目录

目录

浮点数相关实现

知识准备

FLOAT

按照实验指导书说的，我们此处用 FLOAT 来记录浮点型，其格式如下：

```
31  30                  16                   0
+----+-------------------+--------------------+
|sign|      integer      |      fraction      |
+----+-------------------+--------------------+
```

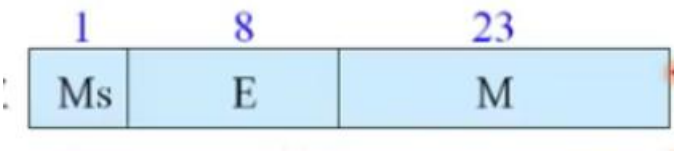
我们约定最高位为符号位，接下来的 15 位表示整数部分，低 16 位表示小数部分，即约定小数点在第 15 和第 16 位之间(从第 0 位开始)．从这个约定可以看到，FLOAT 类型其实是实数的一种定点表示．
这样，对于一个实数 a，它的 FLOAT 类型表示 A = a * 2^16(截断结果的小数部分)．例如实数 1.2 和 5.6 用 FLOAT 类

型来近似表示，就是

```
1.2 * 2^16 = 78643 = 0x13333
+----+-------------------+--------------------+
| 0  |         1         |        3333        |
+----+-------------------+--------------------+


5.6 * 2^16 = 367001 = 0x59999
+----+-------------------+--------------------+
| 0  |         5         |        9999        |
+----+-------------------+--------------------+
```

float

单精度浮点数 float 的存储方式如下



其中 Ms 为符号位
E 为指数为
M 为尾数部分
他的数值的绝对值为(1.M)*2^E
不过 E 的取值范围是 0 到 255，但实际肯定要存小于 1 的数，指数应该可以是复数，所以规定存的时候 E 存的是指数值+中值(127)，然后算的时候-127
所以其绝对值为(1.M)*2^(E-127)
此外还有特殊情况，当 E 为全 0 的时候，没有+1,值直接是(0.M)*2^E
当 E 全为 1 的时候为 M 全为 0 为正负无穷，不是则是 NaN

实现 FLOAT 相关函数

按照实验指导书说的，实现相关函数即可战斗

> **实现binary scaling**
>
> 实现上述函数来在仙剑奇侠传中对浮点数操作进行模拟. 实现正确后, 你就可以在仙剑奇侠传中成功进行战斗了.

然后指导书说仙剑里头的浮点数都能用 FLOAT 表示出来不用关心溢出，所以我直接是一个偷懒，没做溢出检查

成 FLOAT 类型再进行运算.

事实上, 我们并没有考虑计算结果溢出的情况, 不过仙剑奇侠传中的浮点数结果都可以在 FLOAT 类型中表示, 所以你可以不关心溢出的问题. 如果你不放心, 你可以在上述函数的实现中插入assertion来捕捉溢出错误.

F2int

没啥可说的，通过右移 16 位取整即可，注意对符号位处理

```
static inline int F2int(FLOAT a) {
  int sig=a&0x80000000;
    a=a&0x7fffffff;
    a=a>>16;
    a=a|sig;
    return a;
}
```

int2F

和上面一样，左移 16 位即可，仍是注意一下符号位的处理即可

```
static inline FLOAT int2F(int a) {
  int sig=a&0x80000000;
  a=a&0x7fffffff;
  a=a<<16;
  a=a|sig;
  return a;

}
```

F_mul_int/F_div_int

直接乘/除就行了，同样也是注意符号位的处理

```
static inline FLOAT F_mul_int(FLOAT a, int b) {
  int sig=(a&0x80000000)^(b&0x80000000);
  a=a&0x7fffffff,b=b&0x7fffffff;
  a=a*b;
  a=a|sig;
  return a;

}
```

```
static inline FLOAT F_div_int(FLOAT a, int b) {
int sig=(a&0x80000000)^(b&0x80000000);
a=a&0x7fffffff,b=b&0x7fffffff;
a=a/b;
a=a|sig;
return a;
}
```

