电子科技大学信息与软件工程学院

**实 验 报 告**

学 号 2022130102013

姓 名 成棋伟

（实验）课程名称《ARM处理器体系结构及

应用》课程实验

理论教师 兰 刚

实验教师 兰 刚

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：成棋伟 学号： 2022130102013 指导教师： 兰刚**

**实验地点： 信软楼西304 实验时间： 2025.4.8**

1. 实验名称：指令系统及寻址方式
2. 实验学时：4
3. 实验目的：
4. (1)熟悉Keil MDK开发环境，掌握开发平台MDK -ARM的使用；
5. (2)能正确建立ARM汇编程序工程文件，掌握汇编程序调试方法；
6. (3)熟练掌握ARM的指令系统以及指令寻址方式；
7. (4)熟练掌握ARM汇编语言的程序框架，能正确编写ARM汇编程序。
8. 实验原理：

**指令系统寻址方式**

(1)**立即数寻址**：也叫立即寻址，操作数本身就在指令中给出，取出指令也就取到了操作数。这个操作数被称为立即数，对应的寻址方式也就叫做立即数寻址。

1. (2)**寄存器寻址**：就是利用寄存器中的数值作为操作数，这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。
2. (3)**寄存器移位寻址**：当第二操作数为寄存器型时，在执行寄存器寻址操作时，也可以对第二操作数寄存器进行移位，此时第二操作数形式为。
3. (4)**寄存器间接寻址**：就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。
4. (5)**基址变址寻址**：将基址寄存器的内容与指令中给出的地址偏移量相加，得到操作数所在的存储器的有效地址。变址寻址方式常用于访问某基地址附近的地址单元。（4K范围的偏移）。有三种加偏址的方式：前变址、自动变址和后变址寻址方式。

a)前变址模式：先基址+偏址，生成操作数地址，再做指令指定的操作。也叫前索引偏移。

b)自动变址模式：先基址+偏移，生成操作数地址，做指令指定的操作。

c)后变址模式：即先用基地址传数，然后修改基地址（基址+偏移），也叫后索引偏移。

(6)**多寄存器寻址：**采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这种寻址方式是多寄存器传送指令LDM/STM的寻址方式，这种寻址方式中用一条指令最多可传送16个通用寄存器的值。连续的寄存器间用“-”连接，否则用“，”分隔。然后自动修改基址寄存器。

(7)**堆栈寻址：**是对堆栈进行操作的寻址方式堆栈寻址是隐含的，它使用一个专门的寄存器（堆栈指针SP）指向一块存储区域(堆栈)。四种类型的堆栈工作方式：

a)满递增堆栈FA(Full Ascending)：堆栈指针指向最后压入的数据，且由低地址向高地址生长。

b)满递减堆栈FD(Full Descending ） ：堆栈指针指向最后压入的数据，且由高地址向低地址生长。

c)空递增堆栈EA(Empty Ascending)：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生长。

d)空递减堆栈ED(Empty Descending)：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生长。

(8)**相对寻址：**以程序计数器PC的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。

1. 实验内容：

(1)学习Keil MDK-ARM开发平台的的使用，包括新建一个工程、在建立的工程中编写相关程序。

1. (2)掌握对ARM汇编工程编译、调试的方法。
2. (3)对40多条常用的指令，使用不同的寻址方式进行测试，并通过查看寄存器、存储器、程序状态寄存器的内容，检查是否与期望一致。
3. (4)针对这40多条指令中，可以加S标记和条件的指令，使用S标记和不同的条件进行测试，并通过查看寄存器、存储器、程序状态寄存器的内容，检查是否与期望一致。
4. 实验器材（设备、元器件）：

(1)PC机一台；

1. (2)Keil MDK-ARM uVision4开发工具。
2. 实验步骤：

(1)打开Keil MDK-ARM uVision4开发工具；

(2)新建一个工程文件；

(3)在新建的工程文件中，添加新的源程序文件

(4)编写代码

(5)选择“Build target”菜单对编写好的工程文件进行编译链接。

(6)点击““Start/Stop Debug Section””按键，对程序进行跟踪调试，在调试界面，单步执行，对CPU各寄存器的值的变化、以及相关内存的变化进行分析比较，判断程序的执行是否符合预期要求。

