

ARM 指令集

ARM FUNDAMENTAL

DAY07-
DAY08

“

数据处理指令

”

数据处理指令

数据处理指令可细分为4类：

数据传送指令 (MOV、MVN)

算术运算指令 (ADD、ADC、SUB、SBC、
RSB、RSC)

位运算指令 (AND、ORR、EOR、BIC)

比较指令 (CMP、CMN、TST、TEQ)

数据处理指令的一般格式：

<opcode>{cond}{S} <Rd>, <Rn>, <shifter_operand>

ADD	R0, R1, #1
MOVEQ	R0, R2
SUBEQS	R0, R1, R2, LSR #1
CMP	R1, R3



数据处理指令的地址模式

第1操作数恒为寄存器 Rn

shifter_operand 即第2操作数，是一个移位操作数，共有11种形式：

格式	说明
#<immediate>	立即数方式
<Rm>	寄存器
<Rm>, LSL #<shift_imm>	Rm逻辑左移shift_imm位
<Rm>, LSL <Rs>	Rm逻辑左移Rs位
<Rm>, LSR #<shift_imm>	Rm逻辑右移shift_imm位
<Rm>, LSR <Rs>	Rm逻辑右移Rs位
<Rm>, ASR #<shift_imm>	Rm算术右移shift_imm位
<Rm>, ASR <Rs>	Rm算术右移Rs位
<Rm>, ROR #<shift_imm>	Rm循环右移shift_imm位
<Rm>, ROR <Rs>	Rm循环右移Rs位
<Rm>, RRX	Rm扩展的循环右移一位

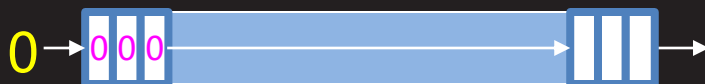


数据处理指令的移位操作

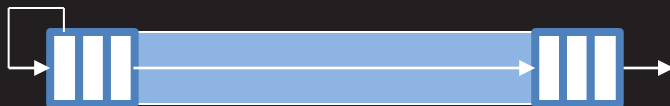
LSL移位：



LSR移位：



ASR移位：



ROR移位：



RRX移位：



shifter_operand 是立即数

指令示例

MOV R0, #0xFF

ADD R1, R2, #2

CMP R1, #0x04800000

指令码格式

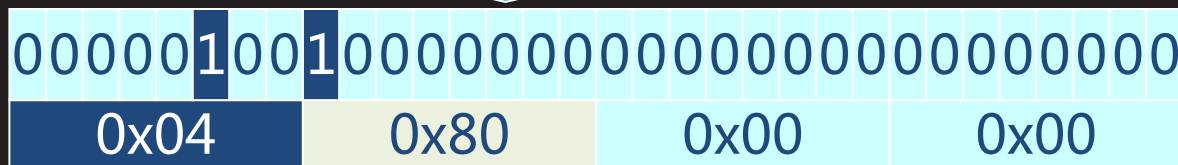
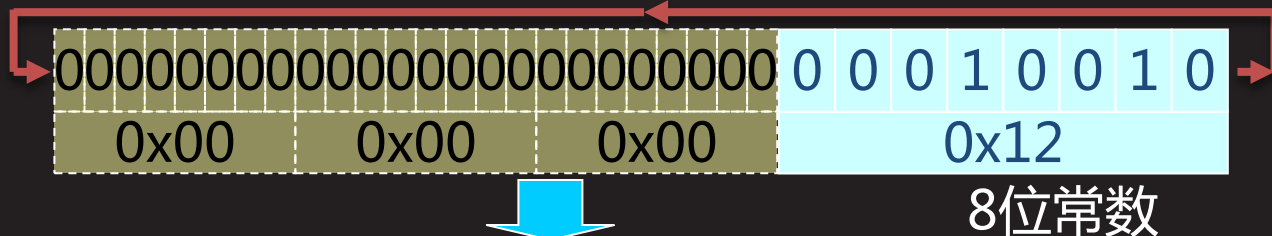
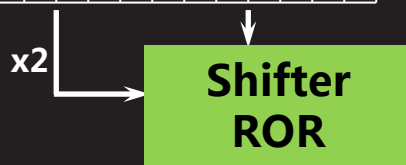
31	28	27	26	25	24	21	20	19	16	15	12	11	8	7	0
cond	0	0	1	opcode	S	Rn		Rd		rotate_imm					immed_8



shifter_operand 为立即数举例

知识讲解

MOV R1, #0x04800000



shifter_operand 为寄存器加移位方式

shifter_operand 为寄存器+移位方式格式

格式

说明

<Rm>, LSL #<shift_imm>	Rm逻辑左移shift_imm位
<Rm>, LSL <Rs>	Rm逻辑左移Rs位
<Rm>, LSR #<shift_imm>	Rm逻辑右移shift_imm位
<Rm>, LSR <Rs>	Rm逻辑右移Rs位
<Rm>, ASR #<shift_imm>	Rm算术右移shift_imm位
<Rm>, ASR <Rs>	Rm算术右移Rs位
<Rm>, ROR #<shift_imm>	Rm循环右移shift_imm位
<Rm>, ROR <Rs>	Rm循环右移Rs位
<Rm>, RRX	Rm扩展的循环右移一位

31	28	27	26	25	24	21	20	19	16	15	12	11	7	6	5	4	3	0
cond	0	0	0	opcode	S	Rn			Rd		shift_imm		0	0	0		Rm	

31	28	27	26	25	24	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond	0	0	0	opcode	S	Rn			Rd			Rs	0	0	0	1		Rm	



shifter_operand 为寄存器举例

MOV R0 , R1 ← shifter_operand
 CMP R1, R2 ← shifter_operand
 ADD R0 , R1 , R2 ← shifter_operand

31	28	27	26	25	24	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond			0	0	0	opcode		S	Rn		Rd		0	0	0	0	0	0	0	Rm	



shifter_operand 为寄存器移位操作 举例

ADD R1, R2, R3, LSL #3 ;R1=R2+R3<<3

SUB R1, R2, R3, LSR R4 ;R1=R2-R3>>R4



“

数据传输指令

”

数据传送指令：MOV

指令格式：

MOV{cond}{S} <Rd>, <shifter_operand>

shifter_operand：立即数（8位图）、寄存器、寄存器移位

示例：

MOVEQ R0, #0x80

MOVS PC, LR @CPSR = SPSR ?

