрозасоби мови Асемблер	63
7.1 Основні поняття макрозасобів	63
7.2 Макровизначення	63
7.3 Особливості використання макросів	64
ТЕМА 8. Процедури	66
8.1 Визначення процедури	66

8.2 Розміщення і передача аргументів процедур 71
ТЕМА 9. Зв'язок з іншими мовами програмування високого рівня 75
9.1 Використання зовнішніх процедур і функцій 76
9.2 Викривлення імен 79
9.3 Виклик асемблерних процедур з C/C++ 79
Перелік джерел посилання 81

ТЕМА 1. Загальні відомості мови Асемблер

1.1 Поняття програмування низького рівня

Мова асемблера - це мова програмування низького рівня.

Мова програмування низького рівня - це мова програмування, максимально наближена до програмування в машинних кодах.

Мова програмування високого рівня - це мова програмування, максимально наближена до людської мови (зазвичай до англійської). Мова високого рівня практично не прив'язана ні до конкретного процесора, ні до операційної системи.

Програмування низького рівня - це програмування, засноване на прямому використанні можливостей і особливостей конкретного комп'ютера. Для того щоб писати програми на цьому рівні, необхідно знати архітектуру апаратної частини системи:

- структуру і функціонування системи в цілому;
- організацію оперативної пам'яті;
- склад зовнішніх пристроїв, їх адреси та формати регістрів;
- організацію та функціонування процесора, склад і формати його регістрів, способи адресації, систему команд;
- систему переривань, тощо.

При цьому не слід забувати, що обчислювальна система - це сукупність не тільки апаратних, але і програмних засобів, і особливості програмного забезпечення (операційної системи) мають значний вплив на розробку програм.

В історії розвитку програмування існували три різновиди мов низького рівня, що послідовно змінювали один одного:

- машинний код;
- мнемокод;
- асемблер.

Машинний код - система команд конкретної обчислювальної машини (процесора), яка інтерпретується безпосередньо процесором.

Процесор - це мозок комп'ютера. Фізично це спеціальна мікросхема з декількома сотнями виводів, яка вставляється в материнську плату.

Процесорів існує досить багато навіть в світі комп'ютерів. Але крім комп'ютерів ще є телевізори, пральні машини, кондиціонери і

т.п., де також дуже широко використовуються процесори (мікропроцесори, мікроконтролери).

Кожен процесор має свій набір регістрів.

Регістри процесора - це такі спеціальні комірки пам'яті, які знаходяться безпосередньо в мікросхемі процесора і використовуються для різних цілей.

Кожен процесор має свій набір команд. Команда процесора записується в певний регістр, і тоді процесор виконує цю команду.

Що таке команда з точки зору процесора?

Команда, як правило, представляє собою ціле число, яке записується в регістр процесора. Процесор читає це число і виконує операцію, яка відповідає цій команді. Сучасні процесори можуть мати кілька сотень команд. Запам'ятати все їх складно і для спрощення роботи програміста була придумана мова Асемблера, де кожній команді відповідає мнемонічний код.

Наприклад, число 4 відповідає мнемоніці ADD - це скорочення від слова ADDITION (додавання). Іноді мова асемблера ще називають мовою мнемонічних команд.

Що таке асемблер?

Саме слово ***асемблер*** (assembler) перекладається з англійської як «збирач». Насправді так називається програма-транслятор, яка приймає на вході текст, що містить умовні позначення машинних команд, зручні для людини, і переводить ці позначення в послідовність відповідних кодів машинних команд, зрозумілих процесору.

На відміну від машинних команд, їх умовні позначення запам'ятати порівняно легко, так як вони представляють собою скорочення від англійських слів. Мова умовних позначень і називається ***мовою асемблера***.

Таким чином, ***асемблер*** - це машино орієнтована мова програмування, що дозволяє працювати з комп'ютером безпосередньо, один на один. Звідси і його повне формулювання - мова програмування низького рівня.

Команди асемблера один в один відповідають командам процесора, але оскільки існують різні моделі процесорів зі своїм власним набором команд, то, відповідно, існують і різновиди, або діалекти, мови асемблера.

Тому використання терміна «мова асемблера» може викликати помилкову думку про існування єдиної мови низького рівня або стандарту на такі мови. Його не існує.

Тому при іменуванні мови, на якій написана конкретна програма, необхідно уточнювати, для якої архітектури вона призначена і на якому діалекті мови написана.

Насправді не так уже й важливо, мову якого саме асемблера вивчати. Головне - зрозуміти сам принцип роботи на рівні команд процесора, і тоді не важко буде освоїти не тільки інший асемблер, але і будь-який інший процесор зі своїм набором команд.

В даний час драйвера і операційні системи пишуть на Сі ++, але при всіх його достоїнствах - мова високого рівня приховує від програміста різні тонкощі і нюанси апаратної частини, а асемблер - мова низького рівня, прямо відображає всі ці тонкощі і нюанси.

Зазвичай програми або ділянки коду пишуться на мові асемблера в випадках, коли розробнику критично важливо оптимізувати такі параметри, як:

– розмір коду (обсяг використовуваної пам'яті) (програми-завантажувачі, вбудоване програмне забезпечення, програми для мікроконтролерів і процесорів з обмеженими ресурсами, віруси, програмні захисти, тощо);

– швидкодію (програми, написані на мові асемблера виконуються набагато швидше, ніж програми-аналоги, написані на мовах програмування високого рівня).

Більшість сучасних компіляторів дозволяють комбінувати в одній програмі код, написаний на різних мовах програмування. Це дає можливість швидко писати складні програми, використовуючи мову високого рівня, не втрачаючи швидкодії в критичних до часу завданнях, застосовуючи для них частини, написані на мові асемблера.

1.2 Оперативна пам'ять. Регістри процесора

Для розуміння мови асемблера необхідно уявляти собі найпростішу реєстрову модель процесора 8086 фірми Intel.

Об'єм оперативної пам'яті комп'ютера - 220 байтів (1 Мб).

Байти нумеруються починаючи з 0, номер байта називається його адресою.

Для посилання на байти пам'яті використовуються 20-розрядні адреси: від 00000 до FFFFF (у 16-річній системі).

Байт містить 8 бітів (розрядів), кожен з яких може приймати значення 1 або 0. Розряди нумеруються справа наліво від 0 до 7 (рис. 1.1):

Рисунок 1.1 – Нумерація розрядів в байті

Байт - це найменша комірка пам'яті, що адресується.

Використовуються і більш великі комірки - ***слова*** і ***подвійні слова***.

Слово - це два сусідніх байти, розмір слова - 16 бітів (вони нумеруються справа наліво від 0 до 15). Адресою слова вважається адреса його першого байта (з меншою адресою); ця адреса може бути парна і непарна.

Подвійне слово - це будь-які чотири сусідніх байти (два сусідніх слова), розмір такої комірки - 32 біти; адресою подвійного слова вважається адреса його першого байта.

Байти використовуються для зберігання невеликих цілих чисел і символів, слова - для зберігання цілих чисел і адрес, подвійні слова - для зберігання "довгих" цілих чисел і адрес в вигляді «сегмент: зсув».

Крім комірок оперативної пам'яті для короткочасного зберігання даних використовуються регістри.

Регістри - це спеціальні комірки пам'яті, фізично розташовані всередині процесора. На відміну від ОЗП, де для звернення до даних потрібно використовувати шину адреси, до регістрів процесор може звертатися безпосередньо. Це значно прискорює роботу з даними.

Для розуміння мови асемблера необхідно представляти найпростішу реєстрову модель процесора 8086 фірми Intel.

Всі регістри процесора 8086 фірми Intel мають розмір слова (16 бітів). Починаючи з моделі 80386 процесори Intel мають розмір подвійного слова (32 біта), за кожним з них закріплено певне ім'я.

За призначенням і способом використання регістри можна розбити на наступні групи:

– регістри загального призначення (**EAX**, **EBX**, **ECX**, **EDX**, **EBP**, **ESI**, **EDI**, **ESP**);

– покажчик команд (**EIP**);

– регістр прапорів (**Flags**);

– сегментні регістри (**CS**, **DS**, **SS**, **ES**).

Ім'я регістрів залежить від їх призначення, відповідно і маємо таку назву:

– **EAX**/**AX**/**AH**/**AL** (accumulator register) - акумулятор;
– **EBX**/**BX**/**BH**/**BL** (base register) - регістр бази;
– **ECX**/**CX**/**CH**/**CL** (counter register) - лічильник;
– **EDX**/**DX**/**DH**/**DL** (data register) - регістр даних;
– **ESI**/**SI** (source index register) - індекс джерела;
– **EDI**/**DI** (destination index register) - індекс приймача (одержувача);
– **ESP**/**SP** (stack pointer register) - регістр покажчика стека;
– **EBP**/**BP** (base pointer register) - регістр покажчика бази кадру стеку;
– **EIP**/**IP** (instruction pointer) - лічильник команд;
– **CS** (code segment) - сегмент команд;
– **DS** (data segment) - сегмент даних;
– **SS** (stack segment) - сегмент стеку;
– **ES** (extra segment) - додатковий сегмент.

До регістрів загального призначення відноситься група з 8 регістрів.

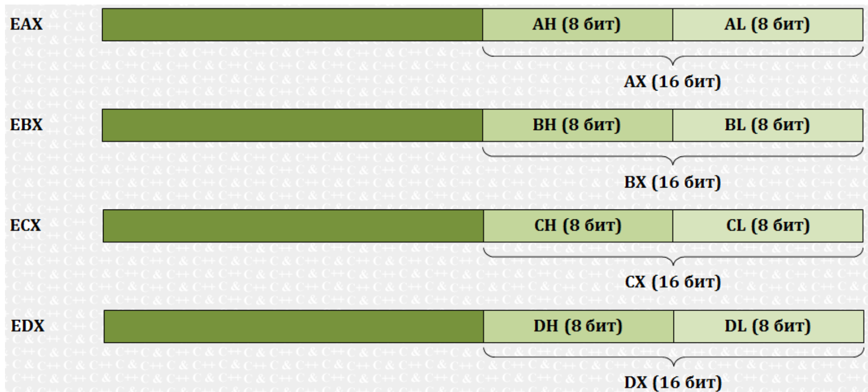


Рисунок 1.2 – Формат регістрів **EAX**, **EBX**, **ECX**, **EDX**

Регістри **EAX**, **EBX**, **ECX**, **EDX** - це регістри загального призначення, які мають певне призначення (акумулятор, регістр бази, лічильник, регістр даних), однак в них можна зберігати будь-яку інформацію.

Регістри **EAX**, **EBX**, **ECX** і **EDX** дозволяють звертатися як до молодших 16 бітам (за іменами **AX**, **BX**, **CX** і **DX**), так і до двом молодшим байтам окремо (за іменами **AH**/**AL**, **BH**/**BL**, **CH**/**CL** і **DH**/**DL**), (**H** - high, старший; **L** - low, молодший) (рис. 1.2).

Нумерація розрядів в регістрах **EAX**, **EBX**, **ECX** і **EDX** показана на рисунку 1.3.

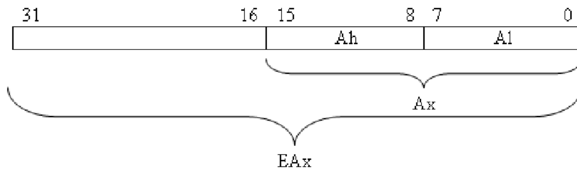


Рисунок 1.3 – Нумерація розрядів в регістрі **EAX**

Регістри **EBP**, **ESP**, **ESI**, **EDI** - це також регістри загального призначення, які мають вже більш конкретне призначення і дозволяють звертатися до молодших 16 бітам по іменах **BP**, **SP**, **SI** і **DI** відповідно (рис. 1.4). Нумерація розрядів в регістрах **EBP**, **ESP**, **ESI**, **EDI** показана на рисунку 1.5.

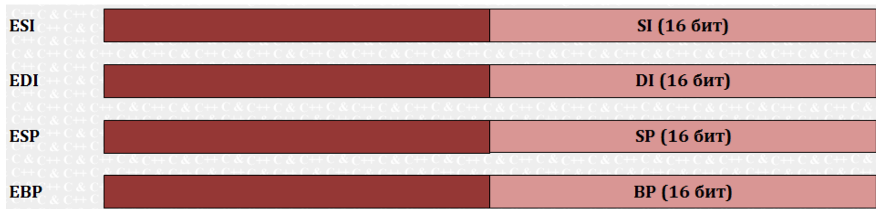


Рисунок 1.4 – Формат регістрів **ESI**, **EDI**, **ESP**, **EBP**

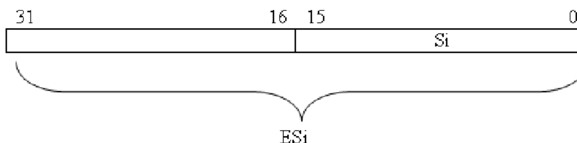


Рисунок 1.5 Нумерація розрядів в регістрі **ESI**

Регістр **EIP** (покажчик команд) містить зміщення наступної команди, яка підлягає виконанню. Цей регістр безпосередньо недоступний програмісту, але завантаження і зміна його значення виконується різними командами управління, до яких відносяться команди умовних і безумовних переходів, виклику процедур і повернення з процедур.

Процесор має спеціальний ***регістр прапорів***.

Прапор - це біт, що приймає значення "1", якщо виконана деяка умова, і значення "0" в іншому випадку. У процесорі i8086 використовуються 9 прапорів, кожному з них присвоєно певне ім'я. Всі вони зібрані в регістрі прапорів, тобто кожен прапор - це один з розрядів регістра. Структура регістра прапорів показана на рис. 1.6.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

Рисунок 1.6 – Регістр прапорів

Умовно прапори можна розділити на:

– прапори стану;

– прапори управління.

Прапори стану автоматично змінюються при виконанні команд і фіксують ті чи інші властивості їх результату, наприклад, чи було переповнення або дорівнює чи результат нулю.

Прапори станів встановлюються програмою і визначають подальшу роботу процесора, наприклад, блокують переривання.

Прапори умов (біти 0, 2, 4, 6, 7 і 11) відображають результат виконання арифметичних інструкцій, таких як **ADD**, **SUB**, **MUL**, **DIV**:

– **CF** (carry flag) - ***прапор переносу***. Приймає значення 1, якщо операція справила перенесення з старшого біта результату. Старшим є 7-й, 15-й, 31-й біти в залежності від розмірності операнда. Це відбувається, наприклад, якщо при додаванні цілих чисел результат вийшов за межі розрядної сітки, або якщо при відніманні чисел без знака перше з них було менше другого. У командах зрушення в **CF** заноситься біт, що вийшов за розрядну сітку;

– **PF** (parity flag) - ***прапор парності***. Дорівнює 1, якщо результат чергової команди містить парну кількість двійкових одиниць. Враховується зазвичай при операціях введення-виведення;

– **AF** (auxiliary carry flag) - ***допоміжний прапор переносу***. Фіксує особливості виконання операцій над двійковій-десятковими числами (binary coded decimal, BCD);

– **ZF** (zero flag) - ***прапор нуля***. Дорівнює 1, якщо результат операції виявився рівним 0;

– **SF** (sign flag) - ***прапор знака***. Встановлюється в 1, якщо в операції над знаковими числами вийшов негативний результат;

– **OF** (overflow flag) - ***прапор переповнення***. Встановлюється в 1, якщо в результаті операції відбувається перенос в старший, знаковий біт (7-й, 15-й, або 31-й). Встановлюється в 1, якщо відбувається позика з цих розрядів. Цей прапор необхідний для роботи з числами зі знаком.

Прапори стану дозволяють одній і тій же арифметичній інструкції видавати результат трьох різних типів:

– беззнаковий (прапор **CF** показує умову переповнення (перенесення або позику));

– знаковий (прапор **OF** показує умову переповнення (перенесення або позику));

– двійково-десяткове (BCD) ціле число (прапор **AF** показує умову переповнення (перенесення або позику)).

Прапор **SF** відображає знак знакового результату.

Прапор **ZF** відображає і беззнаковий, і знаковий нульовий результат.

До прапорам управління (біти 8, 9, 10) відносяться:

– **DF** (direction flag) - ***прапор напрямку***. Встановлює напрямок перегляду рядків у строкових командах: при **DF** = 0 рядки проглядаються від початку до кінця, при **DF** = 1 - в зворотному напрямку;

– **IF** (interrupt flag) - ***прапор переривань***. При **IF** = 0 процесор перестає реагувати на переривання, які до нього поступають, при **IF** = 1 блокування переривань знімається;

– **TF** (trap flag) - ***прапор трасування***. При **TF** = 1 після виконання кожної команди процесор робить переривання (з номером 1), що використовується при налагодженні програми для її трасування (виконання по кроках).

Сегментні регістри **CS**, **DS**, **SS** і **ES** не можуть бути операндами ніяких команд, крім команд пересилання і команд стеку. Ці регістри використовуються тільки для сегментування адрес:

- **CS** (code segment) - сегмент команд;
- **DS** (data segment) - сегмент даних;
- **SS** (stack segment) - сегмент стеку;
- **ES** (extra segment) - додатковий сегмент.

1.3 Подання даних

Розглянемо подання цілих чисел, рядків і адресів в процесорі i8086. Дійсні числа процесором i8086 не обробляються, операції над цими числами реалізуються програмним шляхом або виконуються співпроцесором.

У програмі на мові асемблера цілі числа можуть бути записані в двійковій, восьмирічній, десятковій і шістнадцятирічній системах числення. Для завдання системи числення в кінці числа ставиться літера **b**, **o**/**q**, **d** або **h** відповідно.

