

PA2实验报告

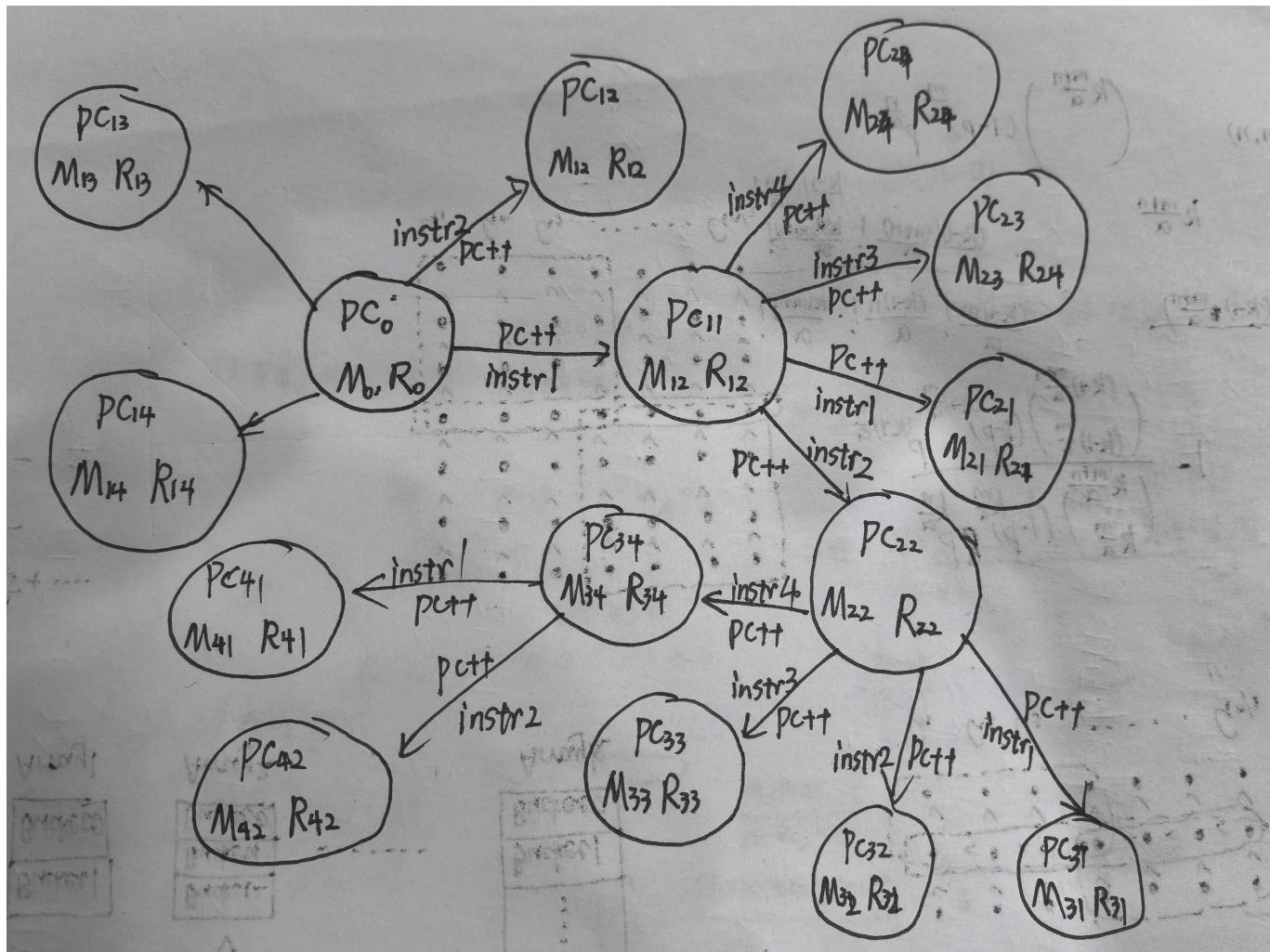
231300027朱士杭

实验进度

已经完成了所有除声卡外的device设备内容，可以正常运行红白机与超级玛丽，klib中的stdio.h和string.h测试完毕，am-kernels里面的测试用例全都测试过，已完成PA基本要求

必答题

程序是个状态机



如图所示，YEMU上执行加法程序的状态机（因为画的比较丑因此稍微看一眼就行）

pc=0, 初始化从M[0]开始执行程序，先取值将`0b11100110->this.inst`获取到了指令，然后操作码译码操作，由于是小端模式，低地址有效字节存储地址（但是这里不是非常严谨，因为这里的低地址有效字节指的是0b11100110中的10，只有2bit而不是1B，有一点点小trick但是在NEMU上面是没有这个问题的），因此`1110->op, 0110->addr`，接下来进行执行指令的操作，将`R[0] <- M[6]`，从而实现了load指令的操作

接下来将pc++更新pc，开始执行下一条指令，`0b00000100->this.inst`译码之后得到`0000->op, 01->rt 00->rs`，然后执行`R[01]<-R[00]`，即`R[1]<-R[0]`，之后再将pc++更新pc，开始进入下一个for循环

执行下一条指令.....

于是乎周而复始，这不正符合冯诺依曼思想——**存储程序，顺序执行！！！**

请整理一条指令在NEMU中的执行过程

我们从cpu_exec函数开始说起吧，我们在pa1当中的RTFSC里面已经提到过为什么传入了-1这个值，相当于是一个死循环（虽然不太严谨但是因为循环次数太多就当死循环了吧），不断执行指令，在这个loop里面会执行 $2^{32} - 1$ 次（已经非常接近无限次了），作为承上启下，这就不再多叙述了

在cpu_exec函数里面调用了execute函数，在这个函数里面才是真正开始进行指令相关的操作，在这个函数里面先是定义了一个Decode解码结构体s，然后在for循环里面 将s传入exec_once函数执行一次指令

在exec_once函数当中，对cpu的pc程序计数器进行操作修改（具体分为pc，static pc,dynamic pc这三种），并且再将s传入到isa_exec_once函数，在这个函数里面，先取指令ifetch函数然后传给s->isa.inst.val，在inst_fetch函数里面调用了vaddr虚拟内存读取函数，读取虚拟内存当中对应地址的指令

然后调用decode_exec函数，在这个函数里面进行解码与执行的操作（包括在这个里面更新pc值），在这个函数里面先进行模式匹配匹配对应的指令，比如说addi指令，通过观察反汇编出来的txt文件可以挑出其中一条

```
80000008: ffc10113 addi    sp,sp,-4 # 80009000 <_end> #addi指令
十六进制ffc10113 = 二进制111111111100 00010 000 00010 0010011
```

将80000008地址当中的内容ffc10113十六进制数转化为二进制数之后为111111111100 00010 000 00010 0010011，然后查看RISCV手册对相应的比特位进行匹配可以得出

```
INSPAT("?????? ???? ???? 000 ???? 00100 11", addi , I, {R(rd) = src1
+ imm});
```

说明这个是I型指令，然后调用decode_operand函数（在INSPAT当中的有一个宏命令INSPAT_MATCH中有这个函数），然后根据对应是I型指令去执行switch语句 case TYPE_I: src1R(); immI(); break; 然后又根据immI的宏命令#define immI() do { *imm = SEXT(BITS(i, 31, 20), 12); } while(0) //将立即数符号扩展到 32 位 注意SEXT表示的意思是取12位然后扩展到32位 去执行字抽取位扩展等等一系列我们在数字系统基础课里面学到的硬件知识，然后根据addi指令手册里面的内容去具体执行addi的内容（对于rd src1 src2寄存器、立即数等等进行操作）

