PA2:简单复杂的机器：冯诺伊曼计算机系统

181840326 张祎扬 匡亚明学院

实验进度：我完成了全部内容

PA2:简单复杂的机器：冯诺伊曼计算机系统
- 必答题
- 不停计算的机器
- RTFSC(2)
  - 理解指令执行的过程
  - 为什么执行了未实现指令会出现上述报错信息
  - 运行第一个客户程序
- 程序，运行时环境与AM
  - 阅读Makefile
  - 实现更多的指令
  - 实现字符串处理函数
  - 实现sprintf
- 基础设施(2)
  - 实现difftest
- 输入输出
  - 理解volatile关键字
  - 运行Hello World
  - 实现printf
  - 实现IOE
  - 看看NEMU跑多快
  - 神秘的MAP宏
  - 如何检测多个键同时被按下?
  - 实现IOE（2）
  - 实现IOE（3)(4)
  - 展示你的计算机系统
- 做完pa的感想

鉴于这次的PA实验内容过于繁多，我就把必答题放在最开始了。

必答题

- RTFSC 整理一条指令在NEMU中的执行过程。

  其实从开始做PA到现在，我一直坚持一边做实验一边码实验报告，以记录我的一些想法和思路。~~这就是我的实验报告为什么这么死长死长的原因qwq~~

  关于这道题，我想就我做完整个PA2的系统理解与回忆，重新做一个更加简洁的阐述。

  一条指令在NEMU中的执行过程分为以下几个阶段：**取指、译码、执行、更新PC。**

取指阶段主要通过调用函数`instr_fetch()`来从pc指向的下一条指令的位置读取指令长度的字节。

译码和**执行**阶段其实很关键的是opcode_table,将取指阶段取得的指令作为操作码表的索引，这需要我通过RTFM来在opcode_table填入正确的译码函数和执行函数。这其中离不开一些很重要的宏，比如`IDEX`,`make_EHelper`,`make_DHelper`之类的。

译码函数大多数已经由框架代码提供，我所做的主要是根据手册的描述，在操作数表中填入正确无误的译码函数。而执行函数则大部分需要我自己实现，这需要正确理解rtl指令（当然我为此也付出了不少功夫），并且不仅要正确实现rtl指令，还要在执行函数中正确调用它们。

更新PC阶段就很简单了。在`instr_fetch()`被不断调用的过程中，`decinfo.seq_pc`会随着取出指令的长度而增加，以达到记录指令长度的目的，从而可以更新PC。

通过对整个PA2的完成，从简单的指令实现，到更为复杂的IOE的实现，我深刻地体会到了它们本质上还是一个**取指->译码->执行**的过程，这是计算机世界井井有条的最好证明！

- 编译与链接

  去掉static之后，程序可以正常编译。

  去掉inline之后，程序也可以正常编译运行。

  同时去掉static和inline之后，出现**报错**multiple definition.

```
+ CC src/isa/x86/decode/decode.c
+ CC src/isa/x86/decode/modrm.c
+ LD build/x86-nemu
/usr/bin/ld: build/obj-x86/isa/x86/intr.o: in function `interpret_rtl_li':
/home/katherine/ics2019/nemu/./include/rtl/rtl.h:17: multiple definition of `interpret_rtl_li'; bui
ld/obj-x86/cpu/cpu.o:/home/katherine/ics2019/nemu/./include/rtl/rtl.h:17: first defined here
/usr/bin/ld: build/obj-x86/isa/x86/exec/special.o: in function `interpret_rtl_li':
/home/katherine/ics2019/nemu/./include/rtl/rtl.h:17: multiple definition of `interpret_rtl_li'; bui
ld/obj-x86/cpu/cpu.o:/home/katherine/ics2019/nemu/./include/rtl/rtl.h:17: first defined here
```

  解释：static修饰的函数不能被外部文件调用，因此在其他文件中可以定义名字相同的函数。inline关键字用来定义内联函数，函数的代码被放入符号表中，在使用时直接进行替换，减少了调用函数的开销。如果把两者同时去除，就会出现函数多重定义的冲突。

  验证：我编写了一个static inline函数,并且在main函数中调用它，但是可以看到在符号表中没有相应的函数入口，这是因为它已经在对应的部分代码被展开了。

```
Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
```

- 编译与链接

  1. 我用`grep -c -r "dummy"`指令查看，发现重新编译后的NEMU含有83个dummy实体。
  2. 此时的NEMU有84个dummy实体。因为如果有多个弱符号，则选择其中的一个。多出来的一个是在debug.h中加入的。
  3. 发生了报错redefinition of 'dummy'

```
+ CC src/cpu/cpu.c
In file included from ./include/common.h:34,
                 from ./include/nemu.h:4,
                 from ./include/cpu/exec.h:4,
                 from src/cpu/cpu.c:1:
./include/debug.h:9:21: error: redefinition of ‘dummy’
 volatile static int dummy=0;
                     ^~~~~
In file included from ./include/nemu.h:4,
                 from ./include/cpu/exec.h:4,
                 from src/cpu/cpu.c:1:
./include/common.h:20:21: note: previous definition of ‘dummy’ was here
 volatile static int dummy=0;
```

之前对dummy进行声明的时候并没有初始化，它们都是弱符号，编译器会选择其中的一个。但是如果同时对他们进行初始化，那么他们就都变成了强符号，但这是不允许的，会发生连接错误。

- 了解Makefile

  代码前12行的主要作用是输出基本信息（isa和name），以及确保ISA是有效的。

```
INC_DIR += ./include ./src/isa/$(ISA)/include
BUILD_DIR ?= ./build

ifdef SHARE
SO = -so
SO_CFLAGS = -fPIC -D_SHARE=1
SO_LDLAGS = -shared -fPIC
endif

OBJ_DIR ?= $(BUILD_DIR)/obj-$(ISA)$(SO)
BINARY ?= $(BUILD_DIR)/$(ISA)-$(NAME)$(SO)
```

  接下来这里定义了所有include文件和build文件和obj文件和binary文件的绝对路径。然后再往下的`#Compilation flags`定义了编译选项，比如-Werror之类的。

```
41 # Files to be compiled
42 SRCS = $(shell find src/ -name "*.c" | grep -v "isa")
43 SRCS += $(shell find src/isa/$(ISA) -name "*.c")
44 OBJS = $(SRCS:src/%.c=$(OBJ_DIR)/%.o)
45
46 # Compilation patterns
47 $(OBJ_DIR)/%.o: src/%.c
48     @echo + CC $<
49     @mkdir -p $(dir $@)
50     @$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
```

  这一块是比较核心的部分，首先寻找源文件是src目录下的.c文件，忽略isa目录。然后再继续寻找isa目录下你所选择的isa所包含的.c文件。接下来是循环编译过程。先将.c文件编译生成.o文件(CFLAGS),再根据是否需要链接，最后生成可执行文件。

  在后面是一些基本命令。make app run gdb clean之类的。make run是编译并运行，make gdb是使用gdb对可执行文件进行调试，make clean是清除之前的编译。

