Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

І. М. Дудзяний, В.В. Черняхівський

Програмування мовою асемблера

Навчальний посібник

Рекомендовано до друку
Вченою Радою Львівського
національного університету
імені Івана Франка
Протокол № 10/2 від 27.02.02 р.

Львів
Видавничий центр ЛНУ імені Івана Франка
2002

УДК 004.438 Асемблер (075.8)
ББК 3973.2-018.19я73-1 Асемблер

Дудзяний І.М., Черняхівський В.В. Програмування мовою асемблера. Навчальний посібник. – Львів: ЛНУ імені Івана Франка, 2002. – 112 с.

У навчальному посібнику розглянуто основи програмування мовою асемблера для комп'ютерів на базі мікропроцесорів Intel. Матеріал посібника відображає теми, які вивчають студенти факультету прикладної математики та інформатики в рамках курсу "ЕОМ та програмування".

Навчальний посібник призначено для студентів факультету прикладної математики та інформатики, а також усіх бажаючих навчитися програмувати мовою асемблера. Матеріал посібника подає основні принципи внутрішньої будови комп'ютера та програмування мовою асемблера. До кожної теми посібника складено низку задач і вправ для самостійної роботи та практичного закріплення матеріалу. Виокремлені базові поняття можна використати як план самостійної роботи до кожної теми.

Рецензенти:
д-р техн. наук, професор В.В. Пасічник
канд. фіз.-мат. наук, доцент А.А. Переймибіда

Редактор Ірина Миколаївна Лоїк

Вступ

Зазвичай програмісти використовують для розробки своїх прикладних програм такі алгоритмічні мови, як Pascal, C++, Basic, Java, Prolog. Однак насправді такі мови, які ще називають мовами високого рівня, є посередниками між програмістом і комп'ютером. Комп'ютер їх безпосередньо не розуміє, тому необхідно перекласти програми на його рідну, *машинну*, мову, тобто компілювати програми.

Мова асемблера також потребує перекладу на машинну мову. Однак вона принципово відрізняється від алгоритмічних мов високого рівня тим, що практично один до одного відповідає машинній мові. За своїм змістом мова асемблера є системою позначень команд процесора (одна команда мови – одна команда процесора). Це єдина мова, яка дає змогу безпосередньо керувати командами процесора. За допомогою мови асемблера можна використовувати усі наявні можливості процесора і будувати максимально ефективні програми. Очевидно, що мова асемблера тісно пов'язана з будовою процесора та комп'ютера загалом. Отже, для програмування мовою асемблера необхідні знання архітектури процесора та основних принципів роботи комп'ютера.

Для професійного програмування необхідно добре знати, що відбувається у комп'ютері під час виконання програми. Вивчити внутрішні засади функціонування комп'ютера найкраще через мову асемблера з відповідним закріпленням матеріалу на практичних заняттях. Якщо ж програміст використовує для роботи лише мови високого рівня, знання асемблера дає змогу у багатьох випадках будувати кращі програми. А комбінування програми мовою високого рівня та мовою асемблера для цілої низки задач є найефективнішим рішенням. Крім того, існує певне коло задач, реалізувати які можна лише мовою асемблера.

Цей посібник дає змогу набути знань з основних принципів роботи комп'ютера та отримати початкові навички програмування мовою асемблера. Посібник відповідає розділові курсу "ЕОМ і програмування", який викладають студентам факультету прикладної математики та інформатики Львівського національного університету імені Івана Франка. Кожна тема посібника налічує дві частини: теоретичні положення та завдання для самостійної роботи. На початку теми перелічено базові поняття, які студент повинен чітко знати. Тлумачення базових понять наведено у першій частині кожної теми. Завдання для самостійної роботи допоможуть студентам закріпити матеріал письмовим опрацюванням, а також шляхом написання та налагодження відповідних програм.

Обізнаність у внутрішніх принципах роботи комп'ютера та мови асемблера – ключ до майбутніх успіхів професійного програміста, чого й бажають автори усім студентам.

Тема 1. Визначення даних

Базові поняття: *сегмент даних; байт, слово, подвійне слово, почетверене слово, десяток байтів; числові та символьні (літерні) дані; розташування старших і молодших розрядів чисел; ім'я змінної·чи константи;* **DB, DW, DD, DQ, DT***; декілька даних в одному рядку;* **DUP***; ініціалізація векторів і матриць.*

1.1. Теоретичні положення

1.1.1. Зображення цілих від'ємних чисел

У знакозмінній арифметиці цілих чисел крайній лівий біт цілого числа вказує на його знак. Додатні числа мають **0** (нуль) у старшому біті, а від'ємні **– 1** (одиницю). Додатні числа мають ідентичне значення у знаковій і беззнаковій арифметиці. Від'ємні **–** різні значення. Щоб зробити число від'ємним (змінити його знак на мінус) **–** його доповнюють і отриманий результат збільшують на одиницю. У чотирибітовому зображенні число **5** має значення **0101B**, в той час як **-5 – 1011B.** Дійсно:

5=0101B ⇒ *Доповнюємо*: **1010B** ⇒ *Додаємо* **1=0001B** ⇒ **-5=1011B.**

Наступний приклад вказує на єдиність нуля:

0=0000B ⇒ *Доповнюємо*: **1111B** ⇒ *Додаємо* **1=0001B** ⇒ **-0=0000B.**

Для будь-якого n−бітового числа в системі з двійковим доповненням найбільшим значенням є 2^{n-1}-1, а найменшим – -2^{n-1}. Нуль – єдиний. У чотирибітовій системі найбільшим числом є 7, а найменшим – -8 (див. табл. 1.1).

Табл. 1.1. Зображення двійкових чисел у чотирибітовій системі

Десяткове	Двійкове	Десяткове	Двійкове
7	0111	-1	1111
6	0110	-2	1110
5	0101	-3	1101
4	0100	-4	1100
3	0011	-5	1011
2	0010	-6	1010
1	0001	-7	1001
0	0000	-8	1000

Числа у шістнадцятковому записі позначають суфіксом **H**, а двійкові числа – суфіксом **B**. Десяткові числа пишуть без суфікса або з суфіксом **D**. Для зображення даних в асемблерній програмі можна користуватися будь-якою з трьох розглянутих систем: десятковою, двійковою чи шістнадцятковою.

Під час запису 16-кових чисел важливо пересвідчитися, що асемблер сприйме їх як число. Якщо введено **FAH**, то це може бути або шістнадцяткове число **FA**, або ім'я змінної **FAH**. Асемблер передбачає, що запис числа розпочинається з цифри, а імені змінної чи позначки – з букви. Тому **FAH** у мові асемблера виявляється змінною. Якщо ж ми маємо на увазі не змінну, а число, то його необхідно записати **0FAH**: це число має бажане значення і, відповідно, його запис розпочинається цифрою. Отже, кожному шістнадцятковому числу, запис якого розпочинається зі значень від **A** до **F**, повинен передувати **0** (нуль).

1.1.2. Константи

- **Двійкова** – послідовність цифр **0** і **1** , що завершується буквою **B** (наприклад, **11001010B**).
- **Десяткова** – послідовність цифр від **0** до **9**, що може завершуватися буквою **D** (наприклад, **419** або **419D**).
- **Шістнадцяткова** – послідовність цифр від **0** до **9** і букв від **A** до **F**, що завершується буквою **H**. Першим символом обов'язково повинна бути цифра (наприклад, **105H** або **0C04BH**).
- **Літерали** – рядок букв, цифр та інших символів у лапках (або між апострофами). Дві форми передбачено для того, щоб за необхідністю вставляти в текст самі лапки або апострофи. Наприклад: **"Значення функції"**, або **'Значення функції'**; **"Відповідь на питання 'Ваш вибір'".**
- **Від'ємні числа.** Якщо число – десяткове, то перед ним ставлять знак мінус (наприклад, **-26**). Якщо число двійкове або шістнадцяткове, то його вводять у доповнювальному коді. Число **–32**: **11100000B** або **0E0H**.

1.1.3. Формат мови асемблера

Кожну команду мови асемблера записують в один рядок. Формат рядка:

Поле імені	*Поле операції*	*Поле операндів*	*Поле коментаря*

або:

Поле (*рядок*) *коментаря*

Поля відокремлюють між собою одним чи декількома інтервалами (пропусками). Обов'язковими є лише поле операції та поле операндів, якщо операція вимагає операндів. Поле чи рядок коментаря розпочинають знаком '**;**'. Операнди розділяють між собою комами.

Неприпустимо використовувати пропуски всередині поля імені, поля операції чи поля операндів. Однак пропуск може бути літерою − значенням операнда у лапках.

Вважатимемо, що імена налічують не більше, ніж вісім латинських букв і цифр, а їхній запис обов'язково розпочинається буквою. Великі та малі букви вважатимемо однаковими. Наприклад:

Abc12, **aBc12**, **ABC12**, **abc12** - це однакові імена.

1.1.4. Біти, байти і слова

Бітом називають двійкову цифру, одиничне значення **0** або **1**. Групу з 8-ми бітів називають **байтом**. Байт заслужив своє власне ім'я з декількох причин. Елементарний елемент пам'яті (комірка) має довжину 8 бітів. Упродовж кожного звертання до памяті процесор опрацьовує рівно 8 бітів інформації. Байт - найменша одиниця інформації, з якою можна маніпулювати безпосередньо. Крім того, байт використовують для зображення одного символу. За допомогою одного байта можна визначити **256** (**2^8**) окремих символів (межі для цілих чисел: **-128 … 127**).

Для розміщення певного значення в одному байті пам'яті асемблер має в своєму розпорядженні спеціальний механізм визначення байта (*define byte*) − псевдокоманду **DB** формату:

```
[ < ім'я > ]  DB < число або літерал>
```

або

```
[ < ім'я > ]  DB ?
```

Отже, псевдокоманда **DB**:

- формує однобайтову константу − число;
- формує у пам'яті заданий рядок символів (літерал);
- просто резервує 1 байт пам'яті (за умови наявності **"?"**).

Псевдокоманда

```
R1   DB   23
```

дає асемблерові завдання зберегти десяткове значення 23 у деякому байті пам'яті, зазначеному як **R1**. Позначку **R1** можна трактувати як назву (ім'я) *константи* (значення байта **R1** у програмі змінюватися не буде) або назву *змінної* (значення байта **R1** у програмі змінюватиметься).

Псевдокоманда

```
R2   DB   1,2,3,4
```

зберігає значення від 1 до 4 за чотирма послідовними адресами в пам'яті.

Псевдокоманда

```
Wer  DB  ?
```

повідомляє асемблеру щодо необхідності виокремити (зарезервувати) один байт пам'яті, не визначаючи початково його вміст. У цьому байті може виявитися будь-яке випадкове число, яке там залишатиметься, доки будь-яка команда не вмістить у нього певне значення.

Часом треба виокремити значну кількість байтів, наприклад, щоб зарезервувати область пам'яті для масиву. Це можна зробити так:

```
Mas1  DB 25 DUP(?)
```

Утакий спосіб виокремлюють 25 байтів пам'яті. Ключове слово **DUP** у цій псевдокоманді означає *повторити* (*duplicate*). Число 25 вказує, скільки разів асемблер повторить визначення байта у пам'яті. Значення в дужках асемблер використовує з метою ініціалізації цієї області пам'яті. У нашому випадку це значення невідоме. Для ініціалізації області з ідентичним значенням, наприклад 31, записують так:

```
Mas2  DB 17 DUP(31)
```

Нарешті,

```
Mas3  DB 30 DUP(1,2,3,4,5)
```

виокремлює 30 груп по 5 байтів (загалом 150 байтів) зі значеннями від 1 до 5. Асемблер повторює значення в дужках доти, доки не буде заповнено усі 30 груп байтів.

Іноді необхідно звернутися до групи бітів, меншої від байта. Прийнято розмір 4 біти. У 4-х бітах можна зобразити усі 10 десяткових цифр. Для значень такого розміру використовують термін *півбайт*.

Термін *слово* має для програміста значення, відмінне від прийнятого у мові. У застосуванні до ЕОМ *слово* - це *найбільша* кількість бітів, з якою машина може працювати як з єдиним елементом. Для IBM/370 слово становить 32 біти, а для Intel 8088 – 16 бітів. Тому термін *слово* є невизначеним доти, доки не визначено конкретної машини.

Оскільки в посібнику розглянуто мову асеблера для комп'ютерів на базі мікропроцесорів Intel, то **розмір слова становить 16 бітів**. Цей розмір визначено каналами передачі даних у процесорі. Над числами до 16-ти бітів операції можуть здійснюватися однією командою.

Псевдокоманда **DW** *визначає слова* (*define word*). Як і у випадку з байтами, можна використати оператор **DUP** з метою визначення великих областей пам'яті, розбитих на слова. Для позначення ділянок пам'яті, що початково не ініціюються, використовують операнд "**?**".

Приклади визначення слів:

```
W1    DW  1234H
MasW  DW  3 DUP(5678H)
Rob   DW  ?
```

Хоч ми і визначали значення слова як **1234H**, асемблер збереже в пам'яті значення **3412H**. Припустимо, що **1234H** збережено у байтах з адресами **100** і **101**. Асемблер вміщує значення **34H** у байт **100**, а **12H** – в **101**. Найлегше запам'ятати це так, що асемблер зберігає молодший байт слова у байті пам'яті з меншою адресою, а старший байт слова – у байті пам'яті з більшою адресою.

На щастя, якщо ви не будете змішувати операції над байтами і над словами однієї і тієї ж ділянки пам'яті, то вам нічого турбуватися щодо цього несподіваного "перемикання" байтів. Програма може спокійно працювати зі словами, а мікропроцесор завжди зорієнтується у ситуації. Тільки у випадку, коли ви захочете звернутися до конкретного байта якого-небудь слова, вам доведеться мати справу з фактичним способом зберігання слів у пам'яті мікропроцесора.

Залишився ще один тип даних, який постійно використовують у програмах мовою асемблера для мікропроцесорів Intel. Це - *подвійне слово*, значення у 32 біти довжиною. У програмах використовують подвійні слова для зберігання адрес і дуже великих чисел. Щоб визначити область, яка містить значення подвійного слова, використовують псевдокоманду **DD** (*define doubleword*). Ця псевдокоманда генерує поле розміром у 4 байти. Так само, як у випадку з **DW**, асемблер розміщує в пам'яті молодший байт подвійного слова нижче, а старший – вище. У такому ж порядку зберігаються середні два байти. Аналогічно **DB** і **DW** можна користуватися функцією **DUP** і застосовувати операнд "**?**" для того, щоб залишити область невизначеною. Подібно визначається пам’ять для *почетвереного слова* (**DQ**) та поля у 10 байтів (**DT**).

Наведемо приклади фрагментів програм, що визначають дані:

```
.data?
; Змінні величини без початкових значень
v1 db ?
v2 dw ?
v3 dd ?

.data
; Змінні величини з початковими значеннями
c2 dw 0CH
c3 dw 101B
```

```
; Літерали
l1  db 'Practica'  ; 8 байтів
l2  dw 'Pr'        ; 2 байти
l3 dw 'Practica'  ; Помилка!

; Приклади векторів і матриць (оператор dup)
vec1 db 8 dup (?) ;  вектор з 8 елементів
vec2 dw 2 dup (3,4) ; вектор (3, 4, 3, 4)
vec3 dw 6 dup(0),6 dup(1) ; 0 0 0 0 0 0 1 1 1 1 1 1
matr1 dw 2 dup(2 dup(1)) ; ((1, 1),(1, 1)) - матриця 2 x 2
                         ; або вектор (1, 1, 1, 1)
matr2 dw 2 dup(3 dup(1),3 dup(2)) ;  1 1 1
                                  ;  2 2 2
                                  ;  1 1 1
                                  ;  2 2 2
; Визначення констант
.const
c1 db 4
```

```
; Якщо в програмі декілька разів записано деяке число
;(наприклад, 4), то для кожної четвірки буде виокремлено пам'ять.
;У такому випадку вигідніше описати константу, довжина якої 1 байт.
```

1.2. Завдання для самостійної роботи

1. Яка різниця між константою та змінною величиною? Чи можна змінювати значення константи під час виконання програми?
2. Яку роль відіграє порядок перелічування констант чи змінних?
3. Визначте дві однобайтові константи: 11; -15.
4. Визначте три константи розміром у слово, які мають значення 100; 99; 98.
5. Задайте значення чотирьох двобайтових констант 26; -12; 18; 74 трьома способами: а) через десяткові значення; б) через шістнадцяткові значення; в) через двійкові значення.
6. Зарезервуйте пам'ять для трьох змінних однобайтових величин.
7. Зарезервуйте пам'ять для двох однобайтових змінних величин і трьох двобайтових змінних величин.
8. Виокремте пам'ять для: п'яти констант 0; 0; 8; -1; -3; двох змінних величин **A1**; **A2**; літерала "**Практичне заняття**". Розмір констант і змінних – довільний.
9. Задайте у пам'яті два літерали: "**О-йо-йой, вже падає дощ**" та "**Ваші висновки:**". Скільки пам'яті буде виокремлено для кожного з цих літералів?
10. Задайте у пам'яті три різні літерали, вибрані вами довільно.

11. Визначте літерал "**Ми вчимося програмувати.**". Запишіть адресу першої та останньої літери кожного слова.

12. Чи є еквівалентними визначення:

```
а) A DB 5
   B DB 10
   C DB 20
б) A DB 5,20,10
```

13. Маємо визначення:

```
XMAS DW 8,-3,14,-7,0,5
```

Запишіть адресу кожної з цих констант окремо.

14. Маємо визначення:

```
TEST_T DB 5,?,?,-8,?,0,"Що це?",3,?,"Ой'ой'ой",0
```

Скільки загалом пам'яті буде виокремлено? Яка адреса кожної константи та літералів?

15. Маємо визначення:

```
а) X1 DW 8 DUP (?)
б) X2 DW 12 DUP (3,?)
в) X3 DW 20 DUP (0,0,0)
```

Що вони задають? Скільки пам'яті виокремлено?

16. Маємо визначення:

```
а) M1 DB 5 DUP (8)
б) M2 DW 5 DUP (8,8)
в) M3 DW 5 DUP (8,?,8)
```

Намалюйте схему розподілу пам'яті щодо кожного випадку.

17. Намалюйте схему розподілу пам'яті для таких визначень:

```
а) T_T_5_5 DB 3 DUP (?,?,1)
б) R_$$ DB 4 DUP ("MEMORY",0)
в) ABC DW 2 DUP (4 DUP (4))
г) BCD3 DW 3 DUP (3 DUP (2,?,0))
д) SIGMA DB 4 DUP (5, 2 DUP (-1,-2), 12)
```

18. Задайте пам'ять для двох матриць розміром **8x4** та **3x15** елементів. Кожен елемент займає слово.

19. Визначте та ініціалізуйте такі дані:

а) вектор однобайтових чисел з восьми елементів зі значеннями нуль;

б) два вектори з 12-ти слів кожен, зі значеннями: 6 нулів, 6 четвірок − перший; 8 одиниць, 3 п'ятірки, нуль − другий;

в) матрицю:

1 1 1 ... 1
2 2 2 ... 2
1 1 1 ... 1
2 2 2 ... 2

з 14-ти елементів у рядку.

Тема 2. Пересилання даних. Стек

Базові поняття*: копіювання даних під час пересилання; джерело, приймач; розмір даних (одно-, двобайтові); команда* **MOV***, порядок запису операндів; варіанти пересилання (пам'ять, регістри, безпосереднє); обмеження на команду* **MOV***; обмін даних* **XCHG***; ділянка пам'яті для стека; вершина стека; вказівник стека* **SP***; запис у стек, читання зі стека; команди* **PUSH**, **POP**, **PUSHF**, **POPF***; розмір стека.*

2.1. Теоретичні положення

2.1.1. Регістри загального призначення (**AX**, **BX**, **CX**, **DX**)

Загальні регістри використовують переважно з метою обчислень. Усі вони мають розмір 16 бітів, однак програма може працювати зі старшими чи молодшими 8-ма бітами кожного регістра окремо. Наприклад, регістр **AX** налічує з 16 бітів. Програма може звернутися до старших 8-ми бітів **AX** як до регістра **AH**, а молодші 8 бітів утворять регістр **AL**. Те ж саме стосується регістрів **BX**, **CX** і **DX**. Програма може розглядати цю групу регістрів як чотири 16-бітових, вісім 8-бітових або деяку комбінацію 8- і 16-бітових регістрів.

Головне призначення загальних регістрів - зберігати операнди. Загальні регістри зберігають як слово, так і байт даних. Однак ці регістри під час виконання певних операцій мають спеціальне призначення.

Регістр **AX** відповідає суматорові (акумуляторові). Безпосередні операції з регістрами **AX** і **AL** (16- і 8-бітовий суматори, відповідно) зазвичай вимагають значно коротшої команди, ніж аналогічні операції із залученням інших регістрів загального призначення. А менший розмір команди дає змогу отримати компактніші й швидкодіючіші програми.

Регістр **BX** є і регістром для обчислень, і адресним регістром. Використвуючи його як 16-бітовий, ним можна визначати адреси операнда. Способи (режими) адресування перелічено в окремому пункті.

Регістр **CX** використовують як лічильник до деяких команд. Ці команди використовують значення , що розташоване в **CX** як покажчик числа ітерацій команди або фрагмента програми.

Регістр **DX** слугує як розширення акумулятора для багаторозрядних операцій множення і ділення. У цих 32-бітових операціях беруть участь одночасно регістри **AX** і **DX**.

2.1.2. Сегментні регістри (CS, DS, SS, ES)

Процесор має чотири сегментних регістри: **CS**, **DS**, **SS** і **ES** - відповідно для кодового, даних, стекового і додаткового сегментів. Це їхнє звичайне використання, проте застосування цих регістрів може змінюватися відповідно до потреб програми.

Процесор використовує регістр сегмента програми (коду) для ідентифікації того сегмента, який містить програму, що виконується в даний момент. У поєднанні з покажчиком команд регістр **CS** використовують для вказівки поточної команди. Кожна команда, що виконується, перебуває в осередку, на який вказує пара регістрів **CS:IP**.

Комбінацію сегментного регістра з регістром зміщення для вказівки фізичної адреси записують у вигляді *сегмента*:*зміщення* (наприклад, **CS:IP**). Значення сегмента стоїть перед двокрапкою, зміщення - після. Таку нотацію використовують і для регістрів, і для абсолютних адрес. Можна записати так: **CS:100, DS:BX, 570:100**, або **630:DI**.

Регістр сегмента даних (**DS**) процесор використовує з метою звичайного доступу до даних. Схеми адресування для операндів дають 16-бітове зміщення, і переважно для формування виконавчої адреси процесор комбінує це зміщення з регістром **DS**.

Регістр сегмента стека вказує на системний стек. Команди **PUSH**, **POP, CALL і RET** керують даними у стеку за адресою **SS:SP**. Регістр **SP** - вказівник стека - слугує для визначення зміщення у стекові. Крім того, сегмент стека беруть до уваги за домовленістю під час адресування з використанням регістра **BP**. Це дає доступ до даних у стеку з використанням регістра **BP**.

Нарешті, регістр додаткового сегмента використовують з метою доступу до даних, коли потрібно більше одного сегмента. Звичайною операцією такого рівня є копіювання даних з однієї ділянки пам'яті в іншу. Між ділянками , розташованими поза одним і тим же блоком пам'яті розміром 64К, неможливо зробити обмін даними, використовуючи єдиний сегментний регістр. Володіючи додатковим сегментним регістром, програма може визначити одночасно початковий і цільовий сегменти.

2.1.3. Регістри адресування (BX, BP, SI, DI)

У процесорі є чотири 16-бітових регістри, які можуть брати участь в адресуванні операндів. Один з них є одночасно регістром загального призначення - регістр бази **BX**.

Інші три - це вказівник бази (*Base Poiner* - **BP**), індекс джерела (*Source Index* - **SI**) та індекс призначення (*Destination Index* - **DI**). Програма може використати регістри **BP**, **SI** і **DI** як 16-бітові операнди (і тільки), тобто їхні байти кожен зокрема (як, наприклад, **AL**) є недоступними. Головне призначення цих регістрів – постачати 16-бітові значення, які використовують з метою формування адрес операндів.

Кожна команда процесора задає для виконання деяку операцію. Різні операції налічують від нуля до двох операндів. Деякі команди неявно задають розташування операнда, проте більшість команд дозволяє програмістові обирати операндом регістр або елемент пам'яті. Якщо операндом вибрано регістр, то програмістові залишається тільки вказати його ім'я. Для визначення операндом ділянки пам'яті існують різноманітні *режими адресування.*

2.1.4. Режими адресування

Режими адресування	*Формат операнда*	*Регістр сегмента*
Регістровий	Регістр	Не використовується
Безпосередній	Число	Не використовується
Прямий	Позначка (ім'я) зміщення	**CS** **DS**
Неявний регістровий	**[BX]** **[BP]** **[DI]** **[SI]**	**DS** **SS** **DS** **DS**
Базовий	**[BX] + зміщення** **[BP] + зміщення**	**DS** **SS**
Індексний	**[SI] + зміщення** **[DI] + зміщення**	**DS** **DS**
Базовий з індексуванням	**[BX][SI]+ зміщення** **[BX][DI] + зміщення** **[BP][SI] + зміщення** **[BP][DI] + зміщення**	**DS** **DS** **SS** **SS**

Зауваження:

1. Зміщення під час базового адресування з індексуванням – трапляється не завжди.
2. Операнд <*регістр*> - будь-який 8- або 16-бітовий регістр, крім **IP**.
3. Операнд <*число*> - 8- або 16-бітове значення константи.
4. Операнд <*зміщення*> - 8- або 16-бітове значенням зі знаком.

Під час **регістрового адресування** мікропроцесор обирає операнд з регістра або завантажує (записує) операнд у регістр, наприклад:

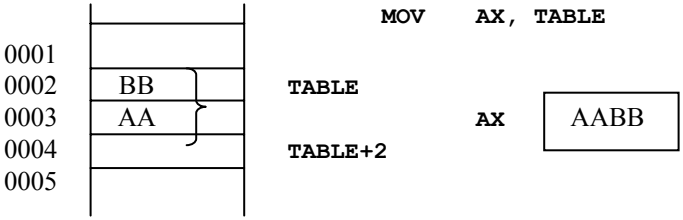
```
MOV  AX, CX
```

Безпосереднє адресування дає змогу вказувати значення константи як операнда джерела, наприклад:

```
MOV  CX, 500

K    EQV    1024

MOV  AL, -30

MOV  CX, K
```

Пряме адресування найчастіше застосовують за умови, коли операндом є окрема комірка пам'яті (змінна чи константа), наприклад:

Сегмент даних:



За умови **неявного регістрового адресування** адреса операнда міститься в одному з регістрів **BX, BP, SI, DI.** Наприклад:

```
MOV BX, OFFSET TABLE     ; оператор OFFSET вираз повертає
                         ; значення зміщення адреси виразу
```

Після цього за допомогою команди:

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

досягаємо того ж результату, що і за умови прямого адресування,

тобто **AX** AABB .