F_mul_F

先把符号处理一下，然后直接扩展到 64 位 int(俩 32 位乘不会超过 64 位)乘法，最后右移 16 位（因为 FLOAT 的低 16 位是小数部分，俩 FLOAT 按 int 乘的结果相当于多挪了 16 位，再挪回去就行），然后仍然是恢复符号即可

```
FLOAT F_mul_F(FLOAT a, FLOAT b) {
  int sig=(a&0x80000000)^(b&0x80000000);
  a=a&0x7fffffff,b=b&0x7fffffff;
  FLOAT result=((int64_t)a*(int64_t)b)>>16;
  result=result|sig;
  return result;

}
```

F_div_F

先判断一下 b 是不是 0
然后处理符号
整数部分直接 a/b
小数部分先取余数，然后余数循环左移，看看能不能比 b 大，比 b 大就-b 然后结果对应位置置 1 即可
最后恢复符号

```
FLOAT F_div_F(FLOAT a, FLOAT b) {
  assert(b!=0);
  int sig=(a&0x80000000)^(b&0x80000000);
  a=a&0x7fffffff,b=b&0x7fffffff;
  int in=a/b;
  int fl=a%b;
  FLOAT res=in<<16;
  for(int i=0;i<16;i++)
  {
    fl<<=1;
    if(fl>=b)
    {
      fl-=b;
      res+=1<<(16-i-1);
    }
  }
  res=res|sig;
  return res;


}
```

f2F

首先看提示他不让直接用对 float 的计算
那我们先把他按照 uint32_t 格式读出来（注意不是强制类型转换成 uint32_t）
然后根据我们前面知识准备里 float 的存储格式，分别把他的符号位，指数位，尾数位都给他提出来
然后我们先根据指数位判断，当为 0 的时候一个小于 1 的数*2^-127，怎么地 FLOAT 都没法表示，直接视作 0 即可
然后看要是 exp==255 那就是无穷大或者是 NaN，指导书没提这情况咋弄，不过前面也说肯定能在 FLOAT 里表示，这里就给个 assert(0)了，不过跑起来之后这里也没报过错。
接下来让 m+=1<<23，这是前面 float 的规则里是 1.M 然后 M 又是 23 位而来。
接下来我们还得让这个 m>>(23-16)，为嘛呢，因为这个 float 的尾数部分他有 23 位，而 FLOAT 的小数是 16 位，得让小数部分对齐了。
然后我们根据 exp 和 127 的大小，决定是让结果左移还是右移|e-127|位
最后我们根据一开始提出来的符号位把符号一补，完事

```c
FLOAT f2F(float a) {
  /* You should figure out how to convert `a' into FLOAT without
   * introducing x87 floating point instructions. Else you can
   * not run this code in NEMU before implementing x87 floating
   * point instructions, which is contrary to our expectation.
   *
   * Hint: The bit representation of `a' is already on the
   * stack. How do you retrieve it to another variable without
   * performing arithmetic operations on it directly?
   */

  uint32_t content=*(uint32_t*)&a;
  uint32_t sig=(content&0x80000000);
  uint32_t exp=(content^sig)>>23;
  uint32_t m=(content&0x007fffff);
  if(exp==0)
  {
    return 0;
  }
  else if(exp==255)
  {
    assert(0);
  }
  else
  {
    m+=1<<23;
    m>>=(23-16);
    if(exp>=127)
    {
      m<<=(exp-127);
    }
    else
    {
      m>>=(127-exp);
    }
  }
  m=m|sig;
  return m;


}
```