Шістнадцятирічні числа, які починаються з «буквеної» цифри, повинні передувати нулем, інакше компілятор не зможе відрізнити число від ідентифікатора.

У загальному випадку під ціле число можна відвести будь-яке число байтів, проте система команд процесора i8086 підтримує числа розміром в байт, слово і подвійне слово.

Розрізняються цілі числа без знака і знакові цілі числа.

У таблиці 1.1 наведені діапазони можливих значень цілих позитивних чисел, з якими може працювати процесор сімейства Intel.

Таблиця 1.1 - Діапазони можливих значень позитивних чисел

Тип числа	Діапазон значень	Ступені двійки
Байт	0 … 255	0 … (2^8 - 1)
Слово	0 … 65535	0 … (2^{16} - 1)
Подвійне слово	0 … 4294967295	0 … (2^{32} - 1)

Числа записуються в двійковій системі числення, займаючи всі розряди комірки.

100_{10} = 64h = 0110 0100b (байт)

150_{10} = 96h = 1001 0110b (байт)

404_{10} = 194h = 000 0001 1001 0100b (слово)
612333_{10} = 957EDh = 0000 0000 1001 0101 0111 1110 1101b (подвійне слово)

Слід відразу зрозуміти особливості зберігання слів і подвійних слів в пам'яті комп'ютера. Числа розміром в слово зберігаються в пам'яті в "перевернутому" вигляді: молодші (праворуч) 8 бітів числа розміщуються в першому байті слова, а старші 8 бітів - у другому байті (в 16-річної системі: дві праві цифри - в першому байті, дві ліві цифри - у другому байті). Наприклад, число 30010 = 012Ch у вигляді слова зберігається в пам'яті так (рис. 1.7):

Рисунок 1.7 – Зберігання в пам'яті числа 30010 = 012Ch

Але, якщо це слово завантажити в регістр **AX**, то в представлення його в регістрі буде звичайним (рис. 1.8).

Рисунок 1.8 – Представлення в регістрі числа 30010 = 012Ch

Аналогічно зберігаються і числа в форматі подвійного слова. У першому його байті розміщуються молодші 8 бітів числа, у другому байті - попередні 8 бітів і т.д. Наприклад, число 12345678h зберігається в пам'яті так (рис. 1.9):

Рисунок 1.9 – Зберігання в пам'яті числа 12345678h

У таблиці 1.2 наведені діапазони можливих значень цілих знакових чисел.

Таблиця 1.2 - Діапазони можливих значень знакових чисел

Тип числа	Діапазон значень	Ступені двійки
Байт	-128 … +127	$-2^7 \ldots (2^7 - 1)$
Слово	-32768 … +32767	$-2^{15} \ldots (2^{15} - 1)$
Подвійне слово	-2147483648 … +2147483647	$-2^{31} \ldots (2^{31} - 1)$

При цьому числа записуються в додатковому коді: невід'ємне число записується так само, як і беззнаковое число (тобто в прямому коді).

Наприклад, додатковим кодом числа -10 є байт F6h (256-10), слово FFF6h або подвійне слово FFFFFFF6h.

Таким чином, самий лівий біт інтерпретується як знаковий, якщо він дорівнює 1, то число вважається негативним, якщо 0 - позитивним.

Якщо знакове ціле число формату байт містить одиницю тільки в знаковому розряді, то воно інтерпретується як -128. Аналогічно для слова це буде -32768, для подвійного слова -2147483648.

Знакові числа розміром в слово і подвійне слово записуються в пам'яті також в "перевернутому" вигляді і знаковий біт виявляється в останньому байті комірки. Наприклад, число -30010 = FED4h у вигляді слова зберігається в пам'яті так (рис. 1.10):

A	A+1
D4	FE

Рисунок 1.10 – Зберігання в пам'яті числа -30010 = FED4h

Число -1234567810 = FF439EB2h у вигляді подвійного слова зберігається в пам'яті так (рис. 1.11):

A	A+1	A+2	A+3
B2	9E	43	FF

Рисунок 1.11 – Зберігання в пам'яті числа -12345678_{10}=FF439EB2h

Символи і рядки. Під символ відводиться один байт пам'яті, в який записується код символу - ціле від 0 до 255. У IBM-сумісних комп'ютерах використовується система кодування ***ASCII*** (American Standard Code for Information Interchange).

Деякі особливості цієї системи кодування:

– код символу «прогалина» менше коду будь-якого літерного символу, цифри та інших графічно представлених символів;

– коди цифр впорядковані за величиною цифр і не містять пропусків - це від 30h до 39h;

– коди великих і малих латинських букв упорядковані відповідно до алфавіту і не містять пропусків.

Рядок - це послідовність символів, яка розміщується в сусідніх байтах пам'яті, код першого символу рядка записується в першому байті, код другого символу - у другому байті і т.д. Адресою рядка вважається адреса її першого байта.

Подання адрес. Адреса - це порядковий номер комірки пам'яті, тобто невід'ємне ціле число, тому в загальному випадку адреси представляються так само, як і беззнакові числа.

Часто під адресою розуміється 16-бітовий ***зсув*** (offset) - адреса комірки, відрахований від початку сегмента (області) пам'яті, якому належить ця комірка. В цьому випадку під адресу відводиться слово пам'яті.

В іншому випадку під адресою розуміється 20-бітова абсолютна адреса деякої комірки пам'яті. Така адреса задається як пара «***сегмент:зсув***», де ***сегмент*** (segment) - це перші 16 бітів початкової адреси сегмента пам'яті, якому належить комірка, а зсув - 16-бітова адреса цієї комірки, відрахований від початку даного сегмента пам'яті.

Абсолютний (реальний) адреса утворюється як сегмент*16 + зсув. Така пара записується в пам'яті у вигляді подвійного слова: в першому слові розміщується зсув, а в другому - сегмент (перевернутий вигляд), причому кожне з цих слів в свою чергу представлено в перевернутому вигляді.

Наприклад, пара 1234h: 5678h буде записана так (рис. 1.12):

Смещение		Сегмент	
78	56	34	12

Рисунок 1.12 – Зберігання в пам'яті адреси у вигляді пари «*сегмент*:*зсув*» 1234h:5678h

Для отримання реального адреси потрібно:

$1234h = 4660_{10}$

$5678h = 22136_{10}$

$4660 * 16 + 22136 = 96696_{10} = 179B8h$

або

$12340h + 5678h = 179B8h$

Відповідно різні пари «***сегмент:зсув***» можуть вказувати на одну і ту ж комірку пам'яті, наприклад пара 11FFh:59C8h вказує на комірку пам'яті з реальним адресою 9669610 = 179B8h так, як:

$11FF0h + 59C8h = 179B8h$

На рисунку 1.13 показана адресація до однієї і тієї ж комірки пам'яті різними способами.

```
C:\windows\system32\debug.exe
-d 0011:0000
0011:0000  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0011:0010  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0011:0020  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0011:0030  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0011:0040  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0011:0050  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0011:0060  28 05 11 E2 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   (...=!..=!..=!..
0011:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-d 0000:0110
0000:0110  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0000:0120  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0000:0130  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0000:0140  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0000:0150  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0000:0160  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0000:0170  28 05 11 E2 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   (...=!..=!..=!..
0000:0180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-d 0010:0010
0010:0010  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0010:0020  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0010:0030  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0010:0040  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0010:0050  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0010:0060  3D 21 00 F0 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   =!..=!..=!..=!..
0010:0070  28 05 11 E2 3D 21 00 F0-3D 21 00 F0 3D 21 00 F0   (...=!..=!..=!..
0010:0080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

Рисунок 1.13 – Адресація до комірки пам'яті

ТЕМА 2. Структура програми на мові Асемблер

2.1 Сегментація в асемблері

Будь-яка програма, написана на мові асемблера, складається з одного або декількох сегментів.

Сегмент - це область пам'яті, зайнята командами або даними, адреси яких обчислюються щодо значення у відповідному сегментному регістрі.

Область пам'яті, в якій знаходяться команди, називається ***сегментом коду***, область з даними - ***сегментом даних*** і область пам'яті, відведена під стек - ***сегментом стека***.

Відзначимо деякі тонкощі сегментування.

Директива сегментації - це адресне вираз, тобто після створення програми ім'я сегмента представлятиме перші шістнадцять бітів початкової адреси даного сегмента.

При асемблюванні всі пропозиції (дані і команди) будуть відраховуватися від цієї адреси. Наприклад, ім'я сегмента даних може мати значення 1000h, ім'я сегмента коду - 1040h і т.д.

Асемблер все пропозиції (наприклад дані) в програмі має на увазі як пару виду «***імя_сегмента:зсув***». Якщо в кожній команді

записувати адреси даних в такому вигляді, то пам'ять для програми буде використовуватися не зовсім раціонально.

Отже, оскільки значення сегмента в ході виконання програми не змінюється, то його досить десь зберігати в одному місці і кожного разу використовувати при створенні фізичних адрес даних і команд. Для цього і використовуються сегментні регістри.

Стеком називають область програми для тимчасового зберігання довільних даних.

Зрозуміло, дані можна зберігати і в сегменті даних, проте в разі якщо дані треба тимчасово зберігати, то заводити окремий іменований елемент пам'яті не доцільно так, як збільшується розмір програми і кількість імен, що використовуються.

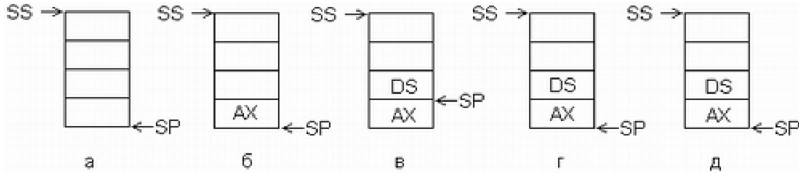
Зручність стека полягає в тому, що його область використовується багато разів, причому збереження в стеку даних і вибірка їх звідти виконується за допомогою ефективних команд **push** і **pop** без зазначення будь-яких імен.

Роботу зі стеком процесор організовує за принципом ***Last-In-First-Out*** - останнім увійшов першим вийшов (***LIFO***). У пам'яті стек "росте" в напрямку зменшення адрес пам'яті.

Елементи стека розташовуються в області пам'яті, відведеної під стек, за адресами починаючи з дна стека (тобто з його максимального адреси) і послідовно зменшуються. Адреса верхнього доступного елемента зберігається в регістрі-покажчику стека **SP**. Сегментна адреса цього сегмента поміщається в сегментний регістр стека **SS**.

Таким чином, пара регістрів **SS:SP** описують адресу доступного організації стека: в **SS** зберігається сегментна адреса стеку, а в **SP** - зсув останнього збереженого в стеку даного (рис. 2.1, а). Зверніть увагу на те, що в початковому стані покажчик стека **SP** вказує на комірку, що лежить під дном стеку і не входить до нього.

Стек може використовуватися для зберігання адреси повернення з підпрограм, динамічних змінних, коли через стек здійснюється передача змінних у функції і процедури.



а) - початковий стан; б) - після завантаження одного елемента (в даному прикладі - вмісту регістра **AX**); в) - після завантаження другого елементу (вмісту регістра **DS**); г) - після вивантаження одного елемента; д) - після вивантаження двох елементів і повернення в початковий стан.

Рисунок 2.1 – Організація стеку

2.2 Директиви асемблера. Моделі пам'яті

Директива - команда, яка виконується транслятором під час обробки програми.

Директиви асемблера можна класифікувати наступним чином (рис. 2.2):

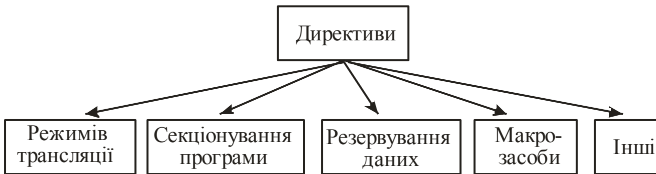


Рисунок 2.2 – Класифікація директив асемблера

Директиви режимів трансляції визначають глобальні особливості трансляції, які повинні враховуватися при обробці тексту програми (рис. 2.3).

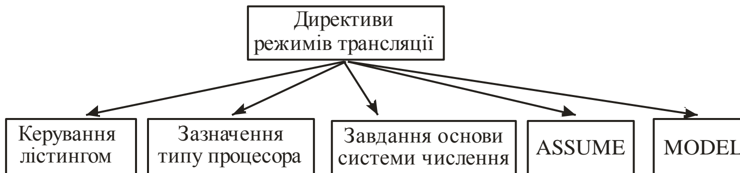


Рисунок 2.3 – Директиви визначення загального режиму трансляції

Директиви управління лістингом визначають, що слід включати в вміст файлу лістингу. Прикладами директив цієї групи є:

– **.LIST** - включити в лістинг усі рядки вихідного тексту програми;

– **.NOLIST** - виключити з лістингу вихідний текст;

– **.SUMS** - включити в лістинг таблицю символів;

– **.NOSUMS** - виключити з лістингу таблицю символів і т.д.

Директиви вказівки типу процесора визначають, які регістри процесора, режими адресації, команди дозволено використовувати в програмі. Приклади цих директив:

– **.8086** - дозволено використовувати апаратуру мікропроцесора 8086;

– **.80286** - те ж саме, але мікропроцесор 80286;

– **.80286P** - дозволяє використання привілейованих команд мікропроцесора 80286;

– **.80386**;

– **.80386P** і т.д.

Основа системи числення визначає, як будуть інтерпретуватися числа за замовчуванням. Для цих цілей використовуються директиви .RADIX:

– **.RADIX 2** - як виконавчі;

– **.RADIX 8** - як восьмеричні;

– **.RADIX 10** - як десяткові;

– **.RADIX 16** - як шістнадцятирічні.

Директиви **ASSUME** фактично визначають точки програми, від яких транслятор відраховує зсув до міток, які використовуються в програмі в якості символічних адрес.

Директива **MODEL** найчастіше використовується спільно з директивами спрощеної сегментації, які будуть розглядатися далі. Вона визначає, як будуть розташовані сегменти по відношенню один до одного, яким чином будуть адресуватися дані, як будуть викликатися підпрограми.

Директиви секціонування програми призначені для оформлення логічно закінчених ділянок програми (рис. 2.4).

Сегменти програми можуть описуватися директивами спрощеної сегментації:

– **.code** - відкриває опис сегмента кодів команд;

– **.data** - починає опис сегмента даних;

– **.stack** - описує сегмент стеку, в якості аргументу цієї директиви може вказуватися розмір стека в байтах.

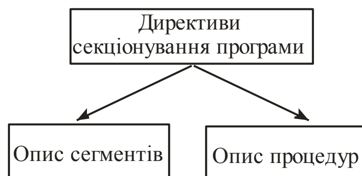


Рисунок 2.4 – Директиви секціонування програми

При використанні директив спрощеної сегментації обов'язково повинна передувати директива **.MODEL**. Директива **.MODEL** визначає модель пам'яті, яку використовує програма (табл. 2.1).

Таблиця 2.1 - Параметри моделей пам'яті

Модель пам'яті	Адресація коду	Адресація даних	Операційна система	Чергування коду та даних
TINY	NEAR	NEAR	MS-DOS	Допустиме
SMALL	NEAR	NEAR	MS-DOS, Windows	Ні
MEDIUM	FAR	NEAR	MS-DOS, Windows	Ні
COMPACT	NEAR	FAR	MS-DOS, Windows	Ні
LARGE	FAR	FAR	MS-DOS, Windows	Ні
HUGE	FAR	FAR	MS-DOS, Windows	Ні
FLAT	NEAR	NEAR	Windows NT, Windows 2000, Windows XP, Windows 2003	Допустиме

Розглянемо деякі з цих параметрів:

– **Tiny** - код, дані та стек містяться в одному і тому ж сегменті (near) розміром до 64 Кбайт. Використовується для написання невеликих програм (загалом com-програм);

– **Small** - код розміщується в одному сегменті (near), дані і стек - в іншому (near). Використовується для невеликих і середніх exe-програм;

– **Medium** - код може займати кілька сегментів (far). Під дані відводиться один сегмент (near). Використовується для написання великих програм з невеликим обсягом даних;

– **Compact** - розмір коду обмежений одним сегментом (near). Розмір даних необмежений (far);

– **Large** і **Huge** - код і дані можуть займати кілька сегментів (far);

– **Flat** - код, дані та стек містяться в одному і тому ж сегменті (near).

Модель **tiny** працює тільки в 16-розрядних додатках MS-DOS. У цій моделі всі дані і код розташовуються в одному фізичному сегменті. Розмір програмного файлу в цьому випадку не перевищує 64 Кбайт. З іншого боку, модель **flat** передбачає несегментовану конфігурацію програми і використовується тільки в 32-розрядних операційних системах. Ця модель подібна моделі **tiny** в тому сенсі, що дані і код розміщені в одному сегменті, тільки 32-розрядному.

При використанні директиви **.MODEL** транслятор робить доступними кілька ідентифікаторів, які можна використовувати в програмі для отримання фізичної адреси сегмента, зокрема:

– **@code** - фізичну адресу сегмента коду;

– **@data** - фізичну адресу сегмента даних;

– **@stack** - фізичну адресу сегмента стеку.

У випадках, коли необхідно використовувати кілька окремих сегментів даних і/або коду, то сегменти програми описуються за допомогою директив **SEGMENT** і **ENDS**, і може бути описана наступним чином:

```
ім'я_сегменту SEGMENT тип_вирівнювання клас_сегменту
...
ім'я_сегменту ENDS
```

де **ім'я_сегменту** - мітка, яка використовується для отримання сегментної адреси;

тип_вирівнювання може бути **byte**, **word**, **para**, **page** і інші, що визначають з якої межі буде починатися сегмент - без вирівнювання, за адресою, кратному двом байтам, слову, параграфу (16) або сторінці (256);

клас_сегменту - це укладений в лапки рядок, що повідомляє компілятору порядок проходження сегментів при складанні. Сегменти з однаковими іменами класу сегменту будуть розташовані в виконавчій програмі один за іншим.

За замовчуванням значення типу вирівнювання встановлюється рівним значенню **para**.

Директива **SEGMENT** може застосовуватися з будь-якою моделлю пам'яті.

При використанні директиви **SEGMENT** потрібно вказати компілятору в якості якого сегмента використовувати описані сегменти. Це можна зробити за допомогою директиви **ASSUME**:

ASSUME регістр:імя_сегмента

За допомогою директиви **ASSUME** асемблеру повідомляється який з сегментних регістрів з яким ім'ям сегмента асоціювати. Зверніть увагу директива **ASSUME** не завантажує фактичні адреси сегментів в регістри, а тільки лише повідомляє асемблеру де брати адреса сегмента при утворенні пар «сегмента:зсув».

Щоб в сегментних регістрах виявилися фактичні адреси, їх необхідно завантажити. Зазвичай на початку програми здійснюється завантаження сегментних регістрів **ds** і **es**. Регістр **cs** завантажувати не треба, це робить сама операційна система. Регістр **ss** можна завантажити явно як і регістри **ds** і **es**, також може завантажити і операційна система, якщо в директиві **SEGMENT** вказати спеціальний параметр **stack**.

Директиви резервування даних. Для резервування комірок пам'яті для змінних і констант та їх ініціалізації в мові асемблера використовуються директиви визначення даних:

– **DB** - визначає дані розміром в ***байт***;
– **DW** - визначає дані розміром в ***слово***;
– **DD** - визначає дані розміром в ***подвійне слово***.

За допомогою директив **DB**, **DW** і **DD** можуть описуватися одна або кілька змінних та присвоювання їм імен. З цій директиві асемблер формує машинне подання значень змінних і записує їх в чергові комірки пам'яті. Адреси цих комірок стають значеннями імен змінних, тобто всі входження імені в програму асемблер буде замінювати на відповідну цьому імені адресу. Імена, зазначені в директивах **DB**, **DW** і **DD**, називаються іменами змінних. Приклади:

```
ADB    162        ; Виділити пам'ять розміром 1 байт з
                  ; ім'ям A та занести 162
CDW    -1         ; Виділити пам'ять розміром 2 байти
```

```
                          ; (слово) з ім'ям C і занести -1
EDD    -10                ; Виділити пам'ять розміром 4 байта
                          ; (1двойное слово) з ім'ям E і занести -10
```

Дані в директиві **DB** можуть описуватися як ***числа***, так і як ***символи***: вказується або код символу (ціле від 0 до 255), або сам символ в лапках (одинарних або подвійних); в останньому випадку асемблер сам замінить символ на його код.

Наприклад, наступні директиви еквівалентні:

```
Star   DB    02Ah   ; Виділити пам'ять розміром для
                    ; символу
Star   DB    '*'    ; (1 байт) з ім'ям Star і занести символ
Star   DB    "*"    ; «*», ASCII-код якого - 2A
```

Якщо адресу необхідно представити як дані, то це робиться так:

```
Star        DB    '*'
adr_star    DW    star
```

У цій директиві відведено 2 байта (слово) пам'яті, якому дається ім'я **adr_star** і в яке запишеться адреса (зсув, який міститься в імені **star**.

Якщо для аналогічної мети використовується директива **DD** асемблер автоматично додасть до зміщення імені його сегмент і запише зсув в першу половину подвійного слова, а сегмент - в другу половину:

```
fadr_star   DD    star
```

За будь-якої з директив **DB**, **DW** і **DD** можна описати змінну, тобто відвести комірку, не давши їй початкового значення, тобто зарезервувати комірку(и) пам'яті. В цьому випадку в правій частині директиви вказується знак питання:

```
Perem  DW    ?      ; виділити 2 байта (слово), привласнити
                    ; йому ім'я Perem і нічого в це слово не
                    ; записувати
```

В одній директиві можна описати відразу кілька змінних одного і того ж розміру, для цього їх треба перерахувати через кому. Вони розміщуються в сусідніх комірках пам'яті. Приклад:

```
Betta  DB    200,-5,10h,?,'F'
```

Ім'я, вказане в директиві, є ім'ям першого з значень. Для посилань на інші використовуються вирази виду:

<имя>+<целое>

Наприклад, для доступу до байту з числом -5 використовується вираз **betta+1**, для доступу до байту з значенням 10h - вираз **betta+2** і т.д.

Індексація починається з нуля.

Якщо в директиві **DB** перераховані тільки символи, тоді цю директиву можна записати коротше, уклавши всі ці символи в одні лапки, наприклад:

```
Str    DB    'a','b','c'
Str    DB    'abc'
Str    DB    "abc"
```

Якщо в директиві описується кілька однакових змінних, то можна скористатися оператором повторення **DUP**. Наприклад:

```
mas    db    5 dup (4)
```

що еквівалентно директиві

```
mas    db    4,4,4,4,4
```

Інший приклад:

```
arr    DW    3 dup (?), -50, 2 dup (7)
```

що еквівалентно директиві

```
arr    DW    ?,?,?,-50,7,7
```

В асемблері є директиви **EQU** і =, за допомогою яких можна визначити ***константи***.

Директива **EQU** привласнює імені значення, яке визначається як результат цілочисельного виразу. Директива **EQU** аналогічна директиві **#define** в мові Сі. Значення, присвоєне ім'я за допомогою директиви **EQU**, не можна надалі змінити. Приклад:

```
Aequ   10
Bequ   21/3
Cequ   "abcdef"
```

Директива = схожа на директиву **EQU**, але значення, присвоєне ім'я повинно бути цілим числом і його можна перевизначити. Наприклад:

```
alfa=20
alfa=alfa+1
```

Для посилань на поточну комірку використовується позначення $, яке є позначенням лічильника поточного адреси. Приклад:

```
mas    db    "assembler"
mas_len=$-mas
```

У цьому прикладі значенням імені **mas_len** буде довжина рядка **mas**, тобто число 9.

Директиви опису процедур. Для опису процедур в асемблері передбачені директиви **proc** і **endp**. Детальний розгляд процедур дається далі.

Директиви макрозасобів. Директиви макрозасобів дозволяють при написанні програми оперувати заздалегідь заготовленими фрагментами текстів. Детальний розгляд макрозасобів наводиться в подальших лекціях, тому поки обмежимося їх класифікацією (рис. 2.5).

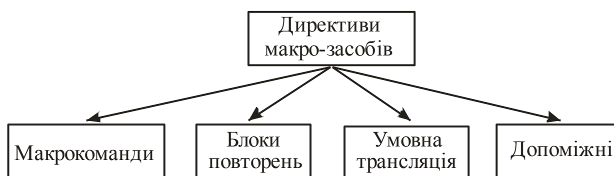


Рисунок 2.5 – Класифікація директив макрозасобів

2.3 Опис структури програми

Проста програма з використанням спрощених директив виглядає так:

```
title Програма       ; дає модулю ім’я Програма
model   small        ; ця директива вказується до
                     ; оголошення сегментів
.stack 100h          ; розмір стеку 256 байт
.data                ; начало сегменту даних
. . .                ; дані
.code                ; начало сегменту коду
start:
mov     ax,@data     ; в ds розміщується адреса сегменту
mov     ds,ax        ; даних вказаного директивою .data
                     ; за допомогою регістру ax
. . .                ; програмний код
end     start
end
```

Регістр **ds** необхідно спеціально ініціалізувати. Це пов'язано з тим, що після завантаження програми в пам'ять, він містить адресу

початку образу програми в пам'яті, який починається з префікса програмного сегмента (PSP), створюваного операційною системою, який дорівнює 256 байт. Після цієї області починається власне сегмент коду і операційна система знає цю адресу і заносить його в регістр сегмента коду. Слідом за сегментом коду розташовується сегмент даних і фізичну адресу початку цього сегмента необхідно завантажити в регістр **ds**. Оскільки передача безпосередніх значень в сегментні регістри заборонена, то це здійснюється зазвичай через регістри загального призначення **ax** або **bx**.

Для 32-розрядних додатків шаблон вихідного тексту виглядає інакше:

```
.model      flat
.stack
.data
. . .                       ; дані
.code
main:
. . .                       ; команди асемблера
end main
end
```

Основна відмінність від попереднього прикладу - інша модель пам'яті (**flat**), що припускає 32-розрядну лінійну адресацію з атрибутом near.

При використанні директиви **SEGMENT** структура програми виглядає наступним чином:

```
title Структура програми
data   segment    para  'data'
abc    db    (?)
data   ends
stk    segment    stack
db     256   dup (?)
stk    ends
code   segment    para  'code'
start:
assume cs:code, ds:data, ss:stk
mov    ax,data
mov    ds,ax
. . . . .
. . . . .
. . . . .
```

```
mov    ax,4c00h
int 21h
code         ends
end          start
```

Директивою **end** завершується будь-яка програма на мові асемблера. В якості операнда зазвичай виступає мітка, яка визначає адресу команди, з якою слід почати виконання програми.

Сегмент стека рекомендується завжди описувати (створювати), навіть якщо в програмі він явно не використовується. В цьому випадку стек повинен мати не менше 128 байт. Якщо стек використовується в програмі, то крім використовуваного обсягу стеку слід додатково резервувати ще 128 байт.

ТЕМА 3. Пересилання даних

3.1 Команди пересилання даних

Команди мови асемблера - це символьна форма запису машинних команд. Команди мають наступний синтаксис:

```
[<Мітка>:] <мнемокод> [<операнди>] [;<коментар>]
```

Мітка - це ім'я. Мітка обов'язково повинна відділятися двокрапкою, але може розміщуватися окремо, в рядку, що передує іншій частині команди. Мітки потрібні для посилань на команди з інших місць, наприклад, в командах переходу. Компілятор мови асемблера замінює мітки адресами команд.

Мнемокод - це службове слово, яке вказує операцію, яка повинна бути виконана.

Операнди команди, якщо вони є, відокремлюються один від одного комами. В якості операндів можуть використовуватися:

– регістри, звернення до яких здійснюється по іменах;

– безпосередні операнди - константи, що записуються безпосередньо в команді;

– комірки пам'яті - в команді записується адреса потрібної комірки.

Для завдання адреси існують такі можливості:

– ім'я змінної, по суті, є адресою цієї змінної. Зустрівши ім'я змінної в операндах команди, компілятор розуміє, що потрібно звернутися до оперативної пам'яті за певною адресою. Зазвичай адреса в команді вказується в квадратних дужках, але ім'я змінної є винятком

і може бути зазначено як в квадратних дужках, так і без них. Наприклад, для звернення до змінної **X** в команді можна вказати **X** або [**X**];

– якщо змінна була оголошена як масив, то до елементу масиву можна звернутися, вказавши ім'я та зсув. Для цього існує ряд синтаксичних форм, наприклад: **<ім'я> [<зсув>]** і **[<ім'я> + <зсув>]**. Однак слід розуміти, що зсув - це зовсім не індекс елемента масиву. Індекс елемента масиву - це його номер, і цей номер не залежить від розміру самого елемента. Зсув же задається в байтах, і при завданні зсуву програміст сам повинен враховувати розмір елемента масиву;

– адреса комірки пам'яті може зберігатися в регістрі. Для звернення до пам'яті за адресою, що зберігається в регістрі, в команді вказується ім'я регістра в квадратних дужках, наприклад: [**ebx**]. В якості регістрів бази рекомендується використовувати регістри **EBX** (**BX**), **ESI** (**SI**), **EDI** (**DI**) і **EBP** (**BP**);

– адреса може бути обчислена за певною формулою. Для цього в квадратних дужках можна вказувати досить складні вирази, наприклад, [**ebx+ecx**] або [**ebx+4*ecx**].

Команди мови асемблера зазвичай мають 1 або 2 операнди, або не мають операндів взагалі. У багатьох випадках операнди (якщо їх два) повинні мати однаковий розмір. Команди мови асемблера зазвичай не працюють з двома комірками пам'яті.

У більшості випадків в системах команд процесорів виділяються групи:

– команди передачі даних, використовуються для переміщення даних між різними вузлами комп'ютера;

– арифметичні команди, призначені для виконання додавання, віднімання, множення і ділення чисел;

– команди перетворення даних, застосовуються для перетворення типів даних;

– логічні команди, виконують логічні додавання і множення, інверсію і т.д.;

– команди зсувів, виконують різні порозрядні зсуви вліво і вправо;

– команди управління виконанням. У цю групу включаються команди, що порушують природний порядок виконання команд програми, такі як програмні переривання і повернення з переривань, виклик і повернення з підпрограм, різні умовні та безумовні переходи.

Одна з основних команд мови асемблер - це команда ***пересилання***. З її допомогою можна записати в регістр значення іншого регістра, константу або значення комірки пам'яті, а також можна записати в комірку пам'яті значення регістра або константу. Команда має наступний синтаксис:

```
MOV     <операнд1>, <операнд2>
```

За командою **MOV** значення другого операнда записується в перший операнд. Операнди повинні мати однаковий розмір. Команда не змінює прапори.

```
mov     eax, ebx    ; Пересилаємо значення регістра EBX в
                    ; регістр EAX
mov     eax, 0ffffh ; Записуємо в регістр EAX 16-ое
                    ; значення ffff
mov     x, 0        ; Записуємо в зміну x значення 0
```

Переслати значення з однієї комірки пам'яті в іншу не можна, але можна використовувати дві команди **MOV**.

```
mov     eax, x
mov     y, eax
```

Насправді процесор має багато команд пересилання - код команди залежить від того, куди і звідки пересилаються дані. Але компілятор мови асемблера сам вибирає потрібний код в залежності від операндів, так що, з точки зору програміста, команда пересилання тільки одна.

Для ***перестановки*** двох величин використовується команда обміну **XCHG**:

```
XCHG    <операнд1>, <операнд2>
```

Кожен з операндів може бути регістром або коміркою пам'яті. Однак переставити вміст двох регістрів можна, а двох комірок пам'яті - ні. Операнди повинні мати однаковий розмір. Команда не змінює прапори.

3.2 Оператор зазначення типу

Операнди команди **MOV** повинні мати однаковий розмір. У деяких випадках компілятор може визначити розмір операнда. Наприклад, регістр **EAX** має розмір 32 біта, а регістр **DX** - 16 біт. Розмір змінної визначається за директивою, зазначеної в її оголошенні. Якщо можна визначити розмір тільки одного операнда, то розмір другого операнда підганяється під розмір першого, якщо це можливо. Якщо ж можна визначити розміри обох операндів, то вони повинні збігатися.

```
xdb    ?
. . .
mov    x, 0          ; 0 може мати будь-який розмір, в
                     ; даному випадку береться 1 байт
mov    eax, 0        ; 0 може мати будь-який розмір, в
                     ; даному випадку береться 4 байта
mov    al, 1000h     ; Помилка - спроба записати 2-байтне
                     ; число в 1-байтний регістр
mov    eax, cx       ; Помилка – розміри операндів не
                     ; співпадають
```

Однак не завжди буває можливо визначити розмір величини, яка пересилається за операндами команди **MOV**. Наприклад, якщо один з операндів є коміркою пам'яті, адреса якої записана в регістрі, то за цією адресою можна записати і 1 байт, і 2 байта, і 4 байта. Якщо другий операнд є регістром, то розмір даних, що пересилаються визначається за розміром регістра. Якщо ж другий операнд є константою, то розмір даних, що пересилаються визначити не можна, і компілятор фіксує помилку. Для того щоб уникнути цієї помилки, треба явно вказати розмір даних, що пересилаються. Для цього використовується оператор **PTR**:

<тип> PTR <вираз>

В якості типу використовується **BYTE**, **WORD** або **DWORD**.