За умови **базового режиму** адреса є сумою вмісту регістра **BX** або **BP** та зміщення. Зручно використовувати з метою доступу до структурованих даних, розташованих у різних ділянках пам'яті.

За умови **індексного режиму** адреса є сумою вмісту регістра **SI** або **DI** та зміщення. Зручно використовувати з метою доступу до елементів таблиці чи вектора.

Базовий з індексуванням режим адресування зручно використовувати для двовимірних масивів: *базовий регістр* – початкова адреса, *індексний регістр* – зміщення до початку відповідного рядка, *зміщення* – зміщення від початку рядка до потрібного елемента. Адреса є сумою вмісту базового регістра, індексного регістра і , можливо, зміщення. Зручно використовувати також з метою адресування масиву, розташованого у стекові.

Три останні з перелічених режимів адресування (базовий, індексний, базовий з індексуванням) детально розглянемо під час висвітлення тем роботи з масивами і структурами.

2.1.5. Команда пересилання

Команда **MOV** - головна команда пересилання даних, яка пересилає байт або слово даних з пам'яті до регістра, з регістра – у пам'ять, або з регістра до регістра. Команда **MOV** може також внести число, визначене програмістом, до регістра чи у пам'ять.

Існує сім різних варіантів команди **MOV**, однак програміст використовує кожну з цих команд за допомогою єдиної назви операції **MOV**. Асемблер породжує правильну машинну команду, аналізуючи типи операндів, які написав програміст; і це одна з причин, за якою асемблер вимагає для операндів призначення типів. Тобто необхідно вказати, чим саме є кожен операнд - регістром, байтом пам'яті, словом пам'яті, сегментним регістром тощо.

Переважно шлях пересилання - з пам'яті до регістрів, і навпаки. З даними, розташованими у регістрах, можна працювати з більшою ефективністю, ніж з даними у пам'яті, оскільки мікропроцесор не звертається до пам'яті щоразу, коли потрібні дані.

У самій команді **MOV** може міститися нове значення як число. Таку форму операнда називають безпосереднім операндом; дані перебувають у самій команді і не вимагають обчислення адреси. Команда може переслати безпосередні дані у регістр чи елемент пам'яті.

Нарешті, команда **MOV** може записати сегментний регістр у пам'ять чи регістр, або ж завантажити сегментний регістр з пам'яті чи з

іншого регістра. Однак не існує команди завантаження сегментного регістра даними з безпосереднім операндом. Якщо за програмою необхідно розташувати відоме значення у сегментний регістр, треба спочатку записати це значення в один з регістрів або в елемент пам'яті, а потім вже пересилати це значення у сегментний регістр.

Приклади пересилання даних:

```
CODE    SEGMENT
ASSUME  CS:CODE, DS:CODE
EXWORD  LABEL   WORD
EXBYTE  LABEL   BYTE
MOV     AX, BX        ; Регістр BX --> Регістр AX
MOV     BX, AX        ; Регістр AX --> Регістр BX
MOV    CX, EXWORD     ;  Пам'ять --> Регістр
MOV     EXWORD, DX    ;  Регістр --> Пам'ять
MOV     CH, EXBYTE    ;  Пам'ять --> Регістр (байт)
MOV     EXBYTE, DH    ;  Регістр --> Пам'ять (байт)
MOV     SI, 1000      ;  Безпосереднє --> Регістр
MOV    BL, 23         ;  Безпосереднє --> Регістр (байт)
MOV    EXWORD, 2000   ;  Безпосереднє --> Пам'ять
MOV     EXBYTE, 46    ;  Безпосереднє --> Пам'ять (байт)
MOV     AX, EXWORD    ;  Пам'ять --> Акумулятор
MOV     AL, EXBYTE    ;  Пам'ять --> Акумулятор (байт)
MOV     EXWORD, AX    ;  Акумулятор --> Пам'ять
MOV     EXBYTE, AL    ;  Акумулятор --> Пам'ять (байт)
MOV     DS, EXWORD    ;  Пам'ять --> Сегментний регістр
MOV     DS, AX        ;  Регістр --> Сегментний регістр
MOV     EXWORD, DS    ;  Сегментний регістр --> Пам'ять
MOV     AX, ES        ;  Сегментний регістр --> Регістр
CODE    ENDS
```

Команда **MOV** має два операнди: джерело та результат. У команді вони розташовані один за одним: джерело слідує за результатом. Команда **MOV** не змінює джерело, тобто команда

```
MOV  AX,  BX
```

змінює регістр **AX** – результат, однак не змінює регістр **BX** – джерело. Жодна з команд **MOV** не змінює ознак стану.

2.1.6. Команда обміну

Команда обміну **XCHG** замінює місцями значення двох осередків. Ця команда може замінити місцями вміст двох регістрів, або регістра і пам'яті. У цьому випадку операндами не використовують сегментні сегментні регістри. Команда **XCHG** замінює три команди пересилання і не вимагає проміжного елемента пам'яті. Якби команда обміну не

існувала, програмі необхідно було б здійснити три пересилання, щоб обміняти значення в регістрі **AX** і в регістрі **BX**.

Спочатку вона повинна була б переслати вміст регістра **AX** в робочий осередок, потім переслати вміст регістра **BX** в регістр **AX**, і, нарешті, переслати вміст робочого осередку в регістр **BX**. Команда **XCHG** одна виконує цю операцію.

Приклади команд обміну:

```
EXDWORD DD   DWORD
EXWORD  DW   WORD
EXBYTE  DB   BYTE
        …
XCHG    BX,  CX          ;  Регістр BX <--> Регістр CX
XCHG    BX,  EXWORD      ;  Регістр BX <--> Пам'ять
XCHG    AX,  BX          ;  Регістр AX <--> Регістр BX
```

2.1.7. Операції зі стеком

Мікропроцесор адресує стек за допомогою регістрової пари **SS:SP**. Переміщення об'єктів у стек спричинює його зростання у бік менших адрес пам'яті. Стек, крім всього іншого, слугує і для запам'ятовування адрес повернення з підпрограм. У цьому розділі розглянемо деякі команди, які безпосередньо працюють зі стеком.

Мнемоніка команд очевидна; за кодами операцій **PUSH** і **POP** записується назва регістра для вказівки операнда. Єдиним винятком є переміщення і читання зі стека регістра ознак, які використовують мнемоніку **PUSHF** і **POPF** відповідно. Вміст будь-якого елемента пам'яті, який програма може адресувати, використовуючи можливі способи адресування, можна розташувати або прочитати зі стека.

При будь-яких діях зі стеком у мікропроцесорі базовою одиницею інформації є слово. Довжина будь-якого об'єкта, який уміщується у стек або читається з нього, становить одне або декілька слів. Байтових команд, спричинених засиланням даних або витяганням їх зі стека, не існує. Якщо, наприклад, програмі необхідно зберегти вміст регістра **AL** у стекові, вона повинна умістити вміст регістра **AX**, оскільки не існує способу збереження регістра як однобайтового елемента.

Основне призначення стека - тимчасове зберігання інформації. Якщо програма хоче використати регістр (наприклад, зберегти поточні дані), вона може надіслати значення цього регістра до стека, де вони зберігатимуться. Згодом ці дані можна відновити.

Приклади:

```
CODE    SEGMENT
ASSUME  CS:CODE, DS:CODE
EXWORD  LABEL   WORD
PUSH    AX  ; Записати регістр до стека
PUSH    SI
PUSH    CS    ; Можна записати до стека сегментний регістр
PUSH   EXWORD ; Можна записати до стека елемент пам'яті
POP    EXWORD ; Можна прочитати у пам’ять
POP    ES     ; Можна прочитати в інше місце
POP      DI
POP      BX
PUSHF         ;  Інша мнемоніка для ознак
POPF
```

Зазначимо, що стек - це структура типу **LIFO**. Якщо у вашій програмі виконується послідовність команд:

```
PUSH  BX
PUSH CX
POP BX
POP CX
```

то результуючим ефектом буде обмін значень у регістрах **BX** і **CX**.

Однак те, що в команді **PUSH** зазначено регістр **BX**, не означає, що команда **POP**, яка вказує на той же регістр, відновлює первинний вміст регістра **BX**. Важливою є збалансованість команд **PUSH** і **POP**, тобто кожній команді **PUSH** повинна відповідати команда **POP**. Якщо записи і читання зі стека не збалансовані, результати будуть невірними. Окрім того, незбалансовані команди **PUSH**/**POP** зазвичай зумовлюють повернення з підпрограм за адресою значення даних, а не значення вказівника команд через те, що мікропроцесор записує до стека адресу повернення.

Нарівні із збереженням даних, програма може використати стек як буфер під час деяких пересилань; зокрема, не існує команди пересилання, яка б переносила дані з одного сегментного регістра до іншого. У звичайному випадку завантаження одного сегментного регістра з іншого вимагає спочатку завантаження його значення у проміжний регістр. Цього досягають наступною послідовністю команд:

```
MOV AX, CS ; переслати значення регістра CS до регістра AX
MOV DS, AX ; завантажити це значення у регістр DS
```

Кожна з цих команд має довжину декілька байт, і ця послідовність руйнує вміст регістра **AX**. Існує альтернативний підхід:

```
PUSH  CS     ;  регістр CS вмістити у стек
POP   DS     ;  вмістити це значення у регістр DS
```

Тут довжина команд - всього два байти, до того ж проміжний регістр не потрібний. Однак ці дві команди займають більше часу, оскільки потрібні додаткові цикли читання і запису до стека.

2.2. Завдання для самостійної роботи

1. Значення з регістра **DX** переписати у два інших регістри.
2. Однобайтову змінну величину **DELTA** переписати у два різних регістри.
3. Слово у пам'яті **SG1** переписати у два інших слова: **SG2** і **SG3**.
4. Число 82 переписати у регістри **AX**, **BX**, **CX**, **DX**.
5. Числа 4, -6, 8, -10 переписати у чотири різні однобайтові змінні.
6. Нехай визначено змінну величину: **RRA DB ?**. Переписати її значення у регістри **AL**, **CX**.
7. Замінити значення у регістрах **SI**, **DI**, не використовуючи **XCHG**.
8. Замінити значення двобайтових змінних **A11**, **A22**:
   а) не використовуючи команду **XCHG**;
   б) використовуючи команду **XCHG**.
9. Замінити значення змінної **TER1** і регістра **SI**, а також змінної **TER2** і регістра **DX**:
   а) не використовуючи команду **XCHG**;
   б) використовуючи команду **XCHG.**
10. Ініціювати нулями усі регістри даних.
11. Ініціювати значеннями -1 змінні величини:
   - **DN1**, **DN2**, **DN3** – однобайтові;
   - **DD1**, **DD2**, **DD3**, **DD4** - двобайтові.
12. Які команди записані правильно, а які - ні, і чому?

```
            ART DB -2
            BCBC DB 0,1,?
            X1 DW ?
            X2 DW ?,3,10
а) MOV AL,ART          д) MOV X1,X2
б) MOV BX,ART          е) MOV BCBC,SI
в) MOV CX,CX           є) XCHG DI,X2+4
г) MOV -5,X1           ж) XCHG AL,X1
ґ) MOVE X2,0           з) XCHG ART,2
            и) MOV AH,BCBC+5
            і) XCHG X2-3,AX
            ї) MOV ART,-1
            й) MOV ART+3,'+'
```

13. Які значення матимуть регістри **AX**, **BX** після виконання таких команд:

```
CHIS DB 2
XXX_1 DW 4

MOV AL,1
MOV BX,XXX_1
XCHG AX,BX
XCHG CX,XXX_1
MOV CX,BX
XCHG AX,XXX_1
MOV BL,CHIS
MOV BH,1
```

14. Перерахуйте ситуації, в яких можна використовувати команду **MOV.**

---

15. Замініть значення регістрів **AX**, **DX**, використовуючи лише команди **PUSH**, **POP**.
16. Запишіть у стек значення двох двобайтових величин **VEL1**, **VEL2**.
17. Запишіть у стек значення двох однобайтових величин **V_1B, V_2B**.
18. У вершині стека розташовані значення трьох величин. Перепишіть їх у різні регістри.
19. У вершині стека розташовані значення двох величин. Перепишіть їх у регістри, залишаючи при цьому в стекові.
20. У вершині стека є три величини. Вилучіть зі стека третю від вершини.
21. Замініть місцями дві верхні величини стека.
22. Значення регістра ознак перепишіть у два різних слова пам'яті.
23. Встановіть у регістрі ознак усі біти в нуль.
24. Які значення матимуть слова **SLOVO**, **DRUGE** після виконання таких команд:

```
MOV DRUGE,15H
MOV BX,33H
PUSH DRUGE
PUSH DRUGE
POP SLOVO
POP AX
XCHG BX,DRUGE
PUSH BX
POP CX
XCHG CX,SLOVO
```

ТЕМА 3. Арифметичні операції для двійкових даних

Базові поняття: *перетворення довжини операндів* **CBW**, **CWD***; числа зі знаком та без знаку, максимально допустимі значення; додавання чисел* **ADD**, **ADC**, **INC***; переповнення; перенос; віднімання чисел* **SUB**, **SBB**, **DEC***; множення чисел* **MUL**, **IMUL**, *довжина операндів та результату; ділення чисел* **DIV**, **IDIV**, *частка, остача, довжина операндів та результату, переривання типу* 0.

3.1. Теоретичні положення

3.1.1. Перетворення довжини операндів

Команда **CBW** розширює однобайтове арифметичне значення в регістрі **AL** до розмірів слова шляхом розмноження знакового біта (сьомого) в регістрі **AL** по всіх бітах регістра **AH**. Використовується для отримання 16-бітового діленого в **AX**.

Команда **CWD** розширює арифметичне значення в регістрі **AX** до розмірів подвійного слова у регістровій парі **DX:AX** шляхом розмноження знакового біта (15-го) регістра **AX** по всім бітам регістра **DX**. Використовується для отримання 32-бітового діленого.

Команди без операнда (за домовленістю **AL** або **AX**).

3.1.2. Додавання та віднімання чисел

Команда **ADD** виконує додавання операндів, зображених в двійковому коді. Мікропроцесор вміщує результат на місце першого операнда після того, як додасть обидва операнди. Другий операнд не змінюється. Команда коректує регістр ознак відповідно до результату додавання. Наприклад, команда

```
ADD   AX, BX
```

додає вміст регістра **BX** до вмісту регістра **AX**, і залишає результат в регістрі **AX**. Регістр ознак повідомляє про те, чи був результат нульовим, від'ємним, чи мав парність, перенесення або переповнення.

Абсолютно аналогічно працює команда віднімання (**SUB**), тільки від першого операнда **віднімається** другий і результат заноситься на місце першого операнда.

Існують дві форми додавання/віднімання: 8-бітове і 16-бітове. У різних формах додавання/віднімання беруть участь різні регістри. Асемблер стежить за тим, щоб операнди відповідали один одному. Вміст байтового регістра (наприклад, **CH**) не можна додавати до елемента пам'яті, який не має типу **BYTE**. Якщо елемент пам'яті є одним з операндів, то він може бути або операндом-результатом, або

незмінним операндом. Одним з операндів може також бути безпосереднє значення. Тобто можливі формати:

- (*регістр*) ± (*регістр або пам'ять*) → *регістр*;
- (*регістр або пам'ять*) ± (*регістр*) → *регістр або пам'ять*;
- (*регістр* **AX** *або* **AL**) ± (*безпосередній операнд*) → *регістр* **AX** *або* **AL**;
- (*регістр або пам'ять*) ± (*безпосередній операнд*) → *регістр або пам'ять*.

Команда додавання з перенесенням **ADC** – це та ж команда **ADD**, за винятком того, що до суми зачислено біт перенесення (**CF**), який додається до молодшого біта результату. Для будь-якої форми команди **ADD** існує подібна до неї команда **ADC**.

Команда **SBB** додає значення біта перенесення (**CF**) до другого операнда і результат віднімається від першого операнда. Різниця заноситься на місце першого операнда.

Команди додавання і віднімання з перенесенням використовують з метою додавання і віднімання багаторозрядних чисел. У цих випадках треба також використовувати явне задання типу за допомогою оператора **PTR**, що має формат:

тип **PTR** ***вираз***

Оператор вказує на те, що заданий ***вираз*** повинен трактуватися як вираз заданого ***типу***. Можна задавати типи **BYTE, WORD, DWORD, FWORD, TBYTE** для змінних і **NEAR**, **FAR** і **PROC** для позначок команд.

Приклад. Маємо дві чотирибайтові змінні величини (подвійні слова). Знайти їхню суму та різницю.

```
…
.data
x  DD ?
y  DD ?
suma  DD ?
rizn DD ?
Start
…
; Додавання молодших 16-ти розрядів (байти числа
; розташовані у пам'яті в оберненому порядку)
MOV AX, WORD PTR x
ADD AX,WORD PTR y
MOV WORD PTR suma, AX
; Додавання старших 16-ти розрядів
MOV AX, WORD PTR x+2
ADC  AX, WORD PTR y+2,  ;урахування переносу
MOV WORD PTR suma+2, AX
```

```
        …
; Віднімання молодшої частини
MOV  AX, WORD PTR x
SUB AX, WORD PTR y
MOV WORD PTR rizn, AX
; Віднімання старшої частини
MOV AX, WORD PTR x+2
SBB AX, WORD PTR y+2  ;урахування переносу
MOV WORD PTR rizn+2, AX
        …
```

Команда **NEG** змінює знак операнда – байта або слова – на протилежний. Команда збільшення **INC** додає 1 до операнда, а команда зменшення **DEC** віднімає 1 від операнда. **Операнд** ~ *регістр або пам'ять*.

За допомогою команд збільшення і зменшення можна переміщувати показник масивом елементів пам'яті. Ці команди також можуть реалізувати лічильник циклу. Кожний перехід циклом зменшує лічильник, а коли його значення досягне 0, то цикл припиняється.

3.1.3. Множення та ділення чисел

Існує дві команди множення. За командою **MUL** перемножуємо два цілих числа без знака і отримуємо результат без знака. За командою **IMUL** перемножуємо цілі числа зі знаком (причому цілі числа зображені у додатковому коді) і отримуємо результат, що має правильний знак і значення.

Обидві команди множення працюють як з байтами, так і зі словами. Однак діапазон форм зображення операндів набагато вужчий, ніж для команд додавання і віднімання.

Щоб помножити 8 бітів на 8 бітів, один з операндів повинен бути в регістрі **AL**, а другий **операнд** (*регістр або пам'ять*) задається в команді. Результат завжди виявляється в регістрі **AX**. Результат може мати довжину аж до 16-ти бітів (максимальне значення, яке отримують при беззнаковому множенні, дорівнює 255 * 255 = 65025).

Щоб перемножити 16 бітів на 16 бітів, один з операндів треба вмістити в регістр **AX,** а другий **операнд** (*регістр або пам'ять*) задається в команді. Результат, який може бути довжиною до 32-ти бітів (максимальне значення при беззнаковому множенні 65535 * 65535) вміщується в пару регістрів: у регістрі **DX** містяться старші 16 бітів результату, а в регістрі **AX** - молодші 16 бітів. Множення не допускає безпосереднього операнда.

Приклад. Помножити дві двобайтові величини зі знаком.

```
.data
x dw ?
y dw ?
Rob dd ?
Start
   …
mov AX, x
imul y
mov WORD PTR Rob, AX
mov WORD PTR Rob+2, DX
   …
```

Як і у випадку з множенням, існує дві форми ділення: одна – для двійкових чисел без знака **DIV**, друга – для чисел в додатковому коді **IDIV**. Будь-яка форма ділення може працювати з байтами і словами.

Команда ділення **DIV** виконує дію ділення без знака і визначає як частку, так і залишок. Операнди повинні перебувати на строго визначених місцях, а ділене має бути операндом подвійної довжини.

Байтові команди ділять 16-бітове ділене у регістрі **AX** на 8-бітовий дільник, який задають у команді (*регістр* або *пам'ять*). Внаслідок ділення отримують два числа. Ділення вміщує частку до регістра **AL**, а залишок – до регістра **AH**.

Команда, яка працює зі словами, ділить 32-бітове ділене на 16-бітовий дільник. Ділене перебуває у парі регістрів **DX**:**AX**, причому регістр **DX** містить старшу частину, а регістр **AX** – молодшу. Дільник задають у команді (*регістр* або *пам'ять*). Ділення слів вміщує частку у регістр **AX**, а залишок у регістр **DX**.

Жодну з ознак стану не визначено після команди ділення.

Однак під час ділення може виникнути помилка значущості. Якщо частка більша, ніж можна вмістити в регістр результату, мікропроцесор не дасть правильний результат (у разі ділення байтів частка повинна бути меншою 256, і менше 65535 у разі операції зі словами). Мікропроцесор не встановлює ніяких ознак для сигналізації цієї помилки, замість цього він виконує програмне переривання типу 0.

Як і у випадку інших програмних переривань, це переривання за діленням на 0 зберігає ознаки, регістр кодового сегмента і показник команди у стекові. Потім мікропроцесор передає керування в осередок, на який посилається вказівник за адресою 0. Підпрограма ділення на 0 повинна виконати відповідні дії щодо опрацювання цієї помилки.

Ділення цілих чисел зі знаком **IDIV** відрізняється від команди **DIV** тільки тим, що воно враховує знаки обох операндів. Якщо результат додатний – все відбувається так само, як описано у випадку з

командою **DIV** (за винятком того, що максимальне значення частки, відповідно, дорівнює 127 і 32767 для байтів і слів).

Якщо результат від'ємний, то залишок має такий самий знак, як і ділене. Мінімальні значення частки для від'ємних результатів -128 і -32768 для байтів і слів, відповідно.

Приклад. Поділити дві змінні двобайтові зі знаком.

```
.data
x dw ?
y dw ?
Chastka dw ?
Ostacha dw ?
Start
   …
mov AX, x
cwd
idiv y
mov Chastka, AX
mov Ostacha, DX
   …
```

3.2. Завдання для самостійної роботи

1. Додати три числа: 10, -25, 2.
2. Додати три числа: 85, 90, 99.
3. Додати чотири однобайтові змінні: **X1, R1, A1, D1.**
4. Кожну з чотирьох двобайтових змінних збільшити на 1.
5. Відняти попарно дві пари змінних: **RRR1-RRR2, TTT1-TTT2.**
6. Задано три однобайтові змінні. Зменшити їхні значення на 1 трьома різними способами.
7. Задано три чотирибайтові змінні величини. Знайти їхню суму.
8. Задано три чотирибайтові змінні величини. Від першої змінної відняти суму другої та третьої.
9. Кожну з трьох змінних величин **ZP, RP, MP** помножити на 3, вважаючи, що величини: а) однобайтові зі знаком; б) двобайтові зі знаком.
10. Перемножити 3 змінні однобайтові величини без знака **VEL1, VEL2, VEL3,** вважаючи, що: а) результат кожного множення менший від 255; б) результат першого або другого множення може бути більшим за 255.
11. Кожну з двох змінних величин зменшити у 5 разів, вважаючи, що: а) змінні однобайтові без знака; б) змінні двобайтові без знака; в) змінні однобайтові зі знаком; г) змінні двобайтові зі знаком.
12. Знайти суму часток та суму остач від ділення двох пар змінних величин зі знаком, довжина яких півслова.
13. Знайти середнє арифметичне значення остач від ділення трьох пар змінних величин без знака, довжина яких слово.

14. Виконати обчислення за такими формулами у двох варіантах для кожної: змінні величини однобайтові без знака; змінні величини однобайтові зі знаком. Вважати, що кожна окрема операція дає коректний результат.

1) $y = (a+b+1) / (c-2)$
2) $y = b - (c-2)/d + (5*d)^2$
3) $y = a^2 + b^2 / c^2$
4) $y = (x1+x2+x3)^2 - 15*c$
5) $y = (x-t) * (x-a) * (x-z)$
6) $a = (b1-b2) * (2*d1-3*d2)$
7) $t = (x+y+z) / x^3$
8) $a = (m*p - y)/(b^2+2)$
9) $t = 101/x^3 + 101/y^3 + 101/z^3$
10) $b = (a-d) / (m^2 - m - 1)$
11) $a = 1 + 3b - 5c + 7d$
12) $b = a1^2 + a2^3 + a3^4$
13) $a = (3*x + y^2) / (4*x - y^2)$
14) $c = a*b + b*d + c*f - 12$

15. Виконати обчислення за такими формулами у двох варіантах для кожної: змінні величини двобайтові без знака; змінні величини двобайтові зі знаком. Вважати, що кожна окрема операція дає коректний результат.

1) $a = 150 / (7 * x^2) - 190 / (11 * x^2)$
2) $a = (x^2 + y^2) / (x^2 - y^2)$
3) $m = (x1+1) * (x2+2) * (x3+3)$
4) $m = a/b + c/d^2 + e/f^3$
5) $c = 3 + 5*x + 7*x^2 + 9*x^3$
6) $c = 91/x + 320/x^2 + 1300/x^3 + 1400/x^4$
7) $t = (1+x^2) / (1+y^2) - (a+b+1) / (2*a-1)$
8) $t = (x*r-1) * (x*p-1) * (x*d-1) - (a+1) * (d+1)$

ТЕМА 4. Введення та виведення цілих чисел. Оформлення програм

Базові поняття: *зовнішні пристрої ПК; введення та виведення даних; макрокоманда – група команд; макро* **RWORD**, **WWORD**; *додаткові можливості виведення даних - макро* **WWORDLN**, **WLINEZ**, **LINEFEED**, **BEEP**; *загальна стуктура програми на асемблері –* **INCLUDE**, **PROGRAM**, .**DATA**,**START**, **RETURN**, **END**; *коментування програм*.

4.1. Теоретичні положення

4.1.1. Макрокоманди введення та виведення даних

Усі макрокоманди введення/виведення працюють зі стандартним пристроєм **CON**: (екран дисплея за замовчуванням). Можна призначити стандартний пристрій введення і стандартний пристрій виведення на файл, принтер (**PRN**:), комунікаційний канал (**AUX**:) тощо.

