嵌入式系统设计与应用

第二章 ARM指令系统（2）

西安交通大学电信学院
孙宏滨

汇编伪指令

- 汇编伪指令：在ARM汇编语言里，有一些特殊指令助记符，没有相对应的操作码（或直接对应指令）。通常称这些特殊指令助记符为伪指令，它们所完成的操作叫做伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作。这些伪指令仅在汇编过程中起作用，一旦汇编结束，伪指令的使命完成。
- **ADR**：小范围的地址读取伪指令
  - **ADR**指令将基于**PC**相对偏移的地址值读取到寄存器中。在汇编编译源程序时，**ADR**被编译器替换成一条合适的指令（**ADD or SUB**）。
  - 指令格式：**ADR{cond}　register, expr**
  - 举例：　**start　MOV　r1,　#10**
  - **ADR　r4,　start**

几条常用伪指令

- **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。
  - 常用属性：
    - **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**；
    - **… …**
  - 一个汇编语言程序至少要包含一个段，程序太长时，可分为多个代码段和数据段。
  - 使用示例：　**AREA　Init，CODE，READONLY**
    - 定义一个代码段，段名为**Init**，属性为只读。

几条常用伪指令

- **ENTRY**：语法格式　**ENTRY**
  - **ENTRY**伪指令用于指定汇编程序的入口点。在一个完成的汇编程序中至少要有一个**ENTRY**，但在一个源文件中最多只能有一个**ENTRY**。
  - 使用示例：　　**AREA　Init，CODE，READONLY**
  - 　　**ENTRY**
- **END**：语法格式　**END**
  - **END**伪指令用于通知汇编器已经到了源程序的结尾。
  - 使用示例：　　**AREA　Init，CODE，READONLY**
  - 　　**… …**
  - 　　**END**

几条常用伪指令

- **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**”
  - 名称为**EQU**伪指令定义的字符名称
  - 当表达式为**32**位常量时，可指定表达式的数据类型，有以下三种类型：**CODE16**、**CODE32**和**DATA**
  - 使用示例：　**Test　EQU　50　;**定义标号**Test**的值为**50**
  - **Addr　EQU　0x55, CODE32　;**
- **ALIGN**：
  - **ALIGN**伪指令通过添加填充字节的方式，使当前位置满足一定的对齐方式。

几条常用伪指令

```
        AREA Example, CODE, READONLY        ; name this block of code
        ENTRY                               ; mark first instruction
                                            ; to execute
start
        MOV     r0, #15                     ; Set up parameters
        MOV     r1, #20
        BL      firstfunc                   ; Call subroutine
        SWI     0x11                        ; terminate
firstfunc                                   ; Subroutine firstfunc
        ADD     r0, r0, r1                  ; r0 = r0 + r1
        MOV     pc, lr                      ; Return from subroutine
                                            ; with result in r0
        END                                 ; mark end of file
```

label

opcode

operands

comment

乘法运算指令 – MUL, MLA

- **MUL r4, r3, r2 ; r4 := (r3 x r2)$_{[31:0]}$**
  - 第二操作数不支持立即数寻址（可使用寄存器寻址替代）
  - 结果寄存器不能与第一源寄存器相同,两源寄存器也不同
  - 如果使用’S’位，则V状态位保留，C状态位无意义
- **MLA r4, r3, r2, r1 ; r4 := (r3xr2 + r1)$_{[31:0]}$**
  - 乘加指令
- **r0x35**
  - **ADD r0, r0, r0 LSL #2 ; r0’ := r0x5**
  - **RSB r0, r0, r0 LSL #3 ; r0’’:= r0’x7(=r0x35)**

ARM汇编程序范例（1）

```
        AREA    helloW, CODE, READONLY  ; declare code area
SWI_WriteC      EQU     &0              ; output character in r0
SWI_Exit        EQU     &11             ; finish program
        ENTRY                           ; code entry point
START   ADR     r1, TEXT                ; r1 -> "Hello World!"
LOOP    LDRB    r0, [r1], #1            ; get the next byte
        CMP     r0, #0                  ; check for 'null' character
        SWINE   SWI_WriteC              ; if not end, print ....
        BNE     LOOP                    ;  ... and loop back
        SWI     SWI_Exit                ; end of execution

TEXT    =       "Hello World!", &0a, &0d, 0  ; string + CR + LF + null
        END
```

ARM汇编程序范例（2）

```
            AREA    BlkCpy, CODE, READONLY  ; declare code area
SWI_WriteC          EQU     &0              ; output character in r0
SWI_Exit            EQU     &11             ; finish program
            ENTRY                           ; code entry point
START       ADR     r1, TABLE1              ; r1 -> TABLE1
            ADR     r2, TABLE2              ; r2 -> TABLE2
            ADR     r3, T1END               ; r3 -> end of TABLE1
LOOP1       LDR     r0, [r1], #4            ; get TABLE1 1st word
            STR     r0, [r2], #4            ; copy into TABLE2
            CMP     r1, r3                  ; finished?
            BLT     LOOP1                   ; if not, do more, else print
            ADR     r1, TABLE2              ; r1 -> TABLE2
LOOP2       LDRB    r0, [r1], #1            ; load the 1st byte of copied string
            CMP     r0, #0                  ; check for end of text string
            SWINE   SWI_WriteC              ; if not end, print ...
            BNE     LOOP2                   ; .... and loop back
            SWI     SWI_Exit                ; finish
TABLE1      =       "This is the right string!", &0a, &0d, 0
T1END
            ALIGN                           ; ensure word alignment
TABLE2      =       "This is the wrong string!", 0
            END
```

练习（1）：赋值语句

- C程序：x = (a + b) - c;
- 汇编程序：???

