ARM汇编程序语言设计

（教学使用，请勿转载）

By Li Yang

E-Mail:liyangth@gmail.com

一．内嵌汇编语法

使用内嵌汇编，要先编写汇编指令模板，然后将C语言表达式与指令的操作数相关联，并告诉GCC对这些操作有哪些限制条件。
例如在下面的汇编语句：

```
__asm__ __volatile__ ("movl %1,%0" : "=r" (result) : "r" (input));
```

说明：
"movl %1,%0\n\t"是指令模板；
"%0"和"%1"代表指令的操作数，称为占位符，内嵌汇编靠它们将C 语言表达式与指令操作数相对应。

指令模板后面用小括号括起来的是C语言表达式，本例中只有两个：“result”和“input”，他们按照出现的顺序分 别与指令操作数“%0”，“%1”对应；

注意对应顺序：第一个C 表达式对应“%0”；第二个表达式对应“%1”，
依次类推，操作数至多有10 个，分别用“%0”,“%1”….“%9”表示。
在每个操作数前面有一个用引号括起来的字符串，字符串的内容是对该操作数的限制或者说要求。

```
__asm__ __volatile__(
    "mov %0,%1\n\t"

    操作限制,r（只读寄存器）， m(内存)， =r(可写，左值),+&r(读写，volatile)
    : "+&" (a)      //输出部分：结果寄存器Rd;左值;
    : "r" (b)       //输入部分：Rn,Rm; 右值;
    : r1         //物理寄存器，指出这个寄存器我修改过.

);
```

. 小结：
1. 输入部分输出部分
   . "r" 输入部分，在进入asm前生成一段代码，用c变量更新占位符；
   . 输入部分可以写常数；
   . "=r"输出部分，在退出asm后生成一段代码，用占位符更新c变量；

2. 修饰符
   . "+r"可读可写，一般在输出部分，在前后都更新；
     "+"可读可写，如果为"="则不能成为输入，即在进入asm前没有对作为输入部分的a(%0)生成代码；
   . "&=r"， 影子寄存器
     "&"为每个占位符分配一个影子寄存器，不会用同一个寄存器来优化
3. 破坏部分
   破坏：指出，告诉在进入__asm__前保存，出来后恢复这个你以后还会用到的物理寄存器。

二．汇编程序设计语言

1．ARM数据处理指令

ARM的数据处理指令由3类组成:

- 数据传输
- 算术逻辑运算
- 乘法指令

特点:

1．ARM CPU只能处理内部的寄存器，所以，ARM数据处理指令只能对寄存器的内容进行操作．

2．所有 ARM 数据处理指令均可选择使用 S 后缀,以影响状态标志.比较指令 CMP,CMN,TST 和 TEQ 不需要后缀 S,它们会直接影响状态标志.

1.1. 数据传输指令

MOV

将 8 位图立即数或寄存器(operant2)传送到目标寄存器 Rd,可用于移位运算等操作.指令格式如下:

e.g:

```
MOV R1,#10     ;R1=10
MOV R1,R2      ;R1=R2
```

MVN

将 8 位图立即数或寄存器(operand2)按位取反后传送到目标寄存器(Rd),因为其具有取反功能,所以可以装载范围更广的立即数.指令格式如下:

e.g:

```
MVN R1,#0XFF      ;R1=0xFFFFFF00
MVN R1,R2      ;R1=~R2
```

1.2. 算术逻辑运算

1） 算术**"+""-""*"**法:

ADD

加法运算指令.将 operand2 数据与 Rn 的值相加,结果保存到 Rd 寄存器.指令格式如下:

e.g:

```
ADD   R1,R1,R2       ;R1=R1+R2
ADD R3,R1,R2,LSL #2  ;R3=R1+R2<<2   (R3=R1+(R2*4))
```

SUB

```
SUBS R1,R1,#1        ;R1=R1-1
SUB R1,R1,R2         ;R1=R1-R2
```

MUL

```
  MUL   R1,R2,R3    ;R1=R2×R3
```

2）逻辑运算(位操作： &, |, ^, &~)

