# Revision History

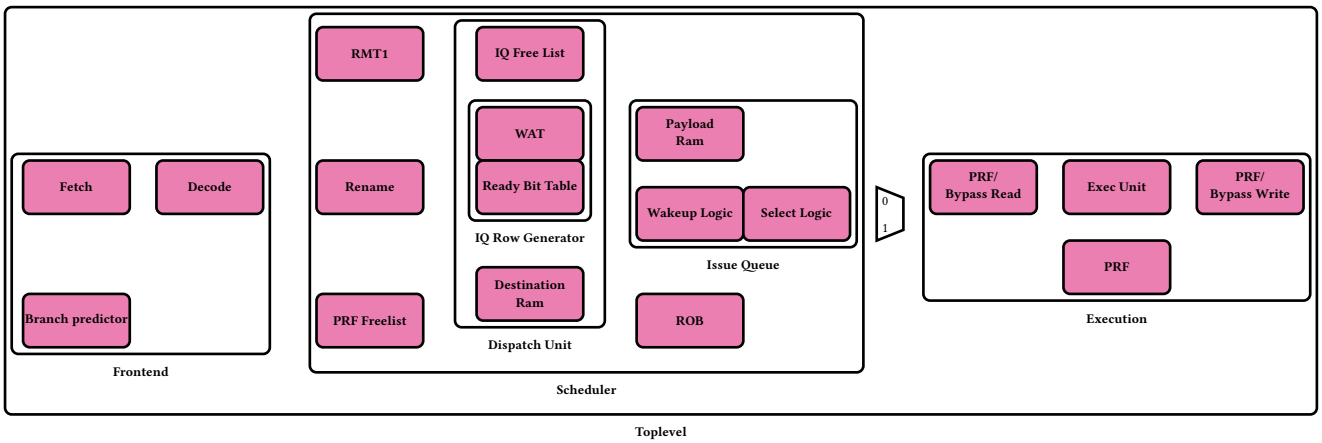
| Author | Description | Date |
|---|---|---|
| 刘恒雨 | Initialised and updated spec doc | 2025/04/10 |

# Contents

# 1 Terminology

**PRF** Physical Register File
**WAT** Wakeup Allocation Table
**RMT** Register Map Table
**SDA** Store Data Array
**STQ** STore Queue
**LDQ** LoaD Queue

# 2 Overview




# 3 Parameters

## 3.1 Parameters > label(module)

# 4 Interface

## 4.1 BP_IFU_Interface > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **PC_cur** | `Input(UInt(p.XLEN.W))` | 当前 IFU 的 PC 值 |
| **PC_target** | `Output(UInt(p.XLEN.W))` | 预测的下个 cycle 取指的目标地址 |
| **BTB_Hit** | `Output(Vec(p.FETCH_WIDTH, Bool()))` | 1 代表 hit，0 相反；将最年轻的命中 BTB 的置为 1，其余为 0 |
| **BHT_Taken** | `Output(Bool())` | branch 指令的 BHT 的预测结果；1 代表跳转，0 相反 |
| **GHR** | `Output(UInt(p.GHR_WIDTH.W))` | 作出预测时的全局历史寄存器快照 |

### 4.2 BP_ROB_Interface > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **PC** | `Input(UInt(p.XLEN.W))` | 当前 ROB 的 PC 值 |
| **instrType** | `Input(UInt(3.W))` | 当前指令类型,该模块需要区分条件分支和无条件分支 |
| **BTB_Hit** | `Input(Bool())` | 该分支指令最初是否命中 BTB |
| **actual_Taken** | `Input(Bool())` | 实际是否 taken |
| **GHR** | `Input(UInt(p.GHR_WIDTH.W))` | 作出预测时的全局历史寄存器快照，使得更新 BHT 时能够生成正确的 index |
| **actualTargetPC** | `Input(UInt(p.XLEN.W))` | 目标地址 |