  然后我使用-n选项来查看make的编译过程，来验证我之前的解读。可以看到一开始make主要用gcc对所有源文件进行预编译。

```
gcc -O2 -MMD -Wall -Werror -ggdb3 -I./include -I./src/isa/x86/include -D__ISA__=x86 -fomit-fra
inter  -c -o build/obj-x86/cpu/cpu.o src/cpu/cpu.c
echo + CC src/cpu/inv.c
mkdir -p build/obj-x86/cpu/
gcc -O2 -MMD -Wall -Werror -ggdb3 -I./include -I./src/isa/x86/include -D__ISA__=x86 -fomit-fra
inter  -c -o build/obj-x86/cpu/inv.o src/cpu/inv.c
echo + CC src/cpu/relop.c
mkdir -p build/obj-x86/cpu/
gcc -O2 -MMD -Wall -Werror -ggdb3 -I./include -I./src/isa/x86/include -D__ISA__=x86 -fomit-fra
inter  -c -o build/obj-x86/cpu/relop.o src/cpu/relop.c
```

然后gcc又对所有的.o文件进行了链接。这是最主要的两个步骤。

```
gcc -O2 -rdynamic  -o build/x86-nemu build/obj-x86/cpu/cpu.o build/obj-x86/cpu/inv.o build/obj-x86/
cpu/relop.o build/obj-x86/device/intr.o build/obj-x86/device/serial.o build/obj-x86/device/keyboar
.o build/obj-x86/device/argsrom.o build/obj-x86/device/device.o build/obj-x86/device/vga.o build/o
j-x86/device/timer.o build/obj-x86/device/io/map.o build/obj-x86/device/io/port-io.o build/obj-x86/
device/io/mmio.o build/obj-x86/monitor/monitor.o build/obj-x86/monitor/cpu-exec.o build/obj-x86/mo
itor/diff-test/diff-test.o build/obj-x86/monitor/diff-test/ref.o build/obj-x86/monitor/debug/log.o
build/obj-x86/monitor/debug/watchpoint.o build/obj-x86/monitor/debug/expr.o build/obj-x86/monitor/
ebug/ui.o build/obj-x86/memory/memory.o build/obj-x86/main.o build/obj-x86/isa/x86/mmu.o build/obj
x86/isa/x86/logo.o build/obj-x86/isa/x86/intr.o build/obj-x86/isa/x86/init.o build/obj-x86/isa/x86/
exec/special.o build/obj-x86/isa/x86/exec/data-mov.o build/obj-x86/isa/x86/exec/arith.o build/obj-
86/isa/x86/exec/prefix.o build/obj-x86/isa/x86/exec/cc.o build/obj-x86/isa/x86/exec/system.o build/
obj-x86/isa/x86/exec/control.o build/obj-x86/isa/x86/exec/logic.o build/obj-x86/isa/x86/exec/exec.o
 build/obj-x86/isa/x86/decode/decode.o build/obj-x86/isa/x86/decode/modrm.o build/obj-x86/isa/x86/
iff-test.o build/obj-x86/isa/x86/reg.o -lSDL2 -lreadline -ldl
```

不停计算的机器

RTFSC(2)

理解指令执行的过程

在`exec_once()`中，框架代码首先将当前的PC保存到全局译码信息`decinfo`的成员`seq_pc`中,然后`decinfo.seq_pc`的地址将被作为参数送进`isa_exec()`函数中。函数`isa_exec()`是一个用于屏蔽ISA差异的API。

取指(instruction fetch, IF)

函数`instr_fetch()`(在`nemu/include/cpu/exec.h`中定义)专门负责取指令的工作。这其中调用了`Vader_read(*pc, len)`函数，其本质就是从内存中pc指针指向的位置读取指定长度的数据。

译码(instruction decode, ID)

实现操作数宽度前缀：先通过`set_width()`函数记录译码查找表中的操作数宽度。如果这个值是0，说明还不能确定，还要继续参考`decinfo.isa.is_operand_size_16`变量来判断。

下面我阅读了mov指令的例子剖析。

简单mov指令的执行。

首先用`instr_fetch()`取得指令的第一个字节`0xb8`,把它存入`decinfo.opcode`中，然后根据它来索引`opcode_table`.然后我查看了`opcode_table`中索引是0xb8的元素，发现是`IDEX(mov_I2r,mov)`.在`nemu/include/cpu/exec.h`中定义了宏`IDEX`,它是宏`IDEXW(id,ex,w)`中w取0的特殊情况，根据上面的分析知道操作数的宽度是4.同时根据`IDEX(mov_I2r,mov)`可以知道取到的是mov指令，作用是把立即数移入寄存器(i to r).

扩展指令有两种方法：

- 使用转义码`0x0f`,`0x0f 0x38`

- 使用`ModR/M`字节中的扩展opcode域，把`ModR/M`字节中的`reg/opcode`域当作opcode的一部分来解释。

I2r指令把寄存器编号也放在指令的第一个字节，可以通过位运算抽取寄存器编号。再使用`instr_fetch()`继续获取立即数。

通过对这个执行环节的阅读和理解，我发现执行环节中定义了**很多很多很多很多的宏，这虽然很令人头疼**，但是我觉得理解这些宏的定义和作用是理解指令执行过程的关键。

复杂mov指令的执行。

这条指令有前缀`0x66`,它的辅助函数`operand_size()`把`decinfo.isa.is_operand_size_16`成员变量设置为true，所以在`set_width()`中会把操作数长度设置为2字节。在识别操作数的过程中，**又是一堆宏！！！！！**我快被宏搞疯了（小声）

根据一般的寻址格式

> addr = R[base_reg] + R[index_reg] * scale_factor + displacement

确定内存地址只需要确定编码在`ModR/M`字节中的`base_reg`,`index_reg`,`scale_factor`和`displacement`四个值。框架代码已经封装好具体的实现细节。

从译码查找表中取得的元素将作为`idex()`函数的参数，它可以进行译码和执行。宏`make_DHelper`定义了译码辅助函数(`nemu/include/cpu/decode.h`)，函数`concat()`的作用是将两个变量名连接起来`(A ## B)`。译码辅助函数的作用是把指令中的信息记录到`decinfo`中。

操作数译码辅助函数通过宏`make_DopHelper`定义，它将操作数的信息记录在结构体`op`中，如果操作数在指令中，就通过`instr_fetch()`把它们从pc所指向的内存位置取出。`load_val`参数用于控制是否将操作数读出到`decinfo`以供后续使用。

执行(execute, EX)

执行辅助函数通过宏`make_EHelper`定义，通过RTL指令描述指令真正的执行功能。框架代码提供了`operand_write()`函数（在`nemu/src/isa/x86/decode/decode.c`中定义）来决定是写入寄存器还是写入内存。

更新PC

在`instr_fetch()`被不断调用的过程中，`decinfo.seq_pc`会随着取出指令的长度而增加，以达到记录指令长度的目的，从而可以更新PC。

RTL指令

在NEMU中，RTL指令有两种。

- RTL基本指令

  特点是不需要使用临时寄存器，属于ISA无关

  在代码中使用它们时无需添加`interpret_`前缀，只需要编写`rtl_xxx`即可

- RTL伪指令

  通过RTL基本指令或者已经实现的RTL伪指令来实现。

  - ISA无关的RTL伪指令
  - ISA相关的RTL伪指令
- 小型调用约定
  - 实现RTL伪指令的时候，尽可能不用`dest`之外的寄存器存放中间结果（`dest`之后会被写入新值，存放的中间结果作为旧址肯定要被覆盖）
  - 实在需要使用临时寄存器时：
    - `ir`只能作为`rtl_li`的目的RTL寄存器
    - `t0`,`t1`只能在RTL伪指令的实现过程中存放中间结果
    - `s0`,`s1`只能在译码辅助函数和执行辅助函数的实现过程中存放中间结果。

