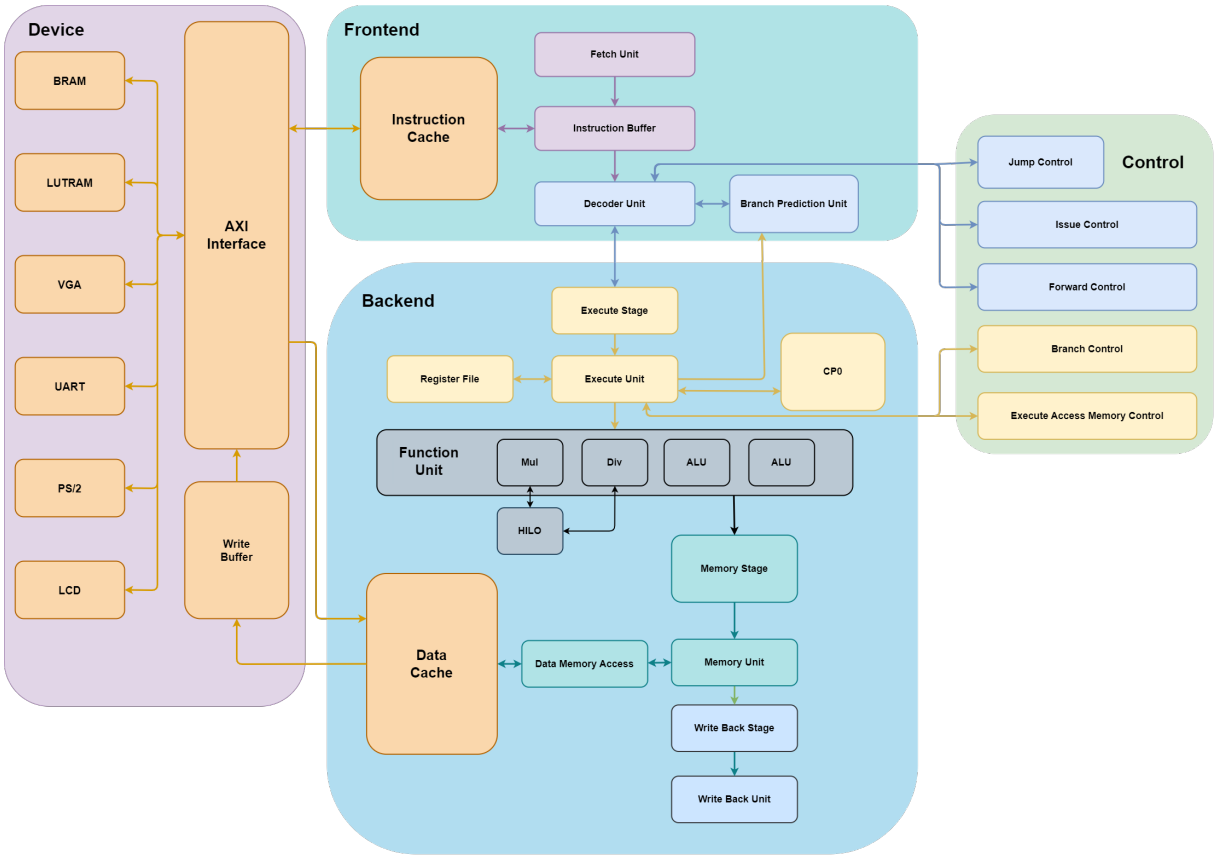
# NSCSCC2023

# 杭州电子科技大学 PUA-MIPS 队

# 决赛设计报告




叶剑豪 奚力丰 胡致尧

# 目录

# 1 概述

我们基于Chisel语言设计并实现了一个MIPS32指令集架构的处理器:PUA-MIPS(Powerful Ultra Architecture MIPS)处理器。能够使用龙芯FPGA实验平台上的大部分外设，并且成功通过了龙芯杯官方框架提供的功能测试、性能测试和系统测试。该处理器可以运行PMON，U-Boot引导程序，uCore操作系统和Linux操作系统。