然后根据nemu框架中把\$0寄存器置0，该函数返回0等操作（虽然不理解为什么要这么做但是这个是riscv32的规定吧估计）

所以其实感觉PA2的第一阶段主要内容还是偏向于硬件方面的实现，而且大部分内容都是在inst.c文件中去实现，而且已经有了一些指令的代码可以让我们去模仿实现，总体难度不是很大，但是自己也出了一些bug，比如immJ等宏命令的实现，自己也出过了bug。

编译与链接（static inline开头定义的inst_fetch()函数）

在C语言中，static 和 inline 是两个用于函数声明和定义的关键字，它们各自有不同的含义和用途。

1. static: 当用于函数定义时，static 关键字限制了函数的链接范围，使其只能在定义它的文件内被访问。这意味着在其他文件中无法通过外部链接访问这个函数。如果去掉 static，函数将成为外部链接，可以在其他文件中使用。
2. inline: inline 关键字是一个请求，它建议编译器在编译时将函数体直接插入到每个函数调用的地方，以减少函数调用的开销不会开辟新的栈帧直接在本栈帧执行代码，但是是否真正内联取决于编译器的优化决策，即使程序员声明要内联最后也是让编译器去决定的。通常，inline 函数不能有复杂的控制流（如循环或条件语句），并且不应该太大。

现在，我们分析去掉 static、inline 或两者的情况：

1. 去掉 static：如果去掉 static，函数 inst_fetch() 将不再是文件内链接，而是外部链接。这意味着它可以被其他文件中的代码调用。如果其他文件中的代码依赖于这个函数，那么去掉 static 不会导致编译错误。但是，如果这个函数只在定义它的文件中使用，那么去掉 static 可能不会产生任何影响，除非有其他文件错误地尝试链接这个函数。经过自己的测试的时候发现确实在编译的过程中是不会报错的。
2. 去掉 inline：如果你去掉 inline，函数 inst_fetch() 将不再是内联函数。这可能会影响到编译器的优化决策，但通常不会导致编译错误，顶多是代码的效率降低了一些，除非编译器依赖于 inline 来实现某些特定的优化。在某些情况下，如果函数体太大或者包含复杂的控制流，编译器可能不会将其内联，即使函数被声明为 inline。
3. 去掉两者：如果同时去掉 static 和 inline，函数 inst_fetch() 将成为一个普通的外部链接函数。这可能会导致编译错误，如果其他文件尝试调用这个函数，但实际上并没有在任何地方定义它。比如说我们在 am-kernels/tests/klib-tests 当中去测试 klib 的代码，编译起来会报错，具体报错信息如下：

```
/usr/bin/ld: /home/edzee3000/ICS2024/ICS2024PA/nemu/build/obj-riscv32-nemu-
interpreter/src/engine/interpreter/hostcall.o:
in function `inst_fetch':
hostcall.c:(.text+0x0):
multiple definition of `inst_fetch';
/home/edzee3000/ICS2024/ICS2024PA/nemu/build/obj-riscv32-nemu-
interpreter/src/isa/riscv32/inst.o:inst.c:(.text+0x1080):
first defined here
collect2: error: ld returned 1 exit status
```

发现主要编译错误是在 inst_fetch 中有多重定义

当函数定义在头文件中时，如果函数 inst_fetch 的定义被放在了一个头文件中，并且这个头文件被多个源文件包含（在nemu框架下面我也找了好久但是也没找到除了inst.c之外的.c文件include进了ifetch.h头文件，所以感觉挺奇怪的），那么每个包含这个头文件的源文件都会得到一个该函数的定义，导致链接时出现多重定义错误。因此如果 inst_fetch 函数只在某个特定的源文件中使用，应该使用 static 关键字来限制其链接范围，使其不会在其他文件中可见。

编译与链接 (dummy)

1. volatile static int dummy

重新编译后的NEMU会含有37个dummy变量的实体，至于我是如何得到这个结果的，其实是通过STFW之后才知道还有nm这个工具来列出目标文件或可执行文件中的符号。例如，`nm your_program` 会列出所有的符号

及其地址，可以搜索 dummy 来找到所有相关的符号，这将给出一个粗略的实体数量估计。但注意，如果 dummy 变量在某些文件中没有被使用，编译器可能会优化掉它，从而不会出现在 nm 的输出中。（因此我用 nm 工具找 am-kernels/tests/am-tests/build/amtest-riscv32-nemu.bin 可执行文件的 dummy 数量就没有找到，会不会就是因为这个原因）

于是我又重新再nemu当中进行make run重新运行，然后使用nm工具对nemu/build/riscv32-nemu-interpreter进行查找，最终发现会有37个dummy变量

```
000000000000fe20 b dummy
00000000000027a80 b dummy
000000000000e554 b dummy
000000000000d058 b dummy
000000000000d0c0 b dummy
000000000000d0c4 b dummy
000000000000d0d0 b dummy
000000000000d0e0 b dummy
000000000000e520 b dummy
000000000000e548 b dummy
000000000000e54c b dummy
000000000000e550 b dummy
000000000000e558 b dummy
000000000000e55c b dummy
000000000000e568 b dummy
000000000000e56c b dummy
000000000000e570 b dummy
000000000000e574 b dummy
000000000000e578 b dummy
000000000000e57c b dummy
000000000000e7a0 b dummy
000000000000e7b8 b dummy
000000000000e9e0 b dummy
:
```

虽然在less当中使用/dummy命令已经可以很方便帮我查找dummy出现在哪里了，但是统计其出现的次数就不是非常友好了.....因此我想到另一个方式就是使用grep命令以及wc命令去统计dummy单词出现的准确次数，运行结果如下，发现一共有37个dummy的实体变量

```
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA/nemu
$ nm build/riscv32-nemu-interpreter | grep -wo dummy | wc -l
37
```

并且自己分析一下也可以知道，在 C 语言中，关键字 volatile 表示一个变量可能会在程序的控制之外发生变化，例如由硬件或其他低级别的程序改变。关键字 static 表示变量的生命周期仅限于定义它的文件内部，并且其生存期贯穿整个程序运行期。

当在头文件 common.h 中声明一个 volatile static int dummy; 变量时，由于 static 关键字的作用，这个变量的实体只会在包含（include）这个 common.h 头文件的每个源文件中分别创建一次。这意味着，如果 common.h 被多个源文件包含，那么 dummy 变量将会在每个源文件中都有一个独立实体，每个实体都具有 volatile 和 static 的属性。因此上述出现了37个dummy的实体变量，也就意味着在nemu这个项目框架下面一共有37个文件include进了common.h这个头文件

2. 在nemu/include/debug.h中添加一行volatile static int dummy

保存更改之后重新编译make run重复上述过程，最后发现和第1小问当中的dummy实体数量竟然是一样的，仍然是37个dummy实体数量

按理来说，如果在common.h与debug.h当中都有volatile static int dummy的话，既include进common.h又debug.h的话，那么在每一个对应的文件下面都会有一个dummy实体，而且分别创建出来的dummy会产生冲突