实现新指令的步骤

1. 在`opcode_table`中填写正确的译码辅助函数、执行辅助函数以及操作数宽度。
2. 用RTL实现正确的执行辅助函数。

所有定义结构的整理

```
struct ISADecodeInfo{
  bool is_operand_size_16;
  uint8_t ext_opcode;
};

typedef struct{
  DHelper decode;
  EHelper execute;
  int width;
}OpcodeEntry;

typedef union{
  struct{
    uint8_t R_M :3;
    uint8_t reg :3;
    uint8_t mod :2;
  };
  struct{
```

```
    uint8_t dont_care :3;
    uint8_t opcode    :3;
  };
  uint8_t val;
}ModR/M;

typedef union{
  struct{
    uint8_t base    :3;
    uint8_t index   :3;
    uint8_t ss      :3;
  };
  uint8_t val;
}SIB;

typedef struct{
  uint32_t type;
  int width;
  union{
    uint32_t reg;
    rtlreg_t addr;
    uint32_t imm;
    int32_t simm;
  };
  rtlreg_t val;
  char str[OP_STR_SIZE];
}Operand;

typedef struct{
  uint32_t opcode;
  uint32_t width;
  vaddr_t seq_pc;
  bool is_jump;
  vaddr_t jmp_pc;
  Operand src,dest,src2;
  struct ISADecodeInfo isa;
}DecodeInfo;
DecodeInfo decinfo;
```

为什么执行了未实现指令会出现上述报错信息

在`exec_once()`函数中，首先将`decinfo.seq_pc`值设置为`cpu.pc`值，然后调用`isa_exec()`函数。在`isa_exec()`函数中，首先调用`instr_fetch()`函数，取得指令的第一个字节作为操作码，再将其存入`decinfo.opcode`中，然后调用`set_width()`函数设定操作数宽度，用`opcode`值作为`opcode_table[]`的索引。因为执行的是未实现指令，所以在`opcode_table[]`中的索引值是`EMPTY`.然后在`nemu/include/cpu/exec.h`中找到了定义`EMPTY`的宏`#define EMPTY EX(inv)`，而`#define EX(ex) EXW(ex,0)`,所以等价于`EXW(inv,0)`.又`#define EXW(ex,w) {NULL,`

`concat(exec_,ex), w}`，所以等价于`{NULL, exec_inv, 0}`.即解码函数为空，执行函数为`exec_inv`.因此，函数`set_width()`的参数为0，根据上述规则设置操作数宽度。

接着调用`idex()`函数，if语句的判断条件是`(e->decode)`,而根据上面的分析可以知道解码函数被设置为`null`,所以不会进入if语句，直接执行`e->execute(pc)`。其中e是指向结构`OpcodeEntry`的指针。所以我先去研究一下OpcodeEntry.其中的第二个元素是`EHlper execute`,于是我去寻找EHelper的定义。同样在`exec.h`中发现了它的定义`typedef void (*EHelper) (Vader_t *)`，EHelper被定义为一个函数指针，它指向一个函数，这个函数的参数是一个指向地址的指针，函数的返回值是void.所以执行`e->execute(pc)`语句的结果是调用`exec_inv()`函数。又根据宏定义`#define make_EHelper(name) void concat(exec_, name) (Vader_t *pc)`，所以`exec_inv()`函数由`make_EHelper(inv)`定义(在`nemu/src/isa/x86/exec/special.c`中定义)。

在`exec_inv()`函数中，创建了一个temp[]数组，用来存放接下来的指令（8字节），然后按顺序打印出这些指令，也就是最后屏幕上呈现的`invalid opcode`,然后调用`display_inv_msg(cpu.pc)`函数打印提示信息和logo.接着调用`rtl_exit()`函数，把程序的状态设置为`NEMU_ABORT`，让程序执行意外停止。最后调用`print_asm()`函数打印反汇编结果的字符串。

运行第一个客户程序

实现call指令

首先阅读手册，call rel32指令的opcode是`E8 cd`

```
cb, cw, cd, cp: a 1-byte (cb), 2-byte (cw), 4-byte (cd) or 6-byte (cp)
value following the opcode that is used to specify a code offset and
possibly a new value for the code segment register.
```

从这一段简介可以知道操作码`E8`后面跟着的四个字节表示偏移量。

```
IF rel16 or rel32 type of call
THEN (* near relative call *)
   IF OperandSize = 16
   THEN
      Push(IP);
      EIP ← (EIP + rel16) AND 0000FFFFH;
   ELSE (* OperandSize = 32 *)
      Push(EIP);
      EIP ← EIP + rel32;
   FI;
FI;
```

所以在操作码`E8`处实现call指令，同时不需要把操作数长度设为四字节(指令没有`0x66`前缀)。所以在`opcode_table`中需要使用宏`IDEX`,**因为还不确定操作数的宽度**（指令`call rel32`和指令`call rel16`的opcode都是`E8`.)`IDEX`的两个参数分别转到译码函数和执行函数。

译码函数由宏`make_DHelper`定义。观察`decode.c`文件，我发现`make_DHelper()`函数中调用了形式类似于`decode_op_name`的函数，由宏`make_DopHelper`定义。对其中一些名称的解释（比如I，M，E等）参考了manual附录A.

```
E   A modR/M byte follows the opcode and specifies the operand. The operand
    is either a general register or a memory address. If it is a memory
    address, the address is computed from a segment register and any of the
    following values: a base register, an index register, a scaling factor,
    a displacement.

F   Flags Register.

G   The reg field of the modR/M byte selects a general register; e.g., ADD
    (00).

I   Immediate data. The value of the operand is encoded in subsequent bytes
    of the instruction.

J   The instruction contains a relative offset to be added to the
    instruction pointer register; e.g., JMP short, LOOP.
```

从这张图可以看出J的定义很符合call指令的寻址，它就是在eip上加上一个偏移量来计算地址。所以译码函数应该是`make_DHelper(J)`,它调用了`decode_op_SI(pc,id_dest,false)`,于是我发现函数`make_DopHelper(SI)`需要自己填写完整。首先使用`instr_fetch()`来读取指定长度的内存，并且将它保存在`op->width`中。因为SI是带符号立即数，所以可以通过先左移，再符号右移的方式设置simm，于是就要确定左移和右移的位数，这和操作数的宽度有关，如果宽度是4则不需要移动，如果宽度是1则需要移动24位，所以要先根据操作数宽度确定移动的位数，然后再进行左移或右移。

接下来是执行函数，由`make_EHelper`宏定义。通过grep搜索make_EHelper前缀，我找到了函数`make_EHelper(call)`(在`src/isa/x86/exec/control.c`中定义)。然后发现这个call指令需要我自己实现。通过对已经实现的执行函数的观察，我发现这里需要用到rtl指令。我需要做的是把在译码阶段得到的已经计算好的目标地址放入pc中。根据我在ics课上学到的知识，执行call指令主要有以下几个步骤：

> 1. esp-4
> 2. push返回地址（pc的值）
> 3. 更新pc的值（已经计算好的目标地址）

所以要调用`rtl_push()`，定位到`rtl_push()`以后，我发现它也需要我来实现。根据以上分析，就是先把esp的值减4，然后再把要存入的值入栈。

全部实现以后，我尝试编译，但是却发生了报错，原因是没有声明`exec_call()`函数，于是我在`exec.c`文件中声明了宏`make_EHelper(call)`，最后编译成功。

