5. ПРОГРАМУВАННЯ НА ASSEMBLER

5.1. ОСНОВИ МОВИ ПРОГРАМУВАННЯ ASSEMBLER

Структура програми мовою *Assembler* залежить від платформи, для якої пишеться програма. Спочатку слід визначити операційну систему MS DOS.

Мова *Assembler* складається з директив та машинних команд. Вони дають комп'ютеру накази, що йому робити. Структура директив та команд однакові й складаються з таких полів:

- ім'я мітки;
- код операцій;
- операнди.

Ім'я мітки	Код операції (мнемокод)	Операнди	Коментарі

Ім'я мітки є ідентифікатором, значення якого визначає початкову адресу команди або змінної.

При написанні програми мітку можна не враховувати. Мнемокод та операнди обов'язкові. Використання міток дозволяє організовувати алгоритми, що розгалужуються.

Код операції – це мнемонічне позначення відповідної машинної команди або макрокоманди.

Операнди – частина команди, які позначають об'єкти, над якими проводяться дії.

Між полями крім коментарів потрібен хоча б один пробіл.

Усі команди та директиви мають мнемонічний код (мнемокод), який описує скорочено (або повністю) призначення директиви або команди національною мовою (у цьому випадку – англійською). Раніше були мнемокоди російською мовою й *Assembler* називався автокод. *Мнемокод* – це та сама машинна команда, однак зрозуміла програмісту.

Типи даних. Поняття типу даних неоднозначне. З погляду розмірності мікропроцесор апаратно підтримує такі типи даних:

Байт – вісім послідовно розміщених бітів, пронумерованих від 0 до 7, при цьому 0-й біт – молодший значущий біт;

Слово – послідовність із 2 байтів, які мають послідовні адреси від 0 до 15: 0÷7 – молодший байт, 8÷15 – старший байт. Мікропроцесорам

Intel властива особливість – молодший байт завжди зберігається за молодшою адресою. Адресою слова є адреса його молодшого байта;

Подвійне слово – послідовність із 4 байтів (32 біти): 0÷7 – молодше слово, 16–32 – старше слово. Адресою подвійного слова є адреса його молодшого слова;

Почетверене слово – послідовність із 8 байтів (64 біти): 0–32 – молодше подвійне слово; 33–62 – старше подвійне слово;

128-бітний упакований тип даних. Цей тип даних з'явився в мікропроцесорах *Pentium III*. Для роботи з ним передбачено спеціальні команди.

Крім трактування типів даних із погляду їх розмірності, мікропроцесор на рівні команд підтримує логічну інтерпретацію цих типів:

Цілий тип зі знаком – двійкове представлення зі знаком 8, 16 або 32 біти. Знак знаходиться в 7, 15 або 31 біти: 0 – "+", 1 – "–". Від'ємні числа представлені у додатковому коді:

8-розрядне ціле – 128÷127;

16-розрядне ціле – 32768÷32767;

32-розрядне ціле – $2^{31}÷2^{31}–1$.

Цілий тип без знака – двійкове представлення без знака розміром 8, 16 або 32 біти:

байт – 0...255;

слово – 0...65535;

подвійне слово – $0…2^{32}–1$.

Покажчик на пам'ять буває двох типів:

а) *ближній тип* – 32-логічна адреса, що представляє відносне розміщення в байтах від початку сегмента;

б) *дальній тип* – 48-розрядна логічна адреса, що складається з 2 частин: 16-розрядна сегментна частина (селектор) і 32-розрядне зміщення.

"Цепочка" – деякий безперервний набір байтів, слів або подвійних слів максимальною довжиною до 4 *Гбайт*.

Бітове поле – неперервна послідовність бітів, у якій кожний біт є незалежним і повинен розглядатись як окрема змінна. Бітове поле повинне розпочинатись з будь-якого біта, будь-якого байта і містити до 32 бітів.

Неупакований двійково-десятковий тип – байтове представлення десяткової цифри від 0 до 9; зберігається як байтове значення без знака по одній цифрі в кожному байті.

Упакований двійково-десятковий тип – упаковане представлення двох десяткових цифр від 0 до 9 в одному байті. Кожна цифра зберігається у своєму півбайті.

Тип даних із рухомою комою – один із декількох власних типів даних сопроцесора, несумісних із типами даних цілочислового пристрою.

Типи даних MMX-розширення (*Pentium MMX/II*). Даний тип даних з'явився в мікропроцесорах *Pentium MMX*. Він являє собою сукупність упакованих цілочислових елементів визначеного розміру.

Тип даних MMX-розширення (*Pentium III*). Даний тип даних з'явився в мікропроцесорах *Pentium III*. Він являє собою сукупність елементів із рухомою комою фіксованого розміру.

Директиви ініціалізації та опису даних мовою *Assembler*. Дані можуть розміщуватись у ділянках пам'яті, які називаються сегментами. Як правило, це або сегмент даних, або сегмент коду. Сегменти описуються за допомогою універсальної директиви SEGMENT або за допомогою сучасних спрощених директив .Model, .Code або .Data.

Для ініціалізації простих типів даних в *Assembler* використовують спеціальні директиви Dx, які є вказівками компілятору виділяти визначений об'єм пам'яті. Для мови *Assembler* має значення тільки довжина комірки, у якій розміщені дані, а які це дані залежить від програміста.

Мнемокоди директив ініціалізації Dx означають:

DB (*Define Byte*) – резервувати пам'ять для даних розміром 1 *байт* (8 бітів, BYTE);

DW (*Define Word*) – резервувати пам'ять для даних розміром 2 *байт* (16 бітів, *WORD*);

DD (*Define Double Word*) – резервувати пам'ять для даних розміром *подвійне слово 4 байт* (32 біти, DWORD);

DF – резервувати пам'ять для даних розміром 6 *байт*;

DQ (*Define Quarter Word*) – резервувати пам'ять для даних розміром *почетверене слово 8 байт* (64 біти, QWORD);

DT (*Define Ten Bytes*) – резервувати пам'ять для даних розміром 10 байтів (TBYTE).

В *Assembler* операція опису й ініціалізації даних відповідає розподілу й ініціалізації пам'яті. При розподілі пам'яті програмісту, як правило, відомо з яким форматом даних доведеться працювати, але невідомо, яке значення буде зберігатися у тій чи іншій комірці пам'яті. У цьому випадку після операнда директиви Dx ставиться знак питання. А

якщо потрібно виділити неперервну ділянку пам’яті із декількох комірок, пишеться параметр коефіцієнта повторювання dup.

У табл. 5.1. наведено адекватність опису та ініціалізації даних різними мовами програмування.

Таблиця 5.1

Адекватність типів даних

Pascal	C/C++	Assembler

N: integer;	int N; //Win16
	//Win32
		N dw ?
		N dd ?

M: single;	float M;	M dd ?
A: double;….B: = –898,68;	double A = –898,68;	A dq –898,68
Mas: array [1..5] of word;	unsigned short int Mas [ ] = {0,0,0,0,0};	Mas DW 5 dup(0)

У цьому випадку всі комірки є поіменованими, тобто з ними можуть працювати команди та директиви.

Загальну структуру програми на *Assembler* для MS DOS надано у табл. 5.2.

Таблиця 5.2

Загальна структура програми на *Assembler*

Title	Заголовок програми
Model **large**	Модель пам’яті
Model **tiny**	
Model **small**	
Model **medium**	
Model **compact**	
Model **huge**	
.STACK [розмір]	Визначає початок або продовження сегмента стеку модуля. Параметр [розмір] задає розмір стеку в байтах
Dx	Директиви розподілу й ініціалізації пам’яті
.DATA	Початок або продовження сегмента даних. Використовується також для опису даних типу near 1
Dx	Директиви розподілу й ініціалізації пам’яті для змінних
Extrn	Директива опису зовнішніх змінних
.CODE [ім’я]	Визначає початок або продовження сегмента коду
Dx	Директиви розподілу й ініціалізації пам’яті для змінних
Extrn	Директиви опису зовнішніх посилань та/або змінних

Закінчення табл. 5.2

Proc … endP	Процедури
END	Кінець програми

Програма може починатися з необов'язкового заголовка, однак обов'язково закінчується директивою END. Наявність сегментів та їх кількість залежить від специфіки задачі. Програмі привласнюється ім'я та надається розширення asm. Нижче наведено приклад програми мовою *Assembler* для MS DOS. У програмі ініціалізовано змінні a, b, c та d. Сегмент коду інструкцій процесору не містить.

```
;загальна структура програми
title priklad.asm
masm            ;режим роботи для TASM, для  MASM - непотрібно
Model small       ;директива моделі пам'яті
.stack 256h       ;опис сегмента стеку
.Data             ;опис сегмента даних
a db 5
b db -2
c dw 10
d dw -4
.Code               ;опис сегмента коду
start:
    mov ax, @data
    mov ds, ax
;----------------далі програмний код ---
;……………………….
;------------ далі передача управління операційній системі
exit:
    mov ax, 4c00h
    int 21h
    end start
```

Обов'язковим параметром директиви Model є модель пам'яті. Цей параметр визначає модель сегментації пам'яті для програмного модуля. Передбачається, що програмний модуль може мати тільки визначені типи сегментів, які визначаються так званими спрощеними директивами опису сегментів (табл. 5.2).

Моделі пам'яті. Операнди директиви Model задають модель пам'яті, яка визначає набір сегментів програми, розміри сегментів даних та коду, способи зв'язування сегментів та сегментних регістрів. У табл. 5.3 наведено деякі значення параметрів моделі пам'яті директиви Model.

Таблиця 5.3

Моделі пам'яті

Модель	Тип покажчика		Кількість та розмір сегментів		Призначення моделі
	коду	даних	коду	даних	
MS DOS, *Win16*					
Tiny	near		один, <= *64Кбайт*		Код та дані об'єднані в одну групу з іменем GROUP. Використовується для створення програм формату .com
Small	near	near	один, <=64 *Кбайт*	один, <=64 *Кбайт*	Код займає один сегмент, дані об'єднані в одну групу з іменем GROUP. Цю модель використовують для більшості програм на *Assembler*.
Medium	far	near	декілька, <=64 *Кбайт*	один, <=64 *Кбайт*	Код займає декілька сегментів по одному на кожний об'єднуваний програмний модуль. Усі посилання на передачу управління – типу far. Дані об'єднані в одній групі; всі посилання на них – типу near.
Com-pact	near	far	один, <=64 *Кбайт*	декілька, <=64 *Кбайт*	Код в одному сегменті; посилання на дані – типу near
Large	far	far	декілька, <=64 *Кбайт*	декілька, <=64 *Кбайт*	Код у декількох сегментах, по одному на кожний програмний модуль, що об'єднується
Win 32 (починаючи з i386)					
Flat	flat	flat	Обмежено можливостями процесора та системної плати		Код і дані в одному 32-бітном сегменті (плоска модель пам'яті).

Модель Tiny використовується для створення виконуваної програми у вигляді com-файлу. У цій моделі є тільки сегмент коду. Дані зберігаються всередині останнього. Решта моделей використовується для отримання exe-файлу. Модель Small складається із сегмента коду та сегмента

даних. Найбільш поширена модель Large. Може працювати з декількома сегментами коду та даних, але кожен із них не повинен перевищувати 64 *Кбайт*. Це здійснюється за допомогою організації дальніх (far) викликів.

При використанні директиви Model транслятор дає доступ до декількох наперед визначених ідентифікаторів, до яких можна звернутися під час виконання програми з метою отримання інформації про ті чи інші параметри вибраної моделі пам'яті. Деякі з них для транслятора TASM наведено в табл. 5.4.

Таблиця 5.4

Зарезервовані ідентифікатори

Ім'я ідентифікатора	Значення змінної
@code	Фізична адреса сегмента коду
@data	Фізична адреса сегмента даних типу near
@fardata	Фізична адреса сегмента даних типу far
@fardata?	Фізична адреса сегмента неініціалізованих даних типу far
@stack	Фізична адреса сегмента стеку

Таблиця 5.5

Спрощені директиви визначення сегмента

Формат директиви	Призначення
.CODE [ім'я]	Початок або продовження сегмента коду
.DATA	Початок або продовження сегмента даних. Використовується також для опису даних типу near 1.
.CONST	Початок або продовження сегмента постійних даних (констант) модуля
.DATA?	Початок або продовження сегмента неініціалізованих даних. Використовується також для опису даних типу near
.STACK [розмір]	Початок або продовження сегмента стеку модуля. Параметр [розмір] задає розмір стеку

5.2. ПОРЯДОК РОЗРОБКИ ПРОГРАМИ НА ASSEMBLER. СТРУКТУРА LST ФАЙЛА В TASM

Загальну схему процесу розробки програми на *Assembler* надано на рис. 5.1. На схемі виділено чотири етапи цього процесу. На першому етапі, коли вводиться код програми, можна використовувати будь-який текстовий редактор. У *Windows* таким редактором може бути Блокнот. Основною вимогою до текстового редактора є те, щоб він не вставляв сторонніх символів (спеціальних символів форматування). Файл повинен мати розширення .asm.

