高等计算机系统结构

现代指令级并行技术

（第五讲）

汤姆苏鲁算法的动机非常明确，一个周期要issue一条指令

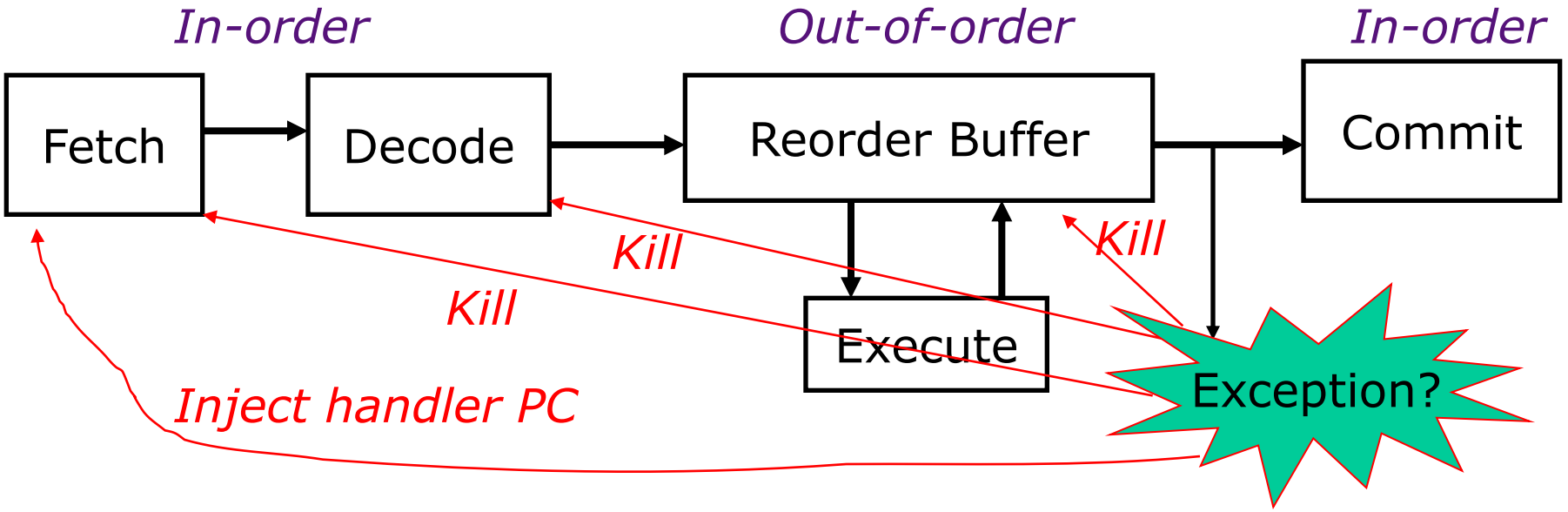
程 旭

2014年4月21日

流水线的性能

- 通过更加复杂的流水线和动态调度开发隐形（*implicit*）*指令级并行性*
- 乱序执行执行，同时保证：
  - 真数据相关(RAW)
  - 精确中断
- 通过寄存器换名，消除WAR和WAW冒险
- 重排序缓冲器（Reorder buffer）保存尚未提交（committing）但已完成的结果，以支持精确中断
- 频繁出现的转移指令会产生控制冒险，从而限制性能的改进

指令流水线的总体结构



- 取指和译码进入指令重排序缓冲器是按序进行的
- 执行是乱序的⇒ 乱序完成
- 提交（*Commit*：回写道体系结构级的状态，即寄存器对 &存储器）按序

在提交之前，需要临时存储来保存结果 (影子寄存器和存储缓冲器)

控制流导致的性能损失

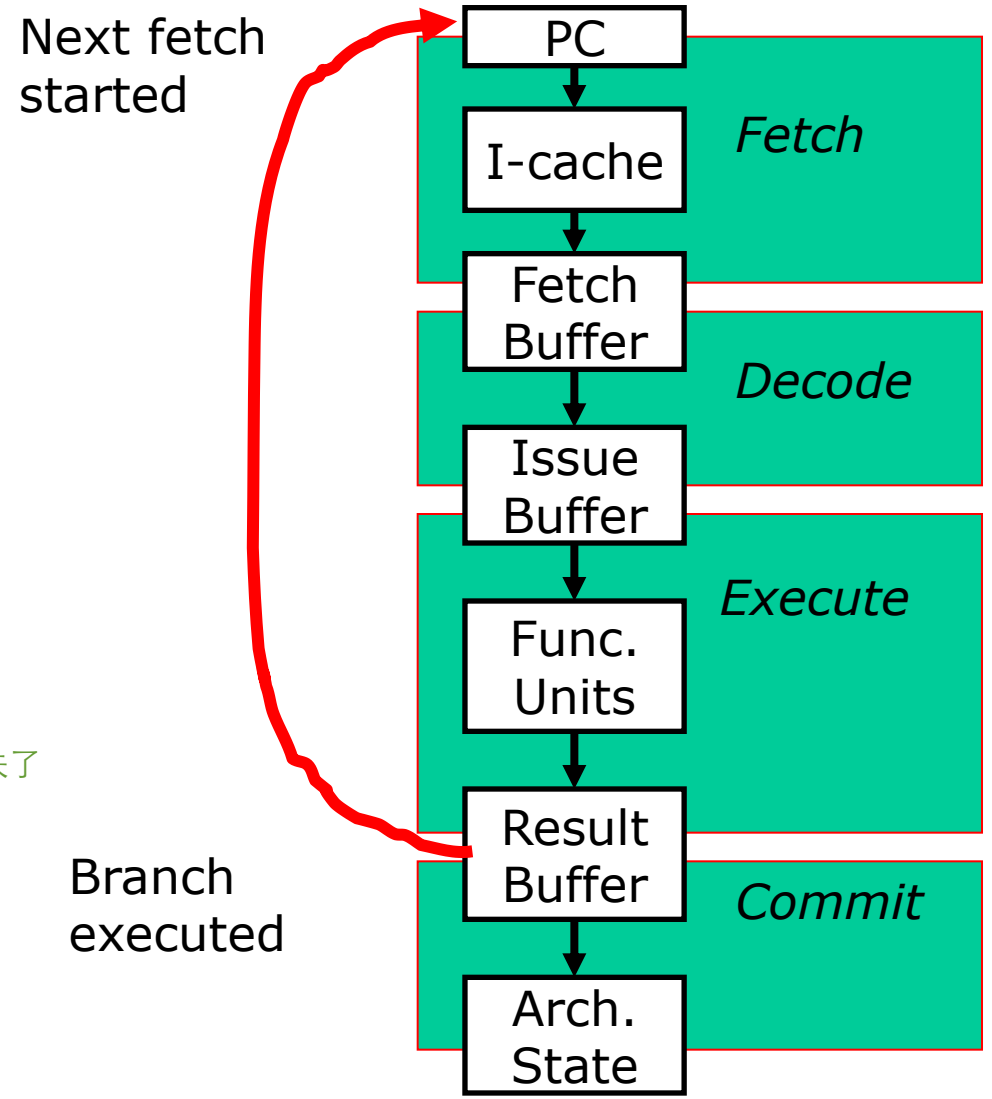
在许多现代处理器中，在下一PC计算和最终确定转移结果之间有 10 个以上的流水级!

奔腾都20多级流水呢，流水越深，转移指令带来的损失越大
超标量就是流水线不仅更深，而且更宽了

如果流水线不能及时选择正确指令，会导致多少损失?

~ Loop length x pipeline width

一旦有一个转移损失，那就是loop * width条指令被损失了



MIPS的转移和跳转

每条指令的取指都依赖于之前指令的一或二项信息:

1) 之前的那条指令是发生转移的指令吗（taken branch）?