### 4.3 BP_IO > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **instrAddr** | `Input(UInt(p.XLEN.W))` | 当前 IFU 的 PC 值 |
| **PC_target** | `Output(UInt(p.XLEN.W))` | 预测的下个 cycle 取指的目标地址 |
| **BTB_Hit** | `Output(Vec(p.FETCH_WIDTH, Bool()))` | 1 代表 hit，0 相反；将最年轻的命中 BTB 的置为 1，其余为 0 |
| **BHT_Taken** | `Output(Bool())` | branch 指令的 BHT 的预测结果；1 代表跳转，0 相反 |
| **GHR** | `Output(UInt(p.GHR_WIDTH.W))` | 作出预测时的全局历史寄存器快照 |
| **rob_commitsignal** | `Valid(Vec(p.DISPATCH_WIDTH, UInt((37 + ((34 + p.GHR_WIDTH) max (37 + log2Ceil(p.PRF_DEPTH))))).W))).flip` | |

### 4.4 Fetch_IO > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **instAddr** | `Output(UInt(p.XLEN.W))` | 当前 IFU 的 PC 值 |

| | | |
|---|---|---|
| **instr** | `Input(Vec(p.FETCH_WIDTH, UInt(32.W)))` | |
| **id_uop** | `Decoupled(Vec(p.FETCH_WIDTH, new IF_ID_uop()))` | |
| **rob_commitsignal** | `Valid(Vec(p.DISPATCH_WIDTH, UInt((37 + ((34 + p.GHR_WIDTH) max (37 + log2Ceil(p.PRF_DEPTH))))).W))).flip` | |
| **PC_target** | `Input(UInt(p.XLEN.W))` | 预测的下个 cycle 取指的目标地址 |
| **BTB_Hit** | `Input(Vec(p.FETCH_WIDTH, Bool()))` | 1 代表 hit，0 相反；将最年轻的命中 BTB 的置为 1，其余为 0 |
| **BHT_Taken** | `Input(Bool())` | branch 指令的 BHT 的预测结果；1 代表跳转，0 相反 |
| **GHR** | `Input(UInt(p.GHR_WIDTH.W))` | 作出预测时的全局历史寄存器快照 |
| **id_uop** | `Decoupled(Vec(p.FETCH_WIDTH, new IF_ID_uop()))` | |

### 4.5 Dispatch_issue_interface > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **dis_valid** | `Output(Vec(p.DISPATCH_WIDTH, Bool()))` | 来自 Dispatch Unit 的有效信号 |
| **dis_uop** | `Output(Vec(p.DISPATCH_WIDTH, new uop()))` | 来自 Dispatch Unit 的输入 |

### 4.6 issue_exu_interface > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **dst_FU** | `Output(Vec(p.ISSUE_WIDTH, UInt(log2Ceil(p.FU_NUM).W)))` | 发射的指令的目标功能单元 |
| **issue_uop** | `Output(Vec(p.ISSUE_WIDTH, new uop()))` | 发射的指令 |
| **issue_uop_valid** | `Output(Vec(p.ISSUE_WIDTH, Bool()))` | 发射的指令的有效信号 |

### 4.7 issue_lsu_interface > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **dst_FU** | `Output(Vec(p.ISSUE_WIDTH, UInt(log2Ceil(p.FU_NUM).W)))` | 发射的指令的目标功能单元 |
| **issue_uop** | `Output(Vec(p.ISSUE_WIDTH, new uop()))` | 发射的指令 |
| **issue_uop_valid** | `Output(Vec(p.ISSUE_WIDTH, Bool()))` | 发射的指令的有效信号 |

### 4.8 iq_freelist_update > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **iq_freelist_id** | `Input(Vec(p.DISPATCH_WIDTH, UInt(log2Ceil(p.IQ_DEPTH).W)))` | IQ Freelist ID |

### 4.9 RenameUnit_IO > label(module)

重命名单元将逻辑寄存器地址映射成实际寄存器。逻辑寄存器指的是 ISA 定义的 x0-x31，而实际寄存器数量多于 32 个，一般可达 128 个。主要解决 WAW，WAR 等问题。

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **id_uop** | `Decoupled(Vec(p.RENAME_WIDTH, new ID_RENAME_uop())).flip` | |
| **rob_commitsignal** | `Valid(Vec(p.DISPATCH_WIDTH, UInt((37 + ((34 + p.GHR_WIDTH) max (37 + log2Ceil(p.PRF_DEPTH))))).W))).flip` | |
| **dispatch** | `Decoupled(Vec(p.DISPATCH_WIDTH, new RENAME_DISPATCH_uop()))` | |

