PA5实验报告

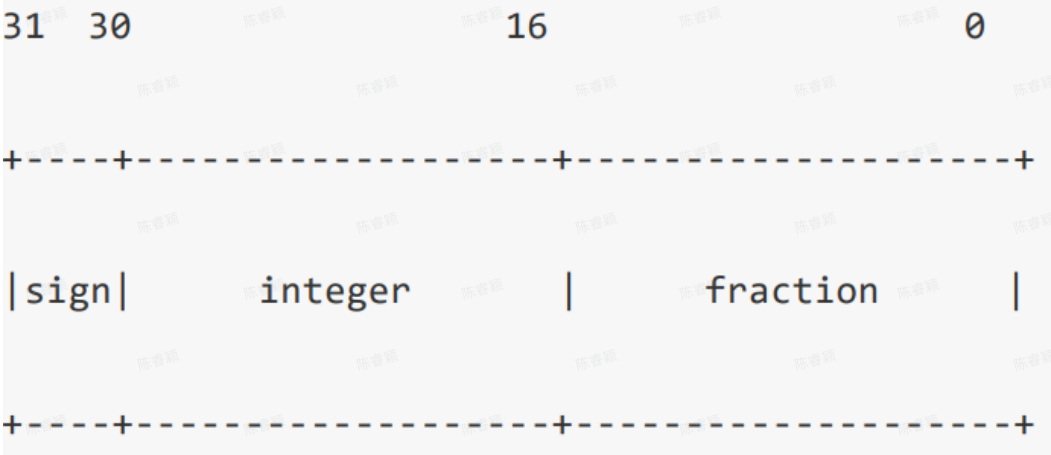
- 姓名：陈睿颖
- 学号：2013544
- 专业：计算机科学与技术

1. 实验内容

通过整数来模拟实数的运算，以实现浮点数的支持

2. 浮点数的支持

由于在 NEMU 中实现浮点数需要设计 x87 架构的很多细节，因此本实验根据 KISS 法则，通过整数来模拟实数的运算。 根据实验指导书上的说明，实验中需要实现的浮点数的结构定义如下：



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。

其实相当于定义了一个映射函数，将一个 32 位的整数映射为一个 “FLOAT” 类型的浮点数。若 “a” 表示一个 32 位的整数，则其对应的 “FLOAT” 类型的浮点数为 “A”，其映射关系如下：

$$A = a \times 2^{16}$$

由于是一个线性的映射，因此 “FLOAT” 类型的加法减法和其他运算 都可以直接运算，然而乘除法需要对结果进行转化。

仙剑奇侠传的框架代码已经用 `FLOAT` 类型对浮点数进行了相应的处理。还需要实现一些和 `FLOAT` 类型相关的函数:

```
/* navy-apps/apps/pal/include/FLOAT.h */
int32_t F2int(FLOAT a);
FLOAT int2F(int a);
FLOAT F_mul_int(FLOAT a, int b);
FLOAT F_div_int(FLOAT a, int b);
/* navy-apps/apps/pal/src/FLOAT/FLOAT.c */
FLOAT f2F(float a);
FLOAT F_mul_F(FLOAT a, FLOAT b);
FLOAT F_div_F(FLOAT a, FLOAT b);
FLOAT Fabs(FLOAT a);
```

其中 `F_mul_int()` 和 `F_div_int()` 用于计算一个 `FLOAT` 类型数据和一个整型数据的积/商, 这两种特殊情况可以快速计算出结果, 不需要将整型数据先转化成 `FLOAT` 类型再进行运算。

`FLOAT.h`

`F2int` 和 `int2F` 实现

```c
static inline int F2int(FLOAT a) {
  //assert(0);
  //return 0;
  if ((a & 0x80000000) == 0)
    return a >> 16;
  else
    return -((-a) >> 16);
}

static inline FLOAT int2F(int a) {
  //assert(0);
  //return 0;
  if ((a & 0x80000000) == 0)
    return a << 16;
  else
    return -((-a) << 16);
}
```

首先，`F2int` 通过与符号位掩码 `0x80000000` 进行位与操作，判断浮点数的符号位。如果符号位为 0，表示浮点数为正数或零，直接将 `a` 右移 16 位，即去除小数部分，得到整数结果。

如果符号位为 1，表示浮点数为负数，将 `-a` 取反后再右移 16 位，即将负数的绝对值去除小数部分，得到负整数结果。最后返回负整数。

同样，将整数转换为 `FLOAT` 的操作为上述的逆操作：

通过与符号位掩码 `0x80000000` 进行位与操作，判断整数的符号位。如果符号位为 0，表示整数为正数或零，将 `a` 左移 16 位，即将整数乘以 65536（2^16），得到转换后的 `FLOAT` 结果。

如果符号位为 1，表示整数为负数，将 `-a` 取反后再左移 16 位，即将负整数的绝对值乘以 65536，得到负的转换结果。最后返回负的转换结果。

`F_mul_int` 和 `F_div_int` 实现

```c
static inline FLOAT F_mul_int(FLOAT a, int b) {
  // assert(0);
  // return 0;
  return a*b;
}

static inline FLOAT F_div_int(FLOAT a, int b) {
  // assert(0);
  // return 0;
  return a/b;
```

```
}
```

