BOOM微架构学习——寄存器重命名

在介绍BOOM处理器对rename流水级的设计之前，我们先来回答几个基本问题：

- 什么是重命名?
- 为什么需要重命名?
- 如何实现重命名?

什么是重命名？

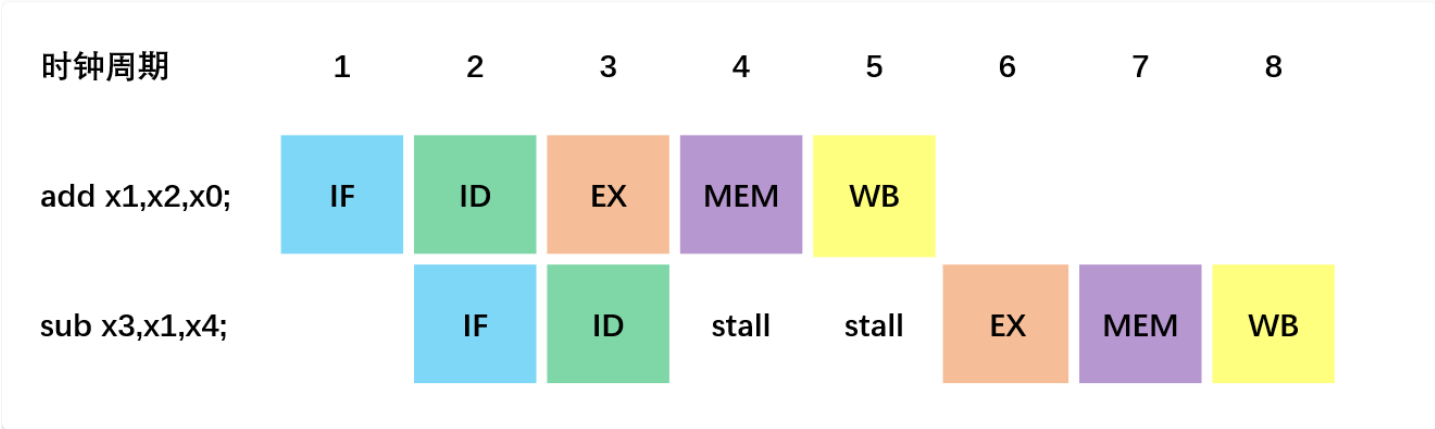
就是重命名就是将指令集(ISA)中规定的寄存器合理地映射为物理寄存器。

为什么需要重命名？

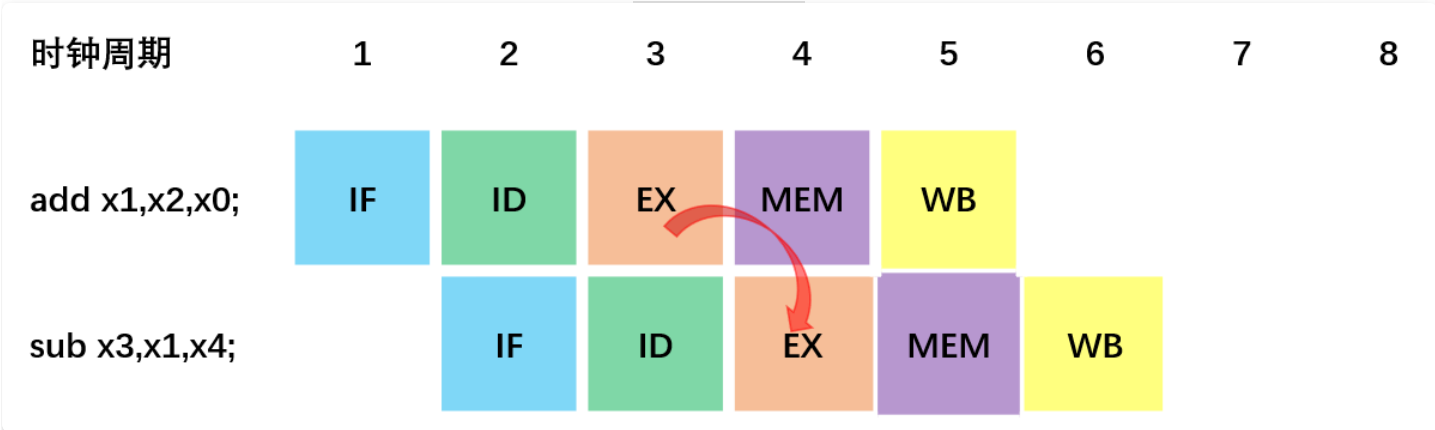
引入重命名技术是为了消除流水线中的名称依赖类型的数据冲突，避免指令不必要的顺序化执行，提高指令并行度。

数据冲突 (data hazard) 是指不同指令的操作数之间存在依赖关系，使用同一寄存器或存储器位置，流水线必须等待相应的依赖关系得到解决后才能继续执行。数据冲突包括写后读(RAW)、写后写(WAW)和写后读(WAR)。