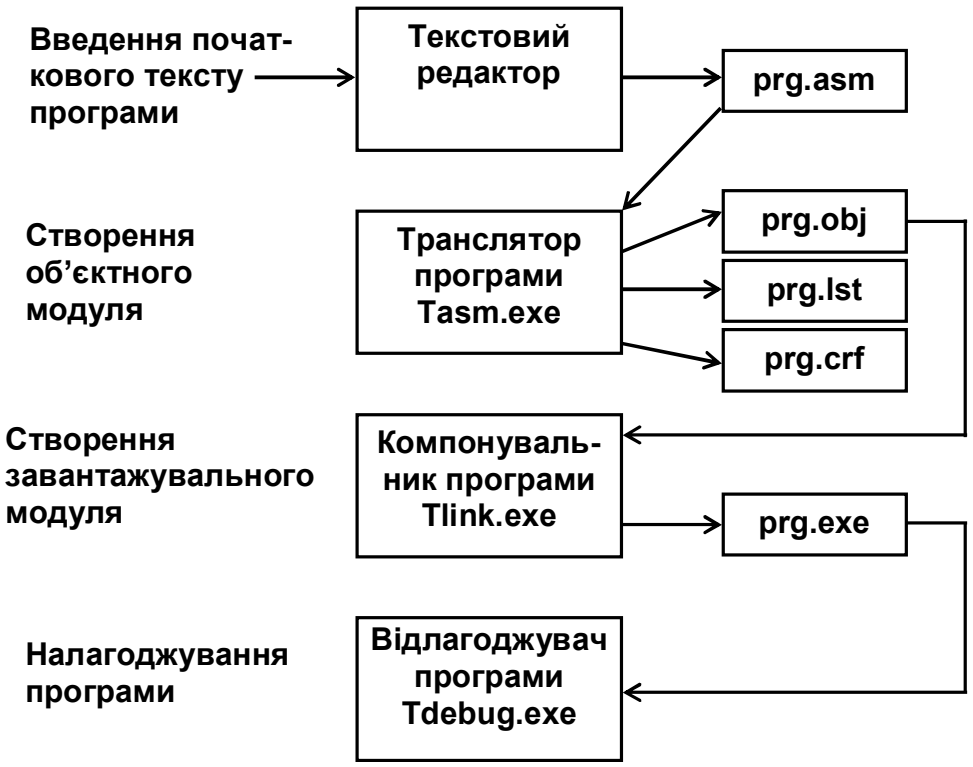

Рис. 5.1. Загальна схема процесу розробки програми на *Assembler*

Для виконання наступних етапів розробки необхідно мати програмні засоби із пакетів MASM (*Macroassembler* фірми *Microsoft*) або TASM (*Turbo Assembler фірми Borland*). У навчальному посібнику викладаються засоби обох пакетів, однак, здебільшого на прикладі TASM, оскільки процес розробки програм *Assembler* з використанням цього пакета більш наочний. Усі пакети *Assembler* виконують майже одну роботу, однак по-різному. Зрозумівши суть перетворювань вихідної програми, які виконуються пакетом TASM, засвоїти інші пакети *Assembler* стає набагато простішим.

Трансляція програми. На етапі трансляції формується об'єктний модуль, який є початковою програмою в машинних кодах і деякою іншою інформацією, що необхідна для налагоджування й компонування його з іншими модулями. Для отримання об'єктного модуля початковий файл необхідно піддати трансляції за допомогою програми tasm.exe із пакета TASM. Формат командного рядка для запуску tasm.exe такий:

tasm *[опції] ім'я початкового файлу, [ім'я об'єктного файлу], [ім'я файлу лістингу], [ім'я файлу перехресних зв'язків].*

Інформація, яка міститься у квадратних дужках є необов'язковою й може бути відсутньою. Якщо ці файли не задати, то вони не будуть створені.

Якщо імена об’єктного файлу, файлу лістингу та файлу перехресних зв’язків повинні збігатися з ім’ям початкового файлу, то необхідно поставити коми замість імен цих файлів

tasm prg, , ,

Внаслідок цього будуть утворені файли

prg. obj
prg. lst
prg. crf

Якщо необхідне вибіркове створення файлів, то замість непотрібних файлів необхідно підставити параметр nul. Наприклад,

tasm.exe prg, , nul,

Внаслідок цього будуть утворені файли

prg. obj
prg. crf

Опція /zi дозволяє включити до складу файлу лістингу (*.lst) всю інформацію транслятора.

Компонування програми. Компонування програми здійснюється для об’єднання кількох модулів, які написані як однією, так і різними мовами. При цьому у функції компонувальника входить також розв’язування зовнішніх посилань у цих модулях. Результатом роботи компонувальника є створення завантажувального файлу з розширеннями .exe. Після цього операційна система може завантажувати такий файл у пам’ять і виконувати його.

Повний формат командного рядка для запуску компонувальника такий:

tlink *[опції] список об’єктних файлів, [ім’я завантажувального модуля], [ім’я файлу карти], [ім’я файлу бібліотеки], [ім’я файлу визначень], [ім’я файлу ресурсів].*

Тут - опції – необов’язковий параметр, який управляє роботою компонувальника;

- список об’єктних файлів – обов’язковий параметр, який містить список файлів із розширенням .obj, що компонуються. Файли повинні бути розділені пробілом або знаком «+». Наприклад,

tlink /v prg + prg1 + prg2.

За необхідності імена файлів можуть містити шлях до них:

- *ім'я завантажувального модуля* – необов'язковий параметр, який визначає ім'я завантажувального модуля. Якщо воно не вказане, то ім'я завантажувального модуля буде збігатися з першим іменем зі списка об'єктних файлів;

- *ім'я файлу карти* – необов'язковий параметр, наявність якого зобов'язує компонувальника створити файл із картою завантажування, у якій перераховуються імена, адреси завантажування та розміри всіх сегментів, які входять до складу програми;

- *ім'я файлу бібліотеки* – необов'язковий параметр, який є шляхом до файлу бібліотеки (.lib). Цей файл створюється й обслуговується спеціальною утилітою tlib.exe пакета TASM. Утиліта дозволяє об'єднувати часто використовувані підпрограми у вигляді об'єктних модулів в один файл.

Відлагоджування програми. Для програм TASM реального режиму використовується 16-розрядний відлагоджувач *Turbo Debugger* (TD), розроблений фірмою *Borland International*. Це найбільш вдалий відлагоджувач для програм *Assembler* реального режиму.

Відлагоджувач TD являє собою віконне середовище відлагоджування програм на рівні початкового тексту *Assembler*. Він дає змогу розв'язати дві головні задачі:

- визначити місце логічної помилки;

- визначити причину логічної помилки.

Можливості TD:

- послідовне виконання (трасування) програми, при якому за один крок виконується одна машинна команда;

- трасування програми у зворотному напрямку;

- перегляд та зміна стану апаратних ресурсів процесора під час трасування.

TD не дозволяє вносити зміни до початкового тексту програми, дозволяє виконувати трасування програми в прямому та зворотному напрямках, а також переглядати та змінювати стан апаратних ресурсів мікропроцесора під час покомандного виконання програми. Структуру *lst*-файлу в TASM надано в табл. 5.6.

Таблиця 5.6

Структура *lst*-файлу в TASM

Turbo Assembler *Version 3.2* 09/04/09 20:36:01 *Page* 1 ім'я файлу.ASM Title-інформація			
1	2	3	4
Номер рядка програми	Адреса (зміщення)	Машинний код	Програма на *Assembler* – початковий код
Turbo Assembler *Version 3.2* 09/04/09 20:36:01 *Page* 2 *Symbol Table* Таблиця розподілу пам'яті			
Groups & Segments Bit Size Align Combine Class Інформація про сегменти та їх довжину			

Поля 2 та 3 містять HEX-коди. Поле 3 (машинний код) – це і є результат компіляції. Ці коди можна побачити, якщо переглянути об'єктний або виконуваний (exe) файл через переглядач.

Лістинг програми priklad.asm, текст якої наведено вище, має такий вигляд:

```
Turbo Assembler      Version 3.2      09/11/09 16:49:20   Page 1
priklad.ASM
    1                                   ;загальна структура програми
    2                       masm
    3  0000                 Model small      ;директива моделі пам'яті
    4  0000                 .stack 256h      ;опис сегмента стеку
    5  0000                 .Data            ;опис сегмента даних
    6  0000  05                  a db 5
    7  0001  FE                  b db -2
    8  0002  000A                c dw 10
    9  0004  FFFC                d dw -4
   10  0006                 .Code            ;опис сегмента коду
   11  0000                 start:
   12  0000  B8 0000s            mov ax, @data
   13  0003  8E D8               mov ds, ax
   14                                   ;далі програмний код
   15
   16
   17                       ;- передача управління операційній системі
   18  0005                      exit:
   19  0005  B8 4C00             mov ax, 4c00h
```

```
20  0008  CD 21            int 21h
21                        end start

Turbo Assembler  Version 3.2    09/11/09 16:49:20        Page 2
Symbol Table
priklad.asm
Symbol Name             Type   Value              Cref       (defined at #)
??DATE                  Text   "09/11/09"
??FILENAME              Text   "priklad    "
??TIME                  Text   "16:49:20"
??VERSION               Number 0314
@32BIT                  Text   0                  #3
@CODE                   Text   _TEXT              #3  #3  #10
@CODESIZE               Text   0                  #3
@CPU                Text   0101H
@CURSEG                 Text   _TEXT              #5  #10
@DATA                   Text   DGROUP                 #3  12
@DATASIZE               Text   0                  #3
@FILENAME               Text   PRIKLAD
@INTERFACE              Text   00H                #3
@MODEL                  Text   2                  #3
@STACK                  Text   DGROUP                 #3
@WORDSIZE               Text   2                  #5  #10
A                   Byte   DGROUP:0000            #6
B                   Byte   DGROUP:0001            #7
C                   Word   DGROUP:0002       #8
D                   Word   DGROUP:0004       #9
EXIT                    Near   _TEXT:0005          #18
START                       Near   _TEXT:0000     #11       21

Groups & Segments       Bit Size Align Combine Class  Cref (defined at #)
DGROUP    Group                                 #3  3  12
STACK       16  0256 Para   Stack       STACK       #4
_DATA       16  0006 Word  Public       DATA        #3  #5
_TEXT 16  000A Word  Public       CODE       #33 #10 10
```

Правильна організація процесу отримання виконуваного модуля дозволяє здійснити відлагоджування на рівні вихідного тексту. Основні моменти процесу трансляції та компонування:

1. У початковій програмі обов’язково повинна бути визначена мітка для першої команди, з якої починається виконання програми. Ім’я

цієї мітки обов'язково повинне бути вказане в кінці програми як операнд директиви END:

end *ім'я_мітки*

2. Початковий модуль повинен бути відтрансльований із ключем /zi

tasm /zi *ім'я_початкового_модуля*, , ,

Ключ /zi дозволяє транслятору зберегти зв'язок символічних імен у програмі з їх зміщеннями в сегменті коду, що дає змогу відлагоджувачу виконати відлагоджування, використовуючи оригінальні імена.

3. Комнонування програмного модуля повинне бути з ключем /v:

tlink /v *ім'я_об'єктного_модуля*

Ключ /v вказує на необхідність збереження відлагоджувальної інформації у виконуваному файлі.

4. Запуск відлагоджувача із командного рядка з указанням виконуваного модуля

td *ім'я_виконуваного_модуля*

При правильному виконанні вказаних дій відкриється вікно відлагоджувача з іменем Module, у якому буде розміщено вихідний текст програми.

Курсор виконання (вигляд трикутника) вказує на першу команду, яка буде виконуватися.

Управління відлагоджувачем здійснюється за допомогою системи меню.

Головне меню розташоване у верхній частині екрана й доступне постійно. Виклик меню – клавіша *<F10>*.

Контекстне меню – для кожного вікна відлагоджувача можна викликати його власне меню, яке враховує особливості цього вікна, натиснувши у вікні правою кнопкою миші або комбінацією клавіш *<Alt+F10>*.

Запустити програму у відлагоджувачі можна одним із чотирьох способів:

- безумовного виконання;
- покрокового виконання;
- виконання до поточного положення курсора;
- виконання зі встановленням точок переривання.

Режим безумовного виконання програми можна застосовувати тоді, коли необхідно переглянути загальну поведінку програми. Для запус-

ку програми в цьому режимі необхідно натиснути клавішу <*F9*>. У точках, де необхідно ввести будь-які дані, відлагоджувач, відповідно до логіки роботи засобу введення, буде здійснювати відповідні дії. Аналогічні дії відлагоджувач виконає також при виведенні даних. Для перегляду або введення цієї інформації можна відкрити вікно користувача, вибравши у меню команду Windows → User screen або натиснувши клавіші <*Alt+F5*>.

Режим покрокового виконання програми використовується для детального вивчення її роботи. У цьому режимі виконання програми переривається на кожній машинній команді. При цьому є можливість спостереження за результатом виконання команд. Для активізації цього режиму необхідно натиснути клавішу <*F7*> (Run → Trace into) або <*F8*> (Run → Step over). У цьому режимі корисно використовувати вікно CPU, яке викликається командою View→CPU. Вікно CPU містить інформацію про стан процесора та складається з п'яти вікон:

Registers
Flags
Stack
Dump

Отже, вікно CPU відбиває видиму частину програмної моделі процесора.

Зупинка виконання програми в будь-якому режимі відбувається натисканням клавіш <*Ctrl+F2*>.

Особливості розробки програм у MASM. Для успішної роботи MASM у сучасних операційних середовищах необхідно мати версію 6.13 цього пакета або вище. До його складу входять такі основні програми:

masm.exe – *Assembler*;
ml.exe – *Assembler* та компонувальник (*Masm and Link*);
link.exe – компонувальник;
cv.exe – відлагоджувач (*Code View*);
lib.exe, implib.exe, nmake.exe, cref.exe, h2inc.exe, exehdr.exe, cvpack.exe, helpmake.exe, rm.exe, undel.exe, exp.exe – допоміжні утиліти.

За допомогою пакета MASM розробка програм виконується традиційним для програмування *Assembler* способом – запуском окремих програм трансляції, компонування та налагоджування. Для цього використовуються програми masm.exe та ml.exe, link.exe та cv.exe.

Необхідно зазначити, що трансляція програми може здійснюватись двома програмами: masm.exe та ml.exe. Різниця між ними полягає в тому, що програма ml.exe крім трансляції файлу викликає компонувальник link.exe. При використанні програми masm.exe запуск компонувальника link.exe здійснюється окремо.

Командний рядок ml.exe має вигляд:

ml [*ключі*] файл_1 [[*ключі*] файл_2] … [/link *ключі_link*]

Ключі командного рядка чутливі до регістра.

Командний рядок masm має вигляд:

masm [*ключі*] файл [, [*об'єктний_файл*] [, *файл_лістингу*] [, [файл_перехресних_зв'язків]]]]

Компонувальник компонує (об'єднує) об'єктні файли та бібліотеки у виконуваний файл або динамічно компоновану бібліотеку (DLL). Командний рядок link.exe має такий вигляд:

link [*ключі*] об'єкт_файли [, [*виконуваний_файл*] [, *файл_карти*] [, [*файли_бібліотек*] [, [*def_файл*]]]]] [;]

Виконувані файли пакета MASM розміщені у двох каталогах: ..\BIN та ..\BINR. Для зручності роботи рекомендується їх розмістити в одному каталозі, наприклад \WORK.