## 1.1 处理器亮点

- 顺序双发射六级流水线处理器PUA-MIPS(Powerful Ultra Architecture MIPS)。
- 处理器参数
  - IPC几何平均值 1.1
  - 极限频率 90M
  - 顺序执行双发射
  - 七级动态流水（2取指，1解码，1~4执行，2访存，1写回）
  - 55%双发率
  - 8KB大容量Cache
  - 2bit分支预测
- 处理器设计
  - 指令集：MIPS32 Release1的除BranchLikely和浮点指令以外的所有指令。
  - 流水线结构：除法2-4级流水，两级访存。非对称双发射分为Path0和Path1，在遇到一些特殊指令时Path1暂停，Path0继续执行。
  - 异常处理：实现MIPS32 Release1中提到的所有异常和中断机制。
  - CP0：实现MIPS32 Release1中提到的所有CP0寄存器行为。
  - 缓存
    - ICache为两路组相连8KB设计，支持四指令读取窗口，状态机直接控制AXI交互。
    - DCache为两路组相连8KB设计，WriteBuffer使得写入不再制约处理器性能。
- Soc
  - CPU：PUA-MIPS CPU（含Cache）
  - DRAM支持：使用板载 DDR3 SDRAM作为主存
  - 串口：实现串口控制器以调整波特率
  - 以太网：使用IP核构建以太网控制器
  - GPIO：使用confreg组件控制LED，数码管等组件
  - 图像输出：实现VGA传输协议
- OS
  - 支持PMON，UBoot引导程序，支持uCore操作系统。
  - 支持最新Linux主线v6.5-rc3版本。
  - 支持OS驱动所有外设。
- Chisel语言
  - 参数化设计，支持流水线长度、取指宽度、访存宽度等参数的配置。
  - KISS设计理念加持，单个文件大小不超过200行。
  - 面向对象带来更高的自由度，同一种算法，四种不同实现随意组合。

### 1.2 参考资料

- NSCSCC往届处理器设计
  - nontrivial-mips 给了我们如何启动Linux操作系统的良好示范。
  - TrivialMIPS 中有许多顺序处理器设计的技巧。
  - amadeus-mips 帮助我们更好的使用Chisel语言。
  - UltraMIPS 给了我们Cache设计的良好示范。
  - CDIM 为我们提供了高效的测试框架。
  - GenshinCPU 有许多频率优化的设计方法。
  - zenCove 在高级语言的使用上启发了我们。
- 开源处理器
  - 香山处理器 给了我们很多架构设计上的启发。
  - 果壳处理器 一个优秀处理器的示范。
  - dinocpu 一个以Chisel语言为基础开设的处理器设计课程，使用基于Scala的交互式调试框架给了我们很多帮助。
- 书籍
  - 计算机组成与设计：硬件/软件接口
  - 计算机体系结构：量化研究方法(第6版)
  - CPU设计实战
  - 超标量处理器设计
  - 自己动手写CPU

## 2 处理器详细设计

PUA-MIPS是顺序双发射结构设计，支持MIPS32 Release1扩展，PUA-MIPS处理器前端流水级包括取指单元、指令队列等单元，顺序取指。后端包括译码、分支预测、运算单元、寄存器堆、访存控制和写回等单元。顶端有一个控制单元，负责控制各个流水级之间的同步和冲突。缓存分为ICache、DCache、TLB等模块。

