

NSCSCC 2023

NOP 项目决赛设计报告

NOP: A Strong Baseline for LoongArch Out-of-order Processor

项目作者：刘明道 高焕昂 王博文 花佳诚

{liu-md20, gha20, wangbw21, hjc21}@mails.tsinghua.edu.cn

指导教师：陈康 郑宁汉

2023 年 8 月 18 日

目录

插图	2
表格	2
1 项目介绍	3
2 NOP 核微架构设计	3
2.1 实现内容概述	3
2.2 前端各流水段介绍	5
2.3 后端各流水段介绍	7
2.4 性能优化手段介绍	8
2.5 NOP 核的外部接口	9
3 SoC 设计	9
3.1 总述	9
3.2 地址分配	10
3.3 外设控制器	10
4 系统软件	10
4.1 PMON	11
4.2 Linux	11
5 实验评估	12
5.1 实现细节	12
5.2 性能结果	12
5.3 分支预测评估	13
5.4 指令/数据缓存评估	14
5.5 访存流水段优化相关评估	14
6 相关工作	15
6.1 LA32R 单发射处理器	15
6.2 LA32R 超标量处理器	15
6.3 “龙芯杯” 乱序处理器	16
7 总结与讨论	16
7.1 总结	16
7.2 未来可能的工作	16
7.3 致谢	17
8 参考文献	18

插图

2.1 NOP 核微架构设计图	4
3.1 NOP-SoC 结构图	10
4.1 启动 Linux 系统并使用 VGA 显示	12
4.2 驱动 LCD 显示屏	13
5.1 NOP 核的指令/数据缓存评估结果	15

表格

2.1 支持的控制状态寄存器一览表	5
2.2 支持的异常编码表	6
2.3 支持的中断编码表	6
3.1 NOP-SoC 地址映射表	11
3.2 NOP-SoC 控制器配置	11
5.1 NOP 核在比赛提供的性能测试程序上的结果	13
5.2 NOP 核的分支预测模块消融实验的结果	14
5.3 NOP 核在访存流水线上做出的优化的消融实验	14

1 | 项目介绍

本项目是第七届“龙芯杯”全国大学生计算机系统能力培养大赛(NSCSCC 2023)的参赛作品。项目成功地开发了一款基于龙芯架构 32 位精简版(LoongArch32-Reduced, LA32R)指令集的 CPU，命名为**NOP**。**NOP** 属于乱序多发射微架构，整体基于 Tomasulo 动态调度算法的思路 + 重排序缓存实现，并实现了分支预测、指令/数据缓存、数据旁路、推测唤醒等特性。

从功能的角度上，**NOP** 实现了基础的算术指令、分支指令、访存指令，支持精确异常处理、虚实地址转换与 LA32R 指令集规定的各类中断，通过使用 PMON 引导程序，**NOP 可以稳定地启动 Linux 操作系统**。从性能上看，**NOP** 作为一款乱序多发射处理器，在大赛提供的 FPGA 实验平台与性能测试程序 / SoC 上达到了 **107.69 MHz** 的主频与 **1.02 的 IPC**。相比于基线 openLA500 处理器，**NOP** 核的整体加速比达到了 **3.00**，IPC 加速比达到了 1.402。

同时，我们充分利用了大赛实验平台的板载硬件资源，设计了 NOP-Soc 系统以驱动板载外设，同时对相关系统软件的代码做了相应的修改适配。最终，我们可以驱动起开发板上除 USB 外的全部外设。在 Soc 设计、系统软件配置与外设驱动层面，我们在 LA32R 指令集上达到了与之前 MIPS 赛道的作品 [1] [2] 相同的水平。为服务开源教育社区建设，本届比赛结束后我们设计的 SoC 工程文件、修改后的系统软件以及处理器核的工程设计文件将会全部开源。

在本报告中，我们首先介绍最核心的部分，即 NOP 核的微架构设计(§2)。然后，我们会介绍在 LA32R 指令集下，我们新设计的 SoC 与验证设计正确性的系统软件(§3, §4)。接着，我们会用一系列实验数据证明我们设计的部分模块的有效性(§5)。我们在开发本项目的初期阶段预调研了一些工作，列于 §6 节中供读者参考。最后，我们总结并讨论本工作进一步可能改进的空间(§7)。

2 | NOP 核微架构设计

2.1 | 实现内容概述

我们实现了基于 LA32R 指令集的乱序多发射 CPU——**NOP**，其采用基于 Tomasulo + ROB 思想的乱序多发射架构。总体上，我们项目设计的 NOP 核可以被分为前端、后端两个部分：

