**Атанас Атанасов**

# Ръководство за упражнения по

# МИКРОПРОЦЕСОРНА ТЕХНИКА

**ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ**

София, 2012 г.

Настоящото ръководство по микропроцесорна техника е предназначено за студенти от III курс (редовно) и IV курс (задочно) обучение по специалност „Автоматизация и изчислителна техника” в Химикотехнологичния и металургичен университет – София.

Ръководството може да се ползва и от всички, които се интересуват от основите на микропроцесорната техника и програмирането на асемблер за 32-битови CISC и RISC микропроцесори.

В първата част на ръководството (глави 1 – 4) са представени аритметичните и логически основи на микропроцесорната техника, както и аритметичните операции над двоични и шестнадесетични числа и такива с плаваща запетая. Разгледани са видовете инструкции и адресации на съвременните CISC и RISC микропроцесори.

Във втората част на ръководството (глави 5 – 11) са разгледани програмният модел, видовете инструкции и адресации на 32-битовия CISC микропроцесор Motorola MC68000, както и разработката на линейни, разклонени, циклични и сложни програми на асемблер за този процесор чрез специализирана симулационна среда Sim68K.

В третата част на ръководството (глави 12 – 14) е разгледан програмният модел, видовете инструкции и адресации на 32-битовата RISC фамилия ARM, както и разработката на програми на асемблер за процесори ARM4 – ARM7 чрез развойната среда ARM Project Manager.

Ръководство за упражнения по
**МИКРОПРОЦЕСОРНА ТЕХНИКА**

Автор: Атанас Атанасов
Рецензент: доц. д-р Андрей Мирев
Коректор: д-р Данка Апостолова

Предпечатна подготовка: УКЦ при ХТМУ – София
Печат: УПД 39 при НИС на ХТМУ – София
Издател: ХТМУ – София

ISBN 978-954-465-054-4

# СЪДЪРЖАНИЕ

1. Аритметични основи на микропроцесорните системи ..... 55

2. Логически основи и представяне на информацията в микропроцесорните системи ..... 12

3. Аритметични операции с цели числа и с числа, представени с фиксирана и плаваща запетая ..... 17

4. Изпълнение на инструкциите в микропроцесор-ните системи ..... 25

5. Програмиране на асемблер за процесор Motorola MC68000 ..... 35

6. Видове адресации на процесора MC68000 ..... 45

7. Симулационна среда за програмиране на асемблер за MC68000 ..... 56

8. Разработка на линейни програми на асемблер за MC68000. ..... 62

9. Разработка на разклоненни програми на асемблер за MC68000. ..... 66

10. Разработка на циклични и сложни програми на асемблер за MC68000 ..... 73

11. Разработка на подпрограми за MC68000 ..... 81

12. Програмиране на асемблер за процесор ARM ..... 89

13. Разработка на линейни и разклонени програми на асемблер за процесор ARM. ..... 98

14. Разработка на циклични и сложни програми на асемблер за процесор ARM. ..... 108

Литература ..... 113

# 1. Аритметични основи на микропроцесорните системи

**Бройни системи**

Бройната система е начин за представяне на числата с помощта на набор от символи, имащи определени количествени значения. Тези символи се наричат цифри.

За представяне на числата различните бройни системи използват различен набор от цифри. Освен цифрите, бройните системи включват и правила за представяне на числата с цифри.

**Видове бройни системи**

Съществуват два вида бройни системи – позиционни и непозиционни. В позиционните бройни системи всяка цифра има определено тегло, зависещо от позицията на цифрата в числото.

Броят на цифрите, които съдържа бройна система, се нарича основа на бройната система.

В непозиционните бройни системи броят и позициите на цифрите в числото не определят неговата големина.

Следните два примера илюстрират десетичната (арабска) бройна система, която е позиционна бройна система, и римската, която е непозиционна бройна система.

В десетичната бройна система използваните цифри са: 0,1,2,3,4,5,6,7,8 и 9, а основата й е 10.

**Пример 1.1.** $5187 = 5.10^3 + 1.10^2 + 8.10^1 + 7.10^0$

**Пример 1.2.** $3{,}46 = 3.10^0 + 4.10^{-1} + 6.10^{-2}$

При непозиционната римска бройна система се използват вместо цифри латинските букви I, V, X, L, C, D, M и т.н. Техните еквиваленти в десетична бройна система са дадени по-долу, както и примери за представяне на числата 8 и 9 (VIII и IX), от които е видно, че по-голямото число се представя с по-малко цифри (символи).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Римски числа** | **I,** | **V,** | **X,** | **L,** | **C,** | **D,** | **M** |
| **Десетични числа** | **1** | **5** | **10** | **50** | **100** | **500** | **1000** |

**Пример 1.3.** $VIII = 5 + 1 + 1 + 1 = 8_{10}$

**Пример 1.4.** $IX = 10 - 1 = 9_{10}$

**Двоична бройна система**

При тази бройна система се използват цифрите 0 и 1, а основата й е 2. Тази система е най-разпространена за представяне на числова информация в компютрите, понеже на двете й цифри – нула и единица – може да се съпоставят състояния (отпушено/запушено) на основни електронни елементи като диоди, транзистори и кондензатори. Последните са градивни елементи на микропроцесорите и паметите на компютрите.

**Пример 1.5.** $1011012 = 1.2^5 + 0.2^4 + 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0$
$= 32 + 0 + 8 + 4 + 0 + 1 = 45$

**Пример 1.6.** $1{,}0112 = 1.2^0 + 0.2^{-1} + 1.2^{-2} + 1.2^{-3}$
$= 1 + 0. + 1/4 + 1/8 = 1{,}625$

**Преобразуване на двоично число в десетично:**

Ако събираемите в горните представяния се изчислят и се сумират, ще се получи десетичният еквивалент на двоичното число.

**Забележка:** Цифрите 0 и 1 се използват и в десетичната бройна система. За да е ясно в каква бройна система се записва числото, обикновено основата се записва като индекс към числото, например $1101001_2$ е двоично. Само в десетичната бройна система този индекс се изпуска, т.е. 1101001 е десетично.

**Преобразуване на цялата част на десетично число в двоично**

Цялата част на десетичното число се дели на основата на новата бройна система (2), като се записват целочислените остатъци (0 или 1) от деленето, по-малки от основата на новата бройна система. Деленето продължава докато се достигне до частно по-малко от основата на новата бройна система, което се записва като последен остатък. Остатъците от деленето, записани в обратен ред дават записа на числото в новата бройна система.

**Пример 1.7.** Преобразуване на 79 от десетична в двоична бройна система.

| | Число в десетична БС | Основа на новата БС | Остатък от деленето |
|---|---|---|---|
| число | 79 | :2 | 1 |
| частно | 39 | :2 | 1 |
| частно | 19 | :2 | 1 |
| частно | 9 | :2 | 1 |
| частно | 4 | :2 | 0 |
| частно | 2 | :2 | 0 |
| частно | 1 | :2 | 1 |

$1001111_{(2)} = 79_{(10)}$

**Преобразуване на дробната част на десетично число в двоично**

Дробната част на десетичното число се умножава с основата на новата бройна система (2). Получава се нова дробна част или цяла част и дробна част, като цялата част е единица (1). Получената дробна част отново се умножава по основата на новата бройна система. Умножението продължава, докато се получи нова дробна част, равна на 0. Получените цели части при умножението представляват дробната част на преобразуваното число. Те се записват в реда, в който са получени.

**Забележка:** При преобразуване на десетични дробни числа в двоични се задава точност на пресмятането, тъй-като рядко се получава дробна част 0.

**Пример 1.8.** Преобразуване на дробната част 0,625 от десетична в двоична бройна система.

| | Число в десетична БС | Основа на новата БС | Цяла част | Дробна част |
|---|---|---|---|---|
| число | 0,625 | * 2 | 1 | 0,250 |
| Др. част | 0,250 | * 2 | 0 | 0,500 |
| Др. част | 0,500 | * 2 | 1 | 0,000 |

$0{,}101_{(2)} = 0{,}625_{(10)}$

Ако в примера вместо 0.625 използваме 0,626 или 0,627, умножението би продължило много повече от три пъти (три разряда на двоичното число), като в някои случаи се получават периодични двоични дроби и никога не се получава 0. За да се избегне това, числото в новата бройна система се представя с определен брой разряди, а в някои случаи се закръглява на по-голямото. Например периодичната дроб 0,10011100111001 1 може да се представи или като 0,10011, или 0,100111, или като 0,1010 със закръгление нагоре.

**Шестнадесетична бройна система**

**Цифри: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F**

**Основа: 16**

**Пример 1.9.** $3FD16 = 3.16^2 + 15.16^1 + 13.16^0 = 768 + 240 + 13 = 1021$

**Пример 1.10.** $C,8A16 = 12.16^0 + 8.16^{-1} + 10.16^{-2} = 12 + 8/16 + 10/256 = 12{,}5390625$

**Забележка:** Буквените комбинации от A до F заместват двуцифрените десетичните числа от 10 до 15. Това се прави, за да се избегне двусмислие при записа на шестнадесетичните числа.

**Преобразуване на шестнадесетично число в десетично:**

Ако събираемите в горните представяния се изчислят и се сумират, ще се получи десетичният еквивалент на двоичното число.

**Преобразуване на цяло десетично число в шестнадесетично**

Прилага се общото правило за преобразуване на десетично число в произволна бройна система**.**

**Пример 1.11.** По-долу е дадено преобразуването на числото 299 от десетична в шестнадесетична бройна система.

| | Число в десетична БС | Основа на новата БС | Остатък от деленето |
|---|---|---|---|
| число | 299 | :16 | 11 - B |
| частно | 18 | :16 | 2 |
| частно | 1 | :16 | 1 |

$12B_{(16)} = 299_{(10)}$

**Осмична бройна система**

**Цифри: 0,1,2,3,4,5,6,7**

**Основа: 8**

**Пример 1.12.** $2568 = 3.8^2 + 5.8^1 + 6.8^0 = 128 + 40 + 6 = 174$

**Пример 1.13.** $7{,}428 = 7.8^0 + 4.8^{-1} + 4.8^{-2} = 7 + 4/8 + 2/64 = 7{,}53125$

**Преобразуване на числата от двоична в осмична и шестнадесетична бройна система и обратно.**

**Пример 1.14** В таблица 1.1 по-долу са показани в първата колона шестнадесетичните числа от 0 до F ($16_{10}$), във втората колона техните еквивалентни осмични числа, в третата колона двоичните числа, а в последната колона десетичният им еквивалент.

**Таблица 1.1.** Преобразуване между различните бройни системи

| Шестнадесетично число | Осмично число | Двоично число | Десетично число |
|---|---|---|---|
| 0 | 0 | 0000 | 0 |
| 1 | 1 | 0001 | 1 |
| 2 | 2 | 0010 | 2 |
| 3 | 3 | 0011 | 3 |
| 4 | 4 | 0100 | 4 |
| 5 | 5 | 0101 | 5 |
| 6 | 6 | 0110 | 6 |
| 7 | 7 | 0111 | 7 |
| 8 | 10 | 1000 | 8 |
| 9 | 11 | 1001 | 9 |
| A | 12 | 1010 | 10 |
| B | 13 | 1011 | 11 |
| C | 14 | 1100 | 12 |
| D | 15 | 1101 | 13 |
| E | 16 | 1110 | 14 |
| F | 17 | 1111 | 15 |

**Преобразуване на числата от двоична в осмична и шестнадесетична бройна система и обратно**

При преобразуване на двоично число в осмично всяка тройка двоични цифри се заменя със съответстващата й осмична цифра. При обратното преобразуване всяка осмична цифра трябва да се замести със съответна тройка двоични цифри.

**Пример 1.14.** $\underline{110}\ \underline{011}\ \underline{111}_2$

$\quad\ \ 6 \quad 3 \quad 7_8$

**Пример 1.15.** $5 \quad 0 \quad 6 \quad 7_8$

$\underline{101}\ \underline{000}\ \underline{110}\ \underline{111}_2$

**Преобразуване на числата от двоична в шестнадесетична бройна система и обратно**

При преобразуване на двоично число в шестнадесетично всяка четворка двоични цифри се заменя със съответстващата й шестнадесетична цифра. При обратното преобразуване всяка шестнадесетична цифра трябва да се замести със съответна четворка двоични цифри.

**Пример 1.16.** $\underline{1010}\ \underline{1011}\ \underline{0111}_2$

$\quad\ \ A \quad B \quad 7_{16}$

**Пример 1.17.** $F \quad 0 \quad 6 \quad 7_{16}$

$\underline{1111}\ \underline{0000}\ \underline{0110}\ \underline{0111}_2$

**Аритметични операции над числа в двоична и шестнадесетична системи**

Следващата таблица 1.2 показва аритметичните операции събиране, изваждане и умножение на двоични числа и събиране и изваждане на шестнадесетични числа, както и правилата за възникване на пренос (+1) или заем (-1) към старшия или от младшия разряд.

**Таблица 1.2**

| | |
|---|---|
| **Двоична аритметика Събиране:** | |
| **0 + 0 = 0  0 + 1 = 1**<br>**1 + 0 = 1  1 + 1 = 0  и пренос 1 към по-старшия бит** | |
| **Пример в двоична бр. с-ма** | **Пример в десетична бр. с-ма** |
| **10  пренос 1 към по старшия бит**<br>**100101**<br>**+ 10110**<br>**----------**<br>**111011** | **37**<br>**+ 22**<br>**----------**<br>**59** |
| **Двоична аритметика Изваждане:** | |
| **0 - 0 = 0  1 - 0 = 1**<br>**1 - 1 = 0  0 - 1 = 1 и заем 1 от по-старшия бит** | |
| **Пример в двоична бр. с-ма** | **Пример в десетична бр. с-ма** |
| **1  и заем 1 от по-старшия бит**<br>**101101**<br>**- 10100**<br>**----------**<br>**11001** | **45**<br>**- 20**<br>**----------**<br>**25** |
| **Двоична аритметика Умножение:** | |
| **0 * 0 = 0  0 * 1 = 0**<br>**1 * 0 = 0  1 * 1 = 1** | |
| **Пример в двоична бр. с-ма** | **Пример в десетична бр. с-ма** |
| **1001**<br>*** 11**<br>**+ 1001**<br>**1001**<br>**----------**<br>**11011** | **9**<br>*** 3**<br>**----------**<br>**27** |

| | |
|---|---|
| **Шестнадесетична аритметика** | |
| **Събиране:**<br>Ако сумата на две числа надвиши 16 в младшия разряд се записва разликата от резултата и 16 и възниква пренос +1 към по-старшия разряд. Например:<br>F + 3 = 2 и пренос 1 към по-старшия разряд – резултат 12 | |
| **Пример в шестнадесетична бр. с-ма** | **Пример в десетична бр. с-ма** |
| **1  пренос 1**<br>**2C**<br>**+1A**<br>**---------**<br>**46** | **44**<br>**+ 26**<br>**----------**<br>**70** |
| **Изваждане:**<br>Ако от по-малко число се изважда по-голямо, към умаляемото се добавя 16 и от получения резултат се изважда умалителят, като възниква заем -1 от по-старшия разряд. | |
| **Пример в шестнадесетична бр. с-ма** | **Пример в десетична бр. с-ма** |
| **-1  и заем -1 бит**<br>**31**<br>**- 14**<br>**----------**<br>**1D** | **49**<br>**- 20**<br>**----------**<br>**29** |

**Задачи:**

1. Преобразувайте следните десетични числа в двоични:
   а) 123 б) 277 в) 654 г) 0.625 д) 89.125
2. Преобразувайте следните двоични числа в десетични:
   а) 110001101 б) 1010110111
   в) 11101110011 г) 101110.011 д) 10011.1101
3. Кое от следните две двоични числа е по-голямото?
   а) 1100111111 или 1011111101
   б) 1011111111 или 1100000001
   в) 1101010101 или 1110001111
4. Преобразувайте следните десетични числа в шестнадесетични:
   а) 623 б) 547 в) 1653 г) 321 д) 296
5. Преобразувайте следните шестнадесетични числа в десетични:
   а) 7A08 б) B3C7 в) F16 г) ACDC д) 9ECA
6. Кое от следните две шестнадесетични числа е по-голямото?
   а) BAF0 или CFF3 б) AB0F или BEDA
   в) CEC0 или CAF7
7. Преобразувайте следните шестнадесетични числа в двоични и в осмични:
   а) 71A08 б) F0B3C7 в) D3F16 г) AE03B2 д) 1209E
8. Преобразувайте следните двоични числа в шестнадесетични и в осмични:
   а) 101100111011100101
   б) 111010101111011001
   в) 110100001001110110
9. Съберете следните двоични числа:
   а) 1010101 + 110011
   б) 111010101 + 1000101
   в) 1001110111 + 10011111
10. Извадете следните двоични числа:
   а) 100000011 - 1111
   б) 101011001-101111
   в) 100111000111-11100110011
11. Умножете следните двоични числа:
   а) 10011 * 101 б) 1110111 * 10101 в) 11011 * 111
12. Съберете следните шестнадесетични числа:
   а) AB09 + 47F б) 12BE1 + 39FC в) 7056 + FCED
13. Извадете следните шестнадесетични числа:
   а) 123A1 - FBD б) BED9-ABBA в) 2DEDO-1BABA
14. Защо и как в микропроцесорните системи се използват двоичната и шестнадесетичната бройни системи?
15. Имат ли всички цели и реални десетични числа точно представяне в двоичната бройна система?

# 2. Логически основи на микропроцесорните системи и представяне информацията в тях

**Логически променливи**

В микропроцесорните системи се използват логически схеми и устройства, реализиращи различни логически операции. Работата им се базира на формален математически апарат, който се нарича логическа алгебра или булева алгебра, носеща името на ирландския математик Джордж Бул (1815 – 1864г.). Булевата алгебра се занимава с обекти, наречени логически променливи. Логическите променливи могат да приемат две стойности – истина (True) или лъжа (False), на които в компютрите се съпоставят **0** или **1.**

Логическите променливи и логическите константи може да участват в логически изрази. За формиране на изразите се използват логически операции, които свързват логическите променливи.

На логическите променливи може да се съпоставят определени съждения, които ако са верни, то логическите променливи имат стойност "1", а ако са грешни – "0".

**Логически операции**

Основните логически операции са: конюнкция, дизюнкция и отрицание. Има и други, като изключваща дизюнкция или изключваща конюнкция, равнозначност и т.н.

**Конюнкция**

Конюнкцията (логическо умножение) е операция, при която две (A, B) или повече логически променливи се свързват с логическата връзка **И**. Резултатът от конюнкцията (виж таблица 2.1) е равен на нула (0), когато поне един от аргументите й има стойност нула. Той е равен на единица (1), когато всички аргументи са равни на единица. Конюнкцията се означава със знака ^ или с ***AND***, например **A *AND* B** или **A ^ B**.

**Дизюнкция**

Дизюнкцията е операция, при която две (A,B) или повече логически променливи се свързват с логическата връзка **ИЛИ.** Резултатът от дизюнкцията (виж таблица 2.1) е равен на единица (1), когато поне един от аргументите й има стойност 1, и е равен на нула (0), когато всички аргументи са равни на 0. Дизюнкцията се означава със знака $\vee$ или с ***OR***, например

**A *OR* B** или **A $\vee$ B**.

**Таблица 2.1. Таблици за истинност за конюнкция и дизюнкция**

| A | B | A ^ B | A \/ B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**Отрицание**

Логическо отрицание е операцията, при която се получава нова логическа променлива със стойност, обратна на първата променлива (виж таблица 2.2). Отрицанието има един аргумент и променя стойността му от 1 в 0 или обратно - от 0 в 1. Срещат се различни варианти на означаване на тази логическа операция: ***!, NOT,*** ¬ .

**Таблица 2.2. Таблица за истинност за отрицание**

| A | ¬A |
|---|---|
| 0 | 1 |
| 1 | 0 |

**Приоритет на логическите операции**

Най-висок приоритет има отрицанието, следвано от конюнкцията и дизюнкцията. Ако операциите имат еднакъв приоритет, то те се изпълняват отляво надясно.

**Закони на Де Морган:**

Законите на Де Морган са свързани с отрицанието на конюнкцията/дизюнкцията на две или повече логически променливи. Те гласят:

1. Отрицанието на конюнкцията на две или повече логически променливи е равно на дизюнкцията от отрицанието на отделните логически променливи.

2. Отрицанието на дизюнкцията на две или повече логически променливи е равно на конюнкцията от отрицанието на отделните логически променливи.

Законите се използват за оптимизиране на логически изрази и при синтеза на електронни компоненти, реализиращи определени логически изрази.

***NOT*** **(A** ***AND*** **B) = (*NOT* A)** ***OR*** **(*NOT* B),**

**<=> <u>A ^ B</u> = <u>A</u> *V* <u>B</u>,**

***NOT*** **(A** ***OR*** **B) =** ***NOT*** **A** ***AND NOT*** **B,**

**<=> <u>A *V* B</u> = <u>A</u> ^ <u>B</u>.**

**Таблица 2.3. Логически електронни схеми, реализиращи логически операции.**

| | | | | | |
|---|---|---|---|---|---|
| AND | A<br>B | | A<br>B | & | A ^ B |
| OR | A<br>B | | A<br>B | ≥1 | A ∨ B |
| NOT | A | | A | 1 | ¬ A |
| NAND | A<br>B | | A<br>B | & | ¬(A ^ B) |
| NOR | A<br>B | | A<br>B | ≥1 | ¬(A ∨B) |

## Представяне на информацията в микропроцесорните системи

Бит е най-малкото количество информация, която може да се разглежда като отговор на въпрос, който има два възможни отговора – ***да*** или ***не*** (***true*** или ***false***, 1/0 – едноцифрено, двоично число).

Информацията, съдържаща се в отговора на такъв въпрос, се нарича бит (bit).

Двоичната информация се групира в множество от битове, наречени байтове (B – byte). Един байт (B – byte) се състои от 8 бита.

Производни единици на байта са: килобайт (1 KB = 1024 B), мегабайт (1 MB = 1024 KB), гигабайт (1 GB = 1024 MB) и т.н. Производните единици KB, MB, GB, TB (терабайт) се различават от стандартните измервателни единици – вместо множител 1000 се използва $2^{10}$ = 1024 (множителят е кратен на основата на двоичната бройна система). По този начин най-икономично се използва паметта в микропроцесорните системи.

## Представяне на символна информация в микропроцесорните системи

В микропроцесорните системи информацията се съхранява само в двоичен вид. Използваните символи също се представят като комбинация от двоични числа.

При натискане на клавиш от клавиатурата се формира определен код наречен *скан код* (*scan code*). Този код не е свързан със символа на клавиша, а се определя от поредния номер на клавиша. Връзката между *скан кода* и символа се определя от специална таблица. В нея на всеки

символ, използван в компютърната система, се присвоява двоично число (байт или 2 байта).

Така текстът, записан в паметта на компютъра, представлява последователност от байтове, съответстващи на символите от текста.

В микропроцесорните системи най-често се използват ASCII и Unicode таблиците. Те са се превърнали в стандарт, чрез който може да се обменя информация между различни компютърни системи.

**ASCII (American Standard Code for Information Interchange)**

В тази таблица (система) буквите от латинската азбука, цифрите, препинателните знаци и други специфични символи, като #, $, %, и някои специални клавишни комбинации са кодирани с числата от 0 до 127. Кодовете с номера над 128 се използват за други символи и азбуки, например за буквите на кирилицата. Поради това някои текстове, написани на кирилица, може да изглеждат странно, когато се смени шрифтът.

**Unicode**

Поради нарастващата нужда от повече символи се въвежда системата за кодиране Unicode. С нея работят съвременните компютри и операционни системи. При тази система, за представянето на един символ се използва не един байт (8 бита), а два байта (16 бита). Това означава, че в Unicode могат да се кодират до $2^{16} = 65536$ различни символа. Първият байт на всеки символ от кирилицата, например в тази система съдържа числото 204, като код на азбуката.

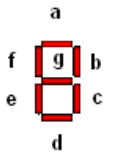
**Седемсегментен код**

Седемсегментния код (фиг. 2.1) се използва за визуализация на десетични и шестнадесетични числа на специализирани индикатори и дисплеи.

**Фигура 2.1. Десетцифров седемсегментен дисплей**

За да се визуализира дадена цифра или буква на седемсегментния индикатор, е необходимо да се активизират (осветят) определен набор от сегменти. Сегментите се асоциират с първите 7 бита на даден байт, като на най-младшя бит се съпоставя латинската буква **a**, а на 6-тия бит буквата **g**, както е дадено в таблица 2.5 по-долу.

**Таблица 2.5. Битове и сегменти в седемсегментния код**

| | **g** | **f** | **e** | **d** | **c** | **b** | **a** | |
|---|---|---|---|---|---|---|---|---|
| Бит 7 | Бит 6 | Бит 5 | Бит 4 | Бит 3 | Бит 2 | Бит 1 | Бит 0 |  |

Например, за да се изпише числото едно, е необходимо да се активизират битове 1 и 2 (букви b и c) или кодова комбинация 00000110, а за да се изпише числото осем, е необходимо да се активизират всички битове, освен най-старшият – 01111111.

В някои дисплеи най-старшият бит се асоциира с десетична точка, изписвана в долния десен ъгъл на седемсегментния индикатор.

**Задачи:**

1. Напишете седемсегментните кодове на числата от 0 до 9.
2. Напишете седемсегментните кодове на буквите, използвани за запис на числата в шестнадесетичен код ( A- F).
3. Какви други букви и символи могат да се визуализират чрез седемсегментен код? Дайте примери.
4. С колко бита в двоична бройна система се записват числата 10, 100 и1000?
5. Колко байта има в един мегабайт и в един терабайт?
6. Напишете логически израз, който да е истина, ако страните A, B и C са страни на триъгълник.
7. Напишете логически израз, който да е истина, ако страните A, B и C са страни на равнобедрен триъгълник.
8. Напишете логически израз, който да е истина, ако страните A, B и C са са страни на равностранен триъгълник.
9. Напишете логически израз, който да е истина, ако страните A, B и C са са страни на правоъгълен триъгълник.
10. Напишете логически израз, който да е истина, ако променливата X принадлежи на интервала X1 - X2, включително и за стойности, равни на X1 и X2.
11. Напишете логически израз, който да е истина, ако променливата X не принадлежи на интервала X1 - X2, включително и за стойности, равни на X1 и X2.

# 3. Аритметични операции с цели числа и с числа, представени с фиксирана и плаваща запетая

**Представяне на числова информацията в микропроцесорните системи**

В микропроцесорните системи се използват основно два вида числа: цели (натурални) числа и реални (веществени) числа.

**Цели числа без знак**

Целите неотрицателни числа се представят в двоична форма чрез преобразуване по начина, показан по-горе в раздела бройни системи. В зависимост от големината на числото за представянето му са необходими различен брой цифри (битове).

В микропроцесорните системи числата се представят с фиксиран брой битове. Или числата се представят с определена точност.

Броят на целите числа, които могат да се запишат с помощта на n двоични цифри, е $2^n$. Тогава целите неотрицателни числа, описани с n бита, са в интервала $0 \div 2^{n-1}$. Нулата (0) е една от възможните комбинации. Съответно максималната стойност е с 1 по-малка от $2^n$.

При целите неотрицателни числа всички битове съдържат цифри, описващи числото. В такъв случай с един байт (8 бита) може да се представи цяло неотрицателно число в интервала $0 \div 2^8 - 1$ или $0 \div 255$.

**Цели числа със знак**

Представянето на тези числа изисква въвеждане на специален бит за знака на числото. Приема се, че ако в бита за знак има стойност 0, числото е положително, а ако стойността е 1, то е отрицателно.

Така например, ако за цяло число се отделя 1 байт, стойността на числото се записва в последните 7 бита, а най-старшият (най-левият) бит се използва за знак. Тогава максималното цяло число със знак, което може да се запише в един байт, е $2^7 = 128$. Диапазонът на възможните числа е от -127 до +128, тъй като се включва и нулата.

**Пример 3.1.** Положително число 37 **$00100101_2$**

**Пример 3.2.** Отрицателно число -37 **$10100101_2$**

**Допълнителен код**

Отрицателните числа се представят в допълнителен код, който се получава от правия код на числото чрез инвертиране (обратен код) и добавяне на единица.

Правият код на числото е самото число, представено в двоична бройна система, като най-старшият му бит (този най-вляво) е знаковият бит.

**Пример 3.3.** **$- 5_{10} =$ $00000101_2$ прав код на 5**
**Инвертиране $11111010_2$ обратен код на 5**
**Прибавяне на 1 + 1 $11111011_2$ допълнителен код 5**

Чрез допълнителния код операциите по изваждане, например A = B - C, се свеждат до събиране на числа в прав и допълнителен код A = B + (-C), където B е в прав код, а (-C) е в допълнителен код.

**Дробни числа**

За да се представят дробни числа в двоична бройна система, се използва позиционна запетая. В микропроцесорните системи дробните числа се представят по два начина – с фиксирана запетая и с плаваща запетая.

