ПЕРСОНАЛЕН КОМПЮТЪР
ПРАВЕЦ·82

А. ХЛЕБАРОВ

ПРОГРАМИРАНЕ НА АСЕМБЛЕР

ТЕХНИКА

Антоан Й. Хлебаров

ПЕРСОНАЛЕН КОМПЮТЪР ПРАВЕЦ·82

ПРОГРАМИРАНЕ НА АСЕМБЛЕР

ДЪРЖАВНО ИЗДАТЕЛСТВО „ТЕХНИКА“
СОФИЯ, 1988

УДК 681.3.06

В книгата са разгледани основните принципи при създаването на програми на езика АСЕМБЛЕР за персоналния микрокомпютър Правец-82. Описани са всички машинни инструкции на микропроцесора 6502, тяхното предназначение и използване, както и съответните им оператори в езика АСЕМБЛЕР. Включени са много примерни програми, които онагледяват и подпомагат усвояването на материала.

Книгата е предназначена за читатели с различна квалификация и специалност, ползуващи микрокомпютъра Правец-82: ученици, студенти, преподаватели, математици, инженерно-технически кадри, икономисти и др.

ПРЕДГОВОР ОТ НАУЧНИЯ РЕДАКТОР

Трудно е да се пише предговор за книга, в която има и твое, макар и косвено участие. Ето защо избрах по-лесния път — да опиша терминологичните си възгледи, които споделих с автора на книгата по време на многочасовите си беседи с него. Благодарен съм му за вниманието и търпението, с които той се отнесе към моите бележки, и най-вече за желанието му да внесе поне малко ред в част от терминологията по информатика на български език, която, общо взето, е в тежко състояние.

1. Относно видовете (начините за, методите за, тип на) *адресиране*. Желателно е да не се използува терминът *адресация*, който дори липсва в „Правописен речник на съвременния български книжовен език", Издателство на БАН, 1983 г. Има два основни вида адресиране: *явно* и *неявно* (подразбиращо се). Явното е *пряко* (*директно*) или *косвено* (*непряко, индиректно*). Адресирането е *съставно* (*комбинирано, изчисляемо*), ако изпълнителният адрес се получава в резултат на *събиране* (чрез *базиране, индексиране*), *конкатенация* и други действия. Относителното адресиране (по отношение на някакъв адрес) е общо понятие и не бива да се използува самостоятелно, освен ако не се зададе *базата* (*базовият адрес*). Дори *абсолютното адресиране* е относително спрямо нулевия адрес на оперативната памет (ОП). При 6502 е по-добре да не се използува понятието абсолютно адресиране, като явното адресиране при него е абсолютно, но бива *късо* (с подразбиращи се нули за допълване до пълен адрес на ОП) и *пълно* (размерът на адреса дава възможност за адресиране на всяка клетка от адресното пространство на вътрешната памет). Абсолютното адресиране не е синоним на прякото, което може да бъде и абсолютно, и относително. В много съвременни компютри няма абсолютно адресиране, като абсолютните адреси се получават чрез изчисляване.

Всяка наука започва от класифицирането на понятията си. Има твърде много видове адресиране, които са били използвани или се използуват и сега. За всеки от 13-те вида адресиране в 6502 тук даваме досега срещаните наименования (*а*), предложение за правилно пълно наименование на дадения вид адресиране (*б*) и предложения за съкратено написване и изговор на пълното наименование (*в*), в частност при общуване и отпечатване на таблици. Съкращенията са доста условни и важат само за 6502.

Неявно (подразбиращо се) адресиране

1. а) вложено, акумулаторно; б) *адресиране с подразбиране на акумулатора;* в) акумулаторно, към акумулатора.
2. а) подразбиращо се, вложено; б) *адресиране с подразбиране на регистър;* в) подразбиращо се, с подразбиране.
3. а) непосредствено; б) *непосредствено адресиране;* в) непосредствено. Това е компромис — няма непосредствено адресиране, а машинната инструкция (или операторът на АСЕМБЛЕР) съдържа *непосредствен операнд*. Адресът на последния се изпраща в програмния брояч (ПБ) при извличането на инструкция от ОП. Понякога се казва, че това е адресиране (съответно *адрес*) *на нулево ниво.*
4. а) относително; б) *относително адресиране спрямо програмния брояч;* в) относително. Преди повече от 20 години акад. Благовест Сендов предложи този вид адресиране да се нарича *диференчно*, като съответната адресна константа (*скокът*) е цяло със знак, обикновено в допълнителен код, а терминът *относително адресиране* да се запази само за днешното *адресиране с базиране*, като съответната константа е цяло без знак.

Явно адресиране

(Явно) пряко адресиране

Пряко късо адресиране

5. а) нулева страница, в нулевата страница; б) *пряко адресиране в нулевата страница;* в) късо, в нул. стр., пряко в нул. стр.

6—7. а) индексно нулева страница по X(У), индексно в нулевата страница с регистъра X(У); б) *пряко адресиране в нулевата страница с индексиране по X(У);* в) късо инд. по X(У), в нул. стр. с инд. по X(У); нул. стр., X(У).

Пряко пълно адресиране

8. а) пълно, абсолютно; б) *пряко пълно адресиране;* в) пряко пълно, пряко.

9—10. а) индексно по X(У), индексно абсолютно с регистъра X(У), б) *пряко пълно адресиране с индексиране по X(У);* в) пълно индексирано по X(У); пряко пълно, X(У).

Явно косвено адресиране

11. а) непряко, абсолютно непряко (индиректно); б) *косвено пълно адресиране;* в) косвено.

12. а) индексно непряко, индексно непряко (индиректно) с регистъра X; б) *косвено пълно адресиране с предварително индексиране по X;* в) предварително индексно по X, косвено с предв. инд. по X, косвено (адрес, X).
13. а) непряко индексно, непряко (индиректно) индексно с регистъра У; б) *косвено пълно адресиране със следващо индексиране по У;* в) следващо индексно по У, косвено със сл. инд. по У; косвено, У.

2. За да разтоварим от много значения термините *инструкция, оператор* и *команда* и да ги използуваме само с едно значение, предлагам да се наричат:

машинна инструкция, съответно *система от* (машинни) *инструкции* (да се откажем от термина *система команди* от руски) — само в машинните езици; инструкцията съдържа код на операция и операнди;

оператор — само в символичен процедурен език, вкл. и АСЕМБЛЕР; това е завършено изречение, твърдение (на англ. *statement*) в такъв език; в асемблерските езици включва етикет, служебна дума (мнемоничен код на операция), операнди, коментар, а в езиците от по-високо ниво — и други обекти;

директива — само в символичен процедурен език (вкл. и АСЕМБЛЕР) за *непроцедурни цели;* например за управление на транслирането; най-често използувано — *асемблерска директива,* която е аналог на асемблерски оператор;

команда — в символичен непроцедурен език (например в езика за управление на заданията в една ОС, в монитора на Правец-82, във входния език на един пакет приложни програми с непроцедурна обвивка, в телекомуникационен метод за достъп и др.); командата е аналог на оператор.

Забележка. Терминът *operator* се превежда на български и руски език като *знак за* (аритметична или логическа) *операция* или *знак за отношение.* В този смисъл не е правилно *служебните* (*ключовите*) *думи* в операторите (командите) на символичните езици да се наричат инструкции, команди или оператори. В частност ключовите думи биват запазени (резервирани) и незапазени. Допустимо е обаче да се казва например *оператор* LDA, но под това да се подразбира *оператор с ключова дума LDA.*

3. Трябва да се прави разлика между *машинен код на инструкция* (той заема цялата дължина на инструкцията) и *машинен код на операция* (*код*), която може да извърши една инструкция, ако бъде изпълнена. Кодът на операцията (предлагам да се казва кодът) заема само едно поле по формата на машинната инструкция, т. е. е част от машинния код на инструкцията.

Машинният код на инструкцията съответствува на понятието оператор в символичен език, по-точно — на оператор в АСЕМБЛЕР, който не е МАКРОАСЕМБЛЕР. Машинният код на операцията съответствува на служебна дума или на знак за опе-

рация или отношение в символичен език, по-точно на служебната дума (мнемоничния код на операция, мнемоничния код) в асемблерски оператор.

4. Правилно е да се казва *проверяване (проверка) и поправяне на програма* или проверяване на програма и отстраняване на грешки в нея вместо настройка (от руски) и тестване на програма, макар и по-кратко (вместо *настройка* се използува и *отладка*, също от руски). У нас настройка се използува с друг смисъл — *настройване* (настройка) *на (под) програма по параметри.* А от няколко години се използува настройка на (вярна) програма към съответните реални обекти, т. е. фино изменение на вече *проверена* програма. А тестоването е само част от проверяването на една програма.

5. Правилно е да се казва *знак*, а не символ, когато става дума за *character* (англ.) или *литера* (руски). *Символ* е по-общо понятие — например един символ може да се състои от няколко знака (във фирмената литература, таблица на символите означава таблица на идентификаторите, имената). Множественото число на знак в семиотиката и оттам — в информатиката е *знакове*, а не знаци, както е било в речевата практика досега (това решение е взето наскоро от Института за български език при БАН).

Затова вместо символна или стрингова променлива трябва да се използува *знакова променлива* (със стойност един знак), *текстова променлива* или *променлива от тип низ* (със стойност един или повече знакове).

6. Термините извиквам, викам (под)програма, извикване, викане на (под)програма, извикваща, извиквана, викаща, викана (под)програма са жаргонни. Те интуитивно се свързват с някакво преместване на програмата. Правилно е да се казва *обръщение (преход) към (под)програма, предаване на управлението на (под)програма, вход в (под)програма, задействуване, активиране, стартиране на (под)програма, активираща* или *стартираща,* както и *активирана* или *стартирана (под)програма.*

7. Има три термина с еднакъв корен и съществено различно значение: отместване, изместване и преместване (пренос).

Отместването (displacement или *offset*) е адресна константа, която се прибавя към стойността на базов регистър (базов адрес) и служи за относителното адресиране в рамките на масив от данни. По-рано отместването беше винаги число без знак, но напоследък в някои компютри може да бъде и число със знак в допълнителен код. Възможно е да се въведе и терминът *отклонение* (*със знак*), за да се запази отместването в стария му смисъл — без знак.

Изместването (*shift*) е преместване (в буквалния смисъл на думата) на информация в някаква „рамка": на един или повече битове наляво или надясно в регистър или клетка, на една или

повече позиции върху екран. Информацията, която излиза извън „рамката", може и да бъде загубена освен при използуване на флаг за пренос (*carry*), при цикличните измествания (ротациите) и др.

Преместването е всъщност *копиране* на информация от източник в приемник, например от регистър в регистър, от клетка в регистър и пр. Това не е истинско преместване, понеже информацията се запазва в източника. По-добре е да се казва *записване (запис) в регистър* или *клетка, зареждане на регистър* и пр. Вместо да се използува преместване (пренос) от един регистър в друг и т. н.

8. В англоезичната литература графичното описание на *програмно достъпните регистри* на 6502 се нарича програмен модел на този процесор. Това не е точно, тъй като в програмния модел на всеки процесор се включват и форматът на машинните му инструкции, и видовете адресиране, и форматът на данните, и др. При 6502 става дума за програмно достъпните му регистри.

9. *Инверсията в машинния адрес* (младшите байтове да са пред старшите) е свързана с апаратните особености на някои микропроцесори и миникомпютри. В повечето компютри *няма инверсия в адреса*.

10. Макросите са вид подпрограми. Понятието *подпрограма* е въведено в края на 40-те години и е начало на модулността в информатиката. По отношение на *компилирането* подпрограмите се делят на *отворени* и *затворени*. При компилирането всяко обръщение към отворена подпрограма се замества с тялото ѝ, като се прави настройка по параметри (*формалните параметри* се заместват с *фактически*). Така тялото на подпрограмата се копира многократно в зависимост от броя на обръщенията към нея и тези настроени по параметри копия се сливат с тялото на основната програма. Ако има много обръщения към отворена програма, тялото ѝ трябва да е късо — в противен случай се получава голям разход на памет. Когато в тялото на основната програма се изпълнява отворена подпрограма, се печели време, понеже не е нужно настройване по параметри.

Тялото на затворена подпрограма се компилира само един път, но параметрите се предават през време на изпълнението на основната програма всеки път, когато се срещне обръщение към подпрограмата. Поради това бързодействието при изпълнението на подпрограмата намалява за сметка на това при компилирането. Обикновено една подпрограма се оформя като затворена, когато е по-голяма и има повече обръщения към нея.

Прието е *отворените подпрограми в макроасемблерите* да се наричат *макрос*, съответно — *макроси* (оттук — макроасемблер). На български би било правилно (но необичайно) да се казва *макро* и съответно *макрос* (за мн. ч.).

11. Трябва да се прави съществена разлика между *клетка* и *дума. Клетката е място* (*поле*) в ОП, *което има адрес*. Докато *думата е данна, която може да се обработи наведнъж* (в някои компютри, като IBM, има думи с различна дължина — *полудума, дума* и *двойна дума*). Дума може да се разположи в клетка, но обратното не е възможно! Напоследък се прави грешка, като се приема, че понятията клетка и дума съвпадат. Причината за това е, че в 8-битовите микропроцесори и думата, и клетката имат еднаква дължина — един байт. Правилно е ОП да се измерва с брой клетки (най-често с дължина един байт). В този смисъл е неправилно обемът на ОП да се измерва с думи, както е при СМ ЕИМ.

Адресът е целочислено име на клетка и *клетката*, а не адресът, *има съдържание* (*стойност*). Даже когато се казва накратко съдържание (изпращане на стойност) на адрес, трябва да се подразбира съдържание на клетка с този адрес.

12. *Размерност* на един масив е броят на *индексите* (*измеренията*) му. За една матрица A_{mn} съответният масив е двумерен — неговата размерност е 2, а не $m \times n$, както понякога неправилно се пише (последното е броят на елементите му). Правилното е, че матрицата е $m \times n$, а не че размерността ѝ е толкова.

13. В асемблерите аритметичните изрази се изчисляват през време на асемблирането на първичната програма, като резултатът е адрес или адресна константа. Изразите в езиците за програмиране от високо ниво се изчисляват при изпълнението на програмата, като стойността на аритметичните изрази е число, а на логическите — логическа стойност. Стойността им се присвоява на променливи. На именуващите изрази обаче, които се използуват в някои езици, стойността е етикет, т. е. в в крайна сметка адрес от паметта, в която се намира програмата.

14. Удобна за използуване е еднокоренната двойка термини *включване* и *изключване* на елемент във и от низ, списък, стек, опашка и др. вместо вкарване, вмъкване, отстраняване и пр. Използува се и *извличане* от стек, но с това не се подчертава, че прочетеният елемент се изключва от стека.

Когато се свързват *знак* (*програма*) и *екран*, се казва *извеждане*, *изписване*, *изпращане на знак* (*програма*) *на екрана*, но в никакъв случай листване на програма.

Когато се свързват знак (програма) и клавиатура, се казва *въвеждане на знак* (*програма*) *от клавиатурата*.

15. При двоични числа в допълнителен код *може да има препълване само при „истинско" събиране* (събиране на числа с еднакви знаци или изваждане на числа с противоположни знаци), т. е. когато при окончателното събиране (понеже и изваждането се свежда до събиране) *операндите са с еднакъв знак*. В този

случай *има препълване, ако резултатът е със знак, противоположен на този на събираемите.*

16. Съгласно изискванията на Института за български език при БАН е правилно да се използува *асемблерски*, а не асемблерен.

17. Щом като супервайзор се пише с малка буква, то и *монитор* (също управляваща програма) трябва да се пише по същия начин — така би трябвало да бъде например в Правец-82. Но може да се напише *управляваща програма Монитор*, тъй като тук с голяма буква се означава име.

София, 9 май 1987 г. *Доц. к. м. н. Димитър Петров Шишков*

ПРЕДГОВОР

Масовото навлизане на българските 8-битови персонални микрокомпютри от фамилията Правец в практиката определи големия интерес към програмирането за тях. Повечето от потребителите се насочиха към езиците за програмиране от високо ниво БЕЙСИК и ПАСКАЛ. За тях бяха издадени много книги за начално запознаване с тези езици. В специализираните издания се появиха доста публикации с представяне на готови програмни продукти. След овладяването на тази начална компютърна грамотност обаче много от хората, работещи с Правец-82, потърсиха по-пълноценно използуване на богатите възможности на компютъра с усвояването на машинно ориентираните програмни езици. За съжаление литературата по тези въпроси все още е малко, а набавянето ѝ е свързано с трудности.

Тази книга има за цел да даде основни познания по програмиране на АСЕМБЛЕР. Предполага се, че читателят е запознат в общи линии с устройството и работата на персоналния микрокомпютър Правец-82, запознат е с теорията и има известен практически опит в програмирането поне на един език за програмиране от високо ниво, знае да работи с числа в различни бройни системи. Тези начални знания са необходими за по-доброто разбиране на представените в книгата примери, но с малко повече усилия и тези, които тепърва навлизат в програмирането, биха могли да се справят. В същото време за начинаещите трябва да бъде ясно, че програмирането на АСЕМБЛЕР е трудно, а понякога и доста уморително занимание. При запознаването с книгата задължително трябва да се ползува асемблер-редактор. Добре е това да бъде програмният продукт MERLYN, но в краен случай може да се използува и друг подобен. Примерите от книгата трябва да бъдат въвеждани в компютъра и след това стартирани, тъй като практическата работа пред клавиатурата и екрана на Правец-82 е единственият начин за изучаване на програмирането на АСЕМБЛЕР.

Изказвам своята благодарност на всички, които допринесоха за възникването на идеята, написването, редактирането и отпечатването на тази книга.

Авторът

1. ОСНОВНИ ПОНЯТИЯ В ПРОГРАМИРАНЕТО НА АСЕМБЛЕР

Основна част във всеки компютър, определяща действителните му възможности, е централният процесор. При персоналните микрокомпютри той е сложна интегрална схема, наричана микропроцесор, която управлява работата на цялата машина. За „мозък" на българския персонален микрокомпютър Правец-82 е избран микропроцесорът 6502. Той е 8-битов, т. е. обработва данни с дължина един байт (8 бита), използува 16 адресни линии и може да се обръща към 65 536 (2^{16}) отделни клетки от вътрешната памет. Всяка от тези клетки се определя еднозначно с число, което се нарича неин адрес. Съвкупността от адресите на клетките образува адресното пространство, а съвкупността от клетките — вътрешната памет на микрокомпютъра.

1.1. Разпределение на паметта

Паметта на Правец-82 е вътрешна и външна. Вътрешната памет се разделя на страници по 255 ($FF) клетки всяка. Обособени са две главни области — оперативна памет за запис и четене (памет със свободен достъп — RAM), и постоянна (системна) памет (памет само за четене — ROM). Първата се разполага между адресите $0000 и $BFFF и включва:

— нулева страница ($00—$FF) — тези адреси се използуват най-често, тъй като за тяхното адресиране са необходими само два шестнайсетични знака; това позволява да се съкрати размерът и да се повиши бързодействието на програмите; една част от нулевата страница се използува от интерпретатора на езика БЕЙСИК, системния монитор и дисковата операционна система (вж. прил. 1);

— страница 1 или стек ($100—$1FF) — използува се като временна памет за междинни данни; записването и четенето на данни във и от тази област се извършва под управлението на специален регистър — указател на стека;

— страница 2 ($200—$2FF) — използува се като входен буфер при въвеждане на информация от клавиатурата или от дискета;

— страница 3 — част от нея ($300—$3CF) е свободна за потребителя и там обикновено се разполагат малки програми на машинен език; останалите клетки се използуват от дисковата операционна система, интерпретатора на БЕЙСИК и системния монитор за съхраняване на началните адреси на някои важни системни подпрограми;

— страници 4, 5, 6 и 7 ($400—$7FF) — първа страница от екранната памет за текст и графика с ниска разделителна способност;

— страници 8, 9, 10 и 11 ($800—$BFF) — втора страница от екранната памет за текст и графика с ниска разделителна способност; използува се също за разполагане на програми; началният адрес на програмите на БЕЙСИК, ако не е определен допълнително, се установява на $800;

— страници 12—31 ($C00—$1FFF) — свободна област;

— страници 32—63 ($2000—$3FFF) — първа страница от екранната памет за графики с висока разделителна способност;

— страници 64—95 ($4000—$5FFF) — втора страница от екранната памет за графики с висока разделителна способност;

— страници 96—191 ($6000—$BFFF) — свободна област, в част от която (над $9600) се разполага дисковата операционна система.

Системната памет (ROM) се намира в адресното пространство от $D000 до $FFFF. Нейното съдържание е постоянно и не може да се променя програмно. Тук са разположени машинните подпрограми, образуващи системния монитор и интерпретатора на БЕЙСИК.

Паметта за входно-изходни операции заема адресите от $C000 до $C0FF. Клетките от нея служат като ключове за управление на входните и изходните устройства (клавиатура, дискови устройства и видеомонитор).

1.2. Регистри и флагове

При работата си микропроцесорът 6502 използува шест регистъра — пет от тях са 8-битови и един е 16-битов (вж. прил. 5). Всеки от тях има точно определено предназначение:

1. Акумулатор (А) — преди изпълнението на повечето от операциите в него се разполага единственият или един от операндите, като резултатът най-често остава също в него. През акумулатора става прехвърлянето на данни от една област на паметта в друга. Акумулаторът е най-често използуваният регистър с общо предназначение.

2. Регистър X — използува се за достъп до елементите на едномерни масиви и низове. Осъществяването на този достъп се нарича индексиране, а регистърът X се нарича още индексен регистър.

3. Регистър Y — и този регистър, както и регистър X, е предназначен за индексиране. Наличието на два индексни регистъра позволява извършването на действия, като присъединяване и разделяне на низове, работа със сложни масиви и др.

4. Указател на стека (S) — използува се при работа с подпрограми, за запазване на междинни данни и др. Тъй като заема само 8 бита, с него могат да се зададат 256 различни клетки от адресната област. Това са адресите в стека от $100 до $1FF.

5. Програмен брояч (PC) — 16-битов регистър, указващ инструкцията, която ще се изпълнява. Има двойна дължина, тъй като за достъп до всяка клетка на паметта (общият им брой е 65 536) е необходим 16-битов адрес.

6. Регистър за състоянието на процесора (P) — това са 8 бита, които съхраняват информация за определени работни състояния на процесора. Тези битове се наричат флагове и се отбелязват по реда си отляво надясно с N,V,B,D,I,Z и C. Между флаговете B и V съществува още един бит, но той не се използува. Казва се, че съответният флаг е вдигнат, установен или зареден, когато му е присвоена стойност 1, и изчистен или нулиран, когато му е присвоена стойност 0. Регистърът за състоянието включва следните флагове:

— флаг за пренос (C) — отразява получаването на пренос или заем при извършването на аритметичните операции събиране и изваждане;

— флаг за нула (Z) — отразява получаването на нулев резултат след последната изпълнена операция; този флаг не може да се установява пряко от програмиста;

— флаг за прекъсване (I) — контролира действието на прекъсващия сигнал и когато е установен от програмиста или автоматично от микропроцесора, забранява се действието на прекъсващия сигнал, а когато е изчистен, прекъсването е възможно;

— флаг за десетичен режим (D) — показва в какъв режим се извършват аритметичните операции; при вдигнат флаг (т. е. D = 1) събирането и изваждането се извършват с двоично кодирани десетични числа, а при изчистен флаг — с двоични числа със или без знак; при определено състояние на флага всички аритметични операции се извършват в съответния режим, докато това състояние не бъде променено от друга инструкция;

— флаг за програмно прекъсване (B) — вдигането на този флаг показва, че последното прекъсване се дължи на изпълнението на инструкция за програмно прекъсване; състоянието му не може да се задава от програмиста;

— флаг за препълване (V) — сигнализира за препълване при събиране или изваждане на еднобайтови двоични числа със знак; този флаг се установява, когато резултатът от действията с тях е извън областта на числата от −128 до +127; установя-

ването става само автоматично, а изчистването може да се извърши и от програмиста;

— флаг за отрицателна стойност (N) — установява се, когато получената стойност при аритметична обработка или обмен на данни е отрицателна; този флаг е винаги равен на седмия бит от резултата.

Всеки от тези седем флага носи определена информация за състоянието на процесора през време на изпълнението на програмата. Проверката на стойностите им, съчетана с използуването на инструкции за условен преход, позволява да се изпълняват циклични действия или да се определят следващите поредици от инструкции.

