计算机系统设计实验报告

PA5 - 从一到无穷大: 程序与性能

姓名：孟启轩
学号：2212452
专业：计算机科学与技术

- 计算机系统设计实验报告
  - PA5 - 从一到无穷大: 程序与性能
  - 一、实验目的
  - 二、实验重要内容
    - （一）如何用一个32位整数来表示一个实数?
    - （二）FLOAT的计算
    - （三）比较FLOAT和float
    - （四）需要实现的内容
  - 三、实验方法与结果
    - （一）实现f2F函数
    - （二）实现F_mul_F函数
    - （三）实现F_div_F函数
    - （四）实现Fabs函数
    - （五）实现FLOAT和int之间的相互转换以及运算
    - （六）实验结果
  - 四、遇到的BUG及解决思路
  - 五、总结与感悟

一、实验目的

（1）解决游戏战斗功能缺失问题：通过处理浮点数运算，使《仙剑奇侠传》在NEMU中能够正常进行战斗。
（2）实现浮点运算的整数模拟：采用binary scaling方法，用整数运算替代x87架构的浮点指令，避免复杂实现。
（3）验证binary scaling的可行性：通过实验验证整数模拟浮点运算的正确性和性能是否满足游戏需求。

二、实验重要内容

（一）如何用一个32位整数来表示一个实数?

我们引入binary scaling方法，使用一个 32 位整数来表示实数：

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

- **最高位（第 31 位）为符号位**，用于表示数值的正负；
- **接下来的 15 位（第 30 位到第 16 位）表示整数部分**；
- **最低的 16 位（第 15 位到第 0 位）表示小数部分**。

换句话说，我们人为地将小数点固定在第 15 和第 16 位之间（从第 0 位开始编号）。这种表示方法属于定点表示方式，与浮点数不同，它的精度和范围是固定的，不随数值大小动态变化。这种设计使得实数的表示和整数操作能够统一在整数处理指令下进行。

从FLOAT类型的定义`typedef int FLOAT;`也不难看出，其本质上是int类型。

（二）FLOAT的计算

若某个实数为 a，则其对应的 FLOAT 类型表示为 $A = a \times 2^{16}$，即将实数放大 2^{16} 倍后取整（截断小数部分）。如果需要从 FLOAT 类型值 A 恢复成近似实数，则可通过 $a \approx A \div 2^{16}$ 实现。

举个例子，把实数1.2FLOAT类型来近似表示：

1. 计算数值：只取整数部分（舍去小数），得到78643

$$1.2 \times 65536 = 78643.2$$

2. 转为十六进制：将十进制 78643 转换为十六进制

$$78643_{10} = 0x13333$$

3. 对齐 FLOAT 格式：现在的 0x13333 是

$$0001\ 0011\ 0011\ 0011\ 0011_2$$

   填充为 32 位：`00000000 00000001 00110011 00110011`

部分	二进制	十六进制
符号 + 整数	`00000000 00000001`	`0x0001`
小数部分	`00110011 00110011`	`0x3333`

```
+----+------------------+--------------------+
| 0  |         1        |        3333        |
+----+------------------+--------------------+
符号位        整数部分              小数部分
```

由于 FLOAT 类型是按比例缩放后的整数，我们可以借助整数运算完成实数的四则运算。设 $A = a \times 2^{16}$，$B = b \times 2^{16}$，则有：

- **加法**：

$$A + B = (a + b) \times 2^{16}$$

- **减法**：

$$A - B = (a - b) \times 2^{16}$$

- **乘法**（结果太大，需缩小）：

$$A \times B = \left((a \times 2^{16}) \times (b \times 2^{16})\right) \div 2^{16} = (a \times b) \times 2^{16}$$

- **除法**（结果太小，需放大）：

$$A \div B = \left((a \times 2^{16}) \div (b \times 2^{16})\right) \times 2^{16} = (a \div b) \times 2^{16}$$