但是观察可以发现，这两个头文件是相互包含的，我在这里猜测一下可能是因为循环依赖以及条件编译指令解决循环依赖的问题之后使得最终这2个头文件里面事实上只有一条volatile static int dummy;定义命令。这样在任何文件下面无论include哪一个文件，事实上2个头文件都包含进来了，都可以实现创建一个且仅有一个dummy实体。

```

> config
  < common.h M
    34  #if CONFIG_MBASE + CONFIG_MSIZE > 0x1000000000
    35  #define PMEM64 1
    36  #endif
    37
    38  typedef MUXDEF(CONFIG_ISA64, uint64_t, uint32_t)
    39  typedef MUXDEF(CONFIG_ISA64, int64_t, int32_t)
    40  #define FMT_WORD MUXDEF(CONFIG_ISA64, "0x%016"
    41
    42  typedef word_t vaddr_t;
    43  typedef MUXDEF(PMEM64, uint64_t, uint32_t) pad
    44  #define FMT_PADDR MUXDEF(PMEM64, "0x%016" PRIx
    45  typedef uint16_t ioaddr_t;
    46
    47  #include <debug.h>
    48
    49
    50 // volatile static int dummy;
    51
    52
    53 #endif
    54

```

```

> config
  < common.h M
    12  *
    13  * See the Mulan PSL v2 for more details.
    14  ****
    15
    16  #ifndef __DEBUG_H__
    17  #define __DEBUG_H__
    18
    19  #include <common.h>
    20  #include <stdio.h>
    21  #include <utils.h>
    22
    23  #define Log(format, ...) \
    24  | Log(ANSI_FMT("[%s:%d %s] " format, ANSI
    25  | | _FILE_, _LINE_, _func_, ##_
    26
    27  #define Assert(cond, format, ...) \
    28  | do { \
    29  | | if (!(cond)) { \
    30  | | | MUXDEF(CONFIG_TARGET_AM, printf(ANSI
    31  | | | (fflush(stdout), fprintf(stderr, ANSI
    32  | | | TENDDEF(CONFIG_TARGET_AM, extern FILE

```

如果两个头文件 common.h 和 debug.h 相互包含，即 common.h 包含 debug.h，同时 debug.h 也包含 debug.h，这会造成循环依赖。在 C 语言中，这种情况通常需要小心处理，以避免编译错误。

对于 volatile static int dummy; 的定义，如果两个头文件相互包含并且都定义了 dummy，那么无论是否使用 static 关键字，都会导致重复定义的问题。这是因为每个包含这些头文件的源文件都会尝试定义两次 dummy。

解决循环依赖和重复定义的一种方法是使用条件编译指令 #ifndef #define 和 #endif 来确保每个头文件的内容只被包含一次。最终导致这两个头文件事实上都只定义了一次dummy，所以通过nm工具查看dummy实体个数最终结果是一样的

3. 为两处dummy变量进行初始化

在这两处如果对dummy进行初始化的话，然后重新make run编译运行，最终发现编译错误，并且报了一个redefinition重复定义的错误

```
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA/nemu
$ make run
+ CC src/device/device.c
In file included from src/device/device.c:16:
/home/edzee3000/ICS2024/ICS2024PA/nemu/include/common.h:50:21: error: redefinition of 'dummy'
  50 | volatile static int dummy=0;
      | ^~~~~~
In file included from /home/edzee3000/ICS2024/ICS2024PA/nemu/include/common.h:47
:
/home/edzee3000/ICS2024/ICS2024PA/nemu/include/debug.h:43:21: note: previous definition of 'dummy' with type 'int'
  43 | volatile static int dummy=0;
      | ^~~~~~
make: *** [/home/edzee3000/ICS2024/ICS2024PA/nemu/scripts/build.mk:37: /home/edzee3000/ICS2024/ICS2024PA/nemu/build/obj-riscv32-nemu-interpreter/src/device/device.o] Error 1
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA/nemu
$
```

我个人猜测啊，之前没有报错是因为没有初始化，如果 dummy 变量没有在头文件中初始化，那么编译器会为每个包含头文件的源文件创建一个独立的 static 变量实体，但不会报错，因为每个实体都是独立的，每个编译单元都有自己的符号表也是独立的，而这个变量实体是放在.bss区的（未初始化的static全局变量被放在BSS段），这样的话这些不同的变量实体就像第2小问中两个头文件只定义了一次dummy符号

但是在第3小问中，如果对dummy变量进行初始化的话，性质就不一样了。

在C语言中，全局变量的存储位置和初始化状态有关。全局变量可以被分为以下几种：

(1) 初始化的全局变量：

这些变量在定义时就被赋予了初始值。它们通常被放在数据段 (data segment) 的初始化数据区域 (initialized data section) 。例如：

```
int global_var = 10; // 初始化的全局变量
```

(2) 未初始化的全局变量：

这些变量在定义时没有被赋予初始值。它们通常被放在数据段的未初始化数据区域 (uninitialized data section) ，也称为BSS段 (Block Started by Symbol) 。例如：

```
int global_var; // 未初始化的全局变量
```

(3) 静态全局变量：

这些变量被声明为static，意味着它们的生命周期仅限于定义它们的文件内部。初始化的static全局变量也被放在初始化数据区域。未初始化的static全局变量同样被放在BSS段。例如：

```
static int static_global_var = 20; // 初始化的静态全局变量
static int static_uninit_global_var; // 未初始化的静态全局变量
```

(4) 常量全局变量：

如果全局变量被声明为const或volatile，它们也会被放在只读数据段（read-only data segment）。例如：

```
const int const_global_var = 30; // 常量全局变量
```

在链接脚本或特定的编译器设置中，这些段可能会有不同的名称，但概念是相同的。初始化的全局变量需要在程序启动时就被赋予一个确定的值，而未初始化的全局变量则需要在程序运行时被初始化，它们在BSS段中默认初始化为零。

在第3小问中如果对两处dummy都初始化的话性质就变了，先以一个初始化全局变量为例：

```
+ CC src/isa/riscv32/system/intr.c
+ CC src/engine/interpreter/init.c
+ CC src/engine/interpreter/hostcall.c
+ CC src/device/io/port-io.c
+ CC src/device/io/map.c
+ CC src/device/io/mmio.c
+ CC src/cpu/difftest/dut.c
+ CC src/cpu/difftest/ref.c
+ CC src/cpu/cpu-exec.c
src/cpu/cpu-exec.c:37:11: error: redefinition of 'cpu'
  37 | CPU_state cpu = {};
      | ^~~
src/cpu/cpu-exec.c:36:11: note: previous definition of 'cpu' with type 'CPU_stat
e' {aka 'riscv32_CPU_state'}
  36 | CPU_state cpu = {};
      | ^~~
make: *** [/home/edzee3000/ICS2024/ICS2024PA/nemu/scripts/build.mk:37: /home/edz
ee3000/ICS2024/ICS2024PA/nemu/build/obj-riscv32-nemu-interpreter/src/cpu/cpu-ex
c.o] Error 1
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/ICS2024/ICS2024PA/nemu
```

CPU_state cpu = {};

CPU_state cpu = {};

uint64_t g_max_inst = 0;

static uint64_t g_timer = 0; // unit: us

static bool g_print_step = false;

void device_update();