### 4.10 Dispatch_ROB_Interface > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **dis_uops** | `Valid(Vec(p.DISPATCH_WIDTH, new uop()))` | Dispatch Unit 的 uop |
| **rob_empty** | `Input(Bool())` | ROB 空标志(0 表示空，1 表示非空) |
| **rob_head** | `Input(UInt(log2Ceil(p.ROB_DEPTH)))` | ROB 头指针 |
| **rob_tail** | `Input(UInt(log2Ceil(p.ROB_DEPTH).W))` | ROB 尾指针 |

## 4.11 WB_ROB_Interface > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **complete_map** | `Input(Vec(p.FU_NUM, Bool()))` | 完成映射表 |
| **complete_uop** | `Input(Vec(p.FU_NUM, new uop()))` | 来自 exu 的 uop |
| **mispred** | `Input(Bool())` | 分支误预测信号 |
| **if_jump** | `Input(Bool())` | 分支指令跳转信号 |

## 4.12 ROB_broadcast > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **commit_signal** | `Valid(Vec(p.DISPATCH_WIDTH, UInt((37 + ((34 + p.GHR_WIDTH) max (37 + log2Ceil(p.PRF_DEPTH))))).W))` | ROB 条目 |

## 4.13 ROBIO > label(module)

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **dis_uops** | `Valid(Vec(p.DISPATCH_WIDTH, new DISPATCH_ROB_uop()))` | Dispatch Unit 的 uop |
| **rob_empty** | `Input(Bool())` | ROB 空标志(0 表示空，1 表示非空) |
| **rob_head** | `Input(UInt(log2Ceil(p.ROB_DEPTH)))` | ROB 头指针 |
| **rob_tail** | `Input(UInt(log2Ceil(p.ROB_DEPTH).W))` | ROB 尾指针 |
| **ALU_complete_uop** | `Flipped(Valid((Vec(p.ALU_NUM, new ALU_WB_uop()))))` | 来自 alu 的 uop |
| **BU_complete_uop** | `Flipped(Valid((Vec(p.BU_NUM, new BU_WB_uop()))))` | 来自 bu 的 uop |
| **jal** | `Bool()` | |
| **jalr** | `Bool()` | |
| **STU_complete_uop** | `Flipped(Valid(new STPIPE_WB_uop()))` | 来自 stu 的 uop |
| **LDU_complete_uop** | `Flipped(Valid(new LDPIPE_WB_uop()))` | 来自 ldu 的 uop |
| **mispred** | `Input(Bool())` | 分支误预测信号 |

| | | |
|---|---|---|
| **if_jump** | `Input(Bool())` | 分支指令跳转信号 |
| **commit_signal** | `Valid(Vec(p.DISPATCH_WIDTH, UInt((37 + ((34 + p.GHR_WIDTH) max (37 + log2Ceil(p.PRF_DEPTH))))).W)))` | ROB 条目 |

## 4.14 ALUIO > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **in1** | `Input(UInt(p(XLen).W))` | |
| **in2** | `Input(UInt(p(XLen).W))` | |
| **fn** | `Input(UInt(4.W))` | |
| **out** | `Output(UInt(p(XLen).W))` | |
| **cmp_out** | `Output(Bool())` | |

## 4.15 BypassNetworkIO > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **exec_units** | `Input(Vec(2, Valid(new BypassInfo)))` | |
| **preg_rd** | `Input(UInt(p(PhysRegIdxSz).W))` | |
| **data_out** | `Output(UInt(p(XLen).W))` | |

## 4.16 DCacheRequest > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **addr** | `UInt(p.CoreMaxAddrbits.W)` | |
| **data** | `Bits(p.CoreDataBits.W)` | |

## 4.17 DCacheResponse > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **addr** | `UInt(p.CoreMaxAddrbits.W)` | |
| **data** | `Bits(p.CoreDataBits.W)` | |