1. 实验结果与分析（含重要数据结果分析或核心代码流程分析）
2. **相关指令执行后，结果如下：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 指令 | 寻址方式或后缀 | 具体执行指令（不是完整程序，只需包含测试相关的代码） | 结果说明 |
| 1 | MOV | 立即数寻址 | MOV R2,#100 | R2 ← 100 |
| 立即数寻址  +EQ条件后缀 | MOV R2,#0X01  MOV R3,#0X03  CMP R2,R3  MOVEQ R2,#0X01 | 由于R2不等于R3，所以MOVEQ不执行  CMP指令影响标志位：  Nzcv |
| 寄存器移位寻址  寄存器移位寻址 | MOV R0，#0x01  MOV R1，R0，LSL#3  MOVS R1，R0，LSL#3 | 将寄存器R0左移3位传送到寄存器R1。本例执行后，R1=0x08 |
| MOV R0, #0x0f  MOV R6, R0, LSR #0x01  MOVS R6, R0, LSR #0x01  MOVS R6, R0, LSR #0x05 | MOVS会根据移动后的结果设置Z、N、C和V标志位，这在进行条件分支或比较操作时非常有用 |
| MOV R0，#0x0F  MOV R0，R0，RRX  MOVS R0，R0，RRX | MOV相比MOVS可以节省更新标志位所需的CPU周期 |
| 2 | MVN | 立即数寻址  +S条件后缀 | MOV R1,#1 | R1与R2不等，故MVN指令不执行 |
| 3 | AND | 寄存器移位寻址（逻辑左移）  +S后缀 | MOV R1,#1  MOV R2,#2  ANDS R3,R1,R2 | R3=0,标志位nZcv |
| 4 | EOR | 寄存器移位寻址（逻辑右移） | MOV R1,#0X11  EOR R2,R1,#0X01 | R2=0X10 |
| 5 | SUB | 寄存器移位寻址（循环右移）  +S后缀 | MOV R1,#0X30  MOV R2,#0X03  SUBS R3,R1,R2,LSL#2 | R3=0X24,标志位nzCv |
| 6 | RSB | 立即数寻址  +NE后缀  +S后缀 | MOV R1,#0X30  MOV R2,#0X03  CMP R1,R2  RSBNES R3,R1,R2,LSL#2 | R3=0XFFFFFFDC,标志位Nzcv |
| 7 | ADD | 寄存器移位寻址（算数左移） | MOV R1,#0X10  MOV R2,#0X20  ADDS R3,R1,R2,LSL#1 | R3=0X50,标志位nzcv |
| 立即数寻址  +S后缀 | MOV R1,#0X10  ADDS R2,R1,#0X20 | R2=0X30,标志位nzcv |
| 8 | ADC | 寄存器寻址  +S后缀 | ADDS R4, R0, R2  ADDCS R5,R1,R3 ADC R5, R1, R3 | 两个128位数相加  (无法直接实现) R0 和 R1 存储第一个128位数的两个64位部分，  假设 R2 和 R3 存储第二个128位数的两个64位部分，  结果将存储在 R4 和 R5 中 |
| 9 | SBC | 寄存器移位寻址（带进位位的循环右移） | MOV R1,#0XF0000000  MOV R2,#0XF8000000  MOV R3,#0XF4000000  MOV R4,#0XF2000000  SUBS R5,R1,R2  SBCS R6,R3,R4 | R5=0XF8000000,R6=0X01FFFFFF,标志位nzCv |
| 寄存器寻址 | MOVS R6,#0  SBC R5, R1,R3 SBC R4, R0, R2 | 两个64位数的减法  假设 R0:R1 存储第一个64位数  假设 R2:R3 存储第二个64位数  结果将存储在 R4:R5 中 |
| 10 | RSC | 寄存器移位寻址（算数右移）  +S后缀 | MOV R1,#0XF0000000  MOV R2,#0XF8000000  MOV R3,#0XF4000000  MOV R4,#0XF2000000  RSBS R5,R2,R1  RSCS R6,R4,R3 | R5=0XF8000000,R6=0X01FFFFFF,标志位nzCv |
| 11 | TST | 立即数寻址 | MOV R1,#0X11  TST R1,#0X00 | 标志位nZcv |
| 12 | TEQ | 立即数寻址 | MOV R1,#0X01  MOV R2,#0X01  CMP R1,R2  TEQ R1,#0X00 | 标志位nzCv |
| 13 | CMP | 寄存器寻址 | MOV R1,#0X01  MOV R2,#0X02  CMP R1,R2 | R1<R2，标志位Nzcv |
| 14 | CMN | 立即数寻址 | MOV R1,#0X01  CMN R1,#0X02 | 标志位nzcv |
| 15 | BIC | 立即数寻址 | MOV R1,#0X11  BIC R2,R1,#0X10 | R2=0X01 |
| 16 | MUL | 寄存器寻址 | MOV R1,#0X02  MOV R2,#0X03  MUL R3,R1,R2 | R3=0X06 |
| 17 | MLA | 寄存器寻址  +S后缀 | MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MLAS R4,R1,R2,R3 | R4=0X0A,标志位nzcv |
| 18 | UMULL | 寄存器寻址+S后缀 | MOV R1,#0XAB000000  MOV R2,#0XFE000000  UMULLS R3,R4,R1,R2 | R3=0,R4=0XA9AA0000,标志位Nzcv |
| 19 | UMLAL | 寄存器寻址  +S后缀 | MOV R1,#0XAB000000  MOV R2,#0XFE000000  MOV R3,#0X01  MOV R4,#0X02  UMLALS R3,R4,R1,R2 | R3=0X01,R4=0XA9AA0002,标志位Nzcv |
| 20 | SMULL | 寄存器寻址 | MOV R1,#0XAB000000  MOV R2,#0X0E000000  SMULL R3,R4,R1,R2 | R3=0,R4=0XFB5A0000 |
| 21 | SMLAL | 寄存器寻址  +S后缀 | MOV R1,#0XAB000000  MOV R2,#0X0E000000  MOV R3,#0X01  MOV R4,#0X02  SMLALS R3,R4,R1,R2 | R3=0X01,R4=0XFB5A0002,标志位Nzcv |
| 22 | LDR | 基址变址寻址-前变址 | MOV R0,#0X40000000  MOV R2,#-0X02  STR R0,[R0,#0X08]  LDR R1,[R0,-R2,LSL #02] | R1=0X40000000 |
| 基址变址寻址-后变址 | MOV R0,#0X40000000  STR R0,[R0]  LDR R1,[R0],#0X03 | R1=0X40000000,R0=0X40000003 |
| 基址变址寻址-自动变址 | MOV R0,#0X40000000  STR R0,[R0,#0X04]  LDR R1,[R0,#0X04]! | R1=0X40000000,R0=0X40000004 |
| 相对寻址 | LDR R0,LABLE  LABLE CMP R1, R2 | 汇编后变成  LDR R0,[PC,#-0X0004] |
| 基址变址寻址 | MOV R0,#0X40000000  STR R0,[R0,#0X04]  LDR R1,[R0,#0X04] | R1=0X40000000,R0=0X40000000 |
| 23 | STR | 寄存器间接寻址 | MOV R0,#0X01  MOV R1,#0X40000000  STR R0,[R1] | 地址为0x40000000的字单元内容为01000000 |
| 基址变址寻址-偏移量寄存器移位 | MOV R0,#0X40000000  MOV R2,#0X02  STR R0,[R0,R2,LSL#0X02] | 地址为0x40000008的字单元内容为00000040 |
| 寄存器间接寻址 | MOV R0,#0X01  MOV R1,#0X40000000  STR R0,[R1] | 地址为0x40000000的字单元内容为01000000 |
| 24 | LDRB | 寄存器间接寻址 | MOV R0,#0XFF000000  MOV R1,#0X40000000  STR R0,[R1],#0X03  LDRB R2,[R1] | R2=0X000000FF |
| 25 | LDRSB | 寄存器间接寻址 | MOV R0,#0XFF000000  MOV R1,#0X40000000  STR R0,[R1],#0X03  LDRSB R2,[R1] | R2=0XFFFFFFFF |
| 26 | STRB | 寄存器间接寻址 | MOV R1,#0XFF  MOV R2,#0X40000000  STRB R1,[R2] | 地址为0X40000000的字节单元内容为FF |
| 27 | LDRH | 寄存器间接寻址 | MOV R0,#0XFF000000  MOV R1,#0X40000000  STR R0,[R1],#0X03  LDRH R2,[R1] | R2=0X0000FF00 |
| 28 | LDRSH | 寄存器间接寻址 | MOV R0,#0XFF000000  MOV R1,#0X40000000  STR R0,[R1],#0X03  LDRSH R2,[R1] | R2=0XFFFFFF00 |
| 28 | STRH | 基址变址寻址 | MOV R0,#0X0F10  MOV R1,#0X40000000  STRH R0,[R1,#0X02] | 地址为0X40000002的半字单元内容为100F |
| 29 | STMIA | 多寄存器寻址 | LDR R0, =0X40000000  MOV R1, #1  MOV R2, #2  MOV R3, #3  STMIA R0!, {R1-R3} | 从地址为0X40000000开始的连续四个字单元内容分别为01000000,02000000,03000000,  04000000 |
| 30 | LDMIB | 多寄存器寻址 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV R4,#0X40000000  STMIA R4!,{R0-R3}  MOV R4,#0X40000000  LDMIB R4!,{R5-R8} | R5=0X02,R6=0X03,R7=0X04,R8=0 |
| 31 | LDMFA | 堆栈寻址 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV SP,#0X40000000  STMEA SP!,{R0-R3}  LDMFA SP!,{R5-R8} | R5=0X02,R6=0X03,R7=0X04,R8=0 |
| 32 | STMFA | 堆栈寻址 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV SP,#0X40000000  STMFA SP!,{R0-R3} | 从地址为0X40000004开始的连续四个字单元内容分别为01000000,02000000,03000000,  04000000 |
| 33 | SWP | 不带B后缀 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X40000000  STR R0,[R2]  SWP R3,R1,[R2] | R3=0X01,地址为0X40000000的字单元内容为02000000 |
| 带B后缀 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X40000000  STR R0,[R2]  SWPB R3,R1,[R2] | R3=0X01,地址为0X40000000的字单元内容为02000000 |
| 34 | B | 相对寻址 | B LABEL  MOV R1,#0X02  LABEL  MOV R0,#0X01 | MOV R1,#0X02指令未执行 |
| 35 | BL | 相对寻址 | BL LABEL  MOV R1,#0X02  LABEL  MOV R0,#0X01 | MOV R1,#0X02指令未执行,LR内容为该指令地址0X00000004 |
| 36 | BX | 从ARM状态跳转到Thumb状态 | CODE32  ARM1  LDR R0, =Thumb1+1  MOV LR,PC  BX R0  MOV R1,#0X01    CODE16  Thumb1  MOV R2,#0X02  BX LR | 第一次跳转时CPSR中T位置1，第二次跳转时T位置0 |
| 从Thumb状态跳转到ARM状态 | CODE32  ARM1  LDR R0, =Thumb1+1  MOV LR,PC  BX R0  MOV R1,#0X01    CODE16  Thumb1  MOV R2,#0X02  BX LR | 第一次跳转时CPSR中T位置1，第二次跳转时T位置0 |
| 37 | MSR | 寄存器寻址 | MOV R0,#0XFFFFFFFF  MSR CPSR\_CXSF,R0 | CPSR=0XFFFFFFFF |
| 立即数寻址 | MSR CPSR\_CXSF,#0XFF000000 | CPSR=0XFF000000 |
| 38 | MRS | 寄存器寻址 | MRS R0,CPSR | R0=0X000000D3 |
|  |  |  |  |  |