1.3. Инструкции и кодове на операциите

Инструкциите, които съобщават на микропроцесора 6502 каква операция да извърши, започват с 8-битови кодове на операциите, които се съхраняват в паметта на компютъра. Тъй като всеки от тях е 8-битов, възможни са най-много 256 различни кода на операциите. Всъщност съществуват само 151 кода, които се разпознават от микропроцесора 6502 като действителни. Останалите не се възприемат от него, наричат се недействителни и предизвикват програмни прекъсвания.

Инструкциите се съхраняват в паметта на компютъра по същия начин, както и данните. Един байт се възприема като инструкция или част от нея само ако програмният брояч може да го посочи (т. е. да съдържа адреса му) в определен момент от изпълнението на програмата. Предполага се, че програмата се съхранява, разположена в последователни клетки на паметта (с някои изключения). Програмният брояч се зарежда с адреса на първата инструкция. Да предположим, че тя е еднобайтова. Процесорът прочита инструкцията и я изпълнява. Програмният брояч се увеличава с единица, така че вече да задава адреса на втората инструкция. Тази инструкция се прочита от процесора и цикълът се повтаря. Компютърът не различава инструкциите от данните. Всичко, което показва програмният брояч, се възприема като инструкция към процесора. Много от инструкциите изискват повече от един байт — два или три. След изпълнението на такива инструкции програмният брояч трябва да се увеличи с толкова, че да прескочи данните от инструкцията. Във втория (и третия) байт на някои инструкции се разполага адресът на единствения или единия от операндите или адресна константа (например отместване, адрес за преход и др.)

Множеството или наборът от инструкции, които микропроцесорът може да изпълнява, определя и възможностите на целия компютър. В зависимост от избраните критерии инс-

трукциите могат да се разпределят в различни групи. Най-разпространена е комбинираната класификация според предназначението на инструкциите, данните, с които работят, и мястото на обработката им.

1. Инструкции за обмен на данни
   - обмен между регистрите:
     TAX, TAY, TXA, TYA, TXS, TSX;
   - обмен между паметта и регистрите:
     LDA, LDX, LDY, STA, STX, STY;
   - обмен на данни между процесора и стека:
     PLA, PHA, PLP, PHP.
2. Инструкции за обработка на данни
   - аритметични операции:
     ADC, SBC;
   - промяна с единица:
     INX, INY, INC, DEX, DEY, DEC,
   - изместване и ротация:
     ASL, LSR, ROL, ROR;
   - логически операции:
     AND, ORA, EOR, CMP, CPX, CPY, BIT.
3. Инструкции за управление на състоянието
   - установяване на флаговете:
     SEC, SEI, SED;
   - нулиране на флаговете:
     CLC, CLI, CLD, CLV;
   - програмно прекъсване:
     BRK;
   - празна операция:
     NOP.
4. Инструкции за програмни преходи
   - безусловен преход:
     JMP,
   - условен преход:
     BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC;
   - преход към и от подпрограма:
     JSR, RTS, RTI.

1.4. Начини на адресиране

Микропроцесорът 6502 използува 56 различни инструкции, на които съответстват 151 действителни кода на операции. Това е така, защото някои операции могат да се изпълнят по няколко начина. Например един тип операция е зареждане на акумулатора с 8-битова стойност. Тя се означава с LDA. Съществуват обаче няколко различни машинни инструкции тип LDA. Акумулаторът може да бъде зареден с константа, със стойност

от даден адрес от паметта, с променлива или елемент от масив. Всички тези инструкции имат нещо общо — в резултат от изпълнението им акумулаторът ще бъде зареден с нова стойност, но начинът за това зареждане е различен. Поради това се използуват и различни инструкции LDA. Разновидностите на задаване на данните за обработка се наричат начини за адресиране. Инструкцията показва какво трябва да извърши компютърът, а начинът на адресиране показва откъде да се вземат данните. Микропроцесорът 6502 използува следните 13 начина на адресиране:

1. Непосредствено адресиране — инструкцията използува 8-битовата константа, разположена непосредствено след кода на операцията. Инструкциите с този начин на адресиране са двубайтови — един байт за типа на операцията и един байт за данните.

2. Пряко пълно адресиране — инструкцията използува променлива или стойност, записана някъде в паметта на компютъра, като необходимите клетки се посочват с абсолютните им адреси. Освен един байт за кода на операцията са необходими и два байта за адреса, като първият от тях съдържа младшия, а вторият — старшия разряд. Това разположение е характерно само за някои микропроцесори, в това число 6502 и Интел 8080.

3. Пряко адресиране към нулевата страница (късо адресиране) — това е особена форма на прякото пълно адресиране. Необходимият адрес се задава само с един байт, показващ младшия разряд. Използуват се 256 адреса от $00 до $FF. В адресното пространство те отговарят на клетките от нулевата страница. Инструкциите с такъв начин на адресиране се изпълняват по-бързо от инструкциите с пряко пълно адресиране. Нулевата страница се използува най-често за запазване стойностите на променливи, както от системните програми, така и от потребителските.

4. Адресиране с подразбиране на регистри (подразбиращо се адресиране) — използува се при инструкции, които извършват действия със стойностите в регистрите или флаговете за състоянието. Инструкциите са еднобайтови, понеже не се изисква явното посочване на адрес от паметта или константа, а самите инструкции показват с данните от кои регистри се работи.

5. Адресиране с подразбиране на акумулатора (акумулаторно адресиране) — за операнд се взема съдържанието на акумулатора. Тези инструкции са еднобайтови и не изискват посочването на адрес или константа.

6. Пряко пълно адресиране с индексиране по X (индексно адресиране по X) — адресът на клетката, от която се извличат данните, се определя от съдържанието на двата байта, следващи кода на операцията, към което се прибавя съдържанието на ре-

гистъра X. Така достъпът до еднобайтовите елементи на масив с размер, по-малък от 256, може да се осъществи, като регистърът X се зареди с желаната стойност на индекса, а акумулаторът се зареди с първия елемент на масива, индексиран с X.

7. Пряко пълно адресиране с индексиране по Y (индексно адресиране по Y) — разликата от предишния начин на адресиране е само в това, че се работи с индексния регистър Y.

8. Пряко адресиране към нулевата страница с индексиране по X (късо индексно адресиране по X) — съдържанието на регистъра X и стойността на байта в адресното поле определят адрес в нулевата страница, от който се извличат данните.

9. Пряко адресиране към нулевата страница с индексиране по Y (късо индексно адресиране по Y) — данните се извличат от адрес в нулевата страница, определен като сума от съдържанието на регистъра Y и стойността на байта в адресното поле. При използуването на този, както и на предишния начин на адресиране, полученият ефективен адрес е винаги в границите на нулевата страница, тъй като събирането се извършва с пренебрегване на преноса.

10. Относително адресиране спрямо програмния брояч (относително адресиране) — използува се при условен преход. Байтът в адресното поле на инструкцията се разглежда като число със знак, което определя т. нар. отместване. То се прибавя към съдържанието на програмния брояч в момента и се получава адресът на инструкцията, към която трябва да се осъществи преходът. Отместването не може да бъде по-малко от −128 и по-голямо от +127.

11. Косвено пълно адресиране (косвено адресиране) — стойността на двата байта в адресното поле на инструкцията указва клетката в паметта, стойността на която се използува като младши разряд, а стойността на следващата клетка — като старши разряд на числото, даващо действителния (ефективния) адрес на търсените данни.

12. Косвено пълно адресиране с предварително индексиране по X — съчетава възможностите на косвеното с индексното адресиране. Стойността на байта в адресното поле на инструкцията се събира със съдържанието на индексния регистър X, като преносът се пренебрегва. Полученият резултат адресира клетка от нулевата страница. Тази клетка и следващата съдържат ефективния адрес на операнда.

13. Косвено пълно адресиране с последващо индексиране по Y — стойността на байта в адресното поле на инструкцията указва клетка от нулевата страница на паметта. Тази клетка и следващата съдържат адреса на началото на блок от данни, които се индексират по Y. Ефективният адрес на всяка от тези данни се намира, като към началния адрес се прибави съдържанието на регистъра Y.

1.5. Програма асемблер и асемблер-редактор. Програмиране на езика АСЕМБЛЕР и на машинен език

За да се накара процесорът да извърши някакво действие, трябва да се използува точно определено число — код на желаната операция. Съществуват обаче 151 такива числа и запомнянето им би било доста уморително. По-добре е да се използуват изрази като „да се зареди акумулаторът с константа $FF" или „да се запише съдържанието на акумулатора в адрес $6782". За тази цел се използува езикът за програмиране АСЕМБЛЕР. В него за означаване на такива изрази се използуват т. нар. мнемонични кодове, представляващи трибуквени съкращения от името на инструкцията (например за зареждане на акумулатора се използува LDA — от англ. LOAD ACCUMULATOR).

Друга важна особеност при програмирането на езика АСЕМБЛЕР е използуването на символични адреси. Шестнайсетичните адреси се означават с буквено-цифрови имена, с които се работи при съставянето, проверките и поправките на програмите, а определянето на точните адреси се отлага до последния момент. Символичният запис на една машинна инструкция в езика АСЕМБЛЕР се състои от мнемоничен код на операцията и операнд.

След съставянето на първичната програма на езика АСЕМБЛЕР е необходима програма, с която тя да се въведе в компютъра и след това всеки асемблерски оператор да се преобразува в съответната машинна инструкция. Тази програма се нарича асемблер, а процесът на преобразуване — асемблиране.

Съществуват много и различни асемблери, програмно съвместими с Правец-82. Най-известните и широко разпространени са APPLE II ASSEMBLER EDITOR, BIG MAC, LIZA, ASSEMBLER 3.0, MINI — и др. В повечето от тях са заложени и възможности за редактиране и проверка на въвежданите програми, поради което се наричат асемблер-редактори. Всички те са разработени на основата на общи принципи, но между тях съществуват и някои различия. Поради това след избора на един от възможните асемблер-редактори е необходимо внимателно да се проучи съпровождащата го документация. Най-богати възможности има асемблер-редакторът MERLYN, както и новата му версия MERLYN—PRO. Заради изключително широката популярност на този програмен продукт всички примери в тази книга са представени чрез него. Допълнително съображение за това е и фактът, че у нас се разпространява програмно съвместимият с него асемблер-редактор МЕЛИН.

В процеса на асемблиране първичната програма, съставена от оператори на АСЕМБЛЕР и наричана първичен код, се превръ-

ща в програма на машинен език и се нарича обектен или изпълним код. Обикновено асемблер-редакторите съхраняват и двата кода — първичната (асемблерската) програма като двоичен или текстов файл за по-нататъшни редакции и поправки, и обектния код като двоичен файл.

Редовете на програмите, написани на АСЕМБЛЕР, се състоят от няколко полета, разделени с интервали. Някои от тези полета не са задължителни или не се използуват от дадения асемблер-редактор. В първото поле отляво надясно се записва (в повечето случаи автоматично) поредният номер на програмния ред. Този номер е само за удобство на програмиста при въвеждане и редактиране на програмата. Следващото поле се използува за т. нар. етикети — кратки имена, които служат за предаване на управлението в дадено място от програмата. При програмиране на машинен език вместо етикети се използуват абсолютните шестнайсетични адреси от паметта, към които трябва да се осъществи преходът. Третото поле, което за разлика от посочените дотук е задължително, съдържа трибуквения мнемоничен код на инструкцията (вж. прил. 4). В четвъртото поле се записват адресът и при необходимост — използуваният начин на адресиране. Различните асемблер-редактори използуват различни знаци за означаване начина на адресиране и затова е необходимо внимателно запознаване със съпровождащата документация. В последното поле се записват коментарите на програмиста. Те се отделят със специален знак и служат за онагледяване на действието или съдържанието на съответния програмен ред.

Програмирането за микропроцесора 6502 може да се осъществи и непосредствено на машинен език без използуването на АСЕМБЛЕР. За това е необходимо да се влезе в режим монитор и да се въвеждат направо машинни инструкции. В някои случаи този начин има своите предимства, но общо взето, е по-бавен и крие опасност от грешки. В режим монитор съществуват някои команди, които улесняват програмирането на машинен език. Особено полезна е командата *адрес* L. Тя дисасемблира, т. е. извършва действия, обратни на тези на асемблера, и показва на екрана 20 инструкции, започващи от дадения адрес. Всяка непосредствено следваща команда L показва по още 20 следващи инструкции и обикновено се използува за проверка на програмите. За това служи и командата MK—E, която показва съдържанието на акумулатора и регистрите.

При дисасемблиране на обектен код информацията се извежда на екрана на компютъра в три полета. В първото се показва шестнайсетичният адрес от паметта, в който се записва програмният ред. Останалите две показват мнемоничния код и адреса и/или начина на адресиране. При това всички етикети и променливи са заменени с техните конкретни шестнайсетични стойности.

1.6. Етикети, променливи и асемблерски директиви

Повечето от асемблер-редакторите предлагат допълнителни улеснения в работата на програмистите. Най-съществени са използуването на етикети и променливи. Етикетите са кратки съчетания от букви и цифри, като първият знак винаги е буква. С тях се означава редът в програмата, към който трябва да се предаде управлението при изпълнение на преход. В някои от редакторите съществуват допълнителни ограничения, например размер до 13 знака, използуване само на латински букви и др. За всички обаче е в сила изискването да не се използуват като етикети съкращения, които съвпадат с мнемоничните кодове на операциите, т. е. не трябва да се употребяват етикети като PHP, BCC и т. н. На всеки етикет от асемблерската програма съответствува точно определен адрес от получения след асемблирането обектен код.

По подобен начин се използуват и образуват и имената на променливите, но докато с етикетите се именуват адреси от паметта, с променливите се означават числови стойности, разположени в паметта.

За работа с етикети и променливи в асемблер-редакторите се използуват допълнителни команди, наречени директиви. Те не генерират обектен код, но оказват влияние на процеса на асемблиране. В приложение 6 са дадени директивите на асемблер-редактора MERLYN, а използуването на някои от тях ще бъде показано по-нататък с примери.

Друго улеснение за програмиста е възможността за включване на аритметични изрази в адресното поле. Обикновено се използуват действията събиране и изваждане, но по-усъвършенстваните асемблер-редактори (например MERLYN, BIG MAC и др.) позволяват работа и с умножение, деление и някои логически операции. Някои от изразите се изчисляват още по време на въвеждането и редактирането, а други — по време на асемблирането на първичната програма, като резултатът е адрес или адресна константа. Това е съществено различие от езиците за програмиране от високо ниво, в които изразите се изчисляват по време на изпълнение на програмата, като стойността на аритметичните изрази е число, а на именуващите изрази — етикет (или в крайна сметка адрес). За използуване на изразите също е необходима справка с документацията на избрания от потребителя редактор.

2. РАБОТА С АКУМУЛАТОРА И РЕГИСТРИТЕ

Съществено предимство на езика АСЕМБЛЕР в сравнение с езиците за програмиране от високо ниво е възможността за изпълнение на операции непосредствено със съдържанието на акумулатора и регистрите. По този начин програмистът може да контролира стойността на всеки байт независимо дали се намира в акумулатора, индексните регистри или клетка от оперативната памет.

2.1. Обмен на данни между регистрите и паметта

Едни от най-често използваните инструкции са тези за зареждане на акумулатора или индексните регистри с някаква стойност и записване на съдържанието им в паметта. В АСЕМБЛЕР за микропроцесора 6502 на тези инструкции съответстват 6 типа оператори със следните мнемонични кодове на операциите:

LDA — зареждане на акумулатора с константа или със съдържанието на клетка от паметта;
LDX — зареждане на регистъра X с константа или със съдържанието на клетка от паметта;
LDY — зареждане на регистъра Y с константа или със съдържанието на клетка от паметта;
STA — записване на съдържанието на акумулатора в клетка от паметта;
STX — записване на съдържанието на регистъра X в клетка от паметта;
STY — записване на съдържанието на регистъра Y в клетка от паметта.

С всеки от тях могат да се използват различни начини на адресиране. При изпълнението на съответните инструкции един байт се копира от клетка на паметта в един от индексните регистри или акумулатора — за първите три инструкции, или в обратна посока — за останалите три. Съдържанието на източника остава непроменено. С помощта на тези оператори могат да се прехвърлят данни и между различни клетки на паметта, като за посредник се използва акумулаторът или някой от индексните регистри. Следната кратка програма прехвърля съдържанието на адреси $301, $302 и $303 в адреси $311, $312 и $313 по три различни начина — през акумулатора и през двата индексни регистъра.

```
1 LDA $301    ; зарежда акумулатора със
              съдържанието на адрес $301
2 LDX $302    ; зарежда регистъра X със
              съдържанието на адрес $302
3 LDY $303    ; зарежда регистъра Y със
              съдържанието на адрес $303
4 STA $311    ; записва съдържанието на
              акумулатора в адрес $311
5 STX $312    ; записва съдържанието на
              регистъра X в адрес $312
6 STY $313    ; записва съдържанието на
              регистъра Y в адрес $313
```

С операторите за зареждане на акумулатора и индексните регистри могат да се въвеждат и числови константи (те се означават със знака # пред стойността). Единствената разлика е, че в този случай се използва непосредствено адресиране. Например след изпълнението на

```
LDA #$FF
```

стойността в акумулатора ще бъде $FF (255).

2.2. Обмен на данни между регистрите

По подобен начин могат да се обменят и данни между различните регистри. Използуват се операторите със следните мнемонични кодове:

TAX — прехвърляне съдържанието на акумулатора в регистъра X;

TAY — прехвърляне съдържанието на акумулатора в регистъра Y;

TXA — прехвърляне съдържанието на регистъра X в акумулатора;

TYA — прехвърляне съдържанието на регистъра Y в акумулатора;

TXS — прехвърляне съдържанието на регистъра X в указателя на стека;

TSX — прехвърляне съдържанието на указателя на стека в регистъра X.

Използува се адресиране с подразбиране, т. е. не се изисква явно посочване на адрес от паметта. Съдържанието на регистъра, от който се вземат данните, не се променя. Тъй като няма инструкция за прехвърляне на данни от регистъра X в регистъра Y или обратно, за целта може да се използува последователността:

```
1 TXA      или      1 TYA
2 TAY               2 TAX
```

2.3. Увеличаване и намаляване с единица

След като данните се заредят в някой от регистрите и се запишат в необходимото място от паметта, с тях могат да се извършат различни действия. Най-простите от тях са увеличаване и намаляване с единица. За това се използуват операторите със следните мнемонични кодове на операциите*:

INX — увеличаване с единица на текущата стойност на регистъра X;
INY — увеличаване с единица на текущата стойност на регистъра Y;
INC — увеличаване с единица на текущата стойност на клетка от паметта;
DEX — намаляване с единица на текущата стойност на регистъра X;
DEY — намаляване с единица на текущата стойност на регистъра Y;
DEC — намаляване с единица на текущата стойност на клетка от паметта.

При изпълнението на тези инструкции могат да се изменят само флаговете N и Z, а останалите остават непроменени. Флагът за отрицателен резултат N се установява, ако след изпълнението на инструкцията се получи единица в седмия бит на резултата, и се изчиства в обратния случай. Флагът за нулев резултат Z се установява, когато резултатът е точно равен на нула и се нулира във всички останали случаи. Съществена особеност на изпълнението на инструкциите за увеличаване и намаляване с единица е това, че никога не се получава препълване. При увеличаване на стойност $FF с 1 се получава $00, а при намаляване на $00 с 1 резултатът е $FF. Операторите от този вид се използуват главно за работа с броячи и за следене на адресите при работа с индексно адресиране.

3. ОРГАНИЗИРАНЕ НА ПРЕХОДИ И ЦИКЛИ

3.1. Оператори за безусловен преход

Когато трябва да се промени ходът на програмата независимо от състоянието на други параметри, се използува операторът за безусловен преход

*По-нататък за краткост терминът мнемоничен код на операцията е пропускан, но винаги когато се споменава например оператор LDA, изпуснатият термин трябва да се подразбира.

JMP — предаване на управлението на програмата на нов адрес.

Използува се пряко пълно или косвено адресиране. И в двата случая адресната част на оператора определя (пряко или непряко) новия адрес, който се зарежда в програмния брояч и от който трябва да продължи изпълнението на програмата.

В следният пример се изпълнява непрекъснато едно и също действие — зарежда се акумулаторът последователно от адреси $C054 и $C055. Така се извършва смяна между двете страници на екранната памет. Както се вижда, в адресната част на оператора за безусловен преход могат да се използуват и етикети.

```
1 ETIKET    LDA #$C054
2           LDA #$C055
3           JMP ETIKET
```

3.2. Оператори за промяна на състоянието на флаговете

В резултат от изпълнението на някои операции съответните битове от регистъра за състоянието променят автоматично стойността си от 0 на 1 или обратно. Съществуват оператори, с които програмистът може непосредствено да влияе върху флаговете:

CLC — изчистване на флага за пренос;
SEC — установяване на флага за пренос;
CLI — изчистване на флага за прекъсване;
SEI — установяване на флага за прекъсване;
CLD — изчистване на флага за десетичен режим;
SED — установяване на флага за десетичен режим;
CLV — изчистване на флага за препълване.

Съответните им инструкции са еднобайтови и използуват адресиране с подразбиране. При изпълнението им се променя състоянието на зададения флаг, а съдържанието на регистрите остава непроменено.

3.3. Оператори за сравняване

Тези оператори дават възможност в даден момент от изпълнението на програмата да се сравнят две стойности, без да се променя съдържанието на регистрите. В зависимост от получения резултат програмата може да се разклони, ако след тези оператори се използуват оператори за условен преход. Според това, къде са разположени сравняваните стойности, се използува един от следните оператори:

CMP — сравняване на константа или на съдържанието на клетка от паметта със съдържанието на акумулатора.

CPX — сравняване на константа или на съдържанието на клетка от паметта със съдържанието на индексния регистър X;

CPY — сравняване на константа или на съдържанието на клетка от паметта със съдържанието на индексния регистър Y.

Могат да се използуват няколко различни начина на адресиране — непосредствено, пряко пълно или непълно, индексно.

При изпълнението на тези инструкции втората стойност се изважда от първата (например съдържанието на посочената клетка на паметта от съдържанието на акумулатора при CMP). Резултатът не се запазва никъде, но състоянието на флага за нула и на флага за пренос се променя по следния начин:

— ако първата посочена стойност е по-малка от втората, флагът за пренос и флагът за нула се изчистват;

— ако първата посочена стойност е по-голяма от втората, флагът за пренос се установява, а флагът за нула се изчиства;

— ако двете стойности са равни, двата флага се установяват.

Тези промени са обяснени в коментарите към следния пример:

```
1 PR    CLC          ; изчистване на флага за пренос
2       LDA #$05
3       STA $FF
4       LDX #$01
5       LDY #$0F
6       CMP $FF      ; двете стойности са равни C=1 Z=1
7       CPX $FF      ; първата стойност (тази в X) е
                     по-малка C=0 Z=0
8       CPY $FF      ; първата стойност (тази в Y) е
                     по-голяма C=1 Z=0
```

3.4. Оператори за условен преход

В много случаи се налага в алгоритъма на програмата да се включат разклонения, като се използуват регистърът за състоянието и оператори за условен преход:

BEQ — преход при нулев резултат (установен флаг за нула);

BNE — преход при ненулев резултат (изчистен флаг за нула);

BCS — преход при наличие на пренос (установен флаг за пренос);
BCC — преход при отсъствие на пренос (изчистен флаг за пренос);
BMI — преход при отрицателен резултат (установен флаг за отрицателна стойност);
BPL — преход при положителен резултат (изчистен флаг за отрицателна стойност);
BVS — преход при препълване (установен флаг за препълване);
BVC — преход при отсъствие на препълване (изчистен флаг за препълване).

Всички те използуват относително адресиране. При изпълнението на съответните им инструкции се проверява дали състоянието на точно определен бит от регистъра за състоянието отговаря на условието, заложено в инструкцията (например дали Z=1 за BEQ). Ако условието е изпълнено, извършва се преход на нов адрес. Двата байта в адресното поле на инструкцията се разглеждат като цели числа със знак в границите от −128 до +127 ($01—$FF) и представляват отместване. Това отместване се прибавя към съдържанието на програмния брояч след инструкцията за условен преход. Резултатът показва адреса на следващата инструкция, която трябва да се изпълни. Ако състоянието на съответния флаг не отговаря на условието на проверката, всички тези действия не се извършват, а се изпълнява следващата по ред инструкция.