AND

位操作，取出某一位;

```
e.g:
AND R0,R0,#x01    ;R0=R0&0x01 取出R0的最低位
```

ORR

位操作: 置某位值;

```
e.g:
ORR R0,R0,#x01    ;R0=R0|0x01 将R0的最低位置"1"
ORR R0,R0,#x0F    ;将R0的低4位置"1"
```

EOR

按位异或;

```
e.g:
EOR R0,R1,R2      ;R0=R1^R2
```

BIC

位清除(&~);

```
e.g:
BIC R1,R1,#x0F    ;将R1的低4位清零,其他
```

TST

位测试指令.测试某位是否为"0",清位;

指令将寄存器 Rn 的值与 operand2 的值按位作逻辑与操作,根据操作的

结果理新 CPSR 中相应的条件标志位,以便后面指令根据相应的条件标志来判断是否执

行.指令格式如下:

```
TST{cond}   Rn,operand2
```

TST 指令举例如下:

```
TST   R0,#0x01    ;判断 R0 的最低位是否为 0
TST   R1,#0x0F    ;判断 R1 的低 4 位是否为 0
```

小结:

TST 指令与 ANDS 指令的区别在于 TST4 指令不保存运算结果.TST 指令通常于 EQ,NE 条件码配合使用,当所有测试位均为 0 时,EQ 有效,而只要有一个测试为不为 0,则 NE 有效.

2. 程序流程

2.1 CPSR 状态寄存器

a) MRS 读状态寄存器指令.

在 ARM 处理器中,只有 MRS 指令可以状态寄存器 CPSR 或 SPSR 读出到通用寄存器中.

指令格式如下;

```
MRS{cond} Rd ,psr
```

其中;

Rd 目标寄存器.Rd 不允许为 R15.

psr CPSR 或 SPSR

指令举例如下

```
MRS R1,CPSR ;将 CPSR 状态寄存器读取,保存到 R1 中
MRS R2,SPSR ;将 SPSR 状态寄存器读取,保存到 R2 中
```

小结:

MRS 指令读取 CPSR,可用来判断 ALU 的状态标志,或 IRQ,FIQ 中断是否允许等;在异常处理程序中,读 SPSR 可知道进行异常前的处理器状态等.MRS 与 MSR 配合使用,实现 CPSR 或 SPSR 寄存器的读–修改---写操作,可用来进行处理器模式切换(),允许/禁止 IRQ/FIQ 中断等设置.另外,进程切换或允许异常中断嵌套时,也需要使用 MRS 指令读取 SPSR 状态值.保存起来.

b) 位域的说明

参考《ARM Architecture Reference Manual.pdf》第 49 页
A2.5 Program status registers 章节.

c) MSR 写状态寄存器指令.

在 ARM 处理器中.只有 MSR 指令可以直接设置状态寄存器 CPSR 或 SPSR.

指令格式如下:

```
MSR{cond} psr_fields,#immed_8r
MSR{cond} psr_fields,Rm
```

说明:

1) psr 是CPSR 或 SPSR
2) fields 指定传送的区域.
   Fields 可以是以下的一种或多种(字母必须为小写):
   - c 控制域屏蔽字节(psr[7...0])
   - x 扩展域屏蔽字节(psr[15...8])
   - s 状态域屏蔽字节(psr[23....16])
   - f 标志域屏蔽字节(psr[31...24])
3) immed_8r 要传送到状态寄存器指定域的立即数,8 位.
4) Rm 要传送到状态寄存器指定域的数据的源寄存器.

MSR 指令举例如下:

```
MSR  CPSR_c,#0xD3 ;CPSR[7...0]=0xD3,即(110 10011)禁止irq,fiq,arm,
```

切换到管理模式.

```
MSR  CPSR_cxsf,R3 ;CPSR=R3
```

注意:

1. 只有在特权模式下才能修改状态寄存器.
2. 程序中不能通过 MSR 指令直接修改 CPSR 中的 T 控制位来实现 ARM 状态/Thumb 状态的切换,必须使用 BX 指令完成处理器状态的切换(因为 BX 指令属转移指令,它会打断流水线状态,实现处理器状态切换).

