A S S E M B L E R

РЪКОВОДСТВО ЗА ПРАКТИКУМ

Стела Русева

Даниела Славкова

Катедра „Изчислителни системи”

2007 г.

Съдържание

Съдържание 1
Вместо предговор 5
Тема 1.Въведение 6
1.1. Процесори 80x86 и Pentium 6
1.2. Сегментен регистър и формирането на адреса 10
1.3. Формати на данните 10
1.3.1. Цели данни без знак 11
1.3.2. Цели данни със знак 11
1.3.3. Десетични (BCD) данни 11
1.3.4. Данни с плаваща запетая 11
1.3.5. Низове 11
1.3.6. ASCII данни 12
1.3.7. Указатели 12
Тема 2.Програма DEBUG 12
2.1. Команди на програмата DEBUG 12
2.2. Пример 19
Тема 3. Прекъсване функциите на ROM-BIOS и MS-DOS 21
3.1. Въвеждане от клавиатурата 23
3.1.1. Процедурата за обработка на INT 09 24
3.1.2. Буфер на клавиатурата 26
3.1.3. Прекъсване 16h 28
3.1.4. Изводи 30
3.2. Извеждане на екрана 30
3.3. Основни прекъсвания на ROM-BIOS 32
3.4. Функции на MS-DOS 38
Тема 4.Изграждане програми на асемблер 39
4.1. Адресация на паметта 39
4.2. Изписване на числа и букви 42
4.3. Дефиниране на променливи и области от паметта 43
4.4. Дефиниране на сегменти 45
4.4.1. Пълна форма на дефиниране на сегменти 45
4.4.2. Кратка форма (Опростени директиви за сегментация) 50
4.4.3. Изводи 53
4.5. Асемблиране и свързване на програмите 54
4.6. Turbo Debugger 55
Тема 5.Команди за прехвърляне на данни 56
5.1. Команди за прехвърляне на данни с общо предназначение 56
5.2. Команди за въвеждане-извеждане от порт 58

5.3. Команди за работа с адреси и указатели в паметта 59
5.4. Команди за преобразуване на данни 60
5.5. Команди за работа със стека 61
Тема 6.Инструкции за преход и цикъл 66
Тема 7.Аритметични операции с цели двоични числа 72
7.1. Представяне на числа със знак 72
7.2. Събиране на двоични числа без знак 74
7.3. Събиране на двоични числа със знак 75
7.4. Изваждане на двоични числа без знак 76
7.5. Изваждане на двоични числа със знак 78
7.6. Умножение на числа без знак 79
7.7. Умножение на числа със знак 81
7.8. Деление на числа без знака 82
7.9. Деление на числа със знак 83
7.10. Спомогателни команди за целочислени операции 84
7.10.1. Команди за преобразуване на типовете 84
7.10.2. Други полезни команди 85
Тема 8. Аритметични операции с двоично-десетични числа 86
8.1. Въведение в BCD числата 86
8.2. Аритметични действия с неопаковани BCD числа 87
8.2.1. Събиране на неопаковани BCD числа 87
8.2.2. Изваждане на неопаковани BCD числа 89
8.2.3. Умножение на неопаковани BCD числа 91
8.2.4. Деление на неопаковани BCD числа 93
8.3. Аритметични действия с опаковани BCD числа 94
8.3.1. Събиране на опаковани BCD числа 94
8.3.2. Изваждане на опаковани BCD числа 96
Тема 9.Логически команди и команди за изместване 97
9.1. Логически команди 98
9.2. Команди за изместване 101
9.3. Оператори за сравнение 105
Тема 10.Преобразуване на числата 107
Тема 11.Команди за обработка на низ от знаци 110
Тема 12.Видове адресиране 118
12.1. Директно адресиране 119
12.2. Индиректно адресиране 120
12.2.1. Адресиране посредством регистър 120
12.2.2. Адресиране спрямо база (базово адресиране) 121
12.2.3. Индексно адресиране 122
12.2.4. Базово-индексно адресиране 123
12.2.5. Базово-индексно адресиране с отместване 123
12.2.6. Изводи 124
Тема 13.Процедури и стекови операции 125
13.1. Процедури 126

13.1.1. Стек за адреси и стек за данни ... 126
13.1.2. Извикване на подпрограми (процедури) ... 127
13.1.3. Подпрограма като процедура с далечен преход (FAR) ... 130
13.2. Макроси ... 132
Тема 14.EXE и COM формат ... 138
14.1. Формат COM ... 138
14.2. Формат EXE ... 141
14.2.1. Въведение в EXE програми ... 141
14.2.2. Зареждане на EXE файл ... 143
14.2.3. Модели на паметта ... 144
14.3. Изводи ... 147
Тема 15.Работа с файлове ... 148
15.1. Функции за работа с файлове ... 149
15.2. Функции на MS DOS за директории ... 155
Тема 16.Програмиране на копроцесор ... 156
16.1. Дефиниране на реални числа ... 159
16.1.1. Define Doubleword (DD) ... 159
16.1.2. Define Quardword (DQ) ... 159
16.1.3. Define Tenbyte (DT) ... 160
16.2. Формати на данните при копроцесор 8087 ... 161
16.3. Команди за обмен на данни. ... 161
16.4. Целочислени аритметични команди ... 162
16.5. Функционален модел на копроцесора 8087 ... 165
16.5.1. Дума на състоянието (Status Word, SW) ... 166
16.5.2. Управляваща дума (Control Word, CW) ... 168
16.5.3. Работна среда и състояние на копроцесора ... 169
Тема 17. Връзка на асемблер с езиците от високо ниво ... 170
Заключение ... 172
Приложения ... 173
Приложение 1. Таблица на ASCII символите ... 173
Приложение 2. Скан кодове на клавишите ... 174
Приложение 3. Инструкции на процесорите IA-32 ... 175
Използвана литература ... 203

Вместо предговор

Изказваме благодарност към нашите студенти, пожелали да видят това издание!

Това ръководство за практикум ще е полезно на всички, които пристъпват към изучаване на асемблер.

Изложението е подбрано и подредено така, че да служи за информационно-методическа база при изучаване на този език.

Езикът асемблер е символно представяне на машинния език и е базов език на компютъра, непосредствено свързан с архитектурата на процесора. В настоящето издание се изследват Intel-съвместимите процесори. За програмиране могат да се използват пакетите masm, tasm и други. Усъвършенстването архитектурата на процесора развива паралелно и езика.

За усвояването на асемблер не е необходимо да се научат наизуст всички команди и директиви. За практическата работа е достатъчно разбирането на базовите концепции и идеи, лежащи в основата на езика. Детайлите, свързани с реализацията им, винаги могат да бъдат намерени в справочник на командите. Много по-важно е да се разбира какво място заема дадената команда в този справочник. Добре е да се знаят и целите, които са преследвали създателите на процесора, въвеждайки дадената команда в системата от машинни команди.

В действителност най-ефективната програма може да бъде написана единствено на асемблер, при условие, че се пише от квалифициран програмист. Но този процес е свързан с огромни трудови и времеви ресурси. Затова на асемблер основно се пишат програми, които осигуряват ефективна работа с апаратната част на компютъра и са с критични за време изпълнения или за заемана памет фрагменти от програми. На практика на асемблер се пише:

- всичко, което изисква максимална скорост за изпълнение на: основните компоненти на компютърните игри, ядрата на операционните системи в реално време и просто критични участъци от програми;
- всичко, което взаимодейства с външни устройства: драйвери, програми, работещи директно с портовете, звукови и видеоплатки;

- всичко, което използва напълно възможностите на процесора: ядрата на многозадачните операционни системи, DPMI сървърите и всякакви програми, превеждащи процесора в защитен режим;
- всичко, което тотално използва възможностите на операционната система: вирусите и антивирусите, защитите от несанкциониран достъп, програмите, обхождащи тези защити, и програмите, защитаващи се от дадените програми и много още друго.

Програмата на асемблер отразява основните особености на архитектиурата на процесора: организация на паметта, начини на адресация на операндите, правила за използване на регистрите.

Знанието на асемблер често пъти помога при компилирането на програми на други езици, защото то дава представата за това как в действителност функционира компютъра и какво става при изпълнението на командите на езика от високо ниво.

Тема 1. Въведение

Програмирането с помощта на асемблер е машинно програмиране. Процесорът се нуждае от инструкции като групи от битове, т. е. във форма на двоични числа. За съжаление такова програмиране в машинен код (обозначаван също като обектен код) не е ефективно, защото едва ли някой може да научи наизуст многото различни инструкции като групи от битове. Затова разработващите микропроцесори въвеждат английски съкращения за отделните инструкции, наричани мнемоника (мнемоничен код). Тази мнемоника трябва най-напред да се преведе в обектен код от програма за превод, наречена асемблер. Мнемоничният код също се означава като асемблер.

Една асемблерна програма се състои от инструкции в мнемоничен код, които са предназначени за процесора, и от директиви, представляващи указания за асемблера. Инструкциите се превеждат на обектен код, а директивите управляват процеса на превода.

1.1. Процесори 80x86 и Pentium

В това представяне са разгледани 8088, 8086, 80186 до 80486 (последните накратко означени 80x86) и Pentium на Intel. Процесорите 8086 до 80286 са 16-разредни, което значи, че те могат да обработват вътрешно данни с 16-разредна структура. Процесорите 80386, 80486 и Pentium са 32-разредни.

Процесорът 8088 има само 8-разредна магистрала за данни, затова трябва да чете 16-разредни числа с две обръщения към паметта, всяко към 1 байт, докато 8086 има 16-разредна магистрала за данни, т. е. при един процес запис или четене от паметта може да се обърне към 16-разредна дума данни.

Познаването на регистрите, с които разполага процесорът, е от голямо значение за програмирането посредством език, близък до процесора (асемблер). Ето защо ще ги обобщим по-долу.

За процесорите 8086 до 80286 най-напред може да се различат грубо две групи регистри: регистри за данни (AX, BX, CX, DX) и регистри за адреси. Адресните регистри се делят от своя страна на две групи регистри указатели (SP, BP, IP) и индексни регистри (SI, DI). Към адресните регистри принадлежат също и сегментните регистри (CS, DS, SS, ES).

Ето списък и описание на 16-разредни регистри:

AX (AH, AL) – Акумулатор
BX (BH, BL) – База
CX (CH, CL) – Брояч
DX (DH, DL) – Данни

Тези четири регистъра са предназначени преди всичко за съхраняване на данни. Понеже много често имаме работа с еднобайтов операнд, 16-разредните регистри AX до DX са разделени на по два 8-разредни регистъра: старши (H от higher) и младши (L от lower) регистър – например AH и AL.

Заедно с предназначението си като регистри за запомняне на данни повечето регистри за данни изпълняват и някои специални функции. Регистърът AX е централен регистър за умножение и деление. За извеждане към входно-изходен канал се използват регистрите AL и AX. Регистърът BX, наричан още базов регистър, е необходим за пресмятане на адреса при специален вид адресиране. Регистърът CX, наричан още броячен регистър, служи като броячен регистър при цикли в инструкции. Регистърът DX, наричан също регистър за данни, се използва за пресмятане на адреса при специален вид адресиране. Също така DX, свързан с AX, се използват при умножение и деление.

SP – Указател на стека
BP – Указател на базата
SI – Индексен регистър за операнд (източник)
DI – Индексен регистър за резултат (цел)

IP – Указател на инструкция

Указателят на стека SP показва началото на адресния стек и на стека за данни, които се намират в паметта и са необходими при работа с подпрограми. Указателят на базата BP се използва предимно при работа с подпрограми за установяване на локална база на стека за подпрограмата.

Регистърът за първични индекси SI и регистърът за целеви индекси DI служат за пресмятане на адрес при инструкции за низове.

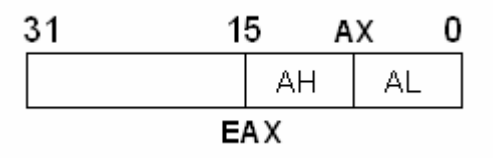
Указателят на инструкции IP показва адресите на следващите наредени в опашката инструкции (това го различава от брояча на инструкции, който постоянно показва следващата инструкция за изпълнение). Само при инструкция за преход, респективно за разклонение в програмата, указателят на инструкции показва адреса на следващата за обработка инструкция.

CS – Програмен сегмент (сегментен регистър на кода, който дефинира текущия сегмент, съдържащ машинните инструкции – кода на изпълняваната програма)
DS – Сегмент за данни (сегментен регистър на данните, който дефинира сегмент в паметта за данни на изпълняваната програма)
SS – Сегмент за стека (сегментен регистър на стека, който дефинира сегмент в паметта, в който се изпълняват стекови операции)
ES – Допълнителен сегмент (допълнителен сегментен регистър за дефиниране на още един сегмент с данни, достъпен за текущата програма)

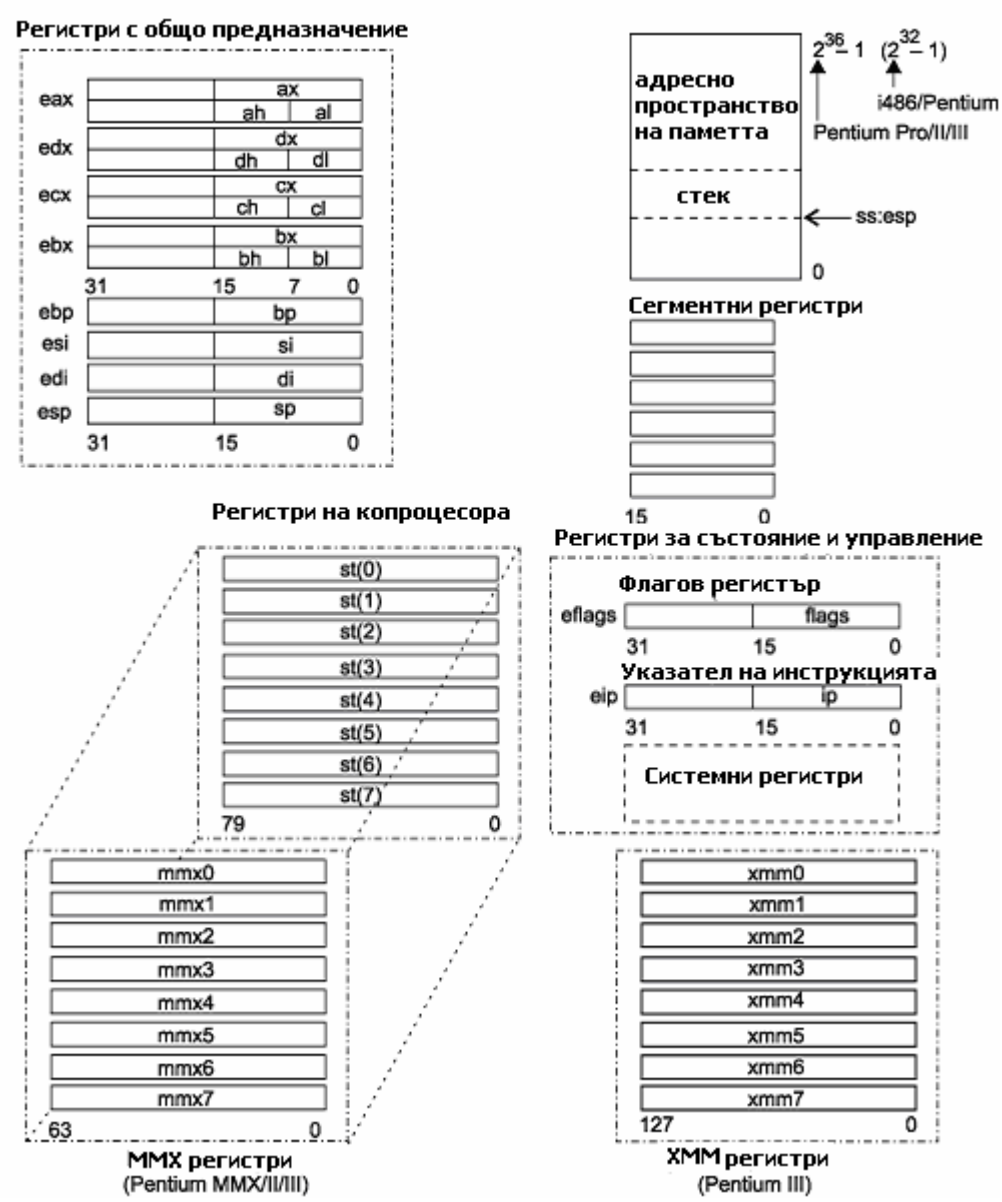
Флагов регистър

Битовете на флаговия регистър EFLAGS / FLAGS (32 / 16) имат определено функционално предназначение. Те отразяват особеностите на резултата от изпълнението на аритметичните или логическите команди, което прави възможно да се анализира състоянието на изчислителния процес и съответното реагиране с помощта на командите за условен преход и извикване на подпрограми.

При 80386, 80486 и Pentium всички регистри освен сегментните може да се разширят до 32-разредни (например EAX е 32-разредната версия на регистър AX). Също така има два допълнителни сегментни регистъра (FS и GS). Старшите 16 бита на разширените девет 32-битови регистри с общо предназначение не се специфицират в отделни регистри, както младшите 16-битови подрегистри (AX, BX….).



Фигура 1. Регистър EAX



Фигура 2. Програмен модел на архитектурата IA-32 на процесорите Intel

1.2. Сегментен регистър и формирането на адреса

Процесорите 8086 и 8088 работят с 20-разредна адресна магистрала, която може да активира 1MB = 1024 KB = 1024 x 1024 B = 2^{20} B от капацитета на паметта. 20 бита обаче не са точно необходимото количество битове при обикновената схема на образуване на числа като степени с основа 2: така при инструкции, които се нуждаят от адрес, трябва да се използват 2 байта, при което 4 бита остават излишни. Затова Intel разработва друг начин за образуване на адреса, при който всеки адрес в паметта се получава чрез сумиране на един базов адрес и съответно отместване. Базовият адрес се запомня в един сегментен регистър, а отместването се намира в един от указателите или индексните регистри.

Процесорът образува 20-битовия адрес, с който трябва да се обърне към паметта – т. н. "физически адрес", като умножава базовия адрес по 16 (=10H), което в двоична бройна система съответства на преместване с 4 бита наляво, и прибавя отместването.

Типичният при асемблер запис на адрес е, при който базовият адрес и отместването, разделени с двоеточие, са дадени в шестнайсетична система:

ABCD: VWXY

т. е. базов адрес: отместване

Пример:

0456: 0123 –> 0456 x 10 + 0123 = 04560 + 0123 = 04683

1.3. Формати на данните

При работа на микропроцесора, устройството за работа с плаваща запетая (Floating Point Unit– FPU) работи паралелно с аритметико-логическото устройство (Arithmetic and Logic Unit– ALU) и осигурява изпълнението на аритметични инструкции за разнообразни формати на данните. При обръщение към данните в паметта процесорът ги интерпретира по няколко начина в зависимост от инструкцията, логиката на програмата и др., като някои от форматите се поддържат от CPU, а други – от FPU.

1.3.1. Цели данни без знак

Използват се три формата на цели данни без знак – с дължина 1 байт (Byte), 2 байта (Word) и 4 байта (Double Word), при което най-младшият значещ бит е с индекс 0, а най-старшият значещ бит – съответно с индекс 7, 15 или 31. Обхватът на представимите стойности очевидно е 0–255 (Byte), 0–65535 (Word) и 0–4294967295 (Double Word).

1.3.2. Цели данни със знак

За представяне на цели данни със знак се използва обикновено допълнителен машинен код (т. е. най-старшият значещ бит е знаков разред: 0 – положителен, а 1 – отрицателен). Обхватът на представимите стойности съответно е от –128 до 128 (Byte), –32678 до 32678 (Word) и –2147483648 до 2147483648 (Double Word). FPU поддържа и 64 битов формат на цяло число със знак, получаван като резултат от умножение или деление на 32-битови операнди.

1.3.3. Десетични (BCD) данни

Десетичните данни BCD (Binary Coded Decimal) се представят без знак в непакетиран вид (една BCD цифра в един байт) и пакетиран вид (две BCD цифри в един байт), като кодирането на цифрите е в ASCII (0 –> 30h, 9 –> 39h). FPU поддържа 80-битово (10 байтово) цяло десетично число със знак (в пряк машинен код) с общо 18 BCD цифри (старшият байт се използва като знаков).

1.3.4. Данни с плаваща запетая

Данните с плаваща запетая (Floating Point) са с пряк машинен код по стандарта IEEE с три дължини – единична (4 байта), двойна (8 байта) и разширена (10 байта), като знакът на мантисата е най-старшият значещ бит. Тези данни са поддържани само от FPU.

1.3.5. Низове

Поддържат се 4 вида низове с произволна обща дължина и формати Byte, Word, Double Word и Bit.

1.3.6. ASCII данни

За кодиране на един символ се използва ASCII код – единственият вътрешен код на процесорите x86.

1.3.7. Указатели

Поддържат се указатели (данни за адресна аритметика) с две дължини – NEAR и FAR.

Тема 2. Програма DEBUG

2.1. Команди на програмата DEBUG

Програмата DEBUG е предложена от Microsoft. Дебъгерът се намира резидентно в оперативната памет като заделя в паметта работна област за въвеждане на програми във вид на машинни кодове и команди на езика асемблер.

Сегментният адрес на тази работна област дебъгерът записва в сегментните регистри CS, DS, ES, SS (размерът на работната област е не повече от 64K).

В регистърът на отместването IP се записва шестнайсетичното число 100, защото пред всяка изпълнима програма в оперативната памет се формира така наречения PSP – Префикс на Програмния Сегмент (Prefix Segment Program) – област с дължина 256 байта (100h = 256), съхраняваща описанието на формата на програмата, ред с параметри, които въвежда потребителя, извиквайки програмата и командата, която приключва изпълнението на програмата, предавайки управлението на MS DOS.

В регистъра на указателя на върха на стека SP се записва отместването относно сегментния адрес в CS (SP = FFEE), а в стека се записва нулева дума (2 байта 00 00).

Фигура 3. Съдържание на регистрите при работа с програмата DEBUG

Най-напред ще опишем в сбита форма действието на най-важните команди на програмата за настройка DEBUG.

Необходимите при командите адреси на паметта се дават в следния формат:

Сегментен адрес: отместване

Въвеждането и извеждането на числа става винаги в шестнайсетична система, без при това да се добавя "H" след числото.

Извикването на програмата за настройки и програмата, която ще се тества става с DEBUG *име_на_програма*.exe.

Ако не разполагаме с асемблер и линкер, по-малки програми може да се редактират и да се асемблират до изпълним код посредством програмата за настройка DEBUG. За това приложение намират командите –A (Assemble) и –W (Write). Това обаче е възможно само при програми, които се състоят от един сегмент, както .COM програмите, понеже при работа с DEBUG не са възможни деклариране и присвояване на сегментите с ASSUME.

За създаването на програма се започва с командата *a* и след това се въвеждат ред по ред инструкциите. Накрая се натиска два пъти клавиша за въвеждане. Така се напуска режима на асемблиране.

Сега с –N (Name) се указва името на файла, под което трябва да бъде запомнена програмата. В регистъра BX:CX трябва да се намира броят на байтове за записване. От адресите, съпровождащи инструкциите, би трябвало да се види колко байта са необходими на програмата. След това програмата

се запомня с –W под предварително избраното име. Накрая програмата DEBUG потвърждава броя на запомнените байтове.

Пример:

```
–a
XXXX:0100      mov ah, 01
XXXX:0102      int 21
XXXX:0104      jmp 100
XXXX:0106      [Enter]
–rcx
CX 0000
:8
–n test.com
–w
Writing 0008 bytes
```

Най-важните команди на програмата за настройка DEBUG са:

- Команда –A (Assemble)

  - –A *адрес*

Асемблира инструкциите и пренася кода на инструкцията директно в паметта от зададения *адрес*. Всички цифрови стойности трябва да бъдат дадени в шестнайсетичен формат.

  - –A

Ако командата –A се използва без *адрес*, програмата започва при адрес 100H

- Команда D (Dump)

  - –D *адрес1, адрес2*

Тази команда извежда на екрана съдържимото на паметта (RAM/ ROM), като започва с първия посочен *адрес1* до втория посочен *адрес2*. Ако вторият адрес не е посочен – извежда 128 байта, по 16 байта на всеки ред – в шестнайсетичен и в символен вид.

Всички числа се въвеждат и извеждат от дебъгера само в шестнайсетична бройна система.

Адресът се посочва във вида сегмент: отместване. Ако сегментът не е посочен, тогава се подразбира този, който в дадения момент е записан в регистъра CS.

- –D *адрес*

Тази команда показва съдържанието на 128 байта от зададения *адрес* в шестнайсетична форма, а в отделен блок показва съответните ASCII знаци.

- –D

–D има същото действие, но като стартов адрес се възприема моментното съдържание на регистър DS.

Пример:

–D 200 –> започва при DS:200

За да се прегледа произволна клетка от паметта (RAM или ROM) извън работната област на дебъгера, в командата –D следва да се посочи както сегмента, така и отместването.

Така например в Таблицата на Данните BIOS с начален адрес 0040:0013 и до адрес 0040:0014 се съхраняват 2 байта, които съдържат обема на оперативната памет на вашия компютър.

```
–D 40:13, 14
0040:0010       80 02
```

Понеже байтовете в думите са с разменени места, 80 02 означава шестнайсетично число 0280h = 2*256 + 8*16 = 512 + 128 = 640K.

На адрес FE00:0000 в ROM паметта се съхранява информация за вашия компютър и за неговата Базова Система за Вход/ Изход (BIOS):

```
-D FE00:0
FE00:0000  FF  FF  FF ... FF  FF  FF  FF  8A 49  42   .............IB…….
FE00:0010  4D C3  FF ... FF  FF  FF  FF  FF FF  FF  M.....……
FE00:0020  43  6F  70 ...  20  28   43  29  20  31  39   Copyright (C) 19
FE00:0030  38  35  2D ... 68  6F  65  6E  69  78  20   85-1990 Phoenix
FE00:0040  54  65  63  ... 69  65  73  20  4C 74   64   Technologies Ltd
```

- Команда –E (Enter)

- –E *адрес*

Тази команда показва съдържанието на избрания *адрес* и чака въвеждане. Ако се въведе 1 байт, старата информация на избраната клетка от паметта се унищожава. При натискане на клавиша за интервал се появява съдържанието на следващата клетка от паметта и то може да се променя. Ако не трябва да се променя съдържанието на една клетка от паметта, трябва само да се натисне клавиша за интервал.

Чрез натискане на клавиша за въвеждане командата Enter се напуска.

Пример:

–E CS:100

- Команда –G (Go)

  - –G *адрес*

Тази команда изпълнява програмата до зададения *адрес* (break point). Ако не е зададен адрес се изпълнява цялата програма.

- Команда –L (Load)

  - –L *адрес*

Тази команда зарежда указания в BX:CX брой байтове на един файл (чието име се задава при извикване на настройващата програма или с командата N) от *адрес* в оперативната памет. Ако не е зададен адрес зареждането започва при CS:100 = базовия адрес в CS.

  - –L *адрес устройство начало брой*

Тази команда зарежда от зададеното външно запомнящо *устройство* (A = 0, B = 1, C= 2) като започва от сектор с номер *начало*, съответния *брой* сектори от зададения *адрес* в основната памет.

Пример:

–L CS:100
–L CS:100 2 0A 07

- Команда –P (Proceed)

  - –P

Тази инструкция работи като инструкция T, но дадена програма (call), цикъл (loop) или програмно прекъсване (INT) се третира като една единствена инструкция и се изпълнява автоматично до край.

- –P = *адрес брой*

Тази команда изпълнява зададен *брой* инструкции започвайки от зададения *адрес*. Ако не се зададен адрес се стартира от адреса на инструкцията в дадения момент. Ако не се зададе брой се изпълнява само една инструкция.

Пример:

–P = 100 2

- Команда –R

- –R *име_на_регистър*

Тази команда показва съдържанието на регистъра и извежда двоеточие. След двоеточието може да бъде въведено ново съдържание на регистъра. Ако R е дадена без име на регистър, се показват съдържанията на всички регистри и флагове.

- -R

Тази команда показва флаговете и също може да се променят съдържанията им.

- Команда –S (Search)

- –S *област списък*

Тази команда претърсва зададената *област* от паметта и показва всички адреси, които се съдържат в *списък.* Списъкът се състои от един или няколко байта, разделени чрез интервали или запетайки.

Пример:

–S CS:100 102 CD

- Команда –T (Trace)

- –T *брой*

Тази команда изпълнява зададения *брой* инструкции на отделни стъпки и показва след всяка стъпка съдържанието на всички регистри и флагове, както и следващата инструкция. Ако е въведено само T се изпълнява точно една инструкция.

Пример:

–t 2

- Команда –U (Unassemble)

  - –U *адрес*

Тази команда деасемблира 16 байта започвайки от зададен *адрес*. Ако е въведен само –U деасемблира от дадения адрес като текущото съдържание на CS:IP.

Пример:

–u 100

- Команда –Q (Quit)

Командата –Q служи за напускане на настройващата програма и връщане в MS DOS.

- Команда –W (Write)

  - –W *адрес*

Тази команда записва върху носител на данни въведения в BX:CX брой байтове от файла, с който току що е работено, като започва от зададения *адрес*. Ако не е зададен адрес пренасянето на данни започва от адрес CS:100.

  - –W *адрес устройство старт брой*

Тази команда записва върху зададеното външно запомнящо *устройство* (0=А, 1=Б и т. н.) от зададения *адрес* на паметта зададения *брой* сектори, започвайки от сектор с номер *старт*. Ако не е зададен адрес, започва се при CS:100.

2.2. Пример

Въведете в режим на асемблиране следната програма:

```
–A
xxxx:0100 MOV AX, [0110] Зарежда в AX съдържимото на клетка с адрес DS:110
xxxx:0103 ADD AX, [0112] Събира с AX съдържимото на клетка с адрес DS:112
xxxx:0107 MOV [0114], AX Резултатът от AX се записва на адрес DS:114
xxxx:010A RETF Възврат
xxxx:010B <Enter>
```

Непосредствената адресация твърде рядко се използва в програмирането. Обикновено данните се прочитат от клетки в паметта на сегмента данни и резултатът се записва в тези клетки.

При работа с дебъгера кодовият сегмент и сегмента за данни започват от един и същ сегментен адрес (CS=DS=SS). Затова ще разположим входните данни 123h и 25h след кода на командите.

Деасемблийте тази програма:

```
–U 100, 10A
xxxx:0100 A11001      MOV AX, [0110]
xxxx:0103 03061201    ADD AX, [0112]
xxxx:0107 A31401      MOV [0114], AX
xxxx:010A CB          RETF
```

Зареждаме данните:

```
123h на адрес DS:110
25h на адрес DS:112
00h на адрес DS:114(за резултата)


–E DS:110 23 01 25 00 00 00
–D CS:100

xxxx:0100 A1 10 01 03 06 12 01 A3 14 01 xx xx xx xx  КОМАНДИ
xxxx:0110 23 01 25 00 00 00 xx xx xx xx xx xx xx xx  ДАННИ
xxxx:0120 xx xxx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
```

Разглеждаме съдържимото на регистрите и изпълняваме програмата. Ако в командата присъства [адрес], дебъгерът показва съдържимото на клетката според посоченото отместване относно DS.

```
-R
AX=0000  BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=2593  ES=2593 SS=2593 CS=2593 IP=0100   NV UP EI PL NZ NA PO NC
2593:0100 A11001       MOV    AX,[0110]           DS:0110 = 0123
```

Съдържимо на клетките на зададения адрес в AX

```
-T
AX=0123  BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=2593  ES=2593 SS=2593 CS=2593 IP=0103   NV UP EI PL NZ NA PO NC
2593:0103 03061201     ADD    AX,[0112]           DS:0112 = 0025
```

Числото от клетката на зададения адрес да се събере с AX

```
-T
AX=0148 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=2593 ES=2593 SS=2593 CS=2593 IP=0107   NV UP EI PL NZ NA PE NC
2593:0107 A31401       MOV    [0114], AX          DS:0114=0000

-T
AX=0148 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=2593 ES=2593 SS=2593 CS=2593 IP=010A NV UP EI PL NZ NA PE NC
2593:010A CB           RETF
```

Нека видим резултата на адрес DS:114

```
-D DS:0110
2593:0110  23 01 25 00 48 01 xx xx xx xx xx xx xx xx xx xx
2593:0120  xx xx  xx  xx xx xx xx xx xx xx xx xx xx xx xx xx
```

Изпълнение на примерна програма

След изпълнение на машинната команда RETF (Return Far) програмата излиза от работната област на дебъгера. За последното говори промяната на сегментния регистър на командите: CS.

```
–T
AX=0148 BX=0000 CX=0000 DX=0000 SP=FFF2 BP=0000 SI=0000 DI=0000
DS=2593 ES=2593 SS=2593 CS=0000 IP=0000 NV UP EI PL NZ NA PE NC
0000:0000 68    DB      68
```

За да се върнем в работната област е необходимо да възстановим регистрите.

Възстановените стойности на регистрите са:

AX = BX = CX = DX = 0
CS = DS = ES = SS = Сегментен адрес на работната област на дебъгера
IP = 100
SP = FFEE

Тема 3. Прекъсване функциите на ROM-BIOS и MS-DOS

Често пъти програмата взаимодейства с потребителя, получавайки от него данни чрез клавиатурата и извеждайки резултата от работата си на екрана (конзола: клавиатура и монитор). За да изпълни подобна операция асемблер използва възможностите на компютъра (прекъсвания на BIOS) и операционната система, в чиято среда работи програмата.

Под прекъсване се разбира временното спиране на хода на текущата програма. Различават се хардуерни (апаратни) и софтуерни (програмни) прекъсвания. Хардуерните прекъсвания се появяват асинхронно, което значи, че главната програма може да бъде прекъсната в произволно място. Софтуерните прекъсвания се задействат от една инструкция (INT), намираща се в програмата, която има действие, аналогично на извикване на подпрограма.

Доколкото 8088, 80x86 и Pentium имат само по два входа за прекъсване, допуска се включване максимално до 256 устройства, изискващи различни условия за прекъсване. Всяко от тези устройства изпраща по една линия, обща за всички устройства, един сигнал за прекъсване и доставя посредством магистралата за данни един номер на прекъсване. От този номер на прекъсване процесорът образува адрес в паметта, в който номерът е умножен по 4. При 256 възможни различни номера на прекъсване това означава, че при всеки компютър 8088 и 80x86 първият килобайт ($2^8 * 2^2 = 2^{10} = 1$KB) на паметта е необходим за целите на прекъсването (адрес 0000H до 003FFH).

На така пресметнатите точки на преход се намират 4-байтови адреси (сегментен адрес и отместване) – т. нар. вектори на прекъсване, които указват първата инструкция на съответната програма за обработка на прекъсванията. Образуваната от тези адреси таблица се нарича още таблица на векторите за

прекъсване. Протичането на обработката на прекъсване се състои от следната последователност от действия:

- На дадено място на главната програма се появява хардуерно или софтуерно прекъсване. Обработваната в момента инструкция се довежда докрай.
- Номерът на прекъсване n се умножава по 4, адресът на следващата инструкция на главната програма (адрес за обратен преход) и флагът се съхраняват в стека.
- С помощта на пресметнатия адрес процесорът се обръща към таблицата за вектори за прекъсване.
- В тази таблица процесорът намира адреса за преход в програмата за обработка на прекъсването n (ISRn). На края на тази програма се намира инструкция за обратен преход след прекъсване IRET (Return from Interrupt).
- Инструкцията IRET прочита от върха на стека и зарежда съответно в кодовия сегмент (CS), указателя на инструкциите (IP) и флаговия регистър с адреса на връщане, запазени в стека автоматично от процесора в началото на прекъсването. С това процесорът може да продължи протичането на главната програма след мястото на прекъсване.

Първите номера на прекъсване са заети от Intel:

- Прекъсване 0 – Препълване след деление на 0
- Прекъсване 1 – Обработка на отделни стъпки
- Прекъсване 3 – Точка на прекъсване (Breakpoint)
- Прекъсване 4 – Прекъсване при препълване.

Следващите прекъсвания са софтуерни. Някои от номерата на прекъсване са уговорени от IBM за управление на периферни устройства (наричат се още функции на BIOS за работа с конзолата):

- INT 05H – Извеждане съдържимото (печатане) на екрана
- INT 09H – Въвеждане от клавиатурата
- INT 10H – Управление на дисплея (видеосървис)
- INT 13H – Дискови операции за запис и четене
- INT 14H – Управление на вход/изход през комуникационния порт RS232
- INT 16H – Въвеждане от клавиатурата в регистъра AX
- INT 17H – Извеждане на данни на принтера
- INT 1AH – Извеждане на текущите време и дата
- INT 1FH – Адрес на таблицата с графични символи. По-точно до символите с кодове от 128 до 255. В 1K от таблицата се съдържат по осем

байта за всеки символ. Пряк достъп в графичен режим има само до първите 128 ASCII символи (от 0 до 127).

Други са уговорени от Microsoft за специални функции на операционна система (наричат се още функции на MS DOS за работа с конзолата):

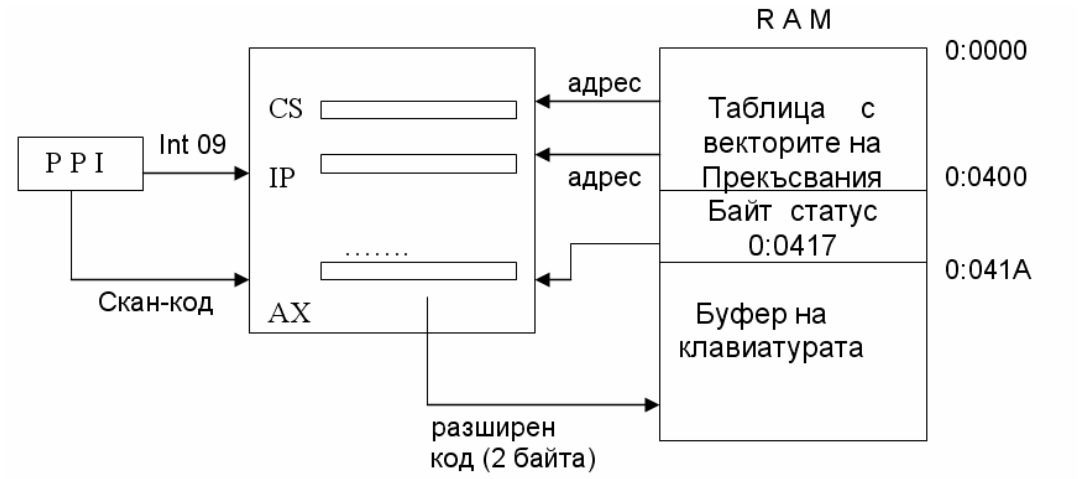
- INT 21H.

3.1. Въвеждане от клавиатурата

При натискане на клавиш процесорът на клавиатурата (Intel 8042) изпраща 2 байта код в порт A (адрес 60h) на контролера на програмируемия интерфейс (PPI, Intel 8255).

Натискане на клавиш <Z>

Контролерът PPI изпраща на централния процесор (CPU) сигнал 9-то прекъсване (INT 09). По този сигнал в CS:IP се записва деветия вектор от Таблицата с вектори – 4 байта с адрес на процедурата, обработваща това прекъсване. Тя може да бъде адрес на стандартна процедура на BIOS-а, записана в ROM или адрес на резидентна програма – драйвер на клавиатурата, зареден в RAM.



Фигура 4. Изпращане на сигнал INT 09

3.1.1. Процедурата за обработка на INT 09

Процедурата за обработка на INT 09 изпълнява:

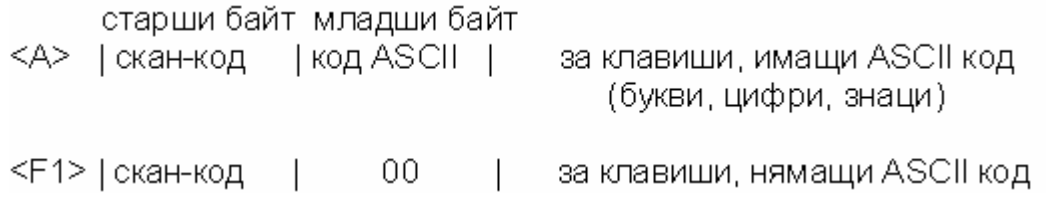
- Чете байт статуса от Таблицата на BIOS на адрес 0:0417, който отразява текущото състояние на клавишите-превключватели (Caps Lock, Num Lock, Scroll Lock) и на модификаторите (Shift, Alt, Ctrl).

Битове	7	6	5	4	3	2	1	0
	Ins	NumLock	CapsLock	ScrLock	Alt	Ctrl	L-Sh Ляв	R-Sh Десен Shift

Фигура 5. Байт статус от Таблицата на BIOS

Например байт статуса 0C = 0000 1100 означава, че в дадения момент едновременно са натиснати <Ctrl> и<Alt>.

- Според таблицата на разположение на клавиатурата (записана в ROM или заредена с драйвер в RAM) се определя байта на кода на символа (код ASCII). С отчитане на байта на статуса и номера на натиснатия клавиш (скан-кода) в регистъра AX се формира двубайтов разширен код.

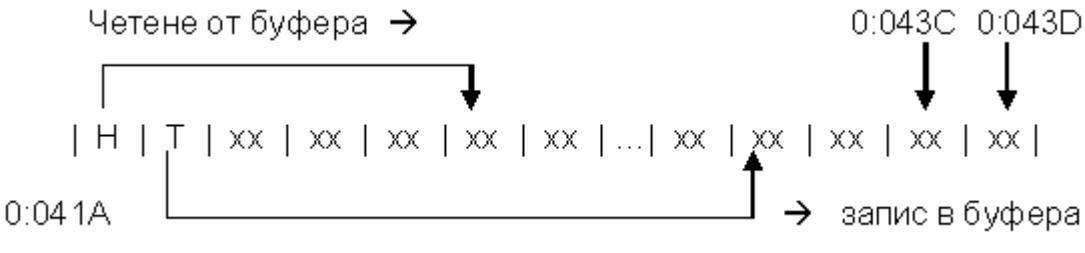


Фигура 6. Двубайтов разширен код при извикване на прекъсване INT 09

Скан-кодът обикновено е номера на натиснатия клавиш, но при едновременно натискане на някои клавиши скан-кода се изчислява по определени правила.

- Двубайтовият разширен код постъпва в буфера на клавиатурата – неголяма област вътре в BIOS таблицата (32 байта за 15 натискания).
- В първата клетка на буфера с адрес 0:041A се съхранява отместването относно началото на BIOS таблицата (0:0400). Това е указател към "главата" на буфера (Head, H). Всяка програма, приемаща въвеждане от клавиатурата чете от главата на буфера поредните 2 байта разширен код. При това указателят към главата се отмества с 2 байта напред (отместването в клетката 0:041A се увеличава с 2).

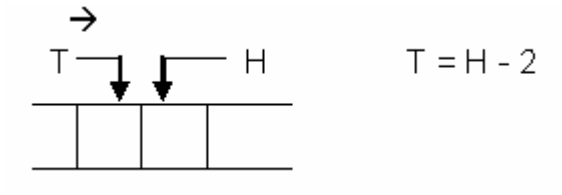
В клетката 0:041C е записано отместването за "опашката" на буфера (Tail, T). Разширеният код, постъпващ в буфера, се записва в опашката на буфера. При това напред с 2 байта се отмества указателя към опашката.



Фигура 7. Четене и запис в буфера на клавиатурата

Буферът е зациклен. След запълването на клетките 0:043C, 0:043D отново се запълват 0:041E, 0:041F и т. н.

- При възникване на ситуацията



(когато програмата не успява да прочита от буфера постъпващите там кодове) възниква препълване, за което съобщава звуков сигнал.

- Програмата, запитваща въвеждане от клавиатурата, е длъжна да очисти буфера, за да не прочита от него останали там от по-рано въведени символи. За това е достатъчно да се изравнят указателите, като отместванията, записани в клетките 0:041A и 0:041C с еднакви.

3.1.2. Буфер на клавиатурата

Въвеждането на цифри и малки (долен регистър) латински букви поражда скан-код, съвпадащ с поредния номер на клавиша. Клавишите се номерират подред, започвайки от цифровите.