- **Дробни числа с фиксирана запетая**

Мястото на позиционната запетая в полето, в което се записват числата е фиксирано. В битовете преди позиционната запетая се записва цялата част, а в битовете след нея – дробната част на числото.

**Пример 3.4.** цяла част, дробна част
**$1001110001,10101_2$**

- **Дробни числа с плаваща запетая**

Позицията на запетаята в полето, в което се записват числата, не е фиксирано. Числата с плаваща запетая се дефинират с мантиса (дробна част) M и порядък (експонента) R.

Едно десетично число N може да се представи като: **$N = M * 10^R$.**

**Пример 3.5.** **-374.25** може да се запише във вида:
**$-3.7425x10^2$.**
В този запис **M = -3.7425,** а **R = 2.**

**Пример 3.6.** **0.000453** може да се запише като**:**
**$4.53x10^{-4}$** , където
мантисата е **M = 4.53,**
а порядъкът е **R = -4.**

Двоичните числа с плаваща запетая се представят в паметта на компютърните системи, като двоичното число се разделя на две части – едната за запис на мантисата, а другата – за порядъка. Мантисата и порядъкът се записват като числа със знак.

**Пример 3.7.** **$01100000 1011_2$**
мантисата е 1 байт, а порядъкът 4 бита
или **$M = 01100000_2$,**
а **R = 1011**
и тогава **$N = 96 \times 2^{-5} = 3$**

Числата, които имат цифри пред десетичната запетая или числата, по-малки от нула, които имат нули след запетаята, се наричат ненормализирани.

Например $123{,}456 \times 10^2$ или $0{,}00789 \times 10^2$ са ненормализирани.

За тяхната нормализация се налага изместване на цялата част надясно, докато числото се представи с водеща нула, запетая и значими разряди след запетаята ($0{,}123456 \times 10^5$), като при всяко преместване надясно порядъкът на числото се увеличава с единица. При нормализация на числа, по-малки от единица, мантисата се премества наляво, а порядъкът (експонентата) се намалява с единица. Преместването продължава, докато се появи водеща цифра след запетаята.

Нормализацията на двоични числа следва същото правило.

**Пример 3.8.** След нормализация на числото от предишния пример (**$M = 01100000_2$** и **$R = 1011_2$ )** мантисата се премества 7 пъти на надясно, а експонентата се увеличва със седем и се получава:

**$M = 0{,}11000000$ и $R = 0010$**,

и тогава **$N = 1/2 + 1/4 \times 2^{-5+7} = 3/4 \times 2^2 = 3$**

**IEEE 754 стандарт за числа с плаваща запетая**

Този стандарт се използва за представяне на дробни числа в съвременните микропроцесорни с-ми, използващи Intel процесори, както и в Apple MacX и Unix платформите. Ако знаковият бит е нула (0), то числото е положително, а ако е единица (1) е отрицателно. Реалната експонента се формира от експонента минус отместване. Например: (200 - 127 = 73).

В таблица 3.1 по-долу са дадени значенията на битовете (големината на експонентата, на мантисата и на отместването) при запис на 32-битови и 64-битови числа.

**Таблица 3.1. IEEE 754 стандарт за числа с плаваща запетая**

| Точност | Знаков бит | Експонента | Дробна част | отместване |
|---|---|---|---|---|
| Единична 32 бита | (1бит) бит 31 | (8 бита) 30-23 | (23 бита) [22-0] | 127 |
| Двойна 64 бита | (1бит) бит 63 | (11 бита) 62-52 | (52 бита) [51-0] | 1023 |

Долната таблица 3.2 показва диапазона на числата, които могат да се запишат, ако се използва единичната или двойна точност (32 или 64 бита) на този стандарт. В някои разновидности на този стандарт се поддържат 80-битово или 124-битово представяне на числата.

**Таблица 3.2. Максимално и минимално число с единична и двойна точност в стандарта IEEE 754**

| | Денормализирано | Нормализирано | Приблизително нормализирано |
|---|---|---|---|
| Единична точност | $\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$ | $\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$ | $\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$ |
| Двойна точност | $\pm 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$ | $\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$ | $\pm \sim 10^{-323.3}$ to $\sim 10^{308}$ |

**Аритметични операции над числа с плаваща запетая**

Аритметичните операции с двоични числа, представени с плаваща запетая, са аналогични на тези с десетични числа.

При операцията умножение се умножават мантисите на числата, а порядъците (експонентите) им се събират.

При деленето се делят мантисите на числата, а порядъците им се изваждат.

След умножение или деление е необходима проверка, за да се установи дали резултантният порядък не е по-голям от максималния (за умножението) или по-малък от минималния (за деленето). В такива случаи е необходимо експонентата (порядъкът) да бъде установена на максимална или минимална стойност, а мантисата– на някакво допустимо число. Това е особено важно в системите за управление, в които трябва да се подаде съобщение за грешка от препълване или загуба на порядък, като се прекратят последващите изчисления.

Понеже повечето съвременни микропроцесори поддържат инструкциите за умножение и деление на цели числа, то реализацията на тези операции за числа с плаваща запетая не изисква големи усилия за програмирането им.

Интерес представляват операциите събиране и изваждане на числата, представени с плаваща запетая.

При събиране или изваждане на две числа с плаваща запетая е необходимо сравнение на експонентите на числата, понеже може да се събират или изваждат само мантисите на числа с еднакви експоненти.

Например, ако едното число е $5 * 10^{+2}$ (500), а другото е $5 * 10^{-3}$ (0.005), мантисите им M1 = 5 и M2 = 5 не могат да бъдат събрани директно, защото експонентите им R1 = 2 и R2 = -3 са различни.

Ако експонентите на числата се различават, е необходимо да се определи кое число е с по-голямата експонента и неговата мантиса да се премести |R1-R2| пъти надясно, като по този начин експонентите се изравнят и едва тогава мантисите на двете числа да се съберат или извадят.

Изместването на по-голямата мантиса наляво и намаляването на експонентата на дадено число не са препоръчителни, защото е възможно да се загубят значими разряди от числото. При измествне на по-малката мантиса надясно и увеличаване на експонентата могат да се загубят най-младшите разряди на числото.

**Пример 3.9.** Дадено е едно число с мантиса M от 16 разряда (например $1101100111110_2$) и експонента E = 0.

Нека разгледаме двата случая с преместване на мантисата надясно и наляво с по 5 разряда.

Преди преместването (таблица 3.3) на мантисата надясно числото е имало десетична стойност $1101100111110_2 = 6974_{10}$.

**Таблица 3.3. Числото преди преместване на мантисата му**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **0** |

След преместването (таблица 3.4) на мантисата му 5 пъти надясно и увеличаване на експонентата му с 5 ще се получи числото $11011001_2 * 2^5$, което в десетичен еквивалент е $6944_{10}$

**Таблица 3.4. Числото след преместване на мантисата му надясно**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **0** | **0** | **1** |

Загубата на последните му 5 най-младши двоични разряди води до загуба на точност в десетиците и единиците (общо 30).

Ако извършим преместване (таблица 3.5) на мантисата на същото число 5 пъти наляво и намаляване на експонентата на число с 5, ще се получи:

$0110011111000000_2 * 2^{-5} = 26560_{10} / 32 = 830_{10}$.

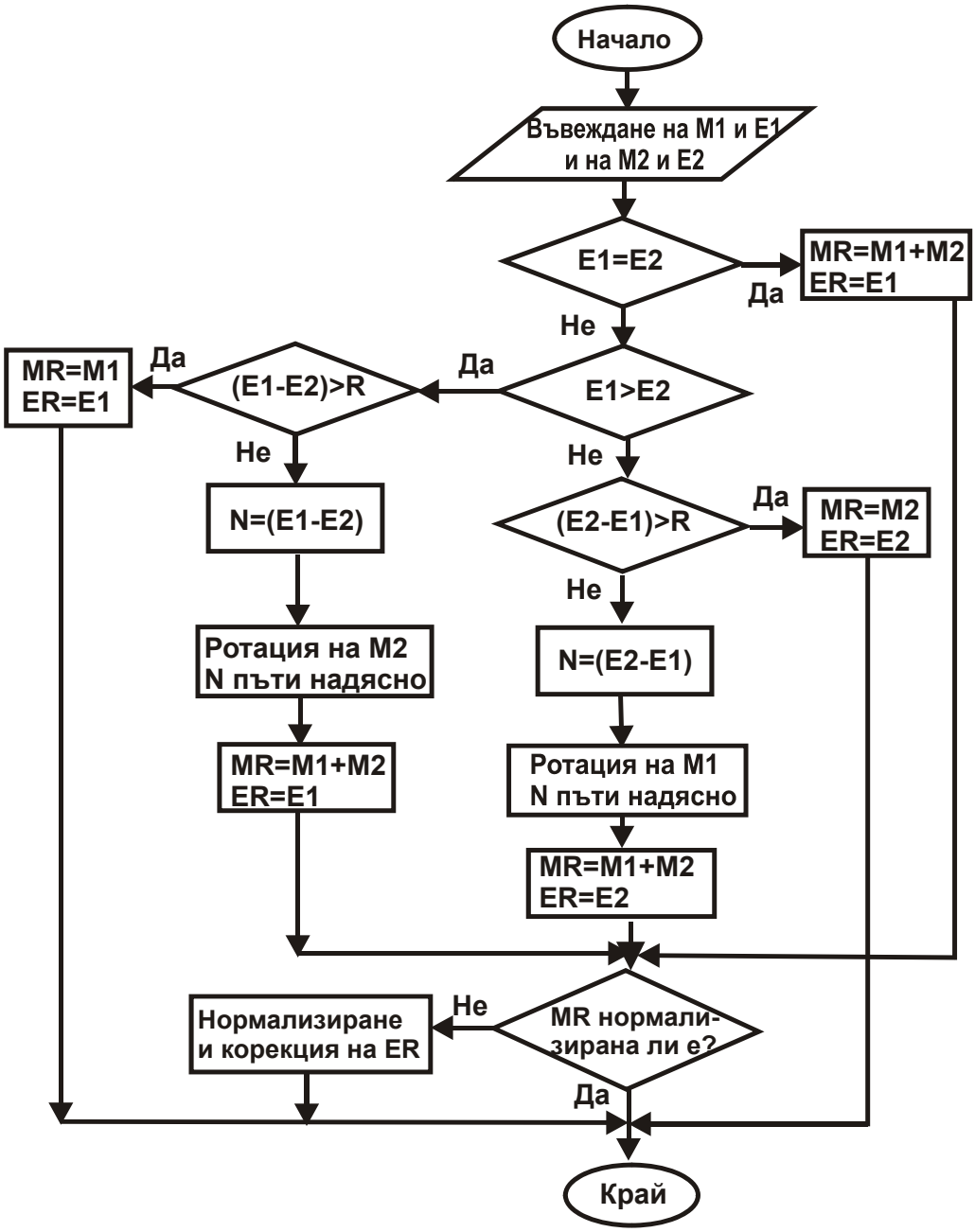
**Таблица 3.5. Числото след преместване на мантисата му наляво**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **1** | **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |

Вижда се, че в този случай загубата на точност е значителна и вместо 6974 или даже 6944 се получава 830, като са загубени значими разряди в стотиците и хилядите. Затова при изравняване на мантисите на две числа се прилага преместване надясно на мантисата на по-малкото число и увеличаване на експонентата му.

Честото преместване на мантисите на числата надясно и наляво (при нормализацията им) води до загуба на точност – проблем, дискутиран в подробности от дисциплината „Числени методи”.

Следният алгоритъм описва събирането на две числа (виж фиг. 3.1), като отчита и разрядността на мантисата им.




**Фигура 3.1**. Блок-схема за събиране на числа с плаваща запетая

Алгоритъмът включва следните стъпки:

1. Ако експонентите на двете числа са равни (E1 = E2), то мантисите им се събират MR = M1 + M2.

2. Ако експонентите са различни, се определя коя е по-голямата от тях.

3. От по-голямата се изважда по малката експонента и се проверява дали разликата между тях е по-голяма от броя на разрядите, предвидени за мантисите на числата. Очевидно е, че ако разликата между експонентите на числата (N = |E1-E2| ) е по-голяма от броя на разрядите P, предвидени за мантисата, резултатът ще е по-голямото число, защото мантисата на по-малкото число ще се нулира след N (N = |E1-E2| ) на брой премествания надясно (N > P) .

4. Ако разликата между експонентите на двете числа е по-малка от броя на разрядите на мантисата, тогава мантисата на по-малкото число се измества надясно N на брой пъти, като по този начин експонентата му се изравнява с тази на по-голямото число. Едва тогава мантисите на числата могат да бъдат събрани.

5. Ако след събирането мантисата на резултата се е денормализирала, то същата се налага да се нормализира, като се премести надясно, а експонентата на резултата се намали с единица.

Алгоритъмът за изваждане на две числа е аналогичен на горния, като разликата е в това, че мантисите на числата се изваждат и ако се наложинормализация на резултантната мантиса, то обикновено се налага изместването й един или няколко пъти наляво и корекция на експонентата й.

**Задачи:**

1. Представете следните числа в 10-битов допълнителен код:

а) -56 б) -341 в) -179 г) -488

2. Извършете следните операции, като представите отрицателните числа в 8-битов допълнителен код:

а) 97 - 49 б) 35 - 76 в) -78 - 31 г) -27 + 91

3. Представете следните числа като двоични числа с фиксирана запетая с 16-битов код, от тях 4 бита за дробна част:

а) 56.125 б) 363.875 в) 129.52 г) 419.8 д) 266.675

4. Представете следните числа като двоични числа с плаваща запетая с 16-битова мантиса и 8-битова експонента със знак:

а) 245 б) 734 в) 129

г) -319 д) 207.675 е) 524.125

5. Нормализирайте следните ненормализирани двоични числа с плаваща запетая във формат мантиса 16 бита и експонента 8 бита:

а) M = 1011101,01100,  R = 00000011

б) M = 100010111,101100,  R = 00010011

в) M = 10111011,01100,  R = 10000011

г) M = 10000011,00001,  R = 10000111

д) M = 0,000101110101100,  R= 00000011

6 . Обяснете защо, ако разликата между експонентите на две числа е по-голяма от разрядите на мантисата, резултатът от събирането на числата е равен на по-голямото число.

7. Представете следните числа като двоични числа с плаваща запетая с 10-битова мантиса и 8-битова експонента със знак:

а) 25  б) 79  в) 122

г) -319  д) 107.675  е) 124.125

8. Кои са най-голямото и най-малкото ненормализирани числа, които могат да се запишат чрез мантиса от 10 бита и експонента от 8 бита?

9. Кои са най-голямото и най-малкото ненормализирани числа, които могат да се запишат чрез мантиса от 16 бита и експонента от 8 бита?

10. Кои са най-голямото и най-малкото ненормализирани числа, които могат да се запишат чрез мантиса от 12 бита и експонента от 5 бита?

11. Ще има ли загуба на точност при представянето на следните числа с плаваща запетая във формат 8 бита мантиса и 5 бита експонента:

а) 12.126  б) 121.13  в) 25  г) 56.125  д) 29

# 4. Изпълнение на инструкциите в микропроцесорните системи

**Формат на инструкциите**

Инструкциите в микропроцесорните системи се състоят от две части:

- Код на операцията (КОП) и
- Операнд или операнди (ОП).

**КОП ОП** или **КОП ОП1, ОП2** или **КОП ОП1, ОП2, ОП**3

Полето на кода на операцията определя броя (вида) на инструкциите и вида на (адресацията) им.

Видът на инстукцията определя какво действие ще се извърши над операнда или операндите на инструкцията.

Адресацията определя как да се интерпретира операндът или операндите. Операндът или операндите, над които се извършва инструкцията, могат да са числа, адреси на клетки от паметта или регистри на микропроцесора.

Известни са няколко формата на инструкциите (таблица 4.1):

**Таблица 4.1.** Формат на инструкциите

| | |
|---|---|
| •без операнд | КОП |
| •с един операнд | КОП \| ОП |
| •с два операнда | КОП \| ОП1 \| ОП2 |
| •с три операнда | КОП \| ОП1 \| ОП2 \| ОП3 |

Полето на операндите е определящо за големината (дължината) на инструкцията. В повечето съвременни 32 или 64-битови микропроцесори единият операнд може да е адрес от паметта, а останалите операнди са регистри на процесора или числа. Това е така, защото адресите, с които работят микропроцесорите, са 32-битови. Ако два или три операнда са 32-битови, това означава дължината на инструкцията да е по-голяма от 10 или 12 байта, което забавя процесора при нейното зареждане/извличане от паметта.

**Изпълнение на инструкциите**

Изпълнението на инстукциите във всички микропроцесорни системи от Фон-Нойманов тип включва два етапа:

- извличане на инструкцията от паметта (instruction fetching);
- и дешифриране и изпълненние на инструкцията (instruction execution).

**Извличане на инструкцията**

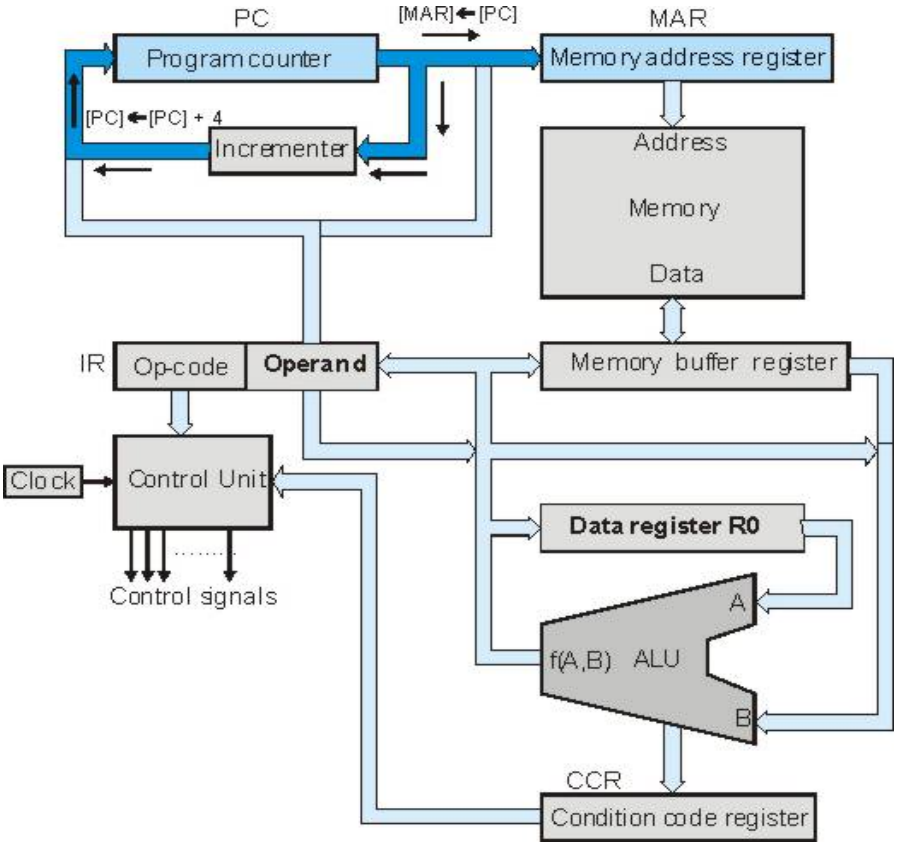
Извличането на инструкцията от паметта на компютъра (фиг. 4.1 и фиг. 4.2) протича по следния начин:

1. Зареждане на адреса на инструкцията в програмния брояч /ПБ/ (program counter /PC/) на микропроцесора.

2. Зареждане на адреса на инструкцията от програмния брояч в адресния регистър /АР/ (memory address register /MAR/) на микропроцесора. Подаване на сигнал за четене от паметта. Обновяване на ПБ, така че той да сочи следващата инстукция от програмата (ПБ = ПБ + 4).

3. Прочитане (извличане) на инструкцията от RAM паметта и зареждането й в буферния регистър на паметта /БРП/ (memory buffer register /MBR/).

4. Преместване на инструкцията от буферния регистър на паметта в информационния регистър /ИР/ на микропроцесора (Information register /IR/).

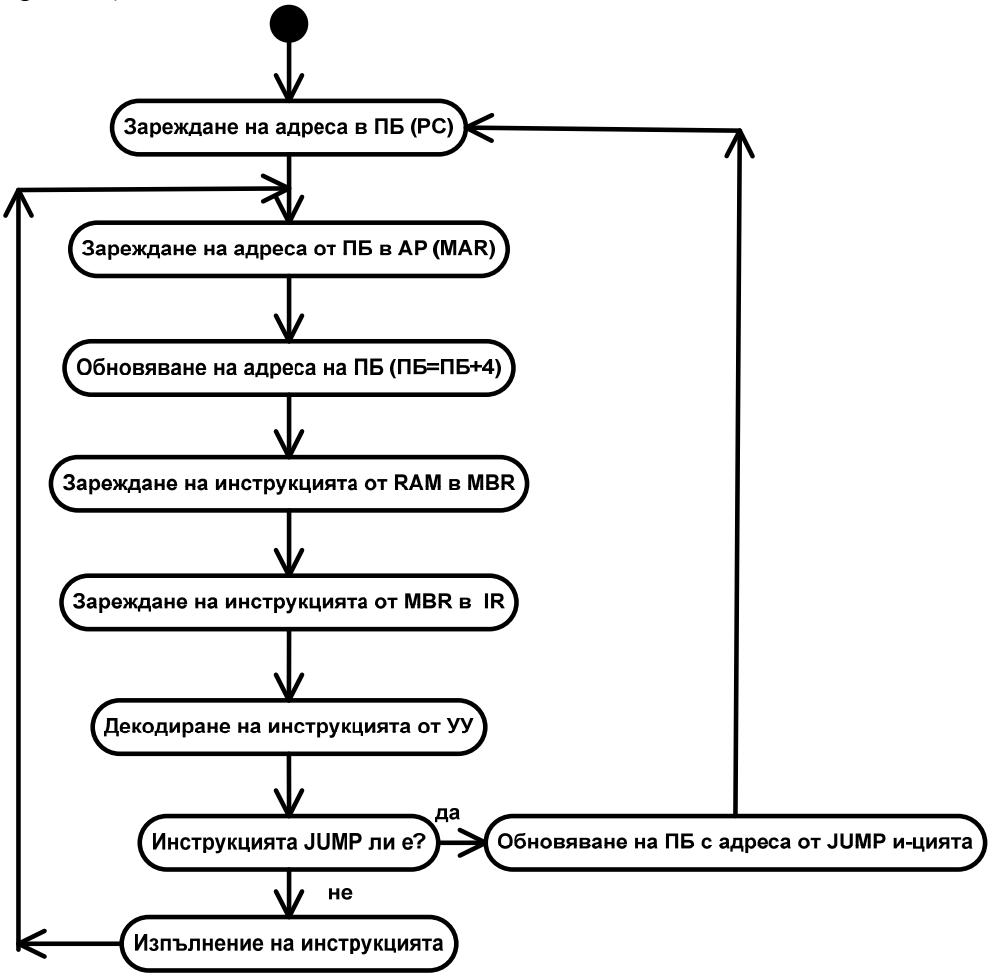



**Фигура 4.1. Регистри и устройсва на микропроцесор с един регистър R0**

В информационния регистър инструкцията се разделя на КОП и операнд/операнди. Кодът на инстукцията се подава на управляващото устройство /УУ/ (control unit) на микропроцесора за дешифриране, а операндът остава в информационния регистър, докато КОП се дешифрира и се реши как операндът да бъде обработен.

**Дешифриране и изпълненние на инструкцията**

Дешифрирането и изпълнението на инструкцията са свързани с генериране на управляващи сигнали от страна на УУ, които са свързани с КОП на инструкцията. Тези сигнали са различни за всяка една инструкция с изключение на инструкциите за условен или безусловен преход, които зареждат нов адрес в програмния брояч (скок към нова инструкция от програмата).




**Фигура 4.2. Блок-схема изпълнение на инструкциите от микропроцесора**

При различните микропроцесори декодирането (на КОП) на инструкциите става или от хардуерен декодер/дешифратор или от микропрограмно устройство. Хардуерният декодер е вграден в управляващото устройство на микропроцесора. С помощта на генератора на управляващи сигнали декодерът генерира поредица от управляващи сигнали към паметта, регистрите и АЛУ.

Микропрограмното устройство интерпретира КОП като поредица от микроинструкции, записани в ROM паметта на микропроцесора. На всяка инструкция на микропроцесора (на всеки КОП) съответства поредица от микроинструкции. При тяхното изпълнение се генерират аналогични управляващи сигнали към паметта, регистрите и АЛУ.

**Примери за изпълнение на инструкциите в микропроцесор с един регистър R0**

Разгледаните примери са свързани със зареждане на данни от паметта в регистъра R0, събиране на данни от паметта с тези в R0 и запис на данни от R0 в паметта:

**MOVE X, R0** зарежда числото от адрес X на RAM в регистър R0.

**ADD Y, R0** събира числото от адрес Y с това в регистър R0.

**MOVE R0, Z** запазва сумата от регистър R0 на адрес Z от памета.

За описание на преместването на инструкции и данни между регистрите и паметта е използван езикът RTL (Regiter Transfer Language), при който преместването на съдържанието на данните от един регистър в друг се преставя по следния начин:

**[P1] ← [P2]**,

където квадратните скоби имат смисъл на съдържание на регистър или клетка от паметта, а стрелката ← показва посоката на преместването (трансфера) на данните от P2 в P1.

В примерите по-долу за всяка една от инструкциите цикълът на инструкцията е еднакъв и включва описаните вече четири стъпки. Последните са представени чрез RTL записи като:

**1. [MAR] ← [PC]**
**2. [PC] ← [PC] + 4**
**3. [MBR] ← [[MAR]]**
**4. [IR] ← [MBR]**

Дешифрирането и изпълнението на инструкцията (изпълнителен цикъл) за всяка конкретна инструкция са различни и са описани подробно в примерите:

**Пример 4.1.**

**MOVE X, R0** Изпълнителен цикъл:

**5. [MAR] ←[IR-address]** – копира адреса на операнда X в MAR.

**6. [MBR] ←[[MAR]** – записва операнда от този адрес на паметта в MBR.

**7. [R0] ← [MBR]** – запазва операнда в регистър R0.

**Пример 4.2.**

**ADD Y, R0** Изпълнителен цикъл

**5. [MAR] ← [IRaddress]** – копира адреса на операнда Y в MAR

**6. [MBR] ←[[MAR]]** – прочита стойността на операнда Y от паметта

**7. ALU ← [MBR]** – изпраща стойността Y от MBR в АЛУ /ALU/

**8. ALU ← [R0]** – изпраща операнда (числото) от R0 в ALU

**9. [R0] ← ALU** – събира операндите и записва резултата в R0

**Пример 4.3.**

**MOVE R0, Z** Изпълнителен цикъл

**5. [MAR] ← [IR-address]** – копира адреса на операнда Z в MAR

**6. [MBR] ← [R0]** – изпраща операнда от R0 в MBR

**7. [[MAR]] ← [MBR]** – записва операнда от MBR на адрес Z, посочен от MAR

Представените примери са свързани с инструкции с два операнда, които са типични за много от съвременните процесори. Следващите примери (таблица 4.2) описват изпълненното на изчисленията

**A = B * C + D**

на базата на инструкции с един, два или три операнда.

**Таблица 4.2. Изчисление на израз при инструкции с 1, 2 и 3 операнда**

| С един операнд | С два операнда | С три операнда |
|---|---|---|
| load B<br>mult C<br>add D<br>store A | load B, A<br>mult C, A<br>add D, A | mult B, C, A<br>add D, A, A |

**Видове адресации**

Адресациите при микропроцесорите са свързани с начина на интерпретиране на операндите. Различават се четири основни вида адресации:

• Пряка адресация – операндът е число (литерал).

• Абсолютна адресация – операндът е адрес от паметта, където са данните.

• Индиректна адресация – операндът е указател към адрес от паметта, където са данните.

• Междурегистрова адресация – операндите са регистри на процесора.

Различните процесори поддържат, освен посочените адресации, и други допълнителни адресации, които са разширениие на индиректната и при които адресът от паметта, където се намират данните, се формира от операнд указател плюс отместване, формирано от стойността на константи или данни в определени регистри.

**Видове инструкции**

Инструкциите в микропроцесорните системи могат да се разделят на няколко вида:

• За преместване на данни:
  - От паметта в регистър;
  - От регистър в паметта;
  - От един регистър в друг;
  - За преместване на числова константа в регистър или в паметта.

• За аритметични операции:
  - Събиране / изваждане;
  - Умножение / деление и др;
  - Умножение по минус 1;
  - Инкраментиране / декраментиране.

• За ротация изместване на данните в регистър или в паметта.

• За условен и безусловен преход и преход към подпрограма.

• За логически операции (AND, OR, NOT, XOR) и др.

• За програмен контрол (Reset, Stop, Trap) и др.

Някои микропроцесори поддържат всички посочени видове инструкции и много от посочените адресации, а някои процесори – само част от тях.