- **取负数**：

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

此外，由于 FLOAT 类型本质上是一个整数，**所有的比较操作可以直接使用整数的比较运算符完成**，无需转换为浮点或实数形式。

（三）比较FLOAT和float

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` 类型：采用 IEEE 754 单精度浮点数格式

- 结构：1 位符号位 + 8 位指数 + 23 位尾数（隐含首位为 1）
- 指数偏移量：真实指数为存储值减去 127
- 表示范围：

$$[-2^{127}, 2^{127}]$$

- 优势：范围大、动态精度高，可表示非常大或非常小的实数，适合科学计算等对数值精度和范围要求高的场合。

`FLOAT` 类型：定点数格式

- 结构：1 位符号位 + 15 位整数部分 + 16 位小数部分（小数点固定在中间）
- 表示范围：

$$[-2^{16}, 2^{16}) = [-65536, 65536)$$

- 优势：表示结构简单，计算可以完全使用整数指令完成，适合硬件不支持浮点运算的场合。

`FLOAT` 模拟 `float` 的取舍分析

将 `FLOAT` 作为 `float` 的模拟，是一种在实现复杂度与表示能力之间的权衡。
取：

1. 简化实现：
   - 使用纯整数运算，无需额外的浮点运算单元，节省硬件资源。
   - 更容易在简化 CPU 模型（如 NEMU）中实现。
2. 满足需求：
   - 对于游戏逻辑、简单物理计算、UI 渲染等对数值精度和范围要求不高的场合，`FLOAT` 提供的精度和范围已足够。
   - 在实验操作系统环境中，通过 `FLOAT` 实现浮点运算逻辑，足以演示基本原理。

舍：

1. 表示范围大幅缩小：
   - `FLOAT` 只能表示大约 $\pm 6.5 \times 10^4$ 的范围，远小于 `float` 的 $\pm 3.4 \times 10^{38}$。
   - 无法表示非常大的数或非常小的非零数。
2. 精度有限且固定：
   - `FLOAT` 的小数部分最多只能表示 $2^{-16} \approx 1.5 \times 10^{-5}$ 的精度。
   - 相比之下，`float` 的动态精度可以表示更精细的数。
3. 不支持非正规数、NaN、无穷大等 IEEE 特性：
   - `FLOAT` 无法处理除以 0、溢出或 NaN情况。

（四）需要实现的内容

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 类型转 int 类型
FLOAT int2F(int a);              // int 类型转 FLOAT 类型
FLOAT F_mul_int(FLOAT a, int b); // FLOAT 类型乘 int 整数
FLOAT F_div_int(FLOAT a, int b); // FLOAT 类型除以 int 整数
/* navy-apps/apps/pal/src/FLOAT/FLOAT.c */
FLOAT f2F(float a);              // float 类型转 FLOAT 类型
FLOAT F_mul_F(FLOAT a, FLOAT b); // FLOAT 类型乘法
FLOAT F_div_F(FLOAT a, FLOAT b); // FLOAT 类型除法
FLOAT Fabs(FLOAT a);             // FLOAT 类型取绝对值运算
```

三、实验方法与结果

（一）实现f2F函数

`f2F` 函数的作用是将标准的 IEEE 754 单精度 `float` 类型转换为自定义的定点数 `FLOAT` 类型。即：

$$\text{f2F}(a) = (int)(a \times 2^{16})$$

但因为在 NEMU 环境下**不允许使用 x87 浮点运算**，所以要从 `float` 的**位表示结构**中直接解码其真实值（整数+小数），再转成定点数。

首先定义一个union `float_raw`，这样我们可以通过字段访问 `float` 类型的二进制结构。

```
typedef union {
  float value;
  struct {
    unsigned mantissa : 23;
    unsigned exponent : 8;
    unsigned sign : 1;
  };
} float_raw;
```

```
FLOAT f2F(float a) {
  float_raw bits = {a};
  if (bits.exponent == 0)
    return 0; // 零或非正规数
  if (bits.exponent == 0xff)
    return bits.sign ? 0x80000000 : 0x7fffffff; // inf 或 NaN

  int shift = bits.exponent - 127 + 16;
  if (shift < 0) //小于1，小于最小非零值
    return 0;

  int base = bits.mantissa | (1 << 23); // 补上IEEE 754隐含的1
  shift -= 23;
  int result = (shift < 0) ? (base >> -shift) : (base << shift);
  return bits.sign ? -result : result;
}
```

首先，通过联合体 `float_raw` 直接访问 `float` 数值的二进制表示，从中提取出符号位、指数位和尾数部分（mantissa）。如果指数为 0，则说明是零或非正规数，返回 0；如果指数为 255（0xff），说明是无穷大或 NaN，根据符号返回最大或最小的 `FLOAT` 值。对正常数值，先还原出尾数的完整有效位（加上隐含的 1），再根据指数和 FLOAT 的定点表示位置（小数点固定在第 16 位），计算应左移或右移的位数以对齐到 `FLOAT` 格式。最后根据符号位决定是否返回负数，从而完成转换。

（二）实现F_mul_F函数

`F_mul_F` 函数用于实现两个自定义定点数 `FLOAT` 类型的乘法运算。

首先定义一个union `Float`，这样可以将 `FLOAT` 类型的 32 位整数拆分为符号位、整数部分和小数部分，方便按位处理和模拟定点数乘法运算。