Използуването на цели числа със знак е разгледано подробно по-нататък. Тук може само да се отбележи, че отрицателните числа от −1 до −128 се представят с шестнайсетичните числа от $81 до $FF в двоично допълнение и най-старшият им бит е 1. Положителните стойности са от $01 до $7F и най-старшият им бит е 0. Обикновено това отместване е достатъчно, но ако се налага, ограниченията могат да бъдат преодолени. Необходимо е просто да се използува оператор за условен преход при обратното условие и оператор за безусловен преход. Така например, ако в оператора

```
BNE ET
```

преходът към посочения етикет е извън границите от −128 до +127, може да се използува последователността

```
1        BEQ STOP
2        JMP ET
3 STOP   ...
```

3.5. Организиране на цикли

С използването на операторите за сравняване и условен преход е възможно да се организира многократното повторение на определен участък от програмата. Най-напред се избира брояч на цикъла и му се присвоява начална стойност. Обикновено за тази цел се използват индексните регистри или клетки от нулевата страница на паметта. След това се извършват повтарящите се действия (тяло на цикъла) и стойността на брояча се изменя с необходимата стъпка. Проверява се дали стойността на брояча съответствува на желаната крайна стойност и ако това условие не е изпълнено, цикълът се повтаря.

```
1        LDX #$00
2 CI     STA $C030
3        INX
4        CPX #$FF
5        BNE CI
6 PR     ...
```

В този пример за брояч на цикъла е избран индексният регистър X, началната стойност е 0, а крайната — $FF (255). Така действието прехвърляне на съдържанието на акумулатора в адрес $C030 ще се повтори 255 пъти. По подобен начин могат да се организират и вложени един в друг цикли. При тях се работи с повече от един брояч, като след изпълнението на най-вътрешния цикъл се променя броячът на следващия и т. н. до изпълнението на най-външния цикъл.

При изпълнението на горната програма, би трябвало да се издаде звук, тъй като адресът $C030 контролира високоговорителя и всяко обръщение към тази клетка променя състоянието на мембраната му. Поради голямата скорост на изпълнението този звук е с много голяма честота и е трудно да бъде възприет. Налага се в програмата да се включи допълнителен празен цикъл, който да забави изпълнението. В такива случаи много подходящ е операторът

NOP — извършване на празна операция (престой).

С него генерирането на звук може да стане и така:

```
1        LDA #$FF
2        STA $FF
3 VN     LDX $00        ;начало на външния цикъл
4        STA $C030
5 VTR    DEX            ; вътрешен
6        NOP            ; празен
7        BNE VTR        ; цикъл
8        DEC $FF
9        BNE VN         ; край на външния цикъл
```

За брояч на външния цикъл се използува клетката от паметта с адрес $FF, като началната му стойност е $FF (255), а крайната 0. За брояч на вътрешния празен цикъл е избран индексният регистър X. Неговата начална стойност зависи от съдържанието на адрес $00, а крайната му стойност е също 0. От това, колко пъти ще се изпълни вътрешният цикъл, преди да се изпълни веднъж външният, зависи и височината на звука.

3.6. Оператори за прекъсване

Често се налага да се наруши последователният ред на изпълнение на инструкциите. Една от възможностите е напълно да се прекрати действието на програмата и управлението да се прехвърли на потребителя. Операторът е

BRK — програмно прекъсване.

При изпълнението на съответната инструкция се издава кратък звуков сигнал и на екрана се изписва адресът, в който е настъпило прекъсването, увеличен с две, и текущото съдържание на регистрите. Това е необходима информация при проверка на програмите.

4. АРИТМЕТИЧНИ ОПЕРАЦИИ

Изчисленията чрез аритметичните операции съставят голяма част от работата на всеки компютър. Всъщност те са само две — събиране и изваждане, а всички останали — умножение, деление, степенуване, коренуване, логаритмуване и т. н., се изпълняват чрез тях. Микропроцесорът 6502 позволява извършването на три типа аритметични операции в зависимост от начина на представяне на числата, с които се работи — аритметични операции с числа без знак, с числа със знак и с двоично кодирани десетични числа.

4.1. Аритметични операции с числа без знак

При този тип аритметични операции се работи с т. нар. числа без знак. Това са всички цели числа, които могат да се представят двоично с осем бита, т. е. от 00000000 ($0 или 0) до 11111111 ($FF или 255). Резултатите от изчисленията винаги са ограничени от тези две стойности.

4.1.1. Събиране на числа без знак

Инструкцията, с която се извършва аритметичната операция събиране и за трите типа числа, и съответният оператор са:

ADC — събиране на съдържанието на акумулатора със константа или със съдържанието на клетка от паметта и преноса от предишната операция.

Използуват се различни начини на адресиране — пряко пълно, непосредствено, пряко в нулевата страница, индексно, пряко в нулевата страница с индексиране, косвено с предварително или с последователно индексиране. Резултатът от събирането се запазва в акумулатора.

Важна особеност на тази инструкция е, че в образуването на сумата участвува и флагът за пренос. В следните два примера, въпреки че се събират едни и същи числа ($02 + $01), се получават различни резултати (съответно $03 и $04):

```
1 CLC          ; пренос 0
2 LDA #$02
3 ADC #$01
4 BRK
```

```
1 SEC          ; пренос 1
2 LDA #$02
3 ADC #$01
4 BRK
```

Това се дължи на различния пренос от предишната операция (CLC и пренос 0 — в първия пример и съответно SEC и пренос 1 — във втория). Ето защо при събиране на числа без знак е необходимо предварително да се изчисти флагът за пренос. Така, както е показано в първия пример, се осигурява винаги получаването на верен резултат.

Друга важна особеност при събиране на числа без знак е това, че резултатът е винаги в границите от $00 до $FF. Ако получената сума е по-голяма от $FF, в акумулатора остават само младшите осем бита от числото и се установява флагът за пренос. Така например след изпълнението на поредицата от инструкции, съответни на операторите:

```
1 CLC          ; изчистване на преноса
2 LDA #$FF
3 ADC #$05
4 BRK          ; печат на резултата
```

полученият резултат ще бъде $04 и пренос 1. При необходимост проверката за препълване може да се извърши, като се използува инструкцията за условен преход в зависимост от състоянието на флага за пренос. Например:

```
1 CLC          ; изчистване на преноса
2 LDA #$AB     ; акумулаторът се зарежда с $AB
3 ADC $FE      ; събира се със съдържанието на ад-
                 рес $FE
```

```
4        BCS PREP; преход при установен флаг за пренос
5        BRK     ; сумата е по-малка от $FF — печат на резултата
6 PREP ...       ; сумата е по-голяма от $FF — други действия
```

При наличие на препълване със сигурност може да се твърди, че сумата е в границите от $100 до $1FE. Ако в горния пример съдържанието на адрес $FE е $60, резултатът в акумулатора ще бъде $1B, а действителната сума би трябвало да бъде $11B. В този случай флагът за пренос може да се разглежда като допълващ бит на полученото след събирането число.

4.1.2. Изваждане на числа без знак

Аритметичната операция изваждане и при трите типа числа се изпълнява с инструкцията

SBC — изваждане от съдържанието на акумулатора на съдържанието на клетка от паметта или константа и заема.

Използуваните начини на адресиране са същите както при събирането. Получената разлика се запазва в акумулатора.

Характерно за тази инструкция е, че използува заем. Той е обратната стойност на флага за пренос (т. е. 1, ако флагът е изчистен, и 0, ако флагът е установен). Тъй като при изваждането на числа без знак заемът не участвува, задължително е флагът за пренос да се установи преди извършването на операцията.

В нормалния случай при работа с числа без знак по-малкото число се изважда от по-голямото.

```
1 SEC
2 LDA #$08
3 SBC #$06
4 BRK
```

Стойността $08 се намалява с $06 и получената разлика е $02. Ако обаче от по-малкото число се изважда по-голямото, полученият резултат е по-малък от най-малкото за този тип числа $00. При тези условия резултатът е равен на разликата между $100 и абсолютната стойност на разликата между двете числа. Сигнал за наличието на такъв случай е изчистването на флага за пренос след извършване на изваждането. Ако в горния пример се разменят местата на двете стойности, полученият резултат е $FE и 0 във флага за пренос.

```
1 SEC
2 LDA #$06
```

```
3 SBC #$08
4 BRK
```

4.2. Аритметични операции с числа със знак

Много често се налага да се работи както с положителни, така и с отрицателни стойности. Тогава се работи с числа със знак. Това са всички цели числа между −128 (10000000) и +127 (01111111), които се представят с осем бита, като най-старшият от тях е за означаване на знака и се нарича знаков бит. При положителните числа се използува нормалното двоично представяне и знаковият бит има стойност 0, докато отрицателните се представят чрез двоичното допълнение на противоположните им положителни числа, при което знаковият им бит е 1. Двоичното допълнение на кое да е число със знак се получава, като се инвертират всичките му битове, включително и знаковият, и към получения резултат се прибави единица. Двоичното допълнение променя знака на числото и се запазва абсолютната му стойност. Например, за да се представи −2 чрез двоичното допълнение на +2, взема се двоичното представяне на 2 (00000010), инвертират се всичките му битове (11111101) и се прибавя 1. Полученото число (11111110 или $FE) е двоичното допълнение на 2, т. е. −2. Използуването на двоичното допълнение позволява изваждането да се сведе до събиране на умаляемото с двоичното допълнение на умалителя.

4.2.1. Събиране на числа със знак

Числа със знак се събират по почти същия начин, както и числа без знак. Използува се същият оператор (ADC), същите начини на адресиране, преносът също предварително трябва да се нулира за получаването на верен резултат. Различно е обаче отчитането на преноса. При числата без знак той се получава, като се прехвърля един бит със стойност 1 от седмия бит на резултата и това е признак за наличието на пренос при преминаването на границите на допустимите за дадения тип числа. В числата със знак седмият бит е знаков бит и преносът от него не е сигнал за преминаване на допустимите граници на резултата. Ето защо тук се използува флагът за препълване. Той се установява, когато се излезе извън границите на числата със знак от −128 до +127, т. е. когато полученият резултат не може да се запише заедно със знака си в 8 бита.

Препълване при аритметични операции с числа със знак се получава, ако знаците на операндите при окончателното събиране (изваждането се свежда до събиране с двоичното допълнение на умалителя) са еднакви, а резултатът е с обратен знак.

Проверката на състоянието на флага за препълване дава възможност да се направят изводи за верността на получените резултати и при необходимост да се поправят.

Следните примери показват всички възможни случаи при събиране на числа със знак:

```
1 CLC
2 LDA #$03
3 ADC #$02
4 BRK
```

Събират се две положителни числа $03 и $02. Полученият резултат $05 е правилен, тъй като флагът за препълване е изчистен.

```
1 CLC
2 LDA #$7D
3 ADC #$04
4 BRK
```

Събират се две положителни числа $7D и $04. Установяването на флага за препълване показва, че получената сума е извън допустимите граници за този тип числа и нейната стойност $81 трябва да се поправи.

```
1 CLC
2 LDA #$FE
3 ADC #$FC
4 BRK
```

Събират се две отрицателни числа $FE (−2) и $FC (−4). Полученият резултат $FA (−6) е правилен, сигнал за което е изчистеният флаг за препълване.

```
1 CLC
2 LDA #$9C
3 ADC #$E0
4 BRK
```

Събират се две отрицателни числа $9C (−100) и $E0 (−32) и се получава $7C (+124). Установеният флаг за препълване обаче показва, че този резултат не е верен и се нуждае от допълнителни поправки. В случая се преминава допустимата долна граница на числата със знак.

```
1 CLC
2 LDA #$21
3 ADC #$E0
4 BRK
```

Събират се положително $21 (33) с отрицателно $E0 (−32) число. Изчистеният флаг за препълване показва правилността на получения резултат $01. При събирането на числа с противо-

полжни знаци, както и при изваждане на числа с еднакви знаци, резултатът е винаги в допустимите граници и никога не се получава препълване. Това може лесно да се провери, като се съберат максималните стойности с различни знаци.

4.2.2. Изваждане на числа със знак

Числа със знак се изваждат с инструкцията SBC, като флагът за пренос предварително трябва да се установи, за да се премахне заемът. Всъщност числото, което се изважда от съдържанието на акумулатора, се преобразува в неговото двоично допълнение (т. е. обръща се знакът му) и след това изваждането се изпълнява чрез събиране на умаляемото с двоичното допълнение на умалителя. След извършването на аритметичната операция установяването на флага за препълване показва, че полученият резултат е извън границите на числата със знак от −128 до 127, а състоянието на флага за пренос е без значение.

```
1       SEC       ; премахване на заема
2       LDA $FE   ; акумулаторът се зарежда със
                  съдържанието на адрес $FE
3       SBC $FF   ; от него се изважда съдържа-
                  нието на адрес $FF
4       BVS IZV   ; преход при установен флаг за
                  препълване
5       BRK       ; резултатът е правилен — пе-
                  чат на стойността му
6 IZV   ...       ; резултатът е извън граници-
                  те от −128 до +127 — налага
                  се поправка
```

4.3. Аритметични операции с двоично кодирани десетични числа

При двоично кодираните десетични числа всяка десетична цифра се кодира с четири бита, а в един байт се пакетират две десетични цифри. Например десетичното число 15 в обикновеното двоично (и съответно шестнайсетично) представяне има вида 0000 1111 ($0F), докато като двоично кодирано десетично число то е 0001 0101 ($15). При този тип числа микропроцесорът обработва само положителни числа в границите от 0 ($00) до 99 ($99).

За преминаване към режим на работа с двоично кодирани десетични числа се използува флагът за десетичен режим. Неговото установяване с инструкция SED е сигнал за микропроцесора да възприема всички следващи числови константи като двуцифрени двоично-десетични. Това продължава до първата срещната инструкция CLD, която изчиства флага за десетичен режим, и

следващите аритметични операции се изпълняват отново с двоични числа. Тъй като в самото начало на програмата състоянието на флага за десетичен режим не е известно, добре е с една от първите инструкции той да бъде изчистен.

4.3.1. Събиране на двоично кодирани десетични числа

Двоично кодираните числа се събират, като се използува инструкцията ADC. Флагът за пренос трябва да бъде предварително изчистен. Ако полученият след събирането резултат е по-голям от 99 ($99), флагът за пренос се установява, а в акумулатора остават само младшите осем бита на сумата.

```
1       SED         ; включване на десетичен режим
2       CLC         ; изчистване на преноса
3       LDA $FE     ; акумулаторът се зарежда със съ-
                      държанието на адрес $FE
4       ADC $FF     ; събира се със съдържанието на ад-
                      рес $FF
5       BCS GR      ; преход при установен флаг за пре-
                      нос
6       BRK         ; правилен резултат — печат на
                      стойността му
7 GR... ...         ; резултатът е по-голям от $99 —
                      налага се поправка
```

4.3.2. Изваждане на двоично кодирани десетични числа

За изваждане на двоично кодирани десетични числа се използува инструкция SBC при установен флаг за десетичен режим и установен флаг за пренос. Изчистването на флага за пренос след изпълнението на операцията показва, че получената разлика е по-малка от 0.

```
1       SED         ; включване на десетичен режим
2       SEC         ; изчистване на заема
3       LDA $FE     ; акумулаторът се зарежда със
                      съдържанието на адрес $FE
4       SBC $FF     ; от него се изважда съдържание-
                      то на адрес $FF
5       BCC MIN     ; преход при изчистен флаг за
                      пренос
6       BRK         ; правилен резултат — печат на
                      стойността му
7 MIN   ...         ; резултатът е по-малък от 0 —
                      налага се поправка
```

4.4. Аритметични операции с повишена точност

Аритметичните операции, които се извършват с 8-битови числа, се наричат аритметични операции с единична точност. Граничните стойности при тях зависят от типа на използваните числа — съответно от 0 до 255 при числата без знак, от −128 до +127 при числата със знак и от 0 до 99 при двоично кодираните десетични числа. Често се оказва, че тези стойности не удовлетворяват нуждите на дадена програма и се налага да се работи със значително по-големи или по-малки числа. В тези случаи едно число може да се представи като съдържание на поредица от два, три или повече байта. Например при числата без знак с два последователни байта могат да се представят стойности от 0 до 65 535, с три байта — от 0 до 16 777 215 и т. н. Такива числа се наричат многобайтови, а аритметичните операции с тях — аритметични операции с повишена точност (двойна, тройна и т. н.)

4.4.1. Събиране с повишена точност

Последователността от действия при събиране на многобайтови числа е еднаква за трите типа числа. Предварително се изчиства флагът за пренос и се определя мястото в паметта, където ще се записва резултатът. Последователно отдясно наляво се събират съответните първи, втори, трети и т. н. байтове на двете числа заедно с възникналите преноси, като при това флагът за пренос не се изчиства. Междинните суми се записват в първия, втория, третия и т. н. байтове на крайния резултат. Показател за получаването на резултат извън границите на числата, които могат да се представят с предварително определеното количество байтове, е установяването на флага за пренос след събирането на съответните най-старши байтове. В следния пример се събират две 3-байтови двоично кодирани десетични числа и се проверява дали полученият резултат не надхвърля максимално допустимата стойност (в този случай тя е 999 999):

```
1 A1    EQU $2003    ; първото число се разпо-
                       лага в адреси $2001,
2 A2    EQU $2002    ; $2002 и $2003
3 A3    EQU $2001
4 B1    EQU $2006    ; второто число се разпо-
                       лага в адреси $2004,
5 B2    EQU $2005    ; $2005 и $2006
6 B3    EQU $2004
```

```
 7 SUMA1   EQU $2009     ; резултатът се разпола-
                         ; га в адреси $2007,
 8 SUMA2   EQU $2008     $2008 и $2009
 9 SUMA3   EQU $2007
10         SED           ; включване на десетичен
                         режим
11         CLC           ;изчистване на преноса
                         преди събирането
12         LDA A1        ; в акумулатора се зарежда
                         първият байт на първото число
13         ADC B1        ; събира се с първия байт
                         на второто число
14         STA SUMA1     ; междинната сума се за-
                         писва в първия байт на
                         резултата
15         LDA A2        ; акумулаторът се зарежда
                         с втория байт на първото число
16         ADC B2        ; събира се с втория байт
                         на второто число и преноса
17         STA SUMA2     ; междинната сума се за-
                         писва във втория байт на
                         резултата
18         LDA A3        ; акумулаторът се зарежда
                         с третия байт на първото число
19         ADC B3        ; събира се с третия байт
                         на второто число
20         STA SUMA3     ; междинната сума се за-
                         писва в третия байт на
                         резултата
21         BCS POPR      ; преход при установен
                         флаг за пренос
22         BRK           ; резултатът е в грани-
                         ците от 0 до 999999
23 POPR    ...           ; резултатът е по-голям
                         от 999999 — налага се
                         поправка
```

4.4.2. Изваждане с повишена точност

Изваждането на многобайтови числа се извършва подобно на събирането им. Флагът за пренос трябва да бъде установен

предварително, за да се изчисти заемът. Байтовете на второто число последователно се изваждат от съответните байтове на първото число. Изчистването на флага за пренос при образуването на междинните суми показва наличието на заем от по-старшия байт. Ако след извършването на операцията полученият резултат е извън предварително определените допустими граници, флагът за пренос се изчиства. В следния пример се изваждат две 2-байтови числа без знак:

```
1  A1      EQU $2002    ; първото число се разполага в
2  A2      EQU $2001    ; адреси $2001 и $2002
3  B1      EQU $2004    ; второто число се разполага в
4  B2      EQU $2003    ; адреси $2003 и  $2004
5  RAZL1   EQU $2006    ; резултатът се разполага в
6  RAZL2   EQU $2005    ; адреси $2005 и $2006
7          SEC          ; изчистване на заема
8          LDA A1       ; акумулаторът се зарежда с първия байт на първото число
9          SBC B1       ; от него се изважда първият байт на второто число
10         STA RAZL1    ; междинната разлика се записва в първия байт на резултата
11         LDA A2       ;акумулаторът се зарежда с втория байт на първото число
12         SBC B2       ; от него се изважда вторият байт на второто число и заемът
13         SBC RAZL2    ; разликата се записва във втория байт на резултата
14         BCS POPR     ; преход при установен флаг за препълване
15         BRK          ; резултатът е в допустимите граници
16 POPR    ...          ; резултатът е извън допустимите граници — налага се поправка
```

5. ЛОГИЧЕСКИ ОПЕРАЦИИ

При програмиране на АСЕМБЛЕР програмистът има възможност да изпълнява операции не само с цели байтове, както в повечето езици за програмиране от високо ниво, но и с отделни битове или поредици от битове, наречени двоични низове. С тях могат да се извършват както логически операции, така и операциите изместване и ротация (циклично изместване).

Логически операции могат да се извършват с отделни двойки битове както от съдържанието на акумулатора, така и от съдържанието на клетка от паметта. За всяка логическа операция е в сила таблица на истинност. Тя показва резултатите за всички възможни комбинации от входни данни. Операторите за логически операции използуват непосредствено, пряко пълно, индексно адресиране, косвено адресиране с предварително или с последващо индексиране, адресиране в нулевата страница — пряко или с индексиране. При изпълнението на съответните инструкции се установява флагът за отрицателна стойност — когато седмият бит на резултата е 1, и флагът за нула — когато всички битове на резултата са 0.

5.1. Логическо умножение

За логическо умножение (конюнкция) се използува операторът

AND — логическо умножение на битовете от съдържанието на акумулатора със съответните битове на константа или на съдържанието на клетка от паметта.

Логическото умножение се означава със знака $\wedge$ и таблицата му на истинност има следния вид:

A	B	A∧B
1	1	1
1	0	0
0	1	0
0	0	0

Най-често тази логическа операция се използува за нулиране на отделни битове от съдържанието на акумулатора за извличане на стойностите на останалите битове. Подбира се такава стойност, която при логическо умножение със съдържанието на акумулатора да нулира само избраните предварително битове. Тази стойност се нарича маска, а логическото умножение — маскиране. Ако трябва да се нулират третият, четвъртият и петият бит от акумулатора, за маска се използува число, което съдържа нули в третия, четвъртия и петия бит и единици в останалите. При логическо умножение, ако единият от двата бита е 0, резултатът е също нула независимо от стойността на другия бит. Поради това след извършване на операцията резултатът, кой-

то остава в акумулатора, съдържа нули в третия, четвъртия и петия бит.

```
1 LDA #$FE   ; в акумулатора се зарежда $FE
             (11111110)
2 AND #$C7   ; логическо умножение с маската $C7
             (11000111)
3 BRK        ; полученият резултат $C6 (11000110)
             съдържа нули в третия, четвъртия и
             петия бит
```

Операторът AND се използува често за изолиране и проверка на определен бит от съдържанието на акумулатора. Избира се маска, която съдържа само нули и единица в бита, който ще се проверява. Ако този бит има стойност 0, след извършване на операцията акумулаторът съдържа нули във всичките си битове, което води до установяване на флага за нула. В обратния случай резултатът е число, различно от нула, и флагът за нула се изчиства. Следващите действия в програмата се определят в зависимост от състоянието на флага за нула.

```
1        LDA ADR     ; акумулаторът се зарежда
                     със съдържанието на даден
                     адрес
2        AND #$04    ; за проверка на втория бит
                     се използува маската
                     00000100 ($04)
3        BEQ NULA    ; преход при установен флаг
                     за нула
4 EDIN   ...         ; проверяваният бит е 1
         ...
10 NULA  ...         ; проверяваният бит е 0
```

При изпълнението на такава проверка началната стойност в акумулатора се губи, тъй като в него се запазва резултатът. В някои случаи това може да се окаже сериозен недостатък. За преодоляването му може да се използува операторът

BIT — сравняване на битовете от съдържанието на клетка от паметта със съответните битове от съдържанието на акумулатора.

Използува се пряко пълно адресиране или пряко адресиране в нулевата страница. При изпълнението на съответната му инструкция се извършва логическо умножение на съдържанието на акумулатора със съдържанието на посочената клетка от паметта. За разлика от инструкцията AND обаче полученият резултат не се запазва никъде, но оказва влияние върху флаговете, като седмият бит от него се зарежда във флага на отрицател-

на стойност, а шестият — във флага за препълване. Флагът за нула също се установява или изчиства в зависимост от получената след логическото умножение стойност.

