专题3-玩转汇编语言

一、ARM汇编语言概述

1.1、为什么使用汇编

在嵌入式开发中，汇编程序常用于非常关键的地方，比如系统启动时的初始化，进出中断时的环境保存，回复，对性能要求非常苛刻的函数等。

1.2、ARM汇编分类

1.2.1、ARM标准汇编（windows平台下）

1.2.2、GNU汇编（linux平台下）

1.3、ARM汇编程序框架

1.3.1、完整框架

```
.section .data
   <初始化的数据>
.section .bss
    <未初始化的数据>
.section .text
.global _start
_start:
    <汇编代码>
```

1.3.2、简化框架

```
.text
.global _start
_start:
    <汇编代码>
```

1.4、汇编环境搭建

二、ARM指令分类学习

GNU汇编中汇编指令使用小写；注释格式为：@注释内容；

2.1、算术和逻辑指令

2.1.1、MOV：传送(Move)

```
MOV{条件}{S}  <dest>, <op 1>

        dest = op_1
```

MOV 从另一个寄存器、被移位的寄存器、或一个立即值装载一个值到目的寄存器。你可以指定相同的寄存器来实现 NOP 指令的效果，你还可以专门移位一个寄存器:

```
MOV    R0, R0              ; R0 = R0... NOP 指令

MOV    R0, R0, LSL#3       ; R0 = R0 * 8
```

如果 R15 是目的寄存器，将修改程序计数器或标志。这用于返回到调用代码，方法是把连接寄存器的内容传送到 R15:

```
MOV    PC, R14             ; 退出到调用者

MOVS   PC, R14             ; 退出到调用者并恢复标志位
                           (不遵从 32-bit 体系)
```

2.1.2、MVN：传送取反的值(Move Negative)

```
MVN{条件}{S}  <dest>, <op 1>

        dest = !op_1
```

MVN 从另一个寄存器、被移位的寄存器、或一个立即值装载一个值到目的寄存器。不同之处是在传送之前位被反转了，所以把一个被取

反的值传送到一个寄存器中。这是逻辑非操作而不是算术操作，这个取反的值加 1 才是它的取负的值:

```
MVN     R0, #4              ; R0 = -5

MVN     R0, #0              ; R0 = -1
```

2.1.3、SUB : 减法 (Subtraction)

```
SUB{条件}{S}  <dest>, <op 1>, <op 2>

            dest = op_1 - op_2
```

SUB 用操作数 one 减去操作数 two，把结果放置到目的寄存器中。操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:

```
SUB     R0, R1, R2          ; R0 = R1 - R2
SUB     R0, R1, #256        ; R0 = R1 - 256
SUB     R0, R2, R3,LSL#1    ; R0 = R2 - (R3 << 1)
```

减法可以在有符号和无符号数上进行。

2.1.4、ADD : 加法 (Addition)

```
ADD{条件}{S}  <dest>, <op 1>, <op 2>

            dest = op_1 + op_2
```

ADD 将把两个操作数加起来，把结果放置到目的寄存器中。操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:

```
ADD     R0, R1, R2          ; R0 = R1 + R2
ADD     R0, R1, #256        ; R0 = R1 + 256
ADD     R0, R2, R3,LSL#1    ; R0 = R2 + (R3 << 1)
```

加法可以在有符号和无符号数上进行。

2.1.5、AND : 逻辑与 (logical AND)

```
AND{条件}{S}  <dest>, <op 1>, <op 2>

            dest = op_1 AND op_2
```

AND 将在两个操作数上进行逻辑与，把结果放置到目的寄存器中；对屏蔽你要在上面工作的位很有用。 操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:

```
AND     R0, R0, #3          ; R0 = 保持 R0 的位 0 和 1，丢弃其余的位。
```

2.1.6、BIC : 位清除 (Bit Clear)

```
BIC{条件}{S}  <dest>, <op 1>, <op 2>

            dest = op_1 AND (!op_2)
```

BIC 是在一个字中清除位的一种方法，与 OR 位设置是相反的操作。操作数 2 是一个 32 位位掩码(mask)。如果如果在掩码中设置了某一位，则清除这一位。未设置的掩码位指示此位保持不变。

```
BIC     R0, R0, #%1011      ; 清除 R0 中的位 0、1、和 3。保持其余的不变。
```

2.2、比较指令

2.2.1、CMP : 比较 (Compare)

```
CMP{条件}{P}  <op 1>, <op 2>

            status = op_1 - op_2
```

CMP 允许把一个寄存器的内容和另一个寄存器的内容或立即值进行比较，更改状态标志来允许进行条件执行。它进行一次减法，但不存

储结果，而是正确的更改标志。标志表示的是操作数 1 比操作数 2 如何(大小等)。如果操作数 1 大于操作操作数 2，则此后的有 GT 后缀的指令将可以执行。 明显的，你不需要显式的指定 S 后缀来更改状态标志... 如果你指定了它则被忽略。