前端 取指 1，取指 2，译码，重命名，分发

后端 发射、读寄存器、执行、写回、提交

项目 CPU 的整体架构如图 2.1 所示。接下来，我们详细介绍我们的 NOP 核支持的功能特性。

2.1.1 | 指令支持

在指令支持方面，基于 LA32R 指令集，项目成功地实现了运行 Linux 系统所需的所有指令，其中包括：

算数相关 ADD.W SUB.W SLT SLTU NOR AND OR XOR SLL.W SRL.W SRA.W

算数相关 + 立即数 SLLI.W SRRI.W SRAI.W SLTI SLTUI ADDI.W ANDI ORI XORI LU12I.W

算数相关 + PC + 立即数 PCADDI PCADDU12I

分支相关 BEQ BNE BLT BGE BLTU BGEU B BL JIRL

乘除相关 MUL.W MULH.W MULH.WU DIV.W DIV.WU MOD.W MOD.WU

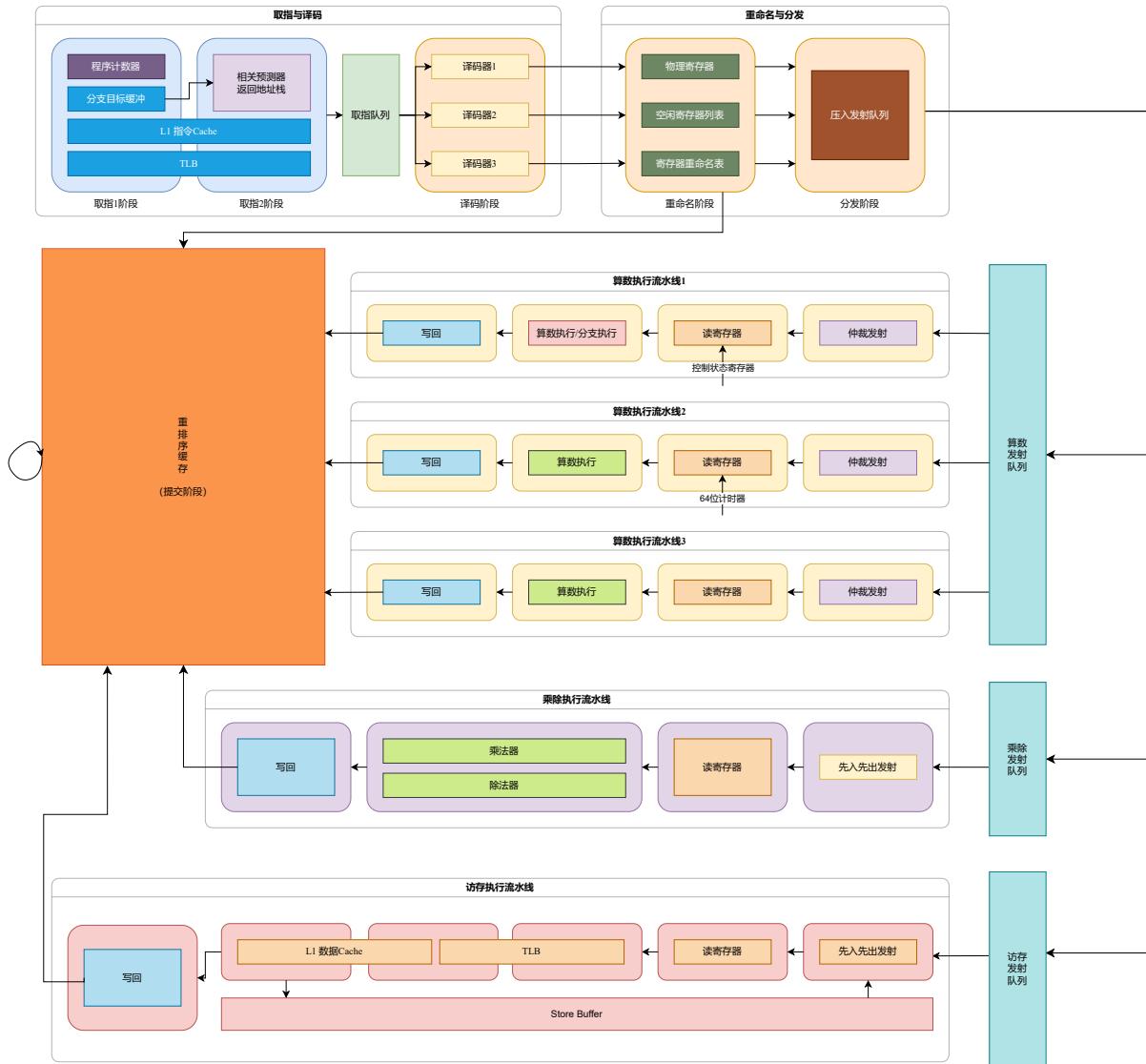


图 2.1: NOP 核微架构设计图

访存与原子访存 LD.W LD.H LD.HU LD.B LD.BU ST.W ST.H ST.B LL.W SC.W

Timer64 计数器相关 RDCNTVL.W RDCNTVH.W RDCNTID.W

特权集指令 CSRRD CSRWR CSRXCHG SYSCALL BREAK ERTN IDLE CACOP

特权级指令 + TLB TLBSRCH TLBRD TLBWR TLBFILL INVTLB

栅障指令 IBAR DBAR

2.1.2 | CSR 寄存器支持

项目支持的 LA32R 规范中要求实现的与异常中断处理和地址翻译有关的控制状态寄存器如表 2.1 所示。

2.1.3 | 异常中断支持

项目实现了精确异常处理，支持的异常见表 2.2，支持的中断见表 2.3。其中，我们自行设计的 NOP-SoC 对 8 位硬件中断的定义见表 3.2。

地址	名称	功能
0x0	CRMD	当前模式信息
0x1	PRMD	例外前模式信息
0x2	EUEN	扩展部件使能
0x4	ECFG	例外配置
0x5	ESTAT	例外状态
0x6	ERA	例外返回地址
0x7	BADV	出错虚地址
0xc	EENTRY	例外入口地址
0x10	TLBIDX	TLB 索引
0x11	TLBEHI	TLB 表项高位
0x12	TLBELO0	TLB 表项地位 0
0x13	TLBELO1	TLB 表项地位 1
0x18	ASID	地址空间标识符
0x19	PGDL	低半地址空间全局目录基址
0x1A	PGDH	高半地址空间全局目录基址
0x1B	PGD	全局目录基址
0x20	CPUID	处理器编号
0x30~0x33	SAVE0~SAVE3	数据保存
0x40	TID	定时器编号
0x41	TCFG	定时器配置
0x42	TVAL	定时器值
0x44	TICLR	定时中断清除
0x60	LLBCTL	LLBit 控制
0x88	TLBREENTRY	TLB 重填例外入口地址
0x180~0x181	DMW0~DMW1	直接映射配置窗口

表 2.1: 支持的控制状态寄存器一览表

2.1.4 | 地址翻译支持

项目支持了基于 TLB 的虚拟地址转换，实现了 LA32R 要求的两种翻译模式：直接地址翻译模式和映射地址翻译模式。

直接地址翻译模式。在这种翻译模式下，物理地址默认直接等于虚拟地址的 [31:0] 位（不足补 0），此时整个虚拟地址空间均为合法的。

映射地址翻译模式。在这种翻译模式下，先尝试按照直接映射模式进行翻译：系统配置两个直接映射配置窗口 DMW0 和 DMW1，每个窗口可以配置一个 2^{29} 字节固定大小的虚拟地址空间，当虚地址命中某个有效窗口时，其物理地址即为虚地址的 [28:0] 位拼接上命中窗口所配置的物理地址高位；失败后再尝试按照页表映射模式进行翻译，将 TLB 表全相连进行查询，如触发例外，交由软件进一步处理。

本项目支持 TLB 的查找、读写、无效指令，实现了所有有关的 CSR 寄存器。对于虚实地址转换，我们在取指和访存阶段均采用了两段式的翻译过程，这有利于对流水线时序的优化。

2.2 | 前端各流水段介绍

本项目的前端采用五级流水，分别为：取指 1 (IF1)、取指 2 (IF2)、译码 (DECODE)、重命名 (RENAME)、分发 (DISPATCH)，下面分别进行细述。

Ecode	例外代号	例外类型
0x0	INT	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
0x7	PPI	页特权等级不合规例外
0x8	ADEF	取指地址错例外
0x9	ALE	地址非对齐例外
0xB	SYS	系统调用例外
0xC	BRK	断点例外
0xD	INE	指令不存在例外
0xE	IPE	指令特权等级错例外
0x3F	TLBR	TLB 重填例外

表 2.2: 支持的异常编码表

中断位	描述
IS[1:0]	软件中断状态位
IS[9:2]	硬中断状态位
IS[11]	定时器中断状态位

表 2.3: 支持的中断编码表

2.2.1 | 取指与译码

取指 1 阶段。在本阶段，程序计数器（PC）产生当前要流入流水线的指令地址。该地址会被运用到前端地址翻译插件中进行虚实地址转换的第一阶段，运用到指令缓存查询的读口与分支预测器的读口。

取指 2 阶段。在此阶段进行虚实地址转换的第二阶段，产生该指令对应的物理地址。产生的物理地址与指令缓存查询的输出口的 Tag 进行比较，如果命中缓存则将对应的指令数据取出压入取指队列，不然将本阶段停滞并进行缓存不命中的处理。在我们的默认配置下，每周期最多向取指队列中压入 4 条指令。分支预测（包括 Correlating Branch Predictor 和返回地址栈 RAS）产生下一条指令预测的地址，并回送给程序计数器。

