南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机系统实验报告

PA5

年级：2022 级

专业：计算机科学与技术

姓名：姚知言

学号：2211290

指导教师：卢冶

2025 年 5 月 22 日

目录

一、 实验目的

在本次实验中，主要目的是通过实现浮点数支持完成完整的仙剑奇侠传运行。并掌握对于计算机系统设计的性能测试和性能改善的基本方式。

二、 实验重要内容

PA5 的实验包括以下要点：

- 以 32 位整数模拟实现浮点数，完成运算函数设计。

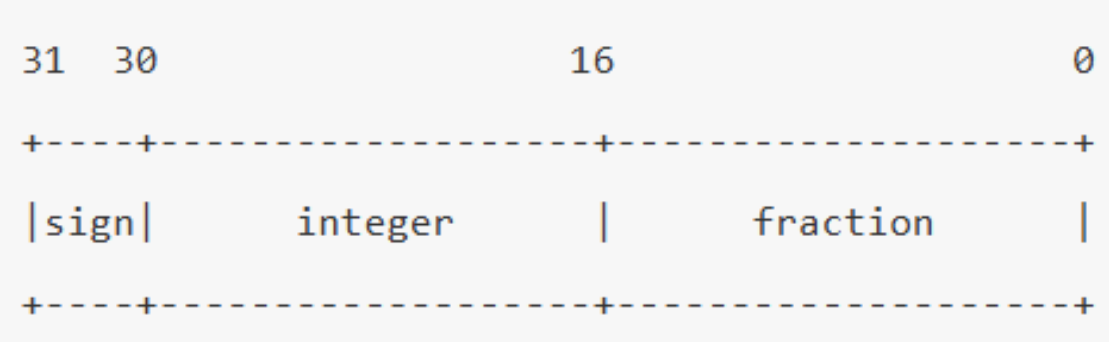

图 1: 浮点数支持

- 通过仙剑奇侠传的战斗，对完整计算机系统设计进行测试。
- 使用 perf 完成计算机系统性能分析。
- 使用 jit 等思想对计算机系统性能进行进一步提升。

三、 实验方法与结果

(一) 实现 binary scaling

由于我们并不能真正支持 float 浮点数的运算，所以用 binary scaling 方法表示实数，这种类型命名为 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)。

在此约定下实现在 FLOAT 类型下的计算。

1. 实现 int 与 FLOAT 之间的转换

首先实现 int 和 FLOAT 的转换。

- 对于 FLOAT 向 int 的转换，相当于舍弃小数部分的结果，即右移 16 位。为尽可能保留精度，我使用四舍五入的方法完成实现，增加 1«15 相当于增加 0.5。
- 对于 int 向 FLOAT 的转换，直接左移 16 位即可，相当于把小数部分初始化为 0。

```c
static inline int F2int(FLOAT a) {
  return (a>=0) ? ((a + (1 << 15) ) >> 16 ) : -((-a + ( 1 << 15 ) ) >> 16 );
}

static inline FLOAT int2F(int a) {
  return a << 16;
}
```

2. 实现 FLOAT 与 int 之间的乘除法

对于 FLOAT 类型，与 int 类型的乘除法相当于对其扩大或者缩小多少倍，直接进行乘法或除法即可。

```c
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;
}
```

3. 实现 FLOAT 类型绝对值的计算

FLOAT 类型绝对值的计算与 int 类型无差别，直接类比 int 进行负数取反即可。

```c
FLOAT Fabs(FLOAT a) {
  return a>=0 ? a : -a;
}
```

4. 实现 float 类型数值向 FLOAT 类型的转换

需要在不适用 x87 浮点指令的情况下，完成浮点数的解析。因此我们需要自己进行 float 浮点数的拆分。首先将 float 结果拆分成符号位 sign，阶码位和尾数位。

对 e 进行初始化时，将阶码位结果转换为实际需要对尾数结果移动的位数，-127 是因为阶码位的偏移量是 127，+16 是因为我们 FLOAT 实现中最后 16 位是小数位，-23 是因为当前尾数有 23 位，也就是当前 m 中保存的结果后 23 位是小数位。

