ARM指令集详解

> 引用:
>
> [ARM指令集详解]

1. 汇编

1.1. 通用寄存器

user mode	Privileged Mode					
	system	exception				
usr	sys	svc	abt	und	irq	fiq
R0 <----> R7						
R8 <----> R12						R8_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC						
CPSR						
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

寄存器类别	寄存器在汇编中的名称	各模式下实际访问的寄存器						
		用户	系统	管理	中止	未定义	中断	快中断
通用寄存器和程序计数器	R0(a1)	R0						
	R1(a2)	R1						
	R2(a3)	R2						
	R3(a4)	R3						
	R4(v1)	R4						
	R5(v2)	R5						
	R6(v3)	R6						
	R7(v4)	R7						
	R8(v5)	R8						R8_fiq *
	R9(SB,v6)	R9						R9_fiq *
	R10(SL,v7)	R10						R10_fiq *
	R11(FP,v8)	R11						R11_fiq *
	R12(IP)	R12						R12_fiq *
	R13(SP)	R13		R13_svc*	R13_abt *	R13_und *	R13_irq *	R13_fiq *
	R14(LR)	R14		R14_svc *	R14_abt *	R14_und *	R14_irq *	R14_fiq *

	R15(PC)	R15						
状态寄存器	R16(CPSR)	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

ARM状态各模式下的寄存器

寄存器类别	寄存器在汇编中的名称	各模式下实际访问的寄存器						
		用户	系统	管理	中止	未定义	中断	快中断
通用寄存器和程序计数器	R0 (a1)	R0						
	R1 (a2)	R1						
	R2 (a3)	R2						
	R3 (a4)	R3						
	R4 (v1)	R4						
	R5 (v2)	R5						
	R6 (v3)	R6						
	R7 (v4)	R7						
	R8 (v5)	R8						R8_fiq *
	R9 (SB, v6)	R9						R9_fiq *
	R10 (SL, v7)	R10						R10_fiq *
	R11 (FP, v8)	R11						R11_fiq *
	R12 (IP)	R12						R12_fiq *
	R13 (SP)	R13		R13_svc*	R13_abt *	R13_und *	R13_irq *	R13_fiq *
	R14 (LR)	R14		R14_svc *	R14_abt *	R14_und *	R14_irq *	R14_fiq *
	R15 (PC)	R15						
状态寄存器	R16 (CPSR)	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

通用寄存器

37个寄存器，31个通用寄存器，6个状态寄存器。

R12是内部调用暂时寄存器 ip。它在过程链接胶合代码（例如，交互操作胶合代码）中用于此角色。在过程调用之间，可以将它用于任何用途。被调用函数在返回之前不必恢复 r12。

R13堆栈指针sp，R14返回指针，R15为PC指针, cpsr_c代表的是这32位中的低8位，也就是控制位

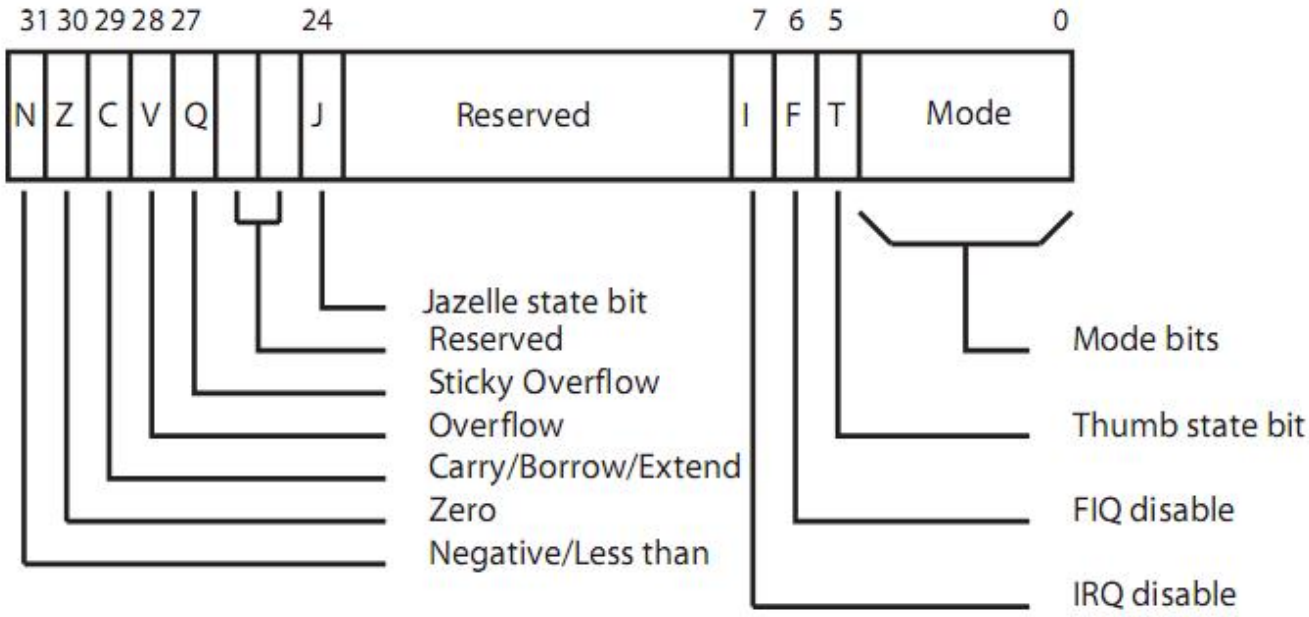
CPSR有4个8位区域：标志域（F）、状态域（S）、扩展域（X）、控制域（C） MSR - Load specified fields of the CPSR or SPSR with an immediate constant, orfrom the contents of a general-purpose register. Syntax: MSR{cond} _, #immed_8rMSR{cond} _, Rm where: cond is an optional condition code. is either CPSR orSPSR. specifies the field or fields to be moved. can be one or more of: ccontrol field mask byte (PSR[7:0]) x extension field mask byte (PSR[15:8]) sstatus field mask byte (PSR[23:16) f flags field mask byte (PSR[31:24]).immed_8r is an expression evaluating to a numeric constant. The constant mustcorrespond to an 8-bit pattern rotated by an even number of bits within a32-bit word. Rm is the source register.

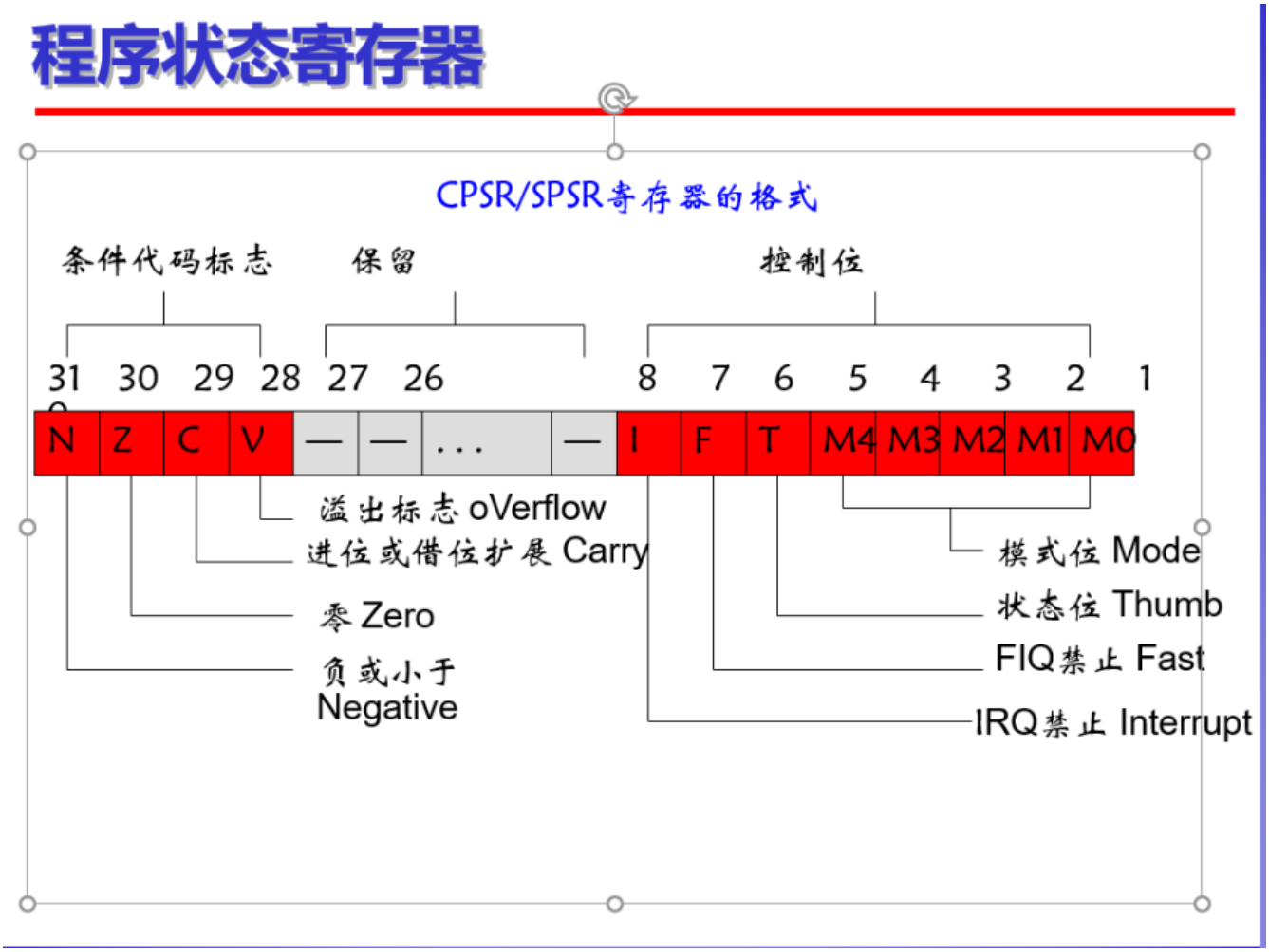
C 控制域屏蔽字节(psr[7:0])

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])





CPSR寄存器

FIQ和IRQ的区别?