Fabs

这个函数是求浮点数绝对值的。。。顺着写这个在最后一个，前面符号处理压根没用这函数。。。
因为 FLOAT 就是 int 一别名

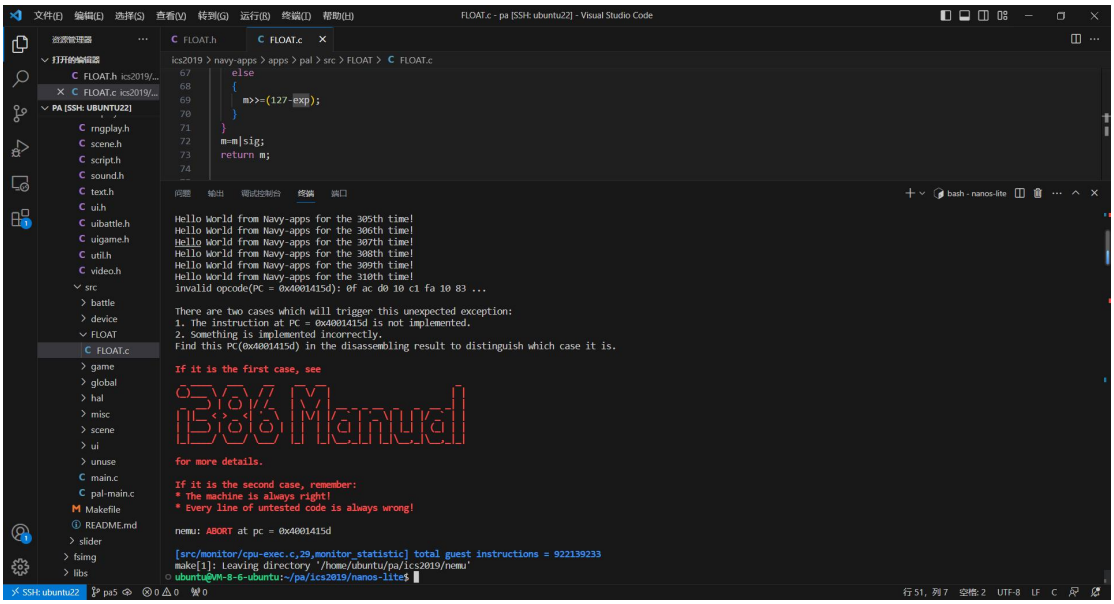
```c
5
6   typedef int FLOAT;
```

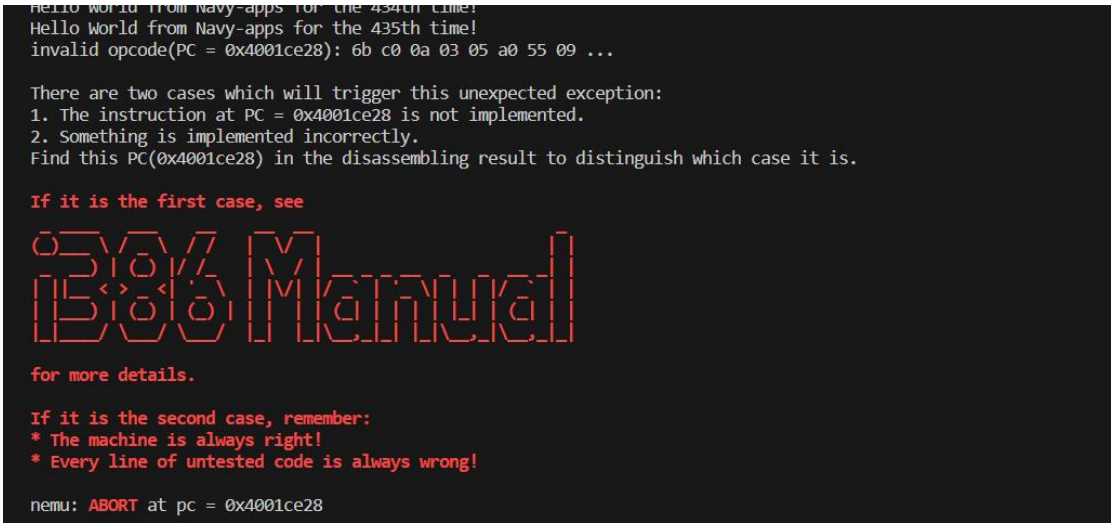
然后指导书规定的正负号和 int 也一个规则，那就直接看 a 是否大于 0，要大于就返回 a 要不然就返回-a

```
78
79  FLOAT Fabs(FLOAT a) {
80    return (a>0?a:-a);
81  }
82
```