MOVEQS R0, R1

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C = shifter_carry_out,

不影响 V

MOV R3, R4, LSL #2

MOV R0, R0

MOV PC, R14



数据取反传送指令：MVN

指令格式：

MVN{cond}{S} <Rd>, <shifter_operand>

shifter_operand：立即数（8位图）、寄存器、寄存器移位

示例

MVNEQ R0, #0x80

MVNS PC, R0

MVNEQS R0, R1

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C = shifter_carry_out,

不影响 V

MVN R3, R4, LSL #2



MOV/MVN指令应用举例

```
MOV    R0, #0xffffffff00
```

```
MVN    R0, #0x000000FF
```



“

算术运算指令

”

加法指令：ADD

指令格式：

ADD{cond}{S} <Rd>, <Rn>, <shifter_operand>

示例

ADD R0, R1, #0x80

ADDEQ R0, R1, R3

ADDS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C = CarryFrom(Rn + shifter_operand)

V = OverflowFrom(Rn + shifter_operand)

ADDS PC, R1, #0



ADD指令使用举例

ADD R0, R1, R2

$R0 = R1 + R2$

ADD R0, R1, #256

$R0 = R1 + 256$

ADD R0, R2, R3, LSL #1

$R0 = R2 + (R3 \ll 1)$



带进位的加法指令：ADC

指令格式：

ADC{cond}{S} <Rd>, <Rn>, <shifter_operand>

示例

ADC R0, R1, #0x80

ADCEQ R0, R1, R3

ADCS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C = CarryFrom(Rn + shifter_operand + C Flag)

V = OverflowFrom(Rn + shifter_operand + C Flag)

ADCS PC, R1, #0



ADC 指令应用举例

加数 (R1、R0) ,
被加数 (R3、R2) ,
结果 (R1、R0) ,
前者为高32位 , 后者为低32位



减法指令：SUB

指令格式：

SUB{cond}{S} <Rd>, <Rn>, <shifter_operand>

示例

SUB R0, R1, #0x80

SUBNE R0, R1, R3

SUBS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C = NOT BorrowFrom(Rn - shifter_operand)

V = OverflowFrom(Rn - shifter_operand)

SUBS PC, R14, #04



SUB指令使用举例

SUB R0, R1, R2

SUB R0, R1, #256

SUB R0, R2, R3, LSL #1

SUBS R1, R2, R3

SUBS PC, R14, #4



带借位的减法指令：SBC

指令格式：

SBC{cond}{S} <Rd>, <Rn>, <shifter_operand>

示例

SBC R0, R1, #0x80

SBCNE R0, R1, R3

SBCS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C = NOT BorrowFrom(Rn - shifter_operand - NOT(C Flag))

V = OverflowFrom(Rn - shifter_operand - NOT(C Flag))

SBCS PC, R1, #0



SBC 指令应用举例

被减数 (R1、R0)

减数 (R3、R2)

结果 (R1、R0)

前者为高32位，后者为低32位



反向减法指令：RSB

指令格式：

RSB{cond}{S} <Rd>, <Rn>, <shifter_operand>

示例

RSB R0, R1, #0x80

RSBNE R0, R1, R3

RSBS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C = NOT BorrowFrom(shifter_operand – Rn)

V = OverflowFrom(shifter_operand – Rn)

RSBS PC, R1, #0



RSB使用举例

RSB R3, R1, #0

$R3 = -R1$

RSBS R1, R2, R2, LSL #2

$R1 = (R2 \ll 2) - R2 = R2 \times 3$

影响标志位



带借位的反向减法指令：RSC Technology **Tarena** 达内科技

指令格式：

RSC{cond}{S} <Rd>, <Rn>, <shifter_operand>

示例

RSC R0, R1, #0x80

RSCNE R0, R1, R3

RSCS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C = NOT BorrowFrom(shifter_operand – Rn –
NOT(C Flag))

V = OverflowFrom(shifter_operand – Rn –
NOT(C Flag))

RSCS PC, R1, #0



RSB/RSC指令应用举例

被减数 (R1、 R0)

减数 (R3、 R2)

结果 (R1、 R0)

前者为高32位，后者为低32位



“

位运算指令

”

位运算指令

与操作指令:

AND{cond}{S} Rd, Rn, op2

$Rd \leftarrow Rn \& op2$

或操作指令

ORR{cond}{S} Rd, Rn, op2

$Rd \leftarrow Rn | op2$

异或操作指令

EOR{cond}{S} Rd, Rn, op2

$Rd \leftarrow Rn \wedge op2$

位清除指令

BIC{cond}{S} Rd, Rn, op2

$Rd \leftarrow Rn \& (\sim op2)$



位与指令：AND

指令格式：

AND{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

示例

AND R0, R1, #0x80

ANDNE R0, R1, R3

ANDS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C= shifter_carry_out

Rn	Op2	结 果
----	-----	-----

0	0	0
---	---	---

0	1	0
---	---	---

1	0	0
---	---	---

1	1	1
---	---	---



AND指令使用举例

```
AND R0 , R1 , R2
```

```
AND R0 , R1 , #0x80
```

```
ANDS R0, R1 , R2 , LSL #1
```

```
ANDEQ R3 , R4 , #0xFF
```



位或指令：ORR

指令格式：

ORR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

示例

ORR R0, R1, #0x80

ORRNE R0, R1, R3

ORRS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C= shifter_carry_out

ORRS PC, R1, #0x0

Rn Op2 结 果

0 0 0

0 1 1

1 0 1

1 1 1



ORR 指令使用举例

```
ORR    R0, R1, #1
```

```
ORR    R0, R1, R2
```

```
ORR    R0, R1, R2, LSL #2
```



异或指令：EOR

指令格式：

EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

示例

EOR R0, R1, #0x80

EOREQ R0, R1, R3

EORS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C= shifter_carry_out

EORS PC, R1, #0x0

Rn	Op2	结 果
----	-----	-----

0	0	0
---	---	---

0	1	1
---	---	---

1	0	1
---	---	---

1	1	0
---	---	---



EOR 使用举例

```
EOR    R0, R0, #FF
```

```
EOR    R0, R0, R1
```

```
EORS   R0, R0, #1
```



位清除指令：BIC

指令格式：

BIC{cond}{S} <Rd>, <Rn>, <shifter_operand>

示例

BIC R0, R1, #0x80

BICEQ R0, R1, R3

BICS R0, R1, R2

N = Rd[31]

Z = if Rd == 0 then 1 else 0

C= shifter_carry_out

Rn	Op2	结 果
0	0	0
0	1	0
1	0	1
1	1	0



BIC 指令使用举例

```
BIC    R0, R0, #0xFF
```

```
BIC    R1 , R2 , R3
```



“

比较测试指令

”

比较测试指令

比较指令

CMP{cond} Rn, operand2

影响N、Z、C、 $V \leftarrow Rn - operand2$

负数比较指令

CMN{cond} Rn, operand2

影响N、Z、C、 $V \leftarrow Rn + operand2$

位测试指令

TST{cond} Rn, operand2

影响N、Z、C、 $V \leftarrow Rn \& operand2$

相等测试指令

TEQ{cond} Rn, operand2

影响N、Z、C、 $V \leftarrow Rn \wedge operand2$



比较指令：CMP

指令格式

CMP{cond} <Rn>, <shifter_operand>

示例

CMP R0, #0x80

CMPEQ R0, R3

N=ALU_out[31]

Z = if ALU_out == 0 then 1 else 0

C=NOT BorrowFrom(Rn - shifter_operand)

V=OverflowFrom(Rn - shifter_operand)



CMP 指令使用举例

```
cmp    r1, #10  
moveq  r0, r1  
bleq   test
```

.....

test:

.....