Като цяло процесорите се делят на два вида: CISC (Complex Instruction Set for Computers) и RISC (Reduced Instruction Set for Computers) микропроцесори.

Първите поддържат комплексен набор от 50 до 100 инструкции, а вторите – редуциран набор (16 – 32) от специализирани инструкции. За първите е характерно (но не задължително) да имат малък брой регистри, а за вторите – голям брой регистри, като инструкциите им са ориентирани за междурегистрови операции. При RISC процесорите постоянният достъп до паметта е нежелателен.

**Пример 4.4.** По-долу е дадена програма на C++, която е интерпретатор на процесор с 8-битови инструкции и един регистър D0. Инструкциите имат 4-битов код на операцията и 4-битов операнд. В таблица 4.3 са описани битовете на КОП, като битове от 5 до 7 описват КОП, а бит 4 – вида на адресацията (абсолютна или пряка).

**КОП :**

**000 – LDA** зареждане на данни от паметта в регистър D0.
**001 – STA** зареждане на данни от регистър D0 в паметта.
**010 – ADD** събиране на данни с тези в регистър D0; резултатът е в D0.
**011 – SUB** изваждане на данни от тези в регистър D0; резултатът е в D0.
**100 – MUL** умножение на данни с тези в регистър D0; резултатът е в D0.
**101 – DIV** делене на данни с тези в регистър D0, като резултатът е в D0.
**111 – STOP** край на програмата.

**Адресиране (адресен режим):**

0 – абсолютно (операндът е абсолютен адрес)

1 – пряко (операндът е директно число)

Битове от 0 до 3 описват или адрес на операнда, или директно стойността на операнда:

– Адрес на операнда – от 0 до 15, ако адресирането е абсолютно.

– Стойност на операнда – число от 0 до 15, ако адресирането е пряко.

**Таблица 4.3. Значение на битовете в инстукция с един операнд**

| 8-битов формат на инструкцията | | | | | | | |
|---|---|---|---|---|---|---|---|
| Бит 7 | Бит 6 | Бит 5 | Бит 4 | Бит 3 | Бит 2 | Бит 1 | Бит 0 |
| Код на операцията | | | адресиране | 4-битов операнд | | | |

Понеже максималният адрес е в диапазона от 0 до15, то програмата с данните трябва да заема максимум 16 байта. Инструкциите на програмата (таблица 4.4) започват от адрес 0 на масива memory (последната колона).

На основата на така зададените инструкции да се пресметне изразът

**D = 4 * (A * B + C),** където A = 10, B = 3, C = 7. Клетките A, B, C и D се намират на адреси 10, 11, 12 и 13.

**Таблица 4.4. Инструкции на програмата, изчисляваща израза 4 * (A * B + C) и техния двоичен и десетичен код:**

| Инструкции | КОП | Адресация | Операнд | 2-чно | 10-чно | memory |
|---|---|---|---|---|---|---|
| LDA A | 000 | 0 – адрес | 1010 | 00001010 | 10 | 0 |
| MUL B | 100 | 0 – адрес | 1011 | 10001011 | 139 | 1 |
| ADD C | 010 | 0 – адрес | 1100 | 01001100 | 76 | 2 |
| MUL 4 | 100 | 1 – число | 0100 | 10010100 | 148 | 3 |
| STA D | 001 | 0 – адрес | 1101 | 00101101 | 45 | 4 |
| STOP | 111 | 0 | 0000 | 11100000 | 224 | 5 |
| | | | | 00000000 | 0 | 6 |
| | | | | 00000000 | 0 | 7 |
| | | | | 00000000 | 0 | 8 |
| | | | | 00000000 | 0 | 9 |
| A | | | | 00001010 | 10 | 10 |
| B | | | | 00000011 | 3 | 11 |
| C | | | | 00000111 | 7 | 12 |
| D | | | | 00000000 | 0 | 13 |

На базата на по-горната таблица програмата и данните представляват поредица от числа, дадени в колона 10-чно, а именно (10, 137, 76, 132, 45, 224, 0, 0, 0, 0,10, 3, 7, 0, 0, 0), които трябва да бъдат въведени от клавиатурата по време на изпълнение на програмния интерпретатор.

Резултатът от изпълнението може да се види в клетка D на адрес 13 и той трябва да е 148.

**Програмен код на интерпретатора на 8-битови инстукции:**

```
#include<iostream.h>
// дефиниране на КОП
#define LDA 0
#define STA 1
#define ADD 2
#define SUB 3
#define MUL 4
#define DIV 5
#define STOP 7
void main () {
unsigned short int PC = 0;  // ПБ сочи първата инструкция
unsigned short int D0 = 0;  // нулиране на D0
unsigned short int MAR;     // Memory Address Register
unsigned short int MBR;     // Memory Buffer Register
unsigned short int IR;      // Information Register
unsigned short int operand; // Operand
unsigned short int source;  // число или данни от паметта
unsigned short int opcode;  // КОП
unsigned short int amode;   // адресен режим
// декларираме на масива - паметта за програмата и данните
unsigned short int memory[16];
//{10,139,108,41,224,0,0,0,0,40,20,3,10,0} ;
```

```
unsigned short int run = 1;     // управляваща променлива
int i;
// Въвеждане на инструкциите и данните, като 10 тични
числа
cout << "Input program instructions and data" << endl;
for (i=0; i < 16; i++) {
    cout << "memory[" << i << "] = ";
    cin >> memory[i];
}

while (run)
{  // докато управляващата променлива е различна от 0
изпълнявай
   // инструкциите на програмата
MAR = PC;             // зареждане на инструкцията,
посочена от ПБ
PC = PC + 1;          // обнвяване на ПБ - сочи следващата
MBR = memory [MAR];  // Извличане на инструкцията от RAM и
запис в MBR
IR = MBR ;          // запис на инструкцията в
информационния регистър IR
opcode = IR >> 5;   // извличане на КОП чрез изместване на
IR 5 пъти надясно
amode = (IR & 0x10) >> 4;  // извличане на адресния режим
operand = IR & 0x0F;  // извличане на операнда чрез
побитов AND s маска0x0F
if (amode == 0) {   // ако операндът е адрес
    source = memory [operand];  // запис на данните от
адреса в source
    cout <<"memory address " << operand << " access ";
}
else {  // ако операндът е число
    cout << "direct number " <<operand << " used " <<
endl;
    source = operand;   // запис на числото в source
}
switch(opcode)
{ // изпълнение на КОП на текущата инструкция
case LDA : {D0 = source ;cout <<"STA"<< endl;   break;}
case STA : {memory[operand] = D0;cout <<
"STA"<<endl;break;}
case ADD : {D0 = D0 + source;cout << "ADD" <<endl;break;}
case SUB : {D0 = D0 - source;cout << "SUB" <<endl;break;}
case MUL : {D0 = D0 * source;cout << "MUL" <<endl;break;}
case DIV : {D0 = D0 / source;cout << "DIV" <<endl;break;}
case STOP: {cout << "STOP" <<endl;run = 0;}
}     //  end of case
}     //  end of while
// разпечатване на програмата и резултатите от нея като
10-чни числа
cout << "Print program instructions and results" <<endl;
for (i = 0; i<16; i++) {
   cout << "memory [" << i << "] = " << memory[i] << endl;
   }     // end of for
}    // end of program
```

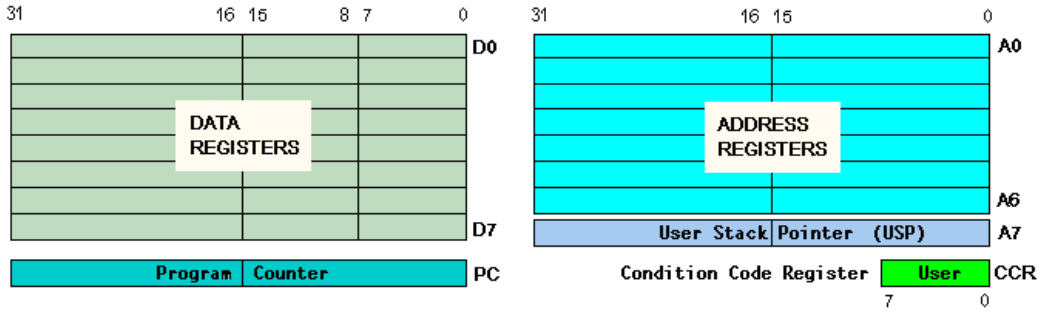
**Задачи:**

1. Обяснете предназначението на следните регистри на микропроцесора: MAR, MBR, IR, PC, CCR.
2. Какъв е максималният брой на инструкциите и какви са максималният адрес или число, които могат да се кодират в 12-битова инструкция с 4-битов КОП?
3. Какъв е максималният брой на инструкциите и какви са максималният адрес или число, които могат да се кодират в 8-битова инструкция с 3-битов КОП?
4. Какъв е максималният брой на инструкциите, адресациите и максималният адрес или число, които могат да се кодират в 24-битова инструкция с 8-битов КОП с два бита за адресация?
5. Какъв е максималният брой на инструкциите, адресациите и максималният адрес или число, които могат да се кодират в 6-байтова инструкция с 16-битов КОП и с 4 бита за адресация?
6. Какъв е максималният брой на инструкциите и какви са максималният адрес или число, които могат да се кодират в 16-битова инструкция с 5-битов КОП?
7. Каква е разликата между CISS и RICS микропроцесорите?
8. Определете всички възможни инструкции за преместване на данни на процесор с два регистъра R0 и R1, който поддържа пряка и абсолютна адресации.
9. Определете всички възможни инструкции за преместване на данни на процесор с четири регистъра R0 -:- R4, който поддържа пряка и абсолютна адресации.
10. Определете всички възможни аритметични инструкции на процесор с четири регистъра R0 -:- R4, който поддържа пряка и абсолютна адресации.
11. Какво е адресното пространство на процесор с операнд с големина 12, 16, 24, 32, 48 и 64 бита?
12. Кои са основните адресации ? Дайте пример.
13. Какво представлява операндът и каква информация може да съдържа?
14. Използвайте програмния интерпретатор и създайте програма, която да изчислява израза D = (A * B + C) / 5, където A = 20, B = 4, C = 8. Клетките A,B,C и D се намират на адреси 9,10,11 и12
15. Използвайте програмния интерпретатор и създайте програма, която да изчислява израза D = M * (A + B-C) / 5, където A = 120, B = 4, C = 8, M = 3. Клетките A, B, C, M и D се намират на адреси 10, 11, 12, 13 и14.
16. Използвайте програмния интерпретатор и създайте програма, която да изчислява израза D = A-(A * B-C) / 6, където A = 20, B = 4, C = 8. Клетките A, B, C и D се намират на адреси 12, 13, 14 и 15.

# 5. Програмиране на асемблер за процесор Motorola MC68000

**Програмен модел на MC68000**

Микропроцесорът Motorola MC68000 има осем регистъра за данни с имена D0, D1 до D7 и осем адресни регистъра с имена A0, A1 до A7 (фиг. 5.1). Регистърът A7 служи за организиране на стек, използван при работа с подпрограми. Всички регистри за данни или адреси са 32-битови (4-байтови).




**Фигура 5.1. Регистри на MC68000**

Освен регистрите за данни и адреси, които са с общо предназначение, в MC68000 има два специални регистъра: програмен брояч (PC), който е 32-битов, и един 8-битов статус регистър (CCR или SR). Програмният брояч (таблица 5.1) съдържа адреса на текущо изпълняваната инструкция, а статус регистъра съдържа специални битове, които се променят (от 0 в 1) в следствие на изпълнението на конкретна инструкция:

- Бит C (carry) става 1-ца при възникване на пренос при събиране на две числа или става 1-ца при заем при изваждане на две числа. Преносът е към по старшия байт, дума или дълга дума, а заемът е от по-старшия байт, дума или дълга дума.
- Бит V (overflow), който става 1-ца при препълване на регистър.
- Бит Z (zero), който става 1-ца, ако резултатът след някаква операция (аритметична, логическа или преместване) е нула (0).
- Бит N (negative) става 1-ца при получаване на отрицателен резултат.
- Бит X (extend), който дублира стойността на бит C.

**Таблица 5.1. Битове на статус регистъра**

| **Бит име** | * | * | * | **X** | **N** | **Z** | **V** | **C** |
|---|---|---|---|---|---|---|---|---|
| **Бит номер** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |

Регистрите за данни може да оперират с данни с големина байт, дума (2 байта) или дълга дума (4 байта).

Адресите в MC68000 са 24-битови, но адресните регистри са 32-битови, затова при работа с адреси старшите 8 бита на адресните регистри се пренебрегват.

Част от инструкциите на MC68000 оперират с един операнд, а друга част с два операнда:

- INST1 OP1 инструкция INST1 с един операнд OP1.
- INST2 OP1, OP2 инструкция INST2 с два операнда OP1, OP2.

При инструкциите с един операнд операндът може да бъде регистър за данни или адреси, адрес от паметта или символичното му име, или директно число. При инструкциите с два операнда първият операнд може да е като операнда при инструкциите с един операнд. Вторият операнд обаче не може да бъде директно число, защото в този операнд се записва резултатът от изпълнението на инструкцията.

За повечето инструкции не се допуска и двата операнда да са адреси или символични имена на клетки от паметта. Изключение прави инструкцията MOVE, както и някои други инструкции по сравнение като CMPM или някои специфични индексни адресации, които са дадени в следващата глава.

След името на всяка една инструкция с разширение се отбелязва големината на данните, с които тя работи. За целта се използват буквите B (байт), W (дума) или L (дълга дума). При липса на разширение се подразбира, че инструкцията работи с данни с големина дума (W).

**Формат на инструкцията на MC68000**

Следва пример на една типична инструкция с два операнда:

**Label MOVE.B P, D0**

където:

- **Label** – име (етикет) на адреса на инструкция, към който се прави преход от друго място на програмата. Етикетът може да е с големина до 7 символа, като първият е буква. Етикетът не е задължителен за всяка инструкция.
- **MOVE.B** – името на инстукцията, в случая за преместване на данни, където с "**.B**", "**.W**" или "**.L**" се означават форматът на данните й.
- **P** – име на клетка (адрес) от паметта.
- **D0** – регистър за данни.

**Видове инструкции и тяхните разновидности:**

По-долу са дадени най-често използваните инструкции на MC68000, който е CISC процесор.

Пълният набор на инструкциите на микропроцесора може да бъде зареден от интернет адрес, който е даден в списъка на ползваната литература в края на ръководството (вж. 12).

**Инструкции за местене на данни:**

| Инструкция | Обяснение |
|---|---|
| **MOVE.B OPIT, D1** | Преместване на данните с големина един байт (.B) от клетка от паметта с име OPIT в регистър D1. |
| **MOVE.W D1, Result** | Преместване на данните с големина дума (.W) от регистър D1 в клетка от паметта с име Result. |
| **MOVE.L D1, D2** | Преместване на данните с големина дълга дума от регистър D1 в D2. |
| **MOVE.L #12,D2** | Записване на константа (десетично число 12) в регистър D2. Знакът # служи за декларираме на константа. Константата може да бъде десетична (например #12), двоична (например #%00001111), шестнадесетична (например #$ABC9), символна (#'M') или текстова (#"TEXT"). Ако знакът # липсва, то тогава вместо константа се разбира адрес на клетка от паметта (например адрес 12). |
| **MOVE $1200, D0** | Зареждане на двубайтовото число от адрес $1200 в регистъра D0. |

**Аритметични инструкции:**

| **Инструкции за събиране:** | |
|---|---|
| **Инструкция** | **Обяснение** |
| **ADD.B P, D0** | Събиране на еднобайтовото число на адрес P с това в регистъра D0, като резултатът се записва в D0. |
| **ADD.W D1, D2** | Събиране на двубайтовите числа от регистрите D1 и D2, като резултатът се записва в D2. |
| **ADD.L V, D3** | Събиране на четирибайтовото число на адрес V и това в регистъра D3, като резултатът се записва в D3. |
| **ADDX D0, D1** | Събиране с пренос на двубайтовите числа от регистри D0 и D1 (D1 = D1 + D0 + X), като резултатът е в D1. X е бит на статус регистъра. |

| **Инструкции за изваждане:** | |
|---|---|
| **Инструкция** | **Обяснение** |
| **SUB.B P, D0** | Изваждане на еднобайтовото число на адрес P от това в регистъра D0 (D0 = D0 - P). Резултатът е в D0. |
| **SUB.W D0, D1** | Изваждане на двубайтовите (.W) числа в регистрите D0 и D1 (D1 = D1 - D0). Резултатът е в D1 |
| **SUB.L D0, D1** | Изваждане на четирибайтовите (.L) числа в регистрите D0 и D1 (D1 = D1 - D0). Резултатът е в D1 |
| **SUBX D0, D1** | Изваждане със заем на двубайтовите (.W) числа в регистрите D0 и D1 (D1 = D1 - D0 - X), като резултатът е в регистър D1. X е бит на статус регистъра. |
| **Инструкции за умножение:** | |
| **Инструкция** | **Обяснение** |
| **MULU D1, D0** | Умножение на двубайтовите числа без знак от регистри D1 и D0 (D0 = D1 * D0), като резултатът (до 4 байта) е в регистър D0. |
| **MULU ABC,D0** | Умножение без знак на двубайтовото число от клетка в паметта с име ABC с това в регистър D0 (D0 = D0 * ABC), като резултатът (до 4 байта) е в D0. |
| **MULU #3, D0** | Умножение на константата 3 с числото в регистър D0 (D0 = 3 * D0), като резултатът е в регистър D0. |
| **MULS #-5, D7** | Умножение на числа със знак (константата -5) и числото в регистър D7 (D7 = -5 * D7), като резултатът е в регистъра D7. |
| **Инструкции за деление:** | |
| **Инструкция** | **Обяснение** |
| **DIVU D1, D0** | Деление на две числа без знак (D0 = D0 / D1) с двубайтов резултат. Младшите два байта на D0 са частно, а остатъкът от деленето е в старшите два байта на D0. Например при деление на 14 на 5 (14 / 5) се получава частно 2 и остатък 4. |
| **DIVS #-6, D4** | Деление на числа със знак D7 = D7 / (-6). |

| **Знаково разширяване:** | |
|---|---|
| **Инструкция** | **Обяснение** |
| **EXT.W D0** | Знаково разширяване на регистъра за данни. При тази операция най-старшият бит на числото от младшия байт се разпространява в следващия старши байт. |
| **Инструкция** | **Обяснение** |
| **EXT.W D0** | Например, ако D0 е съдържал $1122, то резултатът след **EXT.W** ще е $0022, но ако D0 е съдържал $1188, то резултатът ще е $FF88. |

**Инструкции за работа с адреси:**

| **Инструкция** | **Обяснение** |
|---|---|
| **MOVEA.L D1, A0** | Записване на стойността на регистъра D1 като адрес в регистъра A0. |
| **ADDA D1, A3** | Събиране на числото от регистър D1 с адреса в регистър A3, като резултатът е в A3. |
| **SUBA D1, A3** | Изваждане на числото от регистър D1 с адреса в регистър A3, като резултатът е в A3. |
| **LEA LIST, A0** | Зареждане на адреса на клетка с име LIST в адресен регистър A0. |

**Инструкции за безусловен преход:**

| **Инструкция** | **Обяснение** |
|---|---|
| **BRA NEXT** | Безусловен преход към адрес, посочен от етикет NEXT. Адресът може да е в обсег + / - 32Кбайта спрямо адреса на инструкцията BRA. |
| **JMP NEXT1** | Безусловен преход към адрес, посочен от етикет NEXT1, като този адрес може да е 24-битов. |

Възможно е операндът на инструкциите **BRA** или **JMP** да бъде и директен адрес, например: **BRA $3000** или **JMP $AABB00**.

**Инструкции за условен преход**

Тези инструкции се използват в съчетание с инструкцията за сравнение **CMP**. Последната сравнява двата си операнда, като изважда от втория операнд първия. Изпълнението на инструкцията не променя стойността на операндите, а само влияе на определени битове в статус регистъра. На базата на стойностите на тези битове инструкциите за условен преход осъществяват преход (или не) към посочен адрес от програмата.

Битовете на статус регистъра могат да се променят вследствие на изпълнение на инструкциите за аритметични операции или тези за ротация и преместване. Например, ако в резултат на аритметична операция даден регистър се нулира, то това предизвиква установяване на бит Z в единица. Ако се получи препълване на даден регистър (всичките му битове станат единици), то тогава бит V се установява в единица. Ако при събиране на две числа възникне пренос към по-старши бит на следващ байт, дума или дълга дума, то тогава битовете C и X стават единици. Същото важи и за случая на изваждане на две числа, когато възникне заем от по-старши бит на следващ байт, дума или дълга дума.

Следващите инструкции за условен преход зависят от резултата на инструкцията **CMP**.

| **Инструкция** | **Обяснение** |
|---|---|
| **CMP X, D0** | Сравняване на двубайтовото число от адрес X с това в регистър D0 (D0-X), при което се променят битове в статус регистъра. |
| **CMPM Mem1,Mem2** | Сравняване на съдържанието на две двубайтови числа от паметта (Mem2 = Mem2-Mem1), при което се променят битове в статус регистъра. |
| **BPL NEXT** | Преход към NEXT при положителен резултат. |
| **BMI NEXT** | Преход към NEXT при отрицателен резултат |
| **BEQ NEXT** | Преход към етикет NEXT при равенство. |
| **BNE NEXT** | Преход към етикет NEXT при неравенство. |
| **BGE NEXT** | Преход към етикет NEXT, ако D0 >= X. |
| **BGT NEXT** | Преход към етикет NEXT, ако D0 > X. |
| **BLE NEXT** | Преход към етикет NEXT, ако D0 <= X. |
| **BLT NEXT** | Преход към етикет NEXT, ако D0 < X. |
| **DBRA D1, Next** | Декраментиране на регистъра D1 и преход към етикет Next, ако съдържанието на D1 не е станало минус единица (-1). Инструкцията се използва за организиране на цикли, като регистърът (в случая D1) се използва за брояч. |
| **DBxx D1, Next** | Тези инструкции (**DBxx**) декраментират D1 и правят преход към Next, ако съдържанието на D1 не е станало -1, при условие противоположно на това на инстукциите **Bxx** (**BEQ, BNE, BPL** и т.н.). |

По аналогия с инструкциите за безусловен преход и инструкциите за условен преход могат вместо етикет за преход към точка от програмата да използват директен адрес (виж по-горе).

**Инструкции за преход към подпрограма:**

| Инструкция | Обяснение |
|---|---|
| **BSR Func_1** | Извикване на подпрограма, намираща се на адрес от паметта с име Func_1. Подпрограмата трябва да е в обсег + / - 32К байта спрямо адреса на **BSR**. |
| **JSR Func_1** | Извикване на подпрограма, намираща се на адрес от паметта с име Func_1. Подпрограмата може да се намира в пълния 24-битов адресен обхват на паметта. |
| **RTS** | Край на подпрограмата и връщане в главната програма. |

Извикването на подпрограмите може да става както с използването на символичните им имена, така и с използването на абсолютния им адрес в паметта. Например: **BSR $1500**.

**Инструкции за преместване и ротация**

При логическото преместване наляво или надясно на битовете на дадена клетка от паметта или на регистър на тяхно място влизат нули.

При аритметичното преместване наляво на битовете на дадена клетка от паметта или на регистър на тяхно място влизат нули, а при преместване надясно – единици. При ротация наляво излезлите битове отляво се появяват отдясно и обратно при ротация надясно. Преместването и ротацията влияят на битове C и X от статус регистъра.

| Инструкция | Обяснение |
|---|---|
| **LSL.B #1, D0** | Логическо преместване наляво един път (#1). Ако в регистъра D0 е имало %11001100, то след изпълнение на инструкцията същият ще съдържа %10011000. |
| **LSR.B #2, D0** | Логическо преместване надясно 2 пъти (#2). Ако в регистъра D0 е имало %11001100, то след изпълнение на инструкцията същият ще съдържа % 00110011. |
| **ASL.B #1, D0** | Аритметично преместване един път наляво. Ако D0 е имало %11001100, то след изпълнение на инструкцията същият ще съдържа %10011000. |
| **ASR.B #1, D0** | Аритметично преместване един път надясно (#1). Ако в регистъра D0 е имало %11011100, то след изпълнение на инструкцията същият ще съдържа %11100110. |
| **ROL.B #1, D0** | Ротация един път наляво (#1). Ако в регистъра D0 е имало %11011100, то след изпълнение на инструкцията същият ще съдържа %10111001. |
| **ROR.B #2, XX** | Ротация на клетка от паметта XX два пъти надясно. Ако в XX е имало %11011100, то след изпълнение на инструкцията клетката XX ще съдържа %00110111. |
| **SWAP D0** | Разменяне на съдържанието на старшите два байта на регистър D0 със съдържанието на младшите два байта. |

**Логически инструкции:**

| Инструкция | Обяснение |
|---|---|
| **AND.B #%10101010, D0** | Операция AND на константата %10101010 с байт от регистъра D0, като резултатът е в D0. |
| **ORI.B #$%11110000, XY** | Операция OR на константата %11110000 с байт от паметта с име XY. В резултат старшите 4 бита на XY стават в единици. |
| **OR D0, XY** | Операция OR на два байта от регистъра D0 с клетка XY, като резултатът е в XY. |
| **XOR D0, XY** | Операция XOR на два байта от регистъра D0 с клетка XY, като резултатът е в XY. |
| **NOT.B D0** | Логическата операция NOT с D0. Ако D0 е бил %11110000, то същият става %00001111. |

**Инструкции за BCD аритметика:**

| Инструкция | Обяснение |
|---|---|
| **ABCD D0,D1** | Събиране с пренос на BCD числата от регистри D0 и D1 (D1 = D0 + D1 + бит X). |
| **ABCD -(A0), -(A3)** | Събиране с пренос на BCD числата, посочени от адресните регистри A0 и A3. Използвана е адресация с пред-декарментиране. |
| **SBCD D3,D5** | Изваждане със заем на BCD числата от регистри D5 и D3 (D5 = D5-D3-бит X). |
| **SBCD -(A0), -(A3)** | Изваждане със заем на BCD числата, посочени от адресните регистри A0 и A3. Използвана е адресация с пред-декарментиране. |
| **NBCD D3** | Операция минус на BCD числото в регистър D3. От нула се изважда D3 със заем бит X (D3 = 0-D3-бит X). |

**Други инструкции:**

| Инструкция | Обяснение |
|---|---|
| **STOP #$2700** | Край на програмата |
| **TRAP #15** | Генериране на прекъсване с номер 15 към ОС. |
| **NOP** | Нулева операция (инструкция без операнд) |

**Директиви на асемблера**

Директивите на асемблера на MC68000 са указания към компилиращата програма. Чрез тях се задават начални адреси, на които се записва изпълнимият код на програмата и подпрограмите й, дефинират се константи или се резервира памет за променливи, за масиви и др.

| Директива | Обяснение |
|---|---|
| **START ORG $1200** | Директивата **ORG** задава начален адрес ($1200) на програма, подпрограма или на блок от данни. |
| **XY DS.W 2** | Директивата **DS** резервира памет 2 байта за клетка с име (етикет) XY. |
| **XX DC.B 12** | Директивата **DC** резервира памет 1 байт за константа XX. |
| **AA EQU 12** | Директивата **EQU** служи за декларираме на константа с име AA. Ако в програмата се срещне тази константа, то тя автоматично се заменя с конкретната стойност (в случая числото 12-десетично). |
| **F1 EQU $12** | Декларираме на шестнадесетичната константа F1 с директивата **EQU**. |
| **Z EQU %11001111** | Декларираме на двоичната константа Z с директивата **EQU**. |
| **END $1200** | Директивата **END** служи за край на програмата. Всички инструкции и директиви в програмата след нея се игнорират. Обикновено тя сочи началния адрес на програмата, например $1200. |
| **END START** | Край на блок от данни с етикет, който сочи началния адрес на програмата (**START**). |

**Задачи:**

1. Обяснете значението на следните инструкции:

а) **MOVE 3000, 4000**
б) **MOVE D0, D6**
в) **MOVE 3000, D6**
г) **MOVE D6, 3000**
д) **MOVE #4000, 5000**
е) **MOVE #4000, D1**

2. Какво е грешно в следните инструкции и защо:

а) **CMP X, Y**
б) **CMPM D0,D4**
в) **MOVE.B #1500,D5**
г) **MOVE $4000,A9**
д) **ADD.W (A2)-,D3**
е) **SUB $800, (A8)+**
ж) **ADD.B #1,A3**
з) **BEQ.B Next_3**

3. Как може да се зареди адресът $6400 в адресен регистър A5? Дайте примери с инструкциите: **LEA, MOVEA и MOVE**.

4. За какво се използват инструкциите **ADDX** и **SUBX** и по какво се различават от инструкциите **ADD** и **SUB**?

5. Има ли разлика между инструкциите за изместване на данни **ASL** и **LSL** и каква е тя?

6. Има ли разлика между инструкциите за изместване на данни **ASR** и **LSR** и каква е тя?

