PA2 实验报告

一、

首先使用 gdb 调试 nemu 探明指令执行的原理。以执行默认程序的第一条指令 movl $0x1234,%eax 为例，进入 exec_once()函数后，首先将当前的 PC 值 cpu.pc 保存到全局译码信息 decinfo.seq_pc 中，然后调用 isa_exec(&decinfo.seq_pc)执行当前指令。进入 isa_exec()函数后，首先调用 instr_fetch(pc, 1)取得指令码的第一个字节，这一步对于所有指令都是相同的，因为 opcode 及 opcode 前面的指令码部分都是一个字节，且这些字节都能在 opcode_table[]中找到对应序号的元素，这些元素的内容决定了读取第一个字节之后的行为。instr_fetch()函数最终调用了 paddr_read(paddr_t addr, int len)函数，计算 addr 在 nemu 的内存数组 pmem 中相对起点的偏移量，然后读取定长数据，然后更新 pc 到 pmem 的新读取起点(*pc += len)。取得 opcode 后 isa_exec()函数将其保存到 decinfo.opcode 中，然后调用 set_width(int width)根据 opcode_table[opcode]的成员 width 设置 decinfo 的 Operand 类型成员 src 和 dest 的成员 width。最后 isa_exec()函数调用 idex(vaddr_t *pc, OpcodeEntry *e)函数完成指令的执行。idex()函数分为两部分：译码和执行，分别由指令的对应 make_DHelper()和 make_EHelper()形式定义的函数完成。opcode 是否需要进行译码是由 opcode 本身决定的，其信息需要根据指令的定义填入 opcode_table[]中。以 movl 0x1234,%eax 为例，其 opcode 为 b8，opcode_table[]中指定了译码函数 make_DHelper(mov_I2r)、执行函数 make_EHelper(mov)、width 为默认值 4，因此 idex()函数会先调用译码函数 make_DHelper(mov_I2r)读取 b8 后面 4 字节的 Imm 字节，完成对操作数的读取并将结果存入 decinfo 的成员 dest、src(部分指令还需要 src2)中,并更新 pc 到内存数组 pmem 的新读取起点，其中 dest、src 是指令执行中的源操作数和目的操作数存放处；然后 idex()函数调用执行函数 make_EHelper(mov)对 decinfo 的成员 dest 和 src 进行计算。最后返回 isa_exec()函数更新 pc 和 decinfo 的成员 sec_pc 到 pmem 的新读取起点，一条指令执行完成。由上面的过程可知应如何实现一条指令：首先根据 i386 手册寻找指令的信息，根据操作数类型确定译码函数、操作数宽度、执行函数，将信息填入 opcode_table[]表中；然后实现对应的译码函数和执行函数。由于译码和执行是分开的，执行函数最后会使用 decinfo 的成员 src、src2、dest 计算，因此同一类型指令的执行函数相同，不同的是译码函数和宽度，他们负责对 decinfo 的成员 src、src2、dest 初始化。

CPU_state 增加标志符寄存器

根据标志寄存器的结构，在 CPU_state 中添加 CF、ZF、SF、OF，根据标志寄存器的结构定义使用位域指定它们在 16 位的 eflags 中的位置，其实现结构和 eax、ecx 等的结构一样，在 union 内再使用匿名 struct，外层 union 命名为 eflags。

二、

指令表层实现

指令的表层实现主要包括 opcode_table[]填表，make_DHelper()、make_EHelper()函数的实现，下面列举出 dummy.c 和 add.c 中所有出现的指令，其中 mov 类型的指令 nemu 已经全部预先实现。

call(0xe8)