然后我尝试运行dummy，发现invalid opcode的指令发生了改变，观察日志

```
 5 ^[[1;34m[src/monitor/monitor.c,28,welcome] Build time: 16:26:10, Oct 14 2019^[[0m
 6   100000:   bd 00 00 00 00                      movl $0x0,%ebp
 7   100005:   bc 00 90 10 00                      movl $0x109000,%esp
 8   10000a:   e8 05 00 00 00         call 100014
 9   100014:   55 55 89 e5 83 ec 14 68 00              invalid opcode
10 nemu: ^[[1;31mABORT^[[0m at pc = 0x00100014
11
12 ^[[1;34m[src/monitor/cpu-exec.c,30,monitor_statistic] total guest instructions = 4^[[0m
```

发现call指令被成功执行，观察汇编结果，发现下一个需要实现的指令是push.

```
7
8 00100014 <_trm_init>:
9   100014:   55                      push   %ebp
0   100015:   89 e5                   mov    %esp,%ebp
1   100017:   83 ec 14                sub    $0x14,%esp
2   10001a:   68 00 00 00 00          push   $0x0
```

实现push指令

根据讲义提示，只需要实现push r32和push imm32，首先RTFM.手册上对push的介绍是分简略，因为push本来就是一个很简单的指令，先把栈顶指针esp下移，再把需要push的东西写入内存即可。push imm32的开头是0x68,而push r32的指令是以0x50开头的。

```
50 + /r     PUSH r16      2        Push register word
50 + /r     PUSH r32      2        Push register dword
6A          PUSH imm8     2        Push immediate byte
68          PUSH imm16    2        Push immediate word
68          PUSH imm32    2        Push immediate dword
0E          PUSH CS       2        Push CS
```

其中PUSH r32的操作码有些特殊。

```
/r: indicates that the ModR/M byte of the instruction contains both a
register operand and an r/m operand.
```

```
+rb, +rw, +rd: a register code, from 0 through 7, added to the hexadecimal
byte given at the left of the plus sign to form a single opcode byte. The
codes are——

      rb          rw          rd
   AL = 0      AX = 0      EAX = 0
   CL = 1      CX = 1      ECX = 1
   DL = 2      DX = 2      EDX = 2
   BL = 3      BX = 3      EBX = 3
   AH = 4      SP = 4      ESP = 4
   CH = 5      BP = 5      EBP = 5
   DH = 6      SI = 6      ESI = 6
   BH = 7      DI = 7      EDI = 7
```

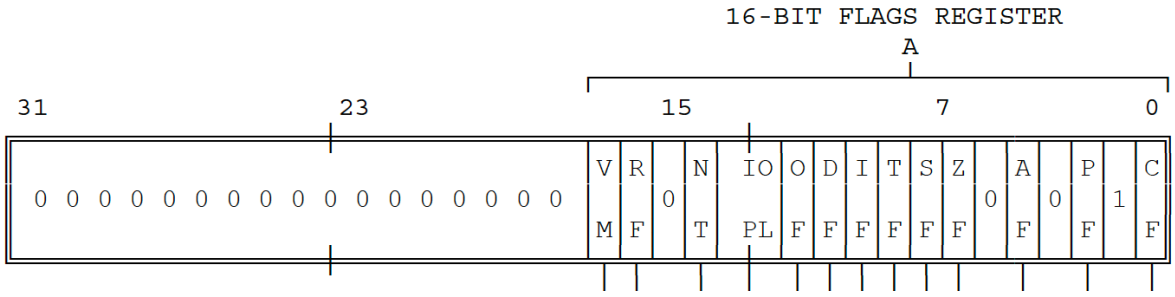
我的理解是50开头的操作码都表示push指令，+表示在原来的指令的十六进制基础上加上一个数（在0～7之间，分别表示不同的寄存器），从图上可以看出，在程序dummy中，我要实现的指令是0x55,它等于0x50+5,而32位寄存器中编号为5的正是寄存器ebp，就是要执行的指令push ebp。所以我觉得在这里关键是要对该一字节操作码的第二位十六进制数进行识别，以判断到底是哪个寄存器。由于在上一个call指令时我已经实现了`rtl_push`,所以现在其实只需要完成一个对寄存器的判断即可。在具体实现中，指令0x50-0x57都是相同的。区别只在于寄存器不同，可以通过取操作码的低8位来实现读取寄存器编号。

在解码时主要调用现成的`make_DHelper(r)`，它调用了`make_DopHelper(r)`,根据代码中的注释，这个函数是用来解码操作数中编码的寄存器的。它正是通过取了操作码的后三位来读取寄存器编码。并将它存储在操作码的元素reg中。接下来实行执行函数，发现push在`data-mov.c`中定义。实现它很简单，只需要在里面调用`rtl_push`即可。

实现push imm32就很简单，它以68开头，后面紧跟着一个字节的立即数。我用grep搜索到在decode.c中定义了`make_DHelper(push_SI)`,它调用了`decode_op_SI`(之前已经实现过了)。执行函数直接调用上面已经实现的make_EHelper(push)`即可。最后编译时同样是遇到了没有声明的问题，声明后得以解决。继续执行dummy，这一次卡在了0x83上，是sub指令。

实现sub指令

根据讲义提示，我需要先实现EFLAGS寄存器。



这是手册中对EFLAGS寄存器的定义，因为我们只要实现CF,ZF,SF,IF,OF,所以只要实现相应的位数即可。（这里严格按照手册上各标志位的位置来实现）。继续查阅手册可以知道`0x83`开头的sub指令表示用寄存器或内存减去一个立即数（符号扩展））。所以在解码的时候可以调用现成的函数`make_DHelper(SI2E)`。

我在`exec.c`中发现了类似这样的表格结构和定义它的宏，也和0x83有关。

```
15 #define make_group(name, item0, item1, item2, item3, item4, item5,
    item6, item7) \
16   static OpcodeEntry concat(opcode_table_, name) [8] = { \
17     /* 0x00 */  item0, item1, item2, item3, \
18     /* 0x04 */  item4, item5, item6, item7  \
19   }; \
20 static make_EHelper(name) { \
21   idex(pc, &concat(opcode_table_, name)[decinfo.isa.ext_opcode]);
    \
22 }
23
24 /* 0x80, 0x81, 0x83 */
25 make_group(gp1,
26     EMPTY, EMPTY, EMPTY, EMPTY,
27     EMPTY, EMPTY, EMPTY, EMPTY)
```

我发现它把操作码进行了分组，并且索引是decinfo.isa.ext_code.但我不知道这个变量的含义，于是我用grep搜索，在`decode/modrm.c`中找到了它，于是我继续RTFSC.在函数`read_ModR_M()`中设置了`ext_opcode`,它等于ModR/M的中间三个字节，用来进一步指示操作码信息。这就可以解释为什么会有如上的类似于group的结构了。以0x80,0x81,0x83开头的操作码应该都有类似的特征。在阅读sub指令的手册时，我发现opcode有一个/5，继续RTFM，它对数字有这样的解释：

> /digit: (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

Reg field包含对指令进行解读的进一步信息，这刚好就是上面提到的ext_opcode,由此可以看出，sub指令应该被填入如上的gp1中，并且因为数字是5，所以应该填在第六个的位置，而根据宏的定义，在原来的EMPTY位置的执行函数只需要填gp1即可。下一步就是完成sub指令的rtl指令，然后完善各处的代码。

这其中最麻烦的是设置各种标志位。好在我在`rtl.h`中看到了定义，我只需要填写其中的`TODO()`即可。

`rtl_sub`由宏`interpret_rtl_sub`定义，而它又由宏`make_rtl_arith_logic`定义，在这里面我又遇到了一个陌生的宏`c_sub`，它在`c_op.h`中定义，即表示a-b。

sub指令影响的标志位是ZF,SF,OF,CF.ZF当结果是0时置1，SF和结果的最高位相同，当发生溢出时OF置1，当发生借位时CF置1.