FLOAT.c

f2F 实现

```
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?
   */

  // assert(0);
  // return 0;
  union float_ {
    struct {
      uint32_t man : 23;
      uint32_t exp : 8;
      uint32_t sign : 1;
    };
    uint32_t val;
  };
  union float_ f;
  f.val = *((uint32_t*)(void*)&a);
  int exp = f.exp - 127;
  FLOAT ret = 0;
  if (exp == 128)
    assert(0);
  if (exp >= 0) {
    int mov = 7 - exp;
    if (mov >= 0)
      ret = (f.man | (1 << 23)) >> mov;
    else
      ret = (f.man | (1 << 23)) << (-mov);
  }
  else
    return 0;
```

```
37    return f.sign == 0 ? ret : -ret;
38 }
```

通过使用联合体 `union` 将 `float` 类型的参数 `a` 转换为对应的位表示，并将其存储在一个联合体 `union float_` 的成员变量 `val` 中。联合体 `float_` 的定义使用了位域来表示浮点数的符号位、指数位和尾数位。

接下来，代码通过取出浮点数的指数位并减去偏移值 127，得到指数的实际值。根据指数的值，对 `FLOAT` 进行适当的转换。如果指数等于 128，即浮点数为 NaN 或无穷大，代码使用断言 `assert(0)` 报错。如果指数大于等于 0，表示浮点数为规格化数或大于 1 的非规格化数，根据指数值将尾数右移或左移相应的位数，并加上隐藏的尾数位 1，得到转换后的 `FLOAT`。如果指数小于 0，则表示浮点数为零或小于 1 的非规格化数，此时返回 0。

最后，根据浮点数的符号位决定返回值的正负。如果符号位为 0，则返回转换后的 `FLOAT` 值，否则返回其相反数。

`F_mul_F` 实现

```
1 FLOAT F_mul_F(FLOAT a, FLOAT b) {
2   //assert(0);
3   //return 0;
4   return ((int64_t)a * (int64_t)b) >> 16;
5 }
```

使用了位移操作和强制类型转换来执行乘法运算。首先，将参数 `a` 和 `b` 转换为 `int64_t` 类型，即 64 位有符号整数类型。然后，将两个整数相乘，得到一个 64 位的乘积。最后，通过右移 16 位来将结果转换回浮点数类型 `FLOAT`。

乘法运算的结果是一个固定点数，因此右移 16 位相当于将小数部分舍去，只保留整数部分。这样做是因为在实现中，浮点数 `FLOAT` 被表示为一个 32 位的整数，其中前 16 位表示整数部分，后 16 位表示小数部分。因此，右移 16 位可以将结果恢复为浮点数类型 `FLOAT`，并返回计算的乘积结果。

`F_div_F` 实现

```
1 FLOAT F_div_F(FLOAT a, FLOAT b) {
2   // assert(0);
3   // return 0;
```

```
  assert(b != 0);
  FLOAT x = Fabs(a);
  FLOAT y = Fabs(b);
  FLOAT ret = x / y;
  x = x % y;

  for (int i = 0; i < 16; i++) {
    x <<= 1;
    ret <<= 1;
    if (x >= y) {
      x -= y;
      ret++;
    }
  }
  if (((a ^ b) & 0x80000000) == 0x80000000) {
    ret = -ret;
  }
  return ret;
}
```

使用了循环和位移操作来执行除法运算。首先，代码通过断言语句 `assert(b != 0)` 检查除数 `b` 是否为零，如果是零，则会触发断言错误。这是为了防止除以零的情况。

接下来，代码通过调用 `Fabs` 函数计算参数 `a` 和 `b` 的绝对值，分别存储在变量 `x` 和 `y` 中。然后，通过将 `x` 除以 `y` 得到一个初始的商 `ret`，并使用取余操作 `x = x % y` 得到一个初始的余数 `x`。

接下来，使用循环进行迭代，重复进行以下操作 16 次：

1. 将余数 `x` 左移 1 位（相当于乘以 2）。
2. 将商 `ret` 左移 1 位。
3. 如果余数 `x` 大于等于除数 `y`，则减去 `y`，并将商 `ret` 加 1。

最后，代码通过与运算和比较操作判断是否需要对结果进行取负。如果原始的被除数 `a` 和除数 `b` 的符号位不同，即异或操作的结果等于 `0x80000000`，则将商 `ret` 取负。