```
typedef union{
  FLOAT value;
  struct{
    unsigned frac : 16;    // 小数部分
    unsigned integer : 15; // 整数部分
    unsigned sign : 1;     // 符号位
  };
} Float;
```

```
#define FLOAT_FACTOR (1 << 16) // 2^16

FLOAT F_mul_F(FLOAT a_raw, FLOAT b_raw){
  Float a = {a_raw}, b = {b_raw};
  int result_sign = a.sign ^ b.sign;

  // 去符号，统一为正数进行乘法
  a.value = a.sign ? -a.value : a.value;
  b.value = b.sign ? -b.value : b.value;

  int int_part = a.integer * b.integer * FLOAT_FACTOR;
  int cross1 = a.frac * b.integer;
  int cross2 = a.integer * b.frac;
  int frac_mul = a.frac * b.frac;
  FLOAT frac_part = (frac_mul / FLOAT_FACTOR) + (frac_mul % FLOAT_FACTOR >= FLOAT_FACTOR / 2);

  FLOAT result = int_part + cross1 + cross2 + frac_part;
  return result_sign ? -result : result;
}
```

首先通过联合体 `Float` 将输入的两个 `FLOAT` 数拆解为符号位、整数部分和小数部分，并根据符号位判断结果的正负。接着，分别计算整数部分相乘（结果需乘以 `2^16` 以保持定点格式）、两个交叉项（整数×小数）、以及小数部分相乘。小数相乘的结果除以 `2^16` 得到低位贡献，并根据余数是否大于等于一半进行四舍五入。最后将上述各部分相加组成最终结果，并根据之前的符号位调整正负，得到正确的 `FLOAT` 格式乘积。

> `FLOAT_FACTOR` 是一个宏定义，值为 2^{16}，表示 `FLOAT` 类型中小数部分的放大倍数，用于将实数转换为定点格式的整数表示。

（三）实现F_div_F函数

`F_div_F` 函数用于实现两个 `FLOAT` 类型定点数的除法运算。

```
FLOAT F_div_F(FLOAT a, FLOAT b) {
  assert(b != 0);
  FLOAT dividend = Fabs(a);
  FLOAT divisor = Fabs(b);
  FLOAT quotient = dividend / divisor;
  dividend %= divisor;

  for (int i = 0; i < 16; i++) {
    dividend <<= 1;
    quotient <<= 1;
    if (dividend >= divisor) {
      dividend -= divisor;
      quotient++;
    }
  }
  // 若符号不同，则结果为负
  if ((a ^ b) & 0x80000000) { quotient = -quotient; }
  return quotient;
}
```

首先对输入的两个 `FLOAT` 操作数取绝对值，保留符号信息用于最终结果处理。接着先执行整数部分除法，得到初步的商，并计算余数。然后通过模拟十进制除法的小数扩展过程，循环左移余数并逐位更新结果的小数部分，共进行 16 次以保持 `FLOAT` 的小数精度。最后根据原始操作数的符号异或判断结果符号，得到最终的定点除法结果。

（四）实现Fabs函数

`Fabs` 函数用于计算 `FLOAT` 类型定点数的绝对值，其实现流程非常简单：判断输入值是否为负数，若为负则取其相反数，否则直接返回原值，从而去除符号位，得到对应的非负数。

