

香山开源处理器

设计文档

适用于 昆明湖 V2R2

52fc49d

2025 年 7 月 15 日

前言

本文档为《香山开源处理器设计文档》，详细介绍了昆明湖 V2R2 的微架构设计实现细节。

您可从以下渠道获取最新版的本文档：

- 网页版：<https://docs.xiangshan.cc/projects/design/>
- PDF 文件：<https://github.com/OpenXiangShan/XiangShan-Design-Doc/releases>

您可访问香山文档网站 docs.xiangshan.cc 获取更多文档。

本文档源语言为中文，托管于 GitHub，采用 Weblate 平台进行协作翻译。欢迎您参与校对、纠错与翻译工作，共同完善文档内容！

声明

版权所有 © 2024 - 2025 香山开源处理器团队·北京开源芯片研究院

本文档采用“知识共享署名 4.0”协议公开发布。您可以根据该协议的规定，自由地使用、修改、分发本文档，但必须给出恰当的署名、表明原始许可协议、标注是否对原始文档作了修改。具体许可条款以知识共享组织公布的完整法律文本为准。

本文档仅提供阶段性信息，可能不定期根据实际情况更新。除非另有约定，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

版本信息

| 日期 | 版本 | 说明 |
|-------|----|----|
| 第一次发布 | | |

目录

| | |
|--------------------------------|----------|
| 第一部分 前端 | 1 |
| 1 BPU | 2 |
| 1.1 昆明湖 BPU 模块文档 | 2 |
| 1.1.1 术语说明 | 2 |
| 1.1.2 设计规格 | 2 |
| 1.1.3 功能描述 | 2 |
| 1.1.4 总体设计 | 11 |
| 1.1.5 寄存器配置 | 13 |
| 1.1.6 参考文档 | 13 |
| 1.2 BPU 子模块 Composer | 14 |
| 1.2.1 功能概述 | 14 |
| 1.2.2 整体框图 | 15 |
| 1.2.3 接口时序 | 15 |
| 1.2.4 关键电路 | 16 |
| 1.3 BPU 子模块 FTB | 18 |
| 1.3.1 功能概述 | 18 |
| 1.3.2 整体框图 | 19 |
| 1.3.3 接口时序 | 19 |
| 1.3.4 FTBBank | 20 |
| 1.3.5 FTB 项的生成条件简述 | 21 |
| 1.3.6 FTB 存储结构 | 21 |
| 1.3.7 目标地址生成逻辑 | 22 |
| 1.3.8 更新流程 | 22 |
| 1.4 BPU 子模块 uFTB | 24 |
| 1.4.1 功能概述 | 24 |
| 1.4.2 整体框图 | 25 |
| 1.4.3 接口时序 | 26 |
| 1.5 BPU 子模块 TAGE-SC | 26 |
| 1.5.1 功能 | 26 |
| 1.5.2 整体框图 | 35 |
| 1.5.3 接口时序 | 35 |
| 1.6 BPU 子模块 ITTAGE | 37 |
| 1.6.1 功能 | 37 |
| 1.6.2 存储结构 | 39 |

| | |
|--|-----------|
| 1.6.3 索引方式 | 39 |
| 1.6.4 预测流程 | 39 |
| 1.6.5 训练流程 | 40 |
| 1.7 BPU 子模块 RAS | 40 |
| 1.7.1 术语说明 | 40 |
| 1.7.2 功能 | 40 |
| 1.7.3 整体框图 | 43 |
| 1.7.4 接口时序 | 43 |
| 1.7.5 RAS 存储结构 | 46 |
| 1.7.6 预测与更新 | 46 |
| 2 昆明湖 FTQ 模块文档 | 49 |
| 2.1 术语说明 | 49 |
| 2.2 功能描述 | 49 |
| 2.2.1 功能概述 | 49 |
| 2.2.2 暂存 BPU 预测的取指目标，并向 IFU 发送取指请求 | 49 |
| 2.2.3 暂存 BPU 的预测信息，并送回 BPU 训练 | 50 |
| 2.2.4 重定向恢复 | 52 |
| 2.2.5 向 ICache 发送预取请求 | 53 |
| 2.3 整体框图 | 53 |
| 2.4 接口时序 | 53 |
| 2.5 职能描述 | 54 |
| 2.6 内部结构 | 54 |
| 2.7 指令在 FTQ 中的生存周期 | 55 |
| 2.8 FTQ 的其它功能 | 55 |
| 3 取指令单元 | 56 |
| 3.1 昆明湖 IFU 模块文档 | 56 |
| 3.1.1 术语说明 | 56 |
| 3.1.2 子模块列表 | 56 |
| 3.1.3 功能描述 | 57 |
| 3.1.4 总体设计 | 63 |
| 3.2 IFU 子模块 PreDecoder | 66 |
| 3.2.1 功能描述 | 66 |
| 3.2.2 整体框图 | 66 |
| 3.2.3 接口时序 | 66 |
| 3.3 IFU 子模块 PredChecker | 68 |
| 3.3.1 功能描述 | 68 |
| 4 指令缓存 | 71 |
| 4.1 XiangShan ICache 设计文档 | 71 |
| 4.1.1 术语说明 | 71 |
| 4.1.2 子模块列表 | 71 |
| 4.1.3 设计规格 | 72 |

| | |
|-----------------------------------|-----------|
| 4.1.4 参数列表 | 72 |
| 4.1.5 功能概述 | 73 |
| 4.1.6 功能详述 | 75 |
| 4.1.7 参考文献 | 83 |
| 4.2 MainPipe 子模块文档 | 83 |
| 4.2.1 S0 流水级 | 84 |
| 4.2.2 S1 流水级 | 84 |
| 4.2.3 S2 流水级 | 84 |
| 4.3 IPrefetchPipe 子模块文档 | 85 |
| 4.3.1 S0 流水级 | 85 |
| 4.3.2 S1 流水级 | 85 |
| 4.3.3 S2 流水级 | 86 |
| 4.3.4 命中信息的更新 | 86 |
| 4.4 WayLookup 子模块文档 | 87 |
| 4.4.1 GPAddr 省面积机制 | 89 |
| 4.5 MissUnit 子模块文档 | 89 |
| 4.5.1 MSHR 的管理 | 91 |
| 4.5.2 请求入队 | 91 |
| 4.5.3 acquire | 91 |
| 4.5.4 grant | 91 |
| 4.6 Replacer 子模块文档 | 91 |
| 4.6.1 touch | 92 |
| 4.6.2 victim | 92 |
| 4.7 CtrlUnit 子模块文档 | 92 |
| 4.7.1 mmio-mapped CSR | 92 |
| 4.7.2 错误校验使能 | 94 |
| 4.7.3 错误注入使能 | 94 |
| 5 昆明湖 PrunedAddr 文档 | 95 |
| 5.1 背景介绍 | 95 |
| 5.2 使用指南 | 95 |
| 5.3 遗留问题 | 96 |
| 第二部分 Backend 后端 | 97 |
| 6 后端整体介绍 | 98 |
| 6.1 基本技术规格 | 98 |
| 7 CtrlBlock 控制模块 | 99 |
| 7.1 CtrlBlock | 99 |
| 7.1.1 术语说明 | 99 |
| 7.1.2 子模块列表 | 99 |
| 7.1.3 设计规格 | 100 |

| | |
|--|-----|
| 7.1.4 功能 | 100 |
| 7.2 译码 | 101 |
| 7.2.1 译码阶段输入 | 102 |
| 7.2.2 译码输出 | 102 |
| 7.3 FusionDecoder | 102 |
| 7.4 Redirect | 103 |
| 7.4.1 RedirectGenerator | 103 |
| 7.4.2 redirect 的生成 | 104 |
| 7.4.3 redirect 分发 | 105 |
| 7.4.4 保证 redirect 发出的顺序 | 106 |
| 7.5 快照恢复 | 107 |
| 7.5.1 SnapshotGenerator | 107 |
| 7.5.2 快照的创建 | 108 |
| 7.5.3 快照的删除 | 108 |
| 7.5.4 快照的管理 | 109 |
| 7.6 pcMem | 109 |
| 7.7 GPAMem | 110 |
| 7.8 Trace | 111 |
| 7.8.1 feature 支持 | 111 |
| 7.8.2 trace 各级流水线功能: | 112 |
| 7.8.3 trace buffer 压缩机制 | 112 |
| 7.9 XiangShan Decode 设计文档 | 112 |
| 7.9.1 术语说明 | 112 |
| 7.9.2 子模块列表 | 112 |
| 7.9.3 设计规格 | 113 |
| 7.9.4 功能 | 113 |
| 7.9.5 整体设计 | 113 |
| 7.9.6 整体框图 | 114 |
| 7.9.7 接口列表 | 114 |
| 7.9.8 二级模块 VTypeGen | 115 |
| 7.9.9 二级模块 DecodeUnit | 116 |
| 7.9.10 二级模块 VecExceptionGen | 119 |
| 7.10 Rename 重命名 | 119 |
| 7.10.1 基本功能 | 120 |
| 7.10.2 重命名输入 | 120 |
| 7.10.3 重命名输出 | 120 |
| 7.10.4 分配整数物理寄存器 | 120 |
| 7.10.5 分配浮点或向量物理寄存器 | 121 |
| 7.10.6 设置源操作数的物理寄存器 (psrc) | 121 |
| 7.10.7 设置目的操作数的物理寄存器 (pdest) | 121 |
| 7.10.8 整数指令提交 | 121 |
| 7.10.9 浮点或向量指令提交 | 121 |
| 7.10.10 重定向 | 121 |

| | |
|---|-----|
| 7.10.11 重新重命名 | 122 |
| 7.10.12 robIdx 分配 | 122 |
| 7.10.13 决定重命名快照的生成 | 122 |
| 7.10.14 整体框图 | 123 |
| 7.10.15 接口时序 | 123 |
| 7.11 RenameTableWrapper | 125 |
| 7.11.1 读推测重命名表 | 125 |
| 7.11.2 重命名阶段写推测重命名表 | 125 |
| 7.11.3 提交阶段写体系结构重命名表 | 125 |
| 7.11.4 提交阶段提供物理寄存器释放信息 | 126 |
| 7.11.5 重新重命名阶段写推测重命名表 | 126 |
| 7.11.6 重命名快照的维护 | 126 |
| 7.11.7 整体框图 | 126 |
| 7.11.8 接口时序 | 127 |
| 7.12 支持 move 消除的 RenameTable | 128 |
| 7.12.1 整体框图 | 129 |
| 7.12.2 接口时序 | 129 |
| 7.13 不支持 move 消除的 RenameTable | 129 |
| 7.13.1 接口时序 | 131 |
| 7.14 StdFreeList | 131 |
| 7.14.1 整体框图 | 131 |
| 7.14.2 接口时序 | 131 |
| 7.14.3 关键电路: 环形队列 | 134 |
| 7.15 MEFreeList | 135 |
| 7.15.1 整体框图 | 135 |
| 7.15.2 接口时序 | 135 |
| 7.16 CompressUnit | 137 |
| 7.16.1 整体框图 | 138 |
| 7.16.2 接口时序 | 138 |
| 7.17 SnapshotGenerator | 138 |
| 7.18 Dispatch | 139 |
| 7.18.1 术语说明 | 139 |
| 7.18.2 子模块列表 | 139 |
| 7.18.3 设计规格 | 140 |
| 7.18.4 功能 | 140 |
| 7.18.5 总体设计 | 142 |
| 7.18.6 模块设计 | 143 |
| 7.19 Rob | 144 |
| 7.19.1 术语说明 | 144 |
| 7.19.2 子模块列表 | 144 |
| 7.19.3 设计规格 | 145 |
| 7.19.4 功能 | 145 |
| 7.19.5 总体设计 | 151 |

| | |
|---|------------|
| 7.19.6 模块设计 | 151 |
| 8 DataPath 数据通路 | 152 |
| 8.1 DataPath | 152 |
| 8.1.1 术语说明 | 152 |
| 8.1.2 总体设计 | 152 |
| 8.1.3 功能 | 154 |
| 8.1.4 模块设计 | 156 |
| 8.2 Og2ForVector | 161 |
| 8.2.1 总体设计 | 162 |
| 8.2.2 功能 | 162 |
| 8.3 WbDataPath | 163 |
| 8.3.1 术语说明 | 163 |
| 8.3.2 子模块列表 | 163 |
| 8.3.3 功能 | 163 |
| 8.4 WbFuBusyTable | 167 |
| 8.4.1 功能 | 167 |
| 8.5 BypassNetwork | 168 |
| 8.5.1 术语说明 | 168 |
| 8.5.2 子模块列表 | 168 |
| 8.5.3 功能 | 168 |
| 8.5.4 模块设计 | 171 |
| 9 Schedule and Issue 调度与发射 | 173 |
| 9.1 Scheduler | 173 |
| 9.2 IssueQueue | 173 |
| 9.2.1 设计规格 | 173 |
| 9.2.2 功能 | 173 |
| 9.2.3 整体框图 | 175 |
| 9.2.4 接口时序 | 176 |
| 9.2.5 二级模块 WakeupQueue | 176 |
| 9.2.6 二级模块 AgeDetector | 176 |
| 9.3 IssueQueueEntries | 177 |
| 9.3.1 术语说明 | 179 |
| 9.3.2 设计规格 | 179 |
| 9.3.3 功能 | 179 |
| 9.3.4 整体框图 | 182 |
| 9.3.5 接口时序 | 182 |
| 9.3.6 二级模块 EnqEntry & OthersEntry | 183 |
| 10 ExuBlock 执行 | 190 |
| 10.1 ExuBlock | 190 |
| 10.1.1 输入输出 | 190 |
| 10.1.2 功能 | 190 |

| | |
|--|------------|
| 10.1.3 设计规格 | 192 |
| 10.2 ExuUnit | 192 |
| 10.2.1 术语说明 | 192 |
| 10.2.2 输入输出 | 193 |
| 10.2.3 功能 | 193 |
| 10.2.4 设计规格 | 194 |
| 10.2.5 门控 | 195 |
| 11 FunctionUnit 功能单元 | 196 |
| 11.1 IntFunctionUnit | 196 |
| 11.1.1 jmp | 196 |
| 11.1.2 brh | 196 |
| 11.1.3 i2f | 196 |
| 11.1.4 i2v | 197 |
| 11.1.5 f2v | 197 |
| 11.1.6 csr | 198 |
| 11.1.7 alu | 198 |
| 11.1.8 mul | 200 |
| 11.1.9 div | 200 |
| 11.1.10 fence | 201 |
| 11.1.11 bku | 201 |
| 11.2 FpFunctionUnit | 202 |
| 11.2.1 falu | 202 |
| 11.2.2 fmac | 204 |
| 11.2.3 fcvt | 204 |
| 11.2.4 fDivSqrt | 205 |
| 11.3 VecFunctionUnit | 205 |
| 11.3.1 vsetiwi vsetiwf vsetfwf | 206 |
| 11.3.2 vipu | 206 |
| 11.3.3 vialuF | 206 |
| 11.3.4 vldu | 210 |
| 11.3.5 vstu | 210 |
| 11.3.6 vppu | 210 |
| 11.3.7 vimac | 211 |
| 11.3.8 vidiv | 212 |
| 11.3.9 vfalu | 212 |
| 11.3.10 vfma | 213 |
| 11.3.11 vfdi | 214 |
| 11.3.12 vfcvt | 215 |
| 11.4 FpFunctionUnit | 215 |
| 11.4.1 ldu | 215 |
| 11.4.2 stu | 216 |
| 11.4.3 mou | 217 |

| | |
|--|------------|
| 12 VFPU | 218 |
| 12.1 术语说明 | 218 |
| 12.2 设计规格 | 218 |
| 12.3 功能 | 218 |
| 12.4 算法设计 | 219 |
| 12.4.1 向量浮点加法 | 219 |
| 12.4.2 向量浮点融合乘加算法 | 225 |
| 12.4.3 向量浮点除法算法 | 237 |
| 12.5 硬件设计 | 244 |
| 12.5.1 向量浮点加法器 | 244 |
| 12.5.2 向量浮点融合乘加器 | 250 |
| 12.5.3 向量浮点除法器 | 252 |
| 12.5.4 向量格式转换模块 <i>VCVT</i> | 256 |
| 13 CSR | 258 |
| 13.1 术语说明 | 258 |
| 13.2 设计规格 | 258 |
| 13.3 功能 | 259 |
| 13.4 自定义 CSR | 260 |
| 13.4.1 sbpctl | 260 |
| 13.4.2 spfctl | 260 |
| 13.4.3 slvpredctl | 261 |
| 13.4.4 smblockctl | 261 |
| 13.4.5 srnctl | 262 |
| 13.4.6 mcorepwr | 262 |
| 13.4.7 mflushpwr | 262 |
| 13.5 CSR 异常检查 | 263 |
| 13.6 CSR 只读指令乱序 | 264 |
| 14 HPM | 265 |
| 14.1 基本信息 | 265 |
| 14.1.1 术语说明 | 265 |
| 14.1.2 子模块列表 | 265 |
| 14.1.3 设计规格 | 265 |
| 14.1.4 功能 | 266 |
| 14.2 总体设计 | 266 |
| 14.2.1 HPerfMonitor 计数器组织模块 | 267 |
| 14.2.2 HperfCounter 单个计数器模块 | 267 |
| 14.2.3 PFEEvent Hpmevent 寄存器的副本 | 267 |
| 14.3 HPM 相关的控制寄存器 | 267 |
| 14.3.1 机器模式性能事件计数禁止寄存器 (MCOUNTINHIBIT) | 267 |
| 14.3.2 机器模式性能事件计数器访问授权寄存器 (MCOUNTEREN) | 268 |
| 14.3.3 监督模式性能事件计数器访问授权寄存器 (SCOUNTEREN) | 268 |

| | |
|---|------------|
| 14.3.4 虚拟化模式性能事件计数器访问授权寄存器 (HCOUNTEREN) | 269 |
| 14.3.5 监督模式时间比较寄存器 (STIMECMP) | 269 |
| 14.3.6 客户虚拟机监督模式时间比较寄存器 (VSTIMECMP) | 270 |
| 14.4 HPM 相关的性能事件选择器 | 270 |
| 14.4.1 Topdown PMU | 279 |
| 14.5 HPM 相关的性能事件计数器 | 281 |
| 15 Debug 模块 | 283 |
| 15.1 Debug Module | 283 |
| 15.1.1 术语说明 | 283 |
| 15.1.2 参数设计 | 283 |
| 15.1.3 总体设计 | 284 |
| 15.1.4 模块设计 | 286 |
| 第三部分 访存 | 289 |
| 16 访存流水线 | 290 |
| 16.1 访存流水线 LSU | 290 |
| 16.1.1 子模块列表 | 290 |
| 16.1.2 设计规格 | 290 |
| 16.1.3 功能描述 | 292 |
| 16.1.4 总体设计 | 296 |
| 16.2 Load 指令执行单元 LoadUnit | 296 |
| 16.2.1 功能描述 | 296 |
| 16.2.2 整体框图 | 300 |
| 16.2.3 接口时序 | 301 |
| 16.3 Store 地址执行单元 StoreUnit | 302 |
| 16.3.1 功能描述 | 302 |
| 16.3.2 整体框图 | 305 |
| 16.3.3 接口时序 | 306 |
| 16.4 Store 数据执行单元 StdExeUnit | 306 |
| 16.4.1 功能描述 | 306 |
| 16.4.2 整体框图 | 306 |
| 16.4.3 接口时序 | 306 |
| 16.5 原子指令执行单元 AtomicsUnit | 309 |
| 16.5.1 功能描述 | 309 |
| 16.5.2 整体框图 | 309 |
| 16.5.3 Zacas 扩展 | 311 |
| 16.5.4 原子指令的 Uop 拆分 | 311 |
| 16.5.5 异常汇总 | 312 |
| 16.6 向量访存 | 313 |
| 16.6.1 向量访存 | 313 |
| 16.6.2 向量 Load 拆分单元 VLSSplit | 314 |

| | |
|---|------------|
| 16.6.3 向量 Store 拆分单元 VSSplit | 317 |
| 16.6.4 向量 Load 合并单元 VLMergeBuffer | 320 |
| 16.6.5 向量 Store 合并单元 VSMergeBuffer | 321 |
| 16.6.6 向量 Segment 访存指令处理单元 VSegmentUnit | 323 |
| 16.6.7 向量 FOF 指令单元 VfofBuffer | 327 |
| 16.7 访存队列 | 328 |
| 16.7.1 访存队列 LSQ | 328 |
| 16.7.2 Load 队列 VirtualLoadQueue | 331 |
| 16.7.3 读后读违例检查 LoadQueueRAR | 333 |
| 16.7.4 写后读违例检查 LoadQueueRAW | 337 |
| 16.7.5 Load 重发队列 LoadQueueReplay | 339 |
| 16.7.6 Uncache Load 处理单元 LoadQueueUncache | 345 |
| 16.7.7 Load 异常缓冲 LqExceptionBuffer | 351 |
| 16.7.8 Store 队列 StoreQueue | 351 |
| 16.8 Uncache 处理单元 Uncache | 361 |
| 16.8.1 功能描述 | 361 |
| 16.8.2 整体框图 | 364 |
| 16.8.3 接口时序 | 365 |
| 16.9 Store 提交缓冲 SBuffer | 366 |
| 16.9.1 功能描述 | 366 |
| 16.9.2 整体框图 | 368 |
| 16.9.3 接口时序 | 368 |
| 16.10 Load 非对齐访存单元 LoadMisalignBuffer | 371 |
| 16.10.1 功能描述 | 371 |
| 16.10.2 整体框图 | 375 |
| 16.10.3 主要端口 | 375 |
| 16.10.4 接口时序 | 376 |
| 16.11 Store 非对齐访存单元 StoreMisalignBuffer | 376 |
| 16.11.1 功能描述 | 376 |
| 16.11.2 整体框图 | 379 |
| 16.11.3 主要端口 | 380 |
| 16.11.4 接口时序 | 380 |
| 17 数据缓存 | 381 |
| 17.1 数据高速缓存 DCache | 381 |
| 17.1.1 术语说明 | 381 |
| 17.1.2 子模块列表 | 381 |
| 17.1.3 DCache 设计规格 | 381 |
| 17.1.4 整体框图 | 382 |
| 17.1.5 功能描述 | 383 |
| 17.2 Load 访存流水线 LoadPipe | 384 |
| 17.2.1 功能描述 | 384 |
| 17.2.2 整体框图 | 384 |

| | |
|------------------------------------|------------|
| 17.2.3 接口时序 | 385 |
| 17.3 缺失队列 MissQueue | 386 |
| 17.3.1 功能描述 | 386 |
| 17.3.2 整体框图 | 387 |
| 17.3.3 接口时序 | 388 |
| 17.3.4 MissEntry 模块 | 389 |
| 17.4 Probe 队列 ProbeQueue | 391 |
| 17.4.1 功能描述 | 391 |
| 17.4.2 整体框图 | 391 |
| 17.4.3 接口时序 | 391 |
| 17.4.4 ProbeEntry 模块 | 392 |
| 17.5 主流水线 MainPipe | 393 |
| 17.5.1 功能描述 | 393 |
| 17.5.2 整体框图 | 394 |
| 17.5.3 接口时序 | 395 |
| 17.6 写回队列 WritebackQueue | 397 |
| 17.6.1 功能描述 | 397 |
| 17.6.2 整体框图 | 397 |
| 17.6.3 接口时序 | 397 |
| 17.6.4 WritebackEntry 模块 | 398 |
| 17.7 错误处理与自定义故障注入指令 | 398 |
| 17.7.1 功能描述 | 398 |
| 17.7.2 整体框图 | 403 |
| 17.7.3 接口时序 | 403 |
| 18 内存管理单元 | 406 |
| 18.1 内存管理单元概述 | 406 |
| 18.1.1 术语说明 | 406 |
| 18.1.2 设计规格 | 407 |
| 18.1.3 功能描述 | 407 |
| 18.1.4 异常处理机制 | 414 |
| 18.1.5 总体设计 | 415 |
| 18.1.6 接口列表 | 417 |
| 18.1.7 接口时序 | 418 |
| 18.2 二级模块 L1 TLB | 419 |
| 18.2.1 设计规格 | 419 |
| 18.2.2 功能 | 419 |
| 18.2.3 整体框图 | 430 |
| 18.2.4 接口时序 | 431 |
| 18.3 二级模块 Repeater | 435 |
| 18.3.1 设计规格 | 435 |
| 18.3.2 功能 | 435 |
| 18.3.3 整体框图 | 437 |

| | |
|--|------------|
| 18.3.4 接口列表 | 438 |
| 18.3.5 接口时序 | 438 |
| 18.4 二级模块 L2TLB | 440 |
| 18.4.1 二级模块 L2 TLB | 440 |
| 18.4.2 三级模块 Page Cache | 448 |
| 18.4.3 三级模块 Page Table Walker | 454 |
| 18.4.4 三级模块 Last Level Page Table Walker | 457 |
| 18.4.5 三级模块 Hypervisor Page Table Walker | 461 |
| 18.4.6 三级模块 Miss Queue | 464 |
| 18.4.7 三级模块 Prefetcher | 464 |
| 18.5 二级模块 PMP&PMA | 466 |
| 18.5.1 设计规格 | 467 |
| 18.5.2 功能 | 467 |
| 18.5.3 整体框图 | 472 |
| 18.5.4 接口列表 | 473 |
| 18.5.5 接口时序 | 473 |
| 19 预取 | 474 |
| 第四部分 缓存子系统 | 475 |
| 20 二级缓存 | 476 |
| 20.1 二级缓存 CoupledL2 | 476 |
| 20.1.1 子模块列表 | 476 |
| 20.1.2 设计规格 | 477 |
| 20.1.3 功能描述 | 478 |
| 20.1.4 总体设计 | 484 |
| 20.2 A 通道请求缓冲 RequestBuffer | 485 |
| 20.2.1 功能描述 | 485 |
| 20.2.2 整体框图 | 486 |
| 20.3 请求仲裁器与主流水线 | 486 |
| 20.3.1 S0 流水级 | 487 |
| 20.3.2 S1 流水级 | 487 |
| 20.3.3 S2 流水级 | 488 |
| 20.3.4 S3 流水级 | 488 |
| 20.3.5 S4 流水级 | 491 |
| 20.3.6 S5 流水级 | 492 |
| 20.4 目录 Directory | 492 |
| 20.5 数据 SRAM DataStorage | 493 |
| 20.6 MSHR | 493 |
| 20.6.1 生命周期 | 493 |
| 20.6.2 状态机 | 493 |
| 20.6.3 任务分发 | 495 |

| | |
|--------------------------------------|-----|
| 20.6.4 Snoop 处理 | 502 |
| 20.6.5 写回嵌套处理 | 506 |
| 20.6.6 Retry 与 P-Credit 机制 | 507 |
| 20.7 上游 TileLink 总线通道 | 508 |
| 20.7.1 SinkA | 508 |
| 20.7.2 SinkC | 508 |
| 20.7.3 GrantBuffer | 510 |
| 20.8 下游 CHI 总线通道 | 512 |
| 20.8.1 TXREQ | 512 |
| 20.8.2 RXRSP | 512 |
| 20.8.3 RXDAT | 514 |
| 20.8.4 RXSNP | 514 |
| 20.8.5 TXDAT | 515 |
| 20.8.6 TXRSP | 516 |
| 20.8.7 P-Credit 管理机制 | 517 |
| 20.8.8 链路层控制器 LinkMonitor | 518 |
| 20.9 MMIO 转接桥 MMIOBridge | 518 |
| 20.9.1 状态机 | 521 |
| 20.9.2 定序 | 521 |
| 20.9.3 内存属性 | 522 |
| 20.9.4 P-Credit 仲裁 | 522 |
| 20.10 错误处理 | 522 |
| 20.10.1 术语说明 | 522 |
| 20.10.2 设计规格 | 523 |
| 20.10.3 Cached 访存请求错误处理 | 523 |
| 20.10.4 Uncached 访存请求错误处理 | 525 |

图目录

| | |
|---------------------------------------|----|
| 1.1 整体框图 | 12 |
| 1.2 BPU 到 FTQ 接口时序 | 12 |
| 1.3 FTQ 到 BPU redirect 接口时序 | 13 |
| 1.4 FTQ 到 BPU update 接口时序 | 13 |
| 1.5 Composer 模块整体框图 | 15 |
| 1.6 控制信号 Ctrl 接口时序 | 15 |
| 1.7 重定向接口时序 | 16 |
| 1.8 分支预测块训练接口时序 | 16 |
| 1.9 Composer meta 拼接 | 17 |
| 1.10 重定向/分支历史更新来源仲裁逻辑 | 17 |
| 1.11 整体框图 | 19 |
| 1.12 结果输出接口 | 19 |
| 1.13 更新接口 | 20 |
| 1.14 读数据接口 | 20 |
| 1.15 写入 SRAM 的流水线 | 24 |
| 1.16 整体框图 | 25 |
| 1.17 结果输出接口 | 26 |
| 1.18 更新接口 | 26 |
| 1.19 TAGE 原理 | 27 |
| 1.20 折叠历史真实实现 | 29 |
| 1.21 TAGE meta 构成 | 31 |
| 1.22 训练流程 | 32 |
| 1.23 SC meta 具体构成 | 35 |
| 1.24 整体框图 | 35 |
| 1.25 pc 和折叠历史时序 | 36 |
| 1.26 训练流程 | 37 |
| 1.27 整体框图 | 43 |
| 1.28 2 阶段更新输入输出接口 | 44 |
| 1.29 3 阶段更新输入输出接口 | 44 |
| 1.30 Stack 模块输入接口 | 44 |
| 1.31 重定向恢复接口 | 45 |
| 1.32 重定向恢复接口 | 45 |
| 1.33 指令提交训练接口 | 46 |
| 1.34 getTop 逻辑细节 | 47 |
| 1.35 specPop 逻辑细节 | 47 |

| | |
|---|-----|
| 1.36 specPush 逻辑细节 | 48 |
| 2.1 FTQ 结构 | 53 |
| 2.2 接口时序 | 54 |
| 3.1 F3 MMIO 状态机示意图 | 60 |
| 3.2 IFU 模块整体框图 | 62 |
| 3.3 IFU 模块流水级 | 62 |
| 3.4 FTQ 请求接口时序示例 | 63 |
| 3.5 ICache 返回接口以及到 Ibuffer 和写回 FTQ 接口时序示例 | 64 |
| 3.6 MMIO 请求接口时序示例 | 65 |
| 3.7 PreDecoder 结构 | 67 |
| 3.8 PreDecode 接口时序 | 67 |
| 3.9 PredChecker 结构 | 69 |
| 3.10 PredChecker 接口时序 | 70 |
| 4.1 FTQ 指针示意 | 73 |
| 4.2 ICache 结构 | 74 |
| 4.3 ICache 两条流水线的关系 | 76 |
| 4.4 ICache 预取请求接收与仲裁 | 77 |
| 4.5 dataArray 分 bank 示意图 | 78 |
| 4.6 dataArray 数据返回示意图 | 79 |
| 4.7 dataArray 数据返回示意图 | 80 |
| 4.8 MainPipe 结构 | 84 |
| 4.9 IPrefetchPipe 结构 | 85 |
| 4.10 IPrefetchPipe S1 状态机 | 86 |
| 4.11 WayLookup 队列结构 | 88 |
| 4.12 WayLookup 命中信息更新 | 89 |
| 4.13 MissUnit 结构 | 90 |
| 4.14 PLRU 算法示意 | 92 |
| 6.1 后端整体框架 | 98 |
| 7.1 CtrlBlock 总览 | 101 |
| 7.2 fusion decoder 总览 | 103 |
| 7.3 redirect 总览 | 104 |
| 7.4 redirect 的生成 | 105 |
| 7.5 发向前端的 redirect | 106 |
| 7.6 snapshots 总览 | 108 |
| 7.7 snapshot 的生成、删除和管理 | 109 |
| 7.8 PCMem 总览 | 110 |
| 7.9 GPAMem 总览 | 111 |
| 7.10 trace 示意图 | 111 |
| 7.11 decode | 114 |
| 7.12 Rename 整体框图 | 123 |

| | |
|---|-----|
| 7.13 Decode 输入接口时序示意图 | 123 |
| 7.14 Rename 输出接口时序示意图 | 123 |
| 7.15 指令提交逻辑时序示意图 | 124 |
| 7.16 重定向和重新重命名时序示意图 | 124 |
| 7.17 RenameTableWrapper 整体框图 | 126 |
| 7.18 整数读写接口时序示意图 | 127 |
| 7.19 重新重命名和提交接口时序示意图 | 128 |
| 7.20 RenameTable 整体框图 | 129 |
| 7.21 支持 move 消除的 RenameTable 读写接口时序示意图 | 130 |
| 7.22 不支持 move 消除的 RenameTable 读写接口时序示意图 | 131 |
| 7.23 StdFreeList 整体框图 | 132 |
| 7.24 StdFreeList 空闲寄存器分配时序示意图 | 132 |
| 7.25 StdFreeList 指令提交时序示意图 | 133 |
| 7.26 StdFreeList 指令重新重命名时序示意图 | 133 |
| 7.27 普通队列 | 134 |
| 7.28 环形队列 | 134 |
| 7.29 MEFreeList 整体框图 | 135 |
| 7.30 MEFreeList 空闲寄存器分配时序示意图 | 136 |
| 7.31 MEFreeList 指令提交时序示意图 | 136 |
| 7.32 MEFreeList 指令重新重命名时序示意图 | 137 |
| 7.33 CompressUnit 整体框图 | 138 |
| 7.34 整体框图 | 141 |
| 7.35 整体框图 | 143 |
| 7.36 rob_entries | 146 |
| 7.37 rob_enq | 147 |
| 7.38 rob_commit | 148 |
| 7.39 rob_walkPtr | 149 |
| 8.1 整体框图 | 154 |
| 8.2 整体框图 | 157 |
| 8.3 RegCache 整体框图 | 159 |
| 8.4 整体框图 | 162 |
| 8.5 WbDataPath 整体框图 | 165 |
| 8.6 向量加载功能单元合并模块 | 166 |
| 8.7 BypassNetwork | 170 |
| 9.1 示意图 | 175 |
| 9.2 示意图 | 176 |
| 9.3 示意图 | 176 |
| 9.4 示意图 | 177 |
| 9.5 示意图 | 178 |
| 9.6 示意图 | 179 |
| 9.7 示意图 | 181 |

| | |
|--|-----|
| 9.8 示意图 | 182 |
| 9.9 示意图 | 183 |
| 9.10 示意图 | 184 |
| 9.11 示意图 | 185 |
| 9.12 示意图 | 185 |
| 9.13 示意图 | 186 |
| 9.14 示意图 | 186 |
| 9.15 示意图 | 187 |
| 9.16 示意图 | 188 |
| 9.17 示意图 | 189 |
| 10.1 ExuBlock 总览 | 191 |
| 10.2 ExuUnit 总览 | 194 |
| 12.1 单通路浮点加法算法 | 220 |
| 12.2 far 路径算法示意图 | 222 |
| 12.3 close 路径算法示意图 | 223 |
| 12.4 FMA 算法流程图 | 226 |
| 12.5 二进制竖式法计算乘法 | 228 |
| 12.6 假设部分积全为负数取反加一示意图 | 229 |
| 12.7 假设部分积全为负数化简后的结果 | 230 |
| 12.8 部分积为正数时修正结果 | 230 |
| 12.9 部分积进位修正 | 230 |
| 12.10 fp_c 有效数字扩展位分布情况 | 235 |
| 12.11 向量共用 Booth 编码示意图 | 237 |
| 12.12 三次 radix-4 组成 radix-64 的简单实现 | 241 |
| 12.13 数字迭代优化算法 | 242 |
| 12.14 标量单一精度浮点加法器架构图 | 245 |
| 12.15 标量混合精度浮点加法器架构图 | 246 |
| 12.16 向量浮点加法器架构图 | 247 |
| 12.17 FloatAdderF64Widen 流水线划分 | 247 |
| 12.18 FloatAdderF32WidenF16 流水线划分 | 248 |
| 12.19 FloatAdderF16 流水线划分 | 248 |
| 12.20 向量浮点融合乘加器架构图 | 250 |
| 12.21 标量浮点除法器架构图 | 253 |
| 12.22 向量浮点除法器架构图 | 254 |
| 12.23 VCVT 总体设计 | 256 |
| 14.1 HPM 总体设计 | 267 |
| 15.1 DebugModule 总览 | 284 |
| 15.2 DebugModule 多时钟域 | 285 |
| 16.1 MemBlock 架构图 | 296 |
| 16.2 LoadUnit 整体框图 | 300 |

| | |
|---|-----|
| 16.3 LoadUnit 接口时序 | 301 |
| 16.4 stage 0 不同源仲裁时序 | 302 |
| 16.5 StoreUnit 流水线 | 303 |
| 16.6 StoreUnit 整体框图 | 305 |
| 16.7 StoreUnit 接口时序 | 306 |
| 16.8 stdExeUnit 整体框图 | 307 |
| 16.9 stdExeUnit 有效请求接口时序 | 307 |
| 16.10 AtomicsUnit 状态机示意图 | 309 |
| 16.11 A 扩展原子指令的 Uop 拆分示意图 | 311 |
| 16.12 AMOCAS.W 和 AMOCAS.D 指令的 Uop 拆分示意图 | 312 |
| 16.13 AMOCAS.Q 指令的 Uop 拆分示意图 | 312 |
| 16.14 alt text | 324 |
| 16.15 alt text | 326 |
| 16.16 LSQ 分配 | 329 |
| 16.17 LSQ 整体框架 | 329 |
| 16.18 入队更新 | 330 |
| 16.19 VirtualLoadQueue 整体框图 | 331 |
| 16.20 VirtualLoadQueue-enqueue | 332 |
| 16.21 VirtualLoadQueue-writeback | 332 |
| 16.22 LoadQueueRAR 整体框图 | 334 |
| 16.23 LoadQueueRAR 请求入队时序 | 335 |
| 16.24 load-load 违例检查时序 | 335 |
| 16.25 LoadQueueRAW 整体框图 | 338 |
| 16.26 LoadQueueRAW 请求入队时序 | 338 |
| 16.27 store-load 违例检查时序 | 339 |
| 16.28 Freelist | 341 |
| 16.29 Freelist 回收 | 341 |
| 16.30 LoadQueueReplay 整体框图 | 343 |
| 16.31 LoadQueueReplay 重发入队时序图 | 343 |
| 16.32 LoadQueueReplay 非重发入队时序图 | 344 |
| 16.33 LoadQueueReplay 重发队时序图 | 344 |
| 16.34 Freelist 分配时序图 | 345 |
| 16.35 Freelist 回收时序图 | 345 |
| 16.36 LoadQueueUncache 整体框图 | 347 |
| 16.37 LoadQueueUncache 入队接口时序示意图 | 347 |
| 16.38 LoadQueueUncache 出队接口时序示意图 | 348 |
| 16.39 LoadQueueUncache 与 Uncache 的接口时序示意图 | 348 |
| 16.40 outstanding 时 LoadQueueUncache 与 Uncache 的接口时序示意图 | 349 |
| 16.41 UncacheEntry 有限状态机示意图 | 350 |
| 16.42 LqExceptionBuffer 整体框图 | 351 |
| 16.43 StoreQueue 前递范围生成 | 353 |
| 16.44 StoreQueue 前递数据选择 | 354 |
| 16.45 向量 Store 指令 | 355 |

| | |
|---|-----|
| 16.46 StoreQueue 整体框架 | 358 |
| 16.47 StoreQueue 整体框架 | 358 |
| 16.48 数据更新接口时序 | 359 |
| 16.49 MMIO 接口时序实例 | 359 |
| 16.50 NonCacheable 接口时序实例 | 360 |
| 16.51 CBO 接口时序实例 | 360 |
| 16.52 CMO 接口时序实例 | 361 |
| 16.53 ustate 状态转换示意图 | 362 |
| 16.54 ubuffer 整体框图 | 364 |
| 16.55 outstanding 时 Uncache 与总线的接口时序示意图 | 366 |
| 16.56 sbuffer 的前递示意图 | 368 |
| 16.57 sbuffer 整体框图 | 368 |
| 16.58 接收 store 指令写入时序 | 369 |
| 16.59 写入到 dcache 时序 | 369 |
| 16.60 前递请求时序 | 370 |
| 16.61 alt text | 371 |
| 16.62 alt text | 372 |
| 16.63 alt text | 373 |
| 16.64 alt text | 375 |
| 16.65 alt text | 377 |
| 16.66 alt text | 377 |
| 16.67 alt text | 378 |
| 16.68 alt text | 379 |
| 17.1 DCache 整体架构 | 383 |
| 17.2 LoadPipe 访问 DCache 示意图 | 385 |
| 17.3 LoadPipe 时序 | 386 |
| 17.4 MissQueue 流程图 | 388 |
| 17.5 MissQueue 时序 | 389 |
| 17.6 MissEntry 流程图 | 390 |
| 17.7 ProbeSnoop 流程图 | 391 |
| 17.8 ProbeSnoop 时序 | 392 |
| 17.9 ProbeEntry 状态机 | 392 |
| 17.10 MainPipe 访问 DCache 示意图 | 395 |
| 17.11 MainPipe 时序 | 396 |
| 17.12 WritebackQueue 流程图 | 397 |
| 17.13 WritebackQueue 时序 | 398 |
| 17.14 WriteBackEntry 状态机示意图 | 398 |
| 17.15 CtrlBank 排布 | 399 |
| 17.16 ECCCTL | 399 |
| 17.17 ECCEID | 400 |
| 17.18 ECCMASK | 400 |
| 17.19 Error 架构 | 403 |

| | |
|---|-----|
| 17.20 配置寄存器时序 | 403 |
| 17.21 Tag 注入时序 | 404 |
| 17.22 Data 注入时序 | 405 |
| 18.1 香山处理器的虚拟地址结构 | 408 |
| 18.2 香山处理器的物理地址结构 | 408 |
| 18.3 香山处理器 Sv39x4 的虚拟地址结构（客户机物理地址） | 409 |
| 18.4 Sv39 - Sv39x4 两阶段地址翻译的过程 | 409 |
| 18.5 Sv48 - Sv48x4 两阶段地址翻译的过程 | 410 |
| 18.6 Sfence.vma 的指令格式 | 411 |
| 18.7 Svinval.vma 的指令格式 | 412 |
| 18.8 Hfence 的指令格式 | 412 |
| 18.9 Hinval 的指令格式 | 412 |
| 18.10 MMU 模块整体框图 | 416 |
| 18.11 TLB 压缩示意图 | 426 |
| 18.12 TLB 项示意图 | 427 |
| 18.13 PTW resp 结构示意图 | 428 |
| 18.14 L1 TLB 模块整体框图 | 431 |
| 18.15 Frontend 向 ITLB 发送的 PTW 请求命中 ITLB 的时序图 | 431 |
| 18.16 Frontend 向 ITLB 发送的 PTW 请求未命中 ITLB 的时序图 | 432 |
| 18.17 Memblock 向 DTLB 发送的 PTW 请求命中 DTLB 的时序图 | 433 |
| 18.18 Memblock 向 DTLB 发送的 PTW 请求未命中 DTLB 的时序图 | 433 |
| 18.19 TLB 向 Repeater 发送 PTW 请求的时序图 | 434 |
| 18.20 itlbRepeater 向 ITLB 返回 PTW 回复的时序图 | 434 |
| 18.21 dtlbRepeater 向 DTLB 返回 PTW 回复的时序图 | 435 |
| 18.22 TLB Hint 示意图 | 436 |
| 18.23 Repeater 模块整体框图 | 438 |
| 18.24 itlbRepeater3 及 dtlbRepeater1 与 L2 TLB 的接口时序 | 439 |
| 18.25 多级 itlbRepeater 之间的接口时序 | 439 |
| 18.26 L2 TLB 压缩示意图 1 | 443 |
| 18.27 L2 TLB 压缩示意图 2 | 443 |
| 18.28 L2 TLB 模块整体框图 | 444 |
| 18.29 L2 TLB 模块 hit 通路 | 446 |
| 18.30 L2TLB 与 Repeater 的接口时序 | 448 |
| 18.31 串行化的 Page Cache 查询流程图 | 452 |
| 18.32 Page Table Walker 状态机的状态转移图 | 455 |
| 18.33 Last Level Page Table Walker 状态机的状态转移图 | 459 |
| 18.34 Last Level Page Table Walker 处理 allStage 请求的状态机的状态转移图 | 460 |
| 18.35 Hypervisor Page Table Walker 状态机的状态转移图 | 463 |
| 18.36 Prefetcher 的整体框图 | 466 |
| 18.37 PMP 模块整体框图 | 472 |
| 18.38 PMA 模块整体框图 | 473 |
| 18.39 ITLB 和 L2 TLB PMP 模块的接口时序 | 473 |

| | |
|----------------------------------|-----|
| 18.40 DTLB PMP 模块的接口时序 | 473 |
| 20.1 Acquire 请求处理流程 | 480 |
| 20.2 Snoop 请求处理流程 | 481 |
| 20.3 Cache 别名原理示意图 | 483 |
| 20.4 XSTile 结构框图 | 484 |
| 20.5 CoupledL2 微结构图 | 485 |
| 20.6 RequestBuf | 487 |
| 20.7 目录流水线框图 | 492 |
| 20.8 SinkA | 508 |
| 20.9 SinkC | 509 |
| 20.10 GrantBuffer | 511 |
| 20.11 TXREQ | 513 |
| 20.12 RXRSP | 513 |
| 20.13 RXDAT | 514 |
| 20.14 RXSNP | 515 |
| 20.15 TXDAT | 516 |
| 20.16 TXRSP | 517 |
| 20.17 LinkMonitor | 519 |

表目录

| | |
|---|-----|
| 7.1 术语说明 | 99 |
| 7.2 子模块列表 | 99 |
| 7.3 术语说明 | 112 |
| 7.4 子模块列表 | 113 |
| 7.7 术语说明 | 139 |
| 7.8 子模块列表 | 140 |
| 7.9 术语说明 | 144 |
| 7.10 子模块列表 | 144 |
| 7.11 RobEntry 信号列表 | 145 |
| | |
| 8.1 术语说明 | 152 |
| 8.2 子模块列表 | 152 |
| 8.3 寄存器堆规格 | 158 |
| 8.4 Reg Cache 规格 | 160 |
| 8.5 术语说明 | 163 |
| 8.6 子模块列表 | 163 |
| 8.7 术语说明 | 168 |
| 8.8 子模块列表 | 168 |
| 8.9 现有唤醒配置 1 | 168 |
| 8.10 现有唤醒配置 2 | 169 |
| 8.11 立即数映射 | 171 |
| 8.12 唤醒源独热码映射 (1) | 171 |
| 8.13 唤醒源独热码映射 (1) | 172 |
| | |
| 9.1 术语说明 | 179 |
| | |
| 10.1 fu 术语说明 | 192 |
| 10.2 intExuBlock 中各个 ExeUnit 包含的 Fu | 194 |
| 10.3 fpExuBlock 中各个 ExeUnit 包含的 Fu | 194 |
| 10.4 vfExuBlock 中各个 ExeUnit 包含的 Fu | 195 |
| | |
| 11.1 jmp fu 支持的指令 | 196 |
| 11.2 brh fu 支持的指令 | 196 |
| 11.3 i2f fu 支持的指令 | 197 |
| 11.4 i2v fu 支持的指令 | 197 |
| 11.5 f2v fu 支持的指令 | 197 |
| 11.6 csr fu 支持的指令 | 198 |

| | |
|---|-----|
| 11.7 alu fu 支持的指令 | 198 |
| 11.8 mul fu 支持的指令 | 200 |
| 11.9 div fu 支持的指令 | 200 |
| 11.10 fence fu 支持的指令 | 201 |
| 11.11 bku fu 支持的指令 | 201 |
| 11.12 falu 支持的指令 | 202 |
| 11.13 fmac 支持的指令 | 204 |
| 11.14 fcvt 支持的指令 | 204 |
| 11.15 fDivSqrt 支持的指令 | 205 |
| 11.16 vipu fu 支持的指令 | 206 |
| 11.17 vialuF fu 支持的指令 | 206 |
| 11.18 vppu fu 支持的指令 | 210 |
| 11.19 vimac fu 支持的指令 | 211 |
| 11.20 vidiv fu 支持的指令 | 212 |
| 11.21 vfalu fu 支持的指令 | 212 |
| 11.22 vfma fu 支持的指令 | 214 |
| 11.23 vfdi fu 支持的指令 | 214 |
| 11.24 vfcvt fu 支持的指令 | 215 |
| 11.25 ldu fu 支持的指令 | 216 |
| 11.26 sdu fu 支持的指令 | 216 |
| 11.27 mou fu 支持的指令 | 217 |
| | |
| 12.2 两种浮点加法算法步骤 | 221 |
| 12.3 优化后的双通路浮点加法算法 | 221 |
| 12.4 参数含义及不同精度下的取值 | 226 |
| 12.5 无符号整数乘法三种算法比较 | 227 |
| 12.6 Booth 编码与 PP 真值表 | 228 |
| 12.7 A + B 和与进位真值表 | 231 |
| 12.8 A + B + C 和与进位真值表 | 231 |
| 12.9 A + B + C + D 和与进位真值表 | 232 |
| 12.10 A + B + C 产生 Cout 真值表 | 232 |
| 12.11 A + B + C 产生 Car 真值表 | 232 |
| 12.12 不同输入异或门综合结果 | 233 |
| 12.13 CSA3_2 CSA4_2 综合结果 | 233 |
| 12.14 不同数据格式部分积数量及位宽 | 233 |
| 12.15 不同数据格式部分积 CSA 压缩过程 | 234 |
| 12.16 不同浮点格式使用的 rshift_value 位宽 | 236 |
| 12.17 预缩放因子真值表 | 239 |
| 12.18 标准商选择函数 | 243 |
| 12.19 逻辑修正后的商选择函数 | 243 |
| 12.20 结果为 f32 混合精度格式表 | 245 |
| 12.21 VFALU 操作码 | 249 |
| 12.22 VFALU 接口和含义 | 249 |

| | |
|---|-----|
| 12.23 VFMA 操作码 | 251 |
| 12.24 VFMA 接口和含义 | 251 |
| 12.25 标量除法器计算周期 | 254 |
| 12.26 向量除法器计算周期 | 255 |
| 12.27 VFDIV 接口和含义 | 255 |
| 13.1 术语说明 | 258 |
| 13.2 sbpctl 的定义 | 260 |
| 13.3 spfctl 的定义 | 260 |
| 13.4 slvpredctl 的定义 | 261 |
| 13.5 smblockctl 的定义 | 262 |
| 13.6 srnctl 的定义 | 262 |
| 13.7 mcorepwr 的定义 | 262 |
| 13.8 mflushpwr 的定义 | 263 |
| 13.9 不同特权级访问 CSR 权限检查 | 263 |
| 14.1 术语说明 | 265 |
| 14.2 子模块列表 | 265 |
| 14.3 机器模式性能事件计数禁止寄存器说明 | 268 |
| 14.4 机器模式性能事件计数器访问授权寄存器说明 | 268 |
| 14.5 监督模式性能事件计数器访问授权寄存器说明 | 269 |
| 14.6 监督模式性能事件计数器访问授权寄存器说明 | 269 |
| 14.7 机器模式性能事件选择器说明 | 270 |
| 14.8 昆明湖前端性能事件索引表 | 271 |
| 14.9 昆明湖后端性能事件索引表 | 273 |
| 14.10 昆明湖访存性能事件索引表 | 275 |
| 14.11 昆明湖缓存性能事件索引表 | 278 |
| 14.12 三层 Topdown 性能事件 | 280 |
| 14.13 Topdown 性能事件 | 280 |
| 14.14 机器模式事件计数器列表 | 281 |
| 14.15 监督模式计数器上溢中断标志寄存器 (SCOUNTOVF) 说明 | 282 |
| 14.16 用户模式事件计数器列表 | 282 |
| 15.1 术语说明 | 283 |
| 15.2 参数设计 | 283 |
| 15.3 debug MMIO 地址空间 | 285 |
| 15.4 访存粒度和 trigger 匹配粒度 | 287 |
| 15.5 昆明湖实现的 debug 相关的 csr | 287 |
| 16.3 LoadUnit 请求优先级 | 297 |
| 16.19 LoadQueueReplay 存储信息 | 339 |
| 16.21 StoreQueue 存储的基础信息 | 352 |
| 16.22 StoreQueue 存储的状态信息 | 352 |
| 17.7 MissEntry 状态列表 | 390 |

| | |
|---|-----|
| 17.8 ProbeEntry 状态寄存器含义 | 392 |
| 17.9 WritebackEntry 状态寄存器含义 | 398 |
| 17.10 Bus Error Unit 保存的信息 | 401 |
| 17.11 Tag ECC 错误与 Tag 命中关系 | 401 |
| 18.1 内存管理单元术语说明 | 406 |
| 18.2 SATP 寄存器的格式 | 413 |
| 18.3 HGATP 寄存器的格式 | 413 |
| 18.4 TLB 返回的异常种类 | 414 |
| 18.5 MMU 可能产生的异常以及处理流程 | 414 |
| 18.6 MMU IO 接口列表 | 417 |
| 18.7 L2 TLB IO 接口列表 | 418 |
| 18.8 ITLB 的项配置 | 419 |
| 18.9 ITLB 的请求来源 | 419 |
| 18.10 DTLB 的项配置 | 420 |
| 18.11 DTLB 的请求来源 | 420 |
| 18.12 ITLB 和 DTLB 的回填策略 | 421 |
| 18.13 两阶段翻译模式 | 422 |
| 18.14 ITLB 和 DTLB 的特权级 | 422 |
| 18.15 TLB 压缩前后每项存储的内容 | 427 |
| 18.16 TLB 项的类型 | 427 |
| 18.17 获取 gpaddr 的新增 Reg | 429 |
| 18.18 信号复制情况以及驱动部件 | 441 |
| 18.19 Page Cache 的项配置 | 449 |
| 18.20 Page Cache 项需要存储的信息 | 449 |
| 18.21 页表项的属性位 | 449 |
| 18.22 h 编码说明 | 450 |
| 18.23 页表项中 X、W、R 位可能的组合及含义 | 450 |
| 18.24 PMP 和 PMA 检查模块的对应关系 | 467 |
| 18.25 PMP 和 PMA 检查请求需要提供的相关信息 | 469 |
| 18.26 PMP 和 PMA 检查需要返回的相关信息 | 470 |
| 18.27 PMP 和 PMA 检查可能产生的异常以及处理流程 | 471 |

第一部分

前端

1 BPU

1.1 昆明湖 BPU 模块文档

1.1.1 术语说明

表 1.1 术语说明

| 缩写 | 全称 | 描述 |
|--------|---|--------------------------------|
| BPU | Branch Prediction Unit | 分支预测单元 |
| IFU | Instruction Fetch Unit | 取指单元 |
| FTQ | Fetch Target Queue | 取指目标单元 |
| uFTB | Micro Fetch Target Buffer | 分支目标缓冲 |
| FTB | Fetch Target Buffer | 取指目标缓冲 |
| TAGE | TAgged GEometric length predictor | 一种条件分支预测器 |
| SC | statistical corrector predictor | 一种用于在统计偏向情况下纠正 TAGE 预测的条件分支预测器 |
| ITTAGE | Indirect Target TAgged GEometric length predictor | 一种用于预测间接跳转指令目标地址的分支预测器 |
| RAS | Return Address Stack | 一种用于预测调用指令对应返回指令目标地址的分支预测器 |

1.1.2 设计规格

- 支持一次生成一个分支预测块及其对应预测器附加信息
- 支持无空泡的简单预测
- 支持多种精确预测器及覆盖机制
- 支持训练预测器
- 支持分支历史信息维护及误预测恢复
- 支持 topdown 性能事件的统计

1.1.3 功能描述

1.1.3.1 功能概述

BPU 模块接收来自模块外部后端执行单元及后续流水级的重定向信号，按照地址采用多种预测器为当前 PC 值开始的位置生成预测块及生成该预测块时各预测器内部的 meta 信息，传递给后续取指目标队列（FTQ）存储，

预测块供取指单元 (IFU) 使用, meta 信息供未来训练恢复预测器使用。其中, BPU 模块使用全相连 uFTB 作为 next line predictor, 生成理想条件下延迟仅 1 周期的无空泡简单预测结果, 该结果会被直接作为输出传递到 FTQ。与此同时, 这一基础预测结果还将在 BPU 后续流水线内流动, 供高级预测部件使用以提供更为精确的预测结果。一旦高级预测器在后续流水级的预测结果与已有结果不一致, 就将会使用高级预测器结果作为新的输出更新后续 FTQ 中存储预测块结果并重定向 s0 级 PC, 清空新结果流水级之前的流水级的错误路径结果。针对不同种类的指令预测的信息也有不同, 条件分支指令的目标地址由 uFTB 提供, 需要预测其方向; 无条件直接跳转指令的目标地址由 uFTB 提供, 不需要做任何特别的结果预测; 间接跳转指令的跳转方向不需要预测, 但 uFTB 提供的跳转地址结果并不一定正确, 需要预测。

BPU 内的高级预测器包括 FTB、TAGE-SC、ITTAGE 和 RAS。其中, FTB 负责维护预测块的起始地址, 终止地址, 所含分支指令 PC 地址、类型 (是否 branch、是否 jalr、是否 jal、是否 call、是否 return)、基础的方向结果。TAGE-SC 是条件分支指令的主预测器, ITTAGE 用于预测间接跳转指令。RAS 负责预测 return 类型的间接跳转指令跳转地址。

预测单元内多种预测器使用了分支预测历史作为预测条件, 为提高历史与执行真实轨迹的匹配度, 分支预测历史也会随预测结果做推测更新。全局分支历史在 BPU 顶层使用多个更新源进行更新维护, 维护时会按照 TAGE、SC 和 ITTAGE 预测需要的长度进行维护, 不同分支历史长度的分支历史维护算法一致。具体地, TAGE 使用的分支历史长度有 8、13、32、119; ITTAGE 使用的历史长度有 4, 8, 13, 16, 32; SC 使用的历史长度有 0, 4, 10, 16。分支预测历史在 BPU 模块顶层统一维护, 更新源按照优先级从低到高依次为 s0 阻塞暂存的分支历史、利用 s1 预测结果更新的分支历史、利用 s2 预测结果更新的分支历史、利用 s3 预测结果更新的分支历史和 BPU 外部重定向的分支历史。每个分支历史的具体维护策略为:

s0 阻塞暂存的分支历史: 不进行任何主动的更新, 始终与最新全局折叠历史保持一致。

s1 预测结果更新的分支历史: 利用 s1 阶段的分支预测结果在随流水线传递来的

s0 全局折叠历史上更新。具体地, 将预测结果根据位于的分支预测 slot shift 进全局分支历史, 在 0 号 slot 则直接 shift 进去, 在 1 号 slot 则 shift 进去 0 (slot 0 没有 taken) 和当前预测结果。

s2 预测结果更新的分支历史: 利用 s2 阶段的分支预测结果在随流水线传递来的

s1 全局折叠历史上更新。更新算法与 s1 相同。s2 的更新仅在 s2 预测结果与之前 s1 不同时生效。

s3 更新策略与 s1、s2 相同, 更新仅在 s3 结果与之前 s1 或 s2 不同时生效。

重定向的分支历史更新仅在发生重定向时进行, 根据重定向信息中 addIntoHist 信号情况, 分别将传回的分支历史直接或添加重定向对应分支指令的方向结果后用于更新 BPU 的全局分支历史。

为保证较为准确的预测结果, 各分支预测器都需要不断使用最新的执行结果训练预测器。具体地, 更新的预测块及做出该预测时各预测器内部状态的 meta 信息将会在 FTQ 模块内生成并传递回 BPU 单元供各预测器更新内部状态。

分支预测并不能保证结果正确性, 在预测结果与真实状态不符时, 需要将状态恢复到使用错误预测而更新的状态之前, 主要为分支历史的恢复及预测块起始地址的重定向。

1.1.3.2 分支预测块及 meta 信息生成

1.1.3.2.1 分支预测块思想

分支预测的目的是对执行流中存在的分支指令跳转方向与目标进行预测以在真实执行当前指令前推测地生成后续取指的 PC 范围信息以保证指令的连续供应。

一个分支预测块内包含了本分支预测块有效位 (BranchPredictionBundle.valid)、起始地址、完整预测结果、FTB 项、折叠的分支历史、RAS 预测器栈顶等信息。其中, 完整预测结果在第一流水级来自 uFTB, 后续来自 FTB 等高级预测器, FTB 项来自 uFTB 与 FTB 读出结果。

完整预测结果内记录了分支指令跳转方向、块内记录的分支指令信息是否有效 (slot_valids, 即是否存在该分支指令, 一个 valid 对应一个 slot, 一个 slot 对应一条预测块内的分支指令, 共计 2 个 slot, 其中最后一个 slot 可能记录块内第二条分支指令或一条无条件跳转/间接跳转指令)、分支指令目标、jalr 指令目标、分支指令块内偏移、无跳转时指令块结束地址、结束地址是否有误 (块起始地址大于结束地址, 表明存在 false hit)、最后一条分支指令类型、最后一条指令是否为 RVI 的 call 指令、第二个分支指令 slot 是否记录分支指令而非无条件/间接跳转指令、是否命中等信息。如前所述, 分支预测块内最多出现 2 条分支指令/1 条分支指令 +1 条无条件跳转指令, 当预测块内实际记录指令数超出该限制时, 后续 FTQ 模块会将其拆分。此外, 若分支预测块内不存在分支指令或未超出分支指令数量限制但达到了分支预测块的最大宽度 (32B) 也将被截断。

FTB 项内记录了项是否有效 (FTB 采用直接映射方式, 若读地址不曾被写入则该标记无效)、第一个分支指令 slot、结尾的分支/跳转共用 slot、结束地址、指令类型、最后一条指令为 RVI call 指令、是否总是跳转等信息。FTB 项可被用于生成完整预测结果。

1.1.3.2.2 分支预测块生成

分支预测块中基础的 PC 信息最初由模块外传入的复位地址指定, 随后处理器运行过程中正常情况下不断按照预测块的跳转地址推测更新, 在遇到误预测时, PC 值将依据 redirect 通道所给值更新。next line 的完整预测结果由 uFTB 模块读出。完整预测结果中, FTB 项由 FTQ 模块根据以往训练结果生成, 条件分支指令的跳转方向由 uFTB 生成并在后续由 TAGE-SC 预测器更新, 间接跳转指令的跳转地址由 uFTB 生成并在后续由 ITTAGE 预测器更新, RAS 预测器会针对 return 类型的间接跳转指令覆盖 ITTAGE 的预测结果。被覆盖的结果仅体现在预测器输出, 不会立刻反向反馈给结果被覆盖的预测器更新内部状态。

1.1.3.2.3 meta 信息生成

各预测器为便于自身的更新, 会将做出预测时预测器内部状态信息 (例如作出该预测时命中预测表序号、命中 index) 作为 meta 信息和预测结果一同随流水线传递。

1.1.3.3 无空泡简单预测

uFTB 作为 BPU 的 next line predictor, 为处理器作出无空泡的基础预测以连续生成下一个推测 PC 值。

1.1.3.3.1 uFTB 请求接收

每次 1 阶段请求有效时, 截取传入预测块起始 PC 的 16 到 1 位生成 tag 发送给本模块内的全相连 uFTB 用于读取 FTB 项, FTB 项记录内容如前所述。uFTB 内有 32 项使用寄存器搭建的全相连结构。由于使用寄存器实现, 各项可在当拍根据其中存储数据是否有效及存储的 tag 值是否与传入信息匹配来生成本项是否命中的信号及读出的 FTB 项数据并返回到 uFTB 层次。

1.1.3.3.2 uFTB 数据读取与返回

在当拍, uFTB 存储体已返回命中信号及读出数据。在本阶段将会从返回的命中信号中选出至多一命中项并利用该命中项生成预测结果, 生成完整预测结果的算法在后续 FTB 模块有详细叙述, 这里 uFTB 有一额外补充的 counter 机制, 为 uFTB 内每一项内至多 2 条分支指令增加一个 2 位宽 counter, 若 counter 大于 1 或 FTB 项内 always_taken 有效 (后者机制也存在于 FTB 模块中) 则预测结果为跳转。此外, 本级的命中信号及选出的命中路编号还被作为本预测器的 meta 信息等待其他预测器一起进入 s3 阶段时送出预测器, 随最终预测结果一起存储到 FTQ。本预测器在 2、3 阶段没有其他额外动作。

1.1.3.3.3 uFTB 数据更新

当该预测块对应指令全部提交时，由 FTQ 传入 BPU 一直连到本模块的 update 通道中将包含 FTQ 模块根据指令提交信息更新的 FTB 项。由于全相连 uFTB 全部采用寄存器搭建存储体，写操作不会影响并行的读操作，传来的更新信息将始终用于更新。在更新通道有效时，在当拍将利用传入的更新 pc 值生成 tag 与 uFTB 内现有各项匹配并生成是否匹配及匹配的路信号。在下一拍，若存在已有匹配，将拉高匹配路的写入信号，否则利用伪 LRU 替换算法选出一待替换路拉高对应路写入信号，写入数据即为更新的 FTB 项。

针对每个分支指令的 counter 维护也在 update 通道拉高时一并更新，在 update 通道拉高下一拍，更新 FTB 项内跳转的分支指令及其之前的分支指令对应的 counter。若 taken 则 counter+1，若不 taken 则 counter-1，如达到饱和（0 或全 1）则维持当前值不变。

伪 LRU 算法也需要数据更新，其共有两个数据源，其一为作出预测时命中的路编码，其二为 uFTB 更新时要写入的路编码，若其中任意有效，则利用其信息更新伪 LRU 状态，当都有效时一拍内使用组合逻辑依次使用两信息更新。

1.1.3.4 多种高级预测器及覆盖机制

各高级预测器输出的结果将与之前流水级生成并随流水线传递来的结果（uFTB 的 minimal 结果或之后流水级由 FTB 提供的完成结果）比较，如有不同将以较新结果冲刷流水线。

1.1.3.4.1 Composer

Composer 是一个用于组合多个预测器的模块。在本项目中，其组合了 uFTB、FTB、TAGE-SC、ITTAGE 和 RAS 五个预测器，并对外抽象成了一个三级流水覆盖预测器。Composer 中的各个预测器可以通过写自定义寄存器 sbpctl 来实现开关，可以按需使用预测器。在检测到来自外部的重定向后，Composer 会把重定向请求发送给各预测器，以用于恢复推测更新的元素。在预测块所有指令提交后，Composer 中的各预测器会进行训练。最终，Composer 将三级预测结果输出至 Predictor。各预测器的 meta 信息在本模块拼接在一起传递给 FTQ，FTQ 返回的训练用 meta 信息也在本模块内拆分后发送给各模块。

1.1.3.4.1.1 起始 PC 的配置

Composer 的 IO 接口 io_reset_vector 可以实现起始 PC 的配置。只需要将期望的起始 PC 传递给该 IO 即可。

1.1.3.4.1.2 与预测器的连接

Composer 将 uFTB、FTB、TAGE-SC、ITTAGE 和 RAS 五个预测器连接起来。共有三个分支预测器的流水级，每个预测器的相同流水级从前向后以组合逻辑连接，且每个预测器是固定延迟的，到那个流水级就一定完成预测，所以 Composer 的只需要在对应流水级输出对应预测器的预测结果即可。

1.1.3.4.1.3 预测器的开关

通过 Zicsr 指令，我们可以读写 sbpctl 这一自定义 CSR 来控制 Composer 中的各预测器的使能。sbpctl[6:0] 代表了 {LOOP, RAS, SC, TAGE, BIM, BTB, uFTB} 这七个预测器的使能。其中，高电平代表使能，低电平代表未使能。具体地，spbctl 这一 CSR 的值通过 Composer 的 IO 接口 io_ctrl_* 传入各个预测器，并由各预测器负责使能的实现。当前架构中未加入 Loop 和 BIM 两预测器，因此对应位无效。

1.1.3.4.1.4 重定向的恢复

Composer 通过 io_s2_redirect、io_s3_redirect 和 io_redirect_* 等 IO 端口接收重定向请求。这些请求被发送给其各个预测器，用于恢复推测更新的元素，如 RAS 栈顶项等。

1.1.3.4.2 FTB

FTB 暂存 FTB 项，为后续的 TAGE、ITTADE、SC、RAS 等高级预测器提供更为精确的分支指令位置、类型、目标地址等分支预测块关键信息，也为总是跳转的分支指令提供基础的方向预测。FTB 模块内有一 FTBBank 模块负责 FTB 项的实际存储，模块内使用了一块多路 SRAM 作为存储器。SRAM 规格格式详见后续。

1.1.3.4.2.1 请求接收

0 阶段时，FTB 模块向内部 FTBBank 发送读请求，其请求 pc 值为 s0 传入的 PC。

1.1.3.4.2.2 数据读取与返回

在发送请求的下一拍也就是预测器的 1 阶段，将暂存从 FTB SRAM 中读出的多路信号。

再下一拍也就是预测器的 2 阶段，从暂存数据中根据各路的 tag 和实际请求时由 PC 高位生成 tag 的匹配情况生成命中信号并在命中时选出命中 FTB 数据。若存在 hit 请求，则返回值为选出的 FTB 项及命中的路信息，若未 hit，则输出数据无意义。

FTBBank 模块读出的数据在 FTB 模块内作为 2 阶段的预测结果传递给 BPU 后续预测器的 s2 阶段以获取分支指令类型、PC 信息，此外这一读出的结果还会被暂存到 FTB 模块内，在 3 阶段作为预测结果以组合逻辑传递给后续预测器。若 FTB 命中，则读出的命中路编号与命中信息、周期数等也会随流水线向后传递，最终若该预测块未被流水线中途冲刷，则在 s3 作为 meta 信息传递给后续 FTQ 模块，其中周期数仅在仿真环境用于性能统计，在 FPGA 等环境不存在。

此外，若 FTB 项内记录的有效分支指令存在 always taken 标志，表示该分支指令历史上不曾有非跳转情况，则 2 阶段的预测结果中对应 br_taken_mask 也在本模块内直接拉高处理，直接预测该分支指令跳转，不再使用其他高级预测器的预测结果。

1.1.3.4.3 TAGE-SC

TAGE-SC 是南湖架构条件分支的主预测器，属于精确预测器（Accurate Predictor，简称 APD）。

其中 TAGE 利用历史长度不同的多个预测表，可以挖掘极长的分支历史信息；SC 是统计校正器。

TAGE 由一个基预测表和多个历史表组成，基预测表用 PC 索引，而历史表用 PC 和一定长度的分支历史折叠后的结果异或索引，不同历史表使用的分支历史长度不同。在预测时，还会用 PC 和每个历史表对应的分支历史的另一种折叠结果异或计算 tag，与表中读出的 tag 进行匹配，如果匹配成功则该表命中。最终的结果取决于命中的历史长度最长的预测表的结果。

当 SC 认为 TAGE 有较大的概率误预测时，它会反转最终的预测结果。

在南湖架构中，每次预测最多同时预测 2 条条件分支指令。在访问 TAGE 的各个历史表时，用预测块的起始地址作为 PC，同时取出两个预测结果，它们所用的分支历史也是相同的。

1.1.3.4.3.1 TAGE 预测时序

TAGE 是高精度条件分支方向预测器。使用不同长度的分支历史和当前 PC 值寻址多个 SRAM 表，当在多个表中出现命中时，优先选择最命中的历史长度最长的对应表项的预测结果作为最终结果。

TAGE 需要 2 拍延迟：

- 0 拍生成 SRAM 寻址用 index。index 的生成过程就是把折叠历史和 pc 异或，折叠历史的管理不在 ITTAGE 和 TAGE 内部，而在 BPU
- 1 拍读出结果
- 2 拍输出预测结果

1.1.3.4.3.2 TAGE：折叠历史

TAGE 类预测器的每一个历史表都有一个特定的历史长度，为了与 PC 异或后进行历史表的索引，很长的分支历史序列需要被分成很多段，然后全部异或起来。每一段的长度一般等于历史表深度的对数。由于异或的次数一般较多，为了避免预测路径上多级异或的时延，我们会直接存储折叠后的历史。由于不同长度历史折叠方式不同，所需折叠历史的份数等于 (历史长度, 折叠后长度) 元组去重后的个数。在更新一位历史时只需要把折叠前的最老的那一位和最新的一位异或到相应的位置，再做一个移位操作即可。

1.1.3.4.3.3 TAGE：备选预测逻辑

实现了 USE_ALT_ON_NA 寄存器，动态决定是否在最长历史匹配结果信心不足时使用备选预测。在实现中处于时序考虑，始终用基预测表的结果作为备选预测，这带来的准确率损失很小。

1.1.3.4.3.4 SC：时序

一些应用上，一些分支行为与分支历史或路径相关性较弱，表现出一个统计上的预测偏向性。对于这些分支，相比 TAGE，使用计数器捕捉统计偏向的方法更为有效。TAGE 在预测非常相关的分支时非常有效，TAGE 未能预测有统计偏向的分支，例如只对一个方向有小偏差，但与历史路径没有强相关性的分支。

统计校正的目的是检测不太可靠的预测并将其恢复，来自 TAGE 的预测以及分支信息（地址、全局历史、全局路径、局部历史）被呈现给统计校正预测器，其决定是否反转预测。SC 负责预测具有统计偏向的条件分支指令并在该情形下反转 TAGE 预测器的结果。

SC 的预测算法依赖 TAGE 里面的是否有历史表 hit 的信号 provided，以及 provider 的预测结果 taken，从而来决定 SC 自己的预测。provided 是使用 SC 预测的必要条件之一，provider 的 taken 作为 choose bit，选出 SC 最终的预测，这是因为 SC 在 TAGE 预测结果不同的场景下可能有不同的预测。

SC 需要 3 拍延迟：

- 0 拍生成寻址 index 得到 s0_idx，index 的生成过程就是把折叠历史和 pc 异或，折叠历史的管理不在 ITTAGE 和 TAGE 内部，而在 BPU 里
- 1 拍读出 SCTable 对应 s0_idx 的计数器数据 s1_scResps
- 2 拍根据 s1_scResps 选择是否需要反转预测结果
- 3 拍输出完整的预测结果

1.1.3.4.4 ITTAGE

ITTAGE 接收来自 BPU 内部的预测请求，其内部由一个基预测表和多个历史表组成，每个表项中都有一个用于存储间接跳转指令目标地址的字段。基预测表用 PC 索引，而历史表用 PC 和一定长度的分支历史折叠后的结果异或索引，不同历史表使用的分支历史长度不同。在预测时，还会用 PC 和每个历史表对应的分支历史的另一种折叠结果异或计算 tag，与表中读出的 tag 进行匹配，如果匹配成功则该表命中。最终的结果取决于命中的历史长度最长的预测表的结果。最终，ITTAGE 将预测结果输出至 composer。

1.1.3.4.4.1 间接跳转指令的预测

ITTAGE 用于预测间接跳转指令。普通分支指令和无条件跳转指令的跳转目标直接编码于指令中，便于预测，而间接跳转指令的跳转地址来自运行时可变的寄存器，从而有多种可能选择，需要根据分支历史对其作出预测。为此，ITTAGE 的每个表项在 TAGE 表项的基础上加入了所预测的跳转地址项，最后输出结果为选出的命中预测跳转地址而非选出的跳转方向。由于每个 FTB 项仅存储至多一条间接跳转指令信息，ITTAGE 预测器每周期也最多预测一条间接跳转指令的目标地址。

ITTAGE 需要 3 拍延迟：

- 0 拍生成寻址 index
- 1 拍读出数据
- 2 拍选出命中结果
- 3 拍输出

1.1.3.4.4.2 折叠分支历史

历史表有特定的历史长度，为了与 PC 异或后进行历史表的索引，很长的分支历史序列需要被分成很多段，然后全部异或起来。每一段的长度一般等于历史表深度的对数。由于异或的次数一般较多，为了避免预测路径上多级异或的时延，我们会直接存储折叠后的历史。由于不同长度历史折叠方式不同，所需折叠历史的份数等于(历史长度, 折叠后长度) 元组去重后的个数。在更新一位历史时只需要把折叠前的最老的那一位和最新的一位异或到相应的位置，再做一个移位操作即可。

1.1.3.4.5 RAS

RAS 使用栈结构来预测函数调用与返回这类具有成对匹配特性的执行流的返回地址。其中调用 (push/call) 类指令的特征为目标寄存器地址为 1 或 5 的 jal/jalr 指令。返回 (ret) 类指令的特征为源寄存器为 1 或 5 的 jalr 指令。这类指令为无条件跳转指令，其类型、所在块内偏移量已在 FTB 中读出。

在实现中，RAS 预测器在 s2 和 s3 两阶段提供预测结果。

1.1.3.4.5.1 2 阶段结果

在 2 阶段，由于 s3 阶段还可能存在预测结果需要更新，当前 FTB 项并不一定为最终执行路径，此时作出的预测推测了此时 3 阶段预测结果（也即前一个预测块）不会刷新当前流水级内的预测起始地址。若 2 阶段从 FTB 传来的 FTB 项有效且其中存在 push 类 (call) 指令，则将该指令之后下一指令的 PC 值压入 RAS 栈；若 s2 阶段传来 FTB 项有效且其中存在 pop 类 (return) 指令，则将当前栈顶的地址作为结果返回并对结果出栈。

在 RAS 栈模块内，上述行为分别体现为，在 push 操作时，如当前地址和栈顶地址不同，则压栈一个新项目，其对应计数器为 0，否则将栈顶项的计数器增加 1。两操作都需将这一顶部信息设置为写 bypass 项以供当前

读操作使用。在 pop 操作时，若当前栈顶项计数器为 0，则栈顶指针减 1，若计数器大于 0，则将计数器减 1。为时序优化考虑，写入到 RAS 栈内的数据将会延迟一拍后写入，考虑到可能存在本拍写入的数据下一拍需要获取读数据的情况，设计了写 bypass 机制，准备写入的数据将在本拍首先用于更新写 bypass 相关的项，包括写操作指针及写操作数据。下一拍要求读取的指针若与写 bypass 记录的指针位置匹配，则使用 bypass 值，否则使用真正的栈顶值。

1.1.3.4.5.2 3 阶段结果

在 3 阶段，2 阶段曾发生过的推测 push/pop 操作记录会随流水线传递过来，3 阶段会根据 3 阶段 FTB 项（此时不再存在后续流水级，也即能进入 3 阶段的 FTB 项不会被后续流水级冲刷）结果以与 2 阶段相同逻辑生成 push/pop 控制信号，若发现 2 阶段推测结果与 3 阶段判定结果不同，即 RAS 位于 2 阶段时，当时 3 阶段预测冲刷过 BPU 流水线，则 2 阶段做出的预测结果所基于的情况已经发生变化，对 RAS 栈的操作不正确，需要仿照误预测进行状态恢复，具体细节见后。

1.1.3.5 预测器训练

预测器作出的每一个预测，在其中所有指令都成功提交后会由 FTQ 生成预测块更新信息，与传递到 FTQ 的各预测器 meta 信息一起送回预测器进行训练。

1.1.3.5.1 预测器训练数据接收

从 FTQ 传入的预测器训练数据在 BPU 模 FTB 项、更新 PC 等其他信号一起传递给各预测器。各预测器视其时序压力再暂存 update 信号或立刻进行

1.1.3.5.2 FTB

FTB 项的更新具体逻辑详见 FTQ 模块。

收到 update 请求后，FTB 模块会根据 meta 信息中记录的这一预测做出时原来的读取结果是否 hit 决定更新时机。若 meta 中显示做出预测时 hit，则在本拍立刻更新将新的 FTB 数据写入 SRAM，否则需要延迟 2 周期等待读出 FTB 内现有结果决定写入路后才可更新。

在 FTBBank 内部，当存在更新请求时，该模块行为也因立即更新和推迟更新两情况而有所不同。立即更新时，FTBBank 内的 SRAM 写通道拉高，按照给定的信息完成写入。推迟 2 周期更新时，FTBBank 首先收到一个 update 的读请求且优先级高于普通预测的读请求，而后下一拍读出数据，选出给定地址命中的路编码传递给外部 FTB 模块（命中场景：两次针对这一 FTB 项的请求，第一次请求未命中，它更新之前发生了第二次访问，当前第一次的更新已完成，此更新为第二次对应的更新）。而若这一拍未命中，则下一拍需要写入到在读出 FTB 项后一拍由路选取算法分配的路中。路选取规则为，若所有路均已写满，则使用替换算法（此处为伪 LRU，详见 ICache 文档）选取要替换的路，否则选取一空路。

1.1.3.5.3 Composer

Composer 通过 IO 端口 io_update_* 将训练信号发送给其各个预测器。总的来说，为防止错误执行路径对预测器内容的污染，各部分预测器在预测块的所有指令提交后进行训练。它们的训练内容来自自身的预测信息和预测块中指令的译码结果和执行结果，它们会被从 FTQ 中读出，并送回 BPU。其中，自身的预测信息会在预测后打包传进 FTQ 中存储；指令的译码结果来自 IFU 的预译码模块，在取到指令后写回 FTQ；而执行结果来自各个执行单元。

1.1.3.5.4 TAGE-SC & ITTAGE

表项中包含一个 useful 域，它的值不为 0 表示该项是一个有用的项，便不会被训练时的分配算法当作空项分配出去。在训练时，用一个饱和计数器动态监测分配的成功/失败次数，当分配失败的次数足够多，计数器达到饱和时，把所有的 useful 域清零。

1.1.3.5.5 RAS

当 RAS 在 2 阶段的推测结果与 3 阶段不同或之前的预测结果遇到了 redirect，需要恢复状态。其中，redirect 信息在实际恢复前被暂存到 RAS 内寄存器，延迟一拍再更新。3 阶段检测到不同的下一拍完成更新。若为 redirect 中的 call 误预测（出错指令预译码为 call 类型指令）或 3 阶段的 push 操作不匹配，则进行 recover_push 操作，将原来错误 pop 的 RAS 栈顶重新 push 进去。若为 redirect 中的 ret 误预测（出错指令预译码为 pop 指令）或 3 阶段的 pop 操作不匹配，则进行 recover_pop 操作，弹出 RAS 栈顶本应弹出的地址。栈指针在 redirect 时恢复为 redirect 传来的栈指针，否则为当前值。栈顶在 redirect 恢复时恢复为 redirect 传来的栈顶项，否则为当前值，恢复的新地址在 redirect 时为 redirect 信号下一条指令的值，否则为 2 阶段推测值。

在 RAS 栈内部，状态恢复时同样生成 push、pop 等操作，这类操作的处理方式与前述 2 阶段相同，此处仅列举存在不同的情况。在 push 操作且非分配新项的情况下，若处于 recover 状态，sp、栈顶指针、顶部返回地址要设置为更新值。在 pop 操作且当前栈顶计数器非 0 时，so、栈顶指针、顶部返回地址要设置为更新值。在既非 push 也非 pop 时，需要恢复 sp、栈顶指针、顶部返回地址同时处理写 bypass。

1.1.3.6 分支历史信息维护

1.1.3.6.1 推测更新

在流水级中预测器生成推测结果后，后续请求所用的分支历史也将包含这一推测值以提高预测准确率。

1.1.3.6.2 redirect 恢复

在遇到分支预测错误时，分支历史也被一并恢复到出错状态前，这样可以保证分支历史的准确度。

1.1.3.7 topdown 性能分析事件统计

在 BPU 流水级中预测器生成推测结果可能因为各种原因阻塞，而阻塞可能最终导致处理器整体的流水线空泡。为对性能瓶颈进行较为准确的分析定位，昆明湖架构增加了 topdown 性能计数器，收集流水线中各流水级的空泡/阻塞信息并将指令提交时空泡（实际提交指令数与发射数理想值间差值）归因到具体的模块，从而实现对瓶颈部件的定位。上述性能分析建模方法细节详见 Intel 发表的论文《A Top-Down method for performance analysis and counters architecture》。在 BPU 可以统计到的阻塞事件包括因各预测器误预测恢复所引入的流水线空泡、因后端访存违例恢复所引入的流水线空泡、因 BPU 内部 override 预测刷新较老预测结果所引入的流水线空泡、因分支指令训练预测器而阻塞 BPU 所引入的流水线空泡和因 FTQ 满无法接收新的分支预测块阻塞 BPU 所引入的流水线空泡。在 BPU 内不处理各空泡原因间的优先级，而只是在符合对应的统计条件时将空泡控制信号拉高并随处理器流水线传递。

目前 BPU 内有 FTB（含 uFTB 和主 FTB）、TAGE、SC、ITTAGE 和 RAS 共计 5 种预测器。Topdown 将分支误预测原因细分到以上 5 个预测器。具体地，将每个误预测空泡分解到各预测器的条件为：

1. FTB：发生了分支指令相关的重定向且误预测的指令在对应预测块的 FTB 内并没有记录

2. TAGE: 发生了分支指令相关的重定向且误预测的指令在对应预测块的 FTB 内有记录，但 SC 预测器并未给出对应的预测
3. SC: 发生了分支指令相关的重定向且误预测的指令在对应预测块的 FTB 内有记录，同时 SC 预测器给出了对应的预测结果
4. ITTAGE: 发生了分支指令相关的重定向，误预测的指令为 jalr 指令但不是 return 指令且在 FTB 项中命中
5. RAS: 发生了分支指令相关的重定向。误预测指令为 return 指令（一类特殊的 jalr 指令，详见后续 RAS 模块说明）。

后端访存违例恢复所引入的流水线空泡判断条件为后端发来的重定向信号指示重定向来自访存违例。

BPU 内部 override 引入的流水线空泡有两个可能的来源，分别为 BPU 第 2 和第 3 流水级的重定向信号。

因分支预测器训练而引入的流水线空泡有 3 个可能的来源，分别为 BPU 第 1, 2 和 3 流水线的 ready 信号。

上述 ready 信号为 BPU 内各预测器的 ready 信号取或操作的结果。

因 FTQ 满而引入的流水线空泡判断条件为 BPU 发往 FTQ 模块的握手接口 ready 信号拉低。

对分支误预测和访存违例所导致的空泡，其将被标记在当前 BPU 各流水级的 topdown 信号中。对 BPU 内部 override 引入的空泡，其将被标记在 override 当前所在流水级和更早流水级，而不影响比这一 override 更早的预测块所在流水级。对分支预测器训练所导致的空泡处理类似 BPU override。

1.1.4 总体设计

1.1.4.1 整体框图

1.1.4.2 接口时序

1.1.4.2.1 BPU 到 FTQ 接口时序

上图展示了 BPU 到 FTQ 的预测结果接口时序。图中针对 0xFFFFF80020 起始地址的预测结果在流水线内 1、2、3 阶段分别输出，若结果与之前流水级不一致则 redirect 信号拉高表明需要刷新预测流水线。

1.1.4.2.2 FTQ 到 BPU redirect 接口时序

上图展示了 FTQ 到 BPU 的 redirect 接口时序，redirect 核心信号为 cfiUpdate_target，其指定了 redirect 的目标地址为 0x200000109a。

1.1.4.2.3 FTQ 到 BPU update 接口时序

上图展示了 FTQ 到 BPU 的 update 接口时序，这一更新是为 0x2000000e00 开始预测块准备的，其目标跳转地址为 0x2000000e0e。

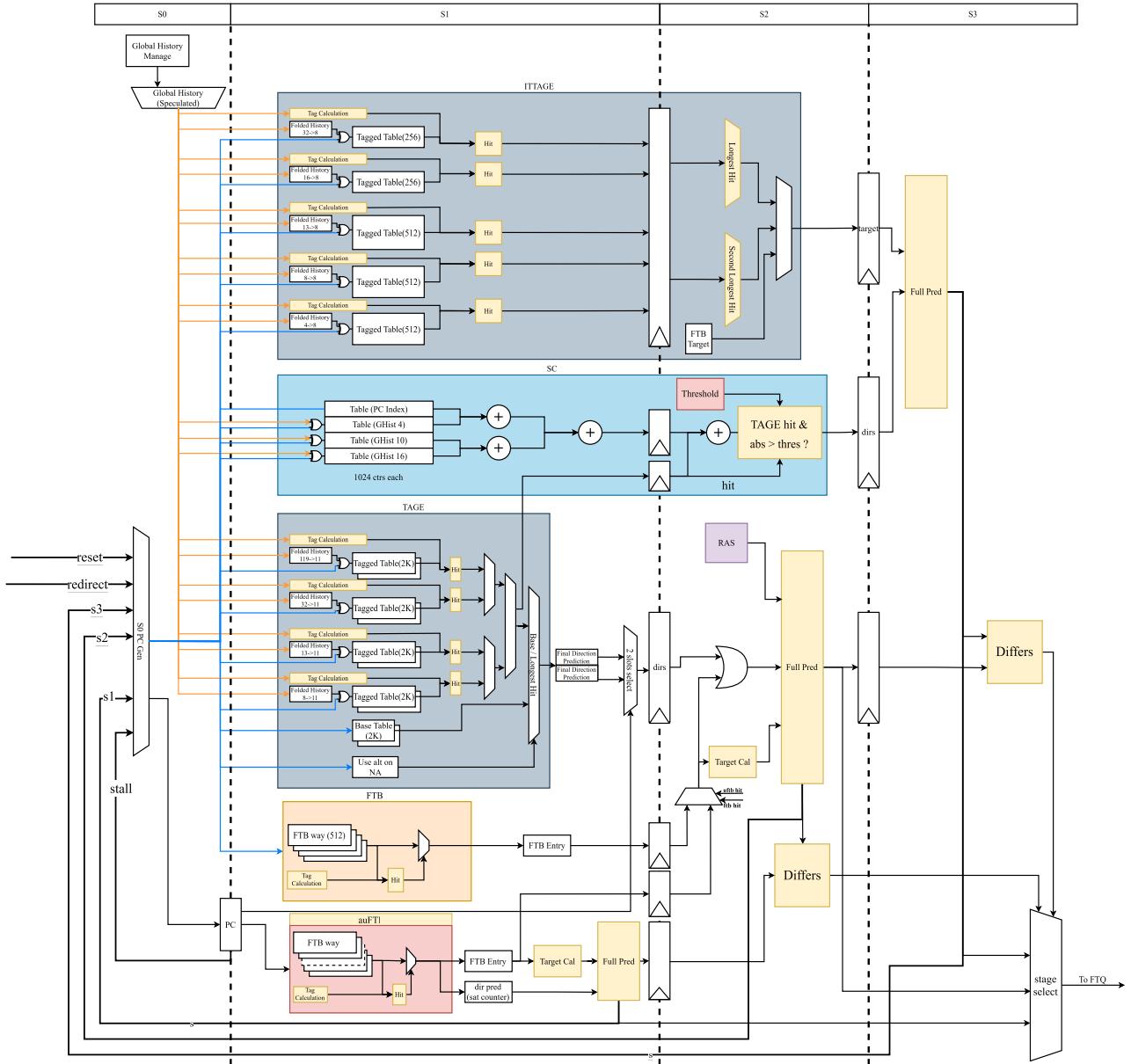


图 1.1: 整体框图

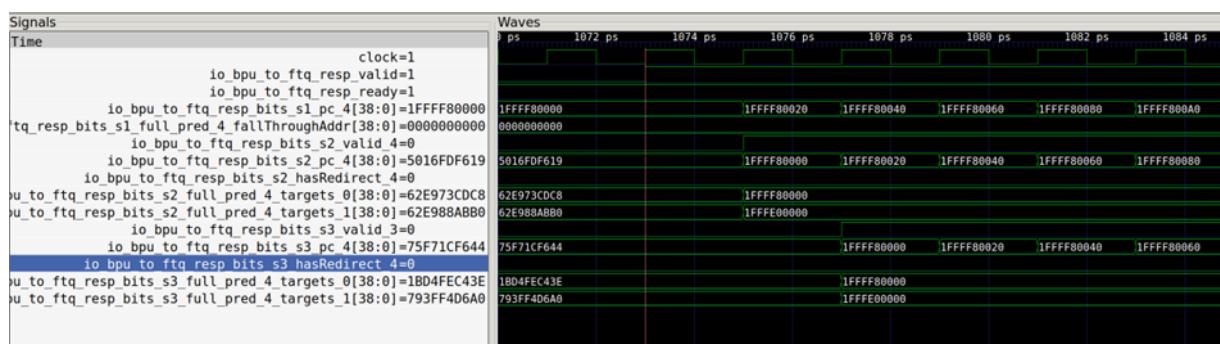


图 1.2: BPU 到 FTQ 接口时序

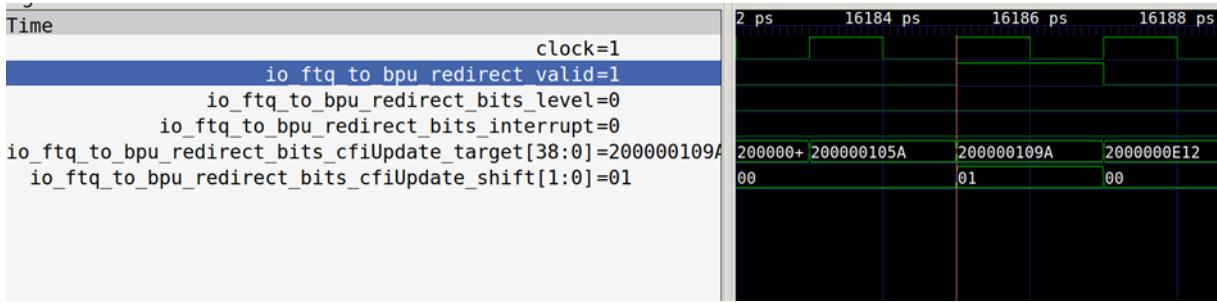


图 1.3: FTQ 到 BPU redirect 接口时序

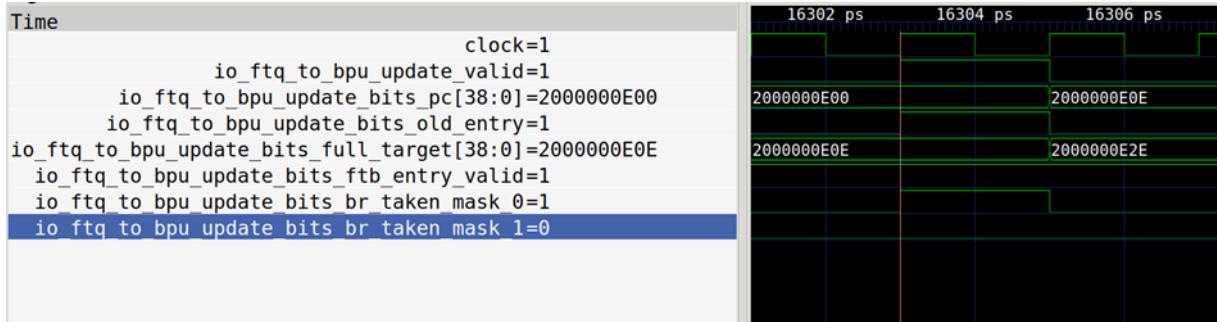


图 1.4: FTQ 到 BPU update 接口时序

1.1.5 寄存器配置

| 寄存器 | 地址 | 复位值 | 属性 | 描述 |
|--------|----------|------|----|---|
| sbpctl | 0x5C064' | d0RW | | bit0: uFTB 使能信号 bit1: FTB 使能信号 bit2: BIM 使能信号（保留） bit3: TAGE 使能信号 bit4: SC 使能信号 bit5: RAS 使能信号 bit6: loop 预测器使能信号（保留） |

注：RO——只读寄存器；RW——可读可写寄存器。

1.1.6 参考文档

1. Reinman G, Austin T, Calder B. A scalable front-end architecture for fast instruction delivery[J]. ACM SIGARCH Computer Architecture News, 1999, 27(2): 234-245.
2. Perais A, Sheikh R, Yen L, et al. Elastic instruction fetching[C]//2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2019: 478-490.
3. Software Optimization Guide for AMD Family 19h Processors (PUB), Chap. 2.8.1.5, <https://www.amd.com/system/files/TechDocs/56665.zip>
4. Seznec A, Michaud P. A case for (partially) TAgged GEometric history length branch prediction[J]. The Journal of Instruction-Level Parallelism, 2006, 8: 23.
5. Seznec A. A 256 kbits l-tage branch predictor[J]. Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2), 2007, 9: 1-6.
6. Seznec A. A new case for the tage branch predictor[C]//Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. 2011: 117-127.

7. Seznec A. The O-GEHL branch predictor[J]. The 1st JILP Championship Branch Prediction Competition (CBP-1), 2004.
8. Jiménez D A, Lin C. Dynamic branch prediction with perceptrons[C]//Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. IEEE, 2001: 197-206.
9. Seznec A. A 64-Kbytes ITTAGE indirect branch predictor[C]//JWAC-2: Championship Branch Prediction. 2011.

1.2 BPU 子模块 Composer

1.2.1 功能概述

Composer 是一个用于组合多个预测器的模块。在南湖中，其组合了 uFTB、FTB、TAGE-SC、ITTAGE 和 RAS 五个预测器，并对外抽象成了一个三级流水覆盖预测器。Composer 中的各个预测器可以通过写自定义寄存器 sbpctl 来实现开关，可以按需使用预测器。在检测到来自外部的重定向后，Composer 会把重定向请求发送给各预测器，以用于恢复推测更新的元素。在预测块所有指令提交后，Composer 中的各预测器会进行训练。最终，Composer 将三级预测结果输出至 Predictor。

三级 BPU 流水级内部重定向的时候如果有预测错误只会恢复那些推测更新的状态，比如说分支历史和 RAS，其它的预测器更新都是在提交后做。

此时如果不刷新预测器，只是刷新流水线，那下次这个地方不是还会预测错误？刷新流水线的同时是从纠正过的正确路径开始预测，如果接下来的路径又经过了同一个地方，是可能再次预测同一个结果的，但是也有可能因为分支历史不同，从而在 TAGE 等预测器里索引不同的表项。

如果在执行时发现目标地址错误，不会发起重定向，而是统一等到指令提交时再重定向。这样设计的一个原因是本身误预测的重定向就是在错误路径上的，它的执行结果可能也是错误的，这种情况下去训练，可能对预测器造成污染。

1.2.1.1 起始 PC 的配置

Composer 的 IO 接口 io_reset_vector 可以实现起始 PC 的配置。只需要将期望的起始 PC 传递给该 IO 即可。

1.2.1.2 与预测器的连接

Composer 将 uFTB、FTB、TAGE-SC、ITTAGE 和 RAS 五个预测器连接起来。因为共有三个分支预测器的流水级，且每个预测器是固定延迟的，到那个流水级就一定完成预测，所以 Composer 只需要在对应流水级输出对应预测器的预测结果即可。

meta 是预测器预测的时候的数据，update 的时候拿回来更新用。都叫 meta 是因为 composer 将所有预测器整合起来，用共同的接口 meta 和外界交互。

1.2.1.3 预测器的开关

通过 Zicsr 指令，我们可以读写 sbpctl 这一自定义 CSR 来控制 Composer 中的各预测器的使能。sbpctl[6:0] 代表了 {LOOP, RAS, SC, TAGE, BIM, BTB, uFTB} 这七个预测器的使能。其中，高电平代表使能，低电平代表未使能。具体地，sbpctl 这一 CSR 的值通过 Composer 的 IO 接口 io_ctrl_* 传入各个预测器，并由各预测器负责使能的实现。

1.2.1.4 重定向的恢复

Composer 通过 io_s2_redirect、io_s3_redirect 和 io_redirect_* 等 IO 端口接收重定向请求。这些请求被发送给其各个预测器，用于恢复推测更新的元素，如 RAS 栈顶项等。

1.2.1.5 预测器训练

Composer 通过 IO 端口 io_update_* 将训练信号发送给其各个预测器。总的来说，为防止错误执行路径对预测器内容的污染，各部分预测器在预测块的所有指令提交后进行训练。它们的训练内容来自自身的预测信息和预测块中指令的译码结果和执行结果，它们会被从 FTQ 中读出，并送回 BPU。其中，自身的预测信息会在预测后打包传进 FTQ 中存储；指令的译码结果来自 IFU 的预译码模块，在取到指令后写回 FTQ；而执行结果来自各个执行单元。

1.2.2 整体框图

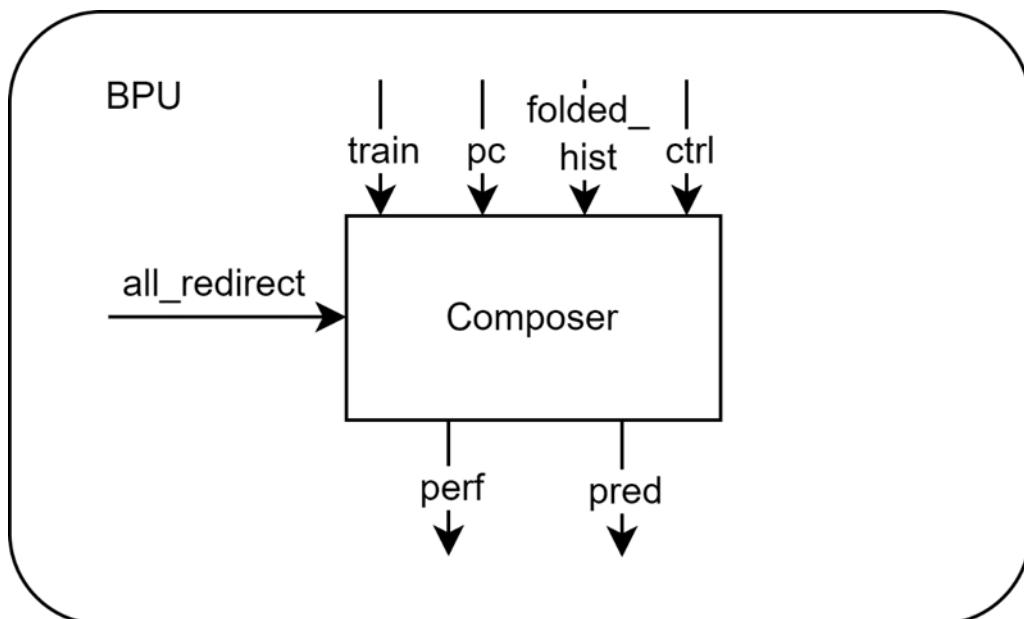


图 1.5: Composer 模块整体框图

1.2.3 接口时序

1.2.3.1 控制信号 Ctrl 接口时序

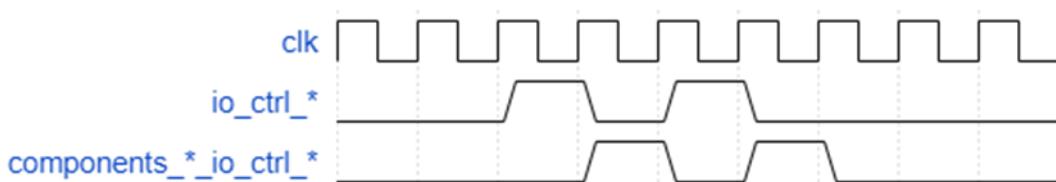


图 1.6: 控制信号 Ctrl 接口时序

上图示意了 Composer 模块控制信号 Ctrl 接口的时序示例，io_ctrl 信号在传入 Composer 模块后，被 delay 一拍传给内部 components 子模块。

1.2.3.2 重定向接口时序

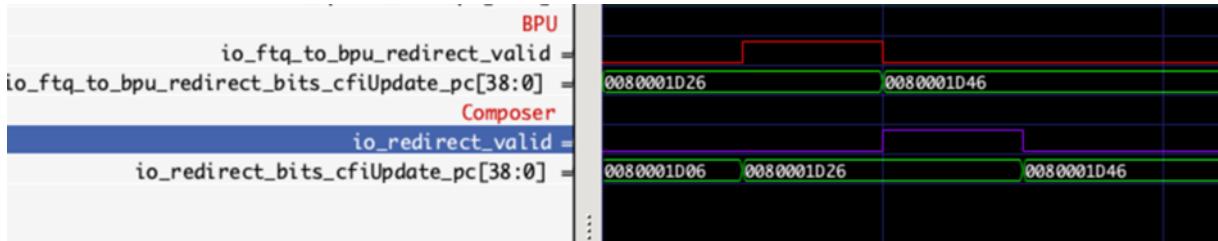


图 1.7: 重定向接口时序

上图展示了 Composer 模块重定向请求的接口，在 BPU 接收到来自后端的重定向请求后，会延迟一拍发往 Composer，因此 Composer 内预测器会晚一拍收到相应请求。

1.2.3.3 分支预测块训练接口时序



图 1.8: 分支预测块训练接口时序

类似重定向，为优化时序，分支预测块训练的 update 接口同样在 BPU 内部被延迟一拍发往 Composer 及其内部各预测器。

1.2.4 关键电路

下图分别展示了 Composer meta 拼接和重定向/分支历史更新来源仲裁逻辑

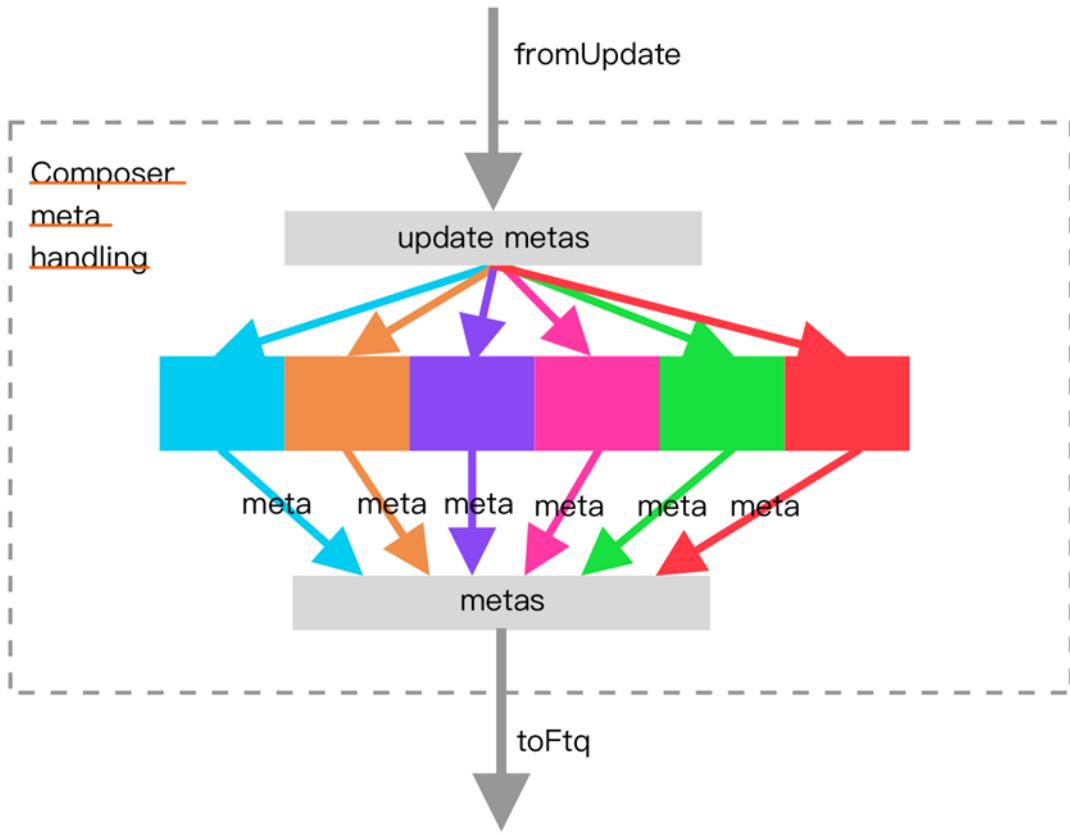


图 1.9: Composer meta 拼接

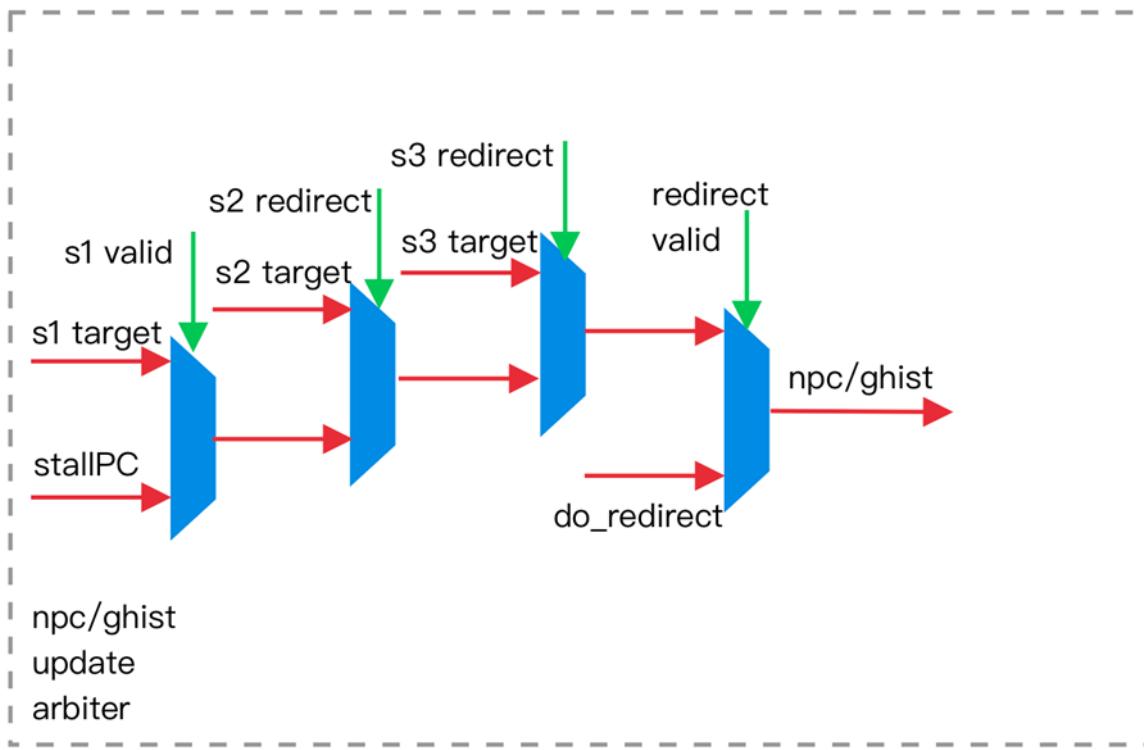


图 1.10: 重定向/分支历史更新来源仲裁逻辑

1.3 BPU 子模块 FTB

1.3.1 功能概述

FTB 暂存 FTB 项，为后续高级预测器提供更为精确的分支指令位置、类型等信息。FTB 模块内有一 FTBBank 模块负责 FTB 项的实际存储，模块内使用了一块多路 SRAM 作为存储器。

1.3.1.1 请求接收

0 阶段时，FTB 模块向内部 FTBBank 发送读请求，其请求 pc 值为 s0 传入的 PC,。

数据读取与返回

在发送请求的下一拍也就是预测器的 1 阶段，将暂存从 FTB SRAM 中读出的多路信号。

再下一拍也就是预测器的 2 阶段，从暂存数据中根据各路的 tag 和实际请求时 tag 的匹配情况生成命中信号并在命中时选出命中 FTB 数据。若存在 hit 请求，则返回值为选出的 FTB 项及命中的路信息，若未 hit，则输出数据无意义。tag 为 PC 的 29 到 10 位。

FTBBank 模块读出的数据在 FTB 模块内作为 2 阶段的预测结果以组合逻辑连线形式在当拍传递给后续预测器，此外这一读出的结果还会被暂存到 FTB 模块内，在 3 阶段作为预测结果再次以组合逻辑连线传递给后续预测器。若 FTB 命中，则读出的命中路编号也会作为 meta 信息在 s3 与命中信息、周期数一起传递给后续 FTQ 模块。

此外，若 FTB 项内存在 always taken 标志，则 2 阶段的预测结果中对应 br_taken_mask 也在本模块内拉高处理。

1.3.1.2 数据更新

收到 update 请求后，FTB 模块会根据 meta 信息中是否 hit 决定更新时机。若 meta 中显示 hit，则在本拍立刻更新，否则需要延迟 2 周期等待读出 FTB 内现有结果后才可更新。

在 FTBBank 内部，当存在更新请求时，该模块行为也因立即更新和推迟更新两情况而有所不同。立即更新时，FTBBank 内的 SRAM 写通道拉高，按照给定的信息完成写入。推迟更新时，FTBBank 首先收到一个 update 的读请求且优先级高于普通预测的读请求，而后下一拍读出数据，选出给定地址命中的路编码传递给外部 FTB 模块。而若这一拍未命中，则下一拍需要写入到分配的路中。路选取规则为，若所有路均已写满，则使用替换算法（此处为伪 LRU，详见 ICache 文档）选取要替换的路，否则选取一空路。

1.3.1.3 SRAM 规格

单 bank, 512 set, 4 way, 使用单口 SRAM, 无读保持, 有上电复位。

20 bit tag, 60 bit FTB 项。

其中 FTB 项

1 bit valid

20 bit br slot (4 bit offset, 12 bit lower 2 bit tarStat, 1bit sharing, 1 bit valid)

28 bit tail slot (4 bit offset , 20 bit lower, 2 bit tarStat, 1 bit sharing, 1 bit valid)

4 bit pftAddr

1 bit carry

1 bit isCall

1 bit isRet

1 bit isJalr

1 bit 末尾可能为 rvi call

2 bit always taken

1.3.2 整体框图

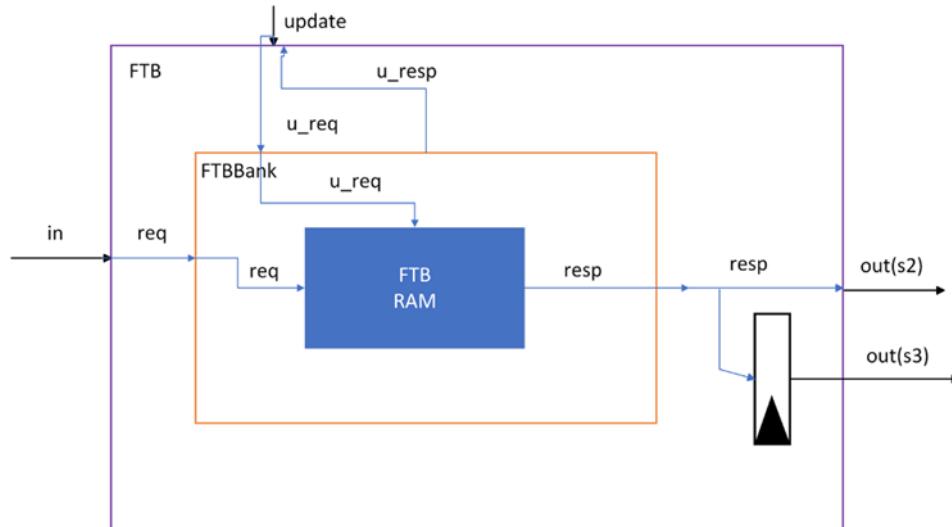


图 1.11: 整体框图

1.3.3 接口时序

1.3.3.1 结果输出接口

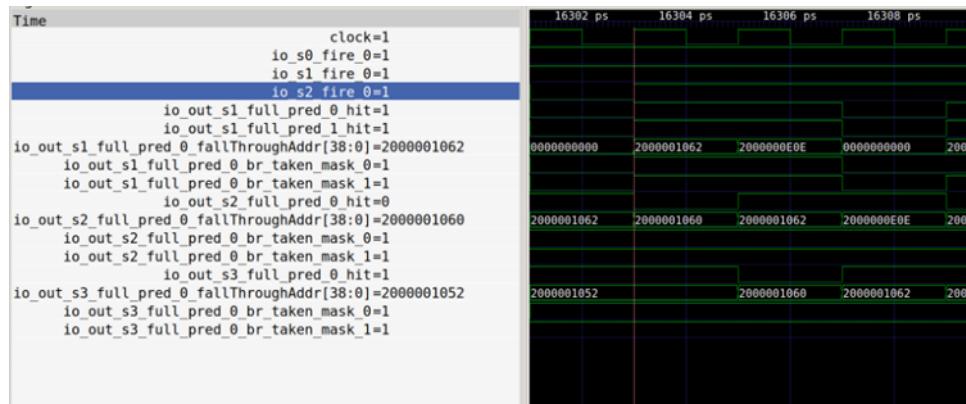


图 1.12: 结果输出接口

上图展示了分支预测器中 FTB 模块针对 fallThrough 地址为 0x2000001062 的请求连续三拍在分支预测器不同阶段输出预测结果的接口。

1.3.3.2 更新接口

上图展示了 FTB 模块的一次针对 0x2000000E00 地址的更新操作，所有更新数据在一拍内全部传递。

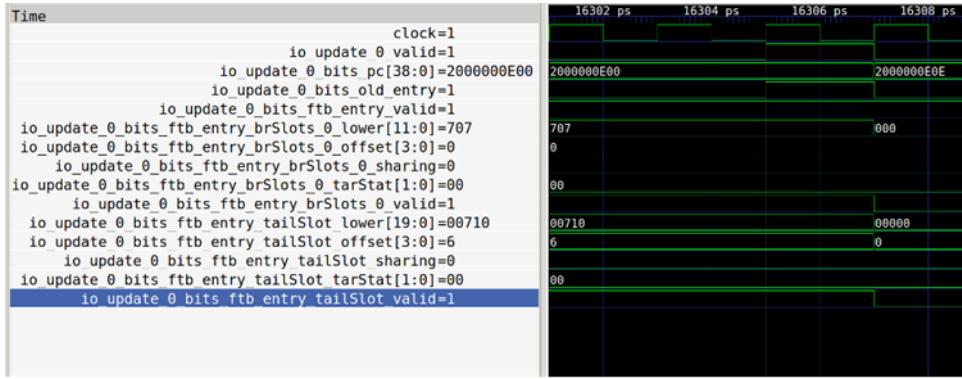


图 1.13: 更新接口

1.3.4 FTBBank

1.3.4.1 接口时序

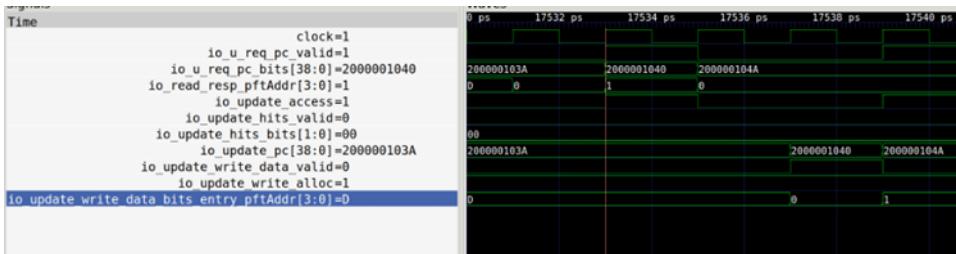
1.3.4.1.1 读数据接口



图 1.14: 读数据接口

上图展示了 FTBBank 读数据接口, FTBBank 在收到请求一拍后回复数据, 即 16303ps 处回复的为 16301ps 的 0x2000001060 地址请求。

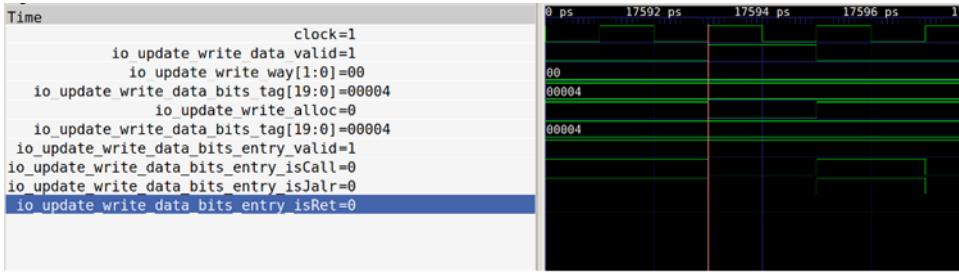
1.3.4.1.2 更新读数据接口



上图展示了 FTBBank

更新读数据接口, FTBBank 在收到更新读请求一拍后回复数据, 回复的数据被外部在一拍后用于更新写数据, 可以注意到请求一拍后的 `pftAddr` 被用于结果读出一拍后的数据写入。

1.3.4.1.3 更新写数据接口



上图展示了 FTBBank

更新写数据接口，在收到写请求后一拍，数据完成写入。

1.3.4.2 功能概述

如上所述，FTBBank 主要存储 FTB 项，为 SRAM 模块的简单封装。

1.3.5 FTB 项的生成条件简述

FTB 是 BPU 的核心。BPU 的其他预测部件所作出的预测全部依赖于 FTB 提供的信息。FTB 除了提供预测块内分支指令的信息之外，还提供预测块的结束地址。对于 FTB 来说，FTB 项的生成策略至关重要。南湖架构在原始论文 1 的基础上，结合这篇论文 2 的思想形成了现有的策略，记 FTB 项的起始地址为 start，结束地址为 end，具体策略如下：

- FTB 项由 start 索引，start 在预测流水线中生成，实际上，start 基本遵循如下原则之一：
 - start 是上一个预测块的 end
 - start 是来自 BPU 外部的重定向的目标地址；
- FTB 项内最多记录两条分支指令，其中第一条一定是条件分支；
- end 一定满足三种条件之一：
 - end - start = 预测宽度
 - end 是从 start 开始的预测宽度范围内第三条分支指令的 PC
 - end 是一条无条件跳转分支的下一条指令的 PC，同时它在从 start 开始的预测宽度范围内

这种训练策略下，同一条分支指令可能存在于多个 FTB 项内。

和论文中的实现1一样，我们只存储结束地址的低位，而高位用起始地址的高位拼接得到。和 AMD3 的做法相似，我们还对 FTB 项中的条件分支指令记录“总是跳转”位，该位在第一次遇到该条件分支跳转时置 1，在其值为 1 的时候，该条件分支的方向总是预测为跳转，也不用它的结果训练条件分支方向预测器；当该条件分支遇到一次执行结果为不跳转的时候，将该位置 0，之后它的方向由条件分支方向预测器预测。

1.3.6 FTB 存储结构

FTB 项结构如下

| total | valid | brSlot | tailSlot | pftAddr | carry | isJalr | isCall, isRet, last_may_be_rvsiable_bias |
|-------------------|------------------|-------------------|-------------------|---------------------|------------------------|--------------------|---|
| 有 效 位 62 | 第一条分 支信息 1 | 第二条分 支信息 21 | 预测块结 束地址 29 | 结束地址高位 是否进位 4 | tailSlot 分支 类型 1 | RAS 标识特殊 位 1 | 强 bias 位 2 |
| | | | | | | | |

FTB slot 的组成，每个 slot 对应一条分支指令

| total | valid | offset | lower | tarStat | sharing | isRVC |
|------------------|---------------------|---------------------|---------------------|--------------------------------------|--------------------------------------|------------------|
| 有效 位 21/29 | 相对起始 PC 的偏移 1 | 目标地址 低位 12/20 | 目标地址高位是 否进位 2 | (对 tailSlot 来说) 是否装了 条件分支 1 | (对 tailSlot 来说) 是否装了 条件分支 1 | 是否是压缩 指令 1 |
| | | | | | | |

FTB 共有 2048 项，4 路组相联，每项最多记录 2 条分支，其中第一条一定是条件分支，第二条可能是任意类型分支指令

1.3.7 目标地址生成逻辑

对于每个 slot，根据三种可能的高位进位情况（进位/退位/不变），在（PC 高位 +1, PC 高位-1, PC 高位）三种情况下选择一个，和存储的目标地址低位信息进行拼位

1.3.8 更新流程

1. 表项生成

1.1 从 FTQ 读取必要信息：

- 起始地址 startAddr
- 预测时读出的旧 FTB 项 old_entry
- 包含 FTQ 项内 32Byte 内所有分支指令的预译码信息 pd
- 此 FTQ 项内有效指令的真实跳转结果 cfiIndex，包括是否跳转，以及跳转指令相对 startAddr 的偏移
- 此 FTQ 项内分支指令（如跳转）的跳转地址（执行结果）
- 预测时 FTB 是否真正命中（旧 FTB 项是否有效）
- 对应 FTQ 项内所有可能指令的误预测 mask

1.2 FTB 项生成逻辑：

- 情况 1：FTB 未命中或存在错误
 - 1) 无条件跳转指令处理：
 - 不论是否被执行，都一定会被写入新 FTB 项的 tailSlot
 - 如果最终 FTQ 项内跳转的指令是条件分支指令，写入新 FTB 项的第一个 brSlot，将对应的 always_taken 位置 1

2) pftAddr 设置:

- 存在无条件跳转指令时: 以第一条无条件跳转指令的结束地址设置
- 无无条件跳转指令时: 以 startAddr+ 取指宽度 (32B) 设置
- 特殊情况: 当 4Byte 宽度的第一条无条件跳转指令起始地址位于 startAddr+30 时, 虽然结束地址超出取指宽度范围, 仍按 startAddr+32 设置

3) carry 位根据 pftAddr 的条件同时设置

4) 设置分支类型标志:

- isJalr、isCall、isRet 按照第一条无条件跳转指令的类型设置
- 特殊标志: 当且仅当 4Byte 宽度的第一条无条件跳转指令起始地址位于 startAddr+30, 且该指令是 call 类型时, 置 last_may_be_rvi_call 位

· 情况 2: FTB 命中且无错误

1) 插入新条件分支:

- 有空闲位置时:
 - a) tailSlot 有无条件跳转: 新条件分支一定指令序先于该无条件跳转, 直接插入 brSlot
 - b) brSlot 有条件分支: 按指令序排列, 保持 FTB 项内 brSlot 的分支在指令序上先于 tailSlot
 - c) 以上情况 pftAddr 均无需修改
- 无空闲位置时:
 - a) tailSlot 有无条件跳转:
 - * 新条件分支指令序一定比该无条件跳转靠前
 - * 新条件分支替代无条件跳转的位置
 - * pftAddr 按无条件跳转的 PC 设置
 - b) tailSlot 有条件分支:
 - c) 新条件分支指令序比 tailSlot 内已有的条件分支指令序靠前: - 将 brSlot 内的条件分支和新条件分支按指令序排布 - pftAddr 按 tailSlot 内原有分支的 PC 设置
- ii) 新条件分支指令序位于已有的所有分支指令后面: - slot 不发生任何改变 - 只把 pftAddr 按新条件分支的 PC 设置

2) 更新 jalr 跳转地址信息:

- 当 tailSlot 内原来记载了 jalr (RISC-V 的无条件间接跳转指令)
- 跳转地址改变时, 修改 tailSlot 内记载的相应目标地址低位和高位进位信息

3) 更新 always_taken 位:

- 如果 always_taken 位置 1, 且对应的条件分支此次执行结果是不跳转, 拉低 always_taken 位

2. 写入 SRAM

2.1 写入条件:

- 新 FTB 项完全没有变化, 或者虽然 FTB 未命中但 uFTB 命中: 不需写入
- 新 FTB 项有变化且非 uFTB 命中、FTB 未命中的情况: 需要写入

2.2 写入流程:

· 情况 1: 预测未命中

1) 先用一拍做一次 FTB 读, 判断此时命中情况

- 2) 如命中，写入对应路
 - 3) 如仍未命中，根据替换算法选择一路写入
 - 4) 总流程需要 3 个时钟周期
 - 5) 过程中由于 FTB 读写口占用，需要 FTQ 配合不发出新的更新请求
 - 注：分 bank 可能提高更新带宽
- 情况 2：预测时命中（包括命中项内有错误信息的情况）
 - 1) 直接写入对应路，无需再次读出

3. 写入 SRAM 的流水线示意图如下：

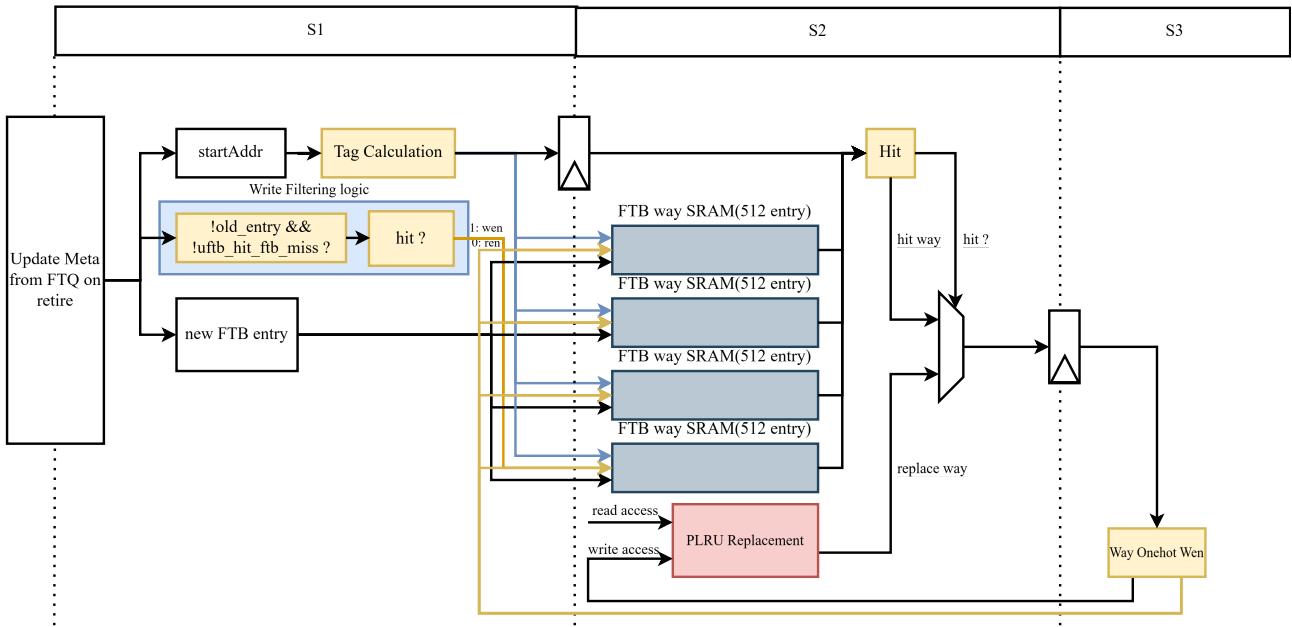


图 1.15: 写入 SRAM 的流水线

1.4 BPU 子模块 uFTB

1.4.1 功能概述

uFTB 作为 BPU 的 next line predictor，为处理器作出无空泡的基础预测以连续生成下一个推测 PC 值。

1.4.1.1 uFTB 请求接收

每次 0 阶段请求有效时，截取传入预测块起始 PC 的 16 到 1 位生成 tag 发送给本模块内的全相连 uFTB 用于读取 FTB 项，FTB 项记录内容如前所述。uFTB 各 bank 内均有 32 项使用寄存器搭建的全相连结构。由于使用寄存器实现，各项可在当拍根据其中存储数据是否有效及存储的 tag 值是否与传入信息匹配来生成本项是否命中的信号及读出的 FTB 项数据并返回到 uFTB 层次，但这一数据并不在本拍而是在下一拍使用。

1.4.1.2 uFTB 数据读取与返回

在下一拍，uFTB 存储体已返回命中信号及读出数据。预测器进入 1 阶段。在本阶段将会从返回的命中信号中选出至多一命中项并利用该命中项生成预测结果，生成完整预测结果的算法在后续 FTB 模块有详细叙述，这

里 uFTB 有一额外补充的 counter 机制，为 uFTB 内每一项内至多 2 条分支指令增加一个 2 位宽 counter，若 counter 大于 1 或 FTB 项内 always_taken 有效（后者机制也存在于 FTB 模块中）则预测结果为跳转。此外，本级的命中信号及选出的命中路编号还被作为本预测器的 meta 信息等待其他预测器一起进入 s3 阶段时送出预测器，随最终预测结果一起存储到 FTQ。本预测器在 2、3 阶段没有其他额外动作。

1.4.1.3 uFTB 数据更新

当该预测块对应指令全部提交时，由 FTQ 传入 BPU 一直连到本模块的 update 通道中将包含 FTQ 模块根据指令提交信息更新的 FTB 项。由于全相连 uFTB 全部采用寄存器搭建存储体，写操作不会影响并行的读操作，传来的更新信息将始终用于更新。在更新通道有效时，在当拍将利用传入的更新 pc 值生成 tag 与 uFTB 内现有各项匹配并生成是否匹配及匹配的路信号。在下一拍，若存在已有匹配，将拉高匹配路的写入信号，否则利用伪 lru 替换算法选出一待替换路拉高对应路写入信号，写入数据即为更新的 FTB 项。

针对每个分支指令的 counter 维护也在 update 通道拉高时一并更新，在 update 通道拉高下一拍，更新 FTB 项内跳转的分支指令及其之前的分支指令对应的 counter。若 taken 则 counter+1，若不 taken 则 counter-1，如达到饱和（0 或全 1）则维持当前值不变。

伪 LRU 算法也需要数据更新，其共有两个数据源，其一为作出预测时命中的路编码，其二为 uFTB 更新时要写入的路编码，若其中任意有效，则利用其信息更新伪 LRU 状态，当都有效时一拍内使用组合逻辑依次使用两信息更新。

1.4.1.4 SRAM 规格

模块内不使用 SRAM，但存在较多的 reg 拼接结构，列举如下。

模块内含有 32 路数据，每一路包含 2 个 2 bit 宽饱和 counter，记录基础的分支方向预测；一个 60 bit FTB 项，具体含义与 FTB 模块相同，详见 FTB SRAM 规格描述；一个 16 bit tag 及一个 1 bit 路有效信号。

1.4.2 整体框图

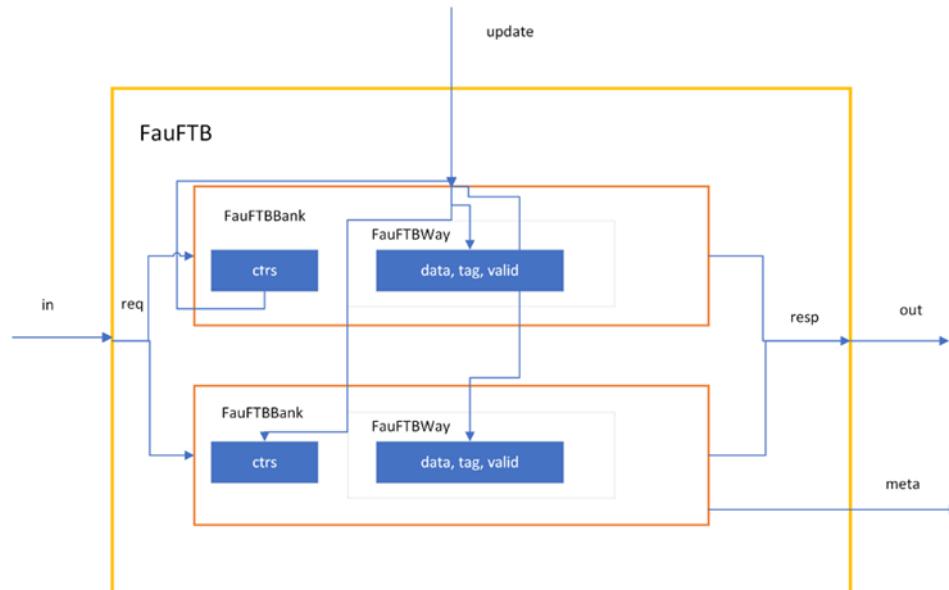


图 1.16: 整体框图

TODO: 图与昆明湖不符，待更新

1.4.3 接口时序

1.4.3.1 结果输出接口

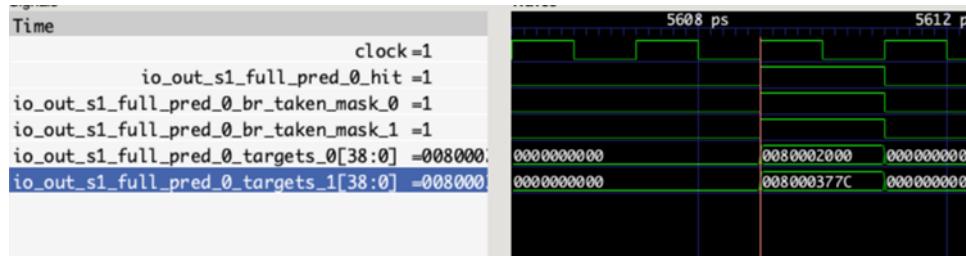


图 1.17: 结果输出接口

上图展示了一次有效的 uFTB 输出，其下一预测块起始地址为 0x80002000。

1.4.3.2 更新接口

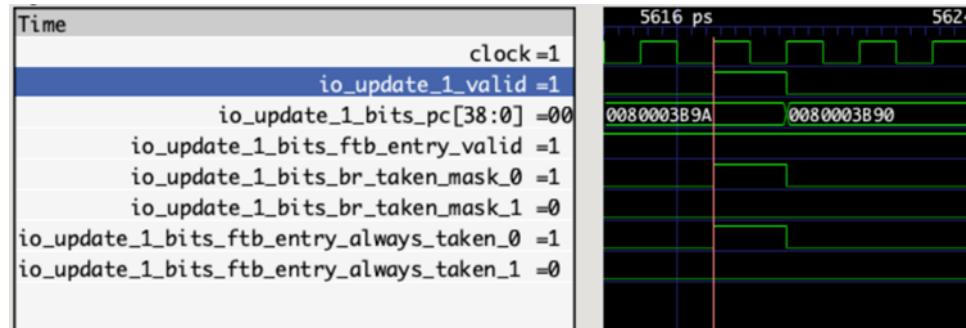


图 1.18: 更新接口

上图展示了一次有效的更新请求，更新了 0x80003b9a 地址的 FTB 项。

1.5 BPU 子模块 TAGE-SC

1.5.1 功能

TAGE-SC 是南湖架构条件分支的主预测器，属于精确预测器 (Accurate Predictor, 简称 APD)。其中 TAGE 利用历史长度不同的多个预测表，可以挖掘极长的分支历史信息；SC 是统计校正器。

TAGE 由一个基预测表和多个历史表组成，基预测表用 PC 索引，而历史表用 PC 和一定长度的分支历史折叠后的结果异或索引，不同历史表使用的分支历史长度不同。在预测时，还会用 PC 和每个历史表对应的分支历史的另一种折叠结果异或计算 tag，与表中读出的 tag 进行匹配，如果匹配成功则该表命中。最终的结果取决于命中的历史长度最长的预测表的结果。

当 SC 认为 TAGE 有较大的概率误预测时，它会反转最终的预测结果。

在南湖架构中，每次预测最多同时预测 2 条条件分支指令。在访问 TAGE 的各个历史表时，用预测块的起始地址作为 PC，同时取出两个预测结果，它们所用的分支历史也是相同的。

1.5.1.1 TAGE: 设计思路

TAGE 作为一种混合式预测器，其优势在于可以同时根据不同长度的分支历史序列，对某一个分支指令分别进行分支预测，并且对在该分支指令在各个历史序列下的准确率进行评估，并且选择历史准确率最高者作为最终分支预测的判断标准。

相较于传统的混合式预测器，TAGE 兼具下两个新的设计特性，使得其预测准确率得到可观的提升：

- 对于每个预测表中的表项添加了 tag 数据。在传统的优先分支预测器中，往往仅采用分支历史以及当前分支指令的 PC 值作为依据来对预测表进行取指。这种情况会导致多条不同的分支指令指向同一个预测表表项的情况 (aliasing) 产生，而这种情况对混合式预测器中采取的分支历史长度较短的部分预测表所给出的分支预测准确率影响尤为显著。因此，在 TAGE 的设计中，通过 partial tagging 的方法，可以更好地将当前指令与预测表中的表项进行实际的匹配，从而较大程度地避免上述情况的产生。
- 采用几何级数变化的分支历史长度来索引不同的预测表，从而使得在分支预测过程中对各个预测表中的表项选择过程的粒度得到了很大的提升。除此之外，在每个表项中还添加了一个 usefulness 计数器，用于记录该表项对于分支预测的有用程度。通过这种设计，各类分支指令都有更高的几率藉由预测准确率最高的分支历史长度进行索引来做出最终的分支预测。

以上两个设计特性使得 TAGE 预测器既不会因为所选取的历史分支长度过短而使得某个预测表中的表项同时映射到多条不同的指令从而降低表项的信息有效程度，也不会因为选取的历史长度分支过大而使得整个 TAGE 需要经过很长时间的更新之后才能够发挥有效的预测功能。因此，TAGE 所可采用的分支历史长度范围非常大，可以用于判断各种代码语境下的分支指令。

1.5.1.2 TAGE: 硬件实现

TAGE 是高精度条件分支方向预测器。使用不同长度的分支历史和当前 PC 值寻址多个 SRAM 表，当在多个表中出现命中时，优先选择最命中的历史长度最长的对应表项的预测结果作为最终结果。

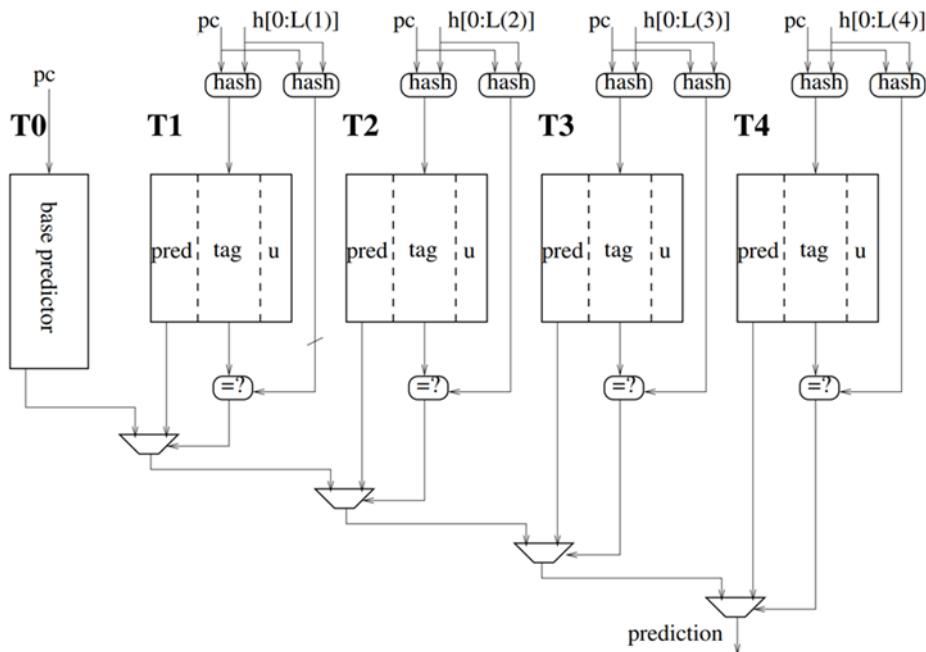


图 1.19: TAGE 原理

TAGE 的结构原理图如上图所示，提供了一个基准预测器 T0 和四个带 tag 的预测表 T1-T4，基准预测器和带 tag 的预测表的基本信息见下表。

| 预测器 | 是否带 tag | 作用 | 表项构成 | 项数 |
|-----------|---------|----------------------------------|--|----------------|
| 基准预测器 T0 | 否 | 用于在四个局部 tag 预测表的 tag 均不匹配时提供预测结果 | ctr 2bits (最高位给出预测结果, 1 跳转, 0 不跳转) | set 2048way |
| 预测表 T1-T4 | 是 | 存在 tag 匹配时, 选历史长度最长者提供预测结果 | valid 1bit tag 8bits ctr 3bits (最高位给出预测结果, 1 跳转, 0 不跳) us: 1bit (usefulness 计数器) | T1-T4 各 4096 项 |

注：上表中的 way 是 chisel 里 SRAMTemplate 类的参数名字，这个参数的值表达在 SRAM 里面存储几份同样类型的数据，不一定代表通常意义上的需要 tag 匹配的 way。T0 有两个 way 是因为预测块内最多两条分支指令，不同的 way 是为了分别对预测块内两条不同的分支指令进行预测。

每个预测块的两条跳转指令的预测内容是分开存储、分开 update 的。带 tag 的预测表 T1-T4 的 4096 项要平均分成两个 bank 分给预测块内最多两条分支，每个 bank 有 2048 项，所以预测表 T1-T4 的 index 只有 11bit，是 \log_2 (预测表的项数/2)。同一个预测块中的两条分支指令的 index 是相同的，但是索引的是两个不同的 bank，一次读出来的来自两个 bank 代表两条预测信息的结果可能会出现一个是 hit，一个没有 hit。

TAGE 类预测器的每一个预测表都有特定的历史长度。为了让原本很长的全局分支历史表能够与 PC 异或后进行预测表的索引或 tag 匹配，原本很长的分支历史序列需要被分成很多段，然后全部异或起来。每一段的长度一般等于历史表深度的对数。由于异或的次数一般较多，为了避免预测路径上多级异或的时延，我们会直接存储折叠后的历史。

每个预测表对应的折叠分支历史都有三份，一份用于索引预测表，两份用于 tag 匹配。在 BPU 模块中维护了一个 256 比特的全局分支历史 g hv，并为 TAGE 的 4 个带 tag 的预测表分别维护了各自的折叠分支历史。具体配置见下表（见 TagTableInfos），其中 g hv 是一个循环队列，“低” n 位指的是从 ptr 指示的位置算起的低位：

| 历史 | index 折叠分支历史长度 | tag 折叠分支历史 1 长度 | tag 折叠分支历史 2 长度 | 设计原理 |
|-------------|----------------|-----------------|-----------------|-------------------------|
| 全局分支历史 g hv | 256 比特（非折叠） | 无 | 无 | 每比特代表对应分支跳转与否 |
| T1 对应折叠分支历史 | 8 比特 | 8 比特 | 7 比特 | ghv 取 ptr 起低 8 比特折叠异或 |
| T2 对应折叠分支历史 | 11 比特 | 8 比特 | 7 比特 | ghv 取 ptr 起低 13 比特折叠异或 |
| T3 对应折叠分支历史 | 11 比特 | 8 比特 | 7 比特 | ghv 取 ptr 起低 32 比特折叠异或 |
| T4 对应折叠分支历史 | 11 比特 | 8 比特 | 7 比特 | ghv 取 ptr 起低 119 比特折叠异或 |

如上图所示，出于时序考虑，折叠分支历史的具体实现方式并不是设计原理中的折叠，而是把折叠前的分支历史最老的那一位（图中对应 h[12]）和最新的一位（图中对应 h[0]）异或到相应的位置（图中对应 c[1] 和 c[4]），再做一个移位操作（图中对应变为 c[0] 和 c[2]）即可，伪代码如下所示：

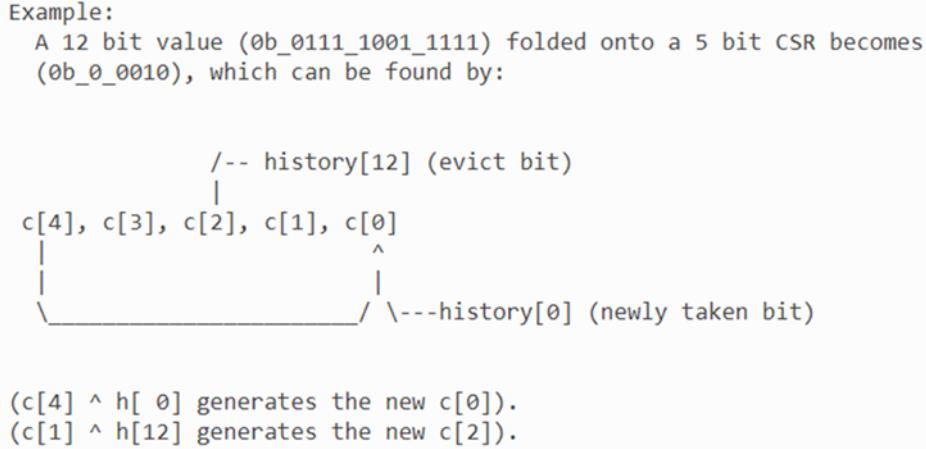


图 1.20: 折叠历史真实实现

```

 $c[0] \leq c[4] \wedge h[0];$ 
 $c[1] \leq c[0];$ 
 $c[2] \leq c[1] \wedge h[12];$ 
 $c[3] \leq c[2];$ 
 $c[4] \leq c[3];$ 

```

分支历史表不只是会在后端提交之后更新，s1~s3 中间每一级都有可能更新，更新时更新的是 pointer 和值。如果在 s1~s3 产生了新的结果就会恢复 pointer，更新新的值。但是如果预测没有错误就不会修改（因为之前已经更新过了）。

注：由 TAGE 的折叠分支历史具体配置表可见，tag 折叠分支历史有 2 个。二者都是使用相同的折叠算法，区别只是在于折叠长度不同，T1-T4 的 tag 折叠分支历史 1 都为 8 比特，而 tag 折叠分支历史 2 都为 7 比特。

1.5.1.3 TAGE: 预测时序

在每次进行预测时，首先在每个带 tag 的预测表中使用 pc 值以及各自的分支历史进行两种不同的哈希函数计算（见下表），计算结果分别用于计算该运算的最终 tag 以及对预测表的索引。

| 计算方式 | |
|-------|--|
| Index | (index 折叠分支历史) \wedge ((pc>>1) 的低位) |
| Tag | (tag 折叠分支历史 1) \wedge (tag 折叠分支历史 2<<1) \wedge ((pc>>1) 的低位) |

若 T1-T4 中索引所得到的 valid 为 1，且 tag 与计算 tag 哈希函数得到的结果匹配，则该预测表中给出的 pred 最高位被加入最终的分支预判序列。最终，通过多级 MUX，可以从所有 tag 匹配的分支预判中选择历史长度最长者作为最终预测结果。

若 T1-T4 无匹配，则采用 T0 为最终的预测结果。T0 的索引方式为直接用 pc 的低 11 位进行索引。

TAGE 需要 2 拍延迟：

- 0 拍生成 SRAM 寻址用 index。index 的生成过程就是把折叠历史和 pc 异或，折叠历史的管理不在 ITTAGE 和 TAGE 内部，而在 BPU 里
- 1 拍读出结果
- 2 拍输出预测结果

1.5.1.4 TAGE: 预测器训练

首先，定义所有产生 tag 匹配的预测表中所需历史长度最长者为 provider，而其余产生 tag 匹配的预测表（若存在的话）被称为 altpred。

TAGE 表项中包含一个 usefulness 域，当 provider 预测正确而 altpred 预测错误时 provider 的 usefulness 置 1，表示该项是一个有用的项，便不会被训练时的分配算法当作空项分配出去。当 provider 产生的预测被证实为一个正确的预测，且此时的 provider 与 altpred 的预测结果不同，则 provider 的 usefulness 域被置若分支指令实际是跳转，则将对应 provider 表项的 ctr 计数器自增 1；若分支指令实际是不跳转，则将对应 provider 表项的 ctr 计数器自减若由于误预测导致 TAGE 表项需要更新，且误预测不是由使用 altpred 而抛弃了正确的 provider 导致的，则说明需要增添表项。但此时并不一定真的能够增添表项。还需要满足 provider 所源自的预测表并非所需历史长度最长的预测表，则此时执行表项增添操作。

这里从逻辑上举一个判断是否满足 provider 所源自的预测表并非所需历史长度最长的预测表的例子：

`s1_providers(i)` 表示预测块中第 i 条分支的 provider 对应的预测表序号，假设 provider 在预测表 T2 中，则 `LowerMask(UIntToOH(s1_providers(i)), TageNTables)` 为 0b0011。`s1_provideds(i)` 表示预测块中第 i 条分支的 provider 是否在 T1~T4，根据刚刚的假设，`Fill(TageNTables, s1_provideds(i).asUInt)` 为 0b1111，二者相与，得到结果为 0b0011，再取反得到 0b1100，于是可以看出 T3、T4 都是比 provider 历史长度更长的预测表。

再举一个例子。在最开始，预测表都是空的，不存在 provider，此时对应的 `s1_providers(i)` 为 0，`s1_provideds(i)` 为 false，则此时 `Fill(TageNTables, s1_provideds(i).asUInt)` 为 0b0000，二者相与取反一定得到 0b1111，说明 T1~T4 都属于所谓的比 provider 历史长度更长的预测表。

具体的表项增添操作如下：

表项增添操作会首先读取所有历史长度长于 provider 的预测表的 usefulness。若此时有某表的 usefulness 域值为 0，则在该表中分配一对对应的表项；若没有找到满足 usefulness 域值为 0 的表，则分配失败。当有多个预测表（如 Tj,Tk 两项）的 usefulness 域均为 0 时，表项的分配概率是随机的，分配的时候随机把某些 table 给 mask 掉，让它不会每次都分配同一个。这个 mask 的具体实现是：待选的历史表中（长度大于 provider 且 u 为 0 的所有历史表），使用产生的随机数随机将一些表屏蔽掉，如果 maskedEntry 的第一个不可用，那么选没 mask 的第一个。这里的表项分配的随机性是通过 chisel 的 util 包里的 64 位线性反馈移位寄存器原语 LFSR64 生成伪随机数来实现的，在 verilog 代码中对应 allocLFSR_lfsr 寄存器。在训练时，用 7bit 饱和计数器 bankTickCtrs 统计分配失败次数-成功次数，当分配失败的次数足够多，bankTickCtrs 计数器计数到满达到饱和时，触发全局 useful bit reset，把所有的 usefulness 域清零。

最后，在初始化时/TAGE 表分配新项时，所有的表项中的 ctr 计数器均被设置为 0，所有的 usefulness 域均被设置为 0。

注：同一个预测块中的两条分支指令对应的 usefulness 不一定是相等的，如果 altpred 的第一条分支预测跳转，第二条分支预测不跳转，provider 的都预测跳转，就只有第一个 u 要置一。

注：meta 是预测器预测的时候的数据，update 的时候拿回来更新用。都叫 meta 是因为 composer 将所有预测器整合起来，用共同的接口 meta 和外界交互。TAGE 的 meta 具体构成见下图：

1.5.1.5 TAGE: 备选预测逻辑 (USE ALT ON NA)

当 tag 匹配的项的置信度计数器 ctr 为 0 时，altpred 有时比正常预测更准确。因此在实现中使用一个 4-bit 计数器 useAltCtr，动态决定是否在最长历史匹配结果信心不足时使用备选预测。在实现中出于时序考虑，始终用基预测表的结果作为备选预测，这带来的准确率损失很小。

备选预测逻辑的具体实现如下：

`ProviderUnconf` 表示最长历史匹配结果的信心不足。当 provider 对应的 ctr 值为 0b100、0b011 时，说

```

class TageMeta(implicit p: Parameters)
  extends TageBundle with HasSCPParameter
{
  val providers = Vec(numBr, ValidUndirected(UInt(log2Ceil(TageNTables).W)))
  val providerResps = Vec(numBr, new TageResp)
  // val altProviders = Vec(numBr, ValidUndirected(UInt(log2Ceil(TageNTables).W)))
  // val altProviderResps = Vec(numBr, new TageResp)
  val altUsed = Vec(numBr, Bool())
  val altDiffer = Vec(numBr, Bool())
  val basecnts = Vec(numBr, UInt(2.W))
  val allocates = Vec(numBr, UInt(TageNTables.W))
  val takens = Vec(numBr, Bool())
  val scMeta = if (EnableSC) Some(new SCMeta(SCNTables)) else None
  val pred_cycle = if (!env.FPGAPlatform) Some(UInt(64.W)) else None
  val use_alt_on_na = if (!env.FPGAPlatform) Some(Vec(numBr, Bool())) else None

  def altPreds = basecnts.map(_(1))
  def allocateValid = allocates.map(_.orR)
}
zoujr, 16个月前 via PR #986 • BPU: Modify Tage to match new frontend interface

```

图 1.21: TAGE meta 构成

明最长历史匹配结果的信心很足, 此时 providerUnconf 为 false; 当 provider 对应的 ctr 值为 0b01、0b10 时, 说明最长历史匹配结果的信心不足, 此时 providerUnconf 为 true。

useAltOnNaCtrs 是 128 个 4-bit 饱和计数器构成的计数器组, 每个计数器都被初始化为 0b1000。在 TAGE 收到训练更新请求时, 如果拿来训练的预测中, 发现 provider 的预测结果与 altpred 不同, 且 provider 的预测结果信心不足, 则讨论备选预测结果是否正确。如果备选预测结果正确而 provider 错误, 则对应 useAltOnNaCtrs 计数器的值 +1; 若备选预测结果错误而 provider 正确, 则对应 useAltOnNaCtrs 计数器的值 -1。因为 useAltOnNaCtrs 是饱和计数器, 所以当 useAltOnNaCtrs 值已经为 0b1111 且正确或已经为 0b0000 且错误时, useAltOnNaCtrs 的值不变。

useAltOnNa 是利用 pc(7, 1), 即 pc 的对应低位索引 useAltOnNaCtrs 计数器组后得到的计数结果的最高位。

当 providerUnconf && useAltOnNa 为真时, 使用备选预测结果 (即基预测表的结果) 为 TAGE 最终的预测结果, 而不是 provider 的预测结果。

1.5.1.6 TAGE: wrbypass

Wrbypass 里面有 Mem, 也有 Cam, 用于给更新做定序, 每次 TAGE 更新时都会写进这个 wrbypass, 同时写进对应预测表的 sram。每次更新的时候会查这个 wrbypass, 如果 hit 了, 那就把读出的 TAGE 的 ctr 值作为旧值, 之前随 branch 指令带到后端再送回前端的 ctr 旧值就不要了。这样如果一个分支重复更新, 那 wrbypass 可以保证某一次更新一定能拿到相邻的上一次更新的最终值。

T0 有一个对应的 wrbypass, T1~T4 各有 16 个。每个预测表对应的 wrbypass 中, Mem 都有 8 个 entry, T0 每个 entry 存储 2 个 (对应两个 bank) 预测表的表项, T1~T4 每个 entry 存储 1 个 (因为 2 个 bank 存在不同的 wrbypass 里) 预测表的表项; Cam 有 8 个 entry, 输入更新的 idx 和 tag (T0 没有 tag) 就可以读到对应数据在 Cam 中的位置。Cam 和 Mem 是同时写的, 所以数据在 Cam 中的位置就是在 Mem 中的位置。于是利用这个 Cam, 我们就可以在更新的时候查看对应 idx 的数据是否在 wrbypass 中。

1.5.1.7 存储结构

- 4 张历史表, 每张表根据 pc 低位分了共 8 个 bank, 每个 bank 有 256 个 set, 每个 set 对应一个 FTB 项的最多 2 条分支。每个 bank 共 512 项, 每张表共 4096 项, 所有表共 16K 项

- 每个表项含有 1bit valid, 8bit tag, 3bit ctr, 1bit useful; 其中 useful bit 独立存储
- base table 有 2048 set, 每 set 两条分支各一个 2bit 饱和计数器, 共记录 4K 条分支
- 历史表每个 bank 有 8 项的写缓存 wrbypass, base table 共有 8 项的 wrbypass
- use_alt_on_na 预测共 128 个 4 bit 饱和计数器

1.5.1.8 索引方式

- index = pc[11:1] ^ folded_hist(11bit)
- tag = pc[11:1] ^ folded_hist(8bit) ^ (folded_hist(7bit) << 1)
- 历史采取基本的分段异或折叠
- 每项两条分支和 FTB 两个 slot 之间的两种对应关系, 用 pc[1] 选取, 由于 FTB 项的建立机制, 第一个 slot 的使用率会大于第二个, 此举可以缓解这种分布不均的情况
- base table 和 use_alt_on_na 都直接使用 pc 低位索引

1.5.1.9 预测流程

- s0 进行索引计算, 地址送入 SRAM, 仅有对应的 bank 被使能
- s1 进行 tag 匹配和 bank 数据选取, 以及两个 slot 间的重排序, 得到每张历史表的总结果, 传到顶层进行最长历史选取, 结合 use_alt_on_na 机制, 在最长历史匹配和 base table 之间进行选取 (将原算法简化为始终用 base table 作为备选预测), 得到每条分支的预测结果后暂存至 s2
- s2 使用预测结果, 在 BPU 内和 s1 结果进行比对, 判断是否需要冲刷流水线

1.5.1.10 训练流程

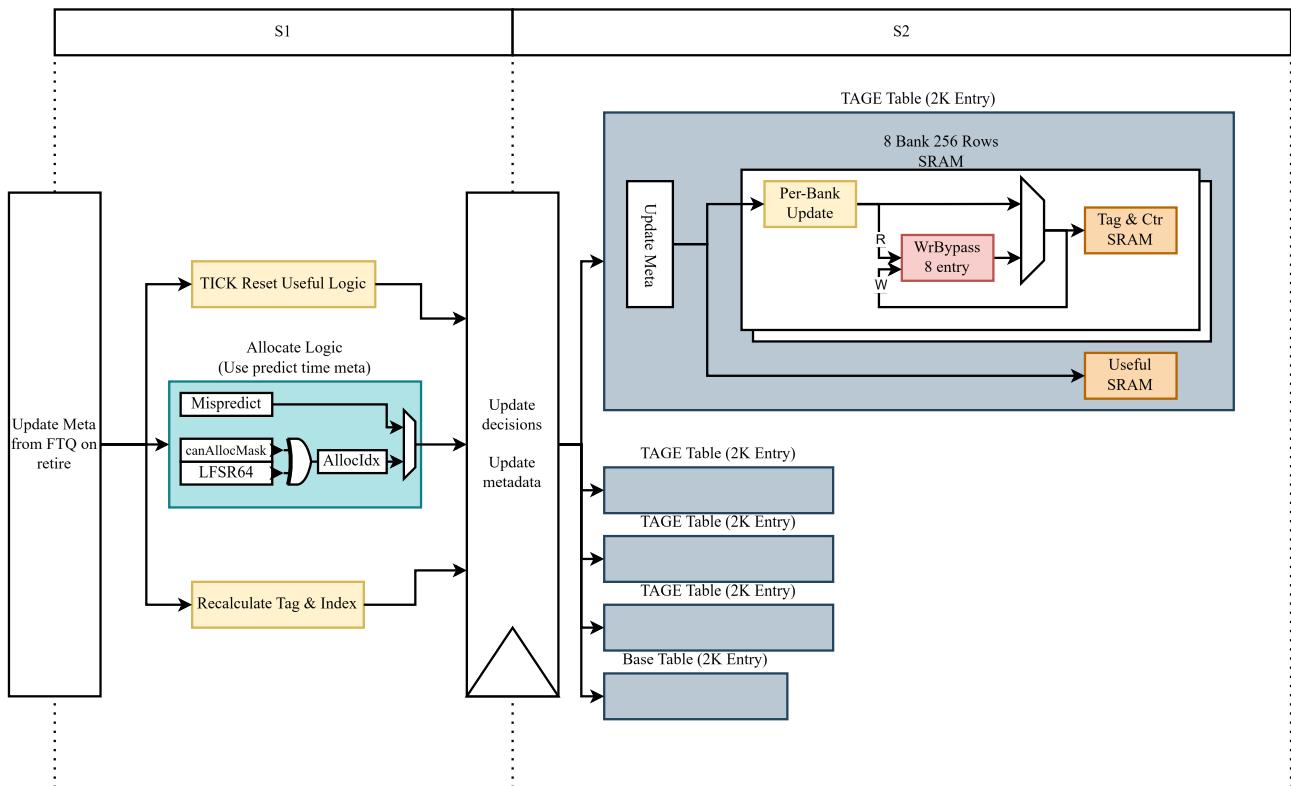


图 1.22: 训练流程

收到来自 FTQ 的训练请求后，根据预测时记录的信息进行更新，更新流程分为两个流水级，其中第一个流水级在历史表外部判断一些更新条件，细节如下：

- provider 更新：预测时命中的最长历史表会进行更新
 - 当备选预测和它不同时，按 provider 是否正确，决定新的 useful bit
 - 根据实际方向决定 ctr 的加减
- alloc：当误预测，且排除以下情况（provider 正确，但预测时选择了错误的备选预测结果）时进行分配
 - 分配时根据预测时记录的所有历史表的 useful bit 信息生成 mask，用随机数 LFSR 随机 mask 掉一些 bit，判断是否仍有比 provider 更长的历史表空余项？
 - * 如有，使用其中最短历史的项
 - * 如无，用 LFSR mask 前，历史长于 provider 的最短历史的可分配项
- useful reset 计数器：根据分配成功/失败的净次数增减一个 8bit 饱和计数器，如失败次数过多以至饱和，清除所有的 useful bit
 - 记比 provider 长的历史表个数为 n，其中 useful bit 为 0 的视为分配成功，反之视为失败，两者次数之差作为此次增减的绝对值
- use_alt_on_na 训练：当 provider 的 ctr 为两个最弱的值，且备选预测和 provider 方向不同时，根据备选预测的正确与否，增减 pc 低位索引的的 use_alt_on_na 饱和计数器

第二个流水级将更新请求发入各个预测表内部，尝试写入 SRAM

- 其中 ctr 的旧值可能来自很久以前的预测内容，也许和现在表内 ctr 的情况已经大相径庭，为解决此问题，引入 wrbypass 机制：
 - wrbypass 是 TAGE table 的全相联写缓存，每当尝试写入时，首先根据历史表的索引查询此全相联表，若命中，将对应的内容当作旧 ctr，根据对应的跳转方向更新后，同时写入 SRAM 和 wrbypass，如果不命中，根据替换算法选取一项替换
 - 每个 bank 有一个对应的 8 项全相联 wrbypass，每个 table 共 64 项
- 由于使用了单口 SRAM，不能同时处理读写请求，故为了避免读写冲突：
 - 使用分 bank 的做法
 - 当饱和且新值等于旧值时，不写入
- 当仍然发生了读写冲突时，处理写请求，但不阻塞读请求，使用读结果时默认当作 not hit，此举可能降低准确率

1.5.1.11 SC：设计思路

一些应用上，一些分支行为与分支历史或路径相关性较弱，表现出一个统计上的预测偏向性。对于这些分支，相比 TAGE，使用计数器捕捉统计偏向的方法更为有效。TAGE 在预测非常相关的分支时非常有效，TAGE 未能预测有统计偏向的分支，例如只对一个方向有小偏差，但与历史路径没有强相关性的分支。

SC 统计校正的目的是检测不太可靠的预测并将其恢复。SC 负责预测具有统计偏向的条件分支指令并在该情形下反转 TAGE 预测器的结果。

1.5.1.12 SC: 硬件实现

SC 共维护了 4 个预测表，预测表的具体参数见下表。

| 序号 | 项数 | ctr 长度 | 折叠后历史长度 | 设计原理 |
|----|-----|--------|---------|------------------------|
| T1 | 512 | 6 | 0 | ghv 取 ptr 起低 0 比特折叠异或 |
| T2 | 512 | 6 | 4 | ghv 取 ptr 起低 4 比特折叠异或 |
| T3 | 512 | 6 | 8 | ghv 取 ptr 起低 10 比特折叠异或 |
| T4 | 512 | 6 | 8 | ghv 取 ptr 起低 16 比特折叠异或 |

1.5.1.13 SC: 预测时序

SC 收到来自 TAGE 的预测结果 pred 和 ctr、来自 BPU 的折叠分支历史信息和 pc，根据 ((index 折叠分支历史) \wedge ((pc>>1) 的低位)) 索引 SC 预测表，根据 SC 预测表的结果 ctr 和 TAGE 的预测结果 pred 和 ctr 动态判定（见 HasSC）是否反转 TAGE 的预测。

SC 的动态判定具体逻辑如下：

SC 中实现了 2 个 bank 的 scThresholds 寄存器组，2 个 bank 是为了和 TAGE 的 2 个 bank 对应，都是因为一个预测块内最多会有两条分支指令。SC 中动态判定的寄存器组一共就是这两个。每个 bank 的 scThresholds 中有一个 5 比特的饱和计数器 ctr (初始值为 0b10000) 和一个 8 比特的 thres (初始值为 6)。为了以示区别，我们后面用 tage_ctr、sc_ctr、thres_ctr 来分别表示 TAGE 传给 SC 的 ctr、SC 自己的 ctr、scThresholds 中的 ctr。

scThresholds 的更新：当 SC 收到训练数据时，如果当时 SC 翻转了 TAGE 的预测结果，且 $(sc_ctr_02+1)+(sc_ctr_12+1)+(sc_ctr_22+1)+(sc_ctr_32+1)$ (有符号进位加) 后取绝对值后在 [thres-4, thres-2] 的范围内，则 scThresholds 会被更新。

- scThresholds 中 ctr 的更新：若预测正确，则 thres_ctr+1；若预测错误，则 thres_ctr-1；若 thres_ctr 已为 0b11111 且预测正确，或 thres_ctr 已为 0b00000 且预测错误，则 thres_ctr 保持不变。
- scThresholds 中 thres 的更新：若 thres_ctr 更新后的值已达 0b11111 且 thres<= 31，则 thres+2；若 thres_ctr 更新后的值为 0 且 thres>=6，则 thres-2。其余情况 thres 不变。
- Thres 更新判断结束后，会对 thres_ctr 再做一次判断，若更新后的 thres_ctr 若为 0b11111 或 0，则 thres_ctr 会被置回初始值 0b10000。

设定 scTableSums 为 $(sc_ctr_02+1)+(sc_ctr_12+1)+(sc_ctr_22+1)+(sc_ctr_32+1)$ (有符号进位加)，tagePrvdCtrCentered 为 $(2(tage_ctr-4)+1)8$ (有符号进位加)，totalSum 为 scSum+tagePvdr (有符号进位加)。若 $scTableSums > (有符号的 thres - tagePrvdCtrCentered)$ 并且 totalSum 为正，或者 $scTableSums < (-有符号的 thres - tagePrvdCtrCentered)$ 并且 totalSum 为负，都说明已经超过了阈值，翻转 TAGE 的预测结果，否则仍使用 TAGE 的预测结果不变。

SC 的预测算法依赖 TAGE 里面的是否有历史表 hit 的信号 provided，以及 provider 的预测结果 taken，从而来决定 SC 自己的预测。provided 是使用 SC 预测的必要条件之一，provider 的 taken 作为 choose bit，选出 SC 最终的预测，这是因为 SC 在 TAGE 预测结果不同的场景下可能有不同的预测。

sc 反转 TAGE 的预测结果会造成 TAGE 表增添新项。

SC 需要 3 拍延迟：

- 0 拍生成寻址 index 得到索引 s0_idx
- 1 拍读出 SCTable 对应 s0_idx 的计数器数据 s1_scResps
- 2 拍根据 s1_scResps 选择是否需要反转预测结果

- 3 拍输出完整的预测结果

1.5.1.14 SC: Wr bypass

Wr bypass 里面有 Mem，也有 Cam，用于给更新做定序，每次 SC 更新时都会写进这个 wr bypass，同时写进对应预测表的 sram。每次更新的时候会查这个 wr bypass，如果 hit 了，那就把读出的 SC 的 ctr 值作为旧值，之前随 branch 指令带到后端再送回前端的 ctr 旧值就不要了。这样如果一个分支重复更新，那 wr bypass 可以保证某一次更新一定能拿到相邻的上一次更新的最终值。

SC 的 T1~T4 各有 2 个 wr bypass，每个预测表的 wr bypass 中，Mem 都有 16 个 entry，每个 entry 存储 2 个预测表的表项；Cam 有 16 个 entry，输入更新的 idx 就可以读到对应数据在 Cam 中的位置。Cam 和 Mem 是同时写的，所以数据在 Cam 中的位置就是在 Mem 中的位置。于是利用这个 Cam，我们就可以在更新的时候查看对应 idx 的数据是否在 wr bypass 中。

1.5.1.15 SC: 预测器训练

sc_ctr (见 signedSatUpdate) 是一个 6 比特的有符号饱和计数器，在指令实际 taken 后 +1，nottaken-1，计数范围是 [-32, 31]。

注：meta 是预测器预测的时候的数据，update 的时候拿回来更新用。都叫 meta 是因为 composer 将所有预测器整合起来，用共同的接口 meta 和外界交互。SC 的 meta 具体构成见下图：

```
class SCMeta(val ntables: Int)(implicit p: Parameters) extends XSBundle with HasSCParameter {
  val tageTakens = Vec(numBr, Bool())
  val scUsed = Vec(numBr, Bool())
  val scPreds = Vec(numBr, Bool())
  // Suppose ctrbits of all tables are identical
  val ctrs = Vec(numBr, Vec(ntables, SInt(SCCtrBits.W)))
}
```

图 1.23: SC meta 具体构成

1.5.2 整体框图

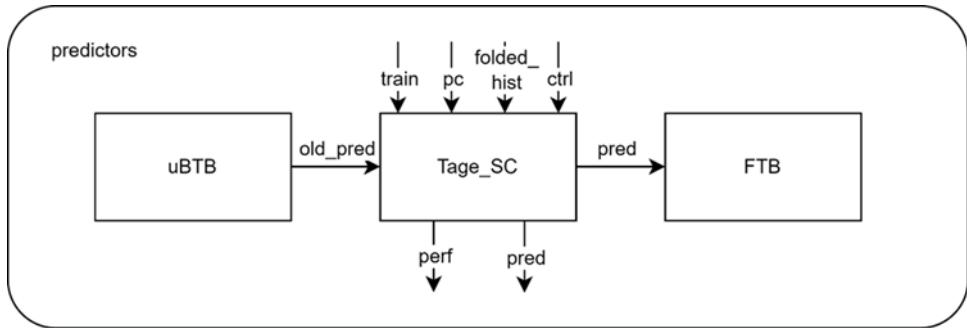


图 1.24: 整体框图

1.5.3 接口时序

s0 输入 pc 和折叠历史时序示例

上图示意了 s0 输入 pc 和折叠历史时序的示例，当 io_s0_fire 为高时，输入的 io_in_bits 数据有效。

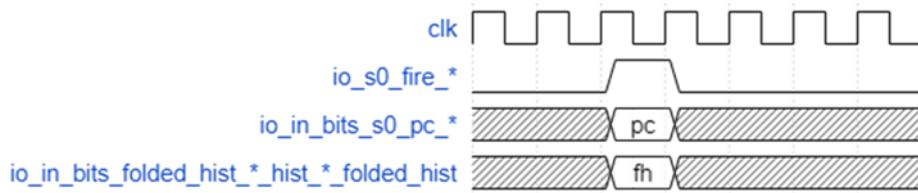


图 1.25: pc 和折叠历史时序

1.5.3.1 存储结构

- 4 张表，历史长度分别为 0、4、10、16，每张表有 256 项，每项中是 4 个 6 bit 宽的饱和计数器，其中 4 个饱和计数器分别对应 2 条分支 *TAGE 的 2 种预测结果 (T/NT)。每张表共 1024 个计数器，存储开销 3KB
- 每张表有 16 项的写缓存 wrbypass
- 2 个 threshold 寄存器，分别对应一个 slot 的分支。每个 threshold 有 8bit 宽的阈值饱和计数器，以及 5bit 的增减缓冲饱和计数器

1.5.3.2 索引方式

- index = pc[8:1] ^ folded_hist(8bit)
- 历史采取基本的分段异或折叠
- 每项两条分支和 FTB 两个 slot 之间的两种对应关系，用 pc[1] 选取，由于 FTB 项的建立机制，第一个 slot 的使用率会大于第二个，此举可以缓解这种分布不均的情况

1.5.3.3 预测流程

- s0 进行索引计算，地址送入 SRAM
- s1 读出饱和计数器，进行 slot 间的重排序，对应 TAGE 两种预测结果的 SC ctr 传到顶层分别做加法，结果寄存到 s2
- s2 将 SC 的 ctr 和与 TAGE 的 provider ctr 做加法，得到最终结果，取绝对值，若大于阈值，且 TAGE 也有 hit，用加法结果的符号作为最终预测的方向，寄存到 s3
- s3 使用方向结果，在 BPU 内和 s2 结果进行比对，判断是否需要冲刷流水线

1.5.3.4 训练流程

收到来自 FTQ 的训练请求后，根据预测时记录的信息进行更新，更新流程分为两个流水级，其中第一个流水级在历史表外部判断一些更新条件，细节如下：

- 若预测时 TAGE 命中，根据预测时存下的旧 SC ctr 与旧 TAGE ctr 再次做加法，和更新所用的阈值（预测阈值 *8+21）比对，若小于阈值，或者此时预测结果和执行方向不符，则根据 perceptron 训练规则训练每个计数器，请求寄存到下一拍发送给每张表

第二个流水级将更新请求发入各个预测表内部，尝试写入 SRAM，仍然尝试查询写缓存 wrbypass 获取最新计数器值（应该可以在上一个流水级先查询，而不是直接使用旧 ctr）

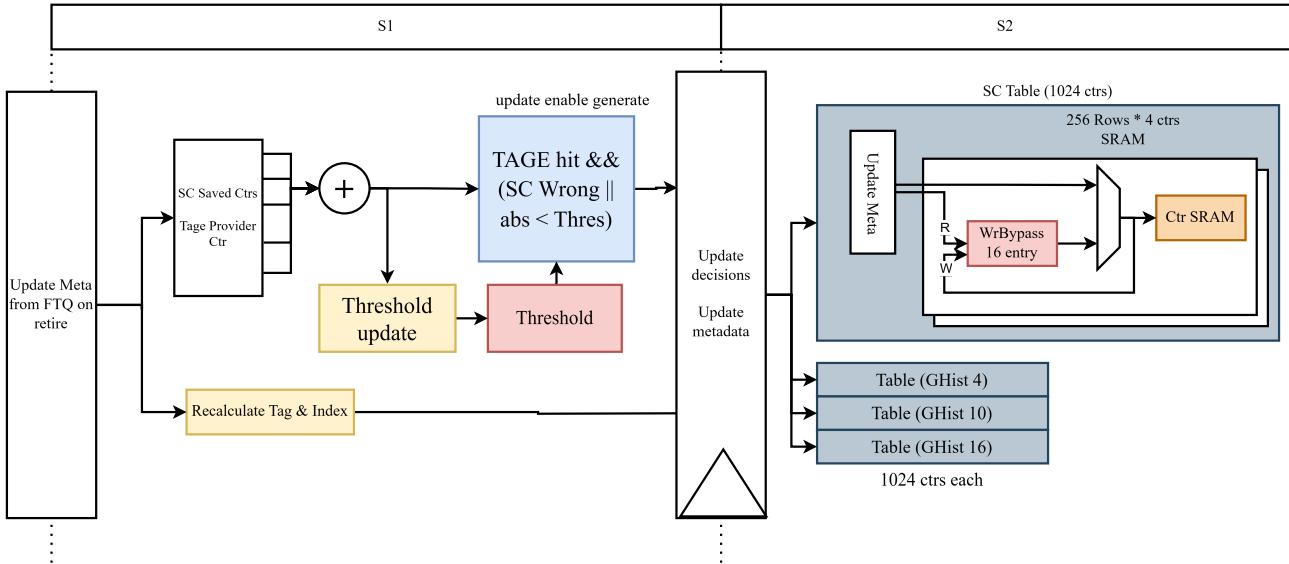


图 1.26: 训练流程

1.6 BPU 子模块 ITTAGE

1.6.1 功能

ITTAGE 接收来自 BPU 内部的预测请求，其内部由一个基预测表和多个历史表组成，每个表项中都有一个用于存储间接跳转指令目标地址的字段。基预测表用 PC 索引，而历史表用 PC 和一定长度的分支历史折叠后的结果异或索引，不同历史表使用的分支历史长度不同。在预测时，还会用 PC 和每个历史表对应的分支历史的另一种折叠结果异或计算 tag，与表中读出的 tag 进行匹配，如果匹配成功则该表命中。最终的结果取决于命中的历史长度最长的预测表的结果。最终，ITTAGE 将预测结果输出至 composer。

1.6.1.1 间接跳转指令的预测

ITTAGE 用于预测间接跳转指令。普通分支指令和无条件跳转指令的跳转目标直接编码于指令中，便于预测，而间接跳转指令的跳转地址来自运行时可变的寄存器，从而有多种可能选择，需要根据分支历史对其作出预测。

为此，ITTAGE 的每个表项在 TAGE 表项的基础上加入了所预测的跳转地址项，最后输出结果为选出的命中预测跳转地址而非选出的跳转方向。

由于每个 FTB 项仅存储至多一条间接跳转指令信息，ITTAGE 预测器每周期也最多预测一条间接跳转指令的目标地址。

香山南湖架构中的 ITTAGE 提供了 5 个带 tag 的预测表 T1-T5，基准预测器和带 tag 的预测表的基本信息见下表。

| 预测器 | 是否带 tag | 作用 | 表项构成 | 项数 |
|----------|---------|-------------------------------|---|------|
| 基准预测器 T0 | 否 | 用于在带 tag 预测表的 tag 均不匹配时提供预测结果 | ITTAGE 中并没有实现 T0，而是直接使用 ftb 的预测结果作为基准预测结果 | ITTB |

| 预测器 | 是否带 tag | 作用 | 表项构成 | 项数 |
|-----------|---------|---------------------------|---|-----------------------------|
| 预测表 T1-T5 | 是 | 存在 tag 匹配时，选历史长度最长者提供预测结果 | valid 1bit tag 9bits ctr 2bits (最高位表示要不要输出这个预测结果) us: 1bit (usefulness 计数器) target: 39 bits | T1-T2 各 256 项 T3-T5 各 512 项 |

在 BPU 模块中维护了一个 256 比特的全局分支历史 ghv，并为 ITTAGE 的 5 个带 tag 的预测表分别维护了各自的折叠分支历史，折叠算法同 TAGE。折叠历史具体配置见下表，其中 ghv 是一个循环队列，“低” n 位指的是从 ptr 指示的位置算起的低位：

| 历史 | index 折叠分支历史长度 | tag 折叠分支历史 1 长度 | tag 折叠分支历史 2 长度 | 设计原理 |
|-------------|----------------|-----------------|-----------------|------------------------|
| 全局分支历史 ghv | 256 比特 | 无 | 无 | 每比特代表对应分支跳转与否 |
| T1 对应折叠分支历史 | 4 比特 | 4 比特 | 4 比特 | ghv 取 ptr 起低 4 比特折叠异或 |
| T2 对应折叠分支历史 | 8 比特 | 8 比特 | 8 比特 | ghv 取 ptr 起低 8 比特折叠异或 |
| T3 对应折叠分支历史 | 9 比特 | 9 比特 | 8 比特 | ghv 取 ptr 起低 13 比特折叠异或 |
| T4 对应折叠分支历史 | 9 比特 | 9 比特 | 8 比特 | ghv 取 ptr 起低 16 比特折叠异或 |
| T5 对应折叠分支历史 | 9 比特 | 9 比特 | 8 比特 | ghv 取 ptr 起低 32 比特折叠异或 |

ITTAGE 需要 3 拍延迟：

- 0 拍生成寻址 index
- 1 拍读出数据
- 2 拍选出命中结果
- 3 拍输出

1.6.1.2 Wr bypass

Wr bypass 里面有 Mem，也有 Cam，用于给更新做定序，每次 ITTAGE 更新时都会写进这个 wr bypass，同时写进对应预测表的 sram。每次更新的时候会查这个 wr bypass，如果 hit 了，那就把读出的 ITTAGE 的 ctr 值作为旧值，之前随 branch 指令带到后端再送回前端的 ctr 旧值就不要了。这样如果一个分支重复更新，那 wr bypass 可以保证某一次更新一定能拿到相邻的上一次更新的最终值。

ITTAGE 的每一个预测表 T1~T5 都有着一个对应的 wr bypass，每个预测表的 wr bypass 中，Mem 都有 4 个 entry，每个 entry 存储 1 个 ctr；Cam 有 4 个 entry，输入更新的 idx 和 tag 就可以读到对应数据在 Cam

中的位置。Cam 和 Mem 是同时写的，所以数据在 Cam 中的位置就是在 Mem 中的位置。于是利用这个 Cam，我们就可以在更新的时候查看对应 idx 的数据是否在 wrbypass 中。

1.6.1.2.1 预测器训练

首先，定义所有产生 tag 匹配的预测表中所需历史长度最长者为 provider，而其余匹配的预测表（若存在的话）被称为 altpred。当 provider 的 ctr 为 0 时，会选择 altpred 的结果作为预测结果。

ITTAGE 表项中包含一个 usefulness 域，当 provider 预测正确而 altpred 预测错误时 provider 的 usefulness 置 1，表示该项是一个有用的项，便不会被训练时的分配算法当作空项分配出去。当 provider 产生的预测被证实为一个正确的预测，且此时的 provider 与 altpred 的预测结果不同，则 provider 的 usefulness 域被置若预测地址与实际一致，则将对应 provider 表项的 ctr 计数器自增 1；若预测地址与实际不一致，则将对应 provider 表项的 ctr 计数器自减 1。ITTAGE 中，会根据 ctr 判断是否采取这个预测的跳转目标结果。当 ctr 为 0 时，会选择 altpred 的结果。

接下来，若该 provider 所源自的预测表并非所需历史长度最高的预测表，则此时执行如下的表项增添操作。表项增添操作会首先读取所有历史长度长于 provider 的预测表的 usefulness 域。若此时有某表的 usefulness 域值为 0，则在该表中分配一对应的表项；若没有找到满足 usefulness 域值为 0 的表，则分配失败。当有多个预测表（如 Tj,Tk 两项）的 usefulness 域均为 0 时，表项的分配概率是随机的，分配的时候随机把某些 table 给 mask 掉，让它不会每次都分配同一个。这里的表项分配的随机性是通过 chisel 的 util 包里的 64 位线性反馈移位寄存器原语 LFSR64 生成伪随机数来实现的，在 verilog 代码中对应 allocLFSR_lfsr 寄存器。在训练时，用 8 位饱和计数器 tickCtr 统计分配失败次数-成功次数，当分配失败的次数足够多，tickCtr 计数器计数到满达到饱和时，触发全局 useful bit reset，把所有的 usefulness 域清零。

注：ITTAGE 的清零 usefulness 域的饱和计数器名字是 tickCtr，长度为 8 比特，名字和长度均与 TAGE 不同。

最后，在初始化时/TAGE 表分配新项时，所有的表项中的 ctr 计数器均被设置为 0，所有的 usefulness 域均被设置为 0。

1.6.2 存储结构

- 5 张历史表，项数分别为 256、256、512、512、512，每张表根据 pc 低位分了共 2 个 bank，每个 bank 有 128 个 set，每个 set 对应一个 FTB 项的最多 1 条间接跳转。
- 每个表项含有 1bit valid, 9bit tag, 2bit ctr, 39bit target, 1bit useful；其中 useful bit 独立存储，valid 使用寄存器堆独立存储
- 以 FTB 结果作为 base table，等效于 2K 项（但 FTB target 位宽不够，无法有效存储远跳转地址）
- 历史表每个 bank 有 4 项的写缓存 wrbypass

1.6.3 索引方式

- index = pc[8:1] ^ folded_hist(8bit) 或 pc 和 folded_hist 各 9bit
- tag = pc[17:9] (或 pc[19:10]) ^ folded_hist(9bit) ^ (folded_hist(8bit) << 1)
 - 此处大概还是应该用 pc 低位比较好，而不是用 index 没用过的另一段 pc，或者
- 历史采取基本的分段异或折叠

1.6.4 预测流程

- s0 进行索引计算，地址送入 SRAM

- s1 读出表项，进行 bank 选取，以及判断是否命中，结果寄存到 s2
- s2 计算最长历史匹配和次长历史匹配：
 - 当存在历史表命中，且 $ctr!=0$ 的时候，尝试使用最长历史结果目标地址
 - 当 provider 信心不足 ($ctr==0$) 时，若次长历史命中，尝试使用次长历史结果目标地址
 - 当没有历史表命中的时候，使用 FTB 结果
 - 尝试使用历史表的结果时，只有 $ctr>1$ 才会真正使用，若 $ctr<=1$ ，则不使用结果
- s3 使用目标地址，在 BPU 内和 s2 结果进行比对，判断是否需要冲刷流水线

1.6.5 训练流程

基本和 TAGE 相同，对于 target 字段，仅当分配新表项或者原 ctr 为最小值 0 时，才会替换新值，否则保持原样

1.7 BPU 子模块 RAS

1.7.1 术语说明

| 缩写 | 全称 | 描述 |
|------|---|-----------|
| TOSW | Top Of Speculative Queue Write Pointer | 推测队列写指针 |
| TOSR | Top Of Speculative Queue Read Pointer | 推测队列读指针 |
| NOS | Next Older Speculative Queue entry Pointer | 推测队列更旧项指针 |
| ssp | Speculative Stack Pointer | 虚拟推测栈指针 |
| nsp | Next Stack Pointer | 提交栈指针 |

1.7.2 功能

RAS 预测器使用栈结构来预测函数调用与返回这类具有成对匹配特性的执行流。其中调用 (push/call) 类指令的特征为类指令的特征为目标寄存器地址为 1 或 5 的 jal/jalr 指令。返回 (ret) 类指令的特征为源寄存器为 1 或 5 的 jalr 指令。

在实现中，RAS 预测器在 s2 和 s3 两阶段提供预测结果。

昆明湖架构的 RAS 预测器与南湖架构差异较大。通过引入持久化队列，新的 RAS 预测器结构解决了局部预测器因推测执行所导致的预测器数据污染问题。同时，新结构也保留了和南湖架构类似的栈结构作为提交栈，存储提交后的压栈信息，以弥补持久化队列存储密度不高的缺点。

1.7.2.1 基于持久化队列设计的 RAS 预测器架构整体概述

RAS 预测器利用分支预测 FTB 块中的 call/return 指令局部的配对信息对 return 指令作出预测。由于正常调用的函数执行完毕会返回到调用指令的下一条指令，RAS 预测器可在遇到 call 指令时根据当前指令 PC 及当前指令是否为 RVC 指令（确定指令宽度为 2 还是 4）生成其函数调用返回地址的预测结果并压栈。

由于现代超标量乱序处理器通常采用深流水线推测执行技术，分支预测器会在之前所预测指令执行结果确定前为后续指令生成预测结果，也即，若面临 ABC 三个连续的分支指令预测请求，RAS 预测器并不能在预测 B

时获取到 A 块内指令的最终执行结果，而只能获取到 A 块的 Z 块内指令的最终执行结果和基于分支预测结果的 Z 到 B 块间推测结果。若 Z 到 B 块间存在错误的分支预测结果，则需要对其涉及的分支预测器状态进行误预测恢复。如前文所述，RAS 预测器基于栈结构对 call-return 指令对作出预测，配对信息准确性对其准确率至关重要。要精确地恢复 RAS 预测器在发生错误预测点的状态，最直观的做法是从当前最新预测点回溯到发生误预测点，撤销其间对 RAS 栈作出的所有改动。而为提升误预测恢复及时性，现代处理器通常难以允许误预测恢复时进行如此高复杂度的操作。在南湖架构设计中，RAS 预测器在发生误预测时仅恢复发生误预测预测块对应栈顶项和 RAS 栈顶指针。这一恢复策略可以应对因推测执行导致的误预测点 RAS 栈顶及其更上方数据的污染，但无法应对推测执行中因先 pop 再 push 对 RAS 栈顶下方数据的污染。

这类污染会导致后续的 return 指令跳转目标出错，从而产生误预测。为解决这一问题，昆明湖架构的新 RAS 预测器通过引入持久化队列，保存 RAS 推测执行过程中的所有局部状态，实现了抗上述污染的预测。具体地，持久化队列模拟的栈结构有读指针 TOSR、写指针 TOSW 和栈底指针 BOS 三个指针，每一项除记录自身数据外，还记录其前一项在持久化队列中的位置指针 NOS；在每次栈 push 操作时均前移一次 TOSW 为当前压栈数据新分配一项存储空间，并将当前 TOSR 指向 TOSW 原位置（即新分配空间），新压栈的项记录的 NOS 存储 push 操作前的读指针位置；在每次栈 pop 操作时，将 TOSR 移动到当前 TOSR 指向的项内 NOS 指针位置。通过上述方式，RAS 能够通过前向链表遍历当前版本（对应一个推测执行路径）栈的所有项，这一设计也不会覆盖任何非本版本（对应其他推测执行路径）的 RAS 栈数据。

为提升 RAS 有效存储容量，并非所有 RAS 项均采用持久化队列形式存储。根据实践数据，为满足推测执行路径需要，持久化队列最大约需要 28 项，因而 RTL 实现中使用了 32 项持久化队列。在 RAS 项对应预测块（即含 call 指令的预测块）指令提交后，我们可以将该块从持久化队列释放移入提交栈中，释放操作通过改变 BOS 指针到提交预测块对应的 TOSW 指针。提交栈采用与南湖架构相同的设计，压栈时增加提交栈指针 nsp 并将压入数据写入新栈顶；出栈时减少栈指针 nsp。由于其仅存放提交后的确定性信息，不存在推测执行污染问题。原 BOS 到新 TOSW 这一区间在提交栈中可能存在其他错误路径上的压栈结果，而这些结果可以在这一 BOS 移动过程中被自然释放。

由于引入了持久化队列和提交栈两个结构，栈顶项可能在二者之一，在提供预测结果时会需要动态判断。持久化队列是一个环形队列，其指针除用于寻址的 value 外还有一 flag 位，这一位可协助判定 BOS 和 TOSW、TOSR 的位置关系。当 TOSR 位于 BOS 之上 TOSW 之下时，栈顶项位于持久化队列内部；当 TOSR 位于 BOS 之下时，栈顶项位于持久化队列外部，即位于提交栈内。因而，我们能够在运行时动态选出栈顶项。注意到，我们从提交栈获取的栈顶项并非始终与已提交指令的栈顶一致，因而，我们需要为 RAS 预测器维护另一个提交栈指针 ssp，其含义为，持久化队列中该项在被压入提交栈后所处的位置。在从提交栈访问栈顶项时利用 ssp 而非 nsp 完成数据读取。

上述讨论全部基于持久化队列容量充足而不发生溢出的情况。若当前程序段内 push 操作过于密集而后端执行速度不够快，可能会出现持久化队列容量不足发生溢出的情况。这一情况有两种可能的处理方案：强行覆盖或者动态关闭返回堆栈的预测。目前昆明湖架构选择后一种实现策略。在当前 BOS 和 TOSW 即将重叠时，将 BOS 强制前进一项来避免持久化队列被意外清空。由于 BOS 记录项并不一定需要在此期间使用，此策略能够略微减少前端的阻塞。不足在于密集的 push 操作可能在错误路径上，被覆写的项如果需要使用也会导致少量出栈错误，且出栈错误原因复杂。动态关闭方案不足在于密集的 push 操作可能在正确路径上，因为存在未被写入返回堆栈的项，将可能导致栈内数据错误。如果是递归这种错误的影响可能减轻。出于可控性考虑，目前昆明湖使用后一种方案。

为时序优化考虑，栈顶项的读取/更新并非在收到读取请求的当拍完成，而是在其上一拍或上 N 拍根据当拍的压/出栈操作更新。为减少对推测队列的写操作，在 BPU 2 流水级的 push/pop 结果也不会直接写入推测队列，而是延迟一拍后写入。考虑到存在本拍写入的数据在下一拍需要获取读数据的情况，设计了写 bypass 机制，准备写入的数据将在本拍首先用于更新写 bypass 的 writeEntry 项。下一拍要求读取的指针若与写 bypass 记录的指针位置匹配，则使用 bypass 值，否则使用从栈顶读取的值（实际因时序优化考虑这一逻辑被提前到读取栈

顶的上一拍)。

1.7.2.2 2 阶段结果

在 BPU 2 阶段, 由于其他分支预测器的预测结果还未完全确定, 可能存在需要更新的预测结果, 当前的预测结果并非最终确定的推测执行路径。当前的预测结果对同一时刻位于 3 阶段的分支预测块不会改变当前 2 阶段预测块的起始地址和现在位于 call/ret 指令前的其他分支指令不会再被预测跳转作出了成立假设。若 2 阶段从 FTB 传来的 FTB 项有效且其中存在 push 类指令, 则将该指令之后下一指令的 PC 值压入 RAS 栈; 若 2 阶段从 FTB 预测器传来的 FTB 项有效且其中存在 pop 类指令, 则将当前栈顶的地址作为结果返回并对结果出栈

在 RAS 内, 上述行为具体分解如下

在 2 阶段的 FTB 项内看到预测跳转的 call 指令, 则拉高 s2_spec_push 信号并根据当前 call 指令 pc 生成压栈的地址信息, 指示内部 RASStack 模块动作。RASStack 模块在检测到 push 动作时, 会利用传入的栈顶地址作为新写入持久化队列 entry 的预测地址, 若新写入地址与原地址相同且原栈顶 counter 未饱和, 则基于原栈顶项 counter+1 作为新项 counter, 否则新项 counter 取 0。生成的新 entry 被用于以下三个用途: 1, 下一拍更新 writeByassEntry 寄存器, 供连续预测读取栈顶项使用; 2, 当拍内更新栈顶项, 供连续预测读取使用; 3, 打拍后用于在下下拍更新持久化队列。与此同时, 如上所述, TOSR 被更新为当前的 TOSW, TOSW 更新为当前 TOSW+1, 全局的 ssp、sctr 类似上述 counter 更新算法, 若新旧栈顶地址相同且原 sctr (与原栈顶 entry 的 ctr 相同) 未饱和, 则 ssp 不变, sctr=sctr+1; 否则 ssp=ssp+1, sctr=0。为处理可能的持久化队列溢出情形, 若此时持久化队列接近溢出, 返回堆栈预测器将被暂停。

在 2 阶段的 FTB 项内看到预测跳转的 call 指令, 则拉高 s2_spec_pop 信号, 指示内部 RASStack 模块动作。RASStack 模块在检测到 pop 动作时, 若当前 sctr 非 0, 则 sctr=sctr-1, ssp 不变; 否则 ssp=ssp-1, sctr= 新栈顶项的 sctr。TOSR= 原 top 的 NOS, TOSW 保持不变。单独维护的栈顶项也会利用更新后的 ssp、sctr、TOSR 和 TOSW 来更新

1.7.2.3 3 阶段结果

在 3 阶段的 FTB 项内看到预测跳转的 return 指令, 则拉高 s3_push 信号; 在 3 阶段的 FTB 项内看到预测跳转的 call 指令, 则拉高 s3_pop 信号。当前预测块在 2 阶段作出的预测结果也分别被打拍到 3 阶段 (s3_pushed_in_s2 和 s3_poped_in_s2)。若 2 阶段与 3 阶段判断要采取的动作不同, 则需要在 3 阶段进行恢复, 无论是否恢复, 3 阶段均使用 2 阶段读出的 RAS 栈顶项作为预测结果

因为 2 阶段的 push/pop 和 3 阶段的 push 和 pop 仅存在如下几种情况, 因而 3 阶段可通过 push/pop 操作撤销 2 阶段的操作。

| | | |
|---------|---------|---------------|
| S2 push | S3 push | No fix needed |
| S2 push | S3 keep | Fix by pop |
| S2 keep | S3 push | Fix by push |
| S2 keep | S3 keep | No fix needed |
| S2 keep | S3 pop | Fix by pop |
| S2 pop | S3 keep | Fix by push |
| S2 pop | S3 pop | No fix needed |

3 阶段 push/pop 操作在 RASStack 内的具体动作与 2 阶段相同, 此处不再重复

1.7.2.4 误预测状态恢复

在预测块离开 BPU 后，在 IFU 或后端执行时可能会触发重定向，在遇到重定向时，RAS 预测器需要进行状态恢复。具体地，RAS 的 TOSR、TOSW、ssp 和 sctr 都需要根据误预测发生前的对应 meta 信息恢复，随后，根据误预测的指令自身是否为 call/return 指令，还需要分别通过 push 和 pop 操作来调整栈结构。Push 和 pop 操作的具体动作同 2、3 预测流水级，此处略。

1.7.2.5 提交 entry 迁移

如上所述，在含 call 指令的预测块提交时，BOS 会被更新为预测时的 TOSW，同时将预测块对应的 entry 写入 commit stack 栈顶（以 nsp 寻址）并更新 nsp。Nsp 更新算法类似 ssp，若存在递归且栈顶 counter 未满则 counter=counter+1，nsp 不变，否则 nsp=nsp+1，counter=0。

1.7.3 整体框图

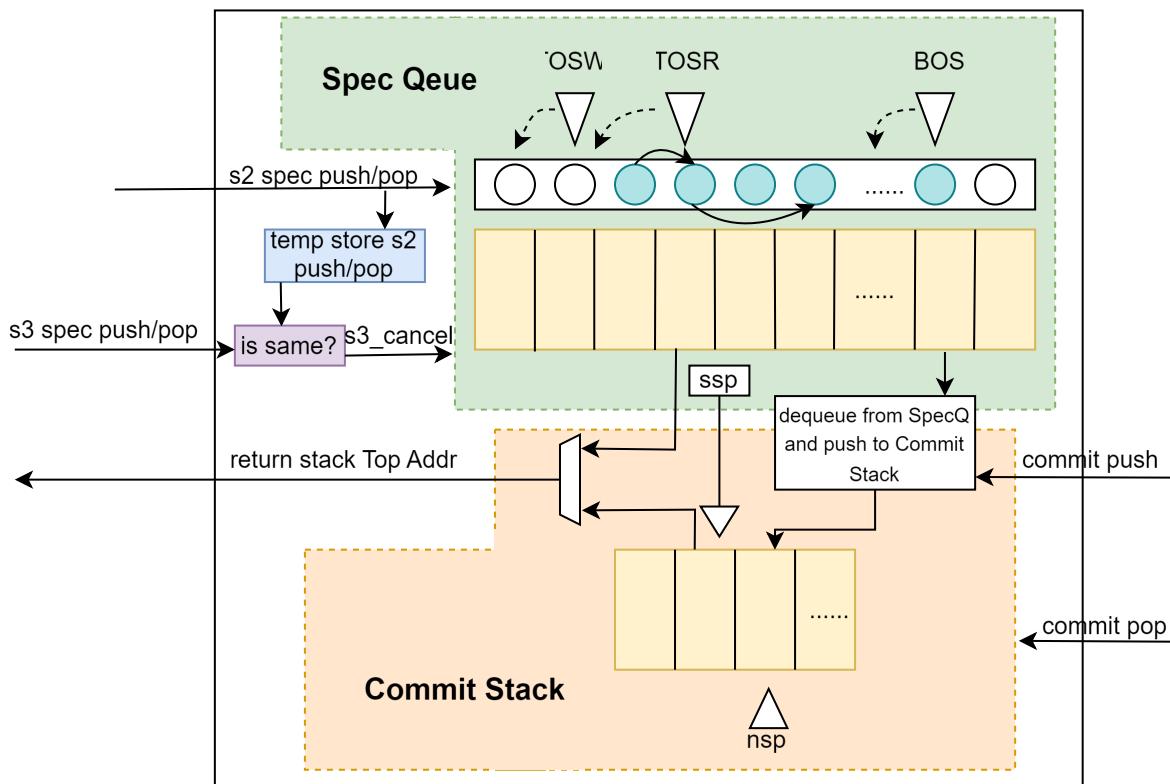


图 1.27: 整体框图

1.7.4 接口时序

1.7.4.1 RAS 模块 2 阶段更新输入输出接口

上图展示了 RAS 模块 2 阶段更新的一次 pop 和一次 push，由于 push 和 pop 的块前一个分支预测 slot 均无效，在本流水级看到 return 和 call 指令跳转，因而分别指示 RASStack 模块出/压栈。在压栈时，使用 FTB 的 fallThrough 地址作为跳转返回地址。若最后一条指令为被截断的 RVI call 指令，则该地址 +2 才是正确的返回地址。

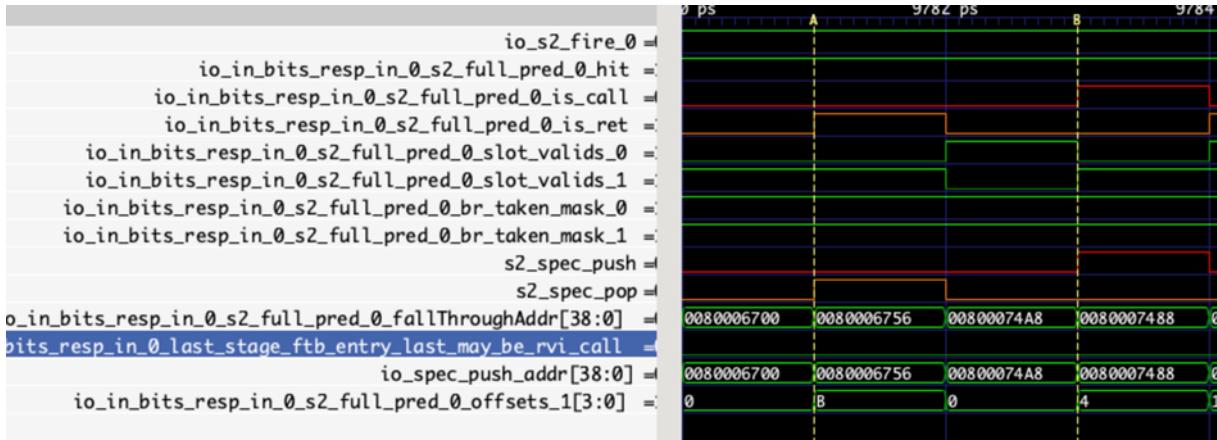


图 1.28: 2 阶段更新输入输出接口

1.7.4.2 RAS 模块 3 阶段更新输入输出接口

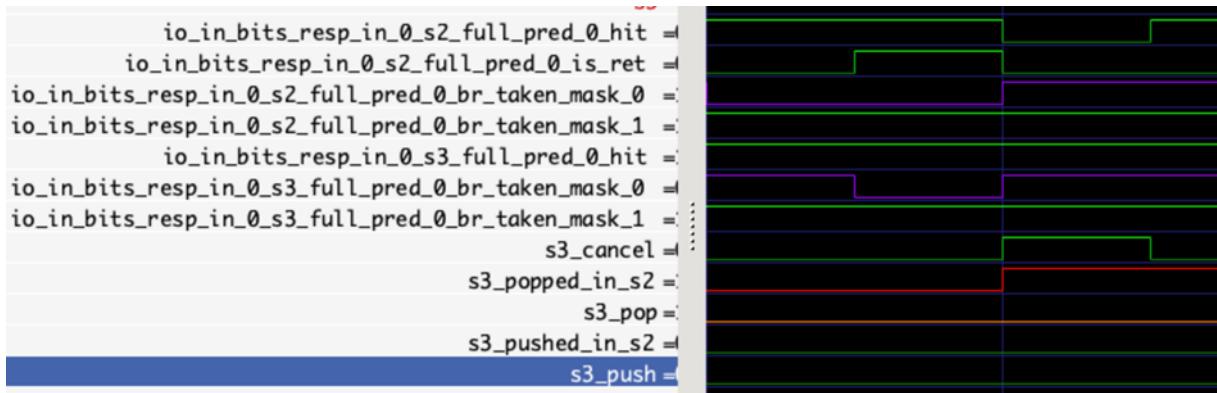


图 1.29: 3 阶段更新输入输出接口

1.7.4.3 RASStack 模块输入接口



图 1.30: Stack 模块输入接口

上图展示了 RASStack 模块 2 阶段更新的一次 push 和一次 pop 操作，可以注意到，在 push 操作后，从 RASStack 模块中读出的栈顶被更新为新 push 的值，而 pop 操作后，栈顶恢复为 push 操作前的值

1.7.4.4 RAS 模块重定向恢复接口

上图展示了 RAS 和 RASStack 模块重定向恢复且恢复点指令为 call 指令的情形，BPU 传入的重定向信号在 RAS 预测器内部被打一拍后送入 RASStack，用于恢复持久化队列中各项指针。由于该误预测点指令为 call 指令，还需压入新项

类似地，若误预测点指令为 return 指令，则需基于当时的栈顶 pop 出一项

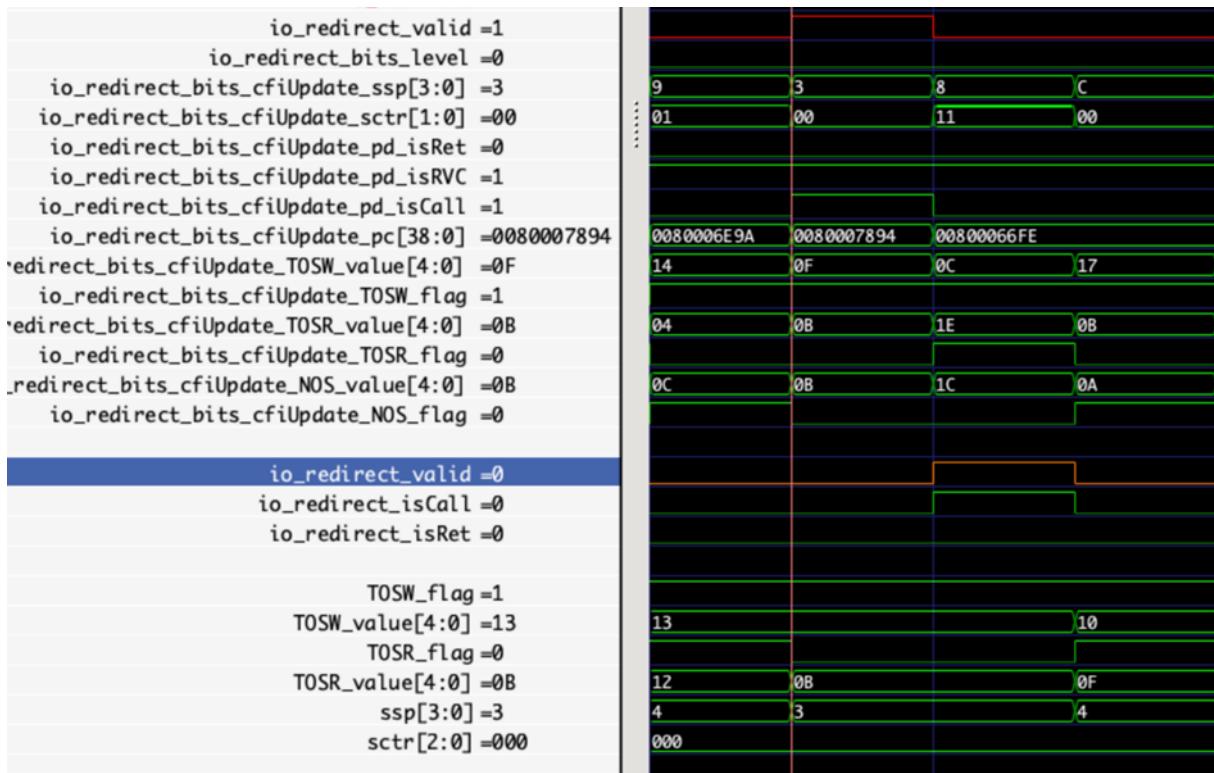


图 1.31: 重定向恢复接口

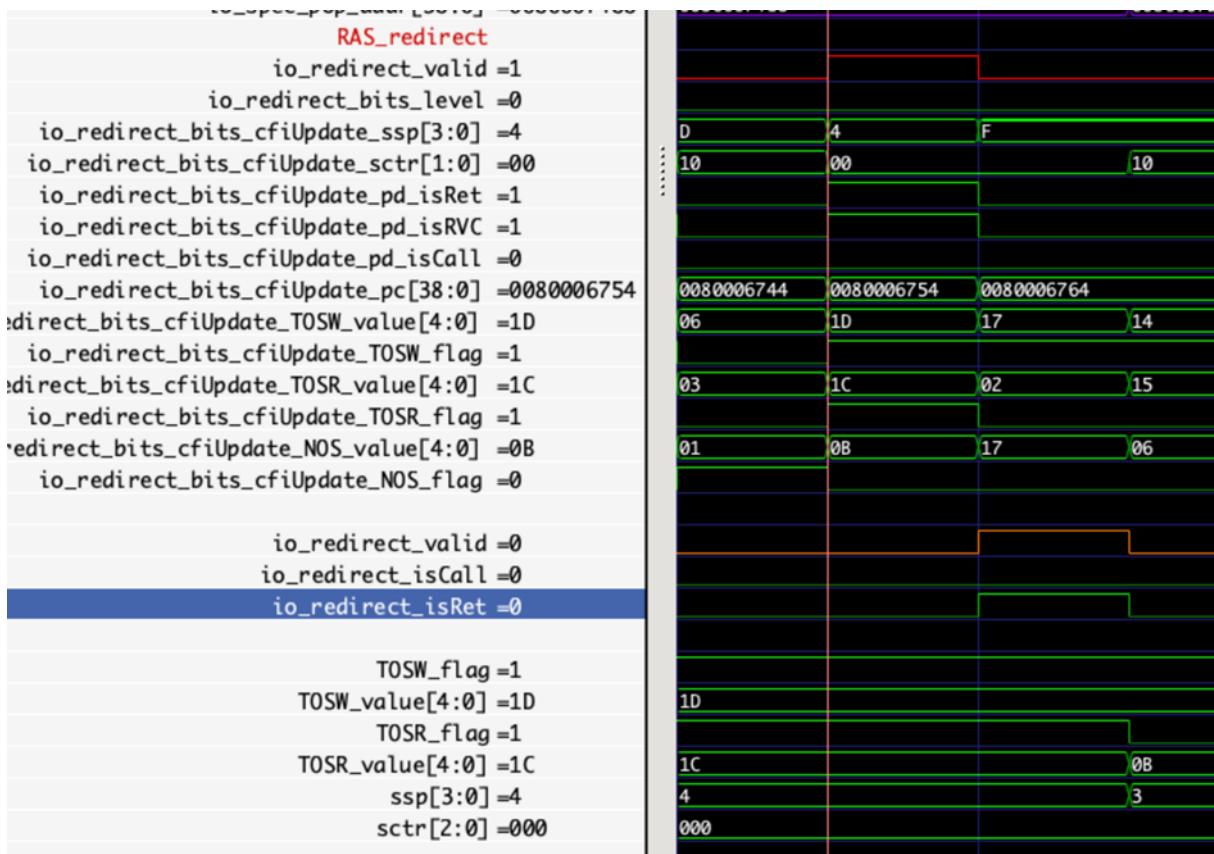


图 1.32: 重定向恢复接口

1.7.4.5 RAS 模块指令提交训练接口

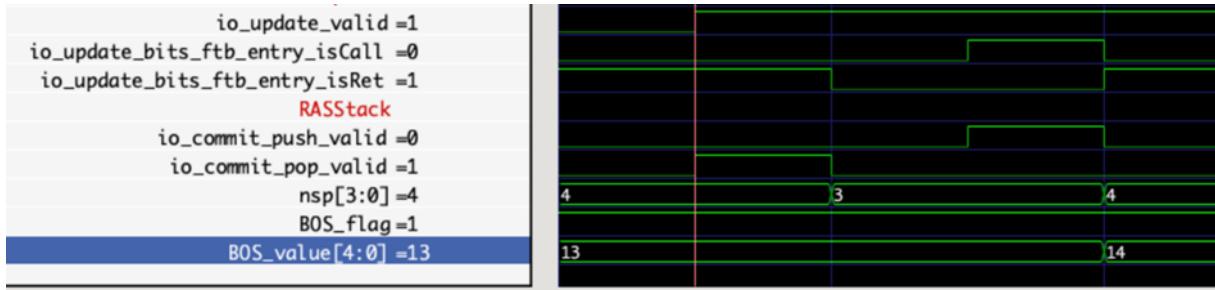


图 1.33: 指令提交训练接口

如图展示了一次含 return 和一次含 call 指令的指令提交, 可以看到, 在提交时提交栈顶发生变化, 且在 call 指令提交后 BOS 指针也相应调整

1.7.5 RAS 存储结构

返回堆栈预测器分为推测队列（持久化队列）和提交栈。推测队列有 32 项，提交栈有 16 项。

推测队列项结构如下

| retAddr | sctr | nos |
|---------|--------------|------------|
| 返回地址 | 连续出现相同返回地址次数 | 推测队列更旧一项位置 |

