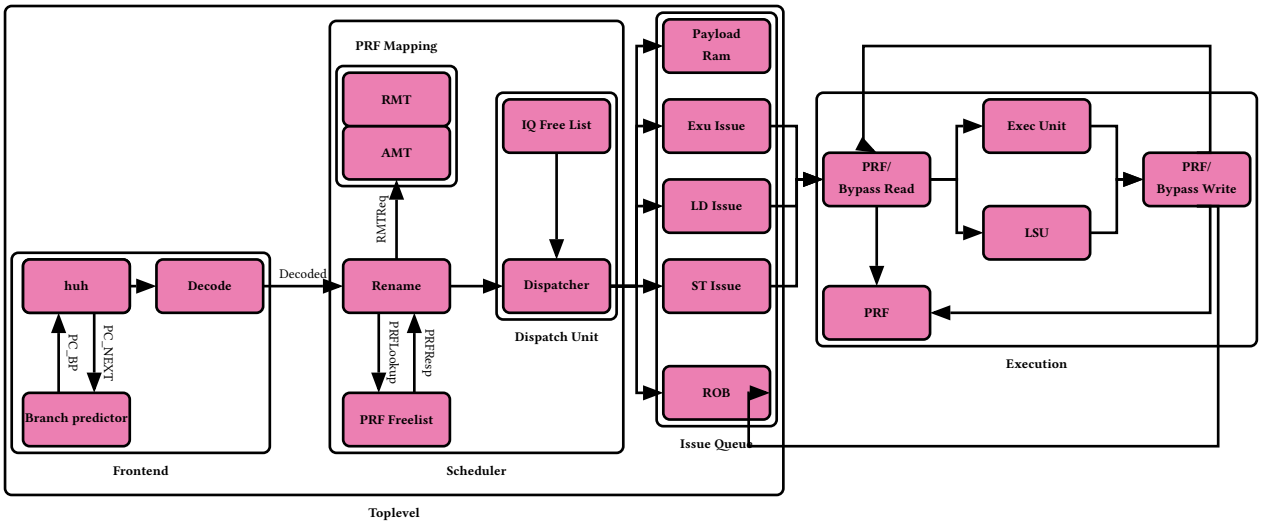
嵌入式荣誉课

第四组

目录

1 整体架构介绍



本处理器为乱序多发射架构，支持 RV32IM 指令集以及 Zicsr 指令拓展集。流水线深度为 8 级以上：取指单元 3 级、译码 1 级、重命名单元 1 级、发射单元 2 级（包括 dispatch）、执行单元若干级，以及写回 1 级。

该处理器取指、译码以及重命名宽度均为 CORE_WIDTH。同样，ROB 分为 CORE_WIDTH 个 bank，派遣单元一周期载入 CORE_WIDTH 条指令，因此退休宽度也是 CORE_WIDTH。

指令通过派遣单元鉴别指令类型后，便会分别发送到三个发射单元之一。若指令组包含 csr 指令，则阻塞流水线，直至 ROB 清空后，**顺序**发射若干个 CSR 指令，发射完后返回乱序执行模式。

指令在写回后会更新发射队列的信息，而由于发射队列与 FU 之间相隔一层寄存器，因此寄存器数据读取和写回是错开的。加之本处理器不考虑执行单元对 PRF 的读写端口，因此整体设计无需数据反馈通路。

转跳指令经 BU 处理后会马上更新处理器的状态。此时，若指令显示错误预测，则整个流水线都会被冲刷，除了当前没有被 branch mask 掩盖的那些指令。由于转跳指令是乱序执行的，所以必须有一个 branch mask 单元判断转跳指令的年龄。若指令被确定是错误判断，且不是最年长的指令，则 branch mask 会将其保留到内部的寄存器。直到下一个转跳指令被执行后，才会被重新检查。只有最年长的 branch mask 执行完毕后才能确定转跳信息（如转跳地址以及 GHR）

2 前端介绍

2.1 取指 (IF)

取指单元在单周期内读取 CORE_WIDTH 个指令，随后在检测指令组是否包含跳转指令并与分支预测结果（BP）进行比较，进而确定 PC_Next，同时将 CORE_WIDTH 个指令发送到译码单元。这 CORE_WIDTH 个指令将保持程序原有的执行次序，并且随带着 valid 位表示是否为有效指令。valid 位主要由 PC 地址低有效位中非对齐部分和分支预测结果决定。同时，取指单元会从 branch_mask 获得当前分支掩码，并载入微指令，用于后续回滚时清除指令。

PC_Next 对于存储器来说永远是（按 CORE_WIDTH 条指令）对齐的（把低有效位置零即可），而 PC_Next 本身不一定是对齐的，因此在取指后，需要将 PC(非对齐)之前的指令作废（vaild = 0)。分支预测单元(BP)也将基于 PC_Next 判断指令组那些指令为有效，并结合预测结果，发送转跳信息给取指单元。

PC_Next 主要来自：branch_mask 单元、分支预测(BP)、$\mathrm{PC}(\text{对齐}) + \mathrm{CORE\_WIDTH} \times 4$，其中优先级由高到低排列。当发生分支预测错误时，branch_mask 单元会发送 bu_commit 信号，而该信号包含正确转调地址。随后，取指单元将跟新 pc_reg 并进入预取指模式，等待指令到达，否则按照 BP 命中与否来决定 PC_Next 是 PC_BP 还是 $\mathrm{PC}(\text{对齐}) + \mathrm{CORE\_WIDTH} \times 4$。

在处理 BTB 命中时，本处理器中的 IF 模块只接收第一条命中的指令，并用 inst_mask_btb 屏蔽后续指令。因为考虑即使当前 IF 返回了多条指令，但若第一条是跳转指令，后续指令都不会起效。在这一阶段，分支预测器的状态（GHR、是否跳转等）会被存入寄存器，并确保与 memory 对齐时序。在取指后，该 pc 前的指令会被屏蔽，避免误处理前一个 fetch packet 的旧指令。再然后，IF 模块会对所取指令的 opcode 进行初步处理，判断其是否为分支指令（bne,beq,etc..）或是 jal 与 jalr。当所取指令是上述类别时，将跳转预测结果更新至 GHR 中，并将 valid 指示调整为 1。

