Технически университет – Варна

Катедра “Компютърни науки и технологии”

Димитър Стоянов Тянев

Жейно Иванов Жейнов

МИКРОПРОЦЕСОРНА ТЕХНИКА И ПРОГРАМИРАНЕ НА АСЕМБЛЕР

2010

УДК 681.325.5

МИКРОПРОЦЕСОРНА ТЕХНИКА И ПРОГРАМИРАНЕ НА АСЕМБЛЕР

Тази книга може да бъде използвана като учебник по дисциплината “Микропроцесорна техника”, включена в учебния план на специалност “Компютърни системи и технологии”. В нея кратко е представена микропроцесорната архитектура i8086(88), структурата на компютърните системи, основани на нея, както и Асемблерният език за програмиране върху нейната система от машинни команди. Представянето на асемблерния език включва командите, операторите, директивите, конструкциите, синтактичните и семантичните правила, както и методите за програмиране в неговата среда. Изложението е съпроводено с множество примери. Представено е и методическо ръководство за самостоятелно създаване, транслиране и експериментиране с програми на Асемблерен език.

Книгата е предназначена за студенти и специалисти, желаещи да изучат тази компютърна архитектура, нейния език, както и спецификата на програмиране с помощта на близкия до машинното ниво език Асемблер, който е основа на съвременните съвместими с нея модели на персонални компютърни системи.

Рецензент доц. д-р инж. Сава Иванов Иванов

Автори: © Димитър Стоянов Тянев
© Жейно Иванов Жейнов

ISBN 978-954-20-0472-1

С Ъ Д Ъ Р Ж А Н И Е

ПРЕДГОВОР ....
ГЛАВА 1. АРХИТЕКТУРА i-16 9
1.1 Апаратна структура на микропроцесор Intel 8086 9
1.2 Организация на адресното пространство 12
1.3 Външен интерфейс на микропроцесора 14
1.4 Организация на обмена 18
1.5 Организация на входно-изходния обмен 21
1.6 Система за прекъсване 22
1.7 Система машинни команди 25
1.8 Методи за адресиране 28
1.9 Даннови формати 35
1.10 Копроцесори 37
1.11 Аритметически копроцесор 38
1.12 Команди на копроцесора 40
1.13 Схеми за поддържане на системата 42
ГЛАВА 2. АСЕМБЛЕРЕН ЕЗИК. НАЧАЛНИ ПОНЯТИЯ 44
2.1 Лексеми 46
2.1.1 Идентификатори 46
2.1.2 Цели числа 47
2.1.3 Символни данни 47
2.2 Изречения 48
2.2.1 Коментари 48
2.2.2 Команди 49
2.2.3 Директиви 50
2.2.4 Обръщения назад и напред 51
2.3 Директиви за описание на данните 52
2.3.1 Директива DB 52
2.3.2 Директива DW 55
2.3.3 Директива DD 56
2.4 Директиви за еквивалентност и присвояване 58
2.5 Изрази 61
2.5.1 Константни изрази 62
2.5.2 Адресни изрази 63
ГЛАВА 3. ПРЕХВЪРЛЯНЕ НА ДАННИ.
АРИТМЕТИЧНИ КОМАНДИ 65
3.1 Обозначаване на операндите на командите 65
3.2 Команди за прехвърляне на данни 65
3.2.1 Команда MOV 65
3.2.2 Оператор за указване на тип (PTR) 67

3.2.3 Команда за размяна XCHG 69
3.3 Команди за събиране и изваждане 69
3.4 Команди за умножение и деление 70
3.5 Промяна на формата на число 72
3.6 Примерни задачи 73
ГЛАВА 4. КОМАНДИ ЗА УПРАВЛЕНИЕ НА ПРЕХОДА. ЦИКЛИ. 77
4.1 Безусловен преход. Оператор SHORT...................... 77
4.2 Косвен преход 79
4.3 Команди за сравнения и за условен преход 80
4.4 Команди за реализация на цикли 84
4.5 Операции Въвеждане / Извеждане 87
4.5.1 Спиране на програмата 87
4.5.2 Въвеждане от клавиатурата 88
4.5.3 Извеждане на екрана 88
4.6 Примери 90
ГЛАВА 5. МАСИВИ. СТРУКТУРИ 95
5.1 Индекси 95
5.2 Променливи с индекс 96
5.2.1 Модификация на адресите 96
5.2.2 Индексиране на операндите 96
5.2.3 Косвени указатели 97
5.2.4 Модификация чрез няколко регистъра 97
5.2.5 Правила за запис на модификациите 98
5.3 Команди LEA и XLAT 100
5.4 Структури 102
5.5 Примерни задачи 108
ГЛАВА 6. БИТОВИ ОПЕРАЦИИ. УПАКОВАНИ ДАННИ 113
6.1 Команди за логически операции 113
6.2 Команди за изместване 115
6.3 Упаковани данни 117
6.4 Множества 119
6.4.1 Машинно представяне на множества 119
6.4.2 Операции върху множества 120
6.5 Записи 121
6.5.1 Деклариране на записи 122
6.5.2 Описание на променливи от тип запис 122
6.5.3 Средства за работа с полетата на записи 123
ГЛАВА 7. ПРОГРАМНИ СЕГМЕНТИ 125
7.1 Сегментиране на адресите 125
7.1.1 Сегментни регистри по подразбиране 127

7.2 Програмни сегменти 128
7.3 Директива ASSUME 132
7.4 Начално зареждане на сегментните регистри 137
7.5 Структура на програмата. Директива INCLUDE 138
ГЛАВА 8. СТЕК **141**
8.1 Стек и сегмент на стека 141
8.2 Стекови команди 141
8.3 Похвати за работа със стека 143
ГЛАВА 9. ПРОЦЕДУРИ **145**
9.1 Дълги преходи 145
9.2 Подпрограми 147
9.2.1 Къде се разполагат подпрограмите? 147
9.2.1 Как се оформят подпрограмите? 148
9.2.3 Обръщение към процедура и връщане от нея 149
9.2.4 Други варианти на командата CALL 150
9.3 Предаване на параметри чрез регистрите 151
9.3.1 Предаване на стойности 152
9.3.2 Предаване на указатели 153
9.3.3 Съхранение на съдържанието на регистрите 154
9.3.4 Предаване на параметри с по-сложен тип 154
9.4 Предаване на параметри чрез стека 155
9.5 Локални данни за процедурите 158
ГЛАВА 10. СИМВОЛНИ НИЗОВЕ **162**
10.1 Команди за обработка на низове 162
10.1.1 Команди за сравняване на символни низове 162
10.1.2 Префикси за повторение 163
10.1.3 Други низови команди 166
10.1.4 Команди за зареждане на адресни двойки рег...... 169
10.2 Символни низове с променлива дължина 170
ГЛАВА 11. МАКРОСРЕДСТВА **173**
11.1 Макроезик 173
11.2 Блокове за повторение 173
11.3 Макроси 179
11.3.1 Макроопределения 179
11.3.2 Макрокоманди 180
11.3.3 Макрозаместване и макроразширение 180
11.3.4 Примери, използващи макроси 181
11.3.5 Макроси и процедури 184
11.3.6 Определяне на макрос чрез макрос 184
11.3.7 Директива LOCAL 185
11.3.8 Директива EXITM 187

11.3.9 Преопределяне и отманя на макроси 187
11.4 Условно асемблиране 188
11.4.1 Директиви IF и IFE 188
11.4.2 Оператори за отношение. Логически оператори ... 190
11.4.3 Директиви IFIDN, IFDIF, IFB и IFNB 192
ГЛАВА 12. МНОГОМОДУЛНИ ПРОГРАМИ 195
12.1 Работа със системата MASM 196
12.2 Модули. Външни и общи имена 198
12.2.1 Структура на модулите. Локализация на имената 198
12.2.2 Външни имена. Директиви EXTRN и PUBLIC 198
12.2.3 Сегментиране на външни имена 200
12.2.4 Достъп до външните имена 202
12.2.5 Пример за многомодулна програма 203
ГЛАВА 13. ВЪВЕЖДАНЕ-ИЗВЕЖДАНЕ. ПРЕКЪСВАНЕ 206
13.1 Команди за вход-изход 206
13.2 Прекъсвания. Функции на DOS 207
13.2.2 Функции на DOS 208
13.2.3 Някои функции на прекъсване 21h 210
13.3 Операции вход-изход 213
13.3.1 Съхраняване и подключване на операциите В/И . 213
ГЛАВА 14. ДОПЪЛНЕНИЯ 216
14.1 Двоично-десетични числа 216
14.1.1 Събиране на двоично-десетични числа 217
14.1.2 Изваждане на двоично-десетични числа 219
14.1.3 Умножение и деление на 2/10-чни числа 220
14.2 Допълнителни команди 221
14.3 Допълнителни оператори 224
14.4 Директиви за управление на листинга 226
14.5 Директиви за управление работата на Асемблер ... 228
14.5.1 Директива %OUT 228
14.5.2 Допълнителни IF-директиви 228
14.5.3 Условно генериране на грешки 229
14.6 Допълнителни директиви 231
14.6.1 Групи от сегменти 231
14.6.2 Промени в брояча за разполагане 232
14.6.3 Други директиви 233
Разработка на асемблерни програми с Турбо Дебъгер .. 235
ПРИЛОЖЕНИЕ А (Команди и признаци) 260
ПРИЛОЖЕНИЕ Б (Текст на файл IOPROC.ASM и IO.ASM) 270
Литература 280

ПРЕДГОВОР

Дадената книга съдържа необходимата информация, за да могат студентите от специалност "Компютърни системи и технологии" да извършат гладък преход от предходните към следващите учебни дисциплини. Имайки предвид, че познанието е неделимо, авторите не възприемат изложеното тук като нещо самостоятелно, а като тематика, която въпреки своята специфика, е разглеждана в светлината на вече познатото на студента. В тази връзка изложението разчита на тези познания и е построено върху това разбиране. Освен това, със своята техническа и софтуерна конкретика, изложеното тук познание е един от необходимите примери, свързващи изучената вече теория с практиката.

Въпреки, че асемблерните езици се използват относително по-рядко и в специфични условия на практиката, изучаването им е необходима част от подготовката на професионалните компютърни специалисти. Програмирането с помощта на асемблерни езици е умение, което свързва хардуерният специалист с проблемите на софтуерния, както и обратно – програмистът е професионалист, когато разбира принципите на строежа и функционирането на средството, с което работи.

В настоящия момент процесорите, които фирмата *Intel* наложи в масовите персонални компютърни системи, са изминали дълъг път на еволюция спрямо времето на 16-битовата архитектура, която разглеждаме тук. Това обаче не ни притеснява, защото виждаме запазващата се приемственост в еволюцията, която е главната особеност на всички поколения процесори до този момент. Приемствеността се отнася най-вече за нивото на системата машинни команди. Тя се изразява в това, че програми, които са написани в миналото за по-младшите модели на процесорите, могат да бъдат изпълнявани върху новите модели. За базова е приета системата машинни команди именно на микропроцесорната фамилия i8086.

Езикът Асемблер представлява символна форма на машинния език на цифровия процесор. Тъй като всеки процесор има свой собствен машинен език, това означава, че Асемблерният език не бива да се разбира като универсален, машинно независим или общ език. Такъв език има всеки отделен модел процесор, а за някои от тях може да съществуват и по няколко версии, разработка на различни фирми. В този смисъл асемблерните езици се определят като езици за програмиране на ниско ниво, ниво най-близко стоящо до вътрешния машинен език на всеки процесор. Ето защо често се говори за програмиране *в кода на машината*.

В тази книга е разгледан разработеният от фирма *Microsoft* асемблерен език, известен като MASM. Имайки предвид, че читателят е вече запознат с програмирането на език от високо ниво, разказът тук за

похватите при програмиране на асемблерен език е основан върху познатите му вече структури от данни (скаларни, масиви, структури и пр.), както и познатите му алгоритмични структури, като линейни, разклонени, циклични и пр. Върху такива задачи в текста са представени примери и множество техни реализации на асемблерен език, което позволява задълбочено изучаване на симбиозата между хардуер и софтуер. Специално внимание е отделено на подпрограмната техника, която е в основата на структурното програмиране.

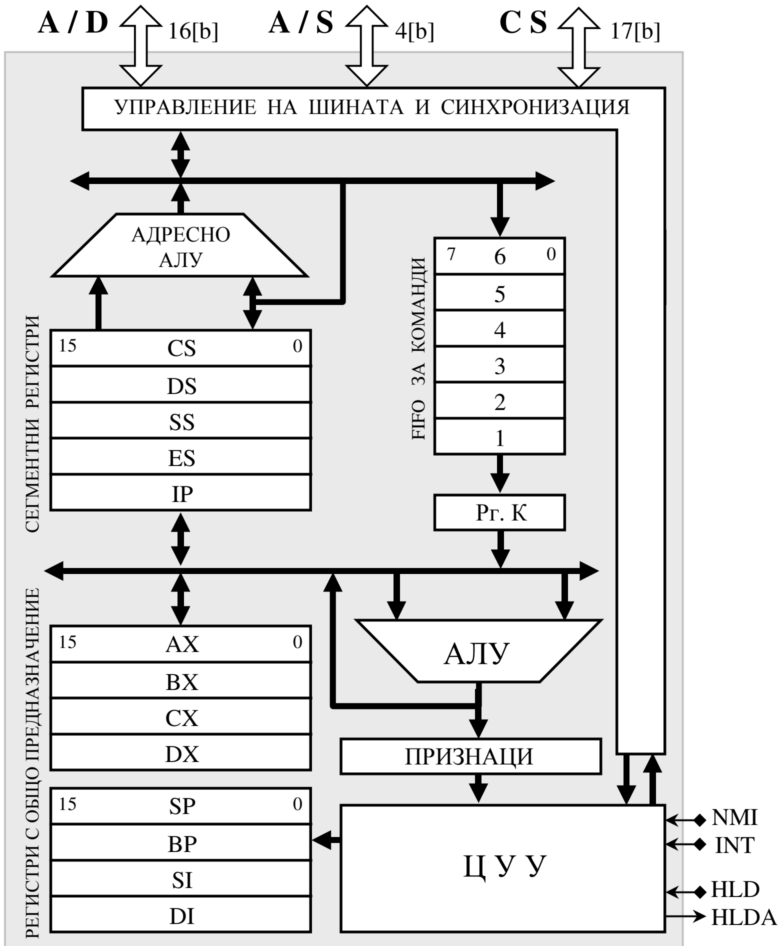
Изложението съдържа основните елементи на езика и не претендира за всеобхватност на проблемите. В същото време авторите са убедени, че изложеното е напълно достатъчно за успешно самостоятелно усвояване на асемблерния език за произволен процесор.

ГЛАВА 1

АРХИТЕКТУРА i-16

1.1 Апаратна структура на микропроцесор Intel 8086

Структурата на микропроцесора, чиято система машинни команди е в основата на всички следващи поколения микропроцесори, е представена обобщено на фигура 1.1.1.



Фиг. 1.1.1 Обобщена структура на микропроцесор I8086

Както може да се види, структурата съдържа две основни и едно спомагателно устройства: АЛУ, ЦУУ и устройство за управление и синхронизация на системната шина (УУССШ).

1. Устройство за преработка на данни (АЛУ)

Устройството за преработка на данните разполага с 8 на брой 16-битови регистри с “общо” предназначение, както и регистър на признаците, който съвместява и регистъра на състоянието, поради което фирменото му наименование е регистър на флаговете. Регистърът на флаговете има следната структура:

1	1	1	1	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

където означенията имат следния смисъл:

DF - Флаг за посока на В/И трансфер
IF - Маска за прекъсване
TF - Флаг за трасиране

Флаг за пренос - CF
Флаг на паритета - PF
Флаг на междинния пренос - AF
Флаг за нулев резултат - ZF
Флаг на знака - SF
Флаг за препълване - OF

Най-съществено значение за програмиста имат следните регистри:

Accumulator - AX	(Акумулатор)	Source index - SI	(Индекс на източник)
Base - BX	(База)	Destination index - DI	(Индекс на приемник)
Counter - CX	(Брояч)	Base pointer - BP	(Указател на базата)
Data - DX	(Данни)	Stack pointer - SP	(Указател на стека)

Всяка от двете половини (старша (H) и младша (L)) на регистрите от първата група (A,B,C,D) може да се използва самостоятелно както за приемник, така и като източник на данни (операнди) в отделните команди. Означенията в тези случаи са: (AH,AL); (BH,BL); (CH,CL); (DH,DL). При необходимост всеки от тези 4 регистри може да се използва за данни с 16-битов формат. Въпреки че тези регистри са предназначени за обща употреба, те нямат пълната свобода в този смисъл, тъй като при изпълнение на някои команди, предназначението им е фиксирано от Конструктора и Програмистът следва да се съобразява с това, което ще поясним по-късно.

Втората група регистри има отношение преди всичко към адреса на данните. Регистър SI е предназначен за отместването на адреса на елемент от масив в паметта при четенето му, а регистър DI – за отместването на адреса на елемент от масив в паметта при записа му.

В устройството за преработка се изпълняват всички аритметически операции върху числа с фиксирана запетая. Освен тях се изпълняват още всички логически операции и измествания. Регистрите на АЛУ се използват още и при изпълнение на операции с паметта.

При изпълнение на операция в АЛУ, освен резултат, се формира и ново съдържание на регистъра на признаците.

Флаг CF записва стойността на преноса (заема) от старшия разряд на обработвания формат, но се използва и при измествания.

Флаг PF допълва броя на единиците в обработвания формат до четен.

Флаг AF записва междутедрадния пренос и се използва при определяне на 2/10-ичната корекция.

Флаг ZF е признак за нулево съдържание на обработвания формат.

Флаг SF дублира старшия (знаков) бит на обработвания формат.

Флаг OF съдържа признака за препълване при събиране на числа в допълнителен код. Стойността му се формира чрез логическата функция за разпознаване на препълване чрез преносите [1].

Останалите флагове се използват за управление на състоянието на процесора. Флаг DF се използва за управление направлението на трансфера при входно-изходни операции.

Флаг IF се използва за управление на достъпа на заявките за прекъсване.

Флаг TF се използва за включване на постъпковия режим за изпълнение на програмите.

2. **Централно управляващо устройство** (ЦУУ)

ЦУУ реализира общата организация за съвместно функциониране на всички устройства в структурата на микропроцесора. То приема всички външни входни сигнали и генерира необходимите изходни управляващи сигнали. Освен това реализира първичния организационен алгоритъм на командния цикъл, при което управлява обмена и последователност-та за изпълнение на машинните команди.

Съществен елемент в управлението е реализацията на паралелна и асинхронна работа на АЛУ и УУССШ. Всеки път, когато системната шина на процесора се окаже свободна, УУССШ я използва за да извлече от ОП поредната последователност от команди и да я достави изпреварващо във FIFO буфера за команди. Така това устройство има за задача да поддържа във времето опашката запълнена. Единствената цел на това изпреварващо във времето извличане на команди е повишаване на производителността.

Ако при изпълнение на текущата команда АЛУ има необходимост да се обърне към ОП, то изпреварващото извличане се прекъсва и се изпълнява цикъл на четене или на запис на данни в паметта.

3. Устройство за управление и синхронизация на системната шина

Задачата на това устройство е да осигури обмена на информация между микропроцесора и външно разположените приемници и източ-ници (ОЗУ, Контролери на ВУ за В/И и пр.).

В състава на това устройство влизат групата на сегментните регистри, програмният брояч, FIFO буфера за команди и всички логически възли на адресното АЛУ.

Сегментните регистри съхраняват началните (базовите) адреси на структурните единици на всяка програма в паметта. Изпълнението на

едно програмно задание се нуждае от 4 структурни единици в оперативната памет. Базовите регистри са:

- Регистър CS (*Code Segment*). В кодовия сегмент на ОП се съхранява програмата. Съдържайки началния адрес, сегментният регистър CS определя адреса на всяка команда;
- Регистър DS (*Data Segment*). В данновия сегмент на ОП се съхраняват данните на програмата. Тяхното адресиране се определя от базовия адрес в този регистър;
- Регистър SS (*Stack Segment*). В тази област на ОП, наречена програмен стек; се съхраняват програмните параметри и адресите за връщане от прекъсване;
- Регистър ES (*Extra Segment*). Този регистър определя мястото в ОП на допълнителна даннова област, в която се съхранява специална информация.

Регистърът, наречен програмен брояч - IP (*Instruction Pointer*), съдържа следващия 16 битов адрес. Следващият адрес се получава по силата на командния цикъл при извличане на текущата команда. Това е абсолютен адрес спрямо началото на кодовия сегмент. Представлява адрес на следващата в текста на програмата команда, който текущата команда може да запази, но може и да промени. Шесте 8 битови регистъра образуват бърза буферна памет и съдържат опашката от чакащи изпълнението си машинни команди.

Всяка порция данни, постъпваща в процесора по шината A/D, може да бъде разпределена и фиксирана във вътрешността му. Аналогично, в обратната посока, такава информация може да бъде изпратена до всяко външно устройство.

1.2 Организация на адресното пространство

Тъй като дължината на адреса е 20 бита, то обемът на адресното пространство е 2^{20}=1[MB]=1048576[B]. Адресите ще бъдат записвани като цели числа без знак най-често в 16-чна бройна система в интервала [00000, FFFFF].

Структурните параметри на процесора и най-вече в лицето на 16 битовата дължина на разрядната мрежа, която е различна от тази на адреса, са наложили структуриране на адресното пространство. Основната му структурна единица е наречена сегмент, който има обем 2^{16}=64[KB], определен от 16 битовата дължина на адреса, възможен за записване в командата. В командата е възможен за запис адрес в интервала [0000, FFFF], който е абсолютен адрес в рамките на сегмента и по тази причина се нарича отместване. Общият брой последователни сегменти в пълния обем на адресното пространство е 16. За да бъде по-гъвкава организацията на адресното пространство се въвежда възможността за променлива в рамките на възможния интервал обем на сегмента. Това се постига чрез относителната функционална връзка:

$$A_{ef} = (SR).2^4 + D \quad (1.2.1)$$

където:

(SR) следва да се чете "*съдържание на сегментен регистър*";

D (*Displacement*) - отместване. Отместването е адрес спрямо началото на сегмента, за който важи. То се определя от програмиста и се съдържа най-често в адресната част на командата. Може да бъде 8-битово или 16-битово число в допълнителен код.

Умноженото на 16 съдържание на сегментен регистър представлява реален 20 битов базов адрес (XXXX0). Според тази схема минималният обем на един сегмент е 16 байта. За реализация на зависимостта (1.2.1) е необходим съответният хардуер. Последният се съдържа в адресното АЛУ (вижте фигура 1.1.1). Например, нека (CS)=1A03, а отместването D=FFFB. Така ще получим следния ефективен адрес:

1A030 + FFFB = 2A12B .

Тъй като сегментните регистри са 4, то във всеки момент е необходимо да бъде известно кой от тях ще подаде числото, с което по формулата (1.2.1) ще се проведе изчислението на ефективния адрес. Тези правила се реализирани хардуерно и се реализират в процеса на управление от страна на ЦУУ и на УУССШ.

Приети са по подразбиране следните правила:

- Ако отместването D се доставя от програмния брояч, т.е. адресира се команда, то базовия адрес доставя регистъра на кодовия сегмент CS и формулата приема вида:

$$A_{ef} = (CS).2^4 + (IP) \ .$$

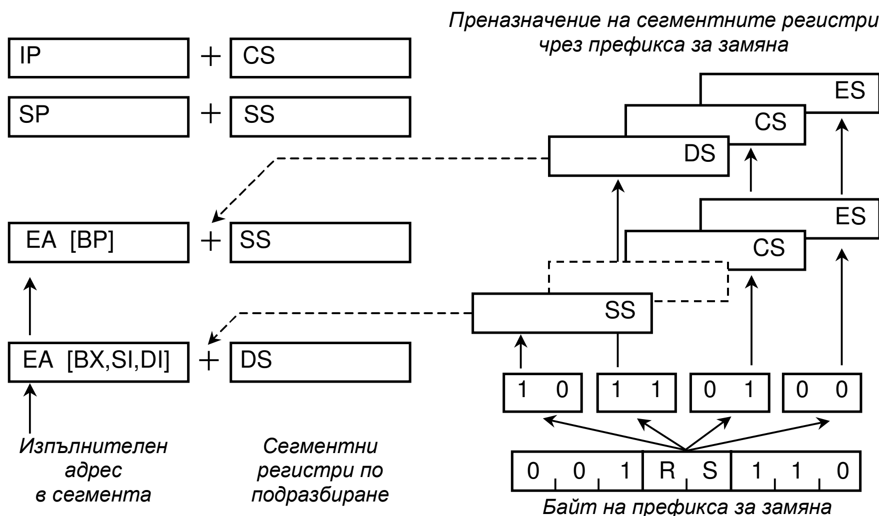
- Ако се извършва обръщение към програмния стек, тогава базата се доставя от регистъра на стековия сегмент SS. Отместването се доставя от стековия указател SP:

$$A_{ef} = (SS).2^4 + (SP) \ .$$

- Ако обръщението се отнася за данни (с цел четене или запис), адресирането използва сегментния регистър DS, а доставката на отместването зависи от метода за адресация. Използва се още един от регистрите BX, DI, SI, BP или тяхна комбинация.

Ефективният адрес е физическият адрес на клетката от ОП. Адресът, който се нарича *изпълнителен* и се определя от структурните елементи на машинната команда, които ще бъдат разгледани по-късно, представлява отместване спрямо базовата стойност (SR)*16. Изпълнителният адрес може да се разбира още като абсолютен, но само спрямо абстрактното начало 00000h на сегмента. Освен това, с кой от сегментните регистри ще работи текущата команда е записано в нейния префикс за подмяна на сегментните регистри. Префиксът е с дължина байт и е част от кода на операцията.

Според (1.2.1) при изчисляване на всеки адрес се използват два елемента. Сдвоените отношения (IP→CS) и (SP→SS) е невъзможно да бъдат променени. Възможно е да бъдат преназначени само сегментни регистри и следователно сегменти в паметта, когато от ефективния адрес се изчислява физическият адрес. Това е възможно да се постигне чрез префиксния байт, който следва да предхожда машинната команда, която се обръща към нестандартен сегмент в паметта. Замените, които префиксният байт може да назначи, са илюстрирани на фигура 1.2.1.



Фиг. 1.2.1 Схеми за назначение на сегмент чрез префиксния байт

Счита се, че сегментната организация на паметта притежава редица положителни качества:

- Така например, програмите, които не променят съдържанието на сегментните регистри, могат да се преместват в паметта, налагащо се например при прекъсване ;
- На второ място припокриването на сегментите позволява ефективно използване на адресното пространство ;
- На трето място непресичащите се сегментни области позволява лесна реализация на защитата на данните в паметта.

1.3 Външен интерфейс на микропроцесора

Ще опишем кратко сигналните пинове (изводи на интергралната схема) на микропроцесора, чрез които той се свързва с другите схеми.

Сигнал $\overline{\text{RST}}$ (*reset – установка в изходно състояние*). Входен за микропроцесорната схема сигнал (отрицанието означава активна нула). Сигналът се издава автоматично след включване на захранването от интегралната схема на тактовия генератор и следва да е активен не по-

малко от 4 такта. Причинява установяване на програмния брояч в състояние 0000, а кодовия сегментен регистър в състояние FFFF. Така след отпадането му, командният цикъл стартира с извличане на първата команда от адрес FFFF0.

Изводите на интегралната схема имат следните особености:

1. Някои изводи (пинове) могат вътрешно да се **мултиплексират**, при което в различни моменти от времето са свързани към различни източници/приемници. Двойната функционалност ще записваме с наклонена черта, например, на един от изводите сигналът $\overline{\mathrm{BHE}}$ (*Byte High Enable*) се съвместява със сигнал S7 (*Status*) за състоянието на процесора. Така този пин се означава както следва: $\overline{\mathrm{BHE}}/\mathrm{S7}$. Аналогично е означението $\mathrm{MN}/\overline{\mathrm{MX}}$, сигнал който определя конфигурацията на системата – минимална или максимална. Чрез него могат да се изменят функциите на други сигнали, за да се приспособява системата към конкретните външни условия.

2. Някои от изводите могат да променят **направлението** си. В различни моменти от времето могат да бъдат входове и да приемат сигнали или да бъдат изходи и да издават сигнали.

3. Някои изводи са вътрешно **буферирани** и могат да се изключват като се поставят в така нареченото "трето логическо състояние" (състояние Z на висок изходен импеданс). Такива изводи са свързани с арбитрирането на системната шина, както и със системата за прекъсване. Ще казваме, че имат 3 състояния 0,1 и Z.

Шина **A/D** – *Address/Data* (вижте фигура 1.1.1). Това е 16 битова мултиплексируема двупосочна (входно-изходна) шина. В такт T1 на системния цикъл шината се конфигурира като изходна и по това време тя извежда 16 младши бита на адрес. По това време тя се определя като адресна. В следващите тактове шината се мултиплексира вътрешно заедно с определяне на посоката й в зависимост от направлението на данните. В този временен интервал шината се определя като даннова. Шината транспортира 16 битови или 8 битови данни. Всяка от половините на данновата шина може да транспортира 8 битова даннова комбинация, в зависимост от това дали адресът е четен или нечетен.

Шина **A/S** – *Address/Data* (вижте фигура 1.1.1). Това е 4 битова вътрешно мултиплекируема шина за транспортиране на старшите 4 бита на адрес или битовете на състоянието от S6 до S3. В такт T1 на системния цикъл шината извежда адресните битове, а в следващите тактове състоянието на процесора. Сигналите S4 и S3 идентифицират съответно адресирания сегмент:

0 0 - указва допълнителния сегмент ;
0 1 - указва сегмента на програмния стек ;

1 0 - указва кодовия сегмент ;

1 1 - указва данновия сегмент.

Сигналът S5 изважда маската за прекъсване (S5=1 означава разрешение за прекъсване, а S5=0 – забрана за прекъсване). Сигналът е винаги S6=0.

Сигналът $\overline{BHE}/S7$ (*Byte High Enable / Status*) се използва като разрешаващ. Разрешението е необходимо в такт Т1, за да се запише издавания от процесора адрес в допълнителния външен адресен регистър, в който се фиксират старшите 4 бита на адреса. Сигналът S7 се извежда в следващите тактове. Това е резервен сигнал и може при необходимост да се използва от Проектантът на системи.

Сигналът за четене $\overline{RD}$ (*Read*) като кратка нула в тактовете Т2 и Т3 още се нарича *строб* за четене. Изводът му е вътрешно буфериран и може да се изключва, като се поставя в трето състояние Z.

Сигнал Готовност RDY (*Ready*) е входен за процесора. Сигналът е предназначен да потвърди пред процесора готовността на паметта или на друго външно устройство да извърши обмен на данни в установения момент. Ако паметта или устройството за вход/изход не успяват да проведат обмена (нуждаят се от задръжка), те са длъжни да снемат активното ниво на сигнал RDY.

Сигнал за немаскируема заявка за прекъсване може да постъпи на вход NMI (*Non-Maskable Interrupt*). Входът се нарича динамичен, защото тригерът, който фиксира тази заявка, е с динамично управление (тригер със структура *Edge* – вижте съответния раздел в [3]). Тази заявка не може да бъде маскирана от флаг IF и винаги причинява прекъсване №2.

Сигнал за маскируема заявка за прекъсване INTR (*Interrupt*). Сигналът се анализира в последния такт от изпълнението на текущата команда (по силата на командния цикъл) и ако не е маскиран от флаг IF, микропроцесорът пристъпва към изпълнение на МППОП (микропрограмна процедура за осъществяване на прекъсване – вижте съответния раздел в [1]).

Сигнал за проверка $\overline{TEST}$. Този сигнал е входен за процесора. Състоянието му се проверява от специална машинна команда WAIT, действието на която е следното:

- Ако входното ниво на сигнала е ниско, продължава изпълнението на текущата програма ;
- Ако входното ниво е високо, микропроцесорът преминава в състояние на очакване, което продължава докато нивото на сигнала не падне. С падането на нивото на входния сигнал изпълнението на текущата програма продължава.

Тактов сигнал CLK (*Clock*). Тактовият сигнал има коефициент на запълване 33%. Изработва се заедно с други синхронизиращи сигнали от интегралната схема на тактовия генератор.

Следващите 8 сигнала имат описаната функционалност когато системата е определена в минимална конфигурация, т.е. $MN/\overline{MX}=1$.

Сигнал потвърждение за приемане на заявка за прекъсване $\overline{INTA}$ (*Interrupt Acknowledge*). Сигналът е изходен и има 3 състояния. Активното ниско ниво се установява в тактовете T2 и T3 на цикъла за обработка на заявката, в който външният източник следва да постави на данновата шина кода за прекъсване, а процесорът да го приеме и преобразува в адрес към таблицата на векторите за прекъсване.

Сигнал строб на адреса ALE (*Address Latch Enable*). Сигналът съпровожда в такт T1 издавания от процесора адрес и се използва от външния адресен регистър за неговия запис. По същото време този регистър е разрешен от сигнала $\overline{BHE}$.

Сигнал, разрешаващ обмена на данни $\overline{DE}$ (*Data Enable*). Изходен сигнал с 3 състояния (0,1,Z). Сигналът се използва за разрешение на двупосочните даннови буфери към системната шина. В циклите на запис активното ниво на сигнала се явява в началото на такт T2, а в циклите на четене – с известно закъснение, в средата на T2, което е необходимо за изчакване на идващите по данновата шина данни. В такт T4 активното ниво на сигнала се прекратява.

Сигнал за определяне на посоката на обмен $DT/\overline{R}$ (*Data Transmit / Receive*). Сигналът е изходен с 3 състояния (0,1,Z). Управлява споменатите по-горе двупосочни шинни буфери. Сигналът се определя в началото на всеки шинен цикъл.

Сигнал за избор на адресно пространство $M/\overline{IO}$ (*Memory / Input-Output*). Този микропроцесор има изолирано от основното входно-изходно адресно пространство и за да се определи назначението на издавания от процесора адрес, той се съпровожда от този сигнал (вижте темата за организация на входно-изходния обмен в [1]). Когато нивото е високо, обменът е с основната памет, а когато е ниско – с входно-изходен порт. Нивото на сигнала се определя в началото на всеки шинен цикъл.

Сигнал код на операция запис $\overline{WR}$ (*Write*). Изходен сигнал с активно ниско ниво и 3 състояния (0,1,Z). Активното ниво на сигнала съпровожда данните във всеки цикъл на запис.

Заявка за заемане на шината HLD (*Hold*). Високото ниво на сигнала по този вход се възприема от ЦУУ на процесора като заявка от външно устройство, копроцесор или друг процесор за правото да владее шината.

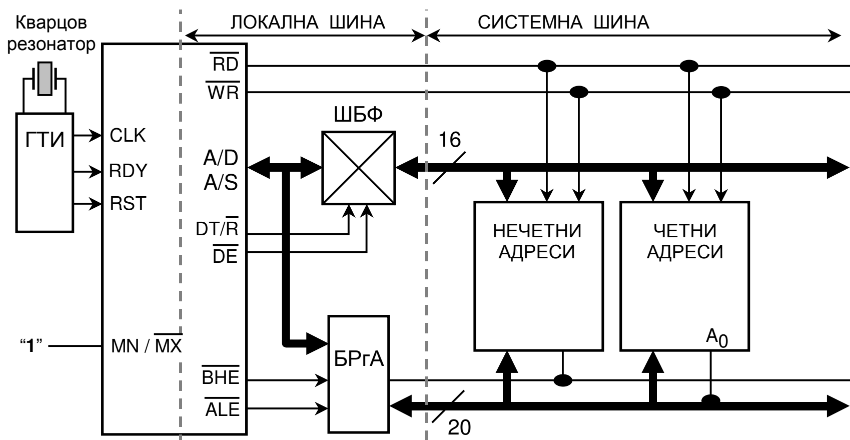
Отговор потвърждение HLDA (*Hold Acknowledge*). Изходен сигнал, който издава процесора към външния контролер. Заедно с това процесорът вътрешно изключва шините A/D, A/S и CS като привежда техните линии в трето състояние. С това всички последователности от

активни и неактивни нива върху линиите на шината следва да се генерират от онзи елемент в системата, който е получил правото да управлява шината. Процесорът се извежда от "замръзналото" състояние с пропадане на сигнала HLD.

Функциите на описаните 8 сигнала в режим на максимална конфигурация няма да описваме.

1.4 Организация на обмена

На фигура 1.4.1 е показана опростена структурна схема на микро процесорната система в минимална конфигурация, в която са показани част от споменатите вече елементи и сигнали.



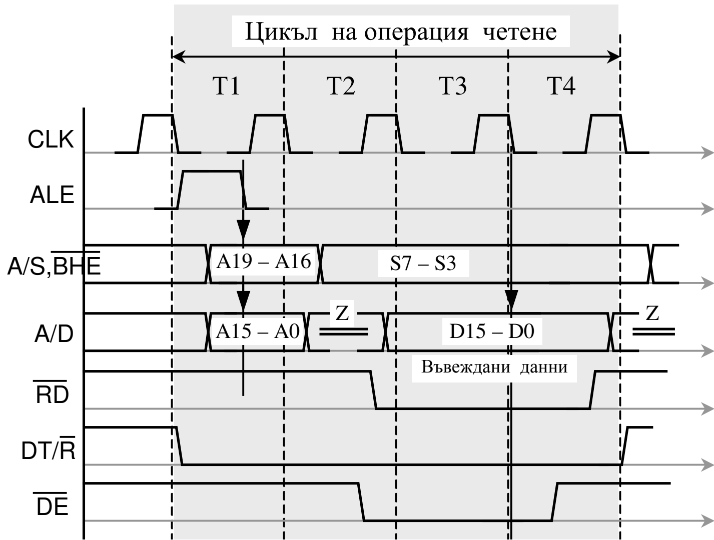
Фиг. 1.4.1 Структура на микропроцесорната система

Микропроцесорът се обръща към паметта извършвайки операции четене или запис. Погледнато отвън върху системната магистрала непрекъснато се изпълняват цикли на обмен, а понякога и "празни" цикли. Както структурата, така и времедиаграмата, поясняваща циклите на обмен, са пояснени в случая за режима на минимална конфигурация на системата, което се вижда от константната логическа единица на входа $MN/\overline{MX}$.

Непосредствените изводи от микропроцесорната интегрална схема образуват локалната шина. Това са шините A/D, A/S и CS (фиг. 1.1.1).

Тъй като адресната шина е мултиплексируема, адресът следва да се запомни - старшите 4 бита в буферния регистър БРг.А, младшите 16 в схемите на ОП. Записът в регистър БРг.А се разрешава от сигнала $\overline{BHE}$, а се осъществява в такт T1 от сигнала ALE. Тъй като младшата 16 битова част се мултиплексира външно и по посока, нейните линии преминават през шинните буферни формирователи ШБФ (вижте фигура 1.4.1). Тези буфери осигуряват както посоката, така и необходимата мощност на линиите.

Всеки цикъл на обмен се състои от 4 такта. За начало на всеки такт се приема задния фронт на тактовите импулси CLK. Ако външният обект, който в дадения цикъл използва шината, не успява да осъществи обмена за това време, той снема сигнала RDY, при което УУССШ вмъква в микропрограмата между тактовете T3 и T4 необходимия брой тактове за изчакване Tw. Процесът на обмен по системната шина е илюстриран с времедиаграмите от фигура 1.4.2 и 1.4.3.



Фиг. 1.4.2 Времедиаграма на цикъл четене

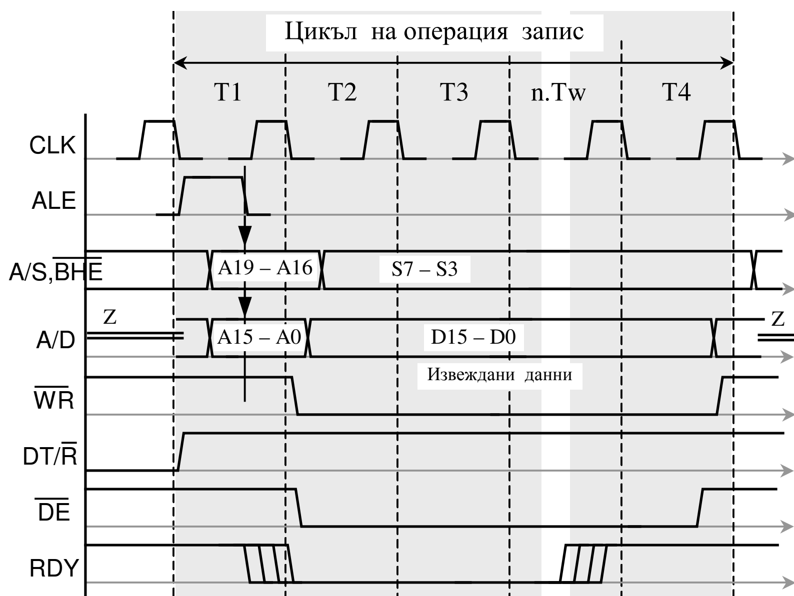
Както се вижда, в такт T1 процесорът подава адрес на клетка в ОП, сигнал ALE и сигнал DT/R, с който шинните буфери се превключват в посока към процесора. Спадащият фронт на сигнала ALE се използва за запис на адреса в буферния регистър на адреса.

В следващия такт T2 адресните шини вътрешно се превключват и мултиплексират така, че шината A/D се свързва с входа на онзи регистър, който машинната команда е назначила за приемник на данните. По време на това вътрешно мултиплексиране линиите на данновата шина се изключват като минават в трето състояние, означено с буква Z. В този такт се появява и активното ниско ниво на разрешаващия сигнал DE, както и кода на операцията за ОП – сигналът RD. След завършване на вътрешното мултиплексиране, в края на T2, линиите на тази шина отново се включват и върху тях се очаква пристигането на логическите нива на прочетените от паметта данни.

Изчаква се един такт за надеждно установяване на данните върху шината и в края на такт T3 по спадащия фронт те се фиксират в регистъра приемник в процесора. Данните остават стабилни още известно време в такт T4, когато се свалят активните нива на разреше-

нието DE и на кода на операцията RD. В края на такт T4 данновата шина отново се изключва за да се извърши вътрешната й настройка в съответствие със следващата машинна команда.

Що се отнася до шината A/S, след записването на адреса във външния буферен регистър, по време на такт T2 тя вътрешно се мултиплексира, като в оставащите тактове T3 и T4, по нейните линии се извеждат стойностите на битовете на състоянието на процесора от S7 до S3. На следващата времедиаграма е илюстриран процеса на операция запис, при която посоката на обмен е обратна.



Фиг. 1.4.3 Времедиаграма на цикъл запис

При изпълнение на операция запис (извеждане) в такт T2 на локалната шина се явяват данните, които процесора осигурява. През шинните буфери в посока към паметта (сигнал DT има високо ниво) или към външен обект, данните се разпространяват чрез системната шина, заедно с кода на операцията сигнал WR – активно ниско ниво. Данните и сигнал WR съхраняват активните си нива до края на такт T4.

Синхронизацията на процесора с бавно работещи външни памети или устройства се осигурява от сигнала готовност RDY. Процесорът проверява нивото на този сигнал в такт T2 на всеки цикъл. Ако нивото му е високо, обменът протича в 4 такта. Ако обаче нивото му е ниско, както е показано на фигура 1.4.3, то след такт T3 процесорът изпълнява цикъл на очакване, който върху шината е изразен чрез вмъкнатите след T3 тактове Tw (тактове на очакване). В този цикъл на очакване във всеки такт процесорът следи нивото на сигнал RDY и когато установи,

че то е станало високо, изпълнява такт T4, с което завършва текущият цикъл на обмен. Сигналът RDY, който се формира от външния обект, се синхронизира чрез допълнителни схеми със сигнала CLK.

Както е показано на фигура 1.4.1, паметта на системата използва двукратно разслоено адресиране, което е възможно, тъй като ширината на данновата шина има дължината на разрядната мрежа. Чрез сигнала BHE и младшата адресна линия A0 могат да бъдат достъпни отделни байтове както с четни, така и с нечетни адреси, т.е. не е необходимо изравняване до четни адреси. Една дума (2 байта) може да се разполага в паметта по 2 начина:

- С младшия байт напред ;
- Със старшия байт напред.

И в двата случая обаче за адрес на думата се приема адресът на младшия байт.

Когато границите на една дума не са изравнени по четните адреси, както е показано на следващата рисунка:

XXXX9	Младши байт (L)		XXXX8
XXXXB		Старши байт (H)	XXXXA

нейните байтове се обменят в два последователни цикъла. Обменът винаги започва с младшия байт, който се адресира с комбинацията XXXX9. Това адресиране се осигурява от стойностите $\overline{BHE}=0$ и A0=1. В следващия цикъл се отваря клетка с адрес XXXXA, при което $\overline{BHE}=1$, а младшата адресна линия е A0=0.

1.5 Организация на входно-изходния обмен

Този микропроцесор има изолирано входно изходно пространство [1]. Обменът с клетките от това адресно пространство е подобен на вече описания. Използва се същата системна шина, но отделните машинни команди разпространяват различни управляващи сигнали. Всяко външно устройство се проектира във входно-изходното адресно пространство. Клетките в него се наричат портове и се адресират аналогично. За достъп до портовете машинните команди използват два метода за адресиране – непосредствен и косвен чрез регистър DX.

Цикълът на обмен с В/И портове се отличава от описания с ОП само в два момента:

1. В началото на всеки цикъл на извода $M/\overline{IO}$ се установява ниско ниво, с помощта на което се разрешава като цяло входно-изходното адресно пространство. Специално предназначение имат и сигналите S2, S1 и S0.
2. В такт T1 на линиите от A15/D15 до A0/D0 се извежда адресът на заявения порт. Старшите 4 бита от адреса A19/S6 – A16/S3 винаги са нула, т.е. проекциите на ВУ са в началото на адресното пространство.

Конструктивно се уговаря областта от адресното пространство, където се проектират портовете, определя се началния адрес на сегмента, неговия тип (обикновено данновия или допълнителния сегмент). Това осигурява по-къс адрес и по-къси команди. Разпределението на портовете по адреси е общо прието и може да се намери в [1].

1.6 Система за прекъсване

Микропроцесор 8086 притежава векторна система за прекъсване, чрез която той може да реагира на вътрешни и външни заявки за прекъсване. Възможните източници на заявки за прекъсване могат да бъдат максимум 256 на брой. Всеки един от тях е свързан със специално предназначена за него обслужваща програма. Връзката се осъществява с помощта на таблицата на векторите за прекъсване, намираща се в оперативната памет. Таблицата съдържа 256 на брой 4 байтова адресна информация за началния адрес на обслужващата програма. Тези 4 байта се четат от ОП в 2 последователни цикъла на системната шина и съдържат:

- Първите 2 байта представляват сегментен адрес SA;
- Вторите 2 байта представляват отместването D в този сегмент.

Ефективният начален адрес на обслужващата програма се изчислява по формулата:

$$A_{ef} = SA.2^4 + D = (\alpha).2^4 + (\alpha + 2) \,. \qquad (1.6.1)$$

където: (α)=SA, (α+2)=D. С α е означен адресът на клетката, в която се намира SA, а с α+2 е означен адресът на следващата клетка, в която се намира отместването D. “Входният” адрес α в таблицата на векторите се изчислява по формулата:

$$\alpha = V.4 \,. \qquad (1.6.2)$$

където с V е означен номерът на прекъсването, който “носи” със себе си всяка заявка за прекъсване. Номерът на заявката, която ще бъде приета за обслужване, се генерира апаратно и автоматично. Адресът α се явява косвен адрес – обръщението към ОП по този адрес доставя адреса A_{ef}, след което следва ново обръщение към ОП за доставка на първата машинна команда от обслужващата програма.

Вътрешните прекъсвания могат да бъдат:

- Апаратни. Апаратните прекъсвания са две – при деление на нула (№0) и постъпков режим (№1). Прекъсване №0 се заявява, когато при изпълнение на команда DIV или IDIV апаратурата в АЛУ разпознае делител равен на нула. Прекъсване №1 се генерира след изпълнението на всяка машинна команда, ако е вдигнат флагът TF ;

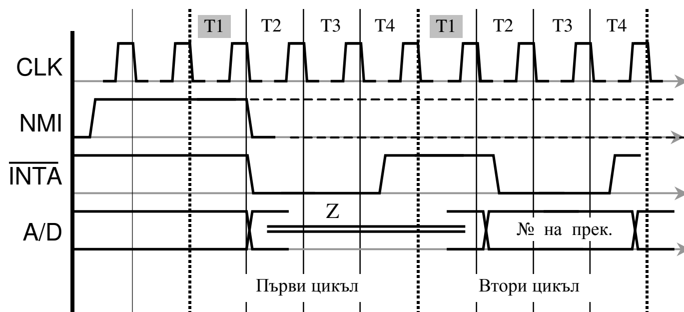
- Апаратно-програмни. Към този тип прекъсване се отнася прекъсване №4, което настъпва, ако при изпълнение на команда INTO флагът за препълване OF е вдигнат ;

- Програмни. Този тип прекъсвания настъпват при изпълнение на команда INT 3 (прекъсване №3), или при изпълнение на двубайтовата команда INT type, която имитира прекъсване от всеки тип.

Външните заявки за прекъсване могат да постъпят по 2 специални входа: Вход NMI за немаскируеми заявки и вход INT за маскируеми заявки (вижте фигура 1.1.1).

Немаскируемите заявки причиняват прекъсване №2 и се обслужват безусловно. Те се възприемат по предния фронт на сигнала и се запомнят в асинхронен тригер на входа NMI.

Маскируемите заявки за прекъсване по входа INT се запомнят по тактовия сигнал CLK във входен тригер, където изчакват своето възприемане. Последното е възможно, ако флагът IF е установен в 1. Сигналът на заявката трябва да бъде активен (високо ниво) не по-малко от един период на тактовата честота CLK. Ако заявката бъде приета, процесорът изпълнява два специални цикъла, наричани “потвърждение на прекъсването”, както е показано на фигура 1.6.1.



Фиг. 1.6.1 Цикъл потвърждение на прекъсване

По време на първия цикъл локалната шина на процесора A/D се намира в трето състояние (Z). По същото време в такт T2 процесорът подава сигнал потвърждение $\overline{INTA}$ към външното устройство (активна нула). Разбирайки по този начин, че заявката му е приета, в следващия цикъл външното устройство предава към процесора по данновата шина 8 битовия код на прекъсването.

При наличие на повече от една заявка за прекъсване, влиза в действие приоритетният принцип за избор на заявка. Приоритетът е конструктивно заложен във всеки вид процесор (вижте в [1] таблица 5.4.2.1) и се реализира с помощта на апаратурата на системата за прекъсване.

След приемане на заявката за прекъсване процесорът изпълнява микропрограмната си процедура за осъществяване на прекъсването, заложена в командния цикъл. Изпълняват се следните действия:

1. Съхраняване на текущото състояние на процесора в стека ;
2. формиране на адреса за преход към обслужващата програма.

Обработката на всяко прието прекъсване започва със запис в стека на съдържанието на регистъра на флаговете, веднага след което на флаговете IF и TF се присвояват нови нулеви стойности, забранявайки по този начин следващите прекъсвания. След това в стека се съхранява съдържанието на сегментния регистър за команди CS и съдържанието на програмния брояч IP. По този начин в стека се съхранява адресът за връщане от прекъсване.

След това от приетия еднобайтов код на прекъсването се формира адресът на вектора за прекъсване. С този адрес се "влиза" в таблицата на векторите за прекъсване и се извлича ново съдържание за регистрите CS и IP. С това микропрограмната процедура за осъществяване на прекъсването МППОП завършва и тя предава управлението на командния цикъл. Командният цикъл влиза във фазата на извличане на машинна команда по новия адрес, с което фактически стартира програмата за обслужване.

Програмата за обслужване винаги завършва с команда IRET (*Interrupt Return*) – "връщане от прекъсване". Микропрограмата на тази команда извършва действия, подобни на онези, които изпълнява МППОП, но в обратен ред – извлича съхранените в стека съдържания на регистрите IP (програмен брояч), CS (сегментен регистър на кода на програмата) и RF (регистър на признаците). Със затварянето на командният цикъл се възстановява изпълнението на прекъснатата преди това програма.

Включване на микропроцесора. След включване на захранването (*Cold bootstrap*) или след рестарт (*Warm bootstrap*) повторно стартиране на хардуерната система се осъществява от сигнал RST. За целта, при включване на захранването, на входа RST е необходимо да се установи високо ниво на сигнала в продължение на не по-малко от 50 [μs]. Този сигнал се генерира от интегралната схема на тактовия генератор (i8284). В случай, че при включено захранване, се прави повторно стартиране, високото ниво на сигнала RST се поддържа не по-кратко от 4 такта на сигнала CLK.

С появата на предния фронт на сигнала RST микропроцесорът прекратява всякакви действия, а във вътрешността му се извършва принудителна инициализация на регистрите му, които получават следното ново съдържание:

Регистър на признаците (флаговете) (RF):=0000h ;
Кодов сегментен регистър (CS):=FFFFh ;
Програмен брояч (IP):=0000h ;
Даннов сегментен регистър (DS):=0000h ;
Стеков сегментен регистър (SS):=0000h ;
Допълнителен сегментен регистър (ES):=0000h .

След така извършена начална инициализация и възстановяване на

тактуването, микропроцесорът изпълнява кратка машинна програма за начална инициализация, чиято първа команда той извлича от адрес FFFF0h. Последните 16 клетки от паметта (от FFFF0h до FFFFFh) са резервирани от Конструктора за тази програма. Задачата на програмата за начална инициализация е да предаде управлението на друга приложна програма, или (най-често) на операционната система.

Стопиране на микропроцесора може да бъде извършено програмно с команда HLT (*halt state*). Извеждане на микропроцесора от това състояние може да се постигне само чрез сигнал RST или чрез заявка за прекъсване по вход NMI или INT.

1.7 Система машинни команди

Системата машинни команди на този микропроцесор съдържа 6 групи команди:

1. За прехвърляне на данни. Тези команди осигуряват обмен на данни между отделните регистри, между регистри и клетки от паметта. Тъй като този микропроцесор има изолирано от основното входно-изходно адресно пространство, той има отделни машинни команди за обмен на данни с входно-изходните портове. Тези команди се наричат команди за вход-изход.

2. За аритметически операции. Това са операциите събиране, събиране с пренос, инкремент, изваждане, изваждане със заем, декремент, умножение, деление, 2/10-чна корекция. Специално следва да се отбележат командите за обработка на числа, представени във форма с плаваща запетая.

3. За логически операции и за измествания. Логическо събиране ИЛИ, логическо умножение И, отрицание НЕ, неравнозначност ⊕, различни видове измествания и сравнения. Изместванията са логически, аритметически и циклически. Логическите операции са поразрядни.

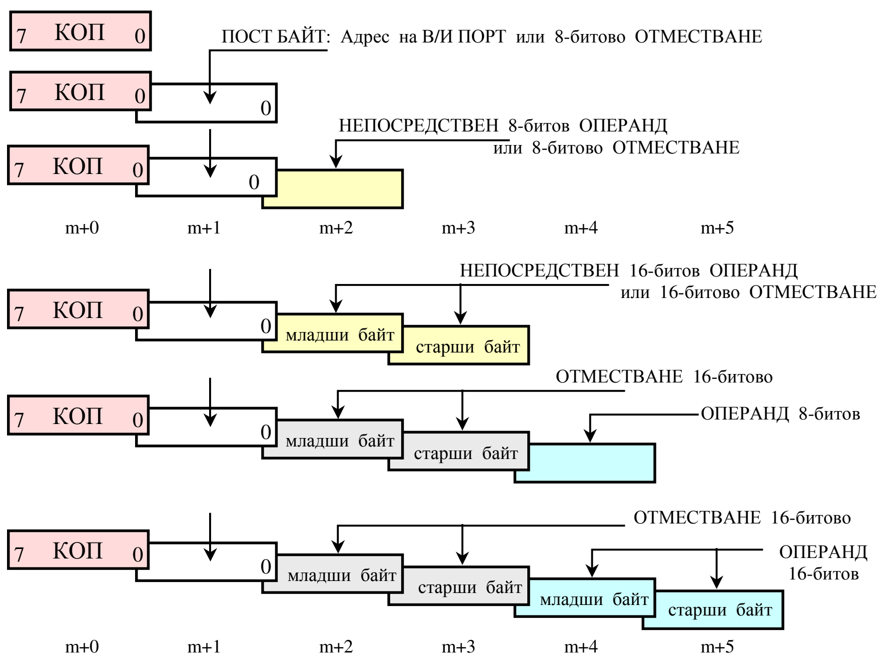
4. За символна обработка. Командите за символна обработка често се използват с префикс за повторение. Така действието на командата може да се повтори многократно за следващите символи. Подобен цикъл се организира лесно когато командата се намира в буферната памет. С помощта на такива команди могат да се преместват последователности от байтове или думи с дължина до 64К, да се сравняват две последователности или да се търси символ в последователност.

5. За управление на прехода. Тази група команди се явява в системата като следствие от естествения метод за адресиране на командите в този микропроцесор. Те реализират различни условни и безусловни алгоритмични преходи както вътре в кодовия сегмент, така и извън него. Реализират подпрограмната техника, както и различни видове прекъсвания.

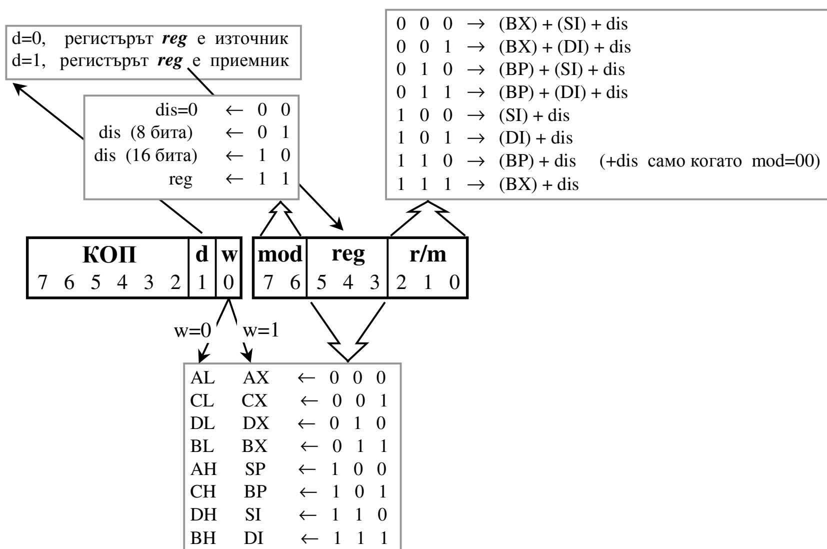
6. За управление на режимите на микропроцесора. Командите за управление на процесора осигуряват на програмиста възможност да манипулира системни регистри като установява или сваля флагове, маски, проверява наличието на външни сигнали, както и да осигурява работата на процесора в максимален режим.

Като цяло системата машинни команди на този микропроцесор се характеризира като CISC (*Complex Instruction Set Computing*), т.е. комплексен набор команди [1]. Основна характеристика на такава система е променливият формат на командите. С цел съкращаване на времето за извличане от паметта, форматът на всяка команда е конструиран с отчитане на относителната честота на употреба. Така често употребяваните команди имат къси формати. Те са едно-байтови, дву-байтови или три-байтови. По-рядко използваните команди са по-дълги. Те са 4-байтови, 5-байтови или 6-байтови. Основните формати на машинните команди са представени на фигура 1.7.1.

Всяка машинна команда (като част от програмата) се разполагат в паметта побайтово във възходящ ред на адресите. За адрес на една командата се приема най-малкият адрес на заеманата последователност от клетки. Командите се извличат от паметта (изпреварващо) по 2 байта едновременно (вижте фигура 1.4.1) и се разполагат в буферната опашка.



Фиг. 1.7.1 Основни формати на машинните команди



Фиг. 1.7.2 Структура на първите два байта на командата

Алгоритъма за изпълнение на командите се определя най-вече от първите два байта, съдържащи КОП и пост данни. Тъй като тези два байта са вътрешно структурирани, то системата машинни команди на този микропроцесор се определя като *ортогонална*. Структурата на първите два байта и предназначението на отделните полета е илюстрирано на фигура 1.7.2. В рисунките се имат предвид следните означения:

d (*direction*)	посока ;
w (*word*)	дума ;
mod (*mode*)	режим ;
reg (*register*)	регистър ;
r/m (*register / memory*)	регистър/памет ;
dis *или още* D (*displacement*)	отместване.

Както се вижда от рисунките и от означенията, в първия байт се кодира същинската операция, определя се регистъра като приемник или предавател на операнд (параметър d) и дължината на последния (параметър w). Във втория байт има 3 подполета. Първото (mod) определя дължината на отместването, второто (reg) адресира регистър, третото (r/m) определя схемата за формиране на адреса.

Адресите на портовете са възможни в диапазона от 00h до FFh.

При формиране на адреса, 8 битовото отместване се разширява знаково до 16 битово. Отместването винаги се интерпретира като число със знак, представено в допълнителен код. Така възможностите на 8 битовото отместване са в диапазона [-128, +127], а на 16 битовото – в диапазона [-32768, +32767].

1.8 Методи за адресиране

Схемите, по които адресното АЛУ (вижте фигура 1.1.1) изчислява ефективния адрес за обръщение към ОП, са 11 на брой. Някои от тях представляват модификации на основните методи за адресация, които ще припомним, са: непосредствен, пряк, косвен, относителен и индексен.

В много от машинните команди на този микропроцесор адресацията се определя от постбайта. Трибитовото поле *reg*, съвместно с бит w от първия байт задава или 8-битов регистър или 16-битов регистър (вижте фигура 1.7.2).

Двубитовото поле *mod* заедно с полето *r/m* задава схемата за формиране на ефективния адрес. Полето *mod* задава дължината на отместването (00 – без отместване, 01 – един байт за стойността на отместването, 10 – два байта за стойността на отместването, или 11 – освен отместването се използва и съдържание на определен регистър). Кой ще бъде регистърът, чието съдържание ще се използва, се адресира от полето *r/m*. Формулите, по които се изчислява адреса, са изписани в горната дясна част на фигура 1.7.2. Възможният пренос от старшия разряд на получената сума се игнорира, във връзка с правилата за събиране в допълнителен код. Така изчисленият адрес се оказва винаги в пределите на дадения сегмент.

ПРИМЕР: Нека (DS)=1002h. Това означава, че данновият сегмент (в обем от 64[KB]) се намира от адрес 10020h до адрес 2002Fh. Нека (BX)=8005h, което се интерпретира като базов адрес. Нека (*r/m*)=111, което определя за изчисление на ефективния адрес следната формула:

$$A_{ef} = (BX) + dis \quad .$$

В зависимост от съдържанието на полето *mod* ще разгледаме следните варианти:

1. (mod)=00. Това означава, че dis=0. Тогава изпълнителният адрес е

$$A_{ef} = (BX) + dis = 8005 + 0 = 8005h.$$

Физическият адрес AdrMem се изчислява според формула (1.2.1) така:

$$AdrMem = (DS).2^4 + A_{ef} = 10020h + 8005h = 18025h$$

2. (mod)=01. Използва се 8-битово отместване, съдържащо се в командата. Нека dis=D0h. Тъй като това отместване е отрицателно число в допълнителен код, преди събиране то се разширява знаково отляво и добива стойността dis=FFD0h. След това изпълнителният адрес се изчислява по същата схема:

$$A_{ef} = (BX) + dis = 8005 + FFD0 = 7FD5h.$$

В този резултат преносът от старшия разряд е игнориран. Тогава физическият адрес AdrMem се изчислява така:

$$AdrMem = (DS).2^4 + A_{ef} = 10020h + 7FD5h = 17FF5h.$$

3. (mod)=10. Използва се 16-битово отместване, съдържащо се в командата. Нека dis=4000h. Изпълнителният адрес се изчислява по схемата:

$$A_{ef} = (BX) + dis = 8005 + 4000 = C005h.$$

Тогава физическият адрес AdrMem се изчислява така:

$$AdrMem = (DS).2^4 + A_{ef} = 10020h + C005h = 1C025h$$

4. (mod)=10. Използва се 16-битово отместване, съдържащо се в командата. Нека dis=C000h. Изпълнителният адрес се изчислява така:

$$A_{ef} = (BX) + dis = 8005 + C000 = 4005h.$$

В този резултат преносът от старшия разряд е игнориран. Тогава физическият адрес AdrMem се изчислява така:

$$AdrMem = (DS).2^4 + A_{ef} = 10020h + 4005h = 14025h$$

Както виждаме, съдържанието на постбайта на командата определя 2 адреса: на регистър и ефективния. В командите за прехвърляне на данни (MOV) един от тези адреси е източник, а другия приемник. Вариантът се определя от съдържанието на полето за посока d така:

Ако (d)=0, тогава A_{ef} := (reg) – регистърът е източник ;

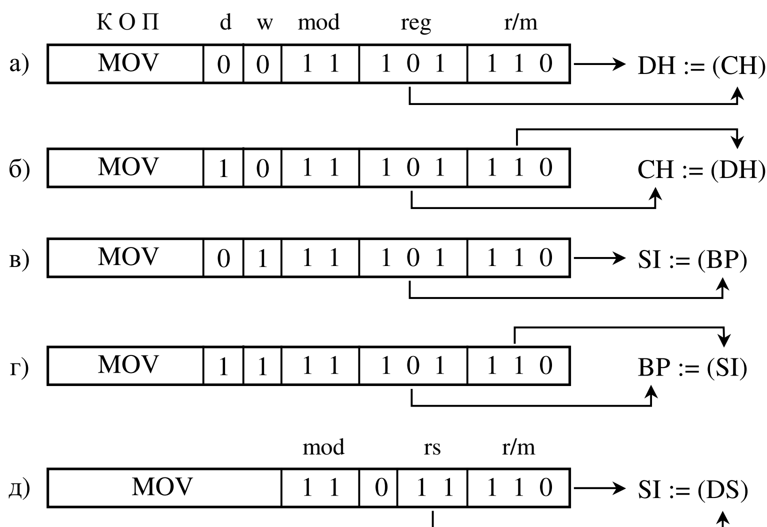
Ако (d)=1, тогава reg := (A_{ef}) – регистърът е приемник .

В редица команди с непосредствен операнд полето *reg* не се възприема като адрес на регистър, а като разширение за кода на операцията КОП.

Регистрова адресация

Това е адресация, при която операндът се намира в регистър. Регистърът, като приемник или като източник, се адресира по указаните по-горе начини с помощта на параметри, намиращи се в първия и във втория байт на машинната команда. Като пример за тази адресация, на фигура 1.8.1, са дадени различни варианти на командата за прехвърляне MOV.

В командите от а) до г) се променят само стойностите на параметрите d (посока) и w (формат) от първия байт. Съдържанието на втория байт остава непроменено. Тъй като в полето *mod* има две единици, то останалите две полета *reg* и *r/m* съдържат адрес на регистър. Така в крайна сметка се определя регистърът приемник и регистъра предавател, например в първия случай четем: регистър DH приема съдържанието на регистър CH.



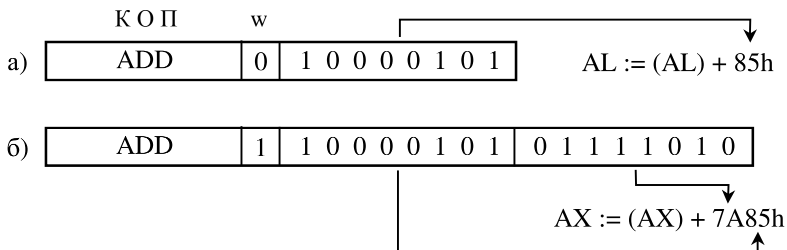
Фиг. 1.8.1 Структура на команди с регистрова адресация

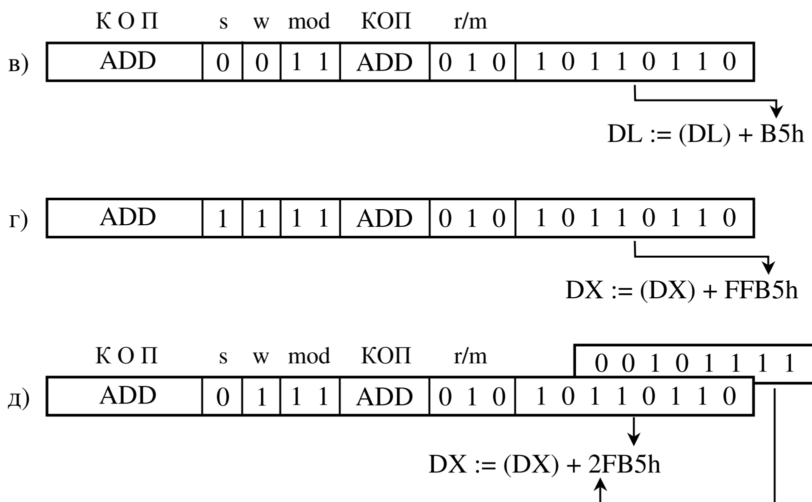
Последната рисунка д) представя структурата на команда, която адресира сегментния регистър DS. Тъй като сегментните регистри са 4, то за тяхното адресиране са необходими 2 бита в полето *rs*. Разпределението на 4-те адреса е следното:

0 0 - ES допълнителен сегментен регистър ;
0 1 - CS кодов сегментен регистър ;
1 0 - SS стеков сегментен регистър ;
1 1 - DS даннов сегментен регистър .

Непосредствена адресация

В този случай командите съдържат в себе си операнда. Както е показано на фигура 1.7.1, операндът е само един, като може да е 8-битов или 16-битов. Непосредственото адресиране е има смисъл само за някои машинни команди. Като пример ще представим команда за събиране (фигура 1.8.2).





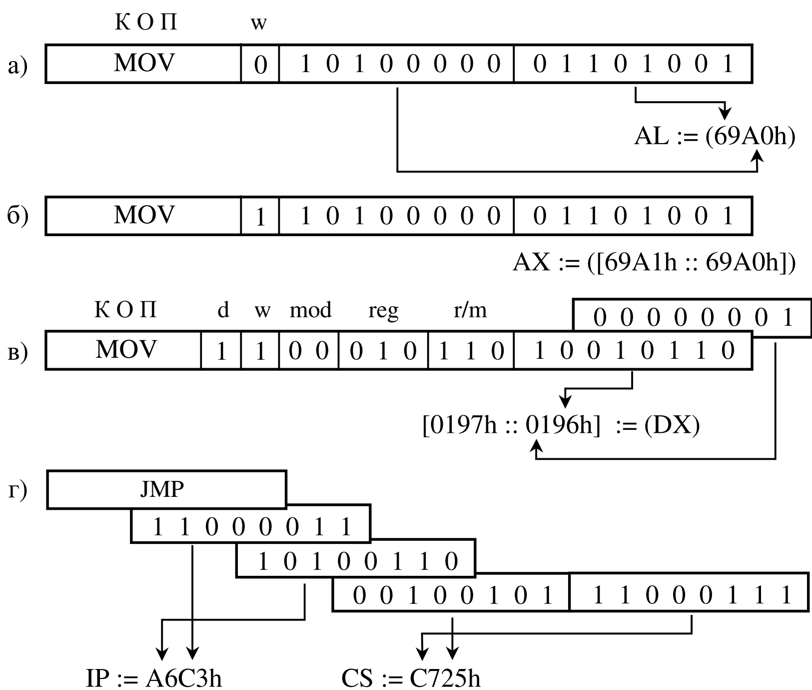
Фиг. 1.8.2 Структура на команди с непосредствена адресация

Полето w в първия байт, както и преди, определя дължината (формата) на операнда. В рисунка а) става дума за 8-битов операнд, а в случай б) – за 16-битов операнд (w=1). Операндът е конкатенация от съдържанията на десните два байта на командата според фигура 1.7.1. Четем: регистър AX приема сумата от съдържанието на регистър AX + числото 7A85h.

В командите с непосредствена адресация постбайтът може да се използва за адресиране на регистър в качеството му на приемник на резултата (вижте рисунки в), г), д) от фигура 1.8.2). В тези команди полето *reg* съдържа част от кода на операция събиране (000) или (101) – изваждане и пр.. Полето *mod*=11 определя младшите 3 бита на постбайта като адрес на регистър. Така r/m=010 адресира регистър DX. Забележете, че дължината на непосредствения операнд се определя от полето w съвместно с полето s, което стои на мястото на d. При комбинация (s:w=11), независимо от това, че операндът е определен като 16-битов (w=1), в командата е заделен само един байт (рисунка г)). В този байт се съдържа само младшата половина на формàта на операнда. При изпълнение на операцията той автоматично се удължава отляво *знаково* и приема вида FFB5h.

Пряка адресация

Методът за пряко адресиране ще илюстрираме отново чрез команда MOV. 16-битовият адрес се съдържа в адресната част на командата (втори и трети байт) – вижте фигура 1.8.3, рисунка а) и б). В зависимост от съдържанието на полето w, от паметта се чете 1 или 2 клетки.



Фиг. 1.8.3 Структура на команди с пряка адресация

В зависимост от това прочетеното се записва в регистър AL или в регистър AX. Фактически съдържанието на младшите два байта (числото 69A0h) представлява адреса на операнда.

На рисунка в) е илюстрирана възможната пряка адресация чрез постбайта, при която в полето *mod* се записва стойността 00, а в полето *r/m* се записва стойността 110. След постбайта следва 16-битово отместване, което се възприема като пряк адрес. Полетата *d*, *w*, и *reg* съхраняват своите функции. Тъй като d=1 и w=1, то регистърът DX (reg=010) се явява източник и неговото съдържание се записва в 2 клетки на ОП [0197h :: 0196h].

Така зададените преки адреси се интерпретират като такива само в рамките на текущия сегмент. Ако прекият адрес се използва в команда за управление на прехода (JMP, CALL), то той задава пряк адрес в текущия кодов сегмент.

Отделно следва да поясним командите за преход с дълъг пряк адрес, в които се съдържа адреса за преход и новото съдържание на кодовия сегментен регистър CS. В такива команди (фигура 1.8.3, рисунка г)) вторият и третият байтове задават адреса на прехода, който при изпълнение се зарежда в програмния брояч, а четвъртият и петият байтове представляват базовия адрес на сегмента в който е насочен

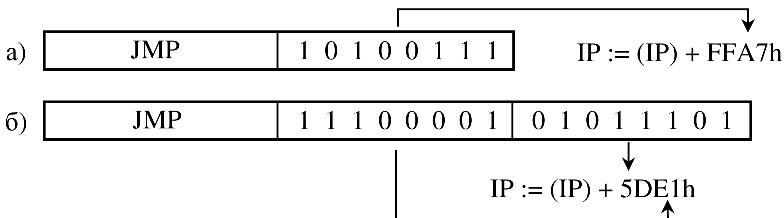
прехода. При изпълнение на командата този адрес се зарежда в кодовия сегментен регистър CS. Изпълнителният адрес се изчислява по формула (1.2.1). Така според примера от рисунката се получава:

$$(C725h).2^4 + A6C3 = D1913h$$

Относителна адресация

Относителна адресация се прилага само в командите за управление на прехода (JMP, CALL, LOOP). Преходът се извършва в текущия сегмент, с отместване, което програмистът поставя в командата. С други думи базовият адрес се съдържа в програмния брояч IP, а отместването – в командата.