负值比较指令：CMN

指令格式

CMN{cond} <Rn>, <shifter_operand>

示例

CMN R0, #0x80

CMNEQ R0, R3

N=ALU_out[31]

Z = if ALU_out == 0 then 1 else 0

C = CarryFrom(Rn + shifter_operand)

V = OverflowFrom(Rn + shifter_operand)



CMN 使用举例

```
CMN    R1 , # 5 ;
```

```
BEQ    SKIP
```

```
.....
```

```
SKIP:
```

```
.....
```



位测试指令：TST

指令格式：

TST{cond} Rn, shifter_operand

示例

TST R0, #0x80

TSTEQ R0, R3

N=ALU_out[31]

Z = if ALU_out == 0 then 1 else 0

C = shifter_carry_out



TST 指令使用举例

```
TST    R0, #0x1  
ADDEQ  R1 , R2 , R3  
TST    R1 , R4
```



相等测试指令：TEQ

指令格式：

TEQ{cond} <Rn>, <shifter_operand>

示例

TEQ R0, #0x80

TEQNE R0, R3

N=ALU_out[31]

Z = if ALU_out == 0 then 1 else 0

C = shifter_carry_out



TEQ 指令使用举例

```
TEQ    R0,R1  
MOVNE  R0,R1
```



最大公约数求解

如果有一个自然数A能被自然数B整除，则称A为B的倍数，B为A的约数。

几个自然数公有的约数，叫做这几个自然数的公约数。

公约数中最大的一个公约数，称为这几个自然数的最大公约数。

这里我们采用循环减法来计算两个自然数的最大公约数。假定我们计算X和Y的最大公约数是L，那么 $X=m \times L$ ， $Y=n \times L$ ，计算X和Y的最大公约数可以采取：

- 1、如果X和Y不相等，则计算X和Y的差值Z
- 2、用Z取代X和Y中的大者
- 3、比较X和Y，如果相等则找到了最大公约数L
- 4、如果X和Y不相等则重复1、2、3，直到找到L。

“

ARM 加载/存储指令

”

ARM加载/存储指令概述

为什么需要加载/存储指令？

数据处理指令只处理寄存器和立即数

为了实现与存储器进行交互

加载/存储指令能做什么？

访问存储器缓冲区中的数据，如变量。

访问处理器外设。

加载函数地址到PC寄存器，则实现程序跳转功能，可实现跳转表等。



ARM加载/存储指令分类

加载存储指令包括

单寄存器操作指令

字和无符号字节加载存储指令

半字和有符号字节加载存储指令

多寄存器操作指令



单寄存器加载指令

指令语法格式

说明

LDR {cond} **Rd**, addressing

加载字数据

LDR {cond} **B** **Rd**, addressing

加载无符号字节数据

LDR {cond} **H** **Rd**, addressing

加载无符号半字数据

LDR {cond} **SB** **Rd**, addressing

加载有符号字节数据

LDR {cond} **SH** **Rd**, addressing

加载有符号半字数据



单寄存器存储指令

指令语法格式	说明
STR {cond} Rd, addressing	存储字数据
STR {cond} B Rd, addressing	存储字节数据
STR {cond} H Rd, addressing	存储半字数据



单寄存器字和无符号字节 加载/存储指令

LDR/LDRB：从内存中读取一个字或字节数据存入寄存器中。

STR/STRB：将寄存器中的一个字或字节数据保存到内存中。

指令的语法格式：

LDR{cond} Rd, <地址模式>

将指定地址单元的字数据读入Rd中。

STR{cond} Rd, <地址模式>

将Rd中的字数据保存到指定地址单元中。

LDR{cond}B Rd, <地址模式>

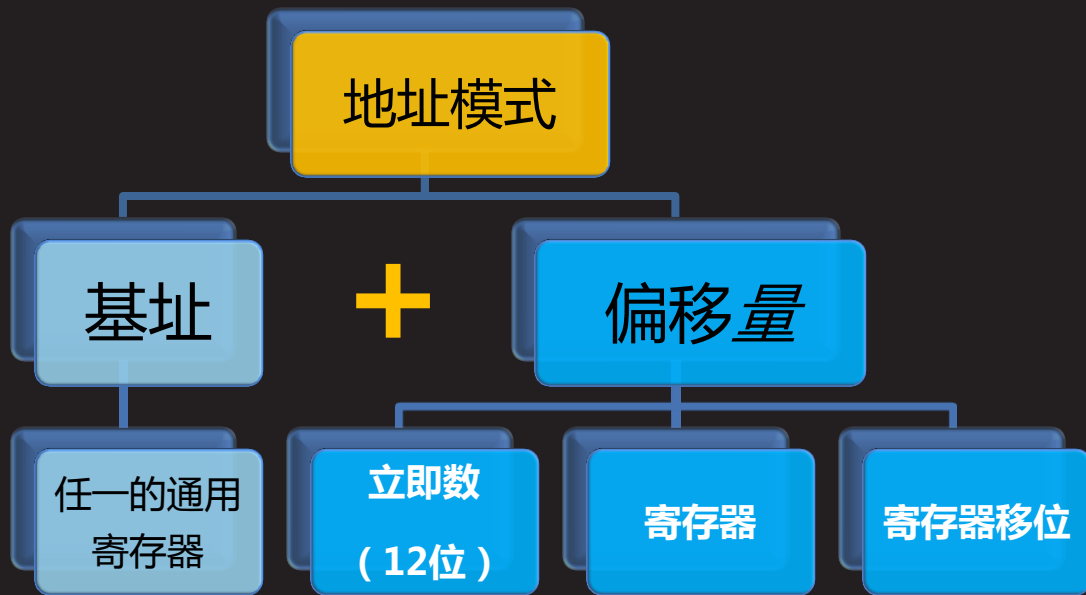
将指定地址单元中的一个字节数据读入Rd中。

STR{cond}B Rd, <地址模式>

将Rd中的一个字节数据保存到指定地址单元中。



单寄存器字和无符号字节 加载/存储指令 地址模式



单寄存器字和无符号字节 加载存储地址模式

1. 零偏移

LDR R0, [R1]

2. [$\langle Rn \rangle$, $\# + / - \langle \text{offset}_{12} \rangle$]

LDR R0, [R1, #0x8] ; R0 \leftarrow [R1+0x8]

LDR R0, [R1, #-0x20] ; R0 \leftarrow [R1 - 0x20]

3. [$\langle Rn \rangle$, $+ / - \langle Rm \rangle$]

LDR R0, [R1, R2] ; R0 \leftarrow [R1+R2]

LDR R0, [R1, -R2] ; R0 \leftarrow [R1-R2]

4. [$\langle Rn \rangle$, $+ / - \langle Rm \rangle$, $\langle \text{shift} \rangle \# \langle \text{shift}_{imm} \rangle$]

LDR R0, [R1, R2, LSL #2] ; R0 \leftarrow [R1+R2*4]

5. [$\langle Rn \rangle$, $\# + / - \langle \text{offset}_{12} \rangle$]!

