Revision History

Rev	Description	Authers	Date
v0.1	初步定义重命名模块的硬件设计方案	杨昕	2024.10.3

1. Teminology

Abbreviations	Full Spelling	Description
LE	Load Elimination	合并不同核间load请求的技术
SRS	Self-Redefined-Streaming	使得MMA指令能够复用其src物理寄存器的技术

2. Overview

重命名模块负责将逻辑寄存器号翻译为物理寄存器号，以消除WAW与WAR冲突，多核的重命名模块将合并为一个此模块。可以通过参数来配置是否开启SRS、LE等技术，目前的实现采用verilog（也可以考虑使用Chisel）。对于本设计，多个核的物理寄存器号将统一编号，即core0使用0-7号，core1使用8-15号，以此类推。

3. Paramters

Name	Description
LE	当其置为1时，启用LE技术
SRS	当其置为1时，启用SRS技术
CORE_NUM	HOST CPU的数量
PREG_PER_CORE	每个HOST CPU配备的物理matrix寄存器的数量
LREG_PER_CORE	每个HOST CPU配备的逻辑matrix寄存器的数量
USE_STATIC_RENAME	如果启用，将直接bypass掉rename级，并在指令槽处检查WAW与WAR冲突
releaseWidth	每个周期RST能够向FreeList释放多少个preg
LRTPAWidth	在LRT中每个PA使用的位宽，可探索面积效率比

4. Interface

Rename核心部件的输入Interface定义为`Flipped(Decoupled(Vec(CORE_NUM, LRegInfo)))`，输出Interface定义为`Flipped(Decoupled(Vec(CORE_NUM, PRegInfo)))`，外围部件暂时不考虑。

下表为LRegInfo的定义。

Name	Width	Description
src_lregs	Valid(Vec(3, UInt(log2Ceil(LREG_PER_CORE)).W))	单个核来的待重命名的src寄存器信息
dest_lreg	Valid(UInt(log2Ceil(LREG_PER_CORE).W))	单个核来的待重命名的dest寄存器信息

下表为PRegInfo的定义。

Name	Width	Description
src_pregs	Valid(Vec(3, UInt(log2Ceil(PREG_PER_CORE * CORE_NUM)).W))	单个核来的src寄存器重命名后的preg号
dest_preg	Valid(UInt(log2Ceil(PREG_PER_CORE * CORE_NUM)).W)	单个核来的dest寄存器重命名后的preg号

5. Micro-architecture

每个核配备一个FreeList，存储那些能够被分配的物理寄存器号，用于对dest reg进行寄存器分配。同时配备一个Register Alias Table (RAT)，保存当前lreg-preg的映射关系，用于对src reg进行重命名。另外，全局配备一个Register Status Table (RST)，用于管理所有核的物理寄存器的状态。当启用LE时，将会配备一个全局的Load Register Table (LRT)，用于存储每个物理寄存器的数据是否是从load中来，以及该load指令是否已经完成。

5.1. baseline重命名

5.1.1. FreeList

FreeList是一个多输入多输出的队列，此队列的构造参数如下。

Name	Description
inputWidth	单周期内能从队列中pop出多少个表项
outputWidth	单周期能向队列中push进多少个表项
entryType	队列中存放数据的类型
depth	队列的深度
bypassI2O	当同时输出输入时，是否使用advance后的指针来判断是否能push in
initData	队列中的初始化数据
initHead	队列head的初始化位置
initTail	队列tail的初始化位置

IO端口定义如下。

Name	Definition	Description
in	Flipped(Decoupled(Vec(inputWidth, entryType)))	入队数据
out	Decoupled(Vec(outputWidth, entryType))	出队数据

对于FreeList而言，entryType为`UInt(log2Ceil(PREG_PER_CORE * CORE_NUM).W)`。

FreeList的IO定义如下。

Name	Definition	Description
release_info	Flipped(Decoupled(Vec(releaseWidth, UInt())))	释放物理寄存器的信息
allocate_info	Decoupled(Vec(allocateWidth, UInt()))	分配物理寄存器的信息