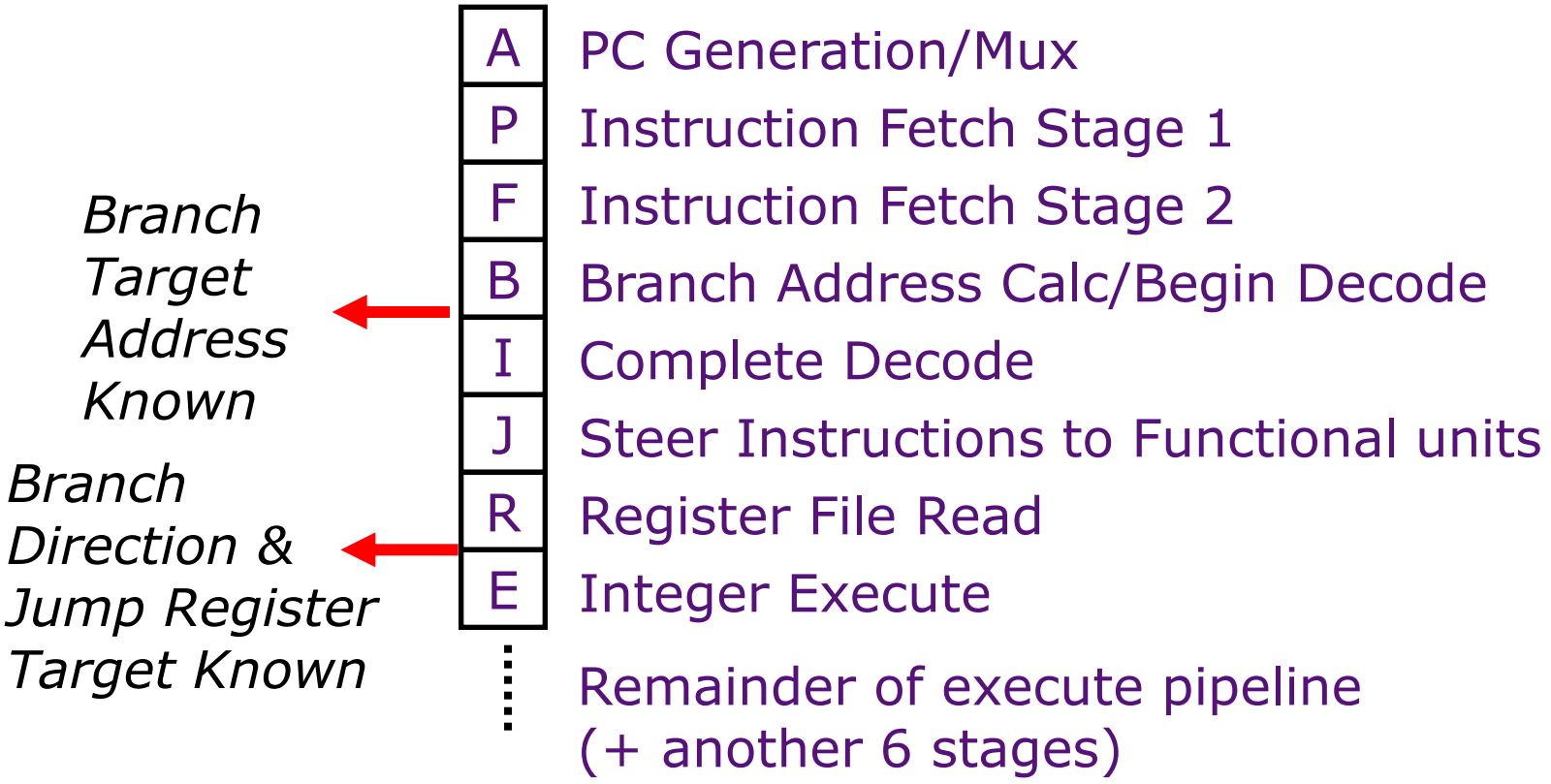
2) 如果是，转移目标地址是什么?

Instruction	*Taken known?*	*Target known?*
J 直接跳转指令 一译码就知道是否跳转、跳转到哪里	After Inst. Decode	After Inst. Decode
JR 间接跳转指令的地址一般放在寄存器 译码完了之后只知道是否跳转，但必须 读完寄存器才知道跳转目标是啥	After Inst. Decode	After Reg. Fetch
BEQZ/BNEZ 条件跳转指令，根据某寄存器里头的值为1还是为0来决定是否跳转	After Reg. Fetch*	After Inst. Decode

*假设在寄存器读时判断是否为 “0”

深度指令流水线中的转移损失

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)



降低转移损失

软件解决方案

- *消除转移 – 循环展开（loop unrolling）*
  增大运行长度（run length）增大基本块的大小，坏处是code size剧烈增大
- *较小转移确定的时间（resolution time）：指令调度*
  尽早计算转移条件

硬件解决方案

- 发现其他一些可以做的事 – 延迟槽（*delay slots*）
  用有效工作替换流水线中的空泡
  (需要软件协助)
- *推测（Speculate） - 转移预测*
  跨越转移的指令推测式执行（Speculative execution）
  通过猜测，从跳转指令后边调一些指令放在brunch slot中执行，猜对就用，猜错就抛弃
  要做brunch prediction，要知道target
  target要么你算出来，要么你猜出来

转移预测

动机:

转移损失（Branch penalties）制约了深度流水化处理器的性能提升

现代转移预测器具有很好的正确率(>95%)，可望显著减少转移损失

需要硬件支持: 无非是两方面的支持，一是我要能够预测，二是如果预测错了我要能够回退
前者就靠BHT和BTB，后者就靠Reorder buffer

预测结构部件:

- 转移历史表（Branch history tables），转移目标缓冲器（branch target buffers）等

错误预测恢复机制:

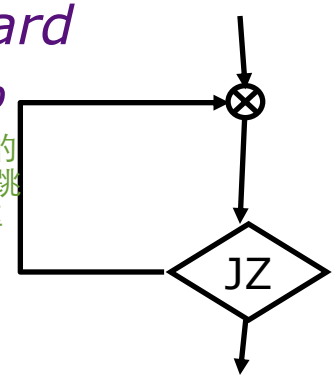
- *将结果计算与确认（commit）分离开来*
- 消除流水线中跟随错误预测转移指令的指令
- 将状态恢复到转移指令之后的正确状态

静态转移预测

总体而言，一条转移指令发生的概率大约 为60-70%，但是:

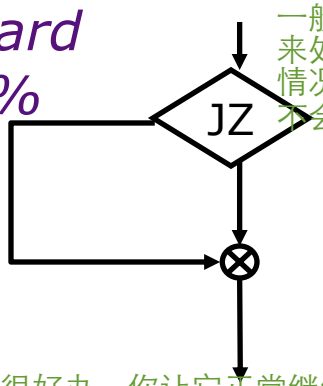
backward
90%

往回跳的情况一般都是loop
而一个loop是要往回跳很多次的
只在最后一轮循环才不再往回跳
因此往回跳转指令taken的概率很高，大约90%



forward
50%

一般来说，写程序的习惯是用不跳转来处理正常情况，用跳转来处理特殊情况，因此往前跳转的指令一般来说是不会taken的？存疑



预测不taken很好办，你让它正常继续执行就好
但若预测taken了，你还得找到target

ISA也可以向转移指令附加上首选转移方向的语义，例如 Motorola MC88110

bne0 *(preferred taken)*　　beq0 *(not taken)*

同是跳转指令，你用前者来写程序，处理器就默认预测taken，你用后者来写程序，处理器就默认not taken
程序员决定采用什么指令时，就已经向处理器传达了是否倾向于taken的语义

