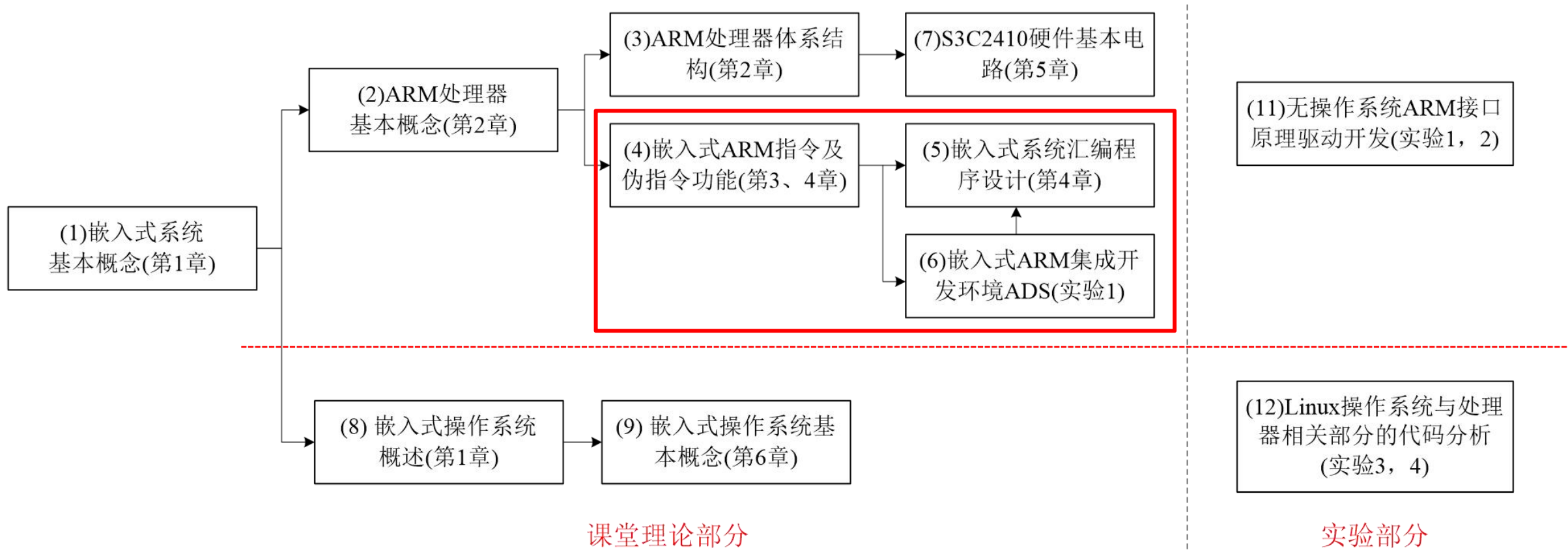


第四章 ARM汇编程序设计





第四章 ARM汇编程序设计

一、ARM汇编伪指令和伪操作

二、ARM汇编语言程序设计

三、汇编语言与C语言的混合编程

四、ARM映像文件

ARM汇编程序设计

一、ARM汇编伪指令和伪操作

二、ARM汇编语言程序设计

三、汇编语言与C语言的混合编程

四、ARM映像文件

- ARM微处理器的指令集可以分为五大类：
 - 跳转指令
 - 数据处理指令
 - 程序状态寄存器（PSR）处理指令
 - 加载/存储指令
 - 协处理器指令和异常产生指令

一个简单的汇编程序示例：

- 程序功能：计算 $20+8$

```
AREA Buf, DATA, READWRITE      ; 声明数据段Buf
Count    DCB    20                ; 定义一个字节单元Count
AREA Example, CODE, READONLY     ; 声明代码段Example
ENTRY    ; 标识程序入口
CODE32   ; 声明32位ARM指令

START

    LDRB R0, Count                ; R0 = Count =20
    MOV  R1, #8                   ; R1 = 8
    ADD  R0, R0, R1               ; R0 = R0 + R1
    B    START

END
```

ARM汇编语言语句格式

ARM汇编语言语句格式如下所示：

{ symbol } { instruction | directive | pseudo-instruction } { ; comment }

其中：

instruction----指令。

directive----伪操作。

pseudo-instruction----伪指令。

symbol---符号、标号。

comment---语句的注释。

START MOV R0, #10

;将立即数送R0寄存器

一、ARM汇编伪指令和伪操作



ARM汇编源程序由**指令**、**伪指令**和**伪操作**组成。

- 所有的**伪指令**、**伪操作**和**宏指令**概念与具体开发工具中的**编译器**有关。
- 目前常用的ARM编译开发环境IDE：
 - ✓ **ADS集成开发环境**：ARM公司开发，CodeWarrior公司的编译器、Keil MDK-ARM；
 - ✓ **GNU集成开发环境**：集成了GNU开发工具（编译器AS，编译器GCC，连接器LD等）。

课内主要介绍**ADS**开发环境下的**伪指令**和**伪操作**。

1. 伪指令和伪操作概念

伪指令——汇编后会被**ARM指令**所替代，转换成**机器码**。

伪操作——在汇编器汇编过程中起作用，**协助汇编**，不会转换成**机器码**，不参与程序运行。（类同单片机的“伪指令”）

2. ARM汇编伪指令

ARM伪指令不属于ARM指令集中的指令，是为了编程方便而定义的。

伪指令可以像其它ARM指令一样使用，但在编译时这些指令将被等效的ARM指令代替。ARM伪指令有四条，分别是：

- (1) ADR： 小范围的地址读取伪指令。
- (2) ADRL： 中等范围的地址读取伪指令。
- (3) LDR： 大范围的地址读取伪指令。😊
- (4) NOP： 空操作伪指令。

(1) ADR——小范围的地址读取

- ADR伪指令功能：将基于PC相对偏移的地址值读取到寄存器中。
- ADR伪指令的替代：在编译时，ADR伪指令被替换。通常，编译器用一条ADD指令或SUB指令来实现此伪指令的功能，若不能用一条指令实现，则产生错误，编译失败。

语法格式：

ADR{cond} register, expr

其中：

- register：加载的目标寄存器。
- expr：地址表达式。地址值字对齐时取值范围是-1020~1020。

- 例1：使用ADR将程序标号Delay所表示的地址存入R1。

```
.....  
(0x20)  ADR  R1,Delay  
.....  
Delay  
(0x64)  MOV  R0,R14  
.....
```

编译后的反汇编代码：

```
.....  
ADD  R1,PC,#0x3C  
.....  
MOV  R0,R14
```

```
PC+0x3C  
=0x20+0x08+0x3C  
=0x64
```

例2：查表

ADR R0, D_TAB ;加载转换表地址

LDRB R1, [R0,R2] ;使用R2作为参数，进行查表

偏移量R2=0,1,2, ...

.....

D_TAB

DCB 0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92

(2) ADRL——中等范围的地址读取

- ADRL伪指令功能：将基于PC相对偏移的地址值读取到寄存器中，比ADR伪指令可以读取更大范围的地址。
- ADRL伪指令的替代：在汇编编译器编译源程序时，ADRL被编译器替换成两条合适的指令。若不能用两条指令实现，则产生错误，编译失败。

语法格式：

```
ADRL{cond}    register,expr
```

其中：

- register：加载的目标寄存器。
- expr：地址表达式。地址值字对齐时取值范围是-256K~256K

例3：使用ADRL将程序标号
Delay所表示的地址存入R1。

.....
(0x20) **ADRL R1,Delay**
.....

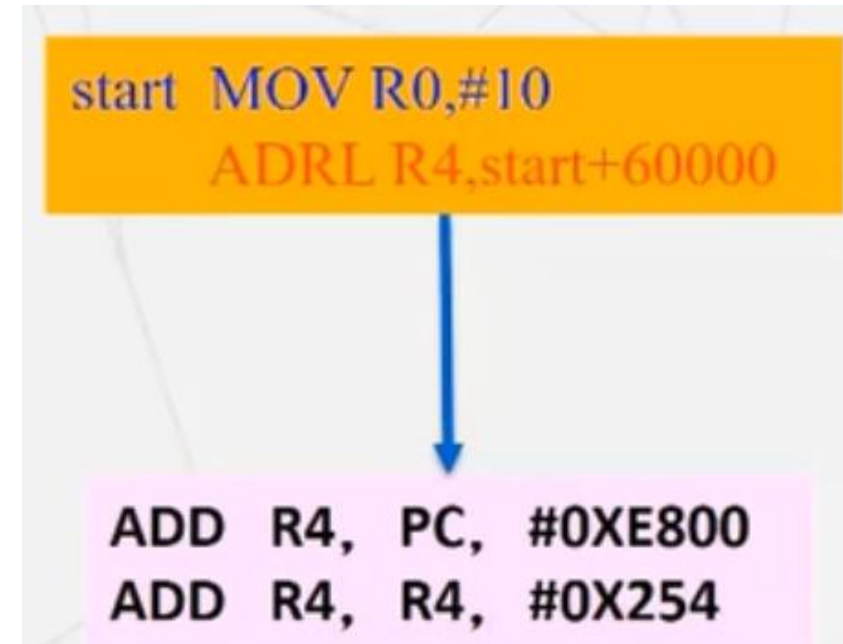
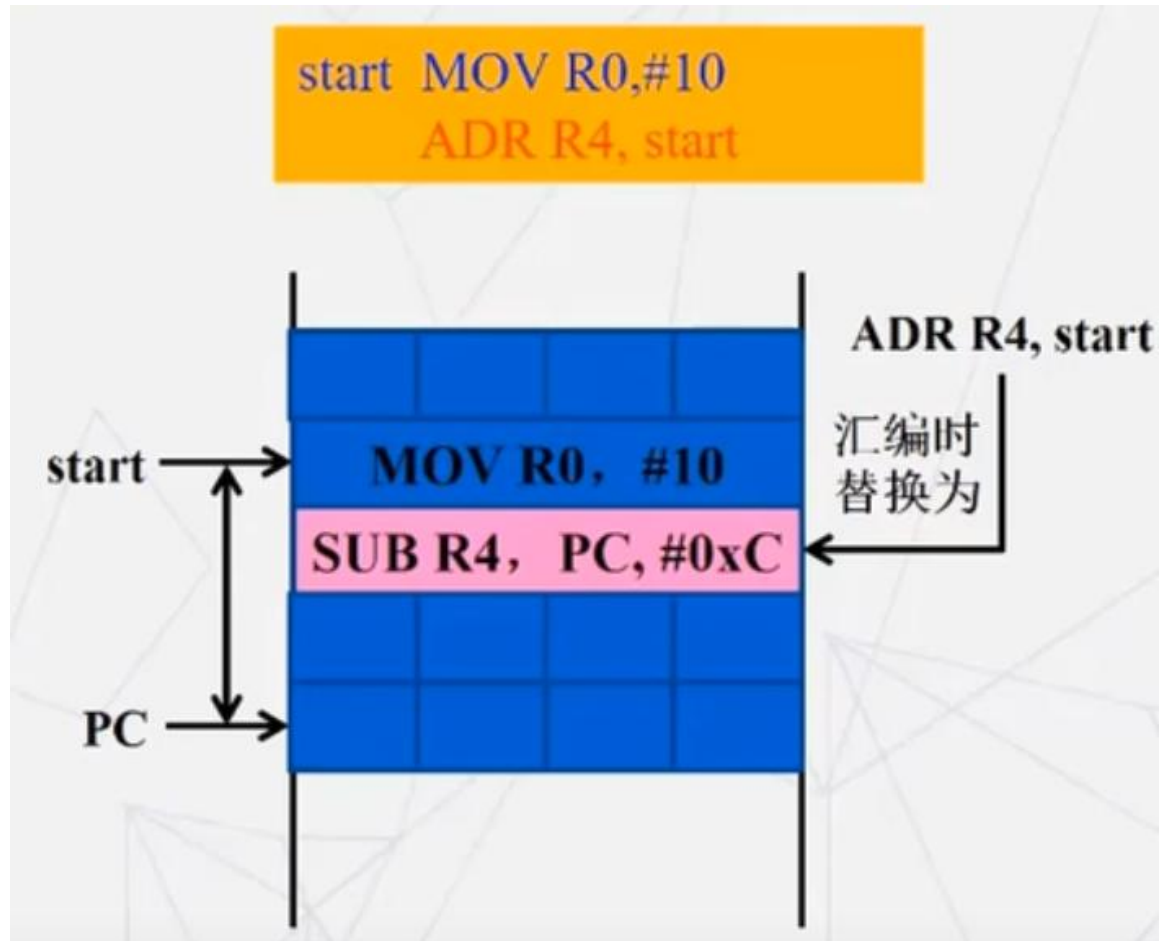
Delay
(0x64) **MOV R0,R14**
.....

编译后的反汇编代码：

.....
ADD R1,PC,#0x3C
ADD R1,R1,#0
.....
MOV R0,R14

ADRL伪指令被汇编成两条指令，尽管有时第2条指令并没有意义。

ADR vs. ADRL



(3) LDR ——大范围的地址读取

- LDR伪指令功能：用于赋值一个32位立即数或一个地址值到指定的寄存器。
- LDR伪指令的替代：在汇编编译源程序时，LDR伪指令被编译器替换成一条合适的指令。
 - 若加载的常数未超过MOV或MVN的范围，是合法常数，则使用MOV或MVN指令代替该LDR伪指令；
 - 否则汇编器将常量放入文字池，并使用一条程序相对偏移的LDR指令从文字池读出常量。

(3) LDR ——大范围的地址读取

语法格式：

```
LDR{cond} register, =expr
```

其中：

- **Register**：加载的目标寄存器。
- **expr**：32位常量或地址表达式。

Note：

- 与ARM指令的LDR的区别：伪指令LDR的参数有“=”号。
- 从指令位置到文字池的偏移量必须小于4KB。

例4：使用LDR将程序标号
Delay所表示的地址存入R1。

.....
(0x060) **LDR R1,=Delay**

.....
Delay
(0xFF) **MOV R0,R14**
.....

