南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机系统设计实验报告

PA5：程序与性能

蒋薇

年级：2021 级

2024 年 6 月 12 日

目录

一、 实验内容

通过整数模拟实数运算

实现浮点数支持

二、 union float_

binary scaling：使用整数来模拟实数运算。

浮点数在计算机中采取科学计数法的形式进行存储：它能分为三个部分：

$$num = (-1)^s * M * 10^E$$

符号位 S：s 为 0，符号位为正；s 为 1，符号位为负

尾数 M：即一个科学计数法的小数部分

指数 E：即科学计数法表示的指数部分

实现一个用 32 位正数 FLOAT 来模拟真正的浮点数

修改 navy-apps/apps/pal/src/FLOAT/FLOAT.c 中的 f2F 函数。我们定义一个 Union 来表示浮点数，其结构包含了：有效位、指数位、符号位。

然后将二进制数转换为浮点数：1. 将真实指数减去固定移码偏移量

2. 指数位大于 7 时小数位不足，需要左移；指数位小于 7 时则小数位溢出，需右移

3. 判断符号正负，负值则左移 31 位取负值

浮点数结构体

```
FLOAT f2T(float a){
  union float_ {
struct {
uint32_t m : 23;
uint32_t e : 8;
uint32_t signal : 1;
};
uint32_t value;
};
union float_ f;
f.value = *((uint32_t*)(void*)&a);
int e = f.e - 127;
FLOAT result;
if (e <= 7) {
result = (f.m | (1 << 23)) >> 7 - e;
}
else {
result = (f.m | (1 << 23)) << (e - 7);
}
return f.signal == 0 ? result : (result|(1<<31));
}
```

三、 运算

1. 加减法：由于 FLOAT 的是用补码的方式表示，加减法直接用整数加减法表示即可

2. 而 FLOAT 的乘除法发现可以采用右移、左移 16 位的方法 (除法需要考虑符号问题)

3. 关系运算可以直接按照整数的关系运算实现

不论 int 还是 FLOAT，都是有符号的、底层存储用补码。如果左移后符号位改变，会发生错误

经过简单计算不难得出结论，如果出现类似于 eg1 的情况 (int 的第 17 位是 0，左移后由负变正)，那么这个 int 数一定小于等于-32769，这显然超过了 FLOAT 数据类型整数部分的表示范围；对于 eg2 也类似，int 数只有超过 32768 才会出现此种情况，也是 FLOAT 所不能表达的

在 pal 中，所有的数都可以用 FLOAT 表示,，可以被前 16 位正确表达；因此我们无需考虑左移后符号位改变的问题，直接左移即能将 int 转为 FLOAT

FLOAT 转 int，就是通过右移 16 位将 FLOAT 的末 16 个小数位抹去

由于 C++ 的 « 默认是算术右移，因此自带符号扩展

F_mul_F 虽然已知 ret 不会溢出，然而两个 FLOAT 类型变量相乘是会溢出的 (相当于两个正常的 int 相乘后，又左移了 32bit，可以想象如果两个 FLOAT 都是整数那么结果必然是 0)，所以需要强转为 64 位暂存；待右移 16bit 后存到 32bit 的 ret 里

必须先乘再右移，不能先让 a 或 b 右移再乘；因为右移会截断 FLOAT 的小数部分

F_div_F

两个 FLOAT 相除

不能简单作除，因为 int 型的除法只能保留到整数，然而作为 FLOAT 还需要考虑小数部分；所以实现思路要复杂一些：先计算除数，再把余数依次扩大作除法从而得到结果的小数部分

对被除数和除数取绝对值，并计算商和余数

对于余数，进行 16 次迭代，每次将余数左移 1 位试除，将商左移 1 位依次遍历到 FLOAT 每一个小数位；新的余数大于等于除数时，说明够除了，则减去除数把商加 1

最后，检查原始的被除数和除数的符号是否相同，如果相同，返回正的商，否则返回负的商。

F_nul_F F_div_F

```
//在乘法中， 两个数
相乘之后，需要右移16位，即将结果除以2^16，才能得到正确的结果
FLOAT F_mul_F(FLOAT a, FLOAT b) {
assert(-((int64_t)1 << 32) < ((int64_t) a * (int64_t) b) >> 16 &&
((int64_t) a * (int64_t) b) >> 16 < ((int64_t)1 << 32));
return ((int64_t) a * (int64_t) b) >> 16;
}

 //除法需要进行依次左移
 FLOAT F_div_F(FLOAT a, FLOAT b) {
int op = 1;
if(a < 0) {
op = -op;
a = -a;
}
if(b < 0) {
op = -op;
b = -b;
```

```
}
int ret = a / b;
a %= b;
int i;
for (i = 0;i < 16;i ++){
a <<= 1;
ret <<= 1;
if (a >= b) a -= b, ret |= 1;
}
return op * ret;
}
```