Приклад командного рядка для отримання виконуваного файлу може бути таким:

ml.exe /Zi /Fl proga.asm

Якщо синтаксичні помилки відсутні, то можна запустити налагоджувач:

cv.exe proga.exe

5.3. КОМАНДИ ПЕРЕМІЩЕННЯ ДАНИХ

Переміщення даних. До цієї групи належать такі команди:

mov *<операнд-приймач>, <операнд-джерело>*

xchg *<операнд 1>, <операнд 2>*

mov – це основна команда переміщення даних.

Особливості команди mov:

- командою mov заборонено переміщувати дані з однієї ділянки пам’яті до іншої. Якщо така необхідність виникає, то необхідно використовувати будь-який доступний регістр як проміжну ланку;

- не слід завантажувати в сегментний регістр значення безпосередньо з пам’яті. Для такого завантаження потрібен проміжний об’єкт, це може бути регістр або стек;

- не слід переміщувати вміст одного сегментного регістра в інший сегментний регістр. Для такого переміщення потрібен проміжний об’єкт;

- не слід використовувати сегментний регістр CS як операнд призначення.

Операнди в команді можуть задаватися так:

1. Операнд задається неявно на мікропрограмному рівні.
2. Операнд задається в самій команді (безпосередній операнд).
3. Операнд міститься в одному з регістрів.
4. Операнд міститься в пам’яті.
5. Операнд є портом введення-виведення.
6. Операнд знаходиться в стеку.

У двоопераднійй машинній команді можливі такі комбінації операндів:

- регістр-регістр;
- регістр-пам’ять;
- пам’ять-регістр;
- безпосередній операнд-регістр;
- безпосередній операнд-пам’ять.

Приклад перенесення значення з регістра ds до регістра es:

```
mov  ax, ds
mov  es, ax
```

Для двостороннього обміну даними застосовується команда xchg, наприклад

```
xchg ax, bx  ; обмін вмістом регістрів ax та bx.
```

Не допускається обмінювати між собою вміст двох комірок пам’яті.

5.4. ОПЕРАЦІЇ НАД ЦІЛИМИ ЧИСЛАМИ

Додавання двійкових чисел без знака. Найпростіша команда add (addition – додавання). Синтаксис

add <*операнд_1*>, <*операнд_2*>.

Логіка роботи

<операнд_1>=<операнд_1> + <операнд_2>.

Команда інкремента, тобто збільшення значення операнда на 1. Синтаксис

inc *операнд.*

Логіка роботи команди

<операнд> = <операнд> + 1.

Команда додавання adc з урахуванням прапорця переносу cf. Синтаксис

adc *<операнд_1>, <операнд_2>.*

Логіка роботи

<операнд_1>=<операнд_1>+<операнд_2>+значення_cf.

Отже, команда adc є засобом процесора для додавання довгих двійкових чисел, розмірність яких перевищує розміри стандартних полів, які підтримуються процесором.

Команди віднімання двійкових чисел без знака. Команда віднімання **sub** (subtract – віднімання). Синтаксис

sub *<операнд 1>, <операнд 2>.*

Логіка роботи

<операнд_1>=<операнд_1>-<операнд_2>.

Команда декремента dec. Синтаксис

dec *операнд.*

Логіка роботи

<операнд> = <операнд> – 1.

Команда віднімання з урахуванням позики, тобто прапорця cf

sbb *<операнд 1>, <операнд 2>.*

Логіка роботи

<операнд_1>=<операнд_1>–<операнд_2>–значення_cf.

Множення цілих чисел без знака. Для множення цілих чисел без знака призначена команда **mul** (multiply – беззнакове множення). Синтаксис

mul *множник.*

Логіка роботи команд:

<добуток> = <множене><множник>.*

Джерелом є множник. Множником можуть бути регістри й пам'ять. Константи не можуть бути множниками.

Множене та добуток за замовчуванням знаходяться у визначеному місці залежно від довжини множника (табл. 5.7).

Таблиця 5.7

Неявні операнди команд mul та imul

Довжина джерела (множник)	Множене (те, що множиться)	Добуток (результат)
byte	al	16 бітів в ax (ah – старша частина результату, al – молодша частина результату>)
word	ax	32 біти в парі dx:ax (dx – старша частина результату, ax – молодша частина результату)
doubl word	eax	64 біти в парі edx:eax (edx – старша частина результату, eax – молодша частина результату)

Якщо результат множення умістився в одному регістрі, тобто старша частина нульова, то після завершення множення cf = 0 та of = 0. Якщо прапорці cf та of ненульові, то результат вийшов за межі молодшої частини й складається з двох частин, що потрібно враховувати при подальшій роботі.

Множення цілих чисел зі знаком. Для множення чисел зі знаком призначена команда

imul *множник_1 (множник_2, множник_3)*

Ця команда виконується так само, як і команда mul. Відмінною рисою є формування знака. Якщо результат множення малий і умістився в одному регістрі (тобто cf = 0 та of = 0), то вміст іншого регістра (старшої частини) є розширенням знака – усі його біти дорівнюють старшому біту (знаковому розряду) молодшої частини результату. В іншому випадку (якщо cf = 1 та of = 1) знаком результату є знаковий біт старшої частини результату, а знаковий біт молодшої частини є значущим бітом двійкового коду результату.

Ділення двійкових чисел без знака. Для ділення чисел без знака призначена команда **div** (divide – беззнакове ділення). Синтаксис

div *дільник.*

Логіка роботи команд:

<частка:залишок>=<ділене>/<дільник-джерело>.

Дільник може розміщуватись у регістрі або пам'яті та мати розмір 8, 16 або 32 біти. Константи не можуть бути дільниками. Місце діленого фіксоване й залежить від розміру дільника. Результатом роботи команди є частка та остача, які знаходяться у визначеному місці залежно від довжини дільника (табл. 5.8).

Таблиця 5.8

Неявні операнди команд div та idiv

Ділене	Дільник	Частка	Остача
Слово (16 бітів) у регістрі ax	Байт у регістрі або комірці пам'яті	Байт в регістрі al	Байт в регістрі ah
Подвійне слово (32 біти), в dx – старша частина, в ax – молодша частина	Слово в регістрі або комірці пам'яті	Слово в регістрі ax	Слово в регістрі dx
Почетверене слово (64 біти), в edx – старша частина, в eax – молодша частина	Подвійне слово в регістрі або комірці пам'яті	Подвійне слово в регістрі eax	Подвійне слово в регістрі edx

Ділення двійкових чисел зі знаком. Для ділення чисел зі знаком призначена команда idiv (integer divide – знакове ділення цілих чисел):

idiv *дільник.*

Для цієї команди справедливими є всі розглянуті раніше принципи.

Остача завжди має знак діленого. Знак частки залежить від стану знакових бітів (старших розрядів) діленого та дільника.

Фрагмент програмного коду, який реалізує описані операції наведено нижче.

```
mov ax, a   ;значення a розміщується в регістрі ax
mov bx, ax  ;вміст регістра ax копіюється до регістра bx
add bx, 5   ;до вмісту регістра bx додається 5 (bx=bx+5)
sub bx, 2   ;від вмісту регістра bx віднімається 25 (bx=bx–2)
mov b, bx   ;у комірку пам'яті b розміщується  вміст регістра bx
```

Допоміжні команди для арифметичних розрахунків. Команди перетворення типів. Відомо, що в арифметичних операціях операнди повинні бути однакового формату. Однак на практиці трапляються

випадки, коли операнди, над якими виконуються дії, мають різний формат. Для узгодження різних форматів існують так звані команди перетворення типів. Ці команди розширюють байти в слова, слова в подвійні слова, подвійні слова в почетверені слова. Існують два види команд перетворення типів.

Команди без операндів. Ці команди працюють із фіксованими (визначеними за замовчуванням) регістрами.

CBW (*Convert Byte to Word*) – команда перетворення байта в регістрі al в слово в регістрі ax шляхом розміщення старшого біта al у всіх розрядах регістра ah.

CWD (*Convert Word to Double*) – команда перетворення слова в регістрі AX в подвійне слово в регістрах dx:ax шляхом розміщення значення старшого біта AX у всіх розрядах регістра dx.

CWDE (*Convert Word to Double*) – команда перетворення слова в регістрі ax в подвійне слово в регістрі eax шляхом розміщення значення старшого біта ax у всіх розрядах старшої половини регістра eax.

CDQ (*Convert Double Word to Quarter Word*) – команда перетворення подвійного слова в регістрі EAX в почетверене слово в регістрах edx:eax шляхом розміщення значення старшого біта ax у всіх розрядах регістра edx.

Команди обробки рядків. Цим командам притаманні корисні властивості:

movsx *операнд_1, операнд_2* – команда переміщення з розповсюдженням знака розширює 8- або 16-розрядне значення *операнд_2*, яке може бути регістром або операндом у пам'яті, до 16- або 32-розрядного значення в одному з регістрів, використовуючи знаковий біт для заповнення старших розрядів в *операнд_1*. Дану команду можна використовувати для підготовки операндів зі знаками до виконання арифметичних операцій;

movzx *операнд_1, операнд_2* – команда переміщення з розповсюдженням нуля розширює 8- або 16-розрядне значення *операнд_2*, до 16- або 32-розрядного шляхом заповнення нулями (обнуленням) старших розрядів значення *операнд_1*. Дану команду можна використовувати для підготовки операндів без знака до виконання арифметичних операцій.

Приклад. Розрахувати значення y = (a+b)/c де a, b, c – байтові знакові змінні.

```
masm
model small
stack 256h
.data
a db 5
b db 10
c db 2
y dw ?
.code
start:                    ;точка входу в програму
;……
mov al, a
cbw
.386
movsx   bx, b
add  ax, bx
idiv  c                   ;в al – частка,  в ah – остача
exit:
mov ax, 4c00h   ;стандартний вихід
int 21h
end start
```

У цій програмі ділене для команди idiv готується завчасно.

Інші корисні команди. У системі команд процесора є дві команди – xadd та neg, які можуть бути корисними для програмування розрахунків.

xadd *приймач-джерело* – обмін місцями та додавання. Команда виконує такі дії:

1) міняє місцями значення *приймач* та *джерело*;
2) розміщує на місті операнда *приймач* суму:

$$приймач = приймач + джерело.$$

neg *операнд* – заперечення з доповненням до двох. Команда виконує інверсію значення *операнд*. Фактично команда виконує одну дію:

$$операнд = 0 - операнд,$$

тобто віднімає операнд від нуля.

Команду можна застосовувати для зміни знака та віднімання із константи.

Команди SUB та SBB не дозволяють здійснювати віднімання від константи, через те, що константа не може бути операндом-приймачем у цих операціях. Тому дану операцію можна виконати за допомогою двох команд:

```
neg  ax   ;зміна знака (ax)
……..
add ax, 340   ; розрахунок (ax)=340-(ax)
```

5.5. АДРЕСАЦІЯ ДАНИХ У ПАМ'ЯТІ

5.5.1. Пряма адресація

При прямій адресації ефективна адреса міститься в самій команді і для її формування не використовується ніяких додаткових джерел або регістрів. Ефективна адреса береться безпосередньо з поля зміщення машинної команди, яке може мати розмір 8, 16, 32 *біт*. Це значення однозначно визначає байт, слово або подвійне слово, яке розміщене в сегменті даних.

Пряма адресація буває двох типів: *відносна пряма* та *абсолютна пряма* адресація.

Відносна пряма адресація використовується для команд умовних та безумовних переходів, для вказання відносної адреси переходу. Відносність такого переходу полягає в тому, що в полі зміщення машинної команди міститься 8-, 16- або 32-бітне значення, яке внаслідок роботи команди буде додаватися до вмісту регістра покажчика команд ip/eip. Внаслідок такого додавання буде отримано адресу, за якою і здійснюється перехід.

Абсолютна пряма адресація. У цьому випадку адреса є частиною машинної команди, однак формується ця адреса тільки зі значення поля зміщення в команді. Для формування фізичної адреси операнда в пам'яті мікропроцесор підсумовує це поле зі зсунутим на 4 біти значенням сегментного регістра.

У команді *Assembler* можна використовувати декілька форм такої адресації. Наприклад:

```
mov ax, dword ptr [0000] ;записати слово за адресою ds:0000 в ax
```

Така адресація використовується рідко. Решта видів адресації належить до непрямих. Слово "непрямий" означає, що в самій команді розміщується лише частина ефективної адреси, а решта її компонентів знаходиться в регістрах, які вказують своїм вмістом байт modr/m та, можливо, байт sib.

5.5.2. Команди передачі управління (команди умовних та безумовних переходів)

У загальному випадку в програмі є місця, у яких потрібно прийняти рішення про те, яка команда буде виконуватись наступною. Це рішення може бути:

– безумовним: із визначеного місця необхідно передати управління не тій команді, яка йде наступною, а іншій, яка розміщується на деякому віддаленні від поточної команди;

– умовним: рішення про те, яка команда буде виконуватись наступною, приймається за результатами аналізу деяких умов або даних.

Те, яка команда програми повинна виконуватись наступною, мікропроцесор визначає за вмістом пари регістрів cs:ip (cs:eip) у якій:

cs – сегментний регістр коду, у якому розміщується фізична (базова) адреса поточного сегмента коду;

eip/ip – регістр покажчика команд, у якому міститься значення, яке являє собою зміщення в пам'яті наступної команди (яка буде виконуватись) відносно початку поточного сегмента коду.

Отже, команди передачі управління змінюють вміст регістрів cs і eip, внаслідок чого процесор вибирає для виконання не наступну за порядком команду програми, а команду в деякій іншій ділянці програми. Конвеєр всередині процесора при цьому обнуляється (скидається).

За принципом дії команди мікропроцесора, які забезпечують організацію переходів у програмі, можна поділити на три групи:

1. Команди безумовної передачі управління:

– команди безумовного переходу;

– виклик процедур та повернення з процедур;

– виклик програмних переривань та повернення з програмних переривань.

2. Команди умовної передачі управління:

– команди переходу за результатом виконання команди порівняння;

– команди переходу за станом визначеного прапорця;

– команди переходу за вмістом регістра ecx/cx.

3. Команди управління циклом:

– команда організації циклу з лічильником ecx/cx;

– команда організації циклу з лічильником ecx/cx із можливістю дострокового виходу з циклу за додаткової умови.

