第3章　ARM程序设计基础

3.1 ARM的指令系统介绍

- 问题：
  - ARM指令分为哪几类？
  - 指令条件码是什么，主要有哪些？
  - ARM寻址方式有哪些？
  - Thumb指令有哪些？
- 重点：
  - ARM指令及其寻址方式。
- 内容：
  - ARM指令系统，
  - ARM寻址方式，
  - ARM指令集,
  - Thumb指令集。

3.1.1 ARM指令系统

任务：了解ARM指令分类，理解ARM指令格式和指令条件码。

ARM指令分类

- ARM指令分为 六大类：
- 数据处理指令：数据传送指令、算术运算指令、逻辑运算指令、比较指令。
- Load／store指令：STR/LDR、LDM/STM
- 跳转指令:BL,BEQ……
- 程序状态寄存器处理指令：MSR， MRS
- 协处理器指令
- 异常产生指令

1ARM 指令集概述

1.1 ARM 指令集特点

(1)RISC ，译码机制简单；

ARM 指令集机器编码基本格式:

条件码	指令码	目的寄存器	操作数1寄存器	操作数2
31-28	27-20	19-16	15-12	11-0

图 1 ARM 指令机器编码基本格式

(2) 程序的启动从 ARM 指令集开始，进入异常转化为 ARM 状态，运行 ARM 指令集指令；

1. ARM指令格式

- ARM指令采用固定的32位二进制编码，其基本格式如下：
- <操作码>{<条件>}{S}　<目标寄存器>,<操作数1寄存器>{,<操作数2>}
- SUBEQs R0,R1,#0x3f
- 操作码：即指令助记符，如ADD表示加法指令、CMP表示比较指令。
- 条件：表示可选的指令条件码，定义执行条件，当指令没有条件码时为无条件执行。关于条件码的说明见后续说明。
- S：为可选后缀，表示该指令的操作结果对CPSR的影响。若指定了S，则根据指令操作结果更新CPSR的条件标志位（N、Z、C、V）。
- 目标寄存器：为操作结果寄存器。
- 操作数1寄存器：为存放第一个操作数的寄存器。
- 操作数2：为第2个操作数，可以是寄存器，也可以是立即数。

2. 指令条件码

- 在ARM指令32位编码中，最高4位[31：28]为指令条件码（即cond）。
- 每种条件码用两个英文缩写字符表示，可添加在指令助记符的后面，表示指令执行时必须要满足的条件。
- ARM指令根据CPSR中的条件标志位自动判断是否执行该指令。当条件满足时指令执行，否则指令被忽略，继续执行下一条指令。

表3-2 条件码含义

条件码[31：28]	助记符后缀	解释	CPSR标志位状态
0000	EQ	相等	Z置位
0001	NE	不相等	Z清零
0010	CS/HS	进位/无符号数大于或等于	C置位
0011	CC/LO	无进位/无符号数小于	C清零
0100	MI	负数	N置位
0101	PL	正数或零	N清零
0110	VS	溢出	V置位
0111	VC	未溢出	V清零
1000	HI	无符号数大于	C置位Z清零
1001	LS	无符号数小于或等于	C清零Z置位
1010	GE	有符号数大于或等于	N等于V
1011	LT	有符号数小于	N不等于V
1100	GT	有符号数大于	Z清零且N等于V
1101	LE	有符号数小于或等于	Z置位且N不等于V
1110	AL	总是	任何状态

实例

- C代码　if(a>b) a++; else b++;
- ARM代码：　CMP　R0,R1
- 　　ADDHI　R0,R0,#1
- 　　ADDLS　R1,R1,#1

- C代码　if ((a!=10)&&(b!=20)) a=a+b;
- ARM代码：　CMP　　R0，#10
- 　　CMPNE　R1，#20
- 　　ADDNE　R0，R0，R1

第3章　ARM程序设计基础

3.1.2 ARM寻址方式

任务：掌握ARM指令的寻址方式分类，理解每一种寻址方式的含义

- 所谓寻址方式就是处理器根据指令中给出的地址信息来寻找物理地址的方式。目前ARM指令系统支持的基本寻址方式有7种：
- 立即寻址
- 寄存器寻址
- 寄存器间接寻址
- 基址寻址
- 相对寻址
- 多寄存器寻址
- 堆栈寻址

1. 立即寻址

- **立即**寻址也称立即数寻址，这种寻址方式就在指令中给出操作数，**只要取出指令也就取到了操作数**，这个操作数称为立即数，对应的寻址方
- 例如：
- ADD R
- AND R
- 在以上两
以十六进
- 立即数一

，对于

- 存在的问题：32位长立即数的编码问题（合法性问题）
- 原因：在指令中，立即数作为操作数2出现，编码格式中仅安排12位空间，32位立即数显然不能直接编码；
- 解决：12位编码包括8位常数和4位循环右移值，由8位常数循环右移4位值的2倍得到最后32位立即数

- 值得庆幸的：其实没必要一个一个数的算，编译系统为我们提供了伪指令LDR
- LDR R1,=0x12345678

2. 寄存器寻址

- **寄存器**寻址就是利用寄存器中的数值作为操作数，指令中地址码给出的是寄存器的编号。
- **例如**：
- ADD R0，R1，R2　　；R0←R1+R2
- 该指令的含义是将**R1和R2的内容相加**，结果放在**R0中**。

3. 寄存器间接寻址

- 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，寄存器起到地址指针的作用。
- 例如：
- ADD R0, R1,[R2] ；R0←R1+[R2]
- LDR R0, [R1] ；R0←[R1]
- STR R0, [R1] ；R0→[R1]
- LDR/STR:左寄存器，右存储器

4. 基址寻址

- **基址**寻址就是将基址寄存器的内容与指令中给出**的地址偏移量相加，得到一个操作数的有效地址，用于**访问基址附近的存储单元。
- **基址**寻址方式常用于访问某某地址附近的单元
- **采用基址**变址寻址方式的指令通常有以下几种**形式**：
- LDR R0，[R1，#4] ；R0←[R1+4]
- LDR R0，[R1]，#4 ；R0←[R1]、R1=R1+4
- LDR R0，[R1，#4]！；R0←[R1+4]、R1=R1+4
- LDR R0，[R1，R2] ；R0←[R1+R2]

5. 多寄存器寻址

- **多寄存器**寻址是指一条指令可以一次传递多个寄存器的值。这种寻址方式允许一条指令一次最多可以传送**16个通用寄存器的**值。

基址寄存器变化方式:

IA (Increment after Operating) ：操作完后地址递增

IB (Increment before Operating) ：地址先增后完成操作

DA (Decrement after Operating) ：操作完后地址递减

DB (Decrement before Operating) ：地址先减后完成操作

多寄存器语法表示:

多寄存器用“{}”包含，连续寄存器使用“-”间隔，否则用“,”分隔；

5. 多寄存器寻址

- **多寄存器**寻址是指一条指令可以一次传递多个寄存器的值。这种寻址方式允许一条指令一次最多可以传送16**个通用寄存器的**值。
- 例如：
- LDMIA R0 ，{R1，R2，{R1-R4}

{R4，R3，R2，R1}

R2←[R0+4]

R3←[R0+8]

R4←[R0+12]

- 例2：LDM指令与STM指令配合实现数据块拷贝

```
LDMIA    R0, {R1-R5}
；以R0为基地址读取五字存储单元数据加载至R1-R5
STMIA    R6, {R1-R5}
；将R1-R5中数据依次存入R6为起始地址的存储单元
```

6. 堆栈寻址

- 堆栈是一种按照“先进后出”或“后进先出”方式进行数据存取的存储区。指向堆栈的地址寄存器称为堆栈指针（SP），堆栈的访问是通过堆栈指针（R13，ARM处理器的不同工作模式对应的物理寄存器各不相同）指向一块存储区域（堆栈）来实现的。
- 堆栈根据其内存地址增长的方向分为递增堆栈和递减堆栈。
- 根据堆栈指针指向数据位置的不同，又可分为满堆栈和空堆栈。

- 递增、递减、空堆栈、满堆栈进行组合就可以产生4种类型的堆栈。
- （1）满递增堆栈（FA）：堆栈指针指向最后压入的数据，而且压入数据时堆栈由低地址向高地址生成。
- （2）空递增堆栈（EA）：堆栈指针指向下一个要压入的数据的空位置，而且压入数据时堆栈由低地址向高地址生成。
- （3）**满递减堆栈**（FD）：堆栈指针指向最后压入的数据，而且压入数据时堆栈由高地址向低地址生成。
- （4）空递减堆栈（ED）：堆栈指针指向下一个要压入的数据的空位置，而且压入数据时堆栈由高地址向低地址生成。

例 1：

```
STMFD  SP!,  {R0-R7, LR}    ;入栈
LDMFD  SP!,  {R0-R7, LR}    ;弹出堆栈
```

第一行指令的入栈效果：

旧数据	0x40000460	← SP原来的位置
LR	0x4000045C	
R7	0x40000458	
R6	0x40000454	
R5	0x40000450	
R4	0x4000044C	
R3	0x40000448	
R2	0x40000444	
R1	0x40000440	
R0	0x4000043C	← 更新后的SP