仅当当前 fetch packet 中所有有效的分支指令都能从 mask freelist 中申请到分支资源时，开始构建 uop 向量。IF 模块通过遍历 fetch packet 中的每条指令，生成对应的 uop，包含指令及其地址、分支预测结果，以及 Branch mask 的相关信息。在构建时，uop 向量内的指令、指令地址取决于当前的 PC 地址。分支预测信息用 BP 模块提供的信息填充。Uop 的有效性决定于指令本身的合法性、BTB 是否屏蔽此条指令、core 是否正在运行，以及（若是）分支指令是否分配了分支 ID。

随后，IF 模块与 Branch mask 模块配合，标记指令的控制流历史，以确保 flush 的时候不会出错。Branch mask 模块会通知 IF 是否有 Branch mask 被释放，如有则把信息写入到指令 uop 中，这样做确保了当前指令不会重用提前被释放的 branch id 直到当前指令退休。

最终，IF 将整个 uop 向量存入寄存器传给 ID 阶段进行后续译码操作，并保持在运行状态，等待下一组指令的处理。

2.2 分支预测 (BP)

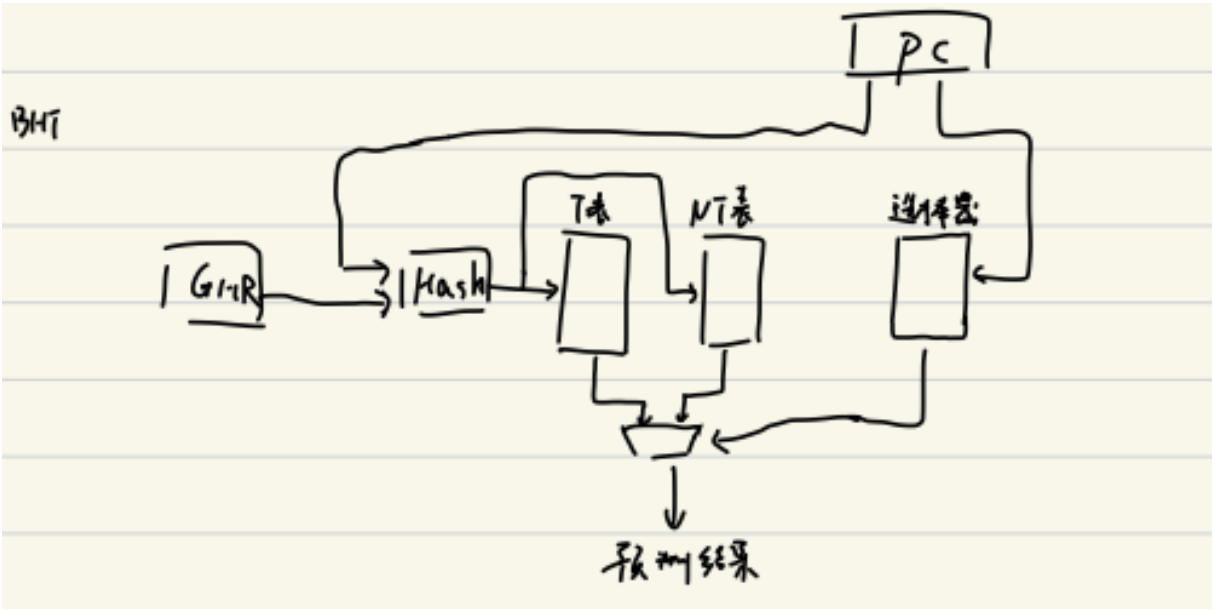


Figure 1: BHT 结构：采用分开的 T/NT 策略，以 PC 部分低位作为选择器索引

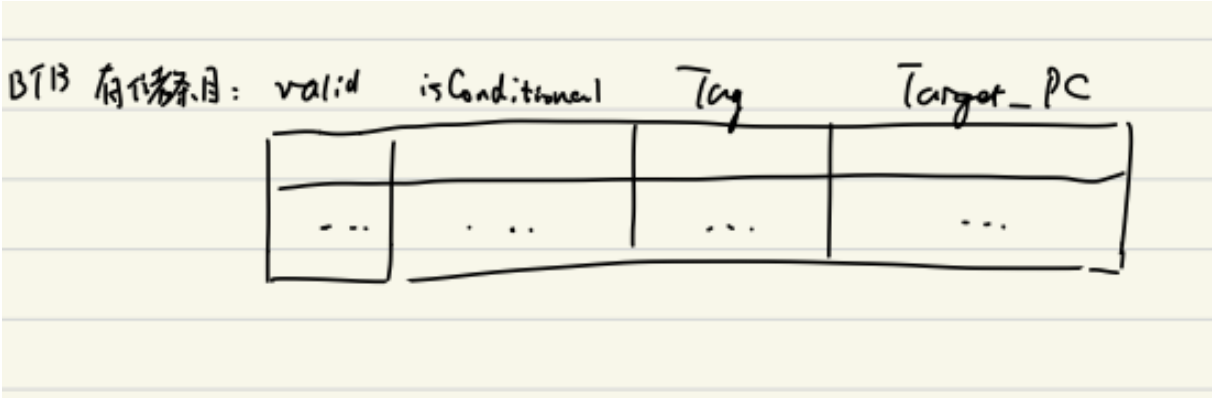


Figure 2: BTB 条目

在本次处理器设计中，Branch Predictor（BP）模块负责预测分支指令的行为，以提升指令流水线的效率。BP 模块由多个子模块构成，包括 Branch History Table（BHT）、Branch Target Buffer（BTB）、Global History Register（GHR）以及与 BranchMask 单元的接口。