IFDEF(CONFIG_WATCHPOINT, void wp_diff_test());

void print_trace_memory();

static void trace_and_difftest(Decode *_this, vaddr_t dnpcc) {
 #ifdef ITRACE_COND
 | if (ITRACE_COND) { log_write("%s\n", _this->logbuf); }
 #endif
}

在cpu-exec.c文件中假设对两个cpu重定义，就会发生冲突，原因是初始化的全局变量是放在读写数据段（data segment）的初始化数据区，而且由于是全局变量会有对应的符号，于是就发生了命名的冲突。

那好现在在头文件中也一样，更何况这两个头文件还是相互包含的，那么假设在debug.h当中include进common.h头文件，在common.h当中定义了初始化的全局变量，现在在debug.h当中又定义了初始化全局变量，这两个全局变量都有名字并且在debug.h当中都有独立的static实体与对应的值，这样就发生了冲突。

了解Makefile

通过在am-kernels/tests/am-tests目录下面运行下面的命令并且将其传入到一个txt文件中查看

```
make -n ARCH=riscv32-nemu run mainargs=t > build/MakeProcess/rtc-amtest-riscv32-nemu-make-process.txt
```

然后再vim中使用下面的命令筛选掉一些无关紧要的命令，仅保留为了理解makefile过程的一些过程这样再去分析

```
:g/fixdep/c
:g/mv/c
:g/echo/c
:g/mkdir/c
:%s#字符串
```

```
am-kernels > tests > am-tests > build > MakeProcess > E tc-amtest-riscv32-nemu-make-process.txt
1 # Building amtest-run [riscv32-nemu]
2 make -C /ICS2024PA/am-kernels/nemu -march=nemu archive
3 # Building klib archive [riscv32-nemu]
4 make -s C /ICS2024PA/abstract-machine/klib archive
5 # Building klib archive [riscv32-nemu]
6 riscv64-linux-gnu-objdump -d /ICS2024PA/am-kernels/tests/am-tests/build/amtest-riscv32-nemu.elf > /ICS2024PA/am-kernels/tests/am-tests/build/amtest-riscv32-nemu.txt
7 riscv64-linux-gnu-objcopy --sectionFlags .bs=alloc_contents -O binary /ICS2024PA/am-kernels/tests/am-tests/build/amtest-riscv32-nemu.elf /ICS2024PA/am-kernels/tests/am-tests/build/amtest-riscv32-nemu.bin
8 python3 /ICS2024PA/am-kernels/tests/am-tests/build/amtest-riscv32-nemu/runArg.py /ICS2024PA/am-kernels/tests/am-tests/build/amtest-riscv32-nemu.bin 64 "The Insert-arg role in Makefile will insert mainargs here."
9 make -C /ICS2024PA/nemu lseriscv32 run ARGS=-l /ICS2024PA/am-kernels/tests/am-tests/build/nemu.log.txt -b -e /ICS2024PA/am-kernels/tests/am-tests/build/amtest-riscv32-nemu.elf
10 make[1]: Leaving directory '/ICS2024PA/nemu'
```

11 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/device.o
12 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/alarm.o
13 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/intr.o
14 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/interrupt.o
15 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/timer.o
16 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/keyboard.o
17 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/audio.o
18 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/memory_main.o
19 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/isa/riscv32/reg.o
20 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/isa/riscv32/init.o
21 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/isa/riscv32/int.o
22 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/isa/riscv32/intf.o
23 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/isa/riscv32/intf_syst.o
24 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/isa/riscv32/intf_syst_syst.o
25 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/isa/riscv32/intf_syst_syst_syst.o
26 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/isa/riscv32/intf_syst_syst_syst_syst.o
27 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/engine/interpret.o
28 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/engine/interpret_syst.o
29 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/DeviceIO/pio.o
30 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/DeviceIO/pio_syst.o
31 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/device/IO/mmio.o
32 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/cpu/difftest/dut.o
33 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/cpu/difftest/ref.o
34 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/cpu/cpu-exec.o
35 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/cpu/cpu-exec_syst.o
36 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/monitor/rdmptc.o
37 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/monitor/sdb/expr.o
38 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/monitor/sdb/sdb.o
39 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/trace/trace.o
40 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/trace/trace_syst.o
41 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/utils/utrace.o
42 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/utils/utrace_syst.o
43 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/utils/dtrace.o
44 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/utils/log.o
45 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/utils/irngbuf.o
46 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/memory/vaddr.o
47 gcc -O2 -MM -Wall -Werror /ICS2024PA/nemu/include /ICS2024PA/nemu/src/isa/riscv32/include /ICS2024PA/nemu/src/engine/interpreter -O2 -D GUEST_ISA=riscv32 -c -o /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter/src/memory/gaddr.o
48 g++ -O /ICS2024PA/nemu/build/riscv32-nemu-interpreter /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter /ICS2024PA/nemu/build/obj/riscv32-nemu-interpreter
49 git add /ICS2024PA/nemu.. -A -ignore-errors
50 while (test -e .git/index.lock); do sleep 0.1; done
51 sync
52 git add /ICS2024PA/nemu.. -A -ignore-errors
53 while (test -e .git/index.lock); do sleep 0.1; done
54 sync
55 /ICS2024PA/nemu/build/riscv32-nemu-interpreter -l /ICS2024PA/nemu/build/nemu.log.txt -b -e /ICS2024PA/nemu/build/nemu.log.txt -b -e /ICS2024PA/nemu/build/nemu.log.txt
56 make[1]: Leaving directory '/ICS2024PA/nemu'

同样对于am-kernels/kernels/hello/目录下敲入make ARCH=\$ISA-nemu 后，也做上述同样的操作，最后可以得到下面的make运行结果，之后再对make程序的组织进行分析

make程序组织.c和.h文件,生成可执行文件elf的过程:

1. 解析命令行参数 make 首先解析命令行参数。ARCH=riscv32-nemu 指定了 ARCH 变量的值，指定了环境变量为 riscv32-nemu，表示目标架构
 2. 读取 Makefile make 会在当前目录 (am-kernels/kernels/hello/) 寻找 Makefile 或 makefile 并读取它。make 会根据 Makefile 中定义的规则和变量来确定如何编译和链接程序。
 3. 处理变量和包含文件 Makefile 中可能定义了多个变量，如 CC (编译器)，CFLAGS (编译器标志)，LDFLAGS (链接器标志) 等。它还可能包含其他 Makefile 文件，使用 include 指令，例如在 abstract-machine/Makefile 当中：

```
-include $(AM_HOME)/scripts/$(ARCH).mk
```

又比如在abstract-machine/scripts/riscv32-nemu.mk当中：

```
include $(AM_HOME)/scripts/isa/riscv.mk  
#注意这里直接把platform/nemu.mk的markdown文件包含进来    这样子运行riscv32的isa的  
nemu的时候不用重复写nemu的makefile了  
include $(AM_HOME)/scripts/platform/nemu.mk
```

这些包含文件可能定义了额外的变量和规则，或者指定了编译和链接过程中的一些特定行为。

4. 应用隐式规则 make 运用并重写了一些隐式规则来编译 .c 文件生成 .o 文件，以及链接 .o 文件生成最终的可执行文件。例如，一个隐式规则可能看起来像这样：

```
% .o: %.c
$(CC) $(CFLAGS) -c $< -o $@
```

这条规则告诉 make 如何从 .c 文件生成 .o 文件。< 表示依赖项 (.c文件) , @ 表示目标 (.o 文件)