№1	2	3	4	5	…	9	10	11	12	13	14
~ `	! 1	@ 2	# 3	$ 4		* 8	(9	) 0	_ –	+ =	←

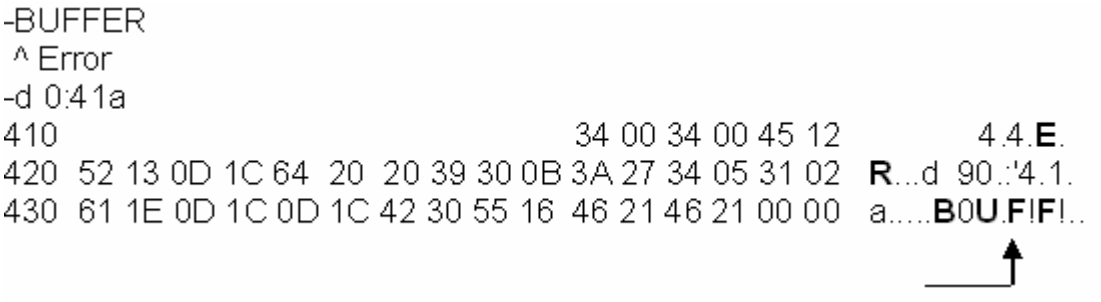
Таблица 1. Скан-кодове на клавишите от 1 до 14

№15	16	17	18	19	…	23	24	25	26	27	28
Tab	Q	W	E	R	…	I	O	P	{ [	}]	Ent

Таблица 2. Скан-кодове на клавишите от 15 до 28

И така нататък.

Нека въведем от клавиатурата думата “BUFFER” с главни букви (заедно с клавиша Shift), а после отново командата –D.



Фигура 8. Буфер на клавиатурата при въвеждане на “BUFFER”

Адрес	Байтове	Код ASCII	Символ	Скан-код
0:0434	0D 1C	0Dh=13	Enter	1Ch=28
0:0436	42 30	42h=66	‘B’	30h=48
0:0438	55 16	55h=85	‘U’	16h=22
0:043A	46 21	46h=70	‘F’	21h=33
0:043C	46 21	46h=70	‘F’	21h=33
0:041E	45 12	45h=69	‘E’	12h=18
0:0420	53 13	53h=83	‘R’	13h=10
0:0422	0D 1C	0Dh=13	Enter	1Ch=28

Таблица 3. Разширените кодове при въвеждането на “BUFFER”

Продължението при въвеждане след клетките 0:043C – 0:043D се записва в клетките 0:041E – 0:041F и по-нататък (в кръг).

Сравнението на разширените кодове на малките и главните букви показва, че клавиша <Shift> променя ASCII кода, а скан-кодовете на клавишите не се променят.

3.1.3. Прекъсване 16h

Съществува още един начин за изследване на разширените кодове – това е прекъсването 16h. Когато програмата очаква въвеждане от клавиатурата, тя изпраща това прекъсване, записвайки 00 в регистър AH (номера на функцията на това прекъсване). След въвеждането на един символ (без натискане на Enter) неговият разширен код се записва в регистър AX (AH– скан-кода, AL– ASCII кода) и може да бъде прочетено директно от там.

Прекъсване	Функция	Въвеждане	Извеждане
INT 16h	00h –> Четене (очакване) на натикнатия клавиш	AH = 00h	CF=0 –> AX – разширен код
	01h –> Проверка за символ в буфера на клавиатурата	AH = 01h	CF=0 –> AX – разширен код

Нека стартираме DEBUG и да въведем следната програма:

```
–a
0DA0:0100 MOV AH, 00
0DA0:0102 INT 16
0DA0:0104 JMP100
0DA0:0109 NOP
0DA0:010A

–u 100, 109
0DA0:0100 B400      MOV AH, 00
0DA0:0102 CD16      INT 16
0DA0:0104 EBFA      JMP 0100
0DA0:0106 90        NOP
```

Тази процедура реализира посимволно въвеждане от клавиатурата без натискане на клавиша <Enter> и без изобразяване на въвеждания символ на екрана (режим No Echo).

Нека да я изпълним до командата JMP. Нека да въведем командата “G 104” и натиснем<Enter>. В началото на следващият ред промптът няма да се появи. Програмата чака въвеждането на един символ. Нека натиснем клавиша <A> (Без Enter).

Извеждаме на екрана съдържимото на регистрите след INT 16h.

AX=1E61 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0DA0 ES=0DA0 SS=0DA0 CS=0DA0 IP=0104 NV UP EI PL NZ NA PO NC
0DA0:0104 EA0001A00D JMP 0DA0:0100

След въвеждане на малкото латинско ‘а’ AH = 1Eh = 30 (номер на клавиша)

AL=61h = 97 ASCII код

Ще изпълним командата за преход към началото на програмата JMP 0DA0:0100:

–t
AX=1E61 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0DA0 ES=0DA0 SS=0DA0 CS=0DA0 IP=0100 NV UP EI PL NZ NA PO NC
0DA0:0100 B400 MOV AH, 00

След това повтаряме процедурата, проверявайки разширените кодове на комбинациите <Shift>+<A> (латинско главно ‘A’), <Ctrl>+<A>, <Alt> +<A>, <F1>, <F2>:

–g 104

<Shift>+<A>
AX=1E41 AH = 1Eh = 30 номер на клавиша <A>
AL = 41h = 65 код на ‘A’ главно

<Ctrl>+<A>
AX=1E01 AH = 1Eh = 30
AL = 01h =1 ASCII код на управляващия символ ^A

<Alt>+<A>
AX=1E00 AH = 1Eh = 30 чист номер на клавиша
AL = 00

<F1>
AX=3B00 AH = 3Bh = 59
AL = 00

<Shift>+<F1>
AX=5400 AH = 54h = 84
AL = 00

За изход от безкраен цикъл на програмата натискаме <Q>.

3.1.4. Изводи

- Натискането на клавишите-модификатори <Shift>, <Alt>, <Ctrl> съвместно с клавиши, имащи ASCII код, променя този код:

<Shift>+<Буква> код ASCII+/–32
<Shift>+<Цифра> код ASCII+/–16
(горен/долен регистър в зависимост от <CapsLock>)

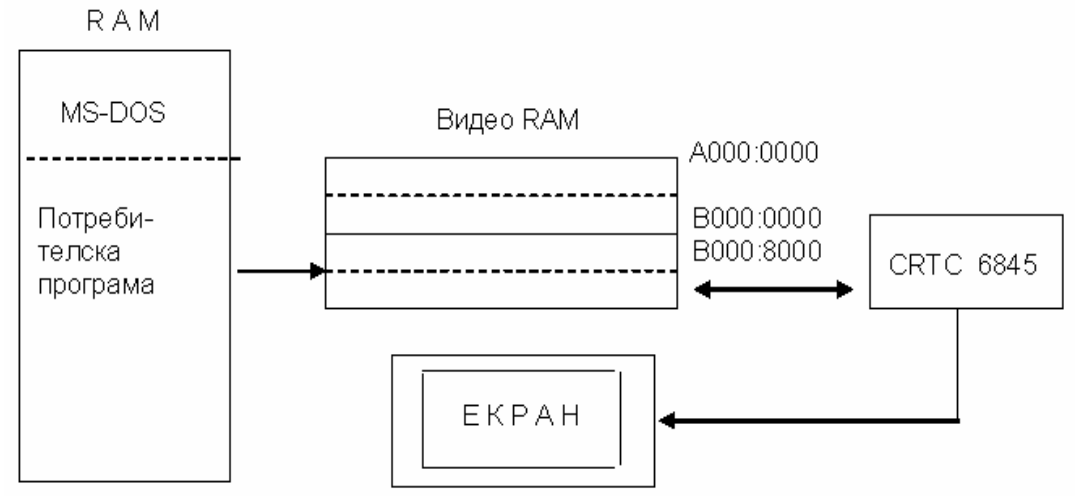
<Ctrl>+<Буква> код ASCII – 96
<Alt >+<Буква> код ASCII = 0

Скан-кодът на клавиша остава непроменен.

- Натискането на модификаторите <Shift>, <Alt >, < Ctrl > едновременно с клавиши, не генериращи ASCII код, променя техния скан-код.

3.2. Извеждане на екрана

Микросхемата Motorola 6845 (контролерът на електронно-лъчевата тръба, Cathod Ray TubeController, CRTC) постоянно сканира участък от RAM върху платката на видеоадаптера (видеопаметта) и изобразява информацията на екрана. Информацията във видеопаметта се записва от MS-DOS или от потребителска програма.



Фигура 9. Схема на процеса извеждане на екрана

Например, прочитайки разширения код на символите от буфера на клавиатурата, програмата отхвърля старшия байт на скан-кода, формира нов двубайтов код (ASCII код + атрибут) и го изпраща във видеопаметта. По такъв начин, въвеждайки символи от клавиатурата, ние ги виждаме на едно или друго място на екрана в съответен цвят (режим Echo).

В текстов режим във видеопаметта с два байта се кодира всяко знакомясто на екрана. В различните режими броя на редовете на екрана и броя на знакоместата в реда е различен.

Режим	**Знакоместа**	**Монохромно изображение**
0	**40 x 25**	**B&W**
1	**40 x 25**	**16 цвята**
2	**80 x 25**	**B&W**
3	**80 x 25**	**16 цвята**

Стандартен текстов режим за адаптерите EGA/VGA е третия.

Двата байта на кода във видеопаметта представляват байт с ASCII кода на изобразявания символ (младшия байт) и байт на атрибута. Байтът на атрибута със свой бит кодира яркостта (I) и цвета на символ, и фон на основните тонове:

I – Intensity (яркост)
G – Green (зелен)
B – Blue (син)
R – Red (червен)

	Фон			Символ			
*	R	G	B	I	R	G	B

Таблица 4. Байт на атрибута

Старшият (седмият) бит на атрибута означава мигане на символа. Обаче, чрез директно обръщение към портовете на микросхемата CRTC, може да се установи режим, при който този бит ще означава яркост на фона (16 цвята за символи + 16 цвята за фон).

По този начин, старшата шестнайсетична цифра на байта на атрибута означава цвят на фона (+ мигане), младшата цифра – цвят на символите.

RGB	№	Color	Цвят	RGB	№	Color	Цвят
0000	0	Black	Черен	1000	8	Dark gray	Тъмно сив
0001	1	Blue	Син	1001	9	Light blue	Св. св. син
0010	2	Green	Зелен	1010	A	Light green	Св. зелен
0011	3	Cyan	Тъмно син	1011	B	Light cyan	Св. син
0100	4	Red	Червен	1100	C	Light red	Св. червен
0101	5	Magenta	Лилав	1101	D	Light magenta	Розов
0110	6	Brown	Кафяв	1110	E	Yellow	Жълт
0111	7	Gray	Сив	1111	F	White	Бял

Таблица 5. Кодиране цвета на символите

Атрибутът 74h означава червено (4) на сив фон (7), т. е. 74h + 80h = F4h – мигане на червен символ на сив фон.

Позицията на символа на екрана се определя с отместването на съответстващите два байта на кода относно началото на видеопаметта. В текстов режим видеопаметта започва от адрес B800:0000 (или B000:8000)

80 знакоместа за всеки ред се кодират с по 160 байта, затова за символ в ред R и стълб C отместването от началото на видеопаметта може да се изчисли по формулата:

Offset = (R – 1)*160 + (C – 1)*2

Така за 5-ти ред и 10-ти стълб получаваме отместване 292h.

Offset = (5 – 1)*160 + (10 – 1)*2 = 658 = 292h.

На адрес B800:Offset първо следва да се запише байта на ASCII кода (младшия байт), а после в следващата клетка байта на атрибута.

Стартирайте DEBUG и въведете командите:

```
–E B800:0292 41 1E 42 87
–E B800:0 20 71 ‘D’ 74 ’e’ 71 ’b’ 71 ’u’ 71 ’g’ 71 20 71
```

3.3. Основни прекъсвания на ROM-BIOS

Извеждането на символи на екрана в програмата може да се реализира по различен начин:

- с директно обръщение към видеопаметта (както беше описано по-горе);
- използвайки системата на прекъсванията;
- с помощта на функциите и операторите на езика за програмиране от високо ниво.

Обаче в машинните кодове на същите тези функции и оператори се съдържат обръщения към прекъсванията на MS-DOS и BIOS. BIOS (Basic Input/ Output System) се намира в ROM паметта и управлява всички прекъсвания в системата. Основно прекъсване, осигуряващо широк видеосървис (управление на дисплея), представлява прекъсването на BIOS INT 10h (видеосървис INT 10h). Това прекъсване изпълнява множество функции – установяване на режима на екрана, установяване на курсора и номера на видимата страница, извеждане на символи с различни цветове и много друго. Преди извикване на INT 10h от програмата номера на функцията е необходимо за се зареди в регистър AH, а в останалите регистри – данни за извеждане (в строго определен ред).

Ще разгледаме някои функции на видеосървис INT 10h.

Прекъсване	**Функция**	**Въвеждане**
INT 10h	00h –> Установяване на режима	AH = 00h
	01h –> Установяване на размера (формата) на курсора	
	02h –> Установяване на курсора на екрана	AH = 02h BH = номер на страницата (не е задължително) DH = номер на реда на екрана DL = номер на стълба
	03h –> Прочитане позицията на курсора на екрана	
	05h –> Избор на активна видеостраница	
	06h –> Прелистване екрана нагоре (скролиране на екрана)	
	07h –> Прелистване екрана надолу	

	08h –> Четене на символ/ атрибут от видеобуфера	
	09h –> Извеждане на символ с атрибут с повторение	AH = 09h AL = ASCII код BH = номер на видеостраницата BL = атрибут CX = брой на повторенията (1 за един символ)
	0Ah –> Извеждане на символ без атрибут	AH = 0Ah AL = ASCII код BH = номер на видеостраницата CX = брой повторения
	0Eh – писане на символ в режим TTY (телетайп)	AH = 0Eh AL = записвания символ BL = цвят за предния план (при графичните режими)
	13h –> Извеждане на ред от символи с атрибути	AH = 13h AL = номер на подфункцията BH = номер на видеостраницата ES:BP = отместване за данните (данните/реда от символи се чете от адрес ES:BP) CX = броя на извежданите байтове (за AL=0/1) или думи (за AL=2/3) DH = номер на реда на екрана DL = номер на стълба на екрана **Подфункции** 0 –> да се използва атрибута в BL; де не се мести курсора 1 –> да се използва атрибута в BL; курсора да се позиционира в края в реда 2 –> формат на реда: char, attr, ...; де не се мести курсора 3 –> формат на реда: char, attr, ...; да се премести курсора

Функцията за установяване на режима 00h може да се използва също и за бързо очистване на екрана.

Стартираме дебъгера и въвеждаме програма, установяваща режим 1 (40 x 25 знакоместа):

```
–A
xxxx:0100 MOV AH, 0
xxxx:0102 MOV AL, 1
xxxx:0104 INT 10
xxxx:0106 NOP
```

Изпълняваме програмата, въвеждайки командата –G 106.

За да възстановим режим 3 (80 x 25) променяме едната от командите:

```
–A 102
xxxx:0102 MOV AL, 3
xxxx:0104 <Enter>
```

Възстановяваме регистър IP = 100 и отново въвеждаме –G 106.

Нека демонстрираме употребата на функцията за извеждане на символ с атрибут с повторение 09h.

Въведете в дебъгера следната програма:

```
MOV AH, 02
MOV DX, 0101
INT 10
NOP
MOV AH, 09
MOV AL, CE
MOV BL, 1B
MOV CX, 800
INT 10
NOP
```

Стартирайте изпълнението й с –G 113.

Функцията 13h предава 2-байтови кодове от областта с данни на програмата във видеопаметта. Тя има 4 подфункции (режима), които е необходимо да се зададат в AL. Ще използваме режим 3 (извеждане в цвят).

Нека да въведем програмата:

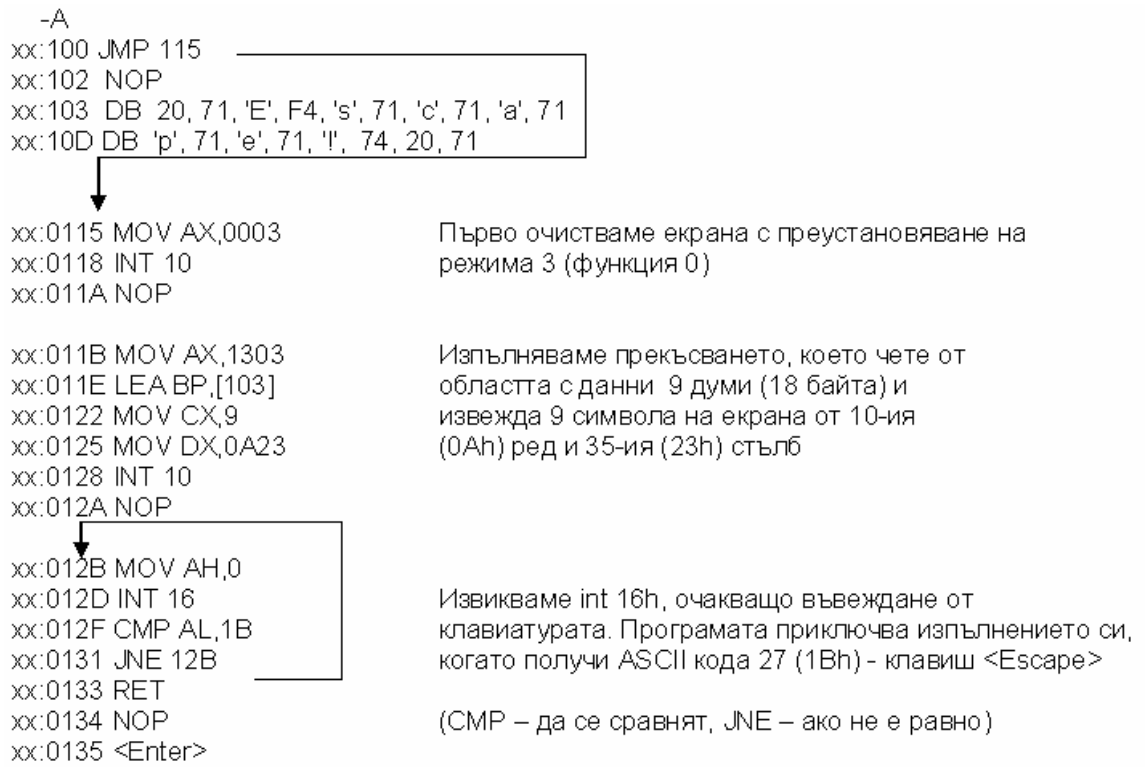
```
JMP 115
NOP
DB 20, 71, ‘E’, F4, ‘s’, 71, ‘c’, 71, ‘a’, 71
DB ’p’, 71, ‘e’, 71, ‘!’, 74, 20, 71
```

```
MOV AX, 0003
INT 10
NOP

MOV AX, 1303
LEA BP, [103]
MOV CX, 9
MOV DX, 0A23
INT 10
NOP

MOV AH, 0
INT 16
CMP AL, 1B
JNE 12B
RET
NOP
```

Данните сме разположили в началото на програмата, (последното е характерно за формат *.COM) като предаваме управлението на първата команда за преход JMP (Jump, скок).



Фигура 10. Изпълнение на програма, използваща функция 13h на прекъсването INT 10

Ако изпълним тази програма до командата RET (–G 133), на очистения екран се появява прочетената от паметта дума “Escape!”. За повторно стартиране на програмата е необходимо възстановяването на регистър IP=100 и отново въвеждане на командата –G 123.

Функциите 09 и 0A не интерпретират никакви управляващи кодове и не преместват курсора при извеждане. Функцията 0E интерпретира следните управляващи кодове:

ASCII	**07h**	**Bell**	**(звънец)**
ASCII 08h	**BS,**	**Back Space**	**(възврат на една стъпка)**
ASCII 0Ah	**LF,**	**Line Feed**	**(преход на нов ред)**
ASCII 0Dh	**CR,**	**Carriage Return**	**(възврат на каретката)**

Функциите 09, 0A и 13h изискват задаване в CX или брой повторения на един и същи символ (09 и 0A), или дължината на извеждания стринг, което често пъти е неудобно при извеждане на редове с различна дължина. Най-удобна за извеждане на прости съобщения от програмата е функцията 0Eh: писане на символ в активната видеостраница (емулация на телетайп). Адресираме извеждания ред с индексния регистър SI и ще запишем символ от реда в AL

```
LEA   SI, string
MOV   AL, [SI]
```

Отмествайки указателя SI в цикъл можем посимволно да изведем на екрана стринга. За целта е необходимо или да се зададе броя на извежданите символи в регистъра CX (за цикъла LOOP), или да се определи някакъв символ в качеството на признак за край на стринга.

Тези функции използват нулев байт в края на реда.

На практика в програмирането операндите могат да се съхраняват в болшинството регистри във всякакви съчетания в командите. Не бива да ни смущава твърдото им функционално предназначение за АЛУ. Наложително е да се помни, че някои команди изискват точно зададени регистри явно или неявно.

3.4. Функции на MS-DOS

Най-често използваното прекъсване за работа с конзолата е системното прекъсване INT 21H. То заема междинно положение между програмите на потребителя и средствата на BIOS. С него едно прекъсване може да изпълни множество функции, като всяка от тези функции се определя чрез придаване на номер на функцията в регистъра AH:

Прекъсване	**Функция**	**Въвеждане**	**Извеждане**
INT 21h	01h –> Изчакване до въвеждане от клавиатурата и показване на екрана (exo)	AH = 01h	AL = въведеният знак
	02h –> Показване върху екрана на съдържимото на DL	AH = 02h DL = символа за извеждане	
	05h –> Извеждане на принтера	AH = 05h	Извежда съдържимото на DL на принтера
	06h –> Четене на буфера на клавиатурата	AH = 06h DL = FFh (255)	ако няма знак в буфера на клавиатурата: ZF=1; ако има знак: ZF= 0 AL = въведеният знак
	08h –> Изчакване на въвеждане от клавиатурата	AH = 08h	AL = въведеният знак
	09h –> Извеждане на екрана на низ от знаци	AH = 09h DS:DX = буфер за въвеждане	
	0Ah –> Въвеждане от клавиатурата на низове до знак за край (CR – клавиш за нов ред)	AH = 0Ah DS:DX = буфер за въвеждане	Първи байт в буфера: максимален брой на байтовете в буфера, вкл. CR (трябва да бъде зададен) Втори байт в буфера: въведения брой

			байтове в дадения момент без CR (по подразбиране се брои възходящо)
	4Ch –> Дефинирано връщане в MS-DOS	AH = 4Ch	

За всички функции, с изключение на 09H важи, че при натискане на Ctrl+C има дефинирано връщане в MS-DOS.

Тема 4. Изграждане програми на асемблер

4.1. Адресация на паметта

Както вече споменахме при адресирането на паметта се използва така наречената *сегментно адресиране*. При него адресът на всяка клетка от паметта се представя чрез две 16-битови числа. За първото се приема, че е изместено на ляво и има 4 двоични нули, залепени след него. Това число се нарича *сегментна част на адреса (базов адрес)*. Второто 16-битово число се нарича *относителна част (отместване)*. Събрани заедно, двете числа образуват пълен адрес, с който може да се адресира всяка клетка от паметта.

Сегментната част определя адреса, кратен на 16 и се нарича *граница на параграф (номер на параграф)*. Относителната част определя точния адрес като относително разстояние от началото на параграфа. Възприет е начин за означаване на сегментираните адреси:

СЕГМЕНТНАТА ЧАСТ : ОТНОСИТЕЛНАТА ЧАСТ

Формиране на действителен (ефективен) адрес:

Сегментна част = 2222
Относителната част = 3333
Сегментиран адрес = 2222:3333
Действителен адрес: 22220
+3333
25553

При този начин на образуване на адреса при един предварително зададен базов адрес с помощта на съответно отместване може да се адресира адресна област от 64 килобайта = 2^{16} (защото отместването е 16-битово число) = 2^{10} * 2^6 = 1024*64 B = 64 KB. Тези области се наричат сегменти на паметта. Съгласно представения по-горе начин за пресмятане на адрес разстоянието между два сегментни начални адреса е 16 байта.

Процесорите могат да се обръщат всеки момент към следните различни сегмента: кодовия (програмния) сегмент, в който се намира кодът на програмата, стековия сегмент (за междинно запомняне на данни и адреси при работа с подпрограми), сегмента за данни и допълнителните сегменти.

Тези сегменти могат да се припокриват. Ако например се знае, че са нужни само 100 байта данни, би било нецелесъобразно да се резервират максимално възможните 64 килобайта за сегмента на данни.

Базовите адреси на тези сегменти, респективно номера на параграфи, се запомнят в съответните сегментни регистри на процесора: кодов сегмент (CS), сегмент за данни (DS), стеков сегмент (SS) и допълнителните сегменти за данни.

Като отместване при кодовия сегмент се приема съдържанието на указателя на инструкции, а при сегмента за данни според вида на адресирането това е съдържанието на регистрите BP, BX, SI, DI, т. е. отместването се взема както при най-често използваното директно адресиране от кода на инструкцията. При стековия сегмент се използва стековия указател (SP, ESP) или базовия указател (BP, EBP). Тъй като допълнителния сегмент е една алтернатива на сегмента за данни, отместването се взема от посочените при този сегмент регистри.

При програмирането се предвиждат същите сегменти както представените по-горе в така наречения сегментен модел на процесора. Така всяка програма може да се състои от няколко сегмента – например кодов сегмент, сегмент за данни, допълнителен сегмент и стеков сегмент, но трябва непременно да има поне един кодов (програмен) сегмент.

В сегмента за код се поместват инструкциите, които трябва да се изпълнят при зареждане на програмата. Първата инструкция, с която ще се запознаем, е инструкцията MOV. Тя принадлежи към групата от инструкции за прехвърляне на данни.

MOV <операнд_1>, <операнд_2>

операнд_1 (цел, приемник)– destination (DST)

операнд_2 (източник)– source (SCR)

Копира стойността на операнда-източник в операнда-приемник.

Прехвърля байт или дума (или двойна дума за i386+) между регистри, между регистри и памет, а също така прехвърля директно стойност в регистър или в паметта.

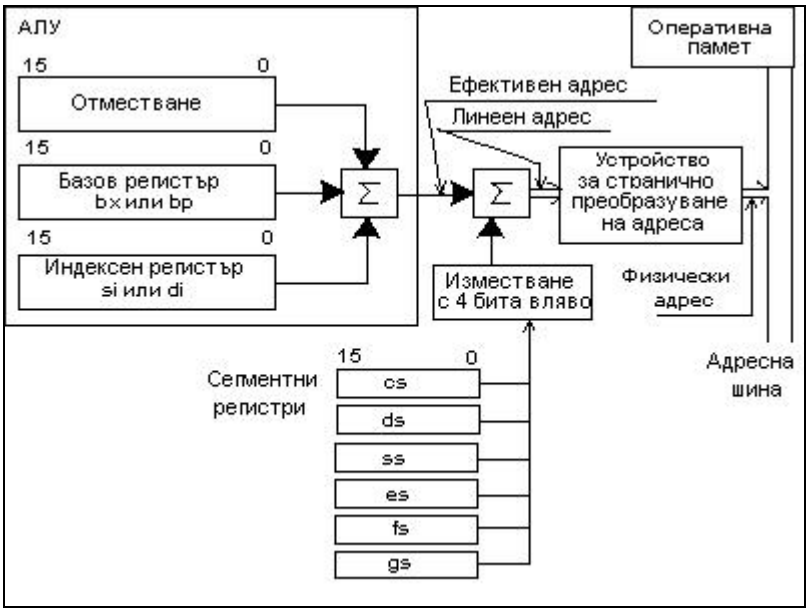
Операндите следва да са от един и същи тип.

Флаговите регистри не се променят.

Препоръчително е използването в качеството на един от операндите регистър al/ax/eax.

В случаите, когато е необходимо уточняване типа на използваните операнди се използва специален оператор за съгласуване размера на операндите PTR. Byte ptr, Word ptr, Dword ptr (указатели към байт, дума и двойна дума).

Пример: mov ax, word ptr [bx] в случай, че [bx] адресира дума в паметта.



Фигура 11. Механизъм на формиране на физическия адрес в реален режим

4.2. Изписване на числа и букви

Съобразно стандарта асемблер разглежда зададените числа като десетични числа и ги преобразува в съответните двоични числа. Ако искаме да използваме шестнайсетично, съответно осмично или двоично число, трябва след последния разред да следва съответно H, O или B. Ако едно шестнайсетично число започва съответно с буква, например F6H, отпред трябва да се прибави една нула – 0F6H, тъй като иначе асемблер ще разглежда числото като име на променлива.

Стандартното установяване на бройната система става с директивата

. RADIX база,

Където базата трябва да е число между 2 и 16, за да се установи съответната бройна система: например с .RADIX 16 се установява шестнайсетична бройна система. В такъв случай едно десетично число трябва да бъде означено чрез прибавяне на буквата D.

По отношение на изписването на буквите: всички инструкции и директиви може да се записват с главни и/или малки букви.

4.3. Дефиниране на променливи и области от паметта

За резервиране на даден обем памет и за дефиниране на променливи се използва директивата Define:

DB (Define Byte)	**– резервира един байт**
DW (Define Word)	**– резервира една дума**
DD (Define Double Word)	**– резервира двойна дума**
DF (Define Far–Word)	**– резервира 6 байта**
DQ (Define Quard–Word)	**– резервира 4 думи**
DT (Define Ten–Bytes)	**– резервира 10 байта**

Пълният формат на директивата Define (тук например DB) е:

***име_на_променлива* DB *израз*, *израз*, *израз*. . .**

Ако не е зададено име, обемът памет се резервира без отнасяне към дадена променлива.

Като *израз* може да се използват:

- константа, респ. аритметично свързани константи – тогава дефинираният обем памет се инициализира с тези константи;
- въпросителен знак – тогава обемът памет се резервира, но не се инициализира;
- адрес-израз (само при DW и DD);
- ASCII низ (низ от знаци) – тогава низът от знаци се записва в резервираният обем памет. Ако не се използва DB низът от знаци може да бъде дълъг само два знака.

С размножителя DUP (Duplicate) може да се използват следните формати:

- *брой* DUP (?)

Резервира *брой* клетки в паметта, без те да се инициализират;

- *брой* DUP (*израз, израз*. . .)

Резервира *брой* клетки в паметта и ги инициализира с дадените вдясно изрази.

За да се дефинира масив трябва с директива за дефиниране на данни да се зададе име на масива и последователност от инициализиращи стойности

и/или повторяеми изрази. Списъкът от начални стойности може да заеме няколко последователни реда, като редовете с продължение трябва да завършват със запетая.

Когато името на променлива, определена с директива за дефиниране на данни, се използва в инструкция, то се отнася за отместването (OFFSET) на първия байт от заделената за променливата памет. Нека например са дефинирани данните:

```
lst     Byte    10, 20, 30, 40, 50
```

Когато името lst се посочи в инструкция като директен операнд-памет, то ще се асемблира с отместването на първия от петте байта, заделени за lst. Ако отместването на първия байт е 120h в сегмента за данни, тогава инструкцията

```
mov     ah, lst
```

е еквивалентна на инструкцията

```
mov     ah, ds:120h
```

и при изпълнение ще доведе до копиране на стойността на първия байт от масива (10) в регистър AH. Ако трябва да се направи обръщение към някой от следващите байтове, като адрес може да се посочи израз с използване на оператора +, т. е. към името lst да се прибави цяла константа, задаваща изместването на байта спрямо началния адрес на масива, например за копиране на стойността на втория байт (20) в регистър AL може да запише:

```
mov     al, lst+1
```

Така обръщението към първия байт на lst става с посочване на името lst, към втория – с lst+1, към третия – с lst+2 и т. н.

Когато данните са дефинирани с директива, различна от BYTE, тогава изместването на всеки следващ елемент няма да е 1, а размера на паметта, която се заделя от асемблера за съответния тип (2 за Word, 4 за Double Word и т.н.).

Друг възможен начин за адресиране на елементи на масив е с посочване на индекс в квадратни скоби след името на масива, например lst[1], wlst[4]. Има обаче съществена разлика между индексите при езиците от високо ниво и индексите в асемблерния език. При езиците от високо ниво стойността на индекса винаги отговаря на позицията на елемента в масива, например в

езика C arr[9] се отнася за десетия елемент на масива arr, независимо дали елементите са с дължина 1 или 8 байта. В асемблерния език стойността на индекса е равна на броя на байтовете между елемента и началото на масива. Тази разлика може да се пренебрегне, само ако данните са от тип BYTE – lst[1] се отнася за втория елемент на масива lst, lst[2] – за третия и т. н.

Изместването за достъп до елементите може да се изчисли по формулата:

n–ти ***елемент_на_масива=име_на_масива*** **[(n–1)*** ***размер_на_елементите*****]**

Двата начина за адресиране на елементи на масив са еквивалентни.

Var DB 0	Резервира 1 байт за променливата Var и го инициализира с 0
Var	Резервира 1 байт за променливата Var и оставя непроменено съдържанието на клетките в паметта
Array DW 100, 200, 300, 400	Резервира 4 думи за променливата Array и ги инициализира с 100, 200, 300, 400
Str DB “Hello, World!”	Резервира 13 байта за променливата Str и ги запълва с низа от знаци “Hello, World!”
Array DB 20 DUP(?)	Резервира 20 байта за променливата Array и оставя непроменено съдържанието на клетките в паметта
Array DB 20 DUP(0)	Резервира 20 байта за променливата Array и ги инициализира с 0

Таблица 6. Примерни дефиниции на променливи чрез DEFINE

4.4. Дефиниране на сегменти

4.4.1. Пълна форма на дефиниране на сегменти

Всеки сегмент на програмата започва с директивата SEGMENT и завършва с директивата ENDS (от End of Segment). На всеки сегмент независимо от типа му се присвоява едно дадено от програмиста име. В края на програмата трябва да се появи директивата END. Тя показва на асемблера, че програмата е завършила, а по-нататъшните байтове в паметта не принадлежат към програмата, т. е. асемблер може да завърши на това място превеждането на инструкциите. След думата END може да се въведе стартов адрес. Когато

готовата изпълнима програма бъде заредена, на това място тя се стартира автоматично. Следователно асемблерската програма има следната структура:

```
име_на_сегмент    SEGMENT
                  …
                  …
име_на_сегмент    ENDS
                  END
```

Пълният формат на директивата SEGMENT изглежда така:

```
                  PARA
                  BYTE      PUBLIC
име_на_сегмент    SEGMENT   WORD      COMMON    USE 16
                  DWORD     STACK     USE 32
                  PAGE      AT
```

Име на сегмента може да бъде всяко валидно име в асемблер. Имената се използват като етикети на инструкции или символични имена на променливи, константи, процедури, сегменти и на дефинирани от потребителя типове – структури, обединения, записи и типове.

Всяко име се състои от символи. То представлява комбинация от букви, цифри и символа за подчертаване (_). То трябва да започва с буква или с един от следните символи: за подчертаване (_), за долар ($), символа at (@) или точка (.). Асемблер не прави разлика между главни и малки букви. Дължината на всяко име е произволна, но асемблер отчита само първите 247 символа. Като име не може да се използва някоя от запазените думи или символични имена.

Използването на символа at (@) като първи символ на потребителско име трябва да се избягва, защото MASM 6.1 предефинира някои имена, започващи с него.

Запазените думи, които следват думата SEGMENT не са задължителни:

- Изравняване

Указанията PARA до PAGE са за начина на подреждане. Ако тук не е дадено нищо, прилага се стандартното подразбиращо се указание PARA, което означава, че сегментът започва с един адрес на параграф, т.е. с един адрес делим на 16 без остатък (в най-лошия случай при това между два сегмента в паметта остават 15 неизползвани байта). При BYTE сегментът започва при

следващия свободен байт в паметта. При указание WORD сегментът започва при следващата свободна клета на паметта с четен адрес, при DWORD – при следващия адрес, делим на 4 без остатък, при PAGE – при следващото цяло число, кратно на 256.

- Комбиниране

Указанието за свързване съобщава на свързващата програма как да бъдат свързани помежду си сегментите с еднакви имена. Ако няма предписание, всеки от представените сегменти се зарежда в собствен физически сегмент с отделен базов адрес. Този стандартен тип се означава също и като PRIVATE. При указание PUBLIC сегментите с еднакви имена се свързват към един единствен сегмент. Отместването на всички адреси на инструкции и данни започва при първия и завършва при последния сегмент, като всички сегменти имат един общ базов адрес, който се намира в текущия сегментен регистър. Типът STACK се държи като типът PUBLIC, само че всички адреси са относителни спрямо базовите адреси в регистъра на стековия сегмент. При указание COMMON всички сегменти с еднакви имена се зареждат, припокривайки се един друг, което означава че всички започват от един и същи адрес. При тип AT всички указания за преход и адреси на променливи се пресмятат спрямо зададения след AT адрес.

- Размер

Указанията USE 16, респ. USE 32, са възможни едва от 80386 и то само ако предварително е била използвана директивата .386. С тях се установява максималния размер на сегмента. С USE 16 се установява 16-битов размер на сегмента, т. е. максимално 64 килобайта. USE 32 дефинира 32-битов размер на сегмента с максимална дължина 4 гигабайта.

Следващият пример извежда съобщението ‘Hello World!’. В него се използват стандартните директиви за сегментация. За извеждането на текст MS-DOS е предвидила специалната функция 09H на системното прекъсване INT 21H. Тя очаква в DS:DX началния адрес на намиращия се в паметта текст. Желаният текст трябва да бъде записан предварително в паметта с директивата Define. Знакът за долар (‘$’) в края на текста се очаква от функцията 09H като знак за край на текста, но не се извежда на екрана.

Вместо да се означи текстът със съответните символи (тук апострофи), може да се въвежда също ASCII стойността на отделните знаци, но тогава отделните байтове трябва да се разделят със запетаи.

За данните в програмата ще дефинираме отделен сегмент за данни:

```
data        segment
            message     db          'Hello World! $'
data        ends
```

За съжаление обаче при зареждане на програмата сегментния регистър за данни не се инициализира с коректна стойност. Поради това инициализирането трябва да бъде извършено от самия потребител.

Отместването (Offset) на адреса на първия байт от текстовата променлива message, което MS-DOS очаква в DX, получаваме с помощта на оператора OFFSET:

```
mov dx, offset message
```

Следователно не е необходимо при обръщение към данни да се интересуваме от абсолютните адреси на местата в паметта, в които те са съхранени. Въз основа на дефиницията в сегмента за данни асемблер знае точния адрес на отделните променливи и го поставя в инструкцията на мястото на използваните от нас променливи.

Много от асемблерните инструкции използват сегмент по подразбиране. Когато асемблер изчислява адрес, той трябва да знае кой сегментен адрес в кой сегментен регистър се съдържа. Информация за това асемблер получава от директивата ASSUME:

ASSUME ***сегм_регистър: сегм_*име *[, сегм_регистър: сегм_име] . . .***
ASSUME ***регистър_за_данни: тип [, регистър_за_данни: тип] . . .***
ASSUME ***регистър: ERROR [, регистър: ERROR] . . .***
ASSUME ***[регистър:] NOTHING[, регистър:] NOTHING . . .***
ASSUME ***регистър: FLAT [, регистър: FLAT] . . .***

Сегментният регистър може да бъде кой да е от регистрите CS, DS, SS и ES (или FS и GS за i386+). Сегментното име може да е:

- име на сегмент, дефиниран с директивата сегмент;
- име на група от сегменти, дефинирана с директивата GROUP;
- ключовата дума NOTHING, ERROR и FLAT;
- SEG израз.

С директивата ASSUME може да се зададе сегмент за всеки от сегментните регистри (обикновено без регистър CS, защото асемблер автоматично присвоява на CS адреса на текущия сегмент за инструкциите). Инструкциите след ASSUME, които използват сегментен регистър по подразбиране,

предполагат, че етикетът или променливата са в сегмента със съответното сегментно име.

Най-често всички сегментни регистри се дефинират с една единствена директива ASSUME в началото на сорс-кода, но тя може да се използва и на произволно място в програмата за смяна на сегментните адреси по подразбиране. Ключовата дума NOTHING отменя стойността по подразбиране за съответния регистър, а ASSUME NOTHING отменя стойностите по подразбиране за всички регистри от предишни директиви ASSUME. Смяната на стойността по подразбиране с ASSUME е еквивалентна на използването на оператора за смяна на сегментния регистър по подразбиране (:), който е по-подходящ за еднократни действия, докато ASSUME е по-удобна за поредица от инструкции.

За да не се допускат грешки от поява на инструкции и етикети на инструкции в сегмент за данни, употребата на CS може да се предотврати с:

ASSUME CS:ERROR

Трябва да се отбележи, че директивата ASSUME оказва влияние само върху процеса на асемблиране, но не и по време на изпълнение на програмата. Затова в кода на програмата трябва да има инструкции за зареждане на сегментните регистри със сегментни адреси.

За пример ще напишем програма, извеждаща съобщението “Hello, World!” на екрана, демонстрираща употребата на стандартните директиви за сегментация.

```
data    segment para public 'data'
        message db 'Hello World!$'
data    ends
stk     segment stack
        db 256 dup ('?')                ;сегмент на стека
stk     ends
code    segment para public 'code'      ;начало на кодовия сегмент
        assume cs:code, ds:data, ss:stk
main:   mov ax, data                    ;адреса на сегмента данни в регистър ax
        mov ds, ax                      ;ax в ds
        mov ah, 9
        mov dx, offset message
        int 21h                         ;извеждане съобщението на екрана
        mov ax, 4c00h                   ;4c00h в ax
        int 21h                         ;стандартен изход от програмата
code    ends                            ;край на кодовия сегмент
end     main                            ;край на програмата с входна точка main
```

Подредбата на сегментите в записа на кода е произволна. Следващият пример го демонстрира.

```
Code    segment
        assume cs:Code, ds:Data
begin:
        mov ax, Data            ;инициализация на сегмента с данни
        mov ds, ax

        mov ah, 09h             ;установяване номера на функцията
        mov dx, offset var      ;установяване стойността на аргумента
        int 21h                 ;обръщение към MS DOS (функция 09h)

        mov ah, 4Ch             ;изход в MS DOS
        int 21h
Code    ends

Data    segment
        var db 'Hello, World!!!', 13, 10, '$'       ;текст за извеждане
Data ends

Stk     segment stack
        dw 100 dup(?)           ;памет за стека
Stk     ends
        end begin
```

Следващият пример съдържа единствено кодов сегмент. Програмата чете един знак от клавиатурата и да го извежда върху екрана.

```
code_seg    segment
            assume cs: code_seg
m1:         mov ah, 01h
            int 21h
            jmp m1
code_seg    ends
            end
```

4.4.2. Кратка форма (Опростени директиви за сегментация)

За по-прости програми, съдържащи по един сегмент за код, данни и стек в транслаторите на MASM и TASM е въведена възможност за използване на опростени директиви за сегментация. Но тук е възникнал проблема, свързан с необходимостта да се компенсира невъзможността директно да се управлява разместването и комбинирането на сегментите. Затова съвместно с опростените директиви за сегментация се използва и директивата за посочване модела на паметта MODEL, която частично управлява разместването на сегментите и изпълнява функциите на директивата

ASSUME (затова при използване на опростените директиви за сегментация директивата ASSUME може да се пропусне). Директивата ASSUME свързва сегментите, които в случая на използване на опростени директиви за сегментация имат предопределени имена, със сегментните регистри (макар че явно да се инициализира DS все пак се налага).

Задължителен параметър на директивата MODEL се явява модела на паметта. Този параметър определя модела на сегментация на паметта за програмния модул. Предполага се, че програмният модул може да има само определени типове сегменти, които се определят от опростените директиви за описание на сегменти. Тези директиви са описани в таблицата.

Формат на директивата (режим MASM)	Предназначение
.CODE [name]	Начало или продължение на кодовия сегмент
.DATA	Начало или продължение на сегмента за инициализирани данни. Също така се използва за определяне на данни от типа near
.CONST	Начало или продължение на сегмента за постоянни данни (константи)
.DATA?	Начало или продължение на сегмента с неинициализирани данни. Също така се използва за определяне на данни от тип near
.STACK [size]	Начало или продължение на сегмента за стека. Параметърът [size] задава размера на стека
.FARDATA [name]	Начало или продължение на сегмента за инициализирани данни от тип far
.FARDATA? [name]	Начало или продължение на сегмента за неинициализирани данни от тип far

Таблица 7. Опростени директиви за определяне на сегмента

Наличието в някои директиви на параметър [name] подсказва за възможността за определяне на няколко сегмента от този тип. От друга страна, наличието на няколко вида сегменти за данни е обусловено от изискването да се осигури съвместимост с някои компилатори на езиците от високо ниво, които създават различни сегменти с данни за инициализирани и за неинициализирани данни, а също и константи.

При използване на директивата MODEL транслаторът прави достъпни няколко идентификатора, към които е възможно обръщение по време на работа на програмата, за да може да се получи информация за едни или други характеристики на дадения модел памет (виж табл. 9). Ще изброим тези идентификатори и тяхното предназначение (табл. 8).

Име на идентификатора	Предназначение
@code	Физически адрес на кодовия сегмент
@data	Физически адрес на сегмента за данни от тип near
@fardata	Физически адрес на сегмента за данни от тип far
@fardata?	Физически адрес на сегмента за неинициализирани данни тип far
@curseg	Физически адрес на сегмента за неинициализирани данни тип far
@stack	Физически адрес на сегмента за стека

Таблица 8. Идентификатори, създавани от директивата MODEL

Операндите на директивата MODEL се използват за задаване модела на паметта, който определя набора от сегменти на програмата, размерите на сегментите за данни и код, начина на свързване на сегментите и сегментните регистри. В табл. 9 са дадени някои стойности на параметъра модел на паметта на директивата MODEL.

