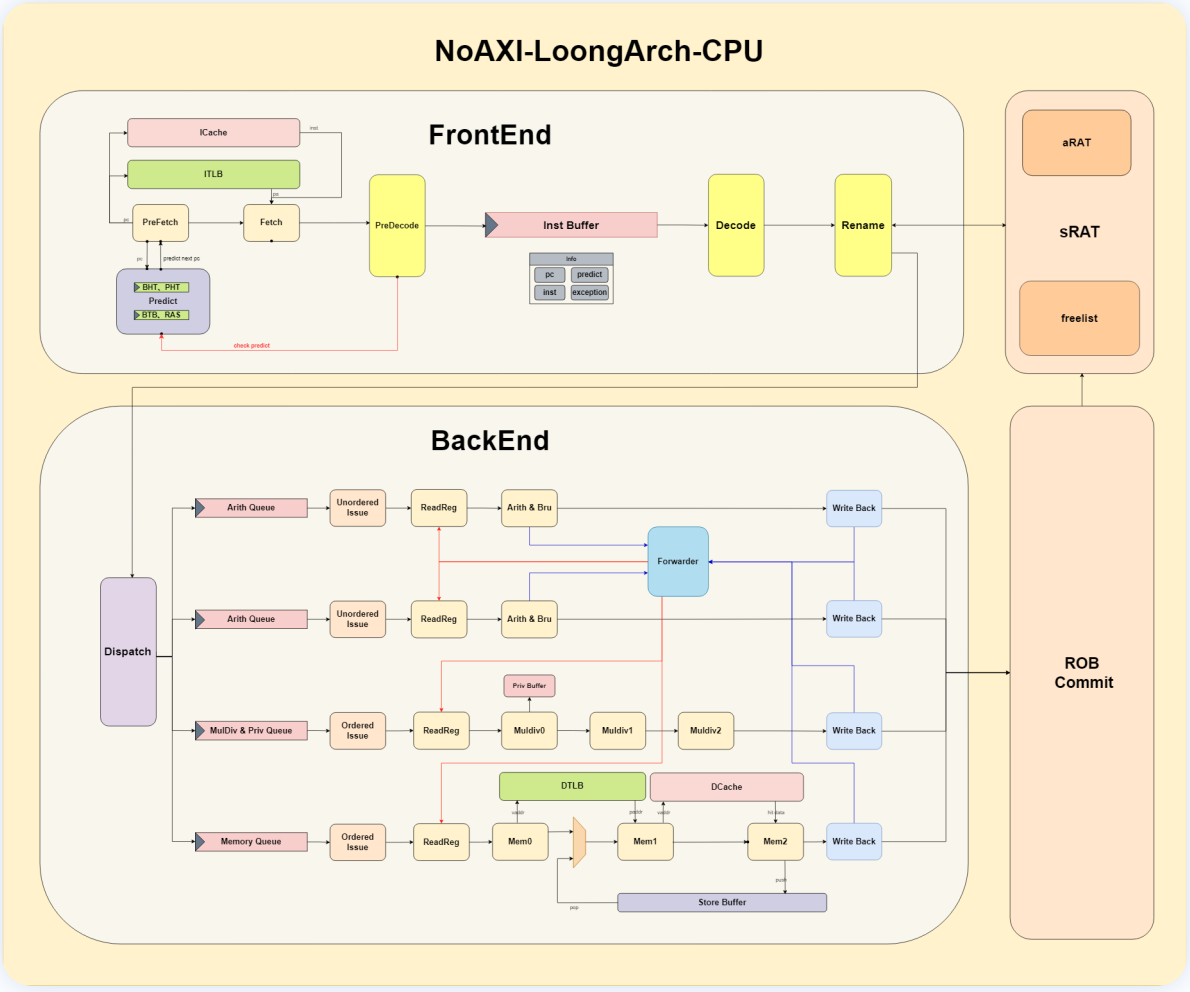
NSCSCC2024 初赛设计报告

杭州电子科技大学 NoAXI队

陈润澎 刘文涛



目 录

1. 概述 3
1.1 项目简介 3
1.2 实现功能 3
1.3 CSR寄存器支持 3
1.4 异常中断支持 4
2. 处理器微架构设计 6
2.1 整体介绍 6
2.2 前端设计 6
2.2.1 预取指 PreFetch 6
2.2.2 取指 Fetch 7
2.2.2.1 分支预测器 BPU 7
2.2.3 预译码 PreDecode 8
2.2.3.1 指令缓冲 Instruction Buffer (IB) 8
2.1.4 译码 Decode 8
2.2.5 重命名 Rename 8
2.3 后端设计 9
2.3.1 分发 Dispatch 9
2.3.2 发射 Issue 10
2.3.3 读寄存器 ReadReg 10
2.3.4 算术流水线 Arithmetic 10
2.3.5 乘除流水线 Muldiv 10
2.3.6 访存流水线 Memory 11
2.3.6.1 Mem0 11
2.3.6.2 Mem1 11
2.3.6.3 Mem2 11
2.3.6.4 写缓存 Store Buffer 11
2.3.7 写回 WriteBack 12
2.3.8 提交 Commit 12
3. 参考文献 13

1. 概述

1.1 项目简介

本项目是第八届"龙芯杯"全国大学生计算机系统能力培养大赛（NSCSCC 2024）的参赛作品。

本项目为基于龙芯架构32位精简版指令集(LA32R)，开发的一款乱序多发射的CPU。项目基于Tomasulo动态调度算法，使用ROB重排序缓存，实现了一个**乱序执行**、**后端四发射**的CPU。此外，我们还实现了分支预测、Cache缓存等诸多特性，使得CPU取得了较好的性能表现。

1.2 实现功能

(前有标记 `*` 的为暂未实现但会在未来实现)

支持指令集：龙芯架构32位精简版中的以下指令