RAW 冲突是指对某一寄存器的写操作尚未完成时后续指令对同一寄存器发起读请求，RAW 冲突是真实依赖，部分 RAW 冲突通过数据前馈解决。

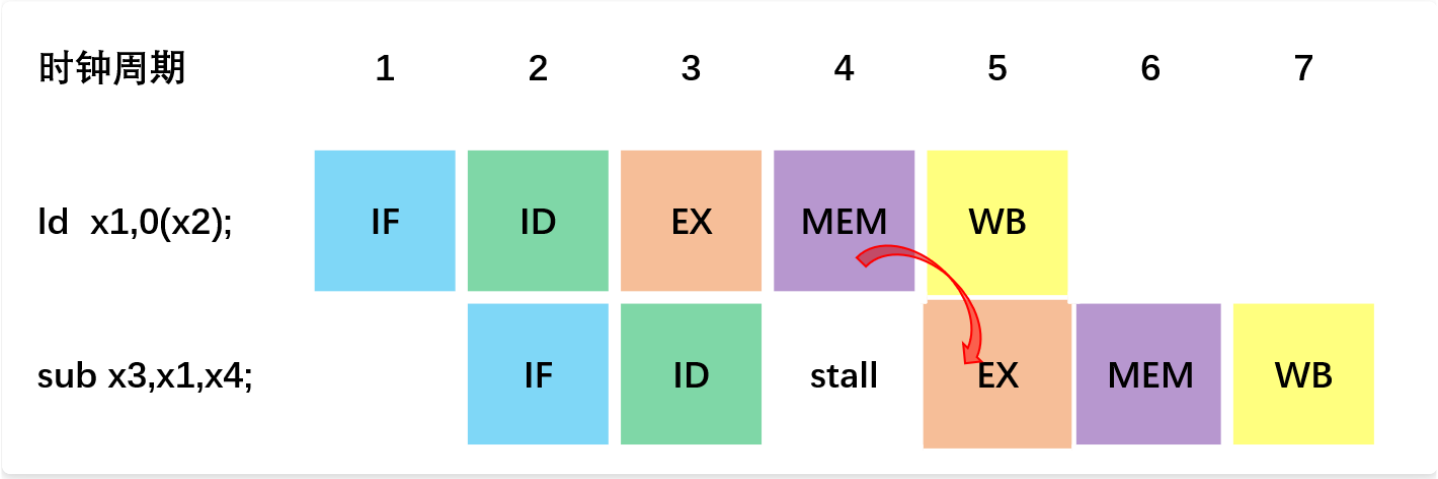


RAW数据冲突



数据前馈解决RAW冲突

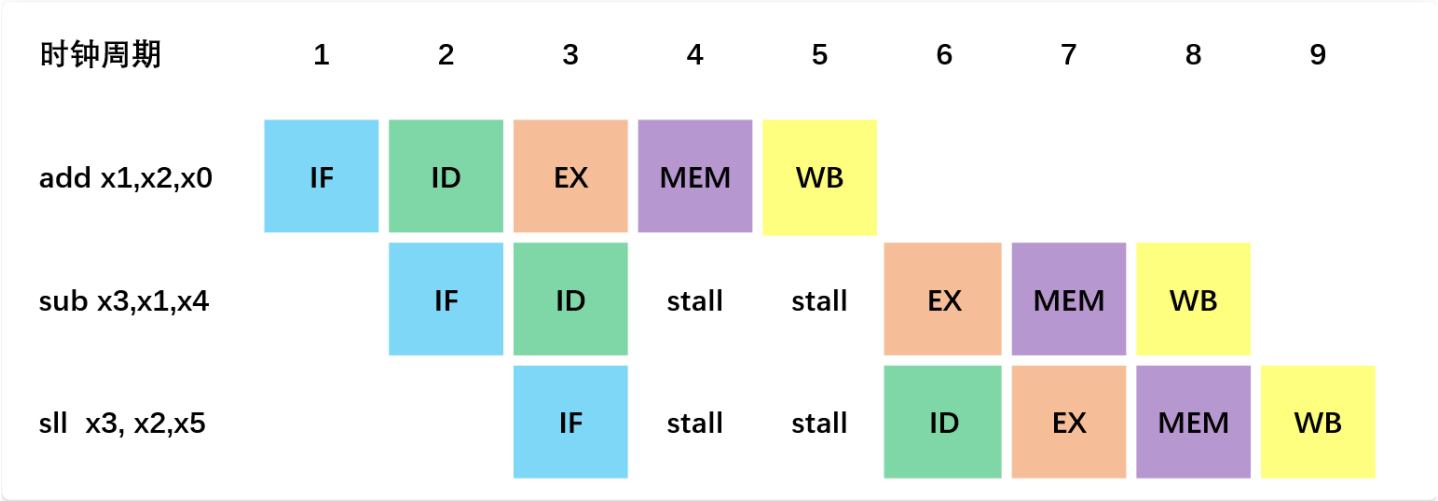
指令在 EX 阶段计算出结果，但在 WB 阶段才能将结果写到寄存器。sub 指令必须等待 add 指令执行完才能从寄存器 x1 中读取正确的值。若将 add 指令在 EX 阶段计算得到的结果前馈到 sub 的 EX 流水级，可以避免等待寄存器写回造成的流水线阻塞，减少时钟周期损失。但并不是所有的RAW冲突都能通过数据前馈解决，例如：



数据前馈不能解决的RAW冲突

load 指令在 MEM 阶段从存储器中读取相应的值，如果紧随其后的指令与其发生 RAW 冲突，流水线将不可避免地暂停一个时钟周期，因为此时前馈的数据尚未产生。

WAW 冲突是指后续指令先于历史指令对同一寄存器发起写操作
WAR 冲突是指后续指令在历史指令读寄存器之前对同一寄存器发起写操作。
以上两种数据冲突仅出现在乱序执行处理器中，因为在顺序执行的处理器中，流水线的暂停是向后传递的，后续指令的读和写必然严格迟于前面的指令。



顺序执行处理器中没有WAW冲突和WAR冲突

WAW 冲突和WAR 冲突是名称依赖，本质是对同名寄存器的竞争访问，可以通过寄存器重命名技术解决。例如以下五条指令之间存在数据冲突，改变任何一条指令的执行顺序或者指令并行都将导致错误。经过重命名之后的五条指令可以以任意顺序执行(包括并行)，只要记录寄存器之间的映射关系，与顺序执行得到的结果完全一致。

Number	Instructions	Instructions after rename
1	add x1, x2, x3	add p1, p2, p3
2	add x8, x1, x4	add p8, p1, p4
3	add x1, x4, x5	add p9, p4, p5
4	add x4, x5, x6	add p10, p5, p6
5	add x4, x6, x7	add p11, p6, p7

