2012543-刘沿辰-PA2报告

学号：2012543
姓名：刘沿辰
专业：计算机科学与技术
指令集：RISC-V 32
日期：2023.4.3

v-2额外说明

在先前提交的版本中，我的isa部分指令实现存在少量问题，导致最终运行程序时可能会出现各种未定义行为，甚至代码区被修改，因此重新制作了v-2版本来提交。

实验目的

- 实现isa，正确运行调试程序
- 实现常用库函数以及各种trace
- 完成串口、时钟、键盘和VGA等输入输出工作

实验内容

- 第一阶段：实现更多指令，在NEMU中运行大部分`cpu-test`
- 第二阶段：实现klib和多种基础设施
- 第三阶段：运行FCEUX，提交完整的实验报告

实验过程

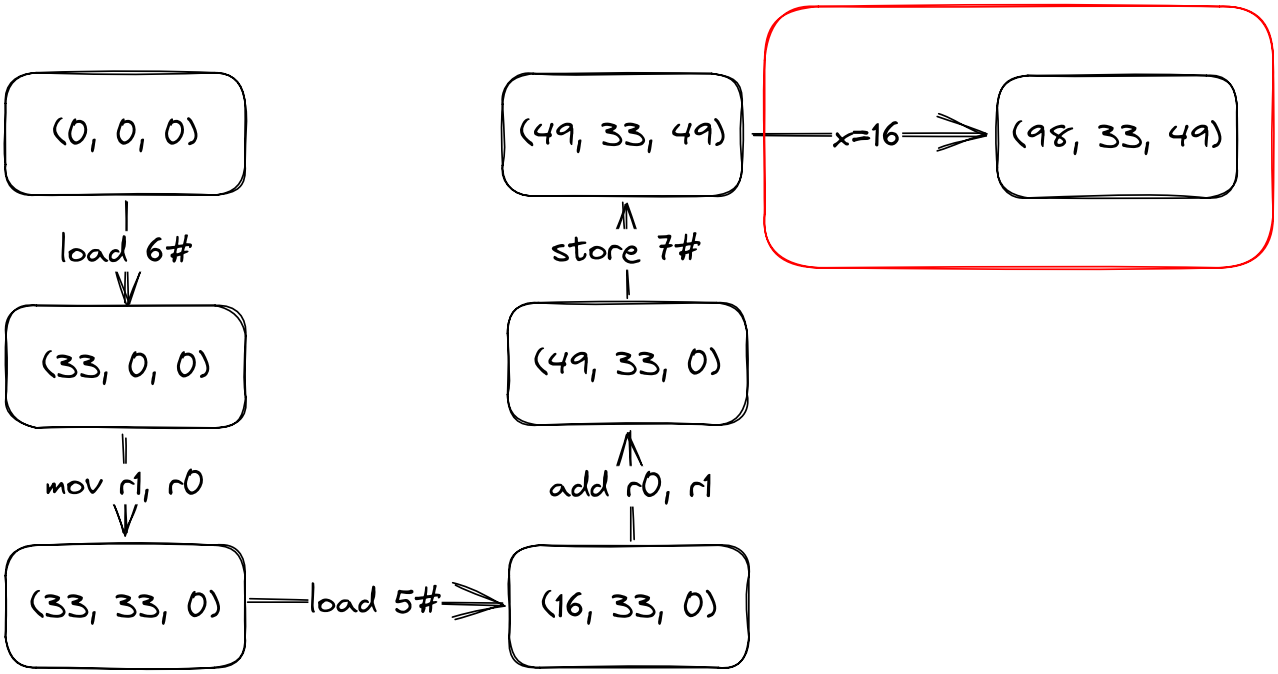
如何理解YEMU程序的执行

状态机

为了理解YEMU程序的执行，我们来画出YEMU执行加法程序的状态机：

- 状态是程序运行到某个阶段的一组参数值，在加法器中存储会改变的部分有：
  - R[0]
  - R[1]
  - M[7]
- 因此，我在绘制状态图时将使用三元组`(R[0], R[1], M[7])`表示程序的状态

- 状态机如下：



> 程序的代码和数据是共同在内存中存放的，因此错误的读取不会被检查

YEMU单指令执行

通过RTFSC，可以看到YEMU模拟器中包含如下几个模块：

- 寄存器与内存
- 指令格式定义
- 指令解码
- 执行一条指令的函数
- 主函数循环
  整体调用关系呈现自底向上的顺序，因此我们从最底层说起。

寄存器与内存：

- YEMU有4个8位寄存器（标号从0到3），16字节的内存，并在开始时将内存初始化为：

```
0b11100110,  // load  6#     | R[0] <- M[y]
0b00000100,  // mov   r1, r0 | R[1] <- R[0]
0b11100101,  // load  5#     | R[0] <- M[x]
0b00010001,  // add   r0, r1 | R[0] <- R[0] + R[1]
0b11110111,  // store 7#     | M[z] <- R[0]
0b00010000,  // x = 16
0b00100001,  // y = 33
0b00000000,  // z = 0
```