提交栈项结构如下

| retAddr | sctr |
|---------|--------------|
| 返回地址 | 连续出现相同返回地址次数 |

1.7.6 预测与更新

返回堆栈与其他预测器有所区别, 对于推测队列来说其在预测的同时实际上也作出了相应的更新。提交栈的更新则是在指令退休的时候。下图分别展示了返回堆栈预测地址的获取, 推测 Pop 以及推测 Push 时对推测队列的更新操作。

1.7.6.1 栈顶地址获取

返回堆栈的栈顶数据可能在推测栈中, 也有可能在提交栈中。当推测栈为空时, 栈顶数据就在提交栈中。推测栈和提交栈采用不同的策略进行维护。推测栈是推测执行的, 允许回退。提交栈的数据是真实有效的, 不允许回退。推测栈判不为空的逻辑是 $BOS \leq TOSR < TOSW$, 反之推测栈为空。(解释一下, 推测栈弹栈时 TOSR 是向 BOS 逼近的。因此 TOSR 不在推测栈内, 就说明推测栈为空。推测栈的数据去那了呢? 在提交栈中, 这与 BOS 的更新逻辑有关, 前提是推测栈不能溢出。)

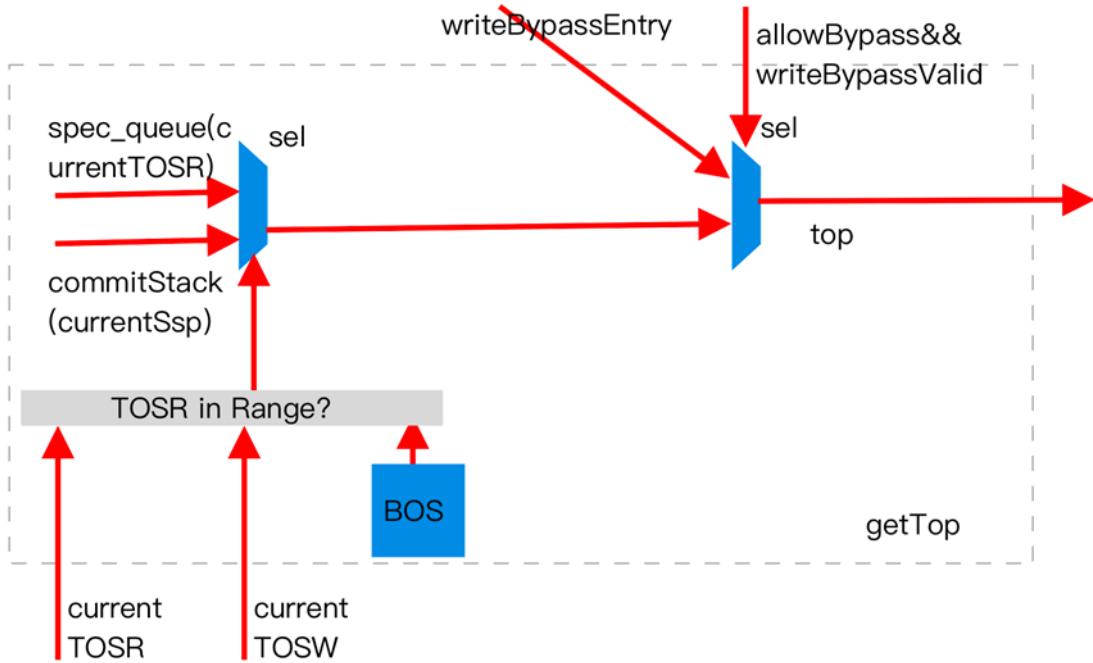


图 1.34: getTop 逻辑细节

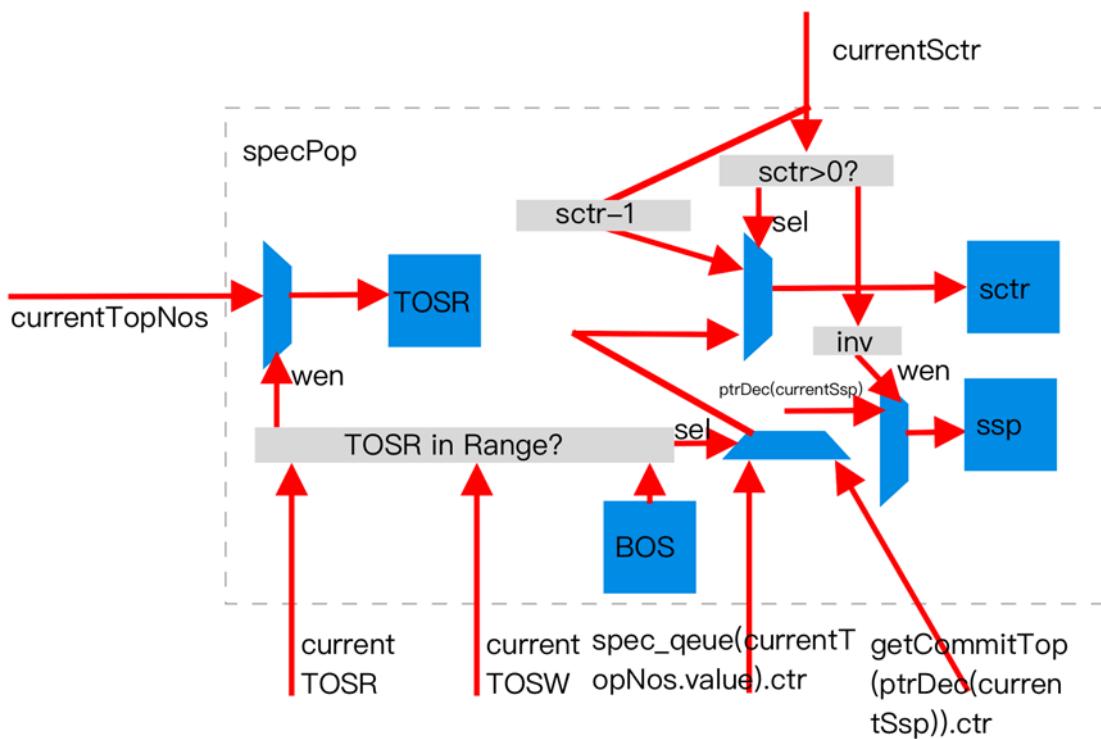


图 1.35: specPop 逻辑细节

1.7.6.2 推测弹栈

推测栈进行 Pop 的时候, 只会移动一个指针 TOSR, 用于指向当前推测栈的栈顶。 $TOSR := spec_nos(TOSR)$ 。NOS 不进行更新, 是因为每一个推测栈项都记录了 nos 点, TOSR 移动, 也就代表着当前的 NOS 跟着移动了。TOSW 不移动, 目的保留压栈历史, 便于追踪回溯。还有一个指针是 ssp, 用于记录预测过程中栈顶的位置。它的更新是按照原始的 RAS 栈结果进行更新的, Pop 的时候, ssp 减一。当然 ssp 是否减 1 需要结合 sctr 的值。

1.7.6.3 推测压栈

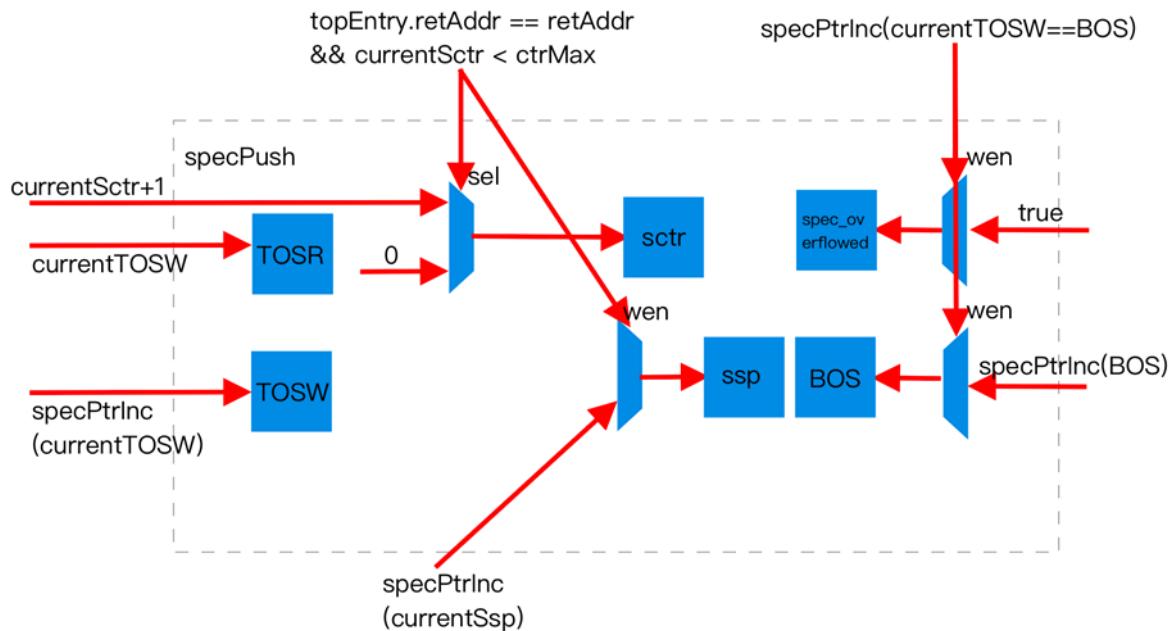


图 1.36: `specPush` 逻辑细节

推测栈进行 Push 的时候, $TOSW := TOSW + 1$, $spec_nos(TOSW) := TOSR$, $TOSR := TOSW$, $spec_queue(TOSW) := io.spec_push_addr$, $ssp = ssp + 1.U$ 。当压入新项时, `TOSW` 指向新的未分配项, `TOSR` 指向新的栈顶, `NOS` 指针将记录上一个栈顶的位置, 即为记录在 `spec_nos` 队列中, 便于后续的恢复使用。新压入项的地址将被记录在 `spec_queue` 中。(你可能会疑惑, 栈一直在增长, 炸了怎么办。— `BOS` 指针更新与栈溢出机制处理)

1.7.6.4 提交栈更新

提交栈的更新与通常的返回堆栈结构类似不再赘述, 稍有点区别是其 Push 和 Pop 信号来自于对应 call 和 ret 指令退休时。

2 昆明湖 FTQ 模块文档

2.1 术语说明

表 1.1 术语说明

| 缩写 | 全称 | 描述 |
|-----|---------------------|--------|
| CRU | Clock Reset Unit | 时钟复位单元 |
| FTQ | Fetch Target Queue | 取指目标队列 |
| FTB | Fetch Target Buffer | 取指目标缓冲 |

2.2 功能描述

2.2.1 功能概述

FTQ 是分支预测和取指单元之间的缓冲队列，它的主要职能是暂存 BPU 预测的取指目标，并根据这些取指目标给 IFU 发送取指请求。它的另一重要职能是暂存 BPU 各个预测器的预测信息，在指令提交后把这些信息送回 BPU 用作预测器的训练，因此它需要维护指令从预测到提交的完整的生命周期。

- 支持暂存 BPU 预测的取指目标，并向 IFU 发送取指请求
- 支持暂存 BPU 的预测信息，并送回 BPU 训练
- 支持重定向恢复
- 支持向 ICache 发送预取请求

2.2.2 暂存 BPU 预测的取指目标，并向 IFU 发送取指请求

2.2.2.1 暂存 BPU 预测的取指目标

2.2.2.1.1 存储 PC 的结构

BPU 的一次预测会经历三个流水级，每一个流水级都会产生新的预测内容。FTQ 接收来自 BPU 每个流水级的预测结果，并且后面的流水级的结果会覆盖前面流水级的结果。

指令以预测块为单位，从 BPU 发出，进入 FTQ，同时 bpuPtr 指针加一，初始化对应 FTQ 项的各种状态，把各种预测信息写入存储结构；如果预测块来自 BPU 覆盖预测逻辑，则恢复 bpuPtr 和 ifuPtr。

BPU 预测的取值目标被 FTQ 暂存于 ftq_pc_mem 中：

- ftq_pc_mem：寄存器堆实现，为存储与指令地址相关的信息，包括如下的域：

- startAddr 预测块起始地址。
- nextLineAddr 预测块下一个缓存行的起始地址。
- isNextMask 预测块每一条可能的指令起始位置是否在按预测宽度对齐的下一个区域内。isNextMask 有 16bit，每个 bit 表示相对起始地址的 2byte^n 位置是否跨 cacheline，表示的是每个位置的性质。
- fallThruError 预测出的下一个顺序取指地址是否存在错误。

每一个域都各自存在自己的寄存器（例如 data_0_startAddr）里，并没有拼接后存进同一个 Reg 里。

2.2.2.1.2 计算 PC 的方式

每次从 ICache 取指都会取一个或两个 CacheLineSize (64Bytes) 长度的缓存行指令数据，是否取两个由预测块是否跨缓存行决定。

而每个预测块的长度为 PredictWidth (16) 个压缩指令的长度 (32Bytes)。每个缓存行的长度为每个预测块长度的两倍，所以每个预测块的 startAddr 要么在当前缓存行的前半部分 (startAddr[5]=0)，要么在当前缓存行的后半部分 (startAddr[5]=1)。

如果 startAddr[5]=0,那么当前预测块必然不会跨缓存行,那么此时预测指令 pc={startAddr[38,6],startAddr[5,1]+offset}

如果 startAddr[5]=1, 那么当前预测块可能会出现跨缓存行的情况。此时:

- 如果 isNextMask(offset)=0,表示当前预测指令 pc 未跨缓存行,那么此时预测指令 pc={startAddr[38,6],startAddr[5,1]}
- 如果 isNextMask(offset)=1,表示当前预测指令 pc 跨越了缓存行,那么此时预测指令 pc={nextLineAddr[38,6],startAddr[5,1]+offset}

2.2.2.2 向 IFU 发送取指请求

FTQ 向 IFU 发出取指请求， ifuPtr 指针加一，等待预译码信息写回。

IFU 写回的预译码信息被 FTQ 暂存于 ftq_pd_mem 中：

- ftq_pd_mem: 寄存器堆实现，存储取指单元返回的预测块内的各条指令的译码信息，包括如下的域：
 - brMask 每条指令是否是条件分支指令。
 - jmpInfo 预测块末尾无条件跳转指令的信息，包括它是否存在、是 jal 还是 jalr、是否是 call 或 ret 指令。
 - jmpOffset 预测块末尾无条件跳转指令的位置。
 - jalTarget 预测块末尾 jal 的跳转地址。
 - rvcMask 每条指令是否是压缩指令。

2.2.3 暂存 BPU 的预测信息，并送回 BPU 训练

2.2.3.1 暂存 BPU 的预测信息

BPU 传给 FTQ 的预测信息除了会暂存到上文提到的 ftq_pc_mem 中，还有部分信息会存储到 ftq_redirect_sram、ftq_pc_mem 和 ftb_entry_mem 中。

- ftq_redirect_sram:SRAM 实现，存储那些在重定向时需要恢复的预测信息，主要包括和 RAS 和分支历史相关的信息。分为 3 个 bank，每个 bank 的深度 \times 宽度为 64×236 。
- ftq_meta_1r_sram:SRAM 实现，存储其余的 BPU 预测信息。SRAM 的深度 \times 宽度为 64×256 。
- ftb_entry_mem: 寄存器堆实现，存储预测时 FTB 项的必要信息，用于提交后训练新的 FTB 项。为什么要存 ftb_entry 呢？因为更新的时候 ftb_entry 需要在原来的基础上继续修改，为了不重新读一遍 ftb，所以这里将 ftb_entry 存在 ftb_entry_mem 中。

FTQ 中的各个 sram/mem 的具体实现机制见下表：

| 写入 时机 (正 向写 入) | 更新时机 (反向更新, 比如重定向等) | 读出时机 | 写入的数据内容 | 更新 数 据 内 容 |
|--|--|---|---|------------------------|
| ftq_BPUmem 不存在 (目前的设计是流水级的 S1 阶段, 创建新的预测 en-try 时写入) | FTQ 汇总重定向发到 BPU 和 IFU, 等 bpu 再把重定向到新地址的阶段, 预测块重新入队的时候, 在 ftq_pc_mem 写入新的块, ftq_pc_mem 的项是表示当前预测块的地址, 而不包括 target, 所以不需要更新预测出错的那个块) | 读数据每个时钟周期都会存进 Reg。如果 IFU 不需要从 bypass 中读取数据, Reg 数据直连给 Icache 和 IFU | startAddr: 预测块起始地址 nextLineAddr: 预测块下一个缓存行的起始地址 isNextMask: 预测块每一条可能的指令起始位置是否在按预测宽度对齐的下一个区域内 (① 如果 isNextMask(offset) = 0, 表示当前预测指令 pc 未跨缓存行, 那么此时预测指令 pc = {startAddr[38, 6], startAddr[5, 1] + offset, 1'b0}。② 如果 isNextMask(offset) = 1, 表示当前预测指令 pc 跨越了缓存行, 那么此时预测指令 pc = {nextLineAddr[38,6], startAddr[5, 1] + offset, 1'b0}。) fallThruError: 预测出的下一个顺序取指地址是否存在错误 | 无 |
| ftq_Beta_1r_sram | FTQ 项中的指令能够 commit 的时候, 将 meta 数据读出, 发送给 bpu 训练 | FTQ 项中的指令能够 commit 的时候, 将 meta 数据读出, 发送给 bpu 训练 | 写入的数据包含了 4 个预测器的预测信息 | |
| ftb_BPBy_mem | 1.backend 重定向 2.ifu 写回 预译码信息 3.ifu 预译码检测 出错误发送重定向 | BrSlot: brSlot_offset/lower/tarStat/sharing/validTailSlot: tail-Slot_offset/lower/tarStat/sharing/validpftAddr,carry,isCall,isRet | | |

| 写入 时机 (正 向写 入) 更新时机 (反向更新, 比如重定向等) | 读出时机 | 写入的数据内容 | 更新 的 数 据 内 容 |
|--|--|--|-----------------------------|
| ftq_pEUMem 阶段 F3 流水 的下 一拍 | 一直在读 commPtr 作为地址 对应的数 据, 赋值 给 ftbEn- tryGen | rvcMaskbrMaskjmpInfojmpOffsetjalTarget | |

2.2.3.2 送回 BPU 训练

指令在后端提交时会通知 FTQ 此指令已经提交。当 FTQ 项中所有的有效指令都已在后端提交, commPtr 指针加一, 从存储结构中读出相应的信息, 送给 BPU 进行训练。

在昆明湖 V2 版本中, 使用 `commitStateQueue` 来记录一个 FTQ 项中指令提交的状态。注意, 由于这一设计并不完备, 且违背 BPU 的更新初衷, 在 V3 中已经联合后端将这一机制全部删除。

`commitStateQueue` 的每一位记录了 FTQ entry 中的指令是否被提交。

由于 V2 的后端会在 ROB 中重新压缩 FTQ entry, 因此并不能保证提交一个 entry 中的每条指令, 甚至不能保证每一个 entry 都有指令提交。判断一个 entry 是否被提交有如下几种可能:

- `robCommPtr` 在 `commPtr` 之前。也就是说, 后端已经开始提交之后 entry 的指令, 在 `robCommPtr` 指向的 entry 之前的 entry 一定都已经提交完成
- `commitStateQueue` 中最后一条指令被提交。entry 的最后一条指令被提交意味着这一 entry 已经全部被提交

在此以外, 还必须要考虑到, 后端存在 flush itself 的 redirect 请求, 这意味着这条指令自身也需要重新执行, 这包括异常、load replay 等情况。在这种情况下, 这一 entry 不应当被提交以更新 BPU, 否则会导致 BPU 准确率显著下降。

2.2.4 重定向恢复

每次预测后, RAS 的栈顶项和栈指针都会存入 FTQ 的 `ftq_redirect_sram`, 同时使用的 BPU 全局历史会存入 FTQ, 用于误预测恢复。

2.2.4.1 预译码检测出预测错误

FTQ 向 IFU 发出取指请求后, IFU 会向 FTQ 写回预译码信息, `ifuWbPtr` 指针加一。如果预译码检测出了预测错误, 则向 BPU 发送相应的重定向请求。FTQ 根据重定向信号中的 `ftqIdx` 恢复 `bpuPtr` 和 `ifuPtr`。

2.2.4.2 后端检测出错预测

如果指令在后端执行时检测出误预测，则通知 FTQ，FTQ 给 IFU 和 BPU 发送对应的重定向请求，同时 FTQ 根据重定向信号中的 ftqIdx 恢复 bpuPtr、ifuPtr 和 ifuWbPtr。

为了实现提前一拍读出在 ftq 中存储的重定向数据，减少 redirect 损失，后端会向 ftq 提前一拍（相对正式的后端 redirect 信号）传送 ftqIdxAhead 信号和 ftqIdxSelOH 信号。但是提前一拍后端无法及时得到准确的 ftqIdx，需要在 4 个 Alu 通路中进行仲裁，但是仲裁结果在正式的后端 redirect 信号有效时才能得到，所以 FTQ 得到的提前一拍 redirect 的 ftqIdx 信号需要四个通路都读。

- io.fromBackend.ftqIdxAhead: 7 个 FtqIdx。表示需要重定向的预测块在 ftq 中存储的索引。有 7 个是因为后端在最终仲裁前有 7 个可能产生 redirect 信号的通路，分别是 Jump1、Alu4、LdReplay1、Exception1，但是其中只有 Alu*4 产生的 redirect 信号我们会提前读，所以 ftqIdxAhead 实际用到的只有 4 个 FtqIdx。
- Io.fromBackend.ftqIdxSelOH: 4 位独热码 +valid，表示 4 条通路的 ftqIdxAhead 有效与否，高有效。

2.2.5 向 ICache 发送预取请求

由于 BPU 基本无阻塞，它经常能走到 IFU 的前面，于是 FTQ 中实现了将 BPU 提供的还没发到 IFU 的取指请求用作指令预取，直接向指令缓存发送预取请求。

2.3 整体框图

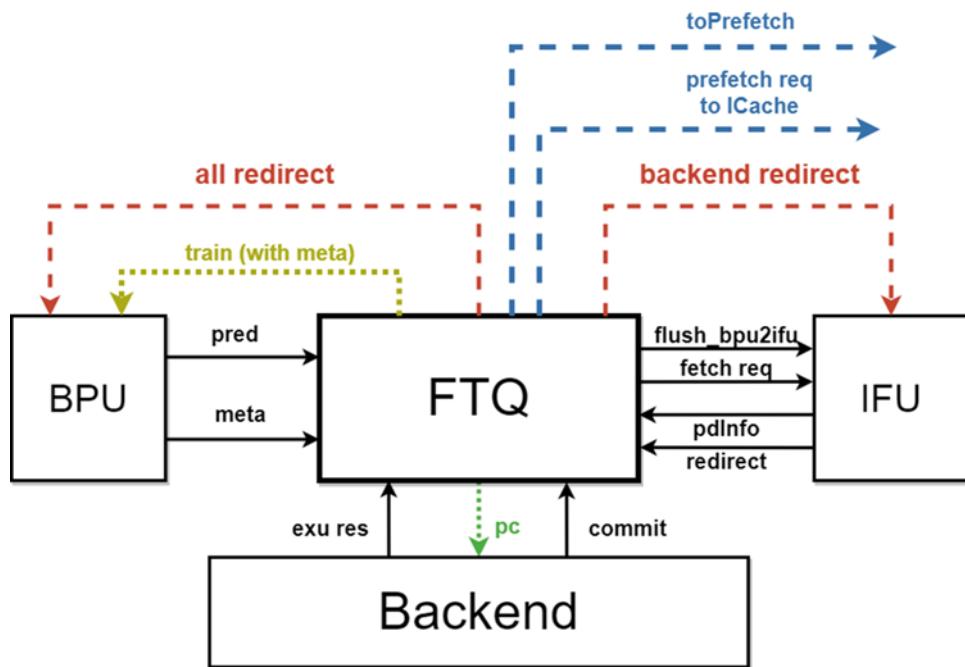


图 2.1: FTQ 结构

2.4 接口时序

1. BPU 到 FTQ 接口时序

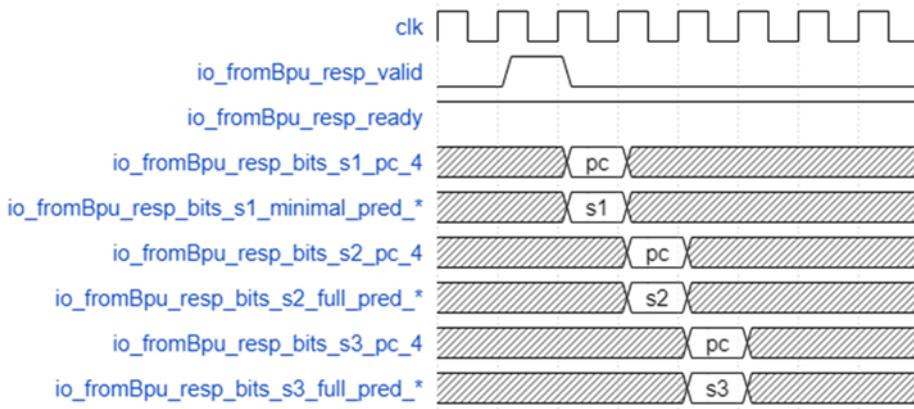


图 2.2: 接口时序

上图示意了 BPU 到 FTQ 的预测结果接口时序。当对应的握手信号 `io_fromBpu_resp_valid` 和 `io_fromBpu_resp_ready` 同时为高时, BPU 三个流水级的预测结果在流水线内 1、2、3 阶段分别输入至 FTQ。

若 BPU 后面流水级的预测结果与之前流水级不一致,则对应的 redirect 信号 `io_fromBpu_resp_bits_s2_hasRedirect_4` 或 `io_fromBpu_resp_bits_s3_hasRedirect_4` 会被拉高, 表明需要刷新预测流水线。

2.5 职能描述

FTQ 是分支预测和取指单元之间的缓冲队列, 它的主要职能是暂存 BPU 预测的取指目标, 并根据这些取指目标给 IFU 发送取指请求。它的另一重要职能是暂存 BPU 各个预测器的预测信息, 在指令提交后把这些信息送回 BPU 用作预测器的训练, 因此它需要维护指令从预测到提交的完整的生命周期。由于后端存储 PC 的开销较大, 当后端需要指令 PC 的时候, 会到 FTQ 读取。

2.6 内部结构

FTQ 共 64 项, 是一个队列结构, 但队列中每一项的内容是根据其自身特点存储在不同的存储结构中的。这些存储结构主要包括以下几种:

- `ftq_pc_mem`: 寄存器堆实现, 存储与指令地址相关的信息, 包括如下的域
 - `startAddr` 预测块起始地址
 - `nextLineAddr` 预测块下一个缓存行的起始地址
 - `isNextMask` 预测块每一条可能的指令起始位置是否在按预测宽度对齐的下一个区域内
 - `fallThruError` 预测出的下一个顺序取指地址是否存在错误
- `ftq_pd_mem`: 寄存器堆实现, 存储取指单元返回的预测块内的各条指令的译码信息, 包括如下的域
 - `brMask` 每条指令是否是条件分支指令
 - `jmpInfo` 预测块末尾无条件跳转指令的信息, 包括它是否存在、是 jal 还是 jalr、是否是 call 或 ret 指令
 - `jmpOffset` 预测块末尾无条件跳转指令的位置
 - `jalTarget` 预测块末尾 jal 的跳转地址
 - `rvcMask` 每条指令是否是压缩指令

- ftq_redirect_sram: SRAM 实现，存储那些在重定向时需要恢复的预测信息，主要包括和 RAS 和分支历史相关的信息
- ftq_meta_1r_sram: SRAM 实现，存储其余的 BPU 预测信息
- ftb_entry_mem: 寄存器堆实现，存储预测时 FTB 项的必要信息，用于提交后训练新的 FTB 项

另外还有一些例如队列指针、队列中各项的状态之类的信息用寄存器实现。

2.7 指令在 FTQ 中的生存周期

指令以预测块为单位，从 BPU 预测后便送进 FTQ，直到指令所在的预测块中的所有指令全部在后端提交完成，FTQ 才会在存储结构中完全释放该预测块所对应的项。这个过程中发生的事如下：

1. 预测块从 BPU 发出，进入 FTQ，bpuPtr 指针加一，初始化对应 FTQ 项的各种状态，把各种预测信息写入存储结构；如果预测块来自 BPU 覆盖预测逻辑，则恢复 bpuPtr 和 ifuPtr
2. FTQ 向 IFU 发出取指请求，ifuPtr 指针加一，等待预译码信息写回
3. IFU 写回预译码信息，ifuWbPtr 指针加一，如果预译码检测出了预测错误，则给 BPU 发送相应的重定向请求，恢复 bpuPtr 和 ifuPtr
4. 指令进入后端执行，如果后端检测出了误预测，则通知 FTQ，给 IFU 和 BPU 发送重定向请求，恢复 bpuPtr 、ifuPtr 和 ifuWbPtr
5. 指令在后端提交，通知 FTQ，等 FTQ 项中所有的有效指令都已提交，commPtr 指针加一，从存储结构中读出相应的信息，送给 BPU 进行训练

预测块 n 内指令的生存周期会涉及到 FTQ 中的 bpuPtr、ifuPtr、ifuWbPtr 和 commPtr 四个指针，当 bpuPtr 开始指向 n+1 时，预测块内的指令进入生存周期，当 commPtr 指向 n+1 后，预测块内的指令完成生存周期。

2.8 FTQ 的其它功能

由于 BPU 基本无阻塞，它经常能走到 IFU 的前面，于是 BPU 提供的这些还没发到 IFU 的取指请求就可以用作指令预取，FTQ 中实现了这部分逻辑，直接给指令缓存发送预取请求

3 取指令单元

3.1 昆明湖 IFU 模块文档

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/03
- commit: 7d889d887f665295eec9cdb987e037e008f875a6

3.1.1 术语说明

| 缩写 | 全称 | 描述 |
|--------------|--|--------------------------------|
| CRU | Clock Reset Unit | 时钟复位单元 |
| RVC | RISC-V Compressed Instructions | RISC-V 手册” C” 扩展规定的 16 位长度压缩指令 |
| RVI | RISC-V Integer Instructions | RISC-V 手册规定的 32 位基本整型指令 |
| IFU | Instruction Fetch Unit | 取指令单元 |
| FTQ | Fetch Target Queue | 取指目标队列 |
| PreDecode | Predecoder Module | 预译码器 |
| PredChecker | Prediction Check Module | 分支预测结果检查器 |
| ICache | L1 Instruction Cache | 一级指令缓存 |
| IBuffer | Instruction Buffer | 指令缓冲 |
| CFI | Control Flow Instruction | 控制流指令 |
| PC | Program Counter | 程序计数器 |
| ITLB | Instruction Translation Lookaside Buffer | 指令地址转译后备缓冲器 |
| InstrUncache | Instruction Ucache Module | 指令 MMIO 取指处理单元 |

3.1.2 子模块列表

| 子模块 | 描述 |
|--------------|----------------|
| PreDecoder | 预译码模块 |
| InstrUncache | 指令 MMIO 取指处理单元 |

3.1.3 功能描述

FTQ 将预测块请求分别发送到 ICache 和 IFU 模块，IFU 等到来自 ICache 返回至多两个缓存行的指令码后，进行切分产生取指令请求范围限定的初始指令码，并送到预译码器进行预译码下一拍根据预译码信息修正有效指令范围，同时进行指令码扩展并将指令码及其他信息发送给 IBuffer 模块。当 ICache 查询地址属性发现是 MMIO 地址空间时，IFU 需要将地址发送给 MMIO 处理单元取指令，这个时候处理器进入多周期顺序执行模式，IFU 阻塞流水线直到收到来自 ROB 的提交信号时，IFU 才允许下一个取指令请求的进行，同时 IFU 需要对跨页的 MMIO 地址空间 32 位指令做特殊处理（重发机制）。

3.1.3.1 接受 FTQ 取指令请求

IFU 接收来自 FTQ 以预测块为单位的取指令请求，包括预测块起始地址、起始地址所在 cacheline 的下一个 cacheline 开始地址、下一个预测块的起始地址、该预测块在 FTQ 里的队列指针、该预测块有无 taken 的 CFI 指令和该 taken 的 CFI 指令在预测块里的位置以及请求控制信号（请求是否有效和 IFU 是否 ready）。每个预测块最多包含 32 字节指令码，最多为 16 条指令。

3.1.3.2 双 cacheline 取指

当且仅当预测块的取指地址在 cacheline 的后半段时，为了满足一个预测块最多 34 字节的需要，IFU 将从 ICache 中取回连续的两个 cacheline，分别产生例外信息（page fault 和 access fault），如后述特性 3 进行切分。

在 2024/06 以后，ICache 实现了低功耗设计，会在内部进行数据的选择和拼接，因此 IFU 不需要关心两个 cacheline 的数据如何拼接和选择，只需要简单地将 ICache 返回的数据复制一份拼接在一起，即可进行切分。请参考 ICache 文档。

亦可参考 IFU.scala 中的注释。

3.1.3.3 指令切分产生初始指令码

下一流水级（F1 级），计算出预测块内每 2 字节的 PC 和其他一些信息，然后进入 F2 流水级等待 ICache 返回指令码，在 F2 级需要检查 ICache 返回的指令码和本流水级是否匹配（因为 IFU 的流水级会被冲刷而 ICache 不会）。然后根据 ICache 返回的缓存行例外信息产生每条指令的例外信息（page fault 和 access fault），同时根据 FTQ 的 taken 信息计算一个跳转时指令有效范围 jump_range（即此预测块从起始地址到第一条跳转地址的指令范围）和无跳转时指令有效范围 ftr_range（即此预测块从起始地址到下一个预测块的起始地址）。为了时序相关的考虑，ICache 的两个端口分别会返回 miss 和 hit 时候两个来源的缓存行，这个四个缓存行需要产生 4 种组合（0 号端口的两个和 1 号端口的两个）同时进行预译码。F2 会并行对返回的 64 字节的数据中（其中 40 字节有效数据）根据预测块的起始地址选择出 17×2 字节的初始指令码，并送到 4 个 PreDecode 模块进行预译码。

3.1.3.4 产生预译码信息

PreDecode 模块接受 F2 切分后的 17 个 2 字节初始指令码，一方面将这些初始指令码根据译码表进行预译码得到预译码信息，包括该指令是否是有效指令的开始、是否是 RVC 指令、是否是 CFI 指令、CFI 指令类型（branch/jal/jalr/call/ret）、CFI 指令的目标地址计算偏移等。输出的预译码信息中 brType 域的编码如下：

表 1.2 CFI 指令类型编码

| CFI 指令类型 | 类型编码 (brType) |
|-----------|---------------|
| 非 CFI 指令 | 00 |
| branch 指令 | 01 |
| jal 指令 | 10 |
| jalr 指令 | 11 |

3.1.3.5 生成指令码和指令码扩展

产生预译码信息的同时将初始指令进行 4 字节组合（从起始地址开始，2 字节做地址递增，地址开始的 4 字节作为一条 32 位初始指令码）产生每条指令的指令码

在产生指令码和预译码信息的下一拍 (F3) 将 16 条指令的指令码分别送到 16 个指令扩展器进行 32 位指令扩展 (RVC 指令根据手册的规定进行扩充，RVI 保留指令码不变)。

3.1.3.6 分支预测 overriding 冲刷流水线

当 FTQ 内未缓存足够预测块时，IFU 可能直接使用简单分支预测器提供的预测地址进行取指，这种情况下，当精确预测器发现简单预测器错误时，需要通知 IFU 取消正在进行的取指请求。具体而言，当 BPU 的 S2 流水级发现错误时，需要冲刷 IFU 的 F0 流水级；当 BPU 的 S3 流水级发现错误时，需要冲刷 IFU 的 F0/F1 流水级 (BPU 的简单预测器在 S1 给出结果，最晚在 S3 进行 overriding，因此 IFU 的 F2/F3 流水级一定是最好的预测，不需要冲刷；类似地，不存在 BPU S2 到 IFU F1 的冲刷)。

IFU 在收到 BPU 发送的冲刷请求时，会将 F0/F1 流水级上取指请求的指针与 BPU 发送的冲刷请求的指针进行比较，若冲刷的指针在取指的指针之前，说明当前取指请求在错误的执行路径上，需要进行流水线冲刷；反之，IFU 可以忽略 BPU 发送的这一冲刷请求。

3.1.3.7 分支预测错误提前检查

为了减少一些比较容易识别的分支预测错误的冲刷，IFU 在 F3 流水级使用 F2 产生的预译码信息做前端的分支预测错误检查。预译码信息首先送到 PredChecker 模块，根据其中的 CFI 指令类型检查 jal 类型错误、ret 类型错误、无效指令预测错误、非 CFI 指令预测错误，同时根据指令码计算 16 个转移目标地址，和预测的目标地址进行比对，检查转移目标地址错误，PredChecker 将纠正 jal 类型错误 ret 错误的预测结果，并重新产生指令有效范围向量 fixedRange (为 1 表示该条指令在预测块内)，fixedRange 在 jump_range 和 ftr_range 的基础上根据 jal 和 ret 的检查结果，把范围缩小到其实地址到没有检测出来的 jal 或者 ret 指令。下面是 PredChecker 模块对分支预测检查的错误类型：

- jal 类型错误：预测块的范围内有 jal 指令，但是预测器没有对这条指令预测跳转；
- ret 类型错误：预测块的范围内有 ret 指令，但是预测器没有对这条指令预测跳转；
- 无效指令预测错误：预测器对一条无效的指令（不在预测块范围/是一条 32 位指令中间）进行了预测；
- 非 CFI 指令预测错误：预测器对一条有效但是不是 CFI 的指令进行了预测；
- 转移目标地址错误：预测器给出的转移目标地址不正确。

3.1.3.8 前端重定向

如果 F3 分支预测的检查结果显示这个预测块有特性 7 里所述的 5 种预测错误，那么 IFU 将在下一拍产生一个前端重定向，将除 F3 之外的流水级冲刷。FTQ 以及预测器的冲刷将由 IFU 写会 FTQ 后由 FTQ 完成。

3.1.3.9 将指令码和前端指令信息送到 IBuffer

F3 流水级最终得到经过扩展的 32 位指令码, 以及 16 条指令中每条指令的例外信息、预译码信息、FTQ 指针、其他后端需要的信息(比如经过折叠的 PC)等。IFU 除了常规的 valid-ready 控制信号外, 还会给 IBuffer 两个特殊的信号: 一个是 16 位的 io_toIbuffer_bits_valid, 标识预测块里有效的指令(为 1 说明是一条指令的开始, 为 0 则是说明是一条指令的中间)。另一个是 16 位的 io_toIbuffer_bits_enqEnable, 这个在 io_toIbuffer_bits_valid 的基础上与上了被修正过的预测块的指令范围 fixedRange。enqEnable 为 1 表示这个 2 字节指令码是一条指令的开始且在预测块表示的指令范围内。

3.1.3.10 指令信息和误预测信息写回 FTQ

在 F3 的下一级 WB 级, IFU 将指令 PC、预译码信息、错误预测指令的位置、正确的跳转地址以及预测块的正确指令范围等信息写回 FTQ, 同时传递该预测块的 FTQ 指针用以区分不同请求。

3.1.3.11 跨预测块 32 位指令处理

因为预测块的长度有限制, 因此存在一条 RVI 指令前后两字节分别在两个预测块的情况。IFU 首先在第一个预测块里检查最后 2 字节是不是一条 RVI 指令的开始, 如果是并且该预测块没有跳转, 那么就设置一个标识寄存器 f3_lastHalf_valid, 告诉接下来的预测块含有后半条指令。在 F2 预译码时, 会产生两种不同的指令有效向量:

- 预测块起始地址开始即为一条指令的开始, 以这种方式根据后续指令是 RVC 还是 RVI 产生指令有效向量
- 预测块起始地址是一条 RVI 指令的中间, 以起始地址 +2 位一条指令的开始产生有效向量

在 F3, 根据是否有跨预测块 RVI 标识来决定选用哪种作为最终的指令有效向量, 如果 f3_lastHalf_valid 为高则选择后一种(即这个预测块第一个 2 字节不是指令的开始)。如前面特性 2 所述, 当且仅当起始地址在后半 cacheline, 就会向 ICache 取两个 cacheline, 因此即使这条跨预测块的 RVI 指令也跨 cacheline, 每个预测块都能拿到它的完整指令码。IFU 所做的处理只是把这条指令算在第一个预测块里, 而把第二个预测块的起始地址位置的 2 字节通过改变指令有效向量来无效掉。

3.1.3.12 MMIO 取指令

在处理器上电解复位时, 由于内存初始化还未完成, 因此处理器需要从 flash 存储里取指令运行, 这种情况下需要 IFU 向 MMIO 总线发送宽度为 64 位的请求从 flash 地址空间取指令执行。同时 IFU 禁止对 MMIO 总线的推测执行, 即 IFU 需要等到每一条指令执行完成得到准确的下一条指令地址之后才继续向总线发送请求。

处理器上电解复位后, 从 0x10000000 地址开始取指令, ICache 经过 ITLB 地址翻译得到物理地址, 物理地址经过 PMP 查询是否属于 MMIO 空间, 并将检查结果返回到 IFU F2 流水级(见 ICache 文档)。如果是 MMIO 地址空间的取指令请求, IFU 将请求阻塞在 F3 并由一个状态机控制 MMIO 取指令, 由下图所示:

1. 状态机默认在 `m_idle` 状态, 若 F3 流水级是 MMIO 取指令请求, 且此前没有发生异常, 状态机进入 `m_waitLastCmt` 状态。
2. (`m_waitLastCmt`) IFU 通过 `mmioCommitRead` 端口到 FTQ 查询, IF3 预测块之前的指令是否都已提交, 如果没有提交则阻塞等待前面的指令都提交完¹。
3. (`m_sendReq`) 将请求发送到 InstrUncache 模块, 向 MMIO 总线发送请求。
4. (`m_waitResp`) InstrUncache 模块返回后根据 pc 从 64 位数据中截取指令码。

¹需要特别指出的是, Svpbmt 扩展增加了一个 NC 属性, 其代表该内存区域是不可缓存的、但是幂等的, 这意味着我们可以对 NC 的区域进行推测执行, 也就是不需要“等待前面的指令提交”就可以向总线发送取指请求, 表现为状态机跳过等待状态。实现见 #3944。

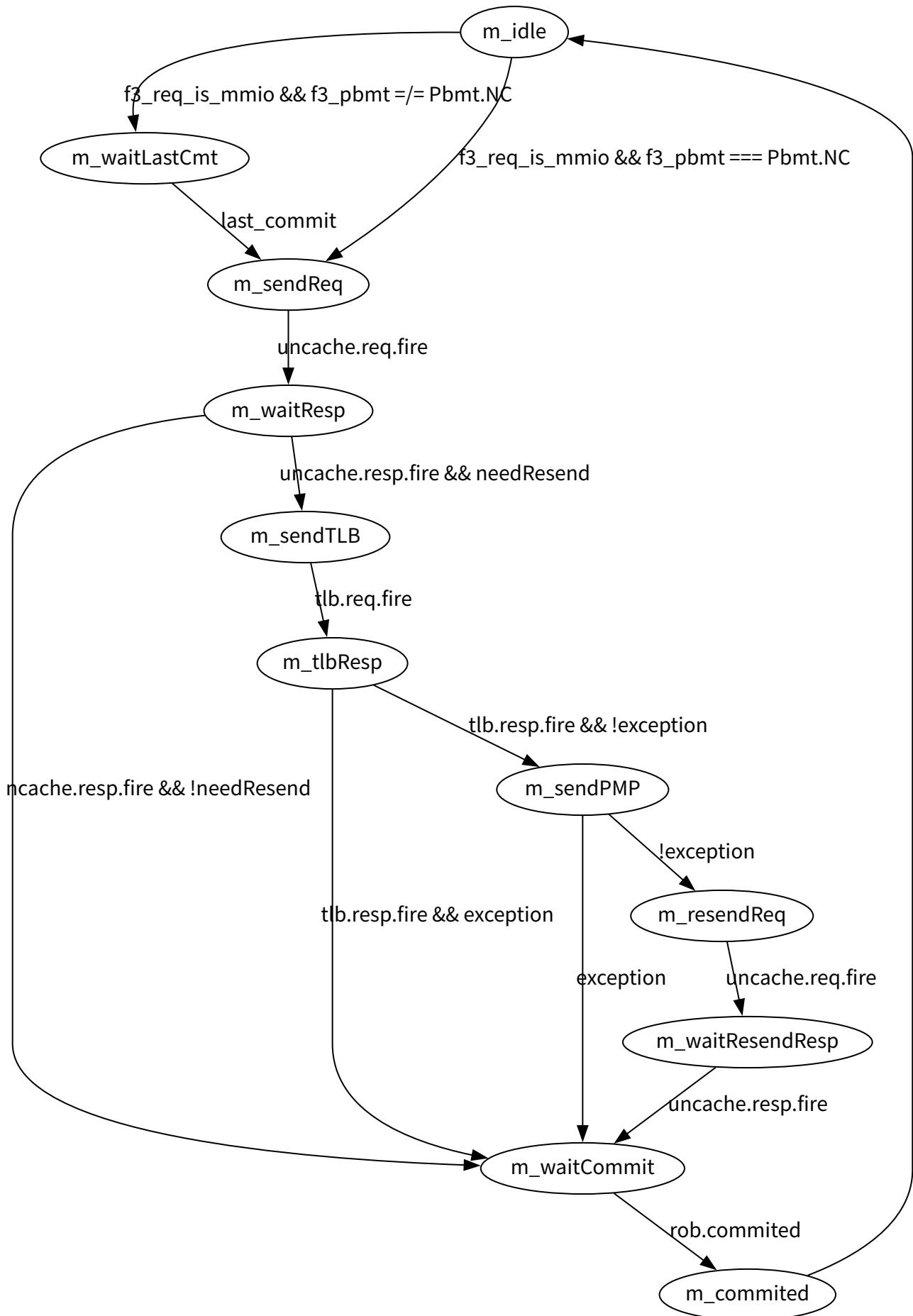


图 3.1: F3 MMIO 状态机示意图

5. 若 pc 低位为 $3'b110$, 由于 MMIO 总线的带宽限制为 8B 且只能访问对齐的区域, 本次请求的高 2B 将不是有效的数据。若返回的指令数据表明指令不是 RVC 指令, 则这种情况需要对 pc+2 的位置 (即对齐到下一个 8B 的位置) 进行重发才能取回完整的 4B 指令码。
1. 重发前, 需要重新对 pc+2 进行 ITLB 地址翻译和 PMP 检查 (因为可能跨页) (`m_sendTLB`、`m_TLBResp`、`m_sendPMP`), 若 ITLB 或 PMP 出现异常 (access fault、page fault、guest page fault)、或检查发现 pc+2 的位置不在 MMIO 地址空间, 则直接将异常信息发送到后端, 不进行取指。
2. 若无异常, (`m_resendReq`、`m_waitResendResp`) 类似 2/3 两步向 InstrUncache 发出请求并收到指令码。
6. 当 IFU 寄存了完整的指令码, 或出错 (重发时的 ITLB/PMP 出错, 或 Uncache 模块 tilelink 总线返回 corrupt) 时, (`m_waitCommit`) 即可将指令数据和异常信息发送到 IBuffer。需要注意, MMIO 取指令每次只能非推测性地向总线发起一条指令的取指请求, 因此也只能向 IBuffer 发送一条指令数据。并等待指令提交。
 1. 若这条指令是 CFI 指令, 由后端发送向 FTQ 发起冲刷。
 2. 若是顺序指令, 则由 IFU 复用前端重定向通路刷新流水线, 同时复用 FTQ 写回机制, 把它当作一条错误预测的指令进行冲刷, 重定向到该指令地址 +2 或者 +4 (根据这条指令是 RVI 还是 RVC 选择)。这一机制保证了 MMIO 每次只取出一条指令。
7. 提交后, (`m_committed`) 状态机复位到 `m_idle` 并清空各类寄存器。

除了上电时, debug 扩展、Svpbmt 扩展可能也会使处理器在运行的任意时刻跳到一块 MMIO 地址空间取指令, 请参考 RISC-V 手册。对这些情况中 MMIO 取指的处理是相同的。

3.1.3.13 Trigger 实现对于 PC 的硬件断点功能

在 IFU 的 FrontendTrigger 模块里共 4 个 Trigger, 编号为 0-3, 每个 Trigger 的配置信息 (断点类型、匹配地址等) 保存在 `tdata` 寄存器中。

当软件向 CSR 寄存器 `tselect`、`tdata1/2` 写入特定的值时, CSR 会向 IFU 发送 `tUpdate` 请求, 更新 FrontendTrigger 内的 `tdata` 寄存器中的配置信息。目前前端的 Trigger 仅可以配置成 PC 断点 (`mcontrol.select` 寄存器为 0; 当 `mcontrol.select=1` 时, 该 Trigger 将永远不会命中, 且不会产生异常)。

在取指时, IFU 的 F3 流水级会向 FrontendTrigger 模块发起查询并在同一周期得到结果。后者会对取指块内每一条指令在每一个 Trigger 上做检查, 当不处于 debug 模式时, 指令的 PC 和 `tdata2` 寄存器内容的关系满足 `mcontrol.match` 位所指示的关系 (香山支持 `mcontrol.match` 位为 0、2、3, 对应等于、大于、小于) 时, 该指令会被标记为 Trigger 命中, 随着执行在后端产生断点异常, 进入 M-Mode 或调试模式。前端的 Trigger 支持 Chain 功能。当它们对应的 `mcontrol.chain` 位被置时, 只有当该 Trigger 和编号在它后面一位的 Trigger 同时命中时, 处理器才会产生异常²。

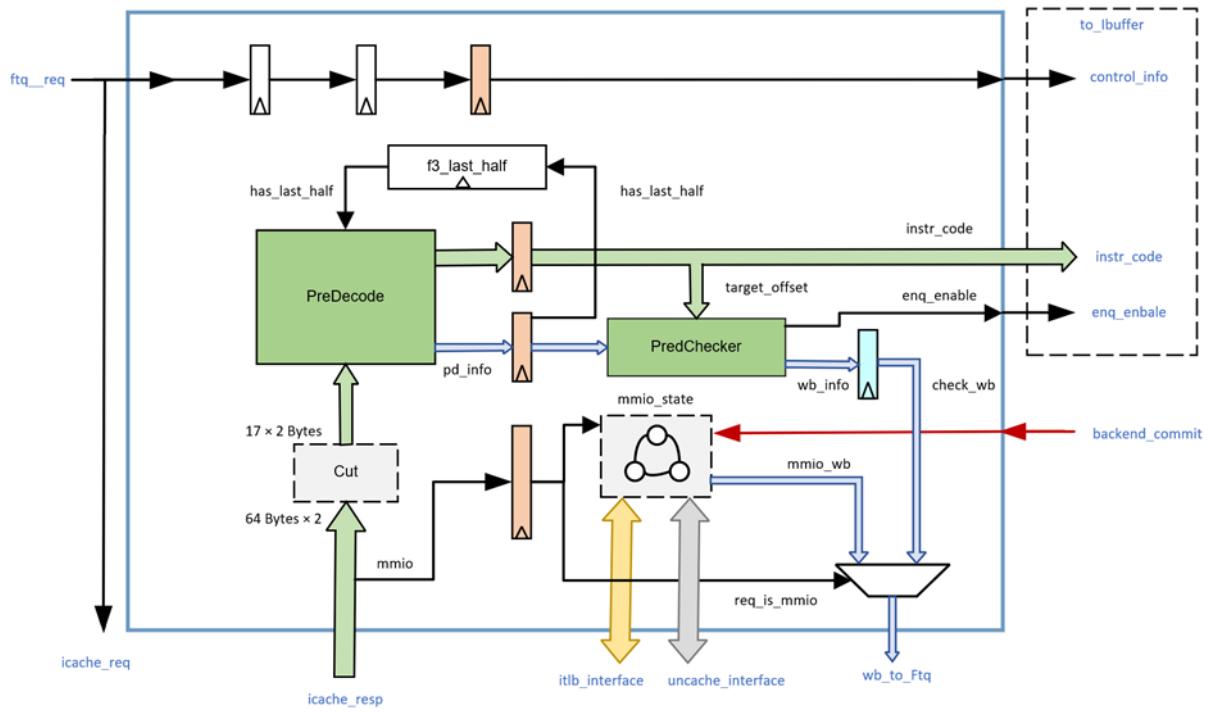


图 3.2: IFU 模块整体框图

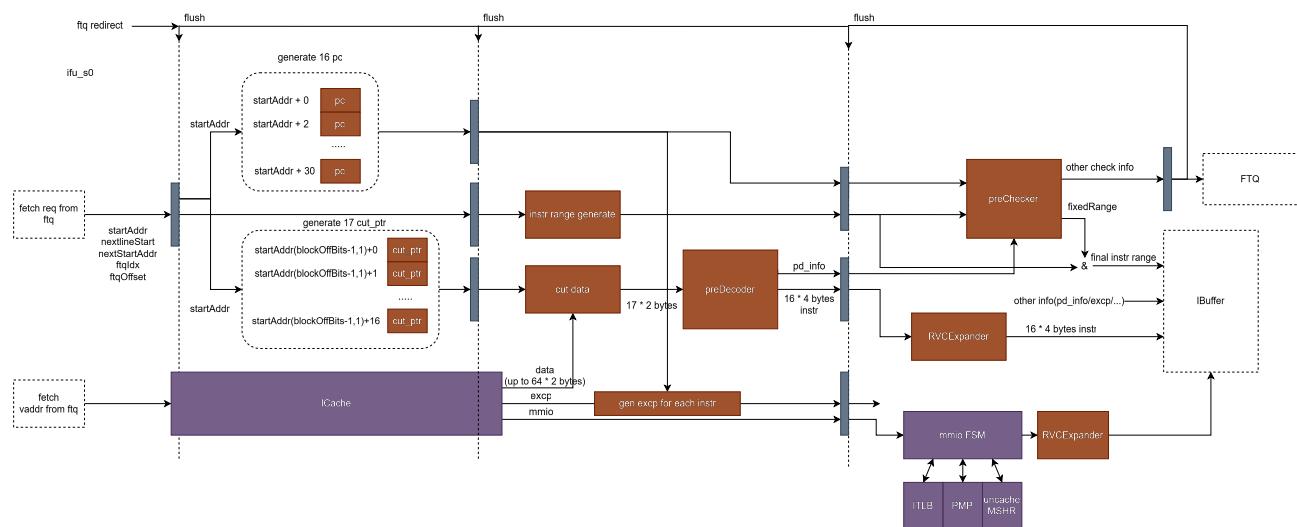


图 3.3: IFU 模块流水级

3.1.4 总体设计

3.1.4.1 整体框图和流水级

3.1.4.2 接口时序

3.1.4.2.1 FTQ 请求接口时序示例

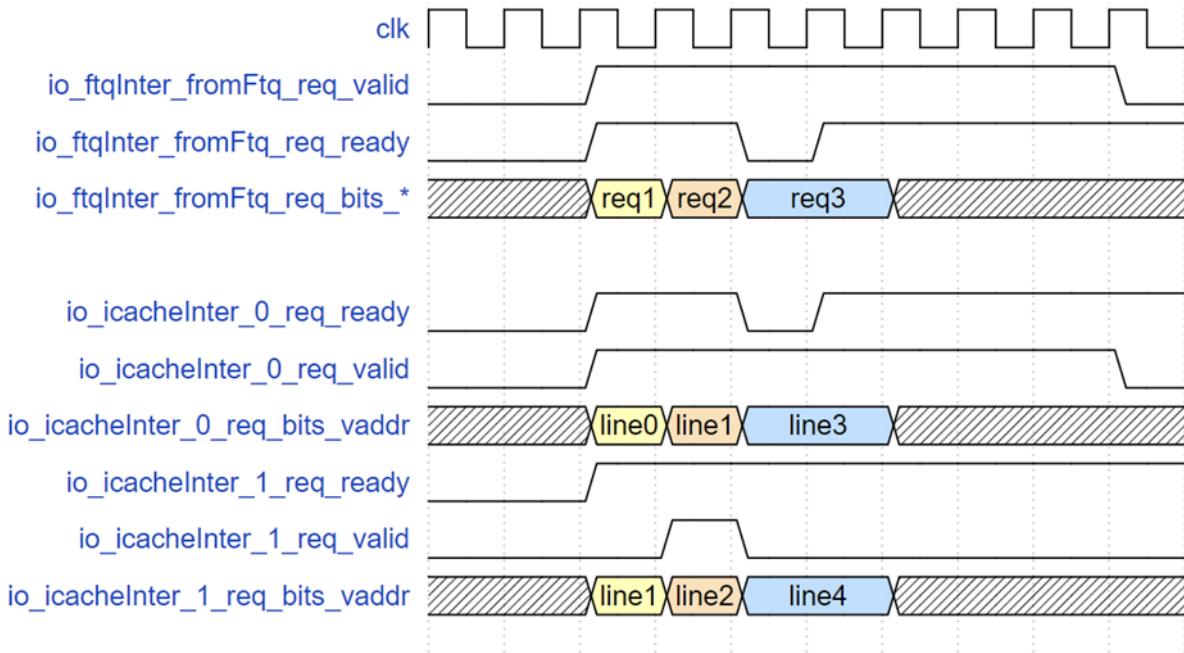


图 3.4: FTQ 请求接口时序示例

上图示意了三个 FTQ 请求的示例, req1 只请求缓存行 line0, 紧接着 req2 请求 line1 和 line2, 当到 req3 时, 由于指令缓存 SRAM 写优先, 此时指令缓存的读请求 ready 被指低, req3 请求的 valid 和地址保持直到请求被接收。

3.1.4.2.2 ICache 返回接口以及到 Ibuffer 和写回 FTQ 接口时序示例

上图展示了指令缓存返回数据到 IFU 发现误预测直到 FTQ 发送正确地址的时序, group0 对应的请求在 f2 阶段了两个缓存行 line0 和 line1, 下一拍 IFU 做误预测检查并同时把指令给 Ibuffer, 但此时后端流水线阻塞导致 Ibuffer 满, Ibuffer 接收端的 ready 置低, group0 相关信号保持直到请求被 Ibuffer 接收。但是 IFU 到 FTQ 的写回在 tio_toIbuffer_valid 有效的下一拍就拉高, 因为此时请求已经无阻塞地进入 wb 阶段, 这个阶段锁存的了 PredChecker 的检查结果, 报告 group0 第 4 (从 0 开始) 个 2 字节位置对应的指令发生了错误预测, 应该重定向到 vaddrA, 之后经过 4 拍 (冲刷和重新走预测器流水线), FTQ 重新发送给 IFU 以 vaddrA 为起始地址的预测块。

3.1.4.2.3 MMIO 请求接口时序示例

上图展示了一个 MMIO 请求 req1 的取指令时序, 首先 ICache 返回的 tlbExcp 信息报告了这是一条 MMIO 空间的指令 (其他例外信号必须为低), 过两拍 IFU 向 InstrUncache 发送请求, 一段时间后收到响应和 32 位

²在过去 (riscv-debug-spec-draft, 对应 XiangShan 2024.10.05 合入的 PR#3693 前) 的版本中, Chain 还需要满足两个 Trigger 的 mcontrol.timing 是相同的。而在新版 (riscv-debug-spec-v1.0.0) 中, mcontrol.timing 被移除。目前 XiangShan 的 scala 实现仍保留了这一位, 但其值永远为 0 且不可写入, 编译生成的 verilog 代码中没有这一位。参考: <https://github.com/riscv/riscv-debug-spec/pull/807>。

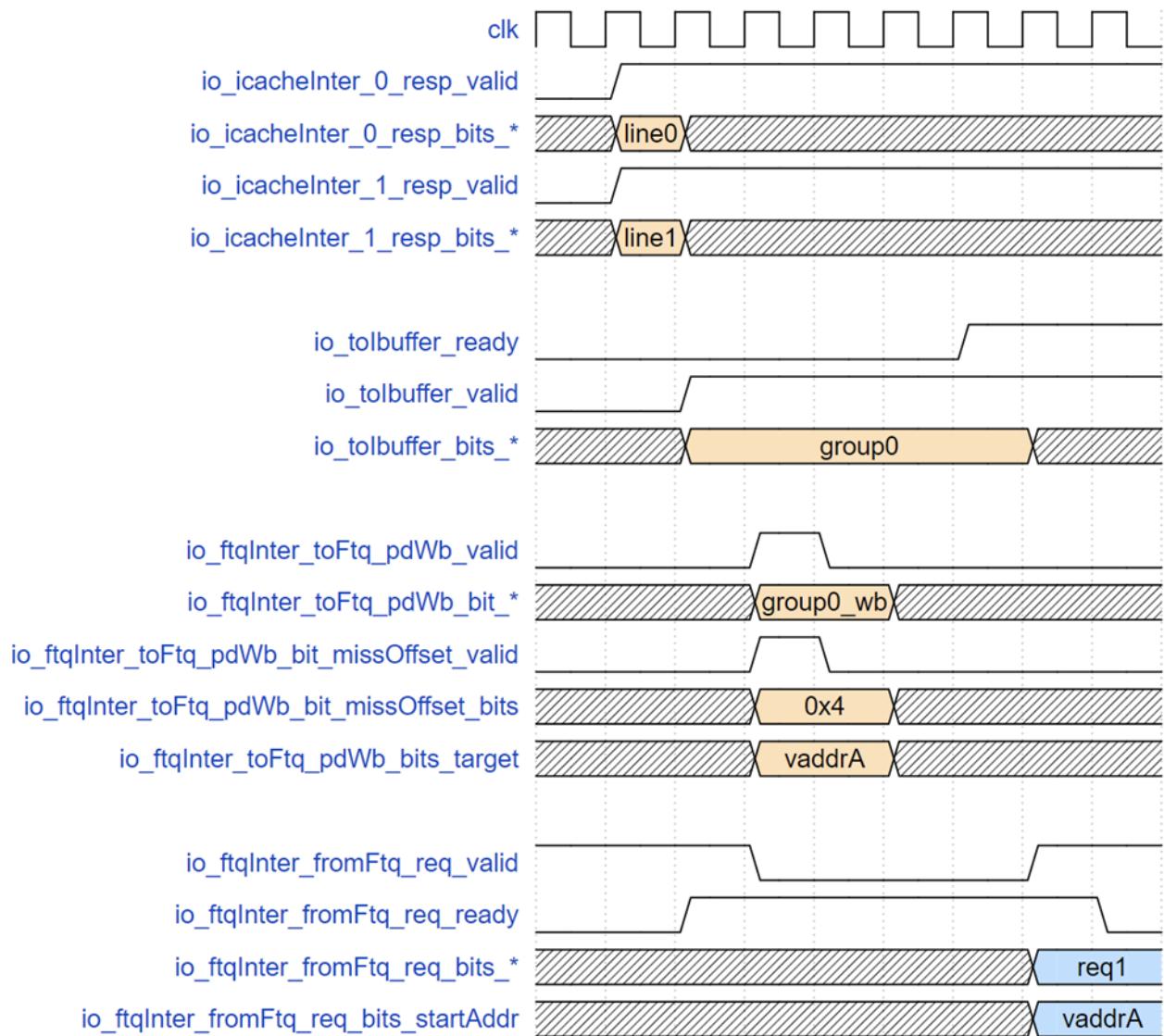


图 3.5: ICache 返回接口以及到 Ibuffer 和写回 FTQ 接口时序示例

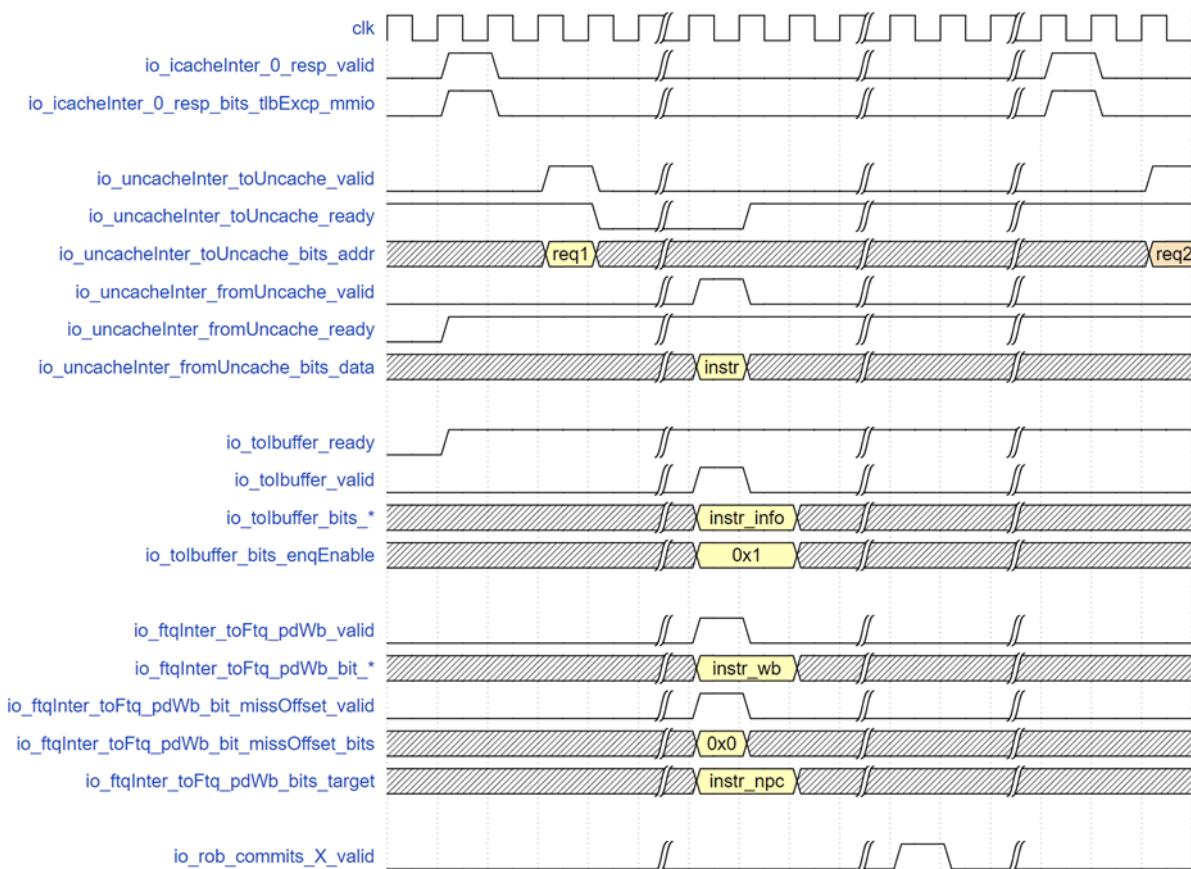


图 3.6: MMIO 请求接口时序示例

指令码，同拍 IFU 将这条指令作为一个预测块发送到 Ibuffer，同时发送对 FTQ 的写回，复用误预测信号端口，重定向地址为紧接着下一条指令的地址。此时 IFU 进入等待指令执行完成。一段时间后 rob_commits 端口报告此条指令执行完成，并且没有后端重定向。则 IFU 重新发起下一条 MMIO 指令的取指令请求。

3.2 IFU 子模块 PreDecoder

3.2.1 功能描述

3.2.1.1 功能概述

预译码器 PreDecoder 接受初始指令码并进行指令码生成，每个指令码查询预译码表产生预译码信息，预译码信息包括该位置是否是有效指令开始、CFI 指令类型、是否是 RVC 指令、是否是 Call 指令以及是否是 Ret 指令。预译码器会产生两种有效指令开始的向量，一种是默认第 1 个二字节必为有效指令开始，另一种是默认第 2 个二字节必为有效指令的开始，最终的选择在 IFU 端做。

3.2.1.2 分特性描述

3.2.1.2.1 特性 1：指令码生成 (instr_gen)

预译码器产生接受来自 IFU 完成指令切分的 17×2 字节的初始指令码，并以 4 字节为窗口，2 字节为步进长度，从第 1 个 2 字节开始，直到第 16 个 2 字节，选出总共 16 个 4 字节的指令码。

3.2.1.2.2 特性 2：有效指令开始向量生成 (vec_gen)

预译码器在生成初始指令码的同时，生成 16bit 有效指令开始向量，这个向量每 bit 标识此位置的指令是一条有效指令的开始。生成逻辑为：

- 正常模式：默认第一个 2 字节为第一条指令开始。如果第 $n-1$ 个 2 字节是有效指令开始且是 RVC 指令，或者第 $n-1$ 个 2 字节不是有效指令开始（肯定是一条 4 字节指令的结尾 2 字节），那么第 n 个 2 字节即为有效指令开始。
- 非正常模式：默认第一个 2 字节为一条 4 字节指令的后半段，第一条有效指令从第二个 2 字节开始，后续生成逻辑和正常模式一样。

两种模式并行生成，最终由 IFU 内部是否有跨缓存行的 RVI 指令进行结果的选择。

3.2.1.2.3 特性 3：预译码信息生成 (decoder)

预译码器根据指令码产生预译码信息，主要包括：是否是 RVC 指令、是否是 CFI 指令、CFI 指令类型 (branch/jal/jalr/call/ret)、CFI 指令的目标地址计算偏移。CFI 指令类型如表 1.2 所示。

3.2.2 整体框图

3.2.3 接口时序

由于 PreDecode 模块均为组合逻辑，因此输入和输出都在同一个时钟周期内

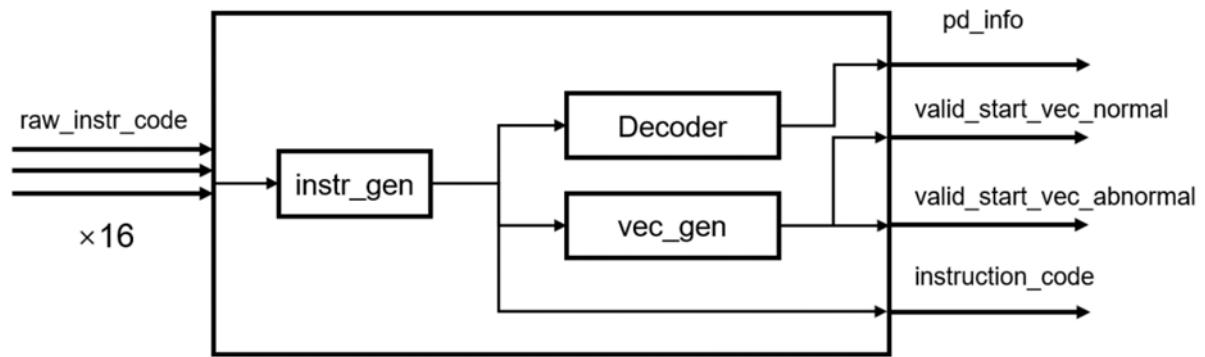


图 3.7: PreDecoder 结构

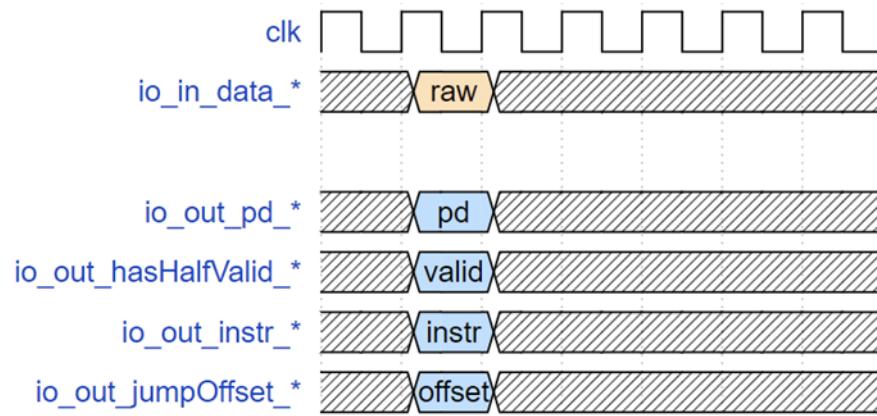


图 3.8: PreDecode 接口时序

3.3 IFU 子模块 PredChecker

3.3.1 功能描述

3.3.1.1 功能概述

分支预测检查器 PredChecker 接收来自 IFU 的预测块信息（包括预测跳转指令在预测块的位置、预测的跳转目标、预译码得到的指令信息、指令 PC 以及预译码得到的跳转目标偏移等），在模块内部检查五种类型的分支预测错误。模块内部分为两个流水线 stage，分别输出信息，第一个 stage 输出给 f3 阶段，用于修正预测块的指令范围和预测结果。第二个 stage 输出给 wb 阶段，用于在发现分支预测错误时产生前端重定向以及写回给 FTQ 正确的预测信息。

3.3.1.2 分特性描述

3.3.1.2.1 特性 1：Jal 指令预测错误检查

jal 指令预测错误的条件是，预测块中有一条 jal 指令（由预译码信息给出），但是要么这个预测块没有预测跳转，要么找个预测块预测跳转的指令在这条 jal 指令之后（即这条 jal 指令没有被预测跳转）。

3.3.1.2.2 特性 2：Ret 指令预测错误检查

ret 指令预测错误的条件是，预测块中有一条 ret 指令（由预译码信息给出），但是要么这个预测块没有预测跳转，要么找个预测块预测跳转的指令在这条 ret 指令之后（即这条 ret 指令没有被预测跳转）。

3.3.1.2.3 特性 6：重新生成指令有效范围向量

PredChecker 在检查出 Jal/Ret 指令预测错误时，需要重新生成指令有效范围向量，有效范围截取到 Jal/Ret 指令的位置，之后的 bit 全部置为 0。需要注意的是，jal 和 ret 指令的错误检查都会导致指令有效范围的缩短，所以需要重新生成指令有效范围 fixedRange，同时修复预测结果（即将原来的预测结果取消，把这个指令块的预测结果根据 jal 指令的位置重新生成）

3.3.1.2.4 特性 3：非 CFI 预测错误检查

非 CFI 预测错误的条件是被预测跳转的指令根据预译码信息显示不是一条 CFI 指令。

3.3.1.2.5 特性 4：无效指令预测错误检查

无效指令预测错误的条件是被预测的指令的位置根据预译码信息中的指令有效向量显示不是一条有效指令的开始。

3.3.1.2.6 特性 5：目标地址预测错误检查

目标地址预测错误的条件是，被预测的是一条有效的 jal 或者 branch 指令，同时预测的跳转目标地址和由指令码计算得到的跳转目标不一致。

3.3.1.2.7 特性 5：分级输出检查结果

以上 PredChecker 检查结果会分为两级分别输出，前面已经提到，Jal/Ret 指令由于需要重新生成指令有效范围向量和重新指定预测位置，所以需要在错误产生的当拍（F3）直接输出结果到 Ibuffer 用于及时更正进入后端的指令。而由于时序的考虑，其他错误信息（比如五种错误的错误位置、正确的跳转地址等）则是等到下一拍（WB）阶段才返回给 IFU 做前端重定向。

3.3.1.2.8 整体框图

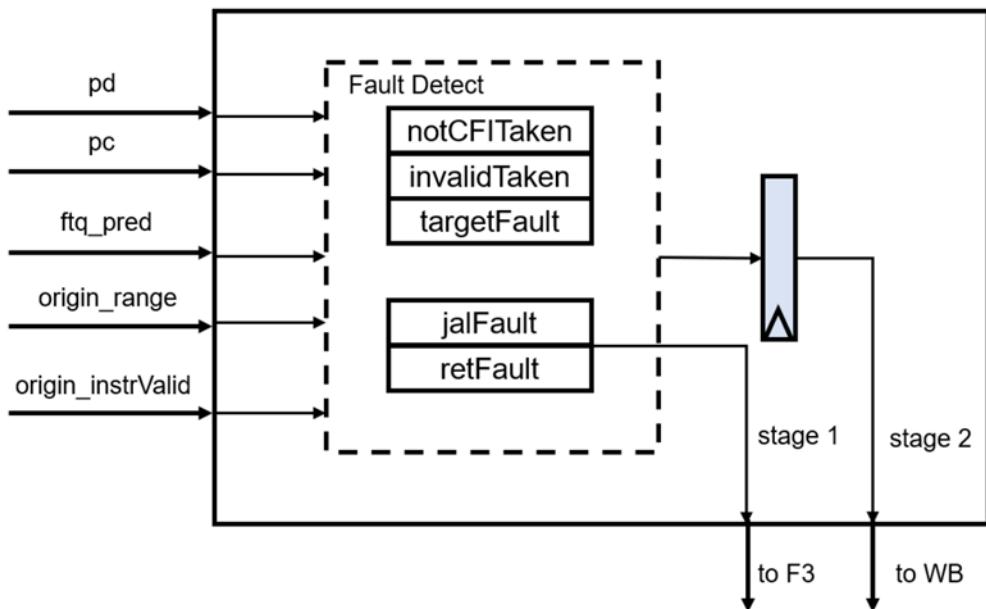


图 3.9: PredChecker 结构

3.3.1.2.9 接口时序

如图，黄色代表同一个预测块在 PredChecker 里的检查过程，这个预测块的第 6 个字节位置被预测为跳转，改预测块原本的有效指令范围为 h7f (即 0000000001111111)，指令有效向量为 hbfeb (即 101111111101011)，但是检查器发现该预测块第 1 个字节的位置（从 0 开始计数）是一条 jal 指令（brType 值为 b10），所以检查器首先在 stage1 将有效指令范围修改为 h3，预测字节位置为 1 给 F3 来进行 Ibuffer 指令入队选择，同时在 WB 阶段将目标地址修正为 h80002120，同时标记误预测位置为 6，通知 WB 阶段做重定向

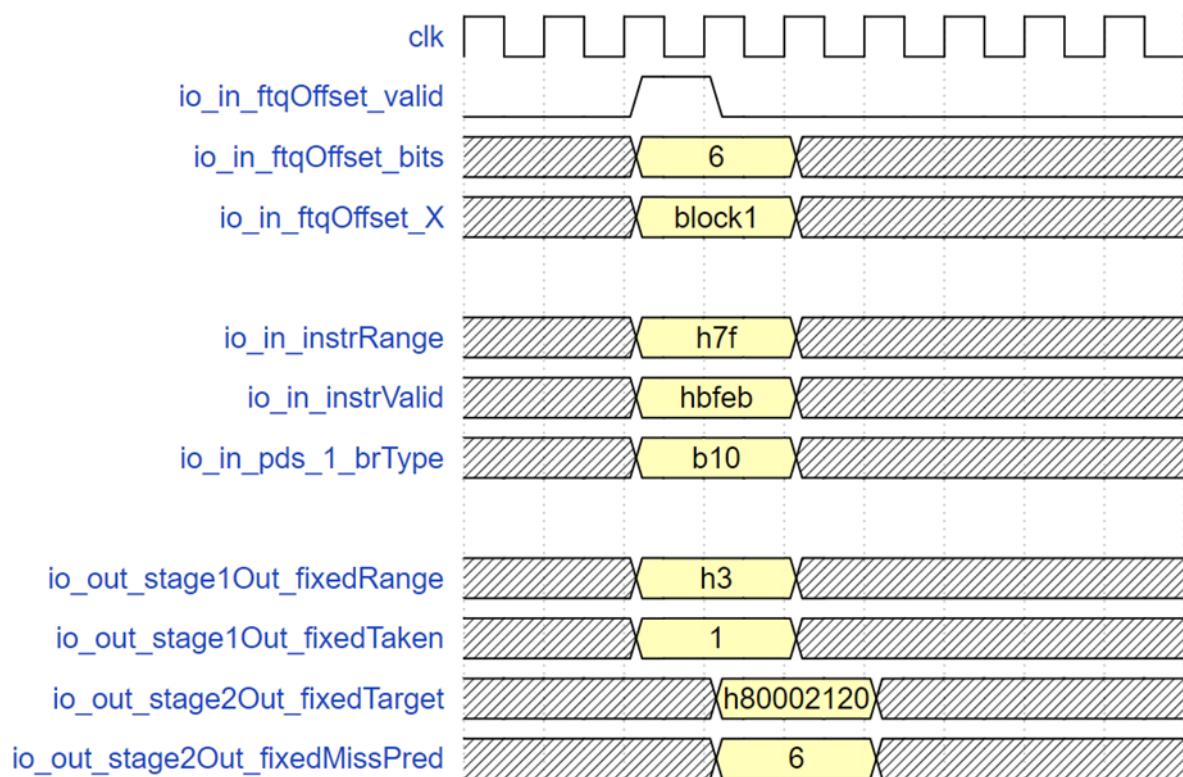


图 3.10: PredChecker 接口时序

4 指令缓存

4.1 XiangShan ICache 设计文档

- 版本: V2R2
- 状态: OK
- 日期: 2025/03/07
- commit: 4b2c87ba1d7965f6f2b0a396be707a6e2f6fb345

4.1.1 术语说明

| 缩写 | 全称 | 描述 |
|---------------|--|-----------------------------|
| ICache/I\$ | Instruction Cache | L1 指令缓存 |
| DCache/D\$ | Data Cache | L1 数据缓存 |
| L2 Cache/L2\$ | Level Two Cache | L2 缓存 |
| IFU | Instruction Fetch Unit | 取指单元 |
| ITLB | Instruction Translation Lookaside Buffer | 地址翻译缓冲 |
| PMP | Physical Memory Protection | 物理内存保护模块 |
| PMA | Physical Memory Attribute | 物理内存属性模块（是 PMP 的一部分） |
| BEU | Bus Error Unit | 总线错误单元 |
| FDIP | Fetch-directed Instruction Prefetch | 取指导向指令预取 |
| MSHR | Miss Status Holding Register | 缺失状态保持寄存器 |
| a/(g)pf | Access / (Guest) Page Fault | 访问错误 / （客户机）页错误 |
| v/(g)paddr | Virtual / (Guest) Physical Address | 虚拟地址 / （客户机）物理地址 |
| PBMT | Page-Based Memory Types | 基于页的内存类型，见特权手册 Svpbmt 扩展 |

4.1.2 子模块列表

| 子模块 | 描述 |
|---------------|-------------------------|
| MainPipe | 主流水线 |
| IPrefetchPipe | 预取流水线 |
| WayLookup | 元数据缓冲队列 |
| MetaArray | 元数据 SRAM |
| DataArray | 数据 SRAM |
| MissUnit | 缺失处理单元 |
| Replacer | 替换策略单元 |
| CtrlUnit | 控制单元，目前仅用于控制错误校验/错误注入功能 |

4.1.3 设计规格

- 缓存指令数据
- 缺失时通过 tilelink 总线向 L2 请求数据
- 软件维护 L1 I/D Cache 一致性 (`fence.i`)
- 支持跨 cacheline (预) 取指请求
- 支持冲刷 (bpu redirect、backend redirect、`fence.i`)
- 支持预取指请求
 - 硬件预取为 FDIP 预取算法
 - 软件预取为 Zicbop 扩展 `prefetch.i` 指令
- 支持可配置的替换算法
- 支持可配置的缺失状态寄存器数量
- 支持检查地址翻译错误、物理内存保护错误
- 支持错误检查 & 错误恢复 & 错误注入¹
 - 默认采用 parity code
 - 通过从 L2 重取实现错误恢复
 - 软件可通过 MMIO 空间访问的错误注入控制寄存器
- dataArray 支持分 bank 存储，细存储粒度实现低功耗

4.1.4 参数列表

| 参数 | 默认值 | 描述 | 要求 |
|---------------|-----|-------------|-------|
| nSets | 256 | SRAM set 数量 | 2 的幂次 |
| nWays | 4 | SRAM way 数量 | |
| nFetchMshr | 4 | 取指 MSHR 的数量 | |
| nPrefetchMshr | 10 | 预取 MSHR 的数量 | |

¹ 本文档也将错误检查 & 错误恢复 & 错误注入相关功能称为 ECC，见 § 4.1.6.5 ECC 一节开始的说明。

| 参数 | 默认值 | 描述 | 要求 |
|---------------------|-----|---------------------------------------|-----------------------------|
| nWayLookupSize | 32 | WayLookup 深度, 同时可以反压限制预取最大距离 | |
| DataCodeUnit | 64 | 校验单元大小, 单位为 bit, 每 64bit 对应 1bit 的校验位 | |
| ICacheDataBanks | 8 | cacheline 划分 bank 数量 | |
| ICacheDataSRAMWidth | 66 | DataArray 基本 SRAM 的宽度 | 大于每 bank 的 data 和 code 宽度之和 |

4.1.5 功能概述

FTQ 中存储着 BPU 生成的预测块, fetchPtr 指向取指预测块, prefetchPtr 指向预取预测块, 当复位时 prefetchPtr 与 fetchPtr 相同, 每成功发送一次取指请求时 fetchPtr++, 每成功发送一次预取请求时 prefetchPtr++。详细说明见 FTQ 设计文档。

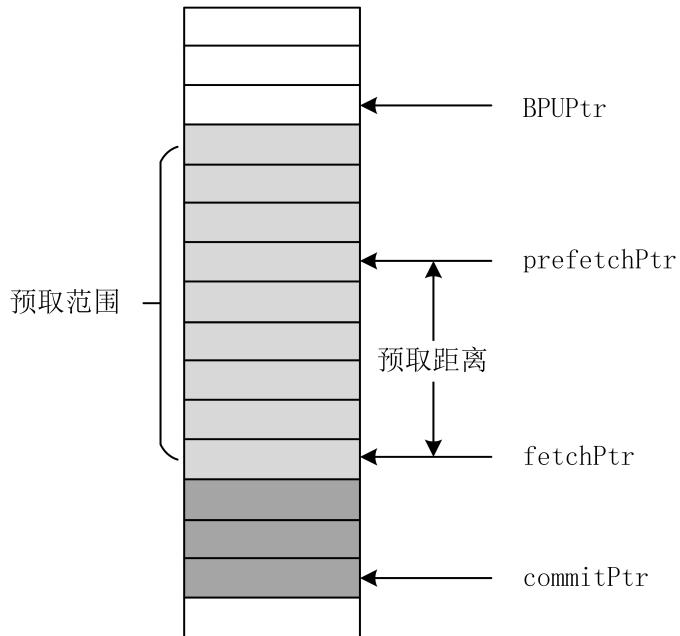


图 4.1: FTQ 指针示意

ICache 结构如下图所示。有 MainPipe 和 IPrefetchPipe 两个流水线, MainPipe 接收来自 FTQ 的取指请求, IPrefetchPipe 接收来自 FTQ/MemBlock 的硬/软件预取请求。对于预取请求, IPrefetch 对 MetaArray 进行查询, 将元数据 (在哪一路命中、ECC 校验码、是否发生异常等) 存储到 WayLookup 中, 如果该请求缺失, 就发送至 MissUnit 进行预取。对于取指请求, MainPipe 首先从 WayLookup 中读取命中信息, 如果 WayLookup 中没有可用信息, MainPipe 就会阻塞, 直至 IPrefetchPipe 将信息写入 WayLookup 中, 该方案将 MetaArray 和 dataArray 的访问分离, 一次只访问 dataArray 单路, 实现了较低的功耗, 代价是产生了一个周期的重定向延迟。

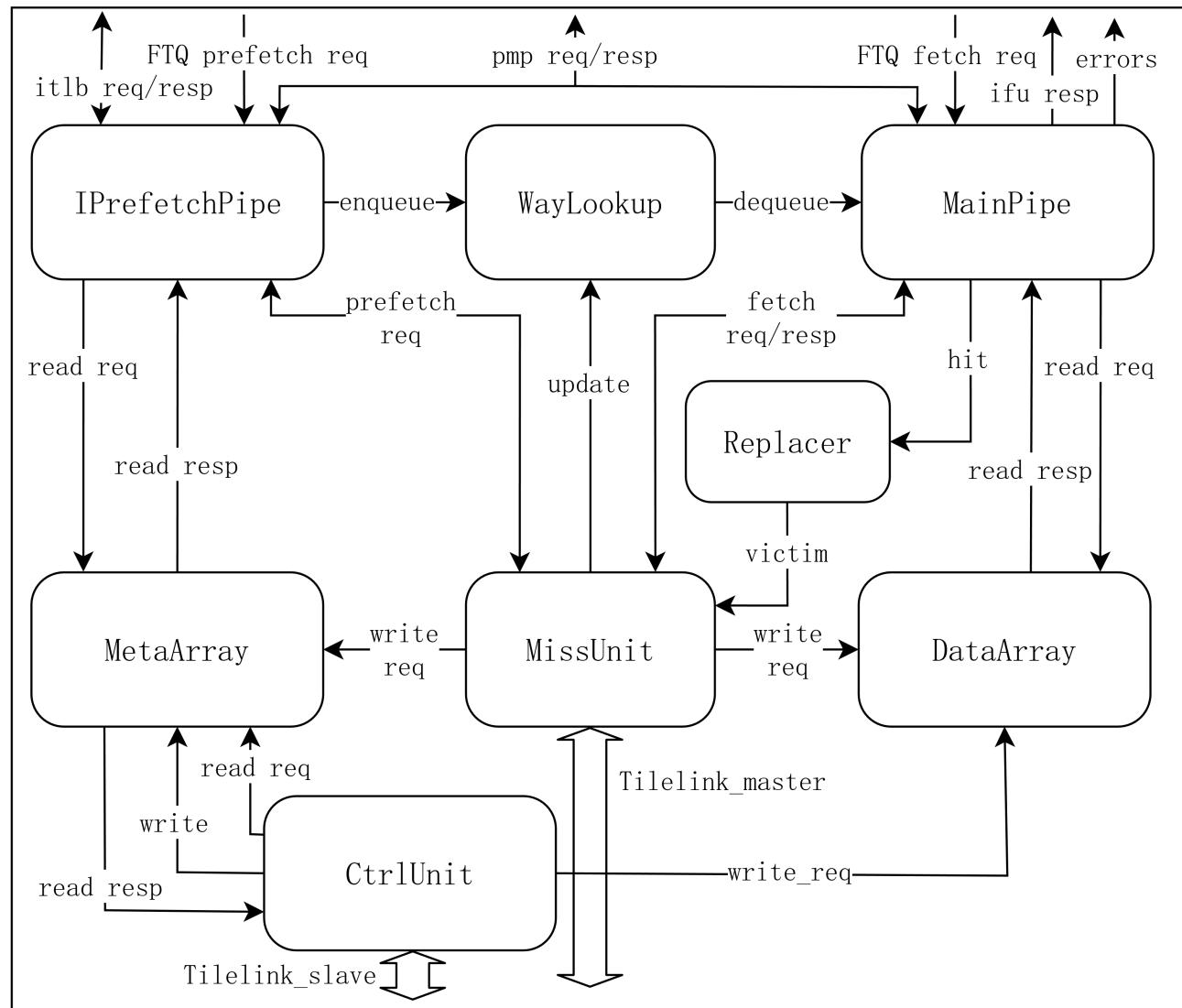


图 4.2: ICache 结构

MissUnit 处理来自 MainPipe 的取指请求和来自 IPrefetchPipe 的预取请求，通过 MSHR 进行管理，所有 MSHR 公用一组数据寄存器以减少面积。

Replacer 为替换器，默认采用 PLRU 替换策略，接收来自 MainPipe 的命中更新，向 MissUnit 提供待替换的 waymask。

MetaArray 分为奇偶两个 bank，用于支持跨 cacheline 的双行访问。

DataArray 中的 cacheline 默认分为 8 个 bank 存储，每个 bank 中存储的有效数据为 64bit，另外对于每 64bit 还需要 1bit 的校验位，由于 65bit 宽度的 SRAM 表现不好，所以选用 256*66bit 的 SRAM 作为基本单元，一共有 32 个这样的基本单元。一次访问需要 34Byte 的指令数据，每次需要访问 5 个 bank ($8 \times 5 > 34$)，根据起始地址进行选择。

4.1.6 功能详述

4.1.6.1 (预) 取指请求

FTQ 分别把 (预) 取指请求发送到 (预) 取指流水线进行处理。如前所述，由 IPrefetch 对 MetaArray 和 ITLB 进行查询，将元数据（在哪一路命中、ECC 校验码、是否发生异常等）在 IPrefetchPipe s1 流水级存储到 WayLookup 中，以供 MainPipe s0 流水级读取。

在上电解复位/重定向时，由于 WayLookup 为空，而 FTQ 的 prefetchPtr、fetchPtr 复位到同一位置，MainPipe s0 流水级不得不阻塞等待 IPrefetchPipe s1 流水级的写入，这引入了一拍的额外重定向延迟。但随着 BPU 向 FTQ 填充预测块的进行和 MainPipe/IFU 因各种原因阻塞 (e.g. miss、IBuffer 满)，IPrefetchPipe 将工作在 MainPipe 前 ($\text{prefetchPtr} > \text{fetchPtr}$)，而 WayLookup 中也会有足够的元数据，此时 MainPipe s0 级和 IPrefetchPipe s0 级的工作将是并行的。

详细的取指过程见 MainPipe 子模块文档、IPrefetchPipe 子模块文档和 WayLookup 子模块文档。

4.1.6.1.1 硬件预取与软件预取

V2R2 后，ICache 可能接受两个来源的预取请求：

1. 来自 Ftq 的硬件预取请求，基于 FDIP 算法。
2. 来自 Memblock 中 LoadUint 的软件预取请求，其本质是 Zicbop 扩展中的 prefetch.i 指令，请参考 RISC-V CMO 手册。

然而，PrefetchPipe 每周期仅可以处理一个预取请求，故需要进行仲裁。ICache 顶层负责缓存软件预取请求，并与来自 Ftq 的硬件预取请求二选一送往 PrefetchPipe，软件预取请求的优先级高于硬件预取请求。

逻辑上来说，每个 LoadUnit 都有可能发出软件预取请求，因此每周期至多会有 LoadUnit 数量（目前默认参数为 LduCnt=3）个软件预取请求。但出于实现成本和性能收益考量，ICache 每周期至多仅接收并处理一个，多余的会被丢弃，端口下标最小的优先。此外，若 PrefetchPipe 阻塞，而 ICache 内已经缓存了一个软件预取请求，那么原先的软件预取请求将被覆盖。

发送到 PrefetchPipe 后，软件预取请求的处理和硬件预取请求的处理几乎是一致的，除了：- 软件预取请求不会影响控制流，即不会发送到 MainPipe (和后续的 Ifu、IBuffer 等环节)，仅做：1) 判断是否 miss 或异常；2) 若 miss 且无异常，发送到 MissUnit 做预取指并重填 SRAM。

关于 PrefetchPipe 的细节请查看子模块文档。

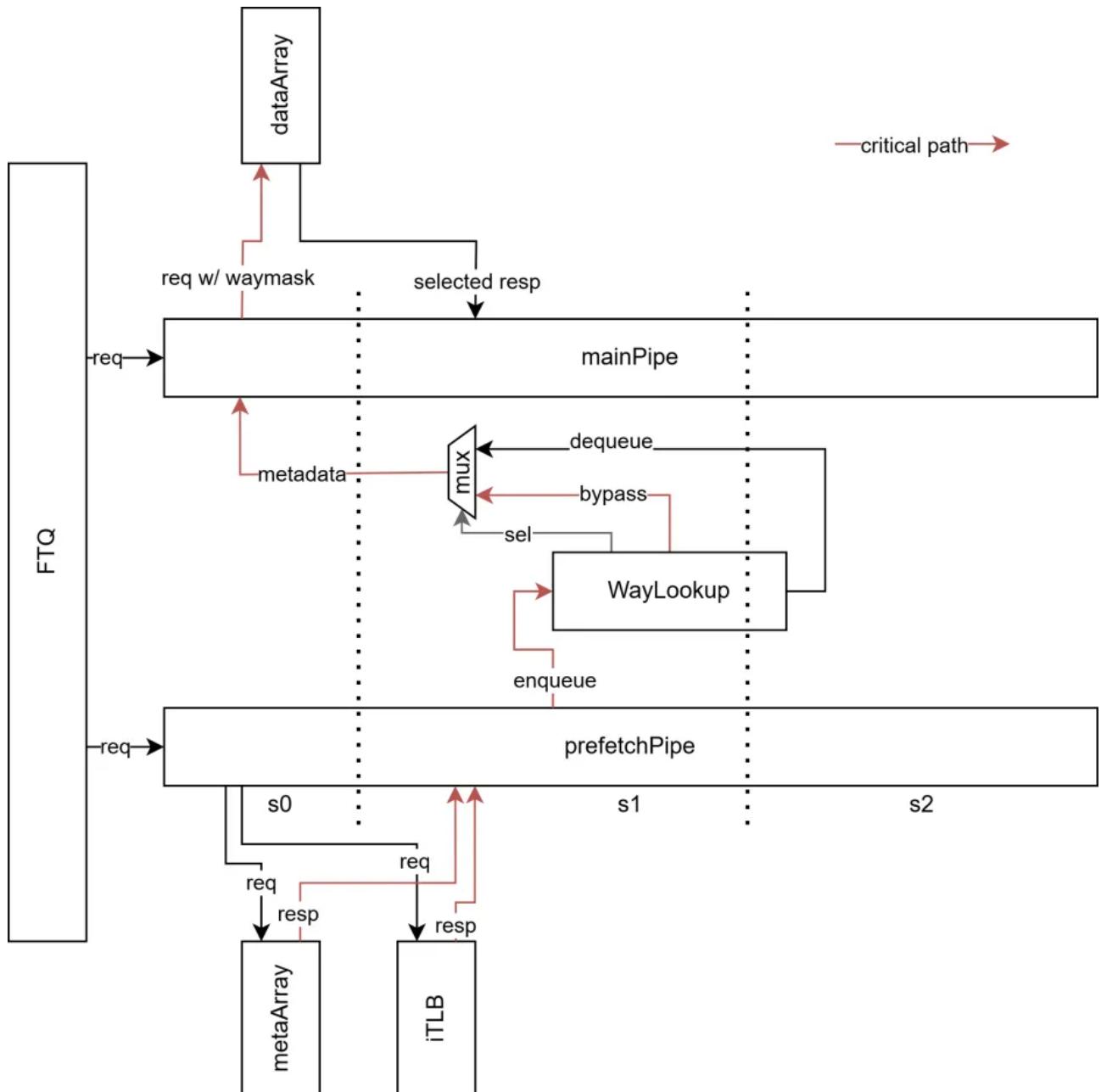


图 4.3: ICache 两条流水线的关系

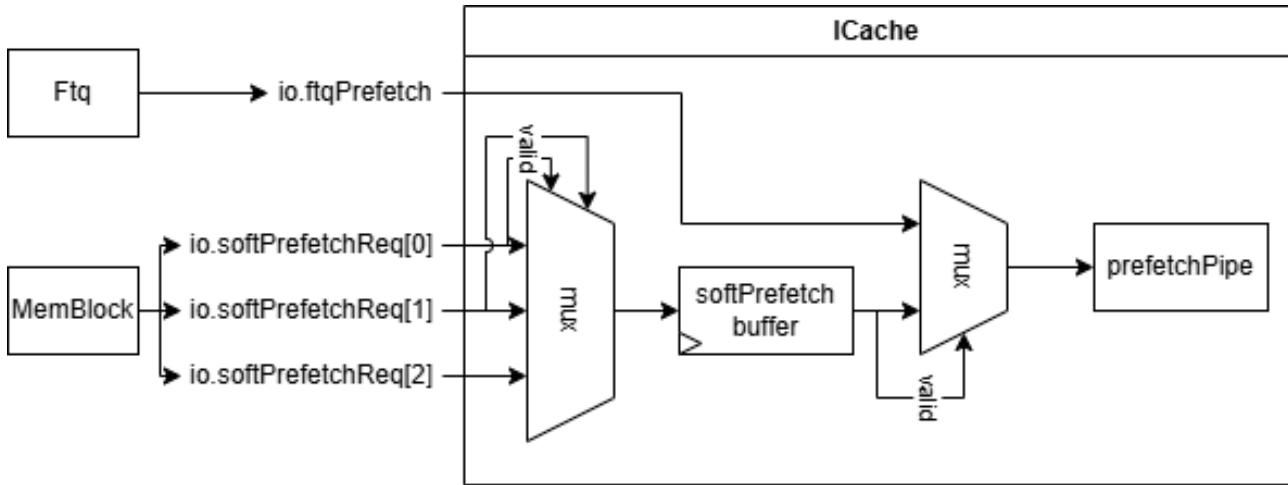


图 4.4: ICache 预取请求接收与仲裁

4.1.6.2 异常传递/特殊情况处理

ICache 负责对取指请求的地址进行权限检查（通过 ITLB 和 PMP），接收 L2 的响应，过程中可能出现的异常有：

| 来源 | 异常 | 描述 | 处理 |
|----------|------------|---------------------|--|
| ITLB | af | 虚拟地址翻译过程出现访问错误 | 禁止取指，标记取指块为 af，经 IFU、IBuffer 发送到后端处理 |
| ITLB | gpf | 客户机页错误 | 禁止取指，标记取指块为 gpf，经 IFU、IBuffer 发送到后端处理，将有效的 gpaddr 和 isForNonLeafPTE 发送到后端的 GPAMem 以备使用 |
| ITLB | pf | 页错误 | 禁止取指，标记取指块为 pf，经 IFU、IBuffer 发送到后端处理 |
| backend | af/pf/gpf | 同 ITLB af/gpf/pf | 同 ITLB af/gpf/pf |
| PMP | af | 物理地址无权限访问 | 同 ITLB af |
| MissUnit | L2 corrupt | L2 cache 响应 corrupt | 标记取指块为 af，经 IFU、IBuffer 发送到后端处理 |

需要指出，对于一般的取指流程来说，并不存在 backend 异常这一项。但 XiangShan 出于节省硬件资源的考虑，在前端传递的 pc 只有 41 / 50 bit (Sv39*4 / Sv48*4)，而对于 jr、jalr 等指令，跳转目标来源于 64 bit 寄存器。根据 RISC-V 规范，高位非全 0 或全 1 时的地址不合法，需要引发异常，这一检查只能由后端完成，并随同后端重定向信号一起发送到 Ftq，进而随同取指请求一起发送到 ICache。其本质是一种 ITLB 异常，因此解释描述和处理方式与 ITLB 相同。

另外，L2 cache 通过 tilelink 总线响应 corrupt 可能是 L2 ECC 错误 (`d.corrupt`)，亦可能是无权限访问总线地址空间导致拒绝访问 (`d.denied`)。tilelink 手册规定，当拉高 `d.denied` 时必须同时拉高 `d.corrupt`。而这两种情况都需要将取指块标记为 access fault，因此目前在 ICache 中无需区分这两种情况（即无需关注 `d.denied`，其可能被 chisel 自动优化掉而导致 verilog 中看不到）。

这些异常间存在优先级：backend 异常 > ITLB 异常 > PMP 异常 > MissUnit 异常。这是自然的：1. 当出现 backend 异常时，发送到前端的 vaddr 不完整且不合法，故 ITLB 地址翻译过程无意义，检查出的异常无效；2. 当出现 ITLB 异常时，翻译得到的 paddr 无效，故 PMP 检查过程无意义，检查出的异常无效；3. 当出现 PMP 异常时，paddr 无权限访问，不会发送（预）取指请求，故不会从 MissUnit 得到响应。

而对于 backend 的三种异常、ITLB 的三种异常，由 backend 和 ITLB 内部进行有优先级的选择，保证同时至多只有一种拉高。

此外，一些机制还会引发一些特殊情况，在旧版文档/代码中也称为异常，但其实际上并不引发 RISC-V 手册定义的 exception，为了避免混淆，此后将称为特殊情况：

| 来源 | 特殊情况 | 描述 | 处理 |
|----------|-----------|------------------------------------|-----------------------------------|
| PMP | mmio | 物理地址为 mmio 空间 | 禁止取指，标记取指块为 mmio，由 IFU 进行非推测性取指 |
| ITLB | pbmt.NC | 页属性为不可缓存、幂等 | 禁止取指，由 IFU 进行推测性取指 |
| ITLB | pbmt.IO | 页属性为不可缓存、非幂等 | 同 pmp mmio |
| MainPipe | ECC error | 主流水检查发现 MetaArray/DataArray ECC 错误 | 见ECC一节，旧版同 ITLB af，新版做自动重取 ECC 错误 |

4.1.6.3 dataArray 分 bank 的低功耗设计

目前，ICache 中每个 cacheline 分为 8 个 bank，bank0-7。一个取指块需要 34B 指令数据，故一次访问连续的 5 个 bank。存在两种情况：

1. 这 5 个 bank 位于单个 cacheline 中（起始地址位于 bank0-3）。假设起始地址位于 bank2，则所需数据位于 bank2-6。如下图 a。
2. 跨 cacheline（起始地址位于 bank4-7）。假设起始地址位于 bank6，则数据位于 cacheline0 的 bank6-7、cacheline1 的 bank0-2。有些类似于环形缓冲区。如下图 b。

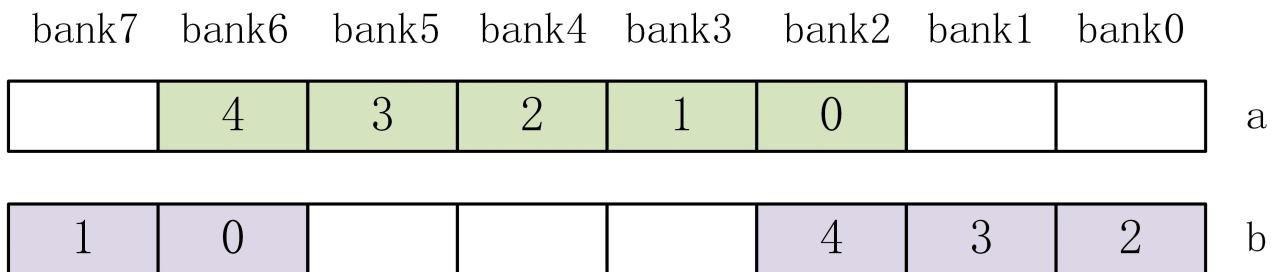


图 4.5: dataArray 分 bank 示意图

当从 SRAM 或 MSHR 中获取 cacheline 时，根据地址将数据放入对应的 bank。

由于每次访问只需要 5 个 bank 的数据，因此 ICache 到 IFU 的端口实际上只需要一个 64B 的端口，将两个 cacheline 各自的 bank 选择出来并拼接在一起返回给 IFU (在 dataArray 模块内完成)；IFU 将这一个 64B 的数据复制一份拼接在一起，即可直接根据取指块起始地址选择出取指块的数据。不跨行/跨行两种情况的示意图如下：

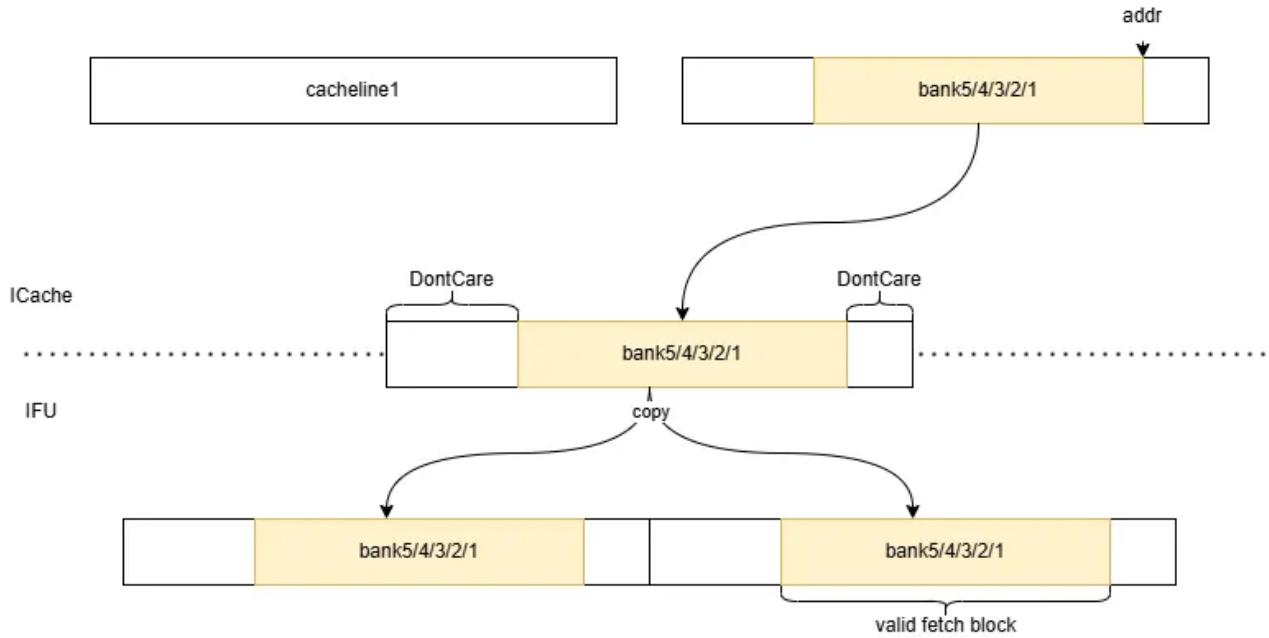


图 4.6: dataArray 数据返回示意图

亦可参考 IFU.scala 中的注释。

4.1.6.4 冲刷

在后端/IFU 重定向、BPU 重定向、`fence.i` 指令执行时，需要视情况对 ICache 内的存储结构和流水级进行冲刷。可能的冲刷目标/动作有：

1. MainPipe、IPrefetchPipe 所有流水级
 - 冲刷时直接将 `s0/1/2_valid` 置为 `false.B` 即可
2. MetaArray 中的 valid
 - 冲刷时直接将 `valid` 置为 `false.B` 即可
 - `tag`、`code` 不需要冲刷，因为它们的有效性由 `valid` 控制
 - dataArray 中的数据不需要冲刷，因为它们的有效性由 MetaArray 中的 `valid` 控制
3. WayLookup
 - 读写指针复位
 - `gpf_entry.valid` 置为 `false.B`
4. MissUnit 中所有 MSHR
 - 若 MSHR 尚未向总线发出请求，直接置无效 (`valid === false.B`) - 若 MSHR 已经向总线发出请求，记录待冲刷 (`flush === true.B` 或 `fencei === true.B`)，等到 d 通道收到 grant 响应时

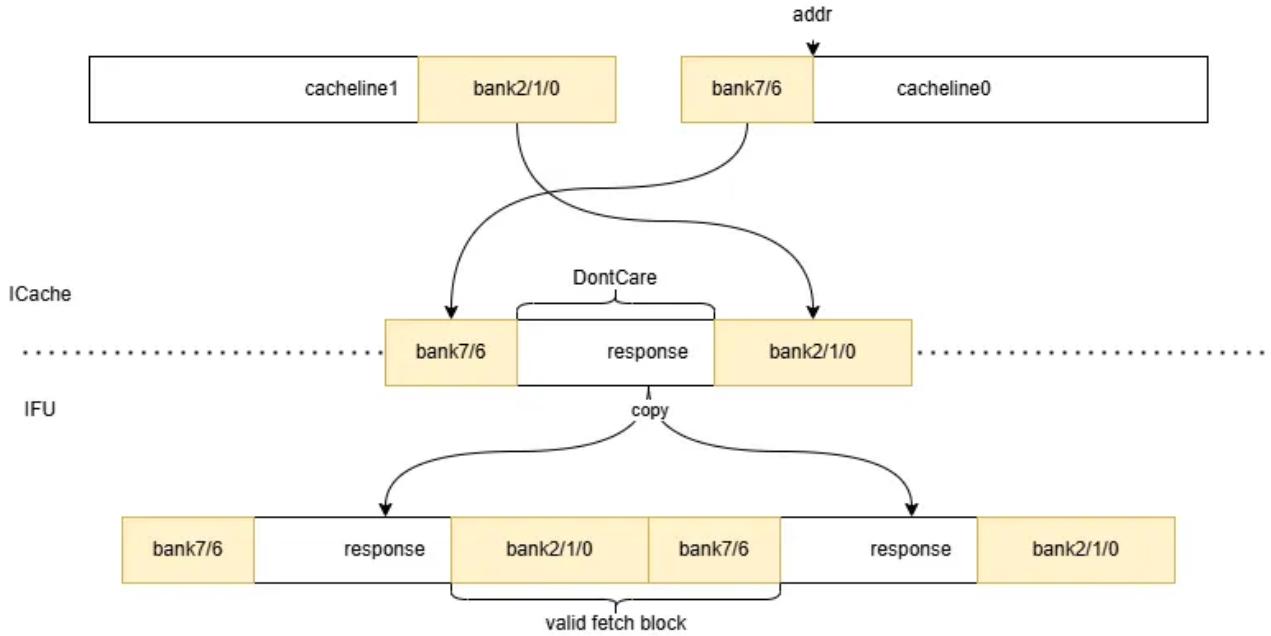


图 4.7: DataArray 数据返回示意图

再置无效，同时不把 grant 的数据回复给 MainPipe/PrefetchPipe，也不写入 SRAM - 需要留意，当 d 通道收到 grant 响应的同时收到冲刷 (io.flush === true.B 或 io.fencei === true.B) 时，MissUnit 同样不写入 SRAM，但会将数据回复给 MainPipe/PrefetchPipe，避免将端口的延时引入响应逻辑中，此时 MainPipe/PrefetchPipe 也同步收到了冲刷请求，因此会将数据丢弃

每种冲刷原因需要执行的冲刷目标：

| 冲刷原因 | 1 | 2 | 3 | 4 |
|------------|----------------|---|----------------|---|
| 后端/IFU 重定向 | Y | | Y | Y |
| BPU 重定向 | Y ² | | | |
| fence.i | Y ³ | Y | Y ⁴ | Y |

ICache 进行冲刷时不接收取指/预取请求 (io.req.ready === false.B)

4.1.6.4.1 对 ITLB 的冲刷

ITLB 的冲刷比较特殊，其缓存的页表项仅需要在执行 `sfence.vma` 指令时冲刷，而这条冲刷通路由后端负责，因此前端/ICache 一般不需要管理 ITLB 的冲刷。只有一个特例：目前 ITLB 为了节省资源，不会存储 `gpaddr`，而是在 `gpf` 发生时去 L2TLB 重取，重取状态由一个 `gpf` 缓存控制，这要求 ICache 在收到 `ITLB.resp.excp.gpf_instr` 时保证下面两个条件之一：

²BPU 精确预测器 (BPU s2/s3 给出结果) 可能覆盖简单预测器 (BPU s0 给出结果) 的预测，显然其重定向请求最晚在预取请求的 1-2 拍之后就到达 ICache，因此仅需要：

BPU s2 redirect: 冲刷 IPrefetchPipe s0

BPU s3 redirect: 冲刷 IPrefetchPipe s0/1

当 IPrefetchPipe 的对应流水级中的请求来自于软件预取时 `isSoftPrefetch === true.B`，不需要进行冲刷

当 IPrefetchPipe 的对应流水级中的请求来自于硬件预取，但 `ftqIdx` 与冲刷请求不匹配时，不需要进行冲刷

³fence.i 在逻辑上需要冲刷 MainPipe 和 IPrefetchPipe (因为此时流水级中的数据可能无效)，但实际上 `io.fencei` 拉高必然伴随一个后端重定向，因此目前的实现中没有冲刷 MainPipe 和 IPrefetchPipe 的必要。

⁴fence.i 在逻辑上需要冲刷 MainPipe 和 IPrefetchPipe (因为此时流水级中的数据可能无效)，但实际上 `io.fencei` 拉高必然伴随一个后端重定向，因此目前的实现中没有冲刷 MainPipe 和 IPrefetchPipe 的必要。

1. 重发相同的 ITLB.req.vaddr，直到 ITLB.resp.miss 拉低（此时 gpf、gpaddr 均有效，正常发往后端处理即可），ITLB 此时会冲刷 gpf 缓存。
2. 给 ITLB.flushPipe，ITLB 在收到该信号时会冲刷 gpf 缓存。

若 ITLB 的 gpf 缓存未被冲刷，就收到了不同 ITLB.req.vaddr 的请求，且再次发生 gpf，将导致核卡死。

因此，每当冲刷 IPrefetchPipe 的 s1 流水级时，无论冲刷原因为何，都需要同步冲刷 ITLB 的 gpf 缓存（即拉高 ITLB.flushPipe）。

4.1.6.5 ECC

首先需要指出，ICache 在默认参数下使用 parity code，其仅具备 1 bit 错误检测能力，不具备错误恢复能力，严格意义上不能算是 ECC (Error Correction Code)。但一方面，其可以配置为使用 secded code；另一方面，我们在代码中大量使用 ECC 来命名错误检测与错误恢复相关的功能 (`ecc_error`、`ecc_inject` 等)。因此本文档仍将使用 ECC 一词来指代错误检测、错误恢复、错误注入相关功能以保证与代码的一致性。

ICache 支持错误检测、错误恢复、错误注入功能，是 RAS⁵ 能力的一部分，可以参考 RISC-V RERI⁶ 手册，由 CtrlUnit 进行控制。

4.1.6.5.1 错误检测

在 MissUnit 向 MetaArray 和 dataArray 重填数据时，会计算 meta 和 data 的校验码，前者和 meta 一起存储在 Meta SRAM 中，后者存储在单独的 Data Code SRAM 中。

当取指请求读取 SRAM 时，会同步读取出校验码，在 MainPipe 的 s1/s2 流水级中分别对 meta/data 进行校验。软件可以通过向 CSR 中相应位置写入特定的值来使能/关闭这一功能，在 6-12 月的版本中为自定义 CSR `sfetchctl`，后续换成 mmio-mapped CSR，详见 CtrlUnit 文档。

在校验码设计方面，ICache 使用的校验码可由参数控制，默认使用的是 parity code，即校验码为对数据做规约异或 $code = \oplus data$ 。检查时只需将校验码和数据一起做规约异或 $error = (\oplus data) \oplus code$ ，结果为 1 则发生错误，反之认为没有错误（可能出现偶数个错误，但此处检查不出来）。

在 #4044 以后的版本中，ICache 支持错误注入，这要求 ICache 支持向 MetaArray/DataArray 写入错误的校验码。因此实现了一个 `poison` 位，当其拉高时，翻转写入的 code，即 $code = (\oplus data) \oplus poison$ 。

为了减少检查不出的情况，目前将 data 划分成 DataCodeUnit（默认为 64bit）的单元分别进行奇偶校验，因此对每个 64B 的缓存行，总计会计算 $8(data) + 1(meta) = 9$ 个校验码。

当 MainPipe 的 s1/s2 流水级检查到错误时，会进行以下处理：

在 6 月至 11 月的版本中：

1. 错误处理：引起 access fault 异常，由软件处理。
2. 错误报告：向 BEU 报告错误，后者会引起中断向软件报告错误。
3. 取消请求：当 MetaArray 被检查出错误时，其读出的 ptag 不可靠，进而对 hit 与否的判断不可靠，因此无论是否 hit 都不向 L2 Cache 发送请求，而是直接将异常传递到 IFU、进而传递到后端处理。

在后续版本（#3899 后）实现了错误自动恢复机制，故只需进行以下处理：

1. 错误处理：从 L2 Cache 重新取指，见下节。
2. 错误报告：同上向 BEU 报告错误。

⁵此 RAS (Reliability, Availability, and Serviceability) 非彼 RAS (Return Address Stack)。

⁶RERI (RAS Error-record Register Interface)，参考 RISC-V RERI 手册。

4.1.6.5.2 错误自动恢复

注意到，ICache 与 DCache 不同，是只读的，因此其数据必然不是 dirty 的，这意味着我们总是可以从下级存储结构（L2/3 Cache、memory）中重新获取正确的数据。因此，ICache 可以通过向 L2 Cache 重新发起 miss 请求来实现错误自动恢复。

实现重取功能本身只需要复用现有的 miss 取指路径，走 MainPipe -> MissUnit -> MSHR -tilelink-> L2 Cache 的请求路径。MissUnit 向 SRAM 重填数据时会自然地计算新的校验码并存储，因此在重取后会回到无错误的状态而不需要额外的处理。

6-11 月和后续代码行为差异的伪代码示意如下：

```
- exception = itlb_exception || pmp_exception || ecc_error
+ exception = itlb_exception || pmp_exception

- should_fetch = !hit && !exception
+ should_fetch = (!hit || ecc_error) && !exception
```

需要留意的是：为了避免重取后出现 multi-hit（即，同一个 set 内存在多个 way 的 ptag 相同），需要在重取前将 metaArray 对应位置的 valid 清空：

- 若 MetaArray 错误：meta 保存的 ptag 本身可能出错，命中结果（one-hot 的 waymask）不可靠，“对应位置”指该 set 的所有 way
- 若 dataArray 错误：命中结果可靠，“对应位置”指该 set 中 waymask 拉高的那一 way

4.1.6.5.3 错误注入

根据 RERI 手册⁷的说明，为了使软件能够测试 ECC 功能，进而更好地判断硬件功能是否正常，需要提供错误注入功能，即主动地触发 ECC 错误。

ICache 的错误注入功能由 CtrlUnit 控制，通过向 mmio-mapped CSR 中相应位置写入特定的值来触发。详见 CtrlUnit 文档。

目前 ICache 支持：

- 向特定 paddr 注入，当请求注入的 paddr 未命中时，注入失败
- 向 MetaArray 或 dataArray 注入
- 当 ECC 校验功能本身未使能时，注入失败

软件注入流程示意如下：

```
inject_target:
# maybe do something
ret

test:
la t0, $BASE_ADDR      # 载入 mmio-mapped CSR 基地址
la t1, inject_target  # 载入注入目标地址
jalr ra, 0(t1)         # 跳转到注入目标以保证其加载到 ICache
```

⁷RERI (RAS Error-record Register Interface)，参考 RISC-V RERI 手册。

```

sd t1, 8(t0)          # 向 CSR 写入注入目标地址
la t2, ($TARGET << 2 | 1 << 1 | 1 << 0) # 设置注入目标、注入使能、校验使能
sd t1, 0(t0)          # 向 CSR 写入注入请求

loop:
    ld t1, 0(t0)        # 读取 CSR
    andi t1, t1, (0b11 << (4+1)) # 读取注入状态
    beqz t1, loop        # 如果注入未完成，继续等待

    addi t1, t1, -1
    bnez t1, error       # 如果注入失败，跳转到错误处理

    jalr ra, 0(t1)        # 注入成功，跳转到注入目标地址以触发错误
    j    finish            # 结束

error:
    # handle error

finish:
    # finish

```

我们编写了一个测试用例，见此仓库，其测试了如下情况：

1. 正常注入 MetaArray
2. 正常注入 dataArray
3. 注入无效的目标
4. 注入但 ECC 校验未使能
5. 注入未命中的地址
6. 尝试写入只读的 CSR 域

4.1.7 参考文献

1. Glenn Reinman, Brad Calder, and Todd Austin. “Fetch directed instruction prefetching.” 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO). 1999.

4.2 MainPipe 子模块文档

MainPipe 为 ICache 的主流水，为 2 级流水设计，负责从 dataArray 中读取数据、PMP 检查、ECC 检查、缺失处理，并且将结果返回给 IFU。

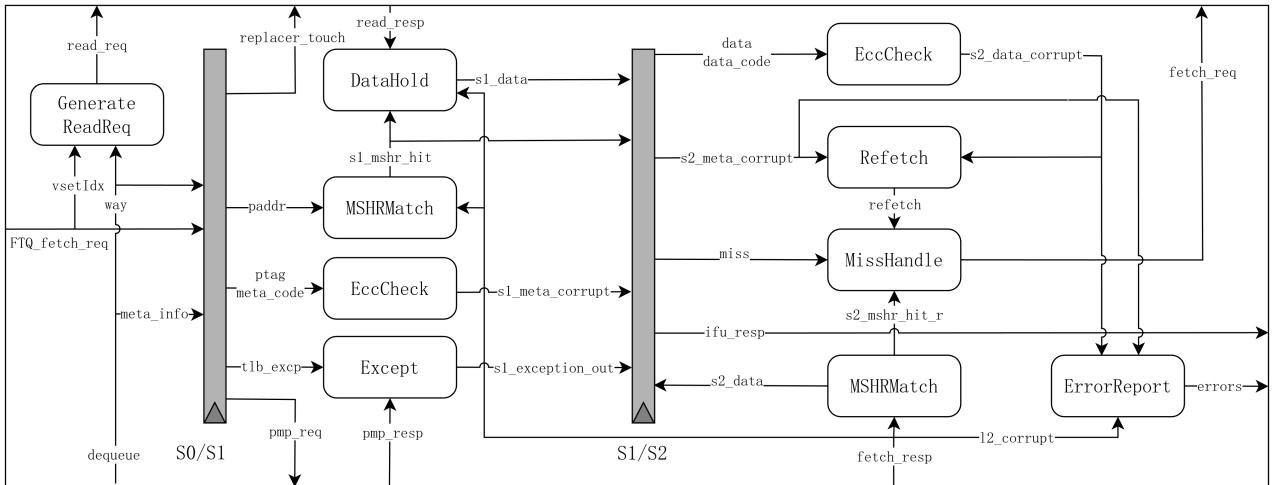


图 4.8: MainPipe 结构

4.2.1 S0 流水级

在 S0 流水级，从 WayLookup 获取元数据，包括路命中信息和 ITLB 查询结果，访问 dataArray 的单路，如果 dataArray 正在被写或 WayLookup 中没有有效表项，流水线就会阻塞。每次重定向后，FTQ 中同一个请求被同时发送到 MainPipe 和 IPrefetchPipe 中，MainPipe 始终需要等待 IPrefetchPipe 将请求的查询信息写入 WayLookup 后才能向下走，导致了 1 拍重定向延迟，当预取超过取指时，该延迟就会被覆盖。

4.2.2 S1 流水级

1. 更新 replacer，向 replacer 发送 touch 请求。
2. PMP 检查，发送 PMP 请求，在当拍收到响应，将结果寄存到下一流水级进行处理。
 - 需要指出，IPrefetchPipe s1 流水级也会进行 PMP 检查，和此处的检查实际上是完全一样的，分别检查只是为了优化时序（避免 ITLB(reg) → ITLB.resp → PMP.req → PMP.resp → WayLookup.write → bypass → WayLookup.read → MainPipe s1(reg) 的超长组合逻辑路径）
3. 接收 dataArray 返回的 data 和 code 并寄存，同时监听 MSHR 的响应，当 dataArray 和 MSHR 的响应同时有效时，后者的优先级更高。

4.2.3 S2 流水级

1. dataArray ECC 校验，对 S1 流水级寄存的 code 进行校验。如果校验出错，就将错误报告给 BEU。
2. MetaArray ECC 校验，IPrefetchPipe 读出 MetaArray 的数据后会直接进行校验，并将校验结果随命中信息一起入队 WayLookup 并随 MainPipe 流水到达 S2 级，在此处随 dataArray 的 ECC 校验结果一起报告给 BEU。
3. Tilelink 错误处理，当监听到 MissUnit 响应的数据 corrupt 为高时（即 L2 cache 的响应数据出错），就将错误报告给 BEU。
4. 缺失处理，缺失时将请求发送至 MissUnit，同时对 MSHR 的响应进行监听，命中时寄存 MSHR 响应的数据，为了时序在下一拍才将数据发送到 IFU。

4.3 IPrefetchPipe 子模块文档

IPrefetchPipe 为预取的流水线，为两级流水设计，负责预取请求的过滤。

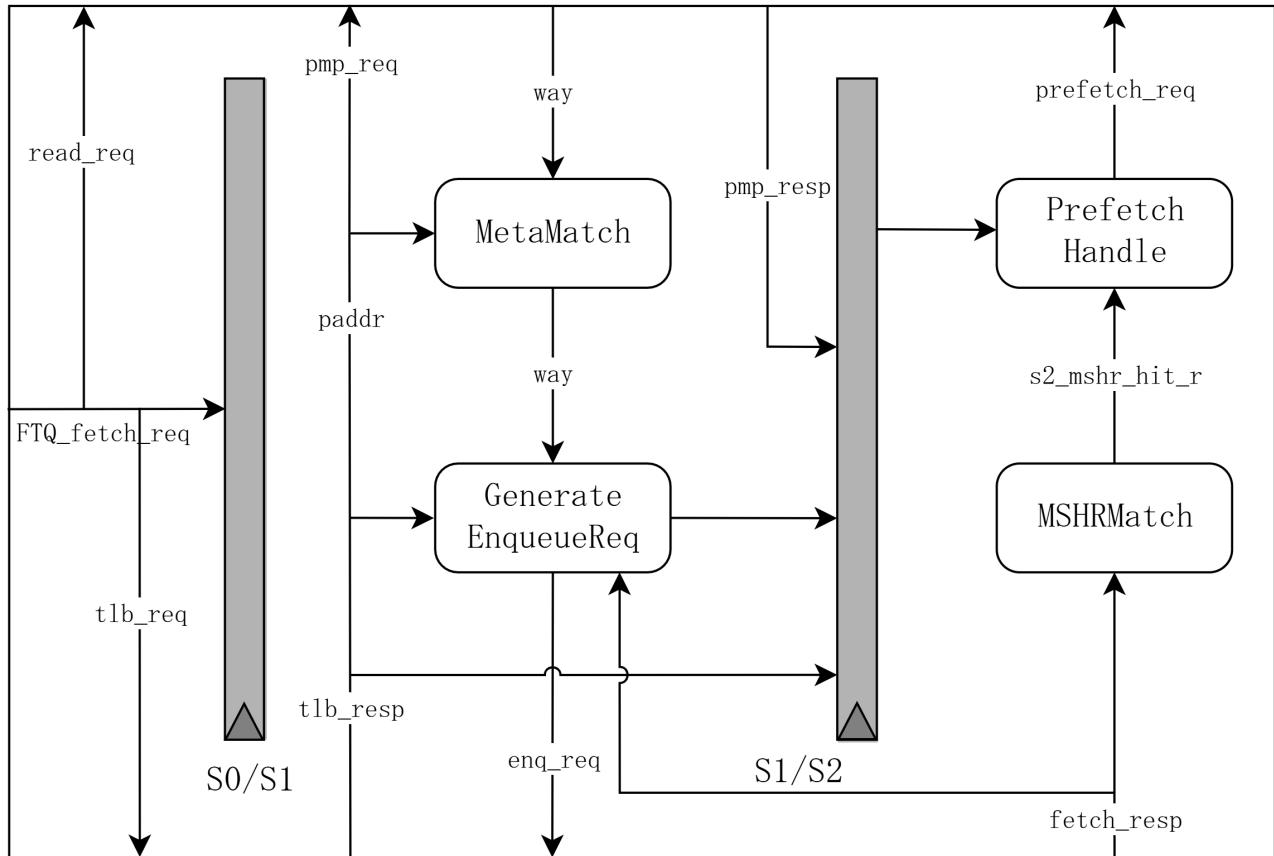


图 4.9: IPrefetchPipe 结构

4.3.1 S0 流水级

在 S0 流水级，接收来自 FTQ/后端的预取请求，向 MetaArray 和 ITLB 发送读请求。

4.3.2 S1 流水级

首先接收 ITLB 的响应得到 `paddr`，然后与 MetaArray 返回的 tag 进行比较得到命中信息，将元数据（命中信息 `waymask`、ITLB 信息 `paddr/af/pf`）写入 WayLookup。同时进行 PMP 检查，将结果寄存到下一级流水。由状态机进行控制：

- 初始状态为 `idle`，当 S1 流水级进入新的请求时，首先判断 ITLB 是否缺失，如果缺失，就进入 `itlbResend`；如果 ITLB 命中但命中信息未入队 WayLookup，就进入 `enqWay`；如果 ITLB 命中且 WayLookup 入队但 S2 请求未处理完，就进入 `enterS2`
- 在 `itlbResend` 状态，重新向 ITLB 发送读请求，此时占用 ITLB 端口（即新的进入 S0 流水级的预取请求被阻塞），直至请求回填完成，在回填完成的当拍向 MetaArray 再次发送读请求，回填期间可能发生新的写入，如果 MetaArray 繁忙（正在被 MSHR 写入），就进入 `metaResend`，否则进入 `enqWay`
- 在 `metaResend` 状态，重新向 MetaArray 发送读请求，发送成功后进入 `enqWay`

- 在 `enqWay` 状态，尝试将元数据入队 WayLookup，如果 WayLookup 队列已满，就阻塞至 WayLookup 入队成功，另外在 MSHR 发生新的写入时禁止入队，主要是为了防止写入的信息与命中信息所冲突，需要对命中信息进行更新。当成功入队 WayLookup 时，如果 S2 空闲，就直接回到 `idle`，否则进入 `enterS2`
 - 若当前请求是软件预取，不会尝试入队 WayLookup，因为该请求不需要进入 MainPipe/IFU 乃至被执行
- 在 `enterS2` 状态，尝试将请求流入下一流水级，流入后回到 `idle`

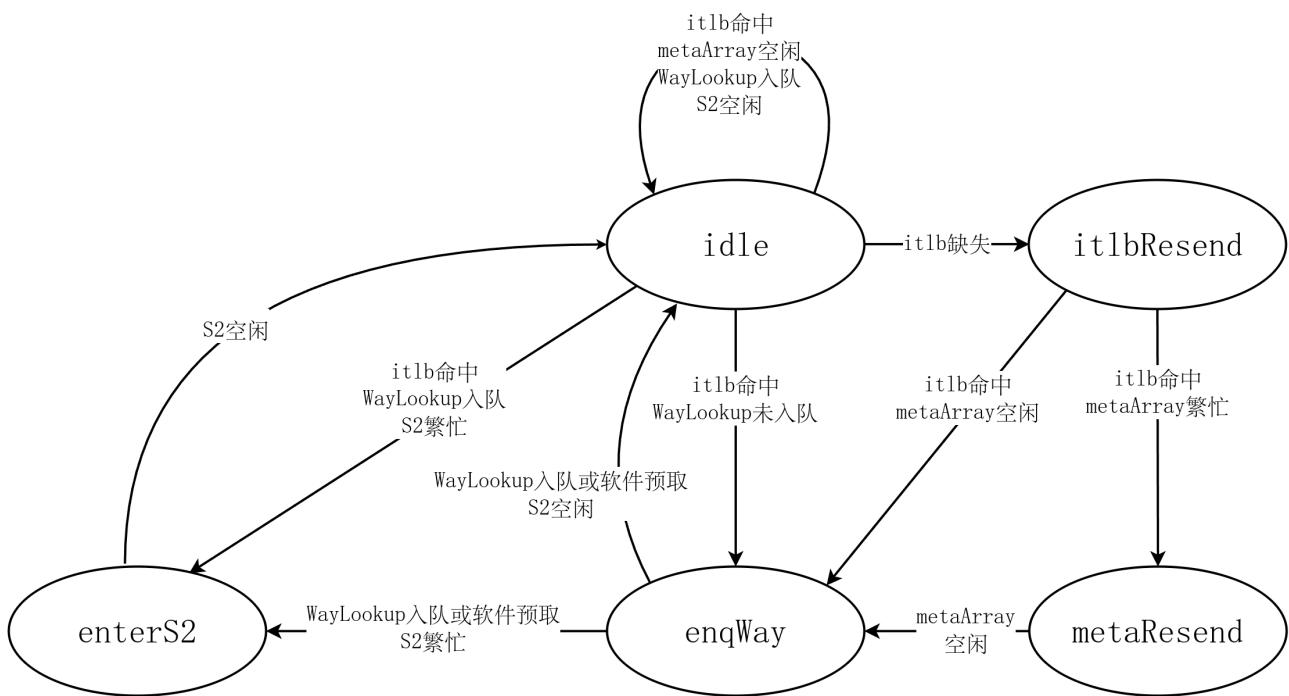


图 4.10: IPrefetchPipe S1 状态机

4.3.3 S2 流水级

综合该请求的命中结果、ITLB 异常、PMP 异常，判断是否需要预取，只有不存在异常时才进行预取，因为同一个预测块可能对应两个 cacheline，所以通过 Arbiter 依次将请求发送至 MissUnit。

4.3.4 命中信息的更新

在 S1 流水级中得到命中信息后，距离命中信息真正在 MainPipe 中被使用要经过两个阶段，分别是在 IPrefetchPipe 中等待入队 WayLookup 阶段和在 WayLookup 中等待出队阶段，在等待期间可能会发生 MSHR 对 Meta/DataArray 的更新，因此需要对 MSHR 的响应进行监听，分为两种情况：

1. 请求在 MetaArray 中未命中，监听到 MSHR 将该请求对应的 cacheline 写入了 SRAM，需要将命中信息更新为命中状态。
2. 请求在 MetaArray 中已经命中，监听到同样的位置发生了其它 cacheline 的写入，原有数据被覆盖，需要将命中信息更新为缺失状态。

为了防止更新逻辑的延迟引入到 DataArray 的访问路径上，在 MSHR 发生新的写入时禁止入队 WayLookup，在下一拍入队。

4.4 WayLookup 子模块文档

WayLookup 为 FIFO 结构, 暂存 IPrefetchPipe 查询 MetaArray 和 ITLB 得到的元数据, 以备 MainPipe 使用。同时监听 MSHR 写入 SRAM 的 cacheline, 对命中信息进行更新。更新逻辑与 IPrefetchPipe 中相同, 见 IPrefetchPipe 子模块文档中的“命中信息的更新”一节。

允许 bypass(即, 当 WayLookup 为空时, 直接将入队请求出队), 为了不将更新逻辑的延迟引入到 dataArray 的访问路径上, 在 MSHR 有新的写入时禁止出队, MainPipe 的 S0 流水级也需要访问 dataArray, 当 MSHR 有新的写入时无法向下走, 所以该措施并不会带来额外影响。

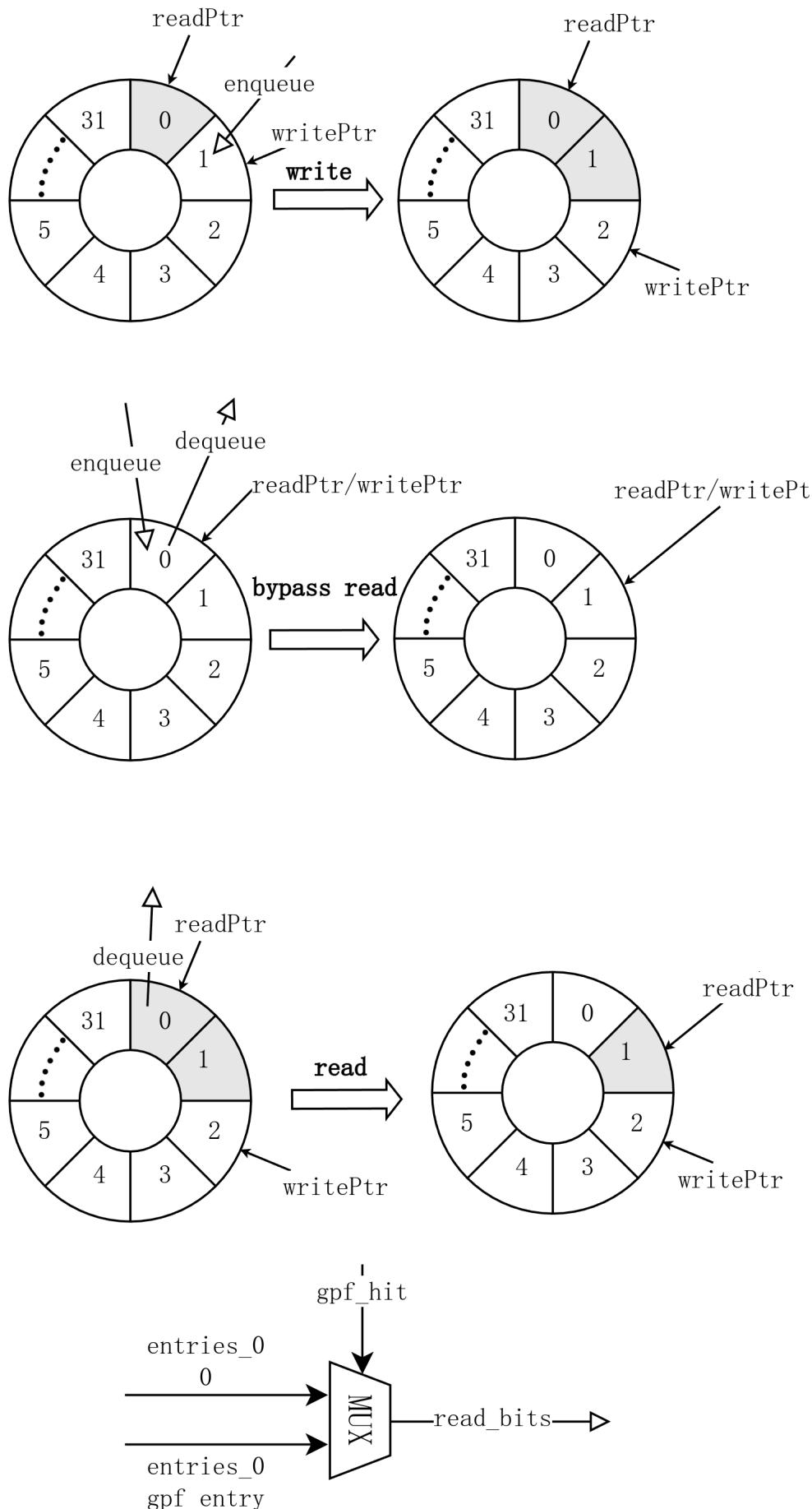


图 4.11: WayLookup 队列结构

prefetcher.io.wayLookupWrite missunit.io.fetchresp

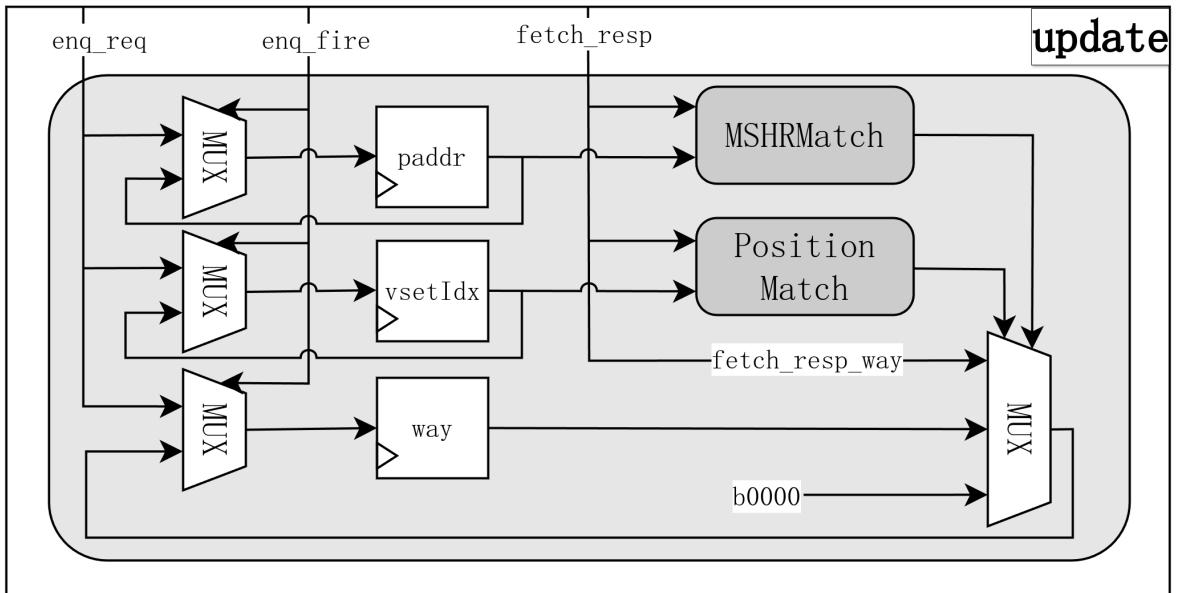


图 4.12: WayLookup 命中信息更新

4.4.1 GPAddr 省面积机制

由于 gpaddr 仅在 guest page fault 发生时有用，并且每次发生 gpf 后前端实际上工作在错误路径上，后端保证会送一个 redirect (WayLookup flush) 到前端（无论是发生 gpf 前就已经预测错误/发生异常中断导致的；还是 gpf 本身导致的），因此在 WayLookup 中只需存储 reset/flush 后第一个 gpf 有效时的 gpaddr。对双行请求，只需存储第一个有 gpf 的行的 gpaddr。

在实现上，把 gpf 相关信号（目前只有 gpaddr）与其它信号（paddr, etc.）拆成两个 bundle，其它信号实例化 nWayLookupSize 个，gpf 相关只实例化一个寄存器。同时另用一个 gpfPtr 指针。总计可以节省 $(nWayLookupSize * 2 - 1) * GPAAddrBits - \log_2(nWayLookupSize) - 1$ bit 的寄存器。当 prefetch 向 WayLookup 写入时，若有 gpf，且 WayLookup 中没有已经存在的 gpf，则将 gpf/gpaddr 写入 gpf_entry 寄存器，同时将 gpfPtr 设置为此时的 writePtr。当 MainPipe 从 WayLookup 读取时，若 bypass，则仍然直接将 prefetch 入队的数据出队；否则，若 readPtr == gpfPtr，则读出 gpf_entry；否则读出全 0。需要指出：

1. 考虑双行请求，gpaddr 只需要存一份（若第一行发生 gpf，则第二行肯定也在错误路径上，不必存储），但 gpf 信号本身仍然需要存两份，因为 ifu 需要判断是否是跨行异常。
2. readPtr==gpfPtr 这一条件可能导致 flush 来的比较慢时 readPtr 转了一圈再次与 gpfPtr 相等，从而错误地再次读出 gpf，但如前所述，此时工作在错误路径上，因此即使再次读出 gpf 也无所谓。
3. 需要注意一个特殊情况：一个跨页的取指块，其 32B 在前一页且无异常，后 2B 在后一页且发生 gpf，若前 32B 正好是 16 条 RVC 压缩指令，则 IFU 会将后 2B 及对应的异常信息丢弃，此时可能导致下一个取指块的 gpaddr 丢失。需要在 WayLookup 中已有一个未被 MainPipe 取走的 gpf 及相关信息时阻塞 WayLookup 的入队（即 IPrefetchPipe s1 流水级），见 PR#3719。

4.5 MissUnit 子模块文档

MissUnit 负责处理 ICache 缺失的请求，通过 MSHR 进行管理，通过 Tilelink 总线与 L2 Cache 进行交互，并负责向 MetaArray 和 DataArray 发送写请求，向 MainPipe 发送响应。

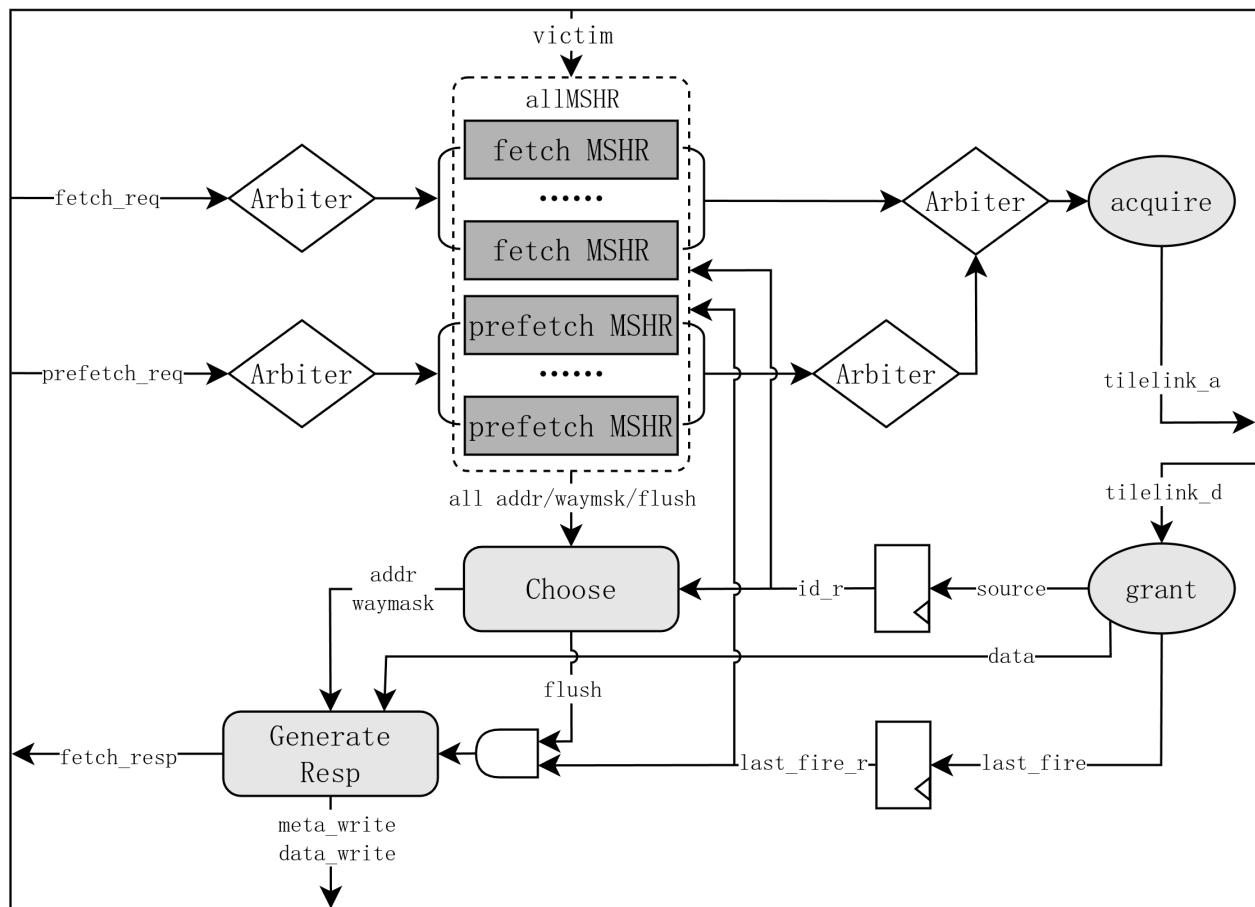


图 4.13: MissUnit 结构

4.5.1 MSHR 的管理

MissUnit 通过 MSHR 分别管理取指请求和预取请求，为了防止 flush 时取指 MSHR 不能完全释放，设置取指 MSHR 的数量为 4，预取 MSHR 的数量为 10。采用数据和地址分离的设计方法，所有的 MSHR 公用一组数据寄存器，在 MSHR 只存储请求的地址信息。

4.5.2 请求入队

MissUnit 接收来自 MainPipe 的取指请求和来自 IPrfetechPipe 的预取请求，取指请求只能被分配到 fetchMSHR，预取请求只能分配到 prefetchMSHR，入队时采用低 index 优先的分配方式。在入队的同时对 MSHR 进行查询，如果请求已经在 MSHR 中存在，就丢弃该请求，对外接口仍表现 fire，只是不入队到 MSHR 中。在入队时向 Replacer 请求写入 waymask。

4.5.3 acquire

当到 L2 的总线空闲时，选择 MSHR 表现进行处理，整体 fetchMSHR 的优先级高于 prefetchMSHR，只有没有需要处理的 fetchMSHR，才会处理 prefetchMSHR。对于 fetchMSHR，采用低 index 优先的优先级策略，因为同时最多只有两个请求需要处理，并且只有当两个请求都处理完成时才能向下走，所有 fetchMSHR 之间的优先级并不重要。对于 prefetchMSHR，考虑到预取请求之间具有时间顺序，采用先到先得的优先级策略，在入队时通过一个 FIFO 记录入队顺序，处理时按照入队顺序进行处理。

4.5.4 grant

通过状态机与 Tilelink 的 D 通道进行交互，到 L2 的带宽为 32byte，需要分 2 次传输，并且不同的请求不会发生交织，所以只需要一组寄存器来存储数据。当一次传输完成时，根据传输的 id 选出对应的 MSHR，从 MSHR 中读取地址、掩码等信息，将相关信息写入 SRAM，同时将 MSHR 释放。

4.6 Replacer 子模块文档

采用 PLRU 更新算法，考虑到每次取指可能访问连续的 doubleline，对于奇地址和偶地址设置两个 replacer，在进行 touch 和 victim 时根据地址的奇偶分别更新 replacer。

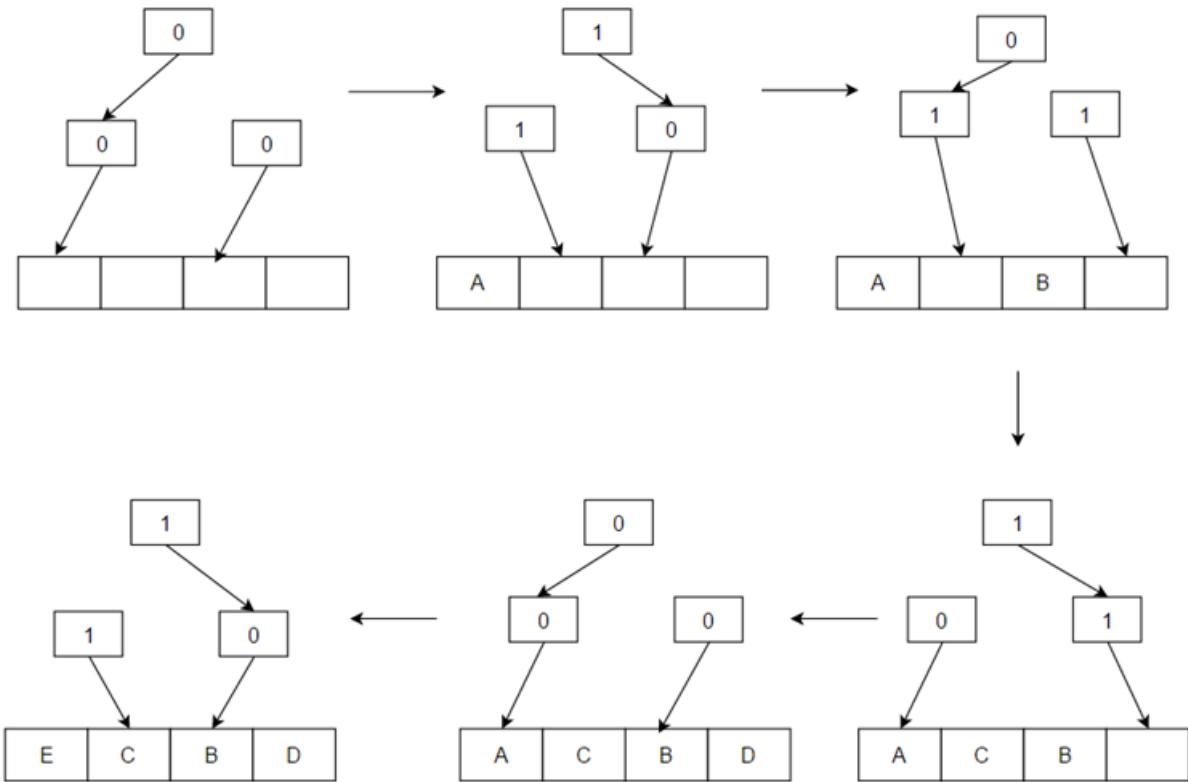


图 4.14: PLRU 算法示意

4.6.1 touch

Replacer 具有两个 touch 端口，用以支持双行，根据 touch 的地址奇偶分配到对应的 replacer 进行更新。

4.6.2 victim

Replacer 只有一个 victim 端口，因为同时只有一个 MSHR 会写入 SRAM，同样根据地址的奇偶从对应的 replacer 获取 waymask。并且在下一拍再进行 touch 操作更新 replacer。

4.7 CtrlUnit 子模块文档

目前 CtrlUnit 主要负责 ECC 校验使能/错误注入等功能

4.7.1 mmio-mapped CSR

CtrlUnit 内实现了一组 mmio-mapped CSR,连接在 tilelink 总线上,地址可由参数 `cacheCtrlAddressOpt` 配置，默认地址为 0x38022080。总大小为 128B。

当参数 `cacheCtrlAddressOpt` 为 `None` 时，CtrlUnit 不会实例化。此时 ECC 校验使能默认开启，软件不可控制关闭；软件不可控制错误注入。

目前实现的 CSR 如下：

| | | | | | | | | |
|------|---------|------|--------|---------|---------|--------|--------|--|
| 64 | 10 | 7 | 4 | 2 | 1 | 0 | | |
| 0x00 | eccctrl | WARL | ierror | istatus | itarget | inject | enable | |

| | |
|------------------------------|---|
| 64 PAddrBits-1 | 0 |
| 0x08 ecciaddr WARL paddr | |

| CSR | field | desp |
|----------|---------|---|
| eccctrl | enable | ECC 错误校验使能, 原 sfetchctl(0) |
| eccctrl | inject | ECC 错误注入使能, 写 1 开始注入, 读恒 0 |
| eccctrl | itarget | ECC 错误注入目标, 见后表 |
| eccctrl | istatus | ECC 错误注入状态 (read-only), 见后表 |
| eccctrl | ierror | ECC 错误原因 (read-only), 仅在 eccctrl.istatus==error 时有 效, 见后表 |
| ecciaddr | paddr | ECC 错误注入物理地址 |

eccctrl.itarget:

| value | target |
|-------|-----------|
| 0 | metaArray |
| 2 | dataArray |
| 1/3 | rsvd |

eccctrl.istatus:

| value | status |
|-------|----------|
| 0 | idle |
| 1 | working |
| 2 | injected |
| 7 | error |
| 3-6 | rsvd |

eccctrl.ierror:

| value | error |
|-------|--|
| 0 | ECC 未使能 (i.e. !eccctrl.enable) |
| 1 | inject 目标 SRAM 无效 (i.e. eccctrl.itarget==rsvd) |
| 2 | inject 目标地址 (i.e. ecciaddr.paddr) 不在 ICache 中 |
| 3-7 | rsvd |

4.7.2 错误校验使能

CtrlUnit 的 `eccctrl.enable` 位直接连接到 MainPipe，控制 ECC 校验使能。当该位为 0 时，ICache 不会进行 ECC 校验。但仍会在重填时计算校验码并存储，这可能会有少量的额外功耗；如果不计算，则在未使能转换成使能时需要冲刷 ICache（否则读出的 parity code 可能是错的）。

4.7.3 错误注入使能

CtrlUnit 内部使用一个状态机控制错误注入过程，其 `status`（注意：与 `eccctrl.istatus` 不同）有：

- `idle`: 注入控制器闲置
- `readMetaReq`: 发送读取 metaArray 请求
- `readMetaResp`: 接收读取 metaArray 响应
- `writeMeta`: 写入 metaArray
- `writeData`: 写入 dataArray

当软件向 `eccctrl.inject` 写入 1 时，进行以下简单检查，检查通过时状态机进入 `readMetaReq` 状态：

- 若 `eccctrl.enable` 为 0，报错 `eccctrl.ierror=0`
- 若 `eccctrl.itarget` 为 `rsvd(1/3)`，报错 `eccctrl.ierror=1`

在 `readMetaReq` 状态下，CtrlUnit 向 MetaArray 发送 `ecciaddr.paddr` 地址对应的 set 读取的请求，等待握手。握手后转移到 `readMetaResp` 状态。

在 `readMetaResp` 状态下，CtrlUnit 接收到 MetaArray 的响应，检查 `ecciaddr.paddr` 地址对应的 ptag 是否命中，若未命中则报错 `eccctrl.ierror=2`。否则，根据 `eccctrl.itarget` 进入 `writeMeta` 或 `writeData` 状态。

在 `writeMeta` 或 `writeData` 状态下，CtrlUnit 向 MetaArray/DataArray 写入任意数据，同时拉高 `poison` 位，写入完成后状态机进入 `idle` 状态。

ICache 顶层中实现了一个 Mux，当 CtrlUnit 的状态机不为 `idle` 时，将 MetaArray/DataArray 的读写口连接到 CtrlUnit，而非 MainPipe/IPrefetchPipe/MissUnit。当状态机 `idle` 时反之。

5 昆明湖 PrunedAddr 文档

5.1 背景介绍

在 RISC-V 手册中，对于 B-Type 和 J-Type 的指令描述如下：

The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

Similarly, the only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

这意味着对于除了 `jalr` 以外的分支指令，跳转目标的最低位一定为 0。而对于 `jalr` 指令，手册中对计算跳转目标的描述如下：

The target address is obtained by adding the sign-extended 12-bit I-immediate to the register `rs1`, then setting the least-significant bit of the result to zero.

这意味着 `jalr` 的跳转目标最低位也为 0。

综合这两部分，我们可以发现所有的 PC 最低位一定为 0。于是，前端所有涉及到 PC 的部分均不用存储最低位，这可以节省面积。

实际上，当 C 扩展关闭时，PC 的低 2 位一定为 0，否则会报指令非对齐异常。

5.2 使用指南

`PrunedAddr` 的设计目标是使用起来和 `UInt` 尽可能相同。但是，出于 Chisel 的限制，这一目标并不能全部实现。在那些和 `UInt` 不同的地方，可以类比 `Reg` 或 `Wire`。具体的使用指南如下：

- 使用 `PrunedAddrInit` 进行从 `UInt` 到 `PrunedAddr` 的转换，例如 `val addr1 = PrunedAddrInit(addr2)`，其中 `addr2` 的类型是 `UInt`
- 使用 `toUInt` 进行从 `PrunedAddr` 到 `UInt` 的转换，例如 `addr.toUInt`
- `def +(offset: UInt)` 这一方法仅当 `offset` 为立即数时才应被使用。对于其他情况，应当使用 `def +(offset: PrunedAddr)`。如果 `offset` 是 `UInt` 且不是立即数，应当将 `offset` 转换为 `PrunedAddr`

5.3 遗留问题

- 目前，对于 PrunedAddr 的使用仅限于前端内部，前端给后端的接口仍然使用 UInt。理想情况下，后端也应当全部使用 PrunedAddr。此时 toUInt 这一方法应当只在输出调试信息时使用
- def >>(offset: Int) 这一方法应当删除以减少混乱。对应的功能可以由直接选择对应位来实现
- 对于裁剪的位数，目前的实现中使用了 instOffset 这一参数。理论上来说，这一参数应当会随着 C 扩展的开关发生变化。但现在香山实际上不支持关闭 C 扩展，关闭 C 扩展的正确性还未经过验证

第二部分

Backend 后端

6 后端整体介绍

Backend 是香山处理器的后端，其中包括指令译码 (Decode)、重命名 (Rename)、分派 (Dispatch)、调度 (Schedule)、发射 (Issue)、执行 (Execute)、写回 (Writeback) 和退休 (Retire) 等多个组件，如图 6.1 所示。

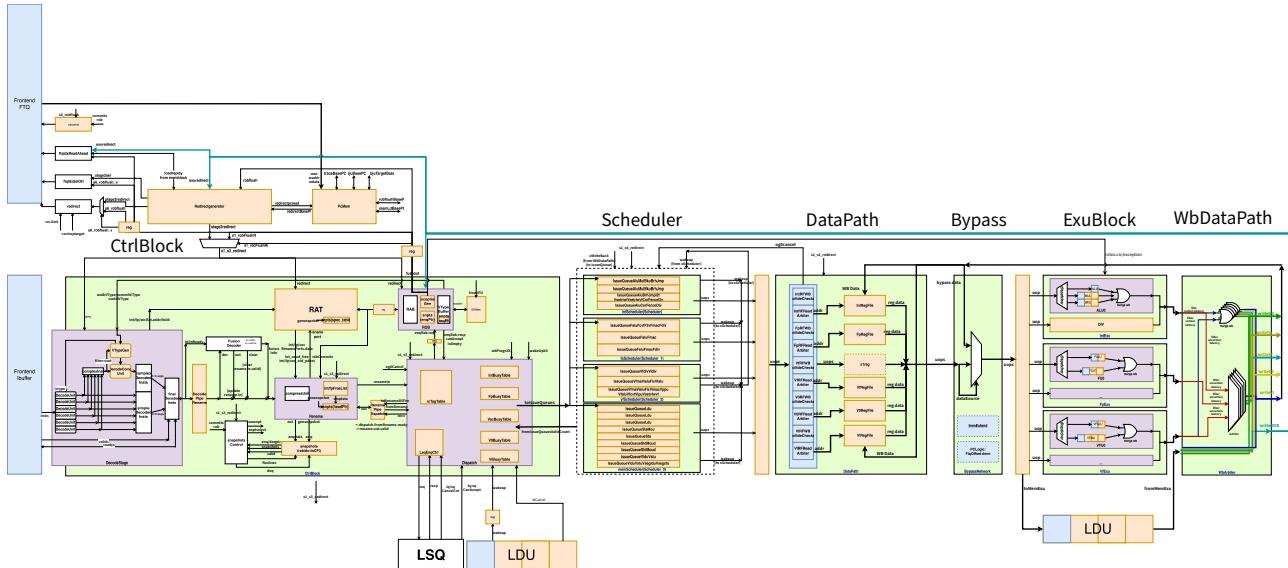


图 6.1: 后端整体框架

6.1 基本技术规格

- 6 宽度的译码、重命名和分派
 - 224 项整数寄存器堆、192 项浮点寄存器堆、128 项向量寄存器堆
 - Move 指令消除
 - 指令融合
- 160 项的 ROB
 - 支持 ROB 压缩 (至多每项 6 个 uop)
 - 每周期最多退休 8 项
 - 快照恢复
- Rename Buffer
 - 256 项 RAB
 - 指令提交和寄存器写回
- 整数、浮点和向量计算

7 CtrlBlock 控制模块

7.1 CtrlBlock

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/15
- commit: xxx

7.1.1 术语说明

表 7.1: 术语说明

| 缩写 | 全称 | 描述 |
|-------|----------------------------|--------|
| - | Decode Unit | 译码单元 |
| - | Fusion Decoder | 指令融合 |
| ROB | Reorder Buffer | 重排序缓存 |
| RAT | Register Alias Table | 重命名映射表 |
| - | Rename | 重命名 |
| LSQ | Load Store Queue | 访存指令队列 |
| - | Dispatch | 派遣 |
| IntDq | Int Dispatch Queue | 定点派遣队列 |
| fpDq | Float Point Dispatch Queue | 浮点派遣队列 |
| lsDq | Load Store Dispatch Queue | 访存派遣队列 |
| - | Redirect | 指令重定向 |
| pcMem | PC MEM | 指令地址缓存 |

7.1.2 子模块列表

表 7.2: 子模块列表

| 子模块 | 描述 |
|----------|--------|
| dispatch | 指令派遣模块 |
| decode | 指令译码模块 |

| 子模块 | 描述 |
|---------------|-------------|
| fusionDecoder | 指令融合模块 |
| rat | 重命名表 |
| rename | 重命名模块 |
| redirectGen | 重定向生成模块 |
| pcMem | 指令地址缓存 |
| rob | 重排序缓冲 |
| trace | 指令 trace 模块 |
| snpt | 快照模块 |

7.1.3 设计规格

译码宽度: 6
 重命名宽度: 6
 分派宽度: 6
 rob 提交宽度: 8
 rab 提交宽度: 6
 rob 大小: 160
 快照大小: 4 项
 整型物理寄存器数: 224
 浮点物理寄存器数: 192
 向量物理寄存器数: 128
 向量 v0 物理寄存器数: 22
 向量 vl 物理寄存器数: 32
 支持重命名快照
 支持 trace 扩展

7.1.4 功能

CtrlBlock 模块包含指令译码 (Decode)、指令融合 (FusionDecoder)、寄存器重命名 (Rename, RenameTable)、指令分派 (Dispatch)、提交部件 (ROB)、重定向处理 (RedirectGenerator) 和快照重命名恢复 (SnapshotGenerator)。

译码功能部件在每个时钟周期会从指令队列头部取出 6 条指令进行译码。译码过程是将指令码翻译为方便功能部件处理的内部码，标识出指令类型、所需要操作的寄存器号以及指令码中可能包含的立即数，用于接下来的寄存器重命名阶段。对于复杂指令，选出后通过复杂译码器 DecodeCompunit 一次一条进行指令拆分，对于 vset 指令存储到 Vtype 中指导指令拆分。最后以复杂指令在前，简单指令在后每周期选出 6 个 uop 传递到重命名阶段。译码阶段还包括发出读 RenameTable 请求。

指令融合会对指令译码得到的 6 个 uop 凑成 (uop0, uop1), (uop1, uop2), (uop2, uop3), (uop3, uop4), (uop4, uop5) 的至多 5 对待融合指令对。然后判断每一对指令是否能够进行指令融合。当前我们支持两种类型的指令融合，分别是融合成为一个带有新的控制信号的指令，以及将第一条指令的操作编码替换为另一个的形式。在判断可以进行指令融合后，我们会对 uop 的操作数，如逻辑寄存器号重新赋值，选择新的操作数。另外，HINT 类指令不支持被指令融合，例如 fence 指令不能够被融合。

重命名阶段负责管理和维护寄存器与物理寄存器之间的映射，通过对逻辑寄存器的重命名，实现指令间依赖的消除，完成指令的乱序调度。重命名模块主要包含 Rename、RenameTable 两个模块，分别负责 Rename 流水级的控制、(体系结构/推测) 重命名表的维护，Rename 中包括 FreeList 以及 CompressUnit 两个模块，负责空闲寄存器的维护以及 Rob 压缩。

派遣阶段将重命名后的指令根据指令类型分发到 4 个调度器中，分别对应于整型，浮点，向量和访存。每个调度器中根据不同的运算操作类型又分为若干的发射队列 (issue queue)，每个发射队列的入口大小为 2。

指令流在 CtrlBlock 的传递过程为：CtrlBlock 读取 Frontend 传入的 6 条指令对应 ctrlflow，经过 decode 增加译码逻辑寄存器、运算操作符等信息，复杂指令经过 DecodeComp 添加指令拆分信息，每周期选出六条 uop 输出，并发出读 RAT 请求。对于可以进行指令融合的 uop，在进入 rename 时进行融合以及清除。之后经过 rename 增加物理寄存器信息以及 rob 压缩信息后传入 dispatch，最后通过 dispatch 进到 rob / rab / vtype 申请 entry，根据指令类型输出到 issue queue。这些模块中只有 issue queue 顺序进，乱序出，其他模块都是顺序进，顺序出。

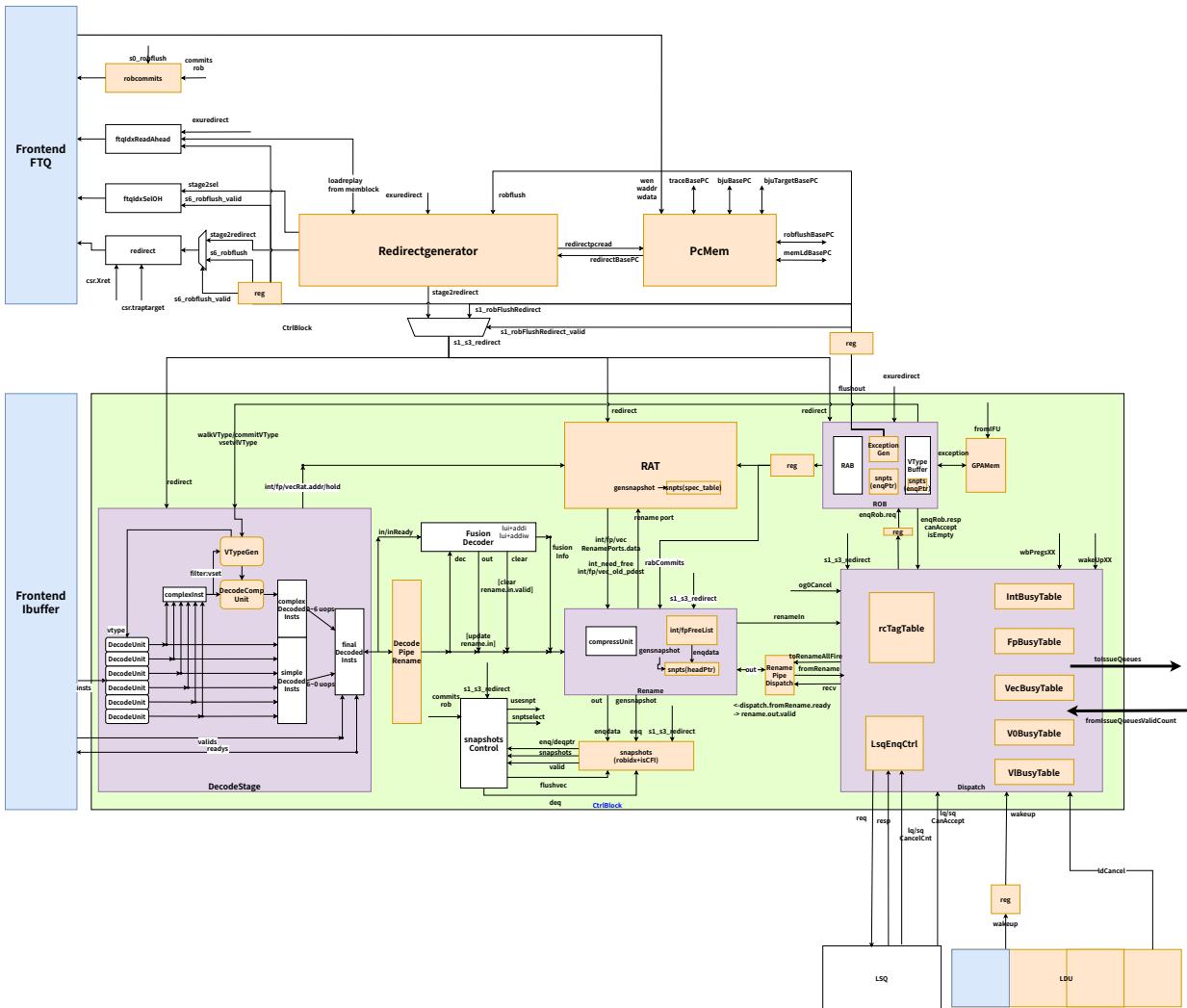


图 7.1: CtrlBlock 总览

7.2 译码

标量指令的译码过程同南湖。

对于向量指令，先使用和标量指令相同结构的译码表进行译码，译码的同时拿到指令拆分类型，接下来会根据指令拆分类型进行拆分，拆分的过程相当于重新修改源寄存器号、源寄存器类型、目标寄存器号、目标寄存器类型、更新 uop 数量，用于控制 rob 写回时一条指令需要写回的数量。直到拆分出的所有 uop 完成 rename 过程后，译码 ready 信号才能够置为 1。

由于除了 i2f 的标量浮点指令现在使用向量浮点模块运行，因此 fpdecoder 中的译码信号只使用其中用到的 4 种 (typeTagOut、wflags、typ、rm)，用法与南湖相同。使用向量浮点模块运行的浮点指令，需要在向量译码单元中获得使用的 futype 以及 fuotype，并使用 1bit isFpToVecInst 信号区分，该浮点指令是浮点指令还是向量浮点指令，从而在共用向量运算单元时能进行区分。

7.2.1 译码阶段输入

译码阶段除了接受来自前端的指令流，还需要接受来自 rob 的 Vtype 相关：walk，commit，vsetvl 信息，指导向量复杂指令译码。

7.2.2 译码输出

与 fusionDecode：输出指令流以及控制指令融合是否开启。

与 rename：流水输出 6 个 uop；如果出现 redirect 阻塞直到 Ctrlblock 中 redirect 发往前端后，前端发出正确指令流。

与 RAT：译码发出读推测重命名请求。

7.3 FusionDecoder

指令融合模块负责找出译码模块译码后的 uop 是否存在一定的联系，从而可以将多个 uop（当前仅支持两条指令的融合）需要做的事情融合为一条 uop 能够完成的事情。

指令融合会对指令译码得到的 6 个 uop 湍成 (uop0, uop1), (uop1, uop2), (uop2, uop3), (uop3, uop4), (uop4, uop5) 形式的至多 5 对待融合指令对。然后判断每一对指令是否能够进行指令融合。当前我们支持两种类型的指令融合，分别是融合成为一个带有新的控制信号的指令，以及将第一条指令的操作编码替换为另一个的形式。在判断可以进行指令融合后，我们会对 uop 的操作数，如逻辑寄存器号重新赋值，选择新的操作数。另外，HINT 类指令不支持被指令融合，例如 fence 指令不能够被融合。

例如，slli r1, r0, 32 和 srli r1, r1, 32 将 r0 中的数左移 32 位后存入 r1，然后再次右移 32 位。其等价于 add.uw r1, r0, zero（伪指令 zext.w r1, r0），即将 r0 中的数扩展后移动到 r1 中。

输入为译码后的至多 6 条 uop 以及他们的原始指令编码，以及相应的 valid 信号，这里输入 inready 只有 5 位（即译码宽度减一），因为我们需要将 uop 错位两两配对为至多 5 对待融合指令。inReady[i] 表示已经准备好可以接受 in(i+1)。

输出宽度为译码宽度减一，包括指令融合替换，需要替换 fuType, fuOpType, lsrc2（第二个操作数的逻辑寄存器号，如有），src2Type（第二个操作数的类型），selImm（立即数类型）。以及指令融合信息，如 rs2 来自于 rs1/rs2/zero。同时还需要输出一组译码宽度的布尔向量 clear，表征该 uop 是否被指令融合需要被清除掉，当前设想每条指令融合后会将第二条指令清除掉。第 0 个 uop 的 clear 是不会为 true 的，因为我们默认将后面的指令融合到前面的指令上，因此无论是否融合，uop 0 始终不会因为指令融合而消失。

输出有效要求：指令对有效（从译码模块传来的 uop 对有效），不能被指令融合清除掉，有可行的指令融合结果，以及不能是 Hint 类指令。同时将 fuType, src2Type, rs2FromZero 等信息赋值

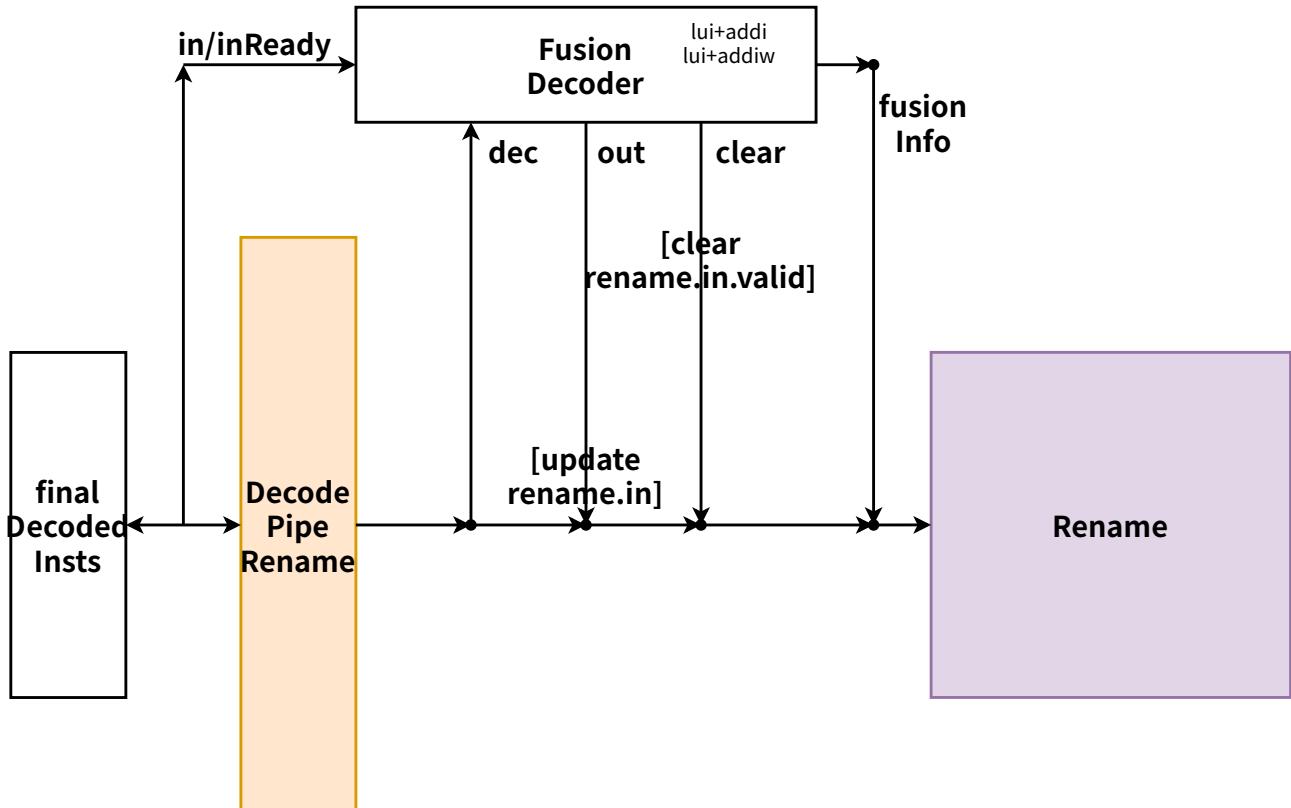


图 7.2: fusion decoder 总览

7.4 Redirect

在 ctrlblock 中主要负责 redirect 的生成以及发往各个模块。

7.4.1 RedirectGenerator

RedirectGenerator 模块管理不同来源的重定向信号（如执行单元和 load），并决定是否发生重定向，如何刷新相关信息。它通过多级寄存器和同步机制确保数据流的正确性，且通过地址转换和错误检测保证指令执行的正确性。

将当前最老的执行 redirect 的 fullTarget 和 cfiUpdate.target 拼接得到 fullTarget 字段。另外如果当前最老的执行 redirect 不是来自于 CSR，则还需要基于指令地址的翻译类型检查 IAF, IPF 和 IGPF 等地址的合法性。

然后从最老的执行 redirect 和 load redirect 中选出一个最老的 redirect，同时还需要保证这个最老的 redirect 不会被 robFlush 或者之前的 redirect 刷掉。

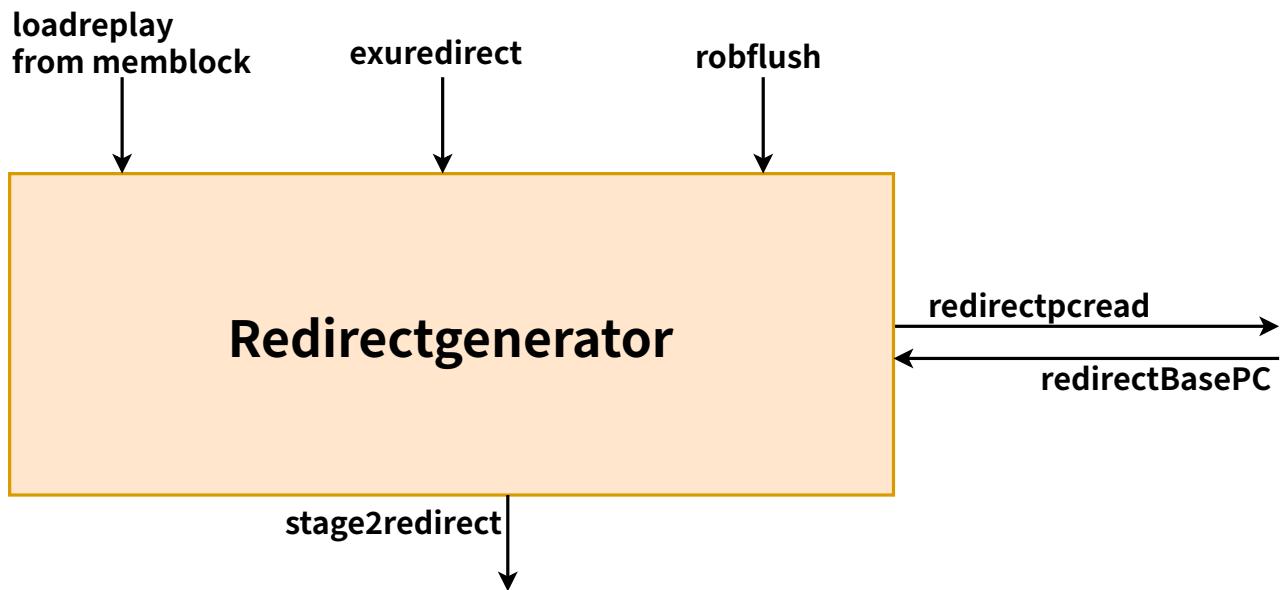


图 7.3: redirect 总览

7.4.2 redirect 的生成

Ctrlblock 中生成的 Redirect 主要包括两个来源:

- 通过 redirectgen 汇总的处理器执行时发生的错误 (包含分支预测和访存违例) (后面称这部分重定向为 exuredirect);
- 以及来自 rob exceptiongen 生成的 robflush : 中断 (csr)/异常/刷流水(csr+fence+load+store+varith+vload+vstore) + 前端异常。Rob 中发来的异常/中断/刷流水重定向处理类似。

对于 redirectgen 汇总的重定向:

- 功能单元写回的 redirect(jump, brh) 在打一拍且没有被更老的已经处理过 redirect 取消的情况下输入到 redirectgen 模块。
- Memblock 写回的 violation (访存违例) 在打一拍且没有被更老的已经处理过的 redirect 取消的情况下输入到 redirectgen。

Redirectgen 选择最老的 redirect 在输入后等待一拍, 加上从 pcMem 读回的数据后再输出。

对于 robflush 信号, 在接收到后, 同样需要等待一拍加上从 pcMem 读回的数据。

Ctrlblock 生成 Redirect 时会优先重定向 robflush 信号, 当不存在 robflush 时才会处理 exuredirect。

上述部分整体框图如下:

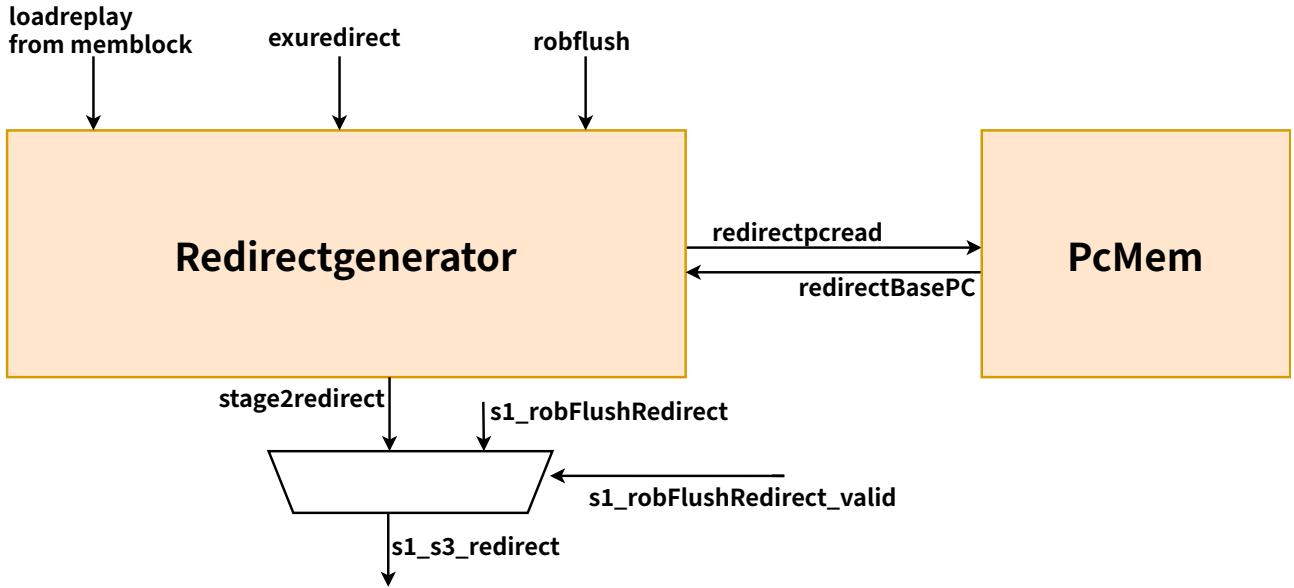


图 7.4: redirect 的生成

7.4.3 redirect 分发

Ctrlblock 在生成出 Redirect 信号后，向流水级各个模块分发重定向信号。

- 对于 decode 发送当前 redirect 或 redirectpending(即 decode 等待直到 Ctrlblock 发给前端的 redirect 准备完成，使得前端有正确指令流到达之后才可继续流水)；
- 对于 rename, rat, rob, dispatch, snpt, mem 发送当前 redirect；
- 对于 issueblock, datapath, exublock 发送打一拍后的 redirect。

其中比较特殊的是发送给前端的 redirect。发送给前端的重定向以及造成的影响，总共包括三部分：rob_commit, redirect, ftqid(readAhead, seloh)。

7.4.3.1 对于 rob commit

由于向前端传递的 flush 信号可能会延迟几个周期，并且如果在 flush 前继续提交，可能会导致提交后 flush 的错误。因此，我们将所有的 flush 视为异常，确保前端的处理行为一致，当 ROB 提交一条带 flush 信号的指令时，我们需要在 ctrlblock 直接刷掉带有 robflush 的 commit，告知前端进行 flush，但是不进行提交。

而对于 exuredirect，其对应的指令需要在写回 rob 等待 walk 完毕后才可以提交，因此这两类 redirect 不需要再特殊处理，其提交一定在其写回之后。

7.4.3.2 对于 redirect：

发送给前端的 redirect 信号还包括额外的 CFIupdate，而 ftq 信息通过额外的 readAhead 以及 seloh 更新。

对于 exuredirect，它们的 CFIupdate 和 ftqid 等信息在从功能单元传递回来的时候已经包含在里面了，因此无需进行特殊处理。

对于 rob 发出的 flush，exception 对 CFI 更新的目的地址需要等待从 CSR 中得到：首先 rob 发出 flush 信号，产生 exception，向 CSR 发送 redirect 表示有 exception 产生，并从 CSR 得到 Trap Target 返回给 ctrlblock，最后再向前端发出 redirect。

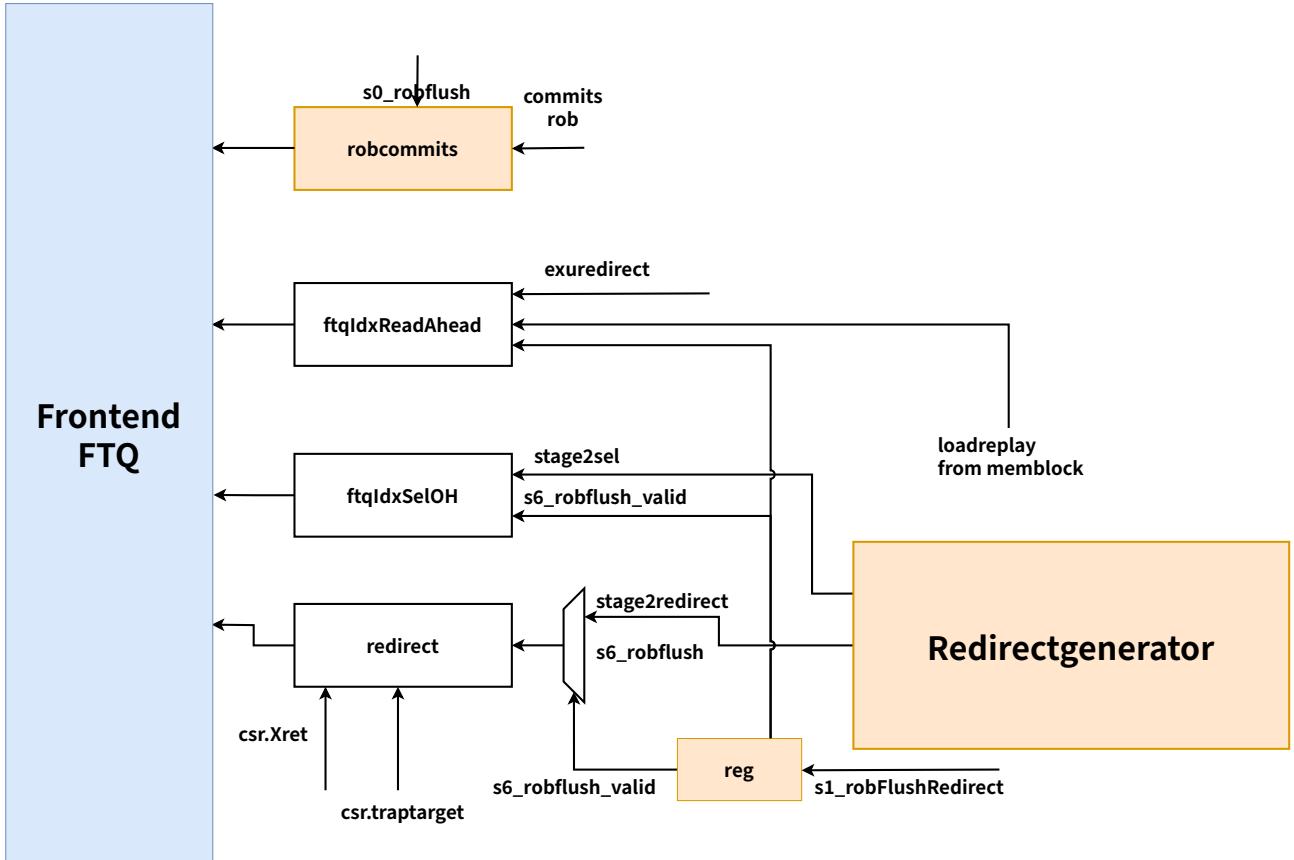


图 7.5: 发向前端的 redirect

其余的冲刷流水导致的目的地址更新，base pc 通过之前与 pcmem 交互得到，并在 ctrlblock 中根据是否刷自身加上偏移来生成目的地址。

其中比较特殊的是 CSR 发出的冲刷流水中 XRet，这种情况下目的地址更新也需要从 CSR 中得到，不过 CSR 生成 Xret 的通路不需要再依赖 rob 发回的 exception，可以直接与 Ctrlblock 通过 csrio 交互。

7.4.3.3 对于 ftqIdx:

Ctrlblock 主要发送两组数据 ftqIdxAhead 以及 ftqIdxSelOH。

其中 ftqIdxAhead 用于前端提前一拍读取到重定向相关的 ftqid。ftqIdxAhead 是一个大小为 3 的 FtqPtr 向量，其中第一个是执行的 redirect (jmp/brh)，第二个是 load 的 redirect，第三个是 robflush。

ftqIdxSelOH 用于选择有效的 ftqid：前两个通过 redirectgen 输出的独热码选择，第三个通过发送到前端的 redirect 是否有效选择。

7.4.4 保证 redirect 发出的顺序

为了保证执行正确，较新的 redirect 不能先于较老的 redirect 分发。以下分四类情况说明：

(1) 新的 exuredirect 在旧的 robflush 之后发出：

exuredirect 在写回时，会往前看是否已经有更老的 redirect。

在 robflush 到来时，对于较晚生成的 exuredirect，会直接在 exublock 中被刷掉；对于较早生成，还未来得及被 robflush 刷掉的 exuredirect，会检查是否有较老的 redirect，如果有则也会被刷掉。

(2) 新的 exuredirect 在旧的 exuredirect 之后：

exuredirect 在写回时，会往前看是否已经有更老的 redirect。

在发生 redirect 时，对于较晚生成的 exuredirect，同样也会直接在 exublock 中被刷掉；对于较早生成，还未来得及被当前 redirect 刷掉的 exuredirect，会检查是否有较老的 redirect，如果有则也会被取消。

(3) 新的 robflush 在旧的 redirect 之后

这种情况，rob 保证了不会出现，robflush 输出结果是当前 robdeq 的指令带有异常/中断标志，而 robdeq 即当前最老的 robidx，一定比现有的 redirect 更老。

(4) 新的 robflush 在旧的 robflush 之后

这一部分主要在 rob 中保证，exceptionGen 获得最老 robflush，同时 robflush 发出时检查上一条 flushout，较新的 robflush 会被取消。

7.5 快照恢复

对于重命名恢复，目前昆明湖采用了快照恢复阶段：在重定向时不一定恢复到 arch 状态，而是可能会恢复到某一个快照状态。快照即根据一定规则，在重命名阶段保存的 spec state，包括 ROB enqptr; Vtypebuffer enqptr; RAT spec table; freelist Headptr（出队指针）以及 ctrlblock 用于总体控制 robidx。目前上述模块均各自维护四份快照。

7.5.1 SnapshotGenerator

SnapshotGenerator 模块主要用于生成快照，存储维护。其本质是一个循环队列，维护最多四份快照。

入队：在循环队列不满，且入队信号未被 redirect 取消的情况下，下一拍在 enqptr 入队，更新 enqptr。

出队：在出队信号未被 redirect 取消的情况下，下一拍在 deqptr 出队，更新 deqptr。

Flush：根据刷新向量在下一拍刷新掉对应的快照。

更新 enqptr：如果有空的快照，选择离 deqptr 最近的作为新的 enq 指针

Snapshots: snapshots 队列寄存器直出

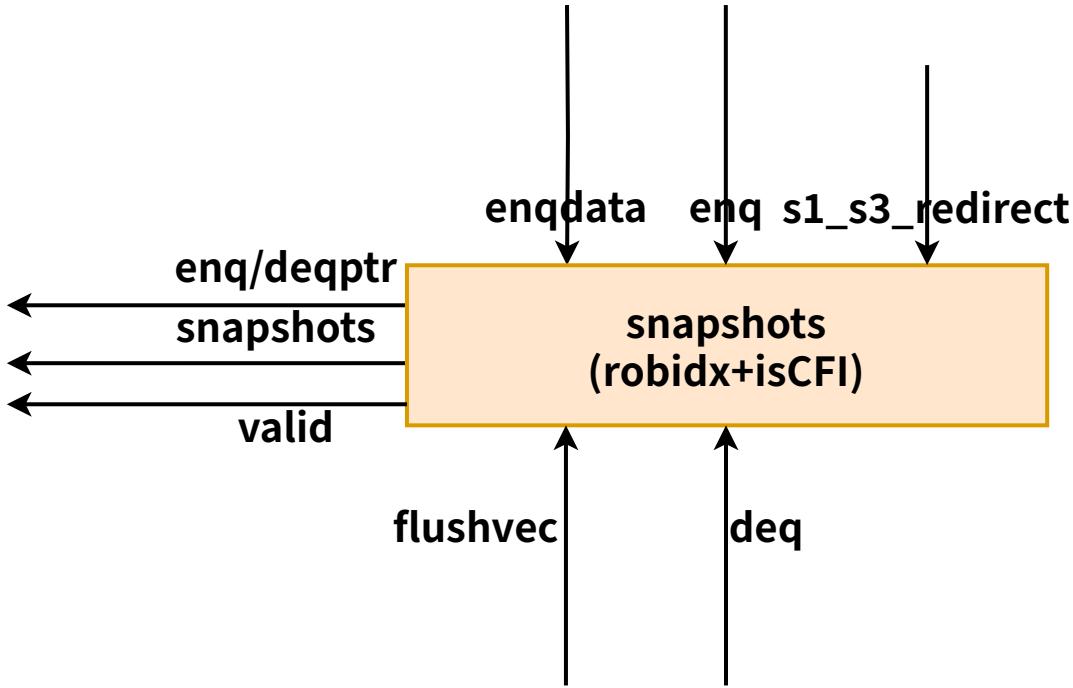


图 7.6: snapshots 总览

7.5.2 快照的创建

对于快照创建时机，目前在 rename 中进行管理。由于注意到对性能造成主要影响的重定向来源仍然是分支错误造成的重定向，因此选择在分支跳转指令处创建快照；同时为了在没有分支跳转的情况下其他的重定向也能用到快照恢复，因此固定每隔 commitwidth*4=32 条 uop 打一份快照。

Rename 模块会对输出的六条 uop 都打上 snapshot 标志，表示 uop 是否需要打上快照，在 Ctrlblock 中会把六条 uop 上的 snapshot 标志汇总到第一条 uop。该操作为了解决快照机制在 blockBackward 下的正确性：即如果在六条 uop 中出现 blockbackward，且在 blockbackward 之后需要打上 snapshot，该 snapshot 会由于 blockbackward 而无法在 rob 中打上快照，将所有 snapshot 放到第一条就可以解决这个问题。

Rat, freelist, 以及 ctrlblock 的快照创建均通过 rename 模块输出的 snapshot 标志控制。存储数据由各个模块自己管理。

Rob, vtype 的快照创建除了 rename 输出流到 rob 的 snapshot 标志还需要考虑非 blockbackward 以及 rab, rob, vtypebuffer 没有满。rob, vtype 的快照创建和前述模块的快照写入可能并不在一个周期，但通过 snapshot 标志随着 rename 输出流到 rob 我们可以保证写入的 robidx 相同即可同步。

7.5.3 快照的删除

快照删除主要包括两种情况，一种在 commit 的时候删除掉过期的快照；另一种是 redirect 的时候删除掉错误路径上的快照。

对于 commit 的时候删除快照：Ctrlblock 通过控制 deq 信号删除快照：robcommit 的八条 uop 有一条与当前 deqptr 指向的快照中第一条 uop 一致则删除过期快照。Ctrlblock 将 deq 信号传递到上述各个模块中同步删除 commit 过期快照。

对于 redirect 的时候：Ctrlblock 通过提供 flushvec 信号删除错误路径上的快照：判断快照的第一条 uop 是否比当前 redirect 要新（这里要注意套圈的情况），如果老则把这条快照刷掉，即 flushvec 相应位置 1。Ctrlblock

将 flushvec 传递到上述模块同步刷新错误路径上的快照。

7.5.4 快照的管理

Ctrlblock 通过自身维护一个存储 robidx 的快照副本，在重定向到来时可以方便的向各个模块告知是否命中快照以及命中快照的编号。Ctrlblock 遍历快照，在存在比当前 redirect 更老（或者不刷自己的情况下相等），允许使用快照恢复，并记录命中快照的编号，传递到上述模块中。

通过快照恢复 spec state 由各个模块自身控制。

上述部分整体框图：

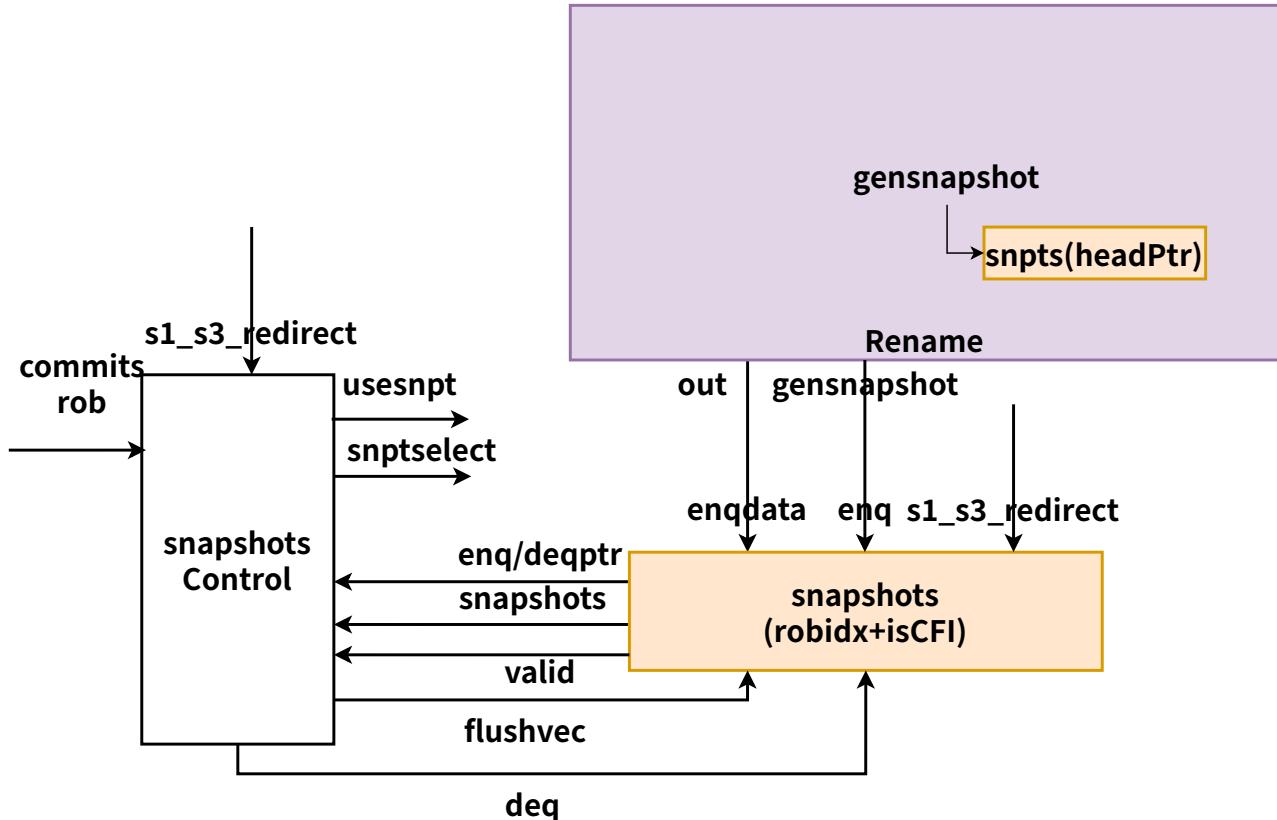


图 7.7: snapshot 的生成、删除和管理

7.6 pcMem

pcMem 实质上是例化了一个 SyncDataModuleTemplate，并需要提供多个读口，1 个写口。大小为 64 项，每一项仅包括 startAddr。

pcMem 读出来的是 base PC，还需要再加上 Ftq Offset 得到完整的 PC。

当前配置下，需要提供 14 个读口，为 redirect, robFlush 各自提供 1 个读口，为 bjuPC 和 bjuTarget 各自提供 3 个读口，为 load 提供 3 个读口，以及为 trace 提供 3 个读口。

输入包括来自前端 Ftq 的写入使能，写入地址和写入数据，以及不同来源的读请求和读地址，分别输出读结果。

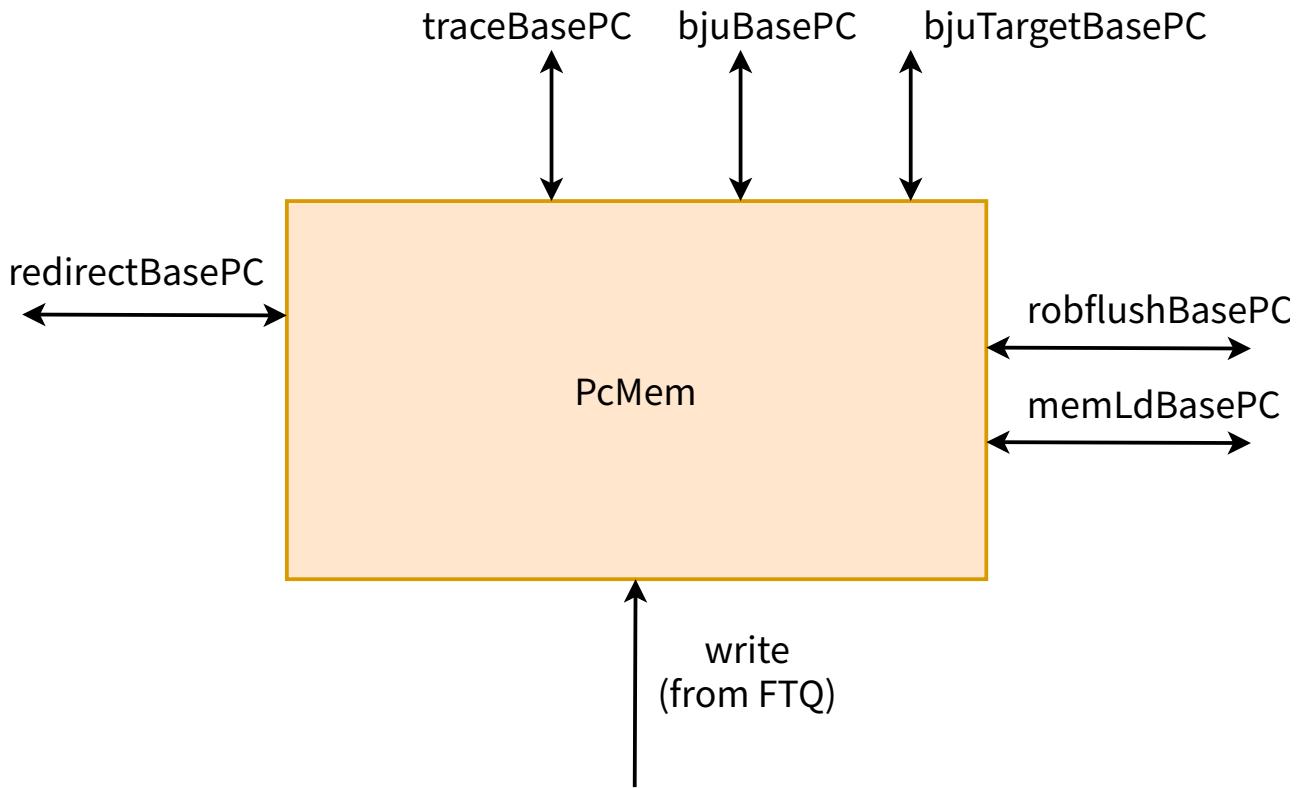


图 7.8: PCMem 总览

7.7 GPAMem

GPAMem 模块类似于 pcMem，例化了一个 SyncDataModuleTemplate，但是只需要提供 1 个读口和 1 个写口，大小为 64 项。每一项主要包括一个 gpaddr，存储前端的 ftq 对应的 gpaddr 信息。

Rob 在 exception 输出的前一拍发出 gpaddr 读请求以读地址的 ftq 信息，第二拍得到返回的 gpaddr 信息。最终通过 robio 与 csr 直接交互。

