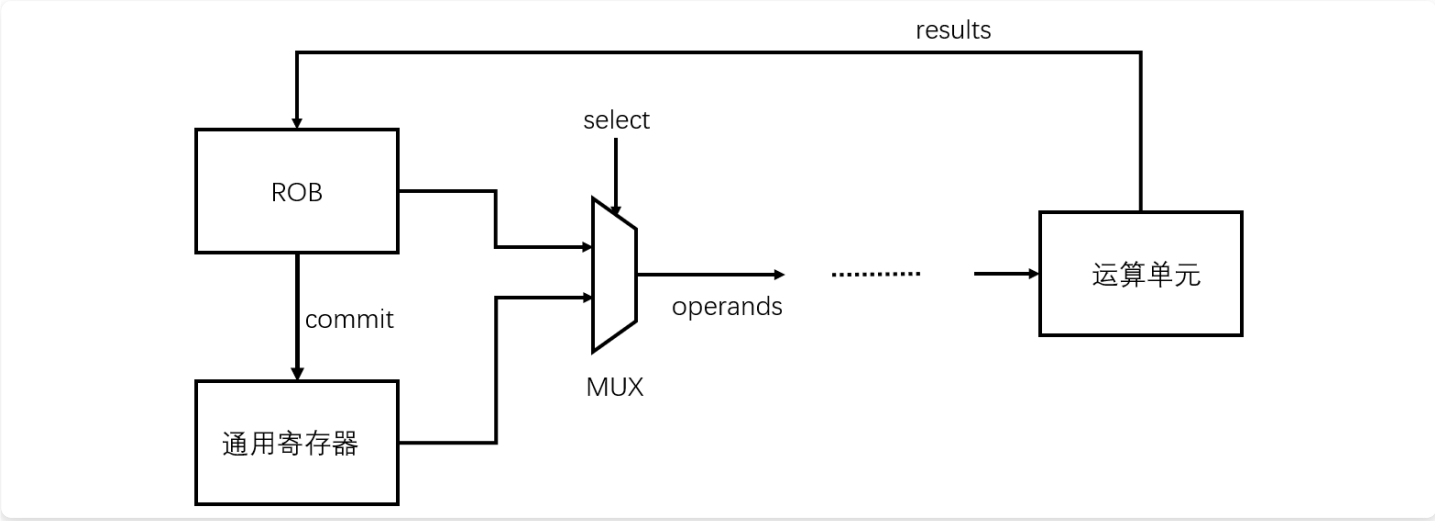
BOOM微架构学习——详解重排序缓存

超标量处理器在每个时钟周期内能够完成多条指令，其主要特征是指令多发射和乱序执行。所谓多发射，指流水线每个时钟周期可以发射多条指令，提高并行度，将 IPC(Instruction Per Cycle) 提高到 1 以上。乱序执行即不按照原始指令的顺序执行，克服指令间不必要的优先性限制，减少流水线停顿向后传递引起的气泡。在本专栏以往的文章中，我们介绍了实现乱序执行的关键技术——寄存器重命名，本文将结合BOOM处理器源码，介绍超标量处理器中的另一重要组成部分：**重排序缓存(Reorder Buffer)**。

为什么需要重排序缓存？

重排序缓存机制是为了实现指令顺序提交、精确异常。重排序缓存将跟踪流水线中已发射的指令，记录指令是否执行完毕、是否异常等信息，顺序提交执行完毕的指令，将指令结果保存到寄存器堆中。重排序缓存机制使处理器实现了精确的异常处理和有效的硬件猜测执行。

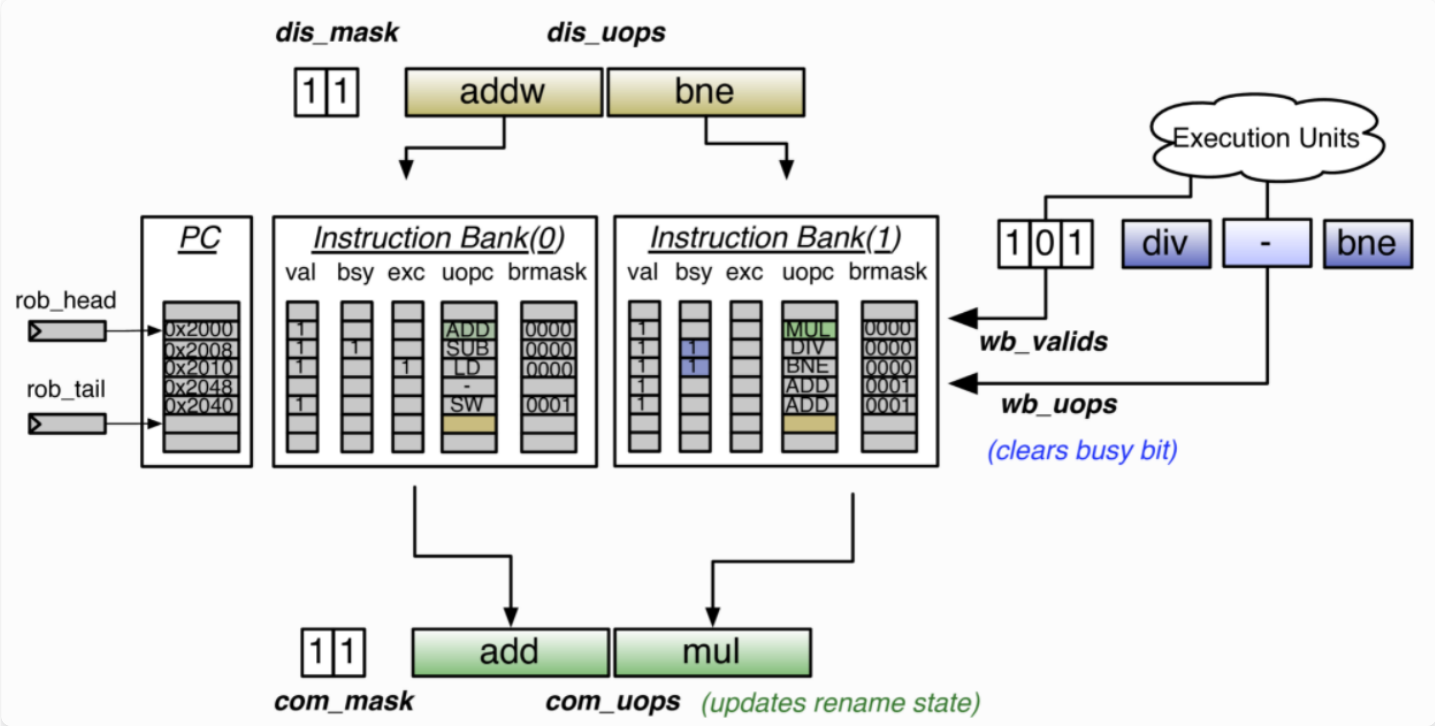
在隐式重命名机制中，ROB 还需负责保存指令运算的结果，并通过旁路实现数据的前馈，但由于BOOM处理器采用的显式重命名，指令运算结果保存在物理寄存器堆中，ROB 不需要实现数据前馈功能。