如果将隐式规则进行展开的话，可能就长这样（以hello-riscv32-nemu-make-process.txt为例）：

```
gcc -O2 -MMD -Wall -Werror -I/nemu/include -I/nemu/src/isa/riscv32/include
-I/nemu/src/engine/interpreter -O2 -DTRACE_COND=true -
D_GUEST_ISA=riscv32 -c -o /nemu/build/obj-riscv32-nemu-
interpreter/src/cpu/cpu-exec.o src/cpu/cpu-exec.c
gcc -O2 -MMD -Wall -Werror -I/nemu/include -I/nemu/src/isa/riscv32/include
-I/nemu/src/engine/interpreter -O2 -DTRACE_COND=true -
D_GUEST_ISA=riscv32 -c -o /nemu/build/obj-riscv32-nemu-
interpreter/src/monitor/monitor.o src/monitor/monitor.c
gcc -O2 -MMD -Wall -Werror -I/nemu/include -I/nemu/src/isa/riscv32/include
-I/nemu/src/engine/interpreter -O2 -DTRACE_COND=true -
D_GUEST_ISA=riscv32 -c -o /nemu/build/obj-riscv32-nemu-
interpreter/src/monitor/sdb/watchpoint.o src/monitor/sdb/watchpoint.c
gcc -O2 -MMD -Wall -Werror -I/nemu/include -I/nemu/src/isa/riscv32/include
-I/nemu/src/engine/interpreter -O2 -DTRACE_COND=true -
D_GUEST_ISA=riscv32 -c -o /nemu/build/obj-riscv32-nemu-
interpreter/src/monitor/sdb/expr.o src/monitor/sdb/expr.c
gcc -O2 -MMD -Wall -Werror -I/nemu/include -I/nemu/src/isa/riscv32/include
-I/nemu/src/engine/interpreter -O2 -DTRACE_COND=true -
D_GUEST_ISA=riscv32 -c -o /nemu/build/obj-riscv32-nemu-
interpreter/src/monitor/sdb/sdb.o src/monitor/sdb/sdb.c
```

5. 编译过程 make 根据 Makefile 中定义的模式规则和变量，编译 .c 文件生成 .o 文件。例如，如果 SRC 变量包含所有源文件的列表，make 可能会遍历这个列表，并对每个 .c 文件应用编译规则。

6. 链接过程一旦所有的 .o 文件都被生成，make 将使用链接规则来生成最终的可执行文件。链接规则可能类似于：

```
hello-$(ARCH)-nemu.elf: $(OBJS)
$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)
```

这里，(OBJS)是所有. o文件的列表，(LIBS) 是链接过程中需要的库文件列表。\$@ 表示最终的目标文件名。

对于在这里面的hello为例的话，链接过程就应该是如下link过程：

```
riscv64-linux-gnu-ld -z noexecstack -T/abstract-machine/scripts/linker.ld
-melf64lriscv
--defsym=_pmem_start=0x80000000
--defsym=_entry_offset=0x0
--gc-sections
```

```

-e _start
-melf32lriscv
-o /am-kernels/kernels/hello/build/hello-riscv32-
nemu.elf
--start-group /am-kernels/kernels/hello/build/riscv32-
nemu/hello.o
/abstract-machine/am/build/am-riscv32-nemu.a
/abstract-machine/klib/build/klib-riscv32-nemu.a
--end-group

```

7. 输出文件 最终 make 将生成的可执行文件放置在 build 目录下，文件名为 hello-riscv32-nemu.elf。并将反汇编结果输入给txt文件以及汇编结果给.bin文件

```

riscv64-linux-gnu-objdump -d /am-kernels/kernels/hello/build/hello-riscv32-
nemu.elf > /am-kernels/kernels/hello/build/hello-riscv32-nemu.txt
riscv64-linux-gnu-objcopy -S --set-section-flags .bss=alloc,contents -O
binary /am-kernels/kernels/hello/build/hello-riscv32-nemu.elf /am-
kernels/kernels/hello/build/hello-riscv32-nemu.bin

```

8. 运行可执行文件 最后运行/nemu/build/riscv32-nemu-interpreter，将其装载进内存里面执行nemu的解释器，之后就是我们最熟悉的取指、译码、执行、更新PC等等一系列周而复始的操作

```

/nemu/build/riscv32-nemu-interpreter -l /am-
kernels/kernels/hello/build/nemu-log.txt -b -e /am-
kernels/kernels/hello/build/hello-riscv32-nemu.elf /am-
kernels/kernels/hello/build/hello-riscv32-nemu.bin

```

实验日志

inst.c中匹配指令的内容

一开始不是很熟悉这些指令，riscv手册英文版我也看不懂，一条指令一条指令去寻找，甚至有些是经过优化给予了特殊含义，而我还不知道（比如ret是可以被等价转化为jal有关的指令格式），我甚至都不太记得那6种基本指令格式。因此也踩过挺多坑的。下面的内容其实是我一边写inst.c一边做的记录，就当做实验日志吧。

1. 对于li指令：在PA2的dummy实现当中还没有实现0x00000413的指令，查看反汇编代码可以发现其助记符为li，表示将0加载到s0当中去

```

80000000 <_start>:
80000000: 00000413 li s0, 0

```

将0x00000413翻译成二进制语言就是0000 0000 0000 0000 0000 0100 0001 0011 为了更加符合指令格式的形式，改写一下则为

```
000000000000 00000 000 01000 0010011
```

观察比较另外一条指令可以发现

```
80000010 <main>:  
80000010: 00000513      li    a0, 0  
# 00000513即为 000000000000 00000 000 01010 0010011
```

两条指令的差别其实就在于寄存器名字s0和a0以及其对应位置01000与01010，可以明白0010011表示的就是li指令

2. 对于addi指令：

```
80000008: ffc10113          addi    sp, sp, -4 # 80009000 <_end> #addi指  
令 111111111100 00010 000 00010 0010011  
8000005c: 00000493          li     s1, 0          #addi指令  
000000000000 00000 000 01001 0010011
```

ffc是imm立即数-4，00010是rs1，00010是rd

3. 对于jal指令：

```
8000000c: 00c000ef          jal    80000018 <_trm_init>      #0000 0000  
1100 0000 0000 00001 1101111  
80000028: fe9ff0ef          jal    80000010 <main>           #1111 1110  
1001 1111 1111 00001 1101111  
0x8000000c: 680ec0ef          jal    0xec680                 #0  
1101000000 0 11101100 00001 1101111  这里的imm为011101100011010000000
```

JAL:PC+4 的结果送 rd 但不送入PC，然后计算下条指令地址。转移地址采用相对寻址，基准地址为当前指令地址(即 PC)，偏移量为立即数 imm20 经符扩展后的值的 2 倍。

4. 对于jalr指令：

```
800001ec: 000780e7          jalr   a5                  #00000000  
00000 01111 000 00001 1100111 我靠为啥指令集手册里面写错了？？？funct3明明是  
000为啥指令集手册里面是010  
80000050: 00078067          jr    a5                  #00000000 00000 01111 000  
00000 1100111
```

5. 对于mv指令：

8000002c:	00050513	mv a0, a0
-----------	----------	-----------

0x00050513即为000000000000 01010 000 01010 0010011 本质上用的还是addi

6. 对于ret指令：

80000014:	00008067	ret #ret指令，会扩展为jalr x0, 0(x1)
000000000000	00001 000 00000 1100111	

