电子科技大学信息与软件工程学院

**实 验 报 告**

学 号 2020090910016

姓 名 瞿俊杰

（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： 瞿俊杰 学号： 2020090910016 指导教师： 兰刚**

**实验地点： 信软楼西305 实验时间：2022.4.20**

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. 熟悉Keil MDK开发环境，掌握开发平台MDK -ARM的使用；
5. 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. 熟练掌握ARM的指令系统以及指令寻址方式；
7. 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。
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。
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。
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。
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。
   1. 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。
   2. 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。
   3. 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：
   1. 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。
   2. 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。
   3. 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。
   4. 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。
9. 实验内容：
10. 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。
11. 掌握对ARM汇编工程编译、调试的方法。
12. 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。
13. 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。
14. 实验器材（设备、元器件）：
15. PC机一台；
16. Keil MDK-ARM uVision4开发工具。
17. 实验步骤：
18. 打开Keil MDK-ARM uVision4开发工具；
19. 新建一个工程文件；
20. 在新建的工程文件中，添加新的源程序文件
21. 编写代码
22. 选择“Build target”菜单对编写好的工程文件进行编译链接。
23. 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。
24. 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）
25. **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** | **结果说明** |
| 1 | MOV | 立即数寻址 | MOV R2,#100 | R2 ← 100 |
| 寄存器寻址  +S条件后缀 | MOV R0, #0x01  MOVS R2,R0 | 指令执行后标志位为nzcv |
| 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 |
| 2 | MVN | 立即数寻址  ++S条件后缀 | MOV R1,#1  MOV R2,#2  CMP R1,R2  MVNEQS R3,#5 | 由于R1不等于R2,所以MVN指令不执行， |
| 3 | AND | 寄存器寻址  +S后缀 | MOV R1,#1  MOV R2,#2  ANDS R3,R1,R2 | R3=0,标志位nZcv |
| 4 | EOR | 立即数寻址 | MOV R1,#0X11  EOR R2,R1,#0X01 | R2=0X10 |
| 5 | SUB | 寄存器移位寻址  +S后缀 | MOV R1,#0X30  MOV R2,#0X03  SUBS R3,R1,R2,LSL#2 | R3=0X24,标志位nzCv |
| 6 | RSB | 寄存器移位寻址  +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  +S后缀 | 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 | MOV R1,#0XF0000000  MOV R2,#0XF8000000  MOV R3,#0XF4000000  MOV R4,#0XF2000000  ADDS R5,R1,R2  ADCS R6,R3,R4 | R5=0XE8000000,R6=0XE6000001,标志位NzCv |
| 9 | SBC | 寄存器寻址+  S后缀 | 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 |
| 10 | RSC | 寄存器寻址  +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 |
| 寄存器移位寻址 | MOV R1,#0XF0000000  MOV R2,#0XF8000000  MOV R3,#0XF4000000  MOV R4,#0XF2000000  RSBS R5,R2,R1  RSCS R6,R4,R3,LSL#1 | R5=0XF8000000,R6=0XF5FFFFFF,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-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 |
| 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 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV R4,#0X40000000  STMIA R4!,{R0-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 | LDMDA | 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,#0X4000000C  LDMDA R4!,{R5-R8} | R5=0X01,R6=0X02,R7=0X03,R8=0X04 |
| 32 | LDMDB | 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,#0X40000010  LDMDB R4!,{R5-R8} | R5=0X01,R6=0X02,R7=0X03,R8=0X04 |
| 33 | 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 |
| 34 | 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 |
| 35 | STMFD | 堆栈寻址 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV SP,#0X40000010  STMFD SP!,{R0-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 |
| 36 | LDMFD | 堆栈寻址 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV SP,#0X40000010  STMED SP!,{R0-R3}  LDMFD SP!,{R5-R8} | R5=0,R6=0X01,R7=0X02,R8=0X03 |
| 37 | LDMEA | 堆栈寻址 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV SP,#0X40000000  STMEA SP!,{R0-R3}  LDMEA SP!,{R5-R8} | R5=0X01,R6=0X02,R7=0X03,R8=0X04 |
| 38 | STMEA | 堆栈寻址 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV SP,#0X40000000  STMEA SP!,{R0-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 |
| 39 | STMED | 堆栈寻址 | MOV R0,#0X01  MOV R1,#0X02  MOV R2,#0X03  MOV R3,#0X04  MOV SP,#0X40000010  STMED 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 |
| 40 | 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 |
| 41 | B | 相对寻址 | B LABEL  MOV R1,#0X02  LABEL  MOV R0,#0X01 | MOV R1,#0X02指令未执行 |
| 42 | 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 |
| 43 | 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 |
| 44 | 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 |
| 45 | MRS | 寄存器寻址 | MRS R0,CPSR | R0=0X000000D3 |