隐式重命名方案中ROB实现数据前馈

BOOM 的重排序缓存概述

在 BOOM 处理器中，指令经过译码和重命名后被派遣到发射队列(Issue Queue)和重排序缓存(Reorder Buffer)。ROB将跟踪指令在流水线中的状态，当位于ROB“头”(head)的指令执行完毕且无异常时，ROB将提交(commit)该指令，并将结果更新到对应逻辑寄存器中。



BOOM的重排序缓存

BOOM 重排序缓存的每一个 bank 的每一行保存一条指令的相关信息。由于每次派遣到 ROB 的一组指令的地址一定是连续的，只需要保存每行ROB中 bank(0) 的地址，节省硬件开销。在派遣预测跳转的分支指令或跳转指令时，分支指令将单独作为一行进行派遣，保证每行 ROB 中指令地址的连续性。发生跳转的分支/跳转指令会导致 ROB 行出现“气泡”，即该行 ROB 中出现空的 bank。ROB 中的 “气泡” 将指令分摊到了更多的 ROB 行，但是相较只记录连续地址指令的首地址节省的硬件开销，“气泡”的代价是可以接受的。

PC	指令
0x0000	addi x1 x0 0x1
0x0004	slli x1 x1 0x4
0x0008	bne x1 x0 0x10
0x000C	mult x1 x0 x1
0x0010	sra x1 x1 x1
.....	
0x0028	add x1 x0 x1
0x002C	j 0x0010
.....	

ROB

	PC	bank0	bank1	bank2	bank3
rob_tail	0x0010	sra			
	0x0028	add	J		
rob_head	0x0000	addi	Slli	bne	

ROB Bubble

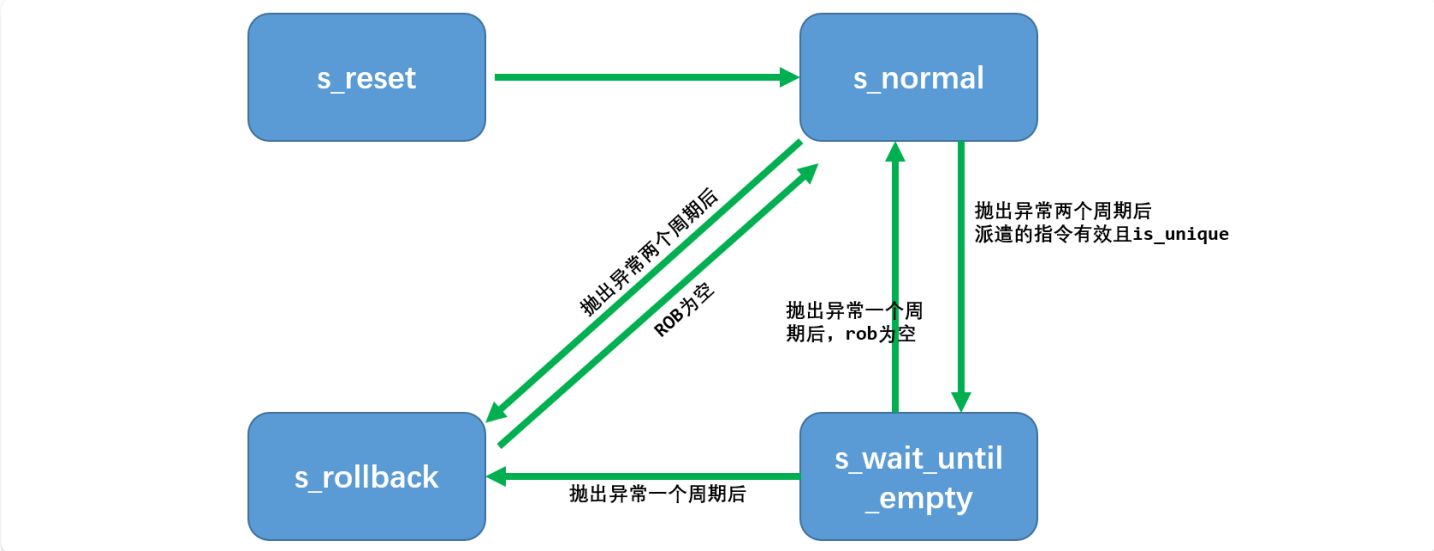
BOOM处理器在 ROB 中以寄存器或线网的形式定义了若干变量，用于记录ROB以及保存在ROB中的指令的各种状态，并响应从写回、派遣等端口输入的信息进行更新，ROB模块的输出由输入和这些状态信息共同决定。本文将结合源代码，详细介绍BOOM对重命名缓存的设计。

ROB 状态机

ROB 模块中设计了一个有限状态机记录 ROB 的四种工作状态

- s_rollback： ROB 正在回滚
- s_normal： ROB 正常状态
- s_reset： ROB 重置
- s_wait_till_empty： 正在等待 ROB 清空

