# 散装 vs 批发谁效率高？变量访问被ARM架构安排的明明白白

原创 土豆居士 一口Linux 2020-12-15 06:10

收录于合集
#从0学arm 27 #所有原创 206

作为过来人，我发现很多程序猿新手，在编写代码的时候，特别喜欢定义很多独立的全局变量，而不是把这些变量封装到一个结构体中，主要原因是图方便，但是要知道，这其实是一个不好的习惯，而且会降低整体代码的性能。

另一方面，最近有幸与大神**「公众号：裸机思维」**的傻孩子交流的时候，他聊到：“其实Cortex在架构层面就是更偏好面向对象的（哪怕你只是使用了结构体），其表现形式就是：**「Cortex所有的寻址模式都是间接寻址」**——换句话说**「一定依赖一个寄存器作为基地址」**。

举例来说，同样是访问外设寄存器，过去在8位和16位机时代，人们喜欢给每一个寄存器都单独绑定地址——当作全局变量来访问，而现在Cortex在架构上更鼓励底层驱动以寄存器页（也就是结构体）为单位来定义寄存器，这也就是说，同一个外设的寄存器是借助拥有同一个基地址的结构体来访问的。”

以Cortex A9架构为前提，下面一口君详细给你解释为什么使用结构体效率会更高一些。

## 一、全局变量代码反汇编

### 1. 源文件

**「gcd.s」**

```
.text
.global _start
_start:
  ldr  sp,=0x70000000          /*get stack top pointer*/
  b  main
```

**「main.c」**

```
/*
 * main.c
 *
 *  Created on: 2020-12-12
 *      Author: pengdan
 */
int xx=0;
int yy=0;
int zz=0;

int main(void)
{
 xx=0x11;
 yy=0x22;
 zz=0x33;

 while(1);
    return 0;
}
```

「map.lds」

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 . = 0x40008000;
 . = ALIGN(4);
 .text      :
 {
  gcd.o(.text)
  *(.text)
 }
 . = ALIGN(4);
    .rodata :
 { *(.rodata) }
    . = ALIGN(4);
    .data :
 { *(.data) }
    . = ALIGN(4);
    .bss :
     { *(.bss) }
}
```

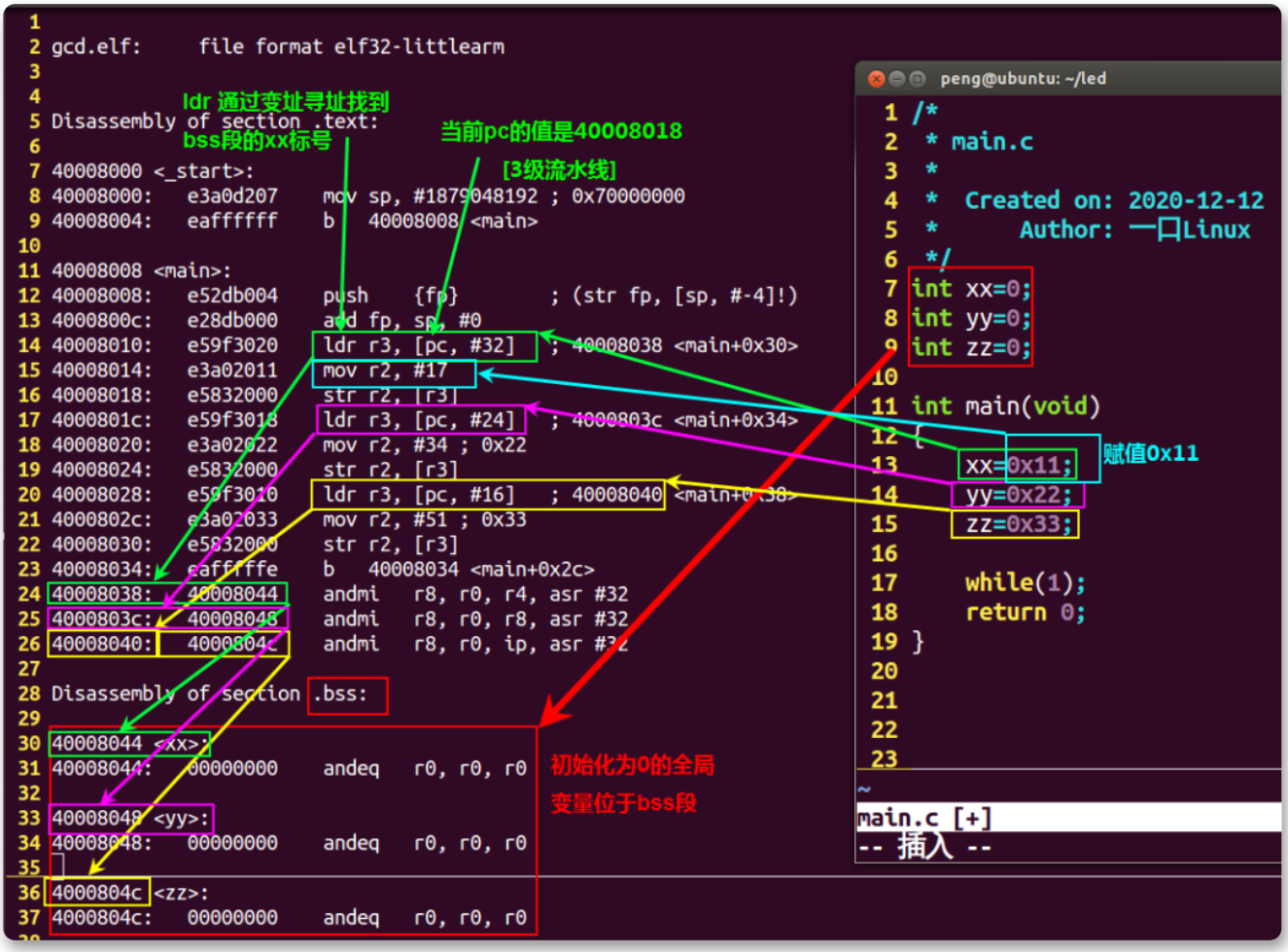
「Makefile」