Register	Register after rename
x1	p9
x8	p8
x4	p11

重命名技术使乱序执行成为可能

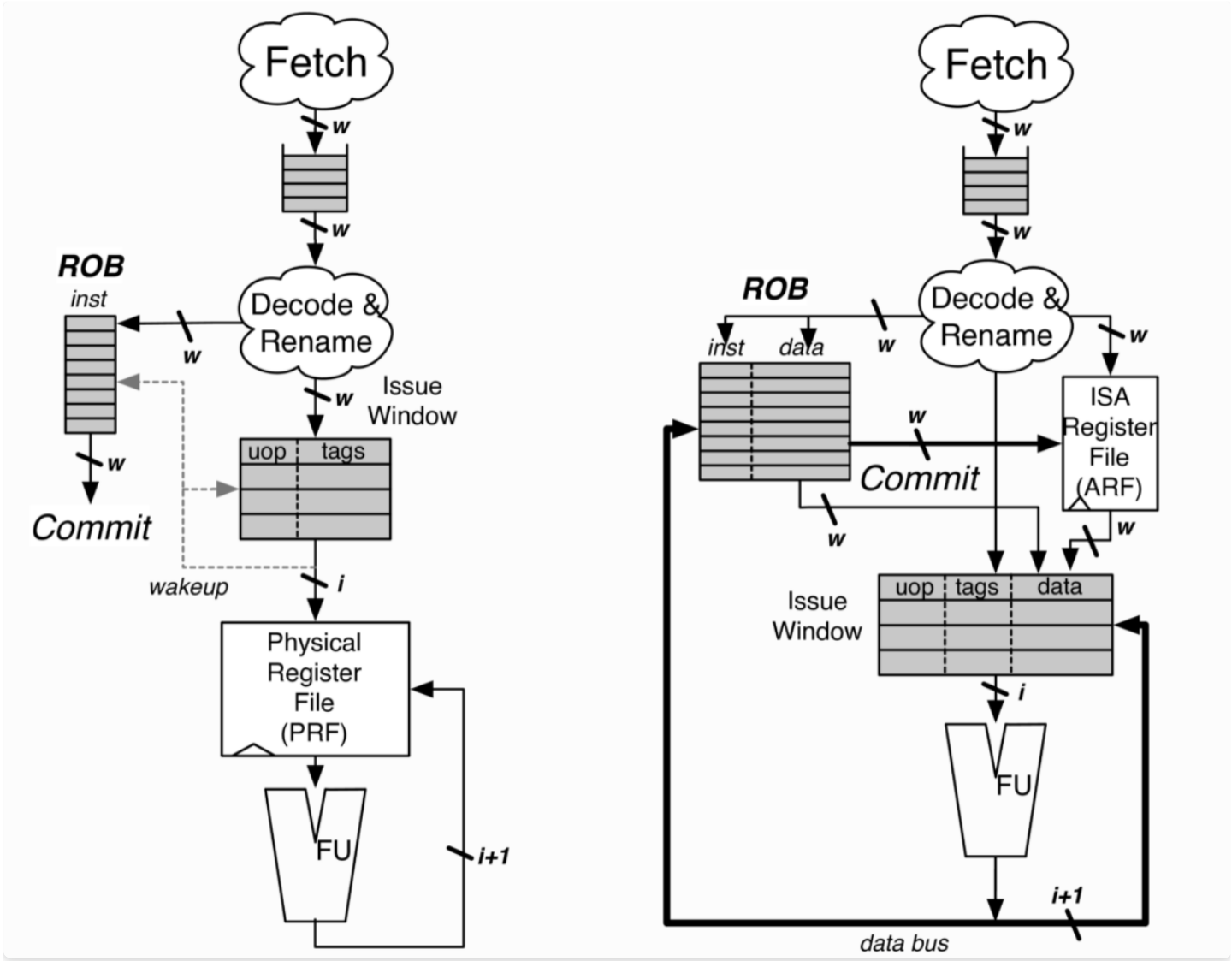
如何实现重命名？

实现重命名的方法在不同的处理器中有不同的具体设计，总体上可以分为两类：隐式重命名和显式重命名。

隐式重命名与显式重命名

显式重命名

BOOM 处理器采用的是显式重命名方案。显式重命名方案的基本结构是：maptable 记录逻辑寄存器与物理寄存器之间的映射关系；freelist记录物理寄存器的空闲状态；busytable 记录寄存器是否可读。显示重命名方案中 ROB 不记录指令的结果，即将提交的数据和处于推测状态的数据都保存在物理寄存器中，因此物理寄存器数目要 高于逻辑寄存器数目。当一条指令发起重命名请求时，通过索引maptable获取其源操作数逻辑寄存器对应的物理寄存器，由freelist分配一个空闲的物理寄存器作为指令的目的寄存器，最后通过busytable 判断指令的源操作数寄存器是否可读，如果可读指令将被发射(issue)。



显式重命名(左)与隐式重命名(右)

隐式重命名

隐式重命名方案见于 Pentium 3 ， Pentium Pro 等处理器。采用隐式重命名方案时，ROB (Recorder Buffer) 保存正在执行、尚未提交的指令的结果；ARF(ISA Register File) 保存已经提交的指令中即将写入寄存器中的值。隐式重命名方案中 ARF 只保存已经提交的指令的值，处于 “推测” 状态的指令的值由 ROB 保存，因此需要的物理寄存器数量与逻辑寄存器数量相同。隐式重命名方案还需要建立一个映射表，记录操作数在 ROB 中的位置。由于流水线中后续指令与已经提交的指令可能有相同的目的寄存器(意味着该寄存器将被修改)，映射表需要增加一个表项，记录对应寄存器的最新值保存在 ROB 还是 ARF 中，这一设计为实现数据前馈、消除 RAW 冲突创造了条件。隐式重命名方案不需要 freelist 来记录物理寄存器状态，指令被写进 ROB 即完成重命名。相比于显式重命名，隐式重命名需要的物理寄存器数目更少，但每个操作数在其生命周期中需要保存在 ROB 和 ARF 两个位置，读取数据的复杂度较高、功耗更高。

映射表的两种结构：RAM与CAM

对于显式的重命名方法，必须设计一个寄存器阵列用于保存逻辑寄存器和物理寄存器之间的映射关系，称为映射表。映射表的结构又可以分为两种：

- RAM：Random Addressed Memory，保存每个逻辑寄存器的映射关系，表项数目等于逻辑寄存器数目
- CAM：Conent Addressed Memory，保存每个物理寄存器的映射关系，表项数目等于物理寄存器数目

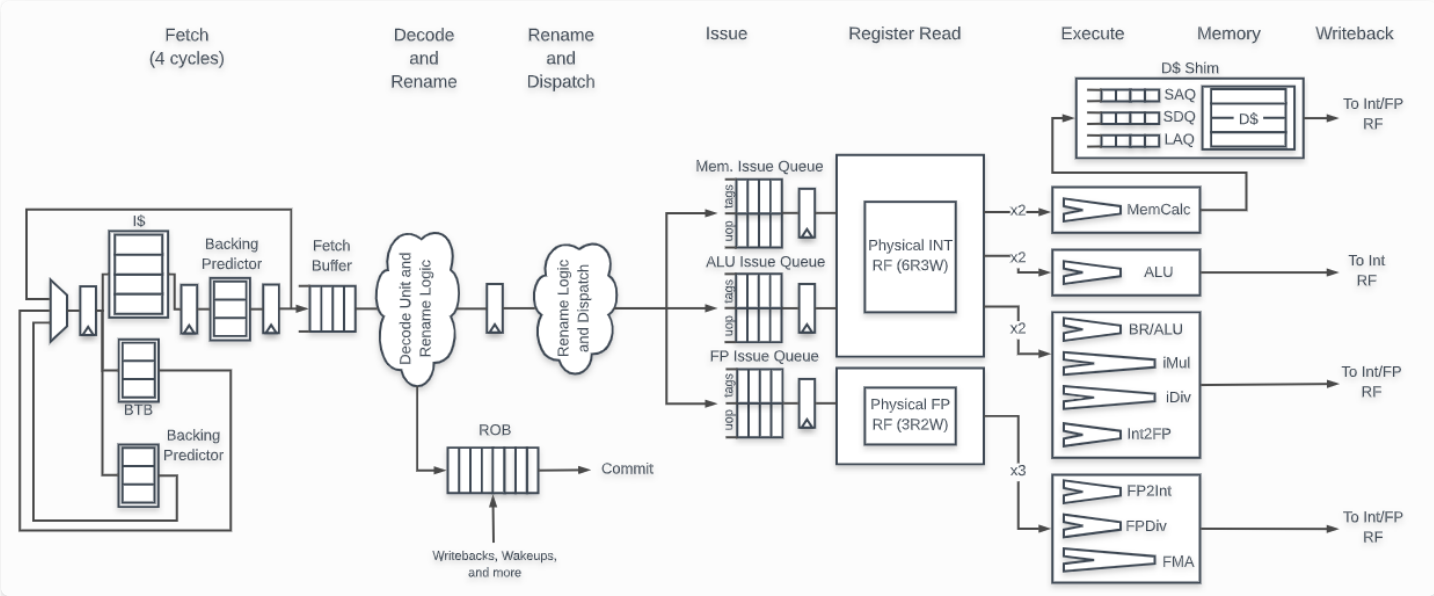
在 CAM 结构中，由于一个物理寄存器可能与多个物理寄存器存在映射关系，必须对表中的每一项加标志位，表示是否为最新的映射关系。RAM 结构的重命名映射表表项数目更少(因为物理寄存器个数大于逻辑寄存器)，且更适合乱序执行的处理