由于我需要赶一下进度，所以我先不写标志位的设置，先执行一个sub就算了嘤

实现xor指令

首先查阅手册，发现xor把标志位CF和OF置零，对其他标志位也有影响。（我下次再实现这个）

```
30  /r      XOR r/m8,r8      2/6      Exclusive-OR byte register to r/m byte
31  /r      XOR r/m16,r16    2/6      Exclusive-OR word register to r/m word
31  /r      XOR r/m32,r32    2/6      Exclusive-OR dword register to r/m dword
32  /r      XOR r8,r/m8      2/7      Exclusive-OR byte register to r/m byte
```

操作码是0x31,需要进一步确认操作数宽度。因为目的操作数是r/m，源操作数是r，所以译码函数应该是G2E。

实现ret指令

ret指令与call相对，先出栈，再把esp指针的值+4.因为ret指令直接返回，不需要解码获取更多的信息，所以译码函数为空，应该使用宏EX。（我又又又又忘记声明了）

Obviously, 实际的过程并没有实验报告看起来这么顺利，思路这么流畅，但是它最终终于出现了

```
Building x86-nemu
[src/monitor/monitor.c,48,load_img] The image is /home/katherine/ics2019/nexus-am/tests/cpu
[src/memory/memory.c,16,register_pmem] Add 'pmem' at [0x00000000, 0x07ffffff]
[src/monitor/monitor.c,20,welcome] Debug: ON
[src/monitor/monitor.c,23,welcome] If debug mode is on, A log file will be generated to rec
o a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 21:03:07, Oct 18 2019
Welcome to x86-NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at pc = 0x00100024
```

害，我只想说，**在tmux各窗口间来回切换好烦，用grep好烦，打开文件好烦……**，我可能要考虑装一个vscode（可能只是考虑）。

程序，运行时环境与AM

在修改了`Makefile.check`之后，执行`make ALL=dummy run`发生了如下报错：

```
katherine@debian:~/ics2019/nexus-am/tests/cputest$ make ALL=dummy run
/home/katherine/ics2019/nexus-am/Makefile.check:24: .mk: No such file or directory
make: *** No rule to make target '.mk'.  Stop.
```

在求助了两个同学以后，我仍然不知道这是为什么，好像根本没识别出我修改后的x86.

有一个同学和我遇到一样的情况，但是另一个同学却可以成功run起来，不过他用的系统是ubuntu……害，真是让人头疼。但是本着代码写不完的情况，和在不影响大局的多一事不如少一事的KISS原则，我决定暂时忽略这个问题，还是用原来的命令运行测试用例。

目前的解决办法是用export设置环境变量。

阅读Makefile

首先，Makefile对我来说一直是比较难理解的一部分，因为我对于它的了解并不多，我也需要一个机会来RTFM。不如就等我写完代码再看吧（（

实现更多的指令

我本来以为这个任务不是特别重的，直到我用ls命令查看了tests目录下的测试用例个数…… **R U SERIOUS?**

先从`add.c`开始吧

第一个`invalid opcode`位于`0x100038`，是`0x8d`开头的**lea**指令。我在`decode.c`中发现了相应的译码函数`make_DHelper(lea_M2G)`,于是可以在`opcode_table[]`中直接填写相应的译码函数。（同样地，指令没有指定操作数的宽度，所以需要用到宏`IDEX`),又发现`make_EHelper(lea)`也已经实现了，所以直接填入`opcode_table`即可。

然后到了`0x83`开头的**and**。`opcode`的说明中有/4，所以类比之前实现的sub，应该把and添加到group1中。并且放在第五位的位置。只需要实现`logic.c`中的`make_EHelper(and)`即可。

接着是`0xff`开头的**pushl**。手册opcode中有/6,参考代码可知要放在gp5的位置.因为push是一元操作，所以解码函数直接调用`make_DHelper(E)`即可。（`opcode_table`中已经写好），push指令之前也已经实现。

然后是`0x01`开头的**add**。根据描述，是ADD register to r/m，所以是add G to E，在译码时应该调用`make_DHelper(G2E)`,执行时调用`make_EHelper(add)`，所以要用rtl指令实现加法。

然后是`0x3b`开头的**cmp**。手册的描述是compare r/m to register,所以解码应该调用`make_DHelper(E2G)`.根据手册对cmp的描述，cmp是对两个操作数进行减法，但是不存储结果，只是根据结果设置eflags寄存器的值。下一条指令`sete`会根据cmp的结果来进行设置，所以我意识到**从这里开始必须实现EFLAGS寄存器标志位的设置**。

插一句题外话，做到这里我碰巧重启了一下虚拟机，一想到我要重新打开tmux，打开那么多文件，我就非常暴躁！于是我果断在虚拟机里安装了VScode（整个世界都好了）

实现寄存器标志位的设置对我来说其实不算一个特别轻松的过程，在完成代码之前，我查阅了很多资料和手册，甚至翻出了以前暑假汇编程序设计的PPT，但我仍然不确定它正确与否。不过这一过程确实**巩固了我对有符号数、带符号数加减法的理解和对补码运算的理解**。我想我以后仍然会感激这个过程。

在实现完标志寄存器之后，就可以实现cmp指令了，并且我将之前已经实现的指令都加入了标志寄存器的设置。

下一个指令是`0x0f 0x94`的**sete**，是两字节指令，需要填写第二个`opcode table`。根据手册中的r/m8,解码函数应该调用`make_DHelper(E)`,并且操作数宽度设为1.

下一个指令是`0x0f 0xb6`的**movzbl**，是mov指令的零扩展。其中的执行函数`make_EHelper(movzx)`已经实行好，解码函数只需调用`make_DHelper(mov_E2G)`.

emm之后的指令实现过程我就不赘述了.....

