x87浮点处理指令

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

开场白

前面几节课主要介绍了**IA-32**指令系统中各种类型的指令，包括传送指令、定点算术运算指令、位操作指令和控制转移指令。

本节课主要介绍**x87 FPU** 浮点处理指令。

同样，所有**IA-32**指令的细节内容不需要记忆，只要用到某条指令时，会查手册并理解手册中所描述的内容。

IA-32的浮点处理架构

- IA-32的浮点处理架构有两种

  (1) x87FPU指令集（gcc默认）

  (2) SSE指令集（x86-64架构所用）

- IA-32中处理的浮点数有三种类型
  - float类型：32位 IEEE 754 单精度格式
  - double类型：64位 IEEE 754 双精度格式
  - long double类型：80位双精度扩展格式

    1位符号位s、15位阶码e（偏置常数为16 383）、1位显式首位有效位（explicit leading significant bit）j 和 63位尾数f。它与IEEE 754单精度和双精度浮点格式的一个重要的区别是：它没有隐藏位，有效位数共64位。

x87 FPU指令

- 早期的浮点处理器是作为CPU的外置协处理器出现的
- x87 FPU 特指与x86处理器配套的浮点协处理器架构
  - 浮点寄存器采用栈结构
    - 深度为8，宽度为80位，即8个80位寄存器 SKIP
    - 名称为 ST(0) ~ ST(7)，栈顶为ST(0)，编号分别为 0~7
  - 所有浮点运算都按80位扩展精度进行
  - 浮点数在浮点寄存器和内存之间传送
    - float、double、long double型变量在内存分别用IEEE 754单精度、双精度和扩展精度表示，分别占32位（4B）、64位（8B）和96位（12B，其中高16位无意义）
    - float、double、long double类型变量在浮点寄存器中都用80位扩展精度表示
    - 从浮点寄存器到内存：80位扩展精度格式转换为32位或64位
    - 从内存到浮点寄存器： 32位或64位格式转换为80位扩展精度格式

Intel处理器

x86前产品	4004 • 4040 • 8008 • 8080 • iAPX 432 • 8085
x87（外置浮点运算器）	8/16位总线: 8087 16位总线: 80187 • 80287 • 80387SX 32位总线: 80387DX • 80487
x86-16（16位）	8086 • 8088 • 80186 • 80188 • 80286
x86-32/IA-32（32位）	80386 • 80486 • Pentium（OverDrive、Pro、II、III、4、M） • Celeron（M、D） • Core
x86-64/Intel 64（64位）	Pentium（4（部份型号）、Pentium D、EE） • Celeron D（部份型号） • Core 2
EPIC/IA-64（64位）	Itanium
RISC	i860 • i960 • StrongARM • XScale
微控制器	8048 • 8051 • MCS-96
x86-32/IA-32	EP80579 • A100 • Atom（CE、SoC）
x86-64/Intel 64	Xeon（E3、E5、E7、Phi） • Atom（部分型号） • Celeron • Pentium • Core（i3、i5、i7）
EPIC/IA-64	Itanium 2

已停产

现有产品

BACK

X87 FPU指令

- **数据传送类**

(1) 装入（转换为80位扩展精度）

FLD：将数据从存储单元装入浮点寄存器栈顶 ST(0)

FILD：将数据从int型转换为浮点格式后，装入浮点寄存器栈顶

(2) 存储（转换为IEEE 754单精度或双精度）

FSTx：x为s/l时，将栈顶ST(0)转换为单/双精度格式，然后存入存储单元

FSTPx：弹出栈顶元素，并完成与FSTx相同的功能

FISTx：将栈顶数据从int型转换为浮点格式后，存入存储单元

FISTP：弹出栈顶元素，并完成与FISTx相同的功能

带P结尾指令表示操作数会出栈，也即ST(1)将变成ST(0)

X87 FPU指令

- **数据传送类**

(3) 交换

FXCH：交换栈顶和次栈顶两元素

(4) 常数装载到栈顶

FLD1 ：装入常数1.0

FLDZ ：装入常数0.0

FLDPI ：装入常数pi (=3.1415926...)

FLDL2E ：装入常数log(2)e

FLDL2T ：装入常数log(2)10

FLDLG2 ：装入常数log(10)2

FLDLN2 ：装入常数Log(e)2

X87 FPU指令

- **算术运算类**

(1) 加法

FADD/FADDP： 相加 / 相加后弹出栈

FIADD：按int型转换后相加

(2) 减法

FSUB/FSUBP ： 相减 / 相减后弹出栈

FSUBR/FSUBRP：调换次序相减 / 相减后弹出栈

FISUB：按int型转换后相减

FISUBR：按int型转换并调换次序相减

若指令未带操作数，则默认操作数为ST(0）、ST(1）

带R后缀指令是指操作数顺序变反，例如：

fsub执行的是x-y，fsubr执行的就是y-x

X87 FPU指令

- **算术运算类**

(3) 乘法

FMUL/FMULP: 相乘/相乘后弹出栈

FIMUL：按int型转换后相乘

(4) 除法

FDIV/FDIVP：相除/相除后弹出栈

FIDIV：按int型转换后相除

FDIVR/FDIVRP：调换次序相除 / 相减后弹出栈

FIDIVR：按int型转换并调换次序相除

IA-32浮点操作举例

问题：使用老版本gcc –O2编译时，程序一输出0，程序二输出是1，是什么原因造成的？ **f(10)的值是多少？机器数是多少？**

程序一：

```
#include <stdio.h>
double f(int x) {
        return 1.0 / x ;
}
void main() {
        double a, b;
        int i ;
        a = f(10) ;
        b = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

程序二：