对于FreeList而言，有额外的参数如下。

Name	Description
coreId	此freeList属于第几个core

多输入多输入队列的微架构实现类似于一个循环队列。总共的空间消耗为depth+1。队列配备`outputWidth`个head指针与一个`inputWidth`tail指针，当tail指针与head指针重合时，队列判空；当tail指针再advance一次会与head重合时，队列判满。当bypassI2O使能时，队列的判满判空将使用下个周期的tail指针与head指针进行计算（即使用tail_nxt与head_nxt）。当输入的valid信号不是连续的1时，仅选择valid为1的数据push入队；当输出的ready信号不是连续的1时，会先数有多少个1，然后从队列的头部拿出这些数据，再分别路由至ready信号为1的输出中。

代码框架:

```
```

```scala
class MultiIOQueueIO[T <: Data](entryType: T, inputWidth: Int, outputWidth: Int)
extends Bundle {
  val in = Flipped(Vec(inputWidth, Decoupled(entryType.cloneType)))
  val out = Vec(outputWidth, Decoupled(entryType.cloneType))
}

class MultiIOQueue[T <: Data](entryType: T, inputWidth: Int, outputWidth: Int,
depth: Int, bypassI2O: Bool, initData: Array[T], initHead: Int, initTail: Int)
extends Module {
  assert(initData.length == depth, "length of initData must equal to depth")

  val io = IO(new MultiIOQueueIO(entryType, inputWidth, outputWidth))

  def addWithMax(a: UInt, b: UInt, max: UInt): UInt = {
    val sum = a + b
    Mux(sum > max, sum - max, sum) // 如果和大于最大值，则返回和减去最大值，否则返回和
  }

  def wrapAround(a: Int, max: Int): Int = {
    if (a > max) a - max else a
  }

  /**---------------------- Derived Paramters ----------------------*/
  val ptrWidth = log2Ceil(depth + 1)

  /**---------------------- Main Component ----------------------*/
  val headPtrs = RegInit(VecInit((0 until outputWidth).map(i => wrapAround(i +
initHead, depth+1).U(ptrWidth.W))))
  val tailPtrs = RegInit(VecInit((0 until inputWidth).map(i => wrapAround(i +
initTail, depth+1).U(ptrWidth.W))))
  val entries = RegInit(VecInit(initData))

  /**---------------------- Enqueue Logic ----------------------*/
  // Generate free entry mask
  val freeMask = VecInit(tailPtrs.zipWithIndex.map{ case(ptr, i) =>
    val ptr_nxt = addWithMax(ptr, 1.U, (depth+1).U)
    ptr_nxt =/= headPtrs(0)
  })

  // Generate compressed input
  val inputCompressCnt = Vec(inputWidth+1, Wire(UInt(log2Ceil(inputWidth).W)))
  inputCompressCnt(0) := 0.U
  val compressedInput = Vec(inputWidth, Wire(Valid(entryType.cloneType)))
  val compressRoute = Vec(inputWidth, Wire(UInt(log2Ceil(inputWidth).W)))
  val inputValidCnt = Wire(UInt(log2Ceil(inputWidth).W))

  for (i <- 0 until inputWidth) {
    when(io.in(i).valid) {
      inputCompressCnt(i+1) := inputCompressCnt(i) + 1.U
    }.otherwise {
      inputCompressCnt(i+1) := inputCompressCnt(i)
    }
  }

  val a = (io.in.map(in => in.valid))

  for (i <- 0 until inputWidth) {
```

```
    val matchMask = VecInit((io.in.map(in => in.valid) zip
inputCompressCnt.map(cnt => cnt === i.U)).
      map{case (a, b) => a & b})
    compressedInput(i).bits := Mux1H(matchMask, io.in.map(in => in.bits))
    compressedInput(i).valid := matchMask.reduce(_|_)
    compressRoute(i) := PriorityEncoder(matchMask)
  }

  inputValidCnt := inputCompressCnt(inputWidth)

  // Generate input ready
  val hand_shake_success_mask = VecInit(compressedInput.map(elem =>
elem.valid)).asUInt & freeMask.asUInt
  for (i <- 0 until inputWidth) {
    val selectThisMask =  Vec(inputWidth, Wire(Bool()))
    selectThisMask := hand_shake_success_mask & VecInit(compressRoute.map(elem
=> elem === i.U)).asUInt
    io.in(i).ready := selectThisMask.reduce(_|_)
  }

  /**--------------------- Dequeue Logic ---------------------*/
  //...
}
```

