PA2 不停止计算的机器

问题答案:

必答题：

1

在 nemu/include/cpu/rtl.h 中,你会看到由 static inline 开头定义的各种 RTL 指令函数.选择其中一个函数,分别尝试去掉 static,去掉 inline 或去掉两者,然后重新进行编译,你会看到发生错误.请分别解释为什么会发生这些错误?你有办法证明你的想法吗?

只去掉一个不会出现错误，同时去掉两个时会发生错误：重定义。原因在于rtl.h会被多个文件引用，因此rtl.h的代码会出现在多个文件中。

如果两个均去掉时，rtl函数的定义就是一个强符号，强符号不能被定义多次，否则会发生链接错误。static可以把符号的作用范围限定在当前文件，因此在链接时不会发生错误。而inline会直接以函数体内的代码替换调用该函数的位置，实际上该函数的符号并不会出现，因此也不会发生链接错误。但是在实际中，inline只能起到建议编译器的作用，不一定编译出来的函数就是内联的，所以在需要内联的地方常常会加上static。

证明：编写一个头文件test.h，两个c文件test1.c, test2.c，两个c文件均引用该头文件，并在其中一个文件中编写main函数。在头文件中编写一个inline函数和一个static函数

```
inline void func1(){}
static void func2(){}
```

使用编译命令

```
gcc -c test1.c -o test1.o
gcc -c test2.c -o test2.o
gcc test1.o test2.o -o test
```

编译成功，证明猜想正确

启示：在头文件中尽量不要定义函数和全局变量，因为头文件会被多个文件引用，如果头文件中有强符号就会发生链接错误。应该在头文件中写函数声明，c和cpp文件中写函数定义。如果需要在头文件中需要用到全局变量，应该声明为extern，然后在其中一个c文件中定义该全局变量。全局变量应该少用，如果变量不需要作用与其他文件，应该声明为static。

2

了解 Makefile 请描述你在 nemu 目录下敲入 make 后,make 程序如何组织.c 和.h 文件,最终生成可执行文件nemu/build/nemu.(这个问题包括两个方面:Makefile 的工作方式和编译链接的过程.)

关于 Makefile 工作方式的提示:

Makefile 中使用了变量,包含文件等特性

首先是列出需要编译的文件, 通过find指令列出所有c文件，并以此得到所需编译的obj文件

```
# Files to be compiled
SRCS = $(shell find src/ -name "*.c" | grep -v "isa")
SRCS += $(shell find src/isa/$(ISA) -name "*.c")
OBJS = $(SRCS:src/%.c=$(OBJ_DIR)/%.o)
```

将相应的c文件编译成.o文件

```
# Compilation patterns
$(OBJ_DIR)/%.o: src/%.c
    @echo + CC $<
    @mkdir -p $(dir $@)
    @$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
```

然后将.o文件以及库文件链接起来，生成最终的二进制文件。

```
$(BINARY): $(OBJS)
    # $(call git_commit, "compile")
    @echo + LD $@
    @$(LD) -O2 -rdynamic $(SO_LDLAGS) -o $@ $^ -lSDL2 -lreadline -ldl
```

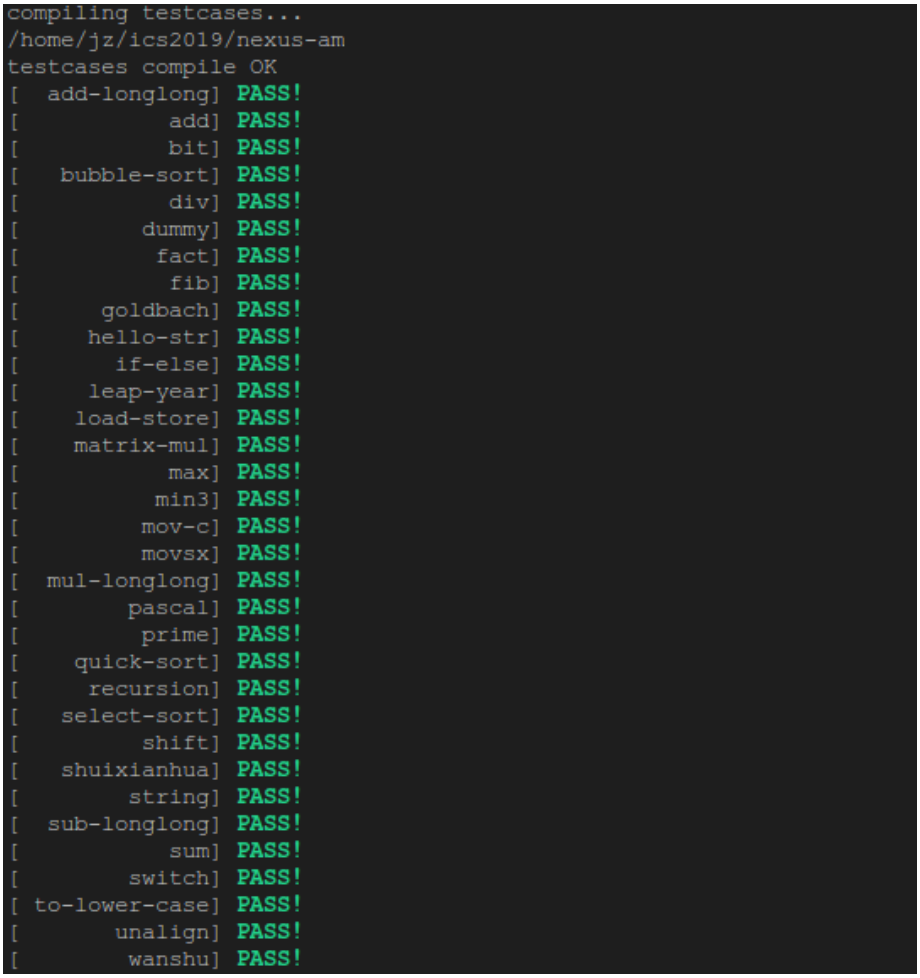
> *Makefile 运用并重写了一些 implicit rules*
> *在 man make 中搜索-n 选项,也许会对你有帮助*

```
-n, --just-print, --dry-run, --recon
Print the commands that would be executed, but do not execute them (except in certain circumstances).
```