```
TARGET=gcd
TARGETC=main
all:
 arm-none-linux-gnueabi-gcc -O1 -g -c -o $(TARGETC).o  $(TARGETC).c
 arm-none-linux-gnueabi-gcc -O1 -g -c -o $(TARGET).o $(TARGET).s
 arm-none-linux-gnueabi-gcc -O1 -g -S -o $(TARGETC).s  $(TARGETC).c
 arm-none-linux-gnueabi-ld $(TARGETC).o $(TARGET).o -Tmap.lds  -o  $(TARGET).elf
 arm-none-linux-gnueabi-objcopy -O binary -S $(TARGET).elf $(TARGET).bin
 arm-none-linux-gnueabi-objdump -D $(TARGET).elf > $(TARGET).dis

clean:
 rm -rf *.o *.elf *.dis *.bin
```

【交叉编译工具，自行搜索安装】

## 2. 反汇编结果：




由上图可知，每存储1个int型全局变量需要「8个字节」，

「literal pool （文字池）占用4个字节」

literal pool的本质就是ARM汇编语言代码节中的一块用来存放常量数据而非可执行代码的内存块。

```
24 40008038:    40008044    andmi    r8, r0, r4, asr #32
```

使用literal pool （文字池）的原因

当想要在一条指令中使用一个 4字节长度的常量数据（这个数据可以是内存地址，也可以是数字常量）的时候，由于ARM指

在C源代码中，文字池的分配是由编译器在编译时自行安排的，在进行汇编程序设计时，开发者可以自己进行文字池的分配

「bss段占用4个字节」

```
30 40008044 <xx>:
31 40008044:    00000000    andeq    r0, r0, r0
```

每访问1次全局变量，总共需要3条指令，访问3次全局变量用了「12条指令」。