**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

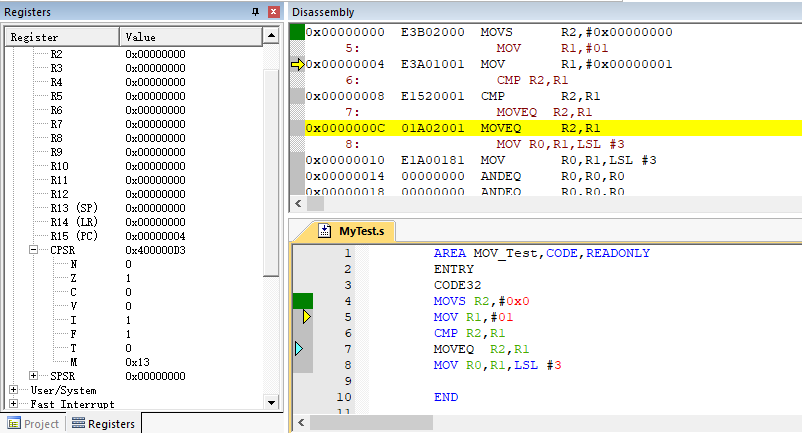


图1 MOVS R2，#0x0 指令执行测试截图

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。

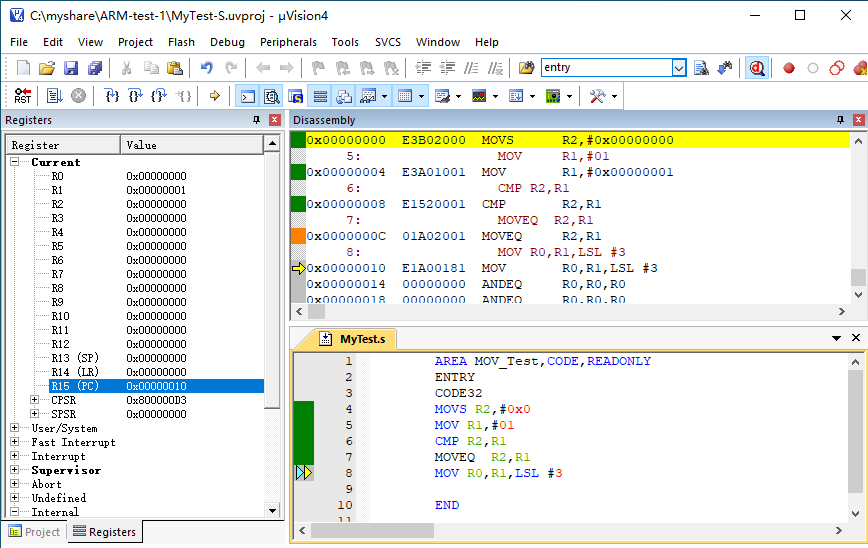


图2 MOVEQ R2，R1指令执行测试截图

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。

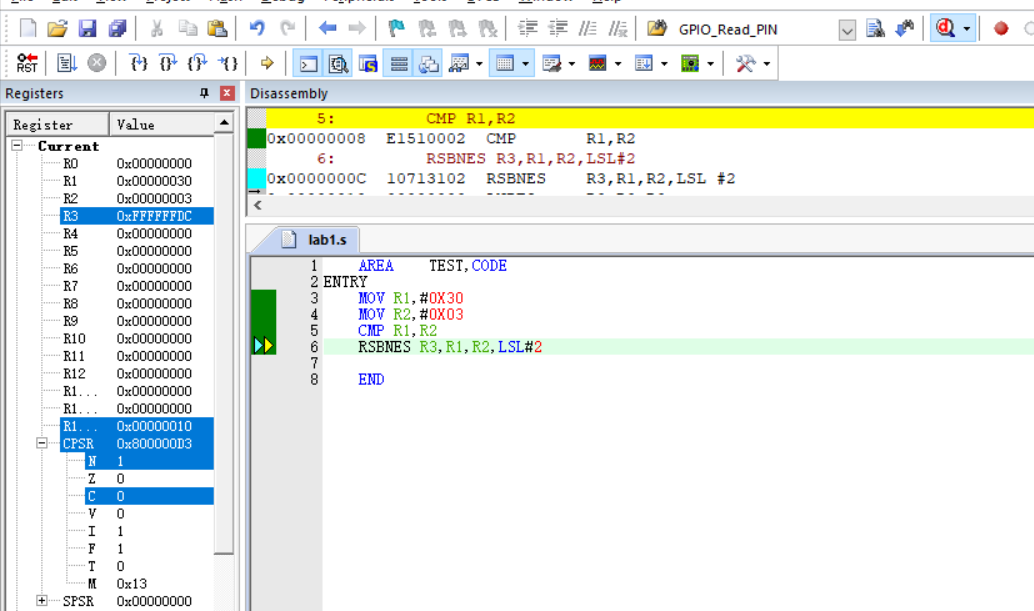


图3 RSBNES R3,R1,R2指令执行测试截图

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。

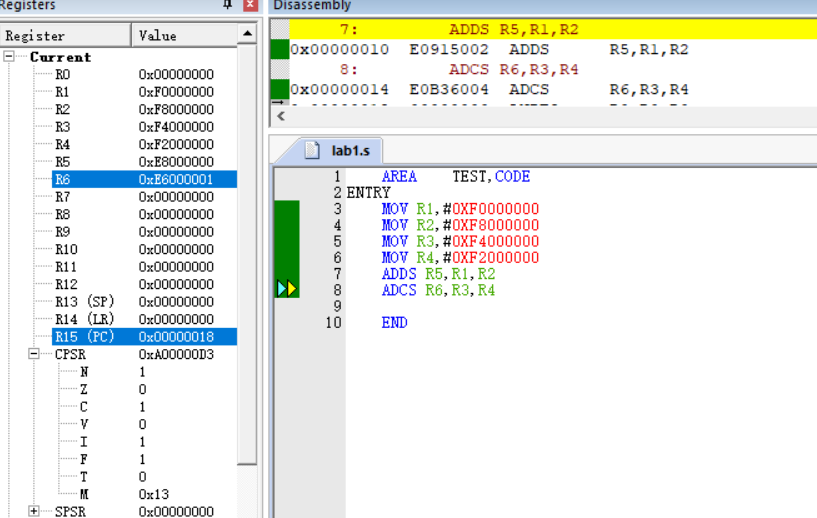


图4 ADCS R6,R3,R4指令执行测试截图

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。

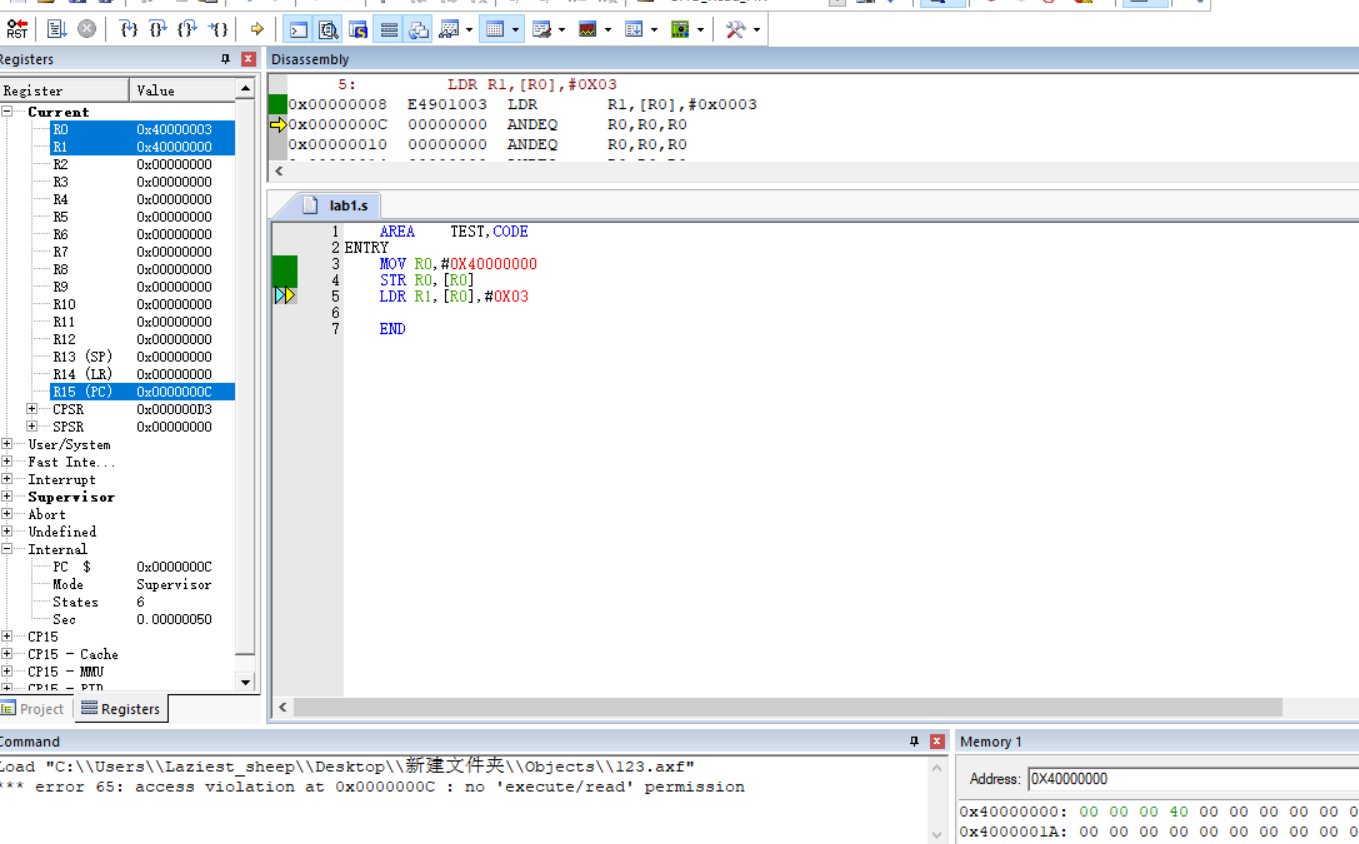


图5 LDR R1,[R0],#0X03指令执行测试截图

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

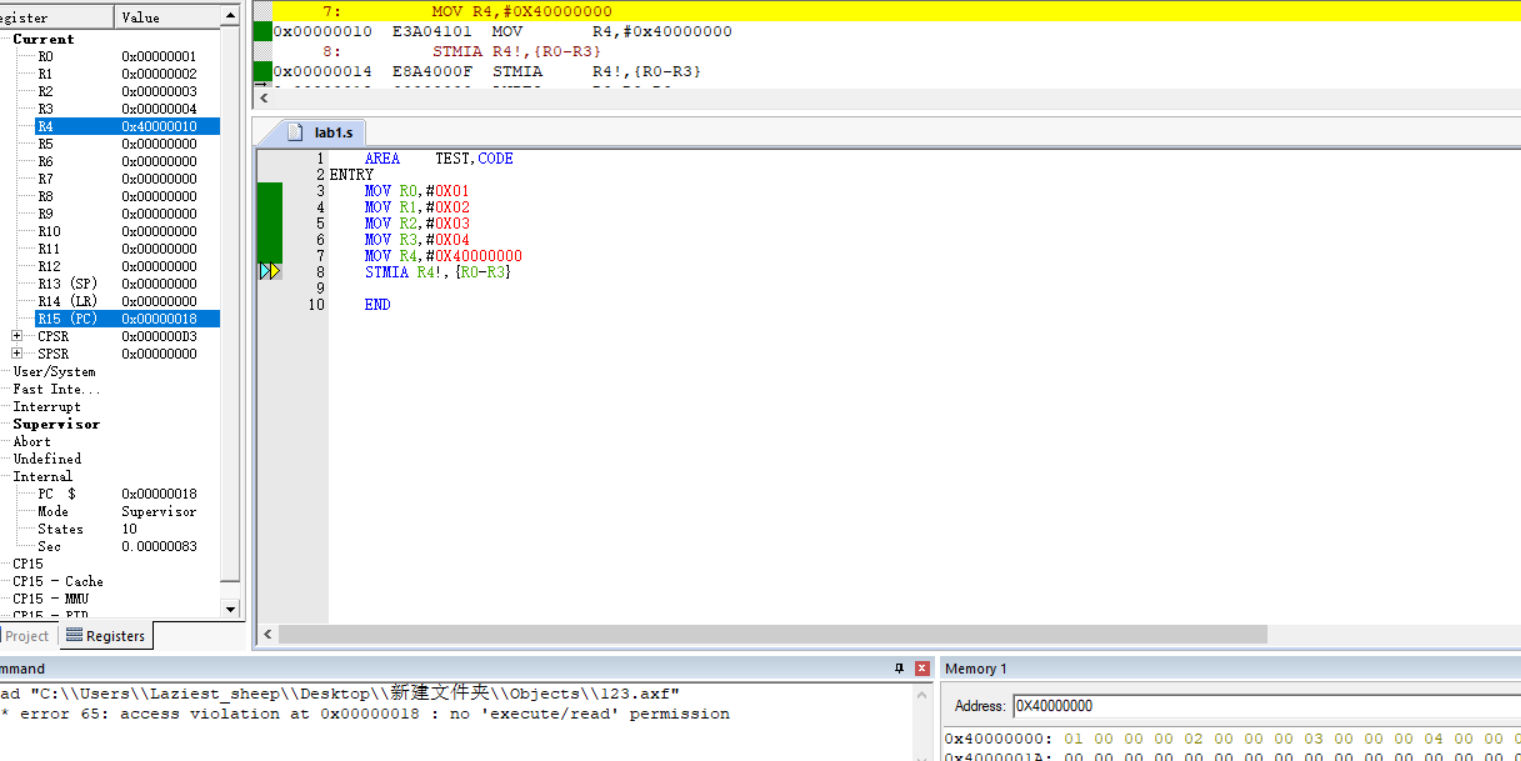


图6 STMIA R4!,{R0-R3}指令执行测试截图

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。

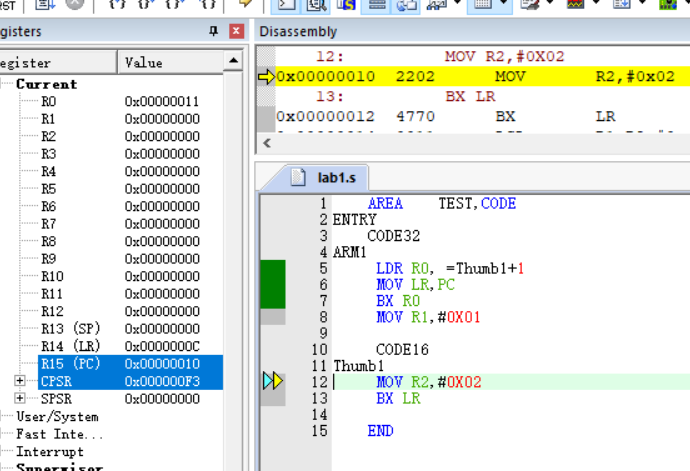


图7 BX 指令执行测试截图

**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：瞿俊杰 学号：2020090910016 指导教师：兰刚**

**实验地点： 信软楼西305 实验时间：2022.4.27**

1. 实验名称：ARM主程序调用ARM/C语言子程序
2. 实验学时：4
3. 实验目的：

(1) 了解ARM应用程序框架。

(2) 了解ARM汇编程序函数和C语言程序函数相互调用时，遵循的ATPCS标准；

(3) 了解和掌握ARM汇编程序调用C语言程序函数的基本方法；

(4) 了解和掌握ARM汇编程序调用C语言程序函数的参数传递过程。

四、 实验原理：

1. **ARM工程**