Модел	Тип на кода	Тип на данните	Предназначение на модела
TINY	near	near	Кодът и данните са обединени в една група с име DGROUP. Използват се за създаване на програми с формат .com.
SMALL	near	near	Кодът заема един сегмент, данните са обединени в една група с име DGROUP. Този модел се използва за болшинството програми на асемблер
MEDIUM	far	near	Кодът заема няколко сегмента, по един за всеки обединяем програмен модул. Всички

			указатели за предаване на управлението са тип far. Данните са обединени в една група; всички указатели към тях са тип near
COMPACT	near	far	Кодът е в един сегмент; указателите към данните са от тип far
LARGE	far	far	Кодът е в няколко сегмента, по един на всеки обединен програмен модул

Таблица 9. Модели на паметта

NEAR и FAR са атрибути за отдалеченост. По подразбиране атрибутът за отдалеченост е NEAR. При него адресирането в паметта става само с 16-разредно отместване, без сегментен адрес. При атрибута за отдалеченост FAR адресирането става със сегментен адрес и отместване.

За пример отново ще напишем програма, извеждаща съобщението “Hello, World!” на екрана, но този път ще използваме съкратените директиви за сегментация:

```
masm                    ;режим на работа за TASM: masm, за MASM няма нужда
model small             ;модел на паметта
.data                   ;сегмент за данни
        message db 'Hello World!$'
.stack  256             ;сегмент на стека
.code                   ;сегмент за кода
main:   mov ax, @data   ;зареждаме адреса на сегмента с данни в регистър ax
        mov ds, ax      ;ax в ds
        mov ah, 9
        mov dx, offset message
        int 21h         ;извеждаме съобщението на екрана
        mov ax, 4c00h   ;зареждаме 4c00h в регистър ax
        int 21h         ;извикване прекъсване с номер 21h
end     main            ;край на програмата с входна точка main
```

4.4.3. Изводи

- *Стандартните директиви* се използват, когато програмистът желае да получи пълен контрол над разместването на сегментите в паметта и комбинирането им със сегментите на други модули.

- *Опростените директиви* е целесъобразно да се използват за прости програми и за програми, предназначени за свързване с програмни модули, написани на езици от високо ниво. Това позволява на линкера ефективно да свързва модулите от различни езици за сметка на стандартизацията на връзките и управлението.
- Функционалното предназначение на сегментацията е много по-широко отколкото просто разбиване на програмата на блокове код, данни и стек. *Сегментацията* е част от по-общ механизъм, свързан с концепцията за модулно програмиране. Тя предполага унификация на формата на обектните модули, създавани от компилатора, в това число и компилируеми от различни езици за програмиране. Последното позволява обединяване на програми, написани на различни езици. Именно за реализацията на различните варианти на такова обединение са предназначени вариантите в директивата SEGMENT.

4.5. Асемблиране и свързване на програмите

За транслация на текста на програмата на асемблер в машинен код ще ползваме компилатора TASM (Turbo Assembler на фирмата Borland International). Въвеждаме командата:

D:\TASM\TASM *име*. ASM

Ако програмата съдържа грешки, на екрана ще се появи съобщение **Error** име.asm (№ на реда) и диагностика на грешките.

Пример:

```
Assembling file: outword. asm
**Error** outword.asm (10) Undefined symbol: F4H
**Error** outword.asm (16) Too few operands to instruction
Error messages:2
Warning messages:None
Passes:1
Remaining memory:460k
```

В дадения пример това означава: В ред 10 е неправилно въведено шестнайсетично число – то трябва да започва с цифра – 0F4H вместо F4H.

В ред 16 “прекалено малко операнди има в командата” – пропусната е запетаята в командата MOV DH, 0AH

Ако в края на командата има опция /z, тогава заедно със съобщението на екрана ще се извежда и сгрешения ред.

При успешно приключване на асемблирането (компилацията) в текущата директория ще се създаде обектен модул *име*.OBJ – файл, съдържащ машинния код, но в който не са установени относителните адреси. Няколко такива модули могат да бъдат впоследствие обединени в единен изпълним модул във файла *име*.EXE или *име*.COM.

Това се изпълнява от специална програма линкер или свързващ редактор (Link Editor). Въвеждаме командата:

D:\TASM\TLINK *име*.OBJ

При успешно изпълнение на свързването се образуват файл *име*.EXE и файл *име*.MAP с карта на разместване на сегментите (в случай, че са няколко).

Start	Stop	Length	Name Class
00000H	0003FH	00040H	SSEG STACK
00040H	00051H	00012H	DSEG DATA
00060H	00096H	00037H	CSEG CODE

Program entry point at 0006:0000

Ако програмата на асемблер е била написана във формат COM, тогава EXE файлът няма да се изпълнява правилно. Той трябва да се преобразува в COM файл с командата EXE2BIN:

D:\TASM\EXE2BIN *име*.EXE *име*.COM

Има още една възможност – да се изпълни свързването с опция /t. При което веднага се създава изпълним COM файл.

D:\TASM\TLINK *име*.OBJ /t

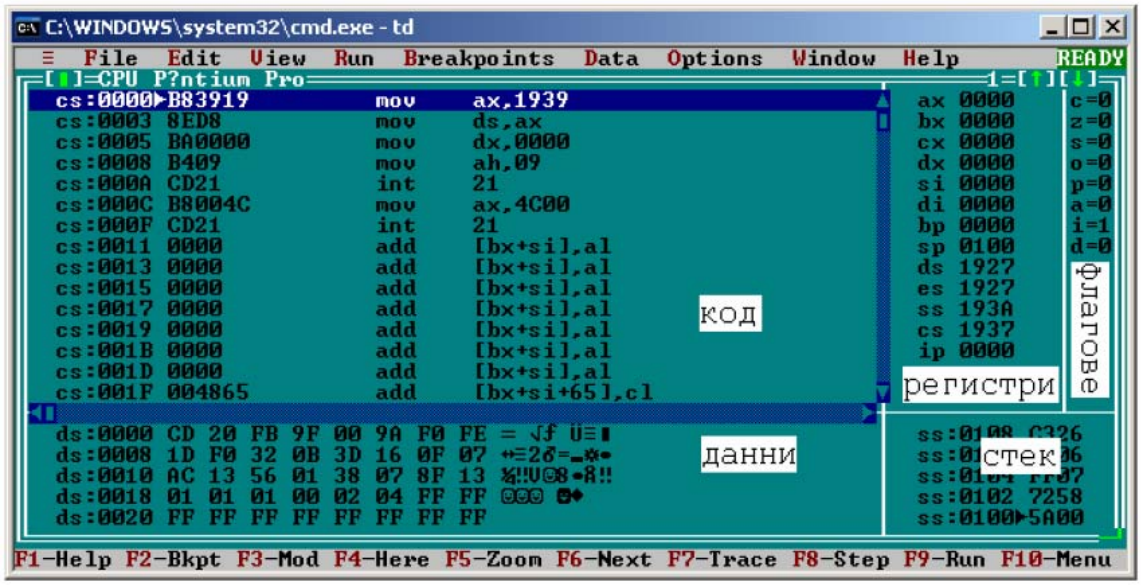
При свързването на програмата може да бъде изведено предупреждение за отсъстващ стек. “Warning: Nostack”. Това съобщение следва да се игнорира. (Отделен сегмент за стек в COM програмите няма).

Разширенията .ASM (за компилатора) и .OBJ (за линкера) могат да не се задават – те се подразбират.

4.6. Turbo Debugger

Програмата Turbo Debugger (TD) позволява да се определи мястото на логическата грешка и нейната причина. Тя притежава следните възможности:

трасиране на програмата в права и обратна посока, преглед и изменение на ресурсите на процесора по време на трасирането. По време на трасирането стойностите на всички прозорци се изменят динамично спрямо текущото изпълнение на инструкциите. Променените стойности се оцветяват в бяло. Във всеки един момент активен може да бъде само един прозорец от тези, които са по подразбиране: за код, данни, регистри на процесора, стек и флагове.



Фигура 12. Прозорец на Turbo Debugger (TD)

Тема 5. Команди за прехвърляне на данни

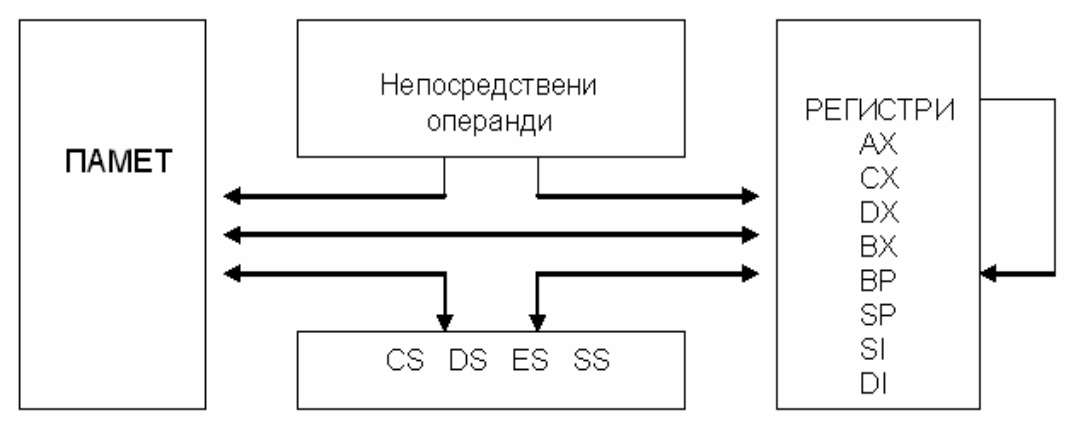
Тези команди могат да се групират според функционалното си предназначение по следния начин:

- прехвърляне на данни с общо предназначение
- въвеждане-извеждане от порт
- работа с адреси и указатели
- преобразуване на данни
- работа със стека

5.1. Команди за прехвърляне на данни с общо предназначение

Към тази група принадлежат командите MOV и XCHG.

Командата MOV прехвърля байт, дума или двойна дума между регистри, между регистри и памет, а също така прехвърля директно стойност в регистър или в паметта. Позволените начини за прехвърляне на данни са показани на схемата.



Фигура 13. Прехвърляне на данни

MOV – е основна команда за прехвърляне на данни. Тя реализира различни варианти за прехвърляне и притежава следните характерни особености:

- не може да се прехвърля от една област на паметта в друга област на паметта;
- не може да се зарежда в сегментен регистър стойност непосредствено от паметта;
- не може да се прехвърля съдържимото на един сегментен регистър в друг сегментен регистър;
- не може да се използва в качеството на операнд-цел сегментният регистър CS.

За двупосочно прехвърляне на данни се използва командата xchg. Операндите следва да са от един и същи тип и не се допуска обмяната на съдържимото на две клетки от паметта.

XCHG <операнд_1>, <операнд_2>

Разменя стойностите на операнда-източник и операнда-приемник.

5.2. Команди за въвеждане-извеждане от порт

Освен адресното пространство на оперативната памет микропроцесорът поддържа адресно пространство за въвеждане-извеждане, което се използва за достъп до устройствата за вход-изход. Размерът на това адресно пространство е 64 Кбайта. За всяко входно-изходно устройство компютърът в това пространство заделя адреси. Конкретните стойности на адреса са в пределите на това пространство и се наричат входно-изходни портове. Физически на порта за въвеждане-извеждане съответства апаратен регистър, достъпът до който се реализира с командите in и out. Регистрите, адресирани с помощта на входно-изходните портове, могат да имат разрядност 8, 16 или 32 бита, като за всеки конкретен порт разрядността на регистъра е фиксирана. В качеството на източници на информация или цели се използват така наричаните регистри-акумулатори EAX, AX, AL. Изборът на последния се определя според разрядността на порта. Номерът на порта може да се задава непосредствено с операнд в командите in и out или със стойност в регистър DX. Сведения за номерата на портовете, за разрядността им, за формата на управляващата информация са приведени в техническото описание на устройствата.

IN акумулатор, номер_на_порт: прехвърля байт или дума (или двойна дума за i386+) от порт към акумулатора. Адресът на порта се посочва от операнда-източник, който може да бъде DX или 8 b константа. Константи могат да се използват само за номера на портове, по-малки от 255, като за по-големи номера се използва съдържанието на DX.

OUT номер_на_порт, акумулатор: прехвърля байт или дума (или двойна дума за i386+) към порт от акумулатора. Адресът на порта се посочва от операнда-приемник, който може да бъде DX или 8 b константа. Константи могат да се използват само за номера на портове, по-малки от 255, като за по-големи номера се използва съдържанието на DX.

Примерно инструкцията

```
in al, 41h
```

копира байт от входно-изходния порт 41h в регистър AL.

При втория начин номера на входно-изходния порт (константа по-малка от 255) може да бъде прочетен от регистър DX:

```
mov dx, 41h
inal, dx
```

В следващият пример се установява в порт 3B4h стойността 0Fh:

```
mov dx, 3b4h
mov al, 0fh
out dx, al
```

5.3. Команди за работа с адреси и указатели в паметта

LEA DST, SRC: изчислява ефективния адрес (отместването) на операнда-източник (памет) и записва резултата в операнда-приемник. Ако операндът-източник е директен адрес в паметта асемблер кодира инструкцията в по-ефективната форма MOV reg, OFFSET mem

LDS DST, SRC: определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в DS, а отместването на указателя – в операнда-приемник.

LES DST, SRC: определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в ES, а отместването на указателя – в операнда-приемник.

LGS DST, SRC (само за i386+): определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в GS, а отместването на указателя – в операнда-приемник.

LFS DST, SRC (само за i386+): определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в FS, а отместването на указателя – в операнда-приемник.

LSS DST, SRC (само за i386+): определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в SS, а отместването на указателя – в операнда-приемник.

Командата LEA прехвърля не данни, а ефективния адрес на данните (отместването на данните относно началото на сегмента данни) в регистър, зададен като цел. Дадената команда представлява алтернатива на оператора offset. Често пъти не е достатъчно да се знае само ефективния адрес, а е необходим пълния адрес за указател към данните. Той има две съставящи: сегмента и отместването. Останалите команди от тази група позволяват получаването на пълния указател в паметта от двойката регистри. Предварително е необходимо да се получат стойностите на пълния указател в някоя област на паметта и после да се посочи в командата получаването на пълния адрес и името на тази област. Използва се директивата за резервиране

и инициализация на паметта DEFINE. Какъв адрес ще се формира (ефективен или пълен), зависи от използваната директива. Ако е dw, тогава в паметта се формира само 16-битова стойност на ефективния адрес, при dd – в паметта се записва пълния адрес. Разместването на този адреса в паметта е следното: в младшата дума се намира отместването, в старшата – 16-битовата сегментна съставляваща на адреса.

Следващата програма извежда ред на екрана с използване на функцията 09h INT 21h.

```
model small
.stack  256
.data
        string db “Ред за извеждане. $”
        adr_string dd string
.code
main:
        mov ax, @data
        mov ds, ax

        lds dx, adr_string      ; адреса на реда за извеждане на екрана в DS:DX
        mov ah, 09h
        int 21h

        mov ax, 4c00h
        int 21h
end     main
```

5.4. Команди за преобразуване на данни

XLAT [адрес_на_прекодиращата_таблица] : транслира стойност от една кодова система в друга чрез търсене на стойността за транслиране в таблица, съхранена в паметта. Преди изпълнението на инструкцията BX трябва да сочи към таблицата в паметта, а AL трябва да съдържа позицията на транслираната стойност в таблицата (цяло число без знак). След изпълнение на инструкцията AL съдържа стойността от таблицата от посочената позиция. Операнд не е необходим, но може да се посочи за отмяна на сегмента по подразбиране (DS). XLATB е синоним на XLAT.

Действието на командата се заключава в зареждането в регистър AL стойността на друг байт от таблица в паметта, разположена на адрес, сочен от адреса на прекодиращата таблицата. “Таблица” е условно наименование – това е низ от знаци с размер байт. Адресът на байта в низа, по който ще става заместването на съдържимото на регистър AL, се определя от сумата BX + AL, т. е. съдържимото на AL изпълнява ролята на индекс в байтовия масив.

В следващия пример прекодираме въведено от клавиатурата шестнайсетично двуцифрено число в двоичен код като резултатът се записва в AL.

```
;Програма за преобразуване на двузначно шестнайсетично число
;(в символен вид) в двоично представяне.
;Вход: шестнадесетично число от две цифри, въвежда се от клавиатурата.
;Изход: резултатът от преобразуването в регистър al.

model small
.stack 256
.data
        mes db 'Въведете две шестнайсетични цифри. $'
        tabl db 48 dup (0), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 7 dup (0), 0ah, 0bh, 0ch, 0dh, 0eh,
        0fh, 26 dup (0), 0ah, 0bh, 0ch, 0dh, 0eh, 0fh, 153 dup (0)

.code
main:
        mov ax, @data           ;адреса на сегмента с данните в регистър ax
        mov ds, ax              ;ax в ds
        lea bx, tabl            ;зареждане адреса на реда с байтове в регистър bx
        mov ah, 9
        mov dx, offset mes
        int 21h
        xor ax, ax              ;занулуване на регистър ax
        mov ah, 1h              ;1h в регистър ah
        int 21h                 ;генерация на прекъсване с номер 21h
        xlat                    ;прекодиране на първия въведен символ в al
        xor dl, dl              ;занулуване на регистър dl
        mov dl, al              ;съдържимото на регистър al в регистър dl
        shl dl, 4               ;изместване съдържимото на dl с 4 разряда вляво
        int 21h                 ;извикване на прекъсване с номер 21h
        xlat                    ;прекодиране на втория въведен символ в al
        add al, dl              ;събиране: (dl)=(dl)+(al)

        mov ax, 4c00h           ;зареждане на 4c00h в регистър ax
        int 21h                 ;извикване на прекъсване с номер 21h

end     main                    ;край на програмата с входна точка main
```

5.5. Команди за работа със стека

За работа със стека са предназначени три регистъра:

- SS – регистър на стековия сегмент;
- SP/ESP – регистър на указателя на стека;
- BP/EBP – регистър на указателя базата на стека.

Особености на работата със стека:

- запис и четене на данни в стека в съответствие с принципа LIFO (Last In First Out);
- при запис на данни в стека, последният расте в посока на младшите адреси;
- регистрите ESP/SP и EBP/BP съдържат отместването относно SS.

Ако е необходим достъп до елемент вътре в стека се използва регистър EBP/BP.

За работа със стека съществуват специални команди за запис и четене:

PUSH SRC: Записва операнда-източник във върха на стека. Инструкциите PUSHW и PUSHD се използват съответно за запис в стека на дума и двойна дума.

Алгоритъмът на работа на командата е:

- SP = SP – 2; стойността на SP се намалява с 2;
- стойността на източника се записва на адрес, задаван с двойката SS:SP.

POP DST: Записва съдържимото от върха на стека на мястото, посочено от операнда-цел. Стойността се извлича от върха на стека.

Алгоритъмът на работа на командата *pop* е обратен на *push*:

- запис на съдържимото от върха на стека на мястото, посочено от операнда-цел;
- SP = SP + 2; увеличаване стойността на SP с 2.

PUSHA (само за i186+): записва в стека стойностите на осемте регистъра с общо предназначение в следния ред: AX. CX, DX, BX. SP, BP, SI, DI. Стойността, записана за регистър SP, е тази преди изпълнение на инструкцията. PUSHA винаги записва в стека 16 b регистри. За запис на 32 b регистри в стека при i386+ се използва **PUSHAD**.

- при use16 – намалява стойността на указателя на стека SP с 16; последователно се записват в стека стойностите на регистрите AX, CX, DX, BX, SP, BP, SI, DI.
- при use32 – намалява стойността на указателя на стека ESP с 32; последователно се записват стойностите на регистрите EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

PUSHAW: последователно се записват в стека стойностите на регистрите AX, CX, DX, BX, SP, BP, SI, DI (при use16 и при use32).

Следващите три команди изпълняват действия, обратни на по-горе описаните команди:

POPA (само за i186+): извлича 16 B от върха на стека и ги записва в осемте регистъра с общо предназначение в следния ред: Dl, SI, BP, SP, BX, DX, CX и AX. Стойността за регистъра SP не се копира в SP. Инструкцията POPA винаги извлича от стека стойности за 16 b регистри. За извличане на стойности от стека за 32 b регистри при i386+ се използва инструкцията **POPAD**.

POPAW: последователно се извличат от върха на стека стека 16 B и се записват в регистрите DI, SI, BP SP, BX, DX, CX и AX (при use16 и при use32).

Следващата група команди позволява съхраняване в стека на флаговия регистър.

PUSHF/PUSHFD: записва в стека стойността на регистъра с флаговете. PUSHF записва стойност само за 16 b регистър на флаговете. За запис на 32 b регистър на флаговете при i386+ се използва PUSHFD.

Работата на командата зависи от атрибута размера на сегмента:

- при use16 – в стека се записва регистъра flags с размер 2 байта;
- при use32 – в стека се записва регистъра eflags с размер 4 байта.

PUSHFW: работи като PUSHF с атрибут use16.

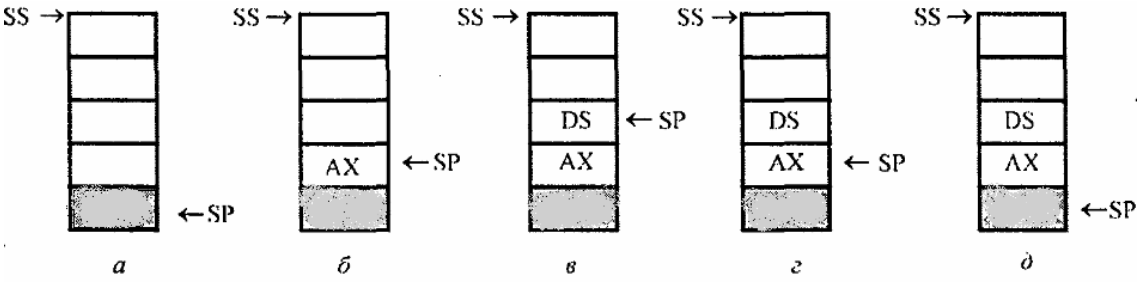
Аналогично, следващите три команди изпълняват действия, обратни на разгледаните по-горе:

POPF/POPFD: извлича стойността от върха на стека в регистъра с флаговете. POPF извлича стойност само за 16 b регистър на флаговете. За извличане на стойност от стека за 32 b регистър на флаговете при i386+ се използва POPFD.

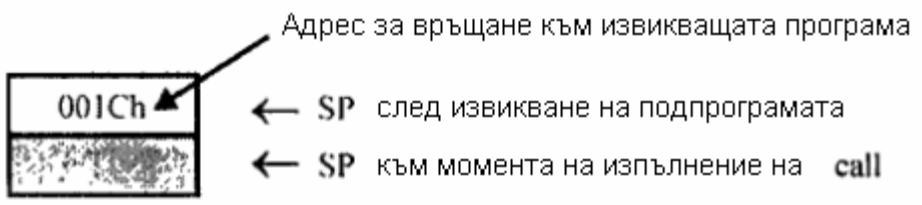
POPFW: работи като POPF с атрибут use16.

Основните видове операции, за които използването на стека е неизбежно, са:

- извикване на подпрограма;
- временно съхранение стойностите на регистрите;
- определяне на локални променливи.



Фигура 14. Организация на стека: а – изходное състояние; б – след зареждане на първия елемент (в дадения пример – съдържимото на регистър AX); в – след зареждане на втория елемент (съдържимото на регистър DS); г – след извличане на един елемент; д – след извличане на два елемента и връщане в изходното състояние



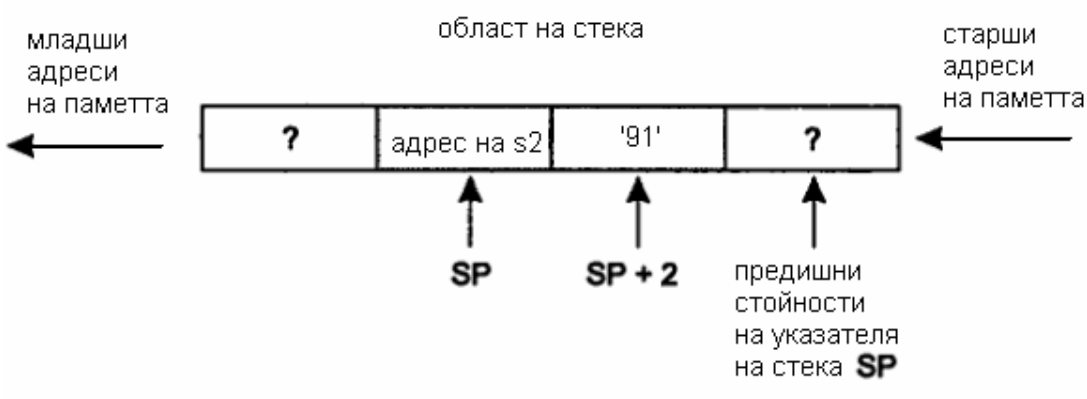
Фигура 15. Състояние на стека след влизане в подпрограма

Следващият пример демонстрира използването на стекови операции. Програмата извежда на екрана последователно съдържимото на символните редове s1, s2 и на променливата num1.

```
model small
.data
        num1 dw '91'
        s1 db "string 1 $"
        s2 db "string 2 $"
```

```
.code
main:
        mov AX, @data
        mov DS, AX
        push DS:num1        ; първо в стека се слага стойността на
                            ; променливата num1
        lea SI, s2
        push SI             ; после се слага адреса на реда s2
                            ; след което указателя на стека се намалява с 4
        lea DX, s1
        mov AH, 9h
        int 21h             ; програмата извежда на екрана s1
        pop DX              ; от стека извлича адреса на s2 и го слага в DX
                            ; указателя се намалява с 2
        int 21h             ; извежда на екрана s2
        pop DX              ; извличане на променливата num1 от стека
        xchg DH, DL         ; следва извеждане съдържимото на DX на екрана,
        ; като се отчита реда на разположение на байтовете в паметта
        mov AH, 2h
        int 21h
        xchg DH, DL
        int 21h

        mov AX, 4c00h
        int 21h
end     main
```



Фигура 16. Съдържимото на стека след зареждане на данните от програмата.

Тема 6. Инструкции за преход и цикъл

Инструкциите за предаване на управлението (за преход) са най-прекият начин за промяна на реда на изпълнение на инструкциите от едно място на програмата в друго. Те променят стойността на брояча на адресите (IP) с отместването на инструкцията, към която се предава управлението, а при преход и в друг сегмент – и на CS с новия сегментен адрес. Има два основни вида инструкции за преход – за безусловен преход и за условен преход.

Инструкция за безусловен преход е инструкцията JMP:

JMP цел: предава безусловно управлението към адреса, посочен от операнда. Преходите могат да бъдат близки (в границите –32768 до +32767 B от инструкцията след прехода), къси (в границите –128 до +127 B) или далечни (в друг кодов сегмент). Ако не е посочена явно отдалеченост на прехода, се избира най-късия възможен преход. При къси и близки преходи операндът задава нов адрес за IP, а при далечни – нови адреси за IP и CS. Адресът за преход е етикет или адрес в паметта, където се намира указателя за преход.

Когато процесорите i386+ са в модел на паметта FLAT, късите преходи са в обхвата от –128 до + 127 B, а близките – от –2 GB до +2 GB.

Както вече споменахме битовете на флаговия регистър EFLAGS / FLAGS отразяват особеностите на резултата от изпълнението на командите.

X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

Таблица 10. Разположение на флаговете във флаговия реистър FLAGS

CF (Carry Flag)
PF (Parity Flag)
AF (Auxiliary Carry Flag)
ZF (Zero Flag)
SF (Sign Flag)
TF (Trace Flag)
IF (Interrupt Flag)
DF (Direction Flag)
OF (Overflow Flag)

Осемнадесетте команди за условен преход реализират преход при изпълнение на определено условие. Такова условие е състоянието на флаговия регистър. Флаговият регистър няма име и не може да фигурира в командите. (С изключение на командите PUSHF/ POPF, съхраняващи и възстановяващи този регистър от стека, също така и LAHF/ SAHF, копираща флаговете от регистъра в AH и обратно).

CMP операнд_1, операнд_2: сравнява два операнда за изпълнение на следващо условно предаване на управлението или на инструкция за условно установяване на байт. Сравнението се прави чрез изваждане на операнда-източник от операнда-приемник и установяване на флаговете в зависимост от резултата. Действието на CMP е аналогично на инструкция SUB, но без съхраняване на резултата.

Флаговете, установявани с командата cmp могат да бъдат анализирани от специални команди за условен преход.

Команда	Стойности на флаговите регистри
JE	ZF=1
JNE	ZF=0
JL / JNGE	SF<>OF
JLE / JNG	SF<>OF или ZF=1
JG / JNLE	SF=OF и ZF=0
JGE / JNL	SF=OF
JB / JNAE	CF=1
JBE / JNA	CF=1 или ZF=1
JA / JNBE	CF=0 и ZF=0
JAE / JNB	CF=0

Таблица 11. Команди за условен преход

JE/JZ	Jump if Equal (Zero)	ZF=1
JNE/JNZ	Not Equal (Non Zero)	ZF=0
JA/JNBE	Above (not Below nor Equal)	ZF=0 и CF=0
JAE/JNB	Above or Equal (not Below)	CF=0
JB/JNAE	Below (not Above nor Equal)	CF=1
JBE/JNA	Below or Equal (Not Above)	ZF=0 или CF=1

Таблица 12. Преход при беззнакови данни

JE/JZ	Jump if Equal (Zero)	ZF=1
JNE/JNZ	Not Equal (Non Zero)	ZF=0
JG/JNLE	Greater (not Less nor Equal)	ZF=0 и SF=0 и OF=0
JGE/JNL	Greater or Equal (not Less)	SF=0
JL/JNGE	Less (not Greater nor Equal)	SF=1
JLE/JNG	Less or Equal (not Greater)	ZF=0 или SF=1

Таблица 13. Преход при знакови данни

JS	Jump if Sign	SF=1
JNS	No Sign	SF=0
JC	Carry	CF=1
JNC	No Carry	CF=0
JO	Overflow	OF=1
JNO	No Overflow	OF=0

Таблица 14. Преход според специални аритметични проверки

За разлика от разгледаните вече команди за условен преход, следващата команда проверява не флаговия регистър, а стойността на регистър CX:

JCXZ <етикет> (Jump if cx is Zero) Преход, при CX=0;

Предава управлението към посочения етикет, ако стойността в CX е нула. При i386+ може да се използва JECXZ за преход при ECX=0.

JECXZ <етикет> (Jump Equal ecx Zero) Преход, при ECX=0;

Ако стойността на регистъра-брояч не е нула, изпълнението продължава от следващата инструкция. Етикетът, посочен като операнд, трябва да е къс (в границите –128 до +127 B от инструкцията след преход).

Вместо последователността от команди: DEC CX ; JNZ <етикет> се използва командата:

LOOP <етикет>

Осигурява циклично изпълнение в обхвата на посочен етикет. LOOP изважда 1 от CX (без да променя флаговете) и ако резултатът не е 0, предава управлението към адреса, зададен с операнда. При i386+ LOOP използва 16 b CX за 16 b режим и 32 b ECX за 32 b режим. Това действие по

подразбиране може да бъде отменено с LOOPW (за CX) или LOOPD (за ECX). Ако стойността на CX след изваждането на 1 е нула, изпълнението продължава със следващата инструкция. Преди влизане в цикъла в CX трябва да се запише максималният брой на повторения. Операндът трябва да посочва къс етикет (в границите –128 до +127 B от инструкцията след LOOP).

Тази команда позволява организация на цикли, подобни на циклите for в езиците от високо ниво с автоматично намаляване брояча на цикъла.
Работата на командата се състои в изпълнението на следните действия:

- Декремент на регистър ecx/cx;
- Сравнение на регистър ecx/cx с нулата:
  - при (ecx/cx) > 0, управлението се предава на етикета за преход;
  - при (ecx/cx) = 0, управлението се предава на следващата след loop команда.

Командата JCXZ обикновено се използва преди цикъла LOOP, за да предотврати изпълнението на цикъла при CX равно на нула.

Има две допълнителни команди, които освен стойността на CX проверяват и тази на ZF.

LOOPE/LOOPZ/LOOPEW/LOOPZW/LOOPED/LOOPZD <етикет>:

Осигурява циклично изпълнение в обхвата на посочен етикет при изпълнени условия ZF=1 и CX>0. При i386+ се използва 16 b CX за 16 b режим и 32 b ECX за 32 b режим. Това действие по подразбиране може да бъде отменено със суфикс W (за CX) или D (за ECX). Инструкцията изважда 1 от CX (без да променя флаговете) и ако резултатът не е 0 и флагът ZF е установен в 1 от предишна инструкция (например CMP), предава управлението към адреса, зададен с операнда. Ако условията не са верни, изпълнението продължава със следващата инструкция. Преди влизане в цикъла в CX трябва да се запише максималният брой на повторения.

LOOPE /LOOPZ (Loop till cx <> 0 or Zero Flag = 0) Командите loope и loopz са идентични. Те работят по следния алгоритъм:

- декремент на регистър ecx/cx;
- сравняване на регистър ecx/cx с нулата;
- анализ състоянието на нулевия флаг zf:
  - ако (ecx/cx) > 0 и zf = 1, управлението се предава на етикета за преход;

- ако (ecx/cx) = 0 или zf = 0, управлението се предава на следващата след loop команда.

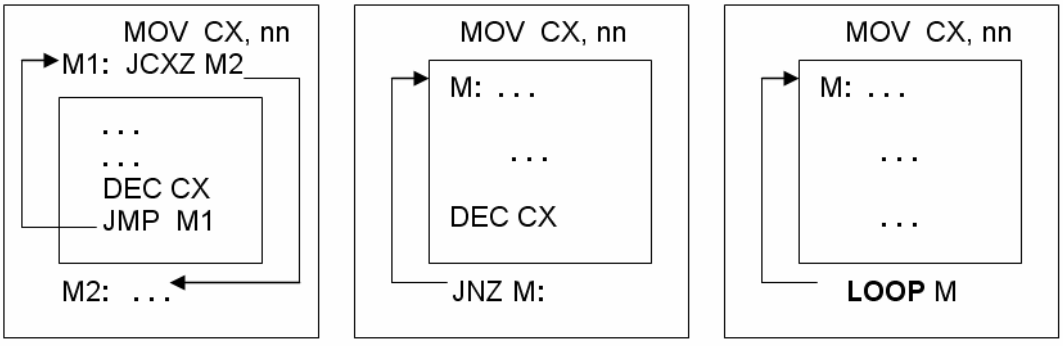
LOOPNE/LOOPNZ/LOOPNEW/LOOPNZW/LOOPNED/LOOPNZD: осигурява циклично изпълнение в обхвата на посочен етикет при изпълнени условия ZF=0 и CX>0. При i386+ се използва 16 b CX за 16 b режим и 32 b ECX за 32 b режим. Това действие по подразбиране може да бъде отменено със суфикс W (за CX) или D (за ECX). Инструкцията изважда 1 от CX (без да променя флаговете) и ако резултатът не е 0 и флагът ZF е установен в 0 от предишна инструкция (например CMP), предава управлението към адреса, зададен с операнда. Ако условията не са верни, изпълнението продължава със следващата инструкция. Преди влизане в цикъла в CX трябва да се запише максималният брой на повторения.

LOOPNE/LOOPNZ (Loop till cx <> 0 or Not Zero flag=0). Командите loopne и loopnz са идентични. Те работят по следния алгоритъм:

- декремент на регистър ecx/cx;
- сравняване на регистър ecx/cx с нулата;
- анализ състоянието на нулевия флаг zf:
  - ако (ecx/cx) > 0 и zf = 0, управлението се предава на етикета за преход;
  - ако (ecx/cx)=0 или zf=1, управлението се предава на следващата след loop команда.

Командите LOOP / LOOPE / LOOPZ / LOOPNE / LOOPNZ , както и JCXZ в качеството на брояч могат да използват само регистър CX.

Недостатък на командите за организация на цикъл **loop, loope/loopz** и **loopne/loopnz** е, че те реализират само къси преходи (от –128 до +127 байта). За реализация на дълги цикли се налага използване на командите за условен преход и на командата *jmp*.



Фигура 17. Организация на цикли в асемблер

Пример: Програма, проверяваща за въведен символ в буфера на клавиатурата.

```
model small
.stack  256
.code
main:
        mov ax, @data
        mov ds, ax

        ;проверка за наличие на символ
m1:
        mov ah, 01h
        int 16h
        jz exit                         ;буферът е празен

        ;извличане на символ
        mov ah, 00h
        int 16h

        ;извеждаме го със средствата на MS-DOS
        mov dl, al
        mov ah, 02h
        int 21h
        jmp m1
exit:
        mov ax, 4c00h
        int 21h
end     main
```

Пример: Преброяване на нулевите байтове (нулевите елементи) на масив с използване на команди за управление на цикъл.

```
model small
.stack  100h
.data
        len equ10                       ;брой на елементите в mas
        mas db 1, 0, 9, 8, 0, 7, 8, 0, 2, 0
.code
start:
        mov ax, @data
        mov ds, ax
        mov cx, len
        xor ax, ax
        xor si, si
        jcxz exit
cycl:
        cmp mas[si], 0
        jne m1
        inc al                          ;в al брояч на нулевите елементи
m1:
        inc si                          ;преход към следващия елемент
```

```
        loop cycl
exit:
        mov ax, 4c00h
        int 21h
end     start
```

Тема 7. Аритметични операции с цели двоични числа

Цяло двоично число е число, кодирано в двоична бройна система. Размерността на цялото двоично число може да бъде 8, 16 или 32 бита. Знакът на двоичното число се определя от това, как се интерпретира старшия бит в представянето на числото. Това са 7-ия, 15-ия или 31-ия бит за числата със съответната размерност. Факт е, че сред аритметичните команди има само две, отчитащи старшия разряд като знаков – това са командите за целочислено умножение и деление (imul и idiv). В останалите случаи отговорността за действията със знакови числа и със знаковия разряд е на програмиста. Диапазонът от стойности на двоичното число зависи от размера му и от това как се интерпретира старшия бит – като старши значещ бит на числото или като знаков бит на числото (табл. 15).

Размерност на данните	Цяло без знак	Цяло със знак
байт	0. . . 255	–128. . . +127
дума	0. . . 65 535	–32 768. . . +32 767
двойна дума	0. . . 4 294 967 295	–2 147 483 648. . . +2 147 483 647

Таблица 15. Диапазон от стойности на двоичните числа

7.1. Представяне на числа със знак

В компютъра положителните числа се представят в двоичен код, а отрицателните – в допълнителен код.

До този момент се предполагаше, че числата са положителни. А как се представят в компютъра числа със знак?

Положителните цели със знак – това са 0 и всички положителни числа. Отрицателните цели със знак – това са всички числа, по-малки от 0. Отличителен признак на число със знак представлява интерпретирането на старшия бит на полето, представляващо числото. В качеството на поле могат да представляват байт, дума или двойна дума. Физически този бит по нищо не се отличава от другите – всичко зависи от командата, обработваща даденото поле. Ако в нейния алгоритъм е заложена възможност за работа с цели числа със знак, тогава командата ще интерпретира по-различно старшия бит на полето. Ако старшият бит е 0, числото се счита за положително и неговата стойност се изчислява по правилата, които разгледахме вече. Ако старшият бит е 1, числото е отрицателно, а това предполага, че то е записано в така наречения *допълнителен код.*

Допълнителният код на отрицателното число представлява резултат от инвертирането (замяната на 1 с 0 и обратно) на всеки бит на двоичното число, равен на модула на изходното отрицателно число плюс единица.

Да разгледаме десетичното число -185_{10}. Модулът му в двоичен вид е равен на 10111001_2. Първо е необходимо да се допълни тази стойност отляво с нули до съответната размерност – байт, дума или двойна дума. В нашия случай е необходимо да се допълни до дума, понеже диапазонът за представяне на знакови числа в един байт е –128. . +127.

Следващото действие е получаването на *двоичното допълнение.* Затова всички разряди на двоичното число е необходимо да се инвертират:

$$\mathbf{000000001011\ 1001_2 \rightarrow 111\ 111\ 11010001\ 10_2}$$

След което прибавяме единица:

$$\mathbf{1111\ 11\ 1\ 1010001\ 10_2 + 0000000000000001_2 = 11111\ 11\ 1010001\ 11_2}$$

Резултатът от преобразуването е равен на 1111111101000111_2.

Именно така се представя числото -185_{10} в компютъра.

При работа с числа със знак вероятно ще се наложи използване на изпълнение на обратното действие – на двоичното допълнение на числото да се определи стойността на неговия

модул. За целта е необходимо изпълнението на следните две действия:

- Инвертиране на битовете на двоичното допълнение.

- Към полученото двоично число да се прибави двоична единица.

Ще определим модула на двоично представеното число:

1185_{10}=1111111101000111_2

Първо инвертираме битовете:

$11010001\ 11_2$ –> $00000000101\ 11000_2$

Прибавяме двоична единица:

$0000000010\ 111000_2$ + 0000000000000001_2 = $00000000101\ 11001_2$ = |–185|

7.2. Събиране на двоични числа без знак

Микропроцесорът реализира събирането на операндите според правилата на събиране на двоични числа. Проблеми няма докато стойността на резултата не превишава размерността на полето на операнда (виж табл. 1). Например при събиране на операнди с размер един байт резултатът не бива да превишава числото 255. Иначе резултатът е неверен. Например събираме: 254 + 5 = 259 в двоичен вид: 11111110 + 0000101 = 1 00000011. Резултатът е преминал границата на осемте бита и верният резултат заема 9 бита, а в 8-битовото поле на операнда е останала стойността 3, което е неверен резултат. В микропроцесора този изход при събирането се прогнозира и са предвидени специални средства за тяхната обработка. В дадения случай се преглежда флага за пренос CF, който фиксира факта за пренос на единица от старшия разряд на операнда. Един лесен вариант е използването на командата за условен преход JC.

В системата от команди на микропроцесора има три команди за двоично събиране:

INC операнд: прибавя 1 към стойността на операнда. Понеже стойността на операнда се разглежда като цяло число без знак, инструкцията не въздейства върху флага CF.

ADD операнд_1, операнд_2: събира операндите източник и приемник и записва резултата в операнда-приемник.

ADC операнд_1, операнд_2: събира операндите източник, приемник и стойността на флага CF и записва резултата в операнда-приемник. Използва се за събиране на числа с дължина няколко байта (или няколко думи).

7.3. Събиране на двоични числа със знак

На практика микропроцесорът не подозира за различието между числа със знак и без знак. Вместо това той има средства за фиксиране възникването на характерни ситуации в процеса на изчисленията. Едно такова средство е регистрацията на състоянието на старшия (знаков) разряд на операнда, което става с флага за препълване OF.

Пример:

30566 = 01110111 01100110
+
00687 = 00000010 10101111
=
31253 = 01111010 00010101

Следим за наличие на пренос от 14-ия и 15-ия разряд и дали е правилен резултата: пренос няма, резултатът е правилен.

Пример:

30566 = 01110111 01100110
+
30566 = 01110111 01100110
=
61132 = 11101110 11001100

Наличие на пренос от 14-тия разряд; от 15-ти разряд няма пренос. Резултатът е неправилен, понеже има препълване – полученото число е по-голямо от +32 767 или това, което може да има 16-битово число със знак.

Пример:

–30566 = 10001000 10011010
+
–04875 = 11101100 11110101
=
–35441 = 01110101 10001111

Наличие на пренос от 15-тия разряд; от 14-тия разряд няма пренос. Резултатът не е правилен, вместо отрицателно число е получено положително (в старшия бит има 0).

Пример:

–4875 = 11101100 11110101
+
–4875 = 11101100 11110101
=
–9750 = 11011001 11101010

Има пренос от 14 и 15-тия разряд. Резултатът е правилен.

С тези примери изяснихме, че препълване (OF= 1) има при пренос:

- от 14-тия разряд (за положителни числа със знак);
- от 15-тия разряд (за отрицателни числа).

И обратното, препълване няма (OF = 0), ако има пренос едновременно от двата разряда или пренос няма и в двата разряда.

Следва демонстрация на събиране на двоични еднобайтови числа без знак:

```
model small
stack 256
.data
        a db 254
.code
main:
        mov ax, @data
        mov ds, ax

        xor ax, ax
        add al, 17
        add al, a
        jnc m1                  ; ако няма перенос, преход към m1
        adc ah, 0               ; в ax сумата с отчитане на переноса
m1:
        exit:
        mov ax, 4c00h
        int 21h
end     main
```

7.4. Изваждане на двоични числа без знак

Ако умаляемото е по-голямо от умалителя то тогава разликата е положително число и резултатът е верен. Ако умаляемото е по-малко от умалителя тогава възниква проблем: резултатът е отрицателен, а това е вече число със знак. Микропроцесорът заема единица от разряда, следващ след старшия, в разрядната решетка на операнда.