Мітки. Місце, у яке необхідно здійснити перехід, визначається за допомогою міток. *Мітка* – це символічне ім'я, яке позначає визначену комірку пам'яті, що призначена для використання як операнда в командах передачі управління.

Як і для змінної транслятор *Assembler* привласнює мітці три атрибути:

– ім'я сегмента коду, де ця мітка описана;

– зміщення – відстань у байтах від початку сегмента коду, у якому описана мітка;

– тип мітки або атрибут відстані.

Атрибут відстані може набувати двох значень:

– near – перехід на цю мітку можливий лише в межах сегмента коду, де ця мітка описана. Фізично це означає, що для переходу на мітку достатньо змінити тільки вміст регістра eip/ip;

– far – перехід на цю мітку можливий тільки внаслідок міжсегментної передачі управління, для здійснення якої потрібна зміна вмісту як регістра eip/ip, так і регістра cs.

Мітку можна визначити двома способами:

– оператором **:** (двокрапка);

– директивою label.

Синтаксис першого способу показано на рис. 5.2.

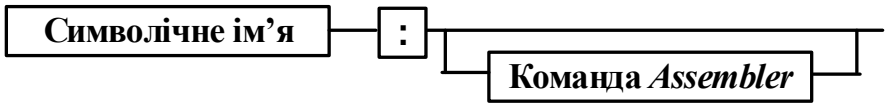

Рис. 5.2. Синтаксис визначення мітки

За допомогою цього способу можна визначити тільки мітку ближнього типу – near. Команда *Assembler* може розташовуватись як в одному рядку з міткою, так і на наступному рядку. Таку мітку можна використовувати як операнд в командах умовного (jcc) та безумовного (jmp, call) переходів, наприклад:

```
m1: mov ax, bx
або
m1:
mov ax, z
```

Другий спосіб визначення міток набуває значень near або far (рис. 5.3).

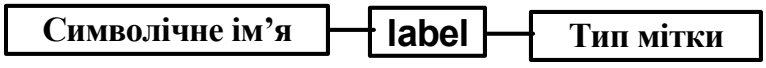

Рис. 5.3. Визначення мітки

Взагалі директиву label використовують для визначення ідентифікатора заданого типу, наприклад:

```
m1 label near
mov ax, z
```

Якщо необхідно використати для тієї самої команди мітку і дальнього, і ближнього типів, то в цьому випадку необхідно визначити дві мітки, при цьому мітку дальнього типу необхідно описати директивою label, як показано у фрагменті:

```
……..
public m_far ;зробити мітку m_far видимою для зовнішніх програм
……..
m_far label far      ;визначення мітки m_far дальнього типу
m_near : mov ax, z
…….
```

Визначивши для команди mov ax, z дві мітки, можна організувати перехід до цієї команди як з даного сегмента команд, так і з інших сегментів команд, зокрема і тих, що належать іншим модулям. Для того, щоб ім'я мітки m_far зробити видимим зовні, застосовується директива public.

Безумовні переходи. Команди переходу модифікують регістр покажчика команд eip/ip і, можливо, сегментний регістр коду cs. Що саме повинно модифікуватись, визначається:

– типом операнда в команді безумовного переходу (ближній або дальній);

– покажчиком у команді переходу (модифікатором) перед адресою переходу – при цьому сама адреса переходу може знаходитись безпосередньо в команді (прямий перехід) або в регістрі чи комірці пам'яті (непрямий перехід).

Модифікатор може набувати таких значень:

near ptr – прямий перехід на мітку всередині поточного сегмента коду. Модифікується тільки регістр eip/ip на підставі вказаних у команді адреси (мітки) або виразу, який використовує символ **$** визначення значення лічильника адреси команд – **$**;

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

word ptr – непрямий перехід на мітку всередині поточного сегмента коду. Модифікується (значенням зміщення з програми за вказаною в команді адресою або з регістра) тільки eip/ip. Розмір зміщення – 16 або 32 біти;

dword ptr – непрямий перехід на мітку в іншому сегменті коду. Модифікуються значенням із пам'яті (з регістра неможливо) обидва регістра cs та eip/ip. Перше слово (подвійне слово) цієї адреси являє собою зміщення й завантажується в eip/ip, друге (третє) слово завантажується в cs.

Синтаксис команди безумовного переходу без збереження інформації про місце повернення:

jmp [модифікатор] адреса переходу ; *безумовний перехід*

Адреса переходу являє собою адресу у вигляді мітки або адресу області пам'яті, у якій знаходиться покажчик переходу.

У системі команд процесора є декілька кодів машинних команд безумовного переходу jmp. Їх різниця визначається дальністю переходу і способом задавання цільової адреси. Дальність переходу визначається місцеположенням операнда "адреса переходу". Ця адреса може розміщуватись у поточному сегменті коду або деякому іншому сегменті. У першому випадку перехід називається ***внутрішньосегментним***, у другому – ***міжсегментним*** або дальнім.

Виділяють три варіанти внутрішньосегментного використання команди jmp:

- прямий короткий;
- прямий;
- непрямий.

Прямий короткий внутрішньосегментний перехід застосовується тоді, коли відстань від команди jmp до адреси переходу не перевищує –128 або +127 байтів. Якщо адреса переходу розміщена до команди jmp, то *Assembler* формує коротку команду безумовного переходу без додаткових вказівок:

```
…….
m1:
……. (не більше 35–40 команд (128 байт))
jmp m1
…….
```

Якщо адреса переходу розміщена після команди jmp, то транслятор не може сам визначити, що перехід короткий через те, що у ньому ще

немає інформації про адресу переходу. Для надання компілятору допомоги у формуванні короткого безумовного переходу додатково використовують модифікатор short ptr, наприклад:

```
…….
jmp short ptr m1
……. (не більше 35–40 команд (127 байт))
m1:
……..
```

Прямий внутрішньосегментний перехід дає змогу виконувати переходи в межах 64 *Кбайт* відносно наступної за jmp команди. Відрізняється від прямого короткого внутрішньосегментного переходу тим, що довжина машинної команди jmp в цьому випадку дорівнює 3 *байт*, наприклад:

```
…….
m1:
……. (відстань більше 128 байт та менша 64 Кбайт)
jmp m1
…….
```

Непрямий внутрішньосегментний перехід передбачає “опосередкованість” (рус. “косвенность”) задавання адреси переходу. Це означає, що в команді вказується не сама адреса переходу, а місце, де вона міститься. Приклад розміщення двобайтової адреси в регістрах:

```
lea bx, m1
jmp bx    ; адреса переходу в bx
……..
m1:
…….
```

Міжсегментний перехід передбачає інший формат команди jmp. Такий перехід підтримує два варіанти команд безумовного переходу: прямий та непрямий.

Команда прямого міжсегментного переходу має довжину 5 *байт*, серед яких два байти складають значення зміщення і два байти – значення сегментної складової адреси, наприклад:

```
seg_1 segment
……
jmp far ptr m2 ; тут far обов’язкове
……
```

```
m1 label far
……
seg_1 ends
seg_2 segment
……
m2 label far
jmp m1 ; (тут far необов'язкове)
……
```

Команда непрямого міжсегментного переходу як операнд використовує адресу області пам'яті, у якій міститься зміщення й сегментна частина цільової адреси переходу.

```
data        segment
addr_m1  dd    m1 ; в поле addr_m1 значення зміщення і адреси
;сегмента мітки m1
data        ends
code_1    segment
……..
    jmp    m1
…….
code_1    ends
code_2    segment
……
m1 label far
    mov    ax, bx
……
code_2    ends
```

Отже, модифікатори short ptr, near ptr та word ptr використовуються для організації переходів всередині сегмента, а far ptr та dword ptr – для міжсегментних переходів.

Команда порівняння (cmp). Принцип роботи команди **cmp** (*compare*) такий самий, як і команди віднімання sub *операнд_1*, *операнд_2*. Команда cmp здійснює віднімання операндів і встановлює відповідні прапорці. Однак на відміну від команди sub, команда cmp не записує результат віднімання на місце першого операнда. Синтаксис команди:

cmp *операнд_1, операнд_2*

Прапорці, які встановлюються командою cmp, можна аналізувати спеціальними командами переходу. Команди умовного переходу jcc.

Таблиця 5.9

Значення абревіатур cc в команді jcc

Мнемонічне позначення	Оригінальний термін	Переклад	Тип операндів
e	Equal	дорівнює	будь-які
n	not	ні	будь-які
g	greater	більше	числа зі знаком
L	less	менше	числа зі знаком
a	above	вище, у розумінні “більше”	числа без знака
b	below	нижче, у розумінні “менше”	числа без знака

Таблиця 5.10

Перелік команд умовного переходу для команди
cmp *оператор_1, оператор_2*

Тип операндів	Мнемокод команди	Критерій умовного переходу	Значення прапорців для здійснення переходу
Будь-які	je	операнд_1=операнд_2	zf = 1
Будь-які	je	операнд_1< >операнд_2	zf = 0
Зі знаком	jl / jnge	операнд_1<операнд_2	sf< > of
Зі знаком	jle / jng	операнд_1<=операнд_2	sf< > of or zf = 0
Зі знаком	jg / jnle	операнд_1>операнд_2	sf = of and zf = 0
Зі знаком	jge / jnl	операнд_1=>операнд_2	sf = of
Без знака	jb / jnae	операнд_1<операнд_2	cf = 1
Без знака	jbe / jna	операнд_1<=операнд_2	cf = 1 or zf = 1
Без знака	ja / jnbe	операнд_1>операнд_2	cf = 0 and zf = 0
Без знака	jae / jnb	операнд_1=>операнд_2	cf = 0

Команди умовного переходу не змінюють прапорці, тому після команди **cmp** може бути декілька команд умовного переходу. Це може бути використано для того, щоб дослідити кожну з альтернатив: “більше”, “менше” або “дорівнює”.

```
……
cmp ax, 5   ;(порівняти вміст ax з 5)
je eql   ;(перехід, якщо ax = 5)
jl low   ;(перехід, якщо ax<5)
jg grt   ;(перехід, якщо ax>5)
```

```
egl:
……..
low:
……..
grt:
……..
```

Команди умовного переходу та регістр ecx/cx. Архітектура мікропроцесора передбачає спеціальне використання деяких регістрів. Регістр ecx/cx виконує функцію лічильника в командах управління циклами й при роботі з ланцюгами символів. Можливо, що функціонально, команду умовного переходу, пов'язану з регістром ecx/cx, правильніше було б віднести до цієї групи команд. Синтаксис цієї команди умовного переходу такий:

jcxz *мітка_переходу* (jump if cx is Zero) – *перехід, якщо* cx *нуль*;
jecxz *мітка_переходу* (jump Equal ecx Zero) – *перехід, якщо* ecx *нуль*.

Ці команди зручно використовувати при організації циклів і при роботі з ланцюгами символів.

Слід зазначити, що команди умовної передачі управління jcxz/jecxz можуть адресувати тільки короткі переходи – на –128 *байт* або на +127 *байт* від наступної за нею командою.

5.5.3. Непряма адресація

Непряма базова (регістрова) адресація. За такої адресації ефективна адреса операнда може розміщуватися в будь-якому з регістрів загального призначення (крім esp/sp та ebp/bp – це специфічні регістри для роботи із сегментом стеку).

Синтаксично в команді цей вид адресації полягає у запису імені регістра у квадратних дужках [ ]. Наприклад, команда

```
mov ax, [ecx]
```

розміщує в регістр ax вміст слова за адресою із сегмента даних за адресою, яка зберігається в регістрі ecx. Вміст регістра легко змінити в ході роботи програми, тому даний спосіб адресації дозволяє динамічно призначати адресу операнда для деякої машинної команди. Це досить корисно для організації циклічних розрахунків.

Непряма базова (регістрова) адресація зі зміщенням. Цей вид адресації є доповненням до попередньої і призначений для доступу до даних із відомим зміщенням відносно деякої базової адреси. Його

зручно використовувати для доступу до елементів структур даних, коли зміщення елементів відоме наперед, на стадії розробки програми, а базова (початкова) адреса структури повинна розраховуватись динамічно, на стадії виконання програми. Модифікація вмісту базового регістра дозволяє звернутись до однойменних елементів різних екземплярів однотипних структур даних. Наприклад, команда

```
mov ax, [edx+3h]
```

пересилає до регістра ax слово з області пам'яті за адресою: вміст edx+3h.

Індексна адресація. Якщо для задавання адреси в команді використовується пряма адресація (у вигляді ідентифікатора) разом з одним з регістрів, то йдеться про індексну адресацію. Регістр вважається індексний і тому можна використовувати масштабування для отримання адреси потрібного елемента масиву, наприклад

```
mov ax, mas[dx]
```

пересилає до регістра ax слово за адресою: вміст dx із додаванням значення ідентифікатора mas (mas+dx) (транслятор надає кожному ідентифікатору значення, що дорівнює зміщенню цього ідентифікатора відносно початку сегмента даних). Команда add eax, mas [ebx*4] – додати до вмісту регістра eax подвійне слово в пам'яті за адресою mas+(ebx)*4.

Непряма індексна адресація зі зміщенням. Схожа на попередню адресацію зі зміщенням. Особливість полягає у можливості масштабування вмісту індексного регістра. З рисунку формату машинної команди становить цікавить байт sib (див. рис. 3.1). Одне з полів цього байта – поле масштабу ss, на значення якого множиться вміст індексного регістра. Наприклад, в команді

```
mov ax, mas[esi*2]
```

значення ефективної адреси другого операнда розраховувався за виразом mas+(esi*2).

Наявність можливості масштабування суттєво допомагає програмісту в організації індексації масивів, за умови, що розмір елемента масиву дорівнює 1, 2, 4 або 8 *байт*.

Непряма базово-індексна адресація. За цього виду адресації ефективна адреса формується як сума вмісту двох регістрів загального призначення: базового та індексного. Як такі регістри можуть викорис-

товуватись будь-які регістри загального призначення, при цьому часто використовується масштабування вмісту індексного регістра.

Наприклад

mov eax, [esi][edx].

У даному прикладі ефективна адреса другого операнда формується з двох компонентів

(esi)+(edx).