-n选项是打印指令但是不会执行。

结果演示

runall.sh



hellostr

```
+ LD -> build/amtest-native
/home/jz/ics2019/nexus-am/tests/amtest/build/amtest-native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
```

time

```
❯ make mainargs=t run
# Building amtest [native] with AM_HOME {/home/jz/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
# Creating binary image [native]
+ LD -> build/amtest-native
/home/jz/ics2019/nexus-am/tests/amtest/build/amtest-native
2020-4-30 18:57:42 GMT (1 second).
2020-4-30 18:57:43 GMT (2 seconds).
2020-4-30 18:57:44 GMT (3 seconds).
2020-4-30 18:57:45 GMT (4 seconds).
2020-4-30 18:57:46 GMT (5 seconds).
2020-4-30 18:57:47 GMT (6 seconds).
```

跑分

microBench

```
==================================================
MicroBench PASS        59896 Marks
                 vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 273 ms
```

coreMark

```
==================================================
CoreMark PASS       49515 Marks
                vs. 100000 Marks (i7-7700K @ 4.20GHz)
```

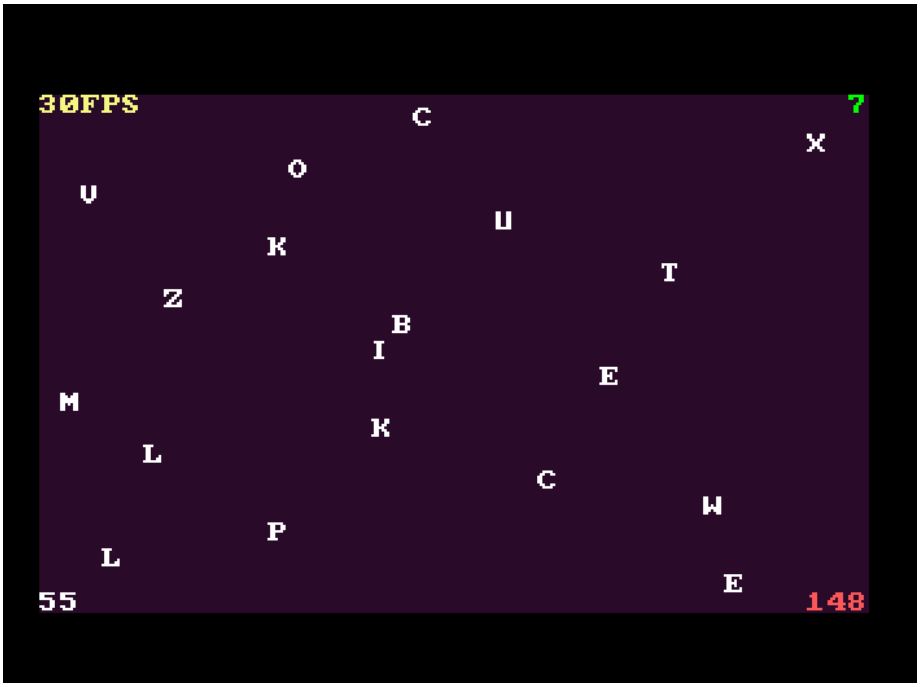
dhrystone

```
==================================================
Dhrystone PASS         55056 Marks
                  vs. 100000 Marks (i7-7700K @ 4.20GHz)
```

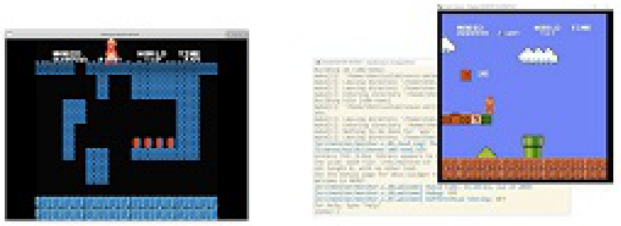
跑分还可以，毕竟用的是物理机，cpu参数是 i5-7200U @2.5GHz

vga

打字游戏

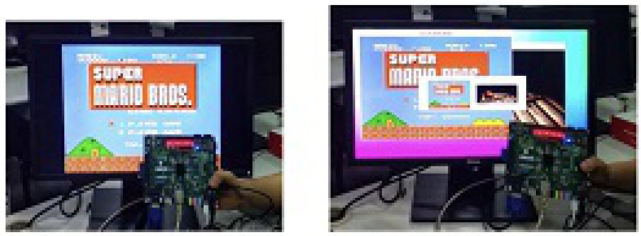


幻灯片



(a) Linux native, back-ended with SDL2 (screenshot)

(b) a teaching x86 full system emulator (screenshot)



(c) a teaching MIPS32 SoC (FPGA, only TRM and IOE)

(d) a teaching OS (with GUI) and MIPS32 SoC (FPGA)

Figure 4. The same LiteNES emulator running on different platforms.