### 4.18 LSUMemIO > label(module)

描述与数据内容交互的各种信号，用于管理 L/S 请求和缓存访问，包含一部分异常处理、顺序控制的功能

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **req** | `new DecoupledIO(p.lsuWidth,Valid(new DCacheRequest))` | LSU 发出的数据缓存请求 |
| **resp** | `new Flipped(p.lsuWidth,Valid(new DCacheResponse))` | LSU 接收数据缓存响应 |
| **req_kill** | `Output(p.lsuWidth,Bool())` | 表示每个 req 是否被 kill |
| **req_nack_adv** | `Input(p.lsuWidth,Bool())` | 表示某个 req 是否被拒绝 |
| **store_ack** | `Flipped(Vec(p.lsuWidth,new ValidIO(new DCacheRequest)))` | 存储请求的确认 |
| **nack** | `Flipped(Vec(p.lsuWidth,new ValidIO(new DCacheRequest)))` | 作为接口接受 neck |
| **load_rel_resp** | `Flipped(new DecoupledIO(new DCacheResponse))` | 作为接口接受缓存的 load/release 响应 |
| **bradate** | `Output(new bradateInfo)` | 报告分支更新信息 |
| **exception** | `Output(Bool())` | 输出异常 |
| **rob_pnr_idx** | `Output(UInt((p.robAddrSz).W))` | |
| **rob_head_idx** | `Output(UInt((p.robAddrSz).W))` | rob 中的后备和头部索引 |
| **release** | `Flipped(new DecoupledIO(new TLBundle(edge.bundle)))` | 处理缓存协议的释放操作 |
| **force_order** | `Output(Bool())` | 强制顺序控制，在保证 l/s 顺序的时候激活 |
| **order** | `Input(Bool())` | 顺序控制信号，表示当前是否满足顺序要求 |

### 4.19 LSU_Issue_IO > label(module)

与 Issue 的 IO 接口，主要用于接收来自 Issue 的 load 和 store 指令，并将其传递给 LSU 的其他模块进行处理

IO 端口定义如下：

| Name | Type | Description |
| --- | --- | --- |
| **store_issue_uop** | `Flipped(Decoupled(new uop()))` | |
| **store_value_i1** | `Input(UInt(p.XLEN.W))` | 存储指令的操作数 1 |
| **store_value_i2** | `Input(UInt(p.XLEN.W))` | 存储指令的操作数 2 |

| | | |
|---|---|---|
| **load_issue_uop** | `Flipped(Decoupled(new uop()))` | |
| **load_value_i1** | `Input(UInt(p.XLEN.W))` | 加载指令的操作数 1 |
| **load_value_i2** | `Input(UInt(p.XLEN.W))` | 加载指令的操作数 2 |

### 4.20 STQ_Dispatcher_IO > label(module)

向 dispatcher 发送 STQ 的头尾指针，方便后续 store，load 指令的调度
STQ 的头尾指针主要用于存储指令的调度和执行

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **stq_tail_ptx** | `Output(log2Ceil(p.STQ_Depth).W)` | stq 的尾部索引 |
| **stq_head_ptx** | `Output(log2Ceil(p.STQ_Depth).W)` | stq 的头部索引 |
| **store_dis_uop** | `Flipped(Valid(Vec(p.DISPATCH_WIDTH,new uop())))` | |
| **stq_empty** | `Output(Bool())` | stq 是否为空 |

### 4.21 LSU_ROB_IO > label(module)

接收来自 ROB 的 CommitSignal 信号，用于执行后续入 STQ 的操作

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **store_signal** | `Input(Vec(p.DISPATCH_WIDTH,` | |

### 4.22 LSU_Broadcast > label(module)

广播信号，store 完成信号主要是给 ROB 使用
load 完成信号则提供给 PRF 跟 ROB

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **store_finish** | `Valid((new uop()))` | 存储完成的信号 |
| **load_finish** | `Valid((new uop()))` | 加载完成的信号 |

### 4.23 LSUIO > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **lsu_broadcast_commit** | `new LSU_Broadcast` | LSU 的提交信号 |
| **lsu_rob** | `new LSU_ROB_IO` | LSU 与 ROB 的交互信号 |
| **lsu_issue** | `new LSU_Issue_IO` | LSU 与 issue 的交互信号 |

| | | |
|---|---|---|
| **stq_dispatch** | `new STQ_Dispatch_IO` | LSU 与 stq 的交互信号 |