- 算数类指令： `ADD.W` , `SUB.W` , `ADDI.W` , `LU12I.W` , `SLT[U]` , `SLT[U]I` , `PCADDU12I` , `AND` , `OR` , `NOR` , `XOR` , `ANDI` , `ORI` , `XORI` , `MUL.W` , `MULH.W[U]` , `DIV.W[U]` , `MOD.W[U]` , `SLL.W` , `SRL.W` , `SRA.W` , `SLLI.W` , `SRLI.W` , `SRAI.W`
- 跳转指令： `BEQ` , `BNE` , `BLT[U]` , `BGE[U]` , `B` , `BL` , `JIRL`
- 访存指令： `LD.B` , `LD.H` , `LD.W` , `LD.BU` , `LD.HU` , `ST.B` , `ST.H` , `ST.W` , `LL` , `SC`
- CSR相关指令： `CSRRD` , `CSRWR` , `CSRXCHG`
- Cache相关指令： `*CACOP`
- TLB相关指令： `TLBSRCH` , `TLBRD` , `TLBWR` , `TLBFILL` , `INVTLB`
- 其他杂项指令： `RDCNTVL.W` , `RDCNTVH.W` , `RDCNTID` , `SYSCALL` , `BREAK` , `ERTN`

1.3 CSR寄存器支持

地址	名称
0x00	CRMD
0x01	PRMD
0x02	EUEN
0x04	ECFG
0x05	ESTAT
0x06	ERA
0x07	BADV

地址	名称
0x0c	EENTRY
0x10	TLBIDX
0x11	TLBEHI
0x12	TLBELO0
0x13	TLBELO1
0x18	ASID
0x19	PGDL
0x1a	PGDH
0x1b	PGD
0x30	SAVE0
0x31	SAVE1
0x32	SAVE2
0x33	SAVE3
0x40	TID
0x41	TCFG
0x42	TVAL
0x44	TICLR
0x60	*LLBCTL
0x88	TLBRENTRY
0x98	*CTAG
0x180	DMW0
0x181	DMW1

1.4 异常中断支持

Ecode	EsubCode	名称
0x00	0	INT
0x01	0	PIL
0x02	0	PIS
0x03	0	PIF
0x04	0	PME
0x07	0	PPI
0x08	0	ADEF
0x08	1	ADEM
0x09	0	ALE
0x0b	0	SYS

