

# gcc 对整型和浮点型参数传递的汇编码生成特点分析

张昱

## 1. 相关资料

关于浮点数(Floating-point)的存储表示: 浮点数的存储目前广泛采用 IEEE 754 标准(1980 年 Intel 提出, 1985 年被 IEEE 采纳, <http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html> )。

- 32 位单精度: Bit 31 是符号位, Bits 30~23 是指数部分, Bits22~0 是尾数部分, 即有效数字部分
  - 64 位双精度: Bit 63 是符号位, Bits 62~52 是指数部分, Bits51~0 是尾数部分, 即有效数字部分
- <http://sourceware.org/binutils/docs/as/CFI-directives.html>

关于 x87 FPU 编程: x87 FPU(Floating-Point Unit)能为图形处理、科学计算等提供高性能的浮点处理能力。它支持浮点数、整数和紧致 BCD 整数数据类型, 支持 IEEE 754 标准中为二进制浮点运算定义的浮点处理算法和异常处理体系。有关 x87 FPU 编程可参阅:

- *IA32 Intel Architecture Software Developer's Manual*, Intel Corporation, 2003. [V1: Basic Architecture](#)  
Chapter 8 Programming with the x87 FPU

**x87 FPU 数据寄存器:** 有 8 个 80 位的数据寄存器, 编号为 0~7。80 位中 Bit79 是符号位, Bits78-64 是指数位, Bits63-0 是存放有效数字。x87 FPU 指令将这 8 个寄存器看成一个寄存器栈。汇编器允许用 ST(0)或 ST 来表示当前的栈顶, ST(i)表示相对于栈顶的第 i 个寄存器。

x87 FPU 数据寄存器的内容不受过程调用影响。正在调用的过程可以使用 x87 FPU 数据寄存器在过程之间传递参数, 被调用的过程可以引用通过寄存器栈传递过来的参数; 被调用过程也可以将返回值存入 ST(0), 再将控制转移给调用者。

**x87 FPU 状态寄存器:** 有 1 个 16 位的状态寄存器。

Bit15: busy flag;

Bits 13-11: 栈顶指针, 取值范围为 0~7, 表示栈顶指向的数据寄存器的编号;

Bits 14,10-8: 条件码 C3-C0; 关于条件码的解释可以参见 [V1: Basic Architecture](#) 的表 8-1。

Bit7: ES flag 标记是否有错; Bit6: SF flag 标记是否是栈错误

Bits5-0: 是一组异常标志位, 依次为 PE(精度)、UE(下溢)、OE(上溢)、ZE(除 0)、DE(不规范的操作数)、IE(无效操作)。

x87 FPU 状态寄存器可以通过 fstsw/fnstsw、fstenv/fnstenv、fsave/fnsave 和 fxsave 指令存入内存, 也可以通过 fstsw/fnstsw 存入整数寄存器 eax 的低 16 位(ax 寄存器)中。

**x87 FPU 控制字:** 16 位 x87 FPU 控制字控制 x87 FPU 的精度和所使用的舍入法(rounding method)。FPU 控制字缓存在控制寄存器中, 可通过 fldcw 指令来加载到控制寄存器, 或通过 fstcw/fnstcw 指令来存入内存。

Bits15-13, 7-6: 保留位

Bit12: Infinity 无穷大控制位;

Bits11-10: Rounding 舍入模式控制位; 00B-Round to nearest(向最近的一边方向)、01B-Round down(向 $-\infty$ 方向舍入)、10B-Round up(向 $+\infty$ 方向舍入)、11B-Round toward zero(向零方向舍入, truncate 截断)

Bits 9-8: Precision 精度控制位;00B-单精度(24 位)、01B-保留、10B-双精度(53 位)、11B-扩展双精度(64 位)

Bits5-0: 是一组异常标志位, 依次为 PE(精度)、UE (下溢)、OE (上溢)、ZE (除 0)、DE (不规范的操作数)、IE (无效操作)。

关于 CFI(Call frame information) directives:

- <http://sourceware.org/binutils/docs/as/CFI-directives.html>

.cfi\_sections 用于描述 CFI directives 是发射.eh\_frame section, 还是.debug\_frame section。缺省为.cfi\_sections .eh\_frame

- 关于 Exception Frames (在汇编码的.eh\_frame section 中) 可参见 [http://refspecs.linuxfoundation.org/LSB\\_3.0.0/LSB-Core-generic-/LSB-Core-generic/ehframechpt.html](http://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Core-generic-/LSB-Core-generic/ehframechpt.html)
- DWARF Debugging Information Format, Version 4, June 10, 2010. <http://www.dwarfstd.org>  
6.4 Call Frame Information