器。

BOOM 的重命名阶段使用的是 RAM 结构的映射表。

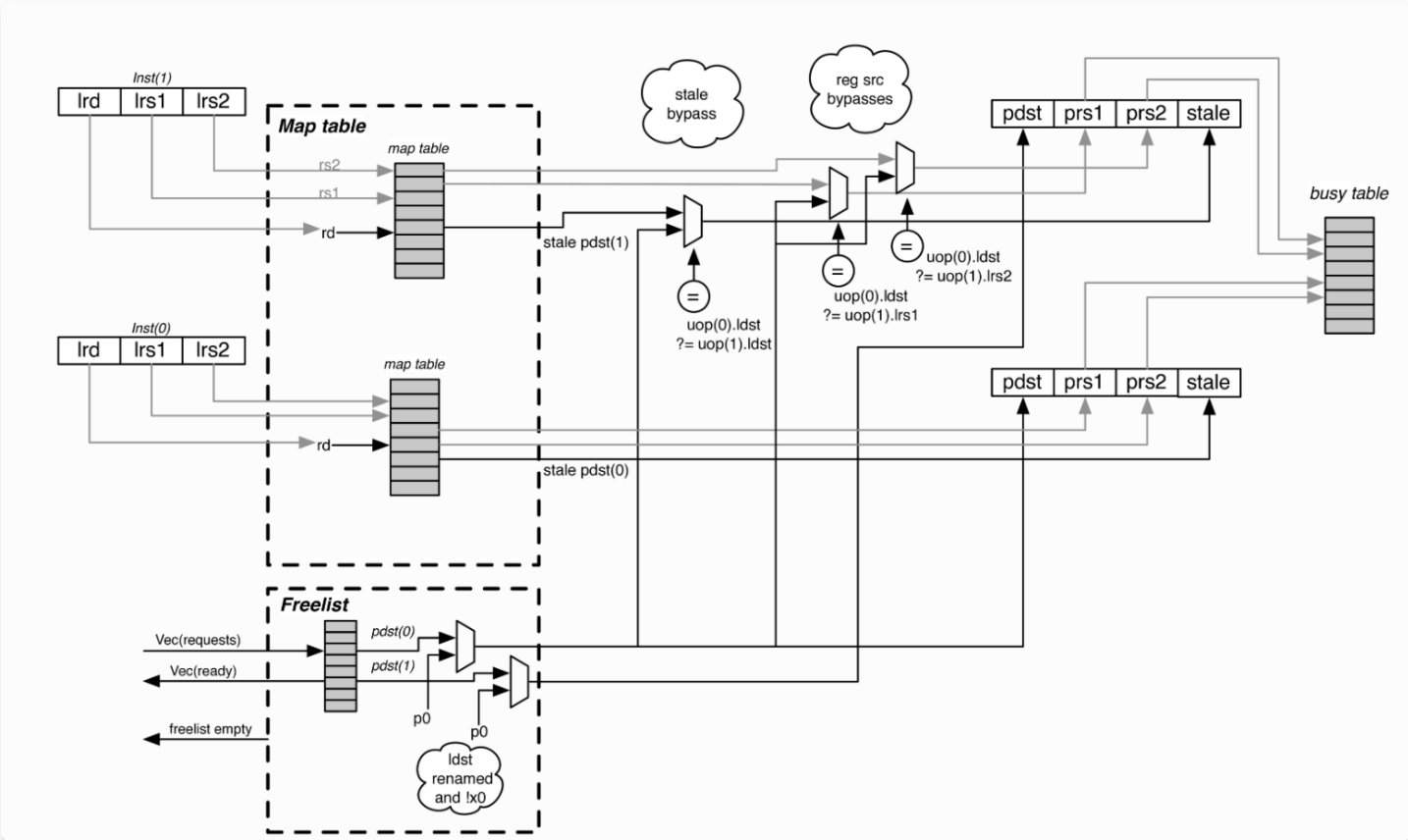
BOOM 处理器概况

BOOM(Berkeley Out-of-Order Machine) 是一款 RISC-V 指令集的开源的处理器，使用chisel 语言设计。BOOM 的流水线共分为十级，实现超标量乱序执行功能。对BOOM 流水线的详细介绍可以参考官方文档 The BOOM Pipeline。



BOOM 流水线

具体到 Rename 阶段， BOOM 采用显式重命名方案，共包括 MapTable、FreeList、BusyTable 三个模块以及顶层模块 RenameStage。



BOOM重命名流程图

Rename 阶段在流水线中占用两个时钟周期，第一个时钟周期中 freelist 和 maptable 模块将译码级传入的微指令中的逻辑寄存器映射为物理寄存器，第二个时钟周期通过 busytable 判断源操作数寄存器是否可读并更新busytable。

下面将结合代码详细介绍BOOM处理器对rename流水级的设计。

MapTable

MapTable模块的主要任务是维护寄存器阵列 `map_table` ， `map_table` 中保存每个逻辑寄存器对应的物理寄存器。Maptable要将微指令中的源操作数寄存器映射为合适的物理寄存器，除了读取和更新映射表之外，还必须考虑的问题有：

- 如何响应重映射请求?
- 如何响应回滚信号?
- 分支预测失误怎么办?
- 进行寄存器映射的同时，如何维护指令间的 RAW 依赖关系?

读者将从下文对BOOM代码的具体分析中找到这些问题的答案。

1.对 map_table 的修改

对 `map_table` 的修改依据分支预测的结果进行:

- 分支预测正确，更新 `map_table`
- 分支预测失误，将 `map_table` 恢复为分支快照中保存的寄存器映射关系

```
when (io.brupdate.b2.mispredict) {
  map_table := br_snapshots(io.brupdate.b2.uop.br_tag)
} .otherwise {
  map_table := remap_table(plWidth)
}
```

更新映射关系

最新的映射关系通过 remap_table 传入 map_table

```
val remap_table = Wire(Vec(plWidth+1, Vec(numLregs, UInt(pregSz.W))))
...
for (i <- 0 until numLregs) {     //遍历每个逻辑寄存器
  if (i == 0 && !float) {
    for (j <- 0 until plWidth+1) {
      remap_table(j)(i) := 0.U       //寄存器x0的值固定为0
    }
  } else {
    val remapped_row = (remap_ldsts_oh.map(ldst => ldst(i)) zip remap_pdsts)
       .scanLeft(map_table(i)) {case (pdst, (ldst, new_pdst)) => Mux(ldst, new_pdst, pdst)}
     for (j <- 0 until plWidth+1) {
      remap_table(j)(i) := remapped_row(j)
    }
  }
}
```

scanLeft函数从左向右进行操作，操作由数据选择器 `Mux(ldst, new_pdst, pdst)` 实现,其中：

- `ldst` 来自 `remap_ldst_oh` ， 标志是否需要重映射；
- `new_pdst` 是第i个逻辑寄存器在 `io.remap_reqs` 中对应的的物理寄存器编码；
- `pdst` 是上一次 `scanleft` 操作的结果，初始值为 `map_table(i)` ， 即第i个逻辑目的寄存器在 `map_table` 中对应的物理寄存器编码

对scanLeft函数的疑问请移步 从BOOM源码学习Chisel语言的设计技巧

通过该部分语句的作用是响应重映射请求 `io.remap_reqs` ，确定逻辑寄存器与物理寄存器之间的映射关系。scanLeft函数的使用维护了指令间的数据依赖关系。

分支快照与恢复

大多数处理器都实现了分支预测的功能，BOOM也不例外。分支标签 br_tag 记录该指令与分支的依赖关系，当分支预测失误时，依赖于该分支的所有指令(通过分支标签检索) 将被清除，其对寄存器的修改也将被恢复到执行分支指令前的状态。分支快照是指保存执行分支指令之前各个寄存器的值，在分支预测失误时用于恢复逻辑。

```
val br_snapshots = Reg(Vec(maxBrCount, Vec(numLregs, UInt(pregSz.W))))
...
for (i <- 0 until plWidth) {
    when (io.ren_br_tags(i).valid) {
    br_snapshots(io.ren_br_tags(i).bits) := remap_table(i+1)
  }
}
```

值得一提的是，由于 `scanLeft` 的特性， `remapped_row` 中元素的个数为 `plWidth+1` ,第 i 条指令对应的逻辑-物理寄存器映射关系保存在 `remapped_row(i+1)` 中。