Ecode	EsubCode	名称
0x0c	0	BRK
0x0d	0	INE
0x0e	0	*IPE
0x3f	0	TLBR

2. 处理器微架构设计

2.1 整体介绍

NoAXI处理器采取前后端设计，前端共5个流水级，后端共6个流水级，共11个流水级，最深级数为13。

前端五个流水级，分别为 **预取指**、**取指**、**预译码**、**译码**、**重命名**。

后端分为六个流水级，分别为 **分发**、**发射**、**读寄存器**、**执行**、**写回**、**提交**。

前端采取顺序双发射设计，取指宽度为2，每个周期可以向后端最多传输2条指令。

后端采取乱序四发射设计，流水线条数为4，每个周期可以进行最多两条指令的提交。

2.2 前端设计

前端分为 **预取指**（`PreFetch`）、**取指**（`Fetch`）、**预译码**（`PreDecode`）、**译码**（`Decode`）、**重命名**（`Rename`） 共五级。

其中，预译码级与译码级之间，使用了一个指令缓冲（`Instruction Buffer`）进行解耦。

2.2.1 预取指 PreFetch

在这一级中进行**pc**的更新，并且将虚地址发送给`TLB`。

1. **pc**发送至`TLB`，`TLB`进行翻译
   - 若是直接地址翻译或直接地址映射，直接将得到的物理地址传递给`Fetch`级
   - 若是映射地址翻译，给`TLB`表项发地址，得到命中信息并传递给`Fetch`级
2. **pc**发送至`I-Cache`，基于`VIPT`原理，给`I-Cache`索引
3. **pc**发送至`BPU`，进行分支预测(发送按双字对齐的两个**pc**进行预测，如果第二个**pc**不满足双字对齐则不预测)
   - **pc**发送至`BTB`读取信息
   - **pc**发送至`BHT`得到跳转历史，寄存器锁存供下一拍使用
4. 根据情况更新**pc**
   - (最高优先级)后端传来的跳转请求：可能是例外中断造成的跳转，也可能是特权指令需要冲刷流水线，也可能是分支预测失败的重新取指

- (第二优先级)分支预测传来的跳转请求
- (第三优先级)更新**pc**为按双字对齐的下一个**pc**，例如 1c001000 → 1c001008，1c00110c → 1c001110

2.2.2 取指 Fetch

在这一级中取出两条指令。

1. 在此级中把命中信息发送至 `TLB`，读出命中表项，翻译出**paddr**；并且判断是否存在 `TLB` 相关例外
2. 将得到的**paddr**发送至 `I-Cache`，判定是否命中，若不命中则进入阻塞状态，并与主存利用axi接口交互取出Cache缺失行
3. 取出两条指令，并且保证pc是按双字对齐的，若第二条指令**pc**未满足条件则会标记为无效
4. 得到分支预测的结果
   - 利用上一拍得到的 `BTB` 信息和 `BHT` 中读取到的 `BHR`，得到预测方向和跳转结果
   - 若预测跳转，清刷 `prefetch` 并对其发出跳转请求

2.2.2.1 分支预测器 BPU

在乱序处理器当中，分支预测失败带来的冲刷会极大的影响性能。因此，我们在 `PreFetch` 级和 `Fetch` 级使用分支预测器进行了分支预测。

1. `BHT(Branch History Table)`：保存跳转指令的跳转历史
   - 适用指令：所有跳转指令
   - 用途：利用饱和计数器值预测跳转方向
   - 更新时机：跳转指令 commit 时
2. `PHT(Pattern History Table)`：记录跳转历史对应的饱和计数器值，对应跳转的方向趋势
   - 适用指令：所有跳转指令
   - 用途：利用饱和计数器值预测跳转方向
   - 更新时机：跳转指令 commit 时
3. `BTB(Branch Target Buffer)`：记录**pc**对应的跳转地址，兼顾判断跳转类型(**CALL**,**Return**,其他)的职责
   - 适用指令：条件跳转类和**CALL**型指令
   - 用途：记录指令的跳转类型，是否为跳转指令(`valid` 位)，跳转地址
   - 仅当 `BTB` 命中时才可能发生分支预测
   - 表项组织形式如下：

Valid	Tag	Target	Br_type
1 b	(32 - index) b	32 b	2 b

4. `RAS(Return Address Stack)`：以栈的形式记录**CALL**指令的下一条指令地址
   - 适用指令：间接跳转**Return**指令
   - 用途：预测**Return**指令的跳转地址
   - `RAS`满时，对`RAS`使用循环更新，即再从底向上开始更新，可以尽可能保证预测准确性
5. 分支预测器不会被flush