Ако вместо оператора AND се използува операторът BIT с пряко адресиране в нулевата страница, горният пример може да изглежда така:

```
1        LDA #$04     ; в акумулатора се зарежда маската за проверка на втория бит
2        STA $FF      ; тази маска се записва в адрес от нулевата страница
3        LDA ADR      ; акумулаторът се зарежда със стойността, която ще се сравнява
4        BIT $FF      ; сравнява се съдържанието на клетка от нулевата страница на паметта със съдържанието на акумулатора
5        BEQ NULA     ; преход при установен флаг за нула
6 EDIN   ...          ; проверяваният бит има
         ...          стойност 1
         ...
10 NULA  ...          ; проверяваният бит има стойност 0
```

5.2. Логическо събиране

Операторът за логическо събиране (дизюнкция) е

ORA — логическо събиране на битовете от съдържанието на акумулатора със съответните битове на константа или съдържанието на клетка от паметта.

Тази операция се означава със знака $\vee$ и съответната ѝ таблица на истинност е:

A	B	A$\vee$B
1	1	1
1	0	1
0	1	1
0	0	0

Инструкцията за логическо събиране ORA се използува главно за установяване на отделни битове от съдържанието на акумулатора, като се запазва състоянието на останалите. От таблицата на истинност се вижда, че ако единият от събираните битове е 1, то и резултатът е 1 независимо от стойността на другия бит. Ето защо за избирателно установяване се използува константа, съ-

държаща 1 само в битовете, които трябва да се установят. Например:

```
1 LDA ADR     ; нека съдържанието на ADR да е
                $51 (01010001)
2 ORA #$FO    ; за установяване на старшите четири
                бита се използува логическо събиране
                с маската $F0 (11110000)
3 BRK         ; полученият резултат $F1 (11110001)
                съдържа 1 в старшите четири бита
```

Логическото събиране може да се използува и за да се провери, дали няколко байта съдържат само нули. За тази цел е достатъчно първият от тях да се зареди в акумулатора и след това последователно да се събере с останалите. Ако след последната операция флагът за нула е изчистен, то поне един от тези байтове съдържа ненулева стойност.

5.3. Логическото събиране по модул 2 (изключващо ИЛИ)

Изключващото ИЛИ се изпълнява с оператор

EOR — събиране по модул 2 на битовете от съдържанието на акумулатора със съответните битове на константа или от съдържанието на клетка от паметта.

Операцията се означава със знака ⊕, а таблицата за истинност има следния вид:

A	B	A ⊕B
1	1	0
1	0	1
0	1	1
0	0	0

Инструкцията EOR, съчетана с подходящо избрана константа, инвертира отделни битове от съдържанието на акумулатора. Например за да се инвертират младшите три бита (т. е. всеки от тях да се установи, ако има стойност 0, или да се нулира, ако има стойност 1), съответната константа трябва да бъде 00000111. Таблицата на истинност показва, че ако единият бит е 1, резултатът е равен на обратната стойност на втория.

```
1 LDA ADR     ; нека съдържанието на ADR е $F1
                (11110001)
2 EOR #$07    ; събиране по модул 2 с константата
                $07 (00000111)
3 BRK         ; в резултата $F6 (C11110110) млад-
                шите три бита са инвертирани спря-
                мо първоначалните им стойности
```

При константа $FF (11111111) се инвертират всички битове на числото, заредено в акумулатора. Ако началното число е било

със знак и след това се прибави 1, се получава точно двоичното му допълнение, т. е. противоположното му число. Една програма за умножение с числото −1 може да изглежда по следния начин:

```
1 CLC          ; изчистване на преноса
2 LDA ADR      ; в акумулатора се зарежда съдържанието на клетка с адрес ADR
3 EOR #$FF     ; инвертират се всички битове
4 ADC #$01     ; прибавя се единица
5 STA ADR      ; полученото двоично допълнение се зарежда обратно в същия адрес
```

6. РАБОТА С ДВОИЧНИ НИЗОВЕ

С операторите за изместване и ротация се променя съдържанието на акумулатора или клетка от паметта, като се променя местоположението на битовете от него. Използува се пряко пълно адресиране, пряко пълно адресиране с индексиране по X, пряко адресиране в нулевата страница, пряко адресиране в нулевата страница с индексиране по X, а когато начинът на адресиране е с подразбиране на акумулатора, операциите се извършват със съдържанието му.

6.1. Оператори за изместване

Микропроцесорът 6502 изпълнява две инструкции за изместване. Първата от тях е

ASL — изместване с един разред наляво на всички битове от съдържанието на акумулатора или клетка от паметта.

При изпълнението ѝ стойността на всеки бит се прехвърля като стойност на следващия вляво бит, нулевият бит се изчиства, а стойността на седмия бит се прехвърля във флага за пренос. Тази операция се нарича още аритметично изместване (за разлика от изместването надясно, което се нарича логическо), тъй като едно изместване наляво на битовете на едно число без знак е равностойно на умножението му с две, две измествания — на умножение с четири, три — на умножение с осем и т. н.

```
1 LDA #$02 ; в акумулатора се зарежда стойност $02 (00000010)
2 ASL      ; изместване с един разред наляво
3 BRK      ; получава се $04 (00000100)
```

По този начин може да се извърши умножение с кое да е чис-

ло, тъй като всяко число може да се представи като сума от степените на 2. Необходимо е само междинните суми да се запазват временно в паметта и накрая да се съберат за получаване на крайния резултат. В следващия пример е показано умножение на число по 10 (2^3+2^1):

```
1        LDA ADR     ; в акумулатора се зарежда
                       съдържанието на
                       клетка с адрес ADR
2        CLC         ; изчистване на преноса
3        ASL         ; умножение с 2
4        BCS POPR    ; проверка за препълване
5        STA $FF     ; резултатът временно
                       се запазва в клетка от
                       паметта
6        ASL         ; умножение с 4
7        BCS POPR    ; проверка за препълване
8        ASL         ; умножение с 8
9        BCS POPR    ; проверка за препълване
10       CLC         ; изчистване на преноса
11       ADC         ; двете суми се събират
12       STA ADR     ; резултатът се записва
                       обратно в същия адрес
13       BRK
14       POPR        ; при едно от умноженията
         ...           резултатът е по-голям
                       от $FF (255) — налага
                       се поправка
```

Когато се работи с голям обем еднотипни данни, налага се да се пести памет. Едно от възможните решения е използването на т. нар. пакетирани данни. Например ако една програма обработва информация за единните граждански номера, за представянето на всеки един от тях са необходими 10 байта — по един за всяка от съставящите ги цифри. ASCII-кодовете на цифрите от 0 до 9 са от $30 до $39. Старшите им 4 бита са еднакви за всички и следователно могат да не се записват. От всяка цифра остават само младшите 4 бита. Понеже в един байт могат да се вместят по две цифри, необходимата памет за запис на единните граждански номера се намалява наполовина — от 10 на 5 байта. Този процес на модифициране на данните се нарича пакетиране и се извършва с помощта на инструкцията ASL. Ето как се пакетират в един байт две цифри, представени в кода ASCII:

```
1 LDA ADR1  ; в акумулатора се зарежда първата
              цифра
2 ASL       ; с 4 измествания наляво младшите 4
3 ASL       ; бита се прехвърлят на мястото на
```

```
4 ASL          ; старшите — вдясно стават нули
5 ASL
6 STA ST       ; резултатът се записва в клетка от
                 паметта
7 LDA ADR2     ; в акумулатора се зарежда втората
                 цифра
8 AND #$0F     ; старшите 4 бита се изчистват чрез
                 логическо умножение с маска 00001111
                 ($0F)
9 ORA ST       ; логическо събиране с битовете на
                 първата цифра
10 STA REZ     ; двете пакетирани цифри се записват
                 в един байт
11 BRK
```

Ако при изпълнение на тази програма се пакетират например 9 (ASCII-код $39) и 2 (ASCII-код $32), резултатът ще бъде $92. При четене на така обработените данни те трябва предварително да бъдат разделени, като се използува оператор

LSR — изместване с един разред надясно на всички битове от съдържанието на акумулатора или клетка от паметта.

Инструкцията LSR действува подобно на ASL, но в обратна посока — надясно. Изчиства се седмият бит, а във флага за пренос се прехвърля стойността на нулевия бит. С нея разделянето на две пакетирани по описания начин цифри се извършва по следния начин:

```
1 LDA REZ      ; в акумулатора се зарежда съдържа-
                 нието на клетка REZ, в която са запи-
                 сани двете пакетирани цифри
2 LSR          ; с 4 измествания надясно старшите
3 LSR          ; 4 бита се прехвърлят на мястото на
4 LSR          ; младшите — вляво остават нули
5 LSR
6 ORA $#30     ; добавя се $30 чрез логическо събиране
7 STA ADR1     ; първата цифра е получена в първона-
                 чалния си вид и се зарежда в отделна
                 клетка
8 LDA REZ      ; отново в акумулатора се зареждат
                 двете пакетирани цифри
9 AND #$0F     ; старшите 4 бита се изчистват чрез
                 логическо умножение с маска 00001111
                 ($0F)
10 ORA #$30    ; добавя се $30 чрез логическо събиране
11 STA ADR1    ; втората цифра е получена в първона-
```

чалния си вид и се зарежда в отделна клетка

```
12 BRK
```

Друго практическо приложение на инструкцията LSR е за проверка по четност. Ако след едно изместване надясно на битовете на някакво число флагът за пренос се установи, това показва, че числото е нечетно (2*N+1), тъй като съдържа 1 в нулевия си бит. В обратния случай изчистеният флаг за пренос показва наличието на четно число (2*N).

```
1       LDA CS      ; акумулаторът се зарежда с чис-
                      лото, което ще се проверява
2       CLC         ; изчистване на флага за пренос
3       LSR         ; изместване надясно
4       BCS ODD
5       BRK         ; четно число — отпечатване
6 ODD   ...         ; нечетно число — други дейст-
                      вия
```

6.2. Оператори за ротация

Инструкциите, съответни на операторите за ротация, се изпълняват по същия начин, както и инструкциите за изместване, с тази разлика, че крайният бит не се изчиства, а приема стойността на флага за пренос преди операцията. Например при ротация надясно стойността на преноса от предишната операция се прехвърля в седмия бит, стойността на всеки бит се прехвърля в бита вдясно, а стойността на нулевия бит се прехвърля във флага за пренос. При изчистен флаг за пренос резултатът от ротацията и изместването в същата посока е съвсем еднакъв. Съществуват два оператора за ротация:

ROL — ротация наляво на битовете от съдържанието на акумулатора или клетка от паметта;

ROR — ротация надясно на битовете от съдържанието на акумулатора или клетка от паметта.

Когато се налага да се изместват числа с повишена точност, т. е. да се изместват битовете на многобайтовото число, се използуват последователни ротации на отделните байтове, като се отчитат възникналите преноси. Следната програма изпълнява изместване наляво на двубайтово число:

```
1 CLC       ; изчистване на преноса
2 LDA ML    ; в акумулатора се зарежда младшият
              байт на числото
3 ROL       ; ротация наляво
4 STA ML    ; изместеният байт се връща в същия
              адрес
```

```
5 LDA ST    ; в акумулатора се зарежда старшият
              байт на числото
6 ROL       ; отново ротация наляво — участвува
              и преносът от предишната ротация
7 STA ST    ; изместеният байт се връща в същия
              адрес
```

Докато тук последователността от ротации е от младшия към старшите байтове, при изместване надясно с повишена точност тя е обратно — от старшите към младшите байтове. В примера съдържанието на отделните клетки се зарежда в акумулатора и след ротацията резултатът се връща на същия адрес. Програмата може да се съкрати значително, ако се използуват инструкции за ротация на съдържанието на клетка от паметта. Така изместването с двойна точност може да се изпълни така:

наляво
```
1 CLC
2 ROL ML
3 ROL ST
```

надясно
```
1 CLC
2 ROR ST
3 ROR ML
```

В предната точка е показана възможността за проверка на състоянието на четния (нулевия) бит. По същия начин може да се проверява кой да е бит. Например за проверка на втория бит се изпълняват три измествания надясно и се проверява флагът за пренос. Ако той е установен, третият бит е имал стойност 1, а ако е изчистен — стойността е била 0.

```
1       LSR ADR
2       LSR ADR
3       LSR ADR
4       BCS ED
5       BRK         ; третият бит от ADR е бил 0
6 ED    ...         ; третият бит от ADR е бил 1
```

При тази проверка обаче числото, чиито битове се проверяват, се променя след изместванията. По-добре е за проверката да се използуват определен брой ротации в едната посока, а след това числото да се възстанови в началния си вид със същия брой ротации в обратната посока. За горния пример тази проверка изглежда така:

```
1       ROR ADR
2       ROR ADR
3       ROR ADR
4       BCS ED
5       ...         ; третият бит от ADR е 0
6       JMP VST
7 ED    ...         ; третият бит от ADR е 1
8 VST ROL ADR
```

```
9       ROL ADR
10      ROL ADR         ; възстановяване на числото
```

7. РАБОТА СЪС СТЕКА И ИЗПОЛЗУВАНЕ НА ПОДПРОГРАМИ

Основен метод при създаване на програмни продукти е т. нар. програмиране отгоре надолу. Неговата същност се състои накратко в това, че основната задача се разделя на множество прости подзадачи, които се разделят на още по-прости и т. н. При достигане на последната степен на детайлизация тези подзадачи се програмират като относително самостоятелни елементи, наречени *подпрограми*, които накрая се свързват в единна система — програмата, която решава основната задача. Такъв подход позволява проектирането, създаването, проверката и поправката на отделните подпрограми да се извършват независимо една от друга дори и от различни програмисти. Освен това веднъж създадената подпрограма може да бъде използувана и в други програми, ако се налага да се изпълняват същите задачи. Добре структурираната програма на АСЕМБЛЕР се състои от множество подпрограми и една главна програма, която съдържа в по-голямата си част обръщения към отделните подпрограми и осъществява връзката между тях.

Всички подпрограми използуват по различни начини адресното пространство от първата страница на паметта, наречено *стек*. Ето защо познаването на особеностите при работа с него е необходимо условие за правилното програмиране и използуване на подпрограми.

7.1. Работа със стека

Стекът включва 256 клетки от паметта, разположени между адресите $100 и $1FF включително. Съдържанието на тези клетки се променя под контрола на специален регистър — *указател на стека*. Докато при запис или четене на данни за всички останали области от паметта е необходимо да се зададе адресът на използуваната клетка, тук е достатъчно да се определи само областта, т. е. стекът. Това става с операторите

PHA — записване на съдържанието на акумулатора в стека;
PLA — зареждане на акумулатора със стойност, извлечена от стека;
PHP — записване на съдържанието на регистъра за състоянието в стека;

PLP — зареждане на регистъра за състоянието със стойност, извлечена от стека.

В самото начало на програмата в указателя на стека се зарежда началната стойност $FF, така че той да сочи началото на стека:

```
1 LDX #$FF
2 TXS
```

След всяко следващо записване на данни в стека тази стойност се намалява автоматично, а преди всяко четене се увеличава с единица. Така адресът на клетката, с която се работи във всеки момент, се определя от съдържанието на указателя на стека (разбира се увеличено с $100). Тъй като указателят на стека е 8-битов, при опит в стека да се запишат повече данни, отколкото са свободните адреси, или да се прочетат повече данни, отколкото са записани, той се „превърта". Например при вече заети адреси в стека от $1FF до $102 указателят ще има стойност $01. Ако се запишат още три данни, последната от тях се намира в адрес в $1FE.

Включените в следния пример операции четене и записване на данни в стека нямат особена практическа стойност, но онагледяват добре начина на работа с тази област на паметта, както и промените, които стават със специалния регистър — указател на стека:

```
1 LDX #$FF    ; указателят се установява в началото
2 TXS         ; на стека и показва адрес $1FF
3 PHP         ; съдържанието на регистъра за състоянието на процесора се зарежда в адрес $FF от стека — указателят се намалява с 1 и показва адрес $FE
4 LDA #$01    ; в акумулатора се зарежда числото 1
5 PHA         ; съдържанието на акумулатора се записва в адрес $FE от стека — указателят се намалява с 1 и показва адрес $1FD
6 PLP         ; указателят се увеличава с 1 и показва адрес $1FE — там се намира последната записана в стека стойност и тя се зарежда в регистъра за състоянието на процесора
7 LDA ADR1    ; акумулаторът се зарежда от адрес ADR1
8 PHA         ; съдържанието на акумулатора се записва в адрес $1FE от стека — указателят показва следващия свободен адрес $1FD
```

```
 9 PLA        ; указателят се увеличава с 1 и показва
                адрес от стека $1FE, чието съдържа-
                ние се записва в акумулатора
10 STA ADR2   ; съдържанието на акумулатора се за-
                писва в ADR2
11 PLA        ; указателят се увеличава с 1 и показва
                адрес от стека $1FF, чието съдържа-
                ние се записва в акумулатора
12 PLA        ; указателят се увеличава с 1 и поради
                „превъртането" си показва не адрес
                $200, а последния адрес в стека $00,
                откъдето се зарежда акумулаторът
13 BRK        ; отпечатване на съдържанието на ре-
                гистрите
```

Този начин на организация на данните се нарича LIFO (последен влязъл, пръв излязъл). При него не е от значение разположението на данните в паметта, а само редът на тяхното използуване, което е голямо предимство при използуването на подпрограми.

7.2. Използуване на подпрограми

7.2.1. Организиране на подпрограми

Подпрограмите се организират като поредици от оператори и съответните им инструкции започват от даден адрес и се разполагат обикновено след главната програма. За предаване на управлението към подпрограма (или както още се казва, за обръщение към нея) се използува операторът.

JSR — преход към подпрограма.

Адресирането е винаги пряко пълно. При изпълнение на такъв преход двата байта от текущата стойност на програмния брояч се записват в стека и указателят му автоматично се намалява с две единици. След това в програмния брояч се зарежда адресът от инструкцията JSR и изпълнението на подпрограмата продължава от този адрес. След изпълнението на подпрограмата управлението се предава обратно на главната програма след точката на обръщението към подпрограмата. Това става с оператора

RTS — връщане от подпрограма.

При изпълнението на съответната му инструкция указателят последователно се увеличава с две единици и от стека се извличат два байта. Адресът, образуван от тях, се зарежда в

програмния брояч, увеличава се с 1, така че да показва следващата след JSR инструкция, и изпълнението на програмата продължава оттам. В примера е организирана подпрограма, която включва програмните редове от 8 до 12 и нулира акумулатора и индексните регистри.

```
 1        LDX #$FF   ; указателят се установява
 2        TXS        ; в началото на стека
 3        LDA ADR1   ; акумулаторът се зарежда
                     от клетка с адрес ADR1
 4        LDX ADR2   ; регистърът X се зарежда
                     от клетка с адрес ADR2
 5        LDY ADR3   ; регистърът Y се зарежда
                     от клетка с адрес ADR3
 6        JSR PODPR  ; преход към подпрограмата
 7        BRK        ; печат на съдържанието на
                     регистрите
 8 PODPR  LDA #$00   ; нулиране на акумулатора
 9        LDX #$00   ; нулиране на регистъра X
10        LDY #$00   ; нулиране на регистъра Y
11        RTS        ; връщане от подпрограмата
```

Използването на стека като място за запазване на адресите за връщане позволява сравнително лесно да се организират вложени една в друга подпрограми. При това всяка инструкция JSR въвежда, а всяка RTS извежда един адрес от стека.

```
1          LDX #$FF
2          TXS
3          JSR PODPR1   ; преход към подпрограма
                        1 — в стека се записва
                        първият адрес за връща-
                        не
4          BRK
5 PODPR1   JSR PODPR2   ; преход към подпрограма
                        2 — в стека се записва
                        втори адрес за връщане
6          RTS          ; от стека се извлича ад-
                        ресът за връщане от
                        подпрограма 1
7 PODPR2   NOP
8          RTS          ; от стека се извлича ад-
                        ресът за връщане от
                        подпрограма 2
```

7.2.2. Предаване на данни между подпрограмите

Всяка подпрограма използува един или повече параметри. Това са променливи със стойности в точно определени клетки от паметта, чрез които се предават данни към и от подпрограми-

те. Съществуват два вида параметри — входни, чрез които данните постъпват за обработка в подпрограмата, и изходни, чрез които резултатите от работата на подпрограмата се използуват след точката на обръщението към подпрограмата. Данните се предават по няколко начина, като всеки от тях има своите предимства и недостатъци.

За предаване на данни могат да се използуват регистрите. При обръщение към подпрограма в тях се записват стойностите на параметрите, извършва се преход към подпрограмата, която от своя страна прочита стойностите от регистрите и изпълнява необходимите действия с тях. Резултатите се записват отново в регистрите и управлението се предава непосредствено след точката на обръщението към подпрограмата. Следващата подпрограма събира две числа, заредени съответно в регистрите X и Y, и изпраща резултата в акумулатора:

```
1       LDX ADR1    ; първото число се зарежда в
                      регистъра X
2       LDY ADR2    ; второто число се зарежда в
                      регистъра Y
3       JSR SUM     ; преход към подпрограмата за
                      събиране
4       BRK         ; печат на сумата, която се на-
                      мира в акумулатора
5 SUM   CLC         ; подпрограма за събиране
6       TXA
7       STY SM
8       ADC SM
9       RTS         ; сумата е в акумулатора —
                      връщане от подпрограмата
```

Този начин за предаване на данните има съществен недостатък — тъй като броят на достъпните регистри е ограничен — (акумулатор, регистър X и регистър Y), може да се работи най-много с три параметъра. Въпреки това много подпрограми от системния монитор и интерпретатора са организирани именно така и ако са известни адресът за обръщението и входните им параметри, те могат да бъдат използувани във всяка потребителска програма. Например, за да се начертае точка в режим на графика с малка разделителна способност, може да се използува подпрограмата с начален адрес $F800, като входните параметри хоризонтална и вертикална позиция се предават съответно чрез акумулатора и регистъра Y. Подробното изучаване на подпрограмите, включени в системния монитор и интерпретатора, са отличен урок по програмиране на АСЕМБЛЕР и програмистът има възможност да използува готови подпрограми. Информация за тези подпрограми може да се намери в почти всяко ръководство за работа с персоналния микрокомпютър Правец-82.

Когато се работи с повече параметри или регистрите се използуват от подпрограмите за други действия, данните могат да се предават чрез променливи, разположени в някоя (най-често нулевата) страница от паметта. В такъв случай се използуват два вида променливи — общи, използувани от главната и една или повече от подпрограмите, и частни (локални), използувани само от една подпрограма. Наличието на повече общи променливи в една програма спестява памет, а включването на повече частни променливи отстранява възможността за грешки от неправилно използуване на данните. Разумен компромис между тези две крайности е за най-важните параметри в програмата да се използуват общи, а за останалите — частни променливи. Точното определяне на това съотношение зависи от целите и организацията на конкретната програма. При този начин за предаване на данните предният пример би могъл да изглежда така:

```
1  X1   EQU $FD     ; запазване на място в нулевата
                     страница за променливата X1
2  X2   EQU $FE     ; запазване на място в нулевата
                     страница за променливата X2
3  REZ  EQU $FF     ; запазване на място в нулевата
                     страница за променливата REZ
4       LDA ADR1    ; първото число се зарежда в аку-
                     мулатора
5       STA X1      ; и оттам се записва в X1
6       LDA ADR2    ; второто число се зарежда в аку-
                     мулатора
7       STA X2      ; и оттам се записва в X2
8       JSR SUM     ; преход към подпрограмата за
                     събиране
9       BRK         ; печат на резултата, който се
                     намира в REZ
10 SUM  CLC         ; подпрограма за събиране
11      LDA X1
12      ADC X2
13      STA REZ
14      RTS         ; сумата е в акумулатора — връ-
                     щане от подпрограмата
```

Тук променливите ADR1 и ADR2 са частни, тъй като се използуват само от главната програма, а X1, X2 и REZ — общи.