练习（1）：赋值语句

- C程序：x = (a + b) - c;
- 汇编程序：
  - ADR r4, a ; get address for a
  - LDR r0, [r4] ; get value of a
  - ADR r4, b ; get address for b, reusing r4
  - LDR r1, [r4] ; get value of b
  - ADD r3, r0, r1 ; compute a+b
  - ADR r4, c ; get address for c
  - LDR r2, [r4] ; get value of c
  - SUB r3, r3, r2 ; complete computation of x
  - ADR r4, x ; get address for x
  - STR r3, [r4] ; store value of x

练习（2）：赋值语句

- C程序：y = a * (b + c);
- 汇编程序：???

练习（2）：赋值语句

- C程序：y = a * (b + c);
- 汇编程序：
  - ADR r4, b ; get address for b
  - LDR r0, [r4] ; get value of b
  - ADR r4, c ; get address for c
  - LDR r1, [r4] ; get value of c
  - ADD r2, r0, r1 ; compute partial result
  - ADR r4, a ; get address for a
  - LDR r0, [r4] ; get value of a
  - MUL r2, r2, r0 ; compute final value for y
  - ADR r4, y ; get address for y
  - STR r2, [r4] ; store y

练习（3）：赋值语句

- C程序：z = (a << 2) | (b & 15);
- 汇编程序：???

练习（3）：赋值语句

- C程序：z = (a << 2) | (b & 15);
- 汇编程序：???
  - ADR r4, a ; get address for a
  - LDR r0, [r4] ; get value of a
  - MOV r0, r0 LSL 2 ; perform shift
  - ADR r4, b ; get address for b
  - LDR r1, [r4] ; get value of b
  - AND r1, r1, #15 ; perform AND
  - ORR r1, r0, r1 ; perform OR
  - ADR r4, z ; get address for z
  - STR r1, [r4] ; store value for z

练习（4）：if语句

- C程序：if (a > b) { x = 5; y = c + d; } else x = c - d;
- 汇编程序：???

练习（4）：if语句

- C程序： if (a < b) { x = 5; y = c + d; } else x = c - d;
- 汇编程序：
  - ; compute and test condition
  - ADR r4, a ; get address for a
  - LDR r0, [r4] ; get value of a
  - ADR r4, b ; get address for b
  - LDR r1, [r4] ; get value for b
  - CMP r0, r1 ; compare a < b
  - BGE fblock ; if a >= b, branch to false block

练习（4）：if语句

```
                    ; true block
MOV  r0, #5         ; generate value for x
ADR  r4, x          ; get address for x
STR  r0, [r4]       ; store x
ADR  r4, c          ; get address for c
LDR  r0, [r4]       ; get value of c
ADR  r4, d          ; get address for d
LDR  r1, [r4]       ; get value of d
ADD  r0, r0, r1     ; compute y
ADR  r4, y          ; get address for y
STR  r0, [r4]       ; store y
B    after          ; branch around false block
```

练习（4）：if语句

```
                                ; false block
fblock  ADR  r4,  c             ; get address for c
        LDR  r0,  [r4]          ; get value of c
        ADR  r4,  d             ; get address for d
        LDR  r1,  [r4]          ; get value for d
        SUB  r0,  r0,  r1       ; compute a-b
        ADR  r4,  x             ; get address for x
        STR  r0,  [r4]          ; store value of x
after ...
```

练习（4）：if语句-条件执行指令

```
                        ; true block
MOVLT  r0, #5           ; generate value for x
ADRLT  r4, x            ; get address for x
STRLT  r0, [r4]         ; store x
ADRLT  r4, c            ; get address for c
LDRLT  r0, [r4]         ; get value of c
ADRLT  r4, d            ; get address for d
LDRLT  r1, [r4]         ; get value of d
ADDLT  r0, r0, r1       ; compute y
ADRLT  r4, y            ; get address for y
STRLT  r0, [r4]         ; store y
```

练习（5）：switch语句

- C程序：switch (test) { case 0: … break; case 1: … }
- 汇编程序：???

练习（5）：switch语句

- C程序：switch (test) { case 0: … break; case 1: … }
- 汇编程序：

```
        ADR  r2,  test              ; get address for test
        LDR  r0,  [r2]              ; load value for test
        ADR  r1,  switchtab         ; load address for switch table
        LDR  r15,  [r1, r0 LSL #2] ; index switch table
switchtab  DCD case0
           DCD case1
           ...
case0      ...          ; code for case 0
           ...
case1      ...          ; code for case 1
```

DCD伪指令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。

练习（6）FIR滤波器

- C程序：
  - for (i=0, f=0; i<N; i++)
  - f = f + c[i]*x[i];
- 汇编程序：???

练习（6）FIR滤波器

- C程序：
  - for (i=0, f=0; i<N; i++)
  - f = f + c[i]*x[i];
- 汇编程序：
  - ; loop initiation code
  - MOV r0, #0 ; use r0 for I
  - MOV r8, #0 ; use separate index for arrays
  - ADR r2, N ; get address for N
  - LDR r1, [r2] ; get value of N
  - MOV r2, #0 ; use r2 for f