举例:

1.使能 IRQ 中断

```
ENABLE_IRQ
    MRS R0,CPSR
    BIC R0.R0,#0x80
    MSR CPSR_c,R0
    MOV PC,LR
```

2. 禁能 IRQ 中断

```
DISABLE_IRQ
   MRS   R0,CPSR
   ORR   R0,R0,#0x80
   MSR   CPSR_c,R0
   MOV   PC,LR
```

2.2 条件执行

a) 条件码

参考《ARM Architecture Reference Manual.pdf》第112页A3.2.1 Condition code 0b1111 章节

b) 条件码与 **CPSR** 标志位的关系

在ARM体系中，所有ARM指令均可条件执行，设置了一些条件执行码，但这些条件码对应设置CPSR中的Z, C, N, V标志位:

0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z set (unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (greater or equal)
1011 = LT - N set and V clear, or N clear and V set (less than)
1100 = GT - Z clear, and either N set and V set, or N clear and V clear (greater than)
1101 = LE - Z set, or N set and V clear, or N clear and V set (less than or equal)
1110 = AL - always
1111 = NV - never

举例说明:
怎样去理解这些设置呢？拿**1001 = LS - C clear or Z set (unsigned lower or same)**来说:
为什么 LS 就对应 C clear and Z set。看下面的例子:

```
MOV R0, #5
MOV R1, #6
CMP R0, R1
MOVLS R2, R0 ; if R0 < R1 则 将小值存入 R2 中
```

在这个例子中，MOVLS 能够正确执行的条件 C=0 and z=1 成立，是通过 CMP 设置了，看看 reference manual 中的 CMP 设置规则:

C flag:For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
Z flag: Is set to 1 if the result of the instruction is zero (which often indicates an equal result from
a comparison), and to 0 otherwise.

这就不难理解了 LS 与标志位的对应了。下面给出标志位的设置规则，便于我们从中推导上面每一个例子。

In either case, the new condition code flags (after the instruction has been executed) usually mean:
N Is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement
signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.
Z Is set to 1 if the result of the instruction is zero (which often indicates an equal result from
a comparison), and to 0 otherwise.
C Is set in one of four ways:
• For an addition, including the comparison instruction CMN, C is set to 1 if the addition
produced a carry (that is, an unsigned overflow), and to 0 otherwise.
• For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
• For non-addition/subtractions that incorporate a shift operation, C is set to the last bit
shifted out of the value by the shifter.
• For other non-addition/subtractions, C is normally left unchanged (but see the individual instruction descriptions for any special cases).
V Is set in one of two ways:
• For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the
operands and result as two's complement signed integers.
• For non-addition/subtractions, V is normally left unchanged (but see the individual instruction descriptions for any special cases).

来看下 GT (signed greater)标志位的设置:

```
 CMP -5, -4    ; 相减为不等于 0，为负值，且有下溢出，所以 N=1, V=1
 CMP 6,5       ; 相减后值为 1（N=0），正值且无下溢（V=0）
```

2.3 比较指令

CMP

指令使用寄存器 Rn 的值减去 operand2 的值,根据操作的结果理新 CPSR 中的相应条件标志位,以便后面的指令根据相应的条件标志来判断是否执行.指令格式如下:

e.g:

```
CMP R1,#10                ;判断R1是否等于"10"
CMP R1,R2
```

小结:

CMP 指令与 SUBS 指令的区别在于 CMP 指令不保存运算结果.在进行两个数据大小判断时,常用 CMP 指令及相应的条件码来操作.

*2.4 if*分支判断

a) 标签

内存地址的别名;

本身不占地方, 仅仅是个别名;

b) 跳转

在 ARM 中有两种方式可以实现程序的跳转,一种是使用跳转指令直接跳转,另一种则是直接向 PC 寄存器赋值实现跳转.跳转指令有跳转指令 B,带链接的跳转指令 BL 带状态切换的跳转指令 BX.