Отместването се интерпретира като число със знак в допълнителен код и може да бъде зададено като 8-битово или като 16-битово. Кратко зададеното отместване при изчисляване на адреса се разширява знаково до 16-битово число. На фигура 1.8.4 са представени два примера за относителна адресация.



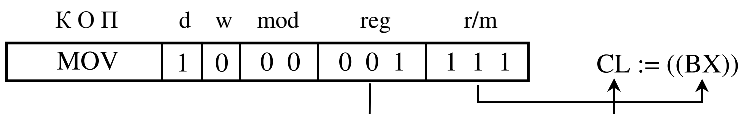
Фиг. 1.8.4 Команди с относителна адресация

Според числата от рисунка а) командата извършва преход назад на 89 клетки. Във втория случай – рисунка б), преходът е в положителна посока на 24033 байта.

Косвена адресация

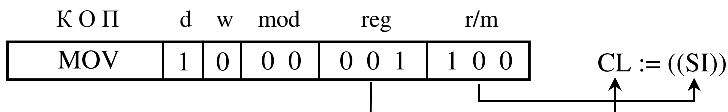
Този метод за адресиране се задава чрез постбайта. Той определя или регистър, съдържащ изпълнителния адрес, или определя схема за изчисляване на адреса от съдържанието на определени регистри. Възможни са три разновидности на този метод:

1. **Косвена базова** адресация. Косвеният адрес е адрес на регистър и се съдържа в полето r/m. Съдържанието на регистъра представлява ефективния адрес. Ефективният адрес може да се съдържа или в регистър BX (r/m=111), или в регистър BP (r/m=110). Ето пример:



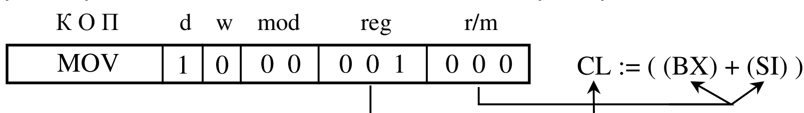
Фиг. 1.8.5 Команда с косвена базова адресация

2. **Косвена индексна** адресация. Аналогична е на горе описаната, но ефективният адрес е възможно да се намира или в регистър DI (r/m=101), или в регистър SI (r/m=100). Ето пример:



Фиг. 1.8.6 Команда с косвена индексна адресация

3. **Косвена индексно-базова** адресация. В тази разновидност ефективният адрес се изчислява като сума от съдържанията на съответната двойка регистри: първото събираемо се избира измежду регистрите BP или BX, а второто събираемо – измежду регистрите SI или DI. Ето един възможен пример:

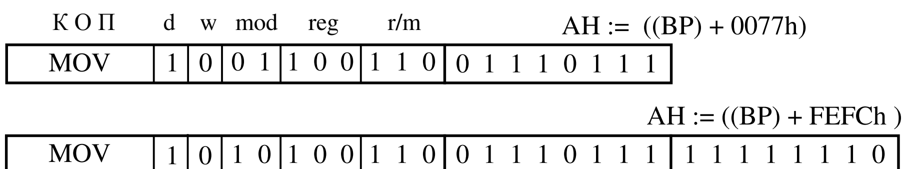


Фиг. 1.8.7 Команда с косвена индексно-базова адресация

Четирите възможни варианта се определят от полето r/m така:

r/m	Сума
000	(BX) + (SI)
001	(BX) + (DI)
010	(BP) + (SI)
011	(BP) + (DI)

Всяка от модификациите на косвената адресация може да бъде без отместване (mod=00) или с 8-битово отместване (mod=10). Ето пример за косвено базова адресация с отместване:



Фиг. 1.8.8 Команда с косвена базова адресация с отместване

При косвена базова адресация с отместване адресът на операнда се изчислява като сума от съдържанието на регистър BP с 8-битовото или с 16-битовото отместване. Тъй като тук се използва базов указател, изпълнителният адрес се намира задължително в стековия сегмент (вижте фигура 1.2.1).

Стекова адресация

Стековата адресация е задължителна за процесори, които реализират подпрограмна техника. За запис на данни в програмния стек

се използва команда PUSH, а за четене – команда POP. Такива команди са безадресни. За адресиране на клетки в стека се използва стековият указател SP. Програмният стек се реализира в паметта като отделна сегментна област за всяко програмно задание. Той се дефинира чрез базовия сегментен адрес, който операционната система назначава и записва в стековия сегментен регистър SS.

Обменът със стека се извършва по думи (2 байта), ето защо стековият указател се модифицира с константата ±2. Характерно за този процесор е, че при запис в стека стековият указател се модифицира декрементно, т.е. SP:=(SP)-2. Това дава основание на някои автори да определят стека като "обърнат". При четене от стека указателят се модифицира инкрементно, т.е. SP:=(SP)+2.

Остава да поясним следните въпроси:

- Как се идентифицира методът за адресиране, който използва дадена команда ?
- Кога вторият байт се възприема като постбайт, кога като непосредствен операнд ?

Отговорите на тези и други въпроси се съдържат в същинския код на операцията КОП, който е част от първия байт на командата (вижте фигура 1.7.2). От намиращата се там комбинация се определя коя е заповяданата операция, има ли командата постбайт и кой метод за адресиране на операндите ще използва тя.

1.9 Даннови формати

Микропроцесорната система поддържа различни формати за всички типове данни. Форматът на числовите данни определя диапазона на представимите числа, който програмистът следва да познава добре. Данновите възможности на микропроцесора са представени кратко в таблица 1.9.1.

Апаратно микропроцесорът обработва само формати, чиято дължина е възможна за разрядната мрежа или за тази на копроцесора. Многобайтовите формати винаги се разполагат в последователни клетки на ОП във възходяща посока на адресите.

Указателят е специфичен тип данни, който по същество представлява адрес. Състои се от две думи – база и отместване. Например, ако последователността от байтове е

A8 : FE : 00 : 01

то базата има стойността 0100h, а отместването стойността FEA8h. Ефективният адрес A се изчислява по формула (1.2.1)

$$A_{ef} = 0100h.2^4 + FEA8h = 10EA8h \ .$$

Следва да се обърне внимание на факта, че различни указатели могат да сочат един и същи адрес. Например, указателят

88 : AA : 42 : 06

Таблица 1.9.1 Даннови формати

Тип	n[b]	8086	8087	Диапазон
Цели без знак				
Байт	8	•		0 ÷ 255
Дума	16	•		0 ÷ 65535
Указател	32	•		0 ÷ 1048575
Цели със знак				
Байт	8	•		-128 ÷ +127
Дума	16	•	•	-32768 ÷ +32767
Къси	32		•	$-2.10^{9} \div +2.10^{9}$
Дълги	64		•	$-9{,}2.10^{18} \div +9{,}2.10^{18}$
Цели 2/10-ни				
Неопаковани		•(1)		
Опаковани		•(1)	•(2)	-99…999 ÷ +99…999 (до 18 цифри)
Реални				
Къси	32		•	$1{,}17.10^{-38} \div 3{,}4.10^{+38}$
Дълги	64		•	$4{,}2.10^{-307} \div 1{,}2.10^{+308}$
Сим. низове				
Байт		•(3)		
Дума		•(3)		

(1) – микропроцесор 8086 апаратно поддържа обработка на 1 байт;
(2) – математическият копроцесор 8087 апаратно поддържа обработка на 18 разрядни десетични числа със знак;
(3) – дължината на символния низ е от 1 до 64К.

води до адрес, който има стойността

$$A = 0642h.2^4 + AAA8h = 10EA8h$$

която, както се вижда, съвпада с по-горе изчислената.

Целите двоични числа със знак, които апаратно могат да се обработват, имат 8 или 16 битов формат. Числата с 32 битов и 64 битов формат се обработват програмно.

Десетичните числа следва да са представени в код 8421, известен като BCD-код. Те могат да бъдат зададени в опакован или неопакован формат. Неопакованият формат съдържа в един байт 2 цифри (2 тетради <d_k1**:**d_k0>), а опакованият – само една <0**:**d_k0>. Десетичната цифра d е означена като една от възможните 10 (k=0,1,2,…,9). Знакът се кодира в старшия бит на отделен знаков байт. Апаратно се поддържа

обработката само на един байт. Изпълнението на операции върху многобайтови BCD-числа е организирано програмно.
Примери:

Неопаковано число: 08 : 09 : 04 : 05 ≡ 5498 .

Опаковано число: 98 : 54 ≡ 5498 .

В микропроцесор i8086 няма устройство за работа с плаваща запетая. Операциите върху числа с плаваща запетая се изпълняват програмно. Ако в системата е конфигуриран копроцесорът i8087, реалните числа се обработват апаратно. Апаратната структура на реалните числа съответства на стандарта IEEE 754 (изместен порядък и техника на скрития бит). Аритметиката на числата с фиксирана запетая и на тези с плаваща запетая са подробно разгледани в [1].

Символните низове се разполагат в паметта като масиви от последователни клетки.

1.10 Копроцесори

Копроцесорите представляват специален клас големи интегрални схеми (ГИС), чието предназначение е да реализират апаратно често срещани операции и функции, като при това поемат управлението на системните елементи. В такива случаи основният микропроцесор е изключен. В съвременни условия тази концепция е силно развита, а техническото изпълнение, благодарение на технологиите, се характеризира с висока степен на интеграция. Копроцесорите основно са предназначени за:

Вход-изход. Този копроцесор (i8089) е предназначен да организира и управлява бърз обмен на големи по обем масиви от данни с външни устройства.

Работа с плаваща запетая. Този копроцесор (i8087), както вече споменахме, реализира апаратно аритметиката върху числа, представени във форма с плаваща запетая. Освен това в него по апаратен път се изчисляват стойностите на редица елементарни математически функции. Тази проблематика и нейните технически решения са изложени подробно в [1].

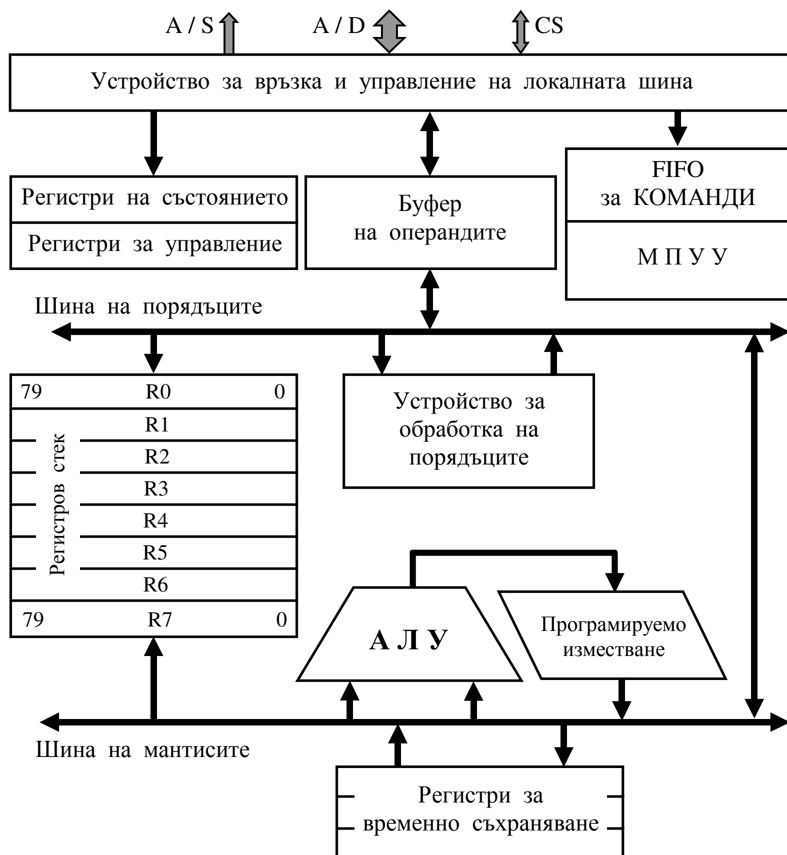
Графически копроцесори. Съдържат апаратна реализация на високо производителни алгоритми за създаване на графични примитиви (линии, окръжности, полигони, текстури и др. п.), за управление на многопрозоречните програмни приложения и за управление на високо качествени графични дисплеи.

Силициеви операционни системи. Това са интегрални схеми, съдържащи ядрото на дадена операционна система. С тяхна помощ се изграждат портативни системи със специализирано приложение.

Комуникационни копроцесори. Съдържат набор от апаратно реализирани комуникационни протоколи. С тяхна помощ се изграждат различни компютърни мрежи.

1.11 Аритметически копроцесор

Обща схема на вътрешната логическа структура на този процесор е представена на фигура 1.11.1.



Фиг. 1.11.1 Структура FPU i8087

Както се вижда от рисунката, процесорът съдържа 6 байтова опашка за команди, буфер за операнди, регистров LIFO стек с дълбочина 8. Дължината на тези регистри е равна на дължината на разрядната мрежа според стандарта IEEE 754, т.е. 80[b]. Този формат се нарича вътрешен. В него се съхраняват всички типове данни, които се въвеждат за обработка в устройството за работа за работа с плаваща запетая. Диапазонът на този формат е огромен - до $\pm 1{,}8.10^{\pm 4932}$. Но само 19 от тези 4932 цифри на значещата част на десетичните числа могат да бъдат представени в 64 битовата мантиса. При извеждане, резултатите се преобразуват в искания за тях тип и формат.

В регистъра на състоянието се съдържа 3 битово поле (*Top*), което адресира върховия регистър в регистровия стек. В този смисъл полето *Top* представлява указател към стека. Стекът е от тип "винаги готов за четене". Всяка операция запис в стека (от тип POP) първоначално инкрементира указателя, след което записва 80 битовото число в новата върхова клетка. След всяка операция четене от стека (от тип PUSH) указателят се декрементира. Машинните команди на копроцесора могат да адресират клетките в стека явно и неявно, задавайки отместване относно текущия указател. Тази двойна адресация облекчава предаването на параметри при обръщение към подпрограма.

При изпреварващото извличане на командите от ОП те се записват в буферните опашки както на основния процесор, така и на копроцесора. На етапа на предварителния анализ, откритите в копроцесора команди, отнасящи се за основния процесор, се игнорират. Само когато процесорът и копроцесорът срещнат команда за разширение ESC, последната се възприема като команда за FPU. Командите за копроцесора са три типа:

- Без обръщение към оперативната памет ;
- С обръщение към ОП с цел четене на операнд ;
- С обръщение към ОП с цел запис на операнд.

В първия случай, след получаване на командата, копроцесорът пристъпва към нейното изпълнение, а основния процесор продължава извличането и изпълнението на команди от програмата.

Във втория случай, след прочитане на командата основният процесор изпълнява фиктивен цикъл за четене от ОП. Прочетената от паметта дума обаче се приема от копроцесора, когато се окаже върху локалната шина. Ако операндът има по-голяма дължина (32 или 64 бита), по време на фиктивния цикъл копроцесорът приема адреса на операнда и заявява искане за управление на шината. Когато го получи, сам управлява прочитането на останалите порции от операнда. След изчитане на операнда връща управлението на основния процесор и се заема с изпълнение на операцията.

В третия случай по време на фиктивния цикъл, който организира основния процесор, копроцесорът прихваща адреса на резултата и го запомня. След като изпълни операция, заявява искане за управление на шината и когато получи правото за това, изпълнява цикли на запис на резултата по запазения адрес.

Тъй като копроцесорът представлява устройство, което е подключено към локалната шина паралелно на основния процесор, е много важно да се организира съвместната им синхронизация. Основната част от тази синхронизация се пада на сигнала BUSY (зает), който копроцесорът формира и подава на вход TEST на основния процесор.

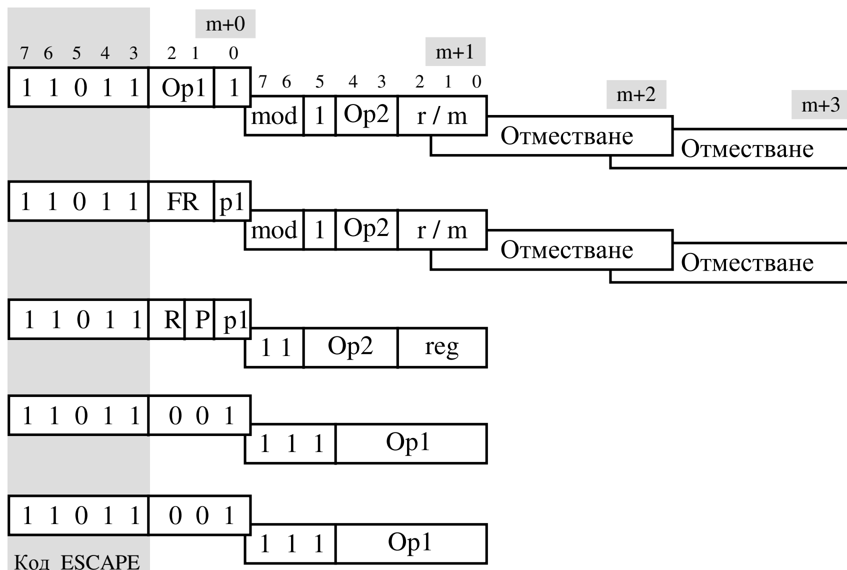
В последния цикъл с команда WAIT може да бъде изчакано завършването на операцията в копроцесора. Това изчакване се налага в два случая:

- Когато програмата прави обръщение към копроцесора преди той да е завършил текущата операция ;
- Когато програмата в основния процесор следва да използва резултатът, който копроцесорът изчислява в текущия момент.

Синхронизиращата команда WAIT се вмъква в програмния текст автоматично преди всяка команда, използваща копроцесора. Вмъкването е задача на компилатора.

1.12 Команди на копроцесора

Командите са 69 на брой и имат форматите, представени на фигура 1.12.1.



Фиг. 1.12.1 Формати на командите на копроцесора

Всички команди съдържат в петте старши бита на първия байт кодовата комбинация 11011, с която се отличават от всички останали команди на основния процесор. Същинската операция за копроцесора се кодира в полетата Op1 (p1) и Op2 (p2). Останалите означения на подполета имат следния смисъл:

- Fr = 00 - кодира къси реални числа (32[b]);
  01 - кодира къси цели числа (32[b]) ;
  10 - кодира дълги реални числа (64[b]);
  11 - кодира цели числа (16[b]) .

- R = 0 - кодира запис на резултата във върховия регистър на стека ;
  1 - кодира запис на резултата в друг регистър .
- P = 0 - след операция не модифицира стековия указател ;
  1 - след операция модифицира стековия указател .
- reg - съдържа номер (адрес) на регистър в стека .

Тъй като адресът на операнда се формира от процесора по време на формалния цикъл за четене от ОП, то полетата *mod* и *r/m* тук имат аналогичен смисъл (вижте фигура 1.7.2), с изключение на *mod*=11.

В част от командите типът на операндите се задава от полето *Fr* (*Format*), но операциите върху дълги цели като и временни реални числа се изпълняват от отделни команди. Командите на копроцесора са класифицирани в 6 групи:

1. За прехвърляне на данни. Както между паметта и върховия регистър на стека, така и между регистрите на стека ;
2. За аритметични операции ;
3. За логически проверки ;
4. Изчисляване на стойност на елементарни функции ;
5. Формиране на константни стойности във върховия регистър:

   $0,\ 1,\ \pi,\ \log_{10}2,\ \log_{2}10,\ \log_{2}e,\ \log_{e}2.$

6. За управление на копроцесора – зареждане на управляваща дума, зареждане на състоянието, назначаване на режим (инициализация, прекъсване, грешки).

Програмистът има възможност да управлява точността на закръгляне на резултата за всеки от форматите на мантисата (24, 53 или 64 бита). Схемите за закръгляне са 4:

- В положителна посока (→0→);
- В отрицателна посока (←0←);
- В посока към нулата (→0←);
- В противоположна на нулата посока (←0→).

Резервирани са представления само на безкрайност (∞) или на знакова безкрайност (±∞).

Прочитането на “празен” регистър дава операнд NAN. Тази ситуация се използва за откриване на неинициализирани променливи.

В копроцесора могат да настъпят 6 особени ситуации, които се отразяват в регистъра на състоянието и могат да бъдат причина за генериране на заявка за прекъсване. Чрез съответните битове в управляващия регистър, всяко прекъсване или комбинация от такива, може да бъде маскирано. Когато особената ситуация е маскирана, тя се отработва апаратно, като се формира отнапред определен резултат. Особените ситуации са следните:

1. Недействителна операция. Настъпва при препълване или при изпразване на стека, при неопределена ситуация, например при деление 0 на 0, при изваждане на безкрайност, или при използване на NAN като операнд.

2. Препълване. Фиксира се когато резултатът не може да бъде представен в обявения формат. Когато тази ситуация е маскирана, за резултат се приема знакова безкрайност.

3. Антипрепълване. Това е ситуация, при която резултатът е различен от 0, но е много малък по абсолютна стойност и не може да бъде представен в обявения формат. Ако ситуацията е маскирана, резултатът се фиксира с ненормализирана мантиса.

4. Делител =0. Ако ситуацията е маскирана, за резултат се приема знакова безкрайност.

5. Ненормализиран операнд. Ситуацията настъпва, когато прочетеният операнд се окаже с ненормализирана мантиса. Ако ситуацията е маскирана, на този факт не се обръща внимание и операцията се изпълнява.

6. Неточен резултат. Настъпва, когато резултатът следва да бъде закръглен, за да бъде представен в обявения формат. Закръглянето се извършва според текущата схема за закръгляне. Ако ситуацията е маскирана, изчисленията продължават без закръгляне (мантисата на резултата се "отрязва", т.е. скъсява).

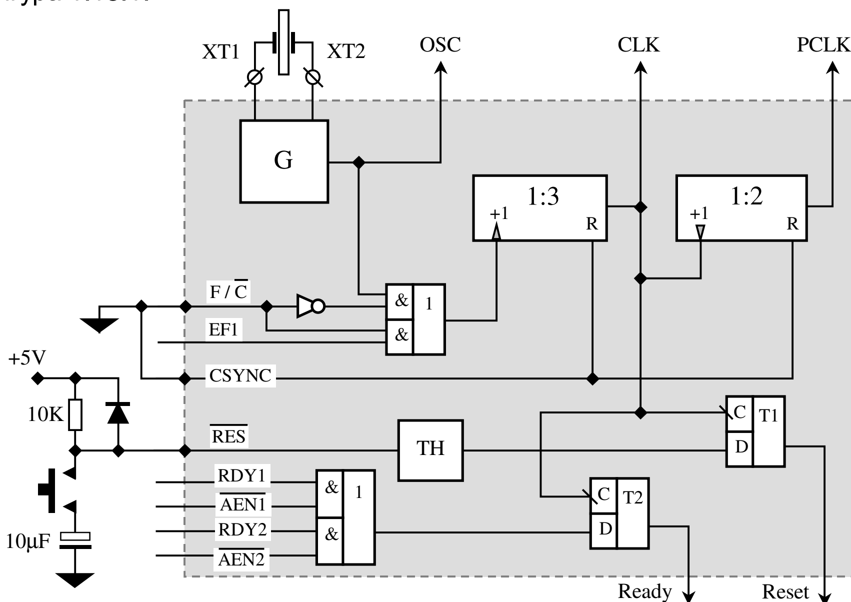
Следва да се отбележи, че ако флагът за особена ситуация в регистъра на състоянието е вдигнат, той си остава вдигнат. Флагът може да бъде свален само от специална команда за презапис на флаговете. Презаписът на флаговете дава възможност след изпълнение на дълга последователност от команди (например след изпълнение на подпрограма), да се провери регистъра на състоянието за настъпили при това особени ситуации.

В заключение следва да се спомене, че производителността на копроцесора за всяка операция е минимум 100 пъти по-голяма в сравнение с програмните еквиваленти, работещи върху основния процесор.

1.13 Схеми за поддържане на системата

За изграждане на реално функционираща компютърна система освен основните 3 устройства – операционно, запомнящо и управляващо, са необходими и други схеми, чиито функции са специализирани и поддържащи като цяло разнообразната функционалност в системата. Тази функционалност включва: общата синхронизация; схеми за поддръжка на динамични запомнящи устройства; схеми за връзка с външния свят чрез системата за прекъсване; схеми за поддръжка на система за реално време; схеми на системата за решаване на вътрешни спорове за общи ресурси и др.

Във връзка с описаната до момента апаратура, тук ще бъде представена единствено схемата на тактовия генератор, показана на фигура 1.13.1.



Фиг. 1.13.1 Логическа структура на тактов генератор i8284

Структурата има 3 синхронизиращи извода:

- Извод OSC – тактова последователност на вътрешния генератор G ;
- Извод CLK – тактова последователност за микропроцесора ;
- Извод PCLK – тактова последователност за синхронизация на периферните устройства.

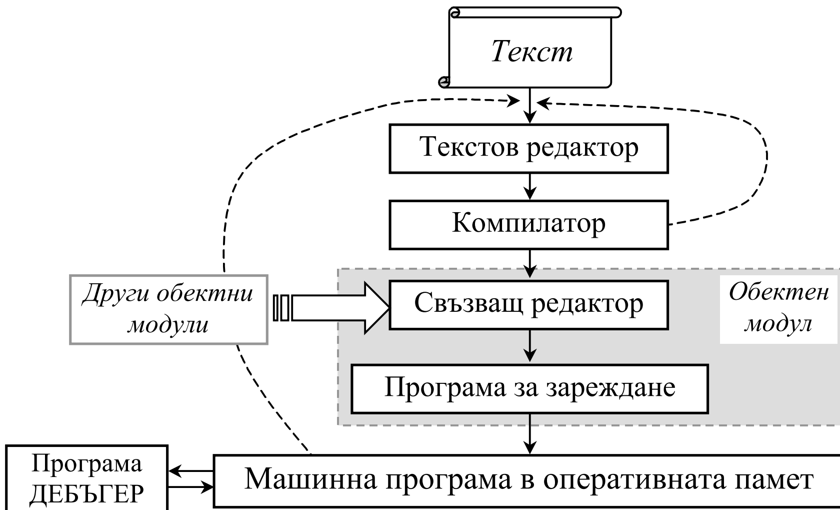
Вътрешният тактов генератор G се комплектова с външен кварцов резонатор в изводите XT1 и XT2. Възможните честоти за различните модели са в границите [12÷25] [MHz]. Освен с вътрешния тактов генератор, схемата може да работи и с външен такъв, включен по входа EF1. С помощта на управляващия сигнал $F/\overline{C}$ вътрешният мултиплексор избира източника на тактова последователност. Тази последователност се пропуска през два делителя – на 3 и на 2. Така се получават другите две синхронизиращи последователности – CLK и PCLK, работещи по преден и по заден фронт съответно.

С помощта на тригера на Шмид TH, външният асинхронен сигнал за рестарт $\overline{RES}$, се фиксира по задния фронт на сигнала CLK от тригер T1. Неговата продължителност е съобразена с изискванията на микропроцесора. Аналогично се генерира и сигналът RDY.

Г Л А В А 2

АСЕМБЛЕРЕН ЕЗИК. НАЧАЛНИ ПОНЯТИЯ

Ефективната реализация на приложни програми, описвани на асемблерен език или на програмен език от високо ниво, изисква програмистът да разполага с подходяща технологична среда (програмна среда) или като минимум да разполага с подходящ набор от помощни програми. На тези помощни програми може да се гледа като на инструменти, които в значителна степен облекчават работата на програмиста. В условията на минимални изисквания те могат да се извикват от командния ред. Така инструментите биват: за създаване на програмния текст, обикновено наричан редактор; за компилиране, т.е. за превеждане на създадения текст на вътрешен машинен език; за връзка с операционната система и за зареждане на програмата като машинна програма, готова за непосредствено изпълнение. Йерархията на тези инструменти е показана на следващата фигура.



Фиг. 2.1 Йерархия на средствата за автоматизация на програмирането

Текстовият редактор позволява да се набира и модифицира програмния текст на създаваната програма. Резултатът от следващата компилация е обектният модул, който обикновено се запомня във вид на файл във външната памет. Терминът “обектен модул” подчертава факта, че програмата може да бъде съставена от няколко части (модули), допускащи самостоятелна компилация. Между отделните модули са възможни връзки – външни обръщения към променливи, определени в други модули. Всички външни обръщения се отбелязват в отделна таблица на обектния модул.

Свързващият редактор обединява отделни обектни модули в един модул за зареждане, в който вече не може да има външни обръщения. Следващият инструмент има задачата да зареди в оперативната памет така подготвения модул и да инициира нейното изпълнение. Главната функция на програмата за зареждане се състои в преизчисление на адресните параметри, които не зависят от текста на програмата, а само от нейното конкретно разположение в адресното пространство. Информацията за преместванията осигурява компилаторът.

И на края, за локализиране на грешките, е необходима програма за настройка, която обикновено се нарича дебъгер.

Да се програмира на машинен език е трудно, дори за системи с много малък набор машинни команди. Основната причина за това е цифровият запис (2-чен или 16-чен) на машинните команди и техните операнди. Ето защо отдавна е въведено средство, което в значителна степен улеснява писането на програмни текстове в системата машинни команди на цифровите процесори. Това средство се основава на подхода на символното представяне на информацията. Така например, когато следва да употребим машинна команда за изпълнение на операция събиране на 2 16-битови числа с фиксирана запетая, чийто машинен код е двоичната комбинация (10000001), вместо това можем да запишем с думи “СЪБЕРИ”. Думата “*събери*” изразява разговорно и разбираемо същността на заповядваната операция и нейната употреба е много по-естествена отколкото запомнянето на съответната й двоична комбинация. В хода на тази логика, вместо машинните двоични комбинации за код на операциите можем да употребяваме техните езикови (символни) означения – “ИЗВАДИ”, “УМНОЖИ”, “РАЗДЕЛИ”, “ПРЕМЕСТИ”, “НУЛИРАЙ”, “ПРОЧЕТИ”, ЗАПИШИ” и т.н. По същия начин, вместо да записваме даден операнд както той се представя в разрядната мрежа, т.е. в двоична бройна система и в допълнителен код за целите числа. Например числото –42, което в една 16 битова разрядна мрежа ще бъде представено от комбинацията 1111111111010110, е възможно и естествено да го запишем като операнд на една команда в неговия естествен вид (например –42), т.е. в десетична бройна система. Веднага може да се направи извода, че такъв програмен текст би се съставял и би се четял много по-лесно. Разбира се непосредствено в символен вид той не би могъл да се зареди и изпълни в цифровия процесор. Необходимо е този разбираем за програмиста символен текст да бъде преведен на вътрешния машинен език. За целта се създава специална програма, за която казваме, че транслира (превежда) символния текст на машинен език. Тази транслираща програма обикновено се нарича “*Асемблер*”.

Следва да отбележим, че за конкретен процесор, с конкретна система машинни команди, могат да съществуват различни асемблерни езици, разработка на различни фирми. Пример за това са асемблерните

езици ASM-86, TASM, MASM за процесорите на *Intel*.

Тук, при описанието на синтаксиса на асемблера, ще бъдат използвани метафизичните формули на Бекус-Наур със следните допълнения.

- В средни (квадратни) скоби ще бъдат указвани конструкции, които не са задължителни и могат да липсват. Така например, записът A[B]C може да означава или текстът ABC или текстът AC.
- В големи (фигурни) скоби ще бъдат указвани конструкции, които могат да бъдат повтаряни неограничен брой пъти, в това число и нито един път. Така например, записът A{BC} може да означава някой от следващите примерни текстове: A, ABC, ABCBC, ABCBCBCBC и т.н.

2.1 Лексеми

Представянето на асемблерния език ще започне с представяне на правилата, по които се записват *лексемите* – определени елементарни конструкции като имена, числа и редове.

2.1.1 Идентификатори

Идентификаторите са необходими за означаване на различни обекти в програмите – променливи, етикети (имена на редове), кодове на операциите и прочие.

В асемблер идентификаторът се определя като последователност от латински букви (големи и малки), цифри и знаци от множеството:

? . @ _ $

При това на тази последователност се налагат следните ограничения:

- Дължината на идентификатора може да бъде произволна, но за значещи се възприемат само началните 31 символа. Идентификатори, които се различават само в 32-я и следващите позиции се считат за еднакви ;
- Идентификаторът не трябва да започва с цифра (записът 7Beta) се приема за грешка ;
- Знакът *точка* може да бъде единствено първи символ на идентификатора (.P е правилно, P. е неправилно) ;
- В идентификаторите едноименните големи и малки букви се възприемат за еквивалентни, т.е. записите AX, Ax, aX и ax означават едно и също нещо.

Особено следва да се подчертае, че в идентификаторите не трябва да се употребяват букви от кирилица или от други азбуки.

Идентификаторите се делят на *имена* и *служебни думи*. Служебните думи имат предварително определен смисъл. Те се използват за означаване на обекти от програмния модел на процесора, като например

регистрите (AX, SI и пр.), имената на команди (ADD, MOV, OR, NEG и пр.) и др. Приложните обекти в програмите потребителят именува по свое желание.

2.1.2 Цели числа

В асемблерния език има възможност за записване на цели и на реални числа.

Целите числа могат да бъдат записани в 10-чна, 2-чна, 8-чна и 16-чна бройна система. Десетичните числа се записват в техния естествен вид, а в останалите бройни системи записите на целите числа окончават със спецификатор – буква, която указва бройната система. Така например записът на двоично число завършва с буква b (*binary*), с буква o (*octal*) или буква h (*hexadecimal*). С цел да се предотврати възможното неразбиране на буква o при осмичните числа, вместо нея се допуска употреба на буква q. От гледна точка на общност записите на десетични числа могат да окончават с буква d (*decimal*), но обикновено тази възможност не се практикува, освен когато цифрите на числото предразполагат читателя на програмния текст към неразбиране на бройната система. Примери:

- Десетични числа: 25, -386, +17, 25d, -255d
- Двоични числа: 10110b, -11011b, +101010b
- Осмични числа: 707q, -203q, +34q
- Шеснадестични числа: A0Eh, -F3h, CD2h

Забележки:

1. Когато се записва 16-чно число, чийто запис започва с цифра, означавана с някоя от буквите A,B,C,D,E или F, то тогава записът на числото напълно съответства на определението за идентификатор. Това естествено може да доведе до неразбиране на текста, тъй като се появява споменатата двойственост. Последната се отстранява лесно, ако се приеме правилото в такива случаи записът да започва поне с една незначеща цифра нула (0A3h вместо A3h).
2. Както при идентификаторите, така и при числата, употребата на големи и на малки букви се отъждествява. За по-голяма прегледност обаче се препоръчва, буквените цифри в 16-чните числа да се изписват с големи букви, а спецификаторите на бройната система – с малки.

2.1.3 Символни данни

Символите и символните низове се ограждат с единични или с двойни кавички: ‘A’ или “A”, ‘A+B’ или “A+B”. Двете кавички трябва да са еднакви – записът ‘A” е неправилен.

Забележки:

- Като символи могат да се употребяват и букви от кирилица;
- В символните низове едноименните символи на големи и малки букви не се отъждествяват, т.е. “A+д” е различно от “a+д”;

- Когато като символ или като символ в символен низ трябва да се представи кавичка, се спазва следното правило: ако символът или символният низ е затворен в единични кавички, то единичната кавичка като символ се представя чрез удвояване (‘don’’t’). Двойната кавичка обаче в този случай не се удвоява (‘don”t’). Ако външните кавички са двойни, то като символ единичната кавичка не се удвоява (“don’t”), а двойната се удвоява (“don””t”). Потребителят за свое улеснение може да избира онзи вариант, при който няма да му се налага удвояване.

2.2 Изречения

Програмите на асемблерен език представляват текст от последователни редове. На всеки ред е записано едно изречение. Не се допуска запис на две и повече изречения на един ред, като пренасяне на дълго изречение на следващ ред. Максималният брой символи в едно изречение (в един ред) е 131.

При изписване на изреченията се допуска следната употреба на празен символ:

- Празният символ е задължителен между два идентификатора или между две числа;
- Там където е допустим един празен символ, са допустими и повече;
- В останалите места празен символ може да има или да няма.

Горните правила не са в сила в границите на символен низ.

По смисъл всички изречения в асемблерния текст могат да бъдат:

- Директиви (декларации към асемблера);
- Команди;
- Коментари.

2.2.1 Коментари

Коментарите не променят алгоритъма и не влияят на изчислителния процес. При транслиране на програмите те се игнорират. Коментарите са предназначени за читателите на програмния текст и подпомагат смисловото изясняване на отделните части от алгоритъма (програмата). Изречение, което се определя като коментар, следва да започва със символа “*точка със запетая*” (**;**). Преди нея може да има празни символи, дори празен ред. В коментарите могат да се употребяват всякакви символи. Например:

```
; следващият ред съдържа команда за събиране
    ADD  AX, 0
; команда за зареждане на регистър B (BX:=2)
    MOV  BX, 2
```

Когато коментарът е дълъг текст, който заема няколко реда, той може да се декларира (да се обяви) като такъв чрез директивата COMENT. Тази директива има следната структура:

COMENT *<маркер> <текст>*

В качеството на маркер се приема първият символ, различен от празния. За край на текста на така обявен коментар се приема края на реда, в който в произволно място се среща същият маркер. Например:

```
COMENT  // подпрограма ABC е разработена от инж. Иван Иванов.
           Тя е предназначена да ...
           Нейните входни параметри са ...     //
```

Този вид коментар може да се използва за бързо изключване (деактивиране) на определен участък (фрагмент) от текста на програмата.

2.2.2 Команди

Командите на асемблерния език, които съответстват на машинните команди, се изписват като изречения в отделни редове. Синтаксисът на този вид изречения се дефинира както следва

[<етикет>:] <мнемоничен код> [<операнди>] [;<коментари>]

Примери:

```
begin:   ADD  SI, 2   ; изменение на индекса
         NEG  A
         CBW
```

Етикет

Според определението, етикетът по същество представлява име или още име на входна точка. След етикета трябва да има двоеточие. Като име на входна точка етикетът се използва от командите за управление на прехода. Така например, една команда за условен преход, която съдържа етикетът *begin*, ще предаде управлението на командата ADD, тъй като тя е маркирана с това име (с този етикет или маркер). Ясно е, че в текста на програмата етикетите са уникални и се срещат само по един път. Груба грешка е две различни команди да са маркирани с един и същи етикет.

Асемблерният език допуска изречения, които съдържат само един етикет. С това се цели подобряване на читаемостта на програмата. Въпреки че етикетът е записан на отделен ред, той се разбира като име на непосредствено следващата команда. Този начин на записване позволява една и съща команда да бъде маркирана с няколко различни етикета.

Етикетът не бива да съвпада с някой от мнемокодовете и командите на Асемблера, както и с имена на регистрите на процесора: AH, AL, AX, BH, BL, BX, CH, CL, CX, DH, DL, DX, DI, SI, SP, BP, CS, DS, ES, SS.

В етикета не може да има интервал. Вместо него използвайте подчертаващо тире. Например:

```
          ; Inicialization:
               Begin:  ADD  BX, AX
      GET_COUNT:  MOV  CX, DI
```

Мнемоничен код

Мнемоничният код на командата представлява задължителен елемент в изречението. Това е служебна дума, указваща по смисъл заповядваната машинна операция. От своя страна именуването на командите по смисъла на техните операции позволява тяхното лесно запомняне и разбиране (*mnemonic* – лесно запомнящ се). Мнемоничните кодове на операциите в даден процесор, като служебни думи на асемблерния език, се декларират обикновено от производителя на транслатора.

Операнди

Операндите на командите, ако такива има, се разделя със запетая. Операндите могат да бъдат записвани в рамките на изрази, което ще бъде пояснено по-късно. За сега ще се имат предвид изрази, в които участват числа и имена на променливи. За разлика от машинния език, в командите на асемблера не се указват адресите на операндите, а техните идентификатори. Разпределението на операндите, т.е. на данните, по адреси в оперативната памет, не е грижа на програмиста. Така на имената на променливите може да се гледа като на означения на съответните им стойности.

Мнемоничните кодове на командите, на разглеждания тук микропроцесор, ще бъдат представени по хода на разглежданите в книгата теми.

Коментари

В края на изречението, след командата, може да бъде поставен коментар. Този коментар трябва да се предхожда от символа точка със запетая (;) и цели да поясни същността и изпълнението на командата.

2.2.3 Директиви

Използваните от програмиста константи и имена на променливи трябва да бъдат декларирани (описани) по смисъл и стойност. Това е естествено да бъде направено преди текста на програмата, описващ тяхната обработка. За целта в асемблерния език се дефинират изречения, наречени директиви. Синтаксисът на директивите е следния

[<име>] <означение на директивата> [<операнди>] [;<коментар>]

Пример:

```
Xmax  DB  10, -5, 0FFh      ;  Xmax  е работен масив от 3 елемента
```

Както може да се види, форматът на директивата като цяло съвпада с формата на командата. Единствената разлика се състои в това, че след името (ако го има) не се поставя двоеточие. Името, което се поставя в началото на директивата, като правило е име на променлива или константа. Означението на директивата (DB в примера по-горе) представлява декларация за дължината на двоичното поле, в което се представят стойностите на обявените в директивата константи и променливи.

Означенията на директивите, както и мнемокодовете представляват служебни думи на езика. При това отнапред е известно кои служебни думи означават директиви, и кои – мнемокодове. Отделните директиви ще бъдат пояснени по хода на изложението.

Останалите части на директивата (операнди и коментар) се записват както и в командите.

2.2.4 Обръщения назад и напред

Необходимо е да бъдат направени следните няколко забележки:

На първо място, етикетите (маркерите) на командите и имената на променливите, константите и пр., които се указват в директивите, по същества са *различни* неща както по смисъл, така и по ред формални признаци. В общия случай обаче етикетите могат да се възприемат като имена на команди. Ето защо по-нататък зад думата име ще се разбира както име на променлива, така и етикет.

На второ място, появилото се в началото на една команда или директива име, се приема за описание на изречението. В асемблерния език действа следното правило: всяко име трябва да бъде описано само един път, т.е. в програмата не трябва да се срещат две изречения с едно и също име. Това правило обаче има изключения, които ще бъдат пояснени по-късно.

На трето място, ако в езици от високо ниво действа правилото "*първо опиши и след това използвай*", то в асемблерния език такова правило няма, към името могат да се правят обръщения както преди неговото описание, така и след това. Пример за това са следните два текста

```
AA:   DB  2      <-┐
      …….          │  обръщение назад
      NEG  AA    --┘

      NEG  AA    --┐
      ……           │  обръщение напред
AA:   DB  2      <-┘
```

За да се различават тези два случая, са въведени изказванията – обръщение напред и обръщение назад. В първия случай обръщението е към име, описано назад (по-нагоре, преди това) в текста на програма-

та, а във втория случай – към име, което е описано в текста на програмата по-надолу.

При транслиране текстът на програмата се обработва отляво надясно и отгоре надолу. Когато бъде срещнато обръщение назад към име, което вече било описано, тъй като за него вече е регистрирана необходимата информация, то даденото обръщение ще бъде транслирано правилно. Когато обаче се срещне обръщение напред към име, за което още нищо не се знае, то такова обръщение не винаги може да се транслира правилно и по тази причина са възможни проблеми. Въпреки че като цяло асемблер допуска обръщения напред, то те не се препоръчват като добър стил за програмиране.

2.3 Директиви за описание на данните

Както беше отбелязано, данните, представени от своите имена, трябва да бъдат описани. Описанието се отнася до техния формат. В този смисъл директивите не се различават съществено помежду си.

2.3.1 Директива DB

Тази директива определя за формàта на името дължина 1 байт. (DB – *define byte*). Синтаксисът на директивата е следния

[<име>] DB [<операнд>] {,<операнд>} [;<коментар>]

Срещайки такава директива, асемблерът формира машинния вид на числата и ги записва в последователни клетки на паметта. В директивата данните се задават по два начина:

- Променливи с неизвестна (неопределена) стойност;
- Константите имат стойност в интервала [-128, 255]. Имат се в предвид числа със и без знак;

В първия случай, за дадена променлива може да се декларира както неопределена, така и определена (известна) стойност. Когато стойността на променливата е неизвестна в началото на изчисленията, нейната стойност се представя чрез въпросителен знак (?). Например:

Xmax DB ?

Така може да се твърди, че като операнд е записан въпросителен знак. За променливата се отделя един байт, а съдържанието му като клетка в паметта е без значение. Тъй като при транслиране на асемблерския текст на програмата машинният код се записва в последователни клетки, то директивите не трябва да се поставят между командите, тъй като отделените за променливите байтове ще объркат хода на изчислителния процес, няма да могат да се разпознаят като машинни команди и ще причинят генериране на изключителни събития.

Така разпределената в паметта променлива се идентифицира с физическия адрес на клетката, а не с нейното съдържание! Само при

алгоритмична интерпретация на името на променливата може да се говори за нейната стойност.

В асемблерния език освен пояснените вече изречения, съществуват още изречения, които се наричат оператори. Операторите ще бъдат дефинирани по хода на изложението, а тук е мястото да се дефинира операторът за тип. Този оператор има вида:

TYPE <име>

В зависимост от името, операторът за тип определя дължината, т.е. формàта (в брой байтове) на полето за представяне на стойностите на променливите. В асемблер е определена стандартна константа с ключовото име байт (BYTE), която има стойност единица. Така променливата X може да се декларира както следва

TYPE X = BYTE =1

Във втория случай, когато операнд на директивата е константа, директивата описва променлива, която има тази начална стойност. Например:

```
AA1  DB  254   ; стойността на променливата AA1  е  0FEh
AB2  DB  -2    ; 0FEh  (изразява допълнителния код 256-2=254)
CC   DB  017h
```

За всяка от горните директиви асемблер записва в последователни клетки стойности на съответните променливи: $AA1=11111110_{(2)}=FE_{(16)}$; $AB2=11111110_{(2)}=FE_{(16)}$; $CC=00010111_{(2)}=17_{(16)}$; Необходимо е да поясним, че за първата променлива стойността се интерпретира като число без знак, а за втората като число със знак в допълнителен код. Двоичните комбинации фактически съвпадат, но интерпретацията е за сметка на програмиста !

Освен за задаване на числена стойност, директивата може да се използва и за задаване на символна стойност. В този случай стойността се загражда с единични или двойни кавички:

```
QQ  DB  2Fh    ; числена стойност 2F(16)
QQ  DB  "/"    ; символна стойност (символ /)
```

За задаване на символни стойности втората директива е по-удобна. Стойността (кодът) на символа, като начална стойност на обявената променлива, се определя според ASCII-таблицата.

Операторът за тип също има възможността да задава начална стойност на променлива. Така например, ако в програмата е вече употребена директивата за описание на променливата QQ, то с друга директива можем да присвоим същите характеристики на друга променлива, по следния начин:

Ver22 DB TYPE QQ

Директива с няколко операнда

Освен скаларните променливи съществуват още и променливи от тип масив. Описанието на масива е възможно да стане и с отделни

директиви за всеки негов елемент. Например, нека опишем 4-елементния масив MM, чиито елементи са еднобайтови, със съответните начални стойности:

```
MM    DB  2
      DB  -2
      DB  ?
      DB  "%"
```

Както се вижда, името на масива MM се дава само на първия елемент, останалите остават неименувани. Явно е, че при много на брой елементи, този начин за описание не е удобен. Ето защо в асемблерния език се допуска запис на много операнди. Съответно за дадения пример, такава директива ще има вида:

MM DB 2, -2, ?, "%"

Указаните в списъка стойности, с изключение на третата, се записват в последователни клетки (еднобайтови). Типът на името на масива съответно е: TYPE MM = BYTE. Обръщение към първия елемент на масива се реализира чрез неговото име (MM), а към следващите елементи – чрез израз от вида:

<име на масив> ± d,

в който е указано отместването d. Например чрез израза MM+1 се посочва вторият елемент на масива, който има стойност –2. Изразът MM+1 не означава, че се променя стойността на елемента. Този израз определя мястото на елемента в паметта, т.е. адреса на елемента в паметта, където се намира неговата стойност.

Директива с операнд във вид на изречение

Когато директивата има за операнди символи, то е възможно тяхното обединяване във вид на символен низ. Например, следните два записа са еквивалентни:

```
Smin  DB  "a", "b", "c"          Smin  DB  "abc"
```

Типът на името Smin е 1 байт (TYPE Smin=BYTE), тъй като всяка от горните директиви представлява съкращение на следните три директиви:

```
Smin  DB  "a"
      DB  "b"
      DB  "c"
```

от които се вижда, че описаното име е еднобайтно.

Въпросът кои символи да бъдат обединени, и в колко низа – е въпрос, който решава програмиста. Ограничения няма.

Директива с конструкция DUP

Възможно е още едно съкращение на записа на директива DB. Тя е удобна в случаите, когато операндите се повтарят. Например, когато се описва масив Rmas от 8 еднобайтови елемента, на които се задава начална стойност нула, директивата може да има следния вид:

Rmas DB 0, 0, 0, 0, 0, 0, 0, 0

Тази директива може да бъде съкратена с помощта на служебната дума DUP (*duplicate* - копие) и поставен в скоби операнд:

Rmas DB 8 DUP(0)

Както се вижда от записа, служебната дума DUP се предхожда от коефициента на повторение (в примера 8). Общият вид на конструкцията за дублиране се дефинира така

$$k \ \ DUP \ \ (p_1,p_2,p_3,\ldots,p_n)$$

където с k е означен броят на повторенията, които се реализират върху списъка в скобите. Така според определението, се повтаря k пъти:

$$p_1,p_2,p_3,\ldots,p_n,\ p_1,p_2,p_3,\ldots,p_n,\ p_1,p_2,p_3,\ \ldots\ ,\ p_{n,\ \ldots}\ p_1,\ \ p_2,p_3,\ldots,p_n$$

Примери:

X1 DB 2 DUP(“ab”,?,1) еквивалентно на: X1 DB “ab”,?,1, “ab”,?,1

Y1 DB –7, 3 DUP(0, 2 DUP(?)) ≡ Y1 DB -7, 0, ?, ?, 0, ?, ?, 0, ?, ?

Вложеността на конструкцията DUP позволява лесно и нагледно описание на многомерни масиви. Например директивата:

MAA DB 20 DUP(30 DUP(?))

Представлява описание на матрицата MAA(20,30) имаща 20 реда и 30 колонки, елементите на която не получават начални стойности. В паметта елементите са подредени по редове – първите 30 клетки съдържат еднобайтовите елементи на първия ред на матрицата. Следващите 30 – на втория ред, и т.н.

2.3.2 Директива DW

Директивата DW (*define word* – определи дума) описва променливи с дължина дума (16 бита). Конструкцията ѝ и правилата на нейното използване са идентични на директивата DB. Тук ще бъдат описани само допустимите ѝ операнди.

Операнд ?

Възможен е следният пример: A DW ?

Според тази директива асемблерът отделя в паметта за променливата A два байта без да променя тяхното съдържание. Променливата има тип 2 (TYPE=2). В асемблера съществува стандартна константа с ключовото име WORD, която има стойност 2. Този факт може да се запише както следва:

TYPE A = WORD = 2

Операнд константа със стойност в интервала [-32768, 65535]

Възможни са следните примери:

BB DW 1234h
CC DW -8

Според тези две директиви, за променливите BB и CC се определят клетки с дължина по 2 байта и в тях се записва двоичният еквивалент

на декларираните начални стойности. Числата се представят в допълнителен код. Числата, които по стойност са по-големи от 32767, могат да се интерпретират правилно единствено от програмиста (евентуално като числа без знак от 0 до 65535). И още една подробност – когато числата са с дължина по-голяма от един байт, те се разполагат в паметта с младшия байт напред. Така подравнените по четен адрес данни изглеждат с разместени части. Това обаче се отчита автоматично от асемблера и не е грижа на програмиста при запис или четене.

Частен случай на разглеждания пример е случаят, когато операндът е низ от един или два символа:

```
S1   DW   "01"
S2   DW   "1"
```

Когато операндът е низ от два символа, то на променливата се присвоява конкатенираната от двата ASCII кода последователност (30h≡"0", 31h≡"1" ; S1=3031h). След образуването на тази двубайтова последователност тя се записва в паметта според указания ред – с младшия байт напред. Когато в директивата е указан само един символ, той се допълва отляво с нула (S2=0031h).

Разместването на кодовете на байтовете при записване на символните низове в паметта може да доведе до объркване от страна на програмиста, ето защо декларирането на символни низове с тази директива се избягва.

Операнд със стойност адрес

Като операнд в директивата DW може да се употреби израз, чиято стойност има смисъл на адрес на клетка в паметта. В повечето случаи това са имена на променливи или на етикети. Ето защо са възможни следните примери:

```
C  DW  ?
D  DW  C
```

С тези две директиви променливата D получава за стойност адреса на променливата С.

Няколко операнда, конструкция за повторение

В дясната част на директивата DW могат да бъдат описани в списък няколко операнда, включително и конструкцията за повторение DUP. Възможен е следният пример:

```
ESS   DW   2008, 12 DUP(?)
```

2.3.3 Директива DD

С директивата DD (*define double word*) се описват променливи, за които е необходим формат с двойна дума, т.е. 32 бита. Такива променливи имат тип 4, който се означава със стандартната константа DWORD, имаща стойност 4. Във всичко останало тази директива е аналогична на вече описаните. Допустимите за тази директива

операнди са:

Операнд ?

Например:

AA DD ? За стойностите на променливата AA директивата заделя 4 байта в паметта, без да определя начална стойност.

Операнд константа със стойност в интервала $[-2^{31}, 2^{32}-1]$

Пример:

BAC DD 123456h

За променливата BAC се отделят 4 последователни байта в паметта. В тях се записва като начална стойност двоичния еквивалент на указания 16-чен операнд,

0000 0000 0001 0010 0011 0100 0101 0110

който се записва с младшите байтове напред, т.е. в съответствие със следната схема:

Адрес	Съдържание
m-2	
m-1	
BAC≡m	**01010110**
m+1	**00110100**
m+2	**00010010**
m+3	**00000000**
m+4	
m+5	

Променливата BAC заема 4 клетки в паметта, а нейното име се отъждествява с началния адрес m.

Константен израз със стойност в интервала $[-2^{15}, 2^{16}-1]$

Тук максималната стойност е два пъти по-малка. Проблемът е в това, че стойностите, които може да изчисли асемблерът са във формàта на 16-битовата дължина, т.е. резултатите се вземат по модул 2^{16}, (10000h). Ето защо да се разчита на по-дълги резултати не е възможно. Единственото изключение е явното задаване в директивата DD "голямо" число. Ако обаче бъде описана поне една операция, резултатът ще бъде орязан във формат 16 бита. Така например по силата на директивата

X DD 8000h + 8002h

началната стойност на променливата X ще бъде числото 2, а не числото 10002h.

Адресен израз

Операнд от този вид задава абсолютен адрес. Подробностите ще бъдат изложени в последствие.

Няколко операнда и конструкция за повторение

За разглежданата директива е възможен следният пример:

W DD 33 DUP(?), 12345h

2.4 Директиви за еквивалентност и присвояване

Тук ще се разгледа представянето на константите. За целта се използва директивата EQU (*equal* – равно), която има следния синтаксис:

<име> EQU <операнд>

В тази конструкция името и операндът са задължителни елементи, при това операндът трябва да е само един.

С тази директива програмистът обявява, че на указания операнд се присвоява указаното име, което навсякъде в програмата носи указания операнд. Например, след директивата:

Star EQU "*"

асемблерът ще разглежда декларацията

T DB Star

като декларация

T DB "*"

В резултат се образува еквивалентност и дали ще се използва името Star или "*" е без значение.

Директивата EQU има чисто информационен смисъл, без задаване на стойност, ето защо тази директива може да се употребява на всяко място в програмата.

Операнд име

Ако в дясната част на директивата е указано име на регистър, на променлива, или константа, то името в началото на директивата става синоним на операнда и при всички последващи срещи на това име асемблерът ще го подменя с името на операнда. Например:

```
A DW ?
B EQU A
C DW B          ; еквивалентно: C DW A
```

Синонимните имена се въвеждат като по-удобни и разбираеми. Например, само по себе си името на регистър AX нищо не ни говори, но ако този регистър се използва за изчисляване на някаква сума, то той може да бъде преименуван с името SUM чрез следната директива:

SUM EQU AX

което позволява по-нататък в програмата да употребяваме по-смисленото за алгоритъма име SUM, вместо AX.

Необходимо е да се отбележи още, че посоченото в дясната част на директивата EQU име, може да бъде описано в програмата както до, така и след разглежданата директива.

Операнд константен израз

За този случай могат да бъдат дадени следните примери:

```
   N  EQU  100
   K  EQU  2*N-1
Star  EQU  "*"
```

Ако в дясната част на директивата EQU се намира константен израз, то записаното в ляво име представлява име на константата. Стойността на израза е стойност на тази константа.. Например, името N е константа със стойност 100, K – със стойност 199, а Star – със стойност 2Ah (според ASCII таблицата). При всички следващи обръщения към указаните имена, същите ще бъдат замествани със съответните константни стойности. Така името N, употребено в директивата:

```
X  DB  N  DUP(?)
```

я прави еквивалентна на следната:

```
X  DB  100  DUP(?)
```

Случаите, когато е удобно да бъдат използвани константи, са аналогични на тези в езиците от високо ниво. Например, като константи е удобно да бъдат обявени размерите на масивите, тъй като тогава лесно може да се настрои програмата за работа с масиви, имащи други размери, за което е достатъчно да се внесе изменение само в директивата EQU, която описва дадената константа.

Ако в константен израз са употребени имена на други константи, то последните трябва да са описани като такива преди това, тъй като асемблерът, който проследява текста на програмата отгоре надолу, не би могъл да изчисли исканата стойност.

Операнд – произволен текст

Примери:

```
 S   EQU   "Вие сте сгрешили"
LX   EQU   X+(N-1)
WP   EQU   WORD  PTR
```

В този случай се приема, че употребеното име указва операнд в онзи вид, в който е записан (операндът не се изчислява). Именно с този текст ще бъде подменяно даденото име, при всяко влизане в програмата. Например, следващите в дясно изречения, са еквивалентни на тези в ляво:

```
ANS  DB  S, "1"          ANS  DB  "Вие сте сгрешили","1"
     NEG  LX                  NEG   X+(N-1)
     INC   WP  [BX]           INC   WORD  PTR  [BX]
```

Такъв вариант на директивата EQU обикновено се използва за да се въведе по-кратки означения за често срещащи се дълги текстове. Въвеждайки кратко име, асемблерът ще го подменя със съответния текст автоматично. Ще отбележим, че текстът в дясната част на директивата EQU, трябва да е балансиран по отношение на скобките и кавичките и да не съдържа извън скобките и кавичките символа “;”. Освен това, тъй като текстът не се изчислява, в него могат да се използват имена, които са описани преди директивата EQU, а така също и имена, описани след нея.

Директива за присвояване

Директивата за присвояване има следната конструкция:

<име> = <константен израз>

Тя определя (описва) константа с име, стоящо в лявата част, което получава числената стойност на израза отдясно. За разлика от константите, описани с директивата EQU, тази константа може да променя стойността си. Например:

```
K=10
A   DW   K      ; еквивалентно на A  DW  10
K=K+4
B   DW   K      ; еквивалентно на A  DW  14
```

Подобни константи могат да се използват “за икономия на имена” – ако в различни части на програмата се използват различни константи и областите на видимостта им не се пресича, тези константи могат да имат едно и също име. Главното достойнство на тези константи обаче се проявява на ниво макроси.

Ако с помощта на директивата EQU е възможно да се определи име, което да означава не само число, но и други конструкции, то директивата за присвояване определя единствено числова константа. Освен това, ако името е указано в лявата част на директивата EQU, то не може да се появява в лявата част на други директиви, т.е. то не трябва да се преопределя. В същото време, името в лявата част на директивата за присвояване може да се появява в началото на други такива директиви.

Появяването на константи, които могат да променят своята стойност, може да внесе известна неопределеност. Например според следния текст:

```
K=1
N   EQU   K
A   DW   N        ; A=1
K=2
B   DW   N        ; B=2
```

какви ще бъдат началните стойности на променливите A и B?

Стойността на A е безспорно 1. Стойността на B обаче ще бъде 2, а не 1. Това е така, защото K е синоним на N и след преопределянето на K се преопределя (подменя) и стойността на N, която пък задава началната стойност на B в последния ред на горния фрагмент.

Според следващия текст:

```
K=1
N   EQU   K+10
C   DW    N          ; C=11
K=2
D   DW    N          ; D=11
```

променливите C и D получават една и съща стойност 11. Тук в директивата EQU е указан истински константен израз, чиято стойност асемблерът изчислява веднага. Стойността на този израз е 11 и тя се обявява за стойност на константата N. Така D получава същата стойност.

Ето защо е необходимо да уточним действието на директивата EQU: ако в дясната ѝ част е указано име на константа, то името отляво следва да се възприема не като име на константа, а като синоним на името отдясно. Ако пък в дясната ѝ част е указано име на всеки друг константен израз, тогава името отляво действително става име на константа. Що се отнася до директивата за присвояване, то нейната дясна част винаги се изчислява веднага и резултатът става новата стойност на константата.

2.5 Изрази

Операндите на директивите, като правило, се описват като изрази. Изразите се използват и за описание на операндите на командите. Изразите приличат на тези в езиците от високо ниво, но имат и различия. Най-съществената разлика се състои в това, че стойностите на изразите в асемблера се изчисляват по време на компилацията на програмата, а не по време на нейното изпълнение. За разбиране на тази разлика тук ще припомним, че резултатът от компилацията е машинна програма (последователност от машинни команди, в текста на която няма място за изрази). Във връзка с това, в асемблерните изрази могат да се употребят само величини, които по време на транслирането имат известни стойности. В обратния смисъл, в изразите не могат да се употребят величини, които ще получат стойност по време на изпълнение на програмата.

В асемблера, според типа на стойността си, изразите се делят на два вида – константни и адресни. Ако стойността е цяло число, изразът се нарича константен. Ако стойността е адрес – адресен. Разбира се, адресът е цяло число, но с друго предназначение, т.е. от друг тип. Съществува и други изрази, които се използват за записване на операнди на командите и директивите, и които не се отнасят към вече

споменатите. Те се отнасят най-вече за имената на регистрите.

В асемблера се използват и така наречените оператори. Операторите се определят като едноместни и двуместни, в зависимост от броя на аргументите си. Чрез операторите, от прости изрази се градят по-сложни изрази.

Операторите имат приоритет, който по-долу е представен в низсходящ ред. Операторите с еднакъв приоритет са изписани в реда:

1. ( ), [ ], LENGTH, SIZE, WIDTH, MASK
2. **.**
3. **:**
4. PTR, OFFSET, SEG, TYPE, THIS
5. HIGH, LOW
6. едноместни + и -
7. *, /, MOD, SHL, SHR
8. двумвестни + и -
9. EQ, NE, LT, LE, GT, GE
10. NOT
11. AND
12. OR, XOR
13. SHORT, TYPE

Операторите с еднакъв приоритет се изчисляват отляво на дясно. Например: A+B-C в реда според скобите: (A+B)-C.

2.5.1 Константни изрази

Стойностите на константните изрази са винаги 16-битови цели числа (единствено изключение прави директивата DD, в която явно може да бъде посочено 32-битово число). При това, ако стойността (в алгебрически смисъл) на константния израз представлява отрицателно число, в машинната програма тя се представя от допълнителния код на числото.

Към простите константни изрази се отнасят:

- Числа в интервала [-2^{15}, 2^{16}-1]. Това са всички възможни за този формат отрицателни числа, т.е. от –32767 до –1.
- Символ (стойността на символ”0” ще бъде числото 30h).
- Два символа (символният низ “01” е числото 3031h).
- Име на константа (името носи определената стойност).

По-късно ще бъдат разгледани и други прости константни изрази.

Сред константните оператори (с числови стойности) засега ще споменем оператор TYPE, както и посочените по-долу:

- Едноместни плюс и минус: +k, -k.
- Оператори събиране и изваждане: k1+k2, k1-k2.
- Оператори умножение и деление: k1*k2, k1/k2, k1 MOD k2.

Ето пример за използване на константен израз с аритметически оператори:

```
K   EQU   30
X   DB   (3*K-1)/2   DUP (?)     ; масив от 44 байта
```

Операндите на аритметическите оператори трябва да бъдат константни изрази. Тук обаче има едно изключение, според което асемблерът разрешава изваждане на един адрес от друг, в резултат на което разликата се приема за число, а не за адрес. Например:

```
        X   DW   1, 2, 3, 4, 5
        Y   DB   ?
SIZE_X   EQU   Y-X              ; SIZE_X = 10
```

Изваждането на адреси се налага когато е необходимо да се определи разстоянието между тези два адреса (в брой байтове). В горния пример разликата Y-X определя броя байтове, които заема масивът X.

Всички аритметически оператори се изчисляват по модула на 16-битовата разрядна мрежа 2^{16} (10000h), т.е. за резултат се вземат винаги младшите 16 бита. Така например:

(2*9000h)/100h = (12000h)/100h $\Rightarrow$ 2000h/100h = 20h (а не 120h)

2.5.2 Адресни изрази

Стойности на адресните изрази са 16-битови адреси. Всички операции се извършват по модула на разрядната мрежа 2^{16} (10000h). Прости адресни изрази са:

- Етикет (маркер) и име на променлива, описано в директива DB, DW или DD (стойността на такива изрази е адрес).
- Брояч за поместване. Записва се със символа $ и означава адрес на онова изречение, в което е срещнат. При транслиране на програмата асемблерът следи адреса, в който ще попадне машинният еквивалент на поредното изречение от асемблерската програма. Ако е необходимо обръщение по този адрес, мястото следва да се отбележи с този символ $. От тук следва, че в различни изречения този символ означава различни адреси. Например, ако адресът на променливата A е 100h, то:

```
A  DW  $        ; е еквивалентно на   A  DW  100h
B  DW  $        ; е еквивалентно на   B  DW  102h
```

Най-често броячът за поместване се използва за изчисляване на размера на паметта, съдържаща някакъв масив. Например:

```
         X  DW  40  DUP(?)
SIZE_X  EQU  $-X              ;  SIZE_X = 80
```

Тук символът $ означава адреса на първия байт след масива X. Така разликата ($-X) представлява обема памет в байтове, който заема масива X.

Измежду операторите, чиито стойности представляват адреси, тук ще отбележим само операторите събиране и изваждане (a – адресен израз, k – константен израз):

- Събиране на адреса с константа: (a+k), (k+a) – сумата е адрес;
- Изваждане на константа от адрес: (a-k) – разликата е адрес;

Адресните изрази от вида <име> ± <цяло число> се използват за обръщение към клетки във вътрешността на масиви, т.е. към елементи от масив. Например, ако имаме описанието:

```
A  DB  10,11,12,13
B  DB  14
```

то именувани се оказват само първият байт от масива A, в който има стойност 10 и 5-тият поред байт, в който се намира стойността на променливата B (B=14). Останалите байтове от масива A са безименни и към тях не можем да се обърнем по име. За достъп до тези безименни клетки асемблерът използва адресни изрази от указания по-горе вид. Например, обръщение към байта, съдържащ стойността 13, е възможно само чрез адресния израз (A+3) или чрез израза (B-1).

Ще отбележим, че в асемблерния език е забранено изваждането на адрес от число. Не се допуска още събиране на два адреса, умножение на два адреса и деление на два адреса. Такива операции са безсмислени. Допуска се обаче изваждане на адреси, но резултатът е константа, а не адрес.

И още една забележка – в асемблерния език не е възможно явно указване на адрес, т.е. изписване във вид на число. Ако е указано число, то винаги се възприема като константа. Ако все пак е необходимо посочване на адрес във явен вид, то това е възможно по специален начин, който ще изясним по-късно.

ГЛАВА 3
ПРЕХВЪРЛЯНЕ НА ДАННИ. АРИТМЕТИЧЕСКИ КОМАНДИ

3.1 Обозначаване на операндите на командите

При представяне на командите ще ни бъде необходимо да поясним кои операнди са допустими и кои не са допустими. За да съкратим този вид пояснения, ще използваме следните означения:

Операндът се намира	***Означение***	***Запис в Асемблер***
В командата	i8, i16, i32	Константен израз
В регистър с общо предназначение	r8, r16 (R8, R16)	Име на регистър
В сегментен регистър	sr, (SR)	CS, DS, SS, ES
В клетка от ОП	m8, m16, m32	Адресен израз

Непосредствените операнди, които се записват в самата команда, ще се означават с буквата **i** (*immediate* - непосредствен) последвана от цяло число, показващо дължината на формàта на операнда в битове (8, 16, 32). В Асемблер непосредствените операнди се записват като константни изрази.

Регистрите се означават с буква **r** или **R** (*register* - регистър) последвана от цяло число, показващо дължината на формàта в битове. Например, като 8-битови регистри (r8) ще имаме предвид означенията AH, AL, BH, BL и пр., а като 16 битови (r16) – означенията AX, BX, SI и пр. Буквата **r** ще бъде употребявана само за означаване на регистри с общо предназначение. Сегментните регистри ще означаваме с буквите sr или SR.

Ако операндът се намира в оперативната памет (ОП), то в командата се записва адресът на съответната клетка. Такива операнди ще се означават с буквата m (*memory* - памет), последвана от цяло число, показващо дължината на клетката в битове. В Асемблер такива операнди се задават чрез адресни изрази.

3.2 Команди за прехвърляне на данни

Системата машинни команди има голям набор от различни команди за прехвърляне на данни от едно място в друго. За сега ще се спрем само на две: MOV и XCHG. Освен това ще разгледаме оператор PTR.

3.2.1 Команда MOV

Могат да се прехвърлят 8-битови и 16-битови операнди. 32-битови не могат да се прехвърлят с една единствена команда. Съдържанието, което се мести може да бъде взето от самата команда, от регистър или от паметта. То може да бъде прехвърлено и записано в регистър или в клетка от паметта. Ще припомним, че в регистъра старшата (H) и млад-

шата (L) половини на едно число са подредени в прав ред, а в паметта двете части на числото се подреждат в обратен ред. Този ред се съобразява от действията на командата автоматично, когато тя се използва за прехвърляне на данни в регистър или в клетка от паметта.

Въпреки че машинните команди за тези действия са много, в Асемблерния език те се записват по един и същи начин. Така че на това ниво програмистът може да счита, че разполага само с една команда за прехвърляне на данни. Тълкуването на символиката на тази асемблерна команда се извършва при транслирането й, при което тя се превръща в необходимата машинна команда.

Командата има следната структура и действие:

```
MOV   op1,  op2         ; op1 := op2
```

където op1, op2 означават операнд1 и операнд2.

Примери:

```
MOV   AX, 500      ; AX := 500  в регистър AX се записва числото 500
MOV   BL, DH       ; BL := (DH)  в регистър BL се записва
                                   съдържанието на регистър DH.
```

Командата допуска следните комбинации за своите операнди:

op1 ←	*op2*
r8	i8, r8, m8
m8	i8, r8,
r16 (без CS)	r16, m16
m16	i16, r16, sr

От таблицата се вижда, че не са възможни прехвърляния от една клетка на ОП в друга, от един сегментен регистър в друг и запис на непосредствен операнд в сегментен регистър. Ако в последния случай такъв запис се налага, той се осъществява с две последователни прехвърляния:

1. От командата в регистър: MOV AX, 100 ; AX:=100
2. От регистъра в сегментен регистър: MOV DS, AX ; DS:=(AX)

Ще отбележим още, че с команда MOV не може да се мени съдържанието на сегментния регистър CS, което би било равносилно на преход. Команда MOV <u>не променя</u> съдържанието на регистъра на признаците.

За да разберем какъв формат прехвърля дадена команда MOV, е необходимо да разбираме типа на данните. Например в следния текст:

```
        X1  DB  ?           ; TYPE  X1=BYTE
        Y2  DW  ?           ; TYPE  Y2=WORD
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
        MOV   BH, 0         ; прехвърля байт, защото рег. BH е 8-битов
        MOV   X1, 0         ; същото, защото X1 е от тип байт
        MOV   SI, 0         ; прехвърля дума, защото рег. SI е 16-битов
```

```
MOV   Y2, 0           ; прехвърля дума, защото Y2 е от тип дума
```

Обърнете внимание на записа на втория операнд (0), който не позволява да се определи формàта му.

Формàтите на двата операнда следва да съвпадат. Например:

```
MOV   DI, ES          ; прехвърляне на дума
MOV   CH, X           ; прехвърляне на байт
MOV   DX, AL          ; грешка! Регистрите са с различен формат
MOV   BH, 300         ; грешка! Непосредственият операнд е голям
```

При прехвърляне не е възможно удължаване или скъсяване на форматите на операндите.

3.2.2 Оператор за указване на тип (PTR)

Съществува ситуация, в която по операндите на команда MOV не може да се определи дължината на прехвърляните данни. В аванс ще поясним, че ако даден адрес A трябва да бъде модифициран, например, чрез съдържанието на индексния регистър SI, това се записва така: A[SI]. С други думи, адресът е функция от SI. Известно е, че изпълнителният адрес при този метод на адресиране се определя по формулата $A_{изп}$=A+(SI). В частност, ако адресът е нула (A=0), тогава се записва само [SI], т.е. ако записът е [SI], то $A_{изп}$=0+(SI).

Във връзка с казаното ще разгледаме следната задача: нека в регистър SI се съдържа адресът на определена клетка, в която трябва да се запише 0. Като че ли това може да стане с помощта на командата:

```
MOV   [SI], 0
```

Това обаче не е вярно. Проблемът е в това, че нито по адреса нито по непосредствения операнд 0 не може да се определи форматът на клетката. Тук следва да напомним, че указаният в командата адрес на операнд е само начален и в колко последователни клетки ще се разпростре стойността на операнда може да се определи само по типа му. Така в този частен случай Асемблерът не е в състояние да подбере подходящата машинна команда. Обикновено след това се генерира грешка. За да бъде тя отстранена, програмистът трябва да укаже типа поне на един от операндите. За такива цели в Асемблера е дефиниран операторът за тип PTR (*pointer* – указател), който има структурата:

<тип> PTR <израз>

където <тип> това е BYTE, WORD или DWORD и др. <израз> може да бъде константен или адресен. Така декларираният тип еднозначно определя формàта на стойността на израза. Затова записът:

BYTE PTR 300

е грешен – стойността 300 не се побира в един байт.