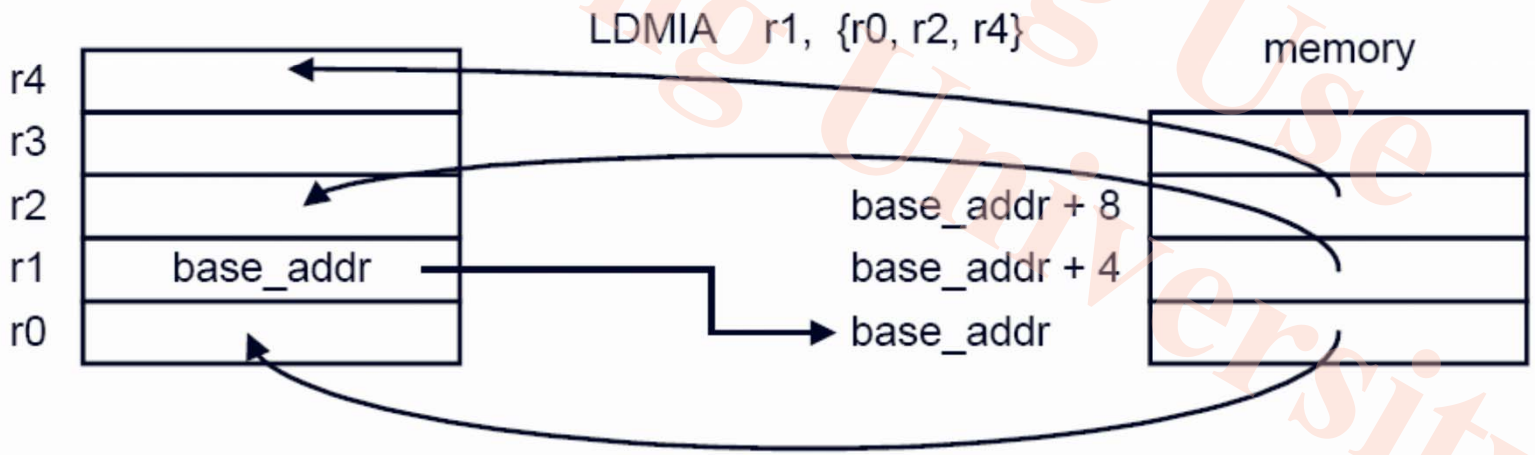
练习（6）FIR滤波器

```
        ADR  r3,  c        ; load r3 with base of c
        ADR  r5,  x        ; load r5 with base of x
                           ; loop body
loop    LDR  r4,  [r3,r8]  ; get c[i]
        LDR  r6,  [r5,r8]  ; get x[i]
        MUL  r4,  r4,  r6  ; compute c[i]*x[i]
        ADD  r2,  r2,  r4  ; add into running sum
        ADD  r8,  r8,  #4  ; add one word offset
        ADD  r0,  r0,  #1  ; add 1 to i
        CMP  r0,  r1       ; exit?
        BLT  loop          ; if i < N, continue
```

栈与子程序

- **LDR和STR指令仅能load/store一个32位字**
- **ARM可以在一条指令中load/store 16个寄存器中的一个任意数目的子集。**

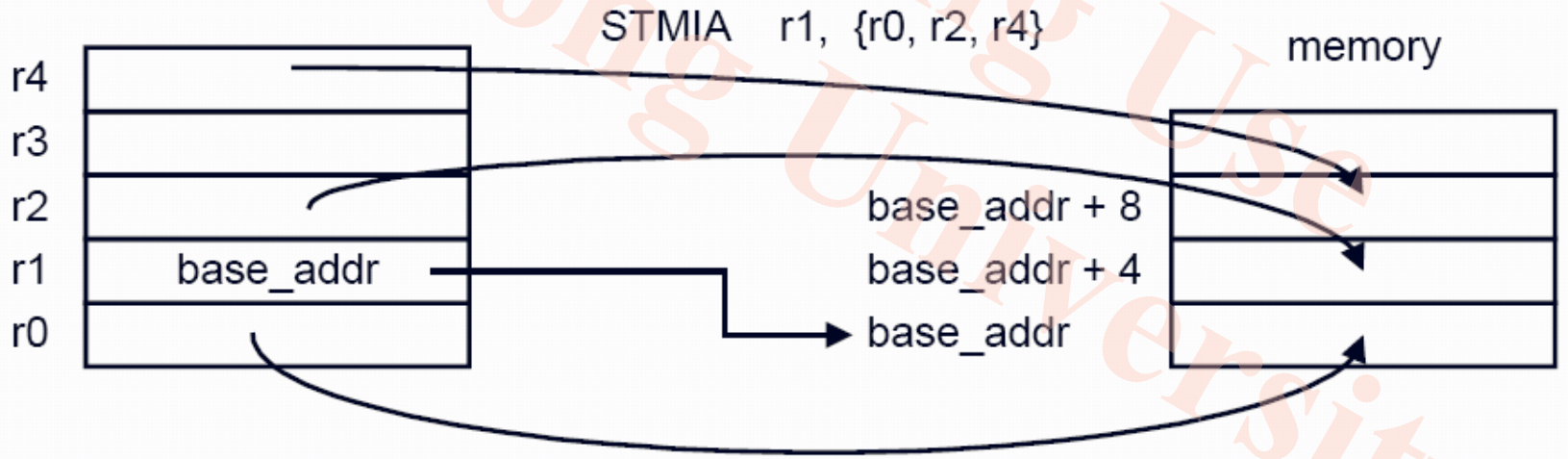
```
LDMIA   r1, {r0, r2, r4}        ; r0 := mem32[r1]
                                ; r2 := mem32[r1+4]
                                ; r4 := mem32[r1+8]
```



多数据传送指令

- 任何寄存器都可以被指定为多数据传送。但是，如果使用**r15(PC)**，则意味着将在程序流中强制跳转。
- **LDMIA**的互补指令是**STMIA**指令。

```
STMIA   r1, {r0, r2, r4}        ; mem32[r1] := r0
                                ; mem32[r1 + 4] := r2
                                ; mem32[r1 + 8] := r4
```