术语

- **CFA(Canonical Frame Address):** 指调用者栈帧中调用点处的栈指针值 (这个值与当前栈帧入口处的栈指针值可能不相同)

## 2. C 源程序 floatarg.c

```
#include <stdio.h>
void f(a, b, c)
short a;
float b;
long c;
{
    printf("a(%p)=%d, b(%p)= %f, %x; c(%p)=%ld\n",
           &a, a, &b, b, (int)b, &c, c);
}

main()
{
    f(3, 1.0, 2);
}
```

## 3. 执行结果

a(0xffb6ec4c)=3, b(0xffb6ec40)= 1.000000, 1; c(0xffb6ec6c)=2

## 4. floatarg.c 对应的 IA32 汇编码和活动记录栈

如何产生汇编码: gcc -m32 -S floatarg.c

```
.file "floatarg.c"
.section .rodata # 只读区
.align 4
.LC0: # printf 中使用的格式串
.string "a(%p)=%d, b(%p)= %f, %x; c(%p)=%ld\n"
.text # 代码段
.globl f
.type f, @function
f:
.LFB0:
.cfi_startproc # 函数开始标识,用于初始化某些内部的数据结构
pushl %ebp # 保存调用者的栈帧基址--控制链
.cfi_def_cfa_offset 8 # 修改计算 CFA 所用的偏移, CFA 地址=偏移+已定义的寄存器
.cfi_offset 5, -8 # 寄存器 5 以前的值保存在相对于 CFA 地址偏移为-8 的位置
movl %esp, %ebp # 设置新的栈帧基址
.cfi_def_cfa_register 5 # 修改计算 CFA 所用的寄存器, 设成 5
pushl %ebx # 保存 ebx 寄存器的值
subl $84, %esp # 分配临时数据区

movl 8(%ebp), %eax # 将调用者传的第 1 个参数(long,4 字节)保存到 eax 寄存器
fldl 12(%ebp) # 将调用者传的第 2 个参数(double,8 字节)保存到 FPU 寄存器
movw %ax, -12(%ebp) # 将 eax 的低 16 位保存到-12(%ebp)开始的 2 个字节--即 a
fstps -16(%ebp) # 从 FPU 寄存器栈取 float 数存入-16(%ebp)开始的 4 字节--即 b
flds -16(%ebp) # 将-16(%ebp)的 float 数加载到 FPU 寄存器
fstps -24(%ebp) # 从 FPU 寄存器栈取 float 数存入-24(%ebp) 开始的 4 字节
movl 20(%ebp), %ebx # 取调用者传的第 3 个参数(long,4 字节)存入 ebx--形参 c
.cfi_offset 3, -12
# 将 FPU 控制字设置成 Round toward zero (truncate, 浮点数向整数值转换) 模式, 计算(int)b;
# 再将 FPU 控制字恢复到设置前的状态
flds -24(%ebp) # 加载起址为-24(%ebp)的 float 数到 FPU 寄存器--形参 b 的值
fnstcw -18(%ebp) # 将 FPU 控制字保存到-18(%ebp)开始的 2 个字节
movzwl -18(%ebp), %eax # 按零扩展方式将-18(%ebp)开始的 2 字节数存入 4 字节的 eax
movb $12, %ah # 将 12 存入 eax 的低 16 位中的高 8 位,旨在将 FPU 控制字中的
# 2 个舍入模式控制位设为 11B, 即设为 Round toward zero
movw %ax, -20(%ebp) # 将 eax 低 16 位保存的新控制字存入-20(%ebp)
fldcw -20(%ebp) # 将起址为-20(%ebp)的 2 字节数加载到 FPU 控制寄存器
# 从而 FPU 的舍入模式设置为 Round toward zero 浮点数转换成整数
fistpl -28(%ebp) # 将 ST(0)寄存器的值转换成整数, 存入-28(%ebp)---(int)b
```

```

fldcw   -18(%ebp)      # 将-18(%ebp)保存的原控制字加载到 FPU 控制寄存器
                          # 即恢复到原来的 FPU 控制字
movl    -28(%ebp), %ecx # 将-28(%ebp)开始的 4 字节存入 ecx---(int)b
flds    -24(%ebp)      # 将-24(%ebp)开始的 float 数存入 FPU 寄存器栈----形参 b 的值

