# 南大PA实验的LoongArch支持

全国大学生计算机系统能力大赛

初赛阶段性汇报

项目成员：李宗逸 刘奕钊 李智锋

指导老师：谢文兰

学校：广东培正学院

全国大学生计算机系统能力大赛

操作系统赛

OS功能挑战赛道

2024年5月

# 摘要

本项目为 2024 全国大学生计算机系统能力设计大赛-操作系统设计赛功能挑战赛道的 241 号赛题：la-pa，项目名称：南大 PA 实验的 LoongArch 支持。本项目由龙芯中科技术股份有限公司和中国科学院计算技术研究所提供技术支持，由广东培正学院的 run 的全都队设计并实现。

PA(Programming Assignment)是南京大学计算机系统基础课程的小型项目。旨在让学生从"零"开始实现一个完整的计算机系统。以深刻理解程序在计算机上的运行过程。目前，PA 官方已经完成了 x86、MIPS 和 RISC-V 版本的框架适配与文档引导，并对 Loongarch32r（以下简称为：LA32r）有了初步的支持。但由于缺少文档引导和出于对于 LA32r 这一国产新生架构的陌生感，暂时应该没有人使用 LA32r 作为完成 PA 的目标架构。

为推进国产指令集 LoongArch 的生态建设，本团队在使用 LA32r 作为完成 PA 的目标架构的同时，力求完善 PA 的框架代码与编写 Loongarch 架构相关的文档，让往后的同学做 PA 实验选择 LA32r 作为目标架构时，上手难度能与 MIPS、RISC-V 基本持平。

项目进展如下：

1、PA1：该部分实现与架构无关，已完成 PA 讲义中本章节的所有内容。包括 sdb 的单步调试、表达式求值、内存打印以及监视点设置。

2、PA2：完成了讲义中除声卡外的所有内容实现。具体工作包括指令集的扩展、基础设施的强化以及运行时环境的构建。

3、PA3：在 nemu 中实现了异常处理支持，确保其能够正确处理和响应异常情况。

4、PA4：正在并成功运行了 rt-thread（选做部分）。

5、适配工作：针对 LA32r 进行了全面的适配，包括指令集的添加、外设模拟（如键盘、时钟、串口和 VGA）以及系统调用和异常处理机制的支持。

本文共分为四个章节：

第一章，项目介绍。本章介绍 la-pa 的项目背景，包括 PA 的具体实验内

容、实验设计理念，以及完成 PA 对 Loongarch 架构适配的意义。

第二章，团队协作。本章介绍在本次项目开发过程中，我们团队的分工和工程管理方式。

第三章，实现与移植。本章介绍团队实现相关实验内容的思路与难点，以及 LA32r 的具体适配内容。

第四章，回顾与展望。本章是从 4 月下旬至今，团队对 la_pa 项目的回顾与展望。

# 目录

# 图目录

# 表目录

# 第一章 项目介绍

本章主要基于对 PA 官方讲义的整理和理解，结合目前已完成的 PA 实验内容和 LA32r 架构适配过程中的感悟，对项目的内容与意义进行介绍。

## 1.1 PA 是什么

PA (Programming Assignment, PA)，南京大学计算机科学与技术系计算机系统基础课程的小型项目。该项目将提出 x86/mips32/riscv32(64)架构相应的教学版子集，指导学生实现一个经过简化但功能完备 x86/mips32/riscv32(64)模拟器 NEMU(NJU EMUlator)，最终在 NEMU 上运行游戏"仙剑奇侠传"，来让学生探究"程序在计算机上运行"的基本原理。

从整体上看，项目被划分成了五个部分：PA0、PA1、PA2、PA3、PA4。

- PA0：主要目的是配置实验环境，同时让学员初步熟悉 GNU/Linux 下的工作方式。
- PA1：热身章节，主要是实现框架代码和实现一个简易的调试器用于后续。
- PA2：添加指令，强化基础设施，实现运行时环境。
- PA3：nemu 支持异常处理，实现批处理系统，能运行仙剑奇侠传。
- PA4：实现支持虚存管理的抢占式多道任务系统、跑通 rt-thread（选做）。

## 1.2 项目意义

本科计算机系统方向中有两个终极问题：“程序如何在计算机上运行？”、“程序如何在计算机上高效运行？”。为了解答这两个问题，我们需要探索的知识领域正是本科计算机教育的核心课程所涵盖的内容。从计算机系统的抽象模型上看，这些核心课程从上到下跨越了多个层次，通常也会被分成多门课程进行传授，如：数据结构与算法、数字电路、计算机组成原理、操作系统等。但基本上，这些课程在被传授的过程中都是各讲各的知识点，没有一门课用具体的例子把这些知识点串起来。这些知识点的串联全靠学生在学完各个独立课程后自己悟，效果并不太好。只有亲自实现一个完整的计算机系统，

并在其上运行真实的程序，才能明白系统栈每一个层次之间的关系，才会对“程序如何在计算机上运行”有深刻的认识。基于以上原因，PA 作为一个打通计算机系统全栈的实验，应运而生。

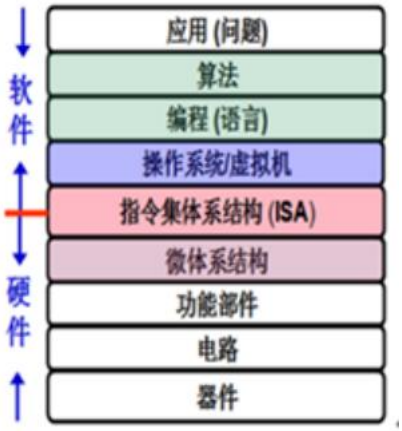



图 1 计算机系统抽象模型

自 2019 年美国首次将华为及其 68 家关联企业列入美国商务部出口管制“实体清单”以来， 我国在信息技术领域的自主可控能力的重要性日益凸显。为了减少对国外技术和供应链的依赖，加强自主研发和技术创新，龙芯中科推出了 LoongArch（龙架构），这一举措有助于我国提高自主研发和创新能力，更好地保障国家的信息安全，同时也提高了我国在全球信息技术领域的竞争力。

而给 PA 项目进行 LoongArch 架构适配不仅有助于完善龙芯的生态系统，增强其在国内外市场的竞争力和影响力； 现在国内 MCU 广为流行的 stm32 ，就是在因为意法半导体（STMicroelectronics）提供了丰富的教程和资料，吸引了大量的用户，特别是高校电子信息类的在校生。这从根本上培养学生对相关技术产品的使用习惯和忠诚度。所以在教育层面上推广 LoongArch 架构，可以达到同类似意法半导体的策略类似的效果，并不断建立和扩大龙芯生态；培养一批熟悉国产技术的专业人才，为我国信息技术产业的发展提供人力支持。

## 1.3 设计思路

为了让学生真正理解程序如何在计算机上运行，余子濠老师采用了一种独特的教学方法——“以史为鉴”，旨在引导学生深入理解“程序如何在计算机上运行”。通过向学生介绍计算机的发展历程，特别是计算机硬件和软件的协

同进化，让学生能够从历史的角度理解计算机系统的演变和当前技术的基础，切身感受计算机硬件的发展需求和程序越发复杂的运行需求之间的相互影响。

通过 nemu 的搭建和使用，可以让学生亲身体验计算机系统的各个层次之间的关系，从而更深入地理解“程序如何在计算机上运行”。进一步锻炼学生的专业技能，正确理解专业知识点，形成良好的全局系统观和专业世界观，知道如何找到相关的代码、资料以及工具等，学会如何分解复杂任务，理解做什么事情是正确、高效以及科学的，促使他们勇敢地投身于探索性项目或开源项目的研发之中。

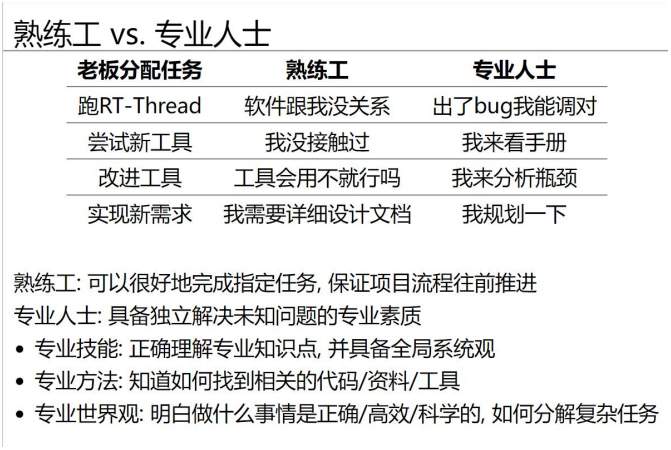



图 2 熟练工与专业人士遇到问题时的应对

为进一步提高学生的专业世界观和解决问题的能力，余子濠老师还对实验文档进行了精心的设计，让学生们在实验过程中不仅锻炼到了阅读和分析的能力，还有利于激发他们的探索精神，借此鼓励他们通过不断试错来一步步深入理解问题，从而解决问题及分析过程中所遇到的一系列“新问题”，提高面对复杂问题时的独立处理能力。