指令格式定义：

- YEMU指令长度为8位，且只有R型和M型两种指令格式，结构如下：

```
R型：|--|--|----|，结构为：|rs|rt|op|
M型：|----|----|，结构为：|addr|op|
```

指令解码：

- 指令解码的实现依托于指令格式的定义，分别为R型指令解码和M型指令解码
- 程序提前定义了`rt`，`rs`和`addr`变量，解码时将对应位置的值赋给该变量即可

执行一条指令的函数：

- YEMU定义了`exec_once()`函数用于执行一条指令
- 函数伪代码如下：

```
void exec_once() {
  根据pc指针取指令
  根据指令类型进行解码和操作，调用解码部分函数
  更新pc指针
}
```

- 该函数在解码操作部分选择先解码后进行运算。由于程序提前定义好了`rt`，`rs`和`addr`变量，且变量在解码阶段会被更新为需要的值，因此在解码后直接对`rt`，`rs`或`addr`操作即可

- 如果函数在解码时发现不存在这个操作码，将`halt`变量置为1（在主函数循环中用到）

主函数循环：

- 主函数包含一个死循环，不断执行`exec_once()`函数
- 死循环的跳出条件是`halt`变量被置为1
- 因此程序会从第一条指令开始一直执行指令，直到遇到内存`M[6] = 0b00010000`

根据上述流程，YEMU就可以执行指令了！如果我们将目光局限在主函数的**一次循环中**，那么我们就可以看到程序执行*一条指令*的过程。

> 但是有一个明显的问题：当YEMU运行到`M[5] = store 7#`指令后并不会停止，而会继续将后一个地址的`M[6] = 0b00010000`这个数字当成指令来继续运行！这个指令解码之后是`add r0, r0`。如果指令可以执行的话，`r0`寄存器的值将翻倍！

两者的关系

程序每执行一条指令都经过了[YEMU的单指令执行](#### YEMU单指令执行)部分的流程，而这条指令执行的结果就是程序状态的一次转移。可以在状态机图上清晰地看到，每个状态转移的箭头上都有一条指令，那就是导致本次状态转移的指令！

RISC与CISC?

斯坦福大学的学者在调研了过去40年的各种CPU之后，经过进一步研究发现：**过去20年半导体工艺与CPU体系结构的进步对CPU性能的飞速增长各贡献了一半**。半导体工艺的进步在过去五十余年中均由摩尔定律准确预言，但是由上面提到的结论可知，CPU体系结构的进步给CPU性能带来的提升和摩尔定律可以相提并论！

说到体系结构，就不得不提到复杂指令集（CISC）和精简指令集（RISC）。CISC和RISC是由帕特森老爷子在上世纪八十年代左右提出的术语，用于描述不同指令集的设计遵循的简洁性原则。两种架构从诞生之初便有着各种各样的概念分歧：CISC旨在减小代码体积，将复杂性从软件转移到硬件设计中；与之对应地，RISC选择将复杂度还给软件，并通过有限的指令和更高的IPC来做好一件事情。在我自己的看法中，CISC指令集本身是为了精简代码。但由于代码的拓展性和指令集自身的复杂性，代码随着时代的进步一定会变得更加复杂，CISC也会由于向上和向下的兼容性而变得越发复杂——直到难以继续维护或添加功能。

时至今日，这两种架构的CPU都在市场上有所作为，也有所交融。比较有代表性的例子是Intel公司的x86芯片依然在主机上有着绝对的领先地位，而苹果等公司已经开始将目光转向RISC，并尝试将其应用到所有产品中。当我们反观两者的发展历史时，CICS架构的工程师们开始尝试使用“类RISC”解码器来简化复杂的微结构；RISC架构的工程师们在设计CPU时引入了乱序执行。这些设计方式实际上与RISC或CISC的设计初衷完全不符，但是确实取得了更好的效果。