编译后的反汇编代码：

.....
MOV R1, #0xFF

.....
Delay
MOV R0,R14

.....

**例5：使用LDR将程序标号
Delay所表示的地址存入R1。**

```
.....  
(0x060)  LDR  R1,=Delay  
.....  
Delay  
(0x102)  MOV  R0,R14  
.....
```

LDR伪指令被汇编成一条LDR指令,并在文字池中定义一个常量,该常量为标号Delay的地址。

编译后的反汇编代码：

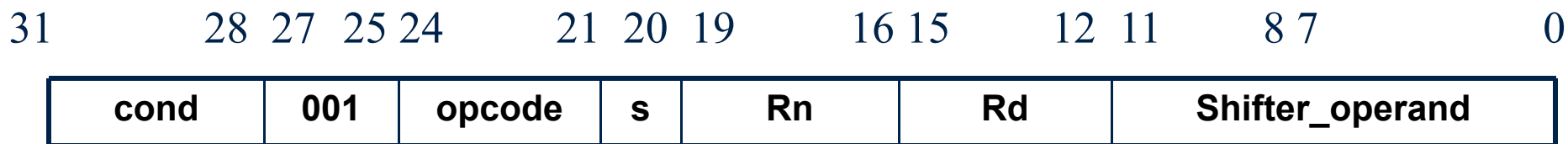
```
.....  
LDR  R1, stack  
.....  
Delay  
MOV  R0,R14  
.....  
  
LTORG  
stack  DCD  0x102
```

ARM指令(数据处理)中的立即数

- 用12位编码间接表示：**前4位**表示**移位位数**，**后8位**表示一个**常数**；
- 合法的32位立即数由 8位常数循环右移 ($2 \times \text{移位位数}$) 位得到。

如：0xF200

可以由0xF2循环右移24 ($24 = 2 \times 12$) 位得到所以是合法的，在指令中表示为
0x**CF**2



- MOV指令传送的常数必须用立即数表示。
- 当不知道一个数是否为合法数时，可以使用LDR伪指令来赋值。

如：

```
MOV R1, #4096
```

```
LDR R1, =4097
```

判别“立即数”的合法性的方法：

四、如何判断一个数是合法立即数还是非法立即数

1.判断一个数是否符合8位位图的原则, 首先看这个数的二进制表示中1的个数是否不超过8个. 如果不超过8个, 再看

这n个1($n \leq 8$)是否能同时放到8个二进制位中, 如果可以放进去, 再看这八个二进制位是否可以循环右移偶数位得到

我们欲使用的数. 如果可以, 则此数符合8位位图原理, 是合法的立即数. 否则, 不符合.

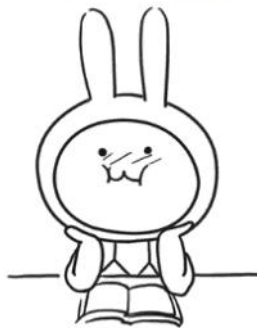
2.无法表示的32位数, 只有通过逻辑或算术运算等其它途径获得了. 比如0xffff00, 可以通过0x000000ff按位取反得到.

五、编程使用

其实你没必要一个一个的算, 只要利用**LDR伪指令**就可以了, 例如: `ldr r1, =12345678`

编译器自然会给你做工作, 现实的编程中应该也是这个居多吧

沉迷自学
无法自拔



(4) NOP——空操作伪指令

- **NOP伪指令功能实现方法：**在汇编时将被替代成ARM中的**空操作**，比如可能是“MOV R0,R0”指令等。
- **用途：**NOP可用于延时操作。
- **语法格式：** NOP

例：延时子程序

Delay

```
NOP    ;空操作
NOP
NOP
SUBS   R1,R1,#1  ;循环次数减1
BNE    Delay
MOV    PC,LR
```


ARM伪指令例程及总结

```
AREA    EXAMPLE3, CODE, READONLY
ENTRY
START
MOV     R0, #10           ;PC值为当前指令地址值加8字节
ADR     R1, START         ;本ADR伪指令被编译器替换成
                          ;SUB R1, PC, #0xc
ADRL    R2, START+60000   ;本ADRL伪指令被编译器替换成
                          ;ADD R2, PC, #0xea00 和
                          ;ADD R2, R2, #0x50
LDR     R3, =0xFF0FF      ;将0xFF0FF读取到R3中

END1
B       END1
END
```

共同点:

- 都是地址读取指令。

不同点:

- 读取地址范围不同;
- ADR, ADRL获得的是相对地址; LDR获得的是绝对地址。

3. ARM汇编伪操作

ADS编译环境下的伪操作可分为以下几类：

- 符号及符号定义（Symbol Definition）伪操作
- 数据定义（Data Definition）伪操作
- 汇编控制（Assembly Control）伪操作
- 其它（Miscellaneous）伪操作



符号的用途分三种：标号、常量、变量

类型	伪指令	功能
等效伪指令	EQU	为常量和标号定义一个等效的字符名称
全局变量定义伪指令	GBLA,GBLL,GBLS	定义一个全局数字变量，或逻辑变量，或字符串变量
局部变量定义伪指令	LCLA,LCLL,LCLS	定义一个局部数字变量，或逻辑变量，或字符串变量
变量赋值伪指令	SETA,SETL,SETS	为一个已经定义的全局或局部变量赋值
寄存器列表定义伪指令	RLIST	定义一个寄存器列表名称，针对 LDM/STM 指令



- **EQU**: 声明常量。
- **GBLA, GBLL, GBLS**: 声明全局变量。
- **LCLA, LCLL, LCLS**: 声明局部变量。
- **SETA, SETL, SETS**: 给变量赋值。
- **RLIST**: 为通用寄存器列表定义名称。

定义常量伪操作

• EQU

EQU伪操作为数字常量、寄存器的值和程序中的标号定义一个字符名称，类似于C语言中的#define。

语法格式：

```
name EQU expr{, type}
```

其中：

-name: 为expr定义的字符名称。

-expr: 为基于寄存器的地址值、程序中的标号、32位的地址常量或者32位的常量。表达式，为常量。

-type: 当expr为32位常量时，可以使用type指示expr的数据的类型。取值为：

✓ CODE32

✓ CODE16

✓ DATA

例：

tmp EQU 2 ;定义tmp符号的值为2

abcd EQU label+16 ;定义abcd符号的值为(label+16)

**test EQU 0x1c, CODE32 ;定义test符号的值为绝对
;地址值0x1c, 而且为32位ARM指令**

定义全局变量伪操作

- **GBLA/GBLL/GBLS**

该伪操用于定义一个全局变量，并将其初始化。

语法格式：

GBLA(GBLL/GBLS) <variable>

- ▣ 全局变量的变量名在整个程序范围内必须具有唯一性；
- ▣ **GBLA**定义一个全局数字变量，其默认初值为0；
- ▣ **GBLL**定义一个全局逻辑变量，其默认初值为假{FALSE0}；
- ▣ **GBLS**定义一个全局字符串变量，其默认初值为空；

GBLA Test1 ； 定义一个全局数字变量，变量名为Test1

GBLL Test2 ； 定义一个全局逻辑变量，变量名为Test2

GBLS Test3 ； 定义一个全局字符串变量，变量名为Test3

定义局部变量伪操作

- **LCLA/LCLL/LCLS**

该伪操用于定义一个局部变量，并将其初始化。

语法格式：

LCLA(LCLL/LCLS) <variable>

- ▣ 局部变量的变量名在变量作用范围内必须具有唯一性，通常只在定义该变量的程序段内有效；
- ▣ **LCLA**定义一个局部数字变量，其默认初值为0；
- ▣ **LCLL**定义一个局部逻辑变量，其默认初值为假{FALSE0}；
- ▣ **LCLS**定义一个局部字符串变量，其默认初值为空；

LCLA Test4 ； 定义一个局部数字变量，变量名为Test4

LCLL Test5 ； 定义一个局部逻辑变量，变量名为Test5

LCLS Test6 ； 定义一个局部字符串变量，变量名为Test6

变量赋值伪操作

- **SETA/SETL/SETS**

该伪操用于定义一个局部变量，并将其初始化。

语法格式：

<variable> SETA(SETL/SETS) <expr>

- SETA、SETL和SETS用于给一个已经定义的全局变量或局部变量进行赋值。

Test1 SETA 0xAA	;将Test1变量赋值为0xAA
Test2 SETL {TRUE}	;将Test2变量赋值为真
Test3 SETS "Testing"	;将Test3变量赋值为 "Testing"
Test4 SETA 0xBB	;将Test4变量赋值为0xBB
Test5 SETL {TRUE}	;将Test5变量赋值为真
Test6 SETS "Testing"	;将Test6变量赋值为 "Testing"

寄存器列表定义伪操作

- **RLIST**

语法格式:

```
<name> RLIST <{list}>
```

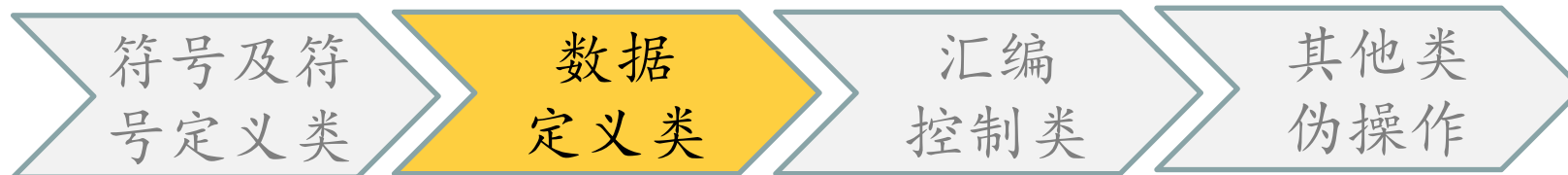
- ▣ LDM/STM需要使用一个比较长的寄存器列表，使用RLIST可以给列表定义一个统一的名称。

```
RegListA RLIST {R0-R5, R8, R10} ; 定义RegListA包含R0-R5,R8和R10。
```



(2) 数据定义伪操作

- **LTORG**: 声明一个数据缓冲池的开始。
- **SPACE**: 分配一块字节内存单元，并用0初始化。
- **DCB**: 分配一段**字节**内存单元，并初始化。
- **DCD、DCDU**: 分配一段**字**内存单元，并初始化。
- **MAP**: 定义一个结构化的内存表的首地址。
- **FIELD**: 定义结构化内存表中的一个数据域。



(2) 数据定义伪操作

类型	伪指令	功能
数据空间分配	DCB(=)/DCW(DCWU)/DCD(&,DCDU)/DCQ(DCQU)	分配连续的字节/半字/字/双字存储单元并初始化
	DCFD(DCFDU)/DCFS(DCFSU)	分配连续的双精度/单精度浮点数存储单元并初始化
	SPACE(%)	分配指定大小的连续存储单元并初始化为0
文字池定义	LTORG	定义一个暂存数据的数据缓冲区,即文字池
数据表定义	MAP, FIELD	定义一个结构化的内存表首地址和数据域

- **DCB**——也可以用符号“=”表示

用于定义并且初始化一个或者多个字节的内存区域，并指定表达式对其进行初始化。

语法格式：

{label} DCB expr{,expr}.....

或

{label} = expr{,expr}

其中expr表示：

- 0到255之间的一个数值常量或者表达式。
- 一个字符串。

例：

Dat1 DCB 0x7E,0x19

;为Dat1分配了2个字节,并
;初始化为0x7E和0x19

string DCB “This is a test!”

;分配连续字节空间并初始化
;成字符串

- **DCD、DCDU分配一段字内存单元**

- ✓ **DCD** ——分配一段字对齐的内存单元。也可用符号“&”表示

用于分配一段字对齐的内存单元，并初始化。

语法格式：

```
{label} DCD expr{,expr}.....
```

或

```
{label} & expr{,expr}.....
```

- ✓ **DCDU** ——分配一段字非严格对齐的内存单元

DCDU与DCD的不同之处在于DCDU分配的内存单元并不严格字对齐。

例：

```
data1 DCD 2,4,6 ;为data1分配三个字,内容初始化为2,4,6
```

```
data2 DCD label+4 ;初始化data2为label+4对应的地址
```

- **SPACE**分配一段字节内存单元

语法格式:

{label} SPACE expr

- 用于分配一片连续的存储区域并初始化为0;
- 表示式为要分配的字节数, **SPACE**也可用“%”代替。

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

DataSpace % 100

• LORG

语法格式:

LTORG

说明某个存储区域用来做一个暂存数据的缓冲区，也叫“文字池”

```
start BL func
      .....
func  LDR  R1,=0x8000 ;子程序
      .....
      MOV  PC,LR      ;子程序返回

      LTORG
Data  SPACE  4200      ;从当前位置开始分配4200字节的内
                          ;存单元，并初始化为0。

      END
```

默认数据缓冲池为空

注意:

LTORG伪操作通常放在无条件跳转指令之后，或者子程序返回指令之后，避免误将数据缓冲池中的数据当作指令来执行。

通常ARM汇编编译器把数据缓冲池放在代码段的最后面，即下一个代码段开始之前，或者END伪操作之前。



- **MAP、FIELD**

指令格式：

MAP <expr> {,<baseregister>}

<label> **FIELD** <expr>

- ▣ **MAP**用于定义一个结构化的内存表的**首地址**；
- ▣ **FIELD**用于定义一个结构化内存表中的**数据域**；
- ▣ 两者配合起来实现一个类似C语言中的结构体的功能。