```
14 40008010:    e59f3020    ldr r3, [pc, #32]   ; 40008038 <main+0x30>
15 40008014:    e3a02011    mov r2, #17
16 40008018:    e5832000    str r2, [r3]
```

14. 通过当前pc值40008018偏移32个字节，找到xx变量的链接地址40008038，然后取出其内容40008044存放在r3中，该
15. 通过将立即数0x11即#17赋值给r2
16. 将r2的内容那个写入到r3对应的指向的内存，即xx标号对应的内存中

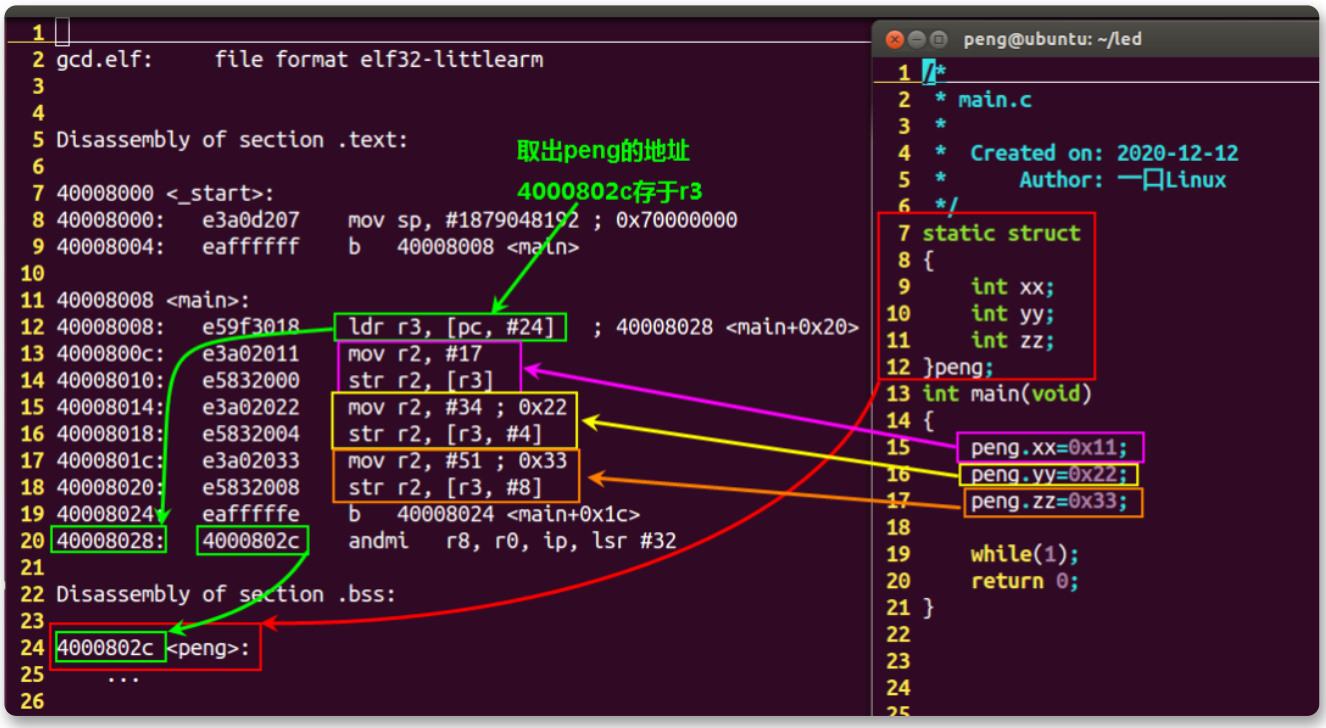
# 二、结构体代码反汇编

## 1. 修改main.c如下：