当实现了一部分指令之后，我突然发现自己竟然**hit bad trap**了，然后我调出了nemu-log，发现在je指令出现了问题，它并没有按照预期，然后我通过一系列printf，发现最终问题出在函数`rtl_update_ZFSF()`上，我将-1改成了0xffffffffffffffff之后解决了这个问题。

在这之后都比较顺利，最后终于hit good trap了。

一点感想：在指令的实现过程中，它是一个环环相扣的过程，之前的一点点错误都可能造成后面的不成功，尤其是标志寄存器的设置，因为它的结果会影响到一系列的条件相关的指令，如果实现不正确，会有很大的麻烦。在这次debug的过程中，我就找到了之前的一点错误。鉴于我本身对自己写出的标志寄存器相关的代码就不是很确信，我相信以后一定会逐渐发现更多的错误。但同时，**通过实现指令，并观察指令有没有按照我们的预期实行，正是一种检验自己之前实现的代码是否正确的方法**。我充分体会到了这些代码之间的环环相扣，并且成体系的特点。

然后实现bubble-sort.c

第一个停顿的指令是xchg,但是该指令的内容是与自身交换，所以其实相当于什么也没有执行，只是稍作停顿。`0x66`在操作表中指向operand_size,而它的作用是读取下一个字节，所以我将`0x66`的下一个字节`0x90`对应的操作码表处的指令设置为nop，也就是停顿一下。

实现着实现着，它又又又hit bad trap了。

debug了半天，我并没有找到问题在哪里，所以我决定先往后实现。连续通过了几个测试用例，当我第一次运行min3.c的时候，它直接hit bad trap了，所以我觉得可能是这两个测试用例中的共同指令的实现出现了错误。实际上，之前没有找到bubble-sort.c中的bug是因为我查看了nemu-log.txt,发现它执行了将近4000条指令……不断地循环，很难找到问题在哪里。于是我期望min3的log短一点。

if-else.c又hit bad trap了！但是它的log只有50行！我决定实现完所有指令后，就从它入手开始找bug!

实现字符串处理函数

这些都是很简单很基本，**程序设计基础课上联系过无数次的**库函数，在这里就不做解释了。反正实现完以后，string.c确实hit good trap了。

实现sprintf

我通过RTFM了解了`va_start()`,`va_end()`的用法。根据已经提供的参数列表，发现在函数`vsprintf()`中提供了参数`va_list ap`，所以先实现`vsprintf()`函数，再在实现`sprintf()`时调用它。

在RTFM以后，我已经对`stdarg.h`提供的宏有了一个大致的了解。鉴于我只打算先实现%s和%d，关于%s则没什么难点，只涉及字符串的拼接。在实现%d时，则要注意把int型变量转化为字符串，我在这里使用了ascii码的加减，同时要注意负数的转化，先在字符串中加入'-'，然后再把num变正，然后再继续后续操作。

在实现完sprintf以后，我发现我的hello-str不再是hit bad trap而是invalid code。于是我继续实现目前invalid的指令。这是一个jmpcc指令。

实现完该指令之后，运行hello-str竟然good trap了！

基础设施(2)

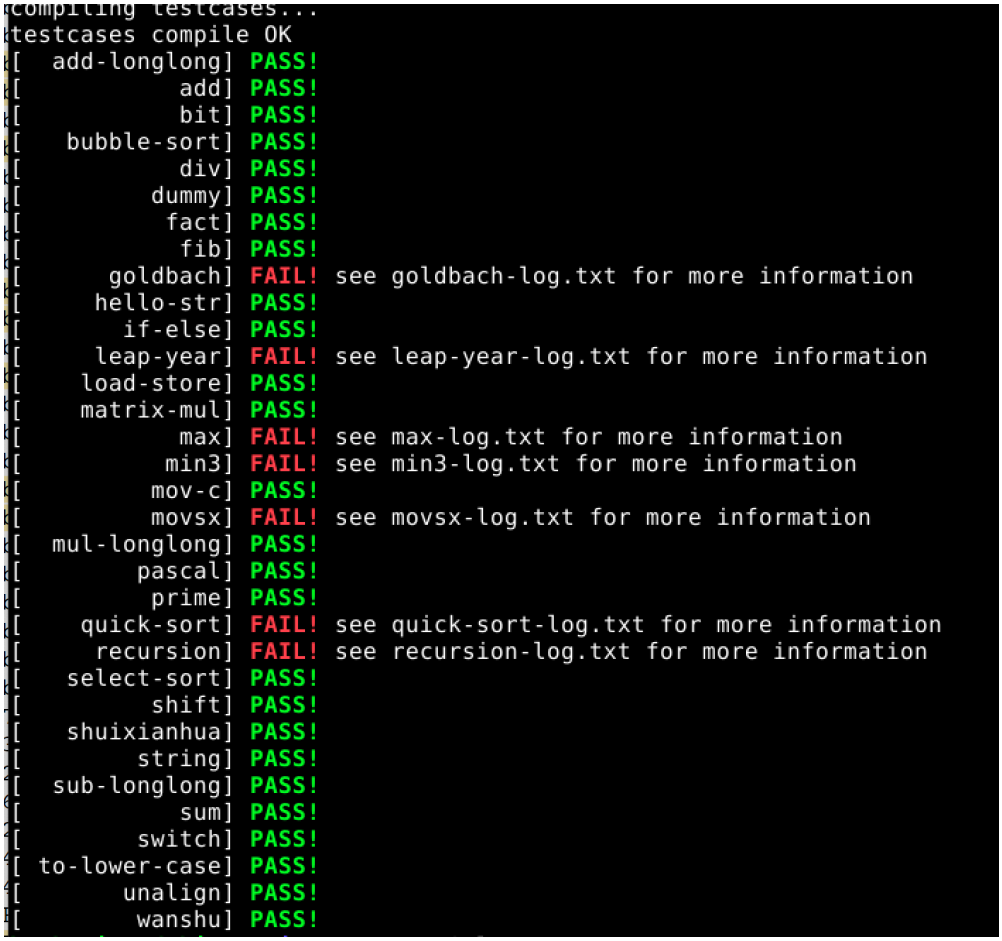
在了解了如何把库函数的调用链接到我自己编写的klib以后，我在编译到native时把库函数链接到klib，用以确认我自己编写的库函数是否完全正确。

在运行完`hello-str`和`string`以后都没有发生报错，这让我对我自己编写的库函数产生了**莫大的信心**。

实现difftest

在阅读讲义后，我了解了difftest的大致原理，并准备着手实现它。在提供的接口中，qemu的寄存器由`ref_r`指针指向，我按照顺序比较`ref_r`指向的寄存器的值和cpu中对应的寄存器的值，如果不一致则将flag改为false，并且输出两个寄存器中分别的值。最后再比较pc。

实现完diff-test以后，我尝试第一次一键回归，并得到了下面的结果（一共七个样例没通过）：



接下来就是针对没有通过的用例，阅读它们的日志文件，并寻找指令实现中的错误。

我先从goldbach下手，我查看它的日志文件，找到了如下的错误：

```
17 cpu.pc:0x10008b ref_r.pc:0x10008c
18 eax 0x0         0
19 ecx 0x3         3
20 edx 0x1         1
21 ebx 0x6         6
22 esp 0x108fa4    1085348
23 ebp 0x108fb4    1085364
24 esi 0x4         4
25 edi 0x6         6
26 pc  0x10008b    1048715
27 nemu: ^[[1;31mABORT^[[0m at pc = 0x00100089
28
```

我发现是两种状态下的pc不一样，即下一条指令的地址不一样，而且只相差1.