马里奥

实现过程

环境变量

既然老师提到了环境变量的问题，这里可以给一个建议，可以写一个简单的脚本文件，作为项目文件的一部分，这样可以方便的导入环境变量。我写的脚本文件export.txt如下

```
export AM_HOME="$(pwd)/nexus-am"
export NEMU_HOME="$(pwd)/nemu"
export NAVY_HOME="$(pwd)/nevy-apps"
```

这里用pwd代替当前路径，避免同学们在.bashrc或者.profile中加入路径时拼写错误带来的问题，而且能够兼容每个项目。使用方法如下，将该文件放在ics目录下，每次进入该目录时使用source export.txt，即可导入环境变量。由于是临时导入，在更改项目路径的时候也不需要修改.bashrc文件

计算机的简单模拟：

```
while (1)
{
    取指
    译码
    执行
    更新 pc
}
```

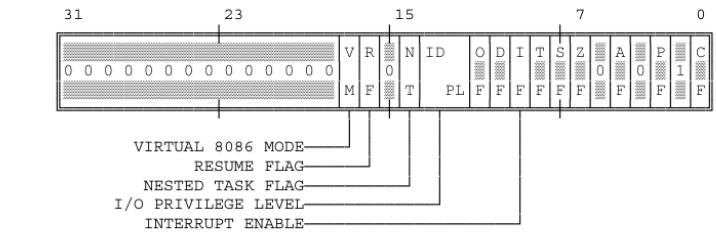
大小端的问题

> *Motorola 68k 系列的处理器都是大端架构的.现在问题来了,考虑以下两种情况:*
> *假设我们需要将 NEMU 运行在 Motorola 68k 的机器上(把 NEMU 的源代码编译成 Motorola 68k 的机器码)*
> *假设我们需要编写一个新的模拟器 NEMU-Motorola-68k,模拟器本身运行在 x86 架构中,但它模拟的是Motorola 68k 程序的执行*
> *在这两种情况下,你需要注意些什么问题?为什么会产生这些问题?怎么解决它们?事实上不仅仅是立即数的*
> *访问,长度大于 1 字节的内存访问都需要考虑类似的问题.*

如果是Motorola 68k模拟x86平台的指令，那么在读取立即数时或者访问内存时不能一次读取多个字节，而是需要一个字节一个字节的读取，把低地址的字节放在高地址，相当于进行字节水平的翻转。

x86模拟Motorola 68k时因为大小端是相反的，所以也要进行字节水平的翻转。

Figure 4-1. System Flags of EFLAGS Register



NOTE
0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE.

EFLAGS

NEMU中用到了EFLAGS中的CF， ZF， SF， IF， OF。意思如下：

CF：0若算术操作产生的结果在最高有效位(most-significant bit)发生进位或借位则将其置1，反之清零。这个标志指示无符号整型运算的溢出状态，这个标志同样在多倍精度运算(multiple-precision arithmetic)中使用。

ZF： 6若结果为0则将其置1，反之清零。

SF(bit 7) [Sign flag] 该标志被设置为有符号整型的最高有效位。(0指示结果为正，反之则为负)

IF(bit 9) 该标志用于控制处理器对可屏蔽中断请求(maskable interrupt requests)的响应。置1以响应可屏蔽中断，反之则禁止可屏蔽中断。

OF(bit 11) [Overflow flag] 如果整型结果是较大的正数或较小的负数，并且无法匹配目的操作数时将该位置1，反之清零。这个标志为带符号整型运算指示溢出状态。

在x86处理器初始化之后，EFLAGS寄存器的状态值为0x2。第1、3、5、15以及22到31位均被保留，这个寄存器中的有些标志通过使用特殊的通用指令可以直接被修改，但并没有指令能够检查或者修改整个寄存器。

在结构CPU中添加eflags变量

```
{
    ......
  vaddr_t pc;
  union {
    struct
    {
      uint32_t CF : 1;
      uint32_t x1 : 5;
      uint32_t ZF : 1;
      uint32_t SF : 1;
      uint32_t x2 : 1;
      uint32_t IF : 1;
      uint32_t x3 : 1;
      uint32_t OF : 1;
      uint32_t x4 : 4;
      uint32_t x5 : 16;
```

```
    } eflags;
    rtlreg_t eflag;
  };
} CPU_state;
```

要使用到位域使得eflags的各个位能紧密的排布在四个字节中。这里需要注意的是如果这个字节剩余的位不足以容纳当前制定的位，那么这个变量会被安排到下一个字节中，我们应该避免这种情况。

需要注意 eflag必须用uint32_t而不是int，如果用int的话取出一位时会被解析成-1，sbb adc等运算错误

在restart中添加eflags初始化函数

```
static void restart() {
  /* Set the initial program counter. */
  cpu.pc = PC_START;
  // init EFLAGS
  cpu.eflags = 0x2;
}
```

指令的实现

Data Movement Instructions:

> *push, pop, leave, cltd(在 i386 手册中为 cdq)*

Binary Arithmetic Instructions:

> *add, inc, sub, dec, cmp, neg*

Logical Instructions:

> *not, and, or, xor, sal(shl), shr, sar, test*

Control Transfer Instructions:

> *jmp, jcc, call, ret*

dummy的实现

dummy的C源代码就是一个return 0。我们可以从dummy.txt中看到他的反汇编

```

```

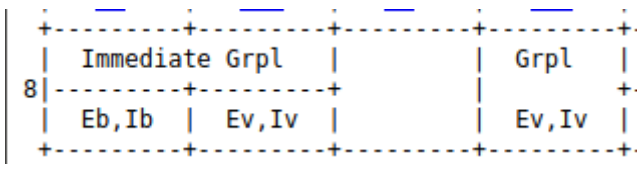
```
00100000 <_start>:
  100000:   bd 00 00 00 00          mov    $0x0,%ebp
  100005:   bc 00 90 10 00          mov    $0x109000,%esp
  10000a:   e8 09 00 00 00          call   100018 <_trm_init>
  10000f:   90                      nop

00100010 <main>:
  100010:   55                      push   %ebp
  100011:   89 e5                   mov    %esp,%ebp
  100013:   31 c0                   xor    %eax,%eax
  100015:   5d                      pop    %ebp
  100016:   c3                      ret
  100017:   90                      nop

00100018 <_trm_init>:
  100018:   55                      push   %ebp
  100019:   89 e5                   mov    %esp,%ebp
  10001b:   83 ec 14                sub    $0x14,%esp
  10001e:   68 00 00 00 00          push   $0x0
  100023:   e8 e8 ff ff ff          call   100010 <main>
  100028:   d6                      (bad)
  100029:   83 c4 10                add    $0x10,%esp
  10002c:   eb fe                   jmp    10002c <_trm_init+0x14>
```

最后的d6就是我们写nemu_trap。在nemu.h中可见。在这里我们需要实现sub, push, pop, xor等指令。下面以push为例，简单介绍一下指令的实现过程

指令首先需要译码，所以我们要先填写opcode_table，填入译码函数和执行函数。这里sub的opcode是83，我们去80386手册附录A中查找第8行，第三列。发现他的执行函数的grp1，源是I，目的是E，因此我们选择的译码函数是SI2E。然后发现项目文件里已经填好这一项了，真是尴尬，当我没说，我们继续。



然后我们的执行函数就是gp1了，也给我们填好了。那是不是没我们什么事了呢？不是。gp1虽然是一个执行函数，但是他会根据R/M位的第5,4,3三位决定具体解释成哪个执行函数。所以我们还需填写group中每一个位置的执行函数。80386手册神通广大，这个也有。我们根据表中的0到7填写即可。那么我们的gp1就是

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
           EX(add), EX(or), EX(adc), EX(sbb),
           EX(and), EX(sub), EX (xor), EX(cmp))
```

Opcodes determined by bits 5,4,3 of modR/M byte

```
G                    +-------+-------+-------+
r                    |  mod  |  nnn  |  R/M  |
o                    +-------+-------+-------+
u
p   000     001     010     011     100     101     110     111
 +-------+-------+-------+-------+-------+-------+-------+-------+
1|  ADD  |  OR   |  ADC  |  SBB  |  AND  |  SUB  |  XOR  |  CMP  |
 |       |       |       |       |       |       |       |       |
 +-------+-------+-------+-------+-------+-------+-------+-------+
2|  ROL  |  ROR  |  RCL  |  RCR  |  SHL  |  SHR  |       |  SAR  |
 |       |       |       |       |       |       |       |       |
 +-------+-------+-------+-------+-------+-------+-------+-------+
3| TEST  |       |  NOT  |  NEG  |  MUL  | IMUL  |  DIV  | IDIV  |
 | Ib/Iv |       |       |       |AL/eAX |AL/eAX |AL/eAX |AL/eAX |
 +-------+-------+-------+-------+-------+-------+-------+-------+
4|  INC  |  DEC  |       |       |       |       |       |       |
 |  Eb   |  Eb   |       |       |       |       |       |       |
 +-------+-------+-------+-------+-------+-------+-------+-------+
5|  INC  |  DEC  | CALL  | CALL  |  JMP  |  JMP  | PUSH  |       |
 |  Ev   |  Ev   |  Ev   |  eP   |  Ev   |  Ep   |  Ev   |       |
 +-------+-------+-------+-------+-------+-------+-------+-------+
```

然后我们需要实现执行函数了

对于这个sub，我们在all-instr.h中加入sub执行函数的定义make_EHelper(sub);

然后在arith.c中实现sub。在实现之前我们还是需要看80386手册，从PA实验中我第一次体会到了手册的重要性。我们需要关注两个方面，一是指令的伪代码，二是影响的eflags位。

Operation

```
IF SRC is a byte and DEST is a word or dword
THEN DEST := DEST - SignExtend(SRC);
ELSE DEST := DEST - SRC;
FI;
```

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

根据指令的伪代码，我们知道他的实现方法是 s0 = dest - src，然后把s0写入目的地。然后根据影响的flags，我们需要更新ZF, SF, CF, OF。因为AF和PF在PA中不需要维护。减法运算通过rtl_sub实现，结果的写入通过operand_write，更新ZF, SF通过rtl_update_ZFSF实现，判断有符号减法是否溢出使用rtl_is_sub_carry，判断无符号减法是否溢出使用rtl_is_sub_overflow实现。

```
make_EHelper(sub)
{
  // TODO();
  rtl_sub(&s0, &id_dest->val, &id_src->val);
  operand_write(id_dest, &s0);
  rtl_update_ZFSF(&s0, id_dest->width);
  rtl_is_sub_carry(&s1, &s0, &id_dest->val);
  rtl_set_CF(&s1);
  rtl_is_sub_overflow(&s1, &s0, &id_dest->val,
&id_src->val, id_dest->width);
  rtl_set_OF(&s1);
  print_asm_template2(sub);
}
```

这里可以总结一下指令实现的基本步骤

译码：填写opcode, 选择正确的译码函数和执行函数，在all-instr.h中加入make_EHelper的该执行函数的定义。

执行：根据指令伪代码调用rtl指令，调用operand_write写入结果。使用rtl_update_ZFSF更新ZFSF，与加法和减法有关的指令使用carry和overflow相关指令更新CF和OF。

太长不看系列

2.2 实现更多的指令

rtl指令

rtl_push : 栈顶减4，把值写入栈顶的位置。

