x86-64体系下一个奇怪问题的定位

作者 张洋 | 发布于 2012-11-13

CPU 浮点数 汇编 x86

问题来源于一个朋友在百度的笔试题。上周六我一个朋友参加了百度举行的专场招聘会，其中第一道笔试题是这样的：

给出下面一段代码

```
1.  #include <stdio.h>
2.  main() {
3.      double a = 10;
4.      printf("a = %d\n", a);
5.  }
```

请问代码的运行结果以及原因。

当朋友参加完笔试和我聊起这道题时，我第一反应是这道题考察的是浮点数的内存表示，当然，在不同的CPU体系下，运行结果可能会有所不同，主要是受CPU位数和字节序的影响。

最初分析

不妨以目前最普遍的x86-64体系（64位，小端序）考虑此问题。在64位机器上，double是符合IEEE754标准的双精度浮点数。根据IEEE标准，双精度浮点数由8个字节共64位组成，其中最高位为符号位，次高的11位为指数位，余下的52位为尾数位。示意见下图：

各位段意义如下：

S = 0表示正数，S = 1表示负数。

E可以看成一个无符号整数，当其二进制位为全0或全1时，表示非规约浮点数或特殊值，此处不讨论，仅讨论其不全为0或全为1的情况。当E不全为零或全为1时，浮点数是规约的，此时E表示以2为底的指数加上一个固定的偏移量。偏移量被定义为$2^{(E)-1}-1$，其中(E)表示E所占的比特数，此处为11，所以偏移量为$2^{(11)-1}-1=1023$。因此实际的指数值要在E的基础上减1023，例如E的位表示是10000000000（十进制1024），则表示实际指数值为$1024-1023=1$。

M在规约形式下，表示一个二进制小数，实际值是这个小数加1。例如，M=101000…0表示$2^{-1}+2^{-3}+1=1.625$。

一个规约的IEEE双精度浮点数的实际值为：

$$V=(-1)^S\times 2^{E-1023}\times(1+M)$$

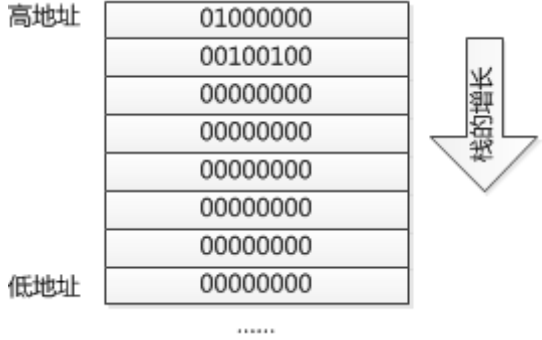
根据以上分析，10可以表示为1.25×8，因此取S = 0，E = 10000000010，M = 0100…0，则整个浮点数的二进制表示为：

01000000, 00100100, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000

为了便于观察我在每8bit之间插入了分隔符。当printf使用“%d”输出时，由于int类型是4字节，所以只能取其中四个字节。当a被当做参数传递给printf时，有两种可能保存a的地方：寄存器或栈帧中。

如果是寄存器，则printf会取低四字节。

如果是栈，在小端序中，高字节存放在高地址，低字节放在低地址，而栈是从高地址向低地址增长的，所以入栈后每个字节的位置如下：



printf会从低地址到高地址读取4个字节当做int型数据去解释并输出，所以，经过分析这段代码的输出应该为“a = 0”。

奇怪的结果

分析完了，下一步当然是通过实践验证，我在我的VPS上（CentOS 64位）用gcc编译。结果非常出乎意料，不但不是0，而且每次运行的结果都不一样！（见下图）

```
[zhangyang@ALIYUN c]$ ./double_as_int.64.o
a = -664910984
[zhangyang@ALIYUN c]$ ./double_as_int.64.o
a = 1562013592
[zhangyang@ALIYUN c]$ ./double_as_int.64.o
a = 2075757944
[zhangyang@ALIYUN c]$ ./double_as_int.64.o
a = 1017263688
[zhangyang@ALIYUN c]$ ./double_as_int.64.o
a = 580864584
[zhangyang@ALIYUN c]$ ./double_as_int.64.o
a = -1409325224
[zhangyang@ALIYUN c]$ ./double_as_int.64.o
a = 466891064
```

定位问题

在试图解释这个奇怪现象时，我最初从C的层面上进行了诸多分析，结果都无法分析出问题所在，所以我怀疑出现这个问题的原因在机器代码层面。于是我将其汇编代码打出来：

```
  .file   "double_as_int.c"
  .section    .rodata.str1.1,"aMS",@progbits,1
  .LC1:
  .string "a = %d\n"
  .text
  .globl main
  .type   main, @function
 main:
  .LFB11:
  .cfi_startproc
  subq    $8, %rsp
  .cfi_def_cfa_offset 16
  movsd   .LC0(%rip), %xmm0
  movl    $.LC1, %edi
  movl    $1, %eax
  call    printf
  addq    $8, %rsp
  .cfi_def_cfa_offset 8
  ret
  .cfi_endproc
  .LFE11:
  .size   main, .-main
  .section    .rodata.cst8,"aM",@progbits,8
  .align 8
  .LC0:
  .long   0
  .long   1076101120
  .ident  "GCC: (GNU) 4.4.6 20120305 (Red Hat 4.4.6-4)"
  .section    .note.GNU-stack,"",@progbits
```