图 3　STMFD 指令入栈效果

- 问题：满递减与块拷贝的哪种方式一致？
- DB

例 2：

事实上，堆栈寻址与多寄存器寻址均可操作堆栈，在需要保存特定某些寄存器值时，采用 STM 进行压栈，采用 LDM 操作弹出堆栈；

保存数据： stmdb sp!,{r0-r12,lr}

弹出数据： ldmia sp!,{r0-r12,lr}

7. 相对寻址

- 相对寻址可以看作将程序计数器PC作为基址的一种基址变址寻址方式。指令的地址标号作为位移量，与PC相加得到操作数的有效地址。
- 例如:
- BL SUBR ；调用子程序SUBR
- … ；返回位置
- SUBR .. .. ；子程序入口地址
- MOVE PC,R14 ；返回

第3章　ARM程序设计基础

3.1.3 ARM指令集介绍

任务：理解每一种指令的含义。

1. 指令分类

- ARM微处理器的指令集可以分为跳转指令、数据处理指令、程序状态寄存器处理指令、load/store指令、协处理器指令和异常产生指令6大类。
- 基本的指令如表3-1所示。

助记符	指令功能描述	助记符	指令功能描述
ADC	带进位加法指令	MRC	从协处理器寄存器到ARM寄存器
ADD	加法指令	MRS	传送CPRS到通用寄存器
AND	逻辑与指令	MSR	传送通用寄存器到CPRS
B	跳转指令	MUL	32位乘法
BIC	位清零指令	MLA	32位加法
BL	带返回的跳转指令	MVN	数据取反传送指令
BLX	带返回和状态切换的跳转指令	ORR	逻辑或指令
BX	带状态切换的跳转指令	RSB	逆向减法指令
CDP	协处理器数据操作指令	RSC	带借位的逆向减法指令
CMN	比较反值指令	SBC	带借位减法指令
CMP	比较指令	STC	协处理器寄存器写入存储器指令
EOR	异或指令	STM	批量内存字写入指令
LDC	存储器到协处理器的数据传送指令	STR	寄存器到存储器的数据传送指令
LDM	加载多个寄存器指令	SUB	减法指令
LDR	存储器到寄存器的数据传送指令	SWI	软件中断指令
MCR	从ARM寄存器传送到协处理器寄存器	SWP	交换指令
MLA	乘加运算指令	TEQ	相等测试指令
MOV	数据传送指令	TST	位测试指令

2. 数据处理指令

- ARM数据处理指令包括：算术运算、逻辑运算、数据传送、比较、测试、乘法等。

2. 数据处理指令

- （1）算术运算指令——ADD、SUB、RSB、ADC、SBC、RSC
- 指令格式：操作码{条件}{S} 目标寄存器，操作数1寄存器，操作数2
- 指令功能：指令用于加、减、反减等算术运算，包括带进位的算术运算。
- ADD指令用于将操作数1寄存器的值和操作数2相加。
- SUB指令用于操作数1寄存器的值中减去操作数2。
- RSB指令用于操作数2的值减去操作数1。反减的优点在于操作数2的可选范围比较大。
- ADC、SBC、RSC指令分别是ADD、SUB、RSB的带进（借）位的运算，运算结果将影响CPSR中进位标志C。

2. 数据处理指令

- 范例1：下面指令完成64位整数相加。
- ADDS R4,R0,R2 ；加低位有效字
- ADC R5,R1,R3 ；加高位有效字（连同低位进位）
- 范例2：下面指令完成96位减法。
- SUBS R3,R6,R9
- SBCS R4,R7,R10
- SBC R5,R8,R11

2. 数据处理指令

- （2）逻辑运算指令——AND、ORR、EOR、BIC
- **指令格式**：**操作**码{**条件**}{S} **目**标寄存器，操作数1**寄存器**，**操作数**2
- **指令功能**：AND、ORR、EOR**分**别完成逻辑与、逻辑或、逻辑异或**运算**，BIC**指令用于将操作数1寄存器中的位与操作数2中相**应位的**反**码进行**“与”运算**，该指令可以实现操作数1**某些位清**0。

2. 数据处理指令

- 范例：
- AND R0,R0,#3 ；保持R0的0、1位，其余清0
- ORR R0,R0,#3 ；置位R0的0、1位，其余不变
- EOR R0,R0,#3 ；反转R0的0、1位，其余不变
- BIC R0,R0,#3 ；清0 R0的0、1位，其余不变

- （3）数据传送指令——MOV和MVN
- 指令格式：操作码{条件}{S} 目标寄存器，源操作数
- 指令功能：
- MOV指令将源操作数的值送往目标寄存器。
- MVN是“取反传送”，该指令将源操作数按位取反后的结果送往目标寄存器。

- 范例：
- MOV R0,R1 ；将R1的值传送到R0
- MOV PC,R14 ；将R14的值传送到PC，常用于子程序返回
- MOVS R1,R0,LSL #2 ；将R0的值左移2位后传送到R1
- MVN R1,R0 ；将R0的值按位取反后传送到R1

- （4）比较指令——CMP和CMN
- 指令格式：操作码{条件} 操作数1寄存器，操作数2
- 指令功能：
- CMP指令用于把操作数1寄存器的值与操作数2（寄存器或立即数）的值进行比较，同时更新CPSR中条件标志位的值。该操作实际上进行了一次减法运算，但不保存结果，只改变条件标志位。
- CMN指令表示“取反比较”，将操作数1和操作数2相加，并根据结果修改条件标志位。

- 范例：
- CMP R1,R0
- CMN R1,#50

- （5）测试指令——TST和TEQ
- **指令格式**：**操作**码{**条件**} **操作数1寄存器**，**操作数2**
- **指令功能**：
- TST**表示位**测试，对两个操作数进行按位**“与”操作**，**并根据**结果更**新条件**标志位。通常用于测试寄存器中某些位是1还是0。
- TEQ**表示**测试相等，对两个操作数进行按位**“异或”运算**，**根据**结**果更新条件**标志位。通常用于比较两个操作数是否相等。

- （6）乘法指令
- 乘法指令完成两个32位寄存器数据的乘法运算，运算结果分32位和64位数据。乘法指令总共有六种格式，如表3-3所示

表3-3 乘法指令的六种格式

助记符	说明	操作
MUL Rd，Rm，Rs	32位乘法指令	Rd←Rm×Rs （Rd≠Rm）
MLA Rd，Rm，Rs，Rn	32位乘加指令	Rd←Rm×Rs+Rn （Rd≠Rm）
UMULL RdLo，RdHi，Rm，Rs	64位无符号乘法指令	（RdLo，RdHi）←Rm×Rs
UMLAL RdLo，RdHi，Rm，Rs	64位无符号乘加指令	（RdLo，RdHi）←Rm×Rs+（RdLo，RdHi）
SMULL RdLo，RdHi，Rm，Rs	64位带符号乘法指令	（RdLo，RdHi）←Rm×Rs
SMLAL RdLo，RdHi，Rm，Rs	64位带符号乘加指令	（RdLo，RdHi）←Rm×Rs+（RdLo，RdHi）

数据交换指令

- ARM微处理器所支持数据交换指令能在存储器和寄存器之间交换数据，数据交换指令有两条
- SWP 字数据交换指令
- SWP R0,R1,[R2] ;[R2]->R0,R1->[R2]
- SWP R0,R0,[R1] ;R0<→[R1]
- SWPB 字节数据交换指令
- SWPB R0,R1,[R2] ;[R2]->R0,R1->[R2]
- SWPB R0,R0,[R1] ; R0<→[R1]

移位指令

- 移位操作在ARM指令集中不作为单独的指令使用，只能作为指令格式中是一个字段
- 移位操作包括如下6种类型：
- LSL逻辑左移 MOVS R3,R1,LSL,#2;R3=R1<<2
- LSR逻辑右移
- ASL 算术左移
- ASR 算术右移
- ROR 循环右移
- RRX 带扩展的循环右移

第3章　ARM程序设计基础

3.1.3 ARM指令集介绍

任务：理解每一种指令的含义。

3. 跳转指令

- 跳转指令用于实现程序流程的转移，在ARM中可以有两种方法实现程序流程的转移。
- 一种方法是直接向PC寄存器（R15）中写入转移的目标地址值，通过改变PC的值实现程序的跳转；

  可以实现在4GB的地址空间中的任意跳转

  MOV PC,R14

3. 跳转指令

- 另一种方法是本节即将介绍的跳转指令。
- ARM的跳转指令可以从当前指令向前或向后的32M空间跳转，包括

  B（简单跳转指令）、

  BL（带返回的跳转指令）、

  BX（带状态切换的跳转指令）、

  BLX（带返回和状态切换的跳转指令）

四条指令。

- （1）简单跳转指令B
- **指令格式**：B{**条件**} **目**标地址
- **指令功能**：**跳**转到目标地址处执行。
- **范例**：
- B LABEL ；**程序无条件跳**转到标号LABEL处执行
- B 0x1400 ；**跳**转到绝对地址0x1400处执行
- **利用B指令**实现循环：
- MOV R0,#10 ；**初始化循**环计数器
- LOOP …
- SUBS R0,#1 ；计数器减1，**并**设置条件码
- BNE LOOP ；**如果**计数器R0**不**为0，则继续循环
- … ；**否**则终止循环