### 2.1 前端

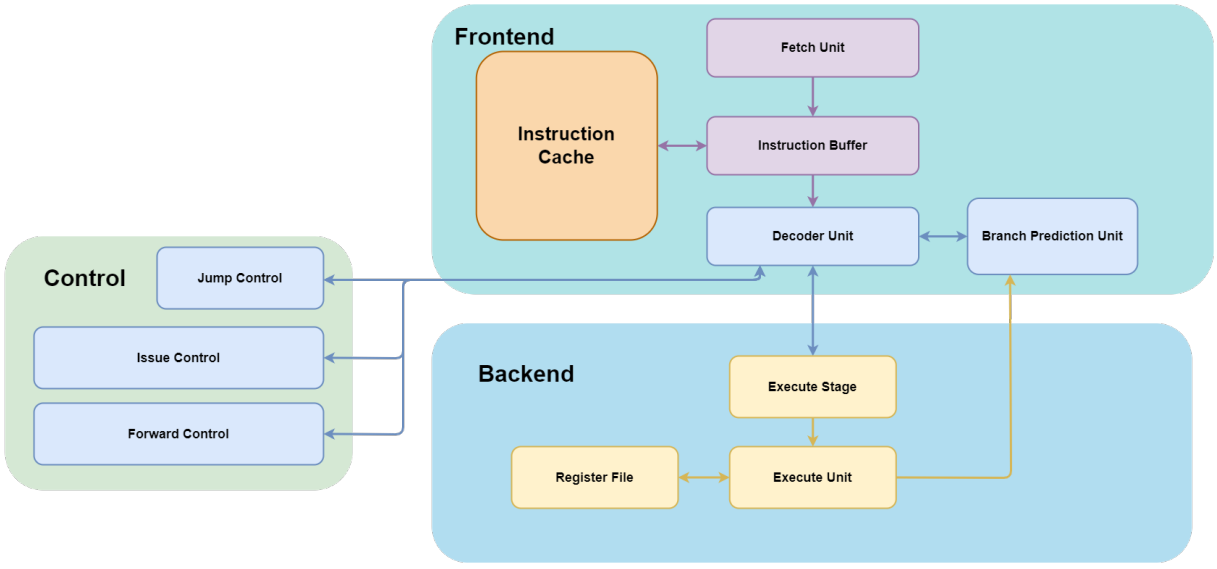



图 2: Frontend Design

### 2.1.1 取指（Fetch）

前端流水级包括取指单元、指令缓存等单元。取指单元（FetchUnit）根据后端的信息决定是否跳转并送给指令缓存（Instruction Cache）接下来四条指令的地址。我们通过取指队列（Instruction Fifo）实现前端（Frontend）和后端（Backend）的充分解耦，使得开发过程能够尽量保持独立。指令队列是一个深度为16的先进先出队列，它负责从取指缓存中获得指令并协调和译码以及后端之间的联系。

### 2.1.2 译码（Decoder）

译码阶段中进行双路译码。从取指队列中取出两条指令，根据指令产生控制信号。同时为了减少分支跳转指令产生的对流水线刷新的影响，我们添加了静态分支预测单元，使用传统2bit分支预测器对PC进行预测。译码会得到指令在执行阶段的阻塞情况，这部分信息会反馈给取指队列以控制取指队列的出队。随后，通过四端口读两端口写寄存器堆，读出寄存器对应的值并发送给执行阶段。

### 2.1.3 分支预测（Branch Predictor）

我们实现了多种分支预测器，包括局部历史分支预测和全局分支预测器以及卷积神经网络分支预测器（因为过拟合严重而被放弃）。在除了卷积神经网络的分支预测方法中，局部分支预测器取得的预测效果最好，因此我们最终使用了局部分支预测器。

| 策略 | 几何平均准确度 |
| --- | --- |
| 卷积神经网络分支预测 | 96.58% |
| 局部历史分支预测 | 94.48% |
| 全局分支历史预测 | 75.86% |

同时，我们对局部历史分支预测进行了设计空间探索。由于局部历史分支预测受到分支历史寄存器查找表（Branch History Register Table，BHT）和模式历史查找表（Pattern History Table，PHT）的深度的影响。因此我们对每一个分支历史寄存器表的深度和模式历史表的深度计算出分支预测的失败率如下图：