**伪指令实验结果分析**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **伪指令** | **执行的程序段** | **结果说明** |
|  | **LTORG** | AREA test,CODE,READONLY  ENTRY  Beg LDR R1,=0xEEEEEEEE  LDR R2,=0xAAAAAAAA  ADDS R3, R1, R2  B Next  LTORG    Next LDR R4,=0xDDDDDDDD  LDR R5,=0xBBBBBBBB  ADDS R6, R4, R5  B Beg    StrMem DCD 0x77777777,0xCCCCCCCC  END | LTORG之后，汇编器会在内存中为这些立即数分配空间，并且更新之前的LDR指令，使它们指向正确的常量池位置。这样，当程序运行时，LDR指令就可以从常量池中正确地加载立即数了。 |
|  | WHILE条件编译 | GBLA NU1  NU1 SETA 1  AREA test,CODE,READONLY  ENTRY  WHILE NU1 < 4  LDR R0,=Src  LDR R1,=0x40000100  MOV R2,#NU1  LDR SP,=0x40000840  NU1 SETA NU1+1  WEND    AREA DataRAM,DATA,READWRITE  Src SPACE 100  END | 第一次迭代中，R2 将被设置为1。在第二次迭代中，R2 将被设置为2，以此类推，直到 NU1 达到4，循环结束 |
|  | MACRO | MACRO  $MML MYM $RN,$NUM  MOV R0,$NUM  $MML.LOOP  ADD R7, R7, $RN  SUBS R0,R0,#1  BNE $MML.LOOP  EOR R7, R7,0xFF  MEND  AREA test,CODE,READONLY  M1 MYM R1, #3  M2 MYM R2, #4  END | 在反编译后的程序中，R7 的值会根据宏的调用而变化。在第一个宏调用 M1 后，R7 将包含 R1 的值加3次的结果，然后与 0xFF 异或。在第二个宏调用 M2 后，R7 将包含 R2 的值加4次的结果，然后与 0xFF 异或。 |
|  |  |  |  |

**2、部分指令测试界面截图**

下图图1为MOVS R2，#0x0，由于MOV指令带了S后缀，所以其执行结果将影响CPSR寄存器的相关标志位，从图中可以看出，CPSR寄存器的Z标志位被置1了。测试结果符合预期要求

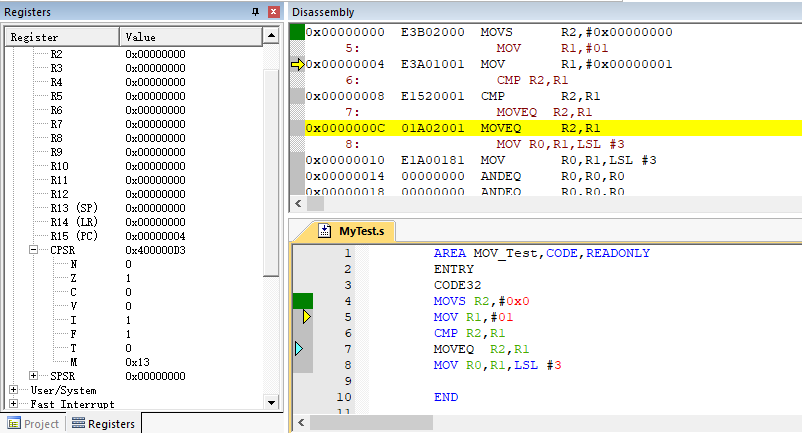


图1 MOVS R2，#0x0 指令执行测试截图

图2显示，由于Z标志位为0，表示CMP指令执行后，两个寄存器的值不相等，因此MOVEQ R2,R1指令由于不满足条件而未执行（左边橙色的方块标识），因此R2寄存器的值维持原值不变，不等于R1中存放的值。