Пример:

```
  05 = 00000000 00000101
–10 = 00000000 00001010
```

За да извадим вземаме въображаем заем от старшия разряд:

```
100000000 00000101
–
  00000000 00001010
=
  11111111 11111011
```

По този начин изпълняваме действието

(65 536 + 5) – 10 = 65 531,

0 тук е като еквивалент на числото 65 536. Резултатът, разбира се е неверен, но микропроцесорът счита че всичко е наред, макар фактът на заем на единица той да фиксира с вдигане на флага за пренос cf. Погледнете още веднъж по-внимателно резултата от операцията изваждане. Но това е –5 в допълнителен код. Ще направим следния експеримент: ще представим разликата във вид на сумата 5 + (–10).

Ако напишем разликата като сума:

Пример:

```
5      =  00000000 00000101
+
(–10) =  11111111 11110110
=
          11111111 11111011
```

т. е. получихме същия резултат. По този начин, след командата за изваждане на числа без знак е необходимо да се анализира флага cf. Ако той е 1, тогава е имало заем от старшия разряд и резултатът се е получил в допълнителен код.

Командите за изваждане са следните:

DEC операнд: изважда 1 от стойността на операнда. Понеже стойността на операнда се разглежда като цяло число без знак, инструкцията не въздейства върху флага CF.

SUB операнд_1, операнд_2: изважда операнда-източник от операнда-

приемник и съхранява резултата в операнда-приемник.

SBB операнд_1, операнд_2: добавя стойността на флага CF към втория операнд и след това изважда получената стойност от първия операнд. Резултатът се записва в първия операнд. Използва се за изваждане на числа с дължина няколко байта (или няколко думи).

7.5. Изваждане на двоични числа със знак

Последният пример показва, че микропроцесорът няма нужда от две устройства за събиране и изваждане – достатъчно е само едно.

Пример:

45 – (–127)

45 = 0010 1101
–
–127 = 1000 0001
=
–44 = 1010 1100

Получава се отрицателен резултат, равен на –44. Правилният е 172. Тук имаме препълване и значещият разряд е променил знаковия на операнда. Проследяване на подобна ситуация може да се реализира с анализ на флага за препълване OF.

Пример:

–45 – 45 = –45 + (–45)= –90

–45 = 1101 0011
+
–45 = 1101 0011
=
–90 = 1010 0110

Тук OF = 0, а 1 в знаковия разряд говори, че резултатът е число в допълнителен код.

Пример: Демонстрация на изваждане на двоични еднобайтови числа без знак.

```
model small
stack 256
.data
.code
main:
        mov ax, @data
        mov ds, ax

        xor ax, ax
        mov al, 5
        sub al, 10
        jnc m1                  ; проверка за наличие на пренос
        neg al                  ; в al модула на резултата
m1:
exit:
        mov ax, 4c00h
        int 21h
end     main
```

От тези примери става ясно, че командите SUB и ADD не различават знакови и беззнакови данни, но въздействат на флаговете AF, CF, OF, PF, SF и ZF.

7.6. Умножение на числа без знак

За умножение на числа без знак е предназначена командата:

MUL множител_1: умножава операнд-приемник по подразбиране с посочения операнд-източник. Двата операнда се разглеждат като числа без знак. Ако е посочен само един операнд с дължина 16 b, приемникът по подразбиране е в AX, а произведението се записва в двойката регистри DX:AX. Ако е посочен само един операнд с дължина 8 b, приемникът по подразбиране е в AL, а произведението се записва в AX. При i386+, ако източникът е EAX, то произведението се записва в двойката регистри EDX:EAX. Флаговете CF и OF се установяват в зависимост от резултата.

Вторият операнд е неявен. Неговото местоположение е фиксирано и зависи от размера на множителите. Понеже в общия случай резултатът от умножението е по-голям, отколкото всеки от множителите, неговият размер и местоположение следва също да бъдат определени еднозначно.

множител_1	множител_2	Резултат
Байт	al	16 бита в ax: al – младшата част на резултата; ah – старшата част на резултата
Дума	ax	32 бита в dx:ax: ax – младшата част на резултата;

		dx – старшата част на резултата
Двойна дума	eax	64 бита в edx:eax: eax – младшата част на резултата; edx – старшата част на резултата

Таблица 16. Разположение на операндите и на резултата при умножение

При умножение е интересен въпросът как динамично (по време на изпълнение на програмата) може да се проследи дали резултатът е достатъчно малък и е в един регистър или е превишил размерността на регистъра и старшата му част е в друг регистър. За тази цел се анализират пак флаговете за пренос CF и за препълване OF:

- Ако старшата част на резултата е равна на нула, тогава след операцията произведение CF=0 и OF = 0;
- Ако тези флагове са ненулеви това означава, че резултатът е излязъл от рамките на младшата част на произведението и се състои от две части.

Пример: Демонстрация на умножение на две еднобайтови числа без знак. Сегментът за данни съдържа директива label. Тя задава още едно символично име rez на същия адрес, към който сочи идентификаторът rez_l. Разликата е в типа на тези идентификатори — rez има тип дума, който му е определен от директивата label. Ползвайки тази директива в програмата, ние сме се подготвили за това, че е възможно резултатът от операцията умножение да заема дума в паметта. Тук не се нарушава принципа: младшият байт на младшия адрес. По-надолу в текста на програмата, използвайки името rez, е възможно обръщение към тази стойност като към дума.

```
model small
stack 256
.data
        rez label word
        rez_l db 45
        rez_h db 0
.code
main:
        mov ax, @data
        mov ds, ax

        xor ax, ax
        mov al, 25
        mul rez_l
        jnc m1                  ; ако няма препълване към m1
        mov rez_h, ah           ; старшата част на резултата в rez_h
m1:
        mov rez_l, al           ; младшата част на резултата в rez_l
```

```
exit:
        mov ax, 4c00h
        int 21h
end     main
```

7.7. Умножение на числа със знак

За умножение на числа със знак е предназначена командата:

IMUL операнд_1: умножава операнд-приемник по подразбиране с посочения операнд-източник. Двата операнда се разглеждат като числа със знак. Ако е посочен само един операнд с дължина 16 b, приемникът по подразбиране е в AX, а произведението се записва в двойката регистри DX:AX. Ако е посочен само един операнд с дължина 8 b, приемникът по подразбиране е в AL, а произведението се записва в AX. При i386+, ако източникът е EAX, произведението се записва в двойката регистри EDX:EAX. Флаговете CF и OF се установяват, ако произведението се разширява знаково в DX за 16 b операнди, в AX – за 8 b или в EDX – за 32 b.

За процесори i186+ инструкцията има две допълнителни синтактически форми. При формата с два операнда първият операнд е 16 b регистър (множимо) и в него се записва резултатът, а вторият операнд е константа (множител). При формата с три операнда първият операнд е 16 b регистър (произведение), вторият е 16 b регистър или операнд–памет (множимо), а третият – константа (множител). И в двата случая флаговете OF и CF се установяват, ако резултатът не може да се побере в 16 b регистър. И двете форми могат да се използват както за числа със знак, така и за числа без знак. За i386+ операндите могат да бъдат 16 или 32 B.

За i386+ съществува четвърта синтактическа форма с посочване на операнди-приемник и източник, като източникът може да бъде 16 или 32 b операнд-памет или регистър, а приемникът – регистър със същия размер. Флаговете OF и CF се установяват в 1, ако произведението не може да се побере в приемника.

Тя е аналогична на mul. Отличителна особеност е само формирането на знака. Ако резултатът се събира в един регистър (CF = OF = 0), тогава съдържимото на другия регистър (старшата част) представлява разширение на знака – всичките му битове са равни на старшия бит (знаковия разряд) на младшата част на резултата. В противен случай (CF = OF = 1) знакът на резултата е знаковият бит на старшата част на резултата, а знаковият бит на младшата част е значещ бит на двоичния код на резултата.

7.8. Деление на числа без знака

За деление на числа без знак е предназначена командата:

DIV делител: дели операнд-приемник по подразбиране с посочения операнд-източник. Двата операнда се разглеждат като числа без знак. Ако операндът-източник (делителят) е с дължина 16 b, приемникът по подразбиране (делимото) е в двойката регистри DX:AX. Частното се записва в AX, а остатъкът – в DX. Ако източникът е с дължина 8 b, делимото по подразбиране е в AX, а частното се записва в AL и остатъкът – в AH При i386+, ако източникът е EAX, частното се записва в EAX, а остатъкът в EDX.

Делителят може да се намира в паметта или в регистър и да има размер 8, 16 или 32 бита. Местоположението на делимото е фиксирано и зависи от размера на операндите. Резултат на командата деление са стойностите на частното и остатъка.

Делимо	Делител	Частно	Остатък
16 бита в регистър ax	Байт регистър или клетка от паметта	Байт в регистър al	Байт в регистър ah
32 бита dx – старшата част ax – младшата част	Дума 16 бита регистър или клетка от паметта	Дума 16 бита в регистър ax	Дума 16 бита в регистър dx
64 бита edx – старшата част eax – младшата част	Двойна дума 32 бита регистър или клетка от паметта	Двойна дума 32 бита в регистър eax	Двойна дума 32 бита в регистър edx

Таблица 17. Разположение на операндите и на резултата при деление

След изпълнението на командата деление съдържимото на флаговете не е определено. Прекъсване 0 или “деление на нула” възниква поради една от следните причини:

- Делителят е равен на нула;
- Частното не се събира в зададената му разрядна решетка, което се случва:

  - при делене за делимо с големина дума на делител с големина байт, при което стойността на делимото е по-голяма 256 пъти от стойността на делителя;

- при деление за делимо с големина двойна дума на делител с големина дума, при което стойността на делимото е по-голяма 65 536 пъти от стойността на делителя;
- при деление за делимо с големина четворна дума на делител с големина двойна дума, при което стойността на делимото е по-голяма 4 294 967 296 пъти от стойността на делителя.

Пример: Демонстрация на деление на две еднобайтови числа без знак.

```
model small
stack 256
.data
        del_b label byte
        del dw 29876    ; please change value 29876 on 298 after the experiment
        delt db 45
.code
main:
        mov ax, @data
        mov ds, ax

        xor ax, ax

        ; следващите две команди могат да се заменят с mov ax, del
        mov ah, del_b           ; старшият байт на делимото в ah
        mov al, del_b+1         ; младшият байт на делимого в al
        div delt                ; в al – частното, в ah – остатъка

 exit:
        mov ax, 4c00h
        int 21h
end     main
```

7.9. Деление на числа със знак

За деление на числа със знак е предназначена командата:

IDIV делител: дели операнд-приемник по подразбиране с посочения операнд-източник. Двата операнда се разглеждат като числа със знак. Ако операндът-източник (делителят) е с дължина 16 b, приемникът по подразбиране (делимото) е в двойката регистри DX:AX. Частното се записва в AX, а остатъкът – в DX. Ако източникът е с дължина 8 b, делимото по подразбиране е в AX, а частното се записва в AL и остатъкът – в AH При i386+, ако източникът е EAX, частното се записва в EAX, а остатъкът в EDX.

Аналогията с div е пълна. “Деление на нула” за числа със знак възниква при изпълнение на командата idiv поради следните причини:

- делителят е равен на нула;
- частното не влиза в заделената му разрядна решетка.

Последното е възможно в следните случаи:

- при деление за делимо с големина дума със знак на делител с големина байт със знак, при което стойността на делимото е по-голяма 128 пъти от стойността на делителя (частното трябва да е в диапазона от –128 до +127);
- при деление за делимо с големина двойна дума със знак на делител с големина дума със знак, при което стойността на делимото е по-голяма 32 768 пъти от стойността на делителя (частното трябва да е в диапазона от –32 768 до +32 768);
- при деление за делимо с големина четворна дума със знак на делител с големина двойна дума със знак, при което стойността на делимото е по-голяма 2 147 483 648 пъти от стойността на делителя (частното трябва да е в диапазона от –2 147 483 648 до +2 147 483 647).

7.10. Спомогателни команди за целочислени операции

7.10.1. Команди за преобразуване на типовете

- Команди без операнди – те работят с фиксирани регистри:

CBW: преобразува числото с дължина байт със знак в регистър AL до число с дължина дума със знак в AX чрез разширяване на знаковия бит на AL във всички битове на AH.

CWD: преобразува число със знак (с дължина дума) в регистър AX до число със знак (двойна дума) в двойката регистри DX:AX чрез разширяване на знаковия бит на AX във всички битове на DX.

CWDE (само за i386+): преобразува число с дължина дума със знак от регистър AX в число със знак и двойна дума в EAX чрез разширяване на знаковия бит в старшата дума на EAX

CDQ (само за i386+): преобразува числото с дължина двойна дума със знак в регистър EAX до число с дължина четворна дума със знак в двойката регистри EDX:EAX чрез разширяване на знаковия бит на EAX във всички

битове на EDX.

- Командите *movsx* и *movzx* са за обработка на низ от знаци:

MOVSX операнд_1,операнд_2 (само за i386+): копира и разширява знака на стойността на операнда-източник в операнда-приемник. Използва се за копиране на 8 b или 16 b число със знак в по-голям 16 или 32 b регистър.

Дадената команда е удобна за използване за подготовка на операндите със знак за изпълнение на аритметични действия.

MOVZX операнд_1,операнд_2 (само за i386+): копира и разширява с нули стойността на операнда-източник в операнда-приемник. Използва се за копиране на 8 b или 16 b число без знак в по-голям 16 или 32 b регистър.

Дадената команда е удобна за използване за подготовка на операндите без знак за изпълнение на аритметични действия.

7.10.2. Други полезни команди

XADD цел, източник: събира стойностите на операнда-източник и операнда-приемник и записва резултата в операнда-приемник, като едновременно с това първоначалната стойност на операнда-приемник се премества в източника. Флаговете се установяват в зависимост от резултата.

Командата изпълнява последователно две действия:

- обменя стойностите на двата операнда;
- помества на мястото на операнда цел сумата: цел = цел + източник.

neg операнд – отрицание; изчислява двоичното допълнение на операнда. Командата променя знака на операнда. Физически изпълнява едно действие:

операнд = 0 – операнд,

т. е. изваждане на операнда от нулата. Командата neg операнд може да се прилага в случаите:

- за смяна на знака;
- за изпълнение на изваждане от константа.

Командите SUB и SBB не позволяват да се извади число от константа, понеже константата не може да бъде операнд-приемник в тези операции. Затова дадената операция може да се реализира с помощта на две команди:

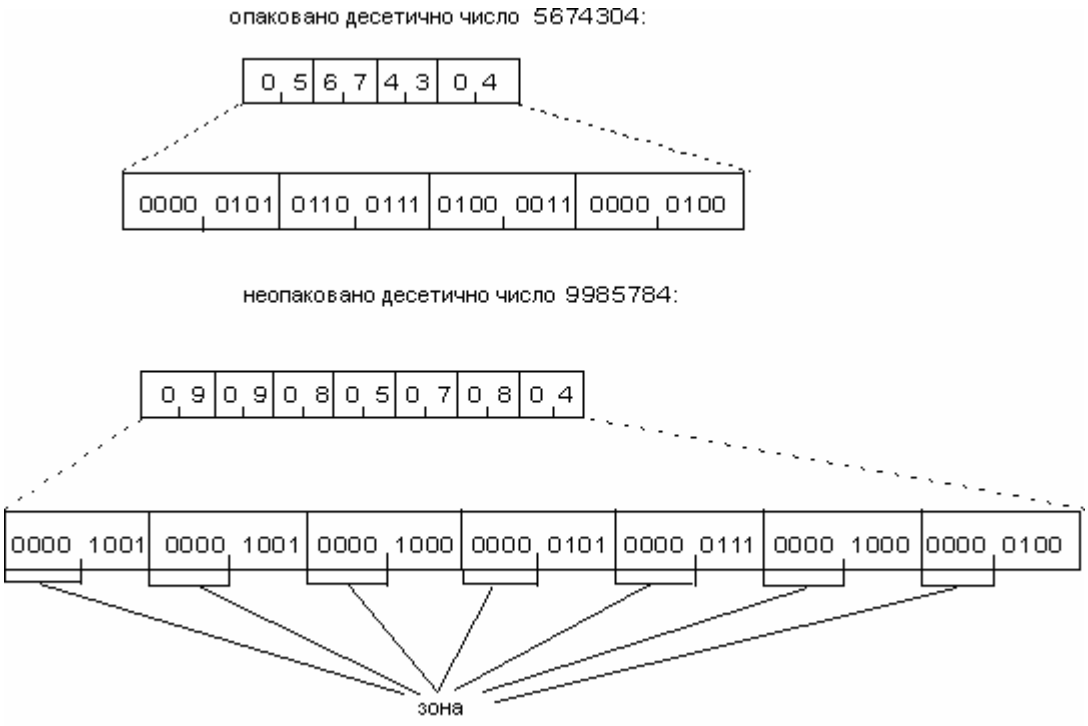
```
neg ax          ;смяна на знака (ax)
add ax, 340     ;изваждането: (ax)=340–(ax)
```

Тема 8. Аритметични операции с двоично-десетични числа

8.1. Въведение в BCD числата

Десетичните числа са специален вид на представяне на числова информация, в основата на който е заложен принципа за кодиране на всяка десетична цифра на числото като набор от четири бита. При това всеки байт на числото съдържа една или две десетични цифри в така наречения двоично-десетичен код (BCD – Binary Coded Decimal). Микропроцесорът съхранява BCD числата в два различни формата:

- опакован формат – в този формат всеки байт съдържа две десетични цифри. Десетичната цифра представлява двоичната стойност в диапазона от 0 до 9 с размер 4 бита. При което кодът на най-старшата цифра на числото заема старшите 4 бита. Следователно диапазонът на представяне на десетичното опаковано (пакетирано) число в един байт е от 00 до 99;
- неопакован формат – в този формат всеки байт съдържа една десетична цифра в четирите си младши бита. Старшите четири бита имат нулева стойност. Наричат сезона. Следователно диапазонът на представяне на десетичното неопаковано (непакетирано) число в един байт е от 0 до 9.



Фигура 18. Представяне на BCD числа

Как се описват двоично-десетичните числа в програмата? За целта могат да се използват само две директиви за описание и инициализация на данни – db и dt. Възможността да се прилагат само тези директиви за описание на BCD числа е обусловена от това, че за тези числа е валиден принципа “младшият байт на младшия адрес”.

BCD числата се използват в тези приложения, които изискват големи (без ограничения за диапазона от стойности) и точни (без грешка при закръгляване) числа. За тази цел двоичните и реалните числа са непригодни.

8.2. Аритметични действия с неопаковани BCD числа

8.2.1. Събиране на неопаковани BCD числа

Пример: Резултатът от събирането не е по-голям от 9

6 = 0000 0110
+
3 = 0000 0011
=

9 = 0000 1001

Пренос от младшата тетрада в старшата няма. Резултатът е правилен.

Пример: Резултатът от събирането е по-голям от 9

06 = 0000 0110
+
07 = 0000 0111
=
13 = 0000 1101

Така се получи вече не BCD число. Резултатът не е правилен. Правилният резултат в неопакован BCD формат следва да бъде:0000 0001 0000 0011 в двоично представяне (или 13 в десетично). Анализирайки дадения проблем при събиране на BCD числа (и подобни проблеми при изпълнение на други аритметични действия) и възможните варианти за разрешаване, създателите на системата команди на микропроцесора са решили не да въвеждат специални команди за работа с BCD числа, а да въведат няколко коригиращи команди. Предназначението на тези команди е да коригират резултата от работата на обикновените аритметични команди в случаите, когато операнди са BCD числа. За корекция на операцията събиране на две едноцифрени неопаковани BCD числа в системата от команди на микропроцесора съществува специална команда:

AAA: коригира резултата след събиране до десетична цифра (0–9). Предходната инструкция за събиране трябва да остави 8-битова сума в AL. Ако сумата е по-голяма от 9h, AH се инкрементира и се установяват флаговете CF и AF.

Тази команда няма операнди. Тя работи неявно само с регистър AL и анализира стойността на младшата му тетрада:

- ако тази стойност е по-малка или равна на 9, тогава флагът CF се нулира и се преминава към следващата команда;
- ако тази стойност е по-голяма от 9, тогава се изпълняват следните действия:

  - към съдържимото на младшата тетрада на AL (а не към съдържимото на целия регистър) се прибавя 6, с което стойността на десетичния резултат се коригира;
  - флагът CF се вдига (CF= 1), с което се фиксира пренос в старшия разряд, за да може да се отчете в следващи действия.

8.2.2. Изваждане на неопаковани BCD числа

Пример: Резултатът от изваждането не е по-голям от 9

6 = 0000 0110
–
3 = 0000 0011
=
3 = 0000 0011

Няма заем от старшата тетрада. Резултатът е верен.

Пример: Резултатът от изваждането е по-голям от 9

6 = 0000 0110
–
7 = 0000 0111
=
–1 = 1111 1111

Изваждането се реализира по правилата на двоичната аритметика. Затова резултатът не е BCD число. Правилният резултат в неопакован BCD формат трябва да е 9 (0000 1001 в двоична бройна система). При това има заем от старшия разряд, т. е. в случая с BCD числа фактически се изпълнява изваждането 16 – 7. Така става ясно, че аналогично на събирането резултатът от изваждането следва да се коригира. За целта има специална команда:

AAS: коригира резултата след изваждане до десетична цифра (0–9). Предходната инструкция за изваждане трябва да остави 8-битов резултат в AL. Ако резултатът е по-голям от 9h, AH се декрементира и се установяват флаговете CF и AF.

Команда AAS също няма операнди и работи с регистър AL, като анализира младшата му тетрада по следния начин:

- ако стойността е по-малка или равна на 9, тогава флагът CF се нулира и управлението се предава на следващата команда;
- ако стойността на тетрадката в AL е по-голяма от 9, тогава командата AAS изпълнява следните действия:

  - от съдържимото на младшата тетрада на регистър AL (а не от съдържимото на регистър AL) се изважда 6;
  - занулява се старшата тетрада на регистър AL;
  - установява се флага CF= 1, с което се фиксира въображаем заем от старшия разряд.

Командата AAS се използва съвместно с командите за изваждане SUB и SBB. При това командата SUB се използва само веднъж при изваждането на най-младшите цифри на операндите, след което следва да се прилага командата SBB, която отчита възможен заем от старшия разряд.

Пример: Демонстрация на събиране на неопаковани BCD числа.

```
model small
stack 256
.data
        len equ 2               ; разрядност на числото
        b db 1, 7               ; неопакованото число 71
        c db 4, 5               ; неопакованото число 54
sum     db 3 dup (0)
.code
main:
        mov ax, @data
        mov ds, ax
        xor bx, bx
        mov cx, len
m1:
        mov al, b[bx]
        adc al, c[bx]
        aaa
        mov sum[bx], al
        inc bx
        loop m1
        adc sum[bx], 0

exit:
        mov ax, 4c00h
        int 21h
end     main
```

Пример: Демонстрация на изваждане на неопаковани BCD числа.

```
model small
stack 256
.data
        b db 1, 7               ; неопакованото число 71
        c db 4, 5               ; неопакованото число 54
        subs db 2 dup (0)
.code
main:
        mov ax, @data
        mov ds, ax
        xor ax, ax
len     equ 2                   ; разрядност на числата
```

```
        xor bx, bx
        mov cx, len             ;в cx брояч на цикъла
m1:
        mov al, b[bx]
        sbb al, c[bx]
        aas
        mov subs[bx], al
        inc bx
        loop m1
        jc m2                   ;анализ на флага за наличие на заем
        jmp exit
m2:     ;. . .
exit:
        mov ax, 4c00h
        int 21h
end     main
```

8.2.3. Умножение на неопаковани BCD числа

Реализацията на умножението и делението е по-сложна. В системата команди на микропроцесора има само средства за умножение и деление на едноцифрени неопаковани BCD числа.

За да се умножат две едноцифрени BCD числа е необходимо:

- да се премести единият множител в регистър AL (както го изисква командата MUL);
- да се премести вторият операнд в регистър или в паметта, заемайки един байт;
- да се умножат множителите с командата MUL (резултатът ще бъде в AX);
- резултатът ще се получи в двоичен код, затова следва да се коригира.

За корекция на резултата след умножение се прилага специална команда:

AAM: преобразува 8-битово двоично число (по-малко от 100D) в AL до непакетирано BCD число в AX. Старшата цифра се записва в AH, а младшата – в AL. Използва се за корекция на произведението след умножение с MUL на непакетирани BCD цифри в AH и AL. Използва се и за корекция на частното след делене с DIV на двоично число (по-малко от 100D) в AX с непакетирано BCD число.

Тя няма операнди и работи с регистър AX по следния начин:

- дели AL на 10;
- резултатът от делението се записва така: частното в AL, остатъкът в AH.

Като резултат след изпълнение на командата AAM в регистрите AL и AH се намират двоично-десетичните цифри на произведението на двете цифри.

Тази команда може да се прилага за преобразуване на двоичното число в регистър AL в неопаковано BCD число, което ще бъде разположено в регистър AX: старшата цифра на резултата в AH, младшата – в AL. Двоичното число следва да е от диапазона 0. . . 99.

Пример за умножение на BCD число с произволна размерност на еднозначно BCD число:

```
model small
stack 256
.data
        b db 6, 7                       ;неопакованото число 76
        c db 4                          ;неопакованото число 4
        proizv db 4 dup (0)
.code
main:
        mov ax, @data
        mov ds, ax
        xor ax, ax
len     equ 2                   ;размерност на първия множител
        xor bx, bx
        xor si, si
        mov cx, len             ;в cx дължината на по-големия множител
m1:
        mov al, b[si]
        mul c
        aam                     ;корекция на умножението
        adc al, dl              ;отчитане на предходния пренос
        aaa                     ;корекция на резултата от събиране с пренос
        mov dl, ah              ;запомняне на преноса
        mov proizv[bx], al
        inc si
        inc bx
        loop m1
        mov proizv[bx], dl      ;отчитане на последния пренос

        mov ax, 4c00h
        int 21h
end     main
```

8.2.4. Деление на неопаковани BCD числа

Процесът на деление на две неопаковани BCD числа се отличава от разгледаните по-горе операции с тях. Тук също има действия за корекция, но те следва да се изпълнят преди основната операция, реализираща непосредственото деление на едното BCD число на другото BCD число. Предварително в регистър AX е необходимо да се получат двете неопаковани BCD цифри на делимото. После е необходимо да използва командата AAD:

AAD: преобразува непакетирани BCD цифри от AH (старша цифра) и AL (младша цифра) до двоично число в AX. Използва се за подготовка на непакетирано BCD число в AX за делене.

Командата няма операнди и преобразува двузначното неопаковано BCD число в регистър AX в двоично число. То впоследствие е делимо в операцията деление. Освен преобразуването, командата AAD премества полученото двоично число в регистър AL. Делимото е двоично число в диапазона 0. . . 99. Алгоритъмът, по който командата AAD реализира това преобразуване, е следния:

- умножаване на старшата цифра на BCD числото в AX (съдържимото на AH) на 10;
- събиране на AH + AL, резултатът от което (двоичното число) се записва в AL;
- зануляване съдържимото на AH.

По-нататък програмистът трябва да използва командата за деление DIV за изпълнение на делението на съдържимото на AX на една BCD цифра, намираща се в регистър с размер един байт или в байт от клетка в паметта.

Аналогично на AAM, командата AAD може да се използва за преобразуване на неопаковани BCD числа от диапазона 0. . 99 в двоичният им еквивалент.

Пример за деление на неопаковани BCD числа:

```
model small
stack 256
.data
        b db 1, 7               ;неопакованото BCD число 71
        c db 4
        rez db 3 dup (0)
.code
main:
        mov ax, @data
        mov ds, ax
```

```
        xor ax, ax
        mov al, b
        mov ah, b+1
        aad                     ;корекция преди делението
        div c                   ;в al BCD частното, в ah BCD остатъка

        mov     rez+1,al
        mov     rez+2,ah
        aaa
        mov     rez,al
        and     rez+1,0fh
        aaa
        mov     rez+1,ah

        mov ax, 4c00h
        int 21h
end     main
```

8.3. Аритметични действия с опаковани BCD числа

Опакованите BCD числа могат само да се събират и изваждат. За да се умножават или делят е необходимо допълнително те да се преобразуват в неопакован формат или в двоично представяне.

8.3.1. Събиране на опаковани BCD числа

Пример: Събиране на опаковани BCD числа (събираме двузначни опаковани BCD числа).

```
 67 = 0110 0111
+
 75 = 0111 0101
=
142 = 1101 1100 = 220
```

В двоичен вид резултатът е равен на 1101 1100 (или 220 в десетичен вид), което не е вярно. Така се получава понеже микропроцесорът не подозира за съществуването на BCD числа и ги събира според правилата за събиране на двоични числа. Резултатът в двоично-десетичен вид трябва да е 0001 0100 0010 (или 142 в десетичен вид).

За корекция на резултата микропроцесорът предоставя командата DAA:

DAA: коригира резултата след събиране до опаковано BCD число (по-малко от 100D). Предишната инструкция за събиране трябва да остави 8 b двоична сума в AL, която се преобразува до пакетиран BCD формат с младша десетична цифра в младшите 4 b и старша десетична цифра – в старшите 4

b. Ако след коригиране сумата е по-голяма от 99h, установяват се флаговете CF и AF, в противен случай те се нулират.

AF е спомагателен флаг за пренос само за BCD числа. Този флаг фиксира факта на заем от младшата тетрада на резултата.

Командата DAA преобразува съдържимото на регистър AL в две опаковани десетични цифри по следния алгоритъм (анализира се наличието на следните ситуации):

- Ситуация 1. Като резултат от предходната команда за събиране флагът AF=1 или стойността на младшата тетрада на регистър AL>9. Флагът AF се вдига в случай на пренос на двоична единица от третия бит на младшата тетрада в старшата тетрада на регистър AL (ако стойността е по-голяма от 0fh).
- Ситуация 2. Като резултат от предходната команда за събиране флагът CF=1 или стойността на регистъра AL>9fh. Флагът CF се вдига в случай на пренос на двоична единица в старшия бит на операнда (ако стойността е по-голяма от 0ffh за регистър AL).

Ако е валидна една от тези ситуации, тогава регистър AL се коригира по следния начин:

- при ситуация 1 съдържимото на регистър AL се увеличава с 6;
- при ситуация 2 съдържимото на регистър AL се увеличива с 60h;
- ако са в сила и двете ситуации, тогава корекцията започва от младшата тетрада.

Пример за събиране на две опаковани BCD числа:

```
model small
stack 256
.data
        b db 17h        ;опакованото число 17h
        c db 45h        ;опакованото число 45h
        sum db 2 dup (0)
.code
main:
        mov ax, @data
        mov ds, ax
        xor ax, ax
        mov al, b
        add al, c
        daa
        jnc $+6         ;преход през една команда, за резултат <= 99
        mov sum+1, ah;отчитане на преноса при събиране (резултатът > 99)
```

```
        mov sum, al     ;младшите упаковани цифри на резултата
        mov ax, 4c00h
        int 21h
end     main
```

Броячът на адреса е специфичен вид операнд. Той се обозначава със знака $. Спецификата на този операнд е в това, че когато транслаторът на асемблер срещне в изходната програма този символ, тогава той го подменя, като вместо него поставя текущата стойност на брояча на адреса. Стойността на брояча или както още го наричат, брояч на отместването, представлява отместването на текущата машинна команда относно началото на кодовия сегмент или сегмента с данни.

8.3.2. Изваждане на опаковани BCD числа

Пример: Изваждане на опаковани BCD числа (микропроцесорът изпълнява изваждане ползвайки събиране).

```
  67 = 0110 0111
+
–75 = 1011 0101
=
  –8 = 0001 1100 = 28 ???
```

Резултатът е равен на 28 в десетична бройна система. В двоично-десетичен код резултатът следва да е 0000 1000 (или 8 в десетична бройна система).

При програмиране на изваждане на опаковани BCD числа, както и при изваждането на неопаковани BCD числа, програмистът е длъжен да контролира знака с помощта на флага CF, който фиксира заем от старшите разряди. Самото изваждане на BCD числа се реализира с командата SUB или SBB. Корекцията на резултата се прави с командата DAS:

DAS: коригира резултата след изваждане до пакетирано BCD число (по-малко от 100D). Предишната инструкция за изваждане трябва да остави 8 b двоична стойност в AL, която се преобразува до пакетиран BCD формат с младша десетична цифра в младшите 4 b и старша десетична цифра – в старшите 4 b. Ако след коригиране стойността е по-голяма от 99h, установяват се флаговете CF и AF, в противен случай те се нулират.

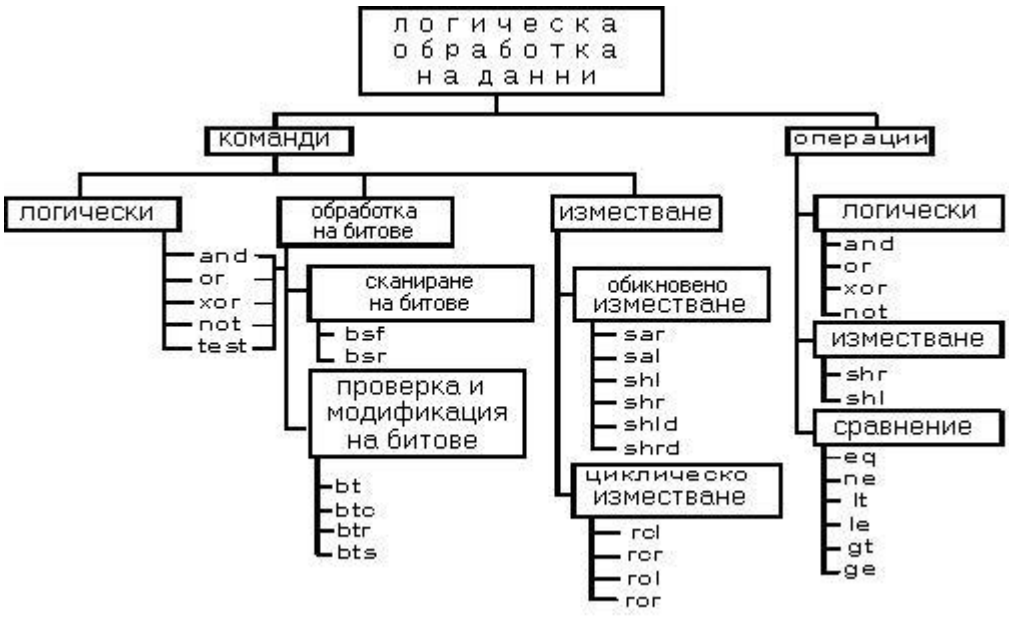
Командата DAS преобразува съдържимото на регистър AL в две опаковани десетични цифри по следния алгоритъм (като анализира наличието на следните ситуации):

- Ситуация 1. Като резултат от предходната команда за събиране флагът AF =1 или стойността на младшата тетрада на регистър AL>9.
- Ситуация 2. Като резултат от предходната команда за събиране флагът CF =1 или стойността на регистъра AL>9fh.

Ако е в сила една от тези ситуации, тогава регистър AL се коригира по следния начин:

- за ситуация 1 съдържимото на регистър AL се намалява с 6;
- за ситуация 2 съдържимото на регистър AL се намалява с 60h;
- ако са в сила и двете ситуации, корекцията започва от младшата тетрада.

Тема 9. Логически команди и команди за изместване



Фигура 19. Команди и операции за логическа обработка на данни

9.1. Логически команди

Логическите операции са представени от командите NOT (инверсия), AND (конюнкция), OR (дизюнкция), XOR (изключващо ИЛИ) и от командата TEST, която изпълнява конюнкция на операндите, без да променя стойностите им. Всички логически операции са битови, т. е. се изпълняват независимо за всички битове на операндите.

Бинарните команди AND, OR, XOR и TEST влияят на флаговете OF, SF, ZF, PF и CF. Унарната операция NOT не влияе на състоянието на флаговете.

Флагът за четност отчита само осемте младши разряда на операнда. PF=1 за четен брой на единиците за осемте младши разряда на операнда и PF=0- за нечетен брой.

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

Таблица 18. Бинарни операции AND, OR и XOR

AND операнд_1, операнд_2: изпълнява поразредна логическа операция AND над операндите източник и приемник и записва резултата в операнда-приемник. Всеки бит на резултата е равен на 1, ако съответните битове на двата операнда са 1, в противен случай резултатът е 0.

Пример:

```
mov dh, 10101100b
and dh, 0f0h
```

В резултат от изпълнението на тези две команди съдържимото на DH ще стане= 10100000b.

OR операнд_1, операнд_2: изпълнява поразредна логическа операция OR над операндите източник и приемник и записва резултата в операнда-приемник. Всеки бит на резултата е равен на 1, ако поне един от

съответните битовете на двата операнда са 1, в противен случай резултатът е 0.

Пример: or bx, dx; ако (BX)=5F0Fh, (DX)=7777h, резултатът е (BX)=7F7Fh

BX	0101 1111 0000 1111 = 5F0F
DX	0111 0111 0111 0111 = 7777
BX (резултат)	0111 1111 0111 1111 = 7F7F

XOR операнд_1, операнд_2: изпълнява поразредна логическа операция "изключващо ИЛИ" (eXOR) над операндите източник и приемник и записва резултата в операнда-приемник. Всеки бит на резултата е равен на 1, ако съответните битовете на двата операнда са различни, в противен случай резултатът е 0.

Пример:

```
xor al, 55h      ;за (AL)=5ah, тогава след операцията (AL)=0fh.
```

NOT операнд: инвертира всеки бит на операнда.

TEST операнд_1, операнд_2: тества посочени битове на операнда и установява флаговете за следващ условен преход или инструкция SET. Единият от операндите съдържа тестваната стойност, а другият – маска от битове, задаваща битовете за тестване. TEST изпълнява поразредно операция AND над двата операнда, като променя флаговете в зависимост от резултата, но не записва резултата в операнда-приемник.

С помощта на логическите команди е възможен анализ на отделни битове в операнда. За организация на подобна индикация често пъти вторият операнд изпълнява ролята на маска.

- За установяване на определени разряди (битове) в **1** се използва командата **or**. В тази команда операнд_2, изпълнява ролята на маска и съдържа единични битове на това място, където трябва да се установят в 1 в операнд_1.
- За сваляне на определени разряди (битове) в **0** се използва командата **and**. В тази команда операнд_2, изпълнява ролята на маска и съдържа нулеви битове там, където следва да се установят в 0 в операнд_1.
- Командата **xor** се използва:
    - За изясняване кои битове в операнд_1 и операнд_2 се **различават**;

- За **инвертиране** състоянието на зададени битове в операнд_1.

  Интересуващите ни битове на маската (операнд_2) при изпълнение на командата xor следва да са единични, а останалите — нулеви.

- За проверка състоянието на зададени битове се прилага командата **test**. Проверяваните битове на операнд_1 в маската (операнд_2) следва да имат стойност единица.

В последните модели на микропроцесорите Intel в групата на логическите команди има още няколко команди, които позволяват достъп до единствен конкретен бит на операнда. Операндът може да се намира както в паметта, така и в регистър с общо предназначение. Положението на бита се задава с отместването му относно младшия бит на операнда. Стойността на отместването може да се задава както непосредствено, така и да се съдържа в регистър с общо предназначения. Като такава стойност може да се ползва резултата от работата на командите BSR и BSF. Всички тези команди присвояват стойността на избрания бит на флаг CF.

BT DST, SRC (Bit Test) (само за i386+): копира стойността на посочен бит във флага CF, където тя може да бъде проверена с инструкция JC или JNC. Операндът-приемник съдържа стойността, в която се съдържа битът, а операндът-източник посочва позицията на бита.

BTS DST, SRC (Bit Test and Set) (само за i386+): копира стойността на посочен бит във флага CF, където тя може да бъде проверена с инструкция JC или JNC, след което битът се установява в 1 в операнда-приемник. Операндът-приемник съдържа стойността, в която се съдържа битът, а операндът-източник посочва позицията на бита.

BTR DST, SRC (Bit Test and Reset) (само за i386+): копира стойността на посочен бит във флага CF, където тя може да бъде проверена с инструкция JC или JNC, след което битът се нулира в операнда-приемник. Операндът-приемник съдържа стойността, в която се съдържа битът, а операндът-източник посочва позицията на бита.

BTC DST, SRC (Bit Test and Convert) (само за i386+): копира стойността на посочен бит във флага CF, където тя може да бъде проверена с инструкция JC или JNC, след което стойността на бита се инвертира в операнда-приемник. Операндът-приемник съдържа стойността, в която се съдържа битът, а операндът-източник посочва позицията на бита.

9.2. Команди за изместване

Командите за изместване по друг начин осигуряват манипулирането с отделни битове на операндите (сравнени с логическите команди). Те се разделят на команди с обикновени (прости) изместванията и команди с циклически изместванията. Циклическите изместванията влияят само на флаговете OF и CF, а обикновените променят пет флага: OF, SF, ZF, PF и CF.

Командите за изместване могат да работят както с байтове, така и с думи.

SAL/SAR/SHL/SHR: измества битовете на операнда-приемник на толкова разреда, колкото е стойността на операнда-източник. SAL и SHL изместват битовете вляво, а SAR и SHR – вдясно.

При SHL, SAL И SHR изместеният бит от края на операнда се копира във флага CF, а най-левият или най-десният бит се попълва с 0. При SAR най-десният изместен бит се копира във флага CF, а най-левият бит запазва предишната си стойност, като по този начин се съхранява знакът на операнда. SAL и SHL са синоними.

При i86/88 операндът-източник може да е CL или 1. При i186+ операндът-източник може да бъде CL или 8 b константа. Флагът за препълване OF се променя само за 1-битови изместванията, за многобитови той остава недефиниран

RCL/RCR/ROL/ROR: измества циклично битовете на операнда-приемник на толкова разреда, колкото е стойността на операнда-източник. RCL и ROL ротират битовете вляво, а RCR и ROR – вдясно.

ROL и ROR изместват битовете на операнда циклично на зададения брой разреди. При всяко изместване най-левият или най-десният бит се копира в флага CF и се ротира. RCL и RCR ротират през флага за пренос CF. CF всъщност става разширение на операнда, така че при 8 b операнди се прави 9 b ротация, а за 16 b операнди –17 b ротация.

При i86/88 операндът-източник може да е CL или 1. При i186+ операндът-източник може да бъде CL или 8 b константа. Флагът за препълване OF се променя само за 1-битови ротации, за многобитови той остава недефиниран.

Командите ROL и ROR реализират просто циклическо изместване наляво и надясно, преместваики стойността от съответния бит в освободилия се бит.

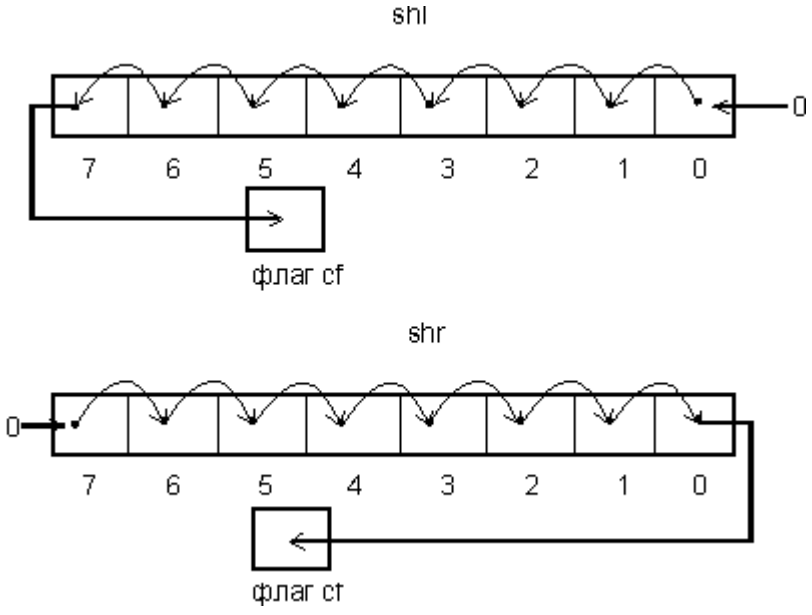
Командите RCL и RCR се наричат команди за циклическо изместване наляво и надясно с пренос, тъй като флаг CF разширява преместваемия операнд с един бит. По този начин стойността от CF се зарежда в освободилия се бит, а преместваният бит се помества в CF.

Командите SHL и SHR реализират логическо изместване наляво и надясно. За логическото изместване е характерно, че в освободения бит се зарежда нула, а преместваемият бит се губи.

