目录

第7章　ARM 程序设计

虽然目前基于 ARM 处理器的程序大多采用 C 语言开发，但在系统启动程序编写、处理器初始化、驱动开发、底层算法实现等场合还是需要用到汇编语言。更重要的一点是，汇编指令可以直接对 ARM 处理器中的寄存器进行操作，掌握必要的汇编程序设计知识，能够更全面和更深入地理解 ARM 处理器的工作原理，为基于 ARM 处理器的嵌入式软硬件开发奠定良好的基础。本章重点介绍 ARM 程序开发环境，汇编程序设计，以及汇编语言和 C 语言混合编程。

7.1 ARM 程序开发环境

ARM 应用软件的开发工具根据功能的不同，分别有编译软件、汇编软件、链接软件、调试软件、嵌入式实时操作系统、函数库、评估板、JTAG 仿真器、在线仿真器等，目前世界上约有四十多家公司提供上述不同类别的产品。

用户选用 ARM 处理器开发嵌入式系统时，选择合适的开发工具可以加快开发进度，节省开发成本。因此一套含有编辑软件、编译软件、汇编软件、链接软件、调试软件、工程管理及函数库的集成开发环境（Integrated Development Environment，IDE）是必不可少的，至于嵌入式实时操作系统、评估板等其他开发工具则可以根据应用软件规模和开发计划选用。

本节先简要介绍几种常见的或者曾经比较常见的 ARM 程序开发环境，再重点介绍目前使用的较多的基于 Windows 平台的集成开发环境 MDK。

7.1.1 常用 ARM 程序开发环境简介

ARM 程序开发环境主要分为基于 Windows 平台和基于 Linux 平台的两大类。基于 Windows 平台的 ARM 程序开发环境主要有 SDT、ADS、RVDS、RealView MDK、ARM Development Studio 5 等，基于 Linux 平台的 ARM 程序开发环境主要是 ARM-Linux-GCC。

1. SDT

ARM SDT 的英文全称是 ARM Software Development Kit 是 ARM 公司最早推出的开发工具，用户可以很方便地利用 SDT 在 ARM 芯片上进行应用软件开发。ARM SDT 经过 ARM 公司逐年的维护和更新，目前的最新版本是 2.5.2，但从版本 2.5.1 开始，ARM 公司宣布推出一套新的集成开发工具 ARM ADS 1.0 取代 ARM SDT。

2. ADS

ADS 的英文全称是 ARM Developer Suite，ARM 公司大约在 1999 年推出，用来代替 SDT 集成开发工具，ADS 包括 4 个模块：SIMULATOR、C 编译器、实时调试器和应用函数库。其最终版为 1.2.1，目前也已经不再更新，逐步由 RVDS 和 RealView MDK 替代。

3. RVDS

RealView Developer Suite（RVDS）是 ARM 公司继 ADS 之后推出的集成开发工具，RVDS 是为从事 SoC、FPGA 和 ASIC 设计的工程师开发复杂的嵌入式应用和平台接口而设

计的。RVDS 主要包含 4 个模块：

- 集成开发环境：RVDS 集成了 Eclipse IDE，用于代码的编辑和管理。RVDS 支持语句高亮和多颜色显示，以工程的方式管理代码，支持第三方 Eclipse 功能插件；
- 编译器：RVCT（RealView Compilation Tools）是 RVDS 集成的编译器，支持二次编译和代码数据压缩技术，提供多种优化级别。RVCT 支持全系列的 ARM 和 XSCALE 架构，支持汇编、C 和 C++；
- 调试器：RVD（RealView Debugger）是 RVDS 中的调试软件，功能强大，支持 Flash 烧写和多核调试，支持多种调试手段，可以快速错误定位；
- 指令集仿真器：RVISS（RealView instruction set simulator）支持外设虚拟，可以使软件开发和硬件开发同步进行，同时可以分析代码性能，加快软件开发速度。

在 RVDS 的基础上，ARM 公司于 2011 年推出了新一代的开发工具 ARM Development Studio 5（DS-5）以取代 RVDS，并在之后逐渐停止了对 RVDS 的更新。

4. MDK-ARM

MicroController Development Kit（MDK）原名 RealView MDK，是由 Keil 公司（现已被 ARM 公司收购）推出的，也称为 MDK-ARM、KEIL MDK、KEIL ARM。Keil 公司是一家业界领先的微控制器（MCU）软件开发工具的独立供应商，公司由两家私人公司联合运行，分别是德国慕尼黑的 Keil Elektronik GmbH 和美国德克萨斯的 Keil Soft-ware，Inc.，一度非常流行的单片机开发工具 Keil C51 就是 Keil 公司的明星产品。

MDK[1]是 Keil 公司为微控制器开发而推出一款软件工具套件，其主要特点如下：

- 支持内核：ARM7、ARM9、Cortex-M4/M3/M1 和 Cortex-R0/R3/R4 等 ARM 微控制器内核，随着 ARM 公司内核产品的不断增加，所支持的内核种类也会有所变化；
- IDE：μVision IDE；
- 编译器：ARM C/C++ 编译器（ARMCC）；
- 调试器：μVision Debugger，仅可连接到 KEIL 设备库中的芯片组；
- 仿真器：μVision CPU & Peripheral Simulation；
- 硬件调试单元：ULink /JLink。

5. ARM-Linux-GCC

GNU Compiler Collection（GCC）是一套由 GNU[2] 开发的编译器集，不仅支持 C 语言编译，还支持 C++、Ada、Object C 等许多语言。GCC 还支持多种处理器架构，包括 X86、ARM、和 MIPS 等处理器架构，是在 Linux 平台下被广泛使用的软件开发工具。

ARM-Linux-GCC 是基于 ARM 目标机的交叉编译软件，所谓交叉编译简单来说就是在

[1] 2005 年 Keil 公司被 ARM 公司收购之后，ARM 公司的开发工具从此分为两大分支：MDK 系列和 RVDS 系列。MDK 系列是 ARM 公司推荐的针对微控制器，或者基于单核 ARMTDMI、Cortex-M 或者 Cortex-R 处理器的开发工具链，基于 Keil 公司一直使用的 μVision 集成开发环境；而 RVDS 系列（后升级为 DS-5）包含全部功能，支持所有 ARM 内核。

2 GNU 是 GNU's Not Unix!的递归缩写，GNU 计划的主要目标是开发一个自由的操作系统，其内容软件完全以 GPL（GNU General Public License GNU 通用公共许可证） 方式发布。

3 所谓平台包含了两个概念：体系结构（Architecture）和操作系统（Operating System），同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。

一个平台上生成另一个平台[3]上的可执行代码。一个常见的例子是：嵌入式软件开发人员通常在个人计算机上为运行在基于 ARM、PowerPC 或 MIPS 的目标机编译软件。

ARM-Linux-GCC 使用命令行来调用命令执行，相比于 RVDS 和 MDK 而言上手更难，使用也颇为不便。但由于 ARM-Linux-GCC 不需要授权费用，因而受到使用 Linux 开发嵌入式系统应用工程师的欢迎。

6. ARM Development Studio 5（DS-5）

2011 年，随着 ARM 公司新发布几款 Cortex-A 系列内核以及即将推出的下一代 ARMv8 架构体系，面对新的内核以及日益复杂的 SoC 系统开发需求，RVDS 逐渐显现出力不从心。为此，ARM 公司推出了新一代的开发套件 DS-5。DS-5 是一款可以支持所有 ARM 内核芯片的开发工具，DS-5 工具链包括一流的 ARM C / C ++编译器，强大的 Linux / Android™/ RTOS 调试器，ARM Streamline™系统性能分析器、实时系统仿真模型和基于 Eclipse 的集成开发环境（IDE）。借助于该工具套件，可以很轻松地进行基于 ARM 和 Linux 系统的开发，缩短开发和测试周期，并且帮助工程师创建资源利用效率更高的软件。

DS-5 也是目前 ARM 公司推出的功能最强大、最全面的开发环境，主要特点如下：

- ❒ 支持内核种类： ARM 公司现有的全部内核；
- ❒ IDE：Eclipse IDE；
- ❒ 编译器：ARM Compiler 6、ARM Compiler 5、GCC（Linaro GNU GCC Compiler for Linux）；
- ❒ 调试器：支持 ETM 指令和数据跟踪、PTM 程序跟踪；
- ❒ 仿真器：支持 ULINK2、ULINKPRO 和 DSTREAM 仿真器；
- ❒ 性能分析器：Streamline；
- ❒ 模拟器：RTSM，支持 Cortex-A8 固定虚拟平台（FVP）、多核 Cortex-A9 实时模拟器、ARMv8 固定虚拟平台（FVP）。

MDK 和 DS-5 都是 ARM 开发工具中的重要组成部分，两者的主要功能差异如表 7.1 所示。

表 7.1 KEIL MDK 与 DS-5 功能对比

功能	KEIL MDK	DS-5
编译器	支持 ARM CC	支持 ARM CC 或者 GCC
开发环境	µVision	Eclipse
内核支持	ARM7/9， ARM Cortex-M 全系列	支持所有 ARM 内核
RTOS RTX 内核库	有	无
调试器	µVision 调试器	DS-5 调试器
模拟器 Simulator	µVision Simlulator	实时系统模型 RTSM
调试硬件	Uink2/Ulink-me/Ulink-pro，支持 STLINK，JLINK 等第三方仿真调试器	DSTREAM， Ulinkpro-D
操作系统调试开发支持	Keil RTX，CMSIS RTOS，Freescale MQX，CMX RTOS，Segger emBos，	Keil RTX，FreeScale MQX，Linux & Android

	Quadros RTXC	
多核支持	不支持	支持
GDB Server	不支持	支持
逻辑分析仪	有	无

通过以上对比可以看出：KEIL MDK 主要面向基于 ARM7、ARM9 和 ARM Cortex-M 系列处理器的开发需求，包括它自带的 RTX 实时操作系统和中间库，这些都属于 MCU 应用领域；DS-5 主要用于创建基于 Linux/Android 操作系统的复杂嵌入式系统应用和系统平台驱动接口，片上系统等的开发应用。如果是开发 MCU 应用，推荐使用 KEIL MDK；如果是开发片上系统、Linux/Android 驱动和应用，推荐使用 DS-5+DSTREAM。

总而言之，用户因根据具体项目的硬件平台类型和功能需求选择合适的的开发环境，同时兼顾用户的使用习惯（比如很多从单片机开发转到嵌入式开发的开发者更习惯使用 MDK）。

7.1.2 MDK 开发环境简介

MDK 是德国知名软件公司 KEIL（现已并入 ARM 公司）开发的微控制器软件开发平台，是目前基于 ARM 内核开发的主流工具之一。KEIL 提供了包括 C 编译器、宏汇编、连接器、库管理和一个功能强大的仿真调试器在内的完整开发方案，通过一个集成开发环境（μVision）将这些功能组合在一起，它的界面和常用的微软 VC++的界面相似，界面友好，在调试程序、软件仿真方面也有很强大的功能。

1. KEIL MDK 的软件开发周期

使用 MDK 来开发嵌入式软件，开发周期和其他的平台软件开发周期是差不多的，大致有以下几个步骤：

（1）创建工程，选择目标芯片，并且做一些必要的工程配置；
（2）编写 C 或者汇编源文件；
（3）编译应用程序；
（4）修改源程序中的错误；
（5）联机调试；

2. μVision 集成开发环境

μVision IDE 是一款集编辑，编译和项目管理于一身的基于窗口的软件开发环境。μVision 集成了 C 语言编译器，宏编译，链接/定位，以及 HEX 文件产生器。μVision 具有如下特性：

（1）功能齐全的源代码编辑器；
（2）配置开发工具的设备库；
（3）用于创建工程和维护工程的项目管理器；
（4）所有的工具配置都采用对话框进行；
（5）集成了源码级的仿真调试器，包括高速 CPU 和外设模拟器；
（6）用于往 Flash ROM 下载应用程序的 Flash 编程工具；
（7）完备的开发工具帮助文档，设备数据表和用户使用向导；

μVision 具有良好的界面风格，图 7.1 是一个 MDK 工程打开时的典型窗口界面。

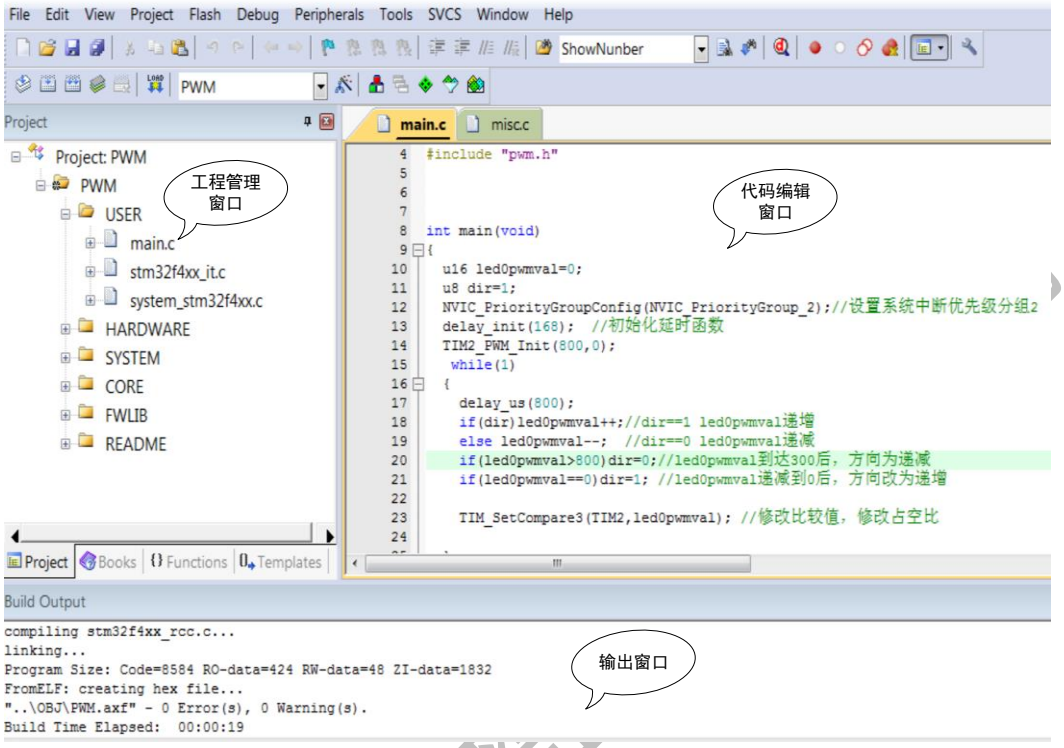

图 7.1 μVision IDE 打开工程后界面

图 7.1 中工程管理窗口用于工程结构设置和工程文件管理；代码编辑窗口用于查看和编辑工程中的文件；输出窗口用于显示编译结果，快速查找错误位置，同时还是调试命令输入输出窗口，也可以用于显示查找结果。

