1. 下载工具（J-Link eopenJTAG usb+uboot.bin（速度最快））
2. eopenJTAG

先安装eop的驱动程序，若安装不成功需要禁止win10的电子签名，并在设备管理器查找到该设备，进行设备驱动程序的安装，找到所下载的驱动程序的路径进行安装，驱动安装成功后，设备管理器不再提示感叹号，USB转串口的驱动安装也是如此。

烧写条件（Windows）适合烧写比较小的.bin文件 烧写速度慢

1. 软件：openocd(oflash)
2. 硬件：op/eop cmd中视为ftdi device设备
3. 可烧写文件类型：所有bin文件（uboot.bin以及所有裸机程序）

烧写过程（windows）

1. 打开cmd,切换到bin文件所在的盘符
2. cd打开bin文件所在的目录
3. oflash xxx.bin 也可通过绝对路径直接烧写
4. 依照命令提示选择烧写开发板、烧写工具、烧写区域
5. USB下载（uboot.bin nor flash启动 dnw程序下载软件）
6. 使用op/eop将u-boot烧写到nor flash
7. 开发板设置为nor flash启动，上电后马上在串口输入空格键，进入UBOOT
8. 连接PC----开发板的usd driver口 安装驱动
9. 根据超级中断里的串口提示输入命令（表示接收usb文件并烧写到NAND）
10. 使用DNW.exe发送bin文件
11. Uboot会自动接收、烧写bin文件
12. 断电、设为NAND启动、上电、运行程序
13. 恢复出厂系统（开发板变砖拯救）
14. 使用op/eop将u-boot烧写到nor flash（使用oflash）
15. 开发板设置为nor flash启动，上电后马上在串口输入空格键，进入UBOOT
16. 连接PC----开发板的usd driver口 安装驱动
17. 下载内核：在uboot的串口菜单中输入k 下载kernel内核文件
18. 使用dnw.exe发送uImage文件
19. Uboot即会自动接收、烧写uImage文件
20. 下载根文件系统RootFs ：在uboot的串口菜单中输入y
21. 使用dnw发送fs\_qtopia.yaffs2文件
22. Uboot即会自动接收、烧写uImage文件
23. 输入q退出uboot串口菜单，执行命令删除参数分区：nand erase params
24. 重启

补充：Linux bootloader+kernel+rootfs 引导程序+内核文件+根文件系统

1. Linux下执行代码的步骤（编译 编辑(gcc) 运行）
2. 编辑 vim hello.c 编写C语言程序
3. 编译 gcc -o hello hello.c (o-output hello-输出文件名 hello.c-编译源文件)

gcc:

gcc是linux系统下主要的编译软件

全称为GNU编译器套件（GNU Compiler Collection）

除了可以编译C语言开发的程序外，还可以开发C++、Java等多个语言的程序

GCC的初衷是为GNU操作系统专门编写的一款编译器

1. 运行 ./hello
2. Linux下进行裸机开发的步骤（编辑 编译（arm-linux-gcc） 烧写 运行）

补充：交叉编译模式

通常编译嵌入式程序的平台成为宿主机（ubuntu系统下的CPU架构为x86架构）

运行嵌入式程序的平台成为目标机（CPU为ARM架构的开发板）

在x86平台编译和编译器arm-linux-gcc编译ARM架构的程序

两者属于不同的架构平台，从而属于交叉编译模式

然后将程序烧写到ARM开发板中（JTAG USB SD卡 网络）

然后在开发板上运行该程序

1. 编辑器

Windows下的source insight和notepad（可查看makefile文件）

1. 编译器

推荐使用arm-linux-gcc（在Linux环境下使用） 一条命令（应用程序）

Windows下：建立工程🡪编辑🡪编译(VC 6.0)🡪运行

Linux下： 编辑🡪编译(gcc)🡪运行

1. 用汇编点亮一个LED灯
2. 常用汇编指令
3. LDR R0 , [R1] （读内存）

假设R1的值是x,把地址x上的数据保存到R0中

1. STR R0 , [R1] （写内存）

假设R1的值是x,把R0的值写到地址x去

1. B 跳转
2. MOV R0 , R1

把R1的值赋给R0，即R0=R1

或者MOV R0,#0X100 R=0x100 #表示立即数

1. LDR R0 , =0x12345678 （伪指令）

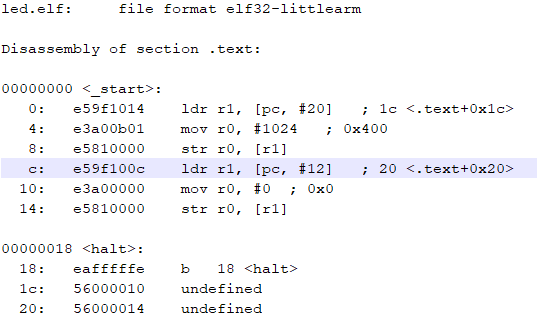
伪指令最终会被拆分为几条指令去执行 R0=0x12345678

但MOV R0 ,#0x12346578是错误的，这是因为ARM指令寄存器有32位，要有几位用来保存R0和MOV，剩下的不足32位，无法保存0x1234578，因此#后只能为简单值而且是立即数

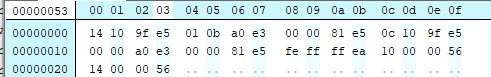
1. 编写汇编代码
2. S文件，在编辑器上进行编辑
3. 将.S文件通过FTP发送到虚拟机中
4. 在虚拟机中用arm-linux-gcc进行编译 连接 生成bin文件
5. 编译：arm-linux-gcc -c -o led.o led.S ->led.o
6. 连接：arm-linux-ld -Ttext 0 led.o -o led.elf ->led.elf
7. 生成bin文件：arm-linux-objcopy -O binary -S led.elf led.bin ->led.bin
8. 将以上三个命令写进Makefile文件中，必须是Makefile，大小写不能变
9. 将Makefile发到虚拟中，则编译生成bin文件只需要一下两条指令
10. make clean：rm \*.bin \*.o \*.elf
11. make 编译，生成bin文件
12. 通过USB和MiniTools将bin文件烧到Nand Flash中 上电重新运行

补充：汇编与机器码

先用指令arm-linux-objdump -D led.elf > led.dis 反编译汇编led.elf来查看其实际执行的汇编语言



其对应的二进制文件如下：

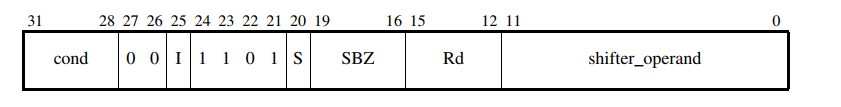


第一列为PC地址，第二列为机器码，即bin文件的内容，第三列为汇编码

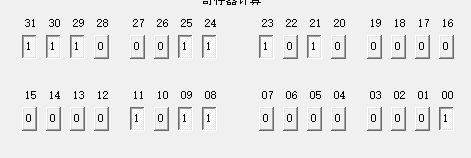
对于伪指令LDR R1 , =0x56000010，它的实际汇编代码为ldr r1, [pc, #20];1c<.text+0x1c>

r1=pc+20=0+8+20(当前pc地址+8+立即数)=0x1c，通过PC跳转到0x1c地址，它的内容为0x56000010，r1=0x56000010，PC可以说是程序指针寄存器

