1 概述

1.1 实验目的

1. 充分理解通过整数来模拟实数的实现机制，并为《仙剑奇侠传》提供浮点数支持

1.2 实验内容

在PA3中已经成功在NEMU上运行了《仙剑奇侠传》，但因为缺乏浮点数相关的支持所以还不能战斗，为了能在尽量简化实现的基础上让在NEMU上运行的《仙剑奇侠传》也能支持战斗，还需要通过利用整数来模拟实数的运算的方法（定点算数，binary scaling）来为《仙剑奇侠传》提供浮点数相关支持。

2 阶段一

NEMU中使用32位整数来表示一个实数，用这样的方法所表示的实数的类型称为`FLOAT`，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），`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$A = a \times 2^{16}$(截断结果的小数部分)，对于负实数，用相应正数的相反数来表示。

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

在此基础上，NEMU中使用整型（`int`）来表示`FLOAT`类型，这样`FLOAT`类型的加法可以直接用整数加法来进行：$A + B = a \times 2^{16} + b \times 2^{16} = (a + b) \times 2^{16}$，并且因为整型（`int`）使用补码方式表示数据，这意味着`FLOAT`类型同样使用补码，所以`FLOAT`类型的减法同样可以用整数减法来进行：$A - B = a \times 2^{16} - b \times 2^{16} = (a - b) \times 2^{16}$；但`FLOAT`类型的乘法和除法就不能直接使用整数乘除法来进行了，还需要进行额外的处理。

且因为可以将从`int`到`FLOAT`类型数据的转换看成一个映射，那么在这个映射的作用下，关系运算是保序的，故`FLOAT`类型的关系运算可以用整数的关系运算来进行。

在此基础上就可以用FLOAT类型来模拟实数运算了（除了乘除法需要额外实现之外，其余运算都可以直接使用相应的整数运算来进行），例如：

```
float a = 1.2;
float b = 10;
int c = 0;
if (b > 7.9)
{
  c = (a + 1) * b / 2.3;
}
```

就可以用`FLOAT`类型来模拟：

```
FLOAT a = f2F(1.2);
FLOAT b = int2F(10);
int c = 0;
if (b > f2F(7.9))
{
  c = F2int(F_div_F(F_mul_F((a + int2F(1)), b), f2F(2.3)));
}
```

其中还引入了一些类型转换函数来实现和 `FLOAT` 相关的类型转换。

在阶段一中只需要依据先前的描述实现 `navy-apps/apps/pal/include/FLOAT.h` 文件中的 `int32_t F2int(FLOAT a)`，`FLOAT int2F(int a)`，`FLOAT F_mul_int(FLOAT a, int b)` 和 `FLOAT F_div_int(FLOAT a, int b)` 函数以及 `navy-apps/apps/pal/src/FLOAT/FLOAT.c` 目录下的 `FLOAT f2F(float a)`，`FLOAT F_mul_F(FLOAT a, FLOAT b)`，`FLOAT F_div_F(FLOAT a, FLOAT b)` 和 `FLOAT Fabs(FLOAT a)` 函数即可，并且因为《仙剑奇侠传》中的浮点数结果都可以在FLOAT 类型中表示，所以可以不关心溢出的问题。

2.1 实现方法及代码分析

`navy-apps/apps/pal/include/FLOAT.h` 文件中的 `int32_t F2int(FLOAT a)` 和 `FLOAT int2F(int a)` 函数用于实现 `FLOAT` 类型与 `int` 类型的转换，因为 `FLOAT` 类型其实是 `int` 类型的一种定点表示，$A = a \times 2^{16}$，所以只需要移位即可实现 `FLOAT` 类型与 `int` 类型的转换。

```
typedef int FLOAT;

static inline int F2int(FLOAT a) {
  //assert(0);
  return (a >> 16); //FLOAT类型数据右移16位变为int类型数据
}

static inline FLOAT int2F(int a) {
  //assert(0);
  return (a << 16); //int类型数据左移16位变为FLOAT类型数据
}
```

`FLOAT F_mul_int(FLOAT a, int b)` 和 `FLOAT F_div_int(FLOAT a, int b)` 函数用于 `FLOAT` 类型与 `int` 类型之间的乘除计算，$A \times b = a \times 2^{16} \times b = (a \times b) \times 2^{16}$，$A / b = a \times 2^{16} / b = (a / b) \times 2^{16}$，所以 `FLOAT` 类型与 `int` 类型之间的乘除计算可以直接进行，不需要进行其他额外处理。