```
static inline void rtl_push(const rtlreg_t *src1)
{
  // esp <- esp - 4
  // M[esp] <- src1
  cpu.esp = cpu.esp - 4;
  vaddr_write(cpu.esp, *src1, 4);
}
```

rtl_pop: 保存栈顶值，栈顶加四

```
static inline void rtl_pop(rtlreg_t *dest)
{
  // dest <- M[esp]
  // esp <- esp + 4
  // printf("esp: 0x%x\n", cpu.esp);
  *dest = vaddr_read(cpu.esp, 4);
  cpu.esp = cpu.esp + 4;
}
```

overflow carry系列。

无符号减法: 相减结果大于被减数

无符号加法: 相加结果小于其中一个操作数

有符号减法: 异号相减，以及0-INT_MIN有可能溢出。负减正变正，正减负变负，0 - INT_MIN，判断好这三种情况即可。

有符号加法: 同号相加可能溢出。两正加变负，两负加变正，判断这两种情况。由于更新CF OF的指令非常多，因此实现这四个api非常有必要，符合代码重用的原则。

```
static inline void rtl_is_sub_overflow(rtlreg_t *dest,
                                       const rtlreg_t *res, const rtlreg_t *src1, const rtlreg_t *src2, int width)
{
  // dest <- is_overflow(src1 - src2)
  //同号相减不会溢出，正数减负数或者负数减正数
  //排除例外，0 - 最大的负数
  if(*src1 == 0 && *src2 == 0x80000000)
  {
    *dest = 1;
    return;
  }
  t0 = (*src1 >> (width * 8 - 1)) == 1 ? 0 : 1;
  t1 = (*src2 >> (width * 8 - 1)) == 1 ? 0 : 1;
  s0 = (*res >> (width * 8 - 1)) == 1 ? 0 : 1;
  if (t0 ^ t1)
  {
    if (t0 ^ s0)
      *dest = 1;
    else
      *dest = 0;
  }
  else
  {
    *dest = 0;
  }
}

```

```
static inline void rtl_is_sub_carry(rtlreg_t *dest,
                                    const rtlreg_t
*res, const rtlreg_t *src1)
{
  // dest <- is_carry(src1 - src2)
  if (*res > *src1)
    *dest = 1;
  else
    *dest = 0;
}

static inline void rtl_is_add_overflow(rtlreg_t
*dest,
                                       const
rtlreg_t *res, const rtlreg_t *src1, const rtlreg_t
*src2, int width)
{
  // dest <- is_overflow(src1 + src2)
  //两个负数相加，结果变成正数或者两个整数相加，结果变成负数
  t0 = (*src1 >> (width * 8 - 1)) == 1 ? 0 : 1;
  t1 = (*src2 >> (width * 8 - 1)) == 1 ? 0 : 1;
  s0 = (*res >> (width * 8 - 1)) == 1 ? 0 : 1;
  if (t0 ^ t1)
  {
    *dest = 0;
  }
  else
  {
    if (t0 ^ s0)
      *dest = 1;
    else
      *dest = 0;
  }
}

static inline void rtl_is_add_carry(rtlreg_t *dest,
                                    const rtlreg_t
*res, const rtlreg_t *src1)
{
  // dest <- is_carry(src1 + src2)
  if (*res < *src1)
    *dest = 1;
  else
    *dest = 0;
}
```

再说一个有符号数拓展，什么移位操作的弱爆了。我们应该尽可能利用最正确的代码。符号拓展最正确的代码？强制类型转换！

```c
static inline void rtl_sext(rtlreg_t *dest, const rtlreg_t *src1, int width)
{
  // dest <- signext(src1[(width * 8 - 1) .. 0])
  // TODO();
  rtlreg_t tmp = (*src1) & (~0u >> ((4 - width) << 3));
  switch (width)
  {
  case 4:
    *dest = (uint32_t)tmp;
    break;
  case 1:
    *dest = (uint32_t)(int8_t)tmp;
    break;
  case 2:
    *dest = (uint32_t)(int16_t)tmp;
    break;
  default:
    assert(0);
  }
}
```

arith.c

主要是加法和减法，注意更新eflags即可。

control.c

跳转指令

data-mov.c

pusha 和 popa注意一下寄存器的宽度是16还是32

leave把栈顶的值更新为ebp的值，并从栈中弹出ebp的值，注意也要区分寄存器长度。

cltd和cwtl根据伪代码实现就好。movsz和movzx的区别是要不要做符号拓展。lea取地址。

logic.c

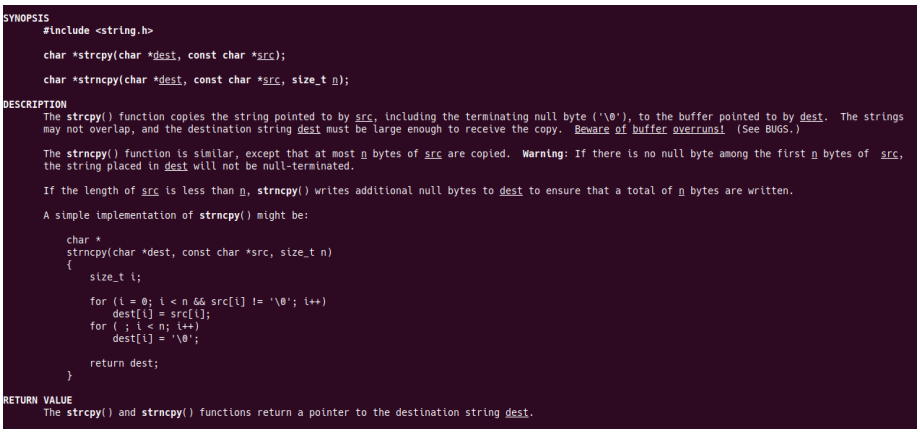
逻辑指令。不用更新OF CF降低了不少难度。

感觉写的有点多了，别的就根据伪代码实现就行了。

实现am中的库函数

实现string.c 和 stdio.c

在实现string.c的时候，有一个比较投机的方法，就是借用库函数的实现。比如我们要实现strcpy我们在终端中输入man strcpy。这里有详细的介绍还有C++的源码。。23333。。然后只能非常难过的拿过来用了。另外str系列指令都可以用mem系列指令实现，就不用写两遍代码了。除了memmove比较难以外，别的指令都是很容易实现的，感觉就是大一C++的基本功吧，这里就不说了。