Като междинна памет за предаване на данни може да се използува и стекът, особено когато се работи и с регистъра за състоянието. Например, ако аритметичните действия в главната програма се извършват с двоични числа, а в подпрограмата — с двоично кодирани десетични числа, състоянието на флаговете преди прехода трябва да бъде запазено в стека, за да може да бъде възстановено след връщането от подпрограмата.

```
1         CLD       ; установяване на режим на работа с двоични числа
2         LDX X1    ; първото събираемо се зарежда в регистъра X
3         LDY X2    ; второто събираемо се зарежда в регистъра Y
4         PHP       ; съдържанието на регистъра за състоянието се запазва в стека
5         JSR SUM   ; преход към подпрограма за събиране
6         PLP       ; възстановява се състоянието на регистъра за състоянието от преди прехода, т. е. установява се отново режимът на работа с двоични числа
7         JMP НК    ; програмата продължава от НК
8 SUM     CLC       ; изчистване на преноса
9         SED       ; установява се режим на работа с двоично кодирани десетични числа
10        TYA
11        STA S1
12        TXA
13        ADC S1
14        RTS       ; сумата е в акумулатора — връщане от подпрограмата
```

Тук от особено значение е редът на записването на данни в стека да бъде обратен на реда на извличането им. Нарушаването на това изискване води до непредвидими последици. В горния пример в стека се зареждат последователно съдържанието на регистъра за състоянието — програмно при PHP, и двата байта на адреса за връщане — автоматично при JSR. Ако след това първо се изпълни PLP, ще се наруши редът на данните и при връщане от подпрограмата изпълнението ще продължи от напълно погрешен адрес, образуван от младшия байт на истинския адрес за връщане и съдържанието на регистъра за състоянието преди прехода. В някои случаи нарушаването на реда се използува съзнателно от програмистите за организиране например на подпрограми за изпълнение на рекурсивни функции, на подпрограми с множество адреси за връщане и т. н. Това обаче са по-сложни техники на програмиране и не са обект на разглеждане в тази книга. В общия случай състоянието на стека преди прехода към дадена подпрограма трябва да бъде възстановено след връщането от нея. С особена сила това важи за вложените една в друга подпрограми, при които в стека се записват множество адреси за връщане.

7.3. Използуване на отворени подпрограми (макроси)

Разгледаните дотук подпрограми са от т. нар. *затворен* тип. Те се появяват само веднъж в програмата, но се използуват многократно чрез обръщения от различни точки. За подпрограмите от този тип е задължително наличието на точка за предаване на управлението обратно на главната програма в съответствие с условията, определени при обръщението към подпрограмата.

Съществуват и втори тип подпрограми. Те се наричат *отворени* и се появяват на повече от едно място в главната програма. В професионалния жаргон на програмистите те са известни като *макроси.* Асемблер-редакторите, които позволяват работа с макроси, се наричат макроасемблери и по своите възможности се доближават до езиците за програмиране от високо ниво. Такъв асемблер-редактор е MERLYN.

Макросът е последователност от асемблерски оператори, съставящи отворена подпрограма, на която с асемблерската директива MAC е присвоено символично име. В следния пример символичното име SAVE се присвоява на поредицата от оператори за записване в стека на съдържанието на акумулатора и двата индексни регистъра.

```
1 SAVE   MAC
2        PHA
3        TXA
4        PHA
5        TYA
6        PHA
7        EOM    или      7 <<<
```

Наборът от оператори с общо символично име се нарича тяло на макроса или макроопределение. С директивата EOM или <<< се означава краят на макроопределението. Дефинирането на макроса се извършва в началото на първичната асемблерска програма. Преди макроопределението се използува директива DO 0, а след него FIN, за да се изключи то от обектния код. Понататък в първичната асемблерска програма така дефинираното символично име се използува като обикновен оператор с подразбиращо се адресиране. Това става с директивата PMC или >>>. Например след горното определение навсякъде в първичната програма, където е необходимо записване на съдържанието на посочените три регистъра в стека, трябва да се включи програмен ред

PMC SAVE или >>> SAVE.

Преди асемблирането в първичната асемблерска програма всеки

такъв оператор се заменя с поредицата от оператори, които образуват тялото на макроопределението, и в получената след това обектна програма се включват съответните им инструкции. Използуването на отворени подпрограми е оправдано при неголям брой на съставящите ги оператори, тъй като в обратния случай се получават много дълги обектни програми и тогава е по-добре да се използуват затворени подпрограми.

Асемблер-редакторът MERLYN позволява и организирането на вложени макроси, т. е. при дефинирането на една отворена подпрограма да се използуват други макроси. Единственото условие при това е броят на нивата на влагане да не надхвърля петнайсет.

Един от недостатъците на затворените подпрограми е, че предаването на параметри при преход към и от тях трябва да се осъществява от програмиста и да се следи непрекъснато от него. Основното предимство на отворените подпрограми е възможността за използуване на аргументи за предаване на параметрите. За целта е необходимо в макроопределението да се включат една или повече променливи като формални аргументи. За такива могат да се използуват както системните променливи от]1 до]8, така и променливи с произволно избрани от програмиста имена. Например отворена подпрограма за размяна на съдържанието на кои да е две клетки от паметта може да се дефинира така:

```
1 SWAP   MAC
2        LDX ]1
3        LDY ]2
4        STY ]1
5        STX ]2
6        EOM
```

В това определение променливите]1 и]2 са формални аргументи. Те се заменят с фактическите аргументи, които се задават при всяко обръщение към макроса. Например при обръщение към макроса SWAP от горния пример, осъществено по следния начин:

```
ADR1   EQU $FF
ADR2   EQU $FE
.
.
.
PMC    SWAP. ADR1; ADR2
```

стойностите на променливите ADR1 и ADR2 са фактически аргументи. Те се присвояват на съответните по ред формални аргументи, т. е. на]1 се присвоява стойността на ADR1 и на]2

се присвоява стойността на ADR2. След асемблирането на тази част от първичната асемблерска програма съответствуват следните инструкции от обектната програма:

```
LDX $FF
LDY $FE
STY $FF
STX $FF
```

При такава организация става възможно чрез използуването на различни фактически аргументи една и съща отворена подпрограма да се използува за извършване на еднакви действия със съдържанието на различни клетки от паметта. Ето няколко такива примера:

```
1         ORG $300
2         DO 0
3 POKE    MAC              ; дефиниране на отворена
4         LDA #]1          ; подпрограма за записване
5         STA ]2           ; на константата ]1 в
6         EOM              ; клетка с адрес ]2
7         FIN
8         PMC POKE.0;      ; макросът се използува за
          $FF              записване на 0 в клетка с ад-
                           рес $FF
 9 >>>  POKE. „A"; $FE     ; макросът се използува за
                           записване на ASCII-кода на
                           знака A в клетка с адрес
                           $FR
10 KL     EQU $FD
11      PMC POKE.$10; KL   ; макросът се използува за
                           записване на стойност $10
                           в клетката с адрес KL!
```

7.4. Външни прекъсвания и подпрограми за обработка на прекъсванията

За да се осигури съгласуваност на изпълнението на програмата с появата на външни събития, при микропроцесора 6502 съществуват два входа за прекъсване — IRQ и NMI. При подаване на сигнал през тях се нарушава нормалният ход на изпълнение на инструкциите, т. е. появява се външно прекъсване.

Първият вид външни прекъсвания са т. нар. маскируеми прекъсвания. Те се появяват при наличие на сигнал на входа IRQ и оформят заявка за прекъсване. Тази заявка се изпълнява в зависимост от състоянието на флага за прекъсване. Ако той е установен, прекъсването се забранява и изпълнението на програ-

мата следва нормалния си ход. Ако обаче той е изчистен, заявката се изпълнява, като се преминава към подпрограмата за обработка на прекъсвания, чийто начален адрес се прочита от адреси $FFFE и $FFFF. При този преход в стека се записват съдържанието на регистъра за състоянието и съдържанието на програмния брояч в момента на прекъсването. Подпрограмата за обработка на прекъсванията включва поредица от действия, които се изпълняват само при появата на външното събитие, предизвикало прекъсването, и завършва с оператора

RTI — връщане от подпрограма за обработка на прекъсванията.

При изпълнението на тази инструкция регистърът за състоянието и програмният брояч възстановяват съдържанието си от момента на прекъсването, като се зареждат със стойностите, запазени в стека. По този начин управлението на програмата се връща в точката на настъпване на прекъсването и изпълнението продължава със следващата инструкция.

Вторият вид външни прекъсвания са немаскируемите прекъсвания. При наличие на сигнал на входа NMI от адреси $FFFA и $FFFB се прочита началният адрес на подпрограмата за обработка на прекъсванията и независимо от състоянието на флага за прекъсване се осъществява преход към нея. Немаскируемите прекъсвания се използуват, когато очакваните външни събития са от такъв характер, че незабавно трябва да се прекрати нормалният ред на изпълнение на програмата.

Външно прекъсване може да се осъществи и чрез входа за изчистване RST. Той се използува за установяване на микропроцесора в начално положение. При наличие на сигнал на този вход се прочита съдържанието на адреси $FFFC и $FFFD и се осъществява преход към адреса, който се съхранява в тях. Обикновено това е $FA62, откъдето започва машинната подпрограма за изчистване. Тя изпълнява някои системни действия, като установяване на екрана в текстов режим, прекъсване на всички входно-изходни действия и други, и извършва събиране по модул 2 със съдържанието на адрес $3F3 и константата $A5 (165). Ако резултатът е еднакъв със съдържанието на адрес $3F4, стартира се дефинираната от програмиста подпрограма, чийто начален адрес се намира на адреси $3F2 и $3F3. В противен случай се търси и зарежда ДОС, като че ли компютърът е току-що включен.

Сигнал на входа RST се появява при първоначално включване на компютъра и при натискане на клавиша RST. В редица случаи представлява интерес възможността за промяна на подпрограмата за обработка на прекъсване за изчистване. Първият възможен начин е промяната на съдържанието на адреси $FFFC и $FFFD. Това е възможно само при използуване на DRAM (езикова

платка), тъй като в противен случай това са адреси от системния монитор.

При втория начин в адресите $3F2 и $3F3 трябва да се зареди адресът на подпрограмата, към която трябва да се осъществи преход при натискане на клавиша RST. След това трябва да се съберат по модул две съдържанието на адрес $3F3 и константата $A5 и резултатът да се запази в $3F4. Това може да стане със следните инструкции:

```
1 LDA ADRM      ; определяне на младшия
2 STA $3F2
3 LDA ADRS      ;и старшия разред на началния адрес
4 STA $3F3      на потребителската подпрограма
5 EOR #$A5
7 STA $3F4
```

След включването на компютъра (студен старт) в адреси $3F2 и $3F3 се зарежда адресът $E000, който е насочен към дисково устройство 1, а след натискане на RST (топъл старт) — адресът $E003, който е насочен към интерпретатора на БЕЙСИК.

8. ОРГАНИЗИРАНЕ НА МАСИВИ И ИНДЕКСИРАНЕ НА ЕЛЕМЕНТИТЕ ИМ

8.1. Организиране на масиви

8.1.1. Запазване на памет за масиви

Най-общо масивът е множество от еднакви по тип и дължина данни, записани в последователно разположени клетки от паметта. Наборът от инструкции за микропроцесора 6502 не съдържа инструкции за организация и работа с масиви за разлика от почти всички езици за програмиране от високо ниво. В повечето от асемблер-редакторите обаче са включени директиви, с които до известна степен се избягва това неудобство.

В тази книга са разгледани само едномерни масиви, които се управляват с един индекс. За организирането на такъв масив е необходимо най-напред да се определи размерът му — броят на елементите в него и еднаквата им дължина. След това се изчислява необходимият обем памет за неговото разполагане. Например, ако дадена програма използува масив от десет числа, всяко с дължина два байта, за него са необходими 20 байта от паметта. Има различни начини за запазване на памет за масиви. При асемблер-редакторите без директиви това може да стане по следния начин:

```
1          JMP NA
2 MASIV    NOP
3          NOP
4          NOP
5          NOP
6          NOP
7          NOP
8 NA       ...
```

В този пример с фиктивни инструкции (в случая са използувани NOP) се запазва памет за масив от 6 байта. По-нататък в програмата в мястото, заето от програмните редове от 2 до 7, се записват числата, образуващи масива. Този начин на запазване на памет за масиви е доста неудобен, особено при по-големи масиви. Ето защо в повечето асемблер-редактори се използува директивата DFS. В адресното ѝ поле се записва броят на байтовете, които се отделят за масива. При изпълнението ѝ програмният брояч се увеличава автоматично, така че да прескочи областта за данни. Ако се използува тази директива, предният пример изглежда така:

```
1 MASIV    DFS 6
2 NA       ...
```

Във всички случаи програмистът трябва да следи масивът да е така разположен в паметта, че данните в него да не бъдат указани от програмния брояч за изпълнение като инструкции.

8.1.2. Начално въвеждане на елементите на масивите в паметта

След като се определи местоположението на масива, необходимо е да се въведат елементите му. Началното въвеждане на стойности може да се осъществи с поредица от инструкции LDA и STA:

```
1 LDA ST1
2 STA MASIV
3 LDA ST2
4 STA MASIV+1
5 LDA ST2
6 STA MASIV+2; и т. н.
```

Асемблер-редакторите използуват за същата цел директивата DFB. Тя се използува за едновременно запазване на памет за масив и за начално въвеждане на данни в него. Стойностите за записване в масива се задават в адресното поле на директивата, разделени със запетаи, като при това могат да се задават както шестнайсетични, така и десетични числа.

По подобен начин се използуват и директивите ASC и HEX.

При първата се запазва памет за масив, чиито елементи съдържат ASCII-кодовете на знаци. Такива масиви се използуват при обработка на текстова информация. Знаците се задават в адресното поле като низ, ограничен от двете страни с кавички. С втората директива се запазва памет за масив, съдържащ само шестнайсетични стойности. Тези стойности образуват поредица от двуцифрени шестнайсетични числа, които се задават в адресното поле без разделители. Различните начини за първоначално въвеждане на данни в масиви са показани в примера:

```
1  MASIV   DFS 3              ; запазва се памет за
                              масив с 3 елемента по
                              1 байт
2          LDA #$01           ; в масива последова-
3          STA MASIV          телно се въвеждат
4          LDA #$02           стойностите $01,
5          STA MASIV+1        ; $02
6          LDA #$03
7          STA MASIV+2        ; и $03
8  BYTE    DFB $FF, #20       ; запазва се памет за
                              масив с 2 елемента и в
                              него се въвеждат
                              шестнайсетичното
                              число $FF и десетично-
                              то 20
9  MHEX    HEX FF00FF         ; запазва се памет за
                              масив с 3 елемента и
                              се въвеждат стойнос-
                              тите $FF, $00 и $FF
10 TEKCT   ASC „АСЕМБЛЕР"     ; запазва се памет за
                              масив с 8 елемента и в
                              него се въвеждат
                              ASCII-кодовете на зна-
                              ците, образуващи ду-
                              мата „АСЕМБЛЕР"
```

8.2. Индексиране

За осъществяването на достъп до всеки отделен елемент от даден масив е необходимо да се изчисли неговият адрес. Адресът на първия елемент в масива се приема за основен, а адресите на отделните елементи се изчисляват, като към този адрес се прибавя съответен брой байтове, наречен отместване. Тази техника на достъп до елементите на масив се нарича индексиране и за прилагането ѝ при програмиране на АСЕМБЛЕР се използуват операторите с индексно адресиране. Съществуват два основни типа индексно адресиране — пряко и косвено.

8.2.1. Пряко индексно адресиране

При този тип адресиране се осъществява индексиране със стойността на регистрите X и Y спрямо някакъв основен адрес. Например при масив, чиито елементи заемат по 2 байта всеки, адресът на първия елемент съвпада с основния, адресът на втория елемент е равен на основния плюс отместване от 2 байта, адресът на третия елемент е равен на основния плюс отместване от 4 байта и т. н. Лесно може да се пресметне, че в масив, чиито елементи заемат M байта всеки, адресът на N-тия елемент се получава, като към основния адрес се прибави отместване от (N—1)* M байта. В следващия пример се използува индексиране по X за нулиране на елементите на масива NULA

```
1       LDA #$0        ; в акумулатора и регистъра X
2       TAX            ; се зарежда 0
3 NL    STA NULA, X    ; елементът от масива, чийто
                       индекс се определя от X, се нулира
4       INX            ; регистърът X се увеличава с 1
5       BNE NL         ; при ненулева стойност на X
                       цикълът се повтаря за следващия елемент от масива
6 NULA  DFS XX         ; масивът се разполага тук
```

Тъй като всеки от индексните регистри може да приема стойности от $00 до $FF, с един от тях при неизменен основен адрес могат да се зададат до 256 различни адреса. Затова при работа с по-големи масиви се използува индексиране, съчетано с последователна промяна на основния адрес. Така броят на указваните адреси може да се увеличи до 65 536 или точно толкова, колкото е цялата налична памет на Правец-82. В следващия пример се инвертират битовете от съдържанието на байтовете от адрес $4000 до $5FFF. След изпълнението на тази програма графичното изображение, заредено във втората страница за графика с висока разделителна способност, се преобразува в инверсно:

```
1 NEG LDY #$00         ; началната стойност на Y е 0
2 L   LDA $4000, Y     ; в акумулатора се зарежда
                       съдържанието на байта,
                       указан чрез основния адрес и Y
3     EOR #$FF         ; битовете му се инвертират
4 S   STA $4000, Y     ; резултатът се връща в същия адрес
```

```
 5      INY          ; Y се увеличава с 1
 6      BNE L        ; при ненулева стойност на
                     Y цикълът се повтаря за
                     следващия байт
 7      INC L+2      ; ако Y=0, основният адрес
 8      INC S+2      ; се увеличава с 1
 9      LDA L+2
10      CMP #$60     ; ако не е достигнат край-
11      BNE NEG      ният адрес $6000, цикълът
                     се повтаря
12      LDA #$40     ; след изпълнение на програ-
                     мата
13      STA L+2      ; основният адрес се
14      STA S+2      ; възстановява за следващо
15      RTS          изпълнение
```

В тази програма се използува самомодифициране (изменение) на операндите в програмните редове 2 и 4 с последващо възстановяване в редове 12, 13 и 14. Тук се използува пряко пълно адресиране с индексиране по Y. При него, както и при прякото пълно адресиране с индексиране по X, основният адрес, спрямо който се индексира, се записва с два байта и може да бъде от $0000 до $FFFF. Организирани са два вложени цикъла. С вътрешния цикъл елементите на масива се индексират по Y. Ако при събирането на съдържанието на индексния регистър с младшия байт на основния адрес резултатът е над $FF, преносът се пренебрегва. Поради това е организиран външен цикъл, с който при всяко „превъртане" на индексния регистър старшият байт на основния адрес в съответните инструкции се увеличава с единица.

Съществуват и две къси форми за индексно адресиране — пряко адресиране в нулевата страница с индексиране по X и пряко адресиране в нулевата страница с индексиране по Y. При тези начини на адресиране се използува основен адрес от един байт, към който се прибавя стойността на съответния индексен регистър. Ако резултатът от това събиране надхвърля $FF, за действителен адрес се приемат само младшите 8 бита. По този начин се адресират клетките от $00 до $FF, образуващи нулевата страница на паметта. Използуването на къси вместо пълни форми на индексно адресиране спестява по един байт за всяка инструкция.

8.2.2. Косвено индексно адресиране

При косвено индексно адресиране се работи с поредици от предварително изчислени адреси, записани в паметта. Достъпът до необходимите данни се осъществява на два етапа. Стойността, записана в адресното поле на инструкцията, се

нарича косвен адрес и указва точно определена клетка от нулевата страница на паметта. От тази клетка се извлича действителният (прекият, ефективният, изпълнителният) адрес на данните.

Инструкциите за микропроцесора 6502 използват два различни начина за косвено индексно адресиране. При косвеното адресиране с предварително индексиране по X косвеният адрес е основата, спрямо която се индексира с помощта на регистъра X. В указаната по този начин клетка от нулевата страница се намира младшият, а в следващата след нея — старшият байт на прекия адрес на данните. При косвеното адресиране с последващо индексиране по Y с косвения адрес се указва клетка от нулевата страница. Нейното съдържание се използува като основа за индексиране с регистъра Y. Резултатът от събирането на съдържанието на клетката и съдържанието на индексния регистър е младшият, а резултатът от събирането на възникналия пренос със съдържанието на следващата клетка — старшият байт на прекия адрес.

Тези два начина на адресиране се използуват главно, когато при разработването на програмата не се знае предварително точната стойност на адреса, от който ще се извличат данните и/или се налага през време на изпълнението на програмата тази стойност да се променя. В такъв случай в нулевата страница се зареждат последователно необходимите стойности, които указват различни адреси, а съдържанието на съответния индексен регистър се използува като указател към една от клетките в нулевата страница. Например, ако в дадена програма се използуват ключовете за управление на изображението на екрана, съответните адреси от $C050 до $C057 трябва да се заредят в последователни клетки от нулевата страница и с помощта на индексния регистър да се определи коя от тях се използува в момента.

```
 1        LDA #$C0
 2        LDY #$0F          ; организира се цикъл по Y, с
 3        LDX #$57          ; който в адреси от $F0 до
 4  C0    STA $F0, Y        ; $FF се зареждат младшите
 5        STX $EF, Y        ; и старшите байтове на
 6        DEX               ; ключовете от $C050 до
 7        DEY               ; $C057
 8        DEY
 9        BPL C0
10        LDX KLADR         ; с X се избира клетката, съ-
                            ответстваща на желания
                            ключ
11        LDA ($F0, X)      ; включва се избраният ключ
```

При изпълнението на тази програма, ако регистърът X има

стойност 0, се използува ключът за извеждане на графика $C050, при стойност 1 — ключът за извеждане на текст $C051, при стойност 2 — ключът за неразделен екран $C052 и т. н.

8.2.3. Косвено пълно адресиране

Когато в програмата е необходимо да се извърши преход към адрес, който не е известен предварително, прилага се непряко пълно адресиране. Този начин на адресиране се използува само от инструкцията JMP. Стойността в адресното поле е косвен адрес. Той определя клетката от паметта, в която се намира младшият, а в следващата — старшият байт на действителния адрес. Тези два байта се зареждат в програмния брояч, като по този начин се изпълнява преход в изпълнението на програмата. Обикновено действителният адрес се получава в резултат на някакви изчисления и се записва в определена клетка, чийто адрес се използува като косвен.

```
1 PREH DFS 2          ; запазват се 2 байта памет, където се записва адресът за преход
2       CLC
3       LDA AM
4       ADC BM
5       STA PREH
6       LDA AC
7       ADC BC
8       STA PREH+1
9       JMP (PREH)    ; преход към адреса, записан в клетките PREH и PREH+1
```

В този пример адресът, към който трябва да се осъществи преходът, се получава като сума на двубайтовите числа AC, AM и BC, BM.

9. ВХОДНО-ИЗХОДНИ ДЕЙСТВИЯ

Едно от основните предимства на персоналните микрокомпютри е, че те позволяват да се провежда пълноценен диалог между машината и човека. Ето защо всяка завършена програма трябва да включва както входни действия — за въвеждане на данните за обработка, така и изходни действия — за извеждане на междинните и крайните резултати. Входно-изходните действия се осъществяват през едно или повече от периферните устройства — печатащи устройства, плотери, модеми, светлинни писалки, четци, различни аналогови устройства и др.

Входно или изходно устройство се избира чрез две от подпрограмите, съставящи системния монитор. Номерът на съответния слот (куплунг) се зарежда в акумулатора и след това се извършва преход към адрес $FE95 (подпрограма OUTPORT) за избор на изходно устройство или към адрес $FE8B (подпрограма INPORT) за избор на входно устройство. Най-често за входно устройство се използува клавиатурата на компютъра, а за изходно — екранът на видеомонитора. За целта като параметри към подпрограмите INPORT и OUTPORT се задават нулеви стойности. Данните се въвеждат и извеждат, като към избраното периферно устройство съответно се подава или от него се приема поредица от данни, представени с по един байт.

9.1. Извеждане на данни на екрана

В паметта на компютъра Правец-82 областта от адрес $400 до адрес $4FF се нарича екранна памет и нейното съдържание съответствува на извеждания текст. Записването на стойност в клетка от тази област води до изписването на знака със съответния ASCII-код на екрана.