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

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

`navy-apps/apps/pal/src/FLOAT/FLOAT.c` 目录下的 `FLOAT F_mul_F(FLOAT a, FLOAT b)` 函数用于计算两个 `FLOAT` 类型数据的乘积，可使用
$A \times B = a \times 2^{16} \times b \times 2^{16} / 2^{16} = (a \times b) \times 2^{32} / 2^{16} = (a \times b) \times 2^{16}$公式来进行计算，但需要注意的是将两个32位数相乘的结果最多可到64位，如果用32位存储乘法结果会导致高32位的数据被丢弃，导致计

算结果不正确，所以需要用64位存储乘法结果然后再进行移位。

```
FLOAT F_mul_F(FLOAT a, FLOAT b) {
  //assert(0);
  int64_t temp_result = (int64_t)a * (int64_t)b; //使用64位int存储惩罚结果
  FLOAT result = (FLOAT)(temp_result >> 16); //右移16位获得最终结果
  return result;
}
```

`FLOAT F_div_F(FLOAT a, FLOAT b)`函数用于计算两个`FLOAT`类型数据相 除的结果，如果直接套用公式$A/B = (a \times 2^{16})/(b \times 2^{16}) \times 2^{16} = (a/b) \times 2^{16}$进行计算，会导致结果中的小数位均被舍去，所以小数位的计算通过模拟除法器的操作来实现。

```
FLOAT F_div_F(FLOAT a, FLOAT b) {
  //assert(0);
  assert(b != 0); //除数不得为0

  if(a & 0x80000000)
    a = -a; //取被除数绝对值
  if(b & 0x80000000)
    b = -b; //取除数绝对值

  FLOAT result = a / b; //计算除法结果整数部分
  int temp = a % b; //计算余数，用于计算除法结果小数部分

  for (int i = 0; i < 16; i++) //通过模拟除法器的操作计算16位小数
  {
    result <<= 1;
    temp <<= 1;
    if (temp >= b) //若移位后余数大于等于除数，则该位商为1，并且余数要减去除数
    {
      temp -= b;
      result++;
    }
  }

  if((a ^ b) & 0x80000000) //若被除数与除数异号，结果为负
    return -result;
  else //若被除数与除数同号，结果为正
    return result;
}
```

`FLOAT f2F(float a)`，ieee754标准中`float`结构与《仙剑奇侠传》中的`FLOAT`结构不同，因此在转换时要计算出原本`float`类型数据的值再转换为`FLOAT`类型数据， 这里还要注意`float`类型数据与`FLOAT`类型数据尾数宽度与指数范围不同的问题。

```
typedef union
{
  struct
  {
    uint32_t fraction : 23; //尾数
    uint32_t exponent : 8; //指数
    uint32_t sign     : 1; //符号位
  };
  uint32_t value;
```

```
} Float; //Float联合体，与ieee754标准中float结构相同，用于float到FLOAT的转换

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);
  //利用指针从栈中取出float32位数据，并赋值给Float f
  //在之后的转换过程中可直接使用Float f来操作待转换的float数据
  void *voidp_temp = (void *)&a;
  uint32_t *intp_temp = (uint32_t *)voidp_temp;
  Float f;
  f.value = *intp_temp;

  FLOAT result = f.fraction | 0x800000; //在float尾数前补1，获取float未乘指数前的数据

  int exp = (int)f.exponent - 127; //获取float数据指数，根据ieee754规定，这里需要减去
127
  //FLOAT尾数长度为16，float尾数长度为23，差值为7
  //所以只有在exp大于7情况下转换结果需要左移(exp-7)位，小于15限制是为了确保结果不会因为溢出而
错误
  if(exp >= 7 && exp < 15)
    result <<= (exp - 7);
  else if(exp < 7 && exp > -17) //否则转换结果需要右移(7-exp)位，大于-17限制为了避免溢出
    result >>= (7 - exp);
  else //否则结果会溢出
    assert(0);

  if(f.sign) //通过float数据符号位来判断结果正负
    return -result;
  else
    return result;
}
```

`FLOAT Fabs(FLOAT a)`函数用于获取`FLOAT`类型数据绝对值，这里只需要对符号位做判断即可。

```
FLOAT Fabs(FLOAT a) {
  //assert(0);
  if(a & 0x80000000) //若符号位为1，该FLOAT类型数据为负数
    return -a; //绝对值为原数据的负数
  else
    return a; //否则绝对值为原数据
}
```