都实现完了，make run 一下开打发现还有未实现的指令

未实现的指令





查表，实现之

```
make_EHelper(shrd);
```

```
make_EHelper(shrd)
{
  rtl_shr(&s0, &id_dest->val, &id_src->val);
  rtl_li(&s1, id_src2->width);
  rtl_shli(&s1, &s1, 3);
  rtl_sub(&s1, &s1, &id_src->val);
  rtl_shl(&s1, &id_src2->val, &s1);
  rtl_or(&s0, &s0, &s1);
  operand_write(id_dest, &s0);

  rtl_update_ZFSF(&s0, id_dest->width);

  print_asm_template3(shrd);
}
```

```
  /* 0xac */  IDEX(Ib_G2E, shrd), EMPTY, EMPTY
```

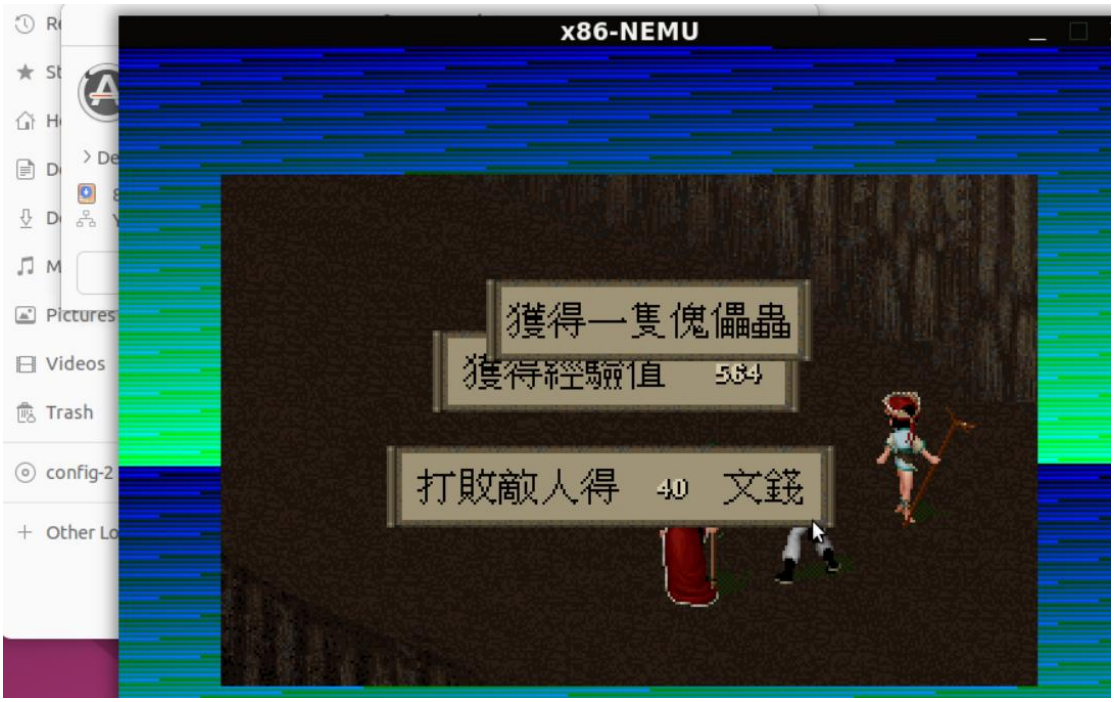
```
make_EHelper(imul3);
```

```
  // imul with three operands
  make_EHelper(imul3) {
    rtl_sext(&s0, &id_src->val, id_src->width);
    rtl_sext(&s1, &id_src2->val, id_src->width);

    rtl_imul_lo(&s0, &s1, &s0);
    operand_write(id_dest, &s0);

    print_asm_template3(imul);
  }
```

```
/* 0x68 */  IDEX(push_SI,push), IDEX(I_E2G,imul3), IDEXW(push_SI,push,1), IDEX(SI_E2G, imul3),
```

运行结果

蝎子精做掉做掉



题目

> 比较FLOAT和float
>
> `FLOAT` 和 `float` 类型的数据都是32位, 它们都可以表示2^32个不同的数. 但由于表示方法不一样, `FLOAT` 和 `float` 能表示的数集是不一样的. 思考一下, 我们用 `FLOAT` 来模拟表示 `float` , 这其中隐含着哪些取舍?