2.2.3 预译码 PreDecode

在这一级进行分支预测的初次验证，若预测失败，清空前两级流水线并重新取指，最小化分支预测失败的代价。

分支预测在这一级中会处理以下情况：

1. 必定跳转指令但预测不跳转：大概率是初次执行，更正预测结果并且更新分支预测器
2. 非跳转指令但预测跳转：不可能产生这种情况，因为仅当`BTB`命中才会产生预测，非跳转指令`BTB`不可能命中

2.2.3.1 指令缓冲 Instruction Buffer (IB)

在预译码级与译码级之间，我们使用自己设计的`FIFO`，实现了一个指令缓冲，用于解耦取指级与译码级。

1. 该缓冲让取指效率最大化，无论后端是否阻塞都不会浪费前端的取指资源，做到了效率的最大利用；前端只需专注取指并填充至`IB`，而后端只需专注于从`IB`中取出指令并执行
2. 当发生flush时，`Instruction Buffer`直接清空即可

取指前端保证`IB`中的信息均为有效指令，无气泡指令

2.1.4 译码 Decode

本级用于译码，我们通过chisel语言的特性，生成了两个树形译码元件进行译码，并输出指令对应的信息。其中包含了指令所用的相关功能模块类型**funcType**，指令在模块内对应的操作符**opType**，指令对应的**rj**、**rk**、**rd**寄存器编号，以及指令对应的后端流水线编号等信息。

2.2.5 重命名 Rename

本级流水线用于进行寄存器重命名。本级与寄存器映射表RAT与后端的重排序缓存ROB连接，向二者发出请求，接收`RAT`返回的寄存器信息，以及`ROB`返回的对应编号。具体操作如下。

1. `RAT` 分配物理寄存器时，从空闲寄存器表 `freelist` 当中取最多两个空闲寄存器，作为原逻辑寄存器areg对应的物理寄存器preg。
2. `ROB` 通过其 `fifo` 指针分配两条指令对应的 `ROB` 编号。