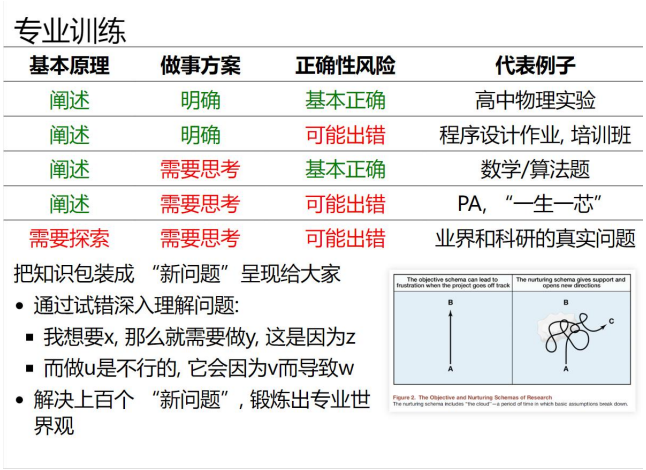



图 3 专业训练

在这种综合的学习策略下，学生不仅能够深入理解程序在计算机上的运行机制，还能够培养出面对未知挑战时的探索精神和解决问题的能力。这样的学习过程有助于学生在完成 PA 实验的同时，达到更高的学术和专业水平。

# 第二章 团队协作

本章主要介绍我们团队在“实验设计思路”和“项目意义”两小节的指导思想下，所定制的团队协作方式。

## 2.1 团队分工

团队成员基础情况：

**李宗逸**：完成过 RISC-V 架构的 PA，熟悉计算机系统结构和操作系统原理，擅长项目统筹与规划，具备较强的资料调研和问题解决能力。

**刘奕钊**：无相关技术背景，擅长写文档编写，工程能力较差。

**李智锋**：尝试做过 RISC-V 架构的 PA，相关基础比较薄弱。

基于以上情况，为了让 PA 实验及讲义的意义发挥到最大，我们团队做出以下分工：

**李宗逸**：不参与或者尽量少参与到项目进度的推进中来，仅负责项目的统筹规划、资料调研和架构适配，以让其他两位队员在用 LA32r 做 PA 时体验尽可能与用其他架构的体验持平。并在不违反 PA 原有设计思路的基础上，给予一定的引导和帮助。

**刘奕钊**与**李智锋**一同分工协力完成 PA 讲义中内容。其中由于近一个多月来刘奕钊琐事缠身，故主要负责部分基础设施的开发及所有基础设施的测试，李智锋则作为项目的推进主力进行开发。

## 2.2 工程管理

由于团队三人身处不同的地理位置，无法进行线下交流，我们采取了一系列措施来进行有效的工程管理和沟通。

首先，我们使用 GitHub 作为主要的项目管理和代码托管平台，利用其丰

富的功能来支持团队开发：

- 代码托管和版本控制：所有工程源码都托管在 GitHub 上，团队成员可以方便地进行代码提交、审查和合并。通过 Git 分支管理策略，我们确保每个功能特性都在独立的分支上开发，合并到主分支之前进行代码审查（Code Review）以保证代码质量。
- Pull Request：开发完成后，通过 Pull Request 提交代码进行审查。团队成员可以在 Pull Request 中讨论代码实现，提出修改建议，并最终通过审查合并到主分支。
- 其次，我们每天都会在群里交流，讨论解决当天遇到的问题，平均每周召开两次线上会议，通过在腾讯会议中进行讨论，及时复盘和调整计划，确保每个成员都清楚当前的项目状态和下一步计划。

## 2.3 协作过程

在协作过程中，我们主要使用飞书作为日常项目进度和文档记录的平台。飞书不仅支持即时消息和文件传输，还提供了强大的文档协作功能，使得我们可以在同一个文档中同时编辑和评论。此外，飞书的多维表格也帮助我们更好地跟踪项目进度，确保每个成员的工作都能得到及时的反馈和调整，也为我们的复盘提供了可视化的数据基础。


图 4 项目数据统计图

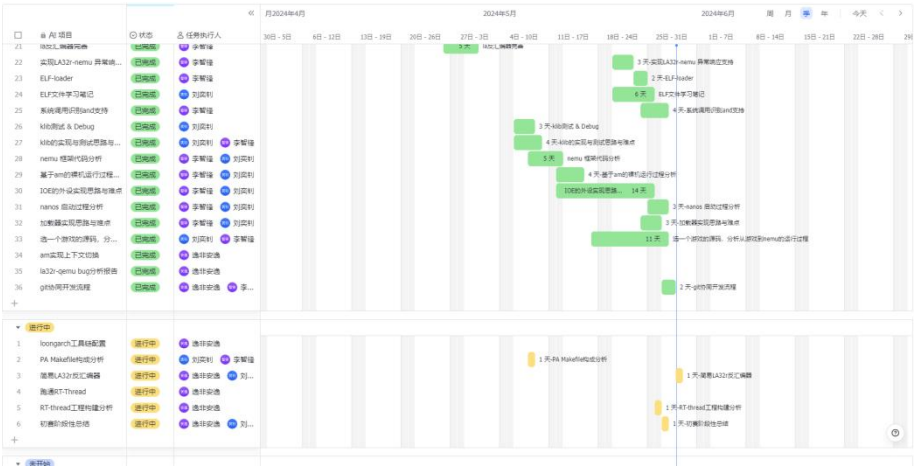


图 5 进展甘特图

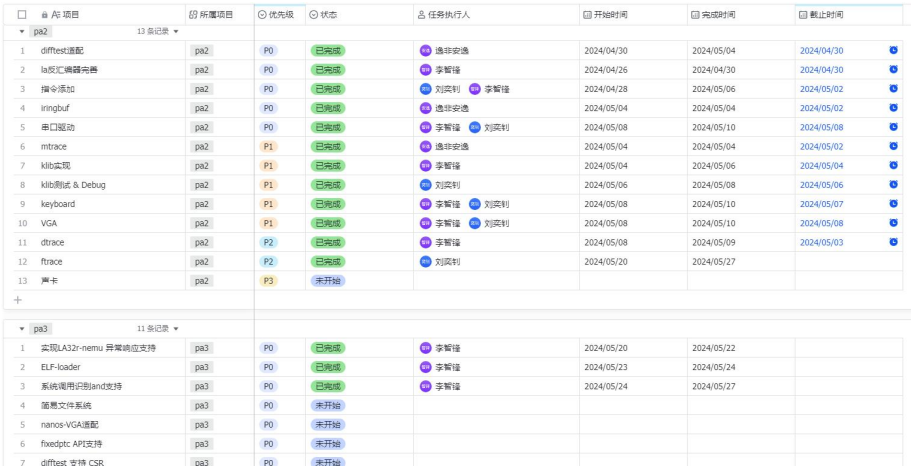


图 6 任务情况表

# 第三章 实现与适配

本章主要介绍截至目前为止本团队在 PA 项目中的工作进展，以及在实现这些内容过程中对 LA32r 架构的适配情况。具体包括：在 NEMU 中实现和验证 LA32r 指令集，使得模拟器能够正确执行 LA32r 架构的指令；实现用于辅助调试的基础设施（如：用于差分测试的 difftest 和用于指令追踪的 itrace）；常用外设的模拟（如：键盘和串口），以便在模拟环境中测试和验证外设驱动程序和系统交互，以及开发了 LA32r 架构的反汇编器，用于解决框架代码中 LA32r 的 itrace 输出只有机器码而没有汇编代码的情况，使 itrace 像其他架构般便于调试和分析。

## 3.0 NEMU 的架构

通过对 PA 框架代码的分析，我们可以绘制出 NEMU 的架构图（如图 7），

它主要由三大部分组成：监视器（Monitor）、追踪控制器（Trace Controller）以及 NEMU 本身。

监视器负责执行控制、参数解析、二进制加载以及简易调试等功能；追踪控制器则包括动态追踪、输入追踪、异常追踪、函数调用关系追踪以及内存分配和释放追踪等功能；NEMU 主要包括内存模型和设备模型两大部分，内存模型含有地址解码和内存子元素，用于模拟内存操作；设备模型则包含键盘、时钟、串口和 VGA 等，用于模拟设备操作。