```
SYNOPSIS
       #include <string.h>

       char *strcpy(char *dest, const char *src);

       char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
       The strcpy() function copies the string pointed to by src, including the terminating null byte ('\0'), to the buffer pointed to by dest.  The strings
       may not overlap, and the destination string dest must be large enough to receive the copy.  Beware of buffer overruns!  (See BUGS.)

       The strncpy() function is similar, except that at most n bytes of src are copied.  Warning: If there is no null byte among the first n bytes of  src,
       the string placed in dest will not be null-terminated.

       If the length of src is less than n, strncpy() writes additional null bytes to dest to ensure that a total of n bytes are written.

       A simple implementation of strncpy() might be:

           char *
           strncpy(char *dest, const char *src, size_t n)
           {
               size_t i;

               for (i = 0; i < n && src[i] != '\0'; i++)
                   dest[i] = src[i];
               for ( ; i < n; i++)
                   dest[i] = '\0';

               return dest;
           }

RETURN VALUE
       The strcpy() and strncpy() functions return a pointer to the destination string dest.
```

stdio.c

sprintf就是一个系统级的工程了。代码超过了两百行，简单说一下思路，主要用到的是编译原理的知识。sprintf的输入可以表示为(s|w)*其中s是其它字符串，w是带百分号的格式化字符串。所以我们的关键在于识别s和w，其中s保留，w有对应的输出替换。检测到%后，后面的字符串开始按照格式化字符串处理。首先是宽度，精度和占位符，然后是后面的整数和字符串，%f太复杂，就没有实现了，而且nemu也不支持浮点。%s可以直接用后面的参数替换。每次拿出一个参数利用宏va_arg(var, type)即可。而且这个操作系统的实验做过了。打印数字因为存在进制的问题，我们需要用到递归，因为转换进制都是模n取余，余数倒排，所以我们编写一个递归的函数，最后的余数最先输出，这样就能实现了。

printf先开一个buffer，然后调用sprintf把字符串打印到buffer中，然后再遍历buffer用_putc()打印出来。

以上runall.sh所需的指令实现完成了。

Diff-test

打开common.h中的diff-test宏

然后实现一个checkregs即可。依次比较qemu中的寄存器与nemu是否相同，如果不同告知错误。

```
1  bool isa_difftest_checkregs(CPU_state *ref_r,
   vaddr_t pc)
2  {
3    for (int i = R_EAX; i <= R_EDI; i++)
4    {
```

```
    if (reg_l(i) != ref_r->gpr[i]._32)
    {
      printf("different at: %s\n", reg_name(i, 4));
      return false;
    }
  }
  if (cpu.pc != ref_r->pc)
  {
    printf("different at pc\n");
    return false;
  }
  return flag;
}
```

再运行一次bash runall.sh还是全部通过。

死循环的检测

如何判断程序进入了死循环呢，如果一个程序反复运行到同一个位置的次数过多，那么可以判断进入了有可能进入死循环。实现方法：哈希表。当程序运行的指令数量超过一定程度时，比如超过了１０２４条，此时开启死循环的检测机制。检测机制如下：每执行一条指令，就在哈希表中记录当时pc的位置并把该位置的计数器加１，当某个位置的计数器超过一万时，将nemu的状态设置为abort，这样就能达到停止nemu的目的。

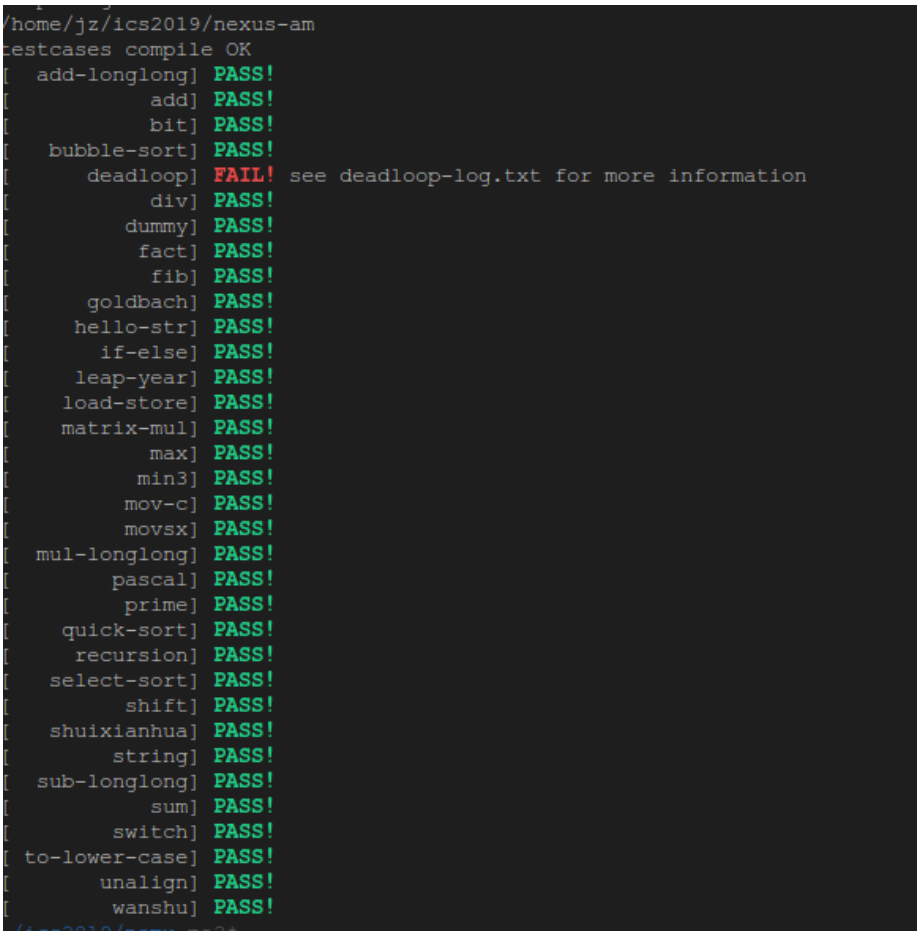
实现代码如下：