9.1.1. Извеждане на знаци

Извеждането на знаци може да се осъществи сравнително просто. Ако например на екрана трябва да се изпише „ТЕКСТ", ASCII-кодовете на отделните букви се зареждат в последователни клетки от екранната памет:

```
1          LDX #$00
2 ЕКРАН    LDA NADP, X
3          STA $400, X
4          INX
5          TXA
6          CMP #$05
7          BNE ЕКРАН
8          RTS
9 NADP     ASC „TEKST"
```

При този начин на извеждане на знаците съществува едно неудобство — разположението на извеждания знак се определя доста трудно поради особената организация на екранната памет. Много по-удобно е да се работи с абсолютните координати на знака върху екрана. Такава възможност предоставят включените в системния монитор подпрограми. Най-използваната от тях е COUT (адрес $FDF0). При преход към нея тя отпечатва върху екрана знак, чийто ASCII-код се намира в акумула-

тора. Редът и колоната на извеждането се определят от текущото местоположение на показалеца. След изпълнението на подпрограмата COUT показалецът се премества с една позиция надясно, а ако това е последната позиция на реда, премества се на следващия ред. Друга често използувана подпрограма е TABV (адрес $FB5B). Тя премества показалеца на ред, определен от съдържанието на акумулатора. В следващия пример с помощта на тези подпрограми и на клетката с адрес $24, определяща хоризонталната позиция на показалеца, е организирана подпрограма, с която могат да се изписват знаци в зададени ред и колона на екрана. При обръщение към нея параметрите (V — номер на реда, N — номер на колоната и SYM — ASCII-код на знака) се предават чрез клетките $FD, $FE и $FF.

```
1  TABV   EQU $FB5B
2  TABH   EQU $24
3  COUT   EQU $FDF0
4  H      EQU $FD
5  V      EQU $FE
6  SYM    EQU $FF
7         LDA V          ; показалецът се пре-
8         JSR TABV       ; места на ред V
9         LDA H          ; показалецът се пре-
10        STA TABH       ; места в колона N
11        LDA SYM        ; изписва се знакът
12        JSR COUT       ; ASCII-код, равен на SYM
13        RTS
```

Към екранната памет могат да се подават както кодовете на нормални, инверсни или мигащи знаци, така и на контролни и някои специални знаци. Те не се изписват на екрана, но предизвикват определени действия от страна на процесора, т. е. изпълняват се като команди. Действието на някои от тях е показано в следния пример:

```
1  COUT   EQU $FDF0
2         LDA #$87       ; МК—Г
3         JSR COUT       ; предизвиква звуков сигнал
4         LDA #$88       ; МК—Х
5         JSR COUT       ; премества показалеца на-
                         ляво
6         LDA #$95       ; МК—У
7         JSR COUT       ; премества показалеца на-
                         дясно
8         LDA #$8D       ; RETURN
9         JSR COUT       ; премества показалеца в
                         началото на следващия ред
10        LDA #$8A       ; LINE FEED
11        JSR COUT       ; премества каретката на
```

```
                              печатащото устройство на
                              нов ред
12          LDA #$83          ; MK—Ь
13          JSR COUT          ; прекъсване
```

От тези специални знаци най-често се използува $8D. Включването му в началото на изписвания текст води до отпечатването на нов ред. Например изписването на три думи, разделени една от друга с по един празен ред, може да стане по следния начин:

```
 1 COUT  EQU $FDF0
 2       LDX #$00
 3 XXX   LDA DUMA, X
 4       BEQ STOP
 5       JSR COUT
 6       INX
 7       JMP XXX
 8 STOP  RTS
 9 DUMA  ASC „ТОВА"
10       DFB $8D, $8D
11       ASC „Е"
12       DFB $8D, $8D
13       ASC „ТЕКСТ."
14       DFB $00
```

Когато се извежда текст, чиято дължина предварително не е известна, като последен се използува знак, който сигнализира за края на текста. Обикновено за това се използува знакът с ASCII-код $00. Той е особено удобен поради това, че извеждането на поредицата от знаци продължава дотогава, докато се срещне ASCII-код $00 и вследствие на това се установи флагът за нула. По този начин е организирана и подпрограмата от интерпретатора на БЕЙСИК STROUT с начален адрес $DB3A. Тя извежда низа с начален адрес, указан от съдържанието на акумулатора (младши байт) и регистър Y (старши байт).

```
1           ORG $300
2 STROUT    EQU $DB3A
3           LDA #$07
4           LDY #$03
5           JSR STROUT
6 TEXT      ASC „ПРАВЕЦ 82" ; това е текстът за
                              извеждане — начал-
                              ният му адрес е $307
7           BRK               ; използува се вместо 0
```

В системния монитор и интерпретатора на БЕЙСИК са включени редица подпрограми, свързани с извеждането на

текст. Тяхното познаване и използуване предоставя на програмиста допълнителни възможности при оформянето на програмния диалог.

9.1.2. Извеждане на числови данни

Тъй като в паметта на компютъра стойностите обикновено се представят като шестнайсетични числа, те могат да бъдат изведени най-лесно в този вид и на екрана. Системният монитор съдържа подпрограма, наречена PRBYTE или HEXOUT, която извежда като шестнайсетично число съдържанието на акумулатора. Нейният начален адрес е $FDDA. Ето как с нейна помощ могат да се отпечатат всички числа от $00 до $FF:

```
1 PRBYTE   EQU $FDDA
2 PRBLNK   EQU $F948     ; тази подпрограма от
                         системния монитор из-
                         вежда три интервала
3          LDY #$00
4 HEXA     TYA
5          JSR PRBYTE
6          JSR PRBLNK
7          INY
8          BNE HEXA
```

Шестнайсетичните числа с дължина повече от един байт могат да се извеждат байт по байт от най-старшия към най-младшия, като се използува същата подпрограма PRBYTE. Следната програма извежда последователно числата от $EF00 до $F5F0 със стъпка $10:

```
1 PRBYTE   EQU $FDDA
2 PRBLNK   EQU $F948
3 ML       EQU $FE       ; младши и
4 ST       EQU $FF       ; старши байт на
                         числото
5          LDA #$00
6          STA ML
7          LDA #$EF
8          STA ST
9 MULTI    LDA ST
10         JSR PRBYTE    ; отпечатва се
                         старшият байт
11         LDA ML
12         JSR PRBYTE    ; отпечатва се
                         младшият байт
13         JSR PRBLNK    ; отпечатват се
                         три интервала
14         LDA ML
```

```
15              CLC
16              ADC #$10        ; прибавя се стъпка-
                                  та $10 към младшия
                                  байт
17              STA ML
18              BNE MULTI       ; проверка за 0 в
                                  младшия байт
19              INC ST          ; увеличава се с 1
                                  старшият байт
20              LDA ST
21              CMP #$F6        ; проверка за край на
                                  цикъла
22              MULTI
```

Тъй като най-често се работи с двубайтови числа, в системния монитор са включени две подпрограми за извеждане на числа с такава дължина. Първата от тях се нарича PRNTAX и започва от адрес $F941. Тя извежда двубайтово шестнайсетично число, като старшият байт се извлича от акумулатора, а младшият — от регистъра X. След изпълнението ѝ съдържанието на акумулатора се разрушава. Втората се нарича PRYX2 и започва от адрес $FD96. Тя предизвиква преминаване на нов ред, след което извежда съдържанието на регистрите X и Y като двубайтово шестнайсетично число. Ако не е необходимо преминаването на нов ред, тази подпрограма може да се извиква и от адрес $FD99.

За потребителите е много по-удобно извежданите от програмата числа да бъдат в десетичен вид. За тази цел съответните шестнайсетични стойности трябва предварително да се преобразуват. Съществуват различни алгоритми за превръщане на числа от една бройна система в друга, но включването им би утежнило излишно всяка програма. Вместо това е по-просто да се използува подпрограмата от интерпретатора на БЕЙСИК, която започва от адрес $ED24 и се нарича LINPRT. Тя извлича от регистъра X младшия и от акумулатора — старшия байт на шестнайсетично число и го извежда в десетично представяне. Така например извеждането на всички числа от 0 до 100 може да се извърши така:

```
1 LINPRT        EQU $ED24
2 PRBLNK        EQU $F948
3 ML            EQU $FE         ; младши и
4 ST            EQU $FF         ; старши байт на
                                  числото
5               LDA #$00
6               STA ML
7               STA ST
8 DESET         LDA ST
9               LDX ML
```

```
10          JSR LINPRT      ; отпечатва се числото
11          JSR PRBLNK      ; отпечатват се 3 интервала
12          INC ML          ; увеличава се младшият байт
13          LDA ML
14          CMP #$65        ; проверка за край на цикъла
15          BNE DESET
```

9.2. Въвеждане на данни от клавиатурата

Данни се въвеждат най-често от клавиатурата, като се използуват две клетки от паметта за входно-изходни действия. Първата от тях е $C000, нарича се KEY и ако нейният седми бит е установен, тя съдържа ASCII-кода на последния натиснат клавиш. Втората е $C010 и записването на каква да е стойност в нея изчиства седмия бит в $C000, с което се освобождава място за въвеждане на нов знак чрез натискане на следващ клавиш.

9.2.1 Въвеждане на знаци

От клавиатурата могат да се въвеждат знаци чрез периодично четене от $C000 и изтриване чрез достъп до $C010. Например.

```
1 KEY       EQU $C000
2 STROBE    EQU $C010
3 SYM       LDA KEY         ; стойността на $C000 се чете,
4           BPL SYM         докато се установи седмият бит
5           STA STROBE      ; $C000 се изчиства за въвеждане на нов знак
```

За въвеждане на знаци се използува и подпрограмата от системния монитор RDKEY с начален адрес $FD0C. При обръщение към нея в текущата позиция се извежда показалецът и управлението се предава към програмата, чийто адрес е записан в адреси $38 и $39. Обикновено те съдържат началния адрес на подпрограмата KEYIN ($FD18), която прочита клавиш от клавиатурата и записва ASCII-кода му в акумулатора. С използуване на тези подпрограми може да се въведе поредица от знаци, техните ASCII-кодове да се разположат в последователни клетки от паметта и след това да се изведат на екрана като цял низ.

```
 1 RDKEY    EQU $FD0C     ; подпрограма за въ-
                          веждане на знак
 2 COUT     EQU $FDED     ; подпрограма за из-
                          веждане на знак
 3 STROUT   EQU $DB3A     ; подпрограма за из-
                          веждане на низ
 4 TEXT     EQU $2000     ; начало на облас-
                          та, където се за-
                          реждат знаците
 5          LDX #$00      ; инициализира се
                          регистърът
 6 IN       JSR RDKEY     ; въвежда се знак
            JSR COUT      ; въведеният знак се
                          извежда на екрана
 7          CMP #$8D      ; проверява се за
                          RETURN
 8          BEQ OUT       ; преход при RETURN
 9          STA TEXT, X   ; знакът се зарежда
                          в следващата сво-
                          бодна клетка
10          INX           ; увеличава се броя-
                          чът
11          JMP IN        ; преход за въвежда-
                          не на следващ знак
12 OUT      LDA #$00      ; подготовка за из-
                          веждане —
13          STA TEXT, X   ; на края на низа се
                          добавя ASCII-код $00
14          LDY #$20
15          JSR STROUT    ; извеждане на низа
```

Тази подпрограма може да се използува за въвеждане на низ, но тя има два много съществени недостатъка. Първият от тях е, че по този начин могат да се въведат най-много 256 знака, след което броячът се превърта и знаците започват да се въвеждат на мястото на вече въведените. Друг недостатък е, че веднъж въведен, низът не може да бъде поправян чрез MK—Y, MK—X или OCB. Тези неудобства могат да се избегнат, като за въвеждане на низ се използува подпрограмата от системния монитор GETLN, която започва от адрес $FD6A. При преход към нея се прочита съдържанието на клетката от нулевата страница $33, наречена PPOMPT, извежда се знакът, чийто ASCII-код е записан в нея, и се въвежда поредица от знаци. Тези знаци могат да се поправят или копират, да се отменя цялата въведена поредица и т.н. При въвеждане на повече от 255 знака се извежда знакът \, въведеният низ се отменя и въвеждането започва отначало. Краят на низа се означава чрез въвеждане на ASCII-кода

за RETURN $8D. След изпълнението на подпрограмата ASCII-кодовете на знаците, образуващи низа, са заредени последователно в клетките от $200 до $2FF, а техният общ брой е записан в индексния регистър X. Подпрограмата GETLN може да се извика и от адрес $FC67 или $FD6F. В първия случай преди изпълнението се преминава на нов ред, а във втория не се извежда знакът от PROMPT. С използуването на GETLN последният пример може да се програмира и така:

```
1  GETLN     EQU $FDA6
2  PROMPT    EQU $33
3  STROUT    EQY $D83A
4            LDA #$21        ; за показалец се из-
5            STA PROMPT      : ползува знакът !
6            JSR GETLN
7            LDY #$02
8            LDA #$00
9            INX
10           STA $200, X
11           JSR STROUT
```

9.2.2. Въвеждане на числови данни

При работа с числови стойности отделните цифри се въвеждат от клавиатурата с ASCII-кодовете от $B0 до $B9. Затова е необходима допълнителна преработка за получаване на желаните шестнайсетични стойности. След всяко въвеждане на знак се проверява дали той е цифра. Ако се установи грешка, тя се отбелязва със звуков сигнал, въведеният знак се отменя и въвеждането се повтаря. Ако е въведена цифра, в нейния ASCII-код се нулират старшите четири бита, за да се получи стойността на числото, представяно с цифрата. Ако цифрата е втори или следващ разред, стойността се умножава по 10, 100 и т. н. Накрая всички стойности се събират за получаване на крайния резултат. Това е показано в следната програма, с която се въвежда двуцифрено десетично число и след това се извежда в шестнайсетично представяне:

```
1  RDKEY     EQU $FD0C
2  COUT      EQU $FDED
3  HEXOUT    EQU $FDDA
4  BELL      EQU $FBDD
5  CS        EQU $FF
6            LDA #$00
7            STA CS
8            JSR IN          ; въвежда се първа-
                             та цифра
```

```
 9              STA CS
10              JSR IN          ; въвежда се втора-
                                  та цифра
11              BCS OUT         ; при RETURN — из-
                                  веждане
12              TAY
13              LDA CS          ; зарежда се първата
                                  цифра
14              JSR X10         ; умножава се по 10
15              STA CS
16              TYA
17              CLC
18              ADC CS          ; стойностите на
                                  двата разреда
19              STA CS          ; се събират
20 OUT          LDA #$BD        ; подпрограма за из-
                                  веждане
21              JSR COUT        ; извежда се знакът
                                  =
22              LDA #$AS
23              JSR COUT        ; извежда се знакът
                                  $
24              LDA CS          ; извежда се въведе-
25              JSR COUT        ; ната стойност в
26              RTS             ; шестнайсетично
                                ; представяне
27 IN           CLC             ; подпрограма за въ-
                                  веждане
28              JSR RDKEY       ; въвежда се знак
29              CMP #$8D        ; проверява се дали е
                                  RETURN
30              BNE RT
31              SEC             ; при RETURN се ус-
                                  тановява флагът за
                                  пренос
32              JMP END
33 RT           JSR COUT        ; знакът се извежда
                                  на екрана
34              JSR PROV        ; проверява се дали е
                                  цифра
35              BCS IN          ; ако не е, въвежда-
                                  нето се повтаря
36              AND #$0F        ; изчистват се
                                  старшите 4 бита
37 END          RTS
38 PROV         CMP #$B0        ; подпрограма за
39              BMI GR          ; проверка на знак
```

```
40              CMP #$BA
41              BPL GR
42              RTS
43 GR           JSR BELL        ; подпрограма за по-
                                  правка
44              LDA #$88        ; връща показалеца с
                                  1 позиция
45              JSR COUT        ; наляво
46              LDA #$A0        ; извежда 1 интер-
                                  вал
47              JSR COUT
48              LDA #$88
49              JSR COUT
50              SEC
60              RTS
61 X10          ASL             ; подпрограма за ум-
62              STA $FE         ; ножаване на съдър-
63              ASL             ; жанието на акуму-
64              ASL             ; латора по 10
65              CLC
66              ADC SFE
67              RTS
```

По подобен начин може да се организира въвеждането и на многобайтови десетични или шестнайсетични числа.

9.3. Някои други входно-изходни действия

Освен клавиатурата и екрана на монитора при създаването на програми на АСЕМБЛЕР често се използуват входно-изходни действия през куплунга за джойстик или изходни действия към високоговорителя. Това се отнася особено за компютърните игри, в които управлението с ръчки за игра (джойстик) и звуковите ефекти са неотменна част. В някои примери от предните глави е показано как чрез записване на каква да е стойност на адрес $C030 се обръща мембраната на високоговорителя. С поредица от такива действия, разделени от подходящо избрани паузи, е възможно да се произведат най-разнообразни звуци. Високоговорителят може да се управлява и с подпрограмите от системния монитор BELL1 ($FBDD) и BELL ($FF3A).

Игровият куплунг се използува при включване на ръчки за игра, които обикновено включват 2 или 3 бутона и 2 или 4 ръчки, свързани с потенциометри. Състоянието на бутоните (т. е. натиснат или отпуснат) може да се установи чрез проверка на

адрес $C061 за бутон 0, $C062 — за бутон 1 и $C063 — за бутон 2. Ако бутонът е натиснат, седмият бит на съответния адрес се установява и се нулира след отпускането му.

По същия начин се проверява и състоянието на потенциометрите от ръчките за игра. Техните стойности (от 0 до 255) се съхраняват в адресите от $C064 до $C067. За тази цел може да се използува и подпрограмата от системния монитор PREAD ($FP1E). Тя чете стойността на игровия контролер, чийто номер се намира в регистъра X. Прочетената стойност се записва в регистъра Y,а съдържанието на акумулатора се разрушава. В следния пример се извеждат стойностите, съответстващи на положението на две от ръчките за игра, а при натискане на кой да е бутон се издава звук и програмата спира.

```
 1 PREAD    EQU $FB1E
 2 PRBLNK   EQU $F948
 3 CROUT    EQU $FD8E        ; извежда RETURN
 4 HEXOUT   EQU $FDDA
 5 BELL     EQU $FF3A
 6 BUTON    EQU $C061
 7          LDA #$02         ; ограничава тек-
 8          STA $23.         стовия прозорец на
                             екрана
 9 CTAPT    JSR CROUT
10          LDX #$00
11          JSR PREAD
12          TYA
13          JSR HEXOUT
14          JSR PRBLNK
15          LDX #$01
16          JSR PREAD
17          TYA
18          JSR HEXOUT
19          LDA BUTON
20          BMI ZV           ; проверка на бутон 0

21          JMP CTAPT
22 ZV       JSR BELL
23          JSR BELL
```

10. ПРОВЕРКА НА ПРОГРАМИ И ОТСТРАНЯВАНЕ НА ГРЕШКИ

Почти винаги в процеса на проектиране и написване на една асемблерска програма се допускат грешки, които водят до един от следните три случая:

— програмата въобще не може да се изпълни;

— програмата се изпълнява неправилно, т. е. не по предварително зададената в алгоритъма логическа последователност;

— програмата се изпълнява правилно, но получените резултати се различават от очакваните. Ето защо проверката на програмите и отстраняването на грешките е особено важен етап в създаването на програми.

10.1. Проверка на програми

Проверката на програмите (този процес се нарича още тестване) има за цел да установи наличието или отсъствието на грешки в програмата. Разбира се, понятието отсъствие на грешки е доста условно. В идеалния случай програмата трябва да бъде проверена при всички възможни комбинации от входни данни и всички получени изходни данни да бъдат сравнени с предварително правилно изчислени данни. В повечето случаи това е практически невъзможно. Затова е разумно проверката на програмата да бъде прекратена, когато броят на оставащите в нея грешки е относително малък и усилията за тяхното откриване не са оправдани. Този момент обаче се определя особено трудно и зависи главно от задачите, умението и практическия опит на конкретния програмист.

Съществуват различни подходи за извършване на проверката. Най-често се използува структурният подход — съставна част от общия процес на структурното програмиране. Първият етап при него е обща проверка на работата на програмата. Ако се установят грешки в нея, пристъпва се към втория етап, на който цялата програма се разделя на отделни съставни модули и се проверява всеки от тях. Разделянето е лесно, когато програмата е съставена в съответствие с принципите на структурното програмиране и съдържа относително независими една от други подпрограми. Модулите, в които се установят грешки, се разбиват на подмодули и проверката продължава по описания начин. Когато се достигне най-ниското ниво, възможно е проверката да бъде пълна, т. е. за всички комбинации от входни данни. Последният етап включва обратния процес на свързване на отделните модули и на проверка за наличието на грешки при осъществяване на връзките между тях.

10.2. Видове грешки

В резултат от проверката се получава списък на грешките в програмата. Според характера им и причините за възникването им те могат да бъдат класифицирани в няколко основни групи.

10.2.1. Синтактични грешки

Синтактични са грешките, които се дължат на неспазване на правилата за образуване на изразите в езика АСЕМБЛЕР. Този тип грешки се откриват от асемблер-редактора през време на асемблиране на първичната програма и се извеждат съобщения за вида им. Асемблер-редакторът MERLYN разпознава и извежда съобщения за следните видове синтактични грешки:

BAD OPCODE	— трибуквеното съкращение за означаване на типа на операцията не съответствува на нито един от действителните мнемонични кодове;
BAD ADRESS MODE	— неправилен начин на адресиране;
BAD BRANCH	— опит за условен преход извън допустимите граници ∓127 байта;
BAD OPERAND	— неправилно определен операнд;
DUPLICATE LABEL	— повторно използуване на етикет;
MEMORY FULL	— препълване на паметта;
UNKNOWN LABEL	— опит за използуване на етикет, който не е определен предварително;
NOT MACRO	— опит за използуване на макрос, който не е определен предварително;
NESTING ERROR	— превишаване на допустимия брой на нивата на влагане (15 при макросите и 8 при директивите за условно асемблиране);
BAD „PUT"	— опит за използуване на PUT в макрос или във файл, въвеждан също с PUT;
BAD „SAV"	— опит за използуване на SAV в макрос;
BAD INPUT	— липса на данни от клавиатурата или данни с дължина повече от 37 знака при използуване на KBD;
BREAK	— прекъсване, предизвикано от ненулева стойност на израза

в полето на операнда след ERR;

BAD LABEL — директива EQU или MAC без етикет, етикет с дължина повече от 13 знака или съдържащ непозволени знаци.

10.2.2. Програмни грешки

Този тип грешки се дължат на несъответствие между логическата конструкция на алгоритъма и практически осъществената конструкция в програмата. Тук са приведени най-често срещаните логически грешки.

1. *Логически грешки:*
— неправилно отчитане на всички възможни условия;
— неправилен преход след проверка на някакво условие;
— липса на блок от алгоритъма в програмата;
— използуване на неподходящи оператори;
— използуване на неподходящи начини на адресиране.

2. *Грешки при организиране на цикли:*
— неправилно определяне на началото на цикъла;
— неправилно определяне на условието за край на цикъла;
— неправилно определяне броя на изпълненията на цикъла;
— преход преди завършване на цикъла;
— безкраен цикъл.

3. *Грешки при аритметични операции:*
— неправилно определяне на типа на числата, с които се работи;
— неотчитане на препълването и загуба на значещ бит;
— използуване на неверни константи;
— неправилен ред на извършване на аритметичните операции;
— неправилно изпълнение на аритметични операции с повишена точност;
— неправилно изпълнение на умножение, деление, степенуване, коренуване и др.

4. *Грешки при логически операции:*
— неправилен избор на логическа операция;
— неотчитане на всички възможни резултати от логическата операция.

5. *Грешки при работа с двоични низове:*
— неправилно определяне дължината на двоичния низ;
— неправилно изпълнение на изместване или ротация с повишена точност.

6. *Грешки при работа със стека:*
— неправилен ред на записване и извличане на данни от стека;
— неотчитане на превъртането на стека.

7. *Грешки при организиране на подпрограми:*

— неправилно обръщение към подпрограма;
— неправилно предаване на данни към и от подпрограма;
— неправилно организиране на вложени подпрограми.

8. *Грешки при работа с масиви:*
— неправилно определяне на размерността и размера на масива;
— неправилно въвеждане на първоначални стойности на елементите на масива;
— неправилно индексиране на елементите на масива.

9. *Грешки при входно-изходни операции:*
— неправилен избор на входно или изходно устройство;
— неправилно определяне типа на данните за въвеждане или извеждане;
— използуване на неподходяща подпрограма за вход или изход.