MODE(以下为二进制)		可见的ARM状态寄存器
10000	用户模式	PC,CPSR,R0~R14
10001	FIQ	PC,CPSR,SPSR_fiq，R14_fiq~R8_fiq,R7~R0
10010	IRQ	PC,CPSR,SPSR_irq，R14_irq~R13_irq,R12~R0
10011	管理模式(svc)	PC,CPSR,SPSR_svc，R14_svc~R13_svc,R12~R0
10111	终止模式	PC,CPSR,SPSR_abt，R14_abt~R13_abt,R12~R0
11011	未定义	PC,CPSR,SPSR_und，R14_und~R13_und,R2~R0
11111	系统模式(sys)	PC,CPSR，R14 ~R0

1.2. 指令格式

1) 基本格式

`<opcode>{<cond>}{S} <Rd>,<Rn>{,<opcode2>}`

其中，<>内的项是必须的，{}内的项是可选的，如是指令助记符，是必须的，而{}为指令执行条件，是可选的，如果不写则使用默认条件AL(无条件执行)。

opcode 指令助记符，如LDR，STR 等

cond 执行条件，如EQ，NE 等

S 是否影响CPSR 寄存器的值，书写时影响CPSR，否则不影响

Rd 目标寄存器

Rn 第一个操作数的寄存器

operand2 第二个操作数

指令格式举例如下：

LDREX--这条指令主要是从memory中取一个数，然后放到register中，但是相比普通的LDR指令，在于其内在的原子操作特性, 信号量和spin lock这些东西最核心的事情基本上就是load-update-store序列，为了防止并发，必须保证这个序列是原子的，所谓原子，即处理器在执行这个指令序列时，得绝对占有处理器而不能够被切换出去。在ARM上，从V6开始，指令LDREX和STREX就是用来干这事的

LDR R0,[R1] ;读取R1 地址上的存储器单元内容，执行条件AL BEQ DATAEVEN ;跳转指令，执行条件EQ，即相等跳转到DATAEVEN ADDS R1,R1,#1 ;加法指令，R1 + 1 = R1 影响CPSR 寄存器，带有S SUBNES R1,R1,#0xD;条件执行减法运算(NE)，R1-0xD=>R1，影响CPSR 寄存器，带有S

2) 第2个操作数

在ARM 指令中，灵活的使用第2个操作数能提高代码效率，第2个操作数的形式如下：

＃immed_8r

常数表达式：

该常数必须对应8 位位图，即常数是由一个8 位的常数循环移位偶数位得到。

合法常量：

0x3FC、0、0xF0000000、200、0xF0000001等都是合法常量。

非法常量：

0x1FE、511、0xFFFF、0x1010、0xF0000010等都是非法常量。

> 常数表达式应用举例如下：

`MOV R0,#1 ;R0=1`

AND R1,R2,#0x0F ;R2 与0x0F，结果保存在R1

LDR R0，[R1],#-4 ;读取R1 地址上的存储器单元内容，且R1 = R1 - 4

Rm

寄存器方式，在寄存器方式下操作数即为寄存器的数值。

寄存器方式应用举例：

SUB R1，R1，R2 ； R1-R2= > R1

MOV PC，R0 ； PC=R0，程序跳转到指定地址

LDR R0，[R1]，-R2 ； 读取R1 地址上的存储器单元内容并存入R0，且R1=R1-R2

Rm， shift

寄存器移位方式。将寄存器的移位结果作为操作数，但RM 值保存不变，移位方法如下：

ASR #n 算术右移n 位（1≤n≤32）

LSL #n 逻辑左移n 位（1≤n≤31）

LSR #n 逻辑左移n 位（1≤n≤32）

ROR #n 循环右移n 位（1≤n≤31）

RRX 带扩展的循环右移1位

type Rs 其中，type 为ASR，LSL，和ROR 中的一种；Rs 偏移量寄存器，低8位有效，若其值大于或等于32，则第2个操作数的结果为0（ASR、ROR例外）。 寄存器偏移方式应用举例：

ADD R1，R1，R1，LSL #3 ； R1=R1*9

SUB R1，R1，R2，LSR#2 ； R1=R1-R2*4

R15 为处理器的程序计数器PC，一般不要对其进行操作，而且有些指令是不允许使用R15，如UMULL 指令。 （3）条件码 使用指令条件码，可实现高效的逻辑操作，提高代码效率。表A-1给出条件码表。

表A-1 条件码表

对于Thumb指令集，只有B 指令具有条件码执行功能，此指令条件码同表A-?，但如果为无条件执行时，条件码助记符“AL”不在指令中书写。

条件码应用举例如下：

比较两个值大小，并进行相应加1 处理，C 代码为： if（a > b）a++ ； else b++ ；对应的ARM 指令如下。其中R0为a，R1为b。 CMP R0，R1 ; R0 与R1 比较 ADDHI R0，R0，#1 ; 若R0 > R1，则R0=R0+1 ADDLS R1，R1，#1 ;若R0 < =R1，则R1=R1+1 若两个条件均成立，则将这两个数值相加，C代码为：

If（a!=10）&&(b!=20))a=a+b;

对应的ARM 指令如下，其中R0 为a，R1为b。 CMP R0,#10 ; 比较R0 是否为10 CMPNE R1,#20 ; 若R0 不为10，则比较R1 是否20 ADDNE R0,R0,R1 ; 若R0 不为10 且R1 不为20，指令执行，R0=R0+R1

1.3. 指令集

1.3.1. ARM 存储器访问指令

ARM 处理是加载/存储体系结构的典型的RISC处理器，对存储器的访问只能使用加载和存储指令实现。ARM 的加载/存储指令是可以实现字、半字、无符/有符字节操作；批量加载/存储指令可实现一条指令加载/存储多个寄存器的内容，大大提高效率；SWP指令是一条寄存器和存储器内容交换的指令，可用于信号量操作等。ARM 处理器是冯?诺依曼存储结构，程序空间、RAM 空间及IO 映射空间统一编址，除对对RAM 操作以外，对外围IO、程序数据的访问均要通过加载/存储指令进行。表A-2给出ARM存储访问指令表。

表A-2 ARM 存储访问指令表

助记符	说明	操作	条件码位置
LDR Rd,addressing	加载字数据	Rd←[addressing],寻址索引	LDR{cond}
LDRB Rd,addressing	加载无符字节数据	Rd←[addressing],寻址索引	LDR{cond}B
LDRT Rd，addressing	以用户模式加载字数据	Rd←[addressing],寻址索引	LDR{cond}T
LDRBT Rd，addressing	以用户模式加载无符号字数据	Rd←[addressing],寻址索引	LDR{cond}BT
LDRH Rd，addressing	加载无符半字数据	Rd←[addressing],寻址索引	LDR{cond}H
LDRSB Rd，addressing	加载有符字节数据	Rd←[addressing],寻址索引	LDR{cond}SB
LDRSH Rd，addressing	加载有符半字数据	Rd←[addressing],寻址索引	LDR{cond}SH
STR Rd，addressing	存储字数据	[addressing]←Rd,寻址索引	STR{cond}
STRB Rd，addressing	存储字节数据	[addressing]←Rd,寻址索引	STR{cond}B
STRT Rd，addressing	以用户模式存储字数据	[addressing]←Rd, 寻址索引	STR{cond}T
SRTBT Rd，addressing	以用户模式存储字节数据	[addressing]←Rd,寻址索引	STR{cond}BT
STRH Rd，addressing	存储半字数据	[addressing]←Rd, 寻址索引	STR{cond}H
LDM{mode} Rn{!},reglist	批量(寄存器)加载	reglist←[Rn…],Rn回存等	LDM{cond}{more}
STM{mode} Rn{!},rtglist	批量(寄存器)存储	[Rn…]←reglist,Rn 回存等	STM{cond}{more}
SWP Rd,Rm,Rn	寄存器和存储器字数据交换	Rd←[Rd],[Rn]←[Rm](Rn≠Rd或Rm)	SWP{cond}
SWPB Rd,Rm,Rn	寄存器和存储器字节数据交换	Rd←[Rd],[Rn]←[Rm](Rn≠Rd或Rm)	SWP{cond}B

LDR 和STR

加载/存储字和无符号字节指令。使用单一数据传送指令(STR 和LDR)来装载和存储单一字节或字的数据从/到内存。LDR指令用于从内存中读取数据放入寄存器中；STR 指令用于将寄存器中的数据保存到内存。指令格式如下：`LDR{cond}{T} Rd,<地址>;` 加载指定地址上的数据(字)，放入Rd中 `STR{cond}{T} Rd,<地址>;` 存储数据(字)到指定地址的存储单元，要存储的数据在Rd中 `LDR{cond}B{T} Rd,<地址>;` 加载字节数据，放入Rd中，即Rd最低字节有效，高24位清零 `STR{cond}B{T} Rd,<地址>;` 存储字节数据，要存储的数据在Rd，最低字节有效 其中，T 为可选后缀，若指令有T，那么即使处理器是在特权模式下，存储系统也将访问看成是处理器是在用户模式下。T在用户模式下无效，不能与前索引偏移一起使用T。 LDR/STR 指令寻址是非常灵活的，由两部分组成，一部分为一个基址寄存器，可以为任一个通用寄存器，另一部分为一个地址偏移量。地址偏移量有以下3种格式： (1) 立即数。立即数可以是一个无符号数值，这个数据可以加到基址寄存器，也可以从基址寄存器中减去这个数值。指令举例如下： `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) (2)寄存器。寄存器中的数值可以加到基址寄存器，也可以从基址寄存器中减去这个数值。指令举例值。指令举例如下：`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。寄存器移位后的值可以加到基址寄存器，也可以从基址寄存器中减去这个数值。指令举例如下： `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) 从寻址方式的地址计算方法分，加载/存储指令有以下4 种形式： (1)零偏移。Rn 的值作为传送数据的地址，即地址偏移量为0。指令举例如下： `LDR Rd,[Rn]` (2)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。指令举例如下： `LDR Rd,[Rn,#0x04]!` `LDR Rd,[Rn,#-0x04]` (3)程序相对偏移。程序相对偏移是索引形式的另一个版本。汇编器由PC 寄存器计算偏移量，并将PC寄存器作为Rn 生成前索引指令。不能使用后缀“！”。指令举例如下： `LDR Rd,label` ;label 为程序标号，label 必须是在当前指令的±4KB范围内 (4) 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` 地址对准--大多数情况下，必须保证用于32 位传送的地址是32 位对准的。 加载/存储字和无符号字节指令举例如下： `LDR R2,[R5]` ;加载R5 指定地址上的数据(字)，放入R2 中 `STR R1,[R0,#0x04]` ;将R1 的数据存储到R0+0x04存储单