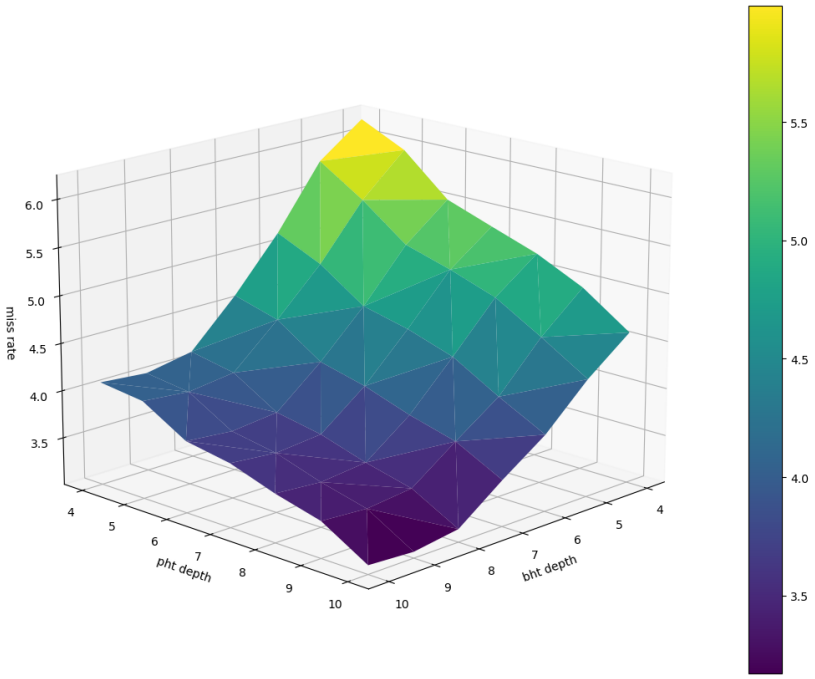



图 3: BPU fail rate

可以看到，在4-8的范围内优化BHT的深度更具性价比。最终出于频率的考虑，我们将BHT的深度设置为6，PHT的深度设置为4。

## 2.2 后端

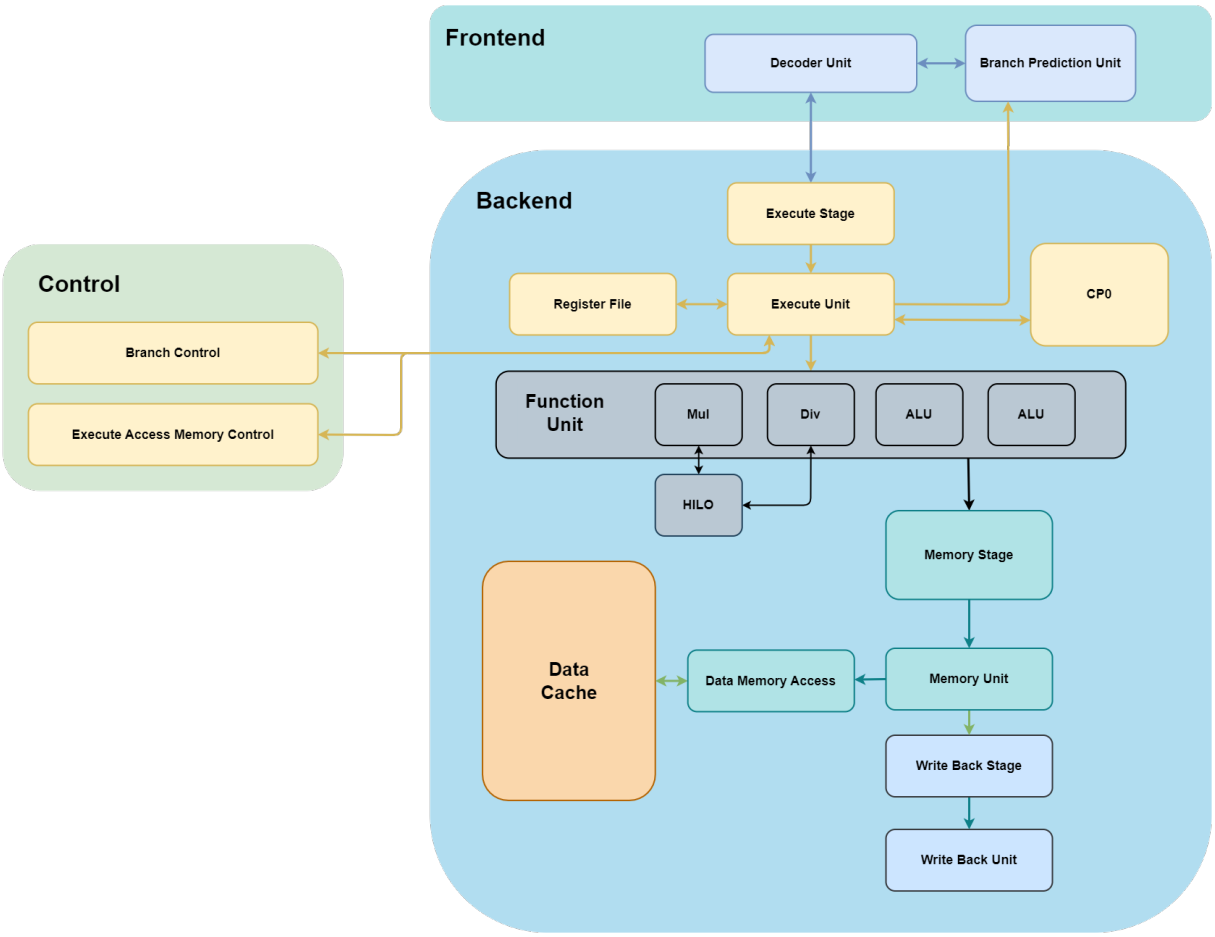



图 4: Backend Design

### 2.2.1 执行（Execute）

执行阶段中对根据译码阶段的信号对数据进行运算。

- 对于普通的运算指令，我们使用两个ALU运算单元来实现双发射极限情况下的普通运算，对于特殊运算，我们使用不同的运算单元。
- 针对乘除法指令，我们使用Xilinx ip核进行运算，其中乘法两周期，除法四周期。通过握手信号与执行单元进行通信，同时在功能单元（Function Unit）中对两条路径上的运算进行仲裁。乘除法运算完成后立即写回HILO寄存器。
- 针对分支指令，我们需要在该阶段判断分支指令是否成功，并将判断的结果返回分支预测单元。
- 针对访存指令，由于访存需要两个周期，因此我们在执行级同时需要向数据缓存（Data Cache）发送访存信号。
- 我们还需要进行TLB的第一级地址转换来减少TLB的时延以及将异常信号发往CP0模块进行集中处理。