1. B

```
B{cond}   label      (pc<-label)
```

跳转指令 B 举例如下:

```
B   WAITA        ;跳转到 WAITA 标号处
B   0x1234       ;跳转到绝对地址 0x1234 处
```

小结:

跳转到指令 B 限制在当前指令的±32Mb 的范围内.

2. BL

BL{cond} label   (LR<-pc-4; pc<-label)

带链接的跳转指令 BL 举例如下:

BL   DELAY

跳转指令 B 限制在当前指令的±32MB 的范围内.BL 指令用于子程序调用

3. BX

带状态切换的跳转指令.跳转到 Rm 指定的地址执行程序,

若 Rm 的位[0]为 1,则跳转时自动将 CPSR 中的标志 T 置位,即把目标地址的代码解释为 Thumb 代码;

若 Rm 的位[0]为 0,则跳转时自动将 CPSR 中的标志 T 复位(清零),即把目标地址的代码解释为 ARM 代码.

格式如下:

BX{cond}   Rm

带状态切换的跳转指令 BX 举例如下

ADRL R0,ThumbFun+1

BX   R0   ;跳转到 R0 指定的地址,并根据 R0 的最低位来切换处理器状态

c) if

```
#if 0
   if(a < 0)
   {
      a = 1;
   }
   else
   {
      a = 2;
   }

#else
      "cmp %0, #0\n"
      "movlt %0, #1\n"
      "movge %0, #2\n"
      :"+r" (a)
   );
#endif
```

2.5 for 循环

a) 一层循环

```
#if 0
   for (i = 0; i < 101; i ++) {
      a = a + i;
   }
#else
   __asm__ __volatile__ (
      "mov %0, #0\n"
      "loop:\n"
      "cmp %1, #101\n"
      "bge loop_end\n"
         "add %0, %0, %1\n"
      "add %1, %1, #1\n"
      "b loop\n"
      "loop_end:\n"
      :"+r" (a)
      :"r" (i)

   );
#endif
```

b) 二层循环

```
#if 0
   a = 0;
   for(i = 0; i < 2; i++){
      for(j = 0; j < 101; j++){
         a = a + j;
      }
   }
#else
   __asm__ __volatile__ (
      "mov %0, #0\n"

      "mov %1, #0\n"
      "loop1:\n"
```

```
      "cmp %1, #2\n"
      "bge loop1_end\n"

         "mov %2, #0\n"
         "loop2:\n"
         "cmp %2, #101\n"
         "bge loop2_end\n"

            "add %0, %0, %2\n"

         "add %2, %2, #1\n"
         "b loop2\n"
         "loop2_end:\n"

         "add %1, %1, #1\n"
         "b loop1\n"
      "loop1_end:\n"

      :"+r" (a)
      :"r" (i), "r" (j)
   );
#endif
```

3. 指针与数组

3.1 LDR 和 *STR*

a) 立即数偏移(数组、表)

LDR R1,[R0,#0x12] ;将 R0+0x12 地址处的数据读出,保存到 R1 中(R0 的值不变)

LDR R1,[R0,#-0x12];将 R0-0x12 地址处的数据读出,保存到 R1 中(R0 的值不变)

LDR R1,[R0] ;将 R0 地址处的数据读出,保存到 R1 中(零偏移)

b)寄存器偏移

LDR R1,[R0,R2] ;将 R0+R2 地址的数据计读出,保存到 R1 中(R0 的值不变)

LDR R1,[R0,-R2] ;将 R0-R2 地址处的数据计读出,保存到 R1 中(R0 的值不变)

3) 寄存器及移位常数

LDR R1,[R0,R2,LSL #2] ;将 R0+R2*4 地址处的数据读出,保存到 R1 中(R0,R2 的值

不变)

LDR R1,[R0,-R2,LSL #2];将 R0-R2*4 地址处的数据计读出,保存到 R1 中(R0,R2 的值不变)

4) 前索引偏移.

在数据传送之前,将偏移量加到 Rn 中,其结果作为传送数据的存储

地址.若使用后缀“!”,则结果写回到 Rn 中,且 Rn 值不允许为 R15.