- （2）带返回的跳转指令 BL
- **指令格式**：BL{**条件**}　**目**标地址
- **指令功能**：该指令在跳转之前会将PC**的当前**值保存到R14**中，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。该指令是实现自程序调用的基本手段。

- 范例：
- …
- BL SUBP ；子程序调用（PC→R14）
- … ；返回到这里
- …
- SUBP … ；子程序入口
- …
- MOV PC,R14 ；子程序返回

- （3）带状态切换的跳转指令BX
- **指令格式**：BX{**条件**} **目**标地址寄存器
- **指令功能**：**指令**执行时将目标地址寄存器的第**0位拷**贝到CPSR**的**T标志位（决**定程序是切**换到Thumb**指令**还是继续执行ARM**指令**），[31：1]**位移入**PC：
- **若目**标地址第**0位**为0，则处理器执行ARM**指令**，**若目**标地址第**0位**为1，则处**理器跳**转到Thumb**指令**执行。
- **范例**：
- BX R0 ；跳转到R0**指定地址**，**并根据**R0**的最低位切**换处理器状态

- （4）带返回和状态切换的跳转指令 BLX
- **指令格式**：BLX 标号 **或** BLX{**条件**} **目**标地址寄存器
- **指令功能**：**将下一条指令的地址拷**贝到**R14中**，转移到标号处或**目**标地址寄存器指定的位置；如果目标地址寄存器的第**0位**为**1或使用**标号，则程序切换到**Thumb状**态。

- 范例：
- CODE32 ；以下是ARM代码
- …
- BLX Tsub ；调用Thumb子程序
- …
- CODE16 ；开始Thumb代码
- Tsub … ；Thumb子程序
- BX R14 ；返回到ARM代码

- 3. 程序状态寄存器处理指令
- ARM 微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器和通用寄存器之间传送数据，程序状态寄存器访问指令包括MRS和MSR两条指令。

- （1）MRS 指令
- **指令格式**：MRS{**条件**} **通用寄存器**，**程序状**态寄存器（CPSR **或**SPSR）
- **指令功能**：MRS **指令用于将程序状**态寄存器的内容传送到通用寄存器中。注**意通用寄存器不能使用**R15。该指令一般用在以下两种情况：
- ①**当需要改**变程序状态寄存器的内容时，可用MRS **将程序状**态寄存器的内**容**读入通用寄存器，修改后再写回程序状态寄存器。
- ②**当在异常**处理或进程切换时，需要保存程序状态寄存器的值，可先用该**指令**读出程序状态寄存器的值，然后保存。

- 范例：
- MRS R0,CPSR ；传送CPSR 的内容到R0
- MRS R1,SPSR ；传送SPSR 的内容到R1

- （2）MSR 指令
- **指令格式**：MSR{**条件**} **程序状**态寄存器_<**域**>，**操作数**
- **指令功能**：MSR **指令用于将操作数的内容**传送到程序状态寄存器（CPSR **或** SPSR）**的特定域中。其中，操作数可以**为通用寄存器或立即数。<**域**>**用于**设置**程序状**态寄存器中需要操作的位，为可选项，32 **位的程序状**态寄存器可**分**为4 **个域（必需用小写字母表示）**：
- **位**[31：24]为条件标志位域，用f **表示**；
- **位**[23：16]为状态位域，用s **表示**；
- **位**[15：8]为扩展位域，用x **表示**；
- **位**[7：0]为控制位域，用c **表示**；

- 范例：
- MSR CPSR,R0 ；传送R0 的内容到CPSR
- MSR SPSR,R0 ；传送R0 的内容到SPSR
- MSR CPSR_c,R0 ；传送R0 的内容到CPSR，但仅仅修改CPSR 中的控制位域

4. 寄存器存取指令

- 寄存器存取指令完成寄存器和内存之间的数据传递。该指令分三种类型：单寄存器传送指令、多寄存器传送指令、交换指令。
- 单寄存器传送指令LDR/STR的作用是将单一的数据传入（LDR）或传出（STR）寄存器。对内存的访问可以是32位的字、16位有符号/无符号的半字或8位有符号/无符号的字节数据。
- 多寄存器传送指令LDM/STM可以用一个指令加载/存储多个寄存器的数据，大大提高存取效率。
- 交换指令SWP是一条寄存器和存储器内容交换的指令，可用于信号量操作等

- （1）单寄存器传送指令——LDR/STR
- 指令格式如下：
- LDR{条件}　Rd，<地址>　；把<地址>处一个字数据装入寄存器Rd
- STR{条件}　Rd，<地址>　；把寄存器Rd的一个字数据保存到<地址>中
- LDR{条件}{S}H Rd，<地址>　；把<地址>处半字数据装入寄存器Rd
- STR{条件}{S}H Rd，<地址>　；把寄存器Rd的半字数据保存到<地址>中
- LDR{条件}{S}B Rd，<地址>　　；把<地址>处一个字节数据装入寄存器Rd
- STR{条件}{S}B Rd，<地址>　　；把Rd的字节数据（即低8位）保存到<地址>中
- 其中选项{S}表示传送带符号的数据。

- LDR/STR**指令的**寻址是非常灵活的，通常由两部分组成，一部分为一个基址寄存器，可以为任何通用寄存器，另一部分是地址偏**移量，地址偏移量有三种形式**：

- ①立即数
- 立即数可以是一个无符号数，这个数可加到基址寄存器上，也可让基址寄存器值减去立即数。
- 范例：
- LDR R1, [R0,#0x08] ；将R0+0x08地址处数据传送给R1，R0不变。
- LDR R1, [R0，#-0x08] ；将R0-0x08]地址处的数据传送给R1，R0不变。
- LDR R1, [R0] ；将R0地址处的数据传送给R1（零偏移量）。
- LDR R1, [R0, #4]! ；首先R0=R0+4，即基址寄存器发生变化，然后将新的R0地址处的数据传送给R1。（注意：基址寄存器先改变，该寻址方式又称为前变址寻址）。
- LDR R1, [R0], #4 ；首先将R0地址处的数据传送给R1，然后改变R0=R0+4；（注意：基址寄存器后改变，该寻址方式又称后变址寻址）。

- ②寄存器
- 寄存器中的值可以加到基址寄存器中，也可以让基址寄存器的值减去偏移寄存器的值。
- 范例：
- LDR R1, [R0,R2] ；将R0+R2地址的数据传送到R1，R0不变。
- LDR R1, [R0,-R2] ；将R0-R2地址的数据传送到R1，R0不变。

- ③寄存器及移位常数
- 寄存器移位后的值可以加到基址寄存器上，也可以用基址寄存器减去这个值。
- 范例：
- LDR R1, [R0,R2 LSL #2] ；将R0+R2×4（即左移两位）地址处的数据传送到R1。
- LDR R1, [R0,-R2 LSL #2] ；将R0-R2×4（即左移两位）地址处的数据传送到R1。

- （2）多寄存器传送指令——LDM和STM
- 多寄存器传送指令LDM和STM可以实现多个寄存器和一块连续的内存单元之间的字数据传递。LDM指令用于加载寄存器，即将连续内存单元的数据加载到多个寄存器中；STM指令用于存储寄存器，即将多个寄存器的数据存储到连续的内存单元中。指令格式如下：
- LDM {条件}<模式> Rn{!},reglist{^}
- STM {条件}<模式> Rn{!},reglist{^}

- 其中<模式>有8种，如下所示（其中前4种用于数据块的传递，后四种用于堆栈操作）：
- IA　　每次传送数据后地址增加4（指向下一个字数据）。
- IB　　每次传送数据前地址增加4。
- DA　　每次传送数据后地址减少4（指向前一个字数据）。
- DB　　每次传送数据前地址增加4。
- FD　　满递减堆栈：堆栈指针指向最后压入的数据，而且压入数据时堆栈由高地址向低地址生成。
- ED　　空递减堆栈：堆栈指针指向下一个要压入的数据的空位置，而且压入数据时堆栈由高地址向低地址生成。
- FA　　满递增堆栈：堆栈指针指向最后压入的数据，而且压入数据时堆栈由低地址向高地址生成。
- EA　　空递增堆栈：堆栈指针指向下一个要压入的数据的空位置，而且压入数据时堆栈由低地址向高地址生成。

寻址方式	说明	pop	=LDM	push	=STM
FA	满递增	LDMFA	LDMDA	STMFA	STMIB
FD	满递减	LDMFD	LDMIA	STMFD	STMDB
EA	空递增	LDMEA	LDMDB	STMEA	STMIA
ED	空递减	LDMED	LDMIB	STMED	STMDA

- 其中Rn为基址寄存器，装有传送数据的初始地址，Rn不允许是R15；后缀{!}表示最后的地址写回到Rn中，即改变Rn的值。
- 其中reglist寄存器列表表示多个寄存器或寄存器范围，用“，”分隔，例如{R0, R2，R4-R6}，寄存器由小到大排列。后缀{^}不允许在用户模式或系统模式下使用，若在LDM指令且寄存器列表中包含PC（即R15）时使用，则除了正常的多寄存器数据传送外，还同时将SPSR数据传给CPSR，可用于异常处理返回；使用后缀{!}且寄存器列表不包含PC时，加载/存储的是用户模式的寄存器，而不是当前模式的寄存器。