在 i386 手册 17.2 节中找到 e8 对应 call 指令中的执行过程，从附录 A 中找到 e8 的操

作数为 Av，A 表示直接寻址，指令没有 ModR/M 字节，操作数直接在指令中给出；v 表示操作数宽度是 16 或 32 位，取决于 opcode 前面的 operand size 属性，因此不必指定 width 的大小。根据以上信息，可以在 opcode_table[0xe8]填入 IDEX(A, call)，make_DHelper(A)尚未实现。根据 call 的直接寻址方式，应该读取 opcode 后面的立即数，然后加上 pc 作为 call 的跳转地址，读取立即数使用操作数辅助函数 make_DopHelper(SI)，跳转地址存入 decinfo 的 jmp_pc 成员中。make_DopHelper(SI)参考 make_DopHelper(I)，instr_fetch()取到的操作数转先存入 op->imm，然后直接赋值给 op->simm，在此函数中 op->width 只有 1 或 4 两种情况，width 为 4 时赋值时无符号数会自动类型转换为有符号数，width 为 1 时需要进行符号位扩展，可以使用 op->simm=(op->simm << 24)>>24 简单实现。make_EHelper(call)根据 i386 手册分两步执行：rtl_push(pc)、rtl_j(decinfo.jmp_pc)，至此 call(e8)的表层实现就完成了。

push(0xff、0x6a、0x68、0x57、0x56、0x55、0x53、0x52、0x51)

0xff 对应 grp5[6]，是指令组 gp5 的第七个指令，指令组是一些 operands 寻址相同的 opcode 码组成的指令组，具体功能由 ModR/M 的 reg/opcode 决定，reg/opcode 此时解释为 opcode，其相当于 index 值用于选择 grp 中的值。grp 指令组选择的 ModR/M 相关代码已给出，只需填表即可。0xff 的操作数为 M，因此 opcode_table[0xff]填入 IDEX(M, gp5)，gp5 由于 opcode_table[0xff]已经给出了操作数，故 gp5[6]填入 EX(push)。此指令的执行过程中，make_EHelper(gp5)会根据 decinfo.isa.ext_opcode 确定应该调用 gp5[]中哪个函数。0x6a 的操作数为 Ib，opcode_table[0x6a]填入 IDEXW(I, push, 1)。0x68 的操作数为 Ib，为立即数，因此 opcode_table[0x68]处填入 IDEX(I, push)，make_DHelper(I)已给出。0x57~0x50 的操作数全部是寄存器，分别对应 8 个寄存器的编码，在 decode.c 文件中发现定义了 make_Dhelper(r)专门用于寄存器译码，因此这一组指令对应 opcode_table[]中全部填入 IDEX(r, push)。make_EHelper(push)使用 rtl_push(&id_dest->val)实现。

pop(0x59、0x5b、0x5d、0x5e、0x5f)

0x58~0x5f 操作数全部是寄存器，分别对应 8 个寄存器的编码，opcode_table[]全部填入 IDEX(r, pop)。make_EHelper(pop)使用 rtl_pop(&id_dest->val)实现

add(0x01、0x03、0x83)

0x01 的操作数为 Ev,Gv，0x03 的操作数为 Gv,Ev，opcode_table[0x01]填入 IDEX(G2E, add)，opcode_table[0x03]填入 IDEX(E2G, add)。0x83 是 grp1 指令组，操作数为 Iv,Ev。opcode_table[0x83]已经填好。add(0x83)对应 gp1[0]，gp1[0]直接填入 EX(add)。make_EHelper(add)参考 make_EHelper(adc)。去除掉 make_EHelper(adc)中加上 CF 的相关代码，并修改相关寄存器即可，标志符寄存器的检测也和 adc 指令中的一样。

sub(0x83)

sub(0x83)对应 gp1[5]，gp1[5]中填入 EX(sub)。make_EHelper(sub)参考 make_Ehelper(sbb)，去除掉其减去 CF 的相关代码，并修改相关寄存器即可，标志符寄存器的检测也和 sbb 指令中的一样。

and(0x83)

and(0x83)对应 gp1[4]，gp1 填入 EX(and)。make_EHelper(and)根据 i386 手册中给出 and 的具体过程，rtl_and()用于计算 id_dest->val 和 id_src-val，然后使用 operand_write()

把结果写入 id_dest，由于 CF、OF 应直接赋值 0，直接使用 rtl_set_CF()和 rtl_set_OF()，ZF 和 SF 的更新使用 rlt_update_ZFSF()。

xor(0x31)

0x31 的操作数为 Ev,Gv，opcode_table[0x31]中填入 IDEX(G2E, xor)，make_DHelper(G2E)已给出，make_EHelper(xor)和 make_EHelper(and)的形式几乎一样，用 rtl_xor()函数代替 rtl_and()函数即可。

jmp(0xeb)