Ако в PTR е указан адресен израз, то операторът се разбира като адрес на клетка с указания тип. Например адресен оператор е:

WORD PTR A

където A е адрес на дума, която заема 2 клетки с адреси A и A+1.

Чрез оператора PTR проблемът с неизвестния формат се решава така:

MOV BYTE PTR [SI], 0 или MOV [SI], BYTE PTR 0

Когато се прехвърля 0 във формат дума, то може да се запише:

MOV WORD PTR [SI], 0 или MOV [SI], WORD PTR 0

Следва да поясним, че обикновено е прието да се уточнява типа на адресен операнд, а не на непосредствен операнд. Освен това операторът PTR е полезен в още една ситуация – когато е необходимо не да се уточни типът на операнда, а да се измени. Нека например: променливата Z да има формàт дума:

```
Z DW 1234h          ; Z: 34h,  Z+1: 12h
```

при което е необходимо да се запише нула не в цялата дума, а само в нейния първи байт, с която да се замени съдържанието 34h. Да се направи това с командата:

MOV Z, 0

е погрешно, тъй като така ще се запише 0 и в двата байта и Z ще съдържа "чиста" нула. Това се получава като следствие от директивата DW, с която е описана променливата Z. За да се запише нула в един от байтовете е необходимо да се смени типа на променливата (но само в рамките на действието на командата), което може да стане с оператора PTR така:

MOV BYTE PTR Z, 0

след което в двата последователни байта на променливата ще се съдържа 00h; 12h.

Така чрез оператора PTR само за командата MOV беше отменен типа на преходната директива и беше използван типът на оператора. Следва добре да се разбере, че това изменение на типа е временно и локално.

Аналогична ситуация възниква когато искаме да получим достъп само до втория байт на променливата. Ако искаме това с цел да запишем там друга стойност, например числото 45, то това не бихме постигнали с командата:

MOV Z+1, 45

тъй като типът на Z е дума и въпреки че е указан вторият адрес, типът не може да се промени. Ще припомним, че в Асемблер типът на конструкции от вида:

<име> ± <цяла константа>

притежават типа на употребеното име. По тази причина горната команда ще запише числото 45 два пъти в два последователни байта: (Z+1)=45, (Z+2)=45. Тъй като задачата се отнася само за втория байт на Z, това се постига с оператора PTR както следва:

MOV BYTE PTR (Z+1), 45

Ще допълним още, че по старшинство в Асемблер оператор PTR се изпълнява преди операция събиране, ето защо горния запис се изпълнява в следния ред: първо се подменя типът (BYTE PTR Z), а

след това се изчислява адресът Z+1.

3.2.3 Команда за размяна XCHG

В програмите често се налага да разменят стойностите на две променливи. Това може да се осъществи с командата MOV, но тъй като за това са необходими поне 2 такива команди, то е създадена специална такава – XCHG (*exchange* – размяна). Командата има следната структура:

```
XCHG   op1,  op2      ; op1 := op2
```

Тя разменя местоположението на два операнда, които са от един и същи тип (формàт). Тази команда не променя съдържанието на регистъра на признаците. Пример:

```
 MOV   AX, 62        ; (AX)=62
 MOV   SI, 135       ; (SI)=135
XCHG   AX, SI        ; (AX)=135,   (SI)=62
```

Командата допуска следните комбинации за своите операнди:

op1 ⟵	⟶ *op2*
r8	r8, m8 размяна на байтове
m8	r8
r16	r16, m16 размяна на думи
m16	r16

Както може да се види, размяна на съдържанието на 2 клетки от ОП не е възможно. Ако се налага, то това се прави с помощта на регистър. Например, ако е необходимо да се разменят стойностите на 2 променливи X и Y, то записваме следния текст:

```
 MOV   AL, X        ; прехвърляне на X в AL
XCHG   AL, Y        ; размяна на съдържанието на AL и Y
 MOV   X, AL        ; прехвърляне на AL в X
```

3.3 Команди за събиране и изваждане

Ще припомним, че при обработка на знакови числа с тези операции могат да се получат както верни така и неверни резултати. В последния случай става дума за препълване. Освен резултат се получават и признаци (флагове) на резултата. Командите имат вида:

```
ADD   op1, op2          SUB   op1,  op2
```

Командата допуска следните комбинации за своите операнди:

op1	*op2*
r8	i8, r8, m8
m8	i8, r8,
r16	i16, r16, m16
m16	i16, r16

Действието на командите се дефинира така : *op1 := op1±op2.*

Примери:

```
ADD   AH, 12       ; AH:=(AH)+12
SUB   SI, Z        ; SI:=(SI)-Z
ADD   Z,  -300     ; Z:=(Z)+(-300)
```

Ще напомним още, че операндите и резултатът са представени в допълнителен код. Операндите трябва да имат един и същи формàт (тип). Разновидност на тези команди са командите с подразбиращ се втори операнд (унитарни команди) инкремент и декремент:

```
INC   op1 ,        ;  op1:=op1+1
DEC   op1 ,        ;  op1:=op1-1
```

За тези команди са допустими следните типове на операнда:

r8, m8, r16, m16.

Примери:

```
INC   BL                  ; BL:=(BL)+1
DEC   WORD  PTR  A        ; A:=A-1
```

В този микропроцесор тези две унитарни команди не променят състоянието на флаг C (CF).

Ако желаете да актуализирате флаг C при същата операция и с този операнд, използвайте командите: ADD op1, 1 или SUB op1, 1. Ползата от командите INC и DEC е в това, че те са къси и се изпълняват по-бързо.

Към тази група команди следва да отнесем и командата за смяна на знака на операнда: NEG op. Действието й се изразява така: op:=-op. Допустими типове за операнда са: r8, m8, r16, m16. Например:

```
MOV   AH, 1         ; AH:=1      (1=01h)
NEG   AH            ;AH:=0FFh   (-1=FF) в допълнителен код
```

Има още един особен случай: – смяна на знака на най-малкото за разрядната мрежа число. За 8-битовата дължина това число е –128 (80h). При смяна на знака -(-128)=+128, полученото число за съжаление не може да се представи в тази дължина – настъпва препълване OF=1. Аналогична е ситуацията с 16-битовата дължина.

За събиране и изваждане на много дълги числа са създадени още две команди – събиране с пренос ADC (*Add with Carry*) и изваждане със заем (*Subtract with borrow*):

```
ADC   op1, op2            SBB   op1,  op2
```

За тези команди важат същите типове на операндите.

3.4 Команди за умножение и деление

3.4.1 Команди за умножение

Съществуват две команди за умножение. Първата е за умножение на цели числа без знак (умножение по модул):

```
MUL  op          ; multiply
```

Втората команда е за умножение на числа със знак:

```
IMUL   op        ; integer multiply
```

Когато операндите са 8-битови, множимото винаги трябва да се намира в регистър AL и действието на командата е следното:

AX:=(AL)*op

Произведението винаги се получава в регистър AX и е число с двойна дължина.

При умножение на 16-битови числа произведението е 32-битово и за това се използва фиксираната двойка регистри (DX,AX). Регистър DX е предназначен за старшите цифри на произведението. Действието на командата е следното:

(DX,AX):=(AX)*op

При това фиксирано положение на множимото командата указва само един операнд (множителя), който може да е от тип r8, m8, r16, m16.

Множителят не може да бъде непосредствен операнд. Примери:

```
N   DB   10        ; N=10
MOV   AL, 2        ; (AL)=2
MUL    N           ; AX:=2*10=20=0014h:  (AH)=00h, (AL)=14h
MOV   AL, 26       ; (AL)=26
MUL    N           ; AX:=26*10=260=0104h:   (AH)=01h, (AL)=04h
MOV   AX, 8        ; (AX)=8
MOV   BX, -1       ; (BX)=-1
IMUL   BX      ; (DX,AX)=-8=0FFFFFFF8h: (DX)=0FFFFh, (AX)=0FFF8h
```

Тъй като не винаги произведението фактически прехвърля дължината на младшата си половина, то може да се съхрани във формàта на операндите. За целта следва да се анализират признаците:

- CF=OF=1   ако произведението има старша част ≠0 ;
- CF=OF=0   ако произведението има старша част =0 .

3.4.2 Команди за деление

За изпълнение на операция деление има две команди. Първата е за деление на цели числа без знак (деление по модул):

DIV op   ; *divide*

Втората команда е за деление на числа със знак:

IDIV op   ; *integer divide*

При деление делимото е с формàт дума. Когато делителят е с формàт байт (r8, m8), действието на командата е следното:

AH:=(AX) mod op   (остатък);   AL:=(AX)/op (частно)

Когато делителят е с формат дума (r16, m16), действието на командата е следното:

DX:=(DX,AX) mod op   (остатък);   AX:=(DX,AX)/op (частно)

Както се вижда, и в тези команди местоположението на първия операнд (делимото) е фиксирано. Вторият операнд (делителят) може да се намира в регистър или в клетка от ОП, но не може да бъде непосредствен операнд. Делението е целочислено. Когато остатъкът е нула, то е точно.

Делението е невъзможно, когато:

1. Делителят е 0 (op=0).
2. Частното не се побира в регистъра. Например:

```
MOV   AX, 600      ; делимо=600
MOV   BH, 2        ; делител=2 ;     частно=300
DIV   BH           ; частното 300 препълва рег. AL
```

И в двата случая програмата се канцелира (прекъсване “Int 0”).

3.5 Промяна на формàта на число

Разглеждаме следната задача: BX:=(BX)+(AL) . Задачата е да се съберат две числа с различен формàт – дума с байт. Това ще стане възможно, ако удължим по-късия формàт до по-дългия. Възможни са два случая:

1. Ако числото е без знаково. Нека (AL)=5Fh.

В този случай отляво на числото се добавят незначещи нули. Това се постига с команда:

```
MOV   AH, 0.
```

Така съдържанието на AX става (AX)=005Fh. Тогава събирането се осъществява с командата:

```
ADD   BX, AX         ; BX:=(BX)+(AL)
```

2. Ако числото е със знак. Нека числото е отрицателно –5Fh.

Тъй като тогава числото ще бъде в представено в допълнителен код:

$$[-5Fh]_{дк}=00A1h,$$

в регистър AL ще бъде заредено числото (AL)=0A1h. Знаковото разширение на това отрицателно число ще бъде с незначещи единици FF, което може да се постигне с командата:

```
MOV   AH, 0FFh
```

Така съдържанието на регистър AX става (AX)=0FFA1h. Необходимата сума може да се получи аналогично:

```
ADD   BX, AX         ; BX:=(BX)+(AL)
```

За да се улесни знаковото разширение на целите числа от по-късия към по-дългия формат е създадена специална команда:

CBW (*Convert Byte to Word*)

Местоположението на операнда на тази команда е фиксирано и той задължително трябва да се намира в регистър AL. Резултатът винаги се получава в регистър AX. Действието на командата може да изразим така:

$$(AH)=\begin{cases}00h, & \text{ako } (AL)\geq 0;\\ 0FFh, & \text{ako } (AL)<0.\end{cases}$$

Тази команда не променя флаговете.

Примери:

```
MOV   AL, 32         ; (AL)=32=20h
CBW                  ; (AX)=0020h
```

```
MOV   AL, -32          ; (AL)=-32=0E0h
CBW                    ; (AX)=0FFE0h
```

Сега да припомним задачата: BX:=(BX)+(AL). В зависимост как програмистът интерпретира съдържанието на разрядната мрежа, са възможни два варианта: за знакови и за без знакови числа:

Числа без знак	*Числа със знак*
MOV AH, 0 ADD BX, AX	CBW ADD BX, AX

Необходимостта от удължаване на формàта възниква преди всичко при операция деление, за която делимото следва да е от тип дума. Аналогично се налага удължаване на формàта от дума до двойна дума. За да не се прави това последователно байт след байт, е създадена командата CWD (*Convert Word to Double word*), чието действие се представя както следва:

$$(DX)=\begin{cases} 0000h, & \text{ako } (AX)\geq 0\ ; \\ 0FFFFh, & \text{ako } (AX)<0\ . \end{cases}$$

3.6 Примерни задачи

Задача № 1

Дефинирана е променливата X: X   DD   ? Да се разменят двете половини на X.

За да се направи размяната трябва да се посочи началния адрес на всяка половина, което може да стане само с оператор PTR:

<u>*Решение*</u>:

```
MOV   AX, WORD  PTR  X         ; старшата половина се записва в AX
XCHG  AX, WORD  PTR  X+2       ; размяна
MOV   WORD  PTR  X,  AX        ; младшата половина се записва в ОП
```

Ако се използва директивата за еквивалентност EQU горният текст може да се опрости:

```
WP   EQU  WORD  PTR
MOV   AX, WP X
XCHG  AX, WP X+2
MOV   WP X, AX
```

Задача № 2

Дефинирана е променливата W: W   DW   1234h. Да се определи съдържанието на регистрите AX и BX след изпълнение на следния код:

```
MOV   AX, W
MOV   BH, BYTE  PTR  W
MOV   BL, BYTE  PTR  W+1
```

<u>*Решение*</u>:

Съдържанието на регистрите е: (AX)=1234h, (BX)=3412h. Разликата

се дължи на факта, че в паметта байтовете на регистрите се подреждат в "обратен" ред.

Задача № 3

Нека X е знакова еднобайтова променлива, а Y е променлива от тип дума. Да се изчисли стойността й по формулата: Y=X*X*X. Приема се, че резултатът не надхвърля формàта дума.

Решение:

По време на аритметически изчисления следва внимателно да се контролират типа на операндите и правилата за удължаване. Освен това трябва да се следи за правилната разстановка на операндите в нужните регистри, както изискват отделните команди.

Относно задачата следва веднага да отбележим, че в следствие на двете последователни умножения, резултатът ще се удължава последователно два пъти. Тъй като стойността на X е число със знак, умножението ще трябва да се извърши от команда IMUL, която изисква множимото да се намира в регистър AL. Ето защо първото, което следва да се направи, е да се запише X в AL. Произведението X*X ще се получи в регистър AX. Тъй като не е възможно да се умножи дума с байт, X следва да се удължи до дума. Крайният резултат е с дължина двойна дума, ето защо ще се получи в двойката регистри (DX,AX). Така се стига до окончателния текст на програмата:

```
MOV   AL, X
IMUL  AL            ; AX:= (AL)*(AL)
MOV   BX, AX
MOV   AL, X
CBW                 ; знаково удължаване на X до дума
IMUL  BX            ; (DX,AX):=(BX)*(AX)=(X*X)*X
MOV   Y, AX         ; запис само на младшата част
```

Задача № 4

Дефинирани са променливите:

```
N   DB  ?               ; еднобайтова променлива N
D   DB  3 DUP(?)        ; масив D от 3 еднобайтови елемента
```

Променливата N да се приема за число без знак, имащо 3-цифрен десетичен вид $d_2d_1d_0$ [000, 255]. Да се създаде масивът:

$$D(1)= d_2, \quad D(2)= d_1, \quad D(3)= d_0$$

Решение:

Алгоритъмът на тази задача следва да даде десетичните цифри като остатъци на съответното деление във вид на символи. Така например d_2 е остатък от деление на числото N с числото 100, d_1 е остатък от деление на този остатък с числото 10, а остатъкът от това деление е d_0.

```
N   DB  ?
D   DB  3 DUP(?)
```

```
MOV   BL,10
MOV   AL, N
MOV   AH, 0
DIV    BL              ; AX:=(AX)/(BL),   (AH)=d0 ,  (AL)=d2d1
ADD   AH, “0”         ; получава се ASCII кода на младшата цифра
MOV   D+2, AH        ; запис на елемент D(3):=(AH)=d0
MOV   AH, 0           ; (AX)=d2d1
DIV    BL              ; (AH)=d1 ,  (AL)=d2
ADD   AX, “00”        ; получава се ASCII кода на двете цифри
MOV   D+1, AH        ; запис на елемент D(2):=(AH)=d1
MOV   D, AL           ; запис на елемент D(1):=(AH)=d2
```

Задача № 5

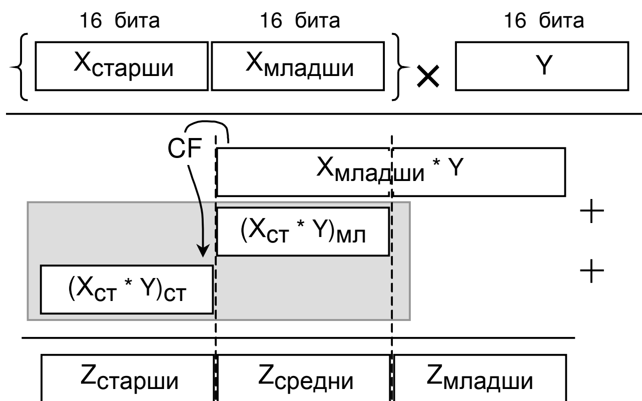
Дефинирани са променливите:

```
X   DD  ?                 ; променлива X от тип двойна дума
Y   DW  ?                 ; променлива Y от тип дума
Z   DW   ?, ?, ?          ; масив Z от 3 елемента от тип дума
```

Да се изчисли Z=X*Y ако се предполага, че X и Y са числа без знак и че тяхното произведение се записва като тройна дума Z в “обратен” ред, т.е. най-напред се записва младшата част (в адрес Z), после средната (в адрес Z+2) и последна най-старшата част (в адрес Z+4).

<u>*Решение*</u>:

Тъй като не разполагаме с команда за умножение на двойна дума с дума, умножението ще реализираме по следната схема:



Ще отбележим, че пред адресите Z, Z+2 и Z+4 не трябва да се поставя указателя WORD PTR, тъй като името Z, съгласно своята директива е описано вече от тип дума. Пред адресите на променливата X обаче този указател трябва да се поставя.

Така се получава следният текст:

```
WP   EQU   WORD  PTR
MOV   AX, WP X               ; AX := X младша част
MUL    Y                     ; (DX,AX) := (Xмл)*Y
MOV   Z, AX                  ; запис на Z младша част
MOV   BX, DX                 ; съхранение на старшата част на (Xмл)*Y
MOV   AX, WP X+2             ; AX := X старша част
MUL    Y                     ; (DX,AX) := (Xст)*Y
ADD    AX, BX                ; AX := [(Xст)*Y]мл + [(Xмл)*Y]ст ,  CF:=p
MOV   Z+2, AX                ; запис на средната част на Z
ADC    DX, 0                 ; DX := (Xст)*Y  +  (CF)
MOV   Z+4, DX
```

Г Л А В А 4

КОМАНДИ ЗА УПРАВЛЕНИЕ НА ПРЕХОДА. ЦИКЛИ

Тук ще бъдат разгледани команди за реализация на алгоритмични преходи, включително и такива за организация на цикли, както и операции за въвеждане и извеждане на данни. Пояснените тук команди ще бъдат използвани в следващите глави.

Машинните команди се изпълняват в онази последователност, в която са били записани. Известно е обаче, че този естествен ред се налага да бъде нарушаван. Това е възможно само с помощта на специално създадените команди от групата за управление на прехода, най-общо делящи се на команди за безусловен и за условен преход.

Командите за преход имат едноадресна структура. Командите за преход не променят съдържанието на регистъра на флаговете. Последните остават непроменени след изпълнение на команди за преход.

Точките, които получават управление чрез командите за преход в Асемблер се маркират (именуват) с помощта на етикети (вижте определението в пункт 2.2.1).

4.1 Безусловен преход. Оператор SHORT.

Съществуват няколко команди за безусловен преход, но в Асемблер те се означават по един и същи начин:

JMP <операнд> ; *jump* – преход, скок.

Операндът на командата за безусловен преход е символичен – той представлява етикет, но по един или друг начин указва (представлява) адреса за преход. По този адрес се намира следващата по действие команда. В микропрограмен смисъл командата за преход зарежда програмния брояч с ново съдържание (нов адрес).

Безусловен преход със скок

Командата се записва така: JMP <етикет>. Например:

```
        JMP   Label
        ....  ....
Label:  MOV   AX, 0
```

Логиката подсказва, че един и същи етикет не може да се употребява за означаване на различни точки в текста на програмата. Все пак е полезно да се поясни, че Асемблерът поема задачата за определяне на съответстващия на етикета адрес. На практика, за да се осигури преместваемостта на програмите, етикетът има същността на отместване или още изразява дължината на прехода в брой адресируеми единици. При това дължината на прехода е знаково число. Дължината на прехода може да се интерпретира и като адрес, но относителен спрямо нулева база. На машинно ниво адресът за преход

се изчислява като сума от съдържанието на програмния брояч и отместването, съдържащо се в командата. Тази сума се записва като ново съдържание на програмния брояч.

Тъй като отместването може да бъде 8-битово и 16-битово (вижте фигура 1.7.1), то командите за безусловен преход със скок са две. Разликата между тях е в дължината на прехода: в интервала $[-2^7, 2^7-1]$ или в интервала $[-2^{15}, 2^{15}-1]$. В езика Асемблер обаче този факт не се забелязва (остава скрит), тъй като означението на цялата команда е символно.

Оператор SHORT

Все пак поясненията, дадени по-горе, имат основание да се разбират, защото те са във връзка с действията, които изпълнява Асемблерът. Срещайки символна команда за преход с етикет, Асемблер изчислява дължината на прехода, след което го оценява. Ако разликата е число в интервала $[-2^7, 2^7-1]$, то той формира 2 байтова машинна команда. А когато разликата е число в интервала $[-2^{15}, 2^{15}-1]$, той формира 3 байтова машинна команда.

Изчислението на дължината на прехода като разлика между целевия и текущия адрес не винаги е възможно. То е възможно само в случаите, когато указаният етикет е вече срещнат и описан, т.е. той изразява преход назад в текста на програмата. Ако обаче указаният в командата етикет още не е срещан и не е описан, то изчислението е невъзможно, тъй като става дума за преход напред, към все още не разглеждани команди в текста на програмата. В такива случаи Асемблерът “залага” на сигурен вариант, като генерира 3 байтова машинна команда, т.е. команда за дълъг преход, въпреки че последният може да се окаже къс. Това естествено е неефективно решение. В много случаи, при програмиране е предварително известно, че дължината на прехода не е голяма и че една 2 байтова команда за преход е напълно подходяща. За такива случаи в помощ на програмиста е предназначен оператор SHORT (къс). С този оператор програмистът може да укаже на Асемблер какъв избор на команда да направи. Например:

```
        JMP   L12             ; дълъг преход (3 байтова команда)
        JMP   SHORT   Lab     ; къс преход (2 байтова команда)
         ....  ....
Lab:    xxxxxxx               ; команда с етикет  Lab
         ....  ....
L12:    zzzzzzz               ; команда с етикет  L12
```

Следва да се отбележи, че ако в програмата е употребен оператор SHORT, но в същност това е грешка, т.е. преходът на практика е дълъг, то при компилиране тази грешка ще бъде изявена и съобщена на програмиста. Обикновено този оператор се използва когато системата изпитва недостиг на оперативна памет.

Оператор SHORT може да бъде поставян и пред етикет, който вече е описан, т.е. при преход назад. В този случай обаче Асемблерът ще игнорира оператора, защото при преход назад той вече “знае” дължината му, тъй като я е изчислил.

4.2 Косвен преход

Косвено адресираният преход има своето приложение и своите преимущества. Теоретично този метод за адресиране е изяснен подробно в [1]. Същността на прехода се състои в това, че командата съдържа адрес на “пощенската кутия”, в която се намира адресът за преход. За такава пощенска кутия може да се използва както регистър с общо предназначение, така и клетка в ОП. Например:

```
JMP   r16  ;              JMP   m16
```

При изпълнение на командата се изпълняват две обръщения. В първия случай към регистър и после към ОП, а във втория – към ОП. И в двата случая извлеченият адрес за преход се зарежда в програмния брояч като изпълнителен, а не като отместване.

В Асемблер, за разлика от езика на микрооперациите, се приема че съдържанието се изразява с ограждане в правоъгълни скоби. Така означението [r2] следва да се чете “*съдържанието на регистър* r2”.

Пример:

```
    A   DW   L              ; описва A като дума, чиято стойност е L
     ....  ....
    JMP   A                 ; goto [A]  ≡  goto L
     ....  ....
    MOV  DX, A              ; DX:=(A),  т.е. (DX)=L
    JMP   DX                ; goto [DX]  ≡  goto L
     ....  ....
L:  xxxxxxxx
```

Косвените преходи се използват, когато адресът за преход става известен по време на изпълнение на програмата. Такива алгоритмични ситуации ще бъдат пояснени по-късно тук.

Асемблерът изпитва известни затруднения при компилиране на безусловни преходи. Нека разгледаме команда JMP Z, в която името Z не е име на регистър. При транслиране на тази команда възниква въпросът: това преход по етикет Z ли е или е косвен преход по адрес, който се съдържа в клетка Z?

Ако името Z е описано до тази команда, то проблем няма – ако с името Z е отбелязана команда, т.е. Z е етикет, тогава е на лице преход по етикет (вижте пример а) в следващата таблица). Ако обаче името Z е описано в директива DW (вижте пример б)), тогава е на лице косвен преход.

а)	б)	в)
Z: INC AX .... JMP Z ; goto Z	Z DW L .... JMP Z ; goto L	JMP Z ; goto Z .... Z: xxxxxxxx

Когато обаче името Z е етикет, който е описан напред в текста на програмата (вижте пример в)), тогава Асемблерът не може да знае какъв е прехода. Това определено е проблем, който е решен в компилатора, който в такива случаи винаги интерпретира името Z като етикет и винаги формира 3 байтова команда за преход (дълъг преход). Ако в последствие Асемблер установи, че името Z не е описано като етикет, тогава генерира съобщение за грешка.

В случай, че този избор не е подходящият за програмиста, то Асемблер трябва да бъде предупреден за това. За целта следва да се използва оператор PTR: вместо името Z програмистът трябва да запише конструкцията (WORD PTR Z), с която се съобщава на Асемблера, че той следва да разглежда името Z като име на променлива с дължина 2 байта, за да може той да формира 3 байтова машинна с косвена адресация. Така при преходи напред са възможни следните конструкции:

JMP Z ; goto Z .... Z: xxxxxx	JMP Z ; *грешка* .... Z DW L	JMP WORD PRT Z ;goto L .... Z DW L

4.3 Команди за сравнения и за условен преход

Структурата и действието както командите за условен преход, така и на командите за сравнение, са изяснени подробно в [1]. Всички условни преходи се осъществяват въз основа на стойностите на признаците на резултата (на флаговете), които се фиксират при всяка операция в регистъра на флаговете. Всеки път, когато е необходимо тези стойности да се актуализират, може да се изпълни команда за сравнение. За тази команда трябва да се помни, че тя е единствената, която не фиксира резултат, а само признаците му. Командата за сравнение има вида:

CMP op1, op2 ; *compare* – сравнение.

Действието на командата е изваждане: (op1-op2).

Командите за условен преход в системата машинни команди на процесора са достатъчно много, но в Асемблер има само една команда, която се записва както следва:

Jxx <етикет>

където с буквите “xx” се изразява проверяваното условие.

Всички команди за условен преход могат да се групират в 3 групи.

1. <u>В първата група</u> са командите, които се употребяват след команда за сравнение, която формира стойността на условието за преход. Условието, което се изразява мнемонично, може да е:

   E – *equal* (равно) ;
   N – *not* (не, отрицание) ;
   G – *greater* (по-голямо) – за числа със знак ;
   L – *less* (по-малко) – за числа със знак ;
   A – above (по-нагоре, по-голямо) – за числа без знак ;
   B – below (по-надолу, по-малко) – за числа без знак

Както се вижда, за условията "по-голямо" и "по-малко" са въведени по две системни означения. Това е свързано с това, че след сравнение на числа със знак и сравнение на числа без знак, за тези условия следва да се анализират различни признаци.

Една и съща команда за условен преход може да има в Асемблер няколко наименования (синоними). Това се обяснява с факта, че за една и съща ситуация могат да бъдат зададени различни въпроси. Например може да се попита така:

По-малко ли е ?   За по-малко отговорът е "да".

Но за същото може да се попита и така:

По-голямо или равно ли е?   За по-малко отговорът е "не".

Ето защо условният преход "по-малко" за числа със знак може да се означи с две команди, чиято мнемоника се конструира така: JL или JNGE. Коя мнемоника от двата синонима ще използва програмистът за Асемблер е без значение. Всички команди от тази група са представени в следващите таблици.

За числа без знак:

Мнемоника	*Проверявано условие*	*Признак*
JB / JNAE	Op1 < Op2	CF=1
JBE / JNA	Op1 <= Op2	CF=1 или ZF=0
JA / JNBE	Op1 > Op2	CF=0 или ZF=0
JAE / JNB	Op1 >= Op2	CF=0

За числа със знак:

Мнемоника	*Проверявано условие*	*Признак*
JL / JNGE	Op1 < Op2	SF<>OF
JLE / JNG	Op1 <= Op2	SF<>OF или ZF=1
JG / JNLE	Op1 > Op2	SF<>OF или ZF=0
JGE / JNL	Op1 >= Op2	SF=OF

където означението <> се чете "*са различни*".

Условието за преход “по-малко”, проверявано от команда JL, се свежда до проверката на това, дали флаговете SF и OF имат различни стойности, по следните причини: При команда “Сравнение” двата операнда са със знак, а те могат да са еднакви или различни. При еднакви знаци (например Op1=+5, Op2=+3, или например Op1=-5, Op2=-3), фактически се изпълнява изваждане, при което препълване не е възможно, т.е. OF=0. Но разликата може да бъде както положителна (+2) (SF=0), така и отрицателна (-2) (SF=1). В този случай е ясно, че отношението Op1<Op2 е вярно само при комбинация OF=0 и SF=1. За примерните числа се получава:

Отношението +5 < +3 (разлика +2) не е вярно (OF=0 и SF=0) ;
Отношението -5 < -3 (разлика –2) е вярно (OF=0 и SF=1) ;
Отношението +3 < +5 (разлика –2) е вярно (OF=0 и SF=1) ;
Отношението -3 < -5 (разлика +2) не е вярно (OF=0 и SF=0) ;

Когато обаче знаците на числата са различни, операция изваждане (Op1-Op2) води до фактическо събиране на числа с еднакви знаци, което пък от своя страна може да доведе до препълване. Ако препълване няма, т.е. OF=0, това означава, че знакът на разликата съвпада със знака на Op1. Тогава ако знакът е положителен (SF=0), това означава, че отношението Op1<Op2 не е вярно. Ако обаче знакът е отрицателен (SF=1), това означава, че отношението Op1<Op2 е вярно. В този случай е налице комбинацията OF=0 и SF=1.

Отношението +5 < -3 (разлика +8) не е вярно (OF=0 и SF=0) ;
Отношението -5 < +3 (разлика –8) е вярно (OF=0 и SF=1) ;
Отношението +3 < -5 (разлика +8) не е вярно (OF=0 и SF=0) ;
Отношението -3 < +5 (разлика -8) е вярно (OF=0 и SF=1) ;

Алтернативната ситуация се характеризира с препълване, т.е. OF=1. Това означава, че полученият знак на резултата не е верен. Ако той е отрицателен (SF=1), това означава, че истинската разлика е положително число, което е възможно само когато Op1>0 и Op2<0. В такъв случай обаче проверяваното отношение Op1<Op2 не е вярно. В другия случай, когато знакът на резултата е положителен (SF=0), истинският знак е всъщност отрицателен, което е възможно само когато Op1<0 и Op2>0. При такива операнди обаче проверяваното отношение е вярно. Ето примери с едноразрядно десетично препълване:

Отношението +7 < -9 не е вярно (OF=1 и SF=0) ;
Отношението -7 < +9 е вярно (OF=1 и SF=1) ;
Отношението +9 < -7 не е вярно (OF=1 и SF=0) ;
Отношението -9 < +7 е вярно (OF=1 и SF=1) ;

Следва да обобщим, че при всички възможни ситуации, които бяха описани, беше констатирано, че когато проверяваното отношение е вярно, е на лице различие в стойностите на признаците OF и SF, т.е. е на лице комбинацията OF=0 и SF=1 или комбинацията OF=1 и SF=0. Именно това различие е изразено с означението “<>” в горната таблица.

За произволни числа:

Мнемоника	*Проверявано условие*	*Признак*
JE	Op1 = Op2	ZF=1
JNE	Op1 <> Op2	ZF=0

Пример: Дадени са променливите X, Y и Z с формат дума. Нека в Z се запише по-голямото от числата X и Y.

Решение: Решението на задачата е различно за числа със знак (текстът отляво) и за числа без знак (текстът отдясно):

; *числа със знак*	; *числа без знак*

MOV AX, X
CMP AX, Y ; x=y?
JGE M
MOV AX, Y
M: MOV Z, AX
	MOV AX, X
	CMP AX, Y
	JAE M
	MOV AX, Y
	M: MOV Z, AX

2. <u>Във втората група</u> команди за условен преход влизат онези, които се поставят след команди, различни от команди за сравнение, и които използват една или друга стойност на определен признак. Мнемоничният код на тези команди се изгражда от буквата J, след която се записва буквата, име на проверявания флаг или нейното отрицание, в зависимост от това, по коя стойност е прехода.

Мнемокод	*Условие*	*Мнемокод*	*Условие*
JZ	ZF=1	JNZ	ZF=0
JS	SF=1	JNS	SF=0
JC	CF=1	JNC	CF=0
JO	OF=1	JNO	OF=0
JP	PF=1	JNP	PF=0

Забележете, че следните двойки мнемокодове са еквивалентни:

(JE ≡ JZ), (JNE ≡ JNZ), (JB ≡ JC), (JNB ≡ JNC).

Пример: Нека A, B и C са беззнакови еднобайтови променливи. Да се изчисли стойността на израза C=A*A+B, но ако стойността препълва дължината байт, управлението да се предаде в точка с етикет ERROR.

Възможно е следното решение:

```
MOV   AL, A
MUL   AL
JC    ERROR      ; A*A > 255   (CF)=1  →  ERROR
ADD   AL, B
JC    ERROR      ; пренос (CF)=1  →  ERROR
MOV   C, AL
```

3. В третата последна група влиза само една команда за условен преход, проверяваща не флагове, а съдържанието на регистъра брояч CX, т.е. условието (CX)=0?. Командата има вида:

JCXZ <етикет>

Действието на командата е следното:

if CX=0 *then goto* <етикет>

Всички команди за условен преход обаче си приличат по едно – те реализират само къс преход, тъй като адресната им част може да съдържа само 8 битово отместване. Разбира се в програмите е напълно възможно да се появи необходимост от условен преход с дължина по-голяма от 128. Единственото решение на този проблем е скокът да се реализира като безусловен. Например, за логиката на прехода:

if AX=BX *then goto* M

където етикетът M е в далечна точка, може да се предложи логиката на следните преходи:

```
     if AX<>BX then goto L  ; къс преход
     goto M                 ; дълъг преход
L:   xxxxxxxx
```

която в Асемблер може да се реализира както следва:

```
    CMP  AX, BX
    JNE  L          ; условен преход на L при не равно
    JMP  M          ; безусловен преход (скок) на M
L:  xxxxxxxx
```

В командите за условен преход използването на оператор SHORT е безсмислено и безполезно.

4.4 Команди за реализация на цикли

С помощта на командите за управление на прехода могат да бъдат реализирани всякакви алгоритмични структури – разклонени и циклически. Циклите като структури се делят на два вида – с отнапред известен брой повторения и с отнапред неизвестен брой повторения.

Команда LOOP

Организацията на циклите от първия вид използва брояч. За тази цел е предназначен регистър CX. Така условието за край на цикъла е въпросът: "*Е ли съдържанието на брояча станало равно на нула?*" Действията, свързани с организацията на циклическите повторения, не са свързани с действията в тялото на цикъла и тези действия са:

- Зареждане на брояча с начална стойност CT:=n (изпълнява се само веднъж преди входната точка на цикъла) ;
- Модифициране на съдържанието на брояча, обикновено декрементно CT:=(CT)-1 (изпълнява се многократно като действие от тялото на цикъла) ;

- Проверка на условието за край на цикъла (CT)=0? (изпълнява се многократно като последно действие от тялото на цикъла).

Изказаната организация на повтарящи се действия Асемблер реализира например така:

```
       MOV   CX, N        ; зареждане на брояча с начална стойност
    L: xxxxxxxxxxx
        ....  ....  ....       ; команди от тялото на цикъла
       xxxxxxxxxxx
       DEC   CX           ; CX:=(CX)-1 формира се нова стойност на ZF
       JNE    L           ; if  ZF=0  then goto L,  else  next line
```

С цел минимизиране на текста и повишение на скоростта на изпълнение на описваната конструкция в програмата, в системата команди е въведена специалната команда LOOP. Тази команда обединява действията на последните две команди в горния текст – модифицира съдържанието на брояча, след което проверява неговото съдържание: CX:=(CX)-1; *if* ZF *then* goto <етикет>. Така текстът на оптимизирания пример ще бъде следния:

```
       MOV   CX, N       ; зареждане на брояча с начална стойност
    L: xxxxxxxxxxx
        ....  ....  ....      ; команди от тялото на цикъла
       xxxxxxxxxxx
       LOOP   L          ; if  ZF=0  then goto L,  else  next line
```

Реализираните по-горе цикли могат да се допълнително определят като цикли с пост условие. В такива конструкции действията в тялото на цикъла се изпълняват задължително поне един път. Цикли с предусловие, когато е възможно действията от тялото на цикъла да бъдат прескочени, не могат да се реализират с команда LOOP. За такива случаи е предназначена командата JCXZ. Например:

```
        MOV   CX, N        ; N ≥ 0
        JCXZ   L1          ; if  (CX)=0  then  goto L1
    L:  xxxxxxxxxxx
         ....  ....  ....       ; команди от тялото на цикъла
        xxxxxxxxxxx
        LOOP   L           ; if  ZF=0  then goto L,  else  next line
    L1: next line
```

Следва да се знае, че команда LOOP реализира къс преход, тъй като отместването което тя съдържа е 8 битово.

<u>Пример</u>: Да се изчисли стойността на функцията n! за n≤8. При това ограничение максимално възможната стойност на функцията е 40320, която е във възможностите на 16 битовата разрядна мрежа.

Решение: Стойността на функцията ще бъде изчислена като натрупващо се произведение в регистър AX, AX:=(AX)*i, i:=i+1.

```
        MOV   AX, 1        ; регистър AX приема начална стойност
        MOV   CL, N        ; регистър CL приема начална стойност
        MOV   CH, 0
        JCXZ  M2
        MOV   SI, 1        ; регистър SI приема начална стойност
M1:     MUL   SI           ; (DX, AX) := (AX)*i
        INC   SI           ; SI := (SI)+1   нов множител
        LOOP  M1
M2:     xxxxxxxxxx
```

Команда LOOPE / LOOPZ и LOOPNE / LOOPNZ

Както и команда LOOP, и тези команди организират повторения, толкова на брой, колкото е началното съдържание на регистър CX. В допълнение обаче те допускат преждевременен изход от цикъла. Командите LOOPE и LOOPZ са синоними. Ще припомним, че командата LOOP не променя стойностите на флаговете. Действието на тази команда може да се опише така:

CX := (CX)-1 ;   *if* ((CX)<>0) *and* (ZF=1) *then* goto <етикет>

Командата съвместява действията: модифициране на брояча, след което проверка на условието, което е логическа конюнкция между признак за нулев резултат от предходна команда и все още не нулиран брояч. Този преход се изпълнява когато цикълът не е завършил, при условие, че в тялото на цикъла, при поредната операция, е получен нулев резултат. Преходът, който тази команда реализира, е къс.

За да се реализира логиката на командата LOOPE тя следва да се предхожда непосредствено от команда, която формира актуалната стойност на признак ZF. Обикновено се използва командата за сравнение CMP.

Какъв е изходът от цикъла при такова програмиране (нормален или преждевременен) е възможно да се провери едва след излизането. Необходимо е да се провери флагът ZF (с команда JE/JZ или JNE/JNZ), а не съдържанието на регистър CX, тъй като условието ZF=0 (не равно) може да се появи тъкмо в последното завъртане в цикъла, когато броячът се е нулирал.

Най-често команда LOOPE се използва в алгоритми, търсещи първия елемент на масив, различен от някакво зададено число. Ще илюстрираме възможностите на командата чрез следния пример: да се запише в регистър BL най-малкото число в интервала [2, K], на което не се дели даденото число N (K и N са еднобайтови променливи, 2≠K<N), или да се запише 0, ако такова число няма. За целта числото N ще бъде последователно делено с числата 2, 3, 4, ...,K, при което остатъците ще бъдат сравнявани с нула. Тази последователност от действия ще бъде прекратена ако не намери точно деление (остатък 0) или ако бъде изчерпан зададеният интервал.

```
          MOV   DL, N
          MOV   DH, 0        ; DX := N   формат дума
          MOV   CL, K
          MOV   CH, 0
          DEC   CX           ; CX := K-1     брояч на цикъла
          MOV   BL, 1
DV:       INC   BL           ; поредно число от интервала
          MOV   AX, DX
          DIV   BL           ; AH := N(mod[BL])
          CMP   AH, 0        ; сравнение с нула  mod=0?
          LOOPE DV           ; проверка на условието за край на цикъла
          JNE   DV1          ; преход в точка DV1 ако ≠0, в противен
                             случай преход на следващата поред команда
          MOV   BL, 0
DV1:      ....  ....  ....
```

Следващите команди, с които могат да се организират цикли с известен брой повторения и с допълнително условие, са командите LOOPNE и LOOPNZ:

LOOPNE <етикет>
LOOPNZ <етикет>

Тези команди изпълняват действията:

CX := (CX)-1 ;   *if* ((CX)<>0) *and* (ZF=0) *then* goto <етикет>

Командата LOOPNE обикновено се използва в алгоритми, които търсят в масив първия елемент, който ще се окаже равен на зададена стойност.

4.5 Операции Въвеждане / Извеждане

Операциите за въвеждане и извеждане на данни са повече от необходими за всяка програма. В системата машинни команди командите за тези операции осъществяват трансфер само на един байт. За по-съдържателни програми обаче тази възможност е крайно недостатъчна – необходими са значително по-мощни възможности, което характерно за така наречените *макрокоманди*. Ще бъдат изяснени по-късно тук. Сега ще отбележим само, че когато операндът е символен низ, в който се среща празен символ, низът трябва да бъде заграден с ъглови скоби <>. Например:

OUTINT <WORD PTR X>, 7

4.5.1 Спиране на програмата

Последното действие във всяка програма естествено е действие по прекратяване на нейното изпълнение, т.е. действие стоп. За тази цел в Асемблер се употребява командата:

FINISH.

Тази команда задължително трябва да бъде достигната.

4.5.2 Въвеждане от клавиатурата

Всички въведени от клавиатурата символи попадат в клавиатурния буфер след натискане на клавиша ENTER. От този буфер техните кодови комбинации се изчитат с помощта на командите за въвеждане. Например, ако в програмата е реализирано въвеждане само на един символ, въпреки което от клавиатурата могат да бъдат набрани много символи. В същото време от клавиатурния буфер ще бъде прочетен само един символ, тъй като така е запрограмирано. При това, докато потребителят не натисне клавиш ENTER, се допуска редактиране на набраната последователност от символи. Така например чрез натискане на клавиш *Esc* може да бъде унищожен въведеният текст, а чрез клавиш *Backspace* се унищожава последният въведен символ.

Тъй като самите команди за въвеждане не подканват потребителя, то в програмата тази команда следва да се предхожда от подходящо подканящо съобщение, например извеждане на екрана на символ >.

Първата команда за въвеждане в Асемблер се записва така:

```
INCH   op      ; допустими типове за операнда са:  r8 и m8
```

Тази команда въвежда поредния символ, при което неговия код се записва в указания еднобайтов регистър или клетка от ОП.

Втората команда се записва така:

```
ININT   op      ; допустими типове за операнда са:  r16 и m16
```

Командата въвежда число, което следва да бъде записано в десетична бройна система. Преди да бъде записано в приемния регистър обаче Асемблерът предвижда действия по неговото преобразуване в двоична бройна система. Ако записът на числото се предхожда от празни символи, те се “поглъщат” от командата. Ако числото е набрано без знак или със знак +, то се въвежда като число без знак, като е ограничено от максимално възможната стойност, която е 2^{16}-1. Ако числото се предхожда от знак минус (-), то се въвежда като отрицателно, представено в допълнителен код. Възможните отрицателни числа са в интервала от -2^{-15} до –1.

Когато се четат символите на въвежданото от клавиатурния буфер число, то за край на цифровата му част се приема всеки символ, който не е цифра.

Когато числото е въведено неправилно или е непредставимо в указания формат, се сигнализира за грешка и програмата се канцелира. Тъй като тази команда работи с формат дума, то командата (ININT AH) например е грешна, а командата (ININT AX) е правилна.

Ако програмистът иска да изчисти клавиатурния буфер преди да започне нови операции въвеждане, той може да употреби команда

```
FLUSH      ; изчистване на клавиатурния буфер
```

4.5.3 Извеждане на екрана

Извеждането на екрана на <u>символи</u> се осъществява незабавно и без

буфериране. Поредният извеждан символ се помества в текущата позиция на курсора, след което той се премества в дясно. За извеждане на екрана на символ се използва командата:

```
OUTCH   op          ; извеждане на символ
```

Допустимите типове за операнда на командата са: r8, i8, m8.

Източникът на операнда е самата команда, еднобайтов регистър или една клетка от ОП.

За принудително преминаване на нов ред програмата следва да съдържа командата:

```
NEWLINE             ; преход на нов ред
```

Когато програмистът иска да изведе на екрана не само един символ, а цяла последователност, т.е. символен низ, може да използва следната команда:

```
OUTSTR              ; извеждане на ред
```

Командата няма операнд, тъй като неговото място е фиксирано – регистър DX, съдържащ началния адрес на символната последователност в ОП. Процесът на последователно извеждане на символи се прекратява когато поредният символ се окаже “$”. С други думи това е символ за означаване на края на символния низ. Самият той не се извежда (не се визуализира).

Пример:

```
  S   DB  “Hello”, “$”
AS  DW  S               ; S е начален адрес на символния низ
  ....  ....  ....
MOV   DX, AS            ; зареждане на началния адрес в рег. DX
OUTSTR                  ; извежда на екрана текста  Hello
```

Забележка: По-късно ще бъде пояснен по-лесен начин за задаване на началния адрес.

Когато на екрана е необходимо да се изведе число, могат да се употребят следните команди:

```
OUTIN    op1  [,op2]         ; извеждане на число със знак
OUTWORD   op1  [,op2]        ; извеждане на число без знак
```

Типовете, допустими за командите, са:

- За операнд 1 – i16, r16, m16 ;
- За операнд 2 – i8, r8, m8 .

Тези две команди работят аналогично – извеждат в десетична бройна система двубайтовото число, зададено като първи операнд. Например:

```
OUTIN    0FFFFh          ; извежда числото -1
OUTWORD    0FFFFh        ; извежда числото  65535
```

Вторият операнд на тези команди, когато го има, винаги се интерпретира като число без знак, което задава броя на позициите,

отделени за поместване на цифрите на извежданото число. Отпечатъкът на числото се визуализира дясно подравнен в отделеното му поле, а оставащите в повече позиции, стоящи в ляво на отпечатъка, стоят празни символи. Ако дължината на полето е недостатъчна (или въобще не е указана), тогава отпечатъкът на числото заема само необходимите му позиции. Например, следващите команди:

```
OUTCH    "*"
OUTIN   12
OUTWORD   345, 6
OUTIN   -67
```

ще отпечатат в текущия ред, от текущата позиция нататък следното:

* 1 2 _ _ _ 3 4 5 - 6 7

4.6 Примери

Пример № 1. Ако регистър BL съдържа ASCII-кода на 16-тична цифра (0, 1, 2, ... , A, B, … или F), на неговото място да се запише двоичният еквивалент на стойността на цифрата.

Решение: Решението на всеки проблем, включително и на тази тривиална задача, е в синтезирания алгоритъм. И така, ако погледнем ASCII-таблицата, ще видим, че се иска следното:

При съдържание 30 (което е "0") да се запише 0 ;
При съдържание 31 (което е "1") да се запише 1 ;
и т.н.
При съдържание 39 (което е "9") да се запише 9 ;
При съдържание 41 (което е "A") да се запише 10 ;
и т.н.
При съдържание 46 (което е "F") да се запише 15 ;

Следователно, ако символите са от 0 до 9 техният числов еквивалент може да се получи като от кода им се извади кода на символ "0". А ако символите са букви, числовият еквивалент може да се получи като от техния код се извади кода на символ "A" и към полученото се прибави числото 10. Тъй като кодовете на символите са беззнакови числа, за тяхната проверка следва да се използват командите JA (по-голямо) и JB (по-малко). Текстът на програмата е следният:

```
      CMP   BL, "0"       ; сравнение на кода на символа с кода на "0"
      JB   LET            ; по-малко? Да - излизане, не - следващ ред
      CMP   BL, "9"       ; сравнение на кода на символа с кода "9"
      JA   LET            ; по-голямо? Да - излизане, не - следващ ред
      SUB   BL, "0"       ; BL:=(BL)-30h
      JMP   FIN
LET:  CMP   BL, "A"       ; сравнение на кода на символа с кода на "A"
      JB   FIN            ; по-малко? Да - излизане, не - следващ ред
      CMP   BL, "F"       ; сравнение на кода на символа с кода на "F"
      JA   FIN            ; по-голямо? Да - излизане, не - следващ ред
```

```
        ADD   BL, 10 - “A”     ; BL:=(BL)+10-41h  вторият операнд е израз
  FIN:  …  …  …
```

Пример № 2. Дадени са 3 двубайтови променливи A, B и C. При условие, че A,B>0 да се намери C=НОД(A,B).

<u>*Решение:*</u> Най-малкият общ делител на две числа ще намерим с алгоритъма на Евклид: докато тези числа не са равни, следва да се изважда от по-голямото по-малкото последователно до постигане на равенство. Ако равенството се постигне – това е НОД. Освен това ще се възползваме от обстоятелството, че командите за преход не променят флаговете, ето защо след команда за сравнение могат да се изпълнят неограничен брой команди за условен преход.

Текстът на програмата е:

```
        MOV   AX, A          ; AX := A
        MOV   BX, B          ; BX := B
  ND:   CMP   AX, BX         ; (AX) = (BX)?  Формиране на признаци
        JE    ND2            ; изход от цикъла, флаговете са непроменени
        JA    ND1            ; ако (AX) > (BX)   преход в точка ND1
        XCHG  AX, BX         ; запис в AX на по-голямото число
  ND1:  SUB   AX, BX         ; изваждане на по-малкото от по-голямото
        JMP   ND             ; затваряне на цикъла
  ND2:  MOV   C, AX          ; C := НОД(A,B)
```

Пример № 3. Дадена е еднобайтова променлива K, която има стойности от 1 до 18. Изисква се в регистър AL да се запише броят на двуцифрените числа (от 10 до 99), чиито цифри имат сума, равна на K.

<u>*Решение:*</u> С цел да избегнем операция деление, ще приложим алгоритъм с вложени цикли – по лявата и по дясната цифра на двуцифрените числа. На език Фортран алгоритъмът е следния:

```
DO  DH=1, 9
   DO  DL=0, 9
      IF  (DH+DL=K)  THEN  AL=AL+1
   END DO
END DO
```

За организиране на циклите в Асемблер ще използваме команда LOOP. Тъй като тя е твърдо свързана с регистър CX, а циклите са вложени, ще се наложи неговото съдържание многократно да се съхранява в отделни регистри, а след това и да се презарежда със стойността на всеки от броячите. След отчитане на тези съображения е получен следният окончателен текст на програмата:

```
        MOV   AL, 0          ; нулиране на брояча на резултата
        MOV   CX, 9          ; брояч на външния цикъл (нач. ст.)
        MOV   DH, 1          ; първа лява цифра
  L0:   MOV   BX, CX         ; съхраняване на брояча
```

```
; начало на вътрешния цикъл
        MOV   CX, 10            ; брояч на вътрешния цикъл (нач. ст.)
        MOV   DL, 0             ; първа дясна цифра (първо число 10)
  L1:   MOV   AH, DH            ; AH := (DH)
        ADD   AH, DL            ; AH := (AH)+(DL)
        CMP   AH, K             ; (AH) = K?   формира признаци
        JNE   L2                ; ако са различни, преход в L2
        INC   AL                ; отброяване
  L2:   INC   DL                ; следваща цифра (дясна, младша)
        LOOP  L1                ; ако има още цифри, на L1
; край на вътрешния цикъл
        MOV   CX, BX            ; възстановяване на външния брояч
        INC   DH                ; следваща цифра (лява, старша)
        LOOP  L0
```

Пример № 4. За въвеждане е зададена последователност от 200 двубайтови числа със знак. Да се определи колко пъти в тази последователност се среща най-малкото число. Да се изведе на екрана на монитора в един ред самото минимално число и броя му в последователността.

Решение: Алгоритъмът на програмата ще бъде следният: ще въведем първото число, което ще приемем за минимално и за срещащо се един път. След това в един цикъл ще въвеждаме по едно число, което ще сравняваме с намереното до момента най-малко. Възможностите са следните: ако числото е по-голямо, нищо повече не правим и минаваме на следващото число; ако е равно, отброяваме го и минаваме на следващото, но ако е по-малко, подменяме с него текущото най-малко, и определяме броя на срещанията му като единица.

В края минималното число се извежда като число със знак чрез командата OUTIN, а броят на срещането му – като число без знак с команда OUTWORD. За да не се “залепят” двата отпечатъка един за друг, за второто число ще посочим зона с по-голяма от необходимата му дължина. Ето текстът на програмата:

```
        N  EQU  200             ; начална стойност за брояча N
        OUTCH  “>”              ; подсказка към потребителя
        ININT   AX              ; AX:=първото число. То е min
        MOV    BX, 1            ; отброяване на min
  L1:   ININT   DX              ; DX:=следващото число
        CMP    DX, AX           ; сравнение с минималното
        JG    L2                ; число > min?  Ако да, то next
        JL    L3                ; число < min?  Ако да, то е new min
        INC     BX              ; отброяване на min
        JMP  L2                 ; скок към next
  L3:   MOV    AX, DX           ; подмяна на min
        MOV    BX, 1            ; отброяване на min
```

```
L2:  LOOP   L1                    ; цикъл (N-1) пъти по L1
     OUTIN   AX                   ; извеждане на min
     OUTWORD   BX, 5              ; извеждане на броя в 5 позиции
     NEWLINE                      ; курсор на нов ред
     FINISH
```

Пример № 5. Да се въведе последователност (възможно и празна) от символи, завършваща с точка. Да се определи дали символите в последователността са подредени в ненамаляващ ред като кодове. Резултатът да се отпечата като текст "ПОДРЕДЕНА" или "НЕ ПОДРЕДЕНА".

<u>Решение:</u> Ще въвеждаме по един символ (до точката), съхранявайки при това предидущия символ. Ако тази двойка символи (k, k+1) е неподредена, то окончателният отговор е готов, можем да прекратим цикъла, да изчистим буфера за въвеждане и да изведем съобщение. Ако двойката символи е подредена както е прието, тогава текущо въведеният символ се измества в мястото на предидущия и всичко се повтаря. Ако текущо въведеният символ се окаже точка, то цикълът завършва, а отговорът е за подредена последователност. Обърнете внимание, че първият отговор (думата подредена) изцяло се съдържа във втория отговор, което означава, че не е необходимо да се съхраняват два различни символни низа.

Отговорът ще се извежда като ред с команда OUTSTR. За целта създаваме в програмата символния низ "НЕ ПОДРЕДЕНА", който завършва със символа "$", и записваме в регистър DX началния адрес на този низ, за което определяме в програмата още една променлива, началната стойност на която е този адрес. Ако изходната последователност не е подредена, то веднага изпълняваме операция OUTSTR, в противен случай увеличаваме съдържанието на регистър DX на 3, за да пропуснем първите 3 символа на нашия низ "НЕ_". Така в DX ще се намира началният адрес на останалата част от низа, т.е. думата "ПОДРЕДЕНА". След това в този случай изпълняваме OUTSTR.

```
     ANS   DB  "НЕ ПОДРЕДЕНА", "$"      ; създава низът отговор
     AA   DW  ANS                ; адресът на низа ANS се поставя в AA
     MOV   DX, AA                ; DX := (AA)
     OUTCH  ">"                  ; покана за потребителя за въвеждане
     MOV   AL, 0                 ; задаваме код 0 за предидущ символ
IN:  INCH   AH                   ; вход на нов символ
     CMP    AH, "."              ; новият символ точка ли е?
     JE    YES                   ; ако да, преход в точка YES
     CMP    AL, AH               ; сравнение на новия със стария
     JA    NO                    ; по-голямо? Ако да – "не подредена"
     MOV    AL, AH               ; текущият символ става предидущ
     JMP   IN                    ; затваряне на цикъла
```

```
 NO:   FLUSH                    ; изчистване на буфера
       JMP   OUT
YES:   ADD   DX, 3              ; увеличаване на адреса с 3
OUT:   OUTSTR                   ; извеждане на отговор
       FINISH
```

Пример № 6. Да се реализира посимволно въвеждане (чрез команда INCH) на десетично число, което да се запише като двубайтова стойност на променливата N. Ще предполагаме, че числото е представимо и при въвеждане се изписва без грешки. В края на числото има празен символ.

Решение: След първия въведен символ ще проверим дали това не е знак. Ако символът е плюс или той не е знак, ще считаме числото положително. Знакът на числото ще запомним. Знакът на числото ще бъде отчетен след въвеждането на всичките му цифри. Тъй като числото се въвежда като десетично, а в разрядната мрежа то трябва да е двоично в допълнителен код, следва в процеса на въвеждане да се извърши преобразование, по схемата на Хорнер [1, 2].

След въвеждане се отчита знакът на числото и се получава допълнителният му код. Текстът на програмата е следния:

```
 S:   MOV    SI, 1              ; SI=1  за неотрицателни числа
      INCH   AL
      CMP    AL, “+”
      JE     D1
      CMP    AL, “-“
      JNE    DS                 ; няма знак -> в AL е първата цифра
      MOV    SI, 0              ; SI=0  за отрицателни числа
 D1:  INCH   AL                 ; въвеждане на първата цифра
; вход на цифри и формиране на модула на числото в регистър AX
 DS:  SUB    AL, “0”
      MOV    AH, 0              ; AX := първата цифра като число дума
      MOV    BX, 10             ; множител
      MOV    CH, 0
 D2:  INCH   CL                 ; вход на поредния символ
      CMP    CL, “  “
      JE     D3                 ; при празен символ край на входа
      SUB    CL, “0”            ; поредната цифра като число
      MUL    BX                 ; (DX,AX) := 10*(AX),    (DX)=0
      ADD    AX, CX             ; AX := (AX)+(CL),     (CL)=(CX)
      JMP    D2
; отчитане на знака
 D3:  CMP    SI, 1
      JE     D4
      NEG    AX                 ; допълнителен код на отрицателно
 D4:  MOV    N, AX
```

ГЛАВА 5

МАСИВИ. СТРУКТУРИ

5.1 Индекси

Вече беше пояснено, че в Асемблер масивите се описват чрез директивите за определение с помощта на конструкцията DUP. Но има още нещо за пояснение.

Нека е даден масив X от 30 елемента във формат дума:

X DW 30 DUP(?)

Както се вижда, в описанието е указан броят на елементите и типът, но не е указана организацията (номерацията) на елементите. Би следвало да се знае дали номерацията е от 0 до 29 или от 1 до 30 или е някаква друга. По условие номерацията може да е въведена, но ако не е или е възможно да бъде променена, то е добре да бъде избрана с начало 0. За да се поясни това предпочитание следва да се поясни връзката между индекса и адреса на елементите. Нека елементите са номерирани от k до 29+k:

X DB 30 DUP(?) ; X[k…29+k]

Тогава е вярно следното съотношение:

Адрес(X[i]) = X+2*(i-k)

или в по-общ вид, където явният размер (2) на елементите е скрит:

адрес(X[i]) = X+(type X)*(i-k) ,

където с i е означено индексното количество на този едномерен масив.

Тази зависимост става най-проста при k=0:

адрес(X[i]) = X + (type X)*i

Ето защо при програмиране на Асемблер обикновено се приема масивите да се номерират от 0:

X DW 30 DUP(?) ; X[0…29]

Тази именно номерация ще бъде спазвана тук.

За многомерни масиви номерацията е аналогична. Нека например да имаме двумерен масив (матрица) A(N,M) – с брой на редовете N и брой на стълбовете M. Елементите на масива са с формàт дума. Нека редовете са номерирани от k1, а стълбовете – от k2:

A DD N DUP(M DUP(?)) ; A[k1…N+(k-1), k2…M+(k2-1)]

Тук ще предполагаме, че елементите на матрицата са разположени в паметта по редове. При всички тези условия адресите на елементите влизат в следната функционална връзка с индексните количества i и j:

адрес(A[i, j]) = A + M*(type A)*(i-k1) + (type A)*(j-k2)

И в този случай горната зависимост се опростява аналогично при нулева начална стойност на индексиращите количества, т.е. при

k1=k2=0: адрес(A[i, j]) = A + M*(type A)*i + (type A)*j

5.2 Променливи с индекс

Във всеки език за програмиране означенията на елементите на масиви се придържат максимално близко към математическата абстракция. Индексните променливи са свързани с алгоритми, които са причина за индексния метод за адресиране, чиито теоретични основи са подробно изяснени в [1]. Става дума за машинни команди, изискващи преадресиране на своя операнд, както и за функционалността, по която се изчислява изпълнителният адрес.

5.2.1 Модификация на адресите

До момента адресите на операндите в командите се подразбираха като точни, например MOV CX, A. В общия случай обаче в командите вместо адрес може да бъде указан в квадратни скоби и регистър, например MOV CX, A[BX]. Изпълнителният (ефективният) адрес в този случай се изчислява по следната формула:

$$A_{ef} = (A+[BX])mod(2^{16}) \ ,$$

където към A се добавя съдържанието на регистър BX. Под A се разбира <u>адресът на променливата A</u>. Записът A[BX] можем да четем "*адресът е A модифициран по BX*"

Както се разбира, командата не съдържа явно адреса, но постоянно сочи къде са неговите елементи и как той да се получи. По този хардуерен начин извън командата е решен проблемът с нейното преадресиране. В разглеждания микропроцесор за целта могат да се използват само фиксирани регистри – BX, BP, SI и DI.

Нека разгледаме командата ADD A[SI], 5. Тук в ролята на модификатор е указан индексният регистър SI. Ако неговото съдържание е равно на 100, то изпълнителният адрес би бил:

$$A_{ef} = A+[SI] = A + 100 \ .$$

Тъй като съдържанието в индексиращия регистър се интерпретира като число със знак, то ефективният адрес от отместен спрямо A на разстояние, равно на индекса. Така чрез предварителна модификация на променливата част от адреса (вън от командата) една и съща команда може да работи в различни моменти с различни операнди.

5.2.2 Индексиране на операндите

Нека е даден масивът:

```
X  DW  100 DUP(?)           ; X[0…99]
```

сумата от чиито елементи трябва да се запише в регистър AX. Чрез X е означен началният адрес на масива. Задачата да се реши при предположението, че сумата като число не ще препълни формата на елементите на масива.

Решението е класическо – то представлява циклическо натрупване по формулата AX := (AX) + X[i], i=i+1. Тъй като елементите са двубайтови, адресите им се формират така:

$$адрес(X[i]) = X + 2*i$$

тогава, имайки предвид казаното в по-горния пункт, командата, която следва да се повтаря в тялото на цикъла, е:

```
ADD   AX, X[SI]    ,
```

където за модификатор на адреса на операнда е избран индексният регистър SI. Неговото модифициране следва да се изпълнява предварително със стъпка 2. Така окончателният програмен текст е:

```
        MOV   AX, 0          ; начална стойност на сумата
        MOV   CX, 100        ; начална стойност на брояча
        MOV   SI, 0          ; начална стойност на индекса i
L:      ADD   AX, X[SI]      ;  AX := (AX) + X[i]
        ADD   SI, 2          ; i = i + 2
        LOOP  L              ; затваряне на цикъла
```

5.2.3 Косвени указатели

Ще бъде изяснено още едно приложение на модифицираните адреси. Нека в ОП има променлива от тип дума, в която е необходимо да се запише числото 355, но нейният адрес е неизвестен. В същото време е известно, че той се намира в регистър BX. По същество адресирането на тази променлива е чрез косвена адресация – адресът не е известен, но се знае къде се намира. Така поставената задача изпълнява следната команда:

```
MOV   [BX], 355          ; (BX) := 355
```

Изписаната команда е коректна, защото типът на константата е явен. Ако обаче константата беше нула, то за Асемблер не е ясно дали се прехвърля байтова нула или нула с формат дума. Ето защо, ако искаме да сме коректни, следва да записваме:

```
MOV   BYTE   PTR  [BX], 0       ; прехвърляне на байт
MOV  WORD  PTR  [BX], 0         ; прехвърляне на дума
```

5.2.4 Модификация чрез няколко регистъра

Хардуерът на адресното АЛУ (вижте глава 1) позволява модифициране на променливата част от адреса с помощта на няколко регистъра. Тези възможности са реализирани в Асемблер и те изглеждат както следва: например записът

```
MOV   AX, A[BX][SI]
```

означава, че адресът на операнд A се изчислява по формулата:

$$адрес = (A + [BX] + [SI]) mod 2^{16} .$$

Модификацията на адрес чрез два регистъра обикновено се използва при работа с двумерни масиви. Нека, например, е дадена матрицата A с размери (10, 20) – десет реда, двадесет стълба:

```
A  DB  10 DUP(20 DUP(?))          ; A[0…9, 0…19]
```

и се изисква в регистър AL да се запише броят на редовете, в които първият елемент се среща още веднъж.

В ОП 20-те елемента от първия ред на матрицата заемат първите 20 байта, след тях са следващите 20 от втория ред и т.н. Така адресът на

произволен елемент от матрицата A(i, j) може да се изчисли по формулата A+20*i+j. За съхранение на индексиращото количество 20*i ще изберем регистър BX, а за второто индексиращото количество j – регистър SI. Тогава записът A[BX] представлява началния адрес i-тия ред, а записът – A[BX][SI] представлява адреса на j-тия елемент в i-ия ред. Най-общо алгоритъмът на задачата е циклически, с два вложени един в друг цикъла – търсене по редове и по стълбове на зададените условия, с отброяване на срещнатите такива в брояча AL.Текстът на програмата за тази задача е следният:

```
        MOV   AL, 0              ; начална стойност на брояча AL
; външен цикъл по редове, т.е. по индекса i
        MOV   CX, 10             ; брояч на външния цикъл
        MOV   BX, 0              ; отместване от A до реда (20*i)
   L:   MOV   AH, A[BX]          ; AH:= началния елемент от реда
        MOV   DX, CX             ; “спасяване” на брояча на редове
; вътрешен цикъл по стълбове, т.е. по индекса j
        MOV   CX, 19             ; брояч на вътрешния цикъл
        MOV   SI, 0              ; начална стойност на j
  L1:   INC   SI                 ; j:=j+1  модификация
        CMP   A[BX][SI], AH      ; проверка за   A[i, j] = (AH)?
        LOOPNE   L1              ; при различно, затваряне на L1
        JNE   L2                 ; на L2 ако няма съвпадение
        INC   AL                 ; отброяване при съвпадение
; край на вътрешния цикъл
  L2:   MOV    CX, DX            ; възстановяване на брояча CX
        ADD    BX, 20            ; начало на нов ред
        LOOP   L                 ; затваряне на външния цикъл
```

5.2.5 Правила за запис на модификациите

Следва строго да се дефинират правилата за запис на модификациите на адресите. Нека A означава адресен израз, а E означава произволен израз (адресен или константен). Тогава в Асемблер са допустими следните 3 основни форми за запис на адреси в командите:

A	$A_{ef} = A$	
E[M]	$A_{ef} = (E + [M]) \bmod 2^{16}$,	(M: BX,BP,SI,DI)
E[M1][M2]	$A_{ef} = (E + [M1] + [M2]) \bmod 2^{16}$,	(M1: BX,BP; M2: SI,DI)

Забележка: ако E=0, вместо 0[M] може да се записва [M].

Ще напомним, че не всеки регистър може да бъде модификатор. Следва да се използват само регистрите BX, BP, SI, DI. Когато модификацията е само по един регистър, модификатор може да бъде всеки един от изброените. Когато модификацията е по два регистъра, то единият е задължително базов (BX, BP), а вторият е задължително индексен (SI, DI).

Има забележка към регистър BP – този регистър се използва обикновено за адресиране на клетки в стека, за което ще пишем по-късно. Използването на този регистър за модификация на адреси в други области на ОП ще приемаме за невъзможно.

Модификацията на адреса не променя типа на адресната променлива. Например, ако X е байтова променлива, то:

TYPE X[SI] = BYTE

Ако модификаторът е предхождан от константен израз (например 1[BX]), типът на такъв адрес е неопределен.

Освен посочените 3 основни записа на адреси в Асемблер се допускат и други, които ще разгледаме по-късно.

В Асемблер са приети следните съглашения относно използването на квадратни скоби:

1. Затварянето на името на регистъра-модификатор в квадратни скоби е еквивалентно на изписване на неговото съдържание. Например:

```
A  DW  99
AA  DW  A
....  ....  ....
MOV   BX, AA            ; в регистър  BX  адреса на A
MOV   CX, [BX]          ; CX := A ,  т.е.  (CX)=99
MOV   CX, BX            ; CX := (BX),  т.е.  CX := адрес на A
```

Забележка: Да се поставят в квадратни скоби имена на регистри, които не са модификатори (AX,SP,DS,AL и пр.) не е позволено.

2. Всеки израз може да бъде затворен в квадратни скоби и от това неговият смисъл не се променя (но премахването на скобите може да измени неговия смисъл). Например:

```
MOV   CX, [2]           ; ≡ MOV  CX, 2
MOV   CX, [A]           ; ≡ MOV  CX, A
MOV   CX, [A+2[BX]]     ; ≡ MOV  CX, A+2[BX]
```

3. Следващите записи са еквивалентни

[x][y] = [x] + [y] = [x+y]

С други думи, изписването на два израза в квадратни скоби един до друг следва да се разбира като тяхна сума. Така например:

```
MOV  CX, [BX][SI]         ; ≡ MOV  CX, [BX]+[CX]   ≡ MOV  CX, [BX+SI]
```

На всеки от следващите редове са записани еквивалентни форми на един и същи адрес:

A+1, [A+1], [A]+[1], [A][1], A[1], 1[A], [A]+1, …
5[SI], [5][SI], [5]+[SI], [5]+[SI], [SI+5], [SI]+5, …
A-2[BX], [A-2]+[BX], [A-2+BX], A[BX-2], A[BX]-2, …
A[BX][DI], A[BX+DI], [A+BX+DI], A[BX]+[DI], …
0[BX][SI], [BX][SI], [BX]+[SI], [BX+SI], [SI][BX], …

И така, в Асемблер един и същи адрес може да бъде записан по раз-

лични начини, стига използваната за запис форма да е еквивалентна на някоя от основните 3, разгледани по-горе. В частност, в записите не трябва да има:

- Сума от два адреса (A+B) ;
- Сума, в която единият от операндите е име на регистър (BX+2) ;
- Забранени съчетания от регистри ([SI]+[DI]) ;
- Регистри, които не могат да бъдат модификатори (A[CX], 2[BL]) ;
- Имена или числа, записани непосредствено след скоби [ ] ([SI]5, [BX]A).

Освен това, адресът не може да бъде сведен до число ([5]), тъй като това би било константен израз, а не адресен.

Други ограничения няма и всеки програмист избира по свой вкус формите за запис. Тук по-нататък ние ще се придържаме към фòрмата, която започва с променлива, а останалите събираеми са поставени в скоби (например, A[SI+3]). Такива записи обикновено приличат на записите на индексни променливи в езиците от високо ниво, например елемент от едномерен масив: A[i+3].

Адресните изрази с модификатори могат да се използват както при запис на операнди в команди, така и при запис на някои директиви (EQU, PTR, и др.), но в никакъв случай не бива да се указват в директиви за определение на данни. Например, директивата

```
X   DW   1[SI]
```

е грешна, защото Асемблер е длъжен да изчисли нейния операнд още на етапа транслиране, за да постави неговата стойност в клетка X, но по това време съдържанието на регистър SI е, разбира се, неизвестно.

5.3 Команди LEA и XLAT

Команда LEA

При използване на регистри модификатори често се налага в тях да се записват адреси. Нека като пример, да е необходимо да запишем в регистър BX адреса на променливата X:

```
X  DW  88
```

За да направим това, можем да въведем нова променлива, чиято стойност да бъде адресът на X.

```
Y  DW  X
```

и след това да прехвърлим стойността й в регистър BX:

```
MOV   BX, Y
```

Този похват обаче е доста изкуствен, а това действие се среща често в програмите. Ето защо за целта е въведена специална команда за зареждане на регистър с адрес:

```
LEA   r16, A     ; Load Effective Address – зареждане на ефективен адрес
```

Тази команда изчислява ефективния адрес на втория операнд A и го записва в регистър r16, $r16{:=}A_{ef}$. Командата не променя съдържанието на флаговия регистър. За първи операнд на командата може да бъде

употребен всеки регистър с общо предназначение, а като втори операнд – всеки адресен израз (с или без модификатори).