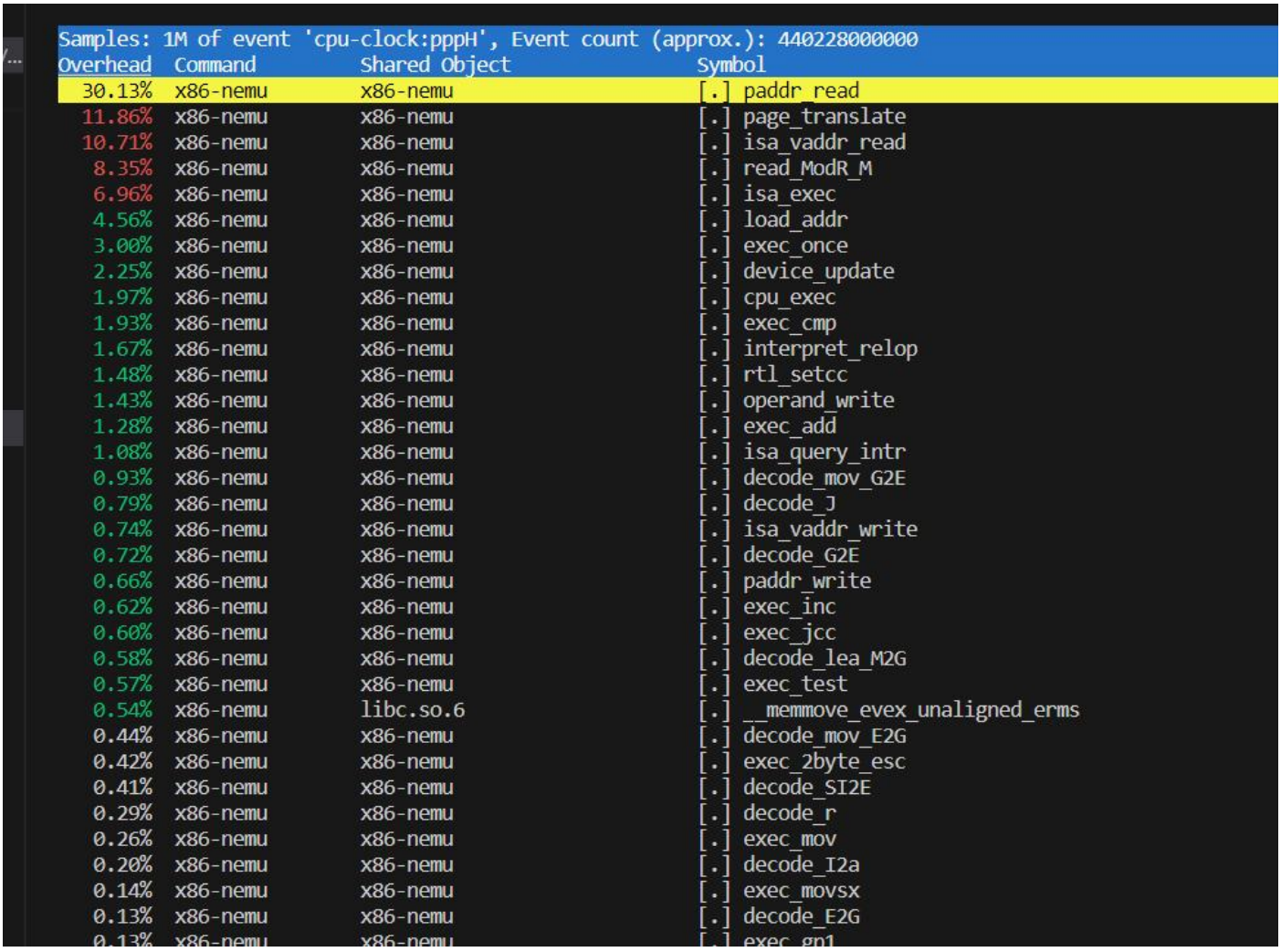
float 能表示的范围是-3.4*10(-38) ~ 3.4*10(38)， 而 FLOAT 是-32768~+32768,
+FLOAT 的特点在于他的密度是均匀的，不管啥时候，两个最近的浮点数差都是 2^-16，是均匀分布的，float 绝对