0xeb 的操作数为 Jb，由于 jmp(0xeb)只有 8 位操作数一种情况，opcode_table[0xe8]填入 IDEXW(J，jmp, 1)。make_DHelper(J)和 make_EHelper(jmp)已给出。

jcc(je(0x74)、jne(0x75))

jcc 代表所有的条件跳转指令，0x70~0x7f 的操作数为都为 Jb，opcode_table[]中全部填入 IDEXW(J, jcc, 1)，make_EHelper(jcc)已给出，它用于识别所有条件跳转指令(例如 je、jne 等)并执行。

cmp(0x3b、0x83) 0x39

0x3b 的操作数为 Gv,Ev，opcode_table[0x3b]填入 IDEX(E2G, cmp)，make_EHelper(cmp)和 make_EHelper(sub)行为一样，只是不需要使用 operand_write()写入结果而已。cmp(0x83)对应 gp1[7]，gp1[7]填入 EX(cmp)。

inc(0x47)

0x47 的操作数为寄存器，opcode_table[0x47]填入 IDEX(r, inc)，make_EHelper(inc)和 make_EHelper(add)的实现基本一样，只需要把 rtl_add()换成 rtl_addi()并让其中一个操作数固定为 1 即可。

movzx(0x0fb6)

0x0f 是双字节指令码的转义码，opcode_table[0x0f]已经填好，转义函数已经定义，在双字节指令码表中 0xb6 的操作数为 Gv,Eb，双字节指令表 opcode_table[0xb6]处填入 IDEXW(E2G, movzx, 1)，make_EHelper(movzx)已给出。

lea(0x8d)

0x8d 的操作数为 Gv,M，发现 decode.h 中声明了 make_DHelper(lea_M2G)，因此 opcode_table[0x8d]填入 IDEX(lea_M2G, lea)，make_EHelper(lea)已给出。

test(0x85)

0x85 的操作数为 Ev,Gv，opcode_table[0x85]处填入 IDEX(G2E, test)，make_EHelper(test)和 make_EHelper(and)行为一样，只是不需要使用 operand_write()写入结果而已。

setcc(sete(0x0f94))

setcc 代表所有条件设置指令，sete 的指令码前面有转义符 0x0f，是双字节指令码，操作数为 Eb，decode.h 中发现声明了 make_DHelper(setcc_E)，因此在双字节指令表

opcode_table[0x94]中填入 IDEXW(setcc_E, setcc, 1)，make_EHelper(setcc)已给出。

leave(0xc9)

0xc9 无操作数，不需要译码函数，opcode_table[0xc9]填入 EX(leave)，根据 i386 手册 make_EHelper(leave)使用 rlt_mv()和 rlt_pop()实现。

ret(0xc3)

0xc3 无操作数，opcode_table[0xc3]填入 EX(ret)，make_EHelper(ret)使用 rtl_pop()将地址存入 decinfo 的成员 jmp_pc，然后使用 rtl_j()跳转。

nop(0x90)

0x90 没有操作数，opcode_table[0x90]中填入 EX(nop)，其中 make_EHelper(nop)已经实现。

xchg(0x6690)

前缀 0x66 表明其后面的操作数要从 32 位变成 16 位，0x90 是 opcode，与 nop 指令是同一条指令。

三、

指令的底层实现(rtl 指令函数)

指令的表层实现中调用了大量 rlt_xxx()函数，这些函数负责指令的 RTL 级实现，大部分函数还尚未定义。

rlt_push()

根据 i386 手册，push 指令分两步：esp <- esp – 4、M[esp] <- src1，因此可以调用 rlt_subi(&cpu.esp, &cpu.esp, 4)和 rtl_sm(&cpu.esp, src1, 4)函数实现，其中 rtl_subi()用于寄存器和立即数减法，rlt_sm()用于向 nemu 的内存地址写入数据。

rtl_pop()

根据 i386 手册，pop()指令分两步：dest <- M[esp]、esp <- esp + 4，分别调用 rlt_lm(dest, &cpu.esp, 4)和 rtl_addi(&cpu.esp, &cpu.esp, 4)实现，rlt_lm()用于从 nemu 内存地址读取数据，rlt_addi()用于寄存器和立即数加法。

rtl_update_ZF()