四种状态之间的转换关系如下图所示



ROB状态机

is_unique 信号是定义在 MicroOp 中的一个成员，表示只允许该指令一条指令存在于流水线中，流水线要对 is_unique 的指令做出的响应包括：

- 等待 STQ (Store Queue) 中的指令全部提交
- 清空该指令之后的取到的指令
- ROB 标记为 unready，等待清空

RISCV 指令集中 is_unique 有效的指令主要包括：

- CSR(Control and Status Register) 指令
- 原子指令
- 内存屏障指令
- 休眠指令
- 机器模式特权指令

原子指令

原子(atom)源自希腊语 άτομο，意为“不可分割”，原子操作则为不能被中断的操作，要么全部执行完、要么一个步骤也不执行。RISCV32A 指令集中包含了内存原子操作(AMO,Atomic Memory Operation)和加载保留/条件存储(LR,Load Reserved;SC,Store Contional)两类原子指令。

AMO 指令的作用是从 rs1 指定的内存地址中读取数据保存到目的寄存器，并将该值与 rs2 中的值（或立即数）进行运算，运算类型由具体的指令决定（例如 加法操作amoadd、与操作amoand等），再把结果保存到原存储地址。AMO 指令是一个“读—改—写回” 的过程，中间不允许中断。

LR/SC 指令与 AMO 指令实现的功能类似，先通过 LR 指令将内存中某一地址的数据加载到寄存器中，对该值进行一系列操作（不必使用 RV32A 中的指令），最后通过 SC 指令将数据写回到原存储地址，从 LR 指令到 SC 指令，中间不允许中断。如果执行成功， SC 指令将会把 0 写到目的寄存器。SC 能够将数据写回需要满足一定的条件：

- 有 LR 指令访问过该地址
- LR 与 SC 之间没有对该地址的写操作
- LR 与 SC 之间没有发生中断

以下两段代码演示了原子操作，两者在改变内存方面的作用是一致的，即将由 a3 的值作为地址取内存中的数据与 a2 的值相加后，写回存储器原位置。两者不同之处在于 LR/SC 代码段中使用 a4 寄存器暂存计算结果、使用 a0 寄存器保存操作是否成功。

AMO:

```
amoadd.w a1,a2,(a3)
```

LR/SC:

```
lr.w a1,(a3)
add a4,a1,a2
sc.w a0,a4,(a3)
```

由于原子操作不可中断的特性，执行原子指令时处理器需要将流水线中位于其后的指令清空、以及保证其前的所有 store 指令完成，需要标记为 is_unique。

CSR指令

RISCV 中除了32个通用寄存器外，还有一组拓展寄存器，称为 Control and Status Register ,与控制和记录处理器状态有关。相应地，RISCV 在 Zicsr 指令集中定义了对 CSR 进行读写的 CSR 指令，CSR 都是原子操作，对 CSR 寄存器进行“读-改-写回”。因此CSR 指令在译码是同样会被标记为 unique。

内存屏障指令

RISCV 采用的是 RVWMO(RISCV Weak Memory Ordering) 内存模型，对访存顺序约束较为宽松，因此需要内存屏障指令来限制指令的执行顺序。

fence 指令的作用是将前面的存储指令与后面的存储指令隔离，即保证 fence 指令之前的存储指令先于 fence 之后的存储指令完成（可被观测）。

fence.i 指令用于同步指令和数据流，fence.i 之后的取指操作一定能观测到 fence.i 之前的所有指令造成的访存结果。 在执行内存屏障指令时，冲刷流水线与缓存，保证其前的指令全部执行完毕再取指后

续指令。

fence.vma 指令的作用是更新页表或 TLB (Translation Lookaside Buffer)，确保在 fence.vma 指令之前的存储操作都是有序的。

休眠指令

WFI(Wait For Interrupt)是 RISCV 指令集中定义的一条休眠指令，当执行到 WFI 指令时处理器会停止当前流水线，进入休眠状态，直至处理器收到中断信号，处理器被“唤醒”。WFI 是实现低功耗机制的一种方法。

机器模式特权指令

- ECALL：向更高特权级发起请求
- EBREAK：控制转回调试环境、引发断点异常
- ERET：环境返回

ROB 状态机的具体代码附上：

```
  if (!enableCommitMapTable) {    //enableCommitMapTable参数默认是0
    switch (rob_state) {
      is (s_reset) {
        rob_state := s_normal
      }
      is (s_normal) {
        // Delay rollback 2 cycles so branch mispredictions can drain
        when (RegNext(RegNext(exception_thrown))) {
          rob_state := s_rollback
        } .otherwise {
          for (w <- 0 until coreWidth) {
            when (io.enq_valids(w) && io.enq_uops(w).is_unique) {
              rob_state := s_wait_till_empty
            }
          }
        }
      }
      is (s_rollback) {
        when (empty) {
          rob_state := s_normal
        }
      }
      is (s_wait_till_empty) {
        when (RegNext(exception_thrown)) {
          rob_state := s_rollback
        } .elsewhen (empty) {
          rob_state := s_normal
        }
      }
    }
  } else {
    switch (rob_state) {
      is (s_reset) {
        rob_state := s_normal
      }
      is (s_normal) {
        when (exception_thrown) {
          ;//rob_state := s_rollback
        } .otherwise {
          for (w <- 0 until coreWidth) {
            when (io.enq_valids(w) && io.enq_uops(w).is_unique) {
              rob_state := s_wait_till_empty
            }
          }
        }
      }
      is (s_rollback) {
        when (rob_tail_idx  === rob_head_idx) {
          rob_state := s_normal
        }
      }
      is (s_wait_till_empty) {
        when (exception_thrown) {
          ;//rob_state := s_rollback
```

```
      } .elsewhen (rob_tail === rob_head) {
        rob_state := s_normal
      }
    }
  }
}
```

响应输入

ROB的本质是一个循环队列，从头部提交，从尾部输入。ROB 源码中遍历每个 bank 对外部输入进行响应。

```
for (w <- 0 until coreWidth) {……}
```