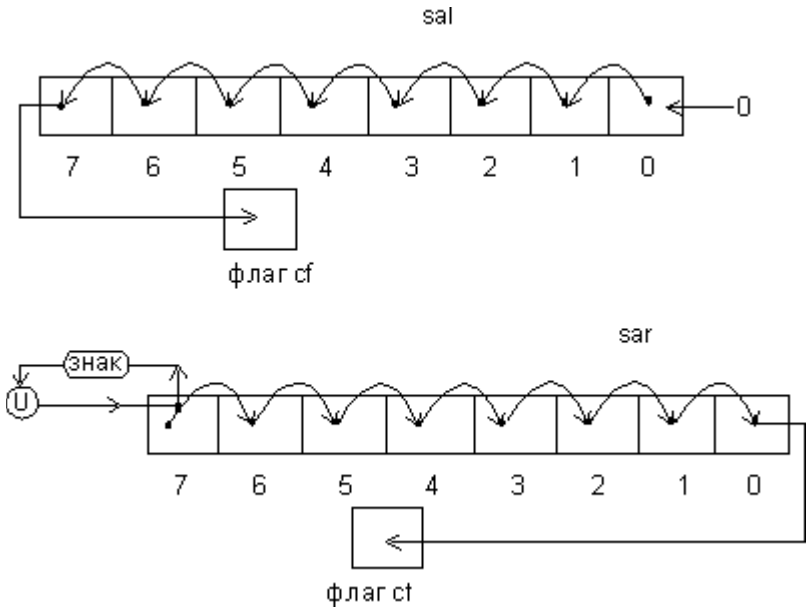
Командите SAL и SAR са предназначени за аритметично изместване наляво и надясно. Аритметичното изместване надясно се различава от логическото изместване по това, че знаковият бит не се премества, а се дублира в съседния десен бит, запазвайки по този начин самия знак на числото. Аритметичното изместване наляво е еквивалентно на логическото, затова мнемониките SAL и SHL означават една и съща машинна команда.

Командите за аритметично изместване по същество реализират умножение и деление на числа без знак на степен на числото 2.

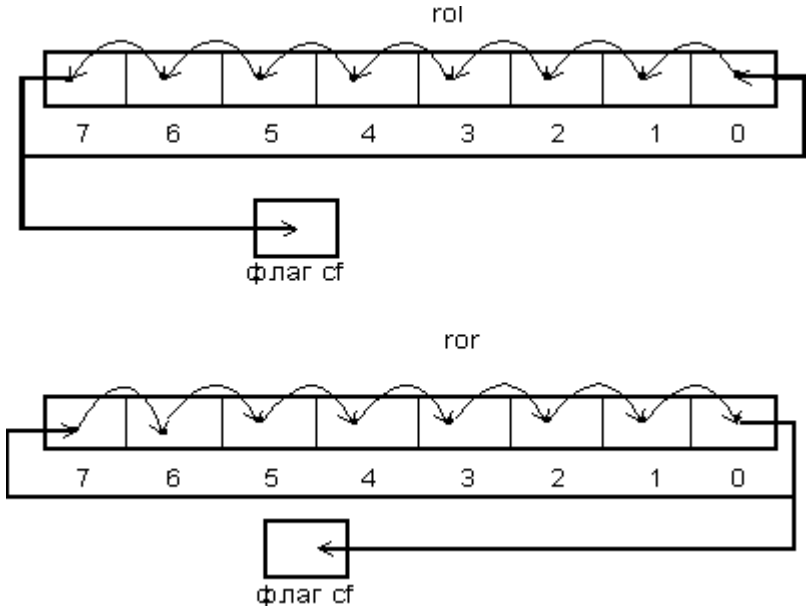
Броят на изместваните битове се задава статично (във втория операнд: CNT) или динамично (в CL).



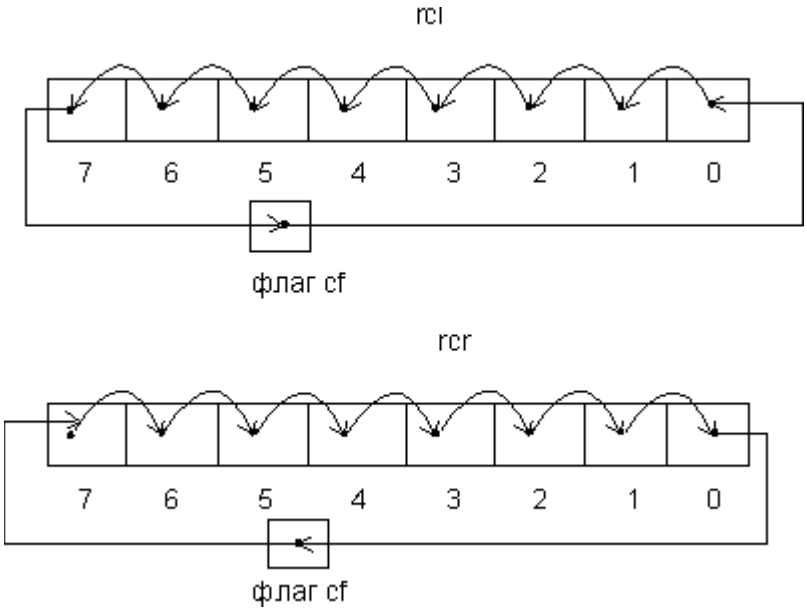
Фигура 20. Схема на работа на командите за линейно логическо изместване



Фигура 21. Схема на работа на командите за аритметично логическо изместване



Фигура 22. Схема на работа на командите за обикновено циклическо изместване



Фигура 23. Схема на работа на командите за циклическо изместване с cf

Пример:

```
mov bl, 10110010b     ;(CF) = x
shr bl, 1             ;(BL) = 01011001, (CF) = 0
```

Пример:

```
mov cl, 4
shr bl, cl            ;(BL) = 00000101, (CF) = 1.
```

Следващите две команди се използват за търсене на първия бит=1 в операнда. Ако такъв бъде открит, неговият номер се записва в първия операнд. Ако всички битове са =0, тогава ZF=1, иначе ZF=0.

BSR операнд_1, операнд_2 (Bit Scaning Reset) (само за i386+): сканира операнда-източник, търсейки бит със стойност 1. Търсенето е в посока от най-старшия бит към бит с индекс 0. Ако се срещне бит със стойност 1, флагът ZF се нулира, а в операнда-приемник се зарежда индексът на първия срещнат ненулев бит. Ако липсват битове със стойност 1, флагът ZF=0.

BSF операнд_1, операнд_2 (Bit Scaning Forward) (само за i386+): сканира операнда-източник, търсейки бит със стойност 1. Търсенето е в посока от бит с индекс 0 към най-старшия бит. Ако се срещне бит със стойност 1, флагът ZF се нулира, а в операнда-приемник се зарежда индексът на първия срещнат ненулев бит. Ако липсват битове със стойност 1, флагът ZF=0.

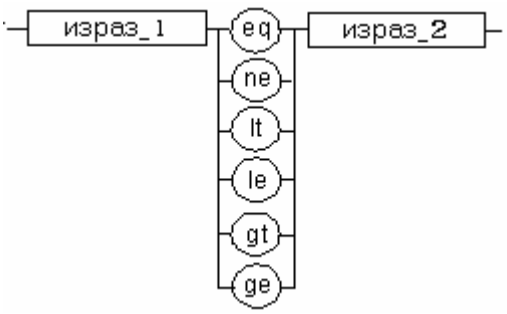
Моделите процесори от 80386 и нагоре включват команди за изместване с двойна точност:

SHLD/SHRD (само за i386+): измества битовете на втория операнд в първия операнд. Броят на изместените битове се определя от третия операнд. SHLD измества първия операнд вляво с брой разреди, определени от брояча. Освободените от изместването разреди се попълват с най-старшите битове на втория операнд. SHRD измества първия операнд вдясно с брой разреди, определени от брояча. Освободените от изместването разреди се попълват с най-младшите битове на втория операнд. Броячът трябва да бъде зададен в CL или с 8 b константа. Ако броячът има стойност, по-голяма от 31, стойността му се коригира, като се използва остатъкът на стойността по модул 31.

Пример:

```
mov EBX, 0FFC8000h
mov ESI, 12340000h
mov CL, 16
shld EBX, ESI, CL       ; резултатът е EBX=80001234h.
```

9.3. Оператори за сравнение



Фигура 24. Синтаксис на операторите за сравнение

Оператор	Резултат
eq	ИСТИНА, за *израз_1* равен на *израз_2*
ne	ИСТИНА, за *израз_1* не равен на *израз_2*
lt	ИСТИНА, за *израз_1* по-малък от *израз_2*
le	ИСТИНА, за *израз_1* по-малък или равен на *израз_2*
gt	ИСТИНА, за *израз_1* по-голям от *израз_2*
ge	ИСТИНА, за *израз_1* по-голям или равен на *израз_2*

Таблица 19. Оператори за сравнение

Пример:

```
tab_size equ 50              ;размер на таблицата
…
mov al, tab_size ge 50       ;зареждане размера на таблицата в al
cmp al, 0                    ;за tab_size < 50
je m1                        ;преход към m1
…
m1:
```

В този пример за tab_size по-голямо или равно на 50, резултатът в al = 0ffh, а за tab_size по-малък от 50, тогава al = 00h.

Задача: Да се преобразува неопаковано BCD число в опаковано BCD число с командите за изместване.

```
model small
stack 256
.data
        len=4                       ;дължина на неопакованото BCD число
        unpck_BCD label dword
        dig_BCD db 2, 4, 3, 6       ;неопакованото BCD число 6342
        pck_BCD dd 0                ;pck_BCD=00006342
.code
main:
        mov ax, @data
        mov ds, ax
        xor ax, ax
        mov cx, len

.386
        mov eax, unpck_BCD
m1:
        shl eax, 4                  ;махаме нулевата тетрада
        shld pck_BCD, eax, 4        ;тетрадата с цифрата в полето pck_BCD
        shl eax, 4                  ;махаме тетрадата с цифрата от eax
        loop m1
```

```
exit:                                   ;pck_BCD=00006342
        mov ax, 4c00h
        int 21h
end     main
```

Тема 10. Преобразуване на числата

Компютърът възприема само тези типове данни, които се поддържат от неговата система от команди. На практика често възниква необходимост от преобразуване на данните от едно представяне в друго.

Данните, въвеждани от конзолата и извеждани към нея, се кодират от операционната система в съответствие с текущата таблица за кодиране. В ASCII таблицата всеки символ се кодира с един байт. Команди за аритметична обработка на числа в символен вид няма. Вижда се, че е необходимо преобразуване на символната информация във формат, поддържан от машинните команди. След такова преобразование се изпълняват необходимите изчисления и пак се преобразува резултата обратно в символен вид, за да може да се изобрази информацията на монитора.

По-рано разгледахме задача за преобразуване на шестнайсетично число (двуцифрено), въведено от конзолата в двоичен вид (в AL). След изпълнението на тази процедура полученото число може да се използва като операнд в двоични аритметични операции.

При въвеждане на десетично цяло число от клавиатурата, например: ___1207 <Enter> с помощта на функцията на MS DOS 09 в буфера за въвеждане от програмата ще бъде записан ред от символи:

‘ 1207’, CR

или последователност от байтове (ASCII кодове):

. . . 20h, 20h, 20h, 31h, 32h, 30h, 37h, 0Dh. . .

За запис на това число в регистър или в паметта за по-следващи изчисления е необходимо то да се преобразува от ASCII формат в двоичен код (ASCII to BIN). За целта е необходимо да се изпълнят следните операции:

- Да се намери първата значеща цифра, т. к. пред цифрите може да има интервали (Space, 20H) и знаци ‘+’ или ‘–’;
- Да се преобразуват всички ASCII кодове в неопакован десетичен формат:

. . . 20h, 20h, 20h, 01h, 02h, 00h, 07h, 0Dh. . .

За целта е необходимо от всеки код в диапазона 30h – 39h да се извади кода на цифрата ‘0’ (30h);

- В регистър AX, започвайки от старшата цифра, отляво надясно, да се добавя поредната стойност, умножена преди това със стойността, записана в акумулатора по 10.

AX=0000

1-ви байт 01 AX=AX*10= 0000
AX=AX+0001= 0001

2-ри байт 02 AX=AX*10= 0010
AX=AX+0002= 0012

3-ти байт 00 AX=AX*10= 0120
AX=AX+0000= 0120

4-ти байт 07 AX=AX*10= 1200
AX=AX+0007= 1207

5-ти байт 0D не е цифра →край

При обратното преобразование на двоичното число във ред от ASCII символи, очевидно, следва да се приложат всички операции в обратен ред:

- Да се смени знака на числото, ако то е отрицателно.
- Да се раздели числото на 10

AX = 1207
AX = AX / 10 AL ← 120 = 78h
AH ←7 = 07h

Командата за деление на дума на байт е **DIV R8.**

Делимото в AX, в резултатът от делението: частното в AL, остатъка в AH.

Командата за деление на двойна дума на дума: **DIV R16.**

Делимото е в DX:AX, в резултат от делението частното е в AX, остатъка в DX.

- Остатъкът (в AH или в AX) да се преобразува в ASCII код.

  AH =07h AH = AH+30h = 37h

- ASCII кодът да се запише в паметта (буфер за извеждане), придвижването трябва да бъде отдясно наляво.
- Ако частното в AX не е равно на нула, преход към т. 2. Иначе да се допише ‘–’ ако числото е било отрицателно.

Пример: Извеждане на екрана на числото, записано в регистър AX, в десетична бройна система. (Използва се делител байт.)

```
model small
stack 100h
.code
main:
        mov ax, 77h             ; използваме маркер за край
        push ax                 ; поставяме маркера в стека
        mov ax, 234d            ; числото за извеждане
        mov CL, 10d             ; делител на числото

labelDIV:
        mov ah, 00h
        div CL                  ; AX/CL –> частното в AL, остатъка в AH
        push AX                 ; запазване на остатъка в стека
        cmp AL, 0               ; проверка за нулево частно
        jne labelDIV

        mov ah, 02h

labelPRINT:
        pop DX
        cmp DX, 77h             ; проверка за край
        je ProgExit
        mov dl, dh
        add dl, 30h
        int 21h
        jmp labelPRINT

ProgExit:
        mov ax, 4c00h
        int 21h
end     main
```

Както при взаимното преобразуване между различните формати на вътрешното представяне на числата, така и при обмен с конзолата се реализира преобразуване на целочислените числа съобразно вътрешното им представяне в компютъра. Това важи и за числата с плаваща запетая.

Тема 11. Команди за обработка на низ от знаци

Оптималната текстообработка изисква елегантни функции за достъп и работа с низове от знаци, като:

- запомняне
- извикване на елементи от низа
- преместване
- претърсване на низове

За тези цели са предвидени следните специални инструкции с низове:

За запомняне на един елемент от низа се използва командата:

STOS/STOSB/ STOSW/ STOSD: записва стойността на акумулатора в низ. Низът се счита за приемник и трябва да бъде указан от ES:DI (даже при посочен операнд). При зареждане на всеки елемент от низа DI се коригира в зависимост от размера на операнда и стойността на флага за посока DF – DI се увеличава при DF=0 или се намалява при DF=1.

Ако се използва формата STOS, трябва да бъде посочен операнд-приемник, за да се укаже размерът на обработваните елементи. Смяна на сегмента по подразбиране не се допуска. При форматите STOSB (байтове), STOSW (думи), STOSD (двойни думи, само за i386+) инструкцията определя сама размера на елементите за обработка и дали елементът постъпва от AL, AX или EAX.

Инструкциите се използват обикновено с префикс REP за попълване на низ с една и съща стойност. Преди да започне изпълнението на повторимата инструкция CX трябва да съдържа броя на елементите за попълване.

За преместване на един елемент от низ в регистъра AL, респ. AX:

LODS/LODSB/LODSW/LODSD: зарежда акумулатора с низ от паметта. DS:SI трябва да сочи към елемента-източник, даже при зададен операнд. При зареждането на всеки елемент от източника SI се коригира в зависимост от размера на операнда и стойността на флага за посока DF (увеличава се при DF=0 и се намалява при DF=1). Ако се използва инструкцията lods, трябва да се посочи операнд, за да се определи размерът на обработваните елементи. За операнда-източник може да се посочи явно използван сегментен регистър. При използване на lodsb (байтове), lodsw (думи) или lodsw (двойни думи, само за i386+), инструкцията сама определя размера на обработваните елементи и дали елементът ще бъде зареден в AL, AX или EAX.

Инструкциите не се използват с префикси за повторение, защото е безсмислено да се зареждат последователни стойности от паметта в регистър.

За преместване на един елемент от низ от една клетка на паметта в друга

MOVS/ MOVSB/ MOVSW/ MOVSD/: копира низ от една област на паметта в друга. DS:SI трябва да сочи към низа-източник, а ES:DI – към низа-приемник, даже ако са зададени операнди. При всеки копиран елемент SI и DI се коригират в зависимост от размера на операндите и стойността на флага за посока DF (увеличават се при DF=0 и се намаляват при DF=1).

Ако се използва инструкцията MOVS, трябва да се посочат операнди, за да се определи размерът на обработваните елементи. За операнда-източник може да се посочи явно използван сегментен регистър. При използване на MOVSB (байтове), MOVSW (думи) или MOVSD (двойни думи, само за i386+), инструкцията сама определя размера на обработваните елементи. Инструкциите се използват обикновено с префикси за повторение REP.

За сравняване на един елемент от низ със съдържанието на AL, респ. AX:

SCAS/SCASB/SCASW/SCASD/: сканира низ, търсейки стойност, която е зададена в акумулаторния регистър. Сканираният низ се счита за приемник. ES:DI трябва да сочи този низ, даже при посочен операнд. Всеки елемент от низа се изважда от стойността на акумулатора и в съответствие с резултата се обновяват флаговете (без съхраняване на резултата). DI се коригира в зависимост от размера на операндите и стойността на флага DF (DI се увеличава при DF=0 или се намалява при

DF=1).

Ако се използва формата SCAS, трябва да се посочи операнд за определяне на размера на обработваните елементи. Не се допуска смяна на сегмента по подразбиране. При формите SCASB (байтове), SCASW (думи) и SCAD (двойни думи, само за i386+) инструкцията сама определя размера на обработваните елементи и дали търсеният елемент се намира в AL, AX или EAX.

Инструкциите се използват обикновено с префиксите за повторение. REPNE (или REPNZ) се използва за намиране на първия елемент в низ, който отговаря на стойността в акумулаторния регистър, а REPE (или REPZ) – за откриване на първото несъответствие. Преди сканирането в CX трябва да се запише максималният брой елементи за сканиране. След REPNE SCAS флагът ZF се нулира, ако низът не съдържа търсената стойност, а след REPE SCAS флагът ZF се установява в 1, ако низът съдържа само търсената стойност.

След завършване на инструкцията ES:DI сочи към елемента след (при DF=0) или преди (при DF=1) съответствието (или несъответствието). Ако CX се намали до 0, ES:DI сочи към елемента след или преди последното сравнение.

Флагът ZF се управлява от резултата на последното сравнение, а не от стойността на CX.

За поелементно сравняване на два низа се използва инструкцията:

CMPS/CMPSB/CMPSW/CMPSD: сравнява два низа. DS:SI трябва да сочи към низа-източник, а ES:DI – към низа-приемник, даже ако са зададени операнди. При всяко сравняване елементът-приемник се изважда от елемента-източник и се установяват флаговете, без да се съхранява резултатът. SI и DI се коригират в зависимост от размера на операндите и стойността на флага за посока DF (увеличават се при DF=0 и се намаляват при DF=1). Ако се използва инструкцията CMPS, трябва да се посочат операнди, за да се определи размерът на обработваните елементи. За операнда-източник може да се посочи явно използван сегментен регистър. При използване на CMPSB (байтове), CMPSW (думи) или CMPSD (двойни думи, само за i386+) инструкцията сама определя размера на обработваните елементи.

Инструкциите се използват обикновено с префикси за повторение. REPNE

(или REPNZ) се използва за откриване на първото съответствие между два низа, а REPE (или REPZ) – за първото несъответствие. Преди сравнението CX трябва да съдържа максималния брой елементи за сравнение. След REPNE COMPS флагът ZF се нулира при липса на съответствие, а след REPE COMPS флагът се установява при съответствие.

Когато инструкцията завърши, ES:DI и DS:SI сочат към следващия елемент (при нулиран флаг DF) или предишен елемент (при установен флаг DF) на съответствието или несъответствието. Ако CX=0, ES:DI и DS:SI сочат към елемента след или преди последното сравнение. Флагът ZF се управлява от резултата на последното сравнение, а не от стойността на CX.

Следващите две команди са се появили за първи път в системата команди на процесора i386.

Те осигуряват много по-висока скорост на предаване на данни, сравнена с тази, която осигурява контролерът DMA (Direct Memory Access – директен достъп до паметта).

- Въвеждане на елемент от низ от входно-изходен порт

INS/INSB/INSW/INSD (само за i286+): получава низ от порт. Низът се счита за приемник и трябва да бъде указан от ES:DI (даже при посочен операнд). Входният порт се задава в DX. При получаването на всеки елемент от низа DI се коригира в зависимост от размера на операнда и стойността на флага за посока DF – DI се увеличава при нулиран флаг DF или се намалява при установен флаг DF.

Ако се използва формата INS, трябва да бъде посочен операнд-приемник, за да се укаже размерът на обработваните елементи, а DX трябва да се посочи като операнд-източник, съдържащ номера на порта. Смяна на сегмента по подразбиране не се допуска. При форматите INSB (байтове), INSW (думи), INSD (двойни думи, само за i386+) инструкцията определя сама размера на получаваните елементи от низа.

Инструкциите се използват обикновено с префикс REP. Преди да започне изпълнението на повторимата инструкция CX трябва да съдържа броя на елементите за получаване. В защитен режим се регистрира грешка от обща защита, когато текущото ниво на привилегии е по-голямо от стойността на флага IOPL.

- Извеждане на елемент от низ във входно-изходен порт

OUTS/OUTSB/OUTSW/OUTSD (само за i186+): изпраща низ към порт. Низът се счита за източник и трябва да бъде указан от DS:SI (даже при посочен операнд). Изходният порт се задава в DX. При изпращането на всеки елемент от низа SI се коригира в зависимост от размера на операнда и стойността на флага за посока DF (SI се увеличава при DF=0 или се намалява при DF=1).

Ако се използва формата OUTS, трябва да бъде посочен операнд-източник, за да се укаже размерът на изпращаните елементи, а DX трябва да се посочи като операнд-източник, съдържащ номера на порта. Смяна на сегмента по подразбиране се допуска. При форматите OUTSB (байтове), OUTSW (думи), OUTSD (двойни думи, само за i386+) инструкцията определя сама размера на изпращаните елементи от низа.

Инструкциите се използват обикновено с префикс REP. Преди да започне изпълнението на повторимата инструкция CX трябва да съдържа броя на елементите за изпращане. В защитен режим се регистрира грешка от обща защита, когато текущото ниво на привилегии е по-голямо от стойността на флага IOPL.

При тези инструкции операндите-източници се адресират чрез регистрите DS:SI, а операндите-цели – посредством ES:DI. Необходимият адрес (адреси) на областта за низа-източник и/или низа-цел винаги трябва да бъдат заредени предварително в тези регистри. Отделните операции върху низовете протичат по следния начин:

- Изпълнява се елементарната операция с низове. След това се променят съдържанията на SI и/или DI. Ако DF = 0, брои се възходящо, следователно низът се обработва в посока на нарастване на адресите на клетките от паметта, ако DF = 1, брои се обратно, низът се обработва следователно в посока на намаляване адресите на клетките от паметта.

Когато е с префикс за повторение като REP, операцията с низ се изпълнява повторно. При това броят на желаните повторения, т. е. дължината на низа от знаци, трябва да бъде записан предварително в CX. След всяка операция с низ съдържанието на CX намалява с единица. Цялата операция се повтаря дотогава, докато съдържанието на CX стане 0. Ако се приложи условен префикс за повтаряне като REPE/REPZ или REPNE/REPNZ с инструкцията SCAS или CMPS, се проверява едно допълнително условие: след проверка на нулевия флаг (ZF) повторението се прекъсва преждевременно, ако условието за повторение (равно, неравно и т. н.) не е вече изпълнено.

REP: повторя операции с низ толкова пъти, колкото е стойността в CX Отначало стойността на CX се сравнява с 0 и ако CX=0, изпълнението продължава със следващата инструкция. В противен случай CX се намалява с 1, изпълнява се инструкцията с низа и цикълът продължава. REP се използва с MOVS и STOS, а също така и с INS и OUTS за i186+. За всички процесори с изключение на i386+ комбинирането на префикс за повторение с явно задаване на сегментен регистър може да причини грешки при поява на прекъсване.

REPE/REPZ/REPNE/REPNZ: повторя операция с низ, докато посочено условие е вярно и стойността на CX>0. REPE и REPZ (синоними) се изпълняват многократно, ако флагът ZF= 1, а REPNE и REPNZ (също синоними) – докато флагът ZF=0. Префиксите за условно повторение могат да се използват само с инструкциите CMPS и SCAS, т. к. само те променят флага ZF. Преди изпълнението на инструкцията в CX трябва да се запише максимално допустимият брой повторения. Отначало стойността на CX се сравнява с 0 и ако CX=0, изпълнението продължава със следващата инструкция. В противен случай CX се намалява с 1, проверява се флагът ZF и в зависимост от стойността му се изпълнява инструкцията с низа, като цикълът продължава. За всички процесори с изключение на i386+ комбинирането на префикс за повторение с явно задаване на сегментен регистър може да причини грешки при поява на прекъсване.

Състоянието на флаг DF може да се управлява с командите (без операнди):

CLD: нулира флага за посока DF. Всички следващи операции над низове се изпълняват в посока нагоре (от ниските към високите адреси) чрез увеличаване на съответните индексни регистри.

STD: установява в 1 флага за посока DF. Всички следващи операции над низове се изпълняват в посока от високите към ниските адреси.

Пример: Фрагмент от програма, която извежда последователност от символи във входно-изходен порт с номер 378 (LPT1, Printer Port), съответстващ на принтера.

```
.data
        str_pech db “Текст за печат”
.code
…
        mov dx, 378h
        lea di, str_pech
        mov cx, 16
        rep outsb
…
```

Пример: Установяване на наличие или отсъствие на символ в зададен низ от знаци.

```
model  small
stack 256
.data
        fnd db 0ah, 0dh, 'find', '$'
        nochar db 0ah, 0dh, 'nochar', '$'
        string db 'find char', 0ah, 0dh, '$'
.code
        assume ds:@data, es:@data
main:
        mov ax, @data
        mov ds, ax
        mov es, ax
        mov ah, 09h
        lea dx, string
        int 21h
        mov al, 'a'
        cld
        lea di, string
        mov cx, 10
        repne scas string
        je found
failed:
        mov ah, 09h
        lea dx, nochar
        int 21h
        jmp exit
found:
        mov ah, 09h
        lea dx, fnd
        int 21h
exit:
        mov ax, 4c00h
        int 21h
end     main
```

Пример: Сравнение на два стринга.

```
model  small
stack 256
.data
        match db 0ah, 0dh, 'match', '$'
        failed db 0ah, 0dh, 'failed', '$'
        string1 db '0123456789', 0ah, 0dh, '$'
        string2 db '0123406789', '$'
.code
        assume ds:@data, es:@data
```

```
main:
        mov ax, @data
        mov ds, ax
        mov es, ax
        mov ah, 09h
        lea dx, string1
        int 21h
        lea dx, string2
        int 21h
        cld
        lea si, string1
        lea di, string2
        mov cx, 10
cycl:
        repe cmps string1, string2
        jcxz equal                          ;cx=0
        jne not_match
equal:
        mov ah, 09h
        lea dx, match
        int 21h
        jmp exit
not_match:
        mov ah, 09h
        lea dx, failed
        int 21h
        dec si
        dec di
        ;inc si
        ;inc di
        ;jmp cycl
exit:
        mov ax, 4c00h
        int 21h
end     main
```

Следващият пример копира един стринг в друг до първото срещане на интервал:

```
model small
stack 256
.data
        str1 db 'Това е някакъв стринг'
        len_str1 = $–str1
        str2 db len_str1 dup (' ')
.code
        assume ds:@data, es:@data
main:
        mov ax, @data
```

```
        mov ds, ax
        mov es, ax
        cld
        mov cx, len_str1
        lea si, str1            ;зареждане на ефективните адреси
        lea di, str2            ;зареждане на ефективните адреси
m1: lodsb
        cmp al, ' '
        jc exit                 ;изход, при интервал
        stosb
        loop m1
exit:
        mov ax, 4c00h
        int 21h
end     main
```

Тема 12. Видове адресиране

Начинът, по който един компютър може да се обръща към множеството съществуващи адреси в паметта, за да записва, респ. чете, данните в и от паметта, се нарича вид на адресирането. Понеже дотук работихме с данни, вече използвахме някои от видовете адресиране, без специално да ги назовем. Когато процесорът се обръща към своите вътрешни регистри, става въпрос за адресиране чрез регистри. При асемблер като адрес се задава означението на регистъра, например MOV AX, BX. Ако непосредствено след инструкцията се задава самата дума, се говори за непосредствено адресиране – понякога то се нарича още инструкция с константа, понеже операндът е една константа, която може да бъде променена само чрез промяна на програмата. Например при инструкцията MOV CL, 125 има адресиране чрез регистър и непосредствено адресиране.

Третият вид адресиране е директно адресиране. При него след операционната част следва адресът на съответната клетка от паметта. Понеже при използване на асемблер не ни трябва непременно абсолютен адрес, тук можем да дадем символични адреси, които асемблер превежда в истински адреси на паметта, например MOV AL, *променлива_1*. Тези три вида адресиране трябва да има всеки процесор. Съгласно твърдение на Intel процесорите разполагат с много повече видове адресиране, ако се броят всички комбинации. Този голям брой видове се получава само чрез множество възможни комбинации на участващите в адресирането регистри. Многото на брой комбинации могат да се сведат до няколко основни вида адресиране.

12.1. Директно адресиране

Физическият адрес се състои от два компонента: базовият адрес, който се намира в сегментен регистър и отместването (Offset). Отместването на една променлива, към която трябва да се обърнем, се нарича още ефективен адрес. То представлява отместването на променливата в байтове от началото на сегмента, в който тази променлива е дефинирана.

При директното адресиране след операционната част следва ефективният адрес. Ако става въпрос за обръщение към данни, по стандарт се използва съдържанието на сегментния регистър за данни като базов адрес. Понеже асемблер не може да разпознае под какво име се крие сегментът за данни, за да може да разполагаме напълно свободно с името на сегмента, с директивата ASSUME съобщаваме на асемблера какво име носи сегментът за данни. При обръщение към клетки от паметта не искаме да работим с абсолютни адреси и за това трябва да съобщим на асемблера избраните от нас символични адреси (имена на променливи) и едновременно да резервираме необходимите за тях клетки в паметта. При превода асемблер установява отместването на всяка отделна променлива спрямо началото на сегмента, в който тя е била дефинирана. При асемблирането избраните от нас символични адреси се заместват с ефективни адреси.

Някои програмисти не използват отделен сегмент за данни. Те дефинират своите променливи в кодовия сегмент. Това е по принцип възможно, обаче за целта в директивата ASSUME на регистъра за сегмента за данни трябва да се присвоят използваните имена на кодовия сегмент. Ако е установен отделен сегмент за данни и неговото име е съобщено на асемблера посредством директивата ASSUME и евентуално трябва да се дефинират променливи в кодовия сегмент, асемблер най-напред запомня отместването на новите променливи спрямо началото на кодовия сегмент и поставя на инструкцията, която се обръща към променливата, един еднобайтов префикс. По време на изпълнение на програмата префиксът съобщава, че за разлика от стандартния начин на обръщение при образуване на физическия адрес, трябва да се използва кодовия сегмент като базов адрес. Ако не е използвана директивата ASSUME, би трябвало при всяка инструкция, която се обръща към паметта, да се даде кой базов адрес трябва да бъде използван при образуване на адреса. Програмистът обаче може да промени стандартното присвояване на сегментния регистър при изчисляването на адреса, при което сам задава префикс, например:

MOV ES: ***променлива1*****, AL**
MOV ***код_сегмент*****:** ***променлива2*****, DL**

12.2. Индиректно адресиране

Ако искаме да запомним по-голямо количество данни, директното адресиране се оказва непрактично. Тъй като при този вид адресиране става директно съпоставяне между променлива и принадлежащия и байт или дума в паметта, то би трябвало например при хиляда данни за прочитане да се използват хиляда инструкции MOV със съответното означаване на променливата. В обработката на данни такава поредица от данни се нарича още "списък" или едномерен масив.

Наред с такава поредица се срещат и по-сложни структури от данни. Ако данните са подредени в редове и колони, се говори за "таблица" или двумерен масив. Според вида на предварително зададената структура на данните се предлагат различни варианти на индиректно образуване на адрес.

Общо за индиректното адресиране може да се каже, че адресът не се доставя с инструкцията, а трябва да бъде получен по непряк път.

12.2.1. Адресиране посредством регистър

При този вид индиректно адресиране адресът на съответната клетка от паметта се намира в един спомагателен регистър. В инструкцията се оказва само този спомагателен регистър. При 8088/80x86 спомагателните регистри могат да бъдат BX, SI и DI, а от 80386 също и регистрите с общо предназначение EAX, EBX, ECX, EDX, ESI и EDI. За индиректно обръщение към стека се използва регистър BP, респ. от 80386 също така EBP. Като базов регистър по стандарт се използва регистърът DS, а при BP, респ. EBP – регистърът SS.

Инструкцията за прехвърляне на данни с помощта на регистровото адресиране има следния формат:

MOV [BX], r **респ.**
MOV r, [BX] **или**
MOV [BX], ***константа***

[BX] трябва да се чете "съдържание на BX", а инструкцията MOV [BX], r да се интерпретира така: прехвърли съдържанието на регистър r в клетка от паметта, чиито адрес е в BX.

Ако се използва инструкцията MOV [BX], константа – т. е. иска се да се

прехвърли константа в клетка от паметта – асемблер не знае дали се иска константата да се запише в байт или в една дума, например дали се иска числото 7 да се запомни като 07 или като 0007. Този проблем възниква при всички видове индиректно адресиране.

За да се постигне еднозначност, трябва да се използва оператор указател:

***Тип* PTR *операнд* (*тип* = Near, Far, Byte, Word, …)**

например:

MOV WORD PTR SI, KONST

Операторът указател също може да се използва за предефиниране на променлива или етикет. Тогава можем да се обръщаме към всеки отделен байт на една променлива-дума или да преобразуваме един близък преход (етикет NEAR) в далечен преход (етикет FAR):

MOV BYTE PTR *променлива_дума*, AL
MOV FAR PTR *етикет_Near*

За да представим нагледно действието на това прехвърляне на данни, извеждаме за показване на екрана съответната област на паметта. Понеже искаме да прилагаме това показване при всички видове адресиране, написваме я с помощта на разглежданата вече инструкция за низове LODSB като отделен макрос.

12.2.2. Адресиране спрямо база (базово адресиране)

При този вид индиректно адресиране се използва, както при регистровото адресиране, един спомагателен регистър, който се указва в инструкцията, но тук допълнително в инструкцията се задава още една константа. Спомагателният регистър при това съдържа началния адрес, наричан база, на един списък, а константата дава относителната позиция спрямо началото на списъка. Процесорът намира адреса на съответната клетка от паметта, като образува сумата от началния адрес (съдържанието на спомагателния регистър) и относителната позиция.

Инструкцията за обръщение към паметта с помощта на адресиране спрямо база има следния формат:

MOV [BX + *константа*], r или
MOV [BX] *константа*, r или

MOV *константа*[BX], r

Квадратните скоби тук заместват знак +. Като спомагателни регистри се използват същите регистри както при адресиране чрез регистри.

Този вид адресиране се използва тогава, когато в различни списъци трябва да се обърнем към един и същи елемент спрямо началото на списъка, например ако в един файл за клиенти искаме да селектираме според пощенския код, който в комплекта данни за клиента стои винаги на една и съща относителна позиция.

12.2.3. Индексно адресиране

При този вид адресиране, както при адресирането спрямо база, се използват един спомагателен регистър и една константа. Константата тук обаче представлява началния адрес на един списък (т. е. отместването спрямо началото на сегмента за данни), а спомагателният регистър съдържа индекса на търсения елемент от списъка (т. е. относителната позиция спрямо началото на списъка).

Процесорът намира адреса на клетката от паметта, към която иска да се обърне, като образува сумата от началния адрес на списъка и индекса. Инструкцията за обръщение към паметта чрез индексно адресиране има следния формат:

MOV [BX + *константа*], r **или**
MOV [BX] *константа*, r **или**
MOV *константа*[BX], r

Спомагателният регистър е същият регистър, за който става дума при адресирането чрез регистър.

Този вид адресиране се използва, ако имаме списък, който не искаме да обработваме последователно, а чрез обръщение към кой да е елемент.

Чрез индексното адресиране пресмятането на адреса се спестява на програмиста. Ако запомним относителната позиция на разглеждания елемент от списъка например в регистъра BX и го използваме като константа на отместването от началото на списъка, можем да получим достъп до елемента [BX] на таблицата чрез инструкцията MOV AL, списък[BX]. По-горе описаната форма за пресмятане на адреса беше [BX]+константа, при което константата беше отместването на началото на таблицата. При инструкцията MOV отместването се поставя автоматично от асемблера на мястото на

списък.

12.2.4. Базово-индексно адресиране

При този вид адресиране се използват два спомагателни регистъра. Процесорът намира адреса на търсената клетка в паметта, като образува сумата от съдържанието на двата спомагателни регистъра. Като спомагателни регистри може да се прилагат комбинациите BX+SI, BX+DI и BP+SI, BP+DI, последните две в стековия сегмент за обръщение към стека. Инструкцията за обръщение към паметта с помощта на базово-индексно адресиране има следния формат:

MOV [BX+SI], r **или**
MOV [BX][SI], r

Този вид адресиране се прилага винаги когато точният адрес на структурата от данни, към която трябва да се обърнем, не е установен твърдо по време на създаване на програмата.

12.2.5. Базово-индексно адресиране с отместване

При този най-сложен вид адресиране на процесорите на Intel наред с двата спомагателни регистъра се използва и една константа. Процесорът намира адреса на съответната клетка от паметта, като прибавя към сумата от съдържанията на двата регистъра и една константа – отместване. Като спомагателни регистри се използват комбинациите: BX+SI+константа, BX+DI+константа или BP+SI+константа, BP+DI+константа – последната със стековия сегмент за обръщение към стека. Инструкцията за обръщение към паметта с помощта на базово-индексно адресиране с отместване има следния формат:

MOV [BX+SI+*константа*], r **или**
MOV [BX+SI]*константа*, r **или**
MOV *константа*[BX+SI], r **или**
MOV *константа*[BX][SI]

Тук също скобите заместват знака “+”. Този най-неикономичен вид адресиране позволява директно обръщение към данни в двумерни масиви, т. е. матрици, респ. таблици.

12.2.6. Изводи

Всяка от трите части на операнда в паметта не е задължителна, макар че е очевиден фактът, че сте длъжни да използвате поне един от трите елемента, защото иначе няма да получите адрес в паметта. Ето как елементите на операнда в паметта изглеждат в друг формат:

BX		**SI**		**offset**
или	**+**	**или**	**+**	**Отместване**
BP		**DI**		
(база)		**(индекс)**		

Така се получават 16 съществуващи начини за задаване на адреса в паметта:

[offset]	**[bp+offset]**
[bx]	**[bx+offset]**
[si]	**[si+offset]**
[di]	**[di+offset]**
[bx+si]	**[bx+si+offset]**
[bx+di]	**[bx+di+offset]**
[bp+si]	**[bp+si+offset]**
[bp+di]	**[bp+di+offset]**

Тук отместването може да се сведе до 16-битова постоянна стойност.

От този списък се вижда, че всички режими на адресация се получават само от няколко елемента, комбинирани по различен начин.

Пример: Демонстрация на инициализация на масив.

```
model small
stack 256
.data
        mes db 0ah, 0dh, 'Масив – ', '$'
        mas db 10 dup (?)       ;масив
        i db 0
.code
main:
        mov ax, @data
        mov ds, ax
        xor ax, ax              ;зануляване на ax
        mov cx, 10              ;стойността на брояча на цикъла в cx
        mov si, 0               ;индекса на началния елемент в si
go:                             ;цикъл за инициализация
        mov bh, i               ;i в bh
```

```
        mov mas[si], bh         ;запис в масива на i
        inc i                   ;инкремент i
        inc si                  ;преход към следващия елемент на масива
        loop go                 ;да се повтори цикъла

        ;извеждане на екрана на получения масив
        mov cx, 10
        mov si, 0
        mov ah, 09h
        lea dx, mes
        int 21h
show:
        mov ah, 02h             ;функция за извеждане стойността на al на екрана
        mov dl, mas[si]
        add dl, 30h             ;преобразуване на число в символ
        int 21h
        inc si
        loop show
exit:
        mov ax, 4c00h           ;стандартен изход
        int 21h
end     main                    ;край на програмата
```

Тема 13. Процедури и стекови операции

Такава програма, която се превежда и сезапомня отделно, след което се извиква от една главна програма, се нарича подпрограма, но се използва често и наименованието процедура.

На края на всяка подпрограма стои инструкция за обратен преход RET (Return), която връща обратно към програмата, която я е извикала. За да може при това следващата по старшинство подпрограма да продължи отново коректно своята работа, трябва предварително да се осигури адреса на обратния преход. Затова този адрес трябва да се постави в стека и при това веднага при разклоняването в която и да е подпрограма.

Извикването на подпрограма става с инструкцията CALL и името на подпрограмата, респ. етикет за преход, който указва началото на програмата. На ниво машинна програма (обектен код) след CALL стои, разбира се, адресът на клетката в паметта, където започва подпрограмата. Тъй като съответният адрес за обратен преход е спасен в стека, то е все едно в кое място на главната програма ще се извика една подпрограма.

Така натрупаните задачи в стека се обработват, обаче в обратна последователност, което значи, че задачата която последна е постъпила в стека, се обработва първа.

Когато адреси се въвеждат в стека, респ. се вземат от стека, съдържанието на стека трябва да бъде намалено, респ. увеличено. Не се налага потребителят да се занимава с тези неща, тъй като инструкциите CALL и RET автоматично управляват указателя на стека.

13.1. Процедури

13.1.1. Стек за адреси и стек за данни

Без проблеми е да се използва един комбиниран стек за адреси и данни.

При работа с комбиниран стек трябва да се спазват следните правила:

- Съдържанията на регистрите, които се спасяват в стека, трябва да бъдат прочитани от стека в обратен ред; това значи, че регистърът, който се спасява последен, трябва да бъде прочетен първи.
- За да се предотврати погрешно интерпретиране на данни (съдържание на регистър) като адрес, данните трябва да бъдат съхранявани в стека и след това прочитани от стека по един от следните начини: данните трябва да бъдат съхранени при започването на подпрограмата, а преди инструкцията RET (обратен преход в главната програма) отново да бъдат извикани (подход а), или те трябва да бъдат доставени в стека в главната програма и следвръщане в главната програма отново да бъдат прочетени от стека (подход б) .

Ако се направи опит съдържанието на регистър, което се доставя в стека при протичане на подпрограмата, да бъде прочетено отново едва във включващата се след това главна програма, инструкцията RET в края на подпрограмата би интерпретирала намиращите се на първо място в стека данни като адрес и би извършила разклонение по този адрес, което води до напълно неконтролирано по-нататъшно протичане на програмата и най-често до “срив”.

Коя от двете възможности е по-добра, може да се спори, защото и двете възможности имат предимства и недостатъци.

За а) Предимство: при написването на главната програма и свързването на подпрограми, които са запомнени върху твърд носител с помощта на библиотечната програма LIB (Library) на Microsoft, не е необходимо да се мисли кои регистри на процесоратрябва да се използват, защото след

връщането от подпрограмата цялата информация стои вътре в регистрите в старата си форма.

Недостатък: Тъй като често не всички регистри, от които се нуждае подпрограмата, се обхващат и от главната програма, съответните инструкции за съхранение в подпрограмата са излишни. Тъй като тези стекови операции се нуждаят от време, то подход а) не е подходящ за програми, при които се цели висока скорост на обработката на информация.

За б) Предимство: Този начин за осигуряване на данните прави възможна висока скорост на обработка на информация.

Недостатък: При програмирането трябва да имаме постоянно под ръка списък, в който заедно с имената и функциите на подпрограмите са представени също така и използваните регистри.

13.1.2. Извикване на подпрограми (процедури)

Има различни варианти за извикване на подпрограми с инструкцията CALL и съответни варианти за връщане в главната програма с инструкцията RET: един близък (NEAR) тип, при който подпрограмата се намира в същия сегмент, в който се намира главната програма, и един далечен (FAR) тип, при който главната програма и подпрограмата се намират в различни сегменти. При близкия тип в стека се спасява само отместването (offset) на следващата инструкция от главната програма (съдържанието на IP), а при далечния тип – съдържанието на кодовия сегмент и отместването.