对此，Jon Stokes在发表于1999年的文章《RISC vs. CISC: the Post-RISC Era》中指出，RISC和CISC两种分类方式早已过时，我们应该着眼于“以实现为中心”和“以ISA为中心”两种分类方式。在这种观点中，我们比较CPU应该着眼于*制程节点、微架构和ISA*三个方面，而后一种分类方式要好于前一种传统的分类方式。目前主流的x86制造厂商可以作为“以实现为中心”的代表，他们并不否定ISA的重要性，但他们认为制造CPU的核心还是在于硬件的制造与实现。在这种观点中，其余的要素在大部分情况下应该为硬件的实现让路，依靠制程和制作工艺来达到更好的效果。对应地，在“以ISA为中心”的观点中，CISC的成功依靠的是制造能力和安装优势，否则RISC CPU的表现将会优于CISC CPU。

至于我为什么选择RISC呢？因为它足够优雅。

理解一条指令执行的过程

我想要执行一条指令，于是我在NEMU终端中输入指令`si`，一条内存中的指令就被取出并执行，这期间发生了什么？

- 输入指令后，程序首先进入cpu_exec函数中，函数简要功能如下：

```
cpu_exec(n = 1) {
  检查NEMU状态（是否已经停止运行）
  使用函数execute(n)执行一条指令
  检查NEMU状态
}
```

- 接下来程序进入execute函数，其功能如下：

```
execute(n = 1) {
  使用函数exec_once()执行一条指令
  计数
  检查分支预测
  检查程序状态
}
```

- 实际上，最核心的点就在于exec_once函数的执行，因此我们来分析这个函数：

```
exec_once(Decode *s, vaddr_t pc) {
  s->pc = pc;
  s->snpc = pc;
  isa_exec_once(s);
  cpu.pc = s->dnpc;
}
```

Decode表示指令的解码，包含一条指令执行所需的信息，具体结构为：

```
Decode {
  pc（pc寄存器）
  snpc（预测的下一条指令地址）
  dnpc（实际的下一条指令地址）
  isa（包含指令本身）
}
```

解码信息被传入到isa_exec_once函数中，开始指令的不同执行阶段：

- **取指**阶段：函数使用`inst_fetch`将指令加载到Decode中
- **解码**阶段：调用`decode_exec`将指令操作和操作数解码
- **执行**阶段：依然在`decode_exec`函数中，一旦确认了操作码和操作数就直接进行对应操作
  所以我们可以看到，常规的*取指-译码-执行-访存-回写*这五个阶段并没有被明确的区分出来，这是因为单周期CPU本身不需要考虑分阶段，只需要分段清晰即可。

立即数背后的故事(2)

> mips32和riscv32的指令长度只有32位，因此它们不能像x86那样，把C代码中的32位常数直接编码到一条指令中．思考一下，mips32和riscv32应该如何解决这个问题?

答案是使用I型指令加载低12位，用U型指令加载高20位。
而我最开始的猜测是使用一个指针之类的结构来处理。但是仔细思考的话，这样的结构会导致代码之外的额外空间被开辟，破坏了设计的统一性。看似简单，实际上导致操作难度大大提升。

指令未定义会发生什么?

在指令匹配规则中有一条优先级最低的inv匹配规则，匹配未定义的非法指令，然后转入hostcall.c文件中，输出错误信息。

愚蠢的错误

这是一个困扰我不少时间的问题：

```
INSTPAT("??????? ????? ????? ??? ????? 01101 11", lui, U,
R(dest) = imm);
```

如上，lui指令是将立即数加载到寄存器高位，为何此处不需要左移12位?

答：早在decode_operand函数中就已经移动了。其中使用的`immU, immI, immS`宏已经完成了这个操作。

mulhu指令的实现

大体上，第一部分指令的实现照葫芦画瓢即可，此处有一条实用C++特性如下：

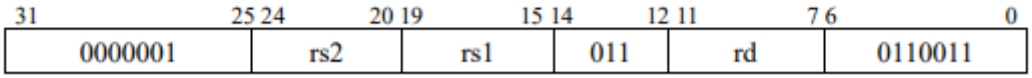
- 对于`int32_t`类型变量，程序在执行`>>`操作时会自动符号扩展，而对于无符号（默认情况）则会零扩展。
  上述特性对于逻辑左移、强制转换等均适用，使用好这一点可以大大减少代码量。

在测试long数据相乘时，我们需要用到一条`mulhu`指令，Fxxking Manual内容如下：

mulhu rd, rs1, rs2 $x[rd] = (x[rs1] \times_u x[rs2]) \gg_u XLEN$

高位无符号乘*(Multiply High Unsigned)*. R-type, RV32M and RV64M.