动态转移预测

learning based on past behavior

时间关联 (Temporal correlation)

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

动态预测要从两个方向下手，一个时间关联，一个空间关联

空间关联（Spatial correlation）

Several branches may resolve in a highly correlated manner *(a preferred path of execution)*

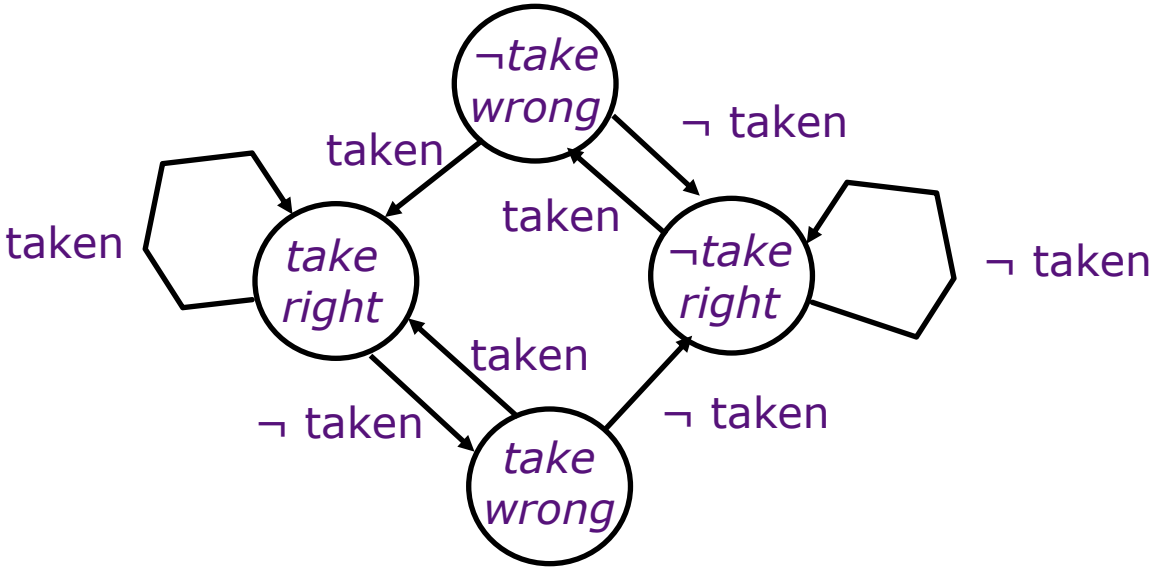
程序是程序流图动态执行的结果，到底是由哪一条路径到达这一指令，可能会决定这一跳转指令的跳转方向。比如连续两个跳转指令，多次运行时都是第一个taken了，第二个没有taken，那很可能第二个的not taken是由于第一个taken而决定了的，也就是说，多个跳转语句之间是有一定的相互影响的。时间关联是程序自身的事情，

转移预测位 这是动态的硬件预测器

Branch Prediction Bits

- 假设每条指令2个转移预测位
- 当连续两次出现转移预测错误时，改变预测方向!

如果只有one bit，就是上次如果taken了，就记住，下次还是预测它taken



状态机如何设置，没有绝对的好坏，哪种最好，你拿真正的程序来测一测嘛。

BP state:

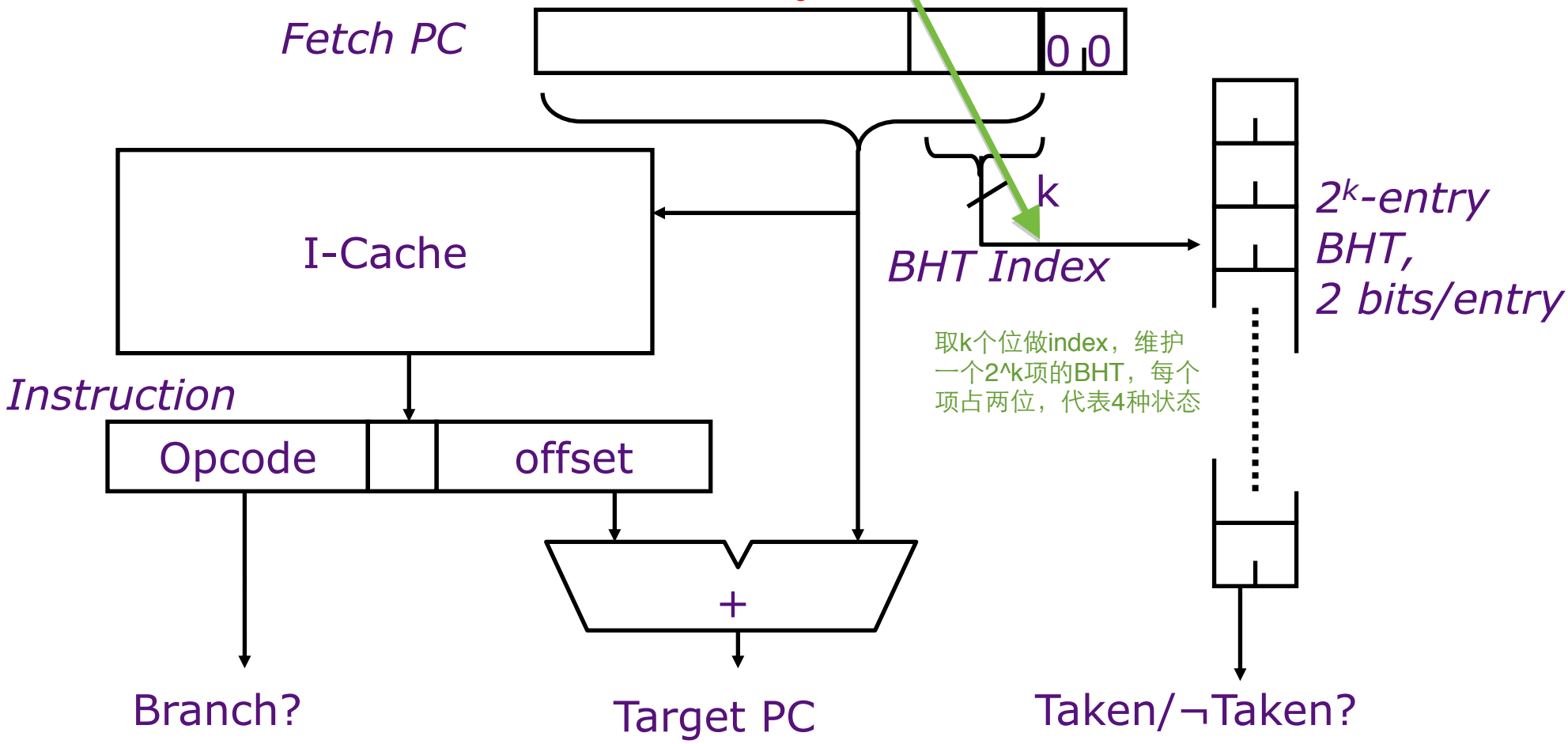
(*predict* take/¬take) x (*last prediction* right/wrong)

同一条指令的PC值一般是不会变的，因此猜测的时候就可以用PC的值做index，根本不用取指和译码就可以查出猜测结果。
PC的值一般不变，单也有可能会变，如果变了那就当我猜错了嘛，反正是猜的，甚至都不用整个PC的值，就取PC的低若干位即可。