至此指令开始占用寄存器映射表信息、 `ROB` 表项、物理寄存器资源。

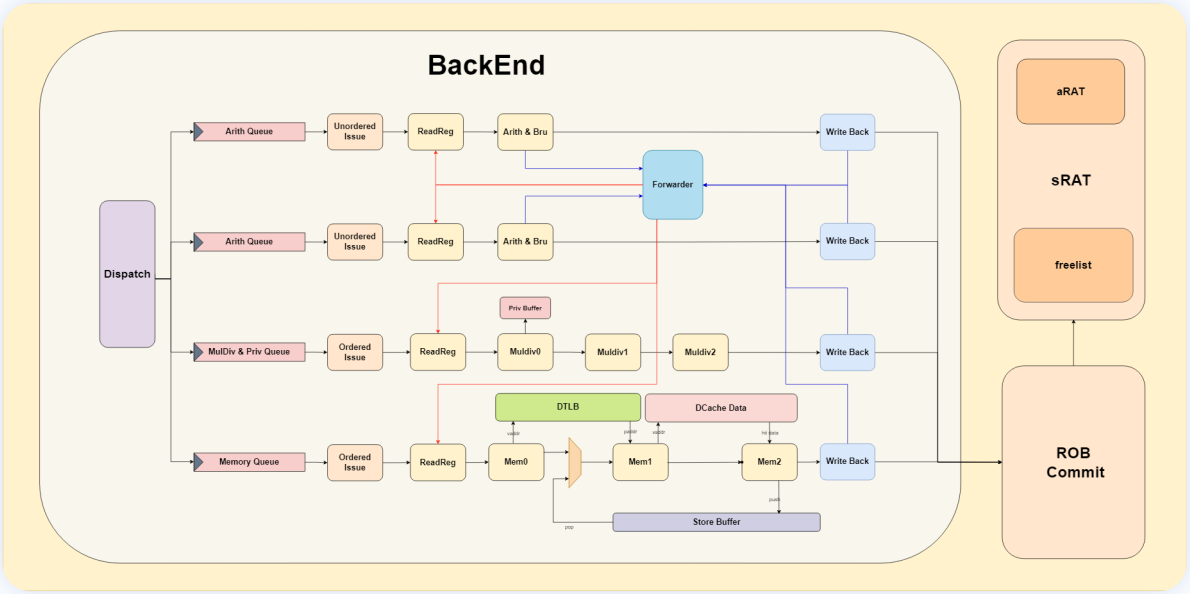
同时，这一级会向发射级发送占用寄存器的请求，以此阻塞发射队列当中的相关指令。

2.3 后端设计

后端分为六个流水级，分别为**分发**（`Dispatch`）、**发射**（`Issue`）、**读寄存器**（`ReadReg`）、**执行**（`Execute`）、**写回**（`WriteBack`）、**提交**（`Commit`）。其中，执行级又根据流水线功能的不同，细分为1~3个流水级。算术流水线只有一级执行级，而乘除和访存有三级执行级。

从发射级开始，后端分为四条独立的流水线，分别是2条**算术**执行流水线，1条**乘除**法执行流水线，1条**访存**执行流水线。

需要注意的是，特权指令放在乘除法。为了保证csr写指令的正确性，在重排序缓存提交指令的时候，才写csr寄存器并冲刷整条流水线。



2.3.1 分发 Dispatch

本级会根据指令的类型，分发指令到各个流水线发射缓存当中。需要注意的是，假如指令涉及的流水线重合，或者需要发射的目标流水线已满，则只会发射满足条件的指令，并会在本级进行阻塞，直至全部指令发射完成。

2.3.2 发射 Issue

这里定义了四个发射队列，对于单个发射队列，每个周期最多接收、发射一条指令。

不同流水线的发射情况如下：

1. 运算流水线，我们已经通过寄存器重命名保证了乱序写回的正确性，因此乱序发射即可。
2. 乘除流水线，由于我们将特权指令分配在了本流水线，因此本流水线需要保证顺序发射。
3. 访存流水线，由于数据缓存只有一个，且需要维护访存数据的一致性，所以必须保证顺序发射。

对于乱序发射的队列，我们使用了压缩队列实现。发射时将检查队列内所有指令的寄存器占用状态，若发现相关寄存器均已被唤醒则可以发射。关于寄存器占用状态的维护，我们会在发出唤醒信号的下一拍更新状态，并额外维护从写缓存中回传的uncached load的对应占用状态，防止冲刷导致相关寄存器的占用状态被解除。

对于乱序发射的队列，会按照指令从旧到新的优先级进行发射，这是因为较旧的指令更有可能拥有较多的相关指令，一次发射能够唤醒更多的指令。对于顺序发射的队列，我们每个周期只会对队列头部的指令进行寄存器占用情况的检查，只会发射队列中的最旧指令。

2.3.3 读寄存器 ReadReg

这一级流水线在每条流水线当中都独立存在，负责从物理寄存器当中读出源操作数对应的数值。寄存器堆对于每一条流水线都分配了一个读接口。同时这一级也会接收执行级和写回级前递的信息，用于实现指令的背靠背执行。

如果是算术流水线的读寄存器阶段，还会进行相关指令的唤醒。

2.3.4 算术流水线 Arithmetic

算术流水线共有2条

只有一级，用于执行算术指令运算

2.3.5 乘除流水线 Muldiv

用于乘除指令与特权指令的执行，分为三级。拆分成多个流水级的主要原因是需要实现乘法运算的流水化。

对于乘法运算，我们使用xilinx ip搭载了一个三级流水的乘法器，实现了乘法运算的流水化。

对于除法运算，我们使用xilinx ip实现了除法器，并在乘除流水线的第一级中进行阻塞执行。

对于特权指令，我们会将写指令放入特权缓存 `PrivBuffer` 当中，并阻塞后续所有csr指令，直到后端ROB进行提交的时候，才进行csr寄存器的写入。

2.3.6 访存流水线 Memory

访存流水线是后端流水线当中最重要的一条，也是设计难度最高的一条。对于访存指令，某些访存指令会对处理器状态进行修改，因为其对处理器状态产生的影响不好撤销，所以当执行这些指令时，使用 `Store Buffer` (写缓存)对这种类型的访存指令进行暂存，只有当提交时，才会“回滚”至访存流水线真正执行