输入包括来自前端 IFU 的写入使能，写入地址和写入数据，以及来自 rob 的读请求和读地址，向 rob 输出读结果。

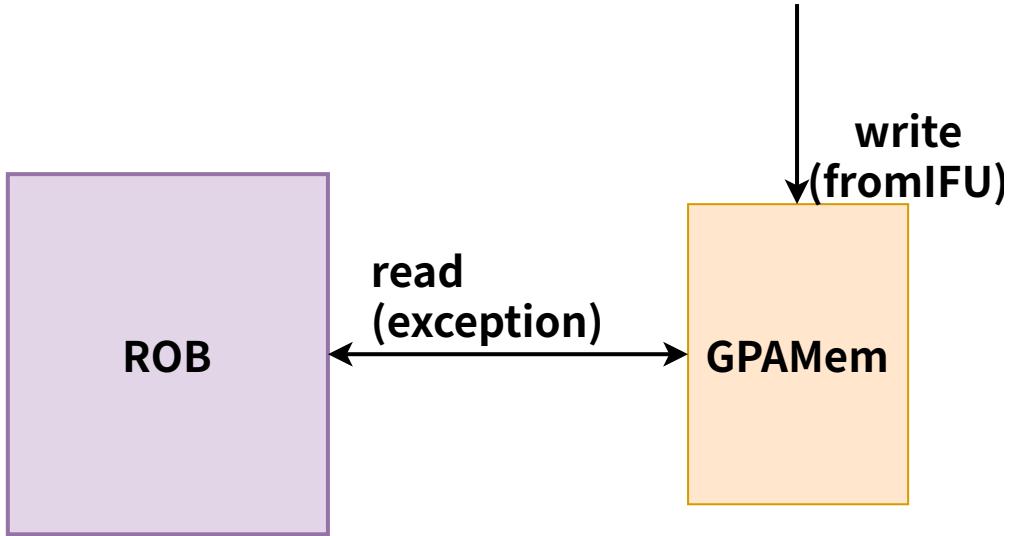


图 7.9: GPAMem 总览

7.8 Trace

ctrlBlock 的 trace 子模块用来收集指令 trace 的信息，其接收来自 rob 指令提交时的信息，在 rob 压缩的基础上进行二次压缩（将不需要 pc 的指令和需要 pc 的指令压缩到一起存入 trace buffer），以减小对 pcMem 的读压力。

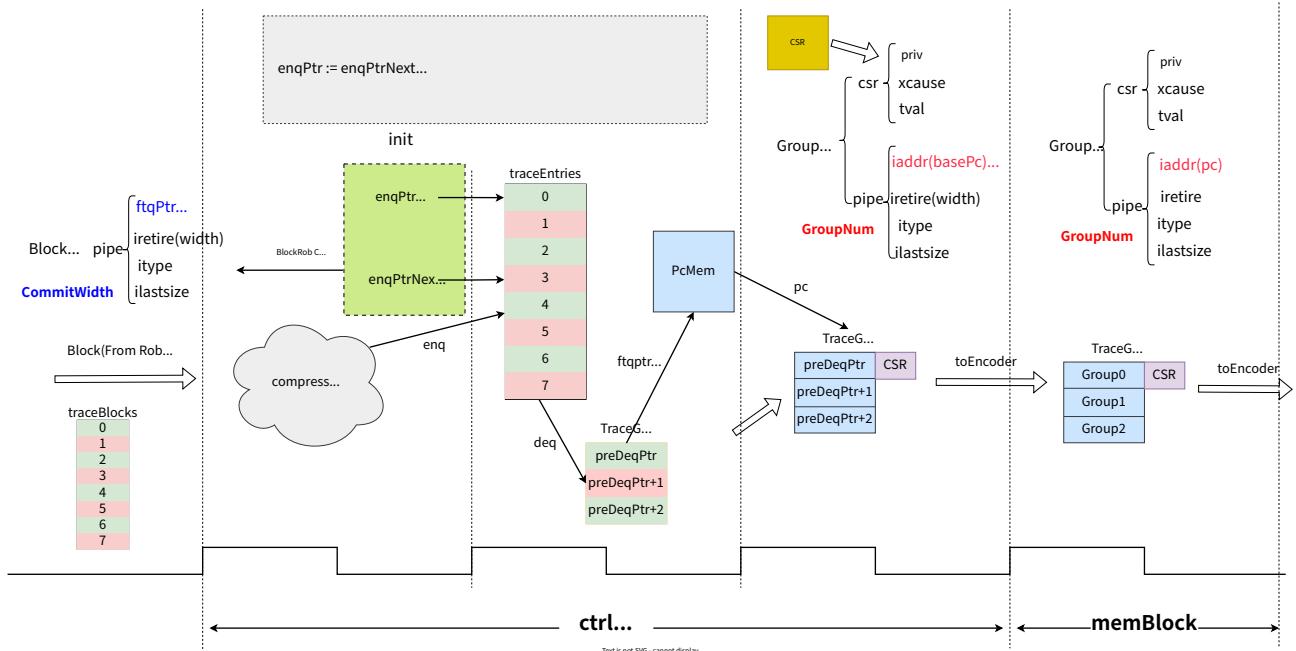


图 7.10: trace 示意图

7.8.1 feature 支持

当前 KMH 核内 trace 的实现只支持指令 trace。核内收集的指令 trace 信息包括: priv, cause, tval, itype, iretire, ilastsize, iaddr；其中 itype 字段支持所有类型。

7.8.2 trace 各级流水线功能:

在 ctrlBlock 里有三拍:

- Stage 0: 将 rob commitInfo 打一拍;
- Stage 1: commitInfo 压缩, 阻塞提交信号产生;
- Stage 2: 根据压缩后的 ftqptr 从 pcmem 中读出 basePc, 从 csr 获取当前提交的指令对应的 priv, xcause, xtval;

memBlock

- Stage 3: 通过 ftqOffset 和从 pcmem 读到的 basePc 算出最终的 iaddr;

7.8.3 trace buffer 压缩机制

当每一组 commitInfo 进入 trace buffer 之前, 都需要做压缩, 即把每一个需要 pc 的 commitinfo 项和其前面的项压缩成一项, 送入 trace buffer, 在进 trace buffer 之前, 会计算当前拍进入 trace buffer 之后, 在下一拍能不能全部出队, 如果不能则去 block rob 的提交, 该 block 会一直 block 到产生该 block 信号的 commitInfo 从 trace buffer 完全出队。产生 blockCommit 信号的 commitInfo 会无脑进 trace buffer, 但是其下一拍的 commitinfo 一定会被堵住。

7.9 XiangShan Decode 设计文档

- 版本: V2R2
- 状态: OK
- 日期: 2025/02/28
- commit: xxx

7.9.1 术语说明

表 7.3: 术语说明

| 缩写 | 全称 | 描述 |
|-----|-----------------|--------------------------|
| - | Decode Unit | 译码单元 |
| uop | Micro Operation | 微操作 |
| - | numOfUop | 一条指令拆分出的 uop 数量 |
| - | numOfWB | 一条指令拆分出的 uop 中需要写回的指令的数量 |
| - | vtypeArch | 最新提交的向量指令 vtype 配置 |
| - | vtypeSpec | 当前向量指令 vtype 配置 |
| - | walkVType | 发生重定向时, 回滚并恢复的 vtype |

7.9.2 子模块列表

表 7.4: 子模块列表

| 子模块 | 描述 |
|-----------------|-------------------|
| DecodeUnit | 译码单元 |
| DecodeUnitComp | 向量指令拆分处理模块 |
| FPDecoder | 浮点指令译码模块 |
| UopInfoGen | 指令拆分类型、数量生成单元 |
| VecDecoder | 向量指令译码模块 |
| VecExceptionGen | 向量异常检查模块 |
| VTypeGen | 向量指令 vtype 配置生成模块 |

7.9.3 设计规格

- 新增向量配置生成模块，向量译码模块，向量指令拆分模块，向量异常检查模块。所有向量指令均进行指令拆分并进入 decoderComp
- 支持同一拍内对 6 条标量指令同时进行译码
- 支持同一拍内对最多 1 条向量指令进行译码
- 部分指令进行转译处理
 - zimop 指令，转译为 src 为 x0, imm 为 0 的 addi 指令
 - 读 vlenb 指令，转译为 src 为 x0, imm 为 VLEN/8 的 addi 指令
 - 读 vl 指令，转译为读 vl 寄存器写标量寄存器的 vset 指令
- 读只读权限的 csr 时，不再置 waitForward 和 blockBackward 信号，支持乱序执行
- 其余功能同南湖

7.9.4 功能

对指令进行译码，将指令 32bits 编码转换为指令的控制信号。指令如果是向量指令或 AMO_CAS 指令，需进行指令拆分。指令拆分的过程是将指令拆分为 1 个或多个 uop，并根据拆分类型对源寄存器号、源寄存器类型、目标寄存器号、目标寄存器类型、使用的功能单元、操作类型进行新的赋值。译码完成后会将带有控制信息的指令传入 rename 模块，rename 模块根据源寄存器号和源寄存器类型进行重命名分配物理寄存器。会在译码阶段对异常指令、异常虚拟化指令进行检查，并将对应的 exceptionVec 中的信号拉高。

7.9.5 整体设计

译码通过例化 6 个 DecodeUnit 模块对输入的指令进行译码，DecodeUnit 会输出指令是否为向量指令的信号，如果是向量指令，则需要将其传入复杂译码器 decoderComp 进行指令拆分。由于向量指令需要经过 DecodeUnit 和 UopInfoGen 进行译码后再进入复杂译码器导致关键路径较长，指令进入复杂译码器后会先暂存一拍，在下一拍进行向量异常检查和指令拆分，会转换为等于等于 1 条 uop，如果 uop 超过 6 条，则需要多拍才能完成译码。如果剩余的 uop 可以在当拍完成译码，会在当拍将需要译码的向量指令传入 decoderComp。假设 rename ready，根据传入的指令的顺序可分为以下几种情况：

1. 标量指令：直接进行译码

2. 向量指令: decoderComp ready 时将向量指令传入 decoderComp 进行指令拆分, 只能处理一条向量指令
3. 向量指令 + 标量指令: decoderComp ready 时将向量指令传入 decoderComp 进行指令拆分, 只能处理一条向量指令, 无法同时处理标量指令
4. 标量指令 + 向量指令: 向量指令前的标量指令直接进行译码。decoderComp ready 时将向量指令传入 decoderComp 进行指令拆分, 只能处理一条向量指令
5. 指令拆分后的 uop+ 标量指令: 假设当拍有 n 个拆分后的 uop 需要 rename, 同时有 m 个标量指令需要 rename, $n+m \leq 6$, 直接进行译码, 否则只译码 $6-n$ 个标量指令
6. 指令拆分后的 uop+ 向量指令: 处理向量指令拆分后的 uop 同向量的情况
7. 指令拆分后的 uop+ 向量指令 + 标量指令: 同标量指令 + 向量指令的情况
8. 指令拆分后的 uop+ 标量指令 + 向量指令: 标量指令的处理同指令拆分后的 uop+ 标量指令的情况, 向量指令的处理同向量指令情况

7.9.6 整体框图

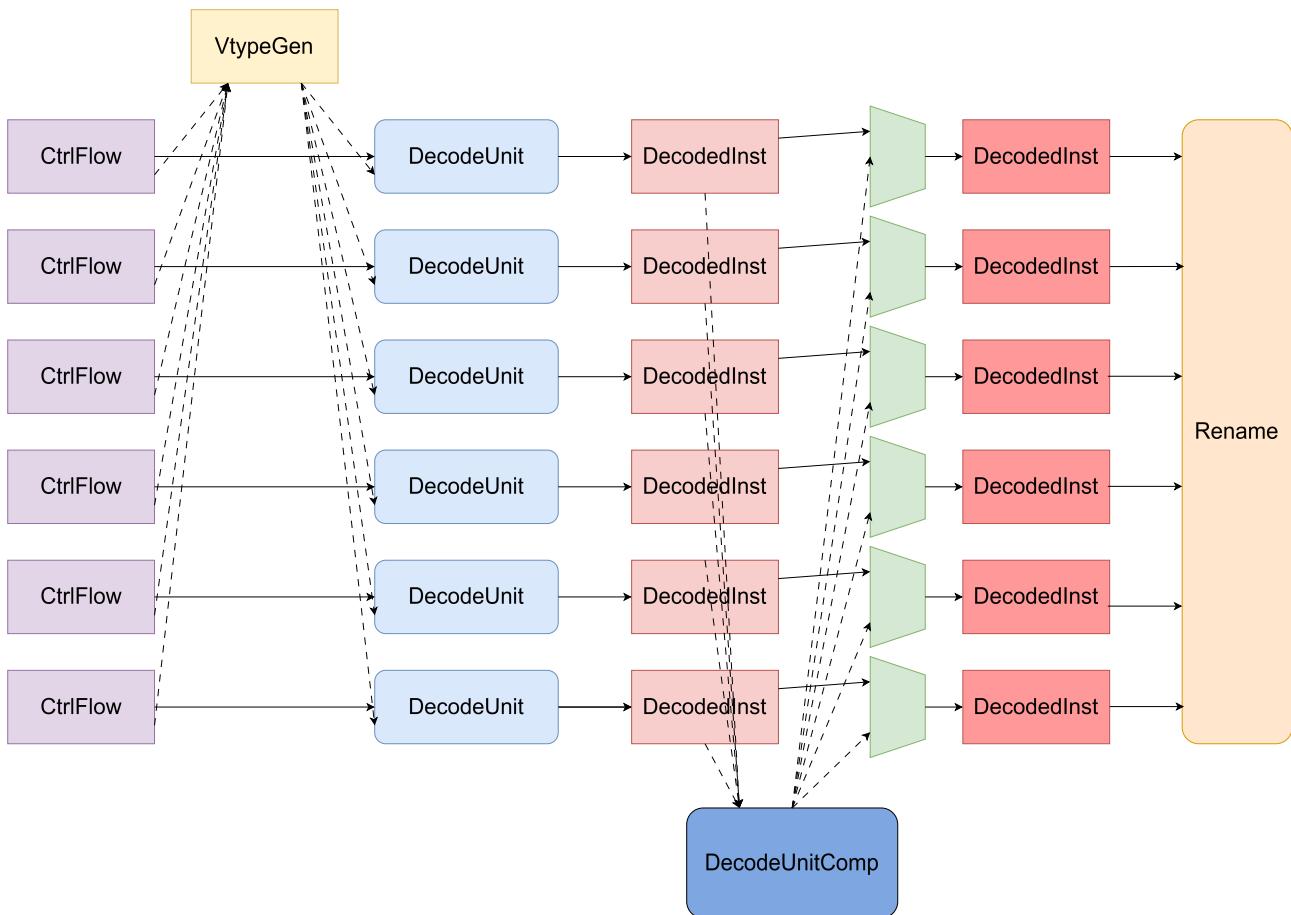


图 7.11: decode

7.9.7 接口列表

见接口文档

7.9.8 二级模块 VTypeGen

VTypeGen 模块主要用于维护当前译码的向量指令需使用的 vtype 配置，每当执行 vset 指令或发生重定向需要回滚时，更新 VTYpeGen 中存储的 vtype 信息。

7.9.8.1 输入

- 来自前端的指令流中的 32bits 指令信息；
- 来自 rob 中 vtype buffer 的 vtype 回滚信息；
- 来自 rob 中 vtype buffer 的 vtype 提交信息；
- 来自 backend 的 vsetvl 指令的 vtype 信息，由于 vsetvl 指令的 vtype 信息需要通过读寄存器而不是译码获得，因此在 vsetvl 指令写回时，会将 vtype 信息传递给 vtypeGen。

7.9.8.2 输出

输出到 Decode Unit 的 vtype 信息（当前处于译码阶段向量指令使用的 vtype 配置）

7.9.8.3 设计规格

vtypeSpec 更新存在 4 种情况：

1. vsetvl 指令提交时，vtypeSpec 更新为 vsetvl 指令的 vtype，其中 vsetvl 指令的 vtype 值在其写回时获得。由于 vsetvl 指令会 flush 流水线，因此不会和其余情况冲突
2. 重定回滚的过程中，vtypeSpec 更新为 vtype buffer 传入的 walkVType
3. 重定向开始时，vtypeSpec 更新为 Arch vtype
4. 译码的指令存在 vsetivli 或 vsetvli 指令且没有发生异常时，vsetivli 指令和 vsetvli 指令的 vtype 信息可通过立即数字段获得，VTypeGen 中存在一个简单的译码器，用于判断输入的指令中是否包含这两种指令。如果存在这两种 vset 指令，会通过一个 PriorityMux 选择出第一个 vset 指令，通过 VsetModule 模块解析出 vtype 信息。

```

when(io.commitVType.hasVsetvl) {
    vtypeSpecNext := io.vsetvlVType
}.elsewhen(io.walkVType.valid) {
    vtypeSpecNext := io.walkVType.bits
}.elsewhen(io.walkToArchVType) {
    vtypeSpecNext := vtypeArch
}.elsewhen(inHasVset && io.canUpdateVType) {
    vtypeSpecNext := vtypeNew
}

```

vtypeArch 更新存在 2 种情况：1. vsetvl 指令提交时，vtypeArch 更新为 vsetvl 指令写回的 vtype 2. vsetivli 指令或 vsetvli 指令提交时，vtypeArch 更新为从 vtype buffer 传入的 vtype 提交信息

7.9.9 二级模块 DecodeUnit

7.9.9.1 输入输出

- 输入

- DecodeUnitEnqIO: 前端传入的指令流信息, 向量指令使用的 vtype、vstart 信息
- CustomCSRIO: csr 控制信号
- CSRTDecode: csr 控制信号

- 输出

- DecodeUnitDeqIO: 译码后的指令信息、是否是向量指令、指令拆分数量

7.9.9.2 功能

该模块是香山后端的译码单元, 该模块将 control flow 转换为信息更丰富的微操作, 包含源寄存器号、源寄存器类型、目标寄存器号、目标寄存器类型、立即数类型、使用的功能单元类型、操作类型等信息。

7.9.9.3 设计规格

1. 译码信息

- XSDecode

DecodeConstants 中定义了 decodeArray, 将指令的 32 位编码转化为 XSDecode, 包含以下信息:

- srcType0: 源寄存器 0 的类型
- srcType1: 源寄存器 1 的类型
- srcType2: 源寄存器 2 的类型, 用于 fma 指令
- fuType: 功能单元类型
- fuOpType: 操作类型
- rfWen: 是否写回标量寄存器
- fpWen: 是否写回浮点寄存器
- vfWen: 是否写回向量寄存器
- isXSTrap: 是否是 XSTrap 指令
- noSpecExec: 是否是可以乱序执行, 即不需要等待前面的指令提交完成再执行
- blockBackward: 是否阻塞后面的指令, 即需要等待当前指令提交完成后续指令才能进入 rob
- flushPipe: 是否需要清空流水线, 即当前指令提交完成后需要清空流水线
- canRobCompress: 指令是否支持 rob 压缩(对于不会触发异常的指令, 其不处于 FTQ 的边界时, 我们认为它是可以进行 Rob 压缩的)
- uopSplitType: 指令拆分类型。标量指令拆分类型均为 UopSplitType.SCA_SIM 无需拆分, 向量指令和 AMO_CAS 指令需要拆分; 向量指如果仅需要拆分出一条 uop 且无需对指令控制信号进行修改, 则拆分类型为 UopSplitType.dummy 从而进入向量复杂译码器进行向量指令异常检查。

- VPUCtrlSignals

向量指令和浮点指令需要设置 VPUCtrlSignals。VPUCtrlSignals 包含用于向量配置的 sew、lmul 等信息。

- 向量指令: 的向量配置信息来源于 DecodeStage 中 VtypeGen 的 vtype 信息。

- 浮点指令：浮点模块和向量模块独立，但复用了和向量相同的运算单元，运算单元通过 sew 信息指定元素的位宽，因此会通过一个专门用于浮点指令的译码子模块 FPToVecDecoder 生成浮点指令的 VPUCtrlSignals 控制信号。
- **FPUCtrlSignals**
在译码子模块 FPDecoder 生成，rm 信号用于控制浮点舍入，wflags 用于控制 i2f 模块和 fflag 更新，其余信号用于控制 i2f 模块 “ ‘scala class FPUCtrlSignals(implicit p: Parameters) extends XSBundle { val typeTagOut = UInt(2.W) // H S D val wflags = Bool() val typ = UInt(2.W) val fmt = UInt(2.W) val rm = UInt(3.W) }
“ “
· **uopnum UopInfoGen** 生成指令拆分的数量。标量指令的指令拆分数量为 1，AMO_CAS 指令根据类型拆分数量可为 2 或 4，向量指令的指令拆分数量需要根据 lmul 计算指令拆分数量，其中向量访存指令还需要根据 lmul、sew、eew 计算指令拆分数量。

2. 转译处理

- **move 指令**
由于 move 指令是一条特殊的 addi 指令，会通过指令字段识别出 move 指令，在后续 rename 阶段进行 move 消除
- **zimop 指令**
由于 zimop 指令只需要将 vd 写为 0，转译为一条 src 为 x0, imm 为 0 的 addi 指令
- **csrr vlenb 指令** vlenb 的值固定，转译为一条 src 为 x0, imm 为 VLEN/8 的 addi 指令
- **csrr vl 指令** vl 使用独立寄存器堆，因此支持重命名并乱序执行，读 vl 指令会转换为一条读 vl 写对应 rd 的 vset 指令
- **软预取指令**
将 fuType 修改为 FuType.ldu.U，传入对应的功能单元进行处理

3. 异常处理

DecodeUnit 中会处理 **illegalInstr** (异常值为 2) 和 **virtualInstr** (异常值为 22) 两种异常

- **illegalInstr**
 - 检查立即数选择是否无效
 - 指令在某些 CSR 设置下执行的异常
 - 向量相关的异常不在该模块检查，在复杂译码器中进行
- **virtualInstr**
 - 指令在某些 CSR 设置下执行的异常

7.9.9.4 二级模块 DecodeUnitComp

7.9.9.5 输入输出

指令拆分只是对指令中的操作数寄存器号、操作数类型等信息进行修改，因此输入和输出的类型都是 DecodeUnitCompInput。由于 vset 指令的 vtype 信息需要通过译码获得，而不是通过 vtypegen 获得，因此会通过 vtypebypass 信号，将 vset 指令使用的 vtype 更新为该 vset 指令的 vtype 信息。 - **DecodeUnitCompIO** “ ‘scala class DecodeUnitCompIO(implicit p: Parameters) extends XSBundle { val redirect = Input(Bool()) val csrCtrl = Input(new CustomCSRCtrlIO) val vtypeBypass = Input(new VType) // When the

```
first inst in decode vector is complex inst, pass it in val in = Flipped(DecoupledIO(new DecodeUnitCompInput)) val out = new DecodeUnitCompOutput val complexNum = Output(UInt(3.W)) }

“”
```

7.9.9.6 功能

将一条向量指令，根据拆分类型以及 lmul 信息，生成多个微操作，并对微操作中的操作数寄存器号、操作数类型等信息进行修改。同时，向量指令的异常检查也在该模块中进行。该模块使用一个状态机，仅当没用指令进行处理或拆分的指令处理完成的当拍，ready 信号才会拉高，从而处理下一条指令。

7.9.9.7 设计规格

目前指令拆分的种类较多，未来会进行精简优化

| 拆分类型 | 对应的指令类型 |
|---|-------------------------------------|
| AMO_CAS_W/AMO_CAS_D/AMO_CAS_Q | AMO_CAS 指令 |
| VSET | vset 指令 |
| VEC_VVV | 两个源寄存器和目标寄存器都是向量寄存器的指令 |
| VEC_VFV | 一个源寄存器是浮点寄存器，一个源寄存器和目标寄存器都是向量寄存器的指令 |
| VEC_EXT2/VEC_EXT4/VEC_EXT8 | 向量符号扩展指令 |
| VEC_0XV | 标量到向量的 move 指令 |
| VEC_VXV | 一个源寄存器是标量寄存器，一个源寄存器和目标寄存器都是向量寄存器的指令 |
| VEC_VVW/VEC_VFW/VEC_WVW/VEC_VXW/VEC_WXW | 向量向量向量向量向量指令 |
| VEC_VVM/VEC_VFM/VEC_VXM | 目标寄存器是 mask 寄存器的向量指令 |
| VEC_SLIDE1UP | vslide1up 指令 |
| VEC_FSLIDE1UP | vfslide1up 指令 |
| VEC_SLIDE1DOWN | vslide1down 指令 |
| VEC_FSLIDE1DOWN | vfslide1down 指令 |
| VEC_VRED | 标量 reduction 指令 |
| VEC_VFRED | 乱序浮点 reduction 指令 |
| VEC_VFREDO_SUM | 顺序浮点 reduction 指令 |
| VEC_SLIDEUP | vslideup 指令 |
| VEC_SLIDEDOWN | vslidedown 指令 |
| VEC_M0X | vcpop 指令 |
| VEC_MVV | vid/viota 指令 |
| VEC_VWW | 标量 widening reduction 指令 |
| VEC_RGATHER | vrgather 指令 |
| VEC_RGATHER_VX | 其中一个操作数来自标量寄存器的 vrgather 指令 |
| VEC_RGATHER_EI16 | vrgatherei16 指令 |
| VEC_COMPRESS | vcompress 指令 |
| VEC_MVNR | vmvnr 指令 |
| VEC_US_LDST | unit-stride load/store 指令 |

| | |
|------------|-----------------------|
| 拆分类型 | 对应的指令类型 |
| VEC_S_LDST | strided load/store 指令 |
| VEC_I_LDST | indexed load/store 指令 |

7.9.10 二级模块 VecExceptionGen

- Inputs:

- inst: 32bits 指令
- decodedInst: 译码后的信息
- vtype: vtype 信息
- vstart: vstart 信息

- Output:

- illegalInst: 指令是否异常

7.9.10.1 功能

检查向量指令是否发生异常，除向量访存指令的访存相关异常，均在译码阶段进行检查。

7.9.10.2 设计规格

将向量指令相关的异常分为了以下八种：

| 异常名称 | 描述 |
|------------------|--|
| inst Illegal | reserved 指令报异常 |
| vill Illegal | vtype 的 vill 字段为 1 时，执行 vset 以外的向量指令时报异常 |
| EEW Illegal | 向量浮点指令、符号拓展指令、widening 指令、narrow 指令 eew 异常 |
| EMUL Illegal | 向量访存指令、符号拓展指令、widening 指令、narrow 指令、vrgatherei16 指令 elmul 异常 |
| Reg Number Align | vs1、vs2、vd 未按 lmul 对齐 |
| v0 Overlap | 部分指令读 v0 寄存器同时修改 v0 时报异常 |
| Src Reg Overlap | 部分指令 vs1、vs2 和 vd 重合时报异常 |
| vstart Illegal | vstart 不等于 0 时，执行 vset 和向量访存指令以外的向量指令时报异常 |

其中一种触发异常，则将异常信号拉高

7.10 Rename 重命名

- 版本：V2R2
- 状态：OK

- 日期: 2025/01/20
- commit: xxx

Rename 模块接收来自 Decode 模块的指令译码信息，并根据译码信息为指令分配 robIdx 和物理寄存器，通过操作数查询对应的物理寄存器。同时，该模块还会根据指令译码信息、指令提交信息和来自 RenameTable 的寄存器释放信息维护 freeList 的状态，以及根据指令译码信息和指令提交信息向 RenameTable 发送写请求，以更新推测执行时的寄存器映射状态。此外，该模块还会处理来自 ROB 的重定向请求，根据重定向信息重新更新 freeList 的状态。在完成重命名后，Rename 将重命名后的指令信息发送至 Dispatch 模块。

7.10.1 基本功能

将逻辑寄存器映射到物理寄存器，为指令中的逻辑寄存器各自分配一个物理寄存器。

寄存器重命名维护重命名相关的表或指针。维护逻辑寄存器到物理寄存器的映射表，记录每一个逻辑寄存器对应的最近分配的物理寄存器号。

对于整型、浮点和向量寄存器分别维护 224 项、192 项和 128 项物理寄存器状态表，记录物理寄存器的状态，记录是否分配，通过空闲物理寄存器分配指针记录未被分配的物理寄存器。

维护一张提交的逻辑寄存器对应物理寄存器的映射表 (RenameTable, RAT)，记录提交状态的逻辑寄存器和物理寄存器的映射关系。

维护一个提交状态的空闲物理寄存器分配的指针。寄存器重命名技术消除指令之间的寄存器读后写相关 (WAR)，和写后写相关 (WAW)，当指令执行发生例外或转移指令猜测错误而取消后面的指令时可以保证现场的精确。

7.10.2 重命名输入

- 来自译码阶段输入（途中 FusionDecoder 会对 DecodeStage 吐出的指令进行宏操作融合的修改，会根据相邻指令的组合种类改变 valid, uop 等信息，而且会根据相邻指令的 ftqptr 与 ftqoffset 的不同组合修改该融合指令的提交类型 CommitType）
- 接受 RAT 的推测重名数据返回
- 指令融合信息，以及根据指令融合情况修改译码输入指令流
- ssit, waittable 信息
- Ctrlblock snapshot 控制信息及出入队指针
- rab 提交信息

7.10.3 重命名输出

- 与 rat: 写重命名信息。
- 与 dispatch: 流水输出 rename 后的 uop 信息：在 dispatch recv 有效时。
- 与 snapshot: enqdata, 允许生成快照。

7.10.4 分配整数物理寄存器

在接收到来自 Decode 模块的有效整数指令译码信息后，Rename 模块会根据 io_in_[0-5]_bits_rfWen 信号和 io_in_[0-5]_bits_ldest 信号来判断是否需要分配新的整数物理寄存器。若 rfWen 为高电平且 ldest 不为 0，则需要分配新的整数物理寄存器。如果需要分配新的整数物理寄存器，则向 intFreeList 发送分配请求，并在

当拍获得分配结果，否则，不发送分配请求。另外，Rename 模块支持整数 Move 指令消除。如果检测到译码后的指令为整数 Move 指令，亦不分配新的整数物理寄存器。

7.10.5 分配浮点或向量物理寄存器

在接收到来自 Decode 模块的有效向量浮点指令译码信息后，Rename 模块会根据 io_in_[0-5]_bits_fpWen 和 io_in_[0-5]_bits_vecWen 信号来判断是否需要分配新的向量浮点物理寄存器。若 fpWen 或 vecWen 信号为高电平，则需要分配新的浮点或向量物理寄存器。如果需要分配新的浮点或向量物理寄存器，则向 fpFreeList 或 vecFreeList 发送分配请求，并在当拍获得分配结果，否则，不发送分配请求。

7.10.6 设置源操作数的物理寄存器 (psrc)

如果 Decode 模块传来的指令译码信息中有整数寄存器或向量浮点寄存器型的源操作数，那么在通常情况下，Decode 模块会提前一拍向 RenameTable 查询逻辑寄存器对应的物理寄存器，并在一拍后于 Rename 模块得到读推测重命名表的结果，而后将结果通过 io_out_[0-5]_bits_psrc_[0-4] 传给 Dispatch 模块。作为例外情况，如果上条指令的目的操作数和本条指令的源操作数相同，则应将本条指令的 psrc 设置为上条指令的 pdest。

7.10.7 设置目的操作数的物理寄存器 (pdest)

如果 Decode 模块传来的指令译码信息中提示存在目的操作数（参见 § 7.10.4 和 § 7.10.5），那么在通常情况下，Rename 模块通过 io_out_[0-5]_bits_pdest 将新分配的物理寄存器传给 Dispatch 模块。作为例外情况，如果该指令为整数 Move 指令，则应将本条指令的 pdest 设置为本条指令的 psrc。

7.10.8 整数指令提交

当整数指令提交后，Rename 会根据 RenameTableWrapper 传来的 io_int_need_free_[0-5] 和 io_int_old_pdest_[0-5] 信息向 intFreeList 发送 free 信号，从而将相应的整数物理寄存器释放，以供新指令使用。当 io_int_need_free_[0-5] 为高电平时，则说明对应通道的 io_int_old_pdest_[0-5] 整数物理寄存器需要被释放。此外，Rename 也会将 RAB 发来的 commit 信号发给 intFreeList，以供其维护体系结构状态重命名指针。

7.10.9 浮点或向量指令提交

当 RAB 传来浮点指令提交信息后，Rename 会结合 RAB 和 RenameTableWrapper 传来的提交信息向 fpFreeList 发送 free 信号，从而释放不再被使用的向量浮点物理寄存器，以供新指令使用。若从 RAB 传来的 io_rabCommits_info_[0-5]_fp/vecWen 信号在打一拍后，与打一拍后的 io_rabCommits_isCommit 和 io_rabCommits_commitValid_[0-5] 信号（这两个信号说明当拍处于提交状态且该通道的提交信号有效，详见 § 7.11.3）均为高电平，则说明对应通道的 io_fp/vec_old_pdest_[0-5] 浮点或向量寄存器需要被释放。此外，Rename 也会将 RAB 发来的 commit 信号发给 fpFreeList，以供其维护体系结构状态重命名指针。

7.10.10 重定向

当重定向信号从 io_redirect 端口传入后，freeList 会暂停物理寄存器的分配，并会将 freeList 的物理寄存器分配指针恢复为体系结构状态或某个快照的状态。此外，Rename 模块也不会再向 RenameTable 发送写请求信号。

7.10.11 重新重命名

重定向信号传入一个周期后，Rename 模块进入重新重命名流程。重新重命名信号由 RAB 从 io_rabCommits 端口传入。重新重命名时，Rename 模块将不再向 Dispatch 输出有效的指令信号，也将不再向 RenameTable 发送写请求信号。

Rename 模块会向 intFreeList、fpFreeList 和 vecFreeList 通过各自的 io_walkReq_[0-5] 端口发送重新重命名信号，这些重新重命名信号是来自 RAB 模块的 io_rabCommits_walkValid_[0-5]，io_rabCommits_info_[0-5]_isMove，io_rabCommits_info_[0-5]_ldest 和 io_rabCommits_info_[0-5]_rf/fp/vecWen 信号。只有当 io_rabCommits_isWalk 为高电平时，io_walkReq_[0-5] 传入的信号才是有效的。

对于 intFreeList 而言，当 io_rabCommits_walkValid_[0-5] 为高电平，且对应通道的 io_rabCommits_info_[0-5]_rfWen 为高电平，io_rabCommits_info_[0-5]_ldest 不为 0，且 io_rabCommits_info_[0-5]_isMove 为低电平时，对应的 io_walkReq_[0-5] 端口会被发送有效信号，意味着需要进行重新重命名。

对于 fpFreeList 和 vecFreeList 而言，当 io_rabCommits_walkValid_[0-5] 为高电平，且对应通道的 io_rabCommits_info_[0-5]_fp/vecWen 信号为高电平时，对应的 io_walkReq_[0-5] 端口会被发送有效信号，意味着需要进行重新重命名。

7.10.12 robIdx 分配

Rename 模块负责为每条微指令分配 robIdx。该模块内部维护了一个 robIdxHead。正常情况下，Rename 模块会为 Decode 传入的译码后的指令依次分配连续的 robIdx，并将 robIdxHead 累加，但如果对应通道的 io_in_[0-5]_bits_lastUop 为低电平，或 compressUnit 传出的对应通道的 io_out_needRobFlags_[0-5] 为低电平，那么下一个通道的微指令将不会被分配 robIdx。

重定向发生的当周期，该模块会将 robIdxHead 重置为重定向的 robIdx，并在下一周期根据 io_redirect_bits_level 的值决定是否为 robIdxHead 加一。

7.10.13 决定重命名快照的生成

Rename 模块还负责决定是否生成重命名快照。重命名快照旨在发生重定向后缩短重新重命名的时间。重命名快照是分布式的，它分布在 RenameTable、RenameTable_1、RenameTable_2、intFreeList、fpFreeList、vecFreeList、Rob、Rab、CtrlBlock 等诸多模块之中，且快照存储的内容各不相同，因此需要一个模块告诉各个模块何时生成快照，而这个模块便是 Rename。对外，Rename 会将快照生成信号通过 io_out_*_bits_snapshot 传递给其他模块；对内，Rename 也会将快照生成信号传递给 intFreeList、fpFreeList 和 vecFreeList。

重命名快照的生成存在诸多限制。首先，Rename 模块内部维护了一个快照计数器 snapshotCtr，只有当这个计数器为 0 时才可以生成快照；其次，如果当前已经存在其他快照，那么本周期重命名的第一条微指令被分配的 robIdx 必须和最近一次生成的快照的 robIdx 相差 6，也就是 ROB 提交宽度以上；最后，本周期重命名的第一条微指令必须是其所属指令的第一条微指令，即 io_in_0_bits_firstUop 必须为高电平。只有满足了以上三个条件，并且被重命名的六条微指令中存在分支跳转指令时，才会生成快照。此时，那些为分支跳转指令的通道的 io_out_*_bits_snapshot 信号会被拉高，Rename 也会将快照生成信号告诉其内部的子模块。

快照计数器 snapshotCtr 是一个控制快照生成间隔的计数器。由于距离过近的快照没有意义且会对快照资源造成浪费，因此实现了这样一个计数器。snapshotCtr 初始值设为 4 倍的 RAB 提交宽度，即 $4 \times 8 = 32$ 。如果当前不存在有效的重命名快照，snapshotCtr 会被置为 0；否则，每重命名 n 条微指令，snapshotCtr 会自减 n，直到 0 为止。当 snapshotCtr 为 0 后，如果某时刻产生了重命名快照，那么 snapshotCtr 会被重置为最大值减去当周期重命名的微指令数，即 $32 - \text{PopCount}(\text{io_out_*_valid} \&\& \text{io_out_*_ready})$ 。

7.10.14 整体框图

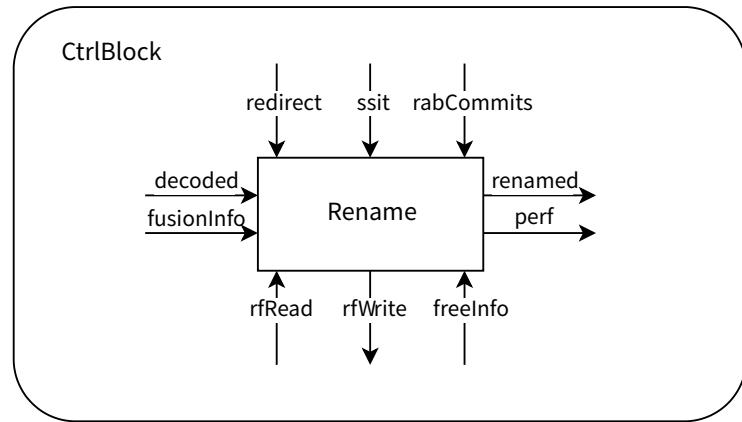


图 7.12: Rename 整体框图

7.10.15 接口时序

7.10.15.1 Decode 输入接口时序示意图

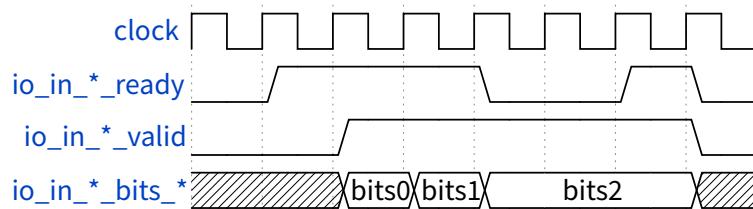


图 7.13: Decode 输入接口时序示意图

图 7.13 示意了三个来自 decode 的译码结输入例。当 ready 和 valid 信号同时为高时，对应的 bits 被 Rename 模块接收。

7.10.15.2 Rename 输出接口时序示意图

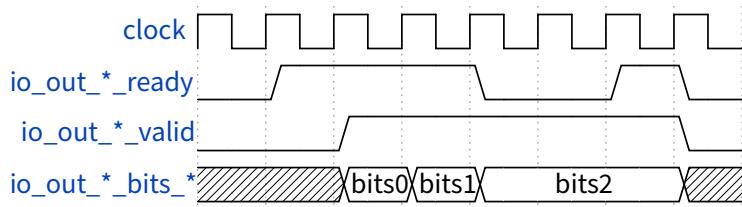


图 7.14: Rename 输出接口时序示意图

图 7.14 示意了三个重命名结果的示例。当 ready 和 valid 信号同时为高时，对应的 bits 被 Rename 模块发送至 Dispatch。

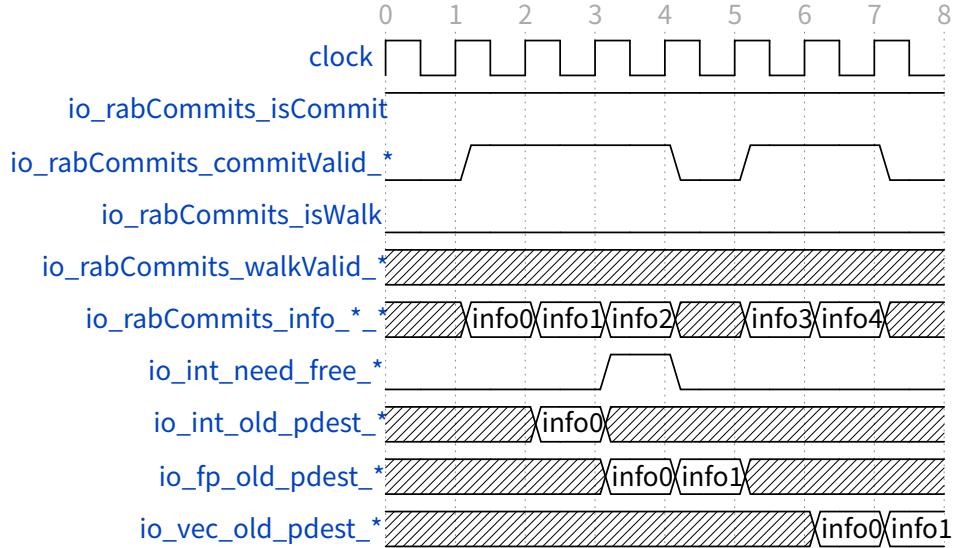


图 7.15: 指令提交逻辑时序示意图

7.10.15.3 指令提交逻辑时序示意图

图 7.15 示意了五个来自 ROB 的指令提交输入。当 `io_rabCommits_isCommit` 为高且 `io_rabCommits_isWalk` 为低时，`io_rabCommits_info_*_*` 为指令提交信息。`io_rabCommits_commitValid_*` 为高时，对应的 `io_rabCommits_info_*_*` 将有效的指令提交信息传入 Rename 模块。同时，`io_*_old_pdest_*` 会在延迟一个周期后将需要释放的旧的物理寄存器号传给 Rename 模块，并在再延迟一个周期后将是否需要释放整数物理寄存器信号通过 `io_int_need_free_*` 端口传入 Rename 模块。

7.10.15.4 重定向和重新重命名时序示意图

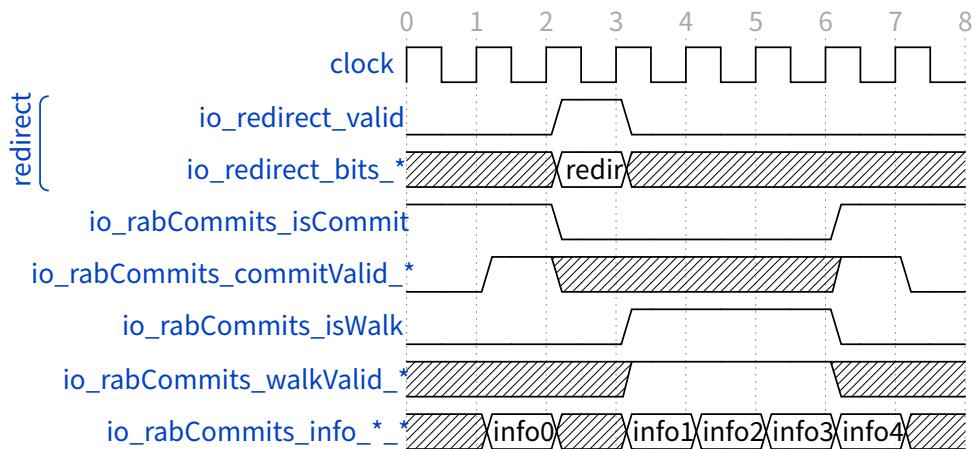


图 7.16: 重定向和重新重命名时序示意图

图 7.16 示意了重定向发生前后的有关信号。在前两个周期，`io_redirect_valid` 为低电平，Rename 处于正常工作状态，与图 7.15 相同。此后，`io_redirect_valid` 信号被拉高一个周期，重定向到来，重定向的有关信息从 `io_redirect_bits_*` 发来，Rename 将从下个周期开始进入重新重命名工作状态。此后的三个周期内，`io_rabCommits_isCommit` 为低电平，`io_rabCommits_info_*_*` 不再发来提交信息；与此相对的，`io_rabCommits_isWalk` 为高电平，标志着 `io_rabCommits_info_*_*` 发来重新重命名信息，Rename 需要进

行重新重命名工作。io_rabCommits_walkValid_* 为高时，对应的 io_rabCommits_info_*_* 发来的重新重命名信息有效。

7.11 RenameTableWrapper

RenameTableWrapper 是一个包装模块，其内部包含整数重命名表 RenameTable 模块、浮点重命名表 RenameTable_1 模块和向量重命名表 RenameTable_2 模块。这一包装模块除了简单地将三个重命名表打包以外，还在内部处理了提交和重新重命名相关的逻辑。RenameTableWrapper 起到了沟通内部重命名表与外部模块的桥梁作用。

7.11.1 读推测重命名表

RenameTableWrapper 共有 12 个整数寄存器读端口、18 个浮点寄存器读端口和 30 个向量浮点寄存器读端口，其中整数寄存器读端口 2 个为一组、浮点寄存器读端口 3 个为一组、向量寄存器读端口 5 个为一组，各自均有 6 组读端口。整数寄存器读端口用于读取整数逻辑寄存器到整数物理寄存器的推测映射关系，浮点寄存器读端口用于读取浮点逻辑寄存器到向量浮点物理寄存器的推测映射关系，向量寄存器读端口用于读取向量逻辑寄存器到向量浮点物理寄存器的推测映射关系。

RenameTableWrapper 的读是同步的。这意味着第 T 个时钟周期通过 io_(int/fp/vec)ReadPorts_*_*_addr 发送的读请求，要到第 T+1 个时钟周期才能从 io_(int/fp/vec)ReadPorts_*_*_data 得到逻辑寄存器在第 T 个时钟周期对应的物理寄存器。

RenameTableWrapper 的读是有前递的。如果第 T 个时钟周期向某个地址发送读请求的同时，也向某个地址发送写请求，那么在第 T+1 个时钟周期时，读到的是在第 T 个时钟周期时向某个地址写入的值。

RenameTableWrapper 的读是有保持功能的。如果第 T 个时钟周期，某个读端口的 io_(int/fp/vec)ReadPorts_*_*_hold 为高电平，那么在第 T+1 个时钟周期时，读到的值和第 T 个时钟周期时读到的值相同。

7.11.2 重命名阶段写推测重命名表

RenameTableWrapper 共有 6 个整数寄存器写端口、6 个浮点寄存器写端口和 6 个向量寄存器写端口，这些端口用于在重命名阶段写推测重命名表。整数寄存器写端口用于在重命名阶段更新整数逻辑寄存器到整数物理寄存器的推测映射关系，浮点寄存器写端口用于在重命名阶段更新浮点逻辑寄存器到向量浮点物理寄存器的推测映射关系，向量寄存器写端口用于在重命名阶段更新向量逻辑寄存器到向量浮点物理寄存器的推测映射关系。

RenameTableWrapper 的写是同步的。这意味着第 T 个时钟周期通过 io_(int/fp/vec)RenamePorts_*_addr 和 io_(int/fp/vec)RenamePorts_*_data 发送的写请求，要到第 T+1 个时钟周期才能被读出。

RenameTableWrapper 的写是有使能的。只有 io_(int/fp/vec)RenamePorts_*_wen 为高的写请求才是有效的。

RenameTableWrapper 的写是有优先级的。写通道的编号越大，优先级越大，这意味着如果两个通道向同一个地址写，那么最终写入的结果将是编号大的那个。

7.11.3 提交阶段写体系结构重命名表

RenameTableWrapper 通过监听来自 RAB 的 commit 信息来更新体系机构重命名表。如果某个周期的 io_rabCommits_isCommit 信号为高电平，则说明该周期正在提交。此时，若某个 io_rabCommits_commitValid_* 信号为高电平，则说明该端口的提交信号有效。此时，需要进一步考察 io_rabCommits_info_*_rfWen、io_rabCommits_info_*_fpWen 和 io_rabCommits_info_*_vecWen。如果 io_rabCommits_info_*_rfWen

为高电平，则说明整数寄存器需要更新体系结构重命名表；如果 io_rabCommits_info_*_fpWen 为高电平，则说明浮点寄存器需要更新体系结构重命名表；如果 io_rabCommits_info_*_vecWen 为高电平，则说明向量寄存器需要更新体系结构重命名表。在以上三种情况下，RenameTableWrapper 将会把整数、浮点或向量体系结构重命名表中地址为 io_rabCommits_info_*_ldest 的项修改为 io_rabCommits_info_*_pdest。

7.11.4 提交阶段提供物理寄存器释放信息

RenameTableWrapper 通过提交阶段写体系结构重命名表的情况来提供物理寄存器的释放信息。这些信息包括要释放的整数物理寄存器号 io_int_old_pdest_* 和对应的有效信号 io_int_need_free_*, 以及要释放的向量浮点物理寄存器号 io_(fp/vec)_old_pdest_*。这些信号直接来自于 RenameTableWrapper 的子模块，并会在 Rename 模块根据结合指令提交的情况进行物理寄存器的释放。

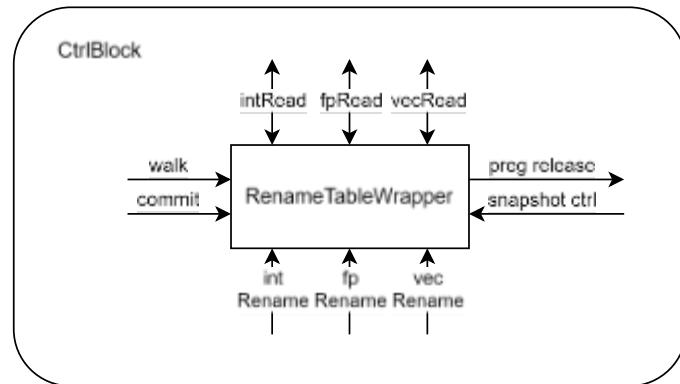
7.11.5 重新重命名阶段写推测重命名表

RenameTableWrapper 通过监听来自 RAB 的 commit 信息来进行重新重命名。如果某个周期的 io_rabCommits_isWalk 信号为高电平，则说明该周期正在重新重命名。此时，若某个 io_rabCommits_walkValid_* 信号为高电平，则说明该端口的重新重命名信号有效。此时，需要进一步考察 io_rabCommits_info_*_rfWen、io_rabCommits_info_*_fpWen 和 io_rabCommits_info_*_vecWen。如果 io_rabCommits_info_*_rfWen 为高电平，则说明整数寄存器需要重新重命名；如果 io_rabCommits_info_*_fpWen 为高电平，则说明浮点寄存器需要重新重命名；如果 io_rabCommits_info_*_vecWen 为高电平，则说明向量寄存器需要重新重命名。在以上三种情况下，RenameTableWrapper 将会把整数、浮点或向量推测重命名表中地址为 io_rabCommits_info_*_ldest 的项修改为 io_rabCommits_info_*_pdest。

7.11.6 重命名快照的维护

RenameTableWrapper 会将来自外部的重命名快照信号 io_snpt_* 传给各个子模块，用于重命名快照的生成、释放、冲刷和使用。

7.11.7 整体框图



7.11.8 接口时序

7.11.8.1 整数读写接口时序示意图（浮点向量同理）

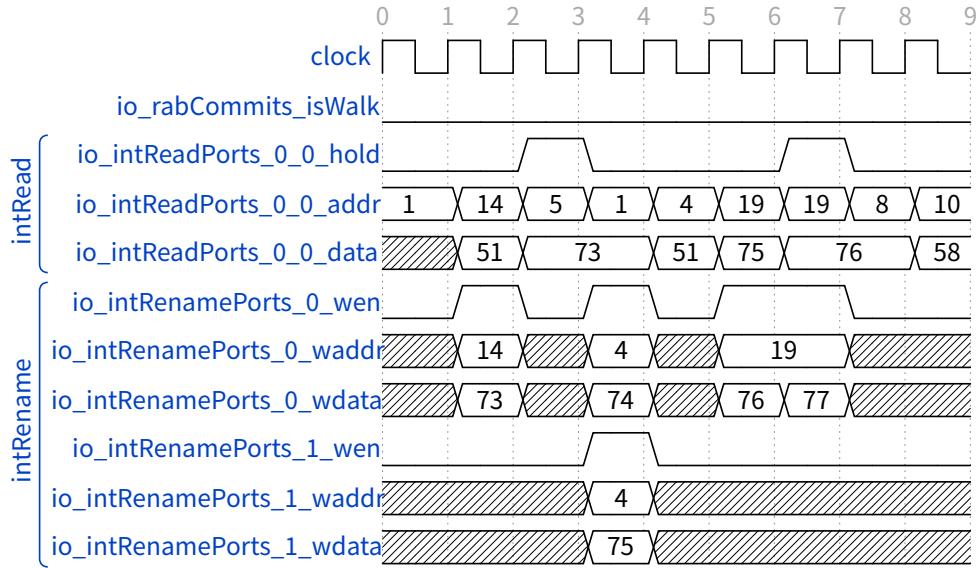


图 7.18: 整数读写接口时序示意图

图 7.18 示意了整数读写的接口时序。

在时刻 2, `io_intRenamePorts_0` 向地址为 14 处写入了 73。同时, `io_intReadPorts_0_0` 也向地址为 14 处发起了读请求, 因此在时刻 3 读到了时刻 2 写入的 73。

在时刻 4, `io_intRenamePorts_0` 向地址为 4 处写入了 74, `io_intRenamePorts_1` 也向地址为 4 处写入了 75。从而, 当 `io_intReadPorts_0_0` 于时刻 5 向地址为 4 处发出读请求后, 时刻 6 读到 `io_intRenamePorts_1` 写入的 75。

在时刻 3 和时刻 7, `io_intReadPorts_0_0_hold` 为高电平, 因此时刻 4 读出的值与时刻 3 读出的值相同, 为 73, 而不是地址为 5 处的值; 类似的, 时刻 8 读出的值与时刻 7 读出的值相同, 为 76, 而非于时刻 7 新写入的值 77。

7.11.8.2 重新重命名和提交接口时序示意图

图 7.19 示意了两个重新重命名和提交接口的时序。

时刻 1 至时刻 4, `io_rabCommits_isWalk` 信号为高, `io_rabCommits_ioCommit` 信号为低, 此时处于重新重命名状态。在时刻 2, `io_rabCommits_walkValid_0` 为高, `io_rabCommits_info_0_rfWen` 为低, `io_rabCommits_info_0_fpWen` 为高, 重新重命名接口 0 于是将 37 写入浮点推测重命名表的地址 0 处。在时刻 3, 两个重新重命名接口均向 12 号逻辑整数寄存器写入了值。此时, 1 号接口比 0 号接口优先级更高, 于是 57 将被实际写入整数推测重命名表的地址 12 处。

时刻 5 至时刻 9, `io_rabCommits_isWalk` 信号为低, `io_rabCommits_ioCommit` 信号为高, 此时处于提交状态。在时刻 7, `io_rabCommits_commitValid_0` 为高, `io_rabCommits_info_0_rfWen` 为低, `io_rabCommits_info_0_fpWen` 为高, 提交接口 0 于是将 92 写入浮点体系结构重命名表的地址 18 处。

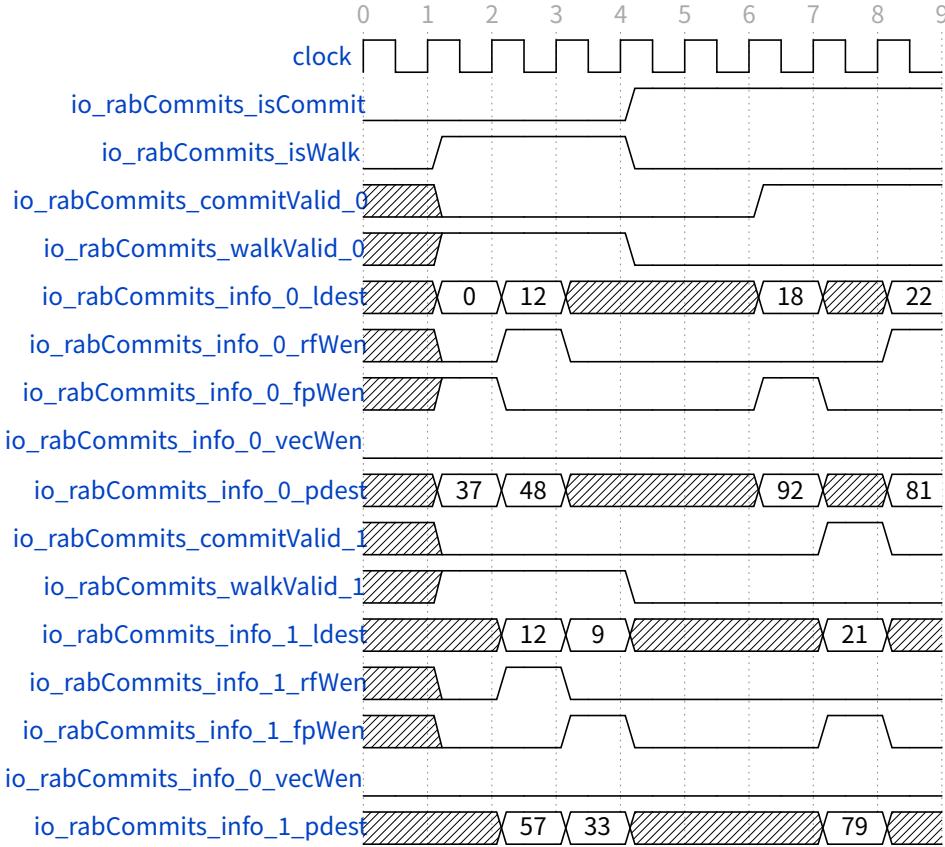


图 7.19: 重新重命名和提交接口时序示意图

7.12 支持 move 消除的 RenameTable

支持 move 消除的 RenameTable 被用于整数寄存器的重命名表，模块名为 `RenameTable`，其中维护了逻辑整数寄存器与物理整数寄存器的映射关系。其有 12 个读推测重命名表端口、6 个写推测重命名表端口和 6 个写体系结构重命名表端口，内部则由 32 个宽度为 8 的寄存器来实际维护映射关系。读端口和写端口的行为与 `RenameTableWrapper` 中所描述的行为完全一致。需要注意的是，为了时序考虑，模块 T0 时刻的写推测重命名表请求会实际于 T1 时刻处理，而 T0 时刻的写推测重命名表数据会被旁路到 T1 时刻的读推测重命名表结果。

其次，模块内部还有 4 份推测重命名表快照用于重定向和重新重命名时的快速恢复。这些快照存储在子模块 `SnapShotGenerator_3` 中，在 `RenameTable` 的名字为 `_snapshots_snapshotGen_io_snapshots_0/1/2/3_[0-31]`。快照的生成、释放、使用和冲刷完全由外部信号 `io_redirect` 和 `io_snpt_*` 进行控制。

外部传入的重定向信号 `io_redirect` 和快照控制信号 `io_snpt_*` 在打一拍后成为 `t1_redirect` 和 `t1_snpt_*` 信号。重定向信号 `t1_redirect` 为高电平时，会检查 `t1_snpt_useSnpt` 信号是否为高电平。如果 `t1_snpt_useSnpt` 信号为低电平，则会将推测重命名表置为体系结构重命名表；如果 `t1_snpt_useSnpt` 为高电平，则会将推测重命名表置为 `_snapshots_snapshotGen_io_snapshots_[t1_snpt_snptSelect]_[0-31]`。

此外，模块还会根据写体系结构重命名表端口和内部的体系结构重命名表输出物理寄存器释放信号。如果一个写体系结构重命名表通道的写使能信号 `io_archWritePorts_n_wen` 为低电平，则下一拍的 `io_old_pdest_n` 则为 0；如果写使能信号不为零，则下一拍的 `io_old_pdest_n` 为当拍的 `arch_table[io_archWritePorts_n_addr]`。需要额外注意的是，`io_old_pdest_n` 是有旁路的。对于 $n > 0$ 的情况，如果存在序号比 n 小的某个通道向体系结构重命名表的同一个逻辑寄存器写入了某个值，则下一拍的 `io_old_pdest_n` 则应该置为这个值，而非 `arch_table[io_archWritePorts_n_addr]`。举例来说，对于 $0 < j < n$ ，如果 `io_archWritePorts_n_wen` 和

`io_archWritePorts_j_wen` 都为高电平, 且 `io_archWritePorts_n_addr == io_archWritePorts_j_addr`, 那么下一拍的 `io_old_pdest_n` 则应该置为 `io_archWritePorts_j_data`, 而非 `arch_table[io_archWritePorts_n_addr]`。此外, 对于 $n > 1$ 的情况, 如果存在多个序号比 n 小的某个通道向体系结构重命名表的同一个逻辑寄存器写入了某个值, 则下一拍的 `io_old_pdest_n` 则应该置为这些通道中序号较大者对应的写入值。举例来说, 对于 $0 \leq j < k \leq n$, 如果 `io_archWritePorts_n_wen`、`io_archWritePorts_j_wen` 和 `io_archWritePorts_k_wen` 都为高电平, 且 `io_archWritePorts_n_addr == io_archWritePorts_j_addr == io_archWritePorts_k_addr`, 那么下一拍的 `io_old_pdest_n` 则应该置为 `io_archWritePorts_k_data`, 而非 `arch_table[io_archWritePorts_n_addr]` 或 `io_archWritePorts_j_data`。

物理寄存器释放信号还包括 `io_need_free_*` 信号。如果当拍某个通道的 `io_old_pdest_n` 信号和 `arch_table_*` 中的任何一项都不同, 则将该通道下一拍的 `io_need_free_n` 信号置高。需要额外注意的是, 对于 $n > 0$ 的情况, 如果存在序号比 n 小的某个通道的 `io_old_pdest_j` 信号和 `io_old_pdest_n` 相同, 那么下一拍 `io_need_free_n` 信号不会被置高。

7.12.1 整体框图

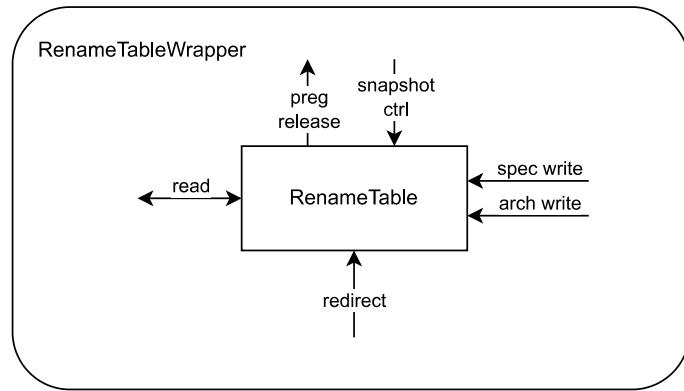


图 7.20: RenameTable 整体框图

7.12.2 接口时序

7.12.2.1 读写接口时序示意图

7.13 不支持 move 消除的 RenameTable

不支持 move 消除的 `RenameTable` 和 § 7.12 基本类似, 但并不包括 `io_need_free_*` 信号。浮点寄存器的重命名表 `RenameTable_1` 和向量寄存器的重命名表 `RenameTable_2` 使用了这种重命名表。

浮点寄存器的重命名表 `RenameTable_1` 维护了逻辑浮点寄存器与物理向量浮点寄存器的映射关系。其有 18 个读推测重命名表端口、6 个写推测重命名表端口和 6 个写体系结构重命名表端口, 内部则由 34 个宽度为 8 的寄存器来实际维护映射关系。

浮点寄存器的重命名表 `RenameTable_2` 维护了逻辑向量寄存器与物理向量浮点寄存器的映射关系。其有 30 个读推测重命名表端口、6 个写推测重命名表端口和 6 个写体系结构重命名表端口, 内部则由 48 个宽度为 8 的寄存器来实际维护映射关系。

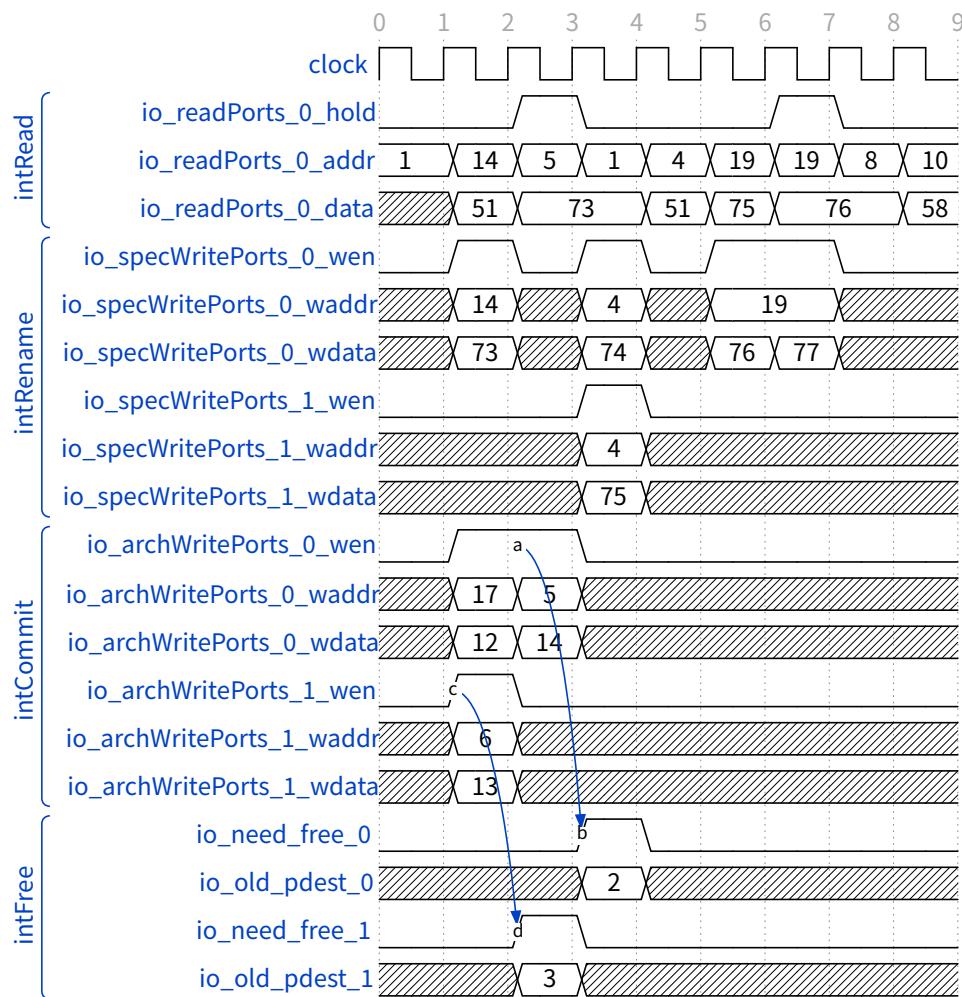


图 7.21: 支持 move 消除的 RenameTable 读写接口时序示意图

7.13.1 接口时序

7.13.1.1 读写接口时序示意图

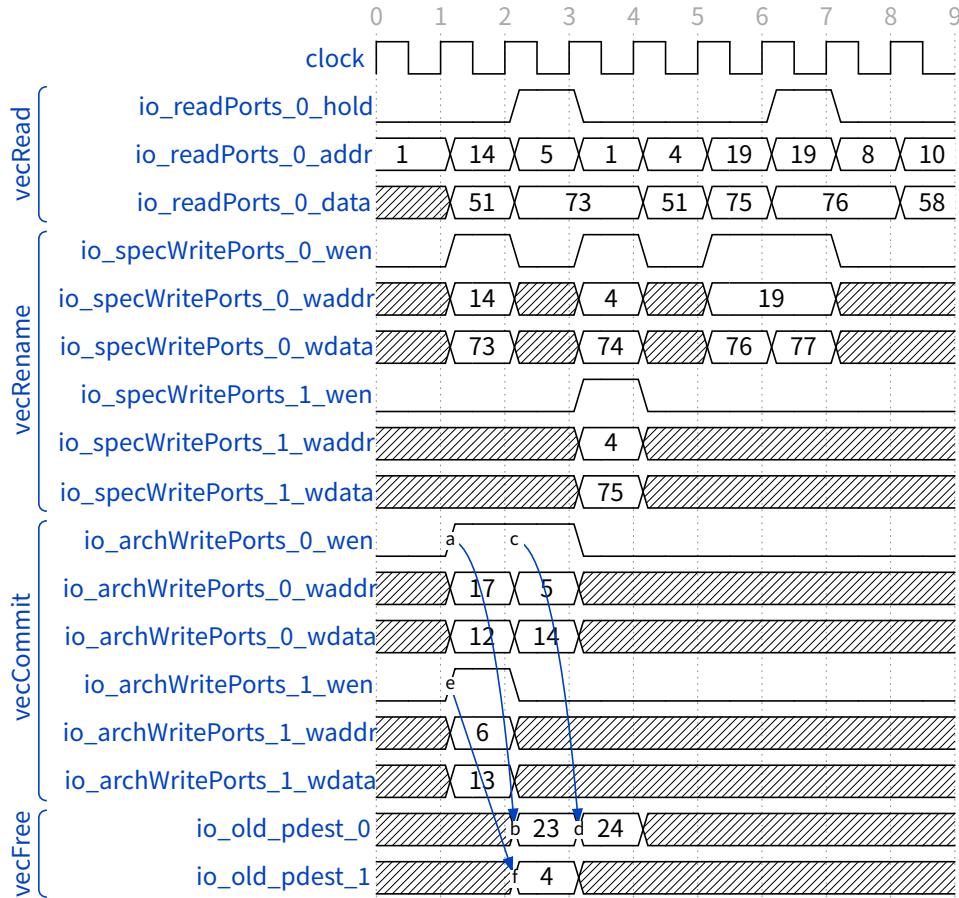


图 7.22: 不支持 move 消除的 RenameTable 读写接口时序示意图

7.14 StdFreeList

StdFreeList 在 Rename 模块中被例化为 fpFreeList 和 vecFreeList。正如 § 7.10.5、§ 7.10.9 和 § 7.10.11 中所提到的那样，fpFreelist 在重命名时负责接收向量浮点物理寄存器的分配请求，返回分配的空闲向量浮点物理寄存器；在重新重命名时负责根据 RAB 传来的重新重命名请求重新对向量浮点物理寄存器进行分配；在提交时负责释放已经不会再被使用的向量浮点物理寄存器和更新体系结构出队指针。

7.14.1 整体框图

7.14.2 接口时序

7.14.2.1 空闲寄存器分配时序示意图

图 7.24 示意了空闲物理寄存器分配的时序。在时刻 3、时刻 5 和时刻 6，io_redirect 和 io_walk 为低电平，io_doAllocate 和 io_canAllocate 为高电平，从而进行了空闲物理寄存器的分配。在时刻 3，io_allocateReq_[2-4] 为高电平，StdFreeList 通过 io_allocatePhyReg_[2-4] 分别返回了分配的空闲物理寄存器号 151、112 和 143；

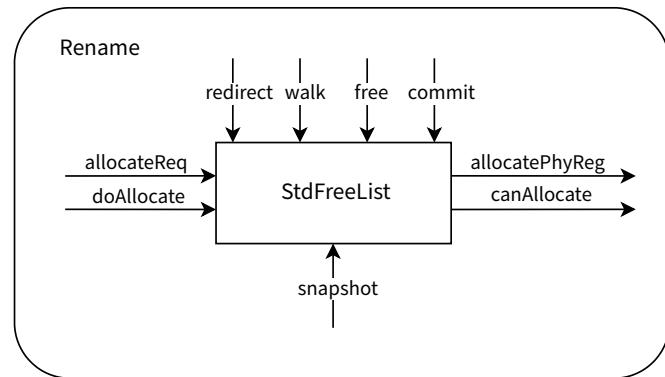


图 7.23: StdFreeList 整体框图

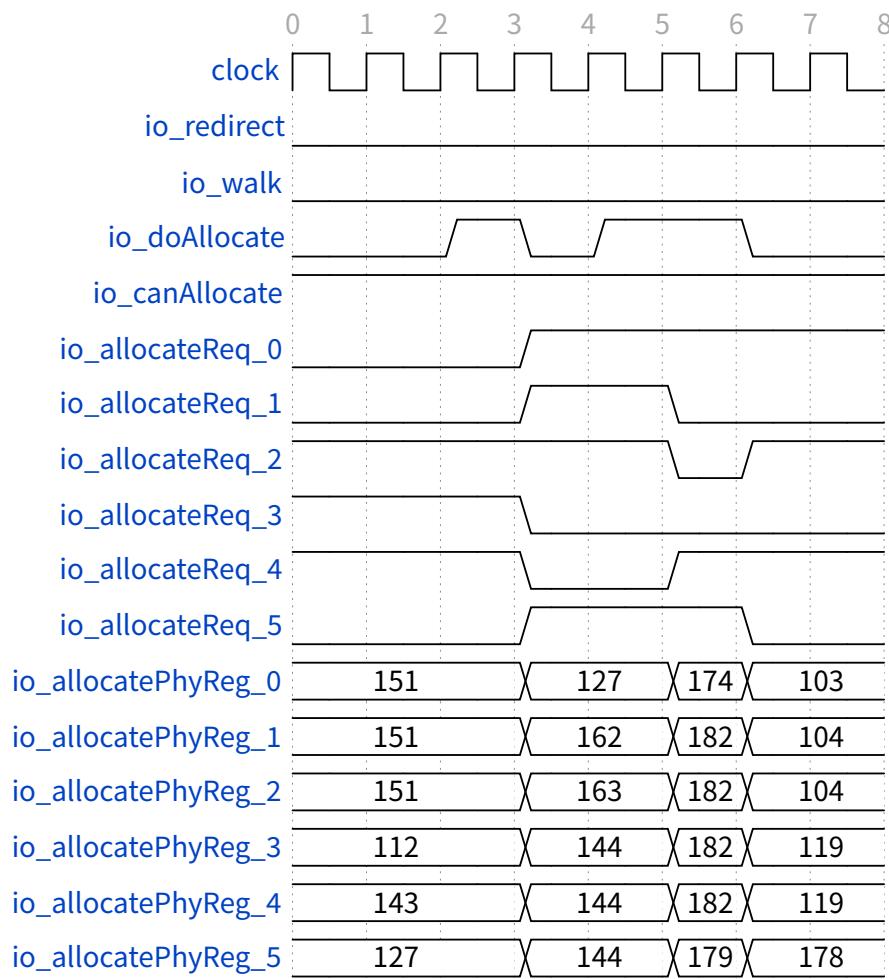


图 7.24: StdFreeList 空闲寄存器分配时序示意图

在时刻 5, `io_allocatePhyReg_[0-2|5]` 分别返回了分配的空闲物理寄存器号 127、162、163 和 144; 在时刻 6, 则返回了 174、182 和 179。每成功分配 n 个空闲物理寄存器, 模块内部的 `headPtr` 便会加 n 。

7.14.2.2 指令提交时序示意图

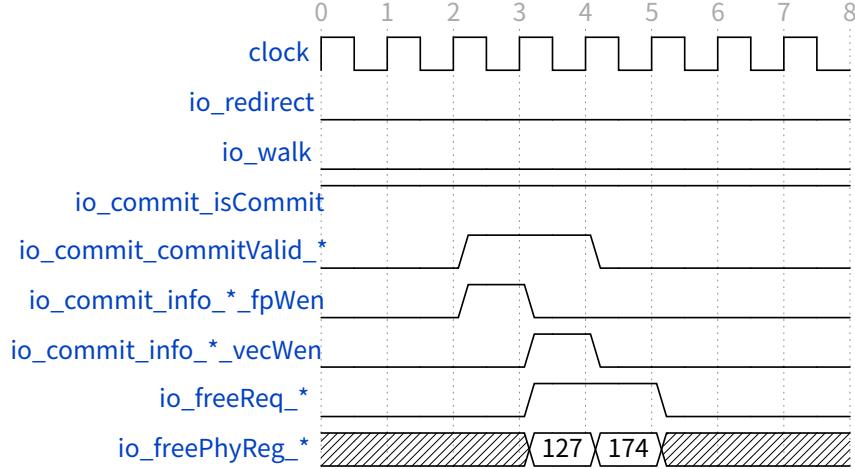


图 7.25: StdFreeList 指令提交时序示意图

图 7.25 示意了指令提交的时序, 其中 `io_freeReq_*` 表示某一路的 `io_freeReq` 信号, `io_freePhyReg_*` 表示与 `io_freeReq_*` 对应某路的 `io_freePhyReg` 信号。当 `io_redirect` 和 `io_walk` 均为低电平时, 若 `io_freeReq` 为高电平, 则 StdFreeList 会将对应的 `io_freePhyReg` 加入空闲队列。

此外, 在 `io_freeReq_*` 的前一个周期, Rename 模块还会将 RAB 的提交信息传入, 以更新体系结构出队指针 `archHeadPtr`。当 `io_commit_isCommit` 和对应通道的 `io_commit_commitValid_*` 信号处于高电平时, 说明对应通道的更新信号有效。此时, 若对应通道的 `io_commit_info_*_fpWen` 或 `io_commit_info_*_vecWen` 为高电平, 则表明该通道会让 `archHeadPtr` 加一。若有 k 个通道满足上述条件, 则 `archHeadPtr` 会加 k 。

7.14.2.3 指令重新重命名时序示意图

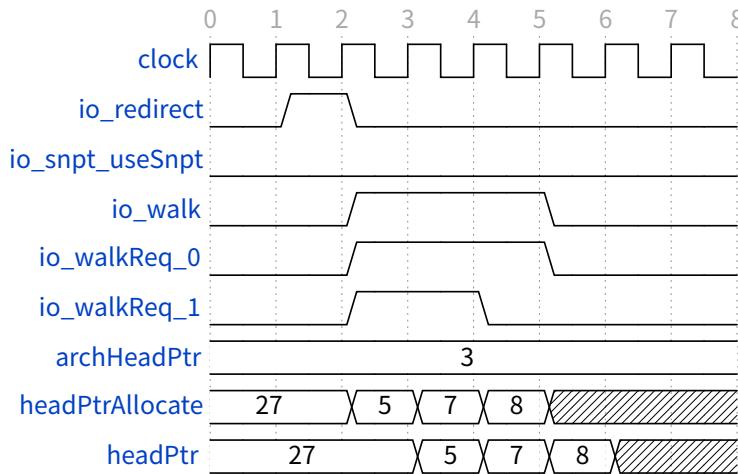


图 7.26: StdFreeList 指令重新重命名时序示意图

图 7.26 示意了指令重新重命名的时序。当 `io_redirect` 在时刻 1 拉高一个周期后, `io_walk` 会被拉高数个周期, 标志着模块进入重新重命名阶段。在时刻 1, 由于 `io_snpt_useSnpt` 为低电平, 因此 `headPtr` 会被恢

复为 archHeadPtr 的值。这一恢复不会立即进行，而是在时刻 2 加上 io_walkReq_* 高电平的数量（2）后得到 headPtrAllocate（5），并在时刻 3 写入 headPtr。此后，当 io_walk 为高电平时，headPtrAllocate 置为 headPtr+PopCount(io_walkReq_*) 的值，并在下一个周期写入 headPtr。

重新重命名过程旨在消除推测执行错误路径上的重命名状态。通过先将 headPtr 恢复到体系结构 archHeadPtr 状态（或当 io_snpt_useSnpt 为高电平时，恢复到快照状态），然后再重新重命名到进入错误路径之前，以达到这一目的。

7.14.3 关键电路：环形队列

空闲物理寄存器由一个环形队列维护。这个环形队列由寄存器组 freeList（即代码中的 freeList_*, 记其数量为 size），以及头指针 headPtr（即代码中的 headPtr_*) 和尾指针 tailPtr（即代码中的 tailPtr_*) 组成。其中 headPtr 为出队指针，tailPtr 为入队指针。

为方便说明，先考虑一个普通的队列，此时 headPtr 和 tailPtr 都是指向 freeList 中某个元素的指针。正常工作时，tailPtr 永远大于或等于 headPtr，队列中的元素为 {headPtr, headPtr + 1, ..., tailPtr - 1}。当有元素希望入队时，则将元素放置于 freeList[tailPtr]，再将 tailPtr 加一；当有元素出队时，则取出 freeList[headPtr]，再将 headPtr 加一。当 tailPtr 与 headPtr 相等时，说明队列为空；当 tailPtr 大于 headPtr 时，说明队列非空。

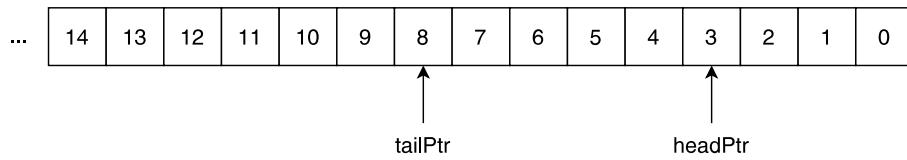


图 7.27: 普通队列

但是，由于 freeList 不可能无限长，我们设计出了环形队列。环形队列可以认为是将有限长的普通队列首尾相接。此时，tailPtr 和 headPtr 将不能仅仅只是指向 freeList 中某个元素的指针了：按原有设计，当 tailPtr 与 headPtr 相等时，环形队列可能是空的，也可能是满的。

为了解决这一问题，我们为 tailPtr 和 headPtr 新增了 flag 字段，该字段初始值为 false，且在每次由 freeList[size - 1] 变为 freeList[0] 时，将 flag 取反。这样，在 value 相同的情况下，若 flag 相同，则说明环形队列为空；若 flag 不同，则说明环形队列为满。

canAllocate 的更新时序：当拍会根据 headPtr, tailPtr, freeReq 以及 allocateReq 计算 freeRegCnt，然后将其打一拍得到 freeRegCntReg（它就是实际上的 size）。freeRegCntReg 大于译码宽度时 canAllocate 置高，并当拍传出。

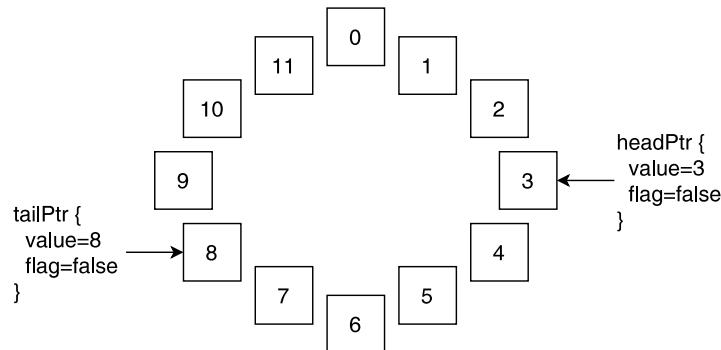


图 7.28: 环形队列

7.15 MEFreeList

MEFreeList 在 Rename 模块中被例化为 intFreeList。正如 § 7.10.4、§ 7.10.8 和 § 7.10.11 中所提到的那样，intFreeList 在重命名时负责接收整数物理寄存器的分配请求，返回分配的空闲整数物理寄存器；在重新重命名时负责根据 RAB 传来的重新重命名请求重新对整数物理寄存器进行分配；在提交时负责释放已经不会再被使用的整数物理寄存器。与 StdFreeList 不同的是，MEFreeList 支持 move 指令消除。如果一条指令是 move 指令，Rename 并不会将 io_allocateReq_*_valid 置高，因此 MEFreeList 并不会为其分配空闲物理寄存器。

7.15.1 整体框图

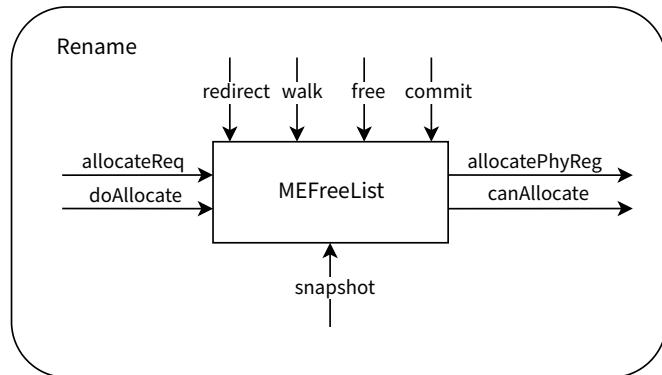


图 7.29: MEFreeList 整体框图

7.15.2 接口时序

7.15.2.1 空闲寄存器分配时序示意图

图 7.30 示意了空闲物理寄存器分配的时序。在时刻 3、时刻 5 和时刻 6，io_redirect 和 io_walk 为低电平，io_doAllocate 和 io_canAllocate 为高电平，从而进行了空闲物理寄存器的分配。在时刻 3，io_allocateReq_[2-4] 为高电平，MEFreeList 通过 io_allocatePhyReg_[2-4] 分别返回了分配的空闲物理寄存器号 151、112 和 143；在时刻 5，io_allocatePhyReg_[0-2|5] 分别返回了分配的空闲物理寄存器号 127、162、163 和 144；在时刻 6，则返回了 174、182 和 179。

7.15.2.2 指令提交时序示意图

图 7.31 示意了指令提交的时序，其中 io_freeReq_* 表示某一路的 io_freeReq 信号，io_freePhyReg_* 表示与 io_freeReq_* 对应某路的 io_freePhyReg 信号。当 io_redirect 和 io_walk 均为低电平时，若 io_freeReq 为高电平，则 StdFreeList 会将对应的 io_freePhyReg 加入空闲队列。

此外，在 io_freeReq_* 的前两个周期，Rename 模块还会将 RAB 的提交信息传入，以更新体系结构出队指针 archHeadPtr。当 io_commit_isCommit 和对应通道的 io_commit_commitValid_* 信号处于高电平时，说明对应通道的更新信号有效。此时，若对应通道的 io_commit_info_*_rfWen 为高电平、io_commit_info_*_ldest 不为 0 且 io_commit_info_*_isMove 为低电平，则表明该通道会让 archHeadPtr 加一。若有 k 个通道满足上述条件，则 archHeadPtr 会加 k。

当 io_commit_commitValid_* 在某个周期为高电平时，io_freeReq_* 信号不总会在两个周期后为高电平。造成这一现象的原因是 move 消除。此处的 io_freeReq_* 来自于 RenameTable 的输出信号 io_need_free。正

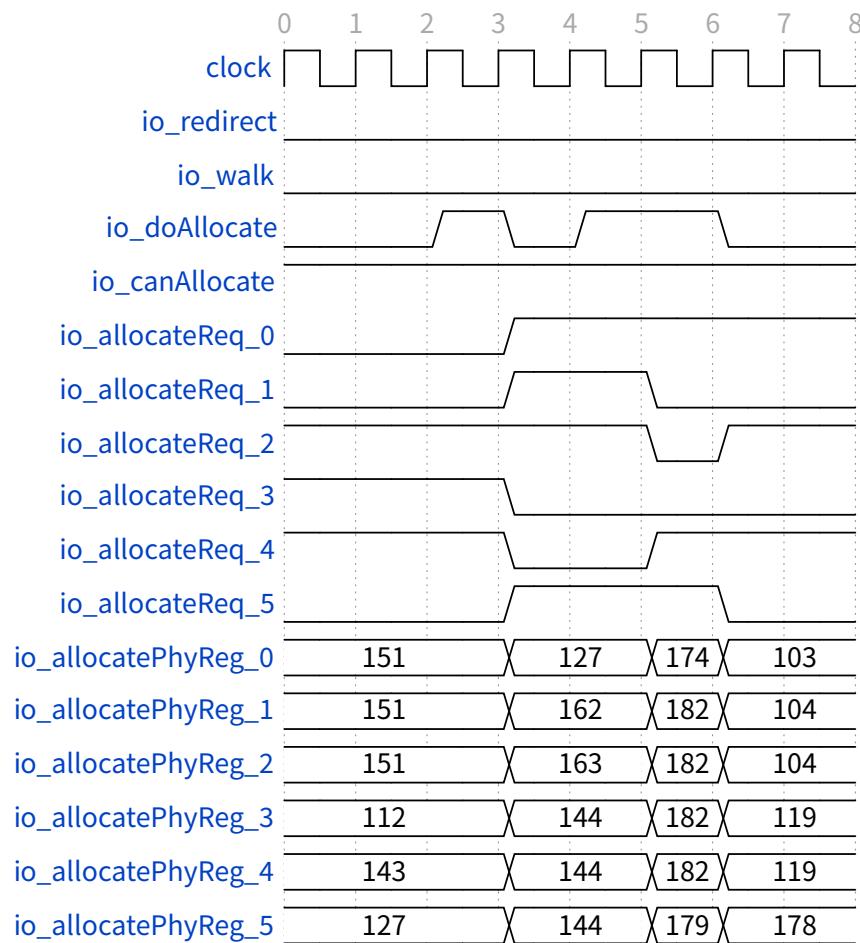


图 7.30: MEFreeList 空闲寄存器分配时序示意图

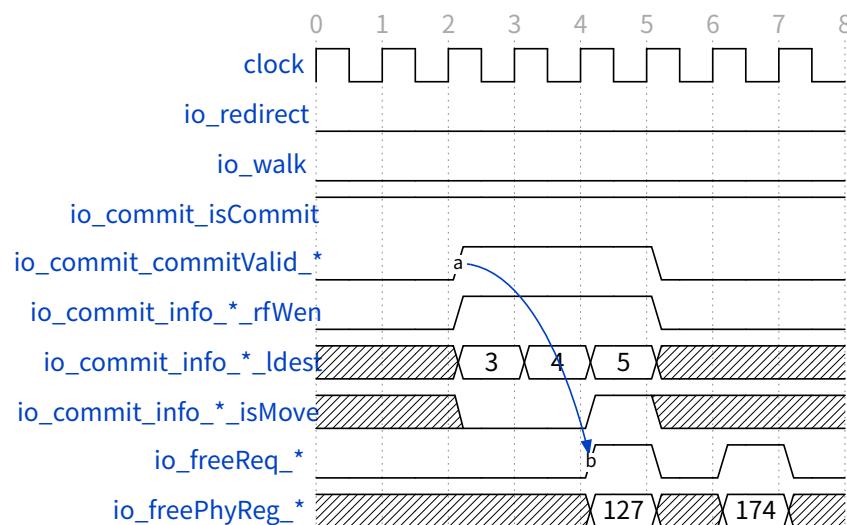


图 7.31: MEFreeList 指令提交时序示意图

如 RenameTable 模块中提到的那样，RenameTable 的 io_need_free 信号在 arch_table 中存在相同的物理寄存器时，可能不会被拉高，而正是因为 move 消除导致不同的逻辑寄存器共享了同一个物理寄存器，才导致在 RenameTable 的不同项中存在相同的物理寄存器。

7.15.2.3 指令重新重命名时序示意图

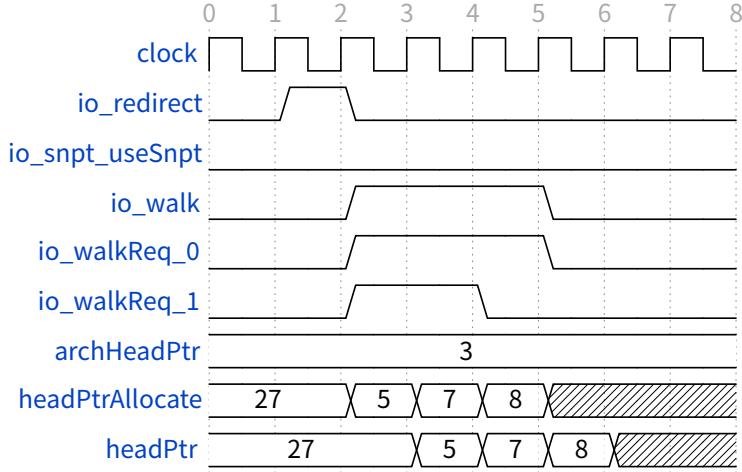


图 7.32: MEFreeList 指令重新重命名时序示意图

图 7.32 示意了指令重新重命名的时序。当 io_redirect 在时刻 1 拉高一个周期后，io_walk 会被拉高数个周期，标志着模块进入重新重命名阶段。在时刻 1，由于 io_snpt_useSnpt 为低电平，因此 headPtr 会被恢复为 archHeadPtr 的值。这一恢复不会立即进行，而是在时刻 2 加上 io_walkReq_* 高电平的数量（2）后得到 headPtrAllocate（5），并在时刻 3 写入 headPtr。此后，当 io_walk 为高电平时，headPtrAllocate 置为 headPtr+PopCount(io_walkReq_*) 的值，并在下一个周期写入 headPtr。

重新重命名过程旨在消除推测执行错误路径上的重命名状态。通过先将 headPtr 恢复到体系结构 archHeadPtr 状态（或当 io_snpt_useSnpt 为高电平时，恢复到快照状态），然后再重新重命名到进入错误路径之前，以达到这一目的。

7.16 CompressUnit

CompressUnit 用于决定哪些指令可以共用同一个 ROB 项，即可以被压缩到同一个 ROB 项。该模块接受来自译码单元的输出，并根据译码输出的结果，得到 ROB 压缩信息。

一个通道被标记为可被 ROB 压缩 (canCompress_[0-5])，当且仅当这条通道传入的译码信息满足：该通道的译码信息有效 (io_in_[0-5]_valid)，且该通道不存在指令融合 (!io_in_[0-5]_bits_commitType[2])，且该通道不存在指令拆分或为指令拆分的最后一条微指令 (io_in_[0-5]_bits_lastUop)，且该通道不存在例外 (io_in_[0-5]_bits_exceptionVec_* 均为低电平)，且该通道被标记为可被 ROB 压缩 (io_in_[0-5]_bits_canRobCompress)。

CompressUnit 会为每个通道输出是否需要分配 ROB 项的标志 io_out_needRobFlags_[0-5]。当且仅当某个通道的 canCompress_[0-5] 为 0，或该通道为自己所在的连续为 1 的 canCompress_[0-5] 组中编号最大的那一个通道时，该通道的 io_out_needRobFlags_[0-5] 会被置为高电平。

CompressUnit 会为每个通道输出该通道所在 ROB 项中的指令数量 io_out_instrSizes_[0-5]。当某个通道的 canCompress_[0-5] 为 0 时，该通道的 io_out_instrSizes_[0-5] 为 1；当某个通道的 canCompress_[0-5]

为 1 时，该通道的 io_out_instrSizes_[0-5] 为自己所在的连续为 1 的 canCompress_[0-5] 组中元素的个数。

CompressUnit 会为每个通道输出与该通道共用同一个 ROB 项的通道掩码 io_out_masks_[0-5]。该信号位宽为 6，与通道数相同。当某个通道的 canCompress_n 为 0 时，该通道的 io_out_masks_n[n] 为 1，除 io_out_masks_n[n] 以外的位为 0；当某个通道的 canCompress_n 为 1 时，io_out_masks_n 中为 1 的那些位的索引为“自己所在的连续为 1 的 canCompress_[0-5] 组”中的通道的编号。

举例来说，如果 {canCompress_5, canCompress_4, canCompress_3, canCompress_2, canCompress_1, canCompress_0} == {1, 0, 0, 1, 1, 0}，那么 {io_out_needRobFlags_5, io_out_needRobFlags_4, io_out_needRobFlags_3, io_out_needRobFlags_2, io_out_needRobFlags_1, io_out_needRobFlags_0} == {1, 1, 1, 1, 0, 1}，{io_out_instrSizes_5, io_out_instrSizes_4, io_out_instrSizes_3, io_out_instrSizes_2, io_out_instrSizes_1, io_out_instrSizes_0} == {1, 1, 1, 2, 2, 1}，{io_out_masks_5, io_out_masks_4, io_out_masks_3, io_out_masks_2, io_out_masks_1, io_out_masks_0} == {{1, 0, 0, 0, 0, 0}, {0, 1, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 0}, {0, 0, 0, 1, 1, 0}, {0, 0, 0, 1, 1, 0}, {0, 0, 0, 0, 0, 1}}。

7.16.1 整体框图

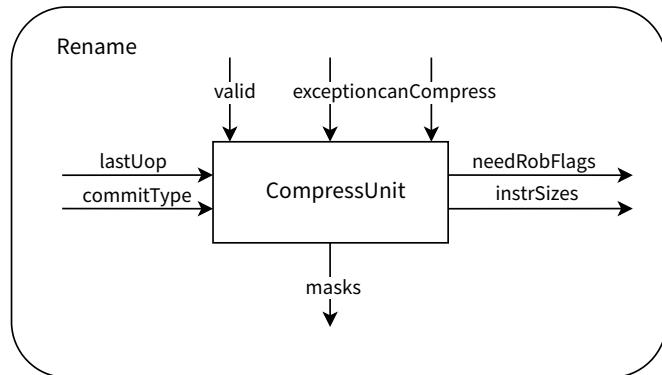


图 7.33: CompressUnit 整体框图

7.16.2 接口时序

该模块为纯组合逻辑，信号当拍进当拍出。

7.17 SnapshotGenerator

正如在 § 7.10.13 中提到的那样，重命名快照是分布式的，它会分布在各个当发生重定向后需要排除错误重命名路径影响的模块之中，以达到加快重新重命名的目的。对于和重命名相关的模块而言，RenameTable、RenameTable_1、RenameTable_2、StdFreeList 和 MEFreeList 中都存在这一子模块。

不同的子模块中具体存储的快照数据 snapshots 各不相同。对于 RenameTable(*) 而言，其中各存储了四份 spec_table；对于 StdFreeList 和 MEFreeList 而言，其中则存储了四份 headPtr。

模块内部维护了一对环形指针 snptEnqPtr 和 snptDeqPtr。当 io_redirect 为低电平时，如果快照存储没有满，且 io_enq 为高电平，那么模块会将 io_enqData_* 传入的数据记录到快照存储 snapshots_[snptEnqPtr_value] 处，并将 snptValids[snptEnqPtr_value] 置一，然后 snptEnqPtr 加一。

与此相对的，当 io_redirect 为低电平时，如果 io_deq 为高电平，说明此时快照模块需要出队一个快照。此时，snptValids_[snptDeqPtr_value] 会被置低，然后 snptDrqPtr 加一。

重定向发生时，快照模块根据 io_flushVec_* 信号对内部快照进行冲刷。首先，如果 io_flushVec_* 为高电平，那么对应通道的 snptValids_* 会被置低；其次，snptEnqPtr 会被回退到被置低后的第一个 snptValids_* 为低的位置。

快照中存储的数据会通过 io_snapshots_[0-3]_* 接口传输到模块外部，以供各模块在重定向时恢复使用。重定向时是否使用快照以及使用哪一快照由 CtrlBlock 统一生成信号，本模块仅仅提供快照数据。

7.18 Dispatch

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

7.18.1 术语说明

表 7.7: 术语说明

| 缩写 | 全称 | 描述 |
|----|-----------------|---|
| - | renameIn | rename 模块输入的 uop 信息 |
| - | fromRename | rename 模块输出后经过打拍的 uop 信息 |
| - | toRenameAllFire | 所有 uop 分派完成的信号 |
| - | enqRob | 传给 rob 的信号，会在 ctrlblock 打一拍再进入 rob |
| - | IQValidNumVec | IQ 中每个 Exu 对应的指令数量 |
| - | toIssueQueues | 分派给所有 IQ 的 uop 信息 |
| - | XXBusyTable | 寄存器堆状态表 |
| - | wbPregsXX | 写回寄存器堆的信息，用与更新 BusyTable |
| - | wakeUpXX | 快速唤醒的信息，用与推测更新 BusyTable |
| - | og0Cancel | 表示在 og0 阶段，该 uop 被 cancel |
| - | ldCancel | 表示访存 uop 执行到 s3 阶段 (s0-s3)，该 uop 被 cancel |
| - | fromMem | 来自访存的信号，包括 lsq commit 和 cancel 的数量 |
| - | toMem | 发给访存的信号，包括 lsqEnqIO |

7.18.2 子模块列表

表 7.8: 子模块列表

| 子模块 | 描述 |
|-------------|---|
| XXBusyTable | 寄存器堆状态表，包括 5 个：Int（整数）Fp（浮点）Vec（向量，不含 V0）V0（向量 V0）V1（vcsr 的 v1） |
| rcTagTable | 整数 reg cache（寄存器堆缓存）的 Tag 表 |
| lsqEnqCtrl | 控制进入 load/store queue 指针的模块 |

7.18.3 设计规格

- 支持将 uop 按照负载均衡的策略分派给所有 IQ
- 支持更新维护 BusyTable 并在分派时写到 srcState 中
- 支持更新维护进入 lsq 的指针并在分派时写到 lqidx sqidx 中
- 支持 uop 根据顺序进行阻塞
- 支持分派给 IQ 时屏蔽发生异常的指令

7.18.4 功能

Dispatch 模块包含各个寄存器堆的 BusyTable、rcTagTable、lsqEnqCtrl，根据写回寄存器堆、快速唤醒、og0Cancel 和 ldCancel 来更新 BusyTable 和 rcTagTable，lsqEnqCtrl 模块则会控制 load/store queue 的入队指针 lqidx/sqidx，当 lsq 容量不足时，会拉低 io_enq_CanAccept 来阻塞分派。

Dispatch 模块在每个时钟周期会将经过 rename 之后的至多 6 个 uop 分派给各个 IQ，所有待分派的 uop 全部分派出去后拉高握手信号 toRenameAllFire，rename 模块收到握手信号后更新下一组 uop 给 Dispatch 模块。

Dispatch 模块在每个时钟周期会统计各个 IQ 中每个 Exu 对应的指令数量，针对每种 fu，包含它的所有 Exu 所在的 IQ 之间都会进行负载比较，严格按照负载顺序生成分派策略，存到寄存器中。

Dispatch 模块收集 rename 的输入信号和输出打拍后的信号，根据输入信号中的 fuType 计算出每个 uop 之前的 uop (idx 比自己小的) 和自己 fu 相同的数量，根据两个信息查表得到分派的 IQ，第一个信息是 fu 的类型，第二个信息是在它之前有几个 uop 与自身相同的 fu 类型，根据 IQ 负载从低到高的顺序进行分派，把第一条该 fu 的指令分给负载最低的 IQ，第二条该 fu 的指令分给负载次低的 IQ，以此类推。

Dispatch 模块会接收各个模块的控制信号来阻塞指令的分派，阻塞原因主要包括：分派到的 IQ 不 ready，分派到同一个 IQ 的指令数量超过了 IQ 的入口数量，rob 不能接收指令，lsq 不能接收指令，该指令之前有指令需要 blockBackward，该指令自身或它之前有指令需要 waitForward。阻塞时按顺序阻塞，一条指令被阻塞，这条指令之后的指令也要一起被阻塞。一旦发生阻塞，toRenameAllFire 就会拉低，需等待阻塞住的指令分派完才能分派下一组指令。

Dispatch 模块会将一些异常情况的指令屏蔽掉（将发送给 IQ 的 valid 置低），不分派给 IQ，比如该指令译码出现异常，或者该指令被挂了 singleStep。

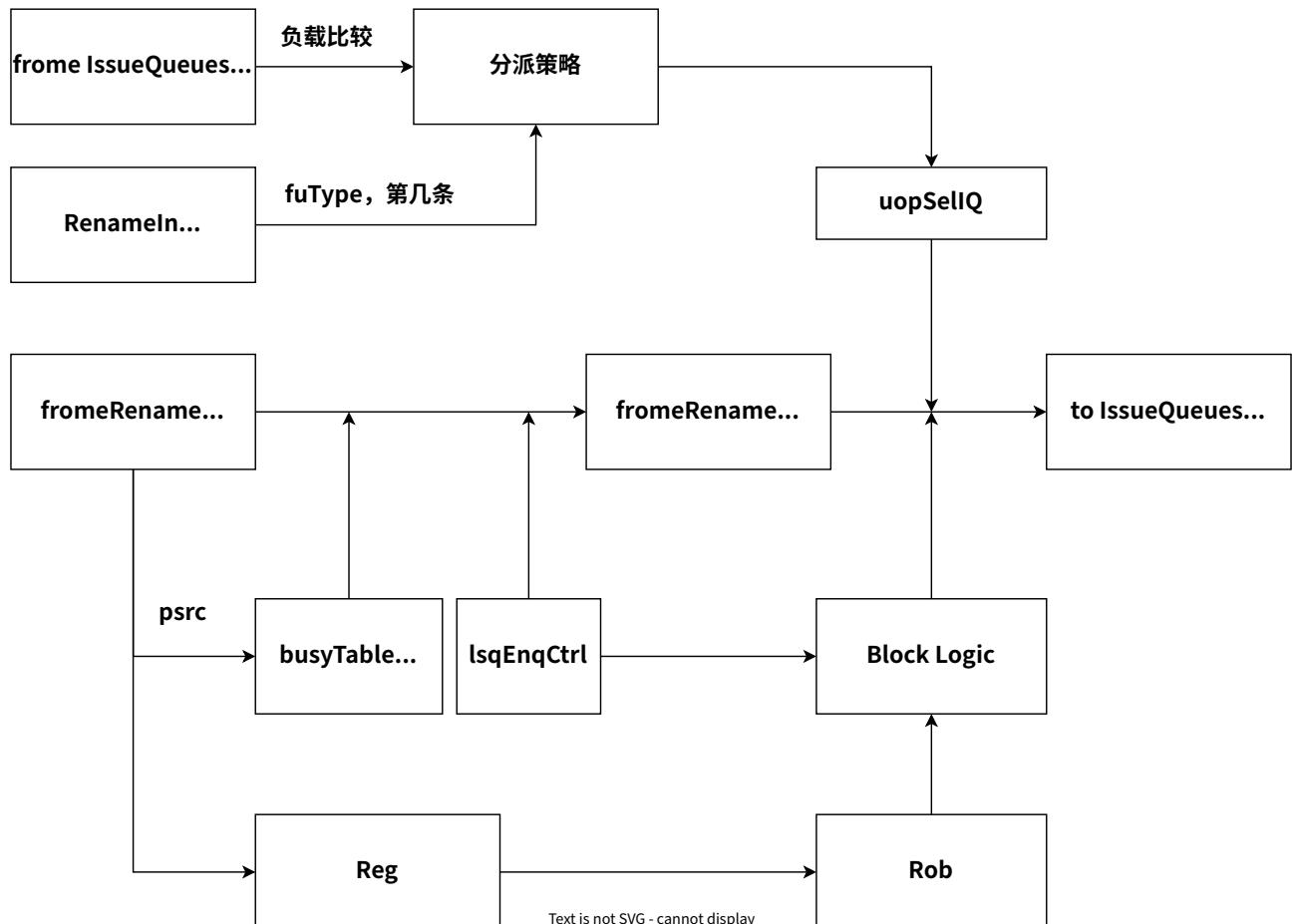


图 7.34: 整体框图

7.18.5 总体设计

7.18.5.1 整体框图

7.18.5.2 接口列表

见接口文档

7.18.6 模块设计

7.18.6.1 二级模块 BusyTable

7.18.6.1.1 功能

BusyTable 模块负责记录寄存器堆繁忙状态， dispatch 的同时需要用 psrc 读 BusyTable 得到源操作数的就绪状态。

每一个寄存器堆对应一个 BusyTable 模块， BusyTable 的项数和寄存器堆保持一致，初始化为 0（空闲状态），当指令经过重命名后，对应的 pdest 信息通过 allocPregs 输入，此时将对应项由 0 变成 1； BusyTable 同时接收推测唤醒的信号 wakeUpXX，当被唤醒时，对应项由 1 变成 0； 推测唤醒的指令有可能会被取消，此时通过 og0Cancel 将对应项由 0 变成 1（可能和 wakeup 是同一拍的，优先级比 wakeup 高），如果是整数的 BusyTable，还需要额外响应 ldCancel。

BusyTable 模块的读口数量根据指令集定义的一条指令需要的对应寄存器操作数数量乘以发射宽度，如 6 发射时整数 BusyTable 读口数量 $2 * 6 = 12$ ，浮点和向量都是 18 个， V0 和 V1 是 6 个。

7.18.6.1.2 整体框图

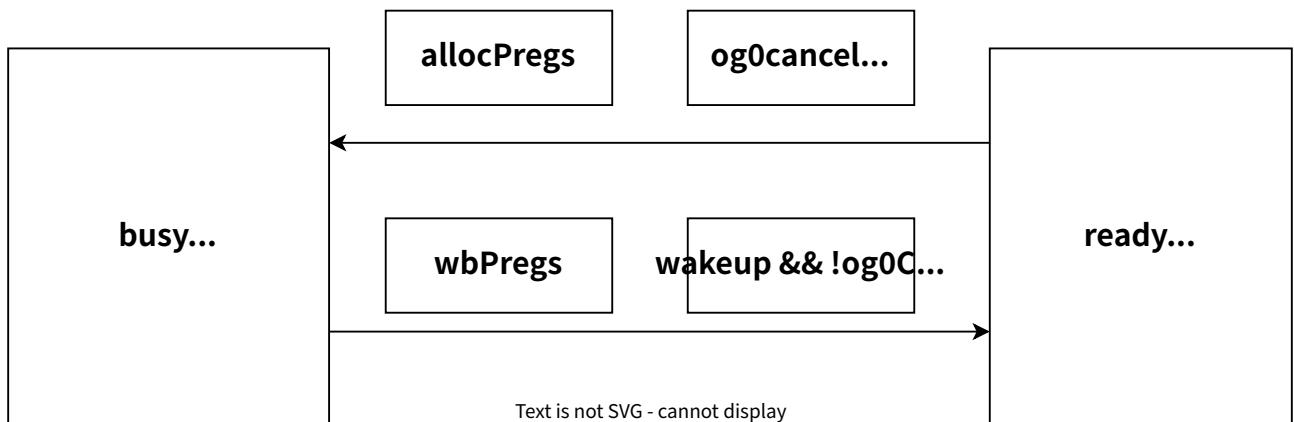


图 7.35: 整体框图

7.18.6.1.3 接口列表

见接口文档

7.18.6.2 二级模块 rcTagTable

7.18.6.2.1 功能

rcTagTable 是整数寄存器堆缓存的 tag，和整数的 BusyTable 模块十分相似，读口也是 12 个。

7.18.6.2.2 接口列表

见接口文档

7.18.6.3 二级模块 lsqEnqCtrl

lsqEnqCtrl 模块负责维护进入 lsq 的指针，并将 uop 发送给 lsq，根据每条指令的 needAlloc (2 比特，低位拉高表示需要进 load queue，高位拉高表示要进 store queue) 和 numLsElem (需要占几项) 进行指针的维护，当 io_enq_iqAccept 拉高时（表示 uop 被 IQ 接收）发送给 lsq。

7.18.6.3.1 接口列表

见接口文档

7.19 Rob

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

7.19.1 术语说明

表 7.9: 术语说明

| 缩写 | 全称 | 描述 |
|------|--------------------|--------------------|
| rob | Reorder Buffer | 重排序缓存 |
| rab | Rename Buffer | 重命名缓存 |
| - | Redirect | ctrlblock 发来的重定向信息 |
| - | Walk | 重定向发生后的回滚过程 |
| snpt | Snapshot | ctrlblock 发来的快照信息 |
| wfi | Wait For Interrupt | 等待中断 |

7.19.2 子模块列表

表 7.10: 子模块列表

| 子模块 | 描述 |
|---------------------|--|
| RobEnqPtrWrapper | 维护 rob 的入队指针 |
| NewRobDeqPtrWrapper | 维护 rob 的出队指针 |
| Rab | 维护 commit 或 walk 时各个 rat 的状态，和 rename 模块交互 |
| VTypeBuffer | 维护 Vtype 的类似 Rab 的结构，和 decode 模块交互 |
| ExceptionGen | 异常产生模块 |
| SnapshotGenerator | 快照产生模块 |

7.19.3 设计规格

- 支持指令写回和提交
- 支持指令重定向
- 支持中断处理
- 支持 Rob 压缩
- 支持快照
- 支持异常处理
- 支持向量访存先写回再处理异常并设置 vstart
- Rob 支持每周期至多 commit/walk 8 个 entry
- Rab 支持每周期至多 commit/walk 6 个 entry

7.19.4 功能

Rob 模块包括：RobEnqPtrWrapper 负责入队指针，NewRobDeqPtrWrapper 负责出队指针，Rab 负责维护 commit 或 walk 时各个 rat 的状态，VTypeBuffer 负责维护 Vtype 的状态，ExceptionGen 负责产生异常，SnapshotGenerator 负责产生快照。

Rob 模块主体是一个循环队列，项数为 160，指针包含 1 比特 flag 和 8 比特 value，当 value 的值从最大值加 1 的时候，flag 会反转，以此来区分指令的顺序。当队列空的时候，enqptr == deqptr，flag 和 value 均相等，当队列满时，enqptr.value == deqptr.value，enqptr.flag /= deqptr.flag，value 相等 flag 不等。每个 RobEntry 包含的信号见下表。

表 7.11: RobEntry 信号列表

| 信号名 | 描述 |
|------------------|---|
| isVset | 是否为 Vset 指令 |
| commitType | 指令的提交类型 |
| isHls | 是否为虚拟化 load/store 指令 |
| wflags | 是否写 fcsr 的 fflags |
| ftqIdx | ftq 的指针，用于读取 pcMem |
| ftqOffset | ftq 的偏移，用于计算得到 pc |
| traceBlockInPipe | trace 在流水线中的数据，包括 iretire、ilastsize、itype |
| instrSize | Rob 压缩的指令条数 |
| fpWen | 用于更新 csr 的 FS |
| isRVC | 是否为压缩指令 |
| dirtyVs | 用于更新 csr 的 VS |
| realDestSize | 指令写目的寄存器的个数 |
| stdWritebacked | store 指令是否写回 |
| uopNum | 需要写回的 uop 个数 |

Rob 采用分 8 个 Bank 读的设计，根据 robidx 的低 3 比特分 bank，例如 robBanks0 包含的 robidx (十进制) : 0 8 16 24 32 ...，robBanks1 包含的 robidx (十进制) : 1 9 17 25 33 ...，每 8 个 entry 为一个 Line, 0-7, 8-15, 16-23 ...，每个 Bank 有 20 个 Entry，一共有 20 个 Line。分 Bank 示意图如下。

| LineAddr | Bank0 | Bank1 | Bank2 | Bank3 | Bank4 | Bank5 | Bank6 | Bank7 |
|----------|-------|-------|-------|-------|-------|-------|-------|-------|
| → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | |
| 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | |
| 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | |
| 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | |
| 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | |
| 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | |
| 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | |
| 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | |
| 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | |
| 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 | |
| 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | |
| 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | |

图 7.36: rob_entries

使用独热的 Line 指针 (20 bit) 来读取 RobEntry 数据，从 8 个 Bank 中读出当前 Line 和下一 Line 的数据 (共 16 个 Entry)，经过当拍的写回信息更新后，从两个 Line 中选一个 Line 写到 8 个 robDeqGroup 寄存器中 (如果第一个 Line 的指令当拍全部提交就选第二个 Line)，指令提交时从 8 个 robDeqGroup 中读数据进行提交。hasCommitted (8 bit) 表示当前行每一条指令是否已经提交，作为其它指令是否可以提交的条件之一，allCommitted 表示当前行全部提交，是切换行指针的控制信号，allCommitted 为 1 时，选读出的第二行数据，也就是后 8 个数据更新后写入到 robDeqGroup。

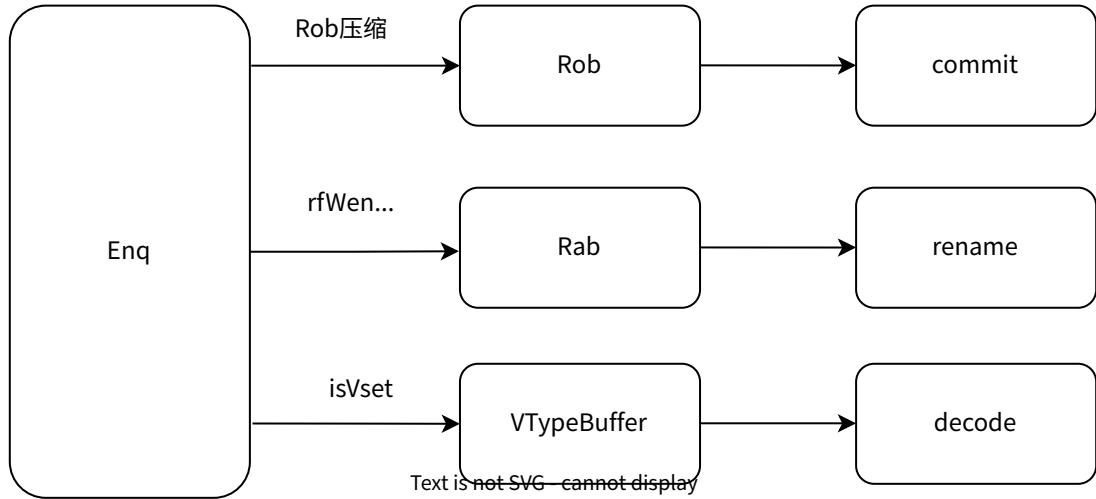


图 7.37: rob_enq

Rob 入队，Rob 可以接受指令时会将 io_enq_canAccept 拉高，此时 Dispatch 可以向 Rob 发送指令，最多发送 6 条。Rob 收到指令后要更新 enqptr，根据入队请求计算 dispatchNum 然后分配 enqptr，如果没有发生 redirect，会将 enqptr 更新为 enqptr + dispatchNum，如果发生了 redirect 信号，则根据 redirect 指令的 robidx 设置 enqptr (和 redirect 的 level 有关)。入队的时候，如果指令需要进行 move 消除，会直接将 writebackd 信号拉高，不需要写回就可以提交，如果译码的时候指令产生异常，指令会在 rename 阶段将 numWB 置 0，指令不会分派给 IQ，进入 Rob 就标记为写回了，特别注意的是向量访存指令需要等 uop 全部写回才能处理异常。allocatePtrVec 是分配的 6 个 enqPtr，分配条件是指令有效并且是第一条 uop (译码或者经过 rob 压缩得到的 firstUp 信号)。canEnqueue (6 bit) 是每一条指令能进入 Rob 的条件：指令有效并且是第一条 uop 并且 rob 可以接收。uopNum 记录了 rob 压缩了多少条指令 (对应 rob 压缩) 或者多少个 uop (对应向量指令拆分) 的，入队的时候更新 uopNum，之后每写回一个 uop (同一拍也可以写回多个 uop) uopNum 减一。对于 store 指令，uopNum 置 1，stdWritebacked 拉低，std 的 uop 不计入 uopNum，它写回时会将 stdWritebacked 拉高。

Rob 写回，Exu 写回 rob 的控制信号会在 ctrlBlock 里打一拍，由于 Rob 压缩会导致多个 Exu 写回相同的 robidx，在 ctrlBlock 里打一拍的同时，会进行 Rob 压缩的计算，每个 Exu 会统计所有可能压缩到一起的 Exu (某些 Exu 之间不可能存在压缩关系，所以不必浪费面积和时序全部统计) 中写回的 robidx 和自己相同的个数，通过 io 里的 writebackNums 传给 Rob。

Rob 提交，出队指针位置的指令在 Rob 状态机处于 idle 状态时、指令有效、uop 全部写回、blockCommit 拉低的时候提交。当出队位置的指令存在异常的时候，blockCommit 会拉高阻止指令提交，直到异常处理完成后该指令才可以被提交。commitValidThisLine 表示 deqptr 所在的那一行的 8 个 entry 是否可提交，判断方式为该 entry 有效并且该 entry 所有 uop 都已经写回并且此时 rob 没有使能中断并且出队指令中没有异常并且出队指令中没有需要 reply 的指令并且没有被比他更老的指令阻塞提交并且它本身没有提交过。注意 allowOnlyOneCommit 情况，当出队的 8 个 Entry 中有发生异常的指令或者使能中断时，rob 每周期只允许提交一条指令。

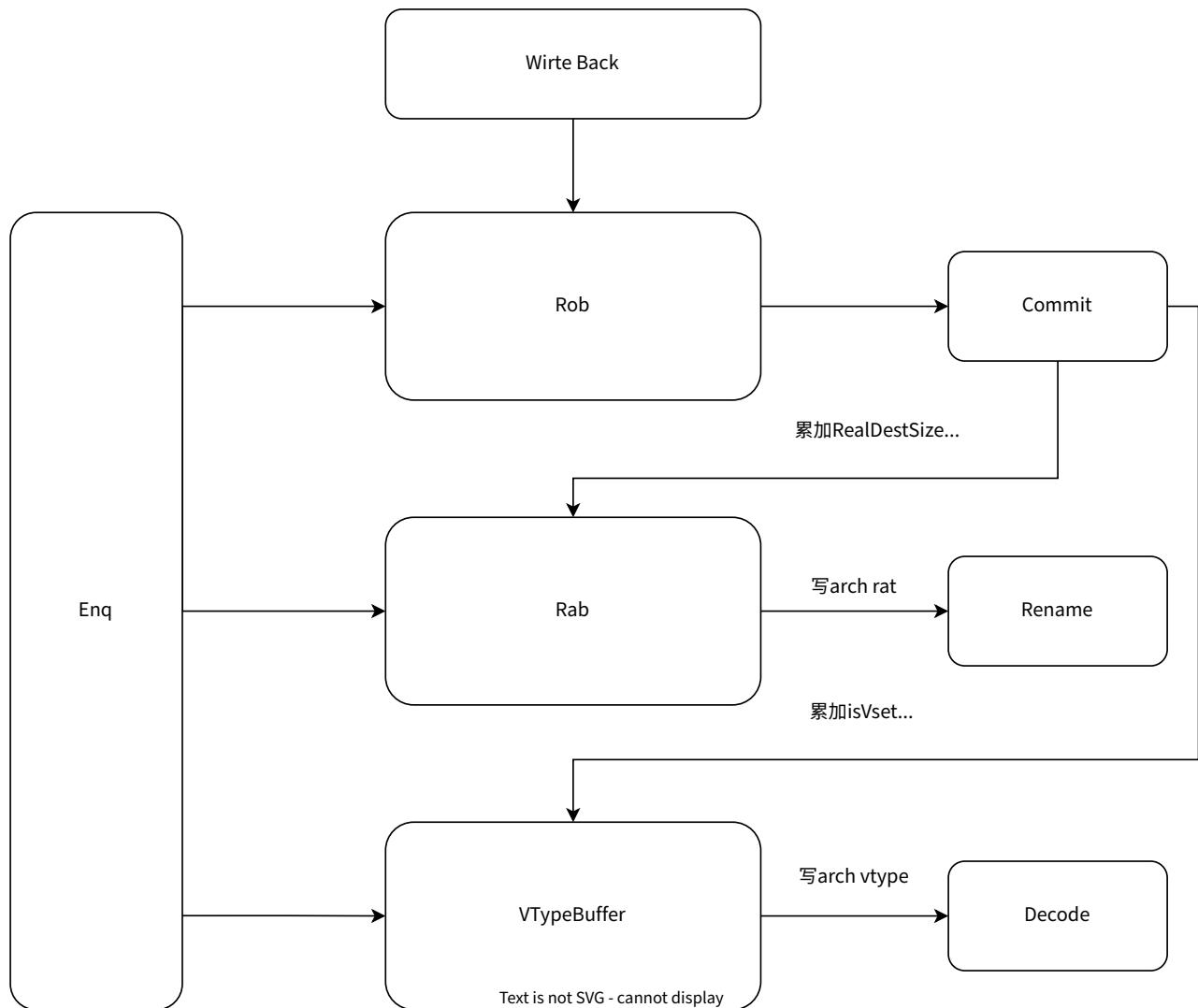


图 7.38: rob_commit

Rob 出队，Rob 会将提交后的指令出队，统计提交 Entry 的数量，将 deqptr 的值加上提交 Entry 的数量，更新出队指针，把出队 Entry 的 valid 置低。

Rob 状态机，有 s_idle 和 s_walk 两种状态，状态的更新主要和 redirect 有关。s_idle：正常状态，可提交指令，redirect 之后至少两拍 walk 状态后才能回到 idle 状态。s_walk：walk 状态，不可以提交指令，等待各模块 walk 结束恢复到 s_idle 状态，状态机切换代码如下。

```
/**
 * state changes
 * (1) redirect: switch to s_walk
 * (2) walk: when walking comes to the end, switch to s_idle
 */
state_next := Mux(
    io.redirect.valid || RegNext(io.redirect.valid), s_walk,
    Mux(
        state === s_walk && walkFinished && rab.io.status.walkEnd && vtypeBuffer.io.status.walkEnd,
        state
    )
)
```

Rob 重定向和快照，Rob 在 redirect valid 的当拍不会提交指令，根据 walk 的起始地址切换 Rob 的读指针，walk 的起始地址来源有两个：snapshot，deqptr，walk 的起始地址会从中选择比发出 redirect 指令更老的并且最近的位置。Rob 中 snapshot 保存的信息是一组 robidx，保存的 robidx 的值在入队的第一条的 robidx 的基础上，+0, +1, +2, +3, +4, +5, +6, +7，共 8 个 robidx。Rob 的 snapshot 会受 ctrlblock 里面的 snapshot 控制，下图为 walkPtr 的选择示例。

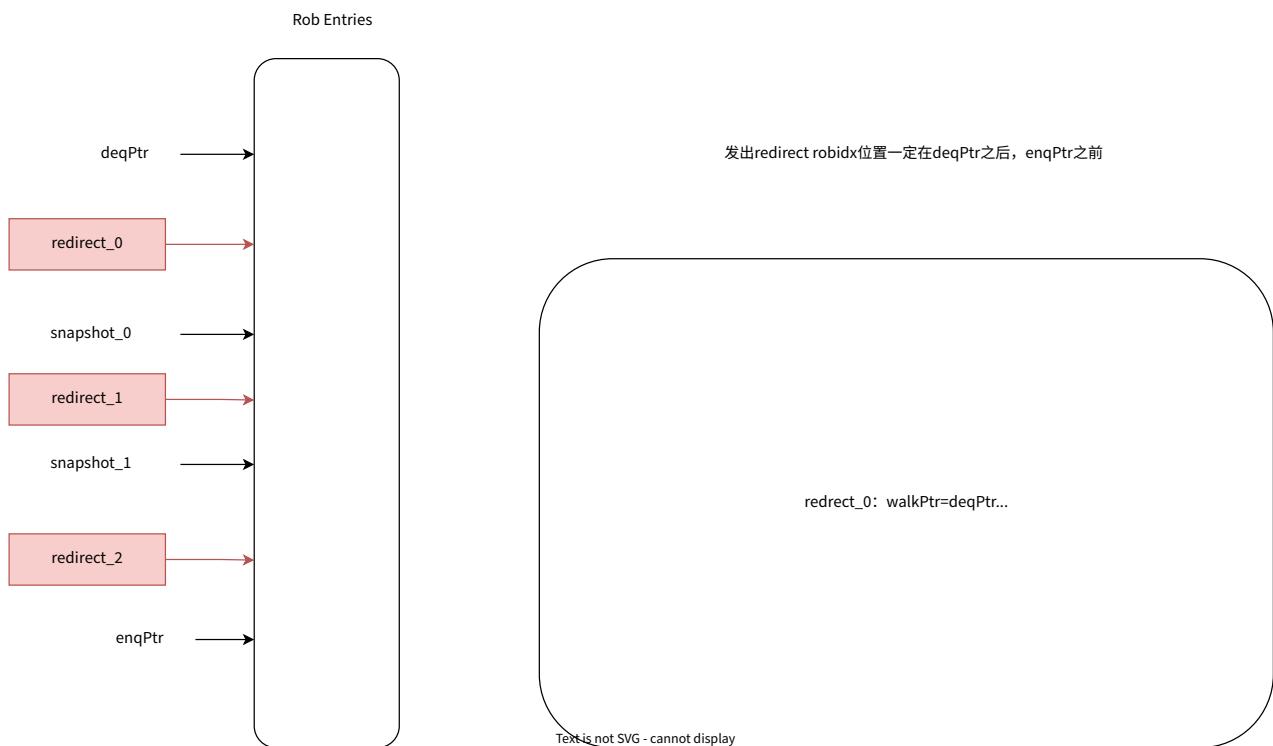


图 7.39: rob_walkPtr

walkPtr 的更新：如果 redirect 有效时，如果 io_snpt_useSnpt 为 1 时根据 io_snpt_snptSelect 选择对应的快照，io_snpt_useSnpt 为 0 时选择 deqPtr，注意 walkptr 要对齐到 bank0 的地址；如果 redirect 无效并且 rob 处于 walk 状态并且没有 walk 结束，walkptr 每周期增加 8；其它条件 walkptr 不更新。lastWalkPtr 是 walk 的终点，根据 redirect 的指令刷不刷自己确定，刷自己 lastWalkPtr 是 redirect 的 robidx - 1，不刷自己 lastWalkPtr 是 redirect 的 robidx。donotNeedWalk 机制，在 walk 的第一拍 8 个 entry 中，比发出 redirect 的 robidx 更老的指令是不需要 walk 的。walk 结束的判断，walkPtrTrue > lastWalkPtr 时 walkFinished 为 1，walkPtrTrue 是不考虑 Bank 地址对齐的 walkPtr，walkFinished 为 1 时把结束 walk 的信息传给 rab 和 vtypeBuffer。shouldWalkVec 表示 8 个 entry 是否应该 walk，判断条件是比 lastWalkPtr 更老的指令，结合 donotNeedWalk 最终判断是否要 walk。

Redirect 有效时，当拍 rob 不可以提交指令，walk 指针更新到 walk 的起点（快照恢复或者出队位置），注意 walk 的起始地址只能是 Bank0 中 entry 对应的 robidx，记录 walk 的终点位置 lastWalkPtr；下一拍状态机变为 walk 状态，更新读 Bank 的指针到 walk 指针相应的位置，将 robEntry 中在 redirect 后面的指令的 valid 置为 0；下下拍从 8 个 robDeqGroup 中取需要 walk 的信息传给 rab (realDestSize), VTypeBuffer (isVset)。处于 walk 状态时，每拍 walk 8 个 rob 的 entry，将 8 个 entry 中的 realDestSize 累加后传给 rab，将 isVset 累加后传给 VTypeBuffer，rob walk 到 lastWalkPtr 时停止 rob 自己的 walk，但是要等到 rab 和 VTypeBuffer 都 walk 结束 rob 才能恢复 idele 状态。Rab 每周期最多 walk 6 个 Entry。VTypeBuffer 每个周期最多 walk 8 个 Entry。

Rob 异常处理，由于产生异常指令之后的所有指令都不会执行，Rob 只需要保存最老的异常，通过 Rob 异常生成模块实现该功能。Rob 内部只需对正在提交的指令进行异常的判断。Rob 的异常生成模块中，enq 信号（和 Rob 入队信号同拍）负责传入来自 frontend 和 decode 产生异常信息，对应最多 6 条指令，wb 信号负责传入功能单元写回的异常信息，(csr + fence + load + store + vload + vstore)，需要输出最老的指令对应的异常信息。其中 current 信号保存了当前的异常信息。enq 传入的指令是有序的，因此只需要使用 priorityMux 就可以得到最老的异常，wb 传入的指令是乱序的，需要使用比较 robidx 的方法来选出最老的异常。异常处理模块会分组选最老的指令，第一拍在各个组选出最老的，第二拍从第一拍结果中选最老的指令。在第二拍得到的最老的异常信息与 current 进行比较，如果 current 更年轻，则将 current 更新为第二拍得到的最老的异常信息。特别的，对于向量访存写回的异常，它们的 robidx 相同但是 uop 有很多个，此时不仅需要比较最老的 robidx，还要对比异常要置的 vstart，保留 vstart 小的异常信息。

Rob 中断处理，中断和异常处方式相似。中断来自 CSR 模块，对于需要发出 flushPipe 和 replayInst 的指令，目前也进入 exceptionGen 中处理。Rob 处理它们的方式都是先发一个 flushOut 给 ctrlBlock，ctrlBlock 会回一个 redirect 来刷流水线。区别是分支跳转失败和访存违例产生的 redirect 获取 target 比较快，直接从 pcMem 读到一个 pc 再结合 ftqOffset 计算出 target 发给前端；对于中断和异常，需要先把信息发给 CSR，CSR 返回对应的 target 再发给前端。中断目前只会在 deqPtr 是非 load、store、fence、csr、vset 的指令时才会响应。

当 wfi_enable 信号拉高时（来自 CSR 寄存器，wait-for-interrupt enable），当 wfi 指令入队 Rob 的时候会将 hasWFI 置为 1，hasWFI 会把 blockCommit 置为 1，阻塞 rob 的提交从而起到暂停流水线等待中断的作用。当 csr 收到中断时，会将 io_csr_wfiEvent 拉高，hasWFI 置为 0（或者超时 1M cycle 没等到中断也会置 0），然后 Rob 可以正常提交指令。

7.19.5 总体设计

7.19.5.1 整体框图

7.19.5.2 接口列表

见接口文档

7.19.6 模块设计

7.19.6.1 二级模块

7.19.6.1.1 功能

7.19.6.1.2 整体框图

7.19.6.1.3 接口列表

见接口文档

8 DataPath 数据通路

8.1 DataPath

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/15
- commit: xxx

8.1.1 术语说明

表 8.1: 术语说明

| 缩写 | 全称 | 描述 |
|--------------|----------------------------|--------------|
| og | operand generation | 源操作数生成阶段 |
| v0 | vector register #0 | 向量 0 号逻辑寄存器 |
| vl | vector length csr register | 向量长度 csr 寄存器 |
| rc/reg cache | register file cache | 寄存器堆缓存 |

8.1.2 总体设计

8.1.2.1 整体框图

8.1.2.2 子模块列表

表 8.2: 子模块列表

| 子模块 | 描述 |
|-----------------------|-----------------|
| IntRFWBCollideChecker | 整数寄存器堆写口仲裁器 |
| FpRFWBCollideChecker | 浮点寄存器堆写口仲裁器 |
| VfRFWBCollideChecker | 向量通用寄存器堆写口仲裁器 |
| V0RFWBCollideChecker | 向量 v0 寄存器堆写口仲裁器 |
| VlRFWBCollideChecker | 整数 vl 寄存器堆写口仲裁器 |
| IntRFReadArbiter | 整数寄存器堆读口仲裁器 |
| FpRFReadArbiter | 浮点寄存器堆读口仲裁器 |

| 子模块 | 描述 |
|-----------------|-----------------|
| VfRFReadArbiter | 向量通用寄存器堆读口仲裁器 |
| V0RFReadArbiter | 向量 v0 寄存器堆读口仲裁器 |
| VlRFReadArbiter | 向量 vl 寄存器堆读口仲裁器 |
| IntRegFile | 整数寄存器堆 |
| FpRegFile | 浮点寄存器堆 |
| VfRegFile | 向量通用寄存器堆 |
| V0RegFile | 向量 v0 寄存器堆 |
| VlRegFile | 向量 vl 寄存器堆 |
| RegCache | 整数寄存器堆缓存 |

8.1.2.3 接口列表

见接口文档

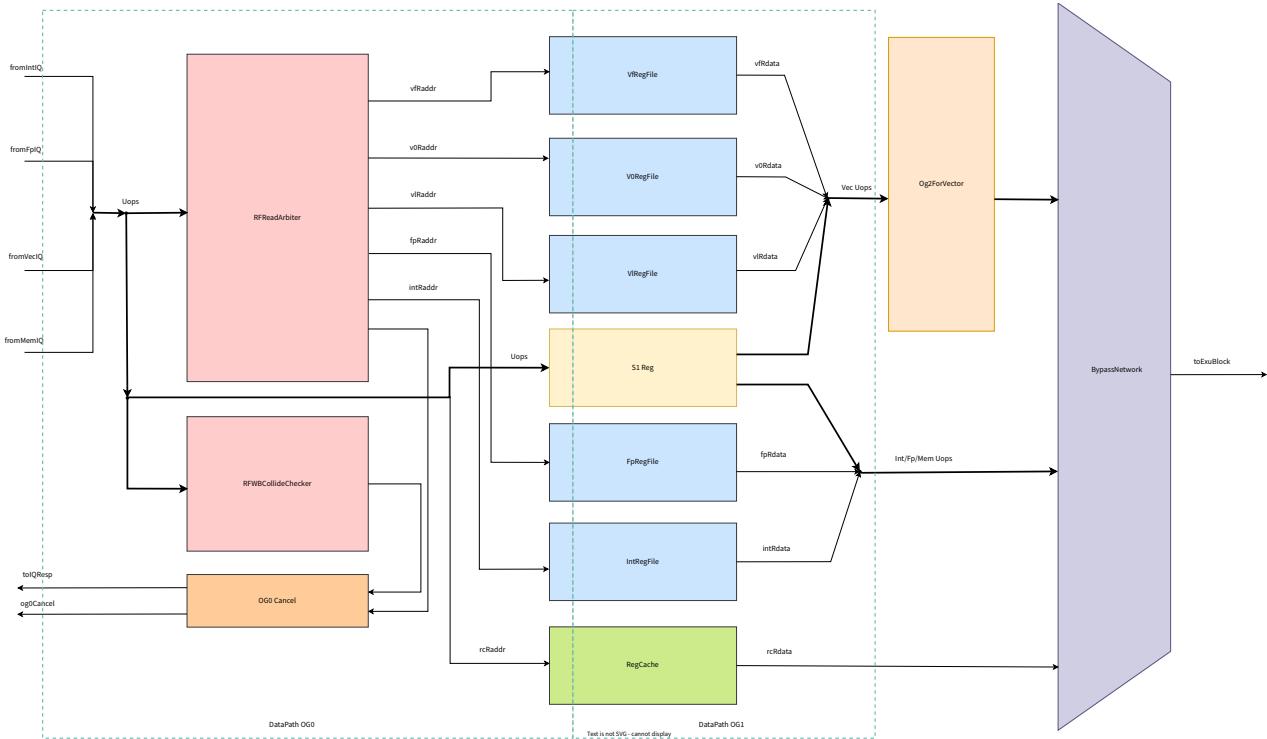


图 8.1: 整体框图

8.1.3 功能

8.1.3.1 整体功能

数据通路在流水线中位于发射之后，进入执行单元之前。数据通路接收来自整数、浮点、向量、访存发射队列每个出口的指令，这些每个出口会与执行单元的 ExeUnit 一一对应。数据通路负责为每条指令读取寄存器堆，构造立即数，在进入 ExeUnit 前生成最终的操作数。数据通路里含有寄存器堆的实体，也负责寄存器堆的读写仲裁，以及数据读出和写入。目前数据通路还设置了整数寄存器堆的缓存，负责一部分整数数据的读出。

DataPath 模块是数据通路的核心部分，包含整个 OG0 流水级和 OG1 流水级的前半部分。在 OG0 流水级里，会进行指令间的读口端口仲裁，只有仲裁成功的指令才读取寄存器堆，进入 OG1 流水级；仲裁失败的指令会刷掉自身，向前级发出发射失败的回应信息，以及 og0 取消信号。在 OG1 流水级，指令拿到寄存器堆返回的数据，和其他状态信息一起发往后续模块。注意在 DataPath 里不会进行指令操作数的最终生成，只会拿到从寄存器堆，Reg Cache，PcTargetMem 里读出的备选数据，最终数据会在外部的 BypassNetwork 里生成。

8.1.3.2 读写仲裁

由于采用发射后读寄存器堆，发射到数据通路中的指令数量会远远超过寄存器堆的读口和写口数量，因此需要进行仲裁，只让读仲裁成功的指令去读寄存器。

每个寄存器堆都有一个读仲裁器，读仲裁器接收所有指令的每个操作数的 valid 和读地址 addr，分别返回一个 ready 作为仲裁是否成功的标记。只有当指令的操作数是对应的数据类型（整数、浮点、向量通用、v0、vl），并且数据来源为 reg 类型，才会将 valid 拉高，发起仲裁请求。如果某个操作数不需要读寄存器堆，对应的 valid 为 0，此时 ready 总会返回 1 让其通过仲裁器。对于每条指令，只有其所有的操作数都仲裁通过，才被视为读仲裁成功。读仲裁器还会输出一组读请求，数量为寄存器堆的读口数量，内容为每个读口最后仲裁成功的读地址。这组读请求会被送到寄存器堆进行实际的数据读取。

每个寄存器堆都有一个写仲裁器，写仲裁器接收所有指令的每个操作数的 valid，分别返回一个 ready 作为

仲裁是否成功的标记。只有当指令的目的寄存器数据类型匹配，并且需要写寄存器，才会将 valid 拉高，发起仲裁请求。与读仲裁类似，不发起请求的指令总会收到 ready=1 让其通过仲裁。

读写仲裁器内部的仲裁逻辑放在二级模块进行具体介绍。

8.1.3.3 读寄存器堆

每个寄存器堆的读仲裁器会输出一组读请求，数量为寄存器堆的读口数量，内容为每个读口最后仲裁成功的读地址。这些地址在 OG0 阶段被送往寄存器堆，发起读请求。在 OG1 阶段，相应数据从寄存器堆的读数据通道送出。各个指令的操作数根据自身的数据类型，和使用的读口编号，从寄存器堆的读数据里选出自己的数据。

8.1.3.4 读 Reg Cache

DataPath 内设置了 Reg Cache，可以缓存整数 ExeUnit 和 Load ExeUnit 最近写回的数据。Reg Cache 不需要进行读写仲裁，其端口数与需要读写的操作数数量一一对应。如果指令操作数的数据来源为 regcache 类型，将会向 Reg Cache 发起读请求。在 OG0 阶段，将 valid 信号和指令自身携带的 RC 地址信号传给 Reg Cache，在 OG1 阶段可以从对应的数据通道中拿到数据。

8.1.3.5 读 PC

部分指令需要读取 PC 作为源操作数，PC 存储在 DataPath 外部的 PcTargetMem 中。在 OG 阶段，DataPath 首先筛选出需要 PC 的指令，将其 ftq 信息通过 io.fromPcTargetMem 接口向外部发起读请求。在 OG1 阶段，外部通过该接口返回读到的 PC 信息。

8.1.3.6 写寄存器堆

执行单元的写回结果会由外部模块 WbDataPath 进行汇总，打包为写请求的格式发到 DataPath。DataPath 会接收到 io.fromIntWb、io.fromFpWb、io.fromVfWb、io.fromV0Wb、io.fromV1Wb 五个接口传回的写回信息，每个都对应各自的寄存器堆。每个接口的通道数量为寄存器堆的写口数，内容为写使能、写地址、写数据。出于时序考虑，这些信号会在 DataPath 内部打一拍，然后直接送往寄存器堆写端口。

8.1.3.7 写 Reg Cache

Reg Cache 的写回数据不是来自执行单元，而是来自旁路网络中执行单元写回两拍之后的数据。Reg Cache 只接受整数 ExeUnit 和 Load ExeUnit 的数据，这些数据通过 io.fromBypassNetwork 接口传入，然后直接送往 Reg Cache。

8.1.3.8 处理 og0 cancel

在 OG0 阶段，指令会因为多种原因被取消，其中因为自身原因的取消被称为 og0 cancel。指令被判定为 og0 cancel 有两种情况，一种是自身读或者写寄存器堆仲裁失败，另一种是 og0 cancel 传递。DataPath 中会保存前一次 og0 cancel 的信息，这是一个与发射宽度相同的向量，每位代表对应指令是否发生了取消。如果某一指令为 0 执行延迟的指令，且发生了取消，那么就会在向量对应位置写 1。下一拍的指令会将自身操作数的唤醒来源向量和取消信息的向量进行比较，如果发现来源指令上拍被取消了，说明现在自身也要被取消。

发生了 og0 cancel 的指令会把自己刷掉，不进入 OG1 阶段。除了发生 og0 cancel，发生重定向刷新或者 load cancel 的指令也会将自己刷掉，不加入 OG1 阶段。不过只有发生 og0 cancel 的指令才向外发出 og0

cancel 信号，该信号与前面提到的 DataPath 内部保存的 og0 cancel 信息相同，通过 io.og0Cancel 接口传到外部，用于取消指令的消费者指令。

8.1.3.9 向发射队列发出回应

DataPath 会向发射队列发出回应信息，告诉发射队列指令是否顺利发射到执行单元。在 OG 阶段，如果指令发生了 og0 cancel，指令发射失败，需要给发射队列回应 block 状态，告诉发射队列需要重新发射指令；如果顺利通过 OGO 阶段，则不发送回应信息。在 OG1 阶段，指令不会因为自身原因被取消，只需要看后级是否能接收指令。如果不能接收，指令无法进入执行单元，需要给发射队列回应 block 状态。如果可以被接收，对于标量计算指令，指令就一定能成功执行，此时就可以向发射队列回应 success 状态，告诉发射队列可以清空对应的队列项；对于向量计算指令，在进入执行单元前还有一个 OG2 阶段，此时还不能确定一定能执行，需要向发射队列回应 uncertain 状态，让发射队列先保持不变；对于访存指令，其进入访存执行单元后才能确定能否成功指令，因此清空队列项的回应由访存部分发出，这里只向发射队列回应 uncertain 状态。

8.1.4 模块设计

8.1.4.1 二级模块 RFWBCollideChecker

8.1.4.1.1 功能

各个寄存器堆都有一个写口仲裁器，负责进行写口的仲裁。写口仲裁器会收集所有可以写该寄存器堆的指令的请求（valid 信号），然后分别返回是否仲裁成功的标记（ready 信号）。

在仲裁器内部，会按照端口号分为多个端口仲裁器。每个端口仲裁器会收集写该端口的指令请求，比如如果指令使用 i 号端口写回该堆，那么其请求会被发到第 i 个端口仲裁器。

在端口仲裁器中，采用的是带保底的优先级仲裁策略。在配置功能单元时，各个单元配置寄存器堆写口时，除了端口号，也会配置一个优先级。如果某一时刻有多个指令同时发出写请求，那么端口仲裁器会让优先级更高的指令仲裁成功。

端口仲裁器保证，如果有指令没发出请求，那么其 ready=1，认为仲裁成功；如果有多个指令同时发出请求，发出请求的里面只会有一个 ready=1，仲裁成功。

端口仲裁配置了保底策略，它给每个指令分配了一个计数器，每当指令发出请求但仲裁失败，就会将计数器加 1；每当指令发出请求并仲裁成功，就会将计数器清空。当计数器累积到最大值 7 后，会触发保底状态，保底状态下，所有非保底的指令请求会被屏蔽，然后进行优先级仲裁。当保底成功，指令得到成功仲裁，就会清空计数器，退出保底状态。

8.1.4.1.2 整体框图

以浮点寄存器堆的写口仲裁器为例，内部结构如下图所示。

8.1.4.1.3 接口列表

见接口文档

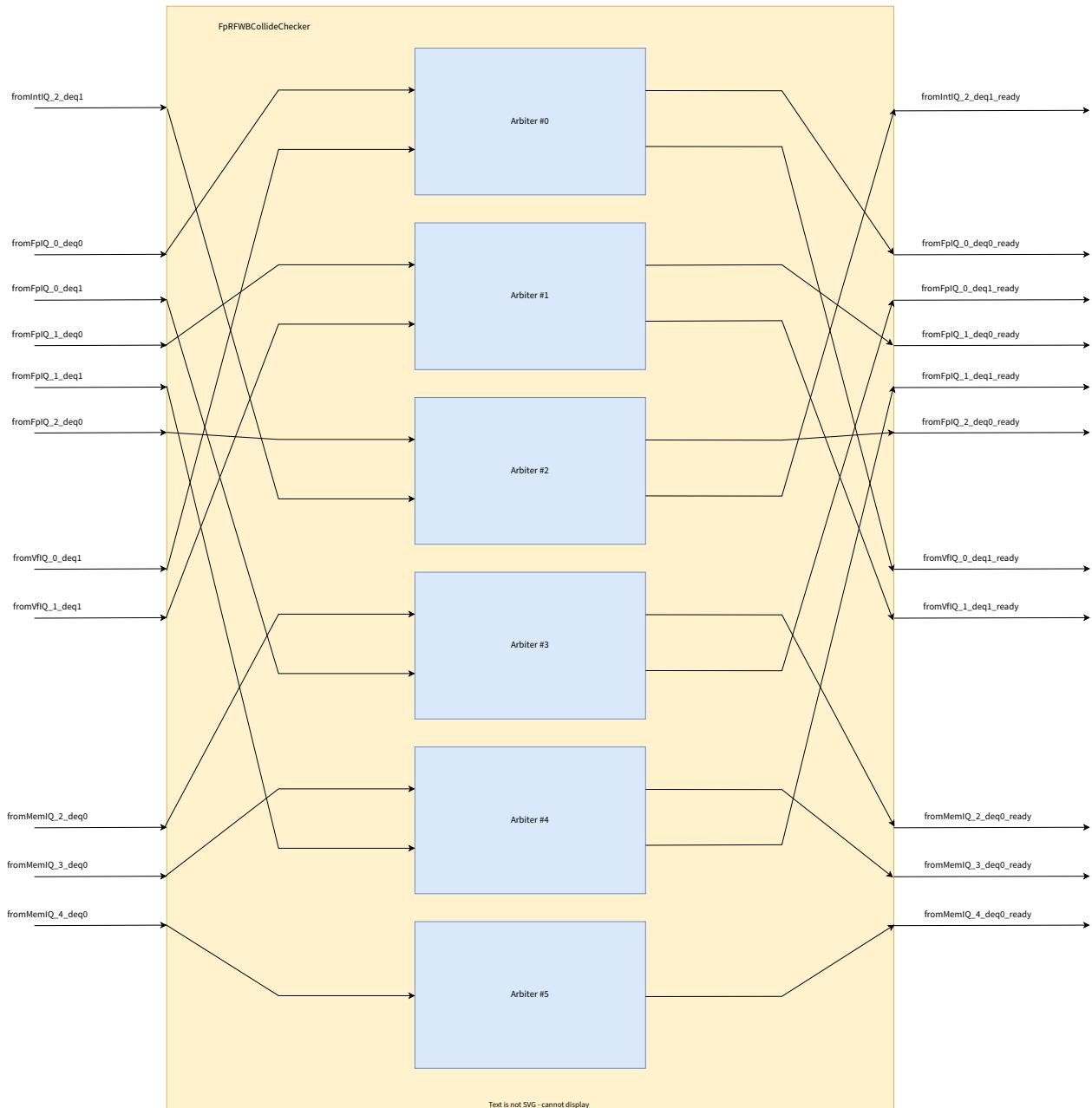


图 8.2: 整体框图

8.1.4.2 二级模块 RFReadArbiter

8.1.4.2.1 功能

各个寄存器堆都有一个读口仲裁器，负责进行读口的仲裁。与写口仲裁器类似，读口仲裁器也会收集所有可以读该寄存器堆的操作数的请求，返回是否仲裁成功的标记。不同之处在于，读口仲裁器收集的请求中还会携带读地址，最后还会输出每个读端口最终仲裁成功的地址，这些地址后续会被送到寄存器堆进行读操作。

在仲裁器内部，同样按照端口号分为多个端口仲裁器分别仲裁。和写仲裁相同，每个端口仲裁器也同样采用带保底的优先级仲裁策略。每个端口最后输出的地址为发出请求且仲裁成功的操作数提供的地址，如果某一时刻没有读请求，最终地址是优先级最低操作数的地址。

8.1.4.2.2 整体框图

读口仲裁器的结构与写口仲裁器基本相同，此处不再赘述，主要区别在于每个读请求会携带读地址，同时内部每个端口仲裁器还会输出一个最终地址。

8.1.4.2.3 接口列表

见接口文档

8.1.4.3 二级模块 RegFile

8.1.4.3.1 功能

DataPath 中设置有 5 个物理寄存器堆，其中整数和浮点都是单独的一个堆，向量则被拆分为 3 个堆，用于减少端口数量和面积开销。其中 VfRegFile 为通用向量寄存器堆，保存向量 #1-#31 逻辑寄存器和一些临时寄存器的值。V0RegFile 为 v0 寄存器堆，只保存向量 #0 逻辑寄存器的值。VlRegFile 为 vl 寄存器堆，只保存向量 vl CSR 寄存器的值。

寄存器堆使用了分块设计，根据具体情况会被分为 S=1、2、4 块。分块是纵向的，即容量为 N，元素大小为 M-bit 的寄存器堆会被分为 S 个 N * (M / S) 大小的寄存器堆。每个元素会被分到各个块里，在读取和写入寄存器堆时，都会同时读取和写入每个分块堆。

寄存器堆有 R 个读口，每个读口有地址 raddr 和数据 rdata 两个信号，没有读使能。在某一时刻提供读地址，寄存器堆将该地址打一拍，下一拍用地址去读取对应的数据，发送到数据接口。

寄存器堆有 W 个写口，每个写口有写使能 wen、写地址 waddr 和写数据 wdata 三个信号。在某一时刻，如果 wen 拉高，那么对应地址会被写入所给的数据，写入的数据会在下一拍被看到。整数寄存器堆有一个特殊位置，0 号地址永远不会被实际写入数据，总是保持 0 值。

8.1.4.3.2 规格

表 8.3: 寄存器堆规格

| 寄存器堆 | 容量 | 位宽 | 读口数量 | 写口数量 | 分块数量 |
|------------|-----|---------|------|------|------|
| IntRegFile | 224 | 64-bit | 11 | 8 | 4 |
| FpRegFile | 192 | 64-bit | 11 | 6 | 4 |
| VfRegFile | 128 | 128-bit | 12 | 6 | 4 |

| 寄存器堆 | 容量 | 位宽 | 读口数量 | 写口数量 | 分块数量 |
|-----------|----|---------|------|------|------|
| V0RegFile | 22 | 128-bit | 4 | 6 | 2 |
| VIRegFile | 32 | 8-bit | 4 | 4 | 1 |

8.1.4.4 二级模块 RegCache

8.1.4.4.1 整体框图

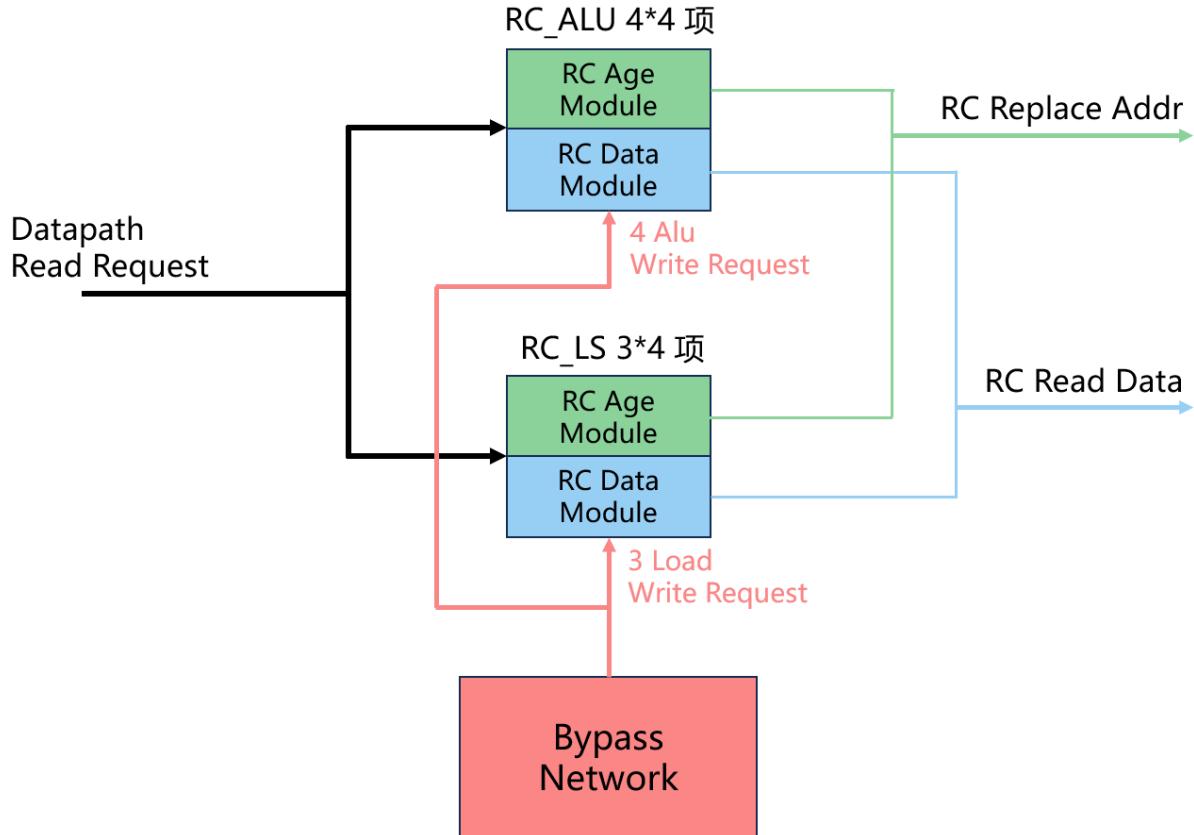


图 8.3: RegCache 整体框图

8.1.4.4.2 功能

Reg Cache 作为 Reg File 的子集，保存部分 EXU 最近的写回结果，承担一部分原来 DataSource 为 reg 类型的读数据请求，减少 Reg File 的读口数量。

目前仅对整数寄存器堆设置 Reg Cache，只保存 4 个 ALU 所在的 EXU 和 3 个 LDU 的写回结果。

8.1.4.4.2.1 数据部分 (RC Data Module)

数据部分流水线位置与 Reg File 相同，都是 OG0 阶段发读请求，OG1 阶段拿到数据。

数据部分分为两部分，RC_INT 负责存放 4 个 ALU 所在的 EXU 的结果，RC_LS 负责存放 3 个 LDU 的结果。

每个部分内部都采用全相连结构，整个 Reg Cache 的寻址采用统一寻址，使用 5 bit，最高位为 0 代表 RC_INT，为 1 代表 RC_LS。

Reg Cache 的具体参数配置如下表所示。

表 8.4: Reg Cache 规格

| Reg Cache | 容量 | 位宽 | 读口数量 | 写口数量 |
|-----------|----|--------|------|------|
| RC_INT | 16 | 64-bit | 23 | 4 |
| RC_LS | 12 | 64-bit | 23 | 3 |

(1) RC data 的读取

指令读取 RC 数据与读取寄存器堆类似，在数据通路的 OG0 阶段，数据来源为 RC 类型的操作数向 RC 发起读数据请求，将对应的 RC 地址发送出去。

在数据通路的 OG1 阶段，得到 RC 数据结果，该结果会传送到 BypassNetwork 中，根据数据来源类型多路选择出最终数据。

读取 RC 时不设置仲裁，每个可以读整数寄存器堆的操作数都设立一个独占的读口。

(2) RC data 的写入

RC data 使用 BypassNetwork 里 bypass 阶段的数据进行写入，写入时的地址使用发出唤醒信号时携带的 RC 地址。

由于发出唤醒到写回数据到达 bypass 阶段有 3 拍的间隔，需要在 RC 里将选出的替换项地址打 3 拍后用于写入数据。

8.1.4.4.2.2 年龄部分 (RC Age Timer)

年龄部分分为年龄计数器和年龄矩阵两个模块。年龄计数器模块对 RC 每项设立一个 2bit 的年龄计数器，根据 RC 项的读写情况进行更新。

同时对 RC_INT 和 RC_LS 分别维护一个年龄矩阵，用于每周期分别选出 4、3 个待替换的项，将它们的 RC_Idx 传给 4 个 ALU 的 WakeUpQueue 和 3 个 LDU 的 WakeUpQueue。每个 WakeUpQueue 在发出快速唤醒的时刻携带对应的 RC_Idx，告知其消费者应当从该位置获取数据。

对于 N 项的 RC，其年龄矩阵是一个 $N \times N$ 的方阵，每项 $Age[i][j]$ 表示 i 项和 j 项的相对年龄次序，为 1 表示 i 项比 j 项更老，i 项应当先被换出。

显然有 $Age[i][j] = \sim Age[j][i]$ (如果 $i \neq j$)，同时我们规定如果 $i == j$, $Age[i][j] = 1$ 。这样，实际需要存储的部分是矩阵的上三角，有 $N * (N - 1) / 2$ bit。

(1) 替换算法

年龄矩阵通过以下方式维护：

- 在 T0 时刻根据各项的状态，通过一个年龄比较函数得到两两之间的年龄次序，写入年龄矩阵
- 在 T1 时刻根据每行中 1 的数量，超过阈值的项认为被选中替换

如果要选出 M 项替换，那么 1 的数量 $\geq N - M + 1$ 的项则被选中。根据 1 的数量 = $N - M + 1$ 到 N 选出这 M 项的位置，即可得到 RC_Idx。

(2) 年龄更新

RC 每项维护一个 AgeTimer，更新规则如下：

- 当该项被更新时，清零计数器
- 当前有读请求，保持不变（包括发了读请求但指令被取消的情况）
- 计数器已达到最大值，维持不变
- 其余情况，计数器 +1

```
if (wen):
    AgeTimerNext = 0
else if (hasReadReq):
    AgeTimerNext = AgeTimer
else if (AgeTimer == 3):
    AgeTimerNext = 3
else:
    AgeTimerNext = AgeTimer + 1
```

(3) 年龄比较

年龄比较函数根据 AgeTimerNext 进行比较，计数器越大认为越老，如果两项计数器值相同，认为序号越小的越老。

```
for(i = 0; i < N ; i++)
    for(j = 0; j < N; j++)
        if (i == j)
            AgeNext[i][j] = 1
        else if (i < j)
            if (AgeTimerNext[i] >= AgeTimerNext[j])
                AgeNext[i][j] = 1
            else
                AgeNext[i][j] = 0
        else
            AgeNext[i][j] = ~AgeNext[j][i]
```

8.1.4.4.3 接口列表

见接口文档

8.2 Og2ForVector

- 版本：V2R2
- 状态：OK
- 日期：2025/01/20
- commit：xxx

8.2.1 总体设计

8.2.1.1 整体框图

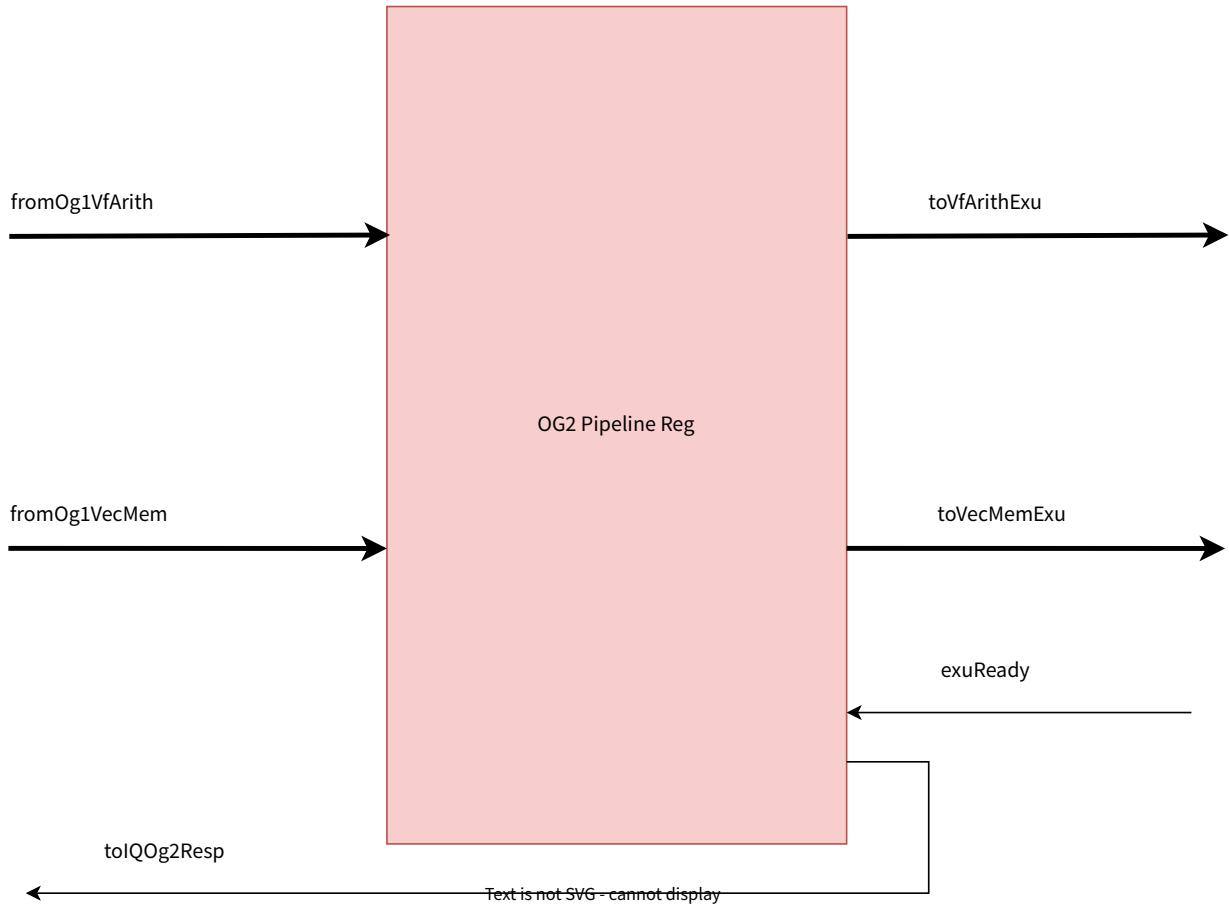


图 8.4: 整体框图

8.2.1.2 接口列表

见接口文档

8.2.2 功能

对于普通标量指令，经过 DataPath 后会直接送到 BypassNetwork 中，通过多路选择生成最终的操作数。对于向量计算指令和向量访存指令，由于读向量寄存器堆的时序比标量更紧张，同时向量执行单元对第一拍数据的时序也有较高的要求，因此在经过 DataPath 后多引入一个 OG2 阶段，然后再进入 BypassNetwork 进行多路选择。

Og2ForVector 模块只进行单纯的打拍，不进行逻辑操作，能否进入 OG2 阶段只需考虑全局的取消。Og2ForVector 模块中含有 OG2 的阶段的流水级寄存器，当 OG1 阶段的指令没有收到 load cancel 或者重定向刷新就可以进入 OG2 阶段。

Og2ForVector 模块的另一个功能是向发射队列发送 OG2 阶段的回应。OG2 阶段没有自身原因的取消，OG2 能否成功发送到后级只需考虑后级能否接收指令。如果后级不能接收，指令无法进入执行单元，需要向发射队列回应 block 状态，告诉发射队列之后需要重新发射指令。如果后级可以接收，对于向量计算指令，指令一定能成

功执行，此时向发射队列回应 success 状态，告诉发射队列可以清空对应的队列项；对于向量访存指令，进入访存执行单元后才能确定能否成功执行指令，这里只向发射队列回应 uncertain 状态，让其保持不动，之后由执行单元发送清空或者重发的回应。

8.3 WbDataPath

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

8.3.1 术语说明

表 8.5: 术语说明

| 缩写 | 全称 | 描述 |
|-----|------------------------|---------|
| v0 | vector mask register | 向量掩码寄存器 |
| vl | Vector length register | 向量长度寄存器 |
| ROB | Reorder Buffer | 重排序缓存 |

8.3.2 子模块列表

表 8.6: 子模块列表

| 子模块 | 描述 |
|----------------------|------------|
| VldMergeUnit | 向量加载数据合并单元 |
| RealWBCollideChecker | 写回仲裁器 |

8.3.3 功能

wbDataPath 模块包含向量加载合并单元 (VldMergeUnit)、写回仲裁器 (RealWBCollideChecker)，主要功能是接收来自各个执行单元的输出信号，对具有向量加载功能的执行单元输出信号进行合并处理，最终输出处理后的信号。

8.3.3.1 数据处理

wbDataPath 接收来自不同执行单元（整数执行单元、浮点执行单元、向量执行单元、访存单元）的输出信号，将这些信号做为输入信号 fromExuPre 使用。从输入信号 fromExuPre 中过滤出向量加载操作 VLoad 相关的信号，并将这些元素组成一个新的元素序列。

根据元素序列的索引例化多个向量加载合并单元模块，并将重定向信号、向量加载操作相关的信号、来自 vstart 寄存器的值赋值给向量加载合并单元模块处理。

注：由于 XiangShan 会刷新流水线，当 vstart 不为 0 且执行向量访存指令时，CSR 中的 vstart 值将做为该向量指令的第一个元素。当发生异常时，写回数据的 vstart 是新的值，所以这个 vstart 不能用作向量访存操作的起始值。

从输入信号 fromExuPre 中过滤出和向量加载合并单元模块拥有相同参数的索引。根据索引使用向量加载合并单元模块处理的结果更新输入信号 fromExuPre 中旧的向量加载操作的数据，得到处理后的来自执行单元的数据 fromExu。

8.3.3.2 仲裁器

由于我们为向量的 v0, vl 寄存器设置了寄存器堆，因此也要对 v0, vl 进行写仲裁。

如果输入信号的 valid 有效，并且整数寄存器堆写使能有效，则整数写仲裁器的输入有效；如果输入信号的 valid 有效，并且浮点寄存器堆写使能有效，则浮点写仲裁器的输入有效；如果输入信号的 valid 有效，并且向量寄存器堆写使能有效，则向量写仲裁器的输入有效；如果输入信号的 valid 有效，并且 v0 寄存器堆写使能有效，则 v0 写仲裁器的输入有效；如果输入信号的 valid 有效，并且 vl 寄存器堆写使能有效，则 vl 写仲裁器的输入有效。

如果是向量执行单元写回整数寄存器堆，则整数写仲裁器的输入延迟一拍。只有不确定延迟的执行单元才需要仲裁器的结果，结果数据可以保留直到仲裁器成功。对于具有确定延迟的执行单元，如果请求在仲裁器中失败，则结果数据将永久丢失。未写回物理寄存器堆的端口始终是就绪的，设置最高优先级的端口始终是就绪的。

8.3.3.3 输出

如果整数写仲裁器的输入有效，并且整数写仲裁器就绪，数据通过整数写仲裁器后输出。如果浮点写仲裁器的输入有效，并且浮点写仲裁器就绪，数据通过浮点写仲裁器后输出。如果向量写仲裁器的输入有效，并且向量写仲裁器就绪，数据通过向量写仲裁器后输出。如果 v0 写仲裁器的输入有效，并且 v0 写仲裁器就绪，数据通过 v0 写仲裁器后输出。如果 vl 写仲裁器的输入有效，并且 vl 写仲裁器就绪，数据通过 vl 写仲裁器后输出。

输出数据发送到 DataPath，在下一拍写入寄存器堆；输出数据发送到派遣模块，设置物理寄存器堆的状态为就绪，用于指令派遣；输出数据发送到调度器中用于写回唤醒。

8.3.3.4 写回 ROB

只有输出成功握手的功能单元，其输出数据能写回 ROB，在 CtrlBlock 中打一拍，并判断写回数据是否刷新流水线、触发异常、trigger fire、重播，之后发送给 ROB，用于 ROB 写回。

8.3.3.5 整体框图

8.3.3.6 接口列表

见接口文档

8.3.3.7 二级模块 VldMergeUnit

8.3.3.7.1 功能

VldMergeUnit 模块主要用于处理向量加载操作的合并逻辑，它接收来自执行单元的写回数据，通过 VldMgu 模块进行合并处理，最终输出合并后的写回数据。该模块使用寄存器 wbReg 存储中间数据，并根据 vlWen 信号选择是否直接使用写回数据或使用合并后的数据。

对于 vl 被 first-only-fault 指令修改的 uop，写回的数据能被直接使用。

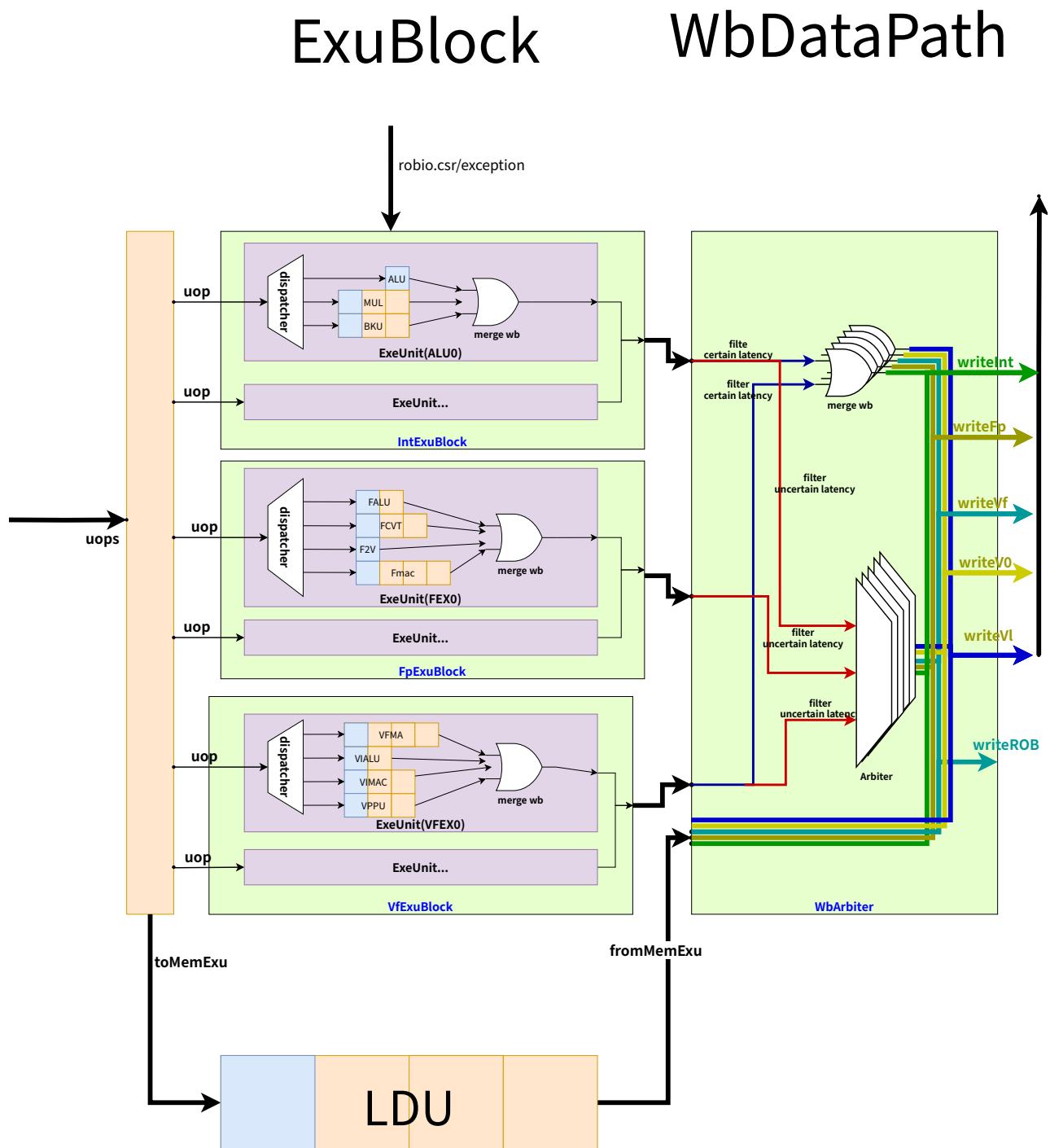


图 8.5: WbDataPath 整体框图

8.3.3.7.2 整体框图

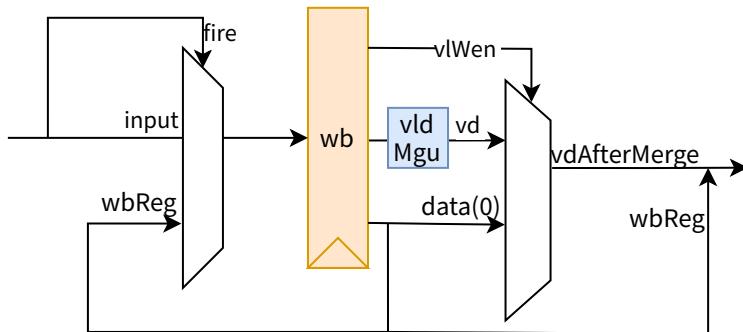


图 8.6: 向量加载功能单元合并模块

8.3.3.8 二级模块 RealWBCollideChecker

8.3.3.8.1 功能

RealWBCollideChecker 模块主要功能是对写回操作的写口进行冲突检查和仲裁，通过将输入端口分组，为每个输出端口实例化一个仲裁器，并将输入和输出端口与仲裁器连接起来，实现写端口的仲裁。

8.3.3.8.1.1 输入输出映射

首先对输入的元素按照端口（port）进行分组，然后对每个分组内的元素按照优先级（priority）进行排序，最终返回分组排序后的映射 `inGroup`。

8.3.3.8.1.2 仲裁器实例化

每个仲裁器负责处理一个输出端口的仲裁，如果映射中包含当前输出端口号 `x`，则实例化一个 `RealWBArbiter` 模块；否则，该输出端口没有对应的仲裁器。

8.3.3.8.1.3 仲裁器输入

遍历每个仲裁器，如果仲裁器不为空，则将仲裁器的输入端口与映射的对应输入分组连接起来。

8.3.3.8.1.4 仲裁器

优先级仲裁器，从多个输入请求中选择一个最高优先级的请求进行响应。

1. 默认选择最低优先级请求，当所有请求均无效时，选择最后一个输入（索引为 `n-1`）作为默认输出。
2. 从次低优先级（`n-2`）到最高优先级（0）遍历输入端口，当某个请求 `i` 的 `valid` 有效时，更新 `chosen` 为 `i`，并将输出数据设置为该请求的数据。优先级规则：索引越小（0 为最高优先级），优先级越高。第一个有效的请求会覆盖后续请求的赋值。
3. 根据所有请求的 `valid` 信号序列生成控制信号 `grant`，表示每个请求是否被授权。如果请求的 `valid` 信号序列为 0，则无请求；如果请求的 `valid` 信号序列为 1，则只有唯一的请求，直接授权；如果请求的 `valid` 信号序列为大于 1，首元素使用原始值，表示最高优先级请求无需判断前置条件，后续元素每个位置的控制信号为！（前置所有请求的或），表示仅当所有更高优先级的请求均未有效时，当前请求被授权。

4. 如果请求被授权，则 ready 信号由下游模块的 out.ready 决定；如果请求无效，则 ready 始终有效。
5. 如果最低优先级请求被授权，则输出有效需要满足最后一个请求有效；否则，输出直接有效（更高优先级请求已被授权）。

8.3.3.8.1.5 仲裁器输出

遍历每个输出端口，如果仲裁器不为空，则将仲裁器的输出端口与对应的输出端口连接起来，并且仲裁器输出端口一直是就绪的；如果仲裁器为空，则输出端口为 0。

8.4 WbFuBusyTable

- 版本：V2R2
- 状态：OK
- 日期：2025/01/20
- commit: xxx

8.4.1 功能

WbFuBusyTable 模块的主要功能是管理和同步不同执行单元的忙碌表与写回端口的交互。

主要流程如下：

1. 来自整数调度器、浮点调度器、向量调度器、访存调度器的忙碌表信息按类型合并为全局忙碌表。
2. 将全局忙碌表与执行单元参数绑定，从参数中获取各写回端口的最大延迟和确定性标志（功能单元执行延迟是否是确定延迟），根据延迟配置生成硬件忙碌表。
3. 根据写回物理寄存器类型判断执行单元是否连接到指定写回端口。
4. 将各执行单元的忙状态合并到写回端口的忙碌表中。
5. 将写回端口的忙碌表分发到各调度器的发射队列中。

8.4.1.1 功能单元忙碌表 FuBusyTable

功能单元忙碌表记录了每个功能单元在不同时间点的忙碌状态，只有确定延迟的功能单元才会有功能单元忙碌表，位宽为功能单元执行的延迟长度，每一位对应一个特定的时间点，“1”表示该功能单元在这个时间点处于忙碌状态，“0”表示空闲。

8.4.1.2 功能单元忙碌表写 FuBusyTableWrite

功能单元忙碌表写主要功能是更新功能单元忙碌表的状态。它接收各种响应信号，如发射队列出队信号 (deqResp)、og0 阶段处理完成后返回的响应信号 (og0Resp)、og1 阶段处理完成后返回的响应信号 (og1Resp)，根据这些信号的有效性和功能单元类型，对忙碌表进行更新。当有指令成功发射时，会根据响应信号中的功能单元类型和延迟信息，将忙碌表中相应位置置“1”，表示该功能单元处于忙碌状态。当 og0 或 og1 阶段指令发射失败时，会根据响应信号中功能单元类型和延迟信息清除掉忙碌表中对应时间点的值，表示该功能单元处于空闲状态。

8.4.1.3 功能单元忙碌表读 FuBusyTableRead

功能单元忙碌表读通过对忙碌表和功能单元类型的处理，生成一个掩码，用于指示哪些指令可以使用相应的功能单元。

8.5 BypassNetwork

8.5.0.1 - 版本：V2R2

- 状态: OK
- 日期: 2025/02/27
- commit: xxx

8.5.1 术语说明

表 8.7: 术语说明

| 全称 | 描述 |
|---------------|------|
| BypassNetWork | 旁路网络 |

8.5.2 子模块列表

表 8.8: 子模块列表

| 子模块 | 描述 |
|---------------|-----------|
| ImmExtractor | 立即数生成模块 |
| UIntExtractor | UInt 解码模块 |

8.5.3 功能

BypassNetWork 位于 DataPath, Exu 流水级之间，主要用于为功能单元提供源操作数。目前 27 个功能单元，总计 71 个源操作数

首先对于可以前递/旁路/两级旁路的源操作数：

根据 Datapath 输入的 ExuSource 信息，由 UIntExtract 提取出独热码，选取可能的来自功能单元的旁路数据，具体现有的唤醒配置见下表。

表 8.9: 现有唤醒配置 1

| Source | Sink |
|--------|--|
| ALU0 | ALU0, BJU0, ALU1, BJU1, ALU2, BJU2, ALU3, BJU3, LDU0, LDU1, LDU2, STA0, STA1, STD0, STD1 |

| Source | Sink |
|--------|--|
| ALU1 | ALU0, BJU0, ALU1, BJU1, ALU2, BJU2, ALU3, BJU3, LDU0, LDU1, LDU2, STA0, STA1, STD0, STD1 |
| ALU2 | ALU0, BJU0, ALU1, BJU1, ALU2, BJU2, ALU3, BJU3, LDU0, LDU1, LDU2, STA0, STA1, STD0, STD1 |
| ALU3 | ALU0, BJU0, ALU1, BJU1, ALU2, BJU2, ALU3, BJU3, LDU0, LDU1, LDU2, STA0, STA1, STD0, STD1 |
| LDU0 | ALU0, BJU0, ALU1, BJU1, ALU2, BJU2, ALU3, BJU3, LDU0, LDU1, LDU2, STA0, STA1, STD0, STD1 |
| LDU1 | ALU0, BJU0, ALU1, BJU1, ALU2, BJU2, ALU3, BJU3, LDU0, LDU1, LDU2, STA0, STA1, STD0, STD1 |
| LDU2 | ALU0, BJU0, ALU1, BJU1, ALU2, BJU2, ALU3, BJU3, LDU0, LDU1, LDU2, STA0, STA1, STD0, STD1 |

表 8.10: 现有唤醒配置 2

| Source | Sink |
|--------|------------------------------|
| FEX0 | FEX0, FEX1, FEX2, FEX3, FEX4 |
| FEX2 | FEX0, FEX1, FEX2, FEX3, FEX4 |
| FEX4 | FEX0, FEX1, FEX2, FEX3, FEX4 |

其中作用于向量浮点以及访存单元之间的二级旁路，目前暂时取消。

其次对于源操作数是立即数部分，根据来自 datapath 的立即数信息，由 ImmExtractor 组装生成 64bit 立即数。

最后根据 datapath 中 data source 信息，从所有可能的数据来源（前递，旁路，二级旁路，v0，寄存器堆，立即数，regcache，0 号寄存器）中选取源操作数，传入功能单元。

另外，对于跳转功能单元，部分 pcoffset 逻辑也放在旁路网络中，立即数信息同样由 ImmExtractor 组装生成。

具体设计见图 8.7：

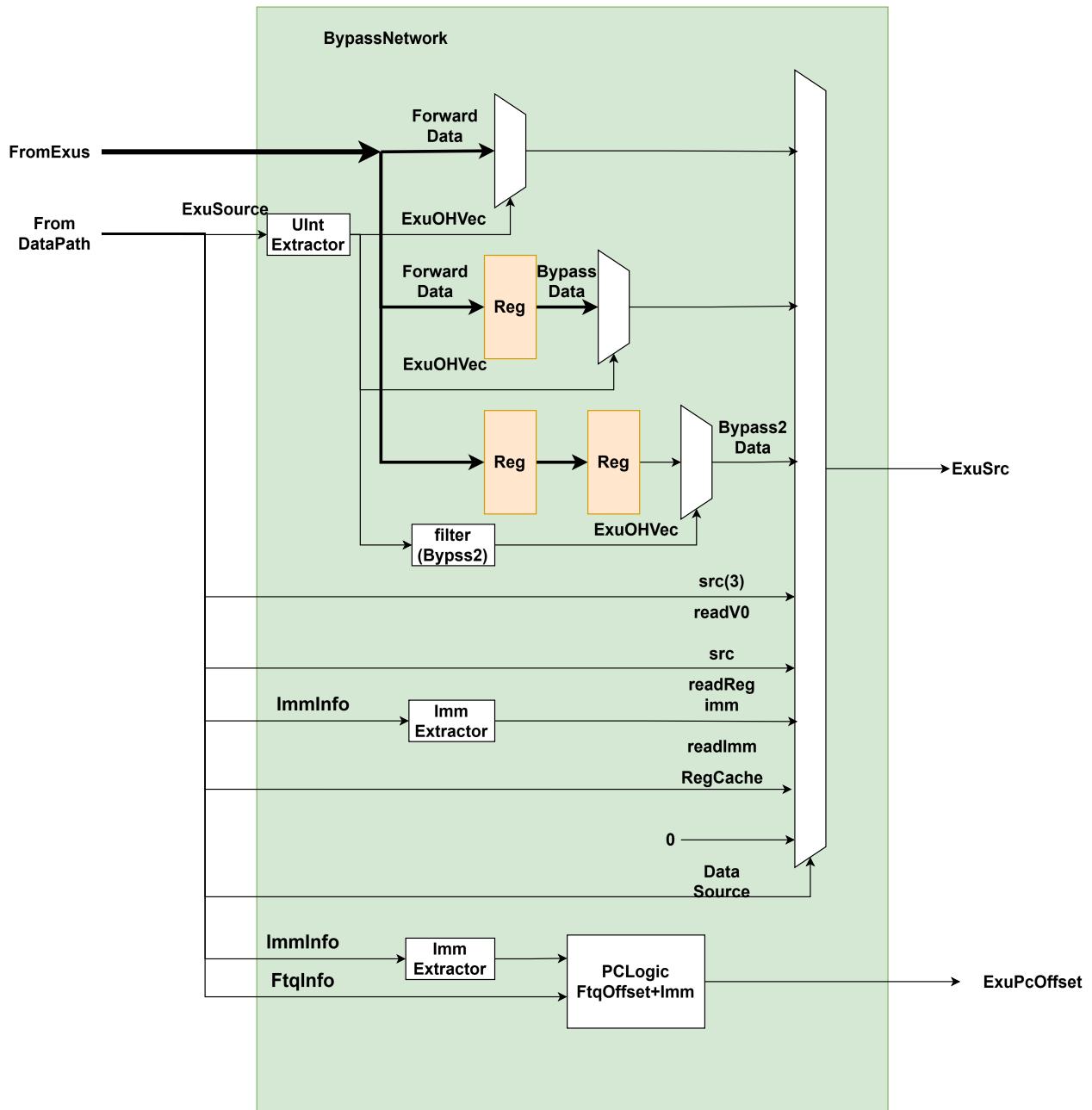


图 8.7: BypassNetwork

8.5.4 模块设计

8.5.4.1 二级模块 ImmExtractor

该模块负责生成 64bit 立即数，首先根据下述映射，将立即数映射为 32bit 形式，之后再对结果进行符号拓展成 64bit 立即数。

表 8.11: 立即数映射

| SelImm | ImmUnion | Immlen | extracter |
|--------------|----------|--------|---------------------------------|
| IMM_I | I | 12 | SignExt(imm(len - 1, 0), 32) |
| IMM_S | S | 12 | SignExt(imm, 32) |
| IMM_SB | B | 12 | SignExt(Cat(imm, 0.U(1.W)), 32) |
| IMM_U | U | 20 | Cat(imm(len - 1, 0), 0.U(12.W)) |
| IMM_UJ | J | 20 | SignExt(Cat(imm, 0.U(1.W)), 32) |
| Z | Z | 22 | imm |
| IMM_B6 | B6 | 6 | ZeroExt(imm, 32) |
| IMM_VSETVLI | VSETVLI | 11 | SignExt(imm, 32) |
| IMM_VSETIVLI | VSETIVLI | 15 | SignExt(imm, 32) |
| IMM_OPIVIS | OPIVIS | 5 | SignExt(imm, 32) |
| IMM_OPIVIU | OPIVIU | 5 | ZeroExt(imm, 32) |
| IMM_LUI32 | LUI32 | 32 | imm(31, 0) |
| IMM_VRORVI | VRORVI | 6 | ZeroExt(imm, 32) |

8.5.4.2 二级模块 UIntExtractor

该模块服务于 toExuOH 功能：负责将压缩成 UInt 的源操作数旁路来源的 exuidx 解码为 one-hot 形式。

记录源操作数旁路来源的 exusource 中的功能单元标号在发射阶段经历了两次压缩：

- 首先将标志 27 个功能单元的独热码，根据旁路唤醒可能的来源，压缩为 7/3 个功能单元的独热码
- 其次将 7/3 个功能单元的独热码，压缩为 UInt 形式，共 3/2bit UInt

因此在旁路网络中需要对来自 DataPath 的压缩后 exusource 进行两次解压缩：

- 首先将 3/2 bit 的 exusource 解压为独热码
- 其次将压缩后的独热码，根据当前功能单元可能的唤醒来源，解压为标志 27 个功能单元的独热码

对于第一步解压操作，在 toExuOH 中只需要简单通过移位（唤醒源与源操作数是一一对应的）完成 UIntExtractor 负责第二步解压操作，完成下述映射：

表 8.12: 唤醒源独热码映射 (1)

| EncodedExuOH | ExtractExuOH |
|--------------|--------------|
| ALU0(0) | 0 |
| ALU1(1) | 2 |
| ALU2(2) | 4 |

| EncodedExuOH | ExtractExuOH |
|--------------|--------------|
| ALU3(3) | 6 |
| LDU0(4) | 20 |
| LDU1(5) | 21 |
| LDU2(6) | 22 |

表 8.13: 唤醒源独热码映射 (1)

| EncodedExuOH | ExtractExuOH |
|--------------|--------------|
| FEX0(0) | 8 |
| FEX1(1) | 10 |
| FEX2(2) | 12 |

9 Schedule and Issue 调度与发射

9.1 Scheduler

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/15
- commit: xxx

不同种类的 Scheduler 及其策略的描述等。

Scheduler 模块的主要作用是将 IQ 包装起来连接 Dispatch 模块和 DataPath 模块，共有 intScheduler、fpScheduler、vfScheduler、memScheduler 四个，分别对应整数、浮点、向量、访存（包括标量访存和向量访存），特别需要注意的是在 memScheduler 内对 sta 和 std 两种 IQ 的 ready 做了与操作后传给了 Dispatch，Dispatch 会根据 IQ 的 ready 状态回 valid，如果 IQ 不 ready 那 Dispatch 给 IQ enq 的 valid 会拉低。

9.2 IssueQueue

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

9.2.1 设计规格

- 支持 4 类不同的发射队列模块以适配标量整型，向量浮点，标量访存和向量访存
- 支持两个入队口两个出对口
- 支持推测唤醒信号产生
- 支持推测唤醒信号寄存器复制
- 支持指令写回冲突提前判定
- 支持就绪指令中选择最老指令发射

9.2.2 功能

发射队列模块作为处理器乱序调度的起点，连接前一级 Dispatch 流水级和后一级 DataPath 流水级。在超标量乱序处理器中，为了实现指令的正确乱序执行，需要正确处理器指令间的依赖关系，描述指令是否能够正确执行的关键就是每条指令源操作数的就绪情况，当源操作数依赖的前面的指令执行完成后，该源操作数才进入就绪状态。IQ 接收从 Dispatch 阶段分派而来的至多两条指令，指令的源操作数可能还未就绪，指令会暂存在 IQ

内部，IQ 实时监听唤醒信号，唤醒信号会将其对应的源操作数由未就绪置为就绪。每个周期，IQ 内部会选出最多两条操作数都就绪的指令，并遵循最老最优先策略，发往后续的 DataPath 流水级。发射队列通过上述方法，保证了指令乱序执行时，所有指令间依赖关系均能得到保证，并尽可能提升乱序调度的性能。

9.2.2.1 指令入队

4 类不同的发射队列的指令入队逻辑基本一致，仅因部分信号的不同而有些许差异。发射队列内部实例化了 Entries 模块负责指令的存储，一般情况下，发射队列支持两个入队口，即每周期至多从前一流水级接收 2 条有效指令，故与之对应的，Entries 模块同样支持两个入队口。指令入队过程，即通过 IQ 的输入端进入，选出关键信号，送至 Entries 的输入端，这个过程中，诸如 robIdx, fuType 等指令自带信号不做额外处理直接接入；而 srcState 等指示指令状态的信号，会在进入 Entries 之前通过部分组合逻辑进行初始化赋值。各信号详细信息参见 Entries 接口文档。本发射队列支持指令入队的同时进行唤醒，基于时序考虑，唤醒逻辑并未直接在指令进入 Entries 之前实现，而是采用将输入唤醒信号送入 Entries 后，同步打拍后再唤醒的方法。

9.2.2.2 指令年龄关系维护

为了实现指令发射选择的最老最优先策略，在发射队列内，需要每个周期对存在 Entries 内的指令进行指今年龄的记录和处理。发射队列内部实例化了若干个 AgeDetector 模块进行这部分功能的实现。对应 3 类不同的 Entry，发射队列需要实例化至多 3 个 AgeDetector。每个年龄矩阵可同时接收多个出队端口的年龄查询，并返回查询中最老的一项。AgeDetector 每周期接收 Entries 反馈的 3 类 entry 的入队状态，负责维护全部指令的年龄关系，直观来说，当一条指令入队时，它的年龄必然是全部指令中最年轻的；发射队列通过 Entries -> AgeDetector 的信号传递实现指今年龄关系的维护，并在指令发射选择阶段通过读取 AgeDetector 完成最老最优先策略的运用。

9.2.2.3 指令发射选择

发射队列将 Entry 分为至多 3 类，并支持了指令在 Entry 之间存储位置的转移，3 类 Entry 之间有着严格的年龄关系，所以指令发射选择，也分别针对 3 类 Entry 进行并行选择，最后再进行 3 选 1 选择最老的指令发射。基于时序考虑，为了满足发射队列两出队端口的需求，设计优先满足第一个出队端口的需要，功能实现为：依据 EnqEntry/SimpleEntry/ComplexEntry 三类 Entry 对应的三个 Detector，分别选出三条最老能发射指令，而后依据三类 Entry 间严格的年龄关系，按照 Complex > Simple > Enq 的优先级顺序最终选出一条，送往第一个出队端口；对于第二个出队端口，取决于出口 fu 的配置：如果两个出口的 fu 不同，那么他们可出队的指令不会重叠，第二个口也按第一口的方法选出一条最老指令；如果两个出口的 fu 相同，就可能重叠，因此屏蔽掉第一个口的选择结果后，“随机”选一条有效指令。目前的 IQ 配置中，两个口的 fu 都是不同的（或者就只有一个出口），因此不存在“随机”选的情况，就不再赘述“随机”的具体流程。

9.2.2.4 推测唤醒信号产生

发射队列负责管理指令，不仅包括对指令何时发射的管理，还包括告知其他指令何时发射的管理，后者即是通过推测唤醒实现的。在一般情况下，如果一条指令的源操作数依赖于前一条指令的写回值，那么只有前一条指令写回时，当前指令的源操作数才能被置为就绪状态。为了提升指令乱序执行的性能，实际上如果前一条指令是固定执行延时的指令，那么当它从发射队列发射出去时，就能够确定其何时写回，相应的便可以在某个时刻由发射队列发出推测唤醒信号，只要保证被推测唤醒的指令，取得源操作数的时刻不早于前一条指令得到结果的时刻，便可通过 forward、bypass 等方式加速指令的乱序执行。在发射队列中，对于非访存 IQ 负责产生推测唤醒信号的模块就是 WakeupQueue，指令被选择发射的同拍，也会进入 WakeupQueue 中，根据其执行延迟，进入不

同的移位 pipeline, 0 lat 则一拍后产生推测唤醒; 2 lat 则三拍之后, 以此类推。通过这种方式, 可以实现推测唤醒信号的产生。对于访存 IQ, 其唤醒信号使用访存 IQ 独有的 loadWakeUp 接口从访存单元传入, 将其打一拍后, 当做 IQ 自己的唤醒信号, 再走和前面 WakeupQueue 相同的接口广播给其他 IQ。

9.2.2.5 写回冲突的提前判断

写回冲突分为两个部分。第一部分, 是 IQ 出口自身的写回冲突。发射队列每个出队端口对应一个 EXU, 每个 EXU 里可能有一组 Fu, 这一组 Fu 的执行延迟不尽相同, 而每个 EXU 只有一个写回端口 (指写回 rob 的写回端口, 与寄存器堆的写口不同, 一或多个 EXU 写回口共用一个寄存器堆写口)。比如 Alu 与 Mul 的 Fu 组合, 会产生 0 lat 和 2 lat 的写回冲突问题, 如果一条 0 lat 指令在 2 lat 指令发出两拍后发射, 就会同时执行完成, 产生 Fu 写回冲突。为了避免这类情况的发生, 发射队列内部实例化了 fuBusyTable 模块进行 Fu 冲突的判断。fuBusyTable 以出队端口为基本单位, 记录各自的出队端口每周期出队的指令及各自的反馈信号, 以此修改记录值, 后续指令的选择发射也需要参考此模块的记录值情况, 以此避免 Fu 写回冲突。第二部分, 是寄存器堆的写回冲突。由于寄存器堆的写口有限, 多个 EXU 的写回端口可能共用一个寄存器堆的写口, 因此发射队列之间可能发生写口共用, 此时同样可能产生写口冲突的问题。类似地, 发射队列中实例化了 intWbBusyTable 和 vfWbBusyTable, 每当指令发射时, 便产生相应的写逻辑, 送至外部的 WbFuBusyTable 模块中将写口相同的取或处理, 得到最终的 WbFuBusyTable; 指令发射选择时, 则根据写口从外部读取 WbFuBusyTable 送入 IQ 内部作为选择参考。

9.2.3 整体框图

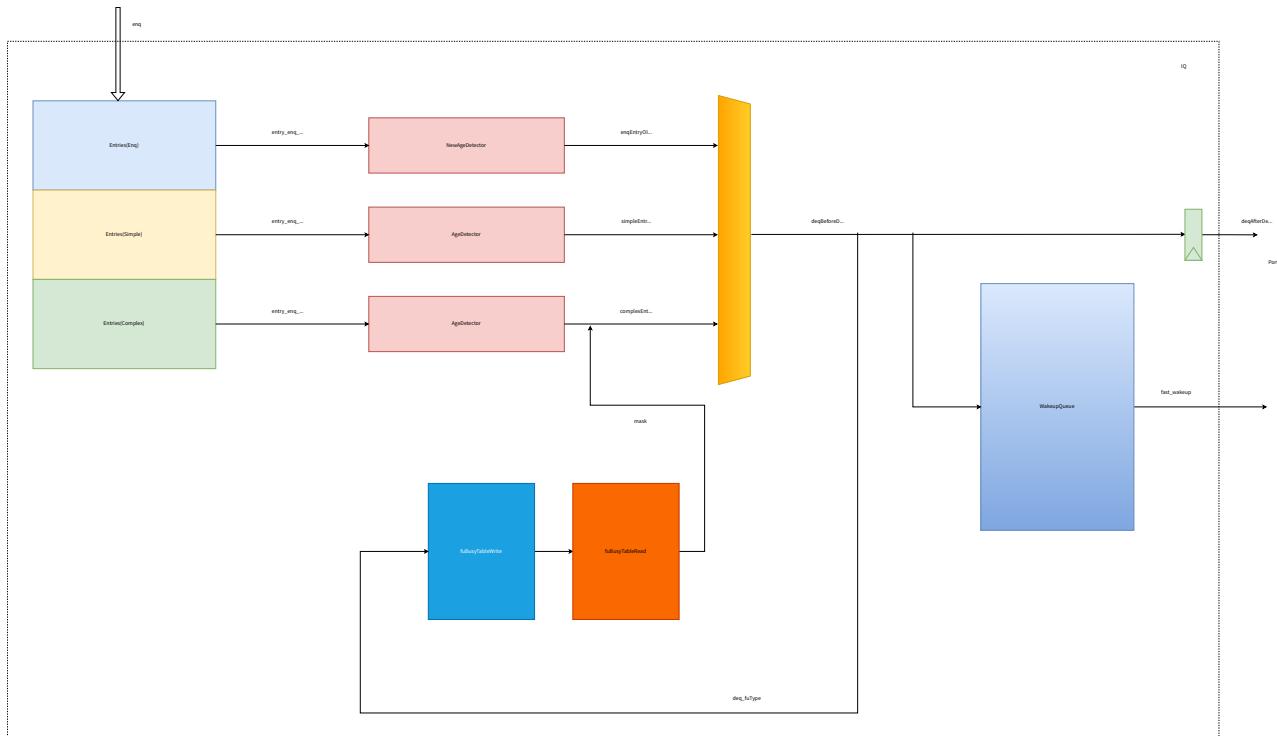


图 9.1: 示意图

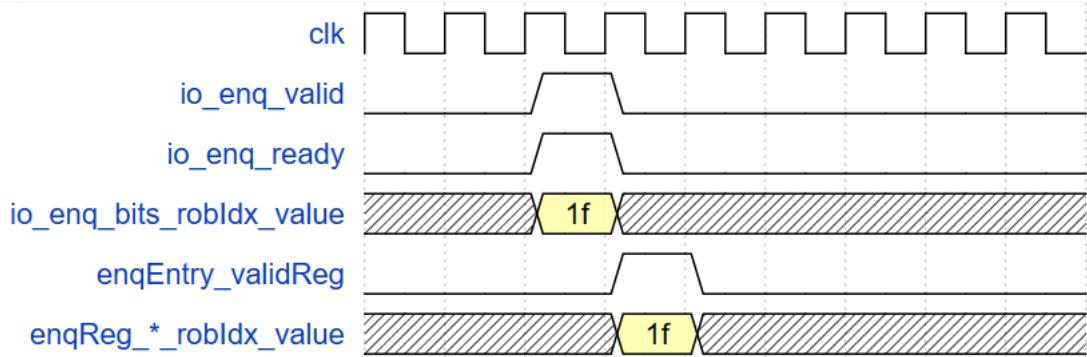


图 9.2: 示意图

9.2.4 接口时序

9.2.5 二级模块 WakeupQueue

控制各个 IQ 发出推测唤醒信号的关键模块，作为推测唤醒的唤醒源且非访存的 IQ 的各出队端口各自对应一个，模块内部由若干条流水线构成，流水线的条数和出队端口对应的 Fu Latency 直接相关，出队端口对应的 Fu 有几种 Lat，就有几条相应的流水线。

9.2.5.1 整体框图

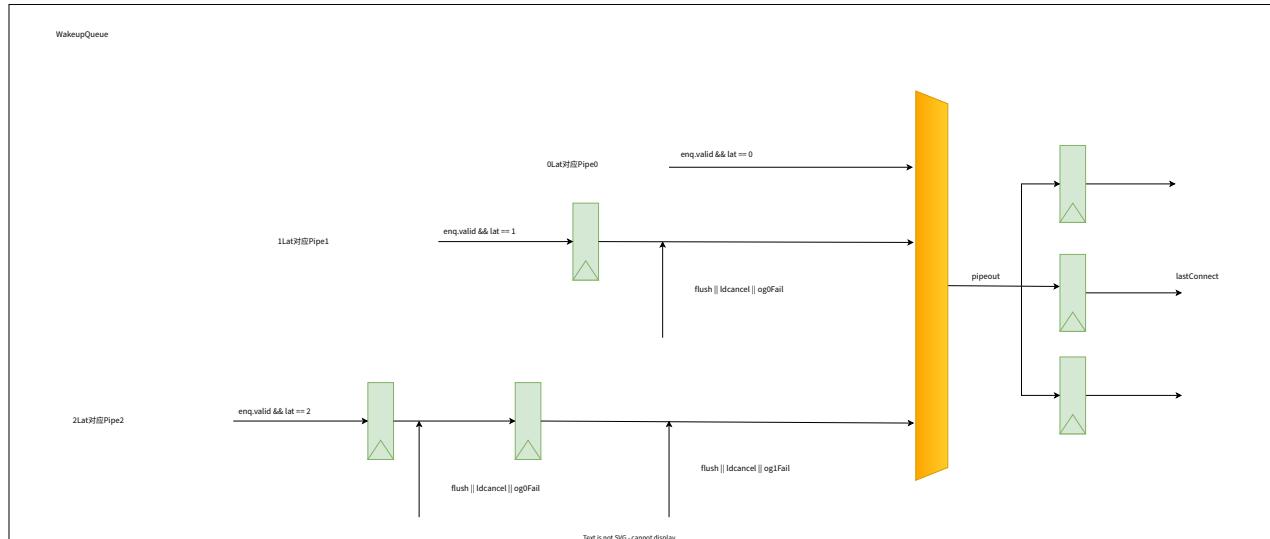


图 9.3: 示意图

9.2.5.2 接口时序

9.2.6 二级模块 AgeDetector

此模块为年龄矩阵模块，维护发射队列中各个 Entry 内指令的新旧顺序，发射队列的 Entry 至多三类，对每类 Entry，它们的 AgeDetector 模块都是独立的。此模块使用矩阵寄存器标识 Entry 间的年龄关系，矩阵的行列均和 Entry 的个数一致，下面以 6 项的 SimpleEntry 为例：

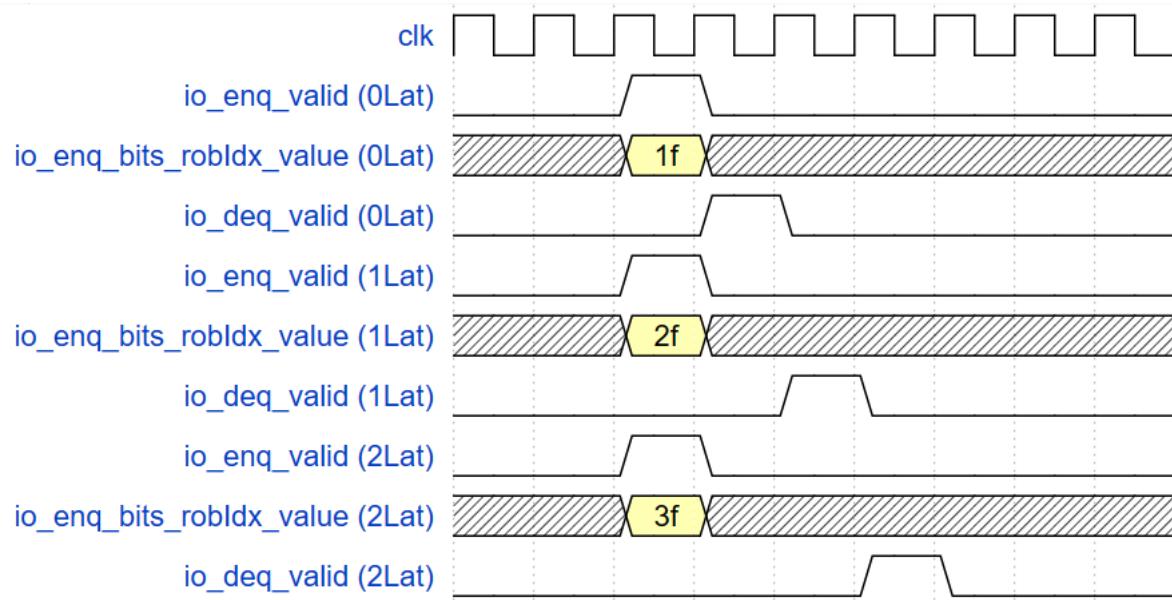


图 9.4: 示意图

9.2.6.1 整体框图

9.3 IssueQueueEntries

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

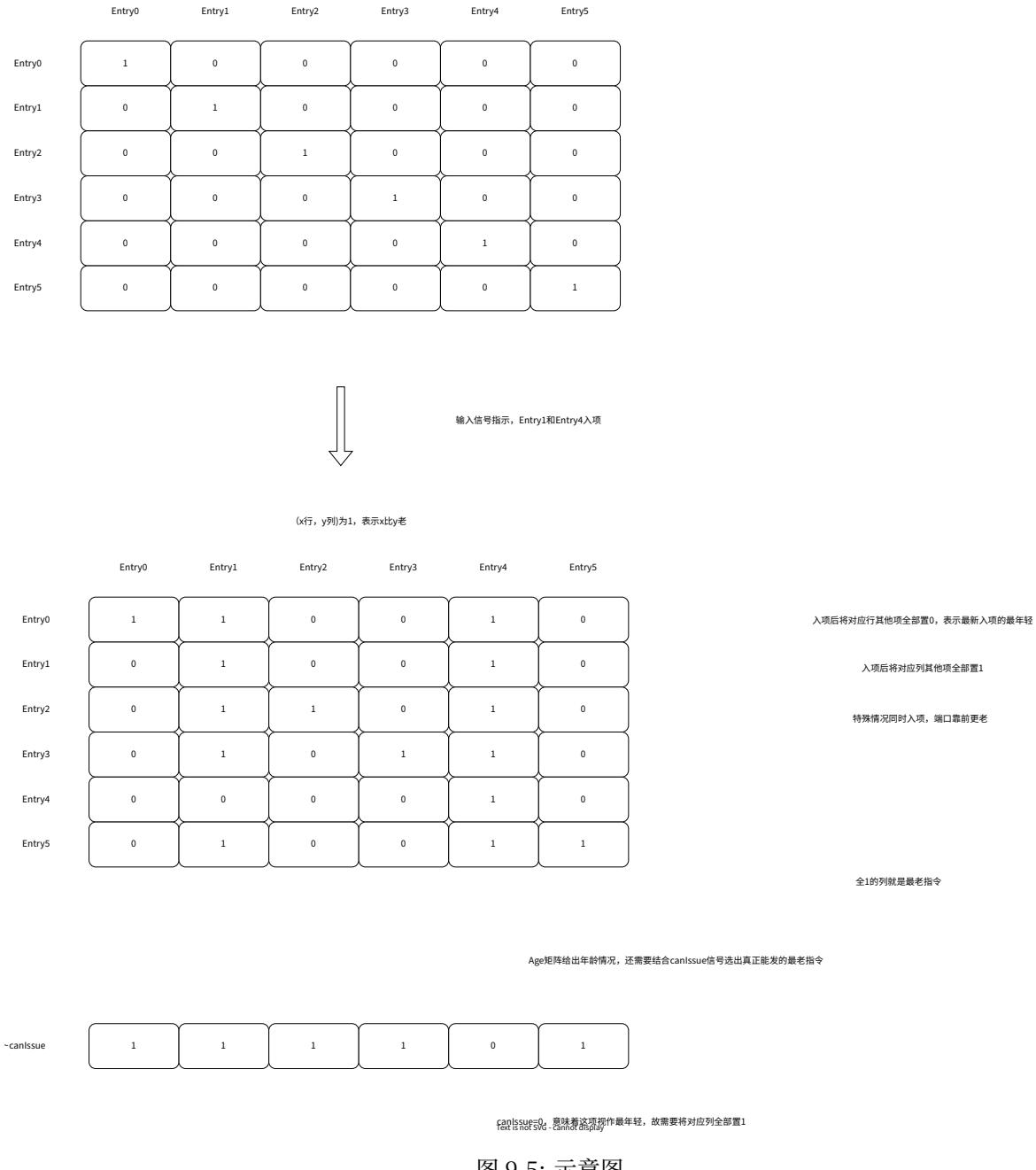


图 9.5: 示意图

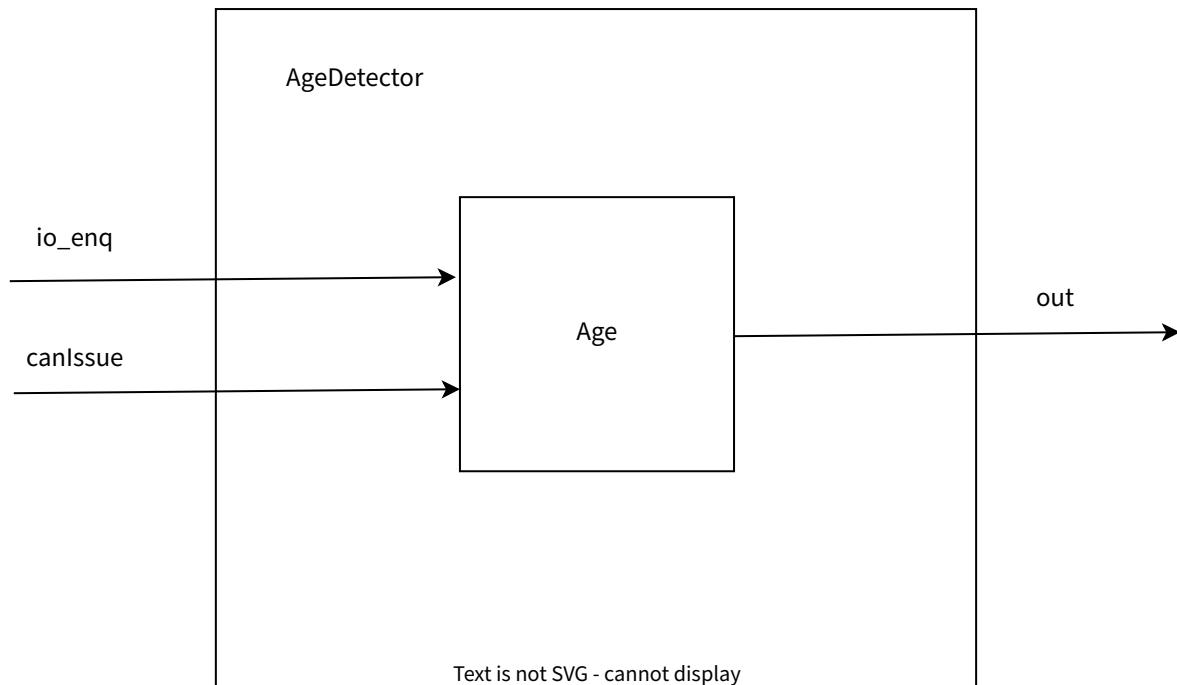


图 9.6: 示意图

9.3.1 术语说明

表 9.1: 术语说明

| 缩写 | 全称 | 描述 |
|----|------------|------|
| IQ | IssueQueue | 发射队列 |

9.3.2 设计规格

- 支持三类发射队列项：EnqEntry、SimpleEntry 和 ComplexEntry
- 支持双端口读写
- 支持写回唤醒和推测唤醒
- 支持 EnqEntry 直接出队
- 支持 Entry 间指令转移
- 支持唤醒取消反馈