	MAP	0x100	；定义结构化内存表首地址为 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



^

#

- **MAP、FIELD**

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

A FIELD 16 ; 定义域A的长度为16字节

B FIELD 32 ; 定义域B的长度为32字节

C FIELD 256 ; 定义域C的长度为256字节



- **MAP、FIELD 应用实例 (2410INIT.s)**

^_ISR_STARTADDRESS_ ; 定义结构化内存表,首地址的值

HandleReset # 4

HandleUndef # 4

HandleSWI # 4

HandlePabort # 4

HandleDabort # 4

HandleReserved # 4

HandleReIRQ # 4

HandleReFIQ # 4

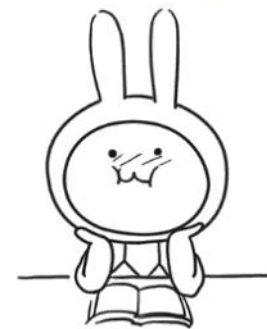


(3) 汇编控制伪操作

- **IF,ELSE及ENDIF**: 有条件选择汇编
- **WHILE及WEND**: 有条件循环（重复）汇编
- **MACRO, MEND及MEXIT**: 宏定义汇编

类型	伪指令	功能
条件汇编	IF,ELSE,ENDIF	根据条件是否成立来决定汇编某个程序段
重复汇编	WHILE, WEND	根据条件是否成立来决定是否重复汇编一个程序段
宏定义	MACRO,MEND,MEXIT	定义一个可以带参数的程序段，可使用宏名来引用

沉迷自学
无法自拔



• IF, ELSE, ENDIF

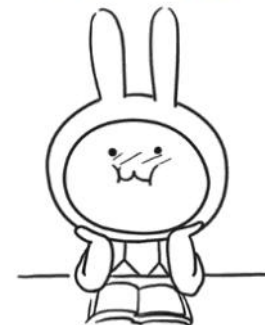
指令格式:

```
IF 逻辑表达式
    程序段1
ELSE
    程序段2
ENDIF
```

- 根据条件的成立与否决定是否执行某个程序段;
- 可以嵌套使用;
- 有时为了简单起见, IF, ELSE, ENDIF可以分别用[,|,]代替。

```
GBLL CONFIG
IF CONFIG = TRUE
    BNE __rt_udiv_1
    LDR R0, =__rt_div0
    BX R0
ELSE
    BEQ __rt_div0
ENDIF
```

沉迷自学
无法自拔



• WHILE, WEND

指令格式:

```
WHILE 逻辑表达式  
    程序段  
WEND
```

```
GBLA no  
no SETA 3  
WHILE no<5  
    no SETA no+1  
    ...  
WEND
```

- 根据条件的成立与否决定是否重复汇编一个程序段;
- 若WHILE后面的逻辑表达式为TURE, 则重复汇编该程序段, 直到逻辑表达式为FALSE;
- 可以使用嵌套。

沉迷自学
无法自拔



- **MACRO, MEND, MEXIT**

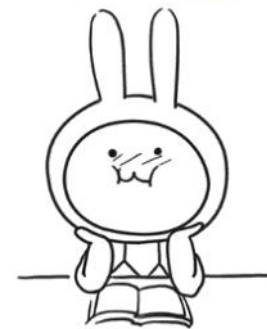
指令格式:

MACRO

\$标号 宏名 \$参数1, \$参数2,
程序段 (宏定义体, 可包含MEXIT)

MEND

沉迷自学
无法自拔



- **MACRO**定义一个宏语句段的开始, **MEND**定义宏语句段的结束, **MEXIT**可以实现宏程序段的跳出。使用方法是通過宏指令的方式来调用。
- 宏指令的参数必须前面加\$符号, 类似于函数的形参, 可以有0-n个。

• MACRO, MEND, MEXIT

定义宏

```
MACRO DTX $date, $time
    LDR R1, = 0x1000
    LDR R0, = $date
    STR R0, [R1], #04
    LDR R2, = $time
    STR R2, [R1]
MEND
```

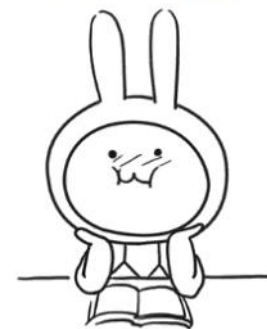
```
...
STMFD SP!, {r0-r3}
DTX 0x858, 12 ;
ADD R3, R0, R2
```

调用宏

宏展开后

```
...
STMFD SP!, {r0-r3}
LDR R1, = 0x1000
LDR R0, = 0x858
STR R0, [R1], #04
LDR R2, = 12
STR R2, [R1]
ADD R3, R0, R2
```

沉迷自学
无法自拔





(4) 其它伪操作

- **AREA:** 定义一个代码段或数据段
- **CODE16、CODE32:** 告诉编译器后面的指令序列位数
- **ENTRY:** 指定程序的入口点
- **ALIGN:** 将当前的位置以某种形式对齐
- **END:** 源程序结尾

- **EXPORT、GLOBAL：** 声明源文件中的符号可以被其他源文件引用
- **IMPORT、EXTERN：** 声明某符号是在其他源文件中定义的
- **GET、INCLUDE：** 将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理。
- **INCBIN：** 将一个文件包含到当前源文件中，而被包含的文件不进行汇编处理（用于包含可执行文件或任意数据）。



(4) 其它伪操作

伪指令	功能
ALIGN	填充字节使当前位置满足对齐要求
CODE16, CODE32	指定采用的指令集
ENTRY	指定汇编程序的入口点
END	通知汇编器结束汇编
EXPORT, GLOBAL	声明一个能被其它源文件引用的符号
IMPORT, EXTERN	声明要引用一个外部符号
GET, INCLUDE	包含一个源文件
INCBIN	包含一个目标文件或数据文件
RN	定义寄存器别名
ROUT	定义局部变量作用范围

• AREA

用于定义一个代码段或是数据段。

语法格式：

```
AREA sectionname{,attr} {,attr}...
```

其中：

- **sectionname**：为所定义的段的名称。
- **attr**：该段的属性。具有的属性为：
 - ▣ **CODE**：定义代码段。
 - ▣ **DATA**：定义数据段。
 - ▣ **READONLY**：指定本段为只读,代码段的默认属性。
 - ▣ **READWRITE**：指定本段为可读可写,数据段的默认属性。

- ▣ **ALIGN:** 指定段的对齐方式为 $2^{\text{expression}}$ 。expression的取值为0~31。
- ▣ **COMMON:** 指定一个通用段。该段不包含任何用户代码和数据。
- ▣ **NOINIT:** 指定此数据段仅仅保留了内存单元，而没有将各初始值写入内存单元，或者将各个内存单元值初始化为0。

```
AREA Init, CODE, READONLY, ALIGN=3 ; ;指定8字节对齐
    代码段
END
```

Note:

- 一个汇编程序至少包含一个代码段。
- 一个大的程序可包含多个代码段和数据段。

- **CODE16和CODE32**

CODE16告诉汇编编译器后面的指令序列为16位的Thumb指令。

CODE32告诉汇编编译器后面的指令序列为32位的ARM指令。

语法格式：

CODE16
CODE32

注意：CODE16和CODE32只是告诉编译器后面指令的类型，该伪操作本身不进行程序状态的切换。

例：计算数组的第一项和第五项之和。

AREA Buf, DATA, READWRITE

Array DCD 0x11, 0x22, 0x33, 0x44, 0x55

DCD 0x66,

AREA ChangeState, CODE, READONLY

ENTRY

CODE32 ;下面为32位ARM指令

LDR R0, = Array

.....

END

例:

AREA ChangeState, CODE, READONLY

ENTRY

CODE32 ;下面为32位ARM指令

LDR R0,=start+1

BX R0 ;切换到Thumb状态，并跳转到start处执行

.....

CODE16 ;下面为16位Thumb指令

start MOV R1,#10

.....

END

- **ENTRY**

指定程序的入口点。

语法格式：

ENTRY

Note:

一个程序（可包含多个源文件）中至少要有一个ENTRY（可以有多个ENTRY），但一个源文件中最多只能有一个ENTRY（可以没有ENTRY）。

• ALIGN

ALIGN伪操作通过填充0将ALIGN指令之前的数据以某种形式对齐。

语法格式：

```
ALIGN {expr{,offset}}
```

其中：

- expr: 一个数字，表示对齐的单位。这个数字是2的整数次幂，范围在 $2^0 \sim 2^{31}$ 之间。
如果没有指定expr，则当前位置对齐到下一个字边界处。
- Offset: 偏移量，可以为常数或数值表达式。不指定offset表示将当前位置对齐到以expr为单位的起始位置。

例1:

short DCB 1 ;本操作使字对齐被破坏
ALIGN ;重新使其为字对齐

例2:

ALIGN 8 ;当前位置以2个字的方式对齐

- **END**

END伪操作告诉编译器已经到了源程序结尾。

语法格式:

END

Note:

每一个汇编源程序都必须包含**END**伪操作，以表明本源程序的结束。

- **EXPORT及GLOBAL**

声明一个源文件中的符号，使此符号可以被其他源文件引用。

语法格式：

```
EXPORT/GLOBAL symbol {[weak]}
```

其中：

- symbol: 声明的符号的名称。（区分大小写）
- [weak]: 声明其他同名符号优先于本符号被引用。

例：

```
AREA example, CODE, READONLY
EXPORT DoAdd
DoAdd ADD R0, R0, R1
```

- **IMPORT及EXTERN**

声明一个符号是在其他源文件中定义的。

语法格式：

```
IMPORT symbol{[weak]}  
EXTERN symbol{[weak]}
```

其中：

-symbol: 声明的符号的名称。

```
IMPORT Main ;表示需要引用Main符号
```

– **[weak]:**

- 当没有指定此项时，如果symbol在所有的源文件中都没有被定义，则连接器会报告错误。
- 当指定此项时，如果symbol在所有的源文件中都没有被定义，则连接器不会报告错误，而是进行下面的操作。
 - 如果该符号被B或者BL指令引用，则该符号被设置成下一条指令的地址，该B或BL指令相当于一NOP指令。
 - 其他情况下此符号被设置成0。

- **GET及INCLUDE**

将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理。

指令格式：

```
GET    filename  
INCLUDE filename
```

其中：

- filename:包含的源文件名，可以使用路径信息（可包含空格）。

例：

```
GET d:\arm\file.s
```


ARM伪指令例程及总结

```
AREA    EXAMPLE3,CODE,READONLY
ENTRY
START
MOV     R0,#10           ;PC值为当前指令地址值加8字节
ADR     R1,START         ;本ADR伪指令被编译器替换成
                        ;SUB R1, PC, #0xc
ADRL    R2,START+60000   ;本ADRL伪指令被编译器替换成
                        ;ADD R2, PC, #0xea00 和
                        ;ADD R2, R2, #0x50
LDR     R3,=0xFF0FF      ;将0xFF0FF读取到R3中

END1
B       END1
END
```

共同点:

- 都是地址读取指令。

不同点:

- 读取地址范围不同;
- ADR, ADRL获得的是相对地址; LDR获得的是绝对地址。

ARM伪操作总结

ADS编译环境下的伪操作可分为以下几类：

- 符号及符号定义（Symbol Definition） 符号常量变量定义伪操作；
- 数据定义（Data Definition） 开辟各种内存空间伪操作；
- 汇编控制（Assembly Control） 条件汇编及宏伪操作
- 其它（Miscellaneous） 段、程序入口、程序结束等伪操作

ARM汇编程序设计

一、ARM汇编伪指令和伪操作

二、ARM汇编语言程序设计

三、汇编语言与C语言的混合编程

四、ARM映像文件

二 ARM汇编语言程序设计

- 一个ARM汇编程序以段（section）为单位组织源文件；
- 段是相对独立的、具有特定名称的、不可分割的指令或数据序列；
- 段又可以分为代码段和数据段，代码段存放执行代码，数据段存放代码运行时需要用到的数据；
- 一个ARM源程序至少有一个代码段，大的程序可以包含多个代码段和数据段。

1、ARM汇编中的文件格式

ARM源程序文件（可简称为源文件）可以由任意一种**文本编辑器**来编写**程序代码**，它一般为文本格式。在ARM程序设计中，常用的源文件可简单分为以下几种：

源程序文件	文件名	说 明
汇编程序文件	*.s	用ARM汇编语言编写的ARM程序或Thumb程序
C程序文件	*.c	用C语言编写的程序代码
头文件	*.h	头文件

2. ARM汇编语言程序基本结构

ADS环境下ARM汇编语言源程序的基本结构:

```
AREA EXAMPLE, CODE, READONLY
ENTRY          ; 标识程序的入口点
start          ; 为地址标号
    MOV r0, #10
    MOV r1, #3
    ADD r0, r0, r1
    B start
END            ; 表示源文件结束
```

;用AREA伪操作定义了一个名为EXAMPLE的代码段, 该代码段是只读属性的

上述程序的程序体部分实现了一个简单的加法运算。

ARM汇编语句输入规则

ARM汇编语言语句格式如下所示：

{ symbol } <instruction | directive | pseudo-instruction> { ; comment }

{标号} <指令/伪指令/伪操作> { ; 注释 }

- 每一条指令的助记符可以全部大写或小写，**不能大小写混合**；
- 如果一条语句太长，则可在末尾用“****”来连接下一行；
- **标号必须顶格**书写，其后**不加“:”**；
- 对于变量的设置、常量的定义，必须顶格书写；
- 所有指令均不能顶格书写；
- **注释用“;”开始**，到此行结束。

错误的书写示例：

```
START MOV R0, #1
ABC:  MOV R1, #2
MOV R2, #3
LOOP  Mov R2, #3
      B loop
```