转移历史表

Branch History Table

0-2预测器，只用两个位描述4种状态，只根据自己的情况做预测。和其它跳转语句无关



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

开采转移的空间关联

Yeh and Patt, 1992

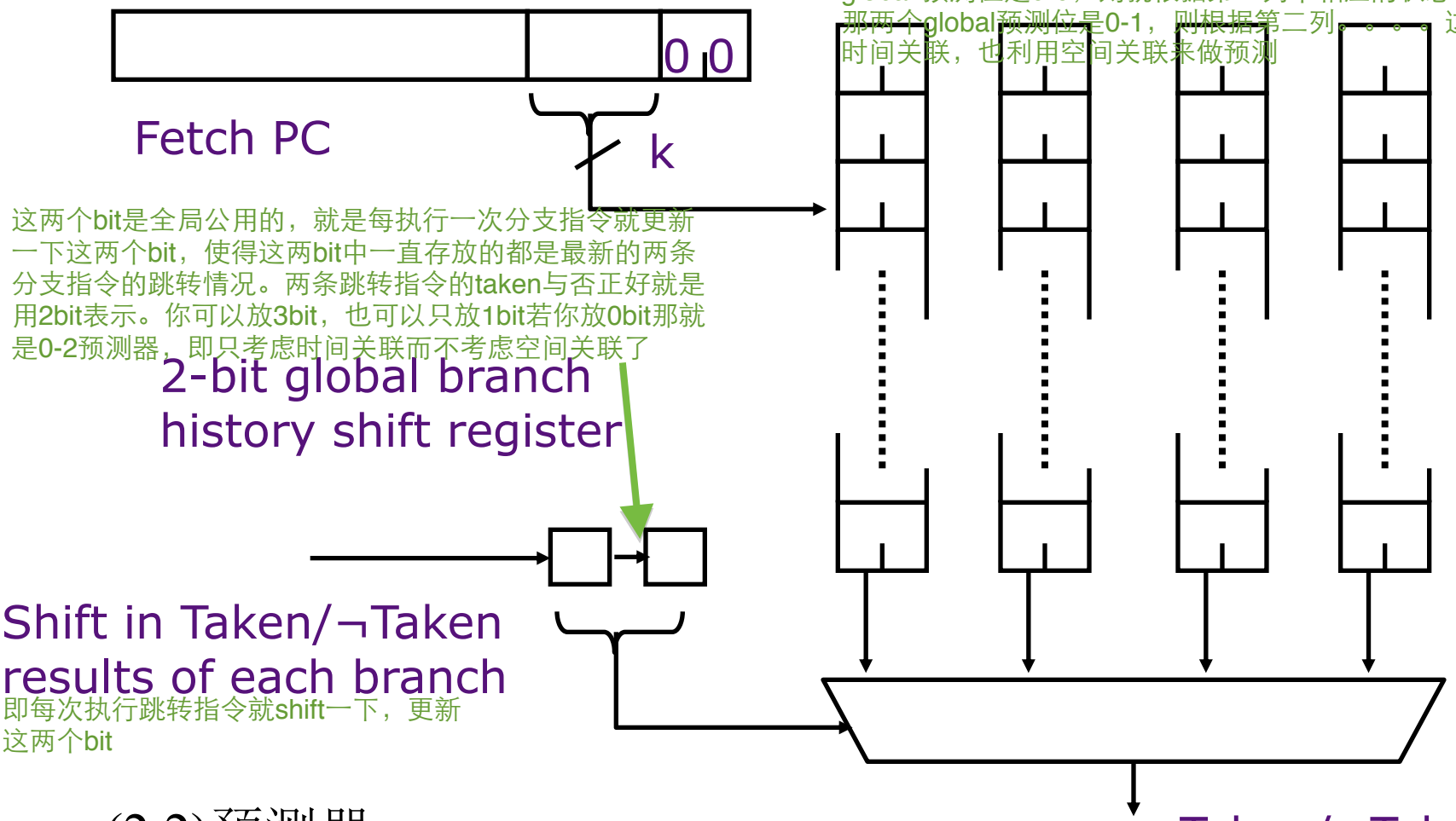
```
if (x[i] < 7) then
      y += 1;
if (x[i] < 5) then
      c -= 4;
```

如果第一转移的条件为“假”，第二个也一定为“假”

历史寄存器（*History register,* H,）记录处理器最近执行的N条转移的方向

两级转移预测器

Pentium Pro 通过利用最近两条转移的结果来从四组BHT位中挑选出一组 (~95% 的正确率)



在0-2预测器中，只有一列BHT，现在2-2预测器中有4列BHT，若那两个global 预测位是0-0，则就根据第一列中相应的状态来预测是否跳转，若那两个global预测位是0-1，则根据第二列。。。。这就实现了即利用时间关联，也利用空间关联来做预测

这两个bit是全局公用的，就是每执行一次分支指令就更新一下这两个bit，使得这两bit中一直存放的都是最新的两条分支指令的跳转情况。两条跳转指令的taken与否正好就是用2bit表示。你可以放3bit，也可以只放1bit若你放0bit那就是0-2预测器，即只考虑时间关联而不考虑空间关联了

即每次执行跳转指令就shift一下，更新这两个bit

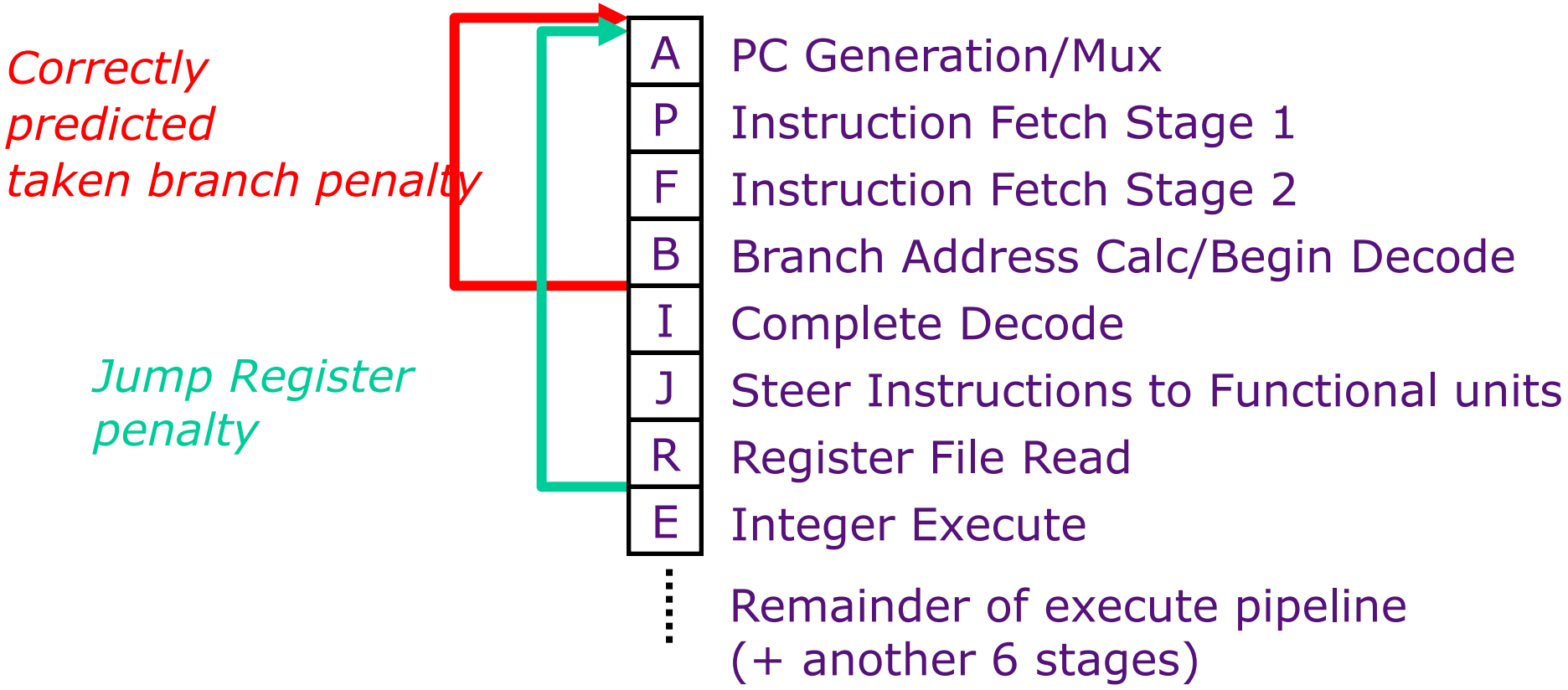
(2,2)预测器

Correlating predictor:(m,n)predictor

- An (m,n) predictor uses the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch.
  - For example, a (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch.

