1. 嵌入式系统的特点：

1） 专用，软硬件可裁剪，可自由配置

2）低功耗，高可靠性

5）嵌入式系统软件开发需要专门靠性，高稳定性

3）在一些场合对实时性要求高

4）交互性的开发工具和开发环境

（交叉工具链，为什么，以后说）

2.计算机最小系统 && 开机流程 （提问）

嵌入式系统 VS pc机 （麻雀虽小，五脏俱全）

硬件，软件各自的作用，CPU的作用， 总线的作用（类比人的神经中枢系统）

3. 二极管

正向导通（电压越小越好），反向截止，击穿电压

二极管类型（约20种）作用：整流，开关作用。

4. 三极管

PNP与 NPN

C集电极，B基极，E发射极

特性：NPN:电压方向C->E 基极是高电平时CE导通（BE控制CE，VC>VB>VE）

PNP:电压方向 E->C 基极是低电平时CE导通（VC<VB<VE）

截止状态，放大状态，饱和导通状态

作用：电流放大，组成基本的放大电路

注意：在设计电路时需要接上上拉电阻或下拉电阻（作用是？）

上拉电阻就是把不确定的信号通过一个电阻钳位在高电平，此电阻还起到限流作用

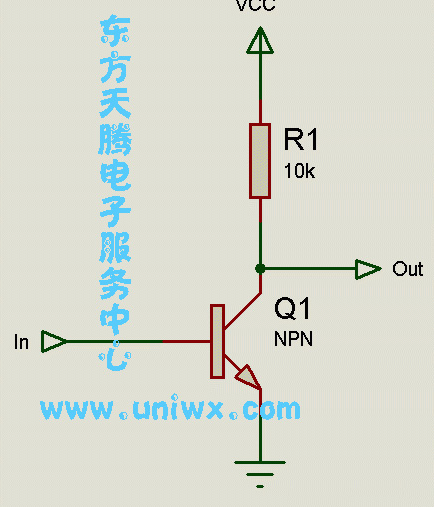
下拉电阻就是把不确定的信号钳位在低电平。

上拉电阻是指器件的输入电流，而下拉指的是输出电流。

作用：（主要几点）

数字电路有三种状态：高天平，低电平和高阻抗状态，有些应用场合不希望出现高阻抗状态，可以通过上拉电阻或下拉电阻的方式使之处于稳定状态。

在CMOS芯片上，为了防止静电造成损坏，不用的管脚不能悬空，一般接上电阻降低输入阻抗，提供泄荷通路；芯片的管脚加上拉电阻来提高输出电平，从而芯片输入信号的噪声容限，增强抗干扰能力；提高总线的抗电磁干扰能力，管脚悬空就比较容易受外界的电磁干扰。



5. 与或非 （位运算）

如何在改变某一位的值时不影响其他位原来的数值。

（这在驱动程序中配置寄存器时非常重要！）

6. 三级存储系统

1）引入三级缓存的原因：速度与成本

软件上可行性的原理：程序运行时局部性

需要解决的问题：速度之间的接力，需要MMU等协处理器

新问题：数据不一致，MMU带来的时间开销，算法换入换出需要耗费CPU时间

2）三级存储结构Cache-主存，主存-辅存之间的相同点：

(1)出发点相同：二者都是为了提高存储系统的性能价格比而构造的分层存储体系，都 力图使存储系统的性能接近高速存储器，而价格和容量接近低速存储器。

(2)原理相同：都是利用了程序运行时的局部性原理把最近常用的信息块从相对慢速而 大容量的存储器调入相对高速而小容量的存储器。

3) cache-主存和主存-辅存这两个存储层次不同之处：

(1)侧重点不同：cache主要解决主存与CPU的速度差异问题

(2)数据通路不同：CPU与cache和主存之间均有直接访问通路，cache不命中时可直 接访问主存；而辅存所依赖的辅存与CPU之间不存在直接的数据通路，当主存不 命中时只能通过调页解决，CPU最终还是要访问主存。

(3)透明性不同：cache的管理完全由硬件完成，对系统程序员和应用程序员均透明； 而虚存管理由软件（操作系统）和硬件共同完成，由于软件的介入，虚存对实现存 储管理的系统程序员不透明，而只对应用程序员透明（段式和段页式管理对应用程 序员“半透明”）。

(4)未命中时的损失不同：由于主存的存取时间是cache的存取时间的5～10倍，而 主存的存取速度通常比辅存的存取速度快上千倍，故主存未命中时系统的性能损失 要远大于cache未命中时的损失。

4） write back && write through

5) 内核主页表更新时必须关闭cache

dma分配内存时要考虑cache

指令cache与数据cache分开的好处？提问

7. IO地址空间（X86），内存地址空间

思考：CPU如何访问硬盘地址？

8. 总线 （类比：城市交通主干道或人的动脉血管）

1）总线（Bus）是计算机各种功能部件之间传送信息的公共通信干线，它是由导线组成的传输线束， 按照计算机所传输的信息种类，计算机的总线可以划分为数据总线、地址总线和控制总线，分别用来传输数据、数据地址和控制信号。总线是一种内部结构，它是cpu、内存、输入、输出设备传递信息的公用通道，主机的各个部件通过总线相连接，外部设备通过相应的接口电路再与总线相连接，从而形成了计算机硬件系统。在计算机系统中，各个部件之间传送信息的公共通路叫总线，微型计算机是以总线结构来连接各个功能部件的。

2）分类

数据总线（Data Bus）：在CPU与RAM之间来回传送需要处理或是需要储存的数据。

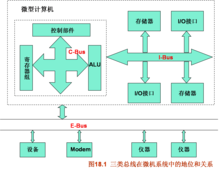
地址总线（Address Bus）：用来指定在RAM（Random Access Memory）之中储存的数据的地址。

控制总线（Control Bus）：将微处理器控制单元（Control Unit）的信号，传送到周边设备

扩展总线（Expansion Bus）：可连接扩展槽和电脑。

局部总线（Local Bus）：取代更高速数据传输的扩展总线。

其中的数据总线DB（Data Bus）、地址总线AB（Address Bus）和控制总线CB（Control Bus），也统称为系统总线，即通常意义上所说的总线。

[](http://baike.baidu.com/pic/æ»çº¿/108823/0/1b0d4f0f0d14b20f6159f330?fr=lemma&ct=single)  
注：ARM 中总线 AMBA APB AHB ASB

3）DMA

数据传输时不需要CPU的参与，但由总线的使用权由CPU控制。

DMA请求，DMA响应，DMA传输，DMA结束（通过中断来通知CPU收回总线控制权）

9. 识别原理图 （韦东山视频有详细介绍）

1） 如何看电路原理图：

同一网络标号的线在电气上是连接在一块的；

通过搜索或者查找目录可以查到对应的器件

2） 搭积木而已！

3）内存类接口（即地址线直接接到CPU中，在CPU中统一寻址）：网卡，nor flash

注意：Nor flash的位宽不同，接到cpu上的地址线不同

协议类接口（CPU不能直接寻址，而是由相应控制器访问）：如nandflash 片选信号，地址线，数据线，命令复用。

10. ARM指令集与thumb指令集

1）ARM 指令集（32-bit）

Thumb 指令集（16-bit）

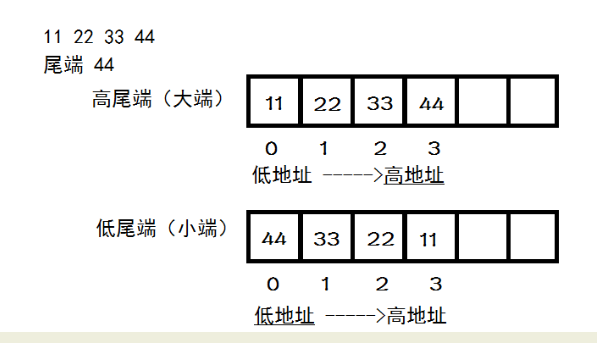
当处理器执行在ARM状态:

所有指令 32 bits 宽

所有指令必须 word 对齐

所以 pc值由bits [31:2]决定, bits [1:0] 未定义

2）大端（高地址放尾端！）对齐，小端对齐 （arm默认小端）



11. ARM中断

1）ARM中有哪几种类型的中断？

FIQ----快速中断，中断响应时间比较短

IRQ---一般中断，响应时间稍长

在应用时，FIQ和IRQ没有太大的区别，在中断初始化的时候，可以设置一个中断源为FIQ或者IRQ。

2） 在ARM的体系结构中，如何保证FIQ快于IRQ？

（1）FIQ的分组寄存器多了R8~R12，这样在中断处理过程中，R8~R12不需要使用stack保存。

（2）FIQ的优先级是高于IRQ中断。

（3）FIQ的向量地址是0x1c，在异常中断向量表的最高位置，这样在0x1c位置可以直接访问FIQ中断的中断服务程序，因此可以比IRQ节省一次跳转。

3）IRQ和FIQ的入口地址？

IRQ---- 0x18

FIQ-----0x1C

4）FIQ和IRQ如何开关？（总开关）

可以通过CPSR寄存器中的I F 位来控制，需要使用汇编语言编程。

中断源->中断控制器->中断总开关（画图说明）

CPSR位介绍一下

5）什么是向量中断和非向量中断？

（1）向量中断：每个中断源对应一个中断入口地址，当该中断有效后，ARM会将该中断对应的入口地址赋给PC，ARM会执行该入口地址上的跳转指令，从而进入了ISR。向量中断优点是响应时间快；当中断源较多时，会占用大量的地址作为向量表（6410是这么干的）。

（2）非向量中断：所有的中断源公用一个向量地址，当一个中断源有效时，ARM将公用向量地址赋给PC，ARM就会进入总的中断服务程序，在总ISR中，通过判断中断控制器中的寄存器，确定中断源，进而进入相应的中断服务程序。

6）中断向量表

（1）向量中断：在每个中断源所对应的向量地址上放置的跳转指令所组成的表格。

（2）非向量中断：非向量中断也可以有一个中断向量表，非向量中断的中断向量表内放置的是中断服务程序的入口地址。（4412是非向量中断）

7）中断响应的过程（S3C2440---非向量中断，IRQ）

注意：CPU是在执行每条指令之前判断有没有中断发生！

（1）进入中断的过程

ARM自动完成的工作：

A、LR\_irq保存返回地址

B、将CPSR保存到SPSR\_irq

C、将ARM的工作模式转换到irq模式（改变 CPSR的模式bits）

D、PC赋值成0x18

软件编程所做的工作：

E、在0x18地址上安装跳转指令（B ISR\_IRQ）

F、创建总的IRQ中断服务程序ISR\_IRQ

F1、将R0~R12，lr保存到这种模式下的stack （保存现场）

F2、读取中断控制器中的中断登记寄存器（INTPND）中的值，确定中断源。

F3、跳转相应中断源的中断服务程序，去执行该中断服务程序。

G、创建具体中断源的中断服务程序（EINT1\_ISR）,在该中断服务程序中，处理该中断，最后要清除中断（通过中断控制器中的相应寄存器（SRCPND、INTPND））。

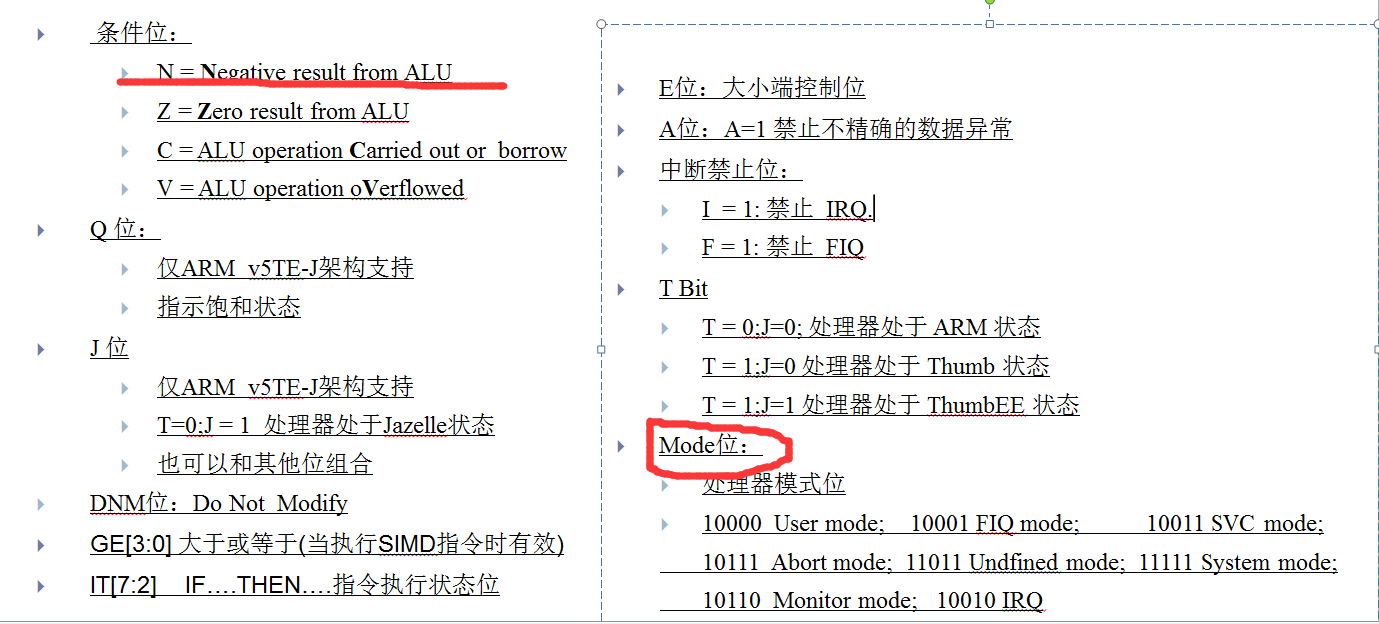
（2）退出中断的过程（必须通过软件编程实现）

A、将R0~R12进行出栈（恢复现场）

B、将LR的值减4赋给PC

C、改变ARM的工作模式（SPSR--->CPSR）

12. CPSR 程序状态寄存器



注意：在ARM状态下，绝大多数的指令都是有条件执行的；在THUMB状态下，仅有分支指令是条件执行的。

13. ARM字节对齐问题 （类比于存放一个大的物品）

1）现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问都可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，这就是对齐。

2）字节对齐的原因大致是如下两条：

（1）平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

（2）性能原因：数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

3）在ARM中，有ARM和Thumb两种指令。

ARM指令：每执行一条指令，PC的值加4个字节（32bits）。一次访问4字节内容，该字节的起始地址必须是4字节对齐的位置上，即地址的低两位为bits[0b00],也就是说地址必须是4的倍数。

Thumb指令：每执行一条指令，PC的值加2个字节（16bits）。一次访问2字节内容，该字节的起始地址必须是2字节对齐的位置上，即地址的低两位为bits[0b0],也就是说地址必须是2的倍数。