元，R0 值不变 `LDRB R3,[R2],#1` ；读取R2 地址上的一字节数据，并保存到R3中，R2=R3+1 `STRB R6,[R7]` ；读R6 的数据保存到R7 指定的地址中，只存储一字节数据 加载/存储半字和带符号字节。这类LDR/STR 指令可能加载带符字节\加载带符号半字、加载/存储无符号半字。偏移量格式、寻址方式与加载/存储字和无符号字节指令相同。指令格式如下： `LDR{cond}SB Rd,<地址>` ；加载指定地址上的数据(带符号字节)，放入Rd中 `LDR{cond}SH Rd,<地址>` ；加载指定地址上的数据(带符号字节)，放入Rd中 `LDR{cond}H Rd,<地址>` ；加载半字数据，放入Rd中，即Rd最低16位有效,高16位清零 `STR{cond}H Rd,<地址>` ；存储半字数据，要存储的数据在Rd，最低16位有效 说明：带符号位半字/字节加载是指带符号位加载扩展到32 位;无符号位半字加载是指零扩展到32位。 地址对准--对半字传送的地址必须为偶数。非半字对准的半字加载将使Rd 内容不可靠，非半字对准的半字存储将使指定地址的2字节存储内容不可靠。

加载/存储半字和带符号字节指令举例如下：

`LDRSB R1[R0,R3]` ；将R0+R3地址上的字节数据读出到R1，高24 位用符号位扩展 `LDRSH R1,[R9]` ；将R9 地址上的半字数据读出到R1，高16位用符号位扩展 `LDRH R6,[R2],#2` ；将R2 地址上的半字数据读出到R6，高16位用零扩展，R2=R2+1 `SHRH R1,[R0,#2]!`; 将R1 的数据保存到R2+2 地址中，只存储低2字节数据，R0=R0+2 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、外设的控制操作等等，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。 变量的访问 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 … 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 … 程序散转 … 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 …

LDM和STM

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。指令格式如下： `LDM{cond}<模式> Rn{!},reglist{^}` `STM{cond}<模式> Rn{!},reglist{^}` LDM /STM 的主要用途是现场保护、数据复制、参数传送等。其模式有8种，如下所列：(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 其中，寄存器Rn 为基址寄存器，装有传送数据的初始地址，Rn 不允许为R15;后缀“！”表示最后的地址写回到Rn中;寄存器列表reglist 可包含多于一个寄存器或寄存器范围，使用“,”分开，如{R1,R2,R6-R9},寄存器排列由小到大排列；“^”后缀不允许在用户模式呈系统模式下使用，若在LDM指令用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 拷贝到CPSR 中，这可用于异常处理返回；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。 地址对准——这些指令忽略地址的位[1：0]。 批量加载/存储指令举例如下： `LDMIA R0!,{R3-R9}` ；加载R0 指向的地址上的多字数据，保存到R3～R9中，R0 值更新 `STMIA R1!,{R3-R9}` ；将R3～R9 的数据存储到R1 指向的地址上，R1值更新 `STMFD SP!,{R0-R7,LR}` ；现场保存，将R0～R7、LR入栈 `LDMFD SP!,{R0-R7,PC}^`; 恢复现场，异常处理返回 在进行数据复制时，先设置好源数据指针，然后使用块拷贝寻址指令LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB 进行读取和存储。而进行堆栈操作时，则要先设置堆栈指针，一般使用SP 然后使用堆栈寻址指令STMFD/LDMFD、STMED。LDMED、STMFA/LDMFA、STMEA/LDMEA实现堆栈操作。 多寄存器传送指令示意图如图A-1所示，其中R1为指令执行前的基址寄存器，R1’则为指令执行完后的基址寄存器。

（a）指令STMIA R1!,{R5-R7} （b）指令STMIB R1!,{R5-R7}

（c）指令STMDA R1!, {R5-R7} （d）指令STMDB R1!,{R5-R7} 图A-1 多寄存器传送指令示意图

数据是存储在基址寄存器的地址之上还是之下，地址是在存储第一个值之前还是之后增加还是减少。表A-3给出多寄存器传送指令映射示意表。

表A-3 多寄存器传送指令映射示意表

```
    使用LDM/STM 进行数据复制例程如下：
    …
    LDR R0,=SrcData ;//设置源数据地址
    LDR R1,=DstData ;//设置目标地址
    LDMIA R0,{R2-R9} ;//加载8 字数据到寄存器R2～R9
    STMIA R1,{R2-R9} ;//存储寄存器R2～R9 到目标地址
```

使用LDM/STM 进行现场寄存器保护，常在子程序中或异常处理使用： SENDBYTE STMFD SP!,{R0-R7,LR} ;寄存器入堆 … BL DELAY ;调用DELAY 子程序 … LDMFD SP!,{R0-R7,PC} ;恢复寄存器，并返回

SWP

寄存器和存储器交换指令。SWP指令用于将一个内存单元（该单元地址放在寄存器Rn中）的内容读取到一个寄存器Rd中，同时将另一个寄存器Rm 的内容写入到该内存单元中。使用SWP 可实现信号量操作。 指令格式如下：`SWP{cond}{B} Rd,Rm,[Rn]` 其中，B 为可选后缀，若有B，则交换字节，否则交换32 位字：Rd 为数据从存储器加载到的寄存器;Rm的数据用于存储到存储器中，若Rm 与Rn 相同，则为寄存器与存储器内容进行交换;Rn 为要进行数据交换的存储器地址，Rn 不能与Rd 和Rm 相同。 SWP 指令举例如下： SWP R1,R1,[R0] ; 将R1 的内容与R0 指向的存储单元的内容进行交换 SWP R1,R2,,[R0] ; 将R0 指向的存储单元内容读取一字节数据到R1中(高24 位清零) ; 并将R2 的内容写入到该内存单元中(最低字节有效) 使用SWP 指令可以方便地进行信号量的操作： 12C_SEM EQU 0x40003000 … 12C_SEM_WAIT MOV R0,#0 LDR R0,=12C_SEM SWP R1,R1,[R0] ;取出信号量，并设置其为0 CMP R1,#0 ;判断是否有信号 BEQ 12C_SEM_WAIT ;若没有信号，则等待

1.3.2. ARM 数据处理指令

数据处理指令大致可分为3 类：

（1） 数据传送指令（如MOV、MVN）

（2） 算术逻辑运算指令（如ADD,SUM,AND）

（3） 比较指令（如CMP、TST）。

数据处理指令只能对寄存器的内容进行操作。 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。ARM数据处理指令列于表A-4中。

表A-4 ARM 数据处理指令

```
    （1）数据传送指令
**MOV**
    数据传送指令。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：
    MOV{cond}{S} Rd,operand2
    MOV 指令举例如下：
    MOV R1#0x10 ;R1=0x10
    MOV R0,R1 ;R0=R1
    MOVS R3,R1,LSL #2 ;R3=R1＜＜2，并影响标志位
    MOV PC,LR   ;PC=LR ，子程序返回
```

MVN

数据非传送指令。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： `MVN{cond}{S} Rd,operand2` MVN 指令举例如下： MVN R1,#0xFF ;R1=0xFFFFFF00 MVN R1,R2 ;将R2 取反，结果存到R1

（2）算术逻辑运算指令

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： `ADD{cond}{S} Rd,Rn,operand2` ADD 指令举例如下： ADDS R1,R1,#1 ;R1=R1+1 ADD R1,R1,R2 ;R1=R1+R2 ADDS R3,R1,R2,LSL #2 ;R3=R1+R2 < < 2

SUB

减法运算指令。用寄存器Rn 减去operand2。结果保存到Rd 中。指令格式如下： `SUB{cond}{S} Rd,Rn,operand2` SUB 指令举例如下： SUBS R0,R0,#1 ;R0=R0-1 SUBS R2,R1,R2 ;R2=R1-R2 SUB R6,R7,#0x10 ;R6=R7-0x10

RSB

逆向减法指令。用寄存器operand2 减法Rn，结果保存到Rd 中。指令格式如下： `RSB{cond}{S} Rd,Rn,operand2` SUB 指令举例如下： RSB R3,R1,#0xFF00 ;R3=0xFF00-R1 RSBS R1,R2,R2,LSL #2 ;R1=R2 < < 2-R2=R2×3 RSB R0,R1,#0 ;R0=-R1

ADC

带进位加法指令。将operand2 的数据与Rn 的值相加，再加上CPSR中的C 条件标志位。结果保存到Rd 寄存器。指令格式如下： `ADC{cond}{S} Rd,Rn,operand2` ADC 指令举例如下： ADDS R0,R0,R2 ADC R1,R1,R3 ;使用ADC 实现64 位加法，(R1、R0)=(R1、R0)+（R3、R2）

SBC

带进位减法指令。用寄存器Rn 减去operand2，再减去CPSR 中的C条件标志位的非（即若C 标志清零，则结果减去1），结果保存到Rd 中。指令格式如下： `SCB{cond}{S}Rd,Rn,operand2` SBC 指令举例如下： SUBS R0，R0，R2 SBC R1，R1，R3 ;使用SBC 实现64 位减法，(R1,R0)-(R3,R2)