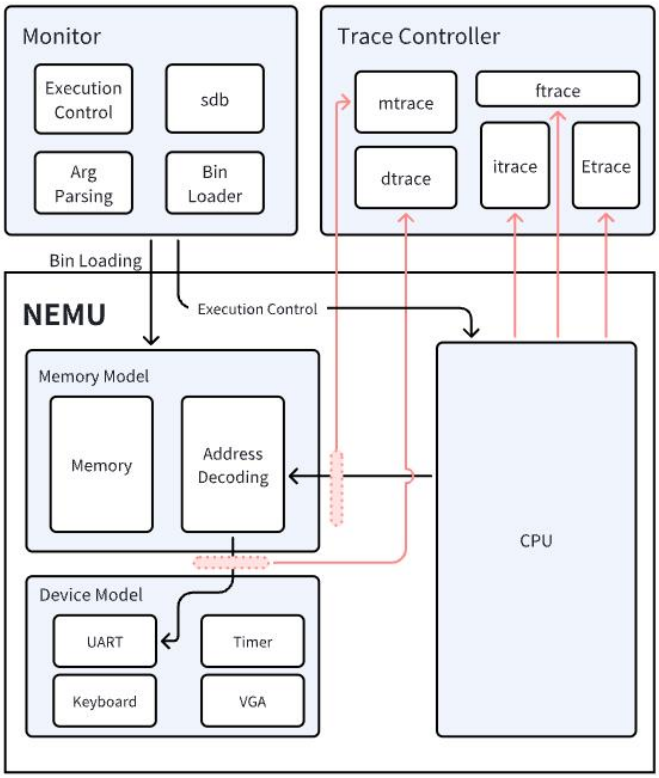



图 7 NEMU 架构图

在 PA 实验手册的安排里，我们所需要实现的部分大致可分为：基础设施实现与测试、指令添加、运行时环境构建、简易操作系统 nanos 等。

## 3.1 基础设施实现与测试

监视器负责执行控制、参数解析、二进制加载以及简易调试等功能，而在基础设施实现部分，我们需要实现的部分包括：Monitor 的简易调试器（sdb）、difftest 以及追踪控制器（Trace Controller）。

### 3.1.1 sdb

简易调试器（Simple Debugger，简称 sdb）是 NEMU 中的一个重要基础设施，可以看作是简化版的 GDB。由于 NEMU 本身是一个执行其他客户程序的程序，它能够直接访问和控制正在执行的客户程序的所有信息，这是外部调试器（如 GDB）难以实现的。为了提高调试效率，同时也作为熟悉框架代码的练习，我们需要在 monitor 中实现一个简易调试器。

**1.实现思路**

简易调试器的功能包括单步执行、内存扫描、表达式求值和监视点设置等。这四个功能及其实现思路如下：

（1）**单步执行**：通过 si 命令和执行参数 N，控制程序执行的指令数量。实现这个功能的思路是将命令行参数 args 传入单步执行函数中，解析需要执行的命令有多少“步”，若传入为空，则默认为 1，最后调用 cpu_exe()函数即可。

（2）**内存扫描**：对客户计算机内存数据的有效访问。实现这个功能的思路是在细致的代码审查过程中，成功识别出关键函数 word_t paddr_read(paddr_t addr, int len)，并根据提供的内存地址 addr 和数据长度 len，执行内存数据的读取操作，并安全返回读取结果。

（3）**表达式求值**：一个简单的表达式求值功能，能够将所输入的表达式进行求值。实现这个功能的思路是将复杂的数学表达式转化为具体的数值结果。实现过程分以下两个步骤：首先识别出表达式中的单元并将其根据类型进行分类、根据表达式的归纳定义进行递归求值。

（4）**监视点**：一个简单的监视点功能，为调试环境提供每条指令执行前后监测特定内存地址变化的能力。实现这个功能的思路是设计监视点(WP)结构体，包含 old_value 与 new_value 等成员，分别记录监视点的原始值与最新值，以供比较之用。此外，增设 use_flag 标志位，标识监视点是否已被激活使用，通过此机制有效管理 wp_pool[NR_WP]中的资源。

其中表达式求值和监视点的实现最具有挑战性。具体实现笔记如下：

- 表达式求值

- SDB 简易调试器：实现监视点

从整体上看，Monitor 把 NEMU 分成了五个状态：stop、running、quit、abort、end。其中图 8 标红部分为 sdb 控制部分。

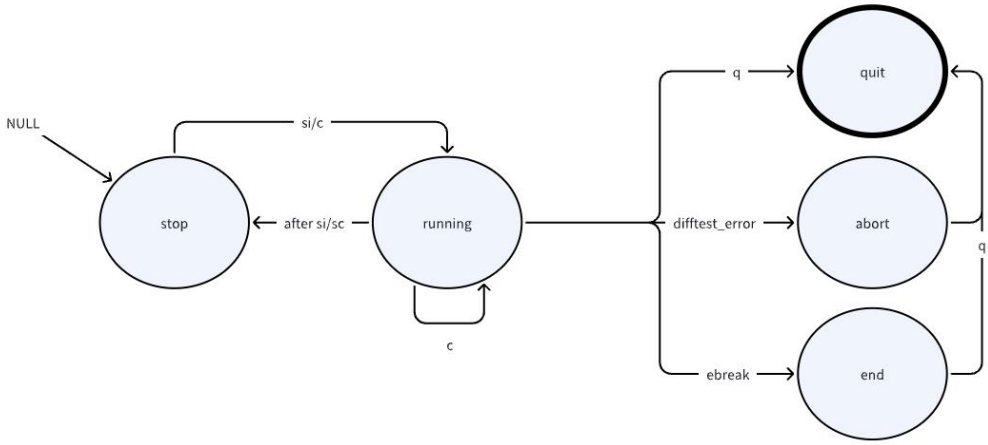



图 8 Monitor 控制下的 NEMU 状态转移图

**2.测试代码**

基础设施出现 bug 所带来的影响远大于项目本身的 bug，因为由工具而产生的 bug 往往会增大我们的分析难度甚至可能无法准确定位到问题关键所在。因此对于基础设施的测试再怎么重视也不为过。本项目对于基础设施所使用的测试大多为随机测试，又称黑盒测试，即通过生成随机、独立的输入样例来进行测试，然后将输出结果与软件规格进行比较，以验证测试输出是否通过或失败。如果没有规格，则使用语言的异常处理机制，即如果在测试执行期间出现异常，则意味着程序中存在缺陷。

在当前进展中，对于工具的测试最主要的集中在表达式求值模块和 Klib 库实现上，以确保工具实现的准确性与稳定性。表达式求值工具可以处理基本表达式计算，同时包含寄存器名和十六进制数的表达式，并将其求值结果以十进制或十六进制的格式进行输出；Klib 库包括内存和字符串的写入函数（如 memset(), strcpy()等）、只读函数（如 memcmp(), strlen()等）以及格式化输出函数（如 sprintf()等）。

（1）**表达式求值测试**：对于表达式求值工具，本项目采用随机生成表达式函数来生成一系列表达式测试样例，并将这些表达式输入到我们的表达式求值

工具中进行求值，通过比较表达式求值工具输出的结果与预期结果，便能够判断这个工具的准确性和稳定性。

一个表达式由数字、运算符、括号及空格组成，因此，我们可以将这些 Token 随机排列组合生成表达式测试样例，如图 11，通过 choose 生成随机数来决定由哪些 Token 进行组合得到完整的表达式测试样例并写入 input 文件中，如图 12，便是生成的随机表达式测试样例（含结果）。

我们在 test 程序中通过文件读取的方式，将这些样例中的表达式输入到 test 的表达式求值 expr 函数中，将结果传入 result 中，然后与 expr 函数的结果 res 使用 assert 进行判断与处理，首先进行符号识别（如图 10 测试样例符号识别过程），若出现无法识别的字符，将会输出该字符并显示“invalid expression”字样，再继续结果判断，若计算结果与测试样例结果不同，则将异常测试样例输出并退出。具体流程如下：