于是我又去查看了goldbach的汇编代码，发现qemu的pc值是0x10008c,是下一条指令的首地址，而nemu的pc值是0x10008b,显然上一条指令并没有完全解码，于是我考虑是解码函数出了问题。

```
76   100084:   83 f9 02                cmp    $0x2,%ecx
77   100087:   74 1d                   je     1000a6 <goldbach+0x36>
78   100089:   f6 c1 01                test   $0x1,%cl
79   10008c:   74 52                   je     1000e0 <goldbach+0x70>
80   10008e:   bb 02 00 00 00          mov    $0x2,%ebx
```

我查看test指令的手册，发现f6开头的指令的instruction是`TEST r/m8,imm8`,故解码函数应该是`make_DHelper(I2E)`,而我在相应的位置填写的解码函数是`make_DHelper(E)`,所以发生错误，少读取了一字节指令。

修改过来之后，这个测试用例hit good trap了。

再一次一键回归之后只剩五个样例没有通过了。于是我从max.c入手，继续寻找错误。

这一次定位到错误出现在条件跳转处，我仔细检查了条件跳转中对标志位的判断，发现没有错误，于是错误只能出现在对标志寄存器的设置上。因为cmp指令是做减法，并且判断GE只涉及标志位OF和SF，而SF值不太可能设置错，所以问题应该出在`is_sub_overflow()`上，经过仔细分析和printf的帮助，我最终找到了问题所在——**在使用临时寄存器时，没有充分考虑前后关系，导致下面使用的临时寄存器中的值已经被更改，而不是我期望的值**，在调整临时寄存器的使用顺序以后，问题得到了解决。

再次一键回归测试，发现还有四个未通过用例。接下来查看min3.c.发现是jcc的一个指令没有填入opcode_table.

再次一键回归测试，发现还有三个未通过用例。接下来查看movsx.c.定位到是movsbl指令出了问题，它并没有对要移动的数进行符号扩展。所以应该是我的符号扩展实现出现了问题，于是我重新检查并修改了`rtl_sext()`指令。我发现之前忽略了宽度为8的符号扩展。

再次一键回归测试，发现还有两个未通过用例。（为什么我不能一次性解决好几个qwq）查看quick-sort.c，发现有一个jmp指令没有实现。害。

最后一个bug我真的找了超级久！！！看log上一直是栈指针有问题，一开始我以为是push指令没写对，反反复复看了很久都没有找到问题在哪。最后我发现是call_rm指令忘记写rtl_push了……**我打我自己！！！！**

我终于实现了所有指令！！感谢一键回归测试！！！

一点感想：difftest实际上对于我来说是一根救命稻草。因为我在之前实现更多指令的时候并没有死磕到底，而是往后翻看了讲义，然后就发现了difftest的妙用。实际上，它为我节省的时间是不可估量的。首先它避免了我一个命令一个命令地运行，可以直接批量测试。其次，它使用qemu与nemu进行对比，有一个绝对正确的参照。另外，因为它在状态开始不一致时就停止，所以我很容易定位到最开始出错的地方，而不是等它像滚雪球一样越滚越大，完全偏离了一开始的错误点。在没有difftest的时候，经历了整整三个多小时的折磨，我终于在凌晨三点找到了一个bug，而那时候我的差不多已经崩溃了（笑）。在nemu log中看到的异常，和真正的错误，可能真的隔了十万八千里……

输入输出

理解volatile关键字

volatile关键字的作用是作为指令关键字，确保本条指令不会因为编译器的优化而省略，而是要求每次直接读值。volatile变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。这其实非常必要。假如p指向的地址最终被映射到一个设备寄存器，而去掉volatile的话，这个地址假如被意外改变，就不能访问设备寄存器了。为了保证io实现的正确性，volatile关键字是非常必要的。

运行Hello World

实现`in`,`out`指令并不难，我在`system.c`中找到了它们，实现它们只需要根据操作数宽度分别调用不同的`pio_read`和`pio_write`函数即可。我用一个`switch-case`实现了它。但是实现完`in`和`out`以后我在运行时还遇到了invalid opcode，于是我又多填写了一点opcode table.

感觉遵循KISS法则的结果就是不断在运行新的测试的时候发现自己需要实现更多指令

实现printf

由于我已经实现了`sprinf()`，它是把指定格式化内容读入到一个字符串中。而拥有了`_putc()`,可以使输出到屏幕上，就只需要对字符串使用`_putc()`即可。代码总体与sprintf相同，只不过多了一步输出。

关于简洁地实现，因为我在实现`sprintf()`的时候调用了`vsprintf()`，所以我的`sprintf()`其实不算太繁琐。当然我可以选择直接调用`sprintf()`，但在我看来不是特别有必要。

实现IOE

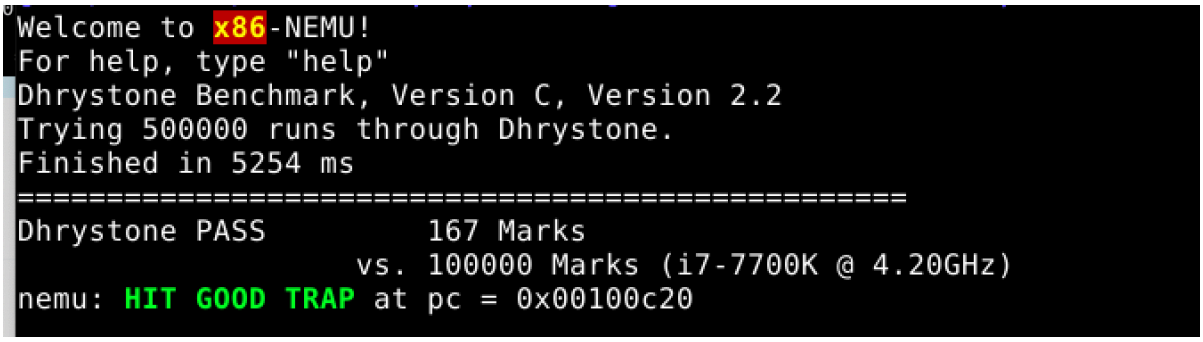
我阅读了`nemu.h`,在其中找到了定义端口地址的宏`RTC_ADDR`。接着阅读`x86.h`,发现其中有很多诸如`asm volatile()`的代码，通过STFW,我了解了这是一种内联汇编的用法。然后我调用`x86.h`中的`inl`来获取指定地址的数据。我注意到下面还有一个`__am_timer_init()`函数与`__am_timer_read()`函数相对，所以在两个函数中分别调用`inl`来读取指定端口的值，它们的差就是AM的启动时间。

在我run `real-time clock test`的时候，我又发现了新的invalid opcode.处理完之后，我观察到程序每相隔一秒在终端输出一句话。

看看NEMU跑多快

- dhrystone

  运行dhrystone之后，跑分167

- coremark

  首次尝试运行coremark，发现还有很多没有实现的指令（KISS法则的坑）。于是我先忙着实现指令去了……而且其实还不少qwq。哇，我真的实现了很久！！好多没实现的指令啊……

最后跑分426

```
Welcome to x86-NEMU!
For help, type "help"
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 6851
Iterations       : 1000
Compiler version : GCC8.3.0
seedcrc          : 0x4x
[]crclist       : 0x4x
[]crcmatrix     : 0x4x
[]crcstate      : 0x4x
[]crcfinal      : 0x4x
Finised in 6851 ms.
==================================================
CoreMark PASS       426 Marks
                vs. 100000 Marks (i7-7700K @ 4.20GHz)
nemu: HIT GOOD TRAP at pc = 0x00101f18
```