LDR R0, [R1, #0x8]! ; R0 \leftarrow [R1+0x8] R1=R1+8

注：先索引（前变址）



单寄存器字和无符号字节 加载存储地址模式

6. [$\langle Rn \rangle$, $+/-\langle Rm \rangle$]!

LDR $R0, [R1, R2]$! ; $R0 \leftarrow -[R1+R2]$ $R1=R1+R2$

7. [$\langle Rn \rangle$, $+/-\langle Rm \rangle$, $\langle \text{shift} \rangle \# \langle \text{shift_imm} \rangle$]!

LDR $R0, [R1, R2, LSL \#2]$! ; $R0 \leftarrow -[R1+R2*4]$ $R1=R1+R2*4$

8. [$\langle Rn \rangle$], $\# +/-\langle \text{offset_12} \rangle$

LDR $R0, [R1], \#0x20$; $R0 \leftarrow -[R1]$ $R1=R1+0x20$

9. [$\langle Rn \rangle$], $+/-\langle Rm \rangle$

LDR $R0, [R1], R2$; $R0 \leftarrow -[R1]$ $R1=R1+R2$

10. [$\langle Rn \rangle$], $+/-\langle Rm \rangle$, $\langle \text{shift} \rangle \# \langle \text{shift_imm} \rangle$

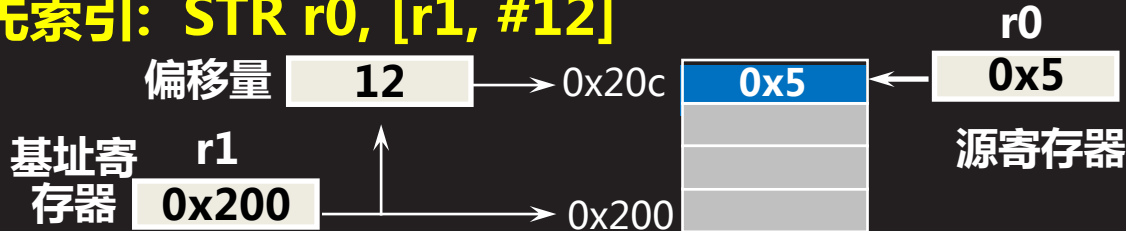
LDR $R0, [R1], R2, LSL \#2$; $R0 \leftarrow -[R1]$ $R1=R1+R2*4$

注：先索引（前变址）、后索引（后变址）



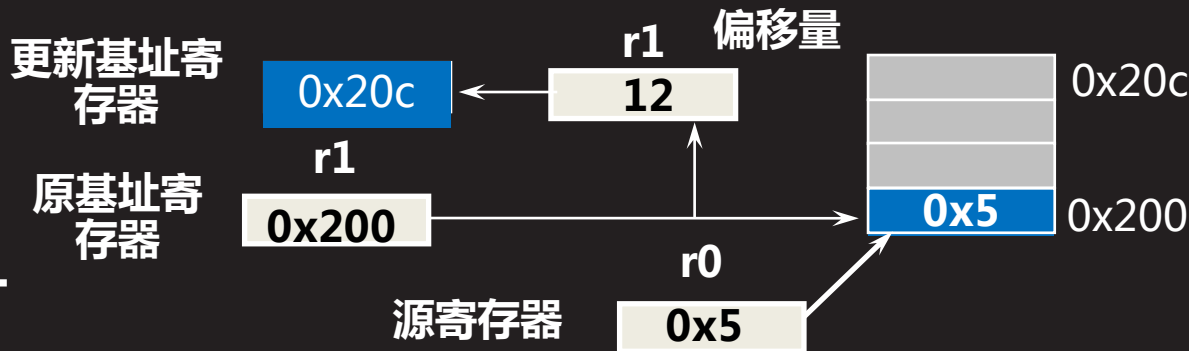
先索引、后索引

先索引: STR r0, [r1, #12]



更新基址寄存器时使用: STR r0, [r1, #12]!

后索引: STR r0, [r1], #12



单寄存器字和无符号字节 加载指令

语法格式

LDR{cond} Rd, <地址模式>
LDR{<cond>}B Rd, <地址模式>

示例

```
LDR  R1, [R2]
LDR  PC, [R0, #8]
    address[1:0]=0
    V5 or Not , PC = ?
LDRB R1, [R2], #1
LDR  R1, [PC, R3]
LDR  R1 , [R2, R3, LSL #2]
LDREQB R1 , [R2, R3]
LDR  R0, [R0, #8]!
```

@x



单寄存器字和无符号字节 存储指令

语法格式

STR{cond} Rd, <地址模式>

STR{<cond>}B Rd, <地址模式>

示例

STR R1, [R2]

STR R1, [R2], #1

STR R1, [R2, R3]

STRB R0, [R1, R2, ASR #2]

STREQB R0, [R1, R2, LSL #2]

STR **PC**, [R0, #8]

[PC, #8] 单元内容

STR **R0**, [**R0**, #8]! @x



半字和有符号字节 加载存储指令

半字和有符号字节加载/存储指令

LDR{cond}SB Rd,<地址模式>

LDR{cond}SH Rd,<地址模式>

LDR{cond}H Rd,<地址模式>

STR{cond}H Rd,<地址模式>

寄存器高位扩展

半字读写时，指定地址必须是2字节对齐，否则结果不可预知



半字和有符号字节加载存储指令寻址方式

1. 零偏移

LDRSH R0, [R1]

2. [$\langle Rn \rangle$, #+/- $\langle \text{offset}_8 \rangle$]

LDRH R0, [R1, #0x04]

3. [$\langle Rn \rangle$, +/- $\langle Rm \rangle$]

LDRSB R0, [R1, R2]

4. [$\langle Rn \rangle$, #+/- $\langle \text{offset}_8 \rangle$]!

LDRH R0, [R1, #0x08]!

5. [$\langle Rn \rangle$, +/- $\langle Rm \rangle$]!

LDRSB R0, [R1, R2]!

6. [$\langle Rn \rangle$], #+/- $\langle \text{offset}_8 \rangle$

LDRSH R0, [R1], #0x4

7. [$\langle Rn \rangle$], +/- $\langle Rm \rangle$

LDRSH R0, [R1], R2

基址 + 偏移量方式

**与字和无符号
字节加载存储
指令偏移量表
示有何不同？**



半字和有符号字节加载指令

语法格式

LDR{cond}H Rd, <地址模式>

LDR{cond}SH Rd, <地址模式>

LDR{cond}SB Rd, <地址模式>

使用示例

LDRH R1, [R0]

LDRSH R8, [R3, #2]

LDREQH R12, [R13, #-6]

LDRSB R7, [R6, #-1]!