多数据传送指令更新基地址寄存器

可以使用“!”使多数据传送指令基地址自动变址。

```
LDMIA   r1 ! , {r2-r9}        ; r2 := mem32[r1]
                              ;  ................
                              ; r9 := mem32[r1+28]
                              ; r1 := r1 + 32
```

load multiple

update r1 after used

meaning of LDMIA instruction

I - increment base address

A - base address is incremented after it is used

load registers r2 to r9

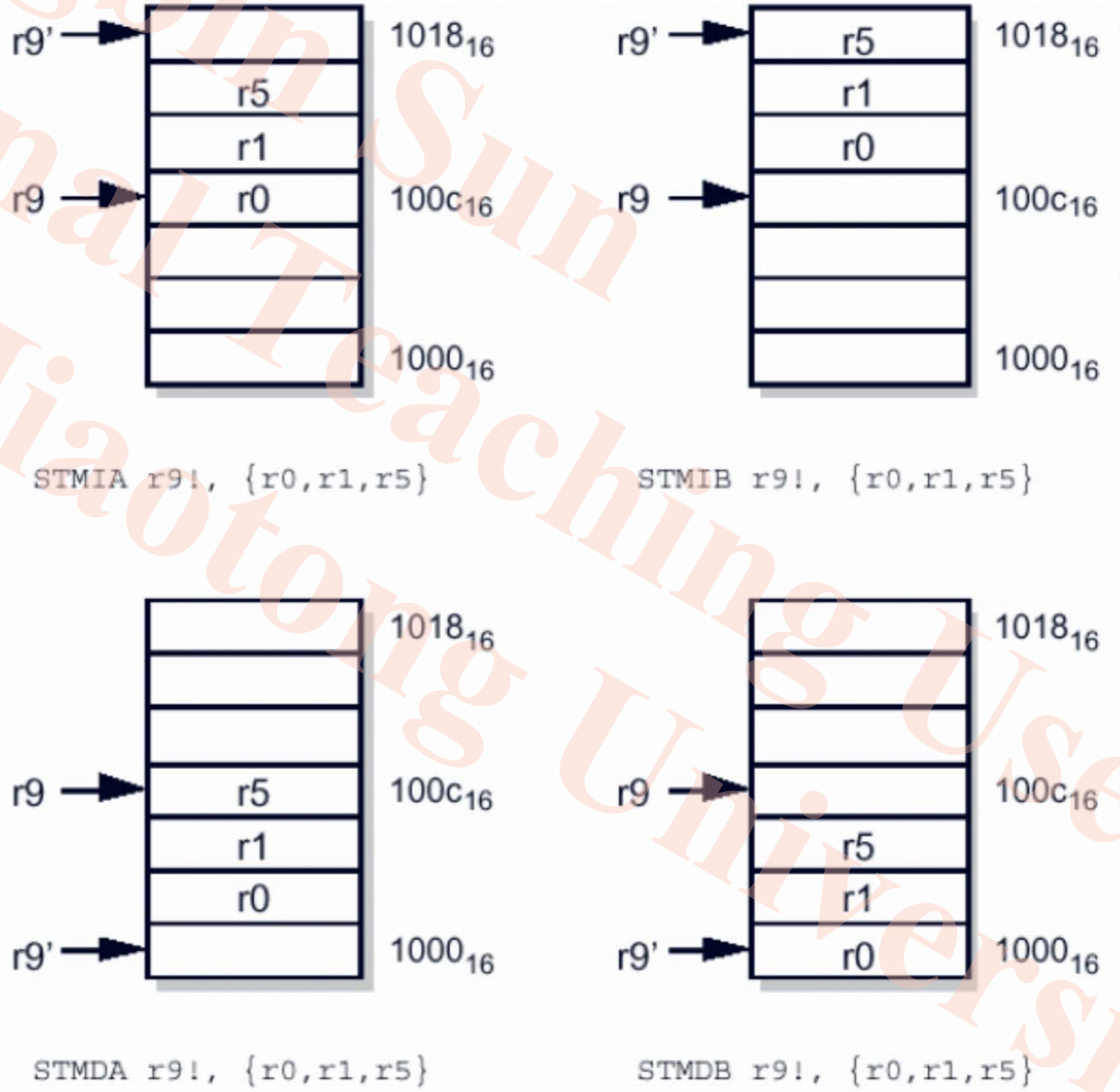
多数据传送指令使用范例

使用多数据传送指令将8个字从存储源地址传送到目的地址。

```
ADR     r0, src_addr        ; initialize src addr
ADR     r1, dest_addr       ; initialize dest addr
LDMIA   r0!, {r2-r9}        ; fetch 8 words from mem
                            ;   ... and update r0 := r0 + 32
STMIA   r1, {r2-r9}         ; copy 8 words to mem, r1 unchanged
```

- 当使用**LDMIA**和**STMIA**指令时，地址增加（**Increment**），且在地址被使用后（**After**）增加。
- 事实上，多数据传送指令有四种格式：
  - Increment - **A**fter　　LDM**IA** and STM**IA**
  - Increment - **B**efore　　LDM**IB** and STM**IB**
  - **D**ecrement - **A**fter　　LDM**DA** and STM**DA**
  - **D**ecrement - **B**efore　　LDM**DB** and STM**DB**