由于C语言便于理解，有大量的支持库，所以它是当前ARM程序设计所使用的主要编程语言。

对硬件系统的初始化、CPU状态设定、中断使能、主频设定 以及RAM控制参数初始化等C程序力所不能及的底层操作，还是要由汇编语言程序 来完成。

ARM工程 的各种源文件之间的关系，以及最后形成可执行文件的过程如下图1所示：

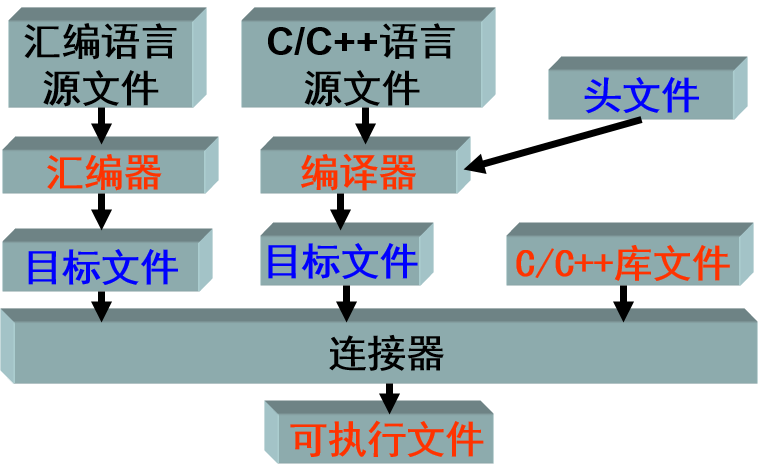


图1 汇编语言和C语言混合编译链接示意图

在应用系统的程序设计中，若所有的编程任务均用汇编语言 来完成，其工作量是可想而知的，这样做也不利于系统升级或应用软件移植。

通常汇编语言部分完成系统硬件的初始化；高级语言部分完成用户的应用。

执行时，首先执行初始化部分，然后再跳转到C/C++部分。整个程序结构显得清晰明了，容易理解。为方便工程开发，ARM公司的开发环境ARM ADS为用户提供了一个可以选用的应用程序框架。该框架把为用户程序做准备工作的程序分成了： 启动代码 和 应用程序初始化 两部分。

用于硬件初始化的汇编语言部分叫做 启动代码；用于应用程序初始化的C部分叫做初始化部分。整个程序如下图2所示：

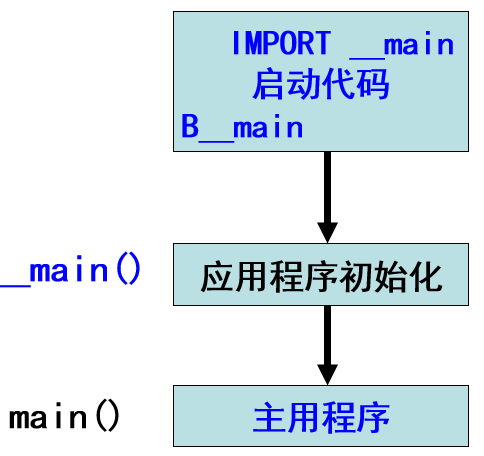


图2 ARM应用程序框架

1. **过程调用标准ATPCS**

在ARM工程中，C程序调用汇编函数和汇编程序调用C函数是经常发生的事情。为此人们制定了ARM-Thumb过程调用标准ATPCS（ARM-Thumb Procedure Call Standard）。

1. ATPCS规定，ARM的数据堆栈为FD型堆栈，即递减满堆栈。
2. ATPCS标准规定，对于参数个数不多于4的函数，编译器必须按参数在列表中的顺序，自左向右为它们分配寄存器R0~R3。其中函数返回时，R0还被用来存放函数的返回值。
3. 如果函数的参数多于4个，那么多余的参数则按自右向左的顺序压入数据堆栈，即参数入栈顺序与参数顺序相反。
4. 根据ATPCS的C语言程序调用汇编函数，参数由左向右依次传递给寄存器R0~R3的规则。
5. **子程序的调用与返回**

人们把可以多次反复调用的、能完成指定功能的程序段称为“子程序”。把调用子程序的程序称为“主程序”。

为进行识别，子程序的 第1条指令 之前必须赋予一个 标号，以便其他程序可以用这个标号调用子程序。

在 ARM 汇编语言程序中，主程序一般通过 BL 指令来调用子程序。该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器 LR 中，同时将程序计数器 PC 指向子程序的入口点。

为使子程序执行完毕能 返回 主程序的调用处，子程序末尾 处应有 MOV、LDMFD 等指令，并在指令中将返回地址重新复制到 PC 中。