遵循以上方式叫对齐（aligned）方式，不遵守这样方式称为非对齐（unaligned）的存储访问操作。

假如，第一次取ARM指令1的地址为 0x0000 0000，由于ARM指令占32位（4个字节），因此地址0x0000 0001、0x0000 0002、0x0000 0003都是指令1的地盘。那么第二次取ARM指令2的地址为 0x0000 0004，同样的道理，0x0000 0005、0x0000 0006、0x0000 0007也都是指令2的地盘，以此类推：

指令1： 0x0000 0000 ——0x0000 0003

指令2： 0x0000 0004 ——0x0000 0007

指令3： 0x0000 0008 ——0x0000 000f

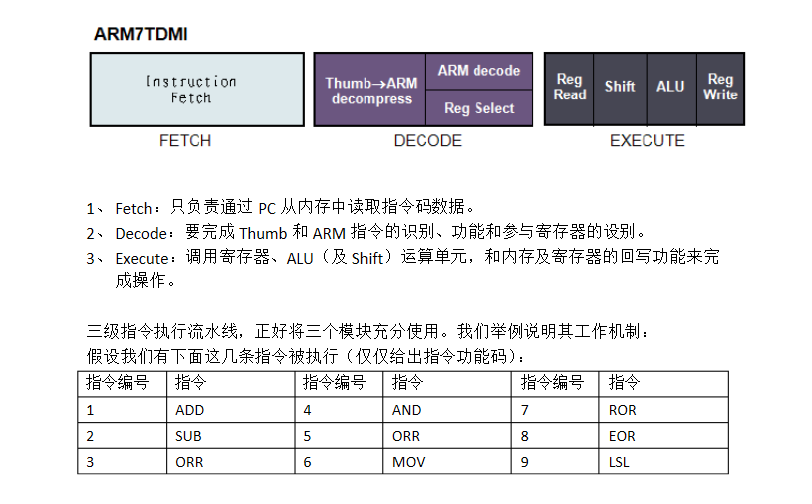
指令4： 0x0000 0010 ——0x0000 0013

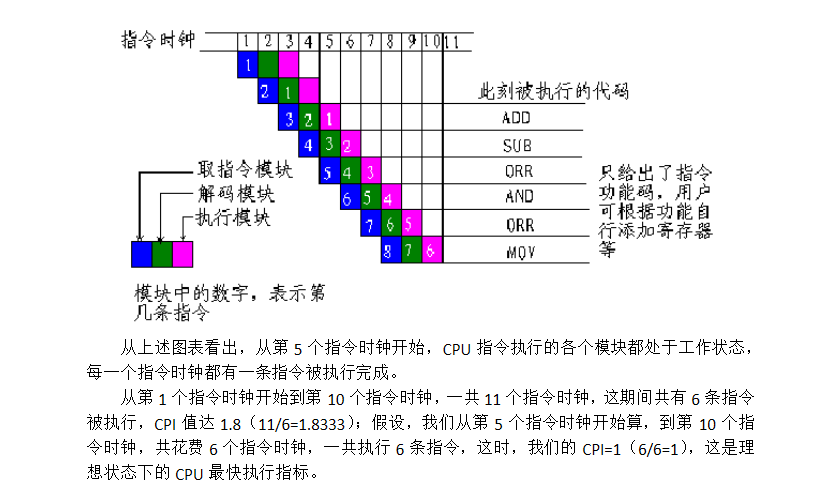
（理解这个就了解了flash不同位宽地址线不同的接法）

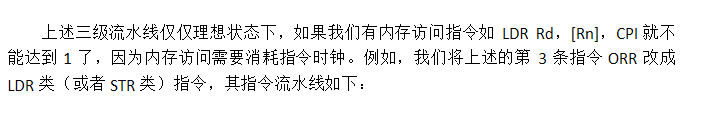
14. ARM 流水线

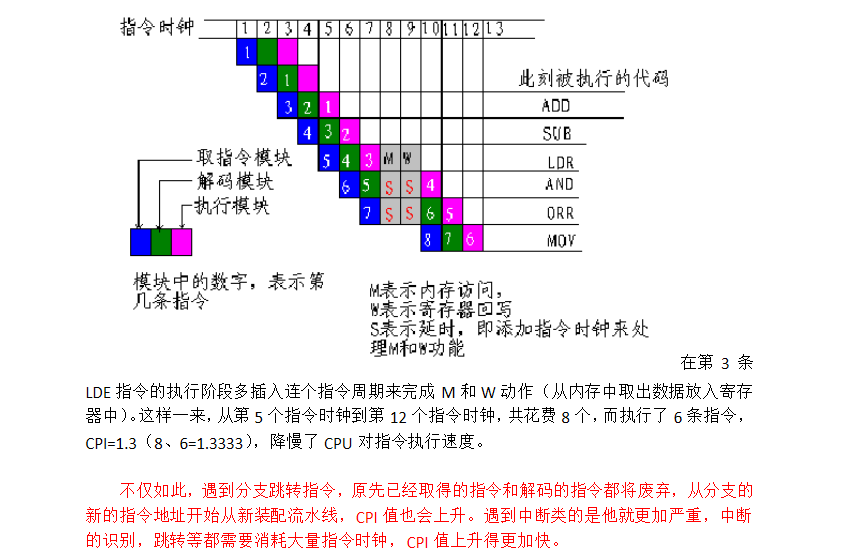
无论处理器处于何种状态，程序计数器R15(PC)总是指向“正在取指”的指令，而不是指向“正在执行”的指令或者正在“译码”的指令。所以，PC总是指向第3条指令，换句话说就是PC总是指向当前正在执行的指令地址再加2条指令的地址。

1）ARM公司的ARM7核使用三级流水线





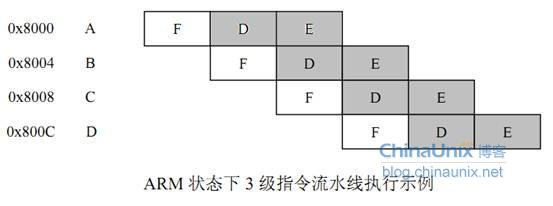




######说明：互锁：一条指令的结果被用作下一条指令的操作数。

2）异常和中断的返回地址问题

不同的异常模式返回地址也是存在差异的，这主要是因为各种异常产生的机理存在差别所导致的。在前面曾提到进入异常时处理器会有一个保存LR 的动作，但是该保存值并不一定是正确的返回地址，下面以一个简单的指令执行流水状态图来对此加以说明。在ARM 架构里，PC值指向当前执行指令的地址加8处，也就是说， 当执行指令A（地址0x8000）时，PC 等于指令C 的地址（0x8008）。假如指令A 是“BL”指令，则当执行该指令时，会把PC（=0x8008）保存到LR 寄存器里面，但是接下去处理器会马上对LR 进行一个自动的调整动作：LR=LR-0x4。这样，最终保存在 LR 里面的是 B 指令的地址，所以当从 BL 返回时，LR 里面正好是正确的返回地址。同样的调整机制在所有LR自动保存操作中都存在，比如进入中断响应时，处理器所做的LR 保存中，也进行了一次自动调整，并且调整动作都是LR=LR-0x4。