Під час введення числових даних використовують внутрішній буфер. У цей буфер читають рядок символів, у якому може перебувати декілька чисел. Ці числа за кожного виклику команди введення читаються почергово. Якщо буфер вичерпано, то із пристрою введення вводиться в буфер новий рядок із наступними значеннями. Роздільником між числами можуть бути такі символи: пропуск, табуляція (код 9), кома, ':' , ';'. Пропуски і табуляції перед числом ігноруються.

Внаслідок виходу числового значення за межі його діапазону або внаслідок некоректного запису числа після виконання макрокоманди встановлюється ознака переносу (**CF**), обнулюється ознака нуля (**ZF**) і при поверненні видається звуковий сигнал. Якщо виникає помилка під час введення одного з елементів масиву, то видається відповідне повідомлення, а сам елемент не змінюється.

Аналогічно цій, виникає ситуація, за якої при введенні числа необхідно задати порожній рядок. У цьому випадку також встановлюються ознаки переносу і нуля (але відсутній звуковий сигнал). Унаслідок введення в будь-якому випадку здійснюється тестування значення введеного числа на нуль і знак (визначаються відповідно до результату ознаки **ZF** та **SF**).

Формати чисел, які вводять, аналогічні відповідним форматам чисел у мові асемблера. Існує лише одна відмінність, яка стосується цілих чисел у десятковій системі числення. Якщо число має наприкінці символ **'D'**, то число сприймається як беззнакове ціле (нижня границя 0). Якщо цей символ відсутній, то число знакове (нижня границя: -32768).

Введення *цілих 16- чи 8-бітових чисел* реалізує макрокоманда:

```
RWORD
RWORD регістр
RWORD пам'ять[,лічильник]
```

Перший аргумент – місце, куди неохідно занести прочитане значення. Розмір числа визначається автоматично за розміром першого аргумента. За домовленістю (перший варіант макрокоманди) число читається в **AX**.

Другий аргумент використовується лише тоді, коли перший є полем пам'яті, і зазначає, що необхідно прочитати декілька значень підряд (масив даних). Цей аргумент повинен бути 16-бітовим і мати формат правого аргумента команди **MOV**. За домовленістю він дорівнює 1.

Введення *цілих 32-розрядних чисел* реалізує макрокоманда:

```
RDWORD регістр
RDWORD пам'ять[,лічильник]
```

Перший аргумент – місце, куди необхідно занести прочитане значення. Якщо аргументом вказано ***регістр***, то з нього беруть адресу операнда. Решта – аналогічно до макрокоманди **RWORD.**

Введення з клавіатури функціонує так:

- усі символи заносяться почергово у пам'ять до натиснення клавіші *'Enter'*;
- якщо довжина рядка досягне значення обмежувача, то зі спробою ввести будь-який символ пролунає звуковий сигнал ('*Enter*' діє так само);
- у будь-якому випадку можна затерти останні введені символи клавішою '<--' (*'Backspace'*).

Виведення *цілих 16- або 8-розрядних чисел* реалізують макрокомандою:

```
WWORD[LN] [,,система]
WWORD[LN] регістр [,,система]
WWORD[LN] безпосереднє значення [,,система]
WWORD[LN] пам'ять [,[лічильник] [,система]]
```

За відсутності першого аргумента число виводиться із регістра **AX**. Аргументи розглядають аналогічно до макрокоманди **RWORD,** однак вони володіють такою властивістю: першим аргументом можна вказати безпосереднє значення (воно завжди сприйматиметься як 16-розрядне).

Усі числа виводяться у стандартному форматі із такими доповненнями:

- для знакових чисел відсутність попереднього знака у числі означає, що число додатнє;
- наприкінці кожного числа ставиться пропуск;
- жодна зі команд виведення чисел не здійснює переведення рядка, з метою переведення рядка до мнемоніки макрокоманди необхідно дописувати **'LN'** (наприклад: **WWORDLN** ...);
- для цілих чисел можна зазначати систему числення, яка дає змогу виводити числа у різних системах числення (формат аналогічний до введення, тобто: за домовленістю – десяткове знакове; символ **'H'** – 16-ве беззнакове; **'D'** – 10-ве беззнакове; **'O'** – 8-ве беззнакове; **'B'** – двійкове беззнакове;
- числом може бути будь-який 8-розрядний аргумент у форматі лівого аргумента команди **MOV**.

*Макрокоманда **виведення** рядка символів, який закінчується нулем*:

```
WLINEZ[LN] регістр
WLINEZ[LN] пам'ять [,обмежувач]
```

Окрім того, що більшість макрокоманд введення/виведення має власні засоби для переведення рядка, досить зручною є макрокоманда

```
LINEFEED
```

без аргументів. Вона спричиняє переведення (завершення) рядка.

Інколи треба попередити користувача програми щодо певних змін, які відбулися, або просто привернути його увагу до чогось. Поширеним засобом для цього є подача звукового сигналу. Макрокоманда

```
BEEP
```

не має аргументів і спричиняє подачу звукового сигналу.

4.1.2. Стандартне оформлення програм

Базовим шаблоном щодо програм має бути таке поєднання макрокоманд і псевдокоманд, які ми подамо далі. Звичайно, це не строгий шаблон. Наприклад, сегменти даних можуть чергуватися із сегментами кодів, а то й взагалі можуть бути відсутніми.

Обов'язковими у програмі за стандартом оформлення повинні бути команди **INCLUDE, PROGRAM, START, RETURN, END.** Існує ще й спрощений варіант програми, в якому **START** та **RETURN** не обов'яз-

кові – їх можна замінити на свій код ініціалізації та повернення із програми.

```
INCLUDE LIBMACRO.INC ;підключення бібліотеки
;макрокоманд
PROGRAM ім'я ;заголовок програми
.DATA? ;сегмент змінних
.............
;визначення змінних
.............
.DATA ;сегмент змінних із початковими значеннями
.............
;визначення змінних та їхніх початкових значень
.............
.CONST ;сегмент констант
.............
;визначення констант
.............
START ;точка старту програми
.............
;тіло програми
.............
RETURN ;точка повернення із програми
END ім'я ;кінець тексту програми
```

Псевдокоманда **COMMENT** дає змогу вводити в текст програми великі коментарі. Формат запису таких коментарів досить своєрідний:

```
COMMENT  обмежувач [[текст]]
[[текст]]
обмежувач [[текст]]
```

Перший символ, що стоїть після слова **COMMENT,** слугує обмежувачем тексту великого коментаря. Будь-який набір символів, що стоїть після обмежувача, ігнорується транслятором доти, доки знову не зустрінеться символ обмеження, причому немає ніяких обмежень на кількість рядків тексту коментаря. Текст у тому ж рядку, що і кінцевий обмежувач, завжди ігнорується, тобто транслятор починає транслювати рядки, які стоять після рядка з кінцевим обмежувачем.

4.1.3. Приклади лінійних програм

Приклад 1

```
INCLUDE LIBMACRO.INC
Program  ADD
COMMENT  \ Введення двох чисел та обчислення їхньої суми. Аргументи та результати виводяться у десятковому та 16-ому вигляді.
\
.const
 pidc db ' Enter number'
Start
  ; Вводимо 1- й доданок
   WLineln  pidc
   RWord
   ; Вводимо 2- й доданок
   WLineln pidc
   RWord BX
  ; Виводимо 1- й доданок у 2-х форматах
   WWord
   WWordLn  ,,H
   ; Реалізація додавання
   ADD     AX,BX
  ; Виводимо 2- й доданок у 2-х форматах
   WWord  BX
   WWordLn  BX,,H
  ; Виводимо результат у 2-х форматах
   WWord
   WWordLn  ,,H
Return
END  ADD
```

Приклад 2

```
INCLUDE LIBMACRO.INC
Program   eval1
; Обчислення y=3x^2+2x-5
.const
 pidc db ' Ввести X ',0
 thr dw 3
 two dw 2
 res db ' y= '
.data?
 rob dw ?
Start
; Вводимо x
  WLineZ pidc
  RWord
  mov bx,ax
  mov rob,ax  ; збереження x
  imul bx
  imul thr
; міняємо ax <- 3x^2 і bx<- x
  xchg ax,rob
  imul two
  add ax,rob
  sub ax,5
 ; Виводимо результат з регістра ax
  WlineЯ res
  WWordLN
Return
END  eval1
```

Приклад 3

```
INCLUDE LIBMACRO.INC
Program  Mult1
; Множення великих чисел
.const
 px  db ' Введіть x ',0
 py  db ' Введіть y ',0
 res db ' Mult= ',0
.data?
 x    DW ?
 y    DW ?
 rob  DD ?
```

```
; Продовження-1 прикладу 3
Start
  WLineZ px
  RWord x
  WLineZ py
  RWord y
  mov AX, x
  imul y
  mov WORD PTR Rob, AX
  mov WORD PTR Rob+2, DX
  WLineZ  res
  WDWordLn rob
Return
END  Mult1
```

Приклад 4

```
INCLUDE LIBMACRO.INC
Program  ASDW
; Додавання та віднімання подвійних слів
.const
 px db ' Введіть x ',0
 py db ' Введіть y ',0
 sres  db  ' Sum= ',0
 rres  db  ' Rizn= ',0
.data?
 x  DD ?
 y  DD ?
 suma  DD ?
 rizn DD ?
Start
  WLineZ px
  RDWord x
  WLineZ py
  RDWord y
; Додавання молодших 16 розрядів
  MOV AX, WORD PTR x
  ADD AX,WORD PTR y
  MOV WORD PTR suma, AX
; Додавання старших 16 розрядів
  MOV AX, WORD PTR x+2
  ADC  AX, WORD PTR y+2
  MOV WORD PTR suma+2, AX
; Віднімання молодшої частини
  MOV  AX, WORD PTR x
  SUB AX, WORD PTR y
  MOV WORD PTR rizn, AX
; Віднімання старшої частини
  MOV AX, WORD PTR x+2
  SBB AX, WORD PTR y+2
  MOV WORD PTR rizn+2, AX
; Виведення результатів
  WLineZ  sres
  WDWordLn suma
  WLineZ  rres
  WDWordLn rizn
Return
END  ASDW
```

Приклад 5

```
INCLUDE LIBMACRO.INC
Program  Div1
; Отримання частки та остачі
.const
 px db ' Введіть x ',0
 py db ' Введіть y ',0
 Chres db '  Chastka= ',0
 Osres db ' Ostacha= ',0
.data?
 x  DW ?
 y  DW ?
 Chast dw ?
 Ostacha dw ?
Start
  WLineZ px
  RWord x
  WLineZ py
  RWord y
  mov AX, x
  cwd
  idiv y
  mov Chast, ax
  mov Ostacha, DX
  WLineZ  Chres
  WWordLn chast
  WLineZ Osres
  WWordLn ostacha
Return
END  Div1
```

4.2. Завдання для самостійної роботи

1. Прочитати 3 числа і надрукувати їх у зворотному порядку.
2. Прочитати 6 чисел і надрукувати їх:

а) усі в одному рядку;

б) по три у рядку.

3. Прочитати два числа і надрукувати їх двічі – спочатку обидва в одному рядку, потім - обидва в стовпець.
4. Прочитати чотири числа і надрукувати їхнє середнє арифметичне. Вивести на екран відповідні допоміжні повідомлення макрокомандою **WLINEZ**.
5. Прочитати два числа та обчислити їхню суму, різницю, добуток, частку. Надрукувати результати з відповідними повідомленнями.
6. Прочитати три числа, обчислити їхню суму, суму квадратів, суму кубів. Надрукувати результати з відповідними повідомленнями.
7. Скласти повну програму для виконання обчислень за вказаними формулами, відповідно до загальної структури програми. Друкувати усі повідомлення перед введенням необхідних змінних величин та перед друкуванням результатів. Записати початкові коментарі щодо призначення програми. Прокоментувати важливі групи команд і деякі окремі команди.

1. $y = (a^2 + b^2 + c) / (a - 4*b - c)$.
2. $y = (a^2 + 2*b^2 + 3*c^2) / (a + b + c)$.
3. $y = (x1^2 - 2*x2^2) (1 + 3*x3^2 - 4*x4^2)$.
4. $y = 1024/(a+b) + 2048/(a-b) + 3000/(a^2-b)$.
5. $a = (b^3 - c^3/2 + d^3/4)/(d - 4*b - c)$.
6. $t = (2*b^2 + 4*d^2/c)/(d + 3*b - c)$.
7. $a = (m - 2*n - 4*p) / (4*p - 2*n - p)$.
8. $t = (51*d + 50*r + 49*s)/(s^2 + r^3)$.
9. $a1 = (m^2 - n^2)/(2*m^2 - 3)$.
10. $a2 = (m + 4*n)/(m^2 + n^2)$.
11. $a3 = (m-n)*(2*m^2-3)/(n^2+3*m)$.
12. $a1 = (4*x^2 + y^3 - 6) / (y + x^2)$.
13. $a2 = (12 + 2*x^2 + y) / (3 + x^2)$.
14. $a3 = (x^3 + 11) / (y^2 + 5) + 6*x^2$.
15. $t = (21*d + 5*r^2 + 49*s)/(s^2 + r^3)$.
16. $a = (m + n^3)*(2*m^2 + 3)/(n^2 + 3*m)$.

ТЕМА 5. Команди умовного та безумовного передавання керування

Базові поняття: *порядок виконання команд програми; команда безумовного переходу* **JMP**; *ознаки результату (регістр ознак); команда порівняння* **CMP**; *умовні переходи для чисел без знака та аналогічних даних* - **JZ**, **JNZ**, **JA**, **JAE**, **JB**, **JBE**; *умовні переходи для чисел зі знаком* - **JZ**, **JNZ**, **JG**, **JGE, JL, JLE;** *умовні переходи для окремих ознак* - **JC, JNC, JS, JNS, JO, JNO, JP, JNP;** *спеціальний умовний перехід* **JCXZ**.

5.1. Теоретичні положення

5.1.1. Команда безумовного переходу

Формат:
```
JMP регістр
JMP пам'ять
```

Команда виконує безумовний перехід за адресою, заданою єдиним операндом. Здебільшого операнд задається як позначка переходу, яку трактують як місце у пам'яті, куди здійснюється перехід. Якщо операндом задано регістр або змінну у пам'яті, то перехід здійснюється за адресою, записаною у цьому регістрі чи змінній (відносний перехід). Розрізняють 3 типи безумовного переходу: далекий (**FAR**), близький (**NEAR**), короткий (**SHORT**). Для далекого переходу адреса складається із двох слів (**сегмент:зміщення**), яку не можна задавати через регістр.

Близький перехід виконується лише в межах поточного кодового сегмента, тому операнд команди - одне слово, яке можна задавати і в регістрі. Короткий перехід здійснюється у межах від -128 до +127 байт щодо адреси команди **JMP**. Такий перехід виконується лише на позначку.

Зазвичай розмірність і тип адреси визначається автоматично (залежно від того, здійснюється перехід у межах поточного кодового сегмента, чи за його межі), однак інколи потрібно чітко зазначити певний тип переходу. Для чіткішого визначення переходу використовують оператор **PTR**. Наприклад:

```
JMP far ptr m2
```

5.1.2. Регістр ознак (**EFLAGS**)

- **CF** - перенесення (**Carry**) - встановлено в 1, коли відбулось перенесення зі старшого розряду при арифметичних операціях чи операціях зсуву;

- **PF** - парність (**Parity**) - встановлено в 1, коли молодші розряди (8 бітів) результату операції містять парне число розрядів, що дорівнюють одиниці;
- **ZF** - нуль (**Zero**) - встановлено в 1, якщо результат операції дорівнює нулю;
- **SF** - знак (**Sign**) дорівнює старшому розрядові результату (0 – додатний, 1 - від'ємний);
- **OF** - переповнення (**Overflow**) - встановлено в 1, якщо результат операції надто великий або надто малий для приймача (не поміщаються значущі розряди).

5.1.3. Команда порівняння **CMP**

Команда порівняння **CMP** порівнює два числа, віднімаючи друге від першого. Вона не записує результат, однак ознаки стану встановлює відповідно до результату. Ця команда змінює тільки ознаки.

5.1.4. Команди переходів за умовою

Ці команди мають формат

Jссс позначка

і перевіряють ознаки стану, встановлені попередніми командами. Ознаки кодуються у мнемоніці команди (**N** від *not* - заперечення):

JZ - перехід на позначку, якщо **ZF**=1;

JNZ - перехід на позначку, якщо **ZF**=0.

Якщо умова не виконується, то відбувається перехід до наступної команди. Аналогічно для команд **JC, JNC, JS, JNS, JO, JNO, JP, JNP.** Скорочення:

L (*less* - менше) і **G** (*greater* - більше) використовують з метою порівняння цілих зі знаком;

A (*above* –над) і **B** (*belov* –під) використовують з метою порівняння цілих без знака;

E (*equal* – рівне).

Приклади.

- Команда **JAE позначка** здійснює короткий перехід на ***позначку***, якщо виконується умова беззнакове "більше або дорівнює" (ознаки результату **ZF**=1, **CF**=0). Якщо умова не виконується, то відбувається перехід до наступної команди.
- Команда **JNGE позначка** здійснює короткий перехід на ***позначку***, якщо виконується умова "не більше або дорівнює" (ознаки результату **ZF**=0, **SF<>OF**). Якщо умова не виконується, то відбувається перехід до наступної команди. Аналог команди – **JL.**

5.1.5. Спеціальний умовний перехід **JCXZ**

Команда **JCXZ позначка** здійснює короткий перехід на ***позначку***, якщо виконується умова: *значення регістра* **CX=**0. Якщо умова не виконується, то відбувається перехід до наступної команди.

Цю команду застосовують на початку *циклу за умовою* з метою заборони входження у цикл , якщо значення регістра **CX=**0.

5.1.6. Приклади програм на галуження

Приклад 1. Обчислити

$$y=\begin{cases} x-5, \text{ якщо } x<1; \\ x+2, \text{ якщо } x\geq 1. \end{cases}$$

```
INCLUDE LIBMACRO.INC
Program  gal1
.const
 pidc db ' Введіть x ',0
 res db ' y= ',0
Start
   WLinez pidc
   RWord
   cmp ax,1
   jl  mitka
   add ax,2
   jmp vyhid
 mitka:
   sub ax,5
 vyhid:
   WLineZ res
   WWord
   LineFeed
Return
END  gal1
```

Приклад 2. Обчислити

$$y=\begin{cases} a-c, \text{ якщо } x=0; \\ a+c, \text{ якщо } x=4; \\ a\cdot c, \text{ якщо } x\neq 0 \text{ і } x\neq 4. \end{cases}$$

```
INCLUDE LIBMACRO.INC
Program  gal2
.const
 pa db ' Введіть a ',0
```

; Продовження прикладу 2:

```
 pc db ' Введіть c ',0
 px db ' Введіть x ',0
 res  db  ' y= ',0
.data?
 ra dw ?
 rc dw ?
Start
   WLineZ pa
   RWord
   mov ra,ax
   WLineZ pc
   RWord
   mov rc,ax
   WLineZ px
   RWord
   jz m1
   cmp ax,4
   je m2
   mov ax,ra
   imul rc
   jmp vyved
 m1: mov ax,ra
   sub ax,rc
   jmp vyved
 m2: mov ax,ra
   add ax,rc
 vyved:  WLineZ  res
   WWordLn
Return
END  gal2
```

5.2. Завдання для самостійної роботи

1. Перейти на позначку **RAB2**, якщо сума чисел без знака в регістрах **AL**, **BL** більша, ніж 10, або продовжити далі за текстом програми у протилежному випадку.
2. Перейти: на позначку **BC**, якщо число зі знаком у регістрі **DH** більше 1; на позначку **CD** - якщо дорівнює 1; продовжити далі - якщо менше 1.
3. Якщо значення змінної **ZMIN** належить інтервалові [-3; 15], то перейти на позначку **W1**, інакше - перейти на позначку **W2**.
4. Якщо значення змінних величин зі знаком **ZM1**, **ZM2**, **ZM3** є однаковими, то перейти на позначку **FB1**, інакше продовжити далі за текстом.
5. Якщо сума чисел без знака у регістрах **CX**, **DX** більша від значення змінної **RKRK**, то перейти на позначку **ABAB**, інакше - продовжити далі за текстом.
6. Перейти на такі позначки залежно від значення регістра **BH**:

A1 - якщо < -10
A2 - якщо = -10
A3 - якщо = -9
A4 - якщо > -9

7. Перейти на такі позначки залежно від значення суми двох змінних зі знаком **PROGA**, **PROGB**:

FIX - якщо < 0
DUBLI - якщо = 0
MA1 - якщо =1 або =2
MA2 - якщо > 2

8. Ввести з клавіатури три числа. Якщо їхній добуток не перевищує числа 525, то надрукувати його, інакше надрукувати квадрати чисел.
9. Ввести з клавіатури три числа. Якщо усі вони мають однакові знаки, надрукувати їхню суму, інакше - надрукувати середнє арифметичне.
10. Виконати обчислення за даними формулами. Довжину змінних (півслово, слово) та їхній тип (зі знаком, без знака) вибрати відповідно до формули або довільно, якщо це можна. Позначити коментарями окремі групи команд і команди перевірки умов.

1) $y = \begin{cases} a + b - c, \text{ якщо } f < 0; \\ a - b + c, \text{ якщо } f \geq 0. \end{cases}$

2) $y = \begin{cases} e^2 + c / d - a, \text{ якщо } a = 2; \\ e^3 - c + a, \text{ якщо } a \neq 2. \end{cases}$

3) $y = \begin{cases} a \cdot b + c, \text{ якщо } x = 0; \\ a + b \cdot c, \text{ якщо } x = 2; \\ (a + b) \cdot c, \text{ якщо } x \neq 0 \text{ і } x \neq 2. \end{cases}$

4) $y = \begin{cases} \dfrac{a}{2} - \dfrac{b}{3}, \text{ якщо } x < -3; \\ \dfrac{a}{3} - \dfrac{b}{2}, \text{ якщо } -3 \le x \le 0; \\ (a - b)/2, \text{ якщо } x > 0. \end{cases}$

5) $y = \begin{cases} a \cdot b + b \cdot c + c \cdot d, \text{ якщо } x = 3; \\ (a + c) \cdot b, \text{ якщо } x = 4; \\ (a - b) \cdot c, \text{ якщо } x \neq 3 \text{ і } x \neq 4. \end{cases}$

6) $z = \begin{cases} a \cdot b + b \cdot c - c, \text{ якщо } x = 6; \\ (a - c) \cdot b, \text{ якщо } x = 2; \\ a \cdot c, \text{ якщо } x \neq 6 \text{ і } x \neq 2. \end{cases}$

7) $y = \begin{cases} \dfrac{a}{3} - \dfrac{b}{4}, \text{ якщо } x < -1; \\ \dfrac{a}{4} + \dfrac{b}{3}, \text{ якщо } -1 \le x \le 2; \\ (a + b)/3, \text{ якщо } x > 2. \end{cases}$

8) $y = \begin{cases} \dfrac{a}{2} + \dfrac{c}{4}, \text{ якщо } x < -4; \\ \dfrac{a}{4} - \dfrac{b}{3}, \text{ якщо } -4 \le x \le 0; \\ (a - b)/2, \text{ якщо } x > 0. \end{cases}$

ТЕМА 6. Команди циклу. Побудова циклів для неіндексованих даних

Базові поняття: *загальна схема циклу з післяумовою; лічильник циклів; команда* **LOOP**; *зв'язок команди* **LOOP** *з регістром* **CX**; *команди циклу* **LOOPE**, **LOOPNE**, *додаткова перевірка ознаки нуля* **ZF**.

6.1. Теоретичні положення

6.1.1. Цикл з післяумовою. Команда LOOP

Команди циклу типу **LOOP** використовують регістр **CX** як лічильник циклу. Найпростіша серед них команда **LOOP**, яка зменшує регістр **CX** на одиницю і передає керування на позначку, якщо вміст регістра **CX** не дорівнює 0. Формат:

```
LOOP позначка
```

Якщо віднімання одиниці від регістра **CX** спричинило нульовий результат, то команда **LOOP** переходу не здійснює, а виконується наступна команда.

Наведений нижче програмний фрагмент демонструє організацію **циклу з післяумовою:**

```
 MOV   CX,  LOOP_COUNT
BEGIN_LOOP:

      ;  тіло циклу

LOOP BEGIN_LOOP
```

Програма записує число ітерацій циклу в регістр **CX** перед виконанням циклу. Потім виконується тіло циклу, а далі – команда **LOOP**. Вона зменшує лічильник на одиницю, що відповідає єдиній, щойно виконаній ітерації циклу.

Якщо тепер лічильник в регістрі **CX** дорівнює 0, то програма продовжує виконуватися після команди **LOOP**. Якщо лічильник не дорівнює 0, керування повертається на початок циклу, щоб здійснити ще один перехід тілом циклу. Тіло циклу виконується стільки разів, скільки було спочатку задано значенням регістра **CX**.

Увага: якщо програма всередині циклу змінює регістр **CX**, то число ітерацій циклу не буде відповідати початковому значенню в регістрі **CX**.

Описаний метод однаково добре працює, коли число циклів відоме наперед (як у прикладі, де LOOP_COUNT безпосереднє значення, що заноситься), і коли число циклів визначається під час виконання. Однак якщо обчислене число дорівнює 0, то цикл виконається 65536 разів. Чому ж так відбувається?

Коли мікропроцесор виконує першу команду LOOP, він зменшує CX від 0 до 0FFFFH. Оскільки тепер регістр CX ненульовий, повторює цикл. Отже, завантаження нульового значення лічильника циклів – *спеціальний випадок*. Аналогічна ситуація виникає і у випадку, коли початкове значення CX є *від'ємним*.

Наступний приклад аналогічний до попереднього за винятком того, що він завантажує регістр CX з елемента пам'яті, вміст якого обчислюється під час виконання програми. З цієї причини може виявитися, що лічильник циклів нульовий або від'ємний, і приклад використовує команду JNG (перейти, якщо не більше 0), щоб перевірити, чи треба цілковито пропустити тіло циклу.

```
MOV   CX, LOOP_COUNT_WORD
CMP CX,0
JNG  END_OF_LOOP
BEGIN_LOOP:

  ;     тіло циклу

LOOP  BEGIN_LOOP
END_OF_LOOP:
```

У програмі не обов'язково використовувати команду JNG в кожному циклі з лічильником, що обчислюється. Якщо програміст знає, що лічильник циклів ніколи не дорівнюватиме нулю або від'ємний, то перевірка не потрібна. Однак досвід показує, що значення, яке "ніколи" не повинне з'явитися, часом чомусь з'являється як тільки ви починаєте виконувати програму.