`F_Fabs` 实现

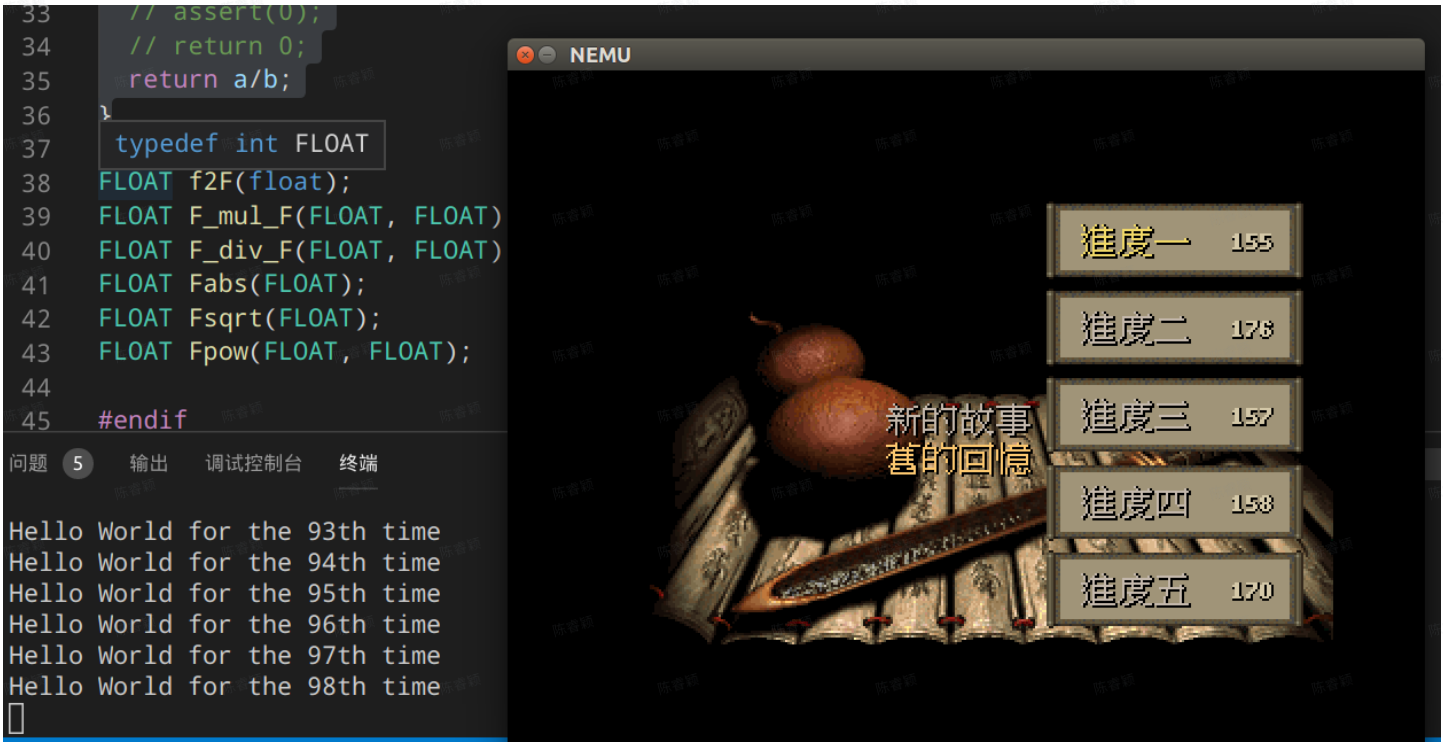
```
FLOAT Fabs(FLOAT a) {
  // assert(0);
  // return 0;
  if ((a & 0x80000000) == 0)
    return a;
```

```
    else
        return -a;
}
```

通过位运算和条件判断来确定浮点数的符号并返回其绝对值。首先，代码将参数 `a` 与掩码 `0x80000000` 进行与运算，掩码的二进制表示中只有最高位为1，其余位为0。如果结果等于0，则表示浮点数 `a` 的符号位为0，即正数或零，直接返回 `a` 本身。如果结果不等于0，则表示浮点数 `a` 的符号位为1，即负数，通过将 `a` 取反得到其绝对值并返回。

运行结果

添加完浮点数的支持后，即可运行仙剑奇侠传的战斗画面：



3. 思考题

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?

使用 `FLOAT` 来模拟表示 `float` 类型数据会涉及一些取舍，主要包括以下几个方面：

1. 精度损失： `float` 类型使用 IEEE 754 标准表示，其中包括一位符号位、八位指数位和 23 位小数位。而自定义的 `FLOAT` 类型通常使用固定点表示，其中包括符号位和固定的小数位数。由于 `FLOAT` 的小数位数较少，因此在模拟表示 `float` 时会损失一定的精度。
2. 范围限制： `float` 类型能够表示的数值范围比较广，可以表示很小的值和很大的值，同时也可以表示无穷大和 NaN（Not a Number）。而自定义的 `FLOAT` 类型可能会对表示的范围进行限制，无法覆盖 `float` 类型的所有取值范围。
3. 转换开销：在使用 `FLOAT` 来模拟表示 `float` 类型时，需要进行数据的转换和格式调整。这会引入一定的计算开销和性能损失。
4. 标准兼容性： `FLOAT` 类型是自定义的数据类型，与标准的 `float` 类型并不完全兼容。这意味着在使用 `FLOAT` 进行模拟表示时，可能无法与现有的 `float` 类型的操作和库函数进行无缝集成，可能需要自行实现相应的函数和操作。