在调用子程序的同时，也可以使用 R0~R3 来进行 参数的传递 和从子程序返回 运算结果。

五、 实验内容：

(1) ARM指令主程序调用ARM指令子程序；

(2) ARM指令主程序调用C语言子程序，输入的6个参数为1、2、3、4、5、6;

(3) 子程序的参数个数要求至少6个， C语言子程序实现的功能为：（i1+i2+i3+i4）\*i5-i6。

(4) 分析通过反汇编得到的C程序的ARM指令代码段，了解参数传递过程。

六、 实验器材（设备、元器件）：

(1) PC机一台；

(2) Keil MDK-ARM uVision4开发工具。

七、 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。

八、 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）

1. **ARM指令主程序调用ARM指令子程序；**
2. **程序代码**

程序代码如代码1所示：

代码1 ARM指令主程序调用ARM指令子程序

|  |
| --- |
| PRESERVE8  AREA TEST,CODE,READONLY  ENTRY  BL func ;跳转至arm子程序    LDR R1,=END1 ;跳转至END  BX R1  func ;arm子程序  MOV R1,#0X12  MOV R2,#0X12  BX LR  END1  END |

1. **运行过程及结果界面截图**

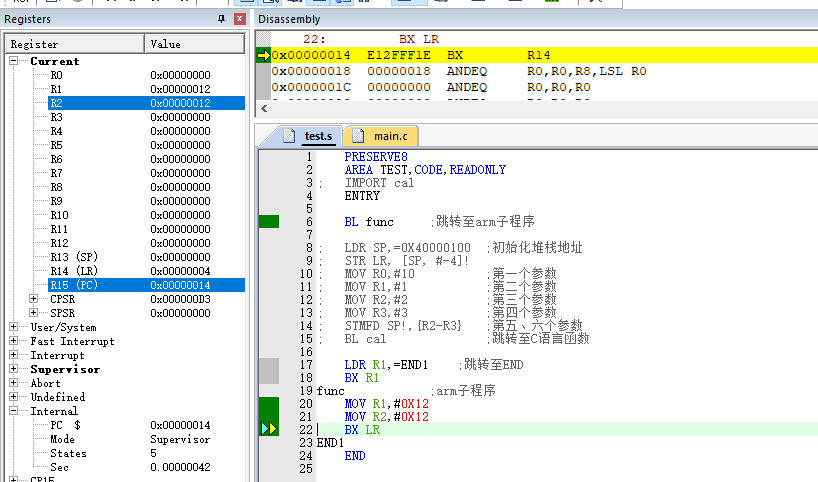


图1 ARM指令主程序调用ARM指令子程序运行结果截图

1. **实验结果分析**

由图1可见，子程序成功调用，向R1 R2传入0X12的数据

1. **实验结论**

ARM汇编可通过调用子程序的方式实现临时运行一段提前准备好的程序，再返回到主程序继续执行

1. **ARM指令主程序调用C语言子程序；**
2. **程序代码**

ARM指令主程序如代码2所示

代码2 RM指令主程序调用C语言子程序的RM指令主程序

|  |
| --- |
| PRESERVE8  AREA TEST,CODE,READONLY  IMPORT cal  ENTRY    LDR SP,=0X40000100 ;初始化堆栈地址  STR LR, [SP, #-4]!  MOV R0,#10 ;第一个参数  MOV R1,#1 ;第二个参数  MOV R2,#2 ;第三个参数  MOV R3,#3 ;第四个参数  STMFD SP!,{R2-R3} ;第五、六个参数  BL cal ;跳转至C语言函数    LDR R1,=END1 ;跳转至END  BX R1  END1  END |