10. *Други грешки.*

10.3. Програмни средства за проверка на програми

10.3.1. Системен монитор

Най-достъпното програмно средство за проверка на асемблираните програми е системният монитор на Правец-82. Чрез него могат да се зареждат програмно достъпните регистри, да се проверява съдържанието им, да се стартират отделни части от програмата, съдържанието на една област от паметта да се прехвърля в друга област, да се проверява и изменя съдържанието на клетки от оперативната памет. Например, ако трябва да се провери подпрограма, започваща от адрес $300, тя може да се стартира от монитора с командата 300G. Тя се изпълнява точно както JSR с тази разлика, че адресът за връщане, който се зарежда в стека, е началният адрес на монитора. Ако към тази подпрограма трябва да се предадат данни с помощта на акумулатора и двата индексни регистъра, това може да стане с помощта на следната последователност от команди:

```
        : MK—E          — след нея на екрана ще се появи текущото съдържание на регистрите, като например
A=98    X=00    Y=12    P=10    S=98
        :00 00 00       — с тази команда се нулира съдържанието на трите регистъра. Ако след това се използува отново команда MK—E, на екрана ще се изпише
A=00    X=00    Y=00    P=00    S=B7
```

Така може да се променя съдържанието на кой да е от петте регистъра. Ако трябва да се промени съдържанието на един от тях, съдържанието на останалите трябва да се повтори. По същия начин могат да се предават данни към подпрограмата и през определени клетки от паметта и да се проверява съдържанието им след изпълнението ѝ. Например проверката на съдържанието на адресите от $330 до $337 може да се извърши така:

```
330 RETURN
330—00
RETURN
08 00 08 00 08 00 08
```

Зареждането на стойности в същите адреси става така:

```
*330:00 01 02 03 04 05 06 07
```

10.3.2. Програмни продукти за проверка на програми и отстраняване на грешки

Използуването на системния монитор за проверка на една асемблирана програма не позволява да се проследи изпълнението ѝ. За тази цел се използуват специални програми, т. нар. дебъгери. Такива програмни продукти са 6502 SIMULATOR, TRACE 65, BUGBYTER, LOCKSMITH 6.0/BOOT TRACER и др. Те предоставят на програмиста възможност да следи на всяка стъпка от изпълнението на програмата съдържанието на регистрите, на флаговете за състоянието и на предварително определени клетки от паметта, вида на изпълняваната инструкция и използувания начин на адресиране и т. н. С тяхна помощ откриването на грешките и причините за възникването им става по-бързо и с по-голяма сигурност.

10.4. Отстраняване на грешки

След откриване на грешките в програмата те трябва да бъдат отстранени. Асемблер-редакторът MERLYN позволява многократно редактиране и асемблиране на първичната асемблерска програма. Освен това след излизане в интерпретатора на БЕЙСИК или монитора винаги е възможно връщане обратно в асемблер-редактора с командата ASSEM. Често грешките се отстраняват трудно, понеже са взаимнообусловени и взаимносвързани, т. е. отстраняването на една грешка може да доведе до появата на други. Затова програмният ред в първичната асемблерска програма трябва да се поправя само след като е напълно сигурно, че именно той е причина за грешката и че внасяното изменение отстранява грешката. Много важно е да се провери дали поправката не предизвиква странични ефекти и

дали не води до възникване на проблеми и в другите части на програмата.

Проверката на програмата и отстраняването на грешките в нея са две съставни и неделими една от друга части в процеса на създаване на програмите. Този процес се извършва като поредица от непрекъснато повтарящи се стъпки, докато се достигне до едно приемливо качество на асемблерската програма.

ПРИЛОЖЕНИЯ

Приложение 1

Разпределение на нулевата страница от паметта на Правец-82

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
$00	Б	Б	Б	Б	Б	Б	**	**	**	**	Б	Б	Б	Б	Б	Б
$10	Б	Б	Б	Б	Б	Б	Б	Б	Б	**	**	**	**	**	**	
$20	М	М	М	М	М	М	ДМ	ДМ	М	М	ДМ	ДМ	ДМ	ДМ	ДМ	ДМ
$30	М	М	М	М	М	ДМ	ДМ	ДМ	ДМ	ДМ	М	М	М	М	ДМ	ДМ
$40	ДМ	ДМ	ДМ	ДМ	ДМ	ДМ	ДМ	ДМ	ДМ	М	Д	Д	Д	Д	М	М
$50	БМ	БМ	БМ	БМ	БМ	БМ	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б
$60	Б	Б	Б	Б	Б	Б	Б	ДБ	ДБ	ДБ	ДБ	Б	Б	Б	Б	ДБ
$70	ДБ	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б
$80	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б
$90	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б
$A0	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	ДБ
$B0	ДБ	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б	Б
$C0	ДБ	Б	Б	Б	Б	Б	Б	Б	Б	Б	ДБ	ДБ	ДБ	ДБ	**	**
$D0	Б	Б	Б	Б	Б	Б	**	**	ДБ	Б	Б	Б	Б	Б	Б	Б
$E0	Б	Б	Б	**	Б	Б	Б	Б	Б	Б	Б	**	**	**	**	**
$F0	Б	Б	Б	Б	Б	Б	Б	Б	Б	**	**	**	**	**	**	**

Адресите се използуват от:

Б — интерпретатора на БЕЙСИК;

М — системния монитор;

Д — дисковата операционна система (ДОС 3.3);

** — свободен за потребителя адрес.

Таблица на ASCII-кодовете на знаците в микрокомпютъра Правец-82

	латиница							кирилица								
	0 $00	16 $10	32 $20	48 $30	64 $40	80 $50	96 $60	112 $70	128 $80	144 $90	160 $A0	176 $B0	192 $C0	208 $D0	224 $E0	240 $F0
0 $00	@	P		0	@	P		0	@	P		0	Ю	P	@	П
1 $01	A	Q	!	1	A	Q	!	1	A	Q	!	1	A	Q	A	Я
2 $02	B	R	"	2	Б	R	"	2	B	R	"	2	B	R	Б	Р
3 $03	C	S	#	3	C	S	#	3	C	S	#	3	C	S	Ц	С
4 $04	D	T	$	4	D	T	$	4	D	T	$	4	D	T	Д	Т
5 $05	E	U	%	5	E	U	%	5	E	U	%	5	E	U	Е	У
6 $06	F	V	&	6	F	V	&	6	F	V	&	6	F	V	Ф	Ж
7 $07	G	W	'	7	G	W	'	7	G	W	'	7	G	W	Г	В
8 $08	H	X	(	8	H	X	(	8	H	X	(	8	H	X	Х	Ь
9 $09	I	Y	)	9	I	Y	)	9	I	Y	)	9	I	Y	И	Ы
10 $0A	J	Z	*	:	J	Z	*	:	J	Z	*	:	J	Z	Й	З
11 $0B	K	ОСВ	+	;	K	[	+	;	K	[	+	;	K	Ш	К	[
12 $0C	L	\	,	<	L	\	,	<	L	/	,	<	L	/	Л	Щ
13 $0D	M	]	—	=	M	]	-	=	M	]	—	=	M	Щ	М	]
14 $0E	N	^	.	>	N	^	.	>	N	^	.	>	N	Ч	Н	^
15 $0F	O	—	/	?	O	—	/	?	O		/	?	O	—	O	■:

инверсни знаци — мигащи знаци — МК — нормални знаци

Указател на операторите в АСЕМБЛЕР и съответните им машинни инструкции в микропроцесора 6502

В този указател са включени всички оператори в АСЕМБЛЕР. За всеки от тях са показани възможните начини на адресиране, форматът на асемблерския оператор (в който с .. се означава еднобайтова, а с .... — двубайтова шестнайсетична стойност) и машинният код на операцията. След всяка таблица е посочена страницата в книгата, в която за пръв път се среща даденият оператор.

ADC

Събиране на съдържанието на акумулатора с константа или със съдържанието на клетка от паметта и преноса от предишната операция

Начин на адресиране	Формат на асемблерския оператор	Машинен код на операцията
Непосредствено	ADC #$..	69
Пряко в нулевата страница	ADC $..	65
Пряко в нулевата страница с индексиране по X	ADC $..,X	75
Пряко пълно	ADC $....	6D
Пряко с индексиране по X	ADC $....,X	7D
Пряко с индексиране по Y	ADC $....,Y	79
Косвено с предварително индексиране по X	ADC ($..,X)	61
Косвено с последващо индексиране по Y	ADC ($..),Y	71

Променят се флаговете N Z C V. стр. 29

AND

Логическо умножение на битовете от съдържанието на акумулатора със съответните битове на константа или от съдържанието на клетка от паметта

Начин на адресиране	Формат	Машинен код
Непосредствено	AND #$..	29
Пряко в нулевата страница	AND $..	25
Пряко в нулевата страница с индексиране по X	AND $..,X	35
Пряко пълно	AND $....	2D
Пряко с индексиране по X	AND $....,X	3D
Пряко с индексиране по Y	AND $....,Y	39
Косвено с предварително индексиране по X	AND ($..,X)	21
Косвено с последващо индексиране по Y	AND ($..),Y	31

Променят се флаговете N Z. стр. 38

ASL

Изместване с един разред наляво на всички битове от съдържанието на акумулатора или клетка от паметта

Начин на адресиране	Формат	Машинен код
Акумулаторно	ASL	OA
Пряко в нулевата страница	ASL $..	06
Пряко в нулевата страница с индексиране по X	ASL $..,X	16
Пряко пълно	ASL $....	OE
Пряко с индексиране по X	ASL $....,X	1E

Променят се флаговете N Z C. стр. 42

BCC

Условен преход при отсъствие на пренос

Начин на адресиране	Формат	Машинен код
Относително	BCC $..	90

Флаговете не се променят. стр. 26

BCS

Условен преход при наличие на пренос

Начин на адресиране	Формат	Машинен код
Относително	BCS $...	BO

Флаговете не се променят. стр 26

BEQ

Условен преход при нулев резултат

Начин на адресиране	Формат	Машинен код
Относително	BEQ $..	FO

Флаговете не се променят. стр 25

BIT

Сравняване на битовете от съдържанието на клетка от паметта със съответните битове от съдържанието на акумулатора

Начин на адресиране	Формат	Машинен код
Пряко в нулевата страница	BIT $..	24
Пряко пълно	BIT $....	2C

Променят се флаговете N Z V. стр. 39

BMI
Условен преход при отрицателен резултат

Начин на адресиране	Формат	Машинен код
Относително	BMI $..	30

Флаговете не се променят. стр. 26

BNE
Условен преход при ненулев резултат

Начин на адресиране	Формат	Машинен код
Относително	BNE $..	D0

Флаговете не се променят. стр. 25

BPL
Условен преход при положителен резултат

Начин на адресиране	Формат	Машинен код
Относително	BPL $...	10

Флаговете не се променят. стр. 26

BRK
Програмно прекъсване

Начин на адресиране	Формат	Машинен код
С подразбиране	BRK	00

Променя се флагът I. стр. 28

BVC
Условен преход при отсъствие на препълване

Начин на адресиране	Формат	Машинен код
Относително	BVC $..	50

Флаговете не се променят. стр. 26

BVS
Условен преход при препълване

Начин на адресиране	Формат	Машинен код
Относително	BVS $..	70

Флаговете не се променят. стр. 26

CLC
Изчистване на флага за пренос

Начин на адресиране	Формат	Машинен код
С подразбиране	CLC	18

Променя се флагът C. стр. 24

CLD
Изчистване на флага за десетичен режим

Начин на адресиране	Формат	Машинен код
С подразбиране	CLD	D8

Променя се флагът D. стр. 24

CLI
Изчистване на флага за прекъсване

Начин на адресиране	Формат	Машинен код
С подразбиране	CLI	58

Променя се флагът I. стр. 24

CLV
Изчистване на флага за препълване

Начин на адресиране	Формат	Машинен код
С подразбиране	CLV	B8

Променя се флагът V. стр. 24

CMP
Сравняване на константа или съдържанието на клетка от паметта със съдържанието на акумулатора

Начин на адресиране	Формат	Машинен код
Непосредствено	CMP #$..	C9
Пряко в нулевата страница	CMP $..	C5
Пряко в нулевата страница с индексиране по X	CMP $..,X	D5
Пряко пълно	CMP $....	CD
Пряко с индексиране по X	CMP $....,X	DD
Пряко с индексиране по Y	CMP $....,Y	D9
Косвено с предварително индексиране по X	CMP ($..,X)	C1
Косвено с последващо индексиране по Y	CMP ($..),Y	D1

Променят се флаговете N Z C. стр. 25

CPX

Сравняване на константа или съдържанието на клетка от паметта със съдържанието на индексния регистър X

Начин на адресиране	Формат	Машинен код
Непосредствено	CPX #$..	E0
Пряко в нулевата страница	CPX $..	E4
Пряко пълно	CPX $....	EC

Променят се флаговете N Z C. стр. 25

CPY

Сравняване на константа или съдържанието на клетка от паметта със съдържанието на индексния регистър Y

Начин на адресиране	Формат	Машинен код
Непосредствено	CPY #$..	C0
Пряко в нулевата страница	CPY $..	C4
Пряко пълно	CPY $....	CC

Променят се флаговете N Z C. стр. 25

DEC

Намаляване с единица на съдържанието на клетка от паметта

Начин на адресиране	Формат	Машинен код
Пряко в нулевата страница	DEC $..	C6
Пряко в нулевата страница с индексиране по X	DEC $..,X	D6
Пряко пълно	DEC $....	CE
Пряко с индексиране по X	DEC $...,X	DE

Променят се флаговете N Z. стр. 23

DEX

Намаляване с единица на съдържанието на индексния регистър X

Начин на адресиране	Формат	Машинен код
С подразбиране	DEX	CA

Променят се флаговете N Z. стр. 23

DEY

Намаляване с единица на съдържанието на индексния регистър Y

Начин на адресиране	Формат	Машинен код
С подразбиране	DEY	88

Променят се флаговете N Z. стр. 23

EOR

Събиране по модул 2 на битовете от съдържанието на акумулатора със съответните битове на константа или от съдържанието на клетка от паметта

Начин на адресиране	Формат	Машинен код
Непосредствено	EOR #$..	49
Пряко в нулевата страница	EOR $..	45
Пряко в нулевата страница с индексиране по X	EOR $..,X	55
Пряко пълно	EOR $....	4D
Пряко с индексиране по X	EOR $....,X	5D
Пряко с индексиране по Y	EOR $....,Y	59
Косвено с предварително индексиране по X	EOR ($..,X)	41
Косвено с последващо индексиране по Y	EOR ($..),Y	51

Променят се флаговете N Z. стр. 41

INC

Увеличаване с единица на съдържанието на клетка от паметта

Начин на адресиране	Формат	Машинен код
Пряко в нулевата страница	INC $..	E6
Пряко в нулевата страница с индексиране по X	INC $..,X	F6
Пряко пълно	INC $....	EE
Пряко с индексиране по X	INC $....,X	FE

Променят се флаговете N Z. стр. 23

INX

Увеличаване с единица на съдържанието на индексния регистър X

Начин на адресиране	Формат	Машинен код
С подразбиране	INX	E8

Променят се флаговете N Z. стр. 23

INY

Увеличаване с единица на съдържанието на индексния регистър Y

Начин на адресиране	Формат	Машинен код
С подразбиране	INY	C8

Променят се флаговете N Z. стр. 23

JMP

Безусловен преход

Начин на адресиране	Формат	Машинен код
Пряко пълно	JMP $....	4C
Косвено	JMP ($....)	6C

Флаговете не се променят. стр. 24

JSR
Преход към подпрограма

Начин на адресиране	Формат	Машинен код
Пряко пълно	JSR $....	20

Флаговете не се променят. стр. 49

LDA
Зареждане на акумулатора с константа или със съдържанието на клетка от паметта

Начин на адресиране	Формат	Машинен код
Непосредствено	LDA #$..	A9
Пряко в нулевата страница	LDA $..	A5
Пряко в нулевата страница с индексиране по X	LDA $..,X	B5
Пряко пълно	LDA $....	AD
Пряко с индексиране по X	LDA $....,X	BD
Пряко с индексиране по Y	LDA $....,Y	B9
Косвено с предварително индексиране по X	LDA ($..,X)	A1
Косвено с последващо индексиране по Y	LDA ($..),Y	B1

Променят се флаговете N Z. стр. 21

LDX
Зареждане на индексния регистър X с константа или със съдържанието на клетка от паметта

Начин на адресиране	Формат	Машинен код
Непосредствено	LDX #$..	A2
Пряко в нулевата страница	LDX $..	A6
Пряко в нулевата страница с индексиране по Y	LDX $..,Y	B6
Пряко пълно	LDX $....	AE
Пряко с индексиране по Y	LDX $....,Y	BE

Променят се флаговете N Z стр. 21

LDY
Зареждане на индексния регистър Y с константа или със съдържанието на клетка от паметта

Начин на адресиране	Формат	Машинен код
Непосредствено	LDY #$..	A0
Пряко в нулевата страница	LDY $..	A4
Пряко в нулевата страница с индексиране по X	LDY $..,X	B4
Пряко пълно	LDY $....	AC
Пряко с индексиране по X	LDY $....,X	BC

Променят се флаговете N Z. стр. 21

LSR

Изместване с един разред надясно на всички битове от съдържанието на акумулатора или клетка от паметта

Начин на адресиране	Формат	Машинен код
Акумулаторно	LSR	4A
Пряко в нулевата страница	LSR $..	46
Пряко в нулевата страница с индексиране по X	LSR $..,X	56
Пряко пълно	LSR $....	4E
Пряко с индексиране по X	LSR $....,X	5E

Променят се флаговете N Z C.

стр. 44

NOP

Празна операция

Начин на адресиране	Формат	Машинен код
С подразбиране	NOP	EA

Флаговете не се променят.

стр. 27

ORA

Логическо събиране на битовете от съдържанието на акумулатора със съответните битове на константа или от съдържанието на клетка от паметта

Начин на адресиране	Формат	Машинен код
Непосредствено	ORA #$..	09
Пряко в нулевата страница	ORA $..	05
Пряко в нулевата страница с индексиране по X	ORA $..,X	15
Пряко пълно	ORA $....	0D
Пряко с индексиране по X	ORA $....,X	1D
Пряко с индексиране по Y	ORA $....,Y	19
Косвено с предварително индексиране по X	ORA ($..,X)	01
Косвено с последващо индексиране по Y	ORA ($..),Y	11

Променят се флаговете N Z.

стр. 40

PHA

Записване на съдържанието на акумулатора в стека

Начин на адресиране	Формат	Машинен код
С подразбиране	PHA	48

Флаговете не се променят.

стр. 47

PHP
Записване на съдържанието на регистъра за състоянието на процесора в стека

Начин на адресиране	Формат	Машинен код
С подразбиране	PHP	08

Флаговете не се променят. стр. 47

PLA
Зареждане на акумулатора със стойност, извлечена от стека

Начин на адресиране	Формат	Машинен код
С подразбиране	PLA	68

Променят се флаговете N Z. стр. 47

PLP
Зареждане на регистъра за състоянието на процесора със стойност, извлечена от стека

Начин на адресиране	Формат	Машинен код
С подразбиране	PLP	28

Променят се флаговете N Z C I D V. стр. 48

ROL
Ротация наляво на битовете от съдържанието на акумулатора или клетка от паметта

Начин на адресиране	Формат	Машинен код
Акумулаторно	ROL	2A
Пряко в нулевата страница	ROL $..	26
Пряко в нулевата страница с индексиране по X	ROL $..,X	36
Пряко пълно	ROL $....	2E
Пряко с индексиране по X	ROL $....,X	3E

Променят се флаговете N Z C. стр. 45

ROR
Ротация надясно на битовете от съдържанието на акумулатора или клетка от паметта

Начин на адресиране	Формат	Машинен код
Акумулаторно	ROR	6A
Пряко в нулевата страница	ROR $..	66
Пряко в нулевата страница с индексиране по X	ROR $..,X	76
Пряко пълно	ROR $....	6E
Пряко с индексиране по X	ROR $....,X	7E

Променят се флаговете N Z C. стр 45

RTI
Връщане след прекъсване

Начин на адресиране	Формат	Машинен код
С подразбиране	RTI	40

Променят се флаговете N Z C I D V. стр 57

RTS
Връщане от подпрограма

Начин на адресиране	Формат	Машинен код
С подразбиране	RTS	60

Флаговете не се променят. стр. 49

SBC
Изваждане от съдържанието на акумулатора на константа или на съдържанието на клетка от паметта или константа и заема

Начин на адресиране	Формат	Машинен код
Непосредствено	SBC #$..	E9
Пряко в нулевата страница	SBC $..	E5
Пряко в нулевата страница с индексиране по X	SBC $..,X	F5
Пряко пълно	SBC $....	ED
Пряко с индексиране по X	SBC $....,X	FD
Пряко с индексиране по Y	SBC $....,Y	F9
Косвено с предварително индексиране по X	SBC ($..,X)	E1
Косвено с последващо индексиране по Y	SBC ($..),Y	F1

Променят се флаговете N Z C V. стр. 30

SEC
Установяване на флага за пренос

Начин на адресиране	Формат	Машинен код
С подразбиране	SEC	38

Променя се флагът C. стр. 24

SED
Установяване на флага за десетичен режим

Начин на адресиране	Формат	Машинен код
С подразбиране	SED	F8

Променя се флагът D. стр. 24

SEI
Установяване на флага за прекъсване

Начин на адресиране	Формат	Машинен код
С подразбиране	SEI	78

Променя се флагът I. стр. 24

STA
Записване на съдържанието на акумулатора в клетка от паметта

Начин на адресиране	Формат	Машинен код
Пряко в нулевата страница	STA $..	85
Пряко в нулевата страница с индексиране по X	STA $..,X	95
Пряко пълно	STA $....	8D
Пряко с индексиране по X	STA $....,X	9D
Пряко с индексиране по Y	STA $....,Y	99
Косвено с предварително индексиране по X	STA ($..,X)	81
Косвено с последващо индексиране по Y	STA ($..),Y	91

Флаговете не се променят. стр. 21

STX
Записване на съдържанието на индексния регистър X в клетка от паметта

Начин на адресиране	Формат	Машинен код
Пряко в нулевата страница	STX $..	86
Пряко в нулевата страница с индексиране по Y	STX $..,Y	96
Пряко пълно	STX $....	8E

Флаговете не се променят. стр. 21

STY
Записване на съдържанието на индексния регистър Y в клетка от паметта

Начин на адресиране	Формат	Машинен код
Пряко в нулевата страница	STY $..	84
Пряко в нулевата страница с индексиране по X	STY $..,X	94
Пряко пълно	STY $....	8C

Флаговете не се променят. стр. 21

TAX
Прехвърляне на съдържанието на акумулатора в индексния регистър X

Начин на адресиране	Формат	Машинен код
С подразбиране	TAX	AA

Променят се флаговете N Z. стр. 22

TAY

Прехвърляне на съдържанието на акумулатора в индексния регистър Y

Начин на адресиране	Формат	Машинен код
С подразбиране	TAY	A8

Променят се флаговете N Z. стр. 22

TSX

Прехвърляне на съдържанието на указателя на стека в индексния регистър X

Начин на адресиране	Формат	Машинен код
С подразбиране	TSX	BA

Променят се флаговете N Z. стр. 22

TXA

Прехвърляне на съдържанието на индексния регистър X в акумулатора

Начин на адресиране	Формат	Машинен код
С подразбиране	TXA	8A

Променят се флаговете N Z. стр. 22

TXS

Прехвърляне на съдържанието на индексния регистър X в указателя на стека

Начин на адресиране	Формат	Машинен код
С подразбиране	TXS	9A

Флаговете не се променят. стр. 22

TYA

Прехвърляне на съдържанието на индексния регистър Y в акумулатора

Начин на адресиране	Формат	Машинен код
С подразбиране	TYA	98

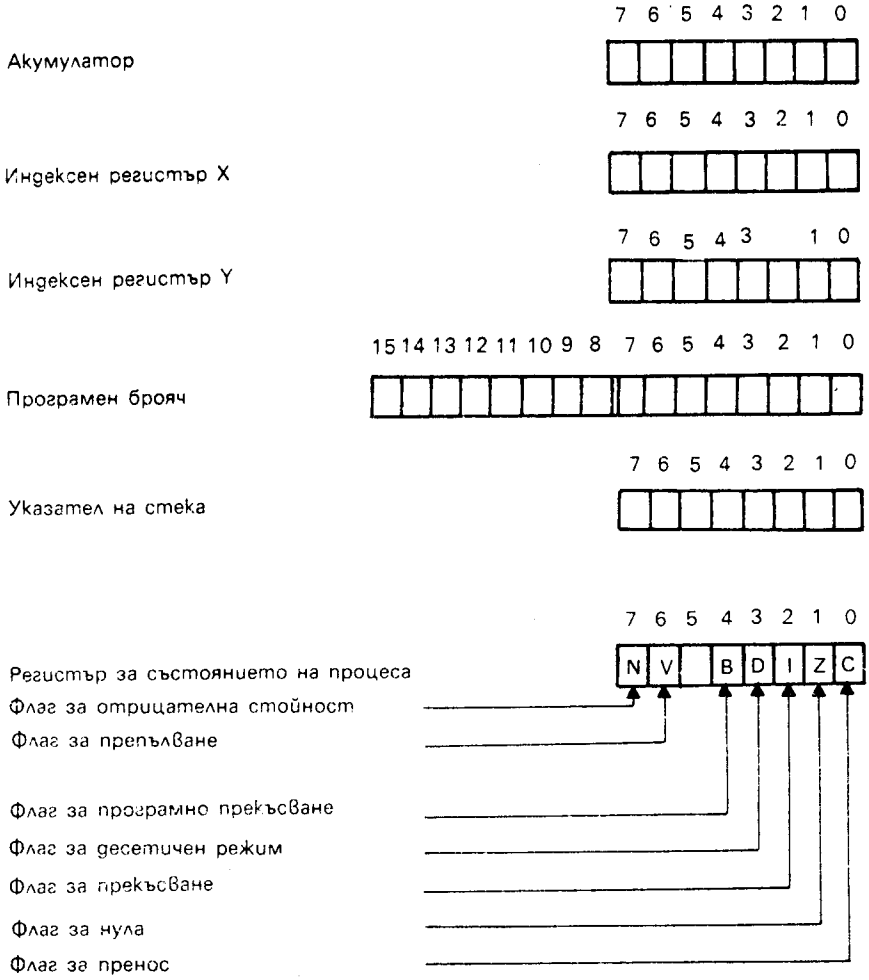
Променят се флаговете N Z. стр. 22

Шестнайсетични кодове на операциите в микропроцесора 6502

00—BRK	20—JSR $...	40—RTI	60—RTS
01—ORA ($.., X)	21—AND ($..,X)	41—EOR ($..,X)	61—ADC ($..,X)
02—???	22—???	42—???	62—???
03—???	23—???	43—???	63—???
04—???	24—BIT $..	44—???	64—???
05—ORA	25—AND $..	45—EOR $..	65—ADC $..
06—ASL $..	26—ROL $..	46—LSR $..	66—ROR $..
07—???	27—???	47—???	67—???
08—PHP	28—PLP	48—PHA	68—PLA
09—ORA #$..	29—AND #$..	49—EOR #$..	69—ADC #$..
0A—ASL	2A—ROL	4A—LSR	6A—ROR
0B—???	2B—???	4B—???	6B—???
0C—???	2C—BIT $....	4C—JMP $....	6C—JMP ($....)
0D—ORA $....	2D—AND $....	4D—EOR $....	6D—ADC $....
0E—ASL $....	2E—ROL $....	4E—LSR $....	6E—ROR $....
0F—???	2F—???	4F—???	6F—???
10—BPL $..	30—BMI $..	50—BVC $..	70—BVS $..
11—ORA ($..),Y	31—AND ($..),Y	51—EOR ($..),Y	71—ADC ($..),Y
12—???	32—???	52—???	72—???
13—???	33—???	53—???	73—???
14—???	34—???	54—???	74—???
15—ORA $..,X	35—AND $..,X	55—EOR $..,X	75—ADC $..,X
16—ASL $..,X	36—ROL $..,X	56—LSR $..,X	76—ROR $..,X
17—???	37—???	57—???	77—???
18—CLC	38—SEC	58—CLI	78—SEI
19—ORA $....,Y	39—AND $....,Y	59—EOR $....,Y	79—ADC $....,Y
1A—???	3A—???	5A—???	7A—???
1B—???	3B—???	5B—???	7B—???
1C—???	3C—???	5C—???	7C—???
1D—ORA $....,X	3D—AND $....,X	5D—EOR $....,X	7D—ADC $....,X
1E—ASL $....,X	3E—ROL $....,X	5E—LSR $....,X	7E—ROR $....,X
1F—???	3F—???	5F—???	7F—???