```
 if (g_nr_guest_instr == 1024)
 {
     deadFlag = true;
 }
if (deadFlag)
{
    hashTable[cpu.pc % 1024]++;
    if (hashTable[cpu.pc % 1024] > 10000)
    {
        nemu_state.state = NEMU_ABORT;
        fprintf(stderr, "dead loop detected!\n");
    }
}
```

在tests/cputest/中添加如下程序

```c
//deadloop.c
#include "trap.h"
int main()
{
    volatile int i = 1;
    while (i > 0)
    {
        i += 1;
    }
    return 0;
}
```

运行回归测试脚本，除了我们自己编写的死循环失败以外，别的测试均能通过，达到了识别死循环的目的。



输入输出

端口映射和内存映射

一种I/O编址方式是端口映射I/O(port-mapped I/O), CPU使用专门的I/O指令对设备进行访问, 并把设备的地址称作端口号. 有了端口号以后, 在I/O指令中给出端口号, 就知道要访问哪一个设备寄存器了.

内存映射I/O

对内存的访问会映射到IO设备上

`nemu/src/device/io/port-io.c`是对端口映射I/O的模拟. `add_pio_map()`函数用于为设备的初始化注册一个端口映射I/O的映射关系. `pio_read_[l|w|b]()`和`pio_write_[l|w|b]()`是面向CPU的端口I/O读写接口, 它们最终会调用`map_read()`和`map_write()`, 对通过`add_pio_map()`注册的I/O空间进行访问.

比如NEMU的VGA显存位于物理地址区间`[0xa0000000, 0xa1000000)`

- `_DEVREG_VIDEO_INFO`, AM显示控制器信息. 从中读出`_DEV_VIDEO_INFO_t`结构体, 其中`width`为屏幕宽度, `height`为屏幕高度. 另外假设AM运行过程中, 屏幕大小不会发生变化.
- `_DEVREG_VIDEO_FBCTL`, AM帧缓冲控制器. 向其写入`_DEV_VIDEO_FBCTL_t`结构体, 向屏幕`(x, y)`坐标处绘制`w*h`的矩形图像. 图像像素按行优先方式存储在`pixels`中, 每个像素用32位整数以`00RRGGBB`的方式描述颜色.

理解volitile

```
void fun() {
  extern unsigned char _end;  // _end是什么?
  volatile unsigned char *p = &_end;
  *p = 0;
  while(*p != 0xff);
  *p = 0x33;
  *p = 0x34;
  *p = 0x86;
}
```

然后使用`-O2`编译代码. 尝试去掉代码中的`volatile`关键字, 重新使用`-O2`编译, 并对比去掉`volatile`前后反汇编结果的不同.

你或许会感到疑惑, 代码优化不是一件好事情吗? 为什么会有`volatile`这种奇葩的存在? 思考一下, 如果代码中`p`指向的地址最终被映射到一个设备寄存器, 去掉`volatile`可能会带来什么问题?

有volatile

```
void fun() {
   0:   48 8d 15 00 00 00 00    lea
 0x0(%rip),%rdx        # 7 <fun+0x7>
  extern unsigned char _end;  // _end是什么?
```

```
  volatile unsigned char *p = &_end;
  *p = 0;
   7:   c6 05 00 00 00 00 00    movb
$0x0,0x0(%rip)        # e <fun+0xe>
   e:   66 90                   xchg   %ax,%ax
  while(*p != 0xff);
  10:   0f b6 02                movzbl (%rdx),%eax
  13:   3c ff                   cmp    $0xff,%al
  15:   75 f9                   jne    10 <fun+0x10>
  *p = 0x33;
  17:   c6 05 00 00 00 00 33    movb
$0x33,0x0(%rip)        # 1e <fun+0x1e>
  *p = 0x34;
  1e:   c6 05 00 00 00 00 34    movb
$0x34,0x0(%rip)        # 25 <fun+0x25>
  *p = 0x86;
  25:   c6 05 00 00 00 00 86    movb
$0x86,0x0(%rip)        # 2c <fun+0x2c>
}
```

去掉volitile后只剩下*p = 0, 然后是一个死循环，因为编译器判断*p 不会等于0xff，所以把比较给省了。