9.3.3 功能

9.3.3.1 总体功能

Entries 是发射队列中存放 uop 的模块，它内部有多个 entry 模块，每个 entry 可以存放一条 uop。这些 entry 可以分为两大类：与发射队列入队端口对应的 EnqEntry，以及数量较多的 OthersEntry。Entries 整合所有 entry 的发射和状态信息，传给发射队列控制逻辑；接收控制逻辑的选择结果，输出要发射 uop 的全部信息。Entries 接受来自 IQ（自身 IQ 或其它 IQ 的快速唤醒）和 WriteBack（写回唤醒）的唤醒信号、来自 datapath 等的取消信号（og0Cancel、og1Cancel 等），接受并整合发射后的反馈信号，送给每个 entry。Entries 还负

负责 entry 之间的转移逻辑。EnqEntry 接收 IQ 入口的 uop，如果 OthersEntry 就绪，它会以一定规则转移到 OthersEntry 中，EnqEntry 支持当拍同时转移出上一个 uop 并入队下一个 uop，实现无缝衔接。在高级的发射队列配置中，OthersEntry 又分为两类：SimpleEntry 与 ComplexEntry。Entries 也负责控制从 SimpleEntry 到 ComplexEntry 的转移策略。

9.3.3.2 转移策略

ComplexEntry 是最终的项，不可转移；SimpleEntry 可向 ComplexEntry 转移；EnqEntry 可向 ComplexEntry 也可向 SimpleEntry 转移。只有没有被发射过的项才可被转移，如果发射过的项反馈发射失败了，会清掉发射过的标记，就变成可被转移的项；如果发射过的项反馈发射成功了，该项会变成无效项，不需要再转移了。EnqEntry 到 OthersEntry 的转移逻辑。EnqEntry 优先转移到 ComplexEntry，其次转移到 SimpleEntry。转移只能是全或者零，要么全部转移到 ComplexEntry，要么全部转移到 SimpleEntry，要么就不转移。EnqEntry 转移到 ComplexEntry 的条件是，ComplexEntry 里有足够的空闲项，同时 SimpleEntry 全空；否则只能向 SimpleEntry 转移。SimpleEntry 到 ComplexEntry 的转移逻辑。每个周期，SimpleEntry 可至多转移 num_enq（相当于 EnqEntry 数量）项到 ComplexEntry，只要 ComplexEntry 每有一个空位，就可以转移过去一条。SimpleEntry 转移的优先级比 EnqEntry 更高。SimpleEntry 转移次序有强要求，更老的项更优先转移。各项年龄次序通过查询 IQ 中的年龄矩阵得到。

9.3.3.3 发射与出队

Entries 收集各个 entry 的 valid 与 canIssue 信号，传递给 IQ，IQ 返回各个出队口选择出要出队的 entry 位置 deqSelOH，以及各个出口能否接受的 deqReady 信号，现在 deqReady 是常数值，一直拉高。两者同时有效时，认为该 entry 将出队，把 deqSel 信号传给该 entry。收到 deqSel 后 entry 还不能清空，只是标记为已发射状态，记录发射的端口和发射后经过的周期，之后还要看发射后后续返回的 resp 信号，只有收到发射成功的 resp 后才能清空。Entries 负责汇总各个 resp，将对应的 resp 传给 entry。非访存 IQ 的 entry 的 resp 只有 og0resp 和 og1resp，根据各 entry 的出队端口和发射后经过的周期来选择 resp。Entry 和 robIdx 和 resp 的 robIdx 一致时，选择对应的 resp 传给 entry。访存 IQ 的 resp 较多，不同访存 IQ 的 resp 也有所区别，需要比对 lqid 和 sqid 来选择 resp。发射时还要将选中 entry 的 uop 信息传给 IQ，因为时序原因不直接使用 deqSelOH 来选择。deqSelOH 的各 bit 形成时间差别较大，为了能减少延迟，IQ 会传进各阶段选择结果，包括 enqEntryOldest、simpEntryOldest、compEntryOldest 的结果。用三组信号分别选出对应的出队 uop，再按 comp、simp、enq 的优先级选出最终出队 uop。

9.3.3.4 唤醒与取消

Entries 不处理唤醒逻辑，只将唤醒和取消信号传入所有 entry。由于时序原因，Entries 也负责处理当拍的取消逻辑。取消的来源延迟较长，如果正常经过唤醒、取消，再给 IQ 进行出队选择，时序就太差了。因此我们将只经过当拍唤醒的结果给 IQ 进行出队选择，然后由 Entries 单独计算当拍取消，最后再对各个出口选出要出队的 uop 进行取消判断。

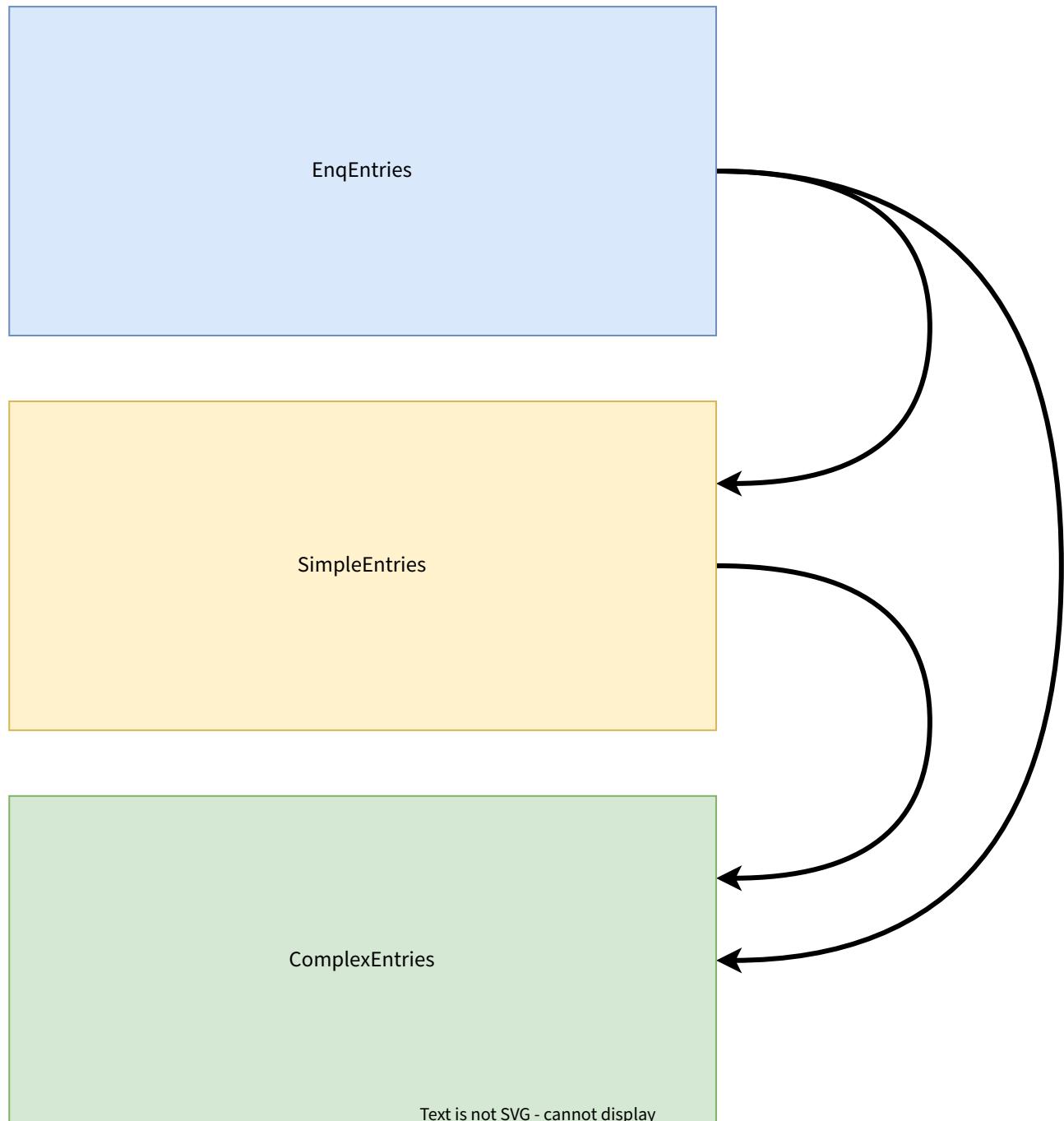


图 9.7: 示意图

9.3.4 整体框图

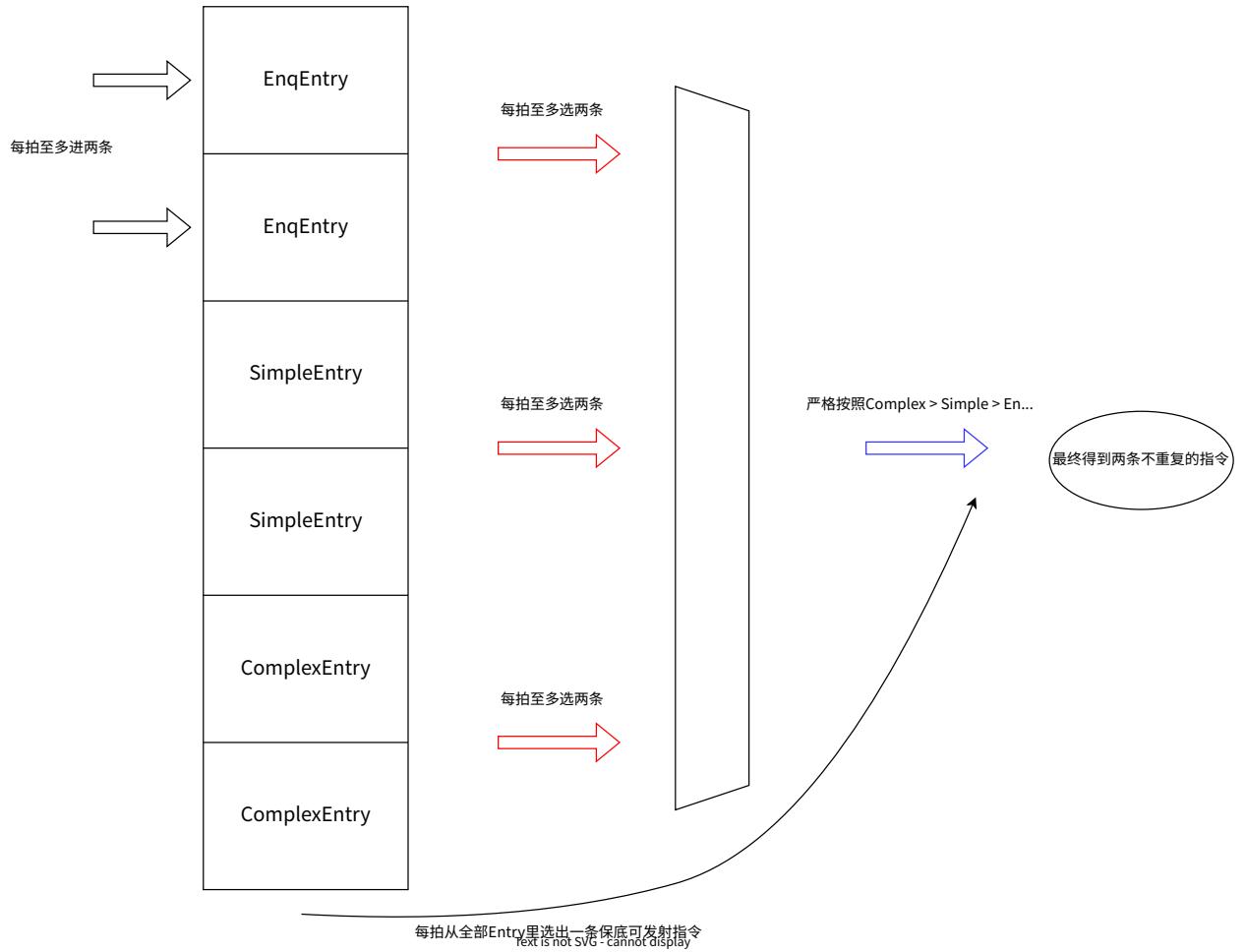


图 9.8: 示意图

9.3.5 接口时序

io_* 信号组为 IQ 入队指令，每拍至多两条，同时伴随着可能的唤醒信号。基于时序考虑，对入队指令被同时唤醒这种情况的处理，选择将 wakeup 打一拍，见图中 enqDelay_wakeup，为了对上拍，这部分唤醒会类似推测唤醒的 bypass 时序，影响 srcStateNext，即影响 canIssueBypass，类似 ComplexEntry 的当拍唤醒当拍发射。

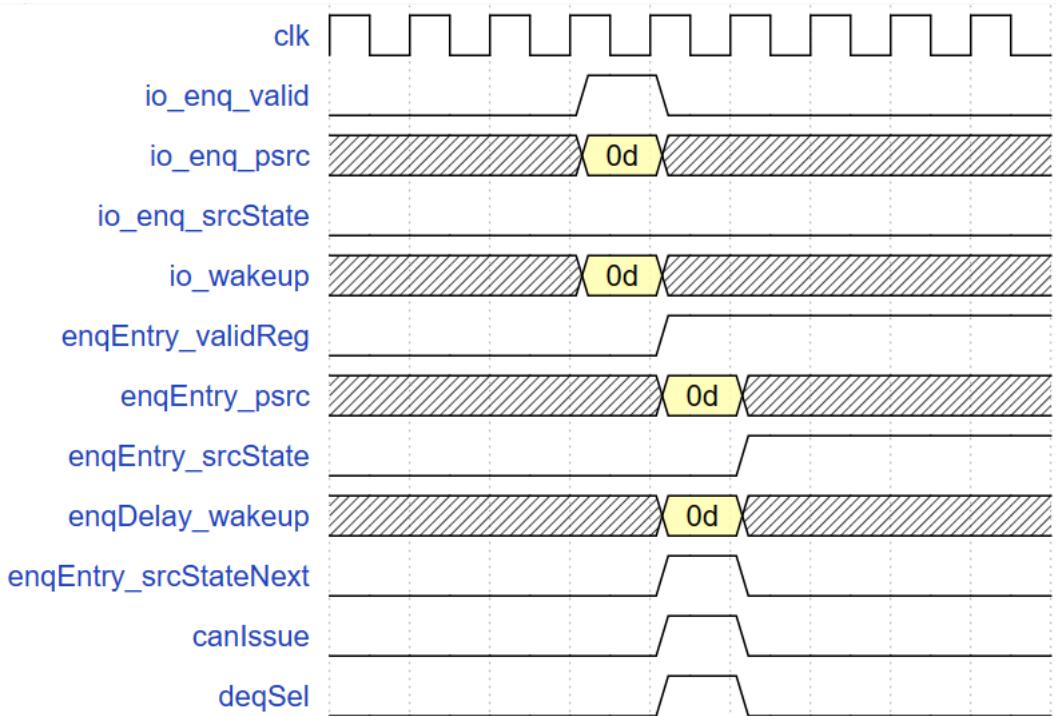


图 9.9: 示意图

9.3.6 二级模块 EnqEntry & OthersEntry

9.3.6.1 功能

EnqEntry 和 OthersEntry 功能基本一致, EnqEntry 因为直接对接入队端口, 会多一层入队唤醒的处理, 其余功能一致, 因此放到一起描述。Entry 有这几个最重要功能: valid、canIssue、issued、status。Valid 表示 entry 是否有效, 有 uop 进入 entry 时, 将 enq 中的 uop 信息写入寄存器, 将 valid 置为有效。当 flush、tranSel 有效或者 issueResp 表示发射成功这三个条件之一成立时, entry 清空, valid 置无效。Issued 记录 uop 是否发射, 当 deqSel 有效时记为已发射; 收到 issueResp 失败或者有操作数被 cancel 而不再就绪时, 记为未发射。当所有源操作数就绪, 且是未发射状态, 将 canIssue 输出为有效。Status 是描述源操作数状态的一系列信息, 包括操作数类型 srcType、状态 srcState、数据来源 dataSources、唤醒该操作数的 load 信息 srcLoadDependency、唤醒该操作数的 exu 信息 srcWakeUpL1ExuOH、唤醒后周期计数器 srcTimer。wakeUpFromWB 和 wakeUpFromIQ 传递要唤醒的 pdest 和寄存器类型 xp、fp、vp, 如果 pdest 号与 entry 操作数的寄存器号一致, 寄存器类型也一致, 该操作数被唤醒, 记为就绪状态。og0Cancel、og1Cancel 传递要取消的 exu 号。对于 ogCancel 如果要取消的 exu 与唤醒该操作数的 exu 一致, 且 srcTimer 对应发出的流水级延迟, 则取消该操作数。对于 ldCancel, 如果要取消的 ld 流水级与 srcLoadDependency 一致, 取消该操作数。当同一操作数唤醒和取消同时达到时, 取消的优先级更高。Entry 向外输出的源操作数状态信息有立刻和延迟两种, 对应快速和慢速唤醒。立刻指源操作数状态信息从寄存器获取后, 经过上述唤醒与取消更新状态之后当拍立刻输出; 延迟指源操作数状态信息经过上述唤醒与取消更新状态之后, 写回寄存器, 下一拍才能从寄存器输出。WB 唤醒总是慢速的, 而 IQ 唤醒可配置快速和慢速。配置为快速的称为 ComplexEntry, 配置为慢速的称为 SimpleEntry。EnqEntry 理论上也可配置, 但实际中总是快速的。EnqEntry 区别于 OthersEntry 的地方在于要多一次入队唤醒。入队时的唤醒与取消因为时序原因不好在写入 EnqEntry 前做, 因此延迟到写入 EnqEntry 下拍开头, 先使用延迟的唤醒与取消信号(enqDelay)更新寄存器直出的状态, 再进行正常的唤醒与取消。注意入队唤醒只在 uop 进入 EnqEntry 的第一拍进行, 此后都是直接使用寄存器直出的状态。

总结: 1. Entry 是 IssueQueue 内部存储 uop 关键信息的结构, 可类比 RS。2. 昆明湖的整数 IssueQueue 标准设计规格下有 24 项 Entry。3. Entry 按照行为逻辑分为三类: EnqEntry, SimpleEntry 和 ComplexEntry。4. 2 个 EnqEntry, 作为入队端口, 每周期进入 IQ 的两条指令只能存入这里。5. 6 个 SimpleEntry + 16 个 ComplexEntry。

9.3.6.2 整体框图

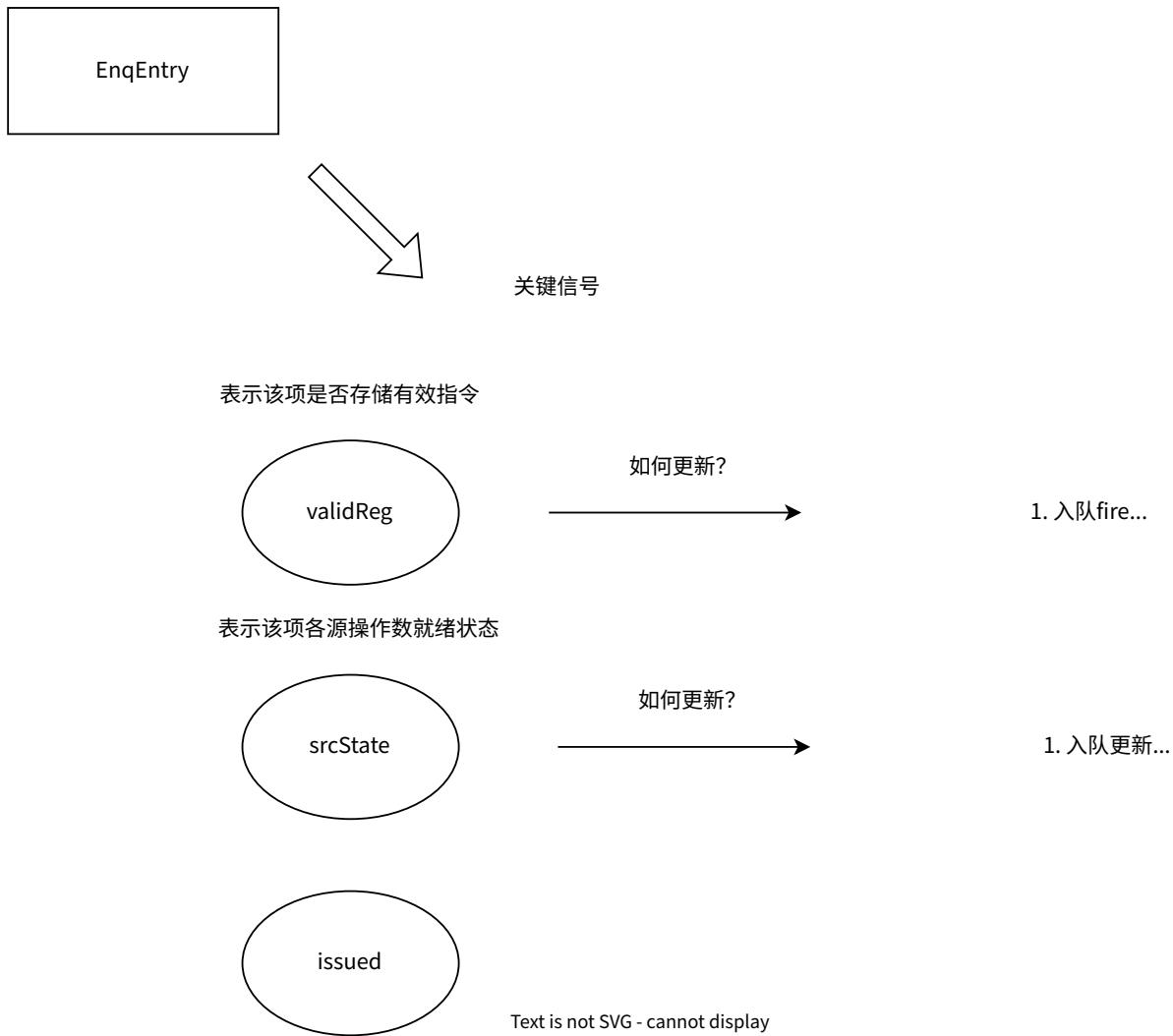


图 9.10: 示意图

imm 存放立即数, payload 存放指令原始信息, entry 不对其进行处理。

srcStatus 指示各项 uop 各源操作数的状态。issued 指示 uop 的发射状态, 因为发射可能成功可能失败, 发射直到成功才能修改 validReg, 所以使用 issued 来标记在不在发射途中。

issueTimer 和 deqPortIdx 的存在是为了适配 entry 转移机制, 指令发射出去后, 要经过 OG0 和 OG1 两级, 只有通过 OG1 进入 EXU 的 uop 才算发射成功, 中途如果失败了就需要告知 IQ 重发, 无转移机制的情况下, uop 可以通过 entryIdx 进行定位; 有了转移机制后, uop 发射出去后, 可能下一拍就转移至其他位置, 这样 OG0/1 的 resp 信号就难以定位, 所以增加 issueTimer 和 deqPortIdx 信号, 一旦 uop 发射出去, 就修改 issueTimer 并每拍自增, deqPortIdx 记录其从哪个出队端口发出, 根据上图的时序关系, OG0 和 OG1 的 resp 只需要识别各 Entry 内部的这两个信号值, 即可定位 uop。

唤醒 -> 修改 srcState srcWakeupL1ExuOH -> 标记推测唤醒信号来自哪个 exu

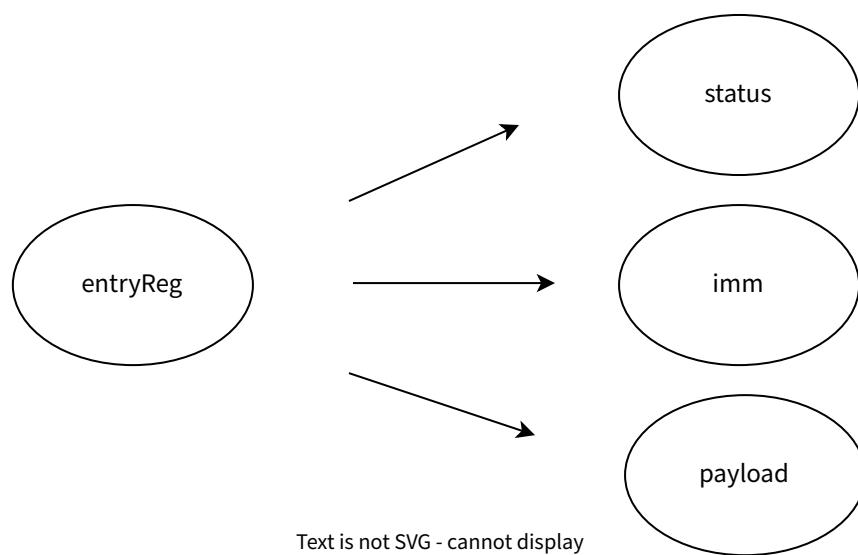


图 9.11: 示意图

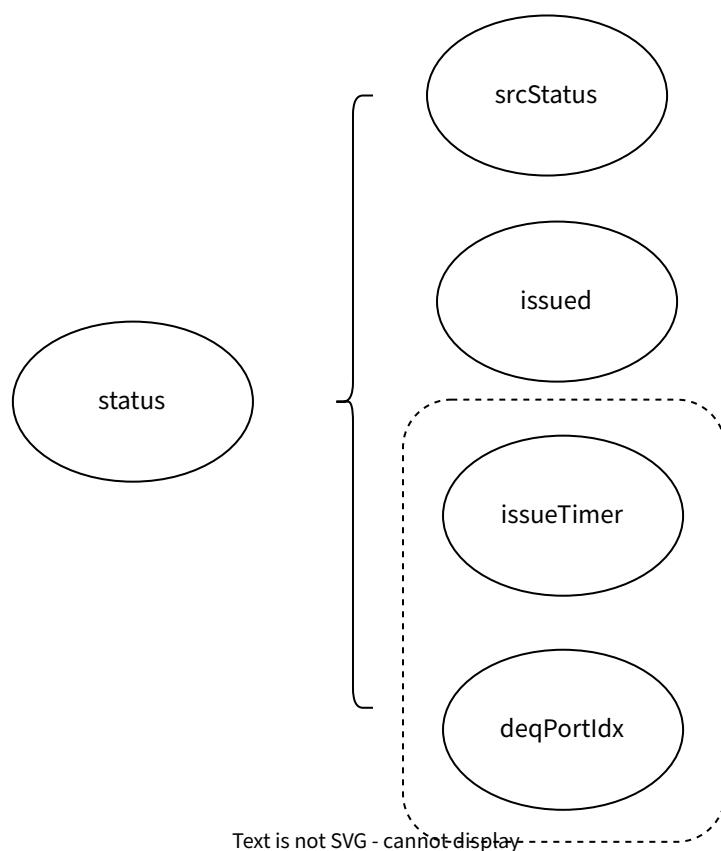


图 9.12: 示意图

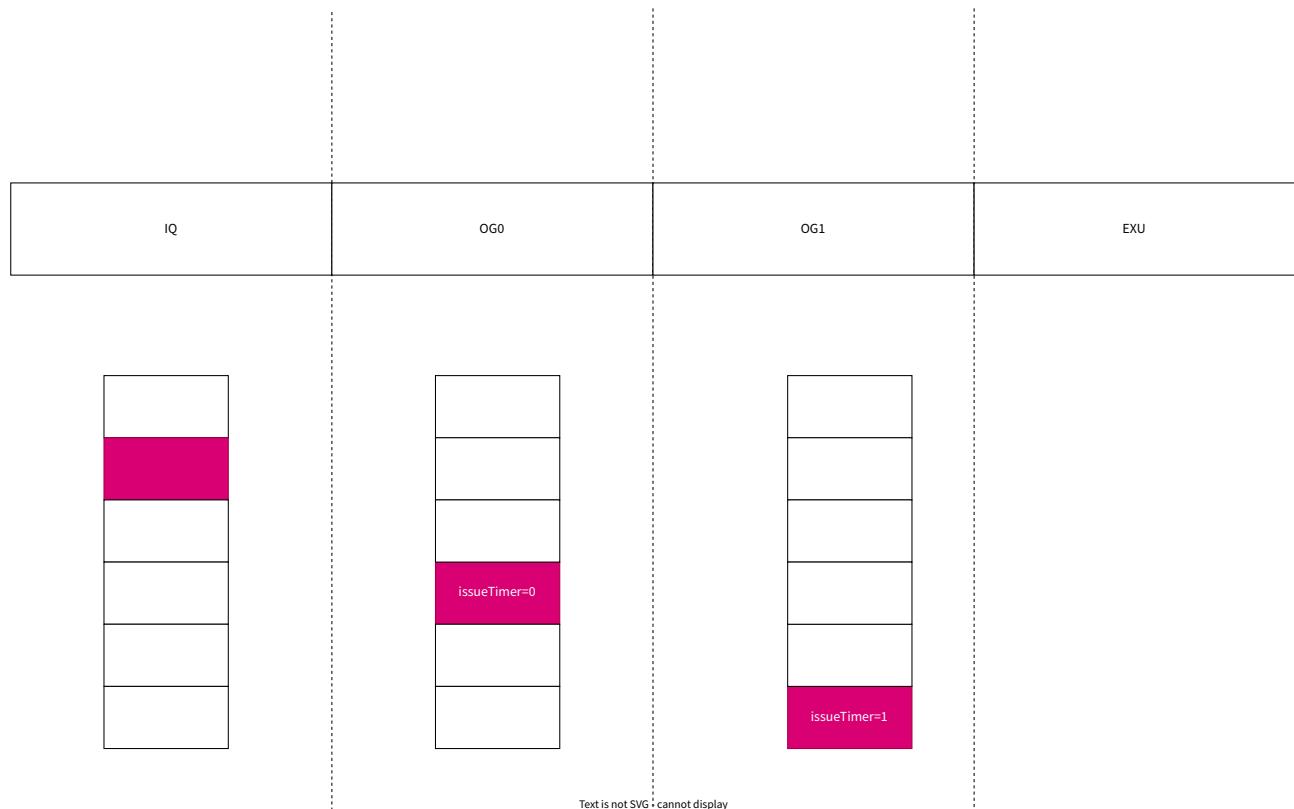


图 9.13: 示意图

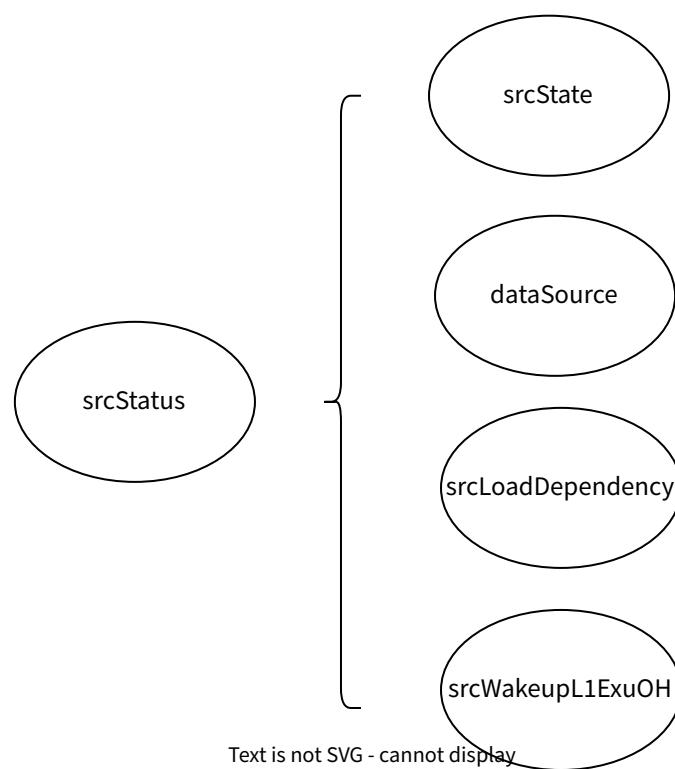


图 9.14: 示意图

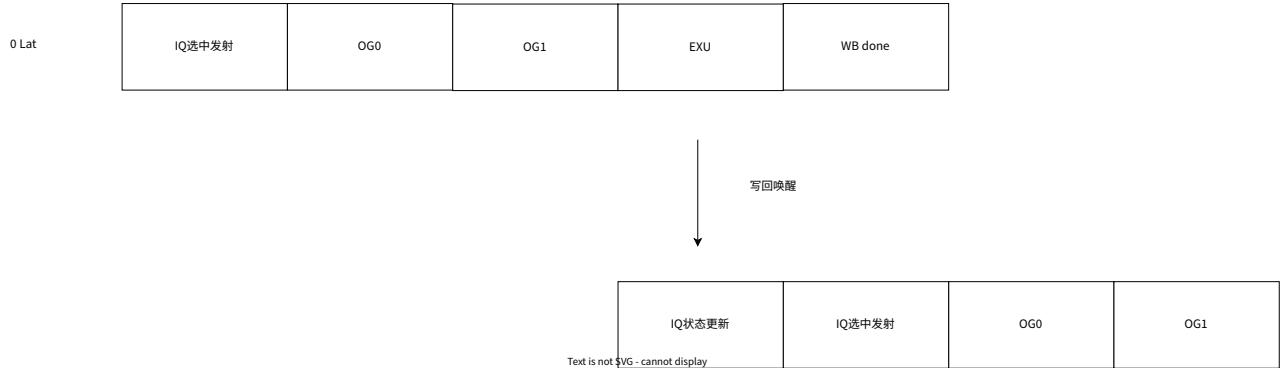


图 9.15: 示意图

写回唤醒在 uop 执行的最后一拍发出，被写回唤醒的项不支持当拍唤醒当拍发射。

dataSource 用于推测唤醒场景写回唤醒直接置为 reg 推测唤醒当拍 \rightarrow forward 每多停留一拍修改一次，最后维持 reg forward \rightarrow bypass \rightarrow reg \rightarrow reg

srcLoadDependency 3 bit, 用于记录各 uop 的 Load 依赖关系。ldCancel 产生时，刷掉唤醒链上的全部 uop。



图 9.16: 示意图

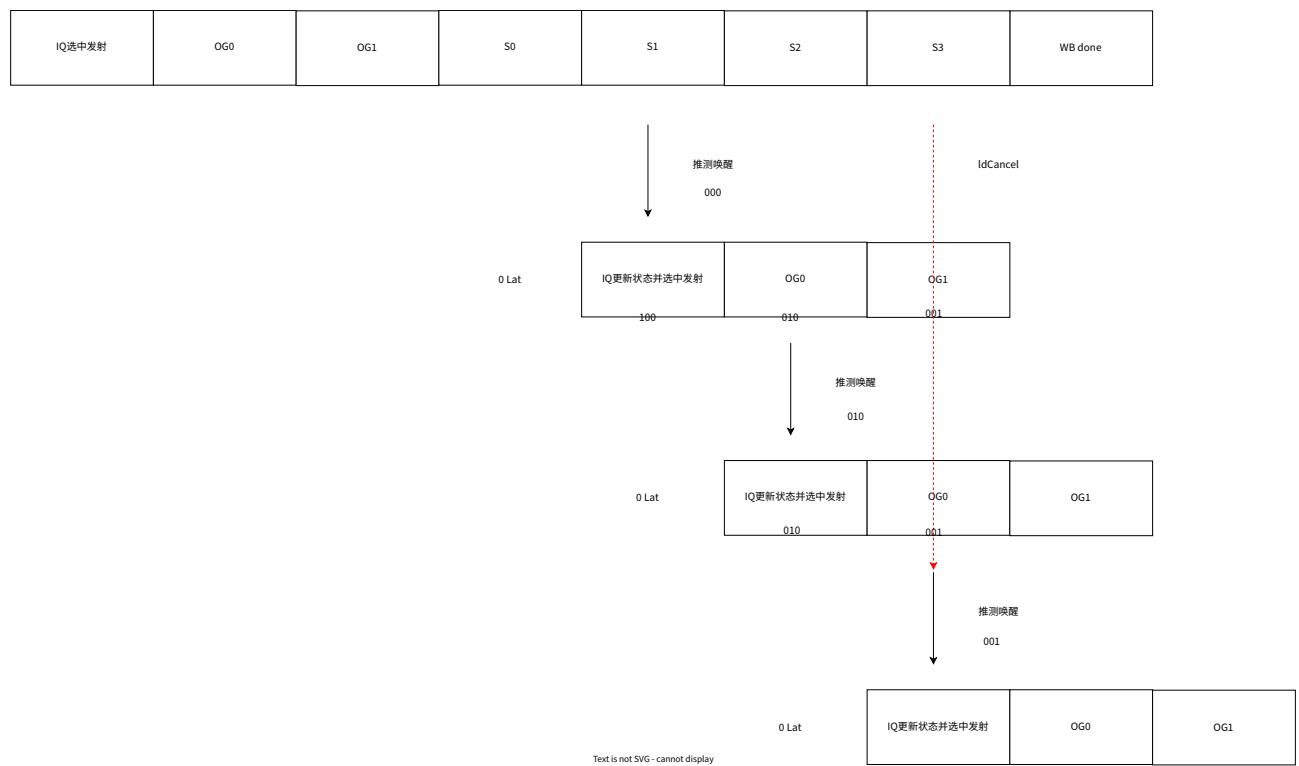


图 9.17: 示意图

10 ExuBlock 执行

10.1 ExuBlock

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

10.1.1 输入输出

`flush` 是一个带 valid 信号的 Redirect 输入

`in` 是按 issueBlock 和每个 issueBlock 包含的 exu 对应的 ExuInput 输入。即 `in(i)(j)` 表示输入来源于第 i 个 issueBlock 中的第 j 个 exu。

`out` 是按 issueBlock 和每个 issueBlock 包含的 exu 对应的 ExuOutput 输出。即 `out(i)(j)` 表示对应第 i 个 issueBlock 中的第 j 个 exu 的输出。

`csrio`、`csrin` 和 `csrToDecode` 只有当该 ExuBlock 中存在 CSR 时才存在。

类似地，`fenceio` 只有当该 ExuBlock 中存在 `fence` 时才存在。`frm` 只有当该 ExuBlock 中需要 `frm` 作为 `src` 时才存在。`vxrm` 只有当该 ExuBlock 中需要 `vxrm` 作为 `src` 时才存在。

`vtype`、`v1IsZero` 和 `v1IsV1max` 只有当该 ExuBlock 中需要写 Vconfig 时才存在。

10.1.2 功能

ExuBlock 主要负责将外部模块传入的信号按照配置需求连接到各个 exu，并将 exu 的输出整理作为 ExuBlock 的输出。

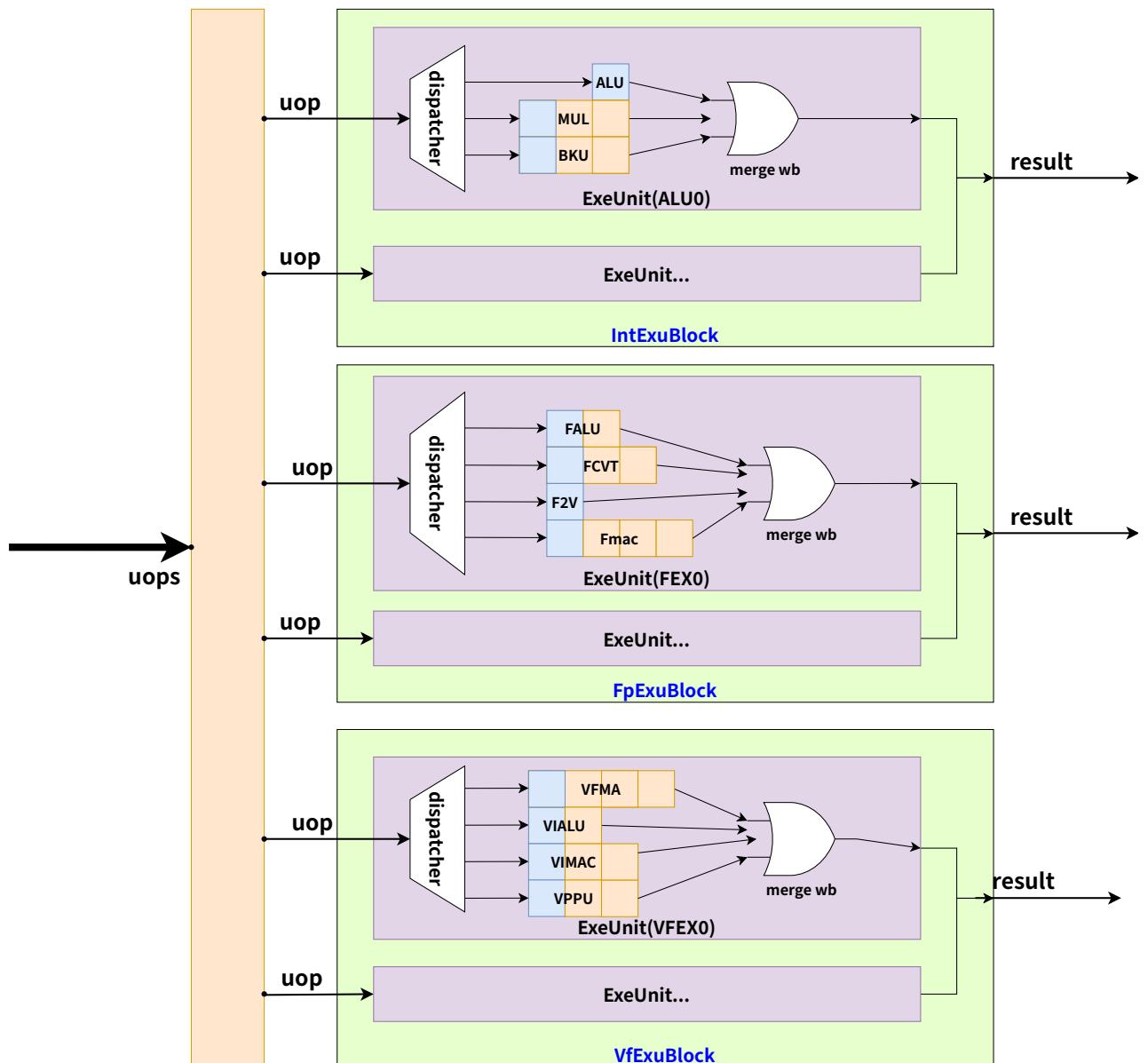


图 10.1: ExuBlock 总览

10.1.3 设计规格

在 Backend 中一共有 3 个 ExuBlock: intExuBlock, fpExuBlock 和 vfExuBlock, 分别是整数、浮点、向量的执行模块。每个 ExuBlock 中包含若干个 ExeUnit 单元。

intExuBlock 中包含了 8 个 ExeUnit, IO 包括了 flush, in, out, csrio, csrin, csrToDecode, fenceio, frm, vtype, vlIsZero 和 vlIsVlmax, 不包括 vxrm。

fpExuBlock 中包含了 5 个 ExeUnit, IO 包括了 flush, in, out 和 frm, 不包括 csrio, csrin, csrToDecode, fenceio, vxrm, vtype, vlIsZero 和 vlIsVlmax。

vfExuBlock 中包含了 5 个 ExeUnit, IO 包括了 flush, in, out, frm, vxrm, vtype, vlIsZero 和 vlIsVlmax, 不包括 csrio, csrin, csrToDecode 和 fenceio。

10.2 ExuUnit

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

10.2.1 术语说明

表 10.1: fu 术语说明

| fu | 描述 |
|-----------|-----------------|
| alu | 算术逻辑单元 |
| mul | 乘法单元 |
| bku | B 扩展位运算和密码学单元 |
| brh | 条件跳转单元 |
| jmp | 直接跳转单元 |
| i2f | 整数转浮点单元 |
| i2v | 整数移动到向量单元 |
| VSetRiWi | 读整数写整数的 vset 单元 |
| VSetRiWvf | 读整数写向量的 vset 单元 |
| csr | 控制状态寄存器单元 |
| fence | 内存同步指令单元 |
| div | 除法单元 |
| falu | 浮点算数逻辑单元 |
| fcvt | 浮点转换单元 |
| f2v | 浮点移动到向量单元 |
| fmac | 浮点融合乘加 |
| fdiv | 浮点除法单元 |
| vfma | 向量浮点融合乘加单元 |
| vialu | 向量整数算术逻辑单元 |

| fu | 描述 |
|------------|-----------------|
| vimac | 向量整数乘加单元 |
| vppu | 向量排列处理单元 |
| vfal | 向量浮点算数逻辑单元 |
| vfcvt | 向量浮点转换单元 |
| vipu | 向量整数处理单元 |
| VSetRvfWvf | 读向量写向量的 vset 单元 |
| vfdi | 向量浮点除法单元 |
| vidiv | 向量整数除法单元 |

10.2.2 输入输出

`flush` 是一个带 `valid` 信号的 Redirect 输入

`in` 是按具体 ExeUnit 参数配置生成的 `ExuInput`

`out` 是按具体 ExeUnit 参数配置生成的 `ExuOutput`

`csrio`、`csrin` 和 `csrToDecode` 只有当该 ExeUnit 中存在 CSR 时才存在。

类似地，`fenceio` 只有当该 ExeUnit 中存在 `fence` 时才存在。`frm` 只有当该 ExeUnit 中需要 `frm` 作为 src 时才存在。`vxrm` 只有当该 ExeUnit 中需要 `vxrm` 作为 src 时才存在。

`vtype`、`v1IsZero` 和 `v1IsV1max` 只有当该 ExeUnit 中需要写 `Vconfig` 时才存在。

另外，对于 ExeUnit 中存在 `JmpFu` 或者 `BrhFu` 的情况，还需要输入指令地址翻译类型 `instrAddrTransType`

10.2.3 功能

每个 ExeUnit 会根据其配置参数生成一系列对应的 FU 模块。

`busy` 用于表示当前 ExeUnit 是否处于繁忙状态。对于延迟确定的 ExeUnit，功能单元永远不会被标记为繁忙，因为延迟是固定的，所有的任务都会按顺序完成。在这种情况下，`busy` 被直接设置为 `false`，表示功能单元始终是空闲的。而对于非确定延迟的 ExeUnit，当有输入 `fire` 时将 `busy` 拉高，在输出 `fire` 时拉低。另外，如果正在输入的 `uop` 或者正在计算的 `uop` 需要被 redirect flush，则也将 `busy` 拉低。

另外，ExeUnit 中会检查混合延迟类型，即检查是否存在同一端口的功能单元具有不同的延迟类型（确定和不确定的）。如果存在这种混合情况，对于不确定延迟的功能单元，确保其优先级是最大值。这种设计逻辑确保了在处理不同类型延迟的功能单元时。写回端口的优先级得到适当的配置，避免优先级的冲突或不一致。

每个 ExeUnit 中除了拥有各个 FU 外，还有一个子模块 `in1ToN`，其是一个 Dispatcher，其作用是将输入到 ExeUnit 的一个 `ExuInput` 进一步派遣到不同的 FU，这里需要保证同一个 `ExuInput` 必须进入 1 个 FU，且不能进入多于 1 个的 FU 中。

另外还有一组寄存器 `inPipe`，是大小为 `latencyMax+1` 的 (`valid`, `input`) 对，其记录了输入，以及输入处于什么哪一周期的计算。对于需要控制流水线的 FU，可以通过 `inPipe` 获得原始的数据。

最后，还需要将不同 FU 的输出结果进行汇总，选出一个 FU 的输出结果作为 ExeUnit 的输出。

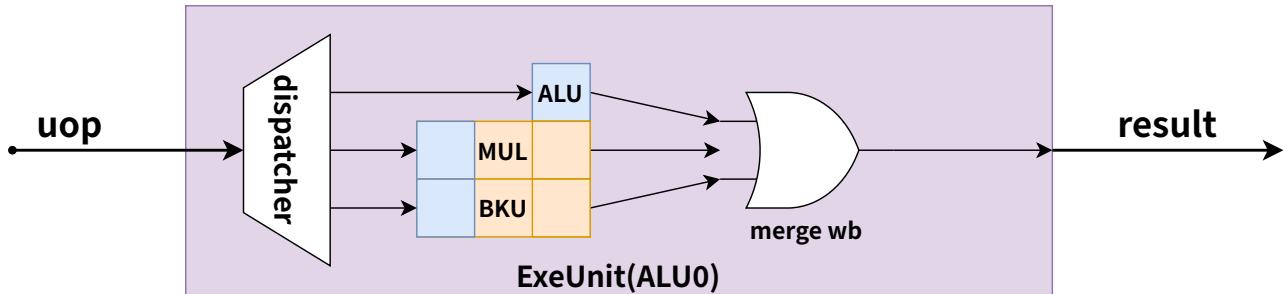


图 10.2: ExeUnit 总览

10.2.4 设计规格

在 Backend 中一共有 3 个 ExuBlock: intExuBlock, fpExuBlock 和 vfExuBlock, 分别是整数、浮点、向量的执行模块。每个 ExuBlock 中包含若干个 ExeUnit 单元。

intExuBlock 中包含了 8 个 ExeUnit, 每个 ExeUnit 对应的功能如下:

表 10.2: intExuBlock 中各个 ExeUnit 包含的 Fu

| ExeUnit | 功能 |
|---------|---|
| exus0 | alu, mul, bku |
| exus1 | brh, jmp |
| exus2 | alu, mul, bku |
| exus3 | brh, jmp |
| exus4 | alu |
| exus5 | brh, jmp, i2f, i2v, VSetRiWi, VSetRiWvf |
| exus6 | alu |
| exus7 | csr, fence, div |

fpExuBlock 中包含了 5 个 ExeUnit, 每个 ExeUnit 对应的功能如下:

表 10.3: fpExuBlock 中各个 ExeUnit 包含的 Fu

| ExeUnit | 功能 |
|---------|-----------------------|
| exus0 | falu, fcvt, f2v, fmac |
| exus1 | fdiv |
| exus2 | falu, fmac |
| exus3 | fdiv |
| exus4 | falu, fmac |

vfExuBlock 中包含了 5 个 ExeUnit, 每个 ExeUnit 对应的功能如下:

表 10.4: vfExuBlock 中各个 ExeUnit 包含的 Fu

| ExeUnit | 功能 |
|---------|--------------------------------|
| exus0 | vfma, vialu, vimac, vppu |
| exus1 | vfalu, vfcvt, vipu, VSetRvfWvf |
| exus2 | vfma, vialu |
| exus3 | vfalu |
| exus4 | vfddiv, vdiv |

10.2.5 门控

ExuUnit 还支持功能单元 FU 的时钟门控 (Clock Gating)。通过控制每个功能单元 FU 的时钟使能信号 `clk_en` 来降低功耗。只有在功能单元需要时，时钟才会被启用，并且根据功能单元的延迟设置和是否启用不确定延迟，动态计算时钟门控的使能信号，从而实现功耗优化。

简单来说，对于固定延迟且延迟周期数大于 0 的 FU，使用两个 `latReal + 1` 长度的向量 `fuVldVec` 和 `fuRdyVec`，在 FU 输入有效时，`fuVldVec(0)` 为 1，在每个周期将 1 向后移动。另外对于 `fuRdyVec(i)`，其值取决于 `fuRdyVec(i+1)` 和 `fuVldVec(i+1)`。这样当 `fuVldVec` 中有 1 时就说明当前有有效的计算。

对于不确定延迟的 FU，使用 `uncer_en_reg` 在 FU 输入 `fire` 时记录，并在 FU 输出 `fire` 时清空。

于是对于可以使用门控的 FU 来说，其 `clk_en` 拉高的条件就是：零延迟的 FU 且 FU 输入 `fire`；多周期延迟的 FU 且输入 `fire`，或当前 FU 中有有效计算；不确定延迟的 FU 且 FU 输入 `fire`，或当前 FU 中有有效的计算。通过这样的条件进行时钟门控。

11 FunctionUnit 功能单元

11.1 IntFunctionUnit

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

整数功能单元包括 jmp, brh ,i2f ,i2v ,f2v ,csr ,alu ,mul ,div, fence, bku; 每个功能单元支持的指令如下表:

11.1.1 jmp

表 11.1: jmp fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-------|----|--------|
| jmp | AUIPC | I | scalar |
| jmp | JAL | I | scalar |
| jmp | JALR | I | scalar |

11.1.2 brh

表 11.2: brh fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|------|----|--------|
| brh | BEQ | I | scalar |
| brh | BNE | I | scalar |
| brh | BGE | I | scalar |
| brh | BGEU | I | scalar |
| brh | BLT | I | scalar |
| brh | BLTU | I | scalar |

11.1.3 i2f

表 11.3: i2f fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-----------|-----|--------|
| i2f | FCVT.S.W | F | scalar |
| i2f | FCVT.S.WU | F | scalar |
| i2f | FCVT.S.L | F | scalar |
| i2f | FCVT.S.LU | F | scalar |
| i2f | FCVT.D.W | D | scalar |
| i2f | FCVT.D.WU | D | scalar |
| i2f | FCVT.D.L | D | scalar |
| i2f | FCVT.D.LU | D | scalar |
| i2f | FCVT.H.W | Zfh | scalar |
| i2f | FCVT.H.WU | Zfh | scalar |
| i2f | FCVT.H.L | Zfh | scalar |
| i2f | FCVT.H.LU | Zfh | scalar |

11.1.4 i2v

表 11.4: i2v fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|---------|-----|--------|
| i2v | FMV.D.X | D | scalar |
| i2v | FMV.W.X | F | scalar |
| i2v | FMV.H.X | Zfh | scalar |

另外作为向量指令拆分的 uop (具体拆分方式请参考 decode), 支持的 UopSplitType 为 VSET, VEC_0XV, VEC_VXV, VEC_VXW, VEC_WXW, VEC_WXV, VEC_VXM, VEC_SLIDE1UP, VEC_SLIDE1DOWN, VEC_SLIDEUP, VEC_SLIDEDOWN, VEC_RGATHER_VX, VEC_US_LDST, VEC_US_FF_LD, VEC_S_LDST, VEC_I_LDST。支持:

- 整数到向量的 move

11.1.5 f2v

表 11.5: f2v fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-------|----|-----|
| f2v | FLI.H | I | zfa |
| f2v | FLI.S | I | zfa |
| f2v | FLI.D | I | zfa |

另外作为向量指令拆分的 uop (具体拆分方式请参考 decode) , 支持的 UopSplitType 为 VEC_VVF, VEC_0XV, VEC_VFW, VEC_WFW, VEC_VFM, VEC_FSLIDE1UP, VEC_FSLIDE1DOWN。支持:

- 浮点到向量的 move

11.1.6 csr

表 11.6: csr fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|----------|-------|--------|
| csr | csrrw | I | scalar |
| csr | csrrs | I | scalar |
| csr | csrrc | I | scalar |
| csr | csrrwi | I | scalar |
| csr | csrrsi | I | scalar |
| csr | csrrci | I | scalar |
| csr | ebreak | I | scalar |
| csr | ecall | I | scalar |
| csr | sret | I | scalar |
| csr | mret | I | scalar |
| csr | mnret | smdt | scalar |
| csr | dret | debug | scalar |
| csr | wfi | | scalar |
| csr | wrs.onto | zawrs | scalar |
| csr | wrs.sto | zawrs | scalar |

11.1.7 alu

表 11.7: alu fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-------|----|--------|
| alu | LUI | I | scalar |
| alu | ADDI | I | scalar |
| alu | ANDI | I | scalar |
| alu | ORI | I | scalar |
| alu | XORI | I | scalar |
| alu | SLTI | I | scalar |
| alu | SLTIU | I | scalar |
| alu | SLL | I | scalar |
| alu | SUB | I | scalar |
| alu | SLT | I | scalar |
| alu | SLTU | I | scalar |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-----------|-----|--------|
| alu | AND | I | scalar |
| alu | OR | I | scalar |
| alu | XOR | I | scalar |
| alu | SRA | I | scalar |
| alu | SRL | I | scalar |
| alu | SLLI | I | scalar |
| alu | SRLI | I | scalar |
| alu | SRAI | I | scalar |
| alu | ADDIW | I | scalar |
| alu | SLLIW | I | scalar |
| alu | SRAIW | I | scalar |
| alu | SRLIW | I | scalar |
| alu | ADDW | I | scalar |
| alu | SUBW | I | scalar |
| alu | SLLW | I | scalar |
| alu | SRAW | I | scalar |
| alu | SRLW | I | scalar |
| alu | ADD.UW | Zba | scalar |
| alu | SH1ADD | Zba | scalar |
| alu | SH1ADD.UW | Zba | scalar |
| alu | SH2ADD | Zba | scalar |
| alu | SH2ADD.UW | Zba | scalar |
| alu | SH3ADD | Zba | scalar |
| alu | SH3ADD.UW | Zba | scalar |
| alu | SLLI.UW | Zba | scalar |
| alu | ANDN | Zbb | scalar |
| alu | ORN | Zbb | scalar |
| alu | XORN | Zbb | scalar |
| alu | MAX | Zbb | scalar |
| alu | MAXU | Zbb | scalar |
| alu | MIN | Zbb | scalar |
| alu | MINU | Zbb | scalar |
| alu | SEXT.B | Zbb | scalar |
| alu | SEXT.H | Zbb | scalar |
| alu | ROL | Zbb | scalar |
| alu | ROLW | Zbb | scalar |
| alu | ROR | Zbb | scalar |
| alu | RORI | Zbb | scalar |
| alu | RORIW | Zbb | scalar |
| alu | RORW | Zbb | scalar |
| alu | ORC.B | Zbb | scalar |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-----------|--------|--------|
| alu | REV8 | Zbb | scalar |
| alu | BCLR | Zbs | scalar |
| alu | BCLRI | Zbs | scalar |
| alu | BEXT | Zbs | scalar |
| alu | BEXTI | Zbs | scalar |
| alu | BINV | Zbs | scalar |
| alu | BINVI | Zbs | scalar |
| alu | BSET | Zbs | scalar |
| alu | BSETI | Zbs | scalar |
| alu | PACK | Zbkb | scalar |
| alu | PACKH | Zbkb | scalar |
| alu | PACKW | Zbkb | scalar |
| alu | BREV8 | Zbkb | scalar |
| alu | CZERO.EQZ | Zicond | scalar |
| alu | CZERO.NEZ | Zicond | scalar |
| alu | MOP.R | Zimop | scalar |
| alu | MOP.RR | Zimop | scalar |
| alu | TRAP | I | scalar |

11.1.8 mul

表 11.8: mul fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|--------|----|--------|
| mul | MUL | M | scalar |
| mul | MULH | M | scalar |
| mul | MULHU | M | scalar |
| mul | MULHSU | M | scalar |
| mul | MULW | M | scalar |

11.1.9 div

表 11.9: div fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|------|----|--------|
| div | DIV | M | scalar |
| div | DIVU | M | scalar |
| div | REM | M | scalar |
| div | REMU | M | scalar |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-------|----|--------|
| div | DIVW | M | scalar |
| div | DIVUW | M | scalar |
| div | REMW | M | scalar |
| div | REMUW | M | scalar |

11.1.10 fence

表 11.10: fence fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|-------|-----------------|---------|--------|
| fence | SFENCE.VMA | | scalar |
| fence | SFENCE.I | | scalar |
| fence | FENCE | | scalar |
| fence | PAUSE | | scalar |
| fence | SINVAL.VMA | Svinval | scalar |
| fence | SFENCE.W.INVAL | Svinval | scalar |
| fence | SFENCE.INVAL.IR | Svinval | scalar |
| fence | HFENCE.GVMA | | scalar |
| fence | HFENCE.VVMA | | scalar |
| fence | HINVAL.GVMA | | scalar |
| fence | HINVAL.VVMA | | scalar |

11.1.11 bku

表 11.11: bku fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|---------|------|--------|
| bku | CLZ | Zbb | scalar |
| bku | CLZW | Zbb | scalar |
| bku | CTZ | Zbb | scalar |
| bku | CTZW | Zbb | scalar |
| bku | CPOP | Zbb | scalar |
| bku | CPOPW | Zbb | scalar |
| bku | CLMUL | Zbc | scalar |
| bku | CLMULH | Zbc | scalar |
| bku | CLMULH | Zbc | scalar |
| bku | XPERM4 | Zbkx | scalar |
| bku | XPERM8 | Zbkx | scalar |
| bku | AES64DS | Zknd | scalar |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|------------|-------|--------|
| bku | AES64DSM | Zknd | scalar |
| bku | AES64IM | Zknd | scalar |
| bku | AES64KS1I | Zknd | scalar |
| bku | AES64KS2 | Zknd | scalar |
| bku | AES64ES | Zkne | scalar |
| bku | AES64ESM | Zkne | scalar |
| bku | SHA256SIG0 | Zknh | scalar |
| bku | SHA256SIG1 | Zknh | scalar |
| bku | SHA256SUM0 | Zknh | scalar |
| bku | SHA256SUM1 | Zknh | scalar |
| bku | SHA512SIG0 | Zknh | scalar |
| bku | SHA512SIG1 | Zknh | scalar |
| bku | SHA512SUM0 | Zknh | scalar |
| bku | SHA512SUM1 | Zknh | scalar |
| bku | SM4ED0 | Zksed | scalar |
| bku | SM4ED1 | Zksed | scalar |
| bku | SM4ED2 | Zksed | scalar |
| bku | SM4ED3 | Zksed | scalar |
| bku | SM4KS0 | Zksed | scalar |
| bku | SM4KS1 | Zksed | scalar |
| bku | SM4KS2 | Zksed | scalar |
| bku | SM4KS3 | Zksed | scalar |
| bku | SM3P0 | Zksh | scalar |
| bku | SM3P1 | Zksh | scalar |

11.2 FpFunctionUnit

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

浮点运算功能单元包括 falu, fmac, fcvt, fDivSqrt; 每个功能单元支持的指令如下表:

11.2.1 falu

表 11.12: falu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|---------|-----|--------|
| falu | FMINM.H | Zfa | scalar |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|------------|-----|--------|
| falu | FMINM.S | Zfa | scalar |
| falu | FMINM.D | Zfa | scalar |
| falu | FMAXM.H | Zfa | scalar |
| falu | FMAXM.S | Zfa | scalar |
| falu | FMAXM.D | Zfa | scalar |
| falu | FLEQ.H | Zfa | scalar |
| falu | FLEQ.S | Zfa | scalar |
| falu | FLEQ.D | Zfa | scalar |
| falu | FLTQ.H | Zfa | scalar |
| falu | FLTQ.S | Zfa | scalar |
| falu | FLTQ.D | Zfa | scalar |
| falu | FADD.H | Zfh | scalar |
| falu | FADD.S | F | scalar |
| falu | FADD.D | D | scalar |
| falu | FSUB.H | Zfh | scalar |
| falu | FSUB.S | F | scalar |
| falu | FSUB.D | D | scalar |
| falu | FEQ.H | Zfh | scalar |
| falu | FEQ.S | F | scalar |
| falu | FEQ.D | D | scalar |
| falu | FLT.H | Zfh | scalar |
| falu | FLT.S | F | scalar |
| falu | FLT.D | D | scalar |
| falu | FLE.H | Zfh | scalar |
| falu | FLE.S | F | scalar |
| falu | FLE.D | D | scalar |
| falu | FMIN.H | Zfh | scalar |
| falu | FMIN.S | F | scalar |
| falu | FMIN.D | D | scalar |
| falu | FCLASS.H | Zfh | scalar |
| falu | FCLASS.S | F | scalar |
| falu | FCLASS.D | D | scalar |
| falu | FSGNJ.H | Zfh | scalar |
| falu | FSGNJ.S | F | scalar |
| falu | FSGNJ.D | D | scalar |
| falu | FSGNJP.H | Zfh | scalar |
| falu | FSGNJP.S | F | scalar |
| falu | FSGNJP.D | D | scalar |
| falu | FSGNJP.N.H | Zfh | scalar |
| falu | FSGNJP.N.S | F | scalar |

| | | | |
|------|----------|---|--------|
| falu | FSGNJN.D | D | scalar |
|------|----------|---|--------|

11.2.2 fmac

表 11.13: fmac 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|----------|-----|--------|
| fmac | FMUL.H | Zfh | scalar |
| fmac | FMUL.S | F | scalar |
| fmac | FMUL.D | D | scalar |
| fmac | FMADD.H | Zfh | scalar |
| fmac | FMADD.S | F | scalar |
| fmac | FMADD.D | D | scalar |
| fmac | FMSUB.H | Zfh | scalar |
| fmac | FMSUB.S | F | scalar |
| fmac | FMSUB.D | D | scalar |
| fmac | FNMADD.H | Zfh | scalar |
| fmac | FNMADD.S | F | scalar |
| fmac | FNMADD.D | D | scalar |
| fmac | FNMSUB.H | Zfh | scalar |
| fmac | FNMSUB.S | F | scalar |
| fmac | FNMSUB.D | D | scalar |

11.2.3 fcvt

表 11.14: fcvt 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-------------|-----|--------|
| fcvt | FROUND.H | zfa | scalar |
| fcvt | FROUND.S | zfa | scalar |
| fcvt | FROUND.D | zfa | scalar |
| fcvt | FROUNDX.H | zfa | scalar |
| fcvt | FROUNDX.S | zfa | scalar |
| fcvt | FROUNDX.D | zfa | scalar |
| fcvt | FCVTMOD.W.D | zfa | scalar |
| fcvt | FCVT.W.S | F | scalar |
| fcvt | FCVT.W.U.S | F | scalar |
| fcvt | FCVT.L.S | F | scalar |
| fcvt | FCVT.L.U.S | F | scalar |
| fcvt | FCVT.D.S | D | scalar |
| fcvt | FCVT.W.D | D | scalar |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-----------|-----|--------|
| fcvt | FCVT.WU.D | D | scalar |
| fcvt | FCVT.L.D | D | scalar |
| fcvt | FCVT.LU.D | D | scalar |
| fcvt | FCVT.S.D | D | scalar |
| fcvt | FCVT.D.S | D | scalar |
| fcvt | FCVT.H.S | Zfh | scalar |
| fcvt | FCVT.S.H | Zfh | scalar |
| fcvt | FCVT.H.D | Zfh | scalar |
| fcvt | FCVT.D.H | Zfh | scalar |
| fcvt | FCVT.W.H | Zfh | scalar |
| fcvt | FCVT.WU.H | Zfh | scalar |
| fcvt | FCVT.L.H | Zfh | scalar |
| fcvt | FCVT.LU.H | Zfh | scalar |
| fcvt | FMV.X.D | D | scalar |
| fcvt | FMV.X.W | F | scalar |
| fcvt | FMV.X.H | Zfh | scalar |

11.2.4 fDivSqrt

表 11.15: fDivSqrt 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|----------|---------|-----|--------|
| fDivSqrt | FDIV.H | Zfh | scalar |
| fDivSqrt | FDIV.S | F | scalar |
| fDivSqrt | FDIV.D | D | scalar |
| fDivSqrt | FSQRT.H | Zfh | scalar |
| fDivSqrt | FSQRT.S | F | scalar |
| fDivSqrt | FSQRT.D | D | scalar |

11.3 VecFunctionUnit

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

向量功能单元包括 vsetiwi, vsetiwf, vsetfwf, vipu, vialuF, vfpu, vldu, vstu, vppu, vimac, vidiv, vfalu, vfma, vfdiv, vfcvt; 每个功能单元支持的指令如下表:

11.3.1 vsetiwi vsetiwf vsetfwf

vsetiwi, vsetiwf, vsetfwf 这三个功能单元是用以支持 vset(VSETVLI, VSETIVLI, VSETVL) 指令 uop 拆分的，具体拆分方式请参考 decode。

11.3.2 vipu

表 11.16: vipu fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|--------------|----|--------|
| vipu | vwredsumu.vs | V | vector |
| vipu | vwredsum.vs | V | vector |
| vipu | vcpop.m | V | vector |
| vipu | vfirst.m | V | vector |
| vipu | vid.v | V | vector |
| vipu | viota.m | V | vector |
| vipu | vmsbf.vv | V | vector |
| vipu | vmsif.vv | V | vector |
| vipu | vmsof.vv | V | vector |
| vipu | vmv.x.s | V | vector |
| vipu | vredand.vs | V | vector |
| vipu | vredmax.vs | V | vector |
| vipu | vredmaxu.vs | V | vector |
| vipu | vredmin.vs | V | vector |
| vipu | vredminu.vs | V | vector |
| vipu | vredor.vs | V | vector |
| vipu | vredsum.vs | V | vector |
| vipu | vredxor.vs | V | vector |

11.3.3 vialuF

表 11.17: vialuF fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|--------|----------|----|--------|
| vialuF | vadd.vv | V | vector |
| vialuF | vsub.vv | V | vector |
| vialuF | vminu.vv | V | vector |
| vialuF | vmin.vv | V | vector |
| vialuF | vmaxu.vv | V | vector |
| vialuF | vmax.vv | V | vector |
| vialuF | vand.vv | V | vector |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|--------|--------------|----|--------|
| vialuF | vor.vv | V | vector |
| vialuF | vxor.vv | V | vector |
| vialuF | vadc.vvm | V | vector |
| vialuF | vmadc.vvm | V | vector |
| vialuF | vmadc.vv | V | vector |
| vialuF | vsbc.vvm | V | vector |
| vialuF | vmsbc.vv | V | vector |
| vialuF | vmsbc.vvm | V | vector |
| vialuF | vmerge.vvm | V | vector |
| vialuF | vmv.v.v | V | vector |
| vialuF | vmseq.vv | V | vector |
| vialuF | vmsne.vv | V | vector |
| vialuF | vmsltu.vv | V | vector |
| vialuF | vmslt.vv | V | vector |
| vialuF | vmsleu.vv | V | vector |
| vialuF | vmsle.vv | V | vector |
| vialuF | vsll.vv | V | vector |
| vialuF | vsrl.vv | V | vector |
| vialuF | vsra.vv | V | vector |
| vialuF | vnsrl.wv | V | vector |
| vialuF | vnsra.wv | V | vector |
| vialuF | vsaddu.wv | V | vector |
| vialuF | vsadd.wv | V | vector |
| vialuF | vssubu.wv | V | vector |
| vialuF | vssub.wv | V | vector |
| vialuF | vssrl.wv | V | vector |
| vialuF | vssra.wv | V | vector |
| vialuF | vnclipu.wv | V | vector |
| vialuF | vnclip.wv | V | vector |
| vialuF | vwredsumu.vs | V | vector |
| vialuF | vwredsum.vs | V | vector |
| vialuF | vandn.vv | V | vector |
| vialuF | vrol.vv | V | vector |
| vialuF | vror.vv | V | vector |
| vialuF | vwsll.vv | V | vector |
| vialuF | vadd.vx | V | vector |
| vialuF | vsub.vx | V | vector |
| vialuF | vrsub.vx | V | vector |
| vialuF | vminu.vx | V | vector |
| vialuF | vmin.vx | V | vector |
| vialuF | vmaxu.vx | V | vector |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|--------|------------|----|--------|
| vialuF | vmax.vx | V | vector |
| vialuF | vand.vx | V | vector |
| vialuF | vor.vx | V | vector |
| vialuF | vxor.vx | V | vector |
| vialuF | vadc.vxm | V | vector |
| vialuF | vmadc.vxm | V | vector |
| vialuF | vmadc.vx | V | vector |
| vialuF | vsbc.vxm | V | vector |
| vialuF | vmsbc.vx | V | vector |
| vialuF | vmsbc.vxm | V | vector |
| vialuF | vmerge.vxm | V | vector |
| vialuF | vmv.v.x | V | vector |
| vialuF | vmseq.vx | V | vector |
| vialuF | vmsne.vx | V | vector |
| vialuF | vmsltu.vx | V | vector |
| vialuF | vmslt.vx | V | vector |
| vialuF | vmsleu.vx | V | vector |
| vialuF | vmsle.vx | V | vector |
| vialuF | vmsgtu.vx | V | vector |
| vialuF | vmsgt.vx | V | vector |
| vialuF | vsll.vx | V | vector |
| vialuF | vsrl.vx | V | vector |
| vialuF | vsra.vx | V | vector |
| vialuF | vnsrl.wx | V | vector |
| vialuF | vnsra.wx | V | vector |
| vialuF | vsaddu.vx | V | vector |
| vialuF | vsadd.vx | V | vector |
| vialuF | vssubu.vx | V | vector |
| vialuF | vssub.vx | V | vector |
| vialuF | vssrl.vx | V | vector |
| vialuF | vssra.vx | V | vector |
| vialuF | vnclipu.wx | V | vector |
| vialuF | vnclip.wx | V | vector |
| vialuF | vandn.vx | V | vector |
| vialuF | vrol.vx | V | vector |
| vialuF | vror.vx | V | vector |
| vialuF | vwsll.vx | V | vector |
| vialuF | vadd.vi | V | vector |
| vialuF | vrsub.vi | V | vector |
| vialuF | vand.vi | V | vector |
| vialuF | vor.vi | V | vector |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|--------|------------|----|--------|
| vialuF | vxor.vi | V | vector |
| vialuF | vadc.vim | V | vector |
| vialuF | vmadc.vim | V | vector |
| vialuF | vmadc.vi | V | vector |
| vialuF | vmerge.vim | V | vector |
| vialuF | vmv.v.i | V | vector |
| vialuF | vmseq.vi | V | vector |
| vialuF | vmsne.vi | V | vector |
| vialuF | vmsleu.vi | V | vector |
| vialuF | vmsle.vi | V | vector |
| vialuF | vmsgtu.vi | V | vector |
| vialuF | vmsgt.vi | V | vector |
| vialuF | vsll.vi | V | vector |
| vialuF | vsrl.vi | V | vector |
| vialuF | vsra.vi | V | vector |
| vialuF | vnsrl.wi | V | vector |
| vialuF | vnsra.wi | V | vector |
| vialuF | vsaddu.vi | V | vector |
| vialuF | vsadd.vi | V | vector |
| vialuF | vssrl.vi | V | vector |
| vialuF | vssra.vi | V | vector |
| vialuF | vnclipu.wi | V | vector |
| vialuF | vnclip.wi | V | vector |
| vialuF | vror.vi | V | vector |
| vialuF | vwsll.vi | V | vector |
| vialuF | vaadd.vv | V | vector |
| vialuF | vaaddu.vv | V | vector |
| vialuF | vasub.vv | V | vector |
| vialuF | vasubu.vv | V | vector |
| vialuF | vmand.mm | V | vector |
| vialuF | vmandn.mm | V | vector |
| vialuF | vmnand.mm | V | vector |
| vialuF | vmnor.mm | V | vector |
| vialuF | vmor.mm | V | vector |
| vialuF | vmorn.mm | V | vector |
| vialuF | vmxnor.mm | V | vector |
| vialuF | vmxor.mm | V | vector |
| vialuF | vsext.vf2 | V | vector |
| vialuF | vsext.vf4 | V | vector |
| vialuF | vsext.vf8 | V | vector |
| vialuF | vzext.vf2 | V | vector |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|--------|-----------|----|--------|
| vialuF | vzext.vf4 | V | vector |
| vialuF | vzext.vf8 | V | vector |
| vialuF | vwadd.vv | V | vector |
| vialuF | vwadd.wv | V | vector |
| vialuF | vwaddu.vv | V | vector |
| vialuF | vwaddu.wv | V | vector |
| vialuF | vbsub.vv | V | vector |
| vialuF | vbsub.wv | V | vector |
| vialuF | vbsubu.vv | V | vector |
| vialuF | vbsubu.wv | V | vector |
| vialuF | vbrev.v | V | vector |
| vialuF | vbrev8.v | V | vector |
| vialuF | vrev8.v | V | vector |
| vialuF | vclz.v | V | vector |
| vialuF | vctz.v | V | vector |
| vialuF | vcpop.v | V | vector |
| vialuF | vaadd.vx | V | vector |
| vialuF | vaaddu.vx | V | vector |
| vialuF | vasub.vx | V | vector |
| vialuF | vasubu.vx | V | vector |
| vialuF | vmv.s.x | V | vector |
| vialuF | vwadd.vx | V | vector |
| vialuF | vwadd.wx | V | vector |
| vialuF | vwaddu.vx | V | vector |
| vialuF | vwaddu.wx | V | vector |
| vialuF | vbsub.vx | V | vector |
| vialuF | vbsub.wx | V | vector |
| vialuF | vbsubu.vx | V | vector |
| vialuF | vbsubu.wx | V | vector |

11.3.4 vldu

11.3.5 vstu

11.3.6 vppu

表 11.18: vppu fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-----------------|----|--------|
| vppu | vrgather.vv | V | vector |
| vppu | vrgatherei16.vx | V | vector |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-----------------|----|--------|
| vppu | vrgather.vx | V | vector |
| vppu | vslideup.vx | V | vector |
| vppu | vslidedown.vx | V | vector |
| vppu | vrgather.vi | V | vector |
| vppu | vslideup.vi | V | vector |
| vppu | vslidedown.vi | V | vector |
| vppu | vmv1r.v | V | vector |
| vppu | vmv2r.v | V | vector |
| vppu | vmv4r.v | V | vector |
| vppu | vmv8r.v | V | vector |
| vppu | vcompress.vm | V | vector |
| vppu | vslide1up.vx | V | vector |
| vppu | vslide1down.vx | V | vector |
| vppu | vfslide1up.vf | V | vector |
| vppu | vfslide1down.vf | V | vector |

11.3.7 vimac

表 11.19: vimac fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|-------|-------------|----|--------|
| vimac | vsmul.vv | V | vector |
| vimac | vsmul.vx | V | vector |
| vimac | vmacc.vv | V | vector |
| vimac | vmadd.vv | V | vector |
| vimac | vmul.vv | V | vector |
| vimac | vmulh.vv | V | vector |
| vimac | vmulhsu.vv | V | vector |
| vimac | vmulhu.vv | V | vector |
| vimac | vnmsac.vv | V | vector |
| vimac | vnmsub.vv | V | vector |
| vimac | vwmacc.vv | V | vector |
| vimac | vwmaccsu.vv | V | vector |
| vimac | vwmaccu.vv | V | vector |
| vimac | vwmul.vv | V | vector |
| vimac | vwmulsu.vv | V | vector |
| vimac | vwmulu.vv | V | vector |
| vimac | vmacc.vx | V | vector |
| vimac | vmadd.vx | V | vector |
| vimac | vmul.vx | V | vector |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|-------|-------------|----|--------|
| vimac | vmulh.vx | V | vector |
| vimac | vmulhsu.vx | V | vector |
| vimac | vmulhu.vx | V | vector |
| vimac | vnmsac.vx | V | vector |
| vimac | vnmsub.vx | V | vector |
| vimac | vwmacc.vx | V | vector |
| vimac | vwmaccsu.vx | V | vector |
| vimac | vwmaccu.vx | V | vector |
| vimac | vwmaccus.vx | V | vector |
| vimac | vwmul.vx | V | vector |
| vimac | vwmulsu.vx | V | vector |
| vimac | vwmulu.wx | V | vector |

11.3.8 vidiv

表 11.20: vidiv fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|-------|----------|----|--------|
| vidiv | vdiv.vv | V | vector |
| vidiv | vdivu.vv | V | vector |
| vidiv | vrem.vv | V | vector |
| vidiv | vremu.vv | V | vector |
| vidiv | vdiv.vx | V | vector |
| vidiv | vdivu.vx | V | vector |
| vidiv | vrem.vx | V | vector |
| vidiv | vremu.vx | V | vector |

11.3.9 vfalu

表 11.21: vfalu fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|-------|-----------|----|--------|
| vfalu | vfadd.vv | V | vector |
| vfalu | vbsub.vv | V | vector |
| vfalu | vfwadd.vv | V | vector |
| vfalu | vfwsub.vv | V | vector |
| vfalu | vfwadd.wv | V | vector |
| vfalu | vfwsub.wv | V | vector |
| vfalu | vfmin.vv | V | vector |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|-------|---------------|----|--------|
| vfalu | vfmax.vv | V | vector |
| vfalu | vfsgnj.vv | V | vector |
| vfalu | vfsgnjn.vv | V | vector |
| vfalu | vfsgnjx.vv | V | vector |
| vfalu | vmfeq.vv | V | vector |
| vfalu | vmfne.vv | V | vector |
| vfalu | vmflt.vv | V | vector |
| vfalu | vmfle.vv | V | vector |
| vfalu | vfclass.v | V | vector |
| vfalu | vfredosum.vs | V | vector |
| vfalu | vfredusum.vs | V | vector |
| vfalu | vfredmax.vs | V | vector |
| vfalu | vfredmin.vs | V | vector |
| vfalu | vfwredosum.vs | V | vector |
| vfalu | vfwredusum.vs | V | vector |
| vfalu | vfadd.vf | V | vector |
| vfalu | vbsub.vf | V | vector |
| vfalu | vfrsub.vf | V | vector |
| vfalu | vfwadd.vf | V | vector |
| vfalu | vfwsub.vf | V | vector |
| vfalu | vfwadd.wf | V | vector |
| vfalu | vfwsub.wf | V | vector |
| vfalu | vfmin.vf | V | vector |
| vfalu | vfmax.vf | V | vector |
| vfalu | vfsgnj.vf | V | vector |
| vfalu | vfsgnjn.vf | V | vector |
| vfalu | vfsgnjx.vf | V | vector |
| vfalu | vmfeq.vf | V | vector |
| vfalu | vmfne.vf | V | vector |
| vfalu | vmflt.vf | V | vector |
| vfalu | vmfle.vf | V | vector |
| vfalu | vmfgt.vf | V | vector |
| vfalu | vmfge.vf | V | vector |
| vfalu | vfmerge.vfm | V | vector |
| vfalu | vfmv.v.f | V | vector |
| vfalu | vfmv.f.s | V | vector |
| vfalu | vfmv.s.f | V | vector |

11.3.10 vfma

表 11.22: vfma fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|------------|----|--------|
| vfma | vfmul.vv | V | vector |
| vfma | vfwmul.vv | V | vector |
| vfma | vfmacc.vv | V | vector |
| vfma | vfnmacc.vv | V | vector |
| vfma | vfmsac.vv | V | vector |
| vfma | vfnmsac.vv | V | vector |
| vfma | vfmadd.vv | V | vector |
| vfma | vfnmadd.vv | V | vector |
| vfma | vfmsub.vv | V | vector |
| vfma | vfnmsub.vv | V | vector |
| vfma | vfwmacc.vv | V | vector |
| vfma | vfnmacc.vv | V | vector |
| vfma | vfwmsac.vv | V | vector |
| vfma | vfnmsac.vv | V | vector |
| vfma | vfmul.vf | V | vector |
| vfma | vfwmul.vf | V | vector |
| vfma | vfmacc.vf | V | vector |
| vfma | vfnmacc.vf | V | vector |
| vfma | vfmsac.vf | V | vector |
| vfma | vfnmsac.vf | V | vector |
| vfma | vfmadd.vf | V | vector |
| vfma | vfnmadd.vf | V | vector |
| vfma | vfmsub.vf | V | vector |
| vfma | vfnmsub.vf | V | vector |
| vfma | vfwmacc.vf | V | vector |
| vfma | vfnmacc.vf | V | vector |
| vfma | vfwmsac.vf | V | vector |
| vfma | vfnmsac.vf | V | vector |

11.3.11 vfdiV

表 11.23: vfdiV fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|-------|-----------|----|--------|
| vfdiV | vfdiV.vv | V | vector |
| vfdiV | vfsqrt.v | V | vector |
| vfdiV | vfdiV.vf | V | vector |
| vfdiV | vfrdiV.vf | V | vector |

11.3.12 vfcvt

表 11.24: vfcvt fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|-------|-------------------|----|--------|
| vfcvt | vfrsqrt7.v | V | vector |
| vfcvt | vfrec7.v | V | vector |
| vfcvt | vfcvt.xu.f.v | V | vector |
| vfcvt | vfcvt.x.f.v | V | vector |
| vfcvt | vfcvt.rtz.xu.f.v | V | vector |
| vfcvt | vfcvt.rtz.x.f.v | V | vector |
| vfcvt | vfcvt.f.xu.v | V | vector |
| vfcvt | vfwcvt.xu.f.v | V | vector |
| vfcvt | vfwcvt.x.f.v | V | vector |
| vfcvt | vfwcvt.rtz.xu.f.v | V | vector |
| vfcvt | vfwcvt.rtz.x.f.v | V | vector |
| vfcvt | vfwcvt.f.xu.v | V | vector |
| vfcvt | vfwcvt.f.x.v | V | vector |
| vfcvt | vfwcvt.f.f.v | V | vector |
| vfcvt | vfncvt.xu.f.w | V | vector |
| vfcvt | vfncvt.x.f.w | V | vector |
| vfcvt | vfncvt.rtz.xu.f.w | V | vector |
| vfcvt | vfncvt.rtz.x.f.w | V | vector |
| vfcvt | vfncvt.f.xu.w | V | vector |
| vfcvt | vfncvt.f.x.w | V | vector |
| vfcvt | vfncvt.f.f.w | V | vector |
| vfcvt | vfncvt.rod.f.f.w | V | vector |

11.4 FpFunctionUnit

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

标量访存功能单元包括 ldu, stu, mou; 每个功能单元支持的指令如下表:

11.4.1 ldu

表 11.25: ldu fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|---------|----|--------|
| ldu | FLD | D | scalar |
| ldu | FLW | F | scalar |
| ldu | LB | I | scalar |
| ldu | LBU | I | scalar |
| ldu | LD | D | scalar |
| ldu | LH | I | scalar |
| ldu | LHU | I | scalar |
| ldu | LW | I | scalar |
| ldu | LWU | I | scalar |
| ldu | HLV.B | | scalar |
| ldu | HLV.BU | | scalar |
| ldu | HLV.D | | scalar |
| ldu | HLV.H | | scalar |
| ldu | HLV.HU | | scalar |
| ldu | HLV.W | | scalar |
| ldu | HLV.WU | | scalar |
| ldu | HLVX.HU | | scalar |
| ldu | HLVX.WU | | scalar |

11.4.2 stu

表 11.26: sdu fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-----------|-----|--------|
| stu | SB | I | scalar |
| stu | SD | I | scalar |
| stu | SH | I | scalar |
| stu | SW | I | scalar |
| stu | FSD | D | scalar |
| stu | FSW | F | scalar |
| stu | FSH | Zfh | scalar |
| stu | CBO_CLEAN | CBO | scalar |
| stu | CBO_FLUSH | CBO | scalar |
| stu | CBO_INVAL | CBO | scalar |
| stu | CBO_ZERO | CBO | scalar |
| stu | HSV.B | | scalar |
| stu | HSV.D | | scalar |
| stu | HSV.H | | scalar |

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-------|----|--------|
| stu | HSV.W | | scalar |

11.4.3 mou

表 11.27: mou fu 支持的指令

| 功能单元 | 支持指令 | 扩展 | 描述 |
|------|-----------|----|--------|
| mou | AMOADD_D | A | scalar |
| mou | AMOADD_W | A | scalar |
| mou | AMOAND_D | A | scalar |
| mou | AMOAND_W | A | scalar |
| mou | AMOMAX_D | A | scalar |
| mou | AMOMAX_W | A | scalar |
| mou | AMOMAXU_D | A | scalar |
| mou | AMOMAXU_W | A | scalar |
| mou | AMOMIN_D | A | scalar |
| mou | AMOMIN_W | A | scalar |
| mou | AMOMINU_D | A | scalar |
| mou | AMOMINU_W | A | scalar |
| mou | AMOOR_D | A | scalar |
| mou | AMOOR_W | A | scalar |
| mou | AMOSWAP_D | A | scalar |
| mou | AMOSWAP_W | A | scalar |
| mou | AMOXOR_D | A | scalar |
| mou | AMOXOR_W | A | scalar |
| mou | LR_D | A | scalar |
| mou | LR_W | A | scalar |
| mou | SC_D | A | scalar |
| mou | SC_W | A | scalar |

12 VFPUs

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

12.1 术语说明

| 缩写 | 全称 | 描述 |
|-------|-------------------------|----------|
| VFPUs | Vector Float Point Unit | 向量浮点功能单元 |
| IQ | Issue Queue | 发射队列 |

12.2 设计规格

1. 支持向量浮点 Mul 计算
2. 支持向量浮点 FMA 计算
3. 支持向量浮点 Div 计算
4. 支持向量浮点 Sqrt 计算
5. 支持 fp32,fp64,fp16 计算
6. 支持 RV-V1.0 版本向量浮点指令的计算

12.3 功能

VFPUs 模块接收来自 Issue Queue 发射的 uop 信息，根据 fuType 和 fuOpType 等信息，完成向量浮点指令的计算，主要包括：VFAlu、VFMA、VFDivSqrt、VFCvt 四个模块。

VFAlu 主要负责 fadd 相关的指令和一些其他简单指令，如比较指令、符号注入指令，特别的是 reduction sum 指令也通过拆 uop 的方式在此模块进行计算。

VFMA 主要负责乘法和乘加相关的指令。

VFDivSqrt 主要负责除法和开方相关的指令。

VFCvt 主要负责格式转换和倒数估计相关的指令。

12.4 算法设计

向量浮点运算单元的难点在于支持多组单一精度格式（操作数和结果的浮点数格式相同）的计算以及支持混合精度（操作数和结果的浮点数格式不同）的计算，以常见的半精度($f16$)、单精度($f32$)、双精度($f64$)为例，对比标量浮点运算单元和向量浮点运算单元的区别。

以典型的浮点加法为例，对于标量浮点运算单元，只需支持三种单一精度格式的计算，该运算单元输入的操作数和输出的结果都应该是64位的，那么它要支持三种格式的计算：

- (1) 1个 $f64 = f64 + f64$;
- (2) 1个 $f32 = f32 + f32$;
- (3) 1个 $f16 = f16 + f16$ 。

看似需要三个模块完成这三种格式的计算，但是因为浮点数都是有符号位、阶码、尾数三部分组成，并且高精度浮点数的阶码位宽和尾数位宽都要比低精度的更大，这就使得高精度浮点数的硬件设计可以完全满足低精度浮点数的硬件需求，稍加处理之后就可以通过在硬件中添加 Mux （多路选择器）的方式兼容多种单一精度格式的计算，并且面积只会略微增加。

而向量浮点运算单元要支持向量操作，向量操作的一个特点是对数据带宽利用率高，比如标量运算单元虽然接口是64位的，但在计算 $f32/f16$ 时，其有效数据只有32/16位，带宽利用率骤减到50%/25%；向量运算单元接口也是64位的，但是在计算单一精度格式 $f32/f16$ 时，它可以同时进行2/4组操作，带宽利用率还是可以达到100%，支持的单一精度格式计算如下：

- (1) 1个 $f64 = f64 + f64$;
- (2) 2个 $f32 = f32 + f32$;
- (3) 4个 $f16 = f16 + f16$ 。

需要同时进行多组相同格式的浮点数相加使得硬件设计比标量更困难，但是也能将高精度格式的硬件复用来计算低精度格式。另外，向量浮点运算单元还要支持的一个特性是混合精度计算，RISC-V向量指令集拓展定义了一系列需要混合精度计算的 *widening* 指令，要求浮点加法运算单元还要支持以下四种计算格式：

- (1) 1个 $f64 = f64 + f32$;
- (2) 1个 $f64 = f32 + f32$;
- (3) 2个 $f32 = f32 + f16$;
- (4) 2个 $f32 = f16 + f16$ 。

混合精度计算的设计难度比多组单一精度格式要大得多。一方面，不同的数据格式操作数需要进行格式转换，转换成和结果相同的格式再计算，逻辑复杂度提高了；另一方面，格式转换对电路时序的压力非常大，尤其是将低精度的非规格化数转换成高精度浮点数时，因此本文专门设计了一种快速数据格式转换算法来解决时序问题。

综上所述，向量浮点运算器的设计难点在于多组单一精度格式的实现和混合精度格式的实现。本节将逐一介绍向量浮点加法算法、浮点顺序累加算法、向量浮点融合乘加算法、向量浮点除法算法来解决向量浮点运算器的设计难点，实现频率可达 $3GHz$ 的高性能向量浮点运算器。

12.4.1 向量浮点加法

浮点加法是科学计算中最常用的算术运算之一。尽管概念简单，但是传统的单通路浮点加法算法需要两到三个有符号加法步骤，这是一个相对耗时的操作。双通路浮点加法算法在最坏情况下关键路径上只有一个有符号加法操作，因此与单通路算法相比具有很大的速度优势，本文在双通路浮点加法算法的基础上，设计了一种更为快速的改进的双通路浮点加法算法。本节先介绍单一精度格式的单通路浮点加法算法、双通路浮点加法算法、改进的双通路浮点加法算法，最后介绍向量浮点加法算法。

浮点加法公式表示为: $fp_result = fp_a + fp_b$ 。当 fp_a 和 fp_b 符号相同时, 有效数字对齐后做加法, 这种情况称作等效加法; 当 fp_a 和 fp_b 符号不同时, 有效数字对齐后做减法, 这种情况称作等效减法。对于非规格化数阶码等于 0, 和规格化数阶码等于 1 对应的规格化的指数是一样的, 所以计算指数差时, 阶码为 0 时应当作 1 处理 (称为规格化的阶码), 规格化的阶码之差的绝对值为规格化的指数之差。

12.4.1.1 单通路浮点加法算法

传统的单通路浮点加法运算如图所示, 由以下步骤组成:

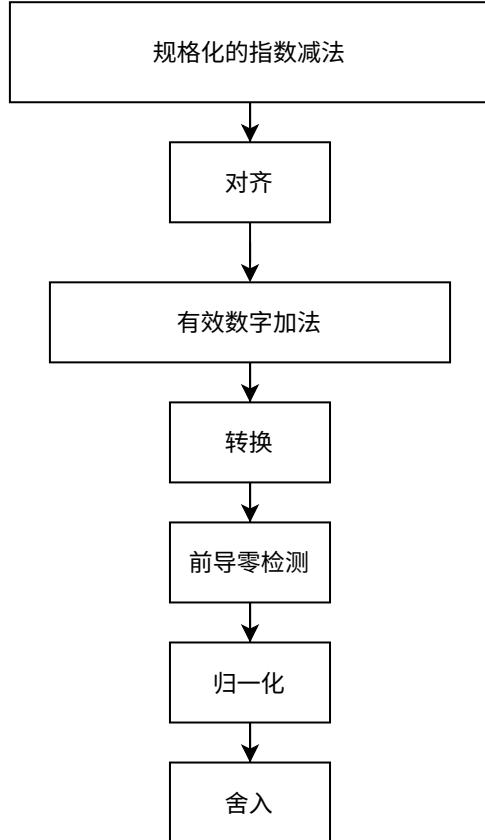


图 12.1: 单通路浮点加法算法

- (1) 规格化的指数减法 (Exponent subtraction, ES): 求规格化的指数之差 $d = |Ea - Eb|$, Ea 、 Eb 都是规格化的阶码。
- (2) 对齐 (Alignment, $Align$): 将较小的操作数的有效数字右移 d 位。较大的指数记作 Ef 。
- (3) 有效数字加法 (Signicand addition, SA): 根据有效操作 Eo 执行加法或减法, Eo 是浮点加法单元中加法器实际执行的算术运算, 根据两浮点操作数的符号位确定。
- (4) 转换 (Conversion, $Conv$): 如果有效数字加法结果为负, 则将结果转换为符号-幅度表示。转换是通过一个加法步骤完成的, 转换结果表示为 Sf 。
- (5) 前导零检测 (Leading zero detection, LZD): 计算所需的左移或右移量, 并将其表示为 En , 右移为正, 否则为负。
- (6) 归一化 (Normalization, $Norm$): 通过移 En 位将有效数字归一化, 并将 En 加到 Ef 上。
- (7) 舍入 (Rounding, $Round$): 根据 IEEE-754 标准舍入, 必要时在 Sf 的 LSB 上加 1, 此步骤可能导致溢出, 需要将尾数结果右移一位, 同时指数 Ef 加 1。

12.4.1.2 双通路浮点加法算法

上述单通路浮点算法很慢，因为加法运算中的步骤基本上都是串行执行的，可以通过以下方式改进该算法：

(1) 在单通路浮点加法算法中，仅当结果为负时才需要 *Conv* 步骤，并且可以通过交换两操作数的有效数字来避免 *Conv* 步骤。通过检查 *ES* 步骤结果的正负号，并可以相应的交换 (*Swap*) 有效数字，总是计算较大的有效数字减去较小的有效数字。在指数相等的情况下，结果仍可能为负，需要进行转换，但在这种情况下不需要舍入。因此，交换步骤使舍入和转换相互排斥，允许将它们并行起来。注意交换还有一个优点是只需要一个移位器。

(2) 前导零检测步骤可以与有效数字加法步骤并行执行，将其从关键路径中移除。在计算减法的情况下，当需要大量左移时，这种优化非常重要。

(3) 到目前为止，已经能够将关键路径步骤减少为：规格化的指数减法，交换，对齐，有效数字加法 || 前导零检测，转换 || 舍入，归一化 (|| 表示可以并行执行的步骤)。对齐和归一化步骤是互斥的，可以进一步优化，只有当 $d \leq 1$ 时或者等效减法时，归一化需要大量的左移。相反，只有当 $d > 1$ 时，对齐步骤需要大量的右移。通过区分这两种情况，只有一个大量的移位关键路径，对齐或归一化。

单通路浮点加法算法和双通路浮点加法算法中的步骤如表所示。在双通路浮点加法算法中， $d \leq 1$ 路径中的预处理步骤 (*Pred*) 根据 d 的值决定是否需要右移一位来对齐有效数字。双通路浮点加法算法通过并行执行更多的步骤来提高速度，因此需要更多的硬件来实现。

表 12.2: 两种浮点加法算法步骤

| 单通路浮点加法 | 双通路浮点加法算法 | |
|----------|------------------|---------------|
| | $d \leq 1$ 且等效减法 | $d > 1$ 或等效加法 |
| 规格化的指数加法 | 预处理 + 交换 | 规格化的指数减法 + 交换 |
| 对齐 | - | 对齐 |
| 有效数字加法 | 有效数字加法或前导零检测 | 有效数字加法 |
| 转换 | 转换或舍入 | 舍入 |
| 前导零检测 | - | - |
| 归一化 | 归一化 | - |
| 舍入 | 选择路径 | 选择路径 |

在双通路浮点加法算法中，*SA* 步骤在等效减法的情况下，其中一个有效数字是 2 的补码，该求补码步骤和舍入步骤是互斥的，因此可以并行，优化后的双通路浮点加法算法见表。

表 12.3: 优化后的双通路浮点加法算法

| $d \leq 1$ 且等效减法 | $d > 1$ 或等效加法 |
|-------------------------|---------------|
| 预处理 + 交换 | 规格化的指令减法 + 交换 |
| 有效数字加法转换 舍入 前导零检测 | 对齐 |
| 归一化 | 有效数字加法 舍入 |
| 选择路径 | 选择路径 |

在 IEEE 舍入到最近值 (*RTN*) 模式中，计算 $A + B$ 和 $A + B + 1$ 足以解决所有归一化可能性（向正负无穷方向舍入时还需额外计算 $A + B + 2$ ）。通过使用 *Cin* 从多组有效数字加法器结果中选择最终尾数舍入后的结果，可以同时完成求补码和舍入，从而节省一个加法步骤。由于在浮点加法中，结果的归一化可能需要右

移一位、不移位或左移，左移的位数可能与有效数字的长度一样多，因此 Cin 需要考虑所有这些归一化的可能性，使得最终选择的结果是舍入后的结果。

12.4.1.3 改进的双通路浮点加法算法

本节详细介绍本文改进的双通路浮点加法算法，等效加法或 $d \geq 1$ 的等效减法的通路称为 *far* 路径， $d \leq 1$ 且等效减法的通路称为 *close* 路径。含有无穷或 *NaN* 操作数的情况单独判断，不属于 *far* 路径或 *close* 路径。

12.4.1.3.1 *far* 路径

far 路径算法如图所示，其中主要步骤如下：

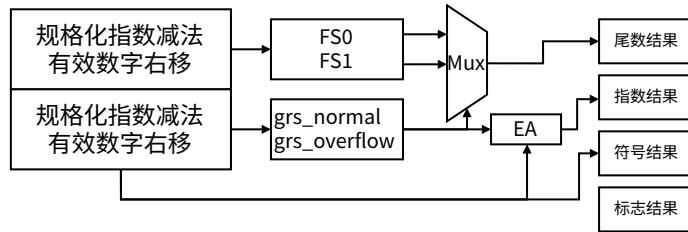


图 12.2: *far* 路径算法示意图

第一步，*far* 路径下指数差 d 大于 1，要对较小的有效数字右移 d 位来对齐较大的有效数字，首先计算规格化的指数差，为了加快计算速度，使用两个加法器来计算规格化的指数差，同时进行 Efp_a 与 Efp_b 大小比较，根据指数大小的比较结果选择出正确的规格化的指数差。

第二步，根据第一步中指数大小比较关系，可以与第一步求规格化的指数差并行选择出指数较大的操作数的有效数字和指数较小的操作数的有效数字，同时选出较大的指数 EA ，当等效减法时， EA 自减 1（此时 EA 不可能等于 0，因为如果等于 0 就是 *close* 路径了），这样做的目的是调整有效数字做完减法后的值域，和等效加法值域统一起来方便后面选择出最终结果。调整后有效数字加或减的结果值域在 [1-4) 之间，分为两种情况：[1-2) 之间、[2-4) 之间。

第三步，对较小的有效数字右移，此处右移分两种情况：在等效减法时，对较小的有效数字先取反再算数右移，比先右移再取反节省了一些时间；等效加法时之间逻辑右移。为了节省右移器的级数，当规格化的指数差的高位全 0 时，右移时用规格化的指数差的低位（具体位数取决于有效数字宽度）进行右移，当规格化的指数差的高位不是全 0 时，右移结果为 0。这里使用第一步中的求两个规格化的指数差的加法器结果，并且先使用最低位进行右移（因为加法结果最低位最先得到），具体如下：假设 fp_a 指数大，则只需对 fp_b 的有效数字进行右移，右移的值就是 fp_a 的规格化的指数减 fp_b 的规格化的指数；假设 fp_b 指数大，则只需对 fp_a 的有效数字进行右移，右移的值就是 fp_b 的规格化的指数减 fp_a 的规格化的指数。然后根据指数大小关系和规格化的指数差的值选出最终右移后的有效数字，并且计算出右移后的 *grs* (*guard*, *round*, *sticky*) 位。此处为了对第二步的两种情况正确舍入，需要计算有效数字加减后结果在 [1-2) 之间，[2-4) 之间的两组 *grs*。

第四步，进行有效数字加法，因为等效减法时较小的有效数字在右移前已经取反了，此处记较大的有效数字为 A ，较小的有效数字右移后为 B ，两个有效数字加法器分别计算 $A + B$ 和 $A + B + 2$ ，最终舍入结果从这两个加法器结果中选择出来。

第五步，产生最终结果，根据有效数字 $A + B$ 的结果在 [1-2) 之间（情况一）还是 [2-4) 之间（情况二），根据之前右移中计算的两套 *grs* 和舍入模式，分别确定情况一选择两个有效数字加法器的条件，情况二选择两个有效数字加法器的条件，最后用四选一的独热码选择出尾数结果。指数结果则是 EA （情况一且尾数舍入后 < 1 ）或 $EA + 1$ （情况二或情况一舍入后 = 2），注意舍入后指数是否 *overflow*，最终结果由 *overflow* 选出 *overflow* 结果和正常计算结果。异常标志位 *far* 路径下只会产生上溢和不精确。

12.4.1.3.2 close 路径

close 路径下一定是等效减法并且 $d \leq 1$ 的情况，具体细分为 $d = 0$ 或 $d = 1$ ，算法如图所示，具体步骤如下：

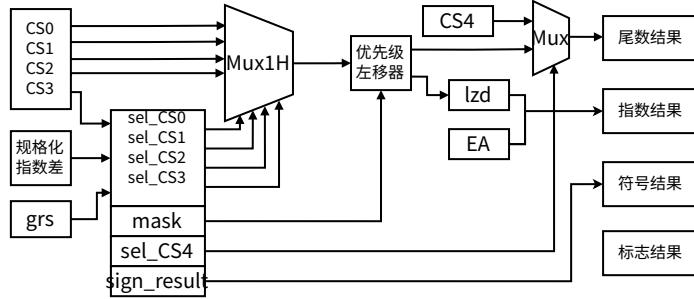


图 12.3: *close* 路径算法示意图

第一步，并行做四组有效数字减法，根据 $d = 0$ (fp_a 尾数大, fp_b 尾数大), $d = 1$ (fp_a 规格化的指数大, fp_b 规格化的指数大)，组合出有效减法的四种情况。第一个减法器： fp_a 有效数字 - fp_b 有效数字；第二个减法器： fp_b 有效数字 - fp_a 有效数字；第三个减法器： fp_a 有效数字 $\times 2 - fp_b$ 有效数字；第四个减法器： fp_b 有效数字 $\times 2 - fp_a$ 有效数字。同时根据指数大小关系计算出 grs 位， $d = 0$ 时 grs 全为 0， $d = 1$ 时只有 g 可能非 0。这四组加法器不能产生所有舍入的结果，增加第五个慢速加法器，指数大的有效数字 - 指数小的有效数字右移一位。

第二步，确定选择四组有效数字减法的四个条件，根据 d 的值，加法器结果的最高位， grs 和舍入模式来确定。从四组加法器中选出减法结果后需要对减法结果进行 $LZD +$ 左移，这里要注意较大指数 EA 的值，左移要受 LZD 和 EA 共同控制，根据 EA 的值产生一个 $mask$ 值（与减法结果位宽相同但最多只有一比特是 1），与减法结果或操作后再进行 $LZD +$ 左移即可。

第三步，确定选择第五个减法器的条件，选择第五个减法器结果时是不需要左移的，所以采用的慢速加法器，最终的尾数结果就可以选择出来了。

第四步，指数结果和符号位结果，指数位结果需要用 EA 减第二步中的 LZD 的值，如果选择的是第五个减法器作为尾数结果，则指数保持原值。当 $d = 1$ 时符号位的取值就是指数较大的操作数的符号， $d = 0$ 时要根据尾数大小选择符号位，注意结果为 0 且向下舍入时，符号位为 1。

12.4.1.4 向量浮点加法算法

向量浮点加法器输出信号宽度为 64，支持混合精度，支持 *widening* 指令，共要支持如下数据格式计算：

- (1) 1 个 $f64 = f64 + f64$;
- (2) 1 个 $f64 = f64 + f32$;
- (3) 1 个 $f64 = f32 + f32$;
- (4) 2 个 $f32 = f32 + f32$;
- (5) 2 个 $f32 = f32 + f16$;
- (6) 2 个 $f32 = f16 + f16$;
- (7) 4 个 $f16 = f16 + f16$ 。

12.4.1.4.1 模块划分

计算思路是用一个模块计算前三种格式，它们输出的结果都是 64 位，将计算 $f64 = f64 + f64$ 的单精度浮点加法器复用，使之能计算 $f64 = f64 + f32$ 和 $f64 = f32 + f32$ ，本文提出一种快速的数据格式转换算

法，将 $f32$ 操作数转换为 $f64$ 后就可以进行 $f64 = f64 + f64$ 计算，得到 $f64$ 的结果格式。

对于输出结果是 $f32$ 的计算格式也采取同样的思路进行，因为 $f32$ 的时序压力没那么大，将一个 $f16 = f16 + f16$ 也融入到计算结果为 $f32$ 的模块节约面积，使它支持：

- (1) 1 个 $f32 = f32 + f32$;
- (2) 1 个 $f32 = f32 + f16$;
- (3) 1 个 $f32 = f16 + f16$;
- (4) 1 个 $f16 = f16 + f16$ 。

显然这个模块需要例化两个，最后还差 2 个 $f16 = f16 + f16$ ，单独例化两个只计算 $f16 = f16 + f16$ 的单精度浮点加法器，一共四个模块，实现所有向量加法计算格式。

12.4.1.4.2 快速格式转换算法

以 $f16$ 转换成 $f32$ 为例，介绍快速格式转换算法。

当 $f16$ 为规格化数时，转换成 $f32$ 也一定是规格化数。对于 $f16$ 指数要偏置到 $f32$ 的指数，因为 $f32$ 的指数范围更大，所以不用担心指数转换之后出现越界的问题，另外 $f16$ 尾数是 10 位的， $f32$ 尾数是 23 位的，只需在 $f16$ 尾数后补 13 个零就可以得到 $f32$ 的尾数，并且这是一个低精度向高精度的转换，结果肯定是精确的。

对于规格化的 $f16$ 的指数（位宽是 5），实际指数 $Ereal = Ef16 - 15$ ，对于规格化的 $f32$ 的指数（位宽是 8）， $Ereal = Ef32 - 127$ ，所以借助 $Ereal$ 将 $Ef16$ 转换到 $Ef32$ ， $Ef16 - 15 = Ef32 - 127$ ， $Ef32 = Ef16 - 15 + 127$ ， $Ef32 = Ef16 + 112$ ，112 的 8 位二进制表示为 01110000，计算 $Ef16 + 112$ 需要一个变量加一个常数的加法器，通过发现规律可以避免这个加法器，规律如下：

当 $Ef16$ 最高位是 0 时， $Ef16 + 112 = 0111\ Ef16\ 3\ 0$

当 $Ef16$ 最高位是 1 时， $Ef16 + 112 = 1000\ Ef16\ 3\ 0$

通过这个规律可以用一个 Mux 快速进行 $Ef16$ 到 $Ef32$ 的转换，所以对于规格化数 $f16$ 转换为 $f32$ 是很快的，指数位用一个 Mux ，尾数位补 0，符号位不变。难点在于当 $f16$ 是非规格化数的时候，此时 $f16$ 指数全为 0，而尾数的前导零个数决定了转换成 $f32$ 的指数，当 $f16$ 的指数位全是零，尾数位只有 lsb 为 1 时，转换成 $f32$ 的指数最小，为 $-15 - 9 = -24$ ，此时仍在 $f32$ 规格化数内。所以对于 $f16$ 非规格化数，要进行尾数的前导零检测 lzd 和左移。

Chisel 自带的优先级编码可以实现 lzd 功能，本文实测比传统使用二分法写的 lzd 综合效果更好。语法为：
PriorityEncoder(Reverse(Cat(in, 1.U)))。*in* 的位宽为 5 生成的 *verilog* 代码如下：

```
module LZDPriorityEncoder(
    input      clock,
    input      reset,
    input [4:0] in,
    output [2:0] out
);
    wire [5:0] _out_T = {in, 1'h1};
    wire [5:0] _out_T_15 = {_out_T[0], _out_T[1], _out_T[2], _out_T[3], _out_T[4], _out_T[5]};
    wire [2:0] _out_T_22 = _out_T_15[4] ? 3'h4 : 3'h5;
    wire [2:0] _out_T_23 = _out_T_15[3] ? 3'h3 : _out_T_22;
    wire [2:0] _out_T_24 = _out_T_15[2] ? 3'h2 : _out_T_23;
    wire [2:0] _out_T_25 = _out_T_15[1] ? 3'h1 : _out_T_24;
    assign out = _out_T_15[0] ? 3'h0 : _out_T_25;
endmodule
```

虽然这段代码看起来使用了很多级联的 *Mux*, 但是综合器对于这种代码综合的时序比较好。受此启发, 本文设计了一种新型基于优先级的左移算法来加快 *lzd+ 左移*, 其 *Chisel* 代码如下:

```

def shiftLeftPriorityWithF32EXPResult(srcValue: UInt, priorityShiftValue: UInt): UInt = {
    val width = srcValue.getWidth
    val lzdWidth = srcValue.getWidth.U.getWidth
    def do_shiftLeftPriority(srcValue: UInt, priorityShiftValue: UInt, i:Int): UInt = {
        if (i==0) Cat(
            Mux(
                priorityShiftValue(i),
                Cat(srcValue(0), 0.U((width-1).W)),
                0.U(width.W)
            ),
            Mux(
                priorityShiftValue(i),
                "b01110000".U-(width-i-1).U(8.W),
                "b01110000".U-(width-i).U(8.W)
            )
        )
    }
    else Mux(
        priorityShiftValue(i),
        if (i==width-1) Cat(srcValue(i,0), "b01110000".U-(width-i-1).U(8.W))
        else Cat(Cat(srcValue(i,0), 0.U((width-1-i).W)), "b01110000".U-(width-i-1).U(8.W)),
        do_shiftLeftPriority(srcValue = srcValue, priorityShiftValue = priorityShiftValue, i = i - 1)
    )
}
do_shiftLeftPriority(srcValue = srcValue, priorityShiftValue = priorityShiftValue, i = width-1)
}

```

srcValue 和 *priorityShiftValue* 都传 *f16* 的尾数, 从尾数最高位开始进行判断, 尾数最高位为 1 则返回 *srcValue* 原值, 同时返回相应的指数 (指数是从多个常数中选一个, 和尾数中第一个 1 出现的位置有关), 如果最高位是 0, 则接着判断下一位是否为 1, 若为 1, 将 *srcValue* 左移一位后返回 (此处不需要真正的左移, 因为不需要保留左移之后高位的结果, 所以直接进行截位操作再拼接零即可), 同时返回相应的指数, 以此类推。这样用一个优先级左移器同时做了 *lzd+ 左移* 两步操作, 并且还产生了对应的 *Ef32*, 省掉根据 *lzd* 计算 *Ef32* 指数, 从而实现了 *f16* 是非规格数的情况下转换成 *f32* 的快速算法。*f32* 转换成 *f64* 采用的算法类似, 不再赘述。

12.4.2 向量浮点融合乘加算法

浮点融合乘加计算 $fpa \times fp_b + fp_c$, 中间的乘法计算 $fpa \times fp_b$ 就像没有范围和精度限制一样, 不进行舍入, 最终只舍入一次到目标格式。FMA 通常作采用流水线实现, 步骤主要包括乘法、加法、归一化移位和舍入。本章介绍向量浮点融合乘加算法, 其功能包括:

- (1) 1 个 $fp64 = fp64 \times fp64 + fp64$;
- (2) 2 个 $fp32 = fp32 \times fp32 + fp32$;

- (3) 4 个 $fp16 = fp16 \times fp16 + fp16$;
- (4) 2 个 $fp32 = fp16 \times fp16 + fp32$;
- (5) 1 个 $fp64 = fp32 \times fp32 + fp64$ 。

(1) (2) (3) 中的源操作数和目的操作数都是同一种浮点数格式, (4) (5) 中两个乘数的宽度相同, 另一个加数和结果的宽度相同并且是乘数宽度的二倍。

12.4.2.1 标量单一精度格式算法

计算流程是先计算两个浮点数相乘的不舍入的结果, 再用这个不舍入的乘积和第三个数相加。算法流程图如图, 用公式 $fp_result = fp_a \times fp_b + fp_c$ 表达计算过程, 图中 $S_a S_b S_c$ 分别是 $fp_a fp_b fp_c$ 的有效数字, $E_a E_b E_c$ 则是 $fp_a fp_b fp_c$ 的指数:

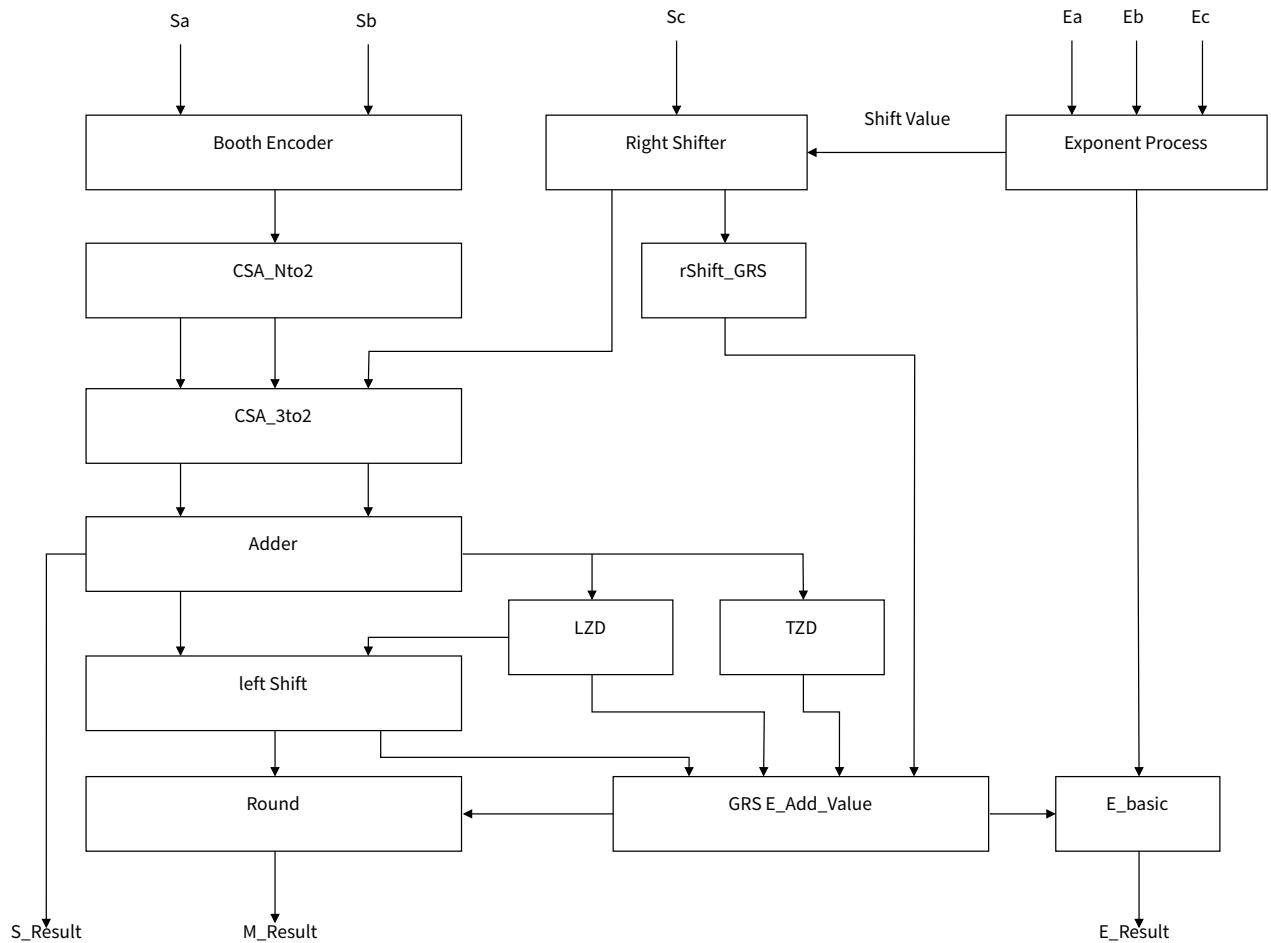


图 12.4: FMA 算法流程图

为了下文描述方便, 定义一些参数, 参数含义及取值如表:

表 12.4: 参数含义及不同精度下的取值

| 参数 | $f16$ | $f32$ | $f64$ | 含义 |
|--------------------|-------|-------|-------|-----------------------------|
| $significandWidth$ | 11 | 24 | 53 | 有效数字宽度 |
| $exponentWidth$ | 5 | 8 | 11 | 指数宽度 |
| $rshiftBasic$ | 14 | 27 | 56 | fp_c 有效数字与乘积有效数字对齐需右移的位数 |

| 参数 | $f16$ | $f32$ | $f64$ | 含义 |
|-------------|-------|-------|-------|--|
| $rshiftMax$ | 37 | 76 | 163 | fp_c 有效数字最大的右移位数（超过这个值 g 和 r 都是 0） |

12.4.2.1.1 无符号整数乘法

两个浮点数相乘规则是符号位相乘、指数位相加（不是单纯相加，需要考虑偏置）、有效数字（包括隐含位和尾数位）相乘。有效数字乘法实际上是定点数乘法，其和无符号整数乘法原理相同。

二进制竖式乘法是原始的乘法算法， n 比特 $C = A \times B$ 竖式法如图。此过程会产生 n 个部分积，并将这些部分积错位相加。

竖式法的乘法算法延时很大，针对乘法运算的优化，主要集中在两个方面：一是减少部分积的个数（比如 *Booth* 编码），二是减少加法器带来的延时（比如 *CSA* 压缩）。

在计算两个浮点相乘时，会将它们的有效数字相乘，有效数字是无符号位的，所以使用无符号整数乘法就可以计算两个有效数字相乘。无符号整数乘法实现的算法有很多种，下面对比三种算法。

方式一：直接使用乘号 \times ，由综合工具自己决定。

方式二：使用类似手算十进制乘法的竖式法，两个 n 位数相乘会产生 n 个部分积，再对 n 个部分积进行 *CSA* 压缩（后面介绍），压缩成两个数相加。

方式三：使用 *Booth* 编码，产生 $(n + 1)/2$ 向上取整个部分积，再通过 *CSA* 压缩成两个数相加。

表中的数据是使用 *TSMC7nm* 工艺库对两个 53 比特无符号整数相乘（对于 $f64$ ）的结果。目标频率为 $3GHz$ ，理论每周期时间 $333.33ps$ ，但是需要考虑 *clock uncertain* 和工艺角偏差问题，给后端留设计余量，每周期可用时间为 $280ps$ ，所以显然一周期内做不完乘法，实际中求隐含位还需要一定的时间，所以更加无法在一周期内实现 $53bit$ 相乘。因此虽然方式一面积更小延时更短一些，但是因为方式一无法进行流水线设计，也只能选择方式二或者方式三，方式三相较与方式二延时更短面积更小，所以选方式三作为无符号整数乘法的实现方式。

表 12.5: 无符号整数乘法三种算法比较

| 算法 | 延时 (ps) | 面积 (um^2) | 是否可流水 |
|-----|---------|---------------|-------|
| 方式一 | 285.15 | 1458.95 | 否 |
| 方式二 | 320.41 | 2426.34 | 是 |
| 方式三 | 302.19 | 2042.46 | 是 |

12.4.2.1.2 *Booth* 编码

Booth 编码目的是减少乘法器部分积的个数，以二进制无符号数整数乘法 $C = A * B$ 为例推导 *Booth* 编码算法。

下式是二进制无符号整数通用的表达形式，为了后续变换，在头尾各加上一个 0，其值不变。

$$B = 2^{n-1}B_{n-1} + 2^{n-2}B_{n-2} + \dots + 2B_1 + B_0 + B_{-1}, \quad B_{-1} = 0$$

等价变换后，相邻两比特 1 会被抵消为 0。如果是连续 1，则最低位的 1 会变成 -1 ，最高位 1 的上一位由 0 变成 1，其中 1 全部变成 0，这种变换就是布斯变换。布斯变换能对连续 1 的数量大于等于 3 起到化简作用，连续 1 的位数越多，化简效果越好。但上述变换并不能在硬件电路中起到优化作用，因为并不能保证某个部分积恒为 0。所以在电路设计中，一般采用改进的布斯编码方式，能真正减少部分积的数量。

| | | An-1 | | A3 | A2 | A1 | A0 |
|----------|--------|--------|--------|--------|--------|--------|------|
| | × | Bn-1 | | B3 | B2 | B1 | B0 |
| | | An-1B0 | | A4B0 | A3B0 | A2B0 | A1B0 |
| | An-1B1 | | A4B1 | A3B1 | A2B1 | A1B1 | A0B1 |
| | An-1B2 | | A4B2 | A3B2 | A2B2 | A1B2 | A0B2 |
| An-1B3 | | A4B3 | A3B3 | A2B3 | A1B3 | A0B3 | |
| | | ⋮ | | | | | |
| | | ⋮ | | | | | |
| An-1Bn-1 | | A4Bn-1 | A3Bn-1 | A2Bn-1 | A1Bn-1 | A0Bn-1 | |
| | | | | | C3 | C2 | C1 |
| | | | | | | | C0 |
| Cn-1 | | | | | | | |

图 12.5: 二进制竖式法计算乘法

$$\begin{aligned}
 B &= 2^{n-1}B_{n-1} + 2^{n-2}B_{n-2} + \dots + 2B_1 + B_0 + B_{-1} \\
 &= 2^{n-1}B_{n-1} + 2^{n-2}B_{n-2} + 2^{n-2}B_{n-2} - 2^{n-2}B_{n-2} + \dots + 2B_1 + 2B_1 - 2B_1 + B_0 + B_0 - B_0 + B_{-1} \\
 &= 2^{n-1}(B_{n-1} + B_{n-2}) + 2^{n-2}(-B_{n-2} + B_{n-3}) + \dots + 2(-B_1 + B_0) + (-B_0 + B_{-1})
 \end{aligned}$$

再次进行等价变换，不过这次对 n 加一些约束，假设 n 为奇数，依旧在最后一位补一个零，补零之后长度变为偶数，再在最高位前面补一个零，将总长度变为 $n+2$ ，这样做的目的是方便后面推导。

$$\begin{aligned}
 B &= 2^nB_n + 2^{n-1}B_{n-1} + 2^{n-2}B_{n-2} + \dots + 2B_1 + B_0 + B_{-1} \\
 &= -2 \times 2^{n-1}B_n + 2^{n-1}B_{n-1} + 2^{n-2}B_{n-2} + 2^{n-2}B_{n-2} - 2^{n-2}B_{n-2} + \dots \\
 &\quad + 2^3B_3 + 2^3B_3 - 2^3B_3 + 2^2B_2 + 2B_1 + 2B_1 - 2B_1 + B_0 + B_{-1} \\
 &= 2^{n-1}(-2B_n + B_{n-1} + B_{n-2}) + 2^{n-2}(-2B_{n-1} + B_{n-2} + B_{n-3}) + \dots \\
 &\quad + 2^2(-2B_3 + B_2 + B_1) + (-2B_1 + B_0 + B_{-1})
 \end{aligned}$$

通过等价变形后可以发现，多项表达式的项数变成了 $n+1/2$ (n 为奇数)，如果 n 为偶数需要在尾部补一个零，最高位前面补两个零，多项表达式的项数变成了 $n/2+1$ (n 为偶数)，综合奇偶数的情况，多项表达式的项数为 $n+1/2$ 向上取整。原二进制数从 LSB 开始，以三位为一组（第一组最低位需补一个附加位 0 即，最高位根据 n 的奇偶，奇数补一个 0，偶数补两个 0，目的是让补完 0 后的长度是奇数），相邻两组之间重叠一位（低位组的最高位与高位组的最低位重叠），构成新的多项式因子，这就是改进的布斯编码方式。

两个二进制数相乘，如果对乘数进行改进型布斯编码，则得出的部分积个数可以减少一半。记被乘数为 A ，乘数为 B , B_{2i+1} , B_{2i} , B_{2i-1} ，分别为 X 的连续三位，其中 i 为自然数 N , PP_i 为 i 不同取值下对应的部分积，对 B 行改进的布斯变换后再与 A 乘，Booth 编码与 PP 真值表如表中所示。

表 12.6: Booth 编码与 PP 真值表

| B_{2i+1} | B_{2i} | B_{2i-1} | PP_i |
|------------|----------|------------|--------|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | A |

| B_{2i+1} | B_{2i} | B_{2i-1} | PP_i |
|------------|----------|------------|--------|
| 0 | 1 | 0 | A |
| 0 | 1 | 1 | $2A$ |
| 1 | 0 | 0 | $-2A$ |
| 1 | 0 | 1 | $-A$ |
| 1 | 1 | 0 | $-A$ |
| 1 | 1 | 1 | 0 |

根据乘数每连续三位的值，求出其对应的部分积，减少了一半部分积数量，就好像将乘数看作是四进制数进行乘法，因此被称为基 4 布斯编码。采用基 4 布斯编码的乘法相较于传统乘法运算，优化效果很明显且易于实现，可以满足大部分应用要求。

在 Booth 编码时需要计算五种部分积的情况，0、 A 、 $2A$ 、 $-A$ 、 $-2A$ ，0 和 A 无需计算， $2A$ 左移一位即可， $-A$ 和 $-2A$ 需要取反加一的操作，本文介绍一种快速处理取反加一的算法。

为介绍原理简单，我们假设计算 f_{16} ，有效数字是 11 位，会生成 6 个部分积，部分积的宽度是 22 位的，如图所示，图中彩色位置宽度是 12 位的，表示 A 可能乘 0、乘 1 或者乘 2。因为最后一个部分积三比特的编码是 0xx，所以其值不可能是负数，我们假设除了最后一个部分积外其它部分积都是负数，则需要对其余每个部分积取反加一，彩色部分表示的是仅取反后的结果，我们把当前部分积加的一放到下一个部分积对应的位置，这样保证了部分积的和不变，并且避免了对当前部分积加一产生一个进位链的问题，而最后一个部分积是非负的，不需要处理它的加一。



上图中的 1 可以先进行求和化简，得到下图的结果。

如果实际部分积的值是正数，那么需要对以上结果进行修正，也就是在彩色左边一比特位置加一，并且把下一个部分积尾部加的一变成零。如图所示， Si (i 从 0 计) 表示第 i 个部分积的符号位，这样就变成了通用的形式，彩色位置只计算 0、 A 、 $2A$ 、 $\sim A$ 、 $\sim 2A$ ，加快部分积生成速度。

这里还有一点需要说明，部分积的和是乘积的结果，但是部分积求和的时候也可能会进位，这个进位对于乘法来说是无意义的，但是当乘积和更宽的数字相加的时候就会错误进位，修正方式是部分积最高位再增加一位，如图所示。

这样就可以保证所有部分积加起来之后进位是正确的，到此 Booth 编码介绍结束，注意是以 11 比特乘法为例介绍的， f_{16} 和 f_{64} 有效数字位数为奇数，但是 f_{32} 有效数字位数为偶数，最高位补零需要稍作区别，其他过程类似，不再赘述。

| | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | | | | | | | | | | | | | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | | | | | | | | | | | | | | | 1 | |
| 0 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | |
| 1 | 0 | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | |

图 12.7: 假设部分积全为负数化简后的结果

| | | | | | | | | | | | | | | | | | | | | | |
|----|-----|----|-----|----|-----|----|-----|----|----|----|----|---|---|---|---|---|---|---|---|----|----|
| 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | ~S0 | S0 | S0 | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | ~S1 | | | | | | | | | | | | | | S0 |
| 0 | 0 | 0 | 0 | 1 | ~S2 | | | | | | | | | | | | | | | S1 | |
| 0 | 0 | 1 | ~S3 | | | | | | | | | | | | | | | | | | |
| 1 | ~S4 | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | |

图 12.8: 部分积为正数时修正结果

| 修正位 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|-----|----|-----|----|-----|----|-----|----|----|----|----|---|---|---|---|---|---|---|---|----|----|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ~S0 | S0 | S0 | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | ~S1 | | | | | | | | | | | | | | S0 |
| 0 | 0 | 0 | 0 | 0 | 1 | ~S2 | | | | | | | | | | | | | | | S1 | |
| 0 | 0 | 0 | 1 | ~S3 | | | | | | | | | | | | | | | | | | |
| 0 | 1 | ~S4 | | | | | | | | | | | | | | | | | | | | |
| 1 | | | | | | | | | | | | | | | | | | | | | | |

图 12.9: 部分积进位修正

12.4.2.1.3 CSA 压缩

Carry-Save-Adder 保留进位加法器，作用是将 n 个加数通过某些逻辑压缩成 m 个加数 ($m < n$)，典型的是 3_2 压缩和 4_2 压缩。

假设计算两个二进制数相加 $A + B$ ，它们的和与进位真值表，表中 $A[i] + B[i]$ 是十进制结果，也是 $A[i]$ 、 $B[i]$ 中 1 的数量：

表 12.7: $A + B$ 和与进位真值表

| $A[i]$ | $B[i]$ | $A[i] + B[i]$ | $Sum[i]$ | $Car[i]$ |
|--------|--------|---------------|----------|----------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 2 | 0 | 1 |

化简成逻辑表达式如下：

$$Sum = A \wedge B$$

$$Car = A \& B$$

$$Result = A + B = Sum + (Car << 1)$$

三个数相加，和是两个数异或操作，进位是两个数同为 1，($Car << 1$) 是因为当前比特的进位是进到下一比特去的，这里只是做一下推导方便后面理解，实际中两个加数产生和与进位对加法没有加速效果。

假设我们要计算三个数的和 $A + B + C$ ，CSA 关键是产生和与进位，真值表如表所示：

表 12.8: $A + B + C$ 和与进位真值表

| $A[i]$ | $B[i]$ | $C[i]$ | $A[i] + B[i] + C[i]$ | $Sum[i]$ | $Car[i]$ |
|--------|--------|--------|----------------------|----------|----------|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 2 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 2 | 0 | 1 |
| 1 | 1 | 0 | 2 | 0 | 1 |
| 1 | 1 | 1 | 3 | 1 | 1 |

从上表中可以看出一些规律，产生 $Sum[i]$ 和 $Car[i]$ 其实只和 $A[i] + B[i] + C[i]$ 的和有关，即 $A[i]$ 、 $B[i]$ 、 $C[i]$ 中 1 的个数，化简之后的表达式如下：

$$Sum = A \wedge B \wedge C$$

$$Car = (A \& B) \mid (A \& C) \mid (B \& C)$$

$$Result = A + B + C = Sum + (Car << 1)$$

三个数相加，和是三个数异或操作，进位是至少有两个数为 1，($Car << 1$) 是因为当前比特的进位是进到下一比特去的。这样通过两级异或门的延迟就可以把三个数相加转化成两个数相加，这种方式节省了很多时间，尤其是在位宽比较长的情况下。

四个数相加情况复杂一些，因为四个数同时为 1 时，总和为 4，需要进位 2，我们把其中一个进位叫做 $Cout$ ，另一个进位叫 Car ，然后把当前四比特相加产生的 $Cout$ 传到下一级叫做 Cin ， Cin 和四个数变成五个数相加，这样操作之后每一比特输入就变成了五个数， $A[i]$ 、 $B[i]$ 、 $C[i]$ 、 $D[i]$ 、 $Cin[i]$ ，输出三个数 $Sum[i]$ 、 $Cout[i]$ 、 $Car[i]$ ，最低位的 $Cin[0]$ 是 0，其他位的 $Cin[i]$ 是低一比特的 $Cout[i-1]$ ，如表所示。

表 12.9: $A + B + C + D$ 和与进位真值表

| $A[i] + B[i] + C[i] + D[i] + Cin[i]$ | $Sum[i]$ | $Cout[i]$ | $Car[i]$ |
|--------------------------------------|----------|-----------|----------|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 2 | 0 | 1/0 | 0/1 |
| 3 | 1 | 1/0 | 0/1 |
| 4 | 0 | 1 | 1 |
| 5 | 1 | 1 | 1 |

这个真值表的化简方式有很多种，下面介绍一种可行的方式， $Sum[i]$ 的值容易得出是五个输入的异或， $Sum[i] = A[i]B[i]C[i]D[i]Cin[i]$ ， $Car[i]$ 和 $Cout[i]$ 较复杂，我们规定 $Cout[i]$ 只由前三个数产生，即前三个数的和大于 1 时， $Cout[i] = 1$ ，表是 $Cout[i]$ 真值表：

表 12.10: $A + B + C$ 产生 $Cout$ 真值表

| $A[i] + B[i] + C[i]$ | $Cout[i]$ |
|----------------------|-----------|
| 0 | 0 |
| 1 | 0 |
| 2 | 1 |
| 3 | 1 |

$Cout[i]$ 可表示为： $Cout[i] = (A[i] \wedge B[i])?C[i] : A[i]$ ， $Car[i]$ 则由 $D[i]$ 、 $Cin[i]$ 产生，表是 $Car[i]$ 的真值表。

表 12.11: $A + B + C$ 产生 Car 真值表

| $A[i] + B[i] + C[i] + D[i]$ | $Car[i]$ |
|-----------------------------|----------|
| 0 | $D[i]$ |
| 1 | $Cin[i]$ |
| 2 | $D[i]$ |
| 3 | $Cin[i]$ |

$Car[i]$ 可表示为： $Car[i] = (A[i] \wedge B[i] \wedge C[i] \wedge D[i])?Cin[i] : D[i]$ ，具体来看 $(A[i] \wedge B[i] \wedge C[i] \wedge D[i]) = 1$ 时， $A[i] + B[i] + C[i] + D[i] = 1/3$ ，此时 $Cin[i] = 1$ 会产生进位， $(A[i] \wedge B[i] \wedge C[i] \wedge D[i]) = 0$ 时， $A[i] + B[i] + C[i] + D[i] = 0/4$ ，此时 $D[i]$ 等于 0 表示 $A[i] + B[i] + C[i] + D[i] = 0$ ，再加 Cin 也不产生进位， $D[i]$ 等于 1 表示 $A[i] + B[i] + C[i] + D[i] = 4$ ，再加 Cin 也会产生进位，所以综合以上推导过程， $CSA4_2$ 的表达式如下：

$$\begin{aligned}
 Sum[i] &= A[i] \wedge B[i] \wedge C[i] \wedge D[i] \$ \wedge Cin[i] \quad Cin[i] = Cout[i-1] \quad Cin[0] = 0 \\
 Cout[i] &= (A[i] \wedge B[i])?C[i] : A[i] \\
 Car[i] &= (A[i] \wedge B[i] \wedge C[i] \wedge D[i])?Cin[i] : D[i] \\
 Result &= A + B + C + D = Sum + (Car << 1)
 \end{aligned}$$

使用 *TSMC7nm* 工艺库对不同输入异或门、*CSA3_2*、*CSA4_2* 分别进行综合对比延时和面积，不同输入异或门综合结果见表。

表 12.12: 不同输入异或门综合结果

| 106bit | 延时 (ps) | 面积 (μm^2) |
|------------------|---------|------------------|
| $A \wedge B$ | 13.74 | 38.66880 |
| $A^B C$ | 23.01 | 63.09120 |
| $A^B C \wedge D$ | 24.69 | 87.51360 |
| $A^B C^D E$ | 37.21 | 99.72480 |

CSA3_2 *CSA4_2* 综合结果见表。

表 12.13: *CSA3_2* *CSA4_2* 综合结果

| 106bit | 延时 (ps) | 面积 (μm^2) |
|---------------|---------|------------------|
| <i>CSA3_2</i> | 23.23 | 104.42880 |
| <i>CSA4_2</i> | 40.63 | 237.86881 |

可见 *CSA4_2* 虽说理论上是三级异或门的延迟，*CSA3_2* 理论上是两级异或门的延迟，但是真正物理实现之后，*CSA4_2* 只是比两级 *CSA3_2* 快一点点，所以尽量用 *CSA3_2*，除非一级 *CSA4_2* 可以替代两级 *CSA3_2* 的情况，比如 $4- > 2$ 压缩， $8- > 2$ 压缩。

12.4.2.1.4 *CSAn_2*

两个无符号整数乘法使用 *Booth* 编码产生的部分积数量为 $\frac{n+1}{2}$ 向上取整，因为部分积压缩之后要和位宽更大的数相加，为保证进位正确，部分积位宽拓展一比特，各数据格式部分积数量及位宽见表。

表 12.14: 不同数据格式部分积数量及位宽

| 数据格式 | 有效数字位数 | 部分积数量 | 部分积位宽 |
|-------------|--------|-------|-------|
| <i>fp16</i> | 11 | 6 | 12 |
| <i>fp32</i> | 24 | 13 | 25 |
| <i>fp64</i> | 53 | 27 | 54 |

按照尽量用 *CSA3_2*，除非一级 *CSA4_2* 可以替代两级 *CSA3_2* 的原则，各数据格式所用 *CSA3_2* *CSA4_2* 级数如表。

表 12.15: 不同数据格式部分积 CSA 压缩过程

| 数据格式 | <i>CSA3_2</i> 级数 | <i>CSA4_2</i> 级数 | 过程 ($->$ 表示 <i>CSA3_2</i> , $-->$ 表示 <i>CSA4_2</i>) |
|-------------|------------------|------------------|---|
| <i>fp16</i> | 1 | 1 | $6 -> 4 --> 2$ |
| <i>fp32</i> | 3 | 1 | $13 -> 9 -> 6 -> 4 --> 2$ |
| <i>fp64</i> | 3 | 2 | $27 -> 18 -> 12 -> 8 --> 4 --> 2$ |

12.4.2.1.5 指数处理和右移

如果按照常规做法, fp_a 和 fp_b 相乘结果的指数和 fp_c 的指数结果大小关系未知, 那么就像计算浮点加法那样需要对其中指数小的进行右移, 那么 fp_a 和 fp_b 相乘结果的尾数和 fp_c 的尾数都可能右移, 这样设计会造成使用两个移位器, 增大了面积, 并且需要等 fp_a 和 fp_b 相乘结果计算出来再对它的有效数字右移, 增大了电路的延迟, 现介绍一种算法避免使用两个移位器, 并且可以和 fp_a 和 fp_b 相乘并行起来减小电路的延迟。

首先指数位是当作无符号数处理的, 但是它和真实指数之间有一个指数偏置, 同时还要考虑 *denormal* 的情况, 记 E_fix 是处理 *denormal* 情况之后的指数位, 记 E_bit 是原始的指数位, 当 E_bit 所有位是 0 时 $E_fix = 1$, 其他情况下 $E_fix = E_bit$ 。

$$E_{real} = E_{fix} - bias, \quad bias = (1 << (exponentWidth - 1)) - 1$$

上式中真正的指数 E_{real} 等于 E_{fix} 减去一个偏置值 $bias$, $exponentWidth$ 是 E_{bit} 的宽度, $bias$ 的值等于 E_{bit} 最高位是 0 其余位全是 1。在不考虑有效数字乘积的进位或借位的情况下, fp_a 和 fp_b 相乘的真实指数结果 Eab_{real} 如下式所示:

$$Ea_{real} = Ea_{fix} + Eb_{fix} - 2 \times bias$$

fp_a 和 fp_b 相乘的二进制指数结果 Eab_{bit} 的计算公式见下式:

$$Eab_{bit} = Cat(0.U, Ea_{fix} + \&Eb_{fix}).asSInt - bias.S$$

其中 $+&$ 的操作将 $Ea_{fix} + Eb_{fix}$ 的结果拓展一位保留进位, 保留进位的原因是因为后面还要减去一个偏置值, 不保留进位结果就不对了, 还要考虑减去偏置值可能出现负数的情况, 所以再拓展一位, 即在最高位拼一比特 0, 最后再减去偏置值 $bias$, 这样就得到了在不考虑有效数字乘积的进位或借位的情况下, fp_a 和 fp_b 相乘的二进制指数结果 Eab_{bit} 。然后我们构造一个指数 Eab , 其值如下式:

$$Eab = Eab_{bit} + rshiftBasic.S$$

并且认为 fp_a 和 fp_b 相乘的二进制指数结果为 Eab , 为了保证无损精度的 fp_a 和 fp_b 相乘的有效数字和 fp_c 的有效数字相加, 我们分别将这两个加数拓展位宽, 对于 fp_c 的有效数字拓展成 $3 \times significandWidth + 4$, 位分布情况见图, 其中 $g0\ r0\ g1\ r1$ 是用来保存右移过程中 *round round* 位的:

如上图所示, fp_c 有效数字比 fp_a 和 fp_b 有效数字乘积高 $significandWidth + 2$ 位, 因为乘积结果小数点前面有两位, 所以按照乘积结果是 1.xxx 对齐, 高 $significandWidth + 3$ 位, 这就是 $rshiftBasic = significandWidth + 3$ 的原因。

| $3 \times significandWidth + 4$ | | | | | | |
|---------------------------------|------------------|----|----|-----------------------------|----|----|
| 位宽 | significandWidth | 1 | 1 | $2 \times significandWidth$ | 1 | 1 |
| 含义 | fp_c有效尾数 | g0 | r0 | 与fp_a和fp_b有效尾数乘积对齐 | g1 | r1 |

图 12.10: fp_c 有效数字扩展位分布情况

令 $fp_c_significand_cat0 = Cat(fp_c_significand, 0.U(2 \times significandWidth + 4))$, $fp_c_significand$ 为 fp_c 的有效数字。如果 $Ec_fix = Eab = Eab_bit + rshiftBasic.S$, $fp_c_significand_cat0$ 正好比 Eab_bit 大 $significandWidth + 3$, $fp_c_significand_cat0$ 不用右移就对齐了; 如果 $Ec_fix > Eab$, 理论上讲 $fp_c_significand_cat0$ 需要左移, 但是因为中间有 $g0 g1$ 做间隔, 并且低比特是不会产生进位的, 低比特只会影响舍入结果, 所以实际不需要对 $fp_c_significand_cat0$ 左移; 如果 $Ec_fix < Eab$, 此时就要对 $fp_c_significand_cat0$ 进行右移了, 右移的位数 $rshift_value = Eab - Cat(0.U, Ec_fix).asSInt$ 。 $rshift_value$ 的是来自多个数的加法, 所以最低位先计算好, 所以再右移时要先用 $rshift_value$ 最低位当作 Mux 的选择信号, 再依次用高位当作 Mux 的选择信号, 右移过程需要计算 *round round sticky* (简称 *grs*), 对于 *round* 和 *sticky*, 已经在拓展位宽时保留了该位置, 无需额外计算, 对于 *sticky* 有两种方法, 方法一是再拓展一些位宽用来存放右移出去的位, 等全部右移结束之后再进行求 *sticky*, 方法二是在右移过程中根据 Mux 选择信号的值同时计算出 *sticky*, 方法二比方法一延时更小。下面是方法二的设计代码:

```
/*
 * 使用Mux进行移位, 先用最低位, 输出位宽为srcValue + 1(Sticky)
 */
def shiftRightWithMuxSticky(srcValue: UInt, shiftValue: UInt): UInt = {
    val vecLength = shiftValue.getWidth + 1
    val res_vec = Wire(Vec(vecLength, UInt(srcValue.getWidth.W)))
    val sticky_vec = Wire(Vec(vecLength, UInt(1.W)))
    res_vec(0) := srcValue
    sticky_vec(0) := 0.U
    for (i <- 0 until shiftValue.getWidth) {
        res_vec(i+1) := Mux(shiftValue(i), res_vec(i) >> (1<<i), res_vec(i))
        sticky_vec(i+1) := Mux(shiftValue(i), sticky_vec(i) | res_vec(i)((1<<i)-1,0).orR,
                               sticky_vec(i))
    }
    Cat(res_vec(vecLength-1), sticky_vec(vecLength-1))
}
```

在右移的时候还有一个加快速度的方法, $rshift_value$ 的位宽是 $exponentWidth + 1$, 而 $fp_c_significand_cat0$ 宽度是 $3 * significandWidth + 4$, $rshift_value$ 中可能会有溢出的位宽, 比如用一个位宽是 5 的数去右移一个宽是 7 的数, $a(6,0) >> b(4,0)$, b 中的第三位最大值是 7, 已经够了 a 的位宽, 所以 b 的高两位只要有非零数, a 的右移结果就是零, 右移结果可简化为 $Mux(b(4,3).orR, 0.U, a(6,0) >> b(2,0))$, 下表列出三种浮点数据格式使用的 $rshift_value$ 位宽。

表 12.16: 不同浮点格式使用的 $rshift_value$ 位宽

| 数据格式 | $fp_c_significand_cat0$ 位宽 | $rshift_value$ 位宽 | 使用的位宽 |
|-------|-------------------------------|--------------------|-------|
| $f16$ | 37 | 6 | 6 |
| $f32$ | 76 | 9 | 7 |
| $f64$ | 163 | 12 | 8 |

根据 $rshift_value$ 的值分为三种情况, $rshift_value \leq 0$ 此时不需要右移, $sticky$ 结果是 0; $rshift_value > rshiftMax$, 右移结果是 0, $sticky$ 结果是 $fp_c_significand_cat0$ 或缩减; $0 < rshift_value \leq rshiftMax$, 右移结果和 $sticky$ 是 $shiftRightWithMuxSticky$ 计算的结果。

至此, 本小节介绍了指数处理的方法, 右移器设计, 右移过程中 grs 的处理。

12.4.2.1.6 有效数字相加

fp_c 有效数字右移之后的结果 $rshift_result$ 要和 $CSAn_2$ 压缩的两个结果进行相加, 由于 fp_c 和 fp_a 乘以 fp_b 的符号可能相反, 当相反时要做减法, 并且减法结果可能是负数, 为了判断正负需要再补充一位符号位, $fp_c_rshiftValue_inv$ 根据符号是否相反选择 $rshift_result$ (符号位补 0) 或者 $rshift_result$ 取反 (符号位补 1), 所以要用 $fp_c_rshiftValue_inv$ 要 $CSAn_2$ 压缩的两个结果进行相加, 不过在做减法的时候 $fp_c_rshiftValue_inv$ 只是对 $rshift_result$ 取反, 还需要在右移 grs 都为 0 的情况下对最低位 +1, 把这一个 +1 放到 $CSAn_2$ 压缩的两个结果中的 $carry$ 位, 因为 $carry$ 位恒为 0, 直接放到这一位可以节省加法器的使用, 节省面积。这三个数位宽不同, fp_c 有效数字右移之后的结果位宽 $3 \times significandWidth + 4$, $CSAn_2$ 压缩的两个结果位宽是 $2 \times significandWidth + 1$, +1 的原因是之前提到的部分积要拓展一位保证进位正确。计算三个不同位宽的和的策略是, 先将 $CSAn_2$ 压缩的两个结果的低 $2 \times significandWidth + 1$ 位和 $rshift_result$ 低 $2 \times significandWidth$ 位 (最高位拼一比特 0 拼成 $2 \times significandWidth + 1$) 进行一次 $CSA3_2$ 压缩之后, 再对 $CSA3_2$ 压缩的两个结果相加, 记作 $adder_low_bits$, 同时计算 $rshift_result$ 高 $significandWidth + 4$ 位 +1, 然后根据低 $2 \times significandWidth + 1$ 位相加的结果最高位是否为 1 来选择 $fp_c_rshiftValue_inv$ 高 $significandWidth + 4$ 位或者 $fp_c_rshiftValue_inv$ 高 $significandWidth + 4$ 位 +1, 记作 $adder_high_bits$ 。

还需要考虑减法时对右移 grs 取反加一, 最终有效数字的加法结果 $adder$ (含右移 grs) 构成是: $adder_high_bits$ 、 $adder_low_bits$ 、右移 grs (减法时取反加一)。因为 $adder$ 可能是负数, 拓展了 1bit 仅用于 $adder$ 的符号判断, 之后不需要使用, $adder_inv$ 在 $adder$ 是负数时进行取反并且去掉了该符号判断位。

12.4.2.1.7 LZD、左移、舍入与未舍入尾数结果

在计算完 $adder_inv$ 后, 需要对 $adder_inv$ 进行前导零检测, 以确定需要左移的位数, 进而对尾数结果规格化和舍入。

对 $adder_inv$ 求 LZD 时有一个指数限位的问题, 记 $E_greater$ 为 Eab (fp_a 与 fp_b 相乘的指数结果), 左移的值不能比 $E_greater$ 更大, 因为等于 $E_greater$ 时, 指数结果已经全为 0 了。为了解决这个问题, 同浮点加法器一样在左移时使用 $mask$ 进行限位。

对于 $adder$ 是负数的情况, $-adder$ 应该是 $adder$ 取反 +1, 由于 +1 会产生长进位链, 只做取反操作, 然后计算 $adder_inv$ 的 LZD , 此时可能会出现一位的偏差, 当 $adder$ 取反之后末尾全是连续 1 时, 此时再加 1 最高位的 1 就要产生进位了, 解决一比特偏差的方法是对 $adder$ 做一个尾随零检测 (TZD), 当

$LZD + TZD = adder$ 的宽度时, $adder$ 取反之后末尾全是连续 1, 这种情况下需要对左移结果修正。左移修正之后得到未舍入的结果, 对未舍入的结果 +1 得到舍入后的结果。

12.4.2.1.8 最终结果

根据 adder 的正负得到符号位结果, grs 的计算需要同时结合第五步右移和第七步左移过程。舍入策略采用 *after rounding*, 为了判断 *underflow*, 需要再使用一组专门用来判断 *underflow* 的 grs 。根据舍入模式和 grs 得到是否需要舍入, 选择出尾数结果, 指数结果根据舍入情况得到。

输入操作数含特殊值，如 NaN ，无穷，零时，结果单独计算，并根据实际输入数的情况从特殊结果和正常结果中选择一个，标志位结果除了除零标志位外，其他四种都可产生。

12.4.2.2 向量单一精度格式算法

向量操作主要设计思路是在时序满足要求的情况下共用硬件。

在 Booth 编码过程中, $f16$ 产生 6 个 pp , $f32$ 产生 13 个 pp , $f64$ 产生 27 个 pp , 所以 Booth 编码时 $f64$ 产生的 27 个 pp 的位置可以放两个 $f32$ 产生的 13 个 pp , 同理 $f32$ 产生的 13 个 pp 的位置可以放两个 $f16$ 产生的 6 个 pp , 这样就可以继续共用一套 CSA_27to2 压缩, 向量共用 Booth 编码见图。

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| f64 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| 2*F32 | 0 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 4*F16 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 0 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

图 12.11: 向量共用 Booth 编码示意图

在 fp_c 尾数右移时， $f64$ 和 $f32$ 中的其中一个尾数右移可以共用一个右移器，其他的右移器都是独立的。

CSA_3to2 也是公用的，第三个操作数来自 *fp_c* 尾数右移的结果，将 2 个 *f32* 或 4 个 *f16* 尾数右移的结果拼接起来，和共用的 *Booth* 编码压缩之后的两个操作数一起进行 *3_2* 压缩。

压缩完之后的加法器也是共用的，通过不同格式分配不同的位，并按位隔开使低位进位不影响高位结果。

LZD TZD，左移器共用思路同右移器， $f64$ 和 $f32$ 中的一个是公用的，其他独立。

12.4.2.3 向量混合精度格式算法

向量混合精度格式计算有两种：

- (1) 2 个 $fp32 = fp16 \times fp16 + fp32$;
 - (2) 1 个 $fp64 = fp32 \times fp32 + fp64$ 。

对于两个相同宽度的乘数，本质还是指数相加有效数字相乘，不用向浮点加法那样先把它们格式转换成和结果相同的格式，只需简单拓展位宽即可，指数高位补零，尾数低位补零，和高精度的浮点操作数对齐。对齐后，按照单一精度格式进行计算即可。

12.4.3 向量浮点除法算法

除法是现代处理器中最具代表性的浮点函数之一。在硬件中计算除法存在两个主要的算法：具有线性收敛性并基于减法的数字迭代算法，以及基于乘法并具有二次收敛性的乘法算法，基于减法的数字迭代算法能效更高，所需面积更小。本文后续的数字迭代都是指基于减法的数字迭代。对于常用的浮点精度，双精度、单精度和半精

度、数字迭代方法要快得多。在数字迭代除法中，最关键的一点是商数位的选择，每次迭代，获得商的一位数。要实现独立于除数的简单 $\text{Radix} - 4$ 选择函数，除数需要调整到足够接近 1 的值。该缩放在数字迭代之前执行。

数字迭代算法由于在性能、面积和功耗方面具有很好的折衷性而被广泛应用于的高性能处理器中，本文基于 SRT 除法 (*Sweeney – Robertson – Tocher Division*)，采用 $\text{Radix} - 64$ 的浮点除法算法，每周期计算 6 比特商。为了减小开销，每次 $\text{Radix} - 64$ 迭代由三次 $\text{Radix} - 4$ 迭代组成；在连续的 $\text{Radix} - 4$ 迭代之间使用推测算法减少延时。

12.4.3.1 标量浮点除法算法

本文实现的 $\text{Radix} - 64$ 标量浮点除法算法，当输入操作数和结果都是规格化数时，对于双精度、单精度、半精度浮点除法，有较低的延迟，分别为 11 6 4 周期延迟，这些延迟包括缩放和舍入的周期。在输入操作数或结果中有非规格化数的情况下，需要一个或两个额外的规格化周期。

指数结果很容易就能得出，重点是有效数字的除法，有效数字除法器执行被除数有效数字 x 和除数有效数字 d 的浮点除法，以获得有效数字的商 $q = x/d$ 。两个操作数需要是规格化数， $x, d \in [1, 2]$ ，非规格化的操作数也是允许的，在数字迭代之前，对非正规化操作数进行规格化。如果两个操作数在 $[1, 2]$ 中规格化，则结果在 $[0.5, 2]$ 中；这样，需要商的最低有效位 (LSB) 右边的两位用于舍入，即保护位和舍入位。

当结果被归一化时，保护位用于舍入， $q \in [1, 2]$ ，当结果未被归一化时，舍入位用于舍入， $q \in [0.5, 1]$ 。在后一种情况下，结果左移 1 位，保护位和舍入位分别成为归一化结果的 LSB 和保护位。为了简化舍入，结果被强制为 $q \in [1, 2]$ 。注意只有当 $x > d$ 时 $q > 1$ 。这种情况在早期被检测到，并将被除数左移 1 位，使得 $q = 2 \times x/d$ 且 $q \in [1, 2]$ ，注意此时指数结果要进行相应调整。

用于除法的算法是 $\text{Radix} - 4$ 数字迭代算法，每个循环三次迭代，商的符号数字表示为数字集 $\{-2, -1, 0, +1, +2\}$ ；即基数 $r = 4$ ，数据集 $a = 2$ 。每次迭代，通过选择函数获得商的一个数字。为了具有独立于除数的商位选择函数，除数必须缩放到接近 1。当然，为了保持结果的正确性，被除数需要按与除数相同的倍数缩放。

利用基 -4 算法，每次迭代获得商的 2 比特。由于每个时钟周期执行三次基 -4 迭代，因此每个周期获得 6 位商，这等效于 $\text{Radix} - 64$ 除法器。另外，注意到作为结果的整数位的第一商数位只能取值 $\{+1, +2\}$ ，并且其计算比其余数位的计算简单得多。把它和操作数预缩放并行计算，节省了一次单精度浮点数的迭代。另一方面，对于异常操作数有一种提前终止模式。当任何操作数为 Nan 、无穷大或零时，或者在两个操作数均归一化的情况下除以 2 的次幂的情况下，引发提前终止。在后一种情况下，仅通过减小被除数的指数来获得结果。 $\text{Radix} - 64$ 除法器的主要特征如下：

- (1) 除数和被除数的预缩放。
- (2) 第一个商数位与预缩放并行执行。
- (3) 比较缩放后的被除数和除数，并将被除数左移，以得到 $[1, 2]$ 的结果。
- (4) 每个周期三次 $\text{Radix} - 4$ 迭代，每个周期 6 位。
- (5) 支持半精度、单精度和双精度。
- (6) 非规格化数支持，迭代前需要额外一周期将非规格化数规格化。
- (7) 异常操作数提前终止。

12.4.3.1.1 数字迭代除法算法

数字迭代除法是一种迭代算法，每次迭代计算一个 $radix - r$ 商数字 q_{i+1} 和一个余数。余数 $rem[i]$ 用于获得下一个 $radix - r$ 的数字。为了快速迭代，余数被保存在有符号数冗余表示的进位保存加法器中，本文选择了 $radix - 2$ 的有符号数表示余数，其中包含一个正数和一个负数。对于基数 $r = 4$ ，下式是第 i 位迭代之前的部分商的表达式

$$Q[i] = \sum_{j=0}^i q_j \times 4^{-j}$$

除数缩放到 1 附近之后的 $radix - 4$ 算法，商和余数由下式描述：

$$q_{i+1} = SEL(\widehat{rem}[i])$$

$$rem[i+1] = 4 \times rem[i] - d \times q_{i+1}$$

其中 $\widehat{rem}[i]$ 是只具有几个比特的余数 $rem[i]$ 的估计值。对于此算法，已确定仅需要余数的 6 个最高有效位 (*MSB*)，即 3 个整数位和 3 个小数位。然后，每次迭代从当前余数中获得商位，并为下一次迭代计算新的余数。那么，下式是迭代次数 it 计算公式：

$$it = [n/\log_2(4)]$$

其中 n 是结果的位数，包括舍入所需的位。除法的延迟，即周期数，与迭代次数直接相关。它还取决于每个周期执行的迭代次数。每个周期已经实现三次迭代以获得每个周期 6 位，这等效于 $Radix - 64$ 除法。那么规范化浮点数所需周期 $cycles$ 由下式确定，除了迭代所需的 $(it/3)$ 个周期之外，有两个额外的周期用于操作数预缩放和舍入。

$$cycles = [it/3] + 2$$

数字迭代除法的一些实例，包括 $Radix - 4$ 算法，可以在 [38] 中找到。简单的实现如图所示。注意，只有余数的最高有效位用于选择商数位。余数使用进位保存加法器 (*CSA*) 更新并存储在冗余表示中。然后，商数位选择需要余数的 t 个最高有效位在进位传播加法器 (*CPA*) 中相加以获得其非冗余表示。但是这种实现太慢了，为了加速迭代的循环，需要使用迭代之间的余数计算和商数位选择中的推测算法。

12.4.3.1.2 操作数预缩放

在预缩放期间，除数被缩放到接近 1 的值，使得商数位的选择与除数无关。对于 $radix - 4$ 算法，缩放除数在 $[1 - 1/64, 1 + 1/8]$ 范围内就足够了。如表预缩放因子真值表所示，仅三比特确定比例因子。注意预缩放时，除数应该被扩大 $1 - 2$ 倍。被除数也应该缩放相同的比例因子。

表 12.17: 预缩放因子真值表

| 0.1xxx | 预缩放因子 |
|--------|-----------------|
| 000 | $1 + 1/2 + 1/2$ |
| 001 | $1 + 1/4 + 1/2$ |

| 0.1xxx | 预缩放因子 |
|--------|-----------------|
| 010 | $1 + 1/2 + 1/8$ |
| 011 | $1 + 1/2 + 0$ |
| 100 | $1 + 1/4 + 1/8$ |
| 101 | $1 + 1/4 + 0$ |
| 110 | $1 + 0 + 1/8$ |
| 111 | $1 + 0 + 1/8$ |

12.4.3.1.3 整数位商计算

整数位商计算的同时，为数字迭代步骤（每个数字迭代执行三个 $radix - 4$ ，对应 $s0 s1 s2$ 三个阶段）提供以下数据：

- (1) 冗余余数的进位保存表示： $f_r_s f_r_c$ 。
- (2) 预缩放后的除数： $divisor$ 。
- (3) 为第一次数字迭代中 $s0$ 阶段商选择提供 6 比特余数结果。
- (4) 为第一次数字迭代中 $s1$ 阶段商选择提供 7 比特余数结果。

12.4.3.1.3.1 数字迭代

浮点除法器的实际实现需要每个周期执行三次 $radix - 4$ 迭代，常规顺序迭代三次速度太慢，无法满足时序要求，对逻辑进行了优化。图示出了数字迭代循环的框图。

- (1) 对除数进行处理，得到五种可能的商选择结果需要用到的除数倍数（商为负数时只取反）。
- (2) $s0$ 阶段，使用四个 CSA 模块（商为 0 时不需要），并行在 $s0$ 预测性的计算出 $s1$ 阶段需要的 5 个余数冗余表示。
- (3) $s0$ 阶段，使用第二步中计算出的 5 个余数冗余表示，为 $s2$ 阶段预测性的计算 5 个 7 比特余数结果。
- (4) $s0$ 阶段，根据输入信号中 6 比特余数结果，选择出 $s0$ 阶段的商，商使用 5 比特独热码表示。
- (5) 根据 $s0$ 阶段的商，选择出 $s1$ 阶段需要使用的冗余余数表示，同时从第三步为 $s2$ 阶段预测性的计算 5 个 7 比特余数结果中选择出一个。
- (6) $s1$ 阶段，使用四个 CSA 模块（商为 0 时不需要），并行在 $s1$ 预测性的计算出 $s2$ 阶段需要的 5 个余数冗余表示。
- (7) $s1$ 阶段，根据输入信号中 7 比特余数结果、5 种商选择结果用到的的除数倍数、 $s0$ 阶段的商，预测性的进行 $s1$ 阶段的商选择。
- (8) 根据 $s1$ 阶段的商，选择出 $s2$ 阶段需要使用的冗余余数表示。
- (9) $s2$ 阶段，使用四个 CSA 模块（商为 0 时不需要），并行在 $s2$ 预测性的计算出下一个数字迭代 $s0$ 阶段需要的 5 个余数冗余表示。
- (10) $s2$ 阶段，分别为下一次数字迭代中 $s0$ 阶段需要的 6 比特余数结果和 $s1$ 阶段需要的 7 比特余数结果预测性的计算 5 种可能的结果。
- (11) $s2$ 阶段，根据第五步中为 $s2$ 阶段选出的的 7 比特余数结果、5 种商选择结果用到的的除数倍数、 $s1$ 阶段的商，预测性的进行 $s2$ 阶段的商选择。
- (12) 根据 $s2$ 阶段的商选择结果，为下一次数字迭代选择出：冗余余数的进位保存表示， $s0$ 阶段需要的 6 比特余数结果， $s1$ 阶段需要的 7 比特余数结果。

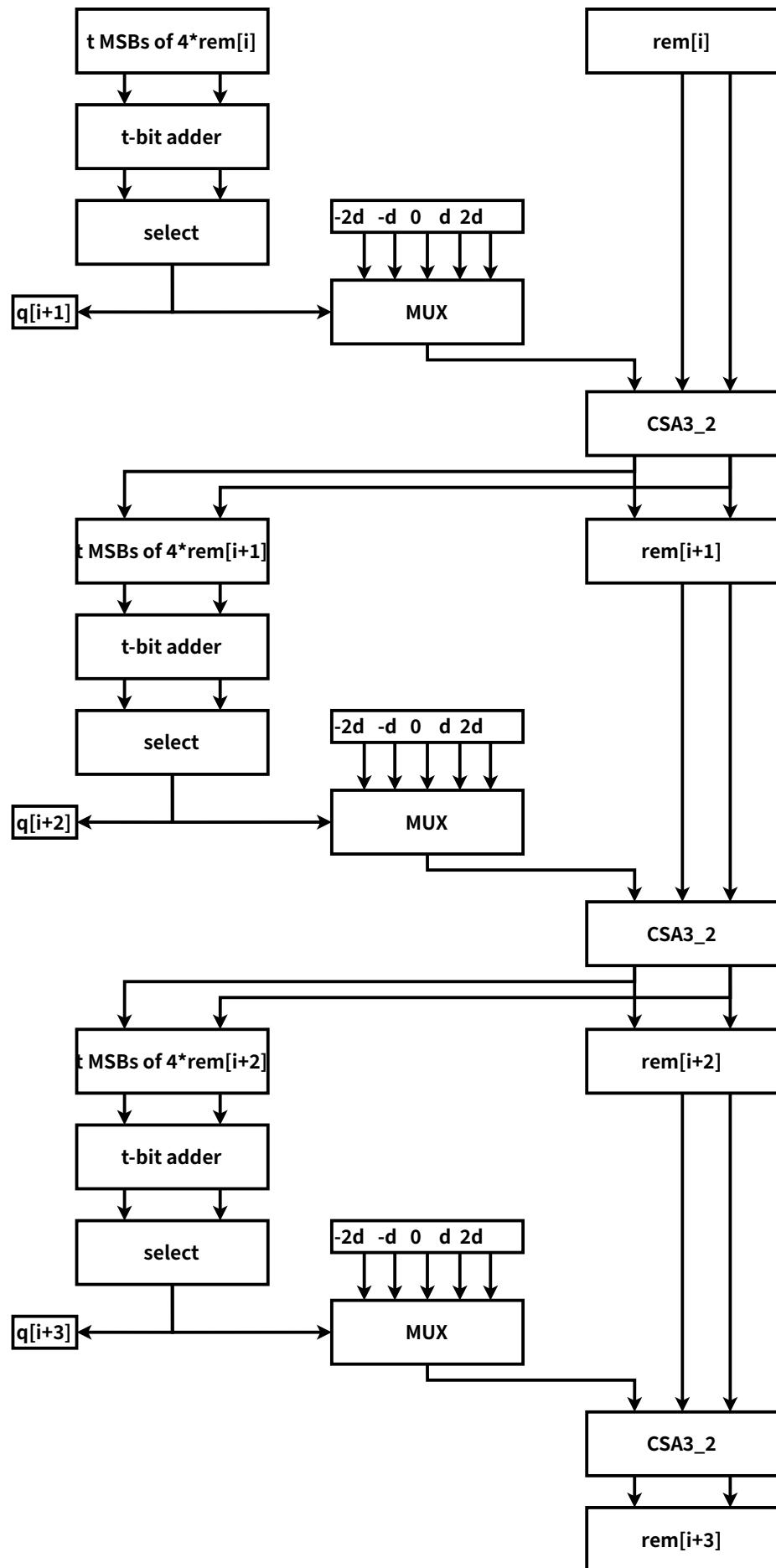


图 12.12: 三次 radix-4 组成 radix-64 的简单实现

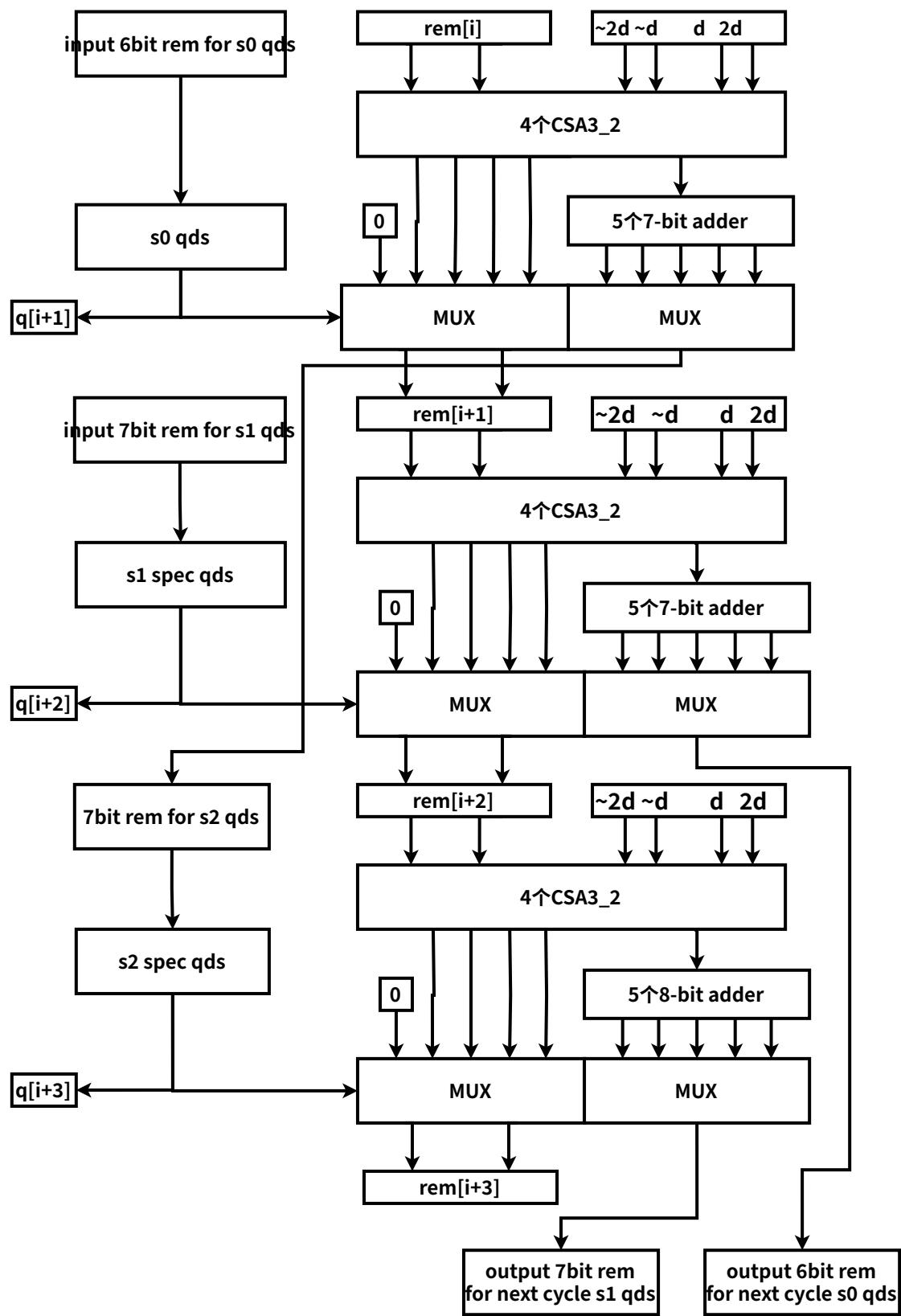


图 12.13: 数字迭代优化算法

由于第一步中对除数的倍数只取反，没有 $+1$ ，导致余数计算时会有偏差，在商选择过程中添加修正逻辑来修正，下表是标准的商选择函数，下表是逻辑修正后的商选择函数。

表 12.18: 标准商选择函数

| $4 \times rem[i]$ | q_{i+1} |
|-------------------|-----------|
| $[13/8, 31/8]$ | +2 |
| $[4/8, 12/8]$ | +1 |
| $[-3/8, 3/8]$ | 0 |
| $[-12/8, -4/8]$ | -1 |
| $[-32/8, -13/8]$ | -2 |

表 12.19: 逻辑修正后的商选择函数

| $4 \times rem[i]$ | $carry$ | q_{i+1} |
|-------------------|---------|-----------|
| $31/8$ | 1 | +2 |
| $[13/8, 30/8]$ | - | +2 |
| $12/8$ | 0 | +2 |
| $12/8$ | 1 | +1 |
| $[4/8, 11/8]$ | - | +1 |
| $3/8$ | 0 | +1 |
| $3/8$ | 1 | 0 |
| $[-3/8, 2/8]$ | - | 0 |
| $-4/8$ | 0 | 0 |
| $-4/8$ | 1 | -1 |
| $[-12/8, -5/8]$ | - | -1 |
| $-13/8$ | 0 | -1 |
| $-13/8$ | 1 | -2 |
| $[-32/8, 14/8]$ | - | -2 |

将数字迭代输出的冗余余数表示恢复成标准余数，通过 *On the Fly Conversion* 将商和商减一计算出来，为商和商减一计算两套 *grs* 和是否需要向上舍入一位，计算选商或商减一的选择信号并选出正确的商结果，最后用正确的商结果和对应的是否需要向上舍入一位信号进行舍入。

12.4.3.1.3.2 非规格化数和提前终止

(1) 输入含有非规格化数，非规格化数的有效数字小于 1，无法和规格化数一起进行预缩放，为此多增加一个周期将非规格化数的有效数字进行规格化，并同时调整非规格化数的指数。

(2) 结果是非规格化数，数字迭代后的商结果是大于 1 的，不符合非规格化有效数字范围，需要多增加一个周期对商结果右移来规格化。

(3) 提前终止，有两种情况：当结果是 *NaN*、无穷、精确 0 的时候，提前终止计算输出结果，因为在第一周期就能得到这个信息，第二周期就可以输出除法结果；当除数是 2 的次幂时，其有效数字 = 1，除法只需对被

除数的指数处理，可跳过数字迭代阶段，最快也可第二周期输出除法结果，但是当被除数或结果是非规格化数时，同样也需要额外的周期去处理。

12.4.3.2 向量浮点除法算法

对于向量浮点除法，*RISC-V* 向量指令集拓展没有混合精度的浮点除法，以此只需支持：

- (1) 1 个 $f64 = f64 + f64$;
- (2) 2 个 $f32 = f32 + f32$;
- (3) 4 个 $f16 = f16 + f16$ 。

考虑到向量除法计算中同时有多组除法计算，而提前终止功能会导致多组数据结果输出不同步，除非都是相同情况的提前终止，所以对于向量除法取消提前终止的机制，如果发生了提前终止的情况，让结果在内部暂存，和其他除法结果同时输出。

为了统一时序，将除法器的周期数统一规定为最差情况下的周期数，即输入含有非规格化数，输出也含有非规格化数的情况，其他原本可以更快输出结果的情况都进行内部数据暂存，等到规定的周期数后输出结果。

主体采用资源复用的方式进行设计，在非数字迭代模块进行如下数据复用：

- (1) 1 个 $f64/f32/f16 = f64/f32/f16 + f64/f32/f16$;
- (2) 1 个 $f32/f16 = f32/f16 + f32/f16$;
- (3) 2 个 $f16 = f16 + f16$ 。

共使用 4 组信号完成 7 组除法的功能。

由于数字迭代模块是关键路径，时序压力比较大，不能实现向非数字迭代模块部分的高度复用，否则时序不满足要求，为此对数字迭代模块进行部分复用设计：

- (1) 接口是 4 组商和冗余余数。
- (2) s_0 阶段采用 7 组 *CSA* 和 7 组预测，4 组商选择。
- (3) $s_1 s_2$ 阶段使用 4 组 *CSA*，4 组预测，4 组商选择。

对于寄存器也采用资源复用的方式，对于除数，冗余余数，商等寄存器，按照 *max* (4 个 $f16$, 2 个 $f32$, 1 个 $f64$) 所需的比特数分配大小。

12.5 硬件设计

12.5.1 向量浮点加法器

12.5.1.1 标量单一精度浮点加法器

根据改进的双通路浮点加法算法设计标量单一精度浮点加法器，其硬件实现架构如图所示。

左侧两个输入操作数 fp_a 和 fp_b ，右侧 fp_c 表示加法结果， $fflags$ 是 5 比特异常标志位， rm 是舍入模式，共五种舍入模式，使用 3 比特来表示， is_sub 为 0 时计算 $fp_c = fp_a + fp_b$ ， is_sub 为 1 时计算 $fp_c = fp_a - fp_b$ ，浮点加法与浮点减法的区别只在于 fp_b 的符号位，所以只需要对 fp_b 的符号位稍作处理就可以将浮点加法器同时支持浮点减法的计算。整体由三部分组成：*far* 通路、*close* 通路、异常通路。

far 通路先并行做两个规格化指数减法有效数字右移，分别处理 $Efp_a \geq Efp_b$ 和 $Efp_b \geq Efp_a$ 的情况，根据 Efp_a 和 Efp_b 大小关系选出正确的右移结果，送入 $FS0$ 和 $FS1$ 两个有效数字加法器。*far* 通路对于减法时 EA 是大的指数减一，加法时 EA 是大的指数，这样有效数字加法结果的值域就在 $[1, 4)$ 之间，右移时计算两套 *grs*: *grs_normal* 用于值域在 $[1, 2)$ 之间的舍入，*grs_overflow* 用于值域在 $[2, 4)$ 之间的舍入。最后根据 $FS0$ 的结果和舍入模式来选择 $FS0$ 或 $FS1$ 作为尾数结果，同时根据舍入情况选择 EA 或

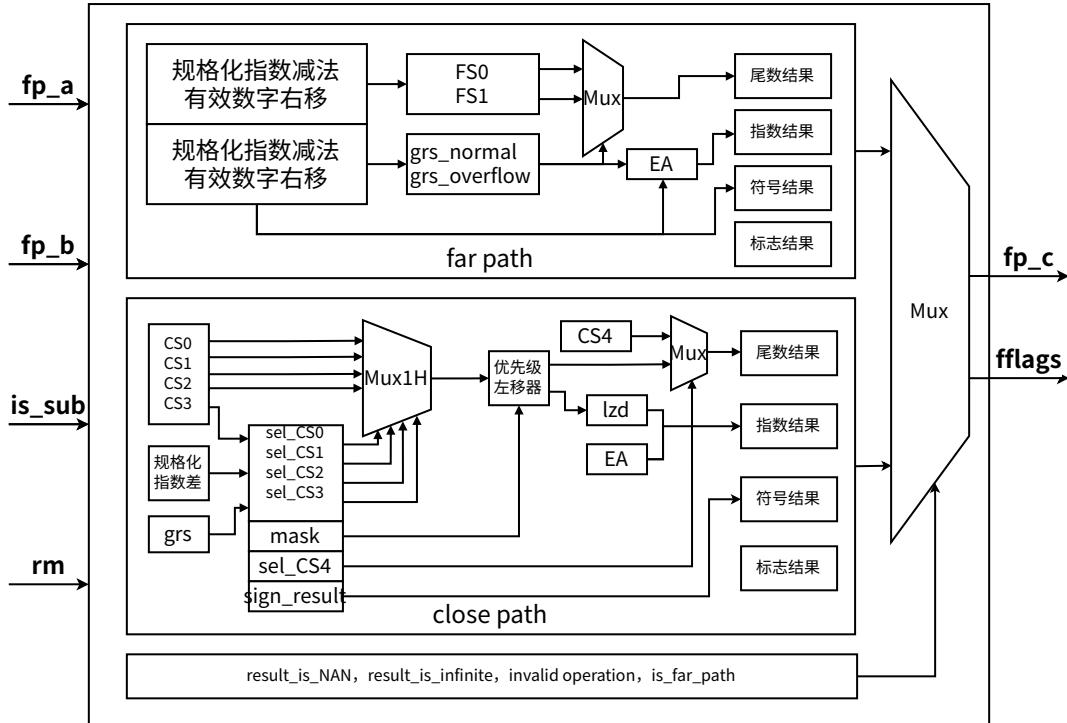


图 12.14: 标量单一精度浮点加法器架构图

$EA + 1$ 作为指数结果，符号位结果则根据指数大小情况就能得到，标志结果根据 $EA + 1$ 是否为全 1 判断是否上溢，根据 grs 判断是否不精确，*far* 通路不会产生除零，无效操作，下溢标志位。

close 通路使用 $CS0, CS1, CS2, CS3$ 四个有效数字加法器处理 $Efp_a = Efp_b, Efp_a = Efp_b + 1, Efp_a = Efp_b - 1$ 三种情况下的有效数字减法，然后根据 $CS0$ 结果和 grs 产生四个独热选择信号 $sel_{CS0}, sel_{CS1}, sel_{CS2}, sel_{CS3}$ ，使用一个四输入独热码多路选择器 $Mux1H$ ，选出一个结果和左移 $mask$ 进行或操作，然后使用的优先级左移器进行尾数规格化，左移的同时输出 lzd 的值，指数结果为 $EA - lzd$ ，尾数结果则需要在尾数规格化后和 $CS4$ 之间选择一个作为尾数结果， $CS4$ 是补充的舍入结果，不需要进行左移规格化。符号结果则由指数差和 $CS0$ 的结果计算得到，标志结果只会产生不精确，其他异常标志位均不会产生。

异常通路是用来判断是否为无效操作，结果是否为 NaN ，结果是否为无穷，当这三种情况均为否时，进行正常计算，产生选择信号，选择 *far* 通路或 *close* 通路中的结果和标志位作为输出。

12.5.1.2 标量混合精度浮点加法器

在标量单一精度浮点加法器基础上进行混合精度硬件设计，主要区别是支持混合精度的计算，以结果为 $f32$ 为例，下表是 res_widen 和 opb_widen 对应操作的真值表。

表 12.20: 结果为 $f32$ 混合精度格式表

| res_widen | opb_widen | $f32$ |
|--------------|--------------|-------------------|
| 0 | 0 | $f32 = f32 + f32$ |
| 1 | 0 | $f32 = f16 + f16$ |
| 1 | 1 | $f32 = f16 + f32$ |
| 0 | 1 | 不允许 |

下图是标量混合精度浮点加法器架构图，其中最主要的区别是在数据输入端加一个快速格式转换模块，根据操作类型将操作数统一转换成结果的数据格式后，和单一精度浮点加法计算流程相同。

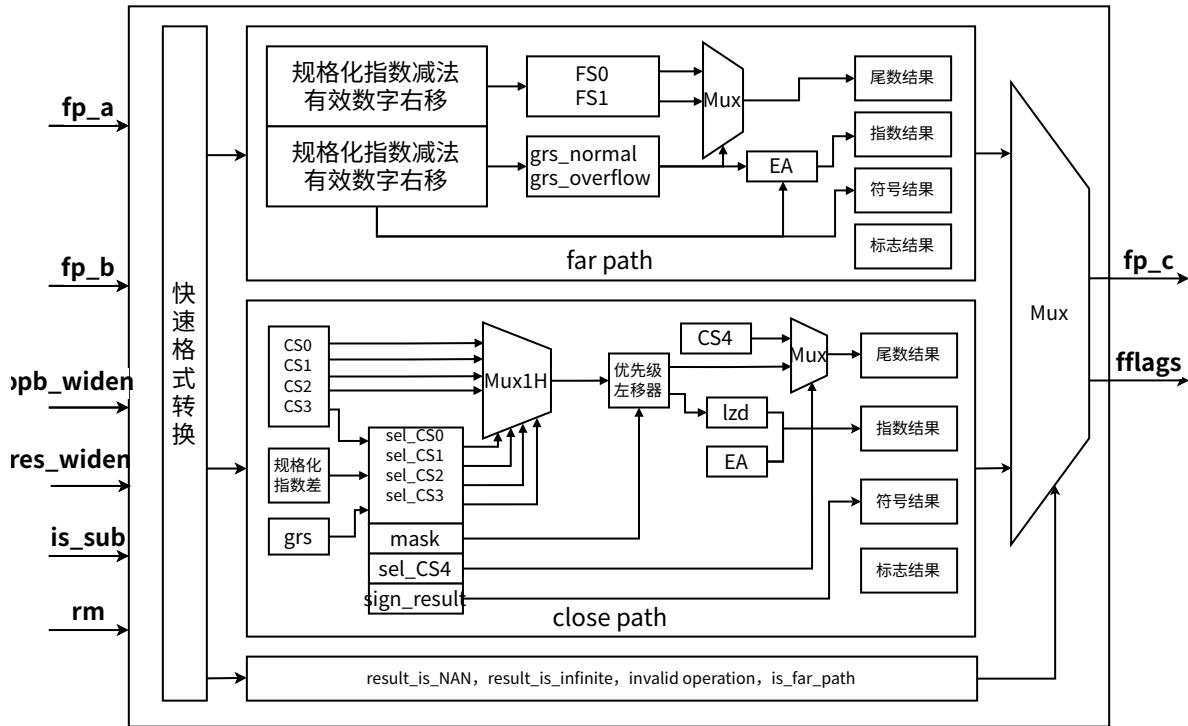


图 12.15: 标量混合精度浮点加法器架构图

12.5.1.3 向量浮点加法器

下图是向量浮点加法器架构图，为了满足时序要求，采用四个模块组合而成，*FloatAdderF64Widen* 计算所有输出结果为 64 位的操作，*FloatAdderF32WidenF16* 计算所有输出结果为 16 位或 32 位的操作，*FloatAdderF16* 只计算输出结果为 16 位的操作。

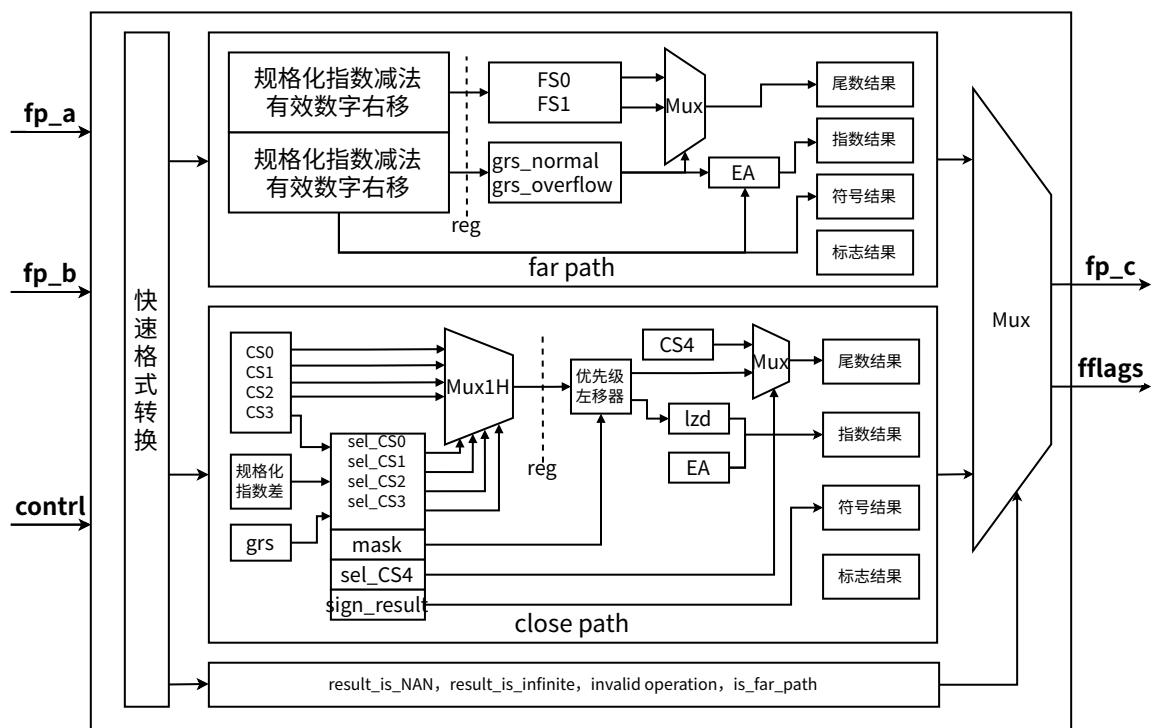
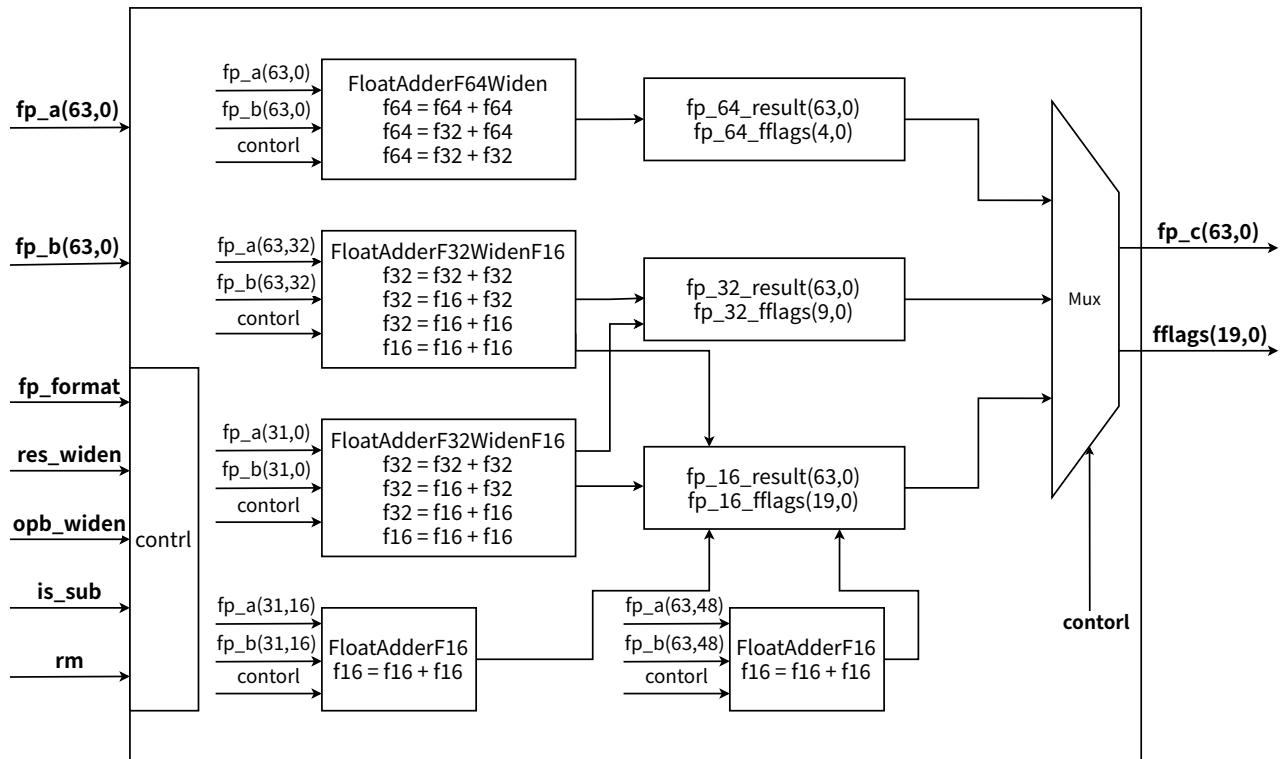
其中 *fp_format* 为 2 比特结果格式控制信号，00 表示结果格式是 *f16*，01 表示结果格式是 *f32*，10 表示结果格式是 *f64*，输出的标志位为 20 比特，低有效排列，当结果格式是 *f16* 时，20 比特全部有效，当结果格式是 *f32* 时，低 10 比特有效，当结果格式是 *f64* 时，低 5 比特有效。

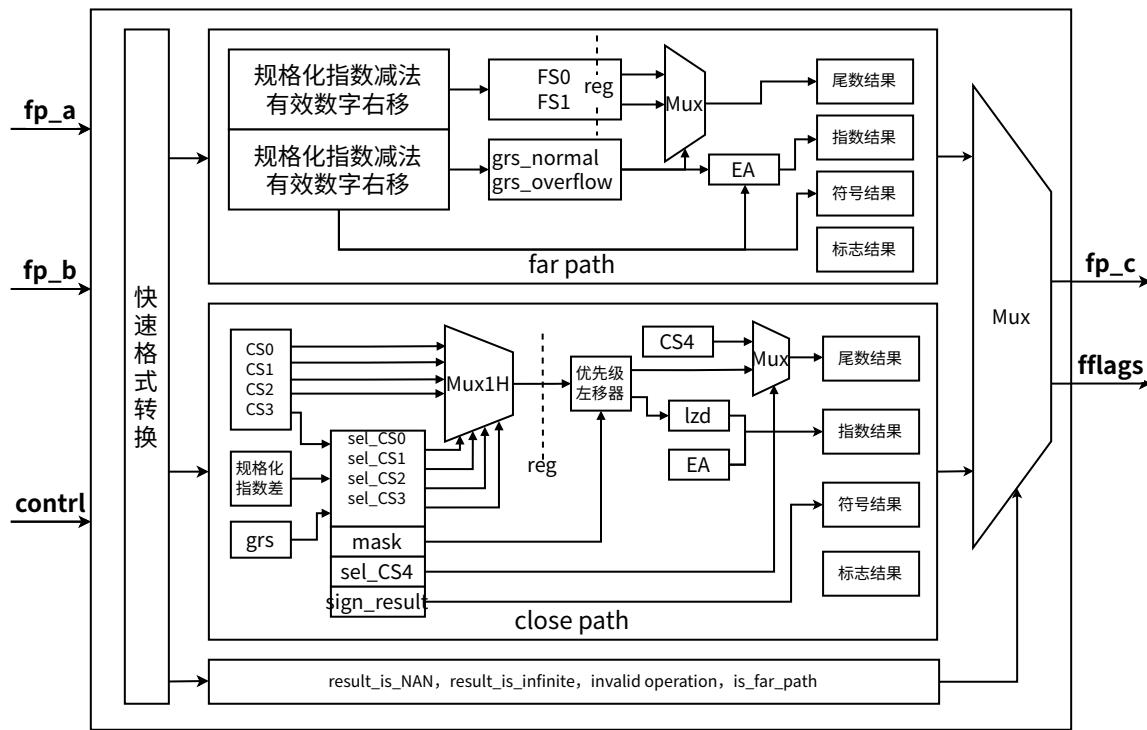
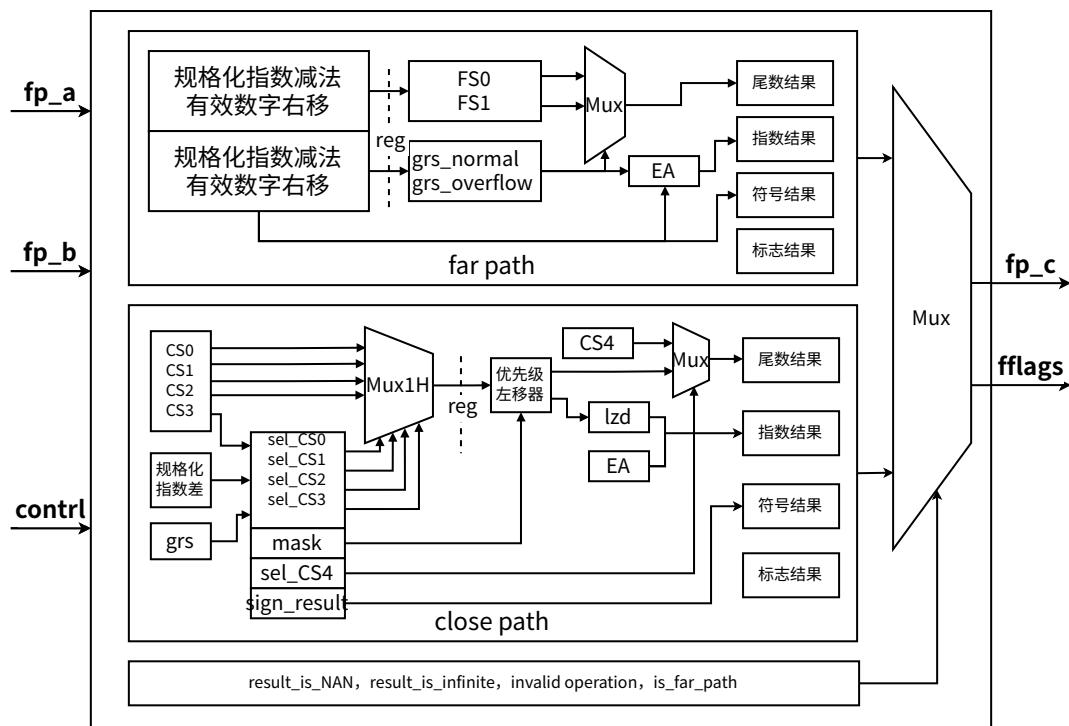
向量浮点加法器采用两周期的流水线设计，为了实现快速唤醒，使加法结果 1.5 周期左右计算好。流水线划分在每个子模块内部进行，只需插入一级寄存器，下面介绍图中三种模块的流水线划分。

下图是 *FloatAdderF64Widen* 模块的流水线划分，*far* 通路在有效数字右移之后插入寄存器，*close* 通路在 *Mux1H* 后插入寄存器。

下图是 *FloatAdderF32WidenF16* 模块的流水线划分，该模块包括两种不同输出格式的计算，第二周期的选择逻辑比较复杂，所以在 *far* 通路种寄存器插在加法器之中，第一周期进行低 18 位的加法和高位部分的加法，第二周期将第一周期的低 18 位加法的进位和高位部分相加得到最终结果，*close* 通路也是在 *Mux1H* 后插入寄存器。

下图是 *FloatAdderF16* 模块的流水线划分，该模块时序压力最小，采用 *far* 通路在有效数字右移之后插入寄存器，*close* 通路在 *Mux1H* 后插入寄存器的划分方式。



图 12.18: `FloatAdderF32WidenF16` 流水线划分图 12.19: `FloatAdderF16` 流水线划分

12.5.1.4 接口说明

之前介绍的向量浮点加法器宽度是 64 位的，输入数据要求两个操作数都是向量的形式，但是 *RVV* 不仅规定了两个操作数都是向量形式 (*vector – vector*, 缩写 *vv*)，还规定了一个操作数是向量另一个操作数是标量的形式 (*vector – scalar* 缩写 *vf*)，另外在 *widening* 指令下，源操作数的排列也不仅仅只有低有效的情况，当源寄存器宽度是目的寄存器宽度的一半时，数据来源可能是低半部分也可能是高半部分。

为了实现 *RVV* 中所有浮点指令的计算操作，并且可进行 *VLEN* 拓展，在向量浮点加法器的基础上增加其他简单指令的计算，把它作为一个向量浮点”*ALU*“，称作 *VFALU*。

因此需要对向量浮点加法器进行改造来适配 *RVV* 的特性。改造分为两部分，一部分是功能改造，一部分是接口改造。

下表是 *VFALU* 支持的操作码，共 16 种操作，(*w*) 表示含有 *widen* 操作。*vfmerge* *vmove* *vfclass* 操作数形式比较特殊：*vfmerge.vfm* 源操作数有三个：一个向量寄存器，一个浮点寄存器，一个 *mask* 寄存器；*vmove.v.f* 源操作数只有一个浮点寄存器；*vfclass* 源操作数只有一个向量寄存器。

表 12.21: *VFALU* 操作码

| <i>op_code</i> | 对应指令 | 操作数形式 | 含义 |
|----------------|-----------------|---------------|--------|
| 0 | <i>vf(w)add</i> | <i>vv, vf</i> | 加法 |
| 1 | <i>vf(w)sub</i> | <i>vv, vf</i> | 减法 |
| 2 | <i>vfmin</i> | <i>vv, vf</i> | 求最小值 |
| 3 | <i>vfmax</i> | <i>vv, vf</i> | 求最大值 |
| 4 | <i>vfmerge</i> | <i>vfm</i> | 数据合并 |
| 5 | <i>vmove</i> | <i>v.f</i> | 数据移动 |
| 6 | <i>vfsnj</i> | <i>vv, vf</i> | 符号注入 |
| 7 | <i>vfsnjin</i> | <i>vv, vf</i> | 反符号注入 |
| 8 | <i>vfsnjax</i> | <i>vv, vf</i> | 异或符号注入 |
| 9 | <i>vmfeq</i> | <i>vv, vf</i> | 是否相等 |
| 10 | <i>vmfnq</i> | <i>vv, vf</i> | 是否不等 |
| 11 | <i>vmflt</i> | <i>vv, vf</i> | 是否小于 |
| 12 | <i>vmfle</i> | <i>vv, vf</i> | 是否小于等于 |
| 13 | <i>vmfgt</i> | <i>vf</i> | 是否大于 |
| 14 | <i>vmfge</i> | <i>vf</i> | 是否大于等于 |
| 15 | <i>vfclass</i> | <i>v</i> | 分类 |

下表是 *VFALU* 接口定义，相比于向量浮点加法器，增加了 *widen_a*、*widen_b* 两个混合精度数据来源，当源操作数和目的操作数格式相同时，数据来自 *fp_a fp_b*，否则来自 *widen_a widen_b*，并且 *uop_idx = 0* 时取自低半部分，*uop_idx = 1* 时取自高半部分。*is_frs1 = 1* 时源操作数 *vs1* 来自浮点寄存器 *frs1*，需要复制成向量寄存器后参与计算，*mask* 参与 *merge* 指令的计算，*op_code* 是操作码，表示进行的操作。

表 12.22: *VFALU* 接口和含义

| 接口 | 方向 | 位宽 | 含义 |
|-------------|--------------|----|-----------------|
| <i>fp_a</i> | <i>input</i> | 64 | 源操作数 <i>vs2</i> |

| 接口 | 方向 | 位宽 | 含义 |
|---------------------|---------------|----|---------------------------|
| <i>fp_b</i> | <i>input</i> | 64 | 源操作数 <i>vs1</i> |
| <i>widen_a</i> | <i>input</i> | 64 | <i>widen_vs2</i> |
| <i>widen_b</i> | <i>input</i> | 64 | <i>widen_vs1</i> |
| <i>frs1</i> | <i>input</i> | 64 | 浮点寄存器数据 |
| <i>is_frs1</i> | <i>input</i> | 64 | 加数来自浮点寄存器数据 |
| <i>mask</i> | <i>input</i> | 4 | 参与 <i>merge</i> 指令计算 |
| <i>uop_idx</i> | <i>input</i> | 1 | <i>widen</i> 时选择高/低半部分 |
| <i>round_mode</i> | <i>input</i> | 3 | 舍入模式 |
| <i>fp_format</i> | <i>input</i> | 2 | 浮点格式 |
| <i>res_widening</i> | <i>input</i> | 1 | <i>widen</i> 指令 |
| <i>opb_widening</i> | <i>input</i> | 1 | 源操作数 <i>vs1</i> 是否和结果格式相同 |
| <i>op_code</i> | <i>input</i> | 5 | 操作码 |
| <i>fp_result</i> | <i>output</i> | 64 | 计算结果 |
| <i>fflags</i> | <i>output</i> | 20 | 标志位 |

12.5.2 向量浮点融合乘加器

12.5.2.1 流水线划分

向量浮点融合乘加器采用四周期的流水线设计，为了实现快速唤醒，使乘加结果在 3.5 周期左右计算好。向量器的延迟为 3.5 周期，下图是向量浮点融合乘加器架构图，图中 *reg_0* 表示第一级寄存器，*reg_1* 表示第二级寄存器，*reg_2* 表示第三级寄存器，向量浮点融合乘加器也支持 *widen* 功能，但是它的 *widen* 功能只有 $f32 = f16 \times f16 + f32$, $f64 = f32 \times f32 + f64$ 两种情况，所以输出格式确定的情况下，只需一比特 *widen* 信号控制即可。输出的 *fflags* 也是 20 比特的，和向量浮点加法器的表示方式一样。

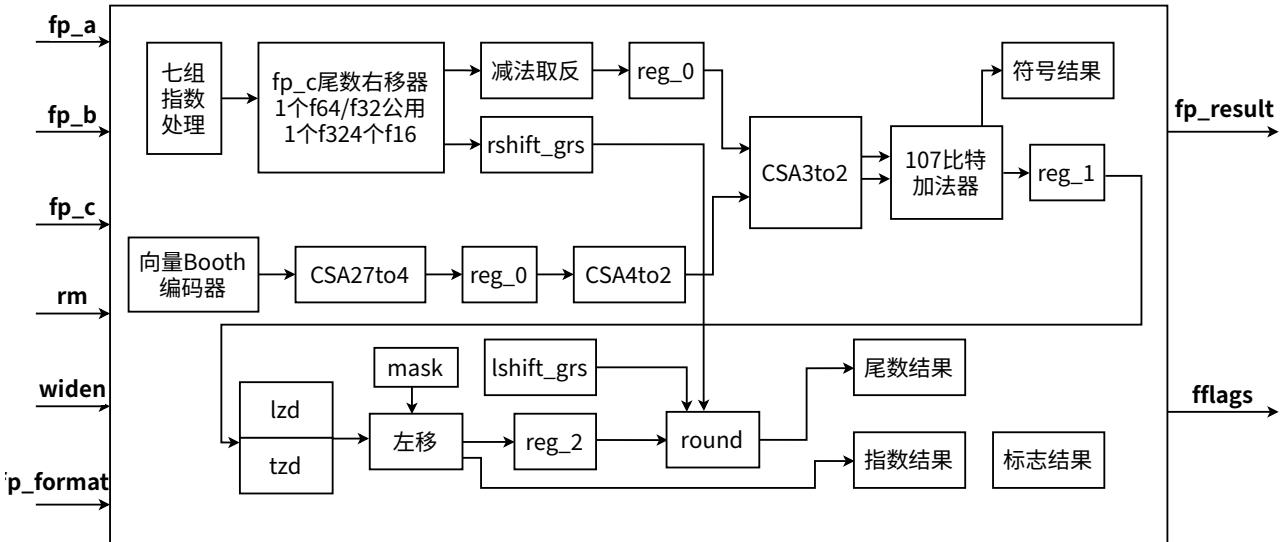


图 12.20: 向量浮点融合乘加器架构图

为了在满足时序约束的情况下节省面积，采用资源复用的实现方式，所有数据格式的计算通过同一个向量 *Booth* 编码器和 *CSA* 压缩，通过间隔摆放，107 比特加法器也实现了资源复用。

第一周期进行七组指数处理，得到七个右移值，根据计算格式选择对应的右移值，右移器方面将 $f64$ 的右移器与一个 $f32$ 共用，另外单独设置 1 个 $f32$ 、4 个 $f16$ 右移器，如果做减法 fp_c 尾数右移结果要取反后输入到第一级寄存器。第一周期同时进行向量 *Booth* 编码，生成 27 个部分积，使用 *CSA* 压缩成 4 个部分积后寄存。

第二周期对剩余 4 个部分积进行 *CSA4_2* 压缩后和第一周期计算的有效数字右移结果进行 *CSA3_2* 压缩，然后进行 107 比特加法，寄存到第二级寄存器。

第三周期对第二周期的求和结果做 *lzd* 和 *tzd*，然后进行带 *mask* 限位的左移，左移结果存到第三级寄存器。

第四周期进行舍入得到尾数结果，根据第三周期左移情况计算出指数结果，符号位可由第二周期的 107 比特加法器得到，标志结果可以产生上溢、下溢、无效操作、不精确四种标志位。注意下溢的检测方式，*IEEE-754* 规定了两种检测下溢的方式，*before rounding* 和 *after rounding*，本设计使用 *RISC-V* 选取的 *after rounding* 方式检测下溢。

12.5.2.2 接口说明

根据 *RVV* 的指令定义，可以使用向量浮点融合乘加器复用来计算乘法，由 *op_code* 控制，当计算乘法时内部将加数置零，同时，*RVV* 规定了一系列浮点融合乘加指令，主要是符号位和操作数顺序的区别。将向量浮点融合乘加器改造为支持所有相关指令的 *VFMA*，增加 *op_code* 和接口。下表是 *VFMA* 操作码，共 9 种操作，都支持 *vv* 和 *vf* 两种操作数形式，当 *vf* 时 *vs1[i]* 替换成浮点寄存器 *frs1*。

表 12.23: *VFMA* 操作码

| <i>op_code</i> | 对应指令 | 操作数形式 | 含义 |
|----------------|-------------------|---------------|---|
| 0 | <i>vf(w)mul</i> | <i>vv, vf</i> | $vd[i] = vs[2] \times vs1[i]$ |
| 1 | <i>vf(w)macc</i> | <i>vv, vf</i> | $vd[i] = +(vs1[i] \times vs2[i]) + vd[i]$ |
| 2 | <i>vf(w)nmacc</i> | <i>vv, vf</i> | $vd[i] = -(vs1[i] \times vs2[i]) - vd[i]$ |
| 3 | <i>vf(w)msac</i> | <i>vv, vf</i> | $vd[i] = +(vs1[i] \times vs2[i]) - vd[i]$ |
| 4 | <i>vf(w)nmsac</i> | <i>vv, vf</i> | $vd[i] = -(vs1[i] \times vs2[i]) + vd[i]$ |
| 5 | <i>vfmadd</i> | <i>vv, vf</i> | $vd[i] = +(vs1[i] \times vd[i]) + vs2[i]$ |
| 6 | <i>vfnamdd</i> | <i>vv, vf</i> | $vd[i] = -(vs1[i] \times vd[i]) - vs2[i]$ |
| 7 | <i>vfmsub</i> | <i>vv, vf</i> | $vd[i] = +(vs1[i] \times vd[i]) - vs2[i]$ |
| 8 | <i>vfnmsub</i> | <i>vv, vf</i> | $vd[i] = -(vs1[i] \times vd[i]) + vs2[i]$ |

下表是 *VFMA* 接口，为了简化控制逻辑的复杂度，送给 *VFMA* 三个操作数顺序固定为 *vs2* *vs1* *vd*，由功能单元内部根据 *op_code* 调换顺序，因为 *fma* 指令在 *widen* 时加数固定为目标格式，所以只需要增加 *widen_a* 和 *widen_b*，*uop_idx* 同样用来控制选 *widen_a* 和 *widen_b* 的高或低半部分，*frs1* 和 *is_frs1* 用来支持 *vf* 指令。