RSC

带进位逆向减法指令。用寄存器operand2 减去Rn，再减去CPSR 中的C条件标志位，结果保存到Rd 中。指令格式如下： `RSC{cond}{S} Rd,Rn,operand2` RSC 指令举例如下： RSBS R2,R0,#0 RSC R3,R1,#0 ;使用RSC 指令实现求64 位数值的负数

AND

逻辑与操作指令。将operand2 值与寄存器Rn 的值按位作逻辑与操作，结果保存到Rd中。指令格式如下： `AND{cond}{S} Rd,Rn,operand2` AND 指令举例如下： ANDS R0,R0,#x01 ;R0=R0&0x01，取出最低位数据 AND R2,R1,R3 ;R2=R1&R3

ORR

逻辑或操作指令。将operand2 的值与寄存器Rn的值按位作逻辑或操作，结果保存到Rd 中。指令格式如下：`ORR{cond}{S} Rd,Rn,operand2` ORR 指令举例如下： ORR R0,R0,#x0F ;将R0 的低4 位置1 MOV R1,R2,LSR #4 ORR R3,R1,R3,LSL #8 ;使用ORR 指令将近R2 的高8位数据移入到R3 低8 位中

EOR

逻辑异或操作指令。将operand2 的值与寄存器Rn 的值按位作逻辑异或操作，结果保存到Rd中。指令格式如下：`EOR{cond}{S}Rd,Rn,operand2` EOR 指令举例如下： EOR R1,R1,#0x0F ;将R1 的低4 位取反 EOR R2,R1,R0 ;R2=R1^R0 EORS R0,R5,#0x01 ;将R5 和0x01 进行逻辑异或，结果保存到R0，并影响标志位

BIC

位清除指令。将寄存器Rn 的值与operand2 的值的反码按位作逻辑与操作，结果保存到Rd中。指令格式如下：`BIC{cond}{S}Rd,Rn,operand2` BIC 指令举例如下： BIC R1,R1,#0x0F ;将R1 的低4 位清零，其它位不变 BIC R1,R2,R3 ;将拭的反码和R2 相逻辑与，结果保存到R1

（3）比较指令

CMP

比较指令。指令使用寄存器Rn 的值减去operand2 的值，根据操作的结果更新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： `CMP{cond} Rn,operand2` CMP 指令举例如下：CMP R1,#10 ;R1 与10 比较，设置相关标志位 CMP R1,R2 ;R1 与R2 比较，设置相关标志位 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。

CMN

负数比较指令。指令使用寄存器Rn 与值加上operand2 的值，根据操作的结果更新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： `CMN{cond} Rn,operand2` CMN R0,#1 ;R0+1，判断R0 是否为1 的补码，若是Z 置位 CMN 指令与ADDS 指令的区别在于CMN 指令不保存运算结果。CMN指令可用于负数比较，比如CMNR0，#1 指令则表示R0 与-1 比较，若R0 为-(即1 的补码)，则Z 置位，否则Z复位。

TST

位测试指令。指令将寄存器Rn 的值与operand2 的值按位作逻辑与操作，根据操作的结果更新CPSR中相应的条件标志位(当结果为0时，EQ位被设置)，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 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。

TEQ

相等测试指令。指令寄存器Rn 的值与operand2 的值按位作逻辑异或操作，根据操作的结果更新CPSR中相应条件标志位，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： `TEQ{cond} Rn,operand2` TEQ 指令举例如下： TEQ R0,R1 ;比较R0 与R1 是否相等(不影响V 位和C 位) TST 指令与EORS 指令的区别在于TST 指令不保存运算结果。使用TEQ进行相等测试，常与EQNE 条件码配合使用，当两个数据相等时，EQ 有效，否则NE 有效。

（4）乘法指令 ARM7TDMI(-S)具有32×32 乘法指令、32×32 乘加指令、32×32结果为64 位的乘法指令。表A-5给出全部的ARM 乘法指令。

表A-5 全部的ARM 乘法指令

MUL

```
    32 位乘法指令。指令将Rm 和Rs 中的值相乘，结果的低32 位保存到Rd中。指令格式如下：
    MUL{cond}{S} Rd,Rm,Rs
    MUL 指令举例如下：
    MUL R1,R2,R3 ;R1=R2×R3
    MULS R0,R3,R7 ;R0=R3×R7，同时设置CPSR 中的N位和Z 位
```

MLA

32 位乘加指令。指令将Rm 和Rs 中的值相乘，再将乘积加上第3 个操作数，结果的低32位保存到Rd 中。指令格式如下： `MLA{cond}{S} Rd,Rm,Rs,Rn` MLA 指令举例如下： MLA R1,R2,R3,R0 ;R1=R2×R3+10

UMULL

64 位无符号乘法指令。指令将Rm 和Rs 中的值作无符号数相乘，结果的低32位保存到RsLo 中，而高32 位保存到RdHi 中。指令格式如下： `UMULL{cond}{S} RdLo,RdHi,Rm,Rs` UMULL 指令举例如下： UMULL R0,R1,R5,R8 ;(R1、R0)=R5×R8

UMLAL

64 位无符号乘加指令。指令将Rm 和Rs 中的值作无符号数相乘，64 位乘积与RdHi、RdLo相加，结果的低32 位保存到RdLo 中，而高32 位保存到RdHi 中。指令格式如下： `UMLAL{cond}{S} RdLo,RdHi,Rm,Rs` UMLAL 指令举例如下： UMLAL R0,R1,R5,R8;(R1,R0)=R5×R8+(R1,R0)

SMULL

64 位有符号乘法指令。指令将Rm 和Rs 中的值作有符号数相乘，结果的低32位保存到RdLo 中，而高32 位保存到RdHi 中。指令格式如下： `SMULL{cond}{S} RdLo,RdHi,Rm,Rs` SMULL 指令举例如下： SMULL R2,R3,R7,R6 ;(R3,R2)=R7×R6

SMLAL

64 位有符号乘加指令。指令将Rm 和Rs 中的值作有符号数相乘，64 位乘积与RdHi、RdLo，相加，结果的低32位保存到RdLo 中，而高32 位保存到RdHi 中。指令格式如下： `SMLAL{cond}{S} RdLo,RdHi,Rm,Rs` SMLAL 指令举例如下： SMLAL R2,R3,R7,R6;(R3,R2)=R7×R6+(R3,R2)

1.3.3. ARM 跳转指令

两种方式可以实现程序的跳转：

（1） 使用跳转指令直接跳转，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。

（2） 直接向PC 寄存器赋值实现跳转。

表A-6给出全部的ARM跳转指令。

表A-6 ARM跳转指令

B

跳转指令，跳转到指定的地址执行程序。

`B{cond} label`

举例如下：

`B WAITA` ;跳转到WAITA 标号处

`B 0x1234` ;跳转到绝对地址0x1234 处

跳转到指令B 限制在当前指令的±32Mb 的范围内。

BL

带链接的跳转指令。指令将下一条指令的地址拷贝到R14(即LR)链接寄存器中，然后跳转到指定地址运行程序。

BL{cond} label

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。

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，即把目标地址的代码解释为ARM代码。指令格式如下： BX{cond} Rm 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

BLX

BLX目标地址：跳转，改变状态及保存PC值

1.3.4. ARM 协处理器指令

5 ARM 支持协处理器操作，协处理器的控制要通过协处理器命令实现。表A-7给出全部的ARM协处理器指令。

表A-7 ARM 协处理器指令

CDP

协处理器数据操作指令。ARM 处理器通过CDP 指令通知ARM 协处理器执行特定的操作。该操作由协处理器完成，即对命令的参数的解释与协处理器有关，指令的使用取决于协处理器。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下： `CDP{cond}coproc,opcodel,CRd,CRn,CRm{,opcode2}` 其中： coproc 指令操作的协处理器名。标准名为pn,n 为0～15。 opcodel 协处理器的特定操作码。 CRd 作为目标寄存器的协处理器寄存器。 CRN 存放第1 个操作数的协处理器寄存器。 CRm 存放第2 个操作数的协处理器寄存器。 Opcode2 可选的协处理器特定操作码。 CDP 指令举例如下： CDP p7,0,c0,c2,c3,0 ;协处理器7 操作，操作码为0，可选操作码为0 CDP p6,1,c3,c4,c5 ;协处理器操作，操作码为1

LDC

协处理器数据读取指令。LDC指令从某一连续的内存单元将数据读取到协处理器的寄存器中。协处理器数据的数据的传送，由协处理器来控传送的字数。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下： `LDC{cond}{L} coproc,CRd,<地址>` 其中： L 可选后缀，指明是长整数传送。 coproc 指令操作的协处理器名。标准名为pn，n 为0～15 CRd 作为目标寄存的协处理器寄存器。 <地址> 指定的内存地址 LDC 指令举例如下： `LDC p5,c2,[R2,#4]`; 读取R2+4指向的内存单元的数据，传送到协处理器p5的c2寄存器中 `LDC p6,c2,[R1]` ;读取是指向的内存单元的数据，传送到协处理器p6的c2 寄存器中

STC

协处理器数据写入指令。STC指令将协处理器的寄存器数据写入到某一连续的内存单元中。进行协处理器数据的数据传送，由协处理器来控制传送的字数。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下： `STC{cond}{L} coproc,CRd,<地址>` 其中： L 可选后缀，指明是长整数传送。 coproc 指令操作的协处理器名。标准名为pn，n 为0～15 CRd 作为目标寄存的协处理器寄存器。 <地址> 指定的内存地址 STC 指令举例如下：STC p5,c1,[R0] STC p5,c1,[Ro,#-0x04]

MCR

ARM寄存器到协处理器寄存器的数据传送指令。MCR 指令将ARM 处理器的寄存器中的数据传送到协处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：`MCR{cond}coproc,opcodel,Rd,CRn,CRm{,opcode2}` 其中：coproc 指令操作的协处理器名。标准名为pn，n 为0～15。 cpcodel 协处理器的特定操作码。 RD 作为目标寄存器。 CRn 存放第1 个操作数的协处理器寄存器 CRm 存放第2 个操作数的协处理器寄存器。 Opcode2 可选的协处理器特定操作码。 MCR 指令举例如下： MCR p6,2,R7,c1,c2，MCR P7,0,R1,c3,c2,1,