译码阶段。在这阶段，我们每周期从取指队列中弹出至多 3 条指令进行译码。译码使用 LA32R 架构规定的指令编码格式，译码主要是使用组合逻辑判断指令类型及其调用参数与写回参数。

2.2.2 | 重命名与分发

重命名阶段。NOP 采用了显式重命名的方法来避免 Scoreboard 算法存在的 WAR 和 WAR 冲突问题。具体来说，我们维护的物理寄存器堆（Physical Register File，PRF）中共有 63 项，而对于每个逻辑寄存器（去除 \$r0 共有 31 个），只需要记录其当前被重命名到的物理寄存器的索引即可。我们将这个重命名关系存储在叫做重命名映射表（RAT）的数据结构中。

这个重命名映射表数据结构需要维护两个示例，sRAT 表示推测性的版本，aRAT 表示架构性的版本，其区别在于在一条指令提交的时候，其会将其重命名影响写入 aRAT 中，而 sRAT 永远存储着当前乱序执行流水线最新的执行结果，这样方便我们在分支预测错误或其他需要转移控制流的情况下方便回滚寄存器状态。

在分配与回收物理寄存器的过程中，NOP 处理器使用“空闲寄存器列表”（Free List）这一数据结构，其采用先入先出策略，优先分配上一次被使用时间最长的物理寄存器给新的逻辑寄存器使用，

而接收 WB 阶段的信号清空空闲寄存器的使用标志。

此外，在本阶段，我们还将译码出的指令插入重排序缓存，并设定其运行状态的初值。

分发阶段。本阶段的主要作用是将指令装入后端的待发射队列。根据指令的类型，我们设置了算数发射队列、乘除发射队列和访存发射队列三种发射队列。本阶段将根据译码阶段产生的指令运行单元类型将指令装载入对应的发射队列中。如果对应的发射队列已满，目前的处理方式是停滞本流水线并等待有空闲槽位时再压入队列。

2.3 | 后端各流水段介绍

我们的后端处理逻辑分不同的流水线进行，包括算数流水线、乘除流水线与访存流水线，其中算数流水线设置了三路，这三路各不相同，其他流水线各设置了一路。这些后端流水线大致都可以切分成发射 (ISS)、读寄存器 (RRD)、执行 (EXE)、写回 (WB) 和提交 (RETIRE) 这五个阶段，下面分别进行叙述。

2.3.1 | 指令发射

发射阶段。这一阶段的作用主要是判断处于发射队列中的哪些指令可以被发射。不同发射队列的指令之间的发射如不产生数据相关，则互不影响，这是乱序执行架构成功的原因。对于乘除发射队列和访存发射队列，由于只有一个可执行功能单元，于是发射采用顺序 FIFO 的方式，等到一条指令的前置相关指令均执行完毕时即可发射；对于算数发射队列，由于我们设置了三个不对称的功能单元，必须设置一个仲裁逻辑对该指令是否可以发射、该指令由谁来发射进行仲裁。

2.3.2 | 指令读寄存器

读寄存器阶段。这一阶段正如其名，主要的作用是对物理寄存器堆进行读操作。

要注意，算数流水段会在这个周期对要用到当前指令执行结果的指令进行远程唤醒（在这个周期，当前指令 A 指示清除等待写寄存器的 Busy 标记）；再下一周期，在发射队列中等待使用该指令 A 执行结果的指令 B 可以被发射；再下一周期，本指令 A 进入 WB 阶段，而使用该寄存器的指令 B 进入 RRD 阶段，可以读取到本阶段指令 A 的执行结果。

如果某访存流水段指令 C 推测唤醒失败，指令 C 会通知当前所有在 RRD 阶段的指令，并将其停滞在本阶段，直到缓存缺失的问题处理完毕，指令 C 才会释放推测唤醒标志。

对于访存流水线，RRD 阶段还需要计算生成访存的虚拟地址。

2.3.3 | 指令执行

不同种类后端流水线的执行阶段有较大的差别，在这里我们分开叙述。

算数执行流水线。算数执行流水线共设置了三条，三条均为 1 个周期即可执行完本阶段。

三条流水线除均设置 ALU 算数组合逻辑单元之外，第 1 条算数执行流水线被赋予了执行分支指令 (BRU) 与读 CSR 寄存器 (CSR Read) 的逻辑，第 2 条算数执行流水线被赋予了读计时器 (Timer64) 的功能。

乘除执行流水线。乘除执行流水线在这个阶段执行乘法、除法操作，这个周期视要执行的指令种类会执行数个周期后退出，其间指令停滞在本流水段中。

访存执行流水线。访存执行流水线的执行阶段分为 MEMADDR，MEM1，MEM2 三段。

MEMADDR 阶段的目的是为了优化全相连 TLB 表的查找时序，在本阶段指令会进行虚实地址翻译第一阶段，并送入 DCache 的读口读相应数据。

MEM1 阶段会完成物理地址的翻译过程，并获得当前访存是否会触发异常的有关信息，并进行相应的处理。本阶段会执行推测唤醒，即假设下一阶段拿到的缓存数据是有效的，唤醒所有等待着本指令结果的指令使其可以被发射。

MEM2 阶段会在当前访存没有异常时获取当前访存地址的数据结果。如果发现本阶段发生了缓存缺失，那么执行缓存重填的逻辑。对于存储（Store）指令、原子存储指令（SC）以及未缓存地址段的读取（Load）指令，由于避免其执行对处理器状态产生的变化，我们设置 StoreBuffer 这一数据结构缓存其行为，并在该指令提交时再执行，使其产生效果。本阶段会控制所有后端流水线的推测唤醒情况，如果本阶段发生了缓存缺失，那么将所有后端流水线处于 RRD 阶段的指令停滞。

StoreBuffer 在指令提交时需要重新进入访存执行流水线。具体来说，访存执行流水线的 ISS 阶段除了从访存发射队列中发射指令，也从 StoreBuffer 中取已经被提交的指令发射，后续该指令执行的逻辑与上述类似，只不过提前缓存了所要写入的物理地址，以及在 MEM2 阶段后会继续执行，经历写回阶段的操作。

2.3.4 | 指令写回与提交

写回阶段。在这一阶段，执行结果会被写入物理寄存器与 sRAT，并将当前指令的最终执行状态更新到 ROB 中，等待 ROB 的对指令序列的顺序提交。

提交阶段。因为精确异常的要求，我们必须要实现指令的顺序提交。这通过在 ROB 本身的提交阶段实现，只有经过了提交阶段的指令才算是真正完成执行，释放出其在 ROB 中所占的表项。ROB 会根据当前指令执行的最终状态决定该周期最多可提交几条指令（最多为 3 条）、指令提交后是否要回滚流水线状态（如发生了分支预测错误或其他需要切换控制流的行为）等。

2.4 | 性能优化手段介绍

2.4.1 | 分支预测

我们的分支预测主要分三个主要模块，分支目标缓存（BTB）、相关预测器（Correlating Predictor）与返回地址栈（Return Address Stack, RAS），下面我们分别详细叙述。