### 2.2.2 访存（Memory1，Memory2）

访存1将完整的信息传递给数据缓存，以避免数据缓存将错误的地址传输给AXI总线。 访存2获得数据，并将数据分配给Path0以及Path1。

### 2.2.3 写回（Write Back）

写回阶段进行双发射写回，若遇到了读写冲突则进行前递操作。

## 2.3 缓存（Cache）

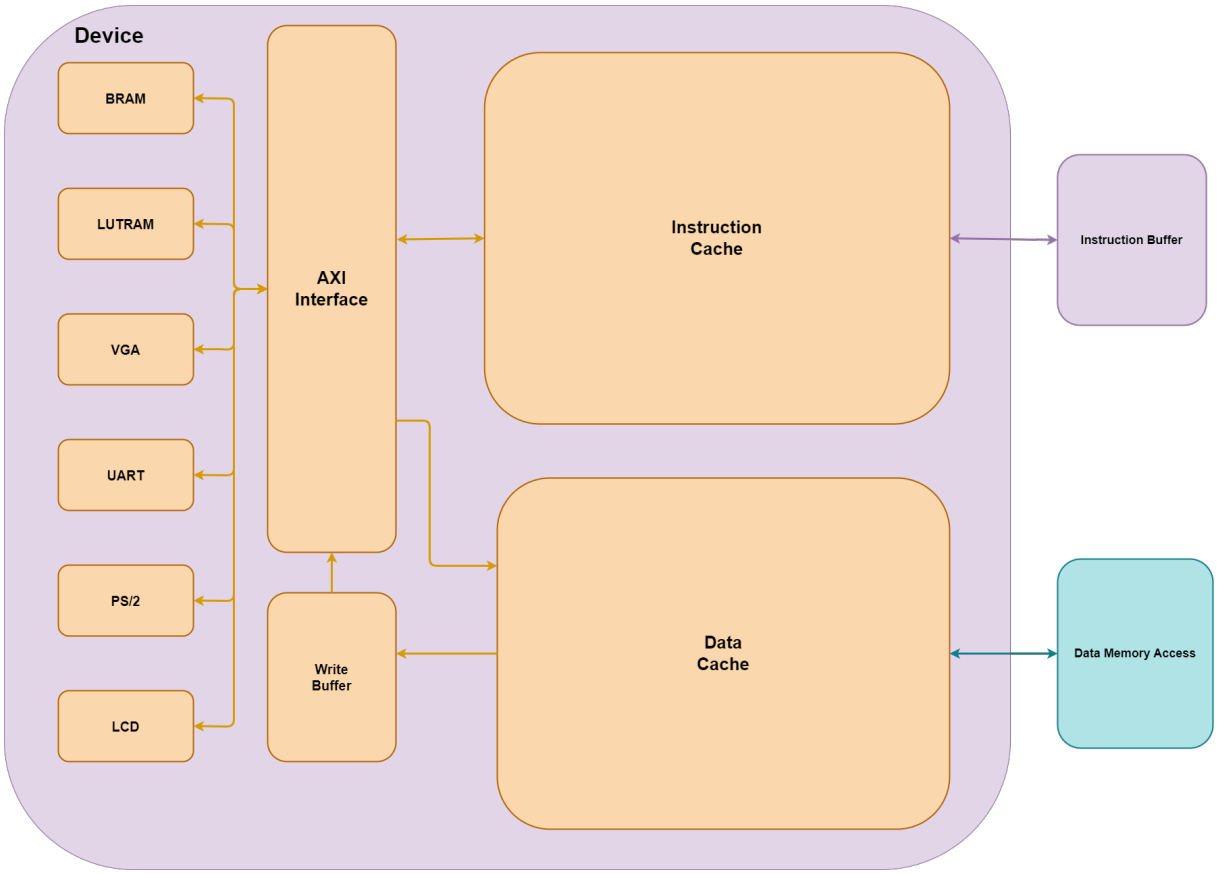



图 5: Cache Design

在PUA-MIPS中，设计实现了指令缓存和数据缓存。

- PLRU替换策略
- VIPT
- AXI突发传输
  - 十六字突发
- 两级TLB平衡延迟
- 两路8KB大缓存
- 高度自由可配置项
- 指令缓存
  - 四宽度取指
  - 单行八条指令
- 数据缓存
  - 写队列
  - victim Cache