表 12.24: *VFMA* 接口和含义

| 接口 | 方向 | 位宽 | 含义 |
|-------------|--------------|----|-----------------|
| <i>fp_a</i> | <i>input</i> | 64 | 源操作数 <i>vs2</i> |
| <i>fp_b</i> | <i>input</i> | 64 | 源操作数 <i>vs1</i> |
| <i>fp_c</i> | <i>input</i> | 64 | 源操作数 <i>vd</i> |

| 接口 | 方向 | 位宽 | 含义 |
|--------------|--------|----|-----------------|
| widen_a | input | 64 | widen_vs2 |
| widen_b | input | 64 | widen_vs1 |
| frs1 | input | 64 | 浮点寄存器数据 |
| is_frs1 | input | 64 | 加数来自浮点寄存器数据 |
| uop_idx | input | 1 | widen 时选择高/低半部分 |
| round_mode | input | 3 | 舍入模式 |
| fp_format | input | 2 | 浮点格式 |
| res_widening | input | 1 | widen 指令 |
| op_code | input | 5 | 操作码 |
| fp_result | output | 64 | 计算结果 |
| fflags | output | 20 | 标志位 |

12.5.3 向量浮点除法器

12.5.3.1 标量浮点除法器

标量浮点除法器支持三种格式的计算:1个 $f16 = f16/f16$ 、1个 $f32 = f32/f32$ 、1个 $f64 = f64/f64$ 。除法器采用 $Radix - 64$ 算法, 迭代模块每周期进行三次 $Radix - 4$ 迭代来实现 $Radix - 64$, 下图是标量浮点除法器的架构图, 除法器是阻塞执行的, 计算过程中不能接受下一次除法操作, 因此需要握手信号来控制, 本设计采用 $start - valid$ 握手信号, 由于 CPU 中会出现分支预测失败要 $flush$ 掉流水线的状态, 所以专门设置了 $flush$ 信号用来清空除法器内部的状态, 使之下周期立刻可以开始进行新的除法计算。

对于输入数据分为三种情况: 都是规格化数 (不含除数是 2 的次幂)、至少有一个是非规格化数、提前终止 (输入含有 Nan , 无穷, 零, 或者除数是 2 的次幂)。结果分为两种情况: 结果是规格化数、结果是非规格化数。

当输入都是规格化数 (不含除数是 2 的次幂) 时, 尾数都是规格化的, 此时直接进入预缩放阶段; 当输入至少有一个是非规格化数时, 相比于输入都是规格化数, 在预缩放之前, 要再加一个周期进行尾数规格化。

预缩放阶段为一个周期, 接着进行整数商选择, 将两比特整数商结果选择出来, 并为 $Radix - 4$ 迭代提供预缩放后的除数被除数以及余数的保留进位冗余表示。 $Radix - 4$ 迭代模块每周期计算 6 比特商, $f16$ 除法需要 2 周期 $Radix - 4$ 迭代, $f32$ 除法需要 6 周期 $Radix - 4$ 迭代, $f64$ 除法需要 9 周期 $Radix - 4$ 迭代。 $Radix - 4$ 迭代完成后, 得到尾数商的结果的范围为 $(1, 2)$ 。当结果是规格化数时, 只需要一个周期进行舍入和指数结果的计算, 就可以得到最终的除法结果; 当结果是非规格化数时, 在舍入之前需要再加一个周期将商的结果进行非规格化。

提前终止分为两种情况: 当输入操作数含有 Nan , 无穷, 零的时候, 不需要进行除法计算, 第二个周期就可将结果输出; 当除数是 2 的次幂时, 第一个周期可求得指数结果, 如果结果不需要非规格化步骤, 第二周期可将结果输出; 如果结果需要非规格化步骤, 再加一个周期, 第三个周期将结果输出。

下表是不同数据格式情况下标量除法器所需计算周期, +1 表示除法结果是非规格化数时再加一个周期进行后处理。提前终止情况下, 只需 1~2 个周期完成所有数据格式除法操作, 在非提前终止情况下, $f16$ 除法需要 5~7 个周期完成除法, $f32$ 除法需要 7~9 个周期完成除法, $f64$ 除法需要 12~14 个周期完成除法。

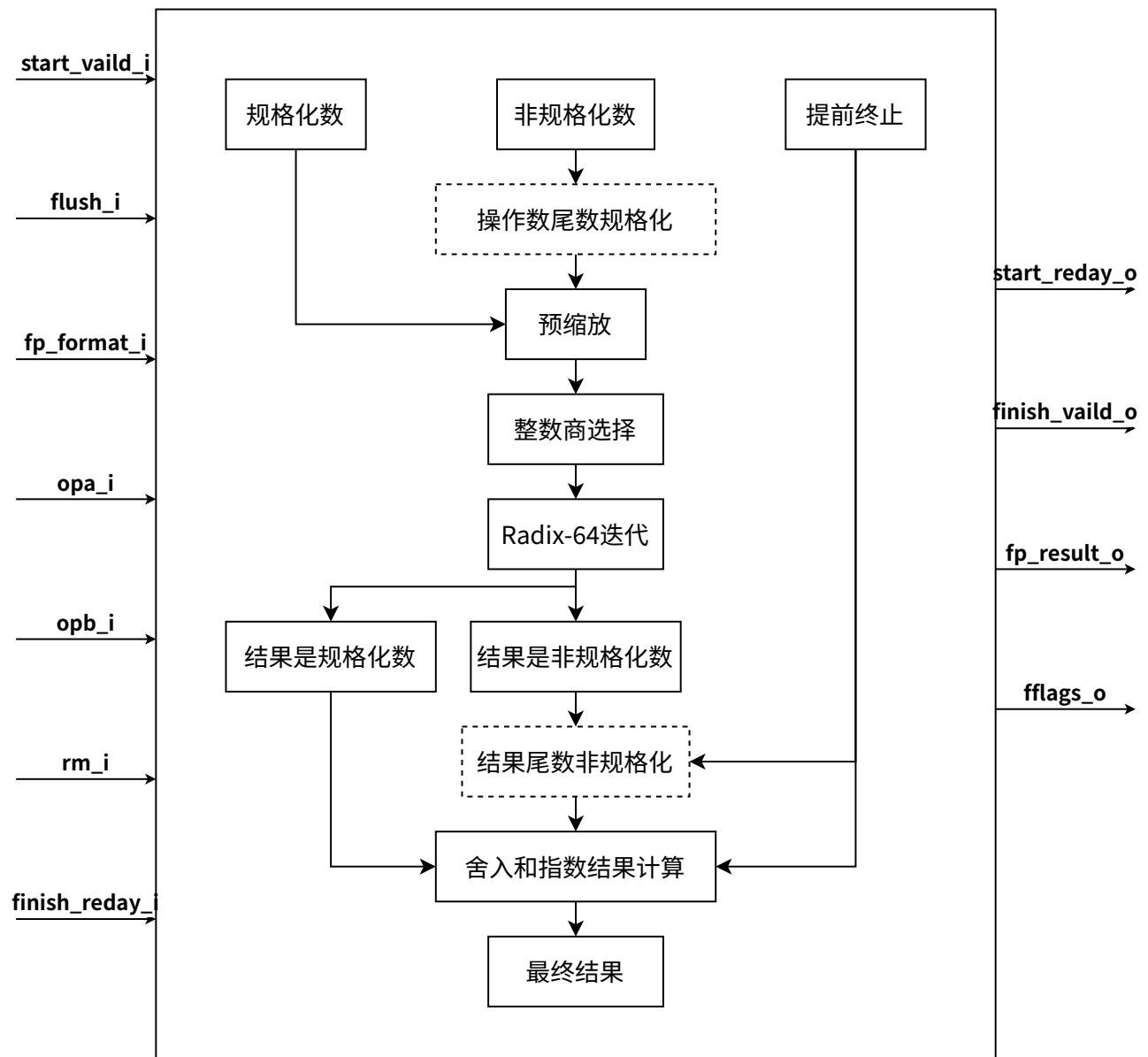


图 12.21: 标量浮点除法器架构图

表 12.25: 标量浮点除法器计算周期

| 数据格式 | 规格化数 | 非规格化数 | 提前终止 |
|------|--------|--------|-------|
| f16 | 5 + 1 | 6 + 1 | 1 + 1 |
| f32 | 7 + 1 | 8 + 1 | 1 + 1 |
| f64 | 12 + 1 | 13 + 1 | 1 + 1 |

12.5.3.2 向量浮点除法器

下图是向量浮点除法器架构图，相比与标量浮点除法器，考虑到向量除法计算同时计算多组除法，所有结果计算完才能需要统一写回到寄存器堆，因此单个除法的提前终止对向量除法加速意义不大，所有取消了输出结果延迟不确定的特性，任何情况下，根据输入的数据格式，向量浮点除法器的延迟数是确定的，如下表所示。

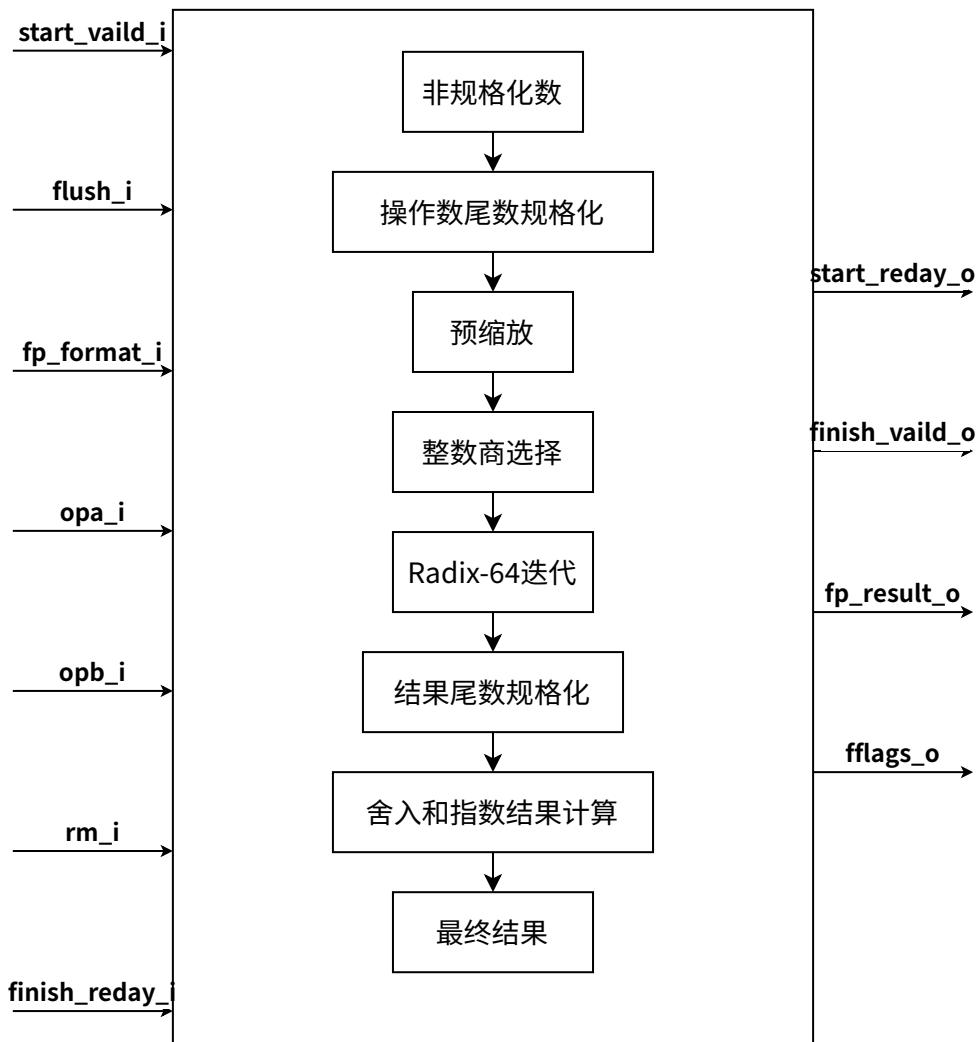


图 12.22: 向量浮点除法器架构图

表 12.26: 向量除法器计算周期

| 数据格式 | 计算周期 |
|-------|------|
| $f16$ | 7 |
| $f32$ | 11 |
| $f64$ | 14 |

在硬件设计上，除了 $Radix - 64$ 迭代模块，向量浮点除法器采用逻辑复用方式，使用四组信号进行计算和控制：第一组用于计算 $f64_0$ 或 $f32_0$ 或 $f16_0$ ，第二组用于计算 $f32_1$, $f16_1$ ，第三组用于计算 $f16_2$ ，第四组用于计算 $f16_3$ 。寄存器方面也采用复用的方式来存储中间结果，寄存器位宽按照 \max (1 个 $f64$, 2 个 $f32$, 4 个 $f16$)，满足最大要求。而 $Radix - 64$ 迭代模块是关键路径，也在满足时序要求下尽量节省面积，第一次 $Radix - 4$ 迭代使用 7 组独立的 CSA 和商选择，第二、三次 $Radix - 4$ 迭代使用 4 组复用的 CSA 和商选择。

12.5.3.3 接口说明

RVV 规定的向量浮点除法指令有三个：

- ① $vfdi.vv \quad vd[i] = vs2[i]/vs1[i]$
- ② $vfdi.vf \quad vd[i] = vs2[i]/f[rs1]$
- ③ $vfrdi.vf \quad vd[i] = f[rs1]/vs2[i]$

其中 ③ 比较特殊，它操作数的顺序和 ①② 不同，对于向量除法单元，第一个操作数由控制逻辑传 $vs2[i]/f[rs1]$ ，第二个操作数传 $vs1[i]/f[rs1]/vs2[i]$ ，所以功能单元看到的被除数是向量或标量形式，除数也是向量或标量形式，因此需要增加两个标量数据接口，增加接口后模块命名为 $VFDIV$ ，接口如下表所示。

表 12.27: $VFDIV$ 接口和含义

| 接口 | 方向 | 位宽 | 含义 |
|--------------------|----------|----|--------------|
| $start_valid_i$ | $input$ | 1 | 握手信号 |
| $finish_ready_i$ | $input$ | 1 | 握手信号 |
| $flush_i$ | $input$ | 1 | 冲刷信号 |
| fp_format_i | $input$ | 2 | 浮点格式 |
| opa_i | $input$ | 64 | 被除数 |
| opb_i | $input$ | 64 | 除数 |
| $frs2_i$ | $input$ | 64 | 被除数来自浮点寄存器数据 |
| $frs1_i$ | $input$ | 64 | 除数来自浮点寄存器数据 |
| is_frs2_i | $input$ | 1 | 被除数来自浮点寄存器 |
| is_frs1_i | $input$ | 1 | 除数来自浮点寄存器 |
| rm_i | $input$ | 3 | 舍入模式 |
| $start_ready_o$ | $output$ | 1 | 握手信号 |
| $finish_valid_o$ | $output$ | 1 | 握手信号 |
| $fdiv_res_o$ | $output$ | 64 | 计算结果 |
| $fflags_o$ | $output$ | 20 | 标志位 |

12.5.4 向量格式转换模块 *VCVT*

VCVT 模块是一个三级流水的向量浮点的格式转换模块。其例化 2 个具有 64bits 数据处理的子模块 *VectorCvt*, 每个 *VectorCvt* 分别包含一个 *cvt64*, 一个 *cvt32*, 两个 *cvt16* 模块。其中 *cvt64* 支持 64 32 16 8bits 浮点/整数格式的处理, *cvt32* 支持 32 16 8bits 浮点/整数格式的处理, *cvt16* 支持 16 8bits 浮点/整数格式的处理。即 *vectorCvt* 能同时处理 1 个 64bits (或者 2 个 32bits, 或者 4 个 16bits\$, 或者 4 个 8bits) 浮点/整数格式输入数据的转换。

12.5.4.1 总体设计

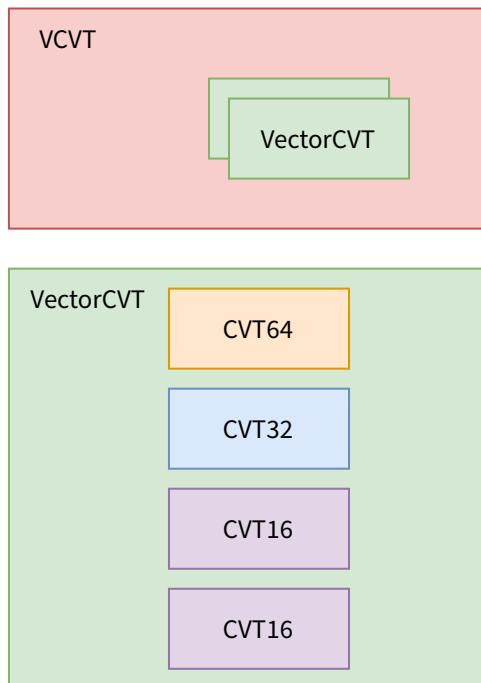


图 12.23: VCVT 总体设计

12.5.4.2 模块设计

CVT 模块包含单宽度浮点/整型类型转换指令、加宽浮点/整型类型转换指令、缩小浮点/整型类型转换指令、向量浮点倒数平方根估计指令及向量浮点倒数估计指令。

根据 *width* 选择不同的 *cvt* 模块调用, *cvt* 模块的设计思路为根据指令类型分为 4 种, *fp2int int2fp fp2fp vfr* 分别实现。*fcvt64* 的整体设计思路, 把输入的 64bit 数据进行格式统一:

*different width unsigned/signed int -> 65 signed int
f16/f32/f64 -> 65bit f64##false.B*

格式统一之后在转换过程中在一定程度上就不需要区分到底来的是哪种数据以及它们的位宽和字段的位置。

在此基础上, *VFCVT64* 分为 5 类: *int -> fp fp -> fpwidens fp -> fpnarrow estimate7(rsqrt7 & rec7) fp -> int.*

12.5.4.3 *FuopType* 译码逻辑

对于 *cvt* 指令来说: 其 *FuopType* 共 9 位, 每一位所表示的信息如下:

其中 [5 : 0] 是从手册里得到的, [8 : 6] 是为了方便生成控制信号设计时额外添加的。

[8] 1 表示其是 *move* 指令, 0 表示 *cvt* 指令或者 *vfrsqrt7* 和 *vfrec7* 这两条估计指令。

[7] 1 表示其输入是 *fp*, 0 表示输入是 *int*。

[6] 1 表示其输出是 *fp*, 0 表示输出是 *int*。

[5] 1 表示其是 *vfrsqrt7* 和 *vfrec7* 这两条估计指令, 否则是 *cvt* 指令; 当其为 1 时, [0] 用以区分其是 *vfrsqrt7* 还是 *vfrec7*。

[4 : 3] 00 表示 *single* 类型, 01 表示 *widen*, 10 表示 *narrow*。

[2 : 0]: 对于不同的指令有不同的作用: 对于浮点跟整数之间的转换, [0] 用以区分整数是有符号数还是无符号数; 其他情况下, [2 : 1] = 11 表示其是 *rtz* 类型指令, [2 : 0] = 101 表示是 *rod vfn cvt rod fw*。

13 CSR

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

13.1 术语说明

表 13.1: 术语说明

| 缩写 | 全称 | 描述 |
|-------|-------------------------------|---|
| CSR | Control and Status Register | 控制和状态寄存器 |
| Trap | Trap | 陷入, 中断和异常的合称 |
| ROB | Reorder Buffer | 重排序缓存 |
| PRVM | Privilege Mode | 特权级模式, 包括 M、S、U |
| VM/V | Virtual Mode | 虚拟化模式, 处于虚拟化模式具有 VS 和 VU 两个特权级 |
| EX_II | Illegal Instruction Exception | 非法指令异常 |
| EX_VI | Virtual Instruction Exception | 虚拟指令异常 |
| TVEC | Trap Vector | Trap 处理程序的入口配置寄存器, m/hs/vs 三个模式独立 |
| IMSIC | Incoming MSI Controller | 传入消息中断控制器, 定义在 The RISC-V Advanced Interrupt Architecture |

13.2 设计规格

- 支持执行 CSR 指令
- 支持执行 CSR 只读指令
- 支持 CSR 只读指令乱序
- 支持执行 mret、sret、ecall、ebreak、wfi 等系统级指令
- 支持接收中断, 取优先级最高的中断发送到 ROB 处理
- 支持产生 EX_II 和 EX_VI 两类异常

- 支持接收来自 ROB Trap (中断 + 异常) 并处理 Trap
- 支持符合 riscv-privileged-spec 规范的 CSR 实现
- 支持中断和异常代理
- 支持 Smaia 和 Ssaia 扩展
- 支持 Sdtrig 和 Sdext 扩展
- 支持 H 扩展
- 支持虚拟化中断
- 支持传入并处理外部中断

13.3 功能

CSR 作为一个功能单元 FU，与 fence 和 div 位于 intExuBlock 中的同一个 ExeUnit。CSR 内主要包含四个子模块，分别是 csrMod、trapInstMod、trapTvalMod 和 imsic。csrMod 是 CSR 的主要功能部件。

trapTvalMod 模块主要用于管理和更新与 trap 相关的目标值 tval。它根据输入信号 flush、targetPc 和 clear 等来更新或清除 tval，并确保在 clear 时 tval 是有效的。模块还包含一些状态逻辑，以确保在特定条件下正确更新 tval。这个模块需要从 csrMod 发出的 targetPc 以及来自于 flush 的 fullTarget 中选择来源，并且通过比较 robIdx 的先后来选择更新或是清除，最终输出 tval 信息。

trapInstMod 模块主要用于管理和更新 trap 的指令编码信息。它根据输入信号 (如 flush、faultCsrUop 和 readClear) 来更新或清除陷阱指令信息，并确保在特定条件下正确更新陷阱指令。模块还包含一些状态逻辑，以确保在特定条件下正确更新陷阱指令信息。这个模块需要从 decode 的指令信息 (包括指令编码、FtqPtr 和 FtqOffset)，以及来自 CSR 本身组合拼接出的 CSR 指令的指令信息中选择来源，并且通过比较 FtqPtr 和 FtqOffset 的先后来选择更新或是清除，以及更新的来源。在需要被 flush 或者 readClear 时设为无效。最终输出 trap 相关的指令编码，以及对应的 FtqPtr 和 FtqOffset。

imsic(Incoming MSI Controller)模块主要用于 csrMod 在通过 indirect alias CSR(mireg/sireg/vsireg) 访问 IMSIC 的内容时进行交互，输入 imsic 其必要的信息，如访问的 CSR 地址，所处于的特权级模式，写数据等，然后等到 imsic 的输出返回。如果 csrMod 本身的权限检查已经发现其应该产生异常，则不再向 imsc 发送请求。

CSR 负责执行 CSR 类别的指令和 mret、sret、ecall、ebreak、wfi 等系统类别的指令，接收来自 Backend 的指令 uop 和数据信息，在执行完成后将数据和跳转地址输出。若发生异常，则根据规则设置 EX_II 或 EX_VI。

CSR 负责接收来自外部中断控制器 CLINT 和 IMSIC 的 MSIP、MTIP、MEIP、SEIP、VSTIP、VSEIP 等中断 pending，根据当前特权级及其全局中断使能位决定是否响应，并对相应中断按优先级排序，选出优先级最高的中断交给 ROB 处理。

CSR 负责接收来自 ROB 的 Trap 信息，根据代理情况 (m[e|i]deleg 和 h[e|i]deleg) 将特权级模式 (PRVM) 和虚拟化模式 (V) 设置为处理 Trap 的特权级，修改相关 CSR 状态，并改变执行流为 TVEC 对应的 Trap Handler 的起始地址。

CSR 负责保存控制浮点和向量执行的配置信息 (Frm、Vstart、Vl、Vtype、Vxrm 等)，并存储浮点和向量指令执行产生的额外结果 (Fflags、Vxsat 等)。

CSR 负责和 IMSIC 通过自定义数据线交互，读写配置在 IMSIC 中的 mireg、sireg 和 vsireg 的部分寄存器 (external interrupts 部分)。

CSR 负责配置和更新 TLB 的相关信号，以确保 TLB 能够正确地进行虚拟地址到物理地址的转换。包括检测 ASID 和 VMID 的变化，satp/vsatp/hgatp 等寄存器值的传递，mstatus/vsstatus 中的 mxr/sum，menvcfg/henvcfg 中的 pmm 等权限和控制位的传递，虚拟内存模式的选择以及物理内存保护扩展的配置。通过这

些配置，TLB 能够在不同的虚拟内存模式下正确地执行地址转换。

CSR 负责根据当前的特权模式和寄存器的状态，设置和传递与指令 decode 相关的非法指令和虚拟指令的标志。这些标志用于指示某些指令在特定的特权模式下是否是非法的或虚拟的。通过这些标志，硬件可以在指令解码阶段正确地处理这些指令。

13.4 自定义 CSR

除去 RISC-V 手册定义的 CSR 外，我们还实现了 7 个自定义的 CSR：sbpctl、spfctl、slvpredctl、smblockctl、srnctl、mcorepwr 和 mflushpwr。

其中，sbpctl、spfctl、slvpredctl、smblockctl 和 srnctl 这 5 个自定义 CSR 定义在 HS 模式，mcorepwr 和 mflushpwr 这 2 个自定义 CSR 定义在 M 模式。

对这些自定义 CSR 的访问除了遵循特权级(低特权不能访问高特权)的约束外，还受到 Smstateen/Ssstateen 扩展中 C 字段对自定义内容访问的控制。

以下是各个自定义 CSR 的定义。

13.4.1 sbpctl

sbpctl (Speculative Branch Prediction Control register) 的地址是 0x5C0，是一个定义在 HS 模式的可读写寄存器。

表 13.2: sbpctl 的定义

| 字段名称 | 字段位置 | 初始值 | 描述 |
|-------------|--------|-----|-------------------------------|
| UBTB_ENABLE | 0 | 1 | UBTB_ENABLE 设 1 代表开启 uftb |
| BTB_ENABLE | 1 | 1 | BTB_ENABLE 设 1 代表开启主 ftb |
| BIM_ENABLE | 2 | 1 | BIM_ENABLE 设 1 代表开启 bim 预测器 |
| TAGE_ENABLE | 3 | 1 | TAGE_ENABLE 设 1 代表开启 TAGE 预测器 |
| SC_ENABLE | 4 | 1 | SC_ENABLE 设 1 代表开启 SC 预测器 |
| RAS_ENABLE | 5 | 1 | RAS_ENABLE 设 1 代表开启 RAS 预测器 |
| LOOP_ENABLE | 6 | 1 | LOOP_ENABLE 设 1 代表开启 loop 预测器 |
| | [63:7] | 0 | 保留 |

13.4.2 spfctl

spfctl (Speculative Prefetch Control register) 的地址是 0x5C1，是一个定义在 HS 模式的可读写寄存器。

表 13.3: spfctl 的定义

| 字段名称 | 字段位置 | 初始值 | 描述 |
|---------------|------|-----|------------------------|
| L1L_PF_ENABLE | 0 | 1 | 控制 L1 指令预取器，设 1 代表开启预取 |
| L2_PF_ENABLE | 1 | 1 | 控制 L2 预取器，设 1 代表开启预取 |
| L1D_PF_ENABLE | 2 | 1 | 控制 SMS 预取器，设 1 代表开启预取 |

| 字段名称 | 字段位置 | 初始值 | 描述 |
|-------------------------|---------|-----|--|
| L1D_PF_TRAIN_ON_HIT | 3 | 0 | 控制 SMS 预取器是否在 hit 时接受训练, 设 1 代表 hit 也会接受训练; 设 0 代表只有 miss 才会训练 |
| L1D_PF_ENABLE_AGT | 4 | 1 | 控制 SMS 预取器的 agt 表, 设 1 代表开启 agt 表 |
| L1D_PF_ENABLE_PHT | 5 | 1 | 控制 SMS 预取器的 pht 表, 设 1 代表开启 pht 表 |
| L1D_PF_ACTIVE_THRESHOLD | [9:6] | 12 | 控制 SMS 预取器的 active page 阈值 |
| L1D_PF_ACTIVE_STRIDE | [15:10] | 30 | 控制 SMS 预取器的 active page 跨度 |
| L1D_PF_ENABLE_STRIDE | 16 | 1 | 控制 SMS 预取器是否启用跨步 |
| L2_PF_STORE_ONLY | 17 | 0 | 控制 L2 预取器是否只对 store 预取 |
| L2_PF_RECV_ENABLE | 18 | 1 | 控制 L2 预取器是否接收 SMS 的预取请求 |
| L2_PF_PBOP_ENABLE | 19 | 1 | 控制 L2 预取器 PBOP 的启用 |
| L2_PF_VBOP_ENABLE | 20 | 1 | 控制 L2 预取器 VBOP 的启用 |
| L2_PF_TP_ENABLE | 21 | 1 | 控制 L2 预取器 TP 的启用 |
| | [63:22] | 0 | 保留 |

13.4.3 slvpredctl

slvpredctl (Speculative Load Violation Predictor Control register) 的地址是 0x5C2, 是一个定义在 HS 模式的可读写寄存器。

表 13.4: slvpredctl 的定义

| 字段名称 | 字段位置 | 初始值 | 描述 |
|-------------------------|--------|-----|---|
| LVPRED_DISABLE | 0 | 0 | 控制访存违例预测器是否禁用, 设 1 代表禁用 |
| NO_SPEC_LOAD | 1 | 0 | 控制访存违例预测器是否禁止 load 指令推测执行, 设 1 代表禁止 |
| STORESET_WAIT_STORE | 2 | 0 | 控制访存违例预测器是否会阻塞 store 指令, 设 1 代表会阻塞 |
| STORESET_NO_FAST_WAKEUP | 3 | 0 | 控制访存违例预测器是否支持快速唤醒, 设 1 代表不会快速唤醒 |
| LVPRED_TIMEOUT | [8:4] | 3 | 访存违例预测器的 reset 间隔, 设该位域的值为 x, 则间隔为 $2^{(10+x)}$ |
| | [63:9] | 0 | 保留 |

13.4.4 smblockctl

smblockctl (Speculative Memory Block Control register) 的地址是 0x5C3, 是一个定义在 HS 模式的可读写寄存器。

表 13.5: smblockctl 的定义

| 字段名称 | 字段位置 | 初始值 | 描述 |
|----------------------------------|---------|-----|---|
| SBUFFER_THRESHOLD | [3:0] | 7 | 控制 sbuffer 的 flush 阈值 |
| LDLD_VIO_CHECK_ENABLE | 4 | 1 | 控制是否开启 ld-lld 违例检查, 设 1 代表开启 |
| SOFT_PREFETCH_ENABLE | 5 | 1 | 控制是否开启 soft prefetch, 设 1 代表开启 |
| CACHE_ERROR_ENABLE | 6 | 1 | 控制是否上报 cache 发生的 ecc 错误, 设 1 代表开启 |
| UNCACHE_WRITE_OUTSTANDING_ENABLE | 7 | 0 | 控制是否支持 uncache 的 outstanding 访问, 设 1 代表开启 |
| HD_MISALIGN_ST_ENABLE | 8 | 1 | 控制是否启用硬件非对齐 store |
| HD_MISALIGN_LD_ENABLE | 9 | 1 | 控制是否启用硬件非对齐 load |
| | [63:10] | 0 | 保留 |

13.4.5 srnctl

srnctl (Speculative Runtime Control register) 的地址是 0x5C4, 是一个定义在 HS 模式的可读写寄存器。

表 13.6: srnctl 的定义

| 字段名称 | 字段位置 | 初始值 | 描述 |
|---------------|--------|-----|---------------------------|
| FUSION_ENABLE | 0 | 1 | fusion decoder 是否开启, 1 开启 |
| | 1 | 0 | 保留 |
| WFI_ENABLE | 2 | 1 | wfi 指令是否开启, 1 开启 |
| | [63:3] | 0 | 保留 |

13.4.6 mcorepwr

mcorepwr (Core Power Down Status Enable) 的地址是 0xBC0, 是一个定义在 M 模式的可读写寄存器。

表 13.7: mcorepwr 的定义

| 字段名称 | 字段位置 | 初始值 | 描述 |
|-------------------|--------|-----|---------------------------------------|
| POWER_DOWN_ENABLE | 0 | 0 | 1 表示当核心处于 WFI (等待中断) 状态时, 核心希望进入低功耗模式 |
| | [63:1] | 0 | 保留 |

13.4.7 mflushpwr

mflushpwr (Flush L2 Cache Enable) 的地址是 0xBC1, 是一个定义在 M 模式的可读写寄存器。

表 13.8: mflushpwr 的定义

| 字段名称 | 字段位置 | 初始值 | 描述 |
|-----------------|--------|-----|-----------------------------|
| FLUSH_L2_ENABLE | 0 | 0 | 1 表示核心希望刷新 L2 缓存并退出一致性状态 |
| L2_FLUSH_DONE | 1 | 0 | 只读位, 1 表示 L2 缓存刷新完成并退出一致性状态 |
| | [63:2] | 0 | 保留 |

13.5 CSR 异常检查

当前 CSR 中的权限检查模块 permitMod 将权限检查分为了多个子模块: xRetPermitMod、mLevelPermitMod、sLevelPermitMod、privilegePermitMod、virtualLevelPermitMod 和 indirectCSRPermitMod。permitMod 会产生 EX_II 和 EX_VI 两种异常。另外, xRetPermitMod 不同于其余子模块, 其对应执行 xret 指令时产生的异常, 而其余子模块则服务于 CSR 访问指令, 这两个部分是互斥的, 即不可能同时产生执行 xret 指令的异常和执行 CSR 访问指令的异常。

其中, xRetPermitMod 会生成执行 mnret/mret/sret/dret 指令时可能产生的异常: EX_II 和 EX_VI。

mLevelPermitMod 中只会产生 EX_II, 在其中会进行几种类型的权限检查: 写只读 CSR; 在 fs/vs 未开启时访问浮点/向量 CSR; 以及一系列由 M 模式 CSR (如 mstateen0 和 menvcfg) 控制的对其他低特权级 CSR 的访问。

sLevelPermitMod 中同样只会产生 EX_II, 在其中会进行一系列由 HS 模式 CSR (如 sstateen0 和 scounteren) 控制的对其他低特权级 CSR 的访问。

privilegePermitMod 中保证了低特权模式不能访问高特权模式的 CSR, 并根据当前所处的特权级和访问的目标 CSR 特权级来产生 EX_II 和 EX_VI 两种异常。

表 13.9: 不同特权级访问 CSR 权限检查

| | M-Level CSR | H/VS-Level CSR | S-Level CSR | U-Level CSR |
|---------|-------------|----------------|-------------|-------------|
| MODE_M | OK | OK | OK | OK |
| MODE_VS | EX_II | EX_VI | OK | OK |
| MODE_VU | EX_II | EX_VI | EX_VI | OK |
| MODE_HS | EX_II | OK | OK | OK |
| MODE_HU | EX_II | EX_II | EX_II | OK |

virtualLevelPermitMod 中会产生 EX_II 和 EX_VI 两种异常, 在其中会进行一系列由 H 模式 CSR (如 hstateen0 和 henvcfg) 控制的对其他 CSR 的访问。

indirectCSRPermitMod 中同样会产生 EX_II 和 EX_VI 两种异常, 在其中会进行一系列对 Alisa 的别名 CSR (mireg、sireg 和 vsireg) 访问的权限检查。

另外, 对于 CSR 访问时产生的异常, 我们优先选取 mLevelPermitMod、sLevelPermitMod、privilegePermitMod 和 virtualLevelPermitMod 的结果, 即直接访问产生的异常结果, 其次再考虑间接访问产生的异常结果 indirectCSRPermitMod。

在直接访问产生的异常结果中, 我们需要保证 mLevelPermitMod 的结果最优先, sLevelPermitMod 其次, 然后是 privilegePermitMod, 最后是 virtualLevelPermitMod。这一限制同时还保证了 EX_II 会优先于 EX_VI。

13.6 CSR 只读指令乱序

我们还支持 CSR 只读指令的乱序。我们注意到，对于绝大多数 CSR，CSRR 指令不需要等待前面的指令。对于所有 CSR，CSRR 指令也不需要阻塞后面的指令。需要注意的是，isCsrr 不仅仅包含 CSRR 指令的情况，还包含其他不需要写入 CSR 的 CSR 指令。

当前对于以下 CSR 执行的 CSRR 指令要求等待前面的指令，顺序执行：fflags, fcsr, vxsat, vcsr, vstart, mstatus, sstatus, hstatus, vsstatus, mnstatus, dcsr。因为这些 CSR 可能会被用户级指令在不需要 fence 的情况下就发生修改，如果乱序执行可能会导致错误的结果，所以对这些 CSR 执行 CSRR 指令要求顺序执行。

另外，由于在读取任何 PMC CSR 之前必须执行 fence 指令，因此没有必要让对 PMC CSR 的指令顺序执行。

CSR 指令过去是在没有流水的情况下执行的，因此 CSR 模块内部不需要一个状态机。在添加了允许对一些 CSR 只读指令的流水加速优化后，就需要一个状态机了，因为整数寄存器堆的仲裁器必须允许 CSRR 指令成功执行之前写入请求。

这个有限状态机有三个状态：空闲（s_idle）、等待 IMSIC（s_waitIMSIC）和完成（s_finish）。

在当前状态是 s_idle 时，如果有有效输入 valid 并且有 flush 信号时，则下一状态仍为 s_idle；如果有有效输入 valid 并且需要异步访问 AIA 时，则下一状态变为 s_waitIMSIC；如果有有效输入 valid，则下一状态变为 s_finish；其他情况下保持 s_idle。

在当前状态是 s_waitIMSIC 时，如果有 flush 信号，则下一状态恢复到 s_idle；如果收到从 AIA 回来的读有效信号，并且输出 ready，则下一状态恢复至 s_idle，否则如果输出没有 ready，则下一状态变为 s_finish，等待输出；其他情况下保持为 s_waitIMSIC。

在当前状态是 s_finish 时，如果有 flush 信号或者输出 ready 信号，则下一状态都将恢复至 s_idle；否则保持 s_finish。

14 HPM

- 版本: V2R2
- 状态: OK
- 日期: 2025/02/27
- commit: xxx

14.1 基本信息

14.1.1 术语说明

表 14.1: 术语说明

| 缩写 | 全称 | 描述 |
|-----|------------------------------|----------|
| HPM | Hardware performance monitor | 硬件性能计数单元 |

14.1.2 子模块列表

表 14.2: 子模块列表

| 子模块 | 描述 |
|--------------|-----------------|
| HPerfCounter | 单个计数器模块 |
| HPerfMonitor | 计数器组织模块 |
| PFEvent | Hpmevent 寄存器的副本 |

14.1.3 设计规格

- 基于 RISC-V 特权手册实现了基本的硬件性能监测功能，并额外支持 sstc 以及 sscofpmf 拓展
- 硬件线程执行的时钟周期数 (cycle)
- 硬件线程已提交的指令数 (minstret)
- 硬件定时器 (time)
- 计数器溢出标志 (time)
- 29 个硬件性能计数器 (hpmcounter3 - hpmcouonter3)
- 29 个硬件性能事件选择器 (mhpmcounter3 - mhpmcounter31)

- 支持最多定义 2^{10} 种性能事件

14.1.4 功能

HPM 的基本功能如下：

- 通过 mcountinhibit 寄存器关闭所有性能事件监测。
- 初始化各个监测单元性能事件计数器，包括：mcycle, minstret, mhpmccounter3 - mhpmccounter31。
- 配置各个监测单元性能事件选择器，包括：mhpmccounter3 - mhpmccounter31。香山昆明湖架构对每个事件选择器可以配置最多四种事件组合，将事件索引值、事件组合方法、采样特权级写入事件选择器后，即可在规定的采样特权级下对配置的事件正常计数，并根据组合后结果累加到事件计数器中。
- 配置 xcounteren 进行访问权限授权
- 通过 mcountinhibit 寄存器开启所有性能事件监测，开始计数。

14.1.4.1 HPM 事件溢出中断

昆明湖性能监测单元发起的溢出中断 LCOFIP，统一中断向量号为 12，中断的使能以及处理过程与普通私有中断一致

14.2 总体设计

在各个子模块中定义性能事件，子模块通过调用 generatePerfEvent 将性能事件组装为 io_perf 输出到四个主要模块：Frontend, Backend, MemBlock, CoupledL2。

上述四个模块通过调用 get_perf 方法获取子模块的性能事件输出，同时各个主要模块中例化 PFEEvent 模块，作为 CSR 中 mhpmevent 的副本，将所需要的性能事件选择器数据以及子模块的性能事件输出集合，接入 HPerfMonitor 模块，计算应用到性能事件计数器的增量结果。

最后，CSR 收集来自四个顶层模块的性能事件计数器的增量结果，分别输入到 CSR 寄存器 mhpmccounter3-31 中，进行累计计数。

特别的，CoupledL2 的性能事件会直接输入到 CSR 模块中，根据 mhpmevent 寄存器读出的事件选择信息，经过 CSR 中例化的 HPerfMonitor 模块处理，输入到 CSR 寄存器 mhpmccounter26-31 中累计计数。

具体 HPM 总体设计框图见图 14.1：

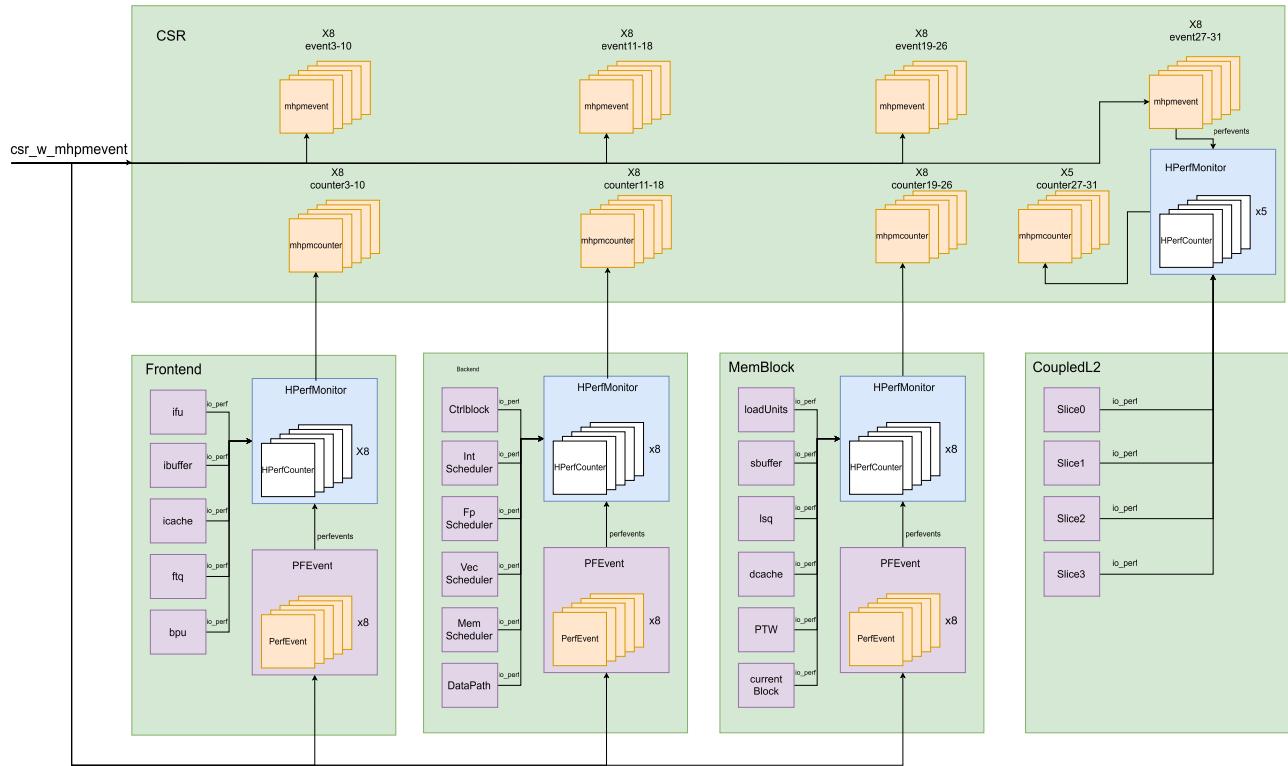


图 14.1: HPM 总体设计

14.2.1 HPerfMonitor 计数器组织模块

将输入的事件选择信息 (events) 输入对应的 HPerfCounter 模块，将所有的性能事件计数信息复制输入到每一个 HPerfCounter 模块。

收集所有的 HPerfCounter 输出。

14.2.2 HperfCounter 单个计数器模块

根据输入的事件选择信息，选择需要的性能事件计数信息，并根据事件选择信息中的计数模式，对输入的性能事件进行组合输出。

14.2.3 PFEVENT Hpmevent 寄存器的副本

CSR 寄存器 mhpmevent 的副本：收集 CSR 写信息，同步 mhpmevent 的变化

14.3 HPM 相关的控制寄存器

14.3.1 机器模式性能事件计数禁止寄存器 (MCOUNTINHIBIT)

机器模式性能事件计数禁止寄存器 (mcOUNTINHIBIT)，是 32 位 WARL 寄存器，主要用与控制硬件性能监测计数器是否计数。在不需要性能分析的场景下，可以关闭计数器，以降低处理器功耗。

表 14.3: 机器模式性能事件计数禁止寄存器说明

| 名称 | 位域 | 读写 | 行为 | 复位值 |
|------|------|------|--|-----|
| HPMx | 31:4 | RW | mhpmcOUNTERx 寄存器禁止计数位: 0: 正常计数 1: 禁止计数 | 0 |
| IR | 3 | RW | minstret 寄存器禁止计数位: 0: 正常计数 1: 禁止计数 | 0 |
| - | 2 | RO 0 | 保留位 | 0 |
| CY | 1 | RW | mcycle 寄存器禁止计数位: 0: 正常计数 1: 禁止计数 | 0 |

14.3.2 机器模式性能事件计数器访问授权寄存器 (MCOUNTEREN)

机器模式性能事件计数器访问授权寄存器 (mcOUNTEREN)，是 32 位 WARL 寄存器，主要用于控制用户态性能监测计数器在机器模式以下特权级模式 (HS-mode/VS-mode/HU-mode/VU-mode) 中的访问权限。

表 14.4: 机器模式性能事件计数器访问授权寄存器说明

| 名称 | 位域 | 读写 | 行为 | 复位值 |
|------|------|----|---|-----|
| HPMx | 31:4 | RW | hpmcounterenx 寄存器 M-mode 以下访问权限位: 0: 访问 hpmcounterx 报非法指令异常 1: 允许正常访问 hpmcounterx | 0 |
| IR | 3 | RW | instret 寄存器 M-mode 以下访问权限位: 0: 访问 instret 报非法指令异常 1: 允许正常访问 | 0 |
| TM | 2 | RW | time/stimecmp 寄存器 M-mode 以下访问权限位: 0: 访问 time 报非法指令异常 1: 允许正常访问 | 0 |
| CY | 1 | RW | cycle 寄存器 M-mode 以下访问权限位: 0: 访问 cycle 报非法指令异常 1: 允许正常访问 | 0 |

14.3.3 监督模式性能事件计数器访问授权寄存器 (SCOUNTEREN)

监督模式性能事件计数器访问授权寄存器 (scOUNTEREN)，是 32 位 WARL 寄存器，主要用于控制用户态性能监测计数器在用户模式 (HU-mode/VU-mode) 中的访问权限。

表 14.5: 监督模式性能事件计数器访问授权寄存器说明

| 名称 | 位域 | 读写 | 行为 | 复位值 |
|------|------|----|---|-----|
| HPMx | 31:4 | RW | hpmcounterenx 寄存器用户模式访问权限位: 0: 访问 hpmcounterx 报非法指令异常 1: 允许正常访问 hpmcounterx | 0 |
| IR | 3 | RW | instret 寄存器用户模式访问权限位: 0: 访问 instret 报非法指令异常 1: 允许正常访问 | 0 |
| TM | 2 | RW | time 寄存器用户模式访问权限位: 0: 访问 time 报非法指令异常 1: 允许正常访问 | 0 |
| CY | 1 | RW | cycle 寄存器用户模式访问权限位: 0: 访问 cycle 报非法指令异常 1: 允许正常访问 | 0 |

14.3.4 虚拟化模式性能事件计数器访问授权寄存器 (HCOUNTEREN)

虚拟化模式性能事件计数器访问授权寄存器 (hcouteren), 是 32 位 WARL 寄存器, 主要用于控制用户态性能监测计数器在客户虚拟机 (VS-mode/VU-mode) 中的访问权限。

表 14.6: 监督模式性能事件计数器访问授权寄存器说明

| 名称 | 位域 | 读写 | 行为 | 复位值 |
|------|------|----|--|-----|
| HPMx | 31:4 | RW | hpmcounterenx 寄存器客户虚拟机访问权限位: 0: 访问 hpmcounterx 报非法指令异常 1: 允许正常访问 hpmcounterx | 0 |
| IR | 3 | RW | instret 寄存器客户虚拟机访问权限位: 0: 访问 instret 报非法指令异常 1: 允许正常访问 | 0 |
| TM | 2 | RW | time/vstimecmp(via stimecmp) 寄存器客户虚拟机 访问权限位: 0: 访问 time 报非法指令异常 1: 允许正常访问 | 0 |
| CY | 1 | RW | cycle 寄存器客户虚拟机访问权限位: 0: 访问 cycle 报非法指令异常 1: 允许正常访问 | 0 |

14.3.5 监督模式时间比较寄存器 (STIMECMP)

监督模式时间比较寄存器 (stimecmp), 是 64 位 WARL 寄存器, 主要用于管理监督模式下的定时器中断 (STIP)。

STIMECMP 寄存器行为说明:

- 复位值为 64 位无符号数 64' hffff_ffff_ffff_ffff。
- 在 menvcfg.STCE 为 0 且当前特权级低于 M-mode (HS-mode/VS-mode/HU-mode/VU-mode) 时，访问 stimecmp 寄存器产生非法指令异常，且不产生 STIP 中断。
- stimecmp 寄存器是 STIP 中断产生源头：在进行无符号整数比较 $time \geq stimecmp$ 时，拉高 STIP 中断等待信号。
- 监督模式软件可以通过写 stimecmp 控制定时器中断的产生。

14.3.6 客户虚拟机监督模式时间比较寄存器 (VSTIMECMP)

客户虚拟机监督模式时间比较寄存器 (vstimecmp)，是 64 位 WARL 寄存器，主要用于管理客户虚拟机监督模式下的定时器中断 (STIP)。

VSTIMECMP 寄存器行为说明：

- 复位值为 64 位无符号数 64' hffff_ffff_ffff_ffff。
- 在 henvcfg.STCE 为 0 或者 hcounteren.TM 时，通过 stimecmp 寄存器访问 vstimecmp 寄存器产生虚拟非法指令异常，且不产生 VSTIP 中断。
- vstimecmp 寄存器是 VSTIP 中断产生源头：在进行无符号整数比较 $time + htimedelta \geq vstimecmp$ 时，拉高 VSTIP 中断等待信号。
- 客户虚拟机监督模式软件可以通过写 vstimecmp 控制 VS-mode 下定时器中断的产生。

14.4 HPM 相关的性能事件选择器

机器模式性能事件选择器 (mhpmevent3 - 31)，是 64 位 WARL 寄存器，用于选择每个性能事件计数器对应的性能事件。在香山昆明湖架构中，每个计数器可以配置最多四个性能事件进行组合计数。用户将事件索引值、事件组合方法、采样特权级写入指定事件选择器后，该事件选择器所匹配的事件计数器开始正常计数。

表 14.7: 机器模式性能事件选择器说明

| 名称 | 位域 | 读写 | 行为 | 复位值 |
|----------|-------|----|---|-----|
| OF | 63 | RW | 性能计数上溢标志位： 0: 对应性能计数器溢出时置 1，产生溢出中断 1: 对应性能计数器溢出时值不变，不产生溢出中断 | 0 |
| MINH | 62 | RW | 置 1 时，禁止 M 模式采样计数 | 0 |
| SINH | 61 | RW | 置 1 时，禁止 S 模式采样计数 | 0 |
| UINH | 60 | RW | 置 1 时，禁止 U 模式采样计数 | 0 |
| VSINH | 59 | RW | 置 1 时，禁止 VS 模式采样计数 | 0 |
| VUINH | 58 | RW | 置 1 时，禁止 VU 模式采样计数 | 0 |
| - | 57:55 | RW | - | 0 |
| OP_TYPE2 | 54:50 | RW | 计数器事件组合方法控制位： 5' b00000: 采用 or 操作组合 | 0 |
| OP_TYPE1 | 49:45 | | 5' b00001: 采用 and 操作组合 | |
| OP_TYPE0 | 44:40 | | 5' b00010: 采用 xor 操作组合 5' b00100: 采用 add 操作组合 | |

| 名称 | 位域 | 读写 | 行为 | 复位值 |
|--------|-------|----|------------------|-----|
| EVENT3 | 39:30 | RW | 计数器性能事件索引值: | - |
| EVENT2 | 29:20 | | 0: 对应的事件计数器不计数 | |
| EVENT1 | 19:10 | | 1: 对应的事件计数器对事件计数 | |
| EVENT0 | 9:0 | | | |

其中，计数器事件的组合方法为：

- EVENT0 和 EVENT1 事件计数采用 OP_TYPE0 操作组合为 RESULT0。
- EVENT2 和 EVENT3 事件计数采用 OP_TYPE1 操作组合为 RESULT1。
- RESULT0 和 RESULT1 组合记过采用 OP_TYPE2 操作组合为 RESULT2。
- RESULT2 累加到对应事件计数器。

对性能事件选择器中事件索引值部分复位值规定为 0

昆明湖架构将提供的性能事件根据来源分为四类，包括：前端，后端，访存，缓存，同时将计数器分为四部分，分别记录来自上述四个源头的性能事件：

- 前端：mhpmevent 3-10
- 后端：mhpmevent11-18
- 访存：mhpmevent19-26
- 缓存：mhpmevent27-31

表 14.8: 昆明湖前端性能事件索引表

| 索引 | 事件 |
|----|-------------------------|
| 0 | noEvent |
| 1 | frontendFlush |
| 2 | ifu_req |
| 3 | ifu_miss |
| 4 | ifu_req_cacheline_0 |
| 5 | ifu_req_cacheline_1 |
| 6 | ifu_req_cacheline_0_hit |
| 7 | ifu_req_cacheline_1_hit |
| 8 | only_0_hit |
| 9 | only_0_miss |
| 10 | hit_0_hit_1 |
| 11 | hit_0_miss_1 |
| 12 | miss_0_hit_1 |
| 13 | miss_0_miss_1 |
| 14 | IBuffer_Flushed |
| 15 | IBuffer_hungry |
| 16 | IBuffer_1_4_valid |
| 17 | IBuffer_2_4_valid |

| 索引 | 事件 |
|----|----------------------|
| 18 | IBuffer_3_4_valid |
| 19 | IBuffer_4_4_valid |
| 20 | IBuffer_full |
| 21 | Front_Bubble |
| 22 | Fetch_Latency_Bound |
| 23 | icache_miss_cnt |
| 24 | icache_miss_penalty |
| 25 | bpu_s2_redirect |
| 26 | bpu_s3_redirect |
| 27 | bpu_to_ftq_stall |
| 28 | mispredictRedirect |
| 29 | replayRedirect |
| 30 | predecodeRedirect |
| 31 | to_ifu_bubble |
| 32 | from_bpu_real_bubble |
| 33 | BpInstr |
| 34 | BpBInstr |
| 35 | BpRight |
| 36 | BpWrong |
| 37 | BpBRight |
| 38 | BpBWrong |
| 39 | BpJRight |
| 40 | BpJWrong |
| 41 | BpIRight |
| 42 | BpIWrong |
| 43 | BpCRight |
| 44 | BpCWrong |
| 45 | BpRRight |
| 46 | BpRWrong |
| 47 | ftb_false_hit |
| 48 | ftb_hit |
| 49 | fauftb_commit_hit |
| 50 | fauftb_commit_miss |
| 51 | tage_tht_hit |
| 52 | sc_update_on_mispred |
| 53 | sc_update_on_unconf |
| 54 | ftb_commit_hits |
| 55 | ftb_commit_misses |

表 14.9: 昆明湖后端性能事件索引表

| 索引 | 事件 |
|----|-----------------------------|
| 0 | noEvent |
| 1 | decoder_fused_instr |
| 2 | decoder_waitInstr |
| 3 | decoder_stall_cycle |
| 4 | decoder_utilization |
| 5 | INST_SPEC |
| 6 | RECOVERY_BUBBLE |
| 7 | rename_in |
| 8 | rename_waitinstr |
| 9 | rename_stall |
| 10 | rename_stall_cycle_walk |
| 11 | rename_stall_cycle_dispatch |
| 12 | rename_stall_cycle_int |
| 13 | rename_stall_cycle_fp |
| 14 | rename_stall_cycle_vec |
| 15 | rename_stall_cycle_v0 |
| 16 | rename_stall_cycle_vl |
| 17 | me_freelist_1_4_valid |
| 18 | me_freelist_2_4_valid |
| 19 | me_freelist_3_4_valid |
| 20 | me_freelist_4_4_valid |
| 21 | std_freelist_1_4_valid |
| 22 | std_freelist_2_4_valid |
| 23 | std_freelist_3_4_valid |
| 24 | std_freelist_4_4_valid |
| 25 | std_freelist_1_4_valid |
| 26 | std_freelist_2_4_valid |
| 27 | std_freelist_3_4_valid |
| 28 | std_freelist_4_4_valid |
| 29 | std_freelist_1_4_valid |
| 30 | std_freelist_2_4_valid |
| 31 | std_freelist_3_4_valid |
| 32 | std_freelist_4_4_valid |
| 33 | std_freelist_1_4_valid |
| 34 | std_freelist_2_4_valid |
| 35 | std_freelist_3_4_valid |
| 36 | std_freelist_4_4_valid |
| 37 | dispatch_in |
| 38 | dispatch_empty |

| 索引 | 事件 |
|----|--|
| 39 | dispatch_util |
| 40 | dispatch_waitinstr |
| 41 | dispatch_stall_cycle_lsq |
| 42 | dispatch_stall_cycle_rob |
| 43 | dispatch_stall_cycle_int_dq |
| 44 | dispatch_stall_cycle_fp_dq |
| 45 | dispatch_stall_cycle_ls_dq |
| 46 | rob_interrupt_num |
| 47 | rob_exception_num |
| 48 | rob_flush_pipe_num |
| 49 | rob_replay_inst_num |
| 50 | rob_commitUop |
| 51 | rob_commitInstr |
| 52 | rob_commitInstrFused |
| 53 | rob_commitInstrLoad |
| 54 | rob_commitInstrBranch |
| 55 | rob_commitInstrStore |
| 56 | rob_walkInstr |
| 57 | rob_walkCycle |
| 58 | rob_1_4_valid |
| 59 | rob_2_4_valid |
| 60 | rob_3_4_valid |
| 61 | rob_4_4_valid |
| 62 | BR_MIS_PRED |
| 63 | TOTAL_FLUSH |
| 64 | EXEC_STALL_CYCLE |
| 65 | MEMSTALL_STORE |
| 66 | MEMSTALL_L1MISS |
| 67 | MEMSTALL_L2MISS |
| 68 | MEMSTALL_L3MISS |
| 69 | issueQueue_enq_fire_cnt |
| 70 | IssueQueueAluMulBkuBrhJmp_full |
| 71 | IssueQueueAluMulBkuBrhJmp_full |
| 72 | IssueQueueAluBrhJmpI2fVsetriwiVsetriwvfl2v_full |
| 73 | IssueQueueAluCsrFenceDiv_full |
| 74 | issueQueue_enq_fire_cnt |
| 75 | IssueQueueFaluFcvtF2vFmacFdiv_full |
| 76 | IssueQueueFaluFmacFdiv_full |
| 77 | IssueQueueFaluFmac_full |
| 78 | issueQueue_enq_fire_cnt |
| 79 | IssueQueueVfmaVialuFixVimacVppuVfaluVfcvtVipuVsetrvfwvf_full |

| 索引 | 事件 |
|----|---------------------------------------|
| 80 | IssueQueueVfmaVialuFixVfalu_full |
| 81 | IssueQueueVfdivVidiv_full |
| 82 | issueQueue_enq_fire_cnt |
| 83 | IssueQueueStaMou_full |
| 84 | IssueQueueStaMou_full |
| 85 | IssueQueueLdu_full |
| 86 | IssueQueueLdu_full |
| 87 | IssueQueueLdu_full |
| 88 | IssueQueueVlduVstuVseglduVsegstu_full |
| 89 | IssueQueueVlduVstu_full |
| 90 | IssueQueueStdMoud_full |
| 91 | IssueQueueStdMoud_full |

表 14.10: 昆明湖访存性能事件索引表

| 索引 | 事件 |
|----|----------------------|
| 0 | noEvent |
| 1 | load_s0_in_fire |
| 2 | load_to_load_forward |
| 3 | stall_dcache |
| 4 | load_s1_in_fire |
| 5 | load_s1_tlb_miss |
| 6 | load_s2_in_fire |
| 7 | load_s2_dcache_miss |
| 8 | load_s0_in_fire |
| 9 | load_to_load_forward |
| 10 | stall_dcache |
| 11 | load_s1_in_fire |
| 12 | load_s1_tlb_miss |
| 13 | load_s2_in_fire |
| 14 | load_s2_dcache_miss |
| 15 | load_s0_in_fire |
| 16 | load_to_load_forward |
| 17 | stall_dcache |
| 18 | load_s1_in_fire |
| 19 | load_s1_tlb_miss |
| 20 | load_s2_in_fire |
| 21 | load_s2_dcache_miss |
| 22 | sbuffer_req_valid |
| 23 | sbuffer_req_fire |

| 索引 | 事件 |
|----|----------------------|
| 24 | sbuffer_merge |
| 25 | sbuffer_newline |
| 26 | dcache_req_valid |
| 27 | dcache_req_fire |
| 28 | sbuffer_idle |
| 29 | sbuffer_flush |
| 30 | sbuffer_replace |
| 31 | mpipe_resp_valid |
| 32 | replay_resp_valid |
| 33 | coh_timeout |
| 34 | sbuffer_1_4_valid |
| 35 | sbuffer_2_4_valid |
| 36 | sbuffer_3_4_valid |
| 37 | sbuffer_full_valid |
| 38 | MEMSTALL_ANY_LOAD |
| 39 | enq |
| 40 | ld_ld_violation |
| 41 | enq |
| 42 | stld_rollback |
| 43 | enq |
| 44 | deq |
| 45 | deq_block |
| 46 | replay_full |
| 47 | replay_rar_nack |
| 48 | replay_raw_nack |
| 49 | replay_nuke |
| 50 | replay_mem_amb |
| 51 | replay_tlb_miss |
| 52 | replay_bank_conflict |
| 53 | replay_dcache_replay |
| 54 | replay_forward_fail |
| 55 | replay_dcache_miss |
| 56 | full_mask_000 |
| 57 | full_mask_001 |
| 58 | full_mask_010 |
| 59 | full_mask_011 |
| 60 | full_mask_100 |
| 61 | full_mask_101 |
| 62 | full_mask_110 |
| 63 | full_mask_111 |
| 64 | nuke_rollback |

| 索引 | 事件 |
|-----|---------------------------|
| 65 | nack_rollback |
| 66 | mmioCycle |
| 67 | mmioCnt |
| 68 | mmio_wb_success |
| 69 | mmio_wb_blocked |
| 70 | stq_1_4_valid |
| 71 | stq_2_4_valid |
| 72 | stq_3_4_valid |
| 73 | stq_4_4_valid |
| 74 | dcache_wbq_req |
| 75 | dcache_wbq_1_4_valid |
| 76 | dcache_wbq_2_4_valid |
| 77 | dcache_wbq_3_4_valid |
| 78 | dcache_wbq_4_4_valid |
| 79 | dcache_mp_req |
| 80 | dcache_mp_total_penalty |
| 81 | dcache_missq_req |
| 82 | dcache_missq_1_4_valid |
| 83 | dcache_missq_2_4_valid |
| 84 | dcache_missq_3_4_valid |
| 85 | dcache_missq_4_4_valid |
| 86 | dcache_probq_req |
| 87 | dcache_probq_1_4_valid |
| 88 | dcache_probq_2_4_valid |
| 89 | dcache_probq_3_4_valid |
| 90 | dcache_probq_4_4_valid |
| 91 | load_req |
| 92 | load_replay |
| 93 | load_replay_for_data_nack |
| 94 | load_replay_for_no_mshr |
| 95 | load_replay_for_conflict |
| 96 | load_req |
| 97 | load_replay |
| 98 | load_replay_for_data_nack |
| 99 | load_replay_for_no_mshr |
| 100 | load_replay_for_conflict |
| 101 | load_req |
| 102 | load_replay |
| 103 | load_replay_for_data_nack |
| 104 | load_replay_for_no_mshr |
| 105 | load_replay_for_conflict |

| 索引 | 事件 |
|-----|----------------------|
| 106 | PTW_tlblptw_incount |
| 107 | PTW_tlblptw_inblock |
| 108 | PTW_tlblptw_memcount |
| 109 | PTW_tlblptw_memcycle |
| 110 | PTW_access |
| 111 | PTW_l2_hit |
| 112 | PTW_l1_hit |
| 113 | PTW_l0_hit |
| 114 | PTW_sp_hit |
| 115 | PTW_pte_hit |
| 116 | PTW_rwHarzad |
| 117 | PTW_out_blocked |
| 118 | PTW_fsm_count |
| 119 | PTW_fsm_busy |
| 120 | PTW_fsm_idle |
| 121 | PTW_resp_blocked |
| 122 | PTW_mem_count |
| 123 | PTW_mem_cycle |
| 124 | PTW_mem_blocked |
| 125 | ldDeqCount |
| 126 | stDeqCount |

表 14.11: 昆明湖缓存性能事件索引表

| 索引 | 事件 |
|----|---------------------------------|
| 0 | noEvent |
| 1 | Slice0_l2_cache_refill |
| 2 | Slice0_l2_cache_rd_refill |
| 3 | Slice0_l2_cache_wr_refill |
| 4 | Slice0_l2_cache_long_miss |
| 5 | Slice0_l2_cache_access |
| 6 | Slice0_l2_cache_l2wb |
| 7 | Slice0_l2_cache_l1wb |
| 8 | Slice0_l2_cache_wb_victim |
| 9 | Slice0_l2_cache_wb_cleaning_coh |
| 10 | Slice0_l2_cache_access_rd |
| 11 | Slice0_l2_cache_access_wr |
| 12 | Slice0_l2_cache_inv |
| 13 | Slice1_l2_cache_refill |
| 14 | Slice1_l2_cache_rd_refill |

| 索引 | 事件 |
|----|---------------------------------|
| 15 | Slice1_l2_cache_wr_refill |
| 16 | Slice1_l2_cache_long_miss |
| 17 | Slice1_l2_cache_access |
| 18 | Slice1_l2_cache_l2wb |
| 19 | Slice1_l2_cache_l1wb |
| 20 | Slice1_l2_cache_wb_victim |
| 21 | Slice1_l2_cache_wb_cleaning_coh |
| 22 | Slice1_l2_cache_access_rd |
| 23 | Slice1_l2_cache_access_wr |
| 24 | Slice1_l2_cache_inv |
| 25 | Slice2_l2_cache_refill |
| 26 | Slice2_l2_cache_rd_refill |
| 27 | Slice2_l2_cache_wr_refill |
| 28 | Slice2_l2_cache_long_miss |
| 29 | Slice2_l2_cache_access |
| 30 | Slice2_l2_cache_l2wb |
| 31 | Slice2_l2_cache_l1wb |
| 32 | Slice2_l2_cache_wb_victim |
| 33 | Slice2_l2_cache_wb_cleaning_coh |
| 34 | Slice2_l2_cache_access_rd |
| 35 | Slice2_l2_cache_access_wr |
| 36 | Slice2_l2_cache_inv |
| 37 | Slice3_l2_cache_refill |
| 38 | Slice3_l2_cache_rd_refill |
| 39 | Slice3_l2_cache_wr_refill |
| 40 | Slice3_l2_cache_long_miss |
| 41 | Slice3_l2_cache_access |
| 42 | Slice3_l2_cache_l2wb |
| 43 | Slice3_l2_cache_l1wb |
| 44 | Slice3_l2_cache_wb_victim |
| 45 | Slice3_l2_cache_wb_cleaning_coh |
| 46 | Slice3_l2_cache_access_rd |
| 47 | Slice3_l2_cache_access_wr |
| 48 | Slice3_l2_cache_inv |

14.4.1 Topdown PMU

Topdown 性能分析是一种自顶向下的分析方法，用来快速分析 CPU 的性能瓶颈，其核心思想是从高层次的性能分类逐步向下分解，逐层细化问题，最终精准定位根本原因。我们实现了三层 Topdown 性能事件，如下所示：

表 14.12: 三层 Topdown 性能事件

| Level 1 | Level 2 | Level 3 | 介绍 | 公式 |
|-------------|-------------|-------------|-----------|--|
| Retiring | . | . | 指令提交影响 | INST_RETIRIED / (IssueBW * CPU_CYCLES) |
| FrontEnd | . | . | 前端影响 | IF_FETCH_BUBBLE / (IssueBW * CPU_CYCLES) |
| Bound | Fetch | . | 取指延迟影响 | IF_FETCH_BUBBLE_EQ_MAX / CPU_CYCLES |
| | Latency | . | | |
| | Bound | . | | |
| | Fetch | . | 取指带宽影响 | FrontEnd Bound - Fetch Latency Bound |
| | Bandwidth | . | | |
| | Bound | . | | |
| Bad | . | . | 错误预测影响 | (INST_SPEC - INST_RETIRIED + RECOVERY_BUBBLE) / (IssueBW * CPU_CYCLES) |
| Speculation | Branch | . | 错误预测的分支影响 | Bad Speculation * BR_MIS_PRED / TOTAL_FLUSH |
| | Misspredict | . | 指令影响 | |
| | Machine | . | 机器清除事件 | Bad Speculation - Branch |
| | Clears | . | 影响 | Misspredict |
| BackEnd | . | . | 后端影响 | 1 - (FrontEnd Bound + Bad Speculation + Retiring) |
| Bound | Core Bound | . | 内核影响 | (EXEC_STALL_CYCLE - MEMSTALL_ANYLOAD - MEMSTALL_STORE) / CPU_CYCLE |
| | Memory | . | 访存影响 | (MEMSTALL_ANYLOAD + MEMSTALL_STORE) / CPU_CYCLES |
| | Bound | L1 Bound | L1 影响 | (MEMSTALL_ANYLOAD - MEMSTALL_L1MISS) / CPU_CYCLES |
| | | L2 Bound | L2 影响 | (MEMSTALL_L1MISS - MEMSTALL_L2MISS) / CPU_CYCLES |
| | | L3 Bound | L3 影响 | (MEMSTALL_L2MISS - MEMSTALL_L3MISS) / CPU_CYCLES |
| | | Mem Bound | 外部内存影响 | MEMSTALL_L3MISS / CPU_CYCLES |
| | | Store Bound | 存储指令影响 | MEMSTALL_STORE / CPU_CYCLES |

其中 IssueBW 为发射宽度，香山昆明湖架构当前是 6 发射。

表 14.13: Topdown 性能事件

| 名称 | 对应性能事件 | 介绍 |
|---------------|-----------------|---------------|
| CPU_CYCLES | - | 所有指令提交后总的时钟周期 |
| INST_RETIRIED | rob_commitInstr | 成功提交的指令个数 |

| 名称 | 对应性能事件 | 介绍 |
|------------------------|---------------------|---|
| INST_SPEC | - | 推测执行的指令个数 |
| IF_FETCH_BUBBLE | Front_Bubble | 从取指缓冲区获取空泡的个数, 且不存在后端停顿 |
| IF_FETCH_BUBBLE_EQ_MAX | Fetch_Latency_Bound | 从取指缓冲区获取 0 条指令的周期, 且不存在后端停顿 |
| BR_MIS_PRED | - | 错误预测的分支指令个数 |
| TOTAL_FLUSH | - | 流水线刷新事件的个数 |
| RECOVERY_BUBBLE | - | 从早期的错误预测中恢复的周期数 |
| EXEC_STALL_CYCLE | - | 发射 Few 个 uop 的周期数 |
| MEMSTALL_ANY_LOAD | - | 没有 uop 发射, 并且至少有一条 Load 指令没有完成 |
| MEMSTALL_STORE | - | 有非 Store 指令的 uop 发射, 并且有 Store 指令没有完成 |
| MEMSTALL_L1MISS | - | 没有 uop 发射, 至少有一条 Load 指令没有完成, 并且发生了 L1-cache Miss |
| MEMSTALL_L2MISS | - | 没有 uop 发射, 至少有一条 Load 指令没有完成, 并且发生了 L2-cache Miss |
| MEMSTALL_L3MISS | - | 没有 uop 发射, 至少有一条 Load 指令没有完成, 并且发生了 L3-cache Miss |

如果要统计一段时间里前端取指延迟的影响, 我们可以设置 mhpmevent3 的 EVENT0 域为 22, 其余位为默认值, 之后进行测试, 测试完成后可以通过 CSR 读指令读取 mhpmccounter3 寄存器, 获取这段时间里前端取指延迟的周期数, 然后通过计算可以得出前端取指延迟所造成的影响。

14.5 HPM 相关的性能事件计数器

香山昆明湖架构的性能事件计数器共分为两组, 分别是: 机器模式事件计数器、监督模式事件计数器、用户模式事件计数器

表 14.14: 机器模式事件计数器列表

| 名称 | 索引 | 读写 | 介绍 | 复位值 |
|-----------------|-------------|----|-------------|-----|
| MCYCLE | 0xB00 | RW | 机器模式时钟周期计数器 | - |
| MINSTRET | 0xB02 | RW | 机器模式退休指令计数器 | - |
| MHPMCOUNTER3-31 | 0XB03-0XB1F | RW | 机器模式性能事件计数器 | 0 |

其中 MHPMCOUNTERx 计数器相应由 MHPMEVENTx 控制, 指定计数相应的性能事件。

监督模式事件计数器包括监督模式计数器上溢中断标志寄存器 (SCOUNTOVF)

表 14.15: 监督模式计数器上溢中断标志寄存器 (SCOUNTOVF) 说明

| 名称 | 位域 | 读写 | 行为 | 复位值 |
|-------|------|------|--|-----|
| OFVEC | 31:3 | RO | mhpmcOUNTERx 寄存器上溢标志位: 1: 发生上溢 0: 没有发生上溢 | 0 |
| - | 2:0 | RO 0 | - | 0 |

scountovf 作为 mhpmcOUNTER 寄存器 OF 位的只读映射，受 xcounteren 控制：

- M-mode 访问 scountovf 可读正确值。
- HS-mode 访问 scountovf：mcounteren.HPMx 为 1 时，对应 OFVECx 可读正确值；否则只读 0。
- VS-mode 访问 scountovf：mcounteren.HPMx 以及 hcounteren.HPMx 均为 1 时，对应 OFVECx 可读正确值；否则只读 0。

表 14.16: 用户模式事件计数器列表

| 名称 | 索引 | 读写 | 介绍 | 复位值 |
|--------------------|-------------|----|-----------------------------|-----|
| CYCLE | 0xC00 | RO | mcycle 寄存器用户模式只读副本 | - |
| TIME | 0xC01 | RO | 内存映射寄存器 mtime 用户模式只读副本 | - |
| INSTRET | 0xC02 | RO | minstret 寄存器用户模式只读副本 | - |
| HPMCOUNTER3- 31 | 0XC03-0XC1F | RO | mhpmcOUNTER3-31 寄存器用户模式只读副本 | 0 |

15 Debug 模块

15.1 Debug Module

- 版本: V2R2
- 状态: OK
- 日期: 2025/01/20
- commit: xxx

15.1.1 术语说明

表 15.1: 术语说明

| 缩写 | 全称 | 描述 |
|-----|------------------------|--------|
| DM | Debug Module | 调试模块 |
| DTM | Debug Transport Module | 调试转换模块 |
| DMI | Debug Module Interface | 调试模块接口 |

15.1.2 参数设计

表 15.2: 参数设计

| 参数 | 默认值 | 描述 |
|----------------------|------------|-----------------------|
| baseAddress | 0x38020800 | debug Module MMIO 基地址 |
| nDMIAddrSize | 7 | DMI 地址宽度 |
| nProgramBufferWords | 16 | Program Buffer 数量 |
| nAbstractDataWords | 4 | Abstract Commands 数量 |
| hasBusMaster | true | system bus master |
| maxSupportedSBAccess | 64 | sysbus 最大访存宽度 |
| supportQuickAccess | false | QuickAccess 支持 |
| supportHartArray | true | hart array 支持 |
| nHaltGroups | 1 | halt group 数量 |
| nExtTriggers | 0 | external triggers 数量 |
| hasHartResets | true | reset 选中的 harts |

| 参数 | 默认值 | 描述 |
|-------------------|-------|--------------|
| hasImplicitEbreak | false | 隐式 ebreak 支持 |

15.1.3 总体设计

15.1.3.1 整体框图

如图 15.1 所示：

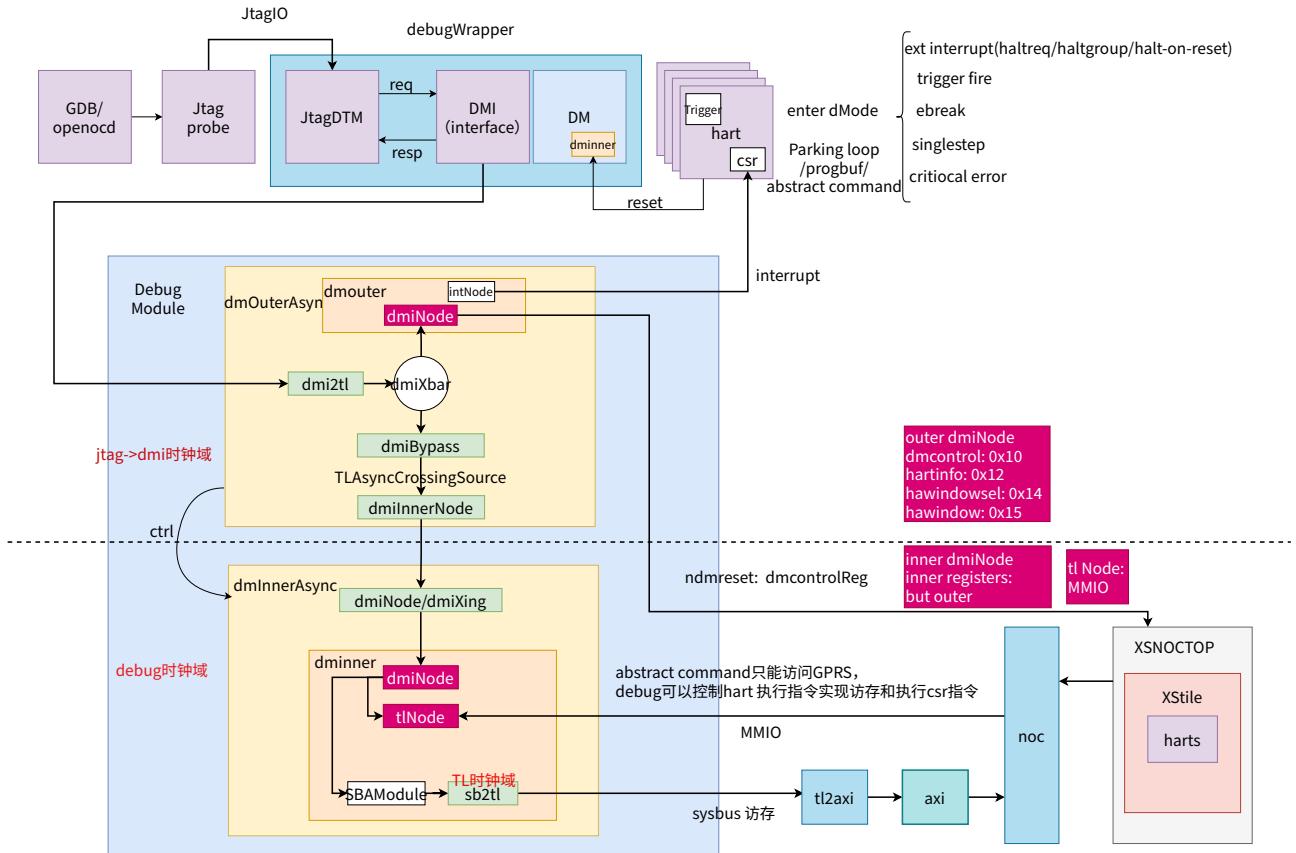


图 15.1: DebugModule 总览

15.1.3.2 多时钟域

如图 15.2 所示：

15.1.3.3 Debug MMIO

如表 15.3 所示：

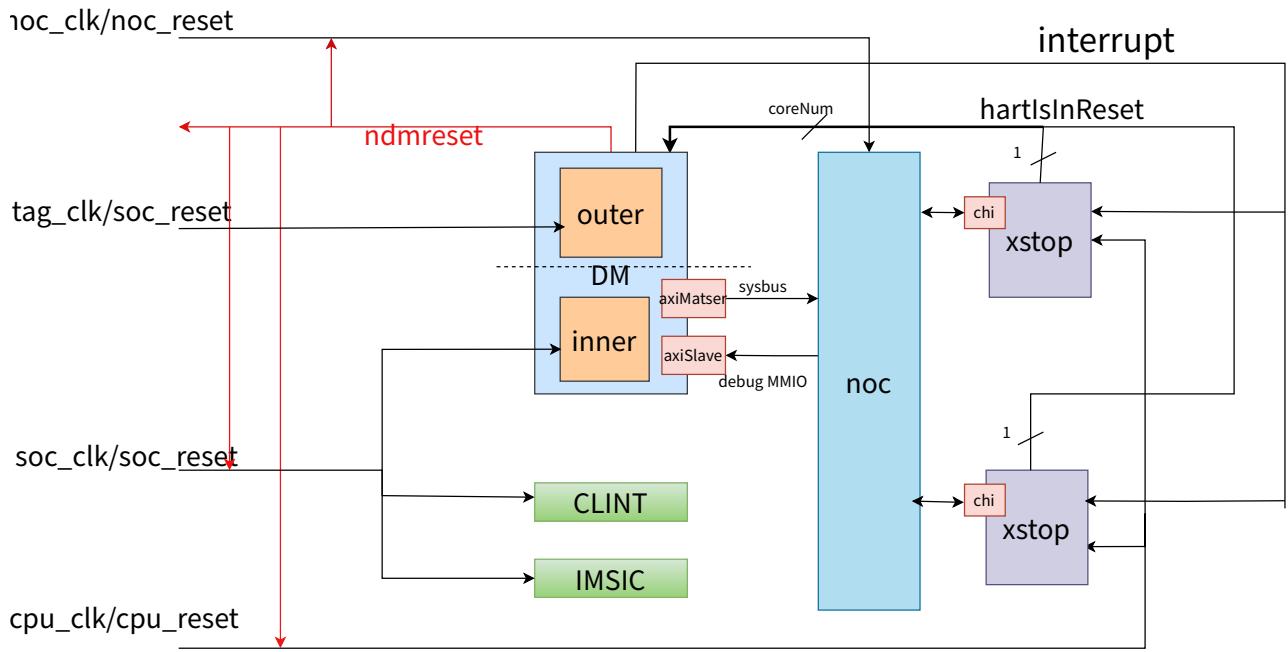


图 15.2: DebugModule 多时钟域

表 15.3: debug MMIO 地址空间

| 地址 (基地址) | 名称 | 描述 | 该地址里存放的内容 |
|-------------------------------------|----------------|----------------------------|---|
| 0x3802_0000 | debugEntry | debug 入口地址 /debug rom 的基地址 | |
| 0x800 | debugException | 在 dmode 执行时出现异常入口地址 | |
| 0x808 | HALTED | | 进入 dmode 的 hart 对应的 hartid, debugmodule 会拿到 whereto, 最终跳转到 ABSTRACT 去执行 |
| 0x100 | GOING | | |
| 0x104 | RESUMING | | 执行 dret |
| 0x10c | EXCEPTION | | |
| 0x300 | WHERETO | 该地址里存放的指令 | dm 生成的跳转到 ABSTRACT 的跳转指令 |
| 0x380 | DATA | DATA 的基地址 (for ld/st) | |
| DATA- 4*nProgBuf | PROGBUF | proobuf0 的地址 | 数据交换 |
| DATA-4 PROGBUF - 4*nAbstractInst | IMPEBREAK | 隐式 ebreak 指令 | dm 生成的指令 (go 之前准备好) |
| | ABSTRACT | AbstractInstructions | dm 生成的指令 (go 之前准备好) |

| 地址 (基地址) | | | |
|--------------|-------|---|--|
| 0x3802_0000) | 名称 | 描述 | 该地址里存放的内容 |
| 0x400 | FLAGS | hartid 对应 flag 的基地址, 每个 flag 是 8bits, 0x400 代表的是 hartid=0 时的 flag 的地址 | 这个 8bits 只有低两位有效, 次低位指的是 resume, 最低位指的 go, 地址空间是 1k 即 0x400->(0x500-0x1) |

15.1.4 模块设计

15.1.4.1 Debug Module

当前昆明湖 debug 实现的情况如下:

- 支持从第一条指令开始的调试, 在 cpu 复位之后进入调试模式。
- 支持单核、多核 (选中的核) 调试的运行控制, 包括 halt, resume, reset。
- 支持单步调试。
- 支持 stopcount, stoptime。
- 支持软断点 (ebreak 指令)、硬断点 (trigger) 和内存断点 (trigger)。
- 支持 GPR, CSR 和内存访问, 支持 protobuf 和 sysbus 两种访存方式。
- 支持通过 debug interrupt(haltreq, haltgroup, halt-on-reset), trigger fire, ebreak, singlestep, critical error 等方式进入 debug mode。

15.1.4.2 Trigger Module

当前昆明湖 trigger module 的实现情况如下:

- 昆明湖 trigger module 当前实现的 debug 相关的 CSR 如下表所示。
- trigger 的默认配置数量是 4 (支持用户自定义配置)。
- 支持 mcontrol6 类型的指令、访存的 trigger。
- match 支持相等, 大于等于, 小于三种类型 (向量访存当前只支持相等类型匹配)。
- 仅支持 address 匹配, 不支持 data 匹配。
- 仅支持 timing = before。
- 仅支持一对 trigger 的 chain。
- 为了防止 trigger 的二次产生 breakpoint 的异常, 支持通过 xSTATUS.xIE 控制。
- 支持 H 扩展的软硬件断点, watchpoint 调试手段。
- 支持原子指令的访存 trigger。

以下表格描述的是当前昆明湖支持的访存指令在微架构里的访存粒度和 trigger 匹配粒度: 对于标量指令和以元素为粒度进行访存的向量指令来说, match type 支持 \geq , $=$, $<$; 对于其余向量指令来说, 仅支持 match type 为 $=$ 的匹配。另外对于向量访存指令来说, 处理元素索引较小的指令触发的 trigger fire (不管其 trigger action 是 breakpoint 还是 debug)。

表 15.4: 访存粒度和 trigger 匹配粒度

| 指令类型 | 访存粒度 | trigger 匹配粒度 |
|-----------------------------|---------------------|---|
| 标量访存指令 | 指令 (元素) | 检查元素小端地址, 支持 $\geq, =, <$ |
| 原子访存指令 (lr/sc) | 指令 (元素) | 检查元素小端地址, 支持 $\geq, =, <; lr$ 视为 load, sc 视为 store (不管成功与否) |
| 原子访存指令 (amo) | 指令 (元素) | 检查元素小端地址, 支持 $\geq, =, <;$ 在拿到 vaddr 同时检查 load 和 store |
| 向量访存指令 (unit-stride) | 向量寄存器宽度 (128bit) | 支持该指令访存地址范围内任意地址的检查 (以 8bit 为粒度), 但仅仅支持 = 匹配 |
| 向量访存指令 (whole) | 向量寄存器宽度 (128bit) | 支持该指令访存地址范围内任意地址的检查 (以 8bit 为粒度), 但仅仅支持 = 匹配 |
| 向量访存指令 (fov unit-stride) | 向量寄存器宽度 (128bit) | 支持该指令访存地址范围内任意地址的检查 (以 8bit 为粒度), 但仅仅支持 0 号元素 = 匹配 |
| 向量访存指令 (segment) | 元素 | 检查每个元素小端地址, 但仅仅支持 = 匹配 |
| 其他类型的向量访存指令 | 元素 | 检查每个元素小端地址, 支持 $\geq, =, <$ |

表 15.5: 昆明湖实现的 debug 相关的 csr

| 名称 | 地址 | 读写 | 介绍 | 复位值 |
|-------------------|-------|----|--------------------------|--------------------|
| Tselect | 0x7A0 | RW | trigger 选择寄存器 | 0X0 |
| Tdata1(Mcontrol6) | 0x7A1 | RW | trigger data1 | 0xF000000000000000 |
| Tdata2 | 0x7A2 | RW | trigger data2 | 0x0 |
| Tinfo | 0x7A4 | RO | trigger info | 0x40 |
| Dcsr | 0x7B0 | RW | Debug Control and Status | 0x40000003 |
| Dpc | 0x7B1 | RW | Debug PC | 0x0 |
| Dscratch0 | 0x7B2 | RW | Debug Scratch Register 0 | - |
| Dscratch1 | 0x7B3 | RW | Debug Scratch Register 1 | - |
| mcontext | 0x7A8 | RW | Machine Context | - |
| hcontext | 0x6A8 | RW | Hypervisor Context | - |
| scontext | 0x5A8 | RW | Supervisor Context | - |

15.1.4.3 调试流程举例

15.1.4.3.1 CSR 访问:

debug module CSR 访问是 abstract command 和 progbuff 配合完成的, 根据 abstract command 会在 ABSTRACT 和 PROGBUFF 地址处分别生成相应的指令 (这两块地址空间是连续的), 让 CPU 去执行, 达到访问 CSR 的目的。ABSTRACT 处生成的是 lw/st 指令, 做的是 MMIO 地址与 GPR s0/s1 之间的数据交换, PROGBUFF 处生成的是 CSR 的读写指令)。下面以访问 mstatus 寄存器为例说明 debug module 是如何访问 CSR 的:

1. 假设软件发出一个写 mstatus CSR 的命令然后该命令会依次经过 JtagProbe, JtagDTM, DMI 转化为 DMI 操作;

2. Dmi 操作经过 dmi2tl 去修改 DebugModule 内部控制信号，修改 DMI_COMMAND 为写 mstatus 寄存器的 command；
3. openocd 首先会把 s0/fp 的值读出来存着，然后往 progbuffer 写 csr 的写指令；
4. 执行 ABSTRACT (ld 指令)，把 DATA 写到 s0；
5. 执行 progbuffer (CRS 写指令)，progbuff 以 ebreak 指令结尾，重新进入 parking loop；
如果是读 csr 的话：
6. 执行 progbuffer (CRS 读指令)，把 csr 的值读出来给 s0；
7. 执行 ABSTRACT (st 指令)，把 s0 写到 DATA；

15.1.4.3.2 硬件断点：

以下内容以打断点为例，说明在调试过程中，软硬件的协同工作流程：

1. 首先软件发出一个 halte d 的命令，然后该命令会依次经过 JtagProbe, JtagDTM, DMI 转化为 DMI 操作；
2. Dmi 操作经过 dmi2tl 去修改 DebugModule 内部控制信号，向 hart 发出一个外部的 debug 中断，该中断最终会传到 hart 内部的 CSR 模块中；
3. CSR 处理来自外部的 debug 中断：hart 将 Trap 到 debugModule 的入口地址去执行（见 Debug Module MMIO），进入 DMode；
4. Hart 进入到 DMode 之后，执行 debug ROM 里的指令，会把自己的 hartid 写到 HALTED（见第 8 节 Debug ROM），告知 Debug module 自己 (hart) 进入了 dmode，Debugger 在 Dmode 下可以对 hart 进行调试；
5. 当软件发出一个打硬件断点的命令时，hart 会跳到 whereto, abstract 和 progbuff 相互配合，控制 hart 执行 CSR 指令 (progbuffer) 配置 trigger CSR 寄存器，把断点的信息写入 trigger CSR；progbuff 以 ebreak 指令结尾，执行该 ebreak 会再次跳到 debugModule 的入口地址；
6. 软件发出一个 resume 的命令，hart 会跳到 _resume 去执行 dret 指令，退出 dMode，回到 1 处 halted 前去执行；(resume 之前有一个准备工作，需要先执行一次 step，只提交一条指令，然后通过 single step 异常 trap 到 debugMode，这块可以去看 openocd 的源码)
7. 当 hart 执行程序到打断点的位置时，指令的 pc 匹配上 trigger CSR 里配置的断点地址，trigger fire，hart 会再次进入 dmode (Trap 到 debugModule 的入口地址) 去执行 debug rom 里的指令，等待 debugger 调试。

第三部分

访存

16 访存流水线

16.1 访存流水线 LSU

16.1.1 子模块列表

| 子模块 | 描述 |
|---------------------|--------------|
| LoadUnit | Load 指令执行单元 |
| StoreUnit | Store 地址执行单元 |
| StdExeUnit | Store 数据执行单元 |
| AtomicsUnit | 原子指令执行单元 |
| VLSU | 向量访存 |
| LSQ | 访存队列 |
| Uncache | Uncache 处理单元 |
| SBuffer | Store 提交缓冲 |
| LoadMisalignBuffer | Load 非对齐缓冲 |
| StoreMisalignBuffer | Store 非对齐缓冲 |

16.1.2 设计规格

16.1.2.1 指令集规格

- 支持 RVI 指令集中 Load / Store 指令的执行与写回
- 支持 RVA 原子指令扩展
- 支持 RVH 虚拟化扩展
- 支持 RVV 向量扩展
- 支持 Cacheable 地址空间的 Load / Store / Atomic 访问
- 支持 MMIO 与 Uncache 地址空间的 Load / Store 访问（不包括向量访存指令和非对齐访存指令）
- 支持 Zicbom 和 Zicboz 等 cache 操作指令，支持 Zicbop 软件预取指令
- 支持非对齐访存 (Zicclsm)，且保证 16B 对齐范围内的非对齐访存的原子性 (Zama16b)
- 支持 Sv39 与 Sv48 分页机制
- 支持连续页地址翻译 (Svnapot)
- 支持基于页的内存属性 (Svpbmt)
- 支持 Pointer masking (Supm, Ssnpm, Sspm)
- 支持 Compare-and-Swap 原子指令 (Zacas)

- 支持 RVWMO 内存一致性模型
- 支持自定义故障注入指令

16.1.2.2 微结构特性

- 支持乱序调度 Load / Store 指令，包括 Cacheable 和 Uncache（非 MMIO）地址空间的访问
- 支持基于标量流水线的向量访存乱序调度
- 支持单元步长（Unit-stride）向量访存的元素合并访问
- 支持地址与数据分离的 Store 指令发射与执行
- 支持基于 LoadQueue 的 Load 指令重发机制
- 支持原子指令的非推测执行
- 支持 SBuffer 优化 Store 指令性能
- 支持基于 StoreQueue 与 SBuffer 的数据前递机制
- 支持 RAR / RAW 访存违例的检测与恢复
- 支持 MESI 缓存一致性协议
- 支持基于 TileLink 总线的多级 cache 访问
- 支持 DCache SECDED 校验
- 支持软件可配置的 Stream, Stride, SMS 等硬件预取器

16.1.2.3 参数配置

| 参数 | 配置 |
|-------------------|----------------------------|
| VAddr Bits | (Sv39) 39, (Sv48) 48 |
| GPAddr Bits | (Sv39x4) 41, (Sv48x4) 50 |
| LoadUnit | 3 x 8B/16B |
| StoreUnit | 2 x 8B/16B |
| StoreExeUnit | 2 |
| LoadQueue | 72 |
| LoadQueueRAR | 72 |
| LoadQueueRAW | 32 |
| LoadQueueReplay | 72 |
| LoadUncacheBuffer | 4 |
| StoreQueue | 56 |
| StoreBuffer | 16 x 64B |
| VLMergeBuffer | 16 |
| VSMergeBuffer | 16 |
| VSegmentBuffer | 8 |
| VFOFBuffer | 1 |
| Load TLB | 48-entry fully associative |
| Store TLB | 48-entry fully associative |
| L1 Prefetch TLB | 48-entry fully associative |
| L2 Prefetch TLB | 48-entry fully associative |

| 参数 | 配置 |
|----------------------|----------------------------|
| DCache | 64KB 4-way set associative |
| DCache MSHR | 16 |
| DCache Probe Queue | 8 |
| DCache Way Predictor | Off |

16.1.3 功能描述

访存流水线负责从发射队列接收访存指令（包括内存、MMIO 与 Uncache 地址空间的 Load / Store 指令，内存地址空间的原子指令），根据访存指令类型完成访存操作，得到指令执行结果，并将结果写回寄存器堆，同时通知前递旁路网络，唤醒后续指令并作数据前递。

16.1.3.1 访存指令的派遣

Load 与 Store 指令有复杂的控制机制，例如定序、前递、违例等，因此需要有一个队列来保存 load 与 store 指令，保证先进先出的顺序，进行相关的控制，这个队列就是 LoadQueue 和 StoreQueue。当指令完成译码和重命名等操作后，Load / Store 指令需要派遣到 ROB 和 LSQ 中，并分配相应的 robIdx、lqIdx 和 sqIdx，然后进入相应的发射队列，等待源操作数全部准备好后发射到 MemBlock 的流水线中。Load / Store 指令在 MemBlock 执行的整个生命周期都会携带 lqIdx 和 sqIdx，用于在访存违例检测、数据前递时进行指令序的定序。

对于标量访存指令，一条指令会分配一项 LoadQueue 或 StoreQueue 项。

对于向量访存指令，一条指令会在译码阶段被拆分成若干个 uop，每个 uop 包含若干个元素，每个元素等同于一次访存操作。在派遣阶段，一个 uop 会分配若干个 LSQ 项，分配项数等于一个 uop 所包含的元素个数。

16.1.3.2 访存指令的执行

访存单元包含 3 条 Load 流水线，2 条 Store 地址流水线，2 条 Store 数据流水线。每一条流水线会独立接收对应的发射队列发射出来的指令并执行。

Load 流水线为 4 级流水线结构：

- **s0:** 计算访存地址，完成不同来源（非对齐 Load，Load replay，MMIO，预取，标量 Load，向量 Load 等等）的请求仲裁，访问 TLB，访问 DCache 目录，发送写回唤醒信号
- **s1:** 接收 TLB 地址翻译的响应，接收 DCache 读目录的结果并做路选择，访问 DCache 数据 SRAM；和 StoreUnit s1 的 store 指令进行 RAW 违例检测；查询 StoreQueue / LoadQueueUncache / SBuffer / DCache MSHR 进行数据前递
- **s2:** 查询 LoadQueueRAR 和 LoadQueueRAW 供后续 Load / Store 指令做违例检查；如果 DCache miss 需要在 s2 分配 MSHR；和 StoreUnit s1 的 store 指令进行 RAW 违例检
- **s3:** 写回；如果没有写回的话需要取消唤醒；如果发生访存违例则刷新流水线；如果需要重发则进入 Load-QueueReplay

Store 地址流水线为 4 级流水线结构：

- **s0:** 计算访存地址，完成不同来源（非对齐 Store，标量 Store，向量 Store 等）的请求仲裁，访问 TLB
- **s1:** 接收 TLB 地址翻译的响应；和 LoadUnit s1 和 s2 的 load 指令进行 RAW 违例检测；查询 Load-QueueRAW 进行违例检查

- s2: 在 StoreQueue 中标记为地址已准备好
- s3: 写回

Store 数据流水线从发射队列接收到数据后将数据写回到 StoreQueue 并标记为数据已准备好。

16.1.3.3 向量访存指令的执行

对于除 Segment 以外的向量访存指令，VLSplit 和 VSSplit 接收从向量访存发射队列发射的 uop，将 uop 拆分成若干元素，VLSplit 和 VSSplit 会将这些元素发射到 LoadUnit / StoreUnit 上执行，执行流程和标量访存相同。元素执行完成后会写回到 VLMerge / VSMerge，Merge 模块负责收集元素、组合成 uop 并写回向量寄存器堆。

Segment 指令由独立的 VSegmentUnit 模块处理。

16.1.3.4 Load 指令重发

Load 指令不支持在发射队列中重发，因此当 Load 指令发生如下特殊情况时需要进入 LoadQueueReplay 等待重新执行：

- C_MA: 访存违例预测算法（MDP）认为 Load 与某条更老的 Store 有地址依赖，且该 Store 地址还没有准备好
- C_TM: TLB 缺失
- C_FF: Load 与某条更老的 Store 存在地址依赖，但是这条 Store 数据还没有准备好
- C_DR: DCache 缺失且 MSHR 满，或者存在同地址 MSHR 暂无法接收新的 Load
- C_DM: DCache 缺失，当前 Load 成功被 MSHR 接收
- C_WF: 路预测器预测失败（路预测器默认关）
- C_BC: 访问 DCache 发生 bank 冲突
- C_RAR: LoadQueueRAR 满
- C_RAW: LoadQueueRAW 满
- C_NK: 与 StoreUnit 的 Store 指令发生了访存违例
- C_MF: LoadMisalignBuffer 满

LoadQueueReplay 会根据重发的原因按如上的优先级从高到低进行重发。

16.1.3.5 Store 指令重发

Store 指令由发射队列负责重发。Store 指令从发射队列中发射后，发射队列不会立即清空这条 Store 指令，而是等待 StoreUnit 的反馈，StoreUnit 会根据 TLB 是否命中给发射队列发送相应的反馈。如果 TLB 缺失，发射队列会负责指令重发。

16.1.3.6 RAR 访存违例的检测与恢复

RAR 访存违例：根据 RVWMO 模型，当 (1) 两个相同地址（包括存在地址重叠的情况）的读操作中间插入同地址的写操作，且 (2) 这两次读操作返回的结果来自于不同的写操作时，这两次读操作需要保持和程序序统一。在单核场景下，访存单元虽然会乱序执行 Load 指令，但是会通过数据前递机制保证两条同地址的 Load 的执行结果一定会保证程序序；但在多核场景下，两条被打乱顺序的同地址 Load 中间插入另一个核的写操作（注意是写操作不是写指令）时，更老的 Load 会读到写操作后的更新的值，更年轻的 Load 会读到写操作前的旧值，即 RAR 访存违例。

RAR 访存违例的检测: LoadQueue 中的 LoadQueueRAR 模块会用 FreeList 的结构记录所有有可能存在相同地址但还没有执行的更老的 Load 的 Load 指令。Load 指令在 LoadUnit 执行到 s2 级（此时已完成地址翻译和 PMA / PMP 检查）时，会分配 LoadQueueRAR 项。当 LoadQueueRAR 中的 Load 指令在程序序上更老的 Load 都已全部写回时，这条 Load 指令即可从 LoadQueueRAR 中释放。当 Load 指令在访问 LoadQueueRAR 时发现有更年轻的、相同地址的 Load，且更年轻的 Load 可能被另一个核访问过（该地址发生过替换，或者被 Probe 过），则发生 RAR 访存违例，需要进行回滚。

RAR 访存违例的恢复: 当检测到 RAR 违例发生时，LoadUnit 会发起回滚，从发生违例的更老的 Load 的下一条指令开始刷新流水线。

16.1.3.7 RAW 访存违例的检测与恢复

RAW 访存违例: 处理器核执行 Load 指令的结果应该来自于当前处理器核所见的全局内存序的最近一次写操作，特别地，如果最近一次写操作来自当前核的 Store 指令，那么 Load 应该拿到这条 Store 所写的数据。超标量乱序处理器为了优化 Load 指令的性能会推测执行 Load，因此一条 Load 指令可能会越过一条更老的、相同地址的 Store 先执行，拿到 Store 之前的旧值，即 RAW 访存违例。

RAW 访存违例的检测: LoadQueue 中的 LoadQueueRAW 模块会用 FreeList 的结构记录所有有可能存在相同地址但还没有执行的更老的 Store 的 Load 指令。Load 指令在 LoadUnit 执行到 s2 级（此时已完成地址翻译和 PMA / PMP 检查）时，会分配 LoadQueueRAW 项。当 StoreQueue 中所有 Store 地址都已就绪，LoadQueueRAW 中所有的 Load 都可以被释放；或者当 LoadQueueRAW 中的指令在程序序上更老的 Store 全部地址准备就绪时，这条 Load 即可从 LoadQueueRAW 中释放。当 Store 指令在查询 LoadQueueRAW 时发现有更年轻的、相同地址的 Load，则发生 RAW 访存违例，需要进行回滚。

RAW 访存违例的恢复: 当检测到 RAW 违例发生时，LoadQueueRAW 会发起回滚，从发生违例的 Store 的下一条指令开始刷新流水线。

16.1.3.8 SBuffer 优化 Store 指令性能

根据 RVWMO 模型，在多核场景下，（没有 FENCE 或其他带有屏障语义的指令前提下）一个核的 Store 指令可以晚于更年轻的不同地址 Load 指令对其他核可见。该内存模型规则主要是为了优化 Store 指令的性能，包括 RVWMO 在内的弱一致性模型允许在处理器核中加入 SBuffer，用于暂存已提交的 Store 指令的写操作，对这些写操作做合并后再写入 DCache，减少 Store 指令对 DCache SRAM 端口的争用，从而提高 Load 指令的执行带宽。

SBuffer 为 $16 \times 512B$ 的全相联结构。当多个 Store 地址落在同一 cache 块时，SBuffer 会对这些 Store 进行合并。

SBuffer 每周期最多写入 2 条 Store 指令，每条 Store 指令的写数据宽度为 16B（特殊地，cbo.zero 指令一次操作一个 cache 块）。

SBuffer 的换出:

- 当 SBuffer 的容量超过一定阈值时会执行换出操作，根据 PLRU 替换算法选出替换块写入 DCache
- SBuffer 支持被动刷新机制，FENCE / 原子 / 向量 Segment 等指令执行时会清空 SBuffer
- SBuffer 支持超时刷新机制，数据块超过 2^{20} 拍没有被替换会被换出

16.1.3.9 Store-to-Load 数据前递

SBuffer 的存在以及 Load 指令的推测执行，导致 Load 指令除了要访问 DCache 还需要访问 SBuffer 和 StoreQueue，因此要求 SBuffer 和 StoreQueue 提供 Store-to-Load 数据前递功能。当多个来源同时命中时，

LoadUnit 需要对多个来源的数据做合并，合并的优先级 StoreQueue > SBuffer > DCache。

16.1.3.10 MMIO 指令的执行

香山核只允许标量访存指令访问 MMIO 地址空间。MMIO 访问和其他任何访存操作都是强定序的 (Strongly-ordered)，因此 MMIO 指令需要等待其成为 RoB 队头才能执行，即这条指令之前的指令已经全部完成。对于 MMIO Load 指令，要求完成虚实地址转换且 PMA / PMP 物理地址检查通过；对于 MMIO Store 指令，要求完成虚实地址转换且物理地址检查通过，且写数据就绪。然后 LSQ 负责将访存请求发往 Uncache 模块，Uncache 模块通过总线访问外设，得到结果后又 LSQ 写回 RoB。

原子指令、向量指令不支持 MMIO 访问，如果有访问 MMIO 地址空间的这类指令会报相应的 AccessFault 异常。

16.1.3.11 Uncache 指令的执行

香山核除了支持访问非幂等的、强定序的 MMIO 地址空间，还支持访问幂等的、弱一致性 (RVWMO) 的 Non-cacheable 地址空间，后者简称 NC，软件通过将页表的 PBMT 位域配置成 NC 来覆盖原有的 PMA 属性。不同于 MMIO 访问，NC 访问允许乱序访存，NC Load 的执行不会有副作用，因此可以推测执行。

在 LoadUnit / StoreUnit 流水线上被认为是 NC 地址 (PBMT = NC) 的访存指令会在 LSQ 中进行标记。LSQ 负责将 NC 访存发往 Uncache 模块。Uncache 支持同时处理多个 NC 请求，支持请求合并，并负责前递 Store 给 LoadUnit 中正在执行的 NC Load。

原子指令、向量指令不支持 NC 访问，如果有访问 NC 地址空间的这类指令会报相应的 AccessFault 异常。

16.1.3.12 非对齐访存

香山核支持标量和向量访存指令对 Memory 空间的非对齐访问。

- 不跨 16B 边界的标量非对齐访存，直接正常访问即可，不需要额外处理
- 跨 16B 边界的标量非对齐访存，会在 MisalignBuffer 拆分成 2 次对齐的访存操作，完成后 MisalignBuffer 负责拼接和写回
- 向量非 Segment 的 Unit-stride 指令会访问连续的一段地址空间，在元素合并后一次访问连续 16B，因此不需要额外处理
- 向量非 Segment 的除 Unit-stride 外的其他指令，在 VSplit 模块中完成元素拆分和地址计算，发往流水线，如果是非对齐的元素会被发往 MisalignBuffer，剩下的流程和非对齐标量相同，区别在于最终 MisalignBuffer 会写回 VMerge 而不是直接写回后端
- 向量 Segment 指令的非对齐处理由 VSegmentUnit 独立完成，不复用标量访存通路，而是通过独立状态机完成

原子指令不支持非对齐访问，MMIO 和 NC 地址空间均不支持非对齐方位，这些情况会报 AccessFault 异常。

16.1.3.13 原子指令的执行

香山核支持 RVA 和 Zacas 指令集。香山目前的设计会将原子指令所访问的 cache 块缓存到 DCache 后再执行原子操作。

访存单元会侦听 Store 发射队列发射的地址和数据，如果是原子指令则进入 AtomicsUnit。AtomicsUnit 会完成 TLB 地址翻译、清空 SBuffer、访问 DCache 等一系列操作。

16.1.4 总体设计

16.1.4.1 整体框图和流水级

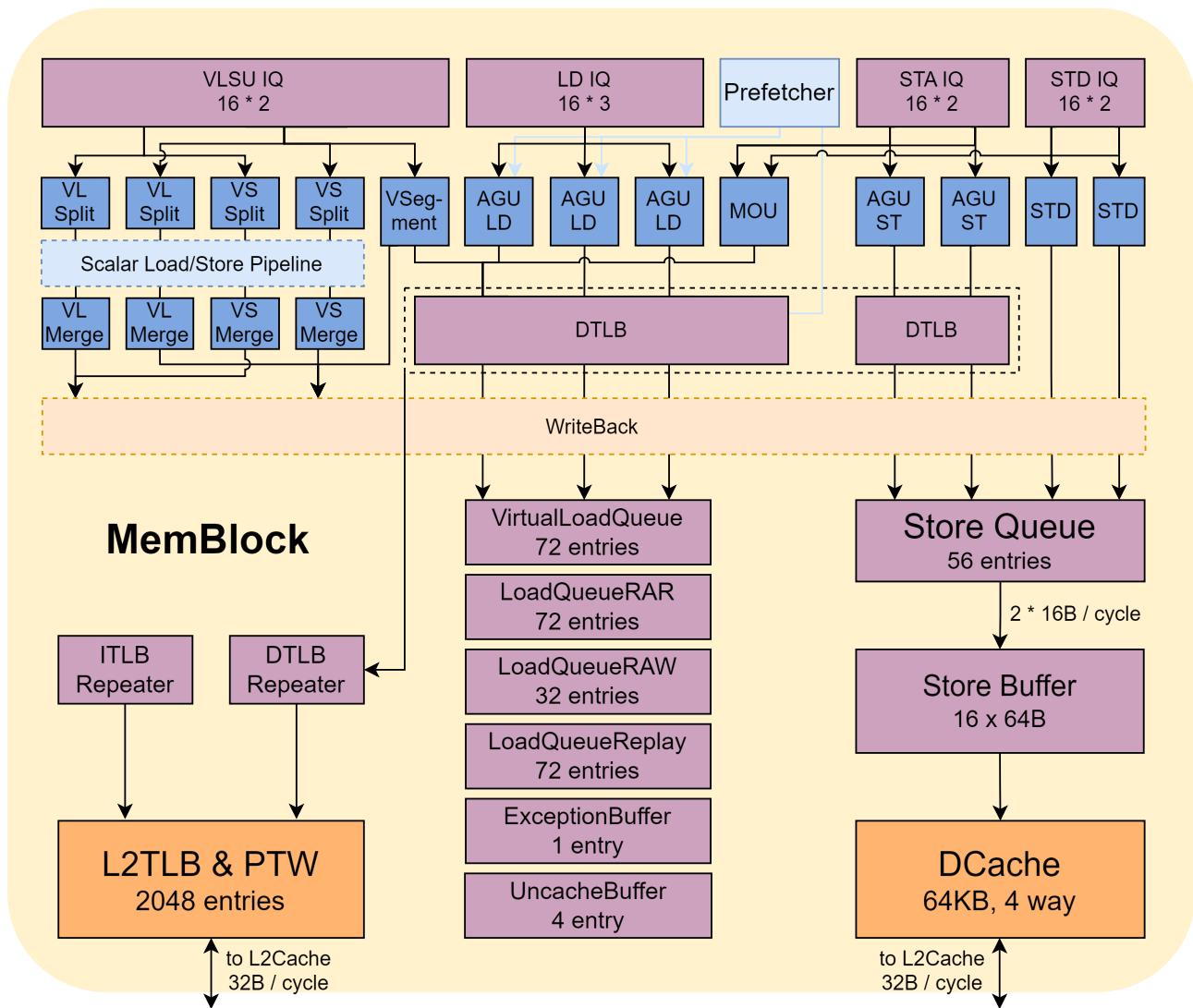


图 16.1: MemBlock 架构图

16.2 Load 指令执行单元 LoadUnit

16.2.1 功能描述

load 指令流水线，接收 load 发射队列发送的 load 指令，在流水线中处理完成后将结果写回 LoadQueue 和 ROB，用于指令提交以及唤醒后续依赖本条指令的其他指令。同时，LoadUnit 需要给发射队列、Load/StoreQueue 反馈一些必要的信息。LoadUnit 支持 128bits 数据宽度。

16.2.1.1 特性 1: LoadUnit 各级流水线功能

- stage 0

- 接收不同来源的请求，并做仲裁。
- 得到仲裁的指令向 tlb 和 dcache 发送查询请求。
- 流水线流给 stage 1。

仲裁的优先级从高到低列于下表。

表 16.3: LoadUnit 请求优先级

| stage 0 请求来源 | 优先级 |
|------------------------------------|-----|
| MisalignBuffer 的 load 请求 | 高 |
| dcache miss 导致的 loadQueueReplay 重发 | |
| LoadUnit 的快速重发 | |
| uncache 请求 | |
| nc 请求 | |
| LoadQueueReplay 的其他重发 | |
| 高置信度的硬件预取请求 | |
| 向量 load 请求 | |
| 标量 load/软件预取请求 | |
| load pointchaisng 请求 | |
| 低置信度的硬件预取请求 | 低 |

目前昆明湖架构不支持 load pointchaisng。

· stage 1

- 接收来自 stage 0 的请求。
- s1_kill: 当 fast replay 虚实地址匹配失败, l2l fwd 失败, 或 redirect 信号有效时, 会将 s1_kill 信号置为 true。
- 可能向 tlb 或 dcache 追发 kill 信号。
- 收到 tlb 的回复, 根据物理地址查询 dcache; 对于 hint 的情况, 一并发给 dcache。
- 向 storequeue && sbuffer 查询 st-ld forward。
- 接收 storeunit 请求, 判断是否存在 st-ld 违例。
- 检查是否发生异常。
- 如果是 nc 指令, 进行 PBMT 检查
- 如果是 prf_i 指令, 向前端发送请求

· stage 2

- 接收来自 stage 1 的请求。
- 接收 pmp 检查的回复, 判断是否发生异常; 同时整合异常来源。
- 接收 dcache 的回复信息, 判断是否需要重发等。
- 查询 LoadQueue 和 StoreQueue 是否发生 ld-ld 或 st-ld 违例

- 向后端发送快速唤醒信号
- 整合重发原因
- 如果是 nc 指令, 进行 PMA & PMP 检查
- stage 3
 - 接收来自 stage 2 的请求。
 - 向 SMS 预取器及 L1 预取器发送预取请求
 - 接收 dcache 返回的数据或前递的数据, 进行拼接和选择
 - 接收 uncache 的 load 请求写回
 - 将完成的 load 请求写回后端
 - 将 load 指令的执行状态更新至 LoadQueue 中
 - 向后端发送重定向请求

16.2.1.2 特性 2: 支持向量 load 指令

- LoadUnit 处理非对齐 Load 指令流程和标量类似, 优先级低于标量。特别的:
- stage 0:
 - * 接受 vlSplit 的执行请求, 优先级高于标量请求, 并且不需要计算虚拟地址
- stage 1:
 - * 计算 vecVaddrOffset 和 vecTriggerMask
- stage 3:
 - * 不需要向后端发送 feedback_slow 响应
 - * 向量 load 发起 Writeback, 通过 vecldout 发送给后端

16.2.1.3 特性 3: 支持 MMIO load 指令

- MMIO load 指令只是为了唤醒依赖于该指令的消费者指令。
- MMIO load 指令在 s0 向后端发送唤醒请求
- MMIO load 在 stage s3 写回数据

16.2.1.4 特性 4: 支持 Noncacheable load 指令

- LoadUnit 处理非对齐 Load 指令流程和标量类似, 优先级高于标量请求。特别的, Noncacheable load 指令将 2 次上流水线:
- 第一次上流水线, 判断指令 NC 属性
- 第二次上流水线:
 - * stage 0: 阶段判断出 NC 指令, 无需进行 tlb 翻译。
 - * stage 1: 发送前递请求到 StoreQueue,

- * stage 2: 判断 store 数据前递情况 (数据未准备好-重发处理, 虚实地址不匹配-重定向 N 处理)。
发送 RAR/RAW 违例请求,
- * stage 3: 判读违例情况 (ldld vio-重定向, stld vio-重定向处理), 如果 RAR 或 RAW 满/没有 ready, 需要 LoadQueueUncache 重发。如果不需要重发, 则通过 ldout 写回。
- 不支持非对齐的 Noncacheable load 指令
- 支持从 LoadQueueUncachce 获得前递数据。

16.2.1.5 特性 5: 支持非对齐 load 指令

- 非对齐 load 指令将 4 次上流水线:
 - 第一次上流水线, 判断是否是非对齐指令, 如果是非对齐指令, 则 LoadMisalignBuffer 发送非对齐请求;
 - 第二次上流水线, 执行拆分的第一条对齐的 load 指令, 成功执行后, 向 LoadMisalignBuffer 发送响应, 否则从 LoadMisalignBuffer 里重发;
 - 第三次上流水线, 执行拆分的第二条对齐的 load 指令, 成功执行后, 向 LoadMisalignBuffer 发送响应, 否则从 LoadMisalignBuffer 里重发;
 - 第四次上流水线, 在 s0 唤醒 load 指令之后的消费者, 同时, load 指令从 LoadMisalignBuffer 写回。
- Load 处理非对齐 Store 指令流程和标量类似, 特别的:
 - stage 0:
 - * 接受来自 LoadMisalignBuffer 的勤求, 优先级高于向量和标量请求, 并且不需要计算虚拟地址
 - stage 3:
 - * 如果不是来自于 LoadMisalignBuffer 的请求并且没有跨越 16 字节边界的非对齐请求, 那么需要进入 LoadMisalignBuffer 处理, 通过 io_misalign_buf 接口, 向 LoadMisalignBuffer 发送入队请求
 - * 如果是来自与 LoadMisalignBuffer 的请求并且没有跨越 16 字节边界请求, 则需要向 LoadMisalignBuffer 发送重发或者写回响应, 通过 io_misalign_ldout 接口, 向 LoadMisalignBuffer 发送响应
 - * 如果 misalignNeedWakeUp == true, 则直接写回, 否则需要进入 LoadMisalignBuffer 重发

16.2.1.6 特性 6: 支持预取请求

- LoadUnit 接受两种预取请求
 - 高置信度预取 (confidence > 0)
 - 低置信度预取 (confidence == 0)
- 支持预取训练
 - stage s2:
 - * 通过 io_prefetch_train_l1 训练 L1 预取
 - * 通过 io_prefetch_train 训练 SMS 预取

16.2.2 整体框图

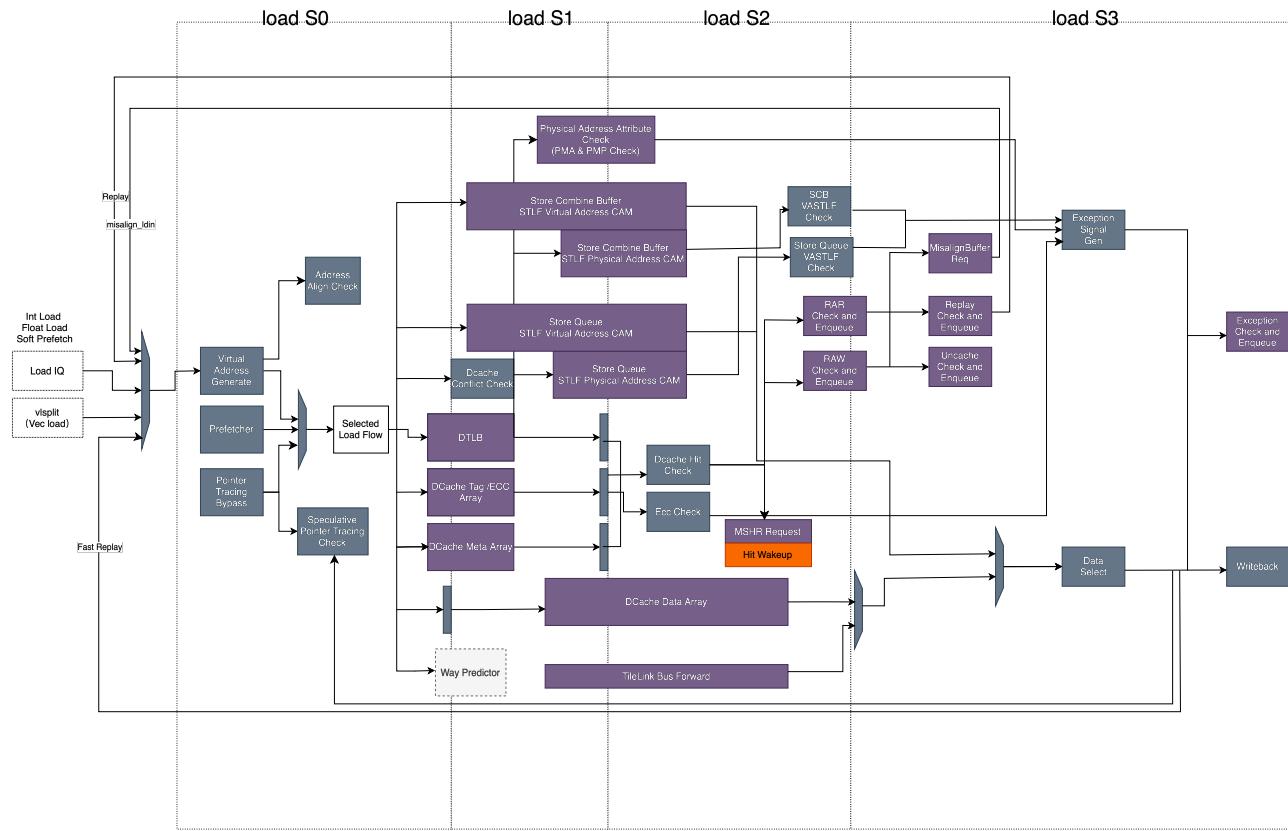


图 16.2: LoadUnit 整体框图

16.2.3 接口时序

16.2.3.1 LoadUnit 接口时序实例

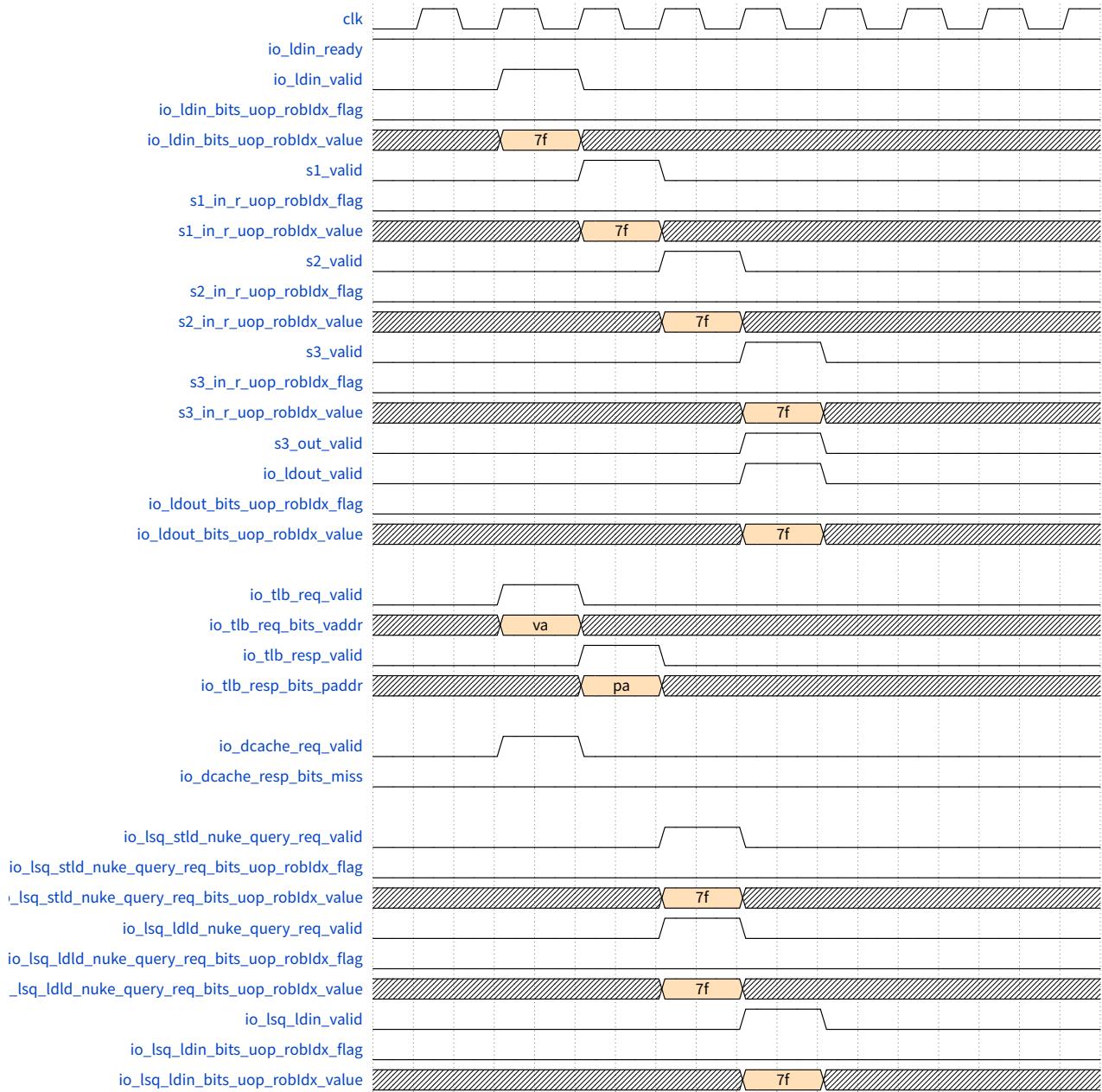


图 16.3: LoadUnit 接口时序

load 指令进入 LoadUnit 后，在 stage 0 请求 TLB 和 DCache，stage 1 得到 TLB 返回的 paddr，stage 2 得到是否命中 DCache。在 stage 2 进行 RAW 和 RAR 违例检查，stage 3 通过 io_lsq_ldin 更新 LoadQueue。在 stage 3 通过 ldout 写回。

16.2.3.2 stage 0 不同源仲裁时序实例

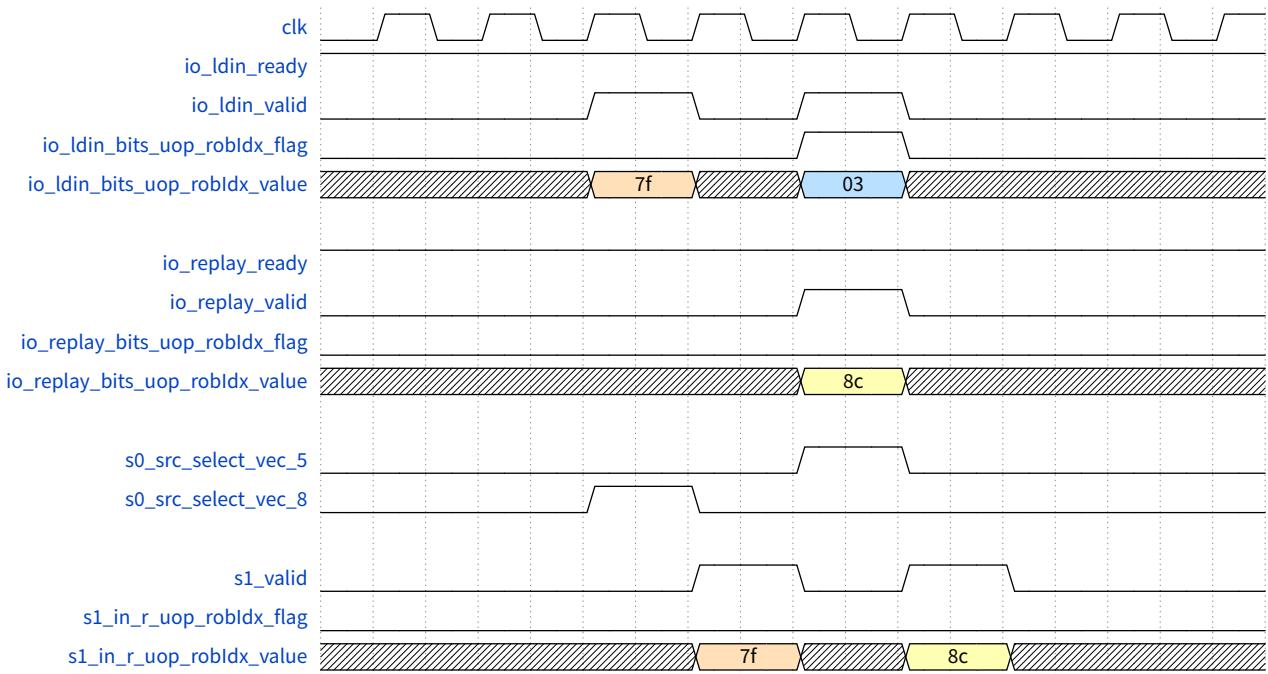


图 16.4: stage 0 不同源仲裁时序

图中示例了不同来源的 load 指令在 stage 0 的仲裁，第三个 clk 只有 io_ldin_valid 有效，且握手成功，在下一拍进入 stage 1。第五个 clk 中 io_ldin_valid 和 io_replay_valid 同时有效，由于 replay 请求比标量 load 的优先级高，所以 replay 请求获得仲裁，进入 stage 1。

16.3 Store 地址执行单元 StoreUnit

16.3.1 功能描述

Store 指令地址流水线分为 S0/S1/S2/S3/S4 五级，16.5所示。接收 store 地址发射队列发来的请求，处理完成之后需要给后端和向量部分响应，处理过程中需要给发射队列反馈信息，给 StoreQueue 反馈信息，最后写回，如果中间出现异常则从发射队列重新发射。

16.3.1.1 特性 1: StoreUnit 支持标量 Store 指令

- stage 0:
 - 计算 VA 地址
 - 地址非对齐检查更新到 uop.cf.exceptionVec(storeAddrMisaligned)
 - 发出 DTLB 读请求到 tlb
 - 更新指令的 mask 信息到 s0_mask_out 发送到 StoreQueue
 - 判断是否为数据宽度为 128bits 的 store 指令。
- stage 1:

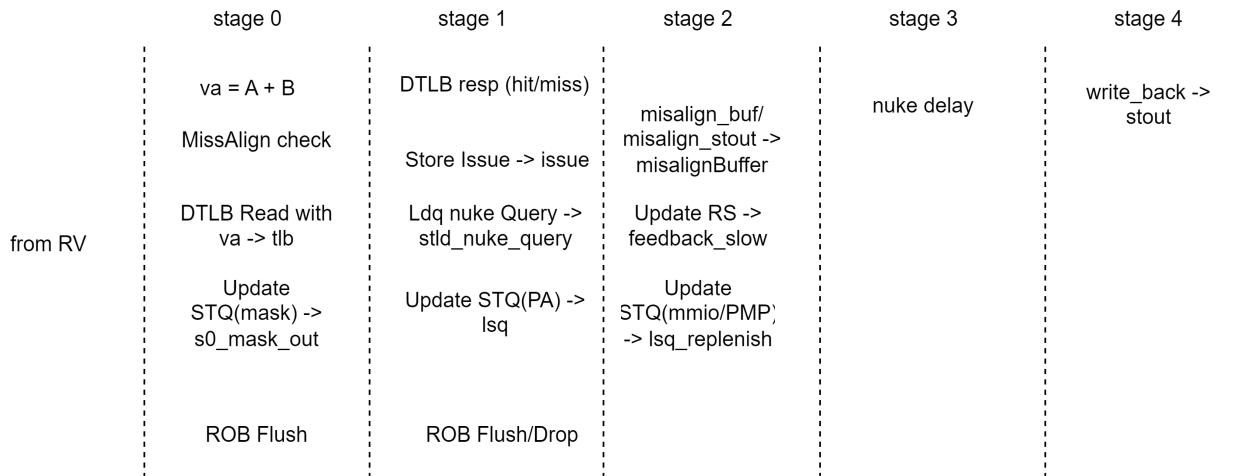


图 16.5: StoreUnit 流水线

- 将 DTLB 查询结果更新到 storeQueue
- 向 LoadQueue 发出 store-load 违例检查请求
- 如果 DTLB hit, 将 store issue 信息发送到后端
- stage 2:
 - mmio/PMP 检查并更新 storeQueue
 - 更新 DTLB 结果通过 feedback_slow 更新到后端
- stage 3
 - 为了和 RAW 违例检查同步发送给后端, 需要增加一拍
- stage 4
 - 标量 store 发起 Writeback, 通过 stout 发送给后端

16.3.1.2 特性 3: StoreUnit 支持向量 Store 指令

StoreUnit 处理非对齐 Store 指令流程和标量类似, 特别的:

- stage 0:
 - 接受 vsSplit 的执行请求, 优先级高于标量请求, 并且不需要计算虚拟地址
- stage 1:
 - 计算 vecVaddrOffset 和 vecTriggerMask
- stage 2:
 - 不需要向后端发送 feedback_slow 响应
- stage 4:
 - 向量 store 发起 Writeback, 通过 vecstout 发送给后端

16.3.1.3 特性 2: StoreUnit 支持非对齐 Store 指令

StoreUnit 处理非对齐 Store 指令流程和标量类似，特别的：

- stage 0:
 - 接受来自 StoreMisalignBuffer 的勤求，优先级高于向量和标量请求，并且不需要计算虚拟地址
- stage 2:
 - 不需要向后端发送 feedback 响应，
 - 如果不是来自于 StoreMisalignBuffer 的请求并且没有跨越 16 字节边界的非对齐请求，那么需要进入 StoreMisalignBuffer 处理
 - * 通过 io_misalign_buf 接口，向 StoreMisalignBuffer 发送入队请求
 - * 不进入 stage 3
 - 如果是来自与 StoreMisalignBuffer 的请求并且没有跨越 16 字节边界请求，则需要向 StoreMisalignBuffer 发送重发或者写回响应
 - * 通过 io_misalign_sout 接口，向 StoreMisalignBuffer 发送响应
 - * 如果出现 TLB miss，则需要重发，否则写回
 - * 不进入 stage 3

16.3.2 整体框图

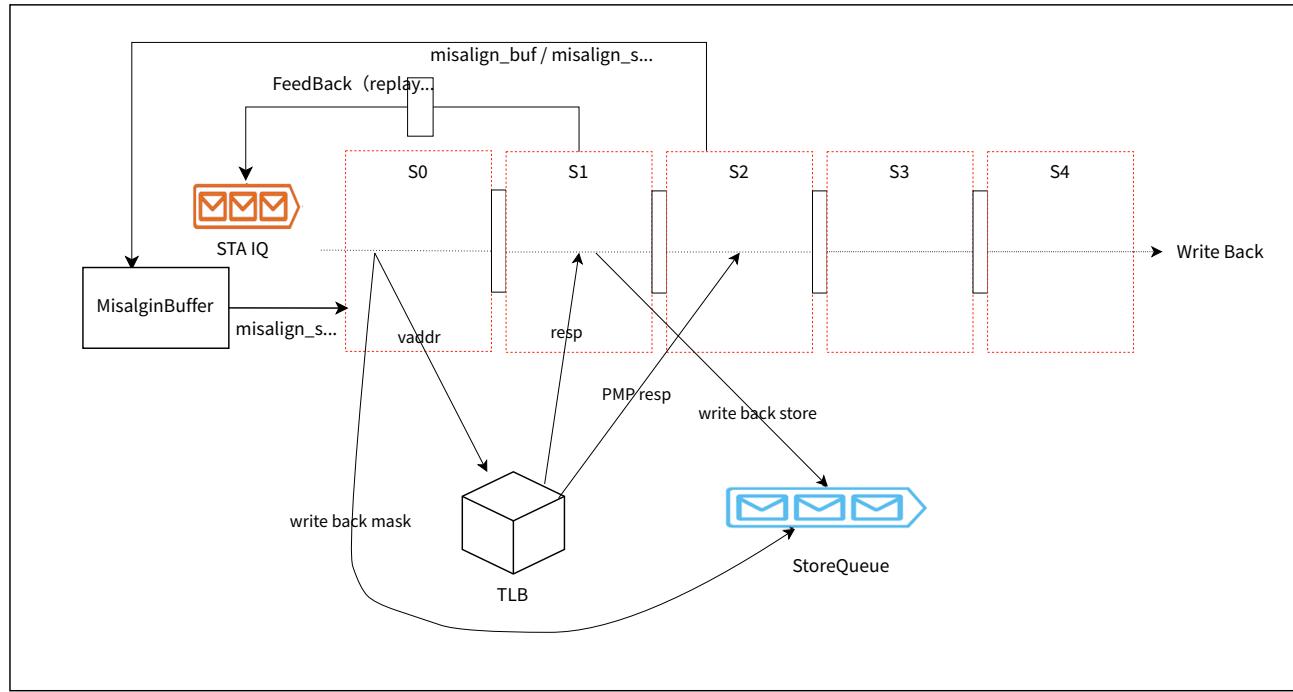


图 16.6: StoreUnit 整体框图

16.3.3 接口时序

16.3.3.1 接口时序实例

如图16.7所示, store 指令进入 StoreUnit 后, 在 stage 0 请求 TLB, stage 1 得到 TLB 返回的 paddr。在 stage 0 将 mask 写入 StoreQueue, stage 1 向 RAW 发送请求, 并通过 io_lsq 将 store 指令的其他信息更新到 LoadStoreQueue。在 stage 2 得到 feedback 相关信息, stage 4 通过 stout 写回。

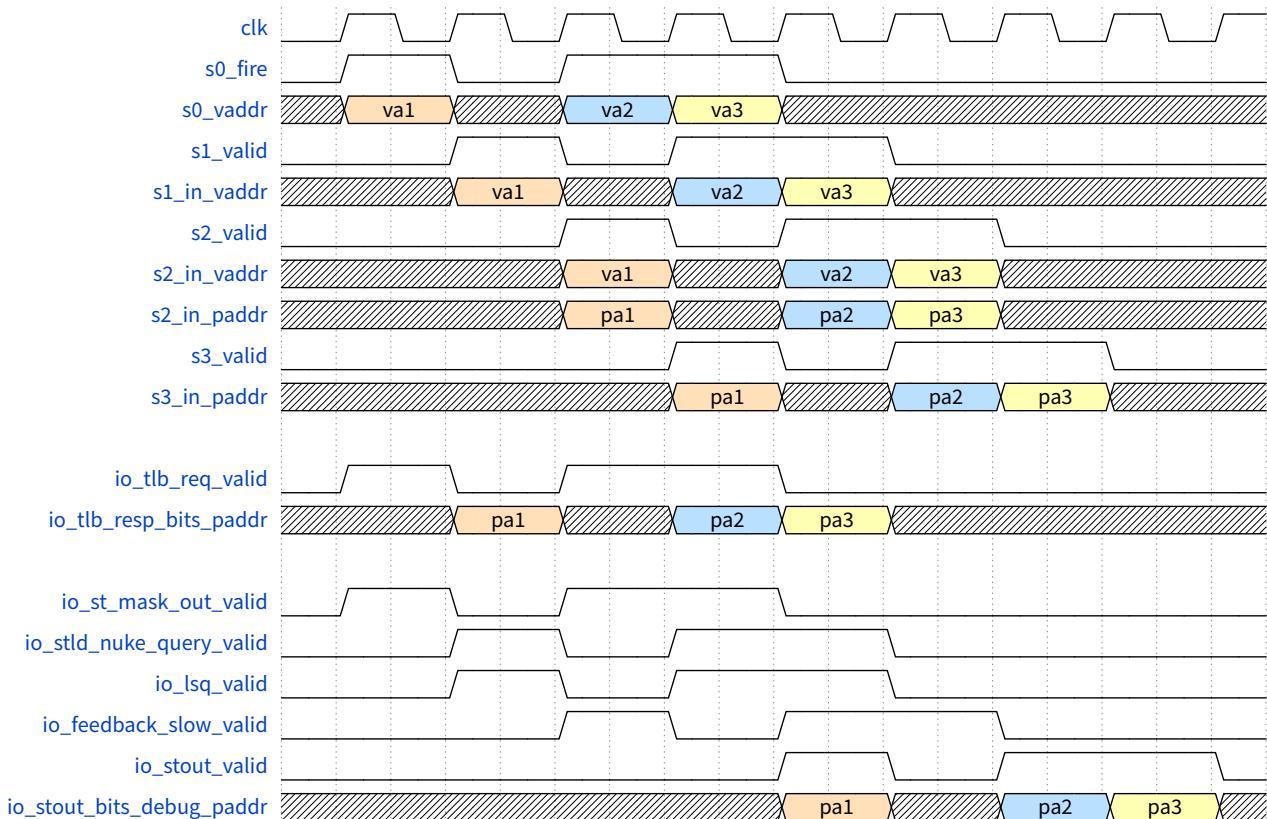


图 16.7: StoreUnit 接口时序

16.4 Store 数据执行单元 StdExeUnit

16.4.1 功能描述

标量 store 指令数据流水线, 用于给 StoreQueue 对应位置写入 store 的数据。

16.4.2 整体框图

16.4.3 接口时序

16.4.3.1 接口时序实例

如图16.9, io_ooo_to_mem_issueStd_0_ready 和 io_ooo_to_mem_issueStd_0_valid 为高握手后, 接收到了有效的写入请求, 数据为 io_ooo_to_mem_issueStd_0_bits_src_0, 上图示例了在第三个 clk 时写入到

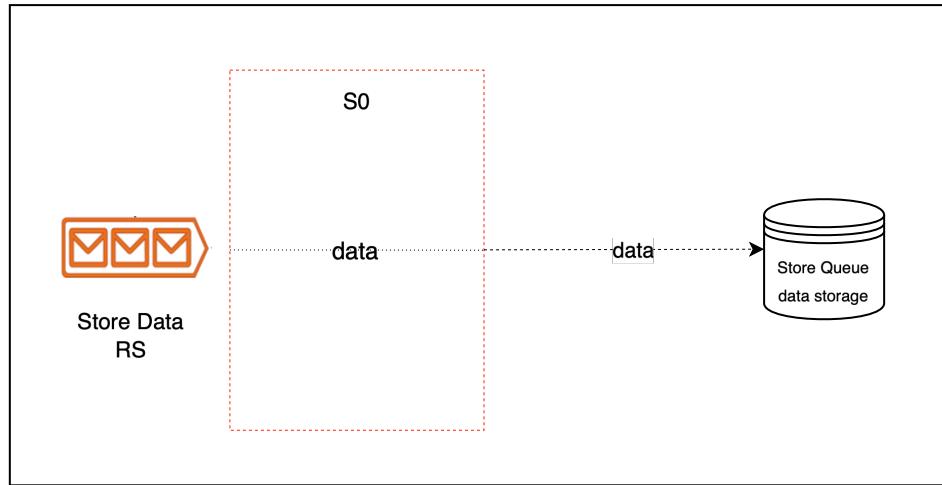


图 16.8: stdExeUnit 整体框图

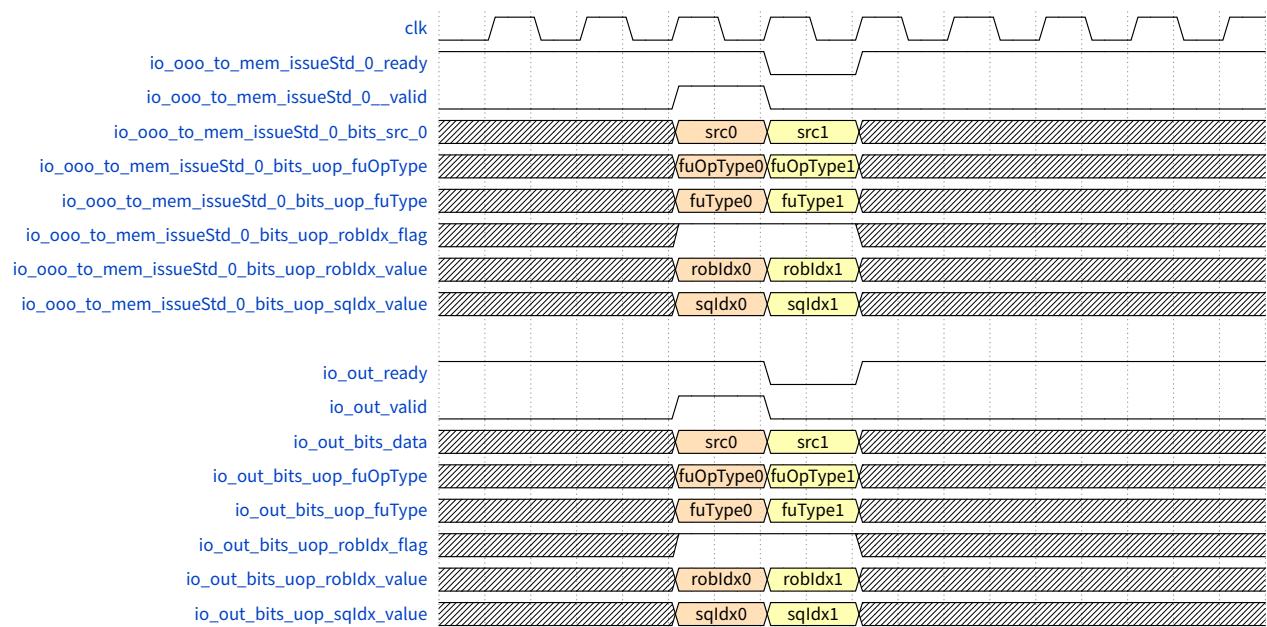


图 16.9: stdExeUnit 有效请求接口时序

StoreQueue 的第 sqIdx0 项, 数据为 src0。在第四个 clk 时 io_ooo_to_mem_issueStd_0_ready 为低电平, 此时数据不写入 StoreQueue。这种情况一般是有向量 store 指令要向 StoreQueue 写数据。

16.5 原子指令执行单元 AtomicsUnit

16.5.1 功能描述

AtomicsUnit 用于执行原子指令，包括 A 扩展 (LR/SC 和 AMO 指令) 和 Zacas 扩展 (AMOCAS.W, AMOCAS.D 和 AMOCAS.Q)。PMA 默认 DDR 地址空间均支持全部 AMO 和 AMOCAS 指令。

原子指令基本执行流程如下：

1. **sta 发射**: AtomicsUnit 与 StoreUnit 共用发射端口, 倾听来自保留站的 sta uop
 2. **std 发射**: 原子指令与 store 指令共用 StdExeUnit 执行单元, StdExeUnit 的执行结果会发往 AtomicsUnit, AtomicsUnit 负责收集原子指令执行所需的全部数据
 3. **地址翻译**: AtomicsUnit 与 LoadUnit_0 共用 DTLB 端口进行地址翻译, 同时需要做 PMA / PMP 等物理地址检查
 4. **清空 SBuffer**: 目前原子指令的执行一律按照 aq/r1 置位处理, 因此执行前需要清空 SBuffer
 5. **访问 DCache**: 向 DCache 发送原子操作请求, DCache 完成后向 AtomicsUnit 返回结果
 6. **写回**: AtomicsUnit 将执行结果写回寄存器堆

16.5.2 整体框图

AtomicsUnit 的有限状态机如图所示：

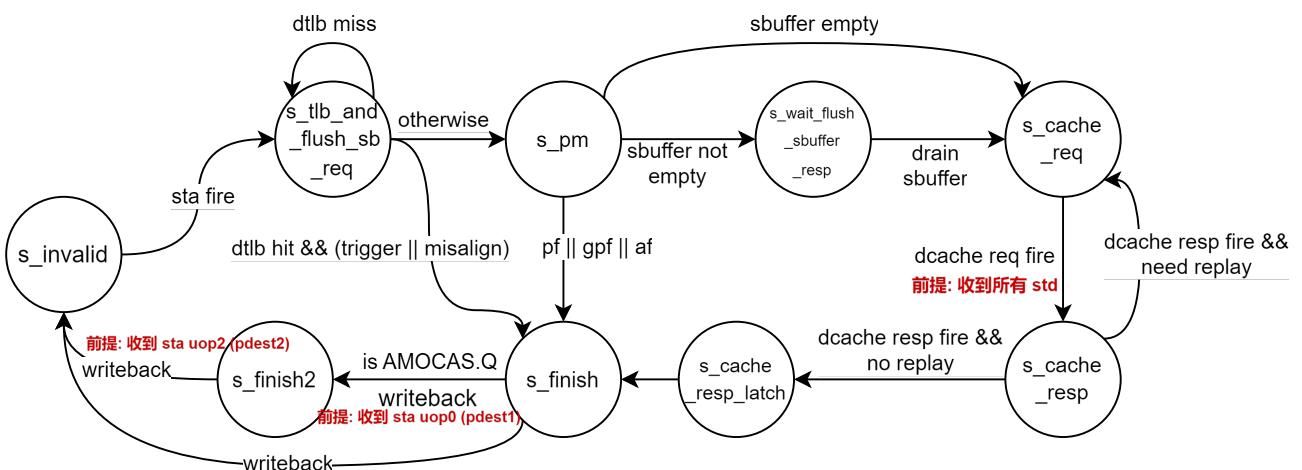


图 16.10: AtomicUnit 状态机示意图

- **s_invalid**: AtomicsUnit 空闲，收到保留站发射的 sta uop 后进入 s_tlb_and_flush_sb_req 状态
 - **s_tlb_and_flush_sb_req**: 访问 TLB 进行地址翻译，如果 TLB 缺失，则持续访问 TLB 直到命中；同时请求 SBuffer 清空。TLB 命中后，如果触发 debug trigger，或者有地址非对齐异常，则直接进入 s_finish 状态写回后端，否则进入 s_pm 状态做物理地址权限检查和进一步的异常检查。其中在访问 TLB 时：
 - 如果是 LR 指令，需要读权限
 - 如果是 SC 指令或其他 AMO 指令，需要写权限
 - **s_pm**: 物理地址权限检查和异常处理，如果发生下面任何一种异常，则进入 s_finish 状态写回后端：

- 如果 LR 指令访问 TLB 返回异常，报相应的 LoadPageFault / LoadAccessFault / LoadGuestPage-Fault 异常
- 如果除 LR 指令以外的其他原子指令访问 TLB 返回异常，报相应的 StorePageFault / StoreAccess-Fault / StoreGuestPageFault 异常
- 如果 PBMT 属性为 PMA，且 PMA 属性为 MMIO，根据是否是 LR 指令报相应的 LoadAccessFault / StoreAccessFault
- 如果 PBMT 属性为 IO 或 NC，根据是否是 LR 指令报相应的 LoadAccessFault / StoreAccessFault
- 如果 PMP 属性为 MMIO，或者返回读 / 写权限检查异常，根据是否是 LR 指令报相应的 LoadAccess-Fault / StoreAccessFault

如果上述异常都没有发生，则开始清空 SBuffer:

- 如果 SBuffer 不空，进入 s_wait_flush_sbuffer_resp 状态等待 SBuffer 清空
- 如果 SBuffer 已经清空，进入 s_cache_req 状态访问 DCache
- **s_wait_flush_sbuffer_resp:** 等待 SBuffer 清空后，清空后进入 s_cache_req 状态访问 DCache
- **s_cache_req:** 在收集全部 std uop 后向 DCache 发送访问请求，成功握手后进入 s_cache_resp 状态等待 DCache 处理完成的响应
 - 需要注意的是，AMOCAS 指令需要从后端接收多个 std uop，AtomicsUnit 在 s_cache_req 状态下需要等到全部 std uop 均接收后才可以开始向 DCache 发请求
- **s_cache_resp:** 等待 DCache 处理原子操作并返回结果
 - 如果 DCache 暂时无法处理该请求，需要 AtomicsUnit 重发，则回到 s_cache_req 状态重新发送请求
 - 否则不需要重发，进入 s_cache_resp_latch 状态
- **s_cache_resp_latch:** 对 DCache 返回的数据进行移位和有符号 / 无符号扩展，由于时序原因所以加了一拍。下一拍进入 s_finish 状态
 - 如果 DCache 返回了 error，需要记录相应的 LoadAccessFault / StoreAccessFault
- **s_finish:** 将原子指令执行结果写回
 - 如果是 LR 指令或 AMO 指令，写回内存中读到的旧值
 - 如果是 SC 指令，写回 SC 指令有无成功执行，成功则写回 0，失败则写回 1

写回成功握手后：

- 如果是 AMOCAS.Q 指令，总共需要写回 16B 的数据，前面提到 AMOCAS.Q 指令需要接收 2 个 sta uop，同理也需要分 2 拍写回，且 2 次写回的 pdest 需要和 2 次发射的 uop 的 pdest 分别对应。AMOCAS.Q 指令的 2 个 sta uop 是没有固定的发射顺序的，但是写回需要按照顺序写回，所以在 s_finish 状态下做第 1 次写回时，需要保证第 1 个 sta uop 已经收到（这样才能保证写回的 pdest 是正确的）。第 1 次写回成功后进入 s_finish2 状态做第 2 次写回
- 如果不是 AMOCAS.Q 指令，写回握手成功后进入 s_invalid 状态，状态机结束

- **s_finish2:** 对于 AMOCAS.Q 指令, AtomicsUnit 需要做第 2 次写回来写回 16B 中的高 8B 数据。写回的条件是需要确保已收到第 2 个 sta uop。写回握手成功后进入 s_invalid 状态, 状态机结束

16.5.3 Zacas 扩展

1. AMOCAS.W 指令从内存中加载 rs1 所指向的 4B 数据, 并和 rd 的低 4B 数据作比较, 如果相等则将 rs2 的低 4B 写入 rs1 所指向的内存; 最终内存加载的旧值写回 rd 寄存器
 2. AMOCAS.D 指令从内存中加载 rs1 所指向的 8B 数据, 并和 rd 作比较, 如果相等则将 rs2 写入 rs1 所指向的内存; 最终内存加载的旧值写回 rd 寄存器
 3. AMOCAS.Q 指令从内存中加载 rs1 所指向的 16B 数据, 并和 rd 和 rd+1 拼接的数据作比较, 如果相等则将 rs2 和 rs2+1 拼接的 16B 数据写入 rs1 所指向的内存; 最终内存加载的旧值低 8B 写回 rd 寄存器, 高 8B 写回 rd+1 寄存器
- 需要注意的是, 关于 rs2 和 rd 的寄存器对, 如果源操作数是 x0 寄存器, 那么寄存器对的读结果为全 0; 如果目的寄存器是 x0 寄存器, 那么寄存器对的每一个寄存器都不会被写

16.5.4 原子指令的 Uop 拆分

A 扩展中每条指令会拆分成一个 sta uop 和一个 std uop, 做一次写回 (写回次数和 sta uop 数量相同, std uop 不需要写回)。

AMOCAS 在指令 uop 拆分、发射和写回上和其他 A 扩展的指令有所不同。AMOCAS 指令在发射时除了需要提供写入内存的数据还需要用于比较的数据, 所以一条 AMOCAS 指令会被拆分成多个 std uop, 甚至多个 sta uop。

AMOCAS 指令复用 fuOpType 来区分多个 std uop 或多个 sta uop。fuOpType 共 9 bits, 原子指令只用到了 6 bits, 因此高 3 bits 用于标记 uopIdx。

具体的 uop 拆分规则如下:

1. **A 扩展指令 (包括 LR / SC 和普通 AMO 指令):** sta 和 std 的 uopIdx 均为 0, 分别携带 rs1 和 rs2 的数据, 存入 AtomicsUnit 中的 rs1 和 rs2_l 寄存器; AtomicsUnit 做一次写回操作, 写回的 uopIdx 为 0, 写回的 pdest 等于 sta uop 的 pdest

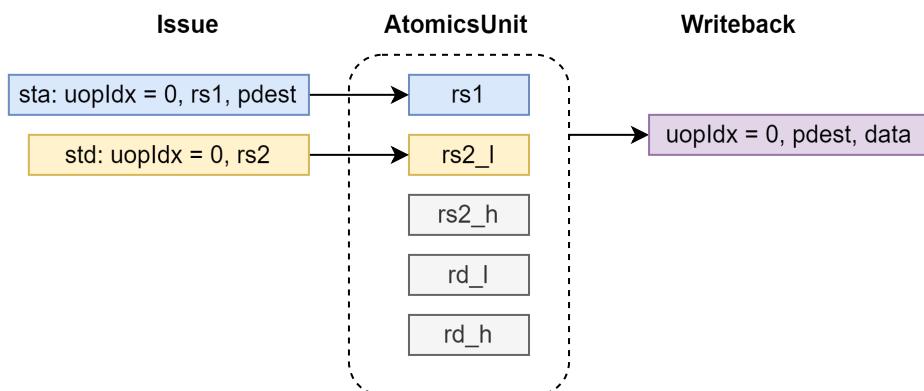


图 16.11: A 扩展原子指令的 Uop 拆分示意图

2. **AMOCAS.W 和 AMOCAS.D 指令:** 后端发射 1 个 sta uop 和 2 个 std uop:

- 1 个 sta uop 的 uopIdx 为 0
- 2 个 std uop 的 uopIdx 分别为 0 和 1, 分别保存 rd (用于比较的数据) 和 rs2 (如果比较成功需要存储的数据), 写入 AtomicsUnit 中的 rd_l 和 rs2_l 寄存器
- 最终做 1 次写回, 写回的 uopIdx 为 0, 写回的 pdest 等于 sta uop 的 pdest

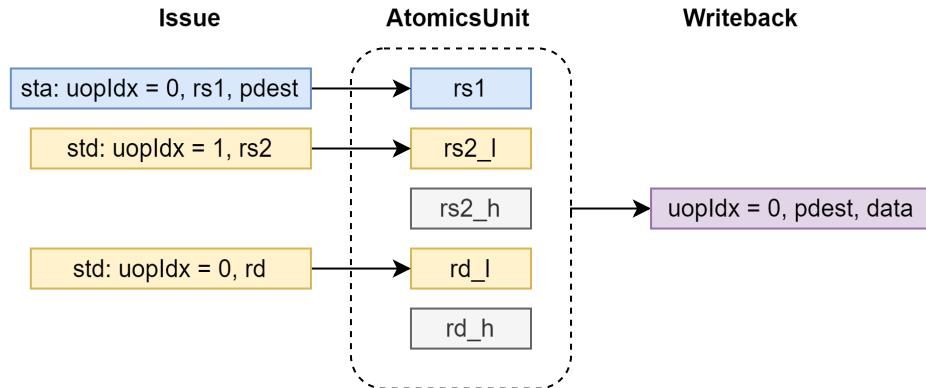


图 16.12: AMOCAS.W 和 AMOCAS.D 指令的 Uop 拆分示意图

3. AMOCAS.Q 指令: 后端发射 2 个 sta uop 和 4 个 std uop:

- 2 个 sta uop 的 uopIdx 分别为 0 和 2, 两个 uop 的 pdest 记为 pdest1 和 pdest2
- 4 个 std uop 的 uopIdx 为 0-3, 其中 0 号和 2 号 uop 分别保存 rd 的低位和高位, 写入 rd_l 和 rd_h 寄存器; 1 号和 3 号 uop 分别保存 rs2 的低位和高位, 写入 rs2_l 和 rs2_h 寄存器
- 最终做 2 次写回, 写回的 uopIdx 分别为 0 和 2, pdest 分别为 pdest1 和 pdest2, 写回数据分别为内存加载的旧值的低位和高位

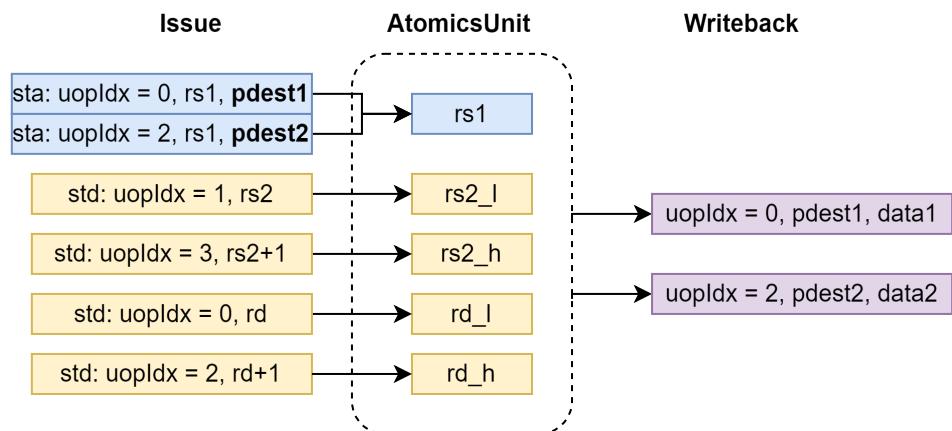


图 16.13: AMOCAS.Q 指令的 Uop 拆分示意图

16.5.5 异常汇总

原子指令可能发生的异常包括:

- 地址非对齐异常: 原子操作的地址必须根据操作类型 (字/双字/四字) 对齐 (4B / 8B / 16B), 否则报地址非对齐异常

- **非法指令异常** (后端译码级完成检查, 与访存无关): AMOCAS.Q 指令要求寄存器对 rs2 和 rd 的寄存器号必须是偶数, 如果是奇数需要报非法指令异常
- **断点异常**: 如果 trigger 比较命中, 需要报断点异常
- **地址翻译与权限检查有关的异常**
 - 如果 TLB 地址翻译返回异常, 根据是否是 LR 指令报相应的 Load 或 Store 的 PageFault / Access-Fault / GuestPageFault 异常
 - 如果 PMP 属性为 MMIO, 或者 PMP 没有相应读 / 写权限, 报 LoadAccessFault / StoreAccessFault
 - 如果 PMA + PBMT 属性为 IO 或 NC (包括下面 3 种情况), 报 LoadAccessFault / StoreAccessFault
 - * PBMT = IO
 - * PBMT = NC
 - * PBMT = PMA 且 PMA = MMIO

16.6 向量访存

16.6.1 向量访存

16.6.1.1 子模块列表

| 子模块 | 描述 |
|---------------|--|
| VLSplit | Vector Load uop 拆分模块 |
| VSSplit | Vector Store uop 拆分模块 |
| VLMergeBuffer | Vector Load flow 合并模块 |
| VSMergeBuffer | Vector Load flow 合并模块 |
| VSegmentUnit | Vector Segment 执行模块 |
| VfofBuffer | Vector fault only first 指令写回 VL 寄存器 uop 收集写回模块 |

16.6.1.2 功能描述

- 支持 RVV 1.0 完整全部访存指令
- 支持乱序调度 Vector Load/Store 指令
- 支持乱序执行 Vector Load/Store 指令拆分的 Uop
- 支持向量乱序违例检查与恢复
- 支持非对齐向量访存
- 不支持非 Memory 空间的向量访存

16.6.1.2.1 参数配置

| 参数 | 配置 (项数) |
|------|---------|
| VLEN | 128 |

| 参数 | 配置 (项数) |
|----------------|---------|
| VLMergeBuffer | 16 |
| VSMergeBuffer | 16 |
| VSegmentBuffer | 8 |
| VFOFBuffer | 1 |

16.6.1.2.2 功能概述

在进入 VLSIssueQueue 前，会在 Dispatch 阶段对 Load Queue 或 Store Queue 的 Index 进行分配。向量访存指令在后端被拆分为 uop 后，会首先在 Vspli 模块中进行译码、计算 mask 和地址偏移，同时也会申请 Mergebuffer 表项。在新向量访存架构中，会复用标量的 LoadUnit & StoreUnit，以及 Load Queue & Store Queue。

向量 Load 与 Store 共用两个 Issue Queue。对于向量 Load，两个 Issue Queue 对接两个 VLSP. 对于向量 Store 两个 Issue Queue 对接两个 VSSplit. 两个 VLSP 分布对应 LoadUnit0、LoadUnit1。两个 VSSplit 分布对应 StoreUnit0、StoreUnit1。当向量 Load 需要 Replay Queue 重发时，可能会被重发到其他 loadunit 上。在向量访存从 pipe 执行完毕后，会由 mergebuffer 汇总并写回。

16.6.1.3 整体框图

整体框图待更新

16.6.2 向量 Load 拆分单元 VLSP.

16.6.2.1 功能描述

接受并处理 Vector Load 指令的 uop。拆分 Uop，计算 Uop 相对基地址的偏移，生成标量访存 Pipeline 的控制型号。VLSP 总体上分为两个实现模块：VLSPipeline 和 VLSPbuffer。

16.6.2.1.1 特性 1：VLSPipeline 为 uop 进行二次译码

Vector Load 指令的拆分流水线。接受 Vector Load 发射队列发射的 Vector Load 指令的 Uop。在流水线中进行更细粒度的译码并计算 Mask 与地址偏移后发送到 VLSPbuffer 中。同时，VLSPipeline 还会根据译码计算的结果来申请 VLMergeBuffer 中的表项。

VLSPipeline 分为两个流水级：

S0:

- 通过传入的 Uop 信息进行更细粒度的译码。
- 根据指令类型生成 alignedType，使用 alignedType 指示 Load Pipeline 的访存宽度。
- 根据指令类型生成 preIsSplit 信号。preIsSplit 置高则表示不是 Unit-Stride 指令。
- 根据指令类型与 vm、emul、lmul、eew、sew 等信息生成该 Uop 的 Mask。
- 计算该条 Uop 的 VdIdx 用于后续后端数据合并写回使用。因为乱序执行的原因，同一条指令的 Uop 并不一定会背靠背执行，因此需要在该阶段根据指令类型、emul、lmul 和 uopidx 计算出 VdIdx。

S1:

- 计算 UopOffset 与 Stride。
- 计算该条 Uop 所需的 FlowNum。在这里，发送给 VMergeBuffer 的 FlowNum 与发送给 VSPLITBuffer 的 FlowNum 有所不同。MergeBuffer 中的 FlowNum 要用来判断这个 Uop 是否完成了所有有效的访存。而 VSPLITBuffer 中所使用的 FlowNum 需要用来进行拆分。
- 申请 VLMergeBuffer 表项。每个 Uop 申请一个表项。
- 发送信息到 VSPLITBuffer。

Mask 计算：

- 首先，我们根据 vm、v0、vstart、evl 来计算生成表示该条 Vector Load 指令的 SrcMask。这其中，evl 为有效向量长度，对于不同类型的 Vector Load 指令，有不同的的 evl 计算方法：
 - 对于 Load Whole 指令，其 $evl = NFIELDS * VLEN / EEW$ 。
 - 对于 Load Unit-Stride Mask 指令，其 $evl = \text{ceil}(vl/8)$ 。
 - 对于除了上述两种指令之外的 Vector Load 指令，其 $evl = vl$ 。
- 然后，我们使用这条指令的【当前 Uop 之前的所有的 Uop 的 FlowNum】和【当前 Uop 在内的的所有 Uop 的 FlowNum】与【当前 Uop 之前的所有的 Vd 的 FlowNum】来计算真正使用的 FlowMask。在这里，因为 Load Indexed 的特殊性，当 Indexed 指令的 $\$signed(emul) > \$signed(lmul)$ 时，我们需要保证同一个 VdIdx 的 Uop 的 FlowNum 在 VdIdx 内进行偏移，具体示例如下：
 - 首先我们假定如下配置向量 vlxuei 指令：
 - * vsetvli t1,t0,e8,m1,ta,ma lmul = 1
 - * vlxuei16.v v2,(a0),v8 emul = 2
 - * vl = 9, v0 = 0x1FF
 - 在这样的配置下，因为 $\$signed(emul) > \$signed(lmul)$ ，因此实际上会产生两个 Uop，表示需要分别从两个向量寄存器中取出 Index，而两个 Uop 对应的目的寄存器是同一个 Vd。也就是两个 Uop 的 VdIdx 应该是相同的，是要写入到同一个目标寄存器中的。因此在这里会产生如下的结果：
 - * uopIdxInField = 0, vdIdxInField = 0, flowMask = 0x00FF, toMergeBuffMask = 0x01FF
 - * uopIdxInField = 1, vdIdxInField = 0, flowMask = 0x0001, toMergeBuffMask = 0x01FF
 - * uopIdxInField = 0, vdIdxInField = 0, flowMask = 0x0000, toMergeBuffMask = 0x0000
 - * uopIdxInField = 0, vdIdxInField = 0, flowMask = 0x0000, toMergeBuffMask = 0x0000
 - 每一个 Uop 计算出的 FlowNum 均为 8。更具体的说明可见 VSPLIT.scala

16.6.2.1.2 特性 2：VSPLITBuffer 根据 VSPLITPipeline 产生的二次译码信息进行拆分

VSPLITBuffer 是只有一项的 Buffer，接受 VSPLITPipeline 发来的相关信息，缓存需要拆分的 Vector Load Uop。

VSPLITBuffer 会根据 Uop 的信息将一个 Uop 拆分成多个可以发送到标量 Load PipeLine 流水线上的信息，并发送到标量 Load PipeLine 流水线进行实际访存。

入队逻辑：

VSPLITBuffer 接受 VSPLITPipeline 发来的表项申请与相关信息，当 VSPLITBuffer 表项有空闲时会为每个申请分配一个 VSPLITBuffer 表项，并将相应表项的 Valid 置高。

出队逻辑：

VSPLITBuffer 接受 VSPLITPipeline 发来的表项申请与相关信息，当 VSPLITBuffer 表项有空闲时会为每个申请分配一个 VSPLITBuffer 表项，并将相应表项的 Valid 置高。

拆分：

- VLSSplitBuffer 会根据指令类型进行拆分。
- 对于 Unit-Stride 指令：
 - 当基地址对齐（不跨 CacheLine）时会一次访问 128 Bit。
 - 当基地址非对齐（跨 CacheLine）时候，我们会进行拆分，发起两次 128Bit 的访存。
- 对于其他的 Vector Load 指令，我们根据指令语义的要求按照元素进行拆分，并按照元素进行访存。
- 每次拆分都会将拆分后产生的相关信息发送到标量 Load PipeLine 流水线进行实际访存。
- 拆分根据 splitIdx 计数器进行判断，splitIdx 表示当前表项已经进行拆分的数量。当 splitIdx 小于需要拆分的数量并且可以发送到标量 Load PipeLine 流水线时，会进行一次拆分，每次拆分会增加 splitIdx 计数器的值。当 splitIdx 大于等于需要拆分的数量时，拆分结束，该表项出队，splitIdx 计数器归零。

地址计算：

- 在拆分时还需要计算将要发送到标量 Load PipeLine 流水线的相关信息，主要是计算每次拆分后需要进行访存的虚拟地址。
- 虚拟地址根据指令类型拆分方式的不同有不同的计算方式。
- 对于 Unit-Stride 指令：
 - 当基地址对齐（不跨 CacheLine）时直接进行一次 128Bit 对齐的访问即可。。
 - 当基地址非对齐（跨 CacheLine）时候，我们会进行拆分，使用两个连续的 128Bit 对齐的地址进行访问。
- 对于其他的 Vector Load 指令，我们根据指令语义的要求按照元素进行拆分，虚拟地址会根据元素以及语义进行计算。

重定向与异常处理：当重定向信号到来时，会根据重定向相关信息冲刷掉 VLSSplitBuffer 的相关表项。

16.6.2.1.3 特性 3：根据 VLMergeBuffer 的 Threshold 信号进行反压

参见 § 16.6.4.1.4 阈值反压。当收到来自 VLMergeBuffer 时，VLSSplitPipeline 会反压住入队请求，阻止后端发送新的 uop。只到 VLMergeBuffer 解除阈值反压。

16.6.2.2 整体框图

单一模块无框图。

16.6.2.3 主要端口

只列出 VLSSplit 对外接口，不包括内部 VLSSplitPipe 与 VLSSplitBuffer 接口。

| 方向 | 说明 |
|-------------------|-----------------------------------|
| redirect | In 重定向端口 |
| in | In 接收来自 Issue Queue 的 uop 发射 |
| toMergeBuffer.req | Out 请求 MergeBuffer 表项 |

| | 方向 | 说明 |
|--------------------|-----|------------------------|
| toMergeBuffer.resp | In | MergeBuffer 的响应 |
| | out | 发送访存请求至 Load Unit |
| threshold | In | 接收 VLMergeBuffer 的阈值信号 |

16.6.2.4 接口时序

接口时序较简单，只提供文字描述。

| | 说明 |
|--------------------|--------------------------------------|
| redirect | 具备 Valid。数据同 Valid 有效 |
| in | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| toMergeBuffer.req | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| toMergeBuffer.resp | 具备 Valid。数据同 Valid 有效 |
| out | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| threshold | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |

16.6.3 向量 Store 拆分单元 VSSplit

16.6.3.1 功能描述

接受并处理 Vector Store 指令的 uop。拆分 Uop，计算 Uop 相对基地址的偏移，生成标量访存 Pipeline 的控制型号。VSSplit 总体上分为两个实现模块：VSSplitPipeline 和 VSSplitBuffer。

16.6.3.1.1 特性 1：VSSplitPipeline 为 uop 进行二次译码

Vector Store 指令的拆分流水线。接受 Vector Store 发射队列发射的 Vector Store 指令的 Uop。在流水线中进行更细粒度的译码并计算 Mask 与地址偏移后发送到 VSSplitBuffer 中。同时，VSSplitPipeline 还会根据译码计算的结果来申请 VLMergeBuffer 中的表项。VSSplitPipeline 分为两个流水级：

16.6.3.1.1.1 S0:

- 通过传入的 Uop 信息进行更细粒度的译码。
- 根据指令类型生成 alignedType，使用 alignedType 指示 Store Pipeline 的访存宽度。
- 根据指令类型生成 preIsSplit 信号。preIsSplit 置高则表示不是 Unit-Stride 指令。
- 根据指令类型与 vm、emul、lmul、eew、sew 等信息生成该 Uop 的 Mask。
- 计算该条 Uop 的 VdIdx 用于后续后端数据合并写回使用。因为乱序执行的原因，同一条指令的 Uop 并不一定会背靠背执行，因此需要在该阶段根据指令类型、emul、lmul 和 uopidx 计算出 VdIdx。

Mask 计算：

- 首先，我们根据 vm、v0、vstart、evl 来计算生成表示该条 Vector Store 指令的 SrcMask。这其中，evl 为有效向量长度，对于不同类型的 Vector Store 指令，有不同的的 evl 计算方法：
 - 对于 Store Whole 指令，其 evl = NFIELDS*VLEN/EEW。

- 对于 Store Unit-Stride Mask 指令，其 $evl = \lceil vl/8 \rceil$ 。
- 对于除了上述两种指令之外的 Vector Store 指令，其 $evl = vl$ 。
- 然后，我们使用这条指令的【当前 Uop 之前的所有的 Uop 的 FlowNum】和【当前 Uop 在内的的所有 Uop 的 FlowNum】与【当前 Uop 之前的所有的 Vd 的 FlowNum】来计算真正使用的 FlowMask。在这里，因为 Store Indexed 的特殊性，当 Indexed 指令的 $\$signed(emul) > \$signed(lmul)$ 时，我们需要保证同一个 VdIdx 的 Uop 的 FlowNum 在 VdIdx 内进行偏移，具体示例如下：
 - 首先我们假定如下配置向量 vlxuei 指令：
 - * vsetvli t1,t0,e8,m1,ta,ma lmul = 1
 - * vsuxei16.v v2,(a0),v8 emul = 2
 - * vl = 9, v0 = 0x1FF
 - 在这样的配置下，因为 $\$signed(emul) > \$signed(lmul)$ ，因此实际上会产生两个 Uop，表示需要分别从两个向量寄存器中取出 Index，而两个 Uop 对应的目的寄存器是同一个 Vd。也就是两个 Uop 的 VdIdx 应该是相同的，是要写入到同一个目标寄存器中的。因此在这里会产生如下的结果：
 - * uopIdxInField = 0, vdIdxInField = 0, flowMask = 0x00FF, toMergeBuffMask = 0x01FF
 - * uopIdxInField = 1, vdIdxInField = 0, flowMask = 0x0001, toMergeBuffMask = 0x01FF
 - * uopIdxInField = 0, vdIdxInField = 0, flowMask = 0x0000, toMergeBuffMask = 0x0000
 - * uopIdxInField = 0, vdIdxInField = 0, flowMask = 0x0000, toMergeBuffMask = 0x0000
 - 每一个 Uop 计算出的 FlowNum 均为 8。更具体的说明可见 VSPLIT.scala

16.6.3.1.1.2 S1:

- 计算 UopOffset 与 Stride。
- 计算该条 Uop 所需的 FlowNum。在这里，发送给 VMergeBuffer 的 FlowNum 与发送给 VSPLITBuffer 的 FlowNum 有所不同。MergeBuffer 中的 FlowNum 要用来判断这个 Uop 是否完成了所有有效的访存。而 VSPLITBuffer 中所使用的 FlowNum 需要用来进行拆分。
- 申请 VSMergeBuffer 表项。每个 Uop 申请一个表项。
- 发送信息到 VSPLITBuffer。

16.6.3.1.2 特性 2：VSPLITBuffer 根据 VSPLITPipeline 产生的二次译码信息进行拆分

VSPLITBuffer 是只有一项的 Buffer，接受 VSPLITPipeline 发来的相关信息，缓存需要拆分的 Vector Store Uop。

VSSplitBuffer 会根据 Uop 的信息将一个 Uop 拆分成多个可以发送到标量 Store PipeLine 流水线上的信息，并发送到标量 Store PipeLine 流水线进行实际访存。

入队逻辑：

VSSplitBuffer 接受 VSPLITPipeline 发来的表项申请与相关信息，当 VSSplitBuffer 表项有空闲时会为每个申请分配一个 VSSplitBuffer 表项，并将相应表项的 Valid 置高。

出队逻辑：

VSSplitBuffer 接受 VSPLITPipeline 发来的表项申请与相关信息，当 VSSplitBuffer 表项有空闲时会为每个申请分配一个 VSSplitBuffer 表项，并将相应表项的 Valid 置高。

拆分：

- VsSplitBuffer 会根据指令类型进行拆分。

- 对于 Unit-Stride 指令：
- 当基地址对齐（不跨 CacheLine）时会一次访问 128 Bit。
- 当基地址非对齐（跨 CacheLine）时候，我们会进行拆分，发起两次 128Bit 的访存。
- 对于其他的 Vector Store 指令，我们根据指令语义的要求按照元素进行拆分，并按照元素进行访存。
- 每次拆分都会将拆分后产生的相关信息发送到标量 Store PipeLine 流水线进行实际访存。
- 拆分根据 splitIdx 计数器进行判断，splitIdx 表示当前表项已经进行拆分的数量。当 splitIdx 小于需要拆分的数量并且可以发送到标量 Store PipeLine 流水线时，会进行一次拆分，每次拆分会增加 splitIdx 计数器的值。当 splitIdx 大于等于需要拆分的数量时，拆分结束，该表项出队，splitIdx 计数器归零。

地址计算：

- 在拆分时还需要计算将要发送到标量 Store PipeLine 流水线的相关信息，主要是计算每次拆分后需要进行访存的虚拟地址。
- 虚拟地址根据指令类型拆分方式的不同有不同的计算方式。
- 对于 Unit-Stride 指令：
 - 当基地址对齐（不跨 CacheLine）时直接进行一次 128Bit 对齐的访问即可。。
 - 当基地址非对齐（跨 CacheLine）时候，我们会进行拆分，使用两个连续的 128Bit 对齐的地址进行访问。
- 对于其他的 Vector Store 指令，我们根据指令语义的要求按照元素进行拆分，虚拟地址会根据元素以及语义进行计算。

数据计算：

- 在拆分时还需要计算将要发送到 Store Queue 的相关信息，主要是计算每次拆分后的需要存储的数据。
- 需要存储的数据根据指令类型拆分方式的不同有不同的计算方式。具体可见上述地址计算部分要求，只需要与地址的粒度对齐即可。

重定向与异常处理：

当重定向信号到来时，会根据重定向相关信息冲刷掉 VSSplitBuffer 的相关表项。

16.6.3.2 整体框图

单一模块无框图。

16.6.3.3 主要端口

只列出 VSSplit 对外接口，不包括内部 VSSplitPipe 与 VSSplitBuffer 接口。

| | 方向 | 说明 |
|--------------------|-----|---------------------------|
| redirect | In | 重定向端口 |
| in | In | 接收来自 Issue Queue 的 uop 发射 |
| toMergeBuffer.req | Out | 请求 MergeBuffer 表项 |
| toMergeBuffer.resp | In | MergeBuffer 的响应 |

| | 方向 | 说明 |
|--|-----------------|---|
| | out Out | 发送访存请求至 Store Unit |
| | vstd Out | 执行结束的 uop 写回后端时更新 Store queue 中表项状态 |
| | vstdMisalign In | 接收 Store Unit 与 Store Misalign Buffer 的 misalign 相关信号 |

16.6.3.4 接口时序

接口时序较简单，只提供文字描述。

| | 说明 |
|--------------------|--------------------------------------|
| redirect | 具备 Valid。数据同 Valid 有效 |
| in | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| toMergeBuffer.req | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| toMergeBuffer.resp | 具备 Valid。数据同 Valid 有效 |
| out | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| vstd | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| vstdMisalign | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |

16.6.4 向量 Load 合并单元 VLMergeBuffer

16.6.4.1 功能描述

一个基于 freelist 的队列，接收 VLSplit 模块发来的请求，为后端发射的每一个 uop 申请一个表项，保存 uop 的相关信息，收集从 Load pipeline 上返回的数据，接收该 uop 拆分的全部访存请求之后写回后端与 Load Queue。

16.6.4.1.1 特性 1：维护 uop 的拆分访存请求

在 VLSplit 模块的 pipeline 第二阶段向 VLMergeBuffer 发起表项申请，同周期 VLMergeBuffer 向 VLSplit 返回一个表项 index，同时相应表项 allocated 置 true。入队同时会在对应项的计数器中写入当前 uop 拆分的访存请求数量。为每个 uop 分配一项，每一项维护所需要收集的 flow 数量。当全部收集之后标记为 uopfinish，按照 uop 为粒度写回。在标记 uopfinish 的表项中选择一项写回后端，当有多项可以写回时 index 小的先写回。同时清空相应标志位。

16.6.4.1.2 特性 2：合并数据

根据 Load 流水线输出信息，以 uop 为粒度合并数据。合并时候根据是否有异常、元素位置、mask 等进行合并。

16.6.4.1.3 特性 3：处理异常

根据流水线的输出信息，出现异常时正确的设置 ExceptionVec、vstart 等相应的数据。

16.6.4.1.4 特性 4：阈值反压

为了避免卡死，当 VLMergeBuffer 的空闲表项小于等于 6 项时，产生 threshold 反应信号至 VLSplit。反压 VLSplit Pipe。参见 § 16.6.2.1.3 根据 VLMergeBuffer 的 Threshold 信号进行反压。

16.6.4.2 整体框图

单一模块无框图

16.6.4.3 主要端口

| | 方向 | 说明 |
|----------------|-----|-------------------------------------|
| frompipeline | In | 接收来自 Load pipeline 的读数据返回 |
| fromSplit.req | In | 接收来自 VLSplit 模块的表项申请 |
| fromSplit.resp | Out | 反馈至 VLSplit 模块，是否成功分配、分配的表项 |
| uopWriteback | Out | 将执行结束的 uop 写回后端 |
| toLsq | Out | 执行结束的 uop 写回后端时更新 Load queue 中表项状态 |
| redirect | In | 重定向端口 |
| feedback | Out | 反馈至后端 Issue Queue，目前后端不做任何处理 |
| toSplit | Out | 反馈至 VLSplit 模块，VLMergeBuffer 即将达到阈值 |

16.6.4.4 接口时序

接口时序较简单，只提供文字描述。

| | 说明 |
|--------------------|--------------------------------------|
| frompipeline | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| fromSplit.req | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| fromSplit.resp | 具备 Valid。数据同 Valid 有效 |
| uopWriteback | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| toLsq | 具备 Valid。数据同 Valid 有效 |
| redirect | 具备 Valid。数据同 Valid 有效 |
| feedback | 具备 Valid。数据同 Valid 有效 |
| fromMisalignBuffer | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |

16.6.5 向量 Store 合并单元 VSMergeBuffer

16.6.5.1 功能描述

一个基于 freelist 的队列，接收 VSSplit 模块发来的请求，为后端发射的每一个 uop 申请一个表项，保存 uop 的相关信息，收集从 Store pipeline 上返回的数据，接收该 uop 拆分的全部访存请求之后写回后端与 Store Queue。

16.6.5.1.1 特性 1：维护 uop 的拆分访存请求

在 VSSplit 模块的 pipeline 第二阶段向 VSMergeBuffer 发起表项申请, 同周期 VSMergeBuffer 向 VSSplit 返回一个表项 index, 同时相应表项 allocated 置 true。入队同时会在对应项的计数器中写入当前 uop 拆分的访存请求数量. 为每个 uop 分配一项, 每一项维护所需要收集的 flow 数量。当全部收集之后标记为 uopfinish, 按照 uop 为粒度写回。在标记 uopfinish 的表项中选择一项写回后端, 当有多项可以写回时 index 小的先写回。同时清空相应标志位。

16.6.5.1.2 特性 2：处理异常

根据流水线的输出信息, 出现异常时正确的设置 ExceptionVec、vstart 等相应的数据。

16.6.5.1.3 特性 3：根据 StoreMisalignBuffer 的 flush 信号标记 uop 是否需要 flush

对于非对齐的 vector store 访存, 具有特殊性, 当 StoreMisalignBuffer 产生 vector store 的 flush 信号时, 会送至 VSMergeBuffer。VSMergeBuffer 会将对应的表现置为 needRSReplay, 从而最终通知 Issue Queue 重发。

16.6.5.2 整体框图

单一模块无框图。

16.6.5.3 主要端口

| | 方向 | 说明 |
|--------------------|-----|-------------------------------------|
| frompipeline | In | 接收来自 Store pipeline 的读数据返回 |
| fromSplit.req | In | 接收来自 VSSplit 模块的表项申请 |
| fromSplit.resp | Out | 反馈至 VSSplit 模块, 是否成功分配、分配的表项 |
| uopWriteback | Out | 将执行结束的 uop 写回后端 |
| toLsq | Out | 执行结束的 uop 写回后端时更新 Store queue 中表项状态 |
| redirect | In | 重定向端口 |
| feedback | Out | 反馈至后端 Issue Queue 是否需要重发 |
| fromMisalignBuffer | In | 接收来自 StoreMisalignBuffer 的 flush 信号 |

16.6.5.4 接口时序

接口时序较简单, 只提供文字描述。

| | 说明 |
|--------------------|--------------------------------------|
| frompipeline | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| fromSplit.req | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| fromSplit.resp | 具备 Valid。数据同 Valid 有效 |
| uopWriteback | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| toLsq | 具备 Valid。数据同 Valid 有效 |
| redirect | 具备 Valid。数据同 Valid 有效 |
| feedback | 具备 Valid。数据同 Valid 有效 |
| fromMisalignBuffer | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |

16.6.6 向量 Segment 访存指令处理单元 VSegmentUnit

16.6.6.1 功能描述

主体是一个 8 项的队列，每一项有一个 128-bits 的地址寄存器、128-bits 的数据寄存器、index/stride 寄存器和用于存储不同 uop 的物理寄存器号、写使能、uopidx 等信息的寄存器。除此之外还有一个用于存储整条指令译码信息的寄存器。内部使用一个状态机控制实现按照 segment 顺序进行拆分。

在 VSegmentUnit.scala 中写有与代码结合的注释，可以结合注释与代码阅读下文，理解 SegmentUnit 的相关逻辑。

在 Segment 指令执行时，需要由流水线乱序后端保证：前面的指令都执行结束，后面的指令都不能进入流水线(与原子指令的等待机制类似)，同时需要保证指令的 uop 按照拆分顺序进入 SegmentUnit。此时 SegmentUnit 对 Segment 指令的顺序才能得到保证。

16.6.6.1.1 特性 1：进行 Segment 指令的拆分

- segmentIdx: segment 的序号，segmentIdx <= vl。用于表示当前发送到哪个 segment，也用于选择数据、合并数据。
- fieldIdx: field 的序号，用于标识当前 segment 是否发送结束。fieldIdx < nfields。
- fieldOffset: 同一个 segment 下各个元素的相对偏移，实现为一个为 1 的累加器。
- segmentOffset: 用于记录不同 Segment 之间的偏移，对于 stride 指令来说是以 stride 为粒度的累加器；对于 unit-stride 来说是以 nfield*eew 为粒度的累加器；对于 index 来说是 segmentIdx 对应的索引寄存器元素。
- vaddr = baseaddr + (fieldIdx < eew) + segmentOffset

上图为队列指针跳转示例，展示了当 lmul=1, nf=2, vl=16 配置下的示例，segmentIdx 指向当前拆分的 segment，SplitPtr 指向拆分的 field 寄存器。上图中 segmentIdx 为 0，splitPtr 为 0，将第一个 uop 的第一个元素拆分并且访存之后，SplitPtr + nf，进行 segment0 的 field1 元素的访存。当进行了 field2 的访存之后，当前 segment 的元素访问结束，segmentIdx+1，同时 SplitPtr 跳转到下一个 segment 的 field0 所在的寄存器。当 segmentIdx 递增到 8 时，对于 field0 的寄存器组来说是下一个 uop 的第一个元素（对应上图中每个 field 寄存器的第二个）。当 segmentIdx=16，并且进行完 field2 元素的访存之后，指令执行结束。对于 segment Index 来说，还有一个指针用于选择索引寄存器，实现方式与上述选择同一 field 的不同寄存器类似。

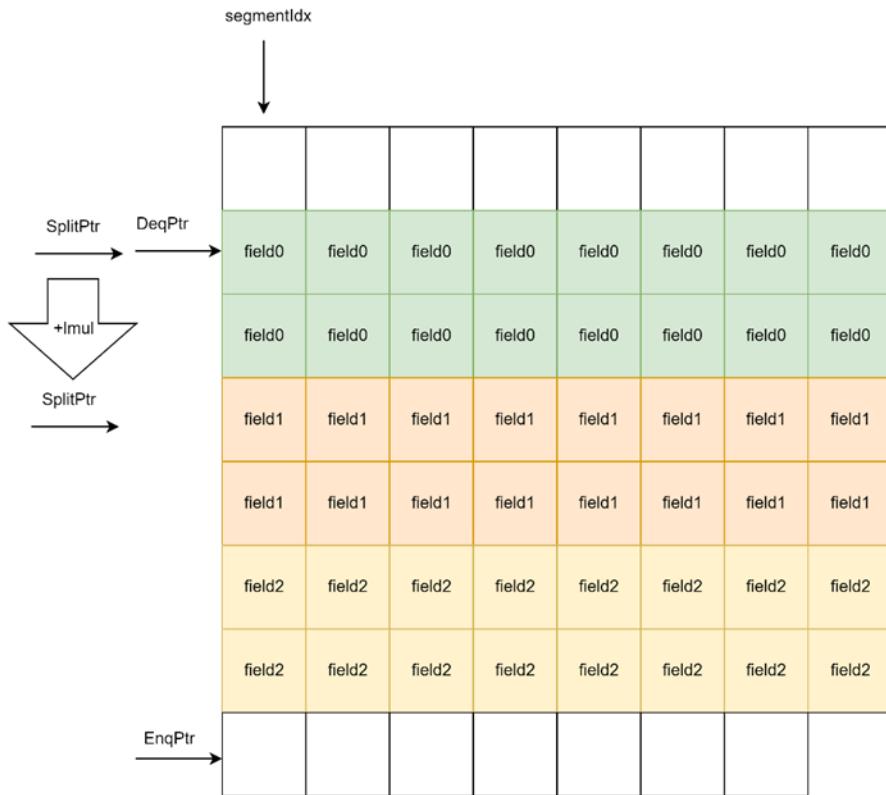


图 16.14: alt text

16.6.6.1.2 特性 2: fault only first 修改 VL 寄存器的 uop 单独写回

对于 fault only first 指令, VSegmentUnit 不使用 VfofBuffer 进行写回额外的 uop。而是自己转进到 s_fof_fix_vl 写回修改 VL 寄存器的 uop。

16.6.6.1.3 特性 3: 支持 Segment 的非对齐访存

VSegmentUnit 指令自己单独执行非对齐访存, 无需借助 MisalignBuffer。由 VSegmentUnit 自身进行非对齐指令的拆分与数据的合并。

16.6.6.2 状态转换图

状态介绍

| 状态 | 说明 |
|---------------------------|----------------------------|
| s_idle | 等待 SegmentUnit uop 进入 |
| s_flush_sbuffer_req | flush sbuffer |
| s_wait_flush_sbuffer_resp | 等待 Sbuffer 和 StoreQueue 为空 |
| s_tlb_req | 查询 DTLB |
| s_wait_tlb_resp | 等待 DTLB 响应 |
| s_pm | 检查执行权限 |
| s_cache_req | 请求读取 DCache |

| 状态 | 说明 |
|------------------------|--|
| s_cache_resp | DCache 响应 |
| s_misalign_merge_data | 合并非对齐的 Load Data |
| s_latch_and_merge_data | 将每个元素的 Data 合并成完整的 uop 粒度的 Data |
| s_send_data | 发送数据至 Sbuffer |
| s_wait_to_sbuffer | 等待发送至 Sbuffer 的流水级清空, 即真正的发送到 Sbuffer |
| s_finish | 该指令执行完成, 开始以 uop 为粒度写回至后端 |
| s_fof_fix_vl | fault only first 指令数据 uop 已经写回, 写回修改 VL 寄存器的 uop |

16.6.6.3 译码实例

16.6.6.3.1 Segment Unit-Stride/Stride

unit-stride 按照 $\text{stride} = \text{eew} * \text{nf}$ 的 stride 指令处理。这一类指令用到的偏移量寄存器是标量寄存器, uop 数量取决于 data 寄存器的数量, 所以 uop 拆分数量 = emul * nf。比如, emul = 2, nf = 4, 则 uop 编号如下: uopIdx = 0, 基地址 rs1, 步长 rs2, 目的寄存器 vd uopIdx = 1, 基地址 rs1, 步长 rs2, 目的寄存器 vd+1 uopIdx = 2, 基地址 rs1, 步长 rs2, 目的寄存器 vd+2 uopIdx = 7, 基地址 rs1, 步长 rs2, 目的寄存器 vd+7

16.6.6.3.2 Segment Index

- 拆分数量为: Max (lmul*nf, emul), 需要保证从第一个 field 的寄存器组按序开始拆分。
- 例如: emul=4, lmul=2, nf=2, uop 拆分如下:
 - uopidx=0, 基地址 src, 偏移量 vs2, 目的寄存器 vd
 - uopidx=1, 基地址 (dontCare), 偏移量 vs2+1, 目的寄存器 vd+1
 - uopidx=2, 基地址 (dontCare), 偏移量 vs2+2, 目的寄存器 vd+2
 - uopidx=3, 基地址 (dontCare), 偏移量 vs2+3, 目的寄存器 vd+3
- 再例如: emul=2, lmul=1, nf=3, uop 拆分如下:
 - uopidx=0, 基地址 src, 偏移量 vs2, 目的寄存器 vd
 - uopidx=1, 基地址 (dontCare), 偏移量 vs2+1, 目的寄存器 vd+1
 - uopidx=2, 基地址 (dontCare), 偏移量 (dontCare), 目的寄存器 vd+2
- 再例如: emul=8, lmul=1, nf=8, uop 拆分如下:
 - uopidx=0, 基地址 src, 偏移量 vs2, 目的寄存器 vd
 - uopidx=1, 基地址 (dontCare), 偏移量 vs2+1, 目的寄存器 vd+1
 - uopidx=2, 基地址 (dontCare), 偏移量 vs2+2, 目的寄存器 vd+2
 - uopidx=3, 基地址 (dontCare), 偏移量 vs2+3, 目的寄存器 vd+3
 - uopidx=4, 基地址 (dontCare), 偏移量 vs2+4, 目的寄存器 vd+4
 - uopidx=5, 基地址 (dontCare), 偏移量 vs2+5, 目的寄存器 vd+5
 - uopidx=6, 基地址 (dontCare), 偏移量 vs2+6, 目的寄存器 vd+6
 - uopidx=7, 基地址 (dontCare), 偏移量 vs2+7, 目的寄存器 vd+7

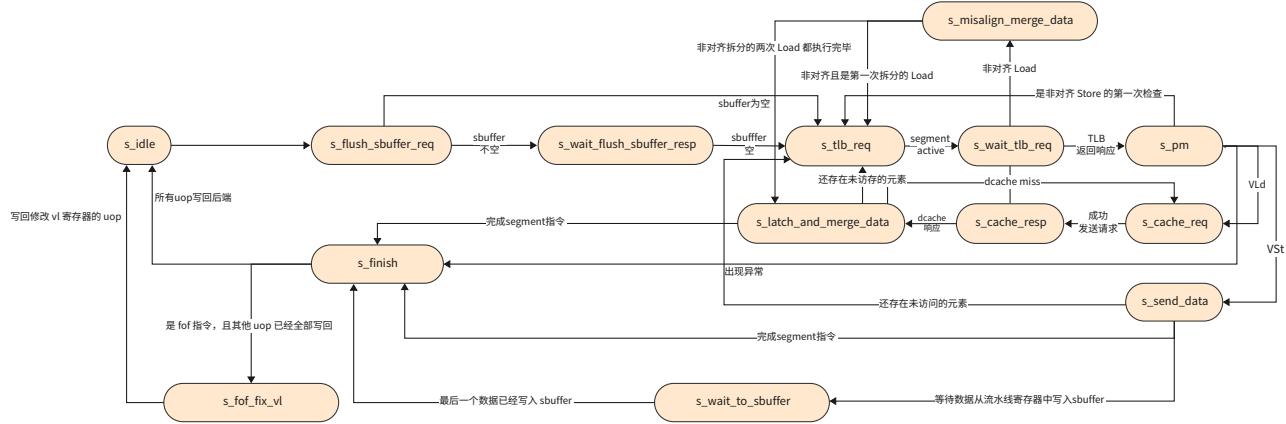


图 16.15: alt text

16.6.6.4 主要端口

| | 方向 | 说明 |
|-----------------|--------|--|
| in | In | 接收来自 Issue Queue 的 uop 发射 |
| uopwriteback | In | 将执行结束的 uop 写回后端 |
| rdcache | In/Out | DCache 请求/响应 |
| sbuffer | Out | 写 Sbuffer 请求 |
| vecDiffTestInfo | Out | Sbuffer 中 DiffTestStoreEvent 所需信息 |
| dtilb | In/out | 读写 DTLB 请求/响应 |
| pmpResp | In | 接收来自 PMP 的访问权限信息 |
| flush_sbuffer | Out | 冲刷 Sbuffer 请求 |
| feedback | Out | 反馈至 Issue Queue 模块 |
| redirect | In | 重定向端口 |
| exceptionInfo | Out | 输出 Exception 信息，参与 MemBlock 中写回异常信息的仲裁 |
| fromCsrTrigger | In | 接收来自 CSR 的 Trigger 相关数据 |

16.6.6.5 接口时序

接口时序较简单，只提供文字描述。|| 说明 || -----: | :----- | | in | 具备 Valid、Ready。数据同 Valid && ready 有效 || uopwriteback | 具备 Valid、Ready。数据同 Valid && ready 有效 || rdcache | 具备 Valid、Ready。数据同 Valid && ready 有效 || sbuffer | 具备 Valid、Ready。数据同 Valid && ready 有效 || vecDiffTestInfo | 与 sbuffer 端口同时有效 || dtilb | 具备 Valid、Ready。数据同 Valid && ready 有效 || pmpResp | 具备 Valid、Ready。数据同有效 || flush_sbuffer | 具备 Valid。数据同 Valid 有效 || feedback | 具备 Valid。数据同 Valid 有效 || redirect | 具备 Valid。数据同 Valid 有效 || exceptionInfo | 具备 Valid 有效 || fromCsrTrigger | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |

16.6.7 向量 FOF 指令单元 VfofBuffer

16.6.7.1 功能描述

处理并写回向量 Fault only frist (fof) 指令的修改 VL 寄存器的 uop。对于 fof 指令，我们会额外拆分出单独的负责修改 VL 寄存器的 uop。目前，我们对 fof 指令采取非推测执行的方式。

16.6.7.1.1 特性 1：收集访存 uop 写回信息

VfofBuffer 负责收集 fof 指令访存 uop 的写回信息，且只有一项。如果需要更新 VL 寄存器，则更新 VfofBuffer 中维护的信息。当一条 Fault only frist 指令被发射时，除了正常的进入 VLSplit 之外，还会在 vfofBuffer 中分配一项。这一项会监听来自 VLMergeBuffer 的同 RobIdx 的 uop 的写回，并不会阻止这些 uop 写回后端，只是会收集这些 uop 的相关元数据来更新维护自身的 VL。VLMergeBuffer 写回至后端的 uop 会携带异常信息与 VL 等，我们需要根据这些写回的信息来判断这个 uop 是否应该导致 VL 发生变化，如果需要 VL 发生变化，则与 VfofBuffer 中维护的 VL 比较，更新为更小的 VL。

16.6.7.1.2 特性 2：写回修改 VL 寄存器的 uop

VfofBuffer 会在该指令的所有访存 uop 写回之后，再写回修改 VL 寄存器的 uop。即使不需要修改 VL 寄存器，该 uop 依然会写回，只是不会使能写入信号。

16.6.7.2 整体框图

单一模块无框图。

16.6.7.3 主要端口

| | 方向 | 说明 |
|-------------------|-----|----------------------------|
| redirect | In | 重定向端口 |
| in | In | 接收来自 Issue Queue 的 uop 发射 |
| mergeUopWriteback | In | 接收 VLMergeBuffer 写回的数据 uop |
| uopWriteback | Out | 写回修改 VL 的 uop 至后端 |

16.6.7.4 接口时序

接口时序较简单，只提供文字描述。

| | 说明 |
|-------------------|--------------------------------------|
| redirect | 具备 Valid。数据同 Valid 有效 |
| in | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| mergeUopWriteback | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| uopWriteback | 具备 Valid、Ready。数据同 Valid && ready 有效 |

16.7 访存队列

16.7.1 访存队列 LSQ

16.7.1.1 子模块列表

| 子模块 | 描述 |
|---------------------|------|
| VirtualLoadQueue | TODO |
| LoadQueueRAR | TODO |
| LoadQueueRAW | TODO |
| LoadQueueReplay | DONE |
| LoadQueueUncache | TODO |
| LoadExceptionBuffer | TODO |
| StoreQueue | DONE |

16.7.1.2 功能描述

LSQ 包括了 LoadQueue 和 StoreQueue 两个部分，并做了一层 wrapper，便于端口的连接。Lsqwrapper 的作用主要只是连线。

- LoadQueue
 - LoadQueueRAR: RAR 违例检查队列
 - LoadQueueRAW: RAW 违例检查队列
 - LoadQueueUncache: MMIO/Noncacheable load 指令处理队列
 - LoadQueueReplay: Load 指令调度重发队列
 - LoadExceptionBuffer: Load 指令异常处理队列
 - VirtualLoadQueue: Load 指令顺序维护队列
- StoreQueue

16.7.1.2.0.1 特性 1：更新 Load 指令的 LqPtr 和 Store 指令的 SqPtr

- 由于时序的影响，LqPtr 和 SqPtr 的分配被拆分为两部分，如图
 - Dispatch 阶段
 - * 统计每条指令的 LoadFlow 或者 StoreFlow 数，并以累加的方式计算出 LqPtr 或者 SqPtr
 - LSQ 入队阶段
 - * 根据 LoadQueue 或者 StoreQueue 维护的 enqPtr 以累加的方式计算出准确的 LqPtr 或者 SqPtr
 - LsqEnqCtrl 更新逻辑
 - * 如果出现刷新流水线，则根据刷新 Load 或者 Store 指令数和 commit 数更新
 - * 如果没有出现刷新流水线，但是有分配的请求，根据 enq 和 commit 数更新
 - * 否则，根据 commit 数更新

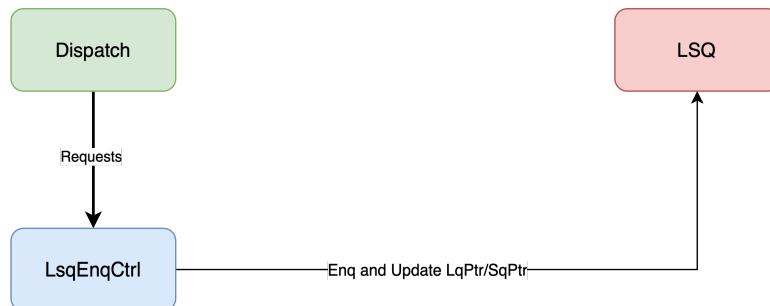


图 16.16: LSQ 分配

16.7.1.3 整体框图

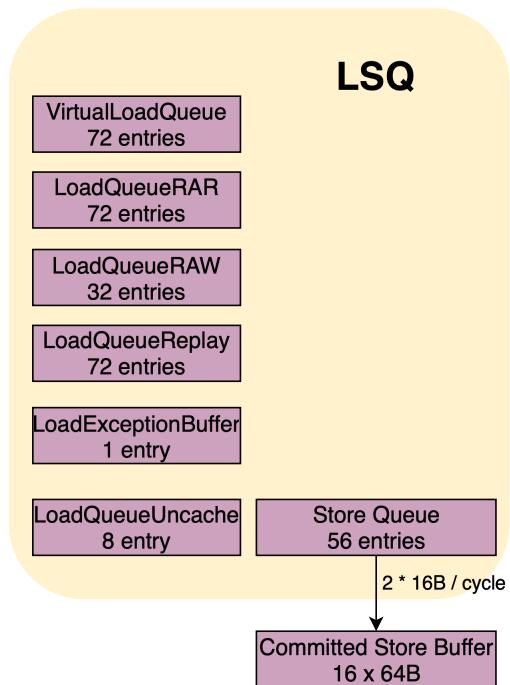


图 16.17: LSQ 整体框架

16.7.1.4 接口时序

16.7.1.4.1 Load 指令和 Store 指令入队接口时序实例

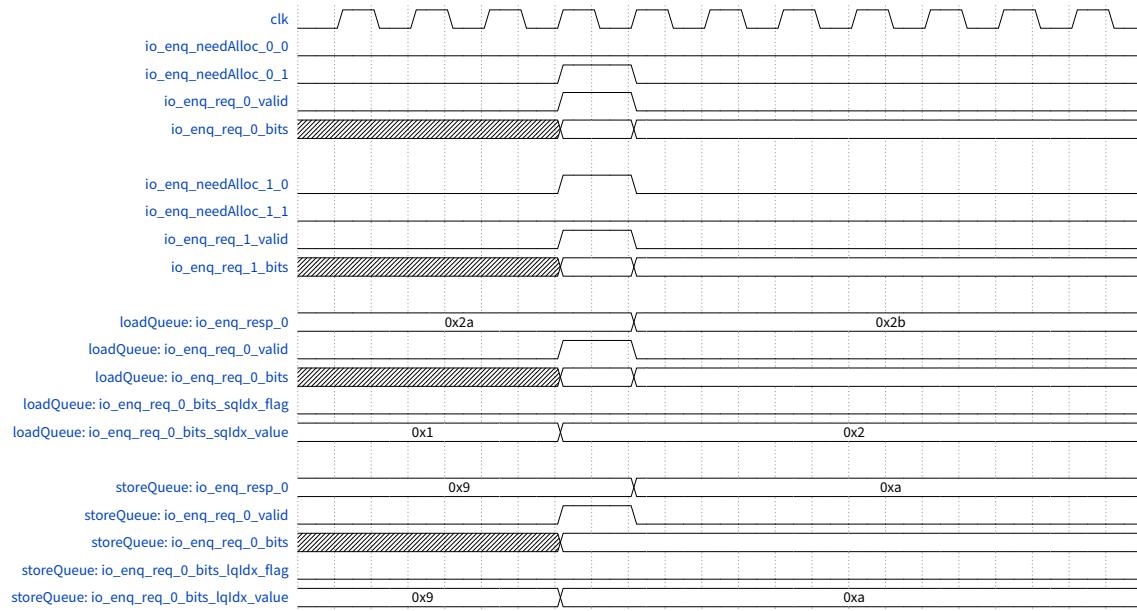


图 16.18: 入队更新

16.7.2 Load 队列 VirtualLoadQueue

16.7.2.1 功能描述

Virtualloadqueue 是一个队列，用于存储所有 load 指令的 MicroOp，维护 load 指令之间的顺序，类似于 load 指令的 ROB，其主要功能为跟踪 Load 指令执行状态。

Virtualloadqueue 对于每一个 entry 中的 load 指令都有若干状态位来标识这个 load 处于什么状态：

- allocated: 该项是否分配了 load，用于确定 load 指令的生命周期。
- isvec: 该指令是否是向量 load 指令。
- committed: 该项是否提交。

16.7.2.1.1 特性 1：入队

- 入队时机：在指令的 dispatch 阶段，会将 load 指令从 dispatch queue 发送到 load queue，Virtual Load Queue 用于保存指令的信息。
- 流水线写回时机：load 从 iq 发出后，经过 load 流水线，到达流水线的 s3 时，将这条 load 的执行信息反馈给 load queue。
- 流水线写回的信息：包括 dcache 是否命中，load 是否正常拿到了数据（包括 dcache miss 但是可以从 sbuffer 和 store queue forward 完整数据的情况），tlb 是否 miss，是否需要重发 load。load 是否发生了异常，load 是否是 MMIO 空间的，是否是向量 load，是否产生写后读违例、读后读违例，是否出现 dcache 的 bank 冲突。

16.7.2.1.2 特性 2：出队

- 出队时机：当被分配的 entries (allocated 为高) 到达队头，同时 allocated 与 committed 都为 1 时，表示可以出队，如果是向量 load，需要每个元素都 committed。

16.7.2.2 整体框图

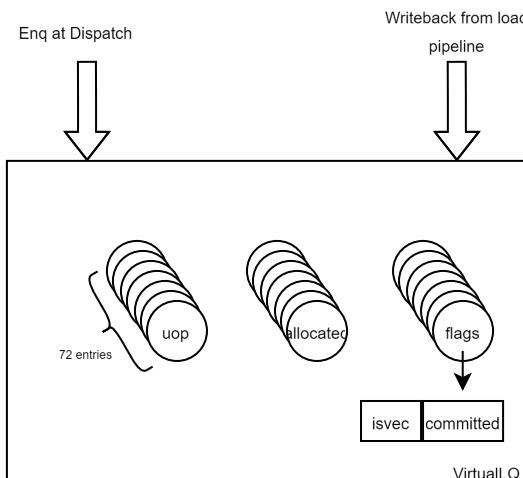


图 16.19: VirtualLoadQueue 整体框图

16.7.2.3 接口时序

16.7.2.3.1 接收入队请求时序实例

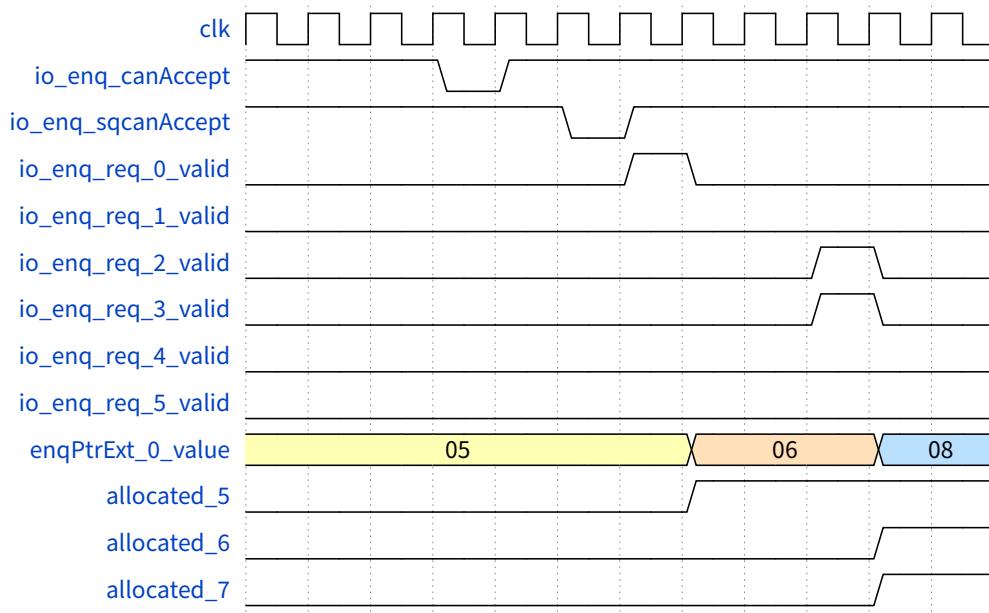


图 16.20: VirtualLoadQueue-enqueue

当 `io_enq_canAccept` 与 `io_enq_sqcanAccept` 为高时，表示可以接收派遣指令。当 `io_enq_req_*_valid` 为高时表示真实派遣指令到 VirtualLoadQueue，派遣指令的信息为 rob 的位置、Virtualloadqueue 的位置以及向量指令元素个数等。完成派遣后对应的 `allocated` 拉高，`enqPtrExt` 根据派遣的 req 个数更新。