首先调用 rtl_shli(&t0, result, 32 – width * 8)把 result 截断至指定位数存入寄存器 t0，然后根据 t0 是否为 0 设置 cpu.eflags.ZF。

rtl_update_SF()

调用 rtl_msb(&t0, result, widty)读取最高位字节并存入 t0，根据 t0 设置 cpu.eflags.SF。

rtl_msb()

调用 rtl_shri()得到最高位字节即可

rtl_set(get)_ZF(OF/SF/CF)()

这一组共 8 个函数利用了宏定义进行函数定义，在/include/isa/rtl.h 中的宏定义内，rtl_set()的内容为 cpu.eflags.f = *src，rtl_get()的内容为*dest = cpu.eflags.f

rtl_is_add_carry()

传入参数 res 为 src1 + src2，src1 为 src1，由于所有参数都 unsigned int 类型，对于加法 CF 的检测，只要 res < src1 或 src2 就表明发生进位，因此可以调用 rtl_setrelop(RELOP_LTU, dest, res, src1)对 res 和 src1 进行比较，设置 dest 的值。

rlt_is_add_overflow()

传入参数 src1、src2、res 与算式一一对应，OF 的检测是针对补码计算，在无符号数中可以通过检测最高位判断其表示的补码数是否为负数。有符号加法出现 OF 只有 正 + 正 = 负、负 + 负 = 正 两种情况，因此加法 OF 的检测方法为：若 src1 和 src2 同号且 res 与它们异号，则 OF 为 1；否则 OF 为 0。分别调用 rtl_shri()获取最高位、rtl_setrelop()进行比较、rtl_and()进行判断即可实现。

rtl_is_sub_carry()

无符号数减法出现借位只可能是 res > src1，调用 rtl_setrelop()函数即可。

rlt_is_sub_overflow()

有符号减法出现 OF 只有 正 – 负 = 负、负 – 正 = 正 两种情况，因此减法 OF 的检测方法为：若 src1 和 src2 异号，且 res 和 src1 异号，则 OF 为 1，否则为 0。分别调用 rtl_shri()、rtl_setrelop()、rtl_and()函数即可实现。

rtl_setcc()

由于 xcc 和 xncc 两种指令序号相邻，因此可以只实现 cc 类型的指令，ncc 类型的指令根据给出的代码中的 invert 值判断是否取反即可。在 switch 结构中分别调用 rtl_get_xx()函数获取寄存器值，根据 i386 手册中不同 cc 的作用调用 rtl_mv()、rtl_or()、rtl_xor()等逻辑函数可实现。

四、

思考 1：指令 movl -0x8(, %eax, 2), %ebx ## AT&T 的编码

这条指令的形式为 mov m32,r32 ## AT&T，opcode 为 8B，需要 ModR/M、SIB 和 disp32，因此 ModR/M 中 mod=00，reg/opcode=011(%ebp)，r/m=100；SIB 中 ss=01(2)，index=000(%eax)，base=101(%ebp)，因为 SIB.base=101 时 base=%ebp 实际上是代表基地址为 disp32；disp32=fffffff8，因此整个指令为 8B 1C 45 FF FF FF F8

思考 2：临时寄存器在不同函数中分类型使用是否有必要

有必要，如果 make_EHelper()使用了某个 RTL 伪指令寄存器比如 t0，它调用的 rtl_xxx()函数也使用了 t0，当从 rtl_xxx()返回到 make_EHelper()中时，t0 的值可能已经不是 make_EHelper()调用 rlt_xxx()前的值了，会产生错误。

思考 3：是否可以直接通过 gcc 将 C 语言文件编译后交由 NEMU 执行

nemu 无法执行 gcc 直接编译好的文件。nemu 自身编译器编译好的二进制文件结构和 gcc 编译的二进制 ELF 文件完全不同，后者会把代码区直接放在文件开始处，对于全局变

量的存储也和 ELF 文件不同。若 nemu 去执行 ELF 文件，则它会直接从 ELF 文件的第一个字节开始读取指令，但 ELF 文件第一个字节是文件头的内容，无法执行，因此会报错。