我们的缓存设计受UltraMIPS和CDIM的启发，在此表示感谢。

我们通过性能计数器对Cache的命中率做了统计：

| 测试集 | ICache hit | DCache hit |
|---|---|---|
| bitcount | 99.83% | 96.91% |
| bubble sort | 99.97% | 99.79% |

| | | |
|---|---|---|
| coremark | 99.36% | 95.59% |
| crc32 | 99.98% | 97.85% |
| dhrystone | 99.89% | 81.88% |
| quick sort | 99.96% | 97.85% |
| select sort | 99.98% | 99.33% |
| sha | 99.94% | 98.89% |
| stream copy | 99.65% | 93.37% |
| stringsearch | 99.96% | 90.52% |

可以看出，8KB的ICache可以做到将测试集中的所有指令存入Cache中，因此命中率基本都在99.9%左右。然而，DCache中尽管大部分数据集表现良好，但是对于dhrystone测试集中大量密集的访存表现较差。

#### 2.3.1 指令缓存（Instruction Cache）

指令缓存提供了取指支持，大小和相连度可配置，我们使用状态机来控制缓存。

- IDLE：Cache空闲状态。地址L1TLB查询或直接翻译。若L1TLB发生缺失则暂停，等待L2TLB并处理有可能遇到的异常。
- UNCACHE：向AXI发送读请求或写请求（无突发传输），并等待AXI访存结束。
- CACHE REPLACE：cache缺失后，若没有脏位，需要重新向AXI发送读请求（突发传输），访存并替换对应路的缓存行。其中替换策略采用PLRU。
- SAVE RESULT：当Cache因为TLB缺失或Uncached导致Cache暂时无法正常访问数据时的保留状态。

#### 2.3.2 数据缓存（Data Cache）

数据缓存相比指令缓存的状态机多了一级。

- CACHE WRITEBACK：当需要清除或替换缓存行时，若存在脏位，则需要将脏数据写回，即向AXI发送写请求（突发传输）。

##### 2.3.2.1 Victim Cache

我们通过统计访存指令发现测试集中经常出现一个程序频繁地使用3个数据，而且恰好处在同一个Cache Set中，这导致一个way中的数据经常被踢出Cache后又马上读入，发生了“抖动”。Victim Cache可以将被踢出Cache的数据暂时保存起来，这一操作极大的减少了Cache的缺失率。

##### 2.3.2.2 写队列（Write Fifo）

在我们的设计中，对于Uncached的AXI写请求，我们设置了写请求队列，将AXI写请求数据缓存下来，再逐个写回，该期间如果没有Cached的访问AXI请求，则流水线不会因为写回数据而阻塞，提高了流水线的执行效率。

### 2.4 内存管理

CPU使用内存管理单元(MMU)以及相应的地址映射关系来进行虚地址到实地址的转换。针对TLB的设计，我们参考了CDIM和GenshinCPU的二级TLB设计。这一设计使得我们在使用8项二级TLB的情况下仍能保持较高频率。

## 3 系统外设

外设方面,我们通过Xilinx的Block Design工具构建了整套SOC框架。这套SOC框架通过AXI总线与处理器核进行交互。外设包括:

- DRAM支持：使用板载 DDR3 SDRAM作为主存
- 串口：实现串口控制器以调整波特率
- 以太网：使用IP核构建以太网控制器
- GPIO：使用confreg组件控制LED，数码管等组件
- NT35510：使用ip核实现NT35510驱动LCD显示屏
- 图像输出：实现VGA传输协议

地址映射如下:

| 设备 | 起始地址 | 终止地址 | 大小 |
|---|---|---|---|
| DDR3 | 0×00000000 | 0×07FFFFFF | 128MB |
| CONFREG | 0×1FAF0000 | 0×1FAFFFFF | 64KB |
| Interrupt | 0×1FB00000 | 0×1FB0FFFF | 64KB |
| BootROM | 0×1FC00000 | 0×1FC3FFFF | 256KB |
| NT35510 | 0×1FD00000 | 0×1FD0FFFF | 64KB |
| VGA | 0×1FD20000 | 0×1FD2FFFF | 64KB |
| UART | 0×1FE40000 | 0×1FE4FFFF | 64KB |
| Ethernet | 0×1FF00000 | 0×1FFFFFFF | 1MB |