- 范例：
- LDMIA R0！,{R1-R5} ；将R0地址处连续的字数据加载到R1-R5中，R0的值改变。
- STMIA R1！,{R3-R7} ；将R3-R7中的数据存储到R1所指的连续地址中，R1改变。
- STMFD SP！,{R0-R7,LR} ；现场保护，将{R0-R7，LR}数据入栈
- LDMFD SP！,{R0-R7,PC} ；现场恢复，从堆栈中加载数据到{R0-R7，PC}异常处理返回。

5. 协处理器指令

- ARM微处理器可支持多达 16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 处理器和其他协处理器的指令。
- ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作，以及在ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在 ARM 协处理器的寄存器和存储器之间传送数据。

- ARM 协处理器指令包括以下 5 条：
- 协处理器数操作指令——CDP
- 协处理器数据加载指令——LDC
- 协处理器数据存储指令——STC
- ARM 处理器寄存器到协处理器寄存器的数据传送指令——MCR
- 协处理器寄存器到ARM 处理器寄存器的数据传送指令——MRC

- （1）协处理器数操作指令——CDP
- CDP 指令用于ARM 处理器通知ARM 协处理器执行特定的操作,若协处理器不能成功完成特定的操作，则产生未定义指令异常。
- CDP 指令的格式为：
- CDP{条件} 协处理器编码，协处理器操作码1，目的寄存器，源寄存器1，源寄存器2，协处理器操作码2。
- 其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，目的寄存器和源寄存器均为协处理器的寄存器，指令不涉及ARM 处理器的寄存器和存储器。
- 范例：
- CDP P3 , 2 , C12 , C10 , C3 , 4 ；该指令完成协处理器 P3 的初始化

- （2）协处理器数据加载指令——LDC
- LDC 指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。
- LDC 指令的格式为：
- LDC{条件}{L} 协处理器编码,目的寄存器，[源寄存器]
- 其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。
- 范例：
- LDC P3 , C4 , [R0] ；将 ARM 处理器的寄存器 R0 所指向的存储器中的字数据传送到协处理器 P3 的寄存器 C4 中。

- （3）协处理器数据存储指令——STC
- STC 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。
- STC 指令的格式为：
- STC{条件}{L} 协处理器编码,源寄存器，[目的寄存器]
- 其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。
- 范例：
- STC P3 , C4 , [R0] ；将协处理器 P3 的寄存器 C4 中的字数据传送到 ARM 处理器的寄存器R0 所指向的存储器中。

- （4）ARM 处理器寄存器到协处理器寄存器的数据传送指令——MCR
- MCR 指令用于将ARM 处理器寄存器中的数据传送到协处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。
- MCR 指令的格式为：
- MCR{条件} 协处理器编码，协处理器操作码1，源寄存器，目的寄存器1，目的寄存器2，协处理器操作码2。
- 其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，源寄存器为ARM 处理器的寄存器，目的寄存器1 和目的寄存器2 均为协处理器的寄存器。
- 范例：
- MCR P3,3,R0,C4,C5,6；该指令将 ARM 处理器寄存器 R0 中的数据传送到协处理器 P3 的寄存器 C4 和 C5 中。

- （5）协处理器寄存器到ARM 处理器寄存器的数据传送指令——MRC
- MRC 指令用于将协处理器寄存器中的数据传送到ARM 处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。
- MRC 指令的格式为：
- MRC{条件} 协处理器编码，协处理器操作码1，目的寄存器，源寄存器1，源寄存器2，协处理器操作码2。
- 其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，目的寄存器为ARM 处理器的寄存器，源寄存器1 和源寄存器2 均为协处理器的寄存器。
- 范例：
- MRC P3,3,R0,C4,C5,6；该指令将协处理器 P3 的寄存器C4、C5中的数据传送到 ARM 处理器寄存器中R0。

3.1.4 Thumb指令集介绍

任务： 了解Thumb指令集与ARM指令集区别, 了解Thumb状态寄存器与ARM状态寄存器关系。

- 为兼容数据总线宽度为16位的应用系统，ARM体系结构除了支持执行效率很高的32位ARM指令集以外，同时支持16位的Thumb指令集。Thumb指令集是ARM指令集的一个子集，是针对代码密度问题而提出的，它具有16位的代码宽度。与等价的32位代码相比较，Thumb指令集在保留32位代码优势的同时，大大的节省了系统的存储空间。Thumb不是一个完整的体系结构，不能指望处理器只执行Thumb指令集而不支持ARM指令集。

- 当处理器在执行ARM程序段时，称ARM处理器处于ARM工作状态，
- 当处理器在执行Thumb程序段时，称ARM处理器处于Thumb工作状态。

- 在一般的情况下，Thumb指令与ARM指令的时间效率和空间效率关系为：
- Thumb代码所需的存储空间约为ARM代码的60%～70%。
- Thumb代码使用的指令数比ARM代码多约30%～40%。
- 若使用32位的存储器，ARM代码比Thumb代码快约40%。
- 若使用16位的存储器，Thumb代码比ARM代码快约40%～50%。
- 与ARM代码相比较，使用Thumb代码，存储器的功耗会降低约30%。

- Thumb指令集与ARM指令集在以下几个方面有区别：
- （1）跳转指令。条件跳转在范围上有更多的限制，转向子程序只具有无条件转移。
- （2）数据处理指令。对通用寄存器进行操作，操作结果需放入其中一个操作数寄存器，而不是第三个寄存器。
- （3）单寄存器加载和存储指令。Thumb状态下，单寄存器加载和存储指令只能访问R0～R7。
- （4）批量寄存器加载和存储指令。LDM和STM指令可以将任何范围为R0～R7的寄存器子集加载或存储，PUSH和POP指令使用堆栈指针R13作为基址实现满递减堆栈，除R0～R7外，PUSH指令还可以存储链接寄存器R14，并且POP指令可以加载程序指令PC。
- （5）Thumb指令集没有包含进行异常处理时需要的一些指令，因此，在异常中断时还是需要使用ARM指令。这种限制决定了Thumb指令不能单独使用需要与ARM指令配合使用。
- Thumb 状态寄存器集是ARM 状态寄存器集的子集，程序员可直接访问8 个通用寄存器R0～R7、PC、堆栈指针SP、链接寄存器LR和CPSR。每个特权模式都有分组的SP、LR和SPSR。

图3-1 Thumb寄存器在ARM寄存器上的映射

- Thumb状态寄存器与ARM状态寄存器有如下关系：
- Thumb状态R0～R7与ARM状态R0～R7相同
- Thumb状态CPSR和SPSR与ARM状态CPSR和SPSR 相同
- Thumb状态SP映射到ARM状态R13
- Thumb状态LR映射到ARM状态R14
- Thumb状态PC映射到ARM状态PC（R15）
- 程序中，可使用BX指令实现ARM状态和Thumb状态的切换。

3.2 ARM汇编语言和汇编程序规范

- 问题：
  - ARM指令分为哪几类？
  - 指令条件码是什么，主要有哪些？
  - ARM寻址方式有哪些？
  - Thumb指令有哪些？
- 重点：
  - ARM指令及其寻址方式。
- 内容：
  - ARM指令系统
  - ARM寻址方式
  - ARM指令集
  - Thumb指令集。