把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，x[*rs1*]、x[*rs2*]均为无符号数，将乘积的高位写入 x[*rd*]。



注意到相乘时需要符号，我写出了第一个版本的实现：

```
R(dest) = ((int32_t)src1 * (int32_t)src2) >> 32;
```

于是立即报错（）

我开始思考，是不是位移的原因？int只有32位，位移32位这个操作是不允许的，然后写出了另一个版本：

```
R(dest) = (int32_t)(((long long)src1 * (long long)src2) >>
32);
```

很微妙！两个32位的数做乘法，当然需要一个64位的`long long`来存储，位置也足够，位移也不会出问题。于是这一次程序没有报错，但是最终运行结果出错了。

> 好的，细节出现了，在我把操作数强制转`long long`的时候，由于无符号数会被默认零扩展，所以强制转换出的`long long`已经不是原来的数字了，此处要使用符号扩展！

那么如何进行符号扩展呢？有两种方案：

- 框架代码中已经提供了`SEXT`符号扩展宏，可以直接对`long long`使用，如下：

```
R(dest) = (int32_t)((SEXT((long long)src1, 32) * SEXT((long
long)src2, 32)) >> 32)
```

- 刚才提到，C++对有符号数做强制类型转换时，会自动进行符号扩展，可以借助这个特性，先将操作数转换为有符号整型，然后再转换为`long long`，如下：

```
R(dest) = (int32_t)(((long long)(int32_t)src1 * (long long)
(int32_t)src2) >> 32)
```

经测试，上述两者均可使用。其他的指令实现起来都比较简单，实现的指令也通过了除`string`相关用例外的所有测试，此处不加赘述。

程序批处理

我们每次打开nemu时，都需要输入一个`c`指令来让程序运行。如果想要省去这一步骤，则需要我们设置nemu的批处理选项。程序总是再main函数处开始运行，main函数捕获参数列表后传给parse_args处理，此处`-b`选项即代表批处理，因此我们需要在`machine/script/platform/nemu.mk`中添加`NEMUFLAGS += -b`。

klib实现

大部分依然是照葫芦画瓢，略去不提。不过有一个困扰了我好久的bug，如下：

```
// 这是我实现的memcmp函数
int memcmp(const void *s1, const void *s2, size_t n) {
  char *t1 = (char *)s1, *t2 = (char *)s2;
  while (*t1 == *t2 && n--) {
    t1++;
    t2++;
  }
  return *t1 - *t2;
}
```

这段代码在我写的时候完全不觉得有任何问题，相似的实现逻辑我也用在了`strcmp`函数上，而`strcmp`函数在测试的时候没有被发现问题，我就顺理成章地用在了这里，结果就是FAIL！！

如何发现问题呢？首先我们把这段代码看成一个状态机，其变量为t1, t2和n。我们假设`n = 1`，且两个字符串相同。程序在第一次循环时，正确进入循环，`*t1 = *t2`且`n--`。到了第二次进入循环时，程序检测到`n == 0`，因此退出循环，返回`*t1 - *t2`。但是注意，此时的`t1`与`t2`已经在第一次循环中+1，所以实际上比较的是下一个字符，因此出现了问题。

多么愚蠢的错误。但是这个错误在我写代码时，review时甚至最开始的`strcmp`函数测试时都没有出现明显的问题。软件工程课上老师讲为什么需要软件工程时，说答案是早先程序员纯在自己脑子里构件程序，最后酿成了一场灾难。在程序测试中也是类似的，不要太相信自己，合理的架构与测试才是代码正确运行的基础——让正确的运行起来是很困难的，没有Hello-World一般的理所当然。

mtrace

首先我们需要在`nemu/Kconfig`文件中加入：

```
config MTRACE
bool "Enable mtrace"
default y
```

用于设置mtrace是否打开。然后在vaddr_read和vaddr_write中插入

```
#ifdef CONFIG_MTRACE
log_write("Address %#.8x: read/write %d bytes", addr, len);
#endif
```