80—???	A0—LDY #$..	C0—CPY #$..	E0—CPX #$
81—STA ($..,X)	A1—LDA ($..,X)	C1—CMP ($..,X)	E1—SBC ($..,X)
82—???	A2—LDX #$..	C2—???	E2—???
83—???	A3—???	C3—???	E3—???
84—STY $..	A4—LDY $..	C4—CPY $..	E4—CPX $..
85—STA $..	A5—LDA $..	C5—CMP $..	E5—SBC $..
86—STX $..	A6—LDX $..	C6—DEC $..	E6—INC $..
87—???	A7—???	C7—???	E7—???
88—DEY	A8—TAY	C8—INY	E8—INX
89—???	A9—LDA #$..	C9—CMP #$..	E9—SBC #$..
8A—TXA	AA—TAX	CA—DEX	EA—NOP
8B—???	AB—???	CB—???	EB—???
8C—STY $....	AC—LDY $....	CC—CPY $....	EC—CPX $....
8D—STA $....	AD—LDA $....	CD—CMP $....	ED—SBC $....
8E—STX $....	AE—LDX $....	CE—DEC $....	EE—INC $....
8F—???	AF—???	CF—???	EF—???
90—BCC $..	B0—BCS $..	D0—BNE $..	F0—BEQ $..
91—STA ($..),Y	B1—LDA ($..),Y	D1—CMP ($..),Y	F1—SBC ($..),Y
92—???	B2—???	D2—???	F2—???
93—???	B3—???	D3—???	F3—???
94—STY $..,X	B4—LDY $..,X	D4—???	F4—???
95—STA $..,X	B5—LDA $..,X	D5—CMP $..,X	F5—SBC $..,X
96—STX $..,Y	B6—LDX $..,Y	D6—DEC $..,X	F6—INC $..,X
97—???	B7—???	D7—???	F7—???
98—TYA	B8—CLV	D8—CLD	F8—SED
99—STA $....,Y	B9—LDA $....,Y	D9—CMP $....,Y	F9—SBC $....,Y
9A—TXS	BA—TSX	DA—???	FA—???
9B—???	BB—???	DB—???	FB—???
9C—???	BC—LDY $....,X	DC—???	FC—???
9D—STA $....,X	BD—LDA $....,X	DD—CMP $....,X	FD—SBC $....,X
9E—???	BE—LDX $....,Y	DE—DEC $....,X	FE—INC $....,X
9F—???	BF—???	DF—???	FF—???

Програмно достъпни регистри на микропроцесора 6502



Асемблер-редактор MERLYN

1. Команди за входно-изходни действия

C:CATALOG — Показва каталога на дискетата. След тази команда могат да се въвеждат и други команди към ДОС.

L:LOAD — Зарежда в паметта първичната асемблерска програма, записана на дискета като двоичен файл. При всяко следващо използуване на тази команда се изписва последното използувано име на файл, което се потвърждава с „Y" или не се приема, ако се натисне друг клавиш, и се въвежда ново име. Разширението „суфиксът" „.S" не се въвежда, а се подразбира. След изпълнение на тази команда се преминава към режим на редактиране.

S:SAVE — Записва въведената първична асемблерска програма като двоичен файл на дискетата. Името на файла се задава както при командата LOAD и към него автоматично се прибавя разширение „.S".

A:APPEND — Зарежда от дискетата файл с първична програма и го разполага на края на първичната асемблерска програма в паметта. Името на файла се задава както при командата LOAD.

R:READ — Зарежда от дискетата първична асемблерска програма, записана като текстов файл. Разширението „.T" не се въвежда, но се подразбира.

W:WRITE — Записва на дискетата първичната асемблерска програма като текстов файл. Името на файла се задава както при командата LOAD и към него автоматично се прибавя разширение „.T".

D:DRIVE — Сменя номера на дисковото устройство, с което се работи, от 1 на 2 или от 2 на 1.

E:EDITOR — Преминава към режим на въвеждане и редактиране на асемблерската програма.

O:OBJECT — Записва асемблираната програма като двоичен файл на дискетата. Името на файла се задава както при командата SAVE, но не му се прибавя разширение „.S".

Q:QUIT — Край на работата с асемблер-редактора с преход към интерпретатора на БЕЙСИК. Асемблер-редакторът може да се стартира отново с командата ASSEM.

2. Команди за работа с асемблер-редактора

2.1. Общи команди

HI — Установява горна граница за първичната програма и начало на обектния код. Ако не се зададе, се подразбира $8000.

NEW — Изтрива първичната асемблерска програма в паметта и установява горната граница на $8000.

PR# — Избор на изходно устройство (както в БЕЙСИК). Обикновено се използуват PR#1 и PR#0 съответно за включване и изключване на печатащото устройство.

USER — Осъществява преход чрез JSR $3F5. Използува се за връзка с потребителски управляващи (драйверни) програми на машинен език.

TABS — Установява ширината на отделните полета в програмните редове при въвеждането на първичната асемблерска програма.

LEN — Изчислява и изписва дължините в байтове на първичната асемблерска програма и на наличната свободна памет до установената горна граница.

W — Изчислява и изписва в шестнайсетичен вид адреса в паметта, от който се разполага даден програмен ред. W0 дава адреса на края на първичната асемблерска програма.

MON — Преминава в режим монитор. Връщането в асемблер-редактора може да се стане с МК—Y.

TRON — По време на изпълнение на LIST или PRINT спира отпечатването на програмните редове до първия срещнат интервал, следван от знака „;".

TROFF — Прекратява действието на TRON.

Q — Край на въвеждането и/или асемблирането на първичната програма и преход към командите за входно-изходни действия.

ASM — Асемблира въведената първична асемблерска програма до обектен (непосредствено изпълним) код. Най-напред се изписва надпис с въпрос, дали са необходими допълнителни поправки. При отговор „Y" се изписва първият ред от първичната програма, съдържащ знака „/", и се преминава в режим на редактиране. При отговор „N" се преминава към асемблиране на първичната програма. На екрана се изписват шестнайсетичните адреси и намиращите се в тях машинни инструкции. След това се изписва таблица на използуваните символични имена (етикети и променливи) по азбучен и числов ред. При наличие на грешки се изписват съобщения за вида им. Асемблирането може да бъде прекратено с МК—Ц. При натискане на интервал изписването продължава ред по ред, а при натискане на RETURN — продължава без прекъсване. При натискане на МК—Д асемблирането се изпълнява без изписване на таблиците.

D — Изтрива един или няколко програмни реда от първичната асемблерска програма и преномерира останалите.

R — Изтрива един или няколко реда от първичната асемблерска програма, изписва поредния

номер на първия от тях и преминава в режим на редактиране за въвеждане на нови редове на тяхно място.

L — Извежда на екрана един или няколко програмни реда. Контролните знаци се изписват като инверсни. При натискане на интервал из[illegible] късване. Изписването може да бъде прекратено с МК—Ц или „/".

. — Извежда на екрана програмните редове от [illegible] първичната програма.

/ — Извежда на екрана програмните редове, [illegible] първичната програма.

P — Изпълнява се точно както L, но не изписва номерата на програмните редове.

PRTR — Извежда един или няколко програмни реда на печатащото устройство.

F — Търси и извежда на екрана всички програмни редове, съдържащи даден низ.

C — Замества в един, няколко или всички програмни редове един низ с друг.

COPY..TO.. — [illegible] съдържанието [illegible] програмен ред [illegible]

MOVE..TO.. — [illegible] програмен ред в друг, като изтрива оригинала.

E — Извежда на екрана номера на един или няколко посочени програмни реда и преминава в режим на редактирането им.

TEXT — Замества всички знаци за интервал в първичната програма с инверсни знаци за интервал. Използува се за запомняне на табулацията при записване на първичната програма като текстов файл. Няма видимо действие върху изписването на екрана.

FIX — Замества всички инверсни знаци за интервал с нормални знаци за интервал и изпълнява някои проверки по първичната програма. Превключва се използуването в преди записване от първичната програма като двоичен файл.

SYM — Указва началния адрес в паметта, от който се разполага таблицата със символичните имена. Ако не се използува, се подразбира адресът $D000 от DRAM.

VID — Използува се за включване или изключване на 80-колонна платка.

FW — Търси и изписва на екрана всички програмни редове, съдържащи даден низ, който е отделен със знаци, различни от цифра или буква.

CW — Замена във всички програмни редове даден низ, отделен със знаци, различни от цифра или буква, с друг низ.

EW — Изписва номерата на програмните редове, съдържащи даден низ, отделен със знаци

различни от цифра или буква, и преминава в режим на редактирането им.

VAL — Търси в първичната програма, изчислява и изписва стойността на изрази, така както се изчислява от асемблера. Знакът „ ^ " може да се използува за означаване на променливата част от израза. Така например F,P ^ R" ще открие, както израза „PRINTER", така и „POINTER".

Към общите команди се отнася и превръщането на числата от десетични в шестнайсетични и обратно. При въвеждане на шестнайсетично число, означено със знака $, и натискане на RETURN се извежда неговият десетичен вид. При въвеждане на десетично число и натискане на RETURN се извежда неговият шестнайсетичен вид. Допустимите граници на числата са от $0000 до $FFFF (съответно от 0 до 65355 за числа без знак и от −128 до +127 за числа със знак).

2.2. Команди за добавяне и вмъкване на програмни редове

A — Включва режим на добавяне с автоматично номериране на програмните редове. От този режим се излиза с въвеждане на RETURN като първи знак на програмния ред, или въвеждане на МК—Ц или МК—Ь където да е в реда.

I — Влиза в режим на вмъкване непосредствено след указания ред. Номерирането на вмъкваните програмни редове става автоматично, а редовете след тях се преномерират. От този режим се излиза с въвеждане на RETURN като първи знак на програмния ред, МК—Ц или МК—Ь.

МК—Л — Превключва от кирилица на латиница и обратно.

2.3. Команди за редактиране на програмен ред

МК—И — Вмъква следващите въведени знаци в програмния ред.

МК—Д — Изтрива знака под показалеца.

МК—Ф — Поставя показалеца върху следващия знак в реда, еднакъв с въведения след МК—Ф знак.

МК—О — Вмъква следващия въведен знак като контролен. В комбинация с определени знаци вмъква в програмния ред знаци, които не могат да бъдат въведени пряко от клавиатурата.

МК—П — Въвежда програмен ред от 32 знака „'".

МК—Ю — Въвежда програмен ред от 30 интервала, ограден със знаци „'".

МК—Ц или МК—Ь — Прекратява редактирането на програмния ред, който запазва първоначалния си вид.

МК—В — Поставя показалеца в началото на програмния ред.

МК—Н — Поставя показалеца една позиция вдясно от края на програмния ред.

МК—Р — Възстановява първоначалния вид на програмния ред от преди редактирането.

МК—Я — Изтрива знаците от показалеца до края на програмния ред.

RETURN — Потвърждава въведения програмен ред и преминава към въвеждане на следващия. Ако е въведен като първи знак в програмния ред, прекратява въвеждането и/или редактирането на първичната асемблерска програма.

3. Формат на данните, изрази и променливи

3.1. Формат на данните

За означаване на типа на числата в асемблер-редактора MERLYN се използват знаците:
— за десетично число;
#$ — за шестнайсетично число;
#% — за двоично число.
Числата, предшествувани от знака „#", се възприемат като данни, а тези без него — като адреси.

Всеки програмен ред се състои от следните полета:

ЕТИКЕТ МНЕМОНИЧЕН КОД НА ОПЕРАНД ;КОМЕНТАР
ОПЕРАЦИЯТА

Програмен ред, който съдържа само коментар, трябва да започва със знак „*". Коментарите се отделят със знак „;". Максималната дължина на полето за етикети е 13 знака. Първият знак от етикетите задължително трябва да бъде знак, чийто ASCII-код е по-голям или равен на този на знака „:", и не трябва да съдържа знак с ASCII-код, по-малък от този на знака „0". Максималната обща дължина на полетата за мнемоничния код на операцията и операнда е 64 знака.

3.2. Изрази

Асемблер-редакторът MERLYN позволява работа с аритметични и логически изрази. В тях могат да се използват:
— етикети;
— десетични числа;
— шестнайсетични числа;
— двоични числа;
— ASCII-кодове на знаци, оградени с кавички (за установен седми бит) или с апострофи (за изчистен седми бит);
— знак „*" за означаване на текущия адрес.

Разрешени са действията:
+ аритметично събиране;
— изваждане;
* умножение;
/ деление;
! логическо изключващо ИЛИ;
. логическо умножение;
& логическо събиране.

Не се допуска използуването на скоби. Изразите се изчисляват не по приоритета на операциите, а последователно отляво надясно.

При въвеждане на двубайтови операнди могат да се използуват знаците „<" — за отделяне само на младшия байт и „>" или „/" — за отделяне само на старшия байт на операнда.

3.3. Променливи

Символичните имена, започващи със знака „]", се възприемат от асемблер-редактора MERLYN като променливи и имената им не могат да се използуват по-нататък в програмата като етикети. Стойностите им се определят първоначално, а след това могат и да се променят с директивата EQU.

В асемблер-редактора са включени и 8 системни променливи с имена]1,]2,]3,]4,]5,]6,]7 и]8. Те са предназначени главно за работа с макроси.

4. Асемблерски директиви

EQU	— Определя адреса на етикет или променлива. Вместо EQU може да се използува знакът „=".
ORG	— Определя началния адрес, от който ще се зарежда и стартира асемблираната програма. Ако не се използува, се подразбира начален адрес $8000.
OBJ	— Определя началния адрес, от който се разполага обектният код по време на асемблирането.
PUT	— По време на асемблирането зарежда от дискетата текстов файл с указаното име и разширение „T" и подменя директивата в първичната програма със съдържанието на файла. Зареденият текстов файл не трябва да използува макроопределения, както и команди PUT.
VAR	— Задава стойност на системните променливи]1,]2,]3 и т. н. до]8. Използува се за предаване на формалните параметри при команда PUT.
SAV	Записва на дискета обектния код като двоичен файл с указаното име. Може да се използува многократно в първичната програма за запис на обектния код на части.
DSK	Указва резултатите от асемблирането на първичната програма да се записват пряко на диска като двоичен файл с указаното име. Използува се за асемблиране на много големи програми.
END	— Спира асемблирането на останалата след него част от първичната програма. Етикетите след END не се разпознават в процеса на асемблиране.
LST ON/OFF	— Включва и изключва извеждането на информацията за процеса на асемблиране към екрана и/или печатащото устройство.
EXP ON OFF	Указва дали по време на асемблирането да се изписва цялото тяло на макроопределенията или те да се означават съкратено с PMC и името им.
PAU	— Указва втората част от асемблирането да се осъществи след натискане на клавиш.
PAG	— Указва извеждането на информацията на печатащото устройство да продължи на нова страница.
SKP	Включва празен ред при изписване на първичната програма.
TRON TROFF	Включва и изключва ограничението до три байта на ред при изписване на обектния код.
ASC	Включва в обектния код на мястото на директивата ASCII-кодовете на знаците от низ. Въвежданият низ трябва да бъде ограден с кавички или апострофи. В първия случай ASCII-кодовете ще бъдат въведени с изчистен седми бит, а във втория — седмият бит ще бъде установен.

DCI — Изпълнява се като ASC с тази разлика, че последният знак от низа се въвежда със седми бит, противоположен на тези на седмите битове на останалите знаци в низа.

INV — Изпълнява се като ASC, но знаците от низа се въвеждат като инверсни.

FLS — Изпълнява се като ASC, но знаците от низа се въвеждат като мигащи.

REV — Изпълнява се като ASC, но знаците от низа се въвеждат в обратен ред (отдясно наляво).

DA — Включва в обектния код на мястото на директивата едно или повече двубайтови шестнайсетични числа, като първо се записва съответният младши и след него старшият байт.

DDB — Изпълнява се като DA, но първо се записва съответният старши и след него младшият байт.

DFB — Включва в обектния код на мястото на директивата няколко числа, зададени в десетичен, шестнайсетичен или двоичен вид, или като израз.

HEX — Включва в обектния код на мястото на директивата едно или няколко двуцифрени шестнайсетични числа, които се задават без знак „$".

DS — Запазва определен брой байтове свободна памет в обектния код.

KBD — Позволява въвеждането на стойност на даден етикет пряко от клавиатурата по време на асемблирането.

LUP — Задава начало на цикъл и брой на повторенията. Броят на повторенията трябва да бъде от $0001 до $8000.

—^ — Задава край на цикъл, определен с директивата LUP.

CHK — Включва в обектния код байт, съдържащ контролна сума. Обикновено това е последният байт от програмата. Не може да се използува заедно с директивата DSK.

ERR — Предизвиква прекъсване и извеждане на съобщение от вида „BREAK IN LINE ???" при получаване на ненулева стойност след изчисляване на израза, зададен в адресната част на директивата.

USR — Осъществява преход чрез инструкция JSR $B6DA при стойност в акумулатора нула, изчистен регистър Y и установен флаг за пренос. Използува се за връзка с потребителски програми на машинен език. В нормалния случай адресът $B6DA съдържа инструкцията RTS.

DO — Използува се заедно с директивите ELSE и FIN и служи за условно асемблиране на части от първичната програма. Ако в процеса на асемблиране след изчисляване на израза, използуван като операнд на директивата, се получи нула, генерирането на обектен код се

прекратява до следващата директива ELSE или FIN. В такъв случай символичните имена в тази част от програмата, с изключение на макроопределенията, не се разпознават. Ако стойността на операнда е ненулева, асемблирането продължава по нормалния начин.

ELSE — Определя началото на частта от първичната програма, която трябва да се асемблира, ако стойността на операнда в последната директива DO е нула.

FIN — Определя края на частта от първичната програма, зададена с директивата DO. Всяка директива DO трябва да е последвана в първичната програма от директива FIN и всяка директива FIN трябва да е предшествана от директива DO. Директивата ELSE може да се появява само след DO преди FIN. Всяко разклонение, образувано с DO и FIN (евентуално включващо и ELSE), може от своя страна да бъде разклонявано, като нивата на разклоняване не трябва да са повече от 8.

MAC — Задава име и определя начало на макроопределение.

EOM или < < < — Определя край на макроопределение.

PMC или > > > — Включва в обектния код на мястото на директивата инструкциите, съставящи тялото на макроопределението с посоченото име.

ЛИТЕРАТУРА

1. Ангелов, А., П.Петров. Микропроцесорът — сърцето на микрокомпютъра. С., Техника, 1986.
2. Вачков, Б., М. Христов. Как работи Правец-82. С., Техника, 1986.
3. Точи, Р., Л. Ласковски. Микропроцесори и микрокомпютри. С., Техника. 1978
4. Шишков, А., И. Маранезов. Работа с персонален компютър. С., Техника, 1986.
5. APPLE II Reference Manual. Cupertino, APPLE Computer, Inc. 1984.
6. Bredon, G.MERLYN — Instruction Manual. Santee, California, Southwestern Data Systems, 1982.
7. Chandor, A. The Penguin Dictionary of Microprocessors, London, Penguin Books, 1983.
8. De Jong, M. J. Programming & Interfacing 6502. London, Howard W. Sam & Co., Inc., 1980.
9. Haskel, R. C. 6502 Assembly Language Tutor, London, Prentice — Hall Inc., 1983.
10. Hyde, R. Using 6502 Assembly Language. Reston, Reston Publishing Company, Inc., 1982.
11. Inman, D., K. Inman. APPLE Machine Language. Reston, Reston Publishing Company, Inc., 1981.
12. Levental, L. A. 6502 Assembly Language Programming. New York, Osborne/MC Graw Hill, 1978.
13. Luebert, W. F. What's Where in The APPLE. New York, Micro Inc, Inc., 1982.
14. MC56500 Microcomputer Family Programming Manual. Noristown, MOS Technology, Inc. 1976.
15. Zaks, R. Programming the 6502, New York, Sybex, 1978.

СЪДЪРЖАНИЕ

ПЕРСОНАЛЕН КОМПЮТЪР ПРАВЕЦ-82
ПРОГРАМИРАНЕ НА АСЕМБЛЕР

Автор АНТОАН ИОРДАНОВ ХЛЕБАРОВ
Рецензенти: инж. АЛЕКСИ СТЕФАНОВ БОЮКЛИЕВ
доц. к. м. н. ДИМИТЪР ПЕТРОВ ШИШКОВ
к. т. н. инж. МИШЕЛ ЛЮСИЕН АВРАМОВ

Българска
Първо издание

Код 03 95331/3207—8—88
Изд. № 15476

Научен редактор доц. к. м. н. ДИМИТЪР ШИШКОВ
Редактор на издателството инж. РУМЯНА КОВАЧЕВА
Художник ЛЮБОМИР МИХАЙЛОВ
Художествен редактор ДОСЮ ДОСЕВ
Технически редактор ЦВЕТАНА ПОПОВСКА
Коректор ЖАНА ДЕЛЧЕВА

Дадена за набор на 30.VI. 1987 г.
Подписана за печат м. юли 1988 г.
Излязла от печат м. юли 1988 г.
Формат 60 × 90/16
Печ. коли 7
Изд. коли 7
УИК 8.02
Тираж 30 000 + 111
Цена 1,12 лв.

Държавно издателство „ТЕХНИКА", бул. Руски 6, София
Държавна печатница „Г. ДИМИТРОВ", София

1,12 тв.