BOOM 为每个bank中的所有指令定义了若干变量记录重命名缓存的状态信息，主要包括：

- rob_val： 当前 bank 中每行指令的有效信号，初始化为0
- rob_bsy： 当前 bank 中每行指令的 busy 信号，busy=1时表示指令还在流水线中
- rob_unsafe： 当前 bank 中每行指令的 unsafe 信号，指令 safe 表示一定可以被提交
- rob_uop： 当前 bank 中的每行指令

响应派遣级的输入

派遣级输入ROB的信息包括入列(enqueue)有效信号、入列微指令。入列信号有效时，将派遣到 ROB 的微指令放入 ROB 的尾(tail)，并更新各个变量在 tail 索引处的值。

```
when (io.enq_valids(w)) {
    rob_val(rob_tail)       := true.B
    rob_bsy(rob_tail)       := !(io.enq_uops(w).is_fence || io.enq_uops(w).is_fencei)
    rob_unsafe(rob_tail)    := io.enq_uops(w).unsafe
    rob_uop(rob_tail)       := io.enq_uops(w)
    rob_exception(rob_tail) := io.enq_uops(w).exception
    ……
    } .elsewhen (io.enq_valids.reduce(_|_) && !rob_val(rob_tail)) {
    ……
    }
```

其中比较特殊的是 rob_bsy 信号的更新，当且仅当派遣到 ROB 的指令是 fence 指令或者 fence.i 时，rob_bsy(tail) 将被标记为 0(unbusy)。

RISC-V 指令集中的 fence 指令用于约束内存及设备I/O内存的访问顺序，在 fence 指令前的访存指令必须比在 fence 指令后的访存指令先完成，fence 本身不执行任何操作，因此被视为不在流水线中(unbusy)。

fence.i 指令来自 Zifencei 扩展，用于同步指令流和数据内存访问，即保证读取的指令的总是最新写入的指令，fence.i 指令同样在 ROB 中标记为 unbusy。

MicroOp 类中定义了 unsafe 成员，当“推测执行”的指令处于下列情形之一时其微指令 unsafe 成员为 1，表示该指令可能无法提交：

- 使用 load 队列
- 使用 store 队列 且不是 fence指令
- 是分支指令
- 是跳转指令 (jalr 或 jal)

```
def unsafe           = uses_ldq || (uses_stq && !is_fence) || is_br || is_jalr
```

响应写回级的输入

从写回级输入ROB的变量为 wb_resps，记录了写回的微指令以及该微指令在 ROB中的位置。
写回的指令已经不在流水线中，标记为 unbusy，相应的ROB条目标记为safe，并对 rob_predicated进行赋值。

```
for (i <- 0 until numWakeupPorts) {
    val wb_resp = io.wb_resps(i)
    val wb_uop = wb_resp.bits.uop
    val row_idx = GetRowIdx(wb_uop.rob_idx)
    when (wb_resp.valid && MatchBank(GetBankIdx(wb_uop.rob_idx))) {
        rob_bsy(row_idx)      := false.B
        rob_unsafe(row_idx)   := false.B
        rob_predicated(row_idx)  := wb_resp.bits.predicated
    }
}
```

GetRowIdx 函数的作用是根据ROB条目的索引返回该条目所在行数；
GetBankIdx 函数的作用是根据ROB条目索引返回该条目所在的bank；
MatchBank 定义在循环for (w <- 0 until coreWidth) {……}内，检测输入的bank与当前正在处理的bank是否一致。

响应LSU的输入

LSU(Load Store Unit) 负责决定何时向内存系统触发访存操作。从LSU输入ROB的数据主要包括：

- lsu_clr_bsy ： 当要 LSU 模块正确接受了要保存的数据时，清除 store 命令的 busy 状态，同时将指令标记为 safe。clr_bsy 信号的值与存储目标地址是否有效、TLB是否命中、是否处于错误的分支预测下、该指令在存储队列中的状态等因素有关。
- lsu_clr_unsafe ： 推测 load 命令除了 Memory Ordering Failure 之外不会出现其他异常时，将 load 指令标记为 safe。lsu_clr_unsafe 信号要等广播异常之后才能输出，采用 RegNext 类型寄存器来延迟一个时钟周期。

- lxcpt： 来自LSU的异常，包括异常的指令、异常是否有效、异常原因等信息。异常的指令在 rob_exception 中对应的值将置为1。

```
for (clr_rob_idx <- io.lsu_clr_bsy) {
    when (clr_rob_idx.valid && MatchBank(GetBankIdx(clr_rob_idx.bits))) {
        val cidx = GetRowIdx(clr_rob_idx.bits)
        rob_bsy(cidx)    := false.B
        rob_unsafe(cidx) := false.B
        assert (rob_val(cidx) === true.B, "[rob] store writing back to invalid entry.")
        assert (rob_bsy(cidx) === true.B, "[rob] store writing back to a not-busy entry.")
    }
}
for (clr <- io.lsu_clr_unsafe) {
    when (clr.valid && MatchBank(GetBankIdx(clr.bits))) {
        val cidx = GetRowIdx(clr.bits)
        rob_unsafe(cidx) := false.B
    }
}
    when (io.lxcpt.valid && MatchBank(GetBankIdx(io.lxcpt.bits.uop.rob_idx))) {
      rob_exception(GetRowIdx(io.lxcpt.bits.uop.rob_idx)) := true.B
      when (io.lxcpt.bits.cause =/= MINI_EXCEPTION_MEM_ORDERING) {
        // In the case of a mem-ordering failure, the failing load will have been marked safe a
        assert(rob_unsafe(GetRowIdx(io.lxcpt.bits.uop.rob_idx)),"An instruction marked as safe
      }
    }
```

store 命令特殊之处在于不需要写回 (Write Back) 寄存器，因此 LSU 模块将 store 指令从存储队列提交后，store 命令就可以从流水线中退休，即 io.lsu_clr_bsy 信号将 store 指令置为 safe 时同时置为 unbusy。