3.2.1 ARM汇编语言语句格式

任务：了解ARM汇编语言源程序的组成，掌握. ARM汇编语句格式。

1. ARM汇编语言源程序的组成

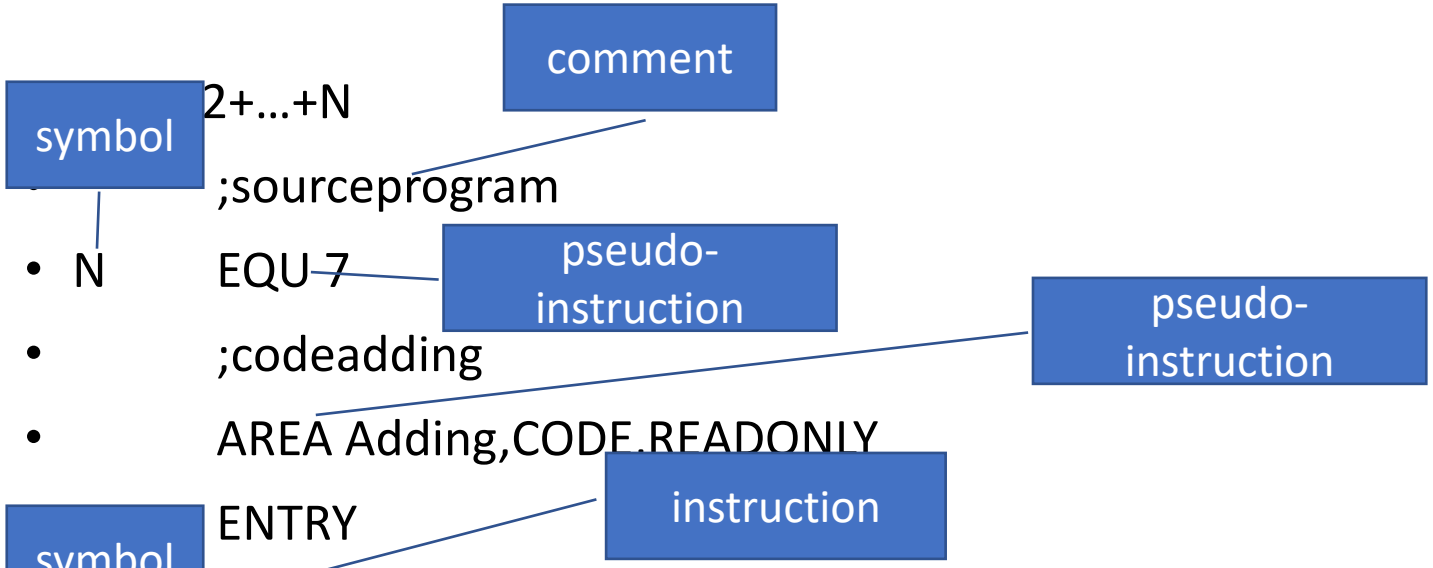
- 汇编语言源程序由若干语句组成，通常，这些语句可以分为3类：
- 指令语句
  - 汇编指令使用助记符表示的机器指令，所以这类语句又称及其指令语句，它们由汇编程序汇编成相应的能被CPU直接识别并执行的目标代码，也称机器代码。
- 宏指令语句
  - 在汇编语言中，允许用户为多次重复使用的程序段命一个名字，然后就可以在程序中用这个名字代替该程序段，我们将定义的过程称为宏定义，该段程序段称为宏，只要在相应的位置使用宏名就相当于使用了这段程序。因此，宏指令语句就是宏的引用。
- 伪指令语句
  - 在 ARM 汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令，它们所完成的操作称为伪操作。

2. ARM汇编语句的格式

- ARM汇编语言语句格式如下所示：
- {symbol}{instruction | directive | pseudo-instruction} {;comment}
- 其中：
- instruction为指令。在ARM汇编语言中，**指令不能从一行的行头开始**。在一行语句中，指令的前面必须有空格或者符号。
- directive是指示符。
- pseudo-instruction是伪指令。

2. ARM汇编语句的格式

- ARM汇编语言语句格式如下所示：
- {symbol}{instruction | directive | pseudo-instruction} {;comment}
- **其中**：
- symbol为符号。在ARM汇编语言中，**符号必须从一行的行头开始，并且符号中不能包含空格。在指令和**伪指令中符号用作地址标号(label)；**在有些指示符中，符号用作**变量或常量。
- comment为语句的注释。在ARM汇编语言中注释以分号**“;”开**头。**注**释的结尾即为一行的结尾。注释也可以单独占用一行。



- LDR R2,=RESULT
- STR R0,[R2]
- HERE B HERE
- ;Dataadding
- AREA Dataspace,DATA,READWRITE
- RESULT DCB 0
- END

3.2.2 ARM汇编器的伪操作

任务：了解什么是ARM汇编语言伪操作，掌握ARM汇编器的各种伪操作的使用方法。

1.结构伪操作

- （1）AREA
- 语法格式：AREA 段名 属性1，属性2，……
- 功能说明：用于定义一个代码段或数据段。其中，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：

- CODE属性：用于定义代码段，默认为READONLY 。
- DATA属性：用于定义数据段，默认为READWRITE 。
- READONLY属性：指定本段为只读，代码段默认为READONLY 。
- READWRITE属性：指定本段为可读可写，数据段的默认属性为 READWRITE 。
- 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， 相应的对齐方式为 2 表式次方。
- 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。
- 范例：
- AREA Init,CODE,READONLY

- （2）ALIGN
- 语法格式：ALIGN { 表达式 { ， 偏移量 }}
- 功能说明：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，则当前位置的对齐方式为：2的表达式次幂 + 偏移量。

- 范例：
- AREA Init,CODE,READONLY,ALIEN＝3 ；
- 指令序列
- END

AREA test1,DATA　　;假设段基地址为0x00000100

STR1=”A”

AREA TEST2,DATA；默认为2，段基址为0x00000104

STR2=”B”,1,2,3,4,5

STR3=

我们可以简单理解为,使用ALIGN这个属性可以让我们给上一个段预留除一部分缓冲区域,以ALIGN=2为例,当上一个段中的数据超过4个字节时,当前段基地器会向后再偏移4个字节,避免数据被覆盖,也就是说内存数据位置会进行重新分布,那么我们可以通过这个值来设置内存数据刷新频率,**值越低,内存利用率越高,但是内存刷新频率也越高,负荷加重,反之,内存浪费越大,但是内存数据不需要频繁重新分布**

- （3）CODE16、CODE32
- 语法格式：CODE16（或CODE32 ）
- 功能说明：
- CODE16伪操作通知编译器，其后的指令序列为16位的Thumb指令。
- CODE32伪操作通知编译器，其后的指令序列为32位的ARM指令。

- 范例：
- AREA Init ,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
- **功能**说明：ENTRY伪操作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**）。

- 范例：
- AREA Init,CODE,READONLY
- ENTRY ;指定应用程序的入口点
- ……

- （5）END
- 语法格式：END
- **功能**说明：END伪指令用于通知编译器已经到了源程序的结尾。
- 【例】
- AREA　Init，　CODE，READONLY
- ……
- END　　；**指定**应用程序的结尾

2. （数据定义Data Definion）伪操作

- DCB
- DCW（DCWU）
- DCD（DCDU）
- DCFD（DCFDU）
- DCFS（DCFSU）
- DCQ（DCQU）
- SPACE
- MAP
- FIELD

2. （数据定义Data Definion）伪操作

- （1）DCB
- 语法格式：标号 DCB 表达式
- 功能说明：DCB伪操作用于分配一片连续的字节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。
- 范例：
- Str DCB "This is a test" ;分配一片连续的字节存储单元并初始化
- Nullstring DCB “Null String”,0 ; 构造一个以0结尾的字符串

- （2）DCW（或DCWU）
- 语法格式：标号 DCW （或 DCWU） 表达式
- 功能说明：DCW（或DCWU）伪操作用于分配一片连续的半字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。
- 范例：
- DataTest DCW 1,2,3 ;分配一片连续的半字存储单元并初始化
- Data1 DCW -128,num1+8; num1必须是已经定义过的

- （3）DCD（或DCDU）
- 语法格式：标号DCD（或DCDU） 表达式
- 功能说明：DCD（或 DCDU）伪操作用于分配一片连续的字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。
- 范例：
- DataTest DCD 4,5,6 ;分配一片连续的字存储单元并初始化
- Data DCD memadrr+4;分配一个字单元，其值为程序中标号memaddr加4个字节

```
AREA Buf, DATA, READWRITE
Array       DCD 0x11,0x22,0x33,0x44
            DCD     0x55,0X66,0X77,0X88
            DCD     0x00,0x00,0x00,0x00
        AREA Adding, CODE, READONLY
        ENTRY
            LDR     R0, =Array
            LDR     R2,[R0]
            MOV     R1,#4
            LDR     R3,[R0,R1,LSL #2]
            ADD     R2,R2,R3
            MOV     R1,#8
            STR     R2,[R0,R1,LSL #2]
HERE    B       HERE
        END
```

- （4）DCFD（或DCFDU）
- 语法格式：标号 DCFD（或DCFDU） 表达式
- 功能说明：DCFD（或DCFDU）伪操作用于为双精度的浮点数分配一片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。
- 范例：
- FDataTest DCFD 2E115, -5E7 ;分配一片连续的字存储单元并初始化为指定的双精度数

- （5）DCFS（或DCFSU）
- 语法格式：标号 DCFS （或DCFSU） 表达式
- 功能说明：DCFS（或DCFSU）伪操作用于为单精度的浮点数分配一片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。
- 范例：
- FDataTest DCFS 2E5,-5E7 ;分配一片连续的字存储单元并初始化为指定的单精度数。

- （6）DCQ(或DCQU）
- 语法格式：标号 DCQ （或DCQU） 表达式
- 功能说明：DCQ （或 DCQU ）伪操作用于分配一片以8个字节为单位的连续存储区域并用伪操作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。
- 范例：
- DataTest DCQ 100 ;分配一片连续的存储单元并初始化为指定的值

- （7）SPACE
- 语法格式：标号 SPACE 表达式
- 功能说明：SPACE 伪操作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。
- 范例：
- DataSpace SPACE 100 ;分配连续 100 字节的存储单元并初始化为0

- （8）MAP
- 语法格式：MAP **表达式** {， **基址寄存器** }
- **功能**说明：MAP 伪指令用于定义一个结构化的内存表的首地址。**MAP也可用“^”代替**。
- **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**存表首地址**为表达式值与基址寄存器的和。
- MAP 伪操作通常与FIELD伪操作配合使用来定义结构化的内存表。
- **范例**：
- MAP 0x100,R0 ;**定**义结构化内存表首地址的值为 0x100 ＋ R0 。