5.1.2. RAT

RAT是一个`LREG_PER_CORE`个entry的全连接的表，此表可以做成一个抽象类，通过给不同的entry类型来实现不同的表，在实现上暂时不允许写指针冲突。此外，要求表的读取是当周期给出数据。当在单周期内出现对同一个表项的读写时，不会将写口的数据bypass到读口上。

表的构造参数如下。

Name	Description
numEntry	该表中有多少个entry
numRdPort	读口数量
numWrPort	写口数量
entryType	entry类型
resetData	复位时的数据

IO端口定义如下。

Name	Definition	Description
wr	Flipped(Decoupled(Vec(numRdPort, entryType)))	写口
rd	Decoupled(Vec(numWrPort, entryType))	读口

RATEntry中包含的内容如下。

Name	Description	reset value
preg_idx	该index的lreg被映射到哪个preg	0
valid	该表项是否有效，即该lreg是否被某条指令真正作为dest而重命名过	0

RAT IO定义如下。

Name	Definition	Description
lreg_lookup_req	Vec(renameWidth, Input(new LRegInfo()))	给入的逻辑寄存器信息
preg_lookup_resp	Vec(renameWidth, Output(new PRegInfo()))	给出的物理寄存器信息
freelist_updt	Flipped(Vec(allocateWidth, Valid(new LogPhyRegPair())))	来自FreeList的allocate信息

5.1.3. RST

RST拥有一个`PREG_PER_CORE * CORE_NUM`个entry的表。RSTEntry内容如下。

Name	Description	reset value
been_redefined	该物理寄存器对应的逻辑寄存器是否已经被重新define	0
lastConsumerId	最后一个consumer对应的ID（每个core一个）	0
lastConsumerComitted	最后一个consumer是否已经commit（每个core一个）	0
allocated	是否被分配	0
haveConsumer	该物理寄存器是否有consumer（每个core一个）	0

RST的IO端口定义如下。

Name	Definition	Description
lookup_info	Flipped(Vec(numCore, Vec(renameWidth, Decoupled(new RenameInfo))))	重命名src时覆盖lastConsumerId的写口
allocate_dest_updt	Flipped(Vec(CORE_NUM * renameWidth, Valid(UInt)))	来自于top的信息，表示哪个dest preg被分配，用于更新allocated信号
commit_updt	Flipped(Vec(CORE_NUM, Decoupled(CommitToRenameInfo)))	commit的指令ID
release_info	Decoupled(Vec(CORE_NUM * releaseWidth, UInt))	发送给freeList的release信息

RenameToRSTInfo中包含的信号如下。

Name	Description
instId	重命名的指令的ID
src_pregs	使用到的src reg的物理寄存器标号

CommitToRenameInfo中包含的信号如下。

Name	Description
instId	重命名的指令的ID
PRegInfo	该指令使用的物理寄存器标号信息

当`rename_src_updt`有效时，对应的表项会为会将instID更新到`lastConsumerId`中，同时将这些表项的`haveConsumer`拉高。

当`allocate_dest_updt`有效时，对应表项的`allocated`将会被拉高。

当`lookup_info`中的dest reg信息有效时有效时，对应的old preg的表项中`been_redefined`信号会被拉高。

当`commit_updt`信号有效时，检查其PRegInfo中的`dest_preg`部分，如果为valid，则查RST表对应表项。如果仅`lastConsumerId`与`instID`，则拉高`lastConsumerComitted`信号。释放操作的流程如下。

RST每个周期将对每个表项进行检查，如果某个表项中每个核的`lastConsumerCommitted | ~haveConsumer`，`allocated`，`been_redefined`均为真，则从中为每个core选取`releaseWidth`个可释放的preg信息发送到对应的FreeList中。

1. 将对应表项的`allocated`，`been_redefined`，`haveConsumer`，`lastConsumerCommitted`拉低
2. 向该preg对应的freeList发送释放信息