2. 读取映射关系

最终输出的物理寄存器与 `map_table` 记录的映射关系以及指令间的数据依赖有关。

```
for (i <- 0 until plWidth) {
  io.map_resps(i).prs1       := (0 until i).foldLeft(map_table(io.map_reqs(i).lrs1)) ((p,k) =>
    Mux(bypass.B && io.remap_reqs(k).valid && io.remap_reqs(k).ldst === io.map_reqs(i).lrs1, io.remap_reqs(k).pdst,
  io.map_resps(i).prs2       := (0 until i).foldLeft(map_table(io.map_reqs(i).lrs2)) ((p,k) =>
    Mux(bypass.B && io.remap_reqs(k).valid && io.remap_reqs(k).ldst === io.map_reqs(i).lrs2, io.remap_reqs(k).pdst,
  io.map_resps(i).prs3       := (0 until i).foldLeft(map_table(io.map_reqs(i).lrs3)) ((p,k) =>
    Mux(bypass.B && io.remap_reqs(k).valid && io.remap_reqs(k).ldst === io.map_reqs(i).lrs3, io.remap_reqs(k).pdst,
  io.map_resps(i).stale_pdst := (0 until i).foldLeft(map_table(io.map_reqs(i).ldst)) ((p,k) =>
    Mux(bypass.B && io.remap_reqs(k).valid && io.remap_reqs(k).ldst === io.map_reqs(i).ldst, io.remap_reqs(k).pdst,
  if (!float) io.map_resps(i).prs3 := DontCare
  }
```

遍历所有指令，对第i条指令，对 `(0 until i)` 调用 `foldLeft` 方法，操作为一个选择器
`Mux(bypass.B && io.remap_reqs(k).valid && io.remap_reqs(k).ldst === io.map_reqs(i).lrs1, io.remap_reqs(k).pdst, p))` ，其中：

- p为物理寄存器地址，初始值为 `map_table` 中lrs1对应的物理寄存器
- k是对历史指令的索引，索引范围是0~i

对foldLeft函数的疑问请移步 从BOOM源码学习Chisel语言的设计技巧

重映射有效时，遍历当前指令前的所有指令，若检测到 RAW 冲突，则当前指令的 prs/stale_pdst 当与发生 RAW 冲突的历史指令中 pdst 保持一致；否则将当前指令的 prs/stale_pdst 由 `map_table` 确定。对于非浮点操作，不需要使用 prs3。

BusyTable

Busytable的主要任务是维护一个寄存器阵列 `busy_table` ，用于保存所有物理寄存器的状态，指示物理寄存器是否可读，并实现 `busy_table` 的更新和读取。

寄存器被标记为busy(1有效)时说明该寄存器中的源操作数没有准备好，寄存器不可读、当前指令不可发射。

1. 更新busy_table

- unbusy:完成写回的寄存器置零
- rebusy:新分配的重命名寄存器置1

```
val busy_table_wb = busy_table & ~(io.wb_pdsts zip io.wb_valids)
    .map {case (pdst, valid) => UIntToOH(pdst) & Fill(numPregs, valid.asUInt)}.reduce(_|_)
val busy_table_next = busy_table_wb | (io.ren_uops zip io.rebusy_reqs)
    .map {case (uop, req) => UIntToOH(uop.pdst) & Fill(numPregs, req.asUInt)}.reduce(_|_)
```

2. 读busy_table

读取某一寄存器的状态的结果，与busytable和旁路逻辑有关。

- busytable记录了写回和重命名阶段对寄存器繁忙状态的影响
- 存在bypass逻辑时，说明当同一批指令之间存在数据依赖(RAW)，寄存器不可读。

```
io.busy_resps(i).prs1_busy := busy_table(io.ren_uops(i).prs1) || prs1_was_bypassed && bypass.B
io.busy_resps(i).prs2_busy := busy_table(io.ren_uops(i).prs2) || prs2_was_bypassed && bypass.B
io.busy_resps(i).prs3_busy := busy_table(io.ren_uops(i).prs3) || prs3_was_bypassed && bypass.B
```

- 第三个源操作数寄存器仅用于浮点操作，对于非浮点操作繁忙状态置为0

```
if (!float) io.busy_resps(i).prs3_busy := false.B
```

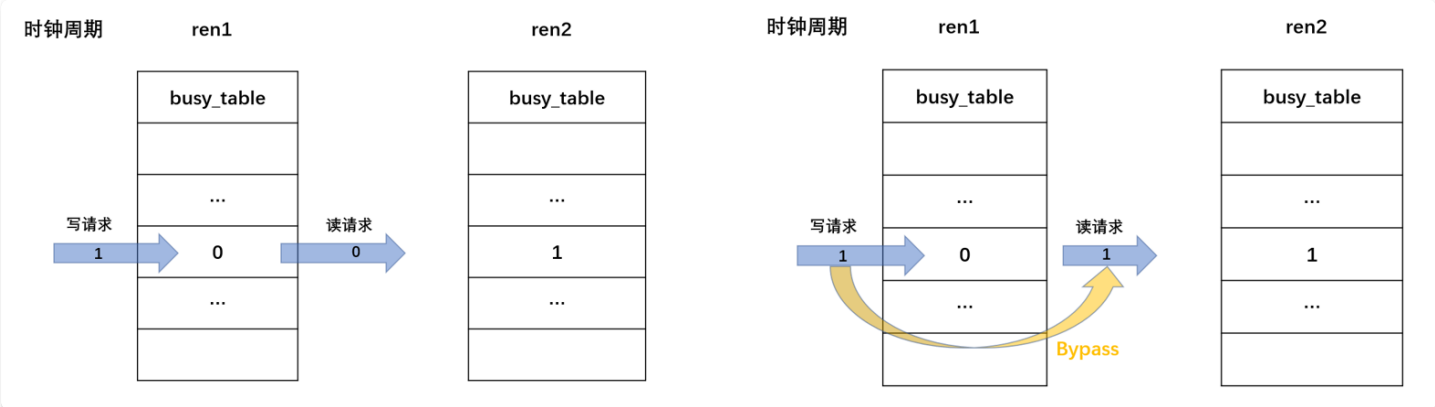
3. 旁路逻辑bypass

BusyTable模块中引入bypass逻辑要解决的问题是：传入rename阶段的同一批指令之间的数据依赖(RAW)无法识别。

上述问题的根源在于BOOM是一个超标量处理器。我们举例进行说明，传入rename阶段的一批指令包括两条(在BOOM中实际应为 plWidth 条)，且之间存在 RAW 冲突：

> add x1,x0,x2
> sub x3,x1,x3

在rename阶段的第二个时钟周期，add指令会将x1映射的物理寄存器，记为p(x1)，在 `busytable` 中的状态置为1(busy)，sub指令将读取p(x1)在 `busytable` 中的值判断源操作数寄存器是否可读。写操作在下一时钟周期才能完成对 `busytable` 的更新，因此当前读到的 `busytable[ p(x1) ]` 不是最新值，add和sub指令之间的RAW冲突无法识别。



Bypass 逻辑

bypass逻辑实现了数据前馈，当同一批指令之间检测到 RAW 冲突时，把将要写入 `busytable` 的值传递到对应的读端口，相当于数据流绕过了寄存器阵列，在当前时钟周期内读取到 `busytable` 更新后的值，识别RAW冲突。

```
val prs1_was_bypassed = (0 until i).map(j =>
      io.ren_uops(i).lrs1 === io.ren_uops(j).ldst && io.rebusy_reqs(j)).foldLeft(false.B)(_||_)
```