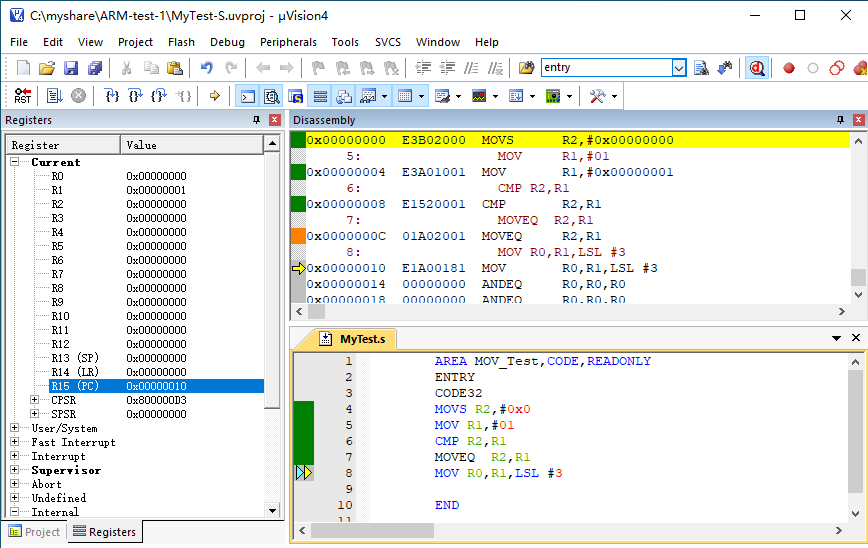


图2 MOVEQ R2，R1指令执行测试截图

图3显示，执行RSBNES指令，将R2的值左移2位后用R1减去，结果放进R3，标志位N为1，表示结果为负数。

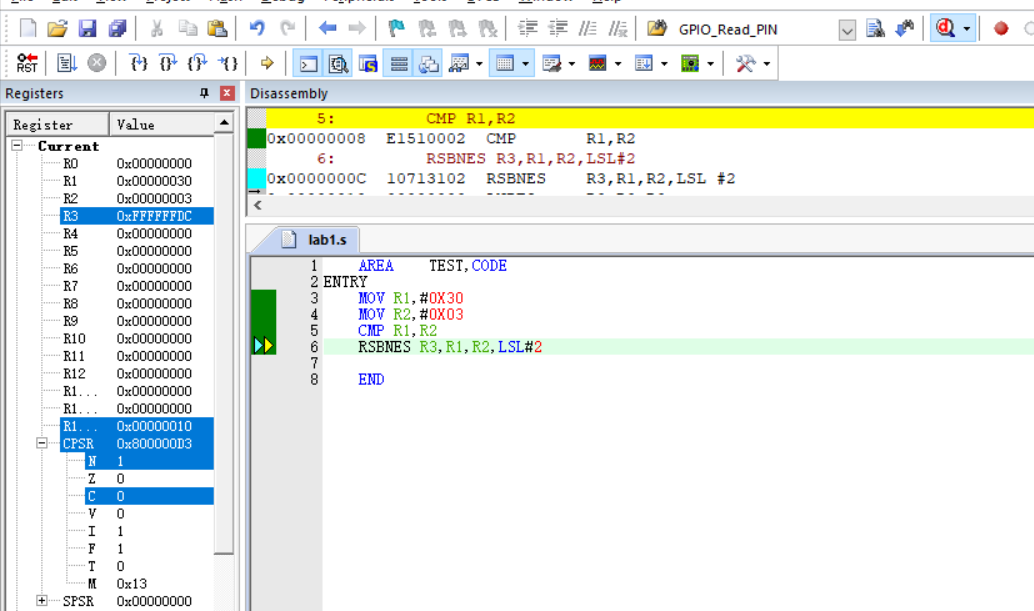


图3 RSBNES R3,R1,R2指令执行测试截图

图4显示，执行ADCS指令后R6最低位为1，说明将符号位C的值也一并加入了计算，且结果影响了CPSR的标志位。

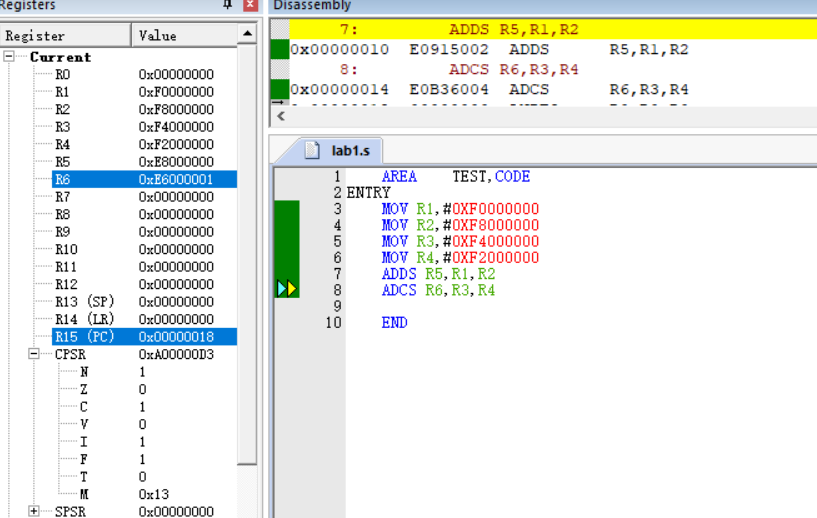


图4 ADCS R6,R3,R4指令执行测试截图

图5显示，LDR指令把地址为0X40000000的内容取至R1，R0后变址为0X40000003，测试结果符合预期。

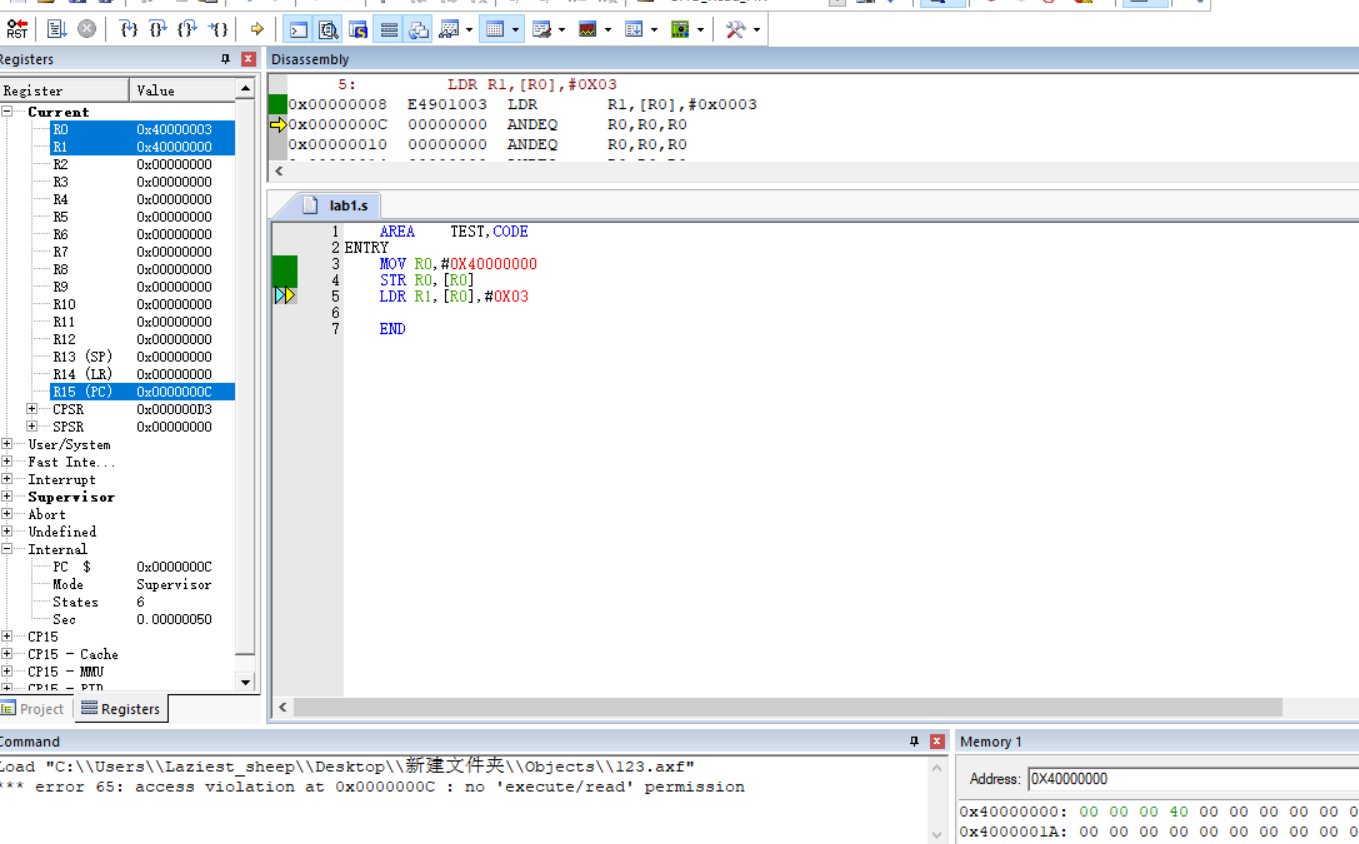


图5 LDR R1,[R0],#0X03指令执行测试截图

图6显示，STMIA指令执行后，将R0-R3的值存入以0X40000000开始的四个字单元，且为小端存储方式，指令测试符合预期

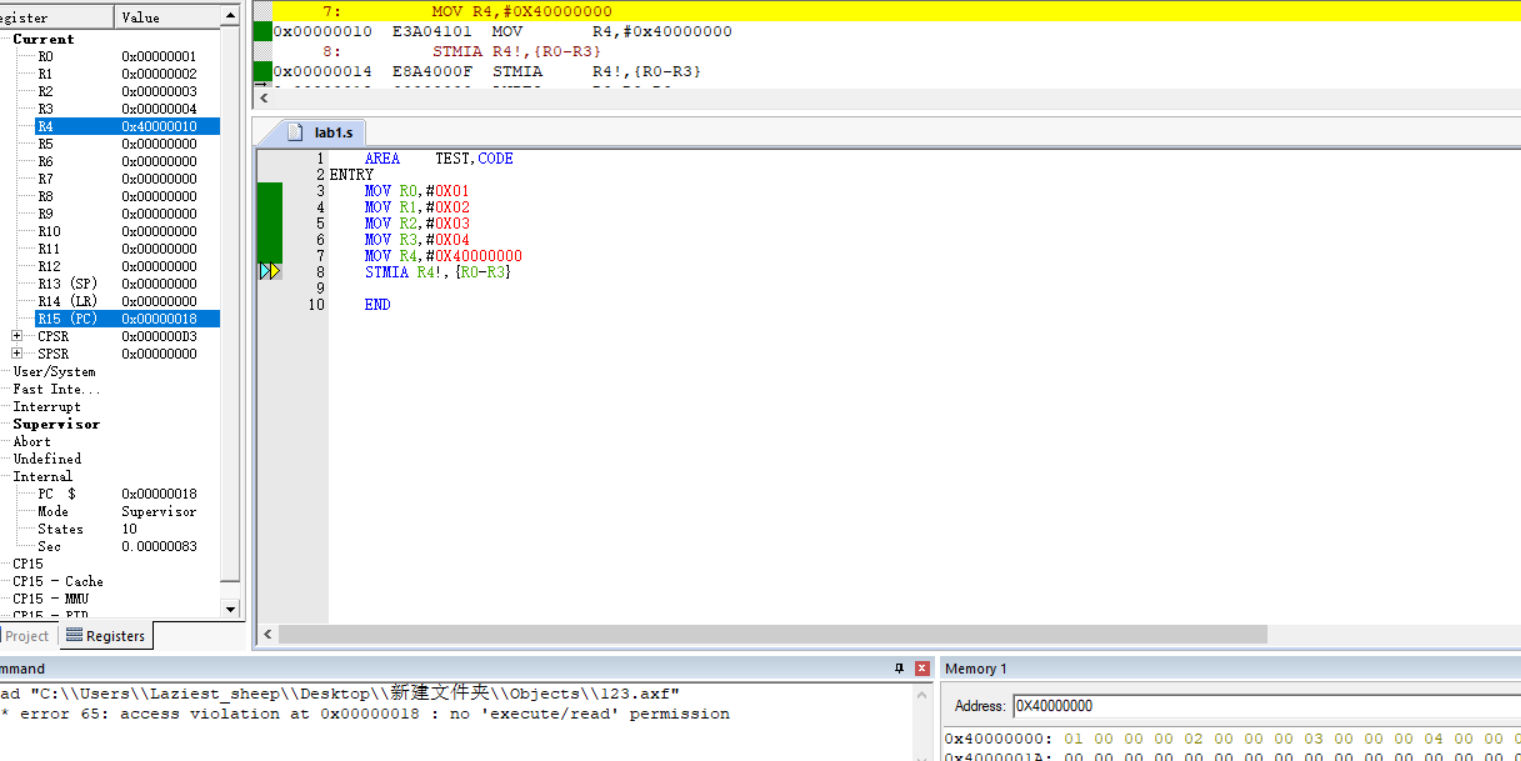