LDRH R3, [R9], #2

LDRSB R1, [R2], R3

LDRH PC, [R0] @ x

LDRH R0, [R0], #4 @ x

LDRSB PC, [R0] @ x

LDRSB R0, [R0], #4 @ x



单寄存器半字存储指令

语法格式

STR{cond}H Rd,<地址模式>

使用示例

STRH R1, [R0]

STRH R8, [R3, #2]

STREQH R12, [R13, #-6]

STRH R7, [R6, #-2]!

STRH R3, [R9], #2

STRH R1, [R2], R3

STRH PC, [R0] @x

STRH R0, [R0], #4 @x

STRH R7, [R6, #-1] @ ?



“

多寄存器加载/存
储指令

”

多寄存器加载存储指令

多寄存器加载指令 LDM

LDM{cond}{addressing_mode} Rb{!}, < Reglist >{^}

多寄存器存储指令 STM

STM{cond} {addressing_mode} Rb{!}, < Reglist >{^}

cond : 条件域

addressing_mode

LDMIA / STMIA	Increment After (先操作, 后增加)
LDMIB / STMIB	Increment Before (先增加, 后操作)
LDMDA / STMDA	Decrement After (先操作, 后递减)
LDMDB / STMDB	Decrement Before (先递减, 后操作)

Rb : 基址寄存器

! : 更新基址寄存器

Reglist: 源/目标寄存器列表 (可以是16个寄存器的任何子集)

^ : 有两种作用, 特权模式下使用用户模式下的寄存器,
CPSR=SPSR



多寄存器存储加载指令（2）

语法:

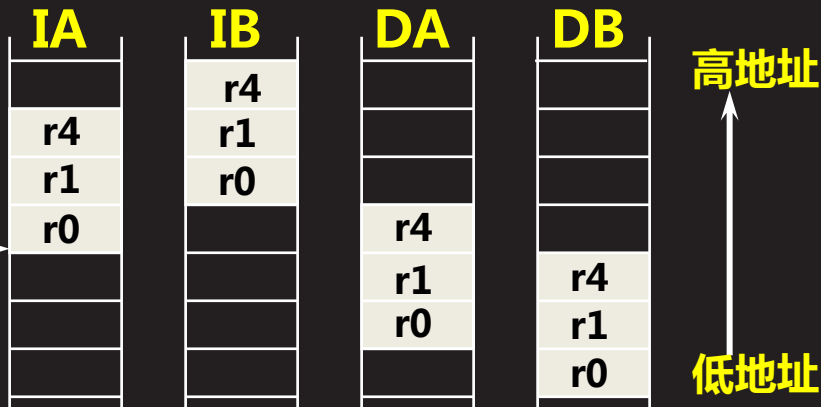
<LDM|STM>{<cond>}<addressing_mode> Rb{!}, <register list>{^}

4种地址模式

LDM IA / STM IA	后增加
LDM IB / STM IB	先增加
LDM DA / STM DA	后减小
LDM DB / STM DB	先减小

LDM**xx** r10, {r0,r1,r4}
STM**xx** r10, {r0,r1,r4}

基址寄存器 (Rb) **r10**



LDM指令使用举例

LDMIA	R0!, {R1-R3}	
LDMIB	R0, {R1-R3, R7}^	@ x usr/sys
LDMDB	SP!, {R1-R3, PC}^	@ x usr/sys
LDMDB	R0, {R0-R2}	
LDMDA	R15 , {R1}	@ x
LDMDB	R0! , { R0 -R2}	@ x Rn值



STM指令使用举例

STMIA	R0!, {R1-R3}	
STMIA	SP!, {R1-R3, LR}	
STMIB	R0, {R1-R3, R9}^	@x usr/sys
STMDB	R0, {R0-R2}	
STMDB	R0! , { R0 -R2}	
STMDA	R15 , {R1}	@ x



内存块拷贝 (1)

使用单寄存器加载存储指令实现

; r8 源数据起始地址

; r9 源数据结束地址

; r10 目标地址

loop:

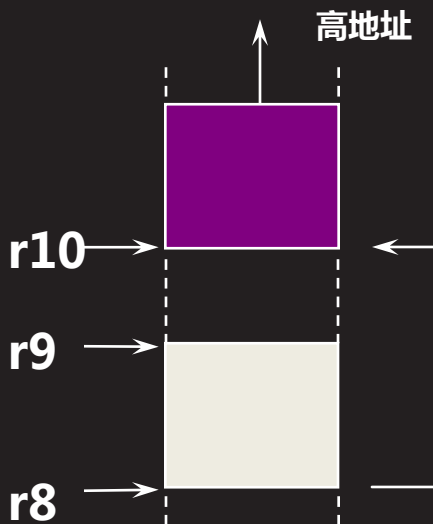
ldr r0, [r8], #4

str r0, [r10], #4

cmp r8, r9

blt loop

每个循环复制一个字



内存块拷贝(2)

使用多寄存器加载存储指令实现内存块拷贝

; r8 源数据起始地址

; r9 源数据结束地址 ; 前提R9-r8 为8的倍数

; r10 目标地址

loop:

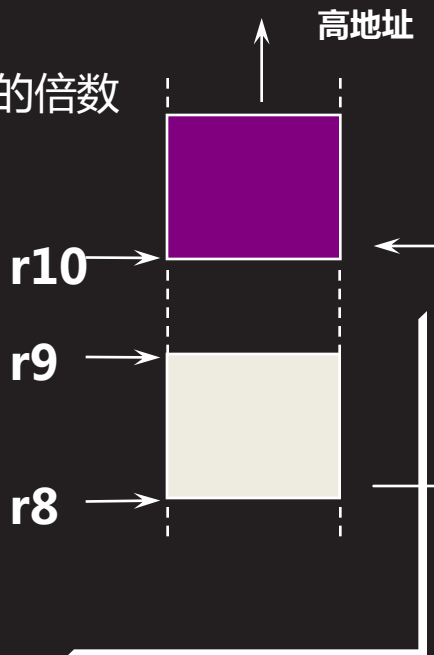
ldmia r8!, {r0-r7}

stmia r10!, {r0-r7}

cmp r8, r9

blt loop

每个循环拷贝8个字



“

ARM 栈操作

”

ARM中栈种类

Descending stacks (减栈)

栈向内存地址减小的方向变化

Ascending stacks (加栈)

栈向内存地址增加的方向变化

Full stacks (满栈)

栈指针指向的栈顶保存有效元素

Empty stacks (空栈)

栈指针指向的栈顶未保存有效元素

综合以上两种特点，有以下4种栈

FD (Full Descending)

ED (Empty Descending)