执行ret函数后，会把PC设置成ra寄存器中保存的值，继续执行函数调用前的指令

7. 对于B型指令：

80000010:	00050463	beqz a0, 80000018 <check+0x8>	#beqz指令
0 000000 00000 01010 000 0100 0 1100011			
800000b4:	fd3410e3	bne s0, s3, 80000074 <main+0x4c>	#bne指令
1 111110 10011 01000 001 0000 1 1100011			
8000007c:	02e94063	blt s2, a4, 8000009c <main+0x74>	#blt指令
0 000001 01110 10010 100 0000 0 1100011			
80000094:	00eb5463	bge s6, a4, 8000009c <main+0x74>	#bge指令
0 000000 01110 10110 101 0100 0 1100011			
800000c8:	fc8912e3	bne s2, s0, 8000008c <main+0x64>	#bne指令
1 111110 01000 10010 001 0010 1 1100011			

8. 对于Loads类型指令：

80000074:	00042703	lw a4, 0(s0) #lw指令	000000000000 01000
010 01110 0000011			
80000190:	fef71fa3	sh a5, -1(a4) #sh指令	111111101111 01110
001 11111 0100011			

load 和 store 指令在 registers 和 memory 之间传输一个值。加载以 I 类型格式编码，存储为 S 类型。通过将寄存器 rs1 添加到符号扩展的 12 位偏移量来获得有效地址。加载将值从内存复制到寄存器 rd。存储将寄存器 rs2 中的值复制到内存。LW 指令从内存中加载一个 32 位值到 rd. LH 从内存中加载一个 16 位值，然后符号扩展到 32 位，然后存储在 rd 中。LHU 从内存中加载一个 16 位值，但随后零扩展到 32 位，然后存储在 rd 中。SW、SH 和 SB 指令将寄存器 rs2 的低位的 32 位、16 位和 8 位值存储到存储器中。

9. 对于移位操作：

8000005c:	01f55513	srli a0, a0, 0x1f #srli指令逻辑右移
000000 011111 01010 101 01010 00100 11		
800002a4:	0ff5f593	zext.b a1, a1 #zext指令
01011 111 01011 0010011 0扩展指令（注意可以用andi指令去实现）		000011 111111
80000188:	00141793	slli a5, s0, 0x1 #slli逻辑左移指令

```
0000000 000001 01000 001 01111 00100 11
800000d0: 01071713          slli    a4, a4, 0x10 #slli逻辑左移指令
0000000 010000 01110 001 01110 00100 11
```

10. 对于算术运算指令：

800000a8: 40f50533	sub a0, a0, a5	#sub指令	0100000 01111
01010 000 01010 0110011	R型指令		
80000098: 00fc87b3	add a5, s9, a5	#add指令	0000000 01111
11001 000 01111 0110011			
800002d4: 03176733	rem a4, a4, a7	#rem指令	0000001 10001
01110 110 01110 0110011	求余数指令		
800000b0: 02f795b3	mulh a1, a5, a5	#mulh指令	0000001 01111
01111 001 01011 0110011			

11. 对于非与或等位操作指令：

800000b4: 00f56533	or a0, a0, a5	#or指令	0000000 01111
01010 110 01010 0110011	R型指令		
800000ac: 00f54533	xor a0, a0, a5	#xor指令	0000000 01111
01010 100 01010 0110011	R型指令		

12. 一些“奇奇怪怪”的置位指令：

800000ac: 00153513	seqz a0, a0	#seqz指令	000000000001
01010 011 01010 0010011			
800000a0: 0197b6b3	sltu a3, a5, s9	#sltu指令	0000000 11001
01111 011 01101 0110011			

在nemu中进行批处理命令使得避免键入c再开始执行

首先RTFSC，在nemu/src/monitor/monitor.c中的parse_args解析函数函数中存在一个批处理模式的匹配，当参数为-b时会调用函数sdb_set_batch_mode打开批处理模式，而parse_args的参数又来自main函数，因此，只需要在对应的AM的Makefile中在运行nemu时传入一个-b的参数就可以打开批处理模式

在AM的Makefile代码框架中，第四步会选择对应的架构配置 这里选择nemu对应的架构mk文件
 $(AM_HOME)/script/riscv32 - nemu.mk$ 发现运行nemu的mk不在这里，再进入运行nemu时的mk，进入 $(A_HOME)/script/platform/nemu.mk$

6个小时成功debug一个小问题

那个lui指令 上面已经帮我处理好了imm左移12位的情况，在下面就不需要自己再重新左移12位了，就因为这个单步调试了6个小时，前途一片完犊子啊

针对mulh的命令

```

80000094: 02fc06b3           mul a3,s8,a5      #对s8*a5低32位给a3 s8
为aeb1c2aa a5为aeb1c2aa    a3为低32位db1a18e4
80000098: 004b8b93           addi   s7,s7,4
8000009c: 00840413           addi   s0,s0,8
800000a0: 02fc17b3           mulh   a5,s8,a5      #a5为高32位 但是我自己计
算出来a5竟然是0x7736200d  实际上应该是0x19d29ab9
800000a4: 00d54533           xor    a0,a0,a3
800000a8: 00f747b3           xor    a5,a4,a5
800000ac: 00f56533           or     a0,a0,a5
800000b0: 00153513           seqz   a0,a0

```

经过持之以恒的努力debug之后终于知道问题出在哪里了 int类型负数强制转换为long long int的时候竟然不会帮我自动进行符号扩展.....

ftrace当中有关riscv的函数调用与返回

注意对于ftrace中，确定某一次函数调用/返回时，执行指令时可以得到跳转的目标地址，遍历从ELF中得到的函数信息，目标地址在什么函数的范围内就是调用/返回了什么函数。

在riscv中，jal指令只会进行调用。但是jalr不是，根据情况可以是调用也可以是返回，所以我们需要判断jalr指令的参数，从而确定本次跳转究竟是什么类型。

根据手册中可以得知，riscv的通用寄存器中，x1 (ra) 寄存器比较特殊，通常用于保存返回地址。返回地址在调用与返回中都很重要，调用时需要保存返回地址，返回时需要读取返回地址，那么判断条件基本明确。

jalr类型条件 调用 (Call) rd == 1 返回 (Ret) rd == 0 && rs1 == 1

在Differential Testing中使用spike模拟器遇到的问题

在/home/edzee3000/ICS2024/ICS2024PA/nemu/tools/spike-diff/repo/riscv/log_file.h当中有一个对类模板的改进（自定义删除器）使得编译的过程中没有警告warning

```

10 // 自定义删除器
11 struct file_deleter {
12     void operator()(FILE* file) {
13         if (file) {
14             fclose(file);
15         }
16     }
17 };
18 // Header-only class wrapping a log file. When constructed with an
19 // actual path, it opens the named file for writing. When constructed
20 // with the null path, it wraps stderr.
21 class log_file_t
22 {
23 public:
24     log_file_t(const char *path)
25         // : wrapped_file(nullptr, &fclose)
26         : wrapped_file(nullptr, file_deleter())
27     {
28         if (!path)
29             return;
30
31         wrapped_file.reset(fopen(path, "w"));
32         if (!wrapped_file) {
33             std::ostringstream oss;
34             oss << "Failed to open log file at `" << path << "' : "
35             << strerror(errno);
36             throw std::runtime_error(oss.str());
37         }
38     }
39
40     FILE *get() { return wrapped_file ? wrapped_file.get() : stderr; }
41
42 private:
43     // std::unique_ptr<FILE, decltype(&fclose)> wrapped_file;
44     std::unique_ptr<FILE, file_deleter> wrapped_file;
45 }