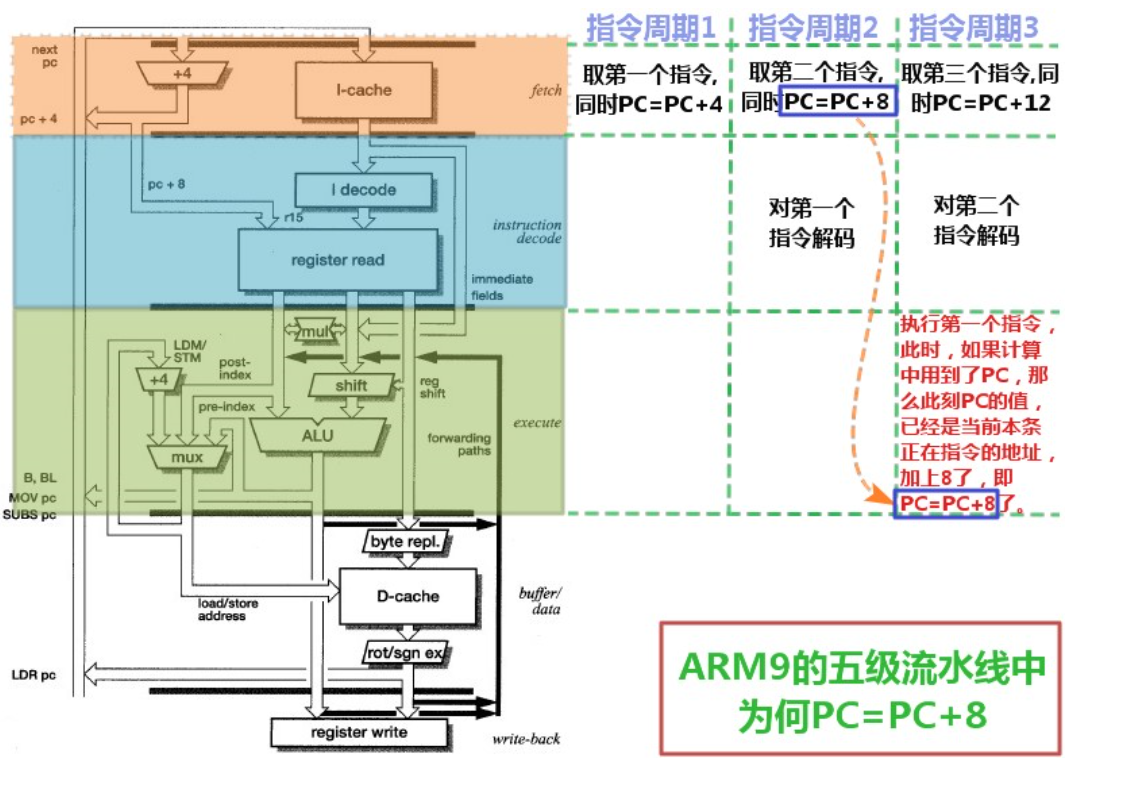
[](http://blog.chinaunix.net/attachment/201205/23/20937170_13377837622XX6.jpg)

“SWI”指令：MOVS pc, lr

发生的是Undefined instruction异常：MOVS pc, lr

发生的是IRQ或FIQ中断：SUBS pc, lr, #4

######说明：因为指令不可能被中断打断，所以A指令执行完以后（CPU在每条指令执行前判断有无中断源产生）才能响应中断，此时PC已更新，指向指令D的地址（地址0x800C），LR 上经过调整保存的地址值是C 的地址0x8008。中断返回后应该执行B指令，所以返回操作是：SUBS pc, lr, #4。



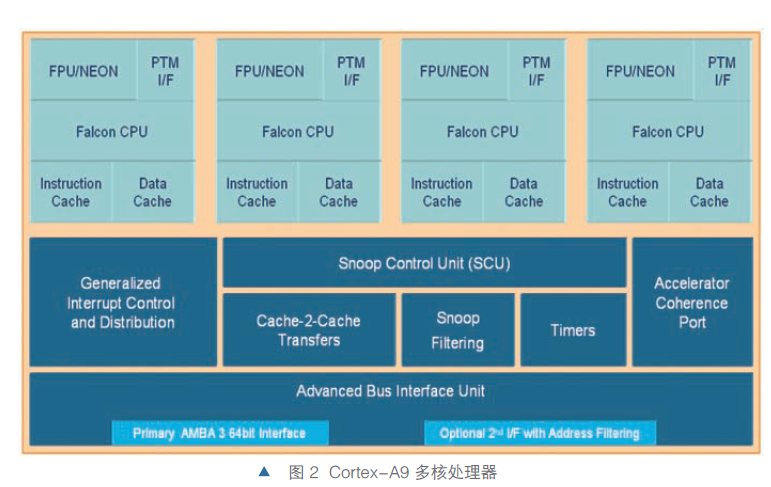
发生的是Prefetch Abort异常：SUBS pc, lr, #4

发生的是“Data Abort”： SUBS pc, lr, #8

3）可重入中断（即中断嵌套）（类比洗碗与打水）

发生中断嵌套时，在中断处理程序中使用BL必须特别小心：如果在第二个中断产生时，BL调用的返回地址(LR\_irq)可能被冲掉，子程序将错误返回，导致无限循环。可以通过下面办法解决该问题：在使用BL之前改变模式来避免LR\_irq被冲掉，通常使用“system模式”（这时BL使用LR\_usr）。（保存软件环境：压栈，保存硬件环境（中断控制器）：需要硬件支持）

15. cortex A9



协处理器CP15： 管理cache 和 MMU。

侦测控制单元 (SCU)：是ARM多核技术的中央情报局，负责为支持MPCore技术的处理器提供互联、仲裁、通信、缓存间及系统内存传输、缓存一致性及其他多核功能的管理。

16. AMBA APB与AMBA AHB

AMBA总线是由ARM公司提出的一种开放性的片上总线标准，它独立于处理器和工艺技术，具有高速度低功耗等特点，受到集成电路设计行业的广泛欢迎。

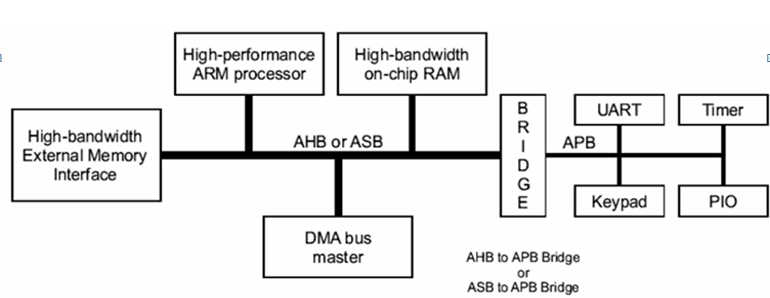
AMBA总线体系定义了三种总线：

（1）AHB：Advanced High-performance Bus，系统总线（AHB）负责连接如嵌入式处理器、DMA控制器、片上存储器和其他外设接口，或者其他需求高带宽的元件。

（2）ASB：Advanced System Bus

（3）APB：Advanced Peripheral Bus，外设总线（APB）则用来连接系统周边的外部设备，其协议相对AHB较为简单。

AHB与APB之间通过桥接器（Bridge）互联。



有高速外设，低速外设，SOC内部核 三大部分的概念

17. 哈佛结构优势

哈佛结构是一种将程序指令存储和数据存储分开的存储器结构。哈佛结构是一种并行体系结构，它的主要特点是将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问。与两个存储器相对应的是系统的4条总线：程序和数据的数据总线与地址总线。这种分离的程序总线和数据总线可允许在一个机器周期内同时获得指令字（来自程序存储器）和操作数（来自数据存储器），从而提高了执行速度，提高了数据的吞吐率。又由于程序和数据存储在两个分开的物理空间中，因此取址和执行能完全重叠。中央处理器首先到程序指令存储器中读取程序指令内容，解码后得到数据地址，再到相应的数据存储器中读取数据，并进行下一步的操作（通常是执行）。程序指令存储和数据存储分开，可以使指令和数据有不同的数据宽度。