```

#short 型的 a 作为实参，需提升到 long 型，保存到 edx 中

```

movzwl  -12(%ebp), %eax # 按零扩展方式将-12(%ebp)开始的 2 字节数(即形参 a)存入 eax
movswl  %ax, %edx      # 按符号扩展方式将 ax 寄存器的值存入 edx

```

# 以下处理 printf 调用的实参入栈

```

movl    $.LC0, %eax    # 将格式串的起址存入 eax
movl    %ebx, 32(%esp) # 将形参 c 的值存入 32(%esp) -- 第 8 个实参，即 c
leal    20(%ebp), %ebx # 将 20(%ebp) (即形参 c 的存储单元) 的有效地址存入 ebx
movl    %ebx, 28(%esp) # 将 ebx 的内容存入 28(%esp) -- 第 7 个实参，即&c
movl    %ecx, 24(%esp) # 将 ecx 的内容存入 24(%esp) -- 第 6 个实参，即(int)b
fstpl   16(%esp)       # 按 double 型取 ST(0)存入 16(%esp) -- 存入第 5 个实参，即 b
                          # 提升成 double 型
leal    -24(%ebp), %ecx # 将-24(%ebp)的有效地址存入 ecx，即&b
movl    %ecx, 12(%esp) # 存入 12(%esp) -- 第 4 个实参，即&b
movl    %edx, 8(%esp)  # 将 edx 的内容存入 8(%esp) -- 第 3 个实参，即 a(提升成 long)
leal    -12(%ebp), %edx # 将-12(%ebp)的有效地址存入 edx，即&a
movl    %edx, 4(%esp)  # 将 edx 的内容存入 4(%esp) ---第 2 个实参，即&a
movl    %eax, (%esp)   # 将 eax 的内容存入(%esp) ---第 1 个实参，即格式串起址$.LC0
call    printf         # 调用 printf

```

# 以下是 f 函数调用返回序列

```

addl    $84, %esp      # 回收 84 字节的空间 (局部数据和临时数据区)
popl    %ebx           # 恢复 ebx 的值
.cfi_restore 3         # 恢复 CFI 寄存器 3 的值，使之恢复到函数开始处时的值
popl    %ebp           # 恢复旧的栈帧基址 (即调用者的栈帧基址)
.cfi_def_cfa 4, 4     # 由 CFI 寄存器 4 和偏移 4 计算 CFA
.cfi_restore 5         # 恢复 CFI 寄存器 5 的值，使之恢复到函数开始处时的值
ret                                            # 返回到调用者执行
.cfi_endproc          # 函数结束标识

```

.LFE0:

```

.size   f, .-f
.globl  main
.type   main, @function

```

main:

.LFB1:

```

.cfi_startproc

```

```

pushl   %ebp           # 保存调用者的栈帧基址
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl    %esp, %ebp     # 设置当前活动记录的栈帧基址
.cfi_def_cfa_register 5
andl    $-16, %esp     # 栈顶指针按 16 字节对齐
subl    $16, %esp     # 分配 16 字节的临时数据区
movl    $2, 12(%esp)   # 2 存入 12(%esp)开始的 4 个字节--f 的第 3 个参数 2
fldl
fstpl   4(%esp)        # 从 FPU 寄存器取 double 数到 4(%esp)开始的 8 个字节
# --f 的第 2 个参数 1.0

movl    $3, (%esp)     # 3 存入(%esp)开始的 4 个字节--f 的第 1 个参数 3
call    f
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc

```

.LFE1:

```

.size   main, .-main
.ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
.section      .note.GNU-stack,"",@progbits

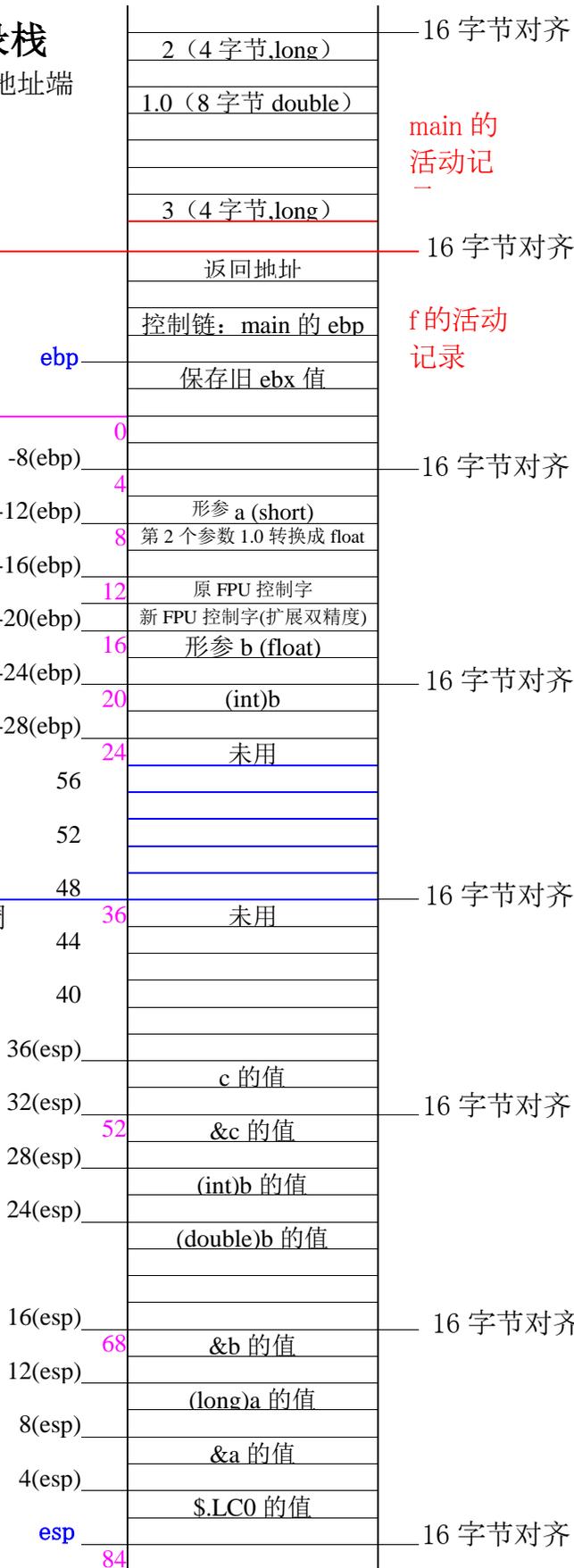
```

# 活动记录栈

高地址端

84字节的局部和临时数据区

存放 f 调用的函数的实参



84

## 5. 进一步的思考

如果将 `floatarg.c` 略做修改，即改成 `floatarg1.c`，编译产生的汇编码如下面 `floatarg1.s` 所示，试自行分析并画出活动记录栈的布局。

### `floatarg1.c`

```
#include <stdio.h>

void f(a, b, c)
short a;
float b;
long c;
{
    printf("a(%p)=%d, b(%p)= %f, %x; c(%p)=%ld\n",
           &a, a, &b, b, (int)b, &c, c);
}

main()
{
    short a = 3;
    float b = 1.0;
    long c = 2;
    f(a, b, c);
}
```

### `floatarg1.s`

```
.file "floatarg1.c"
.section .rodata
.align 4
.LC0:
.string "a(%p)=%d, b(%p)= %f, %x; c(%p)=%ld\n"
.text
.globl f
.type f, @function

f:
.LFB0:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
```

```
.cfi_offset 5, -8
movl    %esp, %ebp
.cfi_def_cfa_register 5
pushl   %ebx
subl    $68, %esp
movl    8(%ebp), %eax
movw    %ax, -12(%ebp)
movl    16(%ebp), %ebx
.cfi_offset 3, -12
flds    12(%ebp)
fstcw   -10(%ebp)
movzwl  -10(%ebp), %eax
movb    $12, %ah
movw    %ax, -14(%ebp)
fldcw   -14(%ebp)
fistpl  -20(%ebp)
fldcw   -10(%ebp)
movl    -20(%ebp), %ecx
flds    12(%ebp)
movzwl  -12(%ebp), %eax
movswl  %ax, %edx
movl    $.LC0, %eax
movl    %ebx, 32(%esp)
leal    16(%ebp), %ebx
movl    %ebx, 28(%esp)
movl    %ecx, 24(%esp)
fstpl   16(%esp)
leal    12(%ebp), %ecx
movl    %ecx, 12(%esp)
movl    %edx, 8(%esp)
leal    -12(%ebp), %edx
movl    %edx, 4(%esp)
```

```
movl    %eax, (%esp)
```

```
call   printf
```

```
addl   $68, %esp
```

```
popl   %ebx
```

```
.cfi_restore 3
```

```
popl   %ebp
```

```
.cfi_def_cfa 4, 4
```

```
.cfi_restore 5
```

```
ret
```

```
.cfi_endproc
```

```
.LFE0:
```

```
.size   f, .-f
```

```
.globl  main
```

```
.type   main, @function
```

```
main:
```

```
.LFB1:
```

```
.cfi_startproc
```

```
pushl  %ebp
```

```
.cfi_def_cfa_offset 8
```

```
.cfi_offset 5, -8
```

```
movl   %esp, %ebp
```

```
.cfi_def_cfa_register 5
```

```
andl   $-16, %esp
```

```
subl   $32, %esp
```

```
movw   $3, 30(%esp)
```

```
movl   $0x3f800000, %eax
```

```
movl   %eax, 20(%esp)
```

```
movl   $2, 24(%esp)
```

```
movswl 30(%esp), %eax
```

```
movl   24(%esp), %edx
```

```
movl   %edx, 8(%esp)
```

```
movl   20(%esp), %edx
```

```
movl    %edx, 4(%esp)
```

```
movl    %eax, (%esp)
```

```
call    f
```

```
leave
```

```
.cfi_restore 5
```

```
.cfi_def_cfa 4, 4
```

```
ret
```

```
.cfi_endproc
```

```
.LFE1:
```

```
.size   main, .-main
```

```
.ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
```