За да се знае кои варианти с CALL искаме да приложим, за означаване на началото и края на процедура трябва да използваме двойката директиви:

PROC . . . ENDP

Допълнително трябва да следва още означението NEAR / FAR. Ако не е зададено нищо, възприема се автоматично тип NEAR. Следователно една подпрограма (процедура) има следната структура:

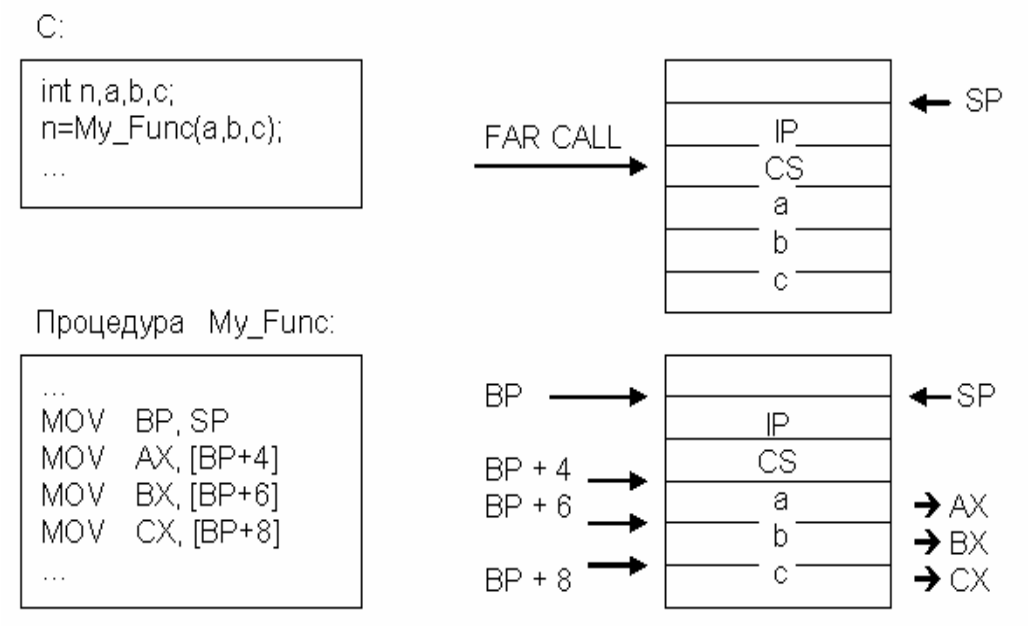
```
име_на_процедура    PROC NEAR (FAR)
                    Инструкция_1
                    . . .
                    Инструкция_n
                    RET
име_на_процедура    ENDP
```

По премълчаване в BX се зарежда базовото отместване относно DS, в BP – базовото отместено относно SS (в стека). Както беше вече споменато регистър BP се използва за организация на произволен достъп в стека.

При извикване на подпрограма с командата CALL автоматично за записва в стека адреса за връщане –2 байта със стойността на IP и се намалява стойността на указателя SP с 2 (SP = SP – 2). Ако процедурата е от тип FAR, в стека първо се записва CS, после IP (SP= SP – 4).

Предаваните в подпрограмата данни могат да се намират в регистрите с общо предназначение. Обаче броят на предаваните параметри може да превишава броя на свободните регистри. Освен това функциите и процедурите в езиците от високо ниво използват друг начин на предаване – чрез стека, записвайки параметрите в определен ред.

"Отгоре" в стека тези параметри са "прикрити" с адреса за връщане и подпрограмата трябва да ги прочете без да ги извлича от стека (без да променя стойността на SP). Това може да стане с помощта на стековия базов регистър BP. (Отместването, записано в BP, по премълчаване се отчита от адреса в SS).



Фигура 25. Извикване на подпрограма

Стойността на регистъра SP не може да се ползва в командите за задаване адрес на данни, записани в стека, за целта можем да използваме единствено BP.

Достъпът до аргументите в стека става с използване на отместването, относно съдържимото на BP:

```
MOV  AX, [BP+4]    ;да се прехвърли стойността на a в AX
MOV  BX, [BP+6]    ;да се прехвърли стойността на b в BX
MOV  CX, [BP+8]    ;да се прехвърли стойността на c в CX
```

Ако в стека са записани 4 или 8-байтови параметри, те следва да се извличат "на порции" в регистрови двойки (например DX:AX).

Ако типа на подпрограмата е NEAR, на върха на стека е записана само стойността на IP за връщане и последният записан в стека параметър ще има адрес [BP+2], а не [BP+4].

При връщане от подпрограмата с командата RET от върха на стека се зарежда в IP (и възможно в CS) адреса за връщане. При това SP=SP+2 или SP=SP+4. Стекът може допълнително да се очисти от параметрите с командата RET n. При това стойността на указателя SP допълнително се увеличива с n.

Следващият пример извежда на екрана текст "Escape" и осигурява излизане от програмата при натискане на клавиша "Escape".

```
sseg        segment para stack 'stack'
            dw 32 dup ('s') ;64 байта размер на стека+инициализацията му
sseg        ends

dseg        segment  para 'data'
            str1 db 20h, 71h, 'e', 0f4h, 's', 71h, 'c', 71h, 'a', 71h
            str2 db 'p', 71h, 'e', 71h, '!', 74h, 20h, 71h
dseg        ends

cseg        segment para 'code'
main        proc far
            assume cs:cseg, ds:dseg, ss:sseg, es:dseg
            push  ds
            xor ax, ax
            push ax         ;инициализация на стека
            mov ax, dseg
            mov ds, ax
            mov es, ax      ;инициализация на ригистрите ds и es
```

```
                mov al, 03h     ;видео режим № 3 (80x25)
                call initscr
                mov dh, 0ah     ;ред 10
                mov dl, 23h     ;стълб 35 (позиция на курсора)
                call outword
                mov bl, 1bh     ;ASCII код на <esc> (=27 десетично)
                call waitkbd
                ret

main            endp

initscr         proc            ;установяване на видео режим №3
                                ; (+изчистване на екрана)
                mov ah, 00h
                int 10h
                ret
initscr         endp

outword         proc            ;извеждане на екрана на низ от символи с
атрибути
                mov ax, 1303h
                lea bp, str1    ;str1->es:bp зарежда началния адрес
                mov cx, 09h     ;задава броя на извежданите символи
                int 10h
                ret
outword         endp

waitkbd         proc            ;проверка за натискане на <esc>
ckl:            mov ah, 0
                int 16h         ;очаква натискане на клавиш от клавиатурата
                cmp al, bl      ;cmp ? да се сравни; bl = 27
                jne ckl         ;jne ? преход, ако не е равно
                ret             ;програмата приключва при натискане на <esc>
waitkbd         endp
cseg            ends
end             main            ;входна точка е main
```

13.1.3. Подпрограма като процедура с далечен преход (FAR)

В модерното програмиране за намаляване на разходите се създават подпрограми, които се свързват в една библиотека и така са на разположение за удобно многократно използване в нови програми по-късно. Понеже при асемблирането на главната програма възниква едно име на процедура, което не е дефинирано в тази програма, асемблер ще изведе в нормалния случай съобщение за грешка, тъй като той не може да замести извикваната процедура с адрес. За решаването на този проблем ни е необходима една друга асемблер директива:

EXTRN ***име1*:*тип1*, *име2*:*тип2*, …**

(където тип е например NEAR, FAR, ABS, BYTE, WORD, . . .)

С тази външна директива се съобщава на асемблер, че дадените след EXTRN имена са били дефинирани в друг програмен модул, респ. представляват процедури от дадения тип или при тип ABS са константи, които са били указани с EQU или ‘=‘.

Етикетът FAR може да стои на произволно място. Най-често обаче е удобно той да се декларира в кодовия сегмент, в който ще бъде необходим, както е при етикета NEAR.

Неупотребеното тук отпращане към външни данни (тип Byte, Word и т. н.) с помощта на директивата EXTRN трябва да стане в сегмента за данни.

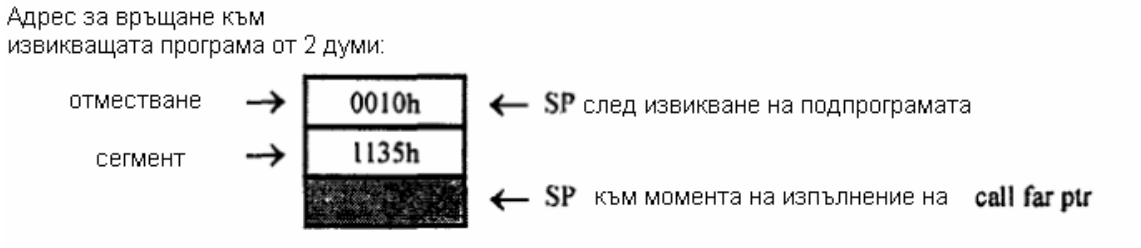
В нашия случай следователно трябва да се напише: EXTRN *име_на_процедура*: FAR. Така проблемът за откриването на адрес при инструкцията CALL *име_на_процедура* се прехвърля на свързващата програма, която трябва да свърже главната главната програма и отделната преведена процедура.

За да разпознае асемблер, че името NAME няма локално значение в програмата, а намира приложение в друга програма, предварително трябва да бъде поставената директивата

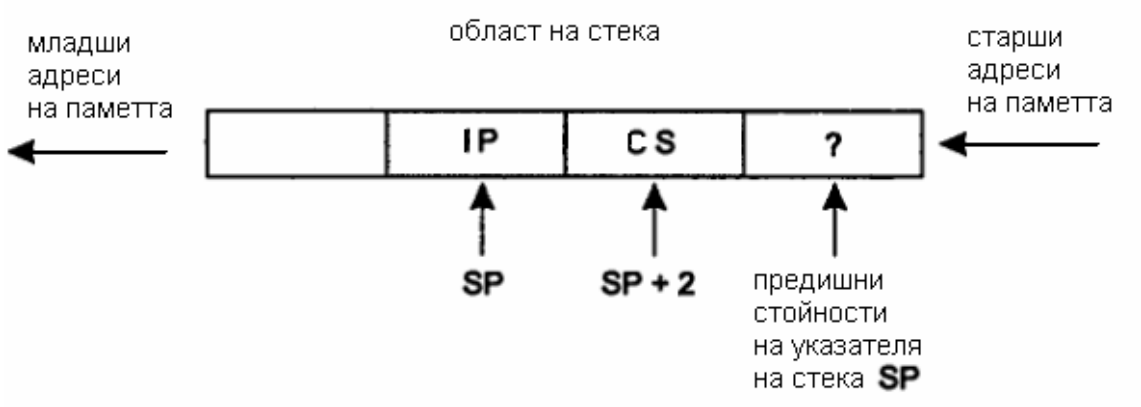
PUBLIC ***символ1*, *символ2*, . . .**

(където символ може да бъде етикет, име на процедура, променлива или константа, указана с EQU).

С това символът е деклариран като PUBLIC (= открит, публичен) и предаден заедно с неговия адрес на обектния файл, което нормално не става, понеже след асемблирането имената на променливите и означенията на етикетите повече не съществуват. Така този символ е достъпен за свързващата програма.



Фигура 26. Състояние на стека след влизане в далечна подпрограма:



Фигура 27. Съдържимото на стека след извикване на далечна процедура

13.2. Макроси

При разгледаните дотук програми ние повтаряхме еднакви последователности от инструкции във формата на функции на системното прекъсване 21H. Тъй като това повикване на функция ще бъде нужно много често, препоръчва се то да бъде дефинирано като макрос. Съпоставянето на една последователност от инструкции към едно дефинирано име се нарича дефиниране на макрос. Тази дефиниция трябва да постъпи преди макросът да бъде повикан за първи път. За дефинирането на макрос се нуждаем от директивата:

MACRO…. ENDM

С това се осъществява стандартната структура на един макрос:

Име **MACRO** ***променлива 1, променлива 2, . . .***

Инструкция 1
. . .
. . .
Инструкция n
ENDM

При ENDM – за разлика от случаите на сегменти и процедури – името не трябва да се повтаря още веднъж. Променливите са фиктивни, което значи, че те пазят места, които при извикване на макроса ще бъдат заменени с реални параметри.

Последователността от инструкции от инструкция 1 до инструкция n се нарича тяло на макроса.

Макросът за разлика от една процедура обаче не се асемблира. Следователно, за да може асемблер да обработи коректно, трябва преди първото извикване на един макрос да бъде прочетена библиотеката от макроси. Записани програми, респ. програмни части в мнемоничен код, може да се прочетат с директивата:

INCLUDE ***име_на_файл***

В резултат те ще се вмъкнат в оригиналната програма от мястото, на което се среща директивата INCLUDE. При изпълняване на макроси тази директива INCLUDE трябва да стои пред първото повикване на макрос. При всяко срещане на макрос асемблер претърсва наличната библиотека и поставя тялото на макроса в програмата на мястото на извикването на макроса.

Понеже макросите се включват в библиотека, при по-късното им използване не се знае съдържанието на кои регистри те унищожават, нито в кои регистри макросите получават, респ. връщат резултати.

Ако в регистрите ще се предават параметри на макроса или ще се връщат резултати от него, това трябва да се подскаже в името на макроса, за да може при програмирането регистрите да се използват подходящо. Съдържанието на използваните регистри от макроса трябва предварително да се съхрани, а старото съдържание трябва отново да се зареди в тези регистри – изключение са регистрите, в които се връщат резултати от макрос.

Нека да имаме макрос, който съдържа цикли и следователно етикети за преход. Както стана ясно по-горе, тялото на макроса се вмъква от асемблера в извикващата програма. Ако сега този макрос се повика повторно в една програма или ако в извикващата програма са използвани същите етикети като

неговите, то асемблер изготвя съобщение за грешка, тъй като поради многозначността на етикета той не може да образува коректен адрес за преход. За да се избегне това, променливите за преход трябва да се дефинират като локални, т. е. като валидни само за макроса променливи. Това става с помощта на директивата

LOCAL *параметър1*, *параметър2*, …

която трябва да стои непосредствено след заглавния ред на макроса.

Ако сега асемблер срещне една такава променлива, дефинирана като локална, той я замества с един символ във формат ??n, където n е едно четиризначно число. При повечето етикети следователно: ??0000, ??0001 и т. н.

Въвеждането на параметър е фиктивно, което значи, че той стои като заместител за един от използваните параметри в извикващата програма. Ако при дефиницията на макроса използваме повече такива фиктивни параметри, вместо тях асемблер ще постави истинските параметри в последователността, в която те стоят след инструкцията макрос в програмата. Освен константи и променливи параметрите могат да бъдат също съдържание на регистри.

Подпрограмите можем да извикваме произволно често. Техният код от инструкции се намира обаче само един път в паметта. Макросите пък водят към разширяване на програмата, понеже при всяко повикване в нея се поставя пълния код на макроса. Това може да раздуе многократно дължината на програмата. Следователно от гледна точка на икономия на място в паметта подпрограмите са за предпочитане От гледна точка на скоростта на обработка обаче за предпочитане са макросите, тъй като необходимите при подпрограмите инструкции CALL и RET (във връзка със стековите операции) изискват допълнително време.

Пример: Изчисляване на функцията 8! (осем факториел) рекурсивно.

```
main        proc
            mov ax, 8
            push ax
            call Factorial
            mov ax, 4c00h
            int 21h
main        endp

Factorial   proc
            push bp
            mov bp, sp
            mov ax, [bp+4]
            cmp ax, 1
```

```
                ja L1
                mov ax, 1
                jmp L2
L1:             dec ax
                push ax
                call Factorial          ; Factorial(n–1)
                mov bx, [bp+4]          ; зарежда n
                mul bx                  ; ax=ax*bx
L2:             pop bp
                ret 2                   ; в ax е резултата
Factorial       endp
```

Типичният фрагмент на програма, съдържаща извикване на процедура с предаване на аргументи чрез стека, може да изглежда по следния начин:

Пример:

```
model small
. . .

proc_l  proc near               ;“близка” процедура (near) с n аргументи
        ;начало на пролога
        push bp
        mov bp, sp
        ;край на пролога
        mov ax, [bp+4]          ;достъп до аргумента arg_n за near процедурата
        mov ax, [bp+6]          ;достъп до аргумента arg_{n–l}
        . . .
        ;начало на епилога
        mov sp, bp              ;възстановяване на sp
        pop bp                  ;възстановяване стойността на старото bp
                                ;до входа в процедурата
        ret                     ;възврат в извикващата програма
        ;край на епилога
        proc_l endp

.code
main    proc
        mov ax, @data
        mov ds, ax
        push arg_l              ;запис в стека на 1-ия аргумент
        push arg_2              ;запис в стека на 2-ия аргумент
        . . .
        push arg_n              ;запис в стека на n–ия аргумент
        call proc_l             ;извикване на процедурата proc_1
        ;действия по очистване на стека след връщане от процедурата
        . . .
ml:
        _exit
        main endp
```

```
end     main
```

Кодът на пролога е от две команди. Първата push bp съхранява съдържимото на BP в стека. Втората команда настройва BP на върха на стека. За достъп до arg_n е достатъчно отместване от съдържимото на BP на 4, за arg_{n–l} – на 6 и т. н. Но тези отмествания са валидни единствено за процедури от тип near. За процедури от тип far тези стойности е необходимо да се коригират още с 2, т. к. при извикване на процедура от далечен тип в стека се записва пълния адрес – съдържимото на CS и IP. Затова за достъп до arg_n командата би изглеждала така: mov ax, [bp+6], а за arg_{n–l}, съответно, mov ax, [bp+8] и т. н. Краят на процедурата е длъжен да осигурява коректно връщане от процедурата. Кодът на епилога е длъжен да възстанови контекста на програмата в точката на извикване на процедурата от по-горното ниво на извикващата програма.

Следващият пример демонстрира предаване на аргументи в процедура с помощта на директивите PUBLIC и EXTRN. В сегмента с данни се обявяват две символни променливи и се извикват с процедура, която ги извежда на екрана. Двете програми се асемблират поотделно, като се обединяват с:

LINK Prog1+Prog2

Пример:

Prog1 ->

```
include mac. inc
extrn my_proc2:far, pr0:byte
public pr1, pr2

stk         segment stack
            db 256 dup (0)
stk         ends
data        segment para public 'data'
            pr1 db '1'
            pr2 db '2'
data        ends

code        segment

main        proc far
            assume cs:code, ds:data, ss:stk
            mov ax, data
            mov ds, ax
            mov dl, pr0
            OutChar
```

```
                call my_proc2
                Exit
main            endp
code            ends
end             main
```

Prog2 ->

```
include mac. inc
extrn pr1:byte, pr2:byte
public my_proc2, pr0

data            segment para public 'data'
                pr0 db '0'
data            ends

code            segment
my_proc2        proc far
                assume cs:code, ds:data
                mov dl, pr0
                OutChar
                mov dl, pr1
                OutChar
                mov dl, pr2
                OutChar
                mov dl, pr0
                OutChar
                ret
my_proc2        endp
code            ends
                end
```

Съдържимото на mac. inc файла е:

```
OutChar         macro
                push ax
                mov ah, 02h
                int 21h
                pop ax
                endm

Exit            macro
                mov ax, 4c00h
                int 21h
                endm
```

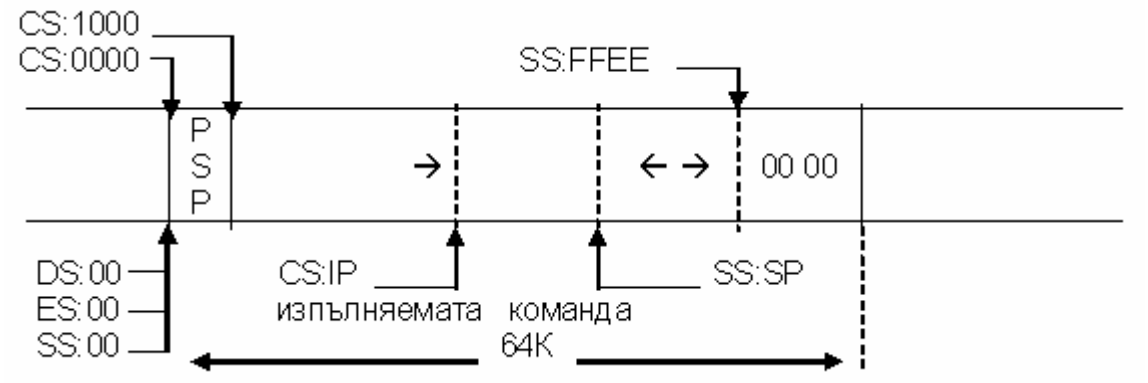
Тема 14. EXE и COM формат
14.1. Формат COM

Изпълнимите модули във формат *име*.COM съдържат образа на оперативната памет. При зареждане на такава програма в оперативната памет, тя не се подлага на реорганизация. Само пред кода се записва Префикса на Програмния Сегмент (PSP, 256 байта), съхраняващ информация, необходима за връщане управлението на MS DOS при приключване на програмата.

Програмата във формат COM се състои от един единствен сегмент за код, данни и стек. Затова при зареждането на програмата в сегментните регистри CS, DS, ES и SS се записва един и същи адрес.

В регистъра на отместването за командите IP се записва 100h. Така CS:00 сочи към PSP, а CS:IP – към първата команда след PSP. Отместването за данните в командите винаги се отчита от DS или от ES.

Пълният адрес е в регистровите двойки DS:AX / DS:DX / DS:SI/ ES:BX / ES:BP / ES:DI.



Фигура 28. Образ на паметта при COM програма

Стекът автоматично се формира на границата 64K. Указателят към върха на стека (SP = 00 00)автоматично се намалява с 2 единици:

$$\mathbf{SP = 00\ 00} \xrightarrow{\mathbf{-2}} \mathbf{SP = FFFE}$$

и в стека се записва нулевата дума 00 00.

При приключване на програмата една дума (00 00) с командата RET се извлича от стека в IP и CS:IP предава управлението на началото на PSP (CS:0000), където се съдържа командата INT 20H – прекъсването, обработващо приключването на програмата.

В програмата на асемблер за формат COM задължително трябва да присъстват следващите директиви (указания за компилатора):

```
име_сег     SEGMENT PARA
            ASSUMECS: име_сег, DS: име_сег, SS: име_сег
            ORG 100H
вх_точка:
            . . . . . .

име_сег     ENDS
            END вх_точка
```

Директивата ORG 100h установява в IP необходимата стойност 100h.

Входната точка сочи MS DOS мястото, откъдето започва изпълнението на програмата. Входната точка трябва да бъде вътре в сегмента на кода. Това е или етикет, или име на основната (главната) процедура.

Вътре в сегмента се разполагат процедурите (една или няколко). Едната от тях е главна и се изпълнява първа. Останалите процедури (освен главната) се извикват с командата

CALL ***име_на_процедура***

Възвратът RET от процедурата тип NEAR извлича от стека една дума (отместване) и го зарежда в IP (в CS адреса PSP).

Възвратът от процедурата типа FAR извлича от стека две думи за CS:IP (първо за IP, после за CS).

По подразбиране (премълчаване) за тип на процедурата се приема NEAR.

Следва кода на програмата Hello_World за формат COM.

```
.model tiny                    ; модел на паметта, използван за COM
.code                          ; начало на кодовия сегмент
        org 100h               ; начална стойност на брояча – 100h
```

```
start:  mov ah, 9                       ; номера на функцията на MS DOS – в AH
        mov dx, offset message          ; адреса на реда – в DX
        int 21h                         ; извикване на функция на MS DOS
        ret                             ; приключване на COM програмата
        message db 'Hello World!', 0Dh, 0Ah, '$'; реда за извеждане
        end start                       ; край на програмата
```

Данните в програмата с формат COM се препоръчва да се разполагат в началото на сегмента пред главната процедура. При това входна точка трябва да бъде етикета на командата JMP.

```
име_на_сегмента          SEGMENT PARA
                         ASSUME CS: име, DS: име , SS: име
                         ORG 100H
етикет_за_вход:
                         JMP име_на_главната_процедура
;-------------------------------------------------------------
име_поле_данни           DB списък_с_данни ; данни тип БАЙТ
. . . . .
име_поле_данни           DW списък_с_данни; данни тип ДУМА
;-------------------------------------------------------------
име_на_главната_процедура    PROC NEAR
. . . . .
                             RET
име_на_главната_процедура    ENDP
;-------------------------------------------------------------
;. . . други процедури
;-------------------------------------------------------------
име_на_сегмента              ENDS
                             END етикет_за_вход
```

Това изискване не е задължително – данните могат да се разположат и в края на програмата, след последната процедура пред ENDS. Обаче показаната по-горе структура на COM програма има предимството, че позволява в процеса на асемблирането да се откриват грешки, сочещи към несъществуващи данни.

Имената на сегмента, процедурите, полетата с данни (променливи) и етикетите са произволни, но са длъжни да съответстват на следното изискване: да се състоят от не повече от 8 латински букви и цифри и да започват само с буква. (Малките букви и главните букви не се различават.)

Етикетът за вход се разполага в началото на отбелязания ред с командата или на предния празен ред. След етикета се слага двоеточие. Обаче при сочене към етикет в командите за цикъл (LOOP) и за преход (JMP, JZ/JNZ и др.) двоеточие не се слага.

След точка със запетая могат да се пишат коментари в програмата .

14.2. Формат EXE

14.2.1. Въведение в EXE програми

Разликата на формат EXE от формат COM се определя от това, че EXE програмата се състои от няколко сегмента. Затова изпълнимите модули на тези програми (име.EXE) имат заглавие, (не по-малко от 512 байта), съдържащо информация за свързванията на сегментите с адресите на RAM след зареждането.

При зареждането на такава програма в паметта с произволен сегментен адрес (ADR) сегментните регистри за данни винаги сочат към този адрес (DS = ES = ADR), а стойностите за CS, IP, SS, SP се определят от Линкера при четене заглавието на файла.

От тук следват особеностите на оформлението на EXE програмите на езика асемблер:

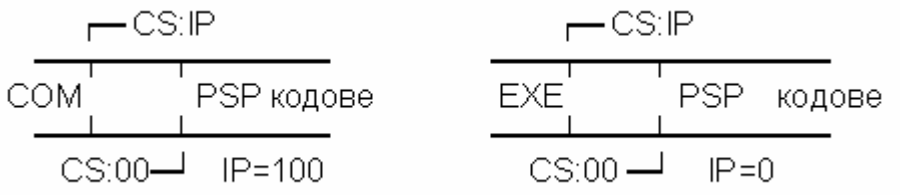
- Наличие на няколко сегменти на програмата с произволни имена.

В директивата ASSUME тези имена е необходимо да бъдат свързани със съответните сегментни регистри. Например:

ASSUME CS:CSEG, DS:DSEG, SS:SSEG, ES:NOTHING

Тук CSEG, DSEG и SSEG са произволни имена на програмни сегменти, NOTHING е ключова дума, означаваща отсъствие на сегмент с данни в програмата.

- Директивата ORG 100h, установяваща в IP=100h, в EXE програмите отсъства, понеже съдържимото на CS може да бъде друго.



Фигура 29. Отсъствие на директивата ORG в EXE програмите

- Сегмента на стека е задължително да бъде определен в програмата и неговата дължина също трябва да е зададена. Например, така:

```
SSEG  SEGMENT PARA STACK
      DW 32 DUP(?)
SSEG  ENDS
```

Дължината на стека тук е определена в 32 думи (DW), т. е. 64 байта, с неопределени стойности (DUP(?) –Duplicate, повтори). За облекчаване на свързването стека може да се запълни с нещо:

```
DB 40h DUP('S')
```

Посочването за обединение (STACK) е задължително. Иначе линкера ще разглежда този сегмент като обикновен сегмент с данни и ще изведе предупреждение "NoStack".

При зареждането на тази програма ще бъде установен SP =0000. Изпълнението на първата команда PUSH ще доведе до установяване върха на стека на границата 64K (SP–2 = FFFE).

- Понеже нито CS, нито SS не са свързани с началото на PSP, в стека е необходимо да се съхрани адреса на началото на PSP, където е записана командата INT 20 (за връщане в MS DOS с командата RET на главната процедура).

При зареждане на програмата за начало на PSP се посочват DS и ES. Затова стандартно начало на EXE програма са командите за инициализация на стека:

```
1E          PUSH DS
2B C0       SUB  AX, AX (или XOR AX, AX / MOV AX, 00)
50          PUSH AX
```

При изпълнение на RET в главната процедура (типа FAR) стойността ADR:0000 ще се извлича от стека и ще се запише в CS:IP.

- Едновременно със запълването на стека е необходимо да се инициализира регистър DS (и ако е необходимо ES) с името на сегмента с данните (в заглавието свързване на DS и ES няма).

B8 xx xx MOV AX, *име_на_сегмента_с_данни*

8E D8 MOV DS, AX
8E C0 MOV ES, AX

Ако в програмата директно или косвено се използват свързвания с регистър ES (например ES: BP или ES: BX, както изискват някои прекъсвания), регистърът ES също е необходимо да се инициализира.

- Ако за входна точка служи името на главната процедура, тя трябва да е от тип FAR. (По премълчаване е NEAR. При преход към тип FAR се предава 4-байтовия адрес [сегмент: отместване], като при извиквания и преходи от типа NEAR само 2 байта отместване вътре в сегмента.)

14.2.2. Зареждане на EXE файл

При зареждане на EXE програмите Линкерът изпълнява следните действия:

- избира в паметта сегментен адрес за зареждане (ADR);
- построява префикс на програмния сегмент (PSP) с дължина 256 байта (100h);
- записва сегментния адрес за зареждане в регистрите DS, ES;
- чете заглавието на EXE файла и според описанието в това заглавие, зарежда регистрите CS, IP, SS, SP

Като резултат управлението се предава на адрес CS:IP.

Съдържимото на регистрите ES и DS се изменя в хода на изпълнение на програмата. При изпълнение на големи програми (ако кодовете са повече от 64K) се презарежда и CS.

Заглавието на EXE файла се състои от следните части:

- 2-байтов дескриптор 4D 5A (‘MZ’) – характерен признак за всеки EXE файл;
- Описание на размера на паметта, необходим за програмата, дължина на самото заглавие и отместванията за регистрите CS, IP, SS, SP;
- Таблици за настройка (или таблици с разместванията), съдържащи настройваните адреси.

Броят на настройваните елементи може да се прочете в заглавието с отместване 06 от началото (TD). Ако TD=0 то това означава, че таблицата за настройка е празна и тя може да се изтрие (с EXE2BIN).

	‘MZ’		Размер				TD								SS	
000	4D	5A	00	00	06	00	01	00	20	00	00	00	FF	FF	98	00
010	80	00	00	00	00	00	00	00	22	00	00	00	01	00	FB	20
	SP				IP		CS									

Таблица 20. Заглавието на EXE файл

Двубайтовото поле SS има отместване 0Eh (14) от началото на Заглавието:

SP 10h (16)
IP 14h (20)
CS 16h (22)

Заглавието за сегментните регистри CS и SS съдържа отместванията относно края на PSP. Затова към стойността от заглавието Линкера добавя сегментния адрес за зареждане (ADR) и размера на PSP (10h). Стойностите в заглавието за IP и SP следва да се тълкуват като отмествания относно CS и SS съответно.

CS = 00 00 IP = 00 00 SS = 00 98 SP = 00 80

Следователно, стойностите на регистрите след зареждането ще бъдат:

CS = ADR + 10h (веднага след PSP) **CS:IP – 1-ия байт след PSP**

SS = ADR + 10 + 98 = ADR+A8 **SS:SP = [ADR+A8]:80**

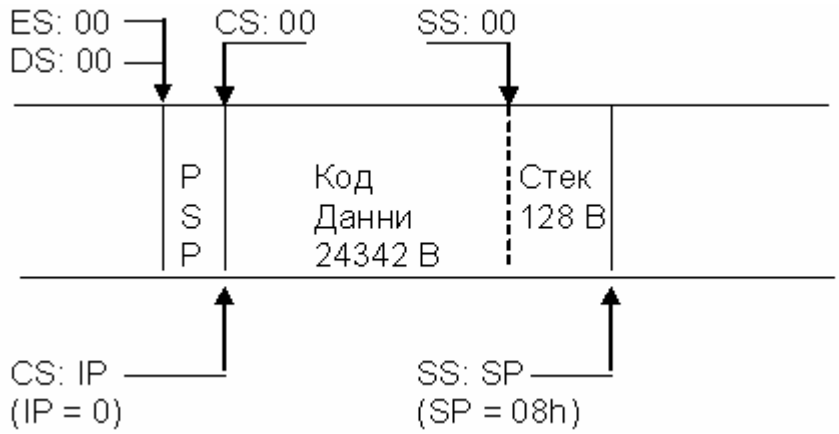
Получихме дължина на стека – 128 байта.

Разликата (SS–CS)*16 = 2432 байта дава размера на сегмента на кода заедно със сегмента данни. Размерите и взаимното разположение на сегментите зависят от избрания по време на компилацията модел на паметта.

14.2.3. Модели на паметта

Следват примери, демонстриращи различните модели.

- Модел SMALL



Фигура 30. Модел SMALL

Стойността на DS се определя в хода на изпълнение на програмата.

- Модел LARGE

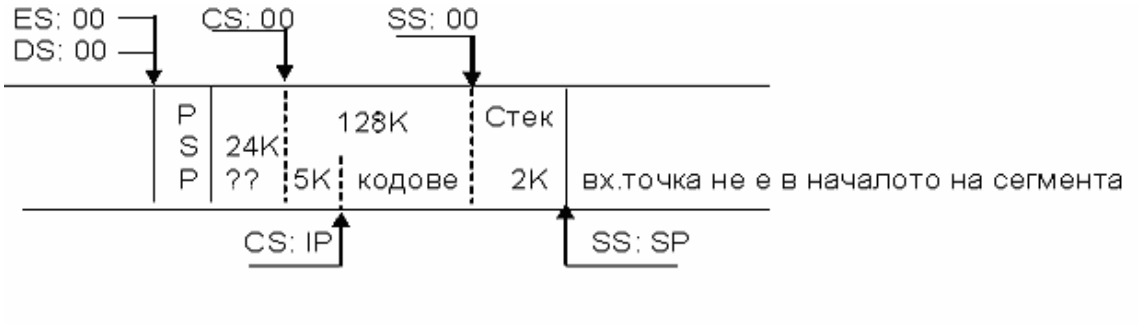
	'MZ'		Размер				TD								SS	
000	4D	5A	C0	00	44	01	8F	08	40	02	80	00	FF	FF	2C	26
010	00	08	9B	37	7A	15	15	06	1E	00	00	00	01	00	0B	00
	SP				IP		CS									

Таблица 21. Образ на паметта при модел LARGE

SP = 08 00h=2K – размер на стека IP = 15 7A=5K

SS = ADR+10+262C CS = ADR+10+0615

(SS – CS)*16 = 128 K



Фигура 31. Модел LARGE

Ще разгледаме заглавието на EXE файла, получен от програма на асемблер във формат COM (без опция /t). Този модул не работи (“зависва”).

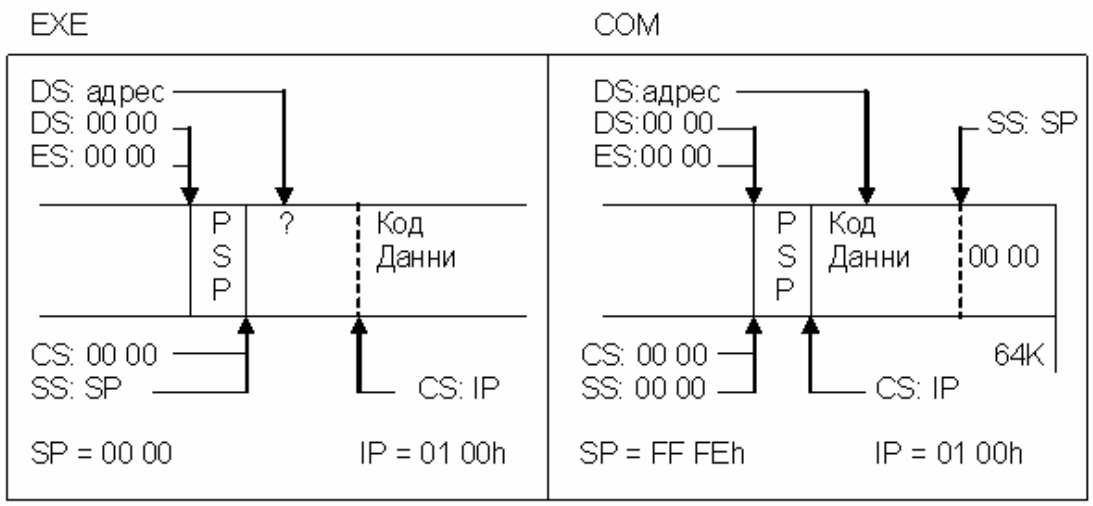
	‘MZ’		Размер				TD								SS	
000	4D	5A	42	01	02	00	00	00	20	00	00	00	FF	FF	00	00
010	00	00	00	00	00	01	00	00	3E	00	00	00	01	00	FB	30
	SP				IP		CS									

Таблица 22. Заглавие на EXE файла, получен от програма на асемблер във формат COM (без опция /t)

CS = ADR+10+0 CS:IP = [ADR+10]: 0100

SS = ADR+10+0 SS:SP = [ADR+10]: 0000

Таблицата за настройка е празна (TD = 0) и може да бъде изтрита с EXE2BIN. След което програма ще работи, понеже COM програмата се зарежда в паметта по друг начин.



Фигура 32. Сравнение на програми на асемблер във формат COM и EXE

14.3. Изводи

- EXE

- CS и SS сочат в КРАЯ на PSP;
- ORG 100H дава отместване на всички команди относно PSP;
- Стекът не е установен: SP=0 и адресът за връщане на PSP (DS:00) в него не е записан;
- Извличане на данните по отместванията в командите (DS: отместване) става неправилно.

- COM

- CS и SS сочат НАЧАЛОТО на PSP;
- ORG 100H установява указателя IP към 1-ия байт след PSP;
- Стекът автоматично се установява на границата 64K (SP = FF FE) и в него се записва 00 00;
- Извличане на данните (DS:отместване) става от областта с данни.

Тема 15. Работа с файлове

При работа с файлове трябва да бъдат изяснени предварително някои основни понятия:

- Име на файла (file name) – знакова поредица, състояща се от 8 букви и/или цифри плюс максимално триразредно разширение (extension), например PROG1.ASM.
- Име на маршрута (path name) – йерархична последователност от означения на файловете (с максимално 64 знака), които са разделени едно от друго чрез обратна наклонена черта (Backslash), например TASM\BIN\PROG1.ASM.
- Дума за обръщение към файла (file handle) – 16-битова дума, която се генерира от MS-DOS при създаването, респ. отварянето на файла и след това трябва да бъде използвана от потребителите при обръщение към този файл.
- Вид на достъпа (access mode) – при отваряне на файла трябва да бъде решено как след това да се обръщаме към него:

  - само за четене (0)
  - само за запис (1)
  - за четене и запис (2)

- Атрибут на файла (file attribute) – MS-DOS познава 7 основни типа файлове, които се означават със съответния атрибут на файла (шестнайсетично число):

  - стандартен файл (00H)
  - файл само за четене (Read–Only) (01H)
  - скрит файл (Hidden) (02H)
  - системен файл (System) (04H)
  - етикет за носителя на данни (Volume Lable) (08H)
  - поддиректория (Subdirectory) (10H)
  - бит за архивиране (Archive–Bit) (20H)

Файлове, които имат повече от една от посочените характеристики, съдържат като число атрибут сумата от съответните числа. Битът за архивиране се установява, когато във файла са записани данни и той е затворен. По това програмите за осигуряване на данни – т. нар. архивиращи програми (Backup програми), разпознават че един файл е междувременно обработван, и в нормалния случай след осигуряването му връщат бита за архивиране в нулирано състояние.

Съобщения за грешки – те се разпознават по това, че след извикването на съответната функция на MS DOS флагът за пренос (Carryflag) CF е установен. В такъв случай съответният код на грешката се намира в регистъра AX.

Какви дейности спадат към работа с файлове? Ако ще се работи за първи път с файл, той трябва да бъде създаден. Ако файлът вече съществува и искаме да се обърнем към него за четене или запис, той трябва да бъде отворен. За целта операционната система предлагаспоред желания вид на достъп един входен и/или един изходен буфер. Тогава може да бъде извършено съответното обръщение към файла. Когато и последните данни са прочетени от файла, респ. записани във файла, той трябва да бъде затворен.

15.1. Функции за работа с файлове

Операционната система MS-DOS предлага всички необходими действия за работа с файлове като подфункции на прекъсването 21H. Под "въвеждане" по-нататък трябва да се разбират съдържанията на регистрите преди извикването на функцията, а под "извеждане" – съдържанията след изпълнение на функцията.

Първото условие за да са възможни операциите четене (запис), е необходимо файлът да се отвори. Отварянето на файла означава, че четящата (пишещата) програма изпълнява следните операции:

- Намира по името на файла съответстващия запис в един от каталозите. (Ако запис за файла не е намерен, файлът първо трябва да се създаде.)
- Копира този запис (елемент на каталога) в своята работна област.
- Създава специална буферна област за обмен с диска (DTA – DISK TRANSFER AREA), където ще се намира прочетения запис или където ще се формира този запис, предназначен за извеждане във файл.

Сега ще изредим функциите за работа с файлове.

Прекъсване	**Функция**	**Въвеждане**	**Извеждане**
INT 21h	3Ch –> Създаване на файл	AH = 3Ch DS:DX = указател към файла, респ. маршрута на новосъздадения файл CX = атрибут на	при коректно протичане на функцията (CF = 0): AX = дума за обръщение към файла при погрешно

		файла	протичане (CF = 1): AX = 3 –> маршрута не е намерен 4 –> прекалено много отворени файлове 5 –> отказано обръщение
	3Dh –> Отваряне на файл	AH = 3Dh DS:DX = указател към име на затворен файла, който трябва да се отвори CX = вид на обръщението: 0 –> четене 1 –> запис 2 –> и двете	при коректно протичане на функцията (CF = 0): AX = дума за обръщение към файла при погрешно протичане (CF = 1): AX = 2 –> файла не е намерен 4 –> прекалено много отворени файлове 5 –> отказано обръщение 12 –> невалидно обръщение (погрешен вид обръщение)
	3Eh –> Затваряне на файл	AH = 3Eh BX = дума за обръщение към файла	при погрешно протичане на функцията (CF = 1): AX = 6 –> невалидна дума за обръщение към файл
	3Fh –> Четене от файл	AH = 3Fh BX = дума за обръщение към файла CX = брой на байтовете, които трябва да се прочетат	DS:DX = указател към буфера при коректно протичане на функцията (CF = 0): AX = брой на прочетените байтове; ако AX = 0, тогава край на файла (EOF) при погрешно

			протичане на функцията (CF = 1): AX = 5 –> отказано обръщение 6 –> невалидна дума за обръщение към файла
	40h –> Записване във файл	AH = 40h BX = дума за обръщение към файла CX = брой на байтовете, които трябва да се запишат DS:DX = указател към буфера	при коректно протичане на функцията (CF = 0): AX = брой на записаните байтове при погрешно протичане на функцията (CF = 1): AX = 5 –> отказано обръщение (Файлът не е отворен за запис) 6 –> невалидна дума за обръщение към файла
	43h –> Манипулиране на атрибута на файл	AH = 43h DS:DX = указател към стринг със спецификацията на файла AL = 1 –> set 0 –> get	при коректно протичане на функцията (CF = 0): CX = атрибут на файла при погрешно протичане на функцията (CF = 1): AX = 1 –> невалиден код на функция 2 –> файлът не е намерен 3 –> невалиден път към файла 5 –> отказано обръщение
	41h –> Изтриване на файл	AH = 41h DS:DX = адрес на стринг съдържащ	при погрешно протичане на функцията (CF = 1): AX =

		спецификацията на файла	2 –> файлът не е намерен 3 –> невалиден път към файла 5 –> отказан достъп, защото файла има read–only атрибут
	56h –> Преименуване на файл	AH = 56h DS:DX = адрес на стринг съдържащ текущото име на файла ES:DI = адрес на стринг съдържащ новото име на файла	при погрешно протичане на функцията (CF = 1): AX = 2 –> файлът не е намерен 3 –> невалиден път към файла 5 –> отказан достъп 11h –> различно устройство (подадено различно дисково устройство на двете имена на файла)

Функцията 3CH създава файла, отваря го за обръщение за запис и четене и предава дума за обръщение към файла. При този процес съдържанието на вече съществуващ файл със същото име се унищожава.

Функцията 3DH отваря вече запомнен файл и предава дума за обръщение към файла. Тя се употребява при всички обръщения към този файл.

Функцията 3FH прехвърля максимално CX байта на файла в зададен от приложната програма буфер.

Функцията 40H прехвърля във файла CX байта от един буфер, предварително зададен от приложната програма.

Функцията 43H може да бъде използвана да върне или да промени атрибутите на един файл. В AL предварително трябва да бъде поставен флаг за това кое действие искаме да извършим.