MRC

协处理器寄存器到ARM寄存器到的数据传送指令。MRC 指令将协处理器寄存器中的数据传送到ARM 处理器的寄存器中。若协处理器不能成功地执行该操作。将产生未定义异常中断。指令格式如下： `MRC{cond}coproc,opcodel,Rd,CRn,CRm{,opcode2}` 其中：coproc 指令操作的协处理器名。标准名为pn，n为0～15。 opcodel 协处理器的特定操作码。 Rd 作为目标寄存器。 CRn 存放第1 个操作数的协处理器寄存器。 CRm 存放第2 个操作数的协处理器寄存器。 opcode2 可选的协处理器特定操作码。 MRC 指令举例如下： MRC p5,2,R2,c3,c2 MRC p7,0,R0,c1,c2,1

1.3.5. ARM 杂项指令

表A-8给出全部的ARM协处理器指令。

表A-8 ARM杂项指令

SWI

软中断指令。SWI 指令用于产生软中断，从而实现在用户模式变换到管理模式，CPSR保存到管理模式的SPSR中，执行转移到SWI 向量，在其它模式下也可使用SWI 指令，处理同样地切换到管理模式。指令格式如下： `SWI{cond} immed_24` 其中：immed_24 24 位立即数，值为0～16777215 之间的整数。 SWI 指令举例如下： SWI 0 ;软中断，中断立即数为0 SWI 0x123456 ;软中断，中断立即数为0x123456 使用SWI 指令时，通常使用以下两种方法进行传递参数，SWI 异常中断处理程序就可以提供相关的服务，这两种方法均是用户软件协定。SWI异常中断处理程序要通过读取引起软中断的SWI 指令，以取得24 位立即数。 （A）指令24 位的立即数指定了用户请求的服务类型，参数通过用寄存器传递。 MOV R0,#34 ;设置了功能号为34 SWI 12 ;调用12 号软中断 （B）指令中的24 位立即数被忽略，用户请求的服务类型由寄存器R0 的值决定，参数通过其它的通用寄存器传递。 MOV R0,#12 ;调用12 号软中断 MOV R1,#34 ;设置子功能号为34 SWI 0 ; 在SWI 异常中断处理程序中，取出SWI 立即数的步骤为：首先确定引起软中断的SWI指令是ARM指令还时Thumb 指令，这可通过对SPSR 访问得到：然后要取得该SWI 指令的地址，这可通过访问LR 寄存器得到：接着读出指令，分解出立即数。 读出SWI 立即数： T_bit EQU 0x20 SWI_Hander STMFD SP!,{R0_R3,R12,LR} ;现场保护 MRS R0,SPSR ;读取SPSR STMFD SP!,{R0} ;保存SPSR TST R0,#T_bit ;测试T标志位 LDRNEH R0,[LR,#-2] ;若是Thumb指令，读取指令码(16 位) BICNE R0,R0,#0xFF00 ;取得Thumb 指令的8 位立即数 LDREQ R0,[LR,#-4] ;若是ARM 指令，读取指令码(32 位) BICNQ R0,R0,#0xFF00000 ;取得ARM 指令的24 位立即数 ... LDMFD SP!,{R0-R3,R12,PC}^ ;SWI 异常中断返回

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 MRS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 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。 使能IRQ 中断例程： ENABLE_IRQ MRS R0,CPSR BIC R0。R0,#0x80 MSR CPSR_c,R0 MOV PC,LR 禁能IRQ 中断例程： DISABLE_IRQ MRS R0,CPSR ORR R0,R0,#0x80 MSR CPSR_c,R0 MOV PC,LR

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` 其中： psr CPSR 或SPSR 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]) immed_8r 要传送到状态寄存器指定域的立即数，8 位。 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，即切换到管理模式。 MSR CPSR_cxsf,R3 ;CPSR=R3 只有在特权模式下才能修改状态寄存器。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)。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。 堆栈指令实始化例程： INITSTACK MOV R0,LR ;保存返回地址 ;设置管理模式堆栈 MSR CPSR_c,#0xD3 LDR SP,StackSvc ;设置中断模式堆栈 MSR CPSR_c,#0xD2 LDR SP,StackIrq …

1.3.6. ARM 伪指令

ARM 伪指令不是ARM 指令集中的指令，只是为了编程方便编译器定义了伪指令，使用时可以像其它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，分别为ADR 伪指令、ADRL 伪指令、LDR 伪指令和NOP 伪指令。

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伪指令被编译器替换成一条合适的指令。通常，编译器用一条ADD 指令或SUB 指令来实现该ADR 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。 ADR 伪指令格式如下： `ADR{cond} register,exper` 其中：register 加载的目标寄存器。 exper 地址表达式。当地址值是非字地齐时，取值范围-255 ~ 255 字节之间；当地址是字对齐时，取值范围-1020 ~ 1020字节之间。 对于基于PC 相对偏移的地址值时，给定范围是相对当前指令地址后两个字处(因为ARM7TDMI为三级流水线)。 ADR 伪指令举例如下： LOOP MOV R1, #0xF0 … ADR R2, LOOP ;将LOOP 的地址放入R2 ADR R3, LOOP+4 可以用ADR 加载地址，实现查表： … ADR R0,DISP_TAB ;加载转换表地址 LDRB R1, [R0,R2] ;使用R2作为参数，进行查表 … DISP_TAB DCB0Xc0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90

ADRL

中等范围的地址读取伪指令。ADRL 指令将基于PC 相对偏移的地址值或基于寄存器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。在汇编编译源程序时，ADRL 伪指令被编译器替换成两个条合适的指令。若不能用两条指令实现ADRL 伪指令功能，则产生错误，编译失败。ADRL伪指令格式如下： `ADR{cond} register,exper` 其中：register 加载的目标寄存器。 expr 地址表达式。当地址值是非字对齐时，取范围-64K ~ 64K 字节之间；当地址值是字对齐时，取值范围-256K ~ 256K字节之间。 ADRL 伪指令举例如下： ADRL R0,DATA_BUF … ADRL R1 DATA_BUF+80 … DATA_BUF SPACE 100 ;定义100 字节缓冲区 可以且用ADRL 加载地址，实现程序跳转，中等范围地址的加载： … ADR LR,RETURNI ;设置返回地址 ADRL R1Thumb_Sub+1 ;取得了

Thumb 子程序入口地址，且R1 的0 位置1 BX R1 ;调用Thumb子程序，并切换处理器状态 RETURNI ... CODE16 Thumb_Sub MOV R1,#10 ...

LDR

大范围的地址读取伪指令。LDR 伪指令用于加载32 位的立即数或一个地址值到指定寄存器。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。若加载的常数未超出MOV 或MVN 的范围，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。LDR 伪指令格式如下： `LDR{cond} register,=expr/label_expr` 其中：register 加载的目标寄存器 expr 32 位立即数。label_expr 基于PC 的地址表达式或外部表达式。 LADR 伪指令举例如下：。 LDR R0,=0x123456 ;加载32 位立即数0x12345678 LDR R0,=DATA_BUF+60 ;加载DATA_BUF 地址+60 ... LTORG ;声明文字池 伪指令LDR 常用于加载芯片外围功能部件的寄存器地址(32 位立即数)，以实现各种控制操作加载32位立即数： ... LDR R0,=IOPIN ;加载GPIO 寄存器IOPIN 的地址 LDR R1,[R0] ;读取IOPIN 寄存器的值 ... LDR R0,=IOSET LDR R1,=0x00500500 STR R1,[R0] ;IOSET=0x00500500 ... 从PC 到文字池的偏移量必须小于4KB。与ARM 指令的LDR 相比，伪指令的LDR的参数有“=”号

NOP

空操作伪指令。NOP 伪指令在汇编时将会被代替成ARM 中的空操作，比如可能为“MOV R0, R0”指令等，NOP 伪指令格式如下： NOP NOP NOP NOP SUBS R1, R1, #1 BNE DELAY1 ...

1.4. 寻址方式

1.4.1. 立即数寻址

立即数前面有“#”号，并且如果是十六进制数则在“#”后添加“0x”或“&”，二进制数“#”后面加“%”。

1.4.2. 寄存器寻址

1.4.3. 寄存器间接寻址

以寄存器中的值作为操作数的地址，而操作数本身放在存储器中。

例如：

```
ADD R0, R1, [R2]
```

1.4.4. 基址变址寻址

将寄存器的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。

例如：`LDR R0, [R1, #4] R0<-[R1+4]`

1.4.5. 多寄存器寻址

一条指令可以完成多个寄存器值得传递，一条指令传送最多16个通用寄存器的值。

`LDMIA R0, {R1, R2, R3, R4}`

1.4.6. 相对寻址

以程序计数器PC的值作为基地址，指令中的地址标号作为偏移量，将两者相加后得到的操作数的有效地址。

例如：`BL NEXT;`

1.4.7. 堆栈寻址

使用一个堆栈指针的专用寄存器指示当前操作位置

递增堆栈：向高地址方向生长

递减堆栈：向低地址方向生长

满堆栈：堆栈指针指向最后压入堆栈的有效数据

空堆栈：堆栈指针指向下一个要放入数据的空位置

2. GNU ARM混编

汇编源程序一般用于系统最基本的初始化：初始化堆栈指针、设置页表、操作 ARM的协处理器等。这些初始化工作完成后就可以跳转到C代码main函数中执行。

1.1. GNU汇编语言语句格式

任何Linux汇编行都是如下结构：**[:][{} @comment**

l instruction为指令

l directive为伪操作

l pseudo-instruction为伪指令

l **:**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。

l comment为语句的注释

下面定义一个"add"的函数，最终返回两个参数的和：

.section.text, “x”

.globaladd @ give the symbol “add” externallinkage

add:

ADD r0, r0, r1 @ add input arguments

MOV pc, lr @ return from subroutine

@ endof program

注意：

l ARM指令，伪指令，伪操作，寄存器名可以全部为大写字母，也可全部为小写字母，但不可大小写混用。

l 如果语句太长，可以将一条语句分几行来书写，在行末用“\”表示换行（即下一行与本行为同一语句）。“\”后不能有任何字符，包含空格和制表符（Tab)。

1.2. GNU汇编程序中的标号symbol（或label）

标号只能由a～z，A～Z，0～9，“.”，_等（由点、字母、数字、下划线等组成，除局部标号外，不能以数字开头）字符组成。

Symbol的本质：代表它所在的地址,因此也可以当作变量或者函数来使用。

l 段内标号的地址值在汇编时确定；

l 段外标号的地址值在连接时确定。

Symbol的分类：3类（依据标号的生成方式）。

<1> 基于PC的标号。基于PC的标号是位于目标指令前的标号或者程序中数据定义伪操作前的标号。这种标号在汇编时将被处理成PC值加上（或减去）一个数字常量，常用于表示跳转指令**“b”**等的目标地址，或者代码段中所嵌入的少量数据。

<2> 基于寄存器的标号。基于寄存器的标号常用MAP和FIELD来定义，也可以用EQU来定义。这种标号在汇编时将被处理成寄存器的值加上（或减去）一个数字常量，常用于访问数据段中的数据。

<3> 绝对地址。绝对地址是一个32位数据。它可以寻址的范围为[0，232-1]即可以直接寻址整个内存空间。

特别说明：局部标号Symbol

局部标号主要在局部范围内使用，而且局部标号可以重复出现。它由两部组成：开头是一个0-99直接的数字，后面紧接一个通常表示该局部变量作用范围的符号。局部变量的作用范围通常为当前段，也可以用**ROUT**来定义局部变量的作用范围。

局部变量定义的语法格式：**N{routname}**

l N：为0~99之间的数字。

l routname：当前局部范围的名称（为符号），通常为该变量作用范围的名称（用ROUT伪操作定义的）。

局部变量引用的语法格式：**%{F|B}{A|T}N{routname}**

l %：表示引用操作

l N：为局部变量的数字号

l routname：为当前作用范围的名称（用ROUT伪操作定义的）

l F：指示编译器只向前搜索

l B：指示编译器只向后搜索

l A：指示编译器搜索宏的所有嵌套层次

l T：指示编译器搜索宏的当前层次

例：使用局部符号的例子，一段循环程序

subs r0, r0, #1 @每次循环使r0=r0-1

bne 1F @跳转到1标号去执行

注意：

l 如果F和B都没有指定，编译器先向前搜索，再向后搜索

l 如果A和T都没有指定，编译器搜索所有从当前层次到宏的最高层次，比当前层次低的层次不再搜索。

l 如果指定了routname，编译器向前搜索最近的ROUT伪操作，若routname与该ROUT伪操作定义的名称不匹配，编译器报告错误，汇编失败。

1.3. GNU汇编程序中的分段

<1> .section伪操作

.section <section_name> {,""}

Startsa new code or data section. Sections in GNU are called .text, a code section, .data, an initializeddata section, and .bss, an uninitialized data section.

Thesesections have default flags, and the linker understands the default names(similardirective to the armasm directive AREA).The following are allowable **.section flags** for ELF format files:

Meaning

a allowable section

w writable section

x executable section

中文解释：

用户可以通过.section伪操作来自定义一个段,格式如下：

.section section_name [,"flags"[, %type[,flag_specific_arguments]]]

每一个段以段名为开始, 以下一个段名或者文件结尾为结束。这些段都有缺省的标志（flags），连接器可以识别这些标志。(与arm asm中的AREA相同)。下面是ELF格式允许的段标志flags：

<标志> 含义

a 允许段

w 可写段

x 执行段

例：定义一个“段”

.section.mysection @自定义数据段，段名为 “.mysection”

.align 2

strtemp:

.ascii "Temp string \n\0" @对这一句的理解，我觉得应该是：将"Temp string \n\0"这个字符串存储在以标号strtemp为起始地址的一段内存空间里

<2> 汇编系统预定义的段名

l .text @代码段

l .data @初始化数据段.data Read-write initialized long data.

l .bss @未初始化数据段

l .sdata @ .sdata Read-write initialized short data.

l .sbss @

注意：源程序中.bss段应该在.text段之前。

1.4. GNU汇编语言定义入口点

汇编程序的缺省入口是**_start**标号，**用户也可以在连接脚本文件中用ENTRY标志指明其它入口点。**

例：定义入口点

.section .data

< initialized data here>

.section .bss

< uninitialized data here>

.section .text

.globl _start

_start:

1.5. GNU汇编程序中的宏定义

格式如下：

.macro 宏名参数名列表 @伪指令.macro定义一个宏

宏体

.endm @.endm表示宏结束

如果宏使用参数,那么在宏体中使用该参数时添加前缀**"\"**。宏定义时的参数还可以使用默认值。可以使用.exitm伪指令来退出宏。

例：宏定义

.macroSHIFTLEFT a, b

.if \b< 0

MOV \a,\a, ASR #-\b

.exitm

.endif

MOV \a,\a, LSL #\b

.endm

1.6. GNU汇编程序中的常数

<1> 十进制数以非0数字开头,如:123和9876；

<2> 二进制数以0b开头,其中字母也可以为大写;

<3> 八进制数以0开始,如:0456,0123;

<4> 十六进制数以0x开头,如:0xabcd,0X123f;

<5> 字符串常量需要用引号括起来,中间也可以使用转义字符,如: “You are welcome!\n”;

<6> 当前地址以“**.**”表示,在**GNU**汇编程序中可以使用这个符号代表当前指令的地址;

<7> 表达式:在汇编程序中的表达式可以使用常数或者数值, “-”表示取负数, “~”表示取补,“<>”表示不相等,其他的符号如:+、-、*、/、%、<、<<、>、>>、|、&、^、!、==、>=、<=、&&、|| 跟C语言中的用法相似。

1.7. GNU ARM汇编的常用伪操作

在前面已经提到过了一些为操作,还有下面一些为操作:

l 数据定义伪操作: .byte,.short,.long,.quad,.float,.string/.asciz/.ascii,重复定义伪操作.rept,赋值语句.equ/.set;

l 函数的定义;

l 对齐方式伪操作 .align;

l 源文件结束伪操作.end;

l .include伪操作;

l if伪操作;

l .global/ .globl 伪操作;

l .type伪操作;

l 列表控制语句;

别于**GNU AS**汇编的通用伪操作,下面是ARM特有的伪操作:

.reg , .unreq , .code , .thumb , .thumb_func , .thumb_set, .ltorg , .pool

<1> 数据定义伪操作

l .byte:单字节定义,如:.byte 1,2,0b01,0x34,072,'s';

l .short:定义双字节数据,如:.short 0x1234,60000;

l .long:定义4字节数据,如:.long 0x12345678,23876565

l .quad:定义8字节,如:.quad 0x1234567890abcd

l .float:定义浮点数,如:.float **0f**-3141592653589793238462643383 27\

95028841971.693993751E-40 @ **- pi**

l .string/.asciz/.ascii:定义多个字符串,如:

.string "abcd","efgh", "hello!"

.asciz "qwer","sun", "world!"

.ascii "welcome\0"

注意：ascii伪操作定义的字符串需要自行添加结尾字符'\0'。

l .rept:重复定义伪操作, 格式如下：

.rept 重复次数

数据定义

.endr @结束重复定义

例：

.rept 3

.byte 0x23

.endr

l .equ/.set: 赋值语句, 格式如下：

.equ(.set)变量名，表达式

例：

.equ abc, 3 @让abc=3

<2> 函数的定义伪操作

l 函数的定义,格式如下：

函数名**:**

函数体

返回语句

一般的,**函数如果需要在其他文件中调用, 需要用到.global伪操作将函数声明为全局函数。**为了不至于在其他程序在调用某个C函数时发生混乱,对寄存器的使用我们需要遵循APCS准则。函数编译器将处理函数代码为一段.global的汇编码。

l 函数的编写应当遵循如下规则：

a. a1-a4寄存器（参数、结果或暂存寄存器，r0到r3 的同义字）以及浮点寄存器f0-f3(如果存在浮点协处理器)在函数中是不必保存的；

b. 如果函数返回一个不大于一个字大小的值，则在函数结束时应该把这个值送到 r0 中；

c. 如果函数返回一个浮点数，则在函数结束时把它放入浮点寄存器f0中；

d. 如果函数的过程改动了sp（堆栈指针，r13）、fp（框架指针，r11）、sl（堆栈限制，r10）、lr（连接寄存器，r14）、v1-v8（变量寄存器，r4 到 r11）和 f4-f7,那么函数结束时这些寄存器应当被恢复为包含在进入函数时它所持有的值。

<3> .align .end .include .incbin伪操作

l .align:用来指定数据的对齐方式,格式如下:

.align [absexpr1, absexpr2]

以某种对齐方式,**在未使用的存储区域填充值**. **第一个值表示对齐方式,4, 8,16或 32.第二个表达式值表示填充的值。**

l .end:表明源文件的结束。

l .include:可以将指定的文件在使用.include 的地方展开,一般是头文件,例如:

.include “myarmasm.h”

l .incbin伪操作可以将原封不动的一个二进制文件编译到当前文件中,使用方法如下:

.incbin"file"[,skip[,count]]

skip表明是从文件开始跳过skip个字节开始读取文件,count是读取的字数.