```

理解volatile关键字

从 diff -u 命令的输出中，我们可以看到两个反汇编代码文件之间的差异。以下是一些关键点，解释了为什么这些差异出现，以及它们可能意味着什么：

1. 代码块的移除：

在包含 volatile 的版本中，有一些额外的代码块，例如：

```
assembly
lea    0x2fc6(%rip),%rdx          # 4018 <_end>
nopw   0x0(%rax,%rax,1)
movzbl (%rdx),%eax
cmp    $0xff,%al
jne    1058 <main+0x18>
```

这些指令在不包含 volatile 的版本中被移除。这表明编译器优化了这部分代码，因为它认为 volatile 变量的值不会改变。

2. 循环的简化：

在包含 volatile 的版本中，有一个循环，它检查 *p 是否等于 0xff。这个循环在不包含 volatile 的版本中被简化为一个无限循环：

```
assembly
jmp    114b <fun+0xb>
```

这表明编译器在优化时认为循环条件永远不会为真，因此将其替换为一个无限循环。指令的重排：

在包含 volatile 的版本中，有一些指令被重排或优化掉了。例如，设置 *p 值的指令在两个版本中的位置不同，或者在不包含 volatile 的版本中被完全移除。额外的函数：

在包含 volatile 的版本中，有一些额外的函数调用，如 __do_global_dtors_aux 和 frame_dummy。这些在不包含 volatile 的版本中可能被优化掉了。初始化和退出代码：

在包含 volatile 的版本中，_fini 函数包含了一些额外的栈操作指令，这些在不包含 volatile 的版本中被简化或移除。这些差异表明，当编译器看到 volatile 关键字时，它会假设变量的值可能会在程序的控制之外改变，因此不会对涉及该变量的代码进行优化。而当 volatile 关键字被移除时，编译器对代码进行了更多的优化，包括移除不必要的指令和循环。

在实际的硬件编程中，volatile 关键字用于确保对硬件寄存器的访问不会被优化掉，因为硬件寄存器的值可能会在任何时候由外部硬件改变。如果去掉 volatile，程序可能会不正确地运行，因为它可能不会按预期那样与硬件设备交互。

去掉 volatile 可能会带来什么问题：

如果 p 指向的地址映射到一个设备寄存器，那么每次读取或写入这个地址时，都需要确保从硬件设备中获取最新的值或将值写入硬件设备。如果没有 volatile，编译器可能会优化掉一些它认为不必要的读取或写入操作，这可能会导致程序无法正确地与硬件设备交互。例如，如果硬件设备在某个时刻将寄存器的值设置为 0xff，但由于编译器的优化，程序可能永远不会看到这一改变，因为它可能使用了过时的寄存器值。

立即数扩展写错导致的jal到了小于0x80000000的部分

在jal的immJ()部分对于imm立即数扩展部分，其实只需要对最高位进行扩展就可以了，结果我对所有的4个部分全都进行符号位扩展之后再按位或运算，这就导致了最终pc的值可能就指向了小于0x80000000的部分（而nemu在虚拟内存中的分布是在0x80000000~0x87fffff之间的）

更离谱的是在写错的情况下cpu-tests里面所有的测试用例都是可以通过的，这个还是在之后运行am-tests的时候回来补的坑，泪目了……

观看完王慧妍老师的PA习题课之后的一些笔记

```
make -nB | grep -ve '^(\#|echo\|mkdir\)' > ./Experiment_Log/nemu_make
make -nB \
| grep -ve '^(\#\|echo\|mkdir\|make\)' \
| sed "s#${AM_HOME}#${AM_HOME#g}" \
| sed "s#${PWD}#.#g" \
| vim -

:g/fixdep/c
:g/mv/c
:g/echo/c
:%s#字符串
```

通过观察最终-nB的结果可以发现有一个-I选项后面直接接目录的名字，在这里-I将这个路径加到找系统的标准库的那个路径

可以在nemu目录下面执行下面的语句

```
make -j4 gdb
```

从而在在gdb里面运行nemu

```
echo -e "\033[43;34m I-LOVE-NANJING \033[0m"
```

可以使用`ANSI_FMT`打印我们想要的对应样式的字符串

Kconfig是一套配置描述语言 更适合维护有依赖关系的宏 显式描述依赖关系 配置选项具备层次关系 根据配置编译出宏定义代码 `nemu/.config`

menuconfig就是为了方便开发者选择配置而实现的一种可视化工具

可以优化的一些地方

重构mtrace以及iringbuf的代码，尽量不要放在cpu-exec.c里面，一个模块专为做某一个模块的事情

为什么错误码是1呢？你知道make程序是如何得到这个错误码的吗？

错误码 1 在 make 程序中通常表示命令执行失败。在 make 过程中，每个命令（recipe）的执行结果都会返回一个退出状态（exit status），也称为退出码（exit code）。在 Unix 和类 Unix 系统中，0 通常表示成功，非零值表示错误或异常。

当 make 执行的某个命令返回非零退出状态时，make 会停止执行并返回该错误码。例如，如果在链接阶段，链接器（如 ld）找不到某个符号的定义，它会返回一个错误，并带有一个错误码，通常是 1。make 捕获到这个错误码后，会停止构建过程，并显示 `make: *** [target] Error 1` 的消息。

为什么错误码是 1，这主要是因为许多程序在发生错误时都返回 1 作为其退出状态。这不是 make 特有的，而是遵循了 Unix 和类 Unix 系统中的惯例。不同的错误可能会有不同的退出码，但 1 是最常用的，因为它简单且易于记忆。

在搜索结果中，有多个来源提到了这一点，例如在 GNU make 的官方文档中就提到了这一点。此外，其他来源也提到了类似的情况，如链接错误、缺少依赖库、编译错误等，这些都可能导致 make 返回错误码 1。

要解决这个问题，通常需要检查 make 执行的命令，查看具体的错误信息，并根据错误信息进行调试。可能的解决方案包括确保所有依赖库都已正确安装和配置、修正源代码中的错误、调整编译选项等。在某些情况下，使用 make 的调试选项（如 `--debug` 或 `-d`）可以帮助诊断问题。

为什么定义宏 `NATIVE_USE_KLIB` 之后就可以把 native 上的这些库函数链接到 klib？这具体是如何发生的？尝试根据你在课堂上学习的链接相关的知识解释这一现象。

在 C 和 C++ 编程中，链接是将编译后的对象文件（.o 文件）或库文件（.a 或 .so 文件）组合成可执行文件的过程。这个过程涉及到解析每个函数和变量的引用，并将它们映射到它们的定义上。当程序被编译时，它会包含对库函数的调用，这些调用在编译时还不是具体的地址，而是符号引用。

宏 `NATIVE_USE_KLIB` 的定义改变了链接过程的行为，使得编译器和链接器将特定的库函数调用重定向到 klib 提供的实现上。这个过程通常涉及以下几个步骤：