Един от примерите, където командата LEA е полезна, е извеждането на екран на ред чрез макроса OUTSTR. Ще припомним, че тази операция изисква началният адрес на извеждания ред да се намира в регистър DX. Тук зареждането в DX на този адрес може да се изпълни с команда LEA:

```
S  DB  "a+b=c", "$"
....  ....  ....
LEA   DX, S              ; DX := адреса на S
OUTSTR                   ; ще бъде изведено:    a+b=c
```

Ще разгледаме особеностите на команда LEA. При това ще предполагаме, че в програмата са направени следните описания:

```
Q  DW  45
R  DW  -8
```

Преди всичко ще отбележим, че команда LEA много прилича на команда MOV, но между тях съществува следната принципна разлика: ако LEA записва в регистъра самия адрес, указан в командата, то MOV записва съдържанието на клетка от ОП с този адрес:

```
MOV  BX, Q               ; BX := (Q) ,     (Q)=45
LEA   BX, Q              ; BX := адресът на Q
```

В командата LEA вторият операнд може да бъде всеки адрес – и адрес на байт, и адрес на дума и т.н. Като втори операнд обаче не трябва да се указва константен израз или име на регистър:

```
LEA  CX, 88              ; грешка
LEA  CX, BX              ; грешка
```

Ако в качеството на втори операнд е указан модифициран адрес, то първо се изчислява изпълнителният адрес и едва след това се зарежда регистъра:

```
MOV   SI, 2
LEA    AX, Q[SI]       ; AX:= Aef = Q+[SI] = адрес(Q+2)=адрес(R)
```

В частност, от това можем да се възползваме за прехвърляне в някакъв регистър съдържанието на регистъра модификатор, увеличено или намалено с някакво число:

```
MOV  BX, 50
LEA   CX, [BX+2]          ; CX:=[BX]+2=50+2
LEA   DI, [DI-3]          ; DI:=(DI-3) ,   (аналогично SUB  DI, 3)
```

Команда XLAT

Тази команда също е свързана с модификациите на адреса. Команда XLAT изпълнява таблично преобразование, т.е. прекодиране (*translate*). Действието на командата се изразява в следното: съдържанието на клетка от ОП се записва в регистър AL. Особеното е в това, че адресът на клетката се формира като сума от съдържанията на регистрите BX и

AL, т.е. AL:=((BX)+(AL)). Изпълнението на командата не променя съдържанието на регистъра на флаговете. Както се вижда, съдържанието на регистър AL се подменя със съдържанието на клетката от ОП. При това съдържанието на регистър BX остава непроменено. Командата работи само с двойката регистри (BX,AL).

Това действие е необходимо когато се налага прекодиране на символи. Предполага се, че съществува множество от таблици, всяка от които съдържа кодовите комбинации на 256 символа. Ако са известни началните адреси на тези таблици, то съответствието между символите би могло лесно да се установява само по техния порядков номер i. Така, зареждайки в регистър AL номера на даден символ, а в регистър BX началния адрес на другата таблица, то сумата (BX)+(AL) ще представлява адреса на търсения символ, който след прочитане ще бъде зареден в регистър AL.

Ето едно приложение на командата: да се преобразува произволна шестнадесетична цифра (0,1,2,...,A,...,F) в символ. При това дадените символи са дефинирани в таблицата DIG16 както следва:

```
DIG16  DB  “0123456789ABCDEF”          ; DIG16[0…15]
```

Ако приемем, че в регистър AL е зареден поредния номер на търсен от дефинираната таблица символ (да кажем номер 14), това означава, че желаем да получим символа “E”. За целта трябва да запишем:

```
LEA  BX, DIG16        ; BX приема началния адрес на таблицата
XLAT                  ; AL := [DIG16[AL]] = ((BX)+(AL))
```

5.4 Структури

Когато говорим за данните, които програмите обработват, следва да имаме добра представа за тях. Най-елементарният вид са скаларните данни – една стойност (например, логическа, числова, символна и пр.). Вторият вид са множествата. Множествата с най-проста организация са масивите. Елементите на масивите обикновено са скаларни стойности. Често обаче като елементи на даден масив се налага да се използват множества, при това с разнообразен по характер (тип) набор от елементи. В такива случаи се говори, че елементите на масива имат своя структура, или те просто се наричат *структури*. В някои програмни езици такива елементи се наричат *записи* – масив от структури или масив от записи.

Описание на типа на структурата

Структурата следва да се разбира като съставен обект (елемент), който заема няколко съседни (последователни) клетки в ОП. Отделните компоненти на структурата се наричат *полета*. Полетата могат да са различни по тип (по размер, т.е. по формат) – байт, дума и пр. Полетата се именуват, за да е възможен директният достъп до тях.

Преди да бъде използвана една структура, тя трябва да бъде описана – колко полета ще има, с каква дължина и пр. Описанието

изглежда по следния начин:

```
<име на типа>      STRUC
<описание на поле>
   ....  ....  ....
<описание на поле>
<име на типа>      ENDS
```

Типът се открива с директивата STRUC (*structure*) и се закрива с директивата ENDS (*end of structure*). Между тези две директиви могат да бъдат записани произволен брой директиви, описващи отделните полета на структурата. Всяко поле се описва от една директива, която задължително определя данни – DB, DW или DD. Името, указано в такава директива, се възприема като име на съответното поле.

Например, типът структура DATE (календарна дата) разбираемо съдържа три полета: Y (*yare* – година), M (*month* – месец) и D (*day* – ден). Тази структура се декларира така:

```
DATE  STRUC
   Y  DW  2009
   M  DB  5
   D  DB  ?
DATE  ENDS
```

Описанието на структурата има чисто информационен характер (това е декларация). По него Асемблер не занася нищо в машинната програма, ето защо описанието може да се срещне в произволно място на текста, но задължително преди описанието на променливите в този тип. Имената на отделните полета следва да са уникални и не трябва да съвпадат с имената на други обекти в програмата. Полетата не могат да бъдат структури, т.е. Асемблер не допуска <u>вложени</u> структури.

Директивите, описващи отделните полета, могат да им назначават определена или неопределена стойност (вижте горния пример).

Имената на полетата имат стойност на отместване спрямо началото на структурата. Така може да изчислим, че структурата DATE заема 4 байта:

2	1	1	← Дължина на полета в байтове
Y	**M**	**D**	
+0	+2	+3	← Относително изместване в байтове

В резултат стойността на Y=0, на M=2, а на D=3. Срещайки в програмата тези имена Асемблер ще ги заменя с техните стойности.

Описание на променливи-структури

След като е определен типът на структурата, в програмата могат да бъдат дефинирани променливи от този тип, с което за тях ще бъде определяно адресно пространство в паметта. Тези променливи ще наричаме структурни. Те се описват с помощта на следната директива:

име на променлива <име на тип> <начална стойност {, начална ст.}>

Забележка: в тази конструкция ъгловите скоби <u>не са метасимволи</u>. Всяка начална стойност може да бъде: празно, ?, израз, низ. Например:

DT1 DATE <?, 6, 9>
DT2 DATE <1998, ,>
DT3 DATE <, ,>

Y	M	D
?	6	9
1998	3	?
2009	3	?

Тези директиви са особен вид. Разгледаните преди това директиви имаха за имена служебни думи, а тези тук се именуват от потребителя.

Всяка такава директива описва една променлива, чието име е най-отляво. За всяка променлива от типа DATE се заделят 4 байта в ОП. В ъгловите скоби се намира списъкът с началните стойности на всяко от полетата, подредени отляво надясно така, както са декларирани в типа на структурата. Естествено, началните стойности трябва да са представими в своите полета, в противен случай се фиксира грешка.

Правилата за задаване на начални стойности са следните:

- Ако за начална стойност е указан знакът ?, то полето не получава начална стойност, т.е. в него не се записва нищо;
- Ако е указан израз или низ, съответното поле получава за начална стойност изчислената за израза стойност или низа;
- Ако начална стойност не е указана, са възможни 2 случая:

1. Ако при описание на типа на структурата в това поле е указана стойност по подразбиране. Например, за полето M в структурните променливи DT2 и DT3.
2. Ако при описание на типа на това поле не е указана стойност по подразбиране, то променливата на това поле не получава начална стойност, т.е. остава неопределена. Например, такова е полето D в променливата DT3.

Защо са необходими стойности по подразбиране? На практика, доста често в различни структури от един и същи тип, някои полета трябва да имат еднакви начални стойности. Например за няколко променливи от тип DATE полето Y (година) може да има стойността 1998. И за да не се изписва тази стойност като начална за всяка променлива от този тип, тя се указва само веднъж, при описание на типа на структурата.

При деклариране на типа на структурните променливи могат да се използват някои съкращения. Например, ако последните полета не се указва стойност и остават само разделящите ги запетаи, то последните могат да се изпуснат:

DT2 DATE <1998, ,>   е еквивалентно на   DT2 DATE <1998>
DT3 DATE <, ,>   е еквивалентно на   DT3 DATE <>

Запетаи, които са в началото на списъка или в средата, не могат да се изпускат. Например:

DT4 DATE <, , 9> не е еквивалентно на DT4 DATE <9> и е грешка.

Последната забележка за този пункт се отнася до това, че с една декларация могат да бъдат да бъдат описани няколко структурни променливи, т.е. може да бъде описан масив, чиито елементи са структурни. За целта в директивата се указва списък или конструкция за повторение с оператор DUP. Например, конструкцията:

DTS DATE <, 12, 5>, 10 DUP (<>)

описва масив от 11 елемента-структури от типа DATE, при това всяко поле на първата от тях ще има следните начални стойности: 1998, 12, 5, а останалите10 структури един и същи набор от начални стойности, приети по подразбиране: 1998, 3, ?. При това името DTS получава само първата от 11-те структури, а останалите се достигат като елементи на масива със съответното отместване, т.е. чрез адресни изрази от вида: DTS+4, DTS+8, и т.н.

Указатели към полета в структурите

След описание на типа на структурата и променливи с нейния тип, програмистът получава правото да работи с тях. Като единно цяло структурите се обработват рядко, обикновено се налага да се обработват техните полета. За да се укаже дадено поле от структурата следва да се използва конструкцията:

<име на структурна променлива>**.**<име на поле> ,

в която ъгловите скоби са метаскоби. Пример: DT3.D

Такава конструкция сочи клетка от ОП, която съответства на името на полето. При това, типът на този адрес е равен на типа на данновото поле, за примера:

за полето D: TYPE DT3.D = BYTE,
а за полето Y: TYPE DT2.Y = WORD.

<u>Пример</u>: Ако в променливата DT1 се съхранява дата от месец март, то на нейно място се иска да се запише датата на следващия ден. Решението е в следния текст:

```
        CMP  DT1.M, 3
        JNE  FIN            ; не е месец март
        CMP  DT1.D, 31
        JE   APR1           ; последна дата, следва април
        INC  DT1.D          ; следваща дата
        JMP  FIN
APR1:   MOV  DT1.M, 4       ; подмяна на месеца
        MOV  DT1.D, 1       ; подмяна на датата
 FIN:   ....  ....  ....
```

Допълнения

В езика Асемблер има още няколко възможности, които ще бъдат уточнени тук по-долу.

Тип на името на структурата

Асемблер приписва на името на типа на структурата и на името на структурата тип (размер), равен на броя на байтовете, които тя заема. Например:

TYPE DATE = TYPE DT1 = TYPE DT2 = TYPE DTS

Във връзка с това, присвояването DT2:=DT1, ако е нежелателно по някакви причини да се укаже явно размерът на тези структури, може да бъде реализирано както следва:

```
        MOV   CX, TYPE  DATE           ; CX:=4 размер на структура DATE
        MOV   SI, 0                    ; брояч j:=0
; побайтно прехвърляне на DT1 в DT2
    L:  MOV   AL, BYTE  PTR  DT1[SI]       ; AL := (DT1[j])
        MOV   BYTE  PTR  DT2[SI], AL       ; DT2[j] := (AL)
        INC   SI                           ; j:=j+1
        LOOP  L                            ; цикъл по CX
```

Забележка: Тук оператор PTR е задължителен, тъй като размерите на операндите AL и DT1[SI], които са 1 и 4 съответно, са различни!

Оператор точка "."

Точката, която изгражда указателите към полетата, в същност представлява оператор на Асемблер, към който се извършва обръщение:

<адресен израз>**.**<име на поле в структура>

Този оператор се отнася към адресните изрази и означава адрес, който се изчислява по формулата:

<адресен израз> + <отместване на полето в структурата>

при което типът на този адрес съвпада с типа (размера) на указаното поле. Например:

DT1.D = DT1 + D = DT3 + 3
TYPE (DT1.D) = TYPE D = BYTE

Ще напомним, че Асемблер заменя името на полето с отместването на полето спрямо началото на структурата.

В общия случай адресният израз може да няма отношение към структурите или към указаното поле. Например, ако имаме описанието:

X DW ?

то е допустим записът X.M, въпреки че между X и M няма връзка. Този запис означава адреса X+M=X+2. Но да се увличаме по тази възможност не следва. За да има смисъл указването на полето, адресният израз естествено трябва да се позовава на структурата. При това изразът може да бъде с произволна сложност.

```
MOV   AX, (DTS+8).Y
MOV   SI, 8
INC   (DTS[SI]).M        ; Aef = (DTS+[SI]).M = (DTS+8).M
LEA   BX, DT1
MOV   [BX].D, 10         ; Aef = [BX]+D = DT1.D
```

Тук обаче трябва да се отчита следното: ако адресният израз не е име и не е косвен указател (от вида [BX]), то той следва да бъде затворен в малки скоби (). Става дума за това, че стойността (адресът) на указателя към полето ще бъде изчислен правилно и без скоби, но типът (размерът) на този адрес (по неизвестни причини) ще бъде определен от Асемблер неправилно. Така записите (DTS[SI]).M и DTS[SI].M задават един и същи адрес DTS+[SI]+M, но в първия случай Асемблер присвоява на този адрес типа на името M (т.е. 1), а във втория случай – типа на името DTS (т.е. 4).

Нескаларни полета в структури

Разглежданите до момента случаи се отнасяха само за полета, които съдържат скаларни стойности. Но в директивите DB, DW и DD могат да бъдат описвани няколко операнда, с това число и с конструкцията за повторение DUP. Подобни директиви могат да бъдат използвани за описание на полета в структури. Например, допустимо е следното описание на типа на структура:

```
STUD    STRUC              ; студент
 FAM    DB 10 DUP (?)      ; фамилия
NAME    DB "****"          ; име
  GR    DW ?               ; група
MARKS   DB 5, 5, 5         ; оценки
        ENDS
```

Тук за полето FAM се заделят 10 байта, за полето NAME – 4 байта, за GR – 2 байта и за MARKS – 3 байта.

И така, ако в директивата, която описва дадено поле, има повече от един операнд или конструкция за повторение, то при описание на променливата от този тип за това поле не трябва да се задава начална стойност, не трябва да се указва и знак за неопределена стойност (?) – съответната позиция в триъгълните скобки трябва да бъде празна. От това правило има само едно изключение: ако полето е описано като низ, то на такова поле може да се задава начална стойност, която задължително трябва да бъде низ, при това с дължина равна или по-малка (при недостатъчна дължина се добавят празни символи). Примери:

```
S1  STUD  <"Иванов", ...>     ; грешка (полето FAM не е низово)
S2  STUD  <, "Яна", 101>      ; S2.NAME:="Яна ",   S2.GR:=101)
```

```
S3  STUD  <, “Марина”>            ; грешка (S3.NAME е за 4 символа)
```

Причината за подобни ограничения е объркването, което може да настъпи, ако се разреши задаването на няколко начални стойности на няколко полета.

5.5 Примерни задачи

Ще бъдат представени примери за обработка на масиви и структури с използване на индексните регистри.

<u>*Пример №*</u> *1.* Деклариран е масивът X(N):

```
N   EQU   100
X   DW   N  DUP (?)              ; масив X[0…N-1]
```

Разглеждайки елементите на масива X като числа със знак, запишете в регистър AL индекса (от 0 до N-1) на максималния елемент в масива (на първия срещнат, ако такива еднакви има няколко).

<u>*Решение*</u>

Очевидно е, че решението е в циклически алгоритъм от вида с предварително известен брой повторения. На лице са проблемите за преадресиране на команди и работа с индексни регистри. Тъй като адресът на произволен елемент е (X[i])=X+2*i, то в регистъра, да кажем SI, който ще бъде използван като индексен, ще се съхранява променливата част от адреса – (2*i). Докато върви търсенето на максималния елемент и неговия индекс, ще работим с удвоения индекс и чак в края ще го разделим на 2.

```
        MOV  BX, X           ; BX := X[0]  начална стойност за Max.
        MOV  AX, 0           ; AX := 2*(индекс Max)
        MOV  SI, 2           ; SI := 2*i
        MOV  CX, N-1         ; начална стойност на брояча
 MAX:   CMP  X[SI], BX
        JLE   MAX1
        MOV  BX, X[SI]       ; нов Max
        MOV  AX, SI          ; неговия индекс
MAX1:   ADD   SI, 2          ; i := i+1
        LOOP  MAX
        MOV   CL, 2
        DIV    CL            ; AL := (AX) / 2  истинен индекс
```

<u>*Пример №*</u> *2.* Деклариран е масивът Y(N):

```
N   EQU   50
Y   DB   N  DUP (?)              ; масив Y[0…N-1]
```

Да се изместят циклически елементите на масива на 2 позиции напред, т.е. в положението:

Y[2], Y[3], …, Y[N-1], Y[0], Y[1]

Решение

Алгоритъмът очевидно е циклически. За изместването е необходима допълнителна клетка, в която временно да се извади текущият елемент от масива и на негово място да се прехвърли съответният, отстоящ на 2 позиции. След това в освободената клетка да се постави предварително извадения. За преадресиране на елементите ще използваме индексния регистър DI. Ще използваме и факта, че:

Адрес(Y[i]) = Y+i

Адрес(Y[i+2]) = Y+(ii2) = (Y+2)+i ,

при което, когато DI=i, i-тият елемент Y[i] ще има адреса Y[DI], а указването на елемент със стъпка +2 Y[ii+2] ще има адресния израз Y+2[DI] или Y[DI+2]. По този начин "добавката" +2 е по-лесно да се впише в командата, вместо да се отчита в индексния регистър.

```
        MOV   AH, Y             ; съхраняване на 2 елемента (0 и 1)
        MOV   AL, Y+1
        MOV   DI, 0             ; i := 0
        MOV   CX, N-2           ; начална стойност на брояча
SHIFT:  MOV   BH, Y[DI+2]
        MOV   Y[DI], BH         ; Y[i] := Y[i+2]
        INC   DI                ; i := i+1
        LOOP  SHIFT
        MOV   Y+N-2, AH         ; Y[N-2] := Y[0]
        MOV   Y+N-1, AL         ; Y[N-1] := Y[1]
```

Пример № 3. Деклариран е масивът S(N):

```
    N  EQU  65
    S  DB  N  DUP (?)        ; масив S[0…N-1]
```

Да се определи симетричен ли е масивът, т.е. равни ли са равно отдалечените от двата края елементи и ако да – да се запише в регистър AL единица, в противен случай – нула.

Решение

Решението е циклически алгоритъм, в тялото на който се повтарят действия сравнение на елементите S[i] и S[N-1-i] и проверка. Повторенията са N / 2 на брой. Чрез ii ще получим достъп до елемент от първата половина, а чрез (N-1-i) – достъп до съответния елемент от втората половина на масива. За улеснение ще въведем помощен индекс j=(N-1-i). Така в цикъла ще увеличаваме i с единица, а j ще намаляваме с единица.

```
        MOV  AL, 0
        MOV  SI, 0              ; i := 0
        MOV  DI, N-1            ; j := N-1
        MOV  CX, N/2            ; брой на сравняваните двойки
SYM:    MOV  AH, S[SI]
        CMP  AH, S[DI]          ; S[i]=S[j] ?
```

```
        JNE   FIN                   ; при не равно изход с (AL)=0
        INC   SI                    ; i := i +1
        DEC  DI                     ; j := j -1
        LOOP   SYM
        MOV  AL, 1                  ; (AL)=1 – масивът е симетричен
  FIN:  ....  ....  ....
```

Пример № 4. Зададена е последователност от главни латински букви за въвеждане, след която следва точка. Да се изведат в азбучен ред всички различни букви, влизащи в последователността.

Решение: Преди всичко ще опишем в програмата нулев байтов масив LET от 26 елемента – първият за буква A, вторият за B, последният за Z. Буквите ще въвеждаме подред, като за всяка от тях, в съответстващия й елемент от масива ще записваме 1. При завършване на въвеждането в масива LET ще бъдат отбелязани с 1 всички букви, които поне веднъж са били срещнати в изходната последователност. В края ще проверим елементите на масива, като за всеки елемент със стойност 1, ще изведем съответстващата му буква.

Ако записваме кода на въведената буква в регистър BX, то индексът на елемента от масива LET, съответстващ на тази буква, е равен на разликата между съдържанието на регистър BX и кода на буква A, ето защо адресирането на този елемент може да се извърши с израза LET[BX-“A”].

```
        LET  DB  26 DUP (0)          ; масивът LET се запълва с нули
        PROMT  DB  “Въведете букви: “, “$”
        ....  ....  ....
; въвеждане на букви и запълване на масива LET
        LEA   DX, PROMPT             ; покана за въвеждане
        OUTSTR
        MOV  BH, 0                   ; необходимо за разширение до BX
  IN:   INCH  BL                     ; BL := поредната буква
        CMP   BL, “.”
        JE    OUT                    ; ако е точка (край), към извеждане
        MOV  LET[BX-“A”], 1          ; буквата е разгледана
        JMP   IN
; извеждане на срещнатите в текста букви
 OUT:   MOV  CX, 26                  ; дължина на масива LET
        MOV  BX, “A”                 ; кода на A – в BX
OUT1:   CMP   LET[BX-“A”], 1         , запис на 1 в съответния елемент
        JNE   OUT2
        OUTCH   BL                   ; при 1 извеждане на буква
OUT2:   INC   BX
        LOOP   OUT1
        NEWLINE
```

<u>Пример №</u> 5. Нека FROM и TO са променливи с формат дума, а стойностите им да са адреси. Необходимо е да се копират 45 байта от ОП, започвайки от адреса, който се намира в FROM, в друга област на ОП, с начален адрес, указан в TO.

<u>Решение</u>

Ще означим адреса в FROM като ff, а адреса в TO като tt. Тогава според условието е необходимо да се прехвърлят байтове от клетки с адреси ff+ii, в клетки с адреси tt+ii, където ii се изменя от 0 до 44. Тези адреси са неизвестни предварително, при това е известно, че се променят, поради което могат да бъдат съхранявани само в регистри. В какви? Възможни са два варианта.

<u>Първи вариант</u>. Адресите ff и tt могат да бъдат записани например в регистрите SI и DI, където ще бъде удобно да бъдат инкрементирани:

```
        MOV  SI, FROM
        MOV  DI, TO
        MOV  CX, 45
COPY:   MOV  AH, [SI]          ; (SI) = ff+ii,   AH := (ff+ii)
        MOV  [DI], AH          ; (DI) = tt+ii,   tt+ii := (AH)
        INC  SI
        INC  DI
        LOOP  COPY
```

<u>Втори вариант</u> – индексните регистри съдържат пълните адреси ff+ii, и tt+ii. Тези адреси обаче бихме могли да разглеждаме като съставени от две части, всяка от които да се съхранява в отделен регистър. Едната част е константна, а другата променлива. Например константните части ff и tt можем да съхраняваме съответно в SI и в DI, а променливата част ii можем да съхраняваме в регистър BX. Тогава пълните адреси ff+ii и tt+ii ще можем да задаваме чрез изразите [SI+BX] и [DI+BX].

```
        MOV  SI, FROM
        MOV  DI, TO
        MOV  BX, 0
        MOV  CX, 45
COPY:   MOV  AH, [SI+BX]       ; (SI)+(BX) = ff+ii,   AH := (ff+ii)
        MOV  [DI+BX], AH       ; (DI)+(BX) = tt+ii,   tt+ii := (AH)
        INC  BX
        LOOP  COPY
```

Естествено възниква въпросът: кой е по-добрият вариант? Еднозначен отговор в случая няма. Първи вариант използва по-малко регистри, а вторият вариант изпълнява по-малко команди в цикъла.

<u>Пример №</u> 6.

Нека в програмата е описана структурата STUD (студент):

```
STUD    STRUC               ; студент
 FAM    DB 20 DUP (?)       ; фамилия
  GR    DW ?                ; група
MARKS   DB 5 DUP(?)         ; оценки
        ENDS
```

и нека в масива: S STUD 300(<>) да е събрана информация за 300 студента. Необходимо е да се определи броят на отличниците и да се запише в регистър DX.

Решение

Алгоритъмът е очевиден – преглеждаме последователно студентите, сравняваме оценките им и ако всички те са отлични (т.е. 6), то отброяваме дадения студент като отличник. За целта ще организираме брояч в регистър DX. Главният въпрос тук е – как да получим достъп до поредния студент и до неговите оценки?

Избираме още в регистър BX да съхраняваме отместването от началото на масива S до началото на поредния му елемент (структура). Така изразът S[BX] ще означава началния адрес на този елемент. Тъй като елементът е структура, то достъпът до неговото поле MARKS ще получаваме чрез конструкцията (S[BX]).MARKS. Това поле от своя страна е декларирано като масив от 5 елемента тип байт. Това означава, че като цяло структурата на алгоритъма на решението представлява два вложени един в друг цикъла. За индекс на този масив избираме регистър SI. Следователно достъп до поредната оценка на студента ще получаваме чрез израза (S[BX]).MARKS[SI].

Отчитайки казаното текстът на програмата е следният:

```
        MOV  DX, 0              ; брой на отличниците
; външен цикъл (по студенти от №0 до №299)
        MOV  CX, 300            ; общ брой на студентите
        MOV  BX, 0              ; начало от първия студент
   LS:  MOV  AX, CX             ; спасение на брояча CX
; вътрешен цикъл (по оценки 5 на брой)
        MOV  CX, 5
        MOV  SI, 0              ; брояч на оценките
   LM:  CMP  (S[BX]).MARKS[SI], 6
        JNE  NEXT
        INC  SI                 ; SI := (SI)+1  - следваща оценка
        LOOP  LM
        INC  DX                 ; отброяване на отличник
; към следващия студент
 NEXT:  MOV  CX, AX             ; възстановяване на брояча CX
        ADD  BX, TYPE STUD
        LOOP  LS
```

Г Л А В А 6

БИТОВИ ОПЕРАЦИИ. УПАКОВАНИ ДАННИ

Тук ще бъдат описани команди, които разглеждат своите операнди не като цели комбинации (стойности, кодове), а като последователности от битове. Това са команди, които изпълняват логически операции и операции изместванния. Често се говори, че операциите са поразрядни.

6.1 Команди за логически операции

Както всички останали команди и тези команди получават резултат и признаци на резултата. Признаците, които се формират от хардуерните схеми, интерпретират резултата винаги като число, ето защо не всички имат адекватна (актуална) стойност спрямо поразрядния резултат. Полезен може да бъде флагът Z, който сигнализира за нулево съдържание на регистъра, фиксирал резултата. Операндите в битовите операции трябва да бъдат от един и същи тип.

1. ***Команда NOT*** (логическо отрицание)

Командата за логическо отрицание е едноместна: (NOT op), където операндът op може да има типа r8, m8, r16, m16. Пример:

```
MOV   AL, 1100b          ; AL := 1100
NOT   AL                 ; (AL) = 0011
```

2. ***Команда AND*** (логическо умножение)

Командата за логическо умножение (конюнкция) е двуместна:

AND op1, op2 ; op1:=op1∩op2

В командата типовете на двата операнда могат да се съчетават по следния начин:

op1	op2
r8	i8, r8, m8
m8	i8, r8
r16	i16, r16, m16
m16	i16, r16

Пример:

```
MOV   AL, 10110110b      ;  AL := 10110110
AND   AL, 00001111b      ; (AL) = 00000110
```

3. ***Команда TEST*** (логическо умножение с цел проверка)

Командата TEST е двуместна, аналогично на командата AND. Изпълнението е същото с разлика, че полученият резултат не се фиксира. Целта на командата е формиране на стойности на флаговете и преди всичко на флага Z. Командата е удобна, когато е необходимо да се провери дали определена част от битовете на два операнда съвпадат или не. Например, ако ни интересува дали битовете в разрядите с четни номера на дадена комбинация са равни на 0, то формираме следната маска: M=01010101, с която умножаваме логичес-

ки друга комбинация К=10100011. Резултатът ще бъде:

```
  1 0 1 0 0 0 1 1
  0 1 0 1 0 1 0 1   ∩
  ---------------
  0 0 0 0 0 0 0 1
```

За този резултат ще се формира флаг Z=0, което означава, че не всички (само четните) проверявани битове са нулеви. Например:

```
MOV   AL, 10100011b
TEST  AL, 01010101b        Z=0
TEST  AL, 1100b            Z=1
```

Втората команда на горния текст изпълнява същата операция, но записаната като втори операнд маска 1100 първо се разширява до вида M=00001100, с което проверката се осъществява само върху дясната половина на комбинацията. Тъй като битовете №2 и №3 съдържат 0, то Z=1. Командата TEST се адаптира от програмиста съобразно нуждите. Например, ако е необходим преход в случай, че десните три бита в регистър AX съдържат единици, записваме командите:

```
TEST   AX, 111b
JZ   Label_02
```

4. *Команда OR* (логическо събиране)

Командата за логическо събиране (дизюнкция) е двуместна:

```
OR   op1, op2          ; op1:=op1∪op2
```

Пример:

```
MOV  CL, 1100b
OR   CL, 1010            (CL)=00001110b
```

5. *Команда XOR* (изключващо ИЛИ, логическа неравнозначност)

Командата за логическа неравнозначност или още сума по модул 2 (а⊕в) е двуместна:

```
XOR   op1, op2          ; op1:=op1⊕op2
```

Пример:

```
MOV  CL, 1100b
XOR   CL, 1010            (CL)=00000110b
```

Командата XOR се използва често за нулиране на съдържанието на клетка или регистър: XOR AX, AX ; (AX)=0

Изчисляване на логически изрази

Ще поясним първо, че за представяне на логическите стойности е необходим само един бит. В същото време, минималната порция, отделена в процесора за представяне на каквито и да било данни, е един байт. Ето защо няма стандартизиран вид за представяне на логическите стойности и всеки програмист би могъл да създава такива както сметне за добре. Избирането на формата за логическите стойности програмистът следва да подчинява на създадените операции, така че при използване на съответните команди, да получава

резултати, съответстващи на избраното от него представяне, тъй като в противен случай няма да получава верни резултати и няма да може да използва стойностите на флаговете. Например, ако изберем следното представяне:

0000 0000b за "лъжа" и 0000 0001b за "изстина",

то изпълнението на текста:

```
MOV  DH, 1b        ; (DH)=true
NOT   DH            ; (DH)=1111 1110b, което не е приетото за 0
```

По тези причини обикновено се предпочита традиционния формат – 8 нули за "лъжа" и 8 единици за "истина", което осигурява правилно изпълнение на командите и правилна интерпретация на флаговете. Например, ако са декларирани променливите:

```
A  DB  ?
B  DB  ?
```

и те се интерпретират като логически, тогава изчислението:

$$A = B \cap \overline{A} \ ,$$

може да се реализира чрез следния текст:

```
MOV  AL, A
NOT   AL
AND   AL, B
MOV   A, AL
```

На практика подобни изчисления се организират рядко. Тъй като логическите изрази се използват в условните преходи, то изразите се реализират последователно с команди за сравнение и за преход. Например, условният оператор:

if (X>0) *or* (Y=1) *then* go to L1

може да бъде запрограмиран така:

```
CMP   X, 0
JG   L1              ; X>0   go to L1
CMP   Y, 1
JE   L1              ; Y=1   go to L1
```

Най-често командите за логически операции се използват при обработка на опаковани данни.

6.2 Команди за изместване

Изместванията биват: логически, аритметически и циклически. Командите за изместване са двуместни. Първият операнд се разглежда като последователност от битове, които се изместват. Вторият операнд е число без знак и задава броя на разрядите за изместване, т.е. броя на еднобитовите измествания. Командите за изместване имат две форми:

<мнемокод> op, 1 или <мнемокод> op, CL

Допустими са следните типове за операнди: r8, m8, r16, m16.

Командите за изместване променят флаговете.

Логически измествания

Логическите измествания наляво и надясно

SHL op1, op2 SHR op1, op2

се характеризират с това, че в освобождаващия се разряд се записва нула, а всички излизащи от разрядната мрежа битове минават през бит CF.

Когато се измества дума в паметта, изпълнението на командата отчита обратния ред на байтовете.

Логическите измествания могат да се използват за бързо изпълнение на операции умножение и деление на цели числа. Когато множителят или делителят са степенни функции с основа 2, изпълнението може да се сведе до съответното изместване на толкова на брой разряди, колкото е степента на функцията. Например:

$op*2^k$ може да се изчисли така: SHL op, k

$op/2^k$ може да се изчисли така: SHR op, k

Програмистът носи отговорността както за настъпване на препълване, така и за верността на получаваните резултати. При операция деление понякога като резултат се търси остатъкът. За това ще бъде казано по-късно в пункт 6.3.

Аритметически измествания

Предназначението на командите за аритметическо изместване е бързото изпълнение на операции умножение и деление на числа със знак, при условие, че вторият им операнд е степенна функция с основа 2. Командата за аритметическо изместване наляво

SAL op1, op2

има същото изпълнение както командата за логическо изместване SHL. Разбира се интерпретацията на резултата като произведение може да бъде полезна само ако той е верен, т.е. ако запазва знака на множимото. Читателят навярно разбира, че в този случай множителят може да бъде само положително число.

Командата за аритметическо изместване надясно

SAR op1, op2

е различна в изпълнението си, което се дължи на знака на числото. При изместване на битовете в разрядната мрежа, ако резултатът трябва да се интерпретира като частно, той следва да притежава знака на делимото, т.е. говорим, че изместването е със запазване на знака. Тук отново трябва да поясним, че тази интерпретация има смисъл само за положителни делители, кратни на 2.

Използването на команда SAR за бързо деление има още една съществена особеност, която ще поясним с пример: нека търсим (-8)/2, което разбира се е (-4). Реализираме това така:

```
MOV   AL, -8      ; (AL) = 11111000b,   което е [-8]ДК.
SAR   AL, 1       ; (AL) = 11111100b,   което е [-4]ДК.
```

Нека сега по същия начин изпълним (-9)/2, което разбира се е (-4):

```
MOV   AL, -9        ; (AL) = 11110111b,   което е [-9]ДК .
SAR   AL, 1         ; (AL) = 11111011b,   което е [-5]ДК .
```

Както се вижда, резултатът не е верен в смисъла на целочисленото деление. В заключение може да се обобщи, че при използване на команда SAR за бързо деление на отрицателни числа, които не се делят точно на 2, полученото частно може да се интерпретира като закръглено в посока минус безкрайност.

Циклически измествания

Командите за циклическо изместване наляво и надясно

ROL op1, op2   ROR op1, op2

се характеризират с това, че излизащият от разрядната мрежа бит не се губи, а се записва от другата й страна, в освободения разряд. И в двете посоки излизащият бит се дублира във флаговия разряд CF.

Тези команди са удобни за размяна на отделни части на разрядното поле. Например, заменянето на лявата половина на съдържанието на регистър AL с дясната половина може да стане чрез циклическо изместване наляво (или надясно):

```
MOV   AL, 57h          ; (AL) = 01010111b
MOV   CL, 4
ROL   AL, CL           ; (AL) = 01110101b = 75h
```

Разработени са още две команди за циклическо изместване, изпълнението на които се характеризира с това, че в кръга на изместването се включва флаговия разряд CF. Така изместването се осъществява през бит C:

RCL op, 1   RCR op, 1

Тези команди са удобни за пренос на битове от един регистър (или клетка) в друг. Например: да се изместят на 3 разряда вляво съдържанията на регистрите AL и DH, така, че в младшите 3 бита на AL да се окажат 3-те старши бита на DH. Решението е в следния текст:

```
      MOV   CX, 3
L:    SHL   DH, 1
      RCL   AL, 1
      LOOP  L
```

Командите за изместване са развити и допълнени в следващите модели микропроцесори на фирмата *Intel*.

6.3 Опаковани данни

Една от главните причини, поради която дадена задача програмират на Асемблер, а не на език от високо ниво, това е силният недостиг на оперативна памет. А един от основните методи за икономия на памет е използването на опаковани данни.

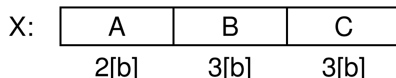
Същността на този метод се състои в това, че в една клетка се съхраняват стойности на няколко величини (ако това е възможно, разбира се). Ще подкрепим тази възможност с по-рано давания тук пример за величината DATE, която е структурирана в три полета Y(година), M(месец) и D(ден). Ако се възприеме представянето на годината чрез младшите ѝ две десетични цифри, то структурната величина DATE може да се представи вместо в 3 последователни байта (по един за всяко отделно поле), в поле със значително по-малка дължина. Като имаме предвид, че стойността на датата не е по-голяма от числото 31, то минимално необходимото за нея поле трябва да има дължина 5 бита. Съответно, за представяне на месеца – 4 бита, и за представяне на годината – 7 бита. Така общата минимална дължина на двоичното поле, необходима за представяне на величината DATE, е равна на 7+4+5=16 бита. Така е спестен цял един байт. Подобен род данни се наричат опаковани.

Разбира се, този начин на представяне на данните затруднява непосредствения достъп до тях, тъй като дължината на отделните полета не съобразена с възможните дължини на форматите за разрядната мрежа – машинната дума може да бъде четена или записвана нацяло. Следва да се разрешат два проблема:

- Как да отделяме части от съдържанието на клетките ;
- Как да съставяме съдържанието на клетките.

Отделяне на части от машинната дума

Нека като пример байтовата величина X е структурирана така:



и е необходимо е ли съдържанието на полето B равно на 5=101b.

Ясно е, че няма команди, с които да проверим това отношение. Ето защо постъпваме по следния начин: ще унищожим онези битове, които не са интересни за задачата и тогава ще изпълним сравнение. Унищожаването на части от разрядната мрежа се постига чрез тяхното маскиране. Съставяме маската M=00111000, в която в съответствие на всички прикрити битове стоят нули, а в съответствие на всички видими битове – единици. Резултатът от маскирането се получава чрез логическо умножение (команда AND).

Така програмният текст, решаващ задачата, е следният:

```
      MOV   AL, X
      AND   AL, 00111000b
      CMP   AL, 00101000b
      JE    YES
NO:   ....  ....
```

Съставяне на машинна дума от отделни части

Пример: дадени са два байта B1 и B2, от които трябва да се състави следният байт B3:

B1:	X	0	Z
B2:	0	Y	0
B3:	X	Y	Z

Лесно е да се досетим, че при това нулево съдържание на отделните полета, резултатът би се получил с операция логическо събиране. Така програмният текст е следният:

```
MOV   AL, B1
OR    AL, B2
MOV   B3, AL
```

Това обаче е частен случай. Ако в изключваните полета се съдържаше нещо различно от нула, горният текст няма да реши задачата. Необходимо е истинско прикриване на изключваните части, така че решението в този случай е комбинация от двете предидущи решения:

```
MOV   AL, B1
AND   AL, 11000111b        ; в AL остават само X и Z
MOV   BL, B2
AND   BL, 00111000b        ; в BL остава само Y
OR    BL, AL
MOV   B3, BL
```

И още един пример, обещан от предходния пункт – определяне на остатъка при целочислено деление. За без знакови числа, при делител от вида 2^k, остатъкът представлява (делимо)mod(2^k). За целта, бързото определяне на остатъкът, се свежда до отделяне на младшите k бита на делимото. Например, при деление на $8=2^3$, остатъкът се отделя с командата:

```
AND   AX, 111b
```

6.4 Множества

Множествата, като съставен тип данни, са добра основа за примери, обработващи опаковани данни. В компютърните системи няма стандартен формат за представяне на множества, ето защо програмистът сам трябва да организира това, както и операциите върху тях. Тук ще бъде изложен един начин за това.

6.4.1 Машинно представяне на множества

Нека имаме множество от цели числа **M**, които принадлежат на интервала [L, R]. Ще заделим за елементите на това множество няколко съседни байта. Освен елементите на множеството, които са числа от указания интервал (но не всичките), следва да се състави указател за

принадлежност на едно число към множеството. Указателят ще представлява последователност от битове (1 – принадлежи, 0 – не принадлежи). Указателите ще подреждаме в същата последователност, в каквато са и числата. Нека границите са L=10 и R=20, а множеството от числа е: **M**=[10,12,17,19]. Така съответствието ще бъде маркирано според следната схема:

У	1	0	1	0	0	0	0	1	0	0	1	Не се използва
M:	10	11	12	13	14	15	16	17	18	19	20	

Като имаме лявата и дясната граница на интервала, броят на байтовете за указателите У може да се определи чрез формулата:

(R-L)/8 + 1

Така променливата У може да се декларира със следния оператор:

У DB (R-L)/8+1 DUP(0)

Указателят за принадлежността на дадено число X от интервала се намира в поредния байт, поредния бит. Местоположението на този указател се определя така:

номер на байта = (X-L)/8 ;
номер на бита = (X-L)mod(8) .

Примерните съответствия са:

X=12: номер на байт = (12-20)/8 = 0 ;
номер на бита = (12-20)mod8 = 2
X=18: номер на байт = (18-20)/8 = 1 ;
номер на бита = (18-20)mod8 = 0 .

6.4.2 Операции върху множества

Определяне на принадлежност

Ще разгледаме 2 операции – проверка на принадлежността на елемент към множеството и обединяване на множества.

Задача 1: Нека имената L и R са описани в една програма като константи (L<=R) и нека са декларирани множеството **M** и променливата K:

```
M  DB  (R-L)/8+1  DUP (?)        ; set of  L…R
K  DW  ?                         ; L<=K<=R
```

които са получили в програмата някакви стойности. Да се определи дали числото K принадлежи на множеството **M**.

Съгласно приетото по-горе представяне на множествата, ако числото принадлежи на множеството, то в съответния му бит ще има записана 1, ако не принадлежи – 0. Следователно решението се свежда до проверка на съдържанието на бита за съответствие. За да се направи това е необходимо най-напред да бъде отделен този байт, в който се намира интересуващият ни бит. И двете позиции ще определим с помощта на формулите от предходния пункт. И така, имаме следния текст:

```
MOV   AX, K
SUB   AX, L
MOV   BH, 8
DIV   BH                ; (AH)= № на бита,  (AL)= № на байта
MOV   BL, AL
MOV   BH, 0             ; (BX)=№ на байта (начало на M) като дума
MOV   AL, M[BX]         ; AL := съответният байт
MOV   CL, AH            ; (CL) :=№ на нужния бит (броено отляво)
SHL   AL, CL            ; изместване на бита в левия край на AL
TEST  AL, 80h
JZ    NO                ; преход към NO (K не принадлежи)
....  ....              ; K принадлежи
```

Операция обединяване

Нека са дадени множествата M, M1 и M2, чиито елементи са числа от интервала [L, R]. Иска се обединение на M1 с M2 и запис в M.

Тъй като и трите множества имат един и същи интервал, то задачата се свежда само до промяна в битовете за принадлежност на множеството M, които следва да се актуализират като приемат битовете за принадлежност на другите две множества. Тази актуализация може да се постигне чрез операция логическо събиране последователно на всички байтове, съдържащи битове за принадлежност в множеството M със съответните байтове за другите две множества.

```
    MOV  CX, (R-L)/8+1   ; брой на байтовете за множествата
    MOV  BX, 0           ; № на поредния байт за принадлежност
L:  MOV  AL, M1[BX]
    OR   AL, M2[BX]
    MOV  M[BX], AL
    INC  BX
    LOOP L
```

По аналогичен начин могат да се реализират и останалите теоретико-множествени операции (пресичане, изваждане и пр.), при което ще се наложи изпълнението на други логически операции.

6.5 Записи

За работа с опаковани данни Асемблер предлага допълнителни средства в лицето на структура, наречена запис.

Под запис се разбира опакована структура, чиито полета заемат части от клетките. При това размерът на структурата може да бъде само байт или дума. Отделните полета в записа са конкатенирани. Ако сумата от дължините на отделните полета не е 8 или 16, остават неизползвани битове в старшата (лявата) част на записа. Полетата се именуват, но техните имена не осигуряват достъп до тях.

6.5.1 Деклариране на записи

Преди в една програма да се използват записи, те следва да бъдат декларирани. За целта се използва следната директива:

<име на типа на записа> RECORD <поле> {, <поле>}

където под <u>поле</u> се разбира:

<поле> ::= <име на поле> : <размер> [= <израз>]

а под <u>размер</u> и <u>израз</u> се разбират константни изрази (неопределена стойност “?” не се допуска). Например:

REC RECORD A:3, B:3=7

	A	B	: име на поле
////	0	7	
	3	3	: размер на поле

DATE RECORD Y:7, M:4, D:5

Y	M	D	: име на поле
0	0	0	
7	4	5	: размер на поле

При описване на типа на записа всички негови полета се изреждат последователно отляво надясно. За всяко поле се указва име, което е последвано от двоеточие и размер (дължина) в битове. Имената на полетата трябва да се различават от всички други имена, използвани в програмата. При описание на полетата може да бъде употребен знак равно “=”, а след него стойността му (по подразбиране). Ако няма указана стойност, тя е нула (0). Определяне на стойност 0 по подразбиране отличава записа от структурата, в която стойността по подразбиране е неопределена (?). Това е така, защото в записа не възможно да не се запише нещо, ето защо по тази причина с нули са запълнени и неизползваните битове, останали отляво.

Описанието на типа на записа може да се постави на всяко място в програмата, но задължително преди употребата на имената на полетата му.

6.5.2 Описание на променливи от тип запис

Променливите-записи обикновено се наричат само записи и за описание на такива променливи се използва следната директива:

Име_на_променлива Име_на_тип <начална ст.> {, <нач.ст.>}

където <начална стойност> е константен израз, ? - неопределена стойност или празен символ. Ъгловите скоби са задължителни. Пример:

```
R1   REC  <5,>            ; R1:  0  5  7
R2   REC  <, ?>           ; R2:  0  0  0
VIC  DATE  <45, 5, 9>     ; VIC:  45,  5  9
```

Смисълът на подобни директиви е аналогичен на директивите описващи променливи-структури: използват се за заделяне на необхо-

димата памет (байт или дума) за всеки запис. Директивата дава име (адрес) на записа, както и начална стойност за отделните полета. По различен начин обаче се разбира знакът за неопределена стойност “?”, който назначава за начална стойност на съответното поле числото нула (0). При празен символ за начална стойност се приема стойността указана по подразбиране, т.е. при деклариране на типа на записа.

При задаване на началните стойности са допустими следните съкращения:

```
D1  DATE  <97, ,>        ; е еквивалентно на  D1  DATE  <97>
D2  DATE  <, ,>          ; е еквивалентно на  D2  DATE  <>
```

Както и при описание на структури, и тук с една директива може да бъде описан масив от записи, за което в дясната част на директивата може да се изпише списък от операнди и/или конструкция за повторение DUP. Например директивата:

```
X  DATE  100 DUP(<>)
```

ще бъде причина за запазване на място за 100 записа, които ще получат еднакви начални стойности, указани при деклариране на типа на записа DATE.

6.5.3 Средства за работа с полетата на записи

Затруднението при работа с полета в записи се дължи на това, че те, като битови части от записа, не могат да бъдат адресирани.

Най-напред ще отбележим, че работата със записи като цели обекти не среща никакви затруднения. Записът е байт или дума – дължина, която е кратна на дължината на клетка. Например присвояването R1:=R2 може да се запрограмира така:

```
MOV  AL, R2
MOV  R1, AL
```

За работа с отделни полета се използват няколко оператора:

Оператор ***WIDTH***:

```
WIDTH  <име на поле в запис>
WIDTH  <име на запис или негов тип>
```

Ако е указано име на поле, то стойност на оператора е дължината на полето в битове, а ако е указано името на записа или името на типа му, тогава операторът има стойността на цялата дължина на записа (сумата от дължините на полетата). Например:

```
WIDTH  Y = 7
WIDTH  REC = 6
```

Оператор ***MASK***:

```
MASK  <име на поле в запис>
МАСК  <имена запис или на негов тип>
```

Стойността на оператора е маска (байт или дума), съдържаща 1 в онези разряди, които принадлежат на указаното име или на всички полета, както и нули във всички останали разряди. Например:

```
MASK   A          = 00111000b
MASK   B          = 00000111b
MASK   REC        = 00111111b
MASK   M          = 0000000111100000b
```

Оператор MASK се използва за отделяне на поле в запис и улеснението в него се състои в автоматичното формиране на маската, която се налага върху записа, за да се отдели нужното поле. Например, ако е необходимо да се провери дали полето D в записа VIC съдържа числото 9, то постигаме това със следния текст:

```
     MOV   AX, VIC
     AND   AX, MASK  D
     CMP   AX, 9
     JE    YES
NO:  ....  ....  ....
```

Стойност на името на поле

Името на всяко поле в запис получава от Асемблер някаква стойност. Тя представлява число, която показва на колко бита следва да се измести полето надясно, за да се окаже дясно подравнено в рамките на размера на записа. Например, стойността на името D (от записа DATE) е числото 0, на името M – числото 5, на името Y – числото 9. Срещайки името на поле на запис, Асемблер заменя името с тази стойност, което е необходимо, когато трябва да се изместват полета.

Например, да се провери равно ли е на 5 полето M в записа VIC:

```
     MOV   AX, VIC          ; AX:  Y  M  D
     AND   AX, MASK  M      ; AX:  0  M  0
     MOV   CL, M
     SHR   AX, CL           ; AX:  0  0  M
     CMP   AX, 5            ; M=5 ?
     JE    YES
NO:  ....  ....  ....
```

Г Л А В А 7

ПРОГРАМНИ СЕГМЕНТИ

Тук малко по-подробно ще се спрем на адресирането и разпределението на програмните части в паметта.

7.1 Сегментиране на адресите

Още в глава 1 изяснихме, че различието между дължината на разрядната мрежа и дължината на адреса на този микропроцесор, се е наложило да се въведе структуриране на адресното пространство или още сегментиране. Всеки 20 битов адрес се получава по формула (1.2.1), която е основна за относителния метод на адресиране. В сегментен регистър се съхранява базов адрес, а в командата отместването относно него. Сегментните регистри са 4: CS, DS, SS, ES. Разбира се, базови адреси могат да се съхраняват и в други регистри, но формирането на изпълнителните адреси с тяхна помощ не е възможно, предвид хардуерните схеми на адресното АЛУ. Ако се наложи използването на допълнителен сегмент в паметта, пети например, то за целта може да се работи с наличните сегментни регистри, като съдържанието на избрания се съхранява в нарочно определено място и преди всяко използване се изпълнява преинициализация.

Взаимното разположение на сегментите в адресното пространство няма ограничения. Те могат да се пресичат или напълно да се припокриват. В първия случай разликата между съдържанията на два сегментни регистъра е по-малка от 2^{16}, а във втория случай – разликата е нула, т.е. (SR1)=(SR2). Клетките от общата област на сегментите могат да се адресират по различни сегментни регистри, което е право на програмиста. Размерите на сегментите също могат да се определят от програмиста, стига да не се нарушава ограничението отгоре (64[KB]).

При записване на командите на Асемблер, за указване на използвания сегментен регистър може да се използва следната конструкция:

<име на сегментен регистър> **:** <адресен израз>

която се нарича "адресна двойка". Например в командата

MOV AX, ES:X

адресната променлива X ще се "сегментира" относно съдържанието на регистър ES.

Записите CS:, DS:, SS: и ES: е прието да се наричат "префикс на сегментния регистър" или просто *префикс* (представка). В Асемблер префиксът винаги се записва преди адреса, който трябва да бъде сегментиран, т.е. определен в съответната област. В машинният език обаче префиксът стои пред цялата команда и за дадения пример то ще изглежда като две команди:

```
ES:
MOV   AX, X
```

където ES: може да се разглежда като специална команда (без операнди), която всъщност се нарича префикс. Такива команди има само 4, колкото са сегментните регистри. Командата носи необходимата на адресното АЛУ информация за предстоящото сегментиране. Ако префиксът е поставен пред команда, в която няма адресен операнд (т.е. няма обръщение към паметта), то префиксната команда ще се възприеме като “празна” команда, т.е. без последствия от нея.

Както вече беше пояснено, адресите, указани в командите, могат да бъдат модифицирани и чрез съдържанието на още 4 регистъра: BX, BP, SI и DI. Ще поясним как се съчетават правилата за модифициране с правилата за сегментиране.

Първоначално се изпълнява модификация на адреса според съдържанието на модифициращия регистър, в резултат на което се получава адрес, наричан *изпълнителен*. Изпълнителният адрес е 16 битов и се разбира като отместване. Отместването е променлива величина, която програмите управляват и преизчисляват с използване на съответния хардуер. Модифицирането на отместването се налага да се извършва, когато адресът на една величина се определя спрямо адреса на друга величина в същия сегмент. Например, ако знаем адреса на променливата X, и е казано, че тя е от тип DD (4 байта), и е казано още, че веднага след нея е поместена стойността на друга променлива Y, то е ясно, че за да прочетем стойността на Y, следва да изчислим адреса на Y като сума от адреса на X+4. Адресът на X представлява отместване спрямо сегментния адрес, а 4 е отместване на Y спрямо X. Адресът на Y е в същото време отместване на Y спрямо същия сегментен адрес. Така модифицирането на адреса на X дава изпълнителния адрес на Y.

На следващото ниво се извършва изчисляване на *ефективния* адрес по формула (1.2.1), т.е.:

A_{ef} = (16 битов сегментен регистър)*16 + 16 битов изпълнителен адрес

Вижте свързаните с тези понятия обяснения, изложени в глава 1. Във връзка с казаното формула (1.2.1) може да бъде обобщена така:

$$A_{ef} = \{ (SR)*2^4 + A_{изп} \} \bmod(2^{20}) . \qquad (7.1.1)$$

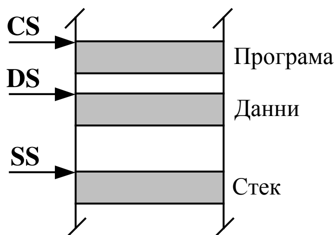
Напомняме, че сегментирането, т.е. добавянето на $(SR)*2^4$, се извършва само ако командата прави обръщение към ОП. Например, командата LEA в текста:

```
Q  DW  0A08Fh
LEA  BX, Q
```

записва 16 битовото отместване Q в регистър BX и не се обръща към паметта, така че стойността Q не се сегментира.

7.1.1 Сегментни регистри по подразбиране

В реалните програми болшинството обръщения към ОП са в областта на кодовия сегмент, в областта на даннови сегмент и в областта на стековия сегмент. В резултат на това е приета уговорката за незадължителното указване на сегментния регистър, който съдържа базовия (сегментния) адрес. Прието е базовият адрес на кодовия сегмент да се съдържа в регистър CS, данновия – в регистър DS, а стековия – в регистър SS. Разпределението на тези 3 области в адресното пространство обикновено е следното:



Що се отнася до допълнителния сегмент, няма специални уговорки. Този сегмент се приема за свободен и в регистър ES може да се намира адрес на всеки сегмент в паметта.

Следващата уговорка се отнася до адресите за преход. Адресите за преход се използват в командите за управление на прехода и по същество са адреси на команди, получаващи управление. Следователно това са обръщения към клетки от кодовия сегмент и не е необходимо изрично да се указва използването на кодовия сегментен регистър CS. Например, в командата за безусловен преход е достатъчно указването само на етикета: JMP L. Префиксът CS: ще бъде добавен автоматично при изчисление на ефективния адрес (JMP CS:L). Аналогично се опростяват обръщенията към данновата област: например при запис на стойност в променливата X, вместо командата (MOV DS:X, 0), е прието да се пише (MOV X, 0). Аналогична уговорка е приета за обръщението и към стека.

Обобщените правила за избор на сегментен регистър са следните:

- Адресите за преход винаги се сегментират относно регистър CS;
- За командите, работещи с низове, действат особени правила (ще бъдат разгледани в глава 10);
- За всички останали команди:

1. Ако адресът в командата на се модифицира, или се модифицира, но сред модификаторите не се използва регистър BP, то се приема, че обръщението е по адрес в данновия сегмент, поради което сегментирането се изпълнява спрямо регистър DS;
2. Ако адресът се модифицира чрез регистър BP, той се приема за указател към стека и подразбиращо се сегментира относно регистър SS.

Трябва да се изложи едно допълнение, което е свързано с косвените преходи. Ще разгледаме текста:

```
        X   DW  L
        ....  ....  ....
        JMP   X              ; JMP  DS:X  ≡  go to CS:L
        ....  ....  ....
L:      ....  ....  ....
```

В командата за преход JMP, името X не е адрес за преход, а адрес, в който се съдържа адресът за преход. Това следва от директивата DW, която определя адресът X като косвен адрес. Ще напомним, че при косвена адресация се извършват две последователни обръщения към паметта. Така в тази примерна команда обръщението по X се сегментира по регистър DS, а следващото сегментиране на адреса за преход L, се сегментира по регистър CS.

Ако все пак в командата бъде явно указан сегментен регистър, то той има предимство пред горе изложените правила.

Една от уговорките не бива да се нарушава по никакъв начин: регистър CS винаги следва да сочи началото на кодовия сегмент. Тъй като програмният брояч се сегментира автоматично само по този регистър, неговото съдържание не бива да бъде подменяно.

Възниква въпросът: има ли случаи, когато указването на сегментния регистър е задължително? Отговорът е: да, има, и те са два:

1. Когато в частен случай искаме да нарушим подразбиращите се правила. Например: искаме в регистър AX да запишем съдържанието на клетка L, намираща се в кодовия сегмент. Използването на команда MOV AX, L е погрешно, тъй като адресът L ще се сегментира спрямо DS. Записът, който постига желаното е командата MOV AX, CS:L
2. Когато се налага обръщение в сегмент, различен от кодовия, данновия или стековия. За целта първоначално е необходимо в регистър ES да се зареди начален адрес. В последствие всяко обръщение следва задължително явно да се сегментира спрямо този регистър. Например инкремент на стойността на променливата Y в допълнителния сегмент се реализира от командата INC ES:Y

7.2 Програмни сегменти

Определянето на сегментите за една програма извършва Асемблер, но информацията за тях дава програмистът. Това се прави по следния начин: всички предложения, за поместване в един сегмент, програмистът е длъжен да обедини в един <u>*програмен сегмент*</u>. Програмният сегмент има следната структура:

```
<име на сегмент>  SEGMENT <параметри>
                  <предложение>
                   ....  ....  ....  ....  ....
                  <предложение>
<име на сегмент>  ENDS
```

Името на програмния сегмент се повтаря два пъти – в директивата SEGMENT и в директивата ENDS. Броят на предложенията между тези две директиви е неограничен. Смисълът на тази конструкция се състои в това, че на Асемблер се заявява, че всички предложения в конструкцията той следва да помести в един сегмент на оперативната памет (с обем до 64[KB]). Ако обемът на указаните в конструкцията надхвърли 64[KB], ще бъде съобщено за грешка.

В програмата може да има няколко сегмента с едно и също име. При това положение се приема, че е на лице описан по части един сегмент, т.е. всички предложения Асемблер ще обедини в едно. Параметрите на директивата SEGMENT са необходими, когато програмата е голяма и се съхранява в няколко файла. Тогава чрез параметрите се указва как да се обединят тези файлове. Когато програмата не голяма и е поместена в един файл, обикновено параметрите не са необходими (с едно изключение, което ще разгледаме по-късно). Пример:

```
A   SEGMENT
        A1  DB  400h  DUP(?)
        A2  DW  8
A   ENDS
B   SEGMENT
        B1  DW  A2
        B2  DD  A2
B   ENDS
C   SEGMENT
        ASSUME  ES:A,  DS:B,  CS:C
    L:  MOV   AX, A2
        MOV   BX, B2
        ....  ....  ....
C   ENDS
```

Ще използваме този пример за да изясним някои въпроси.

Изравняване по границата на параграф

Асемблер помества в паметта програмните сегменти последователно един след друг, като при това техните начални адреси са кратни на 16 (вижте глава 1, пункт 1.2). Обемът на областта между два сегментни адреса е 16 байта и се нарича *параграф*. Ако за горния пример сегмент A е разположен от абсолютния адрес 10000h, то всички предложения, направени за този сегмент, ще заемат областта от адрес 10000h до адрес 10401h. Първият свободен байт след тази област ще има адреса 10402h. Този адрес не е кратен на 16 тъй като младшата цифра не е 0. По тази причина той не може да бъде начален адрес за следващия сегмент с името B. За начален адрес на следващия сегмент се търси адрес, който е кратен на 16, и това може да бъде само адресът 10410h. Сегмент B заема 6 байта, т.е. това са адресите 10410h, 10411h, 10412h,

10413h, 10414h и 10415h. Първият свободен адрес е 10416h, който не е кратен на 16, ето защо следващият сегмент C може да започне от адрес 10420h, т.е. след 10 пропуснати клетки.

В заключение ще обобщим, че Асемблер сам определя началните адреси на сегментите, като между тях са възможни празни области с обем не по-голям от 15 клетки.

Стойност на сегментното име

На името на всеки програмен сегмент Асемблер присвоява определена стойност. Това е старшата 16 битова част от определения 20 битов начален адрес. В горния пример името A=1000h, името B=1041h, името C=1042h. Срещайки в команда името на сегмента, Асемблер ще го замества с неговата стойност. Например, командата:

```
        MOV   AX, B
```

ще се възприема като командата

```
        MOV   AX, 1041h        ; AX := началния адрес на сегмент B
```

Следва да отбележим, че в Асемблер имената на сегментите се отнасят към *константните* изрази, а не към адресните. Ето защо горната команда MOV записва в регистър AX *число*, а не съдържанието на клетка с този адрес. По тази причина командата *Load Effective Address* (LEA AX, B) ще се възприема като грешка, тъй като в нея не може да се указва константа.

Оператор OFFSET и оператор SEG

При компилиране на предложенията на сегмента, Асемблер съпоставя на имената на променливите и на етикетите, адреси, които представляват отместване относно началния адрес на сегмента, т.е. стойности, които могат да се интерпретират като абсолютни адреси, но в рамките на сегмента. Така за горния пример името A1 ще получи отместване 0, името A2 – отместване 400h, името B1 – отместване 0, името B2 – отместване 2, името L – отместване 0. Именно с тези стойности Асемблер замества съответните имена, когато ги срещне в текста на програмата.

Ще напомним, че имената на променливите и на етикетите Асемблер разглежда като адресни изрази. Възможно е обаче и разглеждане на константа. За целта се използва оператор за явно деклариране на стойността: оператор OFFSET (отместване). Операторът се следва от обявеното име:

```
        OFFSET  A1             ; =0
        OFFSET  A2             ; =400h
        OFFSET  B1             ; =0
```

Операторът е константа, а не адрес:

```
        MOV   AX, A2           ; AX := (A2),     (A2)=8
        MOV   AX, OFFSET  A2   ; AX := 400h  ≡  LEA  AX, A2
```

Името A2 означава адрес, а OFFSET A2 е константа, въпреки че по стойност те съвпадат.

Ще разгледаме пример, където този оператор е полезен. Нека в регистър BX се намира адресът на k-тият елемент от масива A1, т.е. (BX)=A1+k и се изисква в регистър BX да се запише номера на този елемент, т.е. стойността на k. Да се направи това с командата

SUB BX, A1

е невъзможно, защото тя изпълни действието BX := (BX)-(A1), т.е. ще извади стойността на първия елемент от масива A1. Решението постига командата:

SUB BX, OFFSET A1

която ще изпълни действието: BX := (BX)-A1 = A1+k – A1 = k.

Операторът SEG позволява на Асемблер да установи в кой сегмент е описано дадено име на променлива или на етикет. Записва се:

SEG <име>

При изпълнение операторът получава (връща чрез името) константна стойност, равна на началния адрес на сегмента, в който е описано споменатото в него име. Така например:

```
SEG  A1  =   SEG  A2   =  A  =  1000h
SEG  B1  =   SEG  B2   =  B  =  1041h
SEG  L                 =  C  =  1042h
```

Тъй като стойността на оператор SEG е константа, то командата:

MOV BX, SEG B1

записва в регистър BX стойността на началния адрес на сегмент B, т.е.

BX := 1041h, BX := B

Адресни променливи

Асемблер обработва имената на променливите като адресни изрази, когато те са описани в директиви DW или DD. В горния пример, като начална стойност на променливите B1 и B2 е указано името A2,

```
B1  DW  A2
B2  DD  A2
```

т.е. адресът на тази променлива. Въпросът е: кой адрес се има предвид? – абсолютният или относителният (отместването). За отстраняване на тази дилема в Асемблер действа следното правило:

- Ако името на променливата или на етикета (или на адресния израз) се описва от директива DW, стойността, която Асемблер доставя е относителния адрес, т.е. отместването от началния адрес на сегмента ;
- Ако името е указано в директива DD, тогава стойността представлява абсолютния адрес, тъй като се има предвид съответната двойка адресни регистри (сегментен:отместване).

Например:

```
B1  DW  A2        ; еквивалентно е на  B1  DW  offset A2
B2  DD  A2        ; еквивалентно е на  B2  DD  seg A2 : offset A2
```

По такъв начин, въпреки че в директивата DD е указано само име,

по него Асемблер разбира сегмента и отместването в него. Ще напомним, че поради "обратното" подреждане на двете части на двойната дума в паметта, стойностите, които се имат предвид по-горе са разположени така:

в клетка B2:offset, и в следващата клетка B2+2:seg.

В директивата DD като операнд може да бъде указана адресна двойка от вида:

<име на сегмент> **:** <адресен израз>

Например: DD A : B2

В този случай Асемблер заменя името на сегмента с неговия начален адрес (старшата му 16 битова част), а името на променливата с 16 битово отместване, но не от началото на сегмента, в който това име е описано, а от началото на указания сегмент. Тогава в горната директива името A ще се замени със стойността 1000h, а името B2 – със стойността 412h. Това отместване е равно на разликата между абсолютния адрес на B2=10412h и абсолютния адрес на A=1000h. С подобни оператори следва да се внимава, тъй като отместването на една променлива, която се намира в друг сегмент, може да бъде по-голямо от 64 [KB]. Освен това при компилиране на програмата е възможно разместване на сегментите в паметта, т.е. сегмент A може да се окаже поместен след сегмент B, в който е описано името B2. Тогава отместването спрямо A ще бъде отрицателно число, което разбира се е недопустимо.

7.3 Директива ASSUME

Беше казано, че ако обединим няколко предложения в един програмен сегмент, то Асемблер ще ги разположи в един сегмент от ОП. Това означава, че всички адреси от този програмен сегмент могат да бъдат сегментирани относно един начален адрес, за който може да се използва един сегментен регистър. Възниква въпросът: кой да бъде той? Това следва да определи сам програмистът. Ако, както в предидущия пример, за сегмент A е избран регистър ES, а за сегмент B е избран регистър DS, то този избор следва да бъде съобщен на Асемблер. В противен случай ние всеки път, когато указваме променлива, следва да я сегментираме с подходящия сегментен префикс. Например, следва да пишем: (MOV AX, ES:A2).

Но да изписваме във всяка команда префикс е доста неудобно. Желателно е Асемблер да прави това вместо нас. Тогава вместо горната команда ние ще пишем (MOV AX, A2). За целта следва да декларираме своето разпределение.

Разпределението на сегментните регистри върху програмните сегменти се съобщава на Асемблер чрез директивата ASSUME (в смисъл на *приписвам на*), която има следния синтаксис:

ASSUME <адресна двойка> {, <адресна двойка>}

или

ASSUME NOTHING

където <адресна двойка> представлява конструкцията:

<сегментен регистър> **:** <име на сегмент> или
<сегментен регистър> **:** NOTHING

В нашия пример беше указана директивата:

ASSUME ES:A, DS:B, CS:C

с която съобщаваме на Асемблер, че за сегментиране на адреси от програмния сегмент А е избран регистър ES, за сегментиране на адреси от програмния сегмент В е избран регистър DS, а за сегментиране на адреси от програмния сегмент С е избран регистър CS. За Асемблер това ще означава, че всички имена от сегмент А следва да транслира с префикс ES, имената от сегмент В – с префикс DS, а за имената от сегмент С – с префикс CS. За нас това ще означава, че няма да записваме явно съответните префикси. Така командата

MOV AX, A2

ще се възприема от Асемблер като команда

MOV AX, ES:A2 ,

тъй като името А2 е описано в сегмент А, а на него, съгласно директивата ASSUME, е приписан (определен) регистър ES. Съответно командата

MOV AX, B2

ще се приема от Асемблер като съкращение на командата

MOV AX, DS:B2

защото променливата В2 е описана в сегмент В, на който директивата ASSUME приписва (определя) регистър DS.

Възможни са случаи (ще бъдат разгледани по-късно), в които Асемблер не може да определи правилно префикса.

Особености на директивата ASSUME

Директивата ASSUME е само информативна и не е в състояние да зареди определеното съдържание в сегментните регистри. Чрез тази директива програмистът само обещава, че такова зареждане ще бъде изпълнено.

Тъй като информацията в директива ASSUME се използва едва при компилиране на командите, то тя, въпреки че може да се постави в произволно място, следва да бъде срещната преди компилиране на първата команда, в която се адресира променлива. Ето защо има смисъл директивата да се поставя в началото на командния сегмент.

При това в директивата задължително трябва да бъде указано съответствието между регистър CS и дадения кодов сегмент. В противен случай, при появата на първия етикет, Асемблер ще генерира грешка.

Ако в директива ASSUME са указани няколко двойки с един и същи

сегментен регистър, то всяка следваща отменя предидущата. Така остава валидна последната. Това е така, защото един сегментен регистър може да бъде приписан само на един сегмент. Например, при директива:

```
        ASSUME   ES:A,  ES:B
```

Асемблер ще назначава префикса ES: само за имена от сегмент B.

В същото време на един и същи сегмент могат да се припишат различни регистри:

```
        ASSUME   ES:A,  DS:A
```

т.е. сегментът A ще бъде сочен от два регистъра. В този случай Асемблер е свободен в избора си. Все пак той предпочита онзи префикс, който в транслиращата се команда се има предвид по подразбиране.

Ако в директивата ASSUME в качеството на втори елемент в адресна двойка е употребена служебната дума NOTHING, това означава, че от дадения момент нататък сегментния регистър не указва на никой сегмент, че Асемблер не бива да използва повече този сегментен регистър и че програмистът поема задължението сам да указва този регистър, когато е необходимо. Ако е необходимо да бъдат отменени по-рано установени съответствия за всички сегментни регистри, то се използва директивата: ASSUME NOTHING

Директивата ASSUME може да бъде записвана неограничен брой пъти. Няколко последователни директиви могат да се обединят в една. Ако обаче между две директиви са записани няколко команди, то обединението на директивите е не винаги възможно. Например, в следващия текст (ако името A2 е описано в сегмент A):

```
    ASSUME   ES:A
        MOV   CX, A2
    ASSUME   ES:B,  SS:A
        INC     A2
```

името A2 в командата MOV ще бъде транслирано с префикс ES:, но командата INC ще бъде транслирано с префикс SS:, тъй като преди това на сегмент A е приписан нов сегментен регистър – SS.

Това обаче няма да се случи в следващия текст:

```
    ASSUME  ES:A,  ES:B,  SS:A
        MOV   CX, A2
        INC     A2
```

защото при последното приписване, за указател на сегмент A е определен регистър SS, а то отменя първоначалното с указател ES. Така и в двете команди променливата A2 ще бъде сегментирана чрез регистър SS.

Избор на сегментен регистър при транслиране на командите

Ще изложим правилата, по които Асемблер сегментира операнди при транслиране на команди, в които префиксът не е указан явно.

Ако в записа на операнда няма име, по което Асемблер би могъл да определи в кой сегмент е разпределен операндът, тогава Асемблер избира този сегментен регистър, който за тази команда се подразбира. Например, в командата MOV AX, [BX], косвено адресираща втория операнд, ще бъде избран префикс DS:, а в командата MOV AX, [BP] – ще бъде избран префикс SS:.

Ако като операнд е указано име на променлива или етикет, тогава Асемблер проверява дали това име е вече описано в програмата. Ако не, или е преход напред, тогава Асемблер предполага, че това име следва да се сегментира относно регистър, който тази команда използва по подразбиране. Например, ако X е етикет, който сочи преход напред, то в команда INC X или например в команда INC X[SI] ще се подразбира префикс DS:, а в команда INC X[BP+1] – префикс SS:. Ако обаче в последствие се установи, че това предположение е невярно, тогава ще бъде генерирана грешка, тъй като със закъснение Асемблер не е в състояние да постави пропуснатия префикс – за него вече няма място.

Ако в транслираната команда името на операнда е вече описано, тогава Асемблер по директивата ASSUME определя сегментните регистри, приписани на съответния сегмент, в който е описано даденото име. Ако такива регистри няма, генерира се грешка. В противен случай от тези регистри Асемблер избира онзи, който командата използва по подразбиране, а ако такъв няма, избира произволен.

След като се установи с какъв префикс ще транслира операнда, Асемблер проверява дали той не съвпада с подразбирания от командата префикс. Ако той не съвпада, Асемблер генерира машинната команда с вече избрания префикс, а ако префиксите съвпадат генерира машинната команда, все едно е била без префикс.

Както навярно читателят сам разбира, изложените превила за определяне на префикс, когато той не е явно указан, са достатъчно сложни и трудни за запомняне. За щастие в реалните програми програмистите се придържат към коректен стил, при който указваното име е вече описано. Случаите, в които се налага явно указване на префикса, са достатъчно малко на брой и са пояснени по-долу.

<u>Първият</u> случай е свързан с косвените обръщения. Например, при транслиране на командата ADD DX,[BX] Асемблер избира префикс DS:, който се използва от тази команда по подразбиране, т.е. приема, че в регистър BX се намира адрес, сегментиран в данновия сегмент. Ако обаче ние знаем, че в BX се намира адрес, който иска обръщение в друг сегмент, например в кодовия, тогава <u>сме длъжни</u> явно да укажем правилния префикс, т.е. да изпишем командата така: ADD DX, CS:[BX].

<u>Вторият</u> случай е свързан с обръщенията напред, т.е. когато в командата е указано име, което ще бъде описано по-късно в текста. В тази ситуация Асемблер използва сегментен регистър, който командата

използва по подразбиране. Ако това не ни устройва и знаем, че не този е регистъра, тогава сме длъжни да го укажем явно. Например:

```
CODE    SEGMENT
            ASSUME  CS:CODE,  ES:DATA1
            MOV   AX,  ES:X
      ....  ....  ....
CODE    ENDS
DATA1   SEGMENT
            X  DW  23
      ....  ....  ....
DATA1   ENDS
```

В горния текст, в команда MOV е необходимо да се запише префиксът ES:, тъй като без него Асемблер би транслирал тази команда без префикс, а в последствие, срещайки описанието на името X в сегмент DATA1, на който в директивата ASSUME е приписан регистър ES, ще генерира грешка.

В тази ситуация обаче е удобно да се постъпи по друг начин – за да не създаваме условия за грешки при обръщения напред и за да записваме префикси по-рядко, следва да обявяваме данновия сегмент преди командния, при което просто няма да има обръщения напред. С други думи, препоръчва се кодовият сегмент да се поставя в края на програмата.

Третият случай е на лице, когато в директивата ASSUME на даден сегмент не е приписан сегментен регистър. Нямайки нужната информация, Асемблер не е в състояние сам да определи с какъв префикс следва да транслира имената от този сегмент (приемайки че авторът на програмата е поел задължението сам да указва такъв префикс). Подобна ситуация е по-добре да бъде избягвана, тъй като не си струва труда за подобни задължения.