16.7.2.3.2 流水线 writeback 时序实例

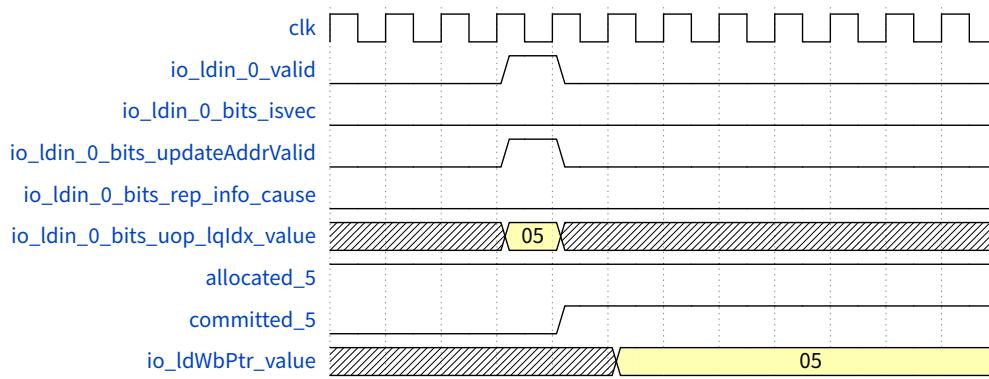


图 16.21: VirtualLoadQueue-writeback

当 `io_ldin_* valid` 为高时表示 load 流水线的 s3 写回 lq，具体内容为 `io_ldin* bits*`。`allocated_5` 表示 lq 的第 5 项是否分配，当 `updateAddrValid`，且没有 `replay` 时，`committed_5` 在下一拍拉高。`allocated` 和 `committed` 同时为高表示可以出队。每写回一个表项队尾指针 +1。

16.7.3 读后读违例检查 LoadQueueRAR

16.7.3.1 功能描述

多核环境下会出现 load to load 违例。单核环境下相同地址的 load 乱序执行本来是不关心的，但是如果两个 load 之间有另外一个核做了相同地址的 store，并且本身这个核的两个 load 做了乱序调度，就有可能导致新的 load 没有看到 store 更新的结果，但是旧的 load 看到了，出现了顺序错误。

多核环境下的 load-load 违例有一个特征，当前 DCache 一定会收到 L2 cache 发来的 Probe 请求，使得 DCache 主动释放掉这个数据副本，这时 DCache 会通知 load queue，将相同地址的 load queue 中已经完成访存的项做一个 release 标记。后续发往流水线的 load 指令会查询 load queue 中在它之后相同地址的 load 指令，如果存在 release 标记，就发生了 load-load 违例。

LoadQueueRAR 用于保存已经完成的 load 指令的用于 load to load 违例检测的信息。当 load 指令处于 load 流水线 s2 栈时，查询并分配空闲项将信息保存入 LQRAR，在流水线 s3 栈时得到 load to load 违例检查的结果，如果出现违例则需要刷新流水线，给 RedirectGenerator 部件发送重定向请求，冲刷违例的 load 之后的所有指令。

LoadQueueRAR 中需要标记以下信息：

- Allocated: 表示 entry 是否有效。
- Uop: MicroOp 相关信息。
- Paddr: 进入 LoadQueueRAR 指令的物理地址压缩后的地址，一共 16bits。
- Released: 表示该指令所访问的 cacheline 是否被 release，多核环境下 L1 cache 会接收到 L2cache 的 probe 请求。需要注意的是如果该指令是 nc 指令，在入队时就会被标记 release。

16.7.3.1.1 特性 1：请求入队

当 query 到达 load 流水线的 s2 时，判断是否满足入队条件，如果在当前 load 指令之前有未完成的 load 指令，且当前指令没有被 flush 时，当前 load 可以入队。

在 freelist 中得到可以分配的 entry 以及 index。

在 PaddrModule 中保存入队信息，包含 query 压缩后的物理地址（16bits），分配 entry 的 index。

16.7.3.1.2 特性 2：load to load 违例检查

当 load 到达流水线的 s2 时，会检查 RAR 队列中是否存在与当前 load 指令物理地址相同且比当前指令年轻的 load 指令，如果这些 load 已经拿到了数据，并且被标记了 release，说明发生 load-load 违例，需要刷新当前发生违例 load 之后的所有指令。

一共分两拍：

- 第一拍进行条件匹配，得到 mask。
- 第二拍生成是否发生违例的响应信号。

16.7.3.1.3 特性 3：Release 条件

LoadQueueRAR 中的 load 指令被标记为 release 有四种情况：

- missQueue 模块的 replace_req 在 mainpipe 流水线的 s3 栈发起 release 释放 dcache 块，release 信号在下一拍进入 loadQueue。

- probeQueue 模块的 probe_req 在 mainpipe 流水线的 s3 栈发起 release 释放 dcache 块, release 信号在下一拍进入 loadQueue。
- atomicsUnit 模块的请求在 mainpipe 流水线的 s3 栈发生 miss 时需要释放 dcache 块, release 信号在下一拍进入 loadQueue。
- 如果入队请求是 nc, 在入队时被标记 release。

16.7.3.2 整体框图

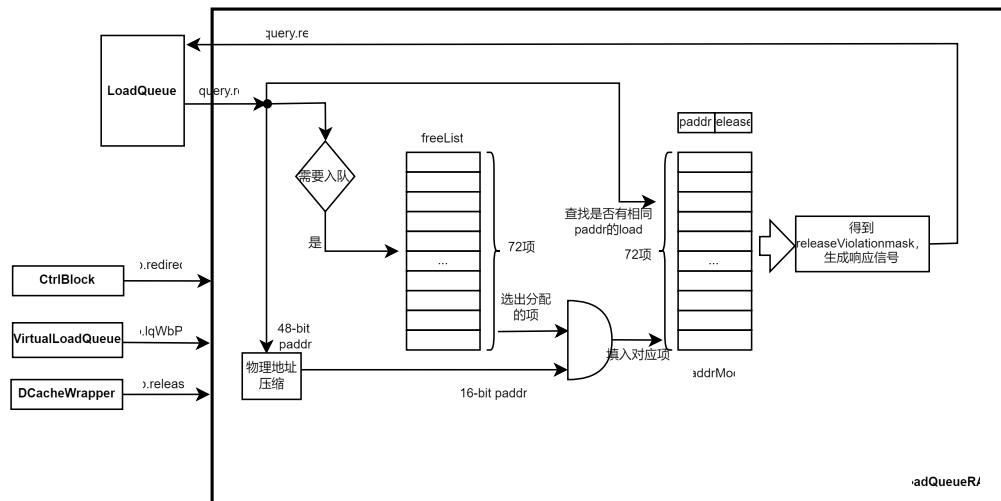


图 16.22: LoadQueueRAR 整体框图

16.7.3.3 接口时序

16.7.3.3.1 LoadQueueRAR 请求入队时序实例

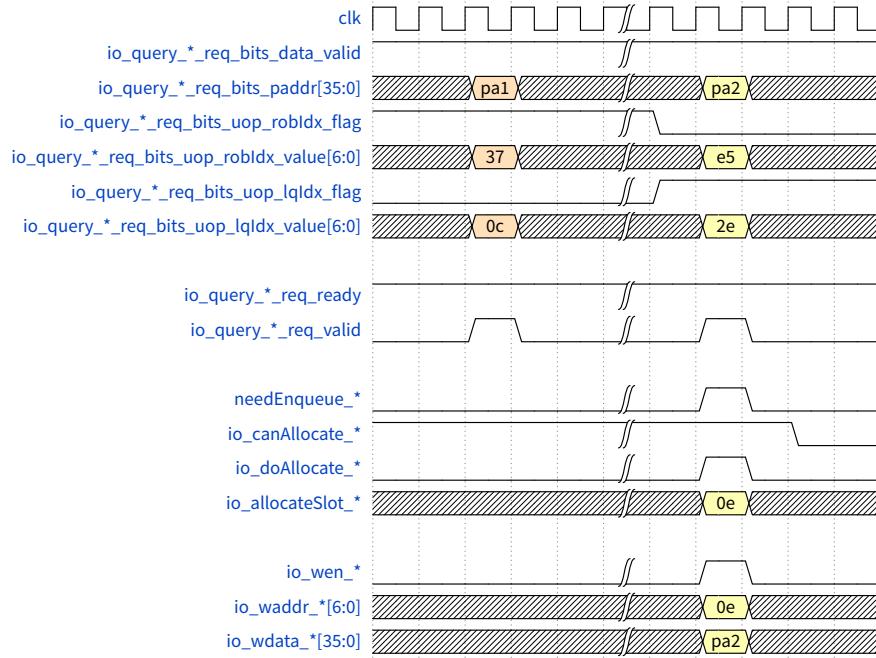


图 16.23: LoadQueueRAR 请求入队时序

当 `io_query_*_req_valid` 和 `io_query_*_req_ready` 都为高时，表示握手成功，`needEnqueue` 和 `io_canAllocate*` 都为高时，将 `io_doAllocate_*` 置为高，表示 query 需要入队且 FreeList 可以分配，`io_allocateSlot_*` 为接收 query 入队的 entry，写入 entry 的信息为 `io_w*`。

16.7.3.3.2 load-load 违例检查时序实例

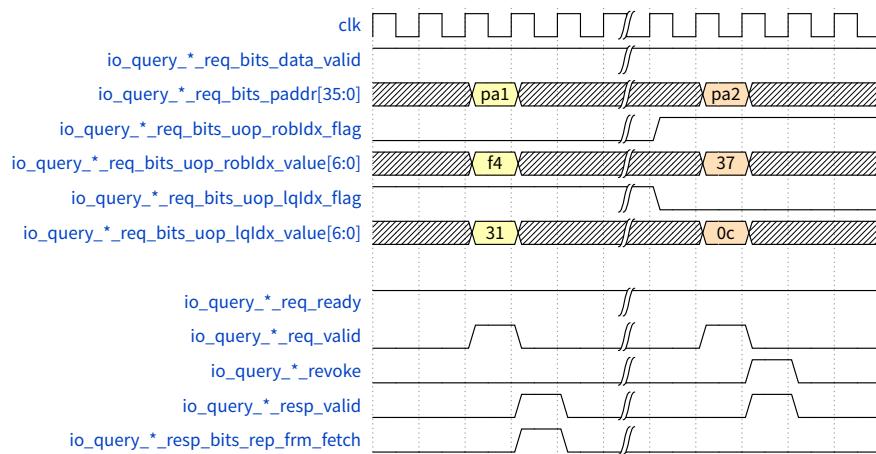


图 16.24: load-load 违例检查时序

当 `io_query_*_req_valid` 和 `io_query_*_req_ready` 都为高时，表示握手成功，LoadQueueRAR 接收 ld-ld 违例查询请求，当拍得到 mask 结果，在下一拍将 `io_query_*_resp_valid` 置为高，给出响应。

图中第 3 拍接收到第一个违例查询请求，在第 4 拍得到违例查询请求的响应。请求的信息为 `io_query_*`

req_bits,响应的信息为 io_query_* resp_bits。当 io_query_* resp_valid 和 io_query*_resp_bits_rep_frm_fetch 都为高时, 表示发生 ld-ld 违例, 刷新当前违例 load 之后的所有指令。

16.7.4 写后读违例检查 LoadQueueRAW

16.7.4.1 功能描述

LoadQueueRAW 是用于处理 store-load 违例的。由于 load 和 store 在流水线中都是乱序执行，会经常出现 load 越过了更老的相同地址的 store，即这条 load 本应该前递 store 的数据，但是由于 store 地址或者数据没有准备好，导致这条 load 没有前递到 store 的数据就已经提交，后续使用这条 load 结果的指令也都发生了错误，于是产生 store to load forwarding 违例。

当 store address 通过 STA 保留站发射出来进入 store 流水线时，会去查询 LQRAW 中在这条 store 后面的所有已经完成访存的相同地址的 load，以及 load 流水线中正在进行的在该条 store 之后的相同地址的 load，一旦发现有，就发生了 store to load forwarding 违例，可能有多个 load 发生了违例，需要找到离 store 最近的 load，也就是最老的违例的 load，然后给 RedirectGenerator 部件发送重定向请求，冲刷最老的违例的 load 及之后的所有指令。

当 store 流水线执行 cbo zero 指令时，也需要进行 store-load 违例检查。

16.7.4.1.1 特性 1：load query 入队

当 query 到达 load 流水线的 s2 时，判断是否满足入队条件，如果在当前 load 指令之前有地址未准备好的 store 指令，且当前指令没有被 flush 时，当前 load 可以入队。

在 freelist 中得到可以分配的 entry 以及 index。

将入队 query 的物理地址压缩为 24-bit 保存到 PaddrModule 中对应的 entry。

将入队 query 的 mask 保存到 maskModule 中对应的 entry。

16.7.4.1.2 特性 2：store-load 违例检查

store 指令到达 store 流水线的 s1 时会进行 store-load 检查，此时 store 需要与 LoadQueueRAW 中已经完成访存的 load，以及 load 流水线中 s1 和 s2 阶段正在访存的 load 作比较，这些 load 可能没有 forward 到 store 的数据。如果检查时发现 load 和 store 访问的物理地址有重叠的地方，且 load 比 store 年轻，就发生了违例，需要找到最老的 load，重发这条 load 以及之后的所有指令（重新取指执行），在 store 流水线的 s4 阶段得到 store-load 违例检查的结果。

一共分四拍：

- 第一拍进行物理地址匹配，条件匹配，得到 mask，匹配的是那些在这条 store 之后的新的 load，如果它们已经拿到了数据 (datavalid) 或者 dcache miss 了，正在等待 refill (miss)，就一定没有 forward 到这个 store 的数据。
- 第二拍 store 流水线中的 store 根据 mask 在 LoadQueueRAW 里面找到所有匹配的 load，LoadQueueRAW 一共有 32 项，将这 32 项平分为八组，从每组的 4 项里面各选出一个 oldest，最多可能得到 4 个 oldest。
- 第三拍从 4 个 oldest 里面选出一个最老的 oldest。
- 第四拍如果两条 store 流水线中的 store 都发生了 store-to-load 违例，从两条 store 流水线各自在 load-Queue 匹配的 oldest load 中选出一个更老的 oldest，产生回滚请求发给 redirect。

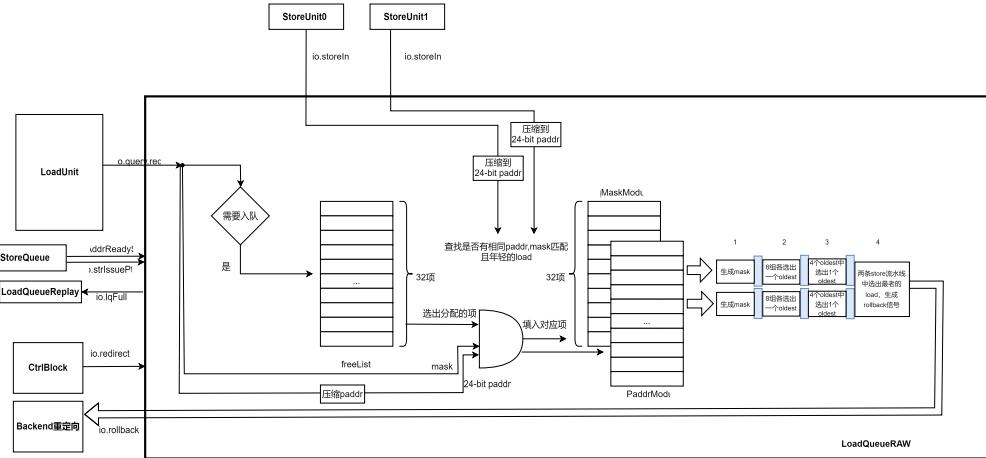


图 16.25: LoadQueueRAW 整体框图

16.7.4.2 整体框图

16.7.4.3 接口时序

16.7.4.3.1 LoadQueueRAW 请求入队时序实例

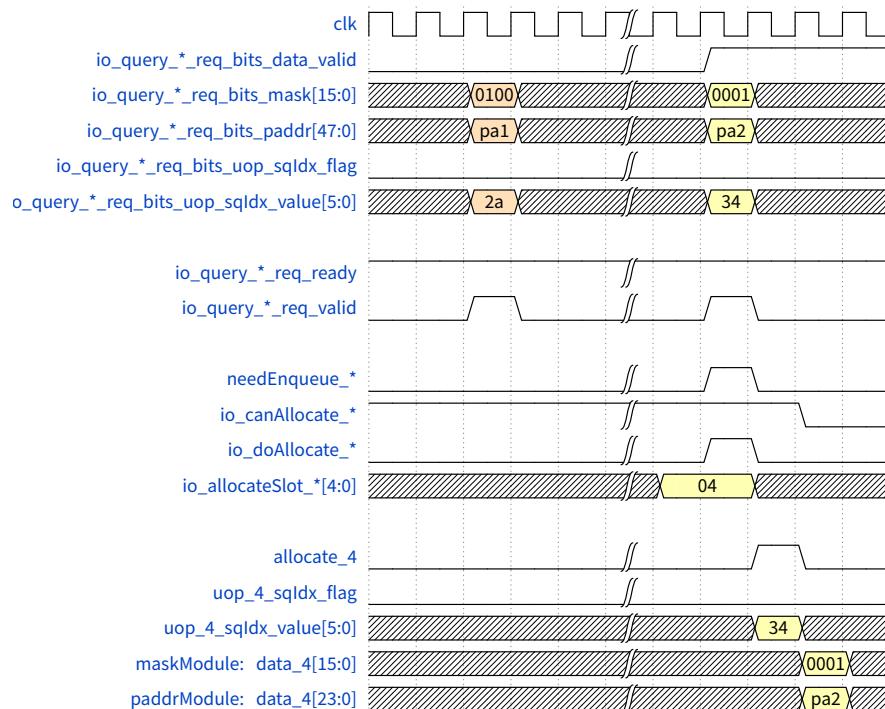


图 16.26: LoadQueueRAW 请求入队时序

当 `io_query_* req_valid` 和 `io_query* req_ready` 都为高时，表示握手成功，`needEnqueue` 和 `io_canAllocate*` 都为高时，将 `io_doAllocate_*` 置为高，表示 query 需要入队且 FreeList 可以分配，`io_allocateSlot_*` 为接收 query 入队的 entry，下一拍对应 entry 的 `allocate` 拉高，`sqIdx` 写入 entry。再下一拍后 `mask` 写入 `LqMaskModule` 对应的 entry，压缩后的物理地址写入到 `LqPAddrModule` 对应的 entry。

16.7.4.3.2 store-load 违例时序实例

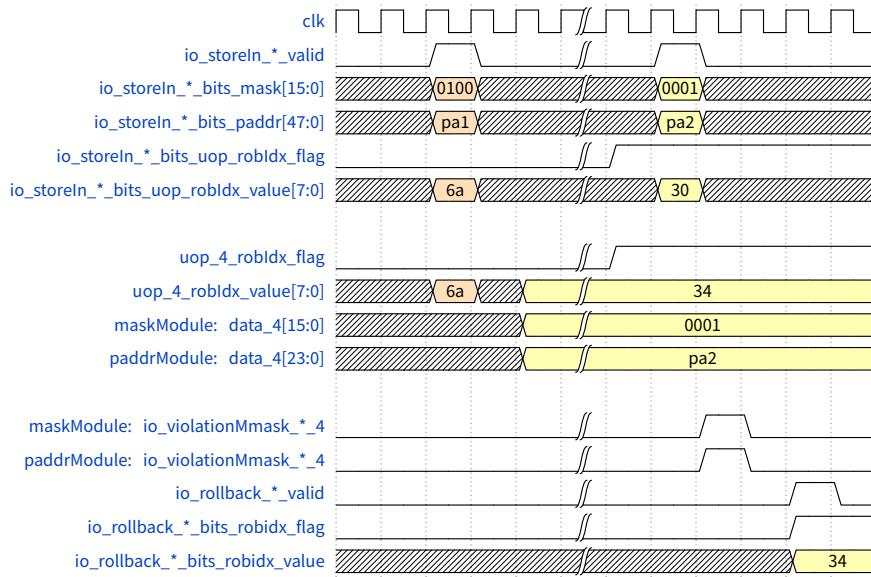


图 16.27: store-load 违例检查时序

当 `io_rollback_valid` 为高时，表示发生了 store-load 违例，违例的信息为 `io_rollback_bits_*`。

16.7.5 Load 重发队列 LoadQueueReplay

16.7.5.1 功能描述

`LoadQueueReplay` 用于存储需要重发的 Load 指令，并根据不同的唤醒条件唤醒指令，调度指令进入 Load-Unit 执行，主要包括以下几个状态和存储的信息：

表 16.19: LoadQueueReplay 存储信息

| Field | 描述 |
|-------------|--|
| allocated | 是否已经被分配，也代表是否该项是否有效。 |
| scheduled | 是否已经被调度，代表该项已经被选出，已经或即将被发送至 LoadUnit 进行重发。 |
| uop | load 指令执行包括的 uop 信息。 |
| vecReplay | 向量 load 指令相关信息 |
| vaddrModule | Load 指令的虚拟地址 |
| cause | 某 load replay queue 项对应 load 指令重发的原因，包括： C_MA(位 0): store-load 预测违例 C_TM(位 1): tlb miss C_FF(位 2): store-to-load-forwarding store 数据为准备好，导致失败 C_DR(位 3): 出现 DCache miss，但是无法分配 MSHR C_DM(位 4): 出现 DCache miss C_WF(位 5): 路预测器预测错误 C_BC(位 6): Bank 冲突 C_RAR(位 7): LoadQueueRAR 没有空间接受指令 |

| Field | 描述 |
|--------------------|---|
| | C_RAR(位 8): LoadQueueRAW 没有空间接受指令 |
| | C_NK(位 9): LoadUnit 监测到 store-to-load-forwarding 违例 |
| | C_MF(位 10): LoadMisalignBuffer 没用空间接受指令 |
| blocking | Load 指令正在被阻塞 |
| strict | 访存依赖预测器判断指令是否需要等待它之前的所有 store 指令执行完毕进入调度阶段 |
| blockSqIdx | 与 load 指令有相关性的 store 指令的 StoreQueue Index |
| missMSHRId | load 指令的 dcache miss 请求接受 ID |
| tlbHintId | load 指令的 tlb miss 请求接受 ID |
| replacementUpdated | DCCahe 的替换算法是否已经更新 |
| replayCarry | DCache 的路预测器预测信息 |
| missDbUpdated | ChiselDB 中 Miss 相关情况更新 |
| dataInLastBeatReg | Load 指令需要的数据在两笔回填请求的最后一笔 |

16.7.5.1.1 特性 1：乱序分配

- LoadUnit S3 传入一条 load 请求后, 首先需要判断是否需要入队。如果不需要重发、发生异常或因 redirect 被冲刷, 均不需要入队。LoadQueueReplay 通过 freelist 进行队列空闲管理。Freelist 的大小为 load replay queue 的项数, 分配宽度为 Load 的宽度 (LoadUnit 的数量), 释放宽度为 4。同时, freelist 可以反馈 load replay queue 的空余项, 以及是否满的信息。LoadQueueReplay 采用 Freelist 进行队列空闲管理。Freelist 的大小为 LoadQueueReplay 的项数, 分配宽度为 Load 的宽度 (LoadUnit 的数量), 释放宽度为 4。

- 分配

- * LoadQueueReplay 从 Freelist 中从空闲的项中 (即图16.28中的 Valid 项), 为每一个 LoadUnit 选出一个项索引 (尽力而为选出空闲项, 例如, 有效项有 5, 10 两项, LoadUnit0 和 LoadUnit2 有效, 则 LoadUnit0 分配到 5, LoadUnit2 分配到 10), 之后根据索引将指令信息填入对应的索引项中。

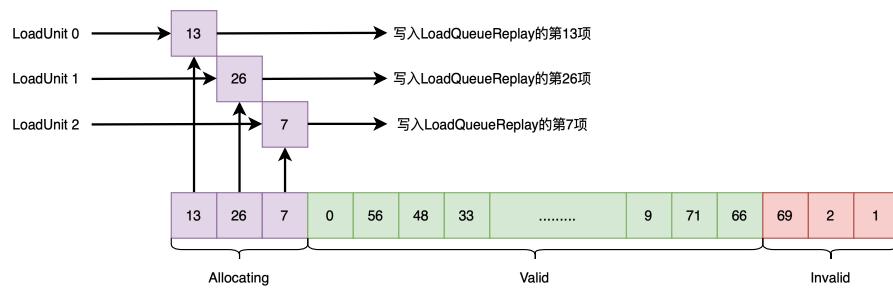


图 16.28: Freelist

- 回收

- * 成功重发或者被刷新的 load 指令占用的项, 需要回收。LoadQueueReplay 通过使用一个位图 FreeMask 保存正在释放项, , 每个周期 Freelist 最多回收 4 项。

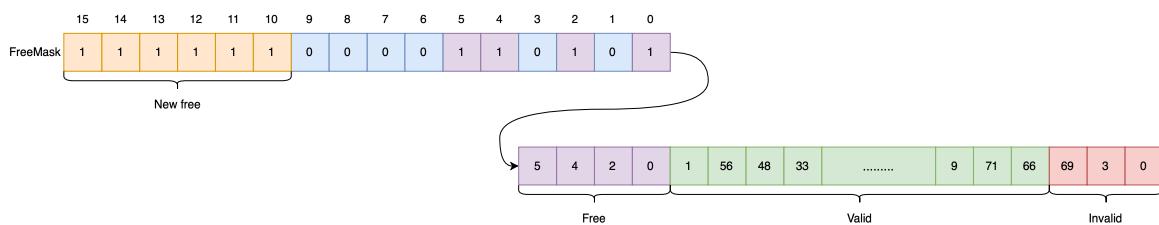


图 16.29: Freelist 回收

16.7.5.1.2 特性 2：唤醒

- 不同的阻塞条件, 有不同的唤醒的条件:

- C_MA: 如果 strict==1, 则需要等待 load 指令之前的所有 store 指令地址计算完成之后才能唤醒, 否则只需要等待 blockSqIdx 对应的 Store 指令的地址计算完成之后唤醒。
- C_TM: 如果 TLB 没有多余空间处理 miss 请求, 则可以标记为可重发状态, 等待调度; 否则需要等待 TLB 返回 tlbHintId 匹配的 hint 信号唤醒。

- C_FF: 需要等待 blockSqIdx 对应的 Store 指令的数据准备之后唤醒。
- C_DR: 可以标记为可重发状态，等待调度。
- C_DM: 等待与 missMSHRIId 匹配的 L2 Hint 信号唤醒。
- C_WF: 可以标记为可重发状态，等待调度。
- C_BC: 可以标记为可重发状态，等待调度。
- C_RAR: 等待 LoadQueueRAR 有空闲空间或者该条指令是最老的 load 指令时，可以唤醒。
- C_RAW: 等待 LoadQueueRAW 有空闲空间或者该条 load 指令在之前的 store 指令的地址都计算完成之后，可以唤醒。
- C_MF: 等待 LoadMisalignBuffer 有空闲空间，可以唤醒。

16.7.5.1.3 特征 3: 选择调度

· LoadQueueReplay 有 3 种选择调度方式：

- 根据入队年龄
 - * LoadQueueReplay 使用 3 个年龄矩阵 (每一个 Bank 对应一个年龄矩阵)，来记录入队的时间。年龄矩阵会从已经准备好可以重发的指令中，选择一个入队时间最长的指令调度重发。
- 根据 Load 指令的年龄
 - * LoadQueueReplay 可以根据 LqPtr 判断靠近最老的 load 指令重发，判断宽度为 OldestSelectStride=4。
- DCache 数据相关的 load 指令优先调度
 - * LoadQueueReply 首先调度因 L2 Hint 调度的重发 (当 dcache miss 后，需要继续查询下级缓存 L2 Cache。在 L2 Cache 回填前的 2 或 3 拍，L2 Cache 会提前给 LoadQueueReplay 唤醒信号，称为 L2 Hint) 当收到 L2 Hint 后，LoadQueueReplay 可以更早地唤醒这条因 dcache miss 而阻塞的 Load 指令进行重发。
 - * 如果不存在 L2 Hint 情况，会将其余 Load Replay 的原因分为高优先级和低优先级。高优先级包括因 dcache 缺失或 st-ld forward 导致的重发，而将其他原因归纳为低优先级。如果能够从 LoadQueueReplay 中找出一条满足重发条件的 Load 指令 (有效、未被调度、且不被阻塞等待唤醒)，则选择该 Load 指令重发，否则按照入队顺序，通过 AgeDetector 模块寻找一系列 load replay queue 项中最早入队的一项进行重发。

16.7.5.2 整体框图

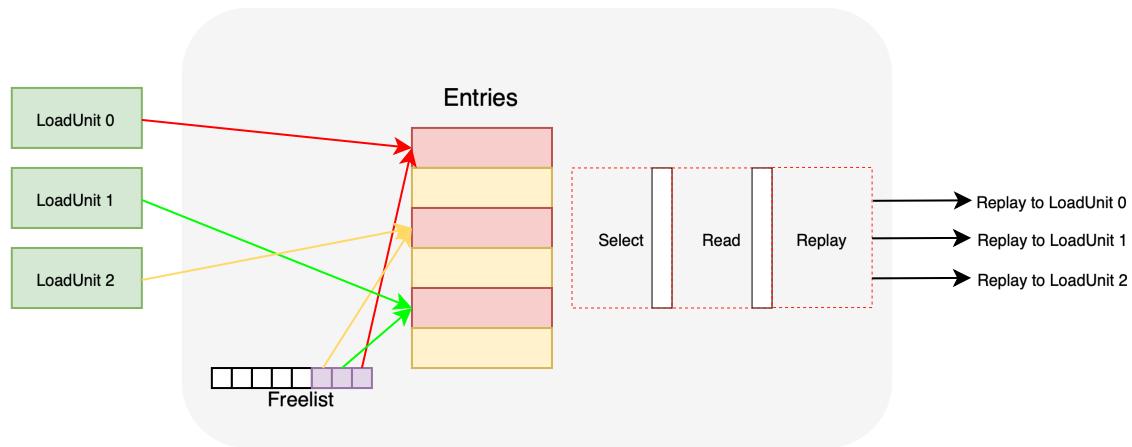


图 16.30: LoadQueueReplay 整体框图

16.7.5.3 接口时序

16.7.5.3.1 入队时序

- 重发入队

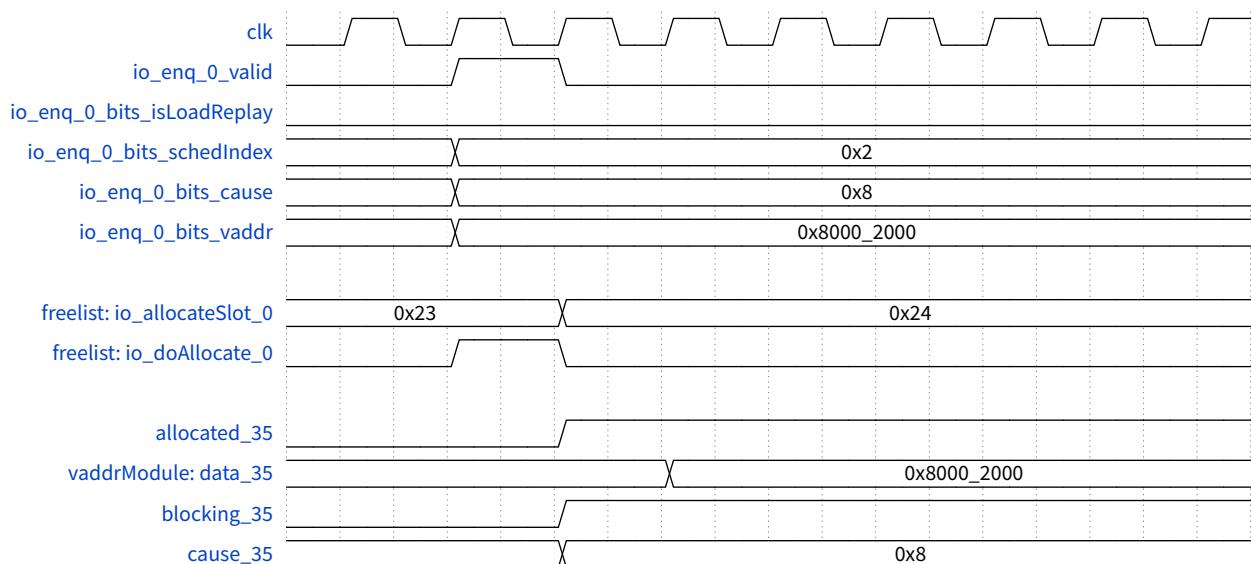


图 16.31: LoadQueueReplay 重发入队时序图

- 非重发入队

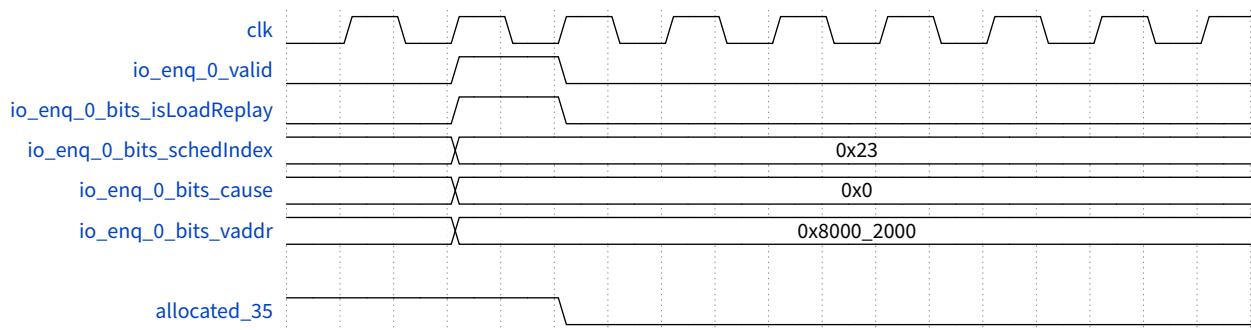


图 16.32: LoadQueueReplay 非重发入队时序图

16.7.5.3.2 重发时序

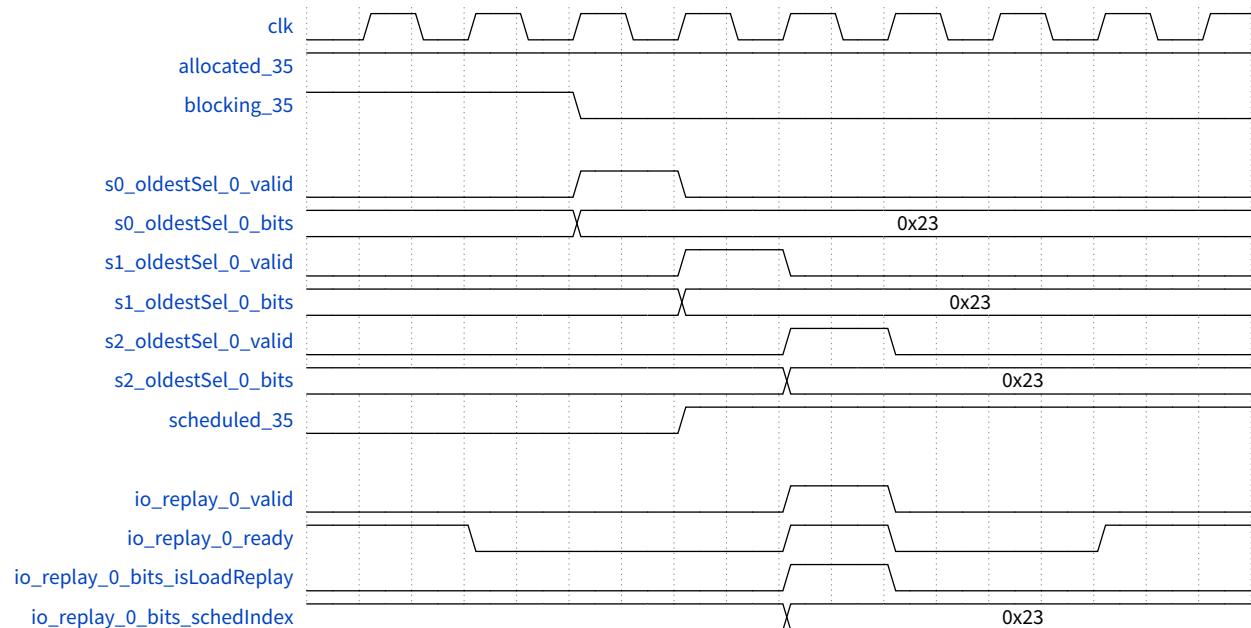


图 16.33: LoadQueueReplay 重发队时序图

16.7.5.3.3 Freelist 时序

- 分配时序

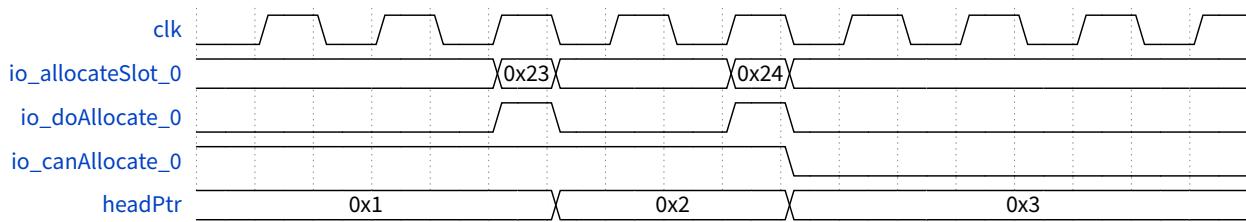


图 16.34: Freelist 分配时序图

- 回收时序

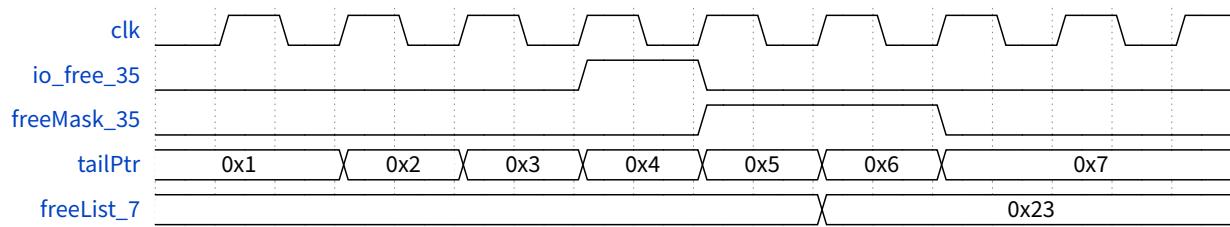


图 16.35: Freelist 回收时序图

16.7.6 Uncache Load 处理单元 LoadQueueUncache

| 更新时间 | 代码版本 | 更新人 | 备注 |
|---------------------|-------------|------|----|
| 2025.02.26 22:09:33 | Maxpicca-Li | 完成初版 | |

16.7.6.1 功能描述

LoadQueueUncache 和 Uncache 模块，对于 uncache load 访问请求来说，起到一个从 LoadUnit 流水线到总线访问的中间站作用。其中 Uncache 模块，作为靠近总线的一方，所起到的作用详见 Uncache。LoadQueueUncache 作为靠近流水线的一方，需要承担以下责任：

1. 接收 LoadUnit 流水线传过来的 uncache load 请求。
2. 选择已准备好 uncache 访问的 uncache load 请求发送到 Uncache Buffer。
3. 接收来自 Uncache Buffer 的处理完的 uncache load 请求。
4. 将处理完的 uncache load 请求返回给 LoadUnit。

LoadQueueUncache 结构上，目前有 4 项（项数可配）UncacheEntry，每一项独立负责一个请求并利用一组状态寄存器控制其具体处理流程；有一个 FreeList，管理各项分配和回收的情况。而 LoadQueueUncache 主要是协同 4 项的新项分配、请求选择、响应分派、出队等统筹逻辑。

16.7.6.1.1 特性 1：入队逻辑

LoadQueueUncache 负责接收来自 LoadUnit 0、1、2 三个模块的请求，这些请求可以是 MMIO 请求，也可以是 NC 请求。首先，系统会根据请求的 robIdx 按照时间顺序（从最老到最新）对请求进行排序，以确保最早的请求能优先分配到空闲项，避免特殊情况下因老项回滚（rollback）而导致死锁。进入处理的条件是：请求没有重发、没有异常，并且系统会根据 FreeList 中可分配的空闲项依次为请求分配项。

当 LoadQueueUncache 达到容量上限，且仍有请求未分配到项时，系统会从这些未分配的请求中选择最早的请求进行 rollback。

16.7.6.1.2 特性 2：出队逻辑

当一个项完成 Uncache 访问操作并返回给 LoadUnit，或被 redirect 刷新时，则该项出队并释放 FreeList 中该项的标志。同一拍可能有多个项出队。返回给 LoadUnit 的请求，会在第一拍中选出，第二拍返回。

其中，可供处理 uncache 返回请求的 LoadUnit 端口是预先设定的。当前，MMIO 只返回到 LoadUnit 2；NC 可返回到 LoadUnit 1\2。在多个端口返回的情况下，利用 uncache entry id 与端口数的余数，来指定每个项可以返回到的 LoadUnit 端口，并从该端口的候选项中选择一个项进行返回。

16.7.6.1.3 特性 3：Uncache 交互逻辑

(1) 发送 req

第一拍先从当前已准备好 uncache 访问中选择一个，第二拍将其发送给 Uncache Buffer。发送的请求中，会标记选中项的 id，称为 mid。其中是否被成功接收，可根据 req.ready 判断。

(2) 接收 idResp

如果发送的请求被 Uncache Buffer 接收，那么会在接收的下一拍收到 Uncache 的 idResp。该响应中，包含 mid 和 Uncache Buffer 为该请求分配 entry id（称为 sid）。LoadQueueUncache 利用 mid 找到内部对应的项，并将 sid 存储在该项中。

(3) 接收 resp

待 Uncache Buffer 完成该请求的总线访问后，会将访问结果返回给 LoadQueueUncache。该响应中，包含 sid。考虑到 Uncache Buffer 的合并特性（详细入队合并逻辑见 Uncache），一个 sid 可能对应 LoadQueueUncache 的多个项。LoadQueueUncache 利用 sid 找到内部所有相关项，并将访问结果传递给这些项。

16.7.6.2 整体框图

16.7.6.3 接口时序

16.7.6.3.1 入队接口时序实例

如下图所示，假设连续 5 个 NC 依次通过 LoadUnit 0\1\2 进入，当前 LoadQueueUncache 只有 4 项。故前四项正常分配现有空项。第 3 拍出现的 r5 因 buffer 满而无法分配项，故在第 5 拍产生回滚。注意，图中假设了每拍 NC 按顺序进入，即 r1 < r2 < r3 且 r4 < r5；如果需要排序，将排序结果依次替换 io_req 即可，其余逻辑一致。

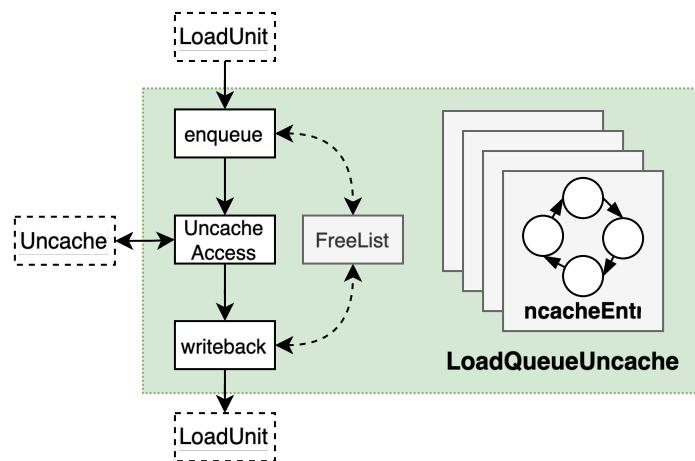


图 16.36: LoadQueueUncache 整体框图

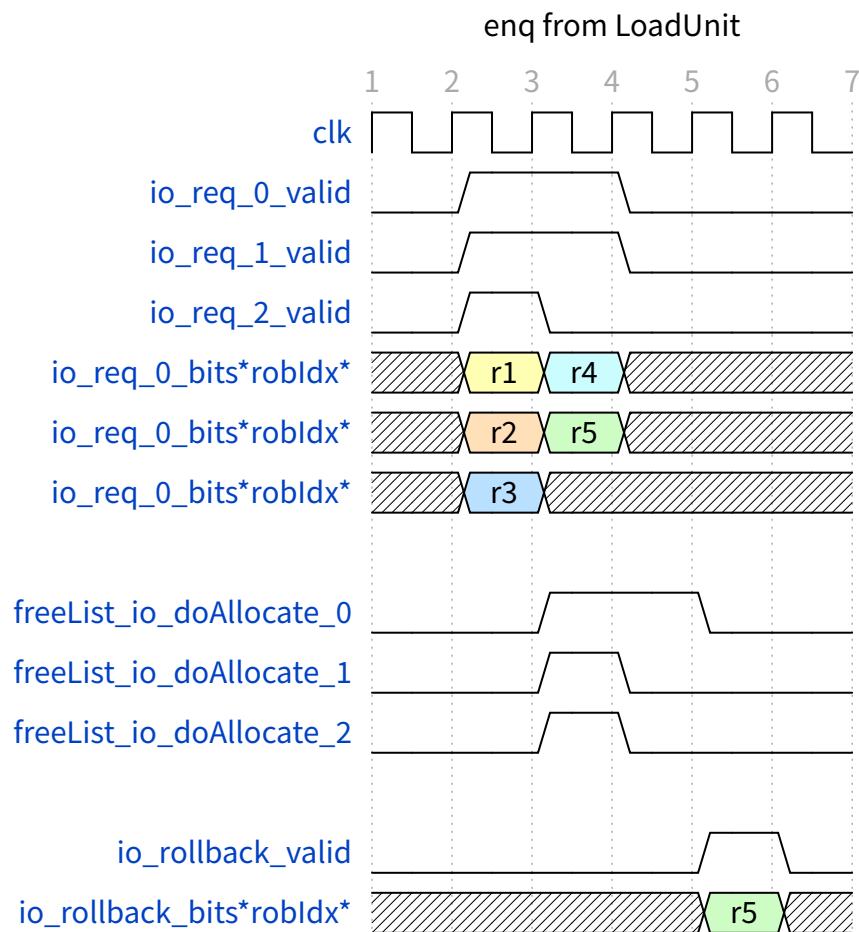


图 16.37: LoadQueueUncache 入队接口时序示意图

16.7.6.3.2 出队接口时序实例

下图展示了 mmioOut、一拍只有一个 ncOut 和一拍同时有两个 ncOut 的情况。用第一个例子详细说明，第 2 拍选出写回项，并更新 freeList，寄存一拍，第 3 拍写回 LoadUnit。后续例子同理可得。

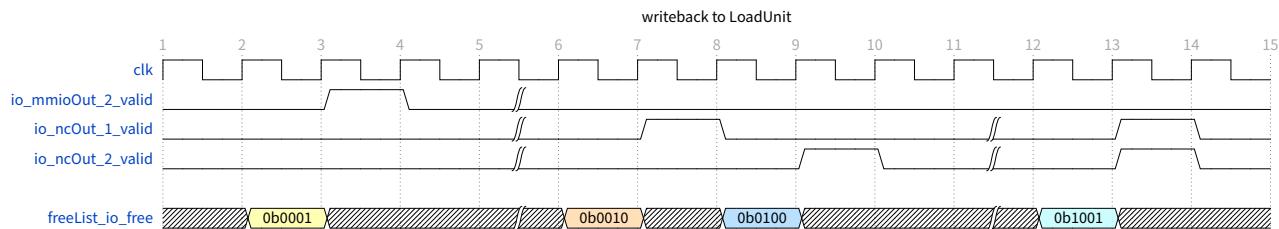


图 16.38: LoadQueueUncache 出队接口时序示意图

16.7.6.3.3 uncache 接口时序实例

(1) 没有 outstanding 时，每段只能发出一个 uncache 访问（由 io_uncache_req_ready 控制流出量），直至收到 uncache 回复。如下图，在第 5 拍 io_uncache_req_ready 拉高时，uncache 请求发出，第 6 拍 Uncache 收到请求并在第 7 拍返回 idResp。经过一段时间访问，在第 10+n 拍收到 Uncache 访问结果。

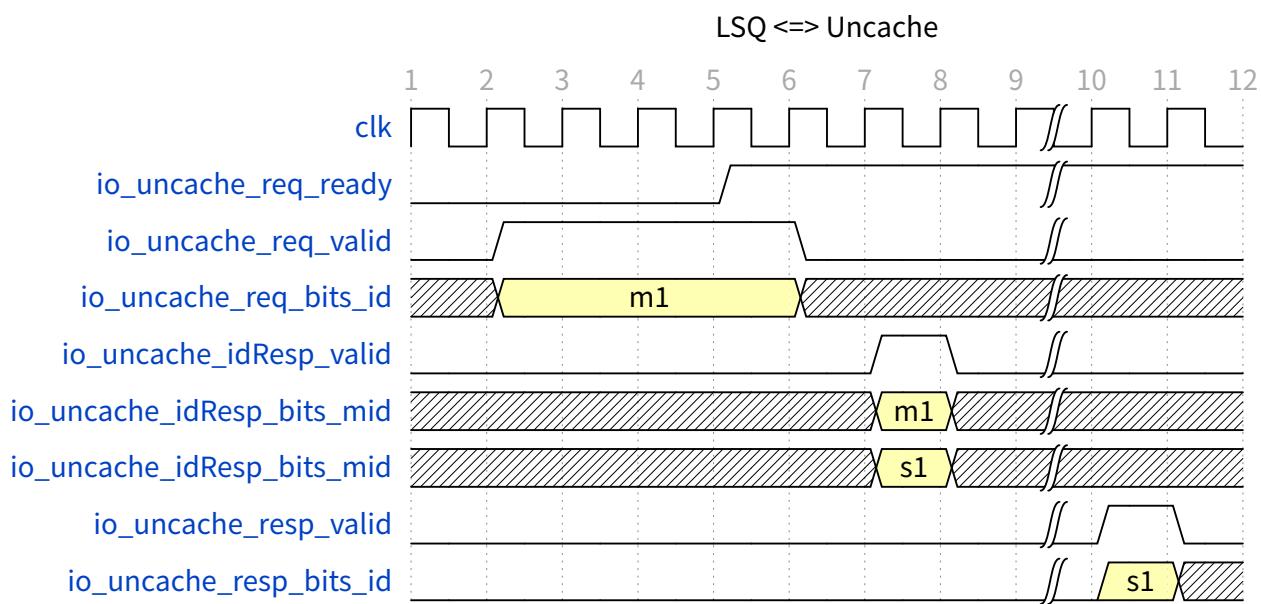


图 16.39: LoadQueueUncache 与 Uncache 的接口时序示意图

(1) 存在 outstanding 时，每段可发出多个 uncache 访问（由 io_uncache_req_ready 控制流出量）。如下图，连续发出 m1, m2, m3, m4 个请求，在第 4 拍和第 5 拍收到前 2 个请求的 Uncache 分派结果，此时 Uncache 满，m3 被中间寄存器寄存，m4 在等待 io_uncache_req_ready 拉高。第 9+n 拍 io_uncache_req_ready 拉高，m4 也发出了，并在第 10+n、11+n 拍分别收到 m3 和 m4 的 Uncache 分派结果。之后的拍数里将陆续收到 Uncache 的访问回复。

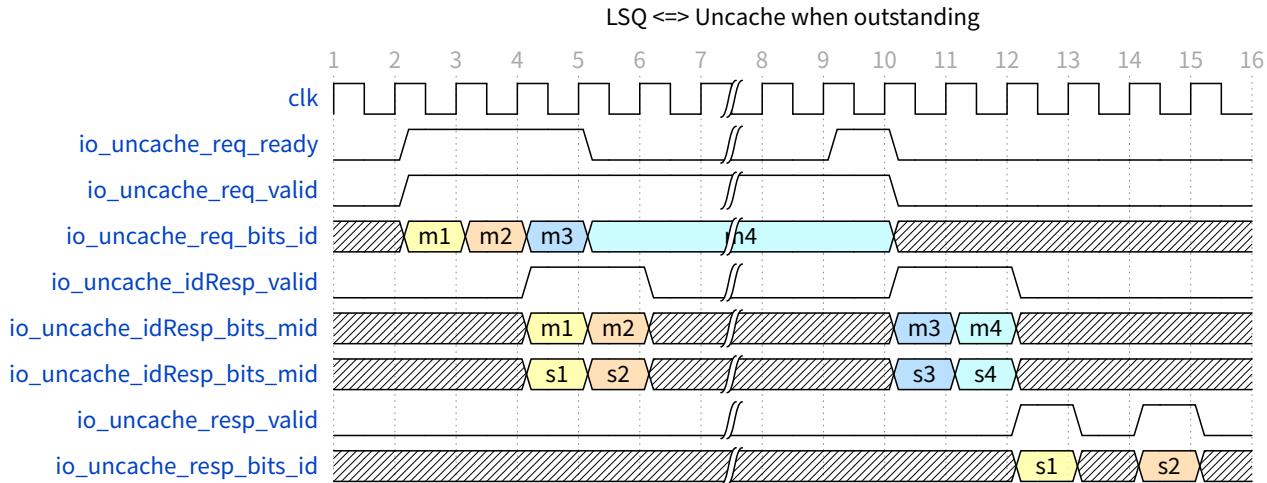


图 16.40: outstanding 时 LoadQueueUncache 与 Uncache 的接口时序示意图

16.7.6.4 UncacheEntry 模块

UncacheEntry 负责独立处理一个请求的生命周期，并利用一组状态寄存器来控制其具体的处理流程。关键结构如下：

- **req_valid**: 表示该项是否有效。
- **req**: 存储收到的请求的所有相关内容。
- **uncacheState**: 记录该项当前的生命阶段。
- **slaveAccept**、**slaveId**: 记录该项是否分配到 Uncache Buffer 以及分配的 Uncache Buffer ID。
- **needFlushReg**: 指示该项是否需要延迟刷新。

16.7.6.4.1 特性 1：生命周期及状态机

每一个 UncacheEntry 的生命周期可以由 **uncacheState** 完全描述。其中包括以下几个状态：

- **s_idle**: 默认状态，表示没有请求，或者请求存在但尚不具备发送到 Uncache Buffer 的条件。
- **s_req**: 表示当前已经具备将请求发送到 Uncache Buffer 的条件，静待被 LoadQueueUncache 选中，并由其中间寄存器接收（理论上应由 Uncache Buffer 接收，但在 LoadQueueUncache 选中后，会先将请求存放一拍，再发送给 Uncache Buffer；若未被 Uncache Buffer 接收，则会继续寄存在中间寄存器中）。对于 UncacheEntry 来说，它并不感知中间寄存器的存在，它只知道请求已发送且成功接收。
- **s_resp**: 表示该请求已被中间寄存器接收，静待 Uncache Buffer 返回访问结果。
- **s_wait**: 表示已经收到 Uncache Buffer 的访问结果，静待被 LoadQueueUncache 选中并由 LoadUnit 接收。

状态转换图如下，其中黑色标识该项正常生命周期，红色标识由于 redirect 需要刷新该项而导致该项生命周期非正常结束。

对于正常的生命周期，各个触发事件详细说明如下：

- **canSendReq**: 对于 MMIO 请求，当其对应的指令到达 ROB 头部时，则可发送该 Uncache 访问。对于 NC 请求，当 **req_valid** 有效时，则可发送该 Uncache 访问。
- **uncacheReq.fire**: 该项被 LoadQueueUncache 中间寄存器接收。该项会在下一拍收到 Uncache Buffer 传递来的 **idResp**，并更新 **slaveAccept** 和 **slaveId**。

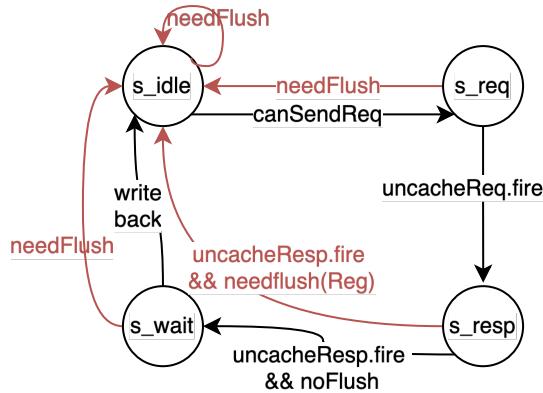


图 16.41: UncacheEntry 有限状态机示意图

- `uncacheReq.fire`: 该项收到的 Uncache Buffer 返回的访问结果。
- `writeback`: 当处于 `s_wait` 状态时，则可以发送写回请求。其中 MMIO 请求和 NC 请求的写回信号不一样，需要区分。

16.7.6.4.2 特性 2: redirect 刷新逻辑

对于非正常生命周期的情况，通常由流水线 redirect 引起。

当接收到流水线 redirect 信号时，判断当前项是否比 redirect 项更新。如果当前项更新，则需要刷新该项，并产生 `needFlush` 信号。一般情况下，会立即刷新该项所有内容，并由 FreeList 回收该项。但 Uncache 的请求和响应需要完整对应到同一个 uncache load 请求，故如果此时该项已经发出了 uncache 请求，需要等待接收到 Uncache 回复时才能结束该项的生命周期，此时产生了“刷新延迟”的情况。因此，在 `needFlush` 信号产生时，如果不能立即刷新该项，则需要将该信号存储到 `needFlushReg` 寄存器中。等到接收到 Uncache 回复时，才会执行刷新操作，并清除 `needFlushReg`。

16.7.6.4.3 特性 3：异常情况

`LoadQueueUncache` 中的异常情况有：

1. 该请求发往总线时，总线返回 `corrupt` 或 `denied` 的情况。该异常需要在 `UncacheEntry` 写回时进行标记，并由 `LoadUnit` 处理。

16.7.7 Load 异常缓冲 LqExceptionBuffer

16.7.7.1 功能描述

LqExceptionBuffer 用于跟踪 load 指令产生的异常情况，有三种来源：

- 来自 LDU s3 的标量 load 指令异常
- 来自 vlMergeBuffer 的向量 load 指令异常
- 来自 LoadUncacheBuffer 的 mmio non-data 异常

根据 robIdx 选择指令中最老的发生异常的指令的虚地址输出。内部有两级流水，第一级流水缓存 LDU 的 s3 阶段输出的信息，第二个周期根据 robIdx 选取最老的发生异常的指令，输出其虚拟地址。

重定向时根据 LqExceptionBuffer 内缓存的指令 robIdx 进行是否需要刷掉的判断。

16.7.7.2 整体框图

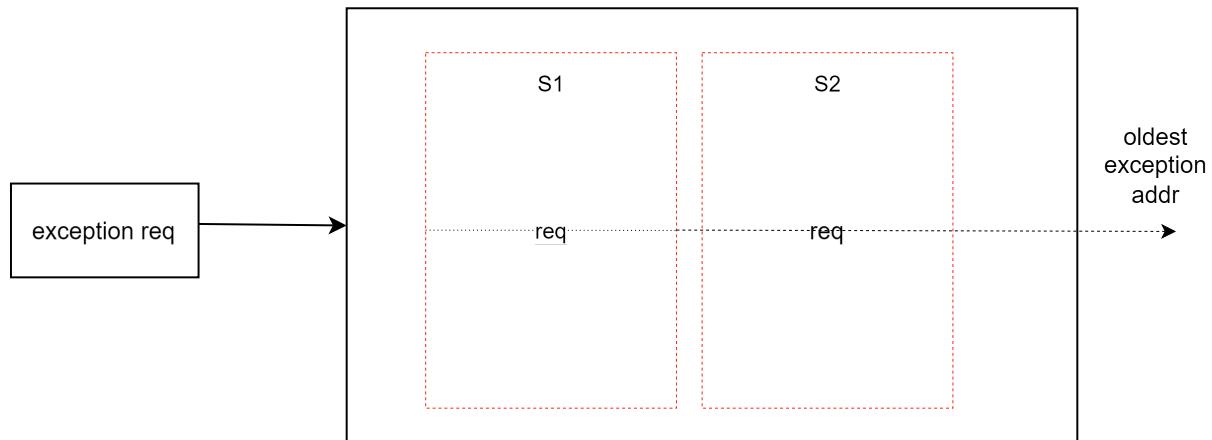


图 16.42: LqExceptionBuffer 整体框图

16.7.8 Store 队列 StoreQueue

16.7.8.1 功能描述

StoreQueue 是一个队列，用来装所有的 store 指令，功能如下：

- 在跟踪 store 指令的执行状态
- 存储 store 的数据，跟踪数据的状态（是否到达）
- 为 load 提供查询接口，让 load 可以 forward 相同地址的 store
- 负责 MMIO store 和 NonCacheable store 的执行
- 将被 ROB 提交的 store 写到 sbuffer 中
- 维护地址和数据就绪指针，用于 LoadQueueRAW 的释放和 LoadQueueReplay 的唤醒

store 进行了地址与数据分离发射的优化，即 StoreUnit 是 store 的地址发射出来走的流水线，StdExeUnit 是 store 的数据发射出来走的流水线，是两个不同的保留站，store 的数据就绪了就可以发射到 StdExeUnit，store 的地址就绪了就可以发射到 StoreUnit。

- StoreQueue 中每一项保存了 store 指令的基础信息：

表 16.21: StoreQueue 存储的基础信息

| Field | 描述 |
|-------------|-------------------|
| uop | store 指令 uop |
| dataModule | 128bits 数据和数据有效掩码 |
| paddrModule | 物理地址 |
| vaddrModule | 虚拟地址 |

- StoreQueue 中每一项都有若干状态位来表示这个 store 处于什么样的状态：

表 16.22: StoreQueue 存储的状态信息

| Field | 描述 |
|---------------|--|
| allocated | 设置这个 entry 的 allocated 状态，开始记录这条 store 的生命周期。 当这条 store 指令被提交到 Sbuffer 时，allocated 状态被清除。 |
| addrvalid | 表示是否已经经过了地址转换得到物理地址，用于 load forward 检查时的 cam 比较。 |
| datavalid | 表示 store 的数据是否已经被发射出来，是否已经可用 |
| committed | store 是否已经被 ROB commit 了 |
| unaligned | 非对齐 Store |
| cross16Byte | 跨 16 字节边界 |
| pending | 在这条 store 是否是 MMIO 空间的 store，主要是用于控制 MMIO 的状态机 |
| nc | NonCacheable store |
| mmio | mmio store |
| atomic | 原子 store |
| memBackTypeMM | 是否是 PMA 为 main memory 类 |
| prefetch | 当提交到 Sbuffer 是否需要预取 |
| isVec | 向量 store |
| vecLastFlow | 向量 store flow 的最后一个 uop |
| vecMbCommit | 从合并缓冲区提交到 rob 的向量 Store |
| hasException | store 指令有异常 |
| waitStoreS2 | 等待 Store Unit s2 的 mmio 和异常结果 |

16.7.8.1.1 特性 1: 数据前递

- load 需要查询 StoreQueue 来找到在它之前的相同地址的与它最近的依赖 store 的数据。

- 查询总线 (io.forwrd.sqIdx) 和 StoreQueue 的 enqPtr 指针比较, 找出所有比 load 指令老的 Store-Queue 中的 entry。以 flag 相同或不同分为 2 种情况
 - * 如果是 same flag, 则 older Store 范围是 [tail, sqIdx - 1], 如图16.43 a) 所示; 否则 older Store 范围是 [tail, VirtualLoadQueueSize - 1] 和 [0, sqIdx], 如图16.43 b) 所示

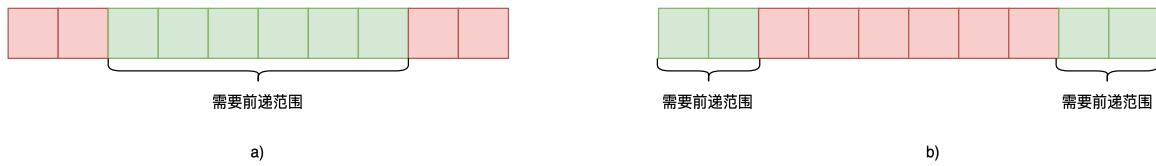


图 16.43: StoreQueue 前递范围生成

- 查询总线用虚拟地址和物理地址同时查询，如果发现物理地址匹配但是虚拟地址不匹配；或者虚拟地址匹配但是物理地址不匹配的情况就需要将那条 load 设置为 replayInst，等 load 到 ROB head 后重新取指令执行。
 - 如果只发现一笔 entry 匹配且数据准备好，则直接 forward
 - 如果只发现一笔 entry 匹配且数据没有准备好，就需要让保留站负责重发
 - 如果发现多笔匹配，则选择最老的一笔 store 前递
 - StoreQueue 以 1 字节为单位，采用树形数据选择逻辑，如图16.44所示

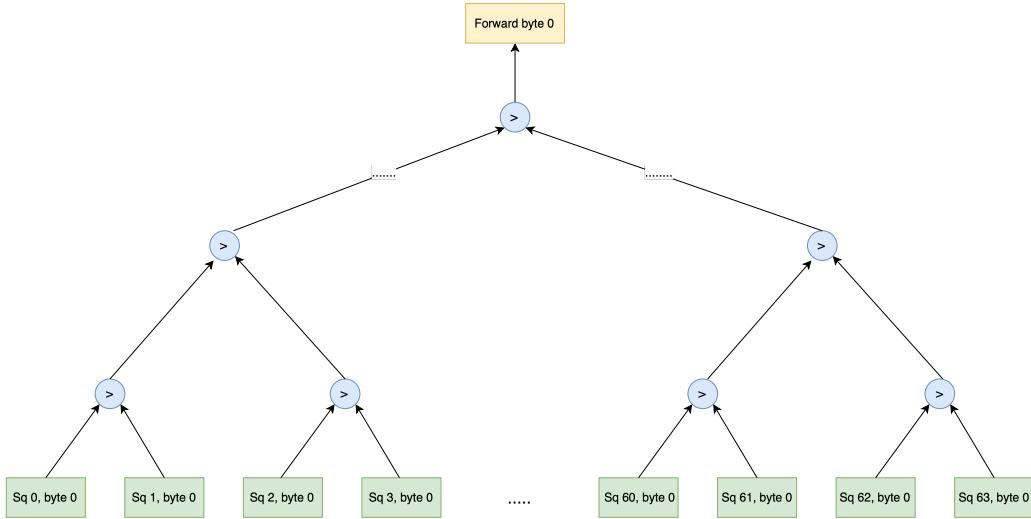


图 16.44: StoreQueue 前递数据选择

- 参与数据前递的 store 需要满足：

- allocated: 这条 store 还在 store queue 内，还没有写到 sbuffer
- datavalid: 这条 store 的数据已经就绪
- addrvalid: 这条 store 已经完成了虚实地址转换，得到了物理地址
- 如果启用了访存以来预测器，则 SSID (Store-Set-ID) 标记了之前 load 预测执行失败历史信息，如果当前 load 命中之前历史中的 SSID，会等之前所有 older 的 store 都执行完；如果没有命中就只会等物理地址相同的 older Store 执行完成。

16.7.8.1.2 特性 2：非对齐 store 指令

StoreQueue 支持处理非对齐的 Store 指令，每一个非对齐的 Store 指令占用一项，并在写入 dataBuffer 对地址和数据对齐后写入。

16.7.8.1.3 特性 3：向量 Store 指令

如图16.45所示，StoreQueue 会给向量 store 指令预分配一些项。StoreQueue 通过 vecMbCommit 控制向量 store 的提交：

- 针对每个 store，从反馈向量 fblk 中获取相应的信息。

判断该 store 是否符合提交条件 (valid 且标记为 commit 或 flush)，并且检查该 store 是否与 uop(i) 对应的指令匹配（通过 robIdx 和 uopIdx）。只有当满足所有条件时，才会将该 store 标记为提交。判断 VecStorePipelineWidth 内是否有指令满足条件，满足则判断该向量 store 提交，否则为提交。

- 特殊情况处理 (Store 跨页)：

在特殊情况下（当 store 跨页且 storeMisalignBuffer 中有相同的 uop），如果该 store 符合条件 io.maControl.toStoreQueue.withSameUop，会强制将 vecMbCommit 设置为 true，表示该 store 无论如何都已提交。

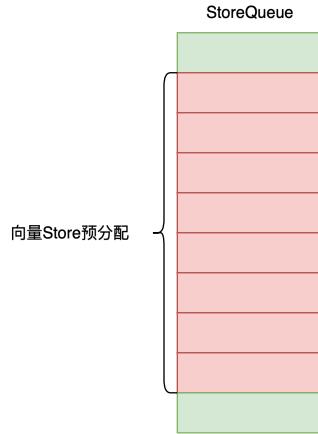


图 16.45: 向量 Store 指令

16.7.8.1.4 特性 4: CMO

StoreQueue 支持 CMO 指令，CMO 指令共用 MMIO 的状态机控制:

- s_idle: 空闲状态, 接收到 CMO 的 store 请求后进入到 s_req 状态;
- s_req: 刷新 Sbuffer, 等待刷行完成之后, 通过 CMORReq 发送 CMO 操作请求, 进入 s_resp 状态
- s_resp: 接受到 CMORResp 返回的响应, 进入 s_wb 状态
- s_wb: 等待 ROB 提交 CMO 指令, 进入 s_idle 状态

16.7.8.1.5 特性 5: CBO

StoreQueue 支持 CBO.zero 指令:

- CBO.zero 指令的数据部分将 0 写入 dataModule
- CBO.zero 写入 Sbuffer 时: 刷新 Sbuffer, 等待刷新完毕之后, 通过 cboZeroStout 写回。

16.7.8.1.6 特性 6: MMIO 与 NonCacheable Store 指令

- MMIO Store 指令执行
 - MMIO 空间的 store 也只能等它到达 ROB 的 head 时才能执行, 但是跟 load 稍微有些不同, store 到达 ROB 的 head 时, 它不一定位于 store queue 的尾部, 有可能有的 store 已经提交, 但是还在 store queue 中没有写入到 sbuffer, 需要等待这些 store 写到 sbuffer 之后, 才能让这条 MMIO 的 store 去执行
 - 利用一个状态机去控制 MMIO 的 store 执行
 - * s_idle: 空闲状态, 接收到 MMIO 的 store 请求后进入到 s_req 状态;
 - * s_req: 给 MMIO 通道发请求, 请求被 MMIO 通道接受后进入 s_resp 状态;
 - * s_resp: MMIO 通道返回响应, 接收后记录是否产生异常, 并进入到 s_wb 状态
 - * s_wb: 将结果转化为内部信号, 写回给 ROB, 成功后, 如果有异常, 则进入 s_idle, 否则进入到 s_wait 状态

- * s_wait: 等待 ROB 将这条 store 指令提交, 提交后重新回到 s_idle 状态
- NonCacheable Store 指令执行
 - NonCacheable 空间的 store 指令, 需要等待提交之后, 才能从 StoreQueue 按序发送请求
 - 利用一个状态机去控制 NonCacheable 的 store 执行
 - * nc_idle: 空闲状态, 接收到 NonCacheable 的 store 请求后进入到 nc_req 状态;
 - * nc_req: 给 NonCacheable 通道发请求, 请求被 NonCacheable 通道接受后, 如果启用 uncachable Outstanding 功能, 则进入 nc_idle, 否则进入 nc_resp 状态;
 - * nc_resp: 接受 NonCacheable 通道返回响应, 并进入到 nc_idle 状态

16.7.8.1.7 特性 7: store 指令提交以及写入 SBuffer

StoreQueue 采用提前提交的方式 * 提前提交规则:

- 检查进入提交阶段的条件
 - 指令有效。
 - 指令的 ROB 对头指针不超过待提交指针。
 - 指令不需要取消。
 - 指令不等待 Store 操作完成, 或者是向量指令
- 如果是 CommitGroup 的第一条指令, 则
 - 检查 MMIO 状态: 没有 MMIO 操作或者有 MMIO 操作并且 MMIO store 以及提交。
 - 如果是向量指令, 否则需满足 vecMbCommit 条件,。
- 如果不是 CommitGroup 的第一条指令, 则:
 - 提交状态依赖于前一条指令的提交状态。
 - 如果是向量指令, 需满足 vecMbCommit 条件。

提交之后可以按顺序写到 sbuffer, 先将这些 store 写到 dataBuffer 中, dataBuffer 是一个两项的缓冲区(0, 1 通道), 用来处理从大项数 store queue 中的读出延迟。只有 0 通道可以编写未对齐的指令, 同时为了简化设计, 即使两个端口出现异常, 但仍然只有一个未对齐出队。

- 写入有效信号生成:
 - 0 通道指令存在非对齐且跨越 16 字节边界时:
 - * 0 通道的指令已分配和提交
 - * dataBuffer 的 0, 1 通道能同时接受指令,
 - * 0 通道的指令不是向量指令, 并且地址和数据有效; 或者是向量且 vsMergeBuffer 以及提交。
 - * 没有跨越 4K 页表; 或者跨越 4K 页表但是可以被出队, 并且 1) 如果是 0 通道: 允许有异常的数据写入; 2) 如果是 1 通道: 不允许有异常的数据写入。
 - * 之前的指令没有 NonCacheable 指令, 如果是第一条指令, 自身不能是 NonCacheable 指令

- 否则，需要满足
 - * 指令已分配和提交。
 - * 不是向量且地址和数据有效，或者是向量且 vsMergeBuffer 以及提交。
 - * 之前的指令没有 NonCacheable 和 MMIO 指令，如果是第一条指令，自身不能是 Noncacheable 和 MMIO 指令。
 - * 如果未对齐 store，则不能跨越 16 字节边界，且地址和数据有效或有异常
- 地址和数据生成：
 - 地址拆分为高低两部分：
 - * 低位地址：8 字节对齐地址
 - * 高位地址：低位地址加上 8 偏移量
 - 数据拆分为高低两部分：
 - * 跨 16 字节边界数据：原始数据左移地址低 4 位偏移量包含的字节数
 - * 低位数据：跨 16 字节边界数据的低 128 位；
 - * 高位数据：跨 16 字节边界数据的高 128 位；
 - 写入选择逻辑：
 - * 如果 dataBuffer 能接受非对齐指令写入，通道 0 的指令是非对齐并且跨越了 16 字节边界，则
 - 检查是否跨 4K 页表同时跨 4K 页表可以出队：通道 0 使用低位地址和低位数据写入 dataBuffer；通道 1 使用 StoreMisaligBuffer 的物理地址和高位数据写入 dataBuffer
 - 否则：通道 0 使用低位地址和低位数据写入 dataBuffer；通道 1 使用高位地址和高位数据写入 dataBuffer
 - * 如果通道指令没有跨越 16 字节并且非对齐，则使用 16 字节对齐地址和对齐数据写入 dataBuffer
 - * 否则，将原始数据和地址写给 dataBuffer

16.7.8.1.8 特征 7：强制刷新 Sbuffer

StoreQueue 采用双阈值的方法控制强制刷新 Sbuffer：上阈值和下阈值。当 StoreQueue 的有效项数大于上阈值时，StoreQueue 强制刷新 Sbuffer，直到 StoreQueue 的有效项数小于下阈值时，停止刷新 Sbuffer，

16.7.8.2 整体框图

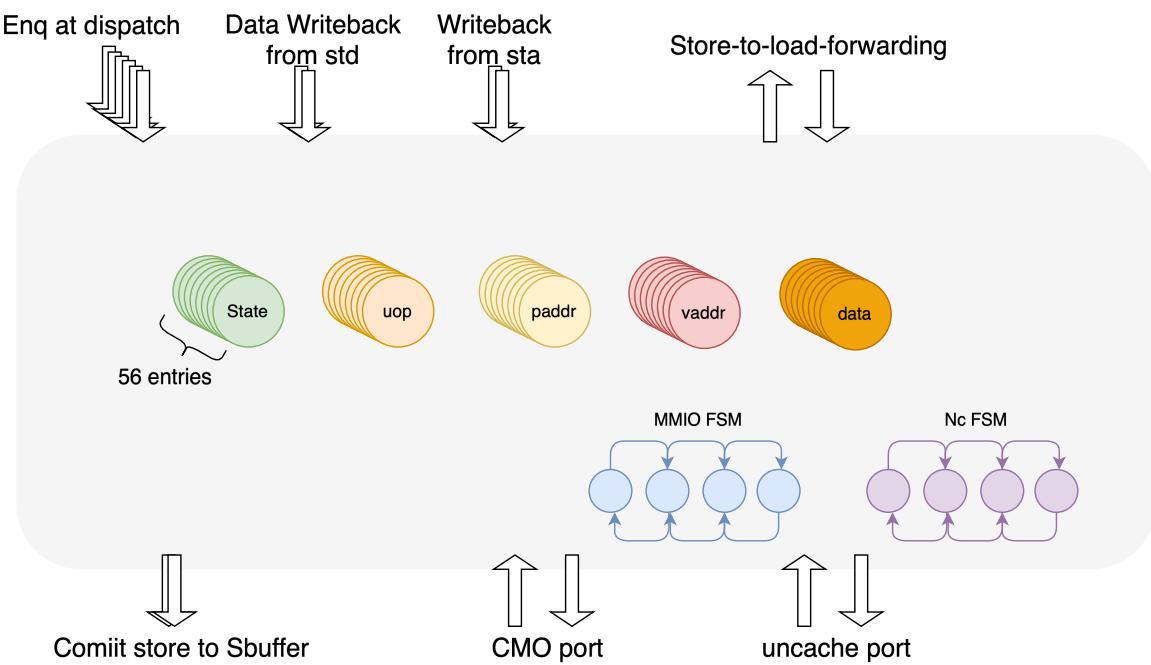


图 16.46: StoreQueue 整体框架

16.7.8.3 接口时序

16.7.8.3.1 入队接口时序实例

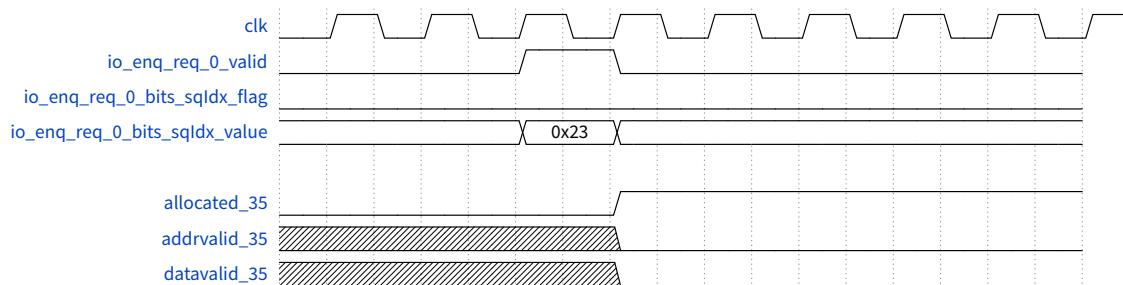


图 16.47: StoreQueue 整体框架

16.7.8.3.2 数据更新接口时序

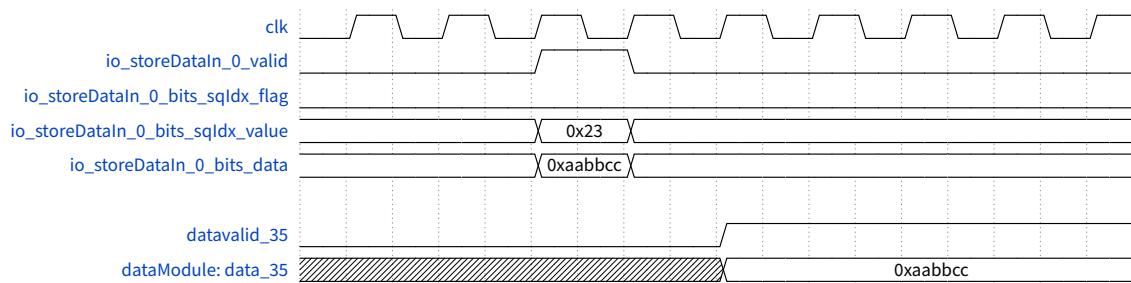


图 16.48: 数据更新接口时序

16.7.8.3.3 地址更新接口时序

StoreQueue 地址更新和数据更新类似, StoreUnit 通过 s1 阶段的 `io_lsq` 更新地址, 在 s2 阶段通过 `io_lsq_replenish` 更新异常, 与数据更新不同的是, 更新地址只需要一拍, 而不是两拍

16.7.8.3.4 MMIO 接口时序实例

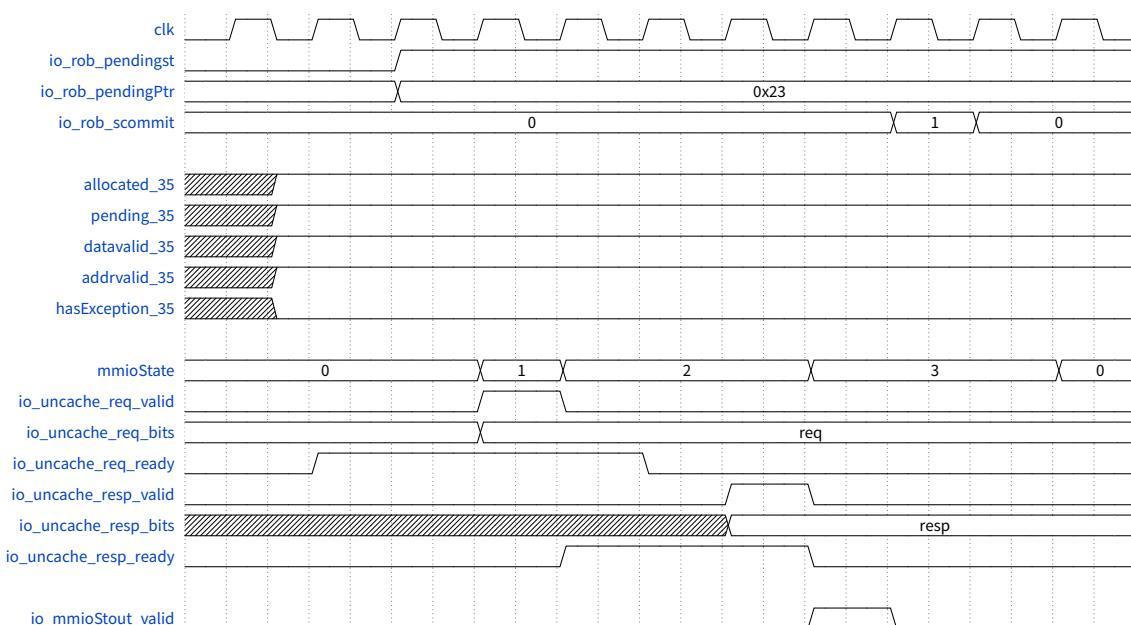


图 16.49: MMIO 接口时序实例

16.7.8.3.5 NonCacheable 接口时序实例

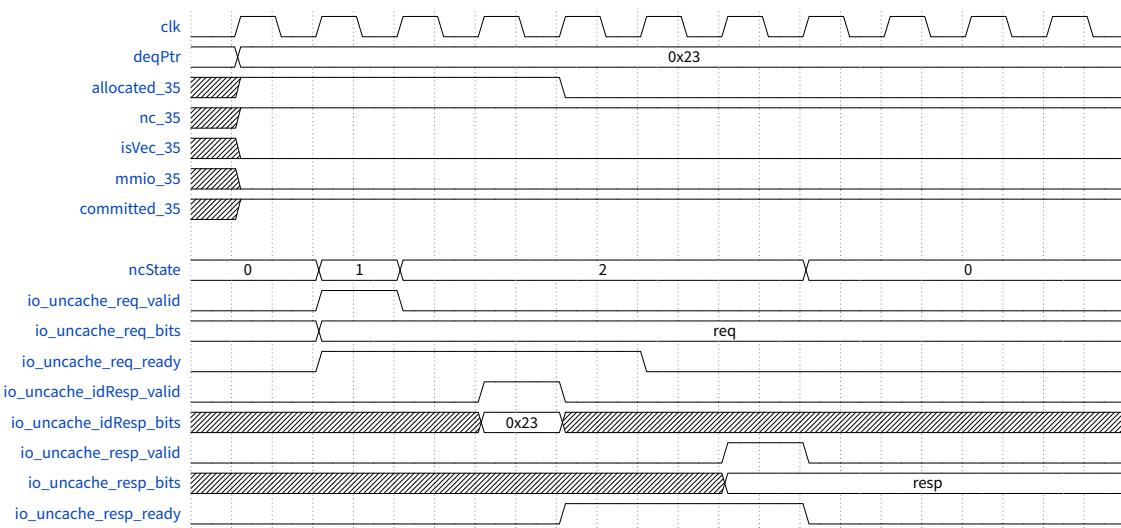


图 16.50: NonCacheable 接口时序实例

16.7.8.3.6 CBO 接口时序实例

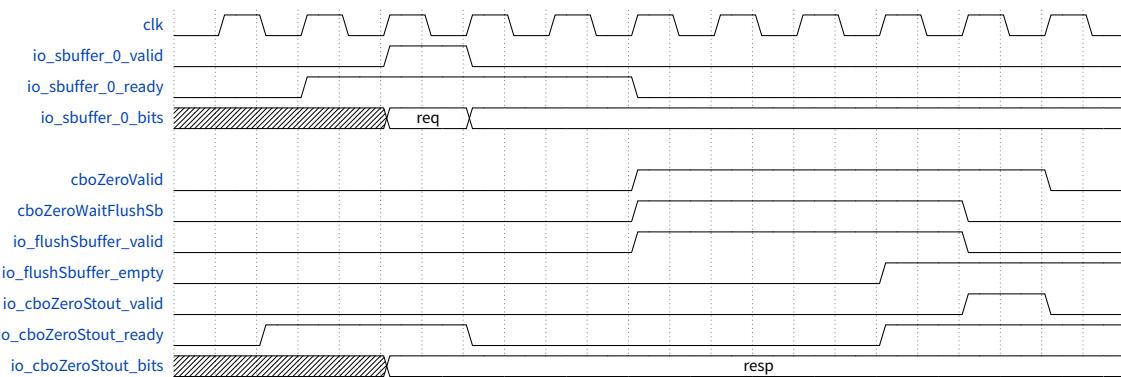


图 16.51: CBO 接口时序实例

16.7.8.3.7 CMO 接口时序实例

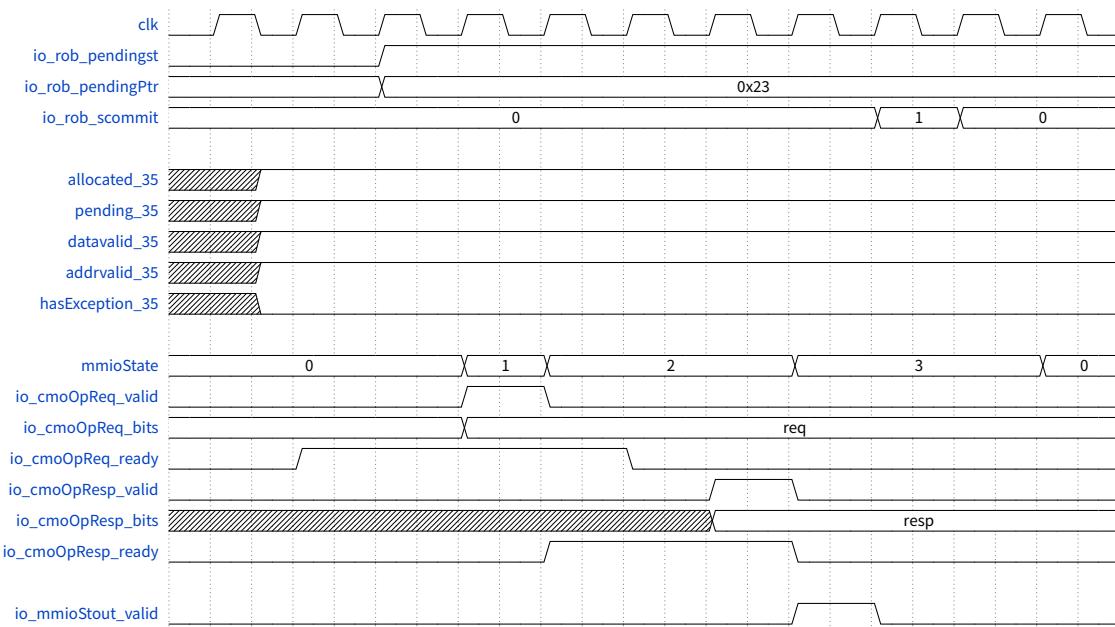


图 16.52: CMO 接口时序实例

16.8 Uncache 处理单元 Uncache

| 更新时间 | 代码版本 | 更新人 | 备注 |
|---------------------|------|-------------|------|
| 2025.02.26 16:59:33 | | Maxpicca-Li | 完成初版 |

16.8.1 功能描述

Uncache 作为 LSQ 和总线的桥梁，主要用于处理 uncache 访问到总线的请求和响应。目前 Uncache 不支持向量访问、非对齐访问、原子访问。

Uncache 的功能概述如下：

1. 接收 LSQ 传过来的 uncache 请求，包括 LoadQueueUncache 传来的 uncache load 请求和 StoreQueue 传来的 uncache store 请求
2. 选择候机的 uncache 请求发送到总线，等待并接收总线回复
3. 将处理完的 uncache 请求返回给 LSQ
4. 前递寄存的 uncache store 请求的数据给 LoadUnit 中正在执行的 load

Uncache Buffer 结构上，目前有 4 项（项数可配）Entries 和 States，一个总的状态 uState。下列为各项具体细节。

Uncache 的 Entry 结构如下：

- cmd：标识请求是 load 还是 store，当前版本 0 为 load，1 为 store。
- addr：请求物理地址。
- vaddr：请求虚拟地址。主要用于前递时判断虚实地址是否 match
- data：store 要写入的数据，或 load 要读取的数据，目前仅支持 64 bits 以内的数据访问。
- mask：请求访问掩码，每 byte 使用一位来表示当前有没有数据，共 8 位。
- nc：请求是否是 NC 访问。
- atomic：请求是否是原子访问。
- memBackTypeMM：请求所访问的地址，是否是 PMA 为 main memory 类型，但 PBMT 为 NC 类型。主要用于 L2 Cache NC 相关逻辑。
- resp_nderr：总线告知 Uncache，该请求是否能处理。

Uncache 的 State 结构如下：

- valid：该项是否有效。
- inflight：1 表示该项请求已经发往总线。
- waitSame：1 表示当前 buffer 里存在与该项请求所访问数据块重合的其他请求，已经发往总线。
- waitReturn：1 表示该项的请求已经接收到总线回复，等待写回 LSQ。

Uncache 的 uState，表征忽视 outstanding 时一个请求项的各个状态：

- s_idle 默认状态
- s_inflight 已经发送了一个请求到总线上，但还未收到回复
- s_wait_return 已经收到回复，但还未返回给 LSQ

状态转换如下：

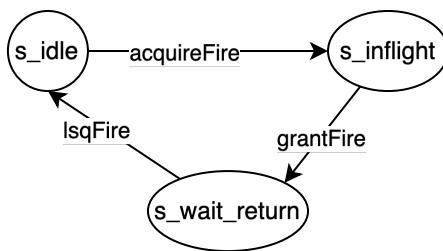


图 16.53: ustate 状态转换示意图

16.8.1.1 特性 1：入队逻辑

(1) 每一拍最多处理 1 个从 LSQ 发来的请求，然后检查请求是否能进入 Buffer，若能，则检查是合并到老项还是分配新项。该请求的入队行为有：

1. 分配新项，标记 valid

1. 无相同块地址项

2. 分配新项，标记 valid 和 waitSame
 1. 有相同块地址项：满足首要合并条件，不满足次要合并条件。
3. 合并到老项
 1. 有相同块地址项：满足首要合并条件，满足次要合并条件。
4. 拒绝
 1. ubuffer 满
 2. 有相同块地址项：不满足首要合并条件

其中，块地址，即 blockAddr，为每 8 Bytes 的起始地址。首要合并条件指，来项和老项均为 NC 访问、各个属性相同、与老项合并后的 mask 满足连续且自然对齐、且该项当拍没有正在或已经完成了总线访问。次要合并条件是指老项有效、还没有发送总线访问、也没有当拍被选中发往总线访问（因为如果一旦正在或已经发往总线，总线请求已经无法更改，只能分配新项，等待老项收到总线请求后再发送该请求）。

另外，分配新项，会设置 entry 的各个内容；合并老项，会更新 mask，data，addr 等内容。其中 addr 更新需要保证自然对齐。

由于总线访问不一定保序，尤其是 outstanding 时，总线上会同时处理多个 uncache 访问请求。故相同地址的请求不能同时出现在总线上，以保证该地址数据块的被访问顺序。故只有新项满足首要和次要的合并条件，才能合并到老项。

(2) 下一拍，返回所分配的 Uncache Buffer 项目 ID。由 LoadQueueUncache 或 StoreQueue 保管该 ID，用于映射 uncache 返回的 resp。因为 Uncache Buffer 有合并功能，即其返回的 resp 可能对应 LoadQueue-Uncache 中的多个项。

16.8.1.2 特性 2：出队逻辑

从当拍已经完成总线访问的项（即状态高位有 valid 和 waitReturn）中，选择一项，返回给 LSQ，并清除所有 state 标识位。

16.8.1.3 特性 3：总线交互和 outstanding 逻辑

总线交互和 outstanding 逻辑分以下两个部分：

(1) 发起请求

无 outstanding 时，仅当 `ustate` 为 `s_idle` 时才能发送请求到总线上。从各个项中选出一个目前可以发往总线的请求，即各状态位仅 `valid` 置 1，发往总线。有 outstanding 时，可无视 `ustate` 即选择请求项，并发往总线。其中 `source` 位为该请求项的 id。

当请求发往总线时，需要遍历请求项，将相同块地址的其他项的 `waitSame` 置位。

(2) 收到回复

当收到总线回复时，根据 `source` 位确定该请求对应的 buffer 项，更新数据并置位 `waitReturn`。此外，需要遍历请求项，将当前相同块地址的 `waitSame` 清除。

16.8.1.4 特性 4：前递逻辑

理论需求上，前递逻辑主要针对 NC 访问。当开启 outstanding 时，uncache NC store 从 StoreQueue 成功写入 Uncache Buffer 后，StoreQueue 便会将该项出队，不在维护。故此时的 Uncache Buffer 将承担前递该 store 数据的责任。由于 Uncache Buffer 的入队逻辑存在合并，同一时间，相同地址在 Uncache Buffer 中最多出现 2 项。若出现 2 项，其中一项一定为 `inflight`，另一项一定为 `waitSame`。因为 StoreQueue 的顺序出队，前者数据更老，后者数据更新。

实际处理上，当 uncache NC load 向 Uncache Buffer 发起前递请求时，Uncache 会比较现有项的块地址，有可能会找到匹配的项，这个项可能是已经发往总线的，也可能是还未发往总线的。前者数据更老，后者数据更新即优先级更高。在第一拍 `f0` 主要进行虚拟块地址匹配，以在当拍返回 `forwardMaskFast`，在第二拍 `f1` 进行的物理块地址匹配和数据合并，并返回结果。

16.8.1.5 特性 5：刷新逻辑

刷新，是指将 Uncache Buffer 内的所有项全部完成总线访问并返回给 LSQ 后才能接受新项的进入。当产生 fence atomic cmo 或前递出现虚实地址不匹配时，会刷新 Uncache Buffer。此时 `do_uarch_drain` 置位，不再接受新项的进入。当所有项都完成任务后，`do_uarch_drain` 清除，开始正常接受新项的进入。

16.8.2 整体框图

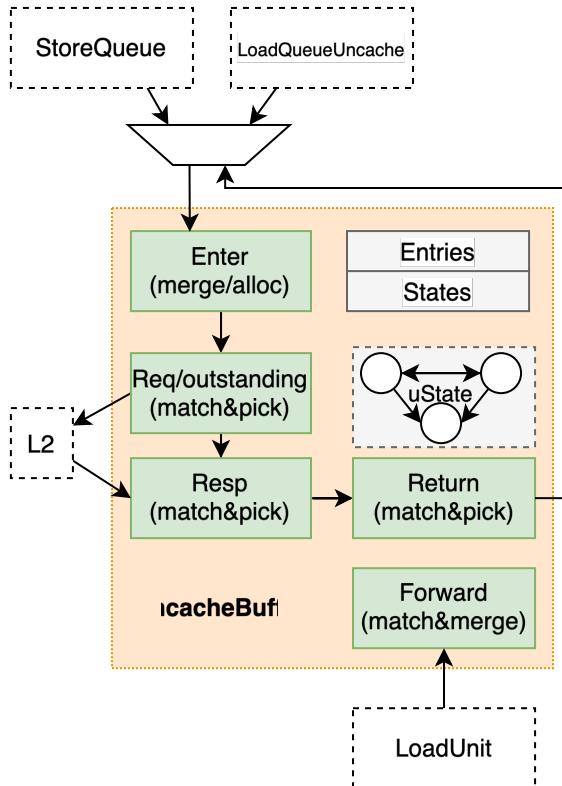
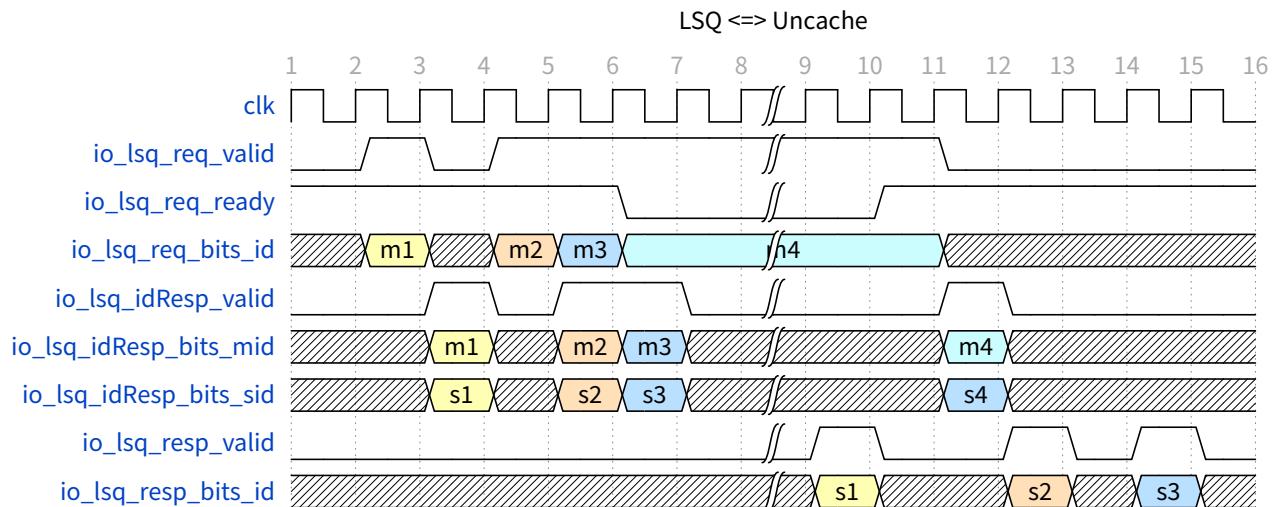


图 16.54: ubuffer 整体框图

16.8.3 接口时序

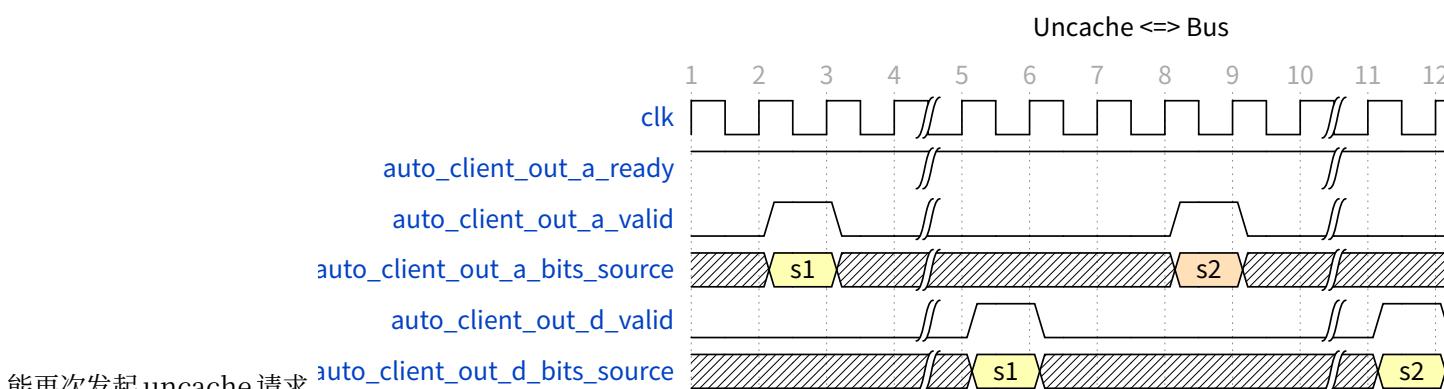
16.8.3.1 LSQ 接口时序实例

下图是一个比较详细的接口案例，共 4 个 uncache 访问。第 5 拍前陆续接收 m1、m2、m3，并在其请求发起的随后一拍返回 idResp。第 6 拍 Uncache 满，m4 被停滞。第 9+n 拍完成 s1 的所有访问并写回，释放了一个项。故第 10+n 拍 io_lsq_req_ready 拉高，m4 被接收。之后的拍数里陆续写回其他 uncache 访问请求。



16.8.3.2 总线接口时序实例

(1) 没有 outstanding 时, 每段只能发出一个 uncache 请求(由 uState 控制流出量), 直至收到 d 通道回复, 才能再次发起 uncache 请求。



(2) 有 outstanding 时, 每段可发出多个 uncache 访问 (由 auto_client_out_a_ready 控制流出量)。如下图, 第 2、3 拍连续发出两个请求, 并在第 6+n、第 8+n 拍收到访问结果。

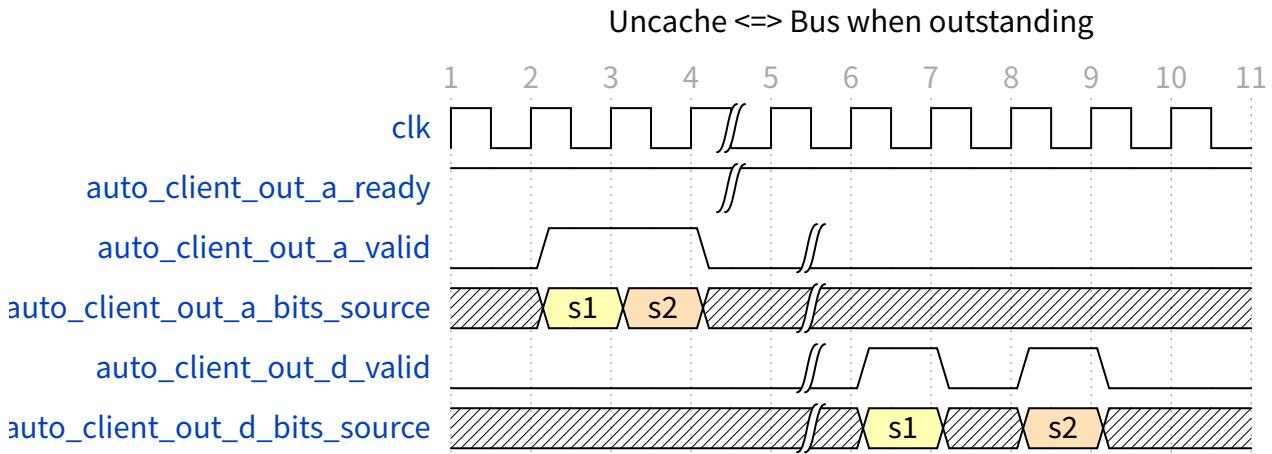


图 16.55: outstanding 时 Uncache 与总线的接口时序示意图

16.9 Store 提交缓冲 SBuffer

16.9.1 功能描述

sbuffer 每一项是一个 cacheline，每个 cacheline 是 64 bytes，也就是 4 个 vwords，每个 vwords 是 16 bytes。

每一个 byte 使用一位的 mask 来指示当前有没有数据。

meta 信息包括 ptag, vtag, state, cohCount, missqReplayCount，具体功能为：

- ptag: 物理地址 tag，物理地址中 cacheline 偏移的其他部分。
- vtag: 虚拟地址 tag，虚拟地址中 cacheline 偏移的其他部分。
- state: 状态，表示当前项处于什么状态。
 - state_valid: 项是否有效。
 - state_inflight: 项已经向 dcache 发送了写请求，还没有响应，或者 dcache 响应了但是 miss 了。
 - w_timeout: 发给 dcache 的请求 miss 了，等待重发。
 - w_sameblock_inflight: 有其他的项和本身这个项具有相同的 cache 块地址，其他的项已经 inflight 了，当前这个项刚刚被分配，需要等待其他项完成 dcache 的写回。
- cohCount: 计数器，从 0 计数到 1M 之后将该项写到 dcache。
- missqReplayCount: 计数器，之前发给 dcache 的请求发生过 miss，从 0 计数到 16 之后将该项重发给 dcache。

16.9.1.1 特性 1: sbuffer 的入队逻辑

- 每一拍最多处理两个从 StoreQueue 发来的请求，然后检查请求是否需要分配新的 entry，如果两个都需要分配新的 entry 则按奇偶选出两个空闲项进行分配。如果两个请求的 ptag 相同，则分配到同一个空闲项。
- 如果已经有相同 cacheline 的项就不需要分配新的项，直接合并到相同项内；如果这个相同的 cacheline 已经被发给 dcache 了 (state_inflight 为 true)，就不能进行合并，需要重新分配一项，并且记录新分配

的项依赖 inflight 的项 (设置 w_sameblock_inflight 为 true, waitInflightMask 为 inflight 项的 id), 记录依赖的目的是让 inflight 的项写到 dcache 之后才能让新的这一项写到 dcache, 保证 store 的顺序。

- 设置这一项状态位为 valid。
- 请求进 sbuffer 进行合并时, 如果这一项刚好被选为要写到 dcache 的项, 就要阻塞 dcache 的写, 等待合并完成再写。

16.9.1.2 特性 2: sbuffer 的出队逻辑

- sbuffer 里的项写入 dcache 分被动和主动的情况。
 - 被动: sbuffer 中的项数量达到阈值, 需要替换。
 - 主动: atomicsUnit 和 fenceUnit 发来的 flush sbuffer 信号, 或者自身在做合并或者给 load 前递的时候发生了 tag 不匹配的情况, 或者之前 miss 的请求重新发送。
- 出 sbuffer 分两拍, 第一拍选择要写到 dcache 的项锁存, 第二拍再给 dcache 发写请求。

16.9.1.3 特性 3: 写 sbuffer data

请求到达 sbuffer 时, 要么分配新的一项, 要么合并到已有的项里面, 写 data 和 mask 的时候分两拍, 第一拍将请求锁存起来, 第二拍根据请求的 mask 写入 (sb, sh, sw, sd), 并将对应的 mask (表示某个 cache line 上的某一个 byte 是否 valid 的信号) 置位。

例如: S0 请求到达 sbuffer, S0 做判断逻辑得到该请求可以合并到已有的一项中, 该项为第 2 项, 于是生成一个 one hot 写入编码为 16' b0000000000000100, 利用这个写入编码, 产生对第 2 项的写入信号, 将其锁存到 S1, 并把 S0 的写入地址 (例如 cache 块内地址为 0), mask(例如 sw, 写入 4 个 bytes), 数据锁存到 S1。S1 根据 S0 锁存的信息, 把第 2 项的第 0 个 word 的低 4 个 byte 的数据写信号拉高, 写入对应的数据, 把第 2 项的第 0 个 word 的低 4 个 byte 的 mask 写信号拉高, 将其改为 true。

16.9.1.4 特性 4: sbuffer 的前递逻辑

- load 需要找在它之前的 store 的数据, 而这个 store 有可能在 storequeue 里, 有可能在 sbuffer 里, 也有可能已经写入到了 cache 里。
- 当它在 sbuffer 里找的时候, 比较现有项的 tag, 有可能会找到匹配的项, 这个项可能是还没有给 dcache 发请求的, 也可能是已经给 dcache 发了请求的, 还没有发的是最新的, 所以还没有发的优先级更高, 将匹配的 data 前递给 load。

如下图所示, 前递查询请求与 sbuffer 的第 0 项与 15 项同时发生了匹配, 而第 0 项的数据是最新的, 第 15 项是旧的, 于是前递结果中第 0 项的优先级高于第 15 项。

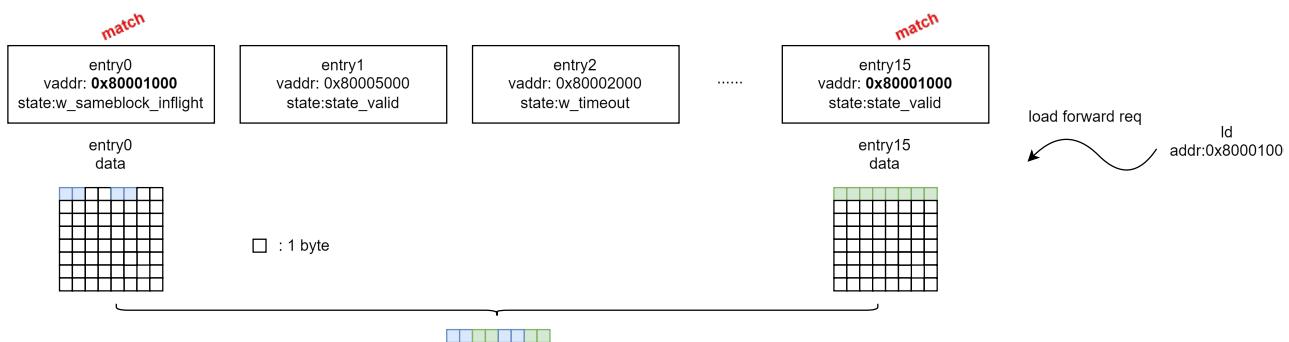


图 16.56: sbuffer 的前递示意图

16.9.2 整体框图

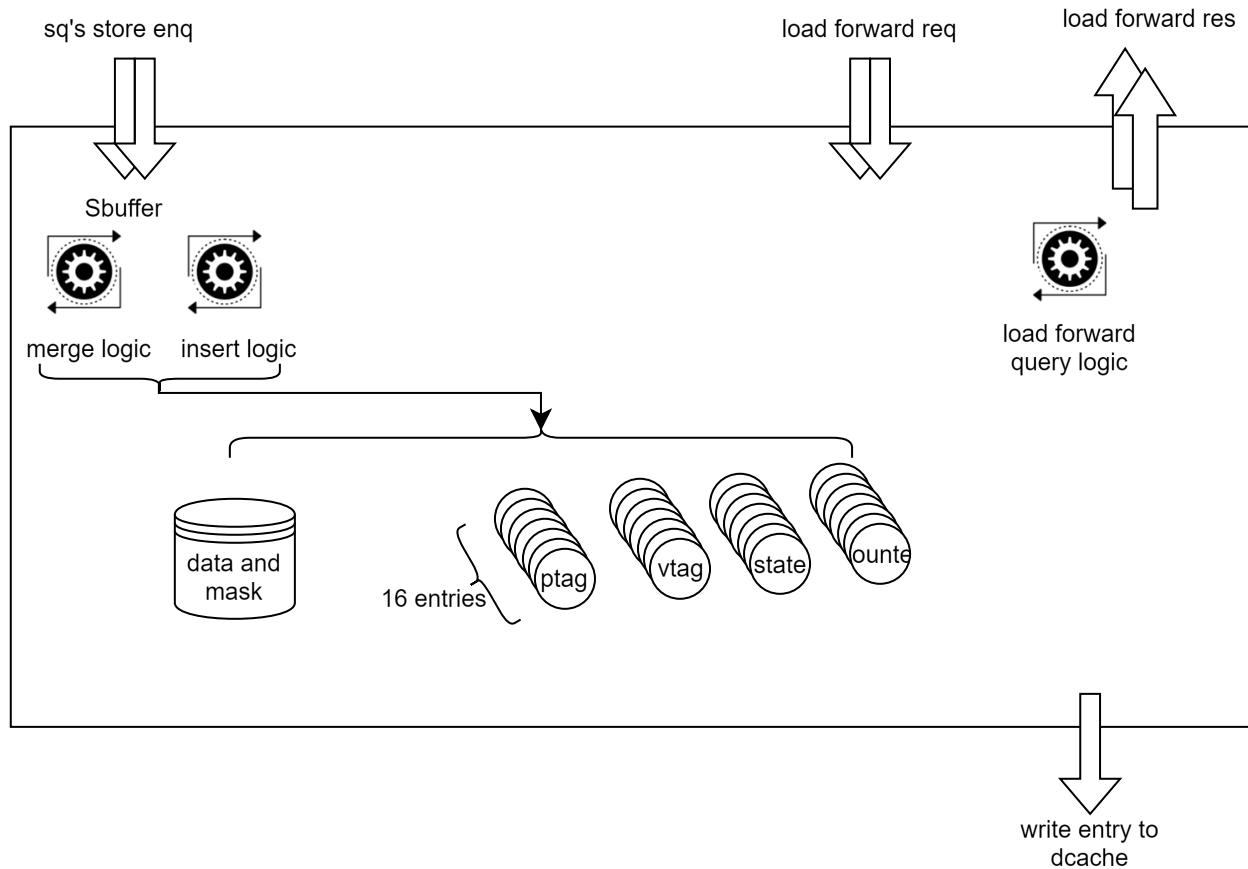


图 16.57: sbuffer 整体框图

16.9.3 接口时序

16.9.3.1 接收 store 指令写入时序实例

当 io_in_valid 与 io_in_ready 握手时, sbuffer 接收到 storeQueue 的写请求, 使用地址去做检查, 要么新分配一项要么合并到已有一项中, 利用 io_in_*_bits 的信息去更新项目。

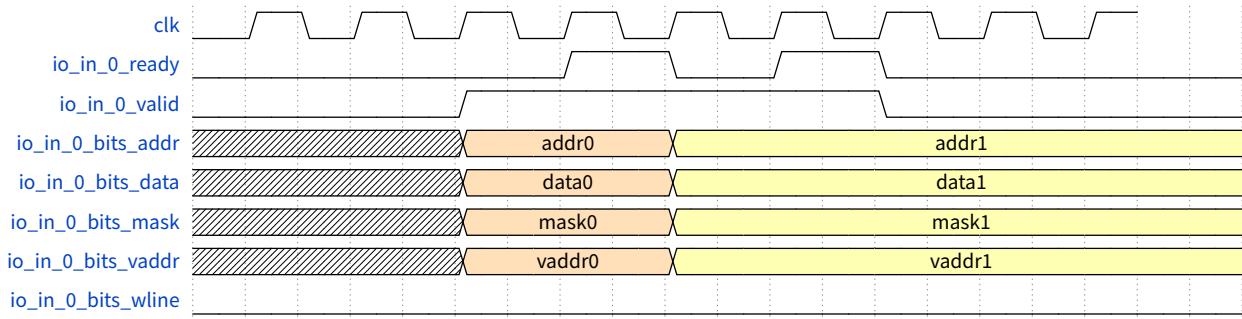


图 16.58: 接收 store 指令写入时序

16.9.3.2 写入到 dcache 时序实例

当 `io_dcache_req_ready` 和 `io_dcache_req_valid` 握手时，将 `io_dcache_req_bits_*` 给到 dcache，将请求传递过去让 dcache 处理。

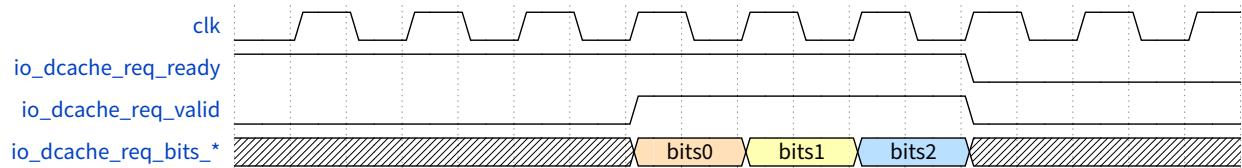


图 16.59: 写入到 dcache 时序

16.9.3.3 前递请求时序实例

前递请求不需要 ready 信号，一旦 `io_forward_*` valid 为高，就需要处理这个请求，利用请求的 paddr 和 vaddr 来进行查询，数据和其他信息在 `io_forward*_valid` 为高的下一拍有效。

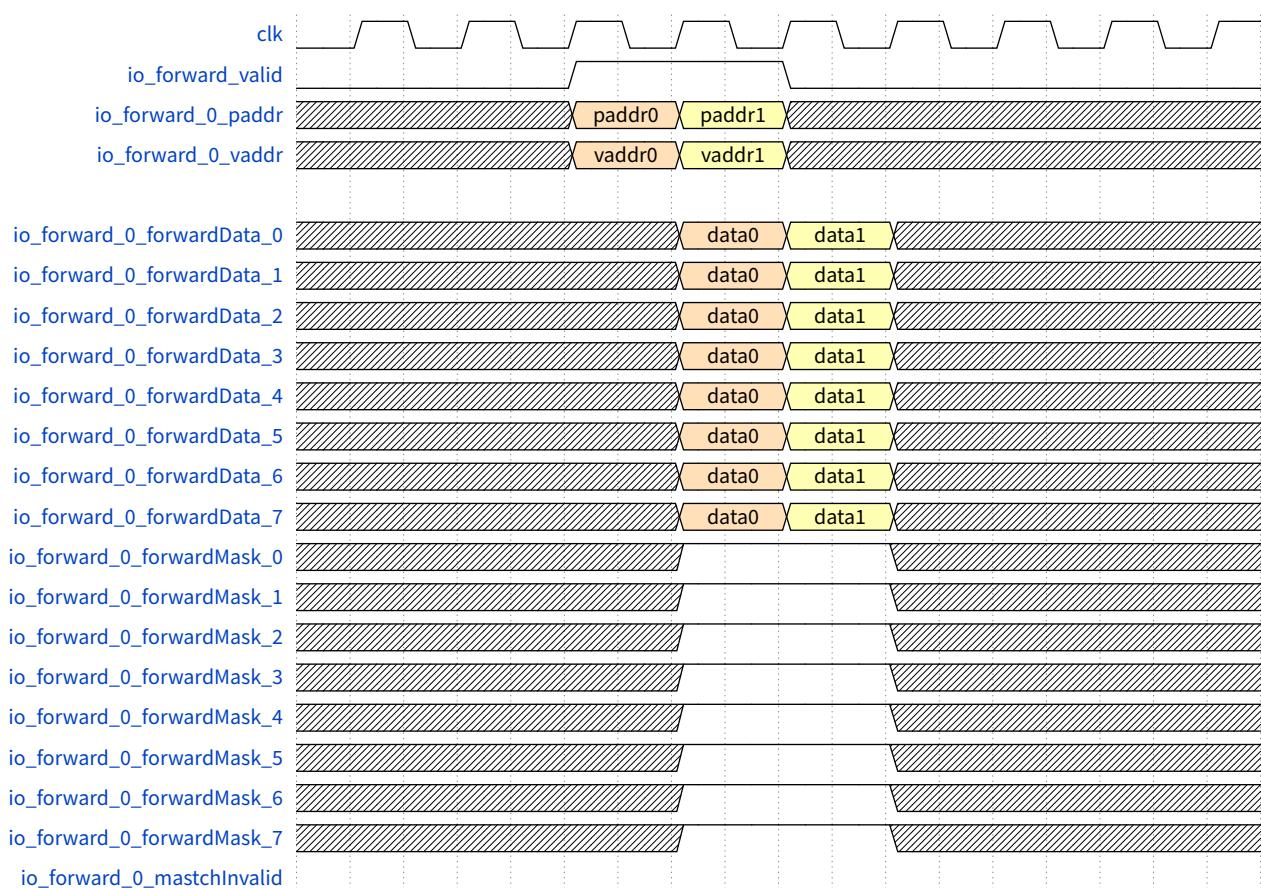


图 16.60: 前递请求时序

16.10 Load 非对齐访存单元 LoadMisalignBuffer

16.10.1 功能描述

LoadMisalignBuffer 中存储 1 条非对齐且跨越 16Byte 边界的 Load 指令。执行逻辑为具有 7 个状态的状态机，当一条指令在 LoadUnit 中检测到非对齐且跨越 16Byte 时，则会申请进入 LoadMisalignBuffer。LoadMisalignBuffer 会锁存住这条 Load，并将其分别拆分成两条 Load 访存 (flow) 重新进入 LoadUnit。

LoadMisalignBuffer 会收集由自身发出的 Load 访存，等两条 Load 访存都执行完成之后，会进行数据拼接，然后向 LoadUnit 再发送一次唤醒操作，该操作不实际进入 LoadUnit 流水线执行，只是触发唤醒信号并打三拍。等三拍后，LoadMisalignBuffer 会再次收到 LoadUnit 的写回请求，并标记为来自唤醒操作，这时，LoadMisalignBuffer 会出队并真正的写回给后端并 bypass。

标量非对齐写回后端需在 LoadUnit 1 未使能标量写回时进行。如不满足，则阻塞 LoadMisalignBuffer 写回后端。向量非对齐写回 VLMergeBuffer 需在 LoadUnit 1 未使能向量标量写回时进行。如不满足，则阻塞 LoadMisalignBuffer 写回 VLMergeBuffer。

16.10.1.1 特性 1：支持跨越 16Byte 边界的非对齐 Load 进行拆分访存

根据已经执行完的 flow 进行不同的变化。转态机会再第一条 flow 写回后再进入 s_req 状态，发送第二条 flow。如果第一次 flow 携带异常写回至 LoadMisalignBuffer，则直接携带异常信息写回给后端，无需进行第二条 flow 的执行。任意一条 flow 写回时都有可能产生任意原因的 replay，LoadMisalignBuffer 选择重新发送该 flow 至 LoadUnit，无论是什么原因的 replay。

- lb 指令永远不可能产生非对齐。
- lh 拆成两个对应的 lb 操作：

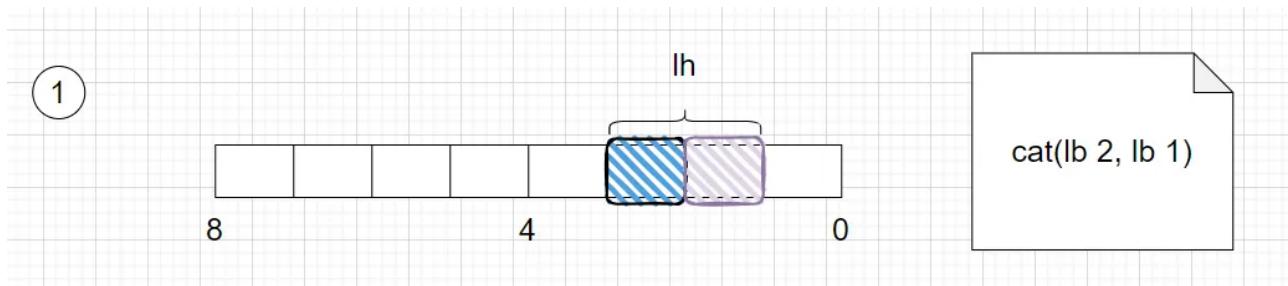


图 16.61: alt text

- lw 根据地址拆分方式不同：

- ld 根据地址拆分方式不同：

16.10.1.2 特性 2：支持向量非对齐

向量非对齐的 flow 与标量非对齐处理方式一致，区别在向量写回至 VLMergeBuffer，而标量直接写回至后端。

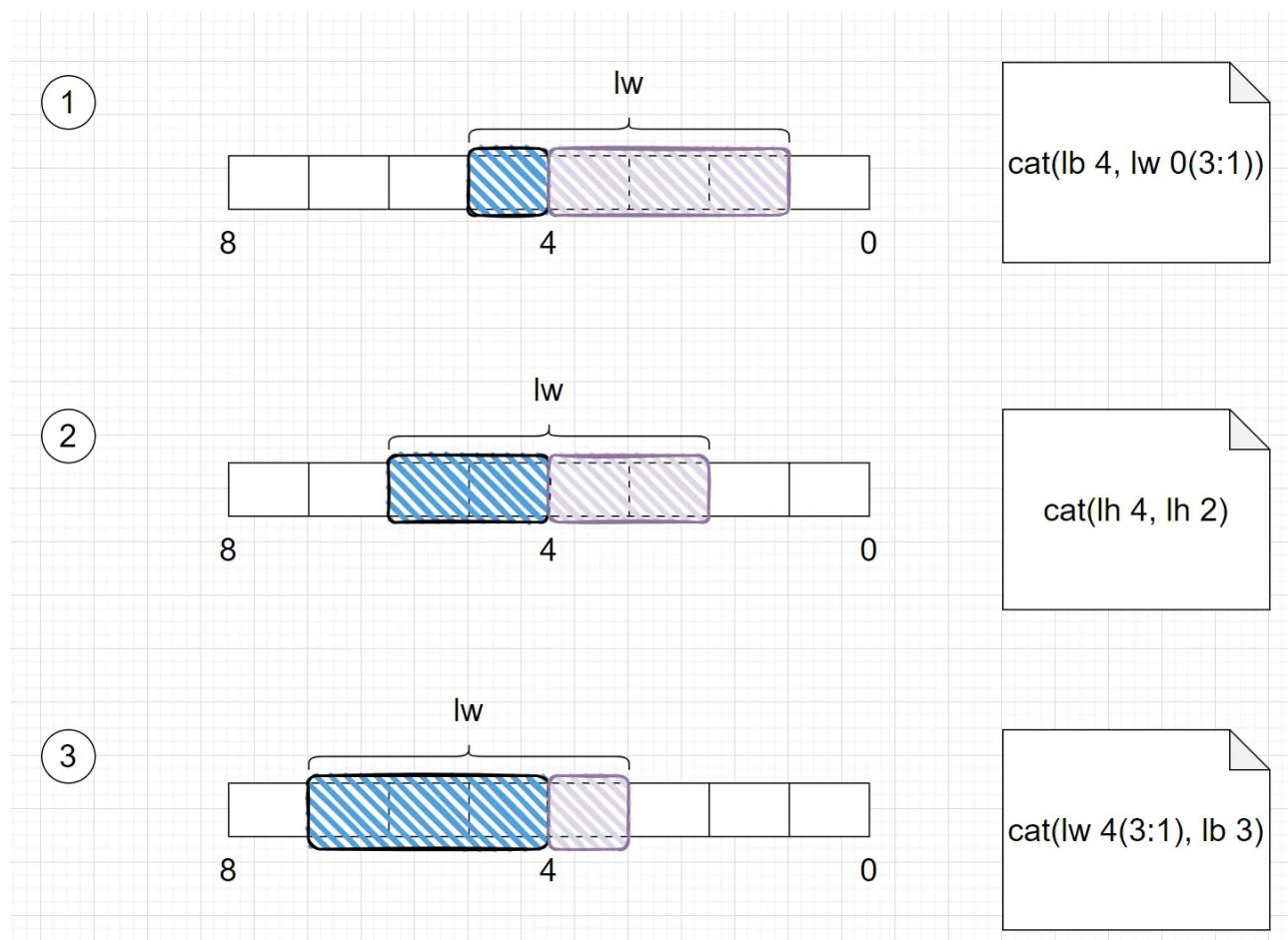


图 16.62: alt text

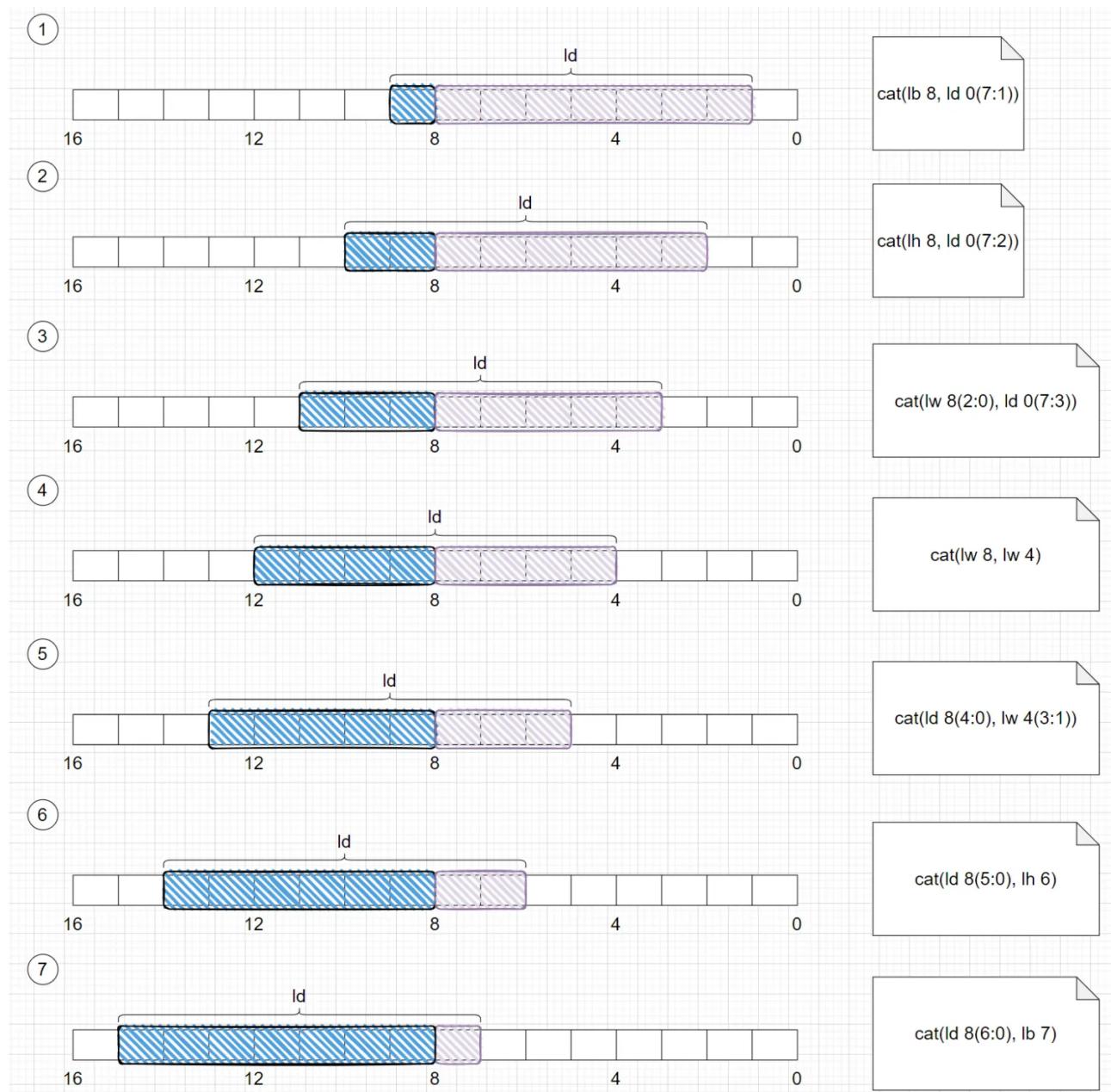


图 16.63: alt text

16.10.1.3 特性 3：不支持非 Memory 空间的非对齐 Load

不支持非 Memory 空间的非对齐 Load，当非 Memory 空间的 Load 产生非对齐时，会产生 LoadAddr-Misalign 异常。

16.10.2 整体框图

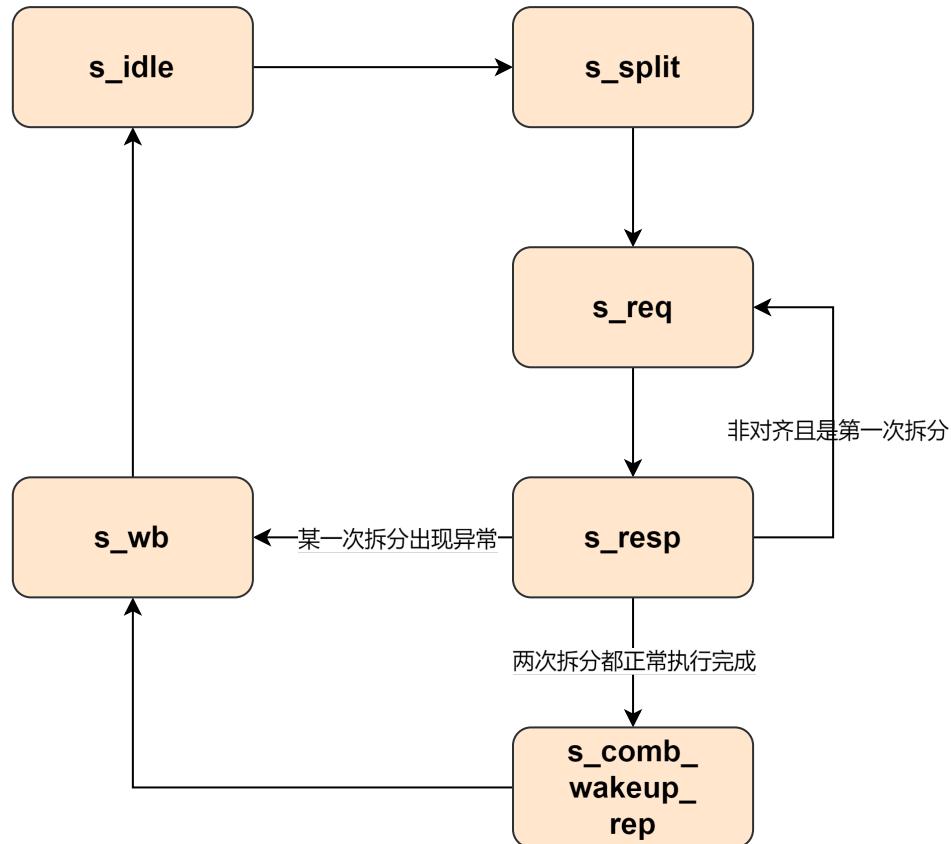


图 16.64: alt text

状态介绍

| 状态 | 说明 |
|--------------------------|--------------------------------|
| s_idle | 等待非对齐的 Load uop 进入 |
| s_split | 拆分非对齐 Load |
| s_req | 发射拆分后的非对齐 Load 操作至 LoadUnit |
| s_resp | LoadUnit 写回 |
| s_comb_wakeup_rep | 合并两条非对齐 Load 的结果，发射 wakeup uop |
| s_wb | 写回后端或 VLMergeBuffer |

16.10.3 主要端口

| | 方向 | 说明 |
|--------------|-----|-------------------------------|
| redirect | In | 重定向端口 |
| req | In | 接收来自 LoadUnit 的入队请求 |
| rob | In | 内部悬空 |
| splitLoadReq | Out | 发送至 LoadUnit 的拆分后的 flow 的访存请求 |

| | 方向 | 说明 |
|------------------|-----|--|
| splitLoadResp | In | 接收 LoadUnit 写回的拆分后的 flow 的访存响应 |
| writeBack | out | 标量非对齐写回至后端 |
| vecWriteBack | Out | 向量非对齐写回至 VLMergeBuffer |
| loadOutValid | In | Load Unit 存在 Load 指令将要写回至后端 |
| loadVecOutValid | In | Load Unit 存在 Vector Load 指令将要写回至 VLMergeBuffer |
| overwriteExpBuf | Out | 悬空 |
| loadMisalignFull | Out | LoadMisalignBuffer 满标记 |

16.10.4 接口时序

接口时序较简单，只提供文字描述。

| | 说明 |
|------------------|--------------------------------------|
| redirect | 具备 Valid。数据同 Valid 有效 |
| req | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| rob | 内部悬空 |
| splitLoadReq | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| splitLoadResp | 具备 Valid。数据同 Valid 有效 |
| writeBack | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| vecWriteBack | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| loadOutValid | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |
| loadVecOutValid | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |
| overwriteExpBuf | 悬空 |
| loadMisalignFull | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |

16.11 Store 非对齐访存单元 StoreMisalignBuffer

16.11.1 功能描述

StoreMisalignBuffer 中存储 1 条非对齐且跨越 16Byte 边界的 Store 指令。执行逻辑为具有 7 个状态的状态机，当一条指令在 StoreUnit 中检测到非对齐且跨越 16Byte 时，则会申请进入 StoreMisalignBuffer。StoreMisalignBuffer 会锁存住这条 Store，并将其分别拆分成两条 Store 访存 (flow) 重新进入 StoreUnit。

StoreMisalignBuffer 会收集由自身发出的 Store 访存，等两条 Store 访存都执行完成之后，如果是非跨页的非对齐，则写回。标量非对齐写回后端需在 StoreUnit 1 未使能标量写回时进行。如不满足，则阻塞 StoreMisalignBuffer 写回后端。向量非对齐写回 VSMergeBuffer 需在 StoreUnit 1 未使能向量标量写回时进行。如不满足，则阻塞 StoreMisalignBuffer 写回 VSMergeBuffer。

对于跨 4K 页的 Store，我们要求当该指令到达 Rob 队头时才能进行执行。如果这期间能老的 Store 进入 StoreMisalignBuffer，则会踢掉当前的跨 4K 页的 Store，并将 needFlushPipe 标记设为 true。当最终某条 Store 写回时，我们会产生一次 redirect。

对于向量，当存在某条向量的 Store flow 被踢出时，会通知 VSMergeBuffer 将该 flow 对应的项置为 needRsReplay，从而让该 uop 被重发。

16.11.1.1 特性 1：支持跨越 16Byte 边界的非对齐 Store 进行拆分访存

根据已经执行完的 flow 进行不同的变化。转态机会再第一条 flow 写回后再进入 s_req 状态，发送第二条 flow。如果第一次 flow 携带异常写回至 StoreMisalignBuffer，则直接携带异常信息写回给后端，无需进行第二条 flow 的执行。任意一条 flow 写回时都有可能产生任意原因的 replay，StoreMisalignBuffer 选择重新发送该 flow 至 StoreUnit，无论是什么原因的 replay。

- sb 指令永远不可能产生非对齐。
- sh 拆成两个对应的 sb 操作：

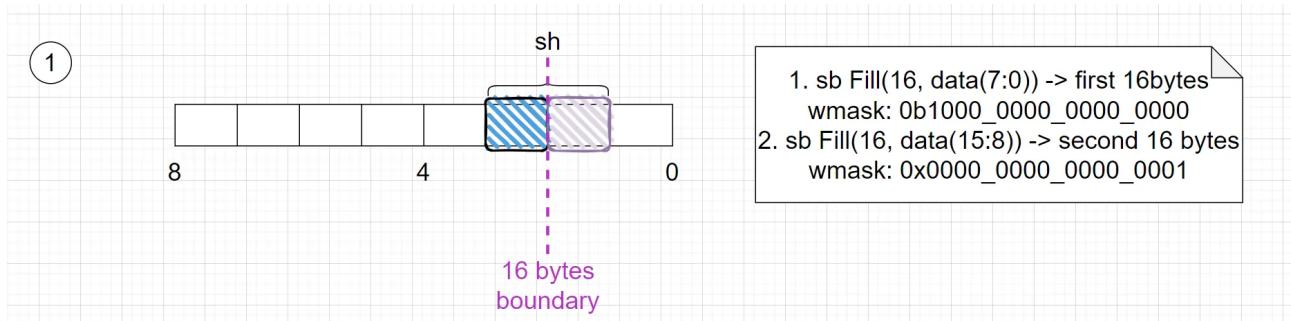


图 16.65: alt text

- sw 根据地址拆分方式不同：

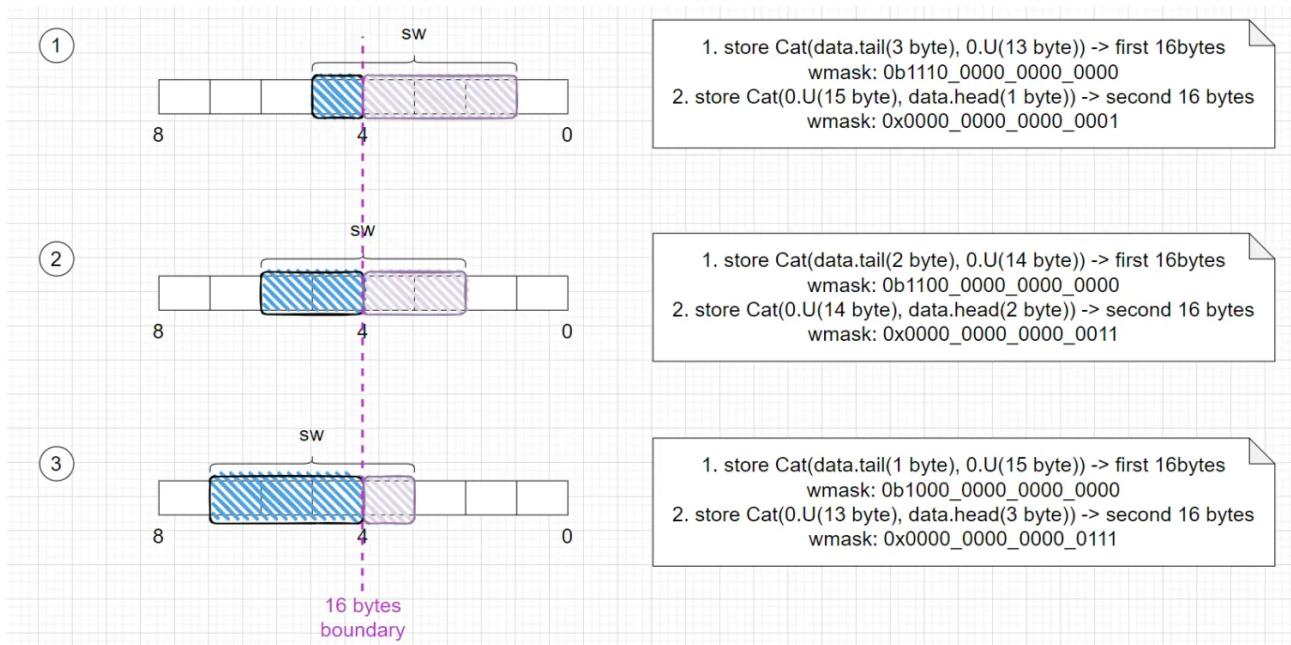


图 16.66: alt text

- sd 根据地址拆分方式不同：

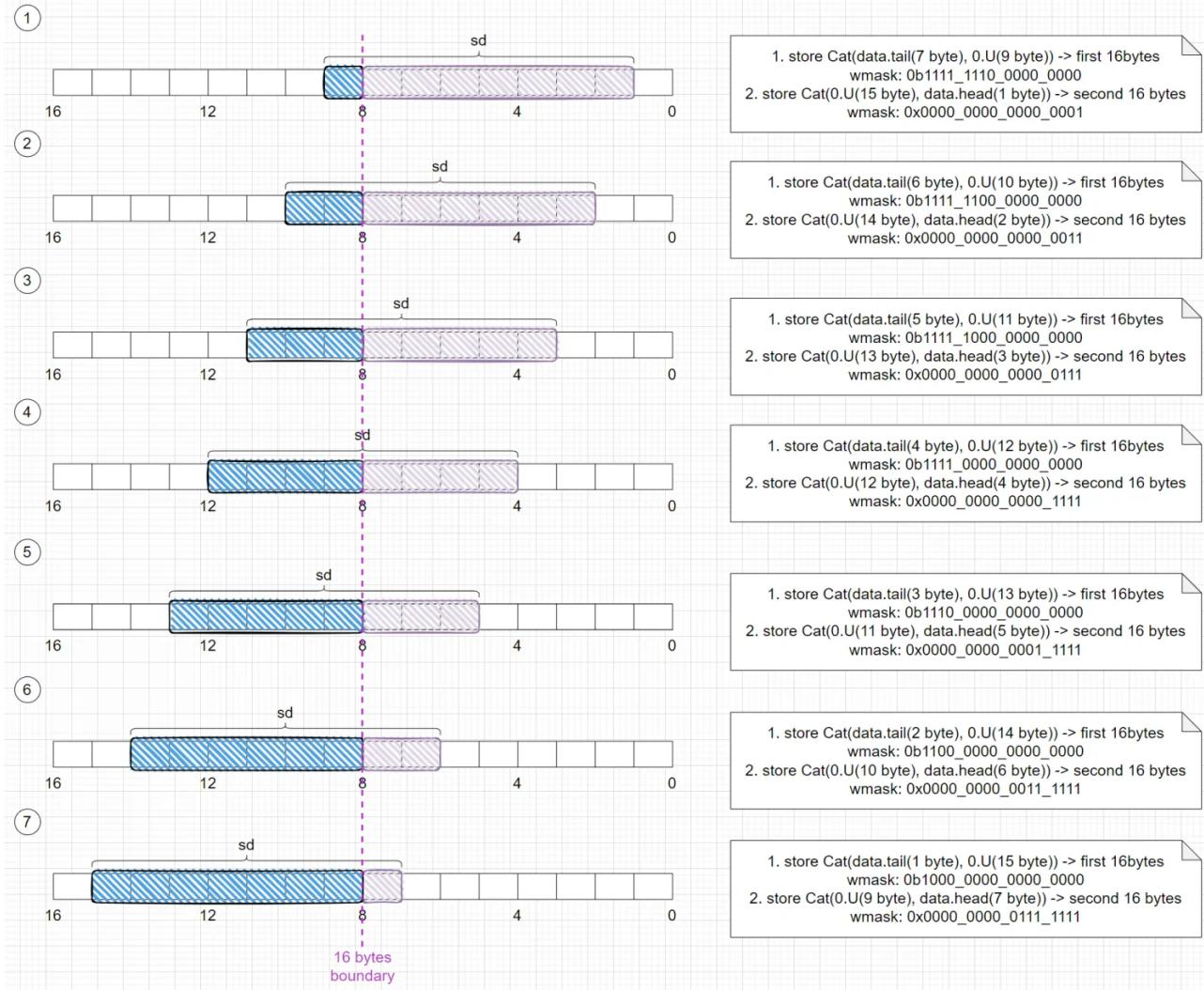


图 16.67: alt text

16.11.1.2 特性 2：支持向量非对齐

向量非对齐的 flow 与标量非对齐处理方式一致，区别在向量写回至 VSMergeBuffer，而标量直接写回至后端。

16.11.1.3 特性 3：不支持非 Memory 空间的非对齐 Store

不支持非 Memory 空间的非对齐 Store，当非 Memory 空间的 Store 产生非对齐时，会产生 StoreAddrMisalign 异常。

16.11.1.4 特性 4：支持跨页 Store

Store 因为需要写入 Sbuffer，在跨页的情况下，会产生两个物理地址。低页的物理地址可以存在 StoreQueue 中，而高页的物理地址需要找一个地方单独存储。我们选择存在 StoreMisalignBuffer 中，这样的话，对于跨页的 Store 来说，我们要等这条指令从 Store Queue 中提交到 Sbuffer 之后才可以在 StoreMisalignBuffer 中清除这一项。因此，我们会为 StoreQueue 提供当前 StoreMisalignBuffer 中锁存的元数据与地址，以供 Store Queue 写回使用。具体的，我们会通过 rob 与 StoreQueue 传来的相关信号来判断是否需要锁存保持住目前的 Store 元数据。

16.11.2 整体框图

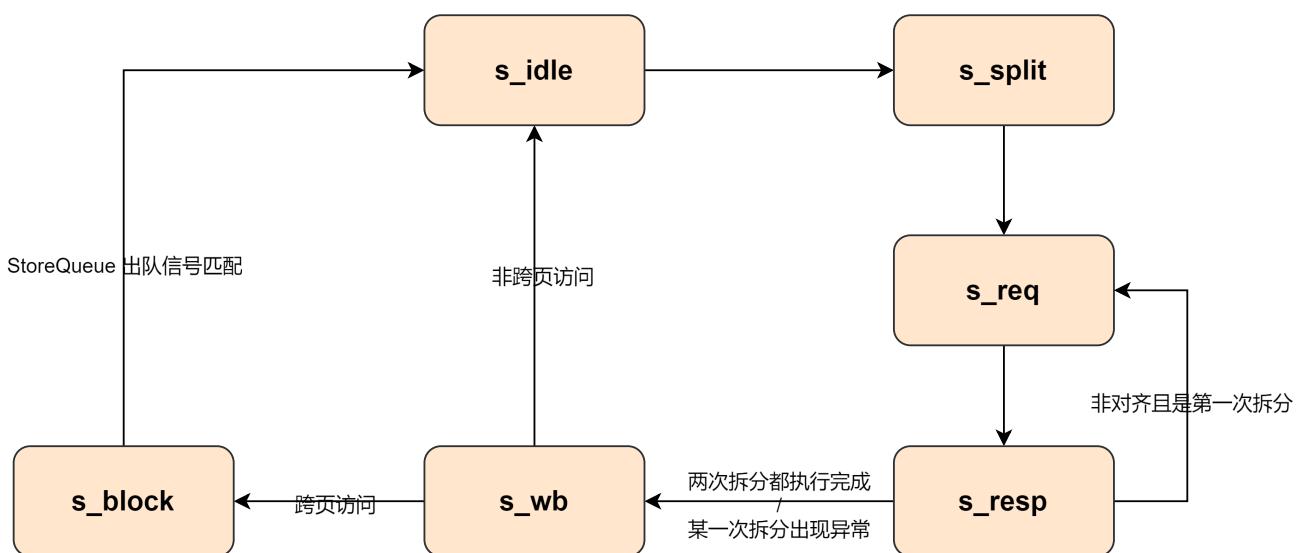


图 16.68: alt text

状态介绍

| 状态 | 说明 |
|---------|-------------------------------|
| s_idle | 等待非对齐的 Store uop 进入 |
| s_split | 拆分非对齐 Store |
| s_req | 发射拆分后的非对齐 Store 操作至 StoreUnit |
| s_resp | StoreUnit 写回 |
| s_wb | 写回后端或 VSMergeBuffer |

| 状态 | 说明 |
|---------|---|
| s_block | 阻塞这条指令出队，直到 Store Queue 中将该项写入到 Sbuffer |

16.11.3 主要端口

| | 方向 | 说明 |
|-----------------------|--------|--|
| redirect | In | 重定向端口 |
| req | In | 接收来自 StoreUnit 的入队请求 |
| rob | In | 接收 Rob 中相关元数据信息 |
| splitStoreReq | Out | 发送至 StoreUnit 的拆分后的 flow 的访存请求 |
| splitStoreResp | In | 接收 StoreUnit 写回的拆分后的 flow 的访存响应 |
| writeBack | out | 标量非对齐写回至后端 |
| vecWriteBack | Out | 向量非对齐写回至 VSMergeBuffer |
| StoreOutValid | In | Store Unit 存在 Store 指令将要写回至后端 |
| StoreVecOutValid | In | Store Unit 存在 Vector Store 指令将要写回至 VSMergeBuffer |
| overwriteExpBuf | Out | 悬空 |
| sqControl | In/Out | 与 Store Queue 的交互接口 |
| toVecStoreMergeBuffer | Out | 将 flush 相关信息发送至 VSMergeBuffer |

16.11.4 接口时序

接口时序较简单，只提供文字描述。

| | 说明 |
|-----------------------|--------------------------------------|
| redirect | 具备 Valid。数据同 Valid 有效 |
| req | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| rob | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |
| splitStoreReq | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| splitStoreResp | 具备 Valid。数据同 Valid 有效 |
| writeBack | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| vecWriteBack | 具备 Valid、Ready。数据同 Valid && ready 有效 |
| StoreOutValid | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |
| StoreVecOutValid | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |
| overwriteExpBuf | 悬空 |
| sqControl | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |
| toVecStoreMergeBuffer | 不具备 Valid，数据始终视为有效，对应信号产生即响应 |

17 数据缓存

17.1 数据高速缓存 DCache

- 版本: V2R2
- 状态: WIP
- 日期: 2025/02/28
- commit: b6c14329cbd4a204593ce03d130052f820439a08

17.1.1 术语说明

| 缩写 | 全称 | 描述 |
|------|------|------|
| TODO | TODO | TODO |

17.1.2 子模块列表

| 子模块 | 描述 |
|-----------------|---------------------|
| BankeddataArray | 数据和 ECC SRAM |
| MetaArray | 元数据寄存器堆 |
| TagArray | Tag 和 ECC SRAM |
| ErrorArray | 错误标志寄存器堆 |
| PrefetchArray | 预取元数据寄存器堆 |
| AccessArray | 访问元数据寄存器堆 |
| LoadPipe | Load 访问 DCache 流水线 |
| StorePipe | Store 访问 DCache 流水线 |
| MainPipe | DCache 主流水线 |
| MissQueue | DCache Miss 状态处理队列 |
| WritebackQueue | DCache 数据写回请求处理队列 |
| ProbeQueue | Probe/Snoop 请求处理队列 |
| CtrlUnit | DCache ECC 注入控制器 |
| AtomicsUnits | 原子指令运算单元 |

17.1.3 DCache 设计规格

| Feature | 描述 |
|-------------|---|
| Data Cache | 64KB, 4way 组相联, 256 组, 每组 8bank Virtually Indexed, Physically Tagged (VIPT) Tag 和每个 bank 采用 SEC-DED ECC |
| Cacheline | 64Bytes |
| Replacement | Pseudo-Least Recently Used (PLRU) |
| 读写接口 | 3*128 bits 读流水线 1*512 bits 写流水线 |

17.1.3.1 Data RAM

对于每个访问 DCache Data 的请求, 对 DCache Data SRAM 的返回数据, 如下表表示的格式。

| 位域 | 描述 |
|----------|--------------------|
| [71, 64] | 64bits 数据 ECC 编码结果 |
| [63, 0] | 64bits 数据 |

17.1.3.2 Tag RAM

对于每个访问 DCache Tag 的请求, 对 DCache Tag SRAM 的返回数据, 如下表表示的格式。

| 位域 | 描述 |
|----------|---------------------|
| [42, 36] | 36bits tag ECC 编码结果 |
| [35, 0] | 36bits tag |

17.1.3.3 Meta

对于每个访问 DCache Meta 的请求, 对 DCache Meta 的返回数据, 如下表表示的格式。

| 位域 | 描述 |
|---------|--|
| [1 : 0] | Cacheline coherence 元数据 2' b00 Nothing 2' b01 Branch 2' b10 Trunk 2' b11 Dirty |

17.1.4 整体框图

DCache 模块整体架构如图 17.1 所示。

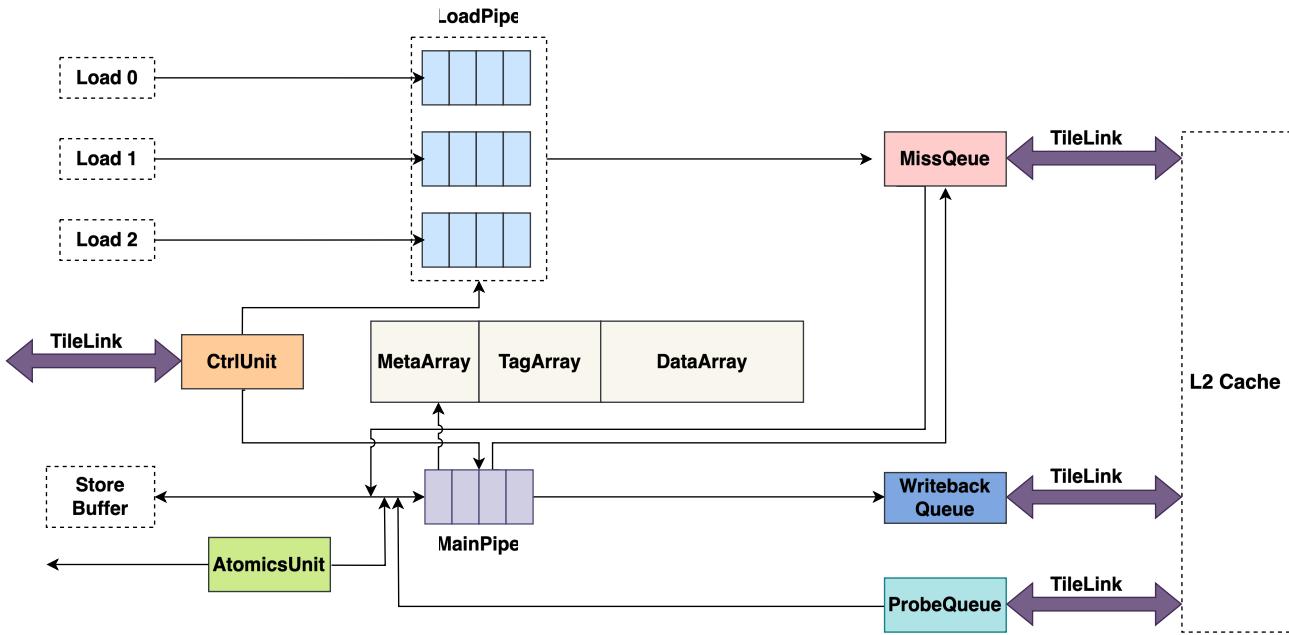


图 17.1: DCACHE 整体架构

17.1.5 功能描述

17.1.5.1 特征 1: Load 请求处理

对于普通的 Load 请求，DCache 从 LoadUnit 接收一条 load 指令后（实现的 Load 流水线有三条，可以并行处理三个 load 请求），根据计算得到的地址查询 tagArray 和 metaArray，比较判断是否命中：若命中缓存行则返回数据响应；若缺失则分配 MSHR (MissEntry) 项，将请求交给 MissQueue 处理，MissQueue 负责向 L2 Cache 发送 Acquire 请求取回重填的数据，并等待 L2 Cache 返回的 hint 信号。当 l2_hint 到达后，向 MainPipe 发起回填请求，进行替换路的选取并将重填数据块写入存储单元，同时把取回的重填数据前递给 LoadUnit 完成响应；若被替换的块需要写回，则在 WritebackQueue 中向 L2 发送 Release 请求将其写回。如果缺失的请求分配 MSHR 项失败，DCache 会反馈一个 MSHR 分配失败的信号，由 LoadUnit 和 LoadQueueReplay 重新调度该 load 请求。

17.1.5.2 特征 2: Store 请求处理

对于普通的 Store 请求，DCache 从 StoreBuffer 接收一条 store 指令后，使用 MainPipe 流水线计算地址查询 tag 和 meta，判断是否命中，若命中缓存行则直接更新 DCache 数据并返回应答；若缺失则分配 MSHR 将请求交给 MissQueue，向 L2 请求要回填到 Dcache 的原目标数据行，并等待 L2 Cache 返回的 hint 信号。当 l2_hint 到达后，向 MainPipe 发起回填请求，进行替换路的选取并将重填数据块写入 DCache 存储单元，在完成对该数据的 store 操作后向 StoreBuffer 返回应答；若被替换的块需要写回，则在 WritebackQueue 中向 L2 发送 Release 请求将其写回。如果缺失的请求分配 MSHR 项失败，DCache 会反馈一个 MSHR 分配失败的信号，由 StoreBuffer 随后重新调度该 store 请求。

17.1.5.3 特征 3: 原子指令处理

对于原子指令，由 DCache 的 MainPipe 流水线完成指令运算及读写操作，并返回响应。若数据缺失则同样向 MissQueue 发起请求，取回数据后继续执行该原子指令；对于 AMO 指令先完成运算操作，再将结果写入；对

于 LR/SC 指令，会设置/检查其 reservation set。在原子指令执行期间，核内不会向 DCache 发出其他请求（参见 Memblock 文档）。

17.1.5.4 特征 4：Probe 请求处理

对于 Probe 请求，DCache 从 L2 Cache 接收 Probe 请求后，进入 MainPipe 流水线修改被 Probe 的数据块的权限，命中后下一拍返回应答。

17.1.5.5 特征 5：替换与写回

DCache 采用 write-back 和 write-allocate 的写策略，由一个 replacer 模块计算决定缺失请求回填后被替换的块，可配置 random、lru、plru 替换策略，默认选择使用 plru 策略；选出替换块后将其放入 WritebackQueue 队列中，向 L2 Cache 发出 Release 请求；而缺失的请求则从 L2 读取目标数据块后填入对应 Cacheline。

17.2 Load 访存流水线 LoadPipe

17.2.1 功能描述

用流水线控制 Load 请求的处理，与 Load 访存流水线紧耦合，经过 4 级流水线读出目标数据或返回 miss/replay 响应

17.2.1.1 特征 1：LoadPipe 各级流水线功能：

- Stage 0: 接收 LoadUnit 中流水线计算得到的虚拟地址：根据地址读 tag 和 meta；
- Stage 1: 获得对应的 tag 和 meta 的查询结果；从 LoadUnit 接收物理地址，进行 tag 比较判断是否命中；根据地址读 data；检查 l2_error；
- Stage 2: 获得对应 data 结果；如果 load miss 则向 MissQueue 发送 miss 请求，尝试分配 MSHR 项；向 LoadUnit 返回 load 请求的响应；校验 tag_error；
- Stage 3: 更新替换算法状态；向 bus error unit 上报 1-bit ecc 校验错误（包括 dcache 发现的 data 错误，dcache 发现的 tag 错误，以及从 L2 获取数据块时已经存在的错误）。

17.2.2 整体框图

LoadPipe 整体架构如图 17.2 所示。

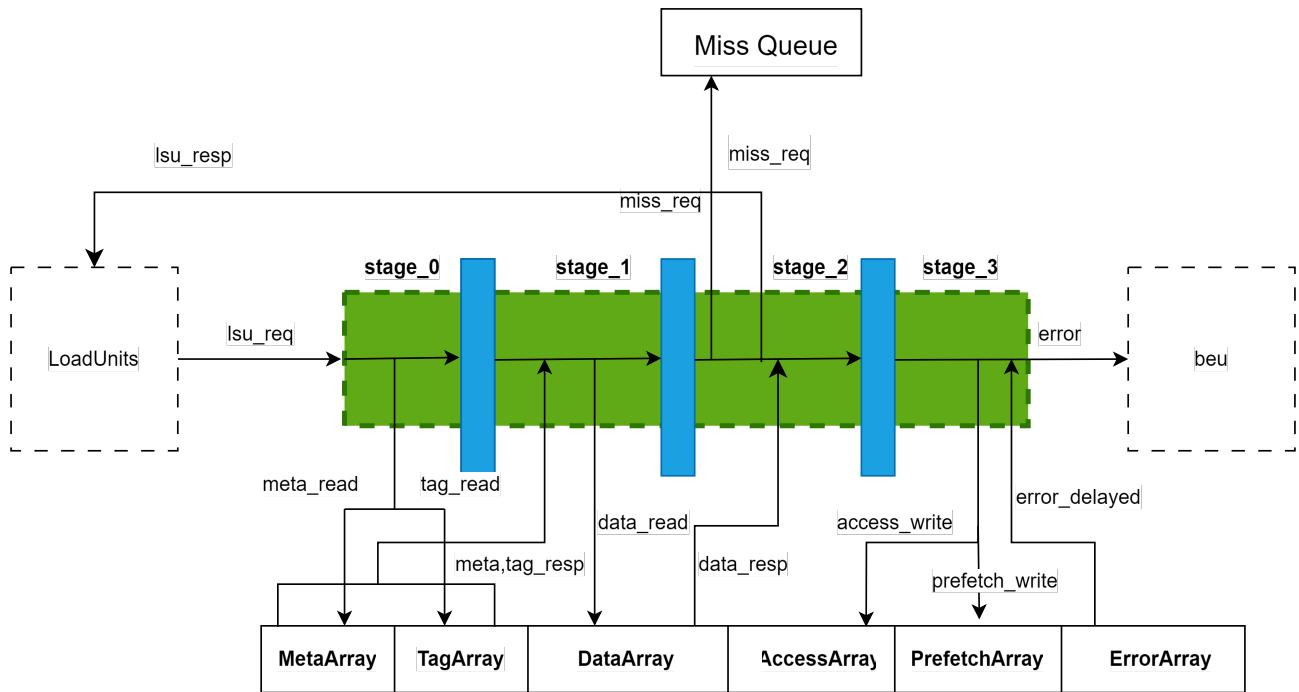


图 17.2: LoadPipe 访问 DCache 示意图

17.2.3 接口时序

17.2.3.1 请求接口时序实例

如图 17.3 所示, req1 第一拍被 LoadPipe 接收, 读 meta 和 tag; 第二拍进行 tag 比较判断 miss; 第三拍向 lsu 返回响应, lsu_resp_miss 拉高表示没有命中, 暂时无法返回数据, 同时向 MissQueue 发出 miss 请求; 第四拍检查报告是否有 ecc 错误。req2 和 req3 紧接着 req1 发出, 同样在 stage_0 被接收, 读 meta 和 tag; 第二拍发现命中, 发出 data 读请求; 第三拍获得 data, 向 lsu 返回带 load 数据的响应; 第四拍更新 PLRU, 报告 ecc 错误。

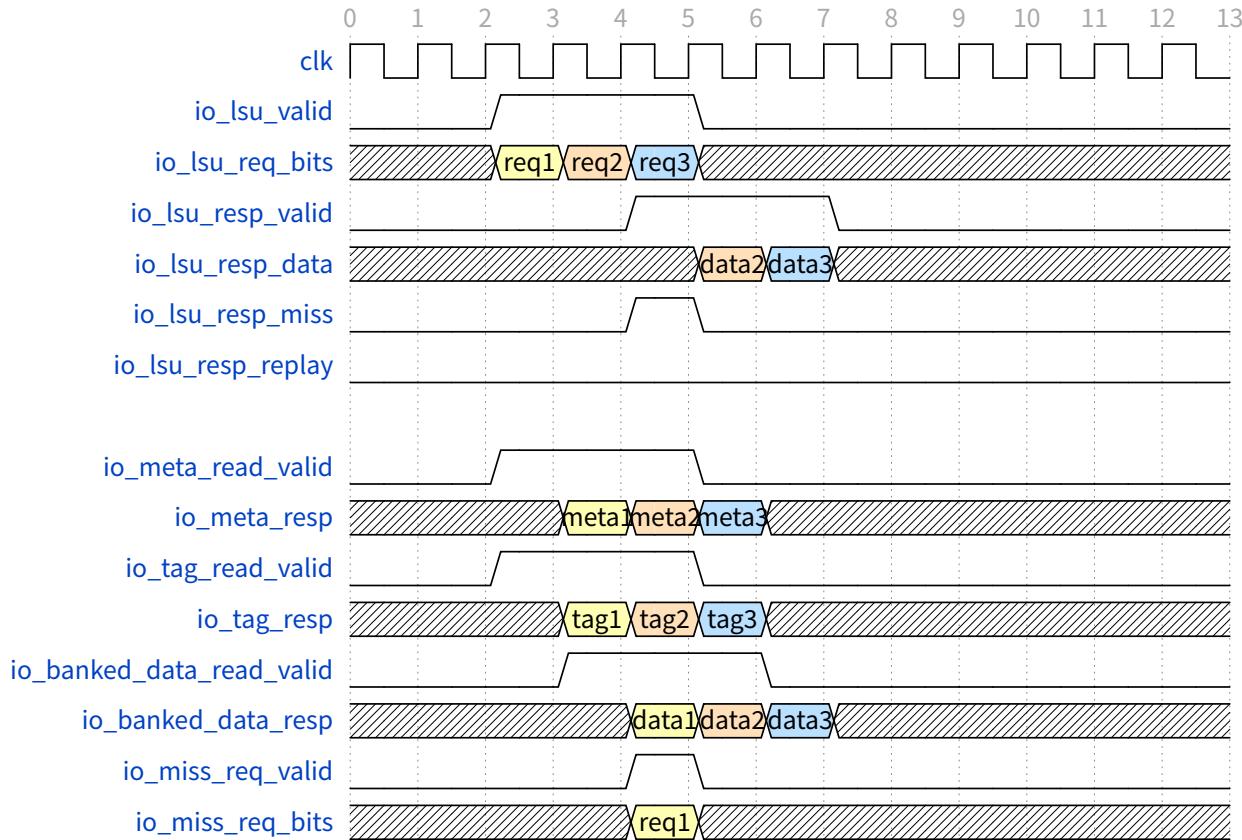


图 17.3: LoadPipe 时序

17.3 缺失队列 MissQueue

17.3.1 功能描述

负责处理 miss 的 load、store 和原子请求，包含 16 项 Miss Entry，每一项负责一个请求，通过一组状态寄存器控制其处理流程。

- miss 的 load 请求：MissQueue 为它分配一项空的 MissEntry，并且可以在一定条件下合并请求或拒绝请求，分配后在 MissEntry 中记录相关信息。进入 MissQueue 的请求会向 L2 发送 Acquire 请求，如果是对整个 block 的覆盖写则发送 AcquirePerm (L2 将会省去一次 sram 读操作)，否则发送 AcquireBlock；等待 L2 返回权限 (Grant) 或者数据加权限 (GrantData)，并在收到 Grant / GrantData 第一个 beat 后向 L2 返回 GrantAck。在收到 L2 返回的结果前，会提前先收到 L2 向上发送的 hint 信号，表明对应的权限和数据会在 2 拍后到达 tilelink D 通道。收到 hint 后 MissQueue 会向 MainPipe 发起回填请求，并在后续收到 Grant / GrantData 后通过前递的方式将回填数据送至 MainPipe 中，等待应答，待完成回填后释放对应的 MissEntry。
- miss 的 store 请求：和 load 的流程基本一致，在最终完成回填后由 MainPipe 向 StoreBuffer 返回应答，表示 store 已完成。
- miss 的原子指令：和 load 的流程基本一致，在最终完成回填后由 MainPipe 向 AtomicsUnit 返回应答，表示原子指令操作已完成。

17.3.1.1 特性 1：MissQueue 入队处理

MissQueue 对于新入队请求，总的操作可分为响应和拒绝，而响应又可以分为分配和合并。MissQueue 支持一定程度的请求合并，从而提高 miss 请求处理的效率。

- 空项分配：如果新的 miss 请求不符合合并或者拒绝条件，则为该请求分配新的 MissEntry。
- 请求合并条件：当已分配的 MissEntry (请求 A) 和新的 miss 请求 B 的块地址相同时，在下述两种情况下可以将请求 B 合并：
 - 向 L2 的 Acquire 请求还没有握手，且 A 是 load 请求，B 是 load 或 store 请求，即 A 还未成功发起对 L2 的读请求前可以合并 B，一起发送 Acquire；
 - 向 L2 的 Acquire 已经发送出去，但是还没有收到 Grant/GrantData，且 A 是 load 或 store 请求，B 是 load 请求，即新来的 load 请求可以在 refill 前合并，而 store 请求只能在 acquire 握手前合并。
- 请求拒绝条件：下述情况下需要将新的 miss 请求拒绝，该请求会在一定时间后重新发出：
 - 新的 miss 请求和某个 MissEntry 中请求的块地址相同，但是不满足请求合并条件；
 - Miss Queue 已满。

17.3.1.2 特征 2：MSHR 给 LoadUnit 数据前递

MissQueue 支持数据前递，如果，lsq 重发信号有效（具体重发逻辑请参考 LoadQueueReplay 部分，选出最合适的最老指令），在 loadUnit 的 stage1，前递指定的 mshrid 以及地址，MissQueue 拿到前递信息后，去比对，如果匹配，直接将重填的数据在 LoadUnits 的 stage2 传给 LoadUnit，通过前递的方式更快地获得先前请求的数据，以减少加载指令的等待时间。

17.3.1.3 特征 3：MissQueue 预取处理

对于进入 MissQueue 的预取请求，会在 MissEntry 内对预取请求的来源进行标记，其余操作与一条普通的 load 指令一致，向 L2 发出 Acquire 请求等待收到 Grant/GrantData 后完成回填。

17.3.1.4 特征 4：MissQueue 中发出的回填请求

为了提升数据回填的效率，可以便于在收到回填数据之后立刻进行 DCache 的写入，因此采用提前向 Main-Pipe 发出回填请求的方式进行完成元数据的读取以及替换路的选取。在收到 L2 返回的 hint 信号后，对应请求所在的 MissEntry 会向 MainPipe 发起回填请求，此时该回填请求中不携带具体的待写回数据，在 MainPipe 进行元数据读取及替换路选择的同时继续等待回填数据的到达。接收到 Grant/GrantData 后，通过前递的方式将回填数据块直接传递到 MainPipe 的 stage2，与提前发出的回填请求进行匹配，完成写回操作。待写回操作完成后接收 MainPipe 传回的 release 信号，释放对应的 MissEntry，完成本次请求。

17.3.2 整体框图

MissQueue 整体架构如图 17.4 所示。

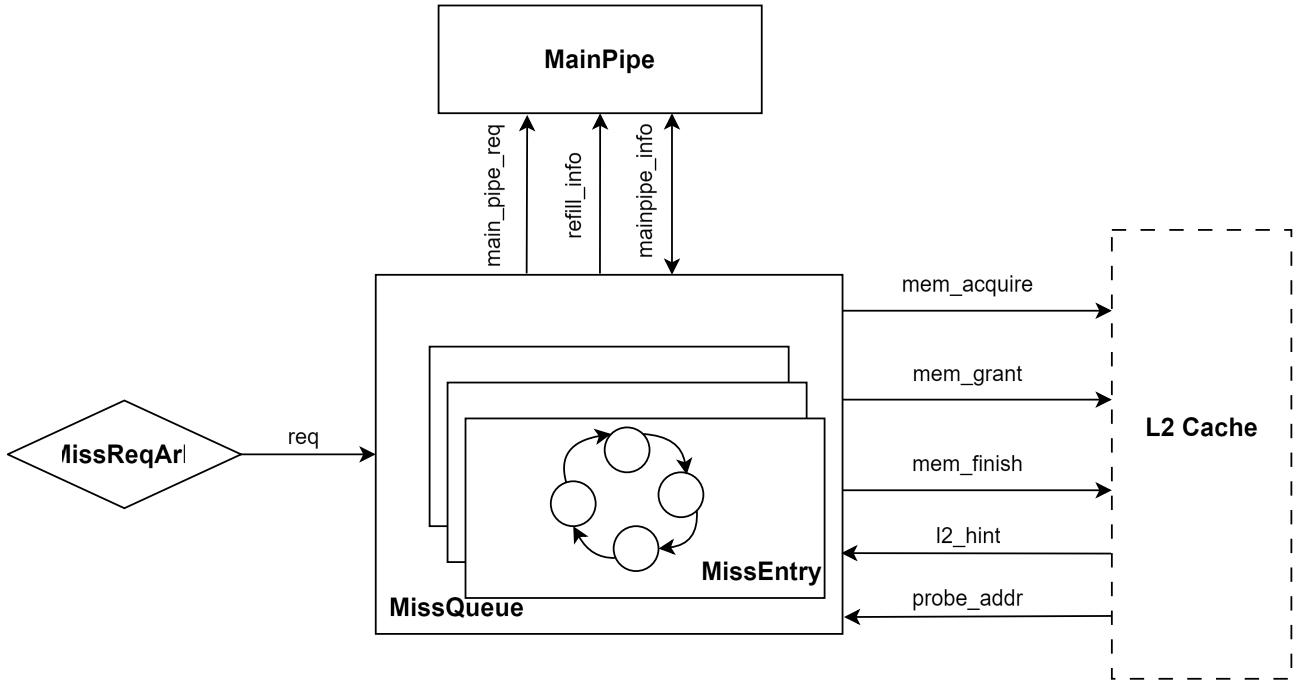


图 17.4: MissQueue 流程图

17.3.3 接口时序

17.3.3.1 请求接口时序实例

图 17.5展示了了一个 load miss 请求进入 MissQueue 之后的接口时序。请求到达后分配 MissEntry，下一拍向 L2 发送 Acquire 请求等待 hint 和数据的响应。接收到 l2_hint 信号后的下一拍向 MainPipe 发起回填请求；接收到 Grant 数据的第一个 beat 之后，向 L2 返回 mem_finish 响应，接收到 Grant 数据的最后一个 beat 之后，下一拍通过 refill_info 将回填数据前递到 MainPipe 的 stage2，完成数据的写入。

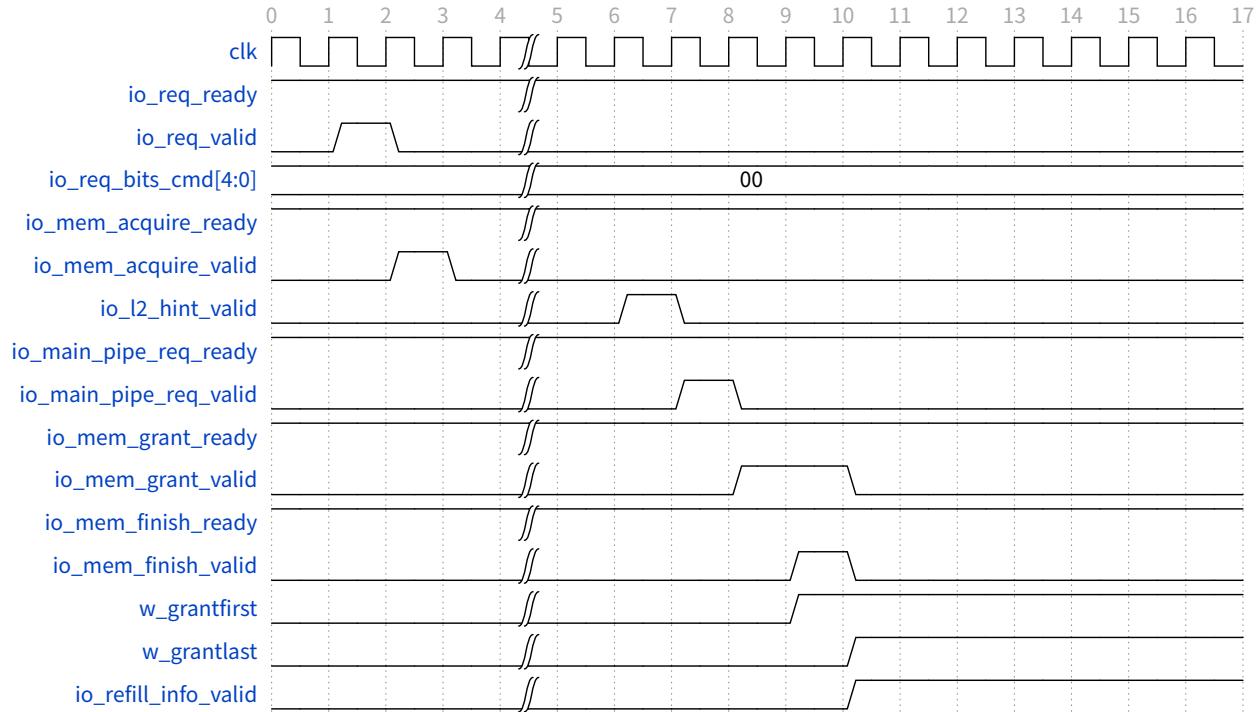


图 17.5: MissQueue 时序

17.3.4 MissEntry 模块

17.3.4.1 特征 1: Miss Entry 分配、合并、拒绝

- 空项分配: 如果新的 miss 请求不符合合并或者拒绝条件, 则为该请求分配新的 Miss Entry。
- 请求合并条件: 当已分配的 Miss Entry (请求 A) 和新的 miss 请求 B 的块地址相同时, 在下述两种情况下可以将请求 B 合并:
 - 向 L2 的 Acquire 请求还没有握手, 且 A 是 load 请求, B 是 load 或 store 请求, 即 A 还未成功发起对 L2 的读请求前可以合并 B, 一起发送 Acquire;
 - 向 L2 的 Acquire 已经发送出去, 但是还没有收到 Grant/GrantData, 且 A 是 load 或 store 请求, B 是 load 请求, 即新来的 load 请求可以在 refill 前合并, 而 store 请求只能在 acquire 握手前合并。
- 请求拒绝条件: 下述情况下需要将新的 miss 请求拒绝, 该请求会在一定时间后重新发出:
 - 新的 miss 请求和某个 Miss Entry 中请求的块地址相同, 但是不满足请求合并条件;
 - Miss Queue 已满。

17.3.4.2 特征 2: MissEntry 状态设计:

Miss Entry 由一系列状态寄存器控制操作的执行, 以及这些操作之间的顺序。s_ 寄存器表示需要调度的请求是否发送, w_ 寄存器表示要等待的应答是否返回, 这些寄存器在初始状态下被置为 true.B, 在为请求分配一项 Miss Entry 时, 会将相应的 s_ 和 w_ 寄存器置为 false.B, 这表示请求还没有发出去, 以及要等待的响应没有握手。表 17.7 和图 17.6 展示了各个寄存器的含义描述以及执行的先后顺序:

表 17.7: MissEntry 状态列表

| 状态 | 描述 |
|-----------------|---|
| s_acquire | 向 L2 发送 AcquireBlock / AcquirePerm 请求 |
| w_grantfirst | 等待接收到 GrantData 的第一个 beat, 拉高表示接收到 |
| w_grantlast | 等待接收到 GrantData 的最后一个 beat, 拉高表示接收到 |
| s_grantack | 在收到 L2 的数据后向 L2 返回应答, 在收到 Grant 的第一个 beat 时就可以返回 GrantAck |
| s_mainpipe_req | 向 Main Pipe 发送回填请求, 将数据回填到 DCache |
| w_mainpipe_resp | 表示将原子请求发送到 Main Pipe 回填入 DCache 后, 接收到 Main Pipe 的应答 |
| w_l2hint | 表示当前 miss 请求已收到 l2_hint 信号, 可以唤醒, 向 MainPipe 发起乎提案请求 |
| w_refill_resp | 表示非原子操作的回填请求已完成, 可以释放 MissEntry |

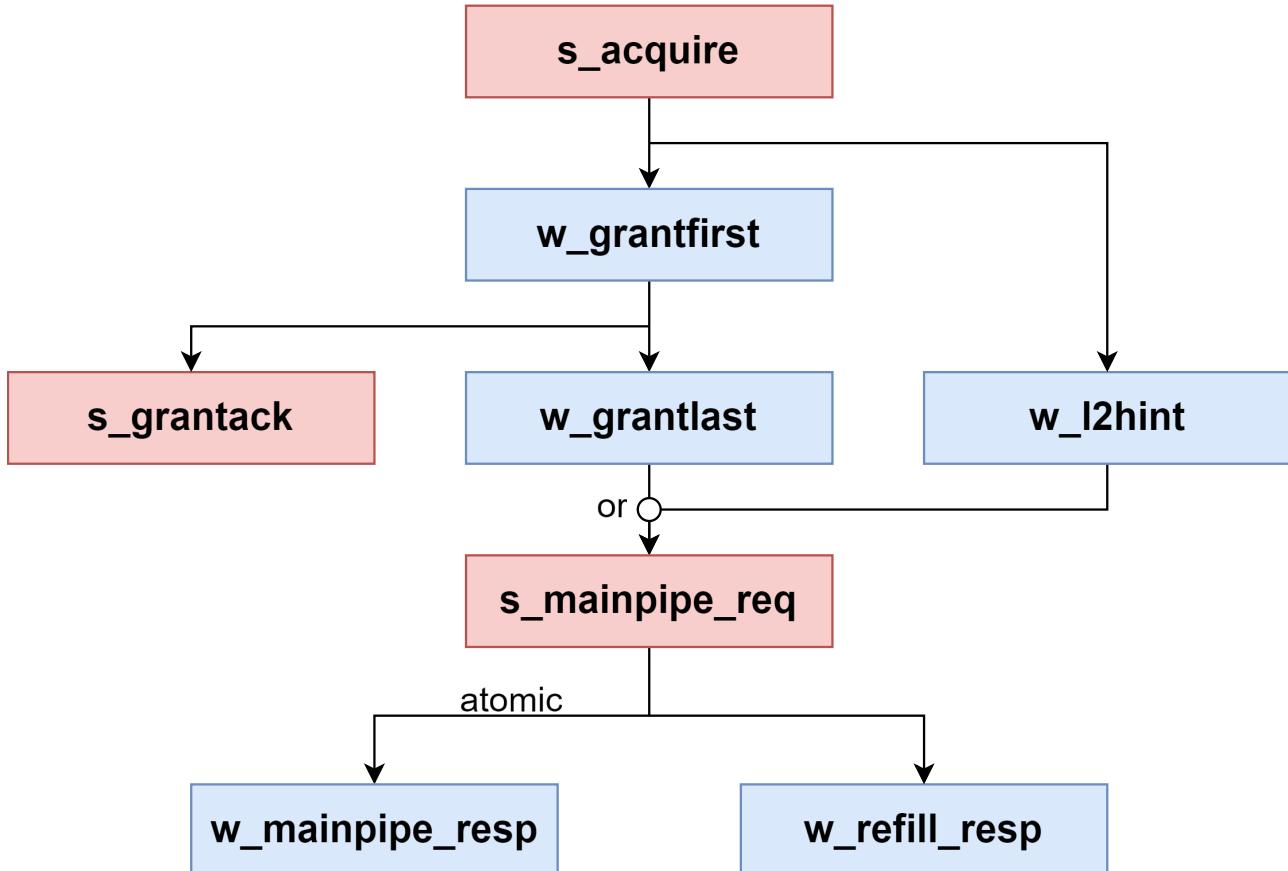


图 17.6: MissEntry 流程图

17.3.4.3 特征 3: MissEntry 别名处理

L1 DCache 支持与 L2 配合处理缓存别名问题，MissEntry 在向 L2 发送 Acquire 请求时会携带请求地址的别名位(vaddr[13:12])，供 L2 保存及判断是否有别名问题需要处理。别名问题的具体处理流程详见 ProbeQueue 一节。

17.4 Probe 队列 ProbeQueue

17.4.1 功能描述

负责接收并处理来自 L2 的一致性请求，包含 8 项 ProbeEntry，每一项负责一个 Probe 请求，将 Probe 请求转成内部信号后发送到 MainPipe，由 MainPipe 修改被 Probe 块的权限，等 MainPipe 返回应答后释放 ProbeEntry。

ProbeQueue 只和 L2 通过 B 通道交互，以及与 MainPipe 互连。内部由 8 项 ProbeEntry 组成，每一项通过一组状态寄存器控制请求信号的接收、转换以及发送。

17.4.1.1 特征 1：别名问题

Kunminghu 架构采用了 64KB 的 VIPT cache，从而引入了 cache 别名问题。为解决别名问题，L2 Cache 的目录会维护在 DCache 中保存的每一个物理块对应的别名位。当 DCache 在某个物理地址上想要获取另一别名位的块时，L2 Cache 会发起 Probe 请求，将 DCache 中原有的别名块 probe 下来，并且在 TileLink B 通道中记录其别名位。ProbeQueue 收到请求后会将别名位和页偏移部分进行拼接，转成内部信号发送到 MainPipe，由 MainPipe 访问 DCache 存储模块读取数据。

17.4.1.2 特征 2：由原子指令引发的阻塞

由于原子操作（包括 lr-sc）在 DCache 中完成，执行 LR 指令时会保证目标地址已经在 DCache 中，此时为了简化设计，LR 在 MainPipe 中会注册一个 reservation set，记录 LR 的地址，并阻塞对该地址的 Probe。为了避免带来死锁，MainPipe 会在等待 SC 一定时间后不再阻塞 Probe（由参数 LRSCCycles 和 LRSCBackOff 决定），此时再收到 SC 指令则均被视为 SC fail。因此，在 LR 注册 reservation set 后等待 SC 配对的时间里需要阻塞 Probe 请求对 DCache 进行操作。

17.4.2 整体框图

ProbeQueue 整体架构如图 17.7 所示。

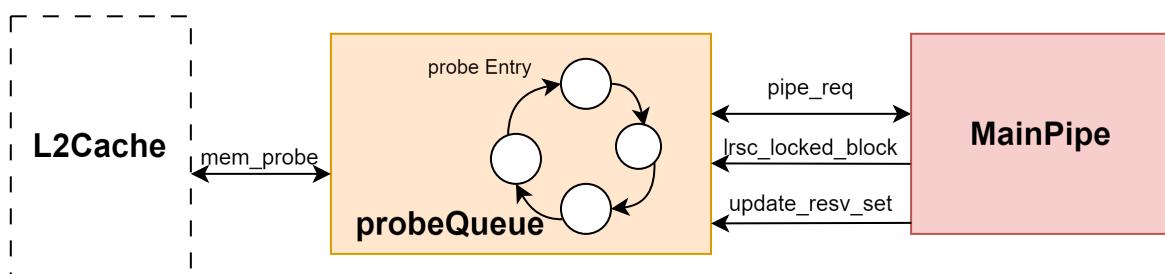


图 17.7: ProbeSnoop 流程图

17.4.3 接口时序

17.4.3.1 请求接口时序实例

图 17.8 展示了 Probe Queue 处理一个 probe 请求的接口时序，Probe Queue 首先收到来自 L2 的 probe 请求，转换成内部请求并为其分配一项空的 ProbeEntry；经过一拍的状态转换可以向 MainPipe 发送 probe 请

求,但由于时序考虑该请求会再被延迟一拍(ProbeQueue里选择一项有一个arbiter, MainPipe入口也有一个arbiter选择各来源的请求,两次仲裁在一拍完成比较困难,因此在这里先锁存一拍),因此两拍后pipe_req_valid拉高;后续等接收到MainPipe的resp后,释放ProbeEntry。

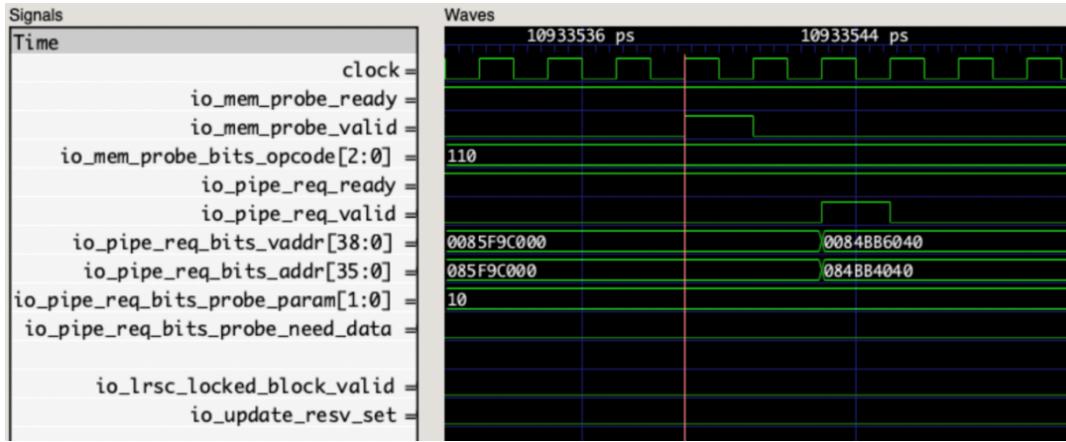


图 17.8: ProbeSnoop 时序

17.4.4 ProbeEntry 模块

Probe Entry 由一系列状态寄存器进行控制,由一个状态机进行 Probe 事务的执行。表 17.8展示了每个 Entry 中包含的三个状态寄存器的含义,状态机设计如图 17.9所示:

表 17.8: ProbeEntry 状态寄存器含义

| 状态 | 描述 |
|-------------|--------------------------------------|
| s_invalid | 复位状态,该 Probe Entry 为空项 |
| s_pipe_req | 已分配 Probe 请求,正在发送 Main Pipe 请求 |
| s_wait_resp | 已完成 Main Pipe 请求的发送,等待 Main Pipe 的应答 |

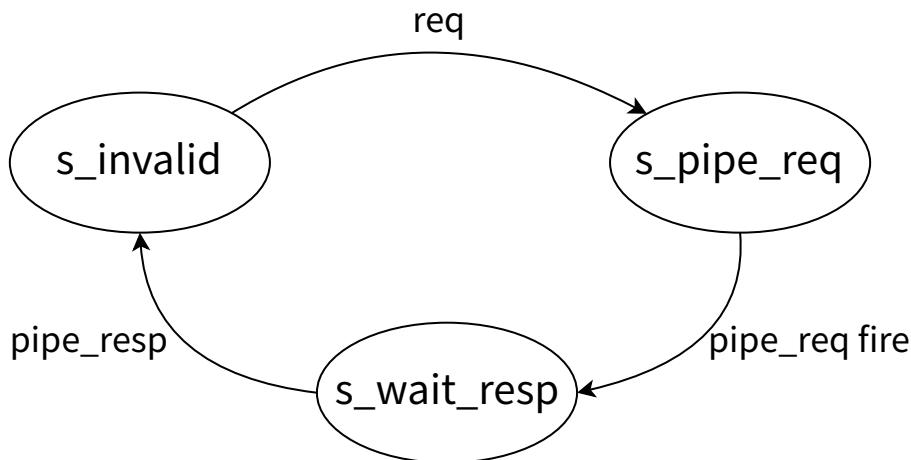


图 17.9: ProbeEntry 状态机

17.5 主流水线 MainPipe

17.5.1 功能描述

用流水线控制 Store, Probe, Refill 以及原子操作的执行 (即所有需要争用 WritebackQueue 向下层 cache 发起请求/写回数据的指令)。

17.5.1.1 特性 1: MainPipe 各级流水线完成的功能:

- Stage 0: 仲裁传入的 MainPipe 请求, 选出优先级最高者; 根据请求信息判断请求所需的资源是否就位; 发出 tag, meta 读请求
- Stage 1: 获得 tag, meta 读请求的结果; 进行 tag 匹配检查, 判断是否命中; 如果需要替换, 获得 PLRU 提供的替换选择结果; 根据读出的 meta 进行权限检查; 提前判断是否需要执行 MissQueue 访问
- Stage 2: 获得读 data 的结果, 与要写入的数据拼合; 如果 miss, 尝试将这次请求信息写入 MissQueue; 检查 tag_error 和 l2_error
- Stage 3: 根据操作的结果, 更新 meta, data, tag; 如果命中则向 lsu 返回 store 响应; 如果指令需要向下层 cache 发起访问/写回数据, 则在这一级生成 WritebackQueue 访问请求, 并尝试写 WritebackQueue; 检查 data_error; 对原子指令的特殊支持: AMO 指令在这一级停留两拍, 先完成 AMO 指令的运算操作, 再将结果写回到 dcache 并返回响应; LR/SC 指令会在这里设置/检查其 reservation set。

17.5.1.2 特性 2: mainpipe 争用和阻塞:

MainPipeline 的争用存在以下优先级: probe_req > refill_req > store_req > atomic_req。一个请求只有在其所请求的资源全部就绪, 不存在 set 冲突, 且没有比它优先级更高的请求的情况下才会被接受. 来自 committed store buffer 的写请求由于时序原因, 拥有单独的检查逻辑。

17.5.1.3 特征 3: set 阻塞逻辑:

确保并行执行的指令不会同时访问到同一个 set 中的不同行, 以维护数据一致性和正确性即防止 s3 (或者 s1,s2) 处理的数据还没写完, s0 进来的数据没读对这类情况发生。在各个阶段 valid 有效的情况下, MainPipe 的 set 冲突要比对 s0 和 s1 的, s0 和 s1, s0 和 s2 的地址索引是否一致, 如果一致则是出发了 set 冲突。阻塞 s0。

17.5.1.4 特征 4: meta 更新

meta 的更新在 s3 中进行, Main Pipe 中四种不同类型的请求中都需要在 MainPipe 中更新对应 cacheline 的 meta data。几种请求都通过端口 meta_write 进行更新, 但具体的行为不同。

Probe 请求在 s3 中根据请求中携带的 probe_param 参数生成需要写入的 meta_coh, 对应本次 Probe 请求希望进行的权限修改。

对于命中的 Store 和 AMO 请求, 在 s1 获取对应数据块的 coh, 在 s2 生成本次访问后的 new_coh, 在 s3 中对比二者决定本次是否需要进行 meta 的写入, 如果需要进行写入则更新为 s2 中生成的 new_coh。

对于第一次请求时发生 miss, 后续由 MissQueue 回填重新进入 MainPipe 的请求, 在 s3 中根据 MissQueue 回填请求中携带的 Acquire 相关的参数生成本次回填访问需要更新的 miss_new_coh, 进行 meta 的写入。

17.5.1.5 特征 5: AMO 指令处理

AMO 请求经过优先级的争用后进入 MainPipe，在前两个流水级与其他几个类型指令的执行流程基本一致，在 s1 获得 tag, meta 读请求的结果，进行 tag 的匹配检查与 meta 的权限检查，判断是否 amo_hit，决定该条 AMO 请求是否需要进入 MissQueue。如果当前的 AMO 请求未命中缓存，则在 s2 阶段将这次请求信息写入 MissQueue；若本次 AMO 请求命中，则会在 s2 获取读 data 的结果，然后继续进入 s3。进入 s3 后 AMO 指令会在这一级停留两拍，第一拍先执行 AMO 指令的运算操作，第二拍将对结果的修改更新写回 dcache，向原子指令处理单元返回响应。

对于 LR/SC 指令，会在 s3 阶段设置/检查其 reservation set，对 lrsc_count 的计数进行更新，维护执行的正确性，防止执行过程中被打断或死锁情况的出现。

17.5.1.6 特征 6: MainPipe 写回

MainPipe 的写回请求在 s3 发起，对于需要向下层 L2 Cache 发起访问/写回数据的指令，通过向 WritebackQueue 发起写回请求，后续经 WritebackQueue 处理后完成真正完成向 L2 Cache 的写入，MainPipe 中需要进行写回的请求共三类。

对于 MissQueue 发回的 refill 请求，如果回填的数据需要替换的数据块当前在 dcache 中为一个有效的数据块（非 Nothing），则该数据块需要被 release 到 L2 Cache，在 s3 会被尝试写入 wbq。

对于 Probe 请求，需要向下级缓存返回 ProbeAck，因此需要向 wbq 写入请求；如果被 Probe 的数据块中含有脏数据的话，需要将其写回下级 L2 Cache，回复 ProbeAckData，也需要向 wbq 发送写回请求。

对于 miss 的 AMO 请求，需要进行写回操作的情形与 refill 的流程类似，miss 的 AMO 请求在回填后重新进入 MainPipe 流水线，此时如果回填的数据块需要替换掉一个有效的数据块时，该数据块需要被 release 到下级 cache 中，会在 s3 生成向 wbq 的写回请求。

17.5.1.7 特征 7: MainPipe 回填数据异常处理

当前所有的回填请求均由 MissQueue 接收到 hint 信号后向 MainPipe 提前发起，回填数据块在 MainPipe 处理请求至 s2 时通过 refill_info 前递获得。因此可能出现由于 l2_hint 与回填数据间隔异常的现象，导致请求进入 s2 后对应的 MSHR 并未能前传有效的回填数据，对于这种异常情况采取下面的两种处理措施。

为了保证回填的效率，减少 replay 的次数，在 s2 阶段允许额外一拍数据未到达的容错，当回填请求进入 s2 阶段后若检查 refill_info 无效 (s2_req_miss_without_data)，则可以额外阻塞一拍，等待下一拍回填数据到达进行后续流程。

若阻塞一拍后，仍未能收到有效的回填数据，则通过 s2_replay_to_mq 通知对应的 MSHR 进行 refill_req 的重发，当前请求退出 MainPipe，不再进行后续的数据操作。

对于缓存别名和其他一些特殊的情况下，可能导致一次回填请求需要替换掉的 Cacheline 此时在一项有效的 MSHR 中，正在等待向 L2 Acquire 请求的响应。为了保证正确性与手册的规范，此时这条替换操作不能进行，对于该回填请求也通过 s2_replay_to_mq 通知对应的 MSHR 进行 refill_req 的重发，当前请求退出 MainPipe，不再进行后续的数据操作。

17.5.2 整体框图

MainPipe 整体架构如图 17.10 所示。

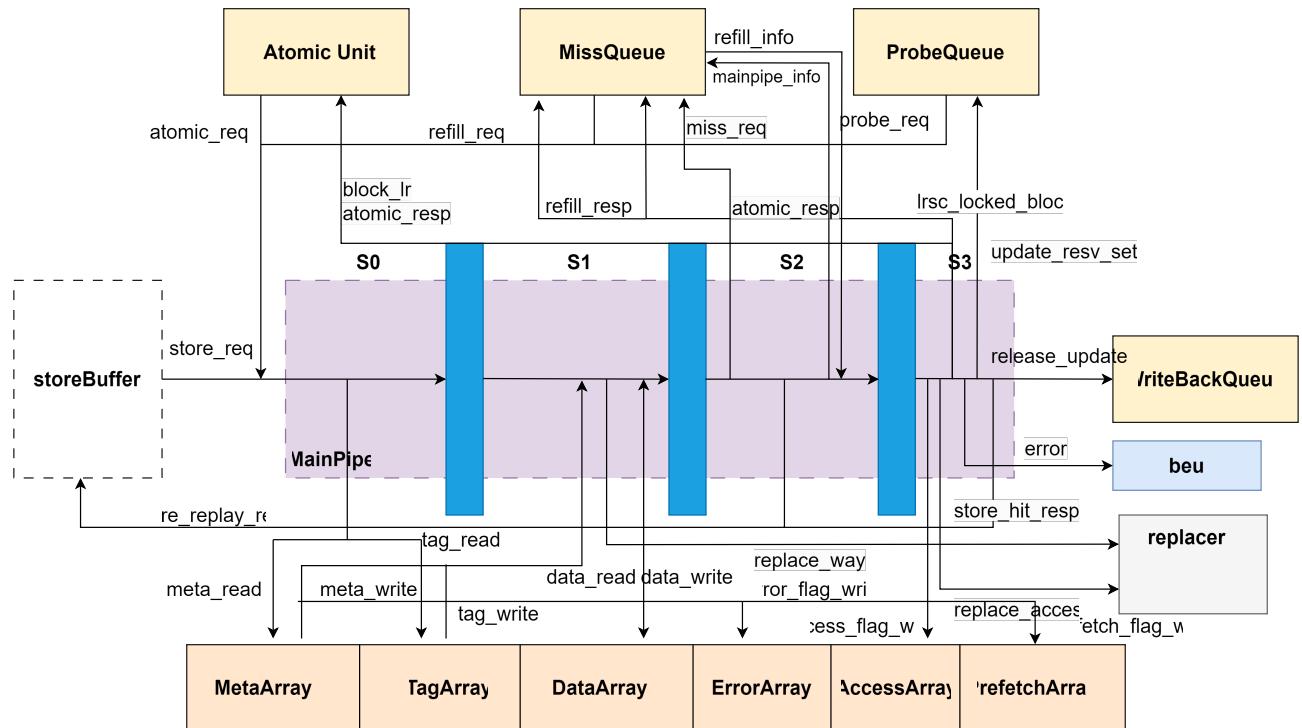


图 17.10: MainPipe 访问 DCACHE 示意图

17.5.3 接口时序

17.5.3.1 请求接口时序实例

接口时序如图 17.11 所示, req1 为 store 请求, 第一拍读 meta 和 tag, 第二拍进行 tag 比较发现请求 miss, 根据替换算法选出要替换的路, 第三拍将 miss 请求发送给 MissQueue, 第四拍因为 miss, 不会向 StoreBuffer 返回响应。req2 为 probe 请求, 第一拍读 meta 和 tag, 第二拍读 data, 第三拍获取 probe 数据块结果, 第四拍根据 probe 命令更新 meta, 并向 WritebackQueue 发起 wb 请求, 返回 probeAck 应答。req3 是 amo 指令, 第一拍读 meta 和 tag, 第二拍进行 tag 比较命中, 发出 data 读请求, 第三拍获得 data 结果, 第四拍和第五拍都处于 stage_3 流水级, 第四拍执行指令运算, 第五拍发出 data 写操作更新原数据块内容, 并向 AtomicsUnit 返回响应。req4 为 req1 对应的 refill 请求, MissQueue 发来 refill_req 的第一拍读 meta, 由于此时 req2 正在进行 meta 写, 而 metaArray 写优先于读, req4 在 stage_0 停留一拍, 下一拍才能成功握手; 第三拍 stage_1 读 data, 同时获得 PLRU 提供的替换选择结果, 由于此时 req3 正在进行 data 写, 再在 stage_1 停留一拍; 第五拍 stage_2 获得要被替换的数据块 data, 同时从 MissQueue 获得前递过来的回填数据; 第六拍 stage_3 向 WritebackQueue 发起 wb 请求, 尝试让替换块进入 wb 队列, 同时将回填的数据写入存储单元, 并向 MissQueue 返回 refill 完成响应。

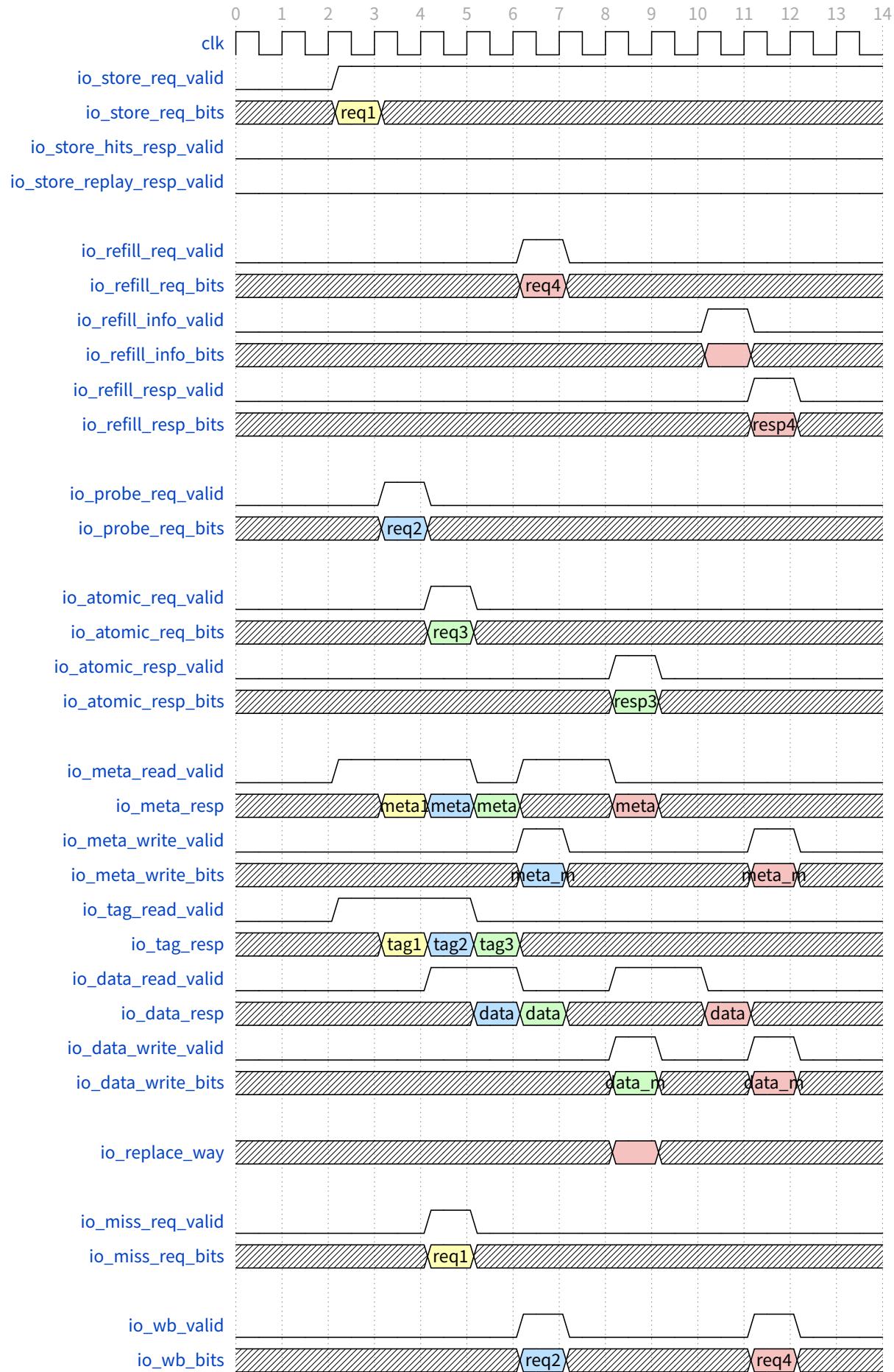


图 17.11: MainPipe 时序

17.6 写回队列 WritebackQueue

17.6.1 功能描述

Writeback Queue 包含 18 项 WritebackEntry, 负责通过 TL-C 的 C 通道向 L2 Cache 写回替换块 (Release), 以及对 Probe 请求做出应答 (ProbeAck)。

17.6.1.1 特性 1: WritebackQueue 空项分配与拒绝

为了时序考虑, 在 wbq 满的时候新请求会被拒绝; 而当 wbq 不满的时候所有请求都会被接收, 此时为新请求分配空项。当前版本中不再支持 WritebackQueue 中请求的合并。

17.6.1.2 特性 2: 请求阻塞条件

TileLink 手册对并发事务的限制要求如果 master 有 pending Grant(即还没有发送 GrantAck), 则不能发送相同地址的 Release. 因此所有 miss 请求在进入 MissQueue 时如果发现和 WritebackQueue 中某一项有相同地址, 则该 miss 请求会被阻塞.