图6 STMIA R4!,{R0-R3}指令执行测试截图

图7显示，R0存入Thumb1地址且最低位置1表示转为Thumb指令，跳转前将BX下一条指令地址存入LR，并成功跳转至Thumb1处。

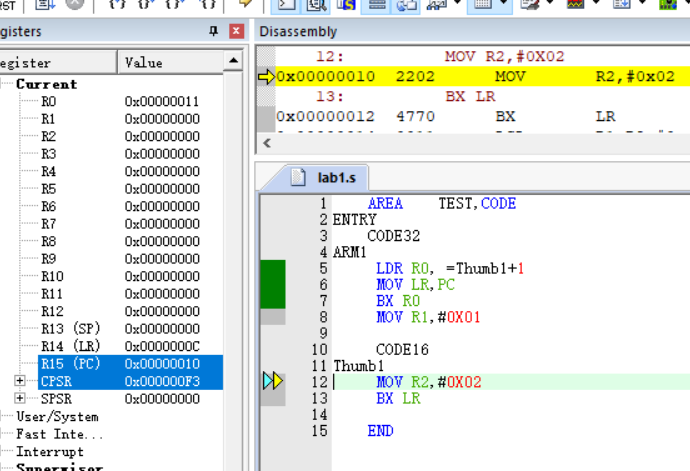


图7 BX 指令执行测试截图

**3、实验结论**

通过测试可知，除BLX指令外的其余指令执行都符合预期的结果。

1. 总结及心得体会：

通过keil自带的软件仿真功能实现了在没有实体S3C2440板子的情况下对汇编代码进行调试，基本掌握了各汇编语句的使用。同时在测试BX指令时发现，即使LDR R0,=Thumb1后没有写+1，实际R0末位也会变为1，在Thumb1标号前将CODE16注释后则不会出现这种情况，初步判断是keil发现即将要跳转的是Thumb指令集所以自动完成了相应工作，不得不说一个优秀的编译器能极大优化代码编写过程与体验。

1. 对本实验过程及方法、手段的改进建议：

汇编代码过多，全部列表实验不太现实，但是考虑到部分语句可以混合实验，典型的比如LDR与STR，完全可以一份代码测试两条指令，不必单独列项测试，建议进一步削减表格行数，去除不必要的测试。