Непряма базово-індексна адресація зі зміщенням. Цей вид адресації є доповненням непрямої індексної адресації. Ефективна адреса формується як сума трьох складових: вмісту базового регістра, вмісту індексного регістра й значення поля зміщення в команді. Наприклад, команда

mov eax, [esi+5][edx]

пересилає в регістр eax подвійне слово за адресою (esi)+5+(edx).

Команда

add ax,mas [esi][ebx]

виконує додавання до вмісту регістра ax вміст слова за адресою: значення ідентифікатора mas+(esi)+(ebx).

Лівий регістр розглядається як базовий, а правий – як індексний.

Приклад

mov ax,mas [ebx][ecx*2] – адреса операнда дорівнює mas+(ebx)+(ecx)*2;
sub dx,[ebx+8][ecx*4] – адреса операнда дорівнює (ebx)+8+(ecx)*4.

Слід нагадати, що масштабування ефективне лише тоді, коли розмірність елементів масиву дорівнює 2, 4 або 8 *байт*.

5.6. МАСИВИ

Масив – структурований тип даних, який складається з визначеної кількості елементів одного типу.

Опис та ініціалізація масиву в програмі. Спеціальних засобів опису масивів у програмах *Assembler* немає. За необхідності використати масив у програмі його моделюють одним із таких способів:

1. Перелічення елементів масиву в полі операндів одною з директив опису даних. При переліченні елементи відділяються комами, наприклад

mas dd 1,2,3,4,5

– масив з 5 елементів. Розмір кожного елемента – 4 *байт*.

2. Використовуючи оператор повторювання dup, наприклад

```
mas dw 5 dup(0)
```

– масив з 5 нульових елементів. Розмір кожного елемента – 2 *байт*.

Такий спосіб визначення використовується для резервування пам'яті з метою розміщення й ініціалізації елементів масиву.

3. Використовуючи директиви label та rept. Пара таких директив може полегшити опис великих масивів у пам'яті та підвищити наочність такого опису.

Директива rept належить до макрозасобів мови *Assembler* і викликає повторення певної кількості команд, які записані між директивою та рядком endm.

Приклад. Визначити масив байт в області пам'яті, позначеної ідентифікатором mas_b. У цьому випадку директива label визначає символічне ім'я mas_b аналогічно до того, як це роблять директиви резервування й ініціалізації пам'яті. Перевагою директиви label є те, що вона не резервує пам'ять, а лише визначає характеристики об'єкта. У цьому випадку об'єкт – це комірка пам'яті. Використовуючи декілька директив label, записаних одна за одною, можна привласнити тій самій області різні імена й типи, що ілюструється прикладом:

```
……
n=0
……
mas_b label byte
mas_w label word
rept 4
    dw 0f1f0h
endm
```

Внаслідок цього в пам'яті буде створено послідовність із чотирьох слів f1f0**.** Цю послідовність можна трактувати як масив байтів або слів залежно від того, яке ім'я області буде використовуватися в програмі mas_b чи mas_w.

4. Використання циклу для ініціалізації значеннями області пам'яті, яку потім можна буде трактувати як масив.

Доступ до елементів масиву. При роботі з масивами необхідно розуміти, що всі елементи масиву розміщуються в пам'яті послідовно.

Для того, щоб локалізувати визначений елемент, до його імені необхідно додати індекс. Якщо моделюється масив, то необхідно моделювати також індекс. У мові *Assembler* індекси масивів – це звичайні адреси, але з ними працюють певним чином. Коли йдеться про індекс, то розуміють не номер елемента в масиві, а деяку адресу.

Якщо припустити, що в програмі статично визначено послідовність даних:

mas dw 0,1,2,3,4,5,

ця послідовність трактується як одновимірний масив. Розмірність кожного елемента визначається директивою dw, тобто дорівнює 2 *байт*. Нумерація (індексація) елементів масиву в *Assembler* починається з нуля. Для того, щоб отримати доступ до третього елемента масиву необхідно до адреси масиву додати 6 *байт*. Тобто, в даному випадку, фактично йдеться про 4-й елемент масиву – 3 (про це знає тільки програміст). Для мікропроцесора це неважливо – йому потрібна лише адреса. У загальному випадку для отримання адреси елемента в масиві необхідно до початкової (базової) адреси масиву додати добуток індексу (^{1} $_{\text{їїð}}$ -1, де ^{1} $_{\text{їїð}}$ – порядковий номер елемента масиву) цього елемента та розміру елемента масиву:

база + (індекс*розмір елемента)

Для роботи з масивами архітектура процесора надає базові та індексні регістри, що дозволяють реалізувати декілька режимів адресації даних. Використовуючи ці режими можна організувати ефективну роботу з масивами пам’яті.

1. *Індексна адресація зі зміщенням* – режим адресації, при якому ефективна адреса формується з двох компонентів:

– постійного (базового) – вказанням прямої адреси масиву у вигляді імені ідентифікатора, який позначає початок масиву;

– змінного (індексного) – вказанням імені індексного регістра. Наприклад:

```
mas dw 0,1,2,3,4,5
……
mov si, 4
mov ax, mas [si]  ;розмістити 3-й елемент масиву mas в регістр ax.
```

2. *Базова індексна адресація зі зміщенням* – режим адресації при якому ефективна адреса формується максимум із трьох компонентів:

– постійного (необов’язкового), як така може виступати пряма адреса масиву у вигляді ідентифікатора, який визначає початок масиву, або безпосереднє значення;

– змінного (базового) – указанням імені базового регістра;

– змінного (індексного) – указанням імені індексного регістра.

Цей вид адресації зручно використовувати при обробці двовимірних масивів.

5.7. ОРГАНІЗАЦІЯ ЦИКЛІВ

Організувати циклічне виконання деякої ділянки програми можна з використанням команд безумовної jmp та умовної jcc передачі управління.

Приклад. Підрахувати кількість нульових байтів у масиві mas, що складається з 5 елементів.

```
.data
mas  db  1,0,9,0,7
.code
start:  mov ax,
@data
mov ds, ax              ;кількість елементів масиву в cx.
mov cx, 5               ;обнулення ax
xor ax, ax              ;обнулення si
xor si, si              ;якщо (cx)=0 то вихід (exit)
cycl:  jcxz exit        ;порівнювання чергового елемента mas з 0
cmp mas[si], 0          ;якщо не дорівнює 0 то на m1
jne m1                  ;збільшити значення al  на 1 (лічильник нульо-
inc al                  вих елементів)
m1: inc si              ;перейти до наступного елемента масиву
dec cx                  ;зменшити cx. на 1 (декремент)
jmp  cycl
exit:  mov ax, 4c00h
int 21h                 ;повернення управління операційній системі
end  start
```

Отже, цикл організовано трьома командами jcxz, dec та jmp.

У системі команд мікропроцесора є три команди, що полегшують програмування циклів. Ці команди також використовують регістр ecx/cx як лічильник циклу. Синтаксис команд

loop *мітка_переходу*

loop – повторити цикл. Команда дозволяє реалізувати цикли (подібні до циклів for у мовах високого рівня) з автоматичним зменшенням лічильника циклу. Робота команди полягає у виконанні таких дій:

1. Декременту регістра ecx/cx.
2. Порівняння регістра ecx/cx з нулем:

- якщо (ecx/cx) > 0, то управління передається на мітку переходу;
- якщо (ecx/cx) = 0, то управління передається на наступну після loop команду.

Команда loope/loopz *мітка_переходу* (loop till cx<>0 or zero flag = 0) – повторити цикл, поки cx<>0 або zf = 0.

Команди loope та loopz повністю однакові, тому можна використовувати будь-яку. Робота команд полягає у виконанні таких дій:

1. Декремент регістра ecx/cx.
2. Порівняння регістра ecx/cx з нулем.
3. Аналіз стану прапорця нуля zf;

– якщо (ecx/cx)>0 та zf=0, то управління передається на мітку переходу;

– якщо (ecx/cx) = 0 або zf = 0, то управління передається на наступну після loope команду.

Команда loopne/loopnz *мітка_переходу* (loop till cx<>0 or nonzero flag = 0) – повторити цикл, поки cx<>0 або zf = 1.

Команди loopne та loopnz також абсолютно однакові. Робота команд полягає у виконанні таких дій:

1. Декремент регістра ecx/cx.
2. Порівняння регістра ecx/cx з нулем.
3. Аналіз стану прапорця нуля zf;

– якщо (ecx/cx)>0 та zf = 1, то управління передається на мітку переходу;

– якщо (ecx/cx)=0 або zf=0, то управління передається на наступну команду.

Команди loope та loopne за принципом дії зворотні одна одній. Вони розширюють дію команди loop тим, що додатково аналізують прапорець zf, що дає можливість, реалізувати достроковий вихід із циклу, використовуючи цей прапорець як індикатор. Недоліком розглянутих команд організації циклів є те, що вони реалізують тільки короткі переходи (від –128 до +127 *байт*). Для реалізації довгих переходів необхідно використовувати команди умовного переходу та команду jmp.

Приклад програми, аналогічної до попередньої однак використовує команду loop.

```
.data
len  equ 5              ;кількість елементів масиву
mas  db  1,0,9,0,7
.code
start:  mov ax,
@data
mov ds, ax              ;довжину поля mas в cx
```

```
mov cx, len          ;обнулення ax
xor ax, ax           ;обнулення cx
xor si, si           ;якщо (cx)=0 то вихід (exit)
jcxz  exit           ;порівнювання чергового елемента mas з 0
cycl: cmp mas[si],   ;якщо не дорівнює 0 то на m1
0                    ;збільшити значення al  на 1 (лічильник нульових
jne m1               елементів)
inc al               ;перейти до наступного елемента масиву
m1: inc si
loop  cycl
exit:  mov ax,       ;повернення управління операційній системі
4c00h
int 21h
end  start
```

5.8. ЗАСОБИ ЛОГІЧНОЇ ОБРОБКИ ДАНИХ

5.8.1. Команди аналізу та переміщення прапорців

Команди аналізу прапорців. Мнемонічне позначення деяких команд умовного переходу відбиває призначення прапорця, з яким вони працюють і має таку структуру: першою літерою є символ “j” (jump, перехід), другим символом – або призначення прапорця, або символ заперечення “n”, після якого стоїть назва прапорця. Якщо символу “n“ немає, то перевіряється стан прапорця (на наявність 1), і якщо він дорівнює 1, то здійснюється перехід на мітку. Якщо символ “n“ присутній, то перевіряється стан прапорця на рівність 0, і якщо прапорець дорівнює нулю, то здійснюється перехід на мітку переходу. Коди команд, назви прапорців та умови переходів наведено в табл. 5.11.

Таблиця 5.11

Команди аналізу прапорців

Назва прапорця	Номер біта	Команда умовного переходу	Значення прапорця для здійснення переходу
Прапорець переносу cf	1	jc jnc	cf = 1 cf = 0
Прапорець парності pf	2	jp jnp	pf = 1 – парну pf = 0
Прапорець нуля zf	6	jz jnz	zf = 1 – результат нуль zf = 0 – результат не нуль
Прапорець знака sf	7	js jns	sf = 1 sf = 0
Прапорець переповнення of	11	jo jno	of = 1 – перенос of=0

Команди переміщення прапорців.

Команда LAHF (*Load AH register from register Flags*) копіює до регістра AH вміст бітів SF, ZF, AF, PF, CF з регістра *Flags*.

Команда SAHF (*Store AH register into register Flags*), навпаки, переміщує ці біти з регістра AH до регістра *Flags*.

Синтаксис

LAHF
SAHF.

Команди PUSHF та POPF розміщують у стек та вилучають зі стеку прапорці регістра стану Flags процесора.

5.8.2. Класифікація засобів логічної обробки даних

Деякі додатки в процесі виконання повинні маніпулювати не тільки байтами, символами, подвійними символами тощо, а й окремими бітами змінних. У систему команд процесора включено велику кількість команд, призначених для виконання операцій над окремими бітами. До цих команд входять як інструкції для реалізації стандартних логічних операцій (І, АБО, НІ, виключне АБО), так і команди зсуву та циклічних зсувів. На рис. 5.4 наведено засоби МП для організації роботи з даними та правилами формальної логіки.

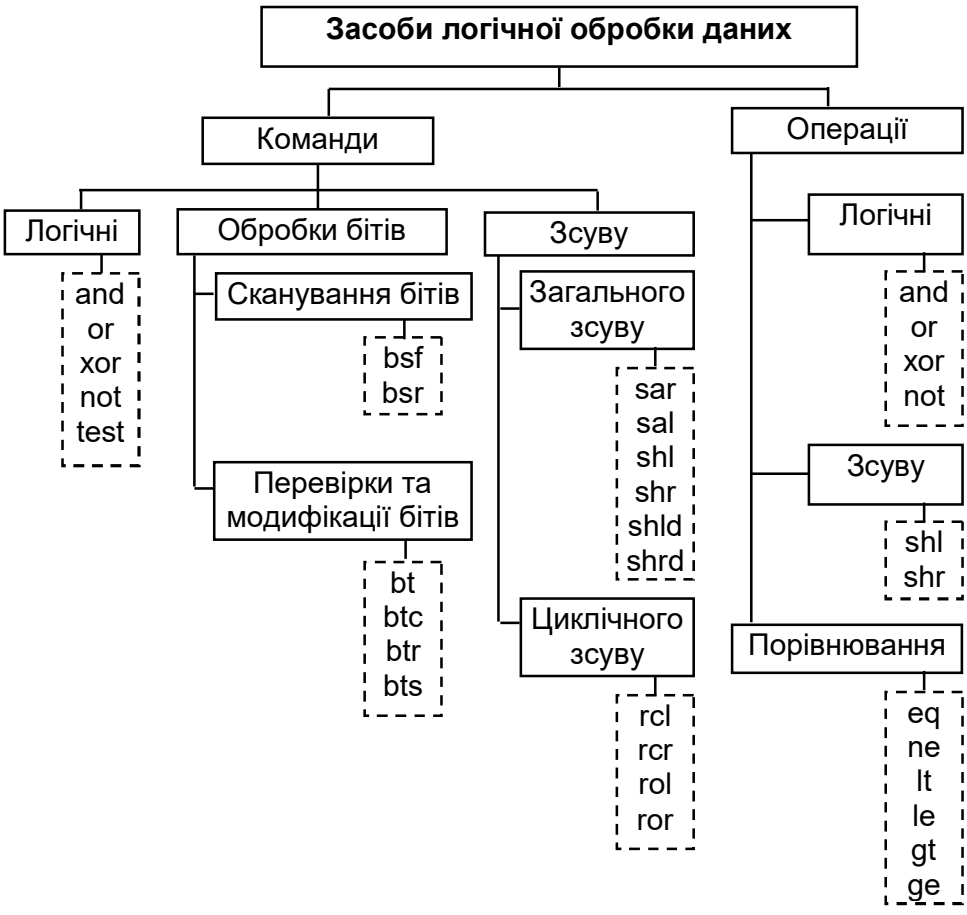

Рис. 5.4. Класифікація засобів логічної обробки даних

Засоби логічної обробки даних поділено на дві групи: логічні команди та логічні операції.