错误!未找到引用源。是一个 MDK 工程在调试状态时的典型窗口界面。

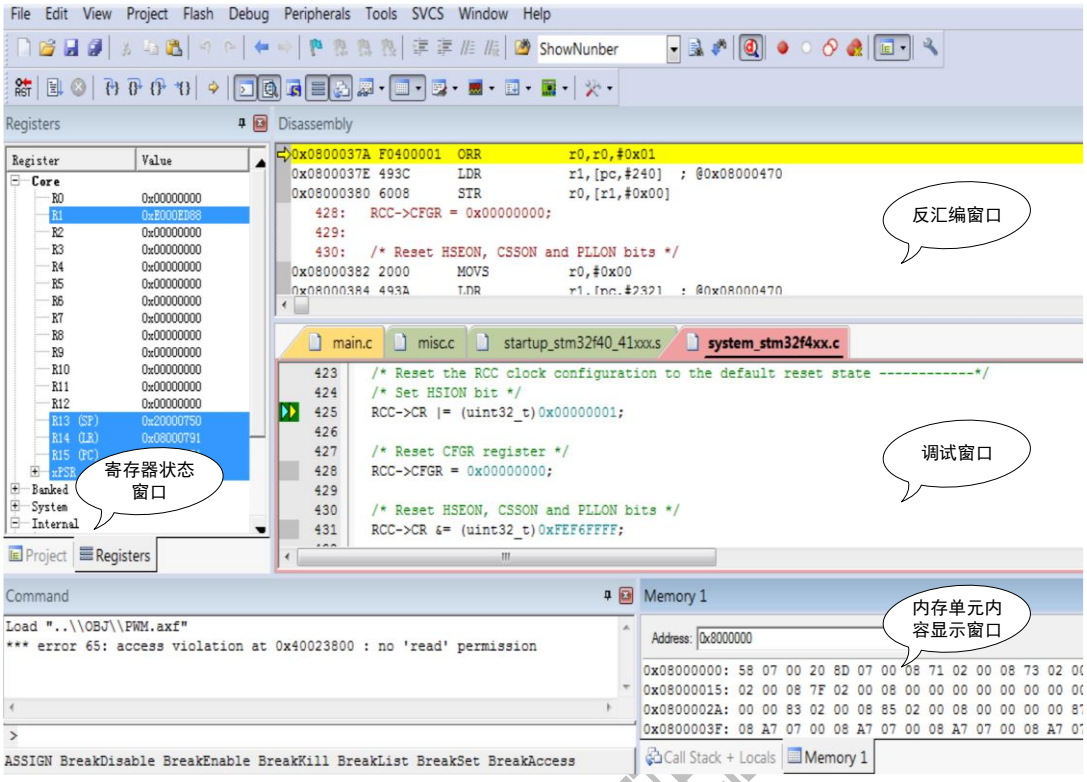

图 7.2 μVision IDE 调试界面

图 7.2 中的寄存器状态窗口在调试状态时可以查看 CPU 寄存器状态；内存单元显示窗口用于显示指定地址里的内容；查看和调用栈窗口用于查看和修改变量的值；反汇编窗口如图 7.3 所示，该窗口可以显示反汇编后的代码、源代码和相应反汇编代码的混合代码，可以在该窗口进行在线汇编、跟踪已执行的代码、按汇编代码的方式单步执行。

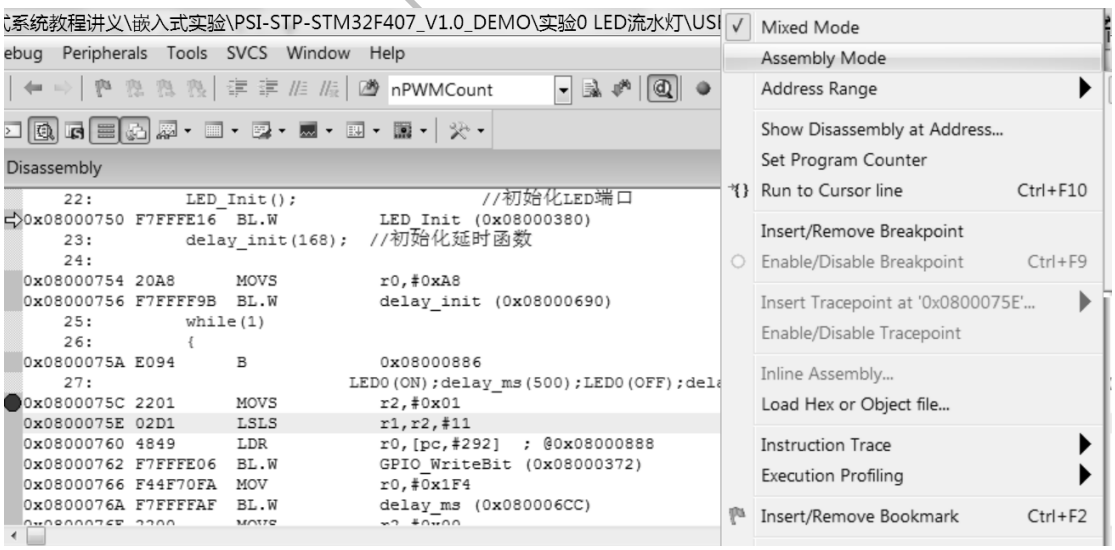

图 7.3 反汇编窗口

3. ARM 仿真器

仿真器可以替代目标系统中的 MCU，仿真其运行。它运行起来和实际的目标处理器一样，但是增加了其它功能，使用户能够通过计算机或其它调试界面来观察 MCU 中的程序和数据，并控制 MCU 的运行，它是调试嵌入式软件的一个经济、有效的手段。

仿真器具有软件模拟器的所有功能，同时具有以下优点：

- 不使用目标系统或 CPU 资源；
- 硬件断点、跟踪功能（TRACE）；
- 条件触发；
- 实时显示存储器和 I/O 口内容；
- 硬件性能分析；

总结来说，ARM 仿真器就是以一种极其经济的方式仿真实际 MCU 的运行、调试，以降低开发成本，提升嵌入式开发的效率。

ARM 官方仿真器有以下几款：DSTREAM、RVI & RVT2、ULINKPRO、ULINK2、ULINK-ME、ULINK 等，用于低端嵌入式微控制器的话，用 ULINK2、ULINKPpro 即可，可以调试 Cortex-M、Cortex-R 芯片，用于高端的嵌入式微控制器如 ARM9、ARM11、Cortex-A 等，就需要用到 DSTREAM 仿真器。

JTAG 仿真器也称为 JTAG 调试器，是通过 ARM 芯片的 JTAG 边界扫描口进行调试的设备。JTAG 仿真器比较便宜，连接比较方便，通过现有的 JTAG 边界扫描口与 ARM CPU 核通信，属于完全非插入式（即不使用片上资源）调试，它无需目标存储器，不占用目标系统的任何端口。另外，由于 JTAG 调试的目标程序是在目标板上执行，仿真更接近于目标硬件，因此，许多接口问题，如高频操作限制、AC 和 DC 参数不匹配，电线长度的限制等被最小化了。使用集成开发环境配合 JTAG 仿真器进行开发是目前采用最多的一种调试方式。

JLink 是 SEGGER 公司为支持仿真 ARM 内核芯片推出的 JTAG 仿真器。配合 ADS、KEIL MDK、RealView 等集成开发环境支持所有 ARM7/ARM9/ARM11，Cortex M0/M1/M3/M4，Cortex A5/A8/A9 等内核芯片的仿真，与 Keil MDK 等编译环境无缝连接，操作方便、连接方便、简单易学，是开发 ARM 应用最好最实用的开发工具，据说功能也是众多仿真器里最强悍的。

7.2 ARM 汇编程序中的伪指令

在 ARM 汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令，他们所完成的操作称为伪操作。 伪指令不像机器指令那样在处理器运行期间由机器执行，而是汇编程序对源程序汇编期间由汇编程序处理，包括：定义变量、分配数据存储空间、控制汇编过程、定义程序入口等，这些伪指令仅在汇编过程中起作用，一旦汇编结束，伪指令的使命就完成了。

7.2.1 符号定义（Symbol Definition）伪指令

符号定义伪指令用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。常见的符号定义伪指令有如下几种：

- 用于定义全局变量的 GBLA、GBLL 和 GBLS。
- 用于定义局部变量的 LCLA、LCLL 和 LCLS。
- 用于对变量赋值的 SETA、SETL、SETS。
- 为通用寄存器列表定义名称的 RLIST。

1. GBLA、GBLL 和 GBLS

语法格式：

GBLA（GBLL 或 GBLS） 全局变量名;

GBLA、GBLL 和 GBLS 伪指令用于定义一个 ARM 程序中的全局变量，并将其初始化。其中：

GBLA 伪指令用于定义一个全局的数字变量，并初始化为 0；

GBLL 伪指令用于定义一个全局的逻辑变量，并初始化为 F（假）；

GBLS 伪指令用于定义一个全局的字符串变量，并初始化为空；

由于以上三条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。

例 7.1

```
GBLA Test1 ;定义一个全局的数字变量，变量名为 Test1
Test1 SETA 0xaa ;将该变量赋值为 0xaa
GBLL Test2 ;定义一个全局的逻辑变量，变量名为 Test2
Test2 SETL {TRUE} ;将该变量赋值为真
GBLS Test3 ;定义一个全局的字符串变量，变量名为 Test3
Test3 SETS “Testing” ;将该变量赋值为“Testing”
```

2. LCLA、LCLL 和 LCLS

语法格式：

LCLA（LCLL 或 LCLS） 局部变量名;

LCLA、LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量，并将其初始化。其中：

LCLA 伪指令用于定义一个局部的数字变量，并初始化为 0；

LCLL 伪指令用于定义一个局部的逻辑变量，并初始化为 F（假）；

LCLS 伪指令用于定义一个局部的字符串变量，并初始化为空；

以上三条伪指令用于声明局部变量，在其作用范围内变量名必须唯一。

例 7.2

```
LCLA Test4 ;声明一个局部的数字变量，变量名为 Test4
Test3 SETA 0xaa ;将该变量赋值为 0xaa
LCLL Test5 ;声明一个局部的逻辑变量，变量名为 Test5
Test4 SETL {TRUE} ;将该变量赋值为真
LCLS Test6 ;定义一个局部的字符串变量，变量名为 Test6
Test6 SETS “Testing” ;将该变量赋值为“Testing”
```

3. SETA、SETL 和 SETS

语法格式：

变量名 SETA（SETL 或 SETS） 表达式;

伪指令 SETA、SETL、SETS 用于给一个已经定义的全局变量或局部变量赋值。

SETA 伪指令用于给一个数学变量赋值；

SETL 伪指令用于给一个逻辑变量赋值；

SETS 伪指令用于给一个字符串变量赋值；

其中，变量名为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。

例 7.3

```
LCLA Test3 ;声明一个局部的数字变量，变量名为 Test3
Test3 SETA 0xaa ;将该变量赋值为 0xaa
LCLL Test4 ;声明一个局部的逻辑变量，变量名为 Test4
Test4 SETL {TRUE} ;将该变量赋值为真
```

4. RLIST

语法格式：

名称　RLIST　{寄存器列表};

RLIST 伪指令可用于对一个通用寄存器列表定义名称，使用该伪指令定义的名称可在 ARM 指令 LDM/STM 中使用。在 LDM/STM 指令中，列表中的寄存器访问次序为根据寄存器的编号由低到高，而与列表中的寄存器排列次序无关。

例 7.4

```
RegList RLIST {R0-R5，R8，R10} ;将寄存器列表名称定义为 RegList，可在 ARM 指令 LDM/STM 中通过该名称访问寄存器列表
```

7.2.2 数据定义（Data Definition ）伪指令

数据定义伪指令一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。常见的数据定义伪指令有如下几种：

- ❒ DCB 用于分配一段连续的字节存储单元并用指定的数据初始化。
- ❒ DCW（DCWU）用于分配一段连续的半字存储单元并用指定的数据初始化。
- ❒ DCD（DCDU）用于分配一段连续的字存储单元并用指定的数据初始化。
- ❒ DCFD（DCFDU）用于为双精度的浮点数分配一段连续的字存储单元并用指定的数据初始化。
- ❒ DCFS（DCFSU）用于为单精度的浮点数分配一段连续的字存储单元并用指定的数据初始化。
- ❒ DCQ（DCQU）用于分配一段以 8 字节为单位的连续的存储单元并用指定的数据初始化。
- ❒ SPACE 用于分配一段连续的存储单元。
- ❒ MAP 用于定义一个结构化的内存表首地址。
- ❒ FIELD 用于定义一个结构化的内存表的数据域。

1. DCB

语法格式：

标号 DCB 表达式;

DCB 伪指令用于分配一段连续的字节（8 位）存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为 0～255 的数字或字符串。DCB 也可用“=”代替。

例 7.5

```
Str DCB “This is a test!”;分配一段连续的字节存储单元并初始化
```

2. DCW（或 DCWU）

语法格式：

标号 DCW（或 DCWU） 表达式;

DCW（或 DCWU）伪指令用于分配一段连续的半字（16 位）存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。

用 DCW 分配的字存储单元是半字对齐的，而用 DCWU 分配的字存储单元并不严格半字对齐。

例 7.6

DataTest DCW 1, 2, 3; 分配 3 个连续的半字存储单元并用 1，2，3 初始化

3. DCD（或 DCDU）

语法格式：

标号 DCD（或 DCDU） 表达式;

DCD（或 DCDU）伪指令用于分配一段连续的字（32 位）存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式，DCD 也可用“&”代替。

用 DCD 分配的字存储单元是字对齐的，而用 DCDU 分配的字存储单元并不严格字对齐。

例 7.7

DataTest DCD 4, 5, 6; 分配 3 个连续的字存储单元并用 4，5，6 初始化

4. DCFD（或 DCFDU）

语法格式：

标号 DCFD（或 DCFDU） 表达式;

DCFD（或 DCFDU）伪指令用于为双精度的浮点数分配一段连续的字存储单元并用伪指令中指定的表达式初始化，每个双精度的浮点数占据两个字单元。

用 DCFD 分配的字存储单元是字对齐的，而用 DCFDU 分配的字存储单元并不严格字对齐。

例 7.8

FDataTest DCFD 0.1, 0.2, 0.3; 分配 3 个连续的双字存储单元并初始化为 0.1，0.2，0.3 的双精度数表达

5. DCFS（或 DCFSU）

语法格式：

标号 DCFS（或 DCFSU） 表达式;

DCFS（或 DCFSU）伪指令用于为单精度的浮点数分配一段连续的字存储单元并用伪指令中指定的表达式初始化，每个单精度的浮点数占据一个字单元。

用 DCFS 分配的字存储单元是字对齐的，而用 DCFSU 分配的字存储单元并不严格字对齐。

例 7.9

FDataTest DCFS -0.1, -0.2, -0.3; 分配 3 个连续的字存储单元并初始化为-0.1，-0.2，-0.3 的单精度数表达

6. DCQ（或 DCQU）

语法格式：

标号 DCQ（或 DCQU）表达式;