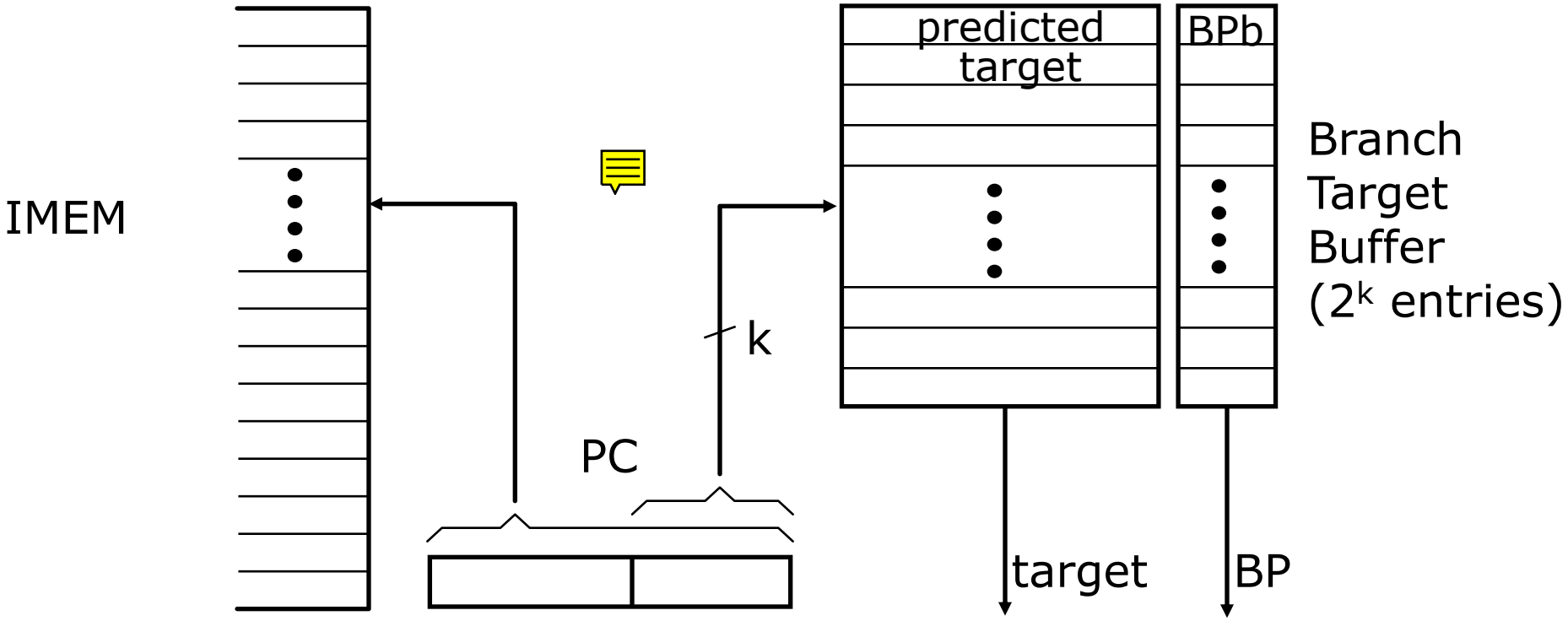
BHT局限性

仅能预测转移方向，因而，在确定转移目标之前，并不能从转移目标处开始取指令流。预测到not taken，很好，但若预测到taken呢，只用前边说的预测器就不行了，因为你还不知道taken的target地址



UltraSPARC-III fetch pipeline

转移目标缓冲器
(Branch Target Buffer)



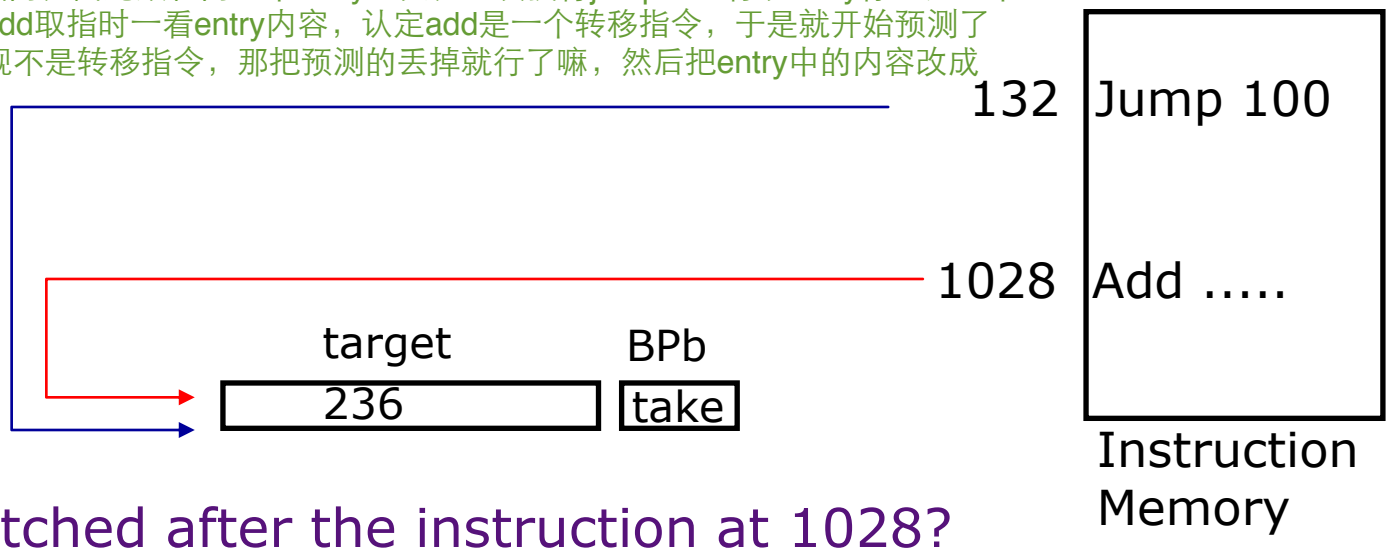
BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

地址冲突（Address Collisions）

jump和add的低位相同，因此放在同一个entry里头，上次执行jump已经将该entry标记为一个转移指令了，因此add取指时一看entry内容，认定add是一个转移指令，于是就开始预测了没关系，一译码发现不是转移指令，那把预测的丢掉就行了嘛，然后把entry中的内容改成非转移指令就好了

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

BTB prediction = 236

Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

Is this a common occurrence?
Can we avoid these bubbles?