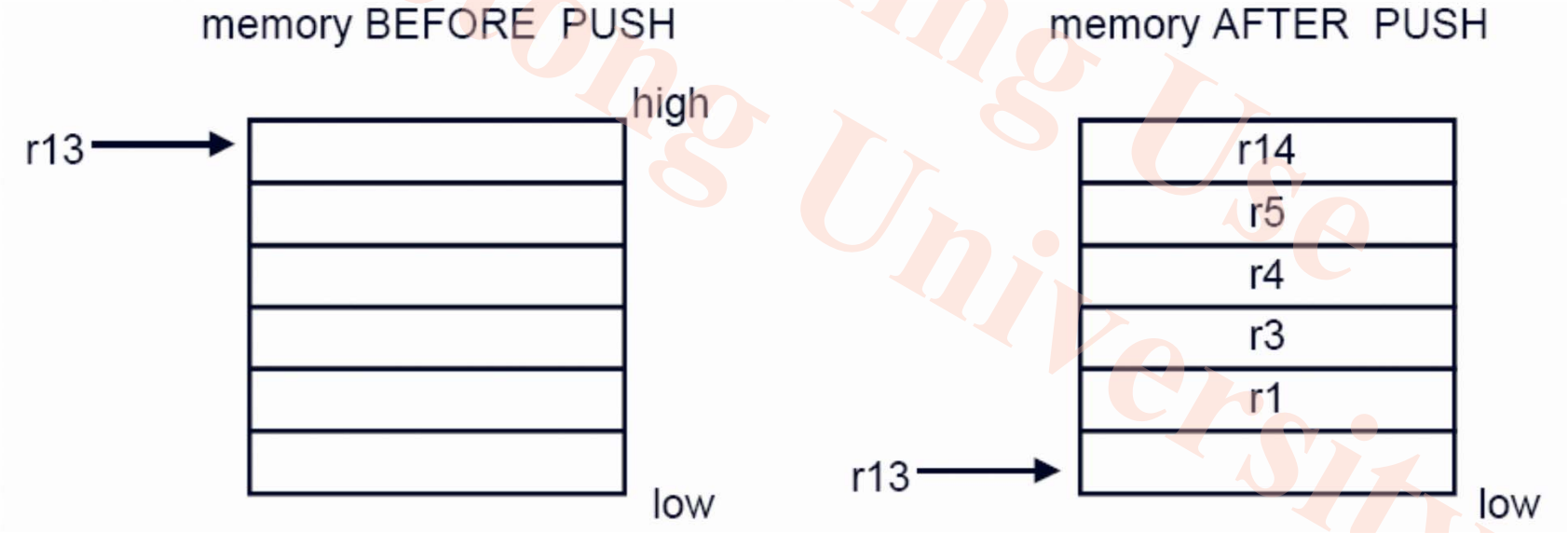
STM指令的四种变形



栈（Stack）的概念

- 栈（Stack）在计算机科学中是限定仅在表尾进行插入或删除操作的线性表。
- 栈是一种在主存空间中临时存储数据的数据结构，按照**后进先出**的原则存储数据。
- 进栈操作（Push）指将多个寄存器存储入栈结构。