```
/*
 2  * main.c
 3  *
 4  *  Created on: 2020-12-12
```

```
 5  *      Author: 一口Linux
 6  */
 7 struct
 8 {
 9     int xx;
10     int yy;
11     int zz;
12 }peng;
13 int main(void)
14 {
15     peng.xx=0x11;
16     peng.yy=0x22;
17     peng.zz=0x33;
18
19     while(1);
20     return 0;
21 }
```

## 2. 反汇编代码如下：




由上图可知：

1. 结构体变量peng位于bss段，地址是4000802c

2. 访问结构体成员也需要利用pc找到结构体变量peng对应的文字池中地址40008028，然后间接找到结构体变量peng地址4000802c

与定义成3个全局变量相比，优点：

1. 结构体的所有成员在literal pool 中共用同一个地址；而每一个全局变量在literal pool 中都有一个地址，**「节省了8个字节」**。
2. 访问结构体其他成员的时候，不需要再次装载基地址，只需要2条指令即可实现赋值；访问3个成员，总共需要**「7条指令」**，**「节省了5条指令」**

**「彩！」**

所以对于需要大量访问结构体成员的功能函数，所有访问结构体成员的操作只需要加载一次基地址即可。

使用结构体就可以大大的节省指令周期，而节省指令周期对于提高cpu的运行效率自然不言而喻。

**「所以，重要问题说3遍」**

**「尽量使用结构体」「尽量使用结构体」「尽量使用结构体」**

# 三、继续优化

那么指令还能不能更少一点呢？答案是可以的， 修改Makefile如下：

```
TARGET=gcd
TARGETC=main
all:
     arm-none-linux-gnueabi-gcc -Os   -lto -g -c -o $(TARGETC).o  $(TARGETC).c
     arm-none-linux-gnueabi-gcc -Os  -lto -g -c -o $(TARGET).o $(TARGET).s
     arm-none-linux-gnueabi-gcc -Os  -lto -g -S -o $(TARGETC).s  $(TARGETC).c
     arm-none-linux-gnueabi-ld   $(TARGETC).o    $(TARGET).o -Tmap.lds  -o  $(TARGET).elf
     arm-none-linux-gnueabi-objcopy -O binary -S $(TARGET).elf $(TARGET).bin
     arm-none-linux-gnueabi-objdump -D $(TARGET).elf > $(TARGET).dis
clean:
     rm -rf *.o *.elf *.dis *.bin
```

仍然用第二章的main.c文件

```
 7 40008000 <_start>:
 8 40008000:   e3a0d207    mov sp, #1879048192 ; 0x70000000
 9 40008004:   eaffffff    b   40008008 <main>
10
11 Disassembly of section .text.startup:
12
13 40008008 <main>:
14 40008008:   e59f3010    ldr r3, [pc, #16]   ; 40008020 <main+0x18>
15 4000800c:   e3a00011    mov r0, #17
16 40008010:   e3a01022    mov r1, #34 ; 0x22
17 40008014:   e3a02033    mov r2, #51 ; 0x33
18 40008018:   e8830007    stm r3, {r0, r1, r2}
19 4000801c:   eafffffe    b   4000801c <main+0x14>
20 40008020:   40008024    andmi   r8, r0, r4, lsr #32
21
22 Disassembly of section .bss:
23
24 40008024 <peng>:
25     ...
```

执行结果

可以看到代码已经被优化到5条。

14. 把peng的地址40008024装载到r3中
15. r0写入立即数0x11
16. r1写入立即数0x22
17. r0写入立即数0x33
18. 通过stm指令将r0、r1、r2的值顺序写入到40008024内存中

「彩！ 彩！ 彩！ 彩！」

文中用到的汇编知识可以参考ARM系列文章

《从0学arm合集》

## 推荐阅读

【1】嵌入式工程师到底要不要学习ARM汇编指令？**必读**

【2】7. 从0学ARM-汇编伪指令、lds详解

【3】IP协议入门**必读**

【4】【从0学ARM】你不了解的ARM处理异常之道

【5】4. 从0开始学ARM-ARM汇编指令其实很简单

【6】【典藏】大佬们都在用的结构体进阶小技巧

【7】[粉丝问答6]子进程进程的父进程关系

【8】C和汇编如何互相调用？嵌入式工程师必须掌握

进群，请加一口君个人微信，带你嵌入式入门进阶。


收录于合集 #从0学arm 27

上一篇
C和汇编如何互相调用？嵌入式工程师必须掌握

下一篇
9. 基于Cortex-A9 LED汇编、C语言驱动编写

喜欢此内容的人还喜欢

Konva实现图片自适应裁剪

A逐梦博客

pinia

睡不着所以学编程

---

# 面试连环问--操作系统

阿Q正砖