6.1.2. Команди циклу LOOPE, LOOPNE

Команда LOOPE (*цикл, доки дорівнює* 0) виконується увесь час, доки значення регістра CX не дорівнює нулю (CX≠0) і результат *додаткової перевірки* на вихід з циклу дорівнює нулю (ZF=1). Формат:

```
LOOPE позначка
```

Унаслідок виконання команди значення регістра CX зменшується на одиницю. Якщо в результаті CX=0 або ознака результату ZF=0, то перехід на позначку не відбувається. Аналог команди - LOOPZ.

Програма може завантажити в регістр CX максимальне число ітерацій циклу, а потім здійснити перевірку деякої додаткової умови завершення циклу.

Команда **LOOPNE** (*цикл, доки не дорівнює* 0) виконується увесь час, доки значення регістра CX не дорівнює нулю (CX≠0) і результат додаткової перевірки на вихід з циклу не дорівнює нулю (ZF=0). Формат:

LOOPNE позначка

Унаслідок виконання команди значення регістра CX зменшується на одиницю. Якщо в результаті CX=0 або ознака результату ZF=1 (результат нульовий), то перехід на позначку не відбувається. Аналог команди – **LOOPNZ**.

6.1.3. Приклади

Приклад 1. Обчислити

$$y = 1^2 + 2^2 + \ldots + 9^2.$$

```
INCLUDE LIBMACRO.INC
Program  Cycl1
.const
  res  db  ' y= ',0
Start
 ; Підготовка для підсумовування:
 ;  y  →  bx:=0
 ; Лічильник: cx:=9
   Sub bx,bx
   mov cx,9
 pcycl:
   mov ax,cx
   mul ax
   add bx,ax
 loop pcycl
   WLineZ  res
   WWordLn bx
Return
END      cycl1
```

Приклад 2. Обчислити

$$y = 1^2 + 2^2 + \ldots + n^2,$$

де n вводиться користувачем.

Реалізація прикладу 2.

```
INCLUDE LIBMACRO.INC
Program  Cycl2
.const
 pa   db ' Введіть n ',0
 per  db ' Увага! N>0!',0
 res  db  ' y= ',0
Start
 vv1:
   WlineZln pa
   RWord
   Jle err
 ; якщо n<=0 – помилка!
   mov cx,ax
   mov bx,0
   jmp pcycl
 err:
   WlineZln per
   Jmp vv1
 pcycl:
   mov ax,cx
   mul ax
   add bx,ax
 loop pcycl
   WLineZ  res
   WWordLn bx
Return
END      cycl2
```

Команда **jcxz** дає змогу організувати **цикл з передумовою**. У цьому випадку програміст сам має дбати щодо зменшення вмісту регістра **cx** на 1 і організовувати передавання керування на початок циклу. Покажемо застосування такого циклу на прикладі 3.

Приклад 3. Обчислити $y = 1^2 + 2^2 + ... + 9^2$, використавши **цикл з передумовою**.

```
INCLUDE LIBMACRO.INC
Program  Cycl3
 .const
  res  db ' y= ',0
Start
   mov bx,0
   mov cx,10 ; лічильник =10
  ; Подумайте, чому це так?
 pcycl:
   dec cx
   jcxz mres
   mov ax,cx
   mul ax
   add bx,ax
   jmp pcycl
 mres:
   WLineZ  res
   WWordLn bx
Return
END      cycl3
```

Приклад 4. Задано 10 чисел. Знайти серед них найменше та номер його першого входження у цю групу чисел.

```
INCLUDE LIBMACRO.INC
Program  Cycl4
 .const
 p1   db 'Введіть 1-е число ',0
 pch  db 'Введіть наступне число',0
 res1 db 'Найменше число: ',0
 res2 db 'Номер 1-го входження: ',0
.data
 povt dw 1
; povt зберігає номер введеного числа
Start
 WlineZln p1
```

Продовження прикладу 4

```
 RWord
 mov bx,ax
; bx зберігає найменше число
 mov dx,1
; dx зберігає номер 1-го входження
;     найменшого числа у групі
 mov cx,9
; cx зберігає число повторень для
;     введення наступних чисел
 pcycl:
   inc povt
   WlineZln pch
   RWord
   cmp ax,bx
; Чи нове число є меншим?
   jge endc ; ні,
; ідем на кінець тіла циклу endc
 ; так, реалізуємо переприсвоєн
; ня найменшого значення і його
; номера
   mov bx,ax
   mov dx,povt
 endc:
 loop pcycl
   WLineZ  res1
   WWordLn bx
   WLineZ  res2
   WWordLn dx
Return
END      cycl4
```

Приклад 5. Ввести з клавіатури декілька чисел. Перше число **N**>0 означає розмір наступної за ним групи чисел. Обчислити суму модулів від'ємних чисел.

```
INCLUDE LIBMACRO.INC
Program  Cycl5
.const
```

Продовження №1 прикладу 5

```
 pa  db 'Введіть n>0', 0
 pch db 'Введіть число', 0
 per db 'Увага! Має бути n>0!', 0
 res db 'Sum= ', 0
Start
   mov bx,0 ; bx зберігає суму
            ;     від'ємних чисел
 vv1:
   WlineZln pa
   RWord
   Jle err
 ; якщо n<=0 – помилка!
   mov cx,ax
   jmp pcycl
 err:
   WlineZln per
   Jmp vv1
```

Продовження №2 прикладу 5

```
 pcycl:
   WlineZln pch
   RWord
   jge endc ; якщо число
; є невід'ємним, то ідем на кінець
; тіла циклу endc
; інакше число є від'ємним, і до-
; даємо його до суми попередніх
; від'ємних чисел
   add bx,ax
 endc:
 loop pcycl
   neg bx  ; змінюємо знак
; отриманої суми від'ємних чисел
; (формуємо модуль суми)
   WLineZ  res
   WWordLn bx
Return .
END  cycl5
```

6.2. Завдання для самостійної роботи*

1. Прочитати з клавіатури 8 чисел, обчислити і надрукувати їхню суму.
2. Прочитати з клавіатури 22 числа, обчислити суму невід'ємних значень.
3. Задано з клавіатури 40 чисел. Обчислити кількість чисел, які дорівнюють нулю.
4. Серед 100 заданих чисел визначити кількість додатних, від'ємних і нульових.
5. Ввести з клавіатури групу чисел. Перше число N>0 означає розмір наступної за ним групи чисел. Обчислити суму модулів від'ємних чисел, що належать інтервалу [-30; -8].
6. Ввести з клавіатури групу чисел. Перше число X>0 означає розмір всієї групи чисел, у тім числі перше. Визначити відсоток додатних чисел (з точністю до цілого).
7. Задано 75 чисел. Знайти суму тих чисел, значення яких належать інтервалу [-12, +23].
8. Задано 32 числа. Знайти серед них найменше.

---

* У задачах цього розділу масиви чисел у пам'яті не зберігати, вводити їх з клавіатури почергово по одному.

9. Задано 60 чисел. Знайти серед них найбільше та його порядковий номер у цій групі чисел.
10. Задано 28 чисел. Знайти різницю між найбільшим і найменшим числом.
11. Обчислити суму добутків таких пар чисел: 3-4, 4-5, 5-6, ..., 11-12.
12. Обчислити суму часток та суму остач таких виразів: $(5p^2+3)/(3p)$, p=1,...,9.
13. Маємо групу, яка складається з не менше трьох чисел. Третє за порядком число означає кількість чисел у групі, у тім числі дане. Обчислити суму всіх чисел.
14. Задано групу з 52-х чисел. Знайти перше за порядком число, що дорівнює нулю, визначивши його позицію.
15. Задано групу з 48-ми чисел. Знайти суму модулів тих чисел, які передують першому нульовому, або усіх чисел групи, якщо нульових немає.
16. Задано групу з 24-х чисел. Знайти порядковий номер першого ненульового числа.
17. Задано групу з 17-ти чисел. Знайти кількість тих чисел, які передують першому ненульовому.
18. Задано групу з 25-ти чисел. Знайти середнє арифметичне чисел, розташованих за порядком за першим ненульовим. Зробити у двох варіантах:
    а) вважати, що обов'язково є хоча б одне ненульове значення, причому не на останньому місці;
    б) вважати, що ненульових значень може не бути взагалі.
19. Задано двобайтове число T>0. Обчислити:
    а) кількість цифр у десятковому записі цього числа;
    б) кількість ненульових цифр у десятковому записі цього числа;
    в) суму цифр заданого числа;
    г) суму цифр, які стоять на непарних позиціях, рахуючи справа наліво.
20. Задано однобайтове число M>0. Визначити число, отримане виписуванням в оберненому порядку цифр заданого числа (наприклад, 107 –> 701).

Тема 7. Індексування даних. Побудова циклів для індексованих даних

Базові поняття: *розташування групи чисел (масиву, вектора) у пам'яті; адреси чисел групи; індексні регістри* **SI**, **DI***; індексування адреси пам'яті; модифікація значень індексних регістрів; початкові та кінцеві значення індексного регістра.*

7.1. Теоретичні положення

7.1.1. Опис та ініціалізація одновимірного масиву в програмі

Масив - структурований тип даних, що складається з деякого числа елементів (групи чисел) одного типу. Якщо для однозначного визначення конкретного елемента масиву використовується один **індекс**, то такий масив називають *одновимірним масивом* (або *вектором*).

Спеціальних засобів опису масивів в програмах асемблера, звичайно, немає. При використанні одновимірного масиву в програмі його треба моделювати одним з **наступних способів**:

- **статичний** - задання елементів масиву в полі операндів однієї з директив опису даних (елементи розділяються комами). Наприклад:

```
; масив з 7 елементів. Розмір кожного елемента 2 байти
mas1 dw  111,112,113,114,115,116,117
```

- **статичний,** використовуючи оператор повторення **dup** з вказівкою початкових значень. Наприклад:

```
; масив з 5 нульових елементів. Розмір кожного елемента 1 байт
mas2 db  5 dup (0)
```

- **динамічний**, використовуючи оператор повторення **dup** без вказівки початкових значень. Наприклад:

```
; масив з 15 елементів. Розмір кожного елемента 4 байти
mas3 dd  15 dup (?)
```

Такий спосіб визначення використовується для **резервування** пам'яті для масиву. Ініціалізація елементів масиву реалізується у програмі з використанням **циклу** (значення формуються за певним законом або просто вводяться з клавіатури).

7.1.2. Доступ до елементів одновимірного масиву

При роботі з масивами необхідно чітко уявляти, що всі елементи масиву розташовуються в пам'яті комп'ютера **послідовно**. Саме по собі таке розташування нічого не говорить про призначення і порядок використання цих елементів. І тільки лише програміст за допомогою складеного ним алгоритму обробки визначає те, як треба трактувати

цю послідовність байтів. Так, одну і ту ж область пам'яті можна трактувати як одновимірний масив, і одночасно ті ж самі дані можуть трактуватися як двохвимірний масив. Все залежить тільки від алгоритму обробки цих даних в конкретній програмі.

Ці ж міркування можна розповсюдити і на **індекси** елементів масиву. Для того щоб локалізувати певний елемент масиву, до його імені треба додати індекс. Оскільки ми моделюємо масив, то повинні потурбуватися і про **моделювання індексу**. У мові асемблера індекси масивів - це звичайні адреси, але з ними працюють особливим чином. Наприклад, в програмі статично визначена група чисел:

```
mex dw  11,12,13,14,15
```

Нехай ця група (послідовність) чисел трактується як одновимірный масив. **Розмірність** кожного елемента визначається директивою **dw**, тобто вона рівна двом байтам. **Нумерація** елементів масиву в асемблері **починається з нуля**. Тобто, нульовий елемент вектора **mex** має значення **11**, перший – має значення **12**, …, а четвертий (останній) елемент має значення **15.** Таким чином, **індекс елемента одновимірного масиву** дорівнює порядковому номеру елемента у групі чисел мінус одиниця.

Щоб отримати доступ до третього елемента вектора **mex,** що має значення **14**, треба до адреси масиву **mex** додати 6. У загальному випадку для отримання адреси елемента в масиві необхідно до початкової (**базової**) адреси масиву додати добуток **індексу** цього елемента на розмір елемента масиву, тобто:

```
база + (індекс × розмір елемента)
```

Архітектура мікропроцесора надає досить зручні програмно-апаратні засоби для роботи з масивами. До них відносяться базові та індексні регістри, що дозволяють реалізувати декілька режимів адресування даних. Використовуючи дані режими адресування, можна організувати ефективну роботу з масивами в пам'яті.

При роботі з одновимірним масивом зручно використовувати *індексний* режим адресування, при якому ефективна адреса розташування елемента масиву в пам’яті формується з двох компонентів:

- **постійного** - вказівка **прямої** адреси масиву (**базової**) шляхом задання назви (імені, ідентифікатора) масиву;
- **змінного** - вказівка назви **індексного регістра**, у якому міститься значення **индекса масиву × розмір елемента**.

Наприклад:

```
mas dw  20,21,22,23,24,25
mov si, 4
; помістити 2-й елемент (значення 22) масиву mas в регістр ax
mov ax, mas[si]
```

Для обробки **всіх** елементів масиву (на прикладі **mas**) потрібно організовувати цикл за такою схемою:

```
        …
    mov cx,6     ; занесення кількості елементів mas в cx
    mov si,0     ; початкове значення індексного регістра
cycl:
        …
     ; реалізація алгоритму роботи з черговим елементом масиву mas
        …
     add si,2    ; перехід на наступний елемент масиву
loop  cycl
```

Зауваження. При організації циклів, в яких задається **додаткова умова** виходу з циклу, збільшити значення індексного регістру безпосередньо перед перед командою типу **loop** не можна, адже там треба розмістити команду **cmp**, що реалізовуватиме цю додаткову умову. Тоді можна запропонувати таку схему обробки всіх елементів масиву:

```
        …
       mov cx,6     ; занесення кількості елементів mas в cx
       mov si,-2    ; початкове значення індексного регістра
cycl:
       add si,2     ; перехід на черговий елемент масиву
       …
      ; реалізація алгоритму роботи з черговим елементом масиву mas
       …
      cmp mas[si],0 ; додаткова умова: черговий елемент mas=0?
                    ; ні – продовження циклу
loopnz  cycl
```

7.1.3. Приклади програм обробки одновимірних масивів

Приклад 1. Підрахувати кількість нульових елементів у деякому статично заданому векторі, де кожний елемент розміром в один байт.

```
 INCLUDE LIBMACRO.INC
 Program  Cycl6
 .const
 ; опис та ініціалізація масиву mas
 mas db 1,0,9,8,0,7,8,0,2,0
 start
     mov cx,10   ; занесення кількості елементів mas в cx
     mov ax,0    ; лічильник кількості нульових елементів
     mov si,0    ; початкове значення індексного регістра
 cycl:
     cmp mas[si],0 ; порівняння чергового елемента mas з 0
     jne m1        ; якщо не рівний 0, то перехід на m1
     inc al        ; збільшення на 1 кількості нульових елементів
 m1:
     inc si        ; перехід на наступний елемент масиву
 loop  cycl
     WWordLn al    ; виведення кількості нульових елементів
 Return
END     cycl6
```

Приклад 2. Підрахувати номер першого (за входженням у групу чисел) нульового значення. Групу чисел для тестування програми задавати статично. Кожне число має розмір у два байти.

```
 INCLUDE LIBMACRO.INC
 Program  Cycl7
 .const
  mes   db   'Немає нулів',0
  nom0  db   'Номер 1-го нульового елемента:',0
  mas   db   1,6,9,8,8,0,8,9,2,4
start
     mov  cx,10 ; настроювання лічильника циклу
     mov  si,-2 ; початкове значення індексного регістра
     mov  ax,0  ; початкове значення лічильника номера чергового
```

```
      ; числа. Рахуємо від 1, адже нас цікавить номер у групі чисел
cycl:
      add si+2    ; настроювання на обробку чергового елемента
      add ax,1    ; збільшення значення лічильника номера числа
      cmp mas[si],0 ; порівнюється черговий елемент mas з 0
loopnz  cycl
      jnz m1   ; аналіз виходу з циклу. Якщо не рівно 0, то на m1
      WlineZ  nom0
      WwordLn
      jmp exit
m1:
      WlineZln mes
exit:
 Return
 END    cycl7
```

Приклад 3. Задано 12 числа, які вводяться з клавіатури. Визначити, скільки серед них відрізняється від останнього числа.

Зауваження. Хоча числа вводяться з клавіатури, однак у програмі треба використати вектор, щоб дочекатися введення останнього числа і організувати порівняння з ним всіх попередніх чисел.

```
 INCLUDE LIBMACRO.INC
 Program  Cycl8
 .const
    pc db 'Введіть число:',0
    res db 'Кількість одинакових з останнім=',0
    mas dw 12 dup (?)
 start
   mov cx,12 ; настроювання лічильника циклу для введення чисел
   mov si,0  ; початкове значення індексного регістра для введення чисел

 ; цикл для введення чмсел
 cyclvv:
        Wlinez pc
        RWord ax
        mov  mas[si],ax
        add   si,2
```

```
loop cyclvv
   mov cx,11 ; настроювання лічильника циклу для порівняння чисел
   mov si,0 ; початкове значення індексного регістра для порівняння чисел
   mov di,22 ; формування зміщення останнього елемента масиву
   mov bx,0 ; настроювання лічильника кількості одинакових чисел з 12-м
 ; цикл для порівняння чмсел масиву з останнім 12-м числом
 cyclper:
        mov ax, mas[si]
        cmp ax, mas[di]
        jne m1 ; якщо числа не рівні , то перехід на m1
        inc bx
 m1:
        add si,2
 loop  cyclper
   Wlinez res
   WWordLn bx
 Return
 END     cycl8
```

Досить часто в програмі треба організовувати вкладені цикли (цикли в циклі). Основна проблема, яка тут виникає, - як зберегти значення лічильника в регістрі **cx** для кожного циклу. Для тимчасового зберігання лічильника зовнішнього циклу на час виконання внутрішнього циклу можна використати декілька способів: використати регістри, пам'ять або стек (див. наступний приклад).

Приклад 4. Задано вектор з 15 чисел. Вивести їх по 3 числа в рядок. Групу чисел для тестування програми задавати статично.

```
 INCLUDE LIBMACRO.INC
 Program  Cycl9
 .const
  mas dw  1,6,9,8,8,9,8,9,2,10,11,12,13,14,15
start
  mov cx,5 ; настроювання лічильника зовнішнього циклу – кількості рядків
  mov si,0 ; початкове значення індексного регістра
cyclzov:
    push cx  ; зберігання лічильника зовнішнього циклу
```

```
    mov cx,3 ;  настроювання лічильника внутрішнього циклу
             ; - кількості чисел в одному рядку
cyclvn:
     WWord mas[si]
     add si,2
loop cyclvn
     Linefeed
     pop cx ;  відновлення лічильника зовнішнього циклу
loop cyclzov
 Return
 END     cycl9
```

7.1.4. Двохвимірні масиви (матриці або прямокутні таблиці)

Як і випадку з одновимірними масивами, спеціальних засобів для опису двохвимірних масивів в асемблері немає - такі масиви треба **моделювати**. На описі самих даних це майже ніяк не відбивається – пам'ять під масив виділяється за допомогою директив резервування та ініціалізації пам'яті.

Безпосередньо моделювання обробки масиву реалізується у сегменті коду, де програміст, описуючи алгоритм обробки даних, визначає, що деяку область пам'яті необхідно трактувати як двохвимірний масив. При цьому програміст вільний у виборі того, як розуміти розташування елементів двохвимірного масиву в пам'яті: за рядками або за стовпцями. Якщо послідовність однотипних елементів в пам'яті трактується як двохвимірний масив розмірності $n \times m$ (тобто, має n рядків і m стовпців), який розташований **за рядками**, то адреса елемента (i, j) обчислюється за формулою:

$$\textbf{база} + m \times \textbf{розмір\_елемента\_масиву} \times i + j$$

Тут i (i=0,1, …, n-1) визначає номер рядка, а j (j=0,1, …, m-1) – номер стовпця. Наприклад, нехай є масив чисел *mas* (кожний елемент масиву займає 1 байт) розмірності 4 × 6 (i = 0, 1, 2, 3; j = 0,1, ..., 5):

23	04	05	67	54	76
05	06	07	99	55	44
67	08	09	73	32	12
87	09	00	08	25	91

У пам'яті елементи цього масиву будуть розташовані в наступній послідовності:

23 04 05 67 54 76 05 06 07 99 55 44 67 08 09 23 32 12 87 09 00 08 25 91

Для прикладу, елемент масиву *mas*(2, 3) має значення 73. Ефек-

тивна адреса *mas*(2, 3) = *mas* + 6 × 1 × 2 + 3 = *mas* + 15. Оскільки *mas* беспосередньо адресує перше значення групи чисел (23), то зміщення рівне 15 дійсно вказуватиме на потрібне значення (73).

Найпростіше організувати адресацію двохвимірного масиву за допомогою **базового режиму з індексуванням** , при якому ефективна адреса формується максимум з трьох компонентів:

• **постійного**, яким може виступати **пряма адреса** масиву у вигляді **назви масиву**;
• **змінного** (*базового*), що задається **назвою базового регістра**;
• **змінного** (*індексного*), що задається **назвою індексного регістра**.

Базовий регістр може містити зміщення до початку відповідного рядка, а **індексний регістр** – зміщення від початку рядка до потрібного елементу (або навпаки). Адреса елемента є сумою вмісту базового регістру, індексного регістру і адреси початку масиву.

7.1.5. Приклад програми обробки двохвимірних масивів

Приклад 1. Задана прямокутна таблиця двобайтових даних розмірності *n*×*m* (*n*=5, *m*=4). Обчислити:

а) максимальний елемент у кожному рядку;
б) суму елементів у кожному стовпці.

```
 INCLUDE LIBMACRO.INC
 Program  Cycl10
 .const
 n      dw  5 ; Кількість рядків
 m      dw  4 ; Кількість стовпців
 pch    db  ' Введіть наступне число ',0
 pvmax db  ' Максимальні значення рядків',0
 pvsum db  ' Суми елементів стовпців',0
.data
 mas    dw  20 dup(?)   ; резервування пам'яті для таблиці
 max    dw   5 dup(?)   ; зберігає максимальні значення рядків
 sum    dw   4 dup(?)   ; зберігає суми елементів стовпців
Start
  ; Введення групи з 20 чисел (рядками), що складають таблицю
  ; Зручно інтерпретувати цю групу спочатку як вектор
    mov cx,20
    mov si,0
cyclvv:
    wlinez pch
```

```
    rword
    mov mas[si], ax
    add si,2
loop cyclvv
     ; виконання завдання а)
    mov cx,n ;  лічильник рядків  таблиці mas
    mov bx,0 ;  початкове значення зміщення для рядків таблиці mas
    mov di,0 ;  початкове значення зміщення для елементів вектора max
cyclz2:
   mov ax,mas[bx]
   mov max[di],ax ;  початкове значення максимума поточного рядка max
   mov si,2       ;  початкове значення зміщення для елементів
                  ;  поточного рядка max
   push cx
   mov cx,m
   dec cx         ;  кількість порівнянь для знаходження максимума
                  ;  поточного рядка max
cyclv2:
    mov ax,mas[bx][si]
    cmp max[di],ax
    jge m1
    mov max[di],ax
m1:
    add si,2
loop cyclv2
    add bx,8        ; перехід на наступний рядок
    pop cx
    add di,2        ; перехід на наступний елемент вектора max
loop cyclz2

  ;  Виведення вектора максимальни значень рядків

    Wlinezln pvmax
    mov cx,n
    mov si,0
cyclvyv:
    wword max[si]
    add si,2
loop cyclvyv
```

```
    Linefeed
    ;  виконання завдання б)
     mov cx,m ;  лічильник стовпців таблиці mas
     mov bx,0 ;  початкове значення зміщення для стовпців таблиці mas
     mov di,0;  початкове значення зміщення для елементів вектора sum
cyclz3:
     mov sum[di],0  ; початкове значення суми поточного стовпця
     mov si,0 ; початкове значення зміщення для елементів поточного
                ; стовпця  mas
      push cx
      mov cx,n ;  лічильник елементів поточного стовпця mas
  cyclv3:
     mov ax,mas[bx][si]
     add sum[di],ax
     add si,8  ; перехід на наступний елемент поточного стовпця mas
loop cyclv3
     add bx,2  ; перехід на наступний стовпець mas
     pop cx
     add di,2  ; перехід на наступний елемент вектора sum
loop cyclz3
     ;  Виведення вектора значень суми кожного рядка
     Wlinezln pvsum
     mov cx,m
     mov si,0
cyclvyv2:
     wword sum[si]
     add si,2
loop cyclvyv2
 Return
 END     cycl10
```

7.2. Завдання для самостійної роботи

1. Прочитати з клавіатури у пам'ять 27 цілих однобайтових чисел.
2. Прочитати з клавіатури в пам'ять 15 однобайтових та 20 двобайтових чисел.

3. У пам'яті є масив із 120 однобайтових чисел. Обчислити різницю між кількістю від'ємних і додатних значень.
4. У пам'яті задано 38 двобайтових чисел. Визначити кількість пар чисел, для яких виконується умова: $x[i]=x[i+1]$; $i = 1, 3,..., 37$.
5. Задано 52 числа. Визначити, скільки серед них відрізняється від останнього числа.
6. Із 29 чисел, що є у пам'яті, надрукувати ті, що належать інтервалу [-18,+11].
7. Задано 100 чисел. Надрукувати спочатку всі від'ємні, потім - всі решта.
8. Задано 100 чисел. Надрукувати їх в оберненому порядку по 5 чисел в рядок.
9. У пам'яті є масив з 12 двобайтових чисел. Зменшити кожне у стільки разів, у скільки число **AL1X** більше від числа **AL2Y**.
10. Кожне із 36 чисел зменшити на величину середнього арифметичного всіх чисел.
11. Змінній величині **A01** присвоїти значення 1, якщо у масиві із 40 чисел є хоча б одне нульове значення, і присвоїти 0 у протилежному випадку.
12. У пам'яті є масив з 300 чисел. Знайти порядковий номер найбільшого числа масиву.
13. Змінній величині **UKAZ** присвоїти значення 10, якщо найбільший елемент вектора **VKT** з 70 елементів знаходиться у першій половині вектора, і присвоїти значення 20 у протилежному випадку.
14. Задано 2 масиви по 70 чисел. Утворити третій масив за правилом: якщо відповідні елементи заданих масивів рівні, то надати такому ж елементу третього масиву значення 0; якщо відповідний елемент першого масиву менший від елемента в другому, то у третьому масиві надати елементу зеачення 1; якщо більший - надати значення 2.
15. Виконати обчислення за формулами, вважаючи, що необхідні масиви задані у пам'яті:

а) $y = b + 2 * (a[2]\text{^}2 + a[3]\text{^}2 +...+ a[21]\text{^}2)$;

б) $m[i] = c[i] - b[i]/3 + 1$; $i=1, 2, ..., 28$;

в) $x[j] = y[j]\text{^}3 + 2*b[j] / (7*c[j])$; $j=1, ..., 10$;

г) $b[i] = \begin{cases} a[i]\text{^}2 - 20 / a[i], \text{ при } a[i] < -1; \\ a[i]\text{^}3 - 5, \text{ при } a[i] \ge -1, \end{cases}$ де $i=5,6,...,15$;

д) $a[m] = \begin{cases} 2 * x[m] - z[m\text{^}3]\text{^}3 / 2, \text{ при } t[m] = 0; \\ 4 * z[m] + 2, \text{ при } t[m] <> 0, \end{cases}$ де $m=-2,-1, ..., 13$;

е) $c[i] = (\max (a[i], b[i]))\text{^}2$; $i=1, 2, ..., 20$.