И последният случай когато се налага да се указва явно префикс, е свързан с задаване на явни адреси. Така например, ако е необходимо в регистър AX да се запише съдържанието на клетка с адрес 5, то да се направи това с командата MOV AX, 5 е невъзможно. Причините за това са две. Едната е формална: в Асемблер явно указано число винаги се възприема като константа (непосредствен операнд), а не като адрес. Така че тази команда ще изпълни действието AX:=5. Втората причина е по-съществена. Даже ако числото 5 се приеме за адрес, то възникват въпросите: в кой сегмент на паметта е този адрес?, по кой регистър трябва да се сегментира този адрес?. В Асемблер проблемът с пряката адресация се решава по следния начин: указването на префикс пред преки (явни) адреси е задължително правило. Ако е известно, че адресът 5 е в данновия сегмент, тогава командата следва да се запише така: (MOV AX, DS:5). С така указаният префикс се “убиват два заека”:

първо, наличието на префикс отменя интерпретацията на числото 5 като непосредствен операнд и налага възприемането му като адрес и второ, определя се сегментиращия регистър.

Съществува още една особеност при транслиране на операндите, когато са записани с явен префикс, която ще разгледаме върху следващия пример:

```
DATA   SEGMENT
         X  DW  ?
   ....  ....  ....
DATA   ENDS
CODE   SEGMENT
         ASSUME  CS:CODE,  ES:DATA
          Y  DW  ?
      L:  MOV  ES:Y, 0
          ....  ....  ....
```

От една страна, в командата MOV трябва да бъде използван префиксът ES:, но от друга страна името Y е описано в сегмент CODE, чието начало се сочи от регистър CS, и по тази причина следва да се използва префикс CS:. Това определено е дилема – кой е правилният префикс? В такива случаи Асемблер транслира командата с явно указания префикс (в примера ES:) с отместване Y, но не отместването в “родния” сегмент CODE, а отместването, което се получава спрямо началото, сочено от явния префикс, т.е. спрямо началото на DATA.

Препоръчва се подобен род записи да се избягват, тъй като разстоянията между сегментите могат да доведат до неправилно адресиране. Правилният стил на програмиране предполага цялостно възлагане на Асемблер задачата за приписване на префиксите или цялостното й поемане от програмиста. Смесването на тези две линии на поведение е признак за лош стил.

7.4 Начално зареждане на сегментните регистри

И така, с директивата ASSUME ние съобщаваме, че имената от обявените програмни сегменти са сегментирани спрямо назначените сегментни регистри и така задължаваме Асемблер да транслира текста в машинни команди със съответните префикси. Това обаче още не означава, че програмата ще работи правилно. Нека имаме командата (MOV AX, A2), която е формирана във вида (MOV AX, ES:A2). Ако обаче регистър ES не сочи началото на сегмент A, то тази команда няма да се изпълни правилно. Необходимо е в сегментните регистри да се намират правилните, актуалните стойности. Въпросът е актуален, защото в началото, когато стартира програмата съдържанието на сегментните регистри общо казано е неизвестно. Ето защо изпълнението на новата програма следва да започва с команди, които зареждат сегментните регистри с актуални за програмата стойности. Напомняме, че директивата ASSUME не извършва такова зареждане.

Нека желаем в регистър DS да поставим началния адрес на сегмент B. Ще напомним, че Асемблер присвоява на името на сегмента старшите 16 бита на началния адрес, т.е. DS:=B и тъкмо това присвояване е необходимо. Тъй като Асемблер разглежда името на сегмента като константа, това присвояване следва да се изпълни чрез допълнителен помощен регистър, например така:

```
MOV  AX, B
MOV  DS, AX
```

Аналогично се зарежда и регистър ES.

Що се отнася до регистър CS, той не бива да се зарежда. Следва да знаем, че със стартирането на нашата програма този регистър е вече зареден с базовия (сегментния) адрес на кодовия сегмент. Това зареждане изпълнява операционната система, с което предава управлението на потребителската програма.

И остана да се поясни зареждането на регистър SS. Неговото зареждане можем да изпълним в нашата програма, така както зареждаме DS и ES, но можем да поръчаме това на операционната система. Ако искаме операционната система да избере стековата област и сама да зареди регистър SS, то трябва да укажем параметър STACK в директивата SEGMENT:

```
S  SEGMENT  STACK
```

Асемблер ще присвои стойност на името S, а зареждането на тази стойност в регистър SS, ще бъде изпълнено от ОС до стартиране на програмата. Ясно е, че от двата представени варианта, за предпочитане е вторият.

7.5 Структура на програмата. Директива INCLUDE

Вече имаме необходимата информация за да поясним как изглежда една напълно завършена програма на Асемблер. Някаква фиксирана структура асемблерската програма няма, но не големите програми, имащи три сегмента – команден, даннов и стеков, типична е следната структура:

```
STACK  SEGMENT  STACK                ; сегмент на стека
   DB  128  DUP(?)
STACK  ENDS
DATA  SEGMENT                        ; даннов сегмент
   <описание на променливите и др. п.>
DATA  ENDS
CODE  SEGMENT                        ; сегмент на командите
   ASSUME  CS:CODE,  DS:DATA,  SS:STACK
START:  MOV  AX, DATA
        MOV  DS, AX                  ; зареждане на DS
      <останалите команди на програмата>
CODE  ENDS
      END  START                     ; край на програмата
```

Взаимното разположение на сегментите на програмата може да бъде всякакво, но както вече беше отбелязано, за да се съкратят обръщенията напред и при това да се избегнат проблемите с префиксите, е необходимо командният сегмент да се разположи в края на програмата.

Стекът е със същото име както името на параметъра STACK, което не е забранено. Дори програмата да не използва стек, описанието му в програмата е необходимо, тъй като по време на нейното изпълнение са възможни прекъсвания, които се обслужват от стека. Размерът на стека, задаван в програмата, е 128 байта.

В края на програмата задължително се записва директивата END. Освен това в нея се посочва входната точка на програмата – етикетът START.

Директива INCLUDE

В една програма не могат да липсват операции за въвеждане на входни данни или за извеждане на готовите резултати. Тук се има предвид, че описанието на тези операции се намира в отделен файл, който се нарича IO.ASM. За да могат да се изпълняват входно-изходните операции, този файл следва да бъде присъединен към програмата. Това изисква първият ред от текста на програмата да бъде следният:

INCLUDE IO.ASM

Както се вижда, на този ред е обявена директивата INCLUDE (включи), която има следния общ синтаксис:

INCLUDE <име на файл>

Срещайки тази директива, Асемблер включва в програмата текста от указания файл.

Директивата INCLUDE може да бъде записана неограничен брой пъти, при това във всяко място на програмата. В нея може да бъде указван всеки файл, а името му може да бъде изписвано по правилата на операционната система, например:

INCLUDE A:MACROS.TXT

Директивата INCLUDE е полезна, когато в различни програми се използва един и същи текст. В нашият вместо директивата ще се присъедини текстът на файла IO.ASM, с което ние ще получим възможността да използваме тези операции.

В заключение ще приведем текста на кратка, но пълна програма на Асемблер. Задачата на програмата е да въведе 50 символа и после да ги изведе в обратен ред. За да постигнем това в програмата е въведен масив от 50 еднобайтови елементи, в който в обратен ред ще записваме въведените символи, а в края ще го изведем във вид на ред.

```
        INCLUDE  IO.ASM                  ; за В/И операции
     S  SEGMENT  STACK                   ; сегмент на стека
        DB  128  DUP(?)                  ; обем на стека
     S  ENDS

     D  SEGMENT                          ; сегмент за данни
     N  EQU  50                          ; брой на символите за вход
     X  DB  N  DUP(?), "$"               ; за място в паметта + $
     D  ENDS

     C  SEGMENT                          ; сегмент за програмата
        ASSUME  SS:S, DS:D,  CS:C
BEG:    MOV  AX, D
        MOV  DS, AX
        OUTCH  ">"                       ; подсказка за въвеждане
        MOV  CX, N                       ; брояч на цикъла
        MOV  SI, N-1                     ; индекс на елемент
  IN:   INCH  X[SI]                      ; X[SI] := входен символ
        DEC  SI
        LOOP  IN
        LEA  DX, X                       ; начален адрес на X в DX
        OUTSTR                           ; извеждане на X като ред
        FINISH
   C:   ENDS
        END  BEG                         ; входна точка BEG
```

ГЛАВА 8
СТЕК

Тук ще се спрем на командите за работа със стека, както и на някои похвати за неговото използване.

8.1 Стек и сегмент на стека

Стекът е запомнящо устройство с дисциплина LIFO. Реализира се в адресното пространство на оперативната памет. Изискванията за него са две: обемът му не може да надхвърля 64[KB] и началният му адрес трябва да е кратен на 16. Началният адрес (сегментният адрес) се съхранява постоянно в регистър SS, а за адресиране на клетките в стека се използва стековият указател – регистър SP. Стекът е от тип "винаги готов за четене". Абсолютният адрес на топ клетката в стека се адресира с двойката регистри SS:SP. Стекът се създава като част на програмата от програмиста, например, чрез описанието:

```
S  SEGMENT  STACK
     DB   k  DUP(?)
S  ENDS
```

Тъй като стекът е безадресно запомнящо устройство (от гледна точка на процесора), имена на неговите клетки не се определят. Те се адресират в резултат на дисциплината LIFO чрез стековия указател SP. Обемът на стека може да се декларира и с директивите DW и DD.

За правилна работа на програмата адресните регистри SS и SP трябва да бъдат първоначално инициализирани. Най-добре е това да се възложи на операционната система чрез изписване на параметъра STACK. По този начин се гарантира, че със стартиране на програмата съдържанието на регистър SP ще бъде нула, а на регистър SS – определеният от ОС начален (сегментен) адрес.

Горната декларация определя и обема на стека. Както беше казано в глава 7, декларирането на стеков сегмент в програмата е задължително. Обемът на стека (числото k) като минимум, дори и когато програмата не го използва, се предлага да бъде 128 байта. Ако тя го използва, то към тези 128 клетки следва да се добави броя на допълнителните.

8.2 Стекови команди

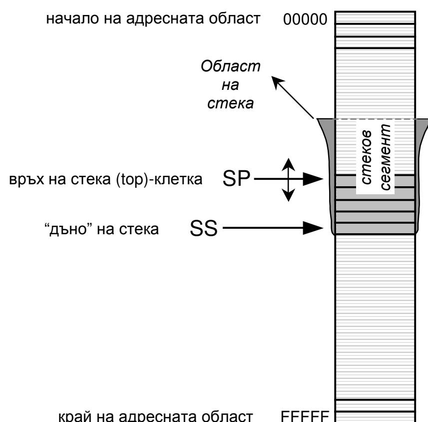
За работа със стека има няколко команди.

Запис на дума в стека: **PUSH op**

Допустимите за операнда типове са: r16, m16, sr.

Действието на командата се състои от два етапа: първоначално се модифицира стековият указател SP:=(SP)-2, като неговото съдържание се намалява с 2. След това в така указаната клетка се записва операнда (съдържание на регистър или на клетка от ОП). Командата не променя флаговете. При тази модификация на стековия указател функ-

ционирането на това запомнящо устройство може да се илюстрира със следващата рисунка.



Вижда се, че при запис стековия указател SP се движи в посока към малките адреси, защото се модифицира чрез изваждане на 2, а при четене – в посока на големите адреси, защото се модифицира чрез събиране с 2. Началният адрес, който определяме като "дъно" на стека, остава постоянен в регистър SS. В стека може да бъде записано и съдържанието на самия стеков указател: (PUSH SP). Но при това в стека се записва предварително модифицираното съдържание на регистър SP (SP-2).

Като операнд на команда PUSH може да бъде указан всеки регистър. Не трябва обаче да се използва непосредствен операнд. Например, командата (PUSH 5) ще генерира грешка. Подобен запис се програмира с допълнителен регистър:

```
MOV   AX, 5
PUSH  AX
```

Обръщаме внимание на това, че формàтът на команда PUSH може да бъде само дума (2 байта). Другите дължини (байт или двойна дума) следва изкуствено да се привеждат към този формàт.

Четене на дума от стека: **POP op**

Допустимите за операнда типове са: r16, m16, sr (но без CS).

Действието на командата се състои от два етапа: първоначално от указаната клетка се чете съдържанието на топ клетката и се присвоява на указания операнд. След това се модифицира стековият указател, като неговото съдържание се увеличава с 2 (SP+2).

Командата променя флаговете.

Запис и четене на флаговия регистър

Съдържанието на флаговия регистър може да се съхрани в стека, от където в последствие да се възстанови. За целта се използват две ко-

манди: PUSHF   ; запис на флаговете
            POPF   ; четене на флаговете

Аналогично на предните две команди, команда PUSH не променя флаговете, а команда POP ги променя, като на мястото на текущите възстановява преди това съхранените в стека. Командите са удобни, когато се налага анализ на даден флаг, когато за него няма друга подходяща команда. Ако конкретно е необходимо, например, да се провери състоянието на флаг TF (за трасировка, вижте глава 1), който се намира в бит №8 на регистър Flags, можем да направим следното:

```
PUSHF               ; запис в стека на (Flags)
PUSHF               ; втори запис в стека на (Flags)
POP   AX            ; извличане от стека в AX
MOV   CL, 8         ; брой на изместванията
SHR   AX, CL        ; изместване
AND   AX, 1b        ; маскиране на младшия бит b0
POPF                ; възстановяване на флаговете
```

След изпълнение на описаните действия съдържанието на стека е същото, в каквото беше заварен.

8.3 Похвати за работа със стека

Първото правило, на което изрично следва се спрем, гласи, че каквото запишем в стека следва да го извлечем обратно. Не спазването на това правило води до пълно объркване на информацията и до загуба на логиката на алгоритъма на програмата.

Съхраняване на съдържанието на регистри

Поради малкия брой регистри в процесора, често се налага един и същи регистър да бъде използван за различни задачи. Това няма да бъде възможно без освобождаването от текущото му съдържание. За целта обикновено се използва стекът, където временно се съхранява неговото съдържание, например:

```
PUSH  CX        ; съхранение на CX в стека
....  ....  ....    ; зареждане и използване на CX
POP   CX        ; възстановяване на CX
```

Прехвърляне на данни с помощта на стека

Стекът често се използва като междинна памет при размяна на местата на данни в паметта, когато няма свободни регистри. Например прехвърляне на клетка Y в клетка X може да се реализира така:

```
PUSH  Y         ; изпращане на съдържанието на Y в стека
POP   X         ; X приема изхвърленото Y от стека
```

Проверка за препълване на стека

Следва да се помни, че обемът на стека е ограничен в рамките на неговата декларация (числото k), така че неконтролируемата употреба на командите за работа със стека могат да доведат до грешки, тъй като те не притежават възможност за проверка. На читателят следва да е

известно, че при пълен стек операция запис е невъзможна, а при празен стек операция четене не е възможна. Състоянието на стека програмистът може да установи по текущото съдържание на стековия указател: ако то е нула, това значи, че стекът е празен, а ако съдържанието е равно на обявеното в декларацията число k – той е пълен.

Изчистване и възстановяване на стека

Понякога се налага да се почисти стека, като се изхвърлят записаните в него елементи като ненужни. Това може да се постигне чрез няколко последователни команди POP, но това е доста дълго решение. По-елегантното решение се състои в директна модификация на съдържанието на стековия указател, който се възприема като регистър с общо предназначение. Ако желаем да “изхвърлим” N на брой думи от стека, увеличаваме съдържанието на регистър SP с числото 2N, което може да постигнем с командата:

```
ADD  SP, 2*N
```

Възможно е и следното решение: запомняме съдържанието на регистър SP например в регистър AX, след което работим свободно със стека, като записваме и четем каквото се налага. Връщането на стека в изходното състояние се постига с възстановяване на съдържанието на стековия указател:

```
MOV   AX, SP            ; AX := (SP)
....  ....  ....  ....  ; разнообразна работа със стека
MOV   SP, AX            ; SP := (AX)  -  възстановяване
```

Цялостно изчистване на стека може да се постигне с нулиране на съдържанието на регистър SP.

Достъп във вътрешността на стека. Регистър BP

Командите PUSH и POP имат достъп само до топ клетката на стека. Понякога обаче се изисква достъп до клетки във вътрешността на стека. Такъв достъп би могъл да ни осигури друг стеков указател, различен от SP. Ако стековият указател SP сочи топ клетката, в която е записана последната дума, но на нас е необходима думата от преди два записа, не бихме могли да я прочетем чрез SP. За целта определяме отместването на желаната клетка относно топ клетката и адресираме чрез друг регистър. Отместването на две думи във вътрешността на стека е равно на 4. Ще използваме регистър BP по следния начин:

```
MOV  BP, SP         ; BP := (SP)
MOV  AX, [BP+4]     ; косвено адресиране, AX := (SS:[BP+4])
```

Защо избираме регистър BP? Първо защото регистър SP не е регистър-модификатор и изразът [SP+4] е грешен. И второ, в по подразбиране регистърът-модификатор BP се сегментира спрямо SS, което разбира се направено нарочно, именно за такива цели.

Г Л А В А 9

П Р О Ц Е Д У Р И

Тук ще се спрем на създаването и на използването на подпрограми.

9.1 Дълги преходи

В общия случай в програмата могат да бъдат описани много командни сегменти.

Например:

```
C1   SEGMENT
          ASSUME  CS:C1, …
START:    MOV  AX, 0
          ....  ....  ....  ....
          JMP  FAR  PTR  L                 ; go to L
          ....  ....  ....  ....
C1   ENDS

C2   SEGMENT
          ASSUME  CS:C2
     L :  INC  BX
          ....  ....  ....  ....
C2   ENDS
```

Ще припомним, че в началото на всеки команден сегмент трябва да бъде употребена директивата ASSUME, в която, освен всичко останало, на сегментния регистър CS се приписва името на сегмента. Именно от тази информация Асемблер разбира, че този сегмент е команден. В противен случай, срещайки първия етикет, той ще генерира грешка.

Ще напомним, че адресите на командите се образуват от двойката регистри CS и IP. Изменението на съдържанието на всеки един от тези регистри означава преход.

Ако се променя съдържанието само на програмния брояч, това означава преход в рамките на текущия сегмент. Ако програмата е голяма и е структурирана в няколко кодови сегмента, то напълно възможно е наличието на преходи от един сегмент в друг. Например, както в горния текст, от сегмент C1 преход към етикет L, който маркира команда от друг сегмент (C2). Такива преходи се наричат *между сегментни* или дълги. Преходите променят съдържанието както на CS така и на IP. За горния текст, за да се реализира преход от сегмент C1 към точка L в сегмент C2 са необходими измененията: CS:=C2, и IP:=Offset L.

В системата машинни команди на разглеждания процесор са реализирани машинни команди за безусловен дълъг преход. Преходите са само безусловни, пряко или косвено адресирани. В Асемблер тези команди имат същата мнемоника (JMP), но използват друг тип операнди. Командите за дълъг преход не променят флаговете.

Пряко адресиран дълъг преход: **JMP FAR PTR <етикет>**

Думата FAR (далече) е служебна и показва на Асемблер, че етикетът, който сочи командата, се намира в друг команден сегмент. Действието на командата се заключава в зареждане на кодовия сегментен регистър CS със сегментния адрес на сегмента, съдържащ етикета, и в зареждане на програмния брояч IP с отместването на маркираната команда в този сегмент:

CS := seg <етикет> , IP := offset <етикет>

За нашия пример това означава:

CS := C2 <L> , IP := offset <L>

Косвено адресиран дълъг преход: **JMP m32**

В такава команда се посочва адрес на двойна дума, в която трябва да се намира абсолютният адрес на прехода във вид на адресна двойка: (seg:offset), записан в обратен ред:

CS := [m32+2] , IP := [m32]

Пример:

```
X DD L       ; X : offset L,  X+2 : seg L ,   записва в X определения
 ....  ....  ....                                               адрес
  JMP  X     ; go to L ,   зарежда X, но отива на L (косвен преход)
 ....  ....  ....
```

При записване на команда с косвен преход следва да се внимава. Ако в командата е указано име X, което е описано до тази команда (както в горния пример), тогава, срещайки го, Асемблер вече ще знае, че това име има тип DD, и по тази причина правилно ще разбере, че преходът е косвен и дълъг. Но ако името е описано по-късно, срещайки командата, Асемблер няма да знае вида на прехода – дали близък преход по етикет, дали близък косвен или далечен косвен преход. В тази ситуация, както е изложено в глава 4, Асемблер ще предположи, че указаното име е етикет в текущия сегмент и ще формира машинна команда за близък преход. В последствие, когато бъде установено, че в текущия сегмент етикет с такова име няма, Асемблер ще генерира грешка. Избягването на такава грешка при обръщение напред при далечен косвен преход изисква явно да бъде описан типът на името като двойна дума. За целта следва да се използва оператор PTR:

JMP DWORD PTR X

Както вече беше отбелязано, подобен род уточнения за обръщенията напред се налага да бъдат правени и за късите преходи.

Ако съберем всичко, което беше казано до момента за обръщенията напред в команди за безусловен преход, то ще се изяви следното правило: ако X е обръщение напред, то командата за безусловен преход следва да бъде записвана така:

JMP X	Близък пряк дълъг
JMP SHORT X	Близък пряк къс

JMP FAR PTR X	Далечен пряк
JMP WORD PTR X	Близък косвен
JMP DWORD PTR X	Далечен косвен

Подобен род уточнения трябва да се правят и тогава, когато в команда JMP е указано косвено обръщение, например JMP [BX], тъй като по подразбиране Асемблер приема този преход за близък и косвен.

Когато X е обръщение назад, то вида на прехода трябва да се уточни задължително само в един случай – при дълъг пряк преход, защото Асемблер, дори и да знае, че X е етикет от друг команден сегмент, няма да разглежда командата (JMP X) като далечен преход. Работата е в това, че срещайки етикет, Асемблер автоматично му присвоява тип NEAR (близък), който по-нататък го ръководи. В аванс ще отбележим, че в Асемблер има директива LABEL, с помощта на която за етикета може да бъде определен тип FAR. Да се промени този тип (в рамките на една команда), е възможно само с оператор PTR.

Имената NEAR и FAR са имена на стандартни константи, съответно със стойности -1 (0FFFFh) и -2 (0FFFEh). Именно тези стойности издава оператор TYPE, ако операндът му не е име на променлива, а етикет (или име на подпрограма): например: TYPE L = NEAR.

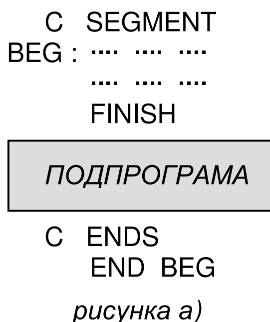
9.2 Подпрограми

Читателят следва да е запознат с понятието подпрограма и средствата за създаване и работа с подпрограми (вижте [1]). Ще припомним, че подпрограмната техника е основен похват в структурното програмиране и се прилага за реализация на не регулярни повторения. Подпрограмите са програмни инструменти, които се използват когато се наложи. В останалото време те са бездейни. Често подпрограмите се наричат процедури или функции, понятия, които са наложени в езиците от високо ниво. На машинно ниво обаче те нямат смисъл, тъй като тази техника се осигурява апаратно. Реализацията на подпрограмната техника на това ниво е значително по-обременителна за програмиста, отколкото на високото ниво. Това е така, защото Асемблер не предоставя автоматизираните средства за програмиране, с помощта на които се решават задачите (проблемите) на този вид програмиране.

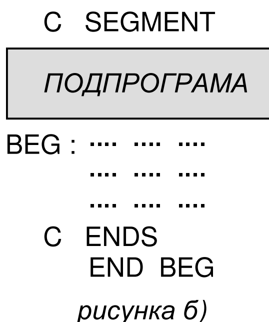
9.2.1 Къде се разполагат подпрограмите?

Първият проблем на програмиста се изразява с този въпрос. Отговорът е: в общия случай – където е удобно. Тук обаче следва да се помни, че намирайки се в избраната област, една подпрограма не работи дотогава, докато към нея не се обърнат или не я повикат. Ето защо тя следва да се разполага така, че да се избегне случайно и ненужно обръщение към нея. Ако програмата използва няколко подпрограми, те обикновено се разполагат последователно, една след

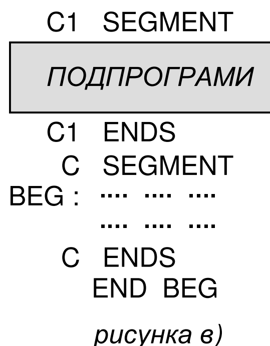
друга. В асемблерския текст на програмата, текстовете на подпрограмите обикновено се разполагат в края на кодовия сегмент, след командата FINISH (вижте по-долу рисунка а), или в началото на този сегмент, преди първата команда, с чието изпълнение стартира програмата (вижте рисунка б). Когато програмите са големи, а подпрограмите са много, последните могат да бъдат разположени в отделен сегмент (рисунка в).



рисунка а)



рисунка б)



рисунка в)

9.2.1 Как се оформят подпрограмите?

Това е вторият проблем. В общия случай групата команди, принадлежащи на подпрограмата, не е необходимо да бъдат отделяни по някакъв начин в текста на програмата. В Асемблер обаче е приет определен начин за оформяне на подпрограмите, във вид на процедура:

<име на процедурата> PROC <параметър>
<тяло на процедурата>
<име на процедурата> ENDP

Както се вижда, всичко започва с директивата PROC и завършва с директивата ENDP. И двете директиви са обявени с едно и също име, което програмистът избира за подпрограмата. Името на процедурата се възприема като етикет, който сочи първата команда в подпрограмата. Този етикет (това име) се използва за обръщение към подпрограмата.

Директивата PROC има параметър. Това е или NEAR (близък), или FAR (далечен). Параметърът може да отсъства. Тогава по подразбиране той се приема NEAR (близък). Така за процедурата може да се говори, че тя е "*близка*" или е "*отдалечена*". Към близка процедура се извършват обръщения само от текущия команден сегмент, но не и от други командни сегменти. Към далечна процедура са възможни обръщения от всеки команден сегмент.

Специално ще отбележим, че имената на променливите и на етикетите, описани в процедурите не са локални, ето защо трябва да са уникални и да не съвпадат с други имена, използвани в програмата. В Асемблер не е позволено вложено описание на процедури, т.е. не се допуска да се описва процедура в рамките на друга процедура.

9.2.3 Обръщение към процедура и връщане от процедура

Обръщането към процедура често се нарича извикване. Тук е необходимо да се изяснят следните въпроси: първоначално възниква въпросът, как да заставим процедурата да работи и как след това, когато тя изпълни своята функция, да се възстанови работата на програмата, която я е извикала. Предполагаме, че читателят има отговорът на тези въпроси, тъй като той се е срещал с тях в [1]. Става дума за командите за работа с подпрограми, които по същество са специализирани машинни команди за безусловен преход. Безусловен е преходът към подпрограмата и такъв е той при връщане от подпрограма. По време на изпълнение на подпрограмата, казваме че преходната (викащата) програма е прекъсната. Такова прекъсване (при преход към подпрограма) се определя като програмно прекъсване.

Въпросът с преходите всъщност е въпрос за адресите. Извикването изисква началния адрес на подпрограмата, който се зарежда в програмния брояч от командата за преход към подпрограма. Преди това обаче тази команда осигурява условията за връщане. Връщането от подпрограма се осигурява чрез запис на съдържанието на програмния брояч в стека, а се реализира от друга команда – команда за връщане от подпрограма, който възстановява съдържанието на програмния брояч, като изважда от стека. Целият този механизъм се нарича подпрограмна техника и се реализира в цифровите процесори обикновено чрез програмния стек. Тук ние нямаме за цел да го изясняваме подробно, тъй като приемаме, че той е детайлно известен на читателя.

В Асемблер работата с процедури (подпрограми) се осъществява с две команди:

```
CALL   <име на процедура>       ; обръщение (извикване)
RET                             ; връщане
```

Действието на командите съответства на горе описаното – команда CALL записва съдържанието на програмния брояч в стека и след това зарежда в него адреса за преход, който е стойност на името на процедурата. Команда RET се определя като безадресна, тъй като според нейния механизъм, тя взема адреса за преход от стека.

Ще разгледаме следния пример: при отработване на програмата си, ние поставяме в нея в различни места команди, извършващи извеждане на екрана на текущи стойности. Тези стойности са контролни и по тях съдим за правилността на хода на програмата. За да не записваме многократно групата команди, извършващи споменатото извеждане, я оформяме като процедура.

```
; главна (викаща) програма        ; процедура
...  ...  ...                     PR  PROC
CALL  PR                              OUTINT  X
…   …   …                             OUTINT  Y, 8
```

```
    CALL  PR                             NEWLINE
    …  …  …                              RET
    CALL  PR                        PR   ENDS
    …  …  …
```

В текста на главната програма се виждат многократни, повтарящи се (нерегулярно) обръщения към процедура PR, действията на която се изпълняват многократно при всяко повикване, но текстът й е записан само веднъж.

Ще напомним, че адресът за преход в този процесор се формира от регистровата двойка CS:IP. Ако описанието на процедурата се намира в същия команден сегмент, в който се намира и викащата програма, тогава преходът към нея и връщането са къси преходи, т.е. преходи към адреси, които имат една и съща база CS. Ако обаче процедурата се намира в друг сегмент, тогава преходът към нея и връщането са дълги преходи, т.е. с промяна на съдържанието и на двата регистъра в адресната двойка.

Тези възможности се отчитат чрез два варианта на командите CALL и RET. Нека чрез името АВ означим адреса за връщане (АВ). Ето какви са действията на команда CALL при къс и при дълъг преход към процедура с името PP:

Близък преход: AB → стек, IP := offset PP
Дълъг преход: (CS) → стек, AB → стек, CS := seg PP, IP := offset PP

При връщане от процедура, действията на команда RET са:

Връщане от близо: стек → IP
Връщане от далече: стек → IP , стек → CS

Ясно е, че командите за преход и връщане действат съгласувано. Различаването на двата вида преходи Асемблер осъществява по описанието на процедурата. Ако то съдържа само име, преходите са къси. Ако името е обявено с оператор PTR – дълги. Така, ако процедурата PP е близка, тя се извиква с командата:

CALL PP ,

а ако процедурата е далечна, тя се извиква с командата:

CALL FAR PTR PP

9.2.4 Други варианти на командата CALL

По-горе, като операнд на командата CALL, беше разглеждано името на процедурата. Възможни са обаче и други варианти за операнд, подобни на командата JMP (с изключение на оператор SHORT).

Пример:

```
NA   DW  P
FA   DD  Q
  …   …   …
```

```
        P   PROC
        .... .... .... ....
P1:     .... .... .... ....
        .... .... .... ....
        P   ENDP

        Q   PROC  FAR
        .... .... .... ....
Q1:     .... .... .... ....
        .... .... .... ....
        Q   ENDQ

        .... .... .... ....
        CALL   P1                ; близък преход към P1 с връщане
        CALL   FAR  PTR  Q1      ; далечен преход към Q1 с връщане
        CALL   Q1                ; близък (!) преход към Q1 с връщане
        CALL   NA                ; близък преход към P
        CALL   FA                ; далечен преход към FA
        LEA    BX, Q             ; BX := Q – начален адрес на Q
        CALL   [BX]        ; близък (!) косвен преход към Q1 с връщане
        CALL   DWORD  PTR  [BX]         ; далечен преход към Q
        .... .... .... ....
```

Използването на команда CALL трябва да е много внимателно.

1. Трябва да се следи за съответствие на типа на прехода с типа на прехода при връщане – за това Асемблер не отговаря.
2. Аналогична на команда JMP, при използване на CALL с преход напред или с косвен преход, следва, ако е необходимо, да се уточнява типът на тези обръщения, тъй като по подразбиране, Асемблер генерира при преход напред късо извикване, а при косвен преход – близко косвено извикване.

9.3 Предаване на параметри чрез регистрите

В Асемблер няма формално определение за "чисти" процедури и за функции. По подобен начин стои и въпросът за формалните и действителните параметрите и тяхното предаване, което се отнася и до резултатите на процедурите и функциите. В езиците от високо ниво, този въпрос се решава с определено абстрактно правило, което програмистът лесно и бързо усвоява. Обикновено параметрите на процедурите се изписват във вид на списък след името й. Те са формални по отношение на нейния текст и стават действителни, когато се запише обръщение към процедурата, тъй като тогава в записа програмистът поставя в съответствие с всеки формален параметър, действителен такъв. Съответствието трябва да бъде по смисъл, по място и по тип. Съответствието или още еквивалентността между реалните данни, обявени за съответни на формалните променливи в процедурите се осъществява фактически от компилаторите.

Възможностите на Асемблер са по-скромни и реализацията на тези механизми се прехвърлят като специфични задачи на програмиста. Практикуват се различни подходи, някои от които ще поясним тук. При малко на брой параметри, между процедурата и главната програма обикновено се уговаря използването на част от регистрите. Регистровата памет играе ролята на пощенска кутия, от която процедурата взема необходимите ѝ данни, и където в последствие поставя получените резултати. От своя страна, главната програма знае за това и очаква необходимите ѝ резултати именно в тези регистри. Изборът на регистрите е право на програмиста.

9.3.1 Предаване на стойности

Ще разгледаме следния пример: необходимо е изчислението на стойността на израза: c=**max**(a,b)+**max**(5,a-1), където всички числа са със знак и имат формàт дума. Предлага се изчислението на функцията ***max*** да се запрограмира отделно, във вид на програмен инструмент, който да се използва при необходимост. Така задачата се структурира в две програмни единици

- Процедура, именувана ***max*** , за определяне на по-голямото от две дадени числа *x* и *y* – max(x,y) ;
- Главна програма за изчисляване на стойността на израза, в която на два пъти ще се обръщаме към процедурата, с една и съща цел, но с различни аргументи:
  max(a,b) и max(5,a-1) .

Освен това се уговаряме, че първият параметър викащата програма ще поставя в (ще предава чрез) регистър AX, вторият параметър – чрез регистър BX, и ще очаква да получи резултата в регистър AX. Това тя ще прави с ясното "съзнание", че стойността на първия параметър ще бъде загубена, тъй като на негово място, след изпълнението на процедурата, ще се окаже резултатът. За процедурите от тип функции обикновено съществува разбирането, че името им "връща" резултата. Това разбиране се изразява в разговорите между програмистите. Фактически обаче името на процедурата има смисъла на *адрес* (начален адрес, адрес за обръщение). Резултатът процедурата обменя според уговорения начин.

```
; Процедура  AX=max([AX], [BX])
     MAX  PROC  FAR
          CMP   AX, BX
          JGE   MAX1
          MOV   AX, BX
    MAX1: RET
     MAX  ENDP
; Главна програма
          .... .... .... ....
          MOV   AX, A
```

```
        MOV   BX, B
        CALL   MAX
        MOV   C, AX            ; C := (AX)=max(a,b)
        MOV   AX, 5
        MOV   BX, A
        DEC   BX
        CALL   MAX
        ADD   C, AX            ; C := (C)+(AX),  (AX)=max(5,a-1)
        ....  ....  ....  ....
```

Разгледаният пример илюстрира предаване на параметри чрез техните стойности. Възможен е вариант, при който в уговорените регистри не се намират стойностите на параметрите, а техните адреси. Тъй като между отделните програмни единици се предават адреси, т.е. указатели, връзката със стойностите на параметрите е косвена.

9.3.2 Предаване на указатели

Записът D(A) или D(B) обикновено се разбира така: изпълни процедура D с аргумент A или с аргумент B. Предполага се, че задачата на процедурата е да помести изчисления резултат в мястото на аргумента. И в двата случая имената на аргументите следва да се заместят от техните стойности, но в Асемблер имената представляват адреси. За да може процедурата D да изпълни своето изчисление, тя трябва да получи адреса на аргумента, за да прочете от паметта стойността. Предаването на адреса на параметър може да се реализира чрез регистър. Най-добре е това да бъде регистър-модификатор – BX, BP, SI или DI, тъй като вероятно на процедурата ще се наложи да модифицира този адрес.

Нека за процедурата D е избран регистър BX. В този регистър викащата програма следва да постави предварително адреса на параметъра A или B. Обръщението към съдържанието на адреса, както вече беше отбелязано, е косвено, т.е. чрез конструкцията [BX].

```
 ; Главна програма
        ....  ....  ....  ....
        LEA   BX, A
        CALL   D
        LEA   BX, B
        CALL   D
        ....  ....  ....  ....
 ; Процедура:  BX=адрес на X,  X := (X) / 16
     D   PROC
        PUSH  CX
        MOV  CL, 4
        SHR  WORD  PTR  [BX], CL
        POP  CX
        RET
     D   ENDP
```

В процедурата командата за изместване използва регистър CL. За да не разруши викащата програма, тя предварително записва CX в стека, а след това го възстановява. Тъй като аргументът е с формàт дума, той е указан с конструкцията WORD PTR [BX].

9.3.3 Съхранение на съдържанието на регистрите след влизане в процедура

Каквито и регистри да бъдат уговорени за предаване на параметрите и резултата между викащата и виканата програми, винаги е възможно част от регистрите да останат извън тази уговорка. Ако след влизане в процедура, за нейните изчисления са необходими още регистри, останали извън уговорените за предаване на параметрите, то тяхното съдържание е "табу" за нея. Ако все пак използването на такива регистри е неизбежно, тяхното съдържание следва да се съхрани непроменено. За целта естествено се използва стекът. За улеснение на програмиста, в следващите модели на процесора, в системата машинни команди са създадени командите PUSHA и POPA, които съхраняват и възстановяват регистровия файл на процесора в стека.

9.3.4 Предаване на параметри с по-сложен тип

Ще разгледаме случай на предаване на параметри по указател и когато параметърът представлява данни с по-сложен тип – масив, структура и пр. Даже процедурата да не променя тези данни, на нея в същност й се предава началния адрес на съответната конструкция. Това е неизбежен подход, тъй като на типът на данните се конструира от програмиста и не е известно дали наличните в процесора регистри ще му бъдат достатъчни. Освен това подходът на началния адрес е универсален, а той осигурява достъп до всяка част от данните в конструкцията на типа.

Ще разгледаме следния пример: дадени са масивите X и Y от едно байтови числа без знак:

```
X  DB  100  DUP (?)
Y  DB  25  DUP (?)
```

Необходимо е да се запише в регистър DL сумата от максималните елементи в двата масива: DL := max(X[k]) + max(Y[k]).

Тъй като тук се налага на два пъти да се определя максимален елемент, то за тази задача ще създадем процедура MAX. За да изпълни задачата, процедурата следва да получи началния адрес на параметъра си, т.е. на масива, както и неговата дължина в брой елементи. Адресът се уговаряме да предаваме чрез регистър BX, а броя на елементите чрез регистър CX. За връщане на резултата може да бъде уговорен регистър AL.

```
; Процедура MAX:  AL := max(W[k=0,1,2,…,n-1]), (BX)=[W], (CX)=n
        MAX  PROC
             MOV  AL, 0
```

```
MAX1:  CMP  [BX], AL
       JLE   MAX2
       MOV  AL, [BX]
MAX2:  INC   BX
       LOOP   MAX1
       RET
  MAX  ENDP
```

; *Викаща програма*

```
       .... .... .... ....
       LEA   BX, X
       MOV  CX, 100
       CALL  MAX
       MOV   DL, AL
       LEA   BX, Y
       MOV  CX, 25
       CALL  MAX
       ADD   DL, AL
       .... .... .... ....
```

9.4 Предаване на параметри чрез стека

Предаване на параметрите чрез регистрите е удобно и се използват често, но само когато параметрите са малко на брой. В противния случай регистрите са недостатъчни и се търси алтернатива. Такава алтернатива представлява стекът, като място, което е общо достъпно за всички програмни единици. Механизмът за реализация може да бъде различен. Тук ще бъде разгледан само един такъв.

Нека процедурата P има k на брой параметъра: P(a1,a2, … ,ak). Уговаряме, че преди обръщение към процедурата, викащата програма записва параметрите в стека. Важен е редът в който се записват параметрите. Редът се избира от програмиста. Тук в примера е избран редът отляво надясно, както са изписани в описанието на процедурата. Така a1 е първи, а ak е последен. За конкретност ще приемем още и ограниченията: всички параметри са с формат дума. След запис на действителните стойности на параметрите в стека, при обръщение към процедурата, върху тях се записва адресът за връщане. Това се постига по следния начин:

; *Викаща програма*

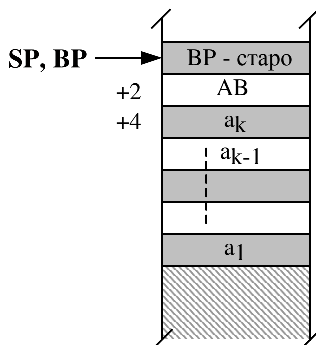
```
    ...  ...  ...
    PUSH   a1
    PUSH   a2
    .... .... .... ....
    PUSH   ak
    CALL   P
    .... .... .... ....
```

С извикването на процедурата, командата CALL записва в стека

адреса за връщане (АВ). Адресът за връщане, за който вече говорихме, е съдържание на адресната двойка CS:IP, като при близко извикване, това е само (IP).

След извикване на процедурата възниква въпросът: как да си вземе параметрите?. Тук съществува сериозен проблем, чиято същност се състои в това, че *четенето на параметрите от стековата област трябва да се извърши* *<u>без обичайното им изхвърляне</u>*, т.е. без модификация на стековия указател. За целта към стека се осигурява достъп чрез друг регистър-указател. За такъв в структурата на процесора е предназначен регистър BP. Първоначално той се инициализира като приема съдържанието на регистър SP. След това, модифицирайки съдържанието му с израза [BP+i], се получава многократен достъп до всяка дума, т.е. достъп без изхвърляне. Тук обаче се създава нов проблем – използвайки регистър BP, неговото съдържание се разрушава, а то може би е необходимо на викащата програма. Ето защо, преди инициализацията на този регистър, неговото съдържание също трябва да се съхрани в стека. По-долу са представени “входните” действия на процедурата и ответното съдържание на стека:

```
; “входни” действия на процедурата
     P  PROC
            PUSH  BP
            MOV   BP, SP
            .... .... .... ....
            .... .... .... ....
            .... .... .... ....
     P  ENDP
```



Както се вижда на рисунката, след записа на старото съдържание на регистър BP в стека, за достъп към последния параметър (ak), трябва да се използва адресния израз [BP+4] и командата MOV AX, [BP+4]. А, към следващия елемент (a(k-1)) – израза [BP+6] и т.н.

След изпълнение на входните действия процедурата изпълнява действията от своето тяло. В края, преди връщане към викащата програма, процедурата следва да изпълни ответни на входните, “изходни” действия. Ответните изходни действия имат за цел да бъде възстановен стека и регистрите в състоянието, в което те са били преди нейното извикване. Най-напред се възстановява съдържанието на регистър BP. Това следва да се извърши с командата (POP BP), която модифицирайки стековия указател SP, ще изхвърли от стека последно записаната дума. Едва тогава трябва да се изпълни командата RET, с което се възстановява действието на прекъснатата програма. Команда

RET изхвърля адреса за връщане АВ от стека. На програмата, получила управлението се пада задачата да освободи стека от преди това внесените параметри. По принцип тук възниква въпросът, кой трябва да изчисти стека – процедурата или викащата програма? В хода на вече описаните действия лесно се разбира, че това ще извърши викащата програма. Изхвърлянето от стека на ненужните вече параметри се постига чрез скокообразна модификация на стековия указател, например с командата ADD SP, 2*k.

Счита се обаче, че изхвърлянето на параметрите е по-добре да извърши процедурата. Мотивите за това са следните: обръщенията към процедурата могат да бъдат многократни, което ще бъде причина във викащата програма горе споменатата команда ADD да бъде записвана многократно. Така възниква общото правило в добрия стил на програмиране: *ако нещо може да се направи както в процедурата, така и в основната програма, то най-добре е това нещо да бъде направено в процедурата.*

И така, първоначално процедурата трябва да изчисти стека от параметрите и чак след това да предава управлението към викащата програма. Именно за целта на тези действия е създадена командата:

```
RET  i16          ; RET  <непосредствен операнд>
```

Това е разширен вариант на командата за връщане, в който непосредственият операнд (от тип дума: i16) се възприема като число без знак, с което се модифицира стековият указател. Действието на тази команда се изразява така:

$$\text{Стек} \rightarrow \text{IP}, \quad [\text{Стек} \rightarrow \text{CS}] \quad \text{SP} := (\text{SP}) + \text{i16}$$

Т.е. при тази команда освен възстановяване на адреса за връщане се извършва и възстановяване на стековия указател. Действията в квадратни скоби се изпълняват при дълъг преход.

Ще направим още няколко коментара. Команда RET – всъщност е разширената команда RET 0, т.е. връщане без изчистване на стека. Операндът i16 показва колко байта (а не думи) в стека следва бъдат изхвърлени. Ето защо изхвърлянето на k параметъра с формат дума изисква записване на числото 2*k, а не просто k. В операнда не следва да се отчита мястото, което заема адресът за връщане, тъй като командата RET вече го е прочела и го е изхвърлила.

Отчитайки всичко казано "изходните" действия на процедурата са:

```
; "изходните" действия на процедурата
        POP  BP       ; възстановяване на BP
        RET  2*k
     P  ENDP
```

След описаното връщане от процедура, състоянието на стека е такова, каквото е било до извикването на процедурата. Що се отнася до резултатите, които процедурата следва да върне, то те крайно рядко се предават чрез стека. Това обикновено става чрез регистър.

Пример: ще разгледаме процедурата NULL(A,N), чиято задача е да изчисти (да нулира) съдържанието на N на брой последователни байта в данновия сегмент, започвайки от клетка с адрес A. Уговорено е параметрите на процедурата да се предават чрез стека. По-долу в дясната колонка е текстът на процедурата, а в лявата – текстът на главната програма, в която процедурата NULL се извиква за нулиране на 100 елемента в масива XX.

```
; главна програма

  XX  DB  100  DUP (?)
   ....  ....  ....  ....
   ....  ....  ....  ....
  LEA    AX, XX
  PUSH  AX
  MOV    AX, 100
  PUSH   AX
  CALL   NULL
   ....  ....  ....  ....
   ....  ....  ....  ....
```

```
NULL   PROC
       PUSH  BP
       MOV   BP, SP
       PUSH  CX
       MOV  CX, [BP+4]
       MOV  BX, [BP+6]
 L1:   MOV  BYTE  PTR  [BX], 0
       INC   BX
       LOOP  L1
       POP  CX
       POP  BX
       POP  BP
       RET   4
NULL   ENDP
```

Както се вижда, главната програма използва регистър AX, за да зареди в стека началния адрес на масива XX (една дума) и броя на байтовете (числото N=100). Веднага след това извиква процедурата.

В процедурата първоначално се изпълняват пояснените "входни" действия, след което от стека (чрез указателя BP) се прочитат (без да се изхвърлят) предадените от главната програма два параметъра: броят на байтовете се зарежда в регистъра-брояч CX, а началният адрес на масива се зарежда в регистър BX. След което в цикъла, организиран с команда LOOP, адресираните чрез регистър BX клетки се зареждат с нули. След излизане от цикъла се изпълняват 2 неща: изпълняват се ответните "изходни" действия, с които се възстановяват старите съдържания на регистрите CX, BX и BP, а с връщането на управлението към главната програма от стека се изхвърлят 4 байта (две думи).

При изучаване на този пример, на читателя се препоръчва съставянето на рисунка, отразяваща съдържанието на стека, подобна на показаната преди това.

9.5 Локални данни за процедурите

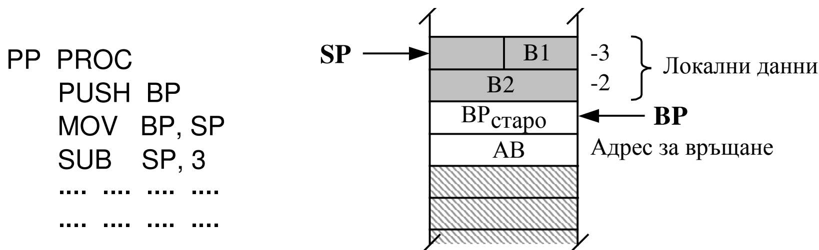
Под локални данни се разбират стойностите на величини, използвани от алгоритъма на процедурата и нямащи отношение към величините извън нея. Когато обаче локалните данни са много на брой, възниква естественият въпрос, къде да бъдат съхранявани. Разбира се,

за тях може да се резервира място в данновия сегмент, но това не е ефективно, тъй като това място в повечето време ще бъде блокирано и недостъпно за съхраняване на други данни. Счита се, че по-добре за целта да се използва отново стекът.

При влизане в процедурата в стека се резервират определен брой байтове, предназначени за локалните данни, които непосредствено преди излизане от процедурата се освобождават (изхвърлят).

Това резервиране може да се практикува както в случаите, когато параметрите се предават чрез регистри, така и когато за това се използва стека. За целта следва да се запомни съдържанието на регистър BP в стека, след което да се зареди с адреса на топ клетката (действия, аналогични на “входните”). След това съдържанието на стековия указател SP се намалява с число, равно на броя на резервираните байтове. Например, ако на процедурата PP са нужни 3 байта за локални данни (2 байта за променлива B2 и 1 байт за B1), то описаните действия могат да се реализират както следва:

```
PP  PROC
    PUSH  BP
    MOV   BP, SP
    SUB   SP, 3
    ....  ....  ....  ....
    ....  ....  ....  ....
```



След това достъпът до локалните данни се осъществява чрез адресни изрази от вида [BP-k]. Например, чрез [BP-2] – за локалната променлива B2 и чрез [BP-3] за локалната променлива B1.
При завършване, процедурата следва да изпълни командите:

```
     ....  ....  ....  ....
     MOV  SP, BP        ; SP := (BP)
     POP  BP             ; възстановяване на старото
     RET   3             ; изхвърляне на 3 байта
PP   ENDP
```

Необходимо е да поясним, че движението на указателя BP при работа с локалните данни (четене и запис) зависи от алгоритъма на процедурата, което не гарантира, че текущото му съдържание (в началото на горе представените действия - команда MOV) ще бъде правилно. Ето защо, ако е така, програмистът следва да осигури по принудителен начин правилното съдържание от вида [BP-k].

Като конкретен пример за текущата тема ще опишем близката процедура DIF, която има за задача да преброи колко са различните символи в даден низ, при условие, че началният адрес на низа се предава от главната програма чрез регистър BX, а дължината на низа –

чрез CX. За връщане на резултата е уговорен регистър AX.

Алгоритъмът на програмата е следния: резервираме в стека локален масив от 256 байта – по един на всеки възможен символ от ASCII-таблицата. Това означава, че на символ с №k в стека ще съответства байт с адрес [BP-256+k]. От формулата се разбира, че последният елемент се намира най-отдолу.

Процедурата претърсва дадения низ за съдържание на всеки един от тези 256 възможни символа, като отбелязва наличието му със запис на 1 в съответния байт на локалния масив. В края се натрупва сумата от елементите на локалния масив и така се получава броят на различните символи в низа.

```
     DIF   PROC
; входни действия на процедурата
           PUSH  BP
           MOV   BP, SP
; резервиране на 256 байта за локален масив
           SUB   SP, 256
; съхранение на регистри, които ще използва процедурата
           PUSH  BX
           PUSH  CX
           PUSH  SI
; изчистване на локалния масив
           MOV   AX, CX               ; съхранение на дължината на низа
           MOV   CX, 256              ; дължина на локалния масив
           MOV   SI, 0                ; в позиция на първия символ
    DIF1:  MOV  BYTE  PTR  [BP-256+SI], 0
           INC    SI
           LOOP  DIF1
           MOV   CX, AX          ; възстановяване в CX дължината на низа
; претърсване на низа
; и запис на 1 в съответния елемент на локалния масив
           MOV   AH, 0           ; нулиране на старшия байт
    DIF2:  MOV   AL, [BX]        ; код (№) на поредния символ от низа
           MOV   SI, AX          ; преписване в регистър модификатор
           MOV  BYTE  PTR  [BP-256+SI], 1       ; 1 в локалния масив
           INC    BX                             ; адрес на следващия символ
           LOOP  DIF2
```

```
; преброяване на единиците в локалния масив
        MOV   AX, 0                     ; начална 0 в брояча на 1
        MOV   CX, 256                   ; дължина на локалния масив
        MOV   SI, 0                     ; начален индекс в този масив

DIF3:   CMP  BYTE  PTR  [BP-256+SI], 1          ; има ли 1 ?
        JNE   DIF4
        INC   AX                                ; отброяване в AX

DIF4:   INC    SI

        LOOP   DIF3
; "изходни" действия на процедурата
        POP   SI
        POP   CX
        POP   BX
        MOV  SP, BP                     ; изхвърляне на локалния масив
        POP   BP
        RET

DIF     ENDP
```

Г Л А В А 10
С И М В О Л Н И   Н И З О В Е

Тук ще се спрем на способите за представяне и обработка на символни низове с фиксирана и с променлива дължина. Това са структури, които се променят в процеса на изпълнение на програмата.

10.1 Команди за обработка на низове. Префикси за повторение

Командите, които бяха изложени до момента са напълно достатъчни за реализиране на всякакви операции върху символни низове, но низовете са така важен тип данни, че в много процесори въвеждат специално разработени за целта команди, с които се цели ускоряване на обработката и облекчаване на програмиста. Особеното в тези команди е това, че с тяхна помощ може да се изпълни определено действие върху всички символи в низа. Командите за обработка на низове имат много общи характеристики.

10.1.1 Команди за сравняване на символни низове

Първо ще обърнем внимание на факта, че под низ тези команди могат да разбират както последователност от байтове, така и от думи. Във връзка с това всяка низова операция се представя с две команди: една е предназначена за обработка на низове от байтове, а другата – за обработка на низове от думи. В Асемблер за различаване на командите в мнемоничния код последната буква съответства на формàта: буква B за байт и буква W за дума, например, команди за сравнение на два низа:

CMPSB - сравни два низа от байтове ;
CMPSW - сравни два низа от думи.

Тъй като действията на командите са идентични, често се говори за команда CMPS (сравни низове) без да се споменава формàта. Командата се записва без операнди, а действието й се представя така:

[DS:SI]=[ES:DI] ? , SI:=(SI)+d ; DI:=(DI)+d ,

където с d е отбелязана стъпката за модификация на индексния регистър според стойността на флага за посока DF (вижте глава 1):

	DF=0	DF=1
CMPSB	+1	-1
CMPSW	+2	-2

По същество командата работи с два операнда, но те явно не са адресирани от нея, тъй като схемата за адресиране е фиксирана. Тя може да се използва, ако предварително е известно, че дължината на двата низа е еднаква. Въпреки че командата е една, поради малката дължина на разрядната мрежа, нейното изпълнение в същност е последователно повтарящо се (в цикъл) върху отделните порции на

низа. Удобството е в това, че програмистът не се грижи за модификацията на адресите. Освен това не е задължително низовете да се намират в един сегмент, т.е. те могат да са достатъчно далеко един от друг. Все пак свободата е ограничена до сегментите DS и ES. Обърнете внимание, че индексните регистри не съдържат индексите на символите, а тяхното отместване, т.е. техните адреси в сегментите. В зависимост от флага за посока, сравнението може да протича от началото към края на низа или обратно. За назначаване на съответната посока следва преди това да се определи стойността на DF. За целта са създадени специални команди: CLD – команда за сваляне (изчистване) на флага за посока и команда STD – за установяване в единица.

Резултатът от сравнението на двата низа е актуализация на флаговия регистър, която обикновено се използва при следващи условни преходи. Ще разгледаме следния малък пример: Нека K е някаква положителна константа и нека символен низ N1 е разположен в данновия сегмент, а низът N2, със същата дължина K, е разположен в допълнителния сегмент. Необходимо е да се провери дали символните низове N1 и N2 съвпадат. Представени са два варианта – сравнение в посока напред и в посока назад:

```
; посока напред
      CLD
      LEA  SI, N1
      LEA  DI, N2
      MOV  CX, K
  L:  CMPSB
      JNE   NE
      LOOP  L
 NE:  ....  ....  ....  ....
```

```
; посока назад
      STD
      LEA  SI, N1+K-1
      LEA  DI, N2+K-1
      MOV  CX, K
  L:  CMPSB
      JNE   NE
      LOOP  L
 NE:  ....  ....  ....  ....
```

Ако разгледаме двата варианта, ще отбележим, че в реализацията на циклите няма разлика. Този факт води до допълнително опростяване на текста на програмата чрез използване на *префикси за повторение*.

10.1.2 Префикси за повторение

В системата машинни команди на процесорите на *Intel* са разработени две команди без операнди, наречени префикси. В езика Асемблер всяка една от тях има по няколко синонимни наименования. Първият префикс е:

REPE (*repeat if equal*) - повтаряй докато е равно ;
REPZ - повтаряй докато е нула ;
REP - повтаряй .

Вторият префикс е:

REPNE (*repeat if not equal*) - повтаряй докато е различно ;
REPNZ - повтаряй докато не е нула.

Тъй като сравнението на X с Y (X=Y?) се реализира след привеждане на отношението към еквивалентното му (X-Y=0?) (вижте [1]) то изказванията "докато е равно" и "докато е нула" са еквивалентни.

Префиксите за повторение се поставят пред командите, за които се отнасят, като се записват на един ред. Например: REPE CMPSB.

Ако командата след префикса не е команда за обработка на символен низ, то той няма да окаже никакво влияние на тази команда, т.е. за текста той ще бъде като прозрачен и несъществуващ или може да се тълкува като "празна" команда.

Двойката команди

<префикс> <низова команда>

се изпълнява по следния алгоритъм:

```
L:   if CX=0 then  go to L1 ;
       CX := (CX)-1 ;
       <низова команда - сравнение> ;
     if ZF=1 then  go to L ;
L1:  .... .... .... ....
```

Префиксите организират цикъл от вида с предварително известен брой повторения, като за целта използват броячния регистър CX. За да се случи това обаче, програмистът следва да зареди този регистър с необходимата начална стойност. За да се изпълни цикълът като цикъл с предусловие, началната стойност в брояча следва да бъде нула.

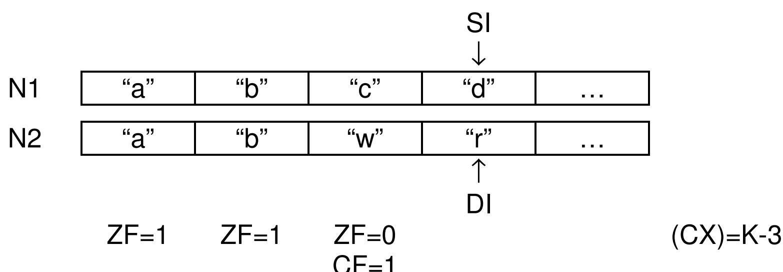
Това обаче не е единствената причина за излизане от цикъла. Наличието на 0 в брояча е причина за нормален изход от цикъла. При нормална работа на низовата команда това означава, че са проверени всички символи в низа и цикълът е прекратен поради тяхното изчерпване. Ако разгледаме обаче по-горе представената алгоритмична схема, ще видим, че е възможен и преждевременен изход от цикъла, ако след поредното сравнение проверяваният флаг се окаже вдигнат. С други думи сравненията продължават, докато сравняваните двойки са еднакви, но се прекратяват ако текущата не е еднаква. Останалите двойки символи е безсмислено да се сравняват, тъй като е достатъчно поне едно различие, за да се твърди, че символните низове не са еднакви, т.е. че са различни.

Както се разбира, има два изхода от цикъла с команда REPE – нормален $(CX)=0 \cap (ZF)=1$ и преждевременен $(CX) \neq 0 \cap (ZF)=1$. Ако е необходимо да се знае точно по коя причина той е прекратен, това следва да се установи с допълнителна проверка на стойността на флага ZF. И така, дадения по-горе пример с помощта на префикс за повторение ще изглежда така:

```
CLD
LEA   SI, N1
LEA   DI, N2
MOV  CX, K
```

```
          REPE  CMPSB
          JE   EQ                  ; еднакви ли са низовете?   (ZF=1?)
NOEQ:     ....  ....  ....  ....
```

За да изясним още някои детайли ще разгледаме два конкретни низа.



Първите два елемента са еднакви и ZF=1. Третите елементи са различни, поради което ZF=0, а цикълът се прекратява. Тъй като след цикъла флагът ZF е различен от 1, управлението в програмата се предава в точка с етикет NOEQ. Ако всички елементи на низовете бяха еднакви, изходът от цикъла щеше да бъде при състояние (CX)=0 и (ZF)=1, т.е. $(CX)=0 \cap (ZF)=1$ и командата за условен преход JE щеше да предаде управлението в точка EQ. Схемата позволява да се направят още следните коментари:

1. Префиксите за повторение са за условието равно (не равно), но и за условието по-голямо (по-малко). При срещане на различна двойка символи цикълът се прекратява (ZF=0). Ако е необходимо да се установи какво точно е различието, следва да отчетем стойностите и на другите флагове. В горния пример “c”<”w” (кодовете на символите са без знакови числа), при което имаме CF=1. Така можем да твърдим, че низовете са в отношение N1<N2. Установяването на този факт естествено се постига чрез допълнителна команда за условен преход, например JA. Така след сравнението на символните низове при необходимост можем да поемем по един от три възможни пътя: N1=N2, N1<N2 и N1>N2.

2. След излизане от цикъла в регистрите SI и DI не остават адресите на онези елементи, причинили излизането. Както се вижда от горната схема, това са адресите на следващите елементи. Това се дължи на реда на действията в командата за сравнение, която първо сравнява, а след това модифицира съдържанието на индексните регистри.

3. След излизане от цикъла в регистър CX винаги се намира число, което показва колко елементи от низовете са останали неанализирани, т.е. несравнени. Не е трудно да се обобщи, че след излизане от цикъла в този случай, в регистрите SI, DI и CX остават такива стойности, които позволяват да се реализира сравнение на

останалата част от низовете, с изключение на двойката различни символи. От това можем да се възползваме например за преброяване на различните двойки:

```
            MOV   AX, 0      ; брояч за различните двойки
            CLD
            LEA   SI, N1
            LEA   DI, N2
            MOV   CX, K
  COMP:     REPE  CMPSB
 ; преход на FIN, ако изходът от цикъла е при еднакви низове
            JE   FIN
 ; в противен случай отброяване на различната двойка AX:=(AX)+1
            INC   AX
            CMP  CX, 0        ; ако не е достигнат края на низа
            JNE   COMP        ; отново влизаме в него
     FIN:   ....  ....  ....  ....
```

И последен коментар. Ако до влизане в цикъла съдържанието на регистър CX е било равно на нула, то стойността на флага ZF след цикъла ще бъде такава, каквато е била преди цикъла. Ето защо по стойността на този флаг не би могла да се установи причината за изхода от цикъла. Това следва да се отчита при сравнение на празни низове.

Сега ще се спрем на другия префикс за повторение REPNE, REPNZ. Тук всичко е аналогично, с тази разлика, че заменена стойността на флага ZF на противоположната. Така двойката команди:

<префикс> <низова команда>

се изпълнява по следния алгоритъм:

```
L:   if CX=0 then  go to L1 ;
       CX := (CX)-1 ;
       <низова команда - сравнение> ;
     if ZF=0 then  go to L ;
L1:  ....  ....  ....  ....
```

Смисълът на префикса REPNE пред команда CMPS се състои в следното: повтаряй сравнението на елементи от низовете докато те не са еднакви (не са равни), но не повече от (CX) пъти. С други думи, докато елементите остават различни, сравнението продължава, но намери ли се двойка еднакви символи, цикълът се прекратява. Двойката команди (REPNE CMPS) се използва, когато е необходимо да се намери първата двойка еднакви символи, започвайки от началото или от края на низа.

10.1.3 Други низови команди

Ще разгледаме и други низови команди, но по-кратко, тъй като в главното те са аналогични на вече описаните.

Команди за сканиране на низове SCASB, SCASW

При команда SCASB съдържанието на регистър AL се сравнява със съдържанието на клетка от паметта, която се адресира чрез регистровата двойка ES:DI, след което съдържанието на DI се модифицира с +1 или –1:

$$(AL)=[ES:DI]? , \quad DI:=(DI)\pm 1$$

Посоката на модификацията се определя от флага за посока: с +1 при DF=0 и с –1 при DF=1.

При команда SCASW действията са аналогични. Сравнява се съдържанието на регистър AX със съдържанието на две клетки от паметта, чийто начален адрес е в адресната двойка ES:DI, след което съдържанието на регистър DI се модифицира с ±2.

$$(AX)=[ES:DI]? , \quad DI:=(DI)\pm 2$$

Командата SCASB(W) се използва при търсене на елемент в низ, еднакъв със зададения в регистър A, или различен от зададения, в зависимост от предхождащия командата префикс:

REPNE SCASB - е команда, търсеща първия еднакъв елемент
(повтаряй сравнението докато са различни) ;

REPE SCASB - е команда, търсеща първия различен елемент
(повтаряй сравнението докато са еднакви).

Ще разгледаме следния пример: В низа SS от 500 символа, описан в данновия сегмент, е необходимо да се замени първият срещнат символ звезда "*" с точка.

Търсеният символ звезда трябва да запишем в регистър AL, а претърсването в низа ще реализираме са командата REPNE SCASB. Ако символът звезда се съдържа в низа, след излизане от цикъла флагът ZF ще има стойност 1, а в регистър DI ще се съдържа адресът на следващия елемент. И така, ето текстът на програмата:

```
        CLD                              ; DF=0,  посока напред
        PUSH  ES                         ; спасяване на ES
        PUSH  DS                         ; спасяване на DS
        POP   ES                         ; ES := (DS)  и изхвърляне
        LEA   DI, SS                     ; DI := offset SS
        MOV   CX, 500
        MOV   AL, "*"
        REPNE  SCASB                     ; докато са различни
        JNE   FIN                        ; ZF=1
        MOV  BYTE  PTR  ES:[DI-1], "."   ; замяна на точка
  FIN:  POP   ES                         ; ES := (ES) и изхвърляне
        ....  ....  ....  ....
```

Програмата прехвърля съдържанието на DS в ES през стека. Тъй като при това съдържанието на ES се разрушава, след обработката на низа то се възстановява от стека.

Команди за прехвърляне на низове MOVSB, MOVSW

При команда MOVSB(W) се прехвърля (копира) байт или дума от данновия сегмент с начален адрес DS:SI в клетка на допълнителния сегмент с начален адрес ES:DI. Действията на командата са подобни на действията на команда CMPS:

[ES:DI] := ([DS:SI]) , SI:=(SI)+d , DI:=(DI)+d

Флаговете не се променят от тези команди.

Командата MOVS (и в двата й варианта) прехвърля символ от един низ в друг, след което се настройва за същото действие върху следващия елемент. За цялостно прехвърляне на символен низ е необходимо само нейното зацикляне с подходящ префикс. Тъй като командите не променят флаговете, изходът от цикъла е възможен само след изчерпване на заявената дължина, т.е. по причина, че (CX)=0. За повторение се използва префиксът REP:

```
REP  MOVSB       ; копиране на CX байта (или думи при MOVSW)
```

Ясно е, че основното приложение на тази команда е за бързо прехвърляне на съдържанието на една област от данновия сегмент в допълнителния сегмент. Например, ако са дефинирани масивите:

```
X  DW  100  DUP (?)
Y  DW  100  DUP (?)
```

и е необходимо присвояването X:=Y, то това може да се направи така:

```
CLD                   ; посока напред
LEA  SI, Y            ; SI := offset Y
PUSH  DS              ; спасяване на DS
POP  ES               ; ES := (DS)
LEA  DI, X            ; DI := offset X
MOV  CX, 100          ; CX := 100
REP  MOVSW            ; копира 100 думи
```

Команди за запис на низове STOSB, STOSW

Командата STOSB записва съдържанието на регистър AL в клетка от паметта, чийто адрес сочи адресната двойка ES:DI, след което модифицира съдържанието на регистър DI според стойността на флага за посока DF, т.е. DI:=(DI)±1 (+1 при DF=0 и –1 при DF=1).

Командата STOSW прави същото, но със съдържанието на AX, като модифицира DI с константа +2 или –2.

Тези команди не променят флаговете.

Ясно е, че тези команди могат да се зациклят с префикс REP, при което определена област в паметта може да бъде запълнена с едно и също съдържание. Например: да се запълни с празен символ 40-символен низ QQ в данновия сегмент.

```
MOV  AL, " "
CLD
PUSH  DS
POP  ES
```

```
        LEA   DI, QQ
        MOV  CX, 40
        REP  STOSB
```

Команди за зареждане на низове LODSB, LODSW

Тези команди четат съдържанието на клетка от паметта и го зареждат в регистър AL (AX). Адресът на клетката се сочи от двойката регистри DS:SI, след което съдържанието на SI се модифицира автоматично с константа ±1 или ±2. Командите не променят стойностите на флаговете.

Поставянето на префикс пред тези команди е безсмислено, тъй като приемникът на данните е с крайно ограничена дължина. Обикновено тази команда се използва в комбинация с команда STOS за презапис на символни низове, когато преди това върху техните елементи се извършва някакво въздействие. Като пример ще разгледаме задачата за смяна на знака на елементите на масива X и тяхното прехвърляне в друг масив Y.

```
        CLD                 ; посока напред
        LEA   SI, X         ; задаване на адреса за четене на символ
        PUSH  DS            ; спасяване
        POP   ES            ; подмяна
        LEA   DI, Y         ; задаване на адреса за запис
        MOV  CX, 100        ; начална стойност на брояча
    L:  LODSB               ; четене в AL
        NEG   AL            ; смяна на знака (доп. код)
        STOSB               ; запис на AL в паметта
        LOOP   L
```

10.1.4 Команди за зареждане на адресни двойки регистри

Съвместното използване на префиксите с командите за обработка на символни низове позволява на програмиста значително да опрости и ускори програмата, но преди използването на тези команди му се налага да запише достатъчно много команди, назначаващи различни необходими за това стойности (флагове, броячи, адреси и пр.). Следващите две команди, които ще разгледаме тук спомагат в известна степен да се съкратят и тези действия. Става дума за зареждането на адресните двойки DS:SI и ES:DI с начални стойности.

Зареждане на указател в DS (Load pointer using DS): LDS r16, m32

Като втори операнд (m32) в тази команда следва да бъде посочен адрес на двойна дума в паметта, където се съхранява двойката *seg*:*offset*, представляваща абсолютен адрес на клетка (да кажем например начален за някакъв низ). Изпълнението на зареждането е:

r16 := (m32) ,   DS := (m32+2)

Тази команда не променя флаговете.

Зареждане на указател в ES (Load pointer using ES): LES r16, m32

Тази команда е аналогична на предидущата, само че тя се отнася до друг сегментен регистър – ES.
Пример:

```
DATA1   SEGMENT
 STR1   DB 400 DUP (?)
  AS2   DD STR2                    ; AS2 = DATA2:STR2
DATA1   ENDS
DATA2   SEGMENT
 STR2   DB 400 DUP (?)
DATA2   ENDS
 CODE   SEGMENT
        ASSUME CS:CODE, DS:DATA1
        .... .... .... ....
; нека в този момент (DS)=DATA1
        CLD
        LEA SI, STR1               ; начало на STR1 (DS:SI)
        LES DI, AS2                ; начало на STR2 (ES:DI)
        MOV CX, 400
        REP MOVSB                  ; копира STR1 в STR2
        .... .... .... ....
```

10.2 Символни низове с променлива дължина

Често алгоритмите на програмите променят дължините на низовете. Низовете с фиксирана дължина се възприемат като масиви и тяхната обработка не се нуждае от допълнителни пояснения. Тук ще стане дума за низовете с променлива дължина, особено когато те се удължават, защото скъсяването е сравнително по-проста ситуация.

Когато стане дума за низ с променлива дължина, на преден план изниква въпросът, как да бъде деклариран той и колко клетки в паметта са му необходими? Възможен изход от това затруднение е предварителното деклариране на максимално възможната дължина. Например, ако е известно, че във всеки един момент от изпълнението на програмата дължината на низа няма да надхвърли 200 символа, тогава е ясно, че следва да се предвидят 200 байта:

STR DB 200 DUP (?) , т.е. дължина ≤ 200.

При това наличните в низа символи винаги ще се разполагат в началото на това пространство (ще бъдат ляво подравнени).

Ако обаче и максималната дължина не е възможно да бъде предвидена, тогава се спасяваме чрез понятието *списък*.

Преди да разгледаме случаят със списъците, ще поясним допълнително ситуацията с максимално възможната дължина.

При положение, че дължината на низа се променя, то въпросът е каква е текущата дължина? Възможностите за определяне или разпоз-

наване края на низа са две – да се дефинира уникален символ, маркиращ края на низа, или дължината да се съдържа в низа, например като число, записано в първия байт.

В първия случай обикновено като маркер се използва байт, съдържащ 8 нули. Според ASCII таблицата 8 нули представляват кодовата комбинация на празен символ (*blanc*). Със скъсяване или удължаване на символния низ се измества и маркерът. Този начин за представяне на текущото състояние на низа обаче има съществен недостатък, който се състои в това, че ако трябва да се определи дължината на низа, то следва да се сравнят всички символи в него до откриване на маркиращия края му. Тези действия често идват в повече.

Второто решение се приема за по-добро. В началото на низа се добавя един байт, в който е записан броят на символите в низа. Това решение обаче също има своите неудобства. Например, ако дължината на низа е по-голяма от 255, то за представяне на числото са необходими вече 2 байта. Ако следва да се получи достъп до байт №n, тогава отместването следва да се коригира: при отделен един байт с (*offset*+1), а при 2 байта с (*offset*+2), тъй като символите в низа са изместени от числото, стоящо в началото. По тази причина, когато се декларира типа на низа трябва да се отчете броя на допълнителните байтове. Например низ NN с не повече от 100 символа трябва да се декларира така:

```
NN  DB  101  DUP (?)
```

Пример: в данновия сегмент е даден низ S, чиято дължина е не повече от 200 байта. Да се премахне от низа първият срещнат символ "+", ако такъв има.

Първоначално търсим символ "+", за което ще използваме команда за сканиране на низа с подходящ префикс за повторение:

```
; търсене на символ "+" в низа S
        PUSH  DS
        POP  ES
        LEA  DI, S+1            ; зареждане на (offset+1) на низ S
        CLD                     ; посока напред
        MOV  CL, S              ; четене на първия байт (брой байтове)
        MOV  CH, 0              ; допълване до CX
        MOV  AL, "+"
        REPNE  SCASB            ; повтаряй докато не съвпадат
        JNE  FIN                ; "+" не се съдържа в низа S
```

Ще отбележим още следното: Ако символ плюс се съдържа в низа, то в момента когато бъде открит и цикълът се прекрати преждевременно по тази причина, двойката ES:DI ще сочи следващия байт, а в брояча CX ще се съдържа число, което ще показва броя на символите след "+".