C语言子程序代码如代码3所示

代码3 RM指令主程序调用C语言子程序的C语言子程序

|  |
| --- |
| int cal(int i1,int i2,int i3,int i4,int i5,int i6){  return (i1+i2+i3+i4)\*i5-i6;  } |

1. **运行过程及结果界面截图**

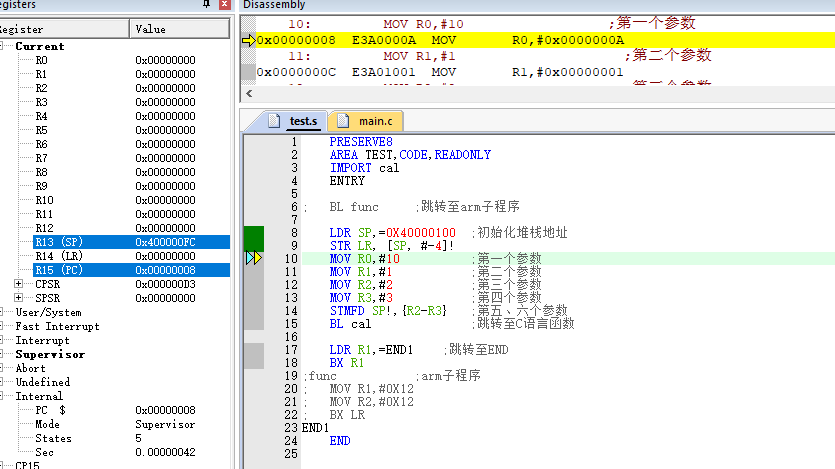


图2 程序执行完STR LR, [SP, #-4]!的界面截图

图3是程序即将调用C子程序的截图。

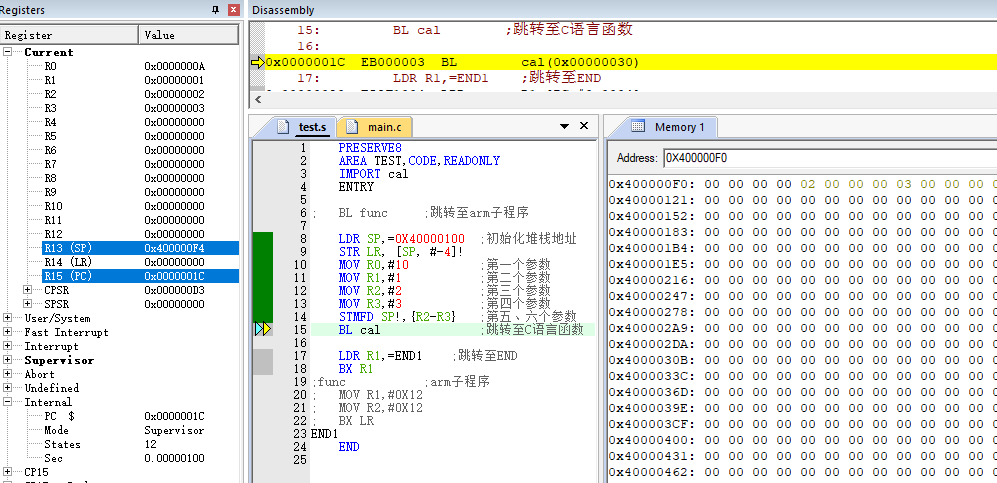


图3 程序即将调用C子程序的截图

图4是程序跟踪进入C语言子程序的截图。

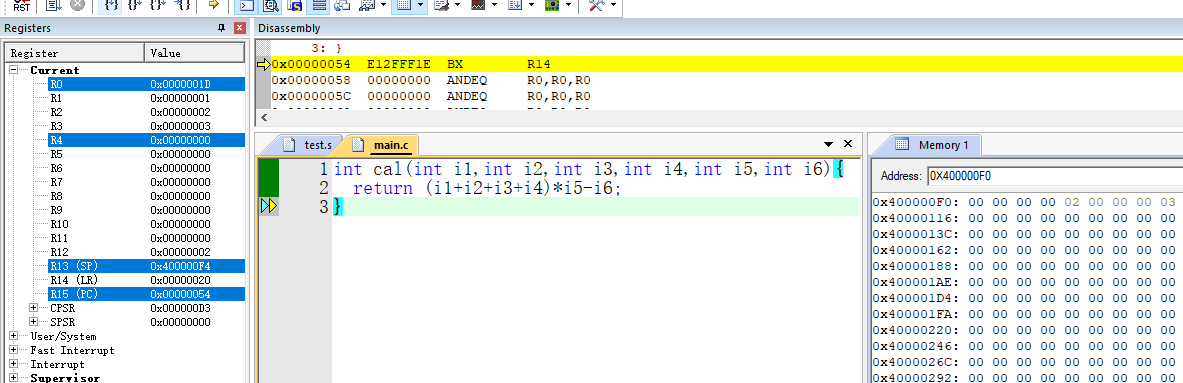


图4 程序跟踪进入C语言子程序的截图

1. **实验结论**

由图3可见，调用C语言子程序前将R0-R3以及堆栈数据设置好以便直接作为参数传入子函数，图4可见C语言子程序成功调用并计算出 (10+1+2+3)\*2-3=29=0X1D结果返回给R0

1. **结论分析**

ARM汇编调用C语言子程序时，前四个参数分别从R0、R1、R2、R3传入，本实验中为10、1、2、3，后续参数按满递减堆栈方式，SP指针从低地址到高地址，参数从左到右依次传入，本实验中为2、3，最终计算结果为(10+1+2+3)\*2-3=29=0X1D。

九、 总结及心得体会：

通过本实验的学习，我掌握了ARM汇编调用ARM子程序以及ARM汇编调用C语言子程序的方法，实践了解了参数的传递过程，同时对堆栈有了进一步的认识。

十、 对本实验过程及方法、手段的改进建议：

本实验中不存在函数调用嵌套的问题，故STR LR, [SP, #-4]!指令的调用显得有如鸡肋，建议重新考虑其应用场景。

报告评分：

指导教师签字：

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名： 瞿俊杰 学号：2020090910016 指导教师：兰刚**