2.2.1 Global History Register（GHR）

GHR 是一个移位寄存器，用于记录最近分支指令的历史结果（taken 或 not taken）。在多发射处理器中，GHR 的更新粒度基于 fetch packet 的大小，即每次更新对应一组 p.CORE_WIDTH 宽度的指令。这种设计简化了 GHR 的更新逻辑，使其能够一次性处理多个指令的历史记录。

更新机制：当 BranchMask 模块发出 should_commit 信号时，GHR 会根据提供的分支结果进行更新，记录当前 fetch packet 的分支历史。

回滚机制：在分支预测错误时，BranchMask 模块提供之前的 GHR 快照，BP 模块将 GHR 回滚至该快照状态，以恢复正确的预测上下文。

2.2.2 Branch History Table（BHT）

BHT 用于存储分支指令的历史行为模式，包含两个表：T_table（taken 表）和 NT_table（not taken 表），以及一个选择器表 choice_table，用于在两者间选择。

表结构：T_table 和 NT_table 采用饱和计数器设计，记录分支指令的 taken 或 not taken 倾向。choice_table 则通过计数器决定预测时使用哪个表。

预测逻辑：在取指阶段，BP 模块根据当前 PC 和 GHR 的状态，从 BHT 中读取预测结果，决定是否跳转。

2.2.3 Branch Target Buffer（BTB）

BTB 是一个高速缓存，用于存储分支指令的目标地址，提升分支预测的准确性和速度。

条目结构：每个 BTB 条目包含有效位、目标地址、是否为条件分支以及 PC 的高位标签。

命中逻辑：在取指时，BP 模块检查 BTB 中是否存在与当前 PC 匹配的条目。若命中且预测为 taken，则使用 BTB 中的目标地址作为下一个 PC。

2.2.4 与 BranchMask 单元的接口

BranchMask 单元负责管理分支指令的掩码，并通过 should_commit 信号触发 BP 模块的更新和恢复操作。

更新信号：当 BranchMask 发出 should_commit 信号时，BP 模块更新 GHR、BHT 和 BTB，确保预测器与实际执行结果一致。

回滚机制：在预测错误时，BranchMask 提供 GHR 快照，BP 模块据此回滚 GHR 并通知流水线冲刷错误指令。

2.2.5 多发射支持

BP 模块支持多发射处理器，能够同时为 fetch packet 中的多个指令进行预测。

并行预测：每个时钟周期，BP 模块检查 fetch packet 中每条指令的 PC 是否命中 BTB，并根据 BHT 结果预测是否跳转。

优先级逻辑：若多个分支指令命中 BTB，BP 模块选择第一个预测为 taken 的分支，并使用其目标地址作为下一个 PC。

2.2.6 更新逻辑

BP 模块在分支指令执行后更新预测器，确保其准确性。

BHT 更新：根据分支指令的实际结果（taken 或 not taken），更新 T_table 或 NT_table 中的饱和计数器，并调整 choice_table 的选择倾向。

BTB 更新：若分支指令为新指令或目标地址发生变化，更新 BTB 条目，包括目标地址和条件分支标记。

2.2.7 恢复机制

分支预测错误时，BP 模块通过以下步骤快速恢复：

GHR 回滚：利用 BranchMask 提供的快照 bu_signal 信号将 GHR 恢复至错误预测前的状态。

流水线冲刷：通知流水线冲刷错误的指令，并从正确的 PC 重新取指。

2.3 译码 (ID)

译码单元从取指单元接受 CORE_WIDTH 个指令，并对有效的指令进行译码操作，最后发送到重命名单元。若整体无效，则不发射指令组。

ID 模块把指令分为了 MUL,DIV,ALU,LD,ST,Branch 等类别。根据指令的操作码 opcode 解析出指令的类型，从而确定操作数在指令中所在的位置，随后解析出指令操作数的类别填入 uop 传给 rename 进行后续操作。对于立即数，本模块不做处理，而是将原本指令截断 opcode 后，传送到后级，由后级自行解构出。

在模块中，引入了对 nop 指令的识别，即“操作类型是 ALU，且源寄存器 rs1、rs2、目标寄存器 rd 均为 x0 的指令”。一旦识别出指令为 nop 指令，就将其屏蔽，这条指令将不会被发射到 Rename 阶段，减少计算资源的消耗。

译码单元不会造成流水线堵塞，因此只向前传递后级的阻塞信号。

3 后端介绍

3.1 重命名 (RU)