对 m 进行初始化时，因为省略了整数位的 1，所以需要最高位的 1 补充。

最后根据 m 和 e 完成 FLOAT 的计算，最后根据符号位判定是否需要取反即可。

```c
FLOAT f2F(float a) {
  uint32_t b;
  char* dest = (char*)(&b),*src = (char*)(&a);
  for(int i = 0; i < sizeof(uint32_t); i++)
    dest[i] = src[i];
  uint32_t sign = b >> 31;
  int32_t e = ((b >> 23) & 0xFF) - 127 + 16 - 23;
  uint32_t m = (b & 0x7FFFFF) | 0x800000;
  int32_t A = (e >= 0) ? (m << e) : (m >> (-e));
```

```
  if(sign) A = -A;
  return A;
}
```

5. 实现 FLOAT 类型数据之间的乘法和除法运算

对于乘法其实可以进行直接进行 64 位运算转换回 32 位后再右移 16 位来实现。但是这样的话需要再回到之前的部分完成长乘法和移位指令的设计。(进行了尝试，但出现了 i386 指令缺失的提示)

为了不进行指令设计，我重新设计了只使用 32 位运算的替代方案。首先进行符号位的计算，然后将两个数分别拆成整数位和小数位，进行分别相乘，最后再拼接。

其中两个整数位的计算结果需要左移 16 位，两个小数位的计算结果需要右移 16 位（并且只保留后 16 位，这里最高位不应该被视为符号位），整数和小数的计算结果不需要移位。

最后进行符号位的判定。

对于除法，符号位的判定和计算方式与乘法没有区别，对于绝对值下的除法运算，我们采取按位除法的形式进行。

商的整数部分可以通过一次性的除法获得，小数部分通过 16 轮的移位计算完成，最终获取商的结果。

```
FLOAT F_mul_F(FLOAT a, FLOAT b) {
  uint32_t sign = ((a >> 31) & 0x1 ) ^ ((b >>31) & 0x1);
  a = Fabs(a);
  b = Fabs(b);
  int32_t a_hi = a >> 16;
  int32_t a_lo = a & 0xFFFF;
  int32_t b_hi = b >> 16;
  int32_t b_lo = b & 0xFFFF;
  FLOAT r = ((a_hi * b_hi) << 16) + a_hi * b_lo + b_hi * a_lo + (((a_lo *
      b_lo)>> 16) & 0xFFFF);
  if(sign) r = -r;
  return r;
}

FLOAT F_div_F(FLOAT a, FLOAT b) {
  uint32_t sign = ((a >> 31) & 0x1 ) ^ ((b >>31) & 0x1);
  a = Fabs(a);
  b = Fabs(b);
  int32_t q = a / b, r = a % b;
  for(int i = 0; i < 16; i++){
    r <<= 1,q <<= 1;
    if(r >= b)
      r -= b,q |= 1;
  }
  if(sign) q = -q;
  return q;
}
```

6. 结果展示

运行仙剑奇侠传的战斗，已经能够正确运行，并且进行攻击。

图 2: 仙剑奇侠传的战斗

(二) 使用 perf 进行性能剖析

perf（Performance Counters for Linux）是 Linux 系统下的性能分析工具，基于硬件性能计数器（PMC）和内核事件追踪，用于监测程序或系统的性能瓶颈。主要功能包括：CPU 性能分析，函数级采样分析，系统调用追踪，硬件事件监控，动态探针等。

首先试图使用 pref record 进行信息收集，但发现 perf 还没有安装，此处根据系统中的提示进行补充安装。

```
latesoon@latesoon-virtual-machine:~/ics2017$ perf record nemu/build/nemu nanos-lite/build/nanos-lite-x86-nemu.bin
程序“perf”尚未安装。 您可以使用以下命令安装：
sudo apt install linux-tools-common
```

图 3: perf 未安装

依次执行以下指令完成安装。

```
sudo apt-get install linux-tools-common
sudo apt install linux-tools-4.15.0-142-generic
```