PUSH {r1, r3-r5, r14}



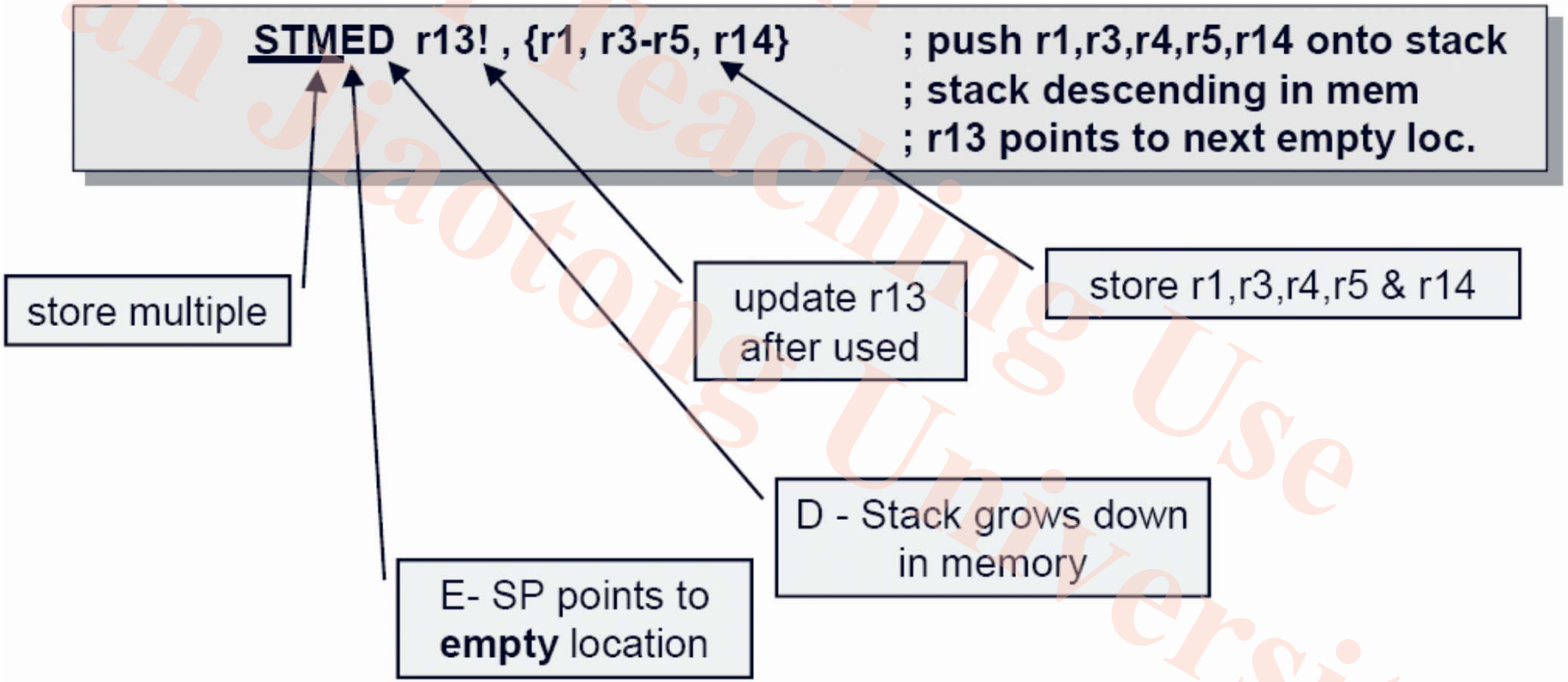
进栈（PUSH）操作

- **PUSH**操作的特点：
  - **r13**通常被用作地址指针，被称为栈指针（**Stack Pointer, SP**）。我们也可使用其他寄存器（**r15**除外），但除非有更好的理由，推荐使用**r13**。
  - 栈通过减小存储器地址来降低。
  - 基地址寄存器指向栈的第一个空位。若向存储器中存储新的数据值，**SP**在使用后应减小至新的空位。
- **ARM**没有**PUSH**指令，但可以使用**STM**指令完成操作

```
STMDA r13!, {r1, r3-r5, r14}        ; Push r1, r3-r5, r14 onto stack
                                    ; Stack grows down in mem
                                    ; r13 points to next empty loc.
```

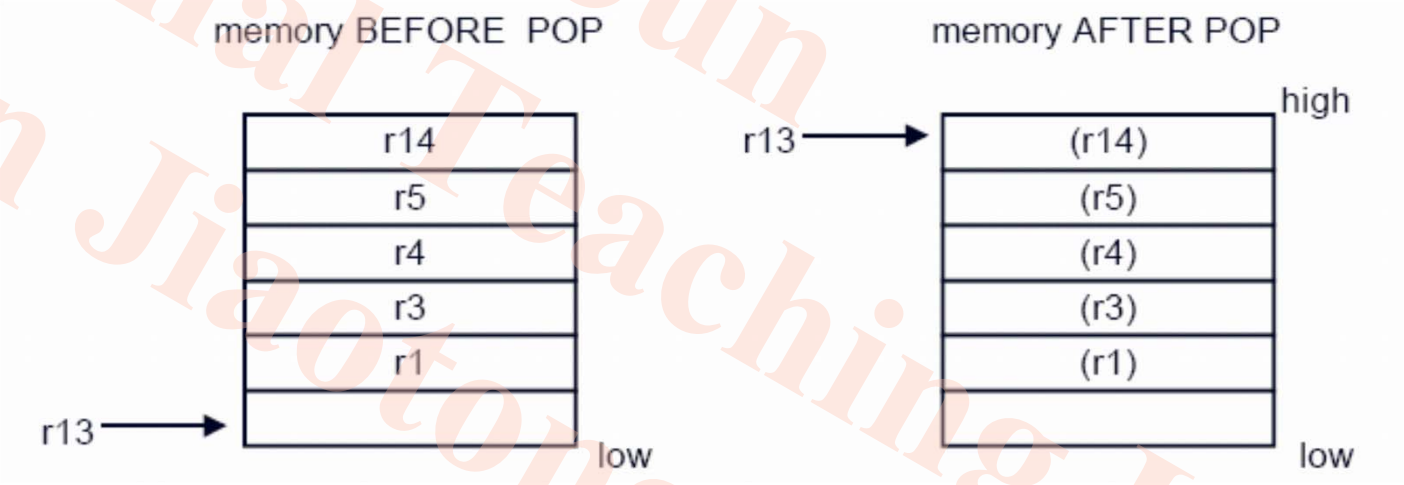
STM指令

- 在ARM术语中，执行栈操作的STM指令可以有个不同的名字。已提到的STMDA等同于STMED指令。



出栈（POP）操作

- 与PUSH互补的操作是POP。 **POP {r1, r3-r5, r14}**



ARM没有POP指令，因此我们使用：

```
LDMIB   r13!, {r1, r3-r5, r14}        ; Pop r1, r3-r5, r14 from stack
```

与LDMIB等同的栈操作指令：

```
LDMED r13!, {r1, r3-r5, r14}          ; Pop r1, r3-r5, r14 from stack
```

设计栈的四种不同方式

- **Ascending/Descending**：当加入数据时，它就向上增大存储空间（递增栈）或向下增大存储空间（递减栈）。
- **Full/Empty**：栈指针总保持在当前栈顶的地址，它指向最后压入栈的有效数据（满栈）或指向将被压入下一个数据的空位（空栈）。