此外在实现了`FLOAT`类型相关函数后在NEMU上运行《仙剑奇侠传》战斗场景发现了尚未实现的指令，0F AC处指令，查询i386手册可知，0F AC处指令为SHRD指令，

SHRD — Double Precision Shift Right

Opcode	Instruction	Clocks	Description
0F AC	SHRD r/m16,r16,imm8	3/7	r/m16 gets SHR of r/m16 concatenated with r16
0F AC	SHRD r/m32,r32,imm8	3/7	r/m32 gets SHR of r/m32 with r32
0F AD	SHRD r/m16,r16,CL	3/7	r/m16 gets SHR of r/m16 concatenated with r16
0F AD	SHRD r/m32,r32,CL	3/7	r/m32 gets SHR of r/m32 concatenated with r32

Operation

```
(* count is an unsigned integer corresponding to the last operand of the
instruction, either an immediate byte or the byte in register CL *)
ShiftAmt ← count MOD 32;
inBits ← register; (* Allow overlapped operands *)
IF ShiftAmt = 0
THEN no operation
ELSE
   IF ShiftAmt ≥ OperandSize
   THEN (* Bad parameters *)
      r/m ← UNDEFINED;
      CF, OF, SF, ZF, AF, PF ← UNDEFINED;
   ELSE (* Perform the shift *)
      CF ← BIT[r/m, ShiftAmt - 1]; (* last bit shifted out on exit *)
      FOR i ← 0 TO OperandSize - 1 - ShiftAmt
      DO
         BIT[r/m, i] ← BIT[r/m, i - ShiftAmt];
      OD;
      FOR i ← OperandSize - ShiftAmt TO OperandSize - 1
      DO
         BIT[r/m,i] ← BIT[inBits,i+ShiftAmt - OperandSize];
      OD;
      Set SF, ZF, PF (r/m);
         (* SF, ZF, PF are set according to the value of the result *)
      Set SF, ZF, PF (r/m);
      AF ← UNDEFINED;
   FI;
FI;
```

指令格式为SHRD r/m16，r16，imm8/r/m32，r32，imm8，译码函数选择`make_DHelper(Ib_G2E)`，

```
/* Ev <- GvIb
 * use for shld/shrd */
make_DHelper(Ib_G2E) {
  decode_op_rm(eip, id_dest, true, id_src2, true);
  id_src->width = 1;
  decode_op_I(eip, id_src, true);
}
```

执行函数选择`make_EHelper(shrd)`，操作数宽度为默认的2/4，指令打包为`IDEX(Ib_G2E, shrd)`，需要先在`nemu/include/cpu/decode.h`文件中添加译码函数`make_DHelper(Ib_G2E)`的声明，

```
//nemu/include/cpu/decode.h
make_DHelper(Ib_G2E);
```

然后在`nemu/src/cpu/exec/all-instr.h`文件中添加执行函数`make_EHelper(shrd)`的声明，

```
//nemu/src/cpu/exec/all-instr.h
make_EHelper(shrd);
```

最后实现对应的执行函数，

```
make_EHelper(shrd)
{
  rtl_shr(&t0, &id_dest->val, &id_src->val); //id_dest->val右移id_src->val位
  if(decoding.is_operand_size_16) //判断操作数宽度
    rtl_addi(&t1, &tzero, 16);
  else
    rtl_addi(&t1, &tzero, 32);
  rtl_sub(&t1, &t1, &id_src->val); //计算需要第二个源操作数补全的位数，结果存储在t1
  rtl_shl(&t2, &id_src2->val, &t1); //id_src2->val左移t1位，结果暂存在t2
  rtl_or(&t0, &t0, &t2); //将id_dest->val右移结果与id_src2->val左移结果拼接
  operand_write(id_dest, &t0); //结果回写

  rtl_update_ZFSF(&t0, id_dest->width); //更新ZF，SF标志位

  print_asm_template3(shrd);
}
```