```
#include <stdio.h>
double f(int x) {
        return 1.0 / x ;
}
void main() {
        double a, b, c;
        int i ;
        a = f(10) ;
        b = f(10) ;
        c = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

IA-32浮点操作举例

```
double f(int x)
{
    return 1.0 / x ;
}
```

```
8048328:    55          push   %ebp
8048329:    89 e5       mov    %esp,%ebp
804832b:    d9 e8       fld1
804832d:    da 75 08    fidivl 0x8(%ebp)
8048330:    c9          leave
8048331:    c3          ret
```

两条重要指令的功能如下

入口参数：int x=10

fld1：将常数1.0压入栈顶ST(0)

fidivl：将指定存储单元操作数M[R[ebp]+8]中的int型数转换为double型，再将ST(0)除以该数，并将结果存入ST(0)中

f(10)=1.0(80位扩展精度)/10(转换为double)=0.1

0.1=0.00011[0011]B= 0.00011 0011 0011 0011 0011 0011 0011…B

IA-32浮点操作举例

```
...
a = f(10) ;
b = f(10) ;
i = a == b;
...
```

```
08048334 <main>:
 8048334:      55                    push   %ebp
 8048335:      89 e5                 mov    %esp,%ebp
 8048337:      83 ec 08              sub    $0x8,%esp
 804833a:      83 e4 f0              and    $0xfffffff0,%esp
 804833d:      83 ec 0c              sub    $0xc,%esp
 8048340:      6a 0a                 push   $0xa
 8048342:      e8 e1 ff ff ff        call   8048328 <f> //计算a=f(10)
 8048347:      dd 5d f8              fstpl  0xfffffff8(%ebp) //a存入内存
 804834a:      c7 04 24 0a 00 00 00  movl   $0xa,(%esp,1)
 8048351:      e8 d2 ff ff ff        call   8048328 <f> //计算b=f(10)
 8048356:      dd 45 f8              fldl   0xfffffff8(%ebp) //a入栈顶
 8048359:      58                    pop    %eax
 804835a:      da e9                 fucompp      //比较ST(0)a和ST(1)b
 804835c:      df e0                 fnstsw %ax  //把FPU状态字送到AX
 804835e:      80 e4 45              and    $0x45,%ah
 8048361:      80 fc 40              cmp    $0x40,%ah
 8048364:      0f 94 c0              sete   %al
 8048367:      5a                    pop    %edx
 8048368:      0f b6 c0              movzbl %al,%eax
 804836b:      50                    push   %eax
 804836c:      68 d8 83 04 08        push   $0x80483d8
 8048371:      e8 f2 fe ff ff        call   8048268 <_init+0x38>
 8048376:      c9                    leave
 8048377:      c3                    ret
```

80位→64位

64位→80位

0.1是无限循环小数，无法精确表示，比较时，a舍入过而b没有舍入过，故 a≠b

IA-32浮点操作举例

```
...
a = f(10) ;
b = f(10) ;
c = f(10) ;
i = a == b;
...
```

```
8048342:    e8 e1 ff ff ff    call   8048328 <f> //计算a
8048347:    dd 5d f8          fstpl  0xfffffff8(%ebp) //把a存回内存
                                     //a产生精度损失
804834a:    c7 04 24 0a 00 00 00   movl   $0xa,(%esp,1)
8048351:    e8 d2 ff ff ff    call   8048328 <f> //计算b
8048356:    dd 5d f0          fstpl  0xfffffff0(%ebp) //把b存回内存
                                     //b产生精度损失
8048359:    c7 04 24 0a 00 00 00   movl   $0xa,(%esp,1)
8048360:    e8 c3 ff ff ff    call   8048328 <f> //计算c
8048365:    dd d8             fstp   %st(0)
8048367:    dd 45 f8          fldl   0xfffffff8(%ebp) //从内存中载入a
804836a:    dd 45 f0          fldl   0xfffffff0(%ebp) //从内存中载入b
804836d:    d9 c9             fxch   %st(1)
804836f:    58                pop    %eax
8048370:    da e9             fucompp //比较a，b
8048372:    df e0             fnstsw %ax
```

0.1是无限循环小数，无法精确表示，比较时，a和b都是舍入过的，故 a=b！

IA-32浮点操作举例

- 从这个例子可以看出
  - 编译器的设计和硬件结构紧密相关。
  - 对于编译器设计者来说，只有真正了解底层硬件结构和真正理解指令集体系结构，才能够翻译出没有错误的目标代码，并为程序员完全屏蔽掉硬件实现的细节，方便应用程序员开发出可靠的程序。
  - 对于应用程序开发者来说，也只有真正了解底层硬件的结构，才有能力编制出高效的程序，能够快速定位出错的地方，并对程序的行为作出正确的判断。