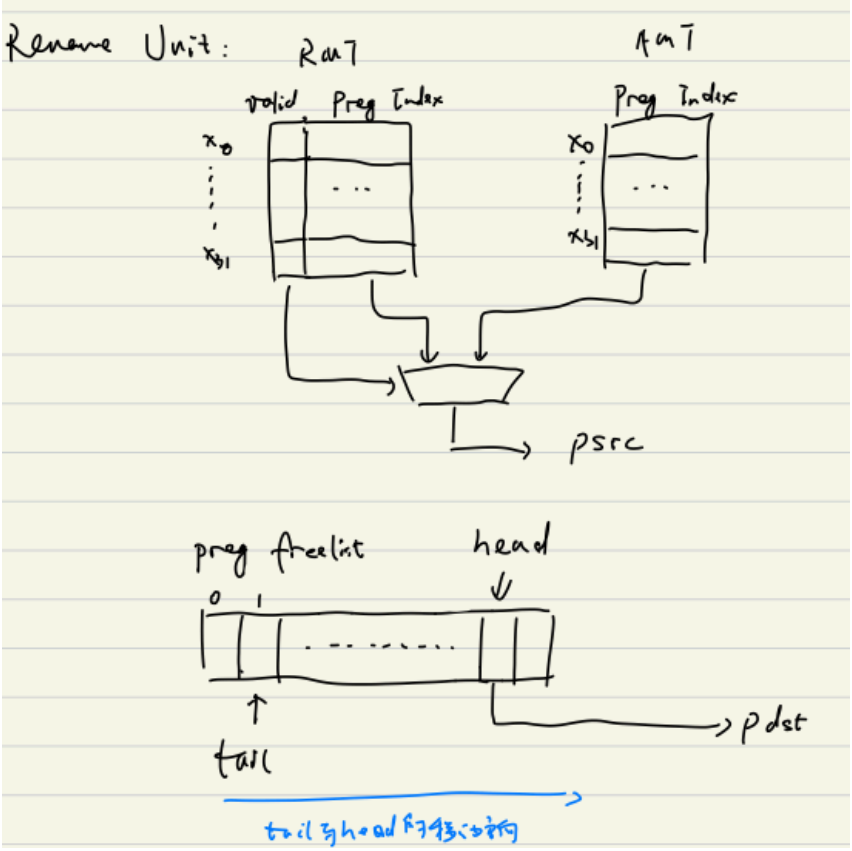


Figure 3: 重命名架构

从译码单元接受 CORE_WIDTH 个指令，执行时必须确保 CORE_WIDTH 个指令能够同时被重命名（不包括无效指令），否则阻塞流水线。

对指令源寄存器重命名时分别从 RMT 和 AMT 获得对应的映射关系。其中 RMT 优先级高于 AMT，亦即当 RMT 中的映射关系被记为有效时，应当选取 RMT，否则 AMT 取胜。RMT 中的有效位只有在回滚时被清零，目的是为了减少冗余的复制操作。

为解决同时对指令组中一种寄存器命名的问题，本模块使用了映射关系矩阵(XLEN * CORE_WIDTH)。对于有写入寄存器的指令，将会在相应的以该架构寄存器地址为索引的位置上将其在指令组中对应的位上置 1。其后的指令若对该指令有依赖，则将从该矩阵相应行中利用有优先编码器寻找最后写入的物理寄存器地址，从而保持了正确的映射关系。

对于目的寄存器，需从 PRF Freelist 获取 CORE_WIDTH 个空闲物理寄存器，获取成功后将 index 写入 RMT，否则 stall。

3.2 PRF Busy

在重命名单元检测到有指令写入寄存器, 会把对应物理寄存器地址写入PRF_BUSY 表进行登记。该表的用途是记录当前是否仍有指令需要写入指定物理寄存器, 主要目的是为了判断指令进入发射队列时，源寄存器就绪的初始状态。如果指令已完成对该寄存器的写入，则后续无需监听该源寄存器的状态。

3.3 派遣 (DP)

从重命名单元接受 CORE_WIDTH 个指令，执行时必须确保 CORE_WIDTH 个指令能够同时被派遣，否则阻塞流水线。

此单元将根据 uop 的类型，将指令分别派遣到 exu_issue, ld_issue, st_issue 等指令发射队列。

当检测到 CSR 指令时，本模块将进入顺序发射模式，并暂停流水线。进入顺序模式之前，将所有乱序指令发送到发射队列中，并将该 CSR 指令及以后的指令屏蔽掉，再把指令组送入 ROB。进入顺序模式之后，等待 ROB 清空了才将眼前的 CSR 指令逐一发送到 EXU（可连续发送），此时不把指令写入 ROB。若 CSR 指令执行完毕后仍有为派遣的乱序指令，则返回乱序模式。较原先乱序模式不同的地方在于此时的掩码屏蔽掉了之前的指令，直到掩码为零时，才释放流水线。

3.4 指令发射 (IQ)