17.6.2 整体框图

WritebackQueue 整体架构如图 17.12 所示。

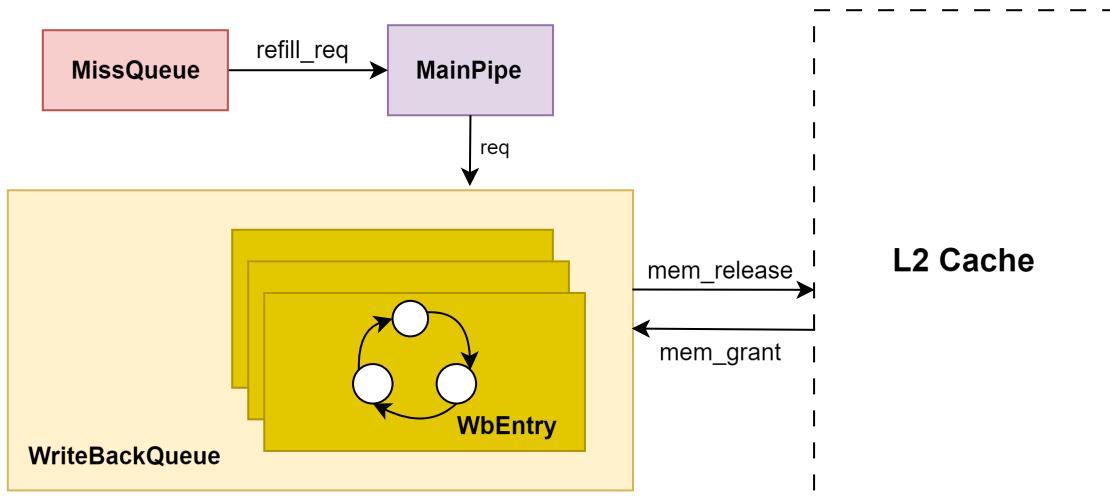


图 17.12: WritebackQueue 流程图

17.6.3 接口时序

17.6.3.1 请求接口时序实例

图 17.13 展示了一个需要写回 L2 的请求在 WritebackQueue 上的接口时序。

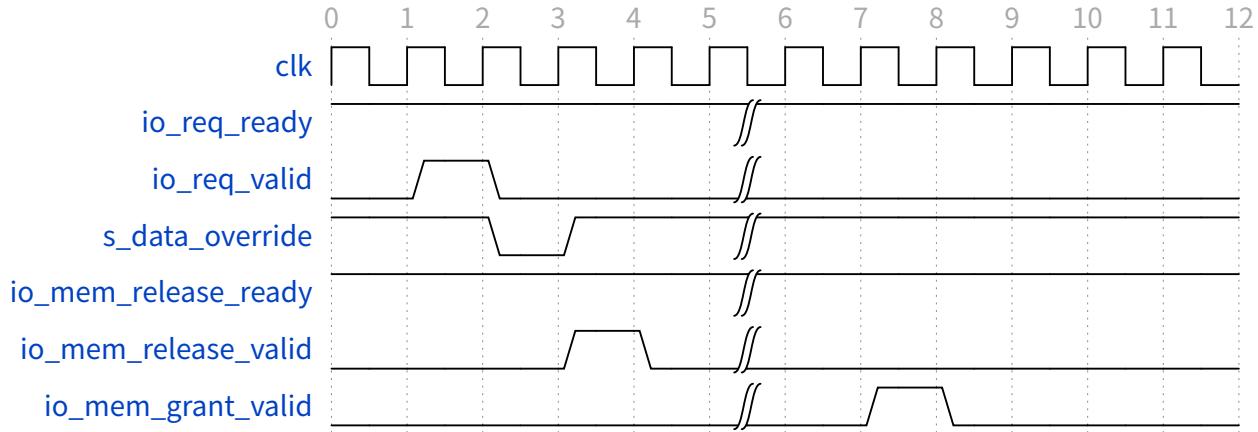


图 17.13: WritebackQueue 时序

17.6.4 WritebackEntry 模块

17.6.4.1 WritebackEntry 状态机设计

状态设计：WritebackEntry 中的状态机设计如表 17.9 和图 17.14 所示：

表 17.9: WritebackEntry 状态寄存器含义

| 状态 | 描述 |
|----------------|-----------------------------|
| s_invalid | 复位状态，该 WritebackEntry 为空项 |
| s_release_req | 正在发送 Release 或者 ProbeAck 请求 |
| s_release_resp | 等待 ReleaseAck 请求 |

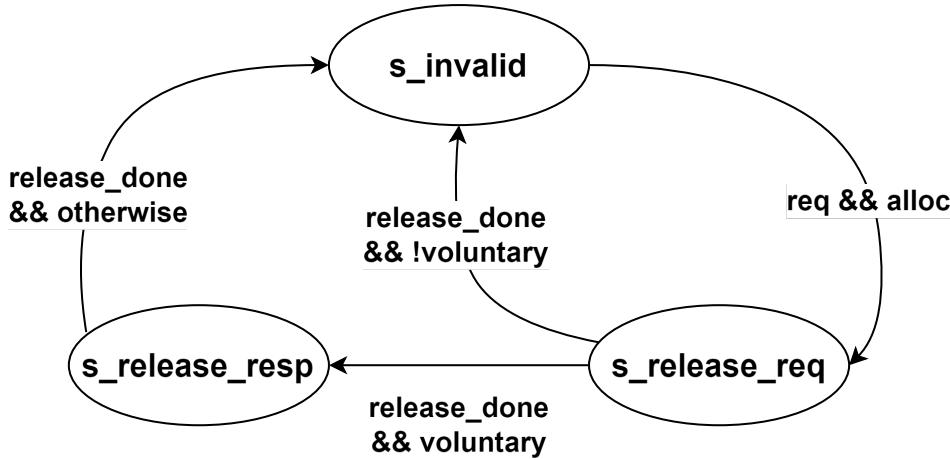


图 17.14: WriteBackEntry 状态机示意图

17.7 错误处理与自定义故障注入指令

17.7.1 功能描述

CtrlUnit 用于控制 DCache 的 ECC 错误注入。每一个核的 L1DCache 配置一个 memory map 的寄存器控制的控制器，每一个支持 ECC 的硬件单元设置一个 Control Bank。通过 MMIO 访存指令读写 CtrlUnit 中的配置寄存器。寄存器配置完成之后，L1 DCache 会对第一次读取 DCache 触发 ecc 错误（比如 load 指令或者 MainPipe）。

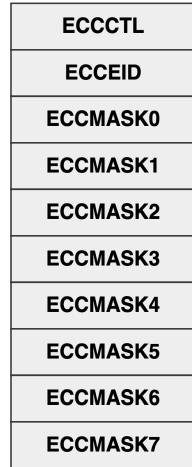


图 17.15: CtrlBank 排布

- ECCCTL (ECC Control): ECC 注入控制寄存器

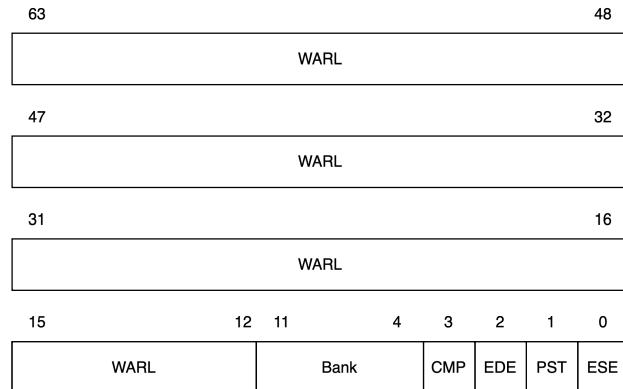


图 17.16: ECCCTL

- ese (error signaling enable): 表示注入有效，初始化为 0。当注入成功后，ese 将拉低。
- pst: 支持注入信号。当 pst=1 时，ECCEID 计数器减到 0 并且成功注入后，注入计时器会被恢复到上一次设置的 ECCEID，重新注入；当 pst==0 时，只注入一次。
- ede (error delay enable): 表示 counter 有效，初始化为 0。
 - * ese==1 并且 ede==0，则 error 注入立即有效。
 - * ese==1 并且 ede==1，则需要等到 ECCEID 递减到 0 之后，注入才有效。
- cmp (component): 表示注入对象，初始化为 0。
 - * 1' b0: 注入对象为 tag
 - * 1' b1: 注入对象为 data
- bank: bank 有效信号，初始化为 0，bank 中的位置位时，对应 mask 有效
- ECCEID (ECC Error Inject Delay): ECC 注入延迟控制器。
 - 当 ese==1 并且 ede==1 时，开始递减，直至减为 0。目前采用和核频率相同的时钟，也可以分频。由于 ECC 注入依赖 DCache 的访问，所以 EID 的时间和 ECC 错误触发的时间可能不一致。

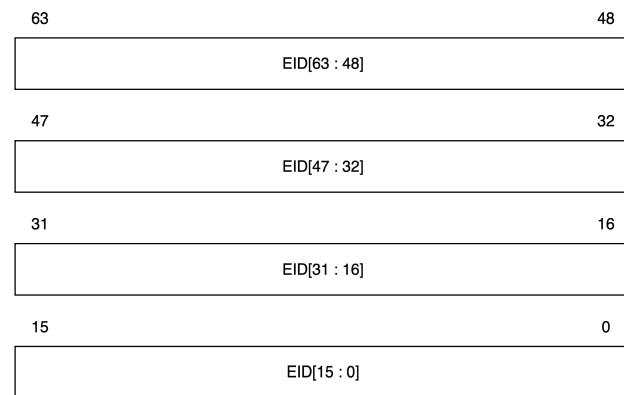


图 17.17: ECCEID

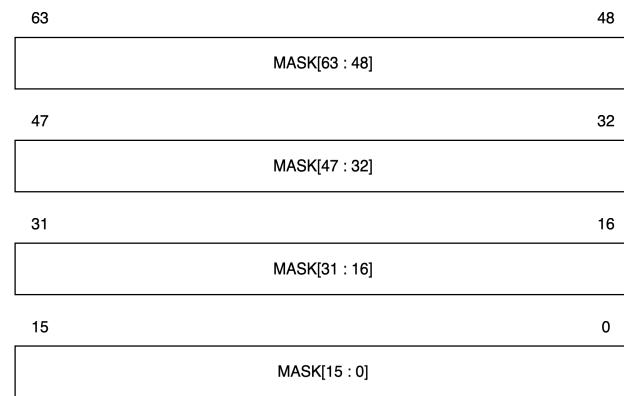


图 17.18: ECCMASK

- ECCMASK (ECC Mask): ECC 注入掩码寄存器。
 - 0 表示不反转, 1 表示翻转。tag 注入只使用 ECCMASK0 中对应 tag 长度的位, 超出部分不起作用。

17.7.1.3 特性 3: Bus Error Unit 控制器

- DCache 的 ECC 错误将统一发送到 Bus Error Unit 控制器处理。Bus Error Unit 控制器保存信息有:

表 17.10: Bus Error Unit 保存的信息

| Field | 描述 | 初始值 | 地址 |
|------------------|---------------|-----|------------|
| cause | 错误事件的原因 | 0 | 0x38010000 |
| value | 错误事件的物理地址 | 未定义 | 0x38010008 |
| enable | 事件有效掩码 | 1 | 0x38010010 |
| global_interrupt | 全局中断使能掩码 | 0 | 0x38010018 |
| accrued | 累积事件掩码 | 0 | 0x38010020 |
| local_interrupt | Hart 本地中断启用掩码 | 0 | 0x38010028 |

- 地址空间

Bus Error Unit 物理地址空间为: 0x38010000 - 0x38010fff
- 支持错误类型
 - * ICache Ecc Error
 - * DCache Ecc Error
 - * L2Cache Ecc Error
- 控制的中断
 - * 局部中断: 只能报告给 Bus Error Unit 所在的 Hart, 上报至后端, 有后端负责中断处理, 目前采用 NMI_31 中断。
 - * 全局中断: 如果出现全局中断,Bus Error Unit 将中断信息发送给 PLIC, 由 PLIC 负责上报中断。

17.7.1.4 特性 4: L1 DCache Ecc 错误处理流程

- 报告错误
 - Tag ECC 错误: 只要某一路出现 ECC 错误, 就判断出现了 ECC 错误。

表 17.11: Tag ECC 错误与 Tag 命中关系

| Hit | Error | Tag Error |
|-----|----------------------|------------------|
| N | N | N |
| N | Y | Y (probably hit) |
| Y | N | N |
| Y | Y(hit with error) | Y |
| Y | Y(hit with no error) | N |

表中 Tag Hit 和 Tag ECC Error 与判断结果之间的关系

- * Data ECC 错误：命中行如果出现 ECC 错误，则认为出现 ECC 错误，如果不命中则不处理。
- * 如果指令访问触发 ECC 错误，则认为出现 Hardware error 并报告异常。
- * 只要出发错误，都需要向 BEU 发送错误信息。硬件检测到错误时，报送给 BEU，触发 NMI 外部中断
- 普通访存指令
 - 对于普通的访存指令，例如 Load 指令，在执行时只会触发 tag 或者 data 的 ECC 错误，并将错误报送给 BEU，并且报告 Hardware Error(19)。
- Probe/Snoop
 - 对于 Probe/Snoop
 - * 如果出现 tag ecc error，不需要更改 cache 状态，并且需要向 l2 返回 corrupt=1 的 ProbeAck 请求。
 - * 如果出现 data ecc error，按规则更改 cache 状态，如果需要返回数据，则需要向 l2 返回 corrupt=1 的 ProbeAckData 请求。
- Replace/Evict
 - 对于 Replace/Evict，
 - * 如果出现 tag ecc error，需要向 l2 返回 corrupt=1 的 Release 请求。
 - * 如果出现 data ecc error，需要向 l2 返回 corrupt=1 的 ReleaseData 请求。
- Store to DCache
 - 对于 Sbuffer 写入数据至 DCache
 - * 如果出现 tag ecc error，则根据 Repalce/Evict 流程释放 cacheline，并将数据写入 dcache 中，不向 l2 报送错误。
 - * 如果出现 data ecc error，则直接写入数据，不向 l2 报送错误
- Atomics
 - 对于 Atomic，报告异常，但是不向 l2 报送错误
- 多错误选择
 - 如果同时出现多个错误，则优先级为 ldu0 > ldu1 > ldu2 > MainPipe

17.7.2 整体框图

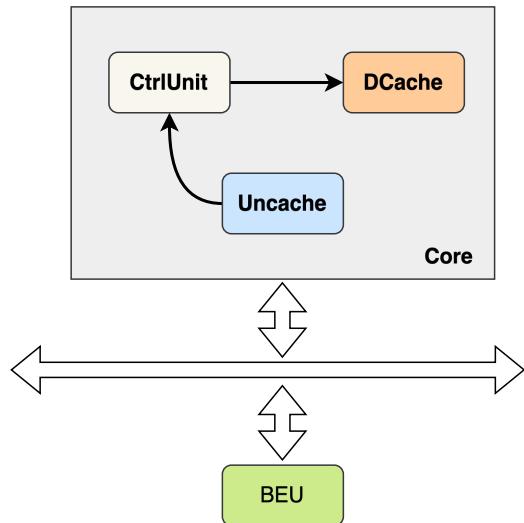


图 17.19: Error 架构

17.7.3 接口时序

17.7.3.1 配置寄存器时序

- 可以通过 tilelink 接口读写配置寄存器，如图17.20, A 通道传递写地址和数据。
 - 配置地址为 0x38022010 的 EccMask0 寄存器，写入的数据为 0xff;
 - 配置地址为 0x38022008 的 EccEid 寄存器，写入的数据为 0x4;
 - 配置地址为 0x38022000 的 EccCtl 寄存器，写入的数据为 0x5

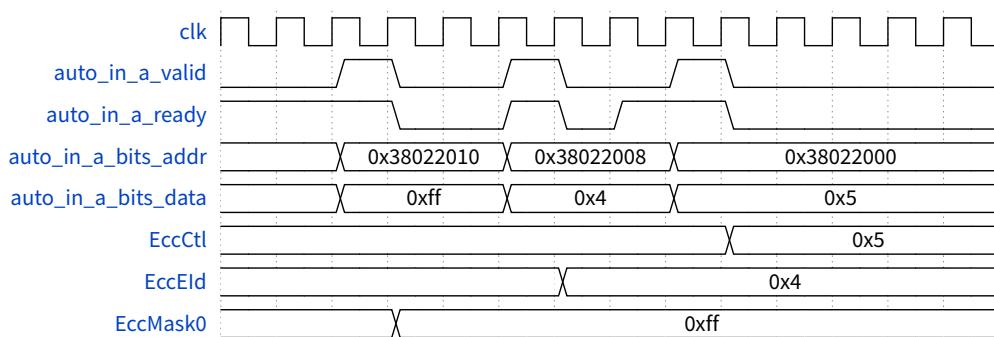


图 17.20: 配置寄存器时序

17.7.3.2 Tag 注入时序

- 如图17.21所示，当配置好寄存器 (EccCtl, EccEid 和 EccMask0) 之后，当计时器计时到 0，开始注入：
 - tag 注入接口 io_pseudoError_0_valid 拉高，

- 当注入成功后（即 `io_pseudoError_0_valid && io_pseudoError_0_ready == 1`），EccCtl 的 ese 位将清零，结束注入；
- 以 MainPipe 为例，`s1_tag_error`、`s2_tag_error` 和 `s3_tag_error` 逐级拉高，最后通过 `io_error` 端口向 BEU 报告错误信息

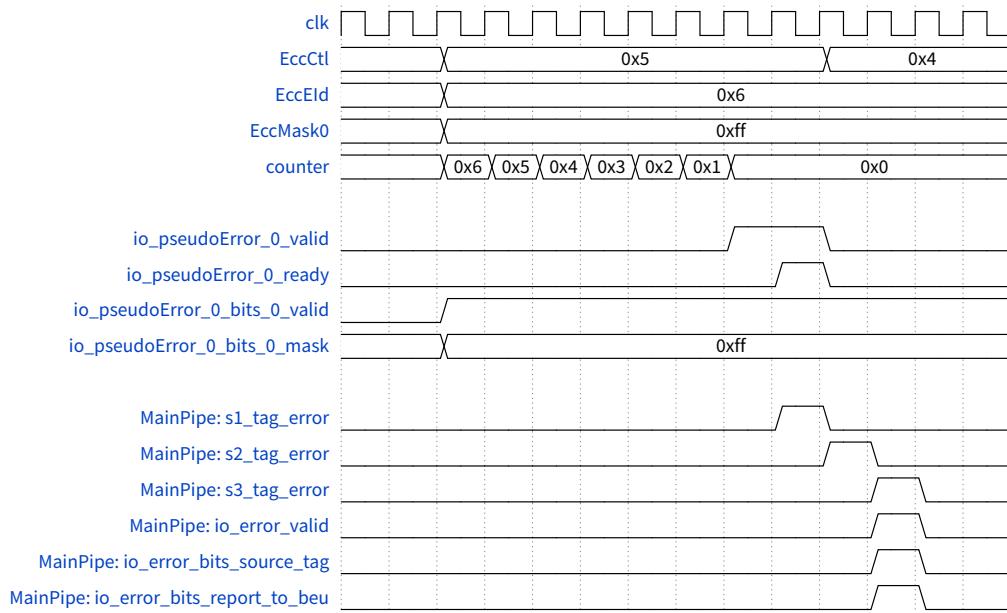


图 17.21: Tag 注入时序

17.7.3.3 Data 注入时序

- 如图17.22所示，当配置好寄存器（EccCtl, EccEid 和 EccMask2）之后，当计时器计时到 0，开始注入：
 - tag 注入接口 io_pseudoError_1_valid 拉高，
 - 当注入成功后（即 io_pseudoError_1_valid && io_pseudoError_1_ready == 1），EccCtl 的 ese 位将清零，结束注入；
 - 以 MainPipe 为例，s2_data_error 和 s3_data_error 逐级拉高，最后通过 io_error 端口向 BEU 报告错误信息

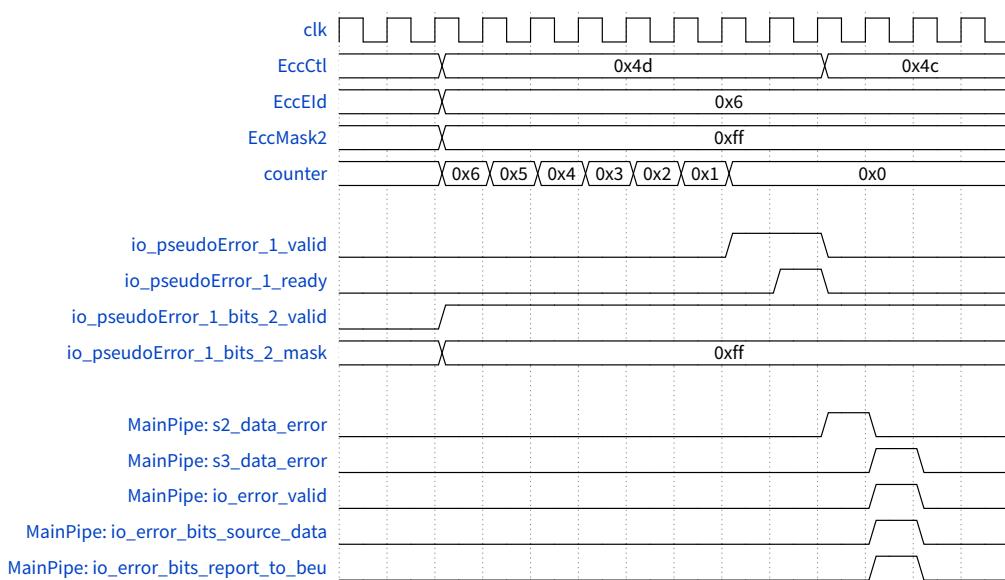


图 17.22: Data 注入时序

18 内存管理单元

18.1 内存管理单元概述

18.1.1 术语说明

表 18.1: 内存管理单元术语说明

| 缩写 | 全称 | 描述 |
|----------|---|----------------------|
| MMU | Memory Management Unit | 内存管理单元 |
| TLB | Translation Lookaside Buffer | 页表的缓存 |
| ITLB | Instruction TLB | 指令页表缓存 |
| DTLB | Data TLB | 数据页表缓存 |
| L1 TLB | Level 1 TLB | 一级 TLB |
| L2 TLB | Level 2 TLB | 二级 TLB |
| SV39 | Page-Based 39-bit Virtual-Memory System | RISC-V 手册中规定的一种分页机制 |
| PGD | Page Global Directory | 页全局目录 |
| PMD | Page Mid-level Directory | 页中间目录 |
| PTE | Page Table Entry | 页表项 |
| PTW | Page Table Walk | 页表查询过程 |
| PMP | Physical Memory Protection | 物理内存保护 |
| PMA | Physical Memory Attributes | 物理地址属性 |
| ASID | Address Space IDentifier | 地址空间标识符 |
| CSR | Control and Status Register | 控制和状态寄存器 |
| VPN | Virtual Page Number | 虚拟页号 |
| PPN | Physical Page Number | 物理页号 |
| PLRU | Pseudo-Least Recently Used | 一种近似最近最少使用的算法 |
| VMID | Virtual Machine Identifier | 虚拟机号 |
| GVPN | Guest Virtual Page Number | 第二阶段翻译的虚拟页号（客户机物理地址） |
| VS-Stage | Virtual Superior Stage | 第一阶段翻译 |
| G-Stage | Guest Stage | 第二阶段翻译 |

| 缩写 | 全称 | 描述 |
|--------|--|--------------------------|
| SV39x4 | A Variation on Page-Based 39-bit Virtual-Memory System | 地址扩大两位的 SV39，根页表此时为 16KB |
| HPTW | Hypervisor Page Table Walker | 负责第二阶段翻译的页表查询 |
| GPA | Guest Physical Address | 客户机物理地址 |

18.1.2 设计规格

MMU 模块的整体设计规格如下：

1. 支持将虚拟地址转换为物理地址
2. 支持 Sv39 分页机制
3. 支持访问内存中的页表
4. 支持动态、静态 PMP 检查
5. 支持动态、静态 PMA 检查
6. 支持 asid
7. 支持 Sfence.vma
8. 支持软件更新 A/D 位
9. 支持 H 拓展的两阶段地址翻译
10. 支持 Sv39x4 分页机制
11. 支持 vmid
12. 支持 hfence.vvma 和 hfence.gvma

18.1.3 功能描述

香山的 MMU 模块由 L1 TLB, Repeater, L2 TLB, PMP 和 PMA 模块组成，其中 L2TLB 模块又分为 Page Cache、Page Table Walker、Last Level Page Table Walker、Miss Queue 和 Prefetcher 五部分。在核内进行内存读写，包括前端取指和后端访存之前，都需要由 MMU 模块进行地址翻译。前端取指和后端访存分别通过 ITLB 和 DTLB 进行地址翻译，均为非阻塞式访问，TLB 需要返回请求是否 miss，返回给请求来源，并由请求来源调度重新发送 TLB 查询请求，直至命中。对于 miss 的 Load 请求，昆明湖架构支持 TLB Hint，即当 L2 TLB refill 页表至 L1 TLB 时，可以精准唤醒因该虚拟地址 TLB miss 而导致阻塞的 Load 指令。当 L1 TLB (ITLB 和 DTLB) 发生 miss 时，会访问 L2 TLB。如果 L2 TLB 依然 miss，则会通过 Page Table Walker 访问内存中的页表。

Repeater 是 L1 TLB 到 L2 TLB 的请求缓冲，在 L1 TLB 和 L2 TLB 之间存在较长的物理距离，需要通过 Repeater 在中间加拍。由于 ITLB 和 DTLB 均支持多个 outstanding 的请求，因此 repeater 会同时承担类似 MSHR 的功能，并过滤重复请求。MMU 模块支持对物理地址访问进行权限检查，分为 PMP 和 PMA 两部分。PMP 和 PMA 检查并行查询，违反任一权限即为非法操作。对核内所有的物理地址访问都需要进行物理地址权限检查，包括在 ITLB 和 DTLB 检查之后，以及 Page Table Walker 访存之前。

增加 H 拓展后，L2TLB 内新增了 Hypervisor Page Table Walker 模块，主要负责第二阶段的翻译，并且对 L2TLB 的架构进行了部分修改。

18.1.3.1 支持 Sv39 分页机制，将虚拟地址翻译成物理地址

为了实现进程隔离，每个进程都会有自己的地址空间，使用的地址都是虚拟地址。MMU 负责将虚拟地址翻译成物理地址，并用翻译得到的物理地址进行访存。香山处理器昆明湖架构支持 Sv39 分页机制（参见 RISC-V 特权级手册），虚拟地址长度为 39 位，低 12 位是页内偏移，高 27 位分为三段（每段 9 位），也就是三级页表。昆明湖架构的物理地址为 36 位，虚拟地址和物理地址的结构如图 18.1, 18.2 所示。遍历页表需要进行三次内存访问，因此我们需要通过 TLB 对页表做缓存。

| | | | | |
|--------|--------|--------|-------------|---|
| 38 | 30 29 | 21 20 | 12 11 | 0 |
| VPN[2] | VPN[1] | VPN[0] | page offset | |
| 9 | 9 | 9 | 12 | |

图 18.1: 香山处理器的虚拟地址结构

| | | | | |
|--------|--------|--------|-------------|---|
| 35 | 30 29 | 21 20 | 12 11 | 0 |
| PPN[2] | PPN[1] | PPN[0] | page offset | |
| 6 | 9 | 9 | 12 | |

图 18.2: 香山处理器的物理地址结构

在进行地址翻译时，前端取指通过 ITLB 进行地址翻译，后端访存通过 DTLB 进行地址翻译。ITLB 和 DTLB 如果 miss，会通过 Repeater 向 L2 TLB 发送请求。在目前设计中，前端取指和后端访存对 TLB 均采用非阻塞式访问，即一个请求 miss 后，会将请求 miss 的信息返回，请求来源调度重新发送 TLB 查询请求，直至命中。

同时，访存拥有 2 个 Load 流水线，2 个 Store 流水线，以及 SMS 预取器、L1 Load stream & stride 预取器。为应对众多请求，两条 Load 流水线及 L1 Load stream & stride 预取器使用 Load DTLB，两条 Store 流水线使用 Store DTLB，预取请求使用 Prefetch DTLB，共 3 个 DTLB。

为避免 TLB 中出现重复项，ITLB repeater 和 DTLB repeater 分别接受来自 ITLB 和 DTLB 的请求，需要过滤掉重复的请求后才能继续发往 L2 TLB。如果 L2 TLB 发生 miss，会使用 Hardware Page Table Walker 访问内存中的页表内容。得到页表内容后，返还给 Repeater，并最终返还给 ITLB 和 DTLB。（参见 § 18.1.5 总体设计）

18.1.3.2 支持用于虚拟化的两阶段地址翻译

H 拓展加入后，在非虚拟化模式且未执行虚拟化访存指令下，地址翻译过程与未加入 H 拓展基本一致；在虚拟化模式下或者执行虚拟化访存指令时，则根据 vsatp 和 hgatp 来判断是否开启两阶段的翻译，即为 VS-stage 和 G-stage，VS-stage 负责将客户机虚拟地址转换成客户机物理地址，G-stage 负责将客户机的物理地址转换成主机的物理地址。第一阶段的翻译和非虚拟化的翻译过程基本一致，第二阶段的翻译在 PTW 与 LLPTW 模块中进行，查询逻辑为：首先在 Page Cache 中查找，如果找到则返回给 PTW 或者 LLPTW，如果没有找到就进入 HPTW 进行翻译，由 HPTW 返回并填入 Page Cache。

在 G-stage 中，分页机制叫做 Sv39x4，即该模式下的虚拟地址为 41 位，根页表变成 16KB。

在两阶段地址翻译中，第一阶段翻译得到的地址（包括翻译过程中计算得到的页表地址）均为客户机的物理地址，需要进行第二阶段翻译得到真实的物理地址后才能进行访存读取页表。逻辑上的翻译过程如下图 18.4, 18.5 所示。

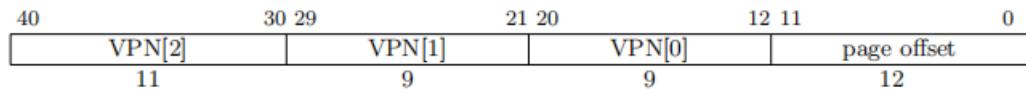


图 18.3: 香山处理器 Sv39x4 的虚拟地址结构 (客户机物理地址)

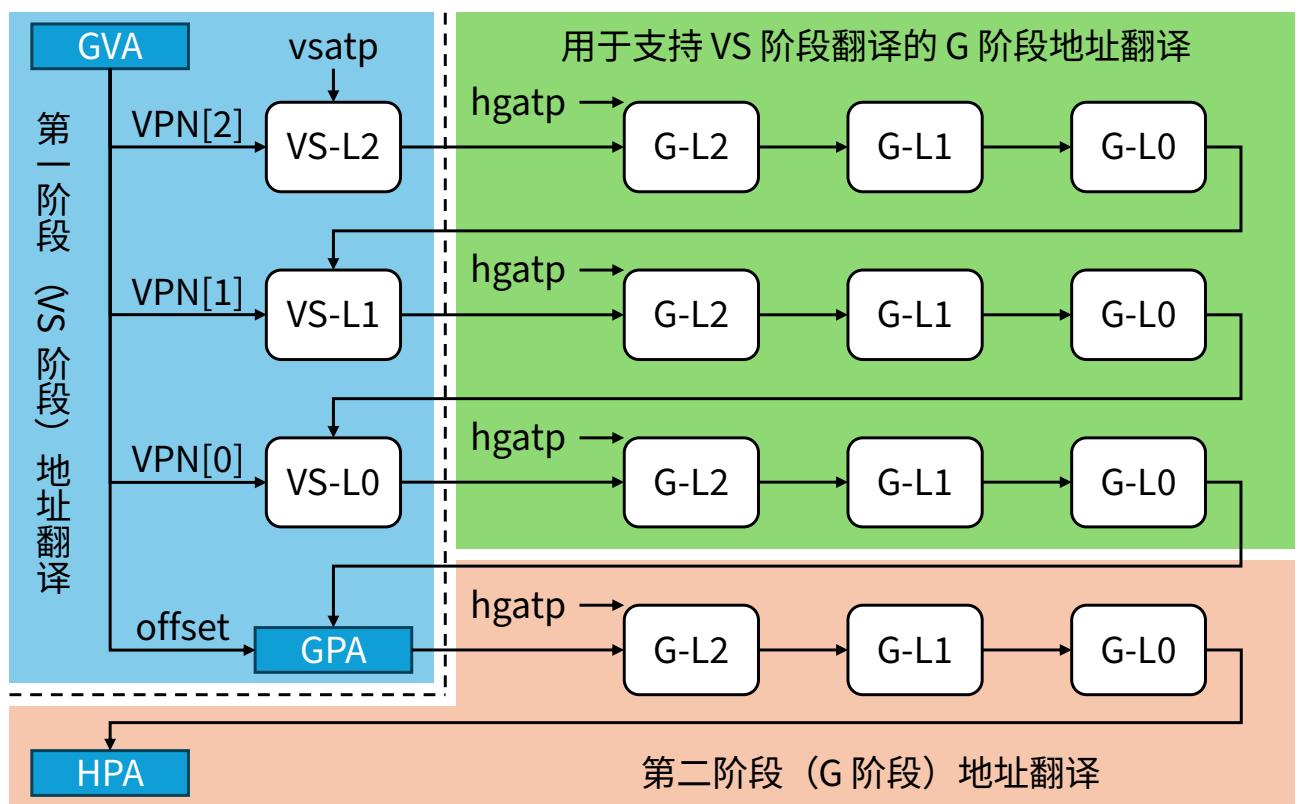


图 18.4: Sv39 - Sv39x4 两阶段地址翻译的过程

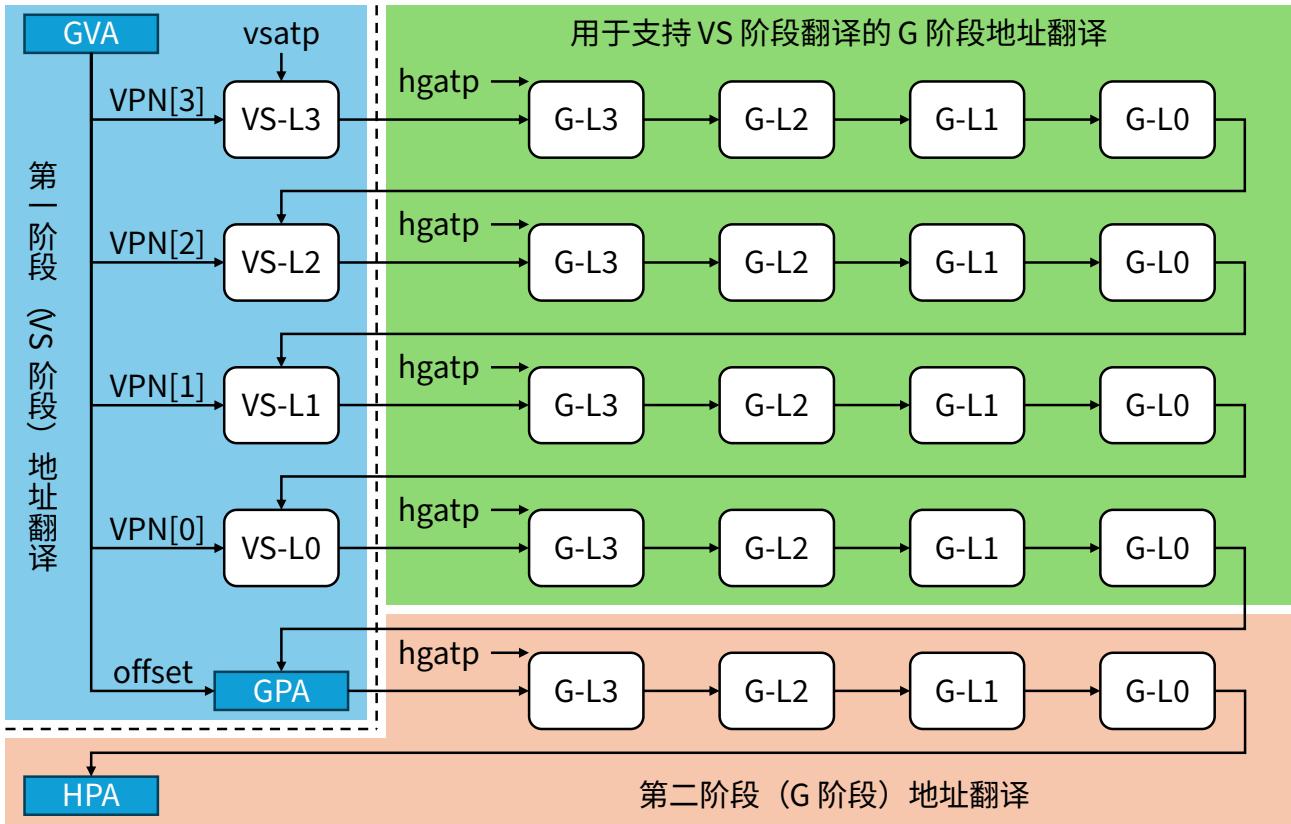


图 18.5: Sv48 - Sv48x4 两阶段地址翻译的过程

18.1.3.3 支持访问内存中的页表内容

在 L1 TLB 向 L2 TLB 发送请求时，将首先访问 Page Cache。对于非两阶段翻译的请求，若命中叶子节点则直接返回给 L1 TLB，否则根据 Page Cache 命中的页表等级以及 Page Table Walker 和 Last Level Page Table Walker 的空闲情况进入 Page Table Walker、Last Level Page Table Walker 或 Miss Queue（参见 5.3 节）。对于两阶段地址翻译的请求，由于 Page Cache 每次只能处理一个查询请求，所以两阶段地址翻译请求在 Page Cache 中首先会查询第一阶段的页表，第一阶段命中则会把请求发给 Page Table Walker，在 Page Table Walker 中进行第二阶段的翻译；如果第一阶段没有命中，则会根据命中的页表级别，发送给 Page Table Walker 或者 Last Level Page Table Walker，在这两个模块中会进行第二阶段的翻译。Page Table Walker 和 Last Level Page Table Walker 发送的第二阶段地址翻译的请求会先发送到 Page Cache 里面进行查询，如果命中，则 Page Cache 直接返回结果给对应的模块，如果没有命中，则发送给 Hypervisor Page Table Walker 进行翻译，翻译结果会直接返回给 Page Table Walker 或者 Last Level Page Table Walker。Page Table Walker 同时只能处理一个请求，进行 Hardware Page Table Walk，Page Table Walker 可以访问内存中前两级页表的内容，不访问 4KB 页表。如果 Page Table Walker 访问到 2MB 或 1GB 的叶子节点，或访问发生 Page fault 或 Access fault，则返回给 L1 TLB，否则送往 Last Level Page Table Walker 访问内存中的最后一级（4KB）页表。Hypervisor Page Table Walker 每次只能处理一个请求，所以在 Last Level Page Table Walker 中第二阶段翻译的请求是串行发送出去的。Hypervisor Page Table Walker 访问可能触发 Page fault 或者 Access fault，会返回给 PTW 或者 LLPTW，PTW 或者 LLPTW 会返回给 L1TLB。

Page Table Walker 和 Last Level Page Table Walker 以及新加的 Hypervisor Page Table Walker 均可以向内存发送请求，访问内存中的页表内容。在通过物理地址访问内存中的页表内容之前，需要通过 PMP 和 PMA 模块对物理地址进行检查（参见 3.2.3 和 5.4 节），如果发生 access fault 则不会向内存发送请求。来自 Page Table Walker、Last Level Page Table Walker、Hypervisor Page Table Walker 的请求在经过仲裁后，

通过 TileLink 总线向 L2 Cache 发送请求。L2 Cache 的访存宽度为 512 bits，因此每次会返回 8 项页表。

昆明湖的 MMU 实现了页表压缩机制，会对连续的页表项进行压缩。具体地，对于虚拟页号高位相同的页表项，当这些页表项的物理页号高位和页表属性也相同时，可以将这些页表项压缩为一项保存，从而提升 TLB 的有效容量。因此，L2 TLB 命中 4KB 页时，会返回至多 8 项连续的页表项（参见 5.2 节 L2 TLB 中的相应描述）。在 H 拓展中，L1TLB 中与虚拟化拓展有关的页表压缩机制被无效掉，视为一个页表；L2TLB 中的与虚拟化拓展有关的页表仍然采用页表压缩机制。

18.1.3.4 支持对物理地址访问进行权限检查

香山支持 PMP 和 PMA 检查，PMP 和 PMA 检查并行查询，如果违反其中一个权限，即为非法操作。PMP 和 PMA 的具体实现分为 CSR Unit、Frontend、Memblock、L2 TLB 四部分。昆明湖架构中 PMP 和 PMA 均为 16 项，关于 PMP 和 PMA 寄存器的地址空间以及配置寄存器的说明，参见 5.4 节。

CSR Unit 负责响应 CSRRW 等 CSR 指令对这些 PMP 和 PMA 寄存器的读写；在 Frontend、Memblock 和 L2 TLB 中包含这些 PMP 和 PMA 寄存器的备份，负责地址检查，通过拉取 CSR 的写信号可以保证这些寄存器的内容一致性。由于 L1 TLB 的面积较小，因此 PMP 和 PMA 寄存器的备份存储在 Frontend 或 Memblock 中，分别为 ITLB 和 DTLB 提供检查。L2 TLB 的面积较大，因此 PMP 和 PMA 寄存器的备份直接存储在 L2 TLB 中。

在 ITLB 和 DTLB 查询得到结果之后，以及 L2 TLB 中使用物理地址访存之前都需要进行 PMP 和 PMA 检查。按照手册规定，PMP 和 PMA 的检查应该为动态检查，即需要经过 TLB 翻译之后，使用翻译后的物理地址进行物理地址权限检查。出于时序考虑，DTLB 的 PMP & PMA 检查结果可以提前查询好，在回填时存入 TLB 项中，此为静态检查。具体地，当 L2 TLB 的页表项回填入 DTLB 时，同时将回填的页表项送给 PMP 和 PMA 进行权限检查，将检查得到的属性位（包括 R、W、X、C、Atomic，这些属性位的具体含义参见 5.4 节）同时存储在 DTLB 中，这样可以直接将这些检查结果返回给 MemBlock，无需再次检查。为实现静态检查，需要提升 PMP 和 PMA 的粒度为 4KB。

需要注意的是，目前 PMP & PMA 检查暂时并非昆明湖的时序瓶颈，因此未采用静态检查，全部使用动态检查的方式，即 TLB 查询得到物理地址后，再进行检查。昆明湖 V1 的代码中不包括静态检查，只包括动态检查，请再次注意。但出于兼容性，PMP 和 PMA 的粒度依然保持为 4KB。

18.1.3.5 支持内存管理栅栏指令指令

昆明湖 V2R2 支持 SFENCE.VMA、HFENCE.VVMA、HFENCE.GVMA 等内存管理栅栏指令。

当 Sfence.vma 指令执行时，会先将 Store Buffer 的全部内容写回到 DCache 中，之后发出刷新信号到 MMU 的各个部分。刷新信号是单向的，只会持续一拍，没有返回信号。Sfence.vma 指令最后会刷新整个流水线，从取指开始重新执行。Sfence.vma 指令会取消所有 inflight 的请求，包括 Repeater 和 Filter，以及 L1TLB 和 L2 TLB 中的 inflight 请求，并且根据地址和 ASID 刷新 L1 TLB 和 L2 TLB 中缓存的页表。Sfence.vma 指令的参数如图 18.6 所示。

| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|------------|-------|-------|--------|-------|--------|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 7 | 5 | 5 | 3 | 5 | 7 | |
| SFENCE.VMA | asid | vaddr | PRIV | 0 | SYSTEM | |

图 18.6: Sfence.vma 的指令格式

另外，香山昆明湖架构支持 Svinval 扩展，Svinval.vma 指令的格式如图 18.7 所示，其中 rs1 和 rs2 的含

义和 Sfence.vma 指令相同。昆明湖架构中 TLB 内部实现 Svinval.vma 指令和 Sfence.vma 指令的逻辑完全一致，TLB 只接受传入的 sfence_valid 信号以及相应的 rs1、rs2 参数。

| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|------------|-------|-------|--------|-------|--------|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 7 | 5 | 5 | 3 | 5 | 7 | |
| SINVAL.VMA | asid | vaddr | PRIV | 0 | SYSTEM | |

图 18.7: Svinval.vma 的指令格式

Hfence 指令包括 Hfence.vvma 和 Hfence.gvma，该类型指令的执行效果与 Sfence.vma 类似，先将 Store Buffer 的全部内容写回到 DCache 中，之后发出刷新信号到 MMU 的各个部分。刷新信号是单向的，只会持续一拍，没有返回信号。指令最后会刷新整个流水线，从取指开始重新执行。指令会取消所有 inflight 的请求，包括 Repeater 和 Filter，以及 L1TLB 和 L2 TLB 中的 inflight 请求，Hfence.vvma 会根据地址和 ASID 与 VMID 来刷新 L1TLB 和 L2TLB 中与 VSATP 有关的页表，Hfence.gvma 会根据地址与 VMID 来刷新 L1TLB 和 L2TLB 中与 HGATP 有关的页表。

| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|-------------|-------|-------|--------|-------|--------|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 7 | 5 | 5 | 3 | 5 | 7 | |
| HFENCE.VVMA | asid | vaddr | PRIV | 0 | SYSTEM | |
| HFENCE.GVMA | vmid | gaddr | PRIV | 0 | SYSTEM | |

图 18.8: Hfence 的指令格式

另外，由于昆明湖架构支持 Svinval 扩展，对应有 hinval.vvma 和 hinval.gvma 指令，这两条指令分别与 hfence 的两条指令对应。

| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|-------------|-------|-------|--------|-------|--------|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 7 | 5 | 5 | 3 | 5 | 7 | |
| HINVAL.VVMA | asid | vaddr | PRIV | 0 | SYSTEM | |

| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|-------------|-------|-------|--------|-------|--------|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 7 | 5 | 5 | 3 | 5 | 7 | |
| HINVAL.GVMA | vmid | gaddr | PRIV | 0 | SYSTEM | |

图 18.9: Hinval 的指令格式

18.1.3.6 支持 ASID 和 VMID

香山昆明湖架构支持长度为 16 的 ASID（地址空间标识符），在 SATP 寄存器中保存。SATP 寄存器的格式如表 18.2 所示。

表 18.2: SATP 寄存器的格式

| 位 | 域 | 描述 |
|---------|------|--|
| [63:60] | MODE | 表示地址转换的模式。该域为 0 时为 Bare mode, 不开启地址翻译或地址保护, 该域为 8 时表示 Sv39 地址转换模式, 如果该域为其他值会上报 illegal instruction fault |
| [59:44] | ASID | 地址空间标识符。ASID 的长度可参数化配置, 对于香山昆明湖架构采用的 Sv39 地址转换模式, ASID 长度最大值为 16 |
| [43:0] | PPN | 表示根页表的物理页号, 由物理地址右移 12 位得到。 |

注意, 在虚拟化模式下, SATP 将被 VSATP 寄存器代替, 并且其中的 PPN 为客户机根页表的客户机物理页号, 而非真实的物理地址, 需要进行第二阶段翻译才能得到真实物理地址。

香山昆明湖架构支持长度为 14 的 VMID (虚拟机标识符), 在 HGATP 寄存器中保存, HGATP 寄存器格式如表 18.3 所示。

表 18.3: HGATP 寄存器的格式

| 位 | 域 | 描述 |
|---------|------|--|
| [63:60] | MODE | 表示地址转换的模式。该域为 0 时为 Bare mode, 不开启地址翻译或地址保护, 该域为 8 时表示 Sv39x4 地址转换模式, 如果该域为其他值会上报 illegal instruction fault |
| [57:44] | VMID | 虚拟机标识符。对于香山昆明湖架构采用的 Sv39x4 地址转换模式, VMID 长度最大值为 14 |
| [43:0] | PPN | 表示第二阶段翻译的根页表的物理页号, 由物理地址右移 12 位得到。 |

18.1.3.7 支持软件更新 A/D 位

香山支持软件对页表中 A/D 位的管理。A 位表示自上次清空 A 位以来, 对该页进行过读, 写或取指操作。D 位表示自上次清空 D 位以来, 对该页进行过写操作。手册中允许通过软件和硬件两种方式更新 A/D 位, 香山选择软件方式, 即当发现如下两种情况时通过报 page fault, 通过软件更新页表。

- 访问某页, 但该页页表的 A 位是 0
- 写入某页, 但该页页表的 D 位是 0

需要注意, 目前香山昆明湖架构并不支持硬件更新 A/D 位。

18.1.3.8 支持异常处理机制

当 PMP、PMA 检查报 access fault, 或出现 page fault、guest page fault 等情况时, TLB 模块会根据 PTW 请求的来源, ITLB 向 Frontend 返回异常, DTLB 向 Memblock 返回异常。TLB 模块可能向 Frontend 和 Memblock 返回的异常种类如表 3.3 所示, 其中 Memblock 又可以细化分为 LoadUnit, AtomicsUnit 和 StoreUnit。TLB 模块只负责向 Frontend 或 Memblock 返回 access fault 或 page fault 或 guest page fault, 后续由 Frontend 或 Memblock 进行处理。关于异常处理的总结和说明, 参见 § 18.1.4 异常处理机制。

表 18.4: TLB 返回的异常种类

| 种类 | 目的 | 描述 |
|-----------|-------------------------|-----------------------------|
| pf_instr | Frontend | 表示发生 inst page fault |
| af_instr | Frontend | 表示发生 inst access fault |
| gpf_instr | Frontend | 表示发生 inst guest page fault |
| pf_ld | LoadUnit 或 AtomicsUnit | 表示发生 load page fault |
| af_ld | LoadUnit 或 AtomicsUnit | 表示发生 load access fault |
| gpf_ld | LoadUnit 或 AtomicsUnit | 表示发生 load guest page fault |
| pf_st | StoreUnit 或 AtomicsUnit | 表示发生 store page fault |
| af_st | StoreUnit 或 AtomicsUnit | 表示发生 store access fault |
| gpf_st | StoreUnit 或 AtomicsUnit | 表示发生 store guest page fault |

18.1.4 异常处理机制

MMU 模块可能产生的异常包括: guest page fault、page fault、access fault、以及 L2 TLB Page Cache 的 ECC 校验出错。ITLB、DTLB 与 L2 TLB 均可能产生 guest page fault、page fault 和 access fault。对于 ITLB 和 DTLB 产生的异常，会根据请求来源分别交付给发送物理地址查询的模块进行处理，ITLB 会交付给 Icache 或 IFU；DTLB 会交付给 LoadUnits、StoreUnits 或 AtomicsUnit 进行处理。

如果 L2 TLB 产生 guest page fault、page fault 或 access fault，L2 TLB 并不会直接将产生的异常进行处理，而是会将该信息返回给 L1 TLB。L1 TLB 在查询发现出现 guest page fault、page fault 或 access fault 后，会根据请求的 cmd，产生不同种类的异常，并根据请求来源交付给各模块处理。

L2 TLB 中 Page Cache 支持 ecc 校验，如果 ecc 检查报错，并不会报例外，而是会向 L2 TLB 发送该请求 miss 信号。同时 Page Cache 将 ecc 报错的项刷新，重新发送 PTW 请求，进行 Page Walk。

也就是说，MMU 模块只会处理 L2 TLB 中 Page Cache 的 ECC 校验出错异常，对于产生的 page fault 和 access fault，均交付给前端或后端流水线进行处理。

可能产生的异常以及 MMU 模块的处理流程如表 18.5 所示：

表 18.5: MMU 可能产生的异常以及处理流程

| 模块 | 可能产生的异常 | 处理流程 |
|------|--------------------------|--|
| ITLB | 产生 inst page fault | 根据请求来源，分别交付给 Icache 或 IFU 处理 |
| | 产生 inst guest page fault | 根据请求来源，分别交付给 Icache 或 IFU 处理 |
| | 产生 inst access fault | 根据请求来源，分别交付给 Icache 或 IFU 处理 |
| DTLB | 产生 load page fault | 交付给 LoadUnits 进行处理 |
| | 产生 load guest page fault | 交付给 LoadUnits 进行处理 |
| | 产生 store page fault | 根据请求来源，分别交付给 StoreUnits 或 AtomicsUnit 处理 |

| 模块 | 可能产生的异常 | 处理流程 |
|--------|---------------------------|---|
| L2 TLB | 产生 store guest page fault | 根据请求来源, 分别交付给 StoreUnits 或 AtomicsUnit 处理 |
| | 产生 load access fault | 交付给 LoadUnits 进行处理 |
| | 产生 store access fault | 根据请求来源, 分别交付给 StoreUnits 或 AtomicsUnit 处理 |
| | 产生 guest page fault | 交付给 L1 TLB, L1 TLB 根据请求来源 交付处理 |
| L2 TLB | 产生 page fault | 交付给 L1 TLB, L1 TLB 根据请求来源 交付处理 |
| | 产生 access fault | 交付给 L1 TLB, L1 TLB 根据请求来源 交付处理 |
| | ecc 校验出错 | 无效掉当前项, 返回 miss 结果并重新 进行 Page Walk |

18.1.5 总体设计

MMU 整体架构如图 18.10 所示。

ITLB 接收来自 Frontend 的 PTW 请求, DTLB 接收来自 Memblock 的 PTW 请求。来自 Frontend 的 PTW 请求包括 ICache 的 3 个请求和 IFU 的 1 个请求, 来自 Memblock 的 PTW 请求包括 LoadUnit 的 2 个请求 (AtomsicsUnit 占用 LoadUnit 的 1 个请求通道), L1 Load stream & stride 预取器的 1 个请求, StoreUnit 的 2 个请求, 以及 SMSPrefetcher 的 1 个请求。ITLB、DTLB 通过 Repeater 与 L2 TLB 连接, 均为非阻塞式访问。这些 Repeater 在加拍功能的基础上, 增加了过滤重复请求的功能, 可以过滤掉 L1 TLB 向 L2 TLB 发送的重复请求, 避免 L1 TLB 中出现重复项。

ITLB 的请求和 DTLB 的请求经过仲裁(Arbiter 2to1), 将首先访问 Page Cache, 若是非两阶段地址翻译的请求, 命中叶子节点则直接返回给 L1 TLB, 没有命中则根据 Page Cache 命中的页表等级以及 Page Table Walker 和 Last Level Page Table Walker 的空闲情况进入 Page Table Walker、Last Level Page Table Walker 或 Miss Queue (参见 5.3 节)。对于来自 Miss Queue、Prefetcher 的请求, 会通过仲裁器 (Arbiter 3to1) 与来自 L1 TLB 的请求一起做仲裁, 并重新访问 Page Cache。另一种情况, Page Cache 收到的是两阶段地址翻译请求, 对于两阶段翻译均启用的情况, 若第一阶段页表命中, 则发送给 PTW 进行第二阶段翻译, 其他情况根据第一阶段页表命中的级别和 PTW 以及 LLPTW 的空闲情况发送给 PTW、LLPTW、Miss Queue; 对于只有第一阶段翻译的情况, 则类似非两阶段地址翻译请求的处理, 根据命中等级和 PTW 以及 LLPTW 的空闲情况发送给 PTW、LLPTW、Miss Queue; 对于只有第二阶段翻译的情况, 如果查询到则返回给 L1TLB, 否则发送给 PTW 进行第二阶段翻译。此外, Page Cache 还会收到 isHptwReq 有效的请求, 代表该请求是一个进行第二阶段翻译的请求, 该类型请求若在 Page Cache 中命中, 则会发送给 hptw_resp_arb, 若没有命中, 则发送给 HPTW 进行查询, HPTW 会将查询结果发送给 hptw_resp_arb。

Page Table Walker 和 Last Level Page Table Walker 均可以进行第二阶段的地址翻译, 在 PTW 和 LLPTW 中, 如果是一个两阶段地址翻译请求, 在 PTW 或者 LLPTW 每次从 PTE 中得到的地址均为客户机物理地址, 在访存前均要进行一次第二阶段地址翻译, 获得真实的物理地址, 参见 PTW 和 LLPTW 模块介绍。

Page Table Walker 和 Last Level Page Table Walker 均可以向内存发送请求, 访问内存中的页表内容。在通过物理地址访问内存中的页表内容之前, 需要通过 PMP 和 PMA 模块对物理地址进行检查, 如果出现 access

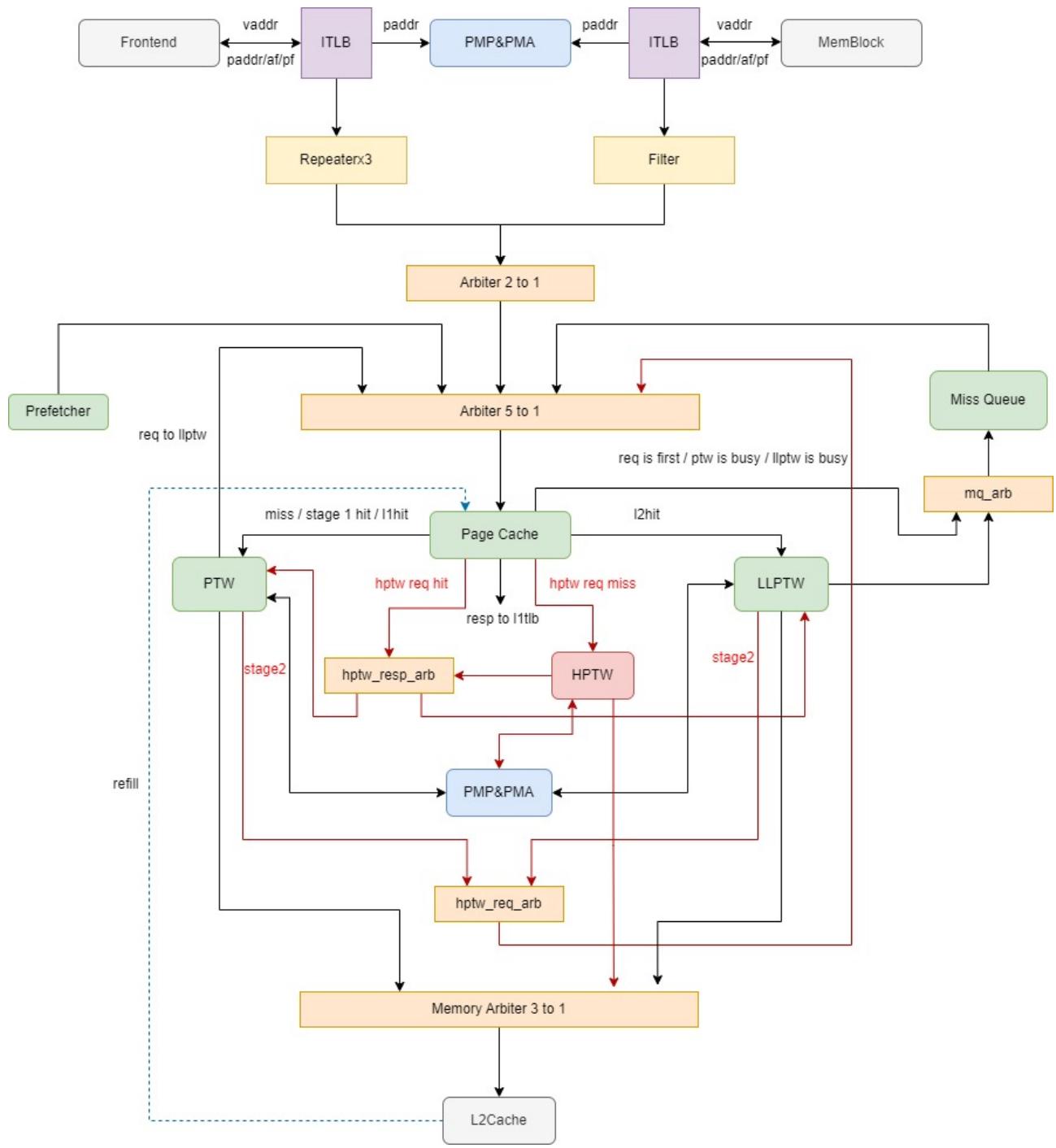


图 18.10: MMU 模块整体框图

fault 则不会向内存发送请求。来自 Page Table Walker 和 Last Level Page Table Walker 的请求在经过仲裁后 (Memory Arbiter 2to1)，通过 TileLink 总线向 L2 Cache 发送请求。L2 TLB 除了需要向 L2 Cache 发送物理地址外，还需要通过 id 表示请求的来源。L2 Cache 的访存宽度为 512 bits，因此每次会返回 8 项页表。每次访存返回的页表都会重填进 Page Cache。

在 ITLB 和 DTLB 查询得到结果之后，以及 L2 TLB 进行 Page Table Walker 之前都需要进行 PMP 和 PMA 检查。L1 TLB 的面积较小，因此 PMP 和 PMA 寄存器的备份并不存储在 L1 TLB 内部，而是存储在 Frontend 或 Memblock 中，分别为 ITLB 和 DTLB 提供检查。L2 TLB 的面积较大，PMP 和 PMA 寄存器的备份直接存储在 L2 TLB 中。

18.1.6 接口列表

MMU 模块的各部分与上层模块接口列表如表 18.6 所示。

表 18.6: MMU IO 接口列表

| 上层模块 | 模块名 | 实例化名 | 说明 |
|----------|---------------|----------------------------|--|
| Frontend | | | |
| | TLB | itlb | ITLB, 在 5.1 节介绍 |
| | PMP | pmp | 分布式 PMP 寄存器, 在 5.4 节介绍 |
| | PMPChecker | PMPChecker | PMP 检查器, 在 5.4 节介绍 |
| | PMPChecker | PMPChecker_1 | PMP 检查器, 在 5.4 节介绍 |
| | PMPChecker | PMPChecker_2 | PMP 检查器, 在 5.4 节介绍 |
| | PMPChecker | PMPChecker_3 | PMP 检查器, 在 5.4 节介绍 |
| | PTWFilter | itlbRepeater1 | ITLB 和 L2 TLB 连接的 Repeater1, 在 5.2 节介绍 |
| | PTWRepeaterNB | itlbRepeater2 | ITLB 和 L2 TLB 连接的 Repeater2, 在 5.2 节介绍 |
| MemBlock | | | |
| | TLBNonBlock | dtlb_ld_tlb_ld | Load DTLB, 在 5.1 节介绍 |
| | TLBNonBlock_1 | dtlb_ld_tlb_st | Store DTLB, 在 5.1 节介绍 |
| | TLBNonBlock_2 | dtlb_prefetch_tlb_prefetch | Prefetch DTLB, 在 5.1 节介绍 |
| | PTWNewFilter | dtlbRepeater | DTLB 和 L2 TLB 连接的 Repeater1, 在 5.2 节介绍 |
| | PTWRepeaterNB | itlbRepeater3 | ITLB 和 L2 TLB 连接的 Repeater3, 在 5.2 节介绍 |
| | PMP_2 | pmp | 分布式 PMP 寄存器, 在 5.4 节介绍 |
| | PMPChecker_8 | PMPChecker | PMP 检查器, 在 5.4 节介绍 |
| | PMPChecker_8 | PMPChecker_1 | PMP 检查器, 在 5.4 节介绍 |
| | PMPChecker_8 | PMPChecker_2 | PMP 检查器, 在 5.4 节介绍 |
| | PMPChecker_8 | PMPChecker_3 | PMP 检查器, 在 5.4 节介绍 |
| | PMPChecker_8 | PMPChecker_4 | PMP 检查器, 在 5.4 节介绍 |

| 上层模块 | 模块名 | 实例化名 | 说明 |
|--------------|--------------|------------------|---|
| | PMPChecker_8 | PMPChecker_5 | PMP 检查器, 在 5.4 节介绍 |
| L2TLBWrapper | | ptw | L2 TLB, 在 5.3 节介绍 |
| | TLBuffer_20 | ptw_to_l2_buffer | L2 TLB 和 L2 Cache 之间的 buffer, 在 5.3 节介绍 |

L2 TLB 模块的各部分与 L2 TLB 的接口列表

表 18.7: L2 TLB IO 接口列表

| 上层模块 | 模块名 | 实例化名 | 说明 |
|--------------|----------------|--------------|---|
| L2TLBWrapper | L2TLB | ptw | L2 TLB, 在 5.3 节介绍 |
| L2TLB | PMP | pmp | 分布式 PMP 寄存器, 在 5.4 节介绍 |
| | PMPChecker | PMPChecker | PMP 检查器, 在 5.4 节介绍 |
| | PMPChecker | PMPChecker_1 | PMP 检查器, 在 5.4 节介绍 |
| | L2TlbMissQueue | missQueue | L2 TLB Miss Queue, 在 5.3.11 节介绍 |
| | PtwCache | cache | L2 TLB Page Table Cache, 在 5.3.7 节介绍 |
| | PTW | ptw | L2 TLB Page Table Walker, 在 5.3.8 节介绍 |
| | LLPTW | llptw | L2 TLB Last Level Page Table Walker, 在 5.3.9 节介绍 |
| | HPTW | hptw | L2 TLB Hypervisor Page Table Walker, 在 5.3.10 节介绍 |
| | L2TlbPrefetch | prefetch | L2 TLB Prefetcher, 在 5.3.12 节介绍 |

详见接口列表文档。另外涉及到某些仲裁器, 在接口列表中省略。

18.1.7 接口时序

MMU 整体和外界的接口涉及 L1 TLB 与 Frontend、Memblock 的接口, 以及 L2 TLB 与内存 (L2 Cache) 之间的接口。

L1 TLB 与 Frontend、Memblock 之间的接口时序参见 § 18.2.4.1 ITLB 与 Frontend 的接口时序、§ 18.2.4.2 DTLB 与 Memblock 的接口时序。

L2 TLB 与 L2 Cache 的接口时序遵循 TileLink 总线协议。

18.2 二级模块 L1 TLB

18.2.1 设计规格

1. 支持接收 Frontend 和 MemBlock 的地址翻译请求
2. 支持 PLRU 替换算法
3. 支持向 Frontend 和 MemBlock 返回物理地址
4. ITLB 和 DTLB 均采用非阻塞式访问
5. ITLB 和 DTLB 项均采用寄存器堆实现
6. ITLB 和 DTLB 项均为全相联结构
7. ITLB 和 DTLB 分别采用处理器当前特权级和访存执行有效特权级
8. 支持在 L1 TLB 内部判断虚存是否开启以及两个阶段翻译是否开启
9. 支持向 L2 TLB 发送 PTW 请求
10. DTLB 支持复制查询返回的物理地址
11. 支持异常处理
12. 支持 TLB 压缩
13. 支持 TLB Hint 机制
14. 存储四种类型的 TLB 项
15. TLB refill 将两个阶段的页表进行融合
16. TLB 项的 hit 的判断逻辑
17. 支持客户机缺页后重新发送 PTW 获取 gpaddr

18.2.2 功能

18.2.2.1 接收 Frontend 和 MemBlock 的地址翻译请求

在核内进行内存读写，包括前端取指和后端访存前，都需要由 L1 TLB 进行地址翻译。因物理距离较远，并且为了避免相互污染，分为前端取指的 ITLB（Instruction TLB）和后端访存的 DTLB（Data TLB）。ITLB 采用全相联模式，48 项全相联保存全部大小页。ITLB 接收来自 Frontend 的地址翻译请求，itlb_requestors(0) 至 itlb_requestors(2) 来自 icache，其中 itlb_requestors(2) 为 icache 的预取请求；itlb_requestors(3) 来自 ifu，为 MMIO 指令的地址翻译请求。

ITLB 的项配置和请求来源分别如表 18.8, 18.9。

表 18.8: ITLB 的项配置

| 项名 | 项数 | 组织结构 | 替换算法 | 存储内容 |
|------|----|------|------|-------|
| Page | 48 | 全相联 | PLRU | 全部大小页 |

表 18.9: ITLB 的请求来源

| 序号 | 来源 |
|---------------|------------------|
| requestors(0) | Icache, mainPipe |
| requestors(1) | Icache, mainPipe |

| 序号 | 来源 |
|---------------|----------------------|
| requestors(2) | Icache, fdipPrefetch |
| requestors(3) | IFU |

香山的访存通道访存拥有 2 个 Load 流水线, 2 个 Store 流水线, 以及 SMS 预取器、L1 Load stream & stride 预取器。为应对众多请求, 两条 Load 流水线及 L1 Load stream & stride 预取器使用 Load DTLB, 两条 Store 流水线使用 Store DTLB, 预取请求使用 Prefetch DTLB, 共 3 个 DTLB, 均采用 PLRU 替换算法 (参见 5.1.1.2 节)。

DTLB 采用全相联模式, 48 项全相联保存全部大小页。DTLB 接收来自 MemBlock 的地址翻译请求, dtlb_ld 接收来自 loadUnits 和 L1 Load stream & stride 预取器的请求, 负责 Load 指令的地址翻译; dtlb_st 接收 Store-Units 的请求, 负责 Store 指令的地址翻译。特别地, 对于 AMO 指令, 会使用 loadUnit(0) 的 dtlb_ld_requestor, 向 dtlb_ld 发送请求。SMSPrefetcher 会向单独的 DTLB 发送预取请求。

DTLB 的项配置和请求来源分别如表 18.10, 18.11。

表 18.10: DTLB 的项配置

| 项名 | 项数 | 组织结构 | 替换算法 | 存储内容 |
|------|----|------|------|-------|
| Page | 48 | 全相联 | PLRU | 全部大小页 |

表 18.11: DTLB 的请求来源

| 模块 | 序号 | 来源 |
|---------|------------------|----------------------------------|
| DTLB_LD | | |
| | ld_requestors(0) | loadUnit(0), AtomicsUnit |
| | ld_requestors(1) | loadUnit(1) |
| | ld_requestors(2) | loadUnit(2) |
| | ld_requestors(3) | L1 Load stream & stride Prefetch |
| DTLB_ST | | |
| | st_requestors(0) | StoreUnit(0) |
| | st_requestors(1) | StoreUnit(1) |
| DTLB_PF | | |
| | pf_requestors(0) | SMSPrefetch |
| | pf_requestors(1) | L2 Prefetch |

18.2.2.2 采用 PLRU 替换算法

L1 TLB 采用可配置的替换策略, 默认为 PLRU 替换算法。在南湖架构中, ITLB 和 DTLB 均包括 NormalPage 和 SuperPage, 回填策略较复杂。南湖架构 ITLB 的 NormalPage 负责 4KB 大小页的地址转换, SuperPage 负责 2MB 和 1GB 大小页的地址转换, 需要根据回填的页大小 (4KB, 2MB 或 1GB) 填入 NormalPage 或 SuperPage。南湖架构 DTLB 的 NormalPage 负责 4KB 大小页的地址转换, SuperPage 负责全部页大小的地址转换。NormalPage 为直接映射, 虽然项数较多, 但利用度较低。SuperPage 为全相联, 利用率较高, 但由于时序限制项数较少, 缺失率很高。

请注意，昆明湖架构对上述问题提出优化，在满足时序的条件下，将 ITLB 和 DTLB 统一设置为 48 项全相联结构，任意大小的页都可以回填，ITLB 和 DTLB 均采用 PLRU 替换策略。

ITLB 和 DTLB 的回填策略如表 18.12 所示。

表 18.12: ITLB 和 DTLB 的回填策略

| 模块 | 项名 | 策略 |
|------|------|--------------------|
| ITLB | Page | 48 项全相联，可以回填任意大小的页 |
| DTLB | Page | 48 项全相联，可以回填任意大小的页 |

18.2.2.3 向 Frontend 和 MemBlock 返回物理地址

在 L1 TLB 通过虚拟地址得到物理地址后，会向 Frontend 和 MemBlock 返回相应请求的物理地址，以及请求是否发生 miss，是否发生 guest page fault, page fault, access fault 等信息。对于 Frontend 或 MemBlock 中的每个请求，都会由 ITLB 或 DTLB 发送回复，通 `tlb_requestor(i)_resp_valid` 表示回复有效。

在南湖架构中，虽然 SuperPage 和 NormalPage 在物理上都采用寄存器堆实现，但 SuperPage 是 16 项全相联结构，NormalPage 是直接相联结构。在从直接相联的 NormalPage 读出数据之后，还需要进行 tag 的比较。尽管 SuperPage 全相联项数为 16，但每次只可能命中一项，并通过 `hitVec` 标记命中，选择 SuperPage 中读出的数据。NormalPage 读出数据 +tag 比较的时间要比 SuperPage 读出数据 + 选择数据长很多。因此，从时序方面考虑，dtlb 会向 MemBlock 返回 `fast_miss` 信号，表示 SuperPage 未命中；miss 信号表示 SuperPage 和 NormalPage 均未命中。

同时，在南湖架构中，由于 DTLB 的 PMP & PMA 检查时序紧张，需要将 PMP 分为动态检查和静态检查两部分。（参见 5.4 节）当 L2 TLB 的页表项回填入 DTLB 时，同时将回填的页表项送给 PMP 和 PMA 进行权限检查，将检查得结果同时存储在 DTLB 中，DTLB 需要额外向 MemBlock 返回表示静态检查有效的信号以及检查结果。

需要注意的是，昆明湖架构优化了 TLB 查询的项配置和相应时序，目前 `fast_miss` 被取消，且无需额外的静态 PMP & PMA 检查。但可能后续由于时序或其他原因重新恢复，因此出于文档的完整和兼容性将前两段保留。昆明湖架构取消了 `fast_miss` 以及静态 PMP & PMA 检查，请再次注意。

18.2.2.4 阻塞式访问和非阻塞式访问

在南湖架构中，前端取指对 ITLB 的需求为阻塞式访问，而后端访存对 DTLB 的需求为非阻塞式访问。事实上，TLB 本体是非阻塞式访问，并不存储请求的信息。TLB 采用阻塞式访问或非阻塞式访问的原因是请求来源的需求，前端取指当 TLB miss 后，需要等待 TLB 取回结果，才能将指令送至处理器后端进行处理，呈现阻塞的效果；而访存操作可以乱序调度，当一个请求缺失后，可以调度另一个 load / store 指令继续执行，因此呈现出非阻塞式的效果。

南湖架构的上述功能通过 TLB 实现，TLB 会通过一些控制逻辑，当 ITLB 发生缺失后，持续等待通过 PTW 取回页表项。昆明湖的上述功能通过 ICache 保证，当 ITLB 发生缺失、并报给 ICache 后，ICache 会持续重发同一条请求，直至 hit，保证非阻塞式访问的效果。

但需要注意，昆明湖架构的 ITLB 和 DTLB 都是非阻塞的，无论外部效果是阻塞式或非阻塞式，均由取指单元或访存单元控制。

18.2.2.5 L1 TLB 表项的存储结构

香山的 TLB 可以对组织结构进行配置，包括相联模式、项数及替换策略等。默认配置为：ITLB 和 DTLB 均为 48 项全相联结构，且均由寄存器堆实现（参见 5.1.2.3 节）。如果在同一拍对某地址同时读写，可以通过 bypass 直接得到结果。

参考的 ILTB 或 DTLB 配置：均采用全相联结构，项数 8 / 16 / 32 / 48。目前并不支持参数化修改全相联 / 组相联 / 直接映射的 TLB 结构，需要手动修改代码。

18.2.2.6 支持在 L1 TLB 内部判断虚存是否开启以及两个阶段翻译是否开启

香山支持 RISC-V 手册中的 Sv39 页表，虚拟地址长度为 39 位。香山的物理地址为 36 位，可参数化修改。

虚存是否开启需要根据特权级和 SATP 寄存器的 MODE 域等共同决定，这一判断在 TLB 内部完成，对 TLB 外透明。关于特权级的描述，参见 5.1.2.7 节；关于 SATP 的 MODE 域，香山的昆明湖架构只支持 MODE 域为 8，也就是 Sv39 分页机制，否则会上报 illegal instruction fault。在 TLB 外的模块（Frontend、LoadUnit、StoreUnit、AtomicsUnit 等）看来，所有地址都经过了 TLB 的地址转换。

当添加了 H 拓展后，地址翻译是否启用还需要判断是否有两阶段地址翻译，两阶段地址翻译开启有两个请求，第一个是此时执行的是虚拟化访存指令，第二个是虚拟化模式开启并且此时 VSATP 或者 HGATP 的 MODE 不为零。此时的翻译模式有以下几种。翻译模式用于在 TLB 中查找对应类型的页表以及向 L2TLB 发送的 PTW 请求。

表 18.13: 两阶段翻译模式

| VSATP Mode | HGATP Mode | 翻译模式 |
|------------|------------|----------------------|
| 非 0 | 非 0 | allStage，两个阶段翻译均有 |
| 非 0 | 0 | onlyStage1，只有第一阶段的翻译 |
| 0 | 非 0 | onlyStage2，只有第二阶段的翻译 |

18.2.2.7 L1 TLB 的特权级

根据 Riscv 手册要求，前端取指（ITLB）的特权级为当前处理器特权级，后端访存（DTLB）的特权级为访存执行有效特权级。当前处理器特权级和访存执行有效特权级均在 CSR 模块中判断，传递到 ITLB 和 DTLB 中。当前处理器特权级保存在 CSR 模块中；访存执行有效特权级由 mstatus 寄存器的 MPRV、MPV 和 MPP 位以及 hstatus 的 SPVP 共同决定。如果执行虚拟化访存指令，则访存执行有效特权级为 hstatus 的 SPVP 位保存的特权级，如果执行的指令不是虚拟化访存指令，MPRV 位为 0，则访存执行有效特权级和当前处理器特权级相同，访存执行有效虚拟化模式也与当前虚拟化模式一致；如果 MPRV 位为 1，则访存执行有效特权级为 mstatus 寄存器的 MPP 中保存的特权级，访存执行有效虚拟化模式位 hstatus 寄存器的 MPV 保存的虚拟化模式。ITLB 和 DTLB 的特权级如表所示。

表 18.14: ITLB 和 DTLB 的特权级

| 模块 | 特权级 |
|------|----------|
| ITLB | 当前处理器特权级 |

| 模块 | 特权级 |
|------|--|
| DTLB | 执行非虚拟化访存指令，如果 mstatus.MPRV=0，为当前处理器特权级和虚拟化模式；如果 mstatus.MPRV=1，为 mtstatus.MPP 保存的特权级和 hstatus.MPV 保存的虚拟化模式 |

18.2.2.8 发送 PTW 请求

当 L1 TLB 发生 miss 时，需要向 L2 TLB 发送 Page Table Walk 请求。由于 L1 TLB 和 L2 TLB 之间有比较长的物理距离，因此需要在中间加拍，称为 Repeater。另外，repeater 需要承担过滤掉重复请求，避免 L1 TLB 中出现重复项的功能。（参见 5.2 节）因此，ITLB 或 DTLB 的第一级 Repeater 也被称作 Filter。L1 TLB 通过 Repeater 向 L2 TLB 发送 PTW 请求与接收 PTW 回复。（参见 5.3 节）

18.2.2.9 DTLB 复制查询返回的物理地址

在物理实现中，Memblock 的 dcache 与 lsu 距离较远，如果在 LoadUnit 的 load_s1 阶段产生 hitVec，再分别送往 dcache 和 lsu 会导致严重的时序问题。因此，需要并行在 dcache 和 lsu 附近产生两个 hitVec，分别送往 dcache 和 lsu。为配合解决 Memblock 的时序问题，DTLB 需要将查询得到的物理地址复制 2 份，分别送往 dcache 和 lsu，送往 dcache 和 lsu 的物理地址完全相同。

18.2.2.10 异常处理机制

ITLB 可能产生的异常包括 inst guest page fault、inst page fault 和 inst access fault，均交付给请求来源的 ICache 或 IFU 进行处理。DTLB 可能产生的异常包括：load guest page fault、load page fault、load access fault、store guest page fault、store page fault 和 store access fault，均交付给请求来源的 LoadUnits、StoreUnits 或 AtomicsUnit 进行处理。L1TLB 没有存储 gpaddr，所以出现客户机缺页时，需要重新进行 PTW。参见本文档的第 6 部分：异常处理机制。

这里需要对虚实地址转换相关的异常做额外补充说明，我们这里将异常分类如下：

1. 与页表相关的异常

1. 处于非虚拟化情况，或虚拟化的 VS-Stage 时，页表出现保留位不为 0 / 非对齐 / 写没有 w 权限等等（具体参见手册），需要上报 page fault
2. 处于虚拟化阶段的 G-Stage 时，页表出现保留位不为 0 / 非对齐 / 写没有 w 权限等等（具体参见手册），需要上报 guest page fault

2. 与虚拟地址或物理地址相关的异常

1. 地址翻译过程中，与虚拟地址或物理地址相关的异常。这部分检查会在 L2 TLB 的 PTW 过程中进行。
 1. 处于非虚拟化情况，或虚拟化的 all-Stage 时，需要检查 G-stage 的 gvpn。如果 hgatp 的 mode 为 8（代表 Sv39x4），则需要 gvpn 的 $(41 - 12 = 29)$ 位以上全部为 0；如果 hgatp 的 mode 为 9（代表 Sv48x4），则需要 gvpn 的 $(50 - 12 = 38)$ 位以上全部为 0。否则，会上报 guest page fault。
 2. 在地址翻译得到页表时，页表的 PPN 部分高 $(48-12=36)$ 位以上全部为 0。否则，会上报 access fault。
2. 原始地址中，虚拟地址或物理地址相关的异常，具体总结如下，这部分理论上均需要在 L1 TLB 做检查，但由于 ITLB 的 redirect 结果完全来自 Backend，因此 ITLB 相应的这部分异常会在 Backend 发送 redirect 给 Frontend 时做记录，并不会在 ITLB 中再次检查，请参考 Backend 对此处的说明。

1. Sv39 模式：包括开启虚存，且未开启虚拟化，此时 satp 的 mode 为 8；或开启虚存，且开启虚拟化，此时 vsatp 的 mode 为 8 这两种情况。此时需要满足 vaddr 的 [63:39] 位与 vaddr 的第 38 位符号相同，否则需要根据取指 / load / store 请求，分别报 instruction page fault, load page fault, store page fault。
2. Sv48 模式：包括开启虚存，且未开启虚拟化，此时 satp 的 mode 为 9；或开启虚存，且开启虚拟化，此时 vsatp 的 mode 为 9 这两种情况。此时需要满足 vaddr 的 [63:48] 位与 vaddr 的第 47 位符号相同，否则需要根据取指 / load / store 请求，分别报 instruction page fault, load page fault, store page fault。
3. Sv39x4 模式：开启虚存，且开启虚拟化，满足 vsatp 的 mode 为 0，且 hgatp 的 mode 为 8。（注：当 vsatp 的 mode 为 8 / 9, hgatp 的 mode 为 8 时，第二阶段地址翻译也为 Sv39x4 模式，也可能产生相应异常。但这部分属于“地址翻译过程中，与虚拟地址或物理地址相关的异常”，会在 L2 TLB 的页表遍历过程中进行处理，不属于 L1 TLB 的处理范畴。L1 TLB 只会额外处理“原始地址中，虚拟地址或物理地址相关的异常”）此时需要满足 vaddr 的 [63:41] 位全部为 0，否则需要根据取指 / load / store 请求，分别报 instruction guest page fault, load guest page fault, store guest page fault。
4. Sv48x4 模式：开启虚存，且开启虚拟化，满足 vsatp 的 mode 为 0，且 hgatp 的 mode 为 9。（注：当 vsatp 的 mode 为 8 / 9, hgatp 的 mode 为 9 时，第二阶段地址翻译也为 Sv48x4 模式，也可能产生相应异常。但这部分属于“地址翻译过程中，与虚拟地址或物理地址相关的异常”，会在 L2 TLB 的页表遍历过程中进行处理，不属于 L1 TLB 的处理范畴。L1 TLB 只会额外处理“原始地址中，虚拟地址或物理地址相关的异常”）此时需要满足 vaddr 的 [63:50] 位全部为 0，否则需要根据取指 / load / store 请求，分别报 instruction guest page fault, load guest page fault, store guest page fault。
5. Bare 模式：未开启虚存，此时 paddr = vaddr。由于香山处理器的物理地址目前限定为 48 位，因此对 vaddr 要求 [63:48] 位全部为 0，否则需要根据取指 / load / store 请求，分别报 instruction access fault, load access fault, store access fault。

为了支持对上述“原始地址中”的异常处理，L1 TLB 需要添加 fullva (64 bits) 和 checkfullva (1 bit) 的 input 信号。同时需要在 output 中添加 vaNeedExt 具体地：

1. checkfullva 并非 fullva 的控制信号。也就是说，fullva 的内容并不止在 checkfullva 拉高时才有效。
2. checkfullva 何时有效（需要拉高）
 1. 对于 ITLB, checkfullva 始终为 false，因此 chisel 生成 verilog 时，可能会将 checkfullva 优化掉，不会体现在 input 中。
 2. 对于 DTLB，对于所有 load / store / amo / vector 指令，在第一次由 Backend 发送至 MemBlock 时，需要做 checkfullva 检查。这里额外说明，“原始地址中，虚拟地址或物理地址相关的异常”是一个只针对 vaddr 的检查（对于 load / store 指令，vaddr 的计算一般为某寄存器的值 + imm 立即数计算得到的 64 bits 值），因此无需等待 TLB 命中，且当出现该检查的异常时，TLB 并不会返回 miss，代表该异常有效。因此，“在第一次由 Backend 发送至 MemBlock 时”，一定能够发现该异常并上报。对于非对齐访存，并不会进入 misalign buffer；对于 load 指令，并不会进入 load replay queue；对于 store 指令，也不会由保留站重发。因此，如果“一次由 Backend 发送至 MemBlock 时”并未发现该异常，由 load replay 重发时，一定不会出现该异常，无需做 checkfullva 检查。对于预取指令，不会拉高 checkfullva。
3. fullva 何时有效（在什么时候被使用）

1. 除一种特定情况外, fullva 只在 checkfullva 为高时有效, 代表要检查的完整 vaddr。这里需要说明, 一条 load / store 指令, 计算得到的原始 vaddr 为 64 位 (寄存器读出来的值就是 64 位的); 但查询 TLB 只会用到低 48 / 50 位 (Sv48 / Sv48x4), 查询异常需要用到完整的 64 位。
2. 特定情况: 非对齐指令出现 gpf, 需要获取 gpaddr。目前访存侧对非对齐异常的处理逻辑如下:
 1. 例如, 原始 vaddr 为 0x81000ffb, 要 ld 8 bytes 数据
 2. misalign buffer 会将该指令拆成 vaddr 为 0x81000ff8 (load 1) 和 0x81001000 (load 2) 的两条 load, 且这两条 load 并不属于同一虚拟页
 3. 对于 load 1, 此时传入 TLB 的 vaddr 为 0x81000ff8, fullva 总为原始 vaddr 0x81000ffb; 对于 load 2, 此时传入 TLB 的 vaddr 为 0x81001000, fullva 总为原始 vaddr 0x81000ffb
 4. load 1 如果出现异常, 写入 tval 寄存器的 offset 约定为原始 addr 的 offset (即 0xffb); load 2 如果出现异常, 写入 tval 寄存器的 offset 约定为下一页的起始值 (0x000)。对于虚拟化场景的 onlyStage2 情况, 此时 gpaddr = 出现异常的 vaddr。因此, 对于跨页的非对齐请求、且跨页后的地址出现异常, gpaddr 的生成只会用到 vaddr (此时 offset 其实为 0x000), 不会用到 fullva; 对于非跨页的非对齐请求, 或对于跨页、且原始地址出现异常的非对齐请求, gpaddr 的生成会用到 fullva 的 offset (0xffb)。这里 fullva 始终是有效的, 和 checkfullva 是否拉高无关。
4. vaNeedExt 何时有效 (在什么情况被使用)
 1. 在访存队列 load queue / store queue 中, 处于节约面积的考虑, 会将 64 位原始地址截断至 50 位保存, 但在写入 *tval 寄存器时, 需要写入 64 位值。上文中已经介绍过, 对于 “原始地址中, 虚拟地址或物理地址相关的异常” 的异常, 要保留原始完整 64 位地址; 而对于其他页表相关的异常, 地址本身高位是满足要求的。例如:
 - fullva = 0xffff,ffff,8000,0000; vaddr = 0xffff,8000,0000。Mode 为非虚拟化的 Sv39。这里原始地址并未产生异常, 假设这是一个 load 请求, 第一次访问 TLB 时 miss, 因此该 load 会进入 load replay queue 等待重发, 且地址会被截断为 50 位。等待 load 指令重发后, 发现该页表的 V 位为 0, 发生 page fault, 需要将 vaddr 写入 *tval 寄存器。由于地址在 load queue replay 中已经被截断, 因此需要做符号位扩展 (例如 Sv39 情况, 即将 39 位以上扩展为 38 位的值), 返回的 vaNeedExt 拉高。
 - fullva = 0x0000,ffff,8000,0000; vaddr = 0xffff,8000,0000。Mode 为非虚拟化的 Sv39。这里可以发现原始地址就产生了异常, 我们会将该地址直接写入对应的 exception buffer 中 (exception buffer 会保存完整的 64 位值)。此时需要直接将 0x0000,ffff,8000,0000 原始值写入 *tval, 不能做符号位扩展, vaNeedExt 为低。

18.2.2.11 支持 pointer masking 扩展

目前香山处理器支持 pointer masking 扩展。

pointer masking 扩展的本质是将访存的 fullva 由 “寄存器堆的值 + imm 立即数” 这个原始值, 变为 “effective vaddr” 这个高位可能被忽略的值。当 pmm 的值为 2 时, 会忽略高 7 位; 当 pmm 的值为 3 时, 会忽略高 16 位。pmm 为 0 代表不忽略高位, pmm 为 1 是保留位。

pmm 的值可能来自于 mseccfg/menvcfg/henvcfg/senvcfg 的 PMM ([33:32]) 位, 也可能来自于 hstatus 寄存器的 HUPMM ([49:48]) 位。具体怎样选择如下:

1. 对于前端取指请求, 或者一条手册规定的 hlvx 指令, 不会使用 pointer masking (pmm 为 0)
2. 当前访存有效特权级 (dmode) 为 M 态, 选择 mseccfg 的 PMM ([33:32]) 位
3. 非虚拟化场景, 且当前访存有效特权级为 S 态 (HS), 选择 menvcfg 的 PMM ([33:32]) 位

4. 虚拟化场景，且当前访存有效特权级为 S 态 (VS)，选择 henvcfg 的 PMM ([33:32]) 位
5. 是虚拟化指令，且当前处理器特权级 (imode) 为 U 态，选择 hstatus 的 HUPMM ([49:48]) 位
6. 其余 U 态场景，选择 senvcfg 的 PMM ([33:32]) 位

由于 pointer masking 的只针对访存生效，并不适用于前端取指。因此 ITLB 不存在“effective vaddr”的概念，也不会在端口中引入 CSR 传入的这些信号。

由于这些地址的高位只在上文提到的“原始地址中，虚拟地址或物理地址相关的异常”中被检查使用，因此对于屏蔽高位的情况，我们直接让其不会触发异常即可。具体地：

1. 对于开启虚存的非虚拟化场景，或虚拟化场景的非 onlyStage2 (vsatp 的 mode 不为 0) 情况；根据 pmm 的值为 2 或 3，分别对地址的高 7 或 16 位做符号扩展
2. 对于虚拟化场景的 onlyStage2 情况，或未开启虚存，根据 pmm 的值为 2 或 3，分别对地址的高 7 或 16 位做零扩展

18.2.2.12 支持 TLB 压缩

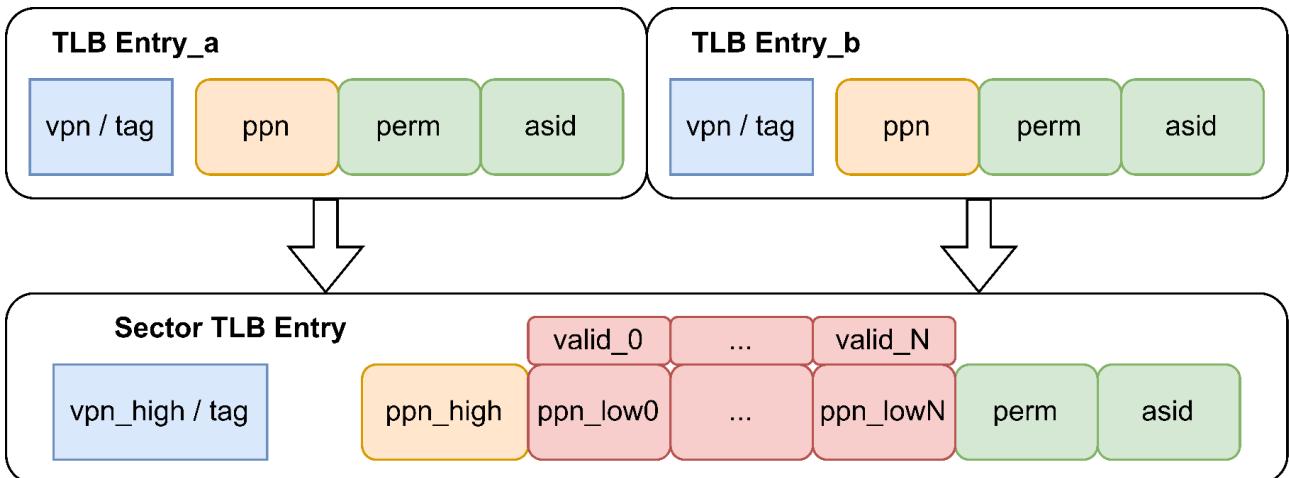


图 18.11: TLB 压缩示意图

昆明湖架构支持 TLB 压缩，每项 TLB 压缩会保存连续 8 项的页表项，如图上所示。TLB 压缩的理论基础是，操作系统在分配页时，由于伙伴分配机制等原因，会更倾向将连续的物理页分配给连续的虚拟页。虽然随着程序的不断运行，页分配从有序逐渐趋向于无序，但这种页的相联性普遍存在。因此，可以将多个连续的页表项在硬件中合并成一个 TLB 项，从而起到提高 TLB 容量的作用。

也就是说，对于虚拟页号高位相同的页表项，当这些页表项的物理页号高位和页表属性也相同时，可以将这些页表项压缩为一项保存，从而提升 TLB 的有效容量。压缩后的 TLB 项共用物理页号高位以及页表属性位，每个页表单独拥有物理页号低位，并通过 valid 表示该页表在压缩后的 TLB 项中有效，如表 5.1.8。

表 5.1.8 展示了压缩前后的对比，压缩前的 tag 即为 vpn，压缩后的 tag 为 vpn 的高 24 位，低 3 位无需保存，事实上连续 8 项页表的第 i 项，i 即为 tag 的低 3 位。ppn 高 21 位相同，ppn_low 分别保存 8 项页表的 ppn 低 3 位。Valididx 表示这 8 项页表的有效性，只有 valididx(i) 为 1 时才有效。pteidx(i) 代表原始请求对应的第 i 项，即原始请求 vpn 的低 3 位的值。

这里举例进行说明。例如，某 vpn 为 0x0000154，低三位为 100，即 4。当回填入 L1 TLB 后，会将 vpn 为 0x0000150 到 0x0000157 的 8 项页表均回填，且压缩为 1 项。例如，vpn 为 0x0000154 的 ppn 高 21 位为 PPN0，页表属性位为 PERM0，如果这 8 项页表的第 i 项 ppn 高 21 位和页表属性也为 PPN0 和 PERM0，则

valididx(i) 为 1，通过 ppn_low(i) 保存第 i 项页表的低 3 位。另外，pteidx(i) 代表原始请求对应的第 i 项，这里原始请求的 vpn 低三位为 4，因此 pteidx(4) 为 1，其余 pteidx(i) 均为 0。

另外，TLB 不会对查询结果为大页（1GB、2MB）情况进行压缩。对于大页，返回时会将 valididx(i) 的每一位都设置为 1，根据页表查询规则，大页事实上不会使用 ppn_low，因此 ppn_low 的值可以为任意值。

表 18.15: TLB 压缩前后每项存储的内容

| 是否压缩 | tag | asid | level | ppn | perm | valididx | pteidx | ppn_low |
|------|------|------|-------|------|------|----------|--------|---------|
| 否 | 27 位 | 16 位 | 2 位 | 24 位 | 页表属性 | 不保存 | 不保存 | 不保存 |
| 是 | 24 位 | 16 位 | 2 位 | 21 位 | 页表属性 | 8 位 | 8 位 | 8×3 位 |

在实现 TLB 压缩后，L1 TLB 的命中条件由 TAG 命中，变为 TAG 命中（vpn 高位匹配），同时还需满足用 vpn 低 3 位索引的 valididx(i) 有效。PPN 由 ppn（高 21 位）与 ppn_low(i) 拼接得到。

但注意的是，添加 H 拓展后，L1TLB 的项分为四种类型，TLB 压缩机制虚拟化的 TLB 项中不启用（但 TLB 压缩在 L2TLB 中仍然使用），接下来会详细介绍这四种类型。

18.2.2.13 存储四种类型的 TLB 项

在添加 H 拓展的 L1TLB 中对 TLB 项进行了修改，如图 18.12 所示。

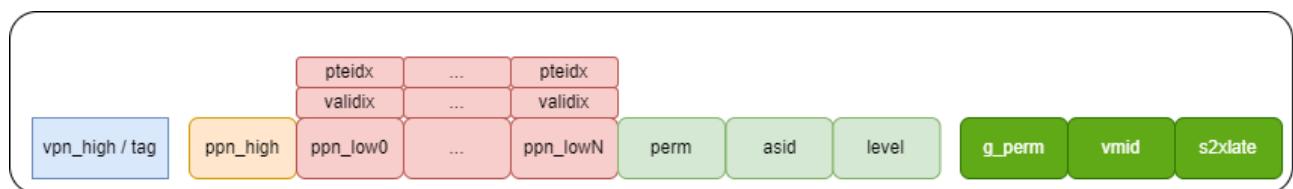


图 18.12: TLB 项示意图

与原先的设计相比，新增了 g_perm、vmid、s2xlate，其中 g_perm 用来存储第二阶段页表的 perm，vmid 用来存储第二阶段页表的 vmid，s2xlate 用来区分 TLB 项的类型。根据 s2xlate 的不同，TLB 项目存储的内容也有所不同。

表 18.16: TLB 项的类型

| 类型 | s2xlate | tag | ppn | perm | g_perm | level |
|------------|---------|-----------------|-----------------|-----------------------|------------------|---------------------|
| noS2xlate | b00 | 非虚拟化下的虚 拟页号 | 非虚拟化下的物 理页号 | 非虚拟化下 的页表项 perm | 不使用 | 非虚拟化下的 页表项 level |
| allStage | b11 | 第一阶段页表的 虚拟页号 | 第二阶段页表的 物理页号 | 第一阶段页 表的 perm | 第二阶段页 表的 perm | 两阶段翻译中 最大的 level |
| onlyStage1 | b01 | 第一阶段页表的 虚拟页号 | 第一阶段页表的 物理页号 | 第一阶段页 表的 perm | 不使用 | 第一阶段页表 的 level |
| onlyStage2 | b10 | 第二阶段页表的 虚拟页号 | 第二阶段页表的 物理页号 | 不使用 | 第二阶段页 表的 perm | 第二阶段页表 的 level |

其中 TLB 压缩技术在 noS2xlate 和 onlyStage1 中启用，在其他情况下不启用，allStage 和 onlyS2xlate 情况下，L1TLB 的 hit 机制会使用 pteidx 来计算有效 pte 的 tag 与 ppn，这两种情况在重填的时候也会有所区别。此外，asid 在 noS2xlate、allStage、onlyStage1 中有效，vmid 在 allStage、onlyStage2 中有效。

18.2.2.14 TLB refill 将两个阶段的页表进行融合

添加了 H 拓展后的 MMU，PTW 返回的结构分为三部分，第一部分 s1 是原先设计中的 PtwSectorResp，存储第一阶段翻译的页表，第二部分 s2 是 HptwResp，存储第二阶段翻译的页表，第三部分是 s2xlate，代表这次 resp 的类型，仍然分为 noS2xlate、allStage、onlyStage1 和 onlyStage2，如图 18.13。其中 PtwSectorEntry 是采用了 TLB 压缩技术的 PtwEntry，两者的主要区别是 tag 和 ppn 的长度

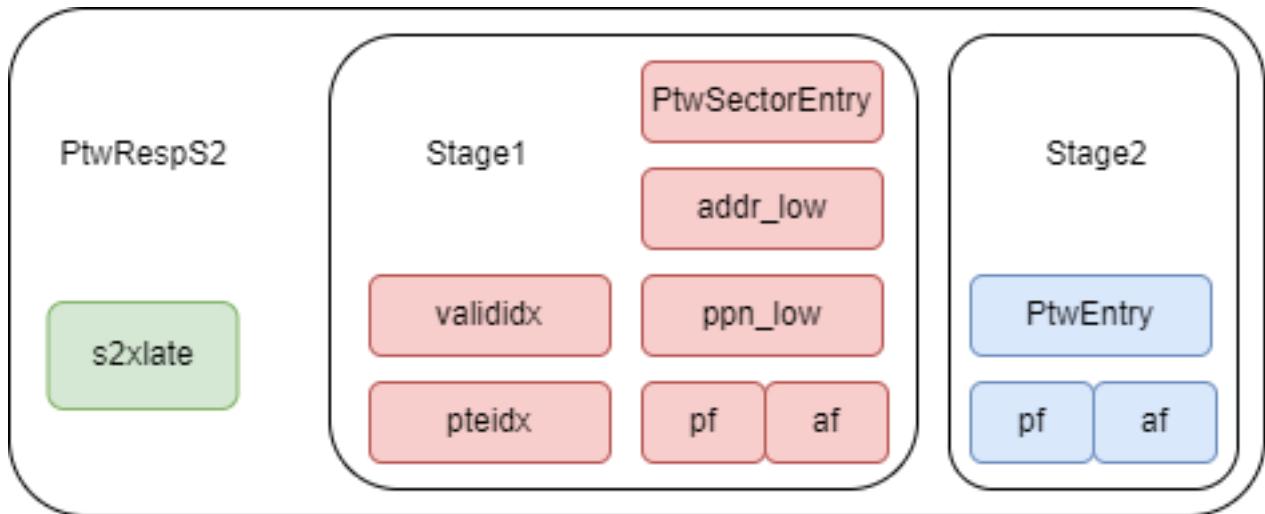


图 18.13: PTW resp 结构示意图

对于 noS2xlate 和 onlyStage1 的情况，只需要将 s1 的结果填入 TLB 项中即可，写入方法与原先的设计类似，将返回的 s1 的对应字段填入 entry 的对应字段即可。需要注意的是，noS2xlate 的时候，vmid 字段无效。

对于 onlyS2xlate 的情况，我们将 s2 的结果给填入 TLB 项，这里由于要符合 TLB 压缩的结构，所以需要进行一些特殊处理。首先该项的 asid、perm 不使用，所以我们不关心此时填入的什么值，vmid 填入 s1 的 vmid（由于 PTW 模块无论什么情况都会填写这个字段，所以可以直接使用这个字段写入）。将 s2 的 tag 填入 TLB 项的 tag，pteidx 根据 s2 的 tag 的低 sectortlbwidth 位来确定，如果 s2 是大页，那么 TLB 项的 valididx 均为有效，否则 TLB 项的 pteidx 对应 valididx 有效。关于 ppn 的填写，复用了 allStage 的逻辑，将在 allStage 的情况下介绍。

对于 allStage，需要将两阶段的页表进行融合，首先根据 s1 填入 tag、asid、vmid 等，由于只有一个 level，level 填入 s1 和 s2 最大的值，这是考虑到如果存在第一阶段是大页和第二阶段是小页的情况，可能会导致某个地址进行查询的时候 hit 大页，但实际已经超出了第二阶段页表的范围，对于这种请求的 tag 也要进行融合，比如第一个 tag 是一级页表，第二个 tag 是二级页表，我们需要取第一个 tag 的第一级页号与第二个 tag 的第二级页号拼合（第三级页号可以直接补零）得到新页表的 tag。此外，还需要填入 s1 和 s2 的 perm 以及 s2xlate，对于 ppn，由于我们不保存客户机物理地址，所以对于第一阶段小页和第二阶段大页的情况，如果直接存储 s2 的 ppn 会导致查询到该页表时计算得到的物理地址出错，所以首先要根据 s2 的 level 将 s2 的 tag 与 ppn 拼接一下，s2ppn 为高位 ppn，s2ppn_tmp 则是为了计算低位构造出来的，然后高位填入 TLB 项的 ppn 字段，低位填入 TLB 项的 ppn_low 字段。

18.2.2.15 TLB 项的 hit 的判断逻辑

L1TLB 中使用的 hit 有三种，查询 TLB 的 hit，填写 TLB 的 hit，以及 PTW 请求 resp 时的 hit。

对于查询 TLB 的 hit，新增了 vmid, hasS2xlate, onlyS2, onlyS1 等参数。Asid 的 hit 在第二阶段翻译的时候一直为 true。H 拓展中增加了 pteidx hit，在小页并且在 allStage 和 onlyS2 的情况下启用，用来屏蔽掉 TLB 压缩机制。

对于填写 TLB 的 hit (wbhit)，输入是 PtWRespS2，需要判断当前的进行对比的 vpn，如果是只有第二阶段的翻译，则使用 s2 的 tag 的高位，其他情况使用 s1vpn 的 tag，然后在低 sectortlbwidth 位补上 0，然后使用 vpn 与 TLB 项的 tag 进行对比。H 拓展对 wb_valid 的判断进行了修改，并且新增了 pteidx_hit 和 s2xlate_hit。如果是只有第二阶段翻译的 PTW resp，则 wb_valididx 根据 s2 的 tag 来确定，否则直接连接 s1 的 valididx。s2xlate hit 则是对比 TLB 项的 s2xlate 与 PTW resp 的 s2xlate，用来筛选 TLB 项的类型。pteidx_hit 则是为了无效 TLB 压缩，如果是只有第二阶段翻译，则对比 s2 的 tag 的低位与 TLB 项的 pteidx，其他的两阶段翻译情况则对比 TLB 项的 ptedix 和 s1 的 pteidx。

对于 PTW 请求的 resp hit，主要用于 PTW resp 的时候判断此时 TLB 发送的 PTW req 是否正好与该 resp 对应或者判断在查询 TLB 的时候 PTW resp 是否是 TLB 这个请求需要的 PTW 结果。该方法在 PtWRespS2 中定义，在该方法内部分为三种 hit，对于 noS2_hit (noS2xlate)，只需要判断 s1 是否 hit 即可，对于 onlyS2_hit (onlyStage2)，则判断 s2 是否 hit 即可，对于 all_onlyS1_hit (allStage 或者 onlyStage1)，需要重新设计 vpnhit 的判断逻辑，不能简单判断 s1hit，判断 vpn_hit 的 level 应该取用 s1 和 s2 的最大值，然后根据 level 来判断 hit，并且增加 vasid (来自 vsatp) 的 hit 和 vmid 的 hit。

18.2.2.16 支持客户机缺页后重新发送 PTW 获取 gpaddr

由于 L1TLB 不保存翻译结果中的 gpaddr，所以当查询 TLB 项后出现 guest page fault 的时候需要重新进行 PTW 获取 gpaddr，此时 TLB resp 仍然是 miss。这里新增了一些寄存器。

表 18.17: 获取 gpaddr 的新增 Reg

| 名称 | 类型 | 作用 |
|-----------------|--------|-----------------------------------|
| need_gpa | Bool | 表示此时有一个请求正在获取 gpaddr |
| need_gpa_robidx | RobPtr | 获取 gpaddr 的请求的 robidx |
| need_gpa_vpn | vpnLen | 获取 gpaddr 的请求的 vpn |
| need_gpa_gvpn | vpnLen | 存储获取的 gpaddr 的 gvpn |
| need_gpa_refill | Bool | 表示该请求的 gpaddr 已经被填入 need_gpa_gvpn |

当一个 TLB 请求查询出来的 TLB 项出现了客户机缺页，则需要重新进行 PTW，此时会把 need_gpa 有效，将请求的 vpn 填入 need_gpa_vpn，将请求的 robidx 填入 need_gpa_robidx，初始化 resp_gpa_refill 为 false。当 PTW resp，并且通过 need_gpa_vpn 判断是之前发送的获取 gpaddr 的请求，则将 PTW resp 的 s2 tag 填入 need_gpa_gvpn，并且将 need_gpa_refill 有效，表示已经获取到 gpaddr 的 gvpn，当之前的请求重新进入 TLB 的时候，就可以使用这个 need_gpa_gvpn 来计算出 gpaddr 并且返回，当一个请求完成以上过程后，将 need_gpa 无效掉。这里的 resp_gpa_refill 依旧有效，所以重填的 gvpn 可能被其他的 TLB 请求使用（只要跟 need_gpa_vpn 相等）。

此外可能出现 redirect 的情况，导致整个指令流变化，之前获取 gpaddr 的请求不会再进入 TLB，所以如果出现 redirect 就根据我们保存的 need_gpa_robidx 来判断是否需要无效掉 TLB 内与获取 gpaddr 有关的寄存器。

此外为了保证获取 gpaddr 的 PTW 请求返回的时候不会 refill TLB，在发送 PTW 请求的时候添加了一个新的 output 信号 getGpa，该信号传递的路径与 memidx 类似，可以参考 memidx，该信号会传入 Repeater 内，当 PTW resp 回 TLB 的时候，该信号也会发送回来，如果该信号有效，则表明这个 PTW 请求只是为了获取 gpaddr，所以此时不会重填 TLB。

关于发生 guest page fault 后获取 gpaddr 的处理流程，这里对于一些关键点做再次说明：

1. 可以将获取 gpa 的机制看作一个只有 1 项的 buffer，当某个请求发生 guest page fault 时，即向该 buffer 写入 need_gpa 的相应信息；直至 need_gpa_vpn_hit && resp_gpa_refill 条件有效，或传入 flush (itlb) / redirect (dtlb) 信号刷新 gpa 信息。
- need_gpa_vpn_hit 指的是：在某个请求发生 guest page fault 后，会将 vpn 信息写入 need_gpa_vpn 中。如果相同的 vpn 再次查询 TLB，need_gpa_vpn_hit 信号会拉高，代表获取的 gpaddr 与原始 get_gpa 请求相对应。如果此时 resp_gpa_refill 也为高，代表 vpn 已经获取得到对应的 gpaddr，可以将 gpaddr 返回给前端取指 / 后端访存进行异常处理。
- 因此，对于前端或访存的任意请求，如果触发 gpa，则后续一定需要满足以下两个条件之一：
 1. 该触发 gpa 的请求一定能够重发（TLB 在获取 gpaddr 前，会一直对该请求返回 miss，直至得到 gpaddr 结果为止）
 2. 需要通过向 TLB 传入 flush 或 redirect 信号，将该 gpa 请求冲刷掉。具体地，对于所有可能的请求：
 1. ITLB 的取指请求：如果出现 gpf 的取指请求处于推测路径上，且发现出现错误的推测，则会通过 flushPipe 信号进行刷新（包括后端 redirect、或前端多级分支预测器出现后级预测器的预测结果更新前级预测器的预测结果等）；对于其他情况，由于 ITLB 会对该请求返回 miss，前端会保证重发相同 vpn 的请求。
 2. DTLB 的 load 请求：如果出现 gpf 的 load 请求处于推测路径上，且发现出现错误的推测，则会通过 redirect 信号进行刷新（需要判断出现 gpf 的 robidx 与传入 redirect 的 robidx 的前后关系）；对于其他情况，由于 DTLB 会对该请求返回 miss，同时会将返回 tlbreplay 信号拉高，使 load queue replay 一定能够重发该请求。
 3. DTLB 的 store 请求：如果出现 gpf 的 store 请求处于推测路径上，且发现出现错误的推测，则会通过 redirect 信号进行刷新（需要判断出现 gpf 的 robidx 与传入 redirect 的 robidx 的前后关系）；对于其他情况，由于 DTLB 会对该请求返回 miss，后端一定会调度该 store 指令再次重发该请求。
 4. DTLB 的 prefetch 请求：返回的 gpf 信号会拉高，代表该预取请求的地址发生 gpf，但不会写入 gpa* 一系列寄存器，不会触发查找 gpaddr 机制，无需考虑。

2. 在目前的处理机制中，需要保证发生 gpf 且等待 gpa 的该 TLB 项在等待 gpa 过程中不会被替换出去。这里我们简单地在出现等待 gpa 情况时，阻止 TLB 的回填，从而避免替换操作发生。由于发生 gpf 时本就需要进行异常处理程序，且在此之后的指令会被重定向冲刷掉，因此在等待 gpa 过程中阻止回填并不会导致性能问题。

18.2.3 整体框图

L1 TLB 的整体框图如图 18.14 所述，包括绿框中的 ITLB 和 DTLB。ITLB 接收来自 Frontend 的 PTW 请求，DTLB 接收来自 Memblock 的 PTW 请求。来自 Frontend 的 PTW 请求包括 ICache 的 3 个请求和 IFU 的 1 个请求，来自 Memblock 的 PTW 请求包括 LoadUnit 的 2 个请求（AtomicsUnit 占用 LoadUnit 的 1 个

请求通道)、L1 Load Stream & Stride prefetch 的 1 个请求, StoreUnit 的 2 个请求, 以及 SMSPrefetcher 的 1 个请求。

在 ITLB 和 DTLB 查询得到结果后, 都需要进行 PMP 和 PMA 检查。由于 L1 TLB 的面积较小, 因此 PMP 和 PMA 寄存器的备份并不存储在 L1 TLB 内部, 而是存储在 Frontend 或 Memblock 中, 分别为 ITLB 和 DTLB 提供检查。ITLB 和 DTLB 缺失后, 需要经过 repeater 向 L2 TLB 发送查询请求。

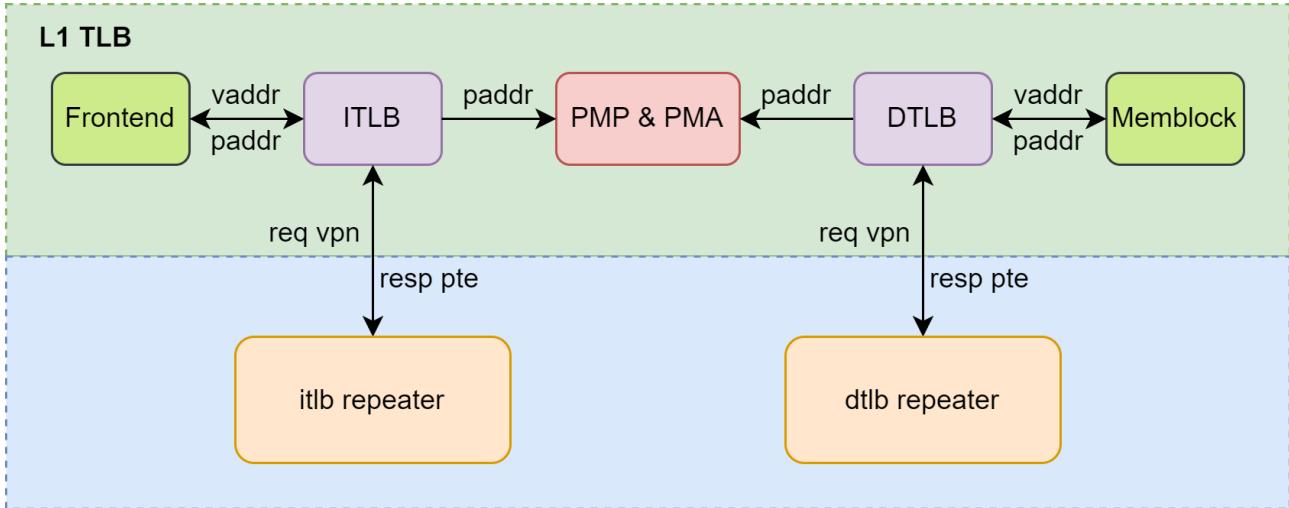


图 18.14: L1 TLB 模块整体框图

18.2.4 接口时序

18.2.4.1 ITLB 与 Frontend 的接口时序

18.2.4.1.1 Frontend 向 ITLB 发送的 PTW 请求命中 ITLB

Frontend 向 ITLB 发送的 PTW 请求在 ITLB 命中时, 时序图如图 18.15 所示。

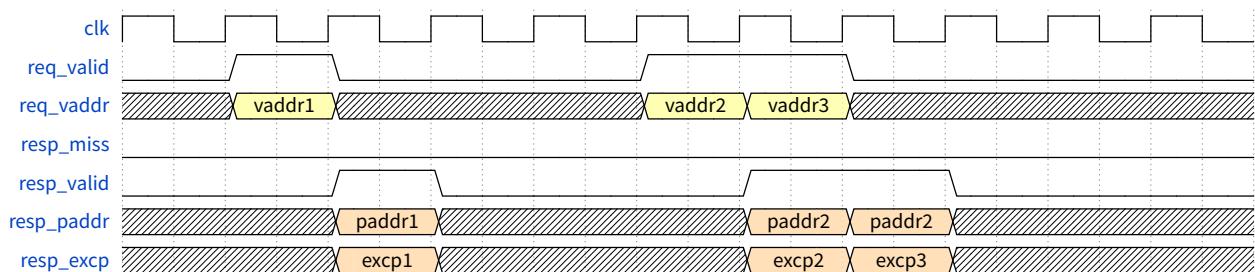


图 18.15: Frontend 向 ITLB 发送的 PTW 请求命中 ITLB 的时序图

当 Frontend 向 ITLB 发送的 PTW 请求在 ITLB 命中时, resp_miss 信号保持为 0。req_valid 为 1 后的下一个时钟上升沿, ITLB 会将 resp_valid 信号置 1, 同时向 Frontend 返回虚拟地址转换后的物理地址, 以及是否发生 guest page fault、page fault 和 access fault 等信息。时序描述如下:

- 第 0 拍: Frontend 向 ITLB 发送 PTW 请求, req_valid 置 1。
- 第 1 拍: ITLB 向 Frontend 返回物理地址, resp_valid 置 1。

18.2.4.1.2 Frontend 向 ITLB 发送的 PTW 请求未命中 ITLB

Frontend 向 ITLB 发送的 PTW 请求在 ITLB 未命中时，时序图如图 18.16 所示。

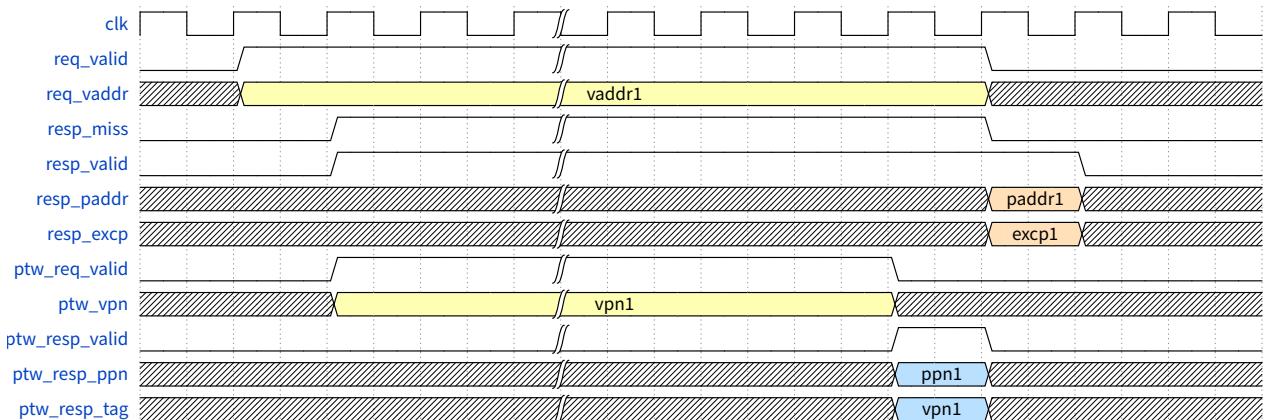


图 18.16: Frontend 向 ITLB 发送的 PTW 请求未命中 ITLB 的时序图

当 Frontend 向 ITLB 发送的 PTW 请求在 ITLB 中未命中时，下一拍会向 ITLB 返回 resp_miss 信号，表示 ITLB 未命中。此时 ITLB 的该条 requestor 通道不再接收新的 PTW 请求，由 Frontend 重复发送该请求，直至查询得到 L2 TLB 或内存中的页表并返回。（请注意，“ITLB 的该条 requestor 通道不再接收新的 PTW 请求”由 Frontend 控制，也就是说，无论 Frontend 选择不重发 miss 的请求，或重发其他请求，Frontend 的行为对 TLB 来说是透明的。如果 Frontend 选择发送新请求，ITLB 会将旧请求直接丢失掉。）

当 Frontend 向 ITLB 发送的 PTW 请求在 ITLB 中未命中时，下一拍会向 ITLB 返回 resp_miss 信号，表示 ITLB 未命中。此时 ITLB 的该条 requestor 通道不再接收新的 PTW 请求，由 Frontend 重复发送该请求，直至查询得到 L2 TLB 或内存中的页表并返回。（请注意，“ITLB 的该条 requestor 通道不再接收新的 PTW 请求”由 Frontend 控制，也就是说，无论 Frontend 选择不重发 miss 的请求，或重发其他请求，Frontend 的行为对 TLB 来说是透明的。如果 Frontend 选择发送新请求，ITLB 会将旧请求直接丢失掉。）

当 ITLB 未命中时，会向 L2 TLB 发送 PTW 请求，直至查询得到结果。ITLB 与 L2 TLB 的时序交互，以及向 Frontend 返回的物理地址等信息参见图 4.4 的时序图以及如下的时序描述：

- 第 0 拍：Frontend 向 ITLB 发送 PTW 请求，req_valid 置 1。
- 第 1 拍：ITLB 查询得到 miss，向 Frontend 返回 resp_miss 为 1，resp_valid 置 1。同时，在当拍 ITLB 向 L2 TLB（事实上为 itlbrepeater1）发送 PTW 请求，ptw_req_valid 置 1。
- 第 X 拍：L2 TLB 向 ITLB 返回 PTW 回复，包括 PTW 请求的虚拟页号、得到的物理页号、页表信息等，ptw_resp_valid 为 1。在该拍 ITLB 已经收到 L2 TLB 的 PTW 回复，ptw_req_valid 置 0。
- 第 X+1 拍：ITLB 此时命中，resp_valid 为 1，resp_miss 为 0。ITLB 向 Frontend 返回物理地址以及是否发生 access fault、page fault 等信息。
- 第 X+2 拍：ITLB 向 Frontend 返回的 resp_valid 信号置 0。

18.2.4.2 DTLB 与 Memblock 的接口时序

18.2.4.2.1 Memblock 向 DTLB 发送的 PTW 请求命中 DTLB

Memblock 向 DTLB 发送的 PTW 请求在 DTLB 命中时，时序图如图 18.17 所示。

当 Memblock 向 DTLB 发送的 PTW 请求在 DTLB 命中时，resp_miss 信号保持为 0。req_valid 为 1 后的下一个时钟上升沿，DTLB 会将 resp_valid 信号置 1，同时向 Memblock 返回虚拟地址转换后的物理地址，以

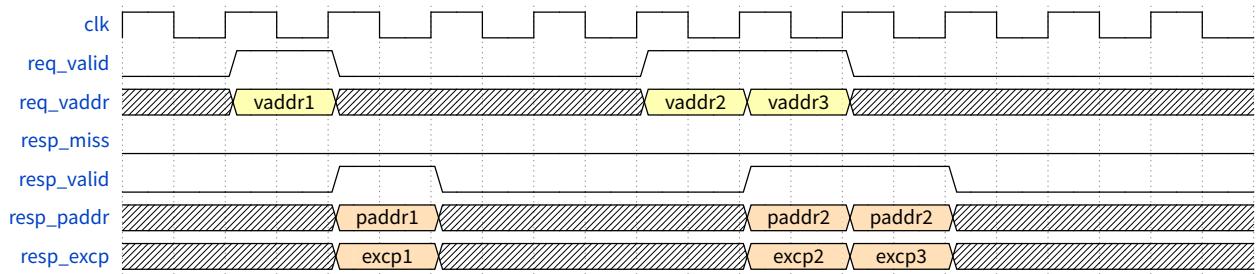


图 18.17: Memblock 向 DTLB 发送的 PTW 请求命中 DTLB 的时序图

及是否发生 page fault 和 access fault 等信息。时序描述如下：

- 第 0 拍：Memblock 向 DTLB 发送 PTW 请求，req_valid 置 1。
- 第 1 拍：DTLB 向 Memblock 返回物理地址，resp_valid 置 1。

18.2.4.2.2 Memblock 向 DTLB 发送的 PTW 请求未命中 DTLB

DTLB 和 ITLB 相同，均为非阻塞式访问（即 TLB 内部并不包括阻塞逻辑，如果请求来源保持不变，即缺失后持续重发同一条请求，则呈现出类似阻塞式访问的效果；如果请求来源在收到缺失的反馈后，调度其他不同请求查询 TLB，则呈现出类似非阻塞式访问的效果）。和前端取指不同，当 Memblock 向 DTLB 发送的 PTW 请求未命中 DTLB，并不会阻塞流水线，DTLB 会在 req_valid 的下一拍向 Memblock 返回请求 miss 以及 resp_valid 的信号，在 Memblock 在收到 miss 信号后可以进行调度，继续查询其他请求。

在 Memblock 访问 DTLB 发生 miss 后，DTLB 会向 L2 TLB 发送 PTW 请求，查询来自 L2 TLB 或内存中的页表。DTLB 通过 Filter 向 L2 TLB 传递请求，Filter 可以合并 DTLB 向 L2 TLB 发送的重复请求，保证 DTLB 中不出现重复项并提高 L2 TLB 的利用率。Memblock 向 DTLB 发送的 PTW 请求未命中 DTLB 的时序图如图 18.18 所示，该图只描述了从请求 miss 到 DTLB 向 L2 TLB 发送 PTW 请求的过程。

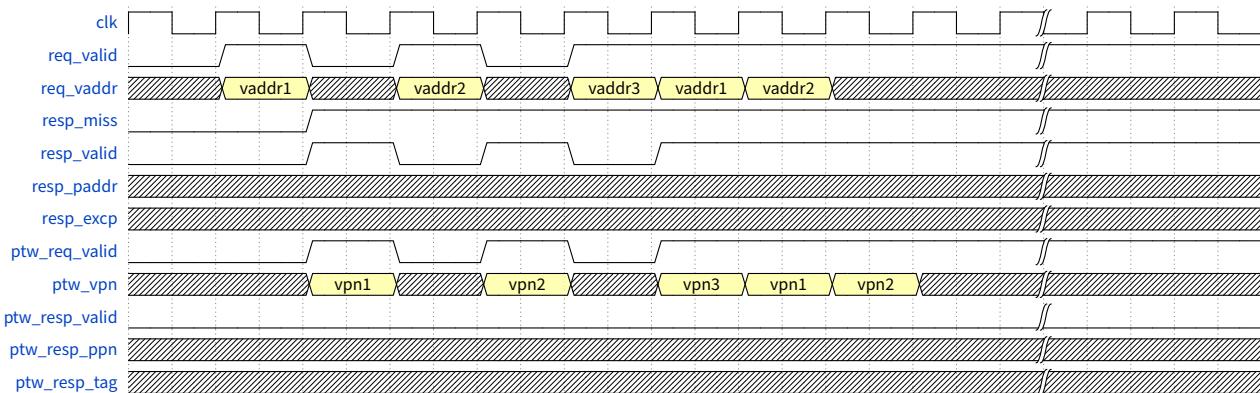


图 18.18: Memblock 向 DTLB 发送的 PTW 请求未命中 DTLB 的时序图

在 DTLB 接收到 L2 TLB 的 PTW 回复后，将页表项存储在 DTLB 中。当 Memblock 再次访问 DTLB 时会发生 hit，情况与图 18.17 相同。DTLB 与 L2 TLB 交互的时序情况与图 18.16 的 ptw_req 和 ptw_resp 部分相同。

18.2.4.3 TLB 与 tlbRepeater 的接口时序

18.2.4.3.1 TLB 向 tlbRepeater 发送 PTW 请求

TLB 向 tlbRepeater 发送 PTW 请求的接口时序图如图 18.19 所示。

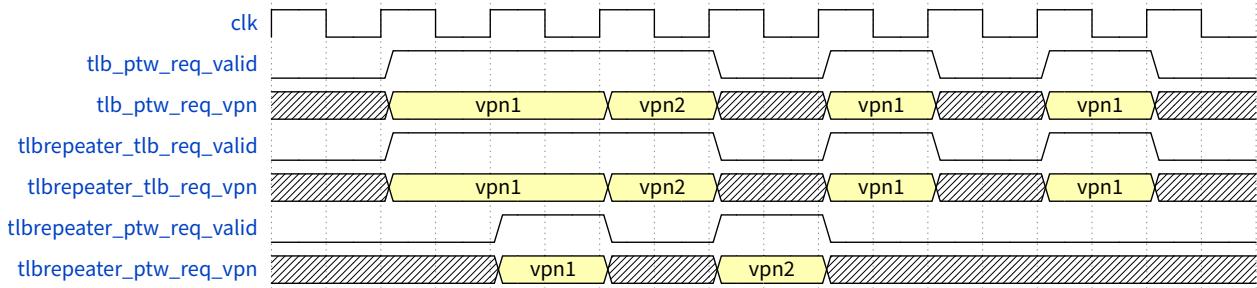


图 18.19: TLB 向 Repeater 发送 PTW 请求的时序图

昆明湖架构中，ITLB 和 DTLB 均采用非阻塞访问，在 TLB miss 时会向 L2 TLB 发送 PTW 请求，但并不会因为未接收到 PTW 回复而阻塞流水线和 TLB 与 Repeater 之间的 PTW 通道。TLB 可以不断向 tlbRepeater 发送 PTW 请求，tlbRepeater 会根据这些请求的虚拟页号，对重复的请求进行合并，避免 L2 TLB 的资源浪费以及 L1 TLB 的重复项。

从图 18.19 的时序关系可以看出，在 tlb 向 Repeater 发送 PTW 请求后的下一拍，Repeater 会继续向下传递 PTW 请求。由于 Repeater 已经向 L2 TLB 发送过虚拟页号为 vpn1 的 PTW 请求，因此当 Repeater 再次接收到相同虚拟页号的 PTW 请求时，不会再传递给 L2 TLB。

18.2.4.3.2 itlbRepeater 向 ITLB 返回 PTW 回复

itlbRepeater 向 ITLB 返回 PTW 回复的接口时序图参见图 18.20。

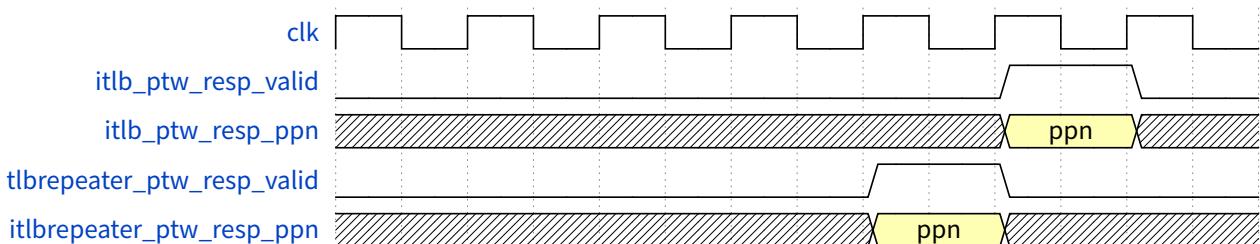


图 18.20: itlbRepeater 向 ITLB 返回 PTW 回复的时序图

时序描述如下：

- 第 X 拍: itlbRepeater 收到通过下级 itlbRepeater 传入的 L2 TLB 的 PTW 回复, itlbrepeater_ptw_resp_valid 为高。
- 第 X+1 拍: ITLB 收到来自 itlbRepeater 的 PTW 回复。

18.2.4.3.3 dtlbRepeater 向 DTLB 返回 PTW 回复

dtlbRepeater 向 DTLB 返回 PTW 回复的接口时序图参见图 18.21。

时序描述如下：

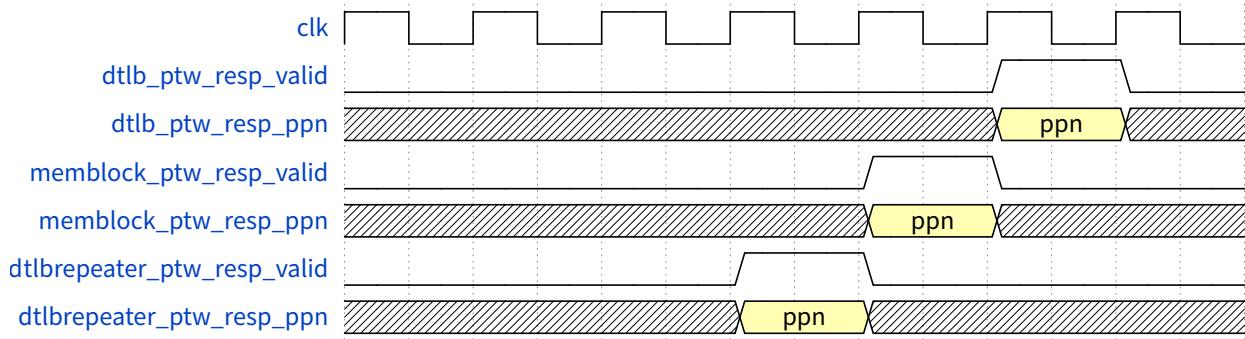


图 18.21: dtlbRepeater 向 DTLB 返回 PTW 回复的时序图

- 第 X 拍: dtlbRepeater 收到通过下级 dtlbRepeater 传入的 L2 TLB 的 PTW 回复, dtlbrepeater_ptw_resp_valid 为高。
- 第 X+1 拍: dtlbRepeater 将 PTW 回复传递到 memblock 中。
- 第 X+2 拍: DTLB 收到 PTW 回复。

18.3 二级模块 Repeater

Repeater 包括如下模块:

- PTWFilter itlbRepeater1
- PTWRepeaterNB itlbRepeater2
- PTWRepeaterNB itlbRepeater3
- PTWNewFilter dtlbRepeater

18.3.1 设计规格

1. 支持在 L1 TLB 和 L2 TLB 之间传递 PTW 请求与回复
2. 支持过滤重复的请求
3. 支持 TLB Hint 机制

18.3.2 功能

18.3.2.1 向 L2 TLB 传输 L1 TLB 的 PTW 请求

在 L1 TLB 和 L2 TLB 之间有比较长的物理距离, 会导致比较长的线延迟, 因此需要通过 Repeater 模块在中间加拍。由于 ITLB 和 DTLB 均支持多个 outstanding 的请求, 因此 repeater 会同时承担类似 MSHR 的功能, 并过滤重复请求。Filter 可以过滤掉重复的请求, 避免 L1 TLB 中出现重复项。Filter 的项数一定程度上决定了 L2 TLB 的并行度。(参见 5.1.1.2 节)

昆明湖架构中, L2 TLB 位于 memblock 模块中, 但与 ITLB 和 DTLB 均有一定距离。香山的 MMU 包含三个 itlbRepeater 和一个 dtlbRepeater, 起到在 L1 TLB 与 L2 TLB 之间加拍的效果, 两级 Repeater 之间通过 valid-ready 信号进行交互。ITLB 将 PTW 请求以及虚拟页号发送给 itlbRepeater1, 进行仲裁后发送给 itlbRepeater2, 并发送给 itlbRepeater3, 通过 itlbRepeater3 向 L2 TLB 传递 PTW 请求。L2 TLB 将 PTW 请求对应的虚拟页号, 查找 L2 TLB 得到的物理页号、页表的权限位、页表等级、是否发生异常等信号返回给 itlbRepeater3、itlbRepeater2, 通过 itlbRepeater1 最终返回给 ITLB。DTLB 与 dtlbRepeater 的交互和 ITLB

类似, dtlbRepeater 和 itlbRepeater1 是 Filter 模块, 可以合并 L1 TLB 中重复的请求。由于昆明湖架构中 ITLB 和 DTLB 均为非阻塞式访问, 因此这些 repeater 也均为阻塞式 Repeater。

18.3.2.2 过滤重复的请求

ITLB 和 DTLB 均包括多个通道, 不同通道之间、同一通道之间的多次缺失请求都可能重复。如果我们只采用普通 Arbiter, 每次只处理一个请求, 那么其他访问 L1 TLB 的请求就会重发, 继续得到 miss, 并发送给 L2 TLB。这样会使 L2 TLB 的利用率不高, 同时重发时也会占用处理器的资源。因此我们使用 Filter 模块, Filter 的本质是一个多进单出的队列, 可以起到重复请求过滤的作用。

需要注意, 在昆明湖架构中, dtlbRepeater 由 load entry、store entry、prefetch entry 三部分组成, 来自于 load dtlb、store dtlb、prefetch dtlb 的请求会分别发送至三种 entry 进行处理。三种 entry 会使用循环仲裁器进行仲裁, 将仲裁后的结果发送给 L2 TLB。另外, itlbRepeater 会对 ITLB 传入的所有请求进行检查, 过滤重复的请求; 但 dtlbRepeater 检查重复请求的粒度是 entry, 只检查同一个 dtlb (load dtlb、store dtlb、prefetch dtlb) 中的请求不会重复, 但不同 dtlb 之间 (例如 load dtlb 和 store dtlb) 发送给 L2 TLB 的请求依然可能重复。

18.3.2.3 支持 TLB Hint 机制

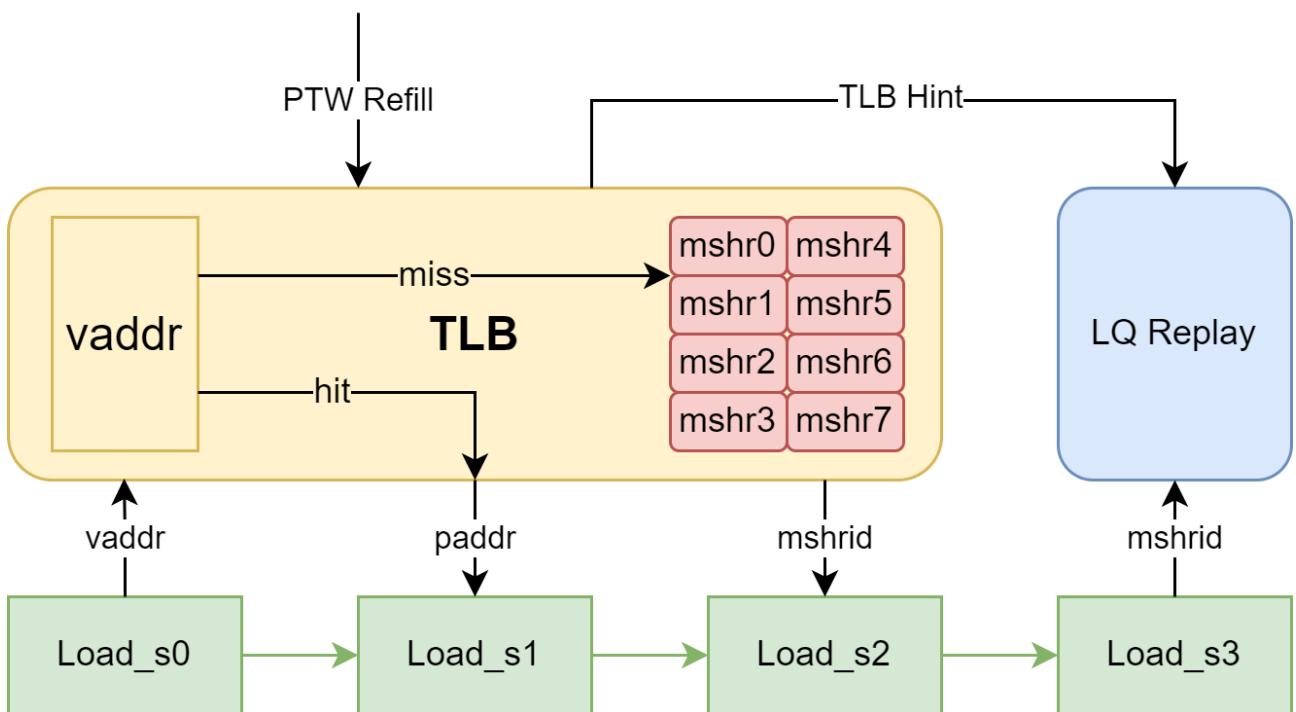


图 18.22: TLB Hint 示意图

当 TLB 命中时, 不会对一条 load 指令的生命周期产生影响 (第 0 拍 loadunit 查询 TLB, 第 1 拍 TLB 返回结果)。当 TLB 缺失后, 会继续查询 L2 TLB, 以及内存中的页表, 直至查询得到结果返回。但从一条 load 指令的生命周期出发, 这条 load 指令查询 TLB 缺失后会进入 load replay queue 进行等待。只有这条 load 指令被 load replay queue 重发, 且命中 TLB 查询得到物理地址后, 才能根据物理地址做后续操作。

因此, 一条 load 指令何时重发是缩短 load 执行时间的关键问题。如果 load 指令不能及时重发, 即使 TLB 回填周期缩短, 对访存的整体性能不会有提升。因此, 昆明湖架构实现了 TLB Hint 机制, 针对性唤醒因 TLB

miss 而需要重发的 load 指令。具体地，load_s0 阶段发送 vaddr 至 TLB，如果未命中，在 load_s1 阶段返回 miss 信息。同时，在 load_s1 阶段 TLB 会发送该条缺失信息至 dtlbrepeater，由 dtlbrepeater 进行处理。

Dtlbrepeater 处理会得到两种结果，返回 MSHRid 或 full 信号。在 dtlbrepeater 的 load entry 中，会首先检查新请求是否和现有项重复，如果和某个已有项重复，则将已有项的 MSHRid 返回。如果不和现有项重复，会检查是否有空余项，如果有空余项则返回 MSHRid，否则返回 full 信号。如果两个 load 通道同时发送给 dtlbrepeater 请求，且虚拟地址相同，则会以 loadunit(0) 的 MSHRid 为准。

在昆明湖架构中，所有因 TLB miss 而进入 load replay queue 的指令均只能等待唤醒重发，如果一条 load 指令在进入 load replay queue 后始终没有等待到唤醒信号，则会出现卡死情况。为了避免卡死，当 DTLB 向 dtlbrepeater 发送请求，dtlbrepeater 没有空余项接收时，需要返回 full 信号，表示 dtlbrepeater 已满，无法接收该条 load 指令对应的 PTW 请求，因此 load replay queue 不会接收到 Hint 信号，需要由 load replay queue 保证能够重发，而不会卡死。除这种情况外，当回填的项已经到达 dtlb 或 dtlbrepeater，但还未真正写入 dtlb 项时，也会给 loadunit 返回 full 信号，表示需要重发。

在 load_s2 阶段，dtlbrepeater 会返回 mshrid 信息至 loadunit，并在 load_s3 阶段写入 load replay queue。如果 MSHRid 有效，load replay queue 需要等待 PTW refill 信息命中 dtlbrepeater 中保存的 MSHRid，此时 dtlbrepeater 会像 load replay queue 发送唤醒 (Hint) 信息，表示该 MSHRid 已经重填，需要重发，此时可以命中 dtlb。另外，当某个 PTW refill 请求对应多个 MSHR entry 时（例如两个 vpn 在同一个 2M 空间内，PTW refill 的页表等级为 2MB 页），在这种情况下 dtlbrepeater 会向 load replay queue 发送 replay_all 信号，代表所有因 dtlb miss 而阻塞的 load 请求均需要被重发。由于这种情况很少，因此是一个几乎不损失性能的便捷方案。

18.3.3 整体框图

Repeater 的整体框图如图 18.23 所述，三个 itlbRepeater 和一个 dtlbRepeater，起到在 L1 TLB 与 L2 TLB 之间加拍的效果，两级 Repeater 之间通过 valid-ready 信号进行交互。Repeater 向上接受 ITLB 和 DTLB 的 PTW 请求，ITLB 和 DTLB 均为非阻塞式访问，因此这些 repeater 也均为阻塞式 Repeater。Repeater 向下将 L1 TLB 的 PTW 请求发送给 L2 TLB。dtlbRepeater 和 itlbRepeater1 是 Filter 模块，可以合并 L1 TLB 中重复的请求。

除 itlbRepeater1 之外，剩下两级 itlbRepeater 的本质只是单纯的加拍。加拍的多少要根据物理距离决定。在香山的昆明湖架构中，L2 TLB 位于 Memblock 中，和 ITLB 所在的 Frontend 模块物理距离较远，因此选择在 Frontend 中增加两级 repeater，Memblock 中增加一级 Repeater。而 DTLB 位于 Memblock 中，和 L2 TLB 之间的距离较近，只需要一级 Repeater 即可满足时序的要求。

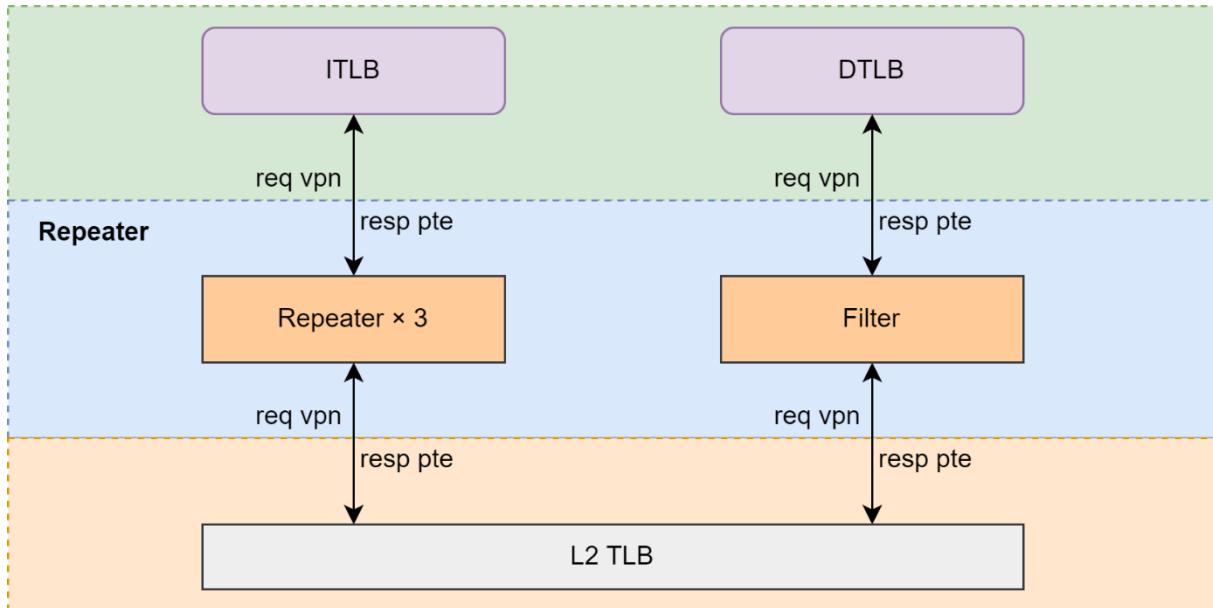


图 18.23: Repeater 模块整体框图

18.3.4 接口列表

参见接口列表文档。

18.3.5 接口时序

18.3.5.1 Repeater1 与 L1 TLB 的接口时序

参见 § 18.2.4.3 TLB 与 tlbRepeater 的接口时序。

18.3.5.2 itlbRepeater3 及 dtlbRepeater1 与 L2 TLB 的接口时序

itlbRepeater3 及 dtlbRepeater1 与 L2 TLB 的接口时序如图 18.24 所示。两者之间通过 valid-ready 信号进行握手，Repeater 将 L1 TLB 发出的 PTW 请求以及请求的虚拟地址发送给 L2 TLB；L2 TLB 查询得到结果后将物理地址以及对应的页表返回给 Repeater。

18.3.5.3 多级 itlbrepeater 之间的接口时序

多级 itlbrepeater 之间的接口时序如图 18.25 所示。两级 Repeater 之间通过 valid-ready 信号进行握手。

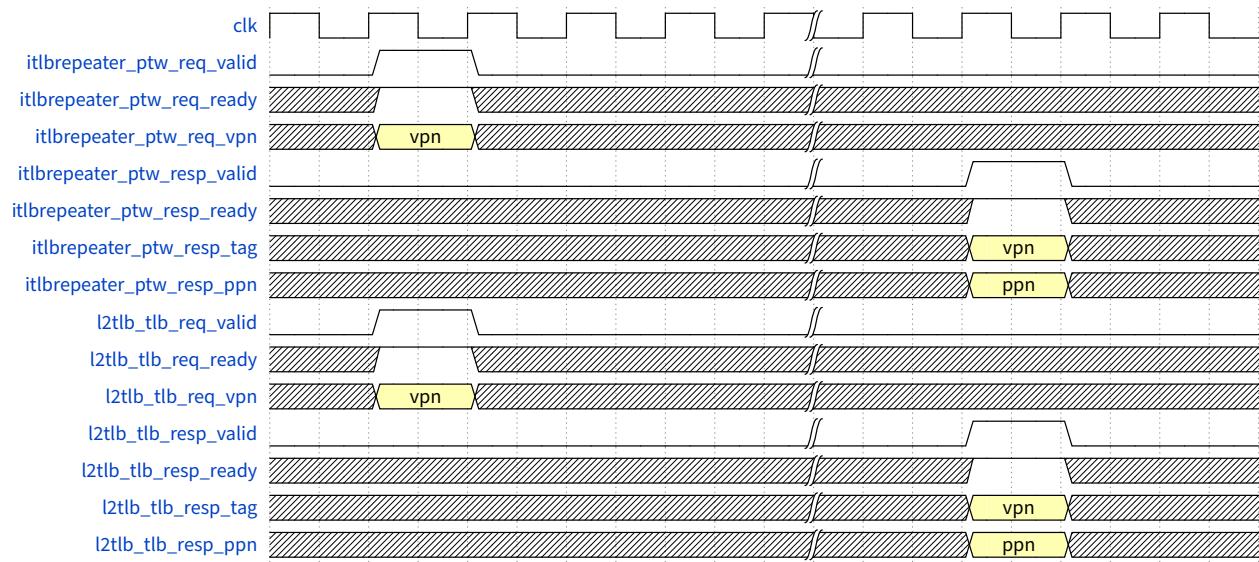


图 18.24: itlbRepeater3 及 dtlbRepeater1 与 L2 TLB 的接口时序



图 18.25: 多级 itlbrepeater 之间的接口时序

18.4 二级模块 L2TLB

18.4.1 二级模块 L2 TLB

L2 TLBWrapper 指的是如下模块：

- L2TLBWrapper ptw, L2TLBWrapper 为 L2 TLB 做了一层抽象。

L2 TLB 指的是：

- L2TLB ptw

18.4.1.1 设计规格

该小节描述了 L2 TLB 模块的整体设计规格，L2 TLB 模块包含子模块的设计规格参见本文档三级模块部分的内容。

1. 支持接收 L1 TLB 的 PTW 请求
2. 支持向 L1 TLB 返回 PTW 回复
3. 支持复制信号和寄存器
4. 支持异常处理机制
5. 支持 TLB 压缩
6. 支持两阶段地址翻译

18.4.1.2 功能

L2 TLB 是更大的页表缓存，由 ITLB 和 DTLB 共享，当 L1 TLB 发生 miss 时，会向 L2 TLB 发送 Page Table Walk 请求。L2 TLB 分为 Page Cache（参见 5.3.7 节），Page Table Walker（参见 5.3.8 节），Last Level Page Table Walker（参见 5.3.9 节）、Hypervisor Page Table Walker（参见 5.3.10）、Miss Queue（参见 5.3.11 节）和 Prefetcher（参见 5.3.12 节）五部分。

来自 L1 TLB 的请求将首先访问 Page Cache，对于非两阶段地址翻译的请求，若命中叶子节点则直接返回给 L1 TLB，否则根据 Page Cache 命中的页表等级以及 Page Table Walker 和 Last Level Page Table Walker 的空闲情况进入 Page Table Walker、Last Level Page Table Walker 或 Miss Queue（参见 5.3.7 节）。而对于两阶段地址翻译请求，如果该请求是 onlyStage1 的，则处理方式与非两阶段地址翻译请求一致；如果该请求是 onlyStage2 的，命中叶子页表，则直接返回，没有命中，则发送给 Page Table Walker 进行翻译；如果该请求是 allStage 的，由于 Page Cache 一次只能查询一次页表，所以首先查询第一阶段页表，分两种情况，如果第一阶段页表命中，则发送给 Page Table Walker，由其进行接下来的翻译过程，如果第一阶段页表没有命中叶子节点，则根据命中页表等级以及 Page Table Walker 和 Last Level Page Table Walker 的空闲情况进入 Page Table Walker、Last Level Page Table Walker 或 Miss Queue。为了加快页表访问，Page Cache 将三级页表都分开做了缓存，可以同时查询三级页表（参见 5.3.7 节）。Page Cache 支持 ecc 校验，如果 ecc 校验出错，则刷新此项，并重新进行 Page Walk。

Page Table Walker 接收 Page Cache 的请求，进行 Hardware Page Table Walk。对于非两阶段地址翻译的请求，Page Table Walker 只访问前两级（1GB 和 2MB）页表，不访问 4KB 页表，对 4KB 页表的访问会由 Last Level Page Table Walker 承担。如果 Page Table Walker 访问到叶子节点（大页），则返回给 L1 TLB，否则需要返回给 Last Level Page Table Walker，由 Last Level Page Table Walker 进行最后一级页表的访问。Page Table Walker 只能同时处理一个请求，无法并行访问前两级页表。对于两阶段地址翻译的请求，第一种情

况，如果是 allStage 的请求并且第一阶段翻译的页表 hit，PTW 会发送第二阶段请求进入 Page Cache 查询，如果没有命中，则会发送给 Hypervisor Page Table Walker，第二阶段翻译结果会返回给 PTW；第二种情况，如果是 allStage 请求并且第一阶段翻译的叶子节点没有命中，则 PTW 翻译过程与非虚拟化请求翻译类似，区别在于 PTW 翻译过程中出现的物理地址为客户机物理地址，需要进行一次第二阶段地址翻译后才能访存，具体可见 Page Table Walker 的模块介绍；第三种情况，如果是 onlyStage2 请求，则 PTW 会向外发送第二阶段翻译的请求，接收到 resp 后，返回给 L1TLB；第四种请求是 onlyStage1 的请求，该请求在 PTW 内部的处理过程与非虚拟化请求的处理过程一致。

Miss Queue 接收来自 Page Cache 和 Last Level Page Table Walker 的请求，等待下次访问 Page Cache。Prefetcher 采用 Next-Line 预取算法，当 miss 或者 hit 但命中项为预取项时，产生下一个预取请求。

18.4.1.2.1 接收 L1 TLB 的请求并返回回复

L2 TLB 作为整体，接收 L1 TLB 的 PTW 请求。L1 TLB 发送的 PTW 请求通过两级 Repeater 传入 L2 TLB 中，L2 TLB 根据请求来自 itlbRepeater 或 dtlbRepeater，分别向 itlbRepeater 或 dtlbRepeater 返回回复。L2 TLB 接收 L1 TLB 发送的虚拟页号，返回给 L1 TLB 的信息包括：第一阶段的页表、第二阶段的页表等。L2 TLB 的行为对 L1 TLB 透明，L1 TLB 与 L2 TLB 之间只需要通过部分信号接口交互。

18.4.1.2.2 向 L2 Cache 发送 PTW 请求

L2 TLB 通过 TileLink 总线向 L2 Cache 发送 PTW 请求，和 L2 Cache 之间通过 ptw_to_l2_buffer 相连，向 ptw_to_l2_buffer 提供和接收 TileLink A 通道和 D 通道的相关信号。

18.4.1.2.3 复制信号和寄存器

由于 L2 TLB 模块较大，例如 sfence 信号、csr 寄存器等需要驱动较多部分，因此需要将 sfence 信号、csr 寄存器复制多份。复制寄存器可以便于时序优化和物理实现，与功能实现无关，复制的内容完全相同。可以使用复制的信号和寄存器驱动不同位置的部件。

复制情况以及分别驱动的部分如表 18.18 所示：

表 18.18: 信号复制情况以及驱动部件

| 复制信号 | 序号 | 驱动部件 |
|--------|---------------|------------------------------|
| sfence | | |
| | sfence_dup(0) | Prefetch |
| | sfence_dup(1) | Last Level Page Table Walker |
| | sfence_dup(2) | cache(0) |
| | sfence_dup(3) | cache(1) |
| | sfence_dup(4) | cache(2) |
| | sfence_dup(5) | cache(3) |
| | sfence_dup(6) | Miss Queue |
| | sfence_dup(7) | Page Table Walker |
| | sfence_dup(8) | Hypervisor Page Table Walker |
| csr | | |
| | csr_dup(0) | Prefetch |

| 复制信号 | 序号 | 驱动部件 |
|------|------------|------------------------------|
| | csr_dup(1) | Last Level Page Table Walker |
| | csr_dup(2) | cache(0) |
| | csr_dup(3) | cache(1) |
| | csr_dup(4) | cache(2) |
| | csr_dup(5) | Miss Queue |
| | csr_dup(6) | Page Table Walker |
| | csr_dup(7) | Hypervisor Page Table Walker |

18.4.1.2.4 异常处理机制

L2 TLB 可能产生的异常包括：guest page fault、page fault、access fault、ecc 校验出错。对于 guest page fault、page fault 和 access fault，会交付给 L1 TLB，L1 TLB 根据请求来源交付处理；对于 ecc 校验出错，会在 L2 TLB 内部处理，将当前项无效掉，返回 miss 结果并重新进行 Page Walk。参见本文档的第 6 部分：异常处理机制。

18.4.1.2.5 TLB 压缩

在添加虚拟化拓展的内容后，L2TLB 中 stage1 复用了 TLB 压缩的逻辑设计，返回给 L1TLB 的结构也是 TLB 压缩的结构，但在 L1TLB 中 TLB 压缩并不启用，而 stage2 则不采用 TLB 压缩结构，resp 的时候只返回单个页表。

L2 TLB 每次访问内存的宽度为 512 bits，每次可以返回 8 项页表。Page Cache 的 l3 项由 SRAM 组成，在回填时可以保存连续的 8 个页表项；Page Cache 的 sp 项由寄存器堆组成，回填时只保存单个页表项。因此，在 Page Cache 命中，返回 L1 TLB 时（非两阶段地址翻译的情况，如果是两阶段地址翻译，第一阶段命中会发送给 PTW 进行后续处理），如果命中的是 4KB 页表，可以将 Page Cache 中存储的连续 8 项页表进行压缩；如果命中的是大页，则不进行压缩，直接回填入 L1 TLB。（事实上，1GB 或 2MB 的大页缺失情况很少，因此只考虑为 4KB 页面进行压缩。而 4KB 页面的 ppn 为 24 位，连续 8 项压缩后，需要满足这些页表的高 21 位 ppn 相同才能进行压缩。对于 1GB 或 2MB 的大页，ppn 的低 9 位并不会用于生成物理地址，因此在当前的设计下无意义）

在 Page Cache 缺失，通过 Page Table Walker 或 Last Level Page Table Walker 访问内存中的页表后，会将内存中的页表返回给 L1 TLB，同时回填入 Page Cache 中，如果需要第二阶段的地址翻译，在 Hypervisor Page Table Walker 中访问到的页表也会重填到 Page Cache，HPTW 会将最终的翻译结果返回给 PTW 或者 LLPTW，PTW 或 LLPTW 会把两阶段的页表都返回给 L1TLB。非两阶段翻译请求在返回至 L1 TLB 时，同样可以将连续的 8 项页表进行压缩。由于 Page Table Walker 只有在访问到叶子节点时才会直接返回给 L1 TLB，因此 Page Table Walker 返回给 L1 TLB 的页表均为大页。由于大页对性能的影响很小，考虑到优化方案的实现简单性，以及复用 Page Cache 中 sp 项的数据通路，Page Table Walker 返回的大页并不会压缩。

L2 TLB 只对 4KB 页表进行压缩，根据 RISC-V 特权级手册中的 Sv39 分页机制，4KB 页表所在物理地址的低 3 位即为虚拟页号的低 3 位。因此，Page Cache 或 Last Level Page Table Walker 返回的连续 8 项页表可以通过虚拟页号的低 3 位进行索引。其中，valid 位表示压缩后的页表项是否有效。根据 L1 TLB 发送的页表查找请求虚拟页号的低三位，索引到该虚拟页号对应的页表项，该页表项的 valid 一定为 1。对于和该页表项连续的其余 7 个页表项，通过比较它们的物理页号高位，以及页表属性位是否相同。如果物理页号高位以及页表属性位都和通过虚拟页号的低三位索引到的页表项相等，则 valid 位为 1，否则为 0。同时，L2 TLB 还会返回 pteidx，即表示这 8 项连续的页表中，哪一项才是 L1 TLB 发送 vpn 对应的页表。L2 TLB 压缩如图 18.26, 18.27 所示。

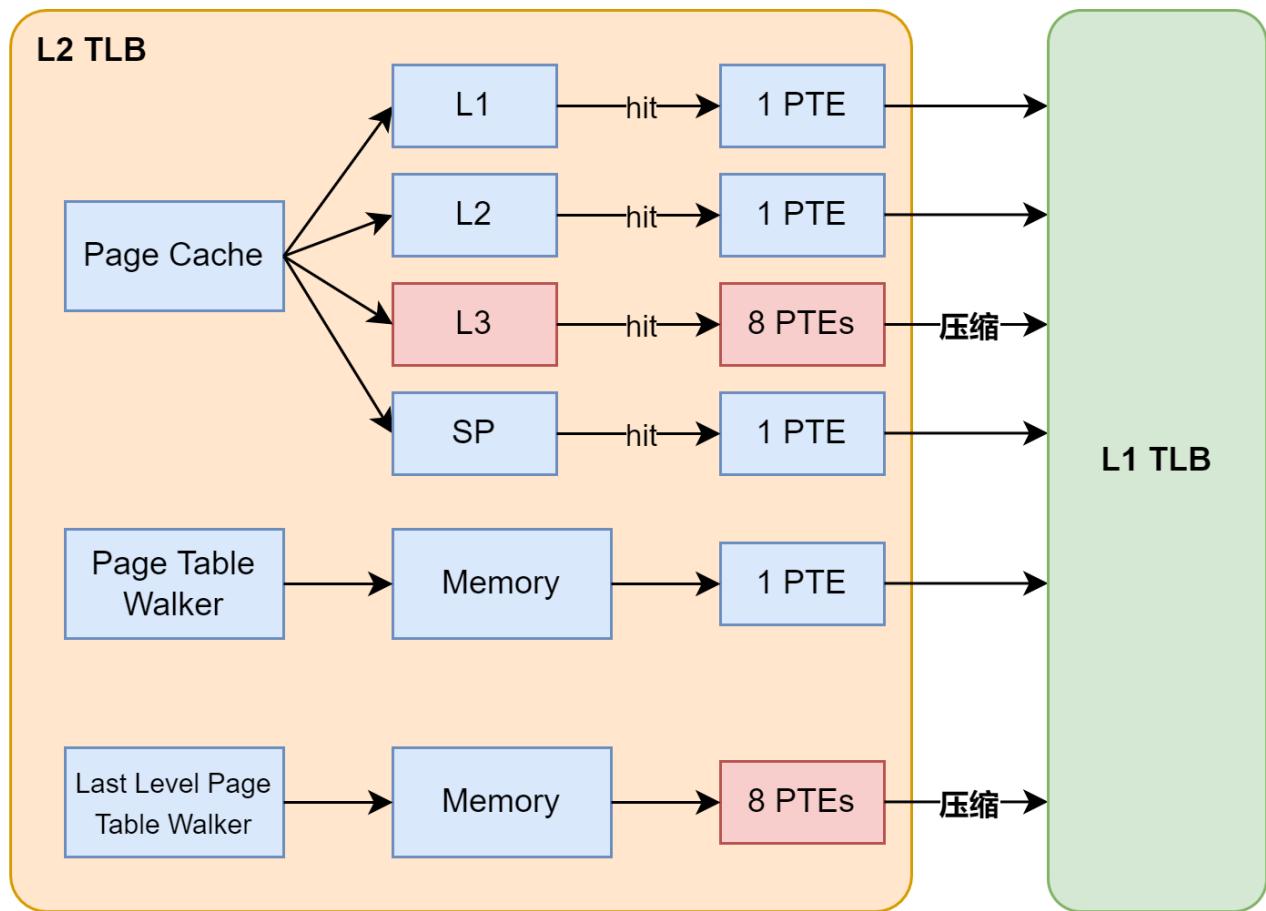


图 18.26: L2 TLB 压缩示意图 1

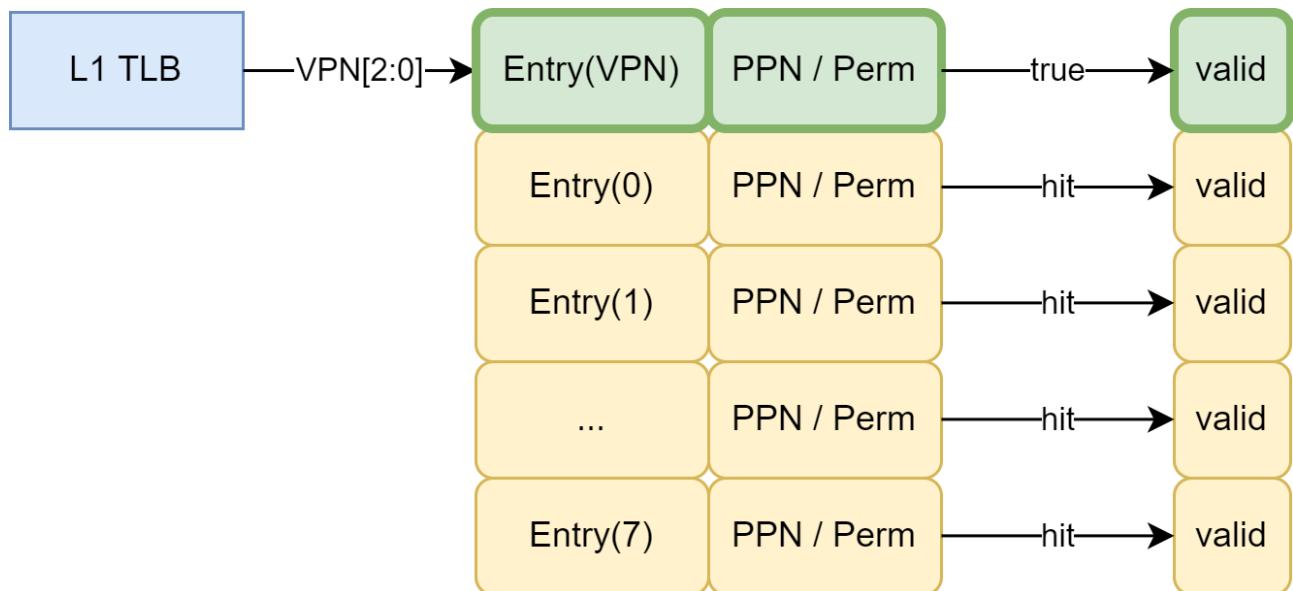


图 18.27: L2 TLB 压缩示意图 2

实现 TLB 压缩后，L1 TLB 的每项都是一个压缩的 TLB 项，通过虚拟页号的高位查找。TLB 项命中的条件除虚拟页号高位相同外，还需要满足虚拟页号低位对应的 valid 位为 1，表示查找的页表项在压缩后的 TLB 项中有效。TLB 压缩和 L1 TLB 相关的部分详见 L1TLB 模块的介绍。

18.4.1.3 整体框图

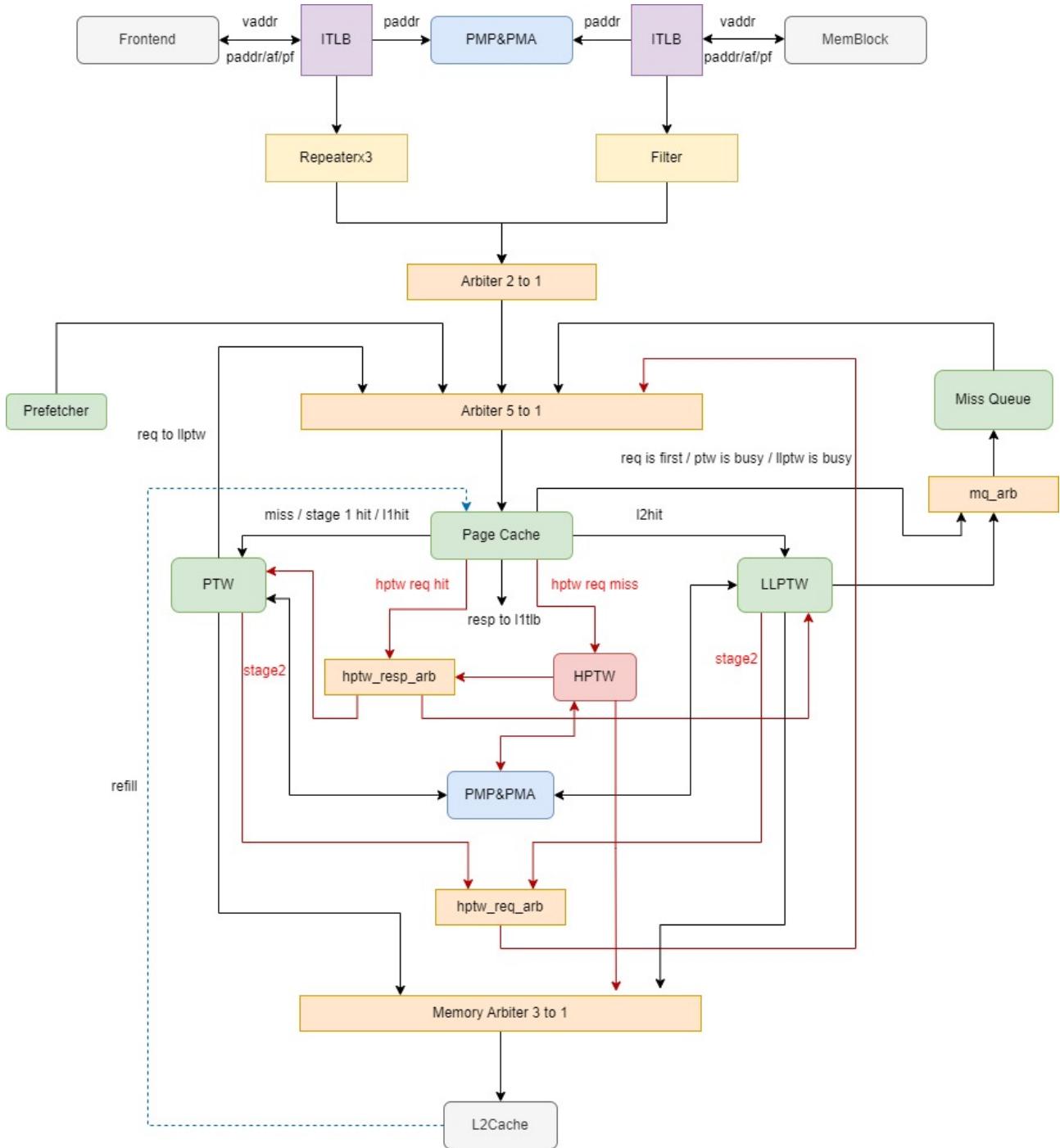


图 18.28: L2 TLB 模块整体框图

如图 18.28 所示，L2 TLB 分为 Page Cache、Page Table Walker、Last Level Page Table Walker、Hypervisor Page Table Walker、Miss Queue 和 Prefetcher 六部分。

来自 L1 TLB 的请求将首先访问 Page Cache，对于非两阶段地址翻译的请求，若命中叶子节点则直接返回

给 L1 TLB，否则根据 Page Cache 命中的页表等级以及 Page Table Walker 和 Last Level Page Table Walker 的空闲情况进入 Page Table Walker、Last Level Page Table Walker 或 Miss Queue（参见 5.3.7 节）。而对于两阶段地址翻译请求，如果该请求是 onlyStage1 的，则处理方式与非两阶段地址翻译请求一致；如果该请求是 onlyStage2 的，命中叶子页表，则直接返回，没有命中，则发送给 Page Table Walker 进行翻译；如果该请求是 allStage 的，由于 Page Cache 一次只能查询一次页表，所以首先查询第一阶段页表，分两种情况，如果第一阶段页表命中，则发送给 Page Table Walker，由其进行接下来的翻译过程，如果第一阶段页表没有命中叶子节点，则根据命中页表等级以及 Page Table Walker 和 Last Level Page Table Walker 的空闲情况进入 Page Table Walker、Last Level Page Table Walker 或 Miss Queue。为了加快页表访问，Page Cache 将三级页表都分开做了缓存，可以同时查询三级页表（参见 5.3.7 节）。Page Cache 支持 ecc 校验，如果 ecc 校验出错，则刷新此项，并重新进行 Page Walk。

Page Table Walker 接收 Page Cache 的请求，进行 Hardware Page Table Walk。对于非两阶段地址翻译的请求，Page Table Walker 只访问前两级（1GB 和 2MB）页表，不访问 4KB 页表，对 4KB 页表的访问会由 Last Level Page Table Walker 承担。如果 Page Table Walker 访问到叶子节点（大页），则返回给 L1 TLB，否则需要返回给 Last Level Page Table Walker，由 Last Level Page Table Walker 进行最后一级页表的访问。Page Table Walker 只能同时处理一个请求，无法并行访问前两级页表。对于两阶段地址翻译的请求，第一种情况，如果是 allStage 的请求并且第一阶段翻译的页表 hit，PTW 会发送第二阶段请求进入 Page Cache 查询，如果没有命中，则会发送给 Hypervisor Page Table Walker，第二阶段翻译结果会返回给 PTW；第二种情况，如果是 allStage 请求并且第一阶段翻译的叶子节点没有命中，则 PTW 翻译过程与非虚拟化请求翻译类似，区别在于 PTW 翻译过程中出现的物理地址为客户机物理地址，需要进行一次第二阶段地址翻译后才能访存，具体可见 Page Table Walker 的模块介绍；第三种情况，如果是 onlyStage2 请求，则 PTW 会向外发送第二阶段翻译的请求，接收到 resp 后，返回给 L1TLB；第四种请求是 onlyStage1 的请求，该请求在 PTW 内部的处理过程与非虚拟化请求的处理过程一致。

Miss Queue 接收来自 Page Cache 和 Last Level Page Table Walker 的请求，等待下次访问 Page Cache。Prefetcher 采用 Next-Line 预取算法，当 miss 或者 hit 但命中项为预取项时，产生下一个预取请求。

图中涉及到如下的仲裁器，名称为 chisel 代码中的名称：

- arb1:2 to 1 的仲裁器，图中的 Arbiter 2 to 1，输入为 ITLB (itlbRepeater2) 和 DTLB (dtlbRepeater2)；输出到 Arbiter 5 to 1
- arb2:5 to 1 的仲裁器，图中的 Arbiter 5 to 1，输入为 Miss Queue、Page Table Walker、arb1、hptw_req_arb 和 Prefetcher；输出到 Page Cache
- hptw_req_arb: 2 to 1 的仲裁器，输入为 Page Table Walker 和 Last Level Page Table Walker，输出到 Page Cache
- hptw_resp_arb: 2 to 1 的仲裁器，输入为 Page Cache 和 Hypervisor Page Table Walker，输出到 PTW 或者 LLPTW
- outArb: 1 to 1 的仲裁器，输入为 mergeArb，输出到 L1TLB 的 resp
- mergeArb: 3 to 1 的仲裁器，输入为 Page Cache、Page Table Walker 和 Last Level Page Table Walker，输出到 outArb
- mq_arb: 2 to 1 的仲裁器，输入为 Page Cache 和 Last Level Page Table Walker；输出为 Miss Queue
- mem_arb: 3 to 1 的仲裁器，输入为 Page Table Walker、Last Level Page Table Walker 和 Last Level Page Table Walker；输出为 L2 Cache（Last Level Page Table Walker 内部也有一个 mem_arb，把所有 Last Level Page Table Walker 向 L2 Cache 发送的 PTW 项做仲裁，之后传到这个 mem_arb 里）

L2 TLB 模块的 hit 通路如图 18.29 所示。来自 ITLB 和 DTLB 的请求会首先通过仲裁，之后送往 Page Cache 进行查询。对于非两阶段地址翻译、只有第二阶段翻译或者只有第一阶段翻译的请求，如果 Page Cache 命中，则

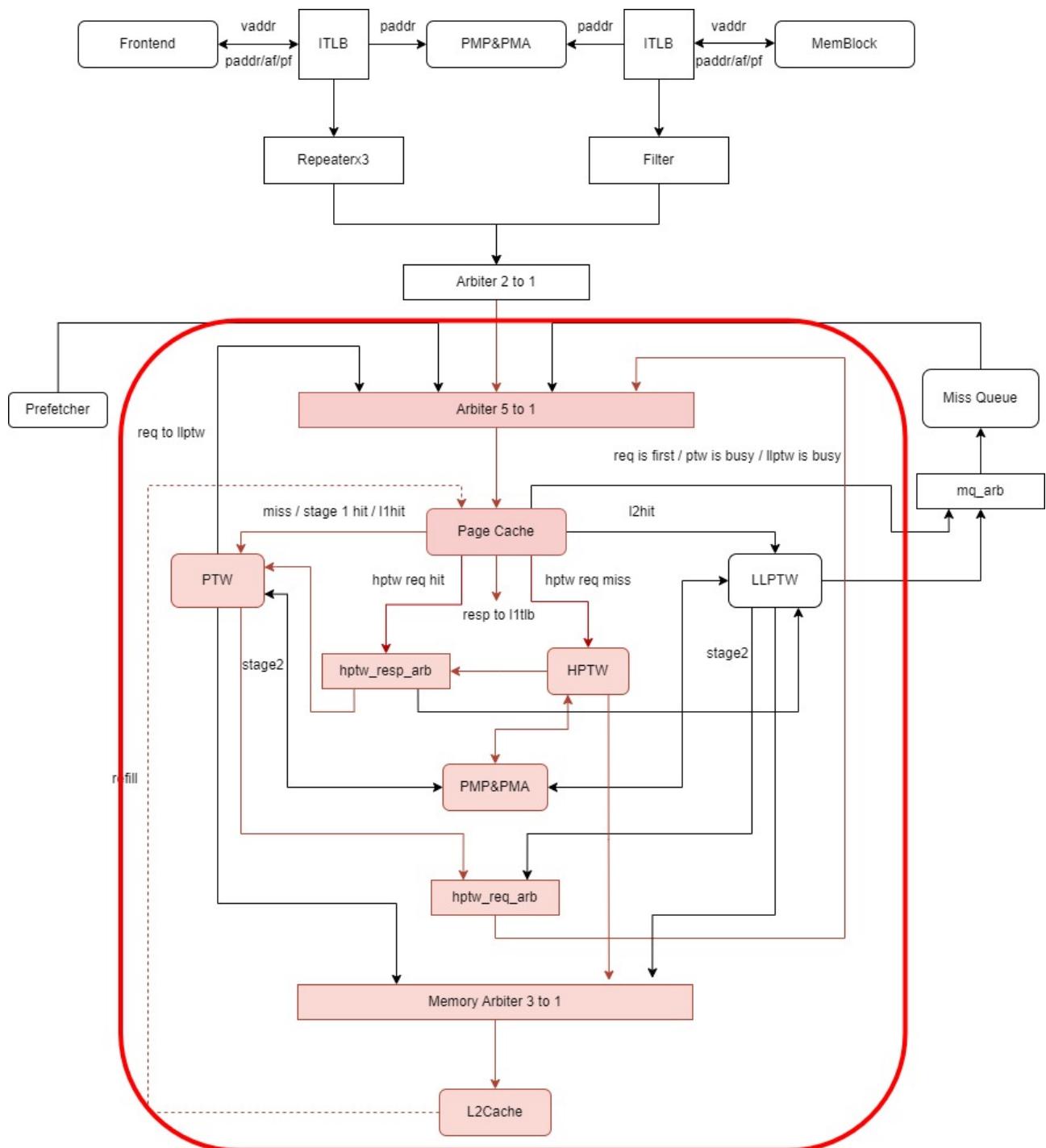


图 18.29: L2 TLB 模块 hit 通路

直接将 Page Cache 命中的页表项以及物理地址等信息返回给 L1 TLB。对于 allStage, Page Cache 中首先查询第一阶段页表, 第一阶段 hit, 则会发送给 PTW, 由 PTW 发送 hptw 请求, hptw 请求会进入 Page Cache 查询, 如果 hit 则发送给 PTW, 如果没有 hit, 则发送给 HPTW, HPTW 查询结束后将结果发送给 PTW, HPTW 访存得到的页表也会回填到 Page Cache。对于 ITLB 和 DTLB 发送的所有 PTW 请求以及 PTW 或者 LLPTW 发送的 hptw 请求, 一定会首先进行 Page Cache 的查询。

而对于 miss 的情况, 所有模块均可能参与。来自 ITLB 和 DTLB 的请求会首先通过仲裁, 之后送往 Page Cache 进行查询。如果 Page Cache 未命中, 则可能分情况进入 MissQueue 中 (对于 PTW 或者 LLPTW 发送给 Page Cache 的 hptw 请求或者是 prefetch 的请求, 则不进入 MissQueue), miss 的请求进入 MissQueue 的情况有出现 bypass 的请求, L1TLB 发送给 PageCache (isFirst) 且要进入 PTW 的请求, MissQueue 发送给 PTW 且 PTW 忙的请求, 发送给 LLPTW 且 LLPTW 忙的请求。Page Cache 需要根据 Page Cache 命中的页表等级决定进入 Page Table Walker 或 Last Level Page Table Walker 进行查询 (如果是 hptw 请求, 则会发送给 HPTW)。Page Table Walker 同时只能处理一个请求, 可以访问内存中前两级页表的内容; Last Level Page Table Walker 负责访问最后一级 4KB 页表, Hypervisor Page Table Walker 同时只能处理一个请求。

Page Table Walker、Last Level Page Table Walker 和 Hypervisor Page Table Walker 均可以向内存发送请求, 访问内存中的页表内容。在通过物理地址访问内存中的页表内容之前, 需要通过 PMP 和 PMA 模块对物理地址进行检查, 如果 PMP 和 PMA 检查失败则不会向内存发送请求。来自 Page Table Walker、Last Level Page Table Walker 和 Hypervisor Page Table Walker 的请求在经过仲裁后, 通过 TileLink 总线向 L2 Cache 发送请求。

Page Table Walker 与 Last Level Page Table Walker 均可能向 L1 TLB 返回 PTW 回复。Page Table Walker 可能在以下几种情况下产生回复:

- 非两阶段翻译请求和只有第一阶段翻译的请求, 访问到叶子节点 (是 1GB 或 2MB 的大页), 直接返回给 L1 TLB
- 只有第二阶段翻译的请求, 收到第二阶段翻译结果后
- 两阶段翻译都有的请求, 第一阶段的叶子页表与第二阶段的叶子页表都得到后
- 第二阶段翻译返回的结果出现 Page fault 或 Access fault
- PMP 或 PMA 检查出现 Page fault 或 Access fault, 同样需要返回给 L1 TLB

Last Level Page Table Walker 一定会向 L1 TLB 返回回复, 包括如下几种可能:

- 非两阶段翻译请求和只有第一阶段翻译的请求, 访问到叶子节点 (4KB 页)
- 两阶段翻译都有的请求, 第一阶段的叶子页表与第二阶段的叶子页表都得到后
- PMP 或 PMA 检查出现 Access fault

18.4.1.4 接口时序

18.4.1.4.1 L2 TLB 与 Repeater 的接口时序

L2 TLB 与 Repeater 的接口时序如图 18.30 所示。L2 TLB 与 Repeater 之间通过 valid-ready 信号进行握手, Repeater 将 L1 TLB 发出的 PTW 请求以及请求的虚拟地址发送给 L2 TLB; L2 TLB 查询得到结果后将物理地址以及对应的页表返回给 Repeater。

18.4.1.4.2 L2 TLB 与 L2 Cache 的接口时序

L2 TLB 与 L2 Cache 的接口时序遵循 TileLink 总线协议。

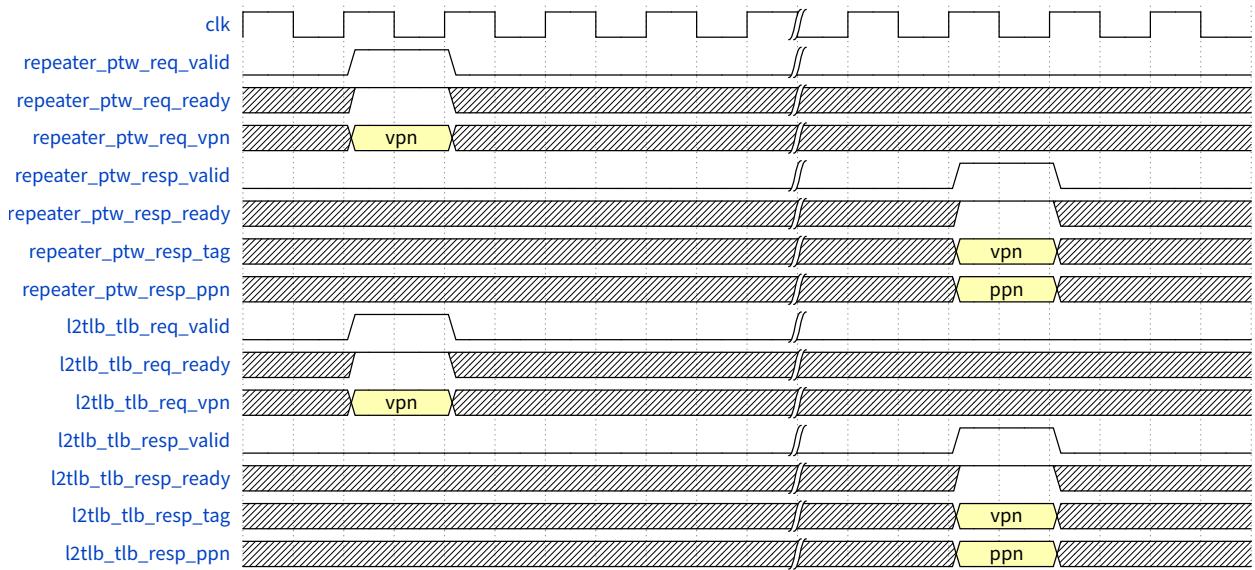


图 18.30: L2TLB 与 Repeater 的接口时序

18.4.2 三级模块 Page Cache

Page Cache 指的是如下模块: * PtwCache cache

18.4.2.1 设计规格

1. 支持分开缓存三级页表
2. 支持接收来自 L1 TLB 的 PTW 请求
3. 支持接收来自 Miss Queue 的 PTW 请求
4. 支持将命中结果返回给 L1 TLB, 返回 PTW 回复
5. 支持将未命中结果返回给 L2 TLB, 转发 PTW 请求
6. 支持 Page Cache 的充填
7. 支持 ecc 校验
8. 支持 sfence 刷新
9. 支持异常处理机制
10. 支持 TLB 压缩
11. 支持各级页表分为三种类型
12. 支持接收第二阶段翻译请求 (hptw 请求)
13. 支持 hfence 刷新

18.4.2.2 功能

18.4.2.2.1 分开缓存三级页表

Page Cache 是“增大版”的 L1 TLB, 也是名副其实的 L2 TLB。在 Page Cache 中分开缓存了三级页表, 可以一拍完成三级信息的查询 (H 拓展每级页表又分为 VS 阶段页表、G 阶段页表和主机页表, 在之后的章节介绍)。Page Cache 根据请求的地址判断是否命中, 获得最靠近叶子节点的结果。由于访存宽度为 512bits, 即 8 项页表, 因此 Page Cache 一项含有 8 个页表。(1 个虚拟页号, 对应 8 个物理页号、8 个权限位)

在 Page Cache 中根据页表的等级分别缓存, 分为 l1, l2, l3, sp 四项。l1、l2、l3 项中只存储有效的页表

项，分别存储一级、二级、三级页表。l1 包含 16 项，为全相联结构；l2 包含 64 项 2 路组相联，l3 包含 512 项 4 路组相联。sp 为 16 项全相联结构，存储大页（是叶子节点的一级、二级页表），以及无效（页表中 V 位为 0，或页表中 W 位为 1、R 位为 0，或页表非对齐）的一级、二级页表。在存储时，l1 和 l2 项不需要存储权限位，l3 和 sp 项需要存储权限位。

Page Cache 的项配置如表 18.19。

表 18.19: Page Cache 的项配置

| 项 | 项数 | 组织结构 | 实现方式 | 替换算法 | 存储内容 |
|----|-----|--------|------|------|--------------------------------------|
| l1 | 16 | 全相联 | 寄存器堆 | PLRU | 有效的一级（1GB 大小）页表，不需要存储权限位 |
| l2 | 64 | 2 路组相联 | SRAM | PLRU | 有效的二级（2MB 大小）页表，不需要存储权限位 |
| l3 | 512 | 4 路组相联 | SRAM | PLRU | 有效的三级（4KB 大小）页表，需要存储权限位 |
| sp | 16 | 全相联 | 寄存器堆 | PLRU | 大页（是叶子节点的一级、二级页表）、无效的一级、二级页表，需要存储权限位 |

Page Cache 项需要存储的信息包括：tag、asid、ppn、perm（可选）、level（可选）、prefetch，H 拓展新增了 vmid 和 h（用来区别三种类型页表）。l1 项和 sp 项由于采用全相联结构，因此 tag 的位数分别为 vpnnlen（9）和 $2 * vpnnlen$ （18），由于第二阶段翻译的地址比第一阶段多两位，所以 tag 需要再加两位。l2、l3 采用组相联结构，需要考虑 set 数，以及每个虚拟页号可以索引 8 项页表。l2 是 2 路组相联，因此 tag 位为 $2 * vpnnlen(18) - \log_2(64) - \log_2(8) + \log_2(2) = 10$ 位；l3 是 4 路组相联，因此 tag 位为 $3 * vpnnlen(27) - \log_2(512) - \log_2(8) + \log_2(4) = 17$ 位。对于 l3 和 sp 项，由于存储叶子节点，因此需要存储 perm 项，而 l1 和 l2 项不需要，perm 项存储 riscv 手册规定的 D、A、G、U、X、W、R 位，无需存储 V 位。对于 sp 项，需要存储 level，表示页表的等级（一级还是二级）。prefetch 表示该页表项是由预取的请求得到的，vmid 只在 VS 阶段页表和 G 阶段页表使用，asid 在 G 阶段页表不使用，h 是个两比特的寄存器，区别这三种页表，编码与 s2xlate 一致。Page Cache 项需要存储的信息如表 18.20 所示，页表的属性位如表 18.21 所示：

表 18.20: Page Cache 项需要存储的信息

| 项 | tag | asid | vmid | ppn | perm | level | prefetch | h |
|----|-----------------|------|------|-----|------|-------|----------|-----|
| l1 | Yes, 9 位 + 2 位 | Yes | Yes | Yes | NO | NO | Yes | Yes |
| l2 | Yes, 10 位 + 2 位 | Yes | Yes | Yes | NO | NO | Yes | Yes |
| l3 | Yes, 17 位 + 2 位 | Yes | Yes | Yes | Yes | NO | Yes | Yes |
| sp | Yes, 18 位 + 2 位 | Yes | Yes | Yes | Yes | Yes | Yes | Yes |

表 18.21: 页表项的属性位

| 位 | 域 | 描述 |
|---|---|------------------------------|
| 7 | D | Dirty，表示自从上次清除 D 位之后，该虚拟页面被读 |

| 位 | 域 | 描述 |
|---|---|--|
| 6 | A | Accessed, 表示自从上次清除 A 位之后, 该虚拟页面被读、写或取指 |
| 5 | G | 表示该页是否为全局映射, 该位为 1 表示该页是一个全局映射, 也就 是存在于所有地址空间中的映射 |
| 4 | U | 表示该页是否可以被 User Mode 访问, 该位为 0 表示不可被 User Mode 访问; 为 1 表示可以被访问 |
| 3 | X | 表示该页是否可执行, 该位为 0 表示不可执行; 该位为 1 表示页可执 行 |
| 2 | W | 表示该页是否可写, 该位为 0 表示不可写; 该位为 1 表示页可写 |
| 1 | R | 表示该页是否可读, 该位为 0 表示不可读; 该位为 1 表示页可读 |
| 0 | V | 表示该页表项是否有效, 如果该位为 0, 则表示该页表项无效, 页表 项的其他位可以由软件自由使用 |

表 18.22: h 编码说明

| h | 描述 |
|----|----------------------|
| 00 | noS2xlate, 主机页表 |
| 01 | onlyStage1, VS 阶段的页表 |
| 10 | onlyStage2, G 阶段的页表 |

手册中允许通过软件和硬件两种方式更新 A/D 位, 香山选择软件方式, 即当发现如下两种情况时通过报 page fault, 通过软件更新页表。

1. 访问某页, 但该页页表的 A 位是 0
2. 写入某页, 但该页页表的 D 位是 0

页表项中 X、W、R 位可能的组合及含义如表 18.23 所示:

表 18.23: 页表项中 X、W、R 位可能的组合及含义

| X | W | R | 描述 |
|---|---|---|----------------------------------|
| 0 | 0 | 0 | 表示该页表项并非叶子节点, 需要通过该页表索引下一 级页表 |
| 0 | 0 | 1 | 表示该页是只可读的 |
| 0 | 1 | 0 | 保留 |
| 0 | 1 | 1 | 表示该页是可读、可写的 |
| 1 | 0 | 0 | 表示该页是只可执行的 |
| 1 | 0 | 1 | 表示该页是可读、可执行的 |
| 1 | 1 | 0 | 保留 |
| 1 | 1 | 1 | 表示该页是可读、可写、可执行的 |

18.4.2.2.2 接收 PTW 请求，并返回结果

Page Cache 接收来自 L2 TLB 的 PTW 请求，L2 TLB 的 PTW 请求通过仲裁器进行仲裁，最终送给 Page Cache。这些 PTW 请求可能来源于 Miss Queue、L1 TLB、hptw_req_arb 或 Prefetcher。由于 Page Cache 每次查询只能处理一个请求，所以对于 allStage 请求首先查询第一阶段，对于 allStage 请求，查询各项 h 时，只查询 onlyStage1 的页表，第二阶段的翻译等该请求发送到 PTW 或者 LLPTW 后由 PTW 或者 LLPTW 进行处理。Page Cache 的查询过程如下所示：

- 第 0 拍：对 l1、l2、l3、sp 四项发出读请求，进行同时查询
- 第 1 拍：得到寄存器堆 (l1、sp 项) 以及 SRAM (对于 l2、l3 项) 读出的结果，但由于时序原因并不在当拍直接使用，等待下一拍再进行后续操作
- 第 2 拍：比对 Page Cache 中各项存储的 tag 和传入请求的 tag，比对各项 h 寄存器与传入的 s2xlate (allStage 转换成查询 onlyStage1)，在 l1、l2、l3、sp 项中同时进行是否匹配的查询，另外还需要进行 ecc 检查
- 第 3 拍：将 l1、l2、l3、sp 四项匹配得到的结果，以及 ecc 检查的结果做汇总

经过上述对 Page Cache 的查询过程如果在 Page Cache 中寻找到叶子节点，则返回给 L1 TLB(对于 allStage 的请求，如果第一阶段 hit，则发送给 PTW 处理)；否则根据不同情况向 LLPTW、PTW、HPTW 或 Miss Queue 转发请求。

18.4.2.2.3 向 L2 TLB 发送 PTW 请求

Page Cache 会根据不同情况向 LLPTW、PTW、HPTW 或 Miss Queue 转发请求。

1. 对于 noS2xlate、onlyStage1 和 allStage，如果 Page Cache 未命中叶子节点，但二级页表命中（对于 onlyStage1 和 allStage 是第一阶段的二级页表命中），并且该 PTW 请求并不是一个 bypass 请求，此时 Page Cache 将请求转发给 llptw。
2. 对于 noS2xlate、onlyStage1 和 allStage，如果 Page Cache 未命中叶子节点，且二级页表也未命中（对于 onlyStage1 和 allStage 是第一阶段的二级页表未命中），此时需要将请求转发给 Miss queue 或 PTW。如果该请求并不是一个 bypass 请求，同时该请求直接来源于 Miss queue，且 PTW 空闲时将 PTW 请求转发给 PTW。如果是 allStage 请求，第一阶段翻译命中叶子节点，也会发送给 PTW 进行最后一次的第二阶段翻译。如果是 onlyStage2 请求，未命中第二阶段的叶子节点也会发送给 PTW 进行后续翻译
3. 如果该请求是来自于 PTW 或者 LLPTW 发送的第二阶段翻译请求 (hptwReq)，则该请求命中则会发送给 hptw_resp_arb，未命中则发送给 HPTW 处理，如果 HPTW 此时忙，则阻塞 Page Cache。
4. 如果 Page Cache 未命中叶子节点，且该请求并非来自于预取请求也不是 hptwReq 请求。此时需要满足如下三个条件之一可以进入 miss queue。
 1. 该请求是一个 bypass 请求
 2. 该请求二级页表未命中或者第一阶段翻译命中，同时该请求来自于 L1 TLB 或 PTW 无法接收 Page Cache 的请求
 3. 该请求二级页表命中，但 LLPTW 无法接收请求

这里需要特别说明，1、2、3、4 四点是一个并行化的过程，对于 Page Cache 转发的每一个请求，都一定会满足且唯一满足 1、2、3、4 之一的条件，但 1、2、3、4 是分开进行判断的，四者之间没有先后关系。为了更清晰地描述转发请求的情况，用串行化的流程图对此进一步说明，但事实上硬件语言描述的一定是一个并行化的过程，不存在串行关系。串行化的流程图如图 18.31 所示。

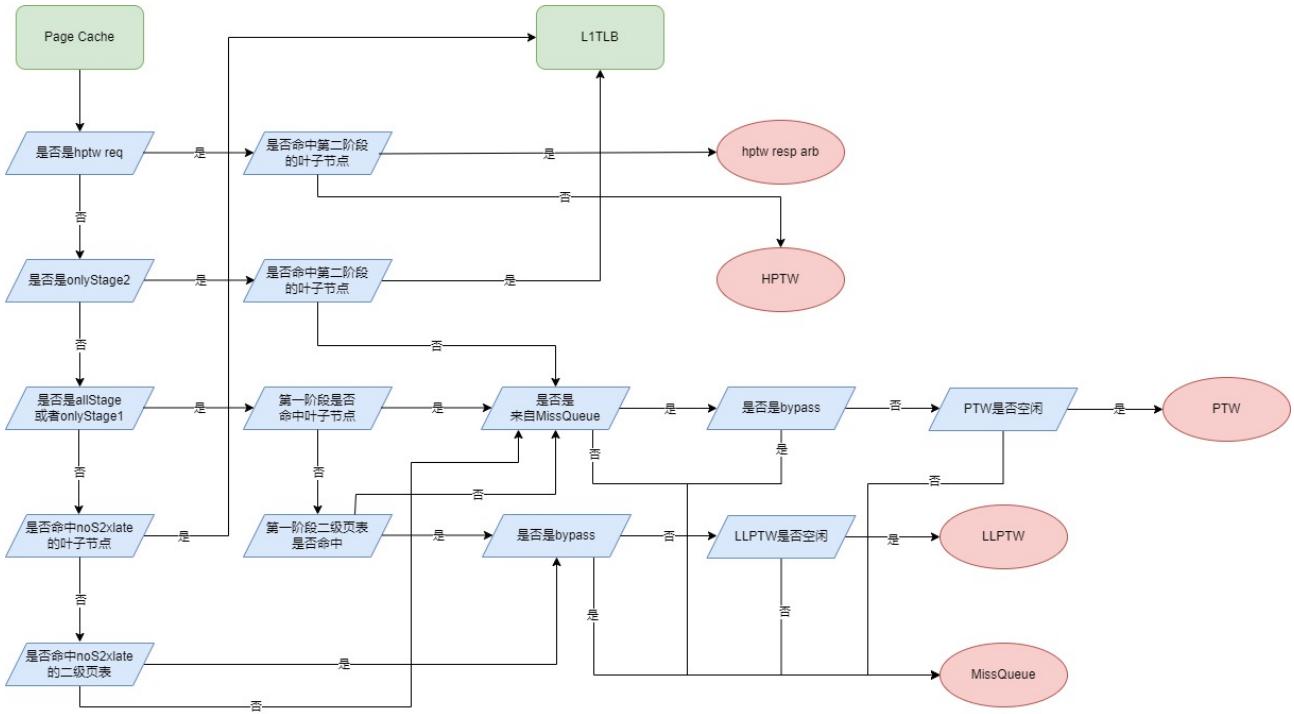


图 18.31: 串行化的 Page Cache 查询流程图

18.4.2.2.4 重填 Cache

当 PTW 或 LLPTW 向 mem 发送的 PTW 请求得到回复时，同时会向 Page Cache 发送 refill 请求。传入 Page Cache 的信息包括：页表项、页表的等级、虚拟页号、页表类型等。在这些信息传入 Cache 后，会根据回填页表的等级以及页表的属性位填入 l1、l2、l3 或 sp 项中。如果页表有效，则根据页表的不同等级分别填入 l1、l2、l3、sp 四项中；如果页表无效，同时页表为一级或二级页表，则填入 sp 项中。对于替换的 Page Cache 项，可以通过 ReplacementPolicy 选定替换策略，目前香山的 Page Cache 采用 PLRU 替换策略。

18.4.2.2.5 支持 bypass 访问

当 Page Cache 的请求发生 miss，但同时对于请求的地址有数据正在写入 Cache，此时会对 Page Cache 的请求做 bypass。如果出现这种情况，并不会直接将写入 Cache 的数据直接交给访问 Page Cache 的请求，Page Cache 会向 L2 TLB 发送 miss 信号，同时向 L2 TLB 发送 bypass 信号，表示该请求是一个 bypass 请求，需要再次访问 Page Cache 以获得结果。bypass 的 PTW 请求并不会进入 PTW，而会直接进入 MissQueue，等待下一次访问 Page Cache 得到结果。但需要注意的是，对于 hptw req（来自 PTW 和 LLPTW）的第二阶段翻译请求，也可能出现 bypass，但 hptw req 不进入 miss queue，所以为了避免重复填入 Page Cache，Page Cache 发送给 HPTW 的信号有 bypassed 信号，该信号有效的时候，这个请求进入 HPTW 后进行的访存的结果不会被重填进入 Page Cache。

18.4.2.2.6 支持 ecc 校验

Page Cache 支持 ecc 校验，当访问 l2 或 l3 项时会同时进行 ecc 检查。如果 ecc 检查报错，并不会报例外，而是会向 L2 TLB 发送该请求 miss 信号。同时 Page Cache 将 ecc 报错的项刷新，重新发送 PTW 请求。其余行为和 Page Cache miss 时相同。ecc 检查采用 seceded 策略。

18.4.2.2.7 支持 sfence 刷新

当 sfence 信号有效时，Page Cache 会根据 sfence 的 rs1 和 rs2 信号以及当前的虚拟化模式对 Cache 项进行刷新。对 Page Cache 的刷新通过将相应 Cache line 的 v 位置 0 进行。由于 l2、l3 项由 SRAM 储存，无法在当拍进行 asid 比较，因此对于 l2、l3 的刷新会忽略 asid（vmid 与 asid 的处理相同）。关于 sfence 信号的有关信息，参见 riscv 手册。虚拟化情况下，sfence 刷新 VS 阶段（第一阶段翻译）的页表（此时需要考虑 vmid）；非虚拟化情况下，sfence 刷新 G 阶段（第二阶段翻译）的页表（此时不考虑 vmid）。

18.4.2.2.8 支持异常处理

在 Page Cache 中可能出现 ecc 校验出错的情况，此时 Page Cache 会将当前项无效掉，返回 miss 结果并重新进行 Page Walk。参见本文档的第 6 部分：异常处理机制。

18.4.2.2.9 支持 TLB 压缩

为支持 TLB 压缩，对于 Page Cache 命中 4KB 页时，需要返回连续的 8 项页表。事实上，Page Cache 每项中由于访存宽度为 512bits 的原因，本就含有 8 个页表，将这 8 个页表直接返回即可。与 L1TLB 不同的是，在 L2TLB 中 H 拓展仍然使用 TLB 压缩。

18.4.2.2.10 支持各级页表分为三种类型

在 H 拓展中有三种类型的页表，分别属于 vsatp、hgatp、satp 管理的页表，在 Page Cache 中新增了一个 h 寄存器，来区别这三种页表，onlyStage1 代表与 vsatp 有关，onlyStage2 代表与 hgatp 有关（此时 asid 无效），noS2xlate 代表与 satp 有关（此时 vmid 无效）。

18.4.2.2.11 支持接收第二阶段翻译请求 (hptw 请求)

L2TLB 中 PTW 和 LLPTW 会发送第二阶段翻译请求（使用 isHptwReq 信号来表示），该类请求会先进入 Page Cache 中查询，查询过程与 onlyStage2 的请求一样，只查询 onlyStage2 类型的页表，但该类请求根据是否命中发送给 hptw_resp_arb 或者 HPTW，Page Cache 的 hptwReq 的返回信号内有 id 信号，该信号用来判断返回给 PTW 还是 LLPTW。返回信号中有个 bypassed 信号，该信号表示该请求出现了 bypass，该请求如果进入 HPTW 翻译，HPTW 访存得到的页表均不会填入 Page Cache。HptwReq 请求也支持 l1Hit 和 l2Hit 的功能。

18.4.2.2.12 支持 hfence 刷新

hfence 指令只能在非虚拟化模式下执行，该类型指令有两条，分别负责刷新 VS 阶段页表（第一阶段翻译，h 字段为 onlyStage1）和 G 阶段页表（第二阶段翻译，h 字段为 onlyStage2）。根据 hfence 的 rs1 和 rs2 以及新增的 vmid 和 h 字段来判断刷新的内容，同样由于 asid 和 vmid 在 l3 和 l2 中在 SRAM 中存储，所以刷新 l3 和 l2 不考虑 vmid 和 asid。此外对于刷新 l3 的实现，采用了简单的做法，直接刷新 VS 或者 G 阶段页表（未来有必要的时候可以进一步细化刷新 addr 所在的 set）。

18.4.2.3 整体框图

Page Cache 的本质是一个 Cache，在上文已经详细介绍过 Page Cache 的内部实现，关于 Page Cache 的内部框图参考意义不大。关于 Page Cache 与其他 L2 TLB 中模块的连接关系，参见 5.3.3 节。

18.4.2.4 接口列表

Page Cache 的信号列表主要可以归纳成如下 3 类:

1. req: arb2 会向 Page Cache 发送 PTW 请求。
2. resp: Page Cache 返回给 L2 TLB 的 PTW 回复, Page Cache 可能向 PTW, LLPTW, Miss Queue, HPTW 发送请求; 向 mergeArb 和 hptw_resp_arb 发送回复。
3. refill: Page Cache 接收 mem 返回的 refill 数据。

具体参见接口列表文档。

18.4.2.5 接口时序

Page Cache 通过 valid-ready 方式与 L2 TLB 中的其他模块进行交互, 涉及到的信号较为琐碎, 且没有特别需要关注的时序关系, 因此不再赘述。

18.4.3 三级模块 Page Table Walker

Page Table Walker 指的是如下模块:

- PTW ptw

18.4.3.1 设计规格

1. 支持访问前两级页表
2. 支持向内存发送 PTW 请求
3. 支持向 LLPTW 转发 PTW 请求
4. 支持向 Page Cache 发送 refill 信号
5. 支持异常处理
6. 支持两阶段地址翻译

18.4.3.2 功能

18.4.3.2.1 访问前两级页表

Page Table Walker 的本质是一个状态机, 逐级通过虚拟地址访问页表得到物理地址。Page Table Walker 只能同时处理一个请求, 同时最多访问前两级页表, 访存能力较弱。Page Table Walker 进行虚实地址转换的行为和手册描述类似, 在访问内存之前需要对访问内存的物理地址进行 PMP 检查。如果 PMP 检查出错, 则直接返回; 否则向内存或 llptw 发送 PTW 请求。添加 H 拓展后, PTW 仍然负责第一阶段前两级页表的翻译, 而前两级翻译中计算出的物理地址都需要进行一次第二阶段地址翻译, 获得真正的物理地址后才能访存, 并且支持了只有第二阶段翻译和只有第一阶段翻译的情况。在内存返回页表项时, 同样需要做 PMP 检查。Page Table Walker 会持续进行以上操作, 直至以下三种情况出现:

1. 访问到叶子节点 (大页), 直接返回给 L1 TLB (如果是 allStage 的翻译, 在返回之前还要进行一次第二阶段翻译)。
2. 访问到二级页表, 返回给 LLPTW, 由 LLPTW 进行最后一级页表的访问
3. 访问出现 Page fault 或 Access fault

18.4.3.2.2 向内存或 llptw 发送 PTW 请求

当 Page Table Walker 访问前两级页表时，需要向内存发送 PTW 请求，当完成前两级页表的访问时，需要向 llptw 发送 PTW 请求。PTW 向内存发送的请求与 LLPTW 向内存发送的请求和 HPTW 向内存发送的请求在发往内存前需要通过仲裁，通过 TileLink 协议的 A 通道与 D 通道的 source 信号标记请求来源于 PTW 或 LLPTW 或 HPTW。

18.4.3.2.3 向 Page Cache 发送 refill 信号

当 Page Table Walker 向内存发送的 PTW 请求得到回复时，会向 Page Cache 发送 refill 请求。Mem 会将返回的页表项回填入 Page Cache，但 Page Table Walker 需要额外提供页表的虚拟页号、页表等级、页表类型等信息。根据翻译请求是否是两阶段地址翻译的请求，填入 Page Cache 的页表分为 noS2xlate 和 onlyStage1

18.4.3.2.4 异常处理机制

Page Table Walker 中可能出现 access fault 异常，会交付给 L1 TLB，L1 TLB 根据请求来源交付处理。参见本文档的第 6 部分：异常处理机制。

18.4.3.3 整体框图

Page Table Walker 的本质是一个 s/w 状态机分为请求和应答事件，每个状态用一对请求事件和应答事件表示，在此以常见的状态转移图以及转移关系来表示，方便理解。关于 Page Table Walker 与其他 L2 TLB 中模块的连接关系，参见 5.3.3 节。

状态机的转移关系图如图 18.32 所示。

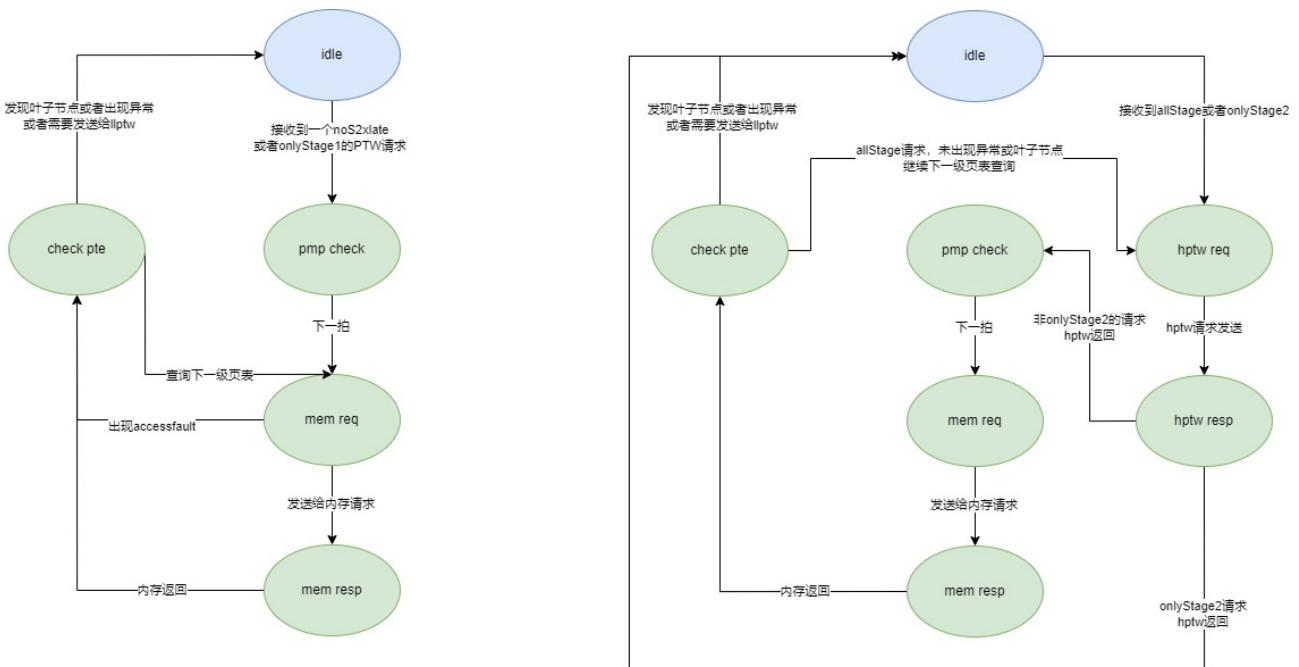


图 18.32: Page Table Walker 状态机的状态转移图

为了清楚表现状态，将不同类型请求分成两种类型状态机分别画出转移图。

对于 noS2xlate 或者 onlyStage1 请求的状态机（上图左边），状态机的各个状态的描述如下：

- idle: Page Table Walker 的初始状态, PTW 接受一个请求后进入 pmp check 状态。
- pmp check: 该状态下将访问的物理地址发送给 PMP 模块做 PMP 和 PMA 检查, 下一拍进入 mem req, PMP 模块当拍返回检查结果是否出现 access fault。
- mem req: 根据检查结果, 如果出现 access fault, 则直接进入最后检查 check pte 状态 (chisel 代码中为 mem_addr_update 信号有效表示 check pte), 如果没有出现 access fault, 则发送内存访问请求, 并且进入 mem resp 状态。
- mem resp: 该状态等待 mem 返回, 返回后进入 check pte 状态。
- check pte: 在该状态中, 会检查当前请求, 决定下一步的动作:
 1. 没有找到叶子节点也没有出现 access fault, 此时的 level 为第一级页表, 转移到 mem req 状态
 2. 出现 access fault, 直接返回 L1TLB, 状态转移 idle
 3. 发现二级页表, 且不是叶子节点, 则发送给 llptw
 4. 找到了叶子节点 (大页), 返回给 L1TLB

对于 allStage 和 onlyStage2 的请求:

- idle, 接收到这两种类型请求后, 会进入 hptw req 状态, 直接开始第二阶段的翻译。
- hptw req: 向 L2TLB 发送第二阶段翻译的请求, 发送完毕后进入 hptw resp 状态。
- hptw resp: 等待 hptw 请求返回, hptw 返回后, 如果当前是个 onlyStage2 的请求, 则直接进入 check pte 状态, 否则进入 pmp check 状态。
- pmp check: 该状态下将访问的物理地址发送给 PMP 模块做 PMP 和 PMA 检查, 下一拍进入 mem req, PMP 模块当拍返回检查结果是否出现 access fault。
- mem req: 根据检查结果, 如果出现 access fault, 则直接进入最后检查 check pte 状态 (chisel 代码中为 mem_addr_update 信号有效表示 check pte), 如果没有出现 access fault, 则发送内存访问请求, 并且进入 mem resp 状态。
- mem resp: 该状态等待 mem 返回, 返回后进入 check pte 状态。
- check pte: 如果是非 onlyStage2 请求 (也就是 allStage), 并且没有找到叶子节点也没有出现 accessfault, 则进入 hptw req 请求。如果没找到叶子节点, 并且 level 已经是二级页表, 则把请求发送给 llptw。如果此时 last s2xlate 信号有效, 则表明返回之前还需要进行一次第二阶段的地址翻译 (onlyStage2 请求则不需要)。如果此时找到了叶子节点, 并且也进行了最后一次地址翻译, 则返回给 L1TLB。