下图为MOV指令寄存器：4: e3a00b01 mov r0, #1024 ; 0x400



下图为MOV机器码对应寄存器的值



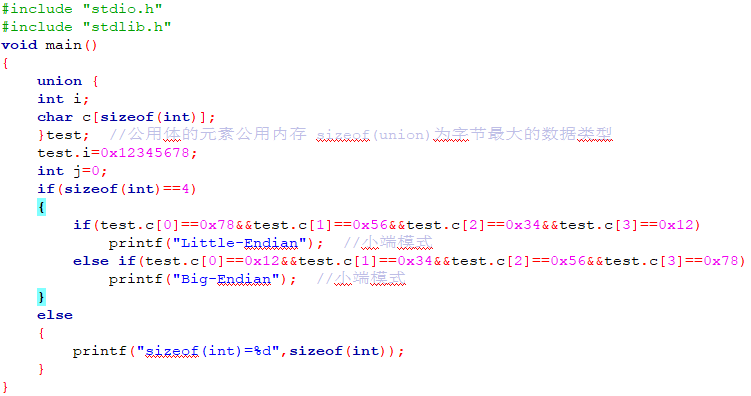
0-11位为立即数（高四位 低八位） 0x400 ->0000 0000 0000 0000 0000 0100 0000 0000

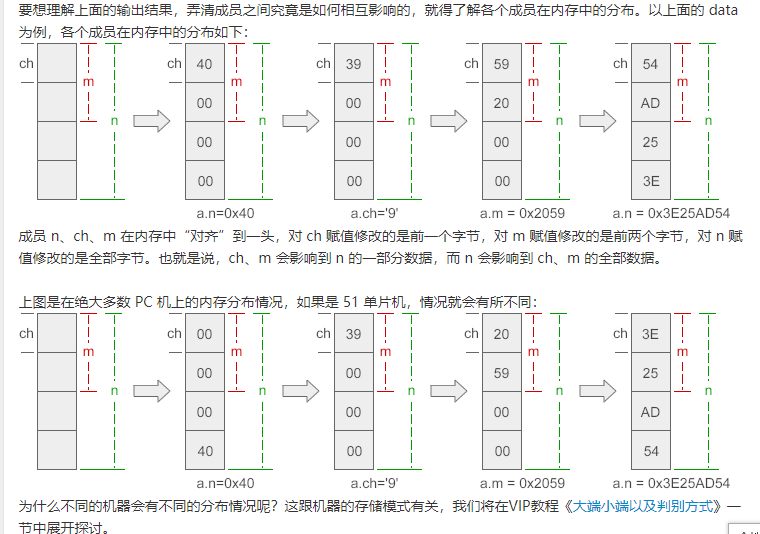
1011=11 高四位=移位数\*2 低八位为立即数

00000000000000000000000000000001->0x400:1循环右移1+21=22 22/2=11->1011(高四位)

因此可根据此规律直接更改机器码来改变运行现象，这也是计算机执行程序的原理

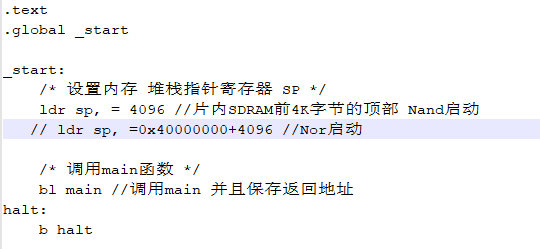
1. 寄存器的存储方式
2. 小端模式：低位存放在低地址 little endian S3C2440默认使用小字节序
3. 大端模式：低位存放在高地址 big endian
4. 掌握左移 右移 取反 位与 位或 置位(|(1<<7)|(1<<8) 清位（&~(1<<7)&(~(1<<8)））
5. 判断CPU字节序的C语言程序

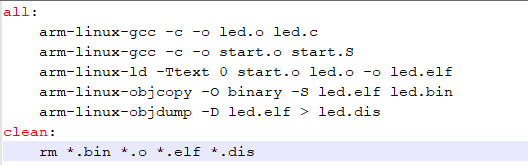




1. 编写C语言代码点灯

通过C指针对地址进行操作，写入对应寄存器的值，但最后还要通过汇编语言跳转到main()中，以及设置sp地址，最后生成bin文件，烧到Nand/NOR中





常用ARM指令：

1. sub

sub r1 , r2 , r3：r1=r2-r3

1. add

add r1 , r2 , r3：r1=r2+r3

1. bl(brarch and linlc)

