**点亮led**

C++版本（led.cpp）：

|  |
| --- |
| int main(int argc,char\* argv[])  {  unsigned int\* pGPFCON = (unsigned int\*)0x56000050;  unsigned int\* pGPFDAT = (unsigned int\*)0x56000054;    \*pGPFCON = 0x100;  \*pGPFDAT = 0;  return 0;  } |

单纯编译该文件生成.bin文件放到开发板上是无法运行的，因为其使用到了局部变量，那就一定会使用栈，但是程序中并没有分配栈空间；我们知道，main也是一个函数，它的执行需要别的东西来调用，但是这个程序中并没有任何调用main的地方，因此该函数也不会执行。解决这两个问题，我们需要一个汇编代码（start.S）：

|  |
| --- |
| .text  .global \_start  \_start:  /\*设置内存 sp\*/  //nand启动  ldr sp, =4096  //nor启动  //ldr sp, =0x40000000+4096  /\*调用 main\*/  bl main  halt:  b halt |

该汇编简单设置了当使用nand启动时栈空间的大小为4K(4096),同时调用了main函数。

我们使用这个makefile来编译上述两个程序：

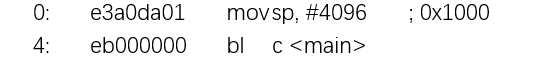
|  |
| --- |
| all:  arm-linux-g++ -c -o led.o led.cpp  arm-linux-gcc -c -o start.o start.S  arm-linux-ld -Ttext 0 start.o led.o -o led.elf  arm-linux-objcopy -O binary -S led.elf led.bin  arm-linux-objdump -D led.elf > led.dis  clean:  rm -rf \*.elf \*.bin \*.o \*.dis |

这里需要注意：在使用arm-linux-ld链接led.o与start.o时，两个文件是有顺序的，必须让start.o在led.o前，否则生成的.bin文件无法起到效果。

接下来我们分析一下生成的.bin文件的反汇编：

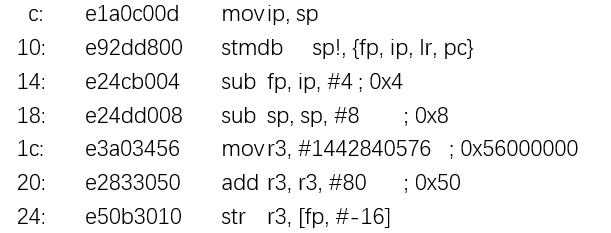
|  |
| --- |
| led.elf: file format elf32-littlearm  Disassembly of section .text:  00000000 <\_start>:  0: e3a0da01 mov sp, #4096 ; 0x1000  4: eb000000 bl c <main>  00000008 <halt>:  8: eafffffe b 8 <halt>  0000000c <main>:  c: e1a0c00d mov ip, sp  10: e92dd800 stmdb sp!, {fp, ip, lr, pc}  14: e24cb004 sub fp, ip, #4 ; 0x4  18: e24dd008 sub sp, sp, #8 ; 0x8  1c: e3a03456 mov r3, #1442840576 ; 0x56000000  20: e2833050 add r3, r3, #80 ; 0x50  24: e50b3010 str r3, [fp, #-16]  28: e3a03456 mov r3, #1442840576 ; 0x56000000  2c: e2833054 add r3, r3, #84 ; 0x54  30: e50b3014 str r3, [fp, #-20]  34: e51b2010 ldr r2, [fp, #-16]  38: e3a03c01 mov r3, #256 ; 0x100  3c: e5823000 str r3, [r2]  40: e51b2014 ldr r2, [fp, #-20]  44: e3a03000 mov r3, #0 ; 0x0  48: e5823000 str r3, [r2]  4c: e3a03000 mov r3, #0 ; 0x0  50: e1a00003 mov r0, r3  54: e24bd00c sub sp, fp, #12 ; 0xc  58: e89da800 ldmia sp, {fp, sp, pc}  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. |

第一列是地址，第二列是机器码，第三列是机器码对应的汇编代码。当程序执行时，系统从0地址处开始读取指令进行处理。因此系统从这里开始执行：



第一条指令表示将立即数4096存放到sp寄存器中，第二条指令表示调用main函数，并返回下一条指令的地址（也就是8）。

接下来进入到main函数，对应到main函数中的语句，



这六条指令表示将局部变量pGPFCON变量的值设为0x56000050。具体的：

第一条指令表示将sp的值存放到ip寄存器中，也就是[ ip = sp = 4096];

第二条指令使用了stm指令，db表示先减，stm指令表示读取寄存器，所以stmdb表示先减再读；sp!表示将改变后的sp的只保存到sp中；stm的读取规则是：高编号的寄存器放到高地址中，这里fp是r11寄存器，ip是r12寄存器，lr是r14寄存器，pc是r15寄存器。

4072(至此，pGPFDAT变量存放)

4076(至此，pGPFCON变量存放)

4080（至此第二条语句结束，sp = 4080）

4088

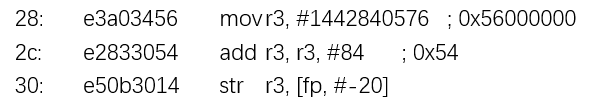
4092

4096

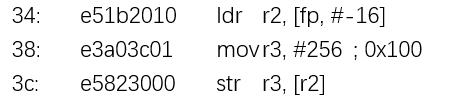
4084

|  |
| --- |
| Pc = 0x18 |
| Lr = 0x8 |
| Ip = 4096 |
| Fp未知 |
| 0x56000050 |
| 0x56000054 |

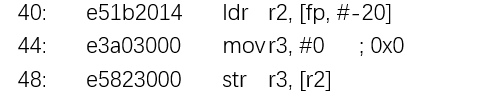
这里lr寄存器中存放了main函数结束时的返回地址，也就是下一条语句开始。接下来就是存储pGPFDAT的值到栈中。



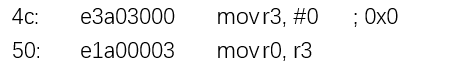
接下来就是为\*pGPFCON赋值：



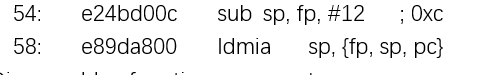
为\*pGPFDAT赋值：



准备结束main函数，给寄存器r0送去返回值0:



Main函数返回，系统将恢复之前的寄存器：



此时，fp=4092,所以sp=4080，ldm是加载的意思，ia后缀表示后增，连起来ldmia就是先加载后增加。加载规则也是高编号寄存器加载高地址，即：fp未定义，sp=ip=4096,pc=0x8。

接下来系统根据pc的值跳转到halt执行。