В този момент, според задачата, символ “+” следва да бъде отстранен от низа. Как да разбираме това? Тъй като празно място в низа не може да бъде оставяно, това означава, че всички следващи символи трябва да се изместят в посока към началото на една позиция. Тези действия могат да бъдат изпълнени в цикъл от команда MOVS. Броят на повторенията е числото, намиращо се в CX, а флагът за посока остава непроменен – напред. Така, че завареното положение е благоприятно. Командата MOVS прехвърля съдържанието на клетка, адресирана от двойката DS:SI в клетка с адрес ES:DI. За целта следва да заредим регистър SI с адреса на символа, следващ “+”, а регистър DI да заредим с адреса на “+”, т.е. да посочим мястото откъдето ще излиза и мястото където ще отива първия преместван символ. Така задачата завършва с текста:

```
; отстраняване на символ "+" от низа S
        MOV  SI, DI              ; DS:SI  - откъдето излиза
        DEC  DI                  ; ES:DI  - където отива
        REP  MOVSB
        DEC  S
FIN:    ....  ....  ....  ....
```

Г Л А В А 11

М А К Р О С Р Е Д С Т В А

Тук ще разгледаме макросите, блоковете за повторение и средства за условно асемблиране, разширяващи възможностите на езика Асемблер.

11.1 Макроезик

Програма, написана на макроезик, се транслира на два етапа. Първоначално тя се преобразува на “чист” Асемблер, след което не съдържа макросредства. Този етап се нарича макрогенерация, който се реализира от специален транслатор – макрогенератор. На втория етап от асемблерския код се получава машинната програма.

Макрогенератор и Асемблер си взаимодействат различно. Те могат да действат самостоятелно и независимо един от друг. В този случай макрогенераторът няма достъп до цялата информация на Асемблер. В алтернативния случай двата транслатора действат съвместно, редувайки се по време на транслацията. В такива условия двата транслатора представляват отделни части на един транслатор, който обикновено се нарича Макросемблер (MASM).

11.2 Блокове за повторение

При съставяне на текстовете на програмите често възниква ситуация, при която даден фрагмент е необходимо да се повтаря няколко пъти в различни части на програмата. В отговор на такива необходимости в Асемблер се реализират конструкции, наречени блокове за повторение (*repeat blocks*):

```
<Заглавие>
    < тяло >
 ENDM
```

Тялото е всяка последователност от оператори и команди (в частност и блокове за повторение). Думата ENDM е директива, указваща края на блока. Срещайки в изходния текст такъв блок, макрогенераторът поставя в крайната програма вместо него копие на тялото му.

Тялото може да се копира без изменения, но може да се копира и с някои модификации. Какви са възможностите и как те се осъществяват, всичко това зависи от заглавния оператор. Съществуват три вида заглавни оператори, а от там и три вида блокове за повторение:
REPT-блок, IRP-блок, IRPC-блок.

REPT-блок

Този блок за повторение има структурата:

```
REPT  k
  < тяло >
ENDM
```

където с k е означен константен израз с неотрицателна стойност. Изразът следва да е такъв, че неговата стойност да може да се изчисли веднага (в него не трябва да се съдържат обръщения напред). Изчислявайки стойността на k, макрогенераторът създава k точни копия на тялото на блока и ги помества в крайния текст на програмата. Например, блокът:

```
REPT  3
  SHR  AX, 1
ENDM
```

ще построи в програмата следната последователност:

```
SHR  AX, 1
SHR  AX, 1
SHR  AX, 1
```

Друг пример – отляво е показан фрагмент от първоначалната програма, а вдясно, построения по него фрагмент в крайната програма:

```
N  EQU  6       ⇒      N  EQU  6
   REPT  N-4              DB  0, 1
   DB  0, 1               DW  ?
   DW  ?                  DB  0, 1
   ENDM                   DW  ?
```

В блоковете за повторение доста често се използва директивата за присвояване (=). Например, да се опише 100-байтов масив X, елементите на който имат началните стойности от 0 до 99. Вляво е описанието, а в дясно реализацията:

```
X  DB  0                   X  DB  0
K=0                        K=0
  REPT  99      ⇒          K=K+1    ⎫ 99 такива двойки
  K=K+1                      DB  K  ⎭
  DB  K                    K=K+1
  ENDM                       DB  K
                           K=K+1
                             DB  K
                           …   …   …
```

IRP-блок

Този блок има следната структура:

```
IRP  p, <v1, v2, … , vk>
  < тяло >
ENDM
```

Забележка: ъгловите скобки са явно изписвани.

В заглавния оператор с p е означено име, което е формален параметър и може да бъде употребено в тялото на блока. В ъгловите скоби е описан списък от действителни параметри v1,…,vk. Това са произволни текстове (включително и празни), но те следва да бъдат:

- Балансирани по броя на кавичките ;
- Не трябва да съдържат запетаи (,), точка и запетая (;), както и ъглови скоби. Ако се налага да се изписват такива символи, то следва да се употребят макрооператори, които ще разгледаме по-късно тук.

Срещайки такъв блок, макрогенераторът генерира k на брой копия на тялото му (по едно на всеки действителен параметър), при което във всяко текущо копие с номер i, всяка употреба на формалния параметър p в тялото се подменя с действителния vi. Например, отляво изходния текст, вдясно неговия реален вид:

```
IRP  REG, <AX, CX, SI>      ⇒     PUSH  AX
  PUSH  REG                       PUSH  CX
ENDM                              PUSH  SI
```

Повторена е командата PUSH, но всеки път с различен действителен операнд. Ще отбележим, че формалният параметър (името REG) е локално за тялото на блока и не може да бъде използвано извън него. Ако то съвпада с името на друг обект в програмата, то няма да пренася смисъла и стойността с в блока за повторение, където ще има само стойностите, с които е дефинирано в заглавния му оператор. Например, в блока:

```
IRP  BX, <1, 5>
   ADD   AX, BX
ENDM
```

името BX е дефинирано в заглавния оператор на блока като име на формален параметър (а не като име на регистър) и като такова се възприема в тялото му. Ето защо макрогенераторът генерира следния текст:

```
ADD   AX, 1
ADD   AX, 5
```

Замяната на формалния параметър с фактическите параметри – това са чисто текстови подстановки, без отчитане на смисъла – просто един участък от текста (p) се заменя на друг (vi). При това с параметъра p може да бъде означена всяка част от едно изречение или дори цялото изречение, но две и повече изречения той не може да означава. Защото след замяната на формалния параметър трябва да се получават правилни изречения от гледна точка на езика Асемблер. Например:

```
IRP  Q, <DEC  WORD  PTR,   L:  INC>   ⇒      DEC  WORD  PTR  W
  Q  W                                       JMP  M2
  JMP  M2                                 L:  INC   W
ENDM                                         JMP  M2
```

Ще отбележим също, че в тялото на блока се заменят само повторенията на формалния параметър, а другите имена се пренасят в копията на тялото без изменения. Например:

```
N  EQU  1                      N  EQU  1
IRP  P, <A, B>      ⇒          A  EQU  N
  P  EQU  N                    B  EQU  N
ENDM
```

Останалите особености при запис на формалните и действителните параметри ще бъдат разгледани по-късно тук.

IRPC-блок

Този блок има следната структура:

```
IRPC  p, s1s2…sk
  < тяло >
ENDM
```

Тук с p е означен формалният параметър, след който са записани последователно символи si. Могат да бъдат записвани произволни символи с изключение на празен символ и символ точка със запетая (;) – припомняме, че този символ (;) е резервиран за разпознаване на коментарите, а с празен символ се означава края на имена и операнди.

Срещайки IRPC-блок, макрогенераторът генерира k на брой копия на тялото на блока (по едно за всеки символ), при което в i-тото копие, всички записи на формалния параметър p се заменят на символ si. Например:

```
IRPC  D, 17W                  ADD  AX, 1
  ADD  AX, D        ⇒         ADD  AX, 7
ENDM                          ADD  AX, W
```

Макрооператори

При използване на блокове за повторения (и макроси, които ще бъдат разгледани по-късно) възникват редица проблеми при записване на техните формални и фактически параметри. Тези проблеми се решават с помощта на макрооператори – оператори, чиято употреба е разрешена само в конструкции на макроезика.

Макрооператор &

Разглеждаме следния блок за повторение и генерираните копия:

```
IRP  W, <VAR1, VAR6>
  W  DW ?                  ⇒      VAR1  DW  ?
ENDM                              VAR6  DW  ?
```

Тук името W, означава име на променлива като цяло, но имената VAR1 и VAR6 се различават частично по последния символ. Възниква идеята за дефиниране именно само на това различие, а не на цялото име. Възможно ли е това с декларацията?

```
IRP  W, <1, 6>
  VARW  DW ?
ENDM
```

Може да се каже не, защото възникват твърде много неясноти. Например, защо формалният параметър да се заменя в името на

променливите, а не следва да се заменя в директивата DW.

Изход от ситуацията се постига чрез внасяне на допълнителна яснота, като се посочва къде точно следва да се извършват замените. Това се постига с помощта на символ “&” , който има смисъла на логическо И. Така горната декларация би се определила еднозначно:

```
IRP  W, <1, 6>
  VAR&W  DW ?
ENDM
```

Знакът “&” може да се поставя няколкократно. Самостоятелната му употреба е без последствия. Макрооператорът & се използва не само когато формалният параметър “се слива” със съседните имена и числа, но и когато трябва да бъде указан вътре в низа. Работата е в това, че макрогенераторът игнорира влизанията на формалния параметър в низа и за да му се обърне внимание на това, преди тях трябва да се поставя знакът &, а ако не е ясна неговата дясна граница, то знак & трябва да се поставя и след формалния параметър. Например:

```
IRPC  A, “<
  DB  “A,&A,&A&B”        ⇒      DB  “A,”, ”B”
ENDM                            DB  “A,<,<B”
```

И още една особеност на макрооператора &: ако редом се поставят няколко знака &, то макрогенератор ще премахне само един от тях. Това е направено специално с отчитане на възможността за вложеност на блоковете за повторение (и/или на макроси). Например:

```
IRPC  P1, AB             IRPC  P2, HL             INC  AH
IRPC  P2, HL             INC  A&P2                INC  AL
INC  P1&&P2       ⇒      ENDM               ⇒     INC  BH
ENDM                     IRPC  P2, HL             INC  BL
ENDM                     INC  B&P2
                         ENDM
```

Срещайки в текста на изходната програма блок за повторение (лявата колонка), макрогенераторът първо ще създаде първото копие на тялото на външния блок, в което всички влизания на неговия формален параметър P1 ще бъдат заменени със символ A (вижте първите 3 реда в средната колонка). При това в команда INC от двата стоящи един след друг знака & ще бъде отстранен един и останалият знак & ще отделя формалния параметър P2 на вътрешния блок. Ако в тази команда имаше само един знак &, то командата би имала вида

INC AP2 ,

и тогава записът AP2 не би се възприемал като състоящ се от две части – A и P2. Доколкото в полученото копие остават конструкции на макроезика, то макрогенератор продължава работата си, като разкрива и вътрешния блок, с което получава чистия асемблерски текст (двата реда в дясната колонка). Следва създаването на второто копие на външния блок, което се обработва аналогично.

Макрооператор <>

Беше казано,че фактическите параметри на IRP-блока не следва да съдържат запетаи, точка със запетая и ъглови скоби, а в IRPC-блока не следва да съдържат празни символи и точка със запетая. Когато тези ограничения трябва да бъдат нарушени, то последователността от символи трябва да се затвори в ъглови скоби. При това се приема, че външните ъглови скоби не се отнасят към параметъра или към последователността и че те указват само техните граници. Примери:

```
IRP  VAL, <<1,2>,3>               DB  1, 2
DB  VAL                  ⇒        DB  3
ENDM
IRPC  S, <A;B>                    DB  “A”
DB  “&S”                 ⇒        DB  “;”
ENDM                              DB  “B”
```

Макрооператор !

Макрооператорът, който има синтаксиса:

!<символ>

е предназначен за задаване на специални символи. Смисълът на записа е следният: удивителният знак (!) се отстранява, но следващият след него символ се възприема такъв, какъвто е, без да му се приписва друга интерпретация. Например:

```
IRP  X, <A!>B, Здравей!, ПК!!>          DB  “A>B”
DB  “&X”                       ⇒        DB  “Здравей, ПК!”
ENDM
```

Макрооператор ! може да бъде използван само при запис на фактически параметри на IRP-блокове (и макроси), докато в IRPC-блокове знакът ! се разглежда като обикновен символ.

Макрооператор %

В записа на фактическите параметри в оператор IRP-блок може да се използва още конструкцията:

%<константен израз>

При среща на такава конструкция в записа на фактическите параметри, макрогенераторът изчислява стойността на указания израз и замества с нея цялата конструкция. Например:

```
K  EQU  4
....  ....  ....  ....
IRP  A, <K+1, %K+1, W%K+1>              DW  K+1
DW  A                          ⇒        DW  5
ENDM                                    DW  W5
```

Влагане на макрооператори % не се допуска. Например, в конструкцията %5-%K ще бъде генерирана грешка от вида “неописано име %K”. Като край на константен израз се приема първият срещнат символ, който според синтаксиса за константен израз, не може да му

принадлежи, например, запетая, ъглова скоба или знак равно. Например, при стойност 4 на константата К, параметърът %К-1+К=К ще бъде преобразуван в 7-К.

В последователността от символи (във втория операнд) на IRPC-блока, знакът % ще се възприема като обикновен символ, а не като макрооператор.

Макрооператор ;;

Ако в тялото на блок за повторение (и макрос) се съдържат коментари, те се пренасят във всички копия на тялото на блока. Но коментарите са полезни при описание на самия блок за повторение, а в копията те са ненужни. Тяхното копиране може да се забрани, ако те се предхождат двукратно от символа точка и запетая (;;). Например:

```
        IRP  R, <AX, BX>
    ;; възстановяване на регистрите          ; да се възстанови AX
    ; да се възстанови R                         POP  AX
        POP  R    ;; топ стек → R            ; да се възстанови BX
        ENDM                                     POP  BX
```

11.3 Макроси

За разлика от блоковете за повторение, които създават регулярни копия, макросите са предназначени да създават нерегулярни копия на фрагменти от текста на програмите. Срещайки името на даден макрос, макрогенераторът подменя обръщението към макроса със самия макрос. При използване на макроси се прилага специална терминология. Описанието (дефинирането) на макроса се нарича *макроопределение*. Обръщението към макрос се нарича *макрокоманда*, а резултатът от подмяната се нарича *макроразширение*.

11.3.1 Макроопределения

Макроопределението има следната структура:

```
    <име на макрос> MACRO <списък от формални параметри>
        < тяло на макроса >
        ENDM
```

Ето два примера:

```
 SUM    MACRO  X, Y,   ;X:=X+Y             VAR   MACRO  NM, TP, VL
        MOV  AX, Y                          NM   D&TP  VL
        ADD  X, AX                                ENDM
        ENDM
```

Първият ред на макроопределението съдържа директивата MACRO. Това е заглавен оператор. В него се декларира името на макроса и формалните му параметри разделени със запетаи. Наличието на параметри се обуславя от необходимостта макросът да се копира с модификации. Параметрите са тези, които влияят на модификациите. Формалните параметри се именуват произволно и са локални за тялото на макроса. Извън тялото на макроса същото име има смисъла, който

му е придаден в съответния обект. Тялото на макроса може да съдържа произволен брой редове, в състава на които могат да влизат формалните параметри. При това, ако редом с параметъра трябва да се укаже име или число, то следва да се използва макрооператор & (както е употребен във десния пример - макрос VAR). Коментарите в тялото на макроса не се копират, ако започват с двата символа (;;).

Макроопределението завършва с оператор ENDM.

Макроопределенията могат да се разполагат произволно в текста на програмата, но задължително до първото обръщение към тях.

11.3.2 Макрокоманди

Ако в дадено място от текста на програмата искаме макрогенераторът да постави тялото на даден макрос, в това място ние трябва да запишем обръщение към него. Обръщението се нарича макрокоманда и се записва така:

<име на макрос> <списък с фактически параметри>

Примери:

```
SUM  A,ES:B        или      SUM  A   ES:B
VAR  Z,W,?         или      VAR   Z  W,?
```

Макрокомандите приличат на обикновените команди и директиви, но имат и различия. Първо, вместо мнемоничен код или служебна дума, те се представят чрез своето име, което е синтезирано от програмиста. Второ, параметрите могат да се разделят както със запетая, така и с празен символ.

Като фактически параметър може да бъде указан всеки текст (в това число и празен), стига да е балансиран по кавички и ъглови скоби, както и не следва да сдържа извън кавичките и скобите запетаи, празни символи или символи точка и запетая.

При записване на параметрите на макрокомандите могат да се използват описаните вече макрооператори <>, ! и %. Например, ако във фактическия параметър се съдържат символи като запетая, точка и запетая или празен символ, то такъв параметър се огражда с ъглови скоби. Например:

```
SUM  <WORD  PTR  [SI]>, A
VAR  C  W  <1,2>
```

Списъкът от фактическите параметри трябва да съответства както по брой, така и по подредба на формалния списък. Ако обаче той съдържа повече елементи, излишните ще бъдат игнорирани. Ако пък съдържа по-малко на брой параметри – недостигащите ще се възприемат като празни текстове.

11.3.3 Макрозаместване и макроразширение

При среща в текста на програмата на макрокоманда макро генераторът извършва макрозаместване: намира описанието на указания чрез името макрос, копира неговото тяло, заменя в това тяло

формалните параметри с указаните фактически такива и записва получения текст (макроразширение) вместо макрокомандата в основния текст на програмата. В следващите примери е показано как една макрокоманда се преобразува в краен текст. В средата на рисунката е изобразена подмяната на формалните параметри с фактическите:

```
                    X → A,   Y → ES:B
SUM   A, ES:B    ──────────────────────────>    MOV   AX, ES:B
                                                ADD   A, AX

                NM → , TP →W ,  VL→1, 2
VAR   ,W,<1, 2>  ──────────────────────────>    DW   1, 2
```

11.3.4 Примери, използващи макроси

Пример №1. Описание на крупни операции във вид на макрос.

За целите на програмирането машинните операции, към които е ориентиран и езика Асемблер, представляват твърде ниско ниво. Това прави програмирането на Асемблер досадно. От ситуацията може частично да се излезе чрез окрупняване на операциите, използвайки представените до момента средства.

Като пример ще разгледаме многократно срещан условен преход по условието “по-малко”: if x<y then go to L. Този алгоритмичен преход се реализира с три команди. За да не ги изписваме всеки път, когато е нужно, има смисъл да ги декларираме в макрос. Ще уговорим, че числата, които ще се сравняват са със знак и с формат дума, а за име на макроса избираме IF_LESS. Ето текста на определението му:

```
IF_LESS     MACRO  X, Y, L
            MOV  AX, X
            CMP  AX, Y
            JL   L
            ENDM
```

Разполагайки с този макрос, ще го използваме в задачата за намиране на най-малкото от три дадени числа (DX=min(A,B,C)).

```
    MOV   DX, A            X→A,  Y→B,  L→M1            MOV   DX, A
    IF_LESS  A, B, M1   ───────────────────────>       MOV   AX, A
    MOV   DX, B                                        CMP   AX, B
M1: IF_LESS  DX, C, M2 ─┐                              JL    M1
    MOV   DX, C         │                              MOV   DX, B
M2: ....  ....  ....  ....  │                      M1: ....  ....  ....  ....
                        │  X→DX,  Y→C,  L→M2
                        └──────────────────────>       MOV   AX, DX
                                                       CMP   AX, C
                                                       JL    M2
                                                       MOV   DX, C
                                                   M2: ....  ....  ....  ....
```

Както се вижда от горната рисунка, в лявата колонка е макросът, който е съставил програмистът, а в дясната – командите, които реално се изпълняват. Забележете, че когато макрокомандата е отбелязана с етикет, в макроразширението този етикет стои в отделен ред. Вижда се, че използването на макроси наистина води до съкратени текстове и позволява да се съставят програми в термините на по-крупни операции. Създавайки множество макроси в сферата на дадено приложение, може да се твърди, че то е описано на "нов език".

Пример №2. Макроси и обръщения към процедури.

Нека имаме процедура NOD, изчисляваща най-големия общ делител на две числа: Z=NOD(X,Y). Нека е уговорено, че параметърът на процедурата X ще се предава чрез регистър AX, параметърът Y – чрез регистър BX, а резултатът Z ще се връща чрез регистър AX, унищожавайки при това първия параметър. Нека в крайна сметка е необходимо да се изчисли сумата CX=NOD(A,B)+NOD(C,D).

Фрагментът от програмата с това изчисление е следният:

```
MOV  AX, A
MOV  BX, B
CALL  NOD
MOV  CX, AX
MOV  AX, C
MOV  BX, D
CALL  NOD
ADD  CX, AX
```

Може да се забележи, че при всяко обръщение към процедурата се изписва една и съща група команди, свързана с предаването на параметрите, което естествено всеки път следва да се изпълнява по една и съща схема. За облекчаване на такива ситуации също може да се създаде макрос. Ето неговия текст:

```
CALL_NOD     MACRO  X, Y
             MOV  AX, X
             MOV  BX, Y
             CALL  NOD
             ENDM
```

тогава текстът в главната програма би се свел до следния:

```
CALL_NOD  A, B
   MOV  CX, AX
CALL_NOD  C, D
   ADD  CX, AX
```

Безспорно, полученото е по-прегледно и по-късо.

Ще отбележим, че по този принцип е реализиран входно-изходния обмен, който в тази книга вече частично беше авансово използван – например в лицето на макроса ININT (въвеждане на цяло десетично число). Създаването и използването на входно-изходните макроси е

представено в глава 13.

Пример №3. Макроси и блокове за повторение

При вход в процедура, както вече беше пояснено, се изпълняват така наречените “входни” действия. Ако програмата използва много на брой процедури, то текстът й се “натоварва” с тези спасителни операции, записвани във всяка процедура. Положението може да се облекчи чрез създаване на подходящ макрос.

Ето някои предварителни съображения. Макросът може да има произволен брой фактически параметри (различен брой имена на различни регистри). В същото време Асемблер може да определя макроси само с фиксиран брой формални параметри. Преодоляването на това ограничение на практика може да се постигне, ако макросът се определя с един формален параметър, но при обръщение към него (в макрокомандата) в ъглови скоби се указва списък от фактическите параметри, което синтактически се разбира като един параметър. В тялото на макроса в съответните команди се селектира (т.е. се откъсва) от списъка необходимия параметър.

Макросът трябва да копира в окончателния текст на програмата няколко еднотипни команди PUSH op. Ясно е, че тук може да се приложи блок за повторение.

След направените съображения, определението на макроса е следното:

```
SAVE    MACRO  REGS
            IRP  R, <REGS>
            PUSH   R
            ENDM
        ENDM
```

Нека е направено обръщението:

```
SAVE  <AX, SI, BP>
```

Преди всичко се копира тялото на макроса, в което е извършена подмяна на формалния параметър REGS с действителния, взет от обръщението:

```
IRP  R, <AX, SI, BP>
PUSH   R
ENDM
```

Ще поясним, че ъгловите скоби в обръщението към макроса се премахват при вземане на действителния параметър, но описанието на блока за повторение има свои скоби, които са показани в горния текст.

В крайна сметка блокът за повторение се разгъва от макро генератора и се получава следният краен текст:

```
PUSH   AX
PUSH   SI
PUSH   BP
```

11.3.5 Макроси и процедури

Необходимо е специално да поясним разликата между макрос и процедура. И макросът и процедурата са програмни структури, които се описват само веднъж в програмата. И в двата случая на използване се записват кратки обръщения към описанията им. Така от гледна точка на процеса на програмиране, особени разлики между макрос и процедура няма.

В транслираната програма процедурата отново остава единствена, докато макросът като такъв не се запазва, а вместо него в местата на обръщение към него са останали копия на неговото тяло.

Краткият извод, който може да се формулира, е, че с процедурите се печели памет, а с макросите се печели време. Това не дава основание за предпочитания. Все пак се е наложило общоприето разбиране, че за къси програмни фрагменти е добре да се използват макроси, а за по-дългите и по-сериозните фрагменти – процедури.

11.3.6 Определяне на макрос чрез макрос

Една процедура има право да се обръща към друга процедура. Аналогично при описанието на един макрос може да има обръщение към друг макрос. В частност, допуска се обръщение на макроса към самия себе си, т.е. разрешени са рекурсивни макроси.

Нека макросът ARR X,N е предназначен за описание на масив X от N елемента:

```
ARR  MACRO  X, N
  X  DB  N  DUP (?)
     ENDM
```

Използвайки този макрос, може да се определи макрос ARR2, който да описва два масива с еднакъв размер:

```
ARR2   MACRO  X1, X2, K
       ARR  X1,<K>
       ARR  X2,<K>
       ENDM
```

След това определение макрозаместването ще протече в два етапа:

```
ARR2  A, B, 20    ⇒    ARR  A, <20>     ⇒    A  DB  20  DUP (?)
                       ARR  B, <20>          B  DB  20  DUP (?)
```

Тук възниква въпросът: защо в тялото на макроса ARR2, при записване на обръщението към макроса ARR, вторият фактически параметър е поставен в ъглови скоби? Работата е в това, че ако по смисъл първият и вторият параметри на макроса ARR2 могат да бъдат само имена, то като трети параметър може да бъде указана достатъчно сложна конструкция, например: (ARR2 A,B,<25 MOD 10>). Така че, ако вместо записа <K> беше използван запис само на името K, то след първия етап от макрозамяната би се получила макрокоманда с 4 операнда: (ARR A, 25 MOD 10), а не с два. Напомняме, че при макро-

замяната ъгловите скоби на фактическите параметри се отстраняват и че в макрокомандите параметрите се отделят както със запетая, така и с празен символ. Ето защо при записа <K> ъгловите скоби указват един параметър.

Ще отбележим още, че в Асемблер се допуска влагане на макро-определения. Например:

```
ARR2    MACRO  X1, X2, K
 ARR    MACRO  X, N
        X   DB  N  DUP (?)
        ENDM
        ARR  X1, <K>
        ARR  X2, <K>
        ENDM
```

Тук обаче трябва да се отчита следното: макросът ARR, макар и определян във ARR2, не се локализира в ARR2, и към него можем да се обръщаме и от други места на програмата. Но Асемблер работи така, че описанието на вътрешния макрос той забелязва едва при първото обръщение към външния макрос. Ето защо обръщенията към вътрешния макрос ARR преди обръщения към външния ARR2, са невъзможни и водят до генериране на грешки.

11.3.7 Директива LOCAL

При използване на макроси възниква неприятен проблем с етикетите, с които могат да бъдат отбелязвани изречения в тялото на макросите. Пример:

```
      M  MACRO
         ....  ....  ....
   L:    ....  ....  ....
         ....  ....  ....
         ENDM
```

и нека в програмата да има две обръщения към този макрос. Тогава, след макрозаместването ще се получи следния текст:

```
....  ....  ....                 ....  ....  ....
M                ⇒         L:    ....  ....  ....
....  ....  ....                 ....  ....  ....

M                ⇒         L:    ....  ....  ....
....  ....  ....                 ....  ....  ....
```

Както се вижда, в текста се появяват два еднакви етикета, което е грешка. Това се получава, защото етикетът не е формален параметър и се копира винаги. За преодоляване на този проблем се предлага следното решение: след заглавния оператор на макроса (с директивата MACRO) следва да се укаже специалната директива на макро езика LOCAL, която има следния запис:

```
LOCAL   v1, v2, … , vk
```

където стойността на k е по-голяма или равна на 1. След директивата LOCAL следват имена, обикновено на етикети, които имена макро генераторът заменя в процеса на транслиране с други специални имена. Тези имена имат вида: ??xxxx.

Четирите младши знака на тези имена представляват 4-разрядно 16-чно число – ??0000, ??0001, ... , ??FFFF. Правилата за замяна на имената от директивата LOCAL са следните.

Макрогенераторът запомня номерът, който последно е използвал, например нека това е бил номер n. Когато макрогенераторът срещне обръщение към същия макрос (в който има тази директива), при копиране на тялото на макроса той подменя имената, изредени в директивата LOCAL, със специалните имена, като ги формира уникално според формулите: v1 $\rightarrow$??(n+1) ; v2 $\rightarrow$??(n+2) ; и т.н. Така в различните макроразширения на едни и същи макрос ще бъдат генерирани на едни и същи места уникални етикети.

Ще разгледаме конкретен пример: Ще опишем макрос, който изчислява остатъкът от делението на едно натурално число с друго, чрез операция изваждане:

```
MD  MACRO  R1,R2        ; r1:=(r1)mod(r2),  r1,r2 - регистри
    LOCAL   M, M1
M:  CMP  R1, R2         ; докато (r1)≥(r2) прави r1:=(r1)-(r2)
    JB   M1
    SUB   R1, R2
    JMP   M
M1: .... .... ....
    ENDM
```

Като предположим, че в последен път макрогенераторът е използвал за специалните имена, примерно номер 0016, то в следващите две обръщения към макроса MD ще бъдат създадени следните две копия:

```
MD  AX, BX    ⇒   ??0017  CMP  AX, BX
                          JB  ??0018
                          SUB  AX, BX
                          JMP  ??0017
                  ??0018:  .... .... ....
MD  CX, DX    ⇒   ??0019: CMP  CX, DX
                          JB  ??001A
                          SUB  CX, DX
                          JMP  ??0019
                  ??001A:  .... .... ....
```

Директивата LOCAL може да бъде записана няколкократно, но задължително веднага след заглавния оператор MACRO. Тази директива не се копира в макроразширението. Специалното в имена, които тя генерира е само видът им, иначе те по нищо не се различават от останалите имена, които програмистът съставя.

11.3.8 Директива EXITM

Директивата EXITM няма операнди. Тя се използва само в макроопределения и блокове за повторение, т.е. само в конструкции на макроезика, завършващи с директивата ENDM. Срещайки директивата EXITM, макрогенераторът завършва обработката на най-близкото мароопределение или блок за повторение. Например, когато е направено макроопределението:

```
A  MACRO  K
       REPT  K
       DB  0
       EXITM
       ENDM
   DW  ?
   ENDM
```

обръщението чрез макрокомандата (A 5) ще генерира следния текст:

```
DB  0
DW  ?
```

Тук макрогенераторът, създавайки първото копие на тялото на блока за повторение REPT, записва в макроразширението декларацията (DB 0), след което, срещайки директивата EXITM, завършва работата си по този блок, но все още остава в тялото на макроса, т.е. прескача най-близката директива ENDM и довършва тялото на макроса като записва в макроразширението втората декларация (DW ?). По-опитните читатели вероятно усещат недоиз-казаното тук. То обаче изисква запознаване с условните директиви, на които ще се спрем по-късно.

11.3.9 Преопределяне и отманя на макроси

За разлика от други обекти в програмите на Асемблер, макросите могат да бъдат преопределяни и даже унищожени.

Ако в текста на една програма е определен макрос с име, с което преди това е бил определен друг макрос, то по-рано определеният с това име макрос се приема за унищожен, а новия – действащ.

Например:

```
A  MACRO  X
   INC  X
   ENDM
       A  CX          ⇒      INC  CX
A  MACRO  Y, Z
   CMP  Y, 0
   JE  Z
   ENDM
       A  BH, EQ      ⇒      CMP  BH, 0
                             JE   EQ
```

Даден макрос може да бъде унищожен и без да се определя нов с неговото име. За целта се използва следната директива:

PURGE <име на макрос> {, <име на макрос>}

След тази директива всички макроси, чиито имена са в списъка й, се считат за несъществуващи.

11.4 Условно асемблиране

Условното асемблиране е възможност, която улеснява отработването на програмите, поддържайки в изходния текст няколко варианта. Участъкът от програмата, който се засяга от условното асемблиране, трябва да се записва в условен блок:

```
<IF – директива>              <IF – директива>
    <фрагмент-1>                  <фрагмент-1>
  ELSE                          ENDIF
    <фрагмент-2>
ENDIF
```

Директивите ELSE и ENDIF трябва да се записват на отделни редове. В клона истина (фрагмент-1) и в клона лъжа (фрагмент-2) на условния блок може да има произволен брой редове, в частност може да съдържат и други IF-блокове.

В IF-директивата се указва някакво условие, което макрогенераторът проверява. Ако условието е изпълнено, в крайния текст на програмата макрогенераторът оставя само фрагмент-1. Фрагмент-2 в този случай не присъства в крайния текст. Ако условието не е изпълнено резултатът е аналогичен, но по отношение на фрагмент-2.

Тъй като условието в IF-директивата се проверява по време на макрогенерацията, то в него не следва да се съдържат обръщения към величини, чиито стойности ще станат известни при изпълнение на програмата. Условието трябва да бъде такова, че макрогенераторът да е в състояние да го изчисли веднага.

В макроезика има не малко на брой разновидности на директивата IF.

11.4.1 Директиви IF и IFE

Тези директиви имат следния вид:

IF <константен израз>
IFE <константен израз>

Срещайки някоя от тези директиви, макрогенераторът изчислява указания в тях константен израз. Условието за преход и в двата случая изисква утвърдителен отговор.

Ще разгледаме следния пример: нека сме в процес на отработване на дадена програма, като за целта в някои места желаем разпечатка на контролни стойности. След настройката на програмата тези разпечатки, разбира се, отстраняваме. За съжаление грешките са значително скрити и е възможно този процес да се повтори, потрети и т.н.

Вмъкването и премахването на контролните разпечатки може да продължи дълго.

В описаната ситуация е удобно да се възползваме от условното асемблиране. В текста на програмата постоянно съхраняваме контролните извеждания, но пред тях поставяме условие, че командата за контролния печат следва да остава в окончателния текст на програмата само при отработка.

Постъпва се по следния начин. Уговаря се, че режимът за отработка ще бъде указван с константата DEBUG, която описваме в началото на текста на програмата като й присвояваме стойност 1. Например:

```
DEBUG  EQU  1
```

Тогава фрагментът от програмата с контролен печат на променливата X например, следва да бъде записан както е показано в лявата колонка, а в окончателния текст същият текст ще изглежда както е показано в дясната колонка:

```
                                                  ....  ....  ....
....  ....  ....          DEBUG  <> 0    →        MOV   X, AX
MOV   X, AX                                       OUTINT   X
  IF  DEBUG                                       MOV   BX, 0
    OUTINT   X                                    ....  ....  ....
  ENDIF
MOV   BX, 0                                       ....  ....  ....
....  ....  ....          DEBUG = 0      →        MOV   X, AX
                                                  MOV   BX, 0
                                                  ....  ....  ....
```

При такова построяване на текста е достатъчно преди контролното изпълнение на програмата да променим само един ред – описанието на константата DEBUG, за да може макрогенраторът да генерира различен и подходящ за нас краен текст.

Ще разгледаме пример, в който директивите за условно асемблиране се използват не самостоятелно, а за описание на макроси, което е основен случай за използване на тези средства.

Ще опишем като макрос (SHIFT X,N) операция изместване на X на N разряда надясно, при условие, че X е скаларна променлива, а N – явно зададено положително число, и при условието, че макро разширението ще съдържа минималния възможен брой команди.

Последното условие означава, че при макрокоманда (SHIFT X,N), ако N=1 ще бъде генериране една команда SHR X,1, а при N>1 ще бъдат генерирани две команди (MOV CL,N) и (SHR X,CL). Тъй като тук в макроразширението трябва да се включват различни фрагменти (описани в макроса), то е ясно, че той следва да бъде описан със средства за условно асемблиране:

```
SHIFT    MACRO   X, N
         IFE  N-1                  ;; (n-1)=0?
```

```
        SHR  X, 1
    ELSE                    ;; n>1
        MOV  CL, N
        SHR  X, CL
    ENDIF
    ENDM
```

Ако в една програма се укаже обръщението (SHIFT A, 5), то макро заместването ще бъде в два етапа, както следва:

```
               X→A, N→5   IFE  5-1
SHIFT  A, 5                  SHR  A, 1    5-1=0? → не   MOV  CL, 5
                          ELSE                          SHR  A, CL
                             MOV  CL, 5
                             SHR  A, CL
                          ENDIF
```

11.4.2 Оператори за отношение. Логически оператори

Смисълът на константния израз в директивите IF и IFE трябва да бъде логически, тъй като те са всъщност въпросителни. Тъй като до момента ние записвахме само числени изрази, тук ще разгледаме как в Асемблер се записват логически изрази. Последните се възприемат като частен случай. Въпросът за представяне на логическите константи ние вече изяснихме ("да" = 0FFFFh , "не" = 0000h).

В Асемблер има 6 оператора за отношение:

```
<израз>  EQ  <израз>          ; equal
<израз>  NE  <израз>          ; not equal
<израз>  LT  <израз>          ; less than
<израз>  LE  <израз>          ; less or equal
<израз>  GT  <израз>          ; greater than
<израз>  GE  <израз>          ; greater or equal
```

Двата операнда на отношението трябва да бъдат константни изрази или адресни изрази, чиито стойности да представляват адреси от един и същи сегмент на ОП. Проверяваното отношение е или вярно или невярно. Например, ако N=5, то:

N-1 LT 5   → "не" ,   N+1 LT 5   → "да"

Следва да отбележим обаче, че действията на тези оператори са свързани по особен начин с аритметиката на процесора. Операторите EQ и NE интерпретират своите операнди като 16 битови числа със знак, представени в допълнителен код, ето защо изразът (-2 EQ 0FFFFh) се оказва истина. Останалите оператори за отношение правилно отчитат знаците на своите операнди. Например отношението (-1 LT 0FFFFh) е вярно. Изключение прави случаят, когато сравняваните операнди са равни като 16 битови знакови в допълнителен код (например отношението (-1 LT 0FFFFh) се приема за лъжа. Ето защо не се препоръчва използването на отрицателни числа в тези оператори, или

правете това при отлично разбиране на цифровата аритметика на процесора, с който работите.

Като пример за използване на оператори за отношение ще разгледаме макрос (SET0 X), който описва присвояването X:=0, при условие, че името X може да има различни формати – байт, дума или двойна дума.

```
SET0   MACRO  X
       IF  TYPE  X  EQ  DWORD
           MOV   WORD  PTR  X, 0
           MOV   WORD  PTR  X+2, 0
       ELSE
           MOV   X, 0              ;; останалите 2 формата
;; се разграничават автоматично от декларацията на името X
       ENDIF
       ENDM
```

Логическите стойности и отношенията могат да се обединяват в по-сложни логически изрази с помощта на следните логически оператори:

```
NOT  <константен израз>
<константен израз>  AND  <константен израз>
<константен израз>  OR  <константен израз>
<константен израз>  XOR  <константен израз>
```

Като пример ще разгледаме описанието на макроса (SHRG B, N). Макросът измества стойността на еднобайтовата променлива вдясно на N разряда (N е положително число). Като подминем изместването при N=1, което може да се реализира с една команда, ще отбележим, че при N=0 изместването е ненужно (макроразширението следва да е празно), а при N>7, резултатът е един и същ, независимо колко е N – празна клетка (B=0), т.е. той е отнапред известен. Ето текста:

```
SHRG    MACRO  B, N
        IF (N GT 0) AND (N LT 0)        ;;  0<N<8  ?
            MOV  CL, N
            SHR  B, CL
        ELSE
            IF  N GE 8                  ;;  N>=8  ?
                MOV  B, 0
            ENDIF
        ENDIF
        ENDM
```

Какви са действията на логическите оператори? Според смисъла на техните операнди – логически, но в Асемблер има константни изрази, чиито стойности са 16 битови думи. Например:

```
SCALE   EQU  1010b
        MOV   AX, SCALE  AND  11b
        AND   AX, SCALE  XOR  11b
```

С други думи, тези оператори се изпълняват аналогично на едноименните машинни команди. Но не следва тези оператори да се бъркат с машинните команди: стойностите на операторите се изчисляват още по време на транслирането на програмата (в машинната програма те вече не съществуват), а машинните команди се изпълняват след стартиране на програмата.

11.4.3 Директиви IFIDN, IFDIF, IFB и IFNB

Разглеждаме директивите:

```
IFIDN  <t1>,<t2>
IFDIF  <t1>,<t2>
```

в които с t1 и t2 са означени произволни текстове, затворени в ъглови скоби. Тези текстове се сравняват символ по символ. Директивата IFIDN проверява дали те са еднакви (идентични *identical*), а директивата IFDIF проверява дали текстовете са различни (*different*). Следва да се отбележи, че при сравнението малките и главните букви не се приемат за еднакви символи. Примери:

```
IFIDN  <a+b>,<a+b>        ; “да”
IFIDN  <a+b>,<a>          ; “не”
IFIDN  <a+b>,<a+B>        ; “не”
```

И двете директиви има смисъл да бъдат използвани в тяло на макрос или на блок за повторение, указвайки в тях като формални параметри някакви текстове. При макрозаместването тези параметри ще се заменят на действителните, като с това може да се проверява дали при обръщението към макроса са зададени фактически параметри от определен вид.

Като пример ще дадем описанието на макрос (MM R1,R2,T), в който с R1 и R2 са означени имената на еднобайтови регистри, съдържащи числа със знак, а с T е означено някое от имената MAX или MIN. Описваната операция се изразява така: R1:=T((R1),(R2)), т.е. в R1 винаги се записва стойността на T-функцията от съдържанията на указаните регистри.

Преди всичко ще отбележим, че този макрос следва да генерира непразно макроразширение, само ако R1 и R2 са различни регистри, т.е. при обръщението (MM AL, AL, MAX) съдържанието на регистър AL не следва да се променя. Така възниква задачата за проверка дали указаните регистри са различни, т.е. дали първите два действителни параметъра в обръщението са различни. За целта е подходяща директивата (IFDIF <R1>,<R2>), в която при макрозаместването формалните параметри R1,R2 ще бъдат заместени с имената на указаните в обръщението регистри. Така за горния пример след заместването тази директива ще приеме вида (IFDIF <AL>,<AL>). Така ще бъдат сравнени имената като последователности от символи.

За да се определи какво точно е необходимо да се изчислява – максимум или минимум, трябва да се провери третият фактически

параметър. Това може да направи директивата (IFIDN <T>,<MAX>). След макрозаместването формалният параметър T ще бъде заместен с името в обръщението към макроса и така ще бъде сравнено символ по символ с името MAX.

Последното съображение се отнася до алгоритъма – изчисленията на максимума и на минимума се различават само по една команда – тази на условния преход:

```
; R1:=max(R1, R2)              ; R1:=max(R1, R2)
    CMP   R1, R2                   CMP   R1, R2
    JGE   L                        JLE   L
    MOV   R1, R2                   MOV   R1, R2
 L: ····  ····  ····            L: ····  ····  ····
```

по тази причина няма смисъл в макроопределението да се описва два пъти практически еднакви фрагменти. Групата команди на изчислението ще бъде записана само веднъж, като вместо командата за условен преход ще бъде поставен IF-блок, който в зависимост от параметъра T ще избере необходимата за макроразширението команда. Така, отчитайки всичко казано по-горе се стига до текста:

```
MM  MACRO  R1, R2, T
    LOCAL   L
    IFDIF   <R1>, <R2>           ;;  R1, R2  различни?
       CMP  R1, R2
          IFIDN   <T>, <MAX>     ;;  T=MAX
             JGE   L             ;;  “да”, команда JGE  L
          ELSE
             JLE   L             ;;  “не”, команда JLE  L
          ENDIF
       MOV   R1, R2
  L:      ····  ····  ····
    ENDIF
    ENDM
```

Ще разгледаме макрозаместванията за обръщението

(MM AL, BH, MIN)

след като в тялото на макроса формалните параметри са заменени на действителните, а локалният етикет L – на специалното име ??0105:

```
   IFDIF  <AL>, <BH>
   CMP  AL, BH                  CMP  AL, BH                CMP  AL, BH
      IFIDN  <MIN>, <MAX>       IFIDN  <MIN>, <MAX>        JLE   ??0105
         JGE   ??0105             JGE   ??0105             MOV  AL, BH
      ELSE                 ⇒    ELSE                  ⇒  ??0105: ····  ····
         JLE   ??0105             JLE   ??0105
      ENDIF                     ENDIF
    MOV  AL, BH                 MOV  AL, BH
 ??0105:  ····  ····  ····    ??0105:  ····  ····  ····
   ENDIF
```

Както вече беше отбелязано, при сравнение на текстовете в директивите IFIDN и IFDIF големите и малките букви не се приемат за еднакви. Това е неблагоприятната особеност, имайки предвид, че езикът позволява по-голяма свобода при именуването. Решение на този проблем естествено съществува, но ние няма да го разглеждаме подробно. Ще отбележим само, че то се основава на IRP-блок, с помощта на който могат да се проверят всички възможни записи на дадено име в комбинация от малки и големи букви, например за регистър AL това са записите: AL, al, aL, Al.

Следва да разгледаме останалите две IF-директиви:

```
IFB  <t>             ; (if blank)
IFNB  <t>            ; (if not blank)
```

Тези директиви представляват разновидност на предидущите, когато вторият параметър е празен текст. Първата директива проверява дали текстът t е празен, а втората – дали е различен от празен.

Директивите се използват в макросите за проверка дали е зададен или не фактическият параметър в обръщението. Например, дадено е макроопределението:

```
DEFMACRO  X, V
    IFB  <V>
        X  DB  ?
    ELSE
        X  DB  V
    ENDIF
    ENDM
```

с което променливата X получава начална стойност V или не:

```
DEF  A, 6   →   A  DB  6
DEF  B      →   B  DB  ?
```

Съществуват и други IF-директиви, които ние няма да разгледаме, предвид рядката им употреба.

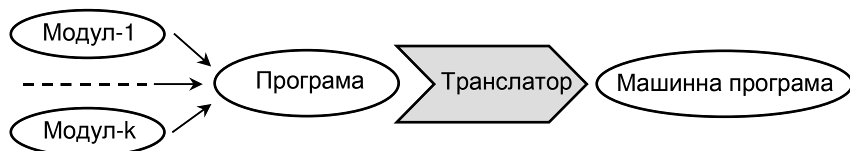
Г Л А В А 12

М Н О Г О М О Д У Л Н И П Р О Г Р А М И

С термина “модул” е прието да се означава част от програмата, която има предназначението да решава определена подзадача, която е съставена отделно, транслирана и отработена е самостоятелно и съществува самостоятелно под формата на файл. Частен случай на това понятие са процедурите.

В крайна сметка работният вид на програмата се “сглобява” от такива части, което, вероятно читателят разбира, е удобно от всяка гледна точка.

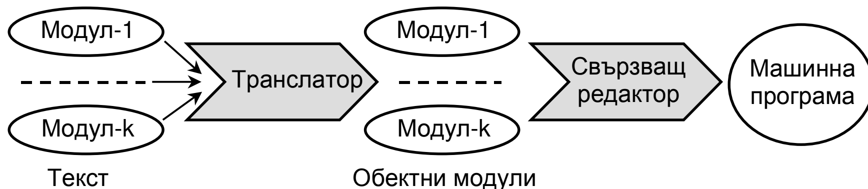
Съществуват два основни варианта за обединяване на модулите. В първия вариант модулите се обединяват в единна програма преди да започне транслирането на програмата, както е илюстрирано на следната рисунката:



Именно този вариант обикновено се подразбира при използване на процедури в структурата на програмата. Процедурите и главната програма се програмират, транслират, отработват и експериментират самостоятелно. Едва в последствие се сглобява и настройва работният вариант на приложението. Аналогично е обединяването с помощта на директивата INCLUDE, по силата на която съдържанието на указания текстов файл се вмъква в програмата преди нейното транслиране.

Споменатият способ за обединяване на модули е основен, когато те са малко на брой. Ако обаче модулите са много се проявява един недостатък, чиято същност е следната: ако в един от модулите се открие грешка, то след нейното отстраняване цялата програма, заедно с всички модули, трябва да се транслира наново. Ако тази процедура се наложи да се повтори няколко пъти, ще има голяма загуба на време.

Според втория вариант, модулите се транслират независимо един от друг и чак след това се обединяват в единната програма. Това поведение е илюстрирано на следващата рисунка:



Обектен модул е прието да се нарича транслиран модул в машинен код. Обединяването на такива обектни модули в единна машинна програма извършва специална програма наричана свързващ редактор (*linker*).

В този вариант редактирането на грешен модул изисква след корекциите само неговото повторно транслиране, което е значително по-удобно и по-бързо. Този вариант ще имаме предвид в тази глава тук.

12.1 Работа със системата MASM

Ще бъдат разгледани само най-необходимите правила за работа със системата MASM. Детайлното описание на системата може да бъде намерено например в [5].

Как се съхранява многомодулна програма

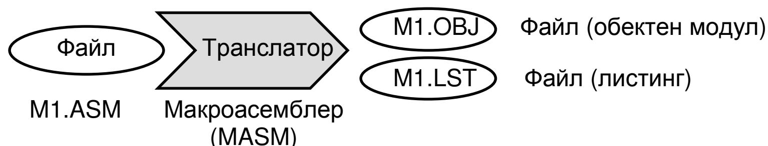
Нека сме разделили програмата на няколко модула, които възнамеряваме да транслираме поотделно. Това изисква създаването на няколко файла с асемблерските текстове са с разширение .ASM. Това не е задължително, но е желателно, например: F1.ASM, M5.ASM.

Транслиране на модулите

Тъй като транслирането на програмите изисква използване на служебни програми, работещи под управлението на операционната система, то искането за такъв вид действие се изразява от командния ред, чрез командата:

MASM   M1.ASM, M1.OBJ, M1.LST ;

(всички запетаи, точки и точка със запетая са задължителни).



Макроасемблер транслира модул M1 от указания текстов файл M1.ASM и създава два нови файла със същото име: M1.OBJ – обектен файл с машинната програма, M1.LST – файл с листинга на програмата и направените от транслатора и ОС съобщения. Листингът следва да съдържа изходния текст на Асемблер, както и машинния вид на програмата. Това е необходимо за локализиране, анализиране и поправяне на грешките.

Въпреки, че имената на двата последни файла могат да бъдат произволни, тук се препоръчва те да съвпадат с името на текстовия файл, което облекчава ориентацията в набора от файлове за програмата като цяло.

Ако при транслация бъдат открити грешки, обектен файл не се създава, но листинг ще бъде създаден задължително.

Обединение (свързване) на модулите

Когато бъдат окончателно транслирани (без грешки) всички модули (части) на програмата, те трябва да бъдат обединени в единна машинна програма. Това се постига със специална системна програма от командния ред чрез командата:

LINK M1.OBJ+M2.OBJ+M3.OBJ+…+Mk.OBJ, M.EXE

Първият параметър представлява списък от всички влизащи в програмата обектни модули (във вид на файлове), между имената на които е поставен знак плюс. Вторият параметър е името на изпълнимия файл на програмата с разширение .EXE. Ще отбележим, че, ако всички файлове изредени в първия параметър имат разширението .OBJ, то може да бъде изпуснато. Така горната команда може да се запише във вида:

LINK M1+M2+M3+…+Mk, M.EXE

Името на изпълнимия файл може да бъде произволно.

Стартиране на програмата (изпълнение)

За да стартираме дадена програма е необходимо да запишем команда от командния ред. Командата за изпълнение е RUN, но на командния ред може да бъде записано само името на изпълнимия файл на програмата, например M.EXE. Тъй като всички изпълними файлове (приложения) имат едно и също разширение (.EXE), то при стартиране то може да бъде изпуснато – достатъчно е само името (M).

За да се изпълни всяка програма, тя трябва да бъде заредена в оперативната памет. Вземането на машинния текст на програмата от файла, който се намира на твърдия диск, и записването му в свободна област на оперативната памет, е задача на друга специална системна програма. Изпълнявайки това зареждане, тя всеки път, предава управлението по нейния начален адрес, който съдържа първата изпълнима машинна команда на програмата.

Изпълнението на приложната програма завършва с изпълнение на последната й команда (FINISH). По същество в този момент заеманата от приложната програма област в оперативната памет се счита за свободна и управлението получава програма от операционната система. Приложната програма може да бъде заредена за изпълнение неограничен брой пъти, но вероятно това ще бъде в различни области на оперативната памет, оказали се свободни в момента.

Поправка на грешки

Ако при изпълнение на програмата е намерена грешка, то следва да определим в кой от модулите тя се локализира и да поправим и транслираме отново само този модул. Свързването и повторното изпълнение е аналогично на описаното по-горе.

Транслация и изпълнение на едномодулна програма

Тривиалният случай е често срещан – програмата представлява един единствен модул. За да бъде подведен и този частен случай към общия е прието свързващото редактиране на се изпълнява и в този случай. Във връзка с това с едномодулна програма например AA.ASM работим както следва:

```
MASM   AA.ASM,  AA.OBJ,  AA.LST        ; транслиране
LINK   AA.OBJ,  AA.EXE                 ; свързване
AA.EXE                                 ; изпълнение
```

12.2 Модули. Външни и общи имена

Многомодулните програмни приложения създават специфични проблеми, ето защо се нуждаят от специално внимание.

12.2.1 Структура на модулите. Локализация на имената

Всеки модул се записва като самостоятелна програма, т.е. представлява последователност от команди (изречения), която завършва с директива END:

```
<изречение>
<изречение>
.... .... .... ....
<изречение>
END   [<входна точка>]
```

Сред множеството модули на програмата винаги се откроява един, който наричаме *главен*. Този модул трябва да се изпълни първи, с неговата първа команда стартира програмата като цяло. Едва в последствие, от неговата среда, управлението може да се разпространи върху останалите модули. Синтактически главният модул се различава от останалите, по това, че в неговата директива END е посочена входната точка на програмата (етикетът на програмата). Ако входната точка е указана в няколко модула, като такава се приема онази, която свързващият редактор среща за първи път.

Всички имена, описани в даден модул се локализират в него. Това означава, че в различните модули програмата може да използва едни и същи имена, без това да я задължава по отношение на смисълът им или стойностите им, присвоени в други модули.

12.2.2 Външни и общи имена. Директиви EXTRN и PUBLIC

Въпреки, че модулите се описват достатъчно независимо един от друг, малко или много те са свързани помежду си. Те си предават управлението, обменят данни и резултати. Това взаимодействие поражда своите проблеми.

Нека в програмата има два модула M1 и M2. Нека в M2 е описана процедурата P, към която ще се обръща модул M1, и нека модул M2, от своя страна, да използва стойността на променливата X и на константата K, описани в модул M1:

```
; модул M1                      ; модул M2
    .... .... ....              .... .... .... ....
X  DB  ?                        P  PROC  FAR
K  EQU  100                         .... .... .... ....
.... .... .... ....                 MOV  X, 0
CALL  P                             MOV  AX, K
.... .... .... ....                 .... .... .... ....
```

При независимо транслиране на модулите Асемблер среща следния проблем: имената X и K принадлежат на модул M1, където са описани. Когато се транслира модул M2, Асемблер не знае за модул M1 и не познава неговите имена. Ето защо, срещайки в M1 имената X и K, той ще генерира грешка. Тази грешка може да бъде предотвратена, ако има средство, с което на се съобщи на транслатора, че срещаните имена са външни за обработвания модул, т.е. те са описани в друг модул. Такова съобщение може да бъде предадено с помощта на директивата EXTRN (*external*), имаща конструкцията:

EXTRN <име>:<тип>, ... , <име>:<тип>

в която <тип> е BYTE, WORD, DWORD, ABS, NEAR или FAR. (*забележка* – ABS е стандартна константа в Асемблер, със стойност 0). Тази директива може да се записва в произволно място, произволен брой пъти. В нашия пример, в модул M2 следва да се запише:

EXTRN X:BYTE, K:ABS

Типовете на имената са необходими за определяне на размера. Типовете BYTE, WORD, DWORD са за променливи, ABS за константни имена, а NEAR и FAR са за етикети и имена на процедури. Например директивата

EXTRN P:FAR

която, както следва от казаното, трябва да се запише в модул M1, съобщава, че P е етикет или име на процедура, описано в друг модул и затова обръщението е далечно. От своя страна в модул M2 процедурата P е описана с директивата FAR.

Отговорността за съответствието между типа на външното име, указано в директивата EXTRN, и типа, с който е описано то в "своя" модул, носи програмистът. Направеното до тук за безпроблемна работа на двата модула обаче не е достатъчно. Ако в модул M2 е обявено, че имената X и K са външни, то в модул M1, където те са описани, трябва да се запише директивата PUBLIC, с която обявяваме имената като общи, достъпни за всички:

PUBLIC <име>, ... , <име>

Тази директива може да се записва в произволно място, произволен брой пъти. Така, за нашия пример, в модул M1 трябва да запишем:

PUBLIC X, K

Имената, изброени в тази директива, задължително трябва да бъдат описани в същия модул.

Тук възниква въпросът – не е ли достатъчна само декларацията в M2, че имената X и K са външни (EXTRN). Оказва се, че не е достатъчно. Работата е в това, че модулите M1 и M2 се транслират по отделно, а се обединяват когато са преведени на машинен език. Но в машинния вариант на модул M1 няма да останат никакви имена, и когато при обединяването на модулите, в модул M2 трябва да бъдат заменени имената X и K на съответните адреси и стойности, това няма да бъде възможно – в модул M1 за това няма да има информация. Ето защо, за да се съхрани тази информация при транслиране на модул M1 за имената X и K, следва да присъства директивата PUBLIC. Така в резултат от указанията на тази директива при транслиране Асемблер ще запази необходимата информация за обявените имена, която Свързващият редактор ще използва при обединяване на модулите. С други думи директивите PUBLIC и EXTRN са като неразделна двойка. При това нито едно име не бива да бъде обявявано като общо в няколко модула, в противен случай ще настъпи объркване.

В крайна сметка модулите от нашия пример следва да бъдат изписани по следния начин:

```
; модул M1
    EXTRN   P:FAR
    PUBLIC  X, K
    ....  ....  ....  ....
    X   DB  ?
    K   EQU  100
    ....  ....  ....  ....
    CALL  P
    ....  ....  ....  ....
```

```
; модул M2
    EXTRN  X:BYTE, K:ABS
    PUBLIC  P
    ....  ....  ....  ....
P   PROC  FAR
    ....  ....  ....  ....
    MOV  X, 0
    MOV  AX, K
    ....  ....  ....  ....
```

12.2.3 Сегментиране на външни имена

Следващият проблем в многомодулните програми е свързан със сегментирането на външните имена. Обърнете внимание на директивата EXTRN, че тя не указва по кой сегментен регистър се адресира външното име. При транслирането Асемблер трябва да знае какъв префикс да сложи към адреса, например, ако името X трябва да бъде сегментирано в допълнителния сегмент, то командата MOV X, 0 трябва да се възприема като MOV ES:X, 0. Ако обаче името X се сегментира в данновия сегмент, то префиксът следва да бъде DS:, който се има предвид по подразбиране за команда MOV. Ако за външното име е указан явно префиксът, то проблем не съществува, но обикновено имената са без префикс, а тогава възниква проблемът със сегментирането на външните имена. В Асемблер този проблем се решава по следния начин:

1. Външните имена на константи (от тип ABS) не се сегментират ;

2. Външните етикети и имена на процедури (от тип NEAR или FAR) винаги се сегментират в кодовия сегмент, т.е. по регистър CS,

при което за дългите етикети и процедури (тип FAR) винаги се формират далечни преходи, а за близките (тип NEAR) – близки преходи. Например:

```
EXTRN   L:FAR,  M:NEAR
.... .... .... ....
CALL   L                ; ≡ CALL  FAR  PTR  L
CALL   M                ; ≡ CALL  NEAR  PTR  M
```

3. За имена на външни променливи (BYTE, WORD, DWORD) са в сила следните правила:

- Ако директивата EXTRN с някакво име е разположена извън всеки програмен сегмент, то всяка команда с това външно име се транслира без префикс, т.е. приема се сегментният регистър по подразбиране за съответната команда ;
- Ако директивата EXTRN с някакво име е разположена в програмния сегмент, то външното име се сегментира по подразбиране спрямо този сегментен регистър, спрямо който се сегментират всички останали имена в този сегмент.

Пример:

```
    EXTRN   X:WORD
A   SEGMENT
    EXTRN   Y:WORD
    .... .... .... ....
A   ENDS

B   SEGMENT
    EXTRN   Z:WORD
    .... .... .... ....
B   ENDS

    .... .... .... ....
    ASSUME   ES:A,  DS:B
    .... .... .... ....
    INC   X                 ; ≡ INC  DS:X
    INC   X[BP]             ; ≡ INC  SS:X[BP]
    INC   Y                 ; ≡ INC  ES:Y
    INC   Y[BP]             ; ≡ INC  ES:Y[BP]
    INC   Z[BP]             ; ≡ INC  DS:Z[BP]
```

По този начин мястото за директивата EXTRN не е съвсем произволно, тъй като от това място зависи как ще бъдат сегментирани имената от нейния списък.

Що се отнася до директивата PUBLIC, то тя може да бъде поставяна произволно – обикновено в началото, за да се видят веднага имената, които ще се експортират от дадения модул и са общи.

12.2.4 Достъп до външните имена

Указването на сегментния регистър все още е половината работа. Това позволява само правилното транслиране на командата, съдържаща външно име и все още не означава правилното й изпълнение. За правилното изпълнение е необходимо в сегментните регистри да се намират необходимите стойности. Ето пример:

```
; модул M1                      ; модул M2
    A  SEGMENT                       PUBLIC  X
    EXTRN  X:WORD                    B  SEGMENT
X   DW  0                        X   DW  ?
A   ENDS                         B   ENDS
    .... .... .... ....              .... .... .... ....
    ASSUME  DS:A
    .... .... .... ....
```

Предполагаме, че се намираме в модул M1 по време на изпълнение на програмата, и че регистър DS вече съдържа началния адрес на сегмента, в който е локализирана променливата A. Искаме да изпълним прехвърлянето на Y в X (X:=Y). За целта като че ли следва да се изпълнят следните две команди:

```
MOV  AX, Y
MOV  X, AX
```

Това предположение обаче не правилно. Първата от тези две команди ще се изпълни правилно: името Y всъщност представлява адресната двойка DS:Y, която правилно адресира името Y в сегмента A. Следващата команда обаче няма да се изпълни правилно, тъй като името X в този модул е обявено за външно и адресната двойка по подразбиране DS:X не адресира правилно тази променлива – тя е описана в модул M2 в сегмент B. Излиза, че ако искаме правилното й изпълнение, то в сегментния регистър DS трябва предварително да запишем сегментния адрес B (DS:=B).

Въпросът е: как ще направим това? Името на сегмент B не е описано в модул M1 и не се явява за него външно. Ще припомним, че зад имената стоят адресите, които при разпределението са им присвоени. Ще отбележим още, че имената на сегментите не могат да бъдат обявявани в директивата EXTRN, ето защо използването на името B в модул M1 е забранено. Но и не е необходимо да бъде явно указвано, ако си припомним оператор SEG. Операторът (SEG X) като функция всъщност връща (ни носи) сегментния адрес на сегмента, в който е разположена променливата X, т.е. SEG X=B. Вероятно читателят вече съобразява, че нашето решение е в присвояването:

DS:=SEG X. Отчитайки казаното, и това, че преди всичко следва да се спаси (съхрани) текущото съдържание (A) на необходимия регистър DS, прехвърлянето X:=Y трябва да бъде запрограмирано както следва:

```
MOV  AX, Y
PUSH  DS            ; спасяване на текущото съдържание
MOV  BX, SEG X
MOV  DS, BX
MOV  X, AX
POP  DS             ; възстановяване на старото съдържание
```

Както се вижда, полученият запис не е малък, при това той следва да се изпълнява винаги в подобни случаи при обръщение към външни имена. Това се получава защото ние се опитахме да работим с един сегментен регистър за различни сегменти. Текстът значително би се минимизирал, ако съобразим да работим с отделни сегментни регистри за отделните сегменти, например, с регистър DS за сегмент A и регистър ES за сегмент B. Ето реализацията на тази идея:

```
MOV  AX, SEG  X
MOV  ES, AX
.... .... .... ....
MOV  AX, Y
MOV  ES:X, AX
```

Трябва да отбележим, че в този случай няма необходимост специално да записваме директивата (EXTRN X:WORD) в текста на сегмент A (модул M1), тъй като полза от нея няма. Тази директива може да се изнесе извън програмните сегменти, например в самото начало на текста на модул M1, където тя по-лесно ще бъде забелязана. Така обикновено решават проблема с достъпа до външните променливи.

Достъпът до външни етикети или процедури се осигурява по-лесно. Нека разгледаме примера:

```
; модул  M1                      ; модул M2
        EXTRN  L:FAR                     PUBLIC
        .... .... .... ....              C  SEGMENT
        JMP   L                             .... .... .... ....
        .... .... .... ....              L: .... .... .... ....
```

Както вече беше казано, външните етикети се сегментират винаги по съдържанието на регистър CS, при това ако са обявени като далечни, за тях се формират далечни преходи. Затова в нашия пример командата (JMP L) ще се възприема като (JMP FAR PTR L), чийто машинен еквивалент е: (JMP seg(L):offset(L)).

По тази машинна команда в регистър CS се записва seg(L)=C, а в регистър IP се записва offset(L). Това означава “преход в сегмент C, към място с етикет L”.

12.2.5 Пример за многомодулна програма

Задачата, с която ще илюстрираме темата е: трябва да се въведе текст с не повече от 100 символа, който завършва с точка. След въвеждането текстът трябва да се изведе в обратен ред, при което всички големи латински букви да бъдат заменени на малки такива.

Обикновено във вид на модул описват набор от процедури и данни, използвани от други модули. Така е постъпено и тук, за което програмата е структурирана в два модула – главен и помощен. В помощния модул ще опишем променливата EOT, чиято стойност е символ, маркиращ края на въвеждания текст, както и процедура LOWLAT, задачата на която ще бъде да заменя големите латински букви с малки. Главният модул ще описва въвеждането на текста, записването му в някакъв масив в обратен ред, заменяйки при това с помощта на функцията LAWLAT при необходимост текущия символ, а в края ще разпечатва получения масив.

```
; помощен модул
            PUBLIC  EOT,  LOWLAT

      D1    SEGMENT
     EOT    DB  "."
      D1    ENDS

      C1    SEGMENT
            ASSUME  CS:C1
  LOWLAT    PROC  FAR

; процедура за замяна на големи латински букви с малки
; на вход чрез регистър AL постъпва неизвестен символ
; на изход регистър AL съдържа малка буква, ако е въведена голяма,
; в противен случай регистър AL остава непроменен.

            CMP  AL, "A"
            JB  NOLAT
            CMP  AL, "Z"
            JA  NOLAT
            ADD  AL, -"A"+"a"       ; AL := (AL) - "A" + "a"
  NOLAT:    RET
  LOWLAT    ENDP
      C1    ENDS
            END

; главен модул
            INCLUDE  IO.ASM
            EXTRN  EOT:BYTE,  LOWLAT:FAR

       S    SEGMENT  STACK
            DB  128  DUP (?)
       S    ENDS

       D    SEGMENT
            TXT   DB  100  DUP (?), "."
       D    ENDS
```

```
      C  SEGMENT
         ASSUME  SS:S, DS:D, CS:C
START:   MOV   AX, D
         MOV   DS, AX              ; DS=D  осигурява достъп до TXT
         MOV   AX, SEG  EOT
         MOV   ES, AX              ; ES=D1 осигурява достъп до EOT
         MOV   SI, 100
         OUTCH  ">"                  ; покана за въвеждане
   INP:  INCH  AL                    ; AL := символ
         CMP   AL, ES:EOT            ; край ли е?
         JE  PR
         CALL  LOWLAT                ; замяна на символ
         DEC   SI
         MOV   TXT[SI], AL           ; запис в масива TXT
         JMP   INP                   ; затваря цикъла в точка INP
   PR:   LEA  DX, TXT[SI]
         OUTSTR                      ; извежда символа
         FINISH
      C  ENDS
         END  START
```

Г Л А В А 13

ВЪВЕЖДАНЕ – ИЗВЕЖДАНЕ. ПРЕКЪСВАНИЯ

Целта на настоящата глава е изясняване на операциите въвеждане и извеждане на данни, както и макросите INCH, OUTINT и пр., които бяха използвани за по-кратко и бързо пояснение на предходните теми.

13.1 Команди за вход-изход

На ниво процесор входно-изходните операции най-често се осъществяват между регистър и порт. Под *порт* се разбира клетка във входно-изходното адресно пространство (за подробности вижте [1]). Полезната част от В/И пространство са адресите от 00000h до 0FFFFh, което потенциално осигурява 65536 на брой еднобайтови порта. Разпределението на тези адреси за нуждите на външните устройства (дискове, клавиатура, дисплей, принтер, скенер, комуникационни устройства и др.) е стандартизирано и се използва от фирмите производителки на оборудване. Ще припомним познатата от [1] престава за външното устройство като регистрова проекция във В/И-то адресно пространство, която проекция се нарича *програмен модел* на ВУ.

В системата машинни команди на разглеждания тук микропроцесор съществуват следните команди за изпълнение на операция "въвеждане" и операция "извеждане":

; *команди за четене от порт* №n (*въвеждане*);

```
        IN  AL, n        или        IN  AX, n
```

; *команди за запис в порт* №n (*извеждане*);

```
        OUT  n, AL       или        OUT  n, AX
```

Номерът на порта в горните команди може да бъде зададен по два начина: явно, като число от 0 до 255, или като двоично съдържание на регистър DX. Например:

```
        IN   AL, 97h           ; AL := (port 97h)
        MOV  DX, 765           ; (DX)=0000001011111101b = 765
        OUT  DX, AX            ; port 765 := (AX)
```

Първият вариант за задаване номера на порта се използва, когато той е число, представимо в един байт. Вторият вариант – когато номерът на порта е голямо число или число, което става известно в хода на изпълнение на програмата.

Сценарият, по който се реализира входно-изходният обмен, съществено зависи от спецификата на външното устройство, но достатъчно типичен е сценарият, на който се спираме по-долу.

Често процесорът е свързан с външното устройство с 2 порта: през единия се обменят данни – даннов порт, а през другия се обменя различна по характер информация, предназначена за управление на ВУ, неговите режими на работа, неговото състояние, и в този смисъл

самия процес на входно-изходен обмен. Като цяло обменът е диалогов процес, по време на който процесорът и ВУ "разговарят". Тук, от гледна точка на процесора, ВУ не следва да се разглежда като "безжизнен" обект. То притежава своята система за управление, която го "оживява", и която му дава съответните функции и начин на работа.

Възможно най-тривиалният сценарий (разбирай алгоритъм), по който може да протече обмена, е следния: процесорът (разбирай изпълняваната в него програма) желае да изведе данни чрез някакво ВУ. Той записва в управляващия порт на това устройство някаква комбинация, разбираема за устройството, като информация за това каква е операцията и как следва да я изпълни. След това записва в данновия порт данните, които изпраща. От своя страна ВУ, прочитайки управляващата информация обявява на всички, че е заето и започва изпълнението на операцията под управлението на собствения си контролер, например отпечатва на хартия предадения символ. През това време централният процесор или изчаква работата на ВУ, запитвайки в непрекъснат цикъл за състоянието му, или се прекъсва и преминава към изпълнението на друга програма. Това продължава, докато в управляващия порт на ВУ не бъде установено състояние на успешен край на текущата операция, както и готовност за нов обмен. Откриването на сбой в работата на ВУ е едно от възможните разклонения в описвания сценарий. Ако всичко е нормално, обменът може да продължи.

Осъществяването на В/И-ят обмен по този сценарий от потребителската програма е изключително тежко. Този подход изисква познаването на голям брой технически детайли, които даже няма да споменаваме. Освен това във всяка нова програма тези действия ще трябва да се описват отново, може би моногократно. Ясно е, че подобен подход е нерационален. Ето защо В/И-ят обмен се описва отделно, което описание скрива от потребителя "кухнята" на входно-изходния обмен, като в същото време това описание е включено в състава на операционната система и постоянно се намира в оперативната памет, готово да бъде използвано от всяка програма, която се нуждае от В/И-ден обмен.

Във връзка с краткото изложение, ние вече няма да говорим за машинните команди, изпълняващи входно-изходни операции, а ще разгледаме входно-изходните операции, които са включени в състава на операционната система.

13.2 Прекъсвания. Функции на DOS

Една от основните задачи на ОС е управление на работата на всички устройства, при непрекъснато оперативно следене на всички събития, случващи се в компютърната система. Основен подход за това са така наречените прекъсвания, принципът и организацията на които тук ще приемем, че са напълно известни на читателя.

Съществена роля в реализацията на всяко прекъсване играе стекът. При прекъсване, следващият адрес, след текущо изпълняваната машинна команда и по време на нейното изпълнение, се съдържа в адресната двойка CS:IP. Този адрес се нарича *адрес за връщане*. Ако процесорът, по силата на командния цикъл, реши да обслужи избраната заявка за прекъсване, той изпълнява микропрограмната процедура за осъществяване на прекъсването (МППОП, вижте [1]), според която в стека се съхранява адресът за връщане – първо съдържанието на регистър CS, а след това и съдържанието на програмния брояч IP. Вече отбелязахме в предходни глави, че дори и да не използва стек, всяка програма следва задължително да го предвиди. Освен адреса за връщане, в стека се запазва текущото съдържание на флаговия регистър. Ако признаците не бъдат съхранени, следващи условни преходи по-надолу в програмата биха били неправилно изпълнени след връщането от прекъсване. Освен съхранение на текущото състояние, преди да се предаде управлението на ОС, МППОП нулира флаговете IF и TF, а така също въвежда номера на приетото прекъсване (k). Уникалният номер на приетото прекъсване се преобразува в адрес. От този адрес МППОП прочита началния адрес на програма от ОС, която е специално предназначена да обслужи приетото прекъсване. За да стартира тази програма, която наричаме *обслужваща*, МППОП трябва да запише прочетеният от ОП адрес, в адресната двойка CS:IP. Това е ново съдържание на тези два регистъра, с което се затваря командния цикъл и се извършва преходът, наречен прекъсване.

Обслужващата даденото прекъсване програма по същество представлява процедура. За да работи с регистрите на процесора, като всяка "дисциплинирана" процедура, тя следва да спаси в стека съдържанието им, а след това, преди да върне управлението, трябва да ги възстанови – процес, който ние вече описахме в предходни глави.