DCQ（或 DCQU）伪指令用于分配一段以 8 个字节为单位的连续存储区域并用伪指令中指定的表达式初始化。

用 DCQ 分配的存储单元是字对齐的，而用 DCQU 分配的存储单元并不严格字对齐。

例 7.10

DataTest DCQ 100, 101, 102; 分配 3 个连续的双字内存单元，并用 100D，101D，102D 的 16 进制数据进行初始化

***以上这些数据定义伪指令可以在代码段中直接使用，而不必非要在数据段中**

7. SPACE

语法格式：

标号 SPACE 表达式;

SPACE 伪指令用于分配一片连续的存储区域并初始化为 0。其中，表达式为要分配的字节数。SPACE 也可用“%”代替。

例 7.11

DataSpace SPACE 100 ;分配连续 100 字节的存储单元并初始化为 0

8. MAP 和 FIELD

伪指令 MAP 和 FIELD 经常结合在一起使用。MAP 用于定义一个结构化的内存表的首地址，可以用“ ^ ”代替。

语法格式：

MAP 表达式{，基址寄存器};

表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。

例 7.12

MAP 0x100，R0 ;定义结构化内存表首地址的值为 0x100＋R0。

FIELD 伪指令常与 MAP 伪指令配合使用来定义结构化的内存表。MAP 伪指令定义内存表的首地址，FIELD 伪指令定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的指令引用。

语法格式：

标号 FIELD 表达式;

表达式的值为当前数据域在内存表中所占的字节数。

注意 MAP 和 FIELD 伪指令仅用于定义数据结构，并不实际分配存储单元。

例 7.13

MAP 0x100 ;定义结构化内存表首地址的值为 0x100。

A FIELD 16 ;定义 A 的长度为 16 字节，起始位置为 0x100

B FIELD 32 ;定义 B 的长度为 32 字节，起始位置为 0x110

S FIELD 256 ;定义 S 的长度为 256 字节，起始位置为 0x130

7.2.3 汇编控制（Assembly Control）伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令包括以下几条：

- ❒ IF、ELSE、ENDIF
- ❒ WHILE、WEND
- ❒ MACRO、MEND
- ❒ MEXIT

1. 条件判断指令：IF、ELSE、ENDIF

语法格式：

IF 逻辑表达式
指令序列 1
ELSE
指令序列 2
ENDIF

IF、ELSE、ENDIF 伪指令能根据条件的成立与否决定是否执行某个指令序列，当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则执行指令序列 2。其中，ELSE 及指令序列 2 可以没有，此时，当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则继续执行后面的指令。

IF、ELSE、ENDIF 伪指令可以嵌套使用。

例 7.14

```
GBLL Test ;声明一个全局的逻辑变量，变量名为 Test
……
IF Test = TRUE
指令序列 1
ELSE
指令序列 2
ENDIF
```

2. 循环控制指令：WHILE、WEND

语法格式：

WHILE 逻辑表达式
指令序列
WEND

WHILE、WEND 伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当 WHILE 后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。

WHILE、WEND 伪指令可以嵌套使用。

例 7.15

```
GBLA Counter ;声明一个全局的数学变量，变量名为 Counter
Counter SETA 3 ;由变量 Counter 控制循环次数
……
```

WHILE Counter < 10
指令序列
WEND

3. 宏指令：MACRO、MEND

宏是一段独立的程序代码，它是通过伪指令定义的，在程序中使用宏指令即可调用宏。程序被汇编时，汇编程序将对每个调用进行展开称为宏展开，用宏定义取代源程序中的宏指令。

语法格式：
MACRO
{$label} macroname {$parameter1, $parameter2, ……}
  指令序列
MEND

MACRO 是伪操作标识宏定义的开始，MEND 标识宏定义的结束。用 MACRO 及 MEND 定义一段代码，称为宏定义体，这样在程序中就可以通过宏指令多次调用该代码段。

其中，$label 通常是一个标号，在宏指令被展开时，label 会被替换成相应的符号。在一个符号前使用$表示程序被汇编时将使用相应的值来替代$后的符号。macroname 为所定义的宏的名称。$parameter 为宏指令的参数。当宏指令被展开时将被替换成相应的值，类似于函数中的形式参数，可以在宏定义时为参数指定相应的默认值。包含在 MACRO 和 MEND 之间的指令序列称为宏定义体，在宏定义体的第一行应声明宏的原型（包含宏名、所需的参数），然后就可以在汇编程序中通过宏名来调用该指令序列。

宏指令的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场，从而增加了系统的开销，因此，在代码较短且需要传递的参数较多时，可以使用宏指令代替子程序。

MACRO、MEND 伪指令可以嵌套使用。

例 7.16

```
MACRO ;宏定义开始
$label test $p1, $p2, $p3 ;宏的名称为 test，有三个参数 p1，p2，p3
; 宏的标号$label 可用于构造宏定义体内的其他标号名称
  CMP $p1, $p2 ;比较参数 p1，p2 大小
  BHI $label.save ;无符号比较后若 p1>p2，跳转到$labe.save 标号处，$label.save
;为宏定义体的内部标号
  MOV $p3, $p2
  B $label.end
$label.save MOV $p3, $p1
$label.end; 宏定义功能即将参数 p1，p2 无符号比较后的大值存入参数 p3
MEND; 宏定义结束
```

在上述代码中，宏名为 test，标号为$label，有三个参数$p1，$p2，$p3。标号和参数在实际调用中可根据需要被替换成不同的符号，而宏名是唯一确定的。调用上述宏可以采用下面的方式：

```
abc test R0, R1, R2; 通过宏的名称 test 调用宏，宏的标号为 abc
; 三个参数为寄存器 R0，R1，R2
```

汇编处理宏时会将宏展开还原为一段代码，结果如下：

CMP R0, R1
BHI abc.save
MOV R2, R1
B abc.end
abc.save MOV R2, R0
abc.end
.....

对比宏定义可以看出，原来宏定义中凡是出现$label 的地方均被替换成了 abc，出现$p1、$p2、$p3 的地方被替换成了 R0、R1、R2。对于程序员而言，采用宏定义的方法可以用一条语句替代一大段指令序列，从而极大地提高编程效率。

4. MEXIT

MEXIT 用于从宏定义中跳转出去。

语法格式：

```
MACRO ;宏定义开始
{$标号} 宏名 {$参数 1, $参数 2, ……}
  指令序列 ;
  IF condition1
    MEXIT
  ELSE
    指令序列 ;
  ENDIF
.....
MEND
```

7.2.4 其他常用的伪指令

还有一些其他的伪指令，在汇编程序中经常会被使用，包括以下几条：

- AREA
- ALIGN
- CODE16、CODE32
- ENTRY
- END
- EQU
- EXPORT（或 GLOBAL）
- IMPORT
- EXTERN
- GET（或 INCLUDE）
- INCBIN
- RN
- ROUT

1. AREA

语法格式：

AREA 段名 属性 1，属性 2，……;

AREA 伪指令用于定义一个代码段或数据段，段名若以数字开头，则该段名需用“|”括起来，如|1_test|。

属性字段表示该代码段（或数据段）的相关属性，多个属性用逗号分隔。常用的属性如下：

（1）CODE 属性：用于定义代码段，默认为 READONLY。

（2）DATA 属性：用于定义数据段，默认为 READWRITE。

（3）READONLY 属性：指定本段为只读，代码段默认为 READONLY。

（4）READWRITE 属性：指定本段为可读可写，数据段的默认属性为 READWRITE。

（5）ALIGN 属性：使用方式为 ALIGN 表达式。在默认时，ELF（可执行连接文件）的代码段和数据段是按字对齐的，表达式的取值范围为 0～31。

（6）COMMON 属性：该属性定义一个通用的段，不包含任何的用户代码和数据。各源文件中同名的 COMMON 段共享同一段存储单元。

一个汇编语言程序至少要包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段。

例 7.17

```
AREA RESET, CODE, READONLY
......
;该伪指令定义了一个代码段，段名为 RESET，属性为只读
```

2. ALIGN

语法格式：

ALIGN {表达式{，偏移量}};

ALIGN 伪指令可通过添加填充字节的方式，使当前位置满足一定的对其方式。其中，表达式的值用于指定对齐方式，可能的取值为 2 的幂，如 1、2、4、8、16 等。若未指定表达式，则将当前位置对齐到下一个字的位置。偏移量也为一个数字表达式，若使用该字段，假设 *N*=表达式，则当前位置的对齐方式为：2^N＋偏移量。

例 7.18

```
AREA RESET, CODE, READONLY, ALIEN＝3 ;指定后面的指令为 8 字节对齐。
......
END
```

3. CODE16、CODE32

语法格式：

CODE16（或 CODE32）;

CODE16 伪指令通知编译器，其后的指令序列为 16 位的 Thumb 指令。

CODE32 伪指令通知编译器，其后的指令序列为 32 位的 ARM 指令。

若在汇编源程序中同时包含 ARM 指令和 Thumb 指令时，可用 CODE16 伪指令通知编译器其后的指令序列为 16 位的 Thumb 指令，CODE32 伪指令通知编译器其后的指令序列为 32 位的 ARM 指令。因此，在使用 ARM 指令和 Thumb 指令混合编程的代码里，可用这两

条伪指令进行切换，但注意他们只通知编译器其后指令的类型，并不能对处理器进行状态的切换。

例 7.19

```
AREA RESET, CODE, READONLY
……
CODE32 ;通知编译器其后的指令为 32 位的 ARM 指令
LDR R0, ＝NEXT＋1 ;将跳转地址放入寄存器 R0
BX R0 ;程序跳转到新的位置执行，并将处理器切换到 Thumb 工作状态
……
CODE16 ;通知编译器其后的指令为 16 位的 Thumb 指令
NEXT LDR R3, ＝0x3FF
……
END ;程序结束
```

4. ENTRY

语法格式：

ENTRY;

ENTRY 伪指令用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个 ENTRY（也可以有多个，当有多个 ENTRY 时，程序的真正入口点由链接器指定），但在一个源文件里最多只能有一个 ENTRY（可以没有）。

例 7.20

```
AREA RESET, CODE, READONLY
ENTRY ;指定应用程序的入口点
……
```

5. END

语法格式：

END;

END 伪指令用于通知编译器已经到了源程序的结尾。

例 7.21

```
AREA RESET, CODE, READONLY
……
END ;指定应用程序的结尾
```

6. EQU

语法格式：

名称 EQU 表达式{，类型};

EQU 伪指令用于为程序中的常量、标号等定义一个等效的字符名称，类似于 C 语言中的＃define。其中 EQU 可用“∗”代替。

例 7.22

```
Test EQU 50 ;定义标号 Test 的值为 50
```

7. EXPORT（或 GLOBAL）

语法格式：

EXPORT 标号 {[WEAK]};

EXPORT 伪指令用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。EXPORT 可用 GLOBAL 代替。标号在程序中区分大小写，[WEAK]选项声明其他的同名标号优先于该标号被引用。

例 7.23

```
AREA RESET, CODE, READONLY
EXPORT Stest; 声明一个可全局引用的标号 Stest
……
END
```

8. IMPORT

语法格式：

IMPORT 标号 {[WEAK]};

IMPORT 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，而且无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号被 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

例 7.24

```
AREA    RESET, CODE, READONLY
IMPORT MAIN ;通知编译器当前文件要引用标号 MAIN，但 MAIN 在其他源文件中定义
……
END
```

9. EXTERN

语法格式：

EXTERN 标号{[WEAK]};

EXTERN 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用。如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

例 7.25

```
AREA    RESET, CODE, READONLY
EXTERN Main ;通知编译器当前文件要引用标号 Main，但 Main 在其他源文件中定义
……
END
```

10. GET（或 INCLUDE）

语法格式：

GET 文件名;

GET 伪指令用于将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置进行汇编处理。可以使用 INCLUDE 代替 GET。

汇编程序中常用的方法是在某源文件中定义一些宏指令，用 EQU 定义常量的符号名称，用 MAP 和 FIELD 定义结构化的数据类型，然后用 GET 伪指令将这个源文件包含到其他的源文件中。使用方法与 C 语言中的“INCLUDE”相似。

GET 伪指令只能用于包含源文件，包含目标文件需要使用 INCBIN 伪指令

例 7.26

```
AREA    RESET, CODE, READONLY
GET a1.s ;通知编译器当前源文件包含当前目录下的 a1.s 文件
GET C:\a2.s ;通知编译器当前源文件包含 C 盘根目录下的 a2.s 文件
……
END
```

11. INCBIN

语法格式：

INCBIN 文件名;

INCBIN 伪指令用于将一个目标文件或数据文件包含到当前的源文件中，被包含的文件不作任何变动的存放在当前文件中，编译器不对文件内容进行编译，编译器从其后开始继续处理。

例 7.27

```
AREA    RESET, CODE, READONLY
GET a1.s
INCBIN a1.dat ;通知编译器当前源文件包含文件 a1.dat
INCBIN C:/a2.txt ;通知编译器当前源文件包含 C 盘根目录下文件 a2.txt
……
END
```

12. RN

语法格式：

名称 RN 表达式;

RN 伪指令用于给一个寄存器定义一个别名。采用这种方式可以方便程序员记忆该寄存器的功能。其中，名称为给寄存器定义的别名，表达式为寄存器的编码。

例 7.28

```
Temp RN R0 ;将 R0 定义一个别名 Temp
```

13. ROUT

语法格式：

{名称} ROUT;

ROUT 伪指令用于给一个局部变量定义作用范围。在程序中未使用该伪指令时，局部变量的作用范围为所在的 AREA，而使用 ROUT 后，局部变量的作为范围为当前 ROUT 和下一个 ROUT 之间。

7.2.5 汇编语言中常用的符号

在汇编语言程序设计中，经常使用各种符号代替地址、变量和常量等，以增加程序的可读性。尽管符号的命名由编程者决定，但并不是任意的，必须遵循以下的约定：

- ❒ 符号区分大小写，同名的大、小写符号会被编译器认为是两个不同的符号。
- ❒ 符号在其作用范围内必须唯一。
- ❒ 自定义的符号名不能与系统的保留字相同。
- ❒ 符号名不应与指令或伪指令同名。

1. 程序中的变量

程序中的变量是指其值在程序的运行过程中可以改变的量。ARM（Thumb）汇编程序所支持的变量有数字变量、逻辑变量和字符串变量。

数字变量用于在程序的运行中保存数字值，但注意数字值的大小不应超出数字变量所能表示的范围。

逻辑变量用于在程序的运行中保存逻辑值，逻辑值只有两种取值情况：真或假。

字符串变量用于在程序的运行中保存一个字符串，但注意字符串的长度不应超出字符串变量所能表示的范围。

在 ARM（Thumb）汇编语言程序设计中，如 7. 2. 1 所述可以使用 GBLA、GBLL、GBLS 伪指令声明全局变量，使用 LCLA、LCLL、LCLS 伪指令声明局部变量，并可使用 SETA、SETL 和 SETS 对其进行初始化。

例 7.29

```
LCLS S1 ;
LCLS S2 ;定义局部字符串变量 S1 和 S2
S1 SETS “Test!” ;字符串变量 S1 的值为“Test!”
S2 SETS “Hello!” ;字符串变量 S2 的值为“Hello!”
```

2. 程序中的常量

程序中的常量是指其值在程序的运行过程中不能被改变的量。ARM（Thumb）汇编程序所支持的常量有数字常量、逻辑常量和字符串常量。

数字常量一般为 32 位的整数，当作为无符号数时，其取值范围为$0 \sim 2^{32}-1$，当作为有符号数时，其取值范围为$-2^{31} \sim 2^{31}-1$。

逻辑常量只有两种取值情况：真或假。

字符串常量为一个固定的字符串，一般用于程序运行时的信息提示。

例 7.30

```
NUM EQU 64
abcd EQU 2   ;定义 abcd 符号的值为 2
abcd EQU label+16 ;定义 abcd 符号的值为（label+16）
abcd EQU 0x1c，CODE32 ;定义 abcd 符号的值为绝对地址值 0x1c，而且此处为 ARM32 指令
```

ARM 汇编指令 EQU 与等号“=”的不同

不同点：

1、使用 EQU 伪指令定义的符号名不能与其它符号名重名，符号名必须唯一，且不能被重新定义;而使用等号伪指令″=″定义的符号名可以重名，可以被重新定义，可被重新赋值。

2、使用 EQU 伪指令定义的符号名不仅可以代表某个常数或常数表达式，还可以代表字符串、关键字、指令码、一串符号（如:WORD，PTR），等等;而使用等号伪指令″=″定义的符号名仅仅用于代表数值表达式

3. 程序中的变量代换

程序中的变量可通过代换操作取得一个常量。代换操作符为“$”。如果在数字变量前面有一个代换操作符“$”，编译器会将该数字变量的值转换为十六进制的字符串，并将该十六进制的字符串代换“$”后的数字变量。

如果在逻辑变量前面有一个代换操作符“$”，编译器会将该逻辑变量代换为它的取值（真或假）。如果在字符串变量前面有一个代换操作符“$”，编译器会将该字符串变量的值代换“$”后的字符串变量。

例 7.31

LCLS S1; 定义局部字符串变量 S1 和 S2

LCLS S2

S1 SETS “Test!”