Freelist

Freelist模块主要维护寄存器状态表(free_list)、分支分配列表(br_alloc_lists)，记录各个物理寄存器的是否空闲，并根据外部输入信号更新寄存器状态、分配空闲的物理寄存器。
由Freelist分配给微指令的物理寄存器应当满足以下条件：

- 寄存器是可写的
- 寄存器是空闲的

Freelist模块在流水线中的主要作用就是从寄存器堆中找到满足上述要求的物理寄存器分配给微指令作为目的寄存器。

1. free_list的初始化与更新

free_list 保存每个物理寄存器的空闲状态，1表示空闲，初始化为111……110，宽度为numPregs。x0寄存器的值始终保持为0，表示始终不空闲，因为x0寄存器的值固定为0，不可用于分配作为目的寄存器写入新的值。

```
val free_list = RegInit(UInt(numPregs.W), ~(1.U(numPregs.W)))
```

free_list更新的逻辑是：

- 将被分配出去(sel_mask=1)的寄存器置0
- 将被释放(dealloc_mask=1)的寄存器置1
- x0寄存器始终不空闲

```
free_list := (free_list & ~sel_mask | dealloc_mask) & ~(1.U(numPregs.W))
```

sel_mask 记录寄存器是否被分配，是由sels和sel_fire组成的掩码。

sels 使用SelctFirstN函数生成，从free_list中寻找第一个值为1的位返回其独热码，并将该位的值修改为0，共执行plWidth次，即返回plWidth个独热码。

sel_fire 指示每条指令的请求是否能够得到满足，1有效。

```
val sels = SelectFirstN(free_list, plWidth)
val sel_fire  = Wire(Vec(plWidth, Bool()))
val sel_mask = (sels zip sel_fire) map { case (s,f) => s & Fill(n,f) } reduce(_|_)
```

dealloc_mask 保存需要释放的物理寄存器独热码集合，需要释放的寄存器包括：

- 在dealloc_pregs中声明释放的寄存器
- 因分支预测失败需要释放的寄存器

```
val dealloc_mask = io.dealloc_pregs.map(d => UIntToOH(d.bits)(numPregs-1,0) & Fill(n,d.valid)).reduce(_|_) | br_de
```

2. br_alloc_lists的初始化与更新

考虑到分支预测失误时，处于 “推测” 执行状态的指令被取消，分配给这些指令的寄存器也应恢复空闲状态，FreeList 模块必须记录与分支有关的指令的寄存器分配状况。

br_alloc_lists 保存对于每一个分支，物理寄存器的分配状况，1表示被分配。其更新逻辑是：

- 若分支指令有效，br_alloc_lists与alloc_masks保持一致
- 否则，将分支中已经释放的寄存器(br_deallocs)置0，新分配的寄存器(alloc_masks[0])置1。

```
val br_alloc_lists = Reg(Vec(maxBrCount, UInt(numPregs.W)))
br_alloc_lists(i) := Mux(new_list, Mux1H(list_req, alloc_masks.slice(1, plWidth+1)),
                                    br_alloc_lists(i) & ~br_deallocs | alloc_masks(0))
```

br_alloc_list：MaxBranchCount = 6

Branch \ Pregs	1	2	3	4	5	6	7	8
1	0	1	1	0	1	1	0	0
2	0	1	1	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	1	1	0	1	1	0	1
5	0	1	1	0	1	1	0	1
6	0	1	1	0	1	1	0	1

Tips:
The pregs in the latter branch is the subset of that in the former, because the predicated branch lists are logically hierarchical. If the upper-level branch prediction is prove false, then all its subsets as well as itself should be deallocated.

Records the newly allocated pregs contained in each branch context

br_alloc_list

new_list 指示每条指令是否为有效的分支指令

Mux1H(m,n) 独热码数据选择器作用是根据独热码m从序列n中选出唯一一个元素。

slice(m,n) 方法的作用是取出alloc_masks中第m-n个元素。

alloc_masks 记录不同的分支槽中寄存器的分配状况，alloc_masks[i] 表示需要分配给第 i 条指令之后的所有指令的寄存器独热码集合，alloc_masks[0]表示所有指令需要分配的寄存器独热码集合。对于越靠后的指令，需要新分配的寄存器越少，因此调用scanRight方法产生alloc_masks。

```
val allocs = io.alloc_pregs map (a => UIntToOH(a.bits))
val alloc_masks = (allocs zip io.reqs).scanRight(0.U(n.W)) { case ((a,r),m) => m | a & Fill(n,r) }
```

Rename Example (NumPregs=8)			
Rename List	Branch Tag	Tag Valid	Allocated Preg
Intr1	1	1	0000_0001
Intr2	1	0	0000_0100
Intr3	2	1	0000_1000
Intr4	3	1	0010_0000
Intr5	3	0	0100_0000
Tips: After ScanRight operation, the alloc_mask records the process of the allocation of pregs instructions by instructions, thus faciliating the logic when a branch slot is detected and a snapshot is to be taken.			

Alloc_mask
0110_1101 (0)
0110_1100 (1)
0110_1000 (2)
0110_0000 (3)
0100_0000 (4)
0000_0000 (5)
ScanRight!

alloc_masks

br_deallocs 保存的是预测错误的分支中空闲物理寄存器的独热码集合

```
val br_deallocs = br_alloc_lists(io.brupdate.b2.uop.br_tag) & Fill(n, io.brupdate.b2.mispredict)
```

这里出现了一个新类型的数据选择器 Mux1H ，我们不妨对chisel中数据选择器的硬件原语作一总结。

硬件原语	功能
Mux(sel,in1,in2)	sel=1时返回in1，sel=0时返回in2。
MuxCase(default, array)	default是默认返回值，arrary是一个数组，元素为(条件，返回值)。
MuxLookup(idx, default, array)	default是默认返回值，arrary是一个数组，元素为(idx取值，对应返回值)
Mux1H(sel,arrary)	独热码数据选择器，sel是独热码 ，arrary是一个元素个数不小于sel位宽的数组，由sel决定返回arrary中的第几个元素。
PriorityMux(sel,arrary)	优先数据选择器，返回sel中第一个有效值(1)对应的arrary中的元素。

chisel中的数据选择器

3. 流水线逻辑和输出端口的连接

sel_fire 是一个寄存器阵列，记录指令的请求是否得到满足(1表示满足)，其更新逻辑为：

- 若该条指令请求的空闲寄存器不可用，置0
- 若寄存器存在但未占用且指令不请求分配寄存器，置0

io.alloc_pregs.valid 与r_valid连接,对应该指令请求的物理寄存器可用状态。

io.alloc_pregs.bits 与r_sel连接，当该指令分配寄存器的请求能够得到满足时，返回分配的空闲寄存器的地址。

```
  for (w <- 0 until plWidth) {
    val can_sel = sels(w).orR
    val r_valid = RegInit(false.B)
    val r_sel   = RegEnable(OHToUInt(sels(w)), sel_fire(w))
    r_valid := r_valid && !io.reqs(w) || can_sel
    sel_fire(w) := (!r_valid || io.reqs(w)) && can_sel
    io.alloc_pregs(w).bits  := r_sel
    io.alloc_pregs(w).valid := r_valid
  }
```