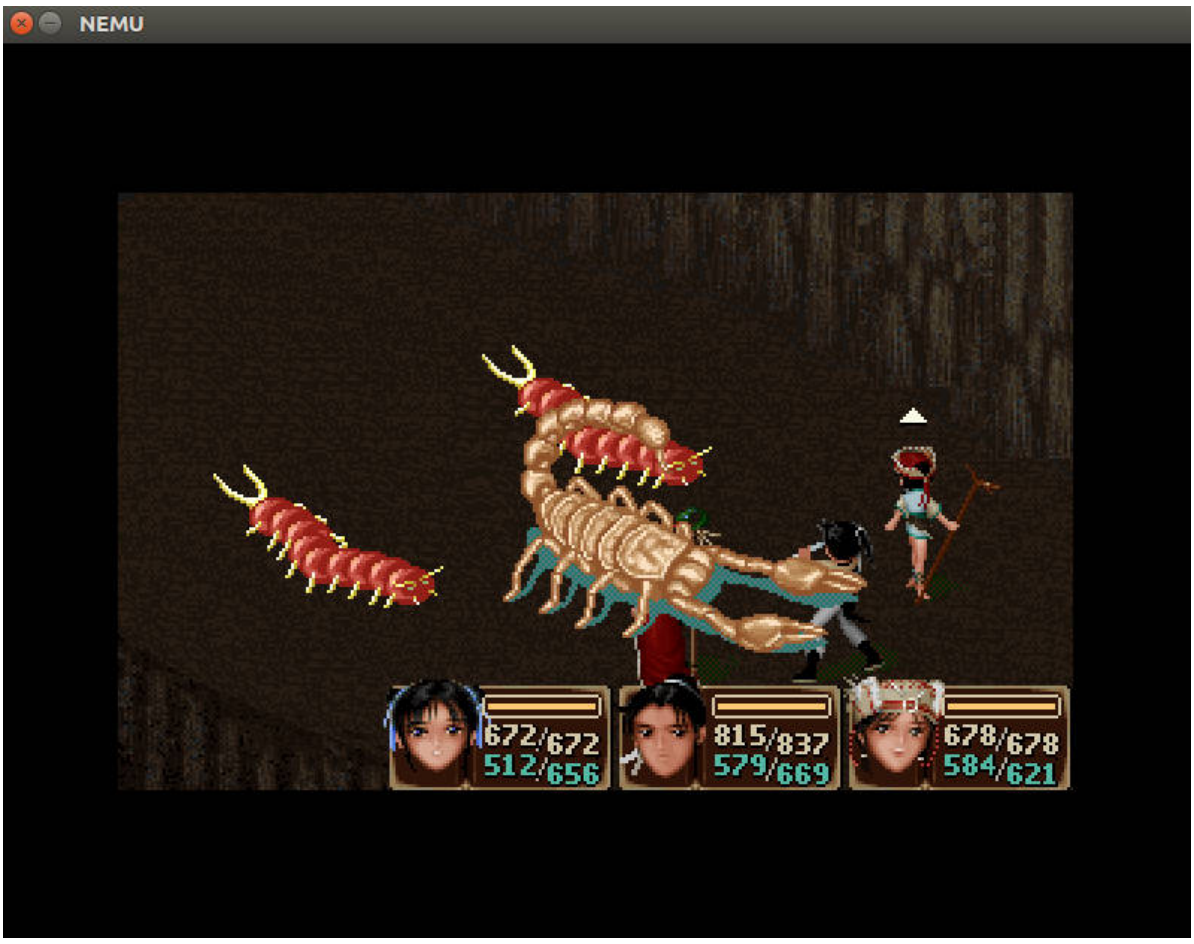
最后在`opcode_table`数组对应位置填写打包好的SHRD指令。

```
   /*2 byte_opcode_table */
/* 0xac */  IDEX(Ib_G2E, shrd), EMPTY, EMPTY, IDEX(E2G, imul2),
```

这样《仙剑奇侠传》就可以成功运行存档一中的战斗场景了。

2.2 运行结果

《仙剑奇侠传》成功运行存档一中的战斗场景。



2.3 Bug总结

实现浮点数支持部分的Bug不多，主要是`F_div_F()`函数与`f2F()`函数的实现，其中`FLOAT`类型相除不可以直接相除然后左移16位，这会导致小数部分全部被舍去，结果误差很大；`float`到`FLOAT`的转换需要注意到`float`本身的定义，不要忘记在尾数前补1，并且因为`FLOAT`与`float`的尾数长度不同，所以移位操作也要做相应的处理，才能确保最终得到正确的结果。

3 感悟与体会

1. 尽管在日常的程序编写中不会用到使用整数来表示的实数，但是在像NEMU这样性能羸弱的机器上使用这样的`FLOAT`类型浮点数来进行相关运算的却可以在满足程序要求的前提下尽可能简化实现并提高程序运行速度。
2. 尽管在PA1-PA4的实验中已经逐步将NEMU和Nanos-lite完善，但想要在NEMU上运行更多的程序并让运行速度尽可能提高依然还有着非常多的工作需要完成。