之前看了很多关于uboot分析类的文章，其中提到为C语言的运行准备栈。而在uboot start.S汇编代码中，关于系统初始化，也看到栈指针初始化，即正确给栈指针sp赋值，却从来没看到有人解释，为何要这样做。接下来，我试图解释这个问题。

首先了解栈的作用。关于这个，详细讲解要很长的篇幅，故此处只做简略介绍。总的来说，它的作用就是：保存现场/上下文，传递参数，保存临时变量

**1.   保存现场/上下文**

现场/上下文相当于案发现场，总有一些案发现场，要记录下来，否则被别人破坏，便无法恢复。而此处说的现场，是指CPU运行时，用到的一些寄存器，比如r0,r1等，对于这些寄存器的值，如果不保存而直接跳转到子函数中执行，其很可能被破坏，因为其函数执行也要用到这些寄存器。因此，在函数调用之前，应该将这些寄存器等现场暂时保存(入栈push)，等调用函数执行完毕后出栈(pop)再恢复现场。这样CPU就可以正确的继续执行了。

保存寄存器的值，一般用push指令，将对应的某些寄存器的值，一个个放到栈中，即所谓的压栈。然后待被调用的子函数执行完毕后再调用pop，把栈中的一个个的值，赋值给对应的那些你刚开始压栈时用到的寄存器，把对应的值从栈中弹出去，即所谓的出栈。

其中保存的寄存器中，也包括lr的值（因为用bl指令进行跳转的话，之前的pc值存在lr中），在子程序执行完毕后，再把栈中的lr值pop出来，赋值给pc，这样就实现了子函数的正确的返回。

**2. 传递参数**

C语言函数调用时，会传给被调用函数一些参数，对于这些C语言级别参数，被编译器翻译成汇编语言时，要找个地方存放下来，并且让被调用函数能访问，否则没法传递。找个地方存放下来分2种情况。一是，本身传递的参数不多于4个，可以通过寄存器传送。因为在前面的保存现场动作中，已经保存好对应的寄存器的值，此时这些寄存器是空闲的，可以供我们使用存放参数。二是，参数多于4个，寄存器不够用，就得用栈。

**3. 临时变量保存在栈中**

这些临时变量包括函数的非静态局部变量以及编译器自动生成的其他临时变量。

**举例分析C语言函数调用如何使用栈**

上面的解释有些抽象，此处再用例子简单说明一下，就容易明白了：

用arm-inux-objdump –d u-boot dump\_u-boot.txt得到dump\_u-boot.txt文件。该文件是包含了u-boot可执行汇编代码，从中我们可以看到相应C程序对应的汇编代码。

下面贴出两个函数的汇编代码，一个是clock\_init，另一个是与clock\_init在同一C源文件中的函数CopyCode2Ram：

33d0091c： CopyCode2Ram:

33d0091c: e92d4070  push   {r4, r5, r6, lr}

33d00920: e1a06000  mov r6, r0

33d00924: e1a05001  mov r5, r1

33d00928: e1a04002  mov r4, r2

33d0092c: ebffffef  bl  33d008f0 b BootFrmNORFlash

......

33d00984: ebffff14  bl  33d005dc nand\_read\_ll

......

33d009a8: e3a00000  mov r0, #0 ; 0x0

33d009ac: e8bd8070  pop {r4, r5, r6, pc}

33d009b0：clock\_init:

33d009b0: e3a02313  mov r2, #1275068416   ;0x4c000000

33d009b4: e3a03005  mov r3, #5 ; 0x5

33d009b8: e5823014  str r3,

......

33d009f8: e1a0f00e  mov pc, lr

(1) 先分析clock\_init对应的汇编代码，可以看到该函数第一行

：33d009b0: e3a02313  mov r2, #1275068416   ;0x4c000000

没有我们期望的push指令，即没有将一些寄存器的值放入栈。这是因为，clock\_init用到的r2,r3等寄存器，和前面调用clock\_init前用到的寄存器r0，没有冲突，故此处不用push保存，有个寄存器要注意，r14，即lr，前面调用clock\_init时，用的bl指令，所以会自动把跳转时的pc值赋值给lr，所以也不需要push将PC值保存到栈。而clock\_init对应的汇编代码最后一行: 33d009f8: e1a0f00e mov pc, lr 就是我们常见的mov pc,lr，把lr值，即之前保存的函数调用时的PC值，赋值给现在的PC，这样便实现了函数的正确返回，即返回到了函数调用时下一个指令的位置。CPU可以继续执行原先函数内剩下的代码。

(2) CopyCode2Ram对应汇编代码第一行：33d0091c: e92d4070 push {r4, r5, r6, lr}

就是我们所期望的，用push保存r4,r5,r6，lr，是因为此函数还包括其他函数调用

：33d0092c:  ebffffef bl  33d008f0  b BootFrmNORFlash......

33d00984: ebffff14  bl  33d005dc nand\_read\_ll

......

也用到bl指令，会改变我们最开始进入clock\_init时的lr值，所以也要push暂时保存起来。

而对应地，CopyCode2Ram最后一行：33d009ac: e8bd8070 pop {r4, r5, r6,pc}是把之前push的值给pop出来，还给对应的寄存器，其中最后一个是将开始push的lr的值pop出来赋给PC，实现了函数的返回。另外我们注意到，CopyCode2Ram的倒数第二行：33d009a8: e3a00000 movr0, #0 ; 0x0 是把0赋值给r0寄存器，就是我们说的返回值的传递，此处的返回值为0，也对应着C代码中的“ return 0”。

当然也可以用其他暂时空闲没有用到的寄存器来传递返回值。

对于使用哪个寄存器来传递返回值，是根据ARM的APCS寄存器的使用约定而设计的，最好按照其约定的来处理，不要随便改变它。这样程序将更加规范。