7. Какъв е резултатът от изместването на числото %00001100, записано в регистъра D0 с инструкцията:

а) **ROL #3, D0**
б) **ROR #4, D0**
в) **ROL.B #3, D0**
г) **ROR.B #4, D0**
д) **ASR #3, D0**
е) **ASR #4, D0**

8. Каква е разликата между директивите:

**XX DC 50** и **XX EQU 50?**

9. Какъв би бил резултът след изпълнението на инструкцията:

**MOVE D0, XX**, ако XX е дефиниран с директивите **DC** или **EQU**?

10. Може ли в една програма да има повече от една директива **ORG**?

11. Кои са инструкциите за умножение и деление на числа със знак?

12 Кои са инструкциите за условен и безусловен преход? Възможно ли е да се използват инструкции за условен преход без преди това да е използвана инструкцията **CMP**?

13. За какво се използват битовете Z, N, C, V и X от статус регистъра (CCR / SR)?

14. На кой бит ще повлияят следните инструкции:

а) **SUB D0, D0**
б) **MOVE #$0, D5**
в) **MULS #-3, D2**, ако в D2 има положително число;
г) **MULS #-3, D2**, ако в D2 има отрицателно число;
д) **ADD.W #$FFFF,D2**, ако в D2 има положително число;
е) **SUB.L #$FFFF,D2**, ако в D2 има положително 4-байтово число.

15. С какви разширения могат да се използват инструкциите, свързани с регистрите за данни и за адреси?

16. Колко битови са адресните регистри и регистрите за данни и кои са най-голямото число и най-големият адрес, които може да се запишат в тях?

# 6. Видове адресации на процесора MC68000

**Адресации**

Процесорът MC68000 има 14 различни вида адресации (адресни режима). Част от адресациите са свързани с обмен на данни между регистри. Такива са директната адресация между регистри за данни и тази между адресни регистри. Други адресации са свързани с работа с абсолютни адреси като например абсолютната кратка адресация и абсолютната дълга адресация. Трети вид адресации са индиректните, като чистата индиректна регистрова адресация и тези със след-инкраментиране и с пред-декраментиране, както и индиректната регистрова адресация с изместване (офсет) и тази индиректна с офсет и индекс. Следваща група са относителните адресации спрямо програмния бояч (PC) и тези с офсет и индекс спрямо него. Често използвани са също така пряката дълга и кратка адресации, както и косвената адресация между специалните регистри.

**1. Директна адресация между регистри за данни**

При тази адресация регистрите за данни (D0-D7) са мястото, където се намират или записват данните. Най-често тази адресация се използва при преместване на данни между регистрите за данни или при извършване на аритметични операции.

**Пример 6.1.** В инструкцията **MOVE.B D0, D3**, разширението ".B" означава, че само най-младшият байт на регистъра D0 ще се копира в най-младшия байт на регистъра D3.

| **Инструкция** | **MOVE.B D0,D3** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | | | D0<br>D3 | 10204FFF<br>1034F88A |
| След | | | D0<br>D3 | 10204FFF<br>1034F8FF |

**Пример 6.2.** В инструкцията **ADD.W D1, D2**, разширението ".W" означава, че само най-младшите два байта на регистъра D1 ще се съберат с тези от регистър D2, като резултатът остава в регистър D2.

| **Инструкция** | **ADD.W D1, D2** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | | | D1<br>D2 | 10201122<br>10342266 |
| След | | | D1<br>D2 | 10201122<br>10343388 |

**2. Директна адресация между адресни регистри**

При тази адресация регистрите за адреси (A0-A7) съдържат резултантните данни. За операнди могат да се ползват само дума или дълга дума. Думата се преобразува към размера на регистрите (дълга дума).

Най-често тази адресация се използва за формиране на нови адреси, използвани от индиректните регистрови адресации. Инструкциите, ориентирани за работа с адреси са **MOVEA, ADDA, SUBA и LEA**.

**Пример 6.3.** Съдържанието на адресен регистър A3 се копира в адресен регистър A0.

| **Инструкция** | **MOVEA.L A3,A0** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | | | A3<br>A0 | 00201234<br>00AAFFFF |
| След | | | A3<br>A0 | 00201234<br>00201234 |

**3. Абсолютна кратка адресация**

Адресът в RAM-паметта, от където се вземат данните или където се записват, се специфицира с една дума (16-битов кратък адрес).

**Пример 6.4.** Операндът (източник на данни) е дирекно число, което автоматично се записва в байт от паметта с адрес, защото инструкцията е с разширение “B”.

| **Инструкция** | **MOVE.B #08, $1000** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди<br>След | $1000<br>$1000 | 00<br>08 | | |

**4. Абсолютна дълга адресация**

Адресът в RAM, от където се вземат данните или където се записват се специфицира с две думи (32-битов адрес). Много често разликата между абсолютната адресацията с кратък или дълъг адрес не може да бъде направена, защото вместо адреси се използват етикети. Наприме в инструкцията **MOVE.W D3, RESULT**, операндът RESULT може да намира на кратък или дълъг адрес в паметта.

**Пример 6.5.** Адресът на данни в паметта е дълъг до 32 бита адрес. Данните от този адрес с големина един байт се записват в регистъра D0.

| **Инструкция** | **MOVE.B $8F000,D0** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | $0008F000<br>$0008F001 | FF<br>34 | D0 | 00000000 |
| След | $0008F000<br>$0008F001 | FF<br>34 | D0 | 000000FF |

Аболютната къса и дълга адресации позволяват инструкцията MOVE да се използва, като и двата й операнда са абслютни адреси или етикети на клетки от паметта.

**5. Индиректна регистрова адресация**

При тази адресация един от адресните регистри съдържа адреса на операнда, от който ще бъдат извлечени данните или адреса на операнда, където ще бъдат записани.

При индиректната регистрова адресация адресният регистър се загражда в скоби. Скобите указват, че данните от адреса в адресния регистър ще бъдат извлечени или на тяхно място ще бъдат записани други данни.

**Пример 6.6.** Инструкцията по-долу запазва данните (дълга дума) от регистър D0 на адреса, посочен от адресен регистър A0.

| **Инструкция** | **MOVE.L D0, (A0)** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | $00001000<br>$00001001<br>$00001002<br>$00001003 | 55<br>02<br>3F<br>00 | A0<br>D0 | 00001000<br>1043834F |
| След | $00001000<br>$00001001<br>$00001002<br>$00001003 | 10<br>43<br>83<br>4F | A0<br>D0 | 00001000<br>1043834F |

Обърнете внимание, че инструкциите **MOVE D0,A0** и **MOVE A0,D0** са допустими инструкции, но те не използват индиректна регистрова адресация. При тяхното изпълнение числото от първия регистър се записва във втория или това са директни междурегистрови адресации.

### 6. Индиректна регистрова адресация със след-инкраментиране

При тази адресация се използва след-инкраментиране, което се индикира със знака "+" след даден адресен регистър **(Ai).** След четенето или записването на данни, адресът, който се намира в адресния регистър, се увеличава с условна единица (в зависимост от разширението на инструкцията ".B", ".W" или ".L") и сочи следващия адрес в паметта:

байт: **[Ai]←[Ai] + 1**
дума: **[Ai]←[Ai] + 2**
дълга дума: **[Ai]←[Ai] + 4**

**Пример 6.7.** В конкретния пример данните, които се намират на адрес посочен от A5 се преместват в регистъра D0, след което адресът в A5 се увеличава с 2 и сочи следващите данни.

| **Инструкция** | **MOVE.W (A5)+, D0** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | $00001000<br>$00001001<br>$00001002<br>$00001003 | 45<br>67<br>89<br>AB | A5<br>D0 | 00001000<br>0000FFFF |
| След | $00001000<br>$00001001<br>$00001002<br>$00001003 | 45<br>67<br>89<br>AB | A5<br>D0 | 00001002<br>00004567 |

### 7. Индиректна регистрова адресация с пред-декраментиране

При тази адресация се използва пред-декраментиране, което се индикира със знака "-" пред адресния регистър **(Ai).**

Адресът, който се намира в адресния регистър, се намалява с условна единица (в зависимост от разширението на инструкцията) преди четенето или записването на данни. Така адресният регисър сочи предходния адрес в паметта:

байт: **[Ai]←[Ai] - 1**
дума: **[Ai]←[Ai] - 2**
дълга дума: **[Ai]←[Ai] - 4**

**Пример 6.8.** В примера по-долу данните, които се намират в регистъра за данни D0 се записват на адрес, посочен регистъра A7, намален предварително 2.

Преди изпълнението на инструкцията A7 е съдържал адрес $1002, който при изпълнение на инструкцията е намален с 2. Така се е получил ефективният адрес $1000, на който са записани данните от регистъра D0.

| **Инструкция** | **MOVE.W D0, -(A7)** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | $00001000<br>$00001001<br>$00001002<br>$00001003 | 10<br>12<br>83<br>47 | A7<br>D0 | 00001002<br>00000143 |
| След | $00001000<br>$00001001<br>$00001002<br>$00001003 | 01<br>43<br>83<br>47 | A7<br>D0 | 00001000<br>00000143 |

**8. Индиректна регистрова адресация с изместване (офсет)**

При тази адресация се използва 16-битово знаково изместване (офсет), което се добавя към стойността, записана в адресния регистър, така че ефективният адрес е сума от адреса плюс офсета.

**[Ai]←[Ai] + изместване**

**Пример 6.9.** Ефективният адрес в този пример се получава като към адреса от адресния регистър се добави числото шест (6). Стойността в адресния регистър A0 не се променя.

| **Инструкция** | **MOVE.W 6(A0), D0** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | $00001026<br>$00001027 | 11<br>22 | A0<br>D0 | 00001020<br>0000FFFF |
| След | $00001026<br>$00001027 | 11<br>22 | A0<br>D0 | 00001020<br>00001122 |

**9. Индиректна регистрова адресация с индекс и изместване**

Това е друга вариация на индиректната адресация, при която ефективният адрес се получава като сума от стойността на регистър за данни, изместването и от адреса в адресния регистър. Регистърът за данни играе ролята на 8-битов индексен регистър.

**[Ai]←[Ai] + изместване + [Di]**

**Пример 6.10.** Ефективният адрес в този пример е:
**[EA] = $10 + $100A + $2 = $101C**

Стойността в адресния регистър A2 се формира от данните (дума) на адреси $101C и $101D. Понеже инструкцията **MOVEA** в случая работи над данни с дължина 2 байта, то старшата част на адреса в адресния регистър A2 се установява в FFFF.

| **Инструкция** | **MOVEA $10(A0,D0.L), A2** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | $0000101C<br>$0000101D | EF<br>10 | A0<br>A2<br>D0 | 0000100A<br>00000000<br>00000002 |
| След | $0000101C<br>$0000101D | EF<br>10 | A0<br>A2<br>D0 | 0000100A<br>FFFFEF10<br>00000002 |

Индиректните индексни адресации с изместване и с изместване и индекс са удобни за достъп до редове и елементи на матрици, записани последователно в паметта или до елементи на група от еднотипни записи.

**Пример 6.11.** На адрес $2000 има таблица с N записа от по четири измервания (EST1, EST2, EST2 и EST4), всяко с големина 2 байта, както е дадено на фиг. 6.1.

Да се прочетат данните на измерване № 3 от запис № 5.

За целта може да използваме инструкцията :

**MOVE EST3(A0, D0), D5 ,**

където EST3 е изместването на третото измерване спрямо началото на всеки запис и е равно на 4.

Регистърът A0 сочи началото на таблицата с измервания (адрес $2000), а регистърът D0 = 4 * (4 * 2) = 32 = $20 сочи изместването на петия запис спрямо началото на таблицата.

Тъй като всеки запис има големина 8 байта (4 измервания по 2 байта), то началото на петия запис ще е след първите 4 записа с големина по 8 байта = 32 байта.

Третото измерване EST3 ще се намира отместено на 4 байта спрямо началото на всек запис.

| | |
|---|---|
| Регистър A0 | $2000 |
| Регистър D0 | $20 |
| Изместване | $4 |
| Адрес | $2024 |

| Записи | Стойност | Адрес |
|---|---|---|
| Запис 1 EST 1 | 1234 | $2000 |
| Запис 1 EST 2 | 1123 | $2002 |
| Запис 1 EST 3 | 3400 | $2004 |
| Запис 1 EST 4 | 3000 | $2006 |
| Запис 2 EST 1 | 1154 | $2008 |
| Запис 2 EST 2 | 1178 | $200A |
| Запис 2 EST 3 | 3500 | $200C |
| Запис 2 EST 4 | 2900 | $200E |
| Запис 3 | ..... | ..... |
| Запис 4 | ..... | |
| Запис 5 EST 1 | 1100 | $2020 |
| Запис 5 EST 2 | 1222 | $2022 |
| Запис 5 EST 3 | 3344 | $2024 |
| Запис 5 EST 4 | 2870 | $2026 |
| Запис 6 | ..... | ..... |
| Запис 7 | ..... | |
| ..... | | |

**Фигура 6.1.** Разположение на записите в паметта

### 10. Относителна адресация с изместване спрямо програмния брояч (PC)

При тази адресация към програмния брояч се добавя 16-битово изместване, за да се формира новият ефективен адрес (EA). Само първият операнд (източник) може да се адресира по този начин.

Основно правило при тази адресация е данните да са декларирани непосредствено след инструкцията за край на програмата, като само програмата има директива **ORG**. Така се осигурява позиционно-независим код на програмата, което гарантира, че тя и данните й може да се зареждат на произволни адреси в паметта.

Ако данните са формирани в отделен блок, адресацията пак може да се използва, но тогава само програмата може да бъде зареждана на произволно място в паметта, а данните й на фиксиран адрес или адреси, както е дадено в следващия пример.

**Пример 6.12.** Ефективният адрес се получава като от етикета COUNT се извади програмният брояч.

**$0FFE = $2000-$1000-$2**

```
1000              1          ORG       $1000
1000 3A3A 0FFE    2          MOVE.W    COUNT(PC), D5
2000              3          ORG       $2000
2000 ABCD         4   COUNT  DC.W      $ABCD
```

| Инструкция | MOVE.W COUNT(PC), D5 | | | |
|---|---|---|---|---|
| | Памет | | Регистър | |
| | Адрес | Съдържание | Име | Съдържание |
| Преди | $00002000<br>$00002001 | AB<br>CD | PC<br>D5 | 00001000<br>12345678 |
| След | $00002000<br>$00002001 | AB<br>CD | PC<br>D5 | 00001004<br>1234ABCD |

### 11. Относителна адресация спрямо програмния брояч (PC) с изместване и индекс

При тази адресация ефективният адрес се получава като изместване спрямо съдържанието на програмния брояч. Така 8-битовото изместване (индекс) от регистъра за данни се сумира с 8-битово изместване от етикета.

**Пример 6.13.**

$1010+$05-100A = $0B (или 11 байта спрямо PC)

```
00001000                  1        ORG       $1000
00001000 303C 0005        2        MOVE.W  #5, D0
00001004 6100 0004        3        BSR       SQU
00001008       4E75       4        STOP      #$2700
0000100A 103B 0004        5 SQU    MOVE.B TABLE(PC,D0.W), D0
0000100E 4E75             6        RTS
00001010 0001040517 19    7 TABLE  DC.B   0, 1, 4, 5, 23, 25
0000101A                  8        END
```

| Инструкция | MOVE.B TABLE(PC,D0.W), D0 | | | |
|---|---|---|---|---|
| | Памет | | Регистър | |
| | Адрес | Съдържание | Име | Съдържание |
| Преди | 00001015 | 19 | PC<br>D0 | 0000100A<br>ABCD0005 |
| След | 00001015 | 19 | PC<br>D0 | 0000100E<br>00000019 |

### 12. Пряка (директна) дълга адресация

При пряката дълга адресация първият операнд в инструкцията е директно число с големина до 4 байта.

Данните могат да се записват като:

- десетични (използва се префикс &, например &15 или 15);
- шестнадесетични (използва се префикс $, например $2F);
- осмични (използва се префикс @, например @67);
- двоични (използва се префикс %, например %10001111);
- ASCII (текст заграден с апострофи,‘This’).

**Пример 6.14.** Запис на константа $1FFFF в регистър D0.

| Инструкция | MOVE.L #$1FFFF, D0 | | | |
|---|---|---|---|---|
| | Памет | | Регистър | |
| | Адрес | Съдържание | Име | Съдържание |
| Преди | | | D0 | 12345678 |
| След | | | D0 | 0001FFFF |

**13. Пряка кратка адресация**

Това е бърза, оптимизирана форма на пряката адресация, чийто бинарен код, включително и операндът, се побира в една дума. Тази форма на адресация е валидна само за следните инструкции:

- **MOVEQ** (операндът трябва да е 8-битово цяло число със знак);
- **ADDQ** (операндът трябва да е число в интервала от 1 до 8 );
- **SUBQ** (операндът трябва да е число в интервала от 1 до 8).

Освен гореизброените инструкции има няколко инструкции като ADDI, SUBI и CMPI, които имат като първи операнд директно число.

**Пример 6.15.** Запис на 8-битовото число $1F в регистър D0.

| Инструкция | MOVEQ #$1F, D0 | | | |
|---|---|---|---|---|
| | Памет | | Регистър | |
| | Адрес | Съдържание | Име | Съдържание |
| Преди | | | D0 | 12345678 |
| След | | | D0 | 0000001F |

**Пример 6.16.** Събиране на числото $1000 с данните на адрес $2000.

| Инструкция | ADDI #$1000, $2000 | | | |
|---|---|---|---|---|
| | Памет | | Регистър | |
| | Адрес | Съдържание | Име | Съдържание |
| Преди | $2000<br>$2001 | 22<br>23 | | |
| След | $2000<br>$2001 | 32<br>23 | | |

**14. Косвена между специалните регистри**

При косвената регистрова адресация ефективният адрес (EA) е един от следните регистри: **CCR, SR, SP,PC**.

[EA] = **CCR** или **SR**, или **SP**, или **PC**

**Пример 6.17.**

| | | |
|---|---|---|
| MOVE.B | #$02, CCR | установява бит V на статус регистъра в единица. |
| MOVE.B | #$00, CCR | изчиства (нулира) всички битове на статус р-ра. |
| MOVE | D0, SP | задава стойност на стековия указател. |

| **Инструкция** | **MOVE.B D0, CCR** | | | |
|---|---|---|---|---|
| | **Памет** | | **Регистър** | |
| | **Адрес** | **Съдържание** | **Име** | **Съдържание** |
| Преди | | | D0 | 02 |
| След | | | CCR | 02 (бит V = 1 ) |

**Задачи:**

1. Определете типа на адресацията на всяка от следните инструкции:

```
SUB     D5,D7
MOVE    D4,A6
MOVE    A3,D2
ADDQ    #$121,D2
MOVE    #$11,D1
MOVE    #1,CCR
MOVE    CCR, D3
ADD     TIME, D3
SUB     MASK,(A3)
MOVE    (A1)+,(A2)+
ADDX    -(A1),-(A2)
ADD     4(A6), D3
MULU    $16(A5,D3), D7
```

2. Какво е грешно в следните инструкции и защо:

```
LEA     (A5)+, A6
MOVE    A3+,D0
MOVE.B  #400,D6
MOVE    $4000,A9
ADD.W   (A2)-,D3
SUB     $800, D8
ADD.B   #1, A3
ADDI    SUM,D4
MOVE.B  (PC,D0.W) TABLE, D0
BEQ     $FFAAA8
CMPM    MEM1,MEM2
ADDQ    #$11223344,D0
```

3. Какъв ще бъде разултатът след изпълнение на следните инструкции:

а) **MOVE #10, D3**
**MULU D3,D3**

б) **SUBA A5,A5**

в)
```
    ORG     $6000
TE  DC      $ABCD
    ORG     $7000
    MOVEA.L TE,A0
    MOVEA.L #TE,A1
    LEA     TE,A0
    LEA     #TE,A0
    LEA     16(TE),A2
```

4. Дайте пример за инструкция, с която към регистъра D4 се добавя числото, посочено от адресен регистър A3.

5. Дайте пример за инструкция, с която към регистъра D6 се добавя числото, посочено от адресен регистър A2, като адресът в адресния регистър се увеличава с 2.

6. Дайте пример за инструкция, с която към регистъра D3 се добавя числото, посочено от адресен регистър A6, като адресът в адресния регистър се увеличава с 4.

7. Дайте пример за инструкция с която от регистъра D2 се изважда със заем числото, посочено от адресен регистър A6, като адресът в адресния регистър се увеличава с 4.

8. Дайте пример за инструкция, с която адресът в адресния регистър A1 се намалява с 4 и числото, сочено от регистъра се изважда от регистъра D2 със заем.

9. Дадена е таблица с 52 записа (по един за всяка седмица от годината). Всеки запис е с по 3 измервания от по 4 байта. Началният адрес на таблицата е на $40000. Дайте пример за инструкция, с която да се прочете първото измерване от записа на седмица с номер12.

10. Дадена е таблица с 52 записа (по един за всяка седмица от годината). Всеки запис е с по 3 измервания от по 4 байта. Началният адрес на таблицата е на $40000. Дайте пример за инструкция, с която да се прочете първото измерване от записа на седмица с номер 12.

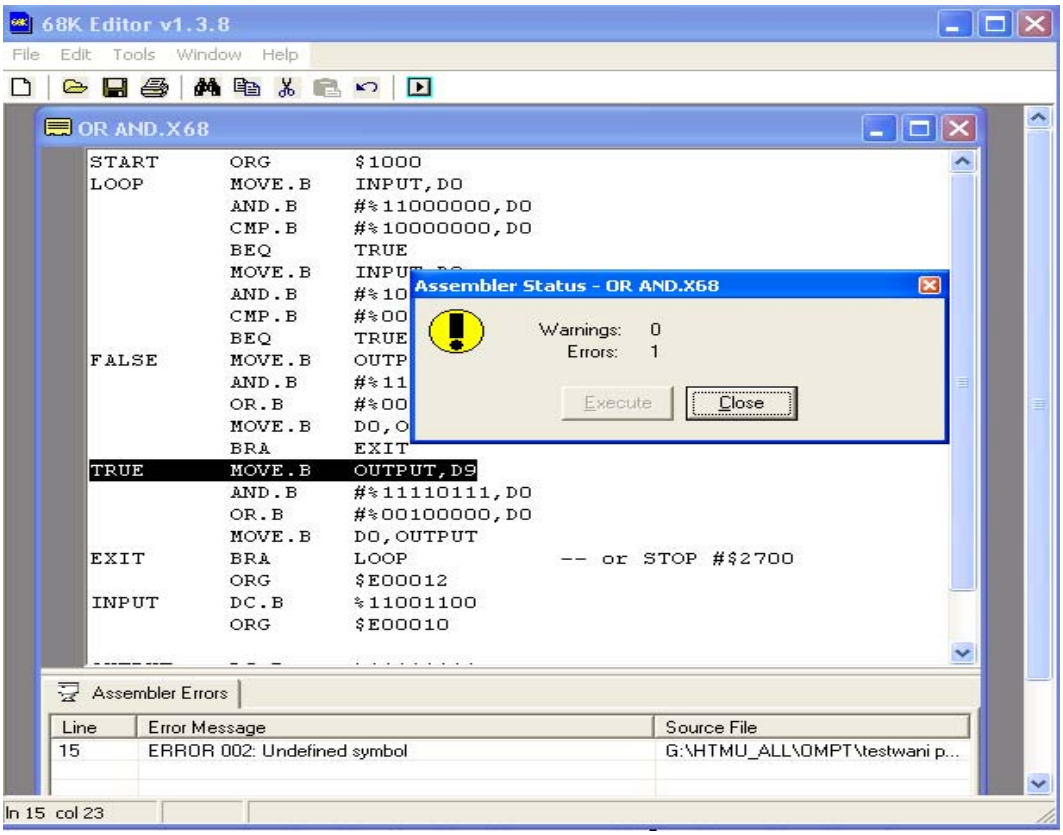
11. Дадена е таблица с 30 записа (по един за всеки ден от месеца). Всеки запис е с по 5 измервания от по 2 байта. Началният адрес на таблицата е на $80000. Дайте пример за инструкция, с която да се прочете третото измерване от записа на ден с номер 16.

# 7. Симулационна среда за програмиране на асемблер за MC68000

**Описание и функции на симулационната среда**

Специализираната развойна среда Easy68K служи за разработка и тестване на програми на асемблер за MC68000. Тя включва два модула – текстов редактор и симулатор. Чрез текстовия редактор се създава програмният код на асемблер и се генерира изпълним код. Чрез симулатора изпълнимият код може да бъде изпълнен еднократно или постъпково.

Симулаторът визуализира съдържанието на всички регистри за данни и адреси, както и съдържанието на програмния брояч, на статус регистра и на стека. На фиг. 7.1 е показан редакторът за създаване на програми на асемблер (EDIT68K.exe).




**Фигура 7.1. Редактор на развойната среда Easy68K**

В менюто си (фиг. 7.2) той включва стандартните за всички приложения на Windows опции File, Edit, Tools, Window Help, както и лентата (тулбар) с най-често използваните бутони.

**Фигура 7.2. Меню и лента с бутони на редактора на 68K**

При стартиране на симулатора автоматично се генерира програмен файл, който за удобство на потребителя съдържа шаблон на примерна програма, както е дадено по долу.

```
*-----------------------------------------------------------
* Program    :
* Written by :
* Date       :
* Description:
*-----------------------------------------------------------
          ORG     $1000
START:                          ; first instruction of program
          STOP    $#2700                ; halt simulator
* Variables and Strings
          END     START                 ; last line of source
```

Потребителят може да въведе собствената си програма от реда, започващ с етикета `START:` до реда `STOP`.

В началната секция (която не е задължителна) може да се въведат име на програмата, авторът й, датата на създаване или модифициране и кратко пояснение за предназначението на програмата.

В секцията за променливите (след **STOP**) се декларират всички променливи, които са използвани в програмата.

При писане на програми трябва да се отчитат изискванията на компилатора, а именно:

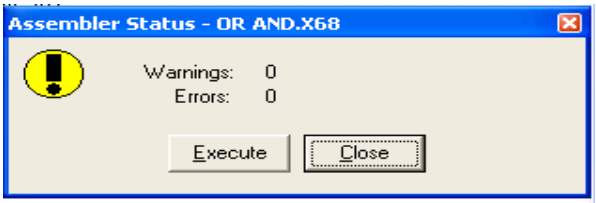
- първата колона от 7 символа е за етикетите, използвани в програмата;
- втората колона е за името на инструкцията;
- третата за операнда или операндите;
- четвъртата е за коментари.

Преход от една колона в друга става чрез клавиша Tab –>|.

Ако тези изисквания не се спазят, компилаторът генерира съобщения за грешки. Например, ако инструкциите се пишат в първа колона, то те се интерпретират като етикети, а операндите им като инструкции и т.н.

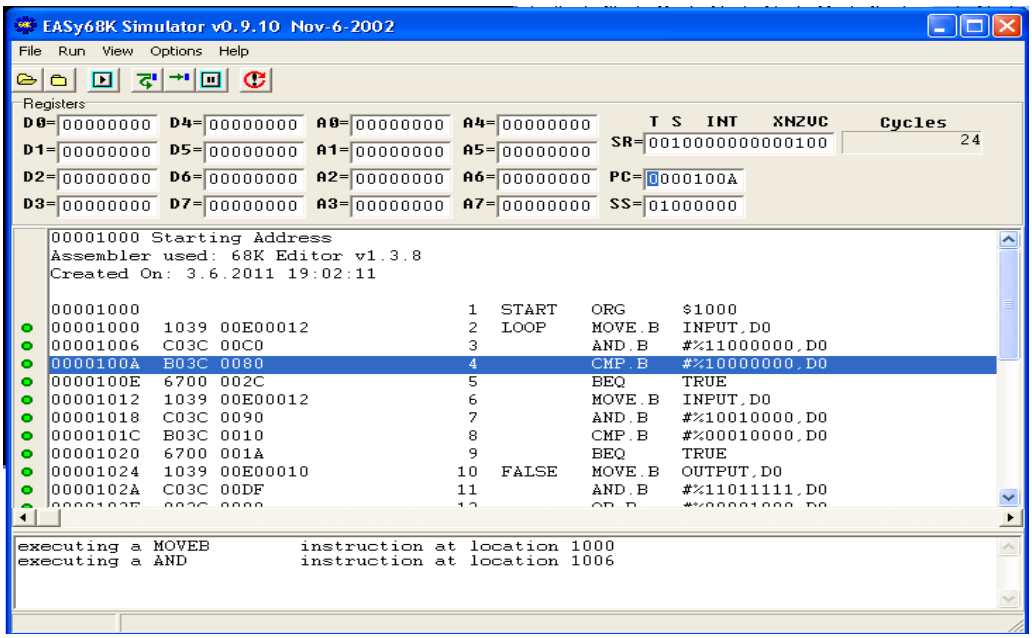
При натискане на бутона програмата се компилира и се генерират съобщения за грешки, ако има такива (например, както е дадено на фиг. 7.1 по-горе).