<4> ..if伪操作

根据一个表达式的值来决定是否要编译下面的代码, 用.endif伪操作来表示条件判断的结束,中间可以使用.else来决定.if的条件不满足的情况下应该编译哪一部分代码。

.if有多个变种:

.ifdefsymbol @判断symbol是否定义

.ifcstring1,string2 @字符串string1和string2是否相等,字符串可以用单引号括起来

.ifeqexpression @判断expression的值是否为0

.ifeqsstring1,string2 @判断string1和string2是否相等,字符串必须用双引号括起来

.ifgeexpression @判断expression的值是否大于等于0

.ifgtabsolute expression @判断expression的值是否大于0

.ifleexpression @判断expression的值是否小于等于0

.ifltabsolute expression @判断expression的值是否小于0

.ifncstring1,string2 @判断string1和string2是否不相等, 其用法跟.ifc恰好相反。

.ifndefsymbol, .ifnotdef symbol @判断是否没有定义symbol, 跟.ifdef恰好相反

.ifneexpression @如果expression的值不是0, 那么编译器将编译下面的代码

.ifnesstring1,string2 @如果字符串string1和string2不相等, 那么编译器将编译下面的代码.

<5> .global .type .title .list

l .global/ .globl： 用来定义一个全局的符号，格式如下:

.global symbol 或者 .globl symbol

l .type：用来指定一个符号的类型是函数类型或者是对象类型, 对象类型一般是数据, 格式如下:

.type 符号, 类型描述

例:

.globla

.data

.align4

.typea, @object

.sizea, 4

a:

.long10

例：

.section.text

.typeasmfunc, @function

.globlasmfunc

asmfunc:

mov pc,lr

<6> 列表控制语句:

.title：用来指定汇编列表的标题,例如:

.title “my program”

.list：用来输出列表文件.

<7> ARM特有的伪操作

l .reg: 用来给寄存器赋予别名,格式如下:

别名 .req 寄存器名

l .unreq: 用来取消一个寄存器的别名,格式如下:

.unreq 寄存器别名

注意被取消的别名必须事先定义过,否则编译器就会报错,这个伪操作也可以用来取消系统预制的别名, 例如r0, 但如果没有必要的话不推荐那样做。

l .code伪操作用来选择ARM或者Thumb指令集,格式如下:

.code 表达式

如果表达式的值为16则表明下面的指令为Thumb指令,如果表达式的值为32则表明下面的指令为ARM指令.

l .thumb伪操作等同于.code 16, 表明使用Thumb指令, 类似的.arm等同于.code 32

l .force_thumb伪操作用来强制目标处理器选择thumb的指令集而不管处理器是否支持

l .thumb_func伪操作用来指明一个函数是thumb指令集的函数

l .thumb_set伪操作的作用类似于.set, 可以用来给一个标志起一个别名, 比.set功能增加的一点是可以把一个标志标记为thumb函数的入口, 这点功能等同于.thumb_func

l .ltorg用于声明一个数据缓冲池(literal pool)的开始,它可以分配很大的空间。

l .pool的作用等同.ltorg

l .space<number_of_bytes> {,<fill_byte>}

分配number_of_bytes字节的数据空间，并填充其值为fill_byte，若未指定该值，缺省填充0。（与armasm中的SPACE功能相同）

l .word {,} ...**插入一个32-bit的数据队列。**（与armasm中的DCD功能相同）**可以使用.word把标识符作为常量使用。**

例：

Start:

valueOfStart:

.word Start

这样程序的开头Start便被存入了内存变量valueOfStart中。

l .hword {,} ...

插入一个16-bit的数据队列。（与armasm中的DCW相同）

1.8. GNU ARM汇编特殊字符和语法

<1> 代码行中的注释符号: '@'

<2> 整行注释符号: '#'

<3> 语句分离符号: ';'

<4> 立即数前缀: '#' 或 '$'

3. ARM GCC 内嵌汇编

对于基于ARM的RISC处理器，GNUC编译器提供了在C代码中内嵌汇编的功能。这种非常酷的特性提供了C代码没有的功能，比如手动优化软件关键部分的代码、使用相关的处理器指令。这里设想了读者是熟练编写ARM汇编程序读者，因为该片文档不是ARM汇编手册。同样也不是C语言手册。这篇文档假设使用的是GCC 4 的版本，但是对于早期的版本也有效。

GCCasm 声明

让我们以一个简单的例子开始。就像C中的声明一样，下面的声明代码可能出现在你的代码中。

/*NOP 例子 */

asm("movr0,r0");

该语句的作用是将r0移动到r0中。换句话讲他并不干任何事。典型的就是NOP指令，作用就是短时的延时。

请接着阅读和学习这篇文档，因为该声明并不像你想象的和其他的C语句一样。内嵌汇编使用汇编指令就像在纯汇编程序中使用的方法一样。可以在一个asm声明中写多个汇编指令。但是为了增加程序的可读性，最好将每一个汇编指令单独放一行。

asm(

"mov r0, r0\n\t"

"mov r0, r0\n\t"

"mov r0, r0\n\t"

"mov r0, r0"

);

换行符和制表符的使用可以使得指令列表看起来变得美观。你第一次看起来可能有点怪异，但是当C编译器编译C语句的是候，它就是按照上面（换行和制表）生成汇编的。到目前为止，汇编指令和你写的纯汇编程序中的代码没什么区别。但是对比其它的C声明，asm的常量和寄存器的处理是不一样的。通用的内嵌汇编模版是这样的。

asm(code : output operand list : input operand list : clobberlist);

汇编和C语句这间的联系是通过上面asm声明中可选的outputoperand list和input operand list。Clobber list后面再讲。

下面是将c语言的一个整型变量传递给汇编，逻辑左移一位后在传递给C语言的另外一个整型变量。

/* Rotating bits example */

asm("mov %[result], %[value], ror #1" :[result] "=r" (y) : [value] "r" (x));

每一个asm语句被冒号（:）分成了四个部分。

汇编指令放在第一部分中的""中间。

"mov %[result], %[value], ror #1"

接下来是冒号后的可选择的output operand list，**每一个条目是由一对[]（方括号）和被他包括的符号名组成，它后面跟着限制性字符串，再后面是圆括号和它括着的C变量。**这个例子中只有一个条目。

[result] "=r" (y)

接着冒号后面是输入操作符列表，它的语法和输入操作列表一样

[value] "r" (x)

破坏符列表，在本例中没有使用

就像上面的NOP例子，asm声明的4个部分中，**只要最尾部没有使用的部分都可以省略。但是有有一点要注意的是，上面的4个部分中只要后面的还要使用，前面的部分没有使用也不能省略，必须空但是保留冒号。**下面的一个例子就是设置ARMSoc的CPSR寄存器，它有input但是没有output operand。

asm("msr cpsr,%[ps]" : : [ps]"r"(status))

即使汇编代码没有使用，代码部分也要保留空字符串。下面的例子**使用了一个特别的破坏符，目的就是告诉编译器内存被修改过**了。这里的破坏符在下面的优化部分在讲解。

asm(""::::"memory");

为了增加代码的可读性，你可以使用换行，空格，还有C风格的注释。

asm("mov %[result], %[value], ror#1"

: [result]"=r" (y) /*Rotation result. */

: [value]"r" (x) /*Rotated value. */

: /* No clobbers */

);

在代码部分%后面跟着的是后面两个部分方括号中的符号，它指的是相同符号操作列表中的一个条目。

%[result]表示第二部分的C变量y，%[value]表示三部分的C变量x；

符号操作符的名字使用了独立的命名空间。这就意味着它使用的是其他的符号表。简单一点就是说你不必关心使用的符号名在C代码中已经使用了。在早期的C代码中，循环移位的例子必须要这么写：

asm("mov %0, %1, ror #1" :"=r" (result) : "r" (value))

在汇编代码中操作数的引用使用的是%后面跟一个数字，%1代表第一个操作数，%2代码第二个操作数，往后的类推。这个方法目前最新的编译器还是支持的。但是它不便于维护代码。试想一下，你写了大量的汇编指令的代码，要是你想插入一个操作数，那么你就不得不从新修改操作数编号。

优化C代码

有两种情况决定了你必须使用汇编。1st，C限制了你更加贴近底层操作硬件，比如，C中没有直接修改程序状态寄存器（PSR）的声明。2nd就是要写出更加优化的代码。毫无疑问GNUC代码优化器做的很好，但是他的结果和我们手工写的汇编代码相差很远。

这一部分有一点很重要，也是被别人忽视最多的就是：我们在C代码中通过内嵌汇编指令添加的汇编代码，也是要被C编译器的优化器处理的。让我们下面做个试验来看看吧。

下面是代码实例。

bigtree@just:~/embedded/basic-C$ arm-linux-gcc -c test.c

bigtree@just:~/embedded/basic-C$ arm-linux-objdump -D test.o

编译器选择r3作为循环移位使用。它也完全可以选择为每一个C变量分配寄存器。Load或者store一个值并不显式的进行。下面是其它编译器的编译结果。

E420A0E1 mov r2, r4, ror #1 @ y, x

编译器为每一个操作数选择一个相应的寄存器，将操作过的值cache到r4中，然后传递该值到r2中。这个过程你能理解不?