```
mov    [ebx],0              ; Помилка, 0 може мати
                            ; будь який розмір
mov    byte ptr [ebx],0     ; Пересилаємо 1 байт
mov    dword ptr [ebx],0    ; Пересилаємо 4 байти
```

3.3 Команди роботи зі стеком

Розміщення операнда в стек:

push [источник]

Команда зменшує на 2 (4) значення регістра - покажчика стеку **SP**/**ESP**, а потім записує значення джерела в вершину стеку. В якості приймача може використовуватися будь-який 16 розрядний регістр, в тому числі і сегментний, або комірка пам'яті. Стан пам'яті при цьому не змінюється. Застосовується для приміщення параметрів в стек перед викликом процедури, також для тимчасового збереження даних в стек.

Приклади використання команди **PUSH**:

```
push AX
push CS
push MEM 1
```

Вилучення операнда зі стеку:

pop [приемник]

Команда відновлює вміст вершини стеку в регістр, комірку пам'яті або сегментний регістр, після чого вміст **SP/ESP** збільшується на 2 (4) байти. Застосовується для повернення зі стеку значень, включених в нього командою **PUSH**.

Приклади використання команди:

```
pop AX
pop CS
pop MEM 1
```

PUSHA та **PUSHAD** - застосовуються для приміщення вмісту всіх 8 регістрів загального призначення в стек (**PUSHA** - оперує 16-бітними регістрами, **PUSHAD** - 32-бітними регістрами). Ці команди використовуються перед викликом процедур.

POPA та **POPAD** - відновлюють зі стеку вміст 8 регістрів загального призначення, будучи доповненням команд **PUSHA** та **PUSHAD**.

3.4 Виведення і введення інформації

У мовах асемблера відсутні готові процедури введення виведення. Для виконання цих операцій існують такі варіанти:

– безпосередньо звертатися до пристроїв введення-виведення. Цей спосіб є єдиним в разі програмування, коли повністю відсутні готові процедури для роботи з зовнішніми пристроями;

– використання процедур BIOS, розміщених в ПЗП і відповідно постійно присутніх в комп'ютері (звернення до функцій BIOS). Цей спосіб використовується при програмуванні під час відсутності ОС;

– звернення до сервісів ОС із запитами на введення-виведення для відповідних пристроїв. Цей варіант є найбільш переважним і часто використовуваним, і розглянемо саме його.

Будь-яка ОС обов'язково забезпечує символьне введення-виведення.

Для ***введення*** символу досить скористатися функцією **01** DOS:

```
mov     ah,01           ; номер функції розміщуємо в ah
int     21h             ; передаємо керування DOS
```

Далі ОС поміщає код натиснутої клавіші в регістр **al** і повертає керування програмі.

Для ***виведення*** символу призначена функція **02**:

```
mov     ah,02           ; номер функції розміщуємо в ah
mov     dl,'*'          ; код символу, яке виводиться
                        ; розміщуємо в dl
int     21h             ; передаємо керування DOS
```

Далі ОС виводить в поточну позицію екрану символ, код якого знаходиться в **dl** (в даному випадку на екран буде виведена '*') і повертає керування програмі.

Для ***виведення рядка*** використовується функція **09**.

Розміщуємо в сегменті даних рядок, який виводиться, останній знак **$** - обмежувач рядка для DOS:

```
msg db 'Цей текст має з'явитися на екрані$'
```

В сегменті коду організовуємо виведення рядка:

```
mov     ah,09                   ; номер функції розміщуємо в ah
mov     dx,offset msg           ; запис в регістр dx адреси рядка
                                ; msg
int     21h                     ; передаємо керування DOS
```

Далі ОС виводить, починаючи з поточної позиції екрану, символи, коди яких адресуються **ds:dx** до тих пір, поки не зустріне код знака **$**, після чого управління повертається програмі.

Для введення рядка не довше 255 байтів використовується функція **0A**.Резервуємо область пам'яті під рядок в сегменті даних, який виводиться:

str db 255, ?, 255 dup (?) ; Перший байт - максимальна довжина рядка, другий - фактична довжина (заповнюється ОС в процесі введення), далі - власне рядок символів

В сегменті коду організовуємо введення рядка:

```
mov   ah,0a               ; номер функції розміщуємо в ah
mov   dx, offset str      ; запис в регістр dx адреси рядка
                          ; str
int   21h                 ; передаємо керування DOS
```

Для введення-виведення чисел ОС не надається ніяких можливостей, тому перетворення символів, що вводять в число і назад повністю покладається на програму.

Перетворення числа в рядок символів проводиться послідовним цілочисельним розподілом на основу системи числення до отримання нуля і збереженням залишків від ділення. Отримані залишки і є цифри, які складають число. Далі необхідно перетворити цифри в коди відповідних символів (додати 48 (30h) - код символу "0") і в зворотному порядку вивести їх на екран. Розглянемо це перетворення на прикладі числа 6543 і десяткової системи числення (табл. 3.1)

Таблиця 3.1

Ділення	Частка	Залишок	Код символу	Символ
6543:10	654	3	51	‘3’
654:10	65	4	52	‘4’
65:10	6	5	53	‘5’
6:10	0	6	54	‘6’

ТЕМА 4. Арифметичні операції

4.1 Команди додавання

Для виконання операції додавання є три команди: **ADD**, **ADC**, **INC**.

ADD (ADDition) - додавання двох операндів джерело і приймач розмірністю байт, слово або подвійне слово.

```
add   приймач,джерело     ; приймач ← приймач + джерело
```

Алгоритм роботи:

– скласти операнди джерело і приймач;
– записати результат складання в приймач;

– встановити прапори.

Команда **ADD** виконує додавання двох операндів і вміст операнда (приймача) заміщається результатом складання. Другий операнд не змінюється. В якості першого операнда команди **ADD** можуть використовуватися регістри (крім сегментних), комірки пам'яті, в якості другого операнда можуть використовуватися регістри (крім сегментних), комірки пам'яті або безпосереднє значення, проте не допускається визначати обидва операнди одночасно як комірки пам'яті. Операнди повинні мати однаковий розмір. Можливо додавання як знакових, так і беззнакових чисел будь-якого розміру. Команда впливає на прапори **OF**, **SF**, **ZF**, **AF**, **PF** і **CF**.

Приклади використання команди **ADD**:

```
add     ax,bx        ; додавання вмісту регістрів
add     bx,mem1      ; додавання вмісту регістру та комірки
                     ; пам’яті
add     ax,5         ; додавання вмісту регістру та числа
add     mem1,5       ; додавання вмісту комірки пам’яті та
                     ; числа
add     mem1,ax      ; додавання вмісту комірки пам’яті та
                     ; регістру
```

ADC (Addition with Carry) - додавання з переносом, додавання двох операндів з урахуванням переносу з молодшого розряду.

```
adc     приймач,джерело     ; приймач ← приймач + джерело
                            ; +cf
```

Алгоритм роботи:

– скласти операнди джерело, приймач і прапорець переносу **cf**;

– записати результат складання в приймач;

– встановити прапори.

Команда **ADC** здійснює додавання першого і другого операндів, додаючи до результату значення прапора переносу **cf**. Початкове значення першого операнда (приймача) втрачається, заміщаючи результатом додавання. Другий операнд не змінюється. В якості першого операнда команди **ADC** можна вказувати регістр (окрім сегментного) або комірку пам'яті, в якості другого - регістр (окрім сегментного), комірку пам'яті або безпосереднє значення, проте не

допускається визначати обидва операнди одночасно як комірки пам'яті. Операнди повинні мати однаковий розмір. Можливо складання як знакових, так і беззнакових чисел будь-якого розміру. Команда впливає на прапори **OF**, **SF**, **ZF**, **AF**, **PF** і **CF**.

Приклади використання команди **ADC**:

```
adc    ax,bx       ; додавання вмісту регістрів та cf
adc    bx,mem1     ; додавання вмісту регістру, комірки
                   ; пам'яті та cf
adc    ax,5        ; додавання вмісту регістру, числа та cf
adc    mem1,5      ; додавання вмісту комірки пам'яті,
                   ; числа та cf
adc    mem1,ax     ; додавання вмісту комірки пам'яті,
                   ; регістру та cf
```

Розглянемо приклад, скласти значення в регістрах **CX:BX** (1111aaaah), розміром в подвійне слово, зі значенням, записаним в регістрах **DX:AX** (2222bbbbh), також розміром в подвійне слово:

```
add    ax,bx       ; додавання молодших слів подвійних
                   ; слів (aaaah+bbbbh) сума в регістрі ax
                   ; (6665h) та cf =1
adc    dx,cx       ; додавання старших слів подвійних слів
                   ; та cf (1111h+2222h+1) сума в регістрі
                   ; dx (3334h) та cf =0
```

INC (INCrement operand by 1) - збільшити операнд на 1.

```
inc    приймач     ; приймач ← приймач +1
```

Команда **inc** додає 1 до операнда, в якості якого можна указувати регістр (окрім сегментного) або комірку пам'яті. Не допускається використовувати в якості операнда безпосереднє значення. Операнд інтерпретується як число без знака. Команда змінює прапори **OF, SF, ZF, AF** и **PF.**

Приклади використання команди **INC**:

```
inc ax        ; збільшення вмісту регістру ax на 1
inc al        ; збільшення вмісту молодшого байта регістру
              ; ax на 1
inc mem1      ; збільшення вмісту комірки пам'яті mem1 на 1
```

4.2 Команди віднімання

В операціях відніманні необхідно враховувати співвідношення значень зменшуваного і від'ємника. Якщо зменшуване більше від'ємника, то результат позитивний і правильний. Якщо зменшуване менше від'ємника, то процесор робить позику з розряду, наступного за старшим, в розрядної сітці операнда. Факт позики фіксується установкою прапора **cf** в одиницю. В цьому випадку результат слід інтерпретувати в додатковому коді. Для виконання операції віднімання є команди: **SUB**, **SBB**, **DEC**.

SUB (SUBtract) – цілочислене віднімання.

sub приймач,джерело ; приймач ← приймач - джерело

Алгоритм роботи:

- виконати віднімання від джерела приймача;
- записати результат в приймач;
- встановити прапори.

Команда **sub** віднімає другий операнд (джерело) з першого (приймача) і поміщає результат на місце першого операнда. Початкове значення першого операнда (зменшуване) втрачається. В якості першого операнда можна вказувати регістр (окрім сегментного) або комірку пам'яті, в якості другого - регістр (окрім сегментного), комірку пам'яті або безпосереднє значення, проте не допускається визначати обидва операнди одночасно як комірки пам'яті. Операнди повинні мати однаковий розмір. Можливо віднімання як знакових, так і беззнакових чисел будь-якого розміру. Команда впливає на прапори **OF, SF, ZF, AF, PF** и **CF.**

Приклади використання команди **SUB**:

```
sub ax,вx      ; віднімання вмісту регістрів
sub bx,mem1    ; віднімання вмісту регістру та комірки пам’яті
sub ax,5       ; віднімання вмісту регістру та числа
sub mem1,5     ; віднімання вмісту комірки пам’яті та числа
sub mem1,ax    ; віднімання вмісту комірки пам’яті та регістру
```

SBB (SuBtract with Borrow) - віднімання з позикою. Цілочисельне віднімання з урахуванням результату попереднього вирахування командами **sbb** і **sub** (станом прапора переносу **cf**).

sbb приймач,джерело ; приймач ← приймач – джерело - **cf**

Алгоритм роботи:

– виконати віднімання від джерела приймача та **cf**;
– записати результат в приймач;
– встановити прапори.

Команда **SBB** віднімає другий операнд (джерело) з першого (приймача). Результат заміщає перший операнд, попереднє значення якого втрачається. Якщо встановлений прапор **cf**, з результату віднімається ще 1. В якості першого операнда можна вказувати регістр (окрім сегментного) або коміру пам'яті, в якості другого - регістр (окрім сегментного), комірку пам'яті або безпосереднє значення, проте не допускається визначати обидва операнди одночасно як комірки пам'яті. Операнди повинні мати однаковий розмір. Можливо віднімання як знакових, так і беззнакових чисел будь-якого розміру. Команда впливає на прапори **OF, SF, ZF, AF, PF** и **CF.**

Приклади використання команди **SBB:**

```
sbb dh,dl      ; віднімання вмісту регістрів та cf
sbb bx,mem1    ; віднімання вмісту регістру, комірки пам’яті та
               ; cf
sbb ax,5       ; віднімання вмісту регістру, числа та cf
sbb mem1,5     ; віднімання вмісту комірки пам’яті, числа і cf
sbb mem1,ax    ; віднімання вмісту комірки пам’яті, регістру та
               ; cf
```

Команда **SBB** зазвичай використовується для вирахування 32-розрядних чисел, а саме, для виконання віднімання старших частин значень багатобайтових операндів з урахуванням можливого попереднього позика при відніманні молодших частин значень цих операндів. Розглянемо приклад, відняти значення в регістрах **CX:BX** (aaaa1111h), розміром в подвійне слово, зі значенням, записаним в регістрах **DX:AX** (2222bbbbh), також розміром в подвійне слово:

```
sub     ax,bx  ; віднімання молодших слів подвійних слів
               ; (1111h-bbbbh) результат у регістрі ax (5556h)
               ; та cf =1
sbb     dx,cx  ; віднімання старших слів подвійних слів та cf
               ; (aaaah-2222h-1) результат у регістрі dx (8887h)
               ; та cf =0
```

DEC (DECrement operand by 1) - зменшення операнда на 1

dec приймач ; приймач ← приймач - 1

Команда **DEC** віднімає 1 з операнда, в якості якого можна указувати регістр (окрім сегментного) або комірку пам'яті. Не допускається використовувати в якості операнда безпосереднє значення. Операнд інтерпретується як число без знака. Команда впливає на прапори **OF, SF, ZF, AF** и **PF.**

Приклади використання команди **DEC**:

```
dec ax      ; зменшення вмісту регістру на 1
dec al      ; зменшення вмісту молодшого байту регістру
            ; на 1
dec mem1    ; зменшення вмісту комірки пам’яті на 1
```

NEG (NEGate operand) - зміна знаку (отримання двійкового доповнення) джерела.

neg приемник ; звернення знака приймача

Команда **NEG** виконує віднімання цілочисельного операнда із знаком з нуля, перетворюючи позитивне число в негативне і навпаки. Вихідний операнд затирається. В якості операнда можна вказувати регістр (окрім сегментного) або комірку пам'яті розміром в байт, або в слово. Не допускається використовувати в якості операнда безпосереднє значення. Команда впливає на прапори **OF, SF, ZF, AF, PF** и **CF**. Якщо значення операнда дорівнює мінімальному від'ємному значенню байта або слова, то команда не змінює операнд.

Приклади використання команди **NEG**:

```
neg ax      ; звернення знака вмісту регістру
neg al      ; звернення знака вмісту молодшого байту
            ; регістру
neg mem1    ; звернення знака вмісту комірки пам’яті
```

4.3 Команди множення

Додавання і віднімання знакових і беззнакових чисел виконуються по одним і тим же алгоритмам. Тому немає окремих команд додавання і віднімання для знакових і беззнакових чисел. А ось множення і ділення знакових і беззнакових чисел виконуються за різними алгоритмами, тому існують по дві команди множення і ділення.

MUL (MULtiply) - множення двох цілих чисел без знака.

mul операнд ; множення цілих чисел без знака

В якості операнда-співмножника команди **mul** можна указувати регістр (окрім сегментного) або комірку пам'яті, не допускається множення на безпосереднє значення. Команда впливає на прапори **OF** і **CF**.

У командах множення один із співмножників повинен знаходитися в регістрі **AL** при множенні байтів, або в регістрі **AX** при множенні слів. Другий співмножник може розміщуватися в регістрах або комірках пам'яті. При множенні байтів результат поміщається в регістр **AX**, а при множенні слів в регістри **DX** і **AX** (старші 16 біт результату зберігаються в **DX**, а молодші 16 біт записуються в регістр **AX**). Розмір результату в два рази більше розміру співмножників. Якщо старша половина результату містить тільки нулі, прапори **CF** і **OF** встановлюються в 0, інакше в 1.

Приклади використання команди **MUL**:

```
mul вх          ; множення вмісту регістра ax на вміст регістра
                ; вх, результат в dx та ax
mul вl          ; множення вмісту регістра al на вміст регістра
                ; вl, результат в ax
mul mem1        ; множення вмісту регістра ax(al) на вміст
                ; комірки пам’яті розміром 2(1) байти, результат
                ; в dx та ax або ax
mul ax          ; зведення в квадрат вмісту регістра ax,
                ; результат в dx та ax
```

IMUL (Integer MULtiply) - множення двох цілочисельних значень зі знаком.

```
imul операнд
imul операнд, безпосередній операнд
imul операнд1, операнд2, безпосередній операнд
imul операнд1, операнд2
```

Команда знакового множення має кілька варіантів

Перший відповідає команді **MUL** - один із співмножників вказується в команді, другий повинен знаходитися в регістрі **EAX**/**AX**/**AL**, а результат поміщається в регістри **EDX**:**EAX**/**DX**:**AX**/**AX**.

Другий варіант команди **IMUL** дозволяє вказати регістр, який буде містити один із співмножників. В цей же регістр буде поміщений результат. Другий співмножник вказується безпосередньо в команді.

Третій варіант команди **IMUL** дозволяє вказати і результат, і обидва співмножники. Однак результат може бути поміщений тільки в регістр, а другий співмножник може бути тільки безпосереднім операндом. Перший співмножник може бути регістром або коміркою пам'яті.

Четвертий варіант команди **IMUL** дозволяє вказати обидва співмножники. Перший повинен бути регістром, а другий - регістром або коміркою пам'яті. Результат поміщається в регістр, який є першим операндом.

Команда **IMUL** встановлює прапори так само, як і команда **MUL**. Однак розширення результату в регістр **EDX**/**DX** відбувається тільки при використанні першого варіанту команди **IMUL**. В інших випадках частина добутку, не поміщається в регістр-результат, втрачається, навіть якщо в якості результату вказано регістр **EAX**/**AX**. При множенні двох 1-байтових чисел, добуток яких більше байта, але менше слова, в регістрі-результаті виходить коректне добуток.

Приклади використання команди **IMUL**:

```
mov     eax,5
mov     ebx,-7
imul    ebx                 ; EAX=ffffffdd,
                            ; EDX=ffffffff, CF=0
mov     ebx,3
imul    ebx,6               ; EBX=EBX*6
mov     ebx,500000
imul    eax,ebx,100000      ; EAX=EBX*100000, старша
                            ; частина результату втрачається
xdd     40
mov     eax,55
imul    eax,x               ; EAX=EAX*x
```

4.4 Команди ділення

Ділення, як і множення, реалізується двома командами, призначеними для знакових і беззнакових чисел.

DIV (DIVide unsigned) - ділення двох беззнакових значень.

IDIV (Integer DIVide) - ділення двох знакових значень.