5.1.4. Top

Top模块的IO端口定义如下。

Name	Definition	Description
rename_req	Flipped(Vec(CORE_NUM * renameWidth, Decoupled(RenameInfo)))	重命名的逻辑寄存器信息
rename_resp	Vec(CORE_NUM * renameWidth, Decoupled(RenameInfo))	重命名完成后的物理寄存器信息
commit_info	Flipped(Decoupled(Vec(CORE_NUM, CommitToRenameInfo)))	来自于commit级的信息
write_snoop	Decoupled(UInt(64.W))	总线上监听到的写snoop请求
translation_done	Vec(CORE_NUM, Decoupled(TranslateToRenameInfo))	来自地址翻译单元产生的翻译完成信号

RenameInfo包含的信号如下。

Name	Definition	Description
lreg_info	LRegInfo	重命名的逻辑寄存器信息
preg_info	PRegInfo	重命名后的物理寄存器信息
instId	UInt	每个rename请求对应的指令ID
instType	UInt	该指令的类型
configReg	ConfigRegInfo	该指令对应的config register
host_srcs	Vec(2, UInt(64.W))	从host CPU发来的src值
eliminated_not_ready	Bool()	这条指令是否是一条已经被eliminated的load但是前方的load指令尚未完成

其中TranslateToRenameInfo的定义如下（仅应当在load指令的一行翻译完时被发送）。

Name	Definition	Description
preg	UInt(pregIndexWidth.W)	被写回的preg
PA	UInt(64.W)	此次翻译出来的PA

当存在有效的rename请求时，处理流程如下：

1. 将 `rename_info` 转发到对应的RAT，并得到对应的preg信息
2. 将 `rename_info` 转发到FreeList，尝试进行分配
3. 将获得的preg信息以及 `rename_instId` 转发到RST，并进行对应的更新

对于某一个rename请求，仅当下列条件均满足时，认为其被处理了，此时拉高输出的resp的valid信号与输入的req的ready信号：

- 无需重新分配物理寄存器 or (物理寄存器分配成功 and RST接受了其分配物理寄存器后的更新请求)
- RST接受了其查RAT后的更新请求
- 该rename请求有效

5.2. Self-Redefined-Streaming Register Rename (SRS)

SRS在baseline的基础上需要修改的部分主要在顶层的连接关系上以及RST。

5.2.1. RST

判断是否能够使用SRS以复用preg的判断由RST负责。当且仅当下列条件满足时，能够SRS bypass preg：

- 该指令的src lreg中有一个与dest lreg相同，且两者均需要为valid
- 相同的src lreg对应的preg的haveConsumer为0
- 相同的src lreg对应的preg需要属于自己的核
- 该输入请求为valid

为此，需要为RST新增下列的IO端口。

Name	Definition	Description
srs_grant	Vec(CORE_NUM * renameWidth, Valid(UInt))	当valid时能够bypass preg，其中的bits是被bypass的preg标号

当不启用SRS时，上述IO的valid将被置为false.B。

当srs_grant有效时，对应preg的表项除了allocated外均需要被重置。

5.2.2. Top

当Top接收到某个srs_grant信号为valid时，将相应FreeList的allocate端口的ready置为0，并将输出的resp中dest preg的标号改为RST传来的preg标号。当srs_grant有效时，即使allocate失败也能够输出valid resp。

统一后的Top连接描述如下：

- FreeList某个allocate端口的ready为对应srs_grant的valid的取反
- resp的输出从FreeList的输出和srs_grant的输出中进行选择，选择信号为srs_grant

Top模块的请求被处理信号的条件更改如下：

- 无需重新分配物理寄存器 or (物理寄存器分配成功 and RST接受了其分配物理寄存器后的更新请求) or srs_grant请求valid
- RST接受了其查RAT后的更新请求
- 该rename请求有效

对于RAT的更新，由于SRS是将preg在相同的lreg之间bypass，因此并不会修改lreg和preg的映射关系，因此无需更新RAT。并且因为FreeList的valid仅在ready信号拉高时才会拉高，因此无需修改FreeList向RAT发起的更新请求中valid信号的逻辑。

5.3. Load Elimination (LE)

LE需要增加一个新的部件：LoadRegisterTable (LRT)。这个表主要用于记录每个preg是否是一个合法的来自于load指令的执行结果，具体的合法判断将在下方说明。