Інструкції зсуву та логічного зсуву досить часто використовуються в алгоритмах швидкого множення та ділення при виконанні цілочислових операцій; інструкції скасування окремих бітів корисні, наприклад, при обробці даних, що надходять від пристроїв введення-виведення в драйверах, аналізу окремих бітів у потоці двійкових даних і т. д.

У сучасних процесорах (включаючи *Intel Pentium* 4) логічні операції виконуються незалежно від математичних операцій із цілими числами і числами з рухомою комою. При цьому використовуються один із виконавчих модулів цілочислових операцій (ALU), що забезпечує високий ступінь паралелізму і, відповідно, ефективність конвеєра команд.

5.8.3. Логічні команди процесора

Операція логічного множення (логічне І):

and *операнд_1, операнд_2*

Команда виконує порозрядну логічну операцію І (кон'юнкцію) над бітами операндів. Результат записується на місце *операнда_1*.

На місці першого операнда може бути регістр (крім сегментного) або комірка пам'яті, а на місті другого операнда – регістр (крім сегментного), комірка пам'яті або безпосереднє значення. Розмір операндів змінюється від байта до подвійного слова. Команда and впливає на прапорці sf, zf, та pf регістра EFLAGS (рис. 5.5).

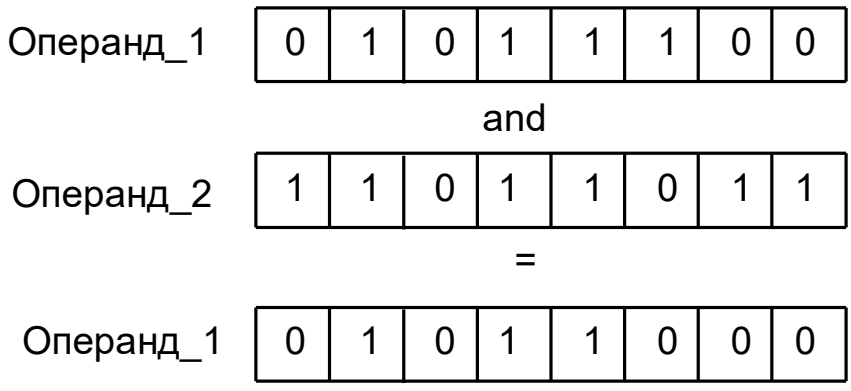

Рис. 5.5. Процес виконання команди and

Операція логічного додавання (логічне АБО).

or *операнд_1, операнд_2.*

Команда виконує порозрядну логічну операцію АБО (диз'юнкцію) над бітами операндів. Результат записується на місце *операнд_1* (рис. 5.6).

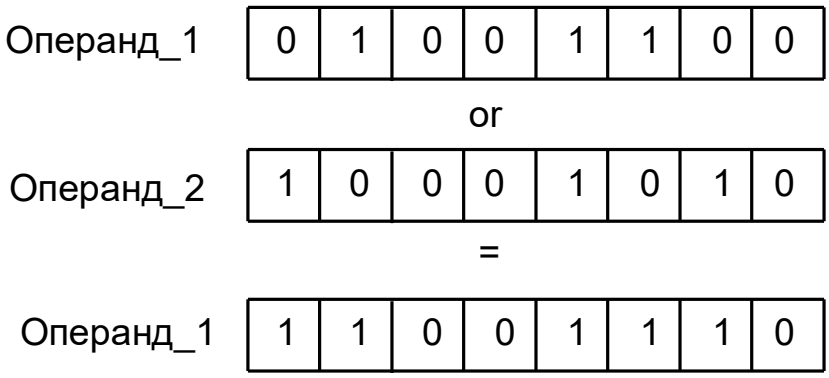

Рис. 5.6. Процес виконання команди or

Як перший операнд може виступати регістр (крім сегментного) або комірка пам'яті, а як другий – регістр (крім сегментного), комірка пам'яті або безпосереднє значення. Не допускається визначати обидва операнди як комірки пам'яті, а сам операнд може бути байтом, словом або подвійним словом. Команда впливає на прапорці of, sf, zf, pf та cf. Прапорці cf та of завжди скидаються в 0.

Операція логічного виключного додавання (виключного АБО).

xor *операнд_1, операнд_2.*

Команда виконує порозрядну логічну операцію виключного АБО над бітами обох операндів. Результат записується на місце *операнд_1* (рис. 5.7).

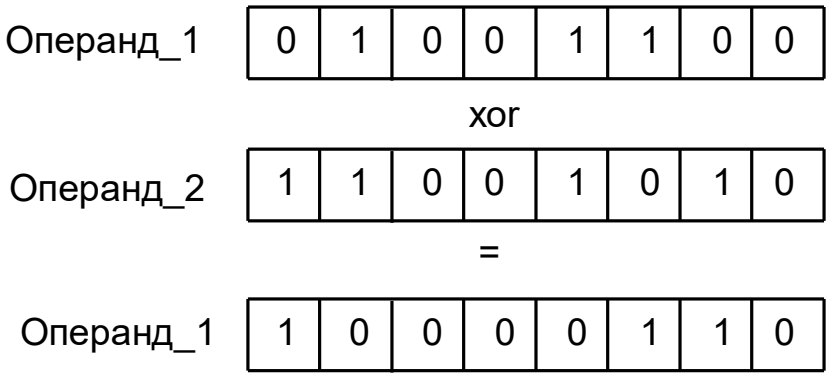

Рис. 5.7. Процес виконання команди xor

Результатом логічної операції виключного додавання є "істина" (1), якщо лише один із двох операндів має значення "істина" (1), та "хибно" (0), якщо обидва операнди мають значення "хибно" (0) або "істина" (1) (Біти результату встановлюються таким чином: біт буде

дорівнювати 1, якщо відповідні біти операндів різні; біт буде дорівнювати 0, якщо відповідні біти операндів однакові).

Як перший операнд допускається використовувати регістр (крім сегментного) або комірку пам'яті, а як другий – регістр (крім сегментного), комірка пам'яті або безпосереднє значення. Не допускається одночасно вказувати як обидва операнди комірки пам'яті. Операнди можуть бути байтами, словами або подвійними словами. Команда впливає на прапорці of, sf, zf, pf та cf, причому прапорці cf та of завжди встановлюється в 0.

Операція логічної перевірки (операція І) (операція "перевірити").

test *операнд_1, операнд_2.*

Команда виконує порозрядну логічну операцію І над бітами обох операндів. Стан операндів не змінюється. Змінюються лише прапорці zf, sf, та pf, що дає можливість аналізувати стан окремих бітів операнда без зміни їх стану.

Операція логічного заперечення (логічне НІ).

not *операнд*

Команда виконує порозрядне інвертування кожного біта операнда. Результат записується на місце операнда.

За допомогою логічних команд можливе відокремлення окремих бітів в операнді з метою їх встановлення, скидування, інвертування або просто перевірки на визначене значення. Для організації такої роботи з бітами *операнда_2*, як правило, виконує роль маски. За допомогою встановлених в 1 бітів цієї маски і визначаються необхідної для конкретної операції біти *операнд_1*.

Приклади. Для встановлення потрібних розрядів (біт) в 1 застосовується команда or. Команда

or eax, 10b

встановлює перший біт регістра eax в 1. Нумерація бітів у регістрі починається з нуля. Отже, *операнд_2*, який виступає в ролі маски, повинен містити одиниці на місцях тих розрядів, які повинні встановлюватись в 1.

Для скидання визначених розрядів (бітів) в 0 застосовують команду and, наприклад

and eax, ffff fffdh

Дана команда скидає в 0 перший біт регістра eax. Отже, *операнд_2* повинен містити нульові біти на місці тих розрядів, які повинні бути встановлені в 0 в *операнд_1*.

Команда xor застосовується:

- для з'ясування того, як біти в операндах відрізняються;
- для інвертування стану заданих бітів в *операнд_1*.

Ті біти, які становлять цікавість, в *операнд_2* повинні бути одиничними, решта нульовими.

```
xor eax, 10b     ;  інвертувати 1-й біт у регістр eax
jz mes           ;  перехід, якщо 1-й біт в al був одиничним.
```

Для перевірки стану заданих бітів застосовується команда

test о*перанд_1*, *операнд_2*

- перевірити *операнд_1*.

Ті біти, які перевіряються в *операнд_1*, у масці (*операнд_2*) повинні мати одиничне значення. Алгоритм роботи команди test аналогічний до алгоритму команди and, але він не змінює значення *операнд_1*. Результатом команди є встановлення значення прапорця нуля zf:

- якщо zf = 0, то внаслідок логічного множення отримано ненульовий результат, тобто хоча б один одиничний біт маски співпав із відповідним одиничним бітом операндом – 1.
- якщо zf = 1, то внаслідок логічного множення отримано нульовий результат, тобто ні один одиничний біт маски не співпав із відповідним бітом операнд – 1.

```
test eax     ; 00000010h
jnz  m1      ; перехід якщо 1-й біт дорівнює 1.
```

Отже, для реакції на результат команди test доречно використовувати команду переходу jnz мітка (jump if not zero) – перехід, якщо прапорець zf ненульовий, або команду зворотної дії – jz мітка (*jump if not zero*) – перехід, якщо прапорець нуля zf = 0.

5.8.4. Команди обробки бітів

Команди обробки бітів поділяються на дві групи.

1. Сканування бітів. Команди сканування бітів створюють групу команд розрядних логічних операцій. Основне призначення цих команд – організація розгалужувань та умовних переходів у програмі на основі аналізу стану окремих бітів.

Нижче наводиться коротка характеристика цих методів:

bsf *операнд_1*, *операнд_2*

(Bit Scaning Forward) – сканування бітів вперед (від молодшого розряду до старшого).

Команда переглядає біти *операнд_2* від молодшого біта до старшого (від біта 0 до старшого) в пошуках першого біта, встановленого в 1. Якщо такий біт знайдено, то в *операнд_1* заноситься номер цього біта у вигляді цілочислового значення. Якщо всі біти *операнд_2* дорівнюють нулю, то прапорець нуля zf встановлюється в 1, в іншому випадку zf = 0. Наприклад:

```
mov al , 02h
bsf  bx , al       ; bx=1
jz  m1             ; перехід на m1, якщо al = 004 (тобто zf=1)
```

Команда

bsr *операнд_1, операнд_2*

(Bit Scaning Reset) – сканування бітів у зворотному порядку.

Команда переглядає (сканує) біти *операнд_2* від старшого розряду до молодшого в пошуках першого біта, встановленого в 1.

Якщо такий виявлено, то в *операнд_1* заноситься номер цього біта у вигляді цілочислового значення. При цьому важливо, що позиція першого одиничного біта зліва починає відлік все одно відносно нульового. Якщо всі біти *операнд_2* дорівнюють нулю, то прапорець нуля встановлюється в zf = 1 , в іншому випадку – zf = 0.

2. Команди перевірки та модифікації бітів. В останніх моделях МП *Intel* у групі логічних команд з'явились ще декілька команд, які дозволяють здійснити доступ до одного конкретного біта операнда. Операнд може знаходитись як в пам'яті, так і в регістрі загального призначення. Значення зміщення може задаватися, як у вигляді безпосереднього значення, так і міститься в регістрі загального призначення. Як значення зміщення можна використовувати результати роботи команд bsr та bsf. Усі команди присвоюють прапорцю cf значення вибраного біта.

bt *операнд, зміщення_біта*

(Bit test) – перевірка біта. Команда переносить значення біта в прапорець cf.

Наприклад:

```
bt  ax, s      ; перевірити значення біта s
jnc  m1        ; перехід, якщо біт = 0
```

Команда

bts *операнд, зміщення_біта*

(Bit test and set) – перевірка і встановлення біта.

Команда переносить значення біта в прапорець cf і потім встановлює біт, що перевіряється, в 1.

Наприклад:

```
mov ax ,  10
bts  pole , ax    ; перевірити та встановити 10-й біт у pole
jc  m1            ; перехід, якщо перевірний біт дорівнював 1
```

Команда

btr *операнд, зміщення_біта*

(Bit test and reset) – перевірка й скидання біта.

Команда переносить значення біта в прапорець cf і потім встановлює біт в 0.

Команда

btc *операнд, зміщення_ біта*

(Bit test and convert) – перевірка та інвертування біта.

Команда переносить значення біта в прапорець cf і потім інвертує значення цього біта.

Команди аналізу прапорця cf:

jc – перевірка прапорця cf, якщо cf = 1 то перехід;

jcn – перевірка прапорця cf, якщо cf = 0 то перехід.

5.8.5. Команди зсуву бітів

Команди цієї групи також забезпечують виконання певних дій над окремими бітами, однак іншими способами, ніж логічні команди. Усі команди зсуву розміщують біти в полі операнда ліворуч або праворуч залежно від коду операції. Усі команди зсуву мають однакову структуру:

КОП *операнд, лічильник зсувів*

Кількість розрядів, які зсуваються (лічильник зсувів), може задаватись двома способами:

статично – передбачає задавання фіксованого значення за допомогою безпосереднього операнда;

динамічно – занесенням значення лічильника зсувів у регістр cl перед виконанням команди зсуву.

Однак із метою оптимізації МП приймає тільки значення п’яти молодших розрядів лічильника, а не всі 8 (регістр cl – восьмирозрядний).