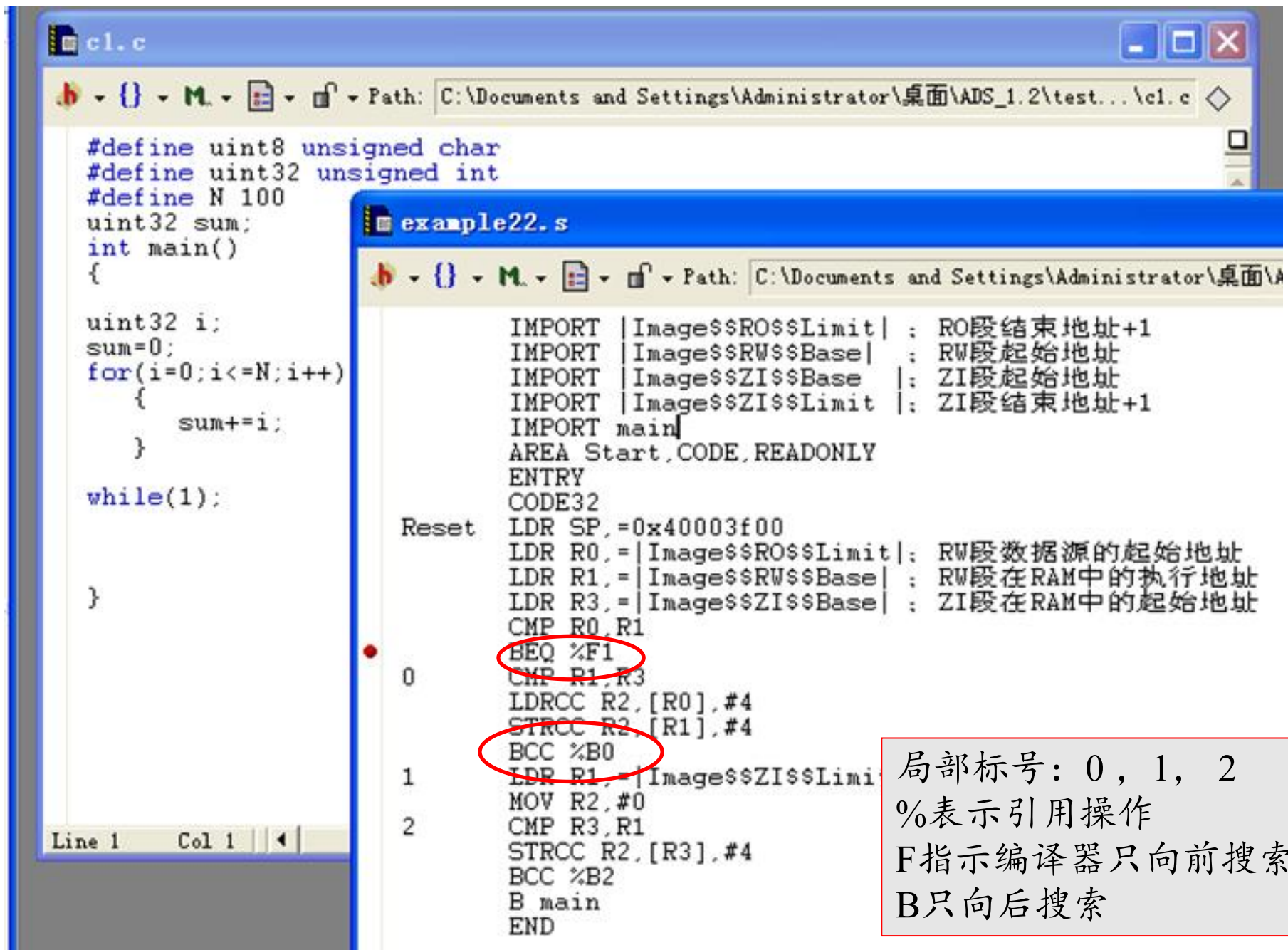
← 标号START没有顶格

← 标号ABC后面不能带“:”

← 指令不能顶格写

← 指令中不能大小写混写

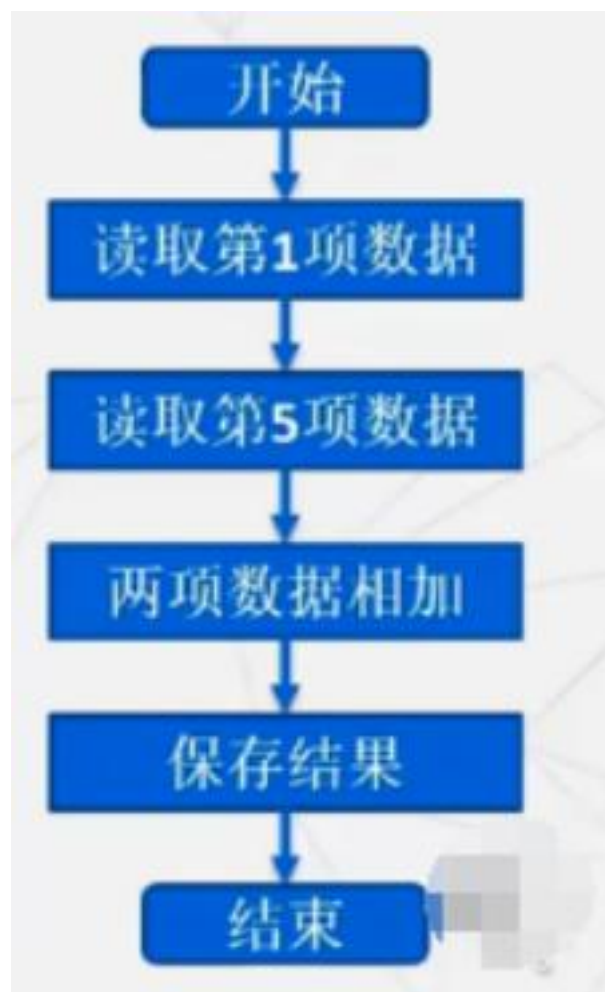
← loop ≠ LOOP



3. ARM汇编程序的结构化编程

- 顺序结构

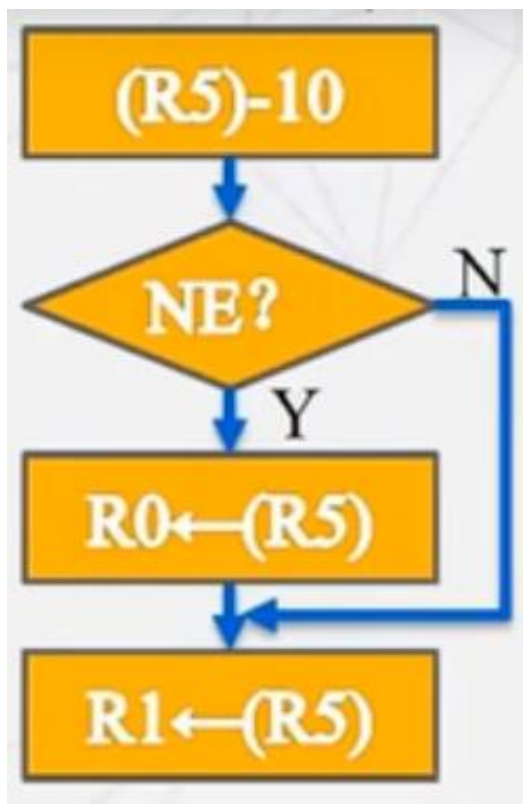
例1：计算数组第一项和第五项之和。



```
AREA Buf, DATA, READWRITE ; 定义数据段Buf
Array      DCD 0x11, 0x22, 0x33, 0x44 ; 定义数组Array
           DCD 0x55, 0x66, 0x77, 0x88
           DCD 0x00, 0x00, 0x00, 0x00
AREA Example, CODE, READONLY
ENTRY
CODE32
    LDR    R0, = Array ; 取得数组Array首地址
    LDR    R2, [R0] ; 数组第1项→ R2
    MOV    R1, #4
    LDR    R3, [R0, R1, LSL #2] ; 数组第5项→ R3
    ADD    R2, R2, R3 ; R2 + R3 → R2
    MOV    R1, #8 ; R1 = 8
    STR    R2, [R0, R1, LSL #2] ; 保存结果到数组第9项
END
```

• 分支结构

例2：如果寄存器R5中的数据等于10，就把R5中的数据存入寄存器R1；否则把R5中的数据分别存入寄存器R0和R1。



```
CMP    R5, #10
MOVNE  R0, R5
MOV     R1, R5
```

方法一：用条件指令实现

```
CMP    R5, #10
BEQ     doequal
MOV     R0, R5
doequal MOV  R1,R5
```

方法二：用条件转移指令实现

- 循环结构

例3：编写一个程序，把首地址为DATA_SRC的80个字的数据复制到首地址为DATA_DST的目标数据块中。

```
LDR    R1, =DATA_SRC
LDR    R0, =DATA_DST
MOV    R10, #10
LOOP   LDMIA R1!, {R2-R9}
        STMIA R0!, {R2-R9}
        SUBS R10, R10, #1
        BNE  LOOP
```

• 子程序结构

例4：编写一个子程序MAX，实现求两个数的最大值，再主程序中调用MAX。

```
X EQU 19
N EQU 20
    AREA Example4, CODE, READONLY
    ENTRY
    CODE32

START    LDR    R0, =X
         LDR    R1, =N
         BL     MAX

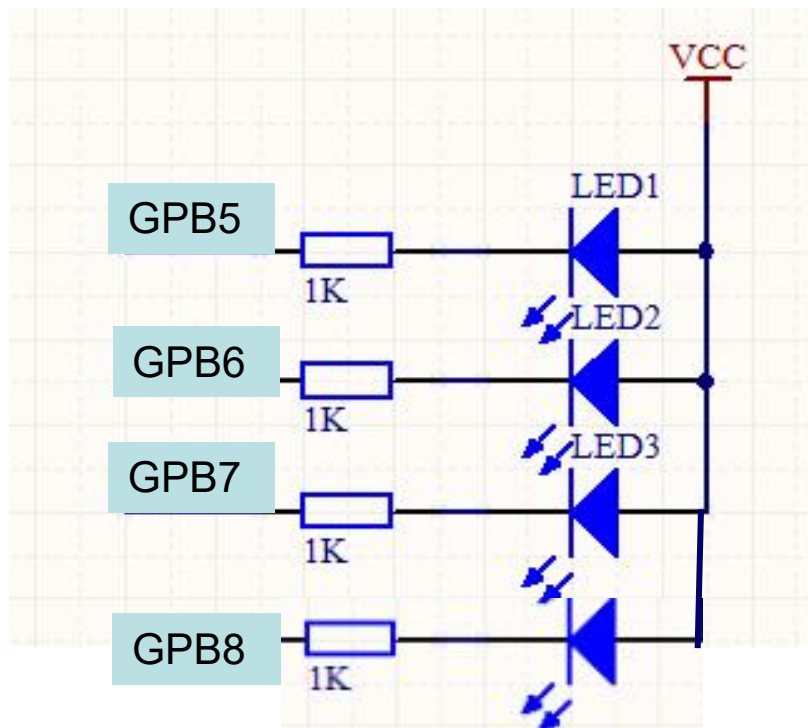
HALT     B      HALT

MAX      CMP    R0, R1
         MOVHI  R2, R0
         MOVLS  R2, R1
         MOV    PC, LR

MAX_END
    END
```


4. ARM汇编语言程序实例

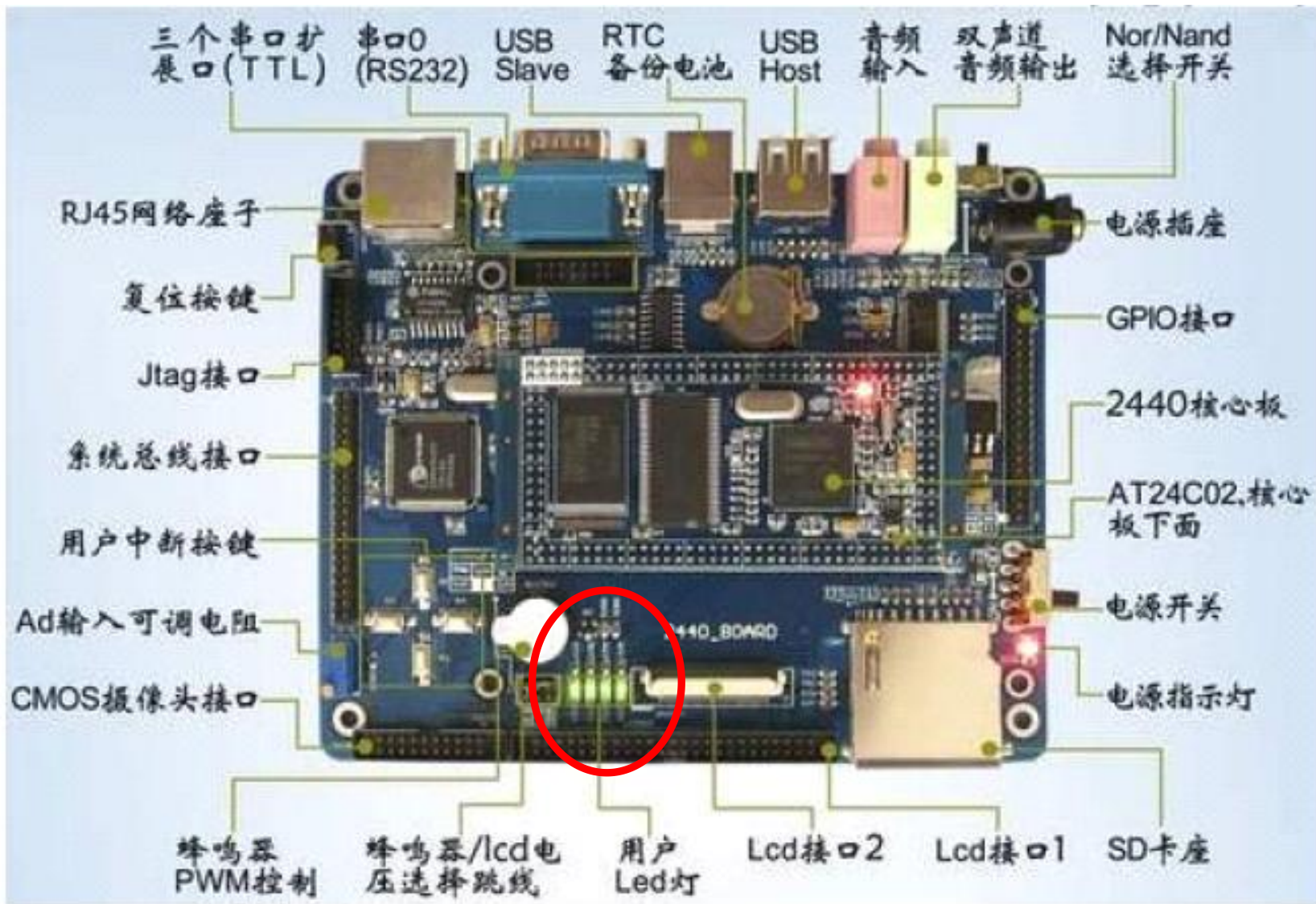
例1：下面的汇编语言设计的程序是针对实验开发板的LED控制器的设计程序，**利用B口**的5~8四个I/O端口做输出，控制四个LED的发光。利用ADS编译生成asm.bin。