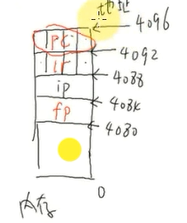
跳转到某条指令并将返回地址（下一条指令地址）保存到lr(连接寄存器)中

1. ldmia 读内存 写入多个寄存器 stmdb 写内存 把多个寄存器的值写入内存

ia:过后增加 ib:预先增加 da:过后减少 db:预先减少

EG： mov ip, sp //现将sp的值写入ip 假设sp=4096(地址)

stmdb sp!, {fp, ip, lr, pc}//先减后存 存储时高编号（R15 R14 ….）的寄存器存在高地址

 在保存数据时 栈的地址在递减，因此这叫降序栈 地址最终指向最后一个，因为这是满栈，如果第一个地址指向一个空单元，则为空栈

先减少：sp’=sp-4 = 4092

后保存：PC == R15 4092-4095存储PC

先减少：sp’=sp-4=4088

后保存：LR == R14 4088-4091存储LR

先减少：sp’=sp-4 = 4084

后保存：IP ==R12 4084-4087存储IP

先减少：sp’=sp-4 = 4080

后保存：FP ==R11 4080-4083存储FP

由于sp后面有!，则存储完毕后 sp=sp’=4080

ldmia sp, {fp, sp, pc} //先读后加 高编号寄存器存放高地址内存值

先读：FP=4090-4083的内存值=原来保存的FP

后增：sp’=sp+4=4084

先读：SP=4084-4087的内存值=原来保存的IP值 此处已经将SP的值更改（mov ip,sp）因此此处sp后方没有加!

后增：sp’= sp+4=4088

先读：PC=4088-4091=原来保存的LR的值 即将返回地址保存到PC当中

后增：sp’=sp+4=4092

补充：SP LR PC简介

1、堆栈指针r13（SP）：每一种异常模式都有其自己独立的r13，它通常指向异常模式所专用的堆栈，也就是说五种异常模式、非异常模式（用户模式和系统模式），都有各自独立的堆栈，用不同的堆栈指针来索引。这样当ARM进入异常模式的时候，程序就可以把一般通用寄存器压入堆栈，返回时再出栈，保证了各种模式下程序的状态的完整性。

2、连接寄存器r14（LR）：每种模式下r14都有自身版组，它有两个特殊功能。

    （1）保存子程序返回地址。使用BL或BLX时，跳转指令自动把返回地址放入r14中；子程序通过把r14复制到PC来实现返回，通常用下列指令之一：  
                        MOV PC, LR   
                        BX LR

    通常子程序这样写，保证了子程序中还可以调用子程序。  
                         stmfd sp!, {lr}  
                         ……  
                         ldmfd sp!, {pc}

    （2）当异常发生时，异常模式的r14用来保存异常返回地址，将r14如栈可以处理嵌套中断。

3、程序计数器r15（PC）：PC是有读写限制的。当没有超过读取限制的时候，读取的值是指令的地址加上8个字节，由于ARM指令总是以字对齐的，故bit[1:0]总是00。当用str或stm存储PC的时候，偏移量有可能是8或12等其它值。在V3及以下版本中，写入bit[1:0]的值将被忽略，而在V4及以上版本写入r15的bit[1:0]必须为00，否则后果不可预测。

5. C语言内部机制 以点灯代码为例 （栈为SP所指向的内存）

led.elf: file format elf32-littlearm

Disassembly of section .text:

00000000 <\_start>:

0: e3a0da01 mov sp, #4096 ; 0x1000 sp=4096

4: eb000000 bl c <main> ;跳转到main，并将返回地址保存为0x0000 0008

00000008 <halt>:

8: eafffffe b 8 <halt> ;无限循环

0000000c <main>:

c: e1a0c00d mov ip, sp ;ip=sp=4096

10: e92dd800 stmdb sp!, {fp, ip, lr, pc} ;保存寄存器 pc=0x18 lr=0x08 ip=4096 fp未知

14: e24cb004 sub fp, ip, #4 ; 0x4 ;fp=ip-4=4092

18: e24dd008 sub sp, sp, #8 ; 0x8 ;sp=sp-8=4076 ;以上四句为保存寄存器

1c: e3a03456 mov r3, #1442840576 ; 0x56000000 r3=0x56000000

20: e2833010 add r3, r3, #16 ; 0x10 r3=0x56000010

24: e50b3010 str r3, [fp, #-16] ;将0x56000010存入SP栈中的fp-16=4076 对应C 语言中的unsigned int \*pGPBCON=(unsigned int \*)0x56000010