有的时候这个过程变得更加糟糕。有时候编译器甚至完全抛弃你嵌入的汇编代码。C编译器的这种行为，取决于代码优化器的策略和嵌入汇编所处的上下文。如果在内嵌汇编语句中不使用任何输出部分，那么C代码优化器很有可能将该内嵌语句完全删除。比如NOP例子，我们可以使用它作为延时操作，但是对于编译器认为这影响了程序的执行速速，认为它是没有任何意义的。

上面的解决方法还是有的。那就是**使用volatile关键字。它的作用就是禁止优化器优化。**将NOP例子修改过后如下：

/* NOP example, revised */

asm volatile("movr0, r0");

下面还有更多的烦恼等着我们。一个设计精细的优化器可能重新排列代码。看下面的代码：

i++;

if (j == 1)

x += 3;

i++;

优化器肯定是要从新组织代码的，两个i++并没有对if的条件产生影响。更进一步的来讲，i的值增加2，仅仅使用一条ARM汇编指令。因而代码要重新组织如下：

if (j == 1)

x += 3;

i += 2;

这样节省了一条ARM指令。结果是：这些操作并没有得到许可。

这些将对你的代码产生很到的影响，这将在下面介绍。下面的代码是c乘b，其中c和b中的一个或者两个可能会被中断处理程序修改。进入该代码前先禁止中断，执行完该代码后再开启中断。

asm volatile("mrs r12,cpsr\n\t"

"orr r12, r12, #0xC0\n\t"

"msr cpsr_c, r12\n\t" ::: "r12", "cc");

c = *b; /* This may fail. */

asm volatile("mrs r12, cpsr\n"

"bic r12, r12, #0xC0\n"

"msr cpsr_c, r12" ::: "r12", "cc");

但是不幸的是针对上面的代码，优化器决定先执行乘法然后执行两个内嵌汇编，或相反。这样将会使得我们的代码变得毫无意义。

我们可以使用clobberlist帮忙。上面例子中的clobber list如下：

"r12","cc"

上面的clobber list将会将向编译器传达如下信息，修改了r12和程序状态寄存器的标志位。Btw，直接指明使用的寄存器，将有可能阻止了最好的优化结果。通常你只要传递一个变量，然后让编译器自己选择适合的寄存器。另外寄存器名，cc（condition registor 状态寄存器标志位），memory都是在clobber list上有效的关键字。它用来向编译器指明，内嵌汇编指令改变了内存中的值。这将强迫编译器在执行汇编代码前存储所有缓存的值，然后在执行完汇编代码后重新加载该值。这将保留程序的执行顺序，因为在使用了带有memory clobber的asm声明后，所有变量的内容都是不可预测的。

asm volatile("mrs r12,cpsr\n\t"

"orr r12, r12, #0xC0\n\t"

"msr cpsr_c, r12\n\t" :: : "r12", "cc","memory");

c = *b; /* This is safe. */

asm volatile("mrs r12, cpsr\n"

"bic r12, r12, #0xC0\n"

"msr cpsr_c, r12" ::: "r12", "cc","memory");

使所有的缓存的值都无效，只是局部最优（suboptimal）。你可以有选择性的添加dummyoperand 来人工添加依赖。

asm volatile("mrs r12,cpsr\n\t"

"orr r12, r12, #0xC0\n\t"

"msr cpsr_c, r12\n\t" : "=X" (b) :: "r12","cc");

c = *b; /* This is safe. */

asm volatile("mrs r12

上面的第一个asm试图修改变量先b，第二个asm试图修改c。这将保留三个语句的执行顺序，而不要使缓存的变量无效。

理解优化器对内嵌汇编的影响很重要。如果你读到这里还是云里雾里，最好是在看下个主题之前再把这段文章读几遍^_^。

Input and output operands

前面我们学到，每一个input和output operand，由被方括号[]中的符号名，限制字符串，圆括号中的C表达式构成。

这些限制性字符串有哪些，为什么我们需要他们？你应该知道每一条汇编指令只接受特定类型的操作数。例如：跳转指令期望的跳转目标地址。不是所有的内存地址都是有效的。因为最后的opcode只接受24位偏移。但矛盾的是跳转指令和数据交换指令都希望寄存器中存储的是32位的目标地址。在所有的例子中，C传给operand的可能是函数指针。所以面对传给内嵌汇编的常量、指针、变量，编译器必须要知道怎样组织到汇编代码中。

对于ARM核的处理器，GCC 4 提供了一下的限制。

Constraint	Usage in ARM state	Usage in Thumb state
f	Floating point registers f0 .. f7	Not available
G	Immediate floating point constant	Not available
H	Same a G, but negated	Not available
I	Immediate value in data processing instructionse.g. ORR R0, R0, #operand	Constant in the range 0 .. 255e.g. SWI operand
J	Indexing constants -4095 .. 4095e.g. LDR R1, [PC, #operand]	Constant in the range -255 .. -1e.g. SUB R0, R0, #operand
K	Same as I, but inverted	Same as I, but shifted
L	Same as I, but negated	Constant in the range -7 .. 7e.g. SUB R0, R1, #operand
l	Same as r	Registers r0..r7e.g. PUSH operand
M	Constant in the range of 0 .. 32 or a power of 2e.g. MOV R2, R1, ROR #operand	Constant that is a multiple of 4 in the range of 0 .. 1020e.g. ADD R0, SP, #operand
m	Any valid memory address	
N	Not available	Constant in the range of 0 .. 31e.g. LSL R0, R1, #operand
o	Not available	Constant that is a multiple of 4 in the range of -508 .. 508e.g. ADD SP, #operand
r	General register r0 .. r15e.g. SUB operand1, operand2, operand3	Not available

Constraint	Usage in ARM state	Usage in Thumb state
W	Vector floating point registers s0 .. s31	Not available
X	Any operand	
=： Write-only operand, usually used for all output operands+： Read-write operand, must be listed as an output operand&： A register that should be used for output only		

Output operands必须为write-only，相应C表达式的值必须是左值。Input operands必须为read-only。C编译器是没有能力做这个检查。

比较严格的规则是：不要试图向input operand写。但是如果你想要使用相同的operand作为input和output。限制性modifier（+）可以达到效果。例子如下：

asm("mov %[value], %[value], ror #1" : [value]"+r" (y))

和上面例子不一样的是，最后的结果存储在input variable中。

可能modifier + 不支持早期的编译器版本。庆幸的是这里提供了其他解决办法，该方法在最新的编译器中依然有效。**对于input operators有可能使用单一的数字n在限制字符串中。使用数字n可以告诉编译器使用的第n个operand，operand都是以0开始计数。**下面是例子：

asm("mov %0, %0, ror #1" : "=r" (value) :"0" (value))

限制性字符串“0”告诉编译器，使用和第一个output operand使用同样input register。

请注意，在相反的情况下不会自动实现。如果我没告诉编译器那样做，编译器也有可能为input和output选择相同的寄存器。第一个例子中就为input和output选择了r3。

在多数情况下这没有什么，但是如果在input使用前output已经被修改过了，这将是致命的。在input和output使用不同寄存器的情况下，你必须使用&modifier来限制outputoperand。下面是代码示例：

asm volatile("ldr %0, [%1]""\n\t"

"str %2, [%1, #4]""\n\t"

: "=&r" (rdv)

: "r"(&table), "r" (wdv)

: "memory");

在以张表中读取一个值然后在写到该表的另一个位置。

其他

内嵌汇编作为预处理宏

要是经常使用使用部分汇编，最好的方法是将它以宏的形式定义在头文件中。使用该头文件在严格的ANSI模式下会出现警告。为了避免该类问题，**可以使用asm代替asm，volatile代替volatile。**这可以等同于别名。下面就是个例程：

```
#define BYTESWAP(val) \
asm volatile ( \
"eor r3, %1, %1, ror #16\n\t" \
"bic r3, r3, #0x00FF0000\n\t" \
"mov %0, %1, ror #8\n\t" \
"eor %0, %0, r3, lsr #8" \
: "=r" (val) \
: "0"(val) \
: "r3", "cc" \
);
```

C 桩函数

宏定义包含的是相同的代码。这在大型routine中是不可以接受的。这种情况下最好定义个桩函数。

```
unsigned long ByteSwap(unsigned longval)
{
asm volatile (
"eor r3, %1, %1, ror #16\n\t"
"bic r3, r3, #0x00FF0000\n\t"
"mov %0, %1, ror #8\n\t"
"eor %0, %0, r3, lsr #8"
: "=r" (val)
: "0"(val)
: "r3"
);
return val;
}
```

替换C变量的符号名

默认的情况下，GCC使用同函数或者变量相同的符号名。你可以使用asm声明，为汇编代码指定一个不同的符号名

```
unsigned long value asm("clock") = 3686400
```

这个声明告诉编译器使用了符号名clock代替了具体的值。

替换C函数的符号名

为了改变函数名，你需要一个原型声明，因为编译器不接受在函数定义中出现asm关键字。

extern long Calc(void) asm ("CALCULATE")

调用函数calc()将会创建调用函数CALCULATE的汇编指令。

强制使用特定的寄存器

局部变量可能存储在一个寄存器中。你可以利用内嵌汇编为该变量指定一个特定的寄存器。

void Count(void) {

register unsigned char counterasm("r3");

... some code...

asm volatile("eor r3, r3,r3");

... more code...

}

汇编指令"eor r3, r3, r3"，会将r3清零。Waring：该例子在到多数情况下是有问题的，因为这将和优化器相冲突。因为GCC不会预留其它寄存器。要是优化器认为该变量在以后一段时间没有使用，那么该寄存器将会被再次使用。但是编译器并没有能力去检查是否和编译器预先定义的寄存器有冲突。如果你用这种方式指定了太多的寄存器，编译器将会在代码生成的时候耗尽寄存器的。

临时使用寄存器

如果你使用了寄存器，而你没有在input或output operand传递，那么你就必须向编译器指明这些。下面的例子中使用r3作为scratch 寄存器，通过在clobber list中写r3，来让编译器得知使用该寄存器。由于ands指令跟新了状态寄存器的标志位，使用cc在clobber list中指明。

asm volatile(

"ands r3, %1, #3" "\n\t"

"eor %0, %0, r3" "\n\t"

"addne %0, #4"

: "=r" (len)

: "0" (len)

: "cc", "r3"

);

最好的方法是使用桩函数并且使用局部临时变量

寄存器的用途

比较好的方法是分析编译后的汇编列表，并且学习C 编译器生成的代码。下面的列表是编译器将ARM核寄存器的典型用途，知道这些将有助于理解代码。

Register	Alt. Name	Usage
r0	a1	First function argumentInteger function resultScratch register
r1	a2	Second function argumentScratch register
r2	a3	Third function argumentScratch register
r3	a4	Fourth function argumentScratch register
r4	v1	Register variable
r5	v2	Register variable
r6	v3	Register variable
r7	v4	Register variable
r8	v5	Register variable=
r9	v6	rfpRegister variableReal frame pointer
r10	sl	Stack limit
r11	fp	Argument pointer
r12	ip	Temporary workspace
r13	sp	Stack pointer
r14	lr	Link register Workspace
r15	pc	Program counter