- **GPBCON:** 32位端口功能设置寄存器。每两位设置一根引脚的输入或输出：
00表示输入、10表示特殊功能
01表示输出、11保留不用。
- **GPBDAT:** 32位端口数据寄存器。用于读写引脚电平；
- **GPBUP:** 引脚悬空态电平设置寄存器，32位。
某位=1，引脚无上拉电阻；
某位=0，引脚接上拉电阻；

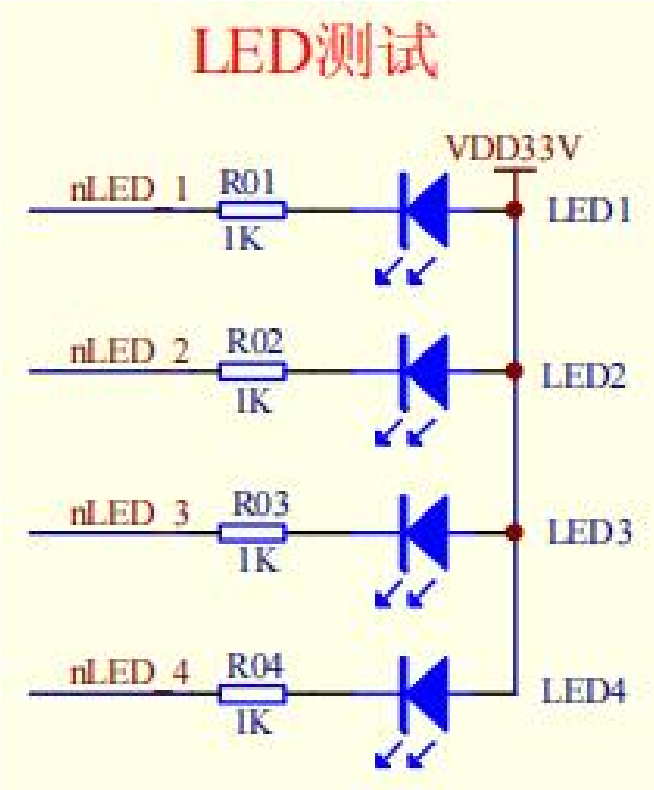
4. ARM汇编语言程序实例

例1：设计汇编语言程序实现实验开发板的LED灯的控制，控制四个LED流水点亮。
利用ADS编译生成asm.bin。

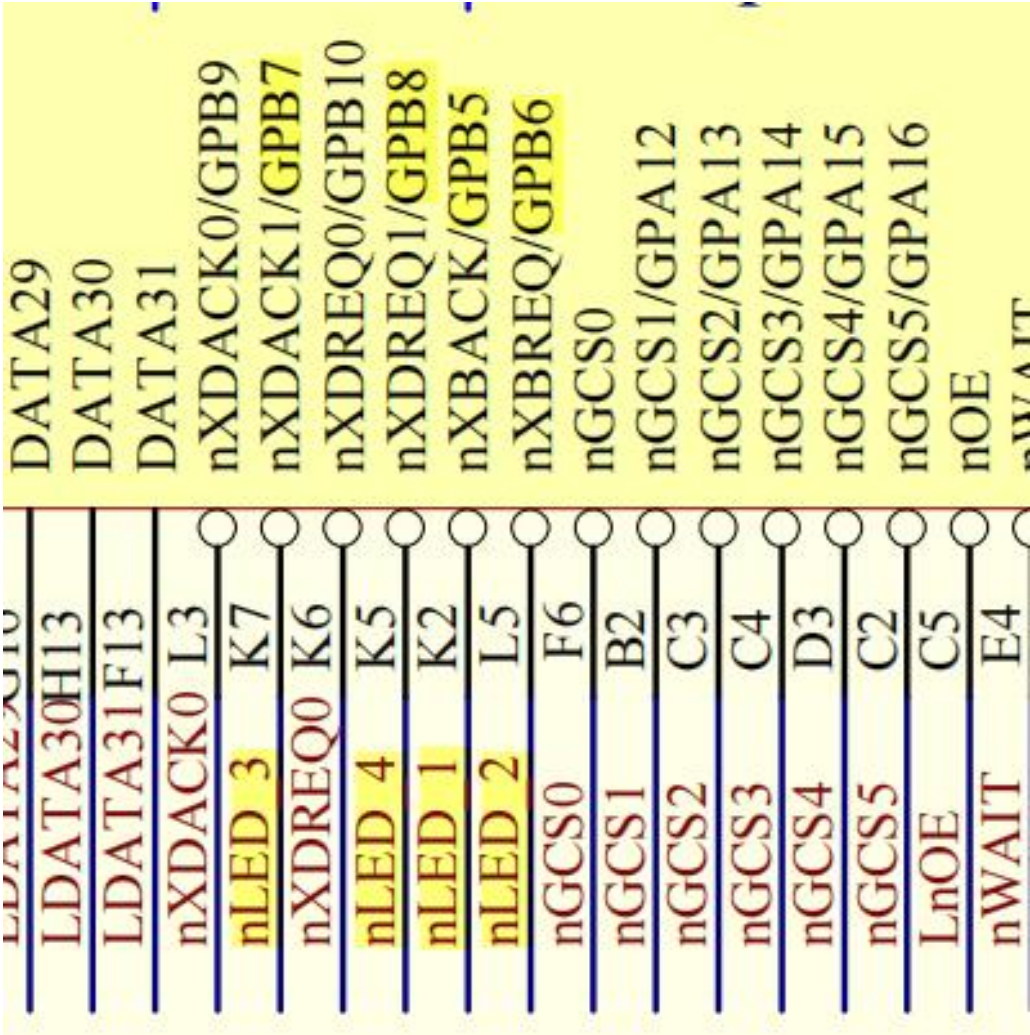


TQ2440开发板

TQ2440开发板原理图



TQ2440底板原理图



TQ2440核心板原理图

S3C2410 GPIO接口

- S3C2410共有117个多功能I/O端口，分为A~H共8组：

端口A(GPA): 23个输出引脚;

端口B(GPB): 11个输入/输出引脚;

端口C(GPC): 16个输入/输出引脚;

端口D(GPD): 16个输入/输出引脚;

端口E(GPE): 16个输入/输出引脚;

端口F(GPF): 8个输入/输出引脚;

端口G(GPG): 16个输入/输出引脚;

端口H(GPH): 11个输入/输出引脚;

通过寄存器来操作GPIO引脚:

GPxCON: 选择引脚功能,

GPxUP: 确定是否使用内部上拉电阻;

GPxDAT: 读/写引脚数据。

Note:

- x为A、B、...、H;
- 没有GPAUP寄存器。

Note:

- 可以通过设置寄存器来确定某个引脚用于输入、输出还是其他特殊功能。比如可以设置GPH6作为一般的输入、输出引脚，或者用于串口。

GPB端口的寄存器

- **GPBCON**寄存器：用于配置引脚功能。每两位对应一根引脚。
 - 00=输入，
 - 01=输出，
 - 10=特殊功能，
 - 11=特殊功能或保留。
- **GPxUP**寄存器：设置某位是否禁止接内部上拉电阻
 - =1，禁止，即不接内部上拉电阻，则相应管脚电平=0；
 - =0，不禁止，即接，则相应管脚电平=1。
- **GPBDAT**寄存器：用于读/写引脚。
 - 引脚设为输入时，读此寄存器可知相应引脚的电平状态是高还是低；
 - 引脚设为输出时，写此寄存器相应位可令此引脚输出高或低电平。

```

;汇编指令实验
;定义端口E寄存器预定义
rGPBCON EQU 0x56000010
rGPBDAT EQU 0x56000014
rGPBUP EQU 0x56000018

AREA Init, CODE, READONLY ;该伪指令定义了一个代码段, 段名为Init, 属性只读
ENTRY ;程序的入口点标识
ResetEntry
;下面这三条语句, 主要是用来设置I/O口GPB5为输出属性
ldr r0, =rGPBCON ;将寄存器rPCONB的地址存放到寄存器r0中
ldr r1, =0x400
str r1, [r0] ;将r1中的数据存放到寄存器rPCONB中

;下面这三条语句, 主要是禁止GPB端口的上拉电阻
ldr r0, =rGPBUP
ldr r1, =0xffff
str r1, [r0]

ldr r2, =rGPBDAT ;将数据端口B的数据寄存器的地址附给寄存器r2
ledloop
ldr r1, =0xff
str r1, [r2] ;使GPB5输出高电平, LED1灭
bl delay ;调用延迟子程序

ldr r1, =0x0
str r1, [r2] ;使GPB5输出低电平, LED1亮
bl delay ;调用延迟
b ledloop ;不断的循环, LED1将不停的闪烁

;下面是延迟子程序
delay
ldr r3, =0xbffff ;设置延迟的时间
delay1
sub r3, r3, #1 ;r3=r3-1
cmp r3, #0x0 ;将r3的值与0相比较
bne delay1 ;比较的结果不为0 (r3不为0), 继续调用delay1, 否则执行下
mov pc, lr ;返回

END ;程序结束符

```

设置I/O口功能为输出

设置I/O口禁止上拉电阻

设置I/O口输出高低电平,
点亮led灯

实验二例程:
实现一个LED灯的闪烁

;汇编指令实验

;定义端口B寄存器预定义

rGPBCON EQU 0x56000010

rGPBDAT EQU 0x56000014

rGPBUP EQU 0x56000018

AREA Init, CODE, READONLY ;定义一个代码段，段名为Init，属性只读
ENTRY ;程序的入口点标识

ResetEntry

;下面这三条语句，主要是用来设置I/O口GPB7为输出属性

ldr r0,=rGPBCON ;将寄存器rPCONB的地址存放到寄存器r0中

ldr r1,=0x15400

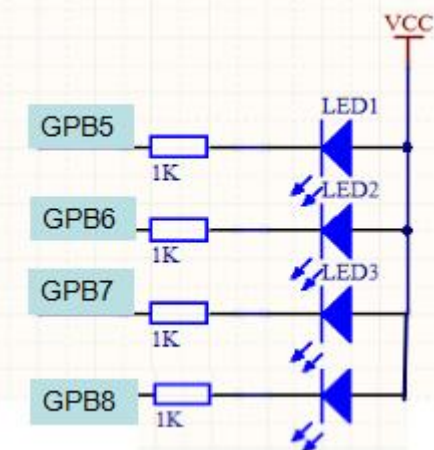
str r1,[r0] ;将r1中的数据存放到寄存器rPCONB中

;下面这三条语句，主要是禁止GPB端口的上拉电阻

ldr r0,=rGPBUP

ldr r1,=0xffff

str r1,[r0]



	ldr r2,=rGPBDAT	;将数据端口B的数据寄存器的地址附给寄存器
r2		
ledloop		
	ldr r1,=0x1C0	
	str r1,[r2]	;使GPB7输出低电平，LED1会亮
	bl delay	;调用延迟子程序
	 ldr r1,=0x1A0	 ;使GPB8输出低电平，LED2会亮
	str r1,[r2]	
	bl delay	
	 ldr r1,=0x160	 ;使GPB9输出低电平，LED3会亮
	str r1,[r2]	
	bl delay	
	 ldr r1,=0x0E0	 ;使GPB10输出低电平，LED4会亮
	str r1,[r2]	
	bl delay	
	 b ledloop	 ;不断的循环

下面是延迟子程序

delay

ldr r3,=0xffffffff ;设置延迟的时间

delay1

sub r3,r3,#1 ;r3=r3-1

cmp r3,#0x0 ;将r3的值与0相比较

bne delay1 ;比较的结果不为0（r3不为0），继续调用delay1,否则执行下一条语句

mov pc,lr ;返回

END ;程序结束符

ARM汇编程序设计

一、ARM汇编伪指令和伪操作

二、ARM汇编语言程序设计

三、汇编语言与C语言的混合编程

四、ARM映像文件

三、汇编语言与C语言的混合编程

- ARM编程可以使用汇编语言和C/C++语言。
- 汇编语言执行效率高，大多用于实时性要求高和精细处理场合。
- 一个完整的程序设计中，一般初始化部分使用汇编语言，主要编程任务使用C/C++语言。

1、基本ARM ATPCS

基本ATPCS（ARM-THUMB procedure call standard）就是ARM程序和Thumb程序中子程序调用的基本规则。ATPCS规定了一些子程序间调用的基本规则。这些基本规则包括：