四、 INT 与 FLOAT 转换

F2intint2F

```
//F2int 的逻辑如果一个参数 a 的最高位为0，将其左移16位并返回，将低16位设为0，相当于将 a 转
换为正数的 FLOAT 类型；否则，将参数 a 取反后左移16位并返回，将低16位设为0，相当于将 a 转换为负数的 FLOAT 类型。
static inline int F2int(FLOAT a) {//浮点数转整数
if ((a & 0x80000000) == 0) {//正数
return a >> 16;//右移16位
}
else {
return -((-a) >> 16);//负数
}
}
//int2F 的逻辑与F2int相反
static inline FLOAT int2F(int a) {
if ((a & 0x80000000) == 0) {
return a << 16;
}
else {
return -((-a) << 16);
}
}
```

五、 INT 与 FLOAT 运算

这两个数据类型的运算，其实就是把 int 类型先转化为 FLOAT 类型，然后与 FLOAT 进行运算。

F2intint2F

```
//类比F\_div\_int & F\_div\_int
static inline FLOAT F_mul_int(FLOAT a, int b) {
```

```
FLOAT temp = int2F(b);
return F_mul_F(a, temp);
}
static inline FLOAT
F_div_int(FLOAT a, int b) {
FLOAT temp = int2F(b);
return F_div_F(a, temp);
}

FLOAT Fabs(FLOAT);

FLOAT Fabs(FLOAT a)
{
return (a > 0) ? a : -a;
}
//FLOAT Fsqrt(FLOAT);
//FLOAT Fpow(FLOAT, FLOAT);
```

六、 shrd shld

SHRD 和 SHLD 双精度右移和双精度左移.

SHLD 是 Double Precision Shift Left

操作数是寄存器和立即数，目的地是内存，imm8 是无符号整数

这个指令将第一个操作数左移由第二个操作数给出的位数，然后将第三个操作数（如果存在）的最低位到第二个操作数所给定的位作为结果中的最高位插入

使用 t3 变量保存第一个操作数 id_dest

使用 rtl_shl $t3$ id_src ( ) $t3$

计算需要将 t2 的值插入到 t3 中的那些位。将 id_src2 id_src rtl_shr id_src2 $t2$

使用 rtl_or $t2$ $t3$ $t3$ $t3$

最后使用 $\mathrm{operand}_write$

使用 rtl_update_ZFSF ZF SF 0

shrdshld

```
//先在 nemu/src/cpu/exec/all-instr.h 中声明函数
make_EHelper(shrd);
make_EHelper(shld);

//补全 nemu/src/cpu/exec/exec.c
/* 0xa4 */ IDEX(I_G2E, shld), EMPTY, EMPTY, EMPTY,
/* 0xac */ IDEX(I_G2E, shrd), EMPTY, EMPTY, IDEX(E2G,imul2),

//nemu/src/cpu/exec/logic.c
make_EHelper(shld)
{
rtl_shl(&t0,&id_dest->val,&id_src->val);
rtl_li(&t2,id_src2->width);
```

```
14 rtl_shli(&t2,&t2,3);
15 rtl_subi(&t2,&t2,id_src->val);
16 rtl_shr(&t2,&id_src2->val,&t2);
17 rtl_or(&t0,&t0,&t2);
18 operand_write(id_dest,&t0);
19 rtl_update_ZFSF(&t0,id_dest->width);
20 print_asm_template3(shld);
21 }
22
23 make_EHelper(shrd)
24 {
25 rtl_shr(&t0,&id_dest->val,&id_src->val);
26 rtl_li(&t2,id_src2->width);
27 rtl_shli(&t2,&t2,3);
28 rtl_subi(&t2,&t2,id_src->val);
29 rtl_shl(&t2,&id_src2->val,&t2);
30 rtl_or(&t0,&t0,&t2);
31 operand_write(id_dest, &t0);
32 rtl_update_ZFSF(&t0, id_dest->width);
33 print_asm_template3(shrd);
34 }
```

图 1: 游戏截图

七、 问题

```
PAL_InitResources success
make[1]: *** [Makefile:49: run] Floating point exception
```

图 2: 问题

查找原因 FmulF 的结果 t2 是 0，导致 div 中浮点异常

解决方法

F_mul_F 中改为 $return((uint64\_t)a * (uinit64\_t)b) >> 16$

八、 必答题

FLOAT 和 float 类型都是 32bit，都能表示 232 次方个不同的数，但由于表示方法的差异，二者表示的数集合也不一样，思考用 FLOAT 模拟表示 float 隐含着哪些取舍?

表示数的集合的差异 float：浮点数分布不均匀，在较小的值域上，浮点数可以表示更多的小数位，而在较大的值域上，浮点数可能只能表示较少的小数位；可以理解为有效数字一共就 1+23 位，整数部分大，就会挤占小数部分的位数；从范围上来看，较大

FLOAT：浮点数分布均匀，较为平坦；因为整数和小数的位数是固定的；从范围上来看，较小

更简单的实现：FLOAT 本质是 int，因此 FLOAT 的运算就不需要 NEMU 额外添加对 FLOAT 的支持，例如要在 NEMU 中实现浮点寄存器、浮点运算器及相关指令等等

更高的运算效率：因为小数点所在位置固定，因此简单的整型指令组合就能实现 FLOAT 的加减乘除运算；而对于 float，往往需要数倍于 int 类指令的时钟周期来运算；因此 FLOAT 的效率更高