18. ARM汇编指令集概述

1）汇编文件中主要包括三部分内容：

（1）指令：编译完成后作为一条指令存储在内存单元当中，CPU执行时能完成一定的操作

（2）伪指令：在编译时替换成其他ARM能够识别的指令

（3）伪操作：指导编译器进行编译，编译完成后不生成指令也不占用内存空间

2）指令分为六类：

（1）数据处理指令：对数据进行逻辑、数学等运算与处理

（2）跳转指令：实现程序的跳转

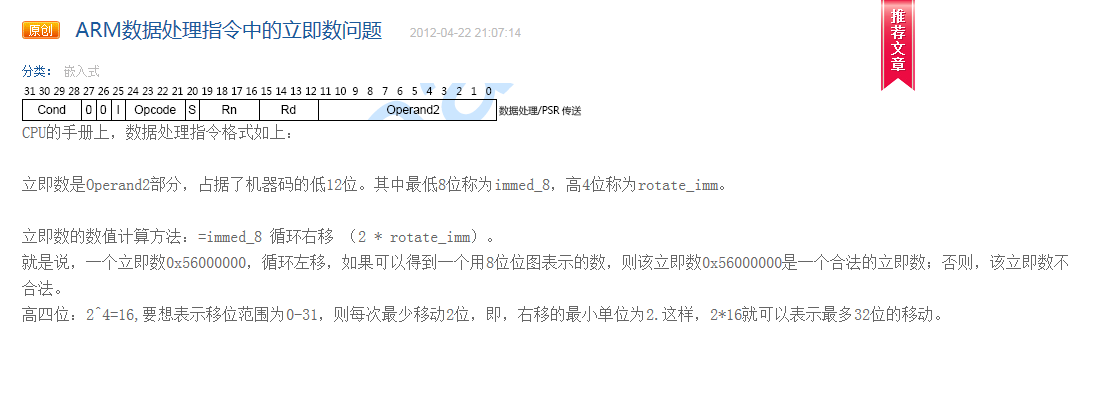
（3）Load/Store指令：CPU与内存之间进行数据的存取

（4）状态寄存器传送指令：对状态寄存器进行读写操作

（5）协处理器指令：对协处理器进行操作

（6）异常中断产生指令：产生异常中断

19. 立即数



20. 数据操作指令：

上课时首先介绍各个寄存器的作用

细分为：数据传送指令，算数指令，逻辑指令，比较与测试指令，乘法指令

#######说明：在数据处理指令前使用**S前缀，指令的执行结果将会影响CPSR中的标志位**

#########数据传送类

1） MOV

举例：

MOV R0，R0 ;NOP

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

MOV PC,R14 ;退出到调用者，用于普通函数返回，PC即R15

MOVS PC,R14 ; 退出到调用者并恢复标志位，用于异常函数返回

作用：

a) 将数据从一个寄存器传到另一个寄存器

b）将一个常数传送到寄存器中

c）实现无算术和逻辑运算的单纯移位操作，如左移

d）当PC用作目标寄存器Rd时，可以实现程序跳转，如：MOV PC,LR。这种指令可以实现子程序调用及从子程序返回，代替B或BL指令

e）当PC作为目标寄存器Rd且指令中S位被设置时，则指令在执行跳转操作的同时，将当前处理器模式的SPSR寄存器的内容复制到CPSR中。指令 MOVS PC,LR可以实现从某些异常中断中返回。

2） MVN 传递相反数

MVN R0，#4;R0=-5

3）AND 逻辑与

AND R0,R0,#3

AND R2,R1,R3

ANDS R0,R0,#0x01

4）EOR 按位异或

EOR R0,R0,#3 ;反转R0中的位0和位1

EOR R1,R1,#0x0F ;将R1中的低4位取反

EOR R2,R1,R0

EORS R0,R5,#0x01;将R5和0x01进行逻辑异或，结果保存到R0，并根据执行结果设置标志位

5） SUB

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

6）RSB 反转减，a-b -> b-a

7）ADC 带进位的加法

SBC 带进位的减法

RSC 带进位的反转减法

RSBS R2,R0,#0

RSC R3,R1,#0 ;求64位数值的负数

8）四条指令均不需要显式设置S标志

TST 测试，不产生放置到目标寄存器中的结果（按位比较，逻辑）

TEQ 测试相等（逻辑），两个数做EOR

CMP 比较，进行一次减法，但不存储结果。（算术）

CMN 负数比较 CMN R0,#1 ;使R0值加1，判断R0是否为1的补码，若是，则z置位（算术）。

9）ORR 逻辑或

下面两条指令的结果是将R2的高8位数据移入到R3的低8位中

MOV R1,R2,LSR #8

ORR R3,R1,R3,LSL #8

10） BIC位清零

BIC R0,R0,#0x1011 ;清除R0中位0，1，3，保持其余位不变

11）移位：逻辑移位与算术移位

举例：

（1）算术左移

mov r0, #0xff

mov r1, r0, lsl #4

将r0逻辑左移四位放入r1中

（2）逻辑左移 高位移出 低位补零

mov r1, r0, lsr #4

（3）逻辑右移 低位移出 高位补零

mov r1, r0, asr #4

（4）算术右移 低位移出 高位补符号位

mov r1, r0, ror r2

（5）循环右移 低位移出 高位补低位

#########乘法指令

#########Load/Store指令

1）单寄存器的～指令

（1）LDR：从内存中将一个32位的字读取到目标寄存器

举例：

LDR R1,[R0,#0X12];将R0+12地址处的数据读出，保存到R1中，R0的值不变

LDR R1,[R0,R2] ;将R0+R2地址处的数据读出，保存到R1中

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

LDR Rd, label ; label必须是当前指令的-4~+4KB范围内

LDR Rd,[Rn],#0x04 ;Rn的值用作数据的存储地址，在数据传送后，将偏移量0x04与Rn相加，结果写回到Rn中。Rd不允许是R15

（2）STR：用于将一个32位的字数据写入到指令中指定的内存单元

举例：

NUMCOUNT EQU 0x40003000

LDR R0,=NUMCOUNT; 伪指令

LDR R1,[R0]

ADD R1,R1,#1

STR R1,[R0]

举例：

COUNT EQU       0x56000054 ; COUNT是我们定义的一个变量，地址为0x56000054。

LDR       R1,=COUNT ; 是将COUNT这个变量的地址，也就是0x56000054放到R1中

MOV       R0,#0; 是将立即数0放到R0中

STR       R0,[R1] ; 完成对变量COUNT赋值

举例：

LDR       R1,=COUNT   ；这条伪指令，是将COUNT的地址赋给R1  
LDR       R0,[R1]   ；将COUNT的值赋给R0

举例：

ldr r0, 0x12345678 ;就是把0x12345678这个地址中的值存放到r0中。

;而mov不能干这个活，mov只能在寄存器之间移动数据，或者把立即数移动到寄存器中

举例：

ldr r0, =0x12345678   
这样，就把0x12345678这个地址写到r0中了。所以，ldr伪指令和mov是比较相似的。只不过mov指令限制了立即数的长度为8位，也就是不能超过512。而ldr伪指令没有这个限制。如果使用ldr伪指令时，后面跟的立即数没有超过8位，那么在实际汇编的时候该ldr伪指令是被转换为 mov指令的。