28: e3a03456 mov r3, #1442840576 ; 0x56000000

2c: e2833014 add r3, r3, #20 ; 0x14

30: e50b3014 str r3, [fp, #-20] ;同上 对应C语言中的unsigned int \*pGPBDATA=(unsigned int \*)0x56000014; 存入4072

34: e51b2010 ldr r2, [fp, #-16] ;读取fp-16=4076的值到r2中 即r2=0x56000010

38: e3a03a01 mov r3, #4096 ; 0x1000 r3=0x1000

3c: e5823000 str r3, [r2] ;r2=r3 对应C语言中的\*pGPBCON=0x1000

40: e51b2014 ldr r2, [fp, #-20]

44: e3a03000 mov r3, #0 ; 0x0

48: e5823000 str r3, [r2] ;同上 对应C语言中的\*pGPBDATA=0X0

4c: e3a03001 mov r3, #1 ; 0x1

50: e1a00003 mov r0, r3 ;对应C语言中的return0 被调用者通过r3向调用者传递参数 r0 r1 r2 r3 通常用来传递调用者和被调用者之间的参数, ;在函数中,r4-r11也可能会被使用,因此在入口处保存他们,在出口处恢复他们

54: e24bd00c sub sp, fp, #12 ; 0xc ;sp=fp-12=4080

58: e89da800 ldmia sp, {fp, sp, pc} ;从栈中恢复寄存器 fp=[4080]=原来保存的的FP sp=[4084]=4096 pc=[4088]=LR=0x08

;comment是注释的意思 以下文字不会被写入bin文件中 它们只是做注释用

Disassembly of section .comment:

00000000 <.comment>:

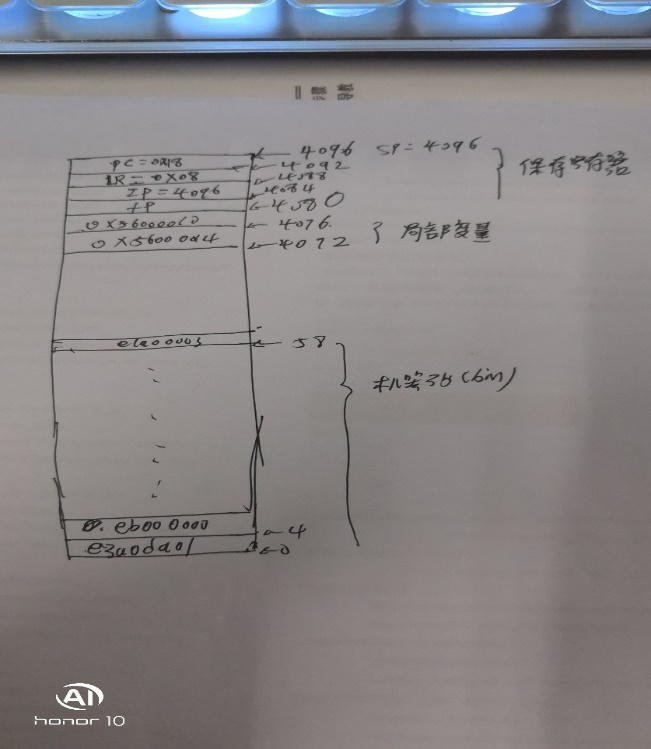
0: 43434700 cmpmi r3, #0 ; 0x0

4: 4728203a undefined

8: 2029554e eorcs r5, r9, lr, asr #10

c: 2e342e33 mrccs 14, 1, r2, cr4, cr3, {1}

10: Address 0x10 is out of bounds.