- microbench跑分643

```
Welcome to x86-NEMU!
For help, type "help"
Empty mainargs. Use "ref" by default
======= Running MicroBench [input *ref*] =======
[qsort] Quick sort: * Passed.
  min time: 501 ms [1020]
[queen] Queen placement: * Passed.
  min time: 589 ms [799]
[bf] Brainf**k interpreter: * Passed.
  min time: 3366 ms [703]
[fib] Fibonacci number: * Passed.
  min time: 7173 ms [394]
[sieve] Eratosthenes sieve: * Passed.
  min time: 5911 ms [665]
[15pz] A* 15-puzzle search: * Passed.
  min time: 1370 ms [327]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 1168 ms [931]
[lzip] Lzip compression: * Passed.
  min time: 1659 ms [457]
[ssort] Suffix sort: * Passed.
  min time: 525 ms [857]
[md5] MD5 digest: * Passed.
  min time: 6073 ms [283]
==================================================
MicroBench PASS        643 Marks
                  vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 33215 ms
nemu: HIT GOOD TRAP at pc = 0x00103870
```

在这里我不得不多吐槽几句，microbench简直对我是个大坑，我在它身上至少耗费了三天。因为按照讲义，它有一个test模式可以用来测试正确性，但出于某种奇奇怪怪的原因它出了一点问题，这就导致我在测试阶段都只能选择完整地跑microbench，更别说开difftest了。更加好巧不巧的是我在实现指令的时候有一点问题，然后就一直卡在debug的阶段。

最后事实证明这个错误和我之前在跑cputest的时候类似，因为pc的地址在指令中间，所以考虑是译码函数出了问题，decoder函数填错了。**在这里我要感谢屈道涵助教对我非常耐心的帮助！**我也深刻地认识到了自己的不足。以及当一个bug卡了你很久的时候，de出它的开心程度真的很难用语言形容。

神秘的MAP宏

在macro.h中，定义了宏`MAP(c,f) c(f)`。在`keyboard.c`的枚举型中调用`MAP(_KEYS, _KEY_NAME)`则被替换为`_KEYS(_KEY_NAME)`,又根据keyboard.c中定义的宏，将被替换为一系列`_KEY_NAME()`，括号内是键的名称。又根据keyboard.c中定义的宏`_KEY_NAME(k) _KEY_##k`，则将其替换为一系列`_KEY_name`，name是一系列键的名称。

通过以上操作，就将所有键名统一成和`_KEY_NONE`相同的格式，并统一放入枚举类型中，这样就可以给每个键名一个唯一的编号。

同时，keyboard.c中又在数组keymap[256]中调用了`MAP(_KEYS, SDL_KEYMAP)`，它被替换为`_KEYS(SDL_KEYMAP)`,然后又被替换为`SDL_KEYMAP()`,括号中是一系列键的名称。又根据在keyboard.c中定义的宏，被替换为`[SDL_SCANCODE_name]=_KEY_name`。通过以上操作，对数组进行分割。

如何检测多个键同时被按下?

键盘码其实是枚举类型，所以我猜测是用一个for循环来检测?

实现IOE（2）

只需要完善两个表达式分别求`keydown`和`keycode`的值，和之前实现timer的方法差不多。

实现IOE（3)(4)

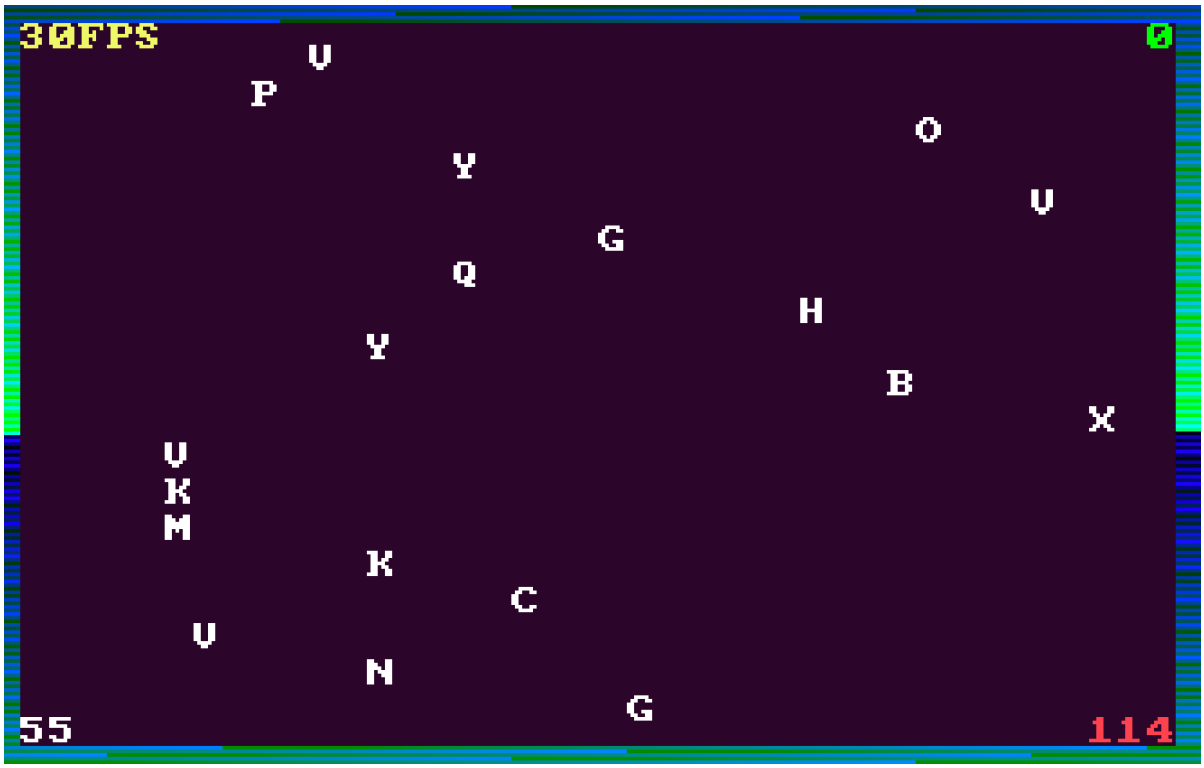
实现VGA的过程确实难到了我，我花了很久去理解那个绘制w*h的矩形矩阵的部分。然后其实实现的过程也很痛苦，我对于讲义一开始所说的**VGA设备还有两个寄存器**一直不是很理解，也完全不知道在哪里写代码，怎么下手。后来我想到可以花一些时间研读一下native的代码，这一读我才发现它们竟然这么相似！有了native的代码作为参照，理解也不是那么困难了，最终我还是将它实现了。

展示你的计算机系统

- 幻灯片播放很成功，只是不够清晰就是了

我好想吐槽这个丑陋的幻灯片qwq

- 打字小游戏运行也挺流畅的



做完pa的感想

1. difftest真的是一个超级好用的东西，可以说它为我的cputest的debug过程帮了很大的忙。同时它其中所蕴含的思想虽然不复杂，但是却是一个普遍适用的法则。
2. 计算机其实并不神秘，仔细体会就会发现看起来再复杂的机器，其本质也就是那几个过程。这是计

算机世界高度统一的规律，我惊呼它的神奇，它可以做到又是那么复杂精巧，可本质又是那么精简。

3. 非常感谢助教小哥哥和身边同学对我的帮助，感谢无数个熬夜的凌晨，让我最终走完了这个**一颗星难度**的无比痛苦的地狱般的PA2.但我自己其实知道我的掌握度还是不够的，我会继续消化这些知识，我知道以后在我回顾往事的时候，会无比感谢这段经历。
4. PA2，再见。