- ① 寄存器的名称及其使用规则。
- 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

ARM处理器中有R0~R15共16个寄存器，这些寄存器的使用必须满足下面的规则：

- 子程序间通过寄存器 R0—R3来传递参数，记作a1-a4。
- 在子程序中，使用寄存器R4~R11来保存局部变量。
- 寄存器R12用作保存SP，在函数返回时使用该寄存器出栈，记作ip。在子程序间的连接代码段中常有这种使用规则。
- 寄存器R13用作数据栈指针，记作sp。
- 寄存器R14称为连接寄存器，记作lr。
- 寄存器R15是程序计数器，记作pc。它不能用作其他用途。

ATPCS中各寄存器的使用规则及其名称

寄存器↵	别名↵	特殊名称↵	使用规则↵
R15↵	↵	Pc↵	程序计数器↵
R14↵	↵	Lr↵	连接寄存器↵
R13↵	↵	Sp↵	数据栈指针↵
R12↵	↵	ip↵	于程序内部调用的scratch寄存器↵
R11↵	v8↵	↵	ARM状态局部变量寄存器8↵
R10↵	V7↵	si↵	ARM状态局部变量寄存器7，在支持数据栈检查的ATPCS中为数据栈限制指针↵
R9↵ ↵	v6↵ ↵	sb↵ ↵	ARM状态局部变量寄存器6，在支持RWPI的ATPCS中为静态基址寄存器↵
R8↵	V5↵	↵	ARM状态局部变量寄存器5↵
R7↵	V4↵	wr↵	局部变量寄存器4,Thumb状态工作寄存器↵
R6↵	V3↵	↵	局部变量寄存器3↵
R5↵	V2↵	↵	局部变量寄存器2↵
R4↵	V1↵	↵	局部变量寄存器 2↵
R3↵	A4↵	↵	参数/结果/ scratch 寄存器 4↵
R2↵	A3↵	↵	参数/结果/ scratch寄存器3↵
R1↵	A2↵	↵	参数/结果/ scratch寄存器2↵
R0↵	A1↵	↵	参数/结果/ scratch寄存器 1↵

② 数据栈使用规则

栈顶指针指向及栈增长方向：

- 指向的栈顶不为空，称为 FULL 栈；
- 指向的为空数据单元，称为 EMPTY 栈；
- 向内存地址减小的方向增长时，称为 DESCENDING；
- 向内存地址增加的方向增长时，称为 ASCENDING。

综合这两种特点可以有以下4种数据栈：

- **FD Full Descending**
- ED Empty Descending
- FA Full Ascending
- EA Empty Ascending

ATPCS规定数据栈为 **FD模式**。

③ 数传递规则

参数个数可变的子程序参数传递规则：

- 对于参数个数可变的子程序，当参数不超过4个时，可以使用寄存器R0~R3来传递参数；当参数超过4个时，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放在连续的内存字单元中的字数据。然后，依次将各字数据传送到寄存器R0、R1、R2、R3中，如果参数多于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。

子程序结果返回规则：

- 结果为一个32位的整数时，可以通过寄存器R0返回。
- 结果为一个64位整数时，可以通过寄存器R0和R1返回，依次类推。
- 更多位数时，使用内存传递。

数传递规则总结

- 当参数不超过4个时，可以使用寄存器R0~R3来传递参数
- 当参数超过4个时，还可以使用数据栈来传递参数（注意入栈的顺序与参数顺序的关系）。
- 返回结果通过r0—r3传递。

例题：关注函数声明和参数传递

例1：C调汇编

例2：C程序中内嵌汇编

例3：汇编调C：

主要是参数传递过程

例4：综合

C程序运行环境初始化：包括系统初始化和代码搬移

初始化结束后从汇编跳到c

C中调用汇编（参数如何传递的）

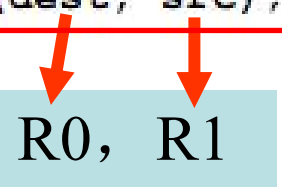
1) C程序调用汇编程序

- 汇编程序编写也要遵循ATPCS规则，以保证程序调用时参数正确传递；
- 首先在汇编程序中用EXPORT伪指令声明被调用的子程序，表示该子程序将在其他文件中被调用；
- 然后在C程序中使用extern关键字声明要调用的汇编子程序为外部函数。

例1 C调汇编

- ▶ 在汇编程序中用export name来定义，C中用**extern**关键字声明外部函数
- ▶ 在C程序中直接调用
- ▶ 一般的链接即可

```
extern void mystrcopy(char *d, const char *s);  
int main(void)  
{  
    const char *src = "Source";  
    char dest[10];  
  
    ...  
    mystrcopy(dest, src);  
    ...  
}
```



R0, R1

C程序文件

```
AREA StringCopy, CODE, READONLY  
EXPORT mystrcopy  
  
mystrcopy  
    LDRB r2, [r1], #1  
    STRB r2, [r0], #1  
    CMP r2, #0  
    BNE mystrcopy  
    MOV pc, lr  
  
END
```

汇编程序文件

2) C程序中内嵌汇编语句

- 在C语言中内嵌汇编语句可以实现一些高级语言不能实现或者不容易实现的功能。
- 对于时间紧迫的功能也可以通过在C语言中内嵌汇编语句来实现。
- 内嵌的汇编器支持大部分ARM指令和Thumb指令。

Note: 使用关键字 asm 来在C程序中加入一段汇编语言程序。

两个下划线



例2 C程序中内嵌汇编

- ▶ 允许使用一些不能由编译器自动生成的指令:
 - ▶ MSR / MRS
 - ▶ 新的指令
 - ▶ 协处理器指令
- ▶ 通常在关联的内嵌函数中使用
- ▶ 使用C变量代替寄存器
 - ▶ 不是一个真正的汇编文件
 - ▶ 通过优化器实现
- ▶ ADS FAQ 入口 “Using the Inline Assembler”

使用关键字 **asm** 来在C程序中加入一段汇编语言程序。

```
#define Q_Flag 0x08000000 // Bit 27

__inline void Clear_Q_flag (void)
{
    int temp;
    asm
    {
        MRS temp, CPSR
        BIC temp, temp, #Q_Flag
        MSR CPSR_f, temp
    }
}

__inline int mult16(short a,
                   short b, int c)
{
    int temp;
    asm
    {
        SMLABB temp, a, b, c
    }
    return temp;
}
```

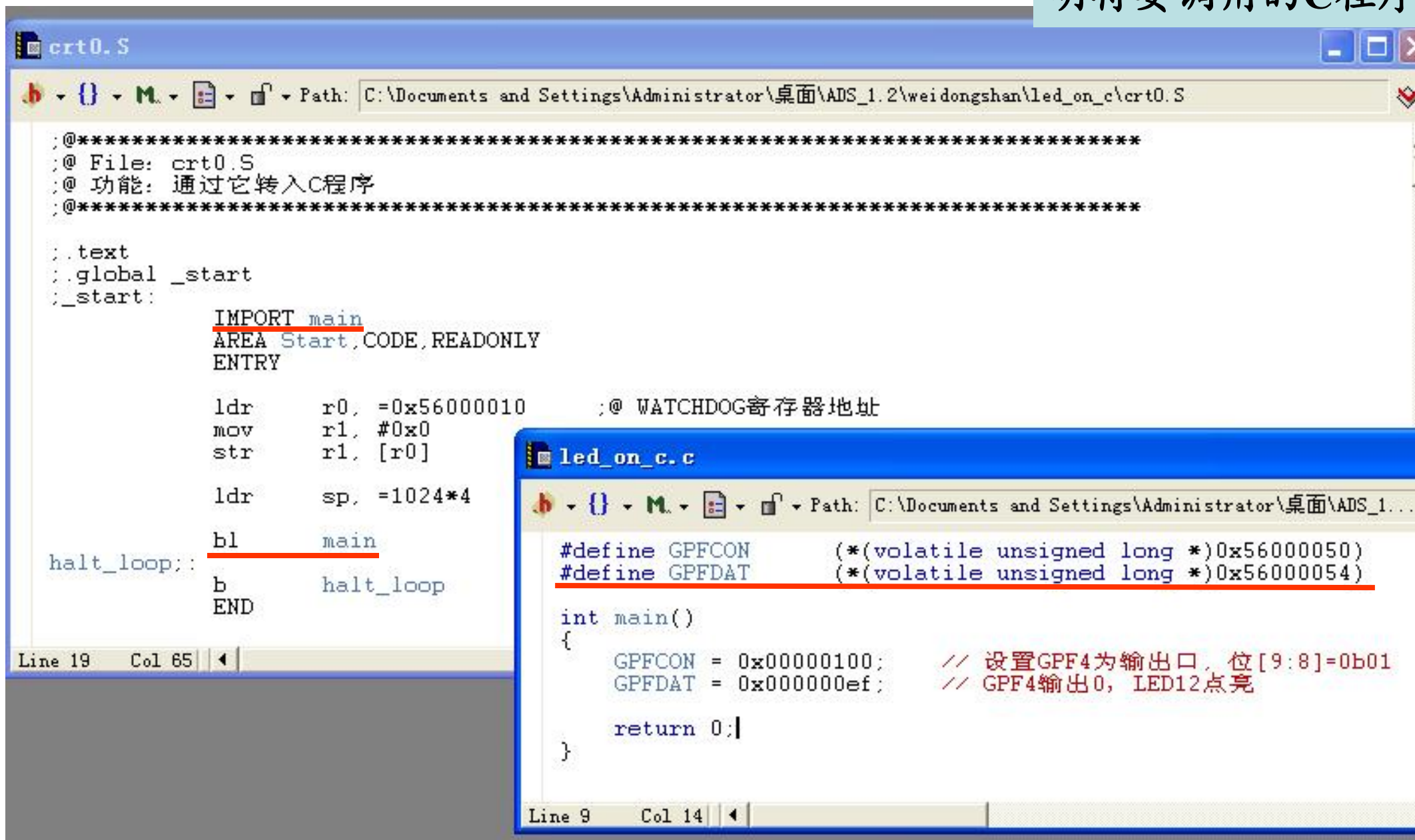
内嵌可通过删除子函数调用的开销来提高性能，这个 **inline** 关键字显示哪个函数将被内嵌

3) 汇编程序调用C语言

- 汇编程序编写要遵循ATPCS规则，以保证程序调用时参数正确传递。
- 首先在汇编程序中使用IMPORT伪指令事先声明将要调用的C语言函数。
- 然后通过BL指令来调用C函数。

例3 汇编调C

汇编中用IMPORT伪操作声明将要调用的C程序函数



The image shows two overlapping windows from a development environment. The top window, titled 'crt0.S', contains assembly code. It starts with a comment block in Chinese, followed by a text segment declaration, a global symbol for '_start', and an entry point. The code uses the 'IMPORT' directive to bring in the 'main' function from 'led_on_c.c'. It then initializes a register 'r0' with the address 0x56000010 (labeled as the watchdog register address), sets 'r1' to 0, and stores it at the address in 'r0'. It then sets the stack pointer 'sp' to 1024*4 and branches to 'main'. A 'halt_loop' label is followed by a 'b' (branch) instruction to 'halt_loop' and an 'END' instruction. The bottom window, titled 'led_on_c.c', contains C code. It defines two macros, 'GPFCON' and 'GPFDAT', with volatile unsigned long pointers to memory addresses 0x56000050 and 0x56000054 respectively. The 'main' function sets 'GPFCON' to 0x00000100 (commented as setting GPF4 as an output, bits [9:8] = 0b01) and 'GPFDAT' to 0x000000ef (commented as GPF4 output 0, LED12 lights up), and then returns 0.

```
crt0.S
;@*****
;@ File: crt0.S
;@ 功能: 通过它转入C程序
;@*****

.text
.global _start
_start:
    IMPORT main
    AREA Start, CODE, READONLY
    ENTRY

    ldr    r0, =0x56000010    ;@ WATCHDOG寄存器地址
    mov    r1, #0x0
    str    r1, [r0]

    ldr    sp, =1024*4

    bl     main
halt_loop:
    b      halt_loop
END

Line 19  Col 65
```

```
led_on_c.c
#define GPFCON      (*(volatile unsigned long *)0x56000050)
#define GPFDAT      (*(volatile unsigned long *)0x56000054)

int main()
{
    GPFCON = 0x00000100;    // 设置GPF4为输出口, 位[9:8]=0b01
    GPFDAT = 0x000000ef;    // GPF4输出0, LED12点亮

    return 0;
}

Line 9  Col 14
```

■ 寄存器的操作 #define A (*(volatile unsigned long *) 0x48000000)

- 若要向寄存器A（地址假定为0x48000000）写入数据0x01，那么就可以这样设置了：

```
#define A  (* (volatile unsigned long *) 0x48000000 )
```

```
A = 0x01;
```

```
#define GPFCON (*(volatile unsigned long *)0x56000050)
#define GPFDAT (*(volatile unsigned long *)0x56000054)
```

■ 寄存器的操作 #define A (*(volatile unsigned long *) 0x48000000)

- 若要向寄存器A（地址假定为0x48000000）写入数据0x01，那么就可以这样设置了：

```
#define A (*(volatile unsigned long *) 0x48000000 )
```

```
A = 0x01;
```