安装完成后，再执行又遇到了权限不足的问题，因此为指令加上 sudo，成功执行。

```
sudo perf record nemu/build/nemu nanos-lite/build/nanos-lite-x86-nemu.bin
```

```
sudo perf report
```

最终导出的报告如图4所示。报告说明如下：

- Overhead（开销占比）：表示某个函数或指令在采样中的出现比例，越高说明 CPU 时间占用越多。
- Command（进程名）：显示消耗资源的进程或二进制文件名称。
- Shared Object（共享库）：指出热点代码所属的动态库（如 libc.so、内核 [kernel.kallsyms]）。
- Symbol（函数名）：具体的函数或符号名称，[k] 表示内核态的调用。

```
Samples: 460K of event 'cpu-clock', Event count (approx.): 115242250000
Overhead  Command      Shared Object          Symbol
  21.88%  nemu         nemu                   [.] paddr_read
  16.26%  nemu         nemu                   [.] is_mmio
  12.02%  nemu         nemu                   [.] page_translate
   9.08%  nemu         nemu                   [.] vaddr_read
   6.44%  nemu         nemu                   [.] exec_real
   5.60%  nemu         nemu                   [.] read_ModR_M
   3.32%  nemu         nemu                   [.] load_addr
   2.94%  nemu         [kernel.kallsyms]      [k] finish_task_switch
   2.33%  nemu         nemu                   [.] exec_wrapper
   2.09%  nemu         nemu                   [.] device_update
   1.79%  nemu         nemu                   [.] operand_write
   1.64%  nemu         nemu                   [.] decode_mov_E2G
   1.15%  nemu         nemu                   [.] decode_mov_G2E
   1.14%  nemu         nemu                   [.] cpu_exec
   1.01%  nemu         nemu                   [.] exec_add
   1.00%  nemu         nemu                   [.] decode_J
   0.84%  nemu         nemu                   [.] exec_cmp
   0.71%  nemu         nemu                   [.] rtl_setcc
   0.70%  nemu         nemu                   [.] decode_G2E
   0.69%  nemu         nemu                   [.] exec_2byte_esc
   0.67%  nemu         nemu                   [.] decode_E2G
   0.61%  nemu         [kernel.kallsyms]      [k] __do_softirq
   0.60%  nemu         [kernel.kallsyms]      [k] __lock_text_start
   0.54%  nemu         nemu                   [.] decode_r
   0.48%  nemu         nemu                   [.] decode_I2a
   0.47%  nemu         nemu                   [.] exec_inc
   0.45%  nemu         nemu                   [.] exec_imul2
   0.42%  nemu         nemu                   [.] vaddr_write
   0.42%  nemu         libc-2.23.so           [.] __memcpy_sse2_unaligned
```

图 4: perf 性能剖析结果

从结果中可以看到，性能的主要瓶颈在 paddr_read，is_mmio，page_translate 等函数，即主要集中在页面读取和转换中。这说明主要的性能开销集中在访存中，加速内存的读取能够较好的提升性能。

(三) 实现即时编译

即时编译（Just-In-Time Compilation，JIT）是一种动态编译技术，在程序运行时将中间代码（如字节码）动态翻译成机器码执行，兼具解释执行的灵活性和静态编译的高效性。

具体到实现上来说就是：同一条指令，已经完成译码的指令不需要重新译码，而是在上一次译码的结果进行缓存，直接完成取指 + 执行上一次译码结果。

基于基本快的即时编译的主要流程如下：

```
while (1) {
  tb = query_cache(cpu.eip);
  if ( cache miss ) {
    tb = jit_translate(cpu.eip); // translate a basic block
    update_cache(cpu.eip, tb);
  }

  jump_to(tb); // cpu.eip will be updated while executing tb
}
```

图 5: JIT

通过缓存已经完成译码的指令，减少指令译码执行开销，从而提升 CPU 性能。

具体到 NEMU 的实现中，以 RTL 为分界，可以把即时编译模式的 NEMU 分为两部分: 前端用于将客户程序的机器语言编译成 RTL，后端负责将 RTL 编译成 native 机器语言。这样的话，无论是后期增加新的客户程序架构，或者增加一种机器架构，都只需要完成前端或者后端单一部分的设计就可以了，增加泛用性。

如果需要进一步优化的话，由于 JIT 运行本身具有一定的开销，我们并不希望在所有代码段均应用 JIT 技术，而是希望通过 profiling 工具或者动态插桩等方式动态识别哪些代码段是经常被执行的，然后我们只需要去对这些代码应用 JIT 技术就可以了，这样能够进一步改善程序的性能。

四、 问答题

(一) 比较 FLOAT 和 float

- Q:

  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?