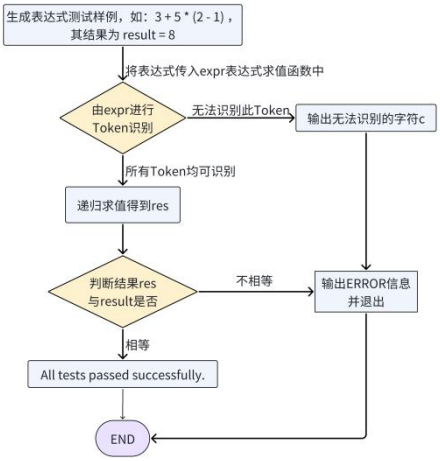



图 9 表达式求值测试流程

| Step | Token | Token Type |
|---|---|---|
| ① | 3 | TK_NUMBER |
| ② | + | ‘+’ |
| ③ | 5 | TK_NUMBER |
| ④ | * | ‘*’ |
| ⑤ | ( | ‘(’ |
| ⑥ | 2 | TK_NUMBER |
| ⑦ | – | ‘–’ |
| ⑧ | 1 | TK_NUMBER |
| ⑨ | ) | ‘)’ |

图 10 测试样例符号识别过程

本次测试通过生成 100 个样例进行测试，最终的测试代码运行结果如图 13 所示：“All tests passed”，从而验证了表达式求值工具的准确性和稳定性。

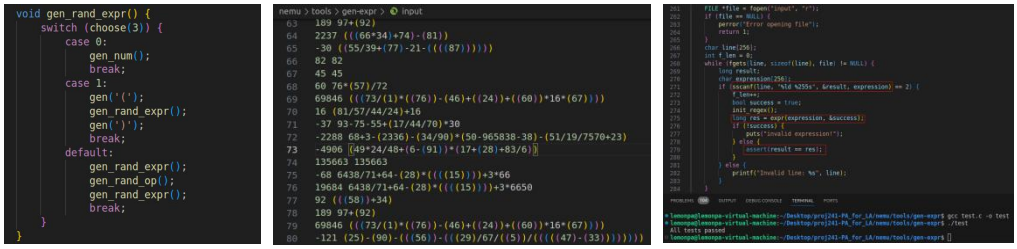


图 11 生成随机表达式 图 12 生成的随机表达式 图 13 表达式求值测试结果

（2）**Klib 库测试**：对于 Klib 库，同样也是采用随机测试方法，通过生成随机字符串作为测试样例，以标准库的函数为标准，对实现的函数进行对比测试（由于在 NEMU 中实现 Klib 库的函数名称与标准库中的函数名称相同，因此在测试时会先将实现的函数名称进行一点小修改以便区分）。

例如，对于 sprintf()格式化输出函数，通过随机生成字符串函数生成测试样例（如图 15），再将随机字符串格式化输出到下测试字符串中，与随机字符串进行判断，观察两个字符串是否一致，从而验证格式化输出函数实现是否完全。具体流程如下图：

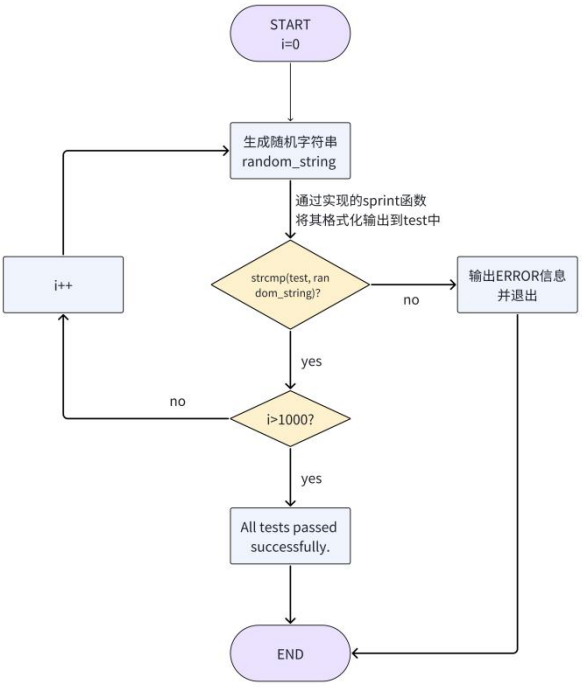



图 14 格式化输出函数测试流程

如图 16 所示，程序将生成的随机字符串格式化输出到 test 字符串中，然后与原字符串 random_string 使用标准库的 strcmp 函数进行判断，重复此操作 1000 次，并无发现异常，则显示“All tests passed successfully.”表示 1000 个测试样例均通过。


图 15 生成随机表达式

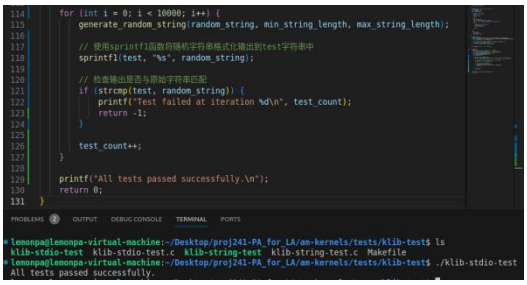



图 16 stdio-test 测试结果

### 3.1.2 difftest

difftest 是一种流行的软件测试技术，它试图通过向一系列相似的应用程序提供相同的输入来检测错误，并观察其执行中的差异。

### 3.1.3 trace 系统

trace 系统用于在模拟器执行过程中记录和追踪指令、内存和设备的访问信息。通过这些记录，开发者可以详细了解程序的执行流程，发现潜在的错误，并优化系统性能。在本项目中，trace 系统主要包括：

- Itrace：用于记录每条指令的执行信息。其核心组件是 iringbuf，一个环形缓冲区，按照先进先出的原则，覆盖旧的信息。遇到错误时，iringbuf 会打印出缓存中的指令信息，以供调试参考。
- Mtrace：可以追踪每次内存访问的结果，并记录访问路径
- Dtrace：用于记录设备访问信息。将设备地址空间的访问信息打印出来，从而获取设备交互的详细记录。
- Ftrace：用于记录函数调用关系。通过解析 ELF 文件的符号表，识别并记录程序中的函数调用和返回信息。使用缩进的方式清晰展示函数调用的层级结构，有助于理解程序的控制流。
- Etrace：

图 18 中，方框表示 NEMU 模拟 CPU 运行过程中涉及到的函数，紫色圆框为相关基础设施的嵌入位置。其中 Itrace、Mtrace、Dtrace、Etrace，只需要在该流程中的相关节点中输出我们想要观测的信息即可，实现起来比较简单。

相对来说，ftrace 的实现就困难了不少。由于 NEMU 上运行的是纯二进制程序，没有任何额外的辅助信息，所以 Monitor 在导入 Bin 文件的同时，需要把 Bin 文件对应的 elf 文件一并导入进来。具体实现思路与难点分析，请看以下我们队员写的文档：ftrace 实现思路及难点。

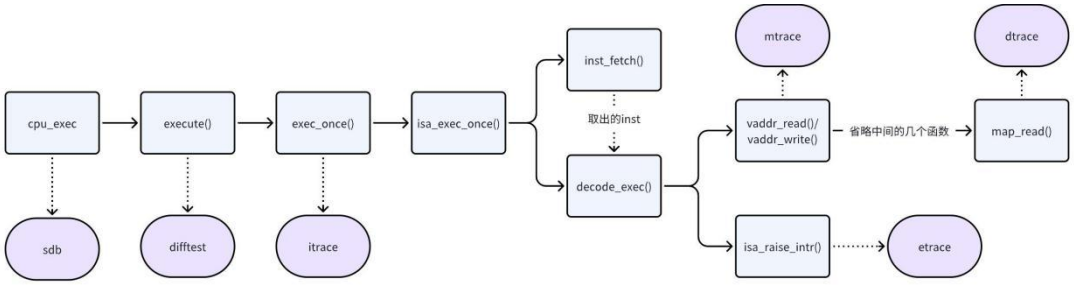



图 17　模拟 CPU 流程图

| 函数名 | 参数 | 功能描述 |
| --- | --- | --- |
| cpu_exec | uint64_t n | 负责控制整个模拟流程，包括启动模拟、统计模拟时间、处理模拟结束的情况等。当模拟开始时，它会记录开始时间，然后调用 execute() 函数来执行指定数量的指令。执行完成后，它会记录结束时间并计算模拟所花费的总时间，并根据模拟的结束状态，打印相关的日志信息。 |
| execute | uint64_t n | 通过循环调用 exec_once() 来执行指定数量的指令。每次执行后，会更新全局变量 g_nr_guest_inst 来记录执行的指令总数，并调用 trace_and_difftest() 来进行跟踪和差异测试。如果模拟状态不是运行状态，那么它会提前退出循环。 |
| exec_once | Decode *s, vaddr_t pc | 执行单条指令。首先设置 Decode 结构体 s 的 pc 字段为给定的物理地址 pc，然后调用 isa_exec_once(s) 来执行指令。执行完成后，更新 cpu.pc 以反映指令执行后的 PC 值。 |
| isa_exec_once | Decode *s | 根据 Decode 结构体 s 中的指令内容来执行相应的指令。具体的执行逻辑会依赖于特定的 ISA 实现。 |