```
FLOAT Fabs(FLOAT a) { return a < 0 ? -a : a; }
```

（五）实现FLOAT和int之间的相互转换以及运算

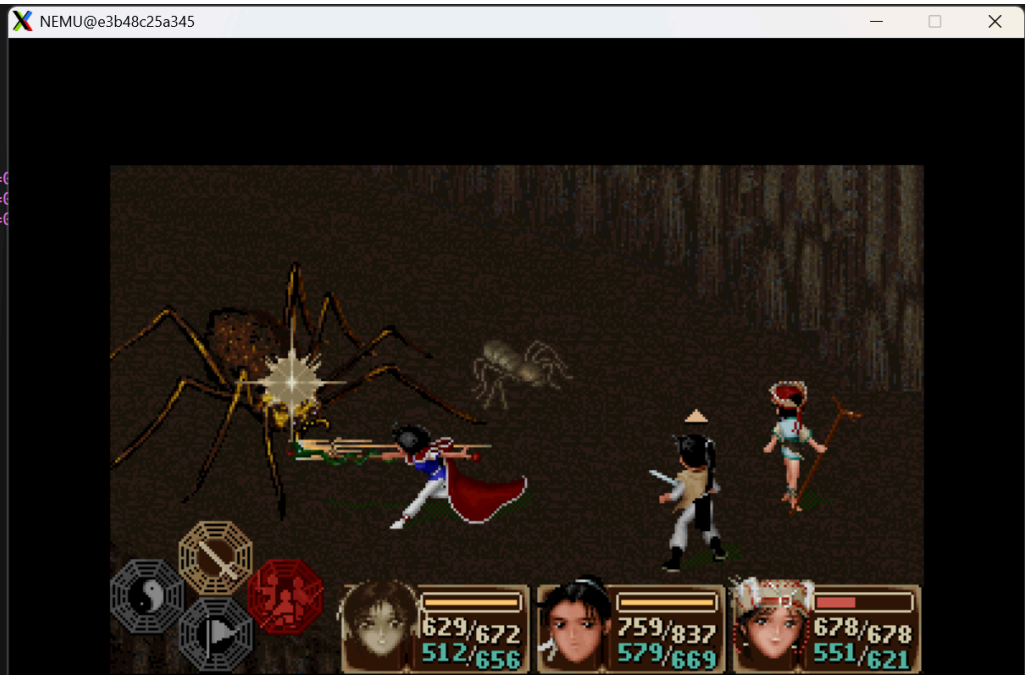
`FLOAT` 类型是以 2^{16} 为比例的定点数表示方式，整数部分在高位，小数部分在低 16 位。要在 `FLOAT` 和 `int` 之间进行转换，只需通过位移操作进行缩放：将 `FLOAT` 右移 16 位即可提取整数部分，而将 `int` 左移 16 位即可转换为 `FLOAT` 类型。
由于 `FLOAT` 本质上就是一个整数，在 `FLOAT` 与 `int` 的乘法和除法中，可以直接使用整数的 `*` 和 `/` 运算，结果仍保持定点格式的正确性。

```
static inline int F2int(FLOAT a) { return a >> 16; }
static inline FLOAT int2F(int a) { return a << 16; }
static inline FLOAT F_mul_int(FLOAT a, int b) { return a * b; }
static inline FLOAT F_div_int(FLOAT a, int b) { return a / b; }
```

（六）实验结果

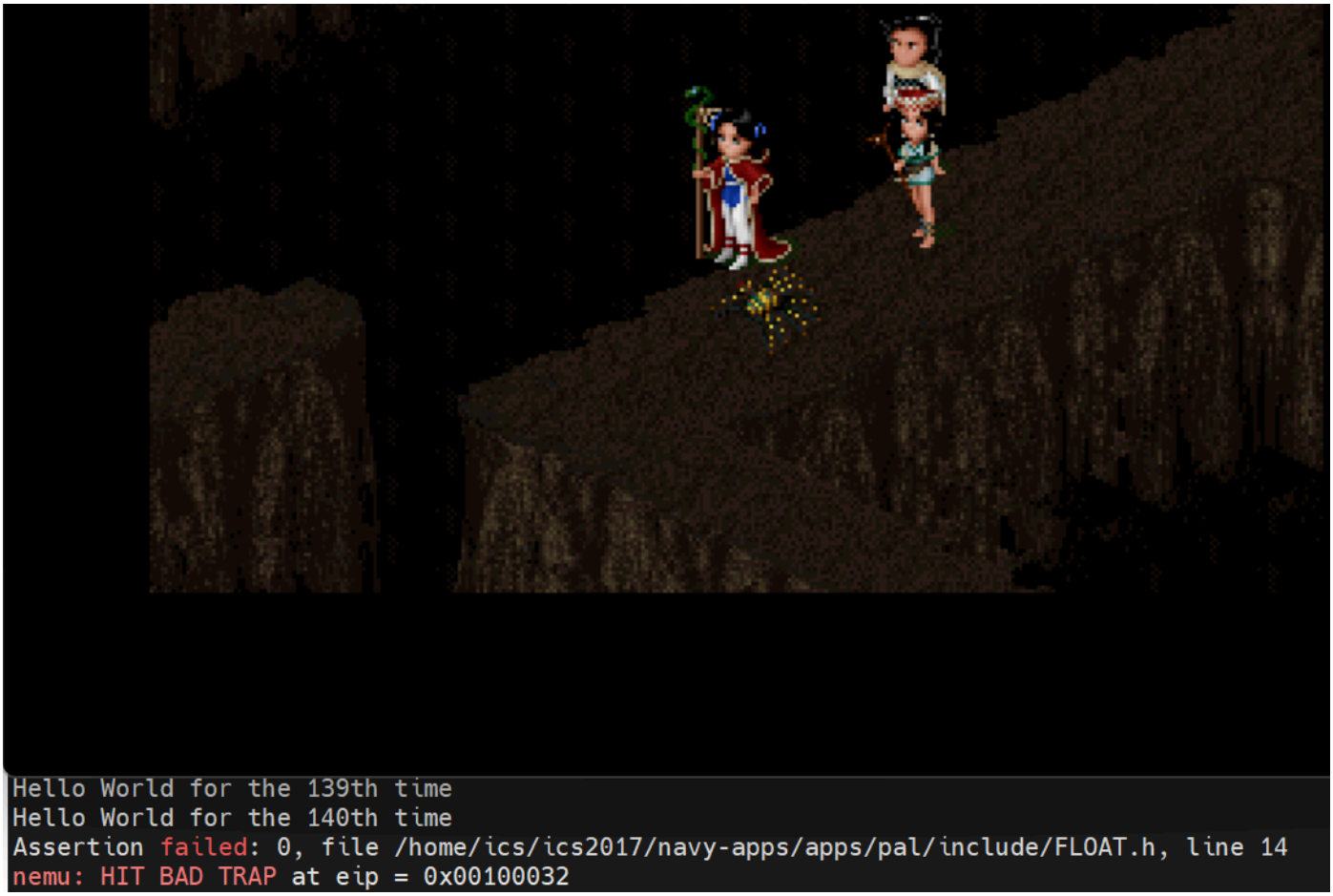
我们现在能够在《仙剑奇侠传》中进行战斗。