**实验地点： 信软楼西305 实验时间：2022.5.11**

一、 实验名称：C语言主程序调用ARM子程序

二、 实验学时：4

三、 实验目的：

(1) 了解ARM应用程序框架。

(2) 了解ARM汇编程序函数和C语言程序函数相互调用时，遵循的ATPCS标准；

(3) 了解和掌握C语言程序调用ARM语言程序函数的基本方法；

(4) 了解和掌握C语言程序调用ARM汇编程序函数的参数传递过程；

(5) 掌握内联汇编和嵌入式汇编的编程方法。

四、 实验原理：

**1. ARM工程**

由于C语言便于理解，有大量的支持库，所以它是当前ARM程序设计所使用的主要编程语言。

对硬件系统的初始化、CPU状态设定、中断使能、主频设定 以及RAM控制参数初始化等C程序力所不能及的底层操作，还是要由汇编语言程序 来完成。

ARM工程 的各种源文件之间的关系，以及最后形成可执行文件的过程如下图1所示：

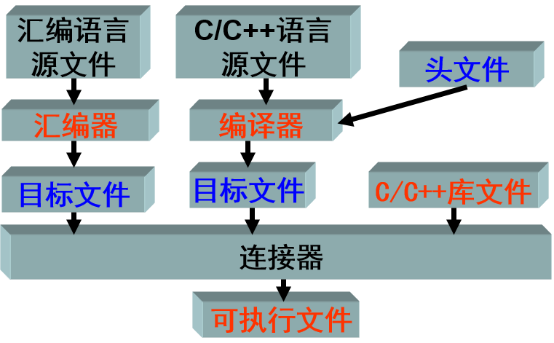


图1 汇编语言和C语言混合编译链接示意图

在应用系统的程序设计中，若所有的编程任务均用汇编语言 来完成，其工作量是可想而知的，这样做也不利于系统升级或应用软件移植。

通常汇编语言部分完成系统硬件的初始化；高级语言部分完成用户的应用。

执行时，首先执行初始化部分，然后再跳转到C/C++部分。整个程序结构显得清晰明了，容易理解。为方便工程开发，ARM公司的开发环境ARM ADS为用户提供了一个可以选用的应用程序框架。该框架把为用户程序做准备工作的程序分成了： 启动代码 和 应用程序初始化 两部分。

用于硬件初始化的汇编语言部分叫做 启动代码；用于应用程序初始化的C部分叫做初始化部分。整个程序如下图2所示：

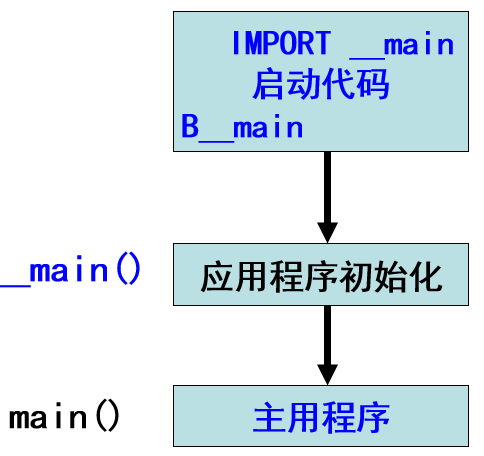


图2 ARM应用程序框架

**2. 过程调用标准ATPCS**

在ARM工程中，C程序调用汇编函数和汇编程序调用C函数是经常发生的事情。为此人们制定了ARM-Thumb过程调用标准ATPCS（ARM-Thumb Procedure Call Standard）。

1. ATPCS规定，ARM的数据堆栈为FD型堆栈，即递减满堆栈。
2. ATPCS标准规定，对于参数个数不多于4的函数，编译器必须按参数在列表中的顺序，自左向右为它们分配寄存器R0~R3。其中函数返回时，R0还被用来存放函数的返回值。
3. 如果函数的参数多于4个，那么多余的参数则按自右向左的顺序压入数据堆栈，即参数入栈顺序与参数顺序相反。
4. 根据ATPCS的C语言程序调用汇编函数，参数由左向右依次传递给寄存器R0~R3的规则。
5. **C/C++语言和汇编语言的混合编程之:** **内联汇编和嵌入式汇编**

除了上面介绍的函数调用方法之外，ARM 编译器 armcc 中含有的内嵌汇编器还允许在 C 程序中 内联或嵌入式汇编代码，以提高程序的效率。

所谓内联汇编程序，就是在 C 程序中直接编写汇编程序段而形成一个语句块，这个语句块可以使用除了 BX 和 BLX之外的全部ARM指令来编写，从而可以使程序实现一些不能从C获得的底层功能。

其格式为：

\_\_asm

{

汇编语句块

}

内联汇编 与 真实汇编 之间有很大区别，会受到很多限制。

（1）它不支持 Thumb 指令；除了程序状态寄存器 PSR 之外，不能直接访问其他任何物理寄存器等；

（2）如果在内联汇编程序指令中出现了以某个寄存器名称命名的操作数，那么它被叫做虚拟寄存器，而不是实际的物理寄存器。编译器在生成和优化代码的过程中，会给每个虚拟寄存器分配实际的物理寄存器，但这个物理寄存器可能与在指令中指定的不同。唯一的一个例外就是状态寄存器 PSR ，任何对 PSR 的引用总是执行指向物理 PSR；

（3）在内联汇编代码中不能使用寄存器 PC（R15）、LR（R14）和SP（R13），任何试图使用这些寄存器的操作都会导致出现错误消息；

（4）虽然内联汇编代码可以更改处理器模式，但更改处理器模式会禁止使用 C 操作数或对已编译 C 代码的调用，直到将处理器模式恢复为原设置之后。

嵌入式汇编

嵌入式汇编程序是一个编写在C程序外的单独汇编程序，该程序段可以像函数那样被 C 程序调用。

与内联汇编不同，嵌入式汇编具有真实汇编的所有特性，数据交换符合 ATPCS 标准，同时支持 ARM 和Thumb，所以它可以对目标处理器进行不受限制的低级访问。但是不能直接引用 C/C++ 的变量。

用 \_ \_asm 声明的嵌入式汇编程序，像 C 函数那样，可以有参数和返回值。定义一个嵌入式汇编函数的语法格式为：

\_\_asm return–type function–name(parameter-list)

{

汇编程序段

}

灵活地使用内联汇编和嵌入式汇编，有助于提高程序效率。

五、实验内容：

(1) C语言主程序调用ARM指令子程序；

(2) 程序的参数个数要求至少6个，ARM子程序实现的功能为：（i1+i2+i3+i4）\*i5-i6，并返回计算结果。观察参数是如何传递的。

(3) 在C语言的main函数中，使用内联汇编编写一个函数，使得c语言中定义的3个int类型的变量(a,b,tmp)有如下的关系：tmp =(a+b)\*8。

(4) 以嵌入式汇编的方式，编写一个子程序，实现字节序的转换。该子程序的函数的格式要求如下：int convertNum(int i);要求把输入的整数i（32位，四个字节顺序为ABCD），转换为另一种字节序（转换后的字节序为DCBA），返回结果为转换后的int值。在主程序中调用该子程序，并验证其结果的正确性。