值越小的时候他的 exp 小，这个时候，他的数是很密集的，但是当我们数比较大的时候，他的数分布就比较稀疏了。FLOAT 相当于是牺牲了取值的范围，但是可以让我的数据分布十分的均匀，画一些坐标轴之类的时候或许是个不错的选择。而 float 相当于是更注重小数据时候的数的密度，我觉得他可能是认为既然你用浮点数这个类型，那说明小数部分对你来说是比较重要的，而在数越小的时候小数的占比也越大，那么他在数较小的时候让数据更密集，然后认为你都选择用我 float 类型了，那应该不会去用特别特别大的数或者说就是在这种很大的数的时候误差稍微大一点影响也会更小。

性能优化

perf 查看性能瓶颈

make 之后，用 perf 执行文件并且查看对应的分析结果，指令如下
perf record nemu/build/x86-nemu nanos-lite/build/nanos-lite-x86-nemu.bin
perf report -i perf.data
结果如下图:



```
Samples: 1M of event 'cpu-clock:pppH', Event count (approx.): 440228000000
Overhead  Command          Shared Object                Symbol
  30.13%  x86-nemu         x86-nemu                     [.] paddr_read
  11.86%  x86-nemu         x86-nemu                     [.] page_translate
  10.71%  x86-nemu         x86-nemu                     [.] isa_vaddr_read
   8.35%  x86-nemu         x86-nemu                     [.] read_ModR_M
   6.96%  x86-nemu         x86-nemu                     [.] isa_exec
   4.56%  x86-nemu         x86-nemu                     [.] load_addr
   3.00%  x86-nemu         x86-nemu                     [.] exec_once
   2.25%  x86-nemu         x86-nemu                     [.] device_update
   1.97%  x86-nemu         x86-nemu                     [.] cpu_exec
   1.93%  x86-nemu         x86-nemu                     [.] exec_cmp
   1.67%  x86-nemu         x86-nemu                     [.] interpret_relop
   1.48%  x86-nemu         x86-nemu                     [.] rtl_setcc
   1.43%  x86-nemu         x86-nemu                     [.] operand_write
   1.28%  x86-nemu         x86-nemu                     [.] exec_add
   1.08%  x86-nemu         x86-nemu                     [.] isa_query_intr
   0.93%  x86-nemu         x86-nemu                     [.] decode_mov_G2E
   0.79%  x86-nemu         x86-nemu                     [.] decode_J
   0.74%  x86-nemu         x86-nemu                     [.] isa_vaddr_write
   0.72%  x86-nemu         x86-nemu                     [.] decode_G2E
   0.66%  x86-nemu         x86-nemu                     [.] paddr_write
   0.62%  x86-nemu         x86-nemu                     [.] exec_inc
   0.60%  x86-nemu         x86-nemu                     [.] exec_jcc
   0.58%  x86-nemu         x86-nemu                     [.] decode_lea_M2G
   0.57%  x86-nemu         x86-nemu                     [.] exec_test
   0.54%  x86-nemu         libc.so.6                    [.] __memmove_evex_unaligned_erms
   0.44%  x86-nemu         x86-nemu                     [.] decode_mov_E2G
   0.42%  x86-nemu         x86-nemu                     [.] exec_2byte_esc
   0.41%  x86-nemu         x86-nemu                     [.] decode_SI2E
   0.29%  x86-nemu         x86-nemu                     [.] decode_r
   0.26%  x86-nemu         x86-nemu                     [.] exec_mov
   0.20%  x86-nemu         x86-nemu                     [.] decode_I2a
   0.14%  x86-nemu         x86-nemu                     [.] exec_movsx
   0.13%  x86-nemu         x86-nemu                     [.] decode_E2G
   0.13%  x86-nemu         x86-nemu                     [.] exec_gp1
```

主要耗时的 paddr_read,page_translate,isa_vaddr_read 等都是加载文件/上下文切换时候的开销，可以着手对这些进行优化

优化方案

摸了摸了

JIT