След корекция на всички грешки бутонът Execute (фиг. 7.3) става активен и програмата може да се изпълни.



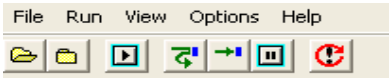

**Фигура 7.3.** Бутон Execute за изпълнение на програмата

Чрез натискането на бутона Execute, се стартира симулаторът на асемблера на MC68000 (EASy68K.exe), даден на фиг. 7.4.




**Фигура 7.4. Симулатор на програми на 68000**

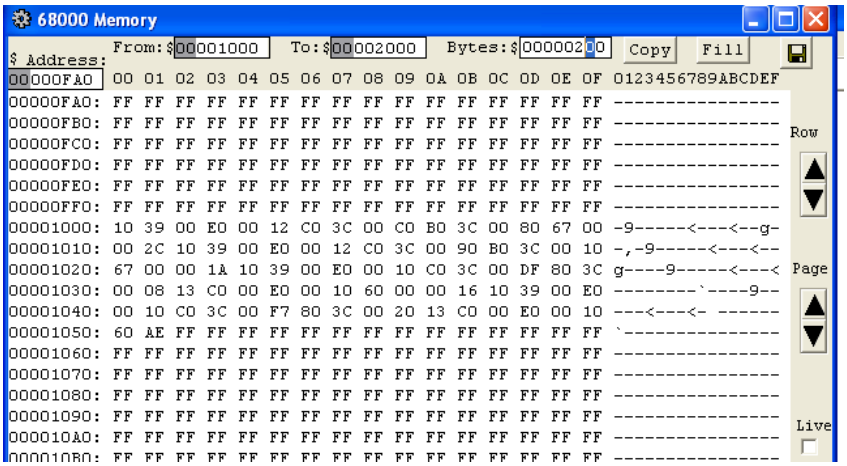
Симулаторът има меню със опции File, Run, View, Options и Help, както и бутони, показани на лентата с бутоните (toolbar) по-долу.




Чрез опцията File могат да се зареждат (отварят) и други файлове на вече компилирани програми.

Чрез опцията Run програмите могат да се стартират или да се проиграват постъпково.

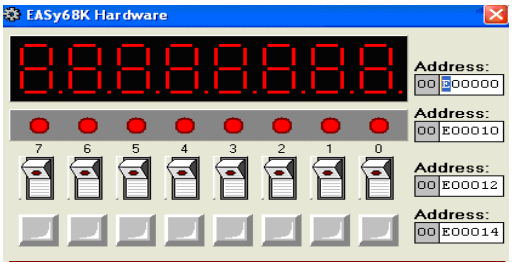
Опцията View дава възможност за преглед на съдържанието на желани адреси от паметта (фиг. 7.5), достъп до стека, до прозореца за изходни данни (ако се използват прекъсвания за писане на екрана) и достъп до хардуерния интерпретатор.




**Фигура 7.5. Достъп до клетки (адреси) от паметта**

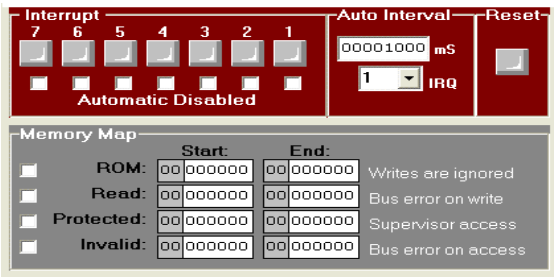
Хардуерният модул, представен на фиг 7.6, съдържа:

- 8-цифров седемсегментен дисплей (от адрес $E00000 до $E00008);
- осем цифрови изхода, асоциирани с един байт на адрес $E0000E;
- осем цифрови входа, управлявани от ключове асоциирани с битовете на байт $E00012;
- ключове за установяване (ресетване) на определени битове асоциирани с байт $E00014.




**Фигура 7.6. Хардуерен интерпретатор**

Модулът дава възможност за контрол (генериране) на прекъсванията (фиг. 7.7), при натискане на съответен бутон или за автоматична генерация на такива. Прекъсванията са дискутирани в лекционния курс на дисциплината „Микропроцесорна техника”.




**Фигура 7.7. Хардуерен интерпретатор (прекъсвания)**

Хардуерният модул позволява симулиране на запис в различни типове памет, както и на адреси извън 24-битовото адресно пространство на MC68000.

Чрез Options могат да се правят настройки по отношение на генерацията на изпълним код.

Интегрираната развойна среда е достъпна за зареждане на адрес www.easy68k.com. Тя е безплатна и представлява софтуер с отворен код.

**Задачи:**

1. Въведете и изпълнете постъпково следната програма:

```
                ORG     $1000
START           Move    #5,D0
                add     #6,D0
                MOVE    D0, X
                STOP    #$2700
X               DS      1
                END     START
```

а) Анализирайте съдържанието на регистър D0 и на клетка X от паметта.

б) Кой е адресът на клетка X?

в) На кой адрес е последната изпълнима инструкция?

г) Каква е стойността на програмния брояч след изпълнение на програмата?

д) Вижте какъв е ефектът от замяната на X латинско с X на български.

е) Променете инструкцията **add #6, D0** на **add #$80, D0** и изпълнете отново програмата. Анализирайте промяната в View / Hardware.

ж) По време на постъпковото изпълнение на програмата анализирайте коя инструкция за колко машинни цикъла се изпълнява.

2. Има ли значение с какви букви (малки и/или големи) се изписват инструкциите и операндите на асемблер?
3. Има ли значение в коя колона се записват инструкциите?
4. Коя е колоната за коментари и коя е за етикети?
5. В програмата от задача 1 добавете следния ред:

**ORG $E00010**

след инструкцията STOP и изпълнете програмата пак. Използвайте опциятаView/Hardware и вижте коя лампа свети и коя не.

6. В програмата от задача 1 добавете следния ред:

**ORG $2000**

след инструкцията **STOP** и изпълнете програмата пак. Използвайте опцията View/Memory и вижте съдържанието на паметта от адрес $1000 до $1100, както и на адрес $2000.

7. Как се активира хардуерният модул?
8. Кои са адресите на елементите на седмсегментния дисплей?

# 8. Разработка на линейни програми на асемблер за MC68000.

Разработката на линейни програми е свързана главно с използването на инструкции, които се изпълняват последователно и безусловно една след друга (фиг. 8.1) Такива инструкции на MC68000 са аритметичните инструкции, инструкциите за преместване на данни или тези за изместване и ротация на данни.

**Фигура 8.1. Блокова схема на линейна програма**

**Пример 8.1.** Да се състави програма за изчисляване на стойността на израза:

$$Z = (X^2 + Y^2) / (X-Y).$$

При решението на задачата съдържанието на клетка X се зарежда в регистър D0, след което се умножава само по себе си и се формира квадрата на X. Аналогично, съдържанието на Y се зарежда в регистър D1 и се повдига на квадрат. Получените междинни резултати се събират, като се формира числителят в D0. Използвайки регистър D2 и операцията по изваждане се формира и знаменателят. Крайният резултат се получава от деленето на D0 на D2 и същият се записва на адрес Z.

```
START   ORG     $1200     начало на програмата на адрес $1200
        MOVE    X, D0     зарежда числото от адрес X в D0
        MULU    D0, D0    умножава D0=D0 * D0 (изчислява X² )
        MOVE    Y, D1     зарежда числото от адрес Y в D1
        MULU    D1, D1    умножава D1=D1 * D1(изчислява Y² )
        ADD     D1, D0    събира D1 + D0(X² + Y²)резултат в D0
        MOVE    X, D2     зарежда числото от адрес X в D2
        SUB     Y, D2     изважда число от адрес Y от D2
                          резултат в D2  (X-Y)
        DIVU    D2, D0    разделя D0 на D2 резултат в D0
                          =(X² + Y²)/(X-Y)
        MOVE.W  D0,Z      записва резултатът от D0
                          в  клетка Z
        STOP    #$2700
        ORG     $2000     резервира памет от адрес $2000
   X    DC.W    50        декл. константа X на адрес$2000
   Y    DC.W    20        декл. константа Y на адрес$2002
   Z    DS.W    1         резервира 2 байта за резултата
                          на адрес $2004
        END     START
```

**Пример 8.2.** Да се състави програма за изчисляване на площта и периметъра на правоъгълник със страни X и У.

По аналогия с предишната задача съдържанието на клетки X и Y се премества в регистри D0 и D1. След събирането на данните от двата регистъра се формира полупериметърът, който после се умножава директно по 2 (#2) и се получава периметърът на правоъгълника в D1. Резултатът от D1 се записва в клетка с име PERIM. Отново D1 се зарежда със стойността на Y, защото D1 към момента съдържа периметъра. След умножението на D0 и D1 се формира площта на правоъгълника, която от D1 се записва в клетка с име PLOSHT.

```
        ORG        $1000
        MOVE.W X,  D0            зареждане на X в D0
        MOVE.W Y,  D1            зареждане на Y в D1
        ADD.W      D0, D1        събиране на X и Y -
                                  полупериметъра
        MULU       #2, D1        приметър = 2 * полупериметър
        MOVE.W     D1, PERIM     записване на резултата в клетка
                                 PERIM
        MOVE.W     Y, D1         зареждане наново на Y в D1
        MULU       D0, D1        площ = X * Y резултат в D1
        MOVE.W     D1, PLOSHT    записване на резултата в PLOSHT
        STOP       #$2700
    X   DC.W       25
    Y   DC.W       47
PERIM   DS.W       1
PLOSHT  DS.W       1
        END        $1000
```

**Пример 8.3.** Да се състави програма за събиране две 64-битови числа M1 и M2. Числата са разположени последователно в паметта като младшите им части M1Low M2Low са разположени на по-големи адреси в паметта, а старшите им части M1High и M2High на по-малки адреси (виж табл. 8.1). Резултатът да се запише след тях в клетки ResLow и ResHigh. При събирането да се отчита преносът от по-младшите към по-старшите части на числата. За целта да се използва инструкцията **ADDX**.

**Таблица 8.1. Разположение на числата в паметта**

| име | големина | адрес | стойност |
|---|---|---|---|
| M1High | 32 бита | $2000 | $00000001 |
| M1Low | 32 бита | $2004 | $FFFFFFFD |
| M2High | 32 бита | $2008 | $00000001 |
| M2Low | 32 бита | $200C | $00000005 |
| ResHigh | 32 бита | $2010 | |
| ResLow | 32 бита | $2014 | |

```
START   ORG     $1000
        LEA     M1Low,A0     зареждане на адреса на младшата
                             част на първото число
        LEA     M2Low,A1     зареждане на адреса на младшата
                             част на резултата
        LEA     ResLow,A2    зареждане на адреса на младшата
                             част на второто число
        MOVE    #$00, CCR    изчистване на статус регистъра CCR
        MOVE.L  (A0), D0     младшата част на число едно в D0
        ADD.L   (A1), D0     събиране на младшите части на
        SUBA    #4,A0        формиране на адреса на старшата
                             част на първото число
        SUBA    #4,A1        формиране на адреса на старшата
                             част на второто число
        MOVE.L  (A0), D1     старшата част на първото число в D1
        MOVE.L  (A1),D2      старшата част на второто число в D2
        ADDX.L  D2, D1       събиране на старшите части с пренос
        MOVE.L  D0, (A2)     запис на младшата част на резултата
                             от D0 на адрес A2
        SUBA    #4,A2        формиране на адреса на старшата
                             част на резултата
        MOVE.L  D1,(A2)      запис на старшата част на резултата
                             от D1 на обновения адрес A2
        STOP    #$2700
        ORG     $2000
M1High  DC.L    $00000001
M1Low   DC.L    $FFFFFFFD
M2High  DC.L    $00000001
M2Low   DC.L    $00000005
ResHigh DS.L    1
ResLow  DS.L    1
        END     START
```

**Задачи:**

1. Какъв би бил резултатът от пример 8.3, ако вместо инструкцията **ADDX** се използва инструкцията **ADD**?

2. Да се състави програма за изчисляване на израза $Z = (4 * P^3 + K^3) / (X^2-Y^2)$, където $P = 20$, $K = 16$, $X = 8$, $Y = 6$.

3. Изчислете обема на пирамида с правоъгълна основа и страни A и B и височина H, където A = 25, B = 47, H = 88.

4. Изчислете обема и околната повърхнина на паралелепипед с правоъгълна основа и страни A и B и височина H, където A = 76, B = 27 и H = 84.

5. Модифицирайте програмата от задача 2 да работи с инструкции с големина байт (B) и проверете какъв е резултатът в клетка Z.

6. Изчислете повърхността на куб със страна A = 15 см.

7. Изчислете повърхността на пирамида със страни на основата A = 10, B = 14 и височина на околната стена H = 25.

8. Изчислете израза от задача 2, като вместо умножение по 4 на $P^3$ използвате инструкцията за изместване на ляво LSL #2 на регистъра, съдържащ междинния резултат $P^3$.

9. Какъв ще е резултатът, ако в регистър D0 заредим числото 48 и след това използваме инструкциите :

LSR #2,D0

LSL #3, D0

RSR #4, D0

RSL #4, D0

10. Изчислете израза: Y = 4 * A - 8 * B + C / 16, като вместо инструкциите за деление и умножение използвате инструкциите за преместване на данни. Стойностите на променливите A, B и C са: A = 40, B = 8 и C = 192.

11. Напишете програма за изваждане на две 64-битови числа, като се отчита евентуален заем от по-старшия разряд.

# 9. Разработка на разклоненни програми на асемблер за MC68000.

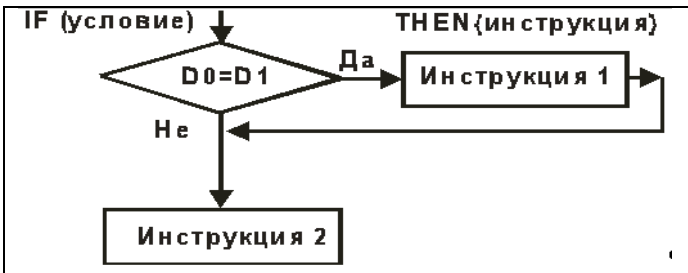
В много програми се налага изпълнението на една инструкция или група от инструкции в зависимост от изпълнението или неизпълнението на някакво условие (фиг. 9.1 и фиг. 9.2). Такива програми се наричат разклонени и за разработката им се използват инструкциите за сравнение CMP на две числа и за разклонение ***Bxx***, където ***xx*** е абревиатура на операциите по сравняване на две числа. По-подробна информация за всички възможни инструкции за преход е дадена в таблица 9.1 по-долу.

Инструкцията **CMP** сравнява числата, намиращи се в операнд 1 и операнд 2, и установява един или група от битове (Z, N, X, V) в статус регистра (**CCR / SR**).

В примера по-долу, ако две числа, намиращи се в регистрите D0 и D1, не са равни (бит Z е равен на 0), се прави преход към инструкция с етикет InstR2. Това става с инструкцията **BNE InstR2**.

Ако числата са равни, се изпълнява инструкцията с етикет InstR1, непосредствено след BNE.

Така се реализира разклонение в програмата, аналогично на инструкцията непълен if от езика за програмиране C++.

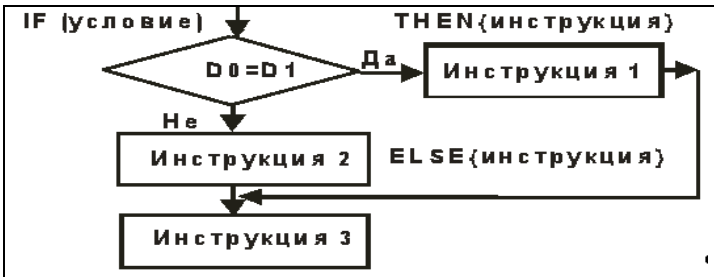



**Фигура 9.1.** Условен преход (непълен IF)

**Пример 9.1.**

```
        CMP   D1,D0
        BNE   InstR2
InstR1  ADD   #1, D1
InstR2  MOVE  D1, D5
Next    MULU  D5, D5
```

В следващия пример (фиг. 9.2) е дадено сравняване на две числа, намиращи се в регистрите D0 и D1. Ако числата са равни (бит Z е равен на 1), с инструкцията **BEQ InstR1** се прави преход към инструкция с етикет InstR1. Ако числата не са равни, се изпълнява инструкцията с етикет InstR2, непосредствено след **BNE**. След изпълнението на инструкцията с етикет InstR2 се прави безусловен преход (**BRA**) към инструкция с етикет Next. Така се реализира разклонение в програмата, аналогично на инструкцията пълен **if** от C++.




**Фигура 9.2. Условен преход (пълен IF)**

**Пример 9.2.**

```
        CMP    D1,D0
        BEQ    InstR1
InstR2  MOVE D0, D5
BRA     Next
InstR1  MOVE  D1, D5
Next    MULU D5, D5
```

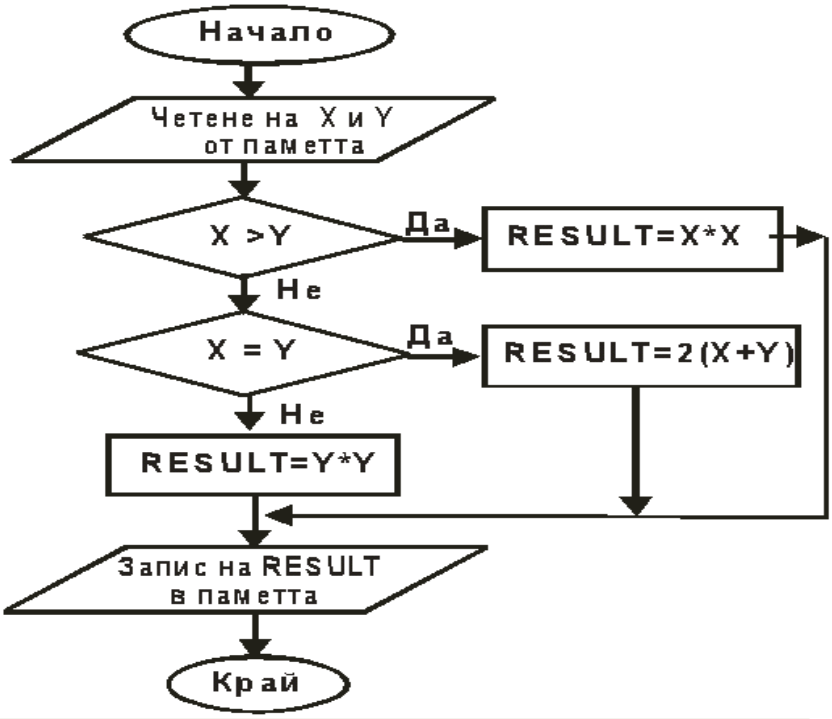
**Таблица 9.1. Инструкции за условен и безусловен преход**

| **Инструкция** | **Преход при:** | **Преход при флаг/флагове на CCR** |
|---|---|---|
| **BCC** | бит за пренос C = 0 | C.= 0 |
| **BCS** | бит за пренос C = 1 | C = 1 |
| **BNE** | неравенство | Z = 0 |
| **BEQ** | равенство | Z = 1 |
| **BPL** | положителен резултат | N = 0 |
| **BMI** | отрицателен резултат | N = 1 |
| **BHI** | по-голямо | -Z + -C = 1 |
| **BLS** | по-малко | C + Z = 1 |
| **BGT** | по-голямо | (V N(-Z)) + (-(ZNV)) = 1 |
| **BLT** | по-малко | ((-V) N) + (V(-N)) = 1 |
| **BGE** | по-голямо или равно | ((-V) N) + (V(-N)) = 0 |
| **BLE** | по-малко или равно | Z+(NV + VN) = 1 |
| **BVC** | когато няма препълване | V = 0 |
| **BVS** | препълване | V = 1 |
| **BRA** | Безусловен преход (goto) | |
| **JMP** | Безусловен преход (goto) | |

**Пример 9.3.** Следната програма на асемблер сравнява две числа X и Y. Ако X е по-голямото число, в клетка с име RESULT се записва резултат, равен на X на квадрат. Ако Y е по-голямото число, резултатът е равен на Y на квадрат, а ако числата са равни, то резултатът е равен на 2 * (X + Y).

Числата X и Y се зареждат в регистрите за данни D0 и D1 и се сравняват с инструкцията **CMP**, която установява флагове в регистъра за състоянията. На основата на стойностите на тези флагове се прави условен преход с инструкциите **BPL** или **BEQ** към етикети BIGGER или EQUAL**.**

С инструкциите **BRA LAST** се прави безусловен преход към края на програмата. След програмата на асемблер е представен и кореспондентният ѝ код на езика C++.




**Фигура 9.3. Блок схема на програмата от пример 9.3**

```
START     ORG    $1000
          MOVE.W X,D0        зареждане на X в D0
          MOVE.W Y,D1        зареждане на Y в D1
          CMP.W  D1,D0      сравнение на  X  с Y
          BGT    BIGGER     ако X > Y преход към адрес BIGGER
          BEQ    EQUAL      ако X = Y преход към адресEQUAL
          MULU   D1,D1      ако X < Y повдигни Y на квадрат
          MOVE.L D1,RESULT  запиши резултата в RESULT
          BRA    LAST       преход към края на програмата
BIGGER    MULU   D0,D0      ако X > Y повдигни X на квадрат
          MOVE.L D0, RESULT     запиши резултата в RESULT
          BRA    LAST       преход към края на програмата
EQUAL     ADD    D0,D1      ако X = Y,то RESULT=2*(X+Y)
          MULU   #2,D1
          MOVE.L D1, RESULT     запиши резултата в RESULT
LAST      STOP   #$2700
          ORG    $1500
X         DC.W   25
Y         DC.W   47
RESULT    DS.L   1
          END    START
```

Код на същата програма на езика C++ :

```
if (X > Y) {          // X  е по-голямо число

        RESULT = X * X;
}
else  {         // Y е по-голямото число

        if (X < Y) {
                RESULT = Y * Y;
        }
        else  {       //  X = Y
                RESULT = 2 * (X + Y);
        }
}
```

**Пример 9.4.** Управление на цифрови изходи на базата на данните, постъпващи на цифровите входове.

Програмата по-долу сканира цифровите входове (битове P, Q и S) на входния байт INPUT (таблица 9.2) и в зависимост от техните стойности управлява стойностите на изходните битове C и E на изходния байт OUTPUT. Ако е изпълнено едно от условията – бит P да е равен на единица (1) и бит Q да е равен на нула (0) или P да е равен на нула (0) и S да е единица (1), то тогава бит C се установява в единица (1), а бит E в нула (0). В противен случай C се нулира, а бит E се установява в единица. Всички останали битове в изходния байт не променят стойностите си.

**Таблица 9.2. Битове на входния и изходния байт**

| INPUT | P | Q | R | S | T | U | V | W |
|---|---|---|---|---|---|---|---|---|
| OUTPUT | A | B | C | D | E | F | G | H |

```
START   ORG       $1000
LOOP    MOVE.B    INPUT, D0
        AND.B     #%11000000, D0
        CMP.B     #%10000000, D0
        BEQ       TRUE
        MOVE.B    INPUT, D0
         AND.B    #%10010000, D0
        CMP.B     #%00010000, D0
        BEQ       TRUE
FALSE   MOVE.B    OUTPUT, D0
        AND.B     #%11011111, D0
        OR.B      #%00001000, D0
        MOVE.B    D0, OUTPUT
        BRA       EXIT
```

```
TRUE    MOVE.B   OUTPUT, D0
        AND.B    #%11110111, D0
        OR.B     #%00100000, D0
        MOVE.B   D0, OUTPUT
EXIT    BRA      LOOP   -- или STOP #$2700
       ORG       $2000
INPUT   DC.B     %11001100
OUTPUT  DC.B     %11111111
        END      START
```

Реципрочен код на програмата на езика C++:

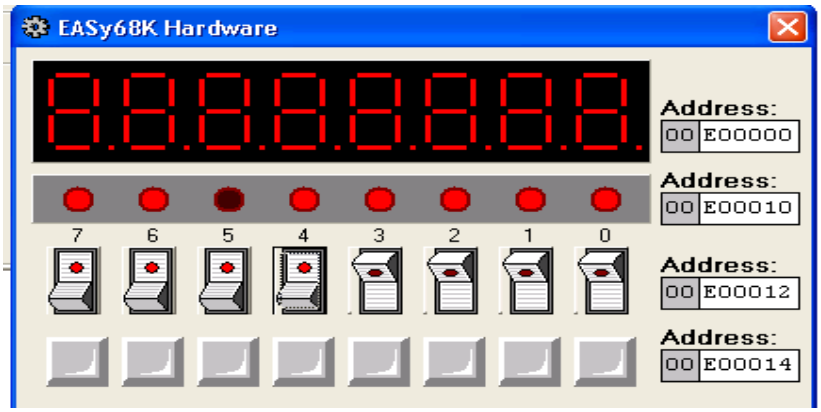
**if ( (P==1) && (Q==0)) || ((P==0) && (S==1) )**

**{ C = 1; E = 0; }**

**else**

**{ C = 0; E = 1; };**

Ако програмата се модифицира, като за входния и изходния (INPUT OUTPUT) байтове се използват адресите на аналогичните входни и изходни байтове ($E000010 и $E000012) на хардуерния модул (фиг. 9.4), интегриран в симулационната среда на EasySim68K, е възможно тестване на основата на включване/изключване на входните битове и анализиране на изходните, които са асоциирани със светлинна индикация.




**Фигура 9.4. Хардуерен модул – адреси и асоциирани битове и бутони**

**Пример 9.5.** Извеждане на числата от 0 до 9 чрез седемсегментен код.

Числото се извежда чрез използване на хардуерния модул, интегриран в симулационната среда на EasySim68K. Този модул визуализира байтовете, записани на адреси от $E000000 до $E0000007, като седемсегментен код. В програмата по-долу числото, което ще се визуализира, се записва в регистър D0. За да се определи кой 7-сегментен

код да се генерира, се използва последователност от инструкции **CMP** и **BEQ**, определящи кое число е въведено. Тази група от инструкции е аналог на инструкцията **switch/case** от езика C или C++.

**Фигура 9.5. Резултат от изпълненние на програмата**

Резултатът от изпълненнието на програмата е изписването на числото 5 (най-ляво на фиг. 9.5). Останалата част от седемсегментния дисплей съдържа осмици, защото всички байтове от паметта на симулатора по подразбиране са установени в FF.

```
START   ORG     $1000
        MOVE.B  #5,D0
        CMP     #0,D0
        BEQ     GOTO0
        CMP     #1,D0
        BEQ     GOTO1
        CMP     #2,D0
        BEQ     GOTO2
        CMP     #3,D0
        BEQ     GOTO3
        CMP     #4,D0
        BEQ     GOTO4
        CMP     #5,D0
        BEQ     GOTO5
        CMP     #6,D0
        BEQ     GOTO6
        CMP     #7,D0
        BEQ     GOTO7
        CMP     #8,D0
        BEQ     GOTO8
        CMP     #9,D0
        BEQ     GOTO9
        BRA     EXIT
GOTO0   MOVE.B  #%00111111,OUTPUT
        BRA     EXIT
GOTO1   MOVE.B  #%00000110,OUTPUT
        BRA     EXIT
GOTO2   MOVE.B  #%01011011,OUTPUT
        BRA     EXIT
GOTO3   MOVE.B  #%01001111,OUTPUT
        BRA     EXIT
GOTO4   MOVE.B  #%01100110,OUTPUT
        BRA     EXIT
```

```
GOTO5   MOVE.B  #%01101101,OUTPUT
        BRA     EXIT
GOTO6   MOVE.B  #%01111101,OUTPUT
        BRA     EXIT
GOTO7   MOVE.B  #%00000111,OUTPUT
        BRA     EXIT
GOTO8   MOVE.B  #%01111111,OUTPUT
        BRA     EXIT
GOTO9   MOVE.B  #%01101111,$E00000
EXIT    STOP    #$2700
        ORG     $E00000
OUTPUT  DS.B    1
        END     START
```

**Задачи:**

1. Изчислете Y = F(X) по следната зависимост :
   $Y = 7 * X^2 - 45$, ако $X < 5$
   $Y = 9 * X^3 - 45$, ако $X = 5$ и
   $Y = 5 * X + 8$, ако $X > 5$
   Тествайте програмата за стойности на X 1, 2, 5 и 7.

2. Изчислете Y = F(X) по следната зависимост :
   $Y = X^4 + A$, ако $X <= 4$,
   $Y = (12 * X + B)/( X + 6)$, ако $X > 4$.
   За стойности на A = 123, B = 13

3. Модифицирайте програмата от пример 9.5 да изписва Err в байтовете с адреси E000000, E000001 и E000002, ако числото, записано в D0 е по-малко от 0 или по-голямо от 9. За целта използвайте информацията за 7-сегментния код.

4. На базата на програмата от пример 9.4 напишете програма, която да установява битове A, B и C в 1-ца, ако U, V и W са със стойност 0 и обратно.