S2 SETS “This is a $S1”; 字符串变量 S2 的值为“This is a Test！”

7.2.6 汇编语言中常用运算符和表达式

在汇编语言程序设计中，也经常使用各种表达式，表达式一般由变量、常量、运算符和括号构成。常用的表达式有数字表达式、逻辑表达式和字符串表达式，其运算次序遵循如下的优先级：

- ❒ 优先级相同的双目运算符的运算顺序为从左到右。
- ❒ 相邻的单目运算符的运算顺序为从右到左，且单目运算符的优先级高于其他运算符。
- ❒ 括号运算符的优先级最高。

1. 数字表达式及运算符

数字表达式一般由数字常量、数字变量、数字运算符和括号构成。与数字表达式相关的运算符如下：

（1）“＋”、“－”、“*”、“/” 及“MOD”算术运算符

以上的算术运算符分别代表加、减、乘、除和取余数运算。例如，以 X 和 Y 表示两个数字表达式，则：

X＋Y 表示 X 与 Y 的和。

X－Y 表示 X 与 Y 的差。

X*Y 表示 X 与 Y 的乘积。

X/Y 表示 X 除以 Y 的商。

X:MOD:Y 表示 X 除以 Y 的余数。

例 7.32

MOV R3, #(3+4); R3=0x07

MOV R3, #(5*10); R3=0x32

（2）“ROL”、“ROR”、“SHL”及“SHR”移位运算符

以 X 和 Y 表示两个数字表达式，以上的移位运算符代表的运算如下：

X:ROL:Y 表示将 X 循环左移 Y 位。

X:ROR:Y 表示将 X 循环右移 Y 位。

X:SHL:Y 表示将 X 左移 Y 位。

X:SHR:Y 表示将 X 右移 Y 位。

例 7.33

```
MOV R1, #(3:ROL:1); R1=0x06
MOV R2, #(2:ROR:1); R2=0x01
MOV R3, #(9:SHL:1); R3=0x12
```

（3）“AND”、“OR”、“NOT”及“EOR”按位逻辑运算符

以 X 和 Y 表示两个数字表达式，以上的按位逻辑运算符代表的运算如下：

X:AND:Y 表示将 X 和 Y 按位作逻辑与的操作。

X:OR:Y 表示将 X 和 Y 按位作逻辑或的操作。

:NOT:Y 表示将 Y 按位作逻辑非的操作。

X:EOR:Y 表示将 X 和 Y 按位作逻辑异或的操作。

例 7.34

```
MOV R0, #(:NOT:3); R0=0xFFFFFFFC
MOV R1, #(3:AND:1); R1=0x01
MOV R2, #(2:OR:4); R2=0x06
MOV R3, #(9:EOR:1); R3=0x08
```

2. 逻辑表达式及运算符

逻辑表达式一般由逻辑量、逻辑运算符和括号构成，其表达式的运算结果为真或假。与逻辑表达式相关的运算符如下：

（1）“=”、“>”、“<”、“>=”、“<= ”、“/=”、“ <>” 运算符

以 X 和 Y 表示两个逻辑表达式，以上的运算符代表的运算如下：

X = Y 表示 X 等于 Y。

X > Y 表示 X 大于 Y。

X < Y 表示 X 小于 Y。

X >= Y 表示 X 大于等于 Y。

X <= Y 表示 X 小于等于 Y。

X /= Y 表示 X 不等于 Y。

X <> Y 表示 X 不等于 Y。

（2）“LAND”、“LOR”、“LNOT”及“LEOR”运算符

以 X 和 Y 表示两个逻辑表达式，以上的逻辑运算符代表的运算如下：

X:LAND:Y 表示将 X 和 Y 作逻辑与的操作。

X:LOR:Y 表示将 X 和 Y 作逻辑或的操作。

:LNOT:Y 表示将 Y 作逻辑非的操作。

X:LEOR:Y 表示将 X 和 Y 作逻辑异或的操作。

3. 字符串表达式及运算符

字符串表达式一般由字符串常量、字符串变量、运算符和括号构成。编译器所支持的字符串最大长度为 512 字节，常用的与字符串表达式相关的运算符如下：

（1）LEN 运算符

LEN 运算符返回字符串的长度（字符数），以 X 表示字符串表达式，其语法格式如下：

:LEN:X

（2）CHR 运算符

CHR 运算符将 0～255 之间的整数转换为一个字符，以 M 表示某一个整数，其语法格式如下：

:CHR:M

（3）STR 运算符

STR 运算符将将一个数字表达式或逻辑表达式转换为一个字符串。对于数字表达式，STR 运算符将其转换为一个以十六进制组成的字符串；对于逻辑表达式，STR 运算符将其转换为字符串 T 或 F，其语法格式如下：

:STR:X

其中，X 为一个数字表达式或逻辑表达式。

（4）LEFT 运算符

LEFT 运算符返回某个字符串左端的一个子串，其语法格式如下：

X:LEFT:Y

其中：X 为源字符串，Y 为一个整数，表示要返回的字符个数。

（5）RIGHT 运算符

与 LEFT 运算符相对应，RIGHT 运算符返回某个字符串右端的一个子串，其语法格式如下：

X:RIGHT:Y

其中：X 为源字符串，Y 为一个整数，表示要返回的字符个数。

（6）CC 运算符

CC 运算符用于将两个字符串连接成一个字符串，其语法格式如下：

X:CC:Y

其中：X 为源字符串 1，Y 为源字符串 2，CC 运算符将 Y 连接到 X 的后面。

4. 与寄存器和程序计数器（PC）相关的表达式及运算符

常用的与寄存器和程序计数器（PC）相关的表达式及运算符如下：

（1）BASE 运算符

BASE 运算符返回基于寄存器的表达式中寄存器的编号，其语法格式如下：

:BASE:X

其中，X 为与寄存器相关的表达式。

（2）INDEX 运算符

INDEX 运算符返回基于寄存器的表达式中相对于其基址寄存器的偏移量，其语法格式如下：

:INDEX:X

其中，X 为与寄存器相关的表达式。

5. 其他常用运算符

（1）？运算符

？运算符返回某代码行所生成的可执行代码的长度，其语法格式如下：

?X

返回定义符号 X 的代码行所生成的可执行代码的字节数。

例 7.35

```
LA MOV R2, #1
MOV R3, ?LA; R3=0x04
```

（2）DEF 运算符

DEF 运算符判断是否定义某个符号，其语法格式如下：

:DEF:X

如果符号 X 已经定义，则结果为真，否则为假。

7.3 ARM 汇编语言程序设计

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，但是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 程序在完成上述功能时通常是采用 C 语言和汇编语言混合编程。

7.3.1 ARM 汇编语言的语句格式

ARM 汇编中语句中所有标号必须在一行的最左边顶格书写，标号后面不能添加“:”，而所有指令均不能顶格书写。ARM 汇编器对标识符大小写敏感，书写标号及指令时字母大小写要一致。在 ARM 汇编程序中，一条 ARM 指令、伪指令、寄存器名可以全部为大写字母，也可以全部为小写字母，但不要大小写混合使用。注释使用“;”，注释内容由“;”开始到此行结束，注释可以在一行的顶格书写。ARM 汇编语句的格式如下：

```
[LABEL]   OPERATION    [OPERAND] [;COMMENT]
标号域    操作助记符域    操作数域       注释域
```

1. 标号域（LABLE）

（1）标号域用来表示指令的地址、变量、过程名、数据的地址和常量。

（2）标号是可以自己起名的标识符，语句标号可以是大小写字母混合，通常以字母开头，由字母、数字、下划线等组成。

（3）语句标号不能与寄存器名、指令助记符、伪指令（操作）助记符、变量名同名。

（4）**ARM 汇编规范规定：语句标号必须在一行的开头书写，不能留空格；指令则在行开头必须要留出空格。**

2. 操作助记符域（OPERATION）

（1）操作助记符域可以为指令、伪操作、宏指令或伪指令的助记符。

（2）ARM 汇编器对大小写敏感，在汇编语言程序设计中，每一条指令的助记符可以全部用大写、或全部用小写，但不允许在一条指令中大、小写混用。

（3）所有的指令都不能在行的开头书写，必须在指令的前面有空格，然后再书写指令。

（4）指令助记符和后面的操作数或操作寄存器之间必须有空格，不可以在这之间使用逗号。

3. 操作数域（OPERAND）

操作数域表示操作的对象，操作数可以是常量、变量、标号、寄存器名或表达式，不同对象之间必须用逗号“,”分开。

7.3.2 ARM 汇编语言程序结构

在 ARM（Thumb）汇编语言程序中，通常以段为单位来组织代码段是具有特定名称且功能相对独立的指令或数据序列根据段的内容，分为代码段和数据段，一个汇编程序至少应该有一个代码段，当程序较长时，可以分割为多个代码段和数据段。

一个汇编语言程序段的基本结构如下所示：

```
AREA Buf，DATA，READWRITE;定义一个可读写属性的数据段
Num     DCD     0x11
Nums    DCD     0x22    ;分配一片连续字存储单元并初始化
;本节及下节所有汇编代码程序都是利用 MDK IDE 编写，Device 设置为 STM32F407ZG，代码段放在 IROM 中起始地址为 0x08000000，数据段放在 IRAM 中起始地址为 0x20000000。
AREA RESET，CODE，READONLY;只读的代码段，RESET 为段名
ENTRY                           ;程序入口点
START    LDR R0，= Num ;    取 Num 地址赋给 R0
LDR R1，[R0]  ;    取 Num 中内容赋给 R1
ADD R1，#0x9A ;    R1=R1+0x9A
STR R1，[R0]  ;    R1 内容赋给 Num 单元
LDR R0，= Nums;    .....
LDR R2，[R0]
ADD R2，#0xAB
STR R2，[R0]
LOOP    B    LOOP    ;       无限循环反复执行
END                     ;       段结束
```

程序在调试模式下运行后，寄存器及相关内存单元内容如图 7.4 所示。

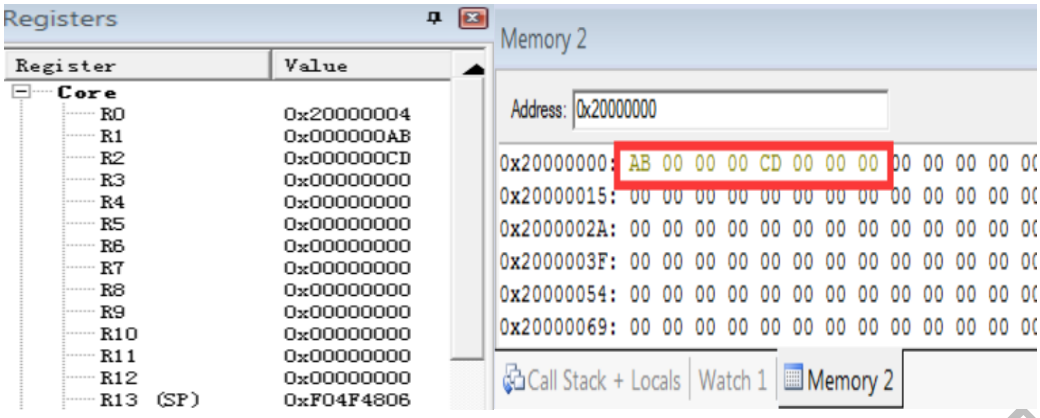

图 7.4 示例程序运行后寄存器和相关内存单元中内容

从以上范例中可以了解 ARM 汇编语言程序的基本结构，整个结构中除了程序的主体要使用 ARM 指令完成以外，在其他部分会大量使用伪指令。在汇编语言程序的开头 GET 等伪指令声明当前源文件需要引用源文件，被引用的源文件在当前位置进行汇日伪指令 AREA 定义段，并说明所定义段的相关属性。本例定义了两个段，先定义名为 Buf 的数据段，属性为可读写（数据段也可以不用单独的段结构而直接在代码段内定义）后又定义代码段，属性为只读。伪指令 ENTRY 标识了程序的入口，即该程序段被执行的第一条指令，接下来为程序主体。程序的末尾为伪指 END 告诉编译器源文件已经结束，每一个汇编程序段都必须有一条伪指令 END 指示代码段的结束。

7.3.3 ARM 汇编程序设计实例

ARM 汇编程序的结构主要分为以下三种：顺序结构、分支结构和循环结构，本节将通过实例来介绍这 3 种结构以及如何使用子程序调用。通过学习这些实例，可帮助读者掌握 ARM 汇编程序设计的基本方法，为设计更复杂的 ARM 程序奠定基础。

1. 顺序结构

顺序结构是一种最简单的程序结构，这种程序按指令排列的先后顺序逐条执行。

例 7.36 对数据段中数据进行寻址操作

```
        AREA    BUF, DATA, READWRITE    ; 定义数据段 Buf
ARR     DCB     0x11, 0x22, 0x33, 0x44
        DCB 0x55, 0x66, 0x77, 0x88
        DCB 0x00, 0x00, 0x00, 0x00          ; 定义 12 个字节的数组 Array

        AREA    RESET, CODE, READONLY
        ENTRY
        LDR R0, =ARR            ; 取得数组 Array 的首地址
        LDR R2, [R0]            ; 从数组第 1 字节取 32 位数据给 R2  即 R2=0x44332211
        MOV R1, #1                  ; R1=1
        LDR R3, [R0, R1, LSL#2]     ;将存储器地址为 R0＋R1×4 的 32 位数据读入 R3，
                                    ; R3=0x88776655
LOOP    B LOOP
        END
```