# 5 Microarchitecture

本处理器为乱序执行多发射 RV32IM 架构，其设计主要借鉴于 RSD 以及 BOOM。

## 5.1 Core > label(module)

Top Level Structure

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **mem** | `new MemInterface` | |

## 5.2 BranchPredictor > label(module)

## 5.3 Fetch > label(module)

## 5.4 PayloadRAM > label(module)

PayloadRAM >是一个 RAM 模块，用于存储指令的有效载荷。

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **we** | `Input(Vec(p.DISPATCH_WIDTH, Bool()))` | 写使能信号 |
| **waddr** | `Input(Vec(p.DISPATCH_WIDTH, UInt(log2Ceil(NUM_PayloadRAM >).W)))` | 写地址 |
| **wdata** | `Input(Vec(p.DISPATCH_WIDTH, UInt(Insr_WIDTH.W)))` | 写数据 |
| **raddr** | `Input(Vec(p.ISSUE_WIDTH, UInt(log2Ceil(NUM_PayloadRAM >).W)))` | 读地址 |
| **rdata** | `Output(Vec(p.ISSUE_WIDTH, UInt(Insr_WIDTH.W)))` | 读数据 |

## 5.5 WakeupLogic > label(module)

WakeupLogic >是一个唤醒逻辑模块，用于处理指令的唤醒信号。它根据指令的依赖关系和状态，决定哪些指令可以被唤醒。

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **stall** | `Input(Bool())` | 停顿信号 |
| **write** | `Input(Vec(p.DISPATCH_WIDTH, Bool()))` | 写使能 |
| **writePtr** | `Input(Vec(p.DISPATCH_WIDTH, UInt(p.ISSUE_QUEUE_INDEX_WIDTH.W)))` | 写指针 |

| | | |
|---|---|---|
| **writeSrcTag** | `Input(Vec(p.DISPATCH_WIDTH, new SrcTag()))` | 写源标签,SrcTag 可能包含寄存器标签与寄存器地址 |
| **writeDstTag** | `Input(Vec(p.DISPATCH_WIDTH, new DstTag()))` | 写目标标签,DstTag 可能包含寄存器标签 |
| **wakeup** | `Input(Vec(p.WAKEUP_WIDTH, Bool()))` | 唤醒信号 |
| **wakeupDstTag** | `Input(Vec(p.WAKEUP_WIDTH, new DstTag()))` | 唤醒目标标签 |
| **wakeupVector** | `Input(Vec(p.WAKEUP_WIDTH + p.ISSUE_STORE_WIDTH, Vec(p.ISSUE_QUEUE_ENTRY_NUM, Bool())))` | 唤醒向量 |
| **notIssued** | `Input(Vec(p.ISSUE_QUEUE_ENTRY_NUM, Bool()))` | 未发射标志 |
| **dispatchStore** | `Input(Vec(p.DISPATCH_WIDTH, Bool()))` | 是否是 Store |
| **dispatchLoad** | `Input(Vec(p.DISPATCH_WIDTH, Bool()))` | 是否是 Load |
| **memDependencyPred** | `Input(Vec(p.DISPATCH_WIDTH, Bool()))` | 内存依赖预测 |
| **opReady** | `Output(Vec(p.ISSUE_QUEUE_ENTRY_NUM, Bool()))` | 操作就绪标志 |

### 5.6 Rename > label(module)

### 5.7 ROB > label(module)

重命名缓冲区，主要用于存储指令的执行结果。它是一个 FIFO 结构，先进先出。指令在执行完成后，将结果写入 ROB >中。ROB >中的数据可以被其他指令读取，从而实现数据的共享和重用。

### 5.8 ALU > label(module)

### 5.9 BypassNetwork > label(module)

### 5.10 ExecutionUnit > label(module)

IO 端口定义如下：

| Name | Type | Description |
|---|---|---|
| **kill** | `Input(Bool())` | Killed upon misprediction/exception |
| **branch_update** | `Input(new BrUpdateInfo)` | |

**issued_uop** `Input(Valid(new uop()))`

### 5.11 LSU > label(module)

LSU >的模块定义，目前只完成了 IO 接口的定义，内部逻辑还未完成