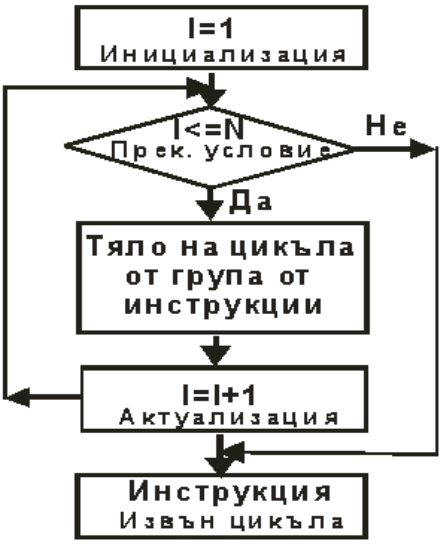
5. Съставете програма, която да повдига числото в регистър D0 на степен от 1 до 5 в зависимост от числото в регистър D1, което може да има стойности от 1 до 5. За всички останали стойности на D1 резултатът да е 0.

6. Изчислете израза $Z = (4 * P^3 + K^3) / (X^2 - Y^2)$, само ако X е по-голямо от Y. В противен случай Z = -1.

# 10. Разработка на циклични и сложни програми на асемблер за MC68000

При цикличните програми една инструкция или група от инструкции се изпълняват определен брой пъти (циклично), докато променливата (броячът) на цикъла не приеме определена стойност, наречена прекъсващо условие. При достигане на тази стойност цикълът се прекратява и се преминава към следващата инструкция извън цикъла. Известни са два вида цикли – с предусловие (фиг.10.1) и със следусловие (фиг. 10.2), при които в единия случай се проверява дали променливата на цикъла е достигнала граничната си стойност в началото на изпълнение на цикъла, а във втория случай – в края на цикъла.

В MC68000 няма специални инструкции за цикъл. Циклите с пред- и следусловие се реализират на основата на инструкциите за сравненение, условен и безусловен преход, коментирани в предходната глава.




**Фигура 10.1. Цикъл с предусловие**

**Фигура 10.2. Цикъл със следусловие**

Една специфична инструкция, ориентирана към цикли със следусловие, е инструкцията **DBRA XX, NEXT**, която декраментира съдържанието на (XX), където XX е или регистър за данни или клетка от паметта и проверява дали стойността му е равна на -1 (минус единица). Ако тази стойност не е достигната, се прави преход към етикета NEXT. Чрез тази инструкция могат да се организират цикли с променлива на цикъла, изменяща се от N - 1 до 0.

В този случай променливата на цикъла трябва да се инициализира с число, което е с единица по малко от броя на повторенията на инструкциите в цикъла. Инструкцията **DBRA** съвместява актуализацията (XX = XX-1) и прекъсващото условие (XX <= 0).

**Пример 10.1.** Да се намери сумата от квадратите на числата от масива X, който има 10 елемента.

$$SUMA = \sum_{i=1}^{N} X_i^2$$

Кореспондентният код на същата програма на езика C++ е:

```
SUMA = 0;
for (int i = 0;  i <= 9; i = i + 1)
{
    SUMA = SUMA + X[i] * X[i];
}
```

Кодът на същата програма на асемблер за MC68000 е даден по-долу. В програмата са използвани регистрите за данни D0, D1 D2 и адресният регистър A0, като в D0 се натрупва сумата, D1 е брояч, аналог на i от цикъла for (вж. по-горе), а в D2 се съхранява текущата стойност на X[i] и квадрата му, преди да се натрупа към сумата. Адресният регистър A0 служи за указател към първия елемент на масива X. Инстукцията **LEA** зарежда адреса на масива X в A0. След натрупване на сумата от квадратите и излизане от цикъла, когато стойността на брояча D1 стане равна на 10, резултатът от D0 се записва в клетка от паметта с име SUMA.

```
START      ORG     $1000
           LEA     X, A0          зарежда адреса на масива X в A0
           MOVE.W  #0, D0         сума за SUMA = 0 използва рег D0
           MOVE.W  #0, D1         брояч на цикъла i = 0 (D1 = 0)
NEXT       MOVE.W  (A0)+, D2      D2 = X[i] и увеличава
                                   адреса с единица, така A0 сочи
                                   следващия елемент от масива X
           MULU    D2,D2          повдига X[i] на квадрат
           ADD.W   D2,D0          добавя го към сумата D0
           ADD.W   #1, D1         брояч = брояч + 1 (D1=D1+1)
           CMP.W   N,D1           брояч < N (D1 < 10?)
           BNE     NEXT           ако броячът (D1 < 10) е различен
                                   От N се прави преход към NEXT
           MOVE.W  D0, SUMA       записва резултата в клетка SUMA
           STOP    #$2700
N          DC.W    10             брой елементи на масива X
X          DC.W    7, 8,12,23,32,11,56,2,7,8  стойности на
                                   масива X
SUMA       DS.W    1              резервирана памет за SUMA
           END     START
```

**Пример 10.2.** Да се намери скаларното произведение на два вектора VEC1 и VEC2 по формулата $SUM = \sum_{i=1}^{N} VEC1_i * VEC2_i$ , като се има предвид, че данните (елементите на векторите) са двубайтови числа.

Броят на елементите на всеки вектор е N и е равен на 10.

Кореспондентният код на същата програма на езика C++ е:

```
SUM = 0;
for (i = 0; i <  N; i++)
{
SUM = SUM + VEC1[i] * VEC2[i];
}
```

В програмата е използвана индиректна индексна адресация с офсет, чрез която достъпът до елементите на двата масива (векторите VEC1 и VEC2) става на базата на адреса на първите им елементи. В случая това са офсетите VEC1 и VEC2, а обхождането на елементите на векторите става чрез актуализиаране на стойността на адресния регистър A0, който се инкраментира с две на всеки цикъл и се проверява дали стойността му е достигнала граничната (**CMPA #2*N, A0**). Междинните резултати (VEC1[i] * VEC2[i]) се записват в регистър D1, а скаларното произведение се натрупва в регистър D0. Накрая резултатът се записва в клетка с име SUM.

```
N       EQU     10
START   ORG     $1000
        CLR     D0              нулиране на сумата
        SUBA    A0, A0          нулиране на адресен регистър A0
LOOP    MOVE    VEC1(A0), D1    зарежда поредното число VEC1[i]в D1
        MULU    VEC2(A0), D1    умножава  VEC2[i] с D1
        ADD     D1, D0          D0 = D0 + VEC1[i] * VEC2[i];
        ADDA    #2, A0           A0 = A0+2  натрупва от 0 до 2*N
        CMPA    #2*N,           A0 сравнява дали A0 е равно на 2*N ?
        BNE     LOOP            при не преход към (goto) LOOP
        MOVE    D0, SUM         записва резултата от D0 в SUM
        STOP    #$2700
        ORG     $2000
VEC1    DC      1,2,34,5,6,7,88,93,2,4
VEC2    DC      2,3,4,5,6,5,1,3,7,2
SUM     DS      1
        END     START
```

**Пример 10.3.** Да се намери най-голямото число (MAX) в един едномерен масив (LIST) с N елемента.

Решение на примера на езика C++:

```
MAX = LIST[0] ;
for (J = 1; J <= N; J++)
{
        if (LIST[J] > MAX)   {   MAX = LIST[J];  }
}
```

При реализирането на програмата е използвана индиректна индексна адресация със следдекраментиране. Използван е подходът, при който първият елемент на масива (LIST[0]) се записва в променливата MAX (в случая регистър D0) и в цикъл се сравнява с всички останали елементи на масива. Ако текущият елемент (LIST[J]) е по- голям (**CMP.B D2, D0** и **BGT LAST** ) от този в клетка MAX ( рег. D0), то този елемент се записва в клетка MAX ( рег. D0). Така след обхождането на масива в клетка MAX ( рег. D0) остава максималният по стойност елемент.

Ако в дадената програма инструкцията **BGT LAST** се замени с **BLT LAST**, то тогава ще се намери най-малкият елемент на масива.

В програмата е използвана инструкицята DBRA и затова броячът на цикъла е инициализиран с 8 (отброява от 8 до 0 – девет повторения на цикъла).

```
N       EQU       10
START   ORG       $1000
        CLR.B     D0
        MOVE.W    #N-2, D1      броя на цикъла D1 = 8
        LEA       LIST, A0      адрес на масива LIST в рег  A0
        MOVE.B    (A0)+, D0     D0=LIST[0] MAX = първия елемент
NEXT    MOVE.B    (A0)+, D2     D2 = LIST[J], J=1 текушия елемент
        CMP.B     D2, D0        сравняване на D2 с D0
        BGT       LAST          ако D0 е по-голям от D2 goto LAST
        MOVE.B    D2, D0        ако не, по-голямото в D0 = D2
LAST    DBRA      D1, NEXT      Dec(брояч); IF брояч > -1 goto
                                NEXT
        MOVE.B    D0, MAX       запис на макс ел. в клетка MAX
        STOP      #$2700
        ORG       $2000
LIST    DC.B      1,65,34,51,6,7,88,93,2,4
MAX     DS.B      1
        END       START
```

**Пример 10.4.** Преместване на блок от данни с големина (SIZE) 16 байта от едно място (FROM) в паметта на друго място (TO). Програмата е на адрес $1000, а масивите FROM и TO са на адреси $2000 и $3000.

Решение на примера на езика C++:

**SIZE = 16;**

**for (J = 0; J < SIZE; J++) { TO[J] = FROM[J]; }**

При решението на задачата от примера адресите на блоковете данни (FROM и TO) се зареждат в адресните регистри A0 и A1 чрез две последователни инструкции **LEA**. В регистъра D0 (брояча) се зарежда големината на блоковете с данни – в случая числото 16, дефинирано с директивата **SIZE EQU 16**. Организира се цикъл със следусловие, в който броячът се декраментира.

С инсрукцията **BNE NEXT** се проверява условието за излизане от цикъла – дали броячът е равен на нула. Ако броячът е различен от нула, цикълът се изпълнява. Ако броячът е станал нула, данните са преместени и цъкълът се прекратява.

```
SIZE    EQU       16
        ORG       $1000
        LEA       FROM, A0      адресът на блок FROM в A0
        LEA       TO, A1        адресът на блок TO в A1
        MOVE .B   #SIZE, D0     брояч = SIZE = 16
NEXT    MOVE .B   (A0)+, (A1)+  мести данни от FROM в TO
        SUB.B#    1, D0,        IF D2 > D0 goto LAST
        BNE       NEXT          ако брояч <> 0 Goto NEXT
        STOP      #$2700
        ORG       $2000
FROM    DC.B      1,65,34,51,6,7,88,93,2,4,23,45,67,55,23,16
        ORG       $3000
TO      DS.B      16
        END       $1000
```

**Пример 10.5.** Намиране на математическото очакване и дисперсията на 10 числа, записани в масива X

Решаването на задачата от примера изисква първо да се намери математическото очакване MX и после стойността му да се използва за изчисляване на дисперсията DX.

За изчисляването на MX и DX са необходими два цикъла: първият натрупващ сумата от елементите на масива X, а вторият – натрупващ сумата от квадратите на разликите между MX и всеки един от елементите на масива X $(MX-X[i])^2$. И двата цикъла се организират чрез инструкцията **DBRA** и брояч в регистъра D1. Броячът се инициализира преди всеки цикъл с числото 9, понеже инструкцията **DBRA** го намалява до 0 и при -1 прекратява изпълненнието на цикъла. Преди стартирането на двата цикъла с инстукцията **LEA X, A0** адресът на масива X се зарежда в адресния регистър A0. Така чрез индиректна индексна адресация със следіnкраментиране се извличат елементите на масива X и се натрупва сумата им или сумата от квадратите на разликата им и MX.

Сумата на елементите на масива X се натрупва в регистъра D0 и след излизане от първия цикъл се дели на 10, за да се получи MX. Стойността на MX, намираща се в D0, се използва във втория цикъл за изчисляване на разликата MX-X[i]. Това става чрез инструкцията **SUB.W D0,D**2. С инструкцията **MULU D2,D2** се осъществява повдигане на тази разлика на квадрат $(MX-X[i])^2$. След това резултатът от регистъра D2 се натрупва в регистъра D3. След излизане от втория цикъл D3 съдържа дисперсията. Накрая числата от регистри D0 и D3 се записват в клетки MX и DX.

```
         ORG      $1000
         LEA      X, A0          зарежда адреса на X в рег. A0
         MOVE.W   #0, D0         нулиране на рег. D0 (MX = 0)
         MOVE.W   #9, D1         D1 е брояч на цикъла-брои от 9
                                  до 0
LOOP     MOVE.W   (A0)+, D2      премества числото от този адрес
                                 (A0) в D2 и увеличава адреса с 1
                                 - така A0 сочи следващия елемент
                                 от масива
         ADD.W    D2, D0         добавя поредното число към
                                  сумата D0
         DBRA     D1, LOOP       намаля брояча с 1 и ако е по-
                                  голям от -1 прави преход към
                                  LOOP
         MOVE.W   #10, D4        зарежда 10 в D4
         DIVU     D4,D0          дели резултата в D0 на 10 ->
                                  средно аритметично (D0 е равно
                                  на MX)
         LEA      X, A0          зарежда адреса на X в рег. A0
         MOVE.W   #0, D3         сума за DX = 0 използва рег D3
         MOVE.W   #9, D1         брояч на цикъла - брои от 9 до 0
LOOP1    MOVE.W   (A0)+, D2      премества числото от този адрес
                                  в D2 и увеличава адреса с 1
         SUB.W    D0,D2          намира MX-Xi (резултат в D2)
         MULU     D2,D2          (MX-Xi)^2 с резултат в D2
         ADD.W    D2,D3          добавя междинния резултат към
                                  сумата в регистър D3
         DBRA     D1, LOOP1      намаля брояча с 1 и ако по-голям
                                  от -1, прави преход към LOOP1
         MOVE.W   D0, MX         записва D0 в MX
         MOVE.W   D3, DX         записва D3 в DX
         STOP     #$2700
         ORG      $2000
X        DC.W     7, 8,12,23,32,11,56,2,7,8  стойности на
                                 масива X
MX       DS.W     1              резервирана памет за MX
DX       DS.W     1              резервирана памет за DX
         END      $1000
```

**Задачи:**

1. Напишете програма, която да изчислява следната сума:

$$Suma = \frac{P_{\min}}{N-1}\sum_{i=1}^{N} Y_i \ * Z_i^2 \ ,$$

където $P_{min}$ е най-малкият елемент на масива $P$ с 15 елемента.
Масивите $Y$ и $Z$ са с по 10 елемента ($N = 10$).

2. Напишете програма, която да изчислява следната сума:

$$S = \frac{F_{\max}}{M-1}\sum_{i=1}^{M} (A_i \ - B_i^2) \ ,$$

където $F_{max}$ е най-големият елемент на масива $F$ с 12 елемента.
Масивите $A$ и $B$ са с по 15 елемента ($M = 15$).

3. Напишете програма, която да изчислява следната сума:

$$Sum = \frac{K_{sum}}{L-1}\sum_{j=1}^{L} (\frac{alfa_j}{beta_j} - gama_j^2) \ ,$$

където $K_{sum}$ е сумата от елементите на масива $K$ с 15 елемента.
Масивите *alfa, beta* и *gama* са с по 10 елемента ($L = 10$).

4. Напишете програма, която да изчислява следната сума:

$$Suma = \frac{P_{sr}}{N-1}\sum_{i=1}^{N} (F_i \ - Q_i^2) \ ,$$

където $P_{sr}$ е средноаритметичното на масива $P$ с 15 елемента.
Масивите $F$ и $Q$ са с по 10 елемента ($N = 10$).

5. Напишете програма, която да изчислява следната сума:

$$S = \frac{F_{sr}}{M-1}\sum_{i=1}^{M} (A_i \ - B_i \ )^2 \ ,$$

където $F_{sr}$ е средноаритметичното на масива $F$ с 12 елемента.
Масивите $A$ и $B$ са с по 10 елемента ($M = 10$).

6. Напишете програма, която да изчислява следната сума:

$$Sum = \frac{K_{sr}}{L-1}\sum_{j=1}^{L} (\frac{gama_j - alfa_j}{beta_j}) \ ,$$

където $K_{sr}$ е средноаритметичното на масива $K$ с 14 елемента.

Масивите *alfa, beta* и *gama* са с по 12 елемента ($L = 12$).

7. Напишете програма, която да изчислява следната сума:

$$Suma = \frac{P_{\min}}{N-1}\sum_{i=1}^{N}(Y_i + Z_i^2),$$

където $P_{min}$ е най-малкият елемент на масива $P$ с 10 елемента.

Масивите $Y$ и $Z$ са с по 12 елемента ($N = 12$).

8. Напишете програма, която да изчислява следната сума:

$$S = \frac{F_{\max}}{M-1}\sum_{i=1}^{M}(A_i^2 - B_i),$$

където $F_{max}$ е най-големият елемент на масива $F$ с 16 елемента.

Масивите $A$ и $B$ са с по 13 елемента ($M = 13$).

9. Напишете програма, която да изчислява следната сума:

$$Sum = \frac{K_{sum}}{L-1}\sum_{j=1}^{L}\left(\frac{alfa_j}{beta_j} + gama_j\right),$$

където $K_{cum}$ е сумата от елементите на масива $K$ с 15 елемента.

Масивите *alfa, beta* и *gama* са с по 10 елемента ($L = 10$).

10. Напишете програма, която да намира разликата между максималния и минималния елемент на масива X с 20 елемента.

11. Напишете програма, която да сортира елементите на масива X с 15 елемента във възходящ ред. Елементите на масива X да съдържат числа, по-големи от 0.

12. Напишете програма, която да сортира елементите на масива X с 10 елемента в низходящ ред. Елементите на масива X да съдържат числа, по-големи от 0.

13. Напишете програма, която да намира сумата от елементите на масива X, които са по-големи от 20 и по-малки от 50

14. Напишете програма, която да намира сумата от квадратите на елементите на масива A, които са по-големи от 36.

15. Напишете програма, която в масив от символи (стринг) намира и подменя малките латински букви с големи. Десетичните еквиваленти на ASCII кодовете на буквите от A до Z и на тези от a до z са съответно от 65 до 90 и от 97 до 122.

16. Напишете програма, която в масив от символи (стринг) да намира и премахва интервалите между думите, като оставя само по един интервал. Десетичният еквивалент на интервал (space) е 32.

# 11. Разработка на подпрограми за MC68000

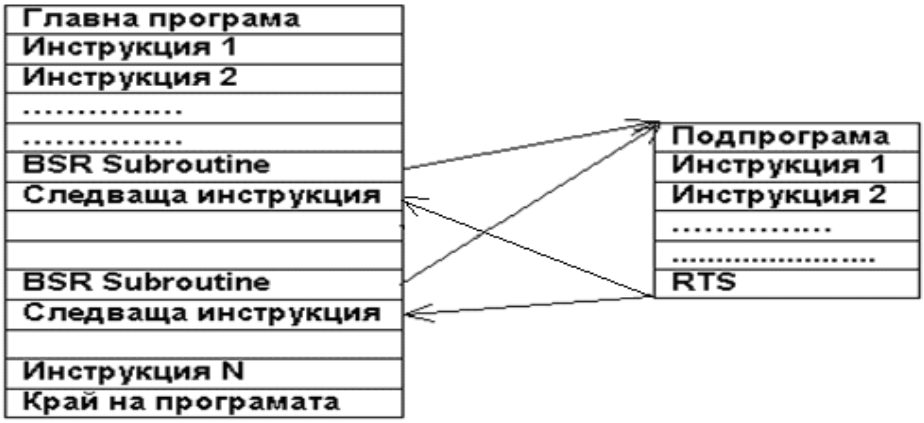
Подпрограмите са обособени фрагменти от програмния код, които могат да се използват многократно. Те са аналог на функциите в другите езици за програмиране. Подпрограмите могат да се оформят в библиотеки от файлове, които могат да се използват многократно за решаване на сложни задачи.

Подпрограмите се извикват с инструкцията **BSR**, например **BSR PPName** или **BSR $2000**, където PPName или $2000 са входните точки/ адреси на подпрограмата.

Ако подпрограмата отстои на повече от 32 килобайта в посока по-голям или по-малък адрес от текущия адрес в програмния брояч, вместо инструкцията **BSR** се използва инструкцията **JSR** по аналогичен на описания по-горе начин.

Всяка подпрограма завършва с инструкцията **RTS**, след която управлението се връща/предава на основната програма в инструкцията, намираща се непосредствено след **BSR** (фиг. 11.1).

За да може след изпълнение на подпрограмата управлението да се предаде на следващата инструкция след инструкцията **BSR**, адресът за връщане се запомня в стека, така че при изпълнение на инструкцията **RTS** този адрес се зарежда в програмния брояч.




**Фигура 11.1. Изпълнение на подпрограми**

**Пример 11.1.** Напишете подпрограма за изчисляване на сумата от квадратите на едномерен масив с брой на елементите N. Използвайте подпрограмата за изчисляване на $S = \sum_{i=1}^{N} X_i^2 + \sum_{i=1}^{M} Y_i^2$, където масивите X и Y са съответно с по N = 10, а M = 8 елемента.

```
N          EQU      10
M          EQU      8
START      ORG      $1000
           LEA      X,A0
           MOVE     #N-1,D1
           BSR      SUMAX2
           MOVE     D0,S
           LEA      Y,A0
           MOVE     #M-1,D1
           BSR      SUMAX2
           ADD      D0,S
           STOP     #$2700
           ORG      $1500
SUMAX2     MOVE     #0,D0
NEXT       MOVE     (A0)+,D2
           MULU     D2,D2
           ADD      D2,D0
           DBRA     D1,NEXT
           RTS
           ORG      $2000
X          DC       1,2,3,4,5,6,7,8,9,10
Y          DC       2,4,6,8,10,1,2,3
S          DS       1
           END     START
```

Подпрограмата SUMAX2 изчислява сумата от квадратите на елементите на даден масив. Ресултатът, който подпрограмата изчислява, се намира в регистъра D0. Входните данни за подпрограмата са адресът на масива и броят на елементите му. Те се записват в регистри A0 (адрес) и D1 (брой на елементите на масива). Понеже в подпрограмата е използвана инстукцията за цикъл **DBRA**, то броят на елементите на масива трябва да е с единица по-малък от действителния.

В главната програма подпрограмата SUMAX2 се извиква два пъти – първия път за изчисляване на сумата от квадратите на масива X, а втория път – за изчисляване на сумата от квадратите на масива Y. Преди всяко извикване на подпрограмата регистрърът A0 се инициализира с адреса на съответния масив (X а после Y), а регистъра D1 – с броя на елементите на масива (10 и 8). Междинните резултати, получени чрез регистъра D0, се натрупват в клетка с име S.

**Пример 11.2.** Използване на стека за предаване на данни към подпрограми.

В приведения по-долу пример се използва регистърът A7 (стеков регистър) за предаване на данни към подпрограмата ADDUP, която събира две двубайтови числа. Подпрограмата получава тези числа, като ги прочита последователно от стека, използвайки индиректна адресация с офсет (в случая офсети 6 и 4). Прочетените числа се записват в регистрите D2 и D3, като сумата им се получава в D3. Тази сума се записва отново в стека с изместване 4 байта спрямо текущото показание на стековия указател, който от своя страна сочи адреса за връщане от подпрограмата.

В главната програма чрез инструкцията **LEA $1000,A7** стекът се инициализира, а инструкциите **MOVE.W D0,-(A7)** и **MOVE.W D1,-(A7)** записват данните в стека, като предварително намаляват адреса, сочен от стековия указател (A7). След записа на данните в стека стековият указател сочи адрес $1000-4 (4 = две числа по два байта).

Инструкцията **BSR ADDUP** записва 4-байтов адрес за връщане след подпрогрмата. Този адрес също се записва в стека, така че стековият указател сочи адрес $1000-8.

Инструкциите **MOVE.W 4(A7),D2** и **MOVE.W 6(A7),D3** на подпрограмата не променят стойността на стековия указател, а просто прочитат числата, които са записани на по-старши адреси от този за връщане от подпрограмата.

Инструкцията **MOVE.W (A7)+,D2** непосредствено след **BSR** прочита разултата от стека и установява стековия указател в началното му положение (адрес $1000).

```
START     ORG      $400
          LEA      $1000,A7
          MOVE.W   #1,D0
          MOVE.W   #2,D1
          MOVE.W   D0,-(A7)
          MOVE.W   D1,-(A7)
          BSR      ADDUP
          MOVE.W   (A7)+,D2
          LEA      2(A7),A7
          MOVE.W   #10,D0
          MOVE.W   #22,D1
          MOVE.W   D0,-(A7)
          MOVE.W   D1,-(A7)
          BSR      ADDUP
          MOVE.W   (A7)+,D2
          STOP     #$2700
ADDUP     MOVE.W   4(A7),D2
          MOVE.W   6(A7),D3
          ADD      D2,D3
          MOVE.W   D3,4(A7)
          RTS
          STOP     #$2000
          END      START
```

**Пример 11.3.**

Примерът е свързан с оформянето на програмата от пример 9.5 в глава 9 като подпрограма с име NUM7S. Тази програма визуализира числата от 0 до 9 чрез седемсегментения им код.

Подпрограмата визуализира числото, заредено в регистър D0.

В приведения пример подпрограмата се извиква 8 пъти като изписва на дисплея на симулационната среда числата от 7 до 0. Адресите на дисплея са от $E0000E до $E00000.

В главната програма адресът $E0000E се зарежда в адресния регистър A0 и на всеки цикъл се намалява с 2, за да сочи следващата позиция (цифра) от дисплея.

```
START   ORG     $1000
        LEA     OUTPUT,A0
        MOVE    #7,D0
NEXT    BSR     NUM7S
        SUBA    #2,A0
        DBRA    D0, Next
        STOP    #$2700
NUM7S   CMP     #0,D0
        BEQ     GOTO0
        CMP     #1,D0
        BEQ     GOTO1
        CMP     #2,D0
        BEQ     GOTO2
        CMP     #3,D0
        BEQ     GOTO3
        CMP     #4,D0
        BEQ     GOTO4
        CMP     #5,D0
        BEQ     GOTO5
        CMP     #6,D0
        BEQ     GOTO6
        CMP     #7,D0
        BEQ     GOTO7
        CMP     #8,D0
        BEQ     GOTO8
        CMP     #9,D0
        BEQ     GOTO9
        BRA     EXIT
GOTO0   MOVE.B  #%00111111,(A0)
        BRA     EXIT
GOTO1   MOVE.B  #%00000110,(A0)
        BRA     EXIT
GOTO2   MOVE.B  #%01011011,(A0)
        BRA     EXIT
GOTO3   MOVE.B  #%01001111,(A0)
        BRA     EXIT
GOTO4   MOVE.B  #%01100110,(A0)
```

```
        BRA     EXIT
GOTO5   MOVE.B  #%01101101,(A0)
        BRA     EXIT
GOTO6   MOVE.B  #%01111101,(A0)
        BRA     EXIT
GOTO7   MOVE.B  #%00000111,(A0)
        BRA     EXIT
GOTO8   MOVE.B  #%01111111,(A0)
        BRA     EXIT
GOTO9   MOVE.B  #%01101111,(A0)
EXIT    RTS

        ORG     $E0000E
OUTPUT  DS.B    1
        END     START
```

**Пример 11.4.**

Дадената долу програма преобразува двубайтови числа в седемсегментен код. За целта тя разлага даденото число на единици, десетици, стотици, хиляди и десет хиляди, използвайки подпрограма за разлагане на шестнадесетични числа HEXto10. След това цифрите на десетичното число се визуализират чрез извикване на подпрограмата NUM7S за изписване на числата от 0 до 9 в седемсегментен код върху 8-цифровия дисплей (Hardware) на симулационната среда.

Подпрограмата HEXto10 зарежда числото, което трябва да се визуализира в регистър D0. Тя използва адресен регистър A0 като указател за запазване на единиците, десетиците и т.н. на числото. За целта регистърът се инициализира с адрес $4000, като този адрес се увеличава (инкраментира) при всяко ново делене на 10 на първоначалното число. Остатъкът от целочисленото делене на 10, който се е получил в регистър D1, се запазва на адреса, посочен от регистър A0.

След разлагането на числото на единици, десетици, стотици и т.н., адресният регистър A0 сочи последния остатък (цифрата) на най-голямата част на числото.