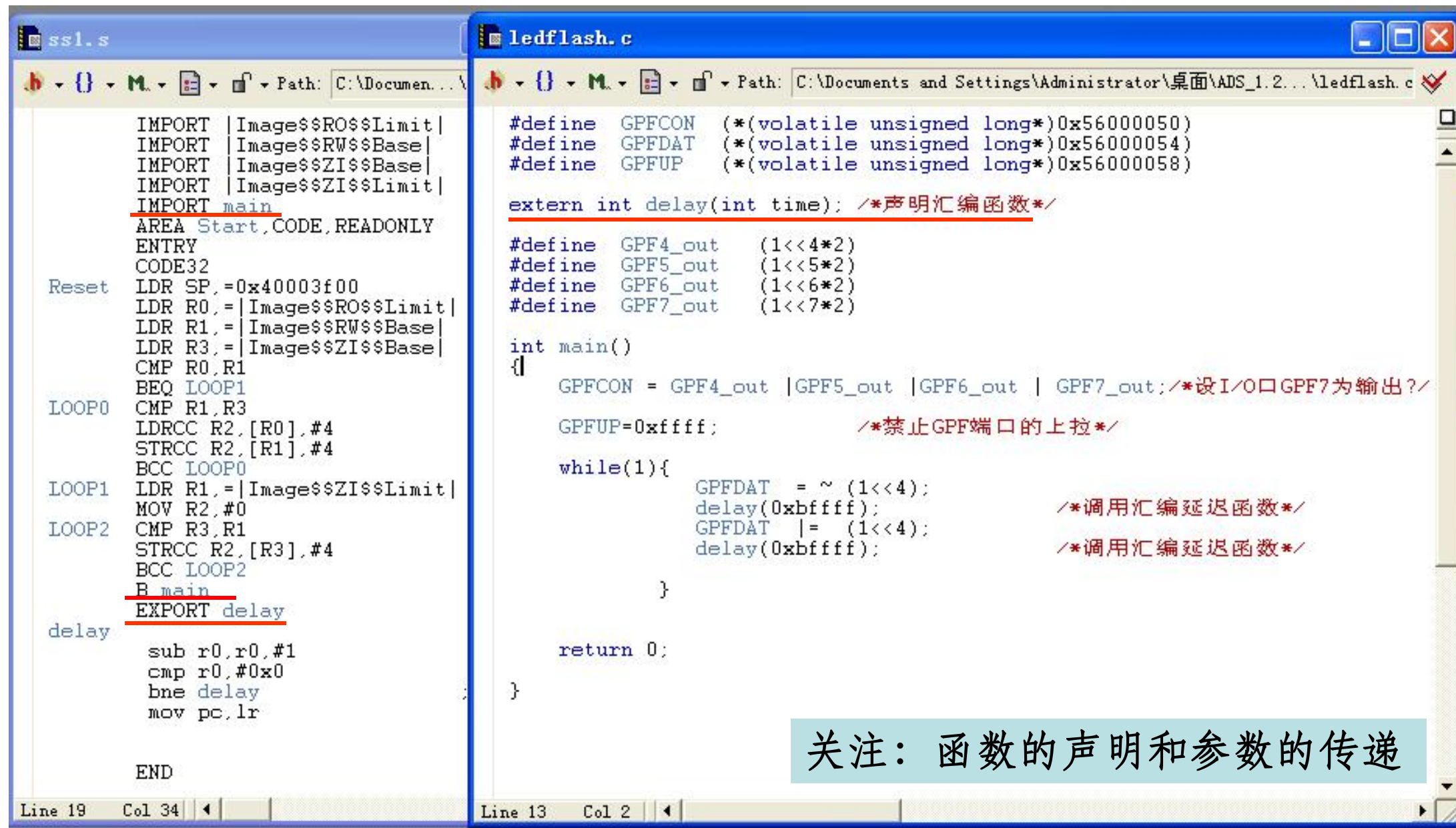
- **volatile**关键字说明这变量可能会被意想不到地由硬件改变。
volatile 限定编译器不对这个指针指向的存储单元进行优化,即**不用通用寄存器暂时代替**这个指针指向的存储单元,而是每次取值都直接到指针指向的存储单元取值。
- (volatile unsigned long *) 0x48000000: 把0x48000000强制转换成volatile unsigned long类型的指针, 即对指针的操作的范围是从0x48000000开始的4个字节(long型), 暂记为p。
- 那么就是#define A *p, 即A为P指针指向位置的内容了。这里就是通过内存寻址访问到寄存器A, 可以读/写操作!

```
#define GPFCON (*(volatile unsigned long *)0x56000050)
#define GPFDAT (*(volatile unsigned long *)0x56000054)
```

- **volatile** 限定编译器不对这个指针的指向的存储单元进行优化,即不用通用寄存器暂时代替这个指针的指向的存储单元,而是每次取值都直接到指针的指向的存储单元取值.
- **volatile** 主要用于变量会异步改变的情况下
主要有三个方面:
 - 1.cpu外设**寄存器**
 - 2.中断和主循环都会用到的**全局变量**
 - 3.操作系统中的线程间都会用到的**公共变量**.

例4 综合运用

实际应用当中程序的初始化部分由汇编语言完成，然后用C语言完成主要编程任务。程序执行完初始化后，跳到C程序执行。



The image shows two side-by-side code editors. The left editor, titled 'ssl.s', contains assembly code for an ARM processor. It includes imports for memory limits and base addresses, a reset routine to initialize registers, two loops (LOOP0 and LOOP1) for memory initialization, and a delay function. The right editor, titled 'ledflash.c', contains C code that defines GPIO pins, declares an external delay function, and implements a main loop that toggles an LED pin with delays. Both editors show the file path as 'C:\Documents and Settings\Administrator\Desktop\ADS_1.2...\ledflash.c'.

```
ssl.s
IMPORT Image$$RO$$Limit|
IMPORT Image$$RW$$Base|
IMPORT Image$$ZI$$Base|
IMPORT Image$$ZI$$Limit|
IMPORT main
AREA Start, CODE, READONLY
ENTRY
CODE32
Reset LDR SP, =0x40003f00
      LDR R0, =Image$$RO$$Limit|
      LDR R1, =Image$$RW$$Base|
      LDR R3, =Image$$ZI$$Base|
      CMP R0, R1
      BEQ LOOP1
LOOP0  CMP R1, R3
      LDRCC R2, [R0], #4
      STRCC R2, [R1], #4
      BCC LOOP0
LOOP1  LDR R1, =Image$$ZI$$Limit|
      MOV R2, #0
LOOP2  CMP R3, R1
      STRCC R2, [R3], #4
      BCC LOOP2
      B main
EXPORT delay
delay  sub r0, r0, #1
      cmp r0, #0x0
      bne delay
      mov pc, lr

END

ledflash.c
#define GPFCON (*(volatile unsigned long*)0x56000050)
#define GPFDAT (*(volatile unsigned long*)0x56000054)
#define GPFUP  (*(volatile unsigned long*)0x56000058)

extern int delay(int time); /*声明汇编函数*/

#define GPF4_out (1<<4*2)
#define GPF5_out (1<<5*2)
#define GPF6_out (1<<6*2)
#define GPF7_out (1<<7*2)

int main()
{
    GPFCON = GPF4_out | GPF5_out | GPF6_out | GPF7_out; /*设I/O口GPF7为输出*/
    GPFUP = 0xffff; /*禁止GPF端口的上拉*/

    while(1){
        GPFDAT = ~(1<<4);
        delay(0xbffff); /*调用汇编延迟函数*/
        GPFDAT |= (1<<4);
        delay(0xbffff); /*调用汇编延迟函数*/
    }

    return 0;
}
```

关注：函数的声明和参数的传递

ARM汇编程序设计

- 一、ARM汇编伪指令和伪操作
- 二、ARM汇编语言程序设计
- 三、汇编语言与C语言的混合编程
- 四、ARM映像文件

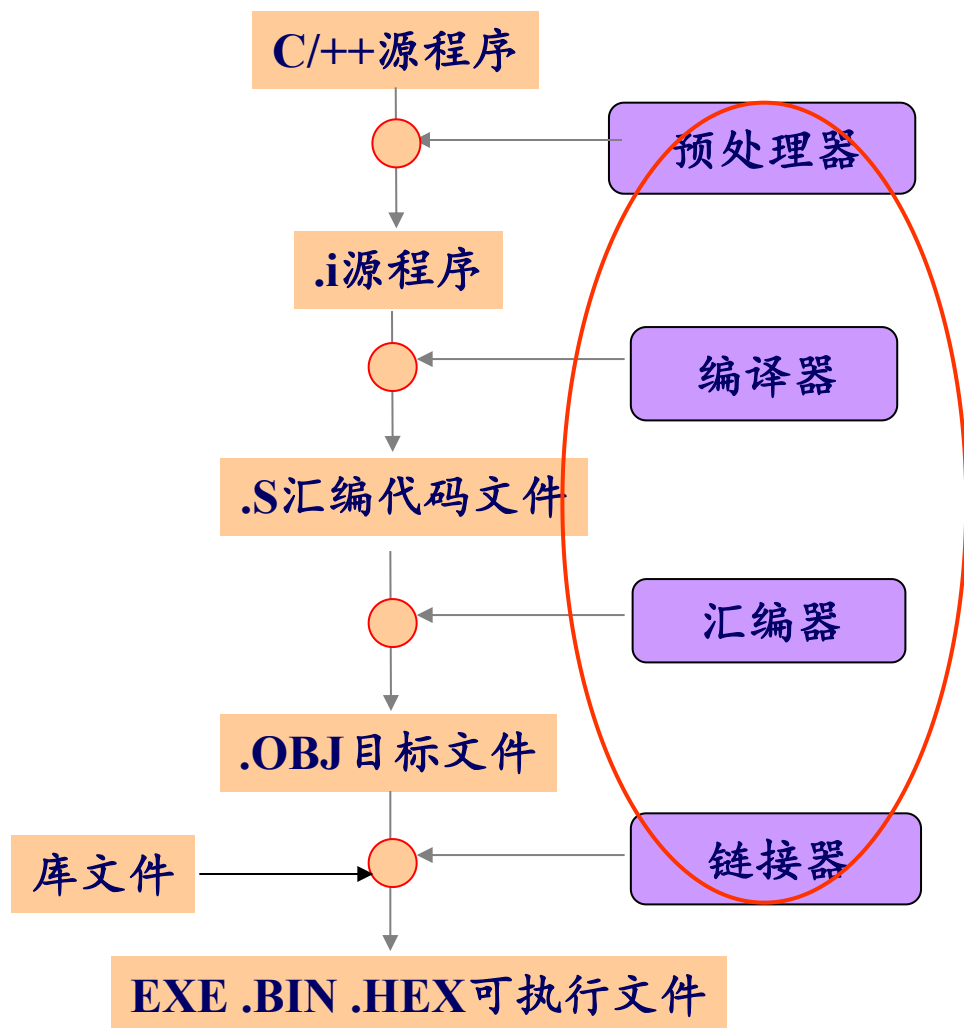
四、ARM映像文件

1. ARM映像文件

映像文件其实就是可执行文件，包括如下格式：

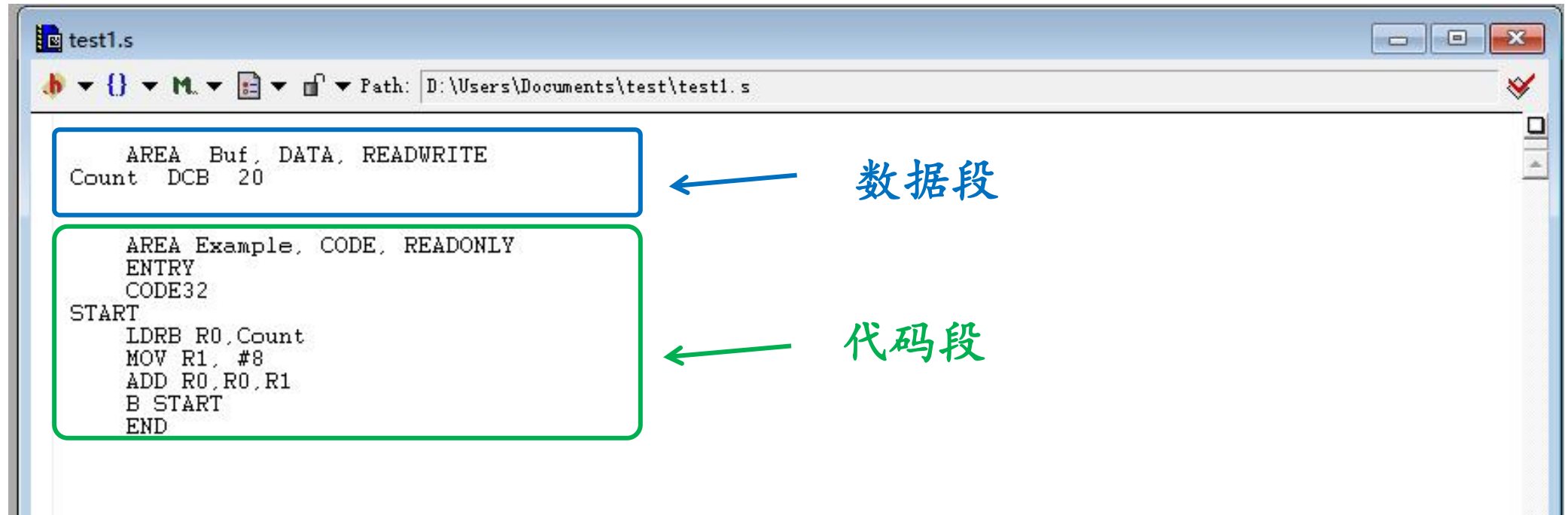
- **bin或hex格式**：可以直接烧到rom里执行。
- **Axf格式**：在axd调试过程中，调试的是axf文件，其实这也是一种映像文件，它只是在bin文件中加了一个文件头和一些调试信息。
- **ELF格式**：arm映像文件是ELF格式。ELF是UNIX系统实验室开发和发布的Executable and Linking Format(ELF)二进制格式。它主要有三种文件类型：
 - 1) 可执行文件；
 - 2) 可重定位文件；
 - 3) 共享object文件（又叫共享库）。

源程序文件↵	文件名↵	说 明↵
汇编程序文件↵	*.S↵	用ARM汇编语言编写的ARM程序或Thumb程序。↵
C程序文件↵	*.C↵	用C语言编写的程序代码。↵
头文件↵	*.H↵	为了简化源程序，把程序中常用到的常量命名、宏定义、数据结构定义等等单独放在一个文件中，一般称为头文件。↵