- A:

  我们实现的 FLOAT 牺牲动态范围, 换取均匀的小数精度 (适合固定范围的数值计算), float 通过指数位动态调整精度和范围（适合科学计算，但小范围内精度可能不足)。FLOAT 能够表示 ± 32768.9999847 中的浮点数，而 float 能够表示 $\pm 1.2\times 10^{-38} - \pm 3.4\times 10^{38}$ 中的所有浮点数。然而，这样的范围和精度在仙剑奇侠传的运行过程中已经足够使用。

  在当今的计算机环境下，浮点数运算的指令集设计已经相对完善。然而，在我们的简易 cpu 实现下并不能实现该功能。使用 FLOAT 的设计以受限范围和手动管理为代价，换取确定性精度和硬件兼容性，是一个较好的权宜的方式。

(二) 性能瓶颈的来源

- Q:

  Profiler 可以找出实现过程中引入的性能问题, 但却几乎无法找出由设计引入的性能问题. NEMU 毕竟是一个教学模拟器, 当设计和性能有冲突时, 为了达到教学目的, 通常会偏向选择易于教学的设计. 这意味着, 如果不从设计上作改动, NEMU 的性能就无法突破上述取舍造成的障壁. 纵观 NEMU 的设计, 你能发现有哪些可能的性能瓶颈吗?

- A:

  – 执行逻辑: NEMU 采用简单的取指-译码-执行进行指令运行。这种设计虽然直观易懂，但每条指令都需要经历完整的流程，无法利用现代 CPU 的流水线、分支预测等硬件优化。

  – 内存访问: NEMU 通过软件完全模拟内存层次，每次访存都需要逐层调用，没有使用缓存等方式减少访存开销。

  – 硬件支持: NEMU 依托 i386 虚拟机运行，受限于 i386 虚拟机的性能限制以及缺乏硬件加速支持。

五、 所遇 BUG 及解决思路

在完成仙剑奇侠传的浮点数设计时, 起初进入战斗后会直接卡死, 也不会显示战斗属性页面。最后经过排查，将每一个子运算的结果打印出来。

```
//FLOAT*FLOAT
printf("a:%d b:%d a_hi:%d a_lo:%d b_hi:%d b_lo:%d\n",a,b,a_hi,a_lo,b_hi,b_lo);
printf("sum1:%d sum2:%d sum3:%d sum4:%d sign:%d r:%d\n",(a_hi * b_hi) << 16, a_hi * b_lo,b_hi * a_lo,((a_lo * b_lo)>> 16) & 0xFFFF,sign,r);

//FLOAT/FLOAT
printf("div: a:%d b:%d q:%d\n",a,b,q);

//float to FLOAT
printf("f2F: b:%d, a:%d\n",b,A);
```

最后发现乘法和除法的结果不太正确。

除法的结果是没能对整数部分的除法进行合理运算，导致所有的计算都被拖到了小数部分进行（即所有运算结果都接近于 65536)。

乘法的结果起初就比较隐蔽，最后发现是 16 位小数部分进行相乘的时候，结果可能会来到 32 位，而进行向右移位操作的时候若首位为 1 会将其判断为负数，从而导致整个运算结果为负数，这也是导致程序运算崩溃的直接原因。

最后增加了一个和 0xFFFF 的与操作后得到了解决。

```
//FLOAT r = ((a_hi * b_hi) << 16) + a_hi * b_lo + b_hi * a_lo + ((a_lo * b_lo)>> 16);
FLOAT r = ((a_hi * b_hi) << 16) + a_hi * b_lo + b_hi * a_lo + (((a_lo * b_lo) >> 16) & 0xFFFF);
```