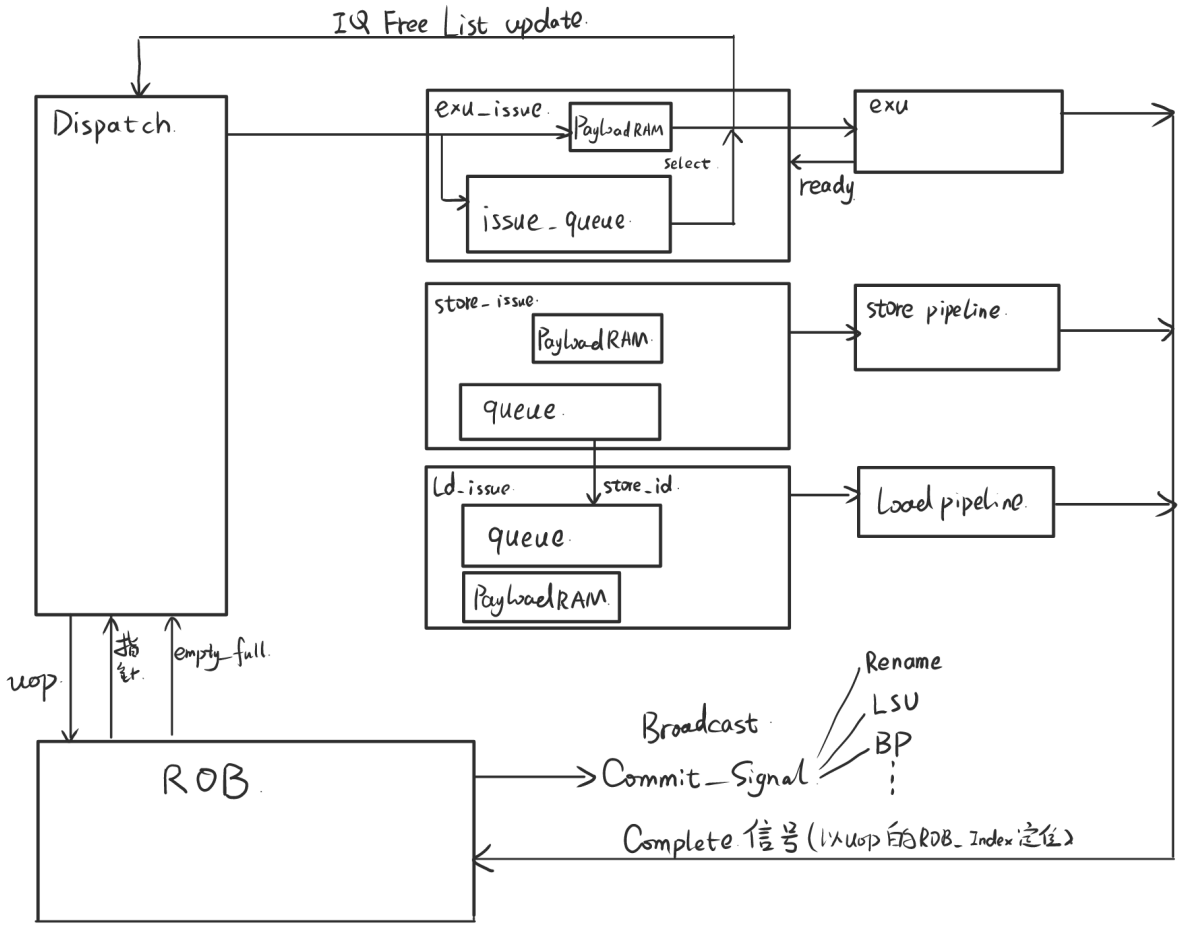


Figure 4: Issue Queue 工作原理图

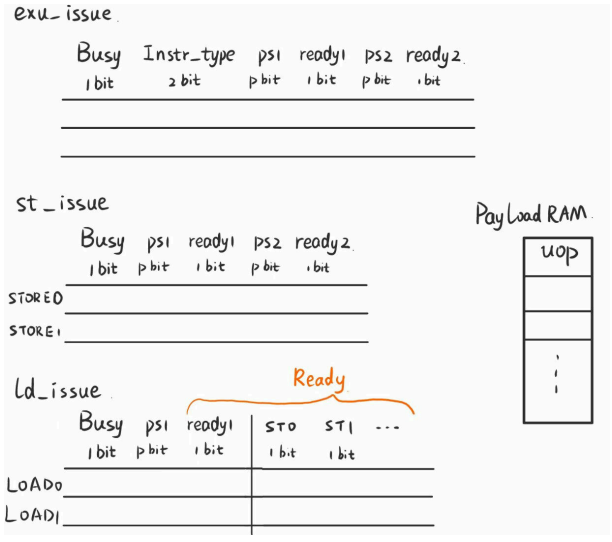


Figure 5: Issue Queue 条目

Issue queue 用于向各个 EXU 发射命令，需要监听后续 EXU 是否空闲以及各操作数的就绪状态。操作数设置为就绪有三种途径：

1. 监听 PRF 中各物理寄存器的 Valid 信号
2. 监听 EXU 处的 ready 信号

3. 监听 EXU 后的级间寄存器的 ready 信号

在本 CPU 中，IQ 分为三部分，分别是 **exu_issue**, **ld_issue**, **st_issue**，其中后面二者间有数据传输，可见后续说明。

exu_issue 的运行过程如下，在非满时接收 dispatch 模块的 uop 存入 payloadRAM 中，将有用信息（instrType、psrc）存入 queue 中用于判断就绪状态，等待发射，每周期根据条目就绪状态选择已就绪的指令发射给 FU（内含两个 ALU、一个 BU、一个乘法器、一个除法器），同时向 dispatch 模块发送本周期发出的指令的 IQ 地址，用以更新 IQ Free List（位于 dispatch 单元）。

st_issue 基本与 **exu_issue** 相同，向 store 流水线（位于 LSU 中）发送 uop，不同的是要向 **ld_issue** 发送每周期仍未发射的条目信息。**ld_issue** 需要从 **st_issue** 接收信息的目的在于解决 load-store 违例。

在本CPU中, store指令走完store流水线的周期固定为2, 而load指令在load流水线第二级才会从stq获取前馈数据(见Figure 6), 所以只需要保证在程序顺序上位于该load指令之前的store指令全部进入STU后再发射load, 就可以避免load指令越过有数据依赖的store指令, 错误地读取数据(store – load violation)。为此, 需要一个存储依赖关系的矩阵, load指令在进入**ld_issue**时, 就要接收当前周期 **st_issue** 中尚未发射的 store 条目，以二进制数(位宽为 ST_ISSUE_QUEUE_DEPTH)的形式存入矩阵的一行中。每当 **st_issue** 发射一条 store 指令，就要更新 **ld_issue** 矩阵中对应的列，将其全部置为 0 表示就绪。当一条 load 命令所在行全为 0 时，表示与其相关的（比该 load 指令更年轻的）store 指令已经全部发射，该 load 指令解除限制（又若 load 指令的 psrc 为 ready 状态，则该 load 指令具备发射条件）。

3.5 重排序 (ROB)

Reorder Buffer(ROB)，是一个 FIFO 结构，用于按照程序序存储指令，可以实现顺序 retire，从而实现分支预测失误后的回滚与精确异常（本项目无需实现精确异常）

在实际工作时，ROB 从 dispatch 接收 uop 并储存，把自身头尾指针以及 empty/full 标志传递给 dispatch 单元。从各个 EXU 接收指令完成信号，更新 complete 状态。ROB 总是广播位于头部的两条指令，用额外的 valid 位（两条指令各有一个）来指示本周期是否 retire 该指令，用于 AMT 的更新、通知 STQ 可以写入内存、通知其他模块发生分支预测错误。

3.6 功能单元 (FU)

FU 为 ALU, BU, MUL, DIV, CSR 的抽象类，主要提供基本的输出入输出端口。指定单元（BU, CSR 等）有额外读写端口专门与 ROB 交互。

3.6.1 ALU

ALUFU 核心计算模块 ALU 支持 12 种 RISC-V 标准算术逻辑运算。计算单元采用全组合逻辑设计，关键路径优化为： 操作数选择器：基于 OprSel 枚举的多路选择器，支持 4 种操作数来源 32 位加法器：支持 SUB 操作的补码转换 译码阶段：根据 fu_signals.opr1_sel 区分 LUI/AUIPC 特殊指令，对 I-type 指令提取 instr[7:5]生成 func3 信号（因为接收到的 instr 是截去后 7 位 opcode 后的 25 位 instr）