为了方便对比，我重新写了下面的C代码：

```
#include <stdio.h>

main() {
    int a = 10;
    printf("a = %d\n", a);
}
```

其汇编为：

```
1.  .file   "int_as_int.c"
2.  .section    .rodata.str1.1,"aMS",@progbits,1
3.  .LC0:
4.  .string "a = %d\n"
5.  .text
6.  .globl main
7.  .type   main, @function
8.  main:
9.  .LFB11:
10. .cfi_startproc
11. subq    $8, %rsp
12. .cfi_def_cfa_offset 16
13. movl    $10, %esi
14. movl    $.LC0, %edi
15. movl    $0, %eax
16. call    printf
17. addq    $8, %rsp
18. .cfi_def_cfa_offset 8
19. ret
20. .cfi_endproc
21. .LFE11:
22. .size   main, .-main
23. .ident  "GCC: (GNU) 4.4.6 20120305 (Red Hat 4.4.6-4)"
24. .section    .note.GNU-stack,"",@progbits
```

将注意力集中在main函数中调用printf之前的行为，可以看到，在第一段代码中，LC0有个常数1076101120，将其转换为二进制刚好是我们上面分析的双精度10的二进制表示，而汇编代码将这个数送入了一个叫xmm0寄存器。通过查阅x86-64处理器的相关资料，知道这个寄存器和SIMD（单指令多数据流）扩展指令集有关。简单来说，在64位操作系统下，x86-64通过SIMD机制提高浮点运算能力，所以double类型的a被送入了xmm0（SIMD会用到8个128bit寄存器，xmm0 - xmm7）。

对比一下第二段代码，当a被声明是int类型时，立即数10被送入了esi（一个通用寄存器，在64位CPU中表示rsi的低32位）。其它部分似乎没有区别。

通过对比，我猜测64位操作系统下由于启用了SIMD，浮点数会被送入mmx寄存器，而整形会被送入通用寄存器。为了证实我的想法，我查阅了x86-64的ABI文档，在“3.2.3 Parameter Passing”一小节找到了如下的文字：

> **INTEGER** This class consists of integral types that fit into one of the general purpose registers.
>
> **SSE** The class consists of types that fit into a vector register.

这段话和相关汇编代码基本印证了我的猜测。为了进一步验证，我考虑手工改一下汇编代码，将movsd .LC0(%rip), %xmm0改为将数据送入rsi（其低32位就是esi），修改后代码如下，注意第13行代码是我修改过的：

```
1.  .file   "double_as_int.c"
2.  .section    .rodata.str1.1,"aMS",@progbits,1
3.  .LC1:
4.  .string "a = %d\n"
5.  .text
6.  .globl main
7.  .type   main, @function
8.  main:
9.  .LFB11:
10. .cfi_startproc
11. subq    $8, %rsp
12. .cfi_def_cfa_offset 16
13. movq    .LC0(%rip), %rsi
14. movl    $.LC1, %edi
15. movl    $1, %eax
16. call    printf
17. addq    $8, %rsp
18. .cfi_def_cfa_offset 8
19. ret
20. .cfi_endproc
21. .LFE11:
22. .size   main, .-main
23. .section    .rodata.cst8,"aM",@progbits,8
24. .align 8
25. .LC0:
26. .long   0
27. .long   1076101120
28. .ident  "GCC: (GNU) 4.4.6 20120305 (Red Hat 4.4.6-4)"
29. .section    .note.GNU-stack,"",@progbits
```

编译这段汇编代码执行，果然结果固定为0：

```
[zhangyang@ALIYUN c]$ ./a.out
a = 0
[zhangyang@ALIYUN c]$ ./a.out
a = 0
[zhangyang@ALIYUN c]$ ./a.out
a = 0
[zhangyang@ALIYUN c]$ ./a.out
a = 0
[zhangyang@ALIYUN c]$ ./a.out
a = 0
[zhangyang@ALIYUN c]$ ./a.out
a = 0
```

最后，我用-m32指令编译成32位代码，结果也固定为0，并且汇编代码中没有看到mmx相关寄存器的使用。然后，我手工用movl将12345送入esi，结果为输出总为12345，证明printf默认认为第一个int参数放在esi中。至此问题原因基本确定。

总结

从上述过程知道，最初的笔试代码，在64位环境下，浮点数参数被送入mmx寄存器，而%d告诉printf第一个参数为int类型，所以printf仍然去默认的esi中寻找第一个int参数，所以从esi中读取了一个未确定的32bit数据并按int解释，最终造成结果的不确定。

所以这道题的正确答案（小端序）是，在32位下，输出为“a = 0”；在64位启用SIMD情况下，输出结果不确定。

特别需要说明的是，由于汇编代码在不同CPU、不同操作系统、不同gcc选项下可能会有差异，所以你得到的汇编代码未必和我的相同，但原因是一致的：64位环境下int和double放置的位置不同，double告诉a放到一个地方，而%d告诉printf到另一个地方取数据，结果自然无法取到变量a。

由此也可以看出，printf最好不要将占位符和实际参数设为不同的类型，因为这样会造成不可预料的结果。

参考文献

System V Application Binary Interface AMD64 Architecture Processor Supplement

IEEE 754: Standard for Binary Floating-Point Arithmetic

x86 Assembly Guide