例 7.37 64 位数据的求和计算

ARM 是 32 位的处理器，一次只能完成两个 32 位数据的之间的运算。要实现 64 位数据的求和，关键在于先计算低 32 位的结果，然后利用带进位的加法运算求得高 32 位的求和结果，即得到 64 位的运算结果。

```
        AREA    BUF, DATA, READWRITE;  定义数据段 Buf
ARR1    DCD    0x11223344，0xFFDDCCBB
ARR2    DCD    0x11223344，0xFFDDCCBB
RESULT DCD    0，0
        ;数据段中数据存入 IRAM 中，在本节所有程序的通用设置中起始地址
        0x20000000

        AREA    RESET, CODE, READONLY
        ENTRY
        LDR R0，=ARR1     ；取得数组 Arr1 的首地址存入 R0
        LDR R1，[R0]          ；读数据 1 的高 32 位到 R1
        LDR R2，[R0，#4]   ; R2=[R0+4]读数据 1 的低 32 位到 R2
        LDR R0，=ARR2     ；取得 ARR2 的首地址存入 R0
        LDR R3，[R0]          ；读数据 2 的高 32 位到 R3
        LDR R4，[R0，#4]   ；读数据 2 的低 32 位到 R4
        ADDS R6，R2，R4   ；低 32 位相加，影响标志位，保存进位，结果放入 R6
        ADC R5，R1，R3     ；带进位的高 32 位相加，结果放入 R5
        LDR R0，=Result       ; R0 中保存 result 的地址
        STR R5，[R0]           ; R5 内容存入[R0]保存结果高位到 result
        STR R6，[R0，#4]    ；保存结果的低位
LOOP    B LOOP
        END
```

2. 分支结构

一般情况下，程序按指令的先后顺序逐条执行，但经常要求程序根据不同条件选择不同的处理方法，也就是说程序的处理步骤中出现了分支，就要根据某一特定条件选择其中一个分支执行，执行到分支点时，利用条件指令或条件转移指令根据当前 CPSR 中的状态标志值选择路径，使用带有条件码的指令实现的分支程序段。

当处理器工作在 ARM 状态时，几乎所有的指令均根据 CPSR 中条件码的状态和指令的条件域有条件的执行。当指令的执行条件满足时，指令被执行，否则指令被忽略。每一条 ARM 指令包含 4 位的条件码，位于指令的最高 4 位[31:28]。条件码共有 16 种，每种条件码可用两个字符表示，这两个字符可以添加在指令助记符的后面和指令同时使用。例如，跳转指令 B 可以加上后缀 EQ 变为 BEQ 表示“相等则跳转”，即当 CPSR 中的 Z 标志置位时发生跳转。在 16 种条件标志码中，只有 15 种可以使用，如表 6.6 所示。

分支结构中经常使用的条件码助记后缀：HI 无符号数大于，LS 无符号数小于或等于；

GT 带符号数大于，LE 带符号数小于或等于。例如寄存器中 R0 和 R1 分别保存两个数，如果 R0 小于 R1，将 R1 值传给 R0，实现代码如下：

CMP R0，R1; 比较 R0 和 R1

MOVLT R0，R1; MOVLT = MOV+ LT，查上表知为 LT 为带符号数小于，也就是说在 R0< R1 时，才执行 MOV 操作。如何查看条件是否成立呢？CMP R0，R1 执行后，CPU 就查看 CPSR 中 N 是否等于 V，不等于说明 R0< R1 成立，执行 MOV 操作。

分支结构的几种实现方法：

（1）利用条件码可以很方便地实现 IF ELSE 分支结构的程序

假设利用 C 语言实现如下分支结构：

```
int x=76;
int y=88;
if(x>y)  z=100;
else z=50;
```

ARM 汇编语言实现同样分支结构代码片段：

```
MOV R0，#76 ; 初始化 R0 的值
MOV R1，#88 ; 初始化 R1 的值
CMP R0，R1 ; 判断 R0>R1？
MOVHI R2，#100 ; R0>R1 时，R2=100
MOVLS R1，#50 ; R0<R1 时，R2=50
......
```

（2）B（Branch）条件转移及衍生指令实现分支结构

1）B 指令

格式：B{条件} + 目标地址

用法：B 指令是最简单的跳转指令，一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的目标地址，从那里继续执行。

例 7.38

```
B  Label ; 程序无条件跳转到标号 Label 处执行
```

例 7.39

```
CMP R1, #0
BEQ Label ; 当 CPSR 寄存器中 Z 条件码置位时，程序跳转到标号 Label 处执行
```

例 7.40

```
B 0x1234 ; 程序跳转到绝对地址 0x1234 处执行
```

例 7.41 R5 中值等于 10 则将 R5 赋给 R1，否则赋给 R0

```
CMP    R5，#10
BEQ    Doequal
MOV    R0，R5
B ENDIF
Doequal  MOV    R1，R5
ENDIF  .....
```

2）BL 指令

格式：BL{条件} + 目标地址

用法：BL 是一个跳转指令，但跳转之前，会在寄存器 R14 中保存 PC 当前的内容。因此，可以通过将 R14 的内容重新加载到 PC 中，来返回到跳转指令之后的那个指令处执行。这个指令是实现子函数调用的一个基本且常用的手段。

BL Label；让程序无条件跳转到标号是 Label 处执行，同时将当前的 PC 值保存到 R14（LR）中。

BL 实现子函数调用时完成如下的三个操作：

（i）将子程序的返回地址（当前 PC）保存在 R14（LR）中。

（ii）将 PC 指向子程序的入口即跳转（也即 BL 后面的目标指令）。

（iii）子程序执行完毕之后需要返回时，只需将 R14 中的 LR 赋给 PC。

使用 BL 调用子程序后，通常在子程序的尾部添加 MOV PC，LR 来返回。

3）BX 指令

带状态切换的跳转指令

指令的语法格式：BX　<Rm>（只能是寄存器）

BX R0; 跳转到 R0 寄存器中存储的地址，如果 R0[0]=1，则进入 Thumb 状态。

带状态切换的跳转指令 BX 使程序跳转到指令中指定的参数 Rm 指定的地址执行程序。

若 Rm 的 bit[0] 为 1，切换到 Thumb 指令执行；

若 Rm 的 bit[0] 为 0，切换到 ARM 指令执行。

示例：带 ARM/Thumb 状态切换的分支程序

从 ARM 指令程序段跳转到 Thumb 指令程序

ARM 指令程序

```
CODE32 ; 以下为 ARM 指令程序段
...
ADR R0, Into_Thumb + 1 ; 目标地址和切换状态传送至 R0
BX R0 ; 跳转
...        ; 其他代码

;Thumb 指令程序
CODE16              ;以下为 Thumb 指令程序段
Into_Thumb   ...    ;目标从 Thumb 指令程序段跳转到 ARM 指令程序
;Thumb 指令程序
CODE16 ; 以下为 Thumb 指令程序段
...
ADR R5，Back_to_ARM ; 目标地址送至 R5
BX R5 ; 跳转
... ; 其他代码
; ARM 指令程序
CODE32 ; 以下为 ARM 指令程序段
Back_to_ARM   ... ; 目标
```

例 7.42　查找两个数中的最大值，假设两个分别在数据段的 Arr1 和 Arr2 中，要求将其中大值存入数据段的 Result 单元。

```
AREA   RESET, CODE, READONLY
ENTRY
LDR R0, Arr1
LDR R1, Arr2
```

```
    CMP R0, R1
    BHI SAVE ; 无符号数比较，大于则跳转
    MOV R0, R1 ; 无符号比较小于
SAVE STR R0, Result ; 将 R0 中较大数存入 Result
    END
```

例 7.43 已知 32 位有符号数 X 存放在数据段 Arr 单元中，要求实现：Y=1（X>0）或 Y=0（X=0）或 Y=−1（X<0）。

```
    AREA    RESET, CODE, READONLY
    ENTRY
    LDR  R1, =Arr
    LDR  R2, [R1]
    CMP  R2, #0
    BEQ  ZERO
    BGT  PLUS
    MOV R0, #-1
    B      FINISH
PLUS    MOV R0, #1
    B      FINISH
ZERO    MOV R0, #0
FINISH  STR R0, [R1]
    END
```

3. 循环结构

循环结构可以减少源程序重复书写的工作量，用来描述重复执行某段算法的问题，这是程序设计中最能发挥计算机特长的程序结构。循环结构有 3 个要素：循环变量、循环体和循环终止条件。在高级语言中有 for 和 while 等不同的语句来设置循环条件，而在汇编语言中循环结构则主要依靠比较指令和带条件的跳转指令来实现。

（1）for 循环结构实现

假设利用 C 语言实现如下循环结构：

```
for(i = 0;  i < 10;  i++)
x++;
```

ARM 汇编语言实现同样循环结构代码片段：R0 为 x，R2 为 i，均为无符号整数

```
    MOV R0, #0 ; 初始化 R0=0
    MOV R2, #0 ; 初始化 R2=0
    LOOP CMP R2, #10 ; 判断 R2<10?
    BCS FOR_E          ; 若条件失败（即 R>=10），退出循环
    ADD R0, R0, #1     ; 执行循环体，R0=R0+1，即 x++ ，ADD R0, #1 也可
    ADD R2, R2, #1     ; R2=R2+1，即 i++
    B LOOP
    FOR_E .....
```

（2）While 循环结构实现

假设利用 C 语言实现如下循环结构：

```
while(x<=y)
x*=2;
```

ARM 汇编代码片段实现：x 为 R0，y 为 R1，均为无符号整数

```
    MOV R0, #1     ; 初始化 R0=1
    MOV R1, #20    ; 初始化 R1=20
W1  CMP R0, R1     ; 判断 R0<=R1，即 x<=y
    MOVLS R0, R0, LSL#1 ; 循环体，R0*=2
    BLS W1                ; 若 R0<=R1，继续循环体
    W_END .....
```

例 7.44 实现 1+2+3+...+N 的累加求和。

```
    AREA RESET, CODE, READONLY
    ENTRY
    MOV R0, #100 ; 循环数，即累加个数 N
    MOV R1, #0   ; 初始化数据
LOOP ADD R1, R1, R0 ; 将数据进行相加，获得最后的数据
    SUBS R0, R0, #1   ; 循环数据 R0 减去 1
    CMP R0, #0        ; 将 R0 与 0 比较看循环是否结束
    BNE LOOP          ; 判断循环是否结束，接受则进行下面的步骤
    LDR R2, =RESULT; RESULT 为数据段存储结果单元，将 RESULT 地址赋给 R2
    STR R1, [R2]
    END
```

有些循环结构比较复杂，需要用多重循环完成。多重循环设计方法与单重相同，但应注意:

（1）各重循环的初始控制条件及程序实现。

（2）内循环可以嵌套在外循环中，也可以多层嵌套，但各层循环之间不能交叉，内外循环之间的跳转要注意循环控制条件的变化。

（3）防止出现死循环，即不能让循环回到初始条件，引起死循环。

例 7.45 有一个无符号数组共有 5 个元素：12，7，19，8，24，它们存放在 SRC 开始的数据单元中，要求编程将数组中的数按从大到小的次序排列（元素个数 n=5）。

采用冒泡排序算法，从第一个数据开始，相邻的数进行比较，如果大数在前，则不做操作，若次序不对，两数交换位置。第一遍比较（N-1）次后，最大的数已到了数组尾，第二遍仅需比较（N-2）次，共比较（N-1）遍就完成了排序，这样共有两重循环。比较过程中数的排列如下所示。

原始数据	12	7	19	8	24	
第一轮比较后	12	19	8	24	7	找出最小值 7
第二轮比较后	19	12	24	8	7	找出第二小值 8
第三轮比较后	19	24	12	8	7	找出第三小值 12
第四轮比较后	24	19	12	8	7	已排好次序，外循环次数为 n-1=4

```
    AREA Array, DATA, READWRITE
    SRC DCD 12,7,19,8,24 ; 初始化待排序数组
```

```
    LEN EQU 5*4
    ; 将数据定义在 IRAM 的数据段中
    AREA RESET, CODE, READONLY
    ENTRY
    MOV R4, #0
    LDR R6, =SRC      ; 设置 R6 保存待排序数组首地址
    ADD R6, R6, #LEN ; 让 R6 保存数组中最后一个地址
OUTER   LDR R1, =SRC; 外循环起始
INNER   LDR R2, [R1]   ; 内循环起始
    LDR R3, [R1, #4]
    CMP R2, R3
    STRHI R3, [R1]
    STRHI R2, [R1, #4]
    ADD   R1, R1, #4
    CMP R1, R6
    BCC INNER          ; 内循环结束
    ADD R4, R4, #4
    CMP R4, #LEN
    SUBLS R6, R6, #4   ; 如果 R4<LEN，R6=R6-4
    BLS OUTER          ; 外循环结束
LOOP B LOOP
    END
```

如代码所示，冒泡算法共有内外两重循环，在例 7.45 中，内环的起点从标号 INNER 到指令 BLT　INNER 结束。循环控制变量存储在 R1 中，当 R1 的值小于 R6 的值即小于待排序最后一个数的地址时一直循环。外循环则从标号 OUTER 开始，到指令 BLS OUTER 结束。循环控制变量为 R4，当外循环次数少于数组大小时一直循环。

4.子程序调用与返回

在 ARM 汇编语言中，子程序的调用一般是通过 BL 指令来完成的。BL 指令的语法格式如下：

```
BL    SUB
```

SUB 是被调用的子程序的名称。

BL 指令完成 2 个操作，即将子程序的返回地址放在 LR 寄存器中，同时将 PC 寄存器指向子程序的入口点，当子程序执行完毕需要返回主程序时，只需将存放在 LR 中的返回地址重新赋给指令指针寄存器 PC 即可。通过调用子程序，能够完成参数的传递和从子程序返回运算的结果（通常使用寄存器 R0-R3 来完成）。

BL 调用子程序的经典用法如下：

```
    BL   NEXT ; 跳转到 NEXT
     ……
  NEXT
     ……
    MOV    PC，LR ; 从子程序返回。
```

当子程序需要使用的寄存器与主程序使用的寄存器发生冲突（即子程序与主程序都要使用同一组寄存器时），为了防止主程序这些寄存器中的有用数据丢失，在子程序的开始应该把这些寄存器数据压入堆栈以保护现场，在子程序返回之前需要把保护到堆栈的数据自堆栈中弹出以恢复现场。

保存和恢复数据可以用 PUSH/POP 指令实现，PUSH {R4，LR}将寄存器 R4 入栈，LR 也入栈。POP {R4，PC}将堆栈中的数据弹出到寄存器 R4 及 PC 中。