```
div источник                ; ділення чисел без знака
idiv источник               ; ділення чисел зі знаком
```

У командах вказується тільки один операнд - дільник, який може бути регістром або коміркою пам'яті, але не може бути безпосереднім операндом. Місцезнаходження діленого і результату для команд ділення фіксоване.

Якщо дільник має розмір 1 байт, то ділене береться з регістра **AX**. Якщо дільник має розмір 2 байти, то ділене береться з реєстрової пари **DX**:**AX**.

Оскільки процесор працює з цілими числами, то в результаті поділу виходить відразу два числа - частка і залишок. Ці два числа також поміщаються в певні регістри. Якщо дільник має розмір 1 байт, то частка поміщається в регістр **AL**, а залишок - в регістр **AH**. Якщо дільник має розмір 2 байти, то частка поміщається в регістр **AX**, а залишок - в регістр **DX**.

Приклади використання команди **DIV**, **IDIV**:

```
mov     ax, 127
mov     bl, 5
div     bl                  ;AL=19h=25, AH=02h=2
mov     ax, 127
mov     bl, -5
idiv    bl                  ;AL=e7h=-25, AH=02h=2
mov     ax, -127
mov     bl, 5
idiv    bl                  ;AL=e7h=-25, AH=feh=-2
mov     ax, -127
mov     bl, -5
idiv    bl                  ;AL=19h=25, AH=feh=-2
```

4.5 Переповнення

При використанні арифметичних операцій може статися переповнення і результат вийти неправильний

Переповнення при додаванні/відніманні. Треба враховувати, що команди складання (**ADD/ADC**) і віднімання (**SUB/SBB**) не роблять різниці між знаковими і беззнаковими числами, вони просто складають і віднімають біти. Нагадаємо, що в знакових числах лівий (старший) біт є знаковим і діапазон для байта від -128 до 127, для беззнакових чисел все біти є даними і діапазон для байта від 0 до 255.

Якщо комірка пам'яті містить значення 11111001b/F9h, то для беззнакового числа це позитивне число 249, а для знакового -7, тому

що старший знаковий біт = 1 і число в додатковому коді. При перекладі числа з додаткового коду 11111001b в прямий код отримаємо 1000111b (-7).

Розглянемо в нижче наведеному лістингу програми кілька прикладів складання чисел, а саме, 249 і 2, 252 і 5, 121 і 11, 246 і 137.

```
title переповнення
MODEL  small
STACK  256
DATASEG
Adb    249           ; F9h
Xdb    252           ; FCh
Ydb    121           ; 79h
Zdb    246           ; F6h
CODESEG
Start:
mov    ax,@data
mov    ds,ax
mov    bl,A
add    bl,2          ; F9h+2h=FBh, sf=1, cf=0, of=0,
                     ; для без знакового 249+2=251,
                     ; для знакового -7+2= -5
mov    bl,X
add    bl,5          ; FCh+5h=01h, sf=0, cf=1, of=0,
                     ; для без знакового 252+5=1 - невірно,
                     ; для знакового -4+5=1
mov    bl,Y
add    bl,11         ; 79h+Bh=84h, sf=1, cf=0, of=1,
                     ; для беззнакового 121+11=132,
                     ; для знакового 121+11=-124 - невірно
mov    bl,Z
add    bl,137        ; F6h+89h=7Fh, sf=0, cf=1, of=1,
                     ; для беззнакового 246+137=127, невірно
                     ; для знакового 246+137=127, невірно
Exit:
mov    ax,04c00h
int    21h
end    Start
```

Таким чином, визначити коректний результат додавання/віднімання чи ні, можна за станом прапорів переносу **cf**, і переповнення **of** (табл. 4.1).

Таблиця 4.1 – Переповнення при додаванні/відніманні

cf	**of**	**Результат**	
		Беззнакові	**Знакові**
0	0	вірний	вірний
0	1	вірний	невірний
1	0	невірний	вірний
1	1	невірний	невірний

Також можливо ***переповнення*** при використанні ***команди ділення*** (**DIV/IDIV**). У ділення великих чисел передбачається, що частка значно менше, ніж ділене. При розподілі на 1 генерується результат, який дорівнює діленому, що може викликати переповнення.

Рекомендується використовувати наступне правило: ***якщо дільник - байт, то його значення повинний бути менше, ніж старший байт (AH) діленого; якщо дільник - слово, то його значення повинний бути менше, ніж старше слово (DX) діленого.***

Для перевірки даного правила рекомендується вставляти перед командою **DIV** команду порівняння **CMP**, яку будемо розглядати далі.

Для команди **IDIV** треба враховувати, що ділене або дільник можуть бути негативними, а тому порівнюються абсолютні значення, то необхідно використовувати команду **NEG** для тимчасового переведення від'ємного значення в позитивне.

4.6 Команди перетворення даних

При діленні великих чисел розмір діленого в два рази більше, ніж розмір дільника. Тому не можна просто завантажити дані в регістр **AX** і поділити його на будь-яке значення, тому що в операції ділення буде задіяний також і регістр **DX**. Тому перш ніж виконувати поділ, треба встановити коректне значення в регістр **DX**, інакше результат буде невірним. Значення регістра **DX** має залежати від значення регістра **AX**. Тут можливі два варіанти - для знакових і беззнакових чисел.

Якщо використовуються беззнакові числа, то в будь-якому випадку в регістр **DX** необхідно записати значення 0: **aah/10101010b** → **00aah/0000000010101010b**.

Якщо ж використовуються знакові числа, то значення регістра **DX** буде залежати від знака числа: **55h/01010101b** → **0055h /0000000001010101b**, **aah/10101010b** → **ffaah /1111111110101010b**.

Записати значення 0 не складно, а ось для знакового розширення необхідно аналізувати знак числа. Однак немає необхідності робити це вручну, тому що мова асемблера має ряд команд, що дозволяють розширювати байт до слова, слово до подвійного слова і подвійне слово до чотирьох слів.

```
cbw                 ; Знакове розширення AL до AX
cwd                 ; Знакове розширення AX до DX:AX
cwde                ; Знакове розширення AX до EAX
cdq                 ; Знакове розширення EAX до EDX:EAX
```

Таким чином, якщо дільник має розмір 2 або 4 байти, то потрібно встановлювати значення не тільки регістра **AX**/**EAX**, а й регістра **DX**/**EDX**. Якщо ж дільник має розмір 1 байт, то можна просто записати ділене в регістр **AX**.

```
x dd    ?
. . .
mov     eax,x  ; Заносимо в регістр EAX значення змінної x
cdq            ; Знакове розширення EAX до EDX:EAX
mov     ebx,7
idiv    ebx
```

У мові асемблера існують також команди, що дозволяють занести в регістр значення іншого регістра або комірки пам'яті із знаковим або беззнаковим розширенням.

Знакове розширення - старші біти заповнюються знаковим бітом:

```
MOVSX  <операнд1>,<операнд2>
```

Беззнакове розширення – старші біти заповнюються нулем:

```
MOVZX  <операнд1>,<операнд2>
```

Операнд1 і операнд2 можуть мати будь-який розмір. Операнд1 повинен бути більше, ніж операнд2. У разі рівного розподілу розміру

операндів слід використовувати звичайну команду пересилання **MOV**, яка виконується швидше.

Розглянемо приклад: необхідно обчислити **x*x*x**, де **x** - 1-байтовая змінна.

Перший варіант

```
mov    al,x        ; пересилаємо x до регістру AL
imul   al          ; множимо регістр AL на себе, AX = x*x
movsx  bx,x        ; пересилаємо x в BX зі знаковим
                   ; розширенням
imul   bx          ; множимо AX на BX, результат до DX:AX
```

Другий варіант

```
mov    al,x        ; пересилаємо x до регістру AL
imul   al          ; множимо регістр AL на себе, AX = x*x
cwde               ; розширяємо  AX до EAX
movsx  ebx,x       ; пересилаємо x до EBX зі знаковим
                   ; розширенням
imul   ebx         ; множимо EAX на EBX, результат до
                   ; EDX:EAX
```

ТЕМА 5. Порозрядні операції

5.1 Логічні команди

Команди цієї групи застосовуються для виробництва булевих операцій і забезпечують роботу з двійковими полями в байтах, словах і подвійних словах.

NOT - інвертує біти свого операнда.

NOT <операнд>

Операнд може бути регістром або коміркою пам'яті. Операція не змінює прапори.

AND - виконує логічну функцію «І» всіх пар біт операндів і зручна для установки довічного поля в нуль.

AND <операнд1>,<операнд2>

OR - виконує логічну функцію «АБО» всіх пар біт операндів. Зазвичай застосовується для установки довічного поля в потрібний стан (поле попередньо очищається командою **AND**).

OR <операнд1>,<операнд2>

XOR - виконує логічну функцію «ВИКЛЮЧНЕ АБО» для двох операндів, виконує додавання по модулю 2 всіх пар біт операндів. Застосовується для інвертування в двійковому полі тільки певних біт.

XOR <операнд1>,<операнд2>

Операції **AND**, **OR** і **XOR** мають по два операнда. Перший може бути регістром або коміркою пам'яті, а другий - регістром, коміркою пам'яті або безпосереднім операндом. Операнди повинні мати однаковий розмір. Результат поміщається на місце першого операнда. Операції змінюють прапори **CF**, **OF**, **PF**, **SF** и **ZF**.

Операція **XOR** має цікаву особливість - якщо значення операндів збігаються, то результатом буде значення 0. Тому операцію **XOR** використовують для обнуління регістрів - вона виконується швидше, ніж запис нуля за допомогою команди **MOV**.

```
xor     eax,eax         ; при будь-якому значенні EAX
                        ; результат буде дорівнює 0
```

TEST - команда логічного порівняння аналогічна команді **AND**, але результат не зберігається. Застосовується для перевірки довічного поля на нуль (або не нуль).

5.2 Команди операцій над бітами

Команди цієї групи дуже зручні для маніпуляцій окремими бітами, допускаючи їх установку і перевірку.

Команда **BT**/**BTS**/**BTR**/**BTC** бере біт з заданим номером зі слова (або з подвійного слова) і заносить його у прапорець переносу **CF**:

```
BT      <операнд1>,<операнд2>
BTS     <операнд1>,<операнд2>
BTR     <операнд1>,<операнд2>
BTC     <операнд1>,<операнд2>
```

Команда перевірки біта **BT** просто поміщає значення зазначеного біта у прапорець переносу **CF**. Три інші команди виконують цю ж функцію, але дозволяють адресується біт встановити в 1 (**BTS**), скинути в 0 (**BTS**) або інвертувати (**BTC**).

Першим операндом задається початкове слово (або подвійне слово), звідки береться біт, що перевіряється. Цей операнд може бути регістр або коміркою пам'яті.

У другому операнді задається номер розряду для біта, що перевіряється. Цей операнд може бути або регістр, або просто числове

значення. (Ці два різні способи завдання другого операнда підтримуються абсолютно різними машинними командами).

Команда впливає тільки на прапорець **CF**, інші прапорці не змінюються.

Команда **BSF**/**BSR** сканують біти в слові (або в подвійному слові) і відшукує перший зустрінутий біт, що дорівнює 1. При цьому команда **BSF** сканує в прямому напрямку, починаючи з молодшого (нульового) розряду, а команда **BSR** сканує в зворотному напрямку, починаючи зі старшого розряду.

Слово (або подвійне слово), що перевіряється, задається другим операндом, може бути регістр або коміркою пам'яті.

Результатом є номер знайденого біта. Перший операнд задає, куди заносити результат, це операнд - приймач, може бути тільки регістр.

Якщо біт з одиницею знайдений, встановлюється прапорець (**ZF**=1), якщо не знайдений, то (**ZF**=0). На інші прапорці ця команда не впливає, інші прапорці не змінюються.

5.3 Команди зсуву

Операції зсуву вправо і зсуву вліво зрушують біти в змінної на задану кількість позицій. Кожна команда зсуву має два різновиди:

<мнемокод> <операнд>,< безпосередній операнд>
<мнемокод> <операнд>,CL

Перший операнд повинен бути регістром або коміркою пам'яті. Саме в ньому здійснюється зсув. Другий операнд визначає кількість позицій для зсуву, яке задається безпосереднім операндом або зберігається в регістрі **CL** (і тільки **CL**).

Команди зсуву змінюють прапори **CF**, **OF**, **PF**, **SF** и **ZF**.

Існує кілька різновидів зсувів, які відрізняються тим, як заповнюються «звільнені» біти.

Логічні зсуви. При логічному зсуві «звільнені» біти заповнюються нулями. Останній «звільнений» біт зберігається у прапорі **CF**.

SHL <операнд>,<кількість> ; Логічний зсув вліво (рис. 5.1)
SHR <операнд>,<кількість> ; Логічний зсув вправо (рис. 5.2)

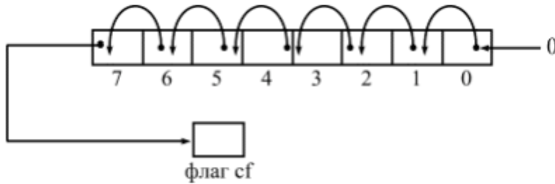


Рисунок 5.1 – Логічний зсув вліво

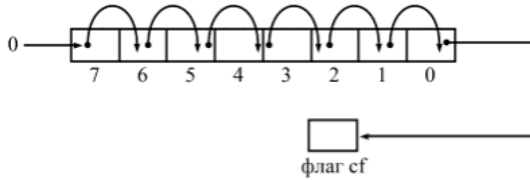


Рисунок 5.2 – Логічний зсув вправо

Арифметичні зсув. Арифметичний зсув вліво еквівалентний логічному зсуву вліво (це одна і та ж команда) - «звільнені» біти заповнюються нулями. При арифметичному зсуві вправо «звільнені» біти заповнюються знаковим бітом. Останній «звільнений» біт зберігається у прапорі **CF**.

SAL <операнд>,<кількість> ; Арифметичний зсув вліво (рис. 5.3)
SAR <операнд>,<кількість> ; Арифметичний зсув вправо (рис. 5.4)

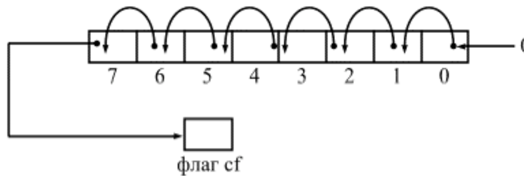


Рисунок 5.3 – Арифметичний зсув вліво

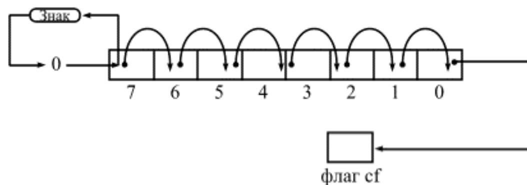


Рисунок 5.4 – Арифметичний зсув вправо

Циклічні зсуву. При циклічному зсуві «звільнені» біти заповнюються «звільненими» бітами. Останній «звільнений» біт зберігається у прапорі **CF**.

ROL <операнд>,<кількість> ; Циклічний зсув вліво (рис. 5.5)

ROL <операнд>,<кількість> ; Циклічний зсув вправо (рис. 5.6)

RCL <операнд>,<кількість> ; Циклічний зсув вліво (рис. 5.7)

RCR <операнд>,<кількість> ; Циклічний зсув вправо (рис. 5.8)

У командах циклічного зсуву **RCL**/**RCR** біт операнда, що висувається, розміщується у прапорець переносу, а старе значення прапорця переносу передається в звільнений біт.

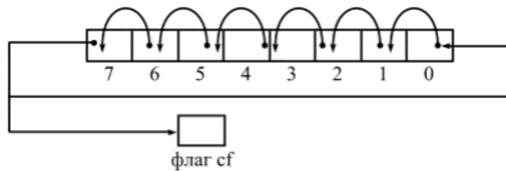


Рисунок 5.5 – Циклічний зсув вліво **ROL**

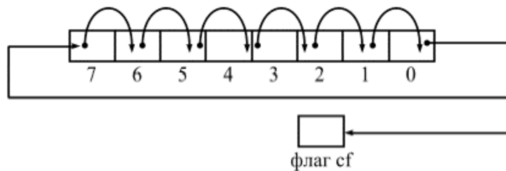


Рисунок 5.6 – Циклічний зсув вправо ROR

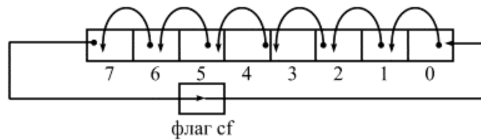


Рисунок 5.7 – Циклічний зсув вліво RCL

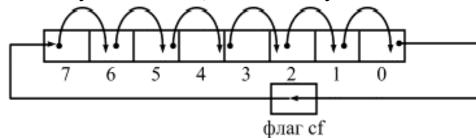


Рисунок 5.8 – Циклічний зсув вправо RCR

Розширені зсуви. Розширені зсуви трохи відрізняються від інших зсувів. В розширених зсувах беруть участь два регістра або комірка пам'яті і регістр, які як би об'єднуються в єдине ціле і «звільнені» біти одного операнда заповнюються бітами з іншого операнда.

SHLD <операнд1>,<операнд2>,<кількість>

; Розширений зсув вліво

SHRD <операнд1>,<операнд2>,<кількість>

; Розширений зсув вправо

Команда **SHLD** виконує зсув вліво біти операнда1 на вказану кількість позицій. Молодші («звільнені») біти операнда1 заповнюються старшими бітами операнда2. Сам операнд2 не змінюється.

Команда **SHRD** виконує зсув вправо біти операнда1 на вказану кількість позицій. Старші («звільнені») біти операнда1 заповнюються молодшими бітами операнда2. Сам операнд2 не змінюється.

Кількість, як і в інших операціях зсуву, задається безпосереднім операндом або зберігається в регістрі **CL**. Але використовуються тільки останні 5 біт операнда, що визначає кількість, тобто максимальну кількість позицій зсуву дорівнює 32.

ТЕМА 6. Команди передачі управління. Цикли

Для зміни природного порядку виконання команд в мові асемблера, як і в інших мовах програмування є команди безумовного і умовного переходу.

Команди безумовного і умовного переходу змінюють вміст регістра покажчика команди **ip**, а іноді і сегментний регістр коду **cs**, в яких зберігається адреса тієї команди, яка повинна виконуватися наступною. Команди переходу не змінюють значення прапорів.

Команди безумовного переходу виконуються завжди. Команди умовного переходу виконуються тільки для деяких комбінацій прапорів.

Місце, куди здійснюється перехід, позначається за допомогою мітки.

Мітка - це символічне ім'я, що означає певну комірку пам'яті і використовується в якості операнда в командах передачі управління.

Асемблер пов'язує з міткою три атрибути:

- ім'я сегмента коду, де мітка описана;
- зсув - відстань в байтах від початку сегмента коду;
- тип мітки, або атрибут відстані.

Атрибут відстані може приймати значення **near** і **far**. Атрибут **near** означає перехід на мітку в межах сегмента. В цьому випадку змінюється лише регістр **ip**. Атрибут **far** визначає перехід між сегментами. При цьому атрибуті змінюється вміст регістрів **ip** і **cs**.

Мітку можна визначити двома способами: оператором двокрапка (так можна визначити тільки мітку ближнього типу) і директивою **label**.

Наприклад:

```
metka1:mov   ax,bx
metka2 label near
add    dx,var
```

6.1 Команди безумовної передачі керування

Всі команди безумовного переходу мають синтаксис:

```
jmp    [модифікатор]     адреса_переходу
```

адреса_переходу являє собою адресу у вигляді мітки (прямий перехід), або адресу області пам'яті, в якій знаходиться покажчик переходу (непрямий перехід).

За допомогою модифікатора можна задати ближній (внутрішньо сегментний) або дальній (міжсегментний перехід).

Близький перехід може бути прямим і непрямим:

Перехід називається прямим, якщо в команді переходу вказується мітка команди, на яку треба перейти. Однак в команді прямого переходу задається не абсолютна адреса, а різниця між адресою переходу і адресою команди переходу. Дія команди переходу полягає в додаванні цієї величини до поточного значення регістра **ip**. Операнд команди переходу розглядається як поле зі знаком, тому при додаванні його до значення регістра **ip** значення в цьому регістрі може як збільшитися, так і зменшитися, тобто можливий перехід і вперед, і назад.

Прямий перехід може бути ***прямим коротким і прямим***.

Перехід називається ***прямим коротким***, коли відстань від команди **jmp** до адреси переходу не більше -128 або +127 байтів. В

цьому випадку операнд займає всього один байт, а вся команда - два байта. Це зроблено з метою економії довжини коду.

Прямий перехід - це перехід, при якому "відстань" переходу більше 128 байт і менше 64К. В цьому випадку операнд займає два байти, а вся команда переходу - три байта.

Перехід назад може бути тільки коротким, але асемблер не може визначити, яким є перехід вперед. Тому для уточнення типу переходу вперед можна використовувати модифікатор **SHORT**. Приклади прямих переходів:

```
. . .
jmp     short met1          ; не більш 127 байтів
. . .
met1:   mov   ax,var
. . .
met2:   add   ax,var1       ; не більш 128 байтів
. . .
jmp     met2
. . .
jmp     met3                ; не більш 64 Кбайтів
. . .
met3:   add   ax,var1
```

Непрямий перехід - це перехід, при якому в якості операнда вказується місце (реєстр, слово в пам'яті), де знаходиться адреса переходу. Непрямий перехід використовується, коли адреса переходу стає відома тільки під час роботи програми. Приклад фрагмента з непрямим переходом:

```
adr     dw    metka
. . .
jmp     adr
. . .
mov     dx,adr
jmp     dx
. . .
metka   add   ax,var
```

6.2 Команди передачі управління за умовою

Команди умовного переходу забезпечують перехід, в залежності від стану регістра прапорів. Стану регістра прапорів можуть змінювати:

– будь-які команди, що змінюють значення арифметичних прапорів;

– команда порівняння **cmp**;

– вміст регістра **cx**.

Команда порівняння має синтаксис:

cmp операнд_1,операнд_2

Команда порівняння аналогічна команді вирахування **sub**, але результат віднімання нікуди не записується, а прапори встановлюються відповідно до результату.

Що стосується команд умовного переходу, то їх досить багато, але всі вони записуються одноманітно:

Jxx <метка>

Усі команди умовного переходу можна розділити на три групи.

До першої групи входять команди, які зазвичай ставляться після команди порівняння (табл. 6.1). В їх мнемокодах вказується той результат порівняння, при якому треба робити перехід.

Таблиця 6.1 - Умовні переходи після команди **cmp**

Мнемоніка	**Назва**	**Значення прапорців**	**Опис**
Для будь-яких чисел			
JE (equal)	Перехід якщо дорівнює	**ZF==1**	op1==op2
JNE (not equal)	Перехід якщо дорівнює	**ZF==0**	op1!=op2
Для чисел зі знаком			
JL/JNGE (less)	Перехід якщо менше	**SF!=OF**	op1<op2
JLE/JNG (less or equal)	Перехід якщо менше або дорівнює	**SF!=OF** || **ZF=1**	op1<=op2
JG/JNLE (greater)	Перехід якщо більше	**SF==OF&&ZF==0**	op1>op2
JGE/JNL (greater or equal)	Перехід якщо більше або дорівнює	**SF==OF**	op1=>op2
Для чисел без знаку			
JA/JNBE (above)	Перехід якщо нижче	**CF==0&&ZF==0**	op1>op2
JAE/JNB (above or equal)	Перехід якщо нижче або дорівнює	**CF==0**	op1=>op2
JB/JNAE (below)	Перехід якщо вище	**CF==1**	op1<op2
JBE/JNA (below or equal)	Перехід якщо вище або дорівнює	**CF==1** || **ZF==1**	op1<=op2

Розглянемо приклад: дано дві змінні **x** і **y**, в змінну **z** потрібно записати максимальне з чисел **x** и **y**.

```
      mov     eax, x
      cmp     eax, y
      jge/jae L               ; Використовуємо JGE для знакових
                              ; чисел і JAE - для беззнакових
      mov     eax, y
L:    mov     z, eax
```

До другої групи команд умовного переходу входять ті, які зазвичай ставляться після команд, відмінних від команди порівняння, і які реагують на те чи інше значення будь-якого прапора (табл. 6.2).

Таблиця 6.2 - Команди умовного переходу і прапори

Мнемоніка	Умова переходу
JC	CF==1
JP	PF==1
JZ	ZF==1
JS	SF==1
JO	OF==1
JNC	CF==0
JNP	PF==0
JNZ	ZF==0
JNS	SF==0
JNO	OF==0

Розглянемо приклад: нехай **a, b** і **c** - беззнакові змінні розміром 1 байт, потрібно обчислити **c=a*a+b**, але якщо результат перевершує розмір байта, передати управління на мітку **ERROR**.

```
      mov     al, a
      mul     al
      jc      ERROR
      add     al, b
      jc      ERROR
      mov     c, al
```

До третьої групи входять дві команди умовного переходу, перевіряючі не прапори, а значення регістра **ECX** або **CX**:

```
      JCXZ    <метка>         ; Перехід, якщо значення регістра CX
                              ; дорівнює 0
      JECXZ   <метка>         ; Перехід, якщо значення регістра ECX
```

; дорівнює 0

Однак ця команда виконується досить довго. Вигідніше провести порівняння з нулем і використовувати звичайну команду умовного переходу.

За допомогою команд переходу можна реалізувати будь-які розгалуження і цикли.

Приклад 1. Виконання **S**, якщо **x** > 0:

```
        cmp     x,0         ; порівняння x с 0
        jle     L           ; якщо x <= 0, то перехід на L, якщо ні
        ...                 ; то виконується S
L:      ...
```

Приклад 2. Якщо **x** не 0, то **S1**, якщо **x** = 0, то **S2**:

```
        cmp     x,0         ; порівняння x с 0
        je      L1          ; якщо x = 0, то перехід на L1, якщо ні
        ...                 ; то виконується S1
        jmp     L2          ; перехід на L2
L1:     ...                 ; виконується S2
L2:
```

Приклад 3. Якщо **a** > 0 та **b** > 0, то **S**:

```
        cmp     a,0         ; порівняння a с 0
        jle     L           ; якщо a <= 0, то перехід на L, якщо ні
        cmp     b,0         ; то порівняння b с 0
        jle     L           ; якщо b <= 0, то перехід на L, якщо ні
        ...                 ; то виконується S
L:
```

Приклад 4. Якщо **a** > 0 або **b** > 0, то **S**:

```
        cmp     a, 0        ; порівняння a с 0
        jg      L1          ; якщо a > 0, то перехід на L1, якщо ні
        cmp     b, 0        ; то порівняння b с 0
        jle     L2          ; якщо b <= 0, то перехід на L2, якщо ні
L1:     ...                 ; то виконується S
L2:
```

Приклад 5. Якщо **a** > 0 або **b** > 0 та **c** > 0, то **S**:

```
        cmp     a,0         ; порівняння a с 0
        jg      L1          ; якщо a > 0, то перехід на L1, якщо ні
        cmp     b,0         ; то порівняння b с 0
```

```
       jle     L2              ; якщо b <= 0, то перехід на L2, якщо ні
       cmp     c,0             ; то порівняння c с 0
       jle     L2              ; якщо c <= 0, то перехід на L2, якщо ні
L1:  ...                       ; то виконується S
L2:
```

Приклад 6. Якщо **x** > 0, то **S**:

```
L1:    cmp x,0                 ; порівняння x с 0
       jle L2                  ; якщо x <= 0, то перехід на L2, якщо ні
       ...                     ; то виконується S
       jmp L1                  ; перехід L1
L2:
```

Приклад 7. **S** поки **x** > 0:

```
L:     ...                     ; виконується S
       cmp x,0                 ; порівняння x с 0
       jg L                    ; якщо x > 0, то перехід на L
```

6.3 Команди організації циклів

Організувати циклічне виконання деяких ділянок програми можна, застосовуючи команди умовного та безумовного переходу. Наприклад, програма, що обчислює суму елементів масиву:

```
len    equ     5
a      db      3,5,6,7,6
       . . .
       mov     cx,len          ; заносимо до cx довжину масиву
       xor     ax,ax           ; обнуляємо ax
       xor     si,si           ; обнуляємо si
       . . .
cycl:jcxz      exit            ; перехід на exit якщо cx = 0
       add     ax,a[si]        ; до ax додаємо елемент масиву
       inc     si              ; збільшуємо si на 1
       dec     cx              ; збільшуємо cx на 1
       jmp     cycl            ; перехід на cycl
exit:mov       ax,4c00h
       int21h
```

Інший приклад фрагмента програми, що використовує команду умовного переходу для організації циклу:

```
       mov     cx,len          ; заносимо до cx довжину масиву
```

```
        . . .
metka:
        тело цикла
        dec     cx              ; зменшуємо cx на 1
        cmp     cx,0            ; порівнюємо cx с 0
        jne     metka           ; перехід на metka
```

У мові асемблера є спеціальні команди для програмування циклів: **loop**, **loope**, **loopz**, **loopne**, **loopnz**. Ці команди використовують в якості лічильника ітерацій регістр **cx**, а останні чотири команди - ще й значення прапора **zf**. Таким чином, в регістр **cx** необхідно попередньо записувати значення числа ітерацій.

Перераховані команди реалізують тільки короткі переходи на мітку (від -128 до 127 байтів), тому для організації циклів з тілом циклу більше 127 байт необхідно використовувати команди безумовного переходу.

Команда **LOOP** дозволяє організувати цикл з відомим числом повторень:

```
        mov     ecx, n
L:
        . . .
        loop    L
```

Команда **LOOP** вимагає, щоб в якості лічильника циклу використовувався регістр **CX**. Власне, команда **LOOP** віднімає одиницю саме з цього регістра, порівнює отримане значення з нулем і здійснює перехід на вказану мітку, якщо значення в регістрі **CX** більше 0. Мітка визначає зсув переходу, яке не може перевищувати 128 байт.

При використанні команди **LOOP** слід також враховувати, що з її допомогою реалізується цикл з постумовою, отже, тіло циклу виконується хоча б один раз. Гірше того, якщо до початку циклу записати в регістр **CX** значення 0, то при відніманні одиниці, яке виконується до порівняння з нулем, в регістрі **CX** виявиться ненульове значення, і цикл буде виконуватися 2^{32} раз.

Команда **LOOP** не відноситься до найшвидшим командам. У більшості випадків її можна замінити послідовністю інших команд.

Команди **LOOPE/LOOPZ** і **LOOPNE/LOOPNZ**. Ці команди схожі на команду **LOOP**, але дозволяють також організувати і достроковий вихід з циклу.

```
LOOPE <метка>          ; Команди є синонімами
LOOPZ <метка>
```

Дію цієї команди можна описати таким чином: : **CX** = **CX** - 1; якщо (**CX** != 0 и **ZF** == 1) перехід на мітку.

До початку циклу в регістр **CX** необхідно записати число повторень циклу. Команда **LOOPE/LOOPZ**, як і команда **LOOP** ставиться в кінці циклу, а перед нею розміщується команда, яка змінює прапор **ZF** (зазвичай це команда порівняння **CMP**). Команда **LOOPE**/**LOOPZ** змушує цикл повторюватиметься **CX** раз, але тільки якщо попередня команда фіксує рівність порівнюваних величин (виробляє нульовий результат, тобто **ZF** = 1).

З якої саме причини стався вихід з циклу треба перевіряти після циклу. Причому треба перевіряти прапор **ZF**, а не регістр **CX**, тому що умова **ZF** = 0 може з'явитися якраз на останньому кроці циклу, коли і регістр **CX** стане нульовим.

Команда **LOOPNE/LOOPNZ** аналогічна команді **LOOPE/LOOPZ**, але достроковий вихід з циклу здійснюється, якщо **ZF** = 1.

Розглянемо приклад: нехай у регістрі **SI** знаходиться адреса початку деякого масиву подвійних слів, а в змінній n - кількість елементів масиву, потрібно перевірити наявність в масиві елементів, кратних заданому числу **x**, і занести в змінну **f** значення 1, якщо такі елементи є, і 0 в іншому випадку.

```
        mov    ebx,x         ; заносимо в ebx  x (перевірка на
                             ; кратність)
        mov    ecx,n         ; заносимо в ecx  n (кількість елементів
                             ; масиву)
        mov    f,1           ; заносимо в f 1
L1:     mov    eax,[esi]     ; заносимо в eax адресу 0-ого елемента
                             ; масиву
        add    esi,4         ; збільшуємо eax на 4 байта (перехід на
                             ; наступний елемент масиву подвійних
                             ; слів)
        cdq                  ; перед поділом перетворюємо eax в
                             ; edx:eax
        idiv ebx             ; ділимо edx:eax на ebx (що містить
                             ; зміну x)
```

```
        cmp     edx,0          ; перевірка залишку від ділення на 0
        loopne  L1             ; зменшення ecx на 1 и перехід на L1,
                               ; якщо ecx != 0 і ZF = 0, якщо ecx = 0
                               ; або ZF = 1, то виконується наступна
                               ; команда
        je      L2             ; якщо ZF = 1, то перехід на L2, якщо ні
                               ; то виконується наступна команда
        mov     f,0            ; заносимо в f 0
L2:
```

6.4 Обробка масивів

Команда **LEA** здійснює завантаження в регістр так званої ефективно адреси:

```
        LEA     <регистр>, <ячейка памяти>
```

Команда не змінює прапори. У найпростішому випадку за допомогою команди **LEA** можна завантажити в регістр адресу змінної або початку масиву:

```
        x       dd    100 dup (0)
        lea     ebx, x
```

Розглянемо приклади обробки масивів. Нехай є масив **x** і змінна **n**, що зберігає кількість елементів цього масиву.

```
        x       dd    100 dup(?)
        n       dd    ?
```

Для обробки масиву можна використовувати кілька способів. Розглянемо способів знаходження суми елементів масиву.

Приклад 1. У регістрі можна зберігати зміщення елемента масиву.

```
        mov     eax,0
        mov     ecx,n          ; заносимо до ecx  n (кількість
                               ; елементів масиву)
        mov     ebx,0
L:      add     eax,x[bx]      ; додаємо до eax 0-ой елемент масиву
        add     ebx,type x     ; ebx збільшуємо на розмір типу x
                               ; (4 байта), тобто перехід на наступний
                               ; елемент масиву
        dec     ecx            ; зменшення ecx на 1
        cmp     ecx,0          ; порівняння ecx з 0
```

```
        jne     L               ; якщо ZF = 0, то перехід на L
```

Приклад 2. В регістрі можна зберігати номер елемента масиву і множити його на розмір елемента.

```
        mov     eax, 0
        mov     ecx, n          ; заносимо до ecx  n (кількість
                                ; елементів масиву)
L:      dec     ecx             ; зменшення ecx на 1
        add     eax,x[ecx*type x]  ; одаємо до eax n-ий елемент
                                   ; масиву
        cmp     ecx,0           ; порівняння ecx з 0
        jne     L               ; якщо ZF = 0, то перехід на L
```

Приклад 3. В регістрі можна зберігати адресу елемента масиву. Адресу початку масиву можна записати в регістр за допомогою команди **LEA**.