举例

ldr r0, \_start ;把 \_start地址存放的值读出来；

adr r0, \_start ;把 \_start地址读出来，而且这个地址是相对当前pc的，所以和当前程序运行地 址相关;

ldr, r0, =\_start ; 取得的是\_start的绝对地址，不管此时正在哪里运行。

2）多寄存器的～内存访问指令：实现在一组寄存器和一块连续的内存单元之间的传送数据，主要用于现场保护，数据复制和参数传递等

LDM{<cond>}<addr\_mode><Rn>{!},<regs>{^}

STM{<cond>}<addr\_mode><Rn>{!},<regs>{^}

（1） 模式：

数据块： IA(i++) IB(++i) DA(i--) DB(--i) （注：偏移的单位都是4）

堆栈： FD ED FA EA （讲解一下堆栈的生长方向）

（2） 寄存器组，不包含R15，之间用，号分隔，如{R1,R2,R6~R9},由小到大排列

（3） ^

^后缀不允许在用户模式下使用，只能在系统模式下使用。若LDM指令在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，还将SPSR复制到CPSR中，可用于异常处理返回；使用^进行数据传送且寄存器列表不包含PC时，加载/存储的是用户模式寄存器，而不是当前模式寄存器。

（4）！更新

举例：

LDMIA R0!,{R3~R9} ,R0更新

STMIA R1!,{R3~R9} ,R1更新

STMFD SP!,{R0~R7,LR} ,保存现场，将寄存器组入栈

LDMFD SP!,{R0~R7,PC}^ ,恢复现场，异常处理返回

SENDBYTE

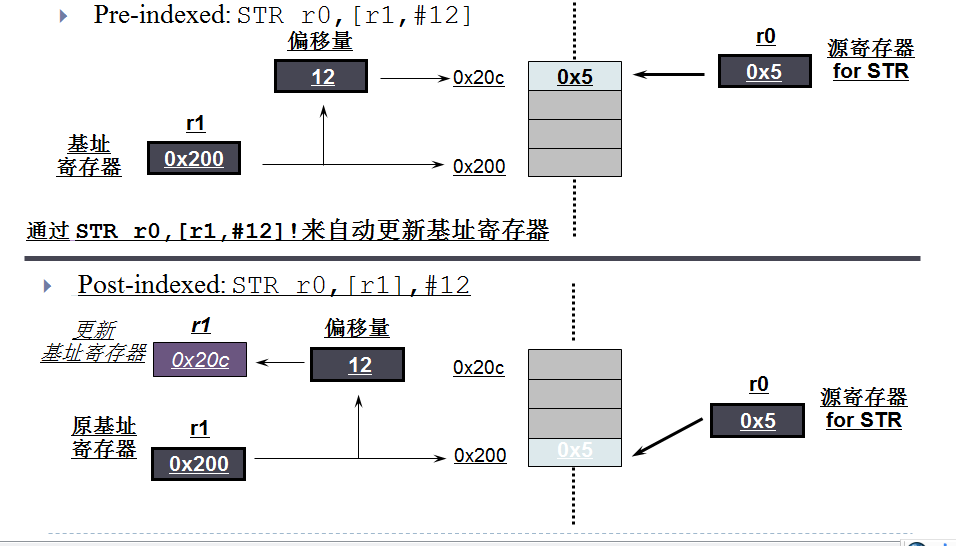
STMFD SP!,{R0~R7,LR}

……

BL DELAY

……

LDMFD SP!, {R0~R7,PC} ,恢复寄存器，并返回（普通程序）



3） SWP 单数据交换指令

作用：方便的进行信号量操作

#########跳转指令

B BL BX BLX

1）说明：在编写位置无关码时，用B，BL，ADR，因为这三条指令是相对PC取指。在编写位置无关码时在程序中不能用LDR=，不能用全局变量。

2）子程序返回有哪些？ （3种方法）

（1）BX r14

（2）MOV PC,R14

（3）STMFD R13!,{<REGS>,R14}

LDMFD R13!,{<REGS>,R14}

#########PSR操作指令

MRS MSR

举例：操作的是位域

MSR CPSR\_c,#0Xd3 ;cpsr[7:0]=0xd3 切换到管理模式

MSR CPSR\_cxsf, R3 ;cpsr=r3

##########协处理器指令

CDP LDC MCR MRC STC

##########异常产生指令

SWI BKPT

21. 伪指令

所有伪指令均以.开始，余下的是字母，通常使用小写字母，伪指令按照不同功能可以分为符号定义伪指令，数据定义伪指令，汇编控制伪指令，杂项

1） 符号定义伪指令

.global:使得符号对链接器ld可见，变为整个工程可用的全局变量

.local

.set 用于给一个全局变量或局部变量赋值

.equ 宏替换

2） 数据定义伪指令

.byte .short .word .long .quard:分配8字节 .float .space: 用于分配一片连续的存储区域并初始化位指定的值 .skip(等价.space) .string .rept:重复执行后面的指令，以此开始，并以.endr结束。

3） 汇编控制伪操作

.If .else .endif

.macro .endm .exitm

4） 杂项

.align .section .data .text .include .arm .code32 .code16 .thumb

.extern .weak .end等

5） GNU 汇编器支持的ARM伪指令

（1） ADR ADRL 相对跳转 ，用于写位置无关码

（2） LDR伪指令 LDR reg, =expr 绝对跳转，如从flash跳到sdram中运行

22. ARM汇编语言的程序结构

1） 汇编语言的程序格式

在ARM汇编语言程序中可以使用.section来进行分段，其中每一段用段名或者文件结尾位结束，a 允许段， w 可写段，x 执行段。

在一个段中，可以定义下列子段：

.text 正文段，包含指令代码

.data 数据段，包含固定的数据，如常量，字符串

.bss 未初始化数据段，bin文件中不存储该段，需要在程序中清零

.sdata 小段，可以不用关注

.sbss

2）GCC 编译程序的过程：预处理，汇编，编译，链接（通过符号，global）

链接地址：程序运行时应该位于的地方。

代码重定位

Makefile

Lds

#######说明：如何书写lds链接脚本，参考uboot

3） 汇编语言的子程序调用

B BL

23. AAPCS

1）了解各种标准名称

ABI : Application Binary Interface 应用程序二进制接口

EABI : Embedded ABI 嵌入式ABI

PCS : Procedure Call Standard 程序调用标准

AAPCS : PCS for ARM Architecture ARM体系结构程序调用标准

ATPCS : ARM-Thumb Procedure Call Standard

PIC, PID : Position-independent code, position-independent data 位置独立代码和数据

2）寄存器使用规则：寄存器的使用规则，堆栈使用规则，参数传递规则

（1）寄存器的使用规则

子程序间通过寄存器R0～R3来传递参数。这时，寄存器R0～R3可记作a0～a3。被调用的子程序在返回前无需恢复寄存器R0～R3的内容。

在子程序中，使用寄存器R4～R11来保存局部变量。这时，寄存器R4～R11可以记作v1～v8。如果在子程序中使用了寄存器v1～v8中的某些寄存器，则子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值。在Thumb程序中，通常只能使用寄存器R4～R7来保存局部变量。