```
void fun() {
  extern unsigned char _end;  // _end是什么?
  unsigned char *p = &_end;
  *p = 0;
   0:   c6 05 00 00 00 00 00    movb   $0x0,0x0(%rip)
       # 7 <fun+0x7>
   7:   eb fe                   jmp    7 <fun+0x7>
```

当p指向的地址是一个设备寄存器时，该地址值改变不一定来自CPU, 而是有可能来自外界，像上面优化过的代码将不再比较变量的值，无法捕捉到来自设备的变化。另外，编译器会优化对同一个变量的连续赋值，如果对设备也采取了优化，将无法实现比如屏幕上一个像素点连续的色彩变化！

实现in, out指令，并跳过diff-test

```
make_EHelper(in)
{
  switch (id_src->width)
  {
  case 1:
    s0 = pio_read_b(id_src->val);
    break;
  case 2:
    s0 = pio_read_w(id_src->val);
    break;
  case 4:
```

```
    s0 = pio_read_l(id_src->val);
    break;
  default:
    break;
  }

  operand_write(id_dest, &s0);
  print_asm_template2(in);

#ifdef DIFF_TEST
  difftest_skip_ref();
#endif
}

make_EHelper(out)
{
  printf("out addr:%x\n", id_dest->val);
  switch (id_src->width)
  {
  case 1:
    pio_write_b(id_dest->val, id_src->val);
    break;
  case 2:
    pio_write_w(id_dest->val, id_src->val);
    break;
  case 4:
    pio_write_l(id_dest->val, id_src->val);
    break;
  default:
    break;
  }

  print_asm_template2(out);

#ifdef DIFF_TEST
  difftest_skip_ref();
#endif
}
```

运行hello world, printf在之前的sprintf那里实现了。

时钟

在`nexus-am/am/src/nemu-common/nemu-timer.c`中实现`_DEVREG_TIMER_UPTIME`的功能. 在`nexus-am/am/include/nemu.h`和`nexus-am/am/include/$ISA.h`中有一些输入输出相关的代码供你使用.

```c
static struct timeval boot_time = {};

size_t __am_timer_read(uintptr_t reg, void *buf, size_t size) {
  switch (reg) {
    case _DEVREG_TIMER_UPTIME: {
      struct timeval now;
      gettimeofday(&now, NULL);
      long seconds = now.tv_sec - boot_time.tv_sec;
      long useconds = now.tv_usec - boot_time.tv_usec;
      _DEV_TIMER_UPTIME_t *uptime = (_DEV_TIMER_UPTIME_t *)buf;
      uptime->hi = 0;
      uptime->lo = seconds * 1000 + (useconds + 500) / 1000;
      return sizeof(_DEV_TIMER_UPTIME_t);
    }
    case _DEVREG_TIMER_DATE: {
      time_t t = time(NULL);
      struct tm *tm = localtime(&t);
      _DEV_TIMER_DATE_t *rtc = (_DEV_TIMER_DATE_t *)buf;
      rtc->second = tm->tm_sec;
      rtc->minute = tm->tm_min;
      rtc->hour   = tm->tm_hour;
      rtc->day    = tm->tm_mday;
      rtc->month  = tm->tm_mon + 1;
      rtc->year   = tm->tm_year + 1900;
      return sizeof(_DEV_TIMER_DATE_t);
    }
  }
  return 0;
}

void __am_timer_init() {
  gettimeofday(&boot_time, NULL);
}
```

键盘

框架代码在`nemu/include/macro.h`中定义了一个`MAP`宏, 并在`nemu/src/device/kerboard.c`中使用了它. 你能明白它是如何工作的吗?

替换之后结果不变

```
static uint32_t keymap[256] = {
  _KEYS(SDL_KEYMAP)
  // MAP(_KEYS, SDL_KEYMAP)
};
```

键盘

```
1   switch (reg)
2   {
3   case _DEVREG_INPUT_KBD:
4   {
5     _DEV_INPUT_KBD_t *kbd = (_DEV_INPUT_KBD_t *)buf;
6     int t = inl(KBD_ADDR);
7     kbd->keydown = (t & KEYDOWN_MASK) ? 1 : 0;
8     kbd->keycode = t & ~KEYDOWN_MASK;
9     return sizeof(_DEV_INPUT_KBD_t);
10  }
11  }
```

```
❯ make mainargs=k run
# Building amtest [native] with AM_HOME {/home/jz/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
# Creating binary image [native]
+ LD -> build/amtest-native
/home/jz/ics2019/nexus-am/tests/amtest/build/amtest-native
Try to press any key...
Get key: 43 A down
Get key: 43 A up
Get key: 47 G down
Get key: 47 G up
Get key: 60 B down
Get key: 60 B up
Get key: 43 A down
Get key: 43 A up
Get key: 44 S down
Get key: 45 D down
Get key: 44 S up
Get key: 50 K down
Get key: 46 F down
Get key: 45 D up
```

VGA

屏幕大小，在am中提供支持

```c
int screen_width() {
  _DEV_VIDEO_INFO_t info;
  _io_read(_DEV_VIDEO, _DEVREG_VIDEO_INFO, &info, sizeof(info));
  return info.width;
}

int screen_height() {
  _DEV_VIDEO_INFO_t info;
  _io_read(_DEV_VIDEO, _DEVREG_VIDEO_INFO, &info, sizeof(info));
  return info.height;
}
```

同步，在nemu中提供支持

```c
static void vga_io_handler(uint32_t offset, int len, bool is_write) {
  // TODO: call `update_screen()` when writing to the sync register
  if (is_write)
    update_screen();
}
```

PA2到此结束