```
        mov eax, 0
        mov     ecx,n           ; заносимо до ecx  n (кількість
                                ; елементів масиву)
        lea     ebx,x           ; заносимо до ebx зсув 0-ого елемента
                                ; масиву
L:      add     eax,[ebx]       ; додаємо до eax 0-ой елемент масиву
        add     ebx, type x     ; ebx збільшуємо на розмір типу x
                                ; (4 байта), тобто перехід на наступний
                                ; елемент масиву
        dec     ecx             ; зменшення ecx на 1
        cmp     ecx,0           ; порівняння ecx з 0
        jne     L               ; якщо ZF = 0, то перехід на L
```

Приклад 4. При необхідності можна в один регістр записати адресу початку масиву, а в іншій - номер або зсув елемента масиву.

```
        mov     eax,0
        mov     ecx,n           ; заносимо до ecx  n (кількість
                                ; елементів масиву)
        lea     ebx,x           ; заносимо до ebx зсуву 0-ого елемента
                                ; масиву
L:      dec     ecx             ; зменшення cx на 1
        add     eax,[ebx+ecx*type x]    ; додаємо до eax  n-ий
                                        ; елемент масиву
        cmp     ecx,0           ; порівняння ecx з 0
```

```
        jne     L                   ; якщо ZF = 0, то перехід на L
```

Модифікацію адреси можна проводити також за двома регістрами: **x[ebx][esi]**. Це може бути зручно при роботі зі структурами даних, які розглядаються як матриці. Розглянемо для прикладу підрахунок кількості рядків матриць з позитивною сумою елементів.

```
        mov     esi,0               ; початковий зсув рядка
        mov     ebx,0               ; ebx буде містити кількість рядків
                                    ; задовольняючій умові
        mov     ecx,m               ; завантажуємо в ecx кількість рядків
L1:     mov     edi,0               ; початковий зсув елемента в рядку
        mov     eax,0               ; eax буде містити суму елементів рядка
        mov     edx,n               ; завантажуємо в edx кількість
                                    ; елементів в рядку
L2:     add     eax,y[esi][edi]     ; додаємо до eax елемент масиву
        add     edi,type y          ; додаємо до зсуву елемента в рядку
                                    ; розмір елемента
        dec     edx                 ; зменшуємо на 1 лічильник
                                    ; внутрішнього циклу
        cmp     edx,0               ; порівняння edx з нулём
        jne     L2                  ; якщо edx не дорівнює 0, то перехід до
                                    ; начала циклу
        cmp     eax,0               ; після циклу порівнюємо суму
                                    ; елементів рядка з нулем
        jle     L3                  ; якщо сума менше або дорівнює 0, то
                                    ; обходимо збільшення ebx
        inc     ebx                 ; якщо ж суму більше 0, то збільшуємо
                                    ; ebx
L3:     mov     eax,n               ; загружаємо до eax кількість елементів
                                    ; в рядку
        imul    eax,type y          ; множимо кількість елементів в рядку
                                    ; на розмір елемента
        add     esi,eax             ; додаємо до зсуву отриманий розмір
                                    ;  рядка
        dec     ecx                 ; зменшуємо на 1 лічильник зовнішнього
                                    ; циклу
```

```
cmp     ecx,0          ; порівнюємо ecx з нулем
jne     L1             ; якщо ecx не дорівнює 0, то перехід до
                       ; началу циклу
```

ТЕМА 7. Макрозасоби мови Асемблер

7.1 Основні поняття макрозасобів

Для попередньої підготовки вихідної програми до компіляції розроблена спеціальна макромова, у якої є спеціальні команди, які називають макрозасобами. Ці команди дозволяють, наприклад, продублювати деякі ділянки програми, або в залежності від умов виключити з компіляції фрагменти програми, оформити часто використовувані частини програми у вигляді макросів.

Фрагмент програми, оформлений спеціальним чином для подальшої підстановки, називають ***макросом***. Таке оформлення або опис називають ***макровизначенням***, посилання на макрос - ***макрокомандою***, процес заміни макрокоманди на макрос - ***макропідстановкою***, а результат макропідстановки - ***макророзширенням***.

Програма, написана за допомогою макрозасобів, транслюється в два етапи. На першому етапі (макрогенерації) в програмі виконуються всі макропідстановки, тобто виконується заміна макрокоманд на асемблерний код згідно макровизначенням. Макрогенерацію виконує спеціальний транслятор - ***макрогенератор***. На другому етапі здійснюється звичайне асемблювання.

7.2 Макровизначення

Макровизначення зазвичай розміщують на початку програми або в окремому модулі (файлі). В останньому випадку вони підключаються до програми за допомогою директиви **include**. Нижче наведено приклад визначення найпростішого макросу завершення роботи програми:

```
exit  macro         ; визначення макросу завершення програми
     mov    ax,4c00h
     int 21h
     endm
model small
stack 256
```

```
     .data
     . . .
     .code
start: mov ax,@data
     mov    ds,ax
     . . .
     exit                   ; макрокоманда
     end    start
```

7.3 Особливості використання макросів

Приклад макросу з параметром. Параметр **str** є формальним параметром. У макросі використовується збереження в стеку вмісту регістра **ax**. Такий прийом робить макрос надійніше, оскільки він сам використовує цей регістр. Рекомендується зберігати в макросах вміст використовуваних регістрів, щоб не виникали конфлікти з основною програмою.

```
     out_strmacro str
     push ax
     mov  ah,09h
     lea  dx,str
     int21h
     pop  ax
     endm
```

Якщо цей код зберегти у вигляді окремого модуля з ім'ям, наприклад, **out.inc**, то використовувати його можна в багатьох програмах, наприклад:

```
     title  programm with macro
     model  small
     includeout.inc
     .stack100h
     .data
     lang   db    'ASSEBLER',10,13,'$'
     .code
     . . .
     out_strlang
```

Макроси можуть мати свої внутрішні мітки і при виклику макросу більше одного разу виникне помилка подвійного визначення макросу, наприклад, якщо визначений макрос пошуку входження символу в рядок:

```
     find   macro symb
```

```
      mov     al,symb
m1:   inc     si
      cmp     al,str1[si]
      loopne m1
```

то при використанні його двох разів макрогенератором згенерується така програма:

```
; вихідний код                ; код після макрогенератора
      . . .                          . . .
      mov     bl,'a'
      find    bl                     mov    al,bl
                              m1:    inc    si
                                     cmp    al,str1[si]
                                     loopne m1
      . . .                          . . .
      mov     bl,'c'
      find    bl                     mov    al,bl
                              m1:    inc    si
                                     cmp    al,str1[si]
                                     loopne m1
```

Компілятор в цьому випадку, звичайно, повідомить про помилку. Щоб уникнути цього використовується спеціальна директива макромови **LOCAL N1,N2,…,Nk**, где **N1,N2,…,Nk** імена, що використовуються в макровизначенні. Макрогенератор виконує спеціальні заміни цих імен на імена виду **??0000**, **??0001** і так далі до **??FFFF**. Приклад:

```
      find    macro symb
      local m1
      mov     al,symb
m1:   inc     si
      cmp     al,str1[si]
      loopne m1
      . . .
      xor     si,si
      mov     cx,len
      mov     bl,'a'
      find    bl
      xor     si,si
      mov     cx,len
      mov     bl,'c'
      find    bl
```

Макрогенератор згенерує текст:

```
              . . .
              mov   al,bl
??0000:       inc   si
              cmp   al,str1[si]
              loopne ??0000
              . . .
              mov   al,bl
??0001:       inc   si
              cmp   al,str1[si]
              loopne ??0001
```

ТЕМА 8. Процедури

8.1 Визначення процедури

При написанні програми можуть з'являтися ділянки коду, що повторюються. Вони можуть бути невеликими, а можуть займати і досить багато місця. Великі фрагменти коду, що повторюються, істотно ускладнюють читання тексту програми, знижуючи її наочність, ускладнюючи налагодження, і служать невичерпним джерелом помилок. У мові асемблера є кілька засобів, які вирішують проблему дублювання ділянок програмного коду. Одним з таких засобів є використання процедур.

Процедура (часто ще її називають ***підпрограмою***) - це основна функціональна одиниця декомпозиції деякої задачі, тобто поділу на кілька частин. Процедура представляє собою групу команд для вирішення конкретної підзадачі і володіє засобами отримання управління з точки виклику задачі більш високого рівня і повернення управління в цю точку. У найпростішому випадку програма складається хоча б з однієї процедури.

Процедуру можна визначити як правильним чином оформлену сукупність команд, яка, будучи одноразово описаної, при необхідності може бути викликана в будь-якому місці програми. Для опису послідовності команд у вигляді процедури в мові асемблера використовуються дві директиви: **PROC** і **ENDP**.

Синтаксис опису процедури поданий у таблиці 8.1.

У заголовку процедури (директиві **PROC**) обов'язковим є тільки завдання імені процедури. Серед великої кількості операндів директиви **PROC** слід особливо виділити [відстань]. Цей атрибут може

приймати значення **near** або **far** і характеризує можливість звернення до процедури з іншого сегмента коду. Замовчуванням атрибут [відстань] приймає значення **near**.

Таблиця 8.1 - Синтаксис опису процедури

і'мя_процедури PROC [[модифікатор_мови] мова] [відстань] [ARG перелік_аргументів] [RETURNS перелік_аргументів] [LOCAL перелік_аргументів] [USES перелік_регістрів]	***Заголовок процедури***
Команди, директиви мови асемблера	***Тіло процедури***
[і'мя_процедури] ENDP	***Кінець процедури***

Процедура може розміщуватися в будь-якому місці програми, але так, щоб на неї випадковим чином не потрапило управління (якщо це заздалегідь не передбачається програмістом). Якщо процедуру просто вставити в загальний потік команд, то мікропроцесор буде сприймати команди процедури як частину цього потоку і, відповідно, буде здійснювати виконання команд процедури. З огляду на цю обставину, є такі варіанти розміщення процедури в програмі:

– на початку програми (до першої виконуваної команди);

– в кінці (після команди, яка повертає керування операційній системі);

– проміжний варіант - тіло процедури розташовується всередині іншої процедури або основної програми. В цьому випадку необхідно передбачити обхід процедури за допомогою команди безумовного переходу **jmp**;

– в іншому модулі.

Розміщення процедури на початку сегмента коду передбачає, що послідовність команд, обмежена парою директив **PROC** і **END**, буде розміщена до мітки, що позначає першу команду, з якої починається виконання програми. Ця мітка повинна бути вказана як параметр директиви **END**, що позначає кінець програми:

```
model small
.stack 100h
.data
.code
```

```
my_proc proc near
. . .
ret
my_proc endp
start:
. . .
end start
```

Оголошення імені процедури в програмі рівнозначно оголошенню мітки, тому директиву **PROC** в окремому випадку можна розглядати як деяку форму визначення мітки в програмі. Тому сама програма, що виконується також може бути оформлена у вигляді процедури, що досить часто і робиться з метою позначити першу команду програми, з якою має починатися виконання. При цьому ім'я цієї процедури потрібно обов'язково вказувати в заключній директиві **END**. Так, останній розглянутий фрагмент буде абсолютно еквівалентний наступному:

```
    model small
    .stack 100h
    .data
    .code
    my_proc proc near
    . . .
    ret
    my_proc endp
start proc
    . . .
    start endp
    end start
```

У цьому фрагменті після завантаження програми в пам'ять управління буде передано першій команді процедури з ім'ям **start**.

Розміщення процедури в кінці програми передбачає, що послідовність команд, обмежена директивами **PROC** і **ENDP**, буде розміщена після команди, яка повертає керування операційній системі.

```
model small
.stack 100h
.data
.code
start:
. . .
```

```
mov ax, 4c00h
int 21h                  ; повернення управління ОС
my_proc proc near
. . .
ret
my_proc endp
end start
```

Проміжний варіант розташування тіла процедури передбачає її розміщення всередині іншої процедури або основної програми. В цьому випадку необхідно передбачити обхід тіла процедури (за винятком певного роду процедур з подвійною обробкою), обмеженого директивами **PROC** і **ENDP**, за допомогою команди безумовного переходу **jmp**:

```
model small
.stack 100h
.data
.code
start:
. . .
jmp m1
my_proc proc near
. . .
ret
my_proc endp
m1:
. . .
mov ax, 4c00h
int 21h                  ; повернення управління ОС
end start
```

Останній варіант розташування описів процедур - в окремому сегменті коду - припускає, що процедури, які часто використовуються, виносяться в окремий файл. Файл з процедурами повинен бути оформлений як звичайний вихідний файл (за винятком вказівки імені мітки/процедури в директиві **END**) і підданий трансляції для отримання об'єктного коду. Згодом цей об'єктний файл необхідно об'єднати з файлом, де процедури використовуються, за допомогою програми лінковки (зв'язки).

Так як ім'я процедури має ті ж атрибутами, що і мітка в команді переходу, то звернутися до процедури, в принципі, можна за

допомогою будь-якої команди переходу. Але є одна важлива властивість, яка можна використовувати завдяки спеціальному механізму виклику процедур. Суть полягає в можливості збереження інформації про стан програми в точці виклику процедур. В системі команд мікропроцесора є дві команди, які здійснюють роботу з контекстом. це команди **call** та **ret**:

call [модифікатор] і'мя_процедури – виклик процедури.

Команда **call**, подібно **jmp**, передає управління за адресою з символічним ім'ям **імя_процедури**, але при цьому в стеку зберігається адреса повернення. Адреса повернення - це адреса команди, наступної після команди **call**.

ret [число] - повернути управління програмі, яка викликає. Команда **ret** зчитує адресу зі стека і завантажує її в регістри **cs** та **ip/eip**, тим самим повертаючи управління на команду, наступну в програмі за командою **call**.

[число] - необов'язковий параметр, що позначає кількість елементів, що видаляються зі стеку при поверненні з процедури. Розмір елемента визначається параметрами директиви **segment** - **use16** або **use32** (або відповідним параметром спрощених директив сегментації). Якщо вказано **use16**, то **[число]** - це значення в байтах; якщо **use32** - в словах.

Для команди **call**, як і для **jmp**, актуальна проблема організації ближніх і дальніх переходів. Це видно з формату команди, де присутня **[модифікатор]**. Як у випадку команди **jmp**, виклик процедури командою **call** може бути:

– внутрішньосегментним - процедура знаходиться в поточному сегменті коду (має тип **near**), і в якості адреси повернення команда **call** зберігає тільки вміст регістра **ip/eip**, що цілком достатньо для здійснення повернення;

– міжсегментним - процедура знаходиться в іншому сегменті коду (має тип **far**), і для забезпечення повернення команда **call** повинна запам'ятати вміст обох регістрів, **cs** та **ip/eip**. Черговість розміщення їх в стеку така: спочатку **cs**, потім **ip/eip**.

Важливо відзначити, що одна і та ж процедура не може бути процедурою і ближнього, і далекого типів. Таким чином, якщо процедура використовується в поточному сегменті коду, але може

викликатися і з іншого сегмента програми, то вона повинна бути оголошена процедурою типу **far**.

Подібно команді **jmp**, існують чотири різновиди команди **call**. Яка саме команда буде сформована, залежить від значення **[модифікатор]** в команді виклику процедури **call** і атрибута дальності в описі процедури. Якщо процедура описана на початку сегмента даних із зазначенням дальності в її заголовку, то при її виклику **[модифікатор]** можна не вказувати. Якщо ж процедура описана після її виклику, наприклад в кінці поточного сегмента або в іншому сегменті, то при її виклику потрібно вказати асемблеру тип виклику, щоб він міг за один прохід правильно сформувати команду **call**. Значення [**[модифікатор]** такі ж, як і у команди **jmp**, за винятком **short ptr** тобто:

– **near ptr** – прямий перехід на мітку всередині поточного сегмента коду. Модифікує тільки регістр **ip/eip** (в залежності від заданого типу сегмента коду **use16** або **use32**) на основі зазначених в команді адреси (мітки) або виразу, що використовує символ вилучення значення лічильника адреси команд – $;

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

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

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

8.2 Розміщення і передача аргументів процедур

Процедури можуть отримувати або не отримувати параметри з процедури, що викликається, і можуть повертати чи не повертати результати (процедури, які що-небудь повертають, називаються

функціями в мові Сі, але в асемблер не робить будь-яких відмінностей між ними).

Параметри можна передавати за допомогою одного з шести механізмів:

– за значенням;
– за посиланням;
– по поверненню значенням;
– по результату;
– по імені;
– відкладеним обчисленням.

Параметри можна передавати в одному з п'яти місць:

– в регістрах;
– в глобальних змінних;
– в стеці;
– в потоці коду;
– в блоці параметрів.

Отже, всього в асемблері можливо 30 різних способів передачі параметрів для процедур. Розглянемо деякі.

Передача параметрів за значенням. Процедурі передається власне значення параметра. При цьому фактично значення параметра копіюється, і процедура використовує його копію, так що модифікація вихідного параметра виявляється неможливою. Цей механізм застосовується для передачі невеликих параметрів, таких як байти або слова, наприклад, якщо параметри передаються в регістрах:

```
mov  ax, word ptr value    ; зробити копію значення
call procedure             ; викликати процедуру
```

Передача параметрів за посиланням. Процедурі передається не значення змінної, а її адресу, за якою процедура сама прочитає значення параметра. Цей механізм зручний для передачі великих масивів даних і в тих випадках, коли процедура повинна модифікувати параметри (хоча він і повільніше через те, що процедура буде виконувати додаткові дії для отримання значень параметрів).