В главната програма се използват регистрите A1 и A2, като единият (A1) се инициализира с адреса на цифровия дисплей, а другият (A2) се увеличава до достигане на стойността, съдържаща се в регистър A0. Цифрите, посочени от регистър A2, се зареждат в регистъра D0 и се интерпретират от подпрограмата за визуализиране NUM7S. Последната от своя страна записва седемсегментния код на адрес, посочен от регистъра A1.

```
START   ORG     $1000
        MOVE    #51230,D0
        MOVEA   #$4000,A0
        BSR     HEXTO10
        LEA     OUTPUT,A1
        MOVE    #$4000,A2
```

```
NEXT1   MOVE    (A2)+,D0
        BSR     NUM7S
        SUBA    #2,A1
        CMPA    A2,A0
        BNE     NEXT1
        STOP    #$2700

      * SUBROUTINES/FUNCTIONS
HEXTO10 ORG     $6000
        MOVE    D0,D3
        AND     #$8000,D3
        CMP     #$8000,D3
        BEQ     NEXT
        CMP     #9,D0
        BLE     LESS10
NEXT    DIVU.W  #10,D0
        MOVE.L  D0,D1
        SWAP    D1
        MOVE    D1,(A0)+
        CLR     D3
        MOVE    D0,D3
        MOVE.L  D3,D0
        CMP     #10,D0
        BGE     NEXT
LESS10  MOVE    D0,(A0)+
        RTS

NUM7S   CMP     #0,D0
        BEQ     GOTO0
        CMP     #1,D0
        BEQ     GOTO1
        CMP     #2,D0
        BEQ     GOTO2
        CMP     #3,D0
        BEQ     GOTO3
        CMP     #4,D0
        BEQ     GOTO4
        CMP     #5,D0
        BEQ     GOTO5
        CMP     #6,D0
        BEQ     GOTO6
        CMP     #7,D0
        BEQ     GOTO7
        CMP     #8,D0
        BEQ     GOTO8
        CMP     #9,D0
        BEQ     GOTO9
        BRA     EXIT
GOTO0   MOVE.B  #%00111111,(A1)
        BRA     EXIT
GOTO1   MOVE.B  #%00000110,(A1)
        BRA     EXIT
GOTO2   MOVE.B  #%01011011,(A1)
        BRA     EXIT
GOTO3   MOVE.B  #%01001111,(A1)
        BRA     EXIT
```

```
GOTO4   MOVE.B  #%01100110,(A1)
        BRA     EXIT
GOTO5   MOVE.B  #%01101101,(A1)
        BRA     EXIT
GOTO6   MOVE.B  #%01111101,(A1)
        BRA     EXIT
GOTO7   MOVE.B  #%00000111,(A1)
        BRA     EXIT
GOTO8   MOVE.B  #%01111111,(A1)
        BRA     EXIT
GOTO9   MOVE.B  #%01101111,(A1)
EXIT    RTS
        ORG     $E0000E
OUTPUT  DS.B    1
        END     START
```

**Задачи:**

1. Какъв е резултатът от изпълнение на следната подпрограма:

```
Start   ORG     $2000
        MOVE    #4,DO
        BSR     ABC
        STOP    #$2700
ABC     MULU    DO,DO
        ASL.L   #1,D0
        RTS
        END     Start
```

2. Използвайте по-горната подпрограма за изчисляване на израза
$P = X^2 + Y^2 + Z^2$,

като извикате подпрограмата 3 пъти.

3. Модифицирайте горната подпрограма за изчисление на израза
$P = X^3 + Y^3 + Z^3$,

така че тя да изчислява кубовете на дадено число.

4. Разработете подпрограма, която да изчислява средноаритметичното на масив от N числа, като адресът на масива се запазва в адресен регистър A5, а броят на елементите му – в регистър D5. Резултатът от подпрограмата да се записва в регистър D6.

5. Използвайте подпрограмата за решаване на следния израз:

$$Suma = \frac{1}{N}\sum_{i=1}^{N}(Z_i), \qquad \text{където } N = 20.$$

6. Разработете подпрограма, която да изчислява сумата от квадратите на едномерен масив с N елемента, като адресът на масива се запазва в адресен регистър A4, а размерът му – в регистър D4. Резултатът от подпрограмата да се записвав регистър D6.

7. Използвайте подпрограмата от по-горната задача за решаване на израза:

$$Suma = \sum_{i=1}^{N}(Z_i^2) + \sum_{i=1}^{M}(Y_i^2), \quad \text{където } N = 10, \text{ а } M = 15.$$

8. Използвайте подпрограмата за визуализиране на числа, като я тествате с едноцифрени, двуцифрени, трицифрени, четирицифрени и петцифрени числа. Анализирайте състоянието на регистрите ѝ. За целта използвайте постъпково изпълнение на инструкциите ѝ.

9. Обяснете значението на следния код в подпрограмата HEXto10:

```
HEXTO10 ORG     $6000
        MOVE    D0,D3
        AND     #$8000,D3
        CMP     #$8000,D3
        BEQ     NEXT
        CMP     #9,D0
        BLE     LESS10
```

10. Какъв би бил резултатът от използването на програмата за визуализ-иране на числа, ако по-горният код е закоментиран? Обърнете внимание на тестовете с едноцифрени и петцифрени числа.

11. Разработете подпрограма за събиране на две числа с плаваща запетая. За целта използвайте алгоритъма от глава 3. Адресите на мантисите на първото и второто число да са в регистри A1 и A3, а на експонентите им – в регистри A2 и A4.

12. Разработете подпрограма за изваждане на две числа с плаваща запетая. За целта използвайте алгоритъма от глава 2. Адресите на мантисите на първото и второто число да са в регистри A1 и A3, а на експонентите им – в регистри A2 и A4.

13. Разработете подпрограма за умножение на две числа с плаваща запетая. За целта използвайте алгоритъма от глава 2.

Адресите на мантисите на първото и второто число да са в регистри A1 и A3, а на експонентите им – в регистри A2 и A4.

14. Разработете подпрограма за делене на две числа с плаваща запетая. За целта използвайте алгоритъма от глава 2. Адресите на мантисите на първото и второто число да са в регистри A1 и A3, а на експонентите – им в регистри A2 и A4.

15. Разработете подпрограма за нормализация на едно число в плаваща запетая. Адресът на мантисата на числото да е в регистър A1, а на експонентата му в – регистър A2.

16. Използвайте разработените подпрограми за работа с плаваща запетая, за да изчислите израза:

$Z = (X^2 + Y^2) / (X- Y)$, където:

а) X = 12.5, а Y = 8.25
б) X = 122.625, а Y = 18.25
б) X = 321.25, а Y = 95.4
г) X = 0.8, а Y = 0.5
д) X = 0.05, а Y = 0.01
е) X = 1000.25, а Y = 66.4

# 12. Програмиране на асемблер за процесор ARM

**Програмен модел**

Процесорната фамилия ARM (Advanced RISC Machines) се състои от RISC микропроцесори, които имат 16 регистъра (фиг. 12.1) с общо предназначение с имена от R0 до R15. Регистрите са 32-битови. Те могат да съдържат както адреси, така и данни. Последният регистър R15 се използва за програмен брояч (PC), а регистър R13 служи за организиране на програмен стек (SP). Регистър R14 (LR) се използва като регистър, съдържащ адреса за връщане след подпрограма. Регистрите могат да се използват за съхранение на 8, 16 и 32-битови числа.

бит 31 бит 0

| |
|---|
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13 (Stack pointer – SP) |
| R14 (link register – LR) |
| R15 (PC) |

**Фигура 12.1. Регистри на процесора ARM**

Процесорите от тази фамилия използват статус регистър (Current Processor Status Register /CPRS/). Битовете N, Z, C и V (фиг. 12.2) на статус регистъра са аналогични по значение с тези на статус регистъра на процесора MC68000.

| 31 | 30 | 29 | 28 | … | …. | …. | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Z | C | V | | | | I | F | T | M4 | M3 | M2 | M1 | M0 | |

**Фигура 12.2. Значение на битовете на статус регистъра (CPRS)**

Процесорът ARM може да работи в пет различни режима, определяни от битовете на статус регистъра с номера от 1 до 5 (флагове M0-M4). Режимите са: потребителски, системен, супервайзорен, на

прекъсване и на бързо прекъсване. Битовете с имена I и F служат за маскиране на стандартни и бързи прекъсвания. Бит T се използва от серията процесори Thumb, които могат да работят и в 16-битов режим.

С цел бързо превключване от режим в режим се прави копие на регистрите R13 и R15. Регистрите от R0 до R7 са общодостъпни за всички режими. Същото се отнася за регистрите от R8 до R12. Изключение има само при режим на бързи прекъсвания.

**Формат на инструкциите на ARM**

Повечето от инструкциите на процесора ARM са с три операнда и имат следният формат:

**КОП ОП1, ОП2, ОП3**

Резултатът от операцията, извършена между операнд3 и операнд 2, се записва в операнд 1.

**[ОП1] <- [ОП2] операция [ОП3]**

Съществуват и инструкции с два и с един операнд, а когато стандартни инструкции се комбинират с инструкции за изместване и ротация, се получават и инструкции с четири операнда. Инструкциите с два операнда са най-често свързани с преместване на данни между регистри и с преместване на данни между регистри и памет, както и такива по сравнение на два операнда. Инструкциите с един операнд са тези за условен преход и преход към подпрограма.

**Видове инструкции на ARM**

Инструкциите на микропроцесора използват основно междурегистрови операции, както и комбинирани изчисления, които не са характерни за други процесори. Инструкциите за комбинирани изчисления са разгледани по-долу след обясненията на основните инструкции.

**Инструкции за преместване на данни:**

| | |
|---|---|
| **MOV** | Преместване между регистри[Rd] <- Op2 |
| **MOV R1,R2** | [R1] = [R2] |
| **MOV R1,#3** | [R1] = 3 |
| **MVN** | Преместване с негативната стойност[Rd]<- (-Op2) |
| **MVN R1,R2** | [R1] = -[R2] |
| **MVN R1,#6** | [R1] = -6 |
| **LDR** | Зареждане на регистър с **дума (2 байта)** от адрес от паметта или от друг регистър [Rd] <- [Мем] или [Rd] <- [R1]. |
| **LDR R1,[R2]** | [R1] <- [R2] |
| **STR** | Запазване на регистър с **дума (2 байта)** на адрес от паметта или в друг регистър [Мем] <- [Rd] |
| **STR R1,[R2]** | [R1] -> [R2] |

**Инструкции за аритметични операции:**

| | |
|---|---|
| **ADD** | Събиране [Rd] <- Op1 + Op2 |
| **ADD R1,R2, R3** | [R1] = [R2] + [R3] |
| **ADC** | Събиране с пренос[Rd] <- Op1 + Op2 + C |
| **ADC R1,R2, R3** | [R1] = [R2] + [R3] + бит C |
| **SUB** | Изваждане [Rd] <- Op1 - Op2 |
| **SUB R1,R2, R3** | [R1] = [R2]-[R3] |
| **SUB R1,R2, #10** | [R1] = [R2]-10 |
| **SBC** | Изваждане с пренос [Rd] <- Op1 - Op2 + C - 1 |
| **SBC R1,R2, R3** | [R1] = [R2]-[R3] + бит C - 1 |
| **RSB** | Реверсивно изваждане [Rd] <- Op2 - Op1 |
| **SUB R1,R2, #10** | [R1] = 10 - [R2] |
| **RSC** | Реверсивно изваждане с пренос[Rd] <<br>- Op2 - Op1 + C -1 |
| **RSC R1,R2, #10** | [R1] = 10 - [R2] + C -1 |
| **MUL** | Умножение [Rd] <- Op1 * Op2 |
| **MUL R1,R2, R3** | [R1] = [R2] * [R3] |
| **MLA** | Умножение с акумулиране[Rd] <- Op1 *Op2+Op3 |
| **MLA R1,R2, R3,R1** | [R1] = [R2] * [R3] + [R1] |

Инструкцията **MLA** може да се използва за натрупване на сума по следния начин: SUM = SUM $A_i$ * $B_i$ , където R1 = SUM, а R2 = $A_i$, а R3 = $B_i$

**Инструкции за логически операции:**

| | |
|---|---|
| **AND** | Логическо умножение [Rd] <- Op1 ^ Op2 |
| **AND R1,R2, R3** | [R1] = [R2]/\[R3] |
| **OR** | Логическо събиране [Rd] <- Op1 v Op2 |
| **OR R1,R2, R3** | [R1] = [R2]V[R3] |
| **BIC** | Логически AND NOT [Rd] <- Op1 ^(-Op2) |
| **BIC R1,R2, R3** | R1] = [R2]/\(-[R3]) |

**Инструкции за условен и безусловен преход**

Инструкциите за разклонение в програмата, като тези за условен или безусловен преход, често се съпровождат от инструкции за сравнение като **CMP** или **CMN**, които установяват битове в статус регистъра. Аритметичните инструкции и тези за преместване на данни оказват влияние на битовете в статус регистъра, само ако същите инструкции имат разширение S, обяснено по-долу.

| | |
|---|---|
| CMP | Сравняване на два операнда ( Op1 - Op2), като установява флаговете на статус регистъра. |
| CMP R1,R2 | [R1]-[R2] |
| CMN | Сравняване на два операнда (Op1+Op2) с негативна-та стойност и установява флаговете на статус р-ра. |
| TST | Установява флагове на статус регистъра на основата на логическо AND на два операнда Op1 ^ Op2. |
| TST R1,R2 | [R1]/\[R2] |

Инструкциите за условен преход имат формат ***BXX***, където разширение-то ***XX*** е дадено в таблица 12.1:

**Таблица 12.1. Инструкции за условен и безусловен преход (разширения)**

| **Разширение *XX*** | **Преход при:** | **Преход при флаг/флагове на CPRS статус регистъра** |
|---|---|---|
| **CC или LO** | бит за пренос C = 0 | C = 0 |
| **CS или HS** | бит за пренос C = 1 | C = 1 |
| **NE** | неравенство | Z = 0 |
| **EQ** | равенство | Z = 1 |
| **PL** | положителен резултат | N = 0 |
| **MI** | отрицателен резултат | N = 1 |
| **HI** | по-голямо | (Z = 0) * (C = 1) |
| **LS** | по-малко | (Z = 1) + (C = 0) |
| **GT** | по-голямо | (Z = 0) * (V = N) |
| **LT** | по-малко | V =/= N |
| **GE** | по-голямо или равно | V = N |
| **LE** | по-малко или равно | (Z = 1) + (V * N) |
| **VC** | когато няма препълване | V = 0 |
| **VS** | препълване | V = 1 |
| **AL** | Безусловен преход (goto) | |

**Примери за използване на инструкции за условен и безусловен преход:**

| | |
|---|---|
| **CMP R5, R6** | сравняват се R5 = R6 |
| **BEQ Next** | ако R5 = R6 се прави преход към етикет Next |
| **BLT Less** | ако R5 < R6 се прави преход към етикет Less |
| **BGT Big** | ако R5 > R6 се прави преход към етикет Big |
| **B Next1** | Безусловен преход (goto) към етикет Next1 |
| **BAL Next1** | Безусловен преход (goto) към етикет Next1 |

**Инструкция за преход към подпрограма**

За преход към подпрограма се използва инструкцията BL (Branch with Link).

| | |
|---|---|
| **BL SUM** | преход към подпрограмата SUM и запазване на адреса за връщане в R14 (LR) |

Инструкцията BL може да се комбинира с разширенията ***XX*** от таблица 12.1, така че преход към подпрограма да се прави, само ако е изпълнено дадено условие, свързано с конкретното разширение ***XX***. Например:

| | |
|---|---|
| **CMP R5, R6** | сравняват се R5 = R6 |
| **BLLT SUM** | преход към подпрограма SUM се прави, ако R5 < R6 |

**Примери за използване на инструкциите**

Установяването на флаговете на статус регистъра (CPRS) не става автоматично при изпълнение на инструкциите. Дадена инструкция може да установи един или няколко флага в CPRS, само ако към края ѝ се добави разширение ***S***. Флаговете от CPRS пък от своя страна могат да се използват с комбиниране за условно изпълнение, дадено по-долу.

| | |
|---|---|
| **ADD R0, R1, R2** | Не установява флагове в CPRS |
| **ADDS R0, R1, R2** | **S** – установява флагове в CPRS |
| **SUBS R0, R1, R2** | **S** – установява флагове в CPRS |

**Комбиниране на инструкции**

Комбинираните изчисления включват комбиниране на стандартните инструкции с тези за изместване и ротация, както и с тези за условен преход. Комбинирането става, като към края на инструцията се добави разширението ***XX*** от инструкциите за условен преход или, след последния ѝ операнд се добави инструкция за изместване или ротация.

**Комбиниране на инструкции с операции по изместване или ротация**

Повечето от инструкциите на процесора могат да се комбинират с операции по изместване или ротация като **LSL, LSR, ASR, ASL, ROR**, които се прилагат по отношение на операнд 2 или 3 в зависимост колко операнда има инструкцията. Това допълнително увеличава възможностите на процесора за изпълненние на комплицирани операции по умножение и събиране или деление и събиране и т.н., както е дадено в примерите по-долу:

| | |
|---|---|
| **ADD R0, R1, R2, LSL #4** | [R0] <- [R1] + [R2] * 16 |

Съдържанието на регистър R2 се измества 4 пъти наляво, което съответства на умножение по $2^4 = 16$. След това се извършва стандарното събиране и записване на резултата в регистър R0.

| | |
|---|---|
| **MOV R0, R1, LSL #2** | умножава R1по 4 ($2^2$) и го записва в R0. |

Съдържанието на регистър R1 се измества 2 пъти наляво, което съответства на умножение по $2^2 = 4$. След това се съдържанието на регистър R1 премества в регистър R0.

| | |
|---|---|
| **MOV R0, R1, ASL #3** | деление на R1 на 8 ($2^3$) и го записва в R0. |

Съдържанието на регистър R1 се измества 3 пъти надясно, което съответства на умножение по $2^3 = 8$. След това съдържанието на регистър R1се премества в регистър R0.

**Условно изпълнение на инструкциите**

Условното изпълнение на инструкциите става, само ако е изпълненно условието, чието разширение е добавено към инструкцията. Условното изпълнение се съпровожда от инструкции за сравнение като **CMP** или **CMN**.

| | |
|---|---|
| **CMP R5, R6** | сравняват се регистрите R5 = R6 |
| **ADDEQ R0, R1, R2** | събирането се изпълнява, ако R5 = R6 |
| **CMP R5, R6** | сравняват се регистрите R5 = R6 |
| **SUBNE R3, R1, R2** | изваждането се изпълнява, ако R5 =/= R6 |
| **ADDGT R0, R1, R2** | събирането се изпълнява, ако R5 > R6 |

**Инструкции за работа с адреси**

| | |
|---|---|
| **ADR R0, Table** | зарежда адреса на Table в R0 |
| **LDR R1, [R0]** | зарежда R1 с данните (4 байта) от адреса, посочен от R0 |
| **LDRB R1, Value1** | зарежда R1 с данните (1 байт) от адреса, посочен от етикет Value |
| **LDRH R1, Value2** | зарежда R1 с данните (2 байта) на адреса, посочен от етикет Value2 |
| **STR R2, [R0]** | запазва R2 (4 байта) на адреса,посочен от R0 |
| **STRB STRH** | запазват съответно 1 или 2 байта |
| **LDR R1, [R0, #4]** | зарежда R1 с данните на адреса,посочен от R0 + 4 |
| **LDR R1, [R0, #4]!** | зарежда R1 с данните на адреса,посочен от R0 + 4, и го инкраментира с 4 |
| **LDR R1, [R0], #4** | зарежда R1 с данните на адреса, посочен от R0, и го инкраментира с 4 |
| **LDR R1, [R0] #-4** | зарежда R1 с данните на адреса, посочен от R0 и го декраментира с 4 |
| **LDM** | зареждане на блок от регистри |
| **LDM R1!,{R2-R5,r7-R10}** | зарежда в регистри R2 до R5 и r7 до R10 с данни от паметта на адрес, сочен от регистър R1, като автоматично инкраментира R1 с по 4 (байта) |
| **STM** | Запазване на блок от регистри |
| **STM R1!,{R2-R5,r7-R10}** | Записва регистрите от R2 до R5 и r7 до R10 в паметта на адрес, сочен регистър R1, като автоматично инкраментира R1 |

**Инструкция за софтуерно прекъсване**

| | |
|---|---|
| **SWI** | софтуерно прекъсване или край на програмата, аналог на STOP от MC68000. |
| **SWI 0x123456**<br>**SWI &11** | |

Адресът след инструкцията **SWI** се асоциира с адрес на подпрограма за обработка на прекъсване или за спиране на програмата и преминаване в супервайзорен режим на операционната система.

**Адресации на ARM**

Адресациите на ARM са подобни на тези в MC68000. Понеже всички регистри на ARM са с общо предназначение и могат да се използват и за данни, и за адреси, то има само един тип междурегистрова адресация като например:

**ADD R1,R2, R3** или **MOV R0, R1**.

Друга адресация е пряката адресация, при която последният операнд от дадена инструкция за работа с данни може да е директно число, предшествано от знака #, например:

**ADD 1,R2, #123** или **MOV R0, #5**.

Големината на числото може да е в интервала от 0 до 65535.

Адресациите, използвани при инструкциите за работа с адреси, , дадени по-долу, са аналогични на индиректните адресации на MC68000 със слединкраментация (!) или преддекраментация (#-4).

| **инструкция** | **операнди** | **RTL описание** |
|---|---|---|
| **LDR** | **R1, [R0]** | R1 = ← [R0] |
| **LDR** | **R1, [R0, #4]** | R1 = ← [R0 + 4] |
| **LDR** | **R1, [R0, #4]!** | R1 = ← [R0 + 4] ; R0 = R0 + 4 |
| **LDR** | **R1, [R0], #4** | R1 = ← [R0] ; R0 = R0 + 4 |
| **LDR** | **R1, [R0], #-4** | R1 = ← [R0] ; R0 = R0 - 4 |

Разликата е само в синтаксиса. Например при MC68000 се използва инструкцията **MOVE (A0), D1**, а при ARM – инструкцията **LDR R1, [R0]**, като R0 играе ролята на адресен регистър, а R1 – на този за данни. Вместо малки скоби (A0) при ARM се ползват квадратни [R0], като използването им означава ”вземи числото (дума от 2 байта) от адреса, посочен от регистъра в скобите”.

## Задачи:

1. Колко са регистрите на ARM? Има ли регистри за данни и адреси? Кой регистър се използва за програмен брояч и кой – за стеков указател?
2. Колко операнда имат инструкциите на ARM и в кой от тях се записва резултатът от дадена математическа операция (инструкция)?
3. Влияе ли изпълнението на дадена инструкция на флаговете в статус регистъра? Дайте примери.
4. Колко инструкции за събиране, изваждане и умножение има в асемблера на ARM? Дайте примери и посочете разликите.
5. Какви видове на комбинирани инструкции има в асемблера на ARM? Дайте примери и посочете разликите.

6. Какви адресации има в асемблера на ARM? Направете сравнение с тези на MC68000.

7. Каква е разликата между следните две инсрукции?

**LDR R5, [R2, #8]** и **LDR R5, [R2], #8**

8. Каква е разликата между следните две инсрукции?

**LDR R3, [R5, #4]!** и **LDR R3, [R5], #4**

9. Какъв е резултатът от изпълнението на следните инструкции:

а) **MOV R1, #0xFF**
б) **MVN R1, #0xFF**
в) **MVN R1, #25**
г) **MVN R1, #0x0F**
д) **MOVS R1, #0xFF**
е) **MLA R3,R6,r9,R2**
ж) **ADDS R1,R2,R3**
з) **ADDEQS R1,R2,R3**
и) **ADD R1,R2,R3, LSL #3**
й) **ADDCSS R1,R2,R3, LSL R4**

10. Какво е грешно в записа на следните инструкции:

а) **MOV R1, R2,#0xFF**
б) **MNV R1, #0xFF**
в) **SUBSEQ R1,R2,R3**
г) **MVN R1, #0x0F**
д) **MOVS R1, R21**
е) **MLA R3, #2,#23,R2**
ж) **ADDS R1, #0xFF, R2**
з) **MVN #25,R2**
и) **ADD R1,R2,R3, #3 LSL**
й) **ADDNES R1,R2,R3, ROL R16**

11. Коя е инструкцията за делене на ARM ?

12. Какво е значението на инструкциите:

**LDR R1,Masiv**
**ADR R1,Masiv**

# 13. Разработка на линейни и разклонени програми на асемблер за процесор ARM.

**Етапи на разработка на програми за процесор ARM в симулационната среда ARM Project Manager**

Разработката на програми на асемблер за ARM включва няколко етапа:

1) Въвеждане на текста на програмата чрез подходящ текстов (ASCII) редактор като Notepad, Notepad++ или друг подобен. Така написаната програма се запазва като файл с разширение .s. Например: test1.s.

2) Генериране на обектен код (обектен файл) чрез асемблера на ARM. Това става с командата:

ARMASM -g test1.s

3) Корекция на грешки, ако компилаторът е рапортувал такива.

4) Свързване на обектния код с библиотеки чрез програмата LINKER, която на основата на обектния файл създава нов имидж (image) файл, който може да се тества (дебъгва) или изпълнява в симулационната среда ARM Project Manager. Генерирането на имидж файла става с командата:

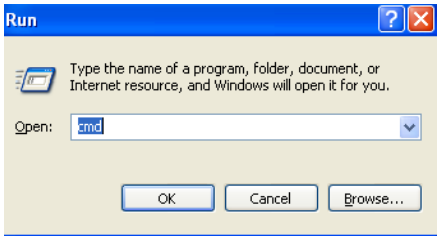
ARMLINK test1.o -o test1

5) Дебъгване (постъпково изпълнение) с цел тест и верификация на програмния код.

При етапите 2) и 4) от разработката на програмите се налага ползването на командите, дадени по-горе. Те трябва да се изпълняват от командния интерпретатор на Windows. За целта текстовият файл с програмата на асемблер (например test1.s) трябва да се копира в директория BIN на симулационната среда на ARM. Там се намират командите на ARM, които генерират обектни и имидж файлове.

Обикновено симулационната среда се намира на диск C: в директория ARM200, а BIN е нейна поддиректория.

След като текстовият файл е копиран там, се активира командният интерпретатор, като за целта се ползва Start бутона на Windows и се избира \Run\cmd, както е дадено на фиг. 13.1.

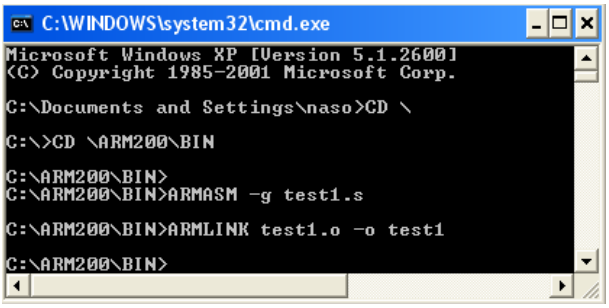



**Фигура 13.1. Активиране на командния интерпретатор**

След натискане на бутона OK се появява следният прозорец (фиг. 13.2), в който трябва да се изпълнят командите:

CD \

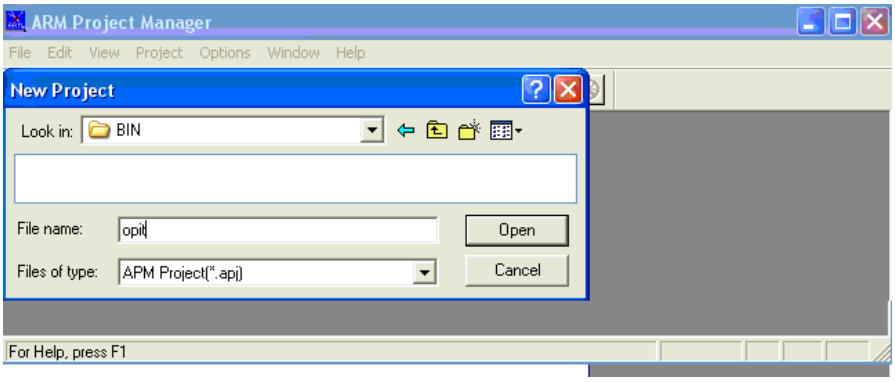
CD \ARM200\BIN




**Фигура 13.2. Асемблиране и свързване на програмния код**

Чрез тях се преминава в директорията ARM200\BIN, където трябва да се изпълнят и останалите команди, както е дадено на фиг. 13.2. Едва след това създаденият имидж файл може да се интерпретира от симулационната среда.

Симулационната среда (ARM Project Manager) е предназначена за тест (дебъгване) на имидж файловете на асемблерски програми за ARM. Тя дава възможност за създаване на нови (фиг. 13.3) и отваряне на съществуващи проекти. Това става чрез менюто Project\New или Project \Open.