Тема 8. Логічні команди та їх використання

Базові поняття: *логічні (булеві) дані; зображення логічних та бітових даних у пам'яті; введення та виведення логічних даних* - **RBOOL**, **WBOOL**; *логічні команди* **AND**, **OR**, **XOR**, **NOT**; *ознаки результату логічних операцій та команда* **TEST**; *обчислення логічних виразів, відношень, умов.*

8.1. Теоретичні положення

8.1.1. Логічні команди

Чотирма основними логічними командами є **AND** (*і*), **OR** (*або*), **XOR** (*додавання за модулем* 2), **NOT** (*не*). Ці команди працюють безпосередньо з нулями і одиницями двійкового коду.

Найпростіша функція виконується командою **NOT**. Ця команда базується на визначенні одиниці і нуля, як істини (**TRUE**) і фальші (**FALSE**) відповідно. Твердження **NOT TRUE** (*не істина*) - це **FALSE** (фальш), а твердження **NOT FALSE** (*не фальш)* - це **TRUE** (*істина*). Тобто, команда **NOT** інвертує всі біти даного:

Значення	**NOT**(*Значення*)
1	0
0	1

Інші три логічні функції мають два операнди. Результати дій, зроблених кожною функцією над парою біт:

X	*Y*	*X* **AND** *Y*	*X* **OR** *Y*	*X* **XOR** *Y*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	1

Оскільки мікропроцесор працює з байтами або словами, він повторює результати таблиці для кожного біту операндів. Наприклад, байтова команда виконує *логічне І* зі значеннями нульового біта обох операндів і вміщує результат у нульовий біт операнда-результату. Потім ця команда повторює функцію *логічне І* з бітами від першого до сьомого. У результаті виходить побітова функція *логічне І* над окремими бітами операндів.

Команді **NOT** потрібний один операнд, а її форма ідентична команді **NEG**. Інші логічні команди копіюють синтаксис команд додавання і віднімання.

Коли мікропроцесор робить логічну операцію, він встановлює прапори відповідно до результату. Оскільки операція не арифметична, прапори перенесення і переповнення завжди встановлюються рівними нулю (0). Прапор додаткового перенесення після логічних операцій залишається невизначеним, в той час як інші прапори (*знак*, *нуль*) правильно відображають результат операції. Винятком є команда **NOT**, яка не змінює жодного прапора.

Першочергове призначення логічних операцій в мікропроцесорі - робота з бітами. Найменшою одиницею даних, з якою може працювати мікропроцесор, є байт. Жодна з арифметичних команд не може безпосередньо виділити або змінити єдиний біт, а логічні команди дозволяють програмі обробляти окремі біти.

Логічні команди можуть виділити окремі біти в байті або слові таким чином, що вони можуть бути *встановленими*, *скиненими* чи *перевіреними*. Для виділення бітів ці команди використовують **маску**.

Значення маски використовується командою побітно. Щоб *встановити* значення деякого біта рівним 1, треба використати команду **OR**. У цьому випадку всі значення маски - нулі, крім одиниці на місці біта, що встановлюється. Команда **OR** над маскою та іншим операндом-результатом встановлює 1 у вибраному біті операнда-результату, а інші біти результату залишають незмінними.

Аналогічно, команда **AND** може *скинути* певний біт. У цьому випадку всі біти маски містять одиниці, крім біта, що скидається (його значення – нуль). Цей біт скинеться в 0, а інші залишаться без змін.

Команда **XOR** (*додавання за модулем* 2) може виконати взаємне доповнення (**інвертування**) одного біта із заданим. Запишіть маску для команди **XOR** так, щоб на місці біта, що інвертується була 1, а на всіх інших місцях 0. Когда команда **XOR** виконається, біти, що відповідали нулям маски, залишаться без змін, а біти, що відповідали одиницям маски, інвертуються. Тобто, якщо початкове значення біта було 0, то 1 **XOR** 0 дає 1 (доповнення до 0), а якщо початкове значення було 1, то 1 **XOR** 1 дає 0 (доповнення до 1).

Остання логічна команда - **TEST** (перевірка). Ця команда ідентична команді **AND**, за винятком того, що вона не записує результат, але встановлює прапори відповідно до нього, тобто команда **TEST** відповідає команді **AND**, як команда **CMP** відповідає команді **SUB**. Ця команда перевіряє заданий біт, або набір бітів всередині байта або слова.

Як працює команда перевірки? Нехай програма хоче перевірити молодший біт байта (тобто, нульовий біт). Программа породжує маску **01H** або в регістрі, або як безпосереднє значення. Команда **TEST** (або

AND) дає результат з гарантованими нулями у всіх бітах, за винятком біта 0; а значення біта 0 відображає значення операнда-оригіналу.

Якщо нульовий біт оригіналу містить 0, то біт залишається рівним 0. Отже, результат нульовий, і встановлюється прапор нуля.

Якщо він спочатку дорівнює 1, то результат ненульовий, і прапор нуля скидається.

Команда **TEST** перевіряє заданий біт без руйнування інших бітів, оскільки ця команда не змінює поле результату.

Як правило, у програмі створюються маски, які одночасно можуть *встановити* чи *скинути* декілька бітів одночасно (див. приклади).

8.1.2. Введення логічних (булевих) значень

Формат:

```
RBOOL [[{регістр | пам'ять[[,лічильник]]} [[,ознака CR]] ]]
```

Перший та другий аргументи цієї команди аналогічні команді **RWORD.** Третій аргумент - ознака, чи потрібно натиснути *'Enter'* (код **CR**) для введення значення ознаки (0 по замовчуванні), чи сприймати введення відразу ж при натисненні першого символу (не нульове значення). Другий випадок корисний при відповідях на запитання типу "**Так/Hi**".

Для логічного "істина" слід ввести:

```
TRUE | T | 1 | YES | Y | TAK
```

Для логічного "фальш" слід ввести:

```
FALSE | F | 0 | NO | N | HI
```

При ненульовому третьому аргументі слід ввести лише одну букву без натиснення *'Enter'* (**T**, **1** чи **Y** для "істина" і **F**, **0** чи **N** для "фальш"). Результат введення - 16-бітове чи 8-бітове значення (в залежності від типу 1-го аргумента), усі розряди якого встановлені ("істина") чи нульові ("фальш").

8.1.3. Виведення логічних (булевих) значень

Формат:

```
WBOOL[[LN]] [[регістр|безпосереднє|пам'ять[[,лічильник]]
        [[,форма]] ]]
```

Перші два аргументи аналогічні аргументам макрокоманди **WWORD**. Якщо на вході значення першого аргумента дорівнює 0, то виводиться "*хибне*", якщо не нульове, то виводиться "*істинне*". Третій

аргумент вказує на те, у якій формі буде виводитися значення (за домовленістю береться 0):

Форма	Вигляд "*істиного*"	Вигляд "*хибного*"
0	*True*	*False*
1	*T*	*F*
2	*Yes*	*No*
3	*Y*	*N*
4	1	0
5	*Так*	*Hi*

8.1.4. Приклади використання логічних команд

Приклад 1. Задано 8 однобайтових чисел. Молодші три розряди кожного числа обнулити. Для спрощеного тестування програми масив чисел задавати статично.

```
INCLUDE LIBMACRO.INc
Program  BIT1
.const
mb db 1101b,1011b,111010b,101101b,1101b,1010b,101101b,100b
start
  mov cl,8
  mov si,0
  MOV bl,11111000B  ; маска для для обнулювання 3-х  молодших бітів
 ; цикл обнулювання 3-х  молодших бітів елементів і виведення результату
Cycl:
   and mb[si],bl
   wword mb[si],,b
   inc si
LOOP Cycl
linefeed
return
END   BIT1
```

Приклад 2. Задано 10 двобайтових чисел. Чотири найстарші та чотири наймолодші розряди кожного числа встановити в одиницю. Для спрощеного тестування програми масив чисел задавати статично.

```
INCLUDE LIBMACRO.INc
Program  BIT2
.const
```

```
mb dw
0a123h,123ah,1110h,1040h,1aa1h,1bbbh,01bh,100bh,3h,56h
start
   mov cl,10
   mov si,0
   MOV  bx,0f00fh ; маска для встановлення 4-х одиниць для найстарших
                  ; та наймолодших розрядів кожного числа
;цикл встановлення одиниць і виведення результату
Cycl:
   or mb[si],bx
   wword mb[si],,h
   add si,2
LOOP Cycl
linefeed
return
END   BIT2
```

Приклад 3. Задано 6 однобайтових чисел. У кожному коді розряди 1,3 і 5 і 7 поміняти на протилежні, а розряди 0, 2, 4 і 6 встановити рівними одиниці. Для спрощеного тестування програми масив чисел задавати статично.

```
INCLUDE LIBMACRO.INc
Program  BIT3
.const
mb db 11010101b,10111b,111010b,10101101b,11001101b,11111010b
start
   mov cl,6
   mov si,0
   MOV  bl,10101010B  ;  маска для інвертування бітів
   MOV  al,01010101B  ;  маска для встановлення одиниць
  ;цикл  інвертування та встановлення одиниць і виведення результату
Cycl:
   xor  mb[si],bl ;  інвертування бітів
   or  mb[si],al  ;  встановлення одиниць
   wword mb[si],,b
   inc si
LOOP Cycl
```

```
linefeed
return
END   BIT3
```

Приклад 4. Задано 10 двобайтових чисел. У всіх числах, у яких найстарший та наймолодший розряди дорівнюють одиниці, а розряди 5 та 6 дорівнюють нулю, встановити 4 молодших розряди рівними 1. Для спрощеного тестування програми масив чисел задавати статично.

```
INCLUDE LIBMACRO.INc
Program  BIT4
.const
mb dw
8a03h,920ah,9100h,1040h,0aaf1h,9bebh,01bh,100bh,3h,56h
start
   mov cl,10
   mov si,0
;цикл  обчислення  і виведення результату
Cycl:
   mov ax,mb[si]
   xor ax,0060h ;  інвертування 5 і 6 бітів (якщо були 0, то стануть 1)
   test ax,8000h ;  якщо найстарший біт=0, то на  m1
   jz m1          ;  нічого не змінювати
   test ax,0001h ;  якщо наймолодший біт=0, то на  m1
   jz m1          ;  нічого не змінювати
   test ax,0020h ;  якщо 5-й біт=0 (до інвертування=1), то на  m1
   jz m1          ;  нічого не змінювати
   test ax,0040h ;  якщо 6-й біт=0 (до інвертування=1), то на  m1
   jz m1          ;  нічого не змінювати
   or mb[si],000fh   ;  встановлення 4-х молодших розрядів рівними 1
m1:
   wword mb[si],,h
   add si,2
LOOP Cycl
linefeed
return
END   BIT4
```

Приклад 5. Обчислити значення відношення $a := 2*x + t < 2*c$, вибравши тип та довжину змінних величин довільно.

```
 INCLUDE LIBMACRO.INC
 Program   evallog
 .const
 pidx db ' Введіть x ',0
 pidt db ' Введіть t ',0
 pidc db ' Введіть c ',0
 two dw 2
 res db ' a= ',0
.data?
 a dw ?
 Start
 WLinez pidt
 RWord bx      ; Вводимо t
 WLinez pidx
 Rword         ; Вводимо x
 imul two
 add bx,ax     ; 2*x + t → bx
 WLinez pidc
 Rword         ; Вводимо c
 imul two      ; 2*c → ax
 cmp bx,ax
 jl m1         ; перехід на m1, якщо умова справджується
 mov ax,0      ; результат false
 jmp vyved
m1:
 mov ax,1      ; результат true
vyved:
 ; виведення результату
 WLinez res
 WBoolLn
Return
 END    evallog
```

Приклад 6. Обчислити значення логічного виразу $x := z \sim p \to a \wedge f \vee e$.

Позначення логічних операцій: знак "$\neg$" означає "*не*"; знак "$\wedge$" означає "*і*"; знак "$\vee$" означає "*або*"; знак "$\oplus$" означає "*додати за модулем* 2"; знак "$\sim$" означає "*еквівалентно*"; знак "$\rightarrow$" означає "*імплікація*".

Зауваження. 1. Операції "*еквівалентно*" та "*імплікація*" в асемблері треба представляти через визначені у ньому операції, тобто:

$$(x \to y) \Leftrightarrow (\neg x \vee y); \quad (x \sim y) \Leftrightarrow \neg(x \oplus y).$$

2. Пріорітет логічних операцій (перераховані у порядку спадання):

"$\neg$", "$\wedge$", "$\oplus$", "$\vee$", "$\sim$", "$\rightarrow$"

Отже, з врахування зауважень, треба обчислити наступний логічний вираз $x := \neg(\neg(z \oplus p)) \vee ((a \wedge f) \vee e)$. Після очевидного спрощення остаточно отримаємо $x := (z \oplus p) \vee ((a \wedge f) \vee e)$.

```
INCLUDE LIBMACRO.INC
 Program   evallog2
 .const
   pidz db ' Введіть z ',0
   pidp db ' Введіть p ',0
   pida db ' Введіть a ',0
   pidf db ' Введіть f ',0
   pide db ' Введіть e ',0
   res db ' x= ',0
 Start
   WLinez pidz
   Rbool bx,,1
   WLinez pidp
   Rbool ax,,1
```

```
; Продовження програми
   xor bx,ax
   WLinez pida
   Rbool ax,,1
   WLinez pidf
   Rbool cx,,1
   and cx,ax
   WLinez pide
   Rbool ax,,1
   or cx,ax
   or bx,cx
   WLinez res
   WBoolLn
Return
 END     evallog2
```

8.2. Завдання для самостійної роботи

1. Обчислити значення таких логічних виразів (позначення логічних операцій: знак "$\neg$" означає "*не*"; знак "$\wedge$" означає "*і*"; знак "$\vee$" означає "*або*"; знак "$\oplus$" означає "*додати за модулем* 2"; знак "$\sim$" означає "*еквівалентно*"; знак "$\rightarrow$" означає "*імплікація*").

а) $x := a \vee (\neg b \to c) \wedge d$; б) $x := z \sim p \to a \wedge f \vee e$;

в) $y := b \vee \neg c \vee \neg a \wedge (b \sim a)$; г) $y := (z \sim true) \wedge (m \oplus false) \wedge (\neg a \vee v)$;

д) $p[i] := z[i] \to r[i] \sim s[i] \oplus (a[i] \vee b[i]), \quad i = 1, ..., 15;$

е) $p[i] := (m[i] \sim true) \wedge (r[i] \rightarrow \neg t[i])$, $i = 1, ..., 8$;

є) $f := (b[1] \oplus b[2] \oplus ... \oplus b[12]) \rightarrow (d[1] \wedge d[2] \wedge ... \wedge d[19])$

ж) $f := (c[1] \vee c[2] \vee ... \vee c[10]) \oplus (b[1] \wedge b[2] \wedge ... \wedge b[10])$

- -

2. Обчислити значення відношень, вибравши тип та довжину змінних величин довільно: а) $a := 2*x + t < 2*c$; б) $a := -3*d <> f$;
в) $m := r - p/3 >= 10$; г) $m := -4*p + r = s+1$.

- -

3. Задано три двобайтові числа. Старші 6 розрядів та останній молодший розряд кожного числа обнулити.

4. Задано 2 двобайтові коди. Всі непарні разряди другої половини кожного коду обнулити.

5. Задано 16 однобайтових чисел. Старші шість розрядів кожного числа обнулити.

6. Задано дві групи по 7 однобайтових кодів. У кожному коді розряди 0,1,4,5 обнулити.

- -

7. Задано 4 однобайтових коди. Наймолодший та найстарший розряд кожного коду встановити в одиницю.

8. Задано 2 двобайтові та 1 однобайтовий код. Середні 4 розряди молодшого байту кожного коду встановити в одиницю.

9. Задано 25 двобайтових чисел. Три найстарші та два наймолодші розряди кожного числа встановити в одиницю.

- -

10. Задано 12 однобайтових кодів. У кожному коді розряди 0,1,2 поміняти на протилежні, а три найстарші встановити в одиницю.

11. Задано 35 двобайтових кодів. 4 найстарші розряди кожного коду обнулити, середні 6 поміняти на протилежні, а наймолодший встановити в одиницю.

- -

12. Задано 60 однобайтових чисел. У всіх числах, у яких найстарший та наймолодший розряди дорівнюють одиниці, а розряди 5 та 6 дорівнюють нулю, поміняти на протилежні 4 молодших розряди.

13. Задано 55 двобайтових чисел. У кожному числі, у якому розряди 0,2,4,6 дорівнюють нулю, а розряди 1,3,9,12,14 дорівнюють 1, встановити в одиницю три наймолодші розряди, і встановити в нуль розряди 7,8,9.

14. Задано 20 однобайтових кодів. У кожному коді, у якому розряди 2,3,4,6,7 дорівнюють відповідно 1,1,0,0,1, поміняти всі розряди на протилежні, якщо ж вказана умова не виконується, то парні розряди поміняти на протилежні, а непарні встановити в нуль.

Тема 9. Команди зсування

Базові поняття: *зсування інформації вліво та вправо; втрачений розряд, звільнений розряд; логічне зсування, арифметичне зсування*; *циклічне зсування; команди* **SHL**, **SHR**, **SAL**, **SAR**, **ROL**, **ROR**, **RCL**, **RCR**; *ознака* **CF** *при зсуваннях інформації.*

9.1. Теоретичні положення

9.1.1. Команди зсування

Команди зсування переміщують паралельно **усі біти** в полі даних або праворуч, або ліворуч. Як і інші логічні команди, зсування працюють з байтами або словами. Кожна команда має один операнд (регістр або елемент пам'яті), що зсувається, а також другий операнд, що є передумовленим.

В усіх командах зсування визначається **лічильник зсування**. Його найпоширеніше значення – одиниця. Такий лічильник задає зсування бітів (розрядів) операнда на одну позицію. Проте команда може задати довільний лічильник зсування, заносячи його значення до регістра **CL** перед зсуванням або через безпосереднє значення.

Число в регістрі **CL** може бути будь-яким від 0 до 255, однак його фактичні значення для зсування перебувають у межах 0 – 16. Значення 0 не спричиняє зсування, а будь-яке значення, більше 16, зсуває бітів більше, ніж містить операнд. Замість **CL** можна вказувати і **безпосереднє значення,** задане натуральним числом (його максимальним значенням для байтів є 8 і 16 – для слів).

Команда:

```
SHL регістр, CL
SHL регістр, натуральне число
SHL пам'ять, CL
SHL пам'ять, натуральне число
```

виконує **логічне** зміщення розрядів операнда-приймача **вліво** на задану кількість розрядів або на вміст регістра **CL**. Старші розряди почергово заносяться в ознаку **CF**, а молодші – доповнюються нулями.

Команда **SHR** має такий самий синтаксис, як і команда **SHL**, і виконує **логічне** зміщення розрядів операнда-приймача **вправо** на задану кількість розрядів або на вміст регістра **CL**. Молодші розряди почергово заносяться в ознаку **CF**, а старші – доповнюються нулями.

Команди логічних зсувань **SHL** і **SHR** розглядають знаковий біт як звичайний біт даних.

Команда **SAL** має такий самий синтаксис, як і команда **SHL**, і виконує **арифметичне** зсування розрядів операнда-приймача вліво на

задану кількість розрядів або на вміст регістра **CL**. Старші розряди, починаючи зі знакового, почергово заносяться в ознаку **CF**, а молодші – доповнюються нулями. Команда після зсування відновлює значення знакового розряду.

Команда **SAR** має такий самий синтаксис, як і команда **SHL**, і виконує **арифметичне** зсування розрядів операнда-приймача вправо на задану кількість розрядів або на вміст регістра **CL**. Молодші розряди почергово заносять в ознаку **CF**, а старші доповнюють тими значеннями, які були у знаковому розряді до початку виконання команди (0 або 1). Значення знакового розряду зберігається.

Спільна властивість команд зсування – це визначення ознаки перенесення (**CF**). Біт, що попадає за межі операнда, має спеціальне місце – **ознаку перенесення**. Якщо зсування відбулося на один біт, то біт з деякого кінця операнда стає новим значенням ознаки перенесення. У випадку багатобітового зсування біти почергово займають цю ознаку. Біти, витіснені з ознаки перенесення, **втрачаються**.

Циклічні зсування (**ROL**, **ROR**, **RCL**, **RCR**) переносять втрачений біт з одного кінця операнда в інший його кінець на **звільнений** біт. Циклічне зсування ліворуч **ROL** і циклічне зсування праворуч **ROR** розрізняють лише напрямом зсування даних. Аналогічно, циклічне зсування ліворуч з перенесенням **RCL** і циклічне зсування праворуч з перенесенням **RCR** є дзеркальним відображенням один одного.

Команди **ROL** і **RCL** розрізняють щодо трактування ознаки перенесення. Байтова команда **RCL** розглядає дані як 9-бітові, причому роль дев'ятого біта відіграє ознака перенесення. Якщо операнд - слово, команда **ROL** циклічно зсуває 16 бітів, а команда **RCL** циклічно зсуває 17 бітів.

Чому зсування називають арифметичним, якщо його зачислено до групи логічних команд? Зсування числа на одну позицію (біт) еквівалентне множенню або діленню цього числа на 2. Дійсно, у десятковій системі числення додавання нуля наприкінці числа означає множення його на 10. Аналогічно, у двійковій арифметиці додавання нуля (0) наприкінці числа означає множення цього числа на 2.

Якщо величина зсування більша за одиницю, то число множиться на 2, що піднесене до степеня, який дорівнює місткості лічильника зсування. Наприклад, зсування ліворуч на 3 біти еквівалентне множенню числа на 8.

Зсування числа праворуч на одну позицію – це те ж саме ділення на 2. Зміщений операнд – частка, а ознака перенесення – залишок. Якщо лічильник зсування більший за 1, зміщений операнд надалі залишається часткою, а залишок втрачається.

9.1.2. Приклади використання команд зсування

Приклад 1. Фрагмент програми, що множить число у регістрі **AX** на 9 без використання команди множення.

```
PUSH   AX        ;  Тимчасове збереження AX
MOV    CL,  3    ;  Будемо зсувати регістр AX на 3 розряди,
SAL    AX,  CL   ;  тобто множити на 8
MOV    DX,  AX   ;  CX <- AX * 8
POP    AX        ;  Відновлення AX
ADD    AX,  DX   ;  AX <- початкове значення * 9
```

Приклад 2. Фрагмент програми, що виокремлює один біт в регістрі **AX**, номер якого задано в регістрі **CL**.

```
PUSH   BX        ;  Збереження регістра BX у стекові
MOV    BX, 1     ;  Створення маски (1 в розряді 0 регістра BX)
ROL    BX, CL    ;  Зсування маски
AND    AX, BX    ;  Виокремлення необхідного розряду
POP    BX        ;  Відновлення регістра BX
```

Зауваження. Можна змінити цей приклад так, щоб виокремити більше одного біта зі слова. Наприклад, для виокремлення тетради певних розрядів треба замінити маску в регістрі **BX** на значення **0fh**.

Приклад 3. Скласти програму обчислення функції $f(x, y) = A_5$, де

$$A_0 = x;\ \ A_i = \begin{cases} 8 \cdot A_{i-1}, \text{ якщо чергова пара бітів у } y \text{ не дорівнює 11;} \\ A_{i-1} \operatorname{div} 3 + A_{i-1} \bmod 3, \text{ в інших випадках.} \end{cases}$$

Біти підраховують, починаючи із 5-ї пари. Значення y вводять.
Зауваження. У виразах операція **div** – цілочисельне ділення, а **mod** – остача від цього ділення.

```
INCLUDE LIBMACRO.INC
Program  BIT5
Start
  Wlinez Pidx
  RWord           ; введення x
  Wlinez Pidy
  RWord  CX       ; введення y
  MOV  BX,1000000000B ; маска для виокремлення бітів
      ; спочатку виокремлюють старший біт 5-ї пари
;цикл обчислення
```

```
LOOPCO: TEST CX,BX   ; перевірка старшого біта пари y
        JZ  NUL
        SHR  BX,1    ; модифікація маски для перегляду
                     ; молодшого біта поточної пари
        TEST CX,BX   ; перевірка молодшого біта пари y
        JZ  NUL
        CWD
        IDIV  TRY
        ADD  AX,DX
        JMP  SHORT  PROD
NUL:    SAL  AX,3
PROD:   SHR  BX,1   ; модифікація маски для перегляду
                    ; наступної пари бітів
JNC  LOOPCO
WWord  AX
Return
.CONST
Pidx db 'Введіть x',0
Pidy db 'Введіть y',0
TRY  DW  3        ;константа для ділення
END  BIT5
```

Приклад 4. Задано 12 двобайтових чисел. Кожне третє число збільшити у 4 рази шляхом зсування.

```
INCLUDE LIBMACRO.INc
Program  BIT9_9m
.data
ms dw  1,2,-3,4,5,-6,7,8,-9,10,11,-12
start
   mov cx,4
   mov si,4
cycl:
   sal ms[si],2 ; множення на 4
   add si,6
loop cycl
   mov cx,12
   mov si,0
; виведення масиву чисел
```

```
    cyclv:
       wword  ms[si]
       add si,2
    loop cyclv
    linefeed
    return
    END  BIT9_9m
```

Приклад 5. Задано масив з 16-ти двобайтових чисел, у яких значущими є лише молодші 4 розряди. Запакувати їх у 8 однобайтових полів пам'яті.

```
    INCLUDE LIBMACRO.INc
    Program  BIT9_12m
    .data
ms dw 1h,2h,3h,4h,5h,6h,7h,8h,9h,0ah,0bh,0ch,0dh,0eh,0fh,2h
    mb db 8 dup(?)
    start
       mov cx,8
       mov si,0
       mov di,0
    cycl:
       mov ax,ms[si]
       and ax,0fh     ; виокремлення значущих розрядів одному слові
       mov mb[di],al  ; запакування значущих розрядів
       shl mb[di],4   ; підготовка до занесення  наступної тетради
       add si,2       ; перехід на наступне слово пари у ms
       mov ax,ms[si]
       and ax,0fh     ; виокремлення значущих розрядів в одному
                      ; слові
       or mb[di],al   ; запакування значущих розрядів
       add si,2       ; перехід на наступне слово у ms
       add di,1       ; перехід на наступний байт у mb
    loop cycl
    ; виведення запакованого масиву чисел mb
       mov cx,8
       mov si,0
    cyclv:
       wword  mb[si],,h
       add si,1
```

```
    loop cyclv
      linefeed
    return
    END   BIT9_12m
```

Приклад 6. Задано число розміром у подвійне слово. Збільшити його у 32 рази шляхом зсування.

```
    INCLUDE LIBMACRO.INc
    Program  BIT9_15m
    .data
      a dd  -20
    start
      mov cx,5
    cycl:
      sal word ptr a+2,1 ; арифметичне зсування старших розрядів
      shl word ptr a, 1 ; логічне зсування молодших розрядів
      jnc m1  ; перевірка на наявність перенесення з молодших розрядів
      or word ptr a+2,1b
    m1:
    loop cycl
      wdwordln a
    return
    END   BIT9_15m
```

Приклад 7. Задано деякий однобайтовий код. Визначити кількість одиниць у його двійковому зображенні.

```
INCLUDE LIBMACRO.INc
Program  BIT9_17m
.data
  a db  11b
start
  mov cx,8
  mov ax,0 ; початкове значення лічильника
cycl:  rol a,1
  jnc m1 ; якщо перенесено 0, то на m1. Задаємо jc для підрахунку 0
  add ax,1 ; збільшення лічильника
m1: loop cycl
    wwordln ax
    return
    END   BIT9_17m
```

Приклад 8. Швидке обчислення значення виразу $\left(\frac{10x+16y}{32}\right)^z$.