Datastruc SPACE 280

MAP Datastruc

分配280个字节单元，内存表的首地址为Datastruc内在块

- （9）FILED
- 语法格式：标号 FIELD **表达式**
- **功能**说明：FIELD伪操作用于定义一个结构化内存表中的数据域。 FILED**也可用“#” 代替**。
- **表达式的**值为当前数据域在内存表中所占的字节数。
- FIELD伪操作常与MAP伪操作配合使用来定义结构化的内存表。MAP伪指令定义内存表首地址，FIELD伪操作定义内存表中各个数据域，并可为每个数据**域指定一个**标号供其他指令引用。
- **注意**MAP**和**FIELD伪操作仅用于定义数据结构，并不实际分配存储单元。

- 范例：
- 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，位置为 0x100
- B FIELD 32 ;定义 B 的长度为 32 字节，位置为 0x110
- S FIELD 256 ;定义 S 的长度为 256 字节，位置为 0x130

```
Datastruc SPACE 280
MAP Datastruc
consta FIELD 4
constab FIELD 4
x FIELD 8
y FIELD 8
string FIELD 256
LDR R6,    [R9,consta]
```

;分配 280 个字节单元

；内存表的首地址为 Datastruc内存块

；字段 consta 长度 4 字节，相对地址 0

； 字段 constab 长度 4 字节， 相对地址 4

； 字段 x 长度 8 字节， 相对地址8

； 字段 y 长度 8 字节， 相对地址16

； 字段 string 长度 256 字节， 相对地址 24

; 引用内存表中的数据域

3. 符号定义伪操作

- 符号定义伪操作用于定义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。常见的符号定义伪操作有如下几种：
  - 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**
- **功能**说明：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 伪操作用于定义一个全局的**数字**变量，**并初始化**为 0 ；
- GBLL 伪操作用于定义一个全局的逻辑变量，**并初始化**为 F （假）；
- GBLS 伪操作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**伪操作用于定义全局变量，因此在整个程序范围内变量名必须**唯一**。

- 范例：
- 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**
- **功能**说明：LCLA 、 LCLL **和** LCLS 伪操作用于定义一个 ARM **程序中的局部**变量，**并将其初始化。其中**：
- LCLA 伪操作用于定义一个局部的**数字**变量，**并初始化**为 0 ；
- LCLL 伪操作用于定义一个局部的逻辑变量，**并初始化**为 F （**假**） ；
- LCLS 伪操作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。

- GBLA count
- count SETA 2
- AREA exam, CODE,READONLY
- CODE32
- ENTRY
- start
- MOV R0,#count
- ADD R0,R0,#2
- B start
- END
- 在赋值过程中，全局变量名必须 顶格写，全局变量常在代码段外定义和赋值

- 范例：
- LCLA Test4 ;声明一个局部的数字变量，变量名为 Test4
- Test4 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
- Test5 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 “Tesing ”

```
    MACRO
    MOV_START
            LCLA    x
            LCLA    y
x   SETA    12
y   SETA    24
    MOV             R0,#x
    MOV     R1,#y
    ADD             R0,R0,R1
    MEND
    AREA lc,        CODE,    READONLY
    ENTRY
start
    MOV_START
    MOV     R0,#0
over            B           over
    END
```

局部变量可以在宏定义内使用

- （3）SETA、SETL、SETS
- 语法格式：变量名 SETA（SETL或SETS） 表达式
- 功能说明：伪操作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 伪操作用于给一个数学变量赋值；
- SETL 伪操作用于给一个逻辑变量赋值；
- SETS 伪操作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。

- （4）RLIST
- 语法格式：名称 RLIST { 寄存器列表 }
- 功能说明：RLIST 伪操作可用于对一个通用寄存器列表定义名称，使用该伪操作定义的名称可在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。
- 范例：
- RegList RLIST {R0-R5, R8,R10} ;将寄存器列表名称定义为 RegList

3.2.2 ARM汇编器的伪操作

任务：了解什么是ARM汇编语言伪操作，掌握ARM汇编器的各种伪操作的使用方法。

5. 其他常用的伪操作

- EQU
- EXPORT （或 GLOBAL）
- IMPORT
- EXTERN
- GET （或 INCLUDE）
- INCBIN
- RN
- ROUT

- （6）EQU
- 语法格式：名称 EQU **表达式**{，类型}
- **功能**说明：EQU伪操作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**。
- **名称**为EQU伪操作定义的字符名称，当表达式为**32位的常量**时，**可以指定表达式的数据**类型，可以有三种类型：CODE16、CODE32和DATA。

- 范例：
- Test EQU 50 ;定义标号 Test 的值为 50
- Addr EQU 0x55 ,CODE32 ;定义 Addr 的值为 0x55， 且该处为 32 位的 ARM 指令。

- （7）EXPORT（或GLOBAL）
- 语法格式：EXPORT 标号 {[WEAK]}
- **功能**说明：EXPORT伪操作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。

- 范例：
- AREA Init,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]}
- **功能**说明：IMPORT伪操作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。

- 指令置为NOP操作。
- 范例：
- AREA Init,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]}
- **功能**说明：EXTERN伪操作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。

- 范例：
- AREA Init,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
- 功能说明：GET伪操作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。

- 范例：
- AREA Init, CODE,READONLY
- GET a1.s　　;通知编译器当前源文件包含源文件a1.s
- GET C:\a2.s　　;通知编译器当前源文件包含源文件C:\ a2.s
- END

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

- （11）INCBIN
- 语法格式：INCBIN **文件名**
- 语法格式：INCBIN伪操作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**，编译器从其后开始继续处理。

- 范例：
- AREA Init,CODE,READONLY
- INCBIN a1.dat　;通知编译器当前源文件包含文件貊1.dat
- INCBIN C:\a2.txt　;通知编译器当前源文件包含文件C:\a2.txt……
- END

- （12）RN
- 语法格式：名称 RN 表达式
- 功能说明：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。
- 范例：
- Temp RN R0　;将R0 定义一个别名Temp

- （13）ROUT
- 语法格式：{ 名称 } ROUT
- 功能说明：ROUT伪操作用于给一个局部变量定义作用范围。在程序中未用该伪指令时，局部变量作用范围为所在AREA，而用ROUT后，局部变量作用范围为当前ROUT和下一 ROUT之间。

3.2.2 ARM汇编器的伪操作

任务：了解什么是ARM汇编语言伪操作，掌握ARM汇编器的各种伪操作的使用方法。

4. 汇编控制（ Assembly Control ）伪操作

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

- （1）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
- 功能说明：IF、ELSE、ENDIF伪操作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伪操作可以嵌套使用。

- 范例：
- 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

```
        AREA Example,CODE,READONLY
        CODE32
Data_in EQU 100 ;定义标号 Data_in 的值为 100 在 ENTRY 入口之前
        GBLA count ;定义全局变量
count SETA 20
        ENTRY
Start
        IF count < Data_in ;条件编译
                MOV R0,#3
        ELSE
                MOV R1,#24
        ENDIF
        MOV R1,#12
        ADD R0,R0,R1
        END
```

- （2）WHILE、WEND
- 语法格式：
- WHILE 逻辑表达式
- **指令序列**
- WEND
- **功能**说明：WHILE、WEND伪操作能根据条件的成**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伪操作可以嵌套使用。

- 范例：
- 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）MACRO、MEND
- 语法格式：
- MACRO
- {$标号} 宏名 {$参数1，$参数2，……}
- 指令序列
- MEND
- 功能说明：MACRO、MEND伪操作可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。

例：

在ARM中完成测试—跳转操作需要两条指令，定义一条宏指令完成测试—跳转操作。

```
        MACRO
$label    TestAndBranch      $dest,$reg,$cc
$label          CMP      $reg,    #0
                B$cc   $dest
        MEND
```

在程序中调用该宏

```
Test    TestAndBranch    NonZero,    R0,    NE
        ……
        ……
NonZero
```

；程序被汇编后，宏展开的结果

```
Test    CMP    R0,#0
        BNE    NonZero
         ……
         ……
NonZero
```

```
    MACRO
    ADD_START    ;宏名
    LCLA sum      ;定义局部变量
    LCLA n
sum SETA 0       ;给变量赋值
n  SETA 100
    MOV     R1,#n
add
    MOV     R2,R1
    ADD     R3,R3,R2
     SUBS    R1,R1,#1
     BNE     add
     LDR     R1,=sum
     STR     R3,[R1]    ;累加和写入sum单元
     MEND
```

```
AREA Example,CODE,READONLY
ENTRY
            CODE32
start
            MOV     R2,#0
             MOV     R3,#0          ;R3存放累加和，初始值为0
             ADD_START
over
            END
```

```
  MACRO
$label sum3          $a,     $b

$label.add
        ADDS $a,$a,#1
        ADDS $b,$b,$a
        CMP  $a,#100
        BNE  $label.add
        MOV R2,$b
  MEND
```

```
        AREA  add, CODE, READONLY
        ENTRY
        CODE32
START
        MOV  R0, #1
        MOV  R1, #1
Hong  sum3 R0,R1
STOP
        MOV  R0, #0X18

        LDR  R1, =0X20026
        SWI  0X123456
        END
```

- （4）MEXIT
- 语法格式：MEXIT
- **功能**说明：用于从宏定义中跳转出去。

- 范例：
- MACRO
- $abc macroabc $param1,$param2
- WHILE condition1
- IF condition2
- MEXIT ;从宏中跳转出去
- ELSE
- ENDIF
- WEND