RenameStage

RenameStage 描述了整个重命名阶段的操作流程，包括rename模块与外部信号的连接、内部子模块之间的互联、流水线时序逻辑等。

1. rename模块与外部信号的连接

rename模块与其他模块的连接主要在 ***AbstractRenameStage*** 类中实现

```
//总体控制信号
    val ren_stalls = Output(Vec(plWidth, Bool()))         //流水线暂停
    val kill = Input(Bool())                              //中止信号
//从decode输入的信息
    val dec_fire  = Input(Vec(plWidth, Bool()))           // will commit state updates
    val dec_uops  = Input(Vec(plWidth, new MicroOp()))    //译码级输出的微指令
//rename阶段输出的信息
    val ren2_mask = Vec(plWidth, Output(Bool()))          // mask of valid instructions
    val ren2_uops = Vec(plWidth, Output(new MicroOp()))   //输出的微指令
// 从execute阶段输入的分支指令信息
    val brupdate = Input(new BrUpdateInfo())
//从dispatch阶段输入的信息
    val dis_fire  = Input(Vec(coreWidth, Bool()))         //派遣级各支完成指令派遣的信号
    val dis_ready = Input(Bool())                         //派遣级可以接收数据的ready信号

    // wakeup ports
    val wakeups = Flipped(Vec(numWbPorts, Valid(new ExeUnitResp(xLen))))
//从commit阶段输入的信息
    val com_valids = Input(Vec(plWidth, Bool()))          //指令是否提交的信号
    val com_uops = Input(Vec(plWidth, new MicroOp()))     //提交阶段的微指令
    val rbk_valids = Input(Vec(plWidth, Bool()))          //指令是否回滚的信号
    val rollback = Input(Bool())                          //回滚的总使能信号
//调试信息
    val debug_rob_empty = Input(Bool())
    val debug = Output(new DebugRenameStageIO(numPhysRegs))
```

ren2_uop 输出重命名之后的微指令，信息保存在寄存器 r_uop 中

```
r_uop := GetNewUopAndBrMask(BypassAllocations(next_uop, ren2_uops, ren2_alloc_reqs), io.brupdate)
```

GetNewUopAndBrMask 的作用是根据输入的分支信息更新 uop 中的 br_mask

```
object GetNewUopAndBrMask
{
  def apply(uop: MicroOp, brupdate: BrUpdateInfo)
    (implicit p: Parameters): MicroOp = {
    val newuop = WireInit(uop)
    newuop.br_mask := uop.br_mask & ~brupdate.b1.resolve_mask
    newuop
  }
}
```

BypassAllocations 的作用是根据历史指令处理微操作uop，将发生RAW冲突的源操作数寄存器替换成对应的历史指令目的寄存器，并将需要bypass的寄存器的状态置为busy

```
def BypassAllocations(uop: MicroOp, older_uops: Seq[MicroOp], alloc_reqs: Seq[Bool]): MicroOp = {
    val bypassed_uop = Wire(new MicroOp)
    bypassed_uop := uop
    //找出目的寄存器与当前源操作数寄存器相同的全部历史指令（即需要bypass），向后拓展
    val bypass_hits_rs1 = (older_uops zip alloc_reqs) map { case (r,a) => a && r.ldst === uop.lrs1 }
    val bypass_hits_rs2 = (older_uops zip alloc_reqs) map { case (r,a) => a && r.ldst === uop.lrs2 }
    val bypass_hits_rs3 = (older_uops zip alloc_reqs) map { case (r,a) => a && r.ldst === uop.lrs3 }
    val bypass_hits_dst = (older_uops zip alloc_reqs) map { case (r,a) => a && r.ldst === uop.ldst }

    //返回命中指令中最近的指令的独热码
    val bypass_sel_rs1 = PriorityEncoderOH(bypass_hits_rs1.reverse).reverse
    val bypass_sel_rs2 = PriorityEncoderOH(bypass_hits_rs2.reverse).reverse
    val bypass_sel_rs3 = PriorityEncoderOH(bypass_hits_rs3.reverse).reverse
    val bypass_sel_dst = PriorityEncoderOH(bypass_hits_dst.reverse).reverse

    //返回1/0表示有无bypass
    val do_bypass_rs1 = bypass_hits_rs1.reduce(_||_)
    val do_bypass_rs2 = bypass_hits_rs2.reduce(_||_)
    val do_bypass_rs3 = bypass_hits_rs3.reduce(_||_)
    val do_bypass_dst = bypass_hits_dst.reduce(_||_)

    //选出需要旁路逻辑的寄存器
    val bypass_pdsts = older_uops.map(_.pdst)

    when (do_bypass_rs1) { bypassed_uop.prs1       := Mux1H(bypass_sel_rs1, bypass_pdsts) }
    when (do_bypass_rs2) { bypassed_uop.prs2       := Mux1H(bypass_sel_rs2, bypass_pdsts) }
    when (do_bypass_rs3) { bypassed_uop.prs3       := Mux1H(bypass_sel_rs3, bypass_pdsts) }
    when (do_bypass_dst) { bypassed_uop.stale_pdst := Mux1H(bypass_sel_dst, bypass_pdsts) }

    //如果需要进行bypass，该寄存器设置为busy
    bypassed_uop.prs1_busy := uop.prs1_busy || do_bypass_rs1
    bypassed_uop.prs2_busy := uop.prs2_busy || do_bypass_rs2
    bypassed_uop.prs3_busy := uop.prs3_busy || do_bypass_rs3

    //浮点操作
    if (!float) {
      bypassed_uop.prs3      := DontCare
      bypassed_uop.prs3_busy := false.B
    }

    //BypassAllocations的返回值为bypassed_uop
    bypassed_uop
  }
```

ren2_valid 输出微指令的有效信号，信息保存在寄存器 r_valid 中，其更新逻辑为:

- 外部终止 (kill) 信号有效时，r_valid置0
- kill 信号无效，派遣级准备好接受新的微指令时，r_valid 与译码级的输入的指令有效信号(ren1_fire) 保持一致，并将next_uop置为decode阶段输入的微指令 (ren1_uop)

- kill信号无效，派遣级尚未准备好接受新的微指令时，若前一条指令已经完成派遣(ren2_fire=1) 将 r_valid 置0，否则保持不变。

```
for (w <- 0 until plWidth) {
  ren1_fire(w)          := io.dec_fire(w)  //译码完成
  ren1_uops(w)          := io.dec_uops(w)  //译码得到的微指令
}

for (w <- 0 until plWidth) {
  val r_valid  = RegInit(false.B)       //寄存器输出的uop有效信号
  val r_uop    = Reg(new MicroOp)
  val next_uop = Wire(new MicroOp)      //uop更新的输入，默认保持为r_uop
  next_uop := r_uop
  when (io.kill) {  //外部的中止指令
    r_valid := false.B
  } .elsewhen (ren2_ready) { //派遣准备好接收新的微指令，next_uop更新为下一条指令译码得到的微指令
    r_valid := ren1_fire(w)
    next_uop := ren1_uops(w)
  } .otherwise {    //派遣级没准备好接受新的微指令，若前一条指令已经完成派遣，r_valid信号置0，next_uop保持
    r_valid := r_valid && !ren2_fire(w) // clear bit if uop gets dispatched
    next_uop := r_uop
  }
  r_uop := GetNewUopAndBrMask(BypassAllocations(next_uop, ren2_uops, ren2_alloc_reqs), io.brupdate)
  //dispatch
  ren2_valids(w) := r_valid
  ren2_uops(w)   := r_uop
}
```