当前PUA MIPS可以启动uboot,ucore和最新Linux 6.5.0-rc3。Linux中去除了Branch-likely指令。我们通过串口与Linux进行交互。

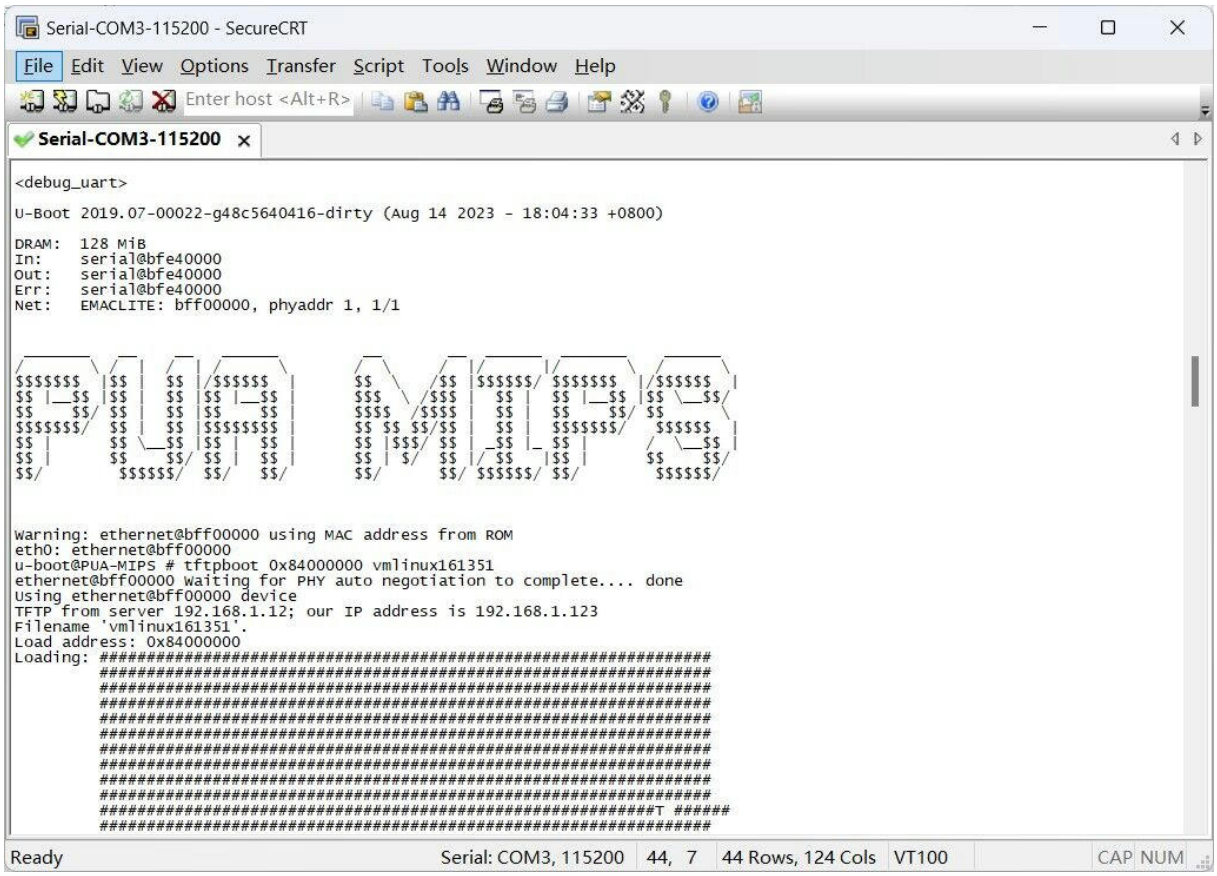



图 6： U-BOOT

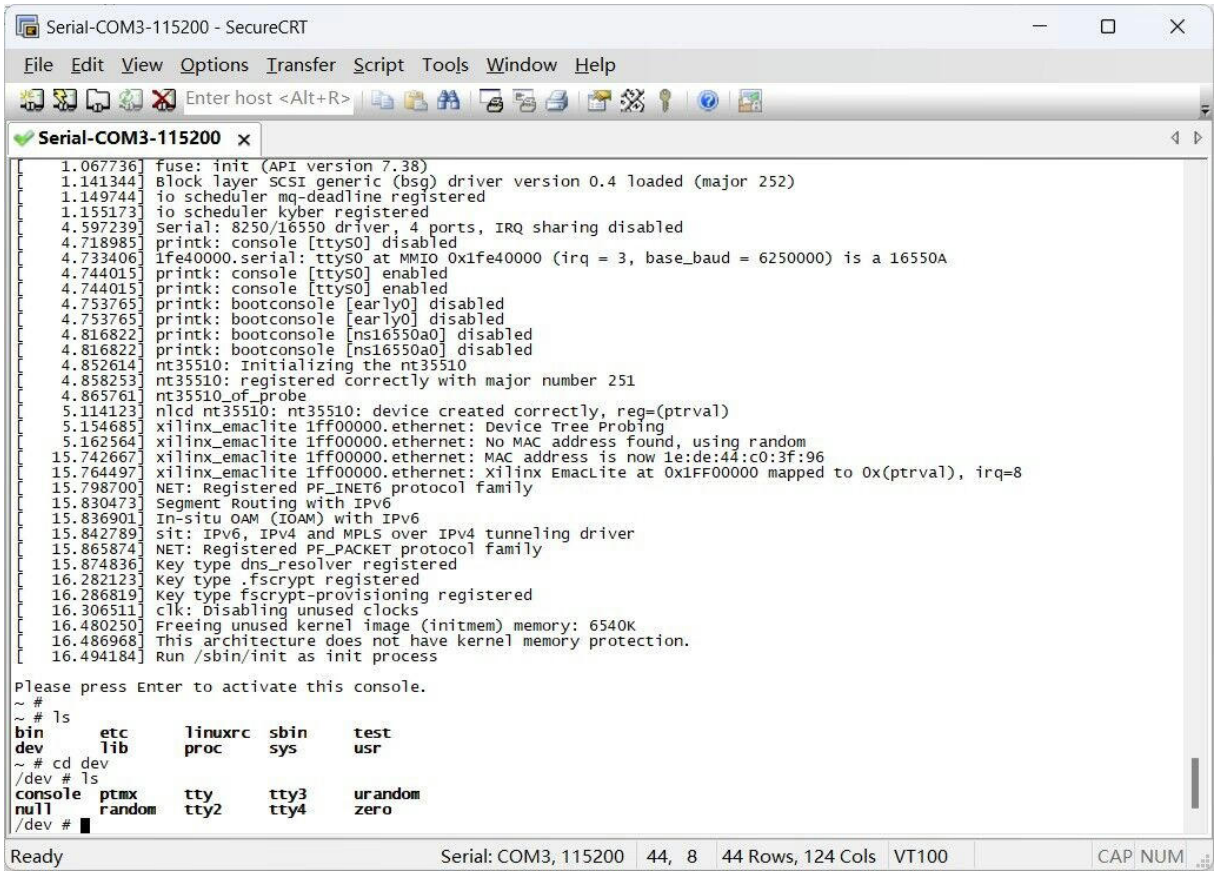



图 7： LINUX

# 4 模拟器设计

我们从NEMU（NJU Emulator）和CEMU（CQU Emulator）中获得启发，设计了我们自己的模拟器HEMU（HDU Emulator）。该模拟器基于Rust语言的强大生态，兼有NEMU的优雅设计模式和CEMU对比赛测试的强大支持，

基于现有的多种模拟器,我们开发了HDU BOARD作为一种抽象的主板。这个主板可以集成任意的SOC，并且能够像真实的主板那样插拔不同的处理器。我们在主板上实现了差分测试框架，可以做到对指令运行状态的ISA级仿真，支持QEMU、NEMU、HEMU、CDMU等多种处理器作为差分测试框架的标准模型。支持RISCV64，RISCV32，MIPS32指令集架构。

通过实现对应的接口，HDU BOARD可以对不同语言的处理器做差分测试。通过Rust语言的强大生态，我们可以在Rust中通过调用动态链接库、verilated-rs库、QEMU-gdb协议与其它模拟器进行交互。我们也可以通过HDU BOARD比较各个模拟器之间的性能差异。

我们使用HEMU来对指令执行、Cache命中率、BPU命中率进行统计发现可能的优化方向，并且对我们的处理器进行差分测试。