4. ARM汇编语言子程序调用

- 在ARM汇编语言程序中，子程序的调用一般是通过BL指令来实现的。在程序中，使用指令：
- BL 子程序名
- 该指令在执行时完成操作：将子程序的返回地址存放在连接寄存器LR中，同时将程序计器PC指向子程序的入口点，当子程序执行完毕需要返回调用处时，只需要将存放在LR中的返回地址重新拷贝给程序计数器PC。在调用子程序的同时，也可以完成参数的传递和从子程序回运算的结果，通常可以使用寄存器R0～R3完成。

主程序在调用子程序时，往往需要向子程序传递一些参数， 同样，子程序在运行完毕后也可能要把结果传回给调用程序。

1.用寄存器传递参数

2.存储区域传递参数

3.堆栈传递参数

用子程序和宏实现1+2+……+N

```
    AREA MAIN, CODE, READONLY
    ENTRY
    CODE32
start
    MOV     R2,#0
    MOV     R3,#0
    LDR     R1,=n
    LDR     R1,[R1]
    BL      add
stop
    MOV     R0,#0x18
     LDR    R1,=0x20026
     SWI    0x123456
```

```
add
            MOV     R2,R1
            ADD     R3,R3,R2
            SUBS    R1,R1,#1
            BNE     add
            LDR     R1,=dst
            STR     R3,[R1]        ;累加和写入dst
单元
            MOV   PC,LR            ;子程序返回

            AREA  NUM,DATA,READWRITE
n    DCD     100
dst  DCD     0
            END
```

说明

- 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。

3.3.1ARM汇编语言程序中常用的符号

任务：掌握ARM汇编语言中常用符号的定义规则，掌握程序中的变量、常量、变量替换方法、标号和局部标号的使用方法。

- 在ARM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，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：
- 符号由大小写字母、数字和下划线组成
- 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，可使用GBLA、GBLL、GBLS伪操作**声明全局**变量，使用 LCLA、LCLL、LCLS 伪操作声明局部变量，并可使用SETA、SETL和SETS 对其进行初始化。

2. 程序中的常量

- **程序中的常量是指其指在程序的运行**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 ～ 232 -1，**当作**为有符号数时，其取值范围为 -231 ～ 231 -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。

- 3. 程序中的变量替换
- 程序中的变量可通过替换操作取得一个常量。替换操作符为“$”。
- 如果在数字变量前面有一个替换操作符“$”，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替换“$”后的数字变量。
- 如果在逻辑变量前面有一个替换操作符“$”，编译器会将该逻辑变量替换为它的取值（真或假）。
- 如果在字符串变量前面有一个替换操作符“$”，编译器会将该字符串变量的值替换“$”后的字符串变量。

3. 程序中的变量替换

- 程序中的变量可通过替换操作取得一个常量。替换操作符为“$”。
- 如果在数字变量前面有一个替换操作符“$”，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替换“$”后的数字变量。
- 如果在逻辑变量前面有一个替换操作符“$”，编译器会将该逻辑变量替换为它的取值（真或假）。
- 如果在字符串变量前面有一个替换操作符“$”，编译器会将该字符串变量的值替换“$”后的字符串变量。

- 范例：
- LCLS S1 ;定义局部字符串变量S1和S2
- LCLS S2
- S1 SETS "Test!"
- S2 SETS "This is a $S1" ;字符串变量S2 的值为 “This is a Test！”
- 如果程序中需要字符$,则用$$来表示，编译器将不进行变量替换，而是将$$当作$。

4. 标号

- 标号是表示程序中的指令或者数据地址的符号。根据标号的生成方式有以下**3种**：
- （1）**基于PC的**标号
- **基于PC的**标号是位于指令前或者程序中数据定义伪操作前的标号。这种标号在汇编时将被处理成PC值加上（或减去）一个数字常量。它常用于表示跳转指令的目标地址，或者代码**段中所嵌入的少量数据**。
- （2）**基于寄存器的**标号
- **基于寄存器的**标号通常用MAP**和**FILED伪操作定义，也可以用EQU伪操作定义，这种标号在汇编时将被处理成寄存器的值加上（或减去）一个数字常量，它常用于访问位于数据段中**的数据**。
- （3）绝对地址
- 绝对地址是一个**32位的数字量，它可以**寻址的范围为0~232-1，**即直接可以**寻址整个内存空间。

5. 局部标号

- 局部标号主要在局部范围内使用，它由两部分组成：开头是一个0~99之间的数字，后面紧接一个通常表示该局部变量作用范围的符号。
- 局部变量的作用范围通常为当前段，也可用伪操作ROUT来定义局部变量的作用范围。

- **局部**变量定义的语法格式如下：
- N（routname）
- **其中**：
- N为0~99**之**间的数字
- routname为符号，通常为该变量作用范围的名称（用ROUT伪操作定义的）
- **局部**变量应用的语法格式如下：
- %{F|B}{A|T} N{routname}
- **其中**：
- N为局部变量的数字号
- routname为当前作用范围的名称（用ROUT伪操作定义的）
- %**表示引用操作**
- F**指示**编译器只向前搜索
- B**指示**编译器只向后搜索
- A**指示**编译器搜索宏的所有嵌套层次
- T**只是**编译器搜索宏的当前层次

3.3.2汇编语言程序中的表达式和运算符

任务：掌握ARM汇编语言中各种表达式和运算符的使用方法。

- 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。

- 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。

- 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。

- 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。

- （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。

- 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。

- ①LEN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
- ②CHR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

- ③STR运算符
- STR 运算符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。

- ④LEFT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。

- ⑤RIGHT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。

- ⑥CC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
- ①BASE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。
- ②INDEX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，例如：
- ?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。
- ②DEF运算符
- DEF运算符判断是否定义某个符号，例如：
- ：DEF：X
- 如果符号X已经定义，则结果为真，否则为假。

3.4嵌入式C语言

引言

- 在ARM程序的开发中，系统初始化部分为缩短执行时间和提高执行效率，通常采用汇编语言来编写，包括ARM的启动代码，ARM操作系统的移植代码等。

```
IMPORT Main
AREA Init,CODE,READONLY
ENTRY
    LDR R0,=0x3FE0000;
    LDR R1,=0xe7FFFF80
    STR  R1,[R0]
    LDR SP,0X3FE1000
    BL Main
END
```

引言

- 在ARM程序的开发中，系统初始化部分为缩短执行时间和提高执行效率，通常采用汇编语言来编写，包括ARM的启动代码，ARM操作系统的移植代码等。
- 在应用程序开发上，绝大多数代码都是使用C语言来完成 的。
- ARM的开发环境实际上就是嵌入了一个C语言的集成开发环境，只不过这个嵌入式C语言开发环境和ARM的硬件紧密相关。

引言

- 在使用C语言时，有时要用到和汇编的混合编程，当汇编代码较为简洁，可使用直接 内嵌汇编的方法。
- 通常需要将汇编程序以文件的形式加入项目当中，并通过ARM体系结构程序调用标准（Procedure Call Standard for the ARM Architecture，简称AAPCS）的规定与C程序相互调用与访问。

3.4.1 AAPCS规则

- AAPCS是ARM架构下应用程序例程调用的二进制接口规范，其前身为著名的ATPCS（ARM/Thumb Procedure Call Standard）标准。
- 这个规范的目的是使不同程序模块可以分别编译，在二进制代码层面上直接配合使用。
- 目前ARM编译器均采用了这一标准，在ARM架构进行调试、逆向工程、漏洞分析、病毒分析时，十分需要熟悉这些调用规范。

- 为了使单独编译的C语言程序和汇编程序之间能够互相调用，必须为子程序之间的调用规定一定的规则。
- 基本AAPCS规定了在子程序调用时的一些基本规则，包括下面3方面的内容：
- 1.寄存器的使用其相应的名称。
- 2.数据栈的使用规则。
- 3.参数 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传递参数。寄存器R0-R3可记作A1-A4。被调用的子程序在返回前无须恢复寄存器R0-R3的内容。
- （2）在子程序中，ARM状态下，用使用寄存器R4-R11来保存局部变量，寄存器R4-R11可记作V1-V8。Thumb状态下只能使用R4-R7。
- （3）寄存器R12用作子程序间调用时临时保存栈指针，函数时使用该寄存器进行出栈，记作IP；在子程序间的链接代码中常有这种使用规则。
- （4）通用寄存器R13用作数据栈指针，记作SP。
- （5）通用寄存器R14用作链接寄存器。
- （6）通用寄存器R15用作程序计数器，记作PC。

2.堆栈的使用规则

- AAPCS规定堆栈为FD类型，即满递减堆栈，并且堆栈的操作是8字节对齐。
- 在进行出栈和入栈操作，则必须使用 LDMFD 和 STMFD 指令（或 LDMIA 和 STMDB)
- 而对于汇编程序来说，如果目标文件中包含了外部调用，则必须满足以下条件：
- （1）外部接口的数据栈一定是8位对齐的。也就是要保证在进入该汇编代码后，直到该汇编程序调用外部代码之间，数据栈的栈指针变化为偶数个字；
- （2）在汇编程序中使用PRESERVE8伪操作告诉连接器，本汇编程序是8字节对齐的。