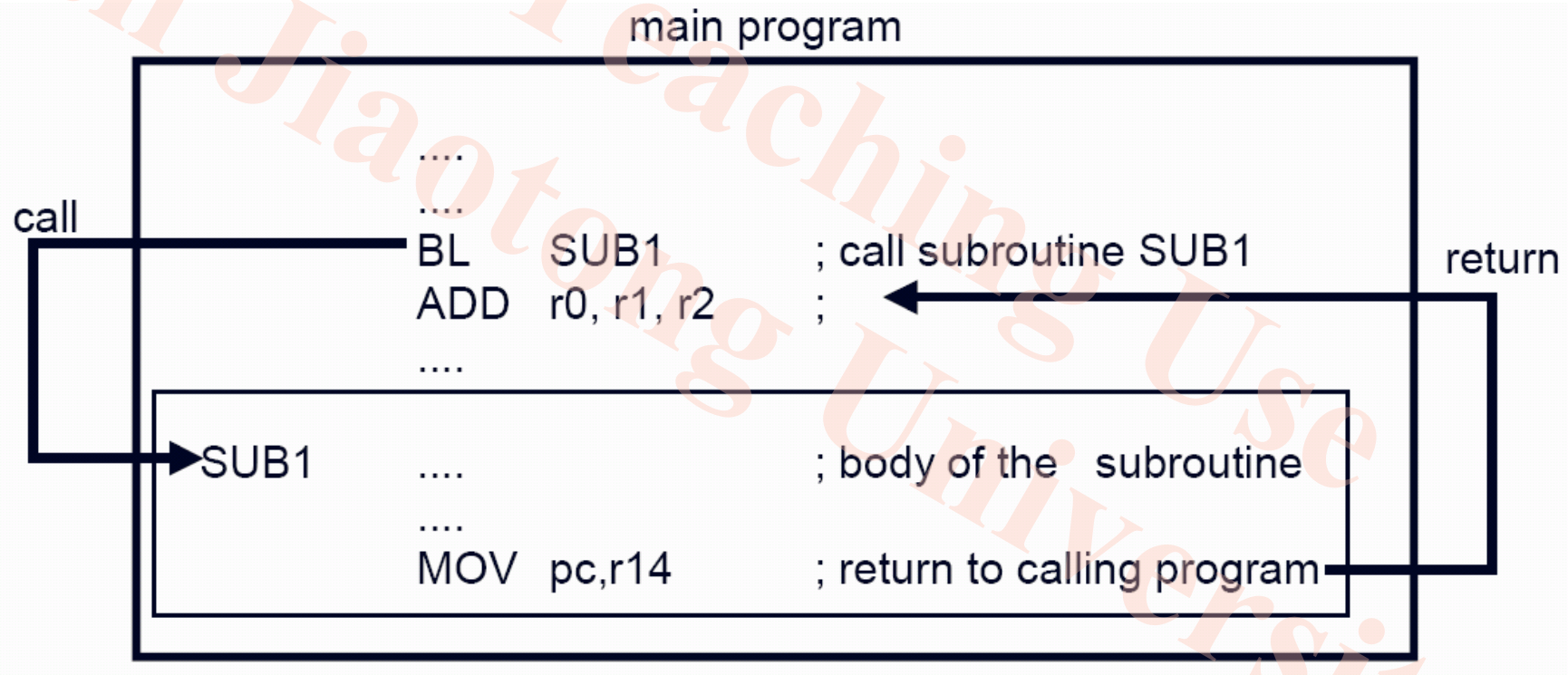
```
STMFA  r13!, {r0-r5}; Push onto a Full Ascending Stack
LDMFA  r13!, {r0-r5}; Pop from a Full Ascending Stack
STMFD  r13!, {r0-r5}; Push onto a Full Descending Stack
LDMFD  r13!, {r0-r5}; Pop from a Full Descending Stack
STMEA  r13!, {r0-r5}; Push onto an Empty Ascending Stack
LDMEA  r13!, {r0-r5}; Pop from an Empty Ascending Stack
STMED  r13!, {r0-r5}; Push onto Empty Descending Stack
LDMED  r13!, {r0-r5}; Pop from an Empty Descending Stack
```

栈操作的所有指令列表

名称	Stack指令	等同指令
pre-increment load	LDMED	LDMIB
post-increment load	LDMFD	LDMIA
pre-decrement load	LDMEA	LDMDB
post-decrement load	LDMFA	LDMDA
pre-increment store	STMFA	STMIB
post-increment store	STMEA	STMIA
pre-decrement store	STMFD	STMDB
post-decrement store	STMED	STMDA

子程序（subroutine）

- 使用子程序可以提高代码的可复用性。
- 程序中，通常的子程序结构如下：



子程序（续）

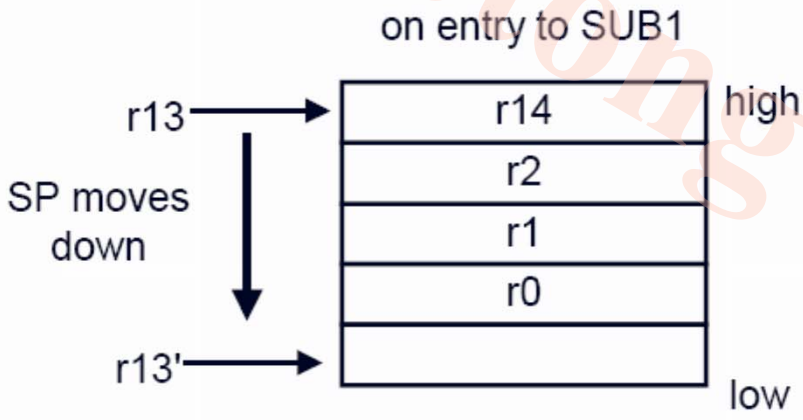
- **BL subroutine_name (Branch and Link)** 是一条跳转到子程序的指令。它执行以下操作：
  - 保存**PC**寄存器的值（下一条指令地址）到**r14**寄存器。这是返回地址。
  - 将子程序的地址装载到**PC**寄存器中。这是执行跳转操作。
- **BL**永远使用**r14**寄存器来保存返回地址，因此**r14**也被称作连接寄存器（**link register, lr or r14**）。
- 子程序返回很简单：直接将**r14**的值放回到**PC**寄存器中。

子程序嵌套

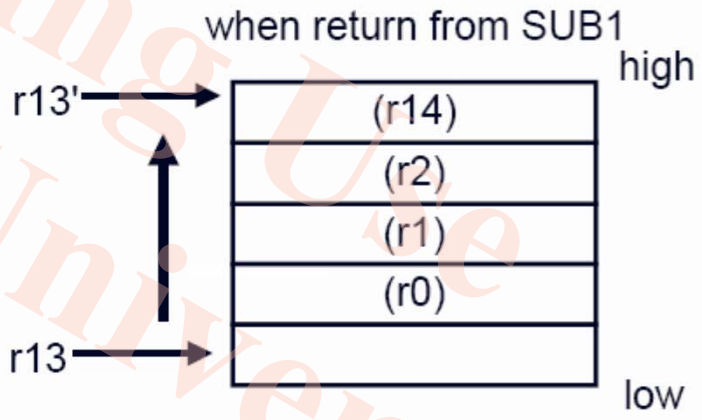
- 由于返回地址保存在寄存器**r14**中，如果不首先保存**r14**则不能再次调用子程序。
- 在软件工程的实践中表明，子程序最好不好改变任何寄存器的值，除非需要给调用程序传送结果。
- 信息隐藏的规则：尽量对调用程序隐藏子程序的所有动作。
- 使用栈来实现：
  - 保存**r14**；
  - 保存所有子程序使用寄存器的值，再返回时恢复。