编译时的宏定义：当编译器遇到 `#define NATIVE_USE_KLIB` 时，它会将代码中的条件编译部分展开，使得 `NATIVE_USE_KLIB` 宏下的条件编译代码被包含在编译结果中。

函数声明：在 klib.h 中，所有的库函数（如 `memset`、`printf` 等）都有对应的声明。这些声明告诉编译器这些函数的存在和它们的接口（参数类型和返回类型）。

函数定义：在 klib 的实现文件中（通常是 .c 或 .cpp 文件），这些库函数被具体定义。这些定义提供了函数的具体实现，以及它们在内存中的地址。

链接时的重定向：在链接阶段，链接器（如 ld）会解析程序中的所有符号引用。如果没有定义 `NATIVE_USE_KLIB`，链接器会默认寻找标准的库实现（如 glibc）。但是，定义了 `NATIVE_USE_KLIB` 后，链接器会被告知去链接 klib 提供的实现。

静态库或动态库：klib 可以被编译成静态库（.a 文件）或动态库（.so 文件）。如果是静态库，链接器会从库中提取需要的对象文件并将其包含在最终的可执行文件中。如果是动态库，链接器会记录运行时需要加载的库的信息，然后在程序运行时动态加载这些库。

运行时解析：当程序运行时，如果 klib 是动态库，动态链接器（如 ld.so）会负责解析函数调用到它们的实际地址。如果 klib 是静态库，所有的函数调用已经被解析并包含在可执行文件中。

通过这种方式，定义 `NATIVE_USE_KLIB` 宏确保了在 native 平台上编译的程序会使用 klib 提供的库函数实现，而不是依赖于系统的 glibc 或其他标准库实现。这样做可以测试 klib 的实现是否正确，也可以在不改变代码的

情况下比较 klib 和 glibc 的性能差异。

如果不定义 **NATIVE_USE_KLIB** 宏，程序会如何链接到 glibc？

如果不定义 **NATIVE_USE_KLIB** 宏，程序会链接到系统的 glibc (GNU C Library)，这是 Linux 系统上的标准 C 库实现。这个过程涉及到几个关键的链接步骤：

编译时的宏未定义：当编译器编译代码时，如果 **NATIVE_USE_KLIB** 宏没有被定义，那么条件编译部分（通常使用 `#ifdef NATIVE_USE_KLIB` 来检查宏是否定义）将会被忽略，这意味着 klib.h 中的库函数声明不会覆盖 glibc 的声明。

编译器的默认行为：编译器在编译时会查找标准库函数的声明，这些声明通常在编译器提供的头文件中定义。例如，`printf` 函数的声明会在编译器包含的 `stdio.h` 中找到。

链接器的默认库：在链接阶段，链接器（如 `ld`）会根据编译器生成的符号引用来解析函数和变量的地址。如果 **NATIVE_USE_KLIB** 宏没有定义，链接器会根据默认的库路径（通常是 `/usr/lib` 或 `/usr/local/lib` 等）来查找 glibc。

库的查找顺序：链接器会根据链接指令和库的路径来查找库。如果没有特别指定库的路径，链接器会按照环境变量 `LIBRARY_PATH` 或默认的库路径来查找库文件。

静态链接和动态链接：

静态链接：如果 glibc 被静态链接，链接器会从 glibc 的静态库版本（通常是 `libgcc.a` 或 `libc.a`）中提取所需的对象文件，并将其包含在最终的可执行文件中。动态链接：如果 glibc 被动态链接，链接器会创建一个包含运行时动态库加载信息的可执行文件。在程序运行时，动态链接器（如 `ld-linux.so` 或 `ld.so`）会负责解析函数调用到它们的实际地址，并加载 glibc 的动态库版本（通常是 `libgcc_s.so` 或 `libc.so`）。运行时解析：对于动态链接的程序，动态链接器在程序启动时解析程序中的符号引用到 glibc 中的实现。对于静态链接的程序，所有的符号引用在链接时已经解析完成。

通过这种方式，如果不定义 **NATIVE_USE_KLIB** 宏，程序将默认使用 glibc 中的库函数实现，这是大多数 Linux 应用程序的标准行为。

NEMU的跑分结果

第一次跑了好几万分甚至比native真机跑的都快，显然出现问题，后来查看源码后发现在 `rtc_io_handler` 中，有这么一句判断：`offset==4`

`if (!is_write && offset == 4) { uint64_t us = get_time(); rtc_port_base[0] = (uint32_t)us; rtc_port_base[1] = us >> 32; }` 意思是获取一次系统时间会触发两次这个回调函数，加一个判断避免重复 `get_time`。因此在 `__am_timer_uptime` 中 `inl` 的顺序应该是先读高32位，让他获取到当前系统时间，然后再读低32位。而当时我先读取的低32位导致 `rtc_port_base` 没有更新，获取到的是上一次 `__am_timer_uptime` 得到的系统时间的低32位，因此跑分出错。修改方法有两个

`rtc_io_handler` 中的判断改成 `offset==0` `__am_timer_uptime` 中先获取高32位 修改完成后，跑分大概是两百多（虽然跟native的一万四千多分没法比），但也算合理的

在游戏中，很多时候需要判断玩家是否同时按下了多个键，例如 RPG 游戏中的八方向行走，格斗游戏中的组合招式等等。根据键盘码的特性，你知道这些功能是如何实现的吗？

在游戏中实现多键同时检测（也称为“键位组合”或“组合键”）通常依赖于游戏引擎或开发框架提供的输入系统。这些系统能够追踪键盘上每个键的状态，并允许开发者查询这些状态。以下是实现这一功能的一些基本步骤：

1. 键盘状态追踪：游戏的输入系统会维护一个键盘状态的记录，通常是一个包含所有键码的数组或哈希表。每次键盘事件发生（如按下或释放）时，系统会更新这个记录。
2. 查询键状态：开发者可以通过查询这个记录来检查一个或多个键是否被按下。例如，如果想知道玩家是否同时按下了“上”和“右”键，可以查询这两个键的状态。
3. 实现组合逻辑：在游戏中，开发者会编写逻辑来判断是否满足了特定的键位组合。例如，如果需要检测“上”和“右”键的组合，代码可能会像这样：

```
python
if is_key_pressed("UP") and is_key_pressed("RIGHT"):
    # 执行八方向行走的逻辑
```

其中 `is_key_pressed` 是一个假设的函数，用来检查特定键是否被按下。

4. 处理键盘布局和多键滚动：不同的键盘可能有不同的布局，而且有些键盘硬件可能不支持检测超过一定数量的键同时按下（这被称为“键位滚动”或“N键滚动”）。游戏的输入系统通常需要处理这些问题，以确保在不同硬件上都能正常工作。
5. 使用游戏引擎或框架的API：大多数游戏引擎或框架（如Unity、Unreal Engine、Godot等）都提供了自己的输入管理API，这些API简化了键位组合的检测。例如，在Unity中，可以使用 `Input.GetKey` 方法来检查单个键的状态，然后组合这些调用来检测多个键。
6. 优化体验：为了提升游戏体验，开发者可能还会实现一些额外的功能，比如按键缓冲（允许在短时间内检测到快速的按键组合），或者为不同的按键组合提供不同的响应时间。

在实际的游戏开发中，这些功能的具体实现细节会根据所使用的编程语言、游戏引擎或框架而有所不同。不过，基本原理是相似的：追踪键盘状态，并在需要时查询这些状态来判断玩家的输入。