寄存器R12用作过程调用中间临时寄存器，记作IP。在子程序之间的连接代码段中常常有这种使用规则。

寄存器R13用作堆栈指针，记作SP。在子程序中寄存器R13不能用作其他用途。寄存器SP在进入子程序时的值和退出子程序时的值必须相等。

寄存器R14称为连接寄存器，记作LR。它用于保存子程序的返回地址。如果在子程序中保存了返回地址，寄存器R14则可以用作其他用途。

寄存器R15是程序计数器，记作PC。它不能用作其它用途。

（2）堆栈使用规则

ATPCS规定堆栈为FD类型，即满递减堆栈，并且对堆栈的操作是8字节对齐。

对于汇编程序来说，如果目标文件中包含了外部调用，则必须满足下列条件：

（a）外部接口的堆栈必须是8字节对齐的。

（b）在汇编程序中使用PRESERVE8伪指令告诉连接器，本汇编程序数据是8字节对齐的。

（3）参数传递规则

根据参数个数是否固定，可以将子程序分为参数个数固定的子程序和参数个数可变化的子程序。这两种子程序的参数传递规则是不一样的。

（4）参数个数可变子程序参数传递规则

对于参数个数可变的子程序，当参数个数不超过4个时，可以使用寄存器R0～R3来传递参数；当参数超过4个时，还可以使用堆栈来传递参数。

在传递参数时，将所有参数看作是存放在连续的内存字单元的字数据。然后，依次将各字数据传递到寄存器R0，R1，R2和R3中。如果参数多于4个，则将剩余的字数据传递到堆栈中。入栈的顺序与参数传递顺序相反，即最后一个字数据先入栈。

3）子程序结果返回规则

结果为一个32位整数时，可以通过寄存器R0返回；

结果为一个64位整数时，可以通过寄存器R0和Rl返回；

结果为一个浮点数时，可以通过浮点运算部件的寄存器f0、d0或s0来返回；

结果为复合型浮点数（如复数）时，可以通过寄存器f0～fn或d0～dn来返回；

对于位数更多的结果，需要通过内存来传递。

4）ARM汇编（子）程序的相互调用

如果一个ARM汇编语言程序文件含有调用外部汇编语言程序文件中子程序（函数）的指令，则需要用IMPORT指示符来指明将要调用的子程序名称。

如果本汇编语言程序文件中的某个子程序（函数），需要被外部的ARM汇编语言程序文件中的语句调用，则需要用EXPORT指示符来指明将要被调用的子程序（函数）名称。

被执行的汇编子程序在运行前，要注意将寄存器组压入栈区，返回时要注意将栈区保存的工作现场恢复到处理器的寄存器组。

24. C和汇编的混合编程

1） GNU内联汇编

有3种情况必须使用内联汇编或嵌入型汇编

程序中使用饱和算术运算

程序中需要对协处理器进行操作

在C中完成对CPSR或SPSR的操作

2）从C中调用汇编语言

3）汇编语言调用C程序 // 需要遵守ATPCS规则



4）内嵌汇编程序中使用物理寄存器有以下限制。

（1）不能直接向PC寄存器赋值，程序跳转只能使用B或BL指令实现。

（2）不要使用过于复杂的C表达式，因为将会需要较多的物理寄存器，这将导致与其他指令中用到的物理寄存器产生使用冲突。

#####说明：编译器可能会使用R12或R13存放编译的中间结果，在计算表达式的值时可能会将寄存器R0～R3，R12和R14用于子程序调用。因此在内嵌的汇（3）编指令中不要将这些寄存器同时指定为指令中的物理寄存器。因为这可能会影响编译器分配寄存器，进而影响代码的效率。

（4）其他内嵌汇编程序的编写注意点:

（a）常量：在内嵌汇编指令中，常量前面的“#”可以省略。

（b）指令展开：内嵌汇编指令中，如果包含常量操作数，该指令可能被内嵌汇编器展开成几条指令。

（c）标号：C程序中的标号可以被内嵌的汇编指令使用，但是只有指令B可以使用C程序中的标号，而指令BL则不能使用。

（d）内存单元的分配：所有的内存分配均由C编译器完成，分配的内存单元通过变量供内嵌汇编器使用。内嵌汇编器不支持内嵌程序中用于内存分配的伪指令。

（5）SWI和BL指令：在两个指令使用到内嵌汇编中，除了正常的操作数域外，还必须增加以下3个可选的寄存器列表：

用于输入参数的寄存器列表。

用于存储返回结果的寄存器列表。

用于表示那些寄存器将有可能会被修改的寄存器列表。

举例

例1：内联汇编

#include<stdio.h>

void str\_cpy(const char \*src,char \*dst)

{

int ch;

\_\_asm

{

loop: //普通ARM汇编代码中的标号后面不能跟冒号。C程序中

//的标号可以被内嵌的汇编指令使用。 ARM内嵌汇编代码中

//只有B指令可以使用C的标号，而BL指令不能够使用C代码

//的标号。 C程序的标号后面跟冒号，由Goto语句转向标号处。

LDRB ch, [src], #1

STRB ch, [dst], #1

CMP ch, #0

BNE loop

}

}

例2 内联汇编

int main(void)

{

const char \* a="Hello world!\n";

char b[20];

// do inline assembly routine str\_cpy(a,b)

\_\_asm

{

MOV R0,a // 将串a的串首地址送到R0寄存器

MOV R1,b // 将串b的串首地址送到R1寄存器

BL str\_cpy, {R0, R1} // 调用C函数str\_cpy()

}

printf("Original string: %s\n",a);

printf("Copied string: %s\n",b); // 半主机方式显示复制前后的两个串

return(0);

}

例3 c调用汇编

比较两个IP地址的大小，a1~a4存放IP地址1的值（按照ATPCS传递参数），b1~b4存放IP地址2的值（通过栈传递参数），如果IP地址1的值大于IP地址2的值则返回1，如果IP地址1的值小于IP地址2的值则返回-1 ，如果两者相等则返回零。

IP地址1取值：192, 168, 1, 152,

IP地址2取值： 172, 0, 0, 151

/\* C代码部分 \*/

#include <stdio.h>

extern int function(void); /\* 声明function是外部函数 \*/

int compare\_ip(int a1, int a2, int a3, int a4, int b1, int b2, int b3, int b4){

if(a1!=b1)

return a1>b1?1:-1;

if(a2!=b2)

return a2>b2?1:-1;

if(a3!=b3)

return a3>b3?1:-1;

if(a4!=b4)

return a4>b4?1:-1;

return 0; }

int main(){

printf("This is a example of semihosting\n");