即可实现mtrace。

性能测试

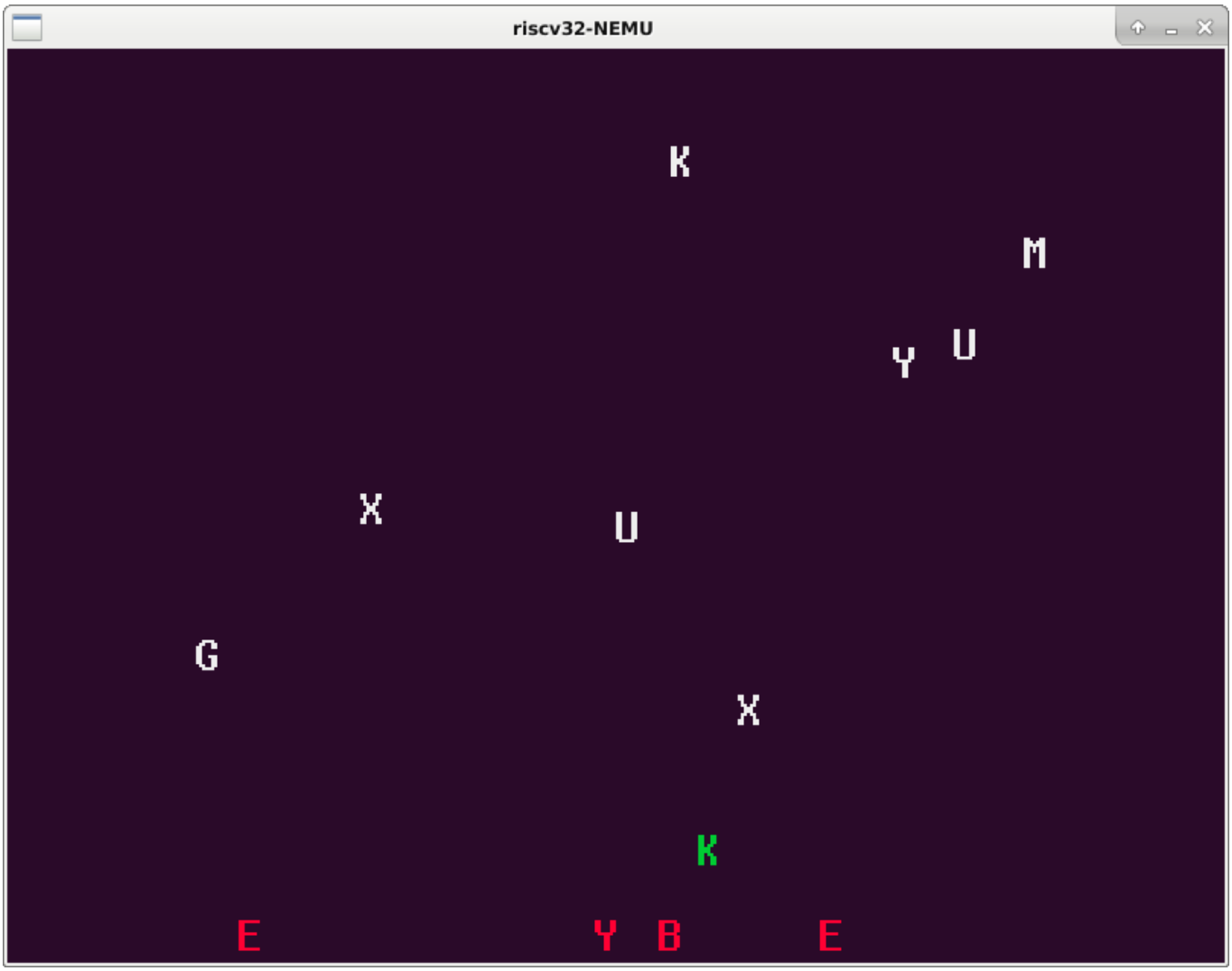
在microbench上测试NEMU的性能，结论如下：

```
MicroBench PASS        391 Marks
                  vs. 100000 Marks (i9-9900K @ 3.60GHz)
```

对比身边同学们的x86，性能提升了大概一倍！（主机为4核8线程的2020款Matebook 14'，虚拟机分配8GB内存与4线程处理，开启了Windows专业工作站版本提供的Ultimate Performance，性能表现还不错）

测试游戏

完成IO部分后即可运行各种测试小游戏，测试打字游戏效果如下：



运行成功，这也从侧面验证了IO的实现正确性！

小问题

在我刚刚实现完这部分代码后，我发现运行所有程序都巨卡无比，打字小游戏只有1帧左右。但是当我重启虚拟机与主机之后这个问题就消失了。与身边的同学探讨了一下，这可能是由于虚拟机的内存不足导致的，而虚拟机的内存不足又可能是VMware管理方式导致的。

Makefile组织方式

Makefile组织方式大体如下：

- 首先是类似预编译的过程，讲include的所有makefile放到一起
- 初始化所有变量并推导隐式规则
- 根据依赖关系生成目标

具体地，例如在abstract-machine中：

- MakeFile首先收入include的内容
- 检查环境并根据输入抽取参数
- 在build文件夹中新建riscv32目录，存放.d和.o文件
- 初始化编译选项
- 使用编译选项选择的编译器将.c文件编译为.o文件，生成可执行文件
- 存储可执行文件，make结束