相关预测器。相关预测器用于预测当前分支指令**是否跳转**。相关预测器的核心在于利用其它分支的行为来辅助对当前分支的预测。具体来说，我们设置 2^{13} 个分支模式表项，每个表项为一个 2-bit 信息，其中如果高位为高则代表预测跳转。这 2^{13} 个分支模式表项组织成了一个矩阵的形式，其中有 2^5 行为当前的全局历史信息（GHR）， 2^8 列为当前分支指令对齐到字的地址。这样的预测既考虑了全局的分支信息（GHR），又考虑了在某个特定位置的分支指令可能具有一些倾向性的局部情况，因此可以取得较好的预测效果，见 §5.3 节的实验结果。

一旦相关预测器预测跳转，我们需要知道跳转的目标地址。而跳转的目标地址由以下两个模块提供，其中返回地址栈的优先级高于分支目标缓存。

分支目标缓存。分支目标缓存主要用于存储与分支指令相关的**目标地址信息**。BTB 的表项组织成了直接映射的形式，其共有 1024 项，索引为当前指令地址按字对齐后的低 10 位。每个 BTB 表项存储以下内容：(i) 当前存储分支指令地址的 Tag；(ii) 当前存储分支目标按字对齐的地址；(iii) 是否是调用指令；(iv) 是否是返回指令。其中 (iii) 和 (iv) 表示对应的分支目标应由返回地址栈提供。

返回地址栈。返回地址栈是对调用和返回这两个特殊指令的优化。因为函数的调用和返回一定是满足后入先出的性质，因此我们特地设置 8 项返回地址栈表项来记录每一次调用，并在返回时返回到上一次记录调用，也就是栈顶项的下一条指令。根据我们对 LA32R 编译器的理解，我们对调用指令的定义为 BL 或链接返回地址到 \$r1 的 JIRL 指令，对返回指令的定义为 JIRL \$r0,\$r1,0 指令。

2.4.2 | 指令/数据缓存

我们设置了容量均为 8KiB 大小的 L1 指令/数据缓存。我们的 L1 指令/数据缓存的缓存块大小均设置为 64 字节，且都设置了 64 行，每个缓存设置 2 路，这构成了缓存的 8KiB 大小。我们的指令/数据缓存均使用 VIPT 机制，即使用虚拟地址作为索引，使用物理地址作为判断是否匹配的标签。由于我们的缓存单路没有超过 4K，即一个页面的大小，虽然我们是 VIPT，但实际和 PIPT 是没有区别的。由于 Linux 本身保证了不会发生缓存别名（Cache Aliasing）现象，即不会出现多个虚拟地址映射同一个物理页的情况，我们不需对这种情况特殊照料。

2.4.3 | 数据旁路

为了进一步缓解算术指令的 RAW 相关，我们设置了旁路机制。即如果读取到当前周期某条在算数流水线中 WB 阶段的指令在写寄存器，则直接将该值作为算数执行流水线 EXE 阶段的输入，这样让具有 RAW 相关的指令对能够连续执行。

2.4.4 | 推测唤醒

如 §2.3.3 节所述，我们在后端的访存执行流水线加入了推测唤醒机制。具体来说，我们在 MEM1 阶段假设下一阶段的缓存数据的有效性，唤醒等待指令，并在实际数据可用之前执行这些指令。这种方法可以提高程序执行效率，减少流水线停顿。一旦在 MEM2 阶段发现缓存数据缺失，则“推测”错误，需要暂停所有后端流水线的读寄存器阶段等待缓存重填结束。虽然等待重填的缺失开销很大，但我们相信，在程序访存的空间局部性原理下，数据缓存缺失的几率是很小的，等待重填的开销并没有将先读后用这种类型的指令对之间的气泡都缩短一个周期带来的收益大。事实上，我们通过实验也证明了推测唤醒机制的有效性，见 §5.5 节。

2.5 | NOP 核的外部接口

NOP 核的外部接口按照 Chiplab SoC 规定的处理器核的外部接口设计。具体来说，NOP 核通过 AXI4 协议与外部进行数据通信，总线位宽 32 位。此外，NOP 核还接收外部传入的 8 位硬件中断。

3 | SoC 设计

3.1 | 总述

为了充分展示 NOP 处理器的功能，我们设计了 SoC 以驱动板载外设（以下简称 NOP-SoC）。除 NOP 核外，NOP-SoC 中可驱动的设备列举如下。

- DDR3 DRAM (K4B1G1646G)，用作主存
- NAND Flash (K9F1G08U0C)，用做外存
- SPI Flash (EN25F80)，用做固件，烧录启动引导程序
- UART 串口控制器
- Ethernet 控制器 (DM9161AEP)
- GPIO，可与板载数码管，LED 灯，拨码开关等外设交互

- LCD 显示器 (NT35510)
- VGA 输出设备
- PS2 输入设备

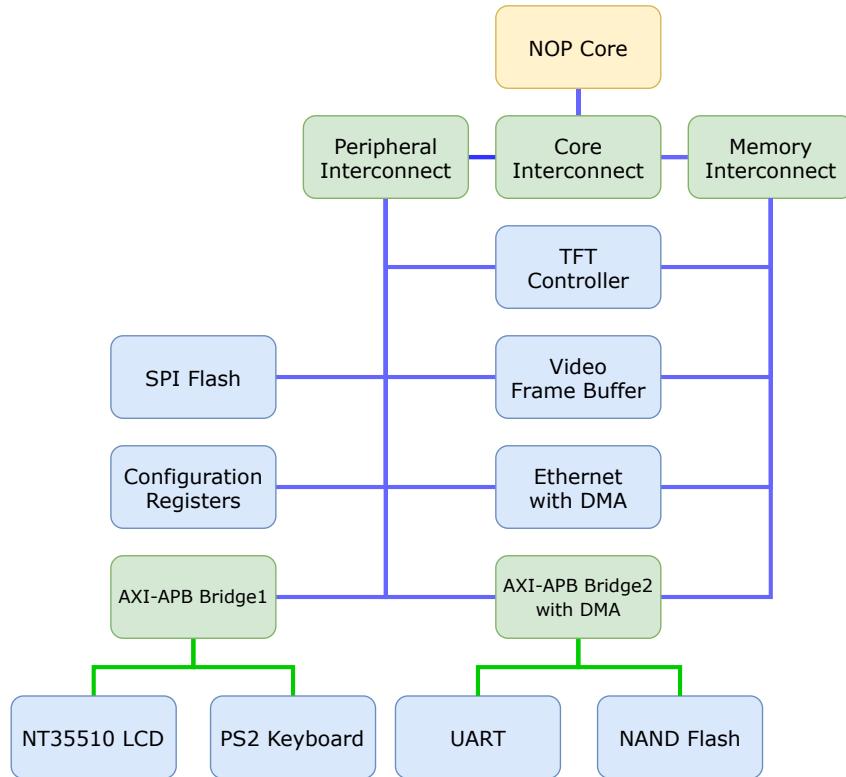


图 3.1: NOP-SoC 结构图

如图 3.1 所示，NOP-SoC 采用 AXI 与 APB 总线进行连接。其中，蓝色线表示 AXI 总线，绿色线表示 APB 总线；黄色设备是处理器核心，绿色设备是协议适配器或总线互联，蓝色设备表示驱动的外设。为简洁清晰，图中省略了中断线等部分连线。