| | | |
|---|---|---|
| inst_fetch | vaddr_t *pc, int len | 从内存中获取指令。读取当前 PC 指向的指令，并返回该指令的值，同时更新 *pc 为下一条指令的地址。 |
| decode_exec | Decode *s | 解码指令。根据 Decode 结构体 s 中的指令内容来确定指令的类型和操作数，然后更新 Decode 结构体以反映解码的结果。 |
| vaddr_read | vaddr_t addr, int len | 从虚拟地址 addr 中读取长度为 len 的数据，并返回读取到的数据。 |
| vaddr_write | vaddr_t addr, const uint8_t *src, size_t len | 将数据从源地址 src 指向的内存位置写入到虚拟地址 addr 指定的内存位置，写入的数据长度为 len。 |
| map_read | paddr_t addr, int len, IOMap *map | 用于处理 I/O 操作，从设备映射表中读取数据，并返回读取到的数据。 |
| isa_raise_intr | word_t NO, vaddr_t epc | 触发一个中断或异常，通过函数接受中断号 NO 和异常发生的地址 epc 作为参数，处理具体的中断或异常。 |

表 1 NEMU 模拟 CPU 运行过程中涉及到的函数

## 3.2 指令添加

LoongArch32r 指令集包含 9 种典型的指令编码格式，即 3 种不含立即数的编码格式 2R、3R、4R，以及 6 种含立即数的编码格式 2RI8、2RI12、2RI14、2RI16、1RI21、I26。指令类型也非常丰富，如整数运算指令（加法、减法、乘法、除法）、浮点运算指令（浮点数的加法、减法、乘法、除法）、逻辑运算指令（与、或、非、异或）等。

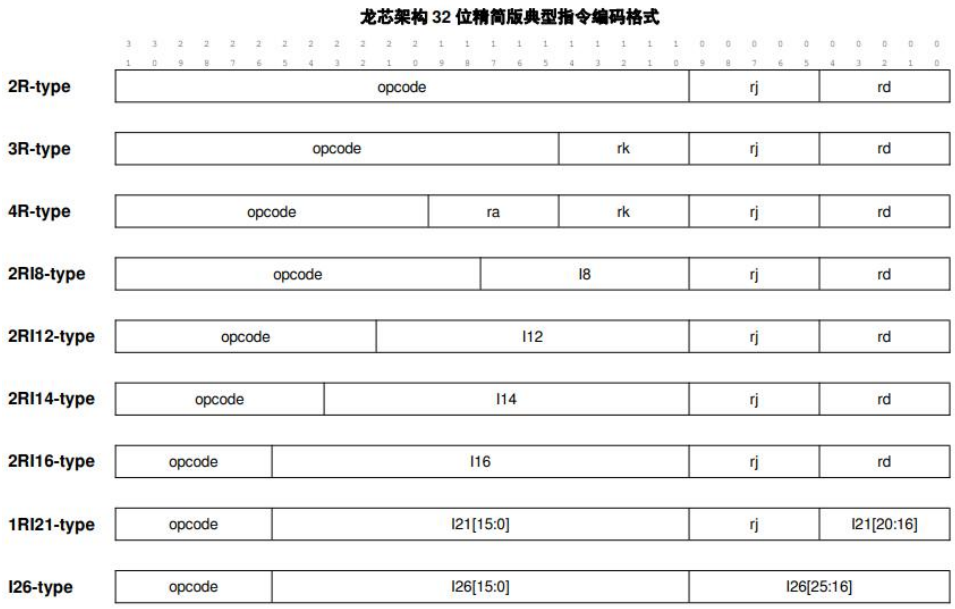



图 18 LA32R 指令编码格式

在 PA 中，我们不需要使用浮点运算指令，所以，只需要实现基本的整数、逻辑运算指令和若干条特权指令即可。

添加方式，见**图 17 模拟 CPU 流程图**,在 decode_exec 函数中，使用框架代码预设的 INSTPAT 宏来定义指令模式。

通过传入模式字符串、指令名称、指令类型和指令执行操作，可以简化指令识别与执行模拟的编写过程。例如：

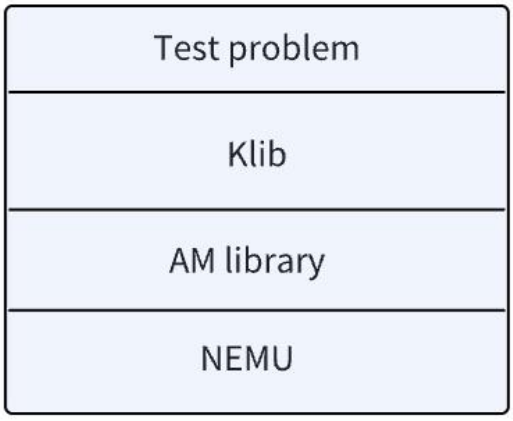
```
INSTPAT("0000000000 0100000????? ????? ?????", add.w , 3R, R(rd) = src1 + src2);
```

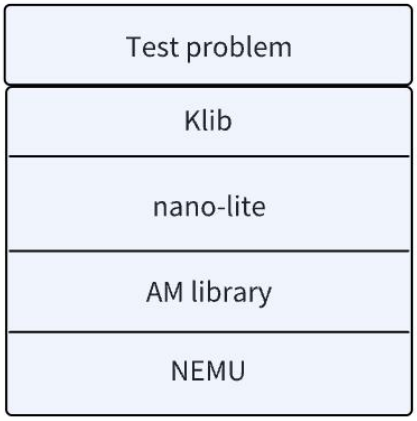
## 3.3 运行时环境构建

运行时环境的构建是为了将机器功能抽象成 C 语言 API，使程序能够屏蔽 ISA（指令集架构）的细节，从而提高程序的可移植性和兼容性。

### 3.3.1 AM 介绍

AM（Abstract Machine，抽象机器）提供了一组接口，这些接口将底层硬件的功能抽象为通用的 API。通过这些 API，应用程序可以在不同的硬件平台上运行，而无需关心底层的硬件实现细节。




a)“裸机”上的层次结构　　b)OS 上的层次结构

图 19 NEMU 上运行层次结构

### 3.3.2 klib 实现

Klib 的实现目标是提供一套基本的、轻量级的库函数，旨在支持在嵌入式系统或特定环境中运行的应用程序。这些库函数包括内存和字符串的写入函数（如 memset()、strcpy()等）、内存和字符串的只读函数（如：memcmp()、strlen()等）以及格式化输出函数（如 sprintf()等）。在实现过程中，需要理解每个函数的行为和它们的输入/输出关系，然后基于这个理解编写相应的代码。

（1）**内存和字符串的写入函数：**如 memset() 是通过使用一个 for 循环，循环 n 次，在每次循环中，将 c 的值（强制转换为 unsigned char 类型）写入到 p 指向的内存区域的第 i 个字节。最后，函数返回 s，即指向被填充的内存区域的指针。而 strcpy() 则是通过一个 while 循环逐个字符地复制源字符串到目标字符串，同时移动源字符串和目标字符串的指针。当遇到源字符串的结尾字符 \0 时，停止复制，保证目标字符串成为一个有效的字符串。

```c
void *memset(void *s, int c, size_t n) {
  unsigned char *p = (unsigned char *)s;
  for(size_t i = 0;i < n;i++){
      p[i] = (unsigned char)c;
  }
  return s;
  // panic("Not implemented");
}
```

图 20 memset 实现

```c
char *strcpy(char *dst, const char *src) {
  char *ret = dst;
  while((*dst++ = *src++)){}
  return ret;
  // panic("Not implemented");
}
```

图 21 strcpy 实现

（2）**内存和字符串的只读函数：**memcmp() 是比较两个内存区域的内容。它接受三个参数：两个指向内存区域的指针和比较的字节数，函数遍历指定的字节数，逐个比较两个内存区域对应位置的字节。如果找到不相等的字节，就