如果需要保存数据较多即需要入栈和出栈多个寄存器时，可以用 PUSH {R0-R7，LR}将寄存器 R0-R7 全部入栈，LR 也入栈；POP {R0-R7，PC} 将堆栈中的数据弹出到寄存器 R0-R7 及 PC 中。

ARM 汇编指令中还有与 PUSH/POP 功能类似的压栈和出栈指令：STMFD SP!, {R0-R7, LR} ，功能是满递减入栈，将寄存器 R0-R7、LR 压栈，SP 不断减 4，执行后 SP=SP-9*4 [SP]=R0，[SP+4]=R1，....，[SP+4*8]=LR。

假设初始 SP=0x40000460，操作如图 7.5 所示：

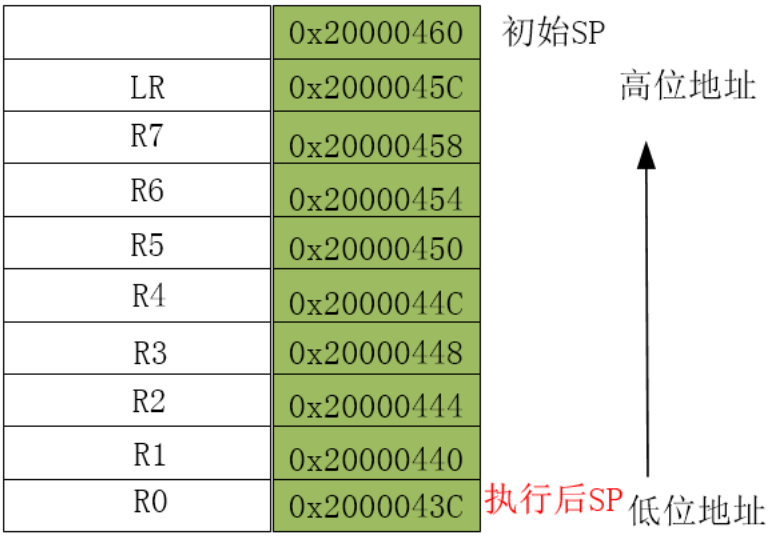

图 7.5 STMFD 入栈操作示意图

LDMFD SP!, {R0-R7, PC}，满递减出栈，给寄存器 R0-R7 出栈，并使程序跳转回函数的调用点，SP 不断增 4；同理，LDMFD 是 STMFD 的逆操作，[SP]->R0，[SP+4]->R1，...，[SP+4*8]->PC　SP=SP+4*9；操作如图 7.6 所示

图 7.6 LDMFD 出栈操作示意图

另外有一点需要注意：BL　SUB ... MOV PC，LR 这种调用子程序结构在一些情况下是会出现问题的，例如当出现子程序嵌套调用时，LR 寄存器中内容在第二次子程序调用时会被覆盖，如果仍使用常用调用方式会出现无法返回现场的问题，这个时候使用 STMFD/LDMFD 指令可以有效避免 LR 被覆盖而出现错误。

例 7.46 子程序嵌套调用时，STMFD/LDMFD 指令的用法

```
AREA   RESET，CODE，READONLY
ENTRY
START      LDR     SP，=0x20000460
           MOV     R0, #0x03
           MOV     R1, #0x04
           MOV     R7, #0x07
           BL   POW
LOOP       B    LOOP
POW        STMFD   SP!，{R0-R7，LR}；指令执行前、后寄存器状态如图 7.7 和 7.8
；所示
           MOVS     R2, R1
           MOVEQ    R0, #1
           BEQ      POW_END
           MOV      R1, R0
           SUB      R2, R2, #1
POW_L1     BL       DO_MUL          ；子程序调用嵌套，这个时候如果不用 STMFD/
；LDMFD 指令会出现 LR 覆盖，影响子程序调用的正常返回
           SUBS     R2, R2, #1
           BNE      POW_L1
POW_END          LDMFD    SP!, {R0-R7，PC}
DO_MUL           MUL      R0, R1, R0
           MOV      PC, LR
  END
```

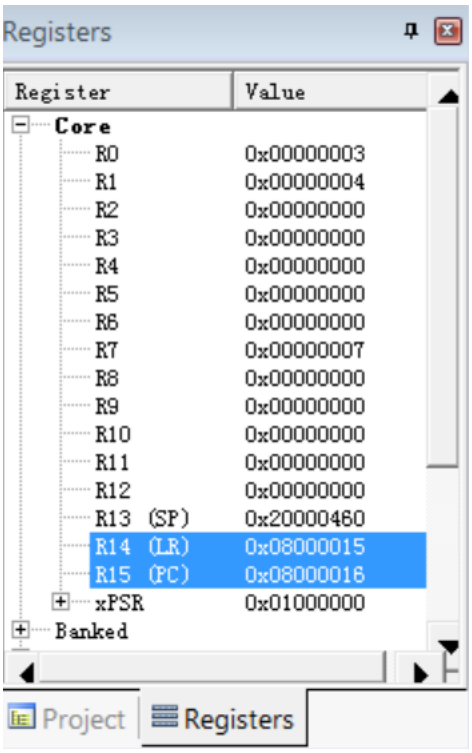

图 7.7 例 7.46 中 STMFD 执行前寄存器状态

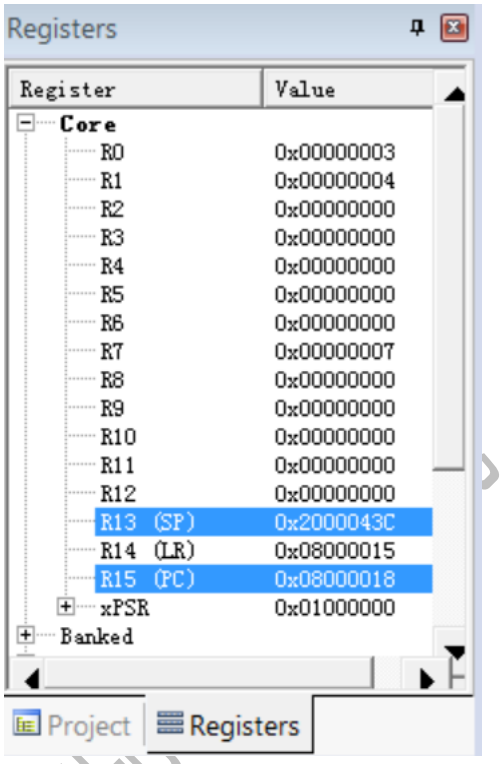

图 7.8 例 7.46 中 STMFD 执行后寄存器状态

例 7.46 中指令 STMFD SP!，{R0-R7，LR}执行后堆栈状态如图 7.9 所示，可知 LR 寄存器最先入栈，R0 寄存器最后入栈。

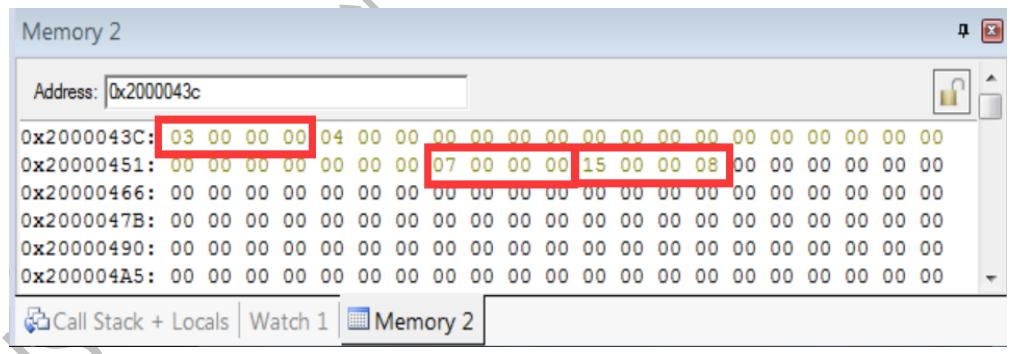

图 7.9 例 7.46 中指令 STMFD SP!，{R0-R7，LR}执行后堆栈状态

例 7.46 中指令 LDMFD SP!, {R0-R7，PC}执行前后寄存器状态如图 7.10 和图 7.11 所示，即将堆栈内的 LR 内容弹回给 PC 恢复执行。

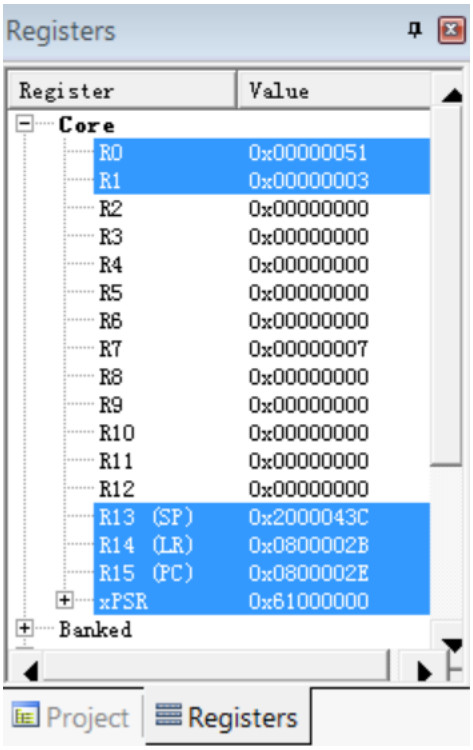

图 7.10 例 7.46 中 LDMFD 执行前寄存器状态

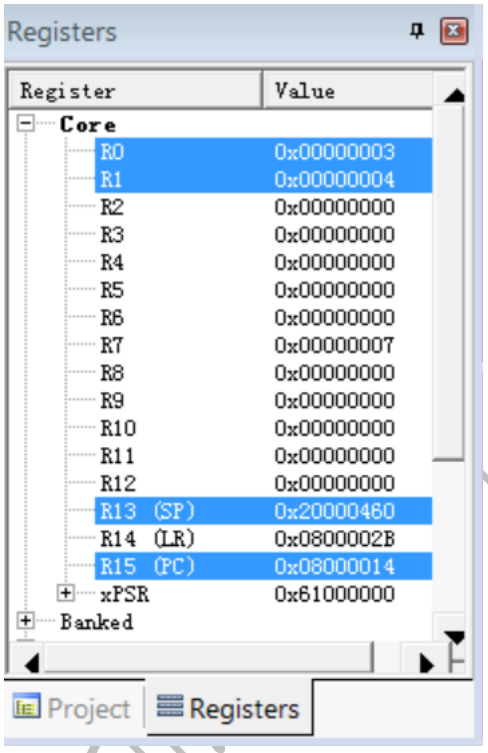

图 7.11 例 7.46 中 LDMFD 执行后寄存器状态

例 7.47 编写完整汇编程序，利用子程序结构进行数据大小判断

```
X EQU 19；定义 X 的值为 19
N EQU 20；定义 N 的值为 20
AREA RESET，CODE，READONLY；声明代码段
ENTRY                          ；标识程序入口
START  LDR R0，=X              ；给 R0、R1 赋初值
LDR R1 ，=N
BL MAX                         ；调用子程序 MAX
HALT B HALT
MAX；声明子程序 MAX
CMP R0，R1
MOVHI R2，R0                   ；比较 R0，R1，R2 等于最大值
MOVLS R2，R1
MOV PC，LR                     ；返回语句
END
```

7.4 ARM 汇编语言与 C/C++的混合编程

嵌入式软件开发过程中，通常会使用包括 ARM 汇编语言和 C/C++语言在内的多种语言。一般情况下，一个 ARM 工程（ Project）应该由多个文件组成，其中有可能包括扩展名为.s 的汇编语言源文件、扩展名为.c 的 C 语言源文件、扩展名为.cpp 的 C++源文件，以及扩展名

为.h 的头文件等。

通常程序会使用汇编完成处理器启动阶段的初始化等工作，有些对处理器运行效率较高的底层算法也会采用汇编语言进行编写和手工优化，而在开发主程序时一般会采用 C/C++语言，因此嵌入式软件开发人员必须掌握 ARM 汇编语言与 C/C++语言混合编程技能。

7.4.1 C 语言与汇编语言之间的函数调用

为了使单独编译的 C 语言程序和汇编程序之间能够相互调用，必须为子程序之间的调用制定一定的规则。 ATPCS 就是 ARM 程序和 Thumb 程序中子程序调用的基本规则。

1. ATPCS 概述

ATPCS（ARM-Thumb Procedure Call Standard，基于 ARM 指令集和 Thumb 指令集过程调用规则）规定了一些不同语言撰写的函数之间相互调用（mix calls）的基本规则，这些基本规则包括子程序调用过程中寄存器的使用规则、数据栈的使用规则、以及参数的传递规则。2007 年 ARM 公司正式推出 AAPCS（ARM Architecture Procedure Call Standard）标准，AAPCS 是 ATPCS 的改进版。目前，AAPCS 和 ATPCS 都是可用的标准。

常用的 C 语言编泽器编译的 C 语言子程序需要用户指定 ATPCS 类型，而对于汇编语言程序来说，完全要依靠用户来保证各子程序满足选定的 ATPCS 类型，ATPCS 规定了在子程序调用时的一些基本规则，主要包括以下 3 方面的内容：

- ❒ 各寄存器的使用规则及其相应的名字；
- ❒ 数据栈的使用规则；
- ❒ 参数传递的规则；

（1）寄存器的使用规则

1）子程序间通过寄存器 R0～R3 来传递参数。这时，寄存器 R0～R3 可记作 a0～a3。被调用的子程序在返回前无需恢复寄存器 R0～R3 的内容。

2）在子程序中，使用寄存器 R4～R11 来保存局部变量。这时，寄存器 R4～R11 可以记作 v1～v8。如果在子程序中使用了寄存器 v1～v8 中的某些寄存器，则子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值。在 Thumb 程序中，通常只能使用寄存器 R4～R7 来保存局部变量。

3）寄存器 R12 用作过程调用中间临时寄存器，记作 IP。在子程序之间的连接代码段中常常有这种使用规则。

4）寄存器 R13 用作堆栈指针，记作 SP。在子程序中寄存器 R13 不能用作其他用途。寄存器 SP 在进入子程序时的值和退出子程序时的值必须相等。

5）寄存器 R14 称为连接寄存器，记作 LR。它用于保存子程序的返回地址。如果在子程序中保存了返回地址，寄存器 R14 则可以用作其他用途。

6）寄存器 R15 是程序计数器，记作 PC。它不能用作其它用途。

（2）数据栈的使用规则

数据栈指针根据指向位置和增长方向的不同可分为 4 种:FD（Full Descending），ED（Empty Descending），FA（Full Ascending）和 EA（ Empty Ascending）。当栈指针指向栈顶元素时，称为 Full 栈。

ARM 的 ATPCS 规定默认的数据栈为 Full Descending（FD）类型，SP 指向最后一个

压入的值，数据栈由高地址向低地址生长类型，即满递减堆栈，经常使用的指令有 STMFD 和 LDMFD。并且对数据栈的操作是 8 字节对齐的。**对于汇编程序来说，如果目标文件中包含了外部调用，在汇编程序中需要使用 PRESERVE8 伪指令告诉链接器，本汇编程序堆栈数据是 8 字节对齐的。**