де x, y, z – значення, які вводять.

```
INCLUDE LIBMACRO.INC
Program    QUICK_CALCULATION
Start
       RWord      AX          ;X
       SAL        AX,1
       MOV        BX,AX
       SAL        AX,2
       ADD        BX,AX       ;10*X
       RWord      AX          ;Y
       SAL        AX,4        ;16*Y
       ADD        AX,BX       ;a=(10*X + 16*Y)
       SAR        AX,5        ;a/32
       RWord      CX          ;Z
       MOV        BX,AX
       MOV        AX,1
LOOPCO: JCXZ KIN   ; цикл обчислення цілого степеня
            SHR        CX,1
            JNC        DAL
            IMUL       BX
DAL:        XCHG       AX,BX
            IMUL       AX
            XCHG       AX,BX
JMP  SHORT LOOPCO
KIN:   WWord    AX        ; результат
Return
END        QUICK_CALCULATION
```

9.2. Завдання для самостійної роботи

1. Задано 3 двобайтові числа. Збільшити їхні значення відповідно у 2, 8, 12 разів шляхом зсування.
2. Задано 3 двобайтові числа. Зменшити їхні значення відповідно у 4, 16, 32 рази шляхом зсування.
3. Задано 5 однобайтових чисел. Перші два числа збільшити у 9 разів, наступні два зменшити у 16 разів, а останнє збільшити у 6 разів шляхом зсування.

4. Задано 2 чотирибайтові числа. Перше число збільшити у два рази, а друге – зменшити у два рази шляхом зсування.
5. Задано чотири двобайтові числа. Запакувати їх в одне двобайтове значення по 4 біти на число.
6. Задано 3 однобайтові коди, у яких значущими є лише молодші 2 розряди. Запакувати їх в один байт по 2 біти на кожен код.
7. В однобайтовому полі пам'яті запаковано 3 значення по 2 біти на кожне. Розпакувати їх в окремі однобайтові поля пам'яті.
8. У двобайтовому полі пам'яті запаковано 3 значення по 5 бітів на кожне. Розпакувати їх в окремі однобайтові поля пам'яті.

- -

9. Задано 24 однобайтові числа. Кожне друге число збільшити у 4 рази шляхом зсування.
10. Задано 32 однобайтові числа. Числа, розташовані на позиціях, кратних 4 (тобто на 4, 8, 12, ..., 32), зменшити у 8 разів шляхом зсування.
11. Задано масив з восьми однобайтових кодів, у яких значущим є лише наймолодший розряд. Запакувати всі коди в один байт.
12. Задано масив з 60-ти двобайтових чисел, у яких значущими є лише молодші 4 розряди. Запакувати їх у 30 однобайтових полів пам'яті.
13. У масиві з семи однобайтових кодів маємо запаковано деякі значення по два біти на кожне. Розпакувати їх в окремі однобайтові поля пам'яті.
14. У чотирибайтовому полі запаковано 10 значень по 3 біти (два ліві біти не використано). Розпакувати їх в окремі слова пам'яті (двобайтові поля).

- -

15. Задано число розміром у подвійне слово. Збільшити його у 64 рази шляхом зсування.
16. Задано 5 чисел розміром у подвійне слово. Зменшити кожне число у 32 рази шляхом зсування.

- -

17. Задано деякий однобайтовий код. Порахувати кількість одиниць у його двійковому зображенні.
18. Задано байтовий код. Порахувати в ньому кількість нульових бітів.
19. Визначити кількість одиничних бітів у середніх двох байтах 4-байтового коду (подвійне слово).
20. Задане однобайтове число без знаку надрукувати у двійковій системі числення.
21. Задане двобайтове число без знаку надрукувати у вісімковій системі числення.

Тема 10. Опрацювання символьних даних

Базові поняття: *зображення літер і символьних рядків у пам'яті; стандарти кодування літер; макрокоманди* **RCHAR**, **WCHAR**, **RLINE**, **WLINE**; *формат рядка для* **RLINE**, **WLINE**; *особливості команд опрацювання рядків: адреси рядків, сегменти, автоматична модифікація адреси, напрям модифікації; команди* **CLD**, **STD**; *команди для рядків* **MOVSB**, **MOVSW**, **LODSB**, **LODSW**, **STOSB**, **STOSW**, **CMPSB**, **CMPSW**, **SCASB**, **SCASW**; *префікси повторення* **REP**, **REPE**, **REPNE**.

10.1. Теоретичні положення

10.1.1. Набір символів

Кожний байт інформації можна розглядати і як двійкове число, і як символьне значення. Кожне з двійкових чисел від 0 до 255 може зображати певний символ. Наприклад, число **41H** відповідає символові "**A**", а код **5EH** представляє символ "**^**".

Набір символів IBM PC є розширенням набору символів ASCII − Американського стандартного коду для обміну інформацією. У наборі ASCII значення символів від **20H** до **7EH** представляють звичайні символи латинського алфавіту, числові символи і розділові знаки.

Коди від **0H** до **1FH** зображають **символи керування**. Наведемо найбільш вживані символи керування:

Код	Символ	Значення	Код	Символ	Значення
0H	**NUL**	Порожньо	**0CH**	**FF**	Перехід на початок наступної сторінки
7H	**BEL**	Звуковий сигнал	**0DH**	**CR**	Повернення каретки в першу позицію
9H	**HT**	Горизонтальна табуляція	**0EH**	**SO**	Крок назад
0AH	**LF**	Перехід на наступний рядок	**0FH**	**SI**	Крок уперед
0BH	**VT**	Вертикальна табуляція	**1BH**	**ESC**	Вихід

Символьні значення від **80H** до **0FFH** є розширенням набору символів ASCII для IBM PC. Ці символи дібрано розробниками IBM з

метою розширення можливостей комп'ютера для виведення символів інших абеток, зокрема кирилиці, графічних і наукових символів.

Для задання рядка символів використовують команду **DB**. Рядок символів у цьому випадку виокремлюють апострофами чи лапками. Асемблер підбере відповідні значення кодів і розмістить їх у пам'яті – код кожного символу в окремий байт. Так асемблер може працювати тільки з символами в діапазоні від **20H** до **0FFH**.

Символи в діапазоні від **0H** до **1FH** повинні вводитися у програму як числа, оскільки у тексті початкового файла деякі символи керування використовують з метою позначення початку і кінця рядка.

10.1.2. Макрокоманди введення/виведення символу

Часто буває незручно виводити один символ, формуючи для нього окремий односимвольний рядок у пам'яті. Для **виведення** одного символу на пристрій передбачено окрему макрокоманду:

```
WCHAR
WCHAR символ
```

Її аргументом може бути будь-яке 8-бітове чи 16-бітове число (використовується лише молодша частина). За домовленістю символ виводиться із регістра **AL**.

Аналогічні міркування справедливі і щодо введення одного символу. Макрокоманда:

```
RCHAR
RCHAR регістр
RCHAR пам'ять
```

чекає на введення лише одного символу із пристрою введення.

Аргументом цієї макрокоманди може бути тільки 16-бітове число. За домовленістю використовується регістр **AX.**

10.1.3. Макрокоманди введення/виведення рядка символів

Введення рядка символів із клавіатури реалізують макрокомандою:

```
RLINE регістр [,обмежувач]
RLINE пам'ять [,обмежувач]
```

Ця макрокоманда вводить рядок символів до надходження коду переведення рядка (**CR** або **CR/LF** за умови введення із файла) або за умови досягнення рядком довжини, яку задано **обмежувачем** (за домовленістю 253). Обмежувачем може бути будь-яке 8-бітове значення. Якщо обмежувач нульовий, то ніяких дій не відбувається.

Якщо перший аргумент є регістром, то з нього надходить адреса операнда, куди заноситиметься рядок символів. Якщо перший аргумент є полем пам'яті – занесення здійснюється за адресою цього поля.

Введення символів із клавіатури функціонує так:
- усі символи заносяться почергово у пам'ять до натиснення клавіші *'Enter'*;
- якщо довжина рядка досягне значення обмежувача, то зі спробою ввести будь-який символ видаватиметься звуковий сигнал;
- у будь-якому випадку можна затерти останні введені символи клавішою '<--' (*'Backspace'*).

Якщо введення перепризначене, то з досягненням кінця рядка введення рядка завершується автоматично (без коду **CR/LF**).

Формат рядка, який використовується цією командою:

Довжина	Рядок символів			Код **CR**	Код **LF**
	Символ	...	Символ	(13)	(10)

Макрокоманда:

```
RLINEZ регістр [,обмежувач]
RLINEZ пам'ять [,обмежувач]
```

діє аналогічно до команди **RLINE,** тільки рядок у цьому випадку матиме вигляд:

Рядок символів			Код **NUL**
Символ	...	Символ	(0)

Виведення рядка символів реалізують макрокомандою:

```
WLINE[LN] регістр [,обмежувач]
WLINE[LN] пам'ять [,обмежувач]
```

Все, що стосується аргументів і формату для макрокоманди введення рядка, стосується і цієї команди. Рядок друкується на екрані або записується у файл. Кількість виведених символів визначається як мінімум із довжини рядка та обмежувача.

Для переведення рядка до мнемоніки макрокоманди необхідно дописувати **'LN'** (тобто **WLINELN**). Окрім того, що більшість макрокоманд введення/виведення мають власні засоби для переведення рядка (за допомогою суфікс **LN**), досить зручною є макрокоманда

```
LINEFEED
```

без аргументів. Вона спричиняє завершення і переведення рядка.

При виведенні підказок досить зручною є макрокоманда виведення рядка символів, який закінчується нулем:

```
WLINEZ[LN] регістр [,обмежувач]
WLINEZ[LN] пам'ять [,обмежувач]
```

Наведемо приклад простої програми демонстрації оформлення при виведенні результатів.

```
INCLUDE LIBMACRO.INC
Program  OFORM
COMMENT *
     Програма демонстрації простої роботи із байтовими
     значеннями та оформлення виведення результатів.
     Обчислює суму та добуток трьох цілих чисел
*
Start
   WLineZ  ZAPYT    ; запрошення до введення
   RWord   AL       ; введення байтового значення
   WLineZ ZAP2      ; запрошення до введення
   RWord  BL        ; введення байтового значення
   WLineZ ZAP3      ; запрошення до введення
   RWord  CL        ; введення байтового значення
   MOV    CH,CL     ; копія CL для множення
   ADD    CH,AL     ; додати перше і третє число
   ADD    CH,BL     ; шукана сума
   IMUL   BL        ; перемножити перше і друге число
   IMUL   CL        ; шуканий добуток
   WLineZ DOBUT     ; повідомлення про результат -добуток
   WWord  AL        ; виведення значення добутку
   WLineZ SUMMA     ; повідомлення про результат - суму
   WWord  CH        ; виведення значення суми
Return
; задання символьних рядків для підказок та повідомлень
.DATA
ZAPYT   DB  'Введіть значення трьох чисел :',13,10
        DB  '                          перше - ',0
ZAP2    DB  '                          друге - ',0
ZAP3    DB  '                          третє - ',0
DOBUT   DB  13,10,'      Добуток = ',0
SUMMA   DB  13,10,'      Сума    = ',0
END   OFORM
```

10.1.4. Особливості виконання команд опрацювання послідовності даних

Команди опрацювання послідовності даних (або ланцюгові команди) також називають командами опрацювання рядків. Назви майже синонімічні. Відмінність полягає у тому, що під *рядком* розуміють *послідовність байтів*, а ланцюг – узагальнена назва – вживається, коли елементи послідовності мають розмір слова чи подвійного слова.

Команди опрацювання послідовності даних дуже конкретні в адресуванні і не допускають яких-небудь варіацій. Рядкові команди адресують операнди комбінаціями регістрів **DS:SI** (операнд джерела) або **ES:DI** (операнд результату або приймача).

Усі команди опрацювання послідовності даних мають вбудовану корекцію адреси після виконання операції. Послідовність сформовано з багатьох елементів, однак команди опрацювання можуть працювати тільки з одним елементом у кожний конкретний момент часу. Автоматичне збільшення або зменшення адреси дає змогу швидко опрацювати усю послідовність.

Ознака напряму (**DF**) у регістрі стану керує напрямом опрацювання послідовності. Коли вона дорівнює одиниці, то адреса зменшується (послідовність опрацьовується справа наліво). Якщо ознаку скинено до нуля, то адреса збільшується (послідовність опрацьовується зліва направо).

Розмір операнда визначає кількість збільшень/зменшень. Байтові команди опрацювання рядків змінюють адресу на 1, а команди опрацювання послідовності над словами змінюють адресу на 2. Тим самим унаслідок виконання команди вказівник вказує на наступний або попередній символ рядка чи пару символів.

Команда без параметрів **CLD** встановлює ознаку напряму **DF** до нуля. Протилежною до команди **CLD** є команда **STD**, яка встановлює ознаку напряму **DF** до одиниці.

10.1.5. Команди опрацювання послідовності даних

Команду

```
[префікс] MOVS [ES:] приймач,[сегм_регістр:] джерело
[префікс] MOVS [ES:] приймач
[префікс] MOVS , [сегм_регістр:] джерело
[префікс] MOVS
```

використовують з метою копіювання даних із однієї ділянки пам'яті в іншу. Приймач може мати лише комірку пам'яті за адресою в **ES:DI**; як джерело за домовленістю беруть комірку за адресою в **DS:SI**, проте

за умови явного задання операндів можна змінити сегментний регістр за допомогою префікса сегмента.

Формально виконання команди зводиться до перенесення одиниці даних із джерела у приймач, причому вміст регістрів **SI** та **DI** збільшується або зменшується залежно від значення ознаки напряму **DF**, на розмір одиниці даних. Щоб переносити декілька одиниць даних однією командою, необхідно скористатися префіксом повторення. Зазвичай для команди **MOVS** використовують префікс **REP**.

За домовленістю одиницею пам'яті є слово, однак якщо використати команду **MOVSB** (або **MOVSD**), то одиницею пам'яті буде байт (або подвійне слово – тільки для процесора 80386). Для явного вказання на те, що одиницею пам'яті є слово, існує мнемоніка **MOVSW**.

Команду

```
LODS сегментний_регістр: джерело
LODS джерело
LODS
```

використовують з метою пересилання даних із послідовних комірок пам'яті в акумулятор (**AL**, **AX** чи **EAX** – залежно від розміру одиниці даних). Команда як джерело за домовленістю має комірку пам'яті за адресою в **DS:SI**, проте за явного задання операндів можна змінити сегментний регістр за допомогою префікса сегмента.

Формально виконання команди зводиться до перенесення одиниці даних із джерела в акумулятор , причому вміст регістра **SI** збільшується або зменшується (залежно від значення ознаки напряму **DF**) на розмір одиниці даних.

На відміну від інших команд опрацювання рядків, цю команду не використовують разом із префіксами повторення. Її зручно використовувати у циклах, які працюють із послідовностями даних.

За домовленістю одиницею пам'яті є слово, але якщо використати команду **LODSB** (або **LODSD**), то одиницею пам'яті буде байт (або подвійне слово – тільки для процесора 80386). Для явної вказівки на те, що одиницею пам'яті є слово, існує мнемоніка **LODSW**.

Команду

```
[префікс] STOS  ES:приймач
[префікс] STOS  приймач
[префікс] STOS
```

використовують з метою заповнення послідовних комірок пам'яті певним значенням. Приймач може бути лише коміркою пам'яті за адресою

в **ES:DI.** Значення для заповнення задається в акумуляторі (**AL, AX** чи **EAX** – залежно від розміру одиниці даних).

Формально виконання команди зводиться до занесення у приймач значення із акумулятора, причому вміст регістра **DI** збільшується або зменшується (залежно від значення ознаки напрямку **DF)** на розмір одиниці даних.

З метою заповнення декількох комірок пам'яті користуються префіксом повторення (зазвичай використовують префікс **REP**).

За домовленістю одиницею пам'яті є слово. Однак якщо використати команду **STOSB** (або **STOSD**), то одиницею пам'яті буде байт (або подвійне слово – тільки для процесора 80386). Щодо явної вказівки на те, що одиницею пам'яті є слово, існує мнемоніка **STOSW**.

Команду

```
[префікс] CMPS [сегм_регістр:] джерело,[ES:] приймач
[префікс] CMPS [сегм_регістр:]джерело
[префікс] CMPS ,[ES:]приймач
[префікс] CMPS
```

використовують з метою лексикографічного порівняння однієї послідовності даних з іншою. Приймач може бути лише коміркою пам'яті за адресою в **ES:DI.** Джерелом за домовленістю вважають комірку пам'яті за адресою в **DS:SI**. Однак за умови явного задання операндів можна змінити сегментний регістр за допомогою префікса сегмента.

Зауважимо, що на відміну від інших команд для цієї команди передавач (або джерело) необхідно розташувати зліва.

Формально виконання команди зводиться до порівняння одиниці даних передавача із даними у приймачі (ознаки встановлюють аналогічно команді **CMP**, тільки віднімання в іншому напрямі – порівняння передавача з приймачем, а не навпаки), причому вміст регістрів **SI** та **DI** збільшується або зменшується (залежно від ознаки напряму **DF**) на розмір одиниці даних.

Щоб порівнювати декілька одиниць даних однією командою, необхідно скористатися префіксом повторення. Зазвичай для команди **CMPS** використовують префікс **REP** та **REPNE**.

За домовленістю одиницею пам'яті є слово. Однак якщо використати команду **CMPSB** (або **CMPSD**), то одиницею пам'яті буде байт (або подвійне слово – тільки для процесора 80386). Щодо явної вказівки на те, що одиницею пам'яті є слово, існує мнемоніка **CMPSW**.

Команду

```
[префікс] SCAS ES:приймач
[префікс] SCAS приймач
[префікс] SCAS
```

використовують з метою пошуку першого входження чи невходження певного значення у послідовність даних. Приймач може бути лише коміркою пам'яті за адресою в **ES:DI.** Значення для пошуку задається в акумуляторі (**AL, AX** чи **EAX** – залежно від розміру одиниці даних).

Формально виконання команди зведено до порівняння одиниці даних у приймачі зі значенням в акумуляторі, причому вміст регістра **DI** збільшується або зменшується (залежно від ознаки напряму **DF**) на розмір одиниці даних.

Щоб здійснити пошук у послідовності даних, необхідно скористатися префіксом повторення (зазвичай, використовують префікс **REPNE** для першого входження і **REPE** для першого невходження).

За домовленістю одиницею пам'яті є слово. Однак якщо використати команду **SCASB** (або **SCASD**), то одиницею пам'яті буде байт (або подвійне слово – тільки для процесора 80386). Щодо явної вказівки на те, що одиницею пам'яті є слово, існує мнемоніка **SCASW**.

Префікс повторення **REP** вказує процесору на те, що ця команда повинна повторити своє виконання. Під час кожного виконання команди вміст регістра **CX** зменшується на 1. Як тільки значення в **CX** стане дорівнювати нулю, повторення припиняться.

Отже, кількість повторень задають перед виконанням повторюваної команди в регістрі **CX**.

Префікси **REPE, REPNE, REPNZ** і **REPZ** за своїм призначенням та способом дії є аналогічними префіксу **REP**, однак вони, крім реагування на вміст регістра **CX**, перевіряють ще й ознаку **ZF** у регістрі стану.

Наприклад, для префікса **REPE** (або **REPZ**) повторення припиняються, як тільки унаслідок чергового виконання команди ознака нуля **ZF** стане рівною нулю (для здійснення повторень перед командою слід встановити ознаку **ZF** в 1). Для префікса **REPNE** (або **REPNZ**), очевидно, все навпаки.

Під час роботи з послідовностями даних досить часто треба використовувати команду **LEA**, яка завантажує вказівник типу **NEAR** у спеціальний регістр. Регістром-приймачем може бути будь-який загальний регістр, операндом-передавачем – будь-який операнд пам'яті. Виконавча адреса оперента-передавача заноситься у регістр-приймач.

Команду

```
LEA  регістр, пам'ять
```

часто застосовують з метою обчислення адреси відносних операндів у пам'яті. Наприклад:

```
LEA  DX, STRING[SI]
```

10.1.6. Приклади опрацювання послідовності даних

Приклад 1. Прочитати з клавіатури два рядки. Якщо вони однакові (за довжиною та символами), то надрукувати слово "Збігаються", інакше – "Не збігаються".

```
INCLUDE LIBMACRO.INC
Program  Char10_6
.const
   pidk db 'Введіть рядок до 20-ти символів:',0
   dest1 db 21 DUP (' ') ; місце для 1-го рядка з NUL в кінці
   dest2 db 21 DUP (' ') ; місце для 2-го рядка з NUL в кінці
   ryes db 'Збігаються',0
   rno  db 'Не збігаються',0
start
    Wlinezln pidk
    Rlinez dest1,20   ; формування 1-го рядка з NUL в кінці
    linefeed
    Wlinezln pidk
    Rlinez dest2,20   ; формування 2-го рядка з NUL в кінці
    linefeed
    mov al,0 ; al містить код NUL – ознаку закінчення рядка
    cld
   ; підрахунок кількості символів 1-го рядка
    lea di,dest1
    mov cx,0
d1:
    scasb
    je endm1
    inc cx
    jmp d1
endm1: wwordln cx ; у cx маємо кількість символів 1-го рядка
   cld
   ; підрахунок кількості елементів 2-го рядка
    lea di,dest2
    mov bx,0
d2: scasb
    je endm2
    inc bx
    jmp d2
```

```
endm2: wwordln bx ; у bx маємо кількість символів 2-го рядка
    cmp cx,bx
    jne mno  ; якщо кількість символів різна, то перехід на mno
   ; порівняння двох рядків однакової довжини
    lea si,dest1
    lea di,dest2
    repe cmpsb
    je myes
mno: wlinezln rno
    jmp vyxid
myes:  wlinezln ryes
vyxid:
return
end     Char10_6
```

Приклад 2. Прочитати з клавіатури рядок символів. Усі символи "A" замінити на "a"; усі символи "B" замінити на "b".

```
INCLUDE LIBMACRO.INC
Program  Char10_10
   .const
   pidk db 'Введіть рядок з символами A та B',0
   dest db 30 DUP (' '),0   ; місце для рядка з NUL в кінці
start
    Wlinezln pidk
    Rlinez dest              ; формування рядка з NUL в кінці
    linefeed
    cld
  ; підрахунок кількості введених символів рядка
    lea di,dest
    mov al,0     ; al містить код NUL – ознаку закінчення рядка
    mov cx,0      ; cx – лічильник кількості символів рядка
d1:
    scasb
    je endd1
    inc cx
    jmp d1
endd1:
    lea di,dest
; організація послідовного за символами перегляду рядка
cycl1:
     jcxz endc1
     mov al,'A'
     scasb          ; виявляємо символ “A”
     jne mb         ; якщо “ні”, то на наступну перевірку
```

```
     dec di          ; корекція di для отримання адреси символу,
                     ; який збігається
     mov al,'a'
      stosb          ; заміна символу  “A” на “a”
mb:
    dec di ; корекція di для отримання адреси поточного символу
    mov al,'B'
    scasb            ; виявляємо символ “B”
    jne m1
    dec di
    mov al,'b'
    stosb
m1:
    dec cx
    jmp cycl1
endc1:
   wlinezln dest
return
end Char10_10
```

Приклад 3. У 5-байтовій ділянці пам'яті задано деякі літери. Переписати їх в іншу 30-байтову ділянку за допомогою 6-разового копіювання.

```
INCLUDE LIBMACRO.INC
Program  Char10ek
.const
sour db "ABCDE"
.data
dest db 30 dup(" "),0
start
       cld
       lea di,dest ; приймач
       mov cx,6    ; лічильник зовнішнього циклу
cycl:
       jcxz zacycl
       lea si,sour  ; джерело
       push cx
       mov cx,5      ; лічильник кількості символів для копіювання
       rep movsb
       pop cx
       dec cx
       jmp cycl
zacycl:
       wlinezln dest
return
end   Char10ek
```

Приклад 4. Задано масив з 4-х двобайтових чисел. Переписати їх в інший масив з 12-ти двобайтових чисел за допомогою триразового копіювання.

```
INCLUDE LIBMACRO.INC
Program  Char10ek
.const
     sour dw 234,75,-23,34
.data
     dest dw 3 dup(" "),0
start
     cld
     lea di,dest    ; приймач
     mov cx,3
cycl:
     jcxz zacycl
     lea si,sour    ; джерело
     push cx
     mov cx,4       ; лічильник кількості символів для копіювання
     rep movsw
     pop cx
     dec cx
     jmp cycl
zacycl:
      mov cx,12
      mov si,0
cvyv:
      wwordln  dest[si] ; виведення масиву-копії
      add si,2
loop cvyv
 return
end     Char10ek
```

10.2. Завдання для самостійної роботи

1. Задано символьний рядок "КОМАНДИ АСЕМБЛЕРА". Відкопіювати його в іншому місці пам'яті.