```
mov  ax, offset value
call procedure
```

Передача параметрів по поверненню значенням. Цей механізм об'єднує передачу за значенням і за посиланням. Процедурі передають

адресу змінної, а процедура робить локальну копію параметра, потім працює з нею, а в кінці записує локальну копію назад по переданої адресі. Цей метод ефективніше звичайної передачі параметрів по посиланню в тих випадках, коли процедура повинна звертатися до параметру велику кількість разів, наприклад, якщо використовується передача параметрів в глобальній змінній:

```
mov global_variable, offset value
call procedure
. . .
procedure proc near
mov dx, global_variable
mov ax, word ptr [dx]      ; команди, що працюють з AX в
                           ; циклі десятки тисяч разів
mov word ptr [dx], ax
procedure endp
```

Передача параметрів по результату. Цей механізм відрізняється від попереднього тільки тим, що при виклику процедури попереднє значення параметра ніяк не визначається, а передана адреса використовується тільки для запису в нього результату.

Тепер розглянемо варіанти, де параметри передавати.

Передача параметрів в регістрах. Якщо процедура отримує число параметрів, ідеальним місцем для їх передачі виявляються регістри. Прикладами служать практично всі виклики переривань DOS і BIOS. Мови високого рівня зазвичай використовують регістр **AX(EAX** для того, щоб повертати результат роботи функції.

Передача параметрів в глобальних змінних. Коли не вистачає регістрів, один із способів обійти це обмеження - записувати параметр в змінну, до якої потім слід звертатися з процедури. Цей метод вважається неефективним.

Передача параметрів в стеку (на цей спосіб слід звернути особливу увагу). Параметри поміщаються в стек відразу перед викликом процедури. Саме цей метод використовують мови високого рівня, такі як Сі і Pascal. Для читання параметрів з стека зазвичай застосовують не команду **POP**, а регістр **BP**, в який поміщають адресу вершини стека після входу в процедуру:

```
push parameter1      ; помістити параметр в стек
push parameter2
call procedure
```

```
add sp,4              ; звільнити стек від параметрів
. . .
procedure proc near
push bp
mov bp,sp             ; команди, які можуть використовувати
                      ; стек
mov ax, [bp+4]        ; зчитати параметр 2, його адреса в
                      ; сегменті стека BP+4, тому що при
                      ; виконанні команди CALL в стеці
                      ; помістили адресу повернення - 2 байта
                      ; для процедури типу NEAR (або 4 - для
                      ; FAR), а потім ще і BP - 2 байта
mov bx, [bp+6]        ; зчитати параметр 1
. . .
pop bp
ret
procedure endp
```

Параметри в стеці, адреса повернення і старе значення **BP** разом називаються ***активізаційним записом функції***.

Для зручності посилань на параметри, передані в стеці, всередині функції іноді використовують директиви **EQU**, щоб не писати кожен раз точний зсув параметра від початку активізаційного запису (тобто **BP**), наприклад так:

```
push X
push Y
push Z
call xyzzy
. . .
xyzzy proc near
xyzzy_z equ [bp+8]
xyzzy_y equ [bp+6]
xyzzy_x equ [bp+4]
push bp
mov bp,sp       ; команди, які можуть використовувати стек
mov ax,xyzzy_x        ; зчитати параметр x
. . .
pop bp
ret 6
xyzzy endp
```

При уважному аналізі цього методу передачі параметрів виникає відразу два питання: хто повинен видаляти параметри зі стеку, процедура або програма, яка її викликає, і в якому порядку поміщати параметри в стек. В обох випадках виявляється, що обидва варіанти мають свої «за» і «проти». Так, наприклад, якщо стек звільняє процедура (командою **RET число_байтів**), то код програми виходить меншим, а якщо за звільнення стеку від параметрів відповідає функція, як в попередньому прикладі, то стає можливим послідовними командами **CALL** викликати кілька функцій з одними і тими ж параметрами. Перший спосіб, більш строгий, використовується при реалізації процедур в мові PASCAL, а другий, що дає більше можливостей для оптимізації, - в мові Сі. Зрозуміло, якщо передача параметрів через стек застосовується і для повернення результатів роботи процедури, з стека не треба видаляти всі параметри, але популярні мови високого рівня не користуються цим методом. Крім того, в мові Сі параметри поміщаються в стек у зворотному порядку (справа наліво), так що стають можливими функції із змінним числом параметрів.

ТЕМА 9. Зв'язок з іншими мовами програмування високого рівня

Існують наступні форми комбінування програм на мовах високого рівня з асемблером:

– використання операторів типу **inline** і асемблерних вставок;

– використання зовнішніх процедур і функцій.

Перша форма сильно залежить від синтаксису мови високого рівня і конкретного компілятора. Вона передбачає, що асемблерні коди у вигляді команд асемблера або прямо в машинних командах вставляються в текст програми на мові високого рівня. Компілятор мови розпізнає їх як команди асемблера (машинні коди) і без змін включають в формований їм об'єктний код. Ця форма зручна, якщо треба вставити невеликий фрагмент.

Друга форма комбінування більш універсальна. У неї є ряд переваг:

– написання і налагодження програм можна робити незалежно;

– написання підпрограми можна використовувати в інших проектах;

– полегшуються модифікація і супровід підпрограм протягом життєвого циклу проекту. Наприклад, якщо процедура на асемблері виробляє роботу з деяким зовнішнім пристроєм, то при зміні пристрою немає необхідності міняти весь проект - досить замінити тільки працюючу з ним процедуру, залишивши незмінним інтерфейс програми на мові високого рівня з асемблером.

Так як використання оператора **`inline`** і асемблерних вставок сильно залежить від синтаксису конкретної мови, то ми розглядати їх не будемо.

9.1 Використання зовнішніх процедур і функцій

Розглянемо використання зовнішніх процедур і функцій для комбінування програм на мовах високого рівня з асемблером. Згадаймо синтаксис директиви **PROC**:

```
i'мя_процедури PROC [[модифікатор_мова] мова]
                     [відстань]
```

Один з операндів - мова. Він служить для того, щоб компілятор міг правильно організувати інтерфейс (зв'язок даних) між процедурою на асемблері і програмою на мові високого рівня. Необхідність такої вказівки виникає внаслідок того, що способи передачі аргументів при виклику процедур різні для різних мов високого рівня.

Найбільш поширені компілятори програм на асемблері (MASM, TASM, NASM) підтримують кілька значень операнда мови. У таблиці 9.1 для деяких з них наведено характерні особливості передачі аргументів і угоди про те, яка процедура очищає стек - яка викликається або яка викликає. Під напрямком передачі аргументів розуміється порядок, в якому аргументи включаються в стек, в порівнянні з порядком їх слідування у виклику процедури. Так, наприклад, для мови PASCAL характерний прямий порядок включення аргументів в стек: першим в стек записується перший аргумент, який передається, з оператора виклику процедури, другим - другий аргумент і т.д. На вершині стека після запису всіх переданих аргументів виявляється останній аргумент. Для мови Сі, навпаки, характерний зворотний порядок передачі аргументів. Відповідно до нього в стек спочатку включається останній аргумент з оператора виклику процедури (або функції), потім передостанній і т.д. В

кінцевому підсумку на вершині стека виявляється перший аргумент. Що ж стосується очищення стека, то зрозуміло, що повинні бути певні домовленості про це. У мові PASCAL цю операцію завжди робить процедура, яка викликається, в мові Сі – яка викликає.

Таблиця 9.1 - Передача аргументів на мовах високого рівня

Операнд «мова»	Мова	Напрямок передачі	Яка процедура очищає стек
NOLANGUAGE	Ассемблер	Зліва направо	що викликається
BASIC	Basic	Зліва направо	що викликається
PROLOG	Prolog	Справа наліво	що викликає
FORTRAN	Fortran	Зліва направо	що викликається
C	C	Справа наліво	що викликає
C++ (CPP)	C++	Справа наліво	що викликає
PASCAL	Pascal	Зліва направо	що викликається
STDCALL	-	Справа наліво	що викликається
SYSCALL	C++	Справа наліво	що викликає

Більшість мов високого рівня передають параметри процедурі, що викликається, в стеку і очікують повернення параметрів в регістрі **AX(EAX)**. Іноді використовуються **DX:AX** (**EDX:EAX**), якщо результат не вміщується в одному регістрі, і **ST(0)**, якщо результат - число з плаваючою комою. В таблиці 9.2 приводяться регістри, які використовуються для повернення різних типів даних.

Таблиця 9.2 - Типи результатів функцій (стосовно Win32)

Тип даних	Байти	DOS Регістр(и)	Байти	Win32 Регістр(и)
unsigned char	1	al	1	al
char	1	al	1	al
enum	2	ax	2	ax
unsigned shor	2	ax	2	ax
short	2	ax	2	ax
unsigned int	4	ax	2	eax
int	4	ax	2	eax
unsigned long	4	dx:ax	4	eax
long	4	dx:ax	4	eax
float	4	st(0) (стек 8087)	4	st(0) (стек 8087)
double	8	st(0) (стек 8087)	8	st(0) (стек 8087)
long double	8	st(0) (стек 8087)	10	st(0) (стек 8087)

Продовження таблиці 9.2

указатель near	4	ax	5	eax
указатель far	4	dx:ax	2	eax

В мовах високого рівня при виклику процедур (функцій) локальні змінні процедури розміщуються в стек і існують тільки до тих пір, поки виконується процедура. У асемблерному модулі можна виконати те ж саме за допомогою директиви **LOCAL**. Наприклад:

```
PROC l_func NEAR
LOCAL i:WORD=stacksize
push ebp
mov ebp,esp
sub esp,stacksize
mov [i],0
...               ; код використовує локальну змінну [i]
mov esp,ebp       ; можливо add esp,stacksize,
                  ; але повільніше
pop ebp
ret
ENDP l_func
```

Директива **LOCAL** готує змінну **i** типу **WORD** (слово). Вказівка **=stacksize** призначає загальне число байтів, яке займає усіма локальними змінними - в даному випадку 2 байта. Це значення віднімається з **esp** після підготовки адресації змінних в стек. При використанні **LOCAL** немає необхідності обчислювати негативні зсуви щодо **ebp**, щоб визначити місце розташування змінних в стеку, - достатньо скористатися іменами змінних.

При організації зв'язку також можна використовувати директиву **ARG**. Це так само позбавить від необхідності підраховувати зсув в стеці для доступу до аргументів і дозволить звертатися до них просто по імені. Зазвичай директиву **ARG** опускають, якщо немає необхідності додатково виділити список аргументів процедури, і список аргументів описують одразу після вказівки **[відстань]** в оголошенні прототипу процедури:

```
PROC l_func NEAR ch:byte; y:word
; це еквівалентно запису:
; PROC l_func NEAR ARG ch:byte; y:word
LOCAL i:WORD=stacksize
```

```
push ebp
mov ebp,esp
sub esp, stacksize
mov [i],0
...              ; код використовує локальну змінну [i]
mov esp,ebp      ; можливо add esp,stacksize,
                 ; але повільніше
pop ebp
ret
ENDP l_func
```

9.2 Викривлення імен

Практично всі компілятори мови С (різних виробників і під різні ОС) змінюють назви процедур, щоб відобразити використовуваний спосіб передачі параметрів. Так, до назв усіх процедур, які застосовують С-конвенцію, додається символ підкреслення. Тобто, якщо в С-програмі записано

```
some_proc();
```

то реально компілятор пише

```
call _some_proc
```

і це означає: якщо процедура написана на асемблері, вона повинна називатися саме **some_proc** (або використовувати складну форму записи директиви **PROC**).

Назви процедур, що використовують **STDCALL** і знаходяться в **DLL**, спотворюються ще більш складним: спереду до назви процедури додається символ підкреслення, а ззаду - символ **@** і розмір області стека в байтах, яку займають параметри (тобто число, що стоїть після команди **RET** в кінці процедури).

```
some_proc(a:word);
```

перетворюється в

```
push a
call _some_proc@4
```

9.3 Виклик асемблерних процедур з С/С++

Для виклику ассмблерной процедури з модуля С/С++ необхідно оголосити прототип процедури, як для звичайної С/С++ функції, але попередньо дописавши на початку префікс використання функції з іншого модуля проекту:

extern "C" void proc();

Тим самим оголошується функція, що не повертає ніякого значення. Префікс **extern** повідомляє компілятору, що параметр знаходиться в іншому модулі (компілятору не потрібно знати, що функція буде написана на іншій мові). Взяте в лапки **"C"** відключає спотворення імені, даючи можливість використовувати в асемблерному модулі символ **_proc** замість викривленого імені (проте символ підкреслення все ще необхідний).

Виклик функції C++ з асемблера.

Виклик функцій C++ з ассмблерного модуля породжує таку ж проблему викривлення імен. Для простоти зазвичай краще відключати викравлення імен, використовуючи методику, а саме оголосити прототип функції C++ в наступному вигляді:

extern "C" void Terminate();

Незважаючи на той факт, що функція оголошена **extern**, вона все одно виконується в C++ в якому вона описана. Але компілятору все одно, як виконуються функції. "Зовнішня" функція може бути написана в іншому модулі на C++, асемблері або будь-якому іншому мовою. Зовнішні функції можуть бути написані в тому ж самому модулі, в якому вони оголошуються. Оголошення **extern** попереджає компілятор, щоб він не очікував виконання функції в поточному модулі. Єдиною причиною використання **extern** є прагнення відключити викривлення імені. Виклик ж функції з програми на асемблері повинен бути попередньо "підготовлений" - тобто необхідно оголосити прототип її виклику:

EXTRN _TERMINATE:PROC

Директива **EXTRN** специфікує функцію (**PROC**), названу **_Terminate**, яка існує в іншому модулі. Асемблеру немає необхідності знати, як ця функція виконується - досить того, що її немає в поточному модулі. Оголошення зовнішньої функції програмі дозволяє звертатися до неї, як

call _Terminate

Директиви, які використовуються в процедурах:

– **ENDP** – відзначає кінець процедури ім'я, попередньо розпочатої з директиви **PROC**:

ім'я ENDP

– **INVOKE** – викликає процедуру за адресою, яка задана параметром виразу, розміщаючи аргументи в стеку або в регістрах відповідно до стандартних угод про виклики конкретної мови (тип мови вказується в директиві **PROC**):

```
INVOKE вираз [, аргументи]
```

Кожен аргумент, який передається процедурі, може бути виразом, регістром, або адресним виразом (вираз, якому передує **ADDR**).

– **PROC** – зазначає початок блоку процедури, позначеного ідентифікатором мітка:

```
мітка PROC [мова] [відстань]
[USES перелік_регістрів] [, параметр[:тип]] ...
команди
мітка ENDP
```

Команди в блоці можуть викликатися в програмі командою **CALL** або директивою **INVOKE**.

– **PROTO** – визначає прототип процедури/функції:

```
мітка PROTO [відстань] [мова] [,[параметр]:тип] ...
```

Перелік джерел посилання

1. Юров В. "Assembler. Учебник." – СПб.: Питер, 2010. – 640с.
2. Титовский, С. В., Титовская Н. В. Языки программирования. Ассемблер. - Конспект лекций. - Красноярск: ИПК СФУ, 2008. -132 с. - ISBN 978-5-7638-1460-6.
3. Юров В. "Практикум. Assembler." – СПб.: Питер, 2006. – 400с.
4. Магда Ю.С. "Ассемблер для процессоров Intel Pentium" – СПб.: Питер, 2006. – 410 с: ил.
5. Марек Р. "Ассемблер на примерах. Базовый курс" – СПб.: Наука и Техника, 2005. – 240с.
6. Калашников О. "Ассемблер? Это просто! Учимся программировать" – БХВ-Петербург, 2007. – 374с.
7. Пирогов В.Ю. "Ассемблер для Windows" – М.: Издатель Молгачева С.В., 2002. – 552с.
8. Рудаков П.И., Финогенов К.Г. "Язык ассемблера: уроки программирования" – М.: ДИАЛОГ-МИФИ, 2001. – 640 с.
9. Пирогов В.Ю. "Assembler. Учебный курс" – Нолидж. 2000. – 354с.

10. Зубков С. В. “Для программистов. Assembler. Для DOS, Windows и Unix. 2-е изд.” – М.: ДМК Прес, 2000 – 608с.
11. Том Сван. “Освоение Turbo Assembler. Второе издание” – К.; М.; СПб.: Диалектика, 1996 – 544с.
12. TASM. [Електронний ресурс]: - Режим доступу: http://www.firststeps.ru/asm/tasm1.html.
13. Асемблер (assembler) і системне програмування. [Електронний ресурс]: - Режим доступу: http://www.znannya.org/?view=asm.
14. Програмування на асемблері під Windows MASM32 для Win32. [Електронний ресурс]: - Режим доступу: http://softihorweb.pp.net.ua/index/0-30.
15. Поляков, А.В. Ассемблер для чайников. [Электронный ресурс]: - Режим доступа: http://av-assembler.ru/asm/afd/assembler-for-dummy.htm.
16. Программирование на языке ассемблера. [Электронный ресурс]: - Режим доступа: http://natalia.appmat.ru/c&c++/assembler.html.
17. Как писать на ассемблере в 2018 году. [Электронный ресурс]: - Режим доступа: https://habrahabr.ru/post/345748.
18. Assembler. [Электронный ресурс]: - Режим доступа: http://progopedia.ru/language/assembler.
19. Андриенко, Дмитрий. Погружение в assembler. Полный курс по программированию на асме от][. [Электронный ресурс]: - Режим доступа: https://xakep.ru/2017/09/11/asm-course-1.