立即返回比较结果，如果比较完毕没有发现差异，则认为两个内存区域内容相同。strlen() 接受一个指向字符串的指针作为参数。函数从字符串的起始位置开始，逐个检查字符直到遇到结尾字符 \0，在遇到 \0 之前的字符数量就是字符串的长度。

```c
void *memcpy(void *out, const void *in, size_t n) {
  unsigned char *d = (unsigned char *)out;
  const unsigned char *s = (const unsigned char *)in;

  for(size_t i = 0;i < n;i++){
    d[i] = s[i];
  }

  return out;
  // panic("Not implemented");
}
```

图 22 memcpy 实现

```c
size_t strlen(const char *s) {
  if (s == NULL) {
    return 0;
  }
  size_t n = 0;
  while(s[n] != '\0') {
    ++n;
  }
  return n;
  //panic("Not implemented");
}
```

图 23 strlen 实现

（3）**格式化输出函数：**sprintf() 函数实现思路如图 26，接受一个输出字符串 out，一个格式字符串 fmt 和一系列可变参数，函数遍历 fmt 中的每个字符，如果字符不是 %，就直接将这个字符复制到 out 中。如果字符是 %，则根据 switch 内的规则来进行匹配，若是符号 d，则说明该 2 变量是一个十进制的数值。

```c
int sprintf1(char *out, const char *fmt, ...) {
  va_list pArgs;
  va_start(pArgs, fmt);
  char *start = out;

  for(;*fmt != '\0';++fmt){
    if(*fmt != '%'){
      *out = *fmt;
      ++out;
    }
    else{
     switch(*(++fmt)){
     case '%': *out = *fmt; ++out; break;
     case 'd': out += itoa1(va_arg(pArgs,int),out,10);break;
     case 's':
        {
        char *s = va_arg(pArgs,char*);
        strcpy(out,s);
        out += strlen(out);
        }
        break;
     }
    }
   }
    *out = '\0';
    va_end(pArgs);
    return out - start;
}
```

图 24 sprintf 实现

### 3.3.3 外设模拟

在没有外设的计算机中，它只是一台无情的计算机器，无法与外界进行任

何交互。因此，为了强化我们对软硬件协同的理解，PA 设置了串口、时钟、键盘、VGA 以及声卡的实现实验。这些实验大致分为两个部分：硬件模拟和 AM 软件驱动。

我们知道，CPU 对于外设的寻址一般有两种方式：端口映射 I/O（Port-Mapped I/O, PMIO）和内存映射 I/O（Memory-Mapped I/O, MMIO）。NEMU 已经实现了对这两种方式的支持。然而，由于端口映射 I/O 存在一些限制和问题，现代处理器主要采用内存映射 I/O 的方式。因此，在 NEMU 的外设模拟实现中，我们主要关注的是内存映射 I/O。

在 PA 提供的 NEMU 框架中，MMIO 的实现方式是先申请一块大小为 2MB 的内存池，各个外设分别从中取出所需的内存用于寄存器（包括数据寄存器、控制寄存器、状态寄存器）注册，然后将这些寄存器添加进映射表中。这种设计确保了外设寄存器的统一管理和高效访问。最后，我们可以得到如下图所示的地址空间关系：


图 25 外设地址空间

由于所有外设的模拟实现都经过了不同程度的简化，串口、时钟、键盘的实现都比较简单，声卡我们团队暂未实现不予讨论。在已实现的外设中，最有挑战性的是 VGA 的实现。若是对软硬件协同的过程理解的不到位，很容易出现包括但不限于如下图 x 和图 y 中的“惨案”。

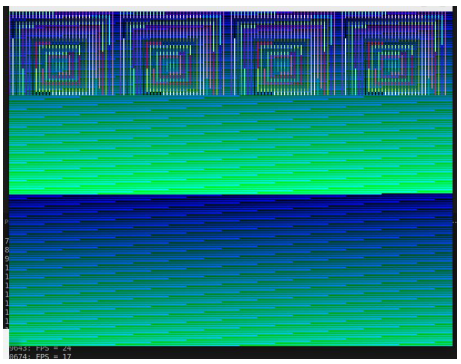


图 26 VGA 显示重叠

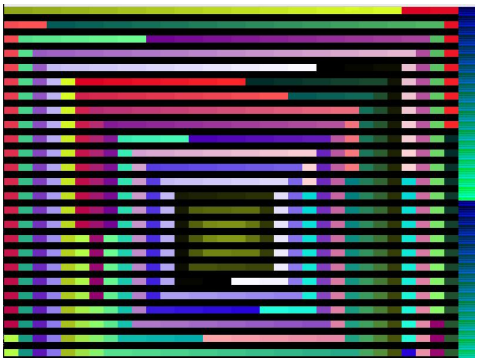


图 27 VGA 显示断层

### 3.3.4 异常处理支持

#### 1.Loongarch 特权级架构

LoongArch 指令集架构（ISA）包含 64 位版（LA64）、标准 32 位版（LA32）和精简 32 位版（LA32R）。LA64 和 LA32 定义了四个特权级（PLV0~PLV3）。其中，PLV0 是最高特权级，主要用于内核操作，而 PLV3 是最低特权级，主要用于运行应用程序。而作为教育和研究用途的 LA32R 则只有 PLV0 和 PLV3。 这些特权级的设计旨在提供灵活的权限管理机制，以支持不同的系统任务和应用需求。

#### 2.异常处理机制

异常处理机制（LA 这里称为例外）是计算机系统用于应对和处理运行过程中出现的异常情况的一种技术。它可以是由于硬件故障、软件错误或其他无法预期的情况引起的。旨在确保系统在遇到异常时能够妥善处理，从而避免系统崩溃，并尽可能恢复正常运行。

其中，操作系统中最重要的上下文切换，就是通过异常处理机制实现的。在 LA32r 中，上下文切换涉及到如图 27 中的几个 CSR：

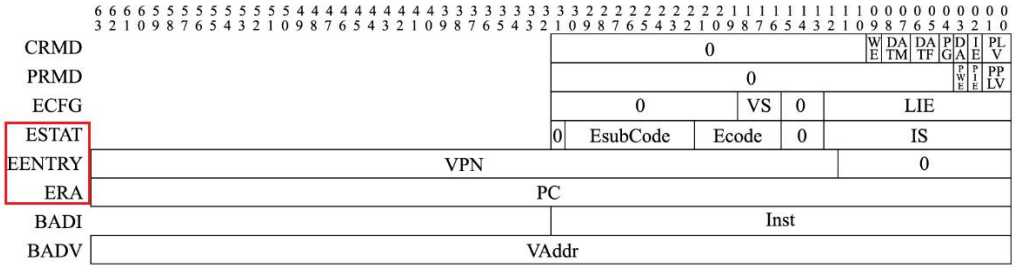



图 28 LA32R 异常处理结构图

但由于，PA 是一个简化的系统，所以不我们仅仅需要关注 ERA、EENTRY、ESTAT 三个寄存器即可。

- **EENTRY**：是存放中断异常入口的 CSR，它在手册中的描述如图 x，通常来说，每个系统初始化的时候它只会被设置一次。

| 位 | 名字 | 读写 | 描述 |
|---|---|---|---|
| 11:0 | 0 | R | 只读恒为 0，写被忽略 |
| GRLEN-1:12 | VPN | RW | 普通例外和中断入口地址所在页的页号 |

表 2 EENTRY 寄存器

- **ERA**：用于存放触发中断异常的 PC 值。
- **ESTAT**：用于存放 CPU 当前的异常中断状态和异常号。在目前的进度中，仅需要关注 Ecode 字段即可。

| 位 | 名字 | 读写 | 描述 |
|---|---|---|---|
| 1:0 | IS[1:0] | RW | 两个软件中断的状态位。比特 0 和 1 分别对应 SWI0 和 SWI1。软件中断的设置也是通过这两位完成，软件写 1 置中断写 0 清中断。 |
| 12:2 | IS[12:2] | R | 中断状态位。其值为 1 表示对应的中断置起。1 个核间中断(IPI)，1 个定时器中断(TI)，1 个性能计数器溢出中断(PMI)，8 个硬中断(HWI0~HWI7 )在线中断模式下,硬件仅是逐拍采样各个中断源并将其状态记录与此。此时对于所有中断须为电平中断的要求，是由中断源负责保证，并不在此处维护。 |
| 15:13 | 0 | RO | 保留域。读返回 0，且软件不允许改变其值。 |

| | | |
|---|---|---|
| 21:16 | R | 例外类型一级编码。触发例外时：<br>如果是 TB 重填例外或机器错误例外，该域保持不变；硬件会根据例外类型将表 4 中 Ecode 栏定义的数值写入该域。 |
| 30:22 | R | 例外类型二级编码。触发例外时：<br>如果是 TB 重填例外或机器错误例外，该域保持不变；否则，硬件会根据例外类型将表 4 中 EsubCode 栏定义的数值写入该域。 |
| 31 | RO | 保留域。读返回 0，且软件不允许改变其值。 |