LDR Rd,[Rn,#0x04]!

5) 后索引偏移.

Rn 的值用做传送数据的存储地址.在数据传送后,将偏移量与 Rn 相加,结果写回到 Rn 中.

Rn 不允许是 R15.

指令举例如下:

LDR Rd,[Rn],#0x04

6) 程序相对偏移.(伪指令)

LDR Rd,label ;label 为程序标号,label 必须是在当前指令的±4KB 范围内

小结:

LDR/STR 指令用于对内存变量的访问,内存缓冲区数据的访问、查表、外围部件的控制操作等等,若使用 LDR 指令加载数据到 PC 寄存器,则实现程序跳转功能,这样也就实现了程序散转。

举例:

1)变量访问(int *p;int i; p=0x40003000; i=*p; i=i+1; *p = i;)

```
NumCount EQU 0x40003000 ;定义变量 NumCount
...
LDR R0,=NumCount ;使用 LDR 伪指令装载 NumCount 的地址到 R0
LDR R1,[R0]      ;取出变量值
ADD R1,R1,#1     ;NumCount=NumCount+1
STR R1,[R0]      ;保存变量值
...
```

2) GPIO 设置

```
GPIO-BASE EQU 0Xe0028000     ;定义 GPIO 寄存器的基地址
 ...
LDR R0,=GPIO-BASE
LDR R1,=0x00FFFF00 ;装载 32 位立即数,即设置值
```

```
    STR R1,[R0,#0x0C]  ;IODIR=0x00FFFF00, IODIR 的地址为 0xE002800C
    MOV R1,#0x00F00000
    STR R1,[R0,#0x04]  ;IOSET=0x00F00000,IOSET 的地址为 0xE0028004
     ...
```

3) 程序散转(中断向量表)

```
        ...
        MOV R2,R2,LSL #2 ;功能号乘上 4,以便查表
        LDR PC,[PC,R2]   ;查表取得对应功能子程序地址,并跳转
        NOP
  FUN-TAB DCD FUN-SUB0
        DCD FUN-SUB1
        DCD FUN-SUB2
        …
```

a) 指针

```
LDR   r0, [r1]       ;r0 := mem [r1]
STR   r0, [r1]       ;mem [r1] := r0
```

b) 数组

```
LDR r0, [r1, #4]     ;r0 := mem [r1+4]
                     ;r1 := r1
LDR r0, [r1, #4]!    ;r0 := mem [r1+4]
                     ;r1 := r1 + 4

LDR r0, [r1], #4     ;r0 := mem [r1]
                     ;r1 := r1 + 4
```

3.2 LDM 和 *STM* 多寄存器访存

a) 介绍

批量加载/存储指令可以实现在一组寄存器和一块连续的内存单元之间传输数据.

LDM 为加载多个寄存器, STM 为存储多个寄存器.

允许一条指令传送 16 个寄存器的任何子集或所有寄存器.

LDM /STM 的主要用途是现场保护、数据复制、 参数传送等。

多寄存器加载/存储指令格式如下:

```
LDM{cond}<模式>  Rn{!},reglist{^}
STM{cond}<模式>  Rn{!},reglist{^}
```

其模式有 8 种,如下:

(前面 4 种用于数据块的传输,后面 4 种是堆栈操作)

(1) IA:每次传送后地址加 4

(2) IB:每次传送前地址加 4

(3) DA:每次传送后地址减 4

(4) DB:每次传送前地址减 4

(5) FD:满递减堆栈

(6) ED:空递增堆栈

(7) FA:满递增堆栈

(8) EA:空递增堆栈

注意:

1. 高地址对应高寄存器编号
2. 堆栈操作的D,A模式与I,D指针加减并不相同。

b) 块内存访问

. 指令

```
ldmia, stmia
ldmib, stmib
ldmda, stmda
ldmdb, stmdb
```

e.g:

```
ldmia r9!, {r0, r1, r2-r5}^
stmia r9!, {r0, r1, r2-r5}
```

. 数据传输方向

```
ldm : cpu <- 内存
stm : cpu -> 内存
```

. 模式说明

1. ! 内存地址更新

```
i - Increment
d - Decrement
```

2. a - after
   b - before

. 寄存器表

1. 每个寄存器用","隔开，还可以连续多个寄存器用"r0-r14", 像word打印;