3.6.2 BU

BranchFU 处理控制流指令，包括条件分支和无条件跳转，通过 instr_type 来区分 采用双校验点设计：方向预测校验和目标地址校验来检验是否预测成功，预测失败时设置 mispred 信号作为错误标记并冻结后续指令提交，并产生 flush 信号 内部实现减法器，检测分支跳转预测结果是否正确。通过额外的端口将跳转信息写入 branch_mask 模块。

3.6.2.1 Branch mask

由于 BU 发送指令跳转指令是乱序的，因此需要判断跳转指令的优先次序。该模块通过比较 BU 写回指令的 branch_mask 来进行年龄先后判断。

Branch mask 单元内含转跳指令信息的缓存，用于保存 BU 写回指令的信息（实际转跳地址，branch mask, 错误预测标志等等）。BU 写回有效时（不论转跳与否）将触发该缓存的更新逻辑，以及到相应模块的信息广播。更新该缓存只有在写回指令比缓存中的指令更老时才有效。

每当 BU 写回指令表示分支判断错误时，branch_mask 就会向所有的模块广播冲刷信号（通过 ROB），对于前端将进行完整冲刷，而后端（执行端和发射队列）则提供对应 branch mask 的冲刷位，如若执行中的指令 branch mask 有效位包含该冲刷位，则该指令被作废。

3.6.3 MULFU

MUL 部分采用基 4（Radix-4）Booth 编码算法进行多周期乘法，并支持 4 种乘法运算。由于 32 位的 Booth 乘法器乘数与被乘数均为有符号数，因此为了同时适配不同的乘法运算，本处理器中均对两者进行位数扩展。比如，MUL 指令将被乘数进行有符号扩展成 64 位，而乘数则有符号扩展成 33 位（不包括运算中向左位移 1 位），而 MULHSU 对乘数进行无符号扩展。

3.6.4 DIVFU

DIV_REM 部分采用非恢复式移位减法除法算法，支持 2 种除法和 2 种取余运算，并遵守 RISCV spec 中对除数分别为 0 和-1 的情况。

3.6.5 CSR

由于写入 CSR 寄存器时可能更改处理器的状态，所以我们按照传统的做法：ID 在检测到 CSRRW 指令后，进入顺序发射模式，直待 ROB 清空后，CSR 才开始接受 CSR 指令。

CSR 寄存器通过 Memory mapping 把地址映射到特定的寄存器（可以是外设寄存器，或是 io 端口）。本处理器只实现简单的 mcycle 寄存器，实现方法是额外设立一个计数器，在 csr 读写时，使能读写端口。

3.7 LSU

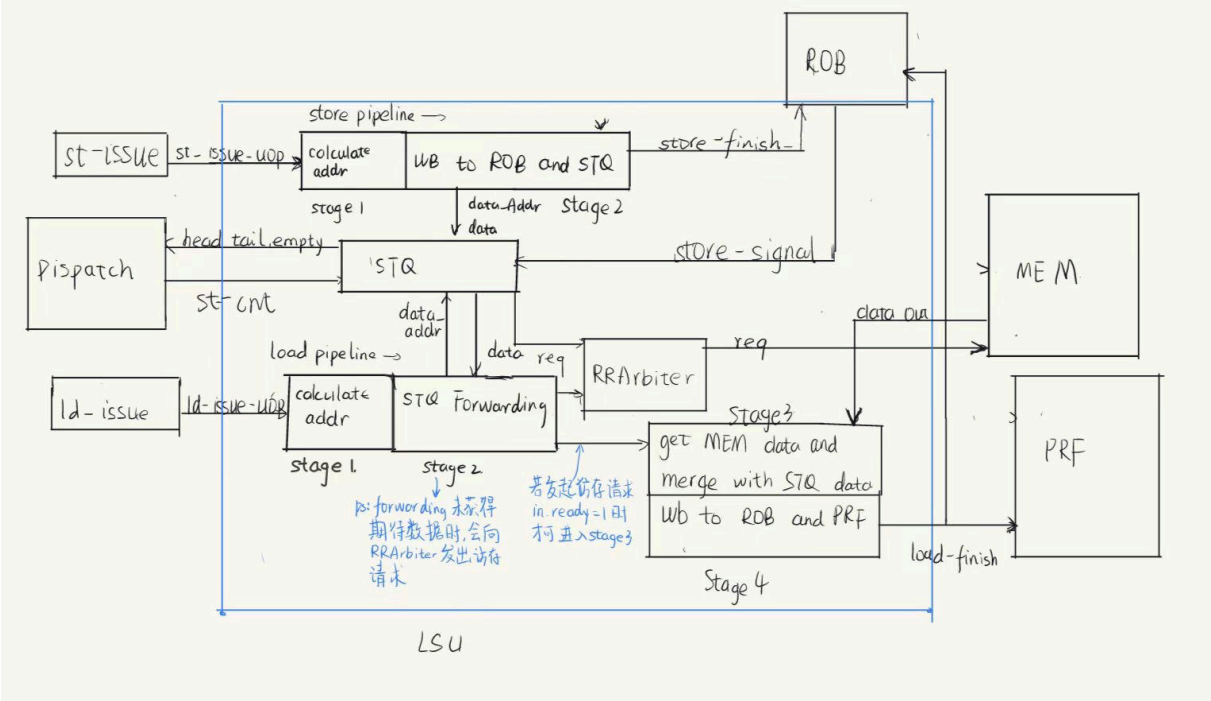


Figure 6: LSU 工作原理图

在本次处理器设计中，Load-Store Unit（LSU）由 4 个子模块，Store Queue（STQ）、LoadPipeline、StorePipeline 以及一个面向 memory 的请求仲裁器（Arbiter）构成。通过子模块的相互协作，LSU 完成数据写入的暂存、加载过程中的数据选择与拼接、以及最终的 memory 请求发起，构建出一个支持乱序指令执行的访存后端。