表 3 ERA 寄存器

| Ecode | EsubCode | 例外代号 | 例外类型 |
|---|---|---|---|
| 0x0 | | INT | 仅 CSR.ECFG.VS=0 时，表示是中断 |
| 0x1 | | PIL | load 操作页无效例外 |
| 0x2 | | PIS | store 操作页无效例外 |
| 0x3 | | PIF | 取指操作页无效例外 |
| 0x4 | | PME | 页修改例外 |
| 0x5 | | PNR | 页不可读例外 |
| 0x6 | | PNX | 页不可执行例外 |
| 0x7 | | PPI | 页特权等级不合规例外 |
| 0x8 | 0 | ADEF | 取指地址错例外 |
| | 1 | ADEM | 访存指令地址错例外 |
| 0x9 | | ALE | 地址非对齐例外 |
| 0xA | | BCE | 边界检查错例外 |
| 0xB | | SYS | 系统调用例外 |
| 0xC | | BRK | 断点例外 |

表 4 例外编码表

值得注意的是，PA 实验可能为了让学生逐步理解异常处理的软硬件协同过程，当执行 SYSCALL 指令时，它有两个异常事件指向：EVENT_SYSCALL 和 EVENT_YIELD。经查手册发现，系统调用号中也有 yield 这个系统调用。 而实

际上，EVENT_YIELD 在这里和后续似乎都没有什么其他作用。所以，当执行 SYSCALL 指令时异常号直接设置为 0xB 即可。

```
// An event of type @event, caused by @cause of pointer @ref
typedef struct {
  enum {
    EVENT_NULL = 0,
    EVENT_YIELD, EVENT_SYSCALL, EVENT_PAGEFAULT, EVENT_ERROR,
    EVENT_IRQ_TIMER, EVENT_IRQ_IODEV,
  } event;
  uintptr_t cause, ref;
  const char *msg;
} Event;
```

图 29 异常处理 Event 结构体

最后，在 Loongarch 的手册中可以发现 syscall 指令后面是跟了一个 code 字段。经与技术支持群的老师们确认，当前该值为未定义行为，可以不用理会。

**指令格式：** syscall code

执行 SYSCALL 指令将立即无条件的触发系统调用例外。

指令码中 code 域携带的信息可供例外处理例程作为所传递的参数使用。

图 30 sycall 指令格式

## 3.4 LA32r 适配

除了完成 PA 官方讲义中的内容外，针对 LA32r 架构还进行了以下适配工作：

### 3.4.1 Diff-QEMU 适配

经调研，在选择 la-NEMU 的 参考模型（Ref）时，一共有过三个备选方案。分别是：

- QEMU：在 PA 框架原有的接口上实现 QEMU 对 LA32r 的 Ref。
- CEMU：用重庆邮电大学 陈泱宇同学开源的 CEMU ，该模拟器支持 LA32r，并能启动 Linux，且留有作为 Ref 的接口。
- LA32R-NEMU：某位打龙芯杯的前辈直接在 NEMU 上魔改的开源项目，初衷是给打龙芯杯的同学作为 Ref 使用。

最后决定直接在 PA 框架原有的接口上，对 QEMU 的接口进行是适配。原因在于我们项目的目标本来就是尽力降低其他同学选择 LA32r 完成实验的门槛，

无故引入新的子项目有违初衷。经过源码分析后发现，余子濠老师在设计 PA 时对各个体系结构的兼容性做得都太好了，适配 LA32r 其实需要添加改动的地方并不多，目前该部分工作已提交 PR 至 PA 的 NEMU 主线，并被接收，如图 30。


图 31 邮箱提交 PR 至 PA 的 NEMU 主线

### 3.4.2 la32r 反汇编器

反汇编器的实现需求在于我们团队使用 itrace 时，发现每条指令只有指令的地址和机器码，没有显示对应的汇编代码（如图 31），这使得 itrace 在调试过程中的作用大大的削减了。

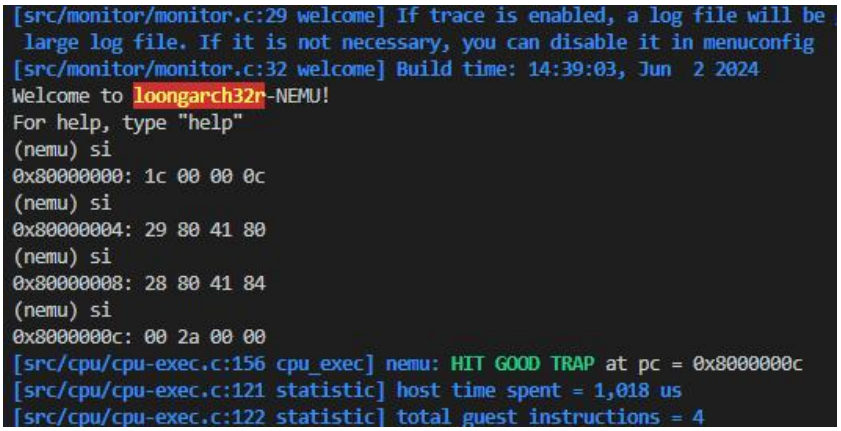



图 32 无反汇编器下的 nemu

在研究源码后发现，PA 框架代码在实现 itrace 的过程中，是使用开源编译框架 LLVM 对其他架构的机器码进行反汇编操作。而对于 Loongarch 架构，LLVM 目前仅支持 LA64，所以 LA32r 的 itrace 无法显示汇编代码。故萌生自己手搓一个的想法。图 32 位实现效果。

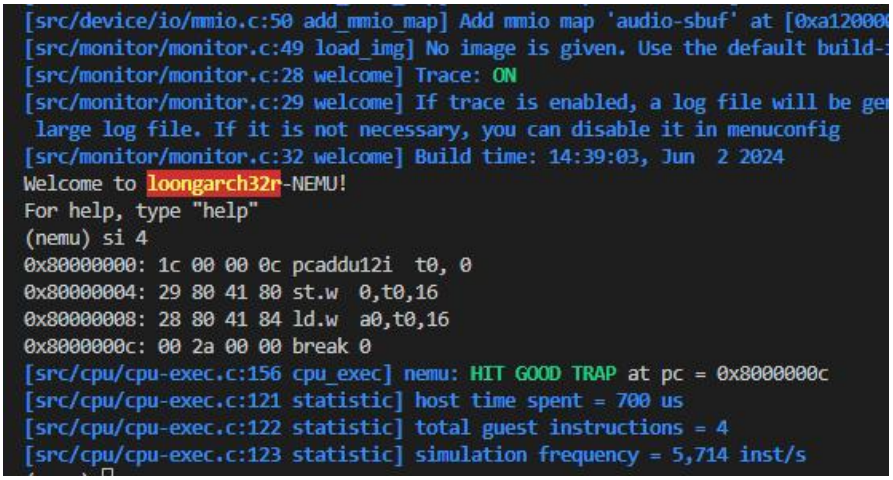



图 33 添加反汇编器后的 nemu

最后，尝试把此部分工作也推到 PA 的 NEMU 主线中，但似乎余子濠老师他们有其他考虑，故作罢。


图 34 余子濠老师回应 PR

### 3.4.3 LA32r-QEMU Bug 发现与修复

在 NEMU 实现指令，并用 QEMU 验证指令实现正确性的过程中，我们发现 LA32r-QEMU 的 mul.h 指令存在 Bug。通过细致的调试和分析，我们成功地定位了这些问题，并撰写了详细的分析报告，反馈给龙芯的老师们，目前该 bug 已修复，如图 35。详细分析过程链接：la32-QEMUbug 分析报告