MINI_EXCEPTION_MEM_ORDERING 是指发生存储-加载顺序异常(Memory Ordering Failure)。当 store 指令与其后的 load 指令有共同的目标地址时，类似 RAW 冲突，若 load 指令在 store 之前发射 (Issue)，load 命令将从内存中读取错误的值。处理器在提交 store 指令时需要检查是否发生了Memory Ordering Failure，如果有，则需要刷新流水线、修改重命名映射表等。Memory Ordering Failure 是处理器乱序执行带来的问题,是处理器设计的缺陷，不属于 RISCV 规定的异常，采用 MINI_EXCEPTION_MEM_ORSERING 来弥补。

```
can_throw_exception(w) := rob_val(rob_head) && rob_exception(rob_head)
```

当位于ROB头(head)的指令有效且异常时，才允许抛出异常。

响应CSR的输入

CSR(Control Status Register) 输入 ROB 的暂停信号 csr_stall 主要影响提交逻辑。

```
can_commit(w) := rob_val(rob_head) && !(rob_bsy(rob_head)) && !io.csr_stall
```

can_commit 在 ROB 头的指令有效且已不在流水线中且未收到来自 CSR 的暂停信号时有效，表示此时在 ROB 头的指令可以提交。

响应分支信息

该部分的主要作用是在分支预测失误时消除处于“推测”状态的指令。rob_uop的br_mask成员记录了指令的分支依赖关系，io.brupdate中记录了分支预测错误。若当前指令依赖的分支预测错误，指令标记为unvalid；否则，根据io.brupdate更新微指令的br_mask成员。

```
for (i <- 0 until numRobRows) {
    val br_mask = rob_uop(i).br_mask
    when (IsKilledByBranch(io.brupdate, br_mask))
    {
        rob_val(i) := false.B
        rob_uop(i.U).debug_inst := BUBBLE
    } .elsewhen (rob_val(i)) {
        rob_uop(i).br_mask := GetNewBrMask(io.brupdate, br_mask)
    }
}
```

IsKilledByBranch 的功能是根据判断当前微指令是否处于一个预测错误的分支下。

```
object IsKilledByBranch
{
  def apply(brupdate: BrUpdateInfo, uop: MicroOp): Bool = {
    return maskMatch(brupdate.b1.mispredict_mask, uop.br_mask)
  }

  def apply(brupdate: BrUpdateInfo, uop_mask: UInt): Bool = {
    return maskMatch(brupdate.b1.mispredict_mask, uop_mask)
  }
}
```

maskMatch 的功能是检测输入的两个整数是否同为1.

```
object maskMatch
{
   def apply(msk1: UInt, msk2: UInt): Bool =
   {
      val br_match = (msk1 & msk2) =/= 0.U
      return br_match
   }
}
```

Commit Logic

ROB 提交的基本规则是：只有位于 ROB 头部的指令可以提交，其余指令的提交状态将被封锁，也只有在 ROB 头的指令发生的异常才可以抛出。

```
can_throw_exception(w) := rob_val(rob_head) && rob_exception(rob_head)
……
can_commit(w) := rob_val(rob_head) && !(rob_bsy(rob_head)) && !io.csr_stall
……
var block_commit = (rob_state =/= s_normal) && (rob_state =/= s_wait_till_empty) || RegNext(exc
var will_throw_exception = false.B
var block_xcpt   = false.B
for (w <- 0 until coreWidth) {  //遍历 每个bank
  will_throw_exception = (can_throw_exception(w) && !block_commit && !block_xcpt) || will_throv
  will_commit(w)       := can_commit(w) && !can_throw_exception(w) && !block_commit

  block_commit         = (rob_head_vals(w) &&
                           (!can_commit(w) || can_throw_exception(w))) || block_commit
  block_xcpt           = will_commit(w)
}
exception_thrown := will_throw_exception
```

will_commit

这一段代码的主要作用是为 head 指针指向的 ROB 行中的每一个 bank 生成 will_commit 信号，will_commit 信号指示下一时钟周期指令是否提交。will_commit 信号有效的条件是：

- 该 bank 中的指令可以提交
- 该 bank 中的指令不会抛出异常
- ROB 的提交没有被封锁

block_commit

block_commit=1 时，ROB 既不能提交指令，也不能抛出异常。对于每个bank，都有一个自己的 block_commit 信号，只要一个 bank 被封锁提交，其后的所有 bank 都将被封锁提交。block_commit 信号保证 ROB 只能顺序提交。若 ROB 处于 s_rollback 或 s_reset 状态，或在前两个时钟周期内抛出异常时，block_commit将被初始化为1,即该行所有指令的提交都被封锁。

will_throw_exception： 表示下一时钟周期将要抛出异常，该信号初始化为0，使信号有效的条件包括：

- 当前bank可以抛出异常
- 没有封锁提交
- 上一个bank没有要提交的指令

Exception Logic

ROB接受的异常信息来自两个方面：

- 前端发生的异常，输入端口为io.enq_valid和io.enq_uops.exception
- LSU发生的异常，输入端口为io.lxcpt

由于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，节省硬件资源。
寄存器 **r_xcpt_uop** 中保存着最古老的异常信息，其更新逻辑为：

- 发生回滚、前端重定向、抛出异常时，不再更新 r_xcpt，因为此时已经发生了异常，ROB 中所有指令都将清空，没有再记录的必要。
- LSU 异常的优先级高于前端异常，若 LSU 异常比当前记录的异常指令更古老，将 r_xcpt_uop 更新为 LSU 输入的异常。
- 由于从派遣端输入的指令总是 ROB 中最新的指令，只有当前未记录异常且LSU未输入异常时，才会将 r_xcpt_uop 更新为前端端输入的第一个异常，也即最老的异常。
- 若异常的指令依赖一个预测失误的分支，该异常无效(r_xcpt_val 置0)