2. Задано символьний рядок "МИ ВИВЧАЄМО ПРОГРАМУВАННЯ". Кожне слово переписати в окрему ділянку пам'яті і надрукувати.

3. Задано символьний рядок "НЕЗАБАРОМ БУДЕ ЕКЗАМЕН". Кожне слово переписати в окрему ділянку пам'яті і надрукувати.

4. Деякий символьний рядок прочитати з клавіатури, замінити перші два символи на знаки "-" і результат вивести на екран.

5. Прочитати з клавіатури два рядки, вважаючи, що довжина кожного не менша за 4. Якщо перші 4 символи у них збігаються, то надрукувати перший рядок, інакше − другий.

6. Прочитати з клавіатури два рядки. Якщо вони однакові (за довжиною та символами), то надрукувати слово "Збігаються", інакше - "Не збігаються".

7. У пам'яті задано деякий рядок довжиною 16 символів. Перевірити, чи збігаються перші та останні чотири літери.

8. Задано деякий рядок з 24-х символів, і відомо, що в ньому є хоча б одна літера "*". Вияснити, в якій половині рядка трапляється найперше зліва ця літера.

9. Задано деякий рядок з 20-ти символів. Останній у рядку символ "+" змінити на символ ".".

10. Прочитати з клавіатури рядок символів. Усі символи "A" змінити на "a".

11. Прочитати з клавіатури рядок символів. Зробити його копію, замінивши при цьому букви "R", "S", "T" на "K", "L", "M" відповідно.

12. Прочитати з клавіатури рядок символів. Перелічити в ньому окремо кількість символів "(" та ")".

13. У пам'яті задано рядок символів. Останній символ "/". Замінити його на "0", а два передуючі йому − на символи "zz" − якщо їх є спереду не менше, ніж два. Надрукувати отриманий новий рядок.

14. У пам'яті задано рядок символів. Усі символи "B" замінити на "b", а символи "1" - на "2". Надрукувати новий рядок.

15. Задано ділянку пам'яті розміром 256 байтів. Прочитати у ній почергово три різні рядки, змінюючи їхню найпершу літеру на пропуск (" "),

та зразу друкувати. Перед читанням наступного рядка очищати всю ділянку пропусками.

16. Задано дві ділянки пам'яті розміром 15 і 40 байтів. Першу ділянку заповнити символом "0", а другу - символом "-".

17. Задано ділянку пам'яті розміром 30 байтів, у перших шести байтах якої є літери "+1*2/3" (шаблон). Копіювати цей шаблон у кожні наступні 6 байтів заданої ділянки.

18. У 4-байтовій ділянці пам'яті задано деякі літери. Переписати їх в іншу 20-байтову ділянку та копіювати у ній 5 разів.

- -

19. Задано масив з 22-х двобайтових чисел. Зробити його копію в інший масив.

20. Задано два масиви по 15 двобайтових чисел. Перевірити,чи на однакових позиціях є рівні за величиною числа, і надрукувати відповідне повідомлення.

21. Задано масив із 14-ти однобайтових чисел. Визначити, чи у його другій половині є число -10.

22. Задано масив з 18-ти однобайтових чисел. Знайти позицію останнього за порядком (зліва направо) числа +23.

23. Задано масив зі 120-ти двобайтових чисел. Усі числа +365 замінити числом +1.

- -

24. Прочитати з клавіатури рядок символів. Кожне його слово "ВЕСНА" замінити словом "ТЕПЛО".

25. Прочитати з клавіатури рядок симолів. Утворити новий рядок, вилучивши з нього усі пропуски і стиснувши решту тексту.

26. Задано рядок символів довжиною 60 літер. Перетворити його у новий рядок так: перед кожною буквою "R" додати по дві букви "VV", а після кожної букви "S" додати по одній букві "L".

Тема 11. Опрацювання таблиць

Базові поняття: *регулярність структури таблиці; поля та рядки таблиці; прямий доступ до таблиці; пошук у таблиці за одно- чи двобайтовим ключем; табличний пошук з використанням порівняння рядків; команда перекодування* **XLAT**, *підготовка таблиці перекодування.*

11.1. Теоретичні положення

11.1.1. Таблиці. Прямий доступ до таблиці

У загальному випадку термін *таблиця* можна застосувати до будь-якої множини зв'язаної інформації, об'єднаної за деякою ознакою і візуально зображеної у вигляді прямокутної таблиці. Наприклад, таблицею можна вважати розклад руху поїздів чи автобусів, книгу реєстрації даних щодо замовлення клієнтів і виконання замовлень тощо.

Найменування	Автор	Видавництво	Рік	Тип видання
Кобзар	Т.Г.Шевченко	Молодь	1995	Художня
Поезії	І.Я.Франко	Дніпро	1994	Художня
Збірник	В.М.Говоров	Наука	1996	Математична
Интернет	В.М.Байков	BHV	1998	Комп'ютерна

Приклад таблиці *Книги*

У будь-якій таблиці маємо поля (стовпці) та рядки. У **полі** розташована деяка **однорідна** інформація. Наприклад, у стовпці "*Рік*" має бути число, а не текст. **Рядок таблиці** містить інформацію, що характеризує один об'єкт, суб'єкт або явище. **Значення поля** - це інформація, що розташована на перетині конкретного рядка і конкретного стовпця. Наприклад, значенням поля "*Рік*" четвертого запису є "1996".

У асемблері таблиці визначають **систематично**, тобто кожне окреме поле таблиці має однаковий формат (числовий або символьний) і однакову довжину для кожного свого значення. Довжину поля визначають так, щоб можна було розташувати найдовше його значення.

Для спрощення роботи з таблицею бажано, щоб усі поля мали символьний формат (тобто 1 символ займав 1 байт і мав двійкове значення згідно з кодовою таблицею **ASCII**). Наприклад:

```
Knygy db 'Кобзар  ','Т.Г.Шевченко','Молодь','1995','Художня    '
      db 'Поезії  ','І.Я.Франко  ','Дніпро','1994','Художня    '
      db 'Збірник ','В.М.Говоров ','Наука ','1996','Математична'
      db 'Интернет','В.М.Байков  ','BHV   ','1996',"Комп'ютерна"
```

Кожне поле вирівнюється до фіксованої довжини шляхом дописування в кінці деякої кількості пропусків. Отже, пропуски у кінці поля вва-

жають незначущими символами. Увага! Необхідно, щоб:

1) кожне поле мало завжди одну і ту ж саму кількість символів у різних рядках;
2) сума довжин полів кожного рядка була однаковою.

У деяких таблицях можлива **послідовна** організація даних, за якою значення деякого поля (назвемо його **ключовим**) регулярно змінюється через деяке фіксоване значення (**крок**), найчастіше одиницю. За такої організації завдяки значенню ключового поля деякого рядка можна вирахувати адреси розташування у пам'яті значень інших полів цього ж рядка. Таку техніку роботи з таблицею називають **прямим доступом** до таблиці. Оскільки у більшості випадків значення ключового поля використовують тільки з метою обчислення потрібної адреси, то зберігати його у таблиці немає потреби. Конкретне значення ключового поля у цьому випадку задають з клавіатури або програмно.

У багатьох випадках значення ключа є символьним, а для визначення адрес полів треба мати його числове зображення. З метою **перетворення** десяткового символьного значення ключа, що має n цифр, у числове двійкове значення треба виконати таку послідовність дій:

1. Починають з правого символу-числа у форматі ASCII і опрацьовують всі n символів справа наліво.
2. Занулюють ліву шістнадцядкову цифру кожного ASCII-байта.
3. Послідовно множать ASCII-цифри на **1**, **10**, **100** (фактично на **01h**, **0ah**, **64h**), …, 10^{n-1} і додають результати.

Наведемо фрагмент програми, що переводить 4-цифрове символьне зображення у числовий формат:

```
        …
   .data
   AscVal db '1234' ; символьне зображення числа
   BinVal dw 0      ; число – результат перетворення
   AscLen dw 4      ; довжина символьного зображення
   M10    dw 1      ; степені 10
        …
   mov cx,10
   lea si, AscVal-1 ; завантаження адреси AscVal і відні-
   ; мання 1 для визначення крайнього правого символу
   mov bx, AscLen   ; завантаження довжини
cycl:
   mov al,[bx+si]   ; вибір чергового ASCII-символу
   and ax,000fh     ; занулення лівої цифри коду
   mul M10          ;  множення на степінь 10
   add binVal,ax    ; додавання чергового числа
   mov ax,M10       ; обчислення наступного
   mul cx           ; значення степеня 10
```

```
   mov M10,ax
   dec bx            ; Останній символ?
jnz cycl             ; Ні - продовжити
```

Приклад 1. Визначте таблицю прямого доступу, яка подає назви днів тижня. За даною літерою – номером дня – надрукувати назву цього дня.

```
INCLUDE LIBMACRO.INc
Program  Tab11_1
.const
   Dni    db 'Понеділок'
          db 'Вівторок '
          db 'Середа   '
          db 'Четвер   '
          db "П'ятниця "
          db 'Субота   '
          db 'Неділя   '
   pidk1 db 'Введіть номер дня',0
   pidk2 db 'Номер дня від 1 до 7',0
   Vyved db ' день - це ',0
.data
   rr  db 9 dup (' '),0  ; назва дня
   dch db ?              ; літера - номер дня
start
   ; початок діалогу отримання номера дня
pv:
   Wlinezln pidk1
   Wlinezln pidk2
   RChar
   linefeed
   mov dch,al
   And ax,000fh      ; занулення лівої цифри коду
    cmp ax,0         ; аналіз номера на приналежність
jle pv
    cmp ax,7         ; до проміжку від 1 до 7
jg  pv
   ; кінець діалогу отримання номера дня
   ; формування адреси назви дня відповідного номера
    lea si,Dni   ; завантаження адреси початку таблиці
    dec ax       ; зменшення номера дня на 1
    mov bx,9     ; обчислення зміщення від початку таблиці
    mul bx       ; до потрібного дня
    add si,ax    ; отримання адреси назви потрібного дня
    mov cx,9     ; кількість символів копіювання
    lea di,rr    ; копіювання назви дня
    rep movsb    ; у робочу комірку
    mov al,dch
    wchar        ; виведення результатів
```

```
    wlinez vyved
    wlinezln rr
return
end     Tab11_1
```

Приклад 2. Опишіть таблицю прямого доступу, яка визначає назви місяців і кількість робочих днів у кожному місяці. За введеними літерами – номером місяця - надрукувати назву цього місяця і кількість робочих днів у ньому.

```
INCLUDE LIBMACRO.INc
Program  Tab11_2
.const
; поля-рядки закінчуються кодом 0 (для спрощення виведення)
  Mis db 'Січень   ',0,'31',0
      db 'Лютий    ',0,'28',0
      db 'Березень ',0,'31',0
      db 'Квітень  ',0,'30',0
      db 'Травень  ',0,'31',0
      db 'Червень  ',0,'30',0
      db 'Липень   ',0,'31',0
      db 'Серпень  ',0,'31',0
      db 'Вересень ',0,'30',0
      db 'Жовтень  ',0,'31',0
      db 'Листопад ',0,'30',0
      db 'Грудень  ',0,'31',0
  pidk1 db 'Введіть номер місяця (01,02,...,12) ',0
  pidk2 db 'Нуль вводити!',0
  Vyv1  db ' Місяць  ',0
  Vyv2  db ' має  ',0
  Vyv3  db ' день(днів) ',0
.data
  ms  db 3 dup (' ')    ; літери - номер місяця
  nm  db 9 dup (' '),0 ; назва місяця
  kd  db 2 dup (' '),0 ; кількість робочих днів
start
 ; початок діалогу отримання номера місяця
 pv1:
    Wlinezln pidk1
    jmp vvd
 pv2:
    Wlinezln pidk2
 vvd:
    Rlinez ms
    linefeed
    cmp ms+1,0     ; якщо номер задається однією цифрою,
 je pv2            ; то на виведення відповідної підказки
    mov ah,ms
```

```
      mov al,ms+1
      And ax,0f0fh  ; занулення лівої цифри коду
      cmp ah,0      ; Місяць 01-09?
   jz m1            ; Так? Обійти!
      mov ah,0
      add al,10
   m1:
      cmp ax,0   ; аналіз номера на належність
   jle pv1
      cmp ax,12  ; до проміжку від 1 до 12
   jg  pv1
    ; кінець діалогу отримання номера місяця
    ; формування адреси назви місяця відповідного номера
      lea si,mis ; завантаження адреси початку таблиці
      dec ax     ; зменшення номера місяця на 1
      mov bx,13  ; обчислення зміщення від початку таблиці
      mul bx     ; до потрібного місяця
      add si,ax  ; отримання адреси назви місяця
      mov cx,9   ; кількість символів копіювання
      lea di,nm  ; копіювання назви місяця
      rep movsb  ; у робочу комірку
      inc si     ; отримання адреси кількості днів місяця
      mov cx,2   ; кількість символів копіювання
      lea di,kd  ; копіювання кількості днів місяця
      rep movsb  ; у робочу комірку
    ; виведення результатів
      wlinez vyv1
      wlinez nm
      wlinez vyv2
      wlinez kd
      wlinez vyv3
return
end     Tab11_2
```

Зауваження. У прикладах 1 і 2 для зображення номера дня чи місяця *спеціально* використано символьне зображення з метою демонстрації техніки роботи з *одно- чи двобайтовим ключем* для організації прямого доступу до таблиці. Зрозуміло, якщо задати номер у числовому форматі, то програма дещо спрощується (немає потреби здійснювати перетворення символьного формату в числовий).

11.1.2. Табличний пошук з використанням порівняння рядків

Хоча пряме табличне адресування є дуже ефективним, однак не для усіх задач воно підходить. Наприклад, якщо у таблиці з прикладу 2 треба знайти і роздрукувати місяці, що мають 31 день, то пряма адресація тут нам не допоможе. У цьому випадку необхідно організувати цикл

перегляду всіх рядків таблиці з метою порівняння поля-рядка *кількості днів* з фіксованим значенням 31. Відповідну програму наведено нижче.

```
INCLUDE LIBMACRO.INc
Program  Tab11_3
.const
Mis db 'Січень   ','31'
    db 'Лютий    ','28'
    db 'Березень ','31'
    db 'Квітень  ','30'
    db 'Травень  ','31'
    db 'Червень  ','30'
    db 'Липень   ','31'
    db 'Серпень  ','31'
    db 'Вересень ','30'
    db 'Жовтень  ','31'
    db 'Листопад ','30'
    db 'Грудень  ','31'
.data
ms  db '31'          ; значення для пошуку
kd  db 2 dup (' ')   ; кількість днів поточного місяця
nm  db 9 dup (' '),0  ; назва поточного місяця
start
  mov cx,12    ; лічильник циклу
  lea si,mis   ; завантаження адреси початку таблиці
cycl:
  push cx
  mov cx,9
  lea di,nm
  rep movsb  ; отримання назви поточного місяця
  lea di,kd
  mov cx,2
  rep movsb    ; отримання кількості днів поточного
               ; місяця
  push si
  lea si,ms
  lea di,kd
  mov cx,2
  rep  cmpsb   ; порівняння кількості днів поточного
  jne m1       ; місяця з '31', якщо 'ні', то на m1
  wlinezln nm ; виведення назви місяця, що має 31 день
m1:
  pop si
  pop cx
loop cycl
return
end     Tab11_3
```

11.1.3. Команда перекодування **XLAT**

Команду **XLAT використовують** з метою завантаження даних із деякої таблиці в пам'ять. Команда корисна для перекодування байтів із однієї системи кодування в іншу. Для команди необхідно завчасно підготувати **таблицю перекодування**, яка враховує усі 256 можливих символів з необхідними кодами. Формат:

```
XLAT
XLAT  [сегмент:] пам'ять
```

Команда не потребує операндів, однак другий формат можна використати для заміни сегмента.

Регістр **BX** має містити зміщення адреси початку таблиці в пам'яті (за домовленістю **DS** містить адресу сегмента, в якому задана таблиця). Перед виконанням **XLAT** у регістрі **AL** має міститися значення, що вказує на елемент таблиці (початком таблиці є 0). Після виконання **XLAT** регістр **AL** матиме значення, на яке вказує заданий елемент таблиці. Наприклад, якщо **AL=7**, то в **AL** після виконання **XLAT буде** занесено восьмий байт таблиці. Команда використовує значення у регістрі **AL** як відносну адресу в таблиці перекодування, тобто додає адресу у **BX** і зміщення у **AL.**

Приклад 4. Прочитати з клавіатури деякий літерний рядок. Використовуючи команду **XLAT** та підготувавши відповідну таблицю перекодування, замінити у рядку цифри 0,1,2,3 на цифру 3, цифри 4,5,6,7 на цифру 7, цифри 8,9 – на цифру 9. Усі інші літери замінити пропусками. Надрукувати отриманий рядок.

```
INCLUDE LIBMACRO.INc
Program  taB11_9
.data
pidk db 'Введіть рядок до 20-ти символів:',0
dest db 21 DUP (' ') ; ділянка пам’яті для зберігання
                     ; введеного рядка
; формування таблиці перекодування XTBL, на місці кодів
; десяткових цифр проставляємо нові коди, решта - пропуски
XTBL db 48 DUP (' ')
     db 33H,33H,33H,33H,37H,37H,37H,37H,39H,39H
     db 198 dup (' ')
start
 Wlinezln pidk
 Rlinez dest  ; введення початкового рядка з NUL у кінці
 linefeed
; початок визначення кількості символів введеного рядка
 mov al,0  ; al містить ознаку закінчення рядка
 cld
```

```
   lea di,dest
   mov cx,0 ; лічильник кількості символів рядка
d1:
   scasb
   je endm1
   inc cx
   jmp d1
endm1:
   wwordln cx ; виводимо кількість символів введеного рядка
  ; кінець визначення кількості символів введеного рядка
   lea bx,xtbl ; у bx заносимо адресу таблиці перекодування
   lea si,dest ; отримання адреси введеного рядка
  ; організація циклу перегляду всіх елементів рядка
cycl:
    mov al,[si] ; код чергового символу інтерпретується як
                ; зміщення у таблиці перекодування
    xlat        ; отримання нового коду в регістрі al
    mov [si],al ; занесення нового (закодованого) символу
                ; у рядок
    inc si
loop cycl
   wlinezln dest ; виведення перекодованого рядка
vyxid:
 return
 end     TaB11_9
```

11.2. Завдання для самостійної роботи

1. Визначити таблицю прямого доступу, яка містить українські та англійські назви днів тижня. За заданою літерою (номером дня) надрукувати назву цього дня двома мовами.

2. Визначити таблицю прямого доступу, яка містить назви вузів міста Львова (наприклад, 6-7 різних), а у другому полі – середню кількість студентів. За заданим числом – порядковим номером у списку ВУЗів - надрукувати у два рядки назву цього вузу та кількість студентів.

3. Визначити таблицю прямого доступу, яка містить 15 рядків - список канцелярських товарів деякого магазину. У кожному рядку є три поля: шифр товару, назва товару, ціна одиниці товару. За заданим словом з двох літер - номером рядка таблиці – надрукуйте у три рядки кожне з полів відповідного рядка таблиці.

---

4. Визначити таблицю, яка має 8 рядків. У кожному рядку є два поля: однобайтове поле ключа (будь-яка літера) та 18-байтове поле - назва фруктового дерева. Прочитати з клавіатури одну літеру (ключ назви дерева) та знайти у таблиці та надрукувати відповідну назву.

5. Визначити таблицю, яка має 14 рядків. У кожному рядку є три поля: дволітерне поле ключа (число, визначене літерами), семилітерне поле деякої абревіатури та двобайтове цілочисельне поле отриманого річного прибутку. За заданим дволітерним числом (ключем) знайти і надрукувати відповідну абревіатуру та річний прибуток.
6. Визначити таблицю з 5-ти рядків, вважаючи, що кількість рядків може змінюватися. Рядки впорядковані за зростанням ключа. За останнім рядком вписати відповідний обмежувач. Кожен рядок має два поля: дволітерне поле ключа (число, визначене літерами), та 20-байтове поле прізвища. За заданим дволітерним числом (ключем) знайти в таблиці і надрукувати відповідне прізвище.
7. Визначити таблицю, яка містить 7 рядків. У кожному рядку − два поля: 4-байтове поле ключа (будь-які літери), та 26-байтове поле назви предмета, який вивчає студент. Прочитати з клавіатури 4-літерний ключ, знайти за ним в таблиці і надрукувати назву відповідного предмета.
8. Визначити таблицю, яка містить 6 рядків, вважаючи, що кількість рядків може змінюватися. Рядки впорядковані за зростанням ключа. За останнім рядком вписати відповідний обмежувач. Кожен рядок складається з трьох полів: 4-літерного поля ключа, (число, визначене літерами), 16-байтового поля назви банку, 3-байтового поля місячного відсотка доходу з вкладів. За заданим 4-літерним ключем знайти в таблиці і надрукувати у два рядки назву банку та місячний відсоток доходу.

- -

9. Прочитати з клавіатури деякий літерний рядок. Використовуючи команду **XLAT** та підготувавши відповідну таблицю перекодування, замінити у рядку цифри 0,1,2,3 на цифру 3, цифри 4,5,6,7 − на цифру 4, цифри 8,9 − на цифру 8, малі латинські літери – на літеру “a”; великі латинські літери – на літеру “A”. Усі інші літери замінити пропусками. Надрукувати отриманий рядок.
10. Прочитати з клавіатури деякий рядок літер. Використовуючи команду **XLAT** та підготувавши відповідну таблицю перекодування, замінити малі та великі латинські літери даного рядка за таким правилом: перші 10 літер алфавіту − на знак '+', наступні 10 за алфавітом − на знак '/', останні 6 − на цифри 1,2,3,4,5,6 відповідно. Усі інші літери рядка замінити крапками. Надрукувати отриманий рядок.
11. Прочитати з клавіатури деякий рядок літер. Використовуючи команду **XLAT** та підготувавши відповідну таблицю прекодування, замінити всі малі та великі латинські літери на відповідні за звуком українські: A -> А, a -> а, B -> Б, C -> Ц, D -> Д, ..., G -> Ж, ..., Z -> З. Усі інші літери рядка залишити без змін. Надрукувати отриманий рядок.

Тема 12. Процедури

Базові поняття: *процедура (підпрограма) як логічна частина задачі*; *визначення процедури, директиви* **PROC**, **ENDP**, *тип процедури* **NEAR**, **FAR**; *виклики процедур* (**CALL**), *повернення до точки виклику* (**RET**); *використання стека для викликів/повернень*; *використання пам'яті та регістрів у процедурах*; *макрокоманди* **PUSHALL**, **POPALL**; *передача параметрів через стек*; *базування у стеку через* **BP**, *кадр стека*; *команда повернення* **RET** ***n***.

12.1. Теоретичні положення

12.1.1. Процедури

Процедура (або підпрограма) – послідовність команд, що реалізує деяку функцію або дію, яка виконується в програмі неодноразово і в різних місцях. Замість багаторазового повторення цієї послідовності у різних частинах програми, програміст розташовує ці команди в одному місці і формує процедуру – певним чином **оформлену** послідовність команд, яку можна викликати з будь-якої частини програми.

Програма може володіти однією або декількома процедурами або ж не мати жодної. До цього часу наші порівняно прості програми не мали процедур. Однак під час програмування достатньо серйозних програм трапляються ділянки коду, які часто повторюються. Вони можуть бути невеликими, а можуть займати чимало місця, суттєво заважаючи читанню тексту програми, знижуючи її наглядність, ускладнюючи налагодження, бути джерелом багатьох помилок. Звичайно, найкраще такі фрагменти повторення оформляти як процедури.

Кожний раз, коли програма вимагатиме функцію, що виконується процедурою, вона передасть керування цій процедурі. Передавання керування процедурі називають **викликом процедури.** Цей процес реалізують командою виклику (**CALL**). Виклик процедури відрізняється від команди переходу тим, що команда виклику **зберігає адресу наступної** за нею **команди**. Ця адреса – **адреса повернення**, – вказує шлях повернення до початкової послідовності команд після завершення роботи процедури. Адреса повернення під час виконання процедури зберігається у стекові.

Остання команда процедури є спеціальною командою повернення **RET** (від англійського *return* – повернення). Команда повернення бере адресу, збережену командою виклику до стека, і вміщує її у вказівник команд. Це примушує програму повернутися до команди, яка є наступною за викликом процедури.

12.1.2. Визначення процедури

За допомогою псевдокоманд **PROC** і **ENDP** визначають послідовність команд (або блок програми), який є процедурою (підпрограмою). Формат опису процедури:

```
нпр PROC [NEAR|FAR]
        ...
команди
        ...
нпр ENDP
```

Позначка **нпр** є назвою процедури (задається довільно, однак бажано вживати осмислені назви), тип якої задають за допомогою першого параметра (**NEAR** або **FAR**) псевдокоманди **PROC**. За домовленістю береться **NEAR**. Тип **NEAR** (ближній виклик) використовують у моделях сегментування **SMALL** і **COMPACT**, в усіх інших моделях використовують **FAR** (далекий виклик).

Тип **FAR** процедура повинна обов'язково мати у тих випадках, коли її опис і виклик розташовано у різних кодових сегментах або коли назва процедури є початком роботи всієї програми. Тип процедури впливає на тип команди повернення з неї. Якщо тип процедури – **NEAR**, то усі команди **RET**, розташовані у межах блоку опису процедури, трактуються як **RETN**, якщо тип далекий – як **RETF**.

Кожний блок опису процедури, який починається псевдокомандою **PROC**, повинен завершуватися псевдокомандою **ENDP**. Назва процедури в обох псевдокомандах має збігатися.

Усі назви позначок і змінних, описаних у блоці процедури, є локальними для цієї процедури щодо інших частин програми (використовувати ці назви можна лише у межах процедури). Позначку можна зробити глобальною, якщо описати її за допомогою двох двокрапок "::".

Блоки визначення процедур у програмі можуть розміщуватися:

- на початку програми перед командою **Start** (тип **FAR**);
- у кінці програми після команди **Return** (тип **NEAR**);
- всередині програми чи іншої процедури (у цьому випадку треба передбачити обхід процедури за допомогою команди **jmp**);
- в іншому файлі (у нашому посібнику не розглядатиметься; див., наприклад, підручник [3]).