（3）参数传递规则

当参数个数不超过 4 个时，可以使用寄存器 R0～R3 来传递参数，如果参数多于 4 个，则将剩余的字数据通过堆栈传递，**入栈的顺序与参数传递顺序相反，即最后一个字数据先入栈，第一个字数据最后入栈。**

子程序结果返回规则：

1）结果为一个 32 位的整数时，可以通过寄存器 R0 返回。

2）结果为一个 64 位整数时，可以通过 R0 和 R1 返回，依此类推。

3）结果为一个浮点数时，可以通过浮点运算部件的寄存器 f0，d0 或者 s0 来返回。

4）结果为一个复合的浮点数时，可以通过寄存器 f0~fN 或者 d0~dN 来返回。

5）对于位数更多的结果，需要通过调用内存来传递。

2. C 程序调用汇编函数实例

ARM 编译器使用的函数调用规则就是 ATPCS 标准，也是设计可被 C 程序调用的汇编函数的编写规则。为了保证程序调用时参数传递正确，C 程序调用的汇编函数时必须严格按照 ATPCS 规则，同时需要对 C 编译器进行设置，确保其和汇编程序使用的规则一致。

如果汇编函数和调用函数的 C 程序不在同一个文件中，则需要在汇编语言中用 EXPORT 声明汇编语言起始处的标号为外部可引用符号，该标号应该为 C 语言中所调用函数的名称。这样，当链接器在链接各个目标文件时，会把标号的实际地址赋给各引用符号。在 C 语言中则需要声明函数原型并加 extern 关键字，然后才能在 C 语言中调用该函数。从 C 语言的角度看，函数名起到的作用是标识函数代码的起始地址，其作用和汇编中的标号类似。

例 7.48 调用汇编函数，实现把字符串 srcstr 复制到字符串 dststr 中。

```
//main.c
extern void strcopy (char * d, char * s ) ;   //需要调用的汇编函数原型并加 extern 关键字
int main ()
{
    char * srcstr = "0123456" ;
    char   dststr [ ] = "abcdefg" ;
    strcopy(dststr, srcstr);
   return 0 ;
}
; 汇编语言源程序 Scopy.s，汇编文件和*.c 文件在同一工程中
AREA Scopy, CODE, READONLY
EXPORT strcopy
strcopy ; 必须与 EXPORT 后面标号一致
LOOP    LDRB    R2, [R1], #1; R1 指向源字符串地址，取出字符内容存入 R2
; 更新 R1=R1+1，第一次调用时 R1 指向源字符串首地址
        STRB    R2, [R0], #1; R0 指向目的字符串地址，R2 中内容存入 R0 指向内存单
```

```
；元，更新 R0=R0+1，第一次调用时 R0 指向目的字符串首地址
        CMP     R2, #0
        BNE     LOOP; 先执行后判断，源字符串的终止符‘\0’也复制到目的字符串
        MOV     PC, LR
END
```

字符串初始化后目的字符串在内存中状态如图 7.12 所示，执行汇编子程序 strcopy 时寄存器状态如图 7.13 所示，R0 和 R1 分别存放的是目的字符串和源字符串的起始地址。

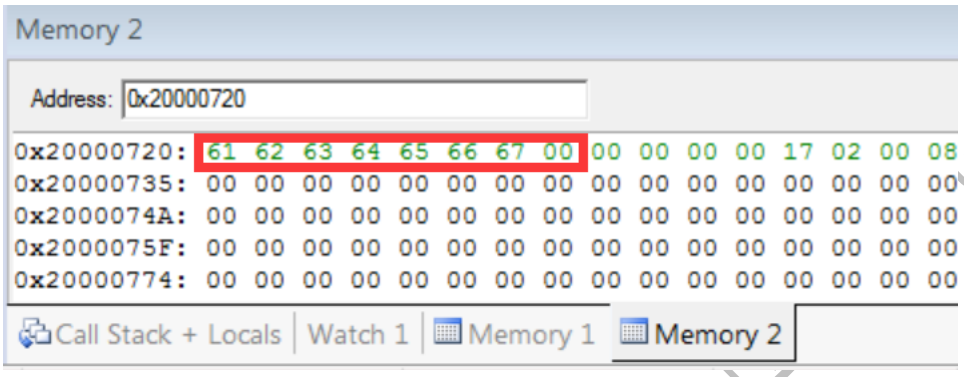

图 7.12 例 7.48 中目的字符串初始化后在内存中存储状态

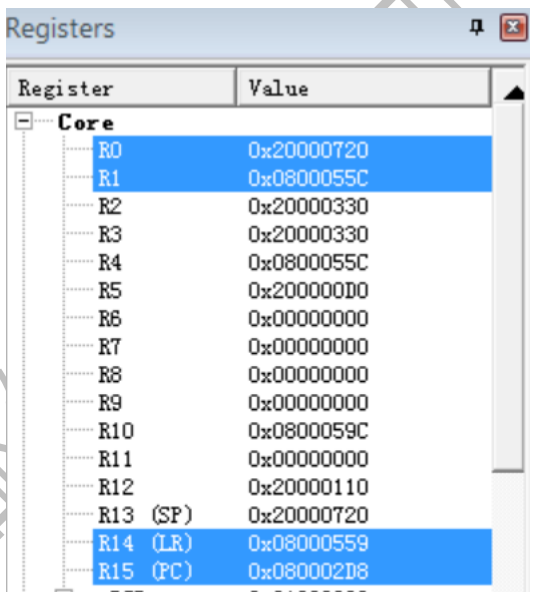

图 7.13 例 7.48 在 strcopy 设置断点后中断时寄存器状态

从例 7.48 中可以看出在 C 语言中使用 strcopy(dststr，srcstr)语句调用汇编函数时，参数 dststr 是第一个参数，因此该参数传递到 R0 中（即字符串 dststr 的首地址保存在 R0 中），而参数 srcstr 是第二个参数，因此该参数传递到 R1 中（即字符串 srcstr 的首地址保存在 R1 中）。汇编语言中利用 R1 间接寻址可以完成源字符串的读取，将源字符串 srcstr 中的字符读到 R2 中，利用 R0 间接寻址将 R2 中的字符存到目的字符 dststr 中，直到读到字符串终止符‘\0’结束循环，执行结果如图 7.14 所示。

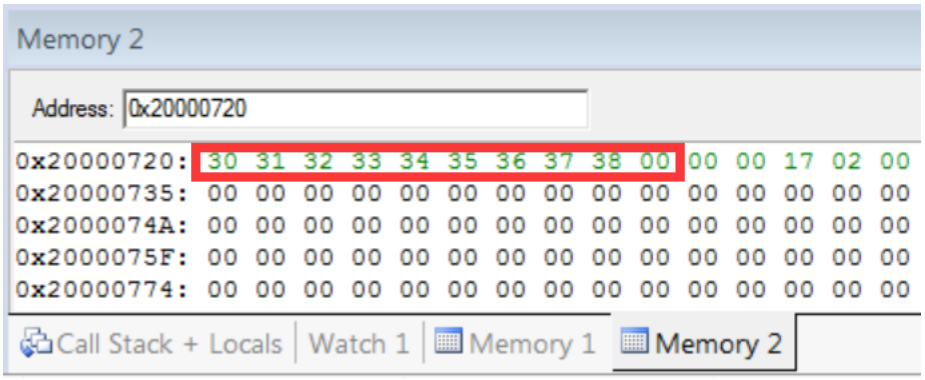

图 7.14 例 7.48 中目的字符串在汇编子程序 strcopy 执行后的内容

3. 汇编程序调用 C 函数实例

在汇编程序中调用 C 语言函数，需要在汇编程序中利用 IMPORT 说明对应的 C 函数名，按照 ATPCS 的规则传递参数（参数少于 4 个时利用 R0-R3 传递，参数大于 4 个时前 4 个参数仍利用 R0-R3 传递，其余的参数利用堆栈传递），完成各项准备工作后利用跳转指令跳转到 C 函数入口处开始执行，跳转指令后所跟标号为 C 函数的函数名。

例 7.49 汇编程序中调用 C 函数实现求两个整数相加的和。

```
PRESERVE8 ; 声明调用 c 接口时栈是 8 字节对齐的
AREA RESET, CODE, READONLY
ENTRY
IMPORT CAL              ; 声明 CAL 为外部引用符号
LDR    SP, =0x20000460 ; 设置堆栈指针
MOV    R0，#1           ; 参数 1 赋 R0
MOV    R1，#2           ; 参数 2 赋 R1
BL    CAL
LDR R4, =0x20000000    ; 结果 R0 存入内存单元
STR R0, [R4]
END
//C 函数定义在文件 example.c 中，和汇编文件在同一工程中
int CAL(int a, int b)
{
    return (a+b);
}
```

例 7.49 中利用 R0 和 R1 传递参数，调用 C 函数 CAL 实现立即数 1+2 的计算求和。执行指令 BL CAL 前后寄存器状态如图 7.15 和 7.16 所示。

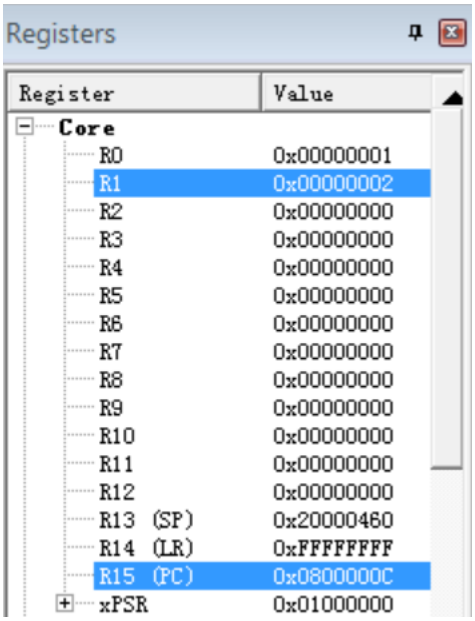

图 7.15 执行 BL CAL 前寄存器状态

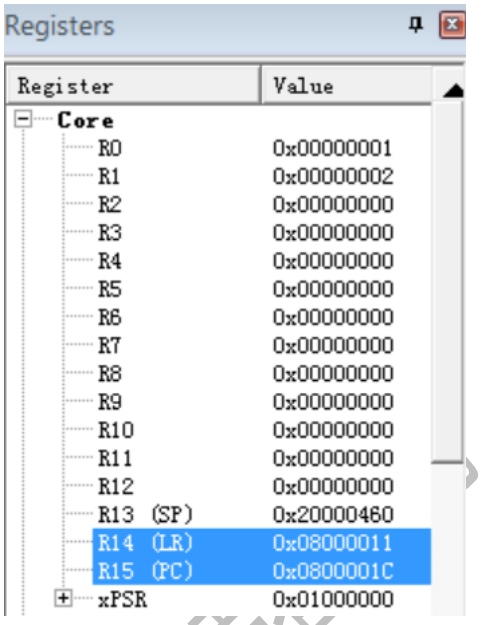

图 7.16 执行 BL CAL 后寄存器状态

由图 7.16 可知 BL 指令完成 2 个操作，即将主程序的返回地址放在 LR 寄存器中，同时将 PC 寄存器指向子程序 CAL 的入口点。子程序 CAL 的操作反汇编后如图 7.17 所示。

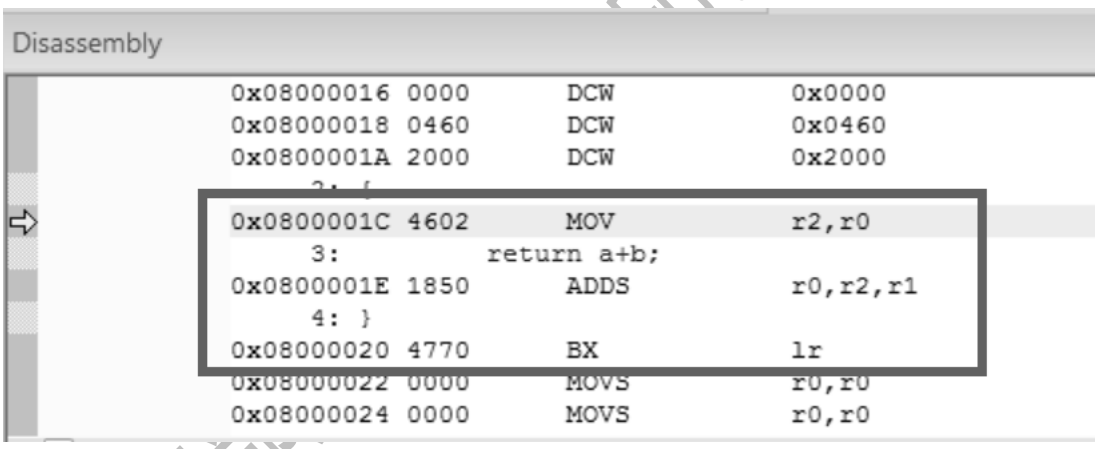

图 7.17 子程序 CAL 执行的指令

由图 7.17 可知子程序 CAL 执行了以下 3 个操作：

（1）MOV R2, R0 因为子程序结果要在 R0 存放，所以保存 R0 中第一个参数到 R2 寄存器中。

（2）ADDS R0, R2, R1 执行加法计算 R0=R2+R1。

（3）BX LR 返回主程序，将保存在 LR 中的主程序返回地址赋给 PC。

例 7.49 中传递的参数个数小于 4，下面看一个参数个数大于 4 的程序示例。

例 7.50 汇编程序中调用 C 函数实现求 5 个整数相加的和。

```
PRESERVE8
AREA RESET, CODE, READONLY
ENTRY
IMPORT CAL ;
LDR    SP, =0x20000460 ; 设置堆栈指针
MOV   R0，#1;   R0=1
ADD   R1，R0，R0;   R1=2
```

```
ADD   R2，R1，R0; R2=3
ADD   R3，R0，R2; R3=4
ADD   R4，R0，R3; R4=5
STR   R4，[SP，#-4]!
BL   CAL
LDR R4, =0x20000000;  结果 R0 存入内存单元
STR R0, [R4]
END
//C 源程序 example.c，和汇编文件在同一工程中
int CAL(int a, int b, int c, int d, int e)
{
    return (a+b+c+d+e);
}
```

例 7.50 调用 C 函数 CAL 实现立即数 1+2+3+4+5 的计算求和，在传递参数超过 4 个时，第 5 个参数采用堆栈方式传送，子程序 CAL 的操作反汇编后如图 7.18 所示。