3.2 | 地址分配

NOP-SoC 对各外设的地址映射规则如表 3.1 所示。

3.3 | 外设控制器

考虑到控制器实现和软件驱动设计的复杂性，NOP-SoC 主要移植了已有的外设控制器进行外设驱动，详见表 3.2。而对于外设中断，LA23R 指令架构规定了 8 个硬件中断位。NOP-SoC 选用了其中的低 6 位，高 2 位置零。具体分配一并列举于表 3.2。

4 | 系统软件

在功能测试与性能测试的基础上，我们使用 PMON 和 Linux 来验证 NOP-CPU 实现的正确性。

地址段	起始地址	结束地址	分配大小
DDR3 Memory	0x00000000	0x07FFFFFF	128 MB
SPI Memory	0x1C000000	0x1C0FFFFF	1MB
TFT Controller	0x1D010000	0x1D01FFFF	64 KB
PS2 Controller	0x1D020000	0x1D02FFFF	64 KB
NT35510 Controller	0x1D030000	0x1D03FFFF	64 KB
Video Frame Buffer Read Controller	0x1D050000	0x1D05FFFF	64 KB
Video Frame Buffer Write Controller	0x1D060000	0x1D06FFFF	64 KB
Configuration Registers	0x1FD00000	0x1FD0FFFF	64 KB
UART Controller	0x1FE00000	0x1FE0FFFF	64 KB
NAND Controller	0x1FE70000	0x1FE7FFFF	64 KB
SPI Controller	0x1FE80000	0x1FE8FFFF	64 KB
Ethernet Controller	0x1FF00000	0x1FF0FFFF	64 KB

表 3.1: NOP-SoC 地址映射表

外设	控制器来源	中断位
Ethernet	Chiplab	HWI0
UART	Chiplab	HWI1
SPI Flash	Chiplab	HWI2
NAND Flash	Chiplab	HWI3
DMA	Chiplab	HWI4
PS2	Altera	HWI5
VGA TFT	Xilinx	-
Video Framebuffer Read/Write	Xilinx	-
LCD	NonTrivial-MIPS	-

表 3.2: NOP-SoC 控制器配置

4.1 | PMON

我们使用 LA32R 指令架构下的 PMON 作为启动引导器。我们首先将 PMON 二进制文件使用 GZip 进行压缩，并将压缩包与解压程序一同使用串口烧写至 SPI Flash 中作为启动固件。上电重置后，CPU 首先执行 SPI Flash 中的解压程序，将 PMON 解压至内存后跳至 PMON 入口地址开始执行。值得一提的是，PMON 本身已是具有一定复杂程度的操作系统，也可以作为硬件正确性验证的一部分。

进入 PMON 后，我们使用 TFTP 协议，通过以太网加载 Linux 系统的镜像。镜像有两种加载方式，一是加载到内存中，接下在配置寄存器后跳至内存指定地址以实现启动；另一种方式则是将镜像写入 NAND Flash 中。每次重置后，从 Flash 中加载内核镜像到内存，再进行跳转。

4.2 | Linux

我们以 la32r-Linux [3] 为基础，构建了适合在 NOP-SoC 上启动的 Linux 镜像。我们首先根据 NOP-SoC 的地址段为 Linux 配置相应的设备树，并修改适配了 NOP-SoC 外设控制器对应的驱动程序。由此，我们可以在 Linux 中驱动串口，以太网，NAND Flash，LCD 显示屏，VGA 和 PS2 键盘等外设。此外，我们使用 BuildRoot 和 BusyBox 构建了常用的用户态程序和用户文件系统，将此集成于系统镜像中，便于测试运行各类常用程序。

NOP 核可以稳定地在 NOP-SoC 上运行 PMON 启动引导器和 Linux 操作系统。在这里我们附上启动 Linux 系统并使用 VGA 显示（图 4.1）和在系统内向设备写入二进制文件驱动 LCD 显示屏（图 4.2）的截图。

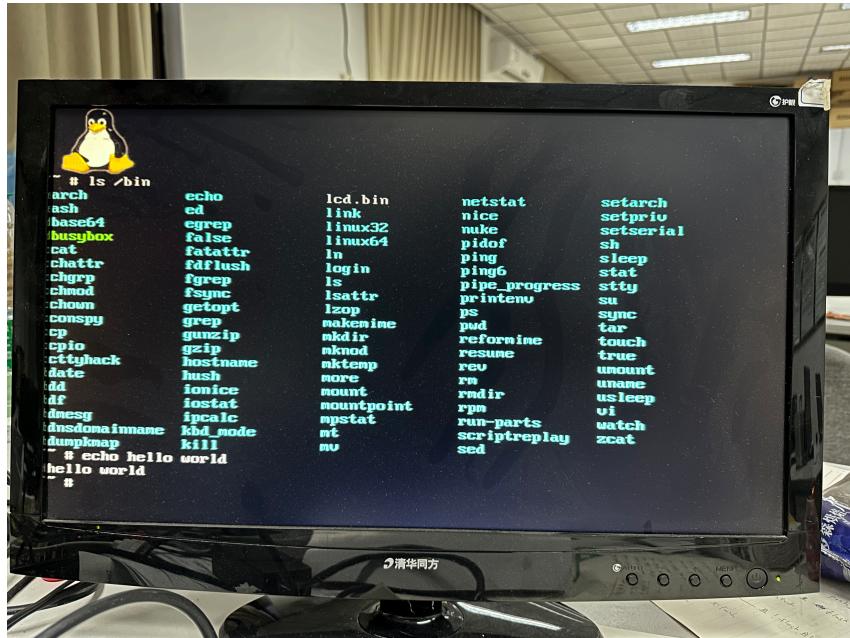


图 4.1: 启动 Linux 系统并使用 VGA 显示

5 | 实验评估

5.1 | 实现细节

开发语言。为加速开发，我们使用高抽象级别的开源硬件描述语言 SpinalHDL 开发本项目，并将项目编译成 Verilog 供仿真与综合实现。这样有助于降低代码的复杂性，设计的理解和维护变得更加容易，还保证了仿真的可靠性，将现代软件开发与硬件设计相结合，使开发效率和设计质量均得到较大的提升。我们还采用了 sbt 作为本项目的管理及自动构建工具，这是使用 Scala 语言进行硬件描述的一个自然选择，使我们的项目构建过程变得高效、可控和可维护。

设计模式。在设计模式上，我们遵循了 SpinalHDL 社区成熟的项目 VexRiscv [4] 的设计模式。作为 CPU 流水线的底层设施，我们复用了其对于流水段（Stage）、流水线内逻辑区域（Plugin）的定义，实现了各个功能部件的深度解耦，使得我们的 CPU 高度可配置，开发更加便捷。