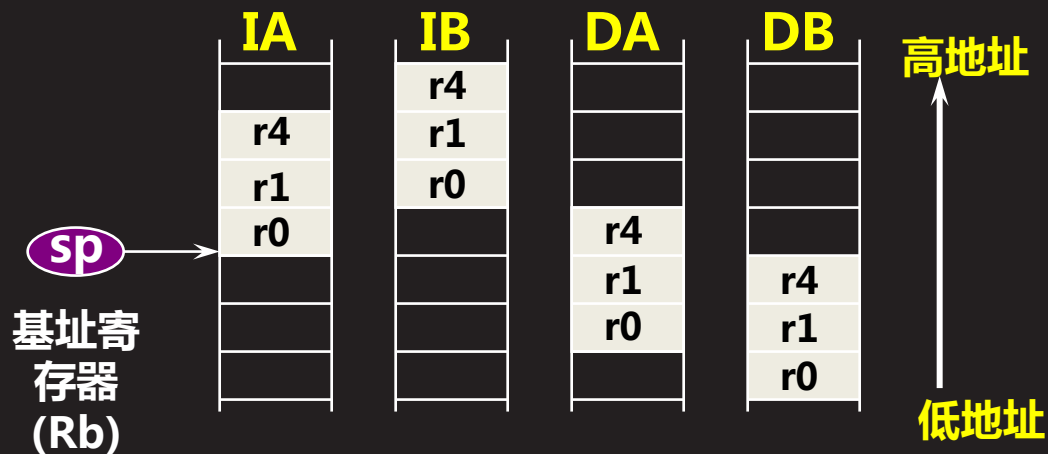
FA (Full Ascending)

EA (Empty Ascending)



ARM中栈种类(2)

LDM_{xx} sp!, {r0,r1,r4}
STM_{xx} sp!, {r0,r1,r4}



栈操作指令

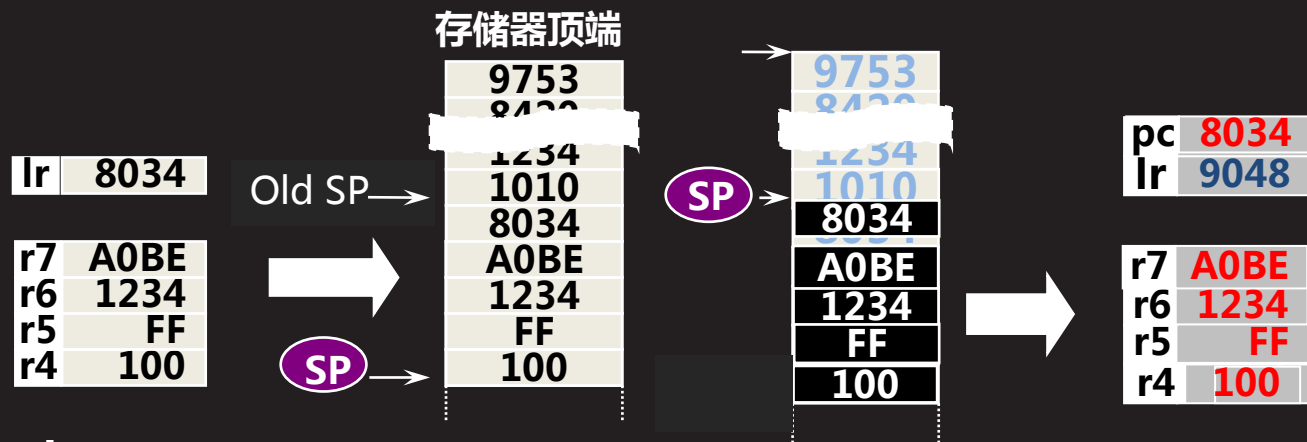
ARM的栈操作指令是通过块传输指令实现的（多寄存器传输指令）

STMFD (Push) Full Descending stack [多寄存器存储 - STMDB]

LDMFD (Pop) Full Descending stack [多寄存器加载 - LDMIA]

STMFD sp!, {r4-r7, lr}

LDMFD sp!, {r4-r7, pc}



“

状态寄存器访问指令

”

MRS程序状态寄存器读指令

MRS指令语法格式

MRS{cond} Rd,psr

cond 条件码

Rd目标寄存器 (不能是R15)

Psr 程序状态寄存器 (CPSR , SPSR)

MRS{<cond>} <Rd>, CPSR

MRS{<cond>} <Rd>, SPSR

示例：

MRS R0, CPSR ;将CPSR状态寄存器读取，保存到R0中

MRS R1, SPSR ;将SPSR状态寄存器读取，保存到R1中



MSR程序状态寄存器写指令

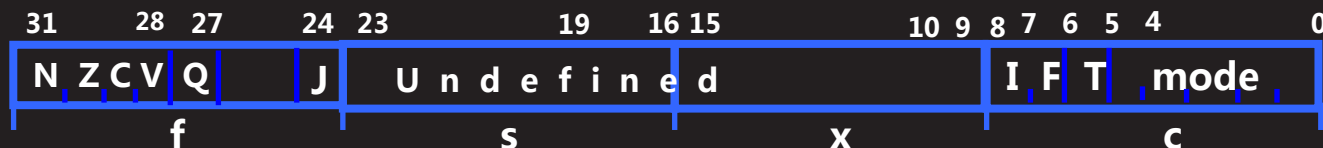
指令的语法格式如下

MSR{<cond>} CPSR_<fields>, #<immediate_8>

MSR{<cond>} CPSR_<fields>, <Rm>

MSR{<cond>} SPSR_<fields>, #<immediate_8>

MSR{<cond>} SPSR_<fields>, <Rm>



域可以为以下字母（必须小写）的一个或者组合

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])

immediate_8 : 8位图立即数



程序状态寄存器访问指令 示例

```
MRS  R0, CPSR
BIC  R0, R0, #0xF0000000
MSR  CPSR_f, R0
```

```
@ Read the CPSR
@Clear the N, Z, C and V bits
@ Update the flag bits in the CPSR
@ N, Z, C and V flags now all clear
```

```
MRS  R0, CPSR
ORR  R0, R0, #0x80
MSR  CPSR_c, R0
```

```
@Read the CPSR
@Set the interrupt disable bit
@ Update the control bits in the CPSR
@ interrupts (IRQ) now disabled
```

```
MRS  R0, CPSR
BIC  R0, R0, #0x1F
ORR  R0, R0, #0x11
MSR  CPSR_c, R0
```

```
@Read the CPSR
@ Clear the mode bits
@ Set the mode bits to FIQ mode
@ Update the control bits in the CPSR
@ now in FIQ mode
```

+ MSR CPSR_c, #0x11

ARM寄存器到协处理器寄存器： MCR

指令格式

MCR{cond} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm> {, opcode_2}

示例

MCR p15, 0, r0, c1, c0, 0

写CP15的C1寄存器的0号物理寄存器 (Control Register)

; ARM register transfer to Coproc 15

; opcode 1 = 0, opcode 2 = 0

; ARM source register = r0

; coproc dest registers are 1 and 0



ARM CORTEX-A8手册截图

To access the Control Register, read or write CP15 with:

```
MRC p15, 0, <Rd>, c1, c0, 0 ; Read Control Register
```

Copyright © 2006-2010 ARM Limited. All rights reserved.
Non-Confidential

3-46

System Control Coprocessor

```
MCR p15, 0, <Rd>, c1, c0, 0 ; Write Control Register
```

Table 3-48 shows the behavior of the processor caching instructions or data for the I bit and C bit of the *c1*, *Control Register* on page 3-44 and the L2EN bit of the *c1*, *Auxiliary Control Register*.