5.3.1. LRT

LRT负责存储每个preg是否是一个合法的来自于load指令的执行结果，且会保存相关load指令是否commit的信息。

LRT表项信息如下。

Name	Definition	Description
canMerge	Bool()	表明这个表项是来自于一个合法的load指令
dataReady	Bool()	表明这个对应的load指令已经commit
VA	UInt(64.W)	这个表项对应的load指令的base va
stride	UInt(16.W)	这个表项对应的load指令的stride
configReg	ConfigRegInfo	这个表项对应的load指令的size
PA	Vec(rowNum, UInt(LRTPAWidth.W))	这个表项对应的load指令的每行对应的PA
PA_updt_cnt	UInt(log2CeilWithMin(rowNum).W)	用于更新PA的计数器

LRT的IO端口定义如下。

Name	Definition	Description
rename_info	Vec(CORE_NUM * renameWidth, Decoupled(new RenameInfo))	完成普通重命名+SRS后的信息
commit_info	Vec(CORE_NUM, Decoupled(new CommitToRenameInfo))	从commit级来的commit信息
lrt_grant	Vec(CORE_NUM * renameWidth, Valid(new MatrixBundle {val pregMergeWith = UInt(pregIndexWidth.W) val dataReady = Bool()}))	LRT发出的许可merge信息，其中pregMergeWith表示需要将此lreg重新映射为哪个preg，dataReady表示merge的那条load指令是否已经commit
write_snoop	Decoupled(UInt(64.W))	总线上监听到的写snoop请求
translation_done	Vec(CORE_NUM, Decoupled(TranslateToRenameInfo))	来自地址翻译单元产生的翻译完成信号

表项的更新流程如下，如果出现对同一表项的先后几次描述，则优先级按出现顺序从高到低排列。

1. 经过LRT比对后，若发现无法merge，且此条指令时load指令，则将`rename_info`中的`stride`和`VA`信息更新到LRT对应的表项中，拉高对应表项的`canMerge`，拉低对应表项的`dataReady`，此时的`PA_updt_cnt`理论上来说须为0；否则，拉低对应表项的`canMerge`，拉低对应表项的`dataReady`，其他信息不更新。
2. 当收到一个valid的commit_info时，LRT查询commit_info中preg对应表项中的`canMerge`信息，如果为真，则拉高对应表项的`dataReady`。对于某一个preg，`dataReady`不应该同时被commit_info拉高和被**1**中描述的行为拉低。

3. 当收到一个valid的`write_snoop`信号时，取高`LRTPAWidth`位作为比对信息，每个`canMerge`被拉高的表项需要查询已翻译完成的PA是否存在一项与其匹配，如果存在匹配，则将`canMerge`拉低，并将`PA_updt_cnt`归零。
4. 当收到一个有效的`translation_done`信息时，若对应表项的`canMerge`为真，将`PA_updt_cnt`位置指示的PA更新为`translation_done`信息中PA的高位，并将`PA_updt_cnt`自增，当`PA_updt_cnt`与`configReg`中的行数信息相同时，`PA_updt_cnt`归零。
5. 当收到来自于RST的release信号时，将对应表项的`canMerge`拉低，以防止后续继续被合并，从而导致某个preg被释放多次。

仅当下列条件均满足时，能够认为对应的指令被merge：

- 存在某个表项的`VA`，`stride`，`configReg`信息能够与当前输入的rename请求的信息相同
- 该表项被标记为`canMerge`
- 该表项当前周期没有在被`write_snoop`无效化
- 对应的指令请求有效
- 对应的指令为load指令

5.3.2. Top

Top的连接关系需要有所修改。在Top定义一个RenameInfo信号`final_result`，作为最终的输出。

对于RAT的更新信号：

- RAT的更新信号将从LRT和FreeList中得到，当lrt_grant为真时，选择lrt_grant中的给出的preg进行更新

对于FreeList的allocate ready信号：

- 当srs_grant为真或lrt_grant信号为真时，将拉低对应port的ready信号

对RST的更新：

- RST的更新将使用`final_result`作为更新的依据

对`final_result`的更新：

- 当srs_grant为真时，dest preg将被替换为old preg
- 当lrt_grant为真时，dest preg将被替换为lrt_grant中给出的preg

Top模块的请求被处理信号的条件更改如下：

- 无需重新分配物理寄存器 or (物理寄存器分配成功 and RST接受了其分配物理寄存器后的更新请求) or srs_grant请求valid or lrt_grant请求 valid
- RST接受了其查RAT后的更新请求
- 该rename请求有效