3.参数传递规则

- 对于参数个数可变的子程序，当参数小于等于 4，用 R0-R3 保存参数，参数多于 4时，还可以使用数据栈来传递参数。
- 在参数传递时，将所有参数看作是存入在连续的内存单元中的字数据。然后，依次将各字数据传送到寄存R0，R1，R2，R3.如里参数多于4个，将剩余的字数据传送到数据栈中。
- 入栈的顺序与参数顺序相反，即最后一个字数据先入栈。
- 函数返回：结果为 32 位整数，通过 R0 返回
- 结果为 64 位整数，通过 R0，R1 返回
- 对于位数更多的结果，通过内存传递

ARM保有的特定关键字

- ARM编译器支持一些ANSI C进行扩展的关键词，这些关键词用于声明变量、声明函数、对特定的数据类型进行一定的限制。
- **用于声明函数的关**键词（双下划线起头）
  - __asm，内嵌汇编
  - __inline，内联展开
  - __irq，声明IRQ或FIQ的ISR
  - __pure，函数不修改该函数之外的数据
  - __softfp，使用软件的浮点连接件
  - __swi，软中断函数
  - __swi_indirect，软中断函数

ARM保有的特定关键字

- 用于声明变量的关键词
  - register
    - 声明一个变量，告诉编译器尽量保存到寄存器中。
  - _int64
    - 该关键词是long long的同义词。
  - _global_reg
    - 将一个已经声明的变量分配到一个全局的整数寄存器中。

4.栈的限制检查

- 如果在程序设计期间能够准确地计算出程序所需的内存总量，就不需要进行数据栈的检查，但是在通常情况下这是很难做到的，这时需要进行数据栈的检查。
- 在进行数据栈的检查时，使用寄存R10作为数据栈限制指针，这里寄存器R10又记作SL，用户在程序中不能控制该寄存器。
- 在已经占有栈的最低地址和SL之间必须有256字节的空间。也就是说，SL所指的内存地址必须比已经占用的栈的最低地址低256个字节。
- 当中断处理程序使用用户的数据栈时，在已经占用的栈的最低地址和SL之间除了必须保留的256个字节的内存单元外，还必须为中断处理预留足够的内存空间。
- 用户在程序中不能修改SL的值 ；数据栈栈指针SP的值必须不小SL的值。

3.4.2 嵌入式C语言编写特点

1.嵌入式C语言的数据存储方法：

- 嵌入式C语言是被编译器先翻译成汇编指令集，然后再将指令集转换为二进制指令代码，这些二进制代码与微处理器指令长度是一致的。
- 微处理器的存储空间有Flash ROM和RAM之分，一般会将常量、常数等存放在Flash ROM中，不常被修改。而把变量、函数堆栈等都存在RAM中，以方便在运行时改变 。
- 寄存器是微处理器中被频繁使用的存储空间，常用来存放做计算的操作数。因此，一个程序被编译成二进制后，分别保存在不同的地方。

嵌入式C语言的编写注意事项

- 1.数据类型尽量考虑整型
- 2.计算符号多加减、少乘，不除

  尽量采用移位运算代替乘除法。能显著提高运算效率
- 3.变量尽量用局部变量，少使用全局变量。

  全局变量全部存入在静态存储区，在程序开始执行时给全局分配存储区，程序执行完毕才释放。全局变量存在于RAM中，而局部变量存在于堆栈中。

  如果定义的全局变量太多就有可能导致溢出

嵌入式C语言的编写注意事项

- 4.注意变量名字
- 全局变量可以使用，但是使用时应注意尽可能使名字易于理解，且不能太短，避免名字空间的污染；避免使用巨大对象的全局变量。
- 局部变量可以和全局变量重名，但是局部变量会屏蔽全局变量。在函数内引用这个变量时，会用到同名的局部变量。
- 全局变量便于传递参数，数据能在整个程序中共享，可以不使用较麻烦的传递参数 方式，也省去了传递参数 的时间，会减少程序的运行时间。

嵌入式C语言的编写注意事项

- 5.子程序嵌套越少越好。

  在子程序运行时，当调用下一个子程序时就需要开辟一个堆栈空间，直到执行到return语句，才会把这个栈去掉。

- 6.循环体多用do-while，少用for、while等结构

  从编译的代码可以看出，do-while循环只需要一个空间放退出的地址。相比较而言，do-while循环所需要的额外开销最少。

3.5C语言与汇编语言混编规范

- 在ARM程序设计中，如果所有的编程任务均用汇编语言来完成，其工作量不仅相当巨大，而且不利于系统升级维护和软件的移植。
- 使用C语言开发的ARM程序可读性和可移植性好，开发周期短，程序修改方便。但C代码的效率还是无法与汇编代码的效率相比，而且一些硬件控制功能也不如汇编语言方便，甚至有些操作C语言无法直接实现
- 因此，ARM体系结构支持汇编语言与C语言的混合编程，在一个完整的程序中，除了初始化部分用汇编语言完成以外，其主要编程任务一般可用C语言完成 。

3.5.1 在C语言中内嵌汇编指令

- 在需要C与汇编混合编程时，若汇编代码较短，则可使用直接内嵌汇编的方法混合编程。内嵌汇编可以提高程序执行效率。
- 在C中内嵌汇编指令包含大部分的ARM和Thumb指令，不过其使用汇编文件中的指令有些不同，存在一些限制，主要有以下几个方面：

3.5.1 在C语言中内嵌汇编指令

- 不能直接向PC寄存器赋值，程序跳转只能使用B或BL指令实现
- 在使用物理寄存器时，不要使用过于复杂的C表达式，以避免物理寄存器冲突。
- 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-R3、R12和R14用于子程序调用，因此要避免直接 使用这些物理寄存器。
- 一般不要直接指定物理寄存器，而让编译器进行分配 。

3.5.1 在C语言中内嵌汇编指令

- 在ARM的C程序中我们可以使用关键字__asm来加入一段汇编语言的程序。
- 语法格式：
- __asm
- {
- 指令[；指令] /*注释*/
- …
- [指令]
- }

```
#include<stdio.h>
void Main (void)          __main
{
   int var=0xAA;
    __asm   //内嵌汇编标识
       {
            MOV R1,var
            CMP  R1,#0xAA
       }
   while(1);
}
```

3.5.2 在汇编中使用C定义的全局变量

- 内嵌汇编不用单独编辑汇编语言文件，使用简洁，但是有诸多限制，当汇编的代码较多时一般放在单独的汇编文件中。
- 这里就需要 在汇编和C之间进行一些数据的传递，最简便的办法就是使用全局变量。
- 使用IMPORT伪指令引入 全局变量，并利用LDR和STR指令根据全局变量的地址访问它们。

3.5.2 在汇编中使用C定义的全局变量

- 下面例子是一个汇编代码的函数，它读取全局变量global，将其加2 后写回：

```
        AREA globals,  CODE, READONLY
                IMPORT global
                ENTRY
                  LDR R1,=global
                  LDR R0,[R1]
                  ADD R0,R0,#2
                  STR R0,[R1]
stop B stop
        END
```

.s文件

```
int global=100;
```

.c文件

3.5.3 在C程序中调用汇编程序

- C程序调用汇编程序时需要做到以下两步：
- （1）在C程序中使用Extern关键字声明外部函数（声明要调用的汇编子程序），即可调用此汇编子程序。
- （2）在汇编程序中使用Export伪指令声明本子程序，使其他程序可以调用此子程序。另外，汇编程序设置要遵循AAPCS规则，保证程序调用时参数的正确传递。

```
#include<stdio.h>
extern int func1(int a,int b,int c,int d);
int main(int argc,char **argv)
{
   int a=1,b=2,c=3,d=4;
   int z;
   z=func1(a,b,c,d);
   printf("%d",z);
   return 0;
}
```

```
    EXPORT      func1
    AREA        exam, CODE, READONLY
func1
    ADD R0,R0,R1
    ADD R0,R0,R2
    ADD R0,R0,R3
    MOV         PC,LR
    END
```

3.5.3 在汇编程序中调用C程序

- 汇编程序的设置要遵循AAPCS规则，即前四个参数 通过R0-R3传递，后面的参数通过堆栈传递，保证程序调用时参数的正确传递。
- 在汇编程序中使用IMPORT伪指令声明 将要调用的C程序函数。在调用C程序时，要正确设置入口参数，然后使用BL调用。

3.5.3 在汇编程序中调用C程序

- 汇编程序调用C程序的C函数 代码如下：

```
int cFun(int a,int b,int c)
{return (a+b+c);}
```

汇编程序调用C程序的代码如下：

```
AREA asmfile,CODE,READONLY
IMPORT cFun
ENTRY
MOV R0,#11
MOV R1,#22
MOV R2,#33
BL  cFun
END
```

3.6嵌入式C语言一些常见用法

3.6.1 define宏定义

- Define 是C语言中的预处理指令，用于宏定义可以提高源代码的可读性。常见的格式为：

  #define 标识符 字符串

- 其中，“标识符”为所定义的宏名；“字符串”可以是常数、表达式和格式串等。
- 例如：
- #define PLL_Q 7 //定义标识符PLL_Q的值为7

- 视频36min处