协处理器寄存器到ARM寄存器： MRC

指令格式

MRC{cond} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm> {, opcode_2}

示例

MRC p15, 0, R0, c1, c0, 0

读CP15的C1寄存器的0号物理寄存器 (Control Register)

; Coproc 15 transfer to ARM register

; opcode 1 = 0, opcode 2 = 0

; ARM destination register = R0

; coproc source registers are 1 and 0



MRC & MCR 使用举例

mrc	p15, 0, r0, c1, c0, 0	
bic	r0, r0, #0x00002000	@ clear bits 13 (--V-)
bic	r0, r0, #0x00000007	@ clear bits 2:0 (-CAM)
orr	r0, r0, #0x00000002	@ set bit 1 (--A-) Align
orr	r0, r0, #0x00000800	@ set bit 12 (Z---) BTB
mcr	p15, 0, r0, c1, c0, 0	



“

ARM 伪指令

”

ARM伪指令

ARM伪指令不属于ARM指令集中的指令。

定义这些指令可以使ARM汇编程序设计变得更方便。

ARM伪指令可以像其他ARM指令一样使用。

汇编器会自动用一条或多条ARM指令替换ARM伪指令。

ARM的伪指令包括

ADR 伪指令

ADRL 伪指令

LDR 伪指令（两种写法）

NOP 伪指令



小范围地址加载伪指令 ADR

语法格式

ADR{cond} register, expr

cond : 条件码

register : 目标寄存器, 如: R0等

expr : 地址表达式 (相对于pc或寄存器)

加载地址范围

地址字对齐时: +/-1020 bytes (255×4)

地址非字对齐时: +/-255bytes

示例

...	...
ADR R1, Delay	0x20 ADD R1, PC,#0x3c
...	...
Delay:	...
MOV R0, R14	0x64 MOV R0, R14
...	...



伪指令 ADRL

语法格式

ADRL{cond} register, expr

cond : 条件码

register : 目标寄存器, 如: R0等

expr : 地址表达式 (相对于pc或寄存器)

加载地址范围

地址字对齐时: -256K ~ 256K

地址非字对齐时: -64K ~ 64K

用两条指令实现

示例

...

ADRL R1, Delay

...

Delay:

MOV R0, r14

...



LDR伪指令

语法格式

LDR{cond} register,=[expr | label_expr]

从指令位置到文字池的偏移量必须小于4KB

示例

...

LDR R1, =TestData

LDR R0, [R1]

...

TestData:

.word 0x12345678

示例2

LDR pc, =label



LDR伪指令2

语法格式

LDR{cond} register, label_expr

从指令位置到文字池的偏移量必须小于4KB

示例

...

LDR R1, TestData

...

TestData:

.word 0x12345678

示例2

LDR pc, jmp_table

jmp_table:

.word func_addr



NOP伪指令

NOP伪指令在汇编时将会被代替成ARM中的空操作，比如可能是“MOV R0,R0”指令等。NOP可用于延时操作。

示例

```
MOV    R1,#0x1234
```

Delay:

```
NOP
```

```
NOP
```

```
SUBS   R1,R1,#1
```

```
BNE    Delay;
```

```
MOV     PC,LR
```



“

GNU ARM 汇编语法

”

GNU ARM 汇编语法格式

[<label>:][<instruction or directive or pseudo-instruction>} @comment

instruction :指令

directive :伪操作

pseudo-instruction :伪指令

<label>: 为标号

GNU汇编中，任何以冒号结尾的标识符
都被认为是一个标号，而不一定非要
在一行的开始。

Comment 为语句的注释

main: mov r1, r0 @主程序入口



GNU ARM 汇编示例

test.s示例

.text @指定代码放到.text段

.global do_sub @声明全局标号

do_sub:

sub r0, r0, r1 @两个数相减

mov pc, lr @子程序返回

.end



汇编语言伪操作

伪操作 (directive)

汇编语言程序里的一些特殊助记符为编程方便，完成一些辅助功能的操作。

在对汇编源程序进行汇编过程中由汇编程序处理，而不是在程序运行期间由机器执行。

也就是说,这些伪操作只在汇编过程中起作用，汇编结束，伪操作作用消失。



GNU ARM汇编伪操作

GNU ARM环境下主要的伪操作

常量定义伪操作

符号声明伪操作

数据定义伪操作

汇编控制伪操作

信息报告伪操作

其他伪操作



符号定义伪操作

常量定义伪操作.equ

语法格式:

.equ symbol, expr

其中：

symbol 为要指定的名称，它可以是以前定义过的符号；

expr 表示数字常量或程序中的标号。

示例

.equ TEST_NUM, #0x20



符号定义伪操作

声明全局常量伪操作.global或.globl

语法格式:

.global symbol

.globl symbol

其中：

symbol 为要声明的全局变量名称

示例

.global start



符号定义伪操作

声明外部常量伪操作.extern

语法格式:

.extern symbol

其中

symbol 为要声明的外部变量名称

示例

.extern main



数据定义伪操作

字节定义.byte

语法格式:

.byte expr {, expr }...

其中:

expr 数字表达式或程序中的标号。

示例

.byte 20



数据定义伪操作

半字定义.hword或.short

语法格式:

.hword expr {, expr }....

.short expr {, expr }...

其中：

expr 数字表达式或程序中的标号



数据定义伪操作

字定义.word或.int或.long

语法格式:

.word expr {, expr }....

.int expr {, expr }...

.long expr {, expr }...

其中：

expr 数字表达式或程序中的标号。



数据定义伪操作

字符串定义.ascii和.asciz或.string

语法格式:

.ascii expr {, expr }...

.asciz expr {, expr }...

.string expr {, expr }...