实现该功能的代码如下：

```
val r_xcpt_val       = RegInit(false.B)
val r_xcpt_uop       = Reg(new MicroOp())
……
val next_xcpt_uop = Wire(new MicroOp())
next_xcpt_uop := r_xcpt_uop
val enq_xcpts = Wire(Vec(coreWidth, Bool()))
for (i <- 0 until coreWidth) {
    enq_xcpts(i) := io.enq_valids(i) && io.enq_uops(i).exception
}
……
when (!(io.flush.valid || exception_thrown) && rob_state =/= s_rollback) {
    when (io.lxcpt.valid) {
        val new_xcpt_uop = io.lxcpt.bits.uop

        when (!r_xcpt_val || IsOlder(new_xcpt_uop.rob_idx, r_xcpt_uop.rob_idx, rob_head_idx)) {
            r_xcpt_val              := true.B
            next_xcpt_uop           := new_xcpt_uop
            next_xcpt_uop.exc_cause := io.lxcpt.bits.cause
            r_xcpt_badvaddr         := io.lxcpt.bits.badvaddr
      }
    } .elsewhen (!r_xcpt_val && enq_xcpts.reduce(_|_)) {
        r_xcpt_val      := true.B
        next_xcpt_uop   := io.enq_uops(idx)
        r_xcpt_badvaddr := AlignPCToBoundary(io.xcpt_fetch_pc, icBlockBytes) | io.enq_uops(idx)
    }
  }
r_xcpt_uop         := next_xcpt_uop
r_xcpt_uop.br_mask := GetNewBrMask(io.brupdate, next_xcpt_uop)
when (io.flush.valid || IsKilledByBranch(io.brupdate, next_xcpt_uop)) { //异常指令依赖的分支预测
    r_xcpt_val := false.B
}
```

IsOlder 的作用是判断三个输入是否单调。

```
object IsOlder
{
  def apply(i0: UInt, i1: UInt, head: UInt) = ((i0 < i1) ^ (i0 < head) ^ (i1 < head))
}
```

ROB Head Logic

ROB 中只有头指针指向的指令才能提交，头指针包括指向行的 rob_head 和指向 bank 的 rob_head_lsb 两部分。头指针的更新逻辑为：

- 头指针指向ROB行中的指令全部提交完毕时，rob_head 指向下一行，rob_head_lsb指向第0个 bank
- 否则，rob_head 不动，rob_head_lsb 指向第一个 rob_head_val=1 的 bank

```
val finished_committing_row =
    (io.commit.valids.asUInt =/= 0.U) &&
    ((will_commit.asUInt ^ rob_head_vals.asUInt) === 0.U) &&
    !(r_partial_row && rob_head === rob_tail && !maybe_full)
……
when (finished_committing_row) {
    rob_head     := WrapInc(rob_head, numRobRows)
    rob_head_lsb := 0.U
    rob_deq      := true.B
} .otherwise {
    rob_head_lsb := OHToUInt(PriorityEncoderOH(rob_head_vals.asUInt))
}
```

rob_head_val 是 rob_val 在 head 行的值。rob_val 的更新取决于各个端口的输入，前文已经叙述详尽，在此做一总结：

- rob_val 初始化为0
- 指令从派遣端口入列，rob_val(tail) 置1
- 指令回滚，rob_val 置0
- 指令依赖的分支预测错误，置0
- 下一时钟周期即将提交(will_commit=1)的指令置0

WrapInc 实现了 rob_head 的自加，该对象的具体代码如下：

```
object WrapInc
{
  def apply(value: UInt, n: Int): UInt = {
    if (isPow2(n)) {
      (value + 1.U)(log2Ceil(n)-1,0)
    } else {
      val wrap = (value === (n-1).U)
      Mux(wrap, 0.U, value + 1.U)
    }
  }
}
```

ROB Tail Logic

ROB 的 tail 指针将按照优先级由高到低的顺序，在以下情形进行更新：

- 行回滚：rob_tail 指向上一行，rob_tail_lsb 指向最末一个bank，出列信号(rob_deq)有效
- 入口回滚：rob_tail 不动，rob_tail_lsb 指向 rob_head_lsb
- 分支预测失误：rob_tail 指向分支预测失误的指令的下一行
- 行派遣：rob_tail 指向下一行，rob_tail_lsb 指向第0个bank，入列信号(rob_enq)有效
- 部分派遣：rob_tail_lsb 指向最后一个有效指令的下一bank

```
when (rob_state === s_rollback && (rob_tail =/= rob_head || maybe_full)) {
    rob_tail     := WrapDec(rob_tail, numRobRows)
    rob_tail_lsb := (coreWidth-1).U
    rob_deq := true.B
} .elsewhen (rob_state === s_rollback && (rob_tail === rob_head) && !maybe_full) {
    rob_tail_lsb := rob_head_lsb
} .elsewhen (io.brupdate.b2.mispredict) {
    rob_tail     := WrapInc(GetRowIdx(io.brupdate.b2.uop.rob_idx), numRobRows)
    rob_tail_lsb := 0.U
} .elsewhen (io.enq_valids.asUInt =/= 0.U && !io.enq_partial_stall) {
    rob_tail     := WrapInc(rob_tail, numRobRows)
    rob_tail_lsb := 0.U
    rob_enq      := true.B    //入列
} .elsewhen (io.enq_valids.asUInt =/= 0.U && io.enq_partial_stall) {
    rob_tail_lsb := PriorityEncoder(~MaskLower(io.enq_valids.asUInt))
}
```

WrapDec 与 WrapInc 类似，实现的是循环自减操作

```
object WrapDec
{
  // "n" is the number of increments, so we wrap at n-1.
  def apply(value: UInt, n: Int): UInt = {
    if (isPow2(n)) {
      (value - 1.U)(log2Ceil(n)-1,0)
    } else {
      val wrap = (value === 0.U)
      Mux(wrap, (n-1).U, value - 1.U)
    }
  }
}
```