•一般来说，ARM源程序文件经过编译（包括预处理、编译、汇编和链接）处理后生成可执行的映像文件（ELF格式或bin等格式）。

- 一个源程序包括只读的代码段和可读写的数据段，称为输入段；



- 源程序文件经过预处理、编译、汇编和链接处理后生成可执行的映像文件。

- 经编译后生成可执行文件中的RO段、RW段和ZI段，称为输出段。
 - 源文件中的指令以及常量被编译后是RO数据类型；
 - 源文件中已初始化成非0值的变量编译后是RW类型数据；
 - 源文件中未初始化或初始化为0的变量编译后是ZI类型数据。

- 经编译后生成可执行文件中的ro段、rw段和zi段，称为输出段。

Metrowerks CodeWarrior for ARM Developer Suite v1.2 - [Errors & Warnings]

File Edit View Search Project Debug Window Help

0 0 13 Errors and warnings for test.mcp

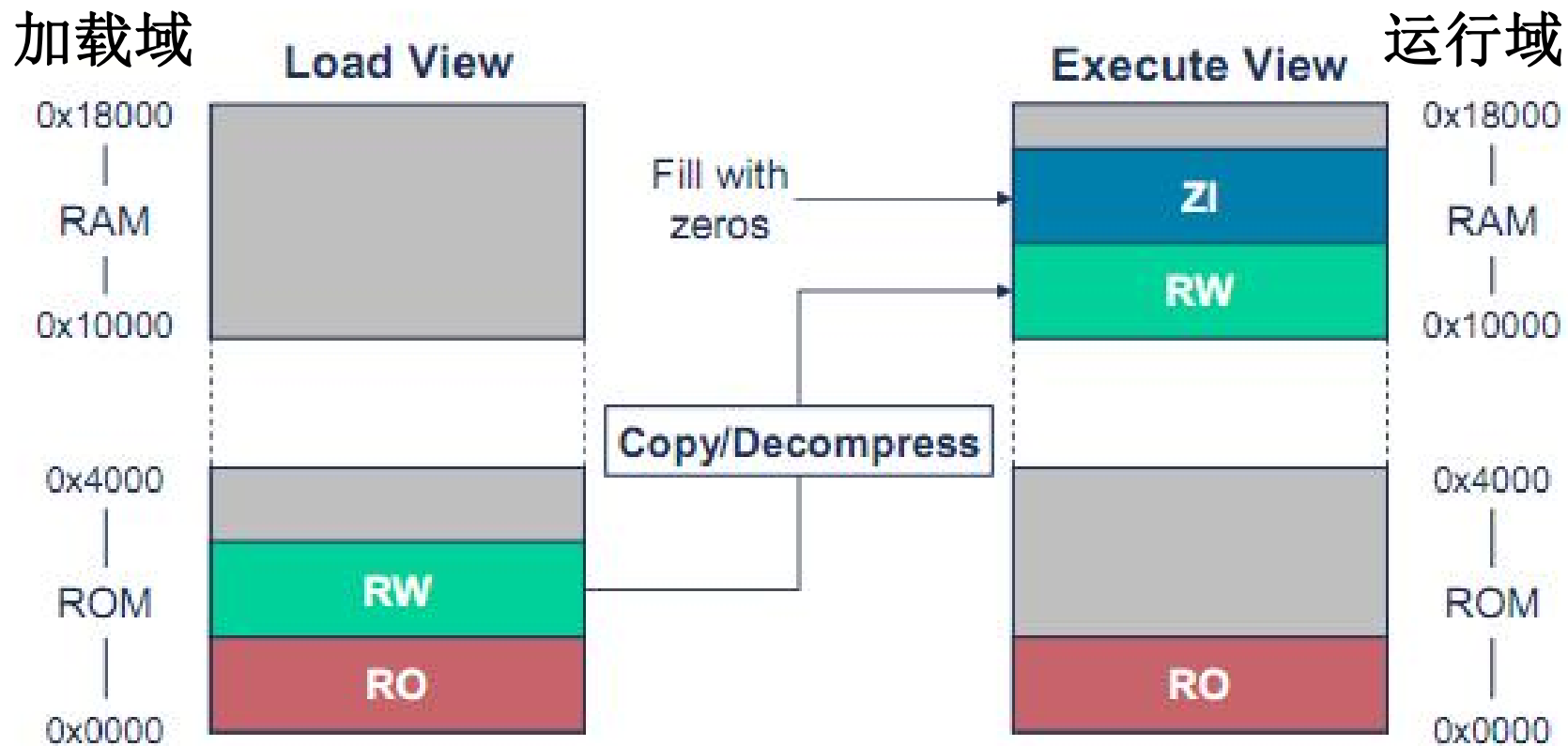
=====
Image component sizes
=====
Code RO Data RW Data ZI Data Debug
16 0 4 0 204 Object Totals
0 0 0 0 0 Library Totals
=====
Code RO Data RW Data ZI Data Debug
16 0 4 0 204 Grand Totals
=====
Total RO Size(Code + RO Data) 16 (0.02kB)
Total RW Size(RW Data + ZI Data) 4 (0.00kB)
Total ROM Size(Code + RO Data + RW Data) 20 (0.02kB)
=====

2. ARM 映像文件各组成部分的地址映射

ARM 映像文件各组成部分在存储系统中的地址映射有两种:

- 当映像文件位于存储器中时(即映像文件还没有执行之前), 称为加载地址, 位于加载域;
- 当映像文件正在运行时的地址称为运行地址, 位于运行域。

- 对于嵌入式系统而言, 程序映像都是存储在Flash存储器等一些非易失性器件中的, 而在运行时, 程序中的RW段必须在被访问前重新装载到可读写的RAM中, 以保证程序正常运行。
- ROM主要指: NAND Flash, Nor Flash
- RAM主要指: PSRAM, SDRAM, SRAM, DDRAM



- RO code and data stays in ROM
- C library initialization code (in `__main`) will :
 - Copy/decompress RW data from ROM to RAM
 - Initialize the ZI data in RAM to zero

← 初始化用户执行环境

对于加载域中的输出段，一般来说ro段后面紧跟着rw段，rw段后面紧跟着zi段。在运行域中这些输出段并不连续，但rw和zi一定是连着的。zi段和rw段中的数据其实可以是rw属性。

总结：

- 一个ARM可执行的映像文件由**RO**、**RW**、**ZI**三个段组成。
其中：
 - ✓ 指令以及常量被编译后是**RO**类型数据。
 - ✓ 已初始化成非0值的全局变量编译后是**RW**类型数据。
 - ✓ 未初始化或初始化为0的全局变量编译后是**ZI**类型数据。
- 映像文件一开始总是存储在ROM/Flash里面的，其RO部分既可以在ROM/Flash中运行，也可以转移到速度更快的RAM中执行，而RW和ZI这两部分是必须转移到可写的RAM里面去。
- 所谓**应用程序执行环境的初始化**，就是完成必要的从ROM到RAM的数据传输和清零。即是把**RO**、**RW**、**ZI**三段拷贝到指定的运行地址处，并将**ZI**段清零。

- 下列几个变量是**编译器通知**的，它们指示了在**运行域中**各个输出段所处的地址空间：

|Image\$\$RO\$\$Base| ;RO段起始地址

|Image\$\$RO\$\$Limit| ;RO段结束地址+1

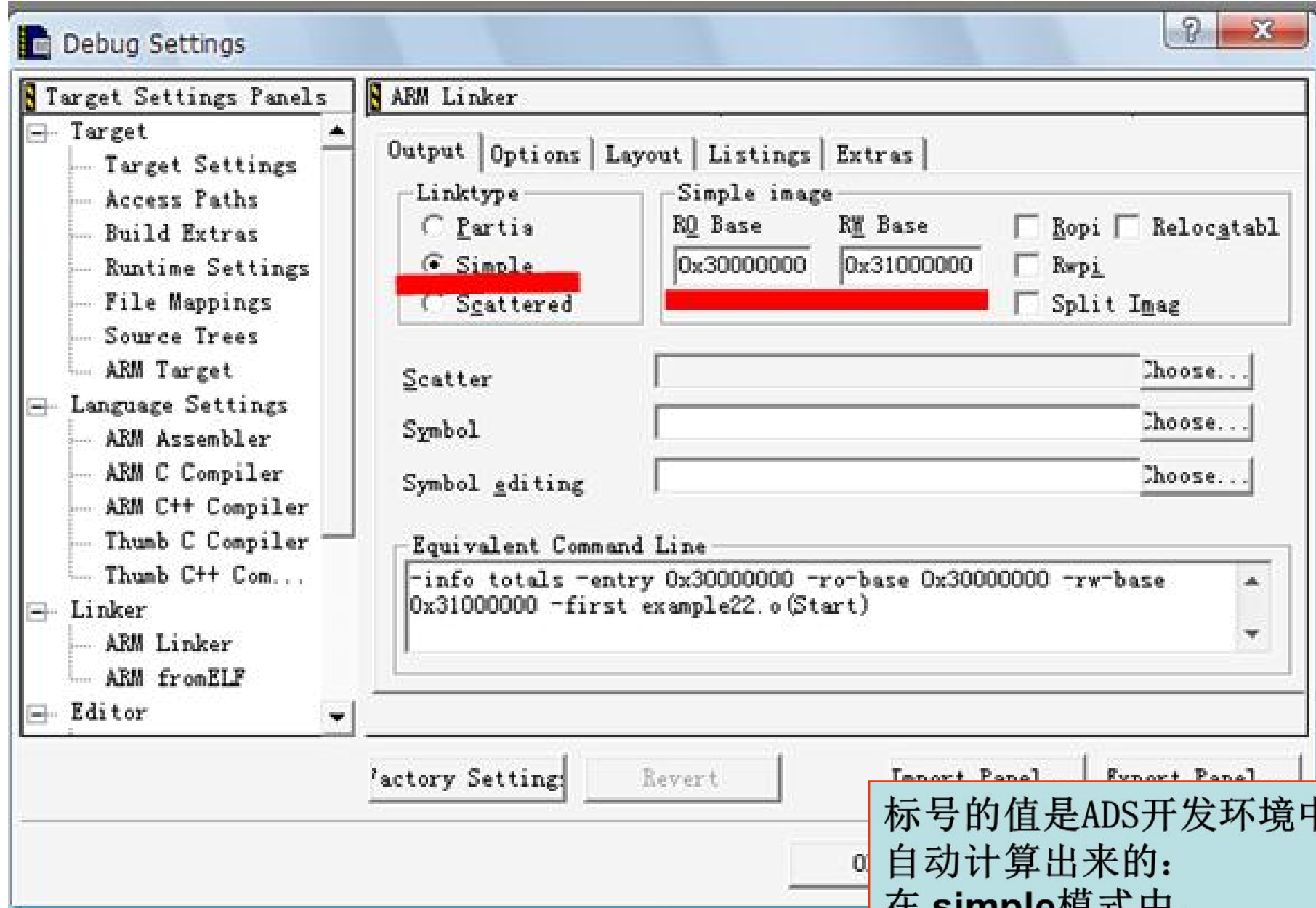
|Image\$\$RW\$\$Base| ;RW段执行域起始地址

|Image\$\$RW\$\$Limit| ;RW段结束地址+1

|Image\$\$ZI\$\$Base| ;ZI段起始地址

|Image\$\$ZI\$\$Limit| ;ZI段结束地址+1

- 对于复杂情况，如RO段被分成几部分并映射到存储空间的多个地方时，需要创建一个称为“分布装载描述文件”的文本文件，通知连接器把程序的某一部分连接在存储器的某个地址空间。



标号的值是ADS开发环境中根据设置值，由编译器自动计算出来的：
在 **simple** 模式中，
ro base=|Image\$\$RO\$\$Base|
rw base =|Image\$\$RW\$\$Base|

应用程序执行环境的初始化

```
BaseOfROM DCD | Image $ $ RO $ $ Base|
TopOfROM DCD | Image $ $ RO $ $ Limit|
BaseOfBSS DCD | Image $ $ RW $ $ Base|
BaseOfZero DCD | Image $ $ ZI $ $ Base|
EndOfBSS vDCD | Image $ $ ZI $ $ Limit|
```

```
ldr r0, ResetEntry ;ResetEntry 是
ldr r2, BaseOfROM
cmp r0, r2
ldreq r0, TopOfROM ;TopOfROM = 0x
beq InitRam
```

```
ldr r3, TopOfROM
part 1,通过比较,将 ro 搬到 SDRAM 里,搬到的
$ $ RO $ $ Limit|结束
```

```
0
luma r0!, {r4 - r7}
stmia r2!, {r4 - r7}
cmp r2, r3
bcc %B0;
```

为RO, RW, ZI段执行起止地址赋值

RO执行地址
=0x00000000

Y

N

将RO段从加载域
搬运至执行域

part1

将RW段从加载域
搬运至执行域

part2

在RW段后开辟ZI
段空间并清0

part3

```

;part 2,搬 rw 段到 SDRAM,目的地址从|Image $ $ RW $ $ Base|开始
sub r2, r2, r3;r2 = 0
sub r0, r0, r2
InitRam                ;carry rw to baseofBSS
ldr r2, BaseOfBSS      ;TopOfROM = 0x30001de0,base
ldr r3, BaseOfZero     ;BaseOfZero = 0x30001de0
0
cmp r2, r3
ldrcc r1, [r0], #4
strcc r1, [r2], #4
bcc %B0

```

```

;part 3,将 SDRAM zi 初始化为 0,地址从|Image $ $ ZI $ $ Base|到|
mov r0, #0;init 0
ldr r3, EndOfBSS      ;EndOfBSS = 30001e40
1
cmp r2, r3
strcc r0, [r2], #4
bcc %B1
...

```