其中：

expr 表示字符串。



数据定义伪操作

固定填充字节内存单元定义.space或.skip

语法格式:

.space size {, value}

.skip size {, value}

其中：

size 所分配的字节数



汇编与反汇编代码控制伪操作

指令集类型标识伪操作

.arm

.code 32

.thumb

.code 16



汇编与反汇编代码控制伪操作

段定义伪操作，语法格式如下：

```
.section <section_name> { " <flags> " }
```

section_name : 段名称

可以是自己定义的名称，也可以是预定义的段名称.text、.data、.bss中的一个

flags : ELF 文件格式的标志

<Flag> 可以是

a 可加载段

w 可写段

x 可执行段



汇编与反汇编代码控制伪操作

GNU ARM预定义的段

具体的语法格式如下：

.text

.data

.bss



再看一下test.s示例

```
.text          @指定代码放到.text段  
.global do_sub @声明全局标号
```

```
do_sub:  
    sub r0, r0, r1    @两个数相减  
    mov pc, lr        @子程序返回  
.end
```



内存中字符串比较

比较内存中两个字符串，实现类似C语言
strcmp函数功能，函数可命名为a_strcmp



“

GNU ARM汇编与
C混合编程、
ATPCCS标准

”

ARM C/C++和汇编混合编程

C/C++和汇编混合编程可以实现

- 可实现在c中无法实现的处理器功能

- 使用新的或不支持的指令

- 产生更高效的代码

C/C++汇编混合引用的内容

- 函数、变量

- C/C++汇编混合引用的方式

- C内嵌汇编、C和汇编相互引用

- C和汇编相互引用的规则与实现



ATPCS (arm/thumb程序调用规范)

规定寄存器使用

规定函数传参方式

规定栈使用方法



寄存器使用 (1)

作为函数传递的参数值

规定寄存器使用
规定函数传参方式
规定栈使用方法

寄存器变量
必须保护

Scratch register
(corruptible)

Stack Pointer
Link Register
Program Counter

Register

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9/sb
r10/sl
r11
r12
r13/sp
r14/lr
r15/pc

编译器使用一套规则的来设置寄存器的用法

ARM-Thumb Procedure Call
Standard or **ATPCS** (or APCS)

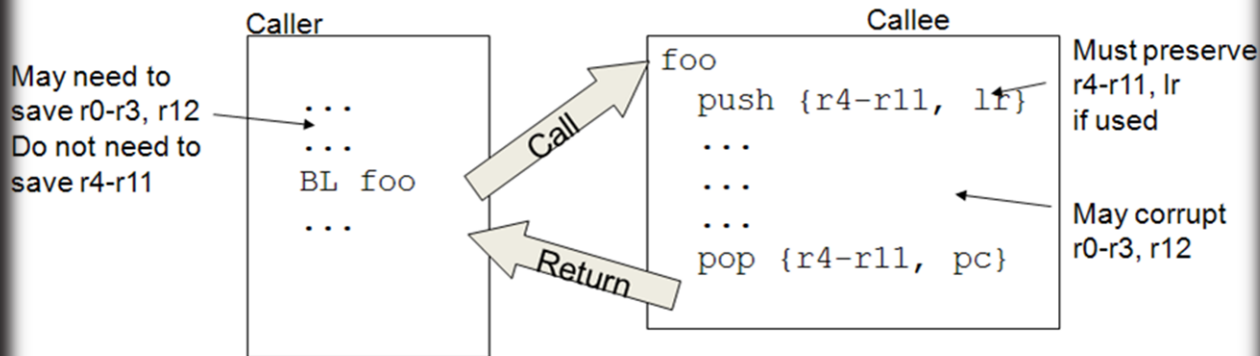
CPSR 标志位可被函数调用所破坏
一些和编译过的代码交互工作的汇编码
在接口层必须满足ATPCS的规范

- 如果 RWPI选项有效，作为栈的基地址
- 如果软件堆栈检查有效，作为栈的限制值



寄存器使用 (2)

参数通过r0-r3传递



Value returns in r0 for int/short/char;
in r0 and r1 for long long/double



参数传递(1)

开始四个字大小的参数直接使用寄存器的R0-R3来传递(快速且高效的) [ATPCS]

如果需要更多的参数，将使用堆栈。(需要额外的指令和慢速的存储器操作)

所以通常限制参数的个数，使它为4或更少。
如果不可避免，把常用的参数放在前4个



参数传递(2)

Parameter Passing (4 parameters)

```
int func1(int a, int b, int c, int d)
{
    return a+b+c+d;
}

int caller1(void)
{
    return func1(1,2,3,4);
}
```

```
func1
0x000000 : ADD    r0,r0,r1
0x000004 : ADD    r0,r0,r2
0x000008 : ADD    r0,r0,r3
0x00000c : MOV     pc,lr
```

```
caller1
0x000014 : MOV     r3,#4
0x000018 : MOV     r2,#3
0x00001c : MOV     r1,#2
0x000020 : MOV     r0,#1
0x000024 : B       func1
```

Parameter Passing (6 parameters)

```
func2
0x000000 : STR     lr, [sp,#-4]!
0x000004 : ADD     r0,r0,r1
0x000008 : ADD     r0,r0,r2
0x00000c : ADD     r0,r0,r3
0x000010 : LDMIB   sp,{r12,r14}
0x000014 : ADD     r0,r0,r12
0x000018 : ADD     r0,r0,r14
0x00001c : LDR     pc,{sp},#4
```

```
caller2
0x000020 : STMFD    sp!,{r2,r3,lr}
0x000024 : MOV     r3,#6
0x000028 : MOV     r2,#5
0x00002c : STMIA    sp,{r2,r3}
0x000030 : MOV     r3,#4
0x000034 : MOV     r2,#3
0x000038 : MOV     r1,#2
0x00003c : MOV     r0,#1
0x000040 : BL      func2
0x000044 : LDMFD    sp!,{r2,r3,pc}
```

This code is compiled with "-O2 -Ono_autoinline"



参数传递(3)

AAPCS 64-bit类型规则

64-bit 类型必须8-byte对齐

64-bit 参数传递必须通过偶数+紧邻的奇数寄存器

(i.e. $r0+r1$ or $r2+r3$) 或者通过8字节对齐的栈传递

如果不是按最优的顺序给出参数，寄存器和栈将会被浪费



fx(int a, double b, int c)



fy(int a, int c, double b)



fz(double a, double b, int c, double d)



Remember the hidden *this* argument in r0 for non-static C++ member functions



ATPCS栈使用规则

ARM 栈种类

FD	(Full Descending)	满递减
ED	(Empty Descending)	空递减
FA	(Full Ascending)	满递增
EA	(Empty Ascending)	空递增

ATPCS规定数据栈为FD（满递减）类型，并且对栈中数据的是8字节对齐的。



C和ARM汇编程序间相互调用

在C和ARM汇编程序之间相互调用须遵守ATPCS规则。
C和汇编之间的相互调用可以：


- 汇编代码中访问C代码中定义的全局常、变量
- C代码中访问汇编代码中定义的全局常、变量
- 在C代码中调用汇编代码中的函数
- 在汇编代码中调用C代码中的函数



C调用汇编程序

- 在汇编文件中定义汇编子程序，并声明成全局标号
- 在C代码中直接引用函数（声明调用）
 - C程序中声明函数原型
 - 如果用C++编译器可以使用 **extern "C"** 声明函数
- 将C程序和汇编代码一起编译、汇编、连接

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



```
                .text  
                .global mystrcopy  
  
mystrcopy:  
    LDRB r2, [r1], #1  
    STRB r2, [r0], #1  
    CMP  r2, #0  
    BNE  mystrcopy  
    BX   lr
```



SHELL混合调用验证

修改Shell控制程序：将字符串比较函数替换成汇编写的。

1、掌握C语言程序调用汇编程序的方法