开发平台。我们使用的硬件平台是比赛提供的龙芯体系结构教学实验箱（Artix-7），而软件上我们使用 IntelliJ IDEA 作为集成开发环境，将构建产生的 Verilog 文件使用 Xilinx Vivado 2019.2 进行行为仿真和综合实现。此外，我们借用了实验发布包提供的基于 nemu 的 difftest 框架对我们的 CPU 进行调试。我们还在 GitLab 平台上配置了 CI/CD 流程以实现自动化地进行 (i) Verilog 生成、(ii) 功能测试、性能测试与系统 SoC 的综合实现与比特流生成、(iii) 提交包的生成，这进一步提升了我们的开发效率、加强了协作效果。

5.2 | 性能结果

在本节我们展示 NOP 核最终在比赛提供的性能测试 SoC 与程序上达到的结果。我们比较的基线是 openLA500 [5] 处理器，其主频设置为 50MHz，我们的主频则为 107.69MHz，SoC 板载计数器的时钟频率为 100MHz。我们展示的指标如下：(i) PerfUp：性能加速比，指参考核 [5] 与我们的 NOP 核在性能测试程序上实际执行时间的比值；(ii) IPCUp：IPC 加速比，指参考核 [5] 与我们的 NOP 核在性能测试程序上展现的 IPC 的比值；(iii) IPCReal：实际 IPC 值，指我们的 NOP 核在性能测试程序

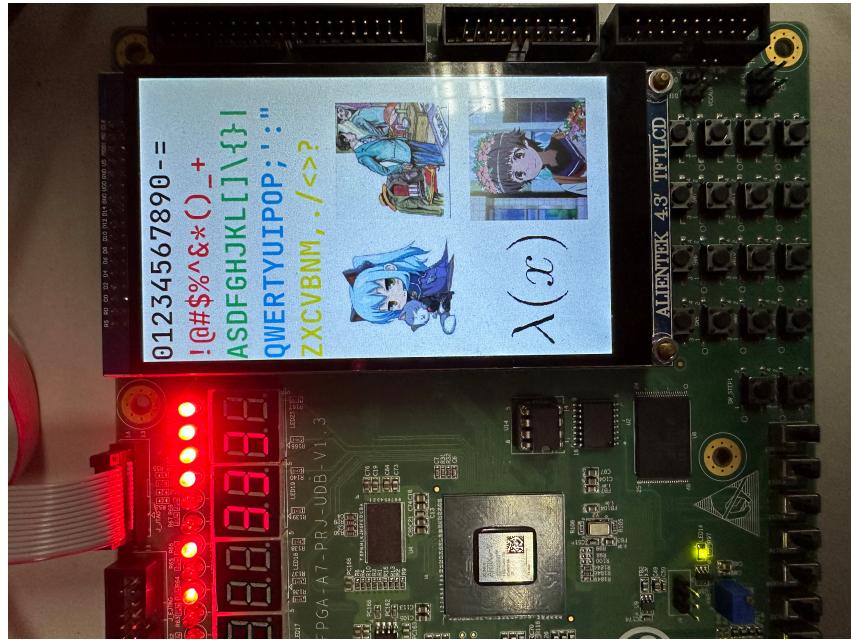


图 4.2: 驱动 LCD 显示屏

上的 IPC。最终的平均值采用几何平均的方式计算。

任务	指标			任务	指标		
	PerfUp	IPCUp	IPCReal		PerfUp	IPCUp	IPCReal
bit_count	3.86	1.79	1.63	quick_sort	1.95	0.91	0.68
bubble_sort	2.28	1.06	0.63	select_sort	3.40	1.58	1.14
coremark	2.58	1.20	0.87	sha	3.81	1.77	1.34
crc32	4.89	2.27	1.42	stream_copy	2.73	1.27	0.87
dhrystone	2.51	1.19	0.89	stringsearch	3.11	1.44	1.15
Geo. Mean	3.00	1.40	1.02	-			

表 5.1: NOP 核在比赛提供的性能测试程序上的结果

从表 5.1 中我们可以看到，我们的乱序多发射架构对于算数类指令，如 bit_count, crc32, sha 的提升明显。而对于存在较多随机性强的分支指令的程序，如与比较和排序有关的 bubble_sort 与 quick_sort，较深的流水线给我们的分支恢复带来了一定量的惩罚。

5.3 | 分支预测评估

在本节我们通过实验数据证明我们分支预测的有效性。我们比较的指标为不同实验设定下的分支预测错误率，即预测错误的分支指令占控制流中所有分支指令的比例。我们提供如下三种实验设定：(i) NOP-B，取消了分支预测学习功能的处理器核，总是预测分支失败；(ii) NOP-R，只开启了相关预测器和分支目标缓存的处理器核，但没有对调用与返回指令做特殊处理；(iii) NOP，所有分支模块都启用的处理器核。相关的实验结果如表 5.2 所示。

首先从表 5.2 中我们可以看到，在开启了相关预测器模块和分支目标缓存模块之后，分支预测错误率有了明显的常数倍数量级的降低，这说明了分支预测模块的有效性。然后我们将 RAS 开启后一些性能进一步加速较大的测例高亮显示，如 bit_count, crc32 等，这些测例都是函数调用密集型的，进一步证明了我们的 RAS 模块的作用。

任务	架构			任务	架构		
	NOP-B	NOP-R	NOP		NOP-B	NOP-R	NOP
bit_count	42.93	19.35	5.55	quick_sort	31.36	22.98	22.95
bubble_sort	50.27	15.94	15.94	select_sort	90.68	5.05	5.05
coremark	46.4	10.59	9.85	sha	86.46	1.65	1.36
crc32	93.23	3.68	1.87	stream_copy	98.75	1.53	1.21
dhryystone	68.19	5.09	2.44	stringsearch	74.48	8.44	6.02
Avg.	68.28	9.43	7.22		-		

表 5.2: NOP 核的分支预测模块消融实验的结果

任务	架构			任务	架构		
	NOP-S	NOP-D	NOP		NOP-S	NOP-D	NOP
bit_count	3.11	3.55	3.86	quick_sort	1.61	1.76	1.95
bubble_sort	1.84	1.95	2.28	select_sort	2.60	3.10	3.40
coremark	2.04	2.22	2.58	sha	2.76	3.28	3.81
crc32	3.19	4.46	4.89	stream_copy	2.23	2.30	2.73
dhryystone	2.24	2.30	2.51	stringsearch	2.57	2.68	3.11
Geo. Mean	2.37	2.66	3.00	Freq. (MHz)	86.67	98.46	107.69

表 5.3: NOP 核在访存流水线上做出的优化的消融实验

5.4 | 指令/数据缓存评估

在本节我们通过实验数据说明我们选择给定的指令与数据缓存参数的原因。我们选择的指标与 §5.2 节相同。我们比较在同样大小的前提下，不同的块大小 (bs) 和缓存行 (sets) 对最终处理器性能的影响。相关的实验结果如图 5.1 所示。

从图 5.1 中我们可以看到，我们选择的指令/数据缓存参数在各个测例上表现均为最优，且在指令缓存上的优化更为明显。我们将这归因于单次更大的 Burst 取出更多指令可以有效地提高单次总线事务的效率。