Отже, значення зсуву лежить у межах від 0 до 31. Однак в останніх моделях мікропроцесорів є додаткові команди, які дозволяють реалізовувати 64-розрядні зсуви. Усі команди зсуву встановлюють прапорець переносу cf. У міру зсуву біта за межі операнда вони спочатку подають у прапорець переносу, встановлюючи його рівним значенню наступного біта, що з’явився за межами операнда. Куди цей біт потрапляє потім, залежить від типу команди зсуву й алгоритму програми.

За принципами дії команди зсуву поділяють на два типи: лінійного зсуву та циклічного зсуву.

1. Лінійний зсув. Команди лінійного зсуву працюють за такими алгоритмами:

- “висувний” біт встановлює прапорець cf (записується у cf);
- біт, що вводиться з іншого кінця, має значення 0;
- при зсуві наступного біта він переходить у прапорець cf, при цьому значення попереднього зсуву того біта губиться.

Команди лінійного зсуву поділяються на дві підгрупи:

- команди логічного лінійного зсуву;
- команди арифметичного лінійного зсуву.

До команд логічного лінійного зсуву належать:

Команда

shl *операнд, лічильник_зсувів*

(Shift Logical Left) – логічний зсув вліво.

Вміст операнда зсувається на кількість бітів, визначену значенням *лічильник_зсувів*. Праворуч (у позицію молодшого біта) вписують нулі (рис. 5.8).

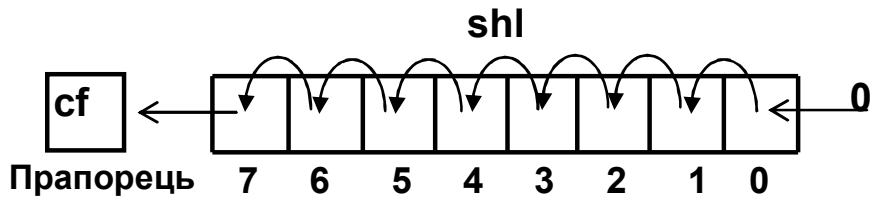

Рис. 5.8. Процес виконання команди shl

Команда

shr *операнд, лічильник_зсувів*

(Shift Logical Right) – логічний зсув вправо.

Вміст операнда зсувається на кількість бітів, визначену значенням *лічильник_зсувів*. Зліва (у позицію старшого біта) вписують нулі (рис. 5.9).

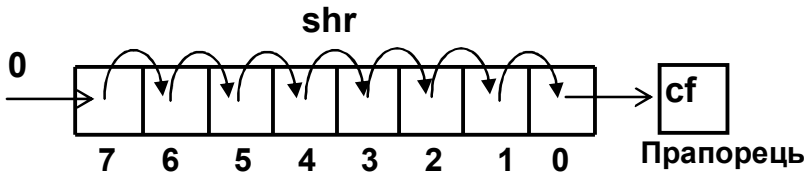

Рис. 5.9. Процес виконання команди shr

Команди арифметичного лінійного зсуву відрізняються від команд логічного зсуву тим, що вони певним чином працюють зі знаковими розрядом операнда.

Команда

sal *операнд, лічильник_зсувів*

(Shift Arithmetic Left) – арифметичний зсув.

Вміст операнда зсувається на кількість бітів, визначену значенням *лічильник_зсувів*. Праворуч (у позицію молодшого біта) вписують нулі. Команда sal не зберігає знака, однак у випадку зміни знака встановлює прапорець cf наступним висувним бітом. В іншому випадку команда sal повністю аналогічна до команди shl.

Команда

sar *операнд, лічильник_зсувів*

(Shift Arithmetic Right) – арифметичний зсув вправо.

Вміст операнда зсувається на кількість бітів, визначену значенням *лічильник_зсувів*. Ліворуч (в позицію старшого біта) вписують нулі. Команда sar зберігає знак, відновлюючи його після кожного наступного біта (рис. 5.10).

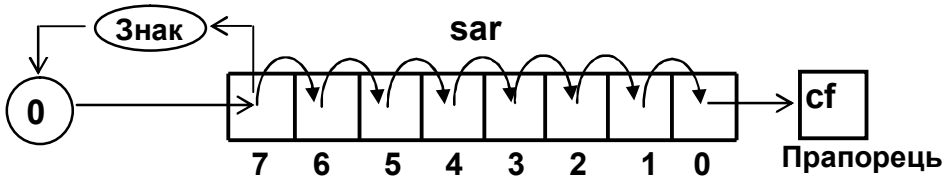

Рис. 5.10. Процес виконання команди sar

Команди арифметичного зсуву дозволять виконати множення та ділення операнди на степені двійки, наприклад:

$$0101 \leftarrow 5$$
$$1010 \leftarrow 10$$

Друге число є зсунутим ліворуч на один розряд відносно першого числа й відповідає операції 5*2. Те саме відбувається з множенням на 4, 8, 16 і т.д. Ця властивість може бути корисною при оптимізації програм.

Циклічний зсув. До команд циклічного зсуву належать команди, які зберігають значення зсувних бітів. Команди циклічного зсуву поділяються на два типи:

- команди простого циклічного зсуву;
- команди циклічного зсуву через прапорець переносу cf.

До **команд простого циклічного зсуву** належать:

rol_ *операнд, лічильник_зсувів*

(Rotate Left) – циклічний зсув вліво.

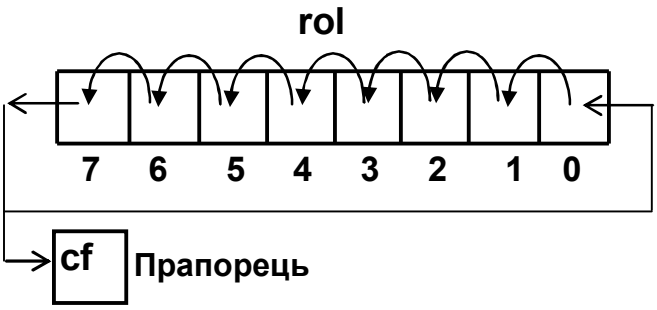

Рис. 5.11. Процес виконання команди rol

Вміст операнда зсувається вліво на кількість бітів, визначену в операнді *лічильник_зсувів*. Біти, що зсуваються вліво, одночасно записуються в той самий операнд праворуч (рис. 5.11).

Команда

ror *операнд, лічильник_зсувів*

(Rotate Right) – циклічний зсув вправо.

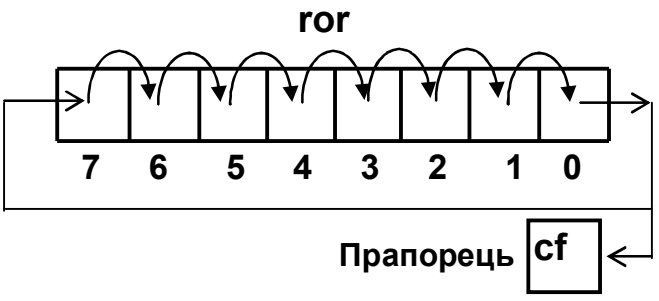

Рис. 5.12. Процес виконання команди ror

Вміст операнда зсувається вправо на кількість бітів, визначену в операнді *лічильник_зсувів*. Біти, що зсуваються вправо, одночасно записуються в той самий операнд ліворуч (рис. 5.12).

Команди простого циклічного зсуву в процесі своєї роботи здійснюють одну корисну дію: біт, що циклічно зсувається, не тільки записується в операнд з іншого кінця, а й одночасно стає значенням

прапорця cf. Наприклад, щоб обміняти вміст двох половинок регістра eax достатньо виконати таку послідовність команд:

```
mov  eax, ffff 0000h
mov  cl, 16
role ax, cl
```

Команди циклічного зсуву через прапорець переносу cf відрізняються від команд простого циклічного зсуву тим, що біт, який зсувається, не зразу потрапляє в операнд з іншого кінця, а спочатку записується в прапорець переносу cf. І тільки наступне виконання даної команди приводить до розміщення висунутого раніше біта в інший кінець операнда.

До команд циклічного зсуву через прапорець переносу cf належать такі:

rcl *операнд, лічильник_зсувів*

(Rotate through Carry Left) – циклічний зсув вліво через перенос (рис. 5.13).

rcr *операнд, лічильник_зсувів*

(Rotate through Carry Right) – циклічний зсув вправо через перенос (рис. 5.14).

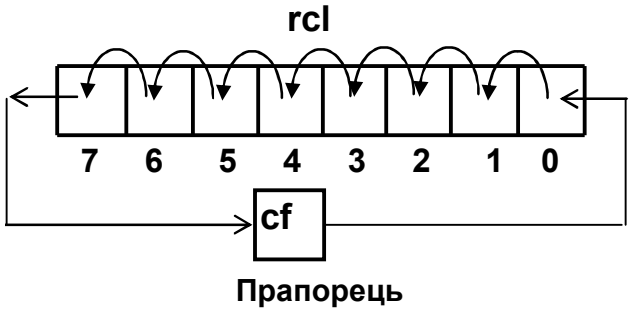

Рис. 5.13. Процес виконання команди rcl

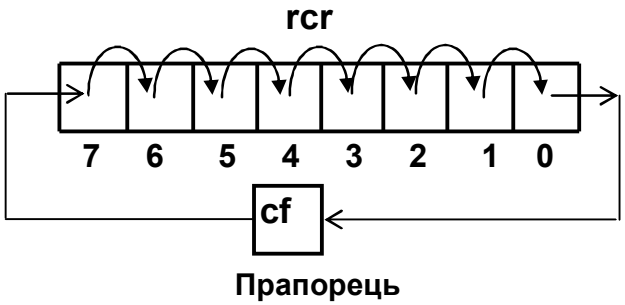

Рис. 5.14. Процес виконання команди rcr

Біти, що зсуваються, спочатку стають значенням прапорця переносу cf. При зсуві через прапорець переносу з'являється новий елемент, за допомогою якого, зокрема, можна здійснювати заміну циклічно-зсувних бітів, наприклад, розгалуження бітових послідовностей. Під *розгалуженням бітових послідовностей*, розуміють дію, яка дозволяє деяким чином локалізувати та вилучити потрібні ділянки цієї послідовності, і записати їх в інше місце.

Приклад. Переписати в регістр bx старшу половину регістра eax з однозначним її обнуленням в регістрі eax:

```
mov  cx, 16          ; кількість зсувів для eax
m1:
clc                  ; скидання прапорця cf в 0
rcl  eax, 1          ; зсув крайнього лівого біта  з eax в cf
rcl  bx, 1           ; переміщення біта із cf в bx
loop m1              ; цикл 16 разів
rol  eax, 16         ; відновити праву частину eax
```

Команди простого циклічного зсуву можна використовувати для підрахунку кількості одиничних бітів у регістрі eax, наприклад:

```
xor  dx, dx      ;обнулення dx (dx лічильник бітів)
mov  cx, 32      ;кількість циклів підрахунку
cycl:
ror  eax, 1      ; циклічний зсув вправо на 1 біт
jnc  not_one     ; перехід, якщо наступний біт в cf не дорівнює 1
inc  dx          ; збільшення лічильника бітів
not_one:
loop cycl        ; перехід на мітку cycl, якщо значення в cx не дорів-
                 нює 0
```

Команди зсувів подвійної точності (додаткові команди зсуву).

shld *операнд_1, операнд_2, лічильник_зсувів*

– зсув вліво подвійної точності.

Команда shld виконує заміну шляхом зсуву бітів операнда *операнд_1* вліво, заповнюючи його біти праворуч значеннями бітів, що виштовхується, із значенням *лічильник_зсувів*, яке лежить в межах від 0 до 31. Це значення може задаватися безпосередньо операндом або міститися в регістрі cl. Значення *операнд_2* не змінюється.

shrd *операнд_1, операнд_2, лічильник_зсувів*

– зсув вправо подвійної точності.

Команда виконує заміну шляхом зсуву бітів операнда *операнд_1* вправо, заповнюючи його біти ліворуч значеннями бітів, що виштовхуються із *операнд_2*. Кількість бітів, що зсуваються, визначаються значенням *лічильник_зсувів*. Значення *операнд_2* не змінюється. Зсув здійснюється через прапорець cf.

5.9. ОРГАНІЗАЦІЯ СТЕКОВОЇ ПАМ'ЯТІ. РОБОТА ЗІ СТЕКОМ

Адресна пам'ять, незважаючи на її широке застосування, має низку істотних недоліків. По-перше, необхідність вказувати в командах адреси операндів збільшують їх довжину і, тому, вимагають великого об'єму пам'яті для зберігання програм. По-друге, пошук інформації за адресою вимагає значних витрат часу на дешифрацію адреси, які збільшуються із зростанням об'єму пам'яті. По-третє, виконання багатьох простих логічних операцій над даними, наприклад, пошук більшого або меншого числа, алфавітний пошук, упорядкування та ін., вимагає великої кількості звернень до пам'яті через необхідність читання всіх операндів, які порівнюються. Тому для розв'язання специфічних задач в ЕОМ широко застосовуються різноманітні типи безадресної пам'яті, у якій розташування та пошук інформації не пов'язані з її адресою в запам'ятовувальному масиві.

Типами безадресної пам'яті є стекова, кеш та асоціативна. Принципи організації запам'ятовувальних пристроїв цих типів:

Стек – це ділянка пам'яті, яка спеціально виділяється для тимчасового збереження даних програми. У структурі програми для стеку передбачено окремий сегмент. Якщо програміст забуде описати сегмент стеку у своїй програмі, то компонувальник tlink видасть попереджувальне повідомлення.

Для роботи зі стеком призначено три регістри:

SS – регістр сегмента стеку;

ESP/SP – регістр покажчика стеку;

EBP/BP – регістр покажчика бази кадру стеку.

Розмір стеку залежить від режиму роботи процесора й обмежується значенням 64 *Кбайт* у реальному режимі або 4 *Гбайт* у захищеному режимі. У певний момент часу доступним є тільки один стек, адреса якого міститься в регістрі SS. Цей стек називається *поточним*. Для того, щоб звернутись до іншого стеку необхідно в регістр SS завантажити іншу адресу. Регістр SS автоматично використовується процесором для виконання всіх команд, які працюють зі стеком.

Особливості роботи зі стеком.

1. Запис та читання даних у стеку здійснюється відповідно до принципу LIFO *(Last In First Out* – останнім прийшов, першим вийшов).

2. У міру запису даних у стек він збільшується в бік молодших адрес. Цю особливість закладено в алгоритм команд роботи зі стеком.