报告评分：

指导教师签字：

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：成棋伟 学号：2022130102013**

**指导教师：兰刚**

**实验地点：信软楼西304 实验时间： 2025.4.15**

1. 实验名称：数字滤波
2. 实验学时：4
3. 实验目的：
4. (7)通过工程中常用的数字滤波编程实现，验培养学生分析问题、解决问题能力。
5. (8)通过该编程，进一步巩固和强化学生ARM汇编编程的能。
6. 实验原理：

**（1）中值滤波**

数字图像在其形成、传输记录的过程中往往会受到很多噪声的的污染，比如：椒盐噪声、高斯噪声等，为了抑制和消除这些随即产生的噪声而改善图像的质量，就需要去、对图像进行去滤波噪处理。

中值滤波是图像平滑的一种方法 它是一种非线性平滑滤波技术，在一定条件下可以克服线性滤波带来的图像细节的模糊问题，特别是针对被椒盐噪声污染的图像。中值滤波对脉冲噪声有良好的滤除作用，特别是在滤除噪声的同时，能够保护信号的边缘，使之不被模糊。这些优良特性是线性滤波方法所不具有的。此外，中值滤波的算法比较简单，也易于用硬件实现。所以，中值滤波方法一经提出后，便在数字信号处理领得到重要的应用。

中值滤波方法：对一个数字信号序列xj(-∞<j<∞)进行滤波处理时，首先要定义一个长度为奇数的L长窗口，L=2N+1，N为正整数。设在某一个时刻，窗口内的信号样本为x(i-N)，…，x(i)，…，x(i+N)，其中x(i)为位于窗口中心的信号样本值。对这L个信号样本值按从小到大的顺序排列后，其中值，在i处的样值，便定义为中值滤波的输出值。

**（2）均值滤波**

均值滤波也用于图像处理，均值滤波也称为线性滤波，其采用的主要方法为邻域平均法。线性滤波的基本原理是用均值代替原图像中的各个像素值，即对待处理的当前像素点（x，y），选择一个模板，该模板由其近邻的若干像素组成，求模板中所有像素的均值，再把该均值赋予当前像素点（x，y），作为处理后图像在该点上的灰度g（x，y），即g（x，y）=∑f（x，y）/m m为该模板中包含当前像素在内的像素总个数。

本事实验要求，将 N 个无符号数进行排序，去掉最大值和最小值，剩余的数求平均值，平均值即为本次滤波所得的结果。

**（3）中值滤波程序流程图**

中值滤波的流程如图1所示：

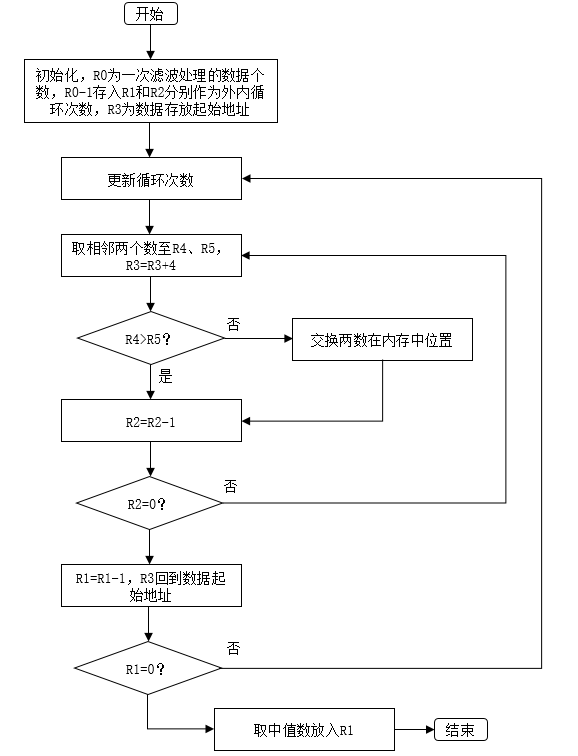


图1 中值滤波流程图

**（4）均值滤波程序流程图**

均值滤波流程图如图2所示：

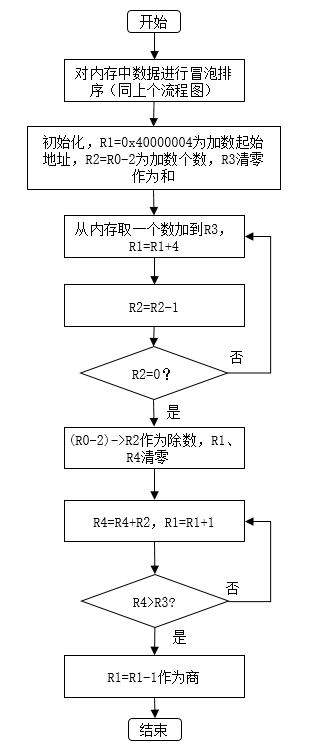


图2 均值滤波流程图

1. 实验内容：

(1)中值滤波及编程实验，要求如下：

* 关于 N（N为奇数）个数的值在程序中能任意、方便设置，并且放在 R0 中；
* 原始数据放在内存 0X40000000 开始的地址空间；
* 中值滤波的结果放在寄存器 R1 中；

1. (2)均值滤波编程实验，要求如下：

* 关于 N（N为偶数）个数的值在程序中能任意、方便设置，如 4、6、10、18 等，该值放在 R0 中；
* 原始数据放在内存中 0X40000000 开始的地址空间；
* 均值滤波的结果放在寄存器 R1 中；

1. 实验器材（设备、元器件）：

(1)PC机一台；

1. Keil MDK-ARM uVision4开发工具。
2. 实验步骤：

(1)打开Keil MDK-ARM uVision4开发工具；

(2)新建一个工程文件；

(3)在新建的工程文件中，添加新的源程序文件

(4)编写代码

(5)选择“Build target”菜单对编写好的工程文件进行编译链接。

(6)点击““Start/Stop Debug Section””按键，对程序进行跟踪调试，在调试界面，单步执行，对CPU各寄存器的值的变化、以及相关内存的变化进行分析比较，判断程序的执行是否符合预期要求。

1. 实验结果与分析（含重要数据结果分析或核心代码流程分析）
2. **中值滤波**

**(1)实验代码**

代码1中值滤波程序代码

|  |
| --- |
| AREA TEST, CODE, READONLY  ENTRY  num EQU 9 ;奇数  LDR R0, =num  SUB R1,R0,#1 ;外循环次数  MOV R2,R1 ;内循环次数  MOV R3,#0X40000000 ;循环起始地址  LOOP3  MOV R2,R1 ;更新循环次数  LOOP1  LDR R4,[R3] ;取数并比较  ADD R3,R3,#4  LDR R5,[R3]  CMP R4,R5  BHS LOOP2 ;若前小于后则交换内存中位置  STR R5,[R3,#-4]  STR R4,[R3]  LOOP2  SUBS R2,R2,#1 ;内循环次数减一  BNE LOOP1 ;一次外循环结束则不跳转  MOV R3,#0X40000000 ;从头开始下一次循环  SUBS R1,R1,#1 ;外循环次数减一  BNE LOOP3 ;外循环全部结束则不跳转    MOV R2,R0 ;取中值处理  SUB R2,R2,#1  MOV R2,R2,LSR #1  MOV R4,#4  MUL R5,R4,R2  ADD R3,R3,R5  LDR R1,[R3]  END |