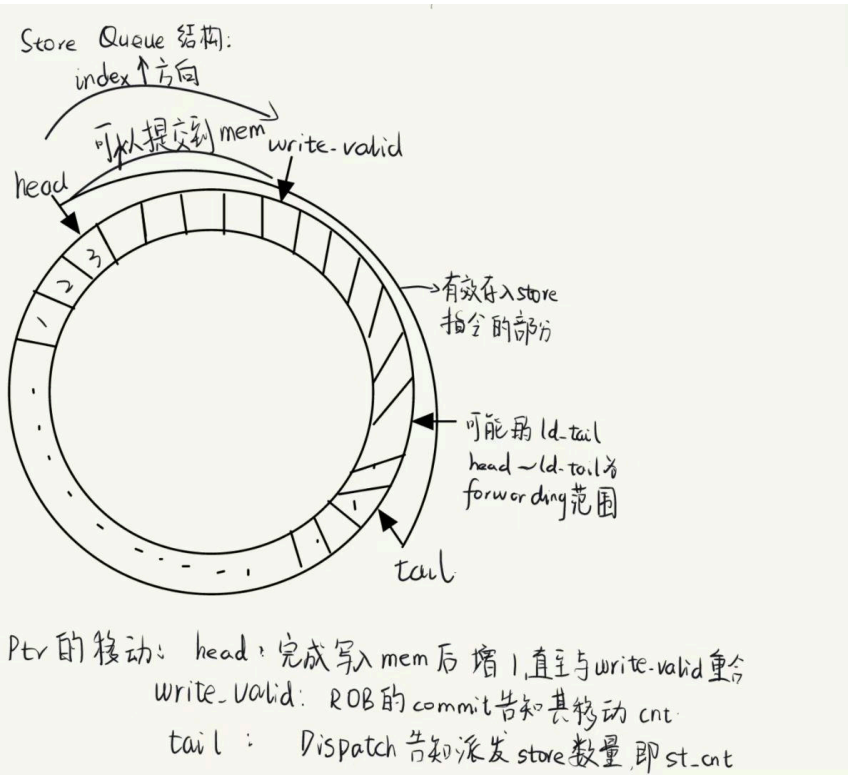


Figure 7: STQ 工作原理图

STQ 是 LSU 的核心缓冲结构，由三个指针(head,write_valid,tail)维护的环形结构，以 STQEntry 的形式保存尚未提交的 store 指令。每个 entry 记录目标地址、写入数据，有效字节掩码以及指令类型（func3），支持 SB，SH，SW 存储操作的按字节精度控制。Store 指令向 mem 的提交通过 head 和 write_valid 指针进行控制，write_valid 的移动依赖于 ROB 的 commit 信号。只要当 head 和 write_valid 不重合时，STQ 会持续向仲裁器发起访存申请，当仲裁器接收后，head 指向的 store 请求会存入 mem，head 后移一位

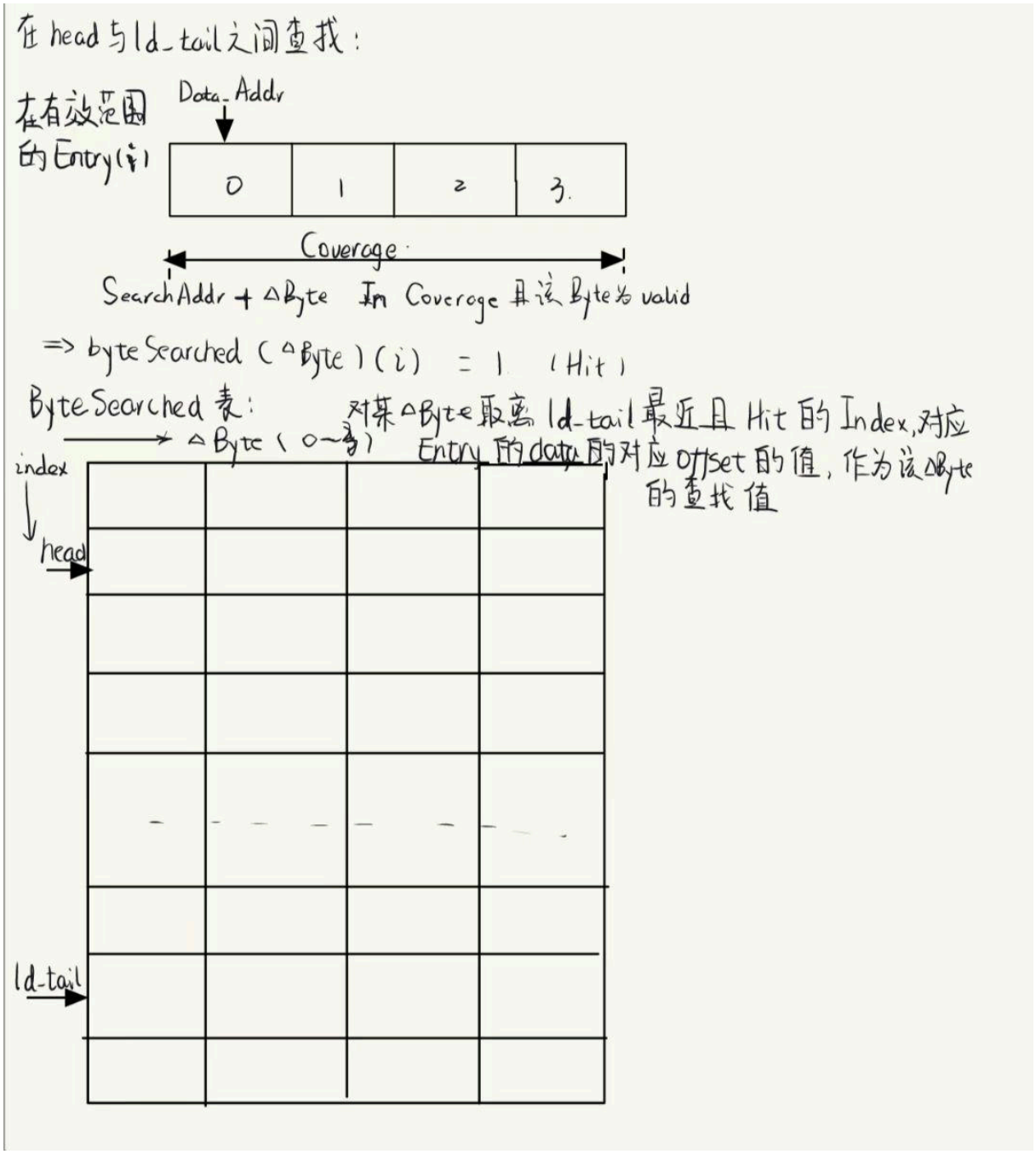


Figure 8: STQ 索引操作

STQ 不仅作为写入缓冲，还承担着 Load 执行过程中的数据前向源这一角色。系统会在 STQ 中根据 LoadPipeline 传入的范围（head-ld_tail）、目标地址和 Load 指令的 func3，将一个 load（最多 4byte）拆成每 byte 单独处理，对于每一个 byte，计算其实际地址，在相应的范围内查找是否存在