printf("result is %d\n",function()); }

AREA FUNCTION, CODE, READONLY ;ARM汇编子程序

IMPORT compare\_ip

EXPORT function

function

STMFD r13!,{r0-r3,r14} ;保存寄存器到栈区

MOV r3,#0x97 ;存入IP地址2的4个数, 0x97=151

MOV r2,#0 ; 存入0

MOV r1,#0 ; 存入0

MOV r0,#0xac ; 存入0xac=172

STMIA r13, {r0-r3} ; R0-R4覆盖存入栈区的R0-R4位置

MOV r3,#0x98 ; 存入IP地址1的4个数, 0x98=152

MOV r2,#1 ; 存入1

MOV r1,#0xa8 ; 存入0xa8=168

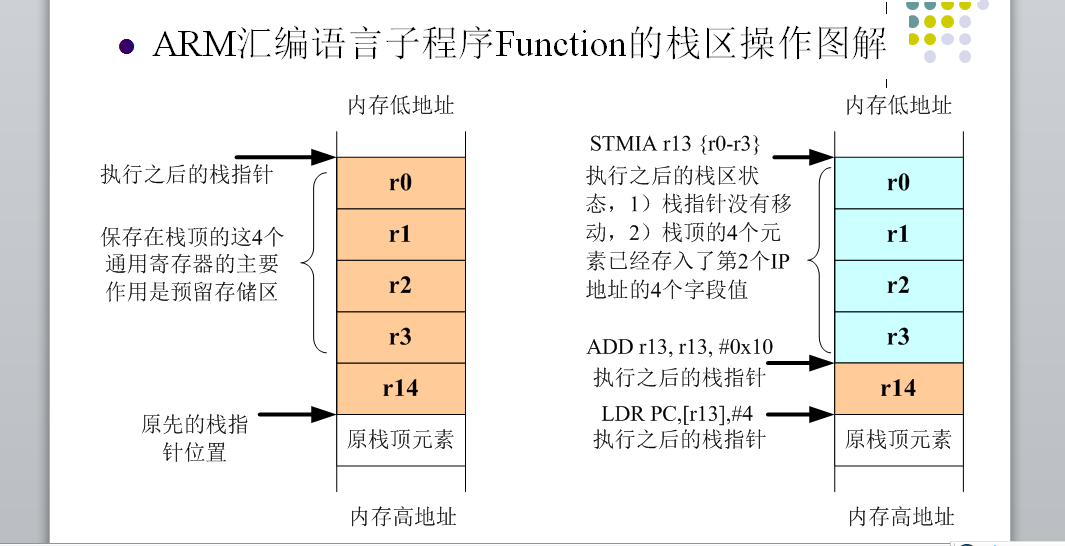
MOV r0,#0xc0 ; 存入0xc0=192

BL compare\_ip ; 0x0 ; 调用C语言函数进行IP值比较

ADD r13,r13,#0x10 ; 栈指针上移4个字（元素）

LDR pc,[r13],#4 ; 将保存的r14值加载到PC，而后r13加4

END ; 本段汇编代码的操作图解参看下一页



25. 关于GCC内联汇编进一步学习

C的参数传递与汇编语言的参数传递规则不同！

1）asm声明

/\* NOP 例子 \*/  
 asm("mov r0,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。

######说明：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"  
);

通用的内嵌汇编模版是这样的。

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

汇编和C语句这间的联系是通过上面asm声明中可选的output operand 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ARM Soc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。

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。

2）优化C代码

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("mov r0, r0");

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clobber list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dummy operand 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。

3）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 |
| h | Not available | Registers r8..r15 |
| G | Immediate floating point constant | Not available |
| H | Same a G, but negated | Not available |
| I | Immediate value in data processing instructions e.g. ORR R0, R0, #operand | Constant in the range 0 .. 255 e.g. SWI operand |
| J | Indexing constants -4095 .. 4095 e.g. LDR R1, [PC, #operand] | Constant in the range -255 .. -1 e.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 .. 7 e.g. SUB R0, R1, #operand |
| l | Same as r | Registers r0..r7 e.g. PUSH operand |
| M | Constant in the range of 0 .. 32 or a power of 2 e.g. MOV R2, R1, ROR #operand | Constant that is a multiple of 4 in the range of 0 .. 1020 e.g. ADD R0, SP, #operand |
| m | Any valid memory address | |
| N | Not available | Constant in the range of 0 .. 31 e.g. LSL R0, R1, #operand |
| O | Not available | Constant that is a multiple of 4 in the range of -508 .. 508 e.g. ADD SP, #operand |
| r | General register r0 .. r15 e.g. SUB operand1, operand2, operand3 | Not available |
| w | Vector floating point registers s0 .. s31 | Not available |
| X | Any operand | |

限制字符可能要单个modifier指示。要是没有modifier指示的默认为read-only operand。

|  |  |
| --- | --- |
| Modifier | Specifies |
| = | 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output operand。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");

4）其他

（1）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" \  
);

（2）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 long 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"   
);  
return val;  
}

（3）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。

（4）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。

（5）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 counter asm("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。

（6）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指令更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。

26. 汇编宏

宏是一段独立的程序代码，它是通过伪指令定义的，在程序中使用宏指令即可调用宏。当程序被汇编时，汇编程序将对每个调用进行展开，用宏定义取代源程序中的宏指令。

MACRO、MEND

语法格式：

MACRO

[$ label] macroname{ $ parameter1， $ parameter，…… }

指令序列

MEND

MACRO伪操作标识宏定义的开始，MEND标识宏定义的结束。用MACRO及MEND定义一段代码，称为宏定义体，这样在程序中就可以通过宏指令多次调用该代码段。其中，$ label在宏指令被展开时，label会被替换成相应的符号，通常是一个标号。在一个符号前使用$表示程序被汇编时将使用相应的值来替代$后的符号。

macroname为所定义的宏的名称。

$parameter为宏指令的参数。当宏指令被展开时将被替换成相应的值，类似于函数中的形式参数，可以在宏定义时为参数指定相应的默认值。

宏指令的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场，从而增加了系统的开销，因此，在代码较短且需要传递的参数较多时，可以使用宏汇编技术。

首先使用MACRO和MEND等伪操作定义宏。包含在 MACRO 和 MEND 之间的代码段称为宏定义体，在MACRO伪操作之后的一行声明宏的原型（包含宏名、所需的参数），然后就可以在汇编程序中通过宏名来调用它。在源程序被汇编时，汇编器将宏调用展开，用宏定义体代替源程序中的宏定义的名称，并用实际参数值代替宏定义时的形式参数。

宏定义中的$label是一个可选参数。当宏定义体中用到多个标号时，可以使用类似$label.$internallabel的标号命名规则使程序易读。

MACRO 、 MEND 伪操作可以嵌套使用。

使用示例：

MACRO   
  
$HandlerLabel HANDLER $HandleLabel   
  
$HandlerLabel   
  
sub sp,sp,#4 ;减少sp(用于存放转跳地址)实质上是在计算返回地址，用来存储PC地址  
stmfd sp!,{r0} ;把将要使用的r0寄存器入栈  
ldr r0,=$HandleLabel;将HandleXXX的址址放入r0   
ldr r0,[r0] ;把HandleXXX所指向的内容(也就是中断程序的入口)放入r0   
str r0,[sp,#4] ;把中断服务程序(ISR)压入栈.   
  
ldmfd sp!,{r0,pc} ;用出栈的方式恢复r0的原值和为pc设定新值(完成了到ISR的转跳)   
  
MEND

;首先这段程序是个宏定义，HANDLER是宏名，不要想歪了

;其次后面程序遇到的HandlerXXX HANDLER HandleXXX这些语句将都被下面这段程序展开

例如：HandlerFIQ   
HANDLER HandleFIQ 被上面那段程序展开后为：  
  
HandlerFIQ  
  
sub sp,sp,#4   
  
stmfd sp!,{r0}   
  
ldr r0,=HandleFIQ  
  
ldr r0,[r0]   
  
str r0,[sp,#4]  
  
ldmfd sp!,{r0,pc}

;再次这段程序目的在于把中断服务程序的首地址装载到PC中，可以称之为“加载程序”  
  
;本初始化程序定义了一个数据区（在文件最后），34个字空间，存放相应中断服务程序的首地址。每个字空间都有一个标号，以Handle\*\*\*命名。

例如：HandlerFIQ HANDLER HandleFIQ  
  
HandlerIRQ HANDLER HandleIRQ  
  
HandlerUndef HANDLER HandleUndef  
  
HandlerSWI HANDLER HandleSWI  
  
HandlerDabort HANDLER HandleDabort  
  
HandlerPabort HANDLER HandlePabort