12.1.3. Виклик процедури

Виклик процедури реалізується командою з форматом:

```
CALL пам'ять

CALL регістр
```

Команда здійснює перехід на виконання підпрограми за адресою, заданою єдиним операндом. Здебільшого як операнд задають позначку переходу, яку трактують як місце у пам'яті, куди здійснюється перехід. Якщо операндом задано регістр або змінна у пам'яті, то перехід здійснюється на адресу, яка міститься у цьому регістрі чи змінній (*відносний* перехід). Розрізняють 2 типи безумовного переходу: далекий (**FAR**), близький (**NEAR**).

Адреса далекого переходу містить два слова (**сегмент:зміщення**) і не може задаватися через регістр. Близький перехід виконується лише в межах поточного кодового сегмента, тому операнд команди - одне слово, яке можна задавати і в регістрі.

Зазвичай за умови правильної організації програми (за використання псевдокоманди **PROC** з метою опису підпрограми) розмірність і тип адреси визначають транслятором автоматично, проте іноді потрібно чітко зазначити певний тип переходу. Для чіткого визначення близького чи далекого переходу використовують оператор **PTR**.

Під час реалізації команди **CALL** процесор виконує такі дії:

1) завантажує код цієї команди, автоматично збільшуючи вказівник команд;
2) заносить у стек поточне значення вказівника команд, тобто адресу наступної команди (за умови далекого переходу за ним до стека заноситься також і значення поточного кодового сегмента - вміст регістра **CS**);
3) вказівник команд встановлюється на адресу, задану операндом команди - виконується перехід.

12.1.4. Повернення до точки виклику

Задля повернення з підпрограми необхідно здійснити зворотні дії щодо тих, які виконує команда **CALL**. Ці зворотні дії релізують командою **RET**, яка має формат:

```
RET
RET  безпосереднє значення
```

Команда здійснює повернення з підпрограми, виконуючи перехід на команду, розташовану відразу ж після команди **CALL**, якою зумовлено виконання підпрограми. Можна задати в команді кількість байтів (**безпосереднє значення**), які необхідно звільнити зі стека після повернення з підпрограми; зазвичай це потрібно при організації передавання аргументів підпрограмі через стек.

Під час виконання команди **RET** процесор виконує такі дії:

1) завантажує код команди, автоматично збільшуючи вказівник команд;

2) вказівник команд встановлюється на адресу, розташовану на вершині стека, причому у випадку далекого переходу (**FAR**) зі стека **вибирається** два слова − зміщення і сегментна адреса, а у випадку близького переходу (**NEAR**) - тільки зміщення. Тому дуже важливо стежити за тим, щоб команда повернення з підпрограми була того ж самого типу, що й команда переходу на підпрограму. За використання псевдокоманди **PROC** з метою опису підпрограми тип команди повернення вибирається компілятором автоматично. Якщо потрібно задати команду конкретного типу, то користуються мнемоніками **RETF** та **RETN**.
3) якщо задано аргумент команди (**безпосереднє значення**), то до покажчика стека **SP** додають **значення**.

Команда:

```
RETF
RETF  безпосереднє значення
```

здійснює далеке повернення з підпрограми.

Команда:

```
RETN
RETN  безпосереднє значення
```

здійснює близьке повернення з підпрограми.

Під час визначення процедур за допомогою псевдокоманди **PROC** ці команди використовують зрідка; адже зручніше скористатися для повернення з підпрограми командою **RET**, для якої компілятор автоматично визначить тип повернення.

12.1.5. Використання пам'яті та регістрів у процедурах

Під час використання процедур оперують термінами: *параметри* і *аргументи* (або *формальні* та *фактичні параметри*).

Аргументи (або *фактичні параметри*) – посилання на деякі дані або власне самі дані, необхідні для виконання покладених на процедуру функцій і розташовані поза процедурою.

Параметр (або *формальний параметр*) можна розглядати як певну формальну назву, яка зберігає місце для дійсних даних, які підставлятимуть у неї за допомогою аргумента. Параметр можна розглядати як елемент інтерфейса процедури, а аргумент – це те, що передається на місце параметра.

Для виконання процедури їй необхідно передавати аргументи. Спосіб передавання аргументів залежить від програміста. З цією метою використовують **регістри**, **пам’ять, стек** або **комбіновані варіанти** (регістри-пам’ять, регістри-стек, пам’ять-стек).

У якому вигляді можна передавати аргументи? При визначенні аргументів зазначено, що можна передавати як *власне дані*, так і *адреси* (*посилання* на дані). У мовах високого рівня це називають передачею за *значенням* і за *адресою*, відповідно.

Найпростіше передати *значення* аргументів в процедуру через регістри або стек. Якщо аргументи використовують значення деяких конкретних змінних програми, то за такої передачі у процедурі опрацьовуються копії цих змінних, а самі змінні не змінюються.

Під час передачі процедурі *адрес* змінних у ній опрацьовуються не копії цих змінних, а самі оригінали. Тому за умови зміни даних у процедурі вони автоматично змінюються і у програмі або ж процедурі, яка зумовила дану процедуру (оскільки зміни стосуються однієї і тієї ж ділянки пам'яті).

Спосіб передавання аргументів через регістри або пам'ять покажемо у цьому пункті на конкретних прикладах.

Приклад 1. Запрограмувати обчислення величини

$$r = \max(x + 2;\ y) + \max(x;\ y^2) + \max(3;\ y).$$

Значення змінних x та y ввести з клавіатури.

Попередні міркування. Очевидно, що для відшукання максимуму двох величин a і b варто визначити відповідну процедуру $\max(a; b)$. Для передавання аргументів домовимося використовувати регістри *ax* і *bx*, максимальне значення отримуватимемо у регістрі *ax*. Отже, регістри *ax* і *bx* у контексті самої процедури є формальними *параметрами*, а їхні кокретні значення під час виклику процедури задають *аргументи* процедури. Визначення процедури $\max(a; b)$ розміщуємо у кінці програми.

```
INCLUDE LIBMACRO.INC
Program  proc1
.const
  px   db  ' Введіть x ',0
  py   db  ' Введіть y ',0
  res  db  ' r= ',0
.data?
  x  Dw ?
  y  Dw ?
Start
  WLineZ px
  RWord x    ; введення значення x
  WLineZ py
  RWord y    ; введення значення y
  mov ax,x
  add ax,2  ; занесення першого аргументу у регістр ax
  mov bx,y  ; занесення другого аргументу у регістр bx
```

```
    call max  ; обчислення max(x+2; y), результат у ax
    mov cx,ax   ; початкове значення суми у регістрі cx
    mov ax,y
    imul ax   ; занесення другого аргументу у регістр ax
    mov bx,x  ; занесення першого аргументу у регістр bx
    call max  ; обчислення max(x; y^2), результат у ax
    add cx,ax  ; нагромадження суми у регістрі cx
    mov ax,3  ; занесення першого аргументу у регістр ax
    mov bx,y  ; занесення другого аргументу у регістр bx
    call max  ; обчислення max(3; y), результат у ax
    add cx,ax  ; отримання результату в регістрі cx
    wlinez res
    wwordln cx  ; виведення результату
  return
    max proc    ; визначення поцедури max(a; b)
      cmp ax,bx
      jge m1 ; якщо "так", то максимальне значення у ax,
      mov ax,bx ; а інакше у bx
      m1: ret
    max endp
  end proc1
```

Приклад 2. Запрограмувати обчислення величини:

$$a = \begin{cases} b^2 + 2d^4, \text{ якщо } b^2 > 25; \\ b^4 - d^2, \text{ інакше.} \end{cases}$$

Значення змінних *b* та *d* ввести з клавіатури.

Попередні міркування. Очевидно, що для відшукання квадрата величини *x* варто визначити відповідну процедуру sqr(*x*). Для передачі аргументу в процедуру і отримання квадрата цього аргументу домовимося використовувати ділянку пам'яті *r*. Для сумісності позначень регістри *bx* і *dx* використовуватимуться у головній програмі з метою збереження величин *b* і *d* відповідно, а регістр *ax* – з метою отримання величини *a*. Оскільки регістри *ax* і *dx* у процедурі sqr(*x*) використовуватимуться щодо множення, то на початку процедури поточні значення цих регістрів треба зберегти у стекові (сформувати **кадр стеку**), а в кінці процедури їх відновити. Визначення процедури sqr(*x*) розміщуємо на початку тексту програми.

```
INCLUDE LIBMACRO.INC
Program  proc2
.const
  pb db  'Введіть b ',0
  pd db  'Введіть d ',0
  res db 'a=',0
```

```
.data?
  r  Dw ?
sqr proc far
   push ax  ; збереження
   push dx  ; регістрів
   mov ax,r
   imul ax
   mov r,ax ; результат
   pop dx   ; відновлення
   pop ax   ; регістрів
   ret
sqr endp
Start
  WLineZ pb
  RWord bx ; введення b
  WLineZ pd
  RWord dx ; введення d
  mov r, bx ; аргумент у r
  call sqr ; результат у r
  cmp r,25 ; перевірка умови
  Jg m1 ; "так" -> 1-е галуження
  call sqr
  mov ax,r
  mov r,dx
  call sqr
  sub ax,r ; результат 2-го галуження
  Jmp m2
m1:
  mov ax,r
  mov r,dx
  call sqr
  call sqr
  add ax,r
  add ax,r ; результат 1-го галуження
m2:
  wlinez res ; виведення
  wwordln ax ; результату
return
end proc2
```

Приклад 3. У пам’яті задано дві групи однобайтових чисел (вектори). У кожній групі чисел замінити усі від'ємні значення на 0 (нуль). Результати оновлення вивести на екран.

Попередні міркування. Для заміни від'ємних значень вектора на нуль запрограмуємо процедуру **repl.** У процедуру передаватимуться два аргументи: **адреса** розташування **вектора** у пам’яті та **кількість** його

елементів. Передача адреси масиву є найпоширенішою практикою при роботі з масивами у процедурі (мінімізується час виконання і пам'ять). Водночас усі зміни елементів вектора у процедурі відображатимуться у реальному векторі, адреса якого передається процедурі.

Для виведення елементів масиву на екран дисплея реалізуємо у програмі процедуру **vyved.** Обидві процедури розмістимо у кінці тексту програми.

```
INCLUDE LIBMACRO.INC
Program  proc3
.data
     mx db -2,3,4,5,-6,7,-9
     my db  1,2,3,-4,5,-2
Start
      lea bx,mx ; адреса вектора mx в bx – 1 аргумент
      mov cx,7 ; кількість елементів вектора mx в cx
      call repl
      lea bx,my ; адреса вектора my в bx – 1 аргумент
      mov cx,6 ; кількість елементів вектора my в cx
      call repl
      lea bx,mx
      mov cx,7
      call vyved ; виведення оновленого вектора mx
      lea bx,my
      mov cx,6
      call vyved ; виведення оновленого вектора my
return
   repl proc
      mov si,0
    cycl:
      mov al,[bx+si] ; черговий елемент вектора в al
      cmp al,0       ; і порівняння його з нулем
      jge m1
      mov ah,0
      mov [bx+si],ah ; заміна на 0 від'ємного значення
     m1:
      inc si ; перехід на наступний елемент вектора
    loop cycl
    ret
   repl endp
   vyved proc
      mov si,0
    cycl1:
      mov al,[bx][si] ; черговий елемент вектора в al
      Wword   ; виведення чергового елемента вектора
      inc si  ; перехід на наступний елемент вектора
```

```
    loop cycl1
      linefeed  ; перехід на новий рядок
    ret
  vyved endp
end proc3
```

12.1.6. Передавання аргументів через стек

Головне призначення стека – тимчасове зберігання інформації. Стек використовують також для збереження адреси повернення з процедури. Крім цього, стек слугує зручним місцем для передачі інформації у підпрограми і з них.

Ми розглянули випадки, коли програма передає аргументи в підпрограму, вміщуючи їх у регістри або пам'ять. У деяких випадках з метою передавання аргументів краще скористатися стеком. Крім цього, стек є **єдиним засобом** передавання аргументів у підпрограми, написані мовою асемблера, з мов високого рівня типу С чи Паскаля. Отже, кожний висококваліфікований програміст повинен володіти цим механізмом передачі аргументів у процедуру.

Регістр вказівника стека (**SP**) - це 16-бітовий регістр, який визначає поточне зміщення вершини стека. Процесор використовує вказівник стека спільно з регістром сегмента стека для формування фізичної адреси. Процесор використовує пару регістрів **SS:SP** щодо всіх стекових посилань, у тім числі **PUSH**, **POP**, **CALL** і **RET**. Пара **SS:SP** завжди вказує на поточну вершину стека. Коли команда заносить у стек слово, регістр **SP** зменшується на два. Отже, стек росте у напрямі менших адрес пам'яті. При читанні зі стека слова процесор збільшує регістр **SP** на два. Усі стекові операції використовують значення розміром в слово.

Процесор змінює регістр **SP** з метою відображення дії над стеком. На регістр **SS** жодна зі стекових операцій не впливає. Це означає, що системний стек обмежений розміром 64 кілобайти та 16-розрядним режимом роботи.

Процедура може дуже просто читати значення аргументів зі стека, використовуючи регістр **BP** як вказівник на ділянку стека. Коли програма передає аргументи через стек, однією з перших команд у підпрограмі виконується команда

```
MOV BP, SP
```

яка завантажує у регістр **BP** поточне значення вказівника стека.

Оскільки регістр **BP** - адресний регістр, то підпрограма може використати його під час адресних обчислень, а це означає, що всі аргументи доступні у підпрограмі як зміщення щодо регістра **BP**.

Щодо доступу до даних регістр **BP** використовує за домовленістю регістр стекового сегмента **SS** як сегментний регістр. В усіх інших випадках доступу до даних мікропроцесор використовує регістр **DS**. Оскільки стек розташований у стековому сегменті, то регістрову пару **SS:BP** природно використати з метою адресування інформації у стекові.

```
 ;Приклад, що демонструє передавання аргументів
  PUSH  AX
  PUSH  BX
  PUSH  CX
  PUSH  DX
  CALL  SUBR ;  Виклик процедури
          …
SUBR  PROC NEAR
  PUSH BP       ; встановлення кадру
  MOV  BP, SP   ; стека
  MOV  AX,  [BP+4]  ;  Вибірка останнього аргументу (DX)
  MOV  BX,  [BP+6]  ;  Вибірка третього аргументу (CX)
  MOV  CX,  [BP+8]  ;  Вибірка другого аргументу (BX)
  MOV  DX,  [BP+10] ;  Вибірка першого аргументу (AX)
          …
         ;команди
          …
  POP  BP
  RET  8    ;  Повернення зі знищенням поля аргументів
SUBR  ENDP
          …
```

У цьому прикладі головна програма перед виконанням команди **CALL** записала чотири слова до стека. Процедура **SUBR** завантажує у **BP** вказівник вершини стека. Зауважимо, що зміщення, яке використовують з метою доступу до даних у стекові, враховує те, що до стека записано адресу повернення з процедури і збережено попереднє значення **BP** – тобто разом 4 байти.

У процедурі **SUBR** у вершині стека маємо попереднє значення **BP**, а за ним – адреса повернення. Потім у стекові вміщено останній аргумент, регістр **DX**; далі – через двобайтові інтервали - регістри **CX, BX** і **AX**. Отже, правильною адресою для читання аргументу, що міститься в регістрі **DX,** буде **[BP+4],** а інші адреси ідуть через двобайтові інтервали. У нашому прикладі значення, яке перебувало у регістрі **DX**, попадає до регістра **AX**, **CX** до **BX** і т.д.

Команда повернення з процедури **RET** може мати операнд – безпосереднє значення, що додається мікропроцесором до вмісту вказівника стека після читання адреси повернення. У прикладі використову-

ється значення 8; а це означає, що вісім байтів, або чотири слова даних необхідно вилучити зі стека після повернення. Ці значення зникають назавжди. Результат відповідає результатові у випадку читання значень зі стека, щоб знищити їх; команда повернення виконала це автоматично.

Процедура може повернути у стекові деяку інформацію програмі виклику. Якщо програма передає аргументи у стек, процедура може змінити їхні значення і залишити у стекові, а програма зможе витягнути їх зі стека після повернення. Якщо процедура повертає тільки один параметр, але викликалася з трьома аргументами у стекові, то виконати повернення можна за допомогою команди **RET 4**. У цьому випадку два аргументи видаляються зі стека, а третій аргумент залишається у ньому.

Зауваження. У наступних прикладах з метою передачі аргументів-масивів у процедури використовуватимемо стек, вважаючи, що ці процедури можуть викликатися з програм, написаних мовою високого рівня. Хоча, для зручності, процедури тестуватимемо в асемблерній програмі.

Приклад 4. У пам'яті задано дві групи двобайтових чисел (вектори). Для кожного вектора обчислити середнє арифметичне.

Попередні міркування. Як домовлено, аргумент-вектор будемо передавати у процедуру обчислення середнього арифметичного (**aver**) через стек. Оскільки у процедурі **aver** кожний елемент аргументу-вектора використовують тільки один раз під час обчислення суми елементів, то елементи аргументу-вектора у процедурі просто вибиратимуться зі стека за допомогою команди **pop.** Звичайно, у процедурі треба потурбуватися про збереження адреси повернення з процедури (з цією метою використовують регістр **bp**).

```
INCLUDE LIBMACRO.INC
Program  proc4
.const
  pv db 'Середнє арифм.: ',0
.data
  mx dw  20,30,40,50,-6,70,6
  my dw  10,20,30,0,5,-5
Start
  mov cx,7
  mov si,0
cycl1:
  push mx[si] ; занесення до стека елементів вектора mx
  add si,2
loop cycl1
  mov cx,7  ; передавання у процедуру кільк. елементів mx
  call aver ; обчислення середнього арифм. вектора mx
  Wlinez pv
  Wword     ; виведення середнього арифм. вектора mx
```

```
    Linefeed
    mov cx,6
    mov si,0
  cycl2:
    push my[si] ; занесення до стека елементів вектора my
    add si,2
  loop cycl2
    mov cx,6 ; передавання у процедуру кільк. елементів my
    call aver ; обчислення середнього арифм. вектора my
    Wlinez  pv
    Wword    ; виведення середнього арифм. вектора my
    Linefeed
  return
   aver proc
     pop bp ; збереження адреси повернення з процедури
     mov dx,cx ; збереження кількості елементів вектора
     mov ax,0 ; початкове значення суми елементів вектора
    cycl3:
     pop bx
     add ax,bx ; накопичення суми елементів вектора
    loop cycl3
     mov cx,dx ; відновлення кількості елементів вектора
     cwd
     div cx ; отримання середнього арифметичного
     push bp ; відновлення адреси повернення з процедури
   ret
   aver endp
  return
  end proc4
```

Приклад 5. У пам'яті задано дві групи двобайтових чисел (вектори). Необхідно скинути (встановити нулі) три молодші розряди кожного числа.

Попередні міркування. Як домовлено, аргумент-вектор передаватимемо у процедуру, що скидає три молодші розряди кожного елемента (**repbit**), через стек. Результати перетворення також отримуватимемо у стекові, а тому у процедурі **repbit** необхідно реалізувати адресування елементів вектора, що там є, через регістр **BP**. Під час переприсвоєння елементам векторів нових значень необхідно враховувати те, що у стекові їх розміщено у зворотному порядку. З метою виведення елементів вектора на екран дисплея реалізуємо у програмі процедуру **vyved.** Обидві процедури розмістимо у кінці тексту програми.

```
INCLUDE LIBMACRO.INc
Program  proc6
.const
```

```
ma dw 0fff9h,0fff8h,0fff7h,0fff6h,0fff5h,0fff4h,0fff3h,0fff2h
mb dw 0ffffh,0fffeh,0fff3h,0fff5h,0fff9h,0fff7h,0fffah,0fffbh
   start
     mov cx,8
     mov si,0
   cycl1:
     push ma[si] ; занесення чергового елемента ma до стека
     add si,2    ; перехід до наступного елемента вектора
   loop cycl1
     call repbit ; перетворення елементів вектора у стекові
     mov cx,8
     mov si,14  ; перегляд вектора ma у зворотному порядку
   cycl11:
     pop ma[si] ; оновлення чергового елемента вектора ma
     sub si,2   ; перехід до наступного елемента вектора
   loop cycl11
     lea bx,ma
     call vyved ; виведення елементів масиву ma
     mov cx,8
     mov si,0
   cycl2:
     push mb[si] ; занесення чергового елемента mb у стек
     add si,2    ; перехід на наступний елемент вектора
   loop cycl2
     call repbit ; перетворення елементів вектора у стекові
     mov cx,8
     mov si,14 ; перегляд вектора mb у зворотному порядку
   cycl12:
     pop mb[si] ; оновлення чергового елемента вектора mb
     sub si,2   ; перехід до наступного елемента вектора
   loop cycl12
     lea bx,mb
      call vyved ; виведення елементів вектора mb
   return
   repbit  proc
     mov bp,sp ; встановлення кадру стека
     mov cx,8
     mov si,2
   cycl3:
     and [bp+si],0fff8h ; встановлення нулів у 3 молодші біти
     add si,2 ; перехід на наступний елемент вектора
   loop cycl3
     ret
   repbit  endp
   vyved proc
     mov cx,8
     mov si,0
```

```
cycl4:
  mov ax,[bx][si] ; черговий елемент вектора в ax
  Wword,,h    ; виведення чергового елемента вектора
  add si,2    ; перехід до наступного елемента вектора
loop cycl4
  linefeed    ; перехід на новий рядок
ret
vyved endp
END   proc5
```

12.2. Завдання для самостійної роботи

1. Для заданого масиву з 20-ти двобайтових чисел зі знаком обчислити окремо суму всіх додатних і суму тих від'ємних значень, які не перевищують -10. Визначити та використати відповідні дві підпрограми.
2. Для заданого масиву з 35-ти однобайтових чисел зі знаком обчислити середнє арифметичне серед додатних значень, а також суму квадратів від'ємних. Визначити та використати відповідні дві підпрограми.
3. Задано 8 груп по 15 однобайтових чисел без знака. У кожній групі чисел виконати такі дії:
а) замінити усі від'ємні значення на -1;
б) усі значення, що належать відрізкові [28; 32], замінити числом 0;
в) усі додатні значення, що не належать відрізкові [28; 32], збільшити на 1.
Визначити та використати відповідні три підпрограми.
4. Задано 55 двобайтових кодів. Виконати таке опрацювання кодів:
а) для кодів, у яких найстарший і наймолодший розряди однакові, середні 4 біти замінити на протилежні;
б) для кодів, у яких три наймолодші розряди, кожен з яких дорівнює 1, встановити в 0 розряди 10,11,12,13,14.
Визначити та використати відповідні дві підпрограми.
5. Задано 10 груп по 5 однобайтових кодів. Для кожної групи окремо виконати таке опрацювання кодів:
а) замінити на протилежні всі розряди тих кодів, у яких два найстарші розряди дорівнюють 11;
б) посунути логічно вправо на 2 розряди ті коди, у яких середні 2 біти дорівнюють 00.
Визначити та використати відповідні дві підпрограми.
6. Задано рядок у 30 літер. Виконати такі дії:
а) відкопіювати початкове значення в іншій ділянці пам'яті;
б) замінити усі знаки '-' на '+';
в) обчислити позицію останнього в рядку пропуску.
Визначити та використати відповідні три підпрограми.

Кожну з наступних задач 7-12 розв'язати у трьох варіантах:
1) з використанням спільного поля пам'яті без передачі параметрів через стек;
2) з передачею відповідних параметрів через стек і їхнім вибором зі стека у підпрограмі;
3) з передачею відповідних параметрів через стек та їхнім адресуванням у стекові через регістр **BP** у підпрограмі.
Визначити підпрограми для кожного виду опрацювання даних (кожної дії) окремо.
7. Задано три масиви по 14 двобайтових чисел без знака. Для кожного масиву здійснити таке опрацювання: числа, які діляться на 3 без остачі, зменшити на 2, 1 та 3 відповідно для першого, другого та третього масиву.
8. Задано три масиви однобайтових чисел зі знаком, сформованих з 15, 25 та 21 елемента відповідно, і розташованих у різних ділянках пам'яті. Знайти найменше значення серед додатніх чисел для кожного масиву.
9. Задано дві групи по 12 двобайтових кодів. Щодо кожної групи здійснити таке опрацювання: а) у кожному коді, у якому найстарші два розряди дорівнюють 00, молодші 4 розряди встановити в 1; б) у кожному коді, у якому наймолодший розряд дорівнює 1, замінити на протилежні розряди 2, 3, 6.
10. Задано дві групи однобайтових кодів, сформованих з 8 та 66 елементів відповідно, і розташованих у різних ділянках пам'яті. Щодо кожної групи здійснити таке опрацювання: а) кожен код, у якого найстарші 3 розряди дорівнюють 000, зсунути вліво на 2 розряди; б) у кожному коді, в якому розряд 4 дорівнює 1, замінити на протилежний розряд 5.
11. Прочитати з клавіатури 4 рядки літер почергово в одне поле пам'яті. У кожному рядку, у якому є не менше 6 знаків '?', замінити ці знаки на інший, заданий в однобайтовому полі ZAMINA.
12. Прочитати з клавіатури 4 рядки літер різної довжини у різні ділянки пам'яті. У кожному рядку всі малі латинські літери замінити на:
- літеру '*' - для першого рядка;
- літеру '/' - для другого рядка;
- літеру, задану у полі пам'яті CHARREP, -для третього рядка;
- першу літеру цього рядка - для четвертого рядка.

Література

1. Абель П. Язык Ассемблера для IBM PC и программирования. – М.: Высш. школа, 1992. – 447 с.
2. Использование Turbo Assembler при разработке программ. – К.: Диалектика, 1994. – 288 с.
3. Юров В. Assembler. – СПб.: Питер, 2001. – 624 с.

Зміст

Навчальне видання

Ігор Михайлович Дудзяний,
Володимир Вікторович Черняхівський

Програмування мовою асемблера

Редактор І.М. Лоїк
Технічний редактор С.В. Сеник
Комп'ютерне макетування І.М. Дудзяний

Підписано до друку 31.07.02. Формат 60×84/16.
Папір друк.№ 3.
Різогр. друк. Умовн. друк. арк. 6,3. Обл.-вид. арк. 6,7.
Тираж 200 прим. Зам. 351

Видавничий центр Львівського національного університету
імені Івана Франка
79602, м. Львів, вул. Дорошенка, 41