И последното уточнение се отнася за възстановяването на прекъснатата програма. Това е задача на обслужващата процедура, която изпълнява последната си команда "*връщане от прекъсване*". Такава машинна команда има в системата машинни команди на всеки процесор. В разглеждания тук тя има мнемоничния код IRET (*interrupt return*) и се записва без операнди. Командата изпълнява действията:

$$\text{Стек} \rightarrow \text{IP}, \quad \text{стек} \rightarrow \text{CS}, \quad \text{стек} \rightarrow \text{Flags}$$

Със изпълнението на тази команда, в адресната двойка се оказва ново съдържание, с което след затваряне на командния цикъл се извършва преход към команда, която се оказва първата в точката на прекъсване на прекъснатата програма. С това се възобновява нейната работа.

13.2.2 Функции на DOS

С помощта на кратко представения по-горе механизъм, компютърната система става много гъвкава и адекватна от към външни събития. С помощта на същия механизъм обаче може да бъде органи-

зирана реакцията на системата и на вътрешни събития. Безспорно такова събитие е ситуацията, която настъпва при изпълнение на операция деление. Ако се установи, че делителят е 0, което програмистът не би могъл да предвиди, операцията е недефинирана, което е основателна причина изпълнението на текущата програма да бъде прекратено. Прекъсването се осъществява по описаната вече схема, а по-нататъшните действия зависят от обслужващата програма.

Друга ситуация, в която може да бъде полезен механизмът на прекъсване, ние вече подсказахме в предходните глави. Става дума за често изпълняваните действия за въвеждане или за извеждане на символи. Споменахме, че за целта са разработени съответни процедури, които са включени в състава на ОС и могат да се използват от потребителя без да му се налага да изпада в досадните подробности на схемите за управление на ВУ. Превключването обаче от потребителската програма към тези функции, които явно следва да връщат обратно управлението в потребителската програма, е вече подразбиращо се въз основа на механизма за прекъсване.

Така стигаме до въпроса: как бихме могли да си облекчим работата като използваме тези функции? Разбира се, бихме могли да използваме команда CALL, за което обаче трябва да знаем началния адрес на всяка функция. За съжаление тези адреси са доста изменчиви от версия към версия на ОС, а така също и във времето. Този проблем е фундаментален и по принцип е свързан с преместваемостта. Отчитайки това, процедурите в състава на ОС е прието да се извикват различно. За целта в системата машинни команди е реализирана специална команда, наречена "*команда за програмно прекъсване*". Като си припомним казаното за разпределението на портовете във входно-изходното адресно пространство, така и тук се прави разпределение на номерата на прекъсванията, които като 8 битови числа, образуват едно множество от 256 възможни причини за прекъсване. Това множество е достатъчно голямо по обем и значителна част от тези номера могат уникално да бъдат свързани с причините за програмни прекъсвания. Тъй като разпределението на номерата на прекъсванията е закон и не се променя, то потребителят може да бъде откъснат от конкретиката на адресирането, което ще остане скрито за него и ще бъде функция на системата. Командата за програмно прекъсване има вида:

INT <№k, номер на прекъсване> ; тип и диапазон [0≤i8≤255]

Тази команда изпълнява принудително прекъсване на текущата програма, с действия аналогични на МППОП. За формиране на адреса в ОП, от който ще се прочете началния адрес на обслужващата програма, се използва операндът на командата.

В състава на MS DOS влизат много процедури, за които възможните номера от множеството [0,255], са недостатъчни. Това е ситуация, която допълнително усложнява организацията на програмното прекъсване.

За целта функциите се групират в групи, чийто брой не може да бъде повече от 256, естествено. Вътре във всяка група се въвежда допълнителна номерация. Това означава, че зад номера на някое прекъсване могат да се различават десетки конкретни функции за обслужване на дадено прекъсване, които са разновидност на основното. Адресирането на обслужваща програма в тези условия изисква освен задаването на номера на прекъсването още и задаване на номера на поредната функция в групата на това прекъсване. Ако стойността на номера на в групата е 8 битово число, то функциите в тази група не могат да бъдат повече от 256. Така за 256 прекъсвания могат да бъдат създадени до 256*256=65536 на брой различни обслужващи програми. Това множество е впечатляващо и няма система, която да го изпълва.

Извикването на желаната за дадено прекъсване функция е уговорено да става чрез предаване на номера ѝ през регистър AH. Програмистът е длъжен да запише следните 2 команди:

```
MOV   AH, <№ на функция>
INT   <№ на прекъсване>
```

След като получи по команда INT управлението, ОС с помощта на записания в регистър AH номер, предава управлението на желаната от програмиста обработваща функция.

Изпълнението на функцията може да изисква допълнителни данни. Обикновено те се предават чрез регистрите и условията за това са описани за всяка функция отделно, с което програмистът следва да се съобрази.

Тук ще бъдат разгледани само част от функциите, които са в групата на прекъсване №21h (21h=33). Това са функции, с помощта на които ще може да се организира по-лесно за програмиста входно-изходен обмен.

13.2.3 Някои функции на прекъсване 21h

Ще започнем с едно неизбежно за всяка програма действие, защото

"*колкото и дълго да работи една програма, тя рано или късно завършва ...*"

Завършване на програмата

Завършвайки своята работа, всяка програма е длъжна да предаде управлението на друга програма (по силата на командния цикъл), която обикновено е част от ОС. Този специфичен преход при завършване на работа се реализира не от самата потребителска програма. Последната само извиква прекъсване, което да осъществи прехода и да "почисти" след нея. Последното действие, което програмата изпълнява, е това на команда INT 21h. Завършващите действия обаче са програмирани във функция 4Ch на това прекъсване. Ето защо като последни действия в своята програма програмистът следва да запише:

```
MOV  AL, <код на завършването>
MOV  AH, 4Ch
INT   21h
```

Тук, преди вече споменатите команди, се вижда още една. Тя е необходима като съобщение, което да каже как (нормално или не) завършва изпълнението си потребителската програма. Това е нещо, което прави завършването по-гъвкаво и дава по-голяма свобода на действие от страна на ОС. Обикновено кодът на завършване е 8 битово число, а случаят на нормално и безгрешно завършване се кодира с нула. Ако кодът на завършване не е нула, то ОС може да интерпретира състоянието на завършване по поставения в регистър AH код. Формирането на кода на завършване е дело на потребителската програма, т.е. на нейния алгоритъм.

Извеждане на екрана (в текстов режим)

За извеждане на графичния дисплей на един символ се използва функция 02h на прекъсване 21h. За целта програмистът записва:

```
MOV   DL, <код на символ за извеждане>
MOV  AH, 02h
INT   21h
```

Указаният символ се визуализира в позицията на курсора, след което курсорът се премества на следващата позиция вдясно. Ако курсорът се е намирал в последната за реда позиция, след визуализирането на символа, той се премества в началото на следващия ред. В случай че курсорът е бил в последната позиция на последния ред на екрана, то той се премества в началото на следващия ред, който се появява на екрана след като всички предидущи редове се преместят с един по-нагоре.

Специално се визуализират символите с номера 7, 8, 9, 10 (0Ah) и 13 (0Dh). Символът с код 7 (*bell* - звънец) не се визуализира на екрана (и курсорът не се придвижва), а се изпълнява, появява се, звуков сигнал. Символ с код 8 (*backspace* – стъпка назад) връща курсора една позиция вляво (назад), ако само не е бил в най-лявата. Символът с код 9 (*tab* - табулация) измества курсора вдясно на най-близката позиция, кратна на 8. Символът с код 10 (*line feed* – преход на нов ред) премества курсора на нов ред, но в същата позиция. Символът с код 13 (*carriage return* – връщане на каретката) позиционира курсора в началото на текущия ред. Последователното извеждане на символите 13 и 10 означава преместване на курсора в началото на следващия ред.

За извеждане на екрана на дисплея на символен низ (последователност от символи) се използва функция 9 на прекъсване 21h, като за целта програмистът следва да запише:

```
; DS:DX := началния адрес на символния низ
MOV  AH, 09h
INT   21h
```

Както може да се види, тази функция изисква низът да се намира в данновия сегмент, а неговия начален адрес да бъде зареден в адресната двойка DS:DX. Още тази функция изисква краят на низа да бъде маркиран със символ “$” (код 24h). Този символ е резервиран за маркиране на края на символни низове и не се визуализира (не се извежда). За кодиране на символите и техните кодови таблици, вижте [1].

Сред множеството функции на DOS няма такава, която извежда числа. Ако е необходимо, такава операция се програмира с помощта на вече описаните функции.

Въвеждане на символи от клавиатурата

При натискане (в произволен момент) на клавиш от клавиатурата, генерираният при това код на символ, се записва в специален буфер, откъдето в последствие се прочита от функциите за въвеждане. Поради паралелността в работата, въвеждането на символи от клавиатурата към буфера може да се осъществява още преди програмата да е задействала програмните функции, свързани с въвеждането на същите символи от буфера към ОП. Размерът на този буфер е 15 символа. Ако въвеждането от клавиатурата изпреварва изчитането на символите от буфера, възприемането на символи от клавиатурата се преустановява и се замества със звуков сигнал. Ако е извикана функция за въвеждане, но буферът се окаже празен, тя остава в състояние на очакване.

Ако програмата не желае да въведе предварително набраните от клавиатурата символи, тя трябва да изчисти входния буфер на ОС, което се реализира от функция №12 (0Ch) на прекъсване 21h при нулево съдържание на регистър AL:

```
MOV  AL, 0
MOV  AH, 0Ch
INT   21h
```

От всички функции на DOS, които реализират въвеждане от клавиатурата, ще разгледаме само една – функция №10 (0Ah) на прекъсване 21h. С помощта на тази функция могат да бъдат въведени няколко символа, с възможност за редактиране на набрания текст:

```
DS:DX := адрес на буфера за запис на набрания текст
MOV  AH, 0Ah
INT   21h
```

Тази функция въвежда символи до тогава, докато не бъде натиснат клавиш *Enter*, и ги записва в указания буфер. Въведените символи се визуализират на екрана, което се нарича *въвеждане с ехо*. Докато не е натиснат клавишът *Enter*, набраният текст може да бъде редактиран с помощта на следните клавиши:

Backspace (←)	- отмяна на последния символ ;
Esc	- отмяна на набрания текст .

Буферът, чийто начален адрес се задава чрез адресната двойка

DS:DX, е буфер не на ОС, а на програмата, трябва да има следната структура:

max	n	s1	s2	s3	…	sn	CR		…		
0	1	2	3	4		n+1	n+2				*max*-1

Преди обръщението към функцията, в началния байт (с индекс 0), трябва да бъде записано число (*max*), показващо максималния брой символи, които функцията има право да въведе (обемът на буфера трябва да бъде оразмерен по този брой). Ако вече са въведени (*max*-1) символа, то следващите не се въвеждат и не се записват в буфера – издава се само звуков сигнал (символът с номер *max* ще бъде *Enter*).

Числото (n) на реално въведените символи се записва от функцията във втория байт на буфера, а самите символи започват да се подреждат от третия байт нататък. В байт номер (n+2), т.е. веднага след последния символ, се записва символ с код 13 (CR – край на реда), съответстващ на натиснатия клавиш *Enter*. В числото (n) този клавиш не се отчита. Пример:

```
BUF   DB  10, ?, 10 DUP (“ “)         ; в данновия сегмент
      .... .... .... ....
      LEA  DX, BUF
      MOV  AH, 0Ah
      INT   21h
```

Ако при въвеждането са набрани клавишите ABC, то съдържанието на буфера BUF ще бъде следното:

BUF[0]=10, BUF[1]=3, BUF[2]=41h, BUF[3]=41h, BUF[4]=43h, BUF[5]=13

останалите байтове в буфера не се променят.

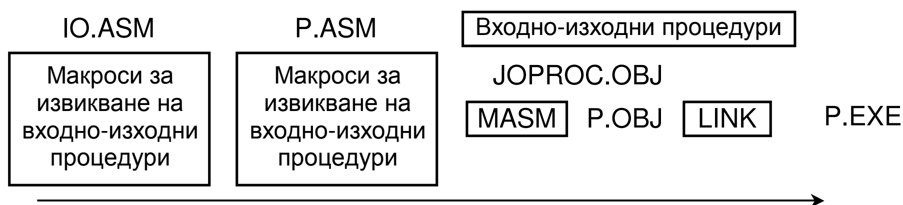
13.3 Операции вход-изход

Както се вижда от изложеното до момента, функциите на операционната система реализират много полезни, но за съжаление “дребни” операции вход-изход, операции, които обменят порции данни с дължината на байт. Това не е особено приятно, защото с тяхна помощ не може да се изведе число на екрана. Освен това на това ниво отсъствието на по-крупни входно-изходни операции затруднява изучаването на езика Асемблер, защото не позволява построяването на завършени програми. В същото време да се изясняват тези операции в началните етапи на изучаване на езика е още рано. Ето защо в тази книга бяха използвани в аванс подобни операции, но сега е вече време те да бъдат пояснени.

Тук е разгледан един възможен вариант за реализация на входно-изходни операции.

13.3.1 Схема за съхраняване и подключване на операциите В/И

Нека разгледаме следната схема:



Една част от входно-изходните операции е реализирана във вид на процедури, описанията на които са събрани в отделен модул, съхраняващ се във файл IOPROC.ASM (текстът на модула е приведен по-долу тук). Предполага се, че този модул е транслиран предварително и съществува като обектен модул, записан във файл IOPROC.OBJ. Даденият модул трябва да бъде присъединен към програмата по време на свързващото редактиране. За целта той е извикан така:

```
LINK  P.OBJ + IOPROC.OBJ, P.EXE
```

Така пълният цикъл за обработка на програмата (транслация, свързване и изпълнение) се реализира от следните команди:

```
MASM  P.ASM. P.OBJ, P.LST
LINK  P.OBJ + IOPROC.OBJ, P.EXE
P.EXE
```

Ако бяха само процедури от модула IOPROC, тогава в програмите би се наложило да се изписват по няколко пъти следните команди за обръщение към тези процедури. Например, извеждането на числа със знак изглежда така:

```
MOV  AX, x              ; извеждане стойността на x
MOV  DL, n              ; ширина на полето за извеждане
CALL  PROCOUTINT        ; извикване на процедурата
```

Подобен текст е достатъчно дълъг, освен това е свързан с осигуряване на съответните регистри, ето защо обикновено тези малки текстове най-често се оформят във вид на макроси. Например, с макрокомандата:

```
OUTINT  x, n
```

може да се извърши същото извикване, както с горните 3 команди.

Ще отбележим, че някои операции (например, FINISH) са толкова кратки в текст, че не си струва да бъдат извиквани като процедури, ето защо се предпочита описването им във вид на макроси.

Необходимо е да се помни, че обръщението към макрос фактически е обръщение не към процедура, а към самата операция. Описанието на такива макроси е събрано в отделен файл, който се нарича IO.ASM (текстът на този файл също е приведен по-долу).

Тъй като в този файл са събрани само макроопределения, а по тях, както е известно, не се формират никакви машинни команди, то този файл не образува модул, който може да бъде транслиран отделно. По тази причина този файл се подключва към програмата не по време на етапа на свързващото редактиране, както например модулът IOPROC.OBJ, а още на етапа транслиране – по директивата

INCLUDE IO.ASM.

По силата на тази директива в текста на програмата се включват всички макроопределения от файла, след което в програмата програмистът има право да използва такива макрокоманди като

OUTINT x,n и FINISH .

В "Приложение-Б" са приведени текстовете на файловете IOPROC.ASM и IO.ASM без допълнителни пояснения, тъй като всички използвани в тях програмни похвати бяха изяснени в отделните глави тук, а освен това тези текстове съдържат достатъчно коментари.

Във файл IO.ASM са употребени директивите .XLIST и .LIST, които са пояснени в следващата глава.

Г Л А В А 14

Д О П Ъ Л Н Е Н И Я

Тук ще се спрем на по-рядко използваните командите в езика Асемблер.

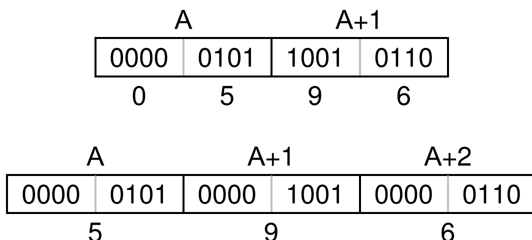
14.1 Двоично-десетични числа

В цифровия процесор освен двоично представените числа, се използват и двоично-десетичните числа (2/10-чни). Това е естественото представяне на числата след тяхното въвеждане, например, от клавиатурата, а също и непосредствено преди тяхната визуализация. Аритметиката на 2/10-чните числа е добре позната на читателя от [1] и от [2]. Кодът за представяне на десетичните цифри е 8421 или още код на прякото заместване, чиято кодова таблица е следната:

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

За всяко 2/10-чно число в паметта се отделят съседни байтове – колкото са необходими. Редът, в който цифрите заемат тези байтове, казано въобще, не е фиксиран и се определя от програмиста. Работата е в това, че обработката на такива числа се провежда цифра по цифра и се организира от самия програмист. За определеност обаче, ние тук ще избегнем тази свобода и ще приемаме, че ляво стоящите в байтовете 2/10-чни цифри са старши, а дясно стоящите – младши. И още, старшите байтове са в клетки с по-малките адреси, а младшите байтовете с младшите цифри – в клетки с по-големи адреси.

Известно е [1], че 2/10-чните числа се представят в два вида формàти: опакован (пакетиран) и не опакован (не пакетиран). В първия формàт всеки байт съдържа по две 2/10-чни цифри (по 2 тетради), а във втория формàт – само по една, която е разположена в дясната му половина. Например, за числото 0596:



В Асемблер, променливите, които имат 2/10-чни стойности, могат да бъдат описани (декларирани) с директивата DB:

```
    PACK    DB  5h, 96h      ; числото 596 в пакетиран формат
  UNPACK    DB  5, 9, 6      ; числото 596 в непакетиран формат
```

За да сме сигурни, че 2/10-чните цифри в един байт ще бъдат представени така както искаме, следва да ги изписваме в 16-чна бройна система или в двоична, защото ако ги запишем в 10-чна, числото ще бъде записано в 2-чна бройна система.

Непакетираният вид на числото се различава от ASCII-символния вид по старшата тетрада. Непакетираното число съдържа старша тетрада нула, а символите на цифрите съдържат кода на своята зона 30h. Например, като символен низ числото 596 има вида:

0011	0101	0011	1001	0011	0110
3**5**h		3**9**h		3**6**h	

Въз основа на казаното, преминаването от единия вид в другия, може лесно да бъде съобразено.

14.1.1 Събиране на двоично-десетични числа

Събиране на неопаковани (непакетирани) числа

Събирането на не опаковани (непакетирани) числа е в същност събиране на 2 байта, всеки от които съдържа само по една десетична цифра, например: 5+8=13. Поради това, че получената сума не е едноцифрена, тя се представя по всички правила на бройната система, т.е. чрез функцията на едноразрядната сума и функцията на преноса. Този десетичен резултат не може да се получи чрез операция двоично събиране, която напомняме, че е реализирана хардуерно чрез суматора в АЛУ на процесора. Ето защо се налага изпълнението на втора, специална операция – операция 2/10-чна корекция. Тази корекция се изпълнява от командата AAA (*ASCII Adjust After Addition*), която работи само с регистър AL. Така примерът, 5+8=13, следва да бъде програмиран с командите:

```
MOV  AL, 5
ADD  AL, 8          ; (AL)=0Dh ,   D ≡ 13
AAA                 ; CF=1 ,    (AL)=03h
```

Получената сума се представя така: старшата цифра 1 се съдържа във флаг CF, а в регистър AL – неопакованата младша цифра 3.

Ще разгледаме следната задача: да се съберат две N-цифрени неопаковани числа Z=X+Y. Текстът на програмата е следния:

```
N   EQU  …              ; N>0
X   DB  N  DUP (?)
Y   DB  N  DUP (?)
Z   DB  ?, N  DUP (?)   ; левият байт е за възможния пренос

; събиране на не опаковани N-цифрени 2/10-чни числа Z=X+Y

        MOV  CX, N
        MOV  SI, N-1
        CLC                 ; CF := 0
  SUM:  MOV  AL, X[SI]
```

```
        ADC  AL, Y[SI]          ; AL := (AL)+Y[SI]
        AAA                     ; AL := (AL)+корекция
        MOV  Z[SI+1], AL
        DEC  SI
        LOOP  SUM               ; DEC и LOOP не променят флаг C
        MOV  Z, 0
        ADC  Z, 0               ; запис на старшата цифра
```

Събиране на ASCII-числа

Командата AAA е разработена така, че тя може да реализира 2/10-чна корекция и при събиране на ASCII-числа. В този случай младшата цифра в байтовата сума се коригира по правилата на код 8421, а старшата цифра следва да бъде равна на кода на зоната, т.е. 3. За примера 5+8=13 имаме: за 5 кода 35h, а за 8 кода 38h. Тогава 5+8 е сумата 35h+38h, което след двоичното събиране е 6Dh. Това разбира се не е верният резултат, затова следва операция корекция, изпълнена от команда AAA. В резултат от нейното действие ще се получи:

CF=1 ,   (AL) = 33h

Като пример ще разгледаме горната задача Z=X+Y, но при условие, че цифрите на числата са ASCII символи.

В програмата се налага използване на команда OR, която променя стойността на флаг C, ето защо се налага преносът в старшите цифри да се отчита чрез регистър AH.

```
; събиране на ASCII числови низове
        MOV  CX, N              ; N – брой на цифрите
        MOV  SI, N-1            ; индекс на цифрите
        MOV  AH, 0              ; пренос в най-младшия разряд
  SUM:  MOV  AL, X[SI]
        ADD   AL, Y[SI]
        ADD   AL, AH            ; събиране с пренос
        MOV  AH, 0
        AAA
        OR  AL, 30h             ; цифрата става символ
        MOV  Z[SI+1], AL
        DEC  SI
        LOOP  SUM
        OR  AH, 30h
        MOV  Z, AH
```

Събирана на опаковани (пакетирани) числа

И в този случай се налага дългите числа (с по няколко байта) да събираме на части (побайтно). Разликата е в това, че във всеки байт са представени по две 2/10-чни цифри. Всяка порция от числата - двойка цифри се събира с две команди: ADD (или ADC) с последваща корекция DAA (*Decimal Adjust Addition*). Тази команда също работи с регистър AL.

Например:

```
            MOV  AL, 59h
            ADD  AL, 69h              ; AL := 0C2h
            DAA                       ; CF=1  ,    (AL)=28h
```

Теорията на двоично-десетичната корекция в този случай е подробно изложена в [1] и в [2] и е позната на читателя, ето защо тук ще бъде спестена. При събиране на две двуцифрени числа може да се получи трицифрена сума, старшата цифра на която се съдържа в CF.

Като пример ще разгледаме същата задача Z=X+Y, когато числата са представени в пакетиран формат. Така за N-цифрените операнди ще са необходими M=½N на брой байта. За простота на примера ще приемем, че N е четно число, тогава M (броят на байтовете, заети от числата) ще бъде цяло число.

```
 ; събиране на опаковани N-цифрени 2/10-чни числа Z=X+Y
            MOV  CX, M
            MOV  SI, M-1
            CLC
    SUM:    MOV  AL, X[SI]
            ADC  AL, Y[SI]
            DAA
            MOV  Z[SI+1], AL
            DEC  SI                   ; командите DEC и LOOP
            LOOP  SUM                 ; не променят флаг C
            MOV  Z, 0
            ADC  Z, 0
```

14.1.2 Изваждане на двоично-десетични числа

Операция изваждане се организира алгоритмично по същия начин, като операция събиране. Случаите са същите. За реализация на корекциите са предвидени две команди:

- ASCII-корекция след двоично изваждане: AAS;
- Десетична корекция след двоично изваждане: DAS.

Командите работят с регистър AL и формират възможния заем във флаговия бит CF.

Пример за изваждане на не опаковани 2/10-чни числа Z=X-Y:

```
            MOV  AL, 05h              ; Z=5-2
            SUB  AL, 02h              ; (AL)=03h ,   AF=0
            AAS                       ; (AL)=03h ,   CF=0    без корекция

            MOV  AL, 02h              ; Z=2-5
            SUB  AL, 05h              ; (AL)=0Fh ,   AF=1
            AAS                       ; (AL)=07h ,   CF=1    с корекция
```

Команда AAS се използва както за неопаковани, така и за ASCII-символи, аналогична на команда AAA.

Когато се изваждат опаковани 2/10-чни числа при обработка на всеки байт се използват две команди – двоично изваждане SUB (или SBB) с

последваща корекция DAS. Например:

```
        MOV  AH, 0
        MOV  AL, 25h
        SUB  AL, 52h        ; (AL)=D3h ,  AF=0 ,  CF=1
        DAS                 ; (AL)=73h ,  CF=1
```

14.1.3 Умножение и деление на двоично-десетични числа

Операции умножение и деление се реализират сложно, което се дължи на това, че основната бройна система на процесора е двоичната. Все пак в системата машинни команди на разглеждания процесор са включени команди за изпълнение на тези операции, но само върху <u>едноцифрени неопаковани операнди</u>. Към тези команди следва да се добавят и командите за корекция след умножение и след деление.

Корекцията след двоично умножение MUL изпълнява командата AAM: AAM (ASCII *Adjust After Multiply*). Командата AAM дели стойността в регистър AL на 10 и записва непълното частно в регистър AH, а остатъка – в регистър AL, получавайки по този начин в тези два регистъра правилните десетични цифри на произведението. Например:

```
        MOV  AL, 6
        MOV  BH, 9
        MUL  BH             ; (AL)=54=36h ,  (AH)=0
        AAM                 ; (AH)=5 ,  (AL)=4
```

Задача: да се умножи N-цифрено непакетирано 2/10-чно число X с едноцифреното число Y, Z=X.Y. Изпълняваме операцията по метода с младшите разряди напред – всяко поцифрено произведение прибавяме към междинната сума.

```
      N  EQU  …                 ; N>0
      X  DB  N  DUP (?)
      Y  DB  ?                  ; цифра от 0 до 9
      Z  DB  ?, N  DUP (?)      ; първият байт е за преноса
         ....  ....  ....  ....

         MOV  CX, N             ; брой на цифрите в X
         MOV  SI, N-1           ; индекс
         MOV  BH, 0             ; най-младшият пренос е 0
   MLT:  MOV  AL, X[SI]         ; зарежда цифра от множимото
         MUL  Y                 ;  (Xi)*Y
         AAM                    ;  (AH)=ст. цифра,  (AL)=мл. цифра
         ADD  AL, BH            ; отчита пренос от мл. цифри
         AAA                    ; 2/10-чна корекция на сумата
         MOV  Z[SI+1], AL
         MOV  BH, AH            ; старша цифра като пренос в BH
         DEC  SI
         LOOP  MLT
         MOV  Z, BH             ; най-ст. цифра на произведението
```

При двоично деление с команда DIV за корекция се използва команда AAD (*ASCII Adjust Before Division*). Тази команда, за разлика от всички команди за корекция, разгледани до момента, се поставя преди командата за двоично деление. Веднага след поставяне на делимото в регистър AX (двойка непакетирани 2/10-чни цифри – старшата в регистър AH и младшата в регистър AL) се поставя команда AAD, която преобразува тази двойка цифри в двоично число и го записва в регистър AX така: AX := (AH)*10+(AL). След това командата DIV изпълнява операция деление. Например:

```
MOV  BH, 9              ; делител
MOV  AX, 0307h          ; делимо = 37  -  не опаковано
AAD                     ; (AX)=25h  - като двоично число
DIV  BH
```

Задача: да се напише програма за деление на N-цифрено не опаковано 2/10-чно число X на едноцифреното число Y със запис на частното в Q и на остатъка в R.

Предлага се следния алгоритъм. Нека X=746, Y=3. Вземаме 0 и първата старша цифра 7 и делим не опакованата двойка (07) на Y. Частното (2) записваме в Q, а остатъка (1) конкатенираме със следващата цифра от делимото X (4). Получената не опакована двойка 2/10-чни цифри (14) делим на Y, след което записваме полученото частно (4) в Q, а остатъкът (2) отново конкатенираме със следващата цифра от делимото (6). Така новата двойка (26) делим на Y, след което записваме полученото частно (8) в Q, а остатъкът (2) – в R.

```
     N  EQU …
     X  DB  N  DUP (?)
     Y  DB  ?
     Q  DB  N  DUP (?)      ; масив за цифрите на частното
     R  DB  ?
        ....  ....  ....
     MOV  CX, N             ; брой на цифрите в X
     MOV  SI, 0             ; индекс – номер на цифра от X
     MOV  AH, 0             ; остатък от деление
DV:  MOV  AL, X[SI]         ; зарежда поредната цифра от X
     AAD
     DIV  Y
     MOV  Q[SI], AL         ; запис на поредната цифра на частното
     INC  SI
     LOOP  DV
     MOV  R, AH             ; запис в R на остатъка
```

14.2 Допълнителни команди

Тук кратко ще представим по-рядко използвани команди. Например:

Команди за управление на флагове LAHF, SAHF, CLC, STC, CLI, STI

Команда LAHF (*Load AH From Flags*) – зареждане на флаговете в

регистър AH.

Команда SAHF (*Save AH into Flags*) – възстановяване на флаговете от регистър AH.

Команда CLC (*Clear Carry Flag*) – нулиране на флаг CF.

Команда STC (*Set Carry Flag*) – установяване на флаг CF.

Команда CMC (*Complement Carry Flag*) – противоположна стойност на флаг CF.

Споменатите команди не се нуждаят от пояснения.

Съществуват и други команди, управляващи стойностите на отделни флагове, например маската за прекъсване:

Команда CLI (*Clear Interrupt Flag*) – нулиране на флаг IF (на маската).

Команда STI (*Set Interrupt Flag*) – установяване на флаг IF.

Ако флагът IF (маската за прекъсване) е нулиран, това означава, че процесорът ще игнорира всички заявки за прекъсване, с изключение на подадените по причина на хардуерни грешки. Тези команди се използват обикновено в процедурите на ОС, когато при обработка на дадено прекъсване се налага да се игнорират други прекъсвания.

Команда INTO

Команда INTO (*Interrupt if Overflow*) – програмно прекъсване при препълване. Ако при изпълнение на тази команда флагът OF е в състояние 1, то тя изпълнява прекъсване на текущата програма и преминава към обслужване на прекъсване №4, т.е. командата сработва аналогично на команда INT 4.

Препълване е възможно да настъпи при операции събиране и изваждане и то е познато на читателя от [1] и [2]. Въпреки, че получаваният в такива случаи резултат не е правилен, текущата програма не се канцелира. Фактът на препълване обаче се фиксира и ако програмистът се опасява от него, той може да заложи в програмата проверка. Проверката може да се реализира с команда за условен преход или с команда INTO, например:

```
ADD  AX, Y
INTO
```

Ако е настъпило препълване, както вече беше казано се изпълнява прекъсване №4, т.е. формира се косвеният адрес (0000:0010h) към таблицата на векторите за прекъсване, откъдето се извлича началният адрес на обслужващата това прекъсване програма. Обикновено алгоритъмът на тази програма съдържа само една команда IRET "връщане от прекъсване". С други думи обслужване като такова няма, "топката" се връща в потребителската програма, която може да продължи изпълнението си. Такова "обслужване" по подразбиране се зарежда от системата при нейното стартиране. Ако програмистът желае друг алгоритъм за обслужване на ситуацията препълване, той следва да напише специална обслужваща програма и да осигурява нейния нача-

лен адрес в таблицата на векторите за прекъсване, т.е. като съдържание на клетка (0000:0010h).

Команда NOP

Празна команда NOP (*No operation*) – единствената машинна команда, която не притежава свой алгоритъм, тя не прави нищо. Единственият ефект от тази команда е загубата на машинен цикъл за нейното извличане от паметта. Използването й е много специфично, например в случаи, когато програмистът иска да резервира програмен текст, който в последствие да промени, без при това да размести вече записаните команди.

Команда HLT

Команда HLT (*Halt*) – стоп. Тази команда преустановява работата на процесора (вижте глава 1), който изпада в състояние на очакване на външен сигнал за прекъсване. Докато процесорът е в състояние на очакване, в регистрите, управлявани от командния цикъл CS и IP се запазва следващият адрес. При поява на очаквания външен сигнал, процесорът изпълнява прекъсване, т.е. вече описаната в глава 13 микропрограмна процедура за осъществяване на прекъсване МППОП, в резултат на която в стека се съхранява адресната двойка CS:IP в качеството й на адрес за връщане от прекъсване. Тази команда е удобна за програмиране на ситуации, когато се очаква натискане на клавиш от клавиатурата или някакво друго външно въздействие.

Команда LOCK

Команда LOCK (*Lock the Bus*) – блокиране на шината. С тази команда е възможно да се блокира предаването на каквато и да е информация по системната шина, докато напълно не бъде завършено изпълнението на следващата команда (която може да бъде записана с префикс за повторение или префикс на сегментен регистър). Тази команда е необходима за реализация на многозадачен режим на системата. Тя се използва за да бъде забранено на другите програми да прекъсват работата на текущата в онзи момент, когато тя изпълнява важно за системата действие (например прехвърляне на една рамка от ОП в друга или при обслужване на кеш паметта).

Команда ESC – превключване към копроцесор

Тази команда има два операнда:

ESC op1, op2

Тази команда, която се изпълнява в CPU, генерира сигнал към копроцесора (FPU) за изпълнение на някоя от неговите операции. Първият операнд задава кода на тази операция, а вторият операнд задава адрес на стойност в стека на копроцесора.

В езика Асемблер, за операциите, изпълнявани в копроцесора, са въведени специални мнемонични кодове, наподобяващи тези, които означават операциите в централния процесор. Така например, опера-

ция събиране на числа с плаваща запетая, се означава FADD. Благодарение на тези кодове команда ESC не присъства явно в асемблерните текстове.

Тъй като изпълнението на операциите в копроцесора е значително по-продължително, неговата работа е организирана паралелно с тази на централния процесор, който след предаване на командата към FPU, може да премине към изпълнение на следващата команда в програмата, стига тя да не е зависима от все още незавършилата.

Проблемите на паралелно работещите устройства и конвейерните зависимости между машинните команди са подробно разгледани в [1].

Ако такава зависимост съществува, то CPU следва да изчака изпълнението на операцията, резултата и неговите признаци. Състоянието на очакване се назначава със специална команда:

Команда WAIT – изчакай

Командата принуждава CPU да изчака сигнал от FPU, за това, че е завършил изпълнението на текущата команда.

14.3 Допълнителни оператори в езика Асемблер

Оператори за изместване.

Те имат следната структура:

<константен израз> SHR <константен израз>
<константен израз> SHL <константен израз>

Операторите изпълняват изместване в съответната посока на първия операнд, на брой разряди, указани от втория операнд. Излизащите от разрядната мрежа битове се губят, а в освободените разряди се записват нули. Например:

```
N  EQU  1011b
   MOV  CX, N  SHL  2        ; (CX)=0000000000101100b
   AND  AH, N  SHR  1        ; (AH)=00000x0xb
   OR   AH, N  SHL  5        ; (AH)=x11xxxxxb
```

Действията на тези оператори съвпадат с действията на едноименните машинни команди, но не следва да се бъркат с тях. Операторите изпълняват своето действие още по време на транслирането на програмата и в машинния й код те не присъстват.

Оператор LENGTH – <има на променлива>

Операторът е константен израз. Операндът му трябва да бъде обявен в директива за тип DB, DW, DD или директива на структура. Операторът има стойност, която показва коефициента на кратност в конструкцията DUP, ако тя е първият операнд в тази директива, и е 1 във всички останали случаи. Например:

```
A  DB   100  DUP (?)                 ; LENGTH  A = 100
B  DW   100  DUP (1, 5, DUP(0))      ; LENGTH  B = 100
C  DD    20, 30, 66  DUP(0)          ; LENGTH  C = 1
D  DB    "abcd"                      ; LENGTH  D = 1
```

Оператор LENGTH има смисъл само, ако в директивата, описваща името на променливата, е указан един операнд, който е конструкцията от вида (k DUP(x)). В този случай операторът съобщава k на брой елемента, описани в тази директива. В останалите случаи смисълът на оператора е неразбираем.

Оператор SIZE <име на променлива>

Операторът е константен израз. Операндът му трябва да бъде обявен в директива за тип DB, DW, DD или директива на структура. Стойността на оператора се изчислява по формулата:

SIZE V=(TYPE V) * (LENGTH V)

Оператор SIZE има смисъл само, ако в директивата, описваща името на променливата, е указан един операнд, който е конструкцията от вида (k DUP(x)). В този случай операторът определя броя на байтовете, заети от всички елементи, описани в тази директива. В останалите случаи операторът няма смисъл.

Оператор .TYPE <име>

Това е константен оператор. Името на операнда е произволно. Стойността на оператора е от тип байт. Стойността се формира побитово, като всеки бит указва за наличие или отсъствие на определена характеристика на името:

№ на бит	Характеристика
0	Името е описано като етикет или име на процедура
1	Името е описано като променлива
5	Името е описано в програмата
7	Името е описано (в EXTRN) като външно

Останалите битове следва да са нули.

За имена на регистри, константи, сегменти, групи, макроси, типове записи и структури, операторът формира стойност 20h. Нулева стойност на оператора означава, че името не описано по никакъв начин в програмата или представлява мнемокод на команда, директива или оператор, или представлява име, на което с директива EQU е поставена в съответствие нечислова стойност.

Оператор .TYPE се използва обикновено в макроопределения за проверка на имена, зададени като фактически параметри.

Оператор THIS <тип>

Операторът е адресен израз. Неговата стойност е адрес, който е равен на текущата стойност на брояча на поместването ($), на който се приписва указания тип. Като операнд могат да се използват служебните думи BYTE, WORD, DWORD, NEAR, FAR или съответните им числени стойности.

Операторът THIS се използва в директивата EQU за генериране на име, адресиращо текущата точка в програмата и имащо зададения тип.

Ще припомним, че за подобни цели се използва описаната вече директива LABEL. Например:

```
  A   EQU  THIS  WORD
  B   DB  20  DUP (?)        ; A и B се отнасят за един и същи масив,
                             ;   но A е масив от думи, а B – масив от байтове.
 L1   EQU  THIS  FAR
 L2:  MOV  AX, 0             ; L1 и L2 маркират една и съща команда,
                             ;  но L1 е далечен етикет, а L2 е близък етикет.
```

Оператори HIGH и LOW

Тези оператори са константни изрази:

HIGH <константен израз>
LOW <константен израз>

Техният резултат е стойността, съдържаща се в съответния байт (младши или старши). Например:

```
X  EQU  1234h
   MOV  CL, HIGH  X          ; MOV  CL, 12h
   MOV  CL, LOW  X           ; MOV  CL, 34h
```

Коментарът за последните 2 команди показва еквивалентното действие на съответния оператор.

14.4 Директиви за управление на листинга

При транслиране на програмата Асемблер води протокол, който обикновено наричаме листинг. Освен текстът на програмата, там е приведен сегментираният машинен код, диагностичните съобщения за грешки, таблица на срещнатите имена и техните атрибути и пр.

Тук са представени директиви, с които може да се влияе на формирането на листинга.

Директива TITLE

TITLE <текст>

След ключовата дума в един ред се изписва текст (без скоби), но не повече от 60 символа. Този текст се използва като заглавен в първия ред на всяка нова страница от листинга. Директивата може да бъде разположена в произволно място на програмата. Допуска се само една директива TITLE. Ако такава директива няма, заглавните редове на страниците в листинга ще бъдат празни.

Директива SUBTTL

SUBTTL <текст>

Изписаният след ключовата дума текст в тази директива се използва като подзаглавие, което ще се изписва на втория ред на всяка страница от листинга, започвайки от следващата страница. Ако директивата липсва, вторият ред остава празен. В програмата може да има няколко директиви SUBTTL. Тази директива може да се употреби и без директивата TITLE.

Директива за странициране

PAGE
PAGE +
PAGE [<дължина>] [, <ширина>]

Дължината е цяло число от 10 до 255 (по подразбиране е 50) и задава броя на редовете в страницата.

Ширината е цяло число от 60 до 132 (по подразбиране е 80) и задава броя на позициите в реда.

Тези размери се назначават от страницата, в която е срещната директивата, нататък. Например:

```
PAGE  100, 60        ; 100 реда с 60 позиции
PAGE  , 80           ; 80 позиции (броя редове не е променен)
```

На всяка страница (в дясната част на първия ред) на листинга се показва нейният номер във формат (ss-pp), където ss е номер на секция, а pp е номер на страницата в секцията.

Вторият вариант на директивата (PAGE +) означава преход към нова секция и нова страница от листинга, при това номерът на новата страница се увеличава с 1, а номерът на страницата става равен на 1. Какво да се приема за секция и кога да се променя номерът на секцията – това решава авторът на програмата.

Третият вариант на директивата означава принудителен преход към нова страница, чийто номер се увеличава с 1 и без промяна на номера на секцията.

Директива .XLIST

Чрез тази директива се прекратява формирането на листинга и всички следващи редове от програмата не попадат в него.

Директива .LIST

Тази директива отменя действието на директивата .XLIST, т.е. възстановява формирането на листинга, започвайки от нея. Директивата .LIST се подразбира в началото на програмата. Пример:

```
.XLIST
INCLUDE  LISTS.ASM     ; текстът на този файл не се включва
.LIST
```

Директива .SALL

След среща на тази директива, в листинга няма да бъдат записвани макроразширенията и копията на блоковете за повторение (в листинга ще попадна само макрокомандите и изходните текстове на блоковете).

Директива .XALL

След среща на тази директива, в листинга ще се записват макроразширенията и копията на блоковете за повторение, но в тях ще се съдържат само тези редове, по които Асемблер генерира командите и данните (редовете с коментар, директивите ASSUME и EQU ще липсват). Този режим е установен първоначално по подразбиране.

Директива .LALL

Тази директива отменя директивите .SALL и .XALL, като след срещането й в листинга ще попадат всички изречения на всички макроразширения, както и текстовете на всички копия на блокове за повторение.

Директива .SFCOND

Тази директива подтиска записването в листинга на изреченията от клоновете "лъжа" на блокове IF. Този режим се установява първоначално по подразбиране.

Директива .LFCOND

Тази директива отменя действието на директивата .SFCOND, т.е. възстановява записа в листинга на всички клонове на блоковете IF.

14.5 Директиви за контрол върху работата на Асемблер

Ще разгледаме някои от директивите, позволяващи в известна степен да се контролира процеса на транслация.

14.5.1 Директива %OUT

Срещайки при транслиране в програмата директивата

```
%OUT  <текст>
```

Асемблер незабавно извежда на екрана на нов ред посочения в директивата текст. Ако директивата се намира във макроопределение или във блок за повторение, то в нейния текст могат да се запишат формалните им параметри, които преди отпечатването ще бъдат подменени с фактическите параметри. В машинната програма тази директива не попада.

Подобни отпечатъци са полезни за проследяване на процеса на транслиране. Например, ако имаме макроопределението:

```
EX    MACRO  X
      %OUT  Обръщение:  EX  X
      INC  X
      ENDM
```

то, ще кажем, по команда EX SI ще бъде формирано макроразширението:

```
%OUT  Обръщение:  EX  SI
INC  SI
```

по което Асемблер извежда на екран текста "Обръщение: EX SI" , а в машинната програма ще запише само командата INC SI.

14.5.2 Допълнителни IF-директиви

Недостатък на директивата %OUT е това, че тя "сработва" два пъти, защото Асемблер преглежда текстът на програмата два пъти. Това обстоятелство се обяснява с трудностите, които изпитва Асемблер при транслация на обръщенията напред. Работата се състои в това, че ако Асемблер, разглеждайки текста на програмата от началото, срещайки

име, което още не е описано, той няма как да знае какво означава това име и от там как да го транслира. За да се решат проблемите с обръщенията напред, Асемблер разглежда текста на програмата два пъти – често се говори, че той прави два паса. При първия пас се събира информация за имената (тип, адрес, обръщение и пр.), а при втория пас, използвайки тази информация, се съставя машинния код на програмата.

За тази технология можем и да не знаем, но понякога тя е полезна и даже необходима. Например, ако чрез директивата %OUT е необходимо само един път да се изведе текстът на екрана (само при първия или само при втория проход), тогава следва да се възползваме от директивите на условното асемблиране IF1 или IF2.

Условието на директивата IF1 се приема за изпълнено, когато Асемблер изпълнява своя първи проход по текста на програмата, и се приема за неизпълнено по време на втория проход. Случаят с директивата IF2 е точно обратен.

По отношение на имената, условните директиви са:

IFDEF <име> , IFNDEF <име>

По време на първия проход условието на директивата IFDEF се счита за изпълнено, ако указаното име е вече описано в програмата. И се приема за неизпълнено, ако описание на името още не е срещано. При втория проход условието на директивата се приема за изпълнено, ако е било срещнато описание на името. Случаят с втората условна директива е аналогичен, но обратен.

Имената се приемат за описани, ако са указани в лявата част на команди и директиви, а също имена на полета в структури и записи, имена на регистри.

Ето пример за използване на условно деклариране:

```
     IFDEF  N
S    DB  N  DUP (?)
     ELSE
S    DB  256  DUP (?)
     ENDIF
```

В окончателния текст на програмата са възможни два варианта – името S, описано като име на масив от N байта, или описано като масив от 256 байта.

14.5.3 Условно генериране на грешки

При транслиране на програмата могат да възникнат ситуации, в които от гледна точка на езика, текстът на програмата е правилен, но от гледна точка на логиката й това да не е правилно, а това би могъл да установи само автора на програмата. За да могат да се фиксират такива грешки в Асемблер са въведени директиви за условно генериране на грешки. Всяка една такава директива проверява някакво условие и, ако то е изпълнено, записва в листинга съобщение за прину-

дителна грешка (*forced error*). Разбира се, обектен файл при грешка не се създава.

За следните директиви за условно генериране на грешка се показва следния отпечатък в листинга:

```
.ERR1        87 Forced error – pass 1
.ERR2        88 Forced error – pass 2
.ERR         89 Forced error
```

Чрез директивата може да се разграничи описанието на дадено име. Например:

```
IFNDEF  X            ; ако името X не е описано и
  .ERR2              ; ако това е вече 2-ри пас,
ENDIF                ; тогава фиксирай грешка
```

Директивите:

```
.ERRE  <израз>        90 Forced error – expression equals 0
.ERRNZ  <израз>       91 Forced error – expression not equals 0
```

генерират грешка, ако стойността на израза е равна на 0, или не е равна на 0. Изразът трябва да е константен, да не съдържа външни имена и обръщения напред. Например:

```
.ERRE  TYPE  L – NEAR     ; грешка, ако L е близък етикет
```

Директивите:

```
.ERRNDEF  <име>       92 Forced error – symbol not defined
.ERRDEF  <име>        93 Forced error – symbol defined
```

генерират грешка, ако указаното име все още не е описано, или вече е описано. Ако името представлява обръщение напред, то при първия проход транслаторът го приема за неописано, а при втория – описано. Например:

```
.ERRNDEF  N           ; грешка, ако името N още не е описано
Y  DB  N DUP (?)
```

Директивите:

```
.ERRB  <текст>                   94 Forced error – string blank
.ERRNB  <текст>                  95 Forced error – string not blank
.ERRIDN  <текст1>,<текст2>       96 Forced error – string identical
.ERRDIF  <текст1>,< текст2>      97 Forced error – string different
```

Първата директива генерира грешка, ако указаният текст е празен, а втората – ако не е празен. Третата директива генерира грешка, ако указаните текстове съвпадат, а четвъртата – ако не съвпадат. Тук под текст се разбира последователност от символи затворени в ъглови скоби. Ако някоя от тези директиви се намира в макропределение и в блок за повторение, то в текстовете могат да се укажат формалните параметри, които преди сравнението ще бъдат заменени с действителните параметри.

Например:

```
    M  MACRO  X, Y
       .ERRB  <X>              ;; грешка, ако 1-ия парам. е изпуснат
       .ERRIDN  <Y>, <CS>      ;; грешка, ако 2-ия парам. е CS
       ....  ....  ....
       ENDM
```

14.6 Допълнителни директиви

Системата машинни команди на микропроцесор 8086 се явява базова за всички следващи модели на този процесор. В следващите модели обаче базовият набор е допълнен с други машинни команди. Без да изпадаме в подробности, тук това не целта, ще кажем, че по подразбиране Асемблер (версия 4.0) допуска за изпълнение само базовия набор команди. За включване на допълнителните команди на съответния модел микропроцесор са разработени директиви. Които се задействат веднага след тяхната среща до следваща подобна директива. Тези директиви имат вида: .8086, .186, .286, .8087, .287 и пр. т.е. те представляват номера на модела.

14.6.1 Групи от сегменти

За обединяване на сегменти в една група се прилага директивата:

<име на групата> GROUP <име на сегмент {, <име на сегмент>}

Това означава, че всички имена от обявените сегменти, ще се сегментират относно един и същи сегментен регистър. По кой именно, определя директивата ASSUME, ако в нея е указан операндът

SR: <име на група>

където SR означава име на някакъв сегментен регистър. Всяко име от обединените сегменти Асемблер ще заменя с адресната двойка [SR:Offset], в която указаното отместване ще се измерва от началния адрес на групата. Зареждането на този начален адрес в избрания сегментен регистър SR следва да извърши самата програма. Това адресиране се извършва, даже ако името от някакъв сегмент в групата е указано със сегментен префикс.

Обединяването на сегментите и отчитането на отместванията спрямо един сегментен регистър не означава, че сегментите са разположени плътно един след друг в паметта. Тук предимство има подредбата в директивата SEGMENT. Това, което не може обаче да се промени, е обемът на сегмента на групата, който не може да надхвърли 64[KB].

Името на групата следва да бъде уникално за програмата, а сегментите от групата могат да бъдат описани в текста на програмата както преди, така и след директивата GROUP. Името на групата се определя като константен израз, чиято стойност се изчислява по аналогичен начин както и името на сегмент.

Пример:

```
        GR    GROUP  S1, S2
        S1    SEGMENT                          ; offset S1 спрямо GR = 0
                 A   DB  20h DUP (?)           ; offset  A  спрямо S1 = 0
        S1    ENDS
        S2    SEGMENT                          ; offset S2 спрямо GR = 20h
                 B   DW  1                     ; offset  B  спрямо S2 = 0
        S2    ENDS
     CODE     SEGMENT
              ASSUME  CS:CODE, ES:GR
                 MOV  AX, GR
                 MOV  ES, AX                   ; (ES)=GR
                 MOV  DH, A             ; еквивалентно на MOV DH,ES:0
                 MOV  CX, B             ; еквивалентно на MOV  CX, ES:20h
                 ....  ....  ....
```

Ще отбележим непоследователната работа на Асемблер при работа с групи. Ако във всички команди имената от сегментите на групата се заменят на отместването им спрямо началото на групата, то стойността на оператора (OFFSET B) представлява отместване на името B спрямо началото на сегмента, в който е описано, даже този сегмент и да влиза в групата. Така в горния пример (OFFSET B)=0. За да отчетем отмест ването на B спрямо началото на групата този оператор трябва да бъде записан с името на групата така: OFFSET GR:B. Тогава (OFFSET GR:B) =20h. Аналогичен е проблемът с описанието на адресни константи в директивите DW и DD. Ако като техен операнд е записано само името, то отместването му ще бъде определено спрямо началото на сегмента, в който е описано. За отместване спрямо началото на групата името трябва да се опише с конструкцията

<име на група>:<име>

14.6.2 Промени в брояча за разполагане

Транслирайки програмата, Асемблер следи за адреса на поредния ред от програмата, който се съхранява в брояча за разполагане. Стойността на този брояч може да се получи в програмата чрез символа $. Следващите директиви позволяват да се променя стойността на този брояч.

Директивата EVEN изравнява брояча за разполагане със стойността на най-близкия четен адрес. Ако текущата стойност на брояча е четна, стойността му не се променя, в противен случай в оставащия “празен” байт Асемблер записва 90h, което е кода на машинната команда NOP и увеличава стойността на брояча с 1.

<u>Директива</u> ORG: <израз>.

ORG: <израз>.

Изразът може да бъде константен или адресен, но всички използвани в него имена трябва да са от текущия сегмент, при това да са били описани до директивата. Освен това, елемент на израза може да бъде означението ($) на брояча за разполагане. Стойността на този израз се присвоява като нова стойност на брояча за разполагане (в пропуснатите байтове не се записва нищо).

Пример:

```
        ORG  100h
        MOV  AX, 0          ; тази команда се помества в адрес 100h
                            ; на текущия сегмент
        ORG  $+20           ; пропускайки 20 байта
```

Ако директивата ORG е причинила намаление на стойността на брояча за разположение, тогава поредния ред от програмата се "налага" върху предидущото съдържание на този адрес.

14.6.3 Други директиви

1. Директива LABEL:

<име> LABEL <тип>.

Тази директива е еквивалентна на следната:

<име> EQU THIS <тип>

и определя името, на което се приписва текущият адрес (стойността на брояча за разположение $) и указаният тип. Името трябва да е уникално за програмата. Допустимите типове са: BYTE, WORD, DWORD, NEAR, FAR. Директивата LABEL се използва, когато желаем да се обръщаме към следващата след нея команда или променлива с име от друг тип, освен разрешеното съгласно описанието на командата или на променливата.

Например:

```
        FL  LABEL  FAR      ; FL и NL маркират една и съща команда
   NL:  MOV  AX, 0          ; но FL е далечен етикет, а NL – близък
```

2. Директива .RADIX <константен израз>

В Асемблер числата могат да бъдат записвани като изобразени в различни бройни системи: 16, 10, 8, 2 (10h, 10d, 10q, 10b). Ако специфициращата буква в края на записа липсва, то се приема, че записаното число е в 10-чна бройна система.

Тези правила могат да бъдат променени с помощта на директивата RADIX. Стойността на нейния операнд показва основата на бройната система, в която ще се възприемат записите на числа без специфицираща буква в своя край. Разбира се стойностите на константния израз в директивата не могат да бъдат други, освен изброените.

Примери:

```
.RADIX  16              ; назначава 16-чна
    MOV  AX, 10         ; AX:=10h=16
    MOV  AX, 10d        ; AX:=10
    MOV  AX, 10Dh       ; AX:=10Dh=269
.RADIX  10              ; назначава 10-чна
    MOV  AX, 10         ; AX:=10
    MOV  AX, 10h        ; AX:=10h=16
```

3. Директива NAME <име на модул>

Чрез тази директива указаното име (вземат се само първите 6 символа) се присвоява на текущия модул на програмата и се използва от свързващия редактор в диагностичните съобщения при откриване на грешки при обединяване на модулите на програмата. Името се изписва в скоби след името на файла с дадения модул. Например, ако в даден модул се съдържа директивата NAME MOD1 и ако транслацията на модула се съхранява от файла PROG.OBJ, тогава е възможно съобщение за грешка на етапа свързване, относно името DDD, указано в дадения модул, но не описано като общо в другите модули. Съобщението ще има вида:

Unresolved externals: DDD in file(s) PROG.OBJ(MOD1)

В модула не следва да има повече от една директива NAME. Ако такава директива липсва, в качеството на име на модула ще бъдат взети първите 6 символа от текста, указан в директивата TITLE, а ако и тя липсва, на модула се присвоява името A.

Разработка на асемблерни програми с Borland Турбо Асемблер (TASM)

Процесът на създаването на програма на Асемблер слабо се отличава от традиционния подход при използване на съществуващите езици за програмиране. Той включва постановка на задачата, получаване на първите резултати и по-нататъшната поддръжка на програмата. Основните моменти на този процес са:

1. Постановка и формулиране на задачата:
    - Изучаване на предметната област и събиране на материал за проблема;
    - Определяне целта на програмата, изработване изискванията към нея и при възможност представянето им във формализиран вид;
    - Формулиране на изискванията към представянето на входните данни и крайните резултати;
    - Определяне на структурата на входните и изходните данни;
    - Формиране на ограничения и допускания за входните и изходните данни.
2. Етап на проектиране:
    - Формиране на ”асемблерен” модел на задачата;
    - Избор на метод за решаване на задачата;
    - Разработване на алгоритъм за решаване на задачата;
    - Разработване на структурата на програмата въз основа на избрания модел на паметта.
3. Етап на кодиране:
    - Уточняване на структурата на входните и изходните данни и определяне на формата на представяне в асемблера;
    - Програмиране на задачата;
    - Коментиране на текста и съставяне на предварително описание на програмата.
4. Етап на настройка и тестване:
    - Съставяне на тестове за проверка на работоспособността на програмата;
    - Намиране, локализация и отстраняване на грешките в програмата, намерени при тестването;
    - Коригиране на кода на програмата и нейното описание.
5. Етап на експлоатация и поддръжка:
    - Настройка на програмата към конкретните условия на потребителя;
    - Обучение на потребителите за работа с програмата;
    - Организиране на събиране на сведения за бъгове на програмата, грешки в изходните данни, предложения за

усъвършенстване на интерфейса и лекотата при работа с програмата;

- Модификация на програмата с цел отстраняване на намерените грешки и при нужда промяна на функционалните и възможности.

Редът и обемът на извършваните действия от горния план зависи от особеностите на конкретната задача, нейното предназначение, размера на кода и обработваните данни, както и от другите характеристики на изходната задача. Някои етапи могат да се изпълняват заедно с други, или изобщо да се изпуснат. Целта е начално организиране на процеса на създаването на нов програмен продукт, като се запазва постановката на задачата и се премахва анархията от процеса на разработка.

В глава 12 е описано програмирането с макроасемблера MASM на фирмата *Microsoft*. Стандартният фирмен екранен дебъгер (програмата за настройка) ***CodeView*** работи под MS-DOS единствено в 16-битов режим. Той е създаден и се предлага оригинално включен в пакета *Microsoft* C 4.0 и по-високи версии. От *Visual* C++ 1.0 нататък той е интегриран в развойни среди за PC като *Quick* C, *Visual Basic* за MS-DOS и др.. Днес *Microsoft* го включва като интегрална част от фамилията *Microsoft Visual Studio*. Под *Windows* в конзолен режим може да се използва предшественика на ***CodeView***, дебъгера ***Debug***, който е редови и има ограничени възможности.

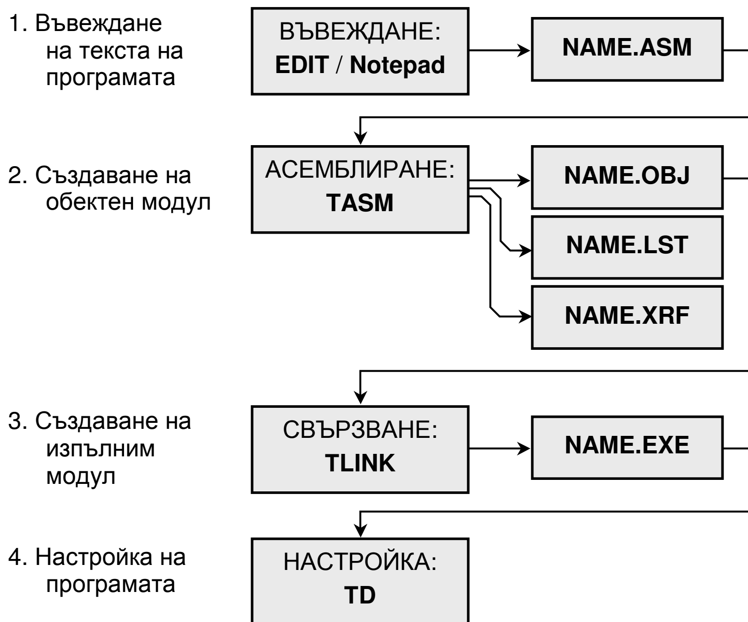
Значително по-удобно е използването на екранния Турбо дебъгер **TD** на *Borland*, който заедно с транслатор на Асемблер и линкер е включен в програмния пакет *Borland* C++ версии 3-5 и се инсталира автоматично при инсталиране на C пакета. Турбо Асемблерът на *Borland* (TASM) може да асемблира MASM асемблерна програма и работи безпроблемно под MS-DOS и Windows. Той може да компилира освен в MASM и в ***ideal***-режим, който предлага известни усъвършенствания. По подразбиране TASM се стартира в MASM-режим.

Фигура 1 показва схематично четирите етапа на програмирането на асемблерна задача. Първо изходният текст на програмата се въвежда с помощта на текстов редактор, който не добавя допълнителни символи към текста (напр. специални символи за форматиране). Под ОС *Windows* това може да се направи с текстовия редактор *Notepad*. Ако се работи в DOS-прозорец, може да се използва вградения текстов редактор EDIT. Създаденият изходен файл с програмата трябва да бъде записан на диска като MS-DOS файл с име не по-дълго от 8 символа и с разширение ***.ASM***. На фиг. 1 този файл има име *NAME*.

Вторият етап е транслация на програмата или асемблиране (*Assembling*). На този етап записания на диска файл с изходния текст на асемблерната програма се преобразува от Турбо Асемблера в междинен обектен файл.

При транслацията:

- Командите на асемблера се превеждат на машинен език;
- Построяват се съответните символни таблици;
- Разширяват се макросите;
- Формират се обектен файл ***NAME.OBJ***, листингов файл ***NAME.LST*** и файл на кръстосаните обръщения ***NAME.XRF***.



Фиг. 1 Схема на програмните дейности

Обектният файл включва машинен код на изходната програма и друга информация, която е необходима за настройката на програмата и свързването и с други модули. Обикновено файловете на TASM се намират в работната папка C:\BORLANDC\BIN. Файловете с асемблерните програми трябва да се записват в нея.

Когато файловете от пакета TASM се извикат с опция ***?***, те извеждат информация за параметрите на командата. Квадратните скоби заграждат незадължителните параметри.

Извикването на Асемблера от командния ред в работната папка става с командата:

TASM *[/опции] [изходен файл] [,обектен файл] [,файл на кръстосаните обръщения]*

Ако в командата не са записани имената на съответните файлове, Асемблерът няма да ги създаде. Задължително е само задаването на

името на изходния файл, който задължително трябва да има разширение *.ASM*. Много е удобно да се използват еднакви имена за обектния файл, файла с листинга от транслацията и файла на кръстосаните обръщения, съвпадащи с името на изходния файл. Тогава в командата вместо имената се поставят запетаи. Транслаторът ще включи за дебъгера пълна информация от изходната програма, ако се използва опцията ***/zi***. В този случай транслирането на асемблерната програма с име *NAME.ASM* става с командата:

TASM /zi name,,,

и създава на диска файловете *NAME.OBJ*, *NAME.LST*, *NAME.XRF*.

Ако асемблерната програма има грешки, транслаторът ще изведе на екрана съобщения, започващи с думите "Error" и "Warning". Съобщението за синтактични грешки има вида:

Error *NAME.ASM* (*Line number*) *Error message*

То означава, че в асемблерния файл *NAME.ASM* на ред с номер *Line number* е намерена грешка от вида *Error message*.

Предупреждението, което говори за синтактически правилна конструкция, но за несъответствия с някои правила на езика, които биха породили проблеми в бъдеще, има вида:

Warning *NAME.ASM* (*Line number*) *Warning message*

То означава, че в асемблерния файл *NAME.ASM* на ред с номер *Line number* е намерена грешка от вида *Warning message*.

- Грешките са фатални за компилацията. Ако бъде създаден обектен файл, той няма да може да се обработи от линкера или няма да се стартира.
- Предупрежденията не са фатални. Създаденият обектен файл вероятно ще може да се обработи от линкера. Създаденият изпълним файл може да се стартира, но може да не се изпълнява коректно.

За отстраняването на грешките трябва да се определи мястото на възникването на грешката и вида и. Добре е с текстов редактор да се разгледа файла с листинга от транслацията *NAME.LST*. Този обикновен текстов файл съдържа резултатите от транслацията. В него съобщенията за грешки се намират веднага след линията, където е открита грешката. Листинговият файл е много важен. Той съдържа асемблерния код на изходната програма, машинния код и отместването в кодовия сегмент за всяка инструкция. В края на листинга има таблица с информация за етикетите и сегментите, използвани в програмата. Освен това в края на този файл са включени съобщенията за грешки и предупреждения, така както се извеждат на екран, а също и съобщения за съмнителни участъци от кода.

След отстраняване на грешките от транслацията и получаването на обектен модул се преминава към създаване на преместваем, заредим и

изпълним модул. Форматът на обектния файл позволява при определени условия да се обединят в един модул няколко отделно транслирани обектни модули, написани на един и същ или различни езици. Свързващия редактор (*Linker*) създава изпълним файл с разширение .exe. Операционната система може да го зареди в паметта и да го изпълни. Свързването (*Linking*) се извършва с командата:

TLINK *[/опции] обектни файлове [,изпълним файл] [,файл карта] [,библиотечен файл] [,файл дефиниции] [,ресурсен файл]*

В командата обектните файлове са изредените имена на обектни файлове с разширение .obj, разделени с интервал или знак плюс. При нужда може да се указват и пътища. Опцията ***/v*** позволява да се съхрани информацията за настройката в изпълнимия файл. Най-често има само един обектен модул ***NAME.OBJ***. Тогава командата ще изглежда така:

TLINK /v *NAME*

Резултатът от работата на програмата е изпълним файл ***NAME.EXE***. Липсата на синтактични грешки обаче не гарантира, че програмата ще работи правилно, макар че се стартира. Затова задължително се прави тестване и настройка на програмата. При настройката се проверява дали отделните фрагменти на кода и програмата като цяло функционират правилно в съответствие с алгоритъма. Дори и правилното функциониране на програмата не гарантира обаче коректната работа на програмата при различни изходни данни. За такава проверка на програмата се създават тестове. Ако резултатите от тях не удовлетворяват програмиста, кодът на програмата се поправя и отново се минават етапите на транслация от началото до края.

Настройката (*Debugging*) на програмата става с автономния Турбо дебъгер на *Borland* ***TD***. С негова помощ програмистът може да определи мястото на логическа грешка в програмата и причината за възникването и. Това става като програмата се трасира (изпълнява се стъпка по стъпка в права или обратна посока). По време на трасировката може да се проследи и измени състоянието на апаратните ресурси на процесора. Когато бъде намерена грешка е възможно да бъде коригиран машинния код направо в средата на дебъгера. Това обаче не променя изходния файл на програмата, тъй като промените не се съхраняват на диска. Ето защо обикновено се коригира изходния текст на програмата, извършва се нова транслация и настройката се повтаря.

За правилното протичане на настройката е необходимо в изходния файл с програмата на Асемблер да се укаже с етикет оператора на програмата, от който ще започне изпълнението и. Това е входната точка на програмата. Същият етикет задължително трябва да присъства в операндното поле на последния оператор на програмата:

END *входна_точка*

Ако изходния файл е транслиран с опция ***/zi***, транслаторът включва в програмата информация за символичните имена и отместването им в съответните сегменти, което позволява при настройката дебъгерът да ползва оригиналните имена. Свързването трябва да се извърши с опция ***/v***, за да се съхрани информация за настройването в получавания изпълним файл. Извикването на дебъгера става от командния ред с указване на името на изпълнимия модул:

TD *име_на_изпълним_файл*

Пример за асемблиране и свързване

Нека въведем с текстов редактор асемблерна програма, която кара обикновен матричен принтер да премести хартията на следваща страница като изпраща към принтера управляващи кодове. Нека я запишем на диска под име *FORMF.ASM*.

```
  STK    SEGMENT  STACK
              DW    256 DUP(?)
  STK    ENDS
DATA     SEGMENT
              ASCCR   EQU  0Dh                ;Ascii carriage return
              ASCFF   EQU  0FFh          ;Ascii form feed control code
DATA     ENDS
CODE     SEGMENT
              ASSUMECS:CODE, DS:DATA, SS:STK
   START:  MOV  AX, DATA                      ;Initilize DS to address
              MOV  DS, AX                     ; of data segment
              MOV  DL, ASCCR                  ;Assign cr code to dl
              MOV  AH, 5                   ;DOS function: Printer output
              INT    21h                    ;Call DOS - carriage return
              MOV  DL, ASCFF                ;Assign ff code to dl
              MOV  AH, 5                   ;DOS function: Printer output
              INT    21h                    ;Call DOS – form feed
     EXIT:  MOV  AX, 04C0h                  ;DOS function: Exit program
              INT    21h                 ;Call DOS. Terminate program
CODE     ENDS
   END   START                           ;End of program /entry point
```

Ако програмата е въведена коректно, след командата, изпълнена от DOS prompt:

TASM ***/zi*** *FORMF,,,*   транслаторът извежда на екрана:

```
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International
Assembling file:   formf.ASM
Error messages:    None
Warning messages:  None
Passes:           1
Remaining memory:  453k
```

На диска се създава обектен файл с име *FORMF.OBJ*, листингов файл с име *FORMF.LST* и файл на кръстосаните обръщения *FORMF.XRF*. Командата, изпълнена от DOS prompt:

TLINK /v *FORMF* извиква линкера, който извежда на екрана:

```
Turbo Link  Version 5.1 Copyright (c) 1992 Borland International
```

На диска се създава коректен изпълним файл с име *FORMF.EXE* и картов файл с име *FORMF.MAP*.

Грешки при транслацията

Независимо, че програмистът внимава, нормално е да допусне грешки. Асемблерът обаче не може да бъде заблуден. Той не може да бъде заставен да приеме неправилна конструкция. Ако се направи такъв опит (умишлено или не), той ще изведе при транслирането съобщение за грешка, предупреждение или и двете. Нека направим една умишлена грешка във файла *FORMF.ASM* и после да запишем коригирания файл под име *FORMFE.ASM*. Да сменим в директивата dw на втория ред буквата w с v, така че да стане dv. Асемблира се с ***TASM FORMFE***. На екрана се получава:

```
Assembling file:   formfe.ASM
**Error** formfe.ASM(2) Illegal instruction
Error messages:    1
Warning messages:  None
Passes:            1
Remaining memory:  453k
```

Съобщението за грешка след „*Assembling file…*" указва в кой файл е намерена грешката, в скоби е номера на линията където е грешката и следва сбитото описание на грешката. Нека направим друга умишлена грешка във файла *FORMF.ASM*. Да изтрием двоеточието след етикета Start и го запишем под име *FORMFE.ASM*. Извежда се:

```
Assembling file:   formfe.ASM
**Error** formfe.ASM(10) Illegal instruction
**Error** formfe.ASM(21) Undefined symbol: START
Error messages:    2
Warning messages:  None
Passes:            1
Remaining memory:  453k
```

Макар че е направена само една грешка, Турбо Асемблерът извежда 2 съобщения за грешки, едно на линия 10 поради липсващото двоеточие, а другото на линия 21, което указва етикета Start. Тъй като първата грешка прави етикета Start неразпознаваем заради липсващото двоеточие, входната точка Start в програмата също не се разпознава. Това е пример за *разпространяваща се грешка*: една грешка причинява други или води до разпространяването им. В голяма програ-

ма е възможно малко на брой грешки да се разпространят навсякъде. Ако това се случи внезапно в секция от програмата, която преди това се е асемблирала без проблеми, коригирането само на първите няколко грешки и повторното асемблиране често е достатъчно за премахването на грешките.

Нека във файла *FORMF.ASM*, като линия 10, вмъкнем един ред с две думи преди етикета Start:

PROC DUMMY

Да премахнем двоеточието след етикета Exit и после да запишем коригирания файл под име *FORMFE.ASM*. Асемблирането дава:

```
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International
Assembling file:   formfe.ASM
**Error** formfe.ASM(19) Illegal instruction
*Warning* formfe.ASM(22) Open procedure: DUMMY
Error messages:    1
Warning messages:  1
Passes:            1
Remaining memory:  453k
```

Предупреждението, изведено от Асемблера, указва нещо некоректно на линия 22, а не на линия 10, както се очаква. Директивата PROC указва начало на процедура - група инструкции, която се третира от програмата като завършена подпрограма. Турбо Асемблерът очаква всички директиви PROC да бъдат следвани от ENDP (End Procedure) директиви. Тъй като не намира подобна директива преди достигането на края на програмата, асемблерът предупреждава, че процедурата е оставена отворена. Съобщението е предупреждение, а не грешка.

Формат на листинга

Листингът от транслацията, файлът *FORMFE.LST*, се извежда на екран с команда

TYPE *FORMFE.LST*

и изглежда така:

```
Turbo Assembler Version 3.1        08/26/09 09:11:42            Page 1
FORMFE.ASM
     1  0000                    STK     SEGMENT  STACK
     2  0000  0100*(????)           DW      256 DUP(?)
     3  0200                    STK     ENDS
     4  0000            DATA    SEGMENT
     5       =000D          ASCCR   EQU     0Dh         ;Ascii carriage return
     6       =00FF          ASCFF   EQU     0FFh        ;Ascii form feed control code
     7  0000            DATA    ENDS
     8  0000            CODE    SEGMENT
     9                 ASSUME  CS:CODE, DS:DATA, SS:STK
    10  0000            PROC    DUMMY
```

```
     11  0000  B8 0000     Start:  MOV    AX, DATA          ;Initilize DS to address
     12  0003  8E D8               MOV    DS, AX            ; of data segment
     13  0005  B2 0D               MOV    DL, ASCCR         ;Assign cr code to dl
     14  0007  B4 05               MOV    AH, 5       ;DOS function: Printer output
     15  0009  CD 21               INT     21h          ;Call DOS - carriage return
     16  000B  B2 FF               MOV    DL, ASCFF            ;Assign ff code to dl
     17  000D  B4 05               MOV    AH, 5       ;DOS function: Printer output
     18  000F  CD 21               INT     21h                 ;Call DOS - form feed
     19                    Exit    MOV    AX, 04C0h   ;DOS function: Exit program
**Error** formfe.ASM(19) Illegal instruction
     20  0011  CD 21               INT    21h          ;Call DOS. Terminate program
     21  0013          CODE        ENDS
     22                    END     Start                ;End of program /entry point
*Warning* formfe.ASM(22) Open procedure: DUMMY
Turbo Assembler Version 3.1       08/26/09 09:11:42                      Page 2
Symbol Table
Symbol Name     Type   Value         Cref  (defined at #)
??DATE          Text   "08/26/09"
??FILENAME      Text   "formfe      "
??TIME          Text   "09:11:42"
??VERSION       Number 030A
@CPU            Text   0101H
@CURSEG         Text   CODE                   #1  #4  #8
@FILENAME       Text   FORMFE
@WORDSIZE          Text   2                   #1  #4  #8
ASCCR           Number 000D                   #5  13
ASCFF           Number 00FF                   #6  16
DUMMY           Near   CODE:0000              #10
START        Near   CODE:0000                 #11  22
Groups & Segments Bit Size Align Combine Class    Cref        (defined at #)
CODE         16  0013 Paranone                #8  9
DATA         16  0000 Paranone                #4  9  11
STK          16  0200 ParaStack               #1  9
Turbo Assembler Version 3.1       08/26/09 09:11:42                      Page 3
Error Summary
**Error** formfe.ASM(19) Illegal instruction
*Warning* formfe.ASM(22) Open procedure: DUMMY
```

Файлът с листинга включва изходния текст на програмата. За всяка команда Асемблерът указва машинния (обектния) и код и отместването му в кодовия сегмент, както и отместването на променливите от началото на сегмента, в който са разположени. Освен това в края на листинга TASM формира таблица с информация за етикетите и сегментите, използвани в програмата. Ако има грешки или съмнителни участъци от кода, TASM включва в края на листинга съобщения за тях.