IA-32浮点操作举例

C/C++ code

```
1  #include "stdafx.h"
2  int main(int argc, char* argv[])
3  {
4      int a=10;
5      double *p=(double*)&a;
6      printf("%f\n",*p);              //结果为0.000000
7      printf("%f\n",(double(a)));     //结果为10.000000
8
9      return 0;
10 }
11 为什么printf("%f",*p)和printf("%f",(double)a)结果不一样呢？
```

不都是强制类型转换吗？怎么会不一样

关键差别在于一条指令：

fldl 和 fildl

IA-32浮点操作举例

```
int a = 10;
 8048425:  c7 44 24 28 0a 00 00 00     movl   $0xa,0x28(%esp)
double *p = (double *)&a;
 804842d:  8d 44 24 28                 lea    0x28(%esp),%eax
 8048431:  89 44 24 2c                 mov    %eax,0x2c(%esp)
```

可以看到关于指针的类型转换在汇编层次并没有体现出来，都是直接 mov 过去

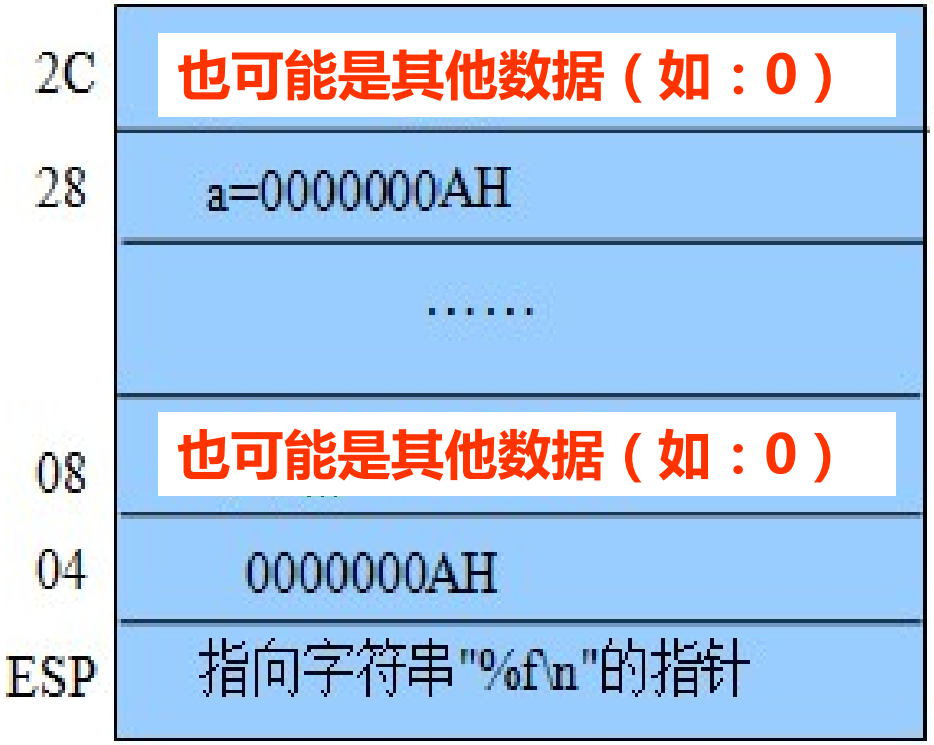
```
printf("%lf\n", *p);
                                      mov    0x2c(%esp),%eax
                                      fldl   (%eax)
```

度加载到浮点栈顶 S`0(7)）

```
                                      fstpl  0x4(%esp)
```

p 的类型是 double，故按 64 位压栈）

```
                                      movl   $0x8048500,(%esp)
                                      call   8048300 <printf@plt>
```




```
                                      mov    0x28(%esp),%eax
                                      mov    %eax,0x1c(%esp)
```

v 操作，把变量 a 的值移来移去

```
                                      fildl  0x1c(%esp)
```

的是 fildl 指令，和上面用的 fldl 指令不一样!

IA-32浮点操作举例

- **有一个跟帖的解释如下**

请问：这个帖子的回答中，哪些是正确的？哪些是错误的？

（1）

10=0000000AH，即0A是LSB，所以00H、00H、00H、0AH是printf所打印的double数据的低四字节，高四字节不确定

a是int型，内存（小端）中的表示是

0000 1010 0000 0000 0000 0000 0000 0000 后面的位不确定

double型占用8个字节，如果将上述字节看做double，第一位是符号位，第2~12位是阶码，第12~64是位数，

0000 1010 0000 0000 0000 0000 0000 0000 后面的不确定，

那么你可以转换成实数算下，应该很小（小数点的后6位肯定都是零），输出的时候默认为6位小数，发生截断，所以是

0.000000

（2）printf("%f\n",(double(a)));发生类型转化，这个可以，一般sizeof比较小的类型可以转换成size比较大的类型，或者是类型提升或者是转换