图 35 反馈 LA32r-QEMU Bug 至龙芯并收到回应

## 3.5 运行演示

### 3.5.1 超级马里奥运行

通过完成外设(时钟，VGA，键盘)的支持，使得能在 loongar32r-nemu 上运行超级马里奥。

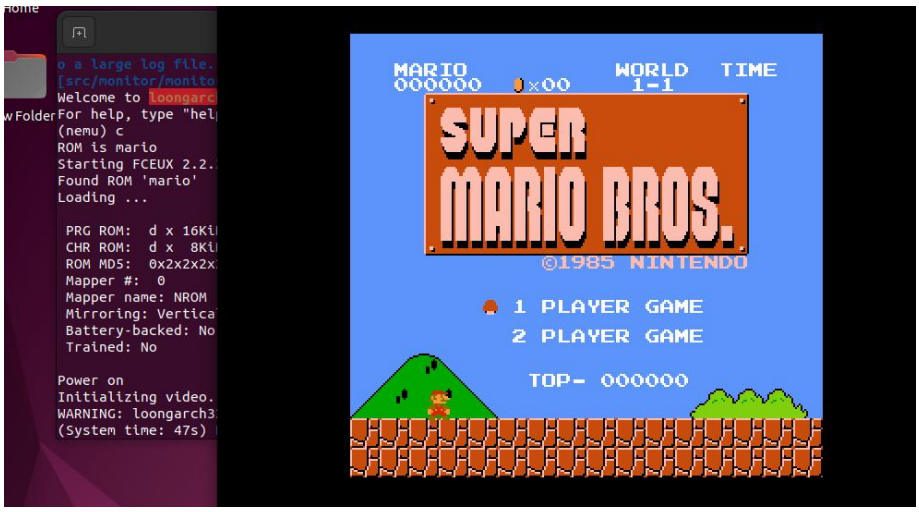



图 36 成功运行超级马里奥

### 3.5.2 nanos-lite 运行

通过完成系统调用(exit)和异常处理机制支持，nanos-lite 能处理这部分

异常和系统调用。

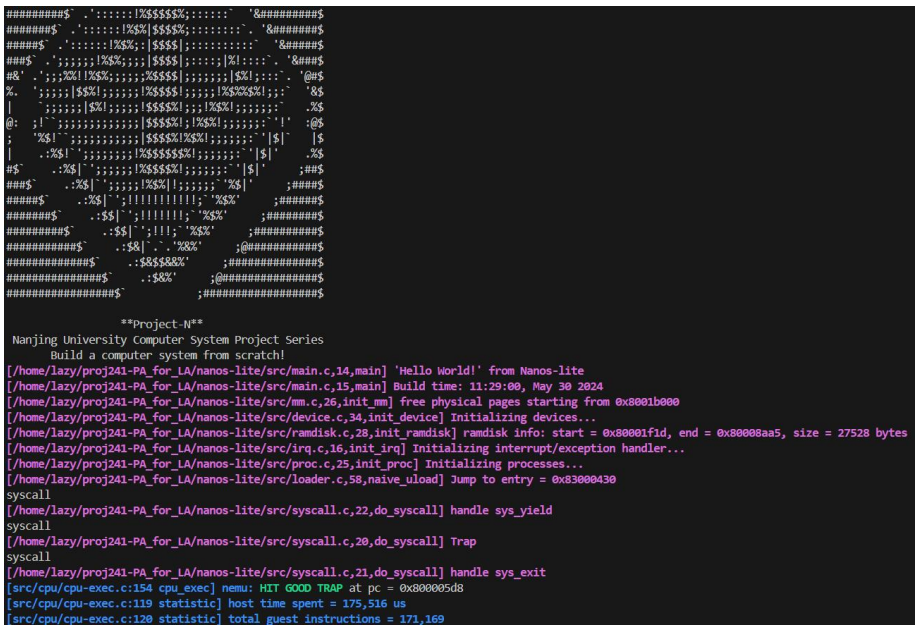



图 37 nanos-lite 运行界面

# 第四章 回顾与展望

在过去的一个月中，团队成员熟悉了 NEMU 代码框架、LoongArch 架构文档、Git 以及飞书等项目工具。截止初赛结束，本团队的工作进展如下：

- 简易调试器（sdb）：构建了一个具备单步调试、内存扫描、表达式求值以及设置监视点功能的简易调试器。
- Bug 检测工具：开发了一套包括 iringbuf、mtrace 和 ftrace 的 Bug 检测工具，为后续指令集添加提供了调试功能。
- 基础指令集和标准库：完成了 LoongArch 基础指令集的添加和简易标准库的实现，使测试程序能在 NEMU（硬件模拟器）上运行。
- 外设支持：完成了对外设（包括串口、时钟、键盘和 VGA）的支持，使 NEMU 不再是一台“无情”的计算机器，能够与外部世界进行交互。
- 简易操作系统：在简易操作系统基础架构上，实现了异常响应机制、用户程序加载以及系统调用（目前只能处理 exit）。

## 4.1 团队开发思路

团队首先学习了 Git 命令，实现了本地文件与远程仓库的同步协作。随后，通过阅读底层源码和南京大学 PA 实验指导书，熟悉了项目结构，为后续推动项目开发奠定基础。接着，阅读了 LoongArch 架构文档，熟悉了基础指令

集，为后续添加指令集做好了准备。最后，通过研究有关 LoongArch 异常处理机制和系统调用的文档，为实现异常响应机制和系统调用做好了准备。

## 4.2 项目推进难点

PA 项目除了内核源码较为复杂，还有更多工具需要学习。例如：编译工具链的适配和使用、Difftest-qemu 适配等。与 x86、RISC-V 相对比，LoongArch 架构的技术文档和撰写的技术博客较少，可供参考的示例程序不多。

## 4.3 项目总结

在项目的最初一个月里，我们团队积累了一系列关键技能和成果。我们提高了对关键开发工具和 NEMU 代码框架的熟悉度，特别是 Git 和飞书，从而提升了团队协作和代码管理的效率。团队开发的 sdb 简易调试器和一套 Bug 检测工具，增强了团队的调试能力，使得团队能快速迭代和优化 LoongArch 指令集的实现。这一进展助力于测试程序能够在硬件模拟器上稳定运行。外设支持的实现不仅展示了技术上的实用性，也使得团队能够在模拟环境中更好地演示和验证功能。对简易操作系统基础架构的开发，包括异常响应机制、上下文管理和系统调用的实现，则加深了团队对操作系统内部工作机制的理解，并提升了系统级编程的实战经验。

团队在项目上取得的进展不仅稳固了我们在 LoongArch 平台的技术基础，同时也为后续阶段的深入研究和开发工作提供了坚实的出发点。

## 4.3 决赛阶段计划

完成 PA4 及相关文档，完成真机适配及性能调优，进一步完善项目及相关文档内容。撰写详细的实验手册，填补 PA 实验讲义对于 LoongArch 架构相关的介绍空白，让往后的同学做 PA 实验选择 LA32r 作为目标架构时，上手难度能与 MIPS、RISC-V 持平。

# 致谢

2024 年 4 月 20 日,团队成员的 loongarch 体系架构的经验可谓是“一穷二白”，截至 5 月末，团队已经熟悉 Git 工具，loongArch 的编译工具链，LoongArch 架构手册，qemu 模拟器等知识，并通过软件实现 loongArch 架构硬件模拟(nemu)支持测试程序的运行。

团队的进步离不开队长(李宗逸)的正向引导和严格要求，以及队员(李智锋，刘奕钊)之间的相互支持和鼓励。团队每周开 2 次组会，讨论技术，积累文档，探索方向，队员分工明确，配合默契，所谓精诚合作，金石为开。

团队的进步更离不开龙芯技术支持群的老师们的辛勤付出。李亚伟老师为团队提出的问题提供正确的解决思路。

团队还要感谢谢文兰老师(指导老师)的支持和帮助。

谨以此篇，向本项目的指导老师，团队成员和其他参与者表达感谢。

# 参考资料

[1] 南京大学 计算机科学与技术系 计算机系统基础 课程实验(PA)讲义 https://nju-projectn.github.io/ics-pa-gitbook

[2] 计算机体系结构基础（第 3 版）

[3] 龙芯架构参考手册卷一

[4] 龙架构 32 位精简版快速入门指南

[5] rCoreloonArch-tutorial https://godones.github.io/rCoreloongArch/sup.html

[6] 2022 年 OS 大赛开源作品：la-sel4 https://github.com/tyyteam/la-seL4

[7] 2022 年 OS 大赛开源作品：rCore 的龙芯平台移植 https://github.com/Godones/rCoreloongArch

[8] la32r 工具链 https://gitee.com/loongson-edu/la32r-toolchains

[9] la32r gdb 安装 https://blog.csdn.net/greenmoss/article/details/127800221

[10] 基于 QMP 实现对 qemu 虚拟机进行交互 https://zhuanlan.zhihu.com/p/56887210

[11] 一生一芯双周分享会(nemu 部分)：视频回放 https://space.bilibili.com/238318574?spm_id_from=333.788.0.0

[12] 一生一芯双周分享会(nemu 部分)：相关主题文档 https://docs.qq.com/sheet/DU05xUmxjWmFvaXhj?tab=8nd1jt&login_t=1715061395395