ROB PNR Logic

PNR(Point of No Return)指针指向第一个(最老的)处于流水线中、可能引起异常或位于错误的分支预测下的指令，换言之，pnr指针之前的指令一定是安全的。
enableFastPNR 参数用于启用一个快速的 PNR 机制，若存在 unsafe 的指令则 rob_pnr_idx 指向头指针之后第一个 unsafe 的指令，否则指向 tail 行中第一个unvalid的指令。enableFastPNR参数默认为FALSE，此时PNR逻辑将更为复杂。

pnr 指针的更新与 safe_to_inc 和 do_inc_row 两个变量密切相关，safe_to_inc 是表示允许后移 pnr 指针的信号，rob_pnr_row 是表示允许 pnr 指针移动到下一行的信号。pnr 指针的更新还与 ROB 当前的状态和 pnr 指针当前的位置有关。pnr 指针更新逻辑比较复杂，如下图所示

ROB当前状态 \ PNR当前位置	head	else	tail
empty	1	1	1
else	2(do_inc_row),3	2(do_inc_row),3	4
full	2(do_inc_row),3	2(do_inc_row),3	5

PNR的更新机制(1)

更新模式编号 \ 更新对象	rob_pnr	rob_pnr_lsb
1	指向rob_head	指向派遣端的第一个有效指令(从bank0算起)
2	指向下一行	指向bank0
3	保持	指向第一个unsafe的指令
4	保持	指向第一个unsafe的指令 或最后一条有效指令的下一位置
5	保持	指向bank0

PNR的更新机制(2)

具体代码：

```
if (enableFastPNR) {
    val unsafe_entry_in_rob = rob_unsafe_masked.reduce(_||_)
    val next_rob_pnr_idx = Mux(unsafe_entry_in_rob,
                               AgePriorityEncoder(rob_unsafe_masked, rob_head_idx),
                               rob_tail << log2Ceil(coreWidth) | PriorityEncoder(~rob_tail_vals
    rob_pnr := next_rob_pnr_idx >> log2Ceil(coreWidth)
    if (coreWidth > 1)
      rob_pnr_lsb := next_rob_pnr_idx(log2Ceil(coreWidth)-1, 0)
} else {
    val pnr_maybe_at_tail = RegInit(false.B)
    val safe_to_inc = rob_state === s_normal || rob_state === s_wait_till_empty
    val do_inc_row  = !rob_pnr_unsafe.reduce(_||_) && (rob_pnr =/= rob_tail || (full && !pnr_ma
    when (empty && io.enq_valids.asUInt =/= 0.U) {
      rob_pnr     := rob_head
      rob_pnr_lsb := PriorityEncoder(io.enq_valids)//1
    } .elsewhen (safe_to_inc && do_inc_row) {  //2
      rob_pnr     := WrapInc(rob_pnr, numRobRows)
      rob_pnr_lsb := 0.U
    } .elsewhen (safe_to_inc && (rob_pnr =/= rob_tail || (full && !pnr_maybe_at_tail))) {  //3
      rob_pnr_lsb := PriorityEncoder(rob_pnr_unsafe)
    } .elsewhen (safe_to_inc && !full && !empty) {   //4
      rob_pnr_lsb := PriorityEncoder(rob_pnr_unsafe.asUInt | ~MaskLower(rob_tail_vals.asUInt))
    } .elsewhen (full && pnr_maybe_at_tail) { //5
      rob_pnr_lsb := 0.U
    }
    pnr_maybe_at_tail := !rob_deq && (do_inc_row || pnr_maybe_at_tail)
}
```

pnr_maybe_at_tail 与 maybe_full 类似，是 pnr 指针指向 tail 的必要不充分条件，用于区别 ROB 已满时 pnr 指针与 tail 重合的情形，其更新逻辑是：

- 有指令出列，置0
- pnr 指针移向下一行，置1

MaskLower 的作用将最高位的1之后的所有低位置为1，例如 MaskLower(0010001000)=0011111111

```
object MaskLower
{
  def apply(in: UInt) = {
    val n = in.getWidth
    (0 until n).map(i => in >> i.U).reduce(_|_)
  }
}
```

PriorityEncoder 是chisel中的优先编码器，低位优先。chisel 还提供了 另一种形式的优先编码器 PriorityEncoderOH，返回的是第一个有效位的独热码，例如

```
PriorityEncoderOH(Seq(true.B,true.B,true.B,false.B))=Seq(false.B,false.B,true.B,false.B)
```

AgePriorityEncoder(in,head) 是BOOM中定义的一个优先编码器，其作用是返回序列in在head位置之后的第一个有效值(1)的位置。

rob_pnr_unsafe 记录的是当前 pnr 指针指向的行的指令是否安全，与 rob_vals , rob_unsafe 和 rob_exception有关，这里对 rob_unsafe 和 rob_exception 的更新情况也做一小结。

```
rob_pnr_unsafe(w) := rob_val(rob_pnr) && (rob_unsafe(rob_pnr) || rob_exception(rob_pnr))
```

rob_unsafe 的更新逻辑为：

- 初始化为0
- 指令入列(派遣有效)时，unsafe(tail)与新派遣的指令的unsafe状态保持一致
- 被写回的指令置为safe
- store指令被置为safe

rob_exception 的更新逻辑为：

- rob_exception(tail)与新派遣的指令的exception成员保持一致
- LSU模块输入的异常指令在rob_exception中置1
- 回滚的指令置0

```
object AgePriorityEncoder
{
  def apply(in: Seq[Bool], head: UInt): UInt = {
    val n = in.size
    val width = log2Ceil(in.size)
    val n_padded = 1 << width
    val temp_vec = (0 until n_padded).map(i => if (i < n) in(i) && i.U >= head else false.B) ++
    val idx = PriorityEncoder(temp_vec)
    idx(width-1, 0)
  }
}
```

Empty & Full

判断 ROB 为空的条件：

- 头尾指针重合
- 头指针指向的行都unvalid

判断 ROB 已满的条件：

- 头尾指针重合
- maybe_full=1

由于在多种情况下头尾指针都有可能重合(例如初始化时、ROB 已满、ROB 为空等)，因此 maybe_full 作为 ROB 为满的必要不充分条件是必须存在的。maybe_full 的更新逻辑：