2. 高编号寄存器放在高低址，打乱的话依然，只是编译器会报个警告。

"^": 可选后缀

1. 后缀“^”不允许在用户模式或系统模式下使用。

2. 若在用LDM指令时，且寄存器列表中包含有PC时使用，那么除了正常的多寄存器传送外，将SPSR也拷贝到CPSR中，这可用于异常处理返回。

3. 使用后缀“^”进行数据传送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。

c) 栈内存访问

两套指令的诀窍:

第一套，栈，从高存，从低取是同模式，如，stmfd, ldmfd;

第二套，多，从高存，从高取是同模式，如:stmdb, ldmdb;

4. 结构体

4.1 结构体实现细节

4.2 链表实现

5. 函数(apcs)

5.1. 调用子函数

1)bl test

2)mov lr, pc
  b test

5.2. 寄存器变量

. 返回值 - r0

. 参数 - r0, r1, r2, r3

. 特殊寄存器定义

r11 - fp

r12 - ip

r13 - sp

r14 - lr

r15 - pc

5.3. 子函数实现

. 堆栈

. 保存现场恢复现场

```
    mov ip, sp
    stmfd sp!, {fp, ip, lr, pc}
    sub fp, ip, #4

    ....

    sub sp, fp, #12
    ldmfd sp, {fp, sp, pc}
```

三、GNU Binutils

1. GCC编译器

程序编译过程

1). 预处理

cpp or gcc -E -> .i

2). 编译

ccl or gcc -S -> .s

3). 汇编

as or gcc -c -> .o

4). 链接

ld -> .elf

指定动态加载器,启动代码

作用，在arm linux系统上执行汇编的elf

```
arm-linux-ld --dynamic-linker=/lib/ld-linux.so.2 1.o 2.o
/usr/local/arm/3.4.1/arm-linux/lib/crt*.o -lc -o test
```

objcopy 工具

```
arm-linux-objcopy [...] infile [outfile]
   -I bfdname        指定输入文件格式
   -O bfdname        指定输出文件格式
   -S                不从源文件中复制重定位信息和符号信息到目标文件中去。
   -R secname        删除secname段
   --info            列出bfdname
```

在 ***bootloader*** 上执行 ***asm bin***

编写链接脚本;

```
arm-linux-gcc -c -o test.o test.S
arm-linux-ld -T test.lds -o test.elf test.o
arm-linux-objcopy -O binary -S test.elf test.bin
```

2. 调试工具

```
man
--help
```

1)addr2line:

用得到程序地址所对应源代码的文件名和行号以及所对应的函数。

[用法]

```
addr2line 0x401100 -f -e test.elf
```

addr2line 工具正确的指出了地址 0x401100 所对于应的程序的具体位置是在哪以及所对应的函数名是什么。

2)nm:

用于列出目标文件、库或是可执行文件(后面统称这三种文件为程序文件)中的代码符号及代码符号所对应的程序开始地址。

第一列表示符号的地址;

第二列表示符合在哪个段，man nm;

A 表示符号所对应的值是绝对的且在以后的连接过程中也不会改变

B 或 b 表示符号位于未初始化的数据段(.bss 段)中

C 表示没有被初始化的共公符号

D 或 d 表示符号位于初始化的数据段(.data 段)中
N 表示符号是调试用的
p 表示符号位于一个栈回朔段中
R 或 r 表示符号位于只读数据段(.rdata 段)中
T 或 t 表示符号位于代码段(.text 段)中
U 表示符号没有被定义

第三列是符合的名字;

[用法]
nm -n test.elf

3) objdump (重点)