2.2.2、TST : 测试位
(Test bits)

```
TST{条件}{P}  <op 1>, <op 2>

        Status = op_1 AND op_2
```

TST 类似于 CMP，不产生放置到目的寄存器中的结果。而是在给出的两个操作数上进行操作并把结果反映到状态标志上。使用 TST 来检查是否设置了特定的位。操作数 1 是要测试的数据字而操作数 2 是一个位掩码。经过测试后，如果匹配则设置 Zero 标志，否则清除它。象 CMP 那样，你不需要指定 S 后缀。

```
TST    R0, #%1              ; 测试在 R0 中是否设置了位 0。
```

2.3、跳转指令

2.3.1、B : 分支
(Branch)

```
B{条件}  <地址>
```

B 是最简单的分支。一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的地址，从那里继续执行。注意存储在分支指令中的实际的值是相对当前的 R15 的值的一个偏移量；而不是一个绝对地址。它的值由汇编器来计算，它是 24 位有符号数，左移两位后有符号扩展为 32 位，表示的有效偏移为 26 位(+/- 32 M)。
在其他处理器上，你可能经常见到这样的指令:

```
 OPT 1
 LDA &70
 CMP #0
 BEQ Zero
 STA &72
.Zero RTS
```

(取自 Acorn Electron User Guide issue 1 page 213)
在 ARM 处理器上，它们将变成下面这些东西:

```
 OPT    1
 ADR    R1, #&70
 LDR    R0, [R1]
 CMP    #0
 BEQ    Zero
 STR    R0, [R1, #2]
.Zero
 MOV    PC, R14
```

这不是一个很好的例子，但你可以构想如何更好的去条件执行而不是分支。另一方面，如果你有大段的代码或者你的代码使用状态标志，那么你可以使用条件执行来实现各类分支: 这样一个单一的简单条件执行指令可以替代在其他处理器中存在的所有这些分支和跳转指令。

```
 OPT    1
 ADR    R1, #&70
 LDR    R0, [R1]
 CMP    R0, #0
 STRNE  R0, [R1, #2]
 MOV    PC, R14
```

2.3.2、BL : 带连接的分支
(Branch with Link)

```
BL{条件}  <地址>
```

BL 是另一个分支指令。就在分支之前，在寄存器 14 中装载上 R15 的内容。你可以重新装载 R14 到 R15 中来返回到在这个分支之后的那个指令，它是子例程的一个基本但强力的实现。它的作用在屏幕装载器 2 (例子 4)中得以很好的展现...

```
  .load_new_format
    BL    switch_screen_mode
    BL    get_screen_info
    BL    load_palette
```

```
.new_loop
  MOV   R1, R5
  BL    read_byte
  CMP   R0, #255
  BLEQ  read_loop
  STRB  R0, [R2, #1]!
```

...在这里我们见到在装载器循环之前调用了三个子例程。接着，一旦满足了条件执行就在循环中调用了 read_byte 子例程。

2.4、移位指令

2.4.1、LSL：逻辑或算术左移
(Logical or Arithmetic Shift Left)

```
Rx, LSL #n   or
Rx, ASL #n   or
Rx, LSL Rn   or
Rx, ASL Rn
```

接受 Rx 的内容并按用‘n’或在寄存器 Rn 中指定的数量向高有效位方向移位。最低有效位用零来填充。除了概念上的第 33 位(就是被移出的最小的那位)之外丢弃移出最左端的高位，如果逻辑类指令中 S 位被设置了，则此位将成为从桶式移位器退出时进位标志的值。
考虑下列:

```
MOV   R1, #12
MOV   R0, R1, LSL#2
```

在退出时，R0 是 48。 这些指令形成的总和是 R0 = #12, LSL#2 等同于 BASIC 的 R0 = 12 << 2

2.4.2、ROR：循环右移
(Rotate Right)

```
Rx, ROR #n   or
Rx, ROR Rn
```

循环右移类似于逻辑右移，但是把从右侧移出去的位放置到左侧，如果逻辑类指令中 S 位被设置了，则同时放置到进位标志中，这就是位的‘循环’。一个移位量为 32 的操作将导致输出与输入完全一致，因为所有位都被移位了 32 个位置，又回到了开始时的位置!

2.5、程序状态字访问指令

2.5.1、MSR

```
msr cpsr, r0      @复制r0到到cpsr中
```

2.5.2、MRS

```
mrs r0, cpsr      @复制cpsr到r0中
```

2.6、内存访问指令

2.6.1、LDR：从内存中读取数据到寄存器

```
LDR{条件}   Rd, <地址>
LDR{条件}B  Rd, <地址>
```

2.6.2、STR：把寄存器的值存储到内存中

```
LDR{条件}   Rd, <地址>
STR{条件}B  Rd, <地址>
```

这些指令装载和存储 Rd 的值从/到指定的地址。如果象后面两个指令那样还指定了‘B’，则只装载或存储一个单一的字节；对于装载，寄存器中高端的三个字节被置零(zeroed)。