地址匹配且尚未提交的 Store 条目覆盖了该 byte。搜索完所有的 byte 后，记录每个 byte 是否成功找到(bytevalid)，并完成按 byte 拼接。在拼接过程中优先使用传入 ld_tail 指针最近的匹配 entry 中匹配该 byte 的数据。最终将 bitvalid（由 bytevalid 生成）和拼接的最终数据传回给 LoadPipeline。

LoadPipeline 是一个四级流水线，依次完成地址计算、STQForwarding 和 memory 请求发起、数据合并以及最后的写回操作。在完成地址计算后，Load 流水线向 STQ 传入 Forwarding 需要的范围（ld_tail）、目标地址和 Load 指令的 func3，进行 Forwarding 查询。此处为了防止 LSV 的发生，LSU 和 Dispatch 之间有 STQ 尾指针状态的传递，每条 load 被 dispatch 时会记录当前 STQ 的 tail 的状态 stq_tail，stq_tail 会跟随这条 load 指令的 uop 一起动，最后作为我们 Forwarding 查找的范围依据，在 load 执行的时候，STQ 中更年轻的 store 被屏蔽，不能 forward 给这个 load。如果查询全部命中，访问 memory 请求将被屏蔽；若仅部分命中，则会同时发起访问 memory 请求。在 Stage3 中通过掩码（bitvalid）按位合并 STQ 和 memory，将合并的数据根据 load 指令的 func3 进行符号扩展，适配 LB/LBU 等指令格式。在 Stage4 完成最终的写回操作

StorePipeline 则是两级流水线，依次进行地址计算、最终写回 STQ 和 ROB。功能简单，只完成地址的初步计算和指令信息的传递。

由于 LSU 内部所有访存操作共享单一 memory 接口，为避免访问冲突，系统采用轻量级的仲裁器对 LoadPipeline 和 STQ 中待提交 Store 的请求进行调度。在本设计中，memory 接口为理想 ready 信号模型，仲裁策略为轮询式公平仲裁，每周期仅允许一个请求通过，保障访存路径在高并发场景下的有序执行。当 LoadPipeline 发起申请但是轮询优先级没有轮到时，会 stall 一个周期

3.8 物理寄存器堆 (PRF)

物理寄存器堆存有 PRF_DEPTH 个寄存器，寄存器宽度为 XLEN。由于存在多个 FU，PRF 必须有多个读写端口。

4 集成仿真

本处理器的仿真使用了 verilator，并通过 surfer 软件观察波形。

4.1 仿真模块

```cpp
int main(int argc, char **argv) {
    Verilated::commandArgs(argc, argv);
    VMachine* top = new VMachine;

    // Initialize VCD tracing
    VerilatedFstC* tfp = new VerilatedFstC;
    Verilated::traceEverOn(true);
    top->trace(tfp, 99);
    tfp->open("wave.fst");

    uint8_t* mem = (uint8_t *)top->rootp->Machine__DOT__core__DOT__mem__DOT__mem_ext__DOT__Memory.data();
    std::ifstream test_prog_stream ("../zig-out/bin/rv32im_test.bin", std::ios_base::binary);
    // FILE* test_prog_fd = fopen("./zig-out/bin/rv32im_test.bin", "r");
    test_prog_stream.seekg(0, std::ios::end);
    std::streampos fileSize = test_prog_stream.tellg();
    test_prog_stream.seekg(0, std::ios::beg);

    test_prog_stream.read(reinterpret_cast<char*>(mem), fileSize);
    test_prog_stream.close();

    for (int i = 0; i < 15000; i++) {
        top->clock = !top->clock; // Toggle clock
        top->eval();
        tfp->dump(i);
        bool got = false;
        // top->rootp->mapping
        bool csr_write_valid = top->rootp->Machine__DOT__core__DOT__exu__DOT__csru__DOT__all_devices_0__DOT__io_wdata_valid_0;
        int csr_addr = top->rootp->Machine__DOT__core__DOT__exu__DOT__csru__DOT__all_devices_0__DOT__io_addr_0;
        int csr_data = top->rootp->Machine__DOT__core__DOT__exu__DOT__csru__DOT__all_devices_0__DOT__io_wdata_bits_0;
        if(top->clock && csr_addr == 0xA && csr_write_valid) {
            std::string str(reinterpret_cast<char*>(&mem[csr_data]));
            if(!str.compare("%T")) {
                // printf("[%d] ", i);
            } else {
                printf("[%d] %s\n", i, &mem[csr_data]);
            }
        }
    }

    // Finalize
    tfp->close();
    delete top;
    return 0;
}
```

Figure 9: 仿真模块

为了减少对 Chisel 模拟器的依赖，本项目中调用了 verilator 的 api，将程序载入处理器的内存中。

4.2 集成测试

5 小组分工说明

组员	分工部分
刘恒雨(组长)	顶层模块+测试+文档排布+优化代码(dispatch)+branch_mask+csr
杨钧铎	ROB + Rename
饶忠禹	LSU
胡英瀚	Issue Queue
李可名/赵力	BP

邢益成	Fetch + Decode
马嘉一	FU(ALU, BP, DIV, MUL)
胡继仁	prf
蔡家麒	RAS

Bibliography