补充：

1. 关于S3C2440的看门狗

看门狗配置寄存器如下图所示，复位之后，看门狗默认开启

在Nor Flash下启动，若不手动关闭看门狗，则需要在程序中定期喂狗，不然程序会不断复位

在Nand Flash下启动，看门狗默认关闭，如果需要使用看门狗，需要手动开启

//开启看门狗

ldr r0 , =0x53000000

ldr r1 , =0x8021

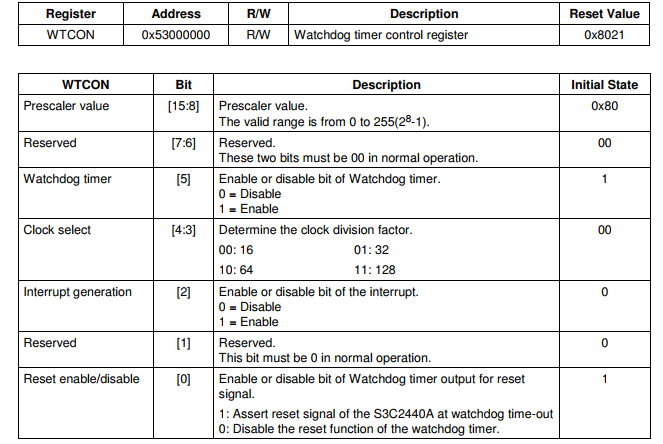
str r1 , [r0]

//关闭看门狗

ldr r0 , =0x53000000

ldr r1 , =0x8021

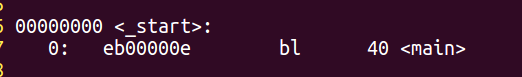
str r1 , [r0]



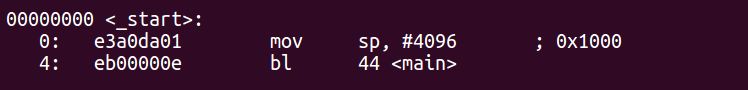
1. Ldr sp,#4096的作用

函数调用和局部变量的存储需要使用到一种叫做栈的数据结构，因此要调用main函数，我们需要开辟栈，这里使用片内4K内存作为栈，即在堆栈指针寄存器sp中进行申请，如果不申请，系统会自动开辟，建议还是在汇编语言中进行栈的申请

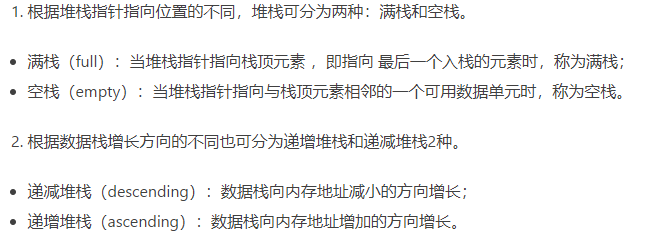
没有手动申请堆栈（反汇编文件 led.dis）：



手动申请堆栈：



另外，s3c2440，采用默认的栈生长方式，这也是我们最常见的方式，高地址往低地址生长，即降序栈，一下是几种栈的介绍（数据栈的使用规则）：





有一点是需要注意的，虽然ARM处理器核对于两种生长方式的堆栈均支持，但ads的C语言编程只支持一种方式，即从上往下生长，并且必须是满递减堆栈。所以stmfd等指令用的最多。

ARMv7 cortex-A9架构下，APCS规定数据栈为满递减的。

ARM对于堆栈的操作一般采用ldmia（pop）和stmdb（push）两个命令



常见名词：

数据栈指针：最后一个写入栈的数据的内存地址

数据栈的基地址：数据栈的最高地址，由于APCS中的数据栈是FD类型的，多以最早入栈的数据所占的内存单元的基地址的下一个内存单元。

数据栈界限：数据栈可使用的最低的内存单元地址

已用的数据栈：数据栈的基地址和数据栈的栈指针之间内存区域，包括栈指针对应的内存单元，但不包括基地址对应的内存单元。