帮助我们显示程序文件的相关信息。
可以用来查看目标程序中的段信息和调试信息,也可以用来对目标程序进行反汇编

1. objdump -h xxx.elf > tmp.txt
   查看程序中的段信息
2. objdump -f xxx.elf > tmp.txt
   采用-f 选项可以显示目标文件的头信息。其中最要注意的是 start address,其指示了这一程序
   被执行时的入口地址是什么。
3. objdump -d xxx.elf > tmp.txt (反汇编, -S 可以同时显示 c 代码)
4. objdump -D xxx.elf > tmp.txt (还显示一些调试信息)
5. objdump -s -j .data test.elf > tmp.txt
   查看某一个段中的具体内容
6. 反汇编 bin 文件
   objdump -D -b binary -m arm test.bin > tmp.txt

4) readelf:

用于显示 ELF 文件的信息。
readelf 完全可以用 objdump 代替

5) size -A:

size -A test.elf
用于显示程序文件的段大小信息。

6) strings:

strings test.elf
用于显示一个程序文件当中的可显示字符串。

7) strip:

strip test.elf
用于剥去程序文件中的符号信息,以减小程序文件的大小。

这对于存储空间有限的嵌入式系统尤为有用

四、 链接脚本

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm","elf32-
littlearm");
```

若有命令行选项-EB, 则使用第 2 个 BFD 格式; 若有命令行选项-EL,则使用第 3 个 BFD 格式.否则默认选第一个 BFD 格式.

```
OUTPUT_ARCH(arm);
```

一句话，照抄.....因为我们没有修改的余地，都是系统默认的关键字。第一句指示系统可以有生成两种格式，默认是 elf32-arm，端格式是 little endian。

```
ENTRY("_start");
```

程序第一条执行的指令，如果不写，默认是.text 段的第一条指令;

一个标准的 lds 模板:

```
--------------------------- xxx.lds -----------------------------
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-
littlearm");
OUTPUT_ARCH("arm");
ENTRY("_start");
SECTIONS
{
   . = 0x30000000;
   .text : {
      *(.text);
   }

   . = ALIGN(4);
   .data : {
      *(.data);
   }

   . = ALIGN(4);
   .bss : {
      *(.bss);
   }
}
--------------------------- end -----------------------------
```

五、 伪指令,伪操作，宏指令

一般与编译程序有关,因此 ARM 汇编语言的伪操作,宏指令在不同的编译环境下有不同的编写形式和规则.
伪指令也是 ARM 汇编语言程序里的特殊助记符,也不在处理器运行期间由机器执行,
他们在汇编时将被合适的机器指令代替成 ARM 或 Thumb 指令,从而实现真正的指令操作.

目前常用的 ARM 编译环境有 2 种.

1. ADS/SDT IDE:ARM 公司开发,使用了 CodeWarrior 公司的编译器. (armasm)
2. 集成了 GNU 开发工具的 IDE 开发环境;它由 GNU 的汇编器 as,交叉汇编器 gcc 和连接器 id 组成.

(gas)

注意 gas 伪指令都是以"."开头:

1. 伪指令

1)小范围，一般在一个段内，通过 pc+offset 实现
```
   adr r0, str    @取标签 str 址， 指针
   ldr r0, str    @取标签 str 值
```

2)大范围，任意位置
```
   ldr r0, =str   @取标签 str 址，指针
```

3)32 位立即数，任意数
```
   ldr r0, =0x31000000
```

2. 伪操作

1) .section 用于定义接下来的代码在哪个段
```
   .section .text
   .section .data
   .section .bss
   .section .xxx
```

2) .global （全局变量，或接口函数)
```
   .global symbol
```
   定义一个全局符号， 通常是为 ld 使用.

2) 段内对齐
```
   .align 2
```

3) 宏.equ(像#define)
```
   .equ NUM 0x100
```

4).asciz "hello world.\n"
   定义一个字符串并为之分配空间.

5).word 0x56000000
   定义一个字,并为之分配空间, 4bytes.

六、编程注意

1. 对齐问题
   注意对齐问题:
   段内对齐, .align 2
   段间对齐, . = ALIGN(4);