```
Hello World for the 205th time
Hello World for the 206th time
Hello World for the 207th time
Hello World for the 208th time
Hello World for the 209th time
Hello World for the 210th time
Hello World for the 211th time
Hello World for the 212th time
[src/mm.c,38,mm_brk] first=0x83e9000,end=0
[src/mm.c,38,mm_brk] first=0x83ea000,end=0
[src/mm.c,38,mm_brk] first=0x83fa000,end=0
Hello World for the 213th time
Hello World for the 214th time
Hello World for the 215th time
Hello World for the 216th time
Hello World for the 217th time
Hello World for the 218th time
Hello World for the 219th time
Hello World for the 220th time
Hello World for the 221th time
Hello World for the 222th time
Hello World for the 223th time
Hello World for the 224th time
Hello World for the 225th time
Hello World for the 226th time
Hello World for the 227th time
Hello World for the 228th time
Hello World for the 229th time
Hello World for the 230th time
Hello World for the 231th time
Hello World for the 232th time
Hello World for the 233th time
Hello World for the 234th time
```



四、遇到的BUG及解决思路

- 遇到的一个BUG是将要进行战斗时，程序卡住不动了，画面不动，按键也没有反应，经检查发现是符号位和边界进位的问题，在原有拆分逻辑上完善符号位的处理和进位处理逻辑后，问题解决。
- 遇到另一个BUG是，刚碰到蝎子时出现图示的错误（Assertion failed:0, FLOAT.h, line14），检查了一会没有发现什么问题，然后尝试make clean重新编译了一下，问题解决。后来发现群中有同学问了相同的问题，助教做出了解答：这个根本原因是，makefile没有把.h作为源代码文件。makefile会检测.c文件是否更改，如果有更改才会重新编译，如果只修改.h文件不会触发重新编译。



五、总结与感悟

本次实验在PA4的基础上实现了对于浮点数的支持，从而能够在《仙剑奇侠传》中进行战斗，但是参考资料少，实现过程中也有不小的挑战。

通过一学期的计算机系统设计课程的学习，巩固了操作系统所学的知识，从宏观微观的角度对计算机系统以及程序有了更深入的理解。整个PA实验工程量很大，是我接触过最完整最大的项目，指导书也非常有趣。完成PA的过程不仅增强了我的编码能力，也锻炼了自己分析问题、解决问题的能力。
十分感谢老师的讲解和指导，感谢助教们和同学们的帮助。