Една от причините тази функция да е важна е това, че тя позволява да се скрие един файл така, че да не се появява когато DIR, DEL и COPY командите се използват. Също така можем да променим атрибута на един файл на read–only, така че да го предпазим от промяна. Има още една

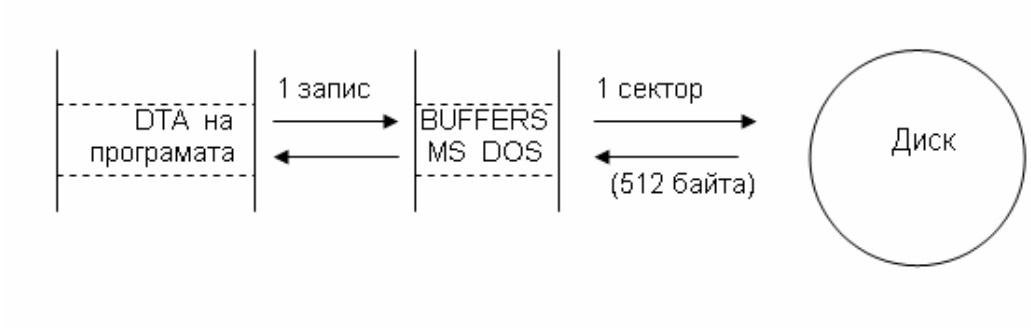
съществена употреба на тази функция – единственият начин да изтрием или обновим един read–only файл е като променим атрибута му на нормален (стандартен) файл.

Функцията 41H може да бъде използвана да бъде изтрит един файл от дисков носител. В DS:DX трябва да се намира адреса на ASCII стринг, съдържащ спецификацията на файла. Спецификацията на файла не може да включва специални символи.

Функцията 56H преименува файл ако указател към текущото му име е поставен в DS:DX и указател към новото му име в ES:DI. Двете имена трябва да бъдат ASCII стрингове. Стрингове не могат да включват специални символи. Тази функция може също да бъде използвана за да се премести един файл от една директория в друга, защото може да се специфицира различен път за всяко име на файла. Преместването на файл е различно от копирането. При преместването файлът няма да продължи да съществува на неговото оригинално място повече.

Всички операции за обмен на данни с диска са буферирани, понеже за един прием от диска може да бъде прочетена (и записана) порция информация по обем равна на един сектор (512 байта) – нито повече, нито по-малко. Операционната система създава за това свои работни области – BUFFERS.

Програмата, на която й е необходим един байт или един запис, като правило, по-къса от 512 байта, трябва да избере от буфера нужното й и да помести своето DTA. В операциите запис във файл, извежданата информация първо се формира в DTA, после се натрупва в системните буфери до 512 байта и едва тогава се записва на диска.



Фигура 33. Запис във файл

Затова при приключване на работа със файл програмата е длъжна да го затвори. При това очиства всички буфери и обновява записа в съответния каталог, иначе част от записваната информация може да се загуби.

Особено е важно затварянето на тепърва създаден файл, понеже само при затварянето на новия файл той се регистрира в каталога.

Следващият пример демонстрира създаване на файл:

```
model small
.data
        handle dw 0
        filename db 'my_file. txt', 0
        point_fname dd filename
.stack 256
.code
main:
        mov ax, @data
        mov ds, ax
        xor cx, cx
        lds dx, point_fname
        mov ah, 3ch
        int 21h
        jc exit
        mov handle, ax
exit:
        mov ax, 4c00h
        int 21h
end     main
```

Пример, демонстриращ четене на файл:

```
model small
.data
        string db 80 dup (" ")
        len_string=$–string
        point_fname dd string
.stack 256
.code
main:
        mov ax, @data
        mov ds, ax
        mov bx, 0
        mov cx, len_string
        lds dx, point_fname
        mov ah, 3fh
        int 21h
        jc exit
```

```
        mov bx, 1
        mov cx, len_string
        lds dx, point_fname
        mov ah, 40h
        int 21h
        jc exit
        nop
exit:
        mov ax, 4c00h
        int 21h
end     main
```

Пример, демонстриращ записване във файл:

```
model small
.data
        string db ‘function 40h’
        len_string=$–string
        point_fname dd string
.stack 256
.code
main:
        mov ax, @data
        mov ds, ax
        mov bx, 1
        mov cx, len_string
        lds dx, point_fname
        mov ah, 40h
        int 21h
        jc exit
        nop
exit:
        mov ax, 4c00h
        int 21h
end     main
```

15.2. Функции на MS DOS за директории

Прекъсване	Функция	Въвеждане	Извеждане
INT 21h	39h –> Създаване на директория (MKDIR)	AH = 39h DS:DX = адрес на стринг, съдържащ текущото име на директорията	CF=0 при успешно преименуване; CF=1 - AX=код за грешка: 3-несъществуващ път; 5-достъпът е забранен

	3Ah –> Изтриване на директория (RMDIR)	AH = 3Ah DS:DX = адрес на стринг, съдържащ текущото име на директорията	CF=0 - AX не е определен; CF=1 - AX=код за грешка: 3- несъществуващ път; 5- достъпът е забранен; 10h- опит за изтриване на текущия каталог
	3Bh –> Промяна на текущата директория (CHDIR)	AH = 3Bh DS:DX = адрес на стринг, съдържащ текущото име на директорията	CF=0 - AX не е определен; CF=1-AX=код за грешка: 3- пътят не е намерен
	47h –> Именуване (получаване) на текущия каталог на зададен диск (PWD)	AH = 47H DS:SI = буфер с име DL = № диска 0 –> текущия диск 1 –> A: 2 –> B: и т. н.	CF=0 при успешно изпълнение на функцията CF=1-AX=код за грешка: 0Fh – недостъпно дисково устройство

Да се изтрие може само празен каталог.

Имената се задават във формат ASCIIZ:

‘диск:\път\име’, 0
‘път\име’, 0 (път от текущия каталог)
‘име’, 0 (в текущия каталог)

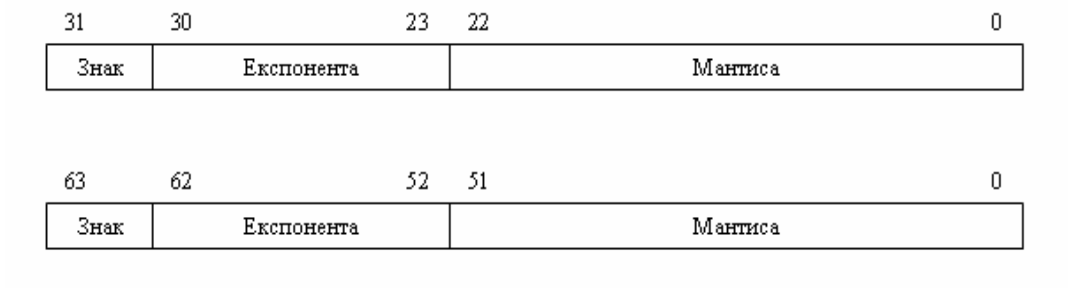
Тема 16. Програмиране на копроцесор

Едно число с плаваща запетая се състои от мантиса и експонента. Мантисата съдържа знака, цялата част и дробната част на числото. В числото 234. 56E+02, примерно, мантисата е 234. 56, а експонентата е +02. Други примери за реални числа са:

```
2. 5
+0. 4646
1. 024E+04      ; (1. 024 * 10 ^ 4)
–80000. 21
```

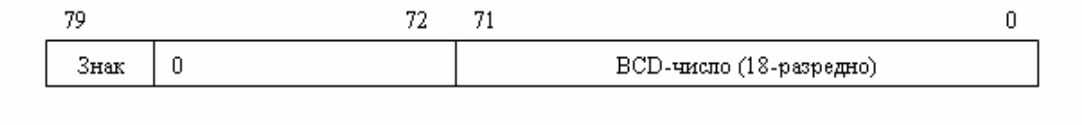
Новите процесори на Intel от 80486 съдържат един интегриран копроцесор за изчисления с числа с плаваща запетая. Поради това ще направим кратко въведение в начина на работа и на програмирането на този "помощник при пресмятането". Отчасти това въведение ще отговаря и за компютрите, имащи отделен копроцесор (например 80287 или 80387), понеже наборът от инструкции е в значителна степен еднакъв.

Копроцесорът образува един самостоятелен блок, който всъщност обменя данни с паметта, обаче резултатите от изчислителните операции все пак поставя в свои регистри. Копроцесорите на Intel могат да обработват три различни формата на числа: формат IEEE, който се използва от почти всички компилатори, двоичният формат на Intel и временният формат на реални числа (80 байта), който процесорите използват вътрешно. Допълнително копроцесорите обработват един целочислен (Integer) BCD формат и три двоични целочислени формата. Тук ще използваме от форматите с плаваща запетая само формата IEEE. Различават се два формата на (реални) числа с плаваща запетая: 32-битов формат на късо реално число и 64-битов формат на дълго реално число:



Фигура 34. Формати на числа с плаваща запетая

Целочисленият BCD формат се състои от 18-разредно число, което се запомня пакетирано в 10 байта – т. е. две цифри за байт. Старшият бит съдържа знака на числото, а останалите седем бита на старшия байт остават неизпозлвани:



Фигура 35. Целочислен BCD формат

Трите двоични целочислени формата съдържат съответно 16-битова дума, 32-битова двойна дума и 64-битова четворна дума като числа със знак (отрицателни числа в допълнителен двоичен код).

Копроцесорите съдържат вътрешно осем 80-битови регистри за данни – ST(0) до ST(7), които са организирани като стек, като ST(0) или накратко ST представлява върхът на стека. Когато се доведе данна в стека, всички данни на стека се придвижват в регистъра съответно с един номер нагоре. Някои инструкции могат да се обръщат директно към отделните стекови регистри (адрес ST(0) до ST(7)).

Аритметичните операции използват винаги върха на стека (ST) и допълнително един друг регистър или операнд от паметта, който трябва да е в поддържан от копроцесора формат. Резултатът от операцията се намира в ST, респ. ST(n).

Инструкциите за зареждане довеждат дадения изходен операнд от паметта в най-горния стеков регистър ST. Инструкциите за запомняне запомнят съдържанието на най-горния стеков регистър в зададения целеви адрес на паметта. При това съществува възможност горният (старшият) стеков регистър само да копира (т. е. стекът да запази съдържанието си) или след копирането да се извърши операцията POP – следователно изходния операнд да изчезне от стека.

При обмен на данни регистрите на стека може да бъдат адресирани като нормалните регистри – да участват с два операнда в инструкция. Единият е същевременно и операнд-цел, което значи, че неговото съдържание ще бъде унищожено от резултата.

Всички въведени числа предварително се преобразуват от копроцесора във временния 80-битов формат с плаваща запетая, а при извеждане се преобразуват отново в зададения формат на числа с плаваща запетая или според инструкцията в 10-байтово пакетирано цяло число или 16, 32 или 64-битово двоично цяло число. При преобразуването в цяло число точността се загубва, тъй като младшият разред пред запетаята се закръглява, а разредите

след запетаята отпадат. Може обаче с един програмен трик да се запазят разредите след запетаята, при което например резултатът от пресмятането – преди обратното преобразуване в BCD число – се умножава по 100, а при извеждането запетаята се вмъква между втория и третия разред от края.

16.1. Дефиниране на реални числа

В тази секция ще използване асемблер директиви, за да дефинираме реални числа.

16.1.1. Define Doubleword (DD)

Директивата DD казва на асемблера да резервира памет за едно 4-байтово число. Числото може да бъде декларирано използвайки или шестнайсетични или десетични цифри. Последното е наречено десетично реално число и може да има до 8 цифри. Примери:

```
dd      12345. 678      ; цифри, десетична точка
dd      +1. 5E+02       ; стойността е 1500
```

Едно кодирано реално число е реално число, което е представено като стринг от шестнайсетични цифри. Шестнайсетичните цифри се отнасят за число, което съответства на бинарното представяне на реално число с плаваща запетая с мантиса, експонента и знак (както беше уточнено в предишната точка). Асемблер изисква буквата R да бъде поставена като суфикс на такова число. Примерно, числото +1. 0 може да бъде представено като следното кодирано реално число:

```
dd      3f80000r        ; кодирано реално, еквивалентно на +1. 0
```

16.1.2. Define Quardword (DQ)

Директивата DQ резервира памет за 8-байтова променлива. Инициализационната стойност може да бъде едно 8-байтово кодирано реално число, едно десетично число или едно двоично цяло число. DQ може да бъде използвано за създаването на число с двойна точност (double число) като десетично реално число или като кодирано реално:

```
float1  dq      +1. 5E+10
float2  dq      2. 56E+307
float3  dq      3f00000000000000r
```

16.1.3. Define Tenbyte (DT)

Директивата DT резервира памет за 10-байтова променлива. Съдържанието й може да бъде кодирано реално число, десетично реално число, пакетирано десетично число, или двоично цяло число. Десетичните цифри се предполага, че са съхранени в пакетиран BCD формат.

Пример:

```
packed_1    dt    123456789012345678
packed_2    dt    –00123456789012345
```

Асемблера предполага, че бройната система на число, декларирано с DT е шестнайсетична; следователно, десетичният индикатор (d) трябва да бъде използван ако декларираме десетична стойност.

Директивата DT се използва и за създаването на 10-байтово реално число, съобразено с IEEE временния реален формат (този формат е използван от копроцесора). Едно реално число може да бъде специфицирано или като десетично или в шестнайсетичен формат. Шестнайсетичното кодирано число трябва да бъде 20 цифрено:

```
float1 dt 1. 5                    ; десетично реално
float2 dt 3f00000000000000r       ; кодирано реално
```

Като прост пример за програма, демонстрираща употребата на инструкции на копроцесора за работа с числа с плаваща запетая, трябва да се пресметнат 15 стойности на функцията y = 3, 5*x^2 + 12, 5 за x = 1 до 15. Резултатът се закръглява и се записва в паметта като двоично цяло число. Резултатът трябва да се извежда на екрана.

Една особеност представлява инструкцията WAIT. Процесорът и копроцесорът могат по принцип да работят паралелно. Ако процесорът се нуждае от данни, които копроцесорът предоставя, трябва да се гарантира, че копроцесорът е завършил обработката си, преди процесорът да продължи работата си с тези данни. Ако това не е така – в общия случай инструкциите за работа с плаваща запетая са по-продължителни от инструкциите на процесора – трябва да се вмъкне една инструкция WAIT. Тогава процесорът изчаква, докато копроцесорът потвърди завършването на последната инструкция за работа с плаваща запетая.

16.2. Формати на данните при копроцесор 8087

Копроцесорът може да изпълнява операции само с данни, съхранявани в неговите 8 десетобайтови (80-разрядни) регистри и в паметта. При четене на данни от паметта копроцесорът ги преобразува в специален формат на временно реално число (Temp. Real). При запис на числа в паметта процесорът може да преобразува числото във всеки от 7-те свои формати.

- WORD – Цяла дума, 2-байтова дума (DW) в двоичен код.
- INTEGER – Късо цяло, 4-байтово число (DD)в двоичен код.
- LONG_INTEGER – Дълго цяло, 8-байтово число (DQ)в двоичен код.
- BCD_INTEGER – Опаковано 10-байтово цяло число в двоично-десетичен код (DT).
- SHORT_REAL – Късо 4-байтово (DD) реално число във формат плаваща запетая.
- LONG_REAL – Дълго 8-байтово (DQ) реално число във формат с плаваща запетая.
- TEMP_REAL – 10-байтово (DT) реално число във временен плаващ формат.

Първите четири формати на цели числа могат да се обработват и от процесора 8088/8086. Останалите 3 формата могат правилно да се обработват само от копроцесора.

16.3. Команди за обмен на данни.

FLD: Записва операнд в стека на копроцесора. Преди това 32 или 64-битовият операнд се преобразува в едно временно реално число.

FILD: Преобразува операнд в едно временно реално число и го записва във върха на стека.

FBLD: Преобразува едно 80-битово число (19-разредно BCD-число) във временно реално число и го изпраща в стека.

FIST: Закръглява стойността на числото във върха на стека (ST) съобразно със съдържанието на контролните битове към едно цяло число и запомня резултата в цел.

FISTP: Закръглява стойността на числото във върха на стека (ST) съобразно със съдържанието на контролните битове към едно цяло число и запомня резултата в цел. След това изпълнява една операция POP.

Следващият пример изследва командите за предаване на данни. Изпълнете го под управлението на дебъгера и прегледайте как се променя състоянието на регистрите на копроцесора в прозореца Numeric processor. Изследвайте работата на тази програма с отворен прозорец на Dump, т. к. тогава добре се виждат различията в представянето на различните типове данни.

```
model use16 small
.stack 100h
.data
        ch_dt dt 43567          ;ch_dt=00 00 00 00 00 00 00 04 35 67
        x dw 3                  ;x=00 03
        y_real dq 34e7          ;y_real=41 b4 43 fd 00 00
        ch_dt_st dt 0
        x_st dw 0
        y_real_st dq 0
.code
main    proc
        mov ax, @data
        mov ds, ax
        fbld ch_dt              ;st(0)=43S67
        fild x                  ;st(l)=43567, st(0)=3
        fild y_real             ;st(2)=43567, st(l)=3, st(0)=340000000
        fxch st(2)              ;st(2)=340000000, st(l)=3, st(0)=43567
        fbstp ch_dt_st          ;st(l)=340000000, st(0)=3
                                ;ch_dt_st=00 00 00 00 00 00 00 04 35 67
        fistp x_st              ;st(0)=340000000, x_st=00
        fstp y_real_st          ;y_real_st=41 b4 43 fd 00
exit:
        mov ax, 4c00h
        int 21h
main    endp
end     main
```

FXCH: Разменят се елементи от стека (съдържанието на ST и ST(n)).

16.4. Целочислени аритметични команди

Целочислените аритметични команди са предназначени за работа в такива участъци от изчислителните алгоритми, където в качеството на входни данни се използват цели числа в паметта във формат дума и късо реално число, с размерност 16 и 32 бита.

FIADD: събира стойността на ST(0) и на целочисления източник, в качеството на който се използва 16 или 32-разряден операнд в паметта. Резултатът от събирането се запомня в регистъра на стека на копроцесора ST(0).

FISUB: изважда стойността на целочисления източник от ST(0). Резултатът от изваждането се запомня в регистъра на стека на копроцесора ST(0). В качеството на източник се използва 16 или 32-разряден целочислен операнд в паметта.

FIMUL: умножава стойността на целочисления източник на съдържимото на ST(0). Резултатът от умножението се запомня в регистъра на стека на копроцесора ST(0). В качеството на източник се използва 16 или 32-разряден целочислен операнд в паметта.

FIDIV: дели съдържимото на ST(0) на стойността на целочисления източник. Резултатът от делението се запомня в регистъра на стека на копроцесора ST(0). В качеството на източник се използва 16 или 32-разряден целочислен операнд в паметта.

За командите, реализиращи аритметичните действия деление и изваждане е важен реда на разположение на операндите. По тази причина системата команди на копроцесора съдържа съответстващи реверсивни команди, повишаващи удобството при програмиране на изчислителни алгоритми. За да се отличават тези команди от обикновените команди за деление и изваждане, техните мнемокодове завършват със символа R.

FISUBR: изважда стойността на ST(0) от целочислен източник. Резултатът от изваждането се запомня в регистъра на стека на копроцесора ST(0). В качеството на източника се използва 16 или 32-разряден целочислен операнд в паметта.

FIDIVR: дели стойността на целочисления източник на съдържимото на ST(0). Резултатът от делението се запомня в регистъра на стека на копроцесора ST(0). В качеството на източник се използва 16 или 32-разряден целочислен операнд в паметта.

Следващият пример изследва целочислените аритметични команди на копроцесора.

Програмата изчислява стойността на ***u*** по формулата:

$$u = \frac{x - y}{a}, \text{за } a \neq 0$$

$u = x+y$, за $a=0$

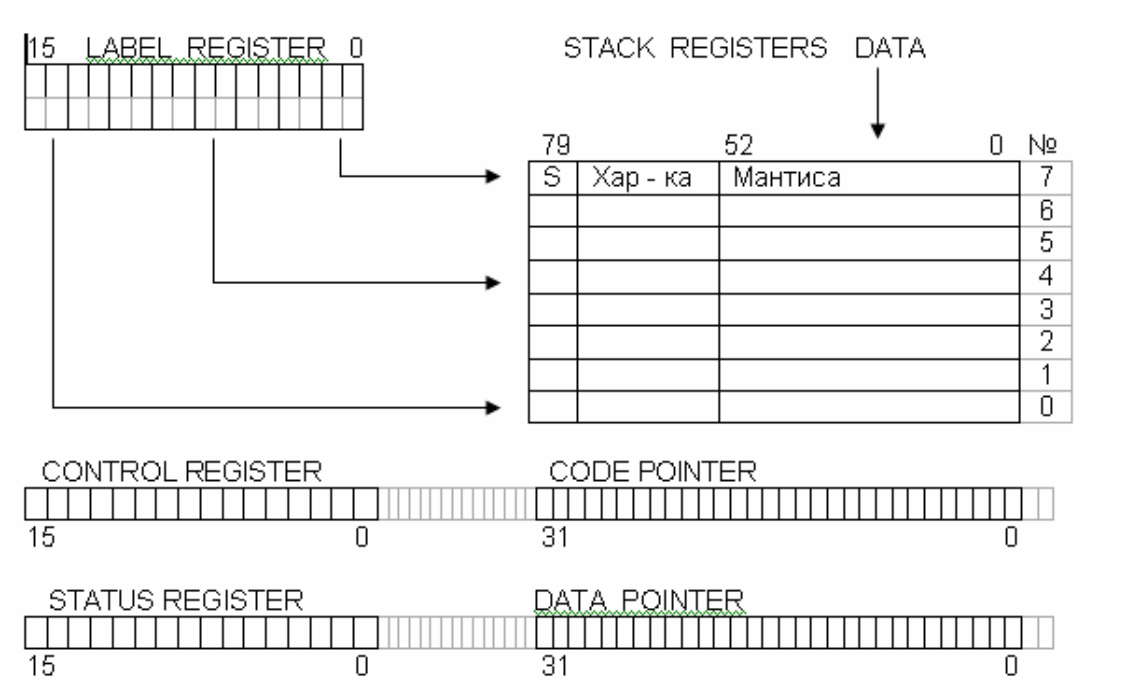
Всички променливи x, y и a са от цял типа във формат дума. Резултатът е необходимо да се съхрани в клетка от паметта във формат за десетично число.

```
model use16 small
.stack 100h
.data
        a dw 0
        x dw 8
        y dw 4
        u dt 0
.code
main    proc
        mov ax, @data
        mov ds, ax
        finit ; привеждаме копроцесора в начално състояние
        fild a ; зареждаме стойността на a в st(0)
        fxam ; определяме типа на a
        jc no_null
        jp no_null
        jnz no_null
        ;за a≠0, изчисляване на формулата u=x+y
        fild x
        fiadd y
        fbstp u ; запомняне на BCD число и операция pop
        jmp exit
        no_null:
        ;за a=0, изчисляване на формулата u=(x–y)/a
        fild x
        fisub y
        fidiv a
        fbstp u ; запомняне на BCD число и операция pop
exit:
        mov ax, 4c00h
        int 21h
main    endp
end     main
```

FINIT: Инициализира копроцесора.

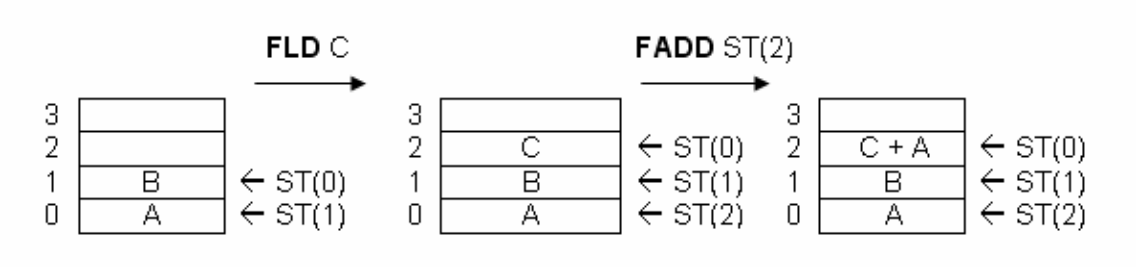
FXAM: Проверява най-горния елемент на стека и установява съответните флагове за условията в думата на състоянието на копроцесора.

16.5. Функционален модел на копроцесора 8087



Фигура 36. Функционален модел на копроцесора 8087

- Регистровият стек на 8087 се състои от осем 80-разрядни регистъра.
- Регистрите на 8087 функционират като обикновен стек на процесора 8088/8086 с едно ограничение – броят на позициите в стека е само 8.
- За всяка операция единият от операндите винаги е върха на стека, обозначен като ST(0) или просто ST. Следващият регистър се обозначава с ST(1), следва – ST(2), ST(3), . . . ST(7).

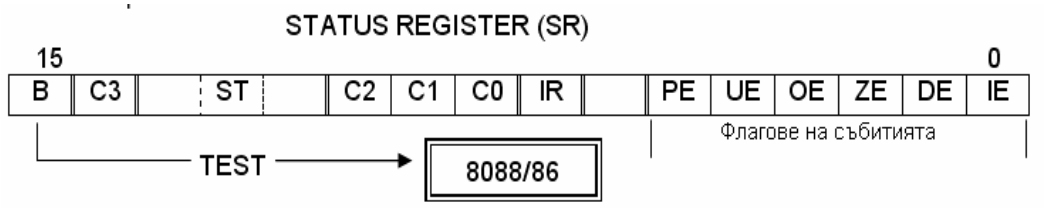


Фигура 37. Работа на командите FLD и FADD

- В специален двубайтов "регистър на етикетите" (Label Register) двубитови полета отбелязват кой от стековите регистри е свободен и кой е зает. Всеки опит за поставяне на данни във вече зает регистър води до особен случай Invalid Operation(недействителна операция). Да се освободи елемент от стека, не явяващ се връх е възможно само със специална команда FFREE.

16.5.1. Дума на състоянието (Status Word, SW)

Текущото положение на върха на стека (номерът на регистъра от 000 до 111 на съответстващия ST(0)) се съхранява в 3-битово поле (ST) на 16-разрядния регистър на статуса (Status Reg.).



Фигура 38. Дума на състоянието

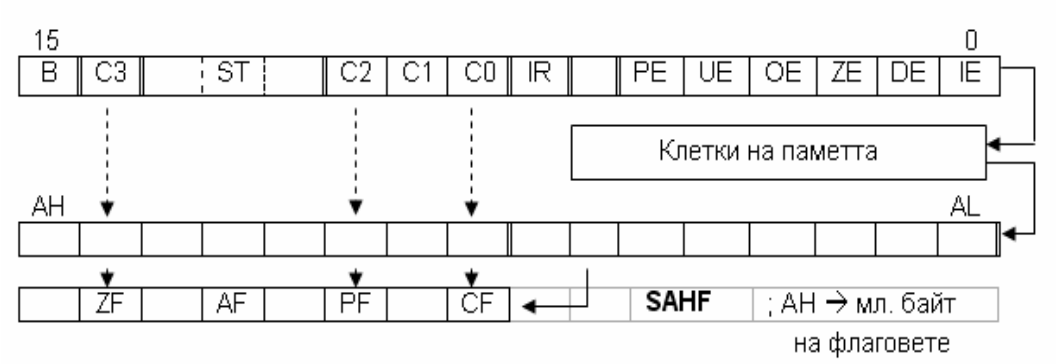
Думата на състоянието (StatusWord, SW), записана в този регистър, отразява текущото състояние на копроцесора. В нея има отделни битови полета (флагове), единичните стойности на които показват възникване на особени ситуации:

IE – Недействителна операция **OE – Препълване**
DE – Денормализиран операнд **UE – Загуба на значимост**
ZE – Деление на нула **PE – Загуба на точност**

Флагът **IR** = 1 показва, че апаратните прекъсвания са забранени. Командата **FCLEX** занулява всички флагове на особени ситуации.

Флаговете **C0, C1, C2, C3** на така наричания код на условието се установяват от командите за проверка на операнда и от командите за сравнение **FCOM**. Разположението истойностите на флаговете C0 и C3 съответстват на флаговете CF и ZF във флаговия регистър на процесора 8088/86.

След изпълнението на операцията за сравнение от копроцесора тези флагове чрез паметта могат да бъдат заредени във флаговия регистър на процесора за изпълнение на необходимата команда за условен преход (собствени команди за преход копроцесора няма).

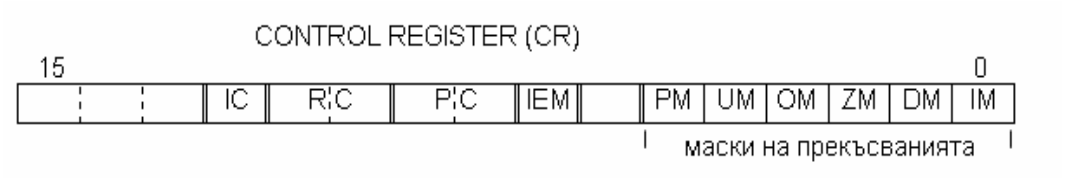


Фигура 39. Изпълнение на операцията за сравнение от копроцесора

Старшият бит на думата на състоянието – флагът “B” (Busy, зает) съхранява стойност 1, докато копроцесорът е зает с изпълнението на някоя операция. Той се предава по линя та TEST на копроцесора и осигурява синхронизмация с процесор по командата WAIT или FWAIT.

16.5.2. Управляваща дума (Control Word, CW)

Още един специален 2-байтов регистър (Control Register) съдържа управляваща дума (Control Word), определяща режимите на работа на копроцесора 8087.



Фигура 40. Управляваща дума

Битовете **PM, DM, ZM, OM, UM и PM** съответстват на флаговете за особени ситуации в думата на състоянието. Единичните стойности на тези битове забраняват (маскират) апаратните прекъсвания от копроцесора при възникване на съответните ситуации. Така, акомаската ZM=1, тогава при деление на нула изпълнението на програмата ще продължи. Флагът ZE=1 в думата на състоянието показва за възникване на тази ситуация, резултатът ще бъде записан в един от формите за безкрайност (Infinity) и програмата ще може по своему да обработи тази ситуация. При невалидна операция (Invalid Operation – извличане на корен от отрицателно число или при запис в стека на 9-ти операнд) копроцесорът установява в думата на състоянието флаг IE=1 и изпраща съответстващото апаратно прекъсване. Но ако в управляващата дума битът IM е маскиран (IM=1), няма да се генерира прекъсването и резултатът ще получи специална стойност **NAN** (Not A Number – не е число).

Двубитовото поле **PC** управлява точността на междинните изчисления (P– Precision, точност).

PC = 00 – 24 бита

10 – 53 бита

11 – 64 бита

Двубитовото поле **RC** управлява закръгляването на резултата (R – Round).

RC = 00 – до най-близкото цяло число
01 – закръгляване нагоре (к +Inf)
10 – закръгляване надолу (к –Inf)
11 – отсичане в посоката на нулата

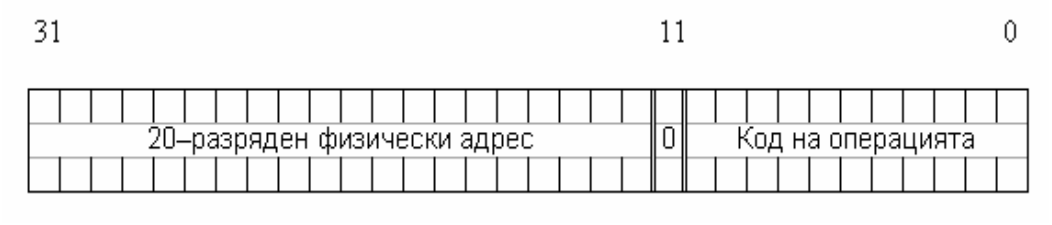
Битът **IC** управлява безкрайността (Inf, Infinity).

при IC = 0 – +Inf и –Inf не се различават
при IC = 1 – +Inf и –Inf са различни

По премълчаване IC=0.

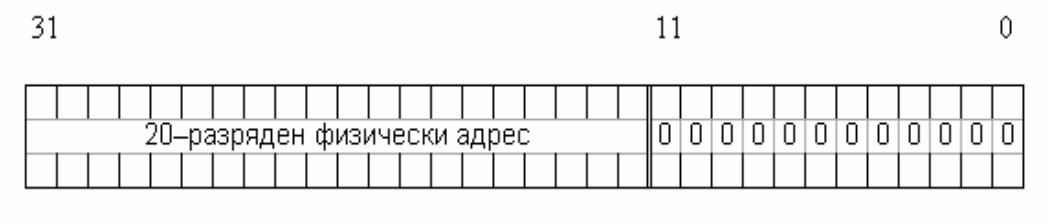
16.5.3. Работна среда и състояние на копроцесора

Два 4-байтови регистъра съхраняват 20-разрядните адреси (Segment*16+Offset) на последната изпълнена команда CODE POINTER (Указател на командата)



Фигура 41. Указател на командата

и на последната от извикваните клетки на паметта DATAPOINTER (Указател на операнда)



Фигура 42. Указател на операнда

Съдържимото на всички специални регистри:

[Control Word] + [Status Word] + [Label Register] + [Code Pointer] + [Data Pointer]

изгражда 14-байтовата работна среда на копроцесора 8087 (ENVIRONMENT). Командата FSTENV (STore ENVironment) записва работната среда в паметта на посочения адрес, а командата FLDENV (LoaD ENVironment) я зарежда от паметта в регистрите.

Командите FSAVE и FRSTOR записват в паметта и зареждат от паметта в регистрите както работната среда, така и съдържимото на всичките 8 стекови регистри (състояние на копроцесора). Затова в паметта следва да се резервира област с размер 94 байта.

Тема 17. Връзка на асемблер с езиците от високо ниво

Пакетът MASM има вградени средства, визуално приближаващи написаната програма до програмите на езиците от високо ниво. Тези директиви за формиране на конструкции са аналогични на условните и циклическите оператори на езиците от високо ниво: .IF, .ELSE, .ELSEIF, .ENDIF, .REPEAT, .UNTIL, .WHILE, .ENDW, .BREAK, .CONTINUE. Всички директиви са предшествани от символа точка. Използването на дадените конструкции може да се препоръчва с цел повишаване надежността на кода. Програмистът на асемблер така повишава логическото ниво на контролируемите

синтактични конструкции, намалява вероятността от логически грешки, като ускорява процеса на разработката.

Писането на асемблер на обемни програми е слабата страна на езика. Разбира се, това не винаги е необходимо. Ако програмата не е предназначена за решаване на някакви системни задачи, изискващи максимално ефективно използване ресурсите на компютъра, ако към нея няма специални изисквания за размера и времето за изпълнение, тогава е уместно да се избере един от езиците на високо ниво. Съществува и трети вариант: да се комбинират програмите на високо ниво с код на асемблер. Последното обикновено се използва в случай, че във вашата програма има фрагменти, които или не е възможно да бъдат реализирани без асемблер или асемблер може значително да подобри ефективността на програмата.

Повечето компилатори отчитат възможността за комбиниране на кода с асемблер. Механизмите им за връзка с асемблер обикновено са идентични. Единствено условие за изпълнение е уточняването в документацията на езика необходимите параметри за връзка и особеностите, свързани с организацията на връзката с кода на асемблер. Не съществуват универсални препоръки по този въпрос.

Съществуват два варианта за комбиниране на програми на езици от високо ниво и асемблер:

- Използване на оператори от тип inline и модул на асемблер за вграждане. Този вариант в значителна степен зависи от синтаксиса на езика от високо ниво и от конкретния компилатор. Тук се предполага, че асемблерните кодове са във вид на команди на асемблер или директно машинни команди и се вграждат в текста на програмата, написана на език от високо ниво. Компилаторът на езика разпознава командите (машинния код), като такива на асемблер и без изменения ги включва във формирания от него обектен код. Този вариант е удобен, когато се налага вмъкване на неголям фрагмент.
- Използване на външни процедури и функции. Този вариант е по-универсален за комбиниране. Той притежава следните предимства:

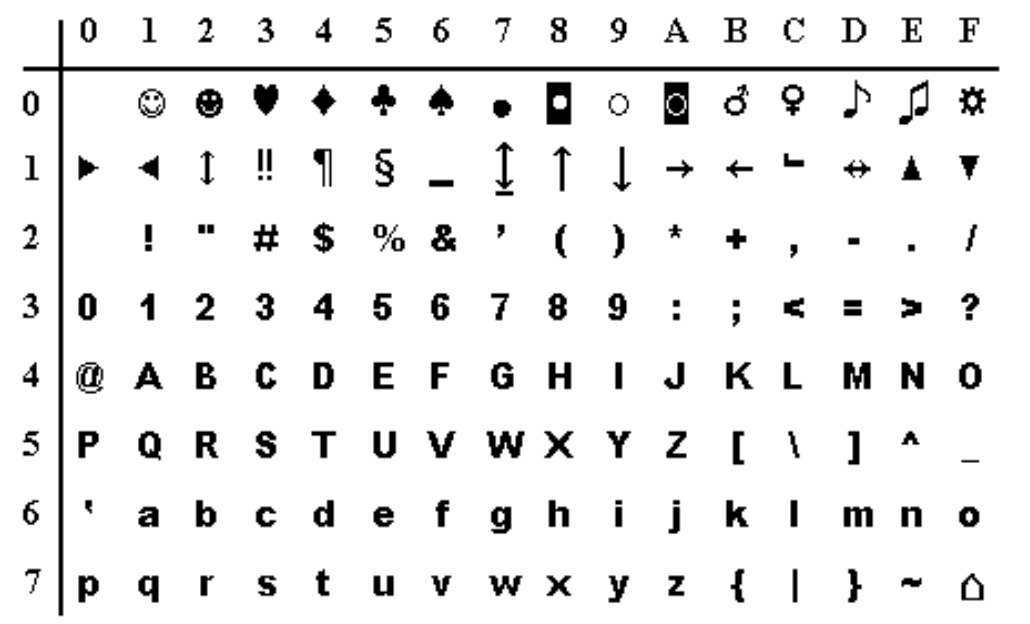
  - Писането и дебъгването на програмите може да става независимо;
  - Написаните подпрограми могат да се използват в други проекти;
  - Облекчава се модифицирането и съпровождането на програмите в течение на жизнения цикъл на проекта.

Заключение

Всичко разказано в тази книга е разчетено да работи под управлението на MS DOS в реален режим на работа на процесора (режиме V86), наследен още от седемдесетте години. В този режим процесорът не е способен да адресира в паметта повече от един мега байт. Поради факта, че при адресацията се използват 16-битови отмествания е невъзможно да се работи с масиви по-големи от 65 536 байта. Защитеният режим е лишен от тези недостатъци, при него могат да се адресират участъци от паметта с размер 4 гига байта като към един непрекъснат масив и да се забрави за сегменти и отмествания. Този режим е по-сложен от реалния, затова за да се превключи процесора в този режим и да работи в него и да поддържа работата в този режим трябва да се напише операционна система. Ако процесорът вече се намира под управлението на тази операционна система, която го е превела в защитен режим, например Windows 95, тя, най-вероятно, няма да разреши на програмата да я отстрани от управлението на компютъра. С такава цел са били разработени специални интерфейси, позволяващи на програмите, стартирани в режиме V86 в MS DOS, да се превключат в защитен режим с извикване на съответното прекъсване – VCPI и DPMI.

Приложения

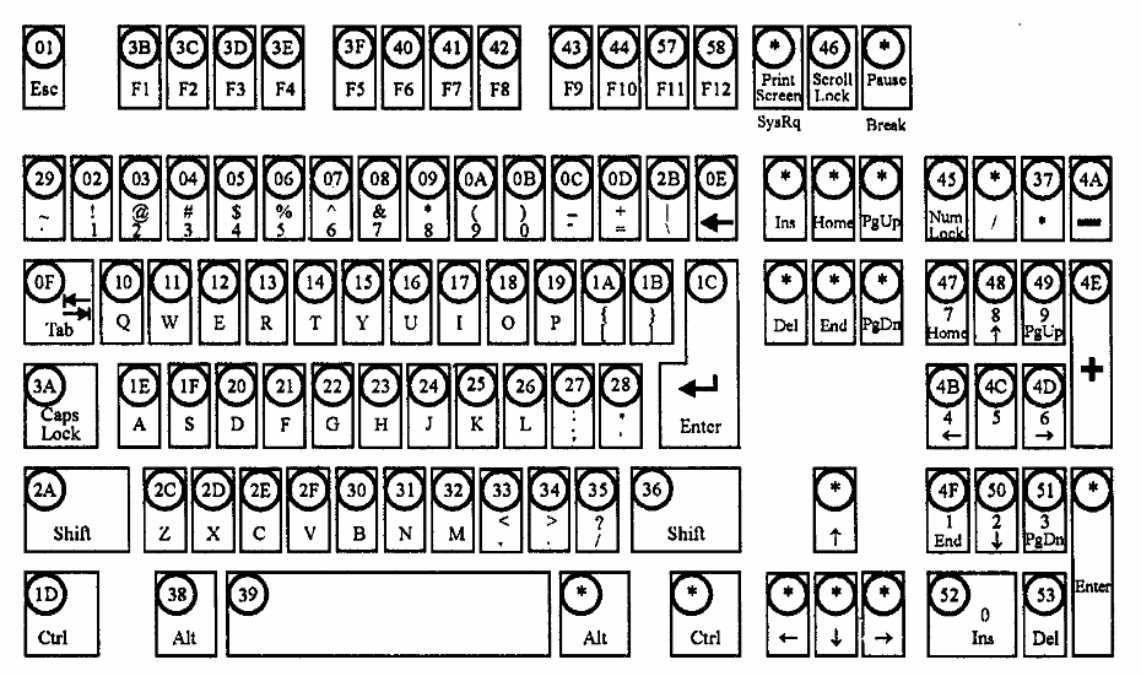
Приложение 1. Таблица на ASCII символите



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
1	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂

Фигура 43. Таблица на ASCII символите

Приложение 2. Скан кодове на клавишите



Фигура 44. Скан кодове на клавишите в шестнайсетичен код

Приложение 3. Инструкции на процесорите IA-32

AAA		ASCII Adjust AL after Addition: коригира резултата след събиране до десетична цифра (0–9). Предходната инструкция за събиране трябва да остави 8-битова сума в AL. Ако сумата е по-голяма от 9h, AH се инкрементира и се установяват флаговете CF и AF.
AAD		ASCII Adjust AX before Devision: преобразува непакетирани BCD цифри от AH (старша цифра) и AL (младша цифра) до двоично число в AX. Използва се за подготовка на непакетирано BCD число в AX за делене.
AAM		ASCII Adjust after Multiply: преобразува 8-битово двоично число (по-малко от 100D) в AL до непакетирано BCD число в AX. Старшата цифра се записва в AH, а младшата – в AL. Използва се за корекция на произведението след умножение с MUL на непакетирани BCD цифри в AH и AL. Използва се и за корекция на частното след делене с DIV на двоично число (по-малко от 100D) в AX с непакетирано BCD число.
AAS		ASCII Adjust after Subtraction: коригира резултата след изваждане до десетична цифра (0–9). Предходната инструкция за изваждане трябва да остави 8-битов резултат в AL. Ако резултатът е по-голям от 9h, AH се декрементира и се установяват флаговете CF и AF.
ADC		ADd with Carry: събира операндите източник, приемник и флага CF и записва резултата в операнда-приемник. Използва се за събиране на числа с дължина няколко байта (или няколко думи).

ADD		ADDition: събира операндите източник и приемник и записва резултата в операнда-приемник.
AND		logical AND: изпълнява поразредна логическа операция AND над операндите източник и приемник и записва резултата в операнда-приемник. Всеки бит на резултата е равен на 1, ако съответните битове на двата операнда са 1, в противен случай резултатът е 0.
ARPL		Adjust Requested Privilege Level (само за i286+ защитен режим): проверява дали поисканото ниво на привилегии RPL на приемника (битове 0 и 1 от стойността на селектора) е по-малко от RPL на източника. Ако не е, инструкцията коригира RPL на приемника да отговаря на стойността на източника. Операндът-приемник трябва да е 16 b (памет или регистър) и да съдържа стойността на селектора. Операндът-източник трябва да е 16 b регистър, съдържащ тестващата стойност. Флагът ZF се установява при корекция на приемника, в противен случай се нулира. Използва се само в защитен режим за системни програми.
BOUND		check array BOUNDs (само за i286+): проверява дали стойността на индекс със знак е в границите на масив. Операндът-приемник може да бъде произволен 16 b регистър, съдържащ проверявания индекс. В този случай операндът-източник трябва да бъде 32 b операнд-памет, в който младшата и старшата думи съдържат съответно началната и крайна граници на масив. (За i386+ операндът-приемник може да бъде 32 b регистър и в този случай операндът-източник трябва да бъде 64 b, съдържащ 32 b граници.) Ако първият операнд е по-малък от първата граница или по-голям от втората, се генерира прекъсване #5.
BSF		Bit Scan Forward (само за i386+): сканира операнда-източник, търсейки бит със стойност 1. Търсенето е в посока от бит с индекс 0 към най-старшия бит. Ако се срещне бит със стойност 1, флагът ZF се нулира, а в операнда-приемник се зарежда индексът на първия

		срещнат ненулев бит. Ако липсват битове със стойност 1, флагът ZF=0.
BSR		Bit Scan Reverse (само за i386+): сканира операнда-източник, търсейки бит със стойност 1. Търсенето е в посока от най-старшия бит към бит с индекс 0. Ако се срещне бит със стойност 1, флагът ZF се нулира, а в операнда-приемник се зарежда индексът на първия срещнат ненулев бит. Ако липсват битове със стойност 1, флагът ZF=0.
BSWAP		Byte SWAP (само за i486+): приема като операнд 32 b регистър и разменя стойностите на първия и четвъртия байт, както и на втория и третия. Стойностите на битовете в байтовете не се променят. Използва се за бързо преминаване от стил за съхранение в 8086 към стил, при който старшият байт се съхранява на по-младши адрес в паметта.
BT		Bit Test (само за i386+): копира стойността на посочен бит във флага CF, където тя може да бъде проверена с инструкция JC или JNC. Операндът-приемник съдържа стойността, в която се съдържа битът, а операндът-източник посочва позицията на бита.
BTC		Bit Test and Complement (само за i386+): копира стойността на посочен бит във флага CF, където тя може да бъде проверена с инструкция JC или JNC, след което стойността на бита се инвертира в операнда-приемник. Операндът-приемник съдържа стойността, в която се съдържа битът, а операндът-източник посочва позицията на бита.
BTR		Bit Test and Reset (само за i386+): копира стойността на посочен бит във флага CF, където тя може да бъде проверена с инструкция JC или JNC, след което битът се нулира в операнда-приемник. Операндът-приемник съдържа стойността, в която се съдържа битът, а операндът-източник посочва позицията на бита.