2. 子模块的实例化与互联

子模块的实例化

```
  val maptable = Module(new RenameMapTable(
    plWidth,
    32,
    numPhysRegs,
    false,
    float))
  val freelist = Module(new RenameFreeList(
    plWidth,
    numPhysRegs,
    if (float) 32 else 31))
  val busytable = Module(new RenameBusyTable(
    plWidth,
    numPhysRegs,
    numWbPorts,
    false,
    float))
```

maptable的连接

值得注意的是输出的微指令中目的寄存器是“僵死的(stale)”，新分配的寄存器将从freelist中得到。

```
  //maptable inputs
  maptable.io.map_reqs    := map_reqs
  maptable.io.remap_reqs  := remap_reqs
  maptable.io.ren_br_tags := ren2_br_tags
  maptable.io.brupdate      := io.brupdate
  maptable.io.rollback    := io.rollback
  // Maptable outputs.
  for ((uop, w) <- ren1_uops.zipWithIndex) {
    val mappings = maptable.io.map_resps(w)
    uop.prs1       := mappings.prs1
    uop.prs2       := mappings.prs2
    uop.prs3       := mappings.prs3 // only FP has 3rd operand
    uop.stale_pdst := mappings.stale_pdst
  }
```

map_reqs 的信息来自 ren1,即译码级传来的微指令

remap_reqs 由数据选择器实现：

- 当请求回滚时，需要重命名的寄存器时来自commit阶段的微指令中的物理寄存器，将其重映射为stale_pdst，即恢复之前的映射关系
- 若不请求回滚，需要重命名的寄存器来自dispatch阶段传来的微指令

```
remap_reqs(w).ldst := Mux(io.rollback, com.ldst      , ren2.ldst)
remap_reqs(w).pdst := Mux(io.rollback, com.stale_pdst, ren2.pdst)
```

rbk_valid 重映射请求有效的条件：从译码级传来的指令请求分配新的物理寄存器，或者，从提交级传来的指令请求回滚

```
ren2_alloc_reqs zip rbk_valids.reverse zip remap_reqs map {
    case ((a,r),rr) => rr.valid := a || r}
```

freelist的连接

freelist的输入 主要包括寄存器的分配和释放请求，以及分支信息、调试信息等。

req 寄存器分配请求有效必须同时满足：

- 译码的目的寄存器有效
- 寄存器类型符合配置
- 派遣级取出微指令

```
 for (w <- 0 until plWidth) {
    ren2_alloc_reqs(w)    := ren2_uops(w).ldst_val && ren2_uops(w).dst_rtype === rtype && ren2_fire(w)
    ...
  }
  ...
  freelist.io.reqs := ren2_alloc_reqs
```

dealloc_pregs 寄存器释放请求，包括释放有效信号 valid 和释放的物理寄存器编码 bits

当roolback或commit有效时释放请求 dealloc_pregs.valid 有效；

- 发生回滚时，需要释放的寄存器时重命名分配的寄存器，因为此时执行恢复逻辑，目的寄存器映射的物理寄存器恢复为stale_pdst；
- 不发生回滚时，需要释放的寄存器则为stale_pdst

```
freelist.io.dealloc_pregs zip com_valids zip rbk_valids map   /
  {case ((d,c),r) => d.valid := c || r}
freelist.io.dealloc_pregs zip io.com_uops map
  {case (d,c) => d.bits := Mux(io.rollback, c.pdst, c.stale_pdst)}
```

freelist的输出 是分配的寄存器，当目的寄存器不为x0,或为浮点操作时输出新分配的寄存器，否则输出0.U

```
  for ((uop, w) <- ren2_uops.zipWithIndex) {
    val preg = freelist.io.alloc_pregs(w).bits
    uop.pdst := Mux(uop.ldst =/= 0.U || float.B, preg, 0.U)
  }
```

busytable的连接

输入

rebusy_reqs 端口与ren2_alloc_reqs连接，对于新一批重命名的指令，将新分配的寄存器标记为busy

wb_valid 端口与外部信号wakeups连接，完成写回操作时，写回的目的寄存器的busy状态将被清除

ren_uops 端口与ren2_uops 连接，微指令ren2_uops 中的目的寄存器将在此阶段被分配空闲的物理寄存器

```
  busytable.io.ren_uops := ren2_uops                          //目的寄存器待分配
  busytable.io.rebusy_reqs := ren2_alloc_reqs                 //busy
  busytable.io.wb_valids := io.wakeups.map(_.valid)           //unbusy
  busytable.io.wb_pdsts := io.wakeups.map(_.bits.uop.pdst)    //写回目的寄存器的地址
```

输出

ren_stall： 寄存器类型不匹配或者无法为当前指令分配寄存器(can_allocate=0)时，流水线暂停

调用BypassAllocations函数，实现旁路读取数据，产生旁路微指令 ***bypassed_uops***

ren2_uop 端口，连接根据分支信息更新后的旁路微指令

```
  for (w <- 0 until plWidth) {
    val can_allocate = freelist.io.alloc_pregs(w).valid
    io.ren_stalls(w) := (ren2_uops(w).dst_rtype === rtype) && !can_allocate
    val bypassed_uop = Wire(new MicroOp)
    if (w > 0) bypassed_uop := BypassAllocations(ren2_uops(w), ren2_uops.slice(0,w), ren2_alloc_reqs.slice(0,w))
    else       bypassed_uop := ren2_uops(w)
    io.ren2_uops(w) := GetNewUopAndBrMask(bypassed_uop, io.brupdate)
  }
```

3. *PredRenameStage

PredRenameStage 继承于AbstractRenameStage 类，主要功能是进行SFB优化，实现预测性的重命名。

busytable: 该类中实现了一个简单的 busy_table，保存的是取指目标队列中每条指令源操作数寄存器的繁忙状况，通过 to_busy 信号和 unbusy 信号对 busy_table 进行更新。

- to_busy 信号与SFB优化有关，当取指目标队列中的指令是可以进行SFB优化的分支指令时，该指令的目的寄存器在to_busy中对应的值置1。
- unbusy 信号与从写回级输入的信息有关，当寄存器完成写回时，其在unbusy中的值置1。

```
busy_table := ((busy_table.asUInt | to_busy.asUInt) & ~unbusy.asUInt).
```

SFB 优化可以分成两类：

- 针对逻辑算数运算进行优化，以 is_sfb_shadow 为有效信号

```
val is_sfb_shadow = ren2_uops(w).is_sfb_shadow && ren2_fire(w)
...
when (is_sfb_shadow) {
    io.ren2_uops(w).ppred := next_ftq_idx
    io.ren2_uops(w).ppred_busy := (busy_table(next_ftq_idx) || to_busy(next_ftq_idx)) && !unbusy(next_ftq_idx)
  }
```

- 针对分支指令进行优化，以 is_sfb_br 为有效信号

```
val is_sfb_br = ren2_uops(w).is_sfb_br && ren2_fire(w)
...
when (is_sfb_br) {
    io.ren2_uops(w).pdst := ftq_idx
    to_busy(ftq_idx) := true.B
  }
```

SFB优化的具体内容本篇不做深入。

本文小结

本文介绍了重命名技术的背景知识，结合源代码详细介绍了 BOOM 处理器 rename 流水级的设计。

参考资料

[1] RISCV-BOOM’s documentation