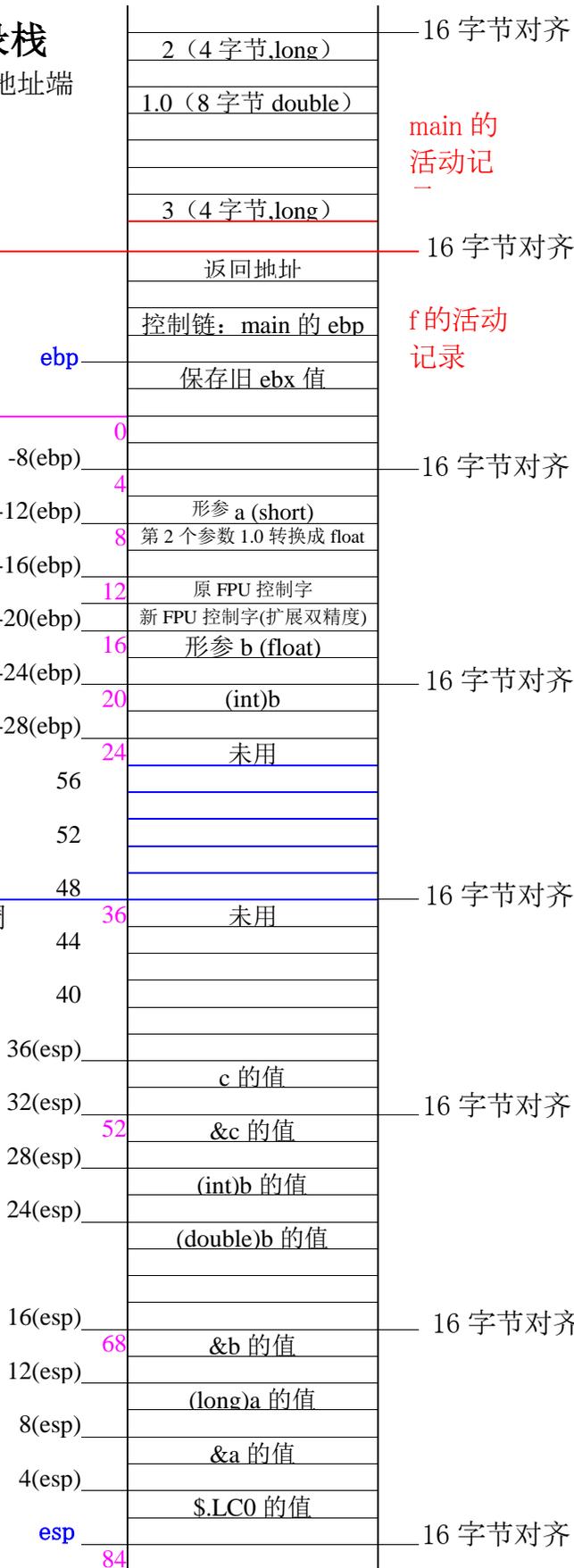
```
.section      .note.GNU-stack,"",@progbits
```

# 活动记录栈

高地址端

84字节的局部和临时数据区

存放 f 调用的函数的实参



84