5.5 | 访存流水段优化相关评估

在本节我们通过实验探究 NOP 对访存流水段优化的实际效果。我们比较的指标为不同架构在分别使用其在板载上可以达到的最高主频的前提下，相比于基线 openLA500 的加速比。我们提供如下三种实验设定：(i) NOP-S，使用传统的单阶段地址翻译，但访存阶段的 MEMADDR 段取消，其所有功能并入 MEM1 段；(ii) NOP-D，将单阶段地址翻译切分成两阶段地址翻译，取指阶段两阶段为 IF1 和 IF2，访存阶段为 MEMADDR 与 MEM1；(iii) NOP，在 NOP-D 的基础上启用了推测唤醒的处理器核。相关的实验结果如表 5.3 所示。

从表 5.3 中我们可以看到，流水段切分给主频的提升带来了显著的影响，相比于 NOP-S 的 86.67 MHz，NOP-D 可以达到 98.46 MHz 的板载频率。在加入了推测唤醒功能之后，我们减弱了组合逻辑已经十分复杂的 MEM2 阶段的逻辑，而将唤醒的逻辑放在了 MEM1 端，MEM2 段只需要控制唤醒是否失败的信号，这不仅成功的提升了处理器的主频，还有效提升了处理器的 IPC。实验数据表明我们对访存流水线这两个初级的优化方案是比较成功的。

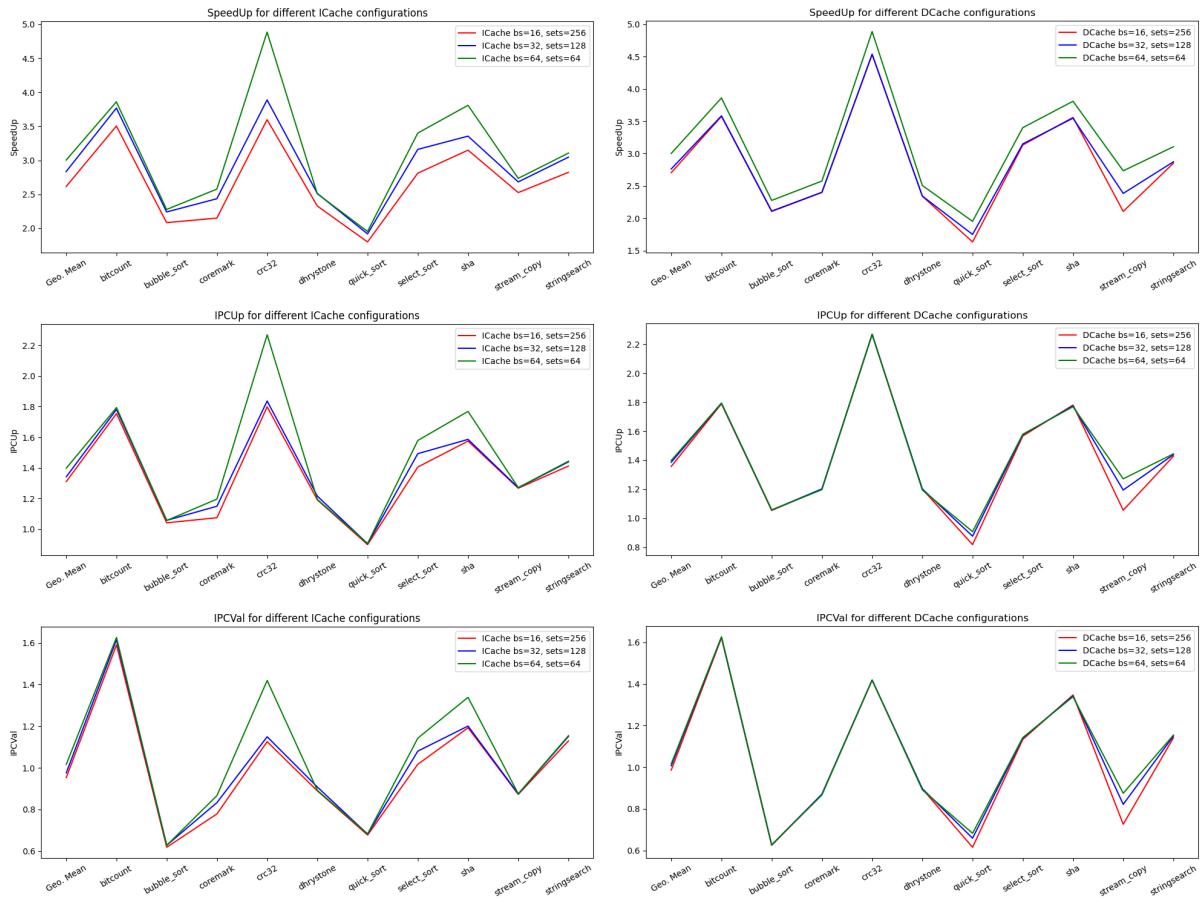


图 5.1: NOP 核的指令/数据缓存评估结果

6 | 相关工作

在开发本项目的过程中，我们调研到了以下与本项目有关的工作，希望可以简化读者的调研过程。

6.1 | LA32R 单发射处理器

我们的基线是 openLA500 处理器 [5]，其为单发射五级流水的示例 CPU，且具备了异常处理、虚实地址翻译功能，并附了简单的分支预测、指令/数据缓存等加速手段；相比于 MIPS 赛道的基线，LA 赛道的基线方法是十分完善的，其已具备了启动 Linux 的能力。“流云”处理器 [6] 是一款基于 LoongArch 指令集的单发射 7 级流水通用中央处理器核，对基线处理器做了进一步的流水段切分。

这些处理器往往可以达到很高的主频，但并未对超标量执行与动态调度算法做出过多探索，其 IPC 指标往往存在较大的缺陷。

6.2 | LA32R 超标量处理器

“卅”处理器 [7] 是一款基于记分牌算法 (Scoreboard 算法) 的动态调度乱序执行处理器，其使用两路算数执行单元、一路乘法执行单元、一路除法执行单元、一路访存执行单元，也支持了 TLB 等 LA 指令集规定的特权级功能，但其存在因记分牌算法导致的 WAR 和 WAW 冒险，进而有大量流水线阻塞现象的出现，无法有效开发出乱序的指令级并行。“争流”处理器 [8] 也是一款基于 LA32R 指

令集的动态超标量处理器，其包括 8 个流水级，整个核分为前端，后端和访存子系统三部分，可仿真进行性能测试，并在仿真环境中启动 Linux 5.14 内核，但并未进行上板验证，且时序有显著不足。

相比于上述处理器，我们的 NOP 处理器在兼具了实现 LA32R 指令集所规定的所有功能特性的同时，在性能测试表现优异，既达到了 107.69 MHz 的主频，也有超过 1.00 的 IPC 指标，可以成为后续 LA32R 指令集乱序执行处理器开发的完善基线。