BTS		Bit Test and Set (само за i386+): копира стойността на посочен бит във флага CF, където тя може да бъде проверена с инструкция JC или JNC, след което битът се установява в 1 в операнда-приемник. Операндът-приемник съдържа стойността, в която се съдържа битът, а операндът-източник посочва позицията на бита.
CALL		CALL procedure: извиква процедура. Адресът на следващата инструкция се записва в стека и управлението се предава към адреса, определен от операнда При близко обръщане в стека се записва стойността на IP и след това в IP се зарежда новия адрес. При далечно обръщане в стека най-напред се записва стойността на CS, а след това в CS се зарежда новият сегментен адрес. След това в стека се записва отместването (IP) и в IP се зарежда новото отместване. Инструкция RET извлича адреса от стека, така че изпълнението на програмата продължава от инструкцията, разположена след CALL.
CBW		Convert Byte to Word: преобразува числото с дължина 1 B със знак в AL до число с дължина дума със знак в AX чрез разширяване на знаковия бит на AL във всички битове на AH.
CDQ		Convert Doubleword to Quadword (само за i386+): преобразува числото с дължина двойна дума със знак в EAX до число с дължина четворна дума със знак в двойката регистри EDX:EAX чрез разширяване на знаковия бит на EAX във всички битове на EDX.
CLC		CLear Carry flag: нулира флага за пренос CF.
CLD		CLear Direction flag: нулира флага за посока DF. Всички следващи операции над низове се изпълняват в посока нагоре (от ниските към високите адреси) чрез увеличаване на съответните индексни регистри.
CLI		CLear Interrupt flag: нулира флага за прекъсване IF.

		Когато флагът IF е нулиран, маскируемите прекъсвания не се разпознават, докато флагът не се установи с инструкция STI. В защитен режим CLI нулира флага само, ако нивото на привилегии на текущата задача е по-малко или равно на стойността на флага IOPL. В противен случай се фиксира грешка за обща защита.
CLTS		CLear Task Switched flag (само за i286+): нулира флага за превключване на задачите в думата за състояние MSW за i286 или на регистъра CR0 за i386+. Тази инструкция може да се използва само в системни програми, изпълнявани на ниво на привилегии 0.
CMC		CoMplement Carry flag: инвертира флага за пренос CF.
CMP		CoMpare oPerands: сравнява два операнда за изпълнение на следващо условно предаване на управлението или на инструкция за условно установяване на байт. Сравнението се прави чрез изваждане на операнда-източник от операнда-приемник и установяване на флаговете в зависимост от резултата. Действието на CMP е аналогично на инструкция SUB, но без съхраняване на резултата.
CMPS/ CMPSB/ CMPSW/ CMPSD		CoMPare Strings: сравнява два низа. DS:SI трябва да сочи към низа-източник, а ES:DI – към низа-приемник, даже ако са зададени операнди. При всяко сравняване елементът-приемник се изважда от елемента-източник и се установяват флаговете, без да се съхранява резултатът. SI и DI се коригират в зависимост от размера на операндите и стойността на флага за посока DF (увеличават се при DF=0 и се намаляват при DF=1). Ако се използва инструкцията CMPS, трябва да се посочат операнди, за да се определи размерът на обработваните елементи. За операнда-източник може да се посочи явно използван сегментен регистър. При използване на CMPSB (байтове), CMPSW (думи) или CMPSD (двойни думи, само за i386+) инструкцията сама определя размера

		на обработваните елементи. Инструкциите се използват обикновено с префикси за повторение. REPNE (или REPNZ) се използва за откриване на първото съответствие между два низа, а REPE (или REPZ) – за първото несъответствие. Преди сравнението CX трябва да съдържа максималния брой елементи за сравнение. След REPNE COMPS флагът ZF се нулира при липса на съответствие, а след REPE COMPS флагът се установява при съответствие. Когато инструкцията завърши, ES:DI и DS:SI сочат към следващия елемент (при нулиран флаг DF) или предишен елемент (при установен флаг DF) на съответствието или несъответствието. Ако CX=0, ES:DI и DS:SI сочат към елемента след или преди последното сравнение. Флагът ZF се управлява от резултата на последното сравнение, а не от стойността на CX.
CMPXCHG		CoMPare and eXCHanGe (само за I486+): сравнява операнда-приемник с акумулатора (AL, AX или EAX). При равенство операндът-източник се копира в приемника, в противен случай приемникът се копира в акумулатора. Флаговете се установяват в зависимост от резултата от сравнението.
CMPXCHG 8B		CoMPare and eXCHanGe 8 Bytes (само за Pentium): сравнява 64 b стойност от EDX:EAX с 64 b стойност на операнда в паметта. При равенство операндът се замества със стойността от ECX:EBX, а при неравенство стойността на операнда се зарежда в EDX:EAX. Флагът ZF се управлява от резултата от сравнението.
CPUID		CPU Identification code to eax (само за Pentium): зарежда идентификационен номер на CPU в EAX.
CWD		Convert Word to Doubleword: преобразува число със знак (с дължина дума) в AX до число със знак (двойна дума) в двойката регистри DX:AX чрез разширяване на знаковия бит на AX във всички

		битове на DX.
CWDE		Convert Word to Doubleword Extended (само за i386+): преобразува число с дължина дума със знак от AX в число със знак и двойна дума в EAX чрез разширяване на знаковия бит в старшата дума на EAX
DAA		Decimal Ajust al after Addition: коригира резултата след събиране до пакетирано BCD число (по-малко от 100D). Предишната инструкция за събиране трябва да остави 8 b двоична сума в AL, която се преобразува до пакетиран BCD формат с младша десетична цифра в младшите 4 b и старша десетична цифра – в старшите 4 b. Ако след коригиране сумата е по-голяма от 99h, установяват се флаговете CF и AF, в противен случай те се нулират.
DAS		Decimal Ajust al after Subtraction: коригира резултата след изваждане до пакетирано BCD число (по-малко от 100D). Предишната инструкция за изваждане трябва да остави 8 b двоична стойност в AL, която се преобразува до пакетиран BCD формат с младша десетична цифра в младшите 4 b и старша десетична цифра – в старшите 4 b. Ако след коригиране стойността е по-голяма от 99h, установяват се флаговете CF и AF, в противен случай те се нулират.
DEC		DECrement: изважда 1 от стойността на операнда. Понеже стойността на операнда се разглежда като цяло число без знак, инструкцията не въздейства върху флага CF.
DIV		DIVision unsigned: дели операнд-приемник по подразбиране с посочения операнд-източник. Двата операнда се разглеждат като числа без знак. Ако операндът-източник (делителят) е с дължина 16 b, приемникът по подразбиране (делимото) е в двойката регистри DX:AX. Частното се записва в AX, а остатъкът – в DX. Ако източникът е с дължина 8 b, делимото по подразбиране е в AX, а частното се записва в AL и остатъкът – в AH При i386+, ако

		източникът е EAX, частното се записва в EAX, а остатъкът в EDX.
ENTER		create a stack frame (само за i286+): създава стекова рамка за процедура, получаваща параметри чрез стека (изисквано в повечето езици за програмиране). Първият операнд задава броя на байтовете (динамична памет), резервирана за локални променливи. Вторият операнд посочва нивото на влагане на процедурата (в сорс-кода на език от високо ниво). Нивото на влагане трябва да бъде 0 за езиците, които не разрешават достъп до локални променливи от процедури от по-високо ниво (C, BASIC и FORTRAN). Когато първият операнд е 0, инструкцията е еквивалентна на PUSH bp; MOV bp, sp. Вижте съответната инструкция LEAVE за излизане от процедура.
ESC		passes information to the numeric coprocessor: подава информация към копроцесор. Изпълнението на ESC извежда съдържанието на операнда на шината за данни (ако се изисква); в противен случай се изпълнява NOP. Копроцесорите i87–i387 използват 6 b от инструкцията ESC за получаване на код на операцията и за започване на изпълнение на копроцесорна инструкция. Инструкцията ESC се използва за превключване към поток от копроцесорни инструкции (FLD, FST, FMUL и др.).
HLT		HaLT: спира изпълнението на CPU, докато ново прекъсване не поднови работата от инструкцията, следваща HLT. В защитен режим тази инструкция работи само на привилегировано ниво.
IDIV		signed DIVision: дели операнд-приемник по подразбиране с посочения операнд-източник. Двата операнда се разглеждат като числа със знак. Ако операндът-източник (делителят) е с дължина 16 b, приемникът по подразбиране (делимото) е в двойката регистри DX:AX. Частното се записва в AX, а остатъкът – в DX. Ако източникът е с дължина 8 b, делимото по подразбиране е в AX, а частното се записва в AL и остатъкът – в AH При i386+, ако

		източникът е EAX, частното се записва в EAX, а остатъкът в EDX.
IMUL		signed MULtiplication: умножава операнд-приемник по подразбиране с посочения операнд-източник. Двата операнда се разглеждат като числа със знак. Ако е посочен само един операнд с дължина 16 b, приемникът по подразбиране е в AX, а произведението се записва в двойката регистри DX:AX. Ако е посочен само един операнд с дължина 8 B, приемникът по подразбиране е в AL, а произведението се записва в AX. При i386+, ако източникът е EAX, произведението се записва в двойката регистри EDX:EAX. Флаговете CF и OF се установяват, ако произведението се разширява знаково в DX за 16 b операнди, в AX – за 8 B или в EDX – за 32 B. За процесори i186+ инструкцията има две допълнителни синтактически форми. При формата с два операнда първият операнд е 16 b регистър (множимо) и в него се записва резултатът, а вторият операнд е константа (множител). При формата с три операнда първият операнд е 16 b регистър (произведение), вторият е 16 b регистър или операнд–памет (множимо), а третият – константа (множител). И в двата случая флаговете OF и CF се установяват, ако резултатът не може да се побере в 16 b регистър. И двете форми могат да се използват както за числа със знак, така и за числа без знак. За i386+ операндите могат да бъдат 16 или 32 B. За i386+ съществува четвърта синтактическа форма с посочване на операнди-приемник и източник, като източникът може да бъде 16 или 32 b операнд-памет или регистър, а приемникът – регистър със същия размер. Флаговете OF и CF се установяват в 1, ако произведението не може да се побере в приемника.
IN		INput data from port: прехвърля байт или дума (или двойна дума за i386+) от порт към акумулатора. Адресът на порта се посочва от операнда-източник, който може да бъде DX или 8 b константа. Константи

		могат да се използват само за номера на портове, по-малки от 255, като за по-големи номера се използва съдържанието на DX. В защитен режим се регистрира грешка от обща защита, когато текущото ниво на привилегии е по-голямо от стойността на флага IOPL.
INC		INCrement: прибавя 1 към стойността на операнда. Понеже стойността на операнда се разглежда като цяло число без знак, инструкцията не въздейства върху флага CF.
INS/ INSB/ INSW/ INSD		INput String from port (само за i286+): получава низ от порт. Низът се счита за приемник и трябва да бъде указан от ES:DI (даже при посочен операнд). Входният порт се задава в DX. При получаването на всеки елемент от низа DI се коригира в зависимост от размера на операнда и стойността на флага за посока DF – DI се увеличава при нулиран флаг DF или се намалява при установен флаг DF. Ако се използва формата INS, трябва да бъде посочен операнд-приемник, за да се укаже размерът на обработваните елементи, а DX трябва да се посочи като операнд-източник, съдържащ номера на порта. Смяна на сегмента по подразбиране не се допуска. При форматите INSB (байтове), INSW (думи), INSD (двойни думи, само за i386+) инструкцията определя сама размера на получаваните елементи от низа. Инструкциите се използват обикновено с префикс REP. Преди да започне изпълнението на повторимата инструкция CX трябва да съдържа броя на елементите за получаване. В защитен режим се регистрира грешка от обща защита, когато текущото ниво на привилегии е по-голямо от стойността на флага IOPL.
INT		INTerrupt: генерира софтуерно прекъсване. Операндът (8 b константа, от 0–255) посочва коя процедура за обработка на прекъсване да бъде извикана. Извикването става чрез индексиране на номера на прекъсването в таблицата с векторите на

		прекъсване (IVT), започвайки от сегментен адрес 0 и отместване 0. В реален режим IVT съдържа 4 B указатели към процедурите за обработка на прекъсване, а в защитен режим – 8 B указатели. Когато се изпълнява прекъсване в реален режим, в стека се записват флаговете, CS и IP (в този ред), а флаговете TF и IF се нулират. За разрешаване на прекъсванията може да се използва инструкция STI. За връщане от прекъсване се използва инструкцията IRET.
INTO		INTerrupt on Overflow: генерира прекъсване #4 при установяване в 1 на флага за препълване OF. Действието за MS-DOS no подразбиране за прекъсване #4 е връщане без каквото и да е друго действие. За да може INTO да има някакво действие, в програмата трябва да се дефинира процедура за обработка на прекъсване #4.
INVD		INValid Data cache (само за i486+): изпразва съдържанието на текущия кеш за данни, без запис на промените в паметта. Правилното използване на тази инструкция изисква познания за управление на кеша.
INVLPG		INValidate TLB entry (само за i486+): използва се за задаване на недостоверност на един елемент от буфера TLB, т. е. кеш-паметта, която се използва за съхранение на елементите от таблиците на страниците. Правилното използване на инструкцията изисква познаване на механизма за управление на буфера TLB.
IRET/ IRETD		Interrupt RETurn: връща управлението от процедура за обработка на прекъсване към прекъснатата програма. В реален режим IRET извлича от стека стойностите на IP, CS и флаговете (в този ред) и възстановява изпълнението. При i386+ IRETD трябва да се използва за извличане от стека на 32 b указател на инструкциите при връщане от прекъсване, повикано от 32 b сегмент. При посочване на суфикс F не се генерира епилогов код при завършване на процедура. Използвайте суфикса F при завършване

		на процедури за обслужване на прекъсвания.
Jcondition		conditional Jump: предава управлението към посочения етикет, ако е вярно зададеното за флаговете условие. Условието се тества чрез проверка на състоянието на съответните флагове. Ако условието не е вярно, преход не се прави и програмата продължава изпълнението си със следващата инструкция. При процесори i86–i286 етикетът, посочен като операнд, трябва да бъде за къс преход (в границите –128 до +127 байта от инструкцията след прехода). Процесорите i386+ позволяват близки преходи (–32768 до +32767 B). При i386+ асемблер генерира най-късия възможен преход, освен ако не е явно зададен размерът на прехода. Когато процесорите i386+ са в модел на паметта FLAT, късите преходи са в обхвата от –128 до + 127 B, а близките – от –2 GB до +2 GB. Не съществуват далечни условни преходи. Термините по-ниско/по-високо се отнасят за числа без знак, а по-малко/по-голямо – за числа със знак.
jo	OF=1	Преход при препълване (overflow)
jno	OF=0	Преход при непрепълване (no overflow)
jb/jnae	CF=1	Преход при по-ниско (below)
jae/jnb	CF=0	Преход при по-високо или равно (above or equal)
je/jz	ZF=1	Преход при равно/нула (equal/zero)
jne/jnz	ZF=0	Преход при неравно/не нула (not equal/not zero)
jbe/jna	CF=1 или ZF=1	Преход при по-ниско или равно (below or equal)
ja/jnbe	CF=0 и ZF=0	Преход при по-високо (above)
js	SF=1	Преход при отрицателно (sign)
jns	SF=0	Преход при неотрицателно (no sign)
jp/jpe	PF=1	Преход при четност (parity even)
jnp/jpo	PF=0	Преход при нечетност (parity odd)
jl/jnge	SF!=OF	Преход при по-малко (less than)
jge/jnl	SF=OF	Преход при по-голямо или равно (greater or equal)
jle/jng	ZF=1 или	Преход при по-малко или равно (less than or equal)

	SF!=OF	
jg/jnle	ZF=0 или SF=OF	Преход при по-голямо (greater)
JCXZ/ JECXZ		Jump If CX (ECX) equals Zero: предава управлението към посочения етикет, ако стойността в CX е нула. При i386+ може да се използва JECXZ за преход при ECX=0. Ако стойността на регистъра-брояч не е нула, изпълнението продължава от следващата инструкция. Етикетът, посочен като операнд, трябва да е къс (в границите –128 до +127 B от инструкцията след преход).
JMP		unconditional JuMP: предава безусловно управлението към адреса, посочен от операнда. Преходите могат да бъдат близки (в границите –32768 до +32767 B от инструкцията след прехода), къси (в границите –128 до +127 B) или далечни (в друг кодов сегмент). Ако не е посочена явно отдалеченост на прехода, Асемблер избира най-късия възможен преход. При къси и близки преходи операндът задава нов адрес за IP, а при далечни – нови адреси за IP и CS. Когато процесорите i386+ са в модел на паметта FLAT, късите преходи са в обхвата от –128 до + 127 B, а близките – от –2 GB до +2 GB.
LAHF		Load AH from Flags: прехвърля стойностите на битовете с индекси от 0 до 7 в AH (това са флаговете CF, PF, AF, ZF и SF).
LAR		Load Access Rights byte (само за i286+): зарежда правата за достъп на селектора в посочен регистър. Операндът-източник трябва да бъде регистър или памет, съдържащ селектор. Операндът-приемник трябва да е регистър, получаващ правото на достъп, ако селекторът е валиден и видим на текущото ниво на привилегии. Флагът CF се установява, ако правата за достъп са прехвърлени, в противен случай се нулира.

LDS		LoaD far pointer: определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в DS, а отместването на указателя – в операнда-приемник.
LEA		Load Effective Address: изчислява ефективния адрес (отместването) на операнда-източник (памет) и записва резултата в операнда-приемник. Ако операндът-източник е директен адрес в паметта асемблер кодира инструкцията в по-ефективната форма MOV reg, OFFSET mem
LEAVE		LEAVE high level procedure (само за i286+): унищожава стековата рамка за процедура. Има обратно действие на предишна инструкция ENTER чрез възстановяване на стойностите на SP и BP, които те са имали преди и инициализацията на стековата рамка за процедурата. Инструкцията LEAVE е еквивалентна на MOV SP, BP;POP BP
LES		Load far pointer: определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в ES, а отместването на указателя – в операнда-приемник.
LFS		Load far pointer (само за i386+): определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в FS, а отместването на указателя – в операнда-приемник.
LGDT		Load Global Descriptor Table (само за i286+): зарежда стойност от операнд в регистър на GDT. Използва се само в привилегирован режим.
LGS		Load far pointer (само за i386+): определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в GS, а отместването на указателя – в операнда-приемник.
LIDT		Load Interrupt Descriptor Table (само за i286+):

		зарежда стойност от операнд в регистър на IDT. Използва се само в привилегирован режим.
LLDT		Load Local Descriptor Table (само за i286+): зарежда стойност от операнд в регистър на LDT. Използва се само в привилегирован режим.
LMSW		Load Machine Status Word (само за i286+, защитен режим): зарежда стойност от операнда в думата за състояние на машината (MSW).
LOCK		LOCK the bus: блокира останалите процесори по време на изпълнение на следващата инструкция. Тази инструкция се използва като префикс и се поставя пред инструкция, обръщаща се към област от паметта, до която може да поиска достъп и друг процесор по същото време.
LODS/ LODSB/ LODSW/ LODSD		LOaD String: зарежда акумулатора с низ от паметта. DS:SI трябва да сочи към елемента-източник, даже при зададен операнд. При зареждането на всеки елемент от източника SI се коригира в зависимост от размера на операнда и стойността на флага за посока DF (увеличава се при DF=0 и се намалява при DF=1). Ако се използва инструкцията lods, трябва да се посочи операнд, за да се определи размерът на обработваните елементи. За операнда-източник може да се посочи явно използван сегментен регистър. При използване на lodsb (байтове), lodsw (думи) или lodsw (двойни думи, само за i386+), инструкцията сама определя размера на обработваните елементи и дали елементът ще бъде зареден в AL, AX или EAX. Инструкциите не се използват с префикси за повторение, защото е безсмислено да се зареждат последователни стойности от паметта в регистър.
LOOP/ LOOPW/ LOOPD		LOOP until CX=0: осигурява циклично изпълнение в обхвата на посочен етикет. LOOP изважда 1 от CX (без да променя флаговете) и ако резултатът не е 0, предава управлението към адреса, зададен с операнда. При i386+ LOOP използва 16 b CX за 16 b

		режим и 32 b ECX за 32 b режим. Това действие по подразбиране може да бъде отменено с LOOPW (за CX) или LOOPD (за ECX). Ако стойността на CX след изваждането на 1 е нула, изпълнението продължава със следващата инструкция. Преди влизане в цикъла в CX трябва да се запише максималният брой на повторения. Операндът трябва да посочва към етикет (в границите –128 до +127 B от инструкцията след LOOP).
LOOPE/ LOOPZ/ LOOPEW/ LOOPZW/ LOOPED/ LOOPZD		LOOP while Equal (Zero): осигурява циклично изпълнение в обхвата на посочен етикет при изпълнени условия ZF=1 и CX>0. При i386+ се използва 16 b CX за 16 b режим и 32 b ECX за 32 b режим. Това действие по подразбиране може да бъде отменено със суфикс W (за CX) или D (за ECX). Инструкцията изважда 1 от CX (без да променя флаговете) и ако резултатът не е 0 и флагът ZF е установен в 1 от предишна инструкция (например CMP), предава управлението към адреса, зададен с операнда. Ако условията не са верни, изпълнението продължава със следващата инструкция. Преди влизане в цикъла в CX трябва да се запише максималният брой на повторения.
LOOPNE/ LOOPNZ/ LOOPNEW / LOOPNZW / LOOPNED/ LOOPNZD		LOOP while Not Equal (Not Zero): осигурява циклично изпълнение в обхвата на посочен етикет при изпълнени условия ZF=0 и CX>0. При i386+ се използва 16 b CX за 16 b режим и 32 b ECX за 32 b режим. Това действие по подразбиране може да бъде отменено със суфикс W (за CX) или D (за ECX). Инструкцията изважда 1 от CX (без да променя флаговете) и ако резултатът не е 0 и флагът ZF е установен в 0 от предишна инструкция (например CMP), предава управлението към адреса, зададен с операнда. Ако условията не са верни, изпълнението продължава със следващата инструкция. Преди влизане в цикъла в CX трябва да се запише максималният брой на повторения.
LSL		Load Segment Limit (само за i286+): зарежда сегментния лимит на селектор в посочен регистър. Операндът-източник трябва да бъде регистър или

		памет и да съдържа селектор. Операндът-приемник трябва да бъде регистър и получава сегментния лимит, ако селекторът е валиден и видим на текущото ниво на привилегии. Флагът ZF се установява, ако сегментният лимит се надвиши, в противен случай се нулира.
LSS		Load far pointer (само за i386+): определя и съхранява далечен указател, определен от операнда-източник (памет). Сегментният адрес на указателя се записва в SS, а отместването на указателя – в операнда-приемник.
LTR		Load Task Register (само за i286+, в защитен режим): зарежда стойност от посочения операнд в регистъра на текущата задача. Използва се само на привилегировано ниво.
MOV		MOVe data: копира стойността на операнда-източник в операнда-приемник. Ако операндът-приемник е SS, се забраняват прекъсванията, докато се изпълни следващата инструкция (освен за старите версии на i86/88). Само за i386+: копира стойност от специален регистър в или от 32 b общ регистър. Специалните регистри включват: управляващите регистри CR0, CR2, CR3; регистрите за настройка DR0, DR1, DR2, DR3, DR6 и DR7; тестовите регистри TR6 и TR7. За i486+ са достъпни още и тестовите регистри TR3, TR4 и TR5.
MOVS/ MOVSB/ MOVSW/ MOVSD/		MOVe String: копира низ от една област на паметта в друга. DS:SI трябва да сочи към низа-източник, а ES:DI – към низа-приемник, даже ако са зададени операнди. При всеки копиран елемент SI и DI се коригират в зависимост от размера на операндите и стойността на флага за посока DF (увеличават се при DF=0 и се намаляват при DF=1). Ако се използва инструкцията MOVS, трябва да се посочат операнди, за да се определи размерът на обработваните елементи. За операнда-източник може

		да се посочи явно използван сегментен регистър. При използване на MOVSB (байтове), MOVSW (думи) или MOVSD (двойни думи, само за i386+), инструкцията сама определя размера на обработваните елементи. Инструкциите се използват обикновено с префикси за повторение REP.
MOVSX		MOVe with Sign eXtended (само за i386+): копира и разширява знака на стойността на операнда-източник в операнда-приемник. Използва се за копиране на 8 b или 16 b число със знак в по-голям 16 или 32 b регистър.
MOVZX		MOVe with Zero eXtended (само за i386+): копира и разширява с нули стойността на операнда-източник в операнда-приемник. Използва се за копиране на 8 b или 16 b число без знак в по-голям 16 или 32 b регистър.
MUL		MULtiplication unsigned: умножава операнд-приемник по подразбиране с посочения операнд-източник. Двата операнда се разглеждат като числа без знак. Ако е посочен само един операнд с дължина 16 B, приемникът по подразбиране е в AX, а произведението се записва в двойката регистри DX:AX. Ако е посочен само един операнд с дължина 8 B, приемникът по подразбиране е в AL, а произведението се записва в AX. При i386+, ако източникът е EAX, то произведението се записва в двойката регистри EDX:EAX. Флаговете CF и OF се установяват.
NEG		NEGate: замества стойността на операнда с допълнението й до 2. Това се извършва чрез изваждане на операнда от 0. Ако операндът е 0, нулира се флагът CF, в противен случай се установява в 1. Ако операндът съдържа минималното възможно отрицателно число (–128 за 8 b операнди, или –32768 за 16 B), стойността не се променя, но се установяват флаговете OF и CF.
NOP		No OPeralion: не предизвиква никакво действие. Използва се за времезакъснения или адресни

		изравнявания.
NOT		One’s complement: инвертира всеки бит на операнда.
OR		Inclusive OR: изпълнява поразредна логическа операция OR над операндите източник и приемник и записва резултата в операнда-приемник. Всеки бит на резултата е равен на 1, ако поне един от съответните битовете на двата операнда са 1, в противен случай резултатът е 0.
OUT		OUTput data to port: прехвърля байт или дума (или двойна дума за i386+) към порт от акумулатора. Адресът на порта се посочва от операнда-приемник, който може да бъде DX или 8 b константа. Константи могат да се използват само за номера на портове, по-малки от 255, като за по-големи номера се използва съдържанието на DX. В защитен режим се регистрира грешка от обща защита, когато текущото ниво на привилегии е по-голямо от стойността на флага IOPL.
OUTS/ OUTSB/ OUTSW/ OUTSD		OUTput String data to port (само за i186+): изпраща низ към порт. Низът се счита за източник и трябва да бъде указан от DS:SI (даже при посочен операнд). Изходният порт се задава в DX. При изпращането на всеки елемент от низа SI се коригира в зависимост от размера на операнда и стойността на флага за посока DF (SI се увеличава при DF=0 или се намалява при DF=1). Ако се използва формата OUTS, трябва да бъде посочен операнд-източник, за да се укаже размерът на изпращаните елементи, а DX трябва да се посочи като операнд-източник, съдържащ номера на порта. Смяна на сегмента по подразбиране се допуска. При форматите OUTSB (байтове), OUTSW (думи), OUTSD (двойни думи, само за i386+) инструкцията определя сама размера на изпращаните елементи от низа. Инструкциите се използват обикновено с префикс REP. Преди да започне изпълнението на повторимата

		инструкция CX трябва да съдържа броя на елементите за изпращане. В защитен режим се регистрира грешка от обща защита, когато текущото ниво на привилегии е по-голямо от стойността на флага IOPL.
POP		POP data from stack: извлича стойност от върха на стека в операнда-приемник. Стойността от адрес SS:SP се копира в операнда-приемник и SP се увеличава с 2. Операндът-приемник може да бъде памет, общ 16 b регистър или произволен сегментен регистър (с изключение на CS). За извличане на стойност от стека в CS се използва инструкция RET. При i386+ от стека може да се извлече 32 b стойност, ако се посочи 32 b операнд (тогава ESP се увеличава с 4).
POPA/ POPAD		POP All registers from stack (само за i186+): извлича 16 B от върха на стека и ги записва в осемте регистъра с общо предназначение в следния ред: Dl, SI, BP, SP, BX, DX, CX и AX. Стойността за регистъра SP не се копира в SP. Инструкцията POPA винаги извлича от стека стойности за 16 b регистри. За извличане на стойности от стека за 32 b регистри при i386+ се използва инструкцията POPAD.
POPF/ POPFD		POP Flags from stack: извлича стойността от върха на стека в регистъра с флаговете. POPF извлича стойност само за 16 b регистър на флаговете. За извличане на стойност от стека за 32 b регистър на флаговете при i386+ се използва POPFD.
PUSH/ PUSHW/ PUSHD		PUSH data onto stack: записва операнда-източник в стека. SP се намалява с 2 и стойността на източника се копира на адрес SS:SP. Операндът може да бъде памет, общ 16 b регистър или произволен сегментен регистър. При i186+ операндът може да бъде и константа. При i386+ в стека могат да се записват 32 b стойности, ако се посочи 32 b операнд (тогава ESP се намалява с 4). При 86/88 инструкцията push sp съхранява стойността на SP след записа в стека, а при i186+ – стойността на SP преди записа в стека. Инструкциите PUSHW и PUSHD се използват

		съответно за запис в стека на дума и двойна дума.
PUSHA/ PUSHAD		PUSH All registers onto stack (само за i186+): записва в стека стойностите на осемте регистъра с общо предназначение в следния ред: AX. CX, DX, BX. SP, BP, SI, DI. Стойността, записана за регистър SP, е тази преди изпълнение на инструкцията. PUSHA винаги записва в стека 16 b регистри. За запис на 32 b регистри в стека при i386+ се използва PUSHAD.
PUSHF/ PUSHFD		PUSH Flags onto stack: записва в стека стойността на регистъра с флаговете. PUSHF записва стойност само за 16 b регистър на флаговете. За запис на 32 b регистър на флаговете при i386+ се използва PUSHFD.
RCL/ RCR/ ROL/ ROR		Rotate: измества циклично битовете на операнда-приемник на толкова разреда, колкото е стойността на операнда-източник. RCL и ROL ротират битовете вляво, а RCR и ROR – вдясно. ROL и ROR изместват битовете на операнда циклично на зададения брой разреди. При всяко изместване най-левият или най-десният бит се копира в флага CF и се ротира. RCL и RCR ротират през флага за пренос CF. CF всъщност става разширение на операнда, така че при 8 b операнди се прави 9 b ротация, а за 16 b операнди –17 b ротация. При i86/88 операндът-източник може да е CL или 1. При i186+ операндът-източник може да бъде CL или 8 b константа. Флагът за препълване OF се променя само за 1-битови ротации, за многобитови той остава недефиниран.
RDMSR		ReaD from Model Specific Register (само за Pentium): четене от моделен регистър.
REP		REPeat prefix: повторя операции с низ толкова пъти, колкото е стойността в CX Отначало стойността на CX се сравнява с 0 и ако CX=0, изпълнението продължава със следващата инструкция. В противен

		случай CX се намалява с 1, изпълнява се инструкцията с низа и цикълът продължава. REP се използва с MOVS и STOS, а също така и с INS и OUTS за i186+. За всички процесори с изключение на i386+ комбинирането на префикс за повторение с явно задаване на сегментен регистър може да причини грешки при поява на прекъсване.
REPE/ REPZ/ REPNE/ REPNZ		REPeat conditional: повторя операция с низ, докато посочено условие е вярно и стойността на CX>0. REPE и REPZ (синоними) се изпълняват многократно, ако флагът ZF= 1, а REPNE и REPNZ (също синоними) – докато флагът ZF=0. Префиксите за условно повторение могат да се използват само с инструкциите CMPS и SCAS, т. к. само те променят флага ZF. Преди изпълнението на инструкцията в CX трябва да се запише максимално допустимият брой повторения. Отначало стойността на CX се сравнява с 0 и ако CX=0, изпълнението продължава със следващата инструкция. В противен случай CX се намалява с 1, проверява се флагът ZF и в зависимост от стойността му се изпълнява инструкцията с низа, като цикълът продължава. За всички процесори с изключение на i386+ комбинирането на префикс за повторение с явно задаване на сегментен регистър може да причини грешки при поява на прекъсване.
RET/ RETN/ RETF		RETurn from procedure: връща от процедура чрез предаване на управлението към адреса, извлечен от върха на стека. Може да бъде посочен операнд (константа), указващ брой на допълнителните байтове за освобождаване. Константата обикновено се използва за коригиране на стека при запис на аргументи преди обръщане към процедурата. Типът на връщането (близко или далечно) се определя от типа на процедурата, в която се среща RET. За посочване на близко връщане може да се използва RETN, а за далечно – RETF. При близко връщане от стека се извлича дума в IP, а при далечно се извличат първо една дума в IP и след това още една дума в CS. След връщането към SP се прибавя число (брой байтове), равно на стойността

		на операнда (ако е посочен).
RSM		Resume from System Menagement mode (само за Pentium): връщане от системен режим на управление.
SAHF		Store AH Into Flags: копира стойността на AH в битове с индекси от 0 до 7 на флаговия регистър. С това се променят флаговете CF, PF, AF, ZF и SF, но не се променят флаговете TF, IF, DF и OF.
SAL/ SAR/ SHL/ SHR		SHift: измества битовете на операнда-приемник на толкова разреда, колкото е стойността на операнда-източник. SAL и SHL изместват битовете вляво, а SAR и SHR – вдясно. При SHL, SAL И SHR изместеният бит от края на операнда се копира във флага CF, а най-левият или най-десният бит се попълва с 0. При SAR най-десният изместен бит се копира във флага CF, а най-левият бит запазва предишната си стойност, като по този начин се съхранява знакът на операнда. SAL и SHL са синоними. При i86/88 операндът-източник може да е CL или 1. При i186+ операндът-източник може да бъде CL или 8 b константа. Флагът за препълване OF се променя само за 1-битови измествания, за многобитови той остава недефиниран
SBB		SuBtract with Borrow: добавя стойността на флага CF към втория операнд и след това изважда получената стойност от първия операнд. Резултатът се записва в първия операнд. Използва се за изваждане на числа с дължина няколко байта (или няколко думи).
SCAS/ SCASB/ SCASW/ SCASD/		SCAn String: сканира низ, търсейки стойност, която е зададена в акумулаторния регистър. Сканираният низ се счита за приемник. ES:DI трябва да сочи този низ, даже при посочен операнд. Всеки елемент от низа се изважда от стойността на акумулатора и в съответствие с резултата се обновяват флаговете (без съхраняване на резултата). DI се коригира в

		зависимост от размера на операндите и стойността на флага DF (DI се увеличава при DF=0 или се намалява при DF=1). Ако се използва формата SCAS, трябва да се посочи операнд за определяне на размера на обработваните елементи. Не се допуска смяна на сегмента по подразбиране. При формите SCASB (байтове), SCASW (думи) и SCAD (двойни думи, само за i386+) инструкцията сама определя размера на обработваните елементи и дали търсеният елемент се намира в AL, AX или EAX. Инструкциите се използват обикновено с префиксите за повторение. REPNE (или REPNZ) се използва за намиране на първия елемент в низ, който отговаря на стойността в акумулаторния регистър, а REPE (или REPZ) – за откриване на първото несъответствие. Преди сканирането в CX трябва да се запише максималният брой елементи за сканиране. След REPNE SCAS флагът ZF се нулира, ако низът не съдържа търсената стойност, а след REPE SCAS флагът ZF се установява в 1, ако низът съдържа само търсената стойност. След завършване на инструкцията ES:DI сочи към елемента след (при DF=0) или преди (при DF=1) съответствието (или несъответствието). Ако CX се намали до 0, ES:DI сочи към елемента след или преди последното сравнение. Флагът ZF се управлява от резултата на последното сравнение, а не от стойността на CX.
SETcondition		SET on condition (само за i386+): записва в байт, посочен като операнд стойност 1 при изпълнено или 0 при неизпълнено условие. Условието се проверява в зависимост от стойностите на флаговете. Използва се за условно установяване на булеви флагове.
seto	OF=1	Установяване при препълване (overflow)
setno	OF=0	Установяване при непрепълване (no overflow)
setb/setnae	CF=1	Установяване при по-ниско (below)
setae/setnb	CF=0	Установяване при по-високо или равно (above or

		equal)
sete/setz	ZF=1	Установяване при равно/нула (equal/zero)
setne/setnz	ZF=0	Установяване при неравно/не нула (not equal/not zero)
setbe/setna	CF=1 или ZF=1	Установяване при по-ниско или равно (below or equal)
seta/setnbe	CF=0 и ZF=0	Установяване при по-високо (above)
sets	SF=1	Установяване при отрицателно (sign)
setns	SF=0	Установяване при неотрицателно (no sign)
setp/setpe	PF=1	Установяване при четност (parity even)
setnp/setpo	PF=0	Установяване при нечетност (parity odd)
setl/setnge	SF!=OF	Установяване при по-малко (less than)
setge/setnl	SF=OF	Установяване при по-голямо или равно (greater or equal)
setle/setng	ZF=1 или SF!=OF	Установяване при по-малко или равно (less than or equal)
setg/setnle	ZF=0 или SF=OF	Установяване при по-голямо (greater than)
SGDT/ SIDT/ SLDT		Store descriptor table (само за i286+): зарежда стойност от регистър на дескрипторна таблица (GDT, LDT или IDT) в посочения операнд. Използва се само в привилегирован режим.
SHLD/ SHRD		Shift with Double precision (само за i386+): измества битовете на втория операнд в първия операнд. Броят на изместените битове се определя от третия операнд. SHLD измества първия операнд вляво с брой разреди, определени от брояча. Освободените от изместването разреди се попълват с най-старшите битове на втория операнд. SHRD измества първия операнд вдясно с брой разреди, определени от брояча. Освободените от изместването разреди се попълват с най-младшите битове на втория операнд. Броячът трябва да бъде зададен в CL или с 8 b константа. Ако броячът има стойност, по-голяма от 31, стойността му се коригира, като се използва остатъкът на стойността по модул 31.
SMSW		Store Machine Status Word (само за i286+): съхранява

		думата за състояние на машината (MSW) в посочен операнд в паметта. Използва се само в защитен режим.
STC		SeT Carry flag: установява в 1 флага за пренос CF.
STD		SeT Direction flag: установява в 1 флага за посока DF. Всички следващи операции над низове се изпълняват в посока от високите към ниските адреси.
STI		SeT Interrupt flag: установява е 1 флага за прекъсване IF. Когато IF=1, се приемат маскируемите прекъсвания. Ако прекъсванията са забранени с предишна инструкция CLI, чакащи прекъсвания не се изпълняват незабавно, а след изпълнението на следваща инструкция STI.
STOS/ STOSB/ STOSW/ STOSD		STOre String data: записва стойността на акумулатора в низ. Низът се счита за приемник и трябва да бъде указан от ES:DI (даже при посочен операнд). При зареждане на всеки елемент от низа DI се коригира в зависимост от размера на операнда и стойността на флага за посока DF – DI се увеличава при DF=0 или се намалява при DF=1. Ако се използва формата STOS, трябва да бъде посочен операнд-приемник, за да се укаже размерът на обработваните елементи. Смяна на сегмента по подразбиране не се допуска. При форматите STOSB (байтове), STOSW (думи), STOSD (двойни думи, само за i386+) инструкцията определя сама размера на елементите за обработка и дали елементът постъпва от AL, AX или EAX. Инструкциите се използват обикновено с префикс REP за попълване на низ с една и съща стойност. Преди да започне изпълнението на повторимата инструкция CX трябва да съдържа броя на елементите за попълване.
STR		Store Task Register (само за i286+): съхранява

		регистъра на текущата задача в посочения операнд. Тази инструкция се използва само на привилегировано ниво.
SUB		SUBtract: изважда операнда-източник от операнда-приемник и съхранява резултата в операнда-приемник.
TEST		TEST operands: тества посочени битове на операнда и установява флаговете за следващ условен преход или инструкция SET. Единият от операндите съдържа тестваната стойност, а другият – маска от битове, задаваща битовете за тестване. TEST изпълнява поразредно операция AND над двата операнда, като променя флаговете в зависимост от резултата, но не записва резултата в операнда-приемник.
VERR/ VERW		VERIfy Read or Write (само за i286+, защитен режим): проверява дали посочен сегментен селектор е валиден и може да бъде прочетен (VERR) или записан (VERW) при текущото ниво на привилегии. Ако сегментният селектор е валиден, флагът ZF се установява в 1, в противен случай се нулира.
WAIT		WAIT for coprocessor: блокира изпълнението на процесора, докато процесорът получи сигнал, че копроцесорът е завършил едновременна операция. Използва се, когато трябва да се предотврати промяна на стойност в паметта от копроцесорна инструкция, която се променя едновременно с това и от процесорна инструкция. Действието е еквивалентно на копроцесорна инструкция FWAIT.
WBINVD		Write Back and INValidate Data cache (само за i486+): изпразва съдържанието на текущия кеш за данни след запис на промени в паметта. За правилното използване на тази инструкция се изискват познания за управлението на кеш-паметта. Използва се за системно програмиране.
WRMSR		Write to Model Specific Register (само за Pentium): запис в моделен регистър.

XADD		eXchange and ADD (само за i486+): събира стойностите на операнда-източник и операнда-приемник и записва резултата в операнда-приемник, като едновременно с това първоначалната стойност на операнда-приемник се премества в източника. Флаговете се установяват в зависимост от резултата.
XCHG		eXCHanGe: разменя стойностите на операнда-източник и операнда-приемник.
XLAT/ XLATB		transLATe: транслира стойност от една кодова система е друга чрез търсене на стойността за транслиране в таблица, съхранена в паметта. Преди изпълнението на инструкцията BX трябва да сочи към таблицата в паметта, а AL трябва да съдържа позицията на транслираната стойност в таблицата (цяло число без знак). След изпълнение на инструкцията AL съдържа стойността от таблицата от посочената позиция. Операнд не е необходим, но може да се посочи за отмяна на сегмента по подразбиране (DS). XLATB е синоним на XLAT.
XOR		eXclusive OR: изпълнява поразредна логическа операция “изключващо ИЛИ” (eXOR) над операндите източник и приемник и записва резултата в операнда-приемник. Всеки бит на резултата е равен на 1, ако съответните’битовете на двата операнда са различни, в противен случай резултатът е 0.

Използвана литература

1. Волфганг Линк, Програмиране на Асемблер, Издателство “Техника”, първо издание, 2002
2. KIP R. IRVINE, Assembly Language for Intel–Based Computers, Third Edition, 1998
3. А. Егоров, Р. Стоянова, Програмиране на асемблерен език за 32-битови микропроцесори Intel, Парафлоу ООД – София, 1997
4. В. И. Юров, Assembler. Учебник для вузов, 2–е изд. – СПб. : Питер, 2004.
5. Peter Abel – IBM PC ASSEMBLY LANGUAGE AND PROGRAMMING, Fourth edition, Prentice Hall, 1997
6. Олифер Мюлер, Асемблер– справочник, ИК “Техника”, 2000
7. David J. Bradley, ASSEMBLY LANGUAGE PROGRAMMING FOR THE IBM PERSONAL COMPUTER, Prentice Hall, Inc., Englewood Cliffs, NJ, 1984
8. Rudolf Marek, Assembly Language For The PC, Computer Press, 2003
9. Зубков С. В., Assembler. Язык неограниченных возможностей, ДМК, 1999
10. П. И. Рудаков, К. Г. Финогенов, Язык ассемблера: уроки программирования, М. : ДИАЛОГ–МИФИ, 2001
11. П. Петров, Програмиране на Асемблер, изд. Техника, 2002
12. http://www.softpanorama.org/Lang/assembler.shtml
13. http://www.programmersheaven.com/zone5/
14. http://webster.cs.ucr.edu/
15. http://asm. shadrinsk.net/books. htm