六、 实验器材（设备、元器件）：

(1) PC机一台；

(2) Keil MDK-ARM uVision4开发工具。

七、 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。

八、 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）

1. **程序代码**

C语言代码如代码1所示。

代码1 C语言主程序调用ARM指令子程序的C语言程序部分

|  |
| --- |
| extern int func(int i1,int i2,int i3,int i4,int i5,int i6);  \_\_asm int convertNum(int i){  MOV R1,#0 //寄存器清零  MOV R2,#0  MOV R3,#0  MOV R4,#0  AND R1,R0,#0XFF //0-7位放入R1  MOV R0,R0,LSR #8 //右移8位  AND R2,R0,#0XFF //8-15位放入R2  MOV R0,R0,LSR #8 //右移8位  AND R3,R0,#0XFF //16-23位放入R3  MOV R0,R0,LSR #8 //右移8位  AND R4,R0,#0XFF //24-31位放入R4  MOV R0,R0,LSR #8 //右移8位    ORR R0,R0,R1 //R1放入低8位  MOV R0,R0,LSL #8 //左移8位  ORR R0,R0,R2 //R2放入低8位  MOV R0,R0,LSL #8 //左移8位  ORR R0,R0,R3 //R3放入低8位  MOV R0,R0,LSL #8 //左移8位  ORR R0,R0,R4 //R4放入低8位  BX LR  }  int main(){  int i = func(1,2,3,4,5,6);  int a = 1,b=2,tmp=0;  \_\_asm{  ADD a,a,b  MOV b,8  MUL tmp,a,b  }  i = convertNum(4);  return 0;  } |

汇编程序代码如代码2所示。

代码2 C语言主程序调用ARM指令子程序的汇编程序部分

|  |
| --- |
| AREA TEST, CODE, READONLY  ENTRY  EXPORT func  func  STMFD SP!,{R4,R5} ;将R4、R5保存  MOV R4,#8  ADD SP,SP,R4 ;SP指针回到传入参数处  ADD R4,R0,R1 ;i1+i2->R4  ADD R5,R2,R3 ;i3+i4->R5  ADD R0,R4,R5 ;i1+i2+i3+i4->R0  LDMFD SP!,{R1,R2} ;i5->R1,i6->R2  MOV R4,#16  SUB SP,SP,R4 ;SP指针回到R4、R5处  LDMFD SP!,{R4,R5} ;恢复R4、R5  MUL R3,R0,R1 ;(i1+i2+i3+i4)\*i5->R3  SUB R0,R3,R2 ;(i1+i2+i3+i4)\*i5-i6->R0    BX LR  END |