- 有入列无出列时置1
- 分支预测失误时置1

```
maybe_full := !rob_deq && (rob_enq || maybe_full) || io.brupdate.b1.mispredict_mask =/= 0.U
full       := rob_tail === rob_head && maybe_full
empty      := (rob_head === rob_tail) && (rob_head_vals.asUInt === 0.U)
```

输出

ROB 模块的输出主要包括：

- 指针：包括rob_head_idx,rob_tail_idx,rob_pnr_idx，由指向行的指针和指向bank的指针组成，以io.rob_hesd_idx为例：

```
val rob_head_idx = if (coreWidth == 1) rob_head else Cat(rob_head, rob_head_lsb)
io.rob_head_idx := rob_head_idx
```

- 提交的信息io.commit，包括
  - io.commit.rbk_valid: 回滚是否有效
  - io.commit.rollback: ROB 状态机当前是否处于 rollback 状态
  - io.commit.valid: 下一时钟周期是否提交指令
  - io.commit.uops: 下一时钟周期要提交的微指令

```
val rbk_row = rob_state === s_rollback && !full
io.commit.rbk_valids(w) := rbk_row && rob_val(com_idx) && !(enableCommitMapTable.B)
io.commit.rollback := (rob_state === s_rollback)
……
io.commit.valids(w) := will_commit(w)
io.commit.arch_valids(w) := will_commit(w) && !rob_predicated(com_idx)
val com_idx = Mux(rob_state === s_rollback, rob_tail, rob_head)
io.commit.uops(w)   := rob_uop(com_idx)
```

- to LSU
  io.com_load_is_at_rob_head: 通知 LSU 有 load 指令在 ROB 的头上，因为部分 load 指令只有在 ROB 的头上时才能执行。

- to CSR
  io.com_xcpt_uop 为第一个有效的提交的指令，用于向 CSR 提交异常。

```
val com_xcpt_uop = PriorityMux(rob_head_vals, io.commit.uops)
io.com_xcpt.bits.ftq_idx   := com_xcpt_uop.ftq_idx
io.com_xcpt.bits.edge_inst := com_xcpt_uop.edge_inst
io.com_xcpt.bits.is_rvc    := com_xcpt_uop.is_rvc
io.com_xcpt.bits.pc_lob    := com_xcpt_uop.pc_lob
```

- to ren2/Dispatch 

```
val empty = Output(Bool())
val ready = Output(Bool())
```

当 ROB unready 时，rename2/dispatch 阶段接收到的 dis_ready 也将为 0，r_uop 不会更新为 ren1 阶段的微指令。

```
val dec_hazards = (0 until coreWidth).map(w =>
                    dec_valids(w) &&
                    (  !dis_ready
                    || rob.io.commit.rollback
                    || dec_xcpt_stall
                    || branch_mask_full(w)
                    || brupdate.b1.mispredict_mask =/= 0.U
                    || brupdate.b2.mispredict
                    || io.ifu.redirect_flush))

val dis_stalls = dis_hazards.scanLeft(false.B) ((s,h) => s || h).takeRight(coreWidth)
dis_fire := dis_valids zip dis_stalls map {case (v,s) => v && !s}
dis_ready := !dis_stalls.last

rename.io.dis_fire := dis_fire
rename.io.dis_ready := dis_ready
```

- to Frontend
  - io.flush_frontend 信号通知前端是否需要PC重定向，等于寄存器 r_xcpt_val 的值
  - io.flush 信号向前端提供具体的异常信息，需要进行 PC 重定向的原因包括：指令要求在提交时PC重定向(io.commit.uops.flush_on_commit)或者发生了异常(exception_thrown)

```
val flush_commit_mask = Range(0,coreWidth).map{i => io.commit.valids(i) && io.commit.uops(
val flush_commit = flush_commit_mask.reduce(_|_)
val flush_val = exception_thrown || flush_commit
val flush_uop = Mux(exception_thrown, com_xcpt_uop, Mux1H(flush_commit_mask, io.commit.uop
……
io.flush.valid          := flush_val
io.flush.bits.ftq_idx   := flush_uop.ftq_idx
io.flush.bits.pc_lob    := flush_uop.pc_lob
io.flush.bits.edge_inst := flush_uop.edge_inst
io.flush.bits.is_rvc    := flush_uop.is_rvc
io.flush.bits.flush_typ := FlushTypes.getType(flush_val,
                                              exception_thrown && !is_mini_exception,
                                              flush_commit && flush_uop.uopc === uopERET,
                                              refetch_inst)
```

小结

本文介绍了 BOOM 处理器中对重排序缓存的设计。
处理器是计算机的核心，如何充分利用处理器资源是提高计算机性能的关键课题。IPC(Instruction per Cycle)是衡量 CPU 并行度的主要指标，提高 IPC 主要有两个途径：流水线和多发射，其中多发射也可以视为增加了流水线的宽度。伴随多发射机制的是乱序执行和重排序问题，重排序包括以下三种情形：编译器优化的重排序、指令级并行重排序、内存重排序。编译器优化重排序是在保证不改变指令间的依赖关系的前提下调整代码语句的顺序；指令间并行重排序，通过重排序缓存维持指令间的依赖关系；内存重排序，利用缓冲区 STQ(Store Queue)写入存储器，原理与 ROB 类似。ROB 的实现，本质上是一个循环队列，指令从队列尾（tail）入列，从队列头（head）提交或提交异常，同时还要执行分支预测失误、原子操作、访存异常等出发的回滚逻辑。

参考文献

[1]邓正宏,康慕宁,罗旻.超标量微处理器研究与应用[J].微电子学与计算机,2004(09):59-63.
[2] RISCV-BOOM’s documentation
[3]李昭,刘有耀,焦继业,潘树朋.超标量处理器乱序提交机制的研究与设计[J].计算机工程,2021,47(04):180-186.
[4] The RISC-V Instruction Set Manual, Volume II: Privileged Architecture