地址可以是一个简单的值、或一个偏移量、或者是一个被移位的偏移量。可以还可以把合成的有效地址写回到基址寄存器(去除了对加/减操作的需要)。

三、ARM伪指令

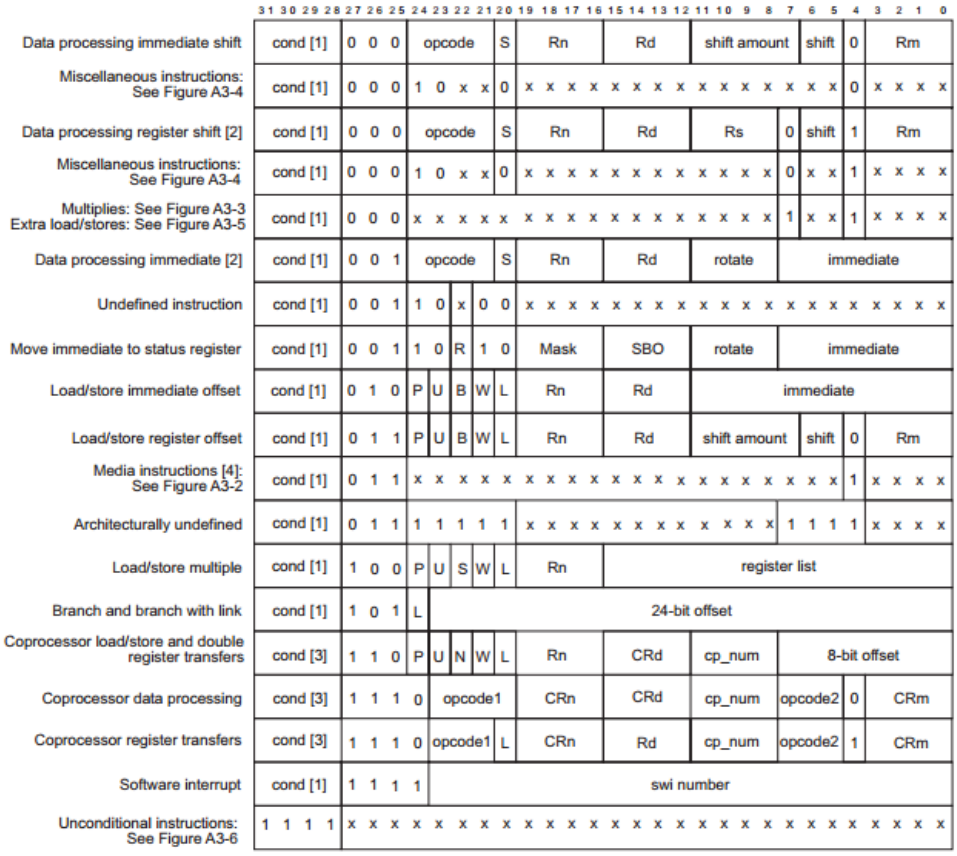
3.1、ARM机器码

A3.1 Instruction set encoding

Figure A3-1 shows the ARM instruction set encoding.

All other bit patterns are UNPREDICTABLE or UNDEFINED. See *Extending the instruction set* on page A3-32 for a description of the cases where instructions are UNDEFINED.

An entry in square brackets, for example [1], indicates that more information is given after the figure.



3.2、伪指令

伪指令不是处理器实际上能理解的指令，但可以转换成它能理解的某种东西。它们的存在能使你的程序更加简单。

3.2.1、定义类伪指令（GNU汇编伪指令前需要加“.”）

global 将程序标号定义为全局的

ascii 定义字符串数据

byte 定义字节数据

int 定义整型数据

word 定义字数据

data 定义数据段

equ 类似于宏定义

```
.equ DA, 0x89
```

align 对齐指针

3.2.2、操作类伪指令

nop 常用于延时（实际是通过 mov r0, r0 来实现）

ldr 将一个立即数赋值给寄存器（在不知道这个数能否用立即数表示时，如果不能用立即数时，它会首先将数据保存在内存中，然后使用内存访问指令来赋值给寄存器）

```
ldr r1, 0x1ff
```

四、协处理器访问指令

4.1、协处理器

协处理器，这是一种协助中央处理器完成其无法执行或执行效率、效果低下的处理工作而开发和应用的处理器。在ARM处理器中，CP15是一个非常重要的协处理器。

4.2、CP15

System control coprocessor
Cache, TCM, and DMA operations are controlled through a dedicated coprocessor, CP15, integrated within the core. This coprocessor provides a standard mechanism for configuring the level one memory system, and also provides functions such as memory barrier instructions. See System control on page 1-21 for more details.

4.3、协处理器访问

You can access CP15 registers with MRC and MCR instructions:
MCR{cond} P15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
MRC{cond} P15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>

建议根据内核手册操作

4.3.1、MCR 把寄存器的值经过操作后赋予CP15

4.3.2、MRC 把CP1的值经过操作后赋予寄存器