需要注意的是, PTW 也处理 stage1 命中的 allStage 请求, 该请求进入后, 进行一次第二阶段翻译后就直接返回 L1TLB。

18.4.3.4 接口列表

Page Table Walker 的信号列表可以归纳为以下几类:

1. req: Page Table Walker 只接受 Page Cache 的请求, 同时需要满足相应条件, 参见 5.3.7 节中对 Page Cache 的介绍。
2. resp: Page Table Walker 如果访问到大页或者 PMP&PMA 检查报错时, 将相应信息返回给 L1 TLB。
3. llptw: Page Table Walker 如果访问只剩最后一级页表或者 PMP&PMA 检查报错时, 将相应信息返回给 L1 TLB。
4. mem: Page Table Walker 需要访问内存时与内存的交互, 涉及到 req 和 resp, Page Table Walker 和内存的握手信号也用来控制 Page Table Walker 状态机的转移关系。
5. pmp: Page Table Walker 和 PMP 模块的交互, 用于 PMP 和 PMA 检查。

6. refill: Page Table Walker 访问内存得到结果后, 需要将内存返回的结果以及相应信息回填入 Page Cache 中。
7. hptw: Page Table Walker 得到客户机物理地址后, 向 L2TLB 发送第二阶段翻译请求。并且 L2TLB 会发送查询结果给 PTW。

具体参见接口列表文档。

18.4.3.5 接口时序

Page Table Walker 通过 valid-ready 方式与 L2 TLB 中的其他模块进行交互, 涉及到的信号较为琐碎, 且没有特别需要关注的时序关系, 因此不再赘述。

18.4.4 三级模块 Last Level Page Table Walker

Last Level Page Table Walker 指的是如下模块:

- LLPTW llptw

18.4.4.1 设计规格

1. 支持访问最后一级页表
2. 支持并行处理多项请求
3. 支持向内存发送 PTW 请求
4. 支持向 Page Cache 发送 refill 信号
5. 支持异常处理机制
6. 支持第二阶段翻译

18.4.4.2 功能

18.4.4.2.1 访问最后一级页表

Last Level Page Table Walker 的作用是访问最后一级页表, 同时可以提高 Page Table Walker 的访问并行度。Page Table Walker 同时只能够处理一个请求, 而 LLPTW 可以同时处理多个请求, 如果多个请求之间存在重复, LLPTW 并不会合并重复的请求, 而是将这些请求记录下来, 共享访存的结果, 避免重复访问内存。

LLPTW 可能接收 Page Cache 或 Page Table Walker 的请求。对于 Page Cache 的请求, 需要满足命中二级页表、未命中第三级页表, 且不是 bypass 请求。对于 Page Table Walker 的请求, 由于已经满足只缺少最后一级页表, 因此可以通过 LLPTW 访问内存。通过仲裁器将 Page Cache 和 Page Table Walker 的请求做仲裁, 并送往 LLPTW。

Page Table Walker 和 LLPTW 共同合作, 可以共同完成 Page Table Walk 的全流程。为了提高访存的并行度, LLPTW 为请求配置不同的 id, 可以同时拥有多个 inflight 的请求。由于不同请求的前两级页表可能相同, 同时考虑到前两级页表的 miss 概率比最后一级页表低, 因此无需考虑提高前两级页表的访问并行度, 只通过 Page Table Walker 处理单条请求, 降低设计的复杂度。

18.4.4.2.2 并行处理多项请求

LLPTW 可以同时处理多个请求, 并行处理的数量为 LLPTW 的项数。如果多个请求之间存在重复, LLPTW 并不会合并重复的请求, 而是将这些请求记录下来, 共享访存的结果, 避免重复访问内存。LLPTW 的每一项通过

状态机维护访问内存的状态，当 LLPTW 新接收一条请求时，会将新请求的地址和已有请求的地址进行比较，如果地址相同，则将已有请求的状态复制给新请求。因此这些地址相同的请求可以共享访存结果，避免重复访问请求。

18.4.4.2.3 向内存发送 PTW 请求

和 Page Table Walker 的行为类似，LLPTW 同样可以向内存发送 PTW 请求。LLPTW 会将重复的请求合并，共享访存结果，避免重复访问内存。由于内存每次返回的数据为 512 bits 较大，因此返回的结果并不会存储在 LLPTW 中。如果在 LLPTW 向内存发送的 PTW 请求得到结果时向 LLPTW 传入 PTW 请求，同时该请求的物理地址与内存返回的物理地址匹配，则将该请求发送给 Miss queue，等待下次访问 Page Cache。

18.4.4.2.4 向 Page Cache 发送 refill 信号

Last Level Page Table Walker 向 Page Cache 发送 refill 信号的逻辑也与 Page Table Walker 类似，这里不再赘述。

18.4.4.2.5 异常处理机制

Last Level Page Table Walker 中可能出现 access fault 异常，会交付给 L1 TLB，L1 TLB 根据请求来源交付处理。参见本文档的第 6 部分：异常处理机制。

18.4.4.2.6 支持第二阶段翻译

新增了 state_hptw_req、state_hptw_resp、state_last_hptw_req 和 state_last_hptw_resp 四个状态，当一个两阶段翻译请求进入 LLPTW 后，首先进行一次第二阶段翻译，获取三级页表的真实物理地址，然后进行地址检查，访存，得到三级页表后，返回之前还需要进行一次第二阶段翻译，获得最后的物理地址。

每个 entry 新增了 hptw resp 结构，用来保存每次第二阶段翻译的结果。在第一次第二阶段翻译，hptw 返回的时候，会检查所有项，如果有在同一个 cacheline 的访存请求已经发送，则直接进入 mem waiting 阶段。

LLPTW 新增了一些仲裁器用于第二阶段翻译。hyper_arb1，用于第一次第二阶段地址翻译，对应 hptw req 状态；hyper_arb2，用于第二次第二阶段地址翻译，对应 last hptw req 状态。hptw_req_arb 输入端为 hyper_arb1 和 hyper_arb2，输出为 LLPTW 的 hptw 请求的输出信号。

18.4.4.3 整体框图

虽然 Last Level Page Table Walker 可以并行处理多项对最后一级页表的访问，但内部的逻辑和 Page Table Walker 同样通过状态机实现。在此介绍状态机的状态转移图以及转移关系。关于 Last Level Page Table Walker 与其他 L2 TLB 中模块的连接关系，参见 5.3.3 节。

状态机的转移关系图如图 18.33 所示，该状态机为非两阶段地址翻译的请求的状态转移情况。

在添加了虚拟化拓展后，LLPTW 接收到两阶段地址翻译请求后，状态机如图 18.34 所示。

对于进入 LLPTW 的请求，并不都从 idle 状态开始，而是根据 LLPTW 中已有项的情况，分别可能进入 idle、addr_check、mem_waiting、mem_out，或 cache 状态。对于两阶段地址翻译的请求，则可能进入 hptw_req、cache、mem_waiting 和 last_hptw_req 状态。

- idle：初始状态，当结束一个 LLPTW 请求后回复为 idle 状态，表示 LLPTW 中的该项为空。当预取请求进入 LLPTW，同时该请求和其他 LLPTW 的请求重复时，不需接收该预取请求，保持 LLPTW 项为 idle。可能从三种情况返回 idle 状态：

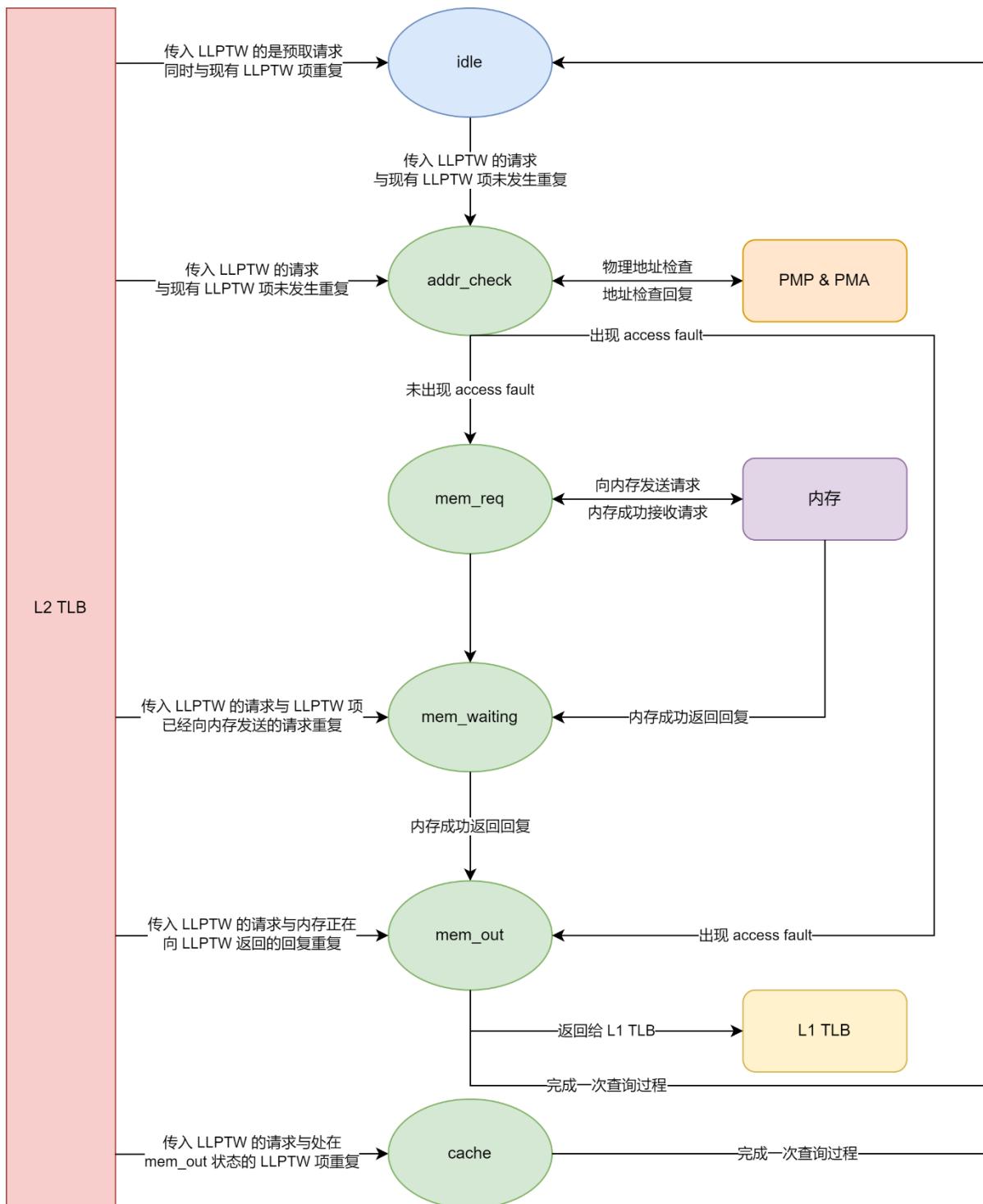


图 18.33: Last Level Page Table Walker 状态机的状态转移图

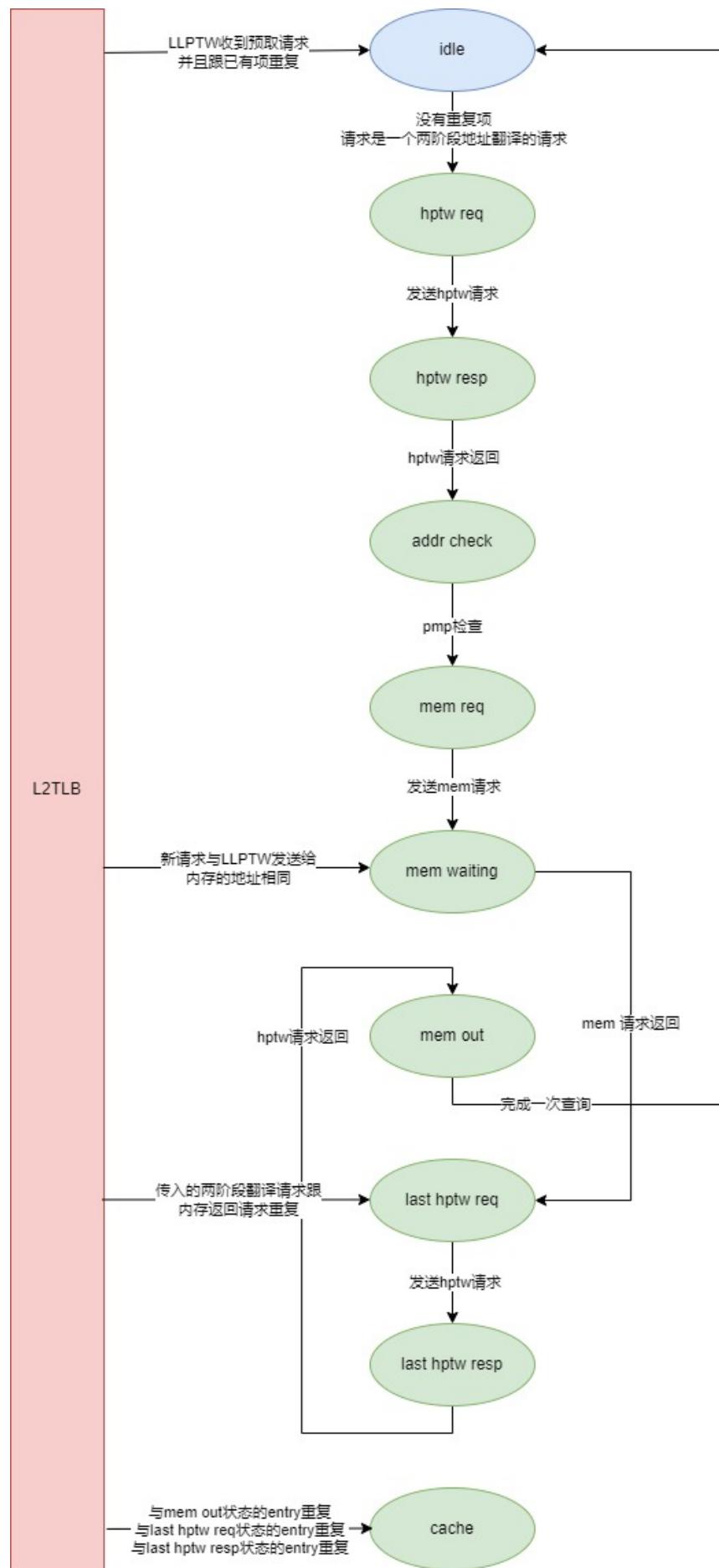


图 18.34: Last Level Page Table Walker 处理 allStage 请求的状态机的状态转移图

1. 当前为 mem_out 状态, PMP&PMA 检查出现 access fault, 返回给 L1 TLB, 状态转移为 idle 状态。
 2. 当前为 mem_out 状态, 查询得到最后一级页表, 返回给 L1 TLB, 状态转移为 idle 状态。
 3. 当前为 cache 状态, 想要查询的页表已经被写入 Page Cache 中, 需要返回给 Page Cache 继续查询, 状态转移为 idle 状态。
- hptw_req: 当传入的请求是两阶段地址翻译请求的时候进入该状态。该状态会发送 hptw 请求给 L2TLB。
 - hptw_resp: 当 hptw 请求发送后, 进入该状态, 等待 hptw 请求返回。请求返回后, 如果与已有处于 mem_waiting 的 LLPTW 项重复, 则进入 mem_waiting, 否则进入 addr_check。
 - addr_check: 当传入 LLPTW 的请求和 LLPTW 中现有的请求未发生重复时且该请求非两阶段翻译的请求, 进入该状态; 此外对于两阶段地址翻译的请求, 当 hptw 请求返回后, 也进入该状态, 同时需要将物理地址发送给 PMP 模块进行 PMP&PMA 检查。PMP 模块需要当拍返回 PMP&PMA 检查结果, 如果未出现 access fault, 则进入 mem_req 状态, 否则进入 mem_out 状态。
 - mem_req: 该状态已经完成了 PMP&PMA 检查, 可以向内存 (mem_arb) 发送请求。对于每个 LLPTW 项, 当 mem_arb 发送的内存访问请求对应的虚拟页号与 LLPTW 项中的虚拟页号相同时, 进入 mem_waiting 状态, 等待内存的回复。
 - mem_waiting: 当传入 LLPTW 的请求和 LLPTW 已经向内存发送的 PTW 请求对应的虚拟页号相同时, 新请求的 LLPTW 项的状态设置为 mem_waiting。该状态等待内存的回复, 当内存回复的页表项对应该 LLPTW 项时, 对于非两阶段地址翻译的 LLPTW 项, 状态转移为 mem_out, 而对于两阶段地址翻译的 LLPTW 项, 状态转移为 last_hptw_req。
 - last_hptw_req: 当传入 LLPTW 的请求和内存正在给 LLPTW 回复的请求对应的虚拟页号相同并且该请求为两阶段翻译的请求, 当访存得到了最后的页表后, 进入该状态, 进行最后一次第二阶段地址翻译, 发送 hptw 请求。
 - last_hptw_resp: 等待 hptw 请求返回。Hptw 请求返回后进入 mem_out 状态
 - mem_out: 当传入 LLPTW 的请求和内存正在给 LLPTW 回复的请求对应的虚拟页号相同时并且该请求非两阶段翻译请求, 新请求的 LLPTW 项的状态设置为 mem_out。由于此时已经完成了三级页表的查找, 因此向 L1 TLB 返回查询得到的虚拟地址以及页表项即可。另外, 对于在 addr_check 状态产生 access fault 的情况, 也需要返回给 L1 TLB, 并向 L1 TLB 报告 access fault。成功将信息返回给 L1 TLB 后, 状态转移为 idle。
 - cache: 当传入 LLPTW 的请求和正在处于 mem_out/last_hptw_req/last_hptw_resp 的 LLPTW 项的虚拟页号相同时, 此时查询内存得到的页表项已经被写回 Cache, 因此需要向 Cache 发送查询请求, 新请求的 LLPTW 项的状态设置为 cache。当 Cache (实际上是 mq_arb) 接收该请求后, 状态转移为 idle。

18.4.4.4 接口时序

Last Level Page Table Walker 通过 valid-ready 方式与 L2 TLB 中的其他模块进行交互, 涉及到的信号较为琐碎, 且没有特别需要关注的时序关系, 因此不再赘述。

18.4.5 三级模块 Hypervisor Page Table Walker

Hypervisor Page Table Walker 指的是如下模块:

- HPTW hptw

18.4.5.1 设计规格

1. 支持访问 G-stage 的三级页表

2. 支持向内存发送请求
3. 支持向 Page Cache 发送 refill 信号
4. 支持异常处理
5. bypass 特殊处理

18.4.5.2 功能

18.4.5.2.1 支持访问 G-stage 的三级页表

HPTW 整体设计与 PTW 基本相同，每次只能处理一个请求，并且 HPTW 可以进行完整的第二阶段的三级页表的翻译，翻译过程中如果需要访问内存，会对访问内存的地址进行 PMP 检查，如果检查出错则直接返回，否则发送访存请求。HPTW 返回的情况有：

1. 访问到叶子节点
2. 访问出现 pagefault 或者 accessfault

18.4.5.2.2 支持向内存发送请求

HPTW 与 PTW 和 LLPTW 类似，在访问页表的时候需要向内存发送请求，通过仲裁器发送请求。

18.4.5.2.3 向 Page Cache 发送 refill 信号

HPTW 得到 PTW 返回的结果后，会向 Page Cache 发送 refill 请求，将返回的页表项填入 Page Cache，HPTW 会提供填入 Page Cache 的信息。

18.4.5.2.4 异常处理机制

当出现 pagefault 或者 accessfault 的异常时候，HPTW 会直接返回给 PTW 或者 LLPTW。

18.4.5.2.5 Bypass 特殊处理

对于 bypass 请求，我们一般将其放入 MissQueue 中重新查询，但对于 hptw 请求（即 isHptwReq 有效）不放入 MissQueue（避免出现阻塞），所以出现 bypass 的请求的时候，为了避免 Page Cache 填入重复的页表项，hptw 请求传入 HPTW 的时候如果 bypass 信号有效，则向内存发送请求的时候，内存返回的结果不会重填 Page Cache。

18.4.5.3 整体框图

HPTW 的状态转移关系图如图 18.35 所示。

状态机的各个状态的描述如下：

- idle：Hypervisor Page Table Walker 状态机的初始状态，接收一个 PTW 请求后，进入 pmp_check 状态。
- pmp_check：在该状态下将需要访问的物理地址发送给 PMP 模块做 PMP 和 PMA 检查，下一拍进入 mem_req 状态。PMP 模块需要当拍返回物理地址检查结果是否出现 access fault。

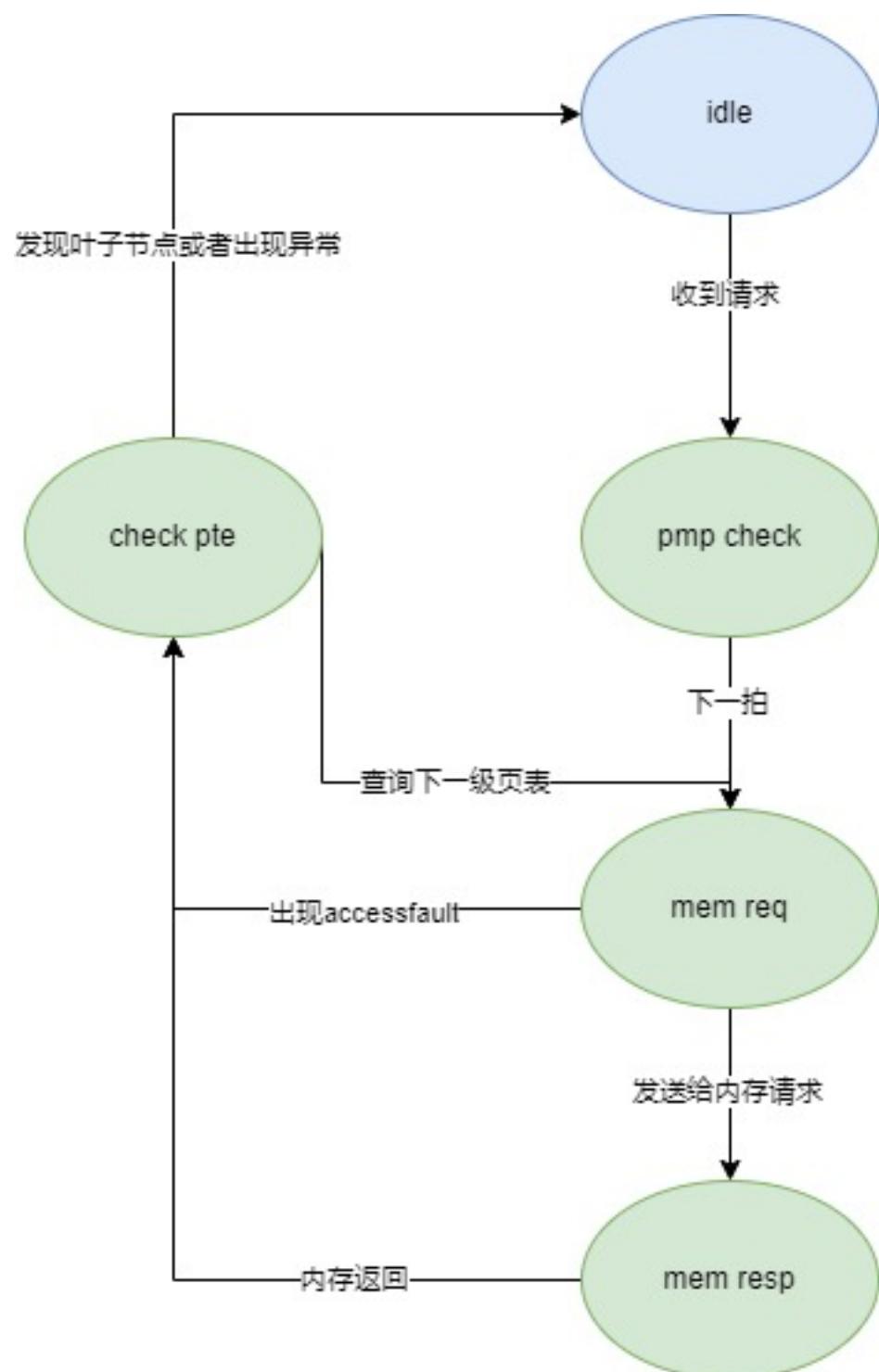


图 18.35: Hypervisor Page Table Walker 状态机的状态转移图

- mem_req: 根据 PMP 和 PMA 的检查结果, 如果检查结果出现 access fault, 则进入 check_pte 状态; 否则向内存发送请求。在 mem_req 状态继续等待, 直至和内存握手成功, 表示成功发送请求, 之后后进入 mem_resp 状态。
- mem_resp: 在 mem_req 状态时, Hypervisor Page Table Walker 已经向内存发送了 PTW 请求, 在 mem_resp 状态下, Hypervisor Page Table Walker 等待内存回复。在收到内存回复, 内存和 Hypervisor Page Table Walker 握手成功后进入 check_pte 状态。
- check_pte: 该状态对目前的查询情况进行判断, 从而决定下一步操作。该状态处理的情况如下:
 1. 出现 accessfault 或者 pagefault, 返回给 PTW 或者 LLPTW。
 2. 在内存返回的页表是叶子节点, 则直接返回给 PTW 或者 LLPTW。
 3. 如果不是叶子节点, 则将物理地址发送给 PMP 模块做 PMP&PMA 检查, 同时状态转移到 mem_req, 重复上文中介绍的流程。

18.4.5.4 接口时序

与 PTW 类似, 不再赘述。

18.4.6 三级模块 Miss Queue

Miss Queue 指的是如下模块:

- L2TlbMissQueue missQueue

18.4.6.1 设计规格

1. 缓冲请求等待资源

18.4.6.2 功能

18.4.6.2.1 缓冲请求等待资源

Miss Queue 的本质是一个队列, 接收来自 Page Cache 和 Last Level Page Table Walker 的请求, 发送给 Page Cache。当 Page Cache 发送给 PTW 但请求是 isFirst 或者 PTW busy, 则发送给 Miss Queue, 当 Page Cache 发送给 LLPTW 但 LLPTW busy, 则发送给 Miss Queue。

18.4.6.3 整体框图

Miss Queue 的整体结构较为简单, 不再赘述。关于 Miss Queue 与其他 L2 TLB 中模块的连接关系, 参见 5.3.3 节。

18.4.6.4 接口时序

Miss Queue 的本质是一个队列, 接口时序比较简单, 不再赘述,

18.4.7 三级模块 Prefetcher

Prefetcher 指的是如下模块:

- L2TlbPrefetch prefetch

18.4.7.1 设计规格

1. 支持 Next-line 预取算法
2. 支持过滤重复的历史请求

18.4.7.2 功能

18.4.7.2.1 发出预取请求

当满足如下两种情况之一时会发出预取请求：

1. Page Cache 发生 miss
2. Page Cache hit, 但命中的是预取项

Prefetcher 采用 Next-Line 预取算法，预取结果会保存在 Page Cache 中，并不会返回给 L1 TLB。由于 Page Table Walker 的访存能力较弱，预取请求并不会进入 Page Table Walker 或 Miss Queue，而是会被直接丢弃。当预取请求只差最后一级页表缺失时，可以访问 LLPTW。同时，在 Prefetcher 中添加了 Filter Buffer，可以起到过滤重复的预取请求的目的。

18.4.7.2.2 过滤重复的历史请求

为避免重复的请求浪费 L2 TLB 的资源，同时提高 Prefetcher 的利用率，当满足 5.3.11.2 节描述的两种情况，发出预取请求时，会判断相同地址的预取请求是否已经发出，如果发出则丢弃新收到的预取请求。当前 Prefetcher 模块会过滤最近的 4 条请求。

18.4.7.3 整体框图

Prefetcher 的整体框图如图 18.36 所示。当 Page Cache 发生 miss 或 Page Cache hit, 但命中的是预取项时，会产生预取请求。通过 Filter Buffer 可以过滤重复的预取请求。

18.4.7.4 接口时序

Prefetcher 是一个 next-line 预取器，接口时序较为简单，这里不再赘述。

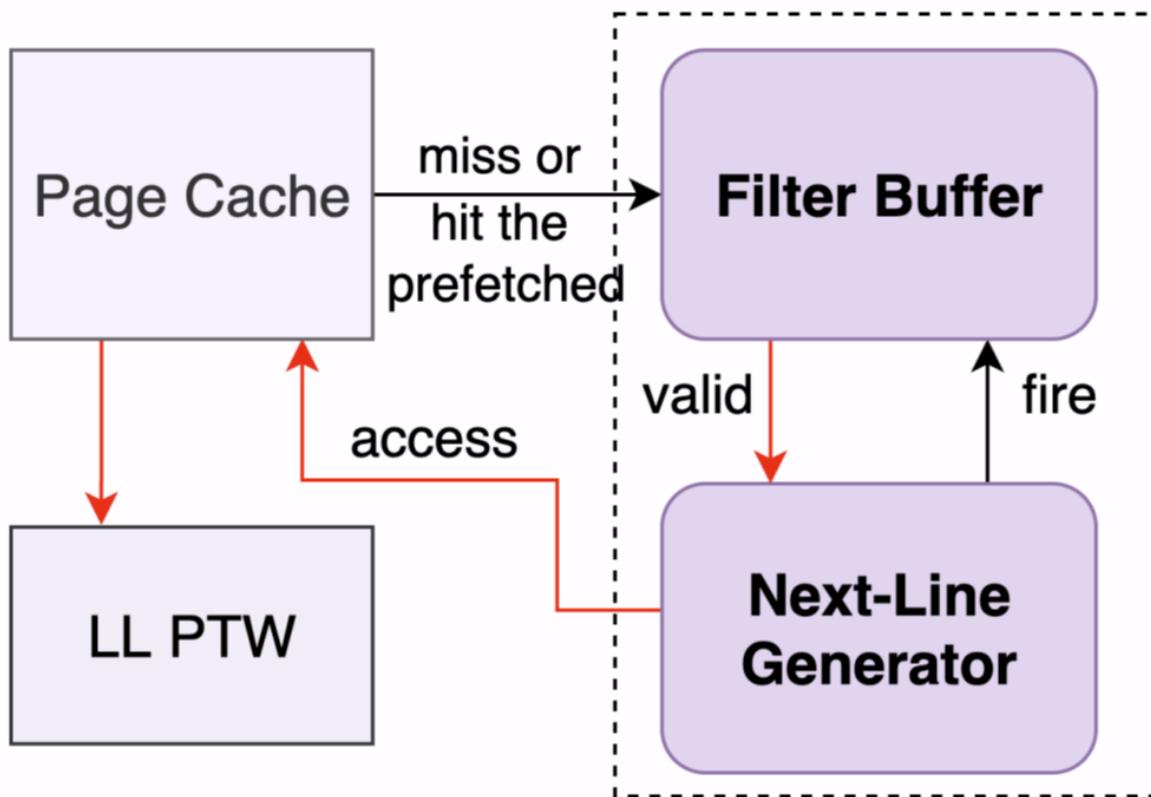


图 18.36: Prefetcher 的整体框图

18.5 二级模块 PMP&PMA

PMP 包括如下模块，PMA 检查包括在 PMP 模块中：

1. PMP（分布式 PMP & PMA 寄存器）
 1. PMP pmp (Frontend)
 2. PMP pmp (Memblock)
 3. PMP pmp (L2TLB)
2. PMPChecker (PMP & PMA 检查器, 当拍返回结果)
 1. PMPChecker PMPChecker (Frontend)
 2. PMPChecker PMPChecker_1 (Frontend)
 3. PMPChecker PMPChecker_2 (Frontend)
 4. PMPChecker PMPChecker_3 (Frontend)
 5. PMPChecker PMPChecker (L2TLB)
 6. PMPChecker PMPChecker_1 (L2TLB)
3. PMPChecker_8 (PMP & PMA 检查器, 下一拍返回结果)
 1. PMPChecker_8 PMPChecker (Memblock)
 2. PMPChecker_8 PMPChecker_1 (Memblock)
 3. PMPChecker_8 PMPChecker_2 (Memblock)
 4. PMPChecker_8 PMPChecker_3 (Memblock)

5. PMPChecker_8 PMPChecker_4 (Memblock)
6. PMPChecker_8 PMPChecker_5 (Memblock)

18.5.1 设计规格

1. 支持物理地址保护
2. 支持物理地址属性
3. 支持 PMP 和 PMA 并行执行检查
4. 支持动态检查和静态检查
5. 支持分布式 PMP 和分布式 PMA
6. 支持异常处理机制

18.5.2 功能

18.5.2.1 支持物理地址保护

香山处理器支持物理地址保护（PMP）检查，PMP 默认为 16 项，可以参数化修改。出于时序考虑，采用分布复制式实现方法，在 CSR 单元中的 PMP 寄存器负责 CSRRW 等指令；在前端取指、后端访存、Page Table Walker 处都拥有一份 PMP 寄存器的拷贝，通过拉取 CSR 写信号保证和 CSR 单元中 PMP 寄存器的一致。

PMP 寄存器的格式、复位值等请参考香山开源处理器用户手册、RISC-V 特权级手册。

18.5.2.2 支持物理地址属性

物理地址属性（PMA）的实现采用了类 PMP 的方式，利用了 PMP Configure 寄存器的两个保留位，设为 atomic 和 cachable，分别为是否支持原子操作和是否可缓存。PMP 寄存器没有初始值，而 PMA 寄存器默认拥有初始值，需手动设置与平台地址属性一致。PMA 寄存器利用了 M 态 CSR 的保留寄存器地址空间，默认为 16 项，可参数化修改。

PMA 默认配置请参考香山开源处理器用户手册。

18.5.2.3 PMP 和 PMA 并行执行检查

PMP 和 PMA 检查并行查询，如果违反其中一个权限，即为非法操作。核内所有的物理地址访问都需要进行物理地址权限检查，包括在 ITLB 和 DTLB 检查之后，以及 Page Table Walker、Hypervisor Page Table Walker 和 Last Level Page Table Walker 访存之前。ITLB、DTLB、Page Table Walker、Last Level Page Table Walker、Hypervisor Page Table Walker 各自使用的分布式 PMP、PMA，以及 PMP、PMA 检查器的对应关系如表 18.24 所示。也就是说，对于 Frontend、Memblock 以及 L2 TLB 各自拥有一份 PMP 和 PMA 寄存器的备份（参见 5.2.5 节），这些备份分别驱动 Frontend、Memblock 以及 L2 TLB 的 PMP 和 PMA 检查器。

表 18.24: PMP 和 PMA 检查模块的对应关系

| 模块 | 通道 | 分布式 PMP&PMA | PMP&PMA 检查器件 |
|------|--------------|----------------|--------------|
| ITLB | | | |
| | requestor(0) | pmp (Frontend) | PMPChecker |
| | requestor(1) | pmp (Frontend) | PMPChecker_1 |
| | requestor(2) | pmp (Frontend) | PMPChecker_2 |

| 模块 | 通道 | 分布式 PMP&PMA | PMP&PMA 检查器件 |
|---------|------------------------------|----------------|--------------|
| | requestor(3) | pmp (Frontend) | PMPChecker_3 |
| DTLB_LD | requestor(0) | pmp (Memblock) | PMPChecker |
| | requestor(1) | pmp (Memblock) | PMPChecker_1 |
| | requestor(2) | pmp (Memblock) | PMPChecker_2 |
| DTLB_ST | requestor(0) | pmp (Memblock) | PMPChecker_3 |
| | requestor(1) | pmp (Memblock) | PMPChecker_4 |
| DTLB_PF | requestor(0) | pmp (Memblock) | PMPChecker_5 |
| L2 TLB | Page Table Walker | pmp (L2 TLB) | PMPChecker |
| | Last Level Page Table Walker | pmp (L2 TLB) | PMPChecker_1 |
| | Hypervisor Page Table Walker | Pmp (L2TLB) | PMPChecker_2 |

根据 RV 手册, Page Fault 的优先级高于 Access Fault, 但是如果 Page Table Walker 或 Last Level Page Table Walker 在 PMP 检查或 PMA 检查时出现 Access Fault, 此时页表项为非法, 会发生 Page Fault 和 Access Fault 一起出现的特殊情况, 香山选择报 Access Fault。手册对于这种情况没有明确说明, 或与手册不符, 其余情况下均满足 Page Fault 的优先级高于 Access Fault。

18.5.2.4 动态检查和静态检查

按照手册规定, PMP 和 PMA 的检查应该为动态检查, 即需要经过 TLB 翻译之后, 使用翻译后的物理地址进行物理地址权限检查。Frontend、L2 TLB 以及 Memblock 的 5 个 PMPChecker (参见表 18.24) 均为动态检查。出于时序考虑, DTLB 的 PMP & PMA 检查结果可以提前查询好, 在回填时存入 TLB 项中, 此为静态检查。具体地, 当 L2 TLB 的页表项回填入 DTLB 时, 同时将回填的页表项送给 PMP 和 PMA 进行权限检查, 将检查得到的属性位 (包括 R、W、X、C、Atomic, 这些属性位的具体含义参见 5.4 节) 同时存储在 DTLB 中, 这样可以直接将这些检查结果返回给 MemBlock, 无需再次检查。为实现静态检查, 需要提升 PMP 和 PMA 的粒度为 4KB。

需要注意的是, 目前 PMP & PMA 检查暂时并非昆明湖的时序瓶颈, 因此未采用静态检查, 全部使用动态检查的方式, 即 TLB 查询得到物理地址后, 再进行检查。昆明湖 V1 的代码中不包括静态检查, 只包括动态检查, 请再次注意。但出于兼容性, PMP 和 PMA 的粒度依然保持为 4KB。

动态检查和静态检查得到的结果信息如下:

- 动态检查: 返回是否发生 inst access fault, load access fault, store access fault; 检查的物理地址是否属于 mmio 地址空间
- 静态检查: 返回检查的物理地址的属性位, 包括 R、W、X、C、Atomic。请注意, 昆明湖 V1 默认不会采用静态检查。

18.5.2.5 分布式 PMP 和 PMA

PMP 和 PMA 的具体实现包括 CSR Unit、Frontend、Memblock、L2 TLB 四部分, 在 CSR Unit 负责响应 CSRRW 等 CSR 指令对这些 PMP 和 PMA 寄存器的读写。由于 CSR Unit 和 ITLB、DTLB、L2 TLB 距离均

较远，需要在 ITLB、DTLB、L2 TLB 中均存储一份 PMP 和 PMA 的备份用于物理地址检查和物理属性检查。为此，我们需要实现分布式 PMP 和 PMA，在 ITLB、DTLB、L2 TLB 附近都保存这些寄存器的备份。

在 Frontend、Memblock 和 L2 TLB 中包含这些 PMP 和 PMA 寄存器的备份，负责地址检查，通过拉取 CSR 的写信号可以保证这些寄存器的内容一致性。由于 L1 TLB 的面积较小，因此 PMP 和 PMA 寄存器的备份存储在 Frontend 或 Memblock 中，分别为 ITLB 和 DTLB 提供检查。L2 TLB 的面积较大，因此 PMP 和 PMA 寄存器的备份直接存储在 L2 TLB 中。

18.5.2.6 PMP 和 PMA 的检查流程

在 ITLB、DTLB 查询得到物理地址，以及 L2 TLB 的 Page Table Walker、Last Level Page Table Walker 和 Hypervisor Page Table Walker 访问内存之前，都需要进行物理地址检查。ITLB、DTLB、L2 TLB 需要向 PMPChecker 提供的信息包括 PMP 和 PMA 配置寄存器、地址寄存器的相关信息；PMP 和 PMA 地址寄存器由低向高数连续 1 的个数（由于 PMP 和 PMA 的粒度为 4KB，因此最小为 12）；需要查询的物理地址；需要查询哪种权限，包括执行 (ITLB)、读写 (L2 TLB、LoadUnits 和 StoreUnits)、原子读写 (AtomicsUnit)。

PMP 和 PMA 检查请求需要提供的相关信息如表 18.25 所示：

表 18.25: PMP 和 PMA 检查请求需要提供的相关信息

| PMPChecker | | |
|------------|--|---|
| 模块 | 需要提供的信息 | 来源 |
| Frontend | | |
| | PMP 和 PMA 配置寄存器 | Frontend pmp |
| | PMP 和 PMA 地址寄存器 | Frontend pmp |
| | PMP 和 PMA 的 mask，也就是地址寄存器由低向高数连续 1 的个数，最小为 12 | Frontend pmp |
| | 查询的 paddr | Icache、IFU |
| | 查询的 cmd, ITLB 固定为 2，表示需要执行权限 | Icache、IFU |
| Memblock | | |
| | PMP 和 PMA 配置寄存器 | Memblock pmp |
| | PMP 和 PMA 地址寄存器 | Memblock pmp |
| | PMP 和 PMA 的 mask，也就是地址寄存器由低向高数连续 1 的个数，最小为 12 | Memblock pmp |
| | 查询的 paddr | LoadUnits、L1 Load Stream & Stride Prefetch StoreUnits、AtomsicsUnit、 SMSprefetcher |
| | 查询的 cmd, DTLB 可能为 0、1、4、5；分别表示需要 read、write、atom_read、atom_write 权限。 | LoadUnits、L1 Load Stream & Stride Prefetch StoreUnits、AtomsicsUnit、 SMSprefetcher |
| Memblock | | |
| 静态检查 | | |
| | PMP 和 PMA 配置寄存器 | Memblock pmp |
| | PMP 和 PMA 地址寄存器 | Memblock pmp |

| PMPChecker | | |
|------------|--|---|
| 模块 | 需要提供的信息 | 来源 |
| | PMP 和 PMA 的 mask, mask 的形式是低 i 位为 1, 高位为 0, i 的个数为 log2(pmp 条目匹配的地址空间) | Memblock pmp |
| | 查询的 paddr | L2 TLB 返回的 PTW |
| L2 TLB | | |
| | PMP 和 PMA 配置寄存器 | L2 TLB pmp |
| | PMP 和 PMA 地址寄存器 | L2 TLB pmp |
| | PMP 和 PMA 的 mask, mask 的形式是低 i 位为 1, 高位为 0, i 的个数为 log2(pmp 条目匹配的地址空间) | L2 TLB pmp |
| | 查询的 paddr | Page Table Walker、Last Level Page Table Walker、Hypervisor Page Table Walker |
| | 查询的 cmd, L2 TLB 固定为 0, 表示需要读权限 | Page Table Walker、Last Level Page Table Walker、Hypervisor Page Table Walker |

PMPChecker 需要向 ITLB、DTLB、L2 TLB 返回是否出现 inst access fault (ITLB)、是否出现 load access fault (LoadUnits、L2 TLB)、是否出现 store access fault (StoreUnits、AtomicsUnit)、地址是否属于 MMIO 空间 (ITLB、DTLB、L2 TLB)。同时静态检查需要向 DTLB 填入地址的属性位，包括 cacheable、atomic、x、w、r。

对于来自 ITLB 和 L2 TLB 的请求，会在当拍给出 PMP 和 PMA 检查的结果；对于来自 DTLB 的请求，会在下一拍给出 PMP 和 PMA 检查的结果。PMP 和 PMA 检查需要返回的相关信息如表 18.26 所示：

表 18.26: PMP 和 PMA 检查需要返回的相关信息

| PMPChecker | | |
|------------|-------------------------|----------------------------------|
| 模块 | 需要返回的信息 | 去向 |
| Frontend | | |
| | 是否出现 inst access fault | Icache、IFU |
| | 地址是否属于 MMIO 空间 | Icache、IFU |
| Memblock | | |
| 动态检查 | 是否出现 load access fault | LoadUnits |
| | 是否出现 store access fault | StoreUnits、AtomicsUnit |
| | 地址是否属于 MMIO 空间 | LoadUnits、StoreUnits、AtomicsUnit |
| Memblock | | |
| 静态检查 | | |

| PMPChecker | | |
|------------|------------------------|---|
| 模块 | 需要返回的信息 | 去向 |
| | 地址是否可缓存 | DTLB |
| | 地址是否为原子性的 | DTLB |
| | 地址是否可执行 | DTLB |
| | 地址是否可写 | DTLB |
| | 地址是否可读 | DTLB |
| L2 TLB | | |
| | 是否出现 load access fault | Page Table Walker、Last Level Page Table Walker、Hypervisor |
| | 地址是否属于 MMIO 空间 | Page Table Walker、Last Level Page Table Walker、Hypervisor |
| | | Page Table Walker |

18.5.2.7 异常处理

PMP 和 PMA 检查可能产生的异常包括: inst access fault (ITLB)、load access fault (LoadUnits、L2 TLB)、store access fault (StoreUnits、AtomicsUnit)。对于 ITLB 和 DTLB 产生的异常，会根据请求来源分别交付给发送物理地址查询的模块进行处理，ITLB 会交付给 Icache 或 IFU；DTLB 会交付给 LoadUnits、StoreUnits 或 AtomicsUnit 进行处理。

由于 Page Table Walker 或 Last Level Page Table Walker 或 Hypervisor Page Table Walker 在访问内存之前需要对访问的物理地址进行 PMP 和 PMA 检查，因此 L2 TLB 可能产生 access fault。L2 TLB 并不会直接将产生的 access fault 进行处理，而是会将该信息返回给 L1 TLB。L1 TLB 在查询发现出现 access fault 后，会根据请求的 cmd，产生 inst access fault、load access fault 或 store access fault；并根据请求来源交付给各模块处理。

可能产生的异常以及 MMU 模块的处理流程如表 18.27 所示：

表 18.27: PMP 和 PMA 检查可能产生的异常以及处理流程

| 模块 | 可能产生的异常 | 处理流程 |
|--------|-----------------------|--|
| ITLB | 产生 inst access fault | 根据请求来源，分别交付给 Icache 或 IFU 处理 |
| DTLB | 产生 load access fault | 交付给 LoadUnits 进行处理 |
| | 产生 store access fault | 根据请求来源，分别交付给 StoreUnits 或 AtomicsUnit 处理 |
| L2 TLB | 产生 access fault | 交付给 L1 TLB，L1 TLB 根据请求来源交付处理 |

18.5.2.8 检查规则

香山昆明湖架构 PMP 和 PMA 的检查规则遵循 RV 手册中的 PMP 和 PMA 部分，在这里只介绍匹配模式。通过 PMP 或 PMA 配置寄存器的 A 位以及 PMP 或 PMA 地址寄存器可以共同决定某条 PMP 或 PMA 项控制的物理地址范围。为支持 DTLB 的静态检查（参见 5.4.2.4 节），需要提升 PMP 和 PMA 的粒度为 4KB，因此某条 PMP 或 PMA 项控制的物理地址范围最小为 4KB。

配置寄存器 A 位对应的匹配模式如下，A 位为 0、1、2、3 分别对应 OFF、TOR、NA4、NAPOT 模式。

- A 为 0, OFF 模式：该 PMP 或 PMA 条目被禁用，不匹配地址；
- A 为 1, TOR 模式 (Top of range)：匹配上一条 PMP 或 PMA 项的地址寄存器至该 PMP 或 PMA 项地址寄存器之间的地址；
- A 为 2, NA4 模式 (Naturally Aligned Four-byte regions)：香山的昆明湖架构不支持 NA4 模式；
- A 为 3, NAPOT 模式 (Naturally Aligned Power-of-two regions)：从 PMP 或 PMA 项地址寄存器的低位开始寻找连续 1 的个数，设 PMP 或 PMA 项地址寄存器为 $ADDR=yyy\dots111$ (有 x 个 1)，则匹配的地址为从 $yyy\dots000$ ($ADDR \gg 2$ 位) 开始的 2^{x+3} 位。由于香山的昆明湖架构规定 PMP 或 PMA 检查的最小粒度为 4KB，因此匹配的地址范围最小为 4KB。

为便于地址匹配，因此分布式 PMP 和 PMA 需要向 PMPChecker 发送 mask 信号。mask 的形式是低 i 位为 1，高位为 0，i 的个数为 $\log_2(\text{pmp 条目匹配的地址空间})$ ，在 PMP 和 PMA 条目更新时会同时更新 mask 值。香山的昆明湖架构对 PMP 和 PMA 支持的最小粒度为 4KB，因此 mask 信号的低 12 位一定为 1。

例如，某 pmp 项的 pmpaddr 为 $16'b1111_0000_0000_0000$ ，由于香山的昆明湖架构对 PMP 和 PMA 支持的最小粒度为 4KB，因此 napot 模式匹配的地址范围大小为 2^{12} B 即 4 KB，mask 信号的值为 $18'hffff$ 。

又例如，某 pmp 项的 pmpaddr 为 $16'b1011_1111_1111_1111$ ，因此 napot 模式匹配的地址范围大小为 2^{17} B 即 128KB，mask 信号的值为 $18'h1ffff$ 。

18.5.3 整体框图

PMP 模块和 PMA 模块的整体框图分别如图 18.37 和图 18.38 所示。在 CSR Unit 负责响应 CSRRW 等 CSR 指令对这些 PMP 和 PMA 寄存器的读写；在 Frontend、Memblock 和 L2 TLB 中包含这些 PMP 和 PMA 寄存器的备份，负责地址检查，通过拉取 CSR 的写信号可以保证这些寄存器的内容一致性。

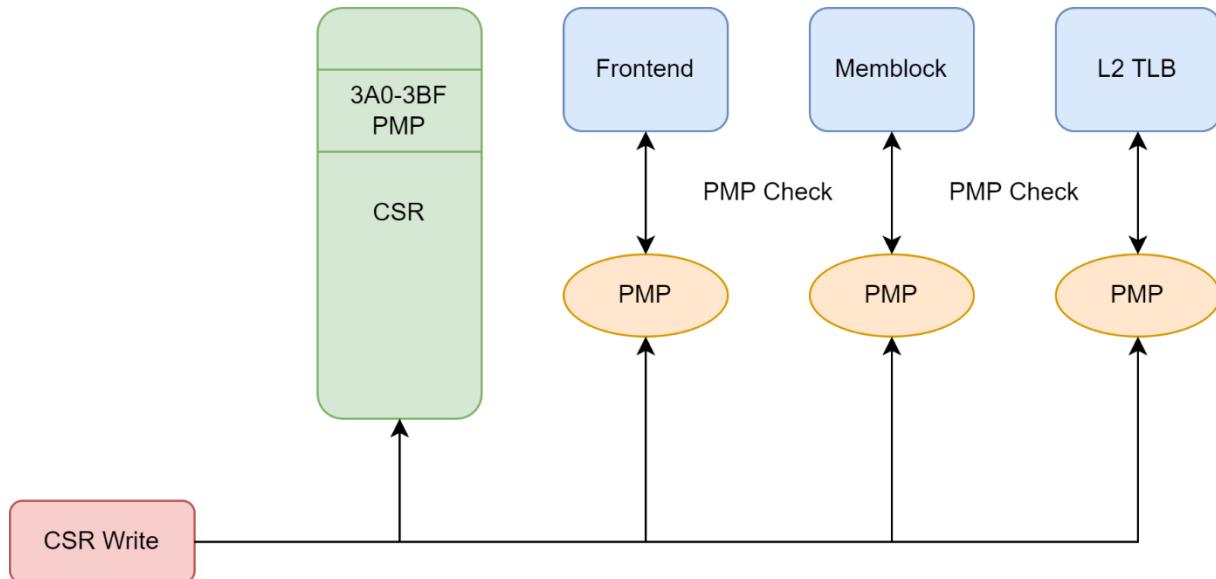


图 18.37: PMP 模块整体框图

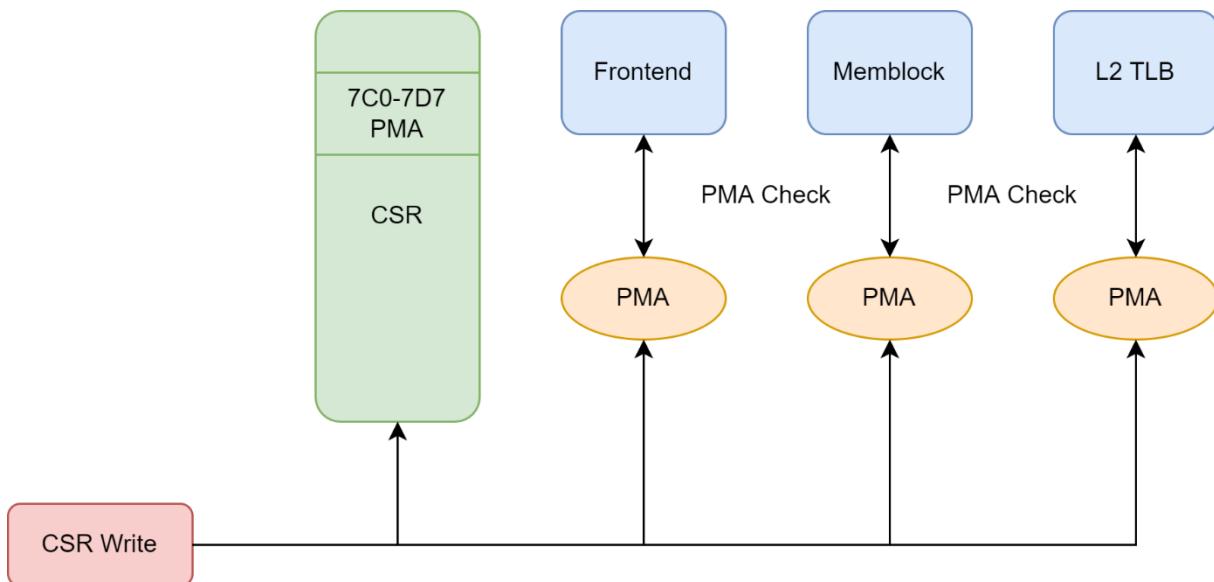


图 18.38: PMA 模块整体框图

18.5.4 接口列表

参见接口列表文档。

18.5.5 接口时序

对于 ITLB 和 L2 TLB, PMP 和 PMA 检查需要当拍返回结果；对于 DTLB, PMP 和 PMA 检查会在下一拍返回结果。ITLB 和 L2 TLB PMP 模块接口时序如图 18.39 所示。

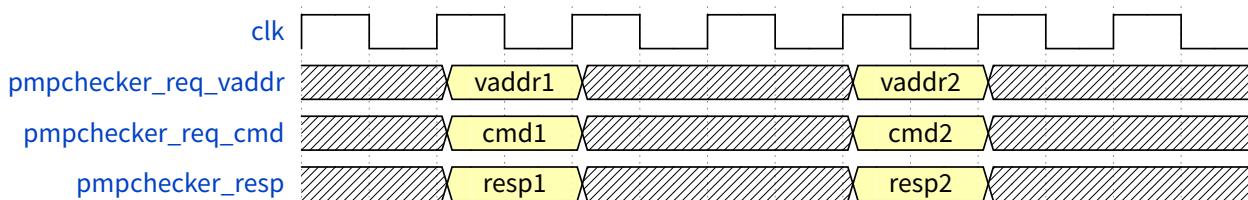


图 18.39: ITLB 和 L2 TLB PMP 模块的接口时序

DTLB PMP 模块接口时序如图 18.40 所示，静态检查和动态检查的接口时序相同。

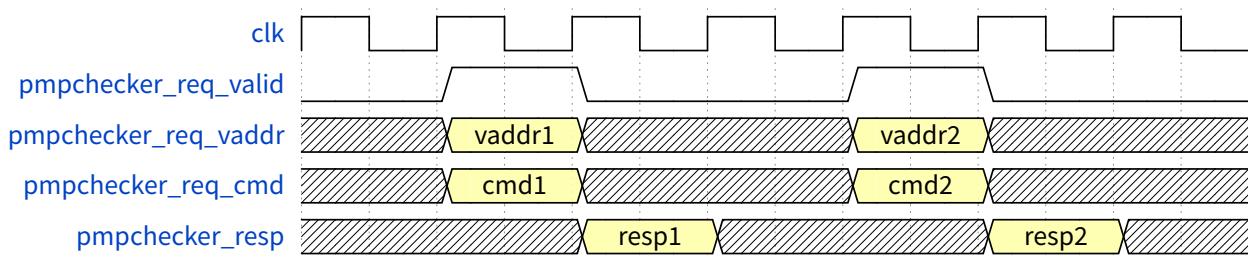


图 18.40: DTLB PMP 模块的接口时序

19 预取

第四部分

缓存子系统

20 二级缓存

20.1 二级缓存 CoupledL2

20.1.1 子模块列表

CoupledL2 顶层分为（默认）4个 Slice、MMIOBridge、预取器。

预取器包括 L2 本地的预取器 Best-Offset Prefetch (BOP), 和 L1 DCache 预取接收器, 用于接收在 DCache 训练但是需要取到 L2 的预取请求。

MMIOBridge 即 MMIO 请求转接桥, 用于将 TileLink 总线的 MMIO 请求转换为 CHI 请求, 并和 cacheable 地址空间的 CHI 请求做仲裁, 从统一的 CHI 接口接入互联总线。

CoupledL2 的 4 个 Slice 按照地址低位划分, 不同地址的请求或预取地址会被分散到不同的 Slice。

每个 Slice 中的子模块列表如下:

| 子模块 | 描述 |
|---------------|--|
| SinkA | 上游 TileLink 总线 A 通道控制器 |
| SinkC | 上游 TileLink 总线 C 通道控制器 |
| GrantBuffer | 上游 TileLink 总线 D/E 通道控制器 |
| TXREQ | 下游 CHI 总线 TXREQ 通道控制器 |
| TXDAT | 下游 CHI 总线 TXDAT 通道控制器 |
| TXRSP | 下游 CHI 总线 TXRSP 通道控制器 |
| RXSNP | 下游 CHI 总线 RXSNP 通道控制器 |
| RXDAT | 下游 CHI 总线 RXDAT 通道控制器 |
| RXRSP | 下游 CHI 总线 RXRSP 通道控制器 |
| Directory | 目录, 保存元数据信息的 SRAM |
| DataStorage | 数据 SRAM |
| RefillBuffer | 回填数据寄存器堆 |
| ReleaseBuffer | 释放数据寄存器堆 |
| RequestBuffer | A 通道请求缓冲 |
| RequestArb | 请求仲裁器, 主流水线 s0~s2 流水级 |
| MainPipe | 主流流水线 s3~s5 流水级 |
| MSHRCtl | MSHR (Miss Status Handling Registers) 控制模块, 默认包含 16 项 MSHR |

20.1.2 设计规格

- 与上游 L1Cache / PTW 互联采用 TileLink 总线协议
- 与下游 HN-F 采用 CHI 总线协议，支持 B/C/E.b 3 个 CHI 总线版本（默认 E.b）
- 支持如下 CHI Read 事务：
 - ReadNoSnp (B/C/E.b) (仅用于 MMIO 与 Uncache 请求)
 - ReadNotSharedDirty (B/C/E.b)
 - ReadUnique (B/C/E.b)
- 支持如下 CHI Dataless 事务：
 - MakeUnique (B/C/E.b)
 - Evict (B/C/E.b)
 - CleanShared (B/C/E.b)
 - CleanInvalid (B/C/E.b)
 - MakeInvalid (B/C/E.b)
- 支持如下 CHI Write 事务：
 - WriteNoSnpPtl (B/C/E.b) (仅用于 MMIO 与 Uncache 请求)
 - WriteBackFull (B/C/E.b)
 - WriteCleanFull (B/C/E.b)
 - WriteEvictOrEvict (E.b)
- 支持如下 CHI Snoop 事务：
 - SnpOnceFwd (B/C/E.b)
 - SnpOnce (B/C/E.b)
 - SnpStashUnique (B/C/E.b)
 - SnpStashShared (B/C/E.b)
 - SnpCleanFwd (B/C/E.b)
 - SnpClean (B/C/E.b)
 - SnpNotSharedDirtyFwd (B/C/E.b)
 - SnpNotSharedDirty (B/C/E.b)
 - SnpSharedFwd (B/C/E.b)
 - SnpShared (B/C/E.b)
 - SnpUniqueFwd (B/C/E.b)
 - SnpUnique (B/C/E.b)
 - SnpUniqueStash (B/C/E.b)
 - SnpCleanShared (B/C/E.b)
 - SnpCleanInvalid (B/C/E.b)
 - SnpMakeInvalid (B/C/E.b)
 - SnpMakeInvalidStash (B/C/E.b)
 - SnpQuery (E.b)

- 1MB 容量，8 路组相联结构，按照地址低位划分为 4 个 Slice
- 缓存行大小 64B，总线数据位宽 32B，一次完整的缓存行传输需要 2 个 beat 的数据传输
- 采用类 MESI 的缓存一致性协议
- 和 DCache 之间采用严格包含策略，和 ICache / PTW 之间采用不严格包含策略
- 采用非阻塞主流水线结构
- 最高访问并行度为 4×16 (每个 Slice 包含 16 项 MSHR，共 4 个 Slice)，每个 Slice 中至多 15 项 MSHR 可用于 L1Cache / PTW 的访问
- 支持相同 set 请求的并行访问
- 支持在收到 L2 Cache 缺失的重填数据后再进行替换路的选取和替换
- 支持访存请求和预取请求的融合
- 支持产生 L2 Refill Hint 信号用于 Load 指令的提前唤醒
- 支持 BOP 预取器
- 支持 L1 训练并回填到 L2 的预取请求的处理
- 支持 DRRIP / PLRU 等替换算法，默认 DRRIP
- 支持硬件处理 Cache 别名
- 支持 MMIO 请求的处理，MMIO 请求在 CoupledL2 中由 TileLink 总线转换为 CHI 总线，并和 4 个 Slice 发起的 cacheable 请求做仲裁

20.1.3 功能描述

CoupledL2 接收香山核 DCache / ICache / PTW 发送的 TileLink 回填、替换请求，完成相应数据块的转移与一致性状态转移，同时在片上网络中作为 RN-F，维护香山核在片上互联系统中的缓存一致性。

CoupledL2 模块通过上游 TileLink 通道控制器 (SinkA / SinkC) 接收，将其转化为内部请求。请求通过请求仲裁进入主流水线，读取目录获取缓存块的状态，根据缓存块状态和请求信息判断是否能够处理：

- 若本层缓存可以直接处理该请求，则继续在主流水线中进行读数据、更新目录等操作，然后进入 GrantBuffer，转化为 TileLink 总线响应。
- 若需要和其它缓存进行交互才能处理该请求，则为其配一个 MSHR。MSHR 根据需求向上下层 Cache 发送子请求，等待收到响应并满足释放条件后，再释放任务重新进入主流水线，进行读缓冲区、读写数据、更新目录等操作，然后进入通道控制器模块，转化为 TileLink 总线响应。

当一个请求所需的全部操作在 MSHR 中完成时，MSHR 被释放，等待接收新的请求。

20.1.3.1 采用类 MESI 的缓存一致性协议

香山核的缓存子系统遵循 TileLink 一致性树的规则。CoupledL2 中的缓存行状态包括 N(Nothing)、B/Branch)、T/Trunk)、TT(Tip) 4 个状态：

- N: 无效
- B: 只读权限
- T: 当前核具有写权限，但是写权限位于上游 cache，当前 cache 层次不可读不可写
- TT: 可读可写

一致性树按照内存、L3、L2、L1 的顺序自下而上生长，内存作为根节点拥有可读可写的权限，子节点的权限都不能超过父节点的权限。其中 TT 代表拥有 T 权限的最上层子节点（也是 T 权限树的叶子节点），说明该节点上层只有 N 或 B 权限，相反 T 权限而不是 TT 权限的节点代表上层一定还有 T/TT 权限节点。详细规则请参考 TileLink 手册。

20.1.3.2 采用目录记录缓存行信息

CoupledL2 是基于目录结构的 Inclusive Cache (此处所指的“目录”是广义的，包含元数据和 Tag)。元数据包含：状态位 state / 脏位 dirty / 是否在上层缓存 clients / 在上层的别名 alias / 是否是预取上来的 prefetch / 来自哪个预取器 prefetchSrc / 是否被访问过 accessed。

在流水线 s1 级 RequestArb 会向目录发起读请求，读取 Tag Array 判断是否命中。如果命中则选择命中路，如果不命中则根据替换算法选择一个替换路，然后将选中路的元数据信息返回给 s3 级 MainPipe。

20.1.3.3 采用非阻塞流水线结构

CoupledL2 采用主流水线架构，来自各通道的请求经过仲裁进入主流水线，进行目录操作，然后根据请求信息和目录结果安排响应的操作：

20.1.3.3.1 Acquire 请求处理流程

如图 20.1 所示。

20.1.3.3.2 Snoop 请求处理流程

如图 20.2 所示。

20.1.3.3.3 Release 请求处理流程

Release 请求处理流程如下：

1. 从 SinkC 接收来自 L1 DCache 的 Release 请求，并转化为内部请求
2. s1 Release 请求进入流水线，并查询目录
3. s3 得到查目录的结果（由于是 L1 DCache 和 L2 是严格包含关系所以 Release 一定会命中）；s3 写目录，如果有脏数据需要在 s3 写入 DataStorage
4. s3 生成 ReleaseAck 响应，在 s3~s5 的某一流水级离开流水线，进入 GrantBuffer 将 ReleaseAck 返回给 L1

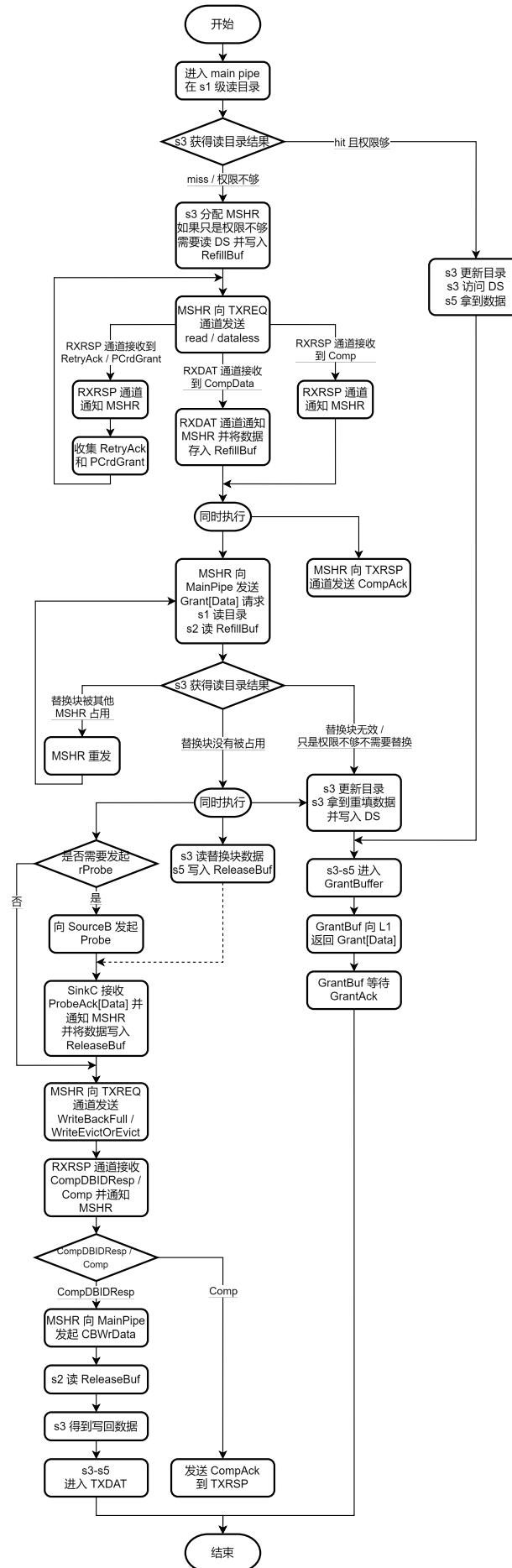


图 20.1: Acquire 请求处理流程

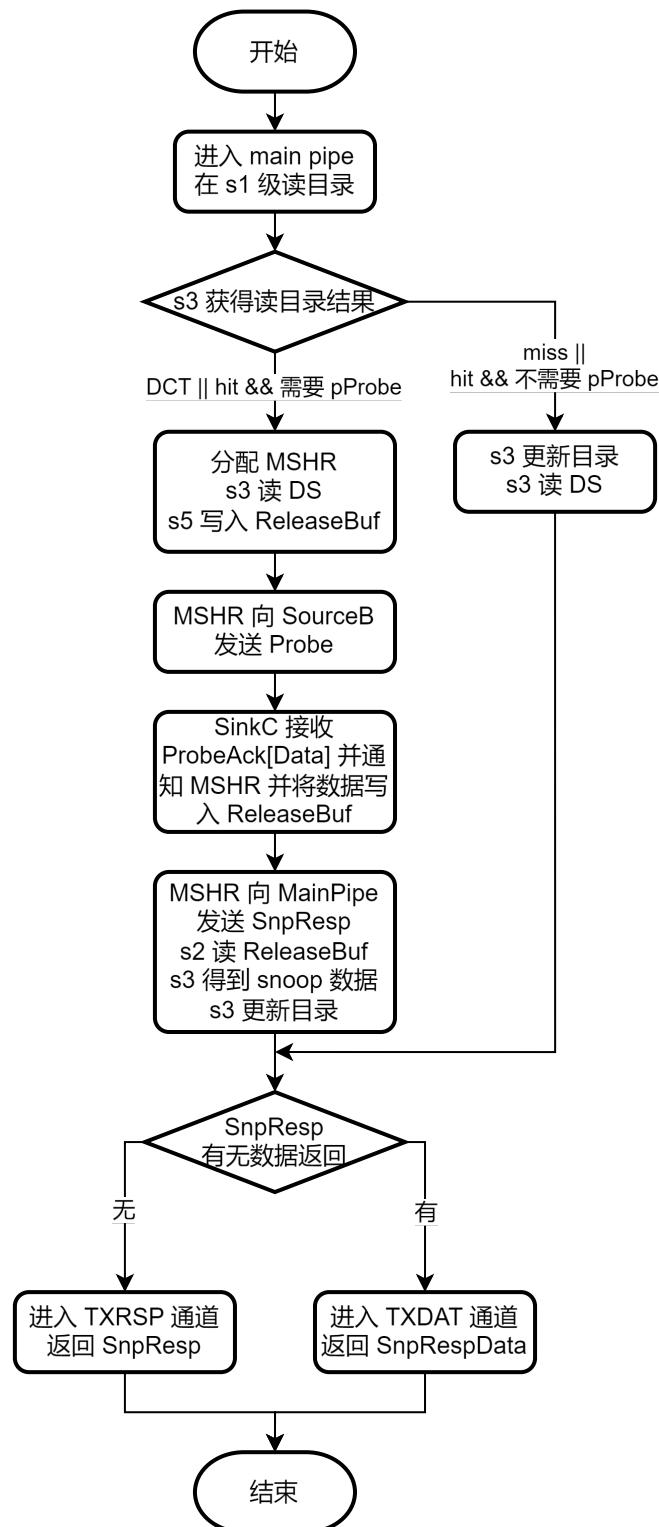


图 20.2: Snoop 请求处理流程

20.1.3.4 收到重填数据后再进行替换路选取和替换

缓存在收到新请求但是 set 已满的时候，按照常规逻辑，需要先选择一个替换路，将其写入下级缓存，从而为即将重填的缺失数据空出位置，然后等待新数据块从下级缓存重填上来，再将其写入。但是这种方式会存在一些问题：

1. 一方面，从下级缓存重填往往需要较长的延迟（几十拍到上千拍），在这段时间内，旧的数据块已经被释放掉，新的数据块尚未收到，所以这个位置实际上是没有有效数据的，从而造成了缓存资源的空闲和浪费，降低了缓存的有效容量。
2. 另一方面，如果在这段时间内，上层缓存正好又要访问被替换的数据块，因为此时数据块已经被释放，所以只能再次向下层缓存获取，从而使得访问延迟大大增加。

CoupledL2 将替换路的选择和替换数据的释放延后到收到重填数据时。具体地，在请求进入缓存时，需要读取目录信息来判断是否命中。如果命中，则读取数据并返回（标准流程）。如果缺失，CoupledL2 不会根据读目录结果来选择一个替换块并安排替换块的释放，而只是为其分配一项 MSHR，并向下游缓存发请求获取数据。等待下游返回重填数据后，让 MSHR 任务再次读目录，此时再选出替换块，从数据存储单元读出替换块的数据，向下游缓存释放。最后再将新数据块写入存储单元。

由于与 DataStorage 交互只在 MainPipe 的 s3 级，且 DataStorage 的 SRAM 是单端口的，因此我们无法利用一个 MSHR Task 同时完成 (1) 将被替换数据块的内容读出并向下游缓存释放，和 (2) 将新数据块写入。所以这 2 步操作需要分成 MSHR Refill 和 MSHR Release 两个任务，Refill 先于 Release 发出。又基于另外 2 点考虑：(1) 读旧数据必须要早于写新数据，(2) 必须尽快向 L1 返回数据，所以我们为两个 MSHR Task 分别安排如下任务：

- MSHR Refill：读 RefillBuffer 获取重填数据反馈给 L1；读 DataStorage 获取旧数据存入 ReleaseBuf；更新目录为新数据的元数据
- MSHR Release：读 ReleaseBuf 并将数据向 L3 释放；读 RefillBuffer 将重填数据写入 DataStorage

20.1.3.5 支持同 Set 请求的并行访问

CoupledL2 支持多个相同 Set 请求的并行访问。对于多个相同 Set 的请求，这些请求在收到重填数据之前是不需要选择替换路的，因此在收到冲天数据之前都是可以并行访问的。收到重填数据后，MSHR 开始选择替换路，并将替换块写入下级缓存。目录在选择替换路时需要确保不会选到正在替换的路，从而保证相同 Set 的多个请求一定会选到不同的替换路。

20.1.3.6 Load 指令提前唤醒

CoupledL2 每一次向 L1 DCache 重填数据时，都会在 GrantData 发出的前 3 拍发送 Refill Hint 信号送到核内的 LoadQueue。LoadQueueReplay 收到唤醒信号会立刻唤醒需要重发的 Load 进入 LoadUnit，Load 指令会在 LoadUnit 的 s2/s3 流水级收到重填的数据，从而减低 Load 在 L1 缺失时的访问延迟。

20.1.3.7 支持硬件预取

CoupledL2 的硬件预取器会同时接收 BOP 预取请求和 L1 DCache 发送的预取请求，并将这些请求送进预取队列（PrefetchQueue）。预取队列满时会自动丢弃队头最老的预取请求，让更新的预取请求入队，从而保证预取时效性。

20.1.3.8 支持请求融合

实验观察发现，L2 Cache 中存在占比较大的不及时预取，即预取器虽然预测到了未来需要的数据，但请求发送较晚，当预取导致的 cache miss 还在 MSHR 中等待下层缓存数据返回时，对同地址的 Acquire 请求已经到达 L2。为了不让此类 Acquire 请求在 RequestBuffer 入口被阻塞，导致 L2 入口被占满，后续请求无法进入，当前 L2 设计了一套合并不及时的 Prefetch 与后续同地址 Acquire 的请求融合机制。请求融合功能实现如下：

1. 在 SinkA 通道的入口 RequestBuffer 中判断来自 L1 的 A 请求是否满足合并条件，条件为：新请求为 Acquire，且在 MSHRs 中存在 miss 请求为 Prefetch，并与 Acquire 地址相同
2. 若满足合并条件，则新请求不需要进入队列被阻塞，而是直接进入同地址 Prefetch 对应的 MSHR 项中，并对该项标记 mergeA，新增加一系列请求状态信息，使其包含两个请求的内容
3. 当目标数据从 L3 返回后，MSHR 项被唤醒，发送任务到主流水线进行处理。在主流水线中选择替换路并回填新数据，而数据块的 meta 则更新为 Acquire 请求处理完成后的状态，同时该请求还会将信息传入预取器作为训练
4. 在处理请求响应时，这个合并请求会从主流水线进入 GrantBuffer，对于 Prefetch 请求，L2 返回预取响应；而对于 Acquire 请求，L2 通过 grantQueue 队列对发出 Acquire 的上游节点返回数据和响应

20.1.3.9 支持硬件处理 Cache 别名

香山核的 L1 Cache 均采用 VIPT 的索引方式，其中 DCache 为 64KB 的 4 路组相联结构，用于访问 DCache 的索引和块内偏移超出了 4KB 页的页偏移，由此引入 Cache 别名问题：如图 20.3 所示，当两个虚拟页映射到同一个物理页时，两个虚拟页的别名位（索引超出 4KB 页偏移的部分）有较大概率是不一样的，如果不做额外处理，通过 VIPT 索引后两个虚页中的缓存块会位于 DCache 的不同 set，导致同一个物理地址在 DCache 中缓存了两份，如果 DCache 不做额外处理的话会引入缓存一致性错误。

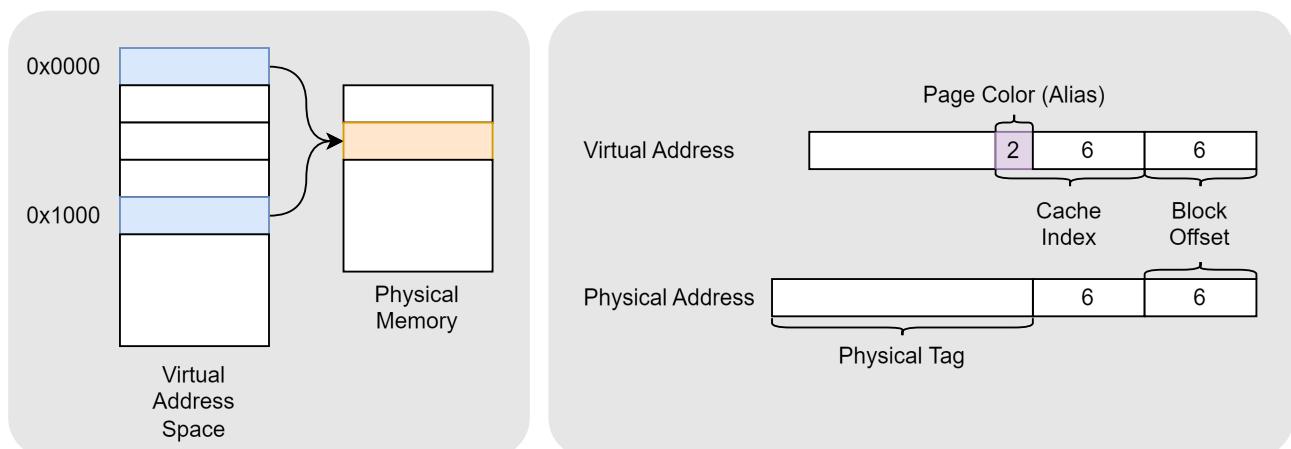


图 20.3: Cache 别名原理示意图

香山核是通过 CoupledL2 以硬件方式解决 Cache 别名问题的。具体解决方式是由 CoupledL2 记录上层数据的别名位，保证一个物理地址缓存块在 L1 DCache 中最多只有一种别名位。当上层缓存发送 Acquire 请求时会带上别名位，L2 Cache 会检查目录，如果命中但是别名不一致，会向上层缓存 Probe 之前记录的别名位，并将 Acquire 的别名位写入目录。

20.1.4 总体设计

20.1.4.1 整体框图

XSTile (包括香山核和 CoupledL2) 结构框图如图 20.4 所示。

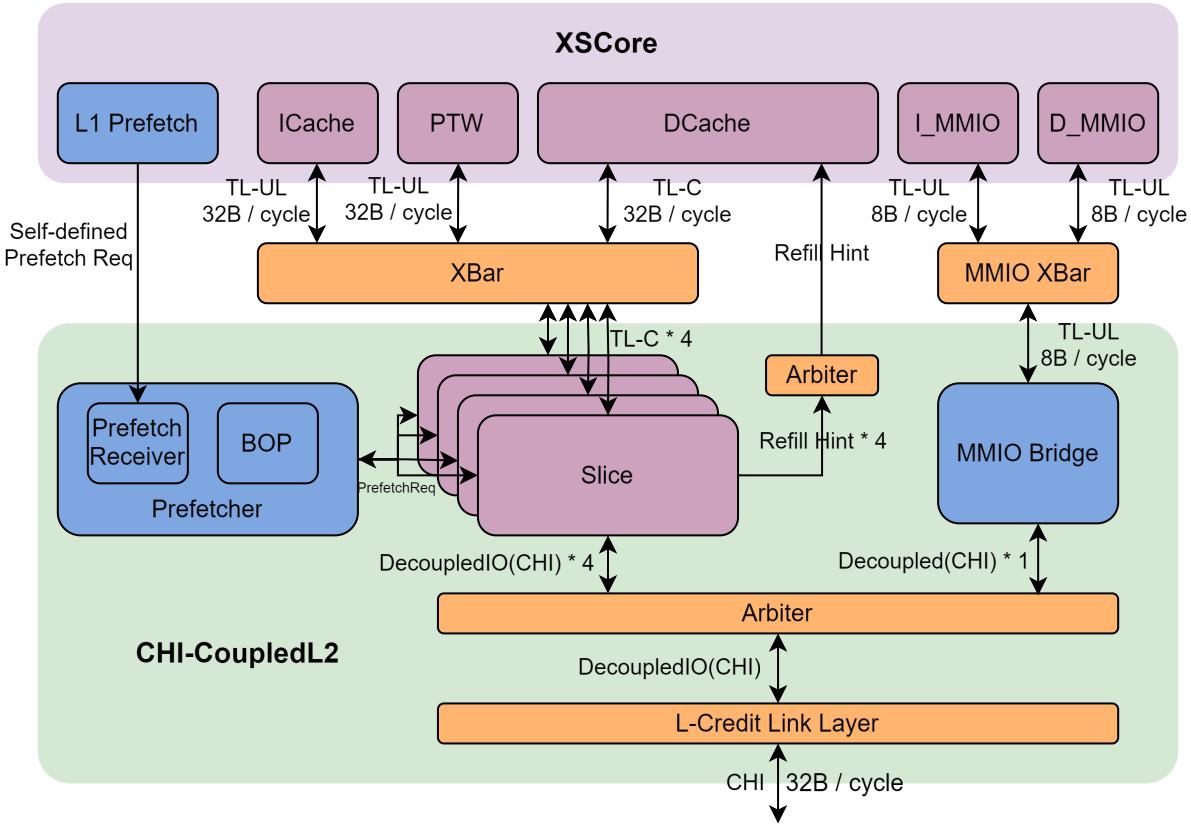


图 20.4: XSTile 结构框图

CoupledL2 微结构框图如图 20.5 所示。

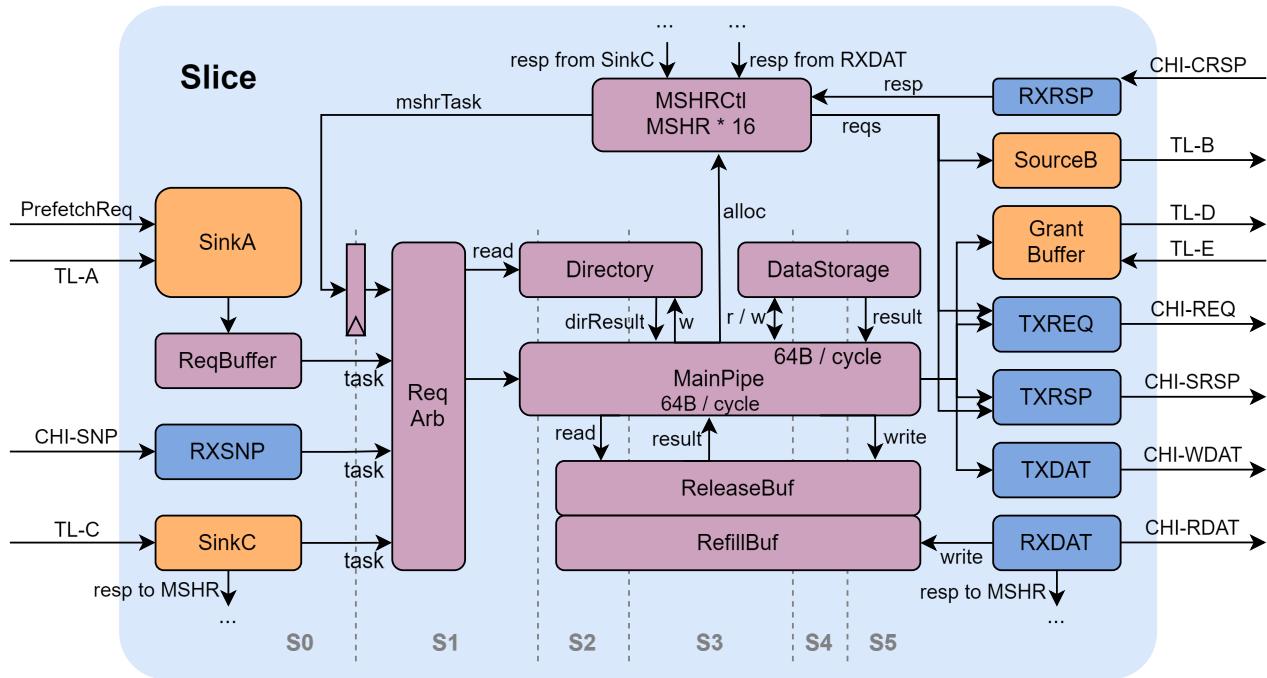


图 20.5: CoupledL2 微结构图

20.2 A 通道请求缓冲 RequestBuffer

20.2.1 功能描述

- Request Buffer 用来缓冲那些暂时需要阻塞的 A 通道请求，同时让满足释放条件/不需要阻塞的 A 通道请求先进入主流水线。
- Request Buffer 可以防止需要阻塞的 A 请求堵住流水线入口，从而避免对后续请求的影响，提高了缓存的处理效率。
- 如果有新到达的 Acquire 与 MSHR 中正在处理的预取请求地址相同，则可以进行请求融合，将 Acquire 的信息直接传到对应 MSHR 中，让 MSHR 处理完成后同时回复 L1 Acquire，从而加快了 Acquire 的处理流程，并减少了对 ReqBuf 和 MSHR 的占用。

20.2.1.1 特性 1：请求融合

当 RequestBuffer 接收到的 Acquire 请求和某一项 MSHR 中的预取请求有相同地址时，RequestBuffer 会将合并请求 (aMergeTask) 发送到相应的 MSHR entry，该项 MSHR 会被标注 mergeA 并更新 MSHR 的相关域。

20.2.1.2 特性 2：请求接收条件

RequestBuffer 入口的请求在哪些情况下允许被接收：

- RequestBuffer 没有满
- RequestBuffer 满了，但是 Acquire 请求可以和前面的预取请求融合
- RequestBuffer 满了，但是该请求是预取请求，并且已经前面已经有一条 Acquire/Prefetch 请求正在被 MSHR 处理

20.2.1.3 特性 3: RequestBuffer 的分配

哪些请求会分配 RequestBuffer: - RequestBuffer 没有满 - 不能直接 flow 进入流水线 (即和 MainPipe 或者某项 MSHR 地址冲突) 或者 chosenQ 也准备发射 - 不能做请求融合

20.2.1.4 特性 4: RequestBuffer 项中的域

- Rdy: 是否准备好被发射/出队
- Task: 请求本身的信息
- WaitMP: 被 MainPipe 哪几级流水线阻塞
- WaitMS: 被哪几项 MSHR 阻塞

20.2.1.5 特性 5: RequestBuffer 如何更新和发射

- WaitMP(4bit): 因为 MainPipe 是非阻塞流水线, 所以 waitMP 每周期会右移一位, 同时每拍会检查 s1 有无新的地址冲突的请求 [3] s1, same set conflict [2] s2, same set conflict [1] s3, same set conflict [0] reserved
- WaitMS(16bit): MSHR 被释放的前一拍会将 waitMS 相应的 bit 复位; 同时有新的 MSHR 项被分配时会检查有无地址冲突 (same set and tag), 如果有需要将 waitMS 相应 bit 置位 onehot 编码每个 bit 代表一个 MSHR
- noFreeWay: 由于相同 set 有可能产生替换, 所以 [MSHR 中 same set 的数目 + 流水线上 S2/S3 的 same set 的数目 \geq l2 way 时, 表示现在相同 set 的全部 way 都有可能被替换。这时就阻塞住 RequestBuf 进入流水线。s2 + s3 + MSHR \geq ways(L2)
- Rdy 条件: 在满足下面所有条件时, rdy 为高表示可以被发射到流水线上进入 RequestArbiter waitMP + waitMS 全部被清零 noFreeway 为低流水线 s1 级即将进入流水线的 A/B 通道请求没有 set 冲突

20.2.2 整体框图

20.3 请求仲裁器与主流水线

请求仲裁器与访存流水线组成了 CoupledL2 的整体五级流水线, 按顺序简称为第一级 s1、第二级 s2、第三级 s3、第四级 s4、第五级 s5。其中请求仲裁器 ReqArbiter 主要组成 s1、s2, 主流水线 MainPipe 主要组成 s3、s4、s5。

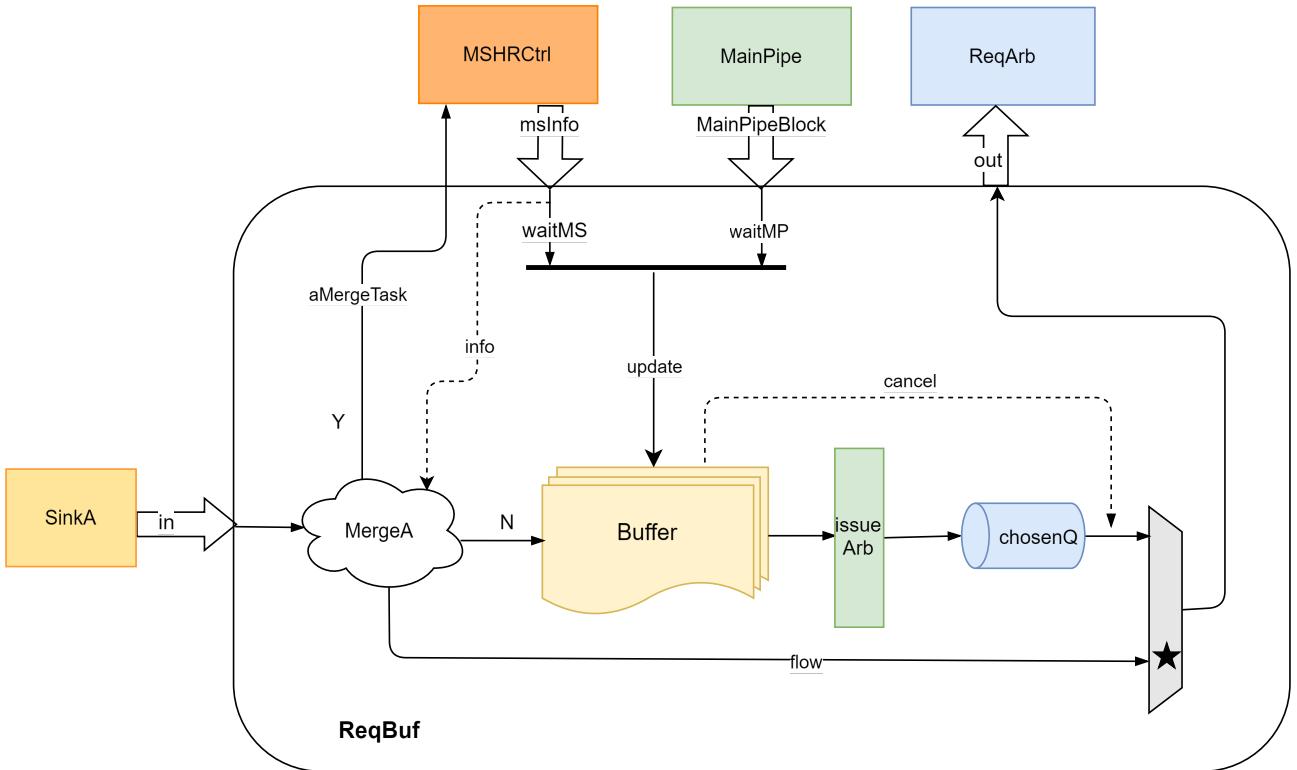


图 20.6: RequestBuf

20.3.1 S0 流水级

s0 仅位于请求仲裁器 ReqArbiter 内，且不算作单独的一个流水级。s0 仅用于产生给每一项 MSHR 的任务反压信号，在以下情况下 ReqArbiter 会阻止来自 MSHR 的任务离开 MSHR 并进入流水线：

- 上一周期存在需要读取 Directory 的 MSHR 任务且被阻塞
- 存在来自于 GrantBuffer 的阻塞信号
- 存在来自于上游 TileLink C 通道的阻塞信号
- 存在来自于下游 TXDAT 通道的阻塞信号
- 存在来自于下游 TXRSP 通道的阻塞信号
- 存在来自于下游 TXREQ 通道的阻塞信号

20.3.2 S1 流水级

s1 仅位于请求仲裁器 ReqArbiter 内。

在 s1，会对以下几种请求来源进行仲裁：

- MSHR
- 上游 TileLink C 通道
- 上游 TileLink B 通道
- 上游 TileLink A 通道

以上列表中，位于上方的请求来源拥有最高的优先级，在他们同时进入 ReqArbiter 的 s1 时，会选择其中优先级最高的一项进行握手，并阻塞其它的任务来源。即其中 MSHR 任务优先级最高，其次是上游 TileLink C 通道、上游 TileLink B 通道、上游 TileLink A 通道。

在 s_1 , ReqArbiter 还需要考虑来自于 MainPipe 的阻塞信号。并且此时在 s_2 就绪时, 请求才可以离开 s_1 , 否则请求被阻塞并寄存在 s_1 。

在完成仲裁后, 在 s_1 向 Directory 发送读取请求。

20.3.3 S2 流水级

s_2 位于请求仲裁器 ReqArbiter 与主流水线 MainPipe 内。

由于 CoupledL2 的 SRAM 在频率上有限制, 所以采用了多周期路径 MCP2 (Multi-Cycle Path 2), 意味着 SRAM 的单次读写请求都需要至少持续两个周期。故在 s_2 , ReqArbiter 会将所有的背靠背请求阻塞一拍, 以使得请求在 MainPipe 上的保持时间以及请求间隔符合 MCP2 的要求。

ReqArbiter 于 s_2 决定是否读取 ReleaseBuffer 或 RefillBuffer。并于 s_2 向 ReleaseBuffer 或 RefillBuffer 发送读请求。

于以下情况之一, ReqArbiter 会于 s_2 向 RefillBuffer 发送读请求:

1. 该任务是由替换任务引起的下游缓存行写回、踢出任务 (此时写回的数据已不再被需要, 替换读取到的数据在此时被写入 DataStorage)
2. 该任务是上游 TileLink A 通道请求, 但不使用上游 Probe 回复的数据 (若使用上游 Probe 回复的数据, 则应当读取 ReleaseBuffer)

于以下情况之一, ReqArbiter 会于 s_2 向 ReleaseBuffer 发送读请求:

1. 该任务是 MSHR 任务, 且下游请求需要读取上游 Probe 回复的数据
2. 该任务是 MSHR 任务, 且上游 TileLink A 通道请求需要使用上游 Probe 回复的数据
3. 该任务不是 MSHR 任务, 且下游 Snoop 与下游写回请求任务发生了嵌套

ReqArbiter 于 s_2 时将任务送入 MainPipe。

MainPipe 会于 s_2 生成对于 s_1 的阻塞信号并送回 ReqArbiter 与 RequestBuffer。MainPipe 需要于如下情况向各个组件、通道发送阻塞信号:

- 在任务到达 s_2 时若无法确定其一定不会对 Directory 进行写操作, 则向 RequestBuffer 发送阻塞同 Set 请求的信号
- 在任务到达 s_2 时若无法确定其一定不会对 Directory 进行写操作, 则向 ReqArbiter 发送阻塞同 Set 的 MSHR 请求的信号
- 在任务到达 s_2 时若无法确定其一定不会对 Directory 进行写操作, 则向 ReqArbiter 发送阻塞同 Set 的上游 TileLink C 通道请求的信号
- 在任务到达 s_2 (以及 s_3 、 s_4 、 s_5 时, 即包括所有仍在 MainPipe 上的任务, 在后续章节中不再赘述) 时, 向 ReqArbiter 发送阻塞同地址的下游 RXSNP 通道请求的信号

20.3.4 S3 流水级

s_3 仅位于主流水线 MainPipe 内。大部分请求的判断、分发逻辑, 以及与各个其它模块的交互都位于 s_3 阶段。

20.3.4.1 缓存行状态收集

在 s_1 由 ReqArbiter 向 Directory 发出的读取请求在 s_3 可以得到读取结果。若来自下游 RXSNP 通道的请求出现了与 MSHR 的嵌套, 即下游 Snoop 请求的地址与某一项未完成的 MSHR 地址相同, 则会使用该项 MSHR 中的缓存行状态覆盖 Directory 的读取结果。

20.3.4.2 MSHR 分配

MainPipe 在 s3 阶段满足以下条件之一时会分配 MSHR:

1. 任务来自于上游 TileLink A 通道

- Acquire*、Hint、Get 请求未命中缓存行
- Acquire* toT 命中 BRANCH 状态的缓存行
- CBO* 类 CMO 请求
- 别名 (Alias) 替换请求
- 任意需要向上游发送 Probe 请求的任务
 - Get 请求命中 TRUNK 状态的缓存行且其存在于上游 L1
 - CBOClean 请求命中 TRUNK 状态的缓存行且其存在于上游 L1
 - CBOFlush 请求命中的缓存行存在于上游 L1
 - CBOInval 请求命中的缓存行存在于上游 L1

2. 任务来自于下游 RXSNP 通道

- Snoop 对应类型命中对应缓存行状态
- Forwarding Snoop 且命中缓存行

对于来自下游的非 Forwarding Snoop 类型的 Snoop 请求，需要分配 MSHR 的情况见下表：

| Snoop 请求类型 | 命中状态 | 存在于 L1 |
|---------------------|-------|--------|
| SnpOnce | TRUNK | 是 |
| SnpClean | TRUNK | 是 |
| SnpShared | TRUNK | 是 |
| SnpNotSharedDirty | TRUNK | 是 |
| SnpUnique | - | 是 |
| SnpCleanShared | TRUNK | 是 |
| SnpCleanInvalid | - | 是 |
| SnpMakeInvalid | - | 是 |
| SnpMakeInvalidStash | - | 是 |
| SnpUniqueStash | - | 是 |
| SnpStashUnique | TRUNK | 是 |
| SnpStashShared | TRUNK | 是 |
| SnpQuery | TRUNK | 是 |

20.3.4.3 Directory 写入

MainPipe 在 s3 会按照任务的要求向 Directory 发送写请求。

20.3.4.4 DataStorage 读写

MainPipe 在 s3 会按照人物的要求向 DataStorage 发送读或写请求。

20.3.4.5 请求与消息分发

MainPipe 在 s3 会按照任务的要求向以下通道方向之一发送请求：

- 上游 TileLink D 通道
- 下游 TXREQ 通道
- 下游 TXRSP 通道
- 下游 TXDAT 通道

具体的分发方向由任务本身决定，详见 § 20.6 MSHR。

20.3.4.6 Snoop 请求处理

来自下游的 Snoop 请求可能不分配 MSHR，而直接在 MainPipe 中完成回复动作。其 Snoop 请求的状态转移在 MainPipe 的 s3 决定。在 s3 发生的 Snoop 请求与其相应状态转移如下表：

| Snoop 请求类型 | 起始状态 | 最终状态 | RetToSrc | Snoop 回复 |
|---------------------|-----------|------|----------|-------------------|
| SnpOnce | I | I | X | SnpResp_I |
| | UC | UC | X | SnpRespData_UC |
| | UD | UD | X | SnpRespData_UD_PD |
| | SC | SC | 0 | SnpResp_SC |
| | | | 1 | SnpRespData_SC |
| SnpClean, | I | I | X | SnpResp_I |
| | UC | SC | X | SnpResp_SC |
| SnpShared, | UD | SC | X | SnpRespData_SC_PD |
| | SC | SC | 0 | SnpResp_SC |
| | | | 1 | SnpRespData_SC |
| | SnpUnique | I | X | SnpResp_I |
| | | UC | X | SnpResp_I |
| | | UD | X | SnpRespData_I_PD |
| | | SC | 0 | SnpResp_I |
| | | | 1 | SnpRespData_I |
| SnpCleanShared | I | I | 0 | SnpResp_I |
| | UC | UC | 0 | SnpResp_UC |
| | UD | UC | 0 | SnpRespData_UC_PD |
| | SC | SC | 0 | SnpResp_SC |
| SnpCleanInvalid | I | I | 0 | SnpResp_I |
| | UC | I | 0 | SnpResp_I |
| | UD | I | 0 | SnpRespData_I_PD |
| | SC | I | 0 | SnpResp_I |
| SnpMakeInvalid | - | I | 0 | SnpResp_I |
| SnpMakeInvalidStash | - | I | 0 | SnpResp_I |
| SnpUniqueStash | I | I | 0 | SnpResp_I |
| | UC | I | 0 | SnpResp_I |

| Snoop 请求类型 | 起始状态 | 最终状态 | RetToSrc | Snoop 回复 |
|-----------------------|------|------|----------|----------------------------|
| | UD | I | 0 | SnpRespData_I_PD |
| | SC | I | 0 | SnpResp_I |
| SnpStashUnique, | I | I | 0 | SnpResp_I |
| SnpStashShared | UC | UC | 0 | SnpResp_UC |
| | UD | UD | 0 | SnpResp_UD |
| | SC | SC | 0 | SnpResp_SC |
| SnpOnceFwd | I | I | 0 | SnpResp_I |
| | UC | UC | 0 | SnpResp_UC_Fwded_I |
| | UD | UD | 0 | SnpResp_UD_Fwded_I |
| | SC | SC | 0 | SnpResp_SC_Fwded_I |
| SnpCleanFwd, | I | I | X | SnpResp_I |
| SnpNotSharedDirtyFwd, | UC | SC | 0 | SnpResp_SC_Fwded_SC |
| SnpSharedFwd | | | 1 | SnpRespData_SC_Fwded_SC |
| | UD | SC | X | SnpRespData_SC_PD_Fwded_SC |
| | SC | SC | 0 | SnpResp_SC_Fwded_SC |
| | | | 1 | SnpRespData_SC_Fwded_SC |
| SnpUniqueFwd | I | I | 0 | SnpResp_I |
| | UC | I | 0 | SnpResp_I_Fwded_UC |
| | UD | I | 0 | SnpResp_I_Fwded_UD_PD |
| | SC | I | 0 | SnpResp_I_Fwded_UC |
| SnpQuery | I | I | 0 | SnpResp_I |
| | UC | UC | 0 | SnpResp_UC |
| | UD | UD | 0 | SnpResp_UD |
| | SC | SC | 0 | SnpResp_SC |

20.3.4.7 任务的提前结束

在 MainPipe 上的任务在符合以下条件之一时，可以在 s3 阶段提前结束，不进入后续的流水级：

1. 任务不需要将 DataStorage 中的数据搬到 ReleaseBuffer 并符合以下条件之一：
 - 任务向上下游通道（上游 TileLink D、下游 TXREQ、下游 TXRSP、下游 TXDAT）的请求在 s3 顺利离开 MainPipe
 - 任务需要分配 MSHR
2. 任务向上游 TileLink D 通道的请求（AccessAckData、HintAck、GrantData、Grant）被重试

20.3.5 S4 流水级

在 MainPipe 上的任务若没有在 s3 阶段被提前结束，则进入 s4 阶段。任务符合以下所有条件时，可以在 s4 阶段提前结束，不进入后续的流水级：

- 任务不需要将 DataStorage 中的数据搬到 ReleaseBuffer

- 任务向上下游通道（上游 TileLink D、下游 TXREQ、下游 TXRSP、下游 TXDAT）的请求在 s4 顺利离开 MainPipe

若任务没有在 s4 被结束，则继续进入 s5 阶段。

20.3.6 S5 流水级

在 MainPipe 上的任务若没有在 s4 阶段被提前结束，则进入 s5 阶段。

若 s3 阶段发起了对 DataStorage 的读取请求，则在 s5 可以得到相应缓存行的数据。

MainPipe 在 s5 会根据任务的要求，以及请求的嵌套情况，将来自 DataStorage 或 MainPipe 上的数据写入 ReleaseBuffer。

20.4 目录 Directory

CoupledL2 采用基于目录的一致性实现方式，利用目录记录 L2 Cache 内数据块的元数据信息。如图 20.7 所示，Directory 会根据读请求的 tag 和 set，查找 L2 Cache 是否存储该数据块（是否命中）。如果命中，则返回该数据块的元数据信息。如果缺失，则挑选一个无效的路/被替换的路，返回该路数据的元数据信息。请求处理完成后，会将新的目录信息写入 Directory 进行更新。

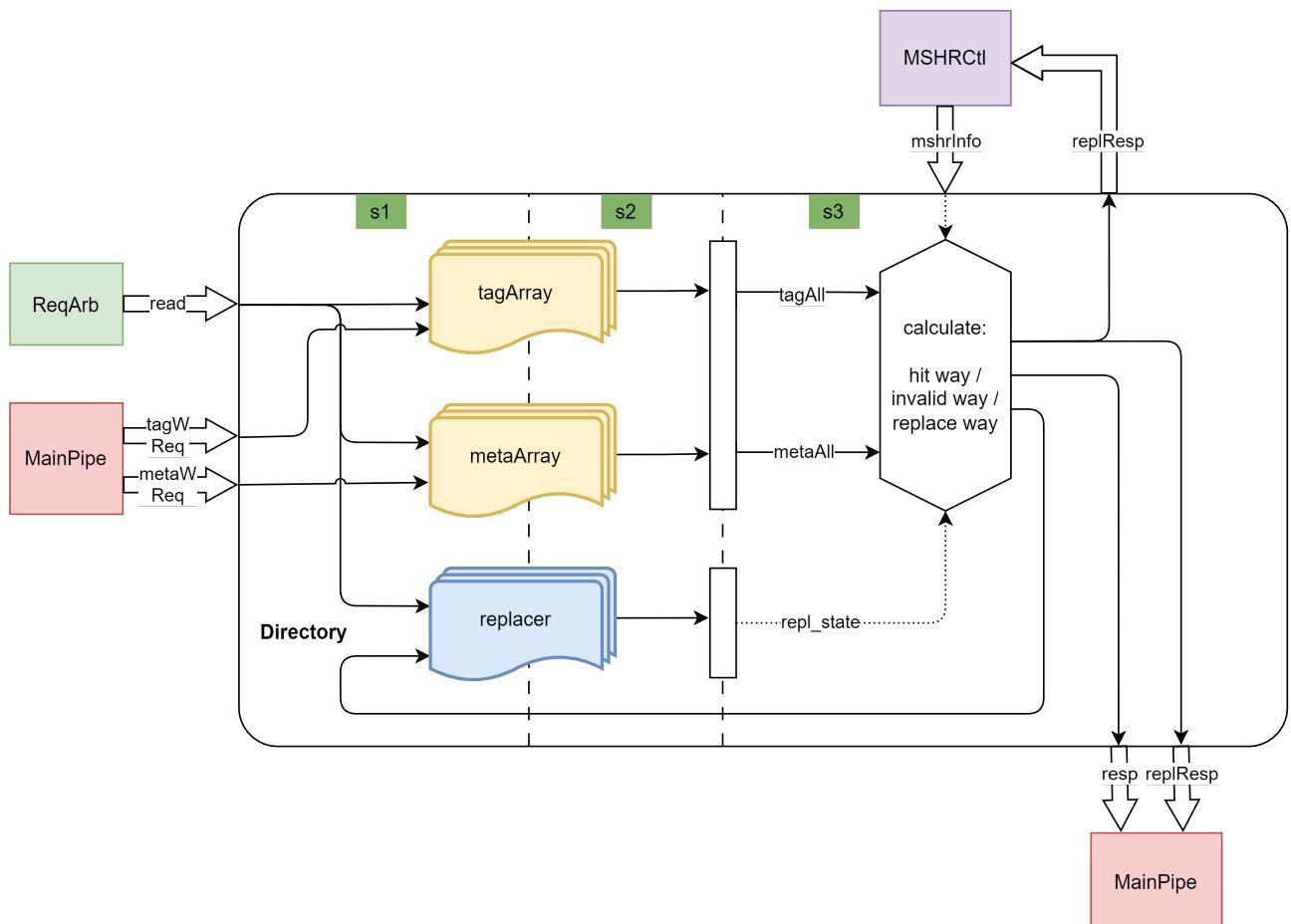


图 20.7: 目录流水线框图

20.5 数据 SRAM DataStorage

DataStorage 模块负责 CoupledL2 数据 SRAM 的读写，采用单端口 SRAM 搭建。请求只会在 MainPipe s3 流水级与 DataStorage 交互。DataStorage 每拍只能处理一个读请求或写请求。

20.6 MSHR

每个任务是否分配 MSHR 由访存流水线 (MainPipe) 根据其是否命中、是否需要 Probe L1、处理流程的复杂程度等决定，详见 § 20.3 请求仲裁器与访存流水线。

20.6.1 生命周期

每一项 MSHR 都有自己的生命周期。MSHR 项由 MainPipe 分配，并在 MSHR 完成所有的任务，清空所有状态机状态项后结束生命周期。每一项 MSHR 都可能因为等待总线事务而在较长的一段时间内保持有效，但必须在有限的时间内结束生命周期，否则意味着出现了活锁或死锁。

20.6.1.1 MSHR ID

每一项 MSHR 都有自己的 ID 值，并且该值是硬编码的，各项 MSHR 之间的 ID 不同。

由 MSHR 发起的 CHI 请求，其中 TxnID 值的低位与 MSHR ID 绑定。

20.6.1.2 分配

MainPipe 请求分配 MSHR 项时，由 MSHRCtl 模块内部的 MSHRSelector 选定一项未被分配的 MSHR。每一项 MSHR 在分配时，MainPipe 需要提供以下信息：

- 该缓存行的命中情况与一致性状态
- MSHR 状态机的初始状态
- 请求的必要原始信息（来自于 TileLink 请求或 CHI 请求）
- 请求与正在执行的写回（L2 向下的 TileLink Release 或 CHI Copy-Back Write）的嵌套情况

这些信息都会在被分配的 MSHR 项内寄存。

20.6.1.3 释放

当 MSHR 内所有的状态机项都被置为已完成时，即可立即原地释放，结束该 MSHR 项的生命周期，并准备好再一次被 MSHRSelector 选中并分配。关于 MSHR 的状态机项，详见 § 20.6.2 状态机。

20.6.2 状态机

状态机项主要分为两类：

- Schedule 状态项
- Wait 状态项

Schedule 状态项又称主动动作状态项，主要用来跟踪 MSHR 主动向 MainPipe、下游 CHI 通道、上游 TileLink 通道发送任务与请求的情况。其值为低有效，表示未完成状态，即任务尚未成功离开 MSHR 并被发出，其原因可能是未完成阻塞条件（有必要的前置动作未完成）或通道阻塞；值为高则表示对应任务已经成功发出，或不需要发出任务。

Wait 状态项又称被动动作状态项，主要用来跟踪 MSHR 期望收到的来自下游 CHI 通道、上游 TileLink 通道或 CoupledL2 内部模块的回复。其值为低有效，表示未完成状态，即对应的回复尚未回到 MSHR 项；值为高则表示对应回复已经收到，或不需要收到回复。

状态项会在 MSHR 被 MainPipe 分配时赋值，也会被 MSHR 内部的动作改变。

本小节的上游通常指 L1 缓存，下游通常指 NoC、LLC 等。

Schedule 状态项以 `s_` 为首命名，其概览如下：

| 名称 | 描述 |
|-------------------------|---|
| <code>s_acquire</code> | 首次需要向下游发送权限提升请求或 CMO 请求，或者需要向下发送被重试的写回或踢出请求 |
| <code>s_rprobe</code> | 由于替换、写回，需要向上游发送 Probe 请求 |
| <code>s_pprobe</code> | 由于下游的 Snoop 请求，需要向上游发送 Probe 请求 |
| <code>s_release</code> | 需要向下游发送的写回或踢出请求 |
| <code>s_probeack</code> | 由于下游的 Snoop 请求，需要向下游发送 Snoop 回复 |
| <code>s_refill</code> | 需要向上游发送 Grant 回复 |
| <code>s_retry</code> | 由于没有空闲的路用于替换，向上游发送的 Grant 回复需要重试 |
| <code>s_cmoresp</code> | 需要向上游发送 CBOAck 回复 |
| <code>s_cmometaw</code> | 由 CMO 引起的向 MainPipe 发送的目录更新请求 |
| <code>s_rcompack</code> | 由于向下游发送了读请求，需要发送对应的 CompAck 回复 |
| <code>s_wcompack</code> | 由于向下游发送了写请求，需要发送对应的 CompAck 回复 |
| <code>s_cbwrdata</code> | 由于向下游发送了写请求，需要发送对应的 CopyBackWrData 以写回数据 |
| <code>s_reissue</code> | 由于下游回复了 RetryAck，且 MSHR 已获得 PCredit，需要向下游重发请求 |
| <code>s_dct</code> | 由于下游的 Forwarding Snoop 请求，需要以 DCT 的形式发送 CompData 以向其它 RN 提供数据 |

Wait 状态项以 `w_` 为首命名，其概览如下：

| 名称 | 描述 |
|-------------------------------|---|
| <code>w_rprobeackfirst</code> | 由于替换、写回，向上游发送了 Probe 请求，需要等待收取来自上游的首个 Probe 回复 |
| <code>w_rprobeacklast</code> | 由于替换、写回，向上游发送了 Probe 请求，需要等待收取来自上游的最后一个 Probe 回复（单次回复时与 <code>w_rprobeackfirst</code> 动作相同） |
| <code>w_pprobeackfirst</code> | 由于下游的 Snoop 请求，向上游发送了 Probe 请求，需要等待收取来自上游的首个 Probe 回复 |

| 名称 | 描述 |
|-----------------|---|
| w_pprobeacklast | 由于下游的 Snoop 请求，向上游发送了 Probe 请求，需要等待收取来自上游的最后一个 Probe 回复（单次回复时与 w_pprobeackfirst 动作相同） |
| w_grantfirst | 由于向下发送了权限提升请求或 CMO 请求，需要等待下游的首个 Comp、CompData 或 DataSepResp 回复 |
| w_grantlast | 由于向下发送了权限提升请求或 CMO 请求，需要等待下游的最后一个 CompData 或 DataSepResp 回复（在收到 Comp 回复时与 w_grantfirst 动作相同） |
| w_grant | 由于向下发送了权限提升请求或 CMO 请求，需要等待下游的 Comp、CompData 或 RespSepData 回复，并在 CompData 与 RespSepData 回复中获得必要的 DBID 与 SrcID 信息 |
| w_releaseack | 由于向下游发送的写回请求或踢出请求，需要等待下游的 Comp 或 CompDBIDResp 回复 |
| w_replResp | 由于替换，需要等待来自 Directory 的替换选择结果 |

20.6.3 任务分发

在 Schedule 状态项未完成时，MSHR 就会尝试向相应模块、通道发送相应的任务。每一个 MSHR 项经过 MSHRCtl 的仲裁可以直接分发任务到以下模块、通道：

- MainPipe
- 上游 TileLink B 通道
- 下游 TXREQ 通道
- 下游 TXRSP 通道

对于 TXDAT 通道任务的分发，则必须经过 MainPipe，详见 § 20.3 请求仲裁器与访存流水线。

送往 MainPipe 的任务还会在同一周期内经过 RequestArb 的仲裁，详见 § 20.3 请求仲裁器与访存流水线。

各个 Schedule 状态项对应的任务分发方向如下表：

| 名称 | 目标模块/通道 |
|------------|------------------|
| s_acquire | 下游 TXREQ 通道 |
| s_rprobe | 上游 TileLink B 通道 |
| s_pprobe | 上游 TileLink B 通道 |
| s_release | MainPipe |
| s_probeack | MainPipe |
| s_refill | MainPipe |
| s_retry | - |
| s_cmoresp | MainPipe |
| s_cmometaw | MainPipe |
| s_rcompack | 下游 TXRSP 通道 |
| s_wcompack | 下游 TXRSP 通道 |

| 名称 | 目标模块/通道 |
|------------|----------|
| s_cbwrdata | MainPipe |
| s_reissue | - |
| s_dct | MainPipe |

20.6.3.1 MainPipe

各个 MSHR 会根据自身状态机项的状态向 MainPipe 发送几种不同类型的任务。

20.6.3.1.1 写回请求任务 (mp_release)

写回请求任务 (mp_release) 由状态机项 s_release 触发，该任务的作用是在 MainPipe 通过 TXREQ 通道发送需要的缓存行写回请求或踢出请求。在状态机项 s_release 的状态未完成，且当前 MSHR 的状态符合一定的条件时，MSHR 就会尝试向 MainPipe 发送写回请求任务。

s_release 被置为未完成时，其在各个情况下的 MSHR 状态需要满足以下条件，才可以向 MainPipe 发送写回请求任务：

1. 来自替换任务

- 已经完成替换路的选择
- 已经收到向上游的 Probe 的所有回复
- 替换读请求已经收到来自下游的全部数据

2. 来自 CMO 请求

- 已经收到向上游的 Probe 的所有回复

写回请求任务会要求 MainPipe 向 TXREQ 通道发送请求：

| 任务来源 | 上游 A 通道请求类型 | 是否有脏数据 | 下游 TXREQ 请求类型 |
|--------|-------------|--------|-------------------|
| 替换任务 | Acquire* | 是 | WriteBackFull |
| | | 否 | WriteEvictOrEvict |
| CMO 请求 | CBOClean | - | WriteCleanFull |
| | CBOFlush | 是 | WriteBackFull |
| | | 否 | Evict |
| | CBOInval | - | Evict |

写回请求任务会按照情况要求 MainPipe 将 MSHR 持有的关联数据写入 DataStorage：

| 任务来源 | 上游 A 通道请求类型 | 是否有脏数据 | 数据来源 | 是否写入 DataStorage |
|--------|-------------|--------------------|--------------------|------------------|
| 替换任务 | Acquire* | - | RefillBuffer | 是 |
| CMO 请求 | CBO* | 来自向上游的 Probe 其它 | ReleaseBuffer - | 是 否 |

且 CMO 请求在写回请求任务中要求 MainPipe 更新 Directory 中的缓存行状态，且清除缓存行的 Dirty 标记：

| 任务来源 | 上游 A 通道请求类型 | 起始状态 | 写入的状态 |
|--------|-------------|---------|---------|
| CMO 请求 | CBOClean | TRUNK | TIP |
| | | TIP | TIP |
| | | BRANCH | BRANCH |
| | | INVALID | INVALID |
| | CBOFlush | - | INVALID |
| | CBOInval | - | INVALID |

20.6.3.1.2 下游 Snoop 回复任务 (mp_probeack)

下游 Snoop 回复任务 (mp_probeack) 由状态机项 s_probeack 触发。该任务的作用为在 MainPipe 通过 TXRSP 或 TXDAT 通道发送向下游的 Snoop 回复。在状态机项 s_probeack 的状态未完成，且当前 MSHR 的状态符合一定的条件时，MSHR 就会尝试向 MainPipe 发送下游 Snoop 回复任务。

s_probeack 被置为未完成时，其 MSHR 状态需要满足以下条件，才可以向 MainPipe 发送下游 Snoop 回复任务：

- 已经收到向上游的 Probe 的所有回复

下游 Snoop 回复任务会要求 MainPipe 向 TXRSP 或 TXDAT 通道发送消息，并在 MSHR 内指定回复的 Snoop Response 类型，详见 § 20.6.4 Snoop 处理。

下游 Snoop 回复任务会在满足以下情况时要求 MainPipe 将 MSHR 持有的关联数据写入 DataStorage：

- 下游 Snoop 请求的目标状态不是 I
- 上游 L1 在 Probe 过程中返回了脏数据 (ProbeAckData)
- 上游 L1 没有在 Probe 结束前嵌套发起脏数据的写回 (ReleaseData)

下游 Snoop 回复任务会要求 MainPipe 更新缓存行状态，详见 § 20.6.4 Snoop 处理。

20.6.3.1.3 替换路查询与上游 Grant/CBOAck 回复任务 (mp_grant)

替换路查询与上游 Grant/CBOAck 回复任务 (mp_grant) 由状态机项 s_refill 或 s_cmoresp 触发，且 s_refill 与 s_cmoresp 不会被同时置为未完成。该任务的作用为以下几项之一：

- 在 MainPipe 向 Directory 发起替换路查询请求
- 在 MainPipe 通过 TileLink D 通道向上游回复 Grant/GrantData
- 在 MainPipe 通过 TileLink D 通道向上游回复 CBOAck

在状态机项 s_release 的状态未完成，且当前 MSHR 的状态符合一定的条件时，MSHR 就会尝试向 MainPipe 发送替换路查询或上游 Grant 回复任务；在状态机项 s_cmoresp 的状态未完成，且当前 MSHR 的状态符合一定的条件时，MSHR 就会尝试向 MainPipe 发送 CBOAck 回复任务。

在 s_refill 被置为未完成时，其 MSHR 状态需要满足以下条件，才可以向 MainPipe 发送替换路查询与上游 Grant 回复任务：

- 已经收到向上游的 Probe 的所有回复
- 已经收到来自下游的首个 Comp、CompData 或 RespSepData 回复
- 若需要，收到来自下游的所有 Comp、CompData 或 DataSepResp 回复
- 替换路查询重试没有超过重试抑制阈值

在连续多次发送替换路重试请求后，MSHR 会将其抑制一段时间，以防止过于密集、连续的重试导致活锁。

在 `s_cmoresp` 被置为未完成时，其 MSHR 状态需要满足以下条件，才可以向 MainPipe 发送替换路查询与上游 CBOAck 回复任务：

- 已经收到向上游的 Probe 的所有回复
- 已经收到属于 `w_releaseack` 的来自下游的 Comp 回复
- 已经收到属于 `w_grant` 的来自下游的 Comp 回复 (`w_releaseack` 完成后 `w_grant` 才可接收下游的 Comp 回复)
- 若需要，已经完成发送所有的 CopyBackWrData

并且这些条件隐含一个特征，即 `s_cmoresp` 在除 CBOAck 回复流程之外的所有 CMO 子动作都已经完成后，才可以发起任务。

在 MSHR 需要等待替换路结果，并且 Directory 给予了重试回复后，MSHR 就会从 `mp_grant` 发送替换路重试任务。

替换路查询与上游 Grant 任务会要求 MainPipe 对 Directory 中的缓存行状态进行更新，若为上游 CBOAck 任务则不会要求，其更新规则如下：

| 任务来源 | 请求类型 | 初始状态 | 更新状态 |
|------------------------|--------------------|-----------------------------------|--|
| <code>s_refill</code> | Get | TIP TRUNK BRANCH INVALID | TIP TIP BRANCH TIP* BRANCH |
| | Acquire* toT | - | TRUNK |
| | Acquire* toB | - | TRUNK* BRANCH |
| | Hint PrefetchWrite | - | TIP |
| | Hint PrefetchRead | - | TIP* BRANCH |
| <code>s_cmoresp</code> | - | - | - |

其中 Get 更新为 TIP、Acquire* toB 更新为 TRUNK、Hint PrefetchRead 更新为 TIP 会发生于以下情况：

- 缓存行不存在于上游 L1 内，且在 L2 本地为 TIP 权限
- 正在对缓存行进行的操作不是 Alias 替换，且在 L2 本地为 TIP 或 TRUNK 权限
- 缓存行不存在于 L2 内，且向下游发送的读取请求返回了写权限

替换路查询与上游 Grant 任务在 Directory 未命中时会要求 MainPipe 将 Directory 中被选中替换的缓存行的对应 Tag 值，若为上游 CBOAck 任务则不会要求。

替换路查询与上游 Grant/CBOAck 任务会在满足以下其中一个条件时要求 MainPipe 将 MSHR 持有的关联数据写入 DataStorage：

- 收到来自下游的 CompData 或 DataSepResp 数据回复
- 在完成 Get 或 Alias 替换流程时向上游发送的 Probe 收到了脏数据

20.6.3.1.4 下游 CopyBackWrData 任务 (mp_cbwrdata)

写回请求任务 (mp_cbwrdata) 由状态机项 s_cbwrdata 触发。该任务的作用是在 MainPipe 通过 TXDAT 通道完成需要向下游发送的 CopyBackWrData。在状态机项 s_cbwrdata 的状态未完成，且当前 MSHR 的状态符合一定的条件时，MSHR 就会尝试向 MainPipe 发送写回请求任务。

s_cbwrdata 通常被 s_release 与 w_releaseack 的以下动作置为未完成：

- 写回请求任务正离开 MSHR，即 s_release 状态项正被置为完成时

需要注意的是，在发送 WriteEvictOrEvict 后收到的回复为 Comp 时，s_cbwrdata 会在 MSHR 未向 MainPipe 发送任何下游 CopyBackWrData 任务的前提下将 s_cbwrdata 置为已完成。

其 MSHR 状态需要满足以下条件，才可以向 MainPipe 发送写回请求任务：

- 写回请求任务已离开 MSHR，即 s_release 状态项已完成

20.6.3.1.5 下游 DCT CompData 任务 (mp_dct)

下游 DCT CompData 任务 (mp_dct) 由状态机项 s_dct 触发。该任务的作用是在 MainPipe 通过 TXDAT 通道完成 Forwarding Snoop 中 DCT 的部分。在状态机项 s_dct 的状态未完成，且当前 MSHR 的状态符合一定的条件时，MSHR 就会尝试向 MainPipe 发送下游 DCT CompData 任务。

在 s_dct 被置为未完成时，其 MSHR 状态需要满足以下条件，才可以向 MainPipe 发送下游 DCT CompData 任务：

- 在 Forwarding Snoop 流程中目标为 HN 的 Snoop 回复任务已经离开 MSHR，即 s_probeack 状态项为已完成

下游 DCT CompData 任务会要求 MainPipe 通过 TXDAT 通道发送 CompData 回复。且根据 DCT 的定义，该 CompData 回复的目标是另一个处理器核（即 RN）。

20.6.3.1.6 CMO 缓存状态更新任务 (mp_cmometaw)

CMO 缓存状态更新任务 (mp_cmometaw) 由状态机项 s_cmometaw 触发。该任务的作用是在 CBOClean 操作不需要进行 WriteCleanFull 写回时，在 MainPipe 更新缓存行状态。

在 s_cmometaw 被置为未完成时，MSHR 就可以向 MainPipe 发送 CMO 缓存状态更新任务，并进行如下更新：

- 在收到 ProbeAck toN 时更新记录为缓存行不存在于上游 L1
- 清除状态至 Clean
- 更新权限为 TIP

20.6.3.2 上游 TileLink B 通道

向上游 TileLink B 通道的请求发送由状态机项 s_pprobe 或 s_rprobe 触发。且 s_pprobe 与 s_rprobe 不会被同时置为未完成。

在 `s_pprobe` 或 `s_rprobe` 中的任意一个状态机项被置为未完成时，MSHR 就可以向上游 TileLink B 通道发送请求。在各情况下的请求类型见下表：

| 任务来源 | 上游请求类型 | 下游请求类型 | 缓存行状态 | 发送请求类型 |
|-----------------------|----------|----------------------|-------|-----------|
| <code>s_pprobe</code> | - | SnpOnce | - | Probe toT |
| | - | SnpClean | - | Probe toB |
| | - | SnpShared | - | Probe toB |
| | - | SnpNotSharedDirty | - | Probe toB |
| | - | SnpUnique | - | Probe toN |
| | - | SnpCleanShared | - | Probe toT |
| | - | SnpCleanInvalid | - | Probe toN |
| | - | SnpMakeInvalid | - | Probe toN |
| | - | SnpMakeInvalidStash | - | Probe toN |
| | - | SnpUniqueStash | - | Probe toN |
| | - | SnpStashUnique | - | Probe toT |
| | - | SnpStashShared | - | Probe toT |
| | - | SnpOnceFwd | - | Probe toT |
| | - | SnpCleanFwd | - | Probe toB |
| | - | SnpNotSharedDirtyFwd | - | Probe toB |
| | - | SnpSharedFwd | - | Probe toB |
| | - | SnpUniqueFwd | - | Probe toN |
| | - | SnpQuery | - | Probe toT |
| <code>s_rprobe</code> | Get | - | TRUNK | Probe toB |
| | Acquire* | - | - | Probe toN |
| | CBOClean | - | TRUNK | Probe toB |

20.6.3.3 下游 TXREQ 通道

向下游 TXREQ 通道的请求发送由状态机项 `s_acquire` 或 `s_reissue` 触发。且 `s_acquire` 与 `s_reissue` 不会被同时置为未完成。

在 `s_acquire` 被置为未完成时，对于替换任务，可以立即向下游发送权限提升请求。但对于 CMO 任务，需要满足以下条件后，才可以向下游发送 CMO 请求：

- 向上游的 Probe 已经收到所有回复
- 向下游的写回请求已经收到 Comp 或 CompDBIDResp
- 向下游的 CopyBackWrData 任务已经离开 MSHR 或不需要该任务

在 `s_reissue` 被置为未完成时，需要满足以下条件，才可以向下游重发被重试的请求：

- 已收到来自下游的 RetryAck
- 已收到并被分配到来自下游的 PCrdGrant
- 处于存在已离开 MSHR 的 `mp_release` 或下游 TXREQ 通道任务但未收到任何回复的状态，即 `s_release` 状态项已完成但 `w_releaseack` 状态项未完成，或 `s_acquire` 状态项已完成但 `w_grant` 状态项未完成

向下游 TXREQ 通道在各个情况下发送的请求类型如下表：

| 任务来源 | 未完成状态 | 上游请求类型 | 未完成请求类型 | 发送请求类型 |
|------------------|---------------------|--------------------|--------------------|--------------------|
| s_acquire | - | Get | - | ReadNotSharedDirty |
| | | AcquirePerm toT | - | MakeUnique |
| | | AcquireBlock toT | - | ReadUnique |
| | | AcquireBlock toB | - | ReadNotSharedDirty |
| | | Hint PrefetchWrite | - | ReadUnique |
| | | Hint PrefetchRead | - | ReadNotSharedDirty |
| | | CBOClean | - | CleanShared |
| | | CBOFlush | - | CleanInvalid |
| | | CBOInval | - | MakeInvalid |
| s_reissue | w_grant | Get | ReadNotSharedDirty | ReadNotSharedDirty |
| | | AcquirePerm toT | MakeUnique | MakeUnique |
| | | AcquireBlock toT | ReadUnique | ReadUnique |
| | | AcquireBlock toB | ReadNotSharedDirty | ReadNotSharedDirty |
| | | Hint PrefetchWrite | ReadUnique | ReadUnique |
| | | Hint PrefetchRead | ReadNotSharedDirty | ReadNotSharedDirty |
| | | CBOClean | CleanShared | CleanShared |
| | | CBOFlush | CleanInvalid | CleanInvalid |
| | | CBOInval | MakeInvalid | MakeInvalid |
| s_reissue | w_releaseack | Acquire* | WriteBackFull | WriteBackFull |
| | | | WriteEvictOrEvict | WriteEvictOrEvict |
| | | CBOClean | WriteCleanFull | WriteCleanFull |
| | | CBOFlush | WriteBackFull | WriteBackFull |
| | | | Evict | Evict |
| | | CBOInval | Evict | Evict |

20.6.3.4 下游 TXRSP 通道

向下游 TXRSP 通道的消息发送由状态机项 **s_rcompack** 或 **s_wcompack** 触发，该通道主要用来向下游发送 CompAck 消息。其中 **s_rcompack** 与 **s_wcompack** 可能被同时置为未完成，且 **s_rcompack** 的优先级更高。

在 **s_rcompack** 被置为未完成时，需要满足以下条件，才可以向下游发送 CompAck 消息：

1. 配置为 Issue B 时
 - 收到下游的 Comp 或所有 CompData
2. 配置为 Issue C 以及更新版本时
 - 收到下游的 Comp 或首个 CompData 或 RespSepData 与首个 DataSepResp

在 **s_wcompack** 被置为未完成时，需要满足以下条件，才可以向下游发送 CompAck 消息：

- **s_rcompack** 已完成或未被置为未完成

20.6.4 Snoop 处理

20.6.4.1 非嵌套 Snoop

当没有同地址的写回请求未完成时，收到的 Snoop 即为非嵌套的普通 Snoop，是 Snoop 最基本的处理情况。非嵌套 Snoop 在 MSHR 中的处理方式如下：

| Snoop 请求类型 | 起始状态 | 最终状态 | RetToSrc | Snoop 回复 |
|---------------------|------|------|----------|--------------------|
| SnpOnce | I | - | - | - |
| | UC | UC | X | SnpRespData_UC |
| | UD | UD | X | SnpRespData_UD_PD |
| | SC | - | - | - |
| SnpClean, | I | - | - | - |
| | UC | SC | X | SnpResp_SC |
| | UD | SC | X | SnpRespData_SC_PD |
| | SC | - | - | - |
| SnpShared, | I | - | - | - |
| | UC | I | X | SnpResp_I |
| | UD | I | X | SnpRespData_I_PD |
| | SC | I | 0 | SnpResp_I |
| SnpUnique | I | - | - | - |
| | UC | I | X | SnpRespData_I |
| | UD | I | X | SnpRespData_I |
| | SC | I | 0 | SnpRespData_I |
| SnpNotSharedDirty | I | - | - | - |
| | UC | SC | X | SnpRespData_SC |
| | UD | SC | X | SnpRespData_SC_PD |
| | SC | - | - | - |
| SnpCleanShared | I | - | - | - |
| | UC | UC | 0 | SnpResp_UC |
| | UD | UC | 0 | SnpRespData_UC_PD |
| | SC | - | - | - |
| SnpCleanInvalid | I | - | - | - |
| | UC | I | 0 | SnpResp_I |
| | UD | I | 0 | SnpRespData_I_PD |
| | SC | I | 0 | SnpResp_I |
| SnpMakeInvalid | - | I | 0 | SnpResp_I |
| SnpMakeInvalidStash | - | I | 0 | SnpResp_I |
| SnpUniqueStash | I | - | - | - |
| | UC | I | 0 | SnpResp_I |
| | UD | I | 0 | SnpRespData_I_PD |
| | SC | I | 0 | SnpResp_I |
| SnpStashUnique, | I | - | - | - |
| | UC | UC | 0 | SnpResp_UC |
| | UD | UD | 0 | SnpResp_UD |
| | SC | - | - | - |
| SnpOnceFwd | I | I | 0 | SnpResp_I |
| | UC | UC | 0 | SnpResp_UC_Fwded_I |
| | UD | UD | 0 | SnpResp_UD_Fwded_I |
| | SC | SC | 0 | SnpResp_SC_Fwded_I |

| Snoop 请求类型 | 起始状态 | 最终状态 | RetToSrc | Snoop 回复 |
|-----------------------|------|------|----------|----------------------------|
| SnpCleanFwd, | I | I | X | SnpResp_I |
| SnpNotSharedDirtyFwd, | UC | SC | 0 | SnpResp_SC_Fwded_SC |
| SnpSharedFwd | | | 1 | SnpRespData_SC_Fwded_SC |
| | UD | SC | X | SnpRespData_SC_PD_Fwded_SC |
| | SC | SC | 0 | SnpResp_SC_Fwded_SC |
| | | | 1 | SnpRespData_SC_Fwded_SC |
| SnpUniqueFwd | I | I | 0 | SnpResp_I |
| | UC | I | 0 | SnpResp_I_Fwded_UC |
| | UD | I | 0 | SnpResp_I_Fwded_UD_PD |
| | SC | I | 0 | SnpResp_I_Fwded_UC |
| SnpQuery | I | - | - | - |
| | UC | UC | 0 | SnpResp_UC |
| | UD | UD | 0 | SnpResp_UD |
| | SC | - | - | - |

“-” 以及未列出的缓存状态表示在相应情况下的该类请求不会进入 MSHR。

关于具体 Snoop 请求在什么情况下会分配 MSHR，详见 § 20.3 请求仲裁器与访存流水线。

20.6.4.2 嵌套 Snoop

在 MSHR 处理向下游的写回请求过程中，仍然需要保证 CoupledL2 需要能够响应下游的 Snoop 请求以及上游的 Release 请求。此时，未完成的写回请求则视为被 Snoop 请求嵌套，或被上游的 Release 请求嵌套。需要注意的是，由于 CHI 协议中存在 Silent Eviction，且 Evict 请求的初始状态为 I，故不发生数据写回的踢出请求（Evict）不会被视为嵌套。

“-” 以及未列出的缓存状态表示在相应情况下的该类请求不会进入 MSHR，或不在嵌套情况范围内。

关于具体 Snoop 请求在什么情况下会分配 MSHR，详见 § 20.3 请求仲裁器与访存流水线。

可能在 MSHR 内发生并处理的下游 Snoop 请求嵌套如下。

20.6.4.2.1 特性 1：Snoop 与 WriteBackFull 的嵌套

| Snoop 请求类型 | 起始状态 | 嵌套前 | | 嵌套后 | |
|-------------------|------|-----|----|----------|----------|
| | | 状态 | 状态 | RetToSrc | Snoop 回复 |
| SnpOnce | - | - | - | - | - |
| SnpClean | - | - | - | - | - |
| SnpShared | - | - | - | - | - |
| SnpNotSharedDirty | - | - | - | - | - |
| SnpCleanShared | - | - | - | - | - |
| SnpCleanInvalid | - | - | - | - | - |
| SnpMakeInvalid | - | - | - | - | - |

| Snoop 请求类型 | 起始状态 | 嵌套前 | | 嵌套后 | | Snoop 回复 |
|----------------------|------|-----|----|----------|---------------------------|----------|
| | | 状态 | 状态 | RetToSrc | Snoop 回复 | |
| SnpUnique | - | - | - | - | - | - |
| SnpUniqueStash | - | - | - | - | - | - |
| SnpMakeInvalidStash | - | - | - | - | - | - |
| SnpStashUnique | - | - | - | - | - | - |
| SnpStashShared | - | - | - | - | - | - |
| SnpOnceFwd | UD | UD | I | X | SnpRespData_I_PD_Fwded_I | |
| SnpCleanFwd | UD | UD | I | X | SnpRespData_I_PD_Fwded_SC | |
| SnpSharedFwd | UD | UD | I | X | SnpRespData_I_PD_Fwded_SC | |
| SnpNotSharedDirtyFwd | UD | UD | I | X | SnpRespData_I_PD_Fwded_SC | |
| SnpUniqueFwd | UD | UD | I | X | SnpResp_I_Fwded_UD_PD | |
| SnpQuery | - | - | - | - | - | - |

20.6.4.2.2 特性 2: Snoop 与 WriteEvictOrEvict 的嵌套

| Snoop 请求类型 | 起始状态 | 嵌套前 | | 嵌套后 | | Snoop 回复 |
|----------------------|------|-----|----|----------|------------------------|----------|
| | | 状态 | 状态 | RetToSrc | Snoop 回复 | |
| SnpOnce | - | - | - | - | - | - |
| SnpClean | - | - | - | - | - | - |
| SnpShared | - | - | - | - | - | - |
| SnpNotSharedDirty | - | - | - | - | - | - |
| SnpCleanShared | - | - | - | - | - | - |
| SnpCleanInvalid | - | - | - | - | - | - |
| SnpMakeInvalid | - | - | - | - | - | - |
| SnpUnique | - | - | - | - | - | - |
| SnpUniqueStash | - | - | - | - | - | - |
| SnpMakeInvalidStash | - | - | - | - | - | - |
| SnpStashUnique | - | - | - | - | - | - |
| SnpStashShared | - | - | - | - | - | - |
| SnpOnceFwd | UC | UC | I | X | SnpRespData_I_Fwded_I | |
| SnpCleanFwd | UC | UC | I | 0 | SnpResp_I_Fwded_SC | |
| | | | | 1 | SnpRespData_I_Fwded_SC | |
| SnpSharedFwd | UC | UC | I | 0 | SnpResp_I_Fwded_SC | |
| | | | | 1 | SnpRespData_I_Fwded_SC | |
| SnpNotSharedDirtyFwd | UC | UC | I | 0 | SnpResp_I_Fwded_SC | |
| | | | | 1 | SnpRespData_I_Fwded_SC | |
| SnpUniqueFwd | UC | UC | I | 0 | SnpResp_I_Fwded_UC | |
| SnpQuery | - | - | - | - | - | - |

20.6.4.2.3 特性 3: Snoop 与 WriteCleanFull 的嵌套

| Snoop 请求类型 | 起始状态 | 嵌套前 | | 嵌套后 | |
|----------------------|------|-----|----|----------|----------------------------|
| | | 状态 | 状态 | RetToSrc | Snoop 回复 |
| SnpOnce | - | - | - | - | - |
| SnpClean | - | - | - | - | - |
| SnpShared | - | - | - | - | - |
| SnpNotSharedDirty | - | - | - | - | - |
| SnpCleanShared | - | - | - | - | - |
| SnpCleanInvalid | UD | UD | I | 0 | SnpRespData_I_PD |
| | | UC | I | 0 | SnpResp_I |
| | | SC | I | 0 | SnpResp_I |
| SnpMakeInvalid | UD | UD | I | 0 | SnpResp_I |
| | | UC | I | 0 | SnpResp_I |
| | | SC | I | 0 | SnpResp_I |
| SnpUnique | UD | UD | I | X | SnpRespData_I_PD |
| | | UC | I | X | SnpResp_I |
| | | SC | I | 0 | SnpResp_I |
| | | | | 1 | SnpRespData_I |
| SnpUniqueStash | UD | UD | I | 0 | SnpRespData_I_PD |
| | | UC | I | 0 | SnpResp_I |
| | | SC | I | 0 | SnpResp_I |
| SnpMakeInvalidStash | UD | UD | I | 0 | SnpResp_I |
| | | UC | I | 0 | SnpResp_I |
| | | SC | I | 0 | SnpResp_I |
| SnpStashUnique | - | - | - | - | - |
| SnpStashShared | - | - | - | - | - |
| SnpOnceFwd | UD | UD | SC | 0 | SnpRespData_SC_PD_Fwded_I |
| | | UC | UC | 0 | SnpResp_UC_Fwded_I |
| | | SC | SC | 0 | SnpResp_SC_Fwded_I |
| SnpCleanFwd | UD | UD | SC | X | SnpRespData_SC_PD_Fwded_SC |
| | | UC | SC | 0 | SnpResp_SC_Fwded_SC |
| | | | | 1 | SnpRespData_SC_Fwded_SC |
| | | SC | SC | 0 | SnpResp_SC_Fwded_SC |
| | | | | 1 | SnpRespData_SC_Fwded_SC |
| SnpSharedFwd | UD | UD | SC | X | SnpRespData_SC_PD_Fwded_SC |
| | | UC | SC | 0 | SnpResp_SC_Fwded_SC |
| | | | | 1 | SnpRespData_SC_Fwded_SC |
| | | SC | SC | 0 | SnpResp_SC_Fwded_SC |
| | | | | 1 | SnpRespData_SC_Fwded_SC |
| SnpNotSharedDirtyFwd | UD | UD | SC | X | SnpRespData_SC_PD_Fwded_SC |
| | | UC | SC | 0 | SnpResp_SC_Fwded_SC |
| | | | | 1 | SnpRespData_SC_Fwded_SC |

| Snoop 请求类型 | 起始状态 | 嵌套前 | 嵌套后 | RetToSrc | Snoop 回复 |
|--------------|------|-----|-----|-------------------------|-----------------------|
| | | 状态 | 状态 | | |
| SnpUniqueFwd | UD | SC | 1 | SnpRespData_SC_Fwded_SC | |
| | | | 0 | SnpResp_SC_Fwded_SC | |
| | | | 1 | SnpRespData_SC_Fwded_SC | |
| | UC | UD | I | 0 | SnpResp_I_Fwded_UD_PD |
| | | UC | I | 0 | SnpResp_I_Fwded_UC |
| SnpQuery | - | - | - | - | - |

20.6.5 写回嵌套处理

在可能发生嵌套时，每一项 MSHR 会收到 MainPipe 广播的请求嵌套信息，其中包含可能发生嵌套的缓存行的 Tag 与 Set 地址以及嵌套行为。具体信号为 MSHR 内的 `nestwb` 端口与 NestedWriteback Bundle 类。

考虑可能导致嵌套的上游 Release/ReleaseData 请求与下游 Snoop 请求，在 MSHR 内需要的各种嵌套处理逻辑如下。

20.6.5.1 特性 1：正被替换的缓存行与上游的 ReleaseData TtoN 嵌套

当 MSHR 中被替换的缓存行的 Tag 和 Set 地址与由 MainPipe 广播到各项 MSHR 的 ReleaseData TtoN 的 Tag 和 Set 地址相同时，发生此种嵌套。对应的信号名为 `c_set_dirty`。

此种嵌套通常出现在 CoupledL2 已经或正在向上游 L1 缓存发送由替换引起的 Probe toN 请求，且上游 L1 缓存对于该 Probe toN 的回复尚未被 CoupledL2 观测到时，上游 L1 缓存主动向 CoupledL2 发起了 ReleaseData TtoN。

此时需要对 MSHR 内记录的缓存行状态进行如下更新：

- 标记为 Dirty
- 更新状态为 TIP
- 更新状态为上游 L1 不再持有该缓存行

20.6.5.2 特性 2：正被替换的缓存行与上游的 Release TtoN 嵌套

当 MSHR 中被替换的缓存行的 Tag 和 Set 地址与由 MainPipe 广播到各项 MSHR 的 Release TtoN 的 Tag 和 Set 地址相同时，发生此种嵌套。对应的信号名为 `c_set_tip`。

此种嵌套通常出现在 CoupledL2 已经或正在向上游 L1 缓存发送由替换引起的 Probe toN 请求，且上游 L1 缓存对于该 Probe toN 的回复尚未被 CoupledL2 观测到时，上游 L1 缓存主动向 CoupledL2 发起了 Release TtoN。

此时需要对 MSHR 内记录的缓存行状态进行如下更新：

- 更新状态为 TIP
- 更新状态为上游 L1 不再持有该缓存行

20.6.5.3 特性 3：正被替换的缓存行与下游的 Snoop 嵌套

当 MSHR 中被替换的缓存行的 Tag 和 Set 地址与由 MainPipe 广播到各项 MSHR 的下游 Snoop 的 Tag 和 Set 地址相同时，发生此嵌套。对应的信号名为 `b_inv_dirty`。

此处的下游 Snoop 需要排除在 CHI 规定中不可改变缓存行状态的一类 Snoop，包括 SnpQuery、SnpStashUnique、SnpStashShared。

此种嵌套通常出现在 CoupledL2 已经或正在向下游发送由替换引起的写回请求，且下游尚未回复 CompDBIDResp 时，下游向 CoupledL2 发起了新的 Snoop 请求。

此时需要对 MSHR 内记录的缓存行状态进行如下更新：

- 清除状态至 Clean
- 更新状态为 INVALID
- 清除由于上游 L1 缓存回复 ProbeAckData 而置的 Dirty 标志

20.6.5.4 特性 4：在下游 Snoop 发生嵌套时向目录写入 BRANCH 状态

当 MSHR 的 Tag 和 Set 地址与由 MainPipe 广播到各项 MSHR 的下游 Snoop 的 Tag 和 Set 地址相同，且该 Snoop 请求在 MainPipe 上写入了 BRANCH 的缓存行状态时，发生此嵌套。

此种嵌套通常出现在 CoupledL2 已经或正在向下游发送由替换引起的写回请求，且下游尚未回复 CompDBIDResp 时，下游向 CoupledL2 发起了新的 Snoop 请求。

此时需要对 MSHR 内记录的缓存行状态进行如下更新：

- 清除状态至 Clean
- 若缓存行权限不为 INVALID，则更新为 BRANCH
- 清除由于上游 L1 缓存回复 ProbeAckData 而置的 Dirty 标志

20.6.5.5 特性 5：在下游 Snoop 发生嵌套时向目录写入 INVALID 状态

当 MSHR 的 Tag 和 Set 地址与由 MainPipe 广播到各项 MSHR 的下游 Snoop 的 Tag 和 Set 地址相同，且该 Snoop 请求在 MainPipe 上写入了 INVALID 的缓存行状态时，发生此嵌套。

此种嵌套通常出现在 CoupledL2 已经或正在向下游发送由替换引起的写回请求，且下游尚未回复 CompDBIDResp 时，下游向 CoupledL2 发起了新的 Snoop 请求。

此时需要对 MSHR 内记录的缓存行状态进行如下更新：

- 清除状态至 Clean
- 更新状态为 INVALID
- 更新状态为上游 L1 不再持有该缓存行
- 清除由于上游 L1 缓存回复 ProbeAckData 而置的 Dirty 标志
- 若为需要执行替换的请求，重新选择被替换的行

20.6.6 Retry 与 P-Credit 机制

若收到了来自下游的 RetryAck 回复，MSHR 就会拉高 P-Credit 查询的有效位，并且将 CHI 的 PCrdType 与 SrcID 域发送给 MainPipe，由 MainPipe 决定是否向 MSHR 的对应事务分配 P-Credit 以进行重试。关于 P-Credit 的接收与分配，详见 § 20.3 请求仲裁器与访存流水线。

20.7 上游 TileLink 总线通道

20.7.1 SinkA

20.7.1.1 功能描述

SinkA 把来自 [总线 A 通道, 预取器] 的请求进行处理, 转化为内部任务格式, 然后发送给 RequestBuffer。二者同时来请求时, 总线 A 通道的请求有着更高的优先级。

20.7.1.1.1 特性 1: A 通道/预取器阻塞

当 RequestBuffer 不能接收 SinkA 的申请时, 总线 A 通道接口/预取器通过 valid/ready 协议来阻塞后面的申请。

20.7.1.2 整体框图

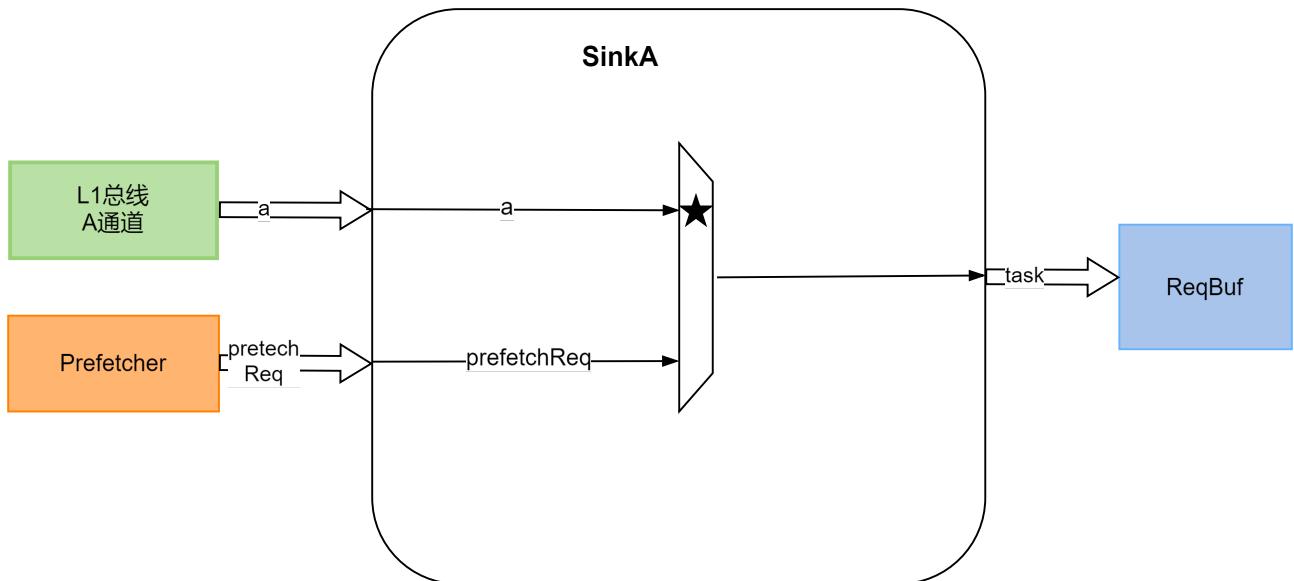


图 20.8: SinkA

20.7.2 SinkC

20.7.2.1 功能描述

SinkC 接收来自总线 C 通道的请求 (Release/ReleaseData/ProbeAck/ProbeAckData), 内部 Buffer 深度为 3, 用 valid/ready 握手协议阻塞 C 通道, 进行如下操作: a. 如果请求是 Release(Data), 则为其分配一个 buffer 项保存, 等到流水线可以接收 C 请求的时候, 就发往 RequestArb 进入主流水线; 如果包含数据, 则延迟两拍在请求进入到 S3 的时候将其数据发送给 MainPipe; b. 如果请求是 ProbeAckData, 则直接向 MSHR 发送反馈, 同时将其数据写入 ReleaseBuf 中。

20.7.2.1.1 特性 1: RefillBuffer 覆盖

因为目前缺失重填操作会首先将数据返回给 L1，然后再安排重填数据（在 RefillBuf 中）写入 L2 的 DataStorage，二者之间存在一个时间差。如果在此期间 L1 就将脏数据释放下来，为了保证写入 L2 的最新数据，我们就让 ReleaseData 也将其数据同步写入一份到 RefillBuf 中，覆盖原有的重填数据。

20.7.2.1.2 特性 2: ReleaseBuffer 覆盖

当 MSHR 处理一笔 release 需要 probe L1D\$ 时，这笔 probeAckData 查找匹配到 MSHR 中的这笔 release 后，会主动把数据写入到 ReleaseBuf 中。

20.7.2.2 整体框图

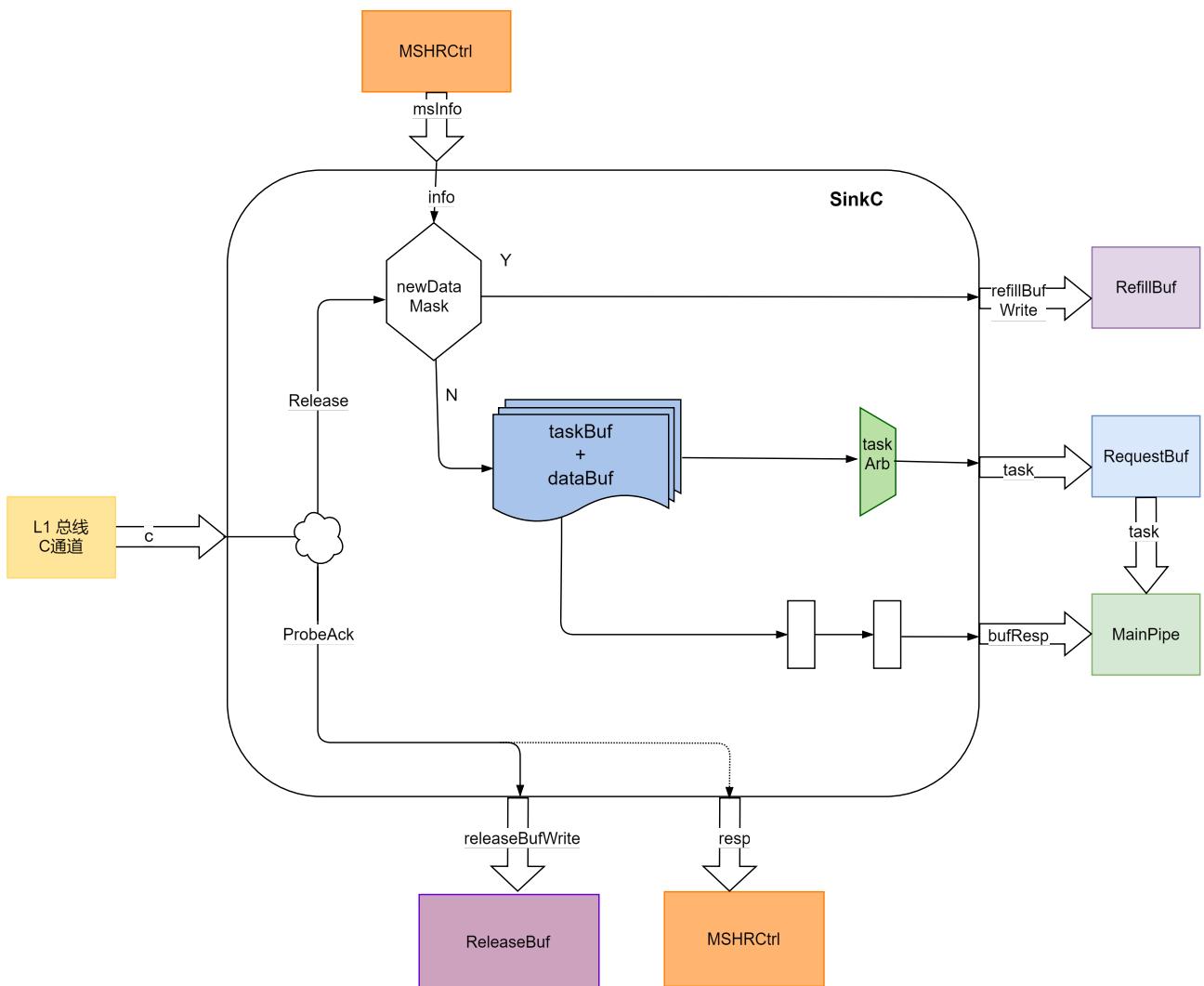


图 20.9: SinkC

20.7.3 GrantBuffer

20.7.3.1 基本功能

GrantBuf 接收来自 MainPipe 的任务，并根据任务类型进行转发。主要分为：- 预取响应(opcode = HintAck)，会安排其进入预取响应队列 pftRespQueue(size=10)，按 FIFO 顺序向预取器发出。- D 通道响应 (opcode = Grant/GrantData/ReleaseAck)，会安排其进入 grantQueue(size=16)，按 FIFO 顺序向总线 D 通道发出；如果是 Grant/GrantAck，则还需要将其信息进入到 inflightGrant 缓冲区 (size=16)（表明 Grant 已发送但是还没收到 GrantAck），等待 L1 从 E 通道返回 GrantAck 后再将信息清除掉。- 融合请求 (task.mergeA = true)，同时执行以上 2 种。

20.7.3.1.1 特性 1：阻塞 MainPipe 入口

GrantBuf 还会根据：【流水线入口信息 + 流水线是 S1/S2/S3/S4/S5 级状态 + 内部 pftRespQueue, inflightGrant, grantQueue 状态】，来向 ReqArb 给出【请求入口的阻塞信息】。

3 种资源的统计：- GrantBuf 资源不足：已占用的 GrantBuf 数量 + 流水线 S1/S2/S3/S4/S5 中是可能 GrantBuf 的数量 (from sinkA or sinkC) > 16 - E 通道资源不足：inflightGrant + 流水线 S1/S2/S3/S4/S5 中可能需要 E 通道回 GrantAck 的数量 (from sinkA) > 16 - prefetch 的 RespQueue 资源不足：已占用的 pftRespQueue 数量 + 流水线 S1/S2/S3/S4/S5 中可能用到 preRespQueue 数量 (from sinkA) > 10

在 S1 阻塞通道进入 MainPipe 的条件 - A 通道：以上任意一种资源不足 - B 通道：只要 inflightGrant 缓冲区有和 B 通道地址一样的未完成操作 - C 通道：GrantBuf 资源不足

MSHR 的 3 种资源不足 (资源最大数-1)：- GrantBuf 的资源不足：已占用的 GrantBuf 数量 + 流水线 S1/S2/S3/S4/S5 中是可能 GrantBuf 的数量 (from sinkA or sinkC) > 15 - E 通道资源不足：inflightGrant + 流水线 S1/S2/S3/S4/S5 中可能需要 E 通道回 GrantAck 的数量 (from sinkA) > 15 - Prefetch 的 RespQueue 资源不足：已占用的 pftRespQueue 数量 + 流水线 S1/S2/S3/S4/S5 中可能用到 preRespQueue 数量 (from sinkA) > 9

阻塞 MSHR 进入 Mainpipe 的阻塞条件为以上 3 种任意一种

20.7.3.1.2 特性 2：提前唤醒

MainPipe 中的 CustomL1Hint 模块会在 GrantBuf 发出前 3 拍发出 l1Hint 信号用于 L1D\$ 中 missq 的提前唤醒。GrantBuffer 负责给出资源信息在 S1 阻塞流水线入口，而 MainPipe 会根据已经进入流水线中场景精确预测何时发出 l1Hint 信号。

20.7.3.1.3 特性 3：不同数据宽度处理

对于包含一个 beat 的 Grant/ReleaseAck，从 grantQueue 出队直接向总线发出。对于包含两个 beat 的 GrantData，在从 grantQueue 出队的时候，第一个 beat 的数据直接向总线发出，同时将第二个 beat 的数据存入 grantBuf；接下来优先发送 grantBuf 里的数据。grantBuf 发送完后，可以出队下一个 grantQueue 元素。

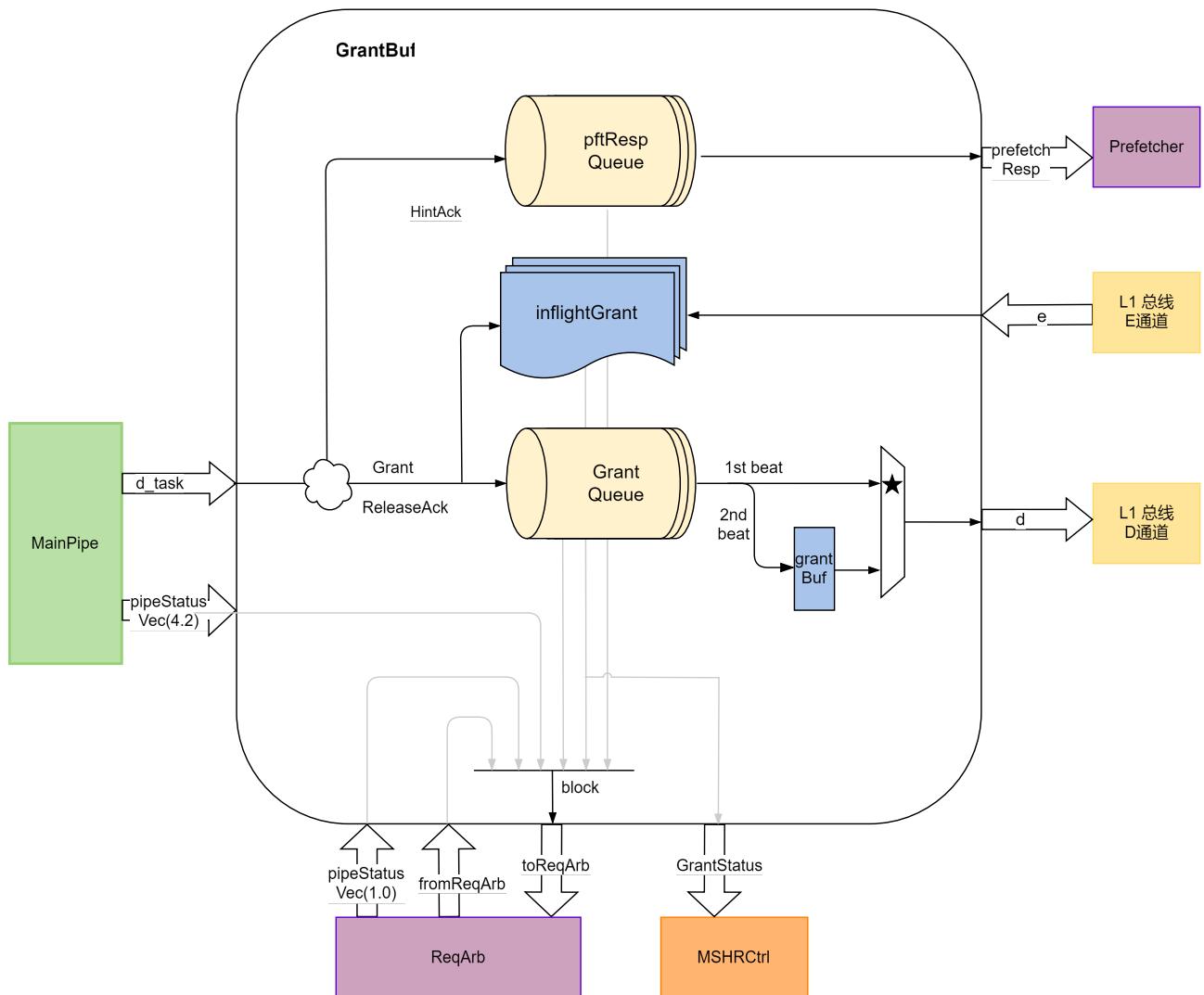


图 20.10: GrantBuffer

20.7.3.2 整体框图

20.8 下游 CHI 总线通道

20.8.1 TXREQ

20.8.1.1 功能描述

TXREQ 模块接收来自 MainPipe 和 MSHR 两个模块发往 REQ 通道的请求，在两者之间进行仲裁，并用队列进行缓冲，最终发送到 CHI 的 TXREQ 总线通道上。来自 MainPipe 出口的请求会被无条件接收，来自 MSHR 的请求有可能会被阻塞。因此 TXREQ 模块需要对 MSHR 进行反压，同时对 MainPipe 入口进行流控，以保证 MainPipe 上的请求能够不被阻塞地进入 TXREQ。

20.8.1.2 功能描述

20.8.1.2.1 特性 1：对 MainPipe 入口的流控

为了保证 MainPipe 上的请求能够非阻塞地在 s3/s4/s5 进入 TXREQ，当 `inflight = MainPipe` 上 `s1/s2/s3/s4/s5` 有可能需要进入 TXREQ 的请求数 + 队列中的有效项数 \geq 队列总项数 (`size=16`) 时，TXREQ 模块需要对 MainPipe 入口即 s0 级进行反压（由于只有 MSHR 才会往下游 TXREQ 发送请求，MSHR task 是从 s0 进入流水线的，所以只需要对 s0 的 MSHR 请求做反压）。由于 s1 的时序比较紧，对于 MainPipe 上 s1 的可能用到 TXREQ 的处理是：先认为 s1 都会用到 TXREQ，s2 发现没有用到就把 `inflight` 数 -1.

姑且将阻塞条件记为 `noSpace`

20.8.1.2.2 特性 2：对 MSHR 的反压逻辑

1. MainPipe 的仲裁优先级大于 MSHR，所以 MainPipe 出口的请求有效时，需要给 MainPipe 反压。
2. 当 `noSpace` 的时候需要给 MainPipe 反压，原因如下：MSHR 发出请求的当拍 MainPipe 可能没有请求和 MSHR 竞争，但是 MainPipe 中有请求还在 s1/s2 级，MSHR 请求有可能抢占了队列中本属于 MainPipe 的空闲项，导致 MainPipe 中的请求到达 s3/s4/s5 级时队列项数不够。所以这种情况也需要阻塞住 MainPipe 的请求。

20.8.1.3 整体框图

20.8.2 RXRSP

20.8.2.1 功能描述

接受来自 RXRSP 通道的无数据的响应消息，直接把响应送到 MSHRCtl，用消息中 `txnid` 来识别 `mshrID`。
CHI.IssueB 需要处理的响应包括：`Comp`, `CompDBIDResp`, `Retry`, `PCrdGrant`. CHI.IssueC 需要响应：`RespSepData`

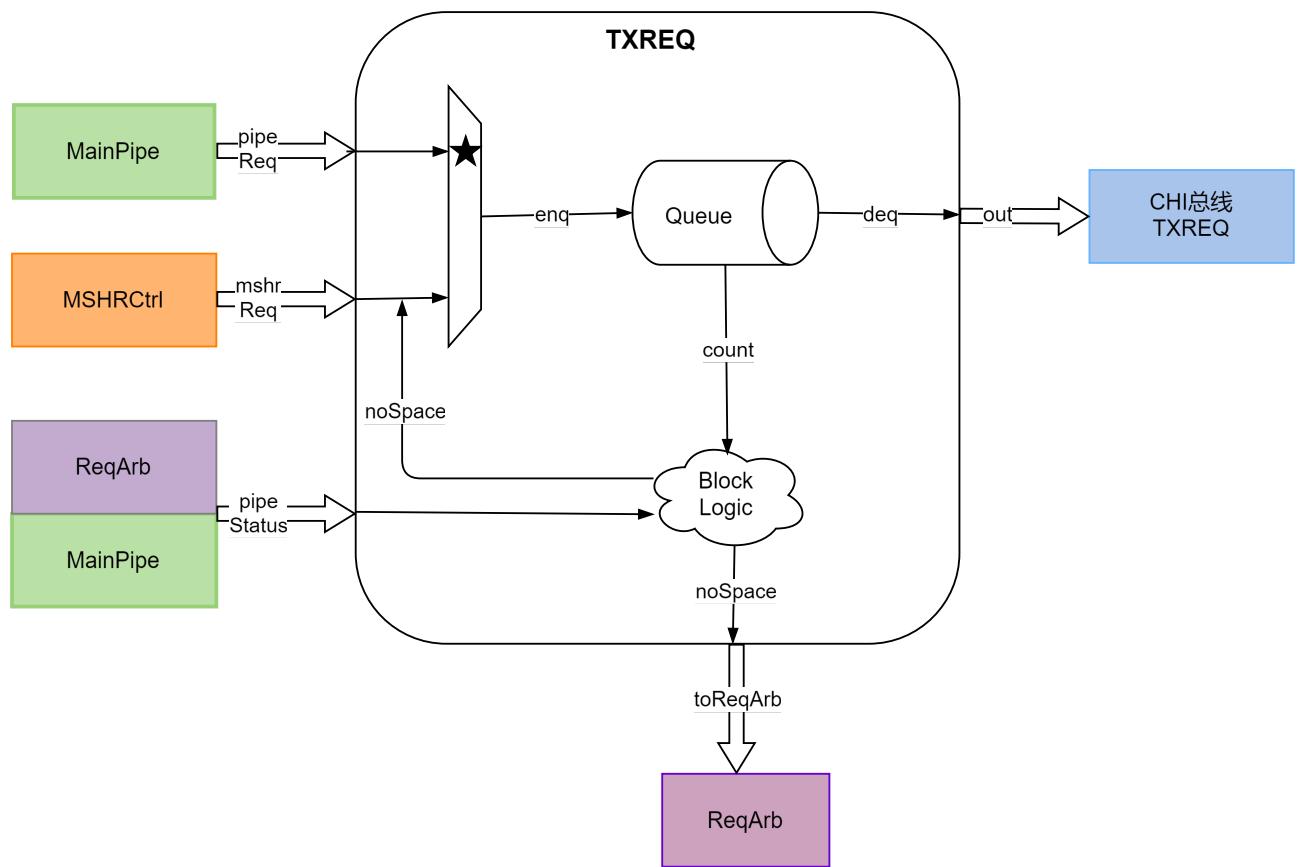


图 20.11: TXREQ

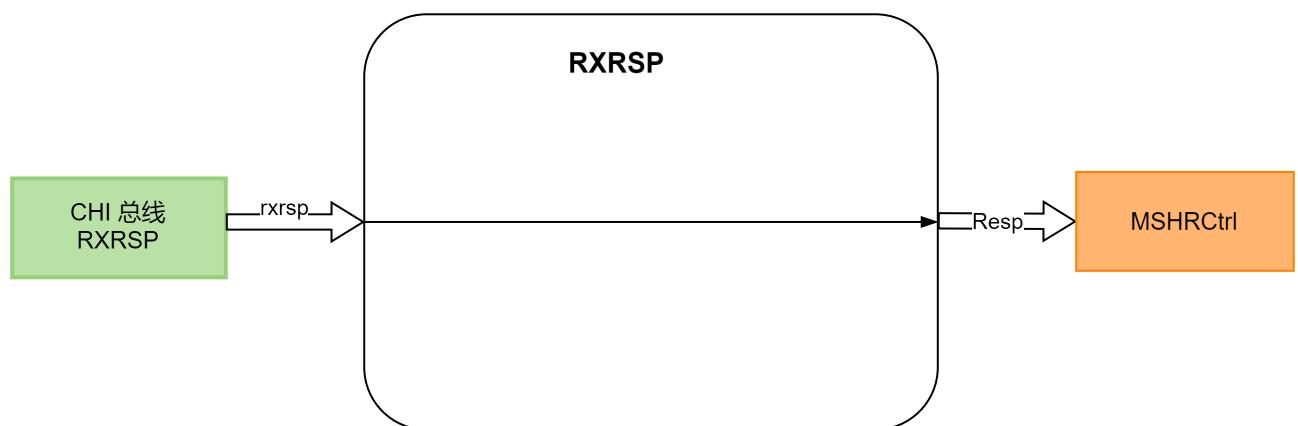


图 20.12: RXRSP

20.8.2.2 整体框图

20.8.3 RXDAT

20.8.3.1 功能描述

接受来自 RXDAT 通道的有数据的响应消息，将数据存入 RefillBuffer，同时把响应送到 MSHRCtrl，用消息中 txnID 用来识别 mshriID。CHI.IssueB 需要处理的响应包括：CompData。CHI.IssueC 需要处理的响应包括：DataSepResp

20.8.3.2 整体框图

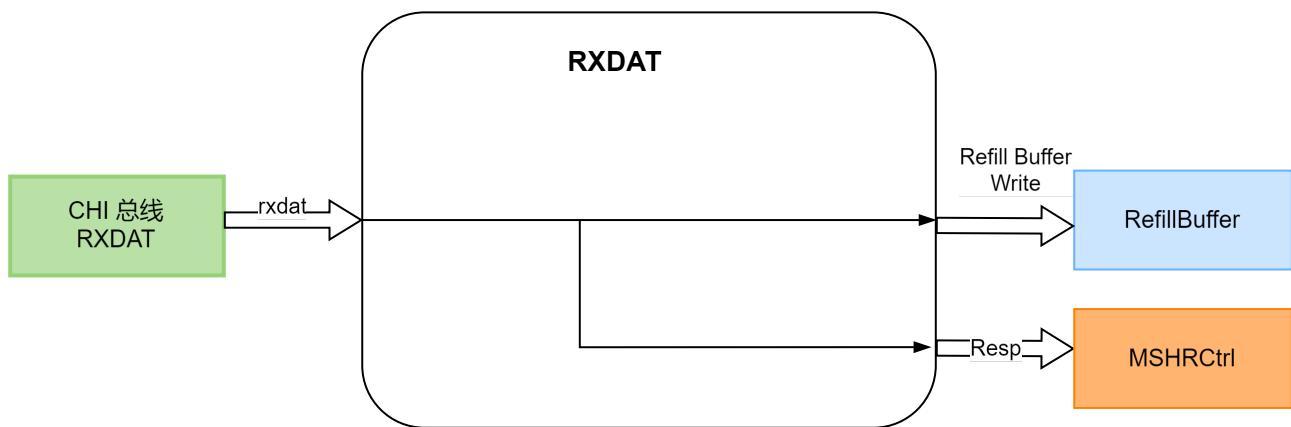


图 20.13: RXDAT

20.8.3.3 接口时序

接收请求的当拍通知 MSHR，如果是第一个 beat 则锁存起来，如果是第二个 beat 则和第一个 beat 的数据组合起来并在当拍写入 RefillBuf。

20.8.4 RXSNP

20.8.4.1 功能描述

RXSNP 模块把来自 RXSNP 总线通道的 Snoop 请求进行处理，转化为内部任务格式，然后发给 RequestArb。同时如果 MSHR 中有正在进行符合一下条件的，则阻塞 RXSNP 总线进入 SinkB：- 地址相同，且不能被这笔 Snoop 请求嵌套 - 准备替换的数据块与这笔 Snoop 请求地址相同且不能被这笔 Snoop 嵌套

20.8.4.1.1 特性 1: Snoop(addr=X) 何时嵌套正在处理的相同地址 MSHR 正在进行的一笔读操作 (addr=X)?

1. 在 MSHR 收到第一笔响应数据前，相同地址的 Snoop 有更高优先级，应该先做。
2. 在 MSHR 收到第一笔响应数据后，相同地址的 Snoop 应该等待这笔 MSHR 做完后再被响应。
3. 在 MSHR 送出 WriteBackFull/Evict 要把回填数据写入 DS 前，Snoop 应该被阻塞，因为 Snoop 需要的数据仍然在 refillBuffer 里，而不是 DS 中。
4. 在 MSHR 送出 WriteBackFull/Evict 已经把回填数据写入 DS，Snoop 应该优先被响应，因为 Snoop 有更高优先级。

20.8.4.1.2 特性 2: Snoop(addr=Y) 何时嵌套正在进行替换的 MSHR (被替换的行地址 addr=Y)?

1. 当 MSHR 确定那一路被替换并且这个替换产生的向 Core 的 probe 没有完成时, 这个 Snoop 被阻塞, 因为这笔 Snoop 有可能产生相同地址的一笔相同 probe 到 Core。
2. 如果 MSHR 的所有向 Core 的 probe 都完成了, Snoop 应该嵌套 MSHR。
3. 当 MSHR 是一笔 CMO 操作, 在产生的所有 Probe 操作完成前 Snoop 都被屏蔽。

20.8.4.2 整体框图

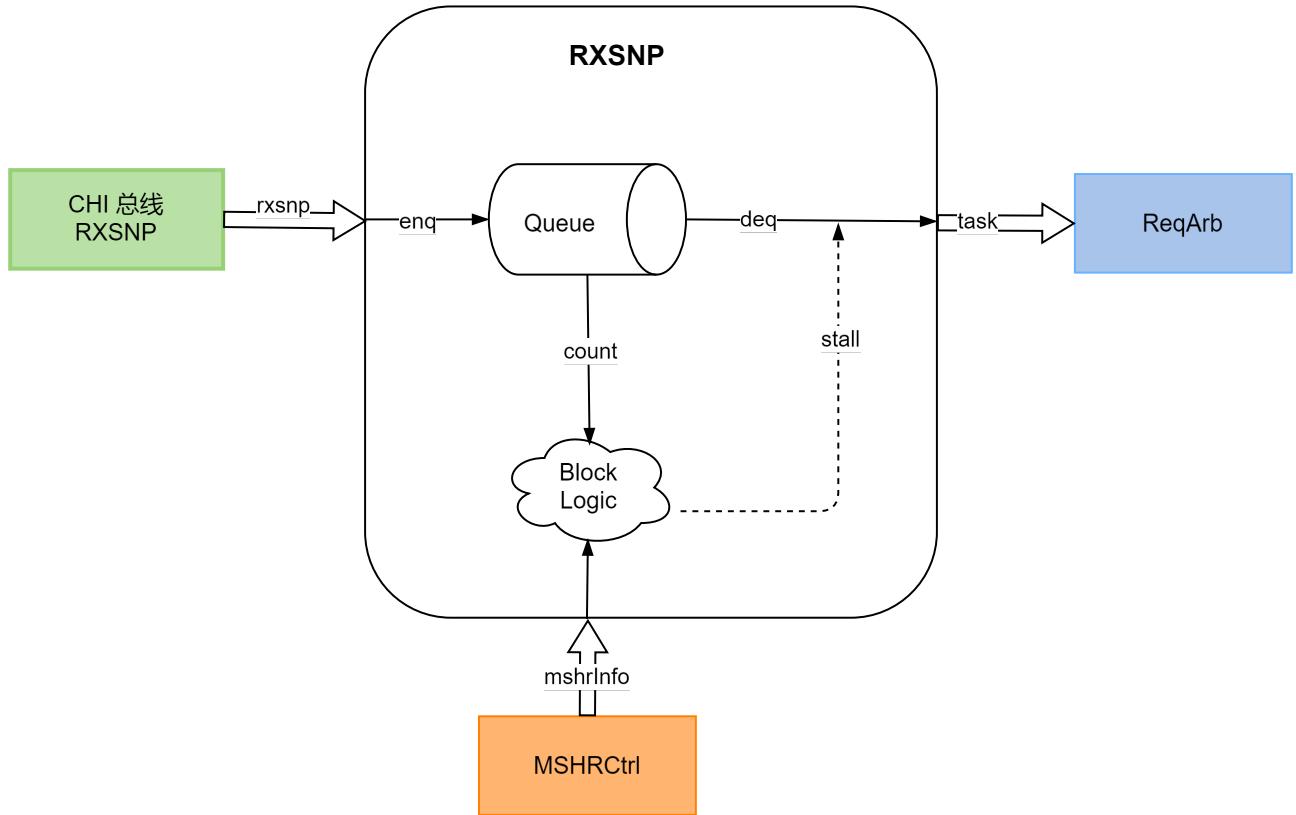


图 20.14: RXSNP

20.8.5 TXDAT

20.8.5.1 功能描述

TXDAT 模块无条件接收来自 MainPipe 发往 WDAT 通道的请求，并用队列进行缓冲，最终发送到 CHI 的 TXDAT 总线通道上。TXDAT 模块需要对 MainPipe 入口进行流控，以保证 MainPipe 上的请求能够不被阻塞地进入 TXDAT。

20.8.5.2 功能描述

20.8.5.2.1 特性 1: 对 MainPipe 的反压

为了保证 MainPipe 上的请求能够非阻塞地在 s3/s4/s5 进入 TXDAT，当【MainPipe 上有可能需要进入 TXDAT 的请求数 + 队列中的有效项数 \geq 队列总项数】时，TXDAT 模块需要对 MainPipe 入口即 s0/s1 级进行反

压。1. 对 s1 级反压是因为 RXSNP 收到的 snoop 可能会直接在 MainPipe 上完成处理，然后进入 TXDAT 通道，所以需要对 s1 的 sinkB 请求做反压 2. 对 s0 级反压是因为一部分 MSHR task 需要进入 TXDAT 通道，MSHR task 是在 s0 进入流水线的，所以需要对 s0 的 mshrtask 做反压

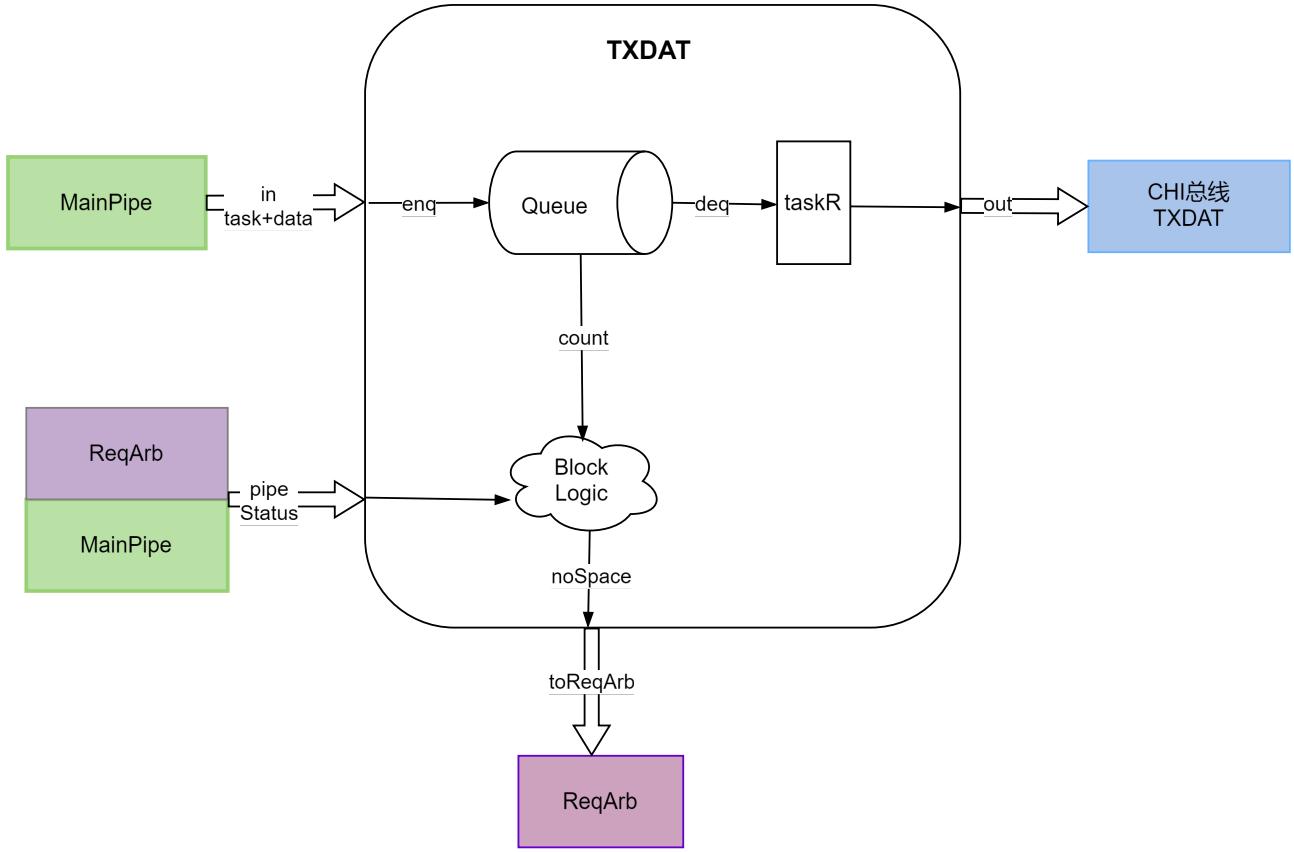


图 20.15: TXDAT

20.8.6 TXRSP

20.8.6.1 功能描述

TXRSP 模块接收来自 MainPipe 和 MSHR 两个模块发往 TXRSP 通道的请求，在两者之间进行仲裁，并用队列进行缓冲，最终发送到 CHI 的 TXRSP 总线通道上。其中来自 MainPipe 出口的请求会被无条件接收，来自 MSHR 的请求有可能会被阻塞。因此 TXRSP 模块需要对 MSHR 进行反压，同时对 MainPipe 入口进行流控，以保证 MainPipe 上的请求能够不被阻塞地进入 TXRSP。

20.8.6.1.1 特性 1：对 MainPipe 入口的流控

- 为了保证 MainPipe 上的请求能够非阻塞地在 s3/s4/s5 进入 TXRSP，当【MainPipe 上有可能需要进入 TXRSP 的请求数 + 队列中的有效项数 \geq 队列总项数】时，TXREQ 模块需要对 MainPipe 入口即 s0/s1 级进行反压。
- 其中，对 s1 级反压是因为 RXSNP 收到的 snoop 可能会直接在 MainPipe 上完成处理，然后进入 TXRSP 通道，所以需要对 s1 的 sinkB 请求做反压；对 s0 级反压是因为一部分 MSHR task 需要进入 TXRSP 通道，MSHR task 是在 s0 进入流水线的，所以需要对 s0 的 mshrtask 做反压。姑且将阻塞条件记为 noSpace。

20.8.6.1.2 特性 2：对 MSHR 的反压逻辑

1. MainPipe 的仲裁优先级大于 MSHR，所以 MainPipe 出口的请求有效时，需要给 MainPipe 反压。
2. 当 noSpace 的时候需要给 MainPipe 反压，原因如下：MSHR 发出请求的当拍 MainPipe 可能没有请求和 MSHR 竞争，但是 MainPipe 中有请求还在 s1/s2 级，MSHR 请求有可能抢占了队列中本属于 MainPipe 的空闲项，导致 MainPipe 中的请求到达 s3/s4/s5 级时队列项数不够。所以这种情况也需要阻塞住 MainPipe 的请求。

20.8.6.2 整体框图

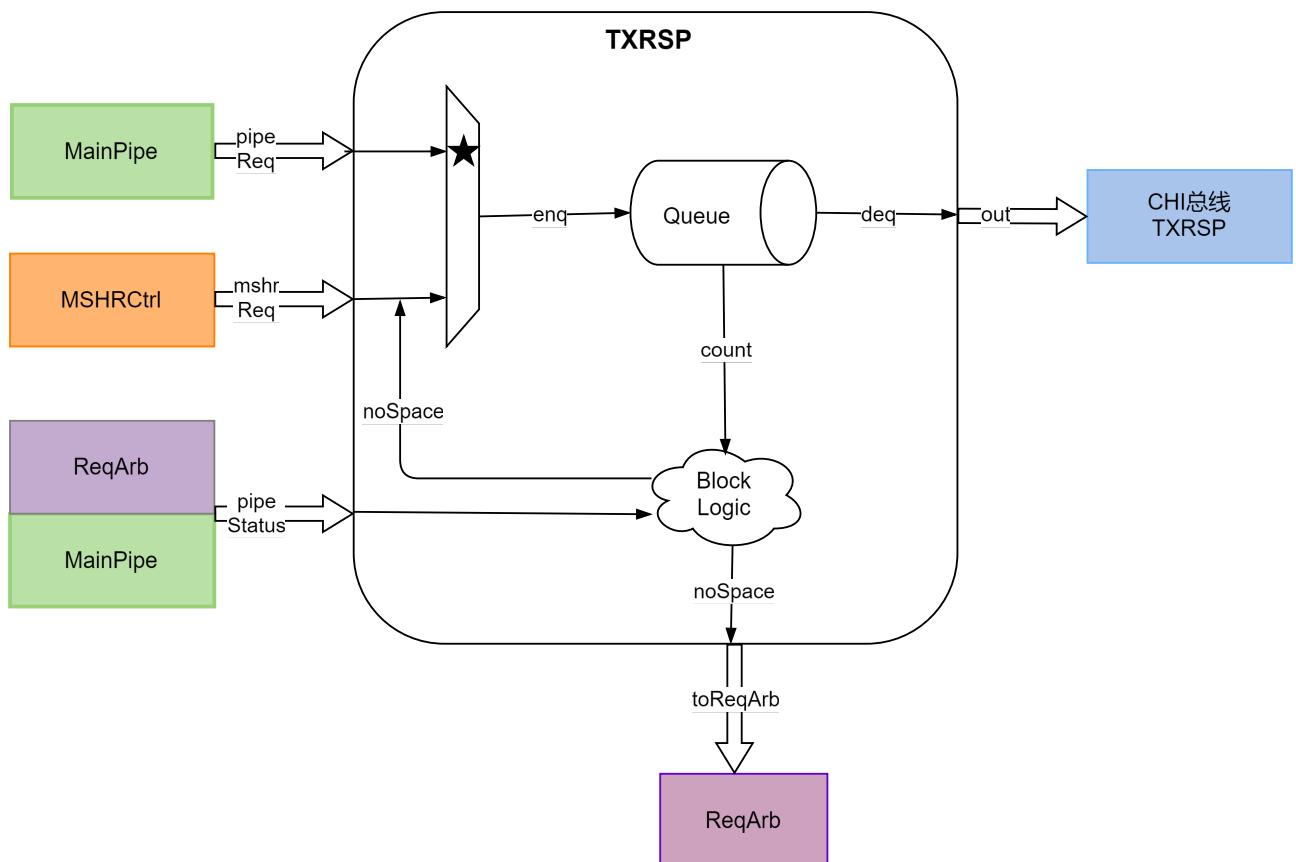


图 20.16: TXRSP

20.8.7 P-Credit 管理机制

20.8.7.1 功能描述

根据 CHI 协议 2.3.2 关于 Retry 的描述，P-Credit 管理遵循以下规则：

1. 当 RXRSP 通道收到 PCrdGrant 时，会有一个 CAM 记录下这笔操作中的 PCrdType 和 SrcID
2. 此时如果有某个 Slice 中有 MSHR 正在等待相同类型 {PCrdType,SrcID} 的 PCredit，则把这个 PCredit 分配给这个 Slice -如果多个 Slice 同时命中，则按 RoundRobin 分配这笔 PCredit，同时删除这笔 PCredit 在 CAM 中的记录。-如果没有 Slice 命中，CAM 则保存 PCrdType 和 SrcID 后续使用 (协议允许 PCrdGrant 先于 RetryAck 发出)
3. 对于命中的 Slice，如果有多个 MSHR 命中 {PCrdType,SrcID}，则按 RoundRobin 分配到某一个 MSHR 中。
4. 对于每个 MSHR：-如果收到 RetryAck 时会保存 PType 和 SrcID，同时拉高表示正在等待 PCredit 的 pValid 信号通知 CAM -如果在 CAM

中找到匹配的 PCredit，则 MSHR 完成操作拉低 pValid 并删除 CAM 中匹配的项。如果在 CAM 中没有找到匹配的 PCredit，则 pValid 一直拉高直到收到响应的 PCredit。

20.8.8 链路层控制器 LinkMonitor

20.8.8.1 功能描述

LinkMonitor 模块将基于 Valid-Ready 握手的消息转化为基于 L-Credit 的握手，同时会维护 TX、RX 两条方向的 Link 的功耗状态。具体内容详见 CHI Spec Link Handshake 一章。

20.8.8.1.1 特性 1：Decoupled 握手转 L-Credit 握手

从三个 TX 通道接出的 Decoupled 握手经过 Decoupled2LCredit 模块转化为 LCredit 握手。Decoupled2LCredit 模块会记录下游 ICN 收到的 LCredit 数量 (lcreditPool)，当 lcreditPool 大于 0 时，才能接收上游的 Decoupled 请求；Decoupled 请求握手成功时，lcreditPool 数量减一。

TX 链路状态的影响：如果 TX 链路状态为 STOP 或者 ACTIVATE 时，应该停止接收 Decoupled 消息。如果 TX 链路状态为 STOP 时，应当停止接收 LCredit，即便下游 lcrdv 信号拉高 lcreditPool 也应该保持不变。

20.8.8.1.2 特性 2：L-Credit 握手转 Decoupled 握手

从三个 RX 通道收到的 LCredit 握手经过 LCredit2Decoupled 模块转化为 Decoupled 握手。LCredit2Decoupled 模块会维护一个默认 4 项 (lcreditNum 可配，要求 lcreditNum ≤ 15) 的队列用于暂存消息，即一个 RX 通道最多向下游发送 lcreditNum 个 outstanding 的 LCredit；同时会维护一个初始值为 lcreditNum 的计数器 (lcreditPool)，用于记录当前通道最多可以发送多少 LCredit。当 lcreditPool > 队列有效项项数 (queueCnt) 时，说明该通道已经发出的 outstanding LCredit 数量小于该通道队列所能接收的消息数量，此时该通道可以向下发送 LCredit；当 lcreditPool < lcreditNum 时，该通道应该无条件接收下游的有效请求，即 fltv 拉高且上一拍 fltpending 拉高的请求。

RX 链路状态的影响：如果 RX 链路状态不是 RUN，则该通道不应该向下游发送 LCredit，即便 lcreditPool > 队列有效项项数。

20.8.8.1.3 特性 3：TXSACTIVE 与 RXSACTIVE

TXSACTIVE 永远拉高。RXSACTIVE 暂时没有用到。

20.8.8.1.4 特性 4：Interface activation and deactivation

TXLINKACTIVEREQ 在复位后就一直拉高。RXLINKACTIVEACK 在 RXLINKACTIVEREQ 置 true 后的下一拍置 true；RXLINKACTIVEREQ 置 false 的下一拍开始侦听三条 RX 通道的状态，当各个 RX 通道都收回所有 outstanding 的 LCredit 后（即 lcreditPool 等于 lcreditNum），RXLINKACTIVEACK 即可置 false。

20.8.8.2 整体框图

20.9 MMIO 转接桥 MMIOBridge

MMIOBridge 独立于 CoupledL2 的 4 个 Slice 之外，接收来自 IFU / LSU 的 MMIO 与 Uncache 请求，通过 CHI 总线与下游 NoC / LLC 进行交互并完成外设的读写操作。默认 MMIOBridge 包含 8 项 MMIOBridgeEntry，

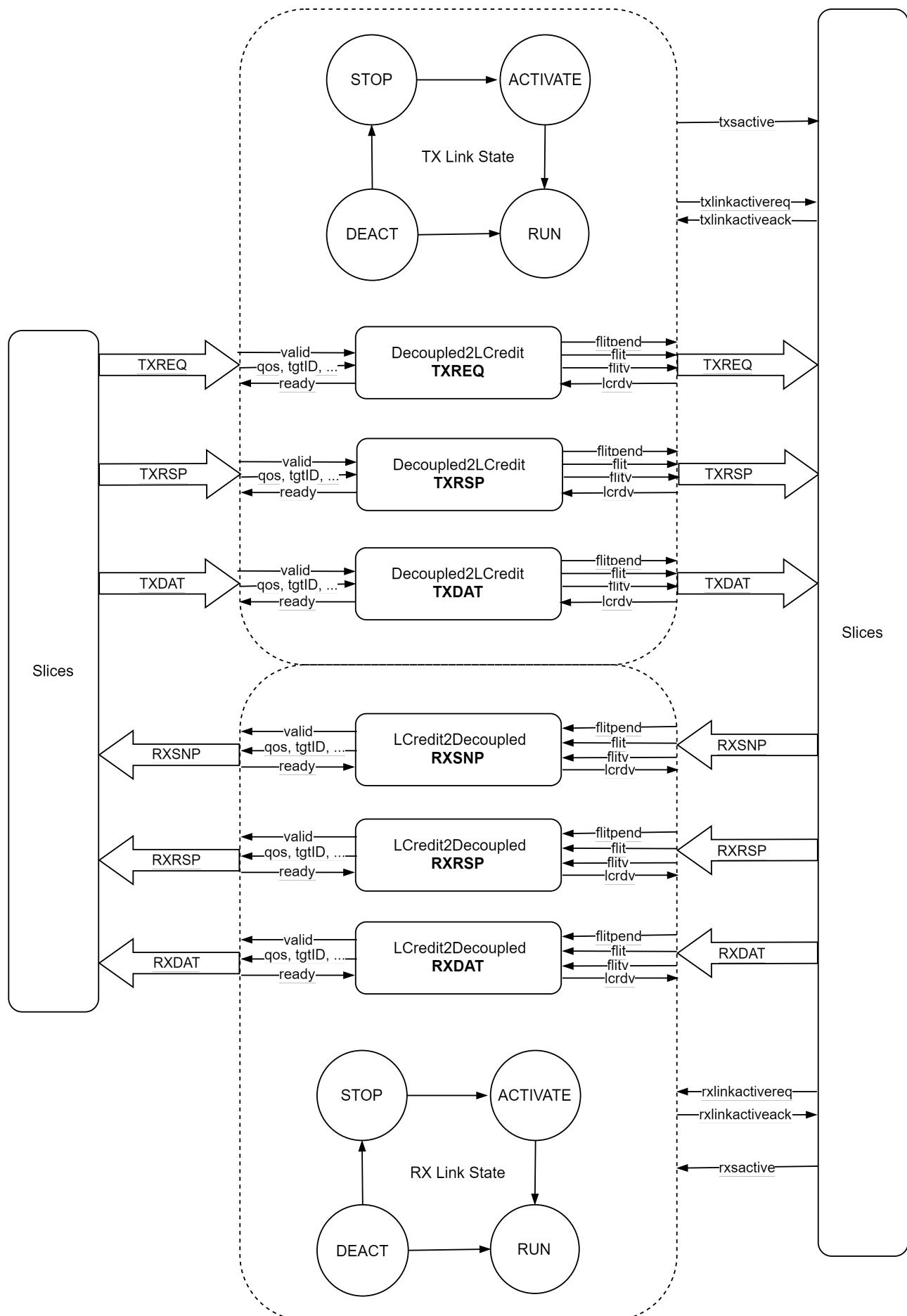


图 20.17: LinkMonitor

每一项 MMIOBridgeEntry 都是独立的状态机，用于处理一条 MMIO 或 Uncache 请求。

20.9.1 状态机

状态机项主要分为两类：

- Schedule 状态项
- Wait 状态项

Schedule 状态项又称主动动作状态项，主要用来跟踪 MMIOBridgeEntry 主动向上游 TileLink 总线、下游 CHI 总线发送任务与请求的情况。其值为低有效，表示未完成状态，即任务尚未成功离开 MMIOBridgeEntry 并被发出，其原因可能是未完成阻塞条件（有必要的前置动作未完成）或通道阻塞；值为高则表示对应任务已经成功发出，或不需要发出任务。

Wait 状态项又称被动动作状态项，主要用来跟踪 MMIOBridgeEntry 期望收到的来自下游 CHI 通道、上游 TileLink 通道的回复。其值为低有效，表示未完成状态，即对应的回复尚未回到当前 entry；值为高则表示对应回复已经收到，或不需要收到回复。

状态寄存器包括：

| 名称 | 描述 |
|---------------|--|
| s_txreq | 向下游 TXREQ 通道发送 ReadNoSnp / WriteNoSnpPtl 请求 |
| s_ncbwrdata | (如果是写操作) 向下游 TXDAT 发送 NCBWrData 数据包，发送 NCBWrData 的前提是 w_dbidresp 拉高 |
| s_resp | CHI 读 / 写事务已完成，向上游 TileLink D 通道返回 AccessAckData / AccessAck 响应 |
| w_comp | (如果是写操作) 等待下游 RXRSP 通道返回的 Comp / CompDBIDResp 响应 |
| w_dbidresp | (如果是写操作) 等待下游 RXRSP 通道返回的 CompDBIDResp / DBIDResp / DBIDRespOrd 响应 |
| w_compdata | (如果是读操作) 等待下游 RXDAT 通道返回的 CompData 读 / 写请求发生协议层重传，需要等待下游返回 PCrdGrant |
| w_pcrdgrant | |
| w_readreceipt | (如果是读请求且 Order 不是 None) 等待下游 RXRSP 通道返回的 ReadReceipt |

20.9.2 定序

MMIOBridge 发起的 CHI 请求默认 Order 为 RequestOrder 或 EndpointOrder。核内在发起 TileLink 读 / 写请求时，会在 A 通道的自定义域中带上 **PMA 属性是否为 Memory**：

- 如果该地址的 PMA 属性是 Memory，但是 PBMT 属性为 IO 或 NC，发起的 CHI 事务的 Order 为 RequestOrder
- 如果该地址的 PMA 属性是 Memory，但是 PBMT 属性为 IO 或 NC，发起的 CHI 事务的 Order 为 EndpointOrder

无论是 RequestOrder 还是 EndpointOrder，发起的 ReadNoSnp 都需要下游返回 ReadReceipt 来对读操作定序。因此 MMIOBridge 会侦听各个 entry 是否在等待 ReadReceipt (即 w_readreceipt 是否拉低)，如果某个 entry 在等待 ReadReceipt 那么所有 entry 都不能向下发送新的 ReadNoSnp。

20.9.3 内存属性

CHI 总线协议定义了如下 4 个维度的内存属性：

- Allocate
- Cacheable
- Device
- EWA (Early Write Acknowledgment)

对于 MMIOBridge 发起的请求, Allocate 和 Cacheable 总是 0; Device 取决于 PMA 属性, 如果 PMA 属性为 Memory 则 Device 为 0, 否则 Device 为 1; EWA 可以理解为该地址能否被 Buffer, 如果该地址的 PMA 属性为 Memory, 或者 PMA 属性不是 Memory 但是 PBMT 属性为 NC, 则认为该地址可以被 Buffer, EWA 置为 1, 否则 EWA 置 0。

20.9.4 P-Credit 仲裁

MMIOBridge 支持 CHI 总线的协议层重传。根据 CHI 总线协议, 一个事务在收到 RetryAck 和 PCrdGrant 后才能发起协议层重传。其中 RetryAck 的 TxnID 和 TXREQ 通道的请求的 TxnID 一致, 因此 RetryAck 可以通过 TxnID 和 MMIOBridge 中的 entry 一一对应。然而 PCrdGrant 只要 SrcID 和 PCrdType 两个字段和 RetryAck 一致即可匹配, 无法通过 TxnID 直接和 MMIOBridge 中的 entry 做匹配。

基于上述 CHI 协议的规定, CoupledL2 在收到 PCrdGrant 时, 无法直接根据 PCrdGrant 的字段判断该 P-Credit 应该仲裁给 MMIOBridge 还是 4 个 Slice, 更无法判断如何将 P-Credit 对应到某一项 MMIOBridgeEntry 或某一项 MSHR。因此 P-Credit 的仲裁逻辑位于 CoupledL2 的顶层。当收到 PCrdGrant 时 CoupledL2 会将其暂存在顶层的寄存器中(主要记录 SrcID 和 PCrdType 等关键字段), 收到 RetryAck 的 MSHR / MIOBridgeEntry 会用 RetryAck 的 SrcID 和 PCrdType 与 PCrdGrant 寄存器堆进行匹配, 如果匹配到, 寄存器堆的这一项会被释放, 如果没有匹配到会一直等待下游的 PCrdGrant 直到匹配成功。

20.10 错误处理

- 版本: V2R2
- 状态: OK
- 日期: 2025/04/24

20.10.1 术语说明

| 缩写 | 全称 | 描述 |
|---------------|--------------------------|--------------|
| ICache/I\$ | Instruction Cache | L1 指令缓存 |
| DCache/D\$ | Data Cache | L1 数据缓存 |
| L1 Cache/L1\$ | Level One Cache | L1 缓存 |
| L2 Cache/L2\$ | Level Two Cache | L2 缓存 |
| L3 Cache/L3\$ | Level Three Cache | L3 缓存 |
| BEU | Bus Error Unit | 总线错误单元 |
| MMIOBridge | Memory-Mapped I/O Bridge | 内存映射 I/O 转接桥 |

| 缩写 | 全称 | 描述 |
|--------|--|----------------|
| ECC | Error Correction Code | 错误校验码 |
| SECDED | Single Error Correct Double Error Detect | 单比特纠错双比特校验 |
| TL | Tile Link | Tile Link 总线协议 |
| CHI | | CHI 总线协议 |

20.10.2 设计规格

- 支持 ECC 校验
- 支持 CHI DataCheck
- 支持 CHI Poison

20.10.3 Cached 访存请求错误处理

基本的错误处理逻辑：由检测到错误的 Cache Level 进行错误上报；保存/传播地址对应错误状态

1. L2 Cache 将在 L2 Cache 检测到的 ECC/DataCheck Error 上报至 BEU，由 BEU 触发中断向软件报告错误
2. 对于来自 L1/L3 Cache 的请求，L2 Cache 会根据检测到的错误类型在通信中通知 L1/L3 Cache
3. 对于来自 L1/L3 Cache 的错误数据，L2 Cache 会将错误类型记录在 meta 中

20.10.3.1 ECC

20.10.3.1.1 ECC 校验码

L2 Cache 目前默认的 ECC 校验码为 SECDED。同时，L2 Cache 支持 parity、SEC 等校验码，可在 Configs 中修改，编译时进行配置。相关校验算法参考

SECDED 要求对于一个 n 位的数据，所需的校验位数 r 需要满足： $2^r \geq n + r + 1$

20.10.3.1.2 ECC 处理流程

L2 Cache 支持 ECC 功能。在 MainPipe 在 s3 向 Directory 和 DataStorage 重填数据时，会计算 tag 和 data 的校验码，前者与 tag 一起存入 Directory 中的 tagArray (SRAM)，后者与 data 一起存入 DataStorage 中的 array (SRAM)

1. 对于 tag，直接以 tag 为单元进行 ECC 编码/解码。
2. 对于 data，基于物理设计以及更好检测错误的需求，目前将 data 划分成 dataBankBits (128 bits) 的单元进行 ECC 编码/解码。因而在 SECDED 算法要求下，对于 1 个 512 bits 的 cache line，应该有 $4 * 8 = 32$ bits 校验位

当访存请求读取 SRAM 时，会同步读取出对应的校验码。MainPipe 会在 s2 和 s5 分别获得 tag 和 data 的校验结果。当 MainPipe 检验到错误后，会在 s5 收集错误信息，CoupledL2 仲裁各个 Slice 错误信号，并上报至 BEU

20.10.3.2 总线端口

20.10.3.2.1 TL 总线

当 L2 Cache 接收来自 L1/L3 Cache 的数据时，若检测到错误 (denied/corrupt = 1)，则 MainPipe 在 s3 写 Directory 时，将对应 meta 中 tagErr/dataErr 置为 1

当 L2 Cache 向 L1/L3 传输数据时，若 L2 Cache 检测到 ECC 错误或者对应 meta 中 tagErr/dataErr = 1，则将对应通道（如 D 通道 GrantBuffer）信号中 denied/corrupt 置为 1；否则均置为 0

- 特别的，对于 TL D 通道返回数据时，若 denied = 1，则需要将对应 corrupt 也置为 1；在当前设计下，L2 Cache 不应认为 L1 Cache 持有对应数据 copy（L1 Cache 在后续 Release 时，会直接丢弃对应 copy）
- 特别的，由于 TL C 通道中只有 corrupt 域而不存在 denied 域。故使用 opcode 域用于辅助区分 denied/corrupt。如 SinkC 中

```
task.corrupt := c.corrupt && (c.opcode === ProbeAckData || c.opcode === ReleaseData)
task.denied := c.corrupt && (c.opcode === ProbeAck || c.opcode === Release)
```

20.10.3.2.2 CHI 总线

L2 Cache 支持可配置的 Poison/DataCheck: - Poison 域: - DAT 中每 8 bytes 设置 1 bit Poison 位 - L2 Cache 中 Poison 采用 over poison 策略 - Poison 错误 L2 Cache 不进行上报

- DataCheck 域:
 - DAT 中每 8 bits 设置 1 bit DataCheck 位
 - L2 Cache 中 DataCheck 默认采用奇校验
 - L2 Cache 中 DataCheck 仅对 data 进行校验，不对 packet 整体进行校验
 - DataCheck 校验错误由 L2 Cache 进行上报

当 L2 Cache 接收来自 L3 Cache 的数据时，若检测到错误：

1. respErr = NDERR，则不会将对应数据写入 L2 Cache，但会完成其余流水线处理（例如，对于来自 L1 Cache 的 Acquire 请求，L2 Cache 将会返回数据并将 denied 和 corrupt 置 1）
2. respErr = NDERR/DERR 或者 poison 域中任意一位为 1 或者 dataCheck 奇校验检验出错误时，则 MainPipe 在 s3 写 Directory 时，将对应 meta 中 dataErr 置为 1
3. dataCheck 检验出错误，则复用 ECC 错误上报流程，MainPipe 在 s5 收集错误信息后，上报 BEU

当 L2 Cache 向 L3 Cache 传输数据时：

1. 若 L2 Cache 检测到 tag ECC 错误或者对应 meta 中 tagErr = 1，则将 respErr 置为 NDERR，将 poison 置为全 0
2. 若 L2 Cache 检测到 data ECC 错误或者对应 meta 中 dataErr = 1，则将 respErr 置为 DERR，并将 poison 域置为全 1
3. 若 L2 Cache 检测到 data ECC 错误或者对应 meta 中 tagErr = 1 且 dataErr = 1，则将 respErr 置为 NDERR，将 poison 置为全 1
4. 若 L2 Cache 未检测到任何错误，则将 respErr 置为 OK，将 poison 置为全 0
5. dataCheck 域填充对 data 进行奇校验的校验码

- 在当前版本中，L2 支持的 Write/Snoop transactions 在相关的 data packet 传输中均不允许 respErr 为 NDERR（故 TXDAT 中 respErr 实际只会为 DERR 或 OK）

一致性状态处理 (RN 接收到包含 NDERR 请求):

1. 对于分配事务，L2 将正常处理流水线，但是不回将包含 NDERR 请求的相关数据写回 Directory 或者 DataStorage，缓存状态不变（具体相关事务类型为 ReadClean, ReadNotSharedDirty, ReadShared, ReadUnique, CleanUnique, MakeUnique）
2. 对于释放事务，L2 正常处理（具体相关事务类型为 WriteBack, WriteEvictFull, Evict, WriteEvictOrEvict）
3. 对于 Snoop，L2 probe L1 (ToN)，回复 SnpResp_I 以及 NDERR，一律不 forward (不回复 CompData)，暂不将 L2 对应缓存行置为 Invalid
4. 对于其他事务，L2 保证相应数据缓存状态不升级（当前版本下，由 1 可保证）

20.10.4 Uncached 访存请求错误处理

CoupledL2 中 MMIOBridge 会将 TL 与 CHI 之间的错误处理相关域进行转换，但不会进行任何错误上报。CHI to TL (RXDAT/RXRSP)

1. 若 respErr = NDERR，则置 denied 为 1
 2. 若 respErr = NDERR/DERR 或者 poison 域中任意一位为 1 或者 dataCheck 奇校验检验出错误时，则置 corrupt 为 1
 3. 否则，denied 与 corrupt 均置为 0
- 特别的，对于 RXRSP (如 Comp)，由于 TL-SPEC 要求部分类型响应 (如 AccessAck) 中 corrupt 必须为 0，故当 respErr = NDERR/DERR 时，均置 denied 为 1
 - 当出现错误时，后续由 ICache 或 DCache 触发 Hardware Error，上报软件处理

TL to CHI (TXDAT)

1. 当 corrupt = 1 时，则置 respErr 为 DERR，置 poison 为全 1
2. 当 corrupt = 0 时，则置 respErr 为 OK，置 poison 为全 0
3. dataCheck 域填充对 data 进行奇校验的校验码