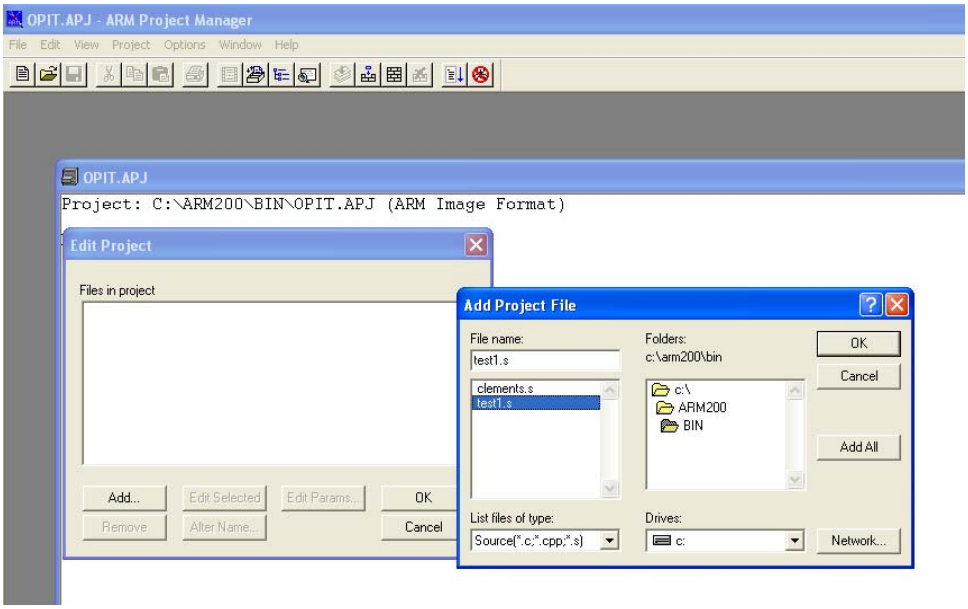



**Фигура 13.3. Създаване на нов проект чрез ARM Project Manager**

Към създадения проект могат да се добавят имидж файлове с изпълним код на програми на асемблер. Добавянето може да стане веднага след създаването на нов проект. В този случай автоматично (фиг. 13.4) се отваря прозорец Edit Project, от който с бутона Add могат да се добавят

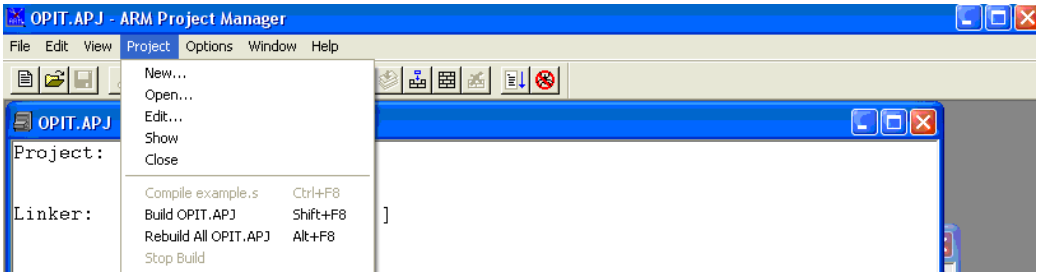
файлове с разширения .s към проекта, като заедно с тях се зареждат и тяхните имидж файлове, необходими на симулационната среда.

Текстови файлове с асемблерски код, които не са обработени по посочения по-горе начин и които нямат имидж файлове, не могат да се заредят към даден проект.




**Фигура 13.4. Добавяне на файл към нов проект**

Ако проектът е създаден без да са добавени файлове към него, това може да бъде направено по-късно от менюто Project\Edit (фиг. 13.5), като в прозореца Edit Project се натисне бутонът Add. По същия начин (чрез бутона Remove) могат и да се премахват файлове от даден проект.

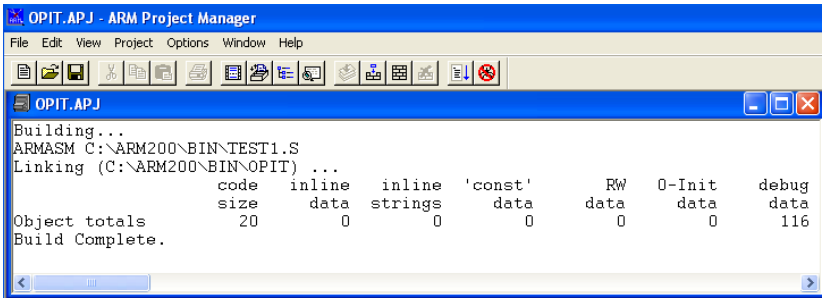



**Фигура 13.5. Добавяне на файл към съществуващ проект**

Следващата стъпка, след добавянето на файл към проекта, е така-нареченото изграждане на проекта (фиг. 13.6), което става от менюто

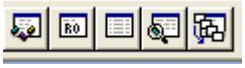
Project \Build OPIT.ARJ. Ако тази стъпка е успешна, е възможно тестването (дебъгването) на кода на програмата.

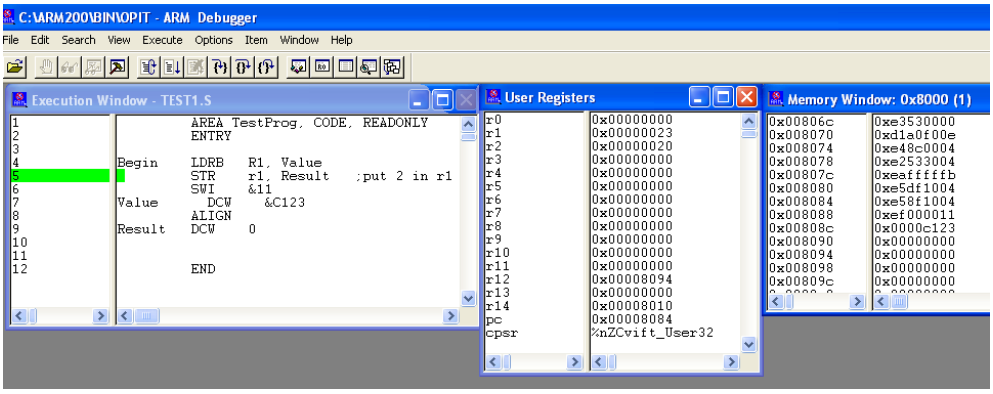
Тестването на кода става чрез менюто Project \Debug OPIT.ARJ, което отваря втори прозорец (ARM Debugger) със собствени менюта и бутони (фиг. 13.7).




**Фигура 13.6. Изграждане на проекта**

Чрез бутоните е възможно постъпково изпълнение на програмата, скок в подпрограми, презареждане и повторно тестване на програмата, спиране и т.н.

Чрез другата група бутони  се показват прозорци със съдържанието на паметта и на регистрите на процесора. По този начин може да се проверяват резултатите от изпълнението на всяка инструкция на програмата.




**Фигура 13.7. Тестване на програмата чрез ARM Debugger**

Симулационната среда, както и асемблерът и свързващият редактор, са достъпни на адрес www.arm.com. Те предоставят големи възможности за разработка на сложни приложения, включващи код, както на асемблер, така и на C++.

Описанието на всички възможности на симулационната среда е извън обсега на настоящото ръководство, но е достъпно за интересуващите се в директорията C:\ARM200\PDF.

**Директиви на асемблера на ARM**

По-долу са описани най-често използваните директиви на асемблера на този процесор.

| **Директива** | **Значение** |
|---|---|
| **AREA** | Област на действие на програма или блок данни |
| **CODE** | Код на програма или подпрограма |
| **DATA** | Начало на блок от данни |
| **ENTRY** | Начало на програмата |
| **END** | Край на програмата (последна директива във файла) |
| **EQU** | Обявяване на константа |
| **DCB** | Дефиниране на константа (**DC**) с дължина байт (B) |
| **ALIGN** | Подравняване след **DCB**, ако са декларирани нечетен брой байтове, например: **Bytes DCB 1, 2 ,3** |
| **DCW** | Дефиниране на константа (**DC**) с дължина 2 байта (W) |
| **DCD** | Дефиниране на константа (**DC**) с дължина 4 байта (D) |
| **INSERT** | Вмъкване на код от външен файл |
| **READONLY** | Определяне на особеностите (properties) на кода на програмата |

Примери за използването на директивите ще бъдат дадени в тази и в следващата глава.

**Линейни и разклонени програми**

**Пример 13.1.** Събиране на две 32-битови числа

С тази програма ще бъдат обяснени основните и задължителни директиви, без които програмата не може да бъде асемблирана, свързана и да ѝ се създаде имидж файл. Тази програма показва събирането на две числа (Value1 и Value2), които се прочитат от паметта, като резултатът се записва в паметта на адрес с име Result.

Първият ред от програмата е задължителен. Чрез директивата **AREA Program1** се дефинира област с име Program1, а с директивите **CODE** и **READONLY** се указва, че тази област е за код на програмата с особености (тип) само за четене **READONLY**.

Директивата **ENTRY** на втория ред указва началото на програмата и също е задължителна, както и директивата за край на програмата **END** на последния ред.

Основният код на програмата се разписва между реда, започващ с етикета Begin, и реда с инструкцията за софтуерно прекъсване **SWI**.

Първо двете числа от адреси Value1 и Value2 се зареждат в регистрите R1 и R2 чрез инструкциите **LDR**. След това числата, намиращи се в R1 и R2, се събират с инструкцията **ADD**, като резултатът се запазва в R3. Накрая чрез инструкцията **STR** този резултат се записва на адрес с име Result.

Непосредствено след кода на програмата следва декларирането на памет и задаване на стойност на променливите Value1, Value2 и нулиране Result.

```
            AREA Program1, CODE, READONLY
            ENTRY

Begin       LDR    r1, Value1 ; зарежда първото число в r1
            LDR    R2, Value2 ; зарежда второто число в R2
            add    R3, r1,r2  ; умножава числата резултат в r3
            str    r3, Result ; записва резултата на адрес Result
            SWI    &11        ; прекъсване / край на
                                програмата
Value1      DCD    &00000050  ; дефинира стойност на число 1
Value2      DCD    &00000020  ; дефинира стойност на число 2
            ALIGN             ; подравнява
Result      DCD    0          ; дефинира място за резултата
            END               ; край/последен ред на
                                програмата
```

В примера нарочно са използвани малки и големи букви при записването на инструкциите и регистрите, за да се подчертае, че те са еднакви за асемблера на ARM. Последният не прави разлика между малки и големи букви.

**Пример 13.2.** Изчисляване на стойността на израза: $Z = (X^2 + Y^2) * (X-Y)$

При решението на задачата съдържанието на клетка X се зарежда (**LDR**) в регистър R0, след което се умножава само по себе си (**MUL**) и се формира квадратът на X, който се запазва в R2. Аналогично, съдържанието на Y се зарежда в регистър R1 и се повдига на квадрат (**MUL**), като

квадратът се запазва в R3. Получените междинни резултати се събират, като се формира междинен резултат в R4. След операцията по изваждане (**SUB**) в R3 се формира и вторият междинен резултат. Крайният резултат R5 се получава от умножението на R3 и на R4 и се записва на адрес Z.

```
            AREA Program2, CODE, READONLY
            ENTRY
Begin       LDR   R0,X      ;зарежда числото X в R0
            LDR   R1,Y      ;зарежда числото Y в R1
            MUL   R2,R0,R0  ;квадрат на X чрез умножение
                             R2=R0*R0
            MUL   R3,R1,R1  ;квадрат на Y чрез умножение R3=R1*R1
            ADD   R4,R3,R2  ;формира X^2 + Y^2 чрез R4=R3+R2
            SUB   R3,R0,R1  ;формира X-Y чрез R3=R0-R1
            MUL   R5,R3,R4  ;формира(X^2+Y^2)*(X-Y)чрез R5=R3*R4
            STR   R5,Z      ;записва резултата на адрес Z
            SWI   &11       ;софтуерно прекъсване/край на
                             програмата
X           DCD   5         ;дефинира стойност на X
Y           DCD   2         ;дефинира стойност на Y
            ALIGN
Z           DCD   0         ;дефинира място за резултата Z
            END
```

**Пример 13.3.** Изчисляване на сумата на две 64-битови числа

В програмата се извършва сумиране на две 64-битови чиска, като едното число е записано в регистри R0 и R1, а другото – в регистри R2 и R3. В регистри R0 и R2 са младшите части на числата, а в R1 и R3 са старшите им части. Резултатът се записва в регистри R0 и R1.

```
AREA Program3, CODE, READONLY
ENTRY
MOV    R0,$6F345678   ; младша част на число 1
MOV    R1,&8966FFAA   ; старша част на число 1
MOV    R2,$B2345678   ; младша част на число 2
MOV    R3,&123456FF   ; старша част на число 2
ADDS   R0,R0,R2       ;събира на младшите части с отчитане
                      ;на пренос-бит C от статус регистъра
ADC    R1,R1,R3       ;събиране на старшите части с пренос
SWI    &11            ;прекъсване край / на програмата
END
```

**Пример 13.4** Изчисляване на стойността на P на базата на следната зависимост: ако X = Y, то P = Q + R, ако X =/= Y, то P = Q-R

Това на езика C++ би изглеждало по следния начин:

**if ( X == Y) { P = Q + R; } else { P = Q - R; }**

В приведената по-долу програма X се въвежда в R0, Y се въвежда в R1, а Q и R – в регистри R3 и R4. Резултатът се получава в регистър R2 и се записва в клетка с адрес P.

В програмата са използвани комбинираните инструкции **ADDEQ** и **SUBNE**, които се изпълняват, само ако е изпълнено съответното логическо условие (EQ или NE), зависещо от сравнението с инструкцията **CMP R0,R1**.

```
          AREA Program4, CODE, READONLY
          ENTRY
Begin     LDR    R0,X
          LDR    R1,Y
          LDR    R3,Q
          LDR    R4,R
          CMP    R0,R1
          ADDEQ  R2,R3,R4
          SUBNE  R2,R3,R4
          STR    R2,P
          SWI    &11
X         DCD    5
Y         DCD    2
Q         DCD    50
R         DCD    20
Z         DCD    0
          END
```

**Пример 13.5** Изчисляване на стойността на E на базата на следната зависимост: ако A = B и C = D, то E = E + 1.

Това на езика C++ би изглеждало по следния начин:

**if (( A == B) &&( C == D)){ E = E + 1; }**

В приведената по-долу програма A, B, C, и D се въвеждат в регистри от R0 до R3, а E – в регистър R4.

Първоначално се проверява първото условие (A == B) с инструкцията **CMP R0,R1** и ако то е изпълнено, се продължава с второто сравнение (C == D) или с условно изпълнение на **CMPEQ R2,R3** в зависимост от първото сравнение. Ако и двете условия са изпълнени, E се увеличава с единица, като това се прави с инструкцията **ADDQE R4,R4,#1**.

Обърнете внимание на директивите за декларираме на област от данни

**AREA Data1, DATA** след инструкцията **SWI**. Понеже всички данни са с големина 4 байта, директива за подравняване не е нужна и поради това е пропусната.

```
                AREA Program5, CODE, READONLY
                ENTRY
Begin           LDR     R0,A
                LDR     R1,B
                LDR     R2,C
                LDR     R3,D
                CMP     R0,R1
                CMPEQ   R2,R3
                ADDQE   R4,R4,#1
                STR     R4,E
                SWI     &11
                AREA Data1, DATA
A               DCD     5
B               DCD     2
C               DCD     50
D               DCD     20
E               DCD     0
                END
```

**Задачи:**

1. Изчислете обема и пълната повърхност на правилна призма със страни на основата A и B и височина H, където A=10, B=16 и H=17.
2. Изчислете стойността на израза $Z = (X^3 + Y^4) * (A * X - K * Y)$ при стойности на променливите X = 5, Y = 3, A = 80 и K = 12.
3. Изчислете стойността на израза S1 = P * (P-A) * (P-B) * (P-C) при P = (A + B + C) / 2. За деленето използвайте преместване на данните в регистъра една позиция надясно.
4. Изчислете стойността на израза $Z = (4 * X^2 - 16 * Y) * (8 * X - 32 * Y)$ за стойности на променливите X = 40 и Y = 7.
5. Решете задачата от пример 13.3, като я модифицирате така, че числата да се прочитат от паметта и резултатът да се запазва в паметта.
6. Напишете програма за изваждане на две 64 битови числа. За целта използвайте програмата от пример 13.3.

7. Решете задачата от пример 13.3, като я модифицирате така, че числата да се прочитат от паметта, като са разположени на адрес с етикет Number. Този адрес да се зарежда в регистър R6 и да се използва за зареждане на младшите и старшите части на числата. За целта използвайте индексна адресация от вида **LDR R1, [R6]** или **LDR R1, [R6, #4]**.

8. Решете по-горната задача 6, като за целта използвате индексна адресация от вида **LDR R1, [R6, #4]!**.

9. Решете задачата от пример 13.4, като използвате обикновени инструкции (**ADD, SUB, BEQ и BLS**), а не комбинирани такива.

10. Намерете най-голямото от три числа A, B и C. За целта използвайте обикновени инструкции за преход.

11. Намерете най-малкото от три числа A, B и C. За целта използвайте обикновени инструкции за преход.

12. Намерете най-голямото от три числа A, B и C. За целта използвайте комбинирани инструкции за преход.

13. Намерете най-малкото от три числа A, B и C. За целта използвайте комбинирани инструкции за преход.

14. Изчислете израза S = A * B * C, ако A = B и B = C. Ако не е изпълнено това условие, S = A * B * C / 8.

15. Решете задачата за управление на цифровите изходи на базата на данните, постъпващи на цифровите входове (вж. глава 9, пример 9.7).

16. Може ли текстов файл с програма на асемблер (с разширение .s) да се компилира и тества директно от симулационната среда ARM Project Manager?

17. Какви са стъпките за получаване на имидж файл от текстов файл с програма на асемблер?

18. Как се декларират константи с големина байт, 2 байта и 4 байта в асемблера? Кога се използва директивата **ALIGN**?

19. Какви са стъпките по създаване на проект в ARM Project Manager и как се добавя имидж файл към него?

# 14. Разработка на циклични и сложни програми на асемблер за процесор ARM.

Разработката на циклични програми на асемблера на ARM е аналогична на тази за асемблера на MC68000, понеже и в двата асемблера липсват специални инструкции за цикъл.

За организиране на цикли с предусловие и следусловие се използват инструкциите за сравнение и условен преход.

**Пример 14.1.** Изчисляване на сумата на числата на масива A с N елемента:

$$Suma = \sum_{i=1}^{N} A_i$$

В програмата по-долу е организиран цикъл със следусловие с брояч в регистър R2. Цикълът се инициализира с константата N, декларирана с директивата **EQU** в началото на програмата. Броячът се намалява (декраментира) с единица чрез инструкцията **SUBS R2,R2,#&01**, а чрез инструкцията **BNE Next** се следи дали е достигнал 0. Докато броячът е различен от нула, цикълът се изпълнява. Когато се достигне нула, цикълът се напуска и резултатът от регистър R0 се записва на адрес с етикет Suma.

Преди цикъла регистър R0 се инициализира с адреса на масива A по индиректен начин, като се взема адресът (**LDR R0,=Data1**), сочен от блока за данни **AREA** Data1, **DATA**. Регистърът R1 се използва за натрупване на сумата, а в R3 се зарежда текущият елемент на масива A. С инструкцията **ADD R0, R0, #+4** адресът в R0 се обновява, като сочи следващия елемент на масива A.

```
                AREA Program6, CODE, READONLY
                ENTRY
N               EQU    10
                LDR    R0,=Data1
                MOV    R1,#0
                MOV    R2,#N
Next            LDR    R3,[R0]
                ADD    R1,R1,R3
                ADD    R0,R0,#+4
                SUBS   R2,R2,#&01
                BNE    Next
                STR    R1, Suma
                SWI    &11
                AREA Data1, DATA
A               DCD    5,6,7,12,34,78,22,88,99,100
                ALIGN
                AREA Data2, DATA
Suma            DCD    0
                END
```

**Пример 14.2.** Намиране на сумата от произведението на масивите A и B по формулата: $SUM = \sum_{i=1}^{N} A_i * B_i$

При решаването на тази задача е използвана индексна адресация със слединкраментиране (**LDR R2,[R0], #4**) на адресните регистри R0 и R1, в които чрез инструкциите **ADR** са заредени адресите на масивите A и B.

Регистри R5 и R4 са използвани за брояч на цикъла и за натрупване на сума. За натрупването на сумата е използвана специално пригодената за тази цел инструкция **MLA**. Така чрез подходящо подбрана адресация и инструкции за натрупване на сума се получава компактен и сбит код на програмата.

```
                AREA Program7, CODE, READONLY
                ENTRY
                ADR    R0,A
                ADR    R1,B
                MOV    R5,#10
                MOV    R4,#0
Next            LDR    R2,[R0],#4
                LDR    R3,[R1],#4
                MLA    R4,R2,R3,R4
                SUBS   R5,#1
                BNE    Next
                STR    R4,SUM
                SWI    &11
                AREA Data1, DATA
A               DCD    5,6,7,12,34,78,22,88,99,100
                ALIGN
B               DCD    1,2,3,4,5,6,7,8,9,10
                ALIGN
SUM             DCD    0
                END
```

**Пример 14.3.** Делене на две цели числа

Понеже в асемблера на ARM няма инструкция за делене, е възможно делене само на числа, които са точни степени на 2, чрез изместване на съдържанието на даден регистър надясно. Настоящата програма реализира делене на основата на изваждане на делителя от делимото. Така в цикъл се отброява колко пъти делителят се съдържа в делимото. Ако делителят е нула, се генерира код за грешка -1 и се преминава към края на програмата.

```
        AREA Program8, CODE, READONLY
        ENTRY
        LDR    R0,Num1   ; число в Num1 е делимо
        LDR    R1,Num2   ; число в Num2 е делител
```

```
          MOV    R3,#0      ; нулиране на частното
LOOP      CMP    R1,#0      ; делителят нула ли е?
          BEQ    ERROR        ; делене на нула – преход към грешка
          CMP    R0,R1        ; делителят  по-голям ли е от делимото
          BLT    DONE         ; да – край на деленето
          ADD    R3,R3,#1     ; увеличаване на частното с 1
          SUB    R0,R0,R1     ; намаляване на делимото с делителя
          B      LOOP         ; преход в началото - етикет LOOP
ERROR     MOV    R3,#&FFFFFFFF   ; код за делене на нула (-1)
DONE      STR    R0, REMAIN      ; остатък от деленето
          STR    R3, QUOTIENT    ; частно от деленето
          SWI    &11
          AREA Data1, DATA
Num1      DCD    &0077CCA3
Num2      DCD    &0124
          ALIGN
          AREA Data2, DATA
REMAIN    DCD    0
QUOTIENT  DCD    0
          ALIGN
          END
```

**Подпрограми**

В асемблера на ARM няма специални инструкции за преход към подпрограма и за връщане от подпрограма, каквито са **BSR** и **RTS** в асемблера на MC68000. Тук за преход към подпрограма се използва инструкцията за условен преход **BL** (Branch with Link). Тя предизвиква записване на програмния брояч, който сочи адреса на следващата след **BL** инструкция в регистър **R14 (LR)**.

За завръщане от подпрограмата се ползва инструкцията **MOV PC, LR**, която зарежда адреса за връщане в програмния брояч, и процесорът продължава изпълнението на останалата част от програмата. Както бе казано в глава 12, инструкцията **BL** може да се комбинира с разширение за условен преход, като по този начин може да се прави преход към подпрограма само при изпълнение на определено условие.

**Пример 14.4.** Събиране на две числа с подпрограма

В примера по-долу е показано използването на подпрограма за изчисляване на сумата на две числа. Входни данни са числата (#10 и #22), заредени директно в регистрите R0 и R1, а резултатът от подпрограмата е в регистър R2. Подпрограмата се извиква с **BL Suma**.

```
          AREA Program9, CODE, READONLY
          ENTRY
Begin     MOV    R0,#10      ; зарежда първото число в R0
          MOV    R1,#22      ; зарежда второто число в R1
          BL     Suma        ; извикване на подпрограмата и
 ; зареждане на адреса за връщане в
 ; R14(LR)
```

```
STR     R2,Result  ; записва резултата в паметта
          SWI     $11      ; край на програмата
Suma      ADD     R2,R1,R0   ; изчисляване на сумата
          MOV     PC,LR      ; return зарежда LR в програмния
брояч

Result    DCD     0
          END
```

**Пример 14.5.** Подпрограма за намиране на сумата на N числа

В дадения по-долу пример е показано извикването на подпрограма за намиране на сумата на N числа, в която като първа инструкция е запазването на регистрите R3 и R14 в стека (чрез стековия указател в регистър R13). Регистърът R3 се запазва в стека, защото съдържанието му се променя вътре в подпрограмата. В останалата част на подпрограмата се изчислява сума на числата от масива POINTER, чийто адрес и размерност са предадени към подпрограмата чрез регистри R2 и R1. Резултантната сума се получава в регистър R0 след приключване на цикъла (**BGT LOOP**) в подпрограмата.

Последната инструкция в подпрограмата възстановява R3 и R14, като ги записва в R3 и в програмния брояч. Така става завръщане от подпрограмата в главната програма.

```
          AREA Progpam10, CODE, READONLY
          ENTRY
N         EQU    10          ; размерност на масива POINTER
          MOV    R1,N        ; размерност на масива POINTER в брояча
          LDR    R2,POINTER ; адрес на масива POINTER в R2
          BL     LISTADD     ; извикване на подпрограмата
          STR    R0, SUM     ; запис на резултата
          SWI    $11

LISTADD   STMFD R13!,{R3, R14}  ;запазва R3 и R14 в стека
          MOV     R0,#0        ; нулира сумата
LOOP      LDR     R3,[R2], #4  ; извлича поредно число от масива
                                 POINTER и увеличава указателя да
                                 сочи следващото число
          ADD     R0,R0,R3     ; добавя числото към сумата
          SUBS    R1,R1,#1     ; намалява брояча с 1
          BGE     LOOP          ;докато брояч >= 0, преход към LOOP
          LDMFD R13!, {R3, R15} ;запис на R3 и R14 в PC(R15) от
                                 стека
          AREA Data1, DATA
POINTER   DCD    1,2,3,4,5,6,7,8,9,10
          ALIGN
          AREA Data2, DATA
SUM       DCD    0
          ALIGN
          END
```

**Задачи:**

1. Напишете програма, която да намира разликата между максималния и минималния елемент на масива X с 20 елемента. Елементите на масива са с големина един байт.

2. Напишете програма, която да сортира елементите на масива X с 15 елемента във възходящ ред. Елементите на масива X да съдържат числа, по-големи от 0. Елементите на масива са с големина един байт.

3. Напишете програма, която да сортира елементите на масива X с 10 елемента в низходящ ред. Елементите на масива X да съдържат числа, по-големи от 0. Елементите на масива са с големина два байта.

4. Напишете програма, която да намира сумата от елементите на масива Bmax, които са по-големи от 50 и по-малки от 50. Масивът е с 20 елемента от по 4 байта.

5. Напишете програма, която да намира сумата от квадратите на елементите на масива M, които са по-големи от 100. Масивът е с 15 елемента от по 2 байта.

6. Напишете програма, която в масив от символи (стринг) да намира и подменя малките латински букви с големи. Десетичните еквиваленти на ASCII кодовете на буквите A – Z и на a – z са съответно от 65 до 90 и от 97 до 122.

7. Напишете програма, която в масив от символи (стринг) да намира и премахва интервалите между думите, като оставя само по един интервал. Десетичният еквивалент на интервал (space) е 32.

8. Оформете програмата от задача 5 като подпрограма.

9. Оформете програмата за събиране на 64-битови числа от глава 13 като подпрограма.

10. Напишете подпрограма за изваждане на 64-битови числа.

11. Разработете подпрограма, която да изчислява средноаритметично на масив от N числа, като адресът на масива се запазва в адресен регистър R5, а размерът му – в R05. Резултатът от подпрограмата да се записва в регистър R6. За разделяне на N използвайте подпрограмата от задача 9.

## Литература


1. A. Clements, The Principles of Computer Hardware, 4th edition, Oxford Press, 2006
2. ARM Software Development Toolkit Version 2.0, Programming Techniques, Advanced RISC Machines Ltd (ARM) 1995
3. ARM Reference Manual Advanced RISC Machines Ltd (ARM) 1995
4. MOTOROLA M68000 FAMILY, Programmer's Reference Manual, MOTOROLA INC., 1992
5. Т. Иванов, Микропроцесорна техника, Мартилен, София, 1993
6. Т. Димов. Компютърни архитектури, Мартилен, София, 2004
7. Точи Р., Ласковски Л. Микропроцесори и микрокомпютри, София, Техника, 1982г.
8. Microchip PIC12F629/675, Data Sheet 8-Pin FLASH-Based 8-Bit, CMOS Microcontrollers, Microchip Technology Inc, 2003.
9. Microchip PIC24FJ256GB110 Family, Data Sheet64/80/100-Pin, 16-Bit Flash Microcontrollers with USB On-The-Go (OTG), Microchip Technology Inc, 2008.
10. Microchip PIC32MX Programming Specification, Microchip Technology Inc, 2010.
11. Computers as Components, Second Edition: Principles of Embedded Computing System Design, Morgan Kaufmann; 2 edition, 2008
12. http://www.freescale.com/files/archives/doc/ref_manual/M68000PRM.pdf
13. http://www.coranac.com/tonc/text/asm.htm
14. http://www.scribd.com/doc/418461/Easy-Motorola-68k-Reference
15. http://www.easy68k.com/paulrsm/doc/dpbm68k2.htm