2.3.6.1 Mem0

计算**vaddr**，并发送至 `TLB` ， `TLB` 进行翻译

- 若是直接地址翻译或直接地址映射，直接将得到的物理地址传递给 `Mem1` 级
- 若是映射地址翻译，给 `TLB` 表项发地址，得到命中信息并传递给 `Mem1` 级

2.3.6.2 Mem1

在这一级中会选择接受是回滚访存指令还是 `Mem0` 流过来的指令，优先选择回滚访存指令

1. 在此级中把命中信息发送至 `TLB` ，读出命中表项，翻译出**paddr**；并且判断是否存在 `TLB` 相关例外
2. **vaddr**发送至 `D-Cache` ，索引 `D-Cache` 的 `Tag-sram` 和 `Data-sram`

2.3.6.3 Mem2

1. 将**paddr**发送至 `D-Cache` ，判定是否命中，若不命中则进入阻塞状态，并与主存利用axi接口交互取出Cache缺失行，若命中则执行**可执行**的访存指令
2. **可执行**访存指令：cached load指令，回滚访存指令
3. **不可执行**访存指令：uncached load指令、store指令。对于这些指令，将其push进 `Store Buffer`
4. 对于cached load指令，我们将从 `D-Cache` 中读出的数据传递下去，在写回级进行前递判断

2.3.6.4 写缓存 Store Buffer

在乱序处理器当中，为了保证处于推测态的访存指令不会对外部状态进行更改，我们使用了写缓存 `Store Buffer` 来维护。

1. 当接收到 `ROB` 的提交后，会将写缓存内的信息通过 `Mem1` 级重新回流到访存流水线当中。
2. 由于 `Mem1` 需要同时接受来自Mem0的数据和来自写缓存的数据，当 `Mem0` 向 `Mem1` 发送数据时，还会与写缓存发送的数据进行竞争，优先让写缓存内数据进行访存。
3. 当发生flush时， `Store Buffer` 直接清空即可

2.3.7 写回 WriteBack

这一级用于写物理寄存器，更新rob当中对应的提交信息，并更新发射队列当中的寄存器占用状态。

在我们的初版设计当中，为了时序考虑，直到这一级才会对访存流水线相关的指令进行串行的唤醒。

2.3.8 提交 Commit

这一级用于进行最终的提交，解除对于CPU资源的占用，并解除寄存器的推测态。

1. 解除对于上一个物理寄存器的占用（opreg），并将opreg对应的寄存器编号插入到空闲寄存器列表freelist
2. 更新aRAT的映射关系，aRAT当中存储的是由已经提交的指令构成的、不涉及推测态指令的寄存器映射表
3. 对于包括分支、访存、特权在内的部分非算术指令，由于分支预测器更新限制、写缓存入队限制等原因，我们在检测到这些指令的时候，不会进行双指令的提交。假如检测到分支预测失败或例外，还会向冲刷控制器发出冲刷请求。
4. 当发生流水线清空的时候，首先向冲刷控制器发出请求。冲刷控制器会延迟一拍后，对于各个流水线及功能部件发出冲刷信号，并令重命名相关部件进行状态恢复。具体而言，状态恢复时，会将aRAT赋值给cRAT，清空rob当中的所有表项，并将freelist的尾指针置为头指针位置。

3. 参考文献

[1]J. Ye, and L. Xi, *PUA-MIPS* , https://github.com/Clo91eaf/PUA-MIPS, 2023

[2] Z. Ma, *LA32R-pipeline-scala* , https://github.com/MaZirui2001/LA32R-pipeline-scala, 2023

[3] H. Gao, and M. Liu, *NOP-Core* , https://github.com/NOP-Processor/NOP-Core, 2023

[4] Y. Zhou, S. Chen, X. Liu and J. Chen, *Nontrivial-mips* ,https://github.com/trivialmips/nontrivial-mips, 2019

[5] W. Wang, and J. Xing, *CPU Design and Practice* , Beijing: China Machine Press, 2021

[6] Y. Yao, *SuperScalar RISC Processor Design* , Beijing: Tsinghua University Press, 2014