6.3 | “龙芯杯” 乱序处理器

在最近几年的“龙芯杯”比赛中，由于只设 MIPS 赛道，所以有了大量基于 MIPS 指令集的高性能开源处理器设计的出现，我们在这里简单回顾。

2020 年的 NonTrivial-MIPS [2] 实现了具有 10 段流水段的双发射处理器，实现了异常处理机制、协处理器、可扩展的专用计算模块，可以启动 Linux 内核，其贡献的系统展示 SoC 设计与配套的 Linux 源码修改方式至今仍有很大的影响力。2021 年的 LLCL-MIPS [9] 实现了顺序双发射八级流水线的处理器，首创性地基于 SpinalHDL 进行开发，继承了 NonTrivial-MIPS [2] 对 VGA、PS/2、串口、Flash 等外设的支持，在大赛提供的开发板上频率达到了 120 MHz。2022 年的 ZenCove [1] 实现了五发射处理器，仿照 VexRiscv [4] 的设计模式实现了高度可配置的处理机核，在支持启动 Linux 的同时驱动起了开发板上除 USB 之外的全部外设，性能跑分超过了 NonTrivial-MIPS [2]。

上述处理器核为我们的 NOP 核开发带来了一定的启发。但是，由于 MIPS 和 LA32R 指令集的不同，如去掉了延迟槽的设计，不同的异常中断、虚拟地址转换逻辑等等，我们最终设计乱序执行的整体思路虽然与这些高性能乱序处理器核相同，但在具体实现细节上有很大的出入，详见 §2 节的描述。同时，系统软件因为指令集的不同而需要重新配置，这使得我们需要搭建全新的 SoC 系统，并对 LA32R 版本的 PMON 和 Linux 内核进行相应的修改，详见 §3 和 §4 节的描述。

7 | 总结与讨论

7.1 | 总结

本项目开发的基于 LA32R 指令集的 CPU **NOP** 基于 Tomasulo 动态调度算法的思路 + 重排序缓存实现。在功能上，我们支持了基础指令、精确异常、中断处理、地址翻译，通过使用 PMON 引导程序，可以稳定启动 Linux 操作系统。在性能上，我们可以作为 LA32R 后续处理器开发的强壮基线，在大赛提供的 FPGA 实验平台与性能测试程序 / SoC 上达到了 **107.69 MHz** 的主频与 **1.02 的 IPC**。同时，我们充分利用了大赛实验平台的周边硬件资源，搭建了首个适配 LA32R 指令集的 SoC 系统，可以驱动起实验开发板上除 USB 之外的所有外设。此外，为了适配新的外设，我们对 LA32R 版本的 Linux 也做出了相应的修改。

总结来说，本次大赛我们完成了从处理器核微架构的设计，到处理器核与外设通信的协议的实现，到适配 LA32R 的板载 SoC 系统的搭建，到对上层相应系统软件的修改与适配工作。我们希望我们的工作可以标志成为基于 LA32R 指令集开发的处理器的强壮基线，并对后续工作产生深远的启发与影响。

7.2 | 未来可能的工作

限于本次比赛时间的限制，我们未能对以下方向做出充分的探索，希望以下内容可以给未来工作带来启发。

对处理器主频的进一步优化。目前我们处理器频率的瓶颈在于重排序缓存（ROB）模块的实现。目前的 ROB 模块设置为循环队列，需要记录每条指令解析的微码（ μ op）和其运行状态（State），采用 SDPRAM 实现。为了保证乱序执行的 IPC 指标，我们设置了 32 项 ROB 表项，综合后 ROB 模块占板上面积约为 25%。目前核的关键路径在于 ROB 循环队列处理插入表项和弹出表项的有关逻辑，后续可以对此处逻辑进行时序优化来进一步提升主频。

对处理器 IPC 的进一步优化。目前我们的 IPC 瓶颈在于两个部分。(i) 流水段更深的乱序执行流水线代表着更大的分支恢复惩罚，这在面对随机性强的分支指令时表现得尤其显著（见表 5.1）。可以考虑使用更加有效的分支预测手段 [10] 来对分支指令进行优化，以此降低分支恢复所带来的惩罚。(ii) 虽然我们支持多口提交，为了调试方便，现在访存相关的指令被设置了一周期最多提交一条。这里可以进一步的进行完善，通过再将提交阶段和 StoreBuffer 的相应逻辑重写，应该可以进一步获得 IPC 指标的提升。

对系统软件的进一步优化。目前我们采用了 ChipLab 中提供的 PMON 作为启动引导程序。该启动引导程序由 MIPS 版移植而来，其中目前存在较多的未定义行为，如 CSR 写入非法值，非法内存段访问等。为了提升系统稳定性，本项目修复了其中的部分问题，同时也保留了 ChipLab SoC 中对内存读写异常进行重定向的设计，以应对尚未完全修复的内存异常。后续工作可以考虑进一步对系统软件进行调试和优化，以提升系统的规范度和稳定性。

USB 外设驱动 USB 外设驱动较为复杂；在本项目中，我们尚未完成对板上 USB 外设的驱动。在“龙芯杯”历史上，NonTrivialMIPS [2] 完成了对 USB 外设的支持，而后续的 LLCL-MIPS [9] 和 ZenCove [1] 均未成功驱动。后续工作可以考虑使用 LA32R 指令集在开发板上完成这一富有挑战性的外设驱动。

7.3 | 致谢

我们要感谢刘卫东、陆游游、陈康、李山山、郑宁汉等老师的指导，也要感谢崔轶锴、高一川、陈嘉杰、陈晟祺等学长们的帮助。同时，我们要感谢清华大学计算机系基础教学实验室提供的 Alder 服务器（i9-12900KS）来作为项目开发与我们 CI/CD 搭建的平台。当然，也要感谢每一位参赛选手的努力、每一位指导教师的支持、每一位赛事组委会成员的付出，让“龙芯杯”系统能力大赛今年能够再创新的辉煌。

8 | 参考文献

- [1] Y. Cui, W. Zhang, and T. Wang, “Zencove-zoom,” <https://github.com/zencove-thu/zencove-zoom>, 2022.
- [2] Y. Zhou, S. Chen, X. Liu, and J. Chen, “Nontrivial-mips,” <https://github.com/trivialmips/nontrivial-mips/>, 2020.
- [3] LoongsonEdu, “la32r-linux,” <https://gitee.com/loongson-edu/la32r-Linux>, 2022.
- [4] SpinalHDL, “Vexriscv,” <https://github.com/SpinalHDL/VexRiscv>, 2018.
- [5] LoongsonEdu, “openla500,” <https://gitee.com/loongson-edu/nscscc-openla500>, 2022.
- [6] Muradil, “流云,” <https://gitee.com/UCAS-Muradil/LiuYun>, 2022.
- [7] M. Wang, “卅,” https://gitee.com/MJ_Wang/spinal-loong-arch-core/tree/master, 2022.
- [8] L. Liang and F. Zhang, “争流,” <https://gitee.com/liangliang678/ZhengLiu>, 2022.
- [9] J. Huang, Y. Yang, T. Yu, and S. Liu, “Llcl-mips,” <https://github.com/huang-jl/LLCL-MIPS/>, 2021.
- [10] A. Seznec, “A 256 kbits 1-tage branch predictor,” *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, vol. 9, pp. 1–6, 2007.