通过栈实现子程序内的数据保护

```
        BL      SUB1
        .....
SUB1    STMED   r13!, {r0-r2, r14}      ; push work & link registers
        ....
        BL      SUB2                    ; jump to a nested subroutine
        ...
        LDMED   r13!, {r0-r2, r14}      ; pop work & link registers
        MOV     pc, r14                 ; return to calling program
```



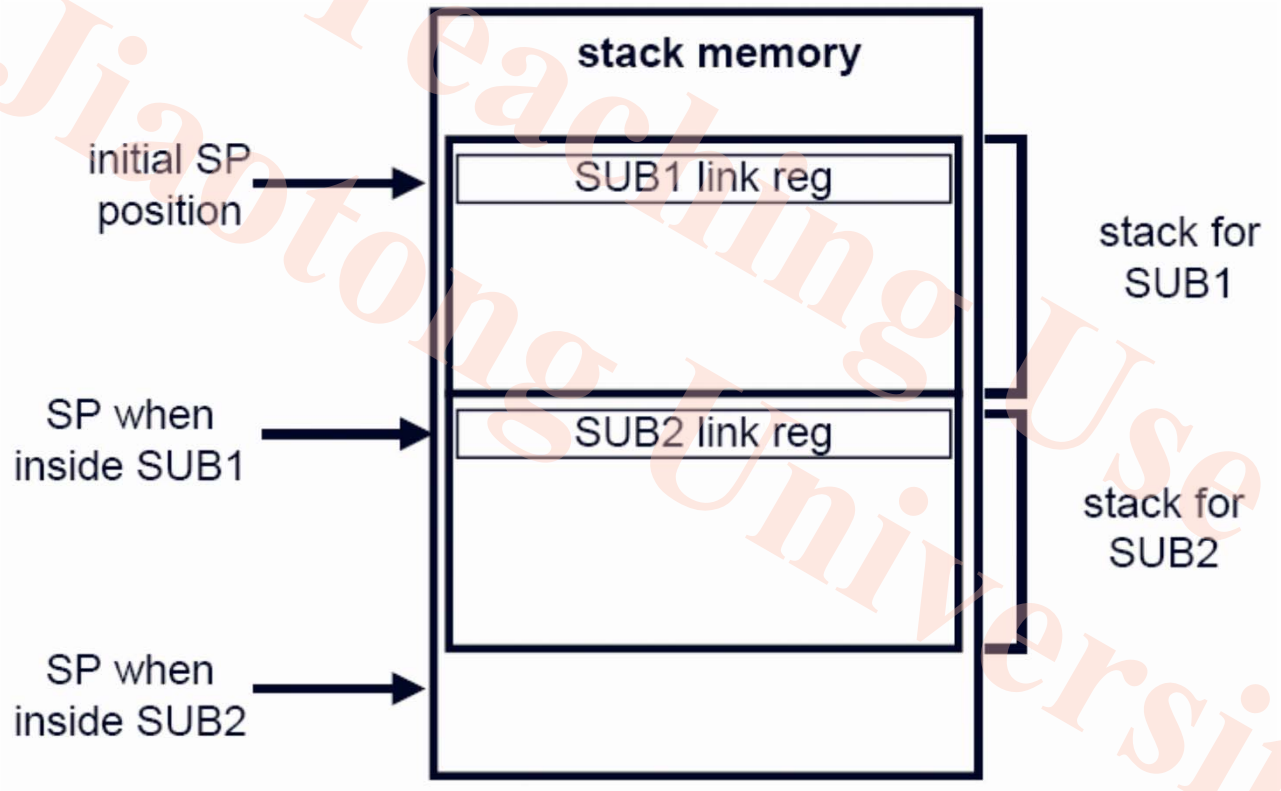
STMED r13!, {r0-r2, r14}



LDMED r13!, {r0-r2, r14}

子程序嵌套效应

- **SUB1**调用了另一个子程序**SUB2**。假设**SUB2**也保存其链接寄存器（**r14**）和工作寄存器入栈，则数据栈如下图所示：



子程序调用的完整例子

```
; Subroutine HexOut - Output 32-bit word as 8 hex digits as ASCII characters
; Input parameters:        r1 contains the 32-bit word to output
; Return parameters:       none
HexOut  STMED   r13!, {r0-r2}      ; save working registers on stack
        MOV     r2, #8             ; r2 has nibble (4-bit digit) count = 8
Loop    MOV     r0, r1, LSR #28    ; get top nibble by shifting right 28 bits
        CMP     r0, #9             ; if nibble <= 9, then
        ADDLE   r0, r0, #"0"               ;     convert to ASCII numeric char
        ADDGT   r0, r0, #"A"-10    ; else  convert to ASCII alphabet char
        SWI     SWI_WriteC         ; print character
        MOV     r1, r1, LSL #4     ; shift left 4 bits to get to next nibble
        SUBS    r2, r2, #1         ; decrement nibble count
        BNE     Loop               ; if more, do next nibble
        LDMED   r13!, {r0-r2}      ; retrieve working registers from stack
        MOV     pc,r14             ;   ... and return to calling program
        END
```

课后作业

- 学习使用**Keil MDK-ARM**
- 使用**Keil**软件编写课上的举例和课后的练习