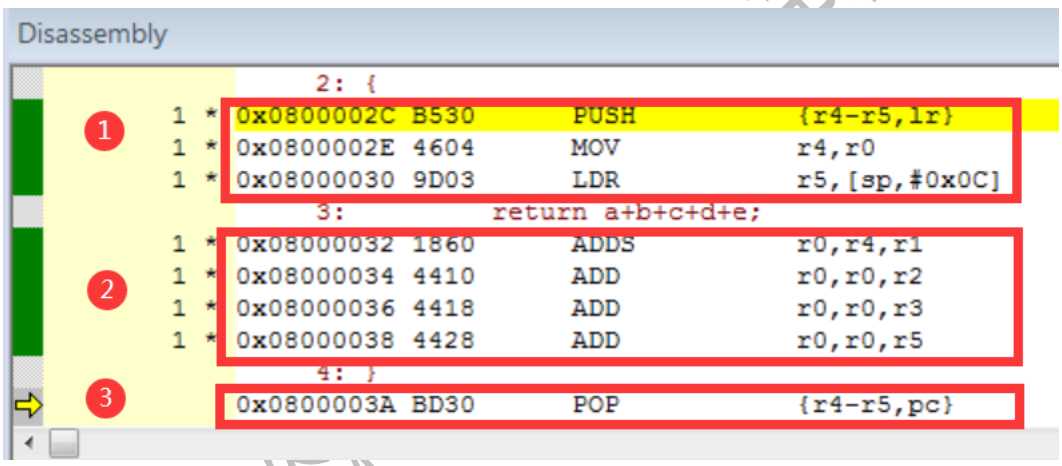

图 7.18 例 7.50 中子程序 CAL 执行的指令

由图 7.18 可知例 7.50 中子程序 CAL 执行了以下 3 个操作：

（1）R4、R5、LR 寄存器入栈，R0 寄存器内容存入 R4，存入堆栈的第 5 个参数存入 R5。

（2）累加求和，计算结果存入 R0。

（3）从堆栈弹出第 1 步保存的 R4、R5，并将 LR 中的主程序返回地址赋给 PC。

7.4.2 C/C++语言与汇编语言的混合编程

在嵌入式系统开发中，目前使用的主要编程语言是 C 和汇编，C++已经有相应的编译器，但是现在使用还是比较少的。在稍大规模的嵌入式软件中，大部分的代码都是用 C 编写的，主要是因为 C 语言的结构比较好，便于理解，而且有大量的支持库。尽管如此，很多地方还是要用到汇编语言，例如开机时硬件系统的初始化，包括 CPU 状态的设定、中断的使能、主频的设定、以及 RAM 的控制参数及初始化、一些中断处理方面也可能涉及汇编，另外一个使用汇编的地方就是一些对性能非常敏感的代码块，为了达到优化的目的，需要编写汇编代码。而且，汇编语言是和 CPU 的指令集紧密相连的，作为涉及底层的嵌入式系统开发，熟练使用汇编语言也是必须的。在 C 程序中使用内嵌汇编代码，可以在 C/C++程序中

实现 C/C++不能完成的一些操作，同时程序的代码效率也会更高。

1. 在 C 语言程序中嵌入汇编指令

如果要在 C 程式中嵌入汇编可以有两种方法：内联汇编和内嵌汇编。嵌入汇编使用的标记是 __asm 或者 asm 关键字，用法如下：

```
__asm
{
    instruction [; instruction]
    …
    [instruction]
}
asm(“instruction [; instruction]”);
```

（1）内联汇编的示例代码如下:

```
int Add(int   i)
{
  int r0;
  __asm
  {
    ADD   R0, i, 1
    EOR   i, R0, i
  }
  return i;
}
```

（2）内嵌汇编示例代码如下:

```
#include <stdio.h>

__asm void my_strcpy(const char *src, char *dst)

{

    LOOP LDRB  R2, [R0], #1; R0 保存第一个参数

    STRB  R2, [R1], #1; R1 保存第二个参数

    CMP   R2, #0

    BNE   LOOP

    BLX    LR;  返回指令须要手动加入

}

int main(void)

{

    const char *a = "Hello world!";

    char b[20];

    my_strcpy (a, b);
```

```
    return 0;
}
```

由两种方法的示例可以看出：内联汇编可以直接嵌入 C 代码使用，而内嵌汇编更像是一个函数。由于内联式汇编只能在 ARM 状态中进行，而 Cortex-M3/M4 只支持 Thumb-2，所以 Cortex-M3/M4 只能使用内嵌汇编的方式，也就是第二种方式。

在 C 语言中内嵌汇编指令与汇编程序中的指令有些不同，存在一些限制，主要有下面几个方面：

（1）不能直接向 PC 寄存器赋值，程序跳转要使用 B 或者 BL 指令。

（2）在使用物理寄存器时，不要使用过于复杂的 C 表达式，避免物理寄存器冲突。

（3）R12 和 R13 可能被编译器用来存放中间编译结果，计算表达式值时可能将 R0 到 R3、R12 及 R14 用于子程序调用，因此要避免直接使用这些物理寄存器。

（4）一般不要直接指定物理寄存器，而让编译器进行分配。

例 7.51 在 C 程序中使用内联汇编代码实现字符串复制。

```
#include <stdio.h>
void asm_strcpy(const char *src，  char *dest)
{
  char ch;
  __asm/ /注意是双下划线
  {
      LOOP:
      LDRB ch, [src], #1
      STRB ch, [dest], #1
      CMP ch, #0
      BNE LOOP
  }
}
int main()
{
  char *a = "ok!";
  char b[64];
  asm_strcpy(a, b);
  return 0;
}
```

在这里 C 和汇编之间的值传递是用指针来实现的，因为指针对应的是地址，所以汇编中也可以访问。

2. 在汇编中调用 C 语言定义的全局变量

使用内联或者内嵌汇编不用单独编辑汇编语言文件使用起来比较方便，但是有诸多限制，当汇编文件较多的时候就需要使用专门的汇编文件编写汇编程序，在 C 语言和汇编语言进行数据传递的最简单的形式是使用全局变量。

7.4.3 C 程序中访问特殊寄存器的指令

1. CMSIS

微控制器软件接口标准（CMSIS，Cortex Microcontroller Software Interface Standard）是 ARM 公司为统一软件结构而为 Cortex 微控制器制定的软件接口标准。CMSIS 为处理器和外设提供了一致且简单的软件接口，方便软件开发，易于软件重用，缩短了开发人员的学习过程和应用项目的开发进程。目前，很多针对 Cortex-M 微控制器的软件产品都是 CMSIS 兼容的。

CMSIS 始于为 Cortex-M 微控制器建立统一的设备驱动程序库，即其核心组件 CMSIS-CORE 之后，添加了其他 CMSIS 组件，如 CMSIS-RTOS，CMSIS-DSP 等。

❒ CMSIS-CORE 为 Cortex-M 处理器核和外设定义应用程序接口 API（Application Programming Interface），也包括一致的系统启动代码。

❒ CMSIS-RTOS：提供标准化的实时操作系统 RTOS（Real-Time Operating System），以便在软件模板、中间件、程序库和其他组件能够获得 RTOS 支持。

❒ CMSIS-DSP：为数字信号处理 DSP（Digital Signal Processing）实现的函数库，包含各种定点和单精度浮点数据类型，超过 60 个函数。

CMSIS 提供了一个与厂家无关的、基于 Cortex-M 处理器的硬件抽象层，如图 7-19 所示。从软件角度看，CMSIS-CORE 进行了一系列标准化工作：标准化处理器外设定义、标准化处理器特性的访问函数、标准化系统异常处理程序的函数名等。用户的应用程序既可以通过 CMSIS 层提供的函数（包括设备厂商提供的外设驱动程序）访问微控制器硬件，也可以利用 CMSIS 的标准化定义直接对外设编程，控制底层的设备。如果移植了实时操作系统，用户应用程序也可以调用操作系统函数。CMSIS 文件包含在微控制器厂商提供的设备驱动程序包中，在开发应用程序的时候，用户应用程序需要引用相关的头文件。

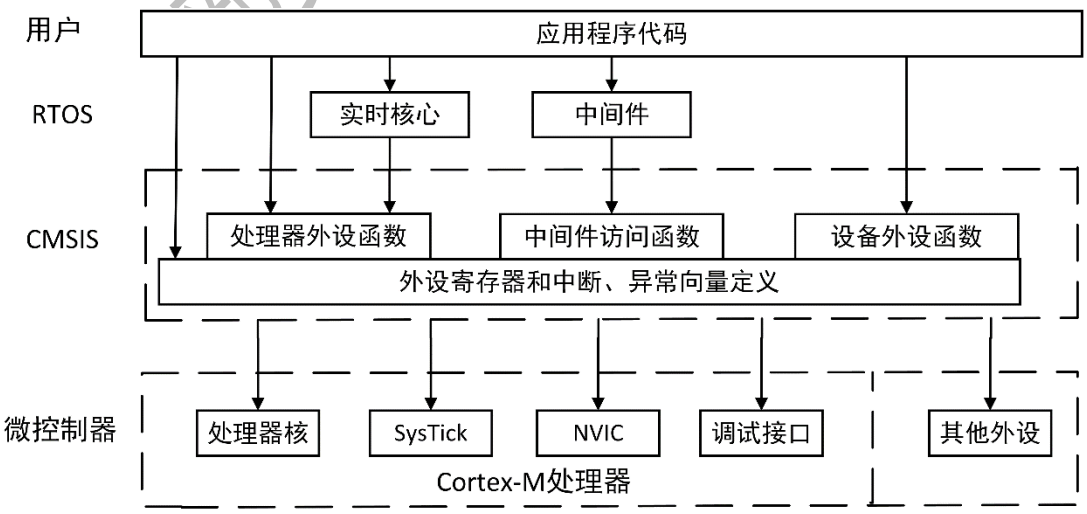

图 7.19 基于 CMSIS-CORE 的开发结构

2.用 C 指令访问特殊寄存器

有些指令无法在 C 编译器中利用普通 C 语句生成，如触发休眠（WFI、WFE）及存储器屏障（ISB、DSB 和 DMB）指令。若要使用这些指令，可以利用下面的方法：

- ❒ 使用 CMSIS 提供的内在函数（位于 CMSIS-Core 头文件中）。
- ❒ 使用编译器提供的内在函数。
- ❒ 利用内联汇编（或 ARM/Keil 工具链的嵌入汇编）插入所需指令
- ❒ 利用关键字（如 ARM/Keil 工具链可用_svc 产生 SvC 指令）或习语识别等编译器相关的特

性。

- ❒ 有些情况下，还需要访问处理器内的特殊寄存器。同样具有多种选择:
- ❒ 使用 CMSIS-Core 提供的处理器访问函数。
- ❒ 使用 ARMC 编译器中的寄存器名变量等编译器相关的特性。
- ❒ 利用内联汇编或嵌入汇编插入汇编代码。

一般来说利用 CMSIS-Core 函数好些，因为和编译器相对独立方便移植。

1）内在函数

内在函数有两种，下面将会逐一介绍。

（1）CMSIS-CORE 内在函数

CMSIS-CORE 的头文件定义了一组用以访问特殊寄存器的内在函数，它们位于 CMSIS-CORE 文件 core_cmInstr.h 和 core_c4_simd.h（用于 Cortex-M4 的 SIMD 指令）中。

（2）编译器相关的内在函数

编译器相关的内在函数的使用方法和 C 函数差不多，只是它们是内置在 C 编译器中的。这种方式通常会提供最优化的代码，不过，函数定义依赖于编译工具，因此，应用代码无法在工具链间移植。

对于一些工具链，编译器直接支持 CMSIS-CORE 内在函数，而其他的工具链（如 ARM C 编译器）则会将这两种内在函数区分开， CMSIS-CORE 内在函数生成的代码可能并不是最优化的。需要注意的是，有时编译器相关的内在函数的名称可能会和 CMSIS-CORE 的类似如 ARMC 编译器中的 void_wfi(void)， CMSIS-CORE 中则是_WFE(void)。不过，所需的参数可能会不同。

2）内联汇编和内嵌汇编

有些情况下，可能需要在 C 代码中利用内联汇编插入汇编指令，如要在 gcc 中使用 SVC 时。它还可用于生成优化的代码，因为可以很好地控制所生成指令的顺序。不过，利用内联汇编创建的程序是和工具链相关的（可移植性差）。ARM C 编译器（包括 Keil MDK-ARM 和 ARM DS-5 Professional）还支持一种名为嵌入汇编的特性，利用嵌入汇编，可以在 C 程序文件内创建汇编函数。

对于版本 5.01 之前的 ARM C 编译器或更早的 Keil MDK-ARM，内联汇编只能用于 ARM 指令，无法生成 Thumb 指令。因此，在较早版本的 ARM 工具链下，需要使用嵌入汇编来插

入汇编代码。ARM C 编译器 5.01 或 Keil MDK-ARM4.60 及之后版本都支持内联汇编。

3）使用其他的编译器相关的特性

多数 C 编译器具有可以方便产生特殊指令的多种特性。例如，对于 ARM C 编译器或 Keil MDK-ARM，可以利用_svc 关键字插入 SVC 指令。

另外一个编译器相关的特性为习语识别。若是以某种形式书写的数据运算 C 语句，C 编译器会识别出这种功能并以简单的指令代替。

4）访问特殊寄存器

CMSIS-CORE 提供了访问 Cortex-M3/M4 处理器内特殊寄存器的多个函数。若使用 ARM C 编译器（包括 Keil MDK-ARM 和 ARM DS-5 Professional），可以使用“已命名寄存器变量”特性来访问特殊寄存器。其语法为:

register type var-name_arm(reg)

其中，type 为已命名寄存器变量的数据类型; var-name 为已命名寄存器变量的名称;reg 为指明要使用哪个寄存器的字符串。

例如可以将寄存器名声明为：

register unsigned int reg_apsr_asm("apsr");

reg_apsr=reg_apsr & 0xF7FFFFFUL; 清除 APSR 中的 Q 标志

可以使用已命名寄存器变量特性来访问表 7.2 所示的寄存器。

表 7.2 利用“已命名寄存器变量”特性访问处理器寄存器

寄存器	asm 中的字符串
APSR	“apsr”
BASEPRI	“basepri”
BASEPRI_MAX	“basepri_max”
CONTROL	“control”
EAPSR (EPSR+ APSR)	“eapsr”
EPSR	“epsr”
FAULTMASK	“faultmask”
IAPSR(IPSR+APSR)	“iapsr”
IEPSR(IPSR+EPSR)	“iepsr”
IPSR	“ipsr”
MSP	“msp”
PRIMASK	“primask”
PSP	“psp”
PSR	“psr”
r0~r12	“r0”~“r12”
r13	“r13”或“sp”
r14	“r14”或“lr”
r15	“r15”或“pc”
XPSR	“xpsr”

习题

1. 汇编语言和 C 语言相比各有什么特点？
2. 什么是 ATPCS 标准？
3. 编写一个完整 ARM 汇编程序实现如下功能：当 R3>R2 时，将 R2+10 存入 R3，否则将 R2+100 存入 R3。
4. 将数据段中 10 个数据中的偶数个数统计后放入 R0 寄存器。
5. 将数据段中 10 个有符号数中的正数个数统计后放入 R0 寄存器。
6. 试编写一个循环程序，实现 1 至 100 的累加。
7. 汇编程序如何定义子程序？如何调用子程序?
8. 编写完整程序并利用汇编子程序计算 N!（N<=10）。
9. 编写完整汇编程序调用 C 函数计算 N!（N<=10）。
10. C 程序调用汇编函数计算字符串长度，并返回长度值。
11. C 程式中嵌入汇编有哪两种方式？Cortex M4 可以使用哪种方式，为什么？