**(2)运行过程及结果界面截图**

下图3是程序运行刚开始的界面。R0设置为9，总共有9个单元。手动修改从0x40000000开始的存储单元，使其初始化为：0x11、0x22、0x33、0x44、0x55、0x66、0x77、0x88、0x99（方便口头计算存储单元的结果）。

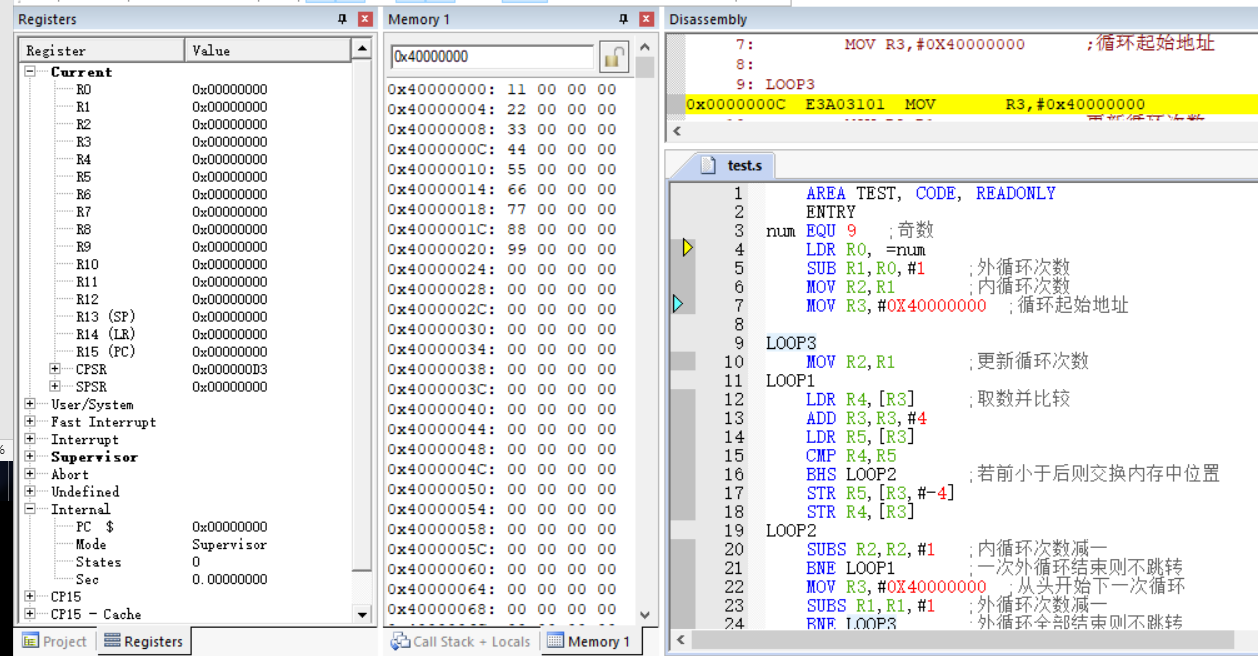


图3 中值滤波程序刚开始运行时的界面截图

下图4是程序运行结束时的界面，可以看出从0x40000000开始的9个存储单元被排序了，运行结果存放于R1中，结果为0x55。

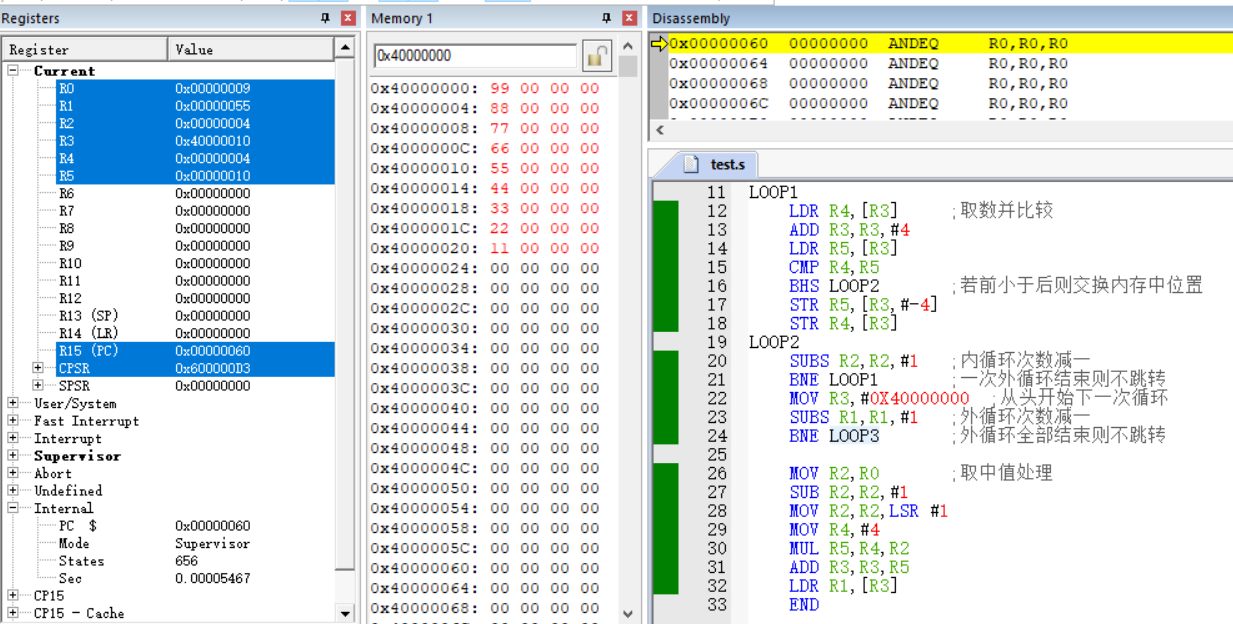


图4 中值滤波程序运行结束时的界面截图

**(3)实验结论**

0x11、0x22、0x33、0x44、0x55、0x66、0x77、0x88、0x99这九个数，其中中值为0x55，程序运行结果符合预期要求。

**2、均值滤波**

**(1)实验代码**

代码2 均值滤波程序代码

|  |
| --- |
| PRESERVE8  AREA TEST,CODE,READONLY  ENTRY  num EQU 8;偶数  LDR R0,=num    SUB R1,R0,#1 ;外循环次数  MOV R2,R1 ;内循环次数  MOV R3,#0X40000000 ;循环起始地址  LOOP3  MOV R2,R1 ;更新循环次数  LOOP1  LDR R4,[R3] ;取数并比较  ADD R3,R3,#4  LDR R5,[R3]  CMP R4,R5  BHS LOOP2 ;若前小于后则交换内存中位置  STR R5,[R3,#-4]  STR R4,[R3]  LOOP2  SUBS R2,R2,#1 ;内循环次数减一  BNE LOOP1 ;一次外循环结束则不跳转  MOV R3,#0X40000000 ;从头开始下一次循环  SUBS R1,R1,#1 ;外循环次数减一  BNE LOOP3 ;外循环全部结束则不跳转  MOV R1,#0X40000004  SUB R2,R0,#2 ;循环加次数  AND R3,#0 ;R3存放和  LOOP4  LDR R4,[R1],#4 ;取一个数  ADD R3,R3,R4 ;加到R3  SUBS R2,R2,#1  BNE LOOP4    SUB R2,R0,#2 ;R2作为除数  AND R4,#0  AND R1,#0  LOOP5  ADD R4,R4,R2 ;R4不断加除数直到超过被除数  ADD R1,#1  CMP R4,R3  BLS LOOP5  SUB R1,R1,#1 ;R1为向下取整值，即为商  END |