Тези съобщения съвпадат с изведените на екрана. Тези съобщения се включват в текста на листинга непосредствено след сгрешения ред. Редовете във файла с листинга имат следния формат:

Вложеност Ред_№ Отместване Маш. код Изх. код

Петте полета имат следния смисъл:

• Полето **Вложеност** показва дълбочината на вложеност на включваните файлове или макрокоманди;

• Полето **Ред_№** е номер на реда във файла с листинга. Този номер се ползва за да се локализира мястото на грешката и при формирането на таблицата на кръстосаните обръщения. Номерът на реда във файла с листинга може да не съответства на номера на реда в изходния файл. Това се отнася например при използване на директивата INCLUDE, която включва във файла редовете на друг файл. Номерацията в този случай, както и при наличие на макрокоманди, ще бъде последователна за редовете на двата файла.

Вложеността на кода на един файл в друг се указва чрез увеличаване с 1 на полето **Вложеност**. Това става при макрокоманди.

• Полето **Отместване** е отместването в байтове на адреса, по който се разполага генерирания код на текущата команда относно началото на кодовия сегмент. Това отместване се нарича адресен брояч. То се изчислява и използва от транслатора за адресация в кодовия сегмент;

• Полето **Маш. код** е машинното представяне на командите на Асемблера, които са записани по-нататък на този ред;

• Полето **Изх. код** е ред от кода на изходния файл с асемблерната програма.

Може да се забележи, че не всички редове на програмата имат съответен машинен код. Някои директиви на асемблера не генерират код. Макрокомандите след обработката от транслатора генерират в обектния модул последователност от команди, директиви или макрокоманди на Асемблера. Форматът на листинга не е твърдо фиксиран и може да се променя с директивите на Асемблера за управление на листинга. В последната програма *FORMFE.ASM* несъществуващата процедура DUMMY не поврежда генерирания машинен код. Това обаче не винаги е така. Опасно е да се игнорират предупрежденията на TASM. Например липсващата ENDP във файла може да бъде получена при случайно изтриване на линията или оставяне по погрешка незавършена процедура. TASM понякога пропуска такива опасности, позволявайки в някои случаи да се направят големи грешки. Асемблерът е достатъчно “умен” да съобщава за потенциални опасности, но познаването на същината на програмата е единственият начин да се разбере къде предупрежденията са важни и къде безопасно могат да се пренебрегнат.

След като се прочетат съобщенията на Асемблера и се поправят синтактичните грешки, програмата може да се транслира без проблемно. Асемблерът обаче не разбира какво трябва да прави тя. Често програмите не правят това, което програмистът мисли, че ще правят. За да се открият и поправят логическите грешки се използва Турбо дебъгера. Като всички дебъгери той поема управлението на програмата и позволява да се проверяват стойностите на променливите в паметта и кодът да се изпълнява бавно. На дебъгера може да се зададе да изпълнява програмата до зададена точка или докато възникне зададено събитие. Може да се променят стойности в паметта, временно да се пробват нови инструкции и да се променят флагове и регистри на микропроцесора. Турбо дебъгерът може да се ползва за да се изпробва някаква идея при програмиране в машинен код на малък брой инструкции. Той позволява да се наблюдава ефекта от изпълнението на различни инструкции. Един начин за това е да се напише къса тестова програма, да се транслира и зареди в Турбо дебъгера и после да се разглеждат резултатите при бавно изпълнение.

Кирилица в програмите на Асемблер

Когато коментарите и символните данни в асемблерната програма са само на латиница, въвеждането на текста и редактирането на асемблерната програма може да се направи с произволен текстов редактор, работещ под DOS или Windows. Използването в асемблерните програми на символи на кирилица може да породи проблеми. Асемблерът преобразува тези символи в еднобайтови разширени ASCII кодове според активната в момента кодова страница (Code Page) в конзолния режим. Текстовият редактор EDIT изобразява кирилицата безпроблемно, но не позволява въвеждането и от клавиатурата. Приложенията на Windows XP поддържат няколко различни представяния (кодови таблици) на символите на кирилицата. Ето защо въвеждането и редактирането на асемблерна програма с кирилица трябва да се извърши с текстов редактор под Windows, а после да се продължи с транслиране и настройка на програмата в DOS прозорец. Изходният текст на програмата се въвежда с текстовите редактори Notepad или WordPad. При записа и в текстов файл (напр. *NAME.TXT*) в диалоговия прозорец “Save As” от падащия списък с възможното представяне (кодиране) на символите от текста във файла трябва да се избере представяне в Unicode и файла да се запише в работната папка. По този начин всеки символ от програмата във файла се замества (кодира) с 2 байта. Преминава се в DOS-прозорец. Работната папка се прави текуща. За дисплея трябва да се установи активна кодова таблица с кирилица. Това става с командата:

MODE *con cp select**=xxxx*** ,

където ***xxxx*** замества номера на кодовата таблица. Ако *Windows* е настроен правилно, кодовите таблици с номер ***866*** (при системна

страна с кирилица, но не *Serbian Cyrillic*), с номер ***855*** (при *Serbian Cyrillic* за системна страна), или с номер ***1251*** може да се ползват за показването на кирилица в DOS прозорец. Трябва да се намери чрез проби при коя кодова таблица от трите изброени текстът на програмата на кирилица се изписва на екрана без промени при извеждането му в DOS прозореца с командата:

TYPE *NAME.TXT* .

Добра идея е да се изведе активната кодова таблица с разширените ASCII-кодове, например чрез проста програма на C:

```
#include <stdio.h>
void main ()
{char row, col;
printf("\n        Code Table\n\n   ");
for (col=0;col<16;col++)  printf(" %X ", col);
   for (row=2;  row<16;  row++)
     { printf("\n%X  ", row);
        for (col=0;  col<16;  col++)
           {printf(" %c ",  row*16+col);}
 }}
```

След като е установена подходяща активна кодова таблица, текстовият файл с асемблерната програма се преобразува от *Unicode* в ASCII - код с командата:

TYPE *NAME.TXT* > *NAME.ASM*

В получения файл с изходната асемблерна програма *NAME.ASM* кирилските символи се представят според активната кодова таблица с еднобайтов разширен ASCII код. При някои решавани задачи изборът на кодова таблица може да бъде съществен.

Пример за тестване

Ще илюстрираме процеса на тестването и настройката чрез малка асемблерна програма, която въвежда и извежда символи, записва и преобразува числа в регистрите на микропроцесора.

Задача: Да се въведе от клавиатурата символ на 16-чна цифра, след което в регистър AL да се запише съответната на символа цифра.

Решение: Клавиатурата на PC записва в паметта на компютъра ASCII кодовете на въведените символи. За прочитането им и за извеждане на символи върху екрана може да използваме функциите на MS-DOS. Те се извикват чрез прекъсване INT 21h. Номерът на извикваната функция се задава в регистър AH. Тук ще използваме за въвеждане на символа DOS функция 1, която чака натискане на бутон от клавиатурата и после записва ASCII-кода на въведения символ в регистър AL на микропроцесора. Ще използваме за изписване на текст DOS функция 9, която извежда на екрана низ от паметта, завършващ със символ ‘$’. Първият символ на низа се намира на адрес, зададен с адресната двойка регис-

три DS:DX (указател към началото). От таблица с ASCII кодовете попълваме по колони в табл. 1 представянето на символите на шестнадесетичните цифри. На ред ‘Символ’ записваме символа на шестнадесетичната цифра. Ред ‘ASCII’ съдържа ASCII кода на символа в колоната. На ред ‘HEX’ записваме шестнадесетичната цифра, чийто символ е на първи ред в същата колона – програмата ще я връща като резултат. Ред ‘Формула’ показва математическата зависимост между числата в една и съща колона на ред ASCII и тези на ред HEX.

Табл. 1. ASCII кодове на символите на шестнадесетични цифри.

Символ	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’	‘6’	‘7’	‘8’	‘9’
ASCII	30h	31h	32h	33h	34h	35h	36h	37h	38h	39h
HEX	0	1	2	3	4	5	6	7	8	9
Формула	HEX=ASCII-30h									
Символ	‘A’	‘B’	‘C’	‘D’	‘E’	‘F’				
ASCII	41h	42h	43h	44h	45h	46h				
HEX	A	B	C	D	E	F				
Формула	HEX=ASCII-37h									

Ясно е, че въведеният ASCII код ще се преобразува до HEX цифра по две различни формули в зависимост от стойността му. От тук произтича алгоритъмът на преобразуване. Първо програмата ще провери дали въведения ASCII код е по-малък или равен на 39h. Ако това е така, съответната шестнадесетична цифра HEX се получава като се извади от ASCII кода числото 30h. В противен случай програмата трябва да извади от разликата още 7. Следва един вариант на асемблерна програма, която решава задачата по този начин.

```
;Програма за преобразуване на символ на шестнадесетична цифра
;въведен от клавиатурата в съответното му шестнадесетично число.
;Резултатът се помества в регистър AL .
      DATA     SEGMENT PARA PUBLIC "DATA"               ;сегмент за данни
MESSAGE         DB  "INPUT A HEXADECIMAL DIGIT:",  "$"
      DATA     ENDS
  STK  SEGMENT   STACK                                  ;стеков сегмент
            DB      256 DUP ("?")
  STK  ENDS
CODE   SEGMENT   PARA PUBLIC "CODE"            ;начало на кодовия сегмент
            ASSUME  CS:CODE, DS:DATA, SS:STK
MAIN    PROC                                   ;начало на процедура MAIN
            ASSUME       CS:CODE,  DS:DATA,  SS:STK
            MOV    AX, DATA                   ;адреса на сегм. за данни в рег. AX
            MOV    DS, AX                          ;AX в DS
            MOV    AH, 9                           ;AH=9 - номер на DOS функция
            MOV    DX, OFFSET MESSAGE              ;DS:DX  начало на низ MESSAGE
            INT     21h                      ;Извод на екран 'MESSAGE' чрез DOS
```

```
        MOV   AH, 1                    ;функция с номер AH=1
        INT   21h          ;Въвежда в AL ASCII код на символ от клавиатурата
        SUB   AL, 30h                  ;AL=AL-30h
        CMP   AL, 9                    ;Провери дали AL<=9
        JBE   L1                       ;Да, AL е готовата цифра
        SUB   AL, 7                    ;Не, извади още 7, ако AL= 'A'-'F'
  L1:   MOV   AX, 4C00h                ;AH=4Ch - номер на DOS функция
        INT       21h                  ;Изход в DOS
MAIN    ENDP                           ;Край на процедурата MAIN
CODE    ENDS                           ;Край на кодовия сегмент
END     MAIN                           ;Край на програма. Вх. точка MAIN
```

Асемблерният текст на програмата се въвежда чрез текстовия редактор Notepad, както бе обяснено по-горе. След преобразуване изходната програма се записва на диска като текстов файл с име *PRG6_1.ASM* в MS-DOS формат. Извършва се транслацията и след отстраняване на синтактичните грешки се стартира Турбо дебъгера.

Менюта на TD

Ако указаните действия са изпълнени правилно се отваря прозорец "*Module*" на дебъгера TD, в който се намира изходния текст на асемблерната програма (вижте фигура 2).

```
C:\WINDOWS\system32\cmd.exe - td prg6_1
 Ё  File  Edit  View  Run  Breakpoints  Data  Options  Window  Help        READY
[■]=Module: prg6_1 File: prg6_1.asm 1═════════════════════════════1=[↑][↓]
 data    segment para public "data"       ;сегмент за данни
 message db "Input a Hexadecimal digit:","$"
 data    ends
 stk     segment stack                    ;стеков сегмент
         db      256 dup ("?")
 stk     ends
 code    segment          para public "code"      ;начало на кодовия сегмент
         assume  cs:code, ds:data,ss:stk
 main    proc             ;начало на процедура main
         assume  cs:code,ds:data,ss:stk
►        mov     ax,data  ;адреса на сегм. за данни в рег. ax
         mov     ds,ax    ;ax в ds
         mov     ah,9     ;ah=9 - номер на DOS функция
         mov     dx,offset message       ;ds:dx сочи началото на
                          ; низ message
         int     21h      ;Извод на екран 'message' чрез DOS
         mov     ah,1     ;функция с номер ah=1
         int     21h      ;Въвежда в al ASCII код на символ
                          ;въведен от клавиатурата
         sub     al,30h   ;al=al-30h
         cmp     al,9     ;Провери дали al<=9
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Фиг. 2. Прозорец *Module* на TD.

В първата колона на прозореца пред следващия оператор, който трябва да бъде изпълнен, има триъгълен маркер. Това е т. нар. „курсор на изпълнението”. При стартирането на дебъгера или след Reset на програмата той сочи входната точка на програмата.

В главния прозорец на дебъгера програмистът обикновено разполага още един или няколко прозореца, като във всеки момент е активен само

един от тях. Активацията на прозорец става чрез кликване на мишката в някоя видима негова точка.

Дебъгерът се управлява чрез система от менюта. Менюто е два вида: главно и локално.

Главното меню се намира най-горе на екрана. То е достъпно постоянно и се извиква с натискане на клавиш F10. След това трябва да се избере нужната точка от менюто.

Локалното меню е отделно меню за всеки прозорец. То се извиква чрез кликване в областта на прозореца с десния бутон на мишката (или активирайки прозореца и натискайки ALT+F10).

Режими на работа на TD

Проверката на работата на асемблерната програма става като се проследи действието на програмата върху микропроцесора – как тя го използва и как мени състоянието на ресурсите му и на компютъра.
Дебъгерът може да изпълни програмата в един от 4 възможни режима:

- Безусловно изпълнение;
- Изпълнение по стъпки;
- Изпълнение до текущото положение на курсора;
- Изпълнение с поставени точки на прекъсване.

Режимът на безусловно изпълнение се ползва когато е нужно да се види общото поведение на програмата. За пускането на програмата в този режим се натиска клавиша F9. В точките на програмата, където трябва да се въведат или изведат данни, дебъгерът действува в съответствие с логиката на работа на използваните средства за вход/изход. За разглеждане или въвеждане на тази информация може да се отвори потребителски прозорец, избирайки от менюто **Windows>User** screen или натискайки клавишите Alt+F5. Ако програмата работи правилно, настройката на програмата приключва. Ако има някакви проблеми или трябва по-детайлно да се изучи действието на програмата се ползват следващите 3 режима на работа.

Режимът на изпълнение на програмата до текущото положение на курсора е целесъобразно да се използва в случая, когато трябва да се провери дали даден участък от програмата функционира правилно. Активацията на този режим става като се позиционира курсора на необходимия оператор на програмата и се натисне бутона F4. Програмата се стартира и спира на посочената команда, без да я изпълнява. После при нужда се преминава към постъпково изпълнение.

При режим на изпълнение на програмата с поставяне на точки на прекъсване след стартирането програмата ще спира на определените от програмиста оператори (точки на прекъсване - **Breakpoints**). Точките на прекъсване в програмата се поставят преди да започне изпълнението и. Това става като се позиционира курсора на съответния оператор и се натисне клавиша F2. Избраните оператори/редове се засветват. Премахването на точка на прекъсване става като се премес-

ти курсора на реда с точката на прекъсване, която желаем да премахнем и се натисне клавиша F2.

След поставянето на точките за прекъсване програмата се пуска с клавиша F9. На първата точка на прекъсване програмата ще спре. Тогава може да се види състоянието на процесора и паметта, а после отново с клавиш F9 да се продължи изпълнението на програмата до следващата точка на прекъсване. При поставени точки на прекъсване програмата може да се изпълнява и по стъпки.

Режимът на постъпково изпълнение на програмата се използва за детайлно изучаване на нейната работа. В този режим изпълнението на програмата се прекъсва на всяка машинна (асемблерна) инструкция. При това прекъсване може да се наблюдава резултата от изпълнението на командата. Активацията на този режим става с бутона F7 (**Run>Trace info**) или F8 (**Run>Step over**). И двата клавиша активират постъпковото изпълнение. Когато програмата срещне команда за преход или за извикване на процедура или прекъсване при натискането на клавиша F7 дебъгерът ще отработи прехода към процедурата или прекъсването и ще спре, а при натискането на клавиша F8 дебъгерът ще извика и изпълни процедурата или прекъсването като една команда и ще спре на следващата линия от програмата.

Прозорци на TD

При работа в режим на постъпково изпълнение е полезно освен прозореца Module да се ползва и прозореца CPU, който се извиква от главното меню с командата **View>CPU**.

Прозорецът CPU е даден на фигура 3. Той показва състоянието на процесора и се състои от 5 подчинени прозорци. Чрез кликване с ляв бутон на мишката вътре в прозореца и натискането на бутона F1 се извежда справочна информация за опциите на локалното меню.

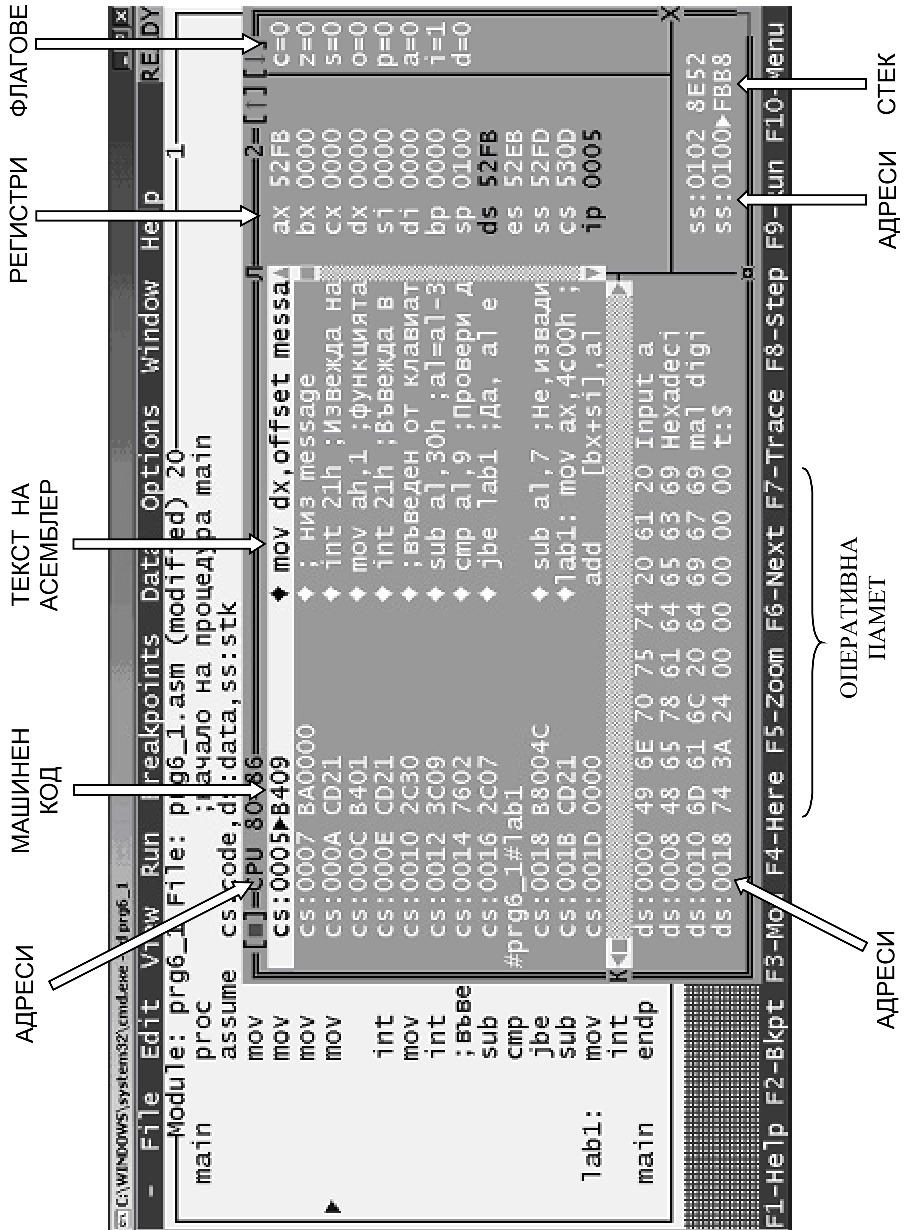
Нека разгледаме петте подчинени прозорци на CPU – прозореца.

1. В прозореца на изходната програма в дизасемблиран вид е представен машинния код на асемблерната програма от прозореца Module. В този прозорец може да се направи постъпковото изпълнение. Текущата команда се засветва.

2. В прозореца на регистрите на процесора (**Registers**) се показва текущото състояние на регистрите на процесора (по подразбиране само на регистрите на процесор i8086). Промяната на стойностите, записани в регистрите, може да стане като се щракне с ляв бутон на мишката върху съответния регистър. Избрания регистър се засветва. После се извиква локалното меню чрез кликване с десен бутон на мишката. Чрез кликване с ляв бутон на мишката върху командата от менюто се избира някоя от следните функции:

- **Increment** – увеличава с 1 стойността на регистъра;
- **Decrement** – намалява с 1 стойността на регистъра;
- **Zero** – нулира стойността на регистъра;

- **Change** – променя стойността на регистъра.



Фиг. 3 Прозорец CPU на дебъгер

Последната опция отваря прозорец, в първия ред на който се въвежда новата стойност на регистъра като шестнадесетична константа, записана по правилата на Асемблерния език. Ако преди това са въвеждани други стойности, те се появяват на някой от следващите редове от прозореца и тези данни могат да се изберат и въведат чрез двукратно кликване с ляв бутон на мишката върху тях.

3. В прозореца на флаговете (**Flags**) се показва текущото съдържание на флаговете на процесора. Те могат да се променят алтернативно като се избере флага, който желаем да променим чрез кликване върху него с ляв бутон на мишката. После се. извиква локалното меню чрез кликване с десен бутон на мишката и после се щраква с ляв бутон на мишката върху командата *Toggle*.

4. В прозореца на стека (**Stack**) се показва съдържанието на паметта, заделена за стек. Адреса на областта на стека се определя от съдържанието на регистрите SS и SP. Клетките от паметта на стека се извеждат по два байта на ред в шестнадесетичен код, като адресите им намаляват отгоре надолу и отляво надясно. На първия ред се показва съдържанието на паметта, указано с двойката двубайтови шестнадесетични числа сегмент:отместване, записана в горния ляв ъгъл на прозореца. Локалното меню на този прозорец предлага следните опции:

- **Goto** – въвеждане на начален адрес на извежданата област от паметта в прозореца;
- **Origin** – задава начален адрес на извежданата област от паметта SS:SP;
- **Follow** – Премества началния адрес на извежданата област от паметта на адреса на засветената дума (т.е. извежда следващия блок от паметта за последователно обхождане);
- **Previous** - Премества началния адрес на извежданата област от паметта на адреса на предишния въведен такъв с командата **Goto** и **Follow**. По този начин могат да се извеждат алтернативно 2 различни области на стека или да се направи обратно проследяване при промяна на данните в стека;
- **Change** - въвеждане на нова стойност на дума в стека. Действува като същата опция при промяна стойността на регистрите.

5. В прозореца на оперативната памет (Dump) се извежда съдържанието на област от паметта по адрес, който се формира от двойката сегмент:отместване, указани в лявата част на прозореца. Може да се изведе или промени произволна част от паметта. Локалното меню предлага следните команди:

- **Goto** – въвеждане на начален адрес на извежданата област от паметта в прозореца;
- **Origin** – задава начален адрес на извежданата област от паметта CS:IP;

- **Follow** – Премества началния адрес на извежданата област от паметта на адреса на прехода, само ако е засветена команда за преход или извикване на процедура. Използва се за проследяване на фрагменти с такива инструкции;
- **Previous** - Премества началния адрес на извежданата област от паметта на адреса на предишния въведен такъв с командата Caller и Follow;
- **Search** – Търсене на въведена инструкция или списък от байтове;
- **Mixed** – Позволява избор на формата на извеждания дизасемблиран код: **No** – само дизасемблирани инструкции, **Both** – сорс код на програмата, ***Yes*** – дизасемблирани инструкции и номерата на линиите в сорса, на които се намират;
- **Caller** - Премества началния адрес на извежданата област от паметта на адреса на извикващата инструкция (за процедури и прекъсвания);
- **New cs:ip** – Въвежда нова входна точка на програмата;
- **Assebble** – Въвежда машинния код на въведена от програмиста асемблерна инструкция. Промените не се записват на диска;
- **I/O** – Позволява четене или запис на байт или дума по адрес на входно-изходен регистър;
- View **source** - отваря прозорец Module.

Да забележим, че прозорецът CPU всъщност показва видимата част на програмния модел на процесора. За по-удобна работа могат да се изведат на екрана само някои от подчинените прозорци на CPU прозореца чрез менюто ***View***.

Прекъсването на изпълнението на програмата в кой да е режим може да стане като се натиснат клавишите **Ctrl+F2**.

Тестване и настройка на програмата

Нека тестваме заредената програма с Турбо дебъгера. За да се убедим, че програмата работи правилно, трябва да я изпълним няколко пъти с различни входни данни – които ще се преобразуват по различен начин. Ще я тестваме с въведен символ от клавиатурата между ‘0’ и ‘9’, например ‘4’, със символ между ‘A’ и ‘F’, например ‘B’, а също така и с въведен символ ‘9’. Тъй като резултатът трябва да се получи в регистър AL, а не се извежда на екран, ще спрем програмата в момента преди да предаде управлението на DOS и ще разгледаме регистрите на микропроцесора. За целта поставяме точка на прекъсване на командата с етикет LAB1. После изпълняваме програмата докрая (с **Run**/**F9**) Изпълнението спира на точката на прекъсване. Когато програмата извежда надписи или въвежда данни, Турбо дебъгерът превключва екрана на дисплея и показва работния DOS прозорец. Там се извежда съобщението:

```
Input a Hexadecimal digit:_
```

Курсорът мига непосредствено след него. Натискаме бутон ‘4’. TD връща на екрана прозореца CPU. Виждаме, че курсорът на изпълнението в прозорец ”*Module*” сочи реда на от програмата, където сме поставили точката на прекъсване:

LAB1: MOV AX, 4C00h

Тази команда предстои да се изпълни. Нека отворим прозорец CPU (**View>CPU**). В прозореца CPU (фигура 3) курсорът на изпълнението сочи байта по адрес CS:001B, където е машинният код на първия байт от командата

MOV AX, 4C00h.

В прозореца на флаговете регистър AL е осветен, защото съдържанието му е променено и стойността му е 04, което е правилния резултат. Ако продължим изпълнението на програмата до край (с бутон **Run** / **F9** или постъпково с двукратно натискане на бутон **F7** / **F8**), TD извежда след изпълнението на последната команда от програмата в централен прозорец надпис: “Terminated, exit code 0”. Това е указание за нормално завършване на една асемблерна програма (фигура 4). Кликване върху бутон OK предава управлението на дебъгера.

Фиг. 4. Нормално завършване на Асемблерна програма.

Ако желаем да разгледаме данните в паметта, заделена за програмата, трябва да поставим точка на прекъсване на командата, която следва командите, зареждащи DS сегментния регистър с базовия адрес на сегмента за данни, т.е. след команда:

MOV DS, AX

слагаме точка на прекъсване на командата:

MOV AH, 9 .

За целта със стрелките преместваме курсора (или щракваме с мишката) на реда с командата и с **Breakpoints>Toggle/F2** фиксираме точката на прекъсване. Отваряме прозорец CPU и изпълняваме програмата до точката на прекъсване с бутон **Run/F9**.

Съдържанието на кодовия сегмент се извежда по подразбиране в горната част на прозореца заедно с дизасемблирания текст на програмата. Паметта на PC се визуализира в долната част на прозореца. За да видим съдържанието на сегмента за данни е необходимо да зададем начален адрес за извеждане на паметта, съвпадащ с началото на сегмента за данни. Трябва да посочим Dump прозореца с мишката, да щракнем с ляв бутон, за да го направим активен и после да щракнем с десен бутон на мишката. От отворилото се локално меню трябва да изберем опцията **GoTo**. После пишем в полето за въвеждане в прозореца "Enter address to position to" **ds:0** и посочваме **OK** и щракваме върху него с мишката (или натискаме **Enter**) за потвърждение.

В лявата долна част на прозореца се появяват адресите на паметта във формат *сегмент:отместване* и отляво надясно последователно се извеждат байтовете от паметта, прочетени от тези адреси. Вдясно от 16-те байта на същия ред е тяхната интерпретация в разширен ASCII код. Както се вижда на фиг. 3, в началото на визуализирания сегмент за данни се намира променливата message. Можем да променим съдържанието на един или няколко байта от паметта, изведена в прозореца, като щракнем с мишката върху първия байт от паметта, който искаме да променим. После извикваме локалното меню и от него избираме опция **Change**. В отварящия се прозорец „Enter new data bytes" записваме новите стойности като последователност от байтове в шестнадесетичен код, разделени с интервали или символен низ, заграден в кавички и потвърждаваме въвеждането с **OK** или **Enter.**

Нека изпълним 3 команди с трикратно натискане на бутона **F7/F8.** TD извежда в DOS прозорец стринга *MESSAGE*. Можем да проследим в прозореца на регистрите как се променят стойностите на регистрите на микропроцесора след всяка една команда. Например след изпълнението на командата: MOV AH, 9 , стойността на старшия байт на регистъра AX в прозореца на регистрите се променя на 09, а самата стойност от черна става бяла. Следващата команда от програмата:

MOV DX, OFFSET MESSAGE

зарежда в регистър DX отместването от началото на сегмента на първия байт от стринга MESSAGE. Като първа променлива той има отместване 0. Ако стойността на регистъра DX е била 0, тя няма да се промени. Бихме могли преди изпълнението на тази команда да променим стойността на регистъра DX чрез активиране на прозореца с регистрите, извикване на локалното меню и избор на опцията "Change" и въвеждане на пример на нова стойност 1111. Тогава промяната на DX

в резултат на изпълнението на командата става видима.

След изпълнението на следващите 2 команди от програмата по стъпки можем да видим как програмата очаква въвеждане от клавиатурата на 1 символ, без да го извежда на екран и после записва ASCII кода му в регистър AL. Например ако сме въвели ‘4’, стойността на регистър AL ще се промени на 34h. Продължавайки последователното изпълнение на програмата стъпка по стъпка, можем да проследим как се менят флаговете на микропроцесора след командата за сравнение и преобразуването на числото в регистъра AL до 04. Можем да пуснем отначало програмата с **Run>Program reset**/**Ctrl+F2** многократно.

Ако изпълнението на програмата в Турбо-дебъгера завършва нормално (фигура 4), изпълнението и в DOS прозорец с указване само на името и с командата:

Prg6_1

също ще завърши нормално, без да се извеждат съобщения за грешки. Програмата ще изведе на екрана съобщението, ще въведе 1 символ от клавиатурата и ще приключи работата си, обаче няма да видим какво става с регистрите на процесора след изпълнението и.

Работа с копроцесор

Когато асемблерната програма програмира вградения копроцесор, е необходимо да се използва и прозореца на числовия процесор (Numeric processor), за да проследим състоянието на регистрите му. Той се отваря от главното меню с **View>Numeric processor** или с горещите клавиши **Alt+V>N**.

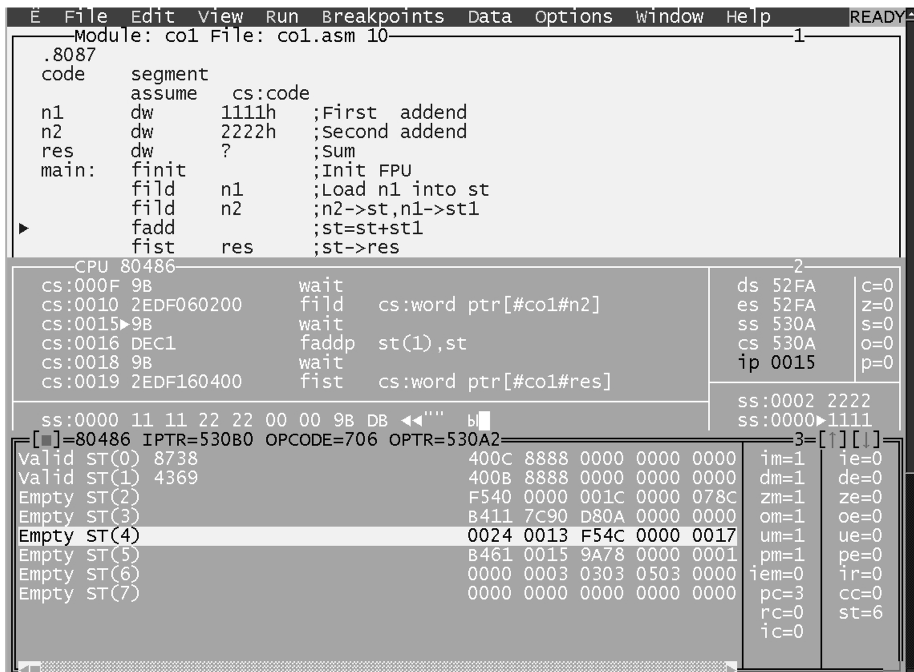
Нека разгледаме асемблерна програма с име *CO1.ASM*. На първата линия е записана директивата .8087, която разрешава на Асемблера да генерира машинен код за математическия процесор 8087. Програмата първо зарежда в регистър ST(0) цяло двубайтово число n1 от паметта, а после пак там двубайтово число n2 от паметта. Втората операция премества съдържанието на ST(0) в ST(1). След това програмата събира ST(0) с регистъра ST(1) и копира получената сума от ST(0) в двубайтова клетка от паметта res. Текстът на програмата *CO1.ASM* е даден по-долу.

```
.8087
    CODE    SEGMENT
            ASSUME CS:CODE
      n1    DW      1111h               ;First  addend
      n2    DW      2222h               ;Second  addend
      res   DW      ?                   ;Sum
   MAIN:    FINIT                       ;Init  FPU
            FILD    n1                  ;Load  n1  into  st
            FILD    n2                  ;n2  →  st,  n1  →  st1
```

```
          FADD                          ;st=st+st1
          FIST      res                 ;st → res
          MOV       AX, 4C00h           ;Exit
          INT       21h
CODE      ENDS
          END       MAIN
```

На фигура 5 транслираната асемблерна програма е заредена в Турбо дебъгера. Отворени са 3 прозореца: Module, CPU и Numeric processor, подредени са един след друг и са разпънати максимално по ширина. Прозорецът (2) на числовия процесор се намира в средата на екрана. В заглавието му се извеждат 4 съобщения:

1. Модел на копроцесора (автоматично се определя от дебъгера).
2. IPTR=...- текущо съдържание на указателя на командите.Това е физическия адрес на паметта, по който е разположена последната изпълнявана команда от копроцесора.
3. OPTR=...- адрес на паметта, към който се е обърнала последната команда на копроцесора, ако тя е имала адресен операнд.
4. OPCODE=...- Код на операцията на последната изпълнявана команда на копроцесора. Това е кода на командата без първите 5 бита.



Фиг. 5. Настройка на програма, ползваща FPU

В прозореца Numeric processor има 3 области. Отляво е разположена областта Registers, която заема по-голямата част от прозореца. В нея се показват стойностите на осемте регистри от стека на копроцесора, означени с логическите номера ST(0)-ST(7). На всеки ред описанието на регистъра започва с поле, поясняващо състоянието на регистъра. Възможните стойности в това поле могат да бъдат:

- **Valid** – ако в регистъра има коректно реално число;
- **Empty** – празен;
- **Zero** – в регистъра има 0;
- **Spec.** – в регистъра има специална стойност напр. **NAN** (Not a Number – не е число), показващо грешка в пресмятанията.

Второто поле на реда е логическия номер на регистъра. Третото поле показва съдържанието на регистъра като реално 80-битово число с плаваща запетая. Четвъртото поле показва съдържанието на регистъра в шестнадесетичен код, като номерата на разрядите намаляват отляво надясно.

При настройката на програмата е възможно да се влияе на съдържанието на регистрите на стека. Чрез избиране с мишката или с клавишите за навигация на желан регистър и кликване с десен бутон върху реда с избрания регистър, се активира меню с 3 опции:

- **Zero** – нулиране на регистъра.
- **Empty** – освобождаване на регистъра. Съдържанието на регистъра не се изменя, а се изменя само таг в регистъра на таговете, където се записва стойността 11b.
- **Change** – промяна на съдържанието на регистъра. Въвежданите цели или реални десетични константи се въвеждат съгласно синтаксиса на Асемблера. Те се преобразуват във вътрешния 80-битов формат на FPU и се записват в съответния регистър.

Следващата област от прозореца Numeric processor-Control - съдържа набор полета, чийто имена съвпадат с имената на битове или битови полета в управляващия регистър на копроцесора CWR. Тези полета са:

- IM – маска недействителна операция;
- DM – маска денормализиран операнд;
- ZM – маска делене на нула;
- OM – маска препълване;
- UM – маска отрицателно препълване;
- PM – маска точност;
- IEM – маска заявка за прекъсване (8087);
- PC – управление на точността;
- RC – управление на закръгляването;
- IC – управление на стойността безкрайност.

Областта Control има локално меню само с една команда **Toggle**. При кликване с десен бутон на мишката върху нея избраното поле се изменя циклично.

Третата област от прозореца Numeric processor-Status – съдържа набор от полета, чийто имена съвпадат с имената на битове или битови полета от регистъра на състоянието на копроцесора SWR. Те са:

- IE – грешка недействителна операция;
- DE – грешка денормализиран операнд;
- ZE – грешка делене на нула;
- OE – грешка препълване;
- UE – грешка отрицателно препълване;
- PE – грешка точност;
- IR – маска заявка за прекъсване (8087);
- CC – код на условието (състояние на битовете C3C2C1C0);
- ST – указател на върха на стека (поле TOP от рег. SWR).

Областта Status има локално меню само с една команда **Toggle**. Кликването с десен бутон на мишката върху нея изменя циклично избраното поле.

Приложение А

Асемблерни команди за микропроцесор i8086 и математически копроцесор i80x87

А.1 Команди за микропроцесор i8086

В следващата таблица е представен базовият набор команди на Асемблер за микропроцесор Intel 80x86 CPU. При описанието на формата на възможните операнди на командите са използвани следните означения:

R	8/16-битов регистър с общо предназначение
R8	8-битов регистър с общо предназначение
R16	16-битов регистър с общо предназначение
ACC	Акумулатор AL или AX
SR	Сегментен регистър
M	Байт или дума в паметта
M8	Байт в паметта
M16	Дума в паметта
P32	Указател (двойна дума – адрес)
I	Непосредствен операнд – число или адрес
□	Без операнд

Тези съкращения могат да се комбинират за команди с няколко операнда. В този случай операндите са изредени в списък, разделени със запетая. Например форматът R,R означава, че инструкцията има два операнда, които са регистри с общо предназначение. Повечето двуоперандни инструкции позволяват такива операнди. Съкращението O2 се използва за представяне на следните операнди: (R,R); (R,M); (R,I); (M,R); (M,I). В едно и също поле вариантите на операнда се показват разделени с наклонена черта. Например, ако за операнд се използва 8 битов регистър или памет, тогава се използва съкращението R8/M. Вариантите на формата се изреждат разделени с интервал или нов ред. При някои команди форматът включва имената на точно определени регистри на микропроцесора. Например, във входно-изходните команди се записват като операнди регистрите AL/AX и DX, а в командите за изместване и ротация броят на изместванията се указва в регистър CL или с константа 1. Префиксите са указани със звездичка (*) след името на командата.

Таблицата с командите показва още как се променят битовете на флаговия регистър след изпълнението на командата от първата колона на реда. Флаговете са пояснени в глава 1 и са означени така:

O - Overflow flag (Препълване);
S - Sign flag (Знак)**;**
Z - Zero flag (Нула);
P - Parity flag (Четност);
A - Auxiliary flag (Междинен пренос);
C - Carry flag (Пренос).

Ако колоната под името на съответния бит е празна, това означава, че битът не се променя. Ако битът винаги се променя еднакво, в колоната се записва новата му стойност – 0 или 1. Ако битът получава стойност, която зависи от операндите в командата, в колоната е записан символ C. Ако стойността на бита се променя, но не е точно определена, в колоната под името му е записано ?.

Флагът D (*Direction flag*) и флагът I (*Interrupt Enable flag*) се променят единствено при командите за установяване и нулиране на съответните флагове – CLD и STD, CLI и STI и затова не са отразени в таблицата.

Ако на един ред от таблицата са записани няколко команди, то те имат еднакъв код на операцията.

КОМАНДА			**Флагове**					
Мнемо код	**Описание**	**Формат**	**O**	**S**	**Z**	**A**	**P**	**C**
AAA	ASCII Adjust After Addition		?	?	?	C	?	C
AAD	ASCII Adjust Before Division		?	C	C	?	C	?
AAM	ASCII Adjust AX After Multiply		?	C	C	?	C	?
AAS	ASCII Adjust AL After Subtraction		?	?	?	C	?	C
ADC	Add with Carry	O2	C	C	C	C	C	C
ADD	Add	O2	C	C	C	C	C	C
AND	Logical AND	O2	0	C	C	?	C	0
CALL	Call Procedure	R16, M16, P32, I						
CBW	Convert Byte to Word							
CLC	Clear Carry Flag							0
CLD	Clear Direction Flag							
CLI	Clear Interrupt Flag							
CMC	Complement Carry Flag							C
CMP	Compare two operands	O2	C	C	C	C	C	C
CMPS	Compare String Operands	M, M						
CMPSB	Compare Byte String		C	C	C	C	C	C

Мнемо код	Описание	Формат	O	S	Z	A	P	C
CMPSW	Compare Word String		C	C	C	C	C	C
CWD	Convert Word to Double Word							
DAA	Decimal Adjust AL after Addition		?	C	C	C	C	C
DAS	Decimal Adjust AL after Subtraction		?	C	C	C	C	C
DEC	Decrement by 1	R8, M8 R16, M16	C	C	C	C	C	
DIV	Unsigned Divide	R,M	?	?	?	?	?	?
ESC	Put operand on the bus	I, R/M						
HLT	Halt							
IDIV	Signed Divide	R, M	?	?	?	?	?	?
IMUL	Signed Multiply	R, M	C	?	?	?	?	C
IN	Input from Port	ACC, I/DX						
INC	Increment by 1	R, M	C	C	C	C	C	
INT	Interrupt	I	T=0, I=0					
INTO	Interrupt on Overflow		T=0, I=0					
IRET	Interrupt Return		C	C	C	C	C	C
JA **JNBE**	Jump if Above Jump if Not Below or Equal	I						
JAE **JNB** **JNC**	Jump if Above or Equal Jump if Not Below Jump if Not Carry	I						
JB **JNAE** **JC**	Jump if Below Jump if Not Above or Equal Jump if Carry	I						
JBE **JNA**	Jump if Below or Equal Jump if Not Above	I						
JCXZ	Jump if CX=0	I						
JE **JZ**	Jump if Equal Jump if Zero	I						
JG **JNLE**	Jump if Greater Jump if Not Less or Equal	I						
JGE **JNL**	Jump if Greater or Equal Jump if Not Less	I						
JL **JNGE**	Jump if Less Jump if Not Greater or Equal	I						
JLE **JNG**	Jump if Less or Equal Jump if Not Greater	I						

Мнемо код	Описание	Формат	O	S	Z	A	P	C
JMP	Jump	R16, M16, I, P32						
JNE **JNZ**	Jump if Not Equal Jump if Not Zero	I						
JNO	Jump if Not Overflow	I						
JNP **JPO**	Jump if Not Parity Jump if Parity Odd	I						
JNS	Jump if Not Sign	I						
JO	Jump if Overflow	I						
JP **JPE**	Jump if Parity Jump if Parity Even	I						
JS	Jump if Sign	I						
LAHF	Loads Flags into AH							
LDS	Load Pointer to DS	R16, P32						
LEA	Load Effective Address	R16, P16						
LES	Load Pointer to ES	R16, P32						
LOCK*	Lock Bus							
LODS	Load String	M						
LODSB	Load Byte of String							
LODSW	Load Word of String							
LOOP	Loop	I						
LOOPE **LOOPZ**	Loop while Equal Loop while Zero	I						
LOOPNE **LOOPNZ**	Loop while Not Equal Loop while Not Zero	I						
MOV	Move Data	O2, SR, R R16/M16, SR						
MOVS	Move String	M, M						
MOVSB	Move Byte of String							
MOVSW	Move Word of String							
MUL	Unsigned Multiply	R, M	C	?	?	?	?	C
NEG	2's Complement Negation	R, M	C	C	C	C	C	C
NOP	No Operation							
NOT	1's Complement	R, M						
OR	Logical Inclusive OR	O2	0	C	C	?	C	0

Мнемо код	Описание	Формат	O	S	Z	A	P	C
OUT	Output to Port	I/DX, ACC						
POP	Pop a word from Stack	R16, M16, SR						
POPF	Pop from stack into Flags		C	C	C	C	C	C
PUSH	Push on to Stack	R16, M16, SR						
PUSHF	Push Flags onto Stack							
RCL	Rotate Left through Carry	R/M, CL/1	C					C
RCR	Rotate Right through Carry	R/M, CL/1	C					C
REP*	Repeat String Operation							
REPE* **REPZ***	Repeat while Equal Repeat while Zero							
REPNE* **REPNZ***	Repeat while Not Equal Repeat while Not Zero							
RET	Return from Procedure	□, I						
ROL	Rotate Left	R/M, CL/1	C					C
ROR	Rotate Right	R/M,CL/1	C					C
SAHF	Store AH into Flags			C	C	C	C	C
SAL **SHL**	Arithmetic Shift Left Shifts Logical Left	R/M,CL/1	C	C	C	?	C	C
SAR	Shift Arithmetic Right	R/M,CL/1	C	C	C	?	C	C
SBB	Subtract with Borrow	O2	C	C	C	C	C	C
SCAS	Scan String Data	M	C	C	C	C	C	C
SCASB	Scan Byte of String		C	C	C	C	C	C
SCASW	Scan Word of String		C	C	C	C	C	C
SHR	Shift Logical Right	R/M, CL/1	C	C	C	?	C	0
STC	Set Carry Flag							1
STD	Set Direction Flag		D=1					
STI	Set Interrupt En. Flag		I=1					
STOS	Store String Data	M						
STOSB	Store Byte of String							
STOSW	Store Word of String							
SUB	Integer Subtraction	O2	C	C	C	C	C	C

Мнемо код	Описание	Формат	O	S	Z	A	P	C
TEST	Logical Compare	R/M, R R/M, I	0	C	C	?	C	0
WAIT	Wait until Busy							
XCHG	Exchange memory/register	R/M, R R, R/M						
XLAT	Table look-up translation	M8						
XLATB	Table look-up translation							
XOR	Bitwise Exclusive OR	O2	0	C	C	?	C	0

A.2 Команди за операции с плаваща запетая

Тук са представени накратко командите за работа с плаваща запетая. Не е дадена информация дали съответната команда извлича данни от стека. В колоната за формàт са дадени типовете на използваните операнди при всяка команда. Използвани са следните означения:

ST	Регистър ST(0) /връх на стека/
STn	Регистър ST(1), ST(2)…ST(7)
AX	Регистър AX на CPU
P	Указател (Адрес в паметта)
M32	32-битово число в паметта
M64	64-битово число в паметта
M80	80-битово число в паметта
□	Без операнд

Колоната за флаговете в таблицата описва промяната на битовете за изключенията от регистъра за състоянията. Ако колоната под името на съответния бит е празна, битът не се променя. Ако битът винаги се променя еднакво, в колоната се записва новата му стойност – 0 или 1. Ако битът получава стойност, която зависи от операндите в командата, в колоната е записан символ C. Флаговете и причината за изключението са означени така:

S – Stack overflow/underflow
Z – Zero-Divide
P – Precision
D – Denormal Operand
U – Underflow

I – Invalid Operand
O – Overflow

Таблица с команди за FPU 80x86 и Pentium

КОМАНДА			**Флагове**						
Мнемо код	**Описание**	**Формати**	**S**	**P**	**U**	**O**	**Z**	**D**	**I**
F2XM1	Compute 2^x -1		C	C	C			C	C
FABS	Absolute value		C						
FADD	Add real	□ M32, M64 ST, STn STn, ST	C	C	C	C		C	C
FADDP	Add real and pop	ST, STn STn, ST	C	C	C	C		C	C
FBLD	Load BCD	M80	C						
FBSTP	Store BCD and pop	M80	C	C					C
FCHS	Change sign		C						
FCLEX	Clear exceptions		0	0	0	0	0	0	0
FCOM	Compare real	□, M32 M64, STn	C					C	C
FCOMP	Compare real and pop	□, M32 M64, STn	C					C	C
FCOMPP	Compare real and pop twice	□, M32 M64, STn	C					C	C
FCOS	Cosine of ST		C	C	C			C	C
FDECSTP	Decrement stack pointer								
FDIV	Divide real	□, M32 M64, STn ST, STn STn, ST	C	C	C	C	C	C	C
FDIVP	Divide real and pop	□, M32, M64 ST, STn STn, ST	C	C	C	C	C	C	C
FDIVR	Divide real reversed	□, STn M32, M64 ST, STn STn, ST	C	C	C	C	C	C	C

Мнемо код	Описание	Формати	S	P	U	O	Z	D	I
FDIVRP	Divide real reversed and pop	□ M32, M64 ST, STn STn, ST	C	C	C	C	C	C	C
FFREE	Free register	ST STn							
FIADD	Integer add	□ M16, M32	C	C	C	C		C	C
FICOM	Integer compare	M16, M32	C					C	C
FICOMP	Integer compare and pop	M16, M32	C					C	C
FIDIV	Integer divide	M16, M32	C	C	C	C	C	C	C
FIDIVR	Integer divide reversed	M16, M32	C	C	C	C	C	C	C
FILD	Load integer	M16, M32, M64	C						
FIMUL	Integer multiply	M16, M32		C	C	C		C	C
FINCSTP	Increment stack pointer								
FINIT	Initialize								
FIST	Integer store	M16, M32	C	C					C
FISTP	Integer store and pop	M16, M32, M64	C	C					C
FISUB	Integer subtract	M16, M32	C	C	C	C		C	C
FISUBR	Integer subtract reversed	M16, M32	C	C	C	C		C	C
FLD	Load real	M32, M64, M80, STn	C					C	C
FLD1	Load constant onto stack, 1.0		C						
FLDCW	Load control word	M16	C	C	C	C	C	C	C
FLDL2E	Load $\log_2$(e)		C						
FLDL2T	Load $\log_2$(10)		C						
FLDLG2	Load $\log_{10}$(2)		C						
FLDLN2	Load ln(2)		C						
FLDPI	Load π		C						
FLDZ	Load 0.0		C						

Мнемо код	Описание	Формати	S	P	U	O	Z	D	I
FLDENV	Load environment state	P	C	C	C	C	C	C	C
FMUL	Multiply real	□, M32 M64, STn ST, STn STn, ST		C	C	C		C	C
FMULP	Multiply real and pop	M32, M64 ST, STn STn, ST	C	C	C	C		C	C
FNCLEX	Clear exceptions, no wait		0	0	0	0	0	0	0
FNINIT	Initialize, no wait								
FNOP	No operation								
FNSAVE	Save status, no wait	P							
FNSTCW	Store control word, no wait	M16							
FNSTENV	Store environment, no wait	P							
FPATAN	Partial arctangent		C	C	C			C	C
FPREM	Partial remainder		C		C			C	C
FPREM1	IEEE partial remainder		C		C			C	C
FPTAN	Partial Tangent		C	C	C			C	C
FRNDINT	Round to integer		C	C				C	C
FRSTOR	Restore saved state	P	C	C	C	C	C	C	C
FSAVE	Save FPU state	P							
FSCALE	Scale by 2^n				C	C			C
FSIN	Sine of ST		C	C	C			C	C
FSINCOS	Sine and cosine of ST		C	C	C			C	C
FSQRT	Square root		C	C				C	C
FST	Store real	M32, M64 STn	C	C	C	C		C	C
FSTCW	Store control word	M16							
FSTENV	Store FPU environment	P							

Мнемо код	Описание	Формати	S	P	U	O	Z	D	I
FSTP	Store real and pop	M32, M64, M80, STn	C	C	C	C		C	C
FSTSW	Store status word	AX, M16							
FSUB	Subtract real	□, M32 M64, STn ST, STn STn, ST	C	C	C	C		C	C
FSUBP	Subtract real and pop	□ M32, M64 ST, STn STn, ST	C	C	C	C		C	C
FSUBR	Subtract real reverse	□, M32 M64, STn ST, STn STn, ST	C	C	C	C		C	C
FSUBRP	Subtract real reverse and pop	ST, STn STn, ST	C	C	C	C		C	C
FTST	Test for zero		C					C	C
FUCOM	Unordered compare		C					C	C
FUCOMP	Unordered compare and pop		C					C	C
FUCOMPP	Unordered compare and pop twice		C					C	C
FWAIT	Wait								
FXAM	Examine stack top								
FXCH	Exchange registers	□, STn	C						
FXTRACT	Extract exponent and significand		C				C	C	C
FYL2X	Compute $Y*\log_2(X)$		C	C	C	C	C	C	C
FYL2XP1	Compute $Y*\log_2(X+1)$		C	C	C			C	C

ПРИЛОЖЕНИЕ Б

Текст на файл IOPROC.ASM

```
; Модул IOPROC:  процедура за вход-изход

   PUBLIC  PROCNL, PROCOUTNUM, PROCFLUSH
   PUBLIC  PROCINCH, PROCININT

   IOCODE  SEGMENT
           ASSUME  CS:IOCODE

; ---------------------------------------------------------------------------------
;          ИЗВЕЖДАНЕ  НА  ЕКРАНА
; ==============================================================
;  ПРОЦЕДУРА  "Преход на курсора на нов ред на екрана"
;  Извикване:  CALL  PROCNL
;  Параметри:  няма
; ---------------------------------------------------------------------------------
PROCNL  PROC  FAR
        PUSH  DX
        PUSH  AX
        MOV   AH, 2
        MOV   DL, 13            ; "CR" (курсор в началото на реда)
        INT   21h
        MOV   DL, 10            ; "LF" (курсор на следващия ред)
        INT   21h
        POP   AX
        POP   DX
        RET
   PROCNL  ENDP
; ==============================================================
;  ПРОЦЕДУРА  "Извеждане на цяло 16 битово число със знак"
;  Извикване:  CALL  PROCOUTNUM
;  Входни параметри:  AX – извеждано число
;                     DH – число със знак (1), или без знак (0)
;                     DL – ширина на отпечатъка (>=0)
;                Ако ширината е недостатъчна, да се увеличи.
;                Отпечатъкът е дясно подравнен.
; ---------------------------------------------------------------------------------

PROCOUTNUM  PROC  FAR
         PUSH  BP
         MOV   BP, SP
         PUSH  AX
         PUSH  DX
         PUSH  SI
         SUB   SP, 6                    ;  6 байта в стека за числото
 ;  Проверка на знака на числото
         CMP  DH, 1                     ; извеждане със знак (DH)=1?
         JNE   PON0
         CMP  AX, 0
```

```
        JGE   PON0
        MOV  DH, 2                      ; ако извеждането е със знак и (AX)<0,
        NEG   AX                        ;   то DH:=2,  AX:=ABS(AX)
PON0:  PUSH  DX                         ; спасяване DH (знак) и DL (ширина)
 ; Запис на цифрите на числото в стека (в обратен ред)
        XOR   SI, SI                    ; (SI) – брой на цифрите в числото
PON1:  MOV  DX, 0                       ; AX → (DX, AX)
        DIV   CS:TEN                    ; AX=AX DIV 10 ;  DX=AX mod(10)
        ADD   DL, "0"
        MOV   [BP-12+SI], DL            ; цифра → стек
        INC   SI
        OR   AX, AX
        JNZ   PON1                      ; още не е 0
  ; Запис на знак минус в стека (ако го има)
        POP   DX
        CMP   DH, 2
        JNE   PON2
        MOV   BYTE  PTR  [BP-12+SI], "-"
        INC   SI
  ;  дясно подравнен печат
PON2:  MOV  DH, 0                       ; DX – ширина на полето за печат
        MOV  AH, 2                      ; функция 02 на прекъсване 21h
PON21:  CMP  DX, SI
        JLE   PON3                      ; ширина <= на дължината на числото
        PUSH  DX
        MOV   DL, "."
        INT   21h
        POP   DX
        DEC   DX
        JMP   PON21
  ;  печат на знака (минус и) на цифрите
PON3:  DEC  SI
        MOV  DL,[BP-12+SI]
        INT  21h
        OR   SI,SI
        JNZ  PON3
  ;  изход от процедурата
         ADD  SP, 6
         POP  SI
         POP  DX
         POP  AX
         POP  BP
         RET
   TEN   DW 10
   PROCOUTNUM  ENDP
  ; ------------------------------------------------------------------------------------------
  ;      ВЪВЕЖДАНЕ ОТ КЛАВИАТУРАТА
```

```
 ; ------------------------------------------------------------------------------------------
 ;  Буфер за въвеждане от клавиатурата (за работа с функция 0Ah)
    MAXB  DB  128          ; максимален размер на буфера
    SIZEB  DB  0           ; брой на въведените символи в буфера
    BUF    DB  128 DUP(?)
    POSB  DB  0            ; № на последния прочетен символ от BUF

 ; ===============================================================
 ; Помощна  ПРОЦЕДУРА  "Въвеждане на символи в буфер BUF"
 ; (включително Enter) – въвеждане без покана
 ; ------------------------------------------------------------------------------------------
READBUF  PROC  NEAR
        PUSH  AX
        PUSH  DX
        PUSH  DS
        MOV   DX, CS
        MOV   DS, DX
        LEA   DX, BUF-2        ; DS:DX  - адрес на елемент BUF[-2]
        MOV   AH, 0Ah          ; въвеждане в буфера (включително Enter)
        INT   21h
        CALL  PROCNL           ; курсорът на нов ред
        INC   CS:SIZEB         ; дължината отчита и Enter
        MOV  CS:POSB, 0        ; брой на прочетените от BUF символи
        POP  DS
        POP  DX
        POP  AX
        RET
    READBUF  ENDP
 ; ===============================================================
 ; ПРОЦЕДУРА  "Изчистване на буфер за вход от клавиатурата"
 ; Извикване:  CALL  PROCFLUSH
 ; Параметри:  няма
 ;------------------------------------------------------------------------------------------
PROCFLUSH  PROC  FAR
        PUSH  AX
        MOV   CS:SIZEB, 0        ; изчистване на BUF
        MOV   CS:POSB, 0
        MOV   AH, 0Ch            ; изчистване на DOS-буфера
        MOV   AL, 0              ; без допълнителни действия
        INT   21h
        POP   AX
        RET
    PROCFLUSH  ENDP
 ; ==============================================================
 ; ПРОЦЕДУРА  "Въвеждане на символ"
 ;                                    (с пропуск или без пропуск Enter)
 ;   Извикване:  CALL  PROCINCH
 ;   Входни параметри:  ако (AL)=0, то Enter се пропуска
```

```
 ;                          ако (AL)=1,  то Enter не се пропуска
 ;  Изходни параметри:  AL – въведения символ (AH не се променя)
 ; -----------------------------------------------------------------------------------------
PROCINCH  PROC  FAR
        PUSH  BX
PRINCH1:
        MOV  BL, CS:POSB       ; № на последния прочетен символ
        INC   BL               ; следващ номер
        CMP   BL, CS:SIZEB     ; последен ли е символа в буфера?
        JB  PRINCH2
        JNE   PRINCH10          ; буферът не е прочетен до край?
        CMP  AL, 0              ; прочетен ли е края на реда (Enter)?
        JNE   PRINCH2
PRINCH10:
        CALL  READBUF           ; още въвеждане в буфера
        JMP    PRINCH1          ; повтори
PRINCH2:
        MOV   CS:POSB, BL       ; запомни № на прочетения символ
        MOV   BH, 0
        MOV   AL, CS:BUF[BX-1]               ; AL:=символ
        POP   BX
        RET
    PROCINCH  ENDP

 ; ===========================================================
 ; ПРОЦЕДУРА  "Въвеждане на цяло число, формат дума
 ;                                                  (с или без знак)
 ;   Обръщение:  CALL  PROCININT
 ;   Входни параметри:  няма
 ;   Изходни параметри:  (AX) = въведеното число
 ; -----------------------------------------------------------------------------------------
PROCININT  PROC  FAR
        PUSH  BX
        PUSH  CX
        PUSH  DX
 ; пропуск на празните символи
PRININT1:
        MOV   AL, 0
        CALL  PROCINCH      ; AL –  пореден символ (с пропуск на Enter)
        CMP   AL, " "                      ; празен символ?
        JE   PRININT1
 ; проверка на знака
        MOV   DX, 0             ; (DX)=  въведеното число
        MOV   CX, 0             ; (CH)=0 –  няма цифри,   (CL)=0 – плюс
        CMP   AL, "+"
        JE   PRININT2
        CMP   AL, "-"
        JNE   PRININT3
        MOV   CL, 1                        ; (CL)=1 –  минус
```

```
  ; цикъл по цифрите
PRININT2:
          MOV   AL, 1
          CALL  PROCINCH          ; (AL)= пореден символ (Enter - символ)
PRININT3:                         ; проверка на цифрата
          CMP   AL, “9”
         JA   PRININT4             ; > “9” ?
         SUB  AL, “0”
         JB   PRININT4             ; < “0” ?
         MOV   CH, 1               ; (CH)=1 – има цифра
         MOV   AH, 0
         MOV   BX, AX              ; BX – цифра като число
         MOV   AX, DX              ; AX – предходно. число
         MUL   CS:PRTEN            ; *10
         JC   PROVFL               ; >0FFFFh (DX<>0) – препълване
         ADD   AX, BX              ; +цифра
         JC   PROVFL
         MOV   DX, AX              ; спасяване на числото в DX
         JMP   PRININT2            ; към следващия символ
  ; няма повече цифри (числото е в DX)
PRININT4:
         MOV   AX, DX
         CMP   CH, 1               ; имаше ли цифри?
         JNE   PRNODIG
         CMP   CL, 1               ; имаше ли минус?
         JNE   PRININT5
         CMP   AX, 8000h           ; модулът на отриц. число > 8000h ?
         JA    PROVFL
         NEG   AX                  ; смяна на знака
PRININT5:
         POP   DX                  ; изход
         POP   CX
         POP   BX
         RET
    PRTEN DW 10
  ; ----------  реакция на грешки при въвеждането на числото
    PROVFL:     LEA CX, PRMSGOVFL        ; препълване
                JMP  PRERR
    PRNODIG:   LEA CX, PRMSGNODIG     ; няма цифри
    PRERR:      PUSH  CS           ; извеждане на съобщение за грешка
         POP   DS                  ; (DS)=(CS)
         LEA    DX, PRMSG
         MOV   AH, 9               ; OUTSTR
         INT    21h
         MOV   DX, CX
         MOV   AH, 9               ; OUTSTR
         INT    21h
         CALL  PROCNL
```

```
        MOV   AH, 4Ch                     ; FINISH
        INT    21h
    PRMSG          DB  "Грешка при въвеждане на числото:  “, “$”
    PRMSGOVFL  DB  “Препълване”, “$”
    PRMSGNODIG   DB  “Няма цифри”, “$”
    PROCININT  ENDP
    IOCODE  ENDS
    END                                        ; край на модул  IOPROC
```

Текст на файл IO.ASM

```
   .XLIST                         ; забрана за запис на този файл в листинг
  ;  Файл с макроси за вход-изход, подключващ се към програмата
  ;  по директивата:   INCLUDE  IO.ASM

  ;          КРАЙ НА РАБОТАТА НА ПРОГРАМАТА
  ; *******************************************************************************

  ; =============================================================
  ;  ЗАВЪРШВАНЕ НА ИЗПЪЛНЕНИЕТО НА ПРОГРАМАТА
  ;  Обръщение:  FINISH
  ;  Входен параметър: AL – код на завършването
  ;                                                  (може да се игнорира)
  ; -------------------------------------------------------------------------------------------

    FINISH  MACRO
          MOV  AH, 4Ch
          INT    21h
          ENDM

  ;      ИЗВЕЖДАНЕ НА ЕКРАНА (В ТЕКСТОВ РЕЖИМ)
  ; *******************************************************************************
  ; Общи забележки:
  ; Извеждането на екрана се осъществява незабавно, без използване
  ; на междинни буфери.
  ; Извеждането започва от позицията на курсора.

  ; =============================================================
  ;  ПРЕХОД НА НОВ РЕД
  ;  Обръщение:  NEWLINE
  ; -------------------------------------------------------------------------------------------
    EXTRN  PROCNL:FAR
NEWLINE  MACRO
          CALL  PROCNL
          ENDM

  ; =============================================================
  ;  ИЗВЕЖДАНЕ НА СИМВОЛ
  ;  Обръщение:  OUTCH  C,  където C е от тип i8, r8 или m8
  ; -------------------------------------------------------------------------------------------
OUTCH  MACRO  C
         PUSH   DX
         PUSH   AX
```

```
          MOV   DL, C
          MOV   AH, 2
          INT   21h
          POP   AX
          POP   DX
         ENDM

  ; =============================================================
  ;  ИЗВЕЖДАНЕ НА СИМВОЛЕН НИЗ
  ;    Обръщение:   OUTSTR
  ;    Входни параметри:  DS:DX –  начален адрес на низа
  ;          (краят на низа е маркиран със символ $, код 36 (24h))
  ; ---------------------------------------------------------------------------------------

OUTSTR  MACRO
            PUSH   AX
            MOV   AH, 9
            INT    21h
            POP   AX
          ENDM

  ; =============================================================
  ;  ИЗВЕЖДАНЕ НА ЦЯЛО ЧИСЛО СЪС ЗНАК, ФОРМАТ ДУМА
  ;   Обръщение:   OUTINT  NUM  [,LENG]
  ;   където NUM е въвежданото числов от тип: i16, r16, m16
  ;    LENG  е дължината на отпечатъка: i8, r8, m8 (за стойности >=0)
  ;    LENG=0  по подразбиране
  ; ---------------------------------------------------------------------------------------

      EXTRN  PROCOUTNUM:FAR
OUTINT  MACRO  NUM, LENG
           OUTNUM   <NUM>, <LENG>, 1
           ENDM

  ; =============================================================
  ;  ИЗВЕЖДАНЕ НА ЦЯЛО ЧИСЛО БЕЗ ЗНАК, ФОРМАТ ДУМА
  ;   Обръщение:    OUTWORD  NUM  [,LENG]
  ;    NUM и LENG  –  както в OUTINT
  ; ---------------------------------------------------------------------------------------
OUTWORD  MACRO  NUM, LENG
              OUTNUM <NUM>, <LENG>, 0
              ENDM

  ; ---------------------------------------------------------------------------------------
  ; ПОМОЩЕН МАКРОС ЗА ПРОВЕРКА НА ЗАПИСА НА ИМЕТО
  ; С РАЗЛИЧНИ (големи и малки) БУКВИ
  ; ---------------------------------------------------------------------------------------

SAME  MACRO  NAME, VARIANTS, ANS
        ANS=0
        IRP  V, <VARIANTS>
```

```
        IFIDN  <NAME>, <V>
        ANS=1
        EXITM
        ENDIF
        ENDM
    ENDM
; -----------------------------------------------------------------------
; ПОМОЩЕН МАКРОС за OUTINT (SIGN=1) и OUTWORD (=0)
; -----------------------------------------------------------------------
OUTNUM  MACRO  NUM, LENG, SIGN
           LOCAL  REGDX?
           PUSH   AX
           PUSH   DX
           PUSH   SI
           PUSH   DI
       SAME  <NUM>, <dx, DX, Dx, dX>, REGDX?
         IF  REGDX?                                   ;; OUT  DX, LENG  →
         IFB  <LENG>                                  ;; MOV  AL, LENG
            MOV   AL, 0                               ;; XCHG  AX, DX
         ELSE
            MOV   AL, LENG
         ENDIF
            XCHG   AX, DX
         ELSE                      ;; OUT  NUM, LENG  (NUM<>DX) →
            IFB   <LENG>           ;; MOV  DL, LENG
              MOV   DL, 0          ;; MOV  AX, NUM
            ELSE
              MOV   DL, LENG
           ENDIF
           MOV   AX, NUM
         ENDIF
           MOV   DH, SIGN
           CALL  PROCOUTNUM     ;;AX=NUM, DL=LENG, DH=SIGN
        POP   DI
        POP   SI
        POP   DX
         POP   AX
        ENDM
;            ВЪВЕЖДАНЕ ОТ КЛАВИАТУРАТА
;*************************************************************************
```

; *Общи забележки*:
;   *Операциите въвеждане не издават на екрана покана за вход.*
; *Набраните от клавиатурата символи се записват във междинен*
; *входен буфер, откъдето в последствие се изчитат от операциите*
; *въвеждане. Във връзка с това се допуска предварително набиране*
; *на данни, преди програмата да ги е поискала*.
; Край на въвеждането се дава с клавиш *Enter*, *който не се занася*

```
; в буфера. Курсорът се премества на следващия ред на екрана.
; Следващо въвеждане е възможно след изчитане на символите от
; буфера.
;     При въвеждане (преди да е натиснат клавиш Enter) се допуска
; редактиране на набрания текст с помощта на следните клавиши:
;     ← Backspace –  отмяна на последно набрания символ
;          Esc           –  отмяна на вече набрания текст

; ===========================================================
;  ИЗЧИСТВАНЕ НА БУФЕРА ЗА ВЪВЕЖДАНЕ ОТ КЛАВИАТУРАТА
;   Обръщение:  FLUSH
; -------------------------------------------------------------------------------------------
       EXTRN  PROCFLUSH:FAR
FLUSH  MACRO
          CALL PROCFLUSH
          ENDM

; ===========================================================
;  ВЪВЕЖДАНЕ НА СИМВОЛ (с пропуск на Enter)
;  Обръщение:  INCH  X
;      където X е от тип  r8, m8
;  Изходен параметър:  X  –  въведеният символ
; -------------------------------------------------------------------------------------------

       EXTRN PROCINCH:FAR
INCH  MACRO  X
        LOCAL  REGAX?
        SAME  <X>, <AH, AH, AH, AH>, REGAX?
        IF  REGAX?
           XCHG   AH, AL                     ;; X=AH
           MOV   AL, 0
              CALL   PROCINCH
           XCHG   AH, AL
         ELSE
           SAME  <X>, <AL, AL, AL, AL>, REGAX?
           IF   REGAX?
              MOV   AL, 0                     ;; X=AL
              CALL   PROCINCH
           ELSE
              PUSH   AX                       ;; X  –  не AH и не AL
              MOV   AL, 0
              CALL   PROCINCH

              MOV   X, AL
              POP   AX
           ENDIF
        ENDIF
        ENDM
```

```
; ==========================================================
;  ВЪВЕЖДАНЕ НА ЦЯЛО ЧИСЛО (със или без знак) ФОРМАТ ДУМА
;   Обръщение:  ININT  X ,  където X е от тип  r16, m16
;   Входен параметър:  X  –  входното число
;   Особености на входа:
;   пропускат се всички празни символи пред числото;
;   числото трябва да започва с цифра, пред която е възможен знак;
;   при минус число се въвежда като отрицателно;
;   въвеждането продължава до първата нецифра (в т.ч. до Enter);
;   при грешка ще се печата съобщение и програмата спира;
;   Грешки:  "няма цифри"   –  в числото няма нито една цифра
;              "препълване"  –  голямо по модул число
;                                     (извън интервала [-32768,+65535])
; ----------------------------------------------------------------------------

     EXTRN  PROCININT:FAR
ININT  MACRO  X
      LOCAL  REGAX?
      SAME  <X>, <AX, AX, AX, AX>, REGAX?
      IF  REGAX?
         CALL  PROCININT                        ;; X=AX
      ELSE
         PUSH   AX                              ;; X<>AX
         CALL  PROCININT
         MOV   X, AX
         POP   AX
      ENDIF
      ENDM
; ==========================================================
; Възстанови записването в листинг:
   .LIST
```

Литература

[1]. Тянев, Д.С., *Организация на компютъра*, том 1 и том 2, ISBN: 978-954-20-0412-7, ISBN 978-954-20-0413-4, Издателство на ТУ - Варна, 2008 год.

[2]. Тянев, Д.С., *Организация на компютъра – упражнения*, ISBN 954:-20-0258-0, Издателство на ТУ - Варна, 2007 год.

[3]. Тянев, Д.С., *Организация на компютъра – проектиране на логически структури*, ISBN: 954-20-0259-9, Издателство на ТУ - Варна, 2004 год.

[4]. Георгиев, Ц.G., *Микропроцесорни системи за управление,* ISBN: 978-954-20-0396-0, Издателство на ТУ - Варна, 2007 год.

[5]. Юров, В.И., ASSEMBLER, Издательство “Питер”, Москва, ISBN: 978-5-94723-581-4, 2007.

[6]. Лямин, Л. В., *Макроассемблер MASM*, Москва, Издательство “Радио и связь”, 1994.

[7]. *Turbo Assembler* , *Quick Reference Guide*, Borland International Inc.

[8]. Михалчук, В.М., Ровдо А.А., Рыжиков С.В., *Микропроцессоры* 80x86, Pentium. *Архитектура, функционирование, программирование, оптимизация кода*, Минск, Издательство “БИТРИКС”, 1994.

[9]. *Справочник по персонални компютри*, под ред. на Кирил Боянов, Издателство “Техника”, София, 1988.

[10]. Al. Schneider, *Fundamentals of IBM PC Assembly Language*, TAB Books Inc., Blue Ridge Summit, ISBN 0-8306-0710-2, 1984.

МИКРОПРОЦЕСОРНА ТЕХНИКА
И ПРОГРАМИРАНЕ НА АСЕМБЛЕР

Автори: © Димитър Стоянов Тянев
Жейно Иванов Жейнов

ISBN 978-954-20-0472-1

....

Пор. № 10/2014 — Формат 16x60x84
Тираж 200 — Печ. коли 17, 50
Протокол № 3/15.12.2008 г. — Изд. коли 16,32

Учебното пособие е освободено от ДДС
по чл. 41,т.3 от ЗДДС.

....

Университетско издателство при ТУ-Варна