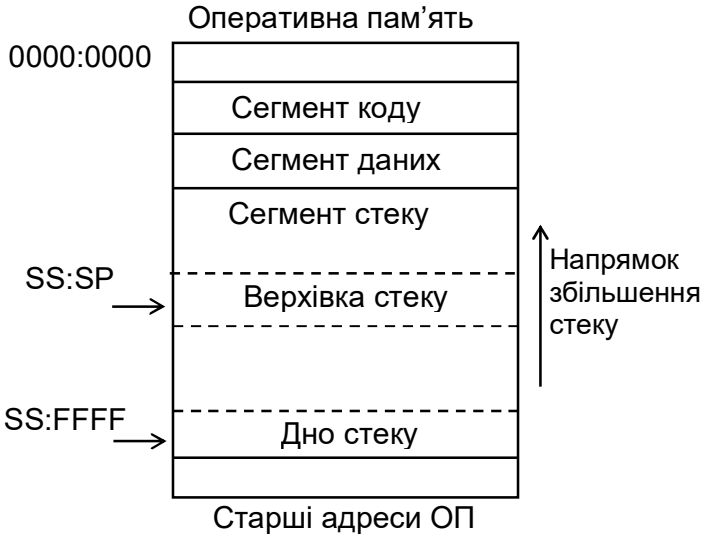

Рис. 5.15. Організація стеку

3. При використанні регістрів ESP/SP та EBP/BP для адресації даних *Assembler* автоматично вважає, що значення, які в ньому розміщені, являють собою зміщення відносно сегментного регістра SS.

У загальному випадку стек організовано так, як показано на рис. 5.15.

Регістри SS, ESP/SP та EBP/BP використовуються комплексно і кожен з них має своє функціональне призначення. Регістр ESP/SP завжди вказує на верхівку стеку, тобто містить зміщення, по якому у стек було записано останній елемент. Команди роботи зі стеком неявно змінюють цей регістр так, щоб він завжди вказував на останній елемент, який був записаний у стек. Якщо стек пустий, то значення ESP/SP дорівнює адресі останнього байта сегмента, що виділений під стек.

При розміщенні елемента в стек, процесор зменшує значення ESP/SP, а потім записує елемент за адресою нової верхівки. При вилученні даних зі стеку процесор копіює елемент, який розміщений за адресою верхівки стеку, а потім збільшує значення регістра покажчика стеку ESP/SP. Отже, стек збільшується вниз, тобто в бік зменшення адрес.

Для доступу до даних всередині стеку застосовують регістр EBP/BP. Регістр EBP/BP – регістр покажчика бази кадру стеку.

Приклад застосування цього регістра. Типовою операцією при вході у підпрограму є передача потрібних параметрів шляхом розміщення їх у стек. Якщо підпрограма теж активно використовує стек, до доступ до цих параметрів стає проблематичним. Вихід із такої ситуації полягає в тому, щоб зберегти адресу верхівки стеку в покажчику бази кадру стеку – регістрі EBP/BP. Потім значення EBP/BP можна використовувати для доступу до переданих даних.

Початок стеку розташовано в області старших адрес пам'яті. На рис. 5.16 ця адреса позначена парою SS:ffff. Зміщення ffff тут надано умовно. Реально це значення визначається величиною, яку програміст задає при описі сегмента стеку у своїй програмі. Наприклад, при застосуванні директиви stack 100h, початку стеку буде відповідати

пара SS:0100h. Адресна пара SS:ffff – це максимальне для реального режиму значення адреси початку стеку, через те, що розмір сегмента в ньому обмежено величиною 64 *Кбайт* (0ffffh).

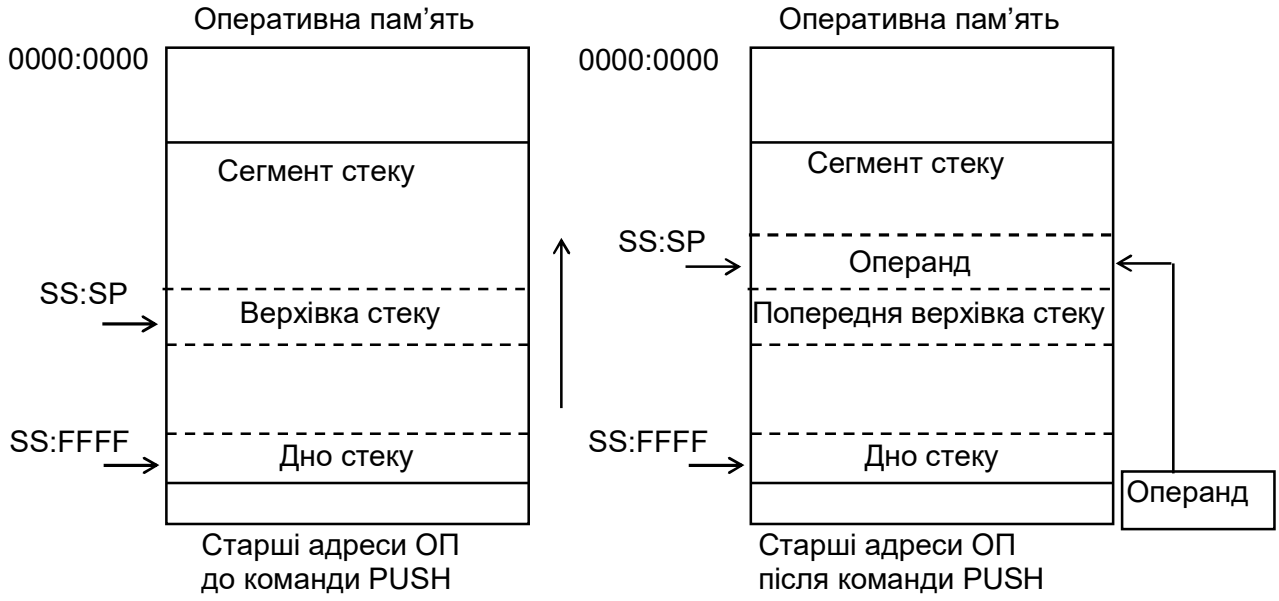

Рис. 5.16. Процес виконання команди PUSH

Для організації роботи зі стеком існують спеціальні команди запису та читання.

Команда PUSH виконує запис значення <джерело> у верхівку стеку:

push <джерело>

Алгоритм виконання даної команди включає два етапи:

1. Значення SP зменшується на 2:

(SP)=(SP)-2

2. Значення джерела записується за адресою, на яку вказує пара SS:SP.

Команда POP виконує вилучення значення з верхівки стеку в місце, яке вказується операндом <приймач> (рис. 5.17)

pop <приймач>

Команда виконується за таким алгоритмом

1. Запис вмісту верхівки стеку в місце, яке вказується операндом <приймач>.

2. Збільшення значення SP на 2:

(SP)=(SP)+2.

Команда PUSHA призначена для групового запису в стек. За цією командою в стек послідовно записується вміст регістрів AX, CX, DX,

BX, SP, BP, SI, DI. Слід зазначити, що в стек записується те значення регістра SP, яке було до видання команди PUSHA (рис. 5.18).

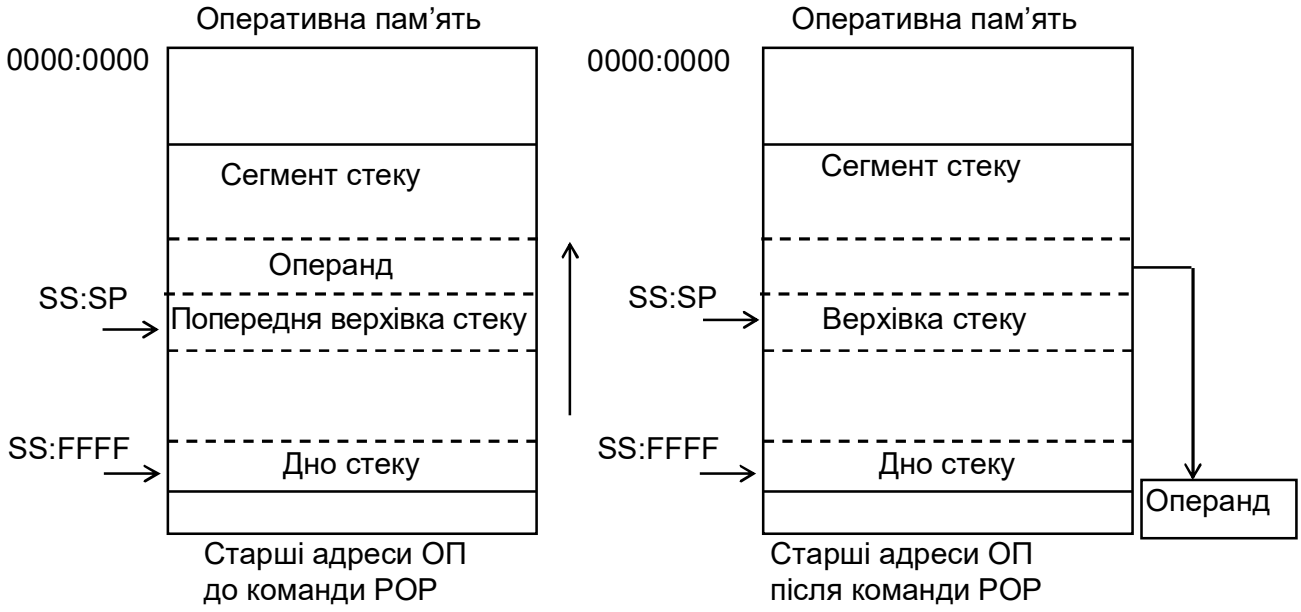

Рис. 5.17. Процес виконання команди POP

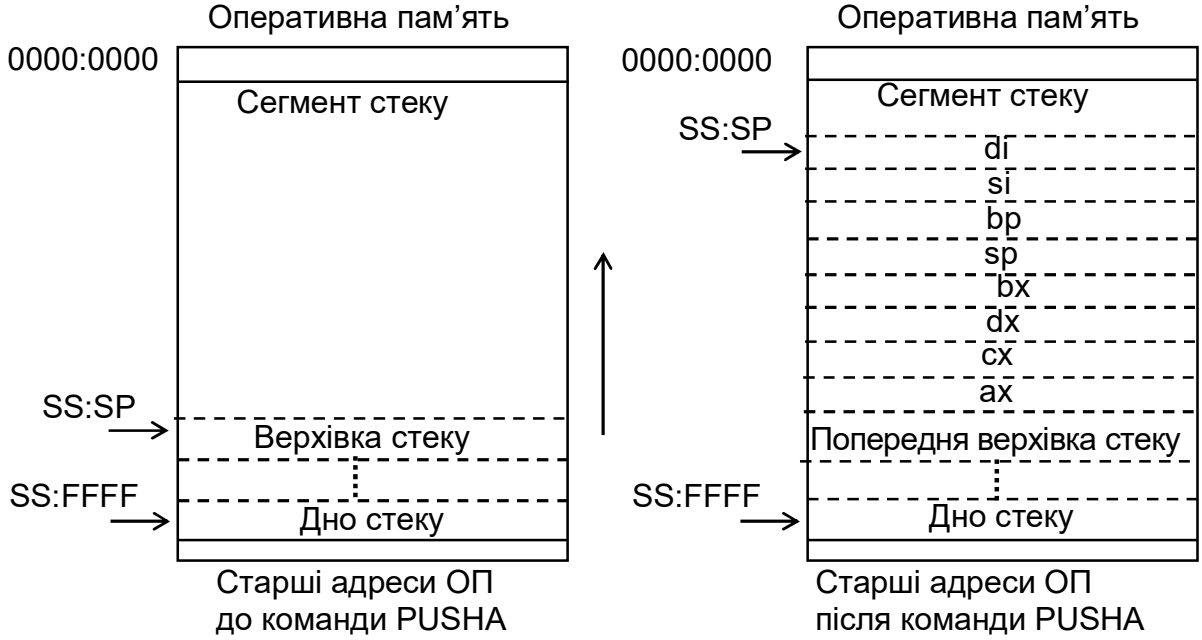

Рис. 5.18. Процес виконання команди PUSHA

Команда PUSHAW майже ідентична команді PUSHA, однак існують деякі відмінності, які стосуються атрибута розміру сегмента. Він може набувати двох значень use16 та use32:

- при use16 алгоритм роботи команд PUSHAW та PUSHA однаковий, працюють з 16-розрядними регістрами;

- при use32 команда PUSHA стає чутливою до розрядності сегмента й при встановлення 32-розрядного сегмента працює з відповідними 32-розрядними регістрами, тобто EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. Алгоритм виконання команди PUSHAW не змінюється, тобто вона завжди працює з 16-розрядними регістрами.

Наступні команди виконують дії, зворотні діям описаних команд:

POPA
POPAW.

Ці команди дозволяють зберегти в стеку регістр прапорців та записати слово або подвійне слово. Слід зазначити, що перераховані команди єдині в системі команд процесора, які дозволяють отримати доступ до вмісту всього регістра прапорців. Команда PUSHF зберігає регістр прапорців у стеку. Робота цієї команди залежить від атрибута розміру сегмента:

- при use16 у стек записується регістр FLAGS розміром 2 *байт*;
- при use32 в стек записується регістр EFLAGS розміром 4 *байт*.

Команда PUSHFW зберігає у стеку регістр прапорців розміром в слово. З атрибутом use16 завжди працює так само, як і команда PUSHF.

Команда PUSHFD зберігає у стеку регістр прапорців FLAGS або EFLAGS залежно від атрибута розміру сегмента, тобто працює так само, як PUSHF.

Наступні команди виконують дії, зворотні діям команд, описаних вище:

POPF
POPFW
POPFD

Контрольні питання

1. Назвати види адресації даних у пам’яті комп’ютера.
2. Дати характеристику прямої адресації даних.
3. Назвати види непрямої адресації.
4. Як організовано стекову пам’ять?
5. Які команди використовує процесор для роботи зі стеком?
6. Які команди використовує процесор для реалізації прямих переходів?
7. Назвати етапи розробки програм на *Assembler*.
8. Якою є структура програми на *Assembler*?
9. Як ініціалізувати дані в програмах на *Assembler*?
10. Назвати директиви опису сегментів.
11. Як виконується команда loop?