**(2)运行过程及结果界面截图**

下图5是程序运行刚开始的界面。R0设置为8，总共有8个单元。手动修改从0x40000000开始的存储单元，使其初始化为：0x01，0x02，0x03，0x03，0x04，0x05，0x00，0x06。

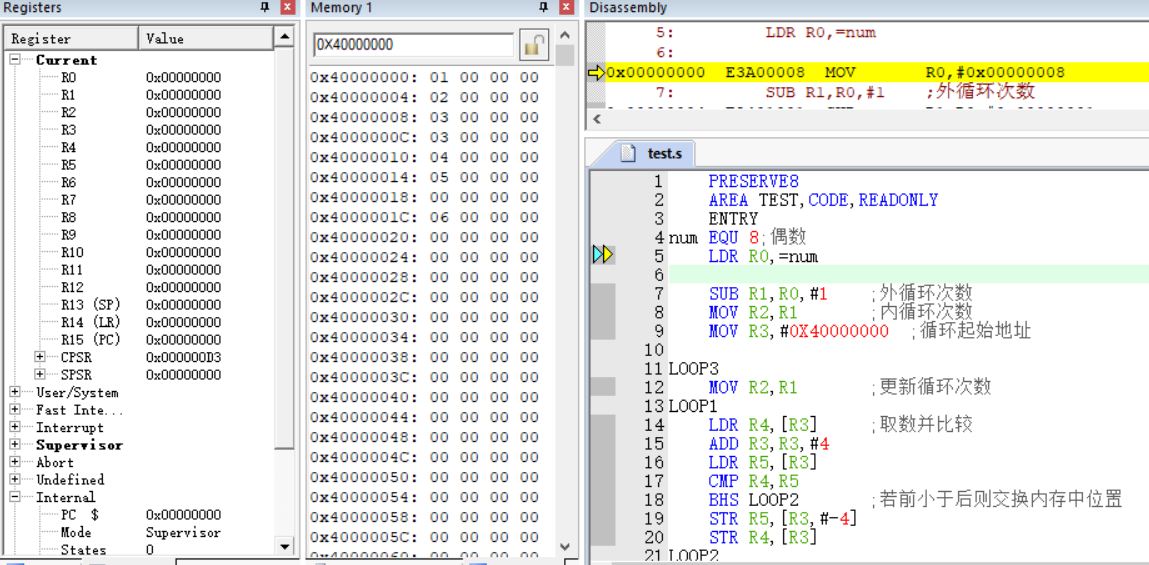


图5 均值滤波程序起始时界面截图

下图6是程序运行结束时的界面，可以看出从0x40000000开始的8个存储单元被排序了，运行结果存放于R1中，结果为3。

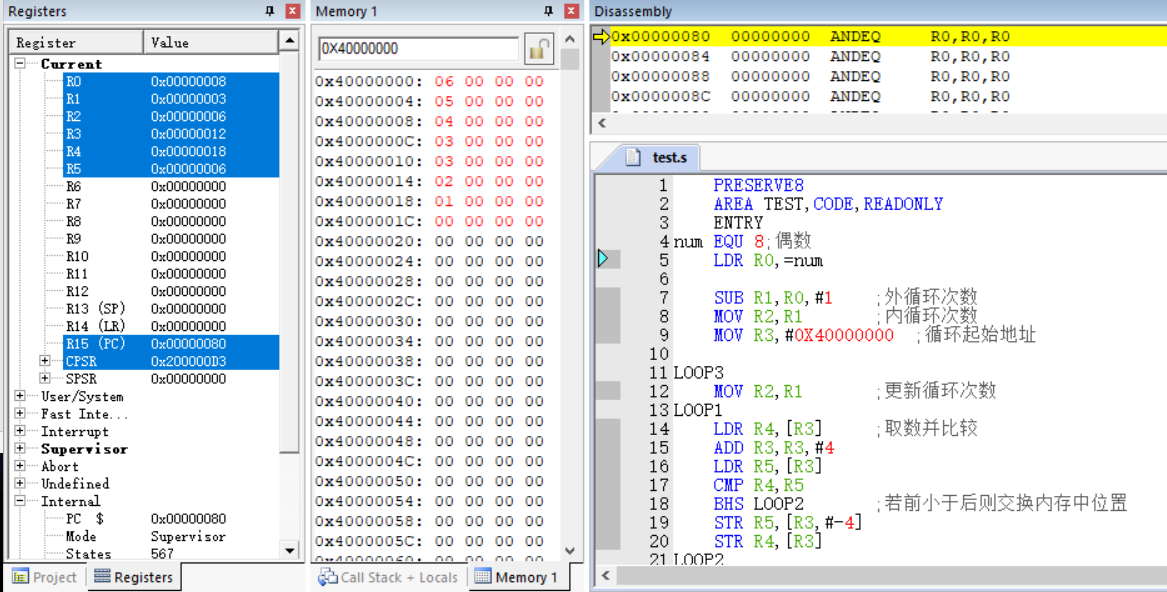


图6 均值滤波程序结束时界面截图

**(3)实验结论**

0x0A，0x08，0x02，0x01，0x06，0x04这六个数，去掉最大值A和最小值1后，其平均值为5，程序运行结果符合预期要求。

1. 总结及心得体会：

实验二用到的算法很简单，仅是大一下就接触到的冒泡排序，但是汇编语言相比于C语言复杂很多，汇编指令能够执行的任务有限，以至于需要小心谨慎排查代码逻辑。此外，我还学习了均值滤波与中值滤波在实际生活中的应用，收获很多。

**思考题：**在均值滤波实验中，求多个数的平均值是通过右移实现除法的，再给出一种其他实现除法的方式，并进行简单描述。

答：还可以使用乘法逆元结合定点数运算的方法来实现除法。

简单步骤：

步骤 1：确定定点数的精度位数 N，例如 N=16（即 Q16 格式）。

步骤 2：计算乘法逆元 = (2^N)/d + 0.5，即对1/d 进行定点数近似。

步骤 3：将总和与逆元相乘，结果右移 N 位，得到最终商。

1. 对本实验过程及方法、手段的改进建议：

本次实验是通过累加并比较大小来实现除法，当计算量较大时这种方法效率较低且耗时久，可以考虑更换算法进行优化，使得程序在参与计算数字较多时仍然具有较高的执行效率。

报告评分：

指导教师签字：