1. **运行过程及结果界面截图**

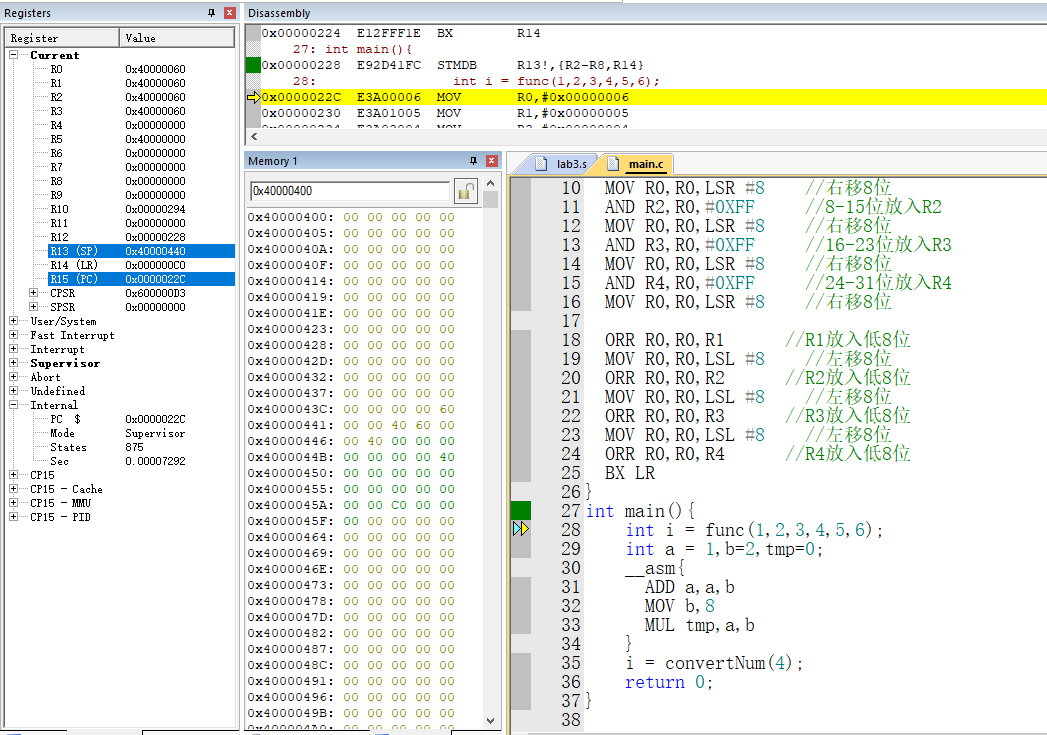


图3 程序即将调用汇编子程序

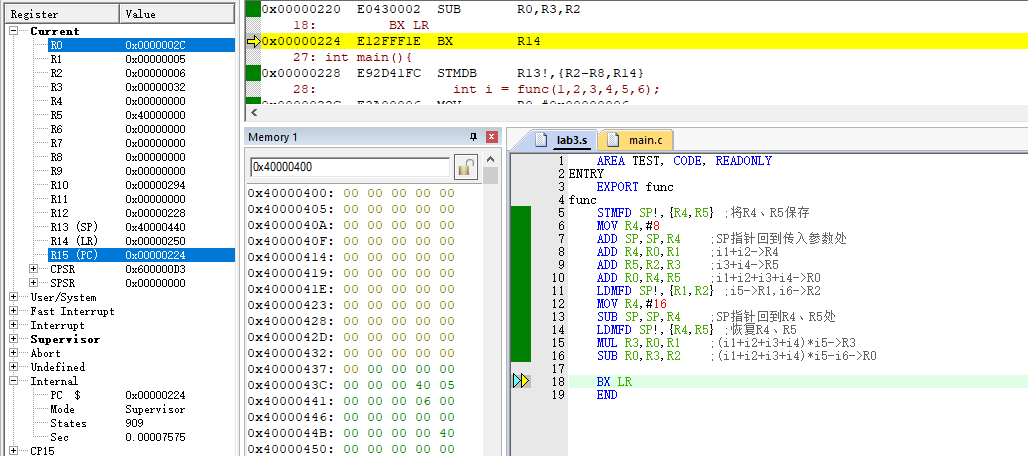


图4 程序执行完子程序

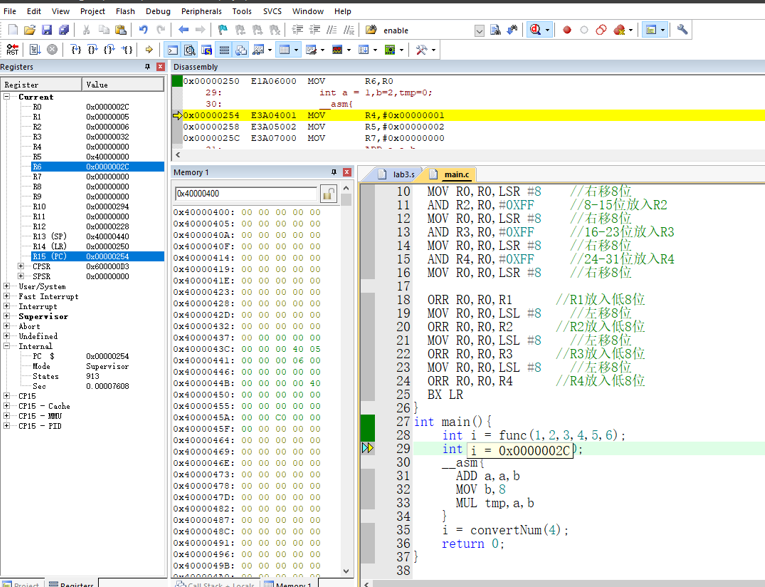


图5程序从子程序返回

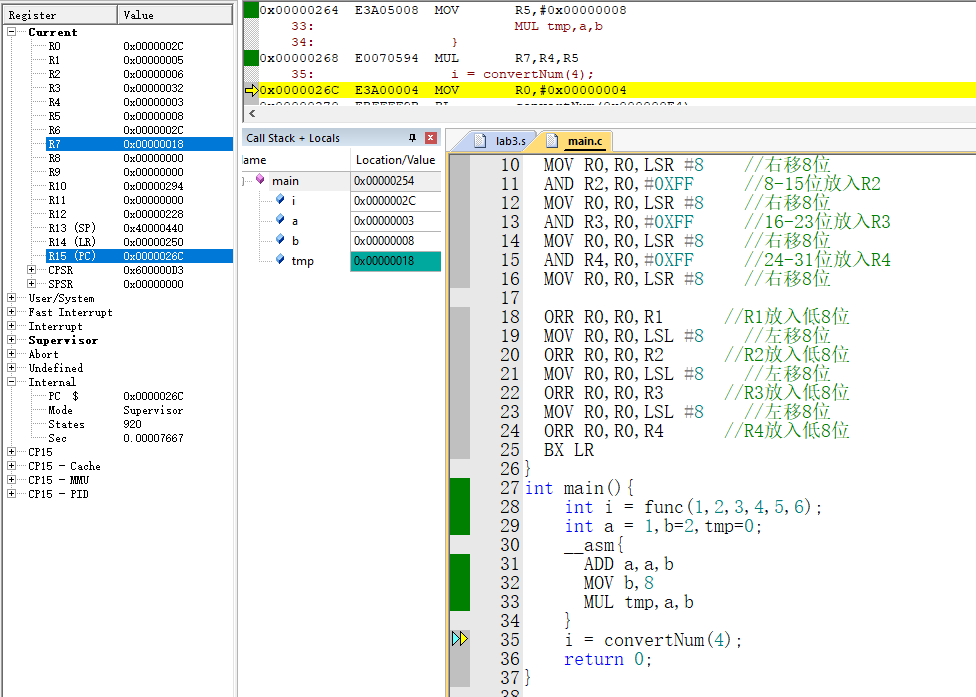


图6程序执行完内联汇编

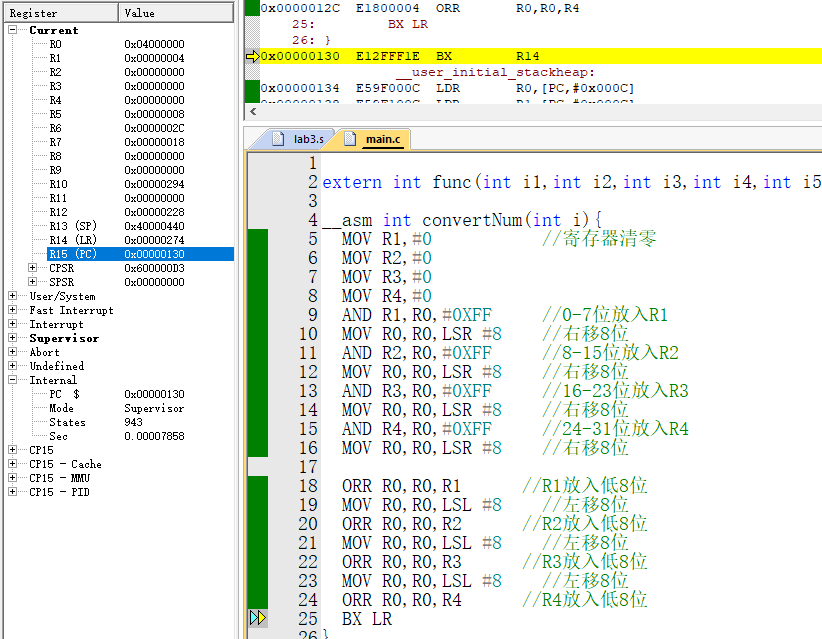


图7程序执行完嵌入式汇编子程序

**3、结论分析**

图4可看出汇编子程序成功计算传入参数(1+2+3+4)\*5-6=44=0X2C，图5可看出结果成功从R0返回给变量i，图6可看出内联汇编成功计算(a+b)\*8并传入tmp，图7可看出嵌入式汇编子程序成功将传入参数4实现字节序反转。

**4、实验结论**

C语言调用ARM汇编函数的参数传递规则与ARM汇编调用C语言子程序恰好相反，R0-R3对应从左往右前4个参数，剩余参数从右往左按地址从高到底压入堆栈。

内联汇编无法直接使用寄存器，若出现寄存器名称则会被编译器认为是一个未定义的变量，但是ADD、MUL等指令可直接对变量进行计算，不需先传入寄存器。

嵌入式汇编类似于加了\_\_asm前缀的普通函数，使用汇编语言编写，与在汇编文件内编写并在C文件中调用效果相同。

九、 总结及心得体会：

本次实验我掌握了C语言调用ARM汇编子程序以及内联汇编、嵌入式汇编的基本使用方法，对汇编语言的使用有了进一步理解。同时我在同学的帮助下发现，Option for Target中C/C++一栏Optimization选项用来控制编译优化程度，如果优化程度过高则会导致内联汇编部分不被编译从而不执行

十、 对本实验过程及方法、手段的改进建议：

本实验过程清晰有条理，方法恰当，暂无改进建议

报告评分：

指导教师签字：