BTB仅对控制指令有效

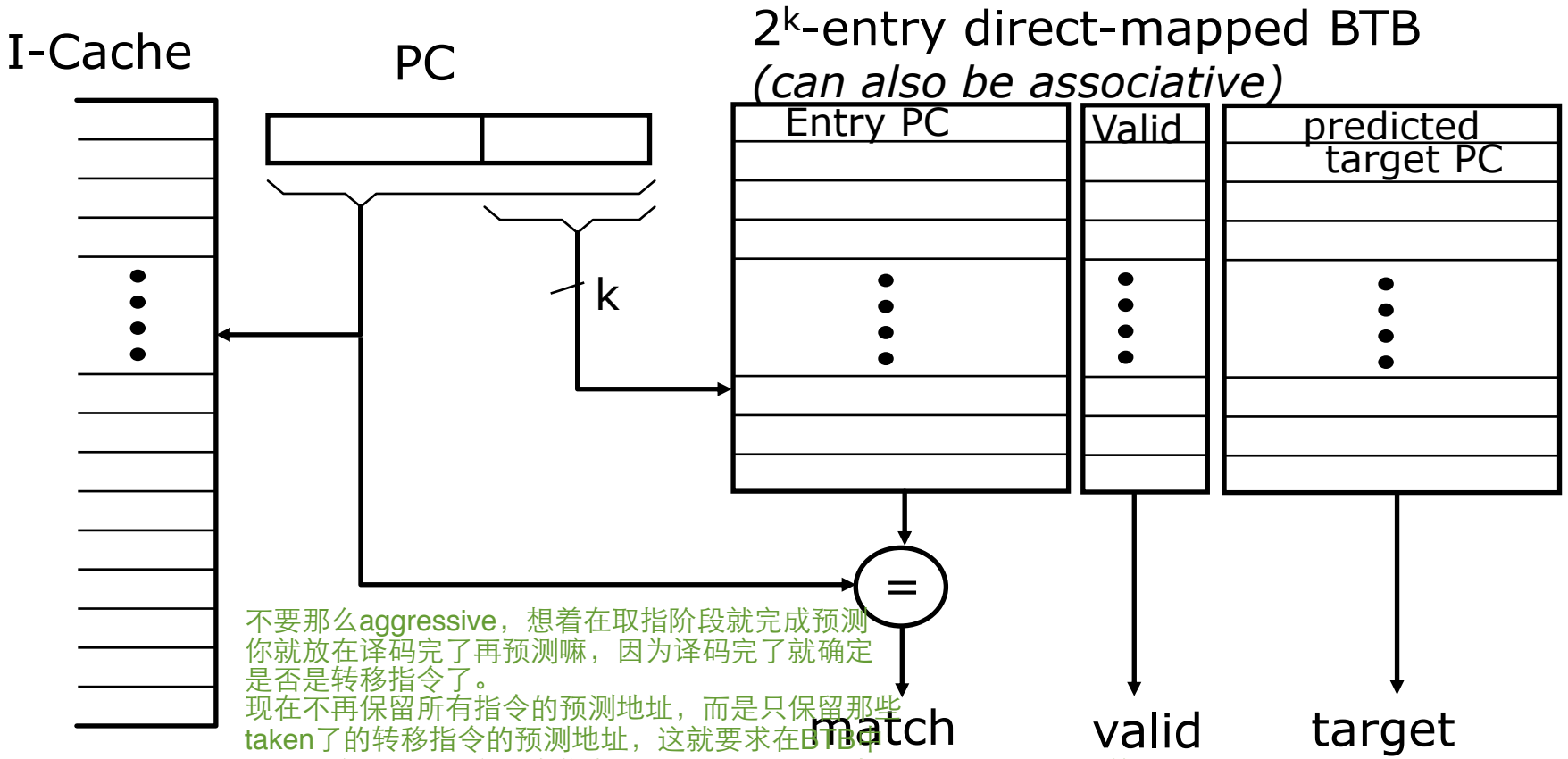
BTB仅包含针对转移指令和跳转指令的有用信息

⇒ 对其他指令，不能改变BTB内部的状态

对所有其他指令的下一PC 都是 PC+4！

如何在指令译码之前，就达到上述效果?

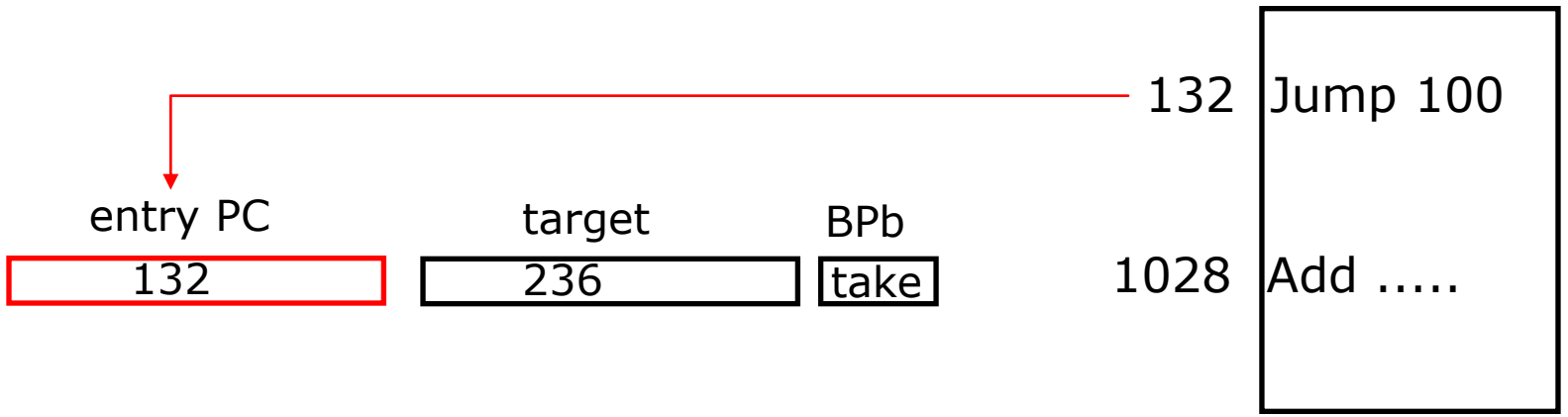
Branch Target Buffer (BTB)



不要那么aggressive，想着在取指阶段就完成预测
你就放在译码完了再预测嘛，因为译码完了就确定
是否是转移指令了。
现在不再保留所有指令的预测地址，而是只保留那些
taken了的转移指令的预测地址，这就要求在BTB中
引入一个entry PC字段来作为tag，而且可能需要相联存储器以方便查询

- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

在译码前查询BTB



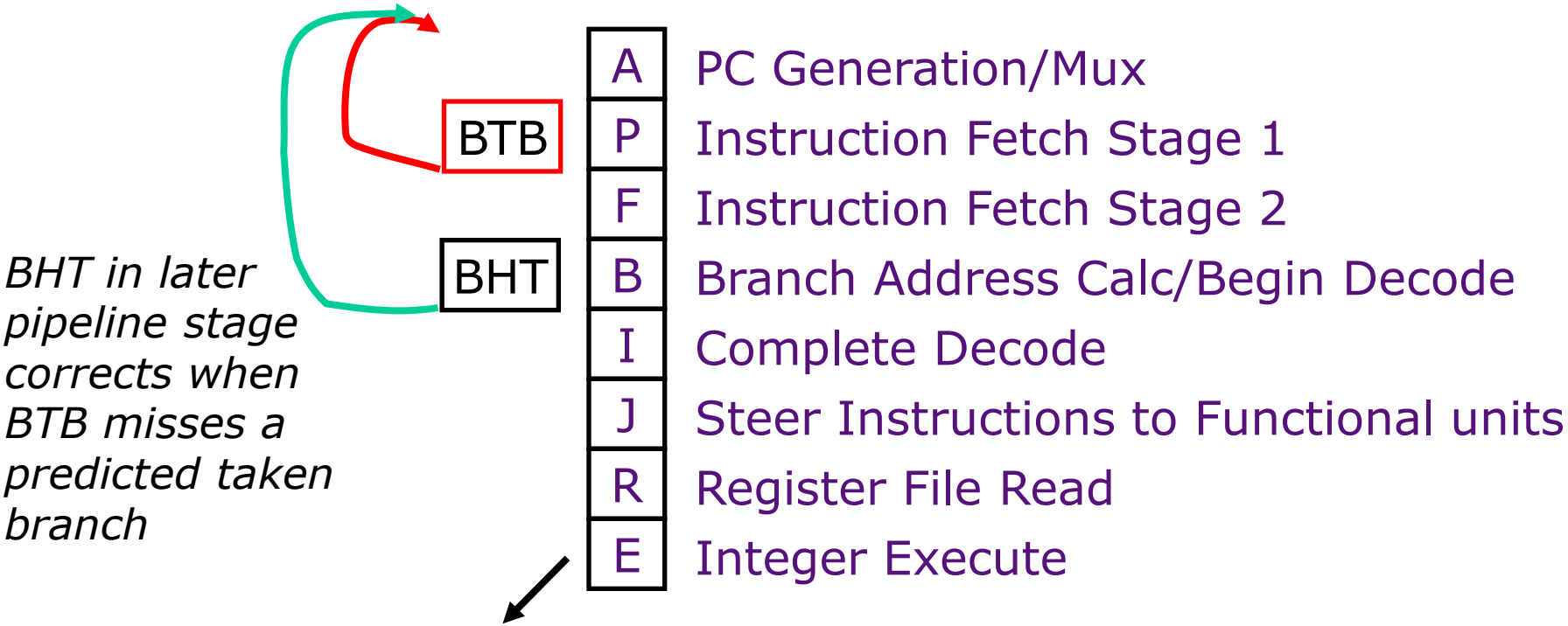
现在不是每个PC地址都有一个entry来存它的target buffer了，而是要从所有entry里头匹配1028，但1028，但buffer里头根本没有存入一个标记为1028的PC地址，因此1028match失败，就直接不预测target，而是PC+4
注意到这一步的时候，根本都还没有译码呢，还不知道1028是不是一个跳转指令，只是知道PC的值而已，就可以做这么多事情

- The match for PC=1028 fails and 1028+4 is fetched
  ⇒ *eliminates false predictions after ALU instructions*

- BTB contains entries only for control transfer instructions
  ⇒ *more room to store branch targets*

合并BTB和BHT

- 相对BHT而言，BTB的表项的实现成本更高，但是可以在流水线较早的时候就对取指流进行重定向，并能够加速间接转移(JR)
- BHT可以包含更多的表项，并更加准确



Comparison of 2-bit predictors.

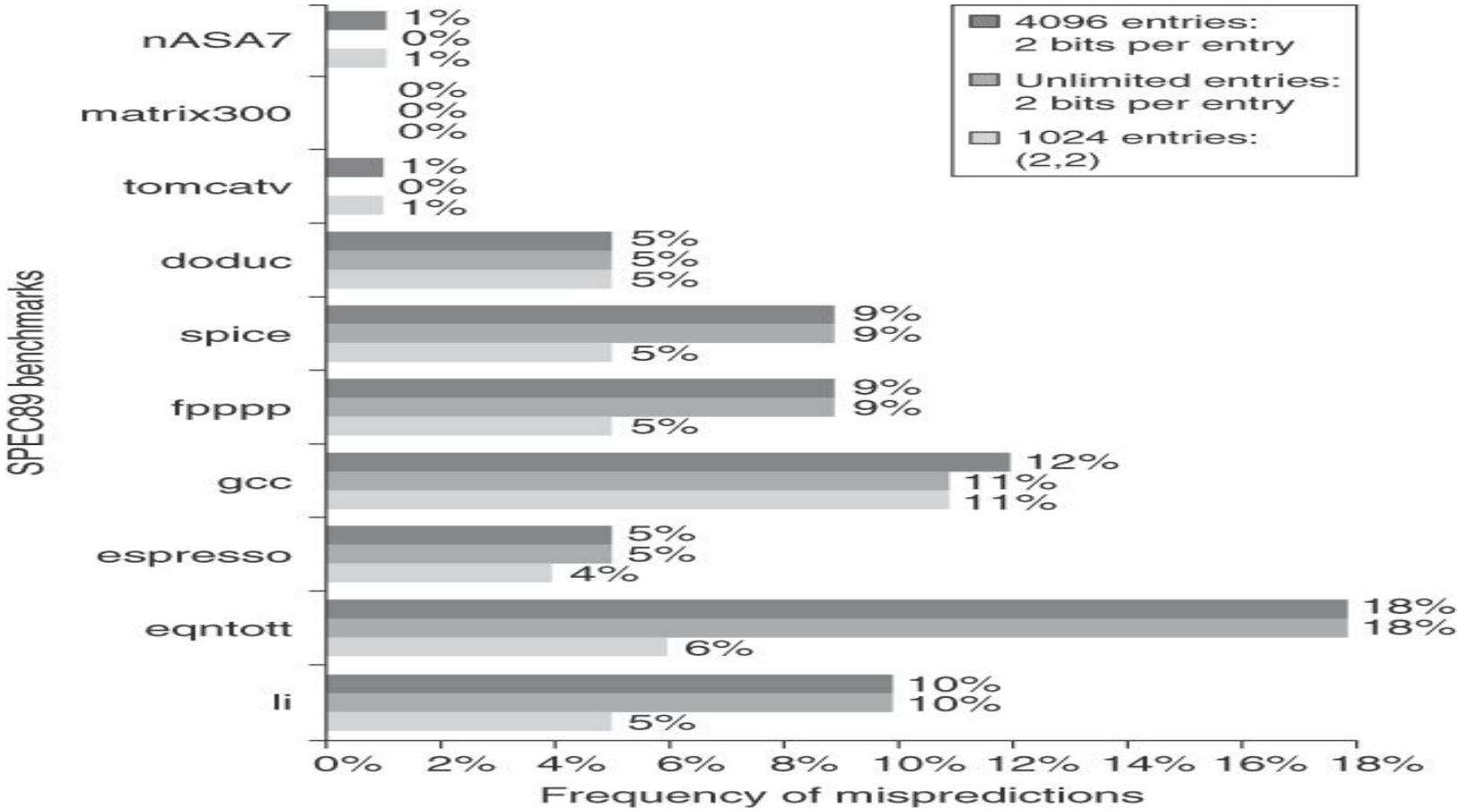
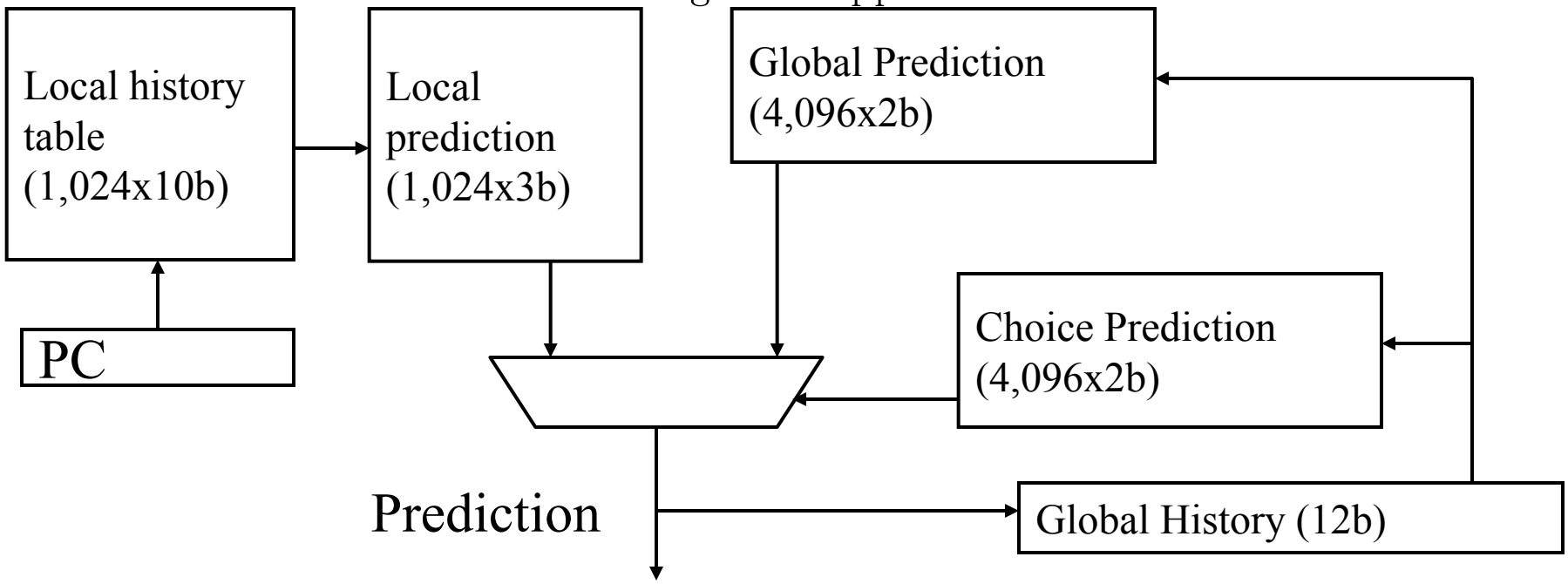


Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

Tournament Branch Predictor (Alpha 21264)

- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications

tournament branch predictor就是结合全局预测和局部预测，局部预测器还可以搞成多级的。根据学习，不同的转移指令可以选择对自己最好的预测器，局部或全局。



Misprediction rate for 3 predictors

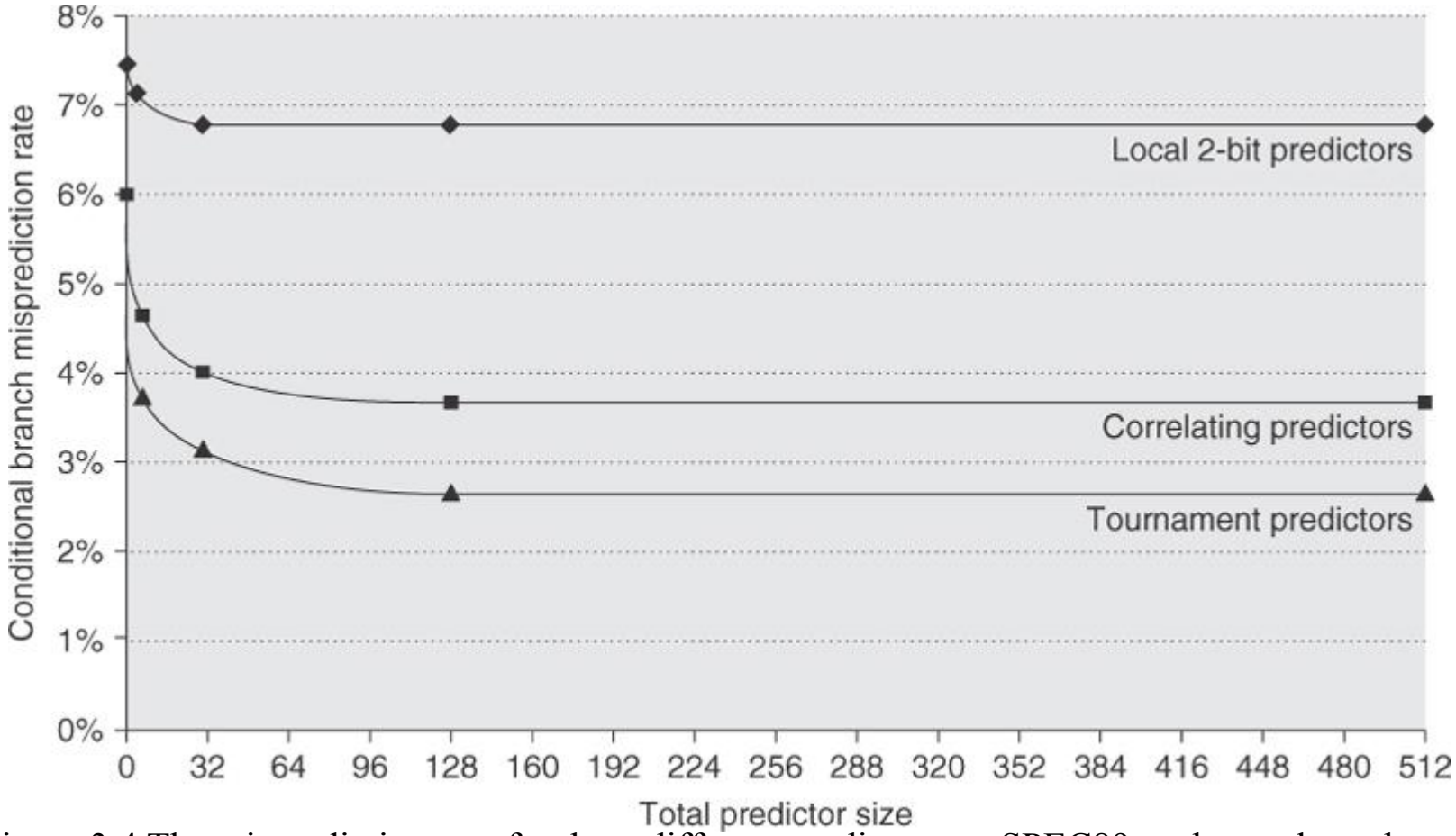


Figure 3.4 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

跳转寄存器(JR)的使用

- 切换状态（jump to address of matching case）

  BTB works well if same case used repeatedly

之前是泛泛的对所有转移的概论
下面说具体的。比如面向对象语言
比如虚拟机，就会大量使用间接跳转

- 动态过程调用（jump to run-time function address）

  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- 子程序返回（jump to return address）

  BTB works well if usually return to the same place

  ⇒ *Often one function called from many distinct call sites!*

对于上述情况，BTB都可以很好工作吗?

子程序返回栈（Subroutine Return Stack）

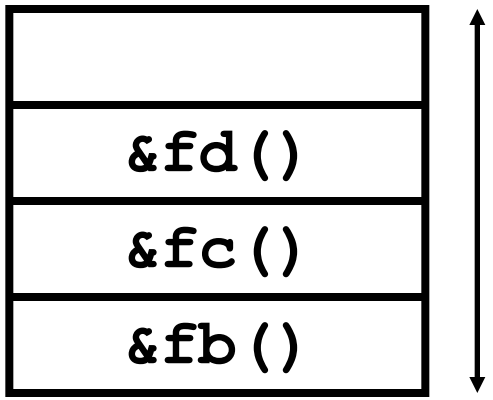
专设一个小的结构来加速针对子程序返回的JR处理，通常比BTBs会更加准确.

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

这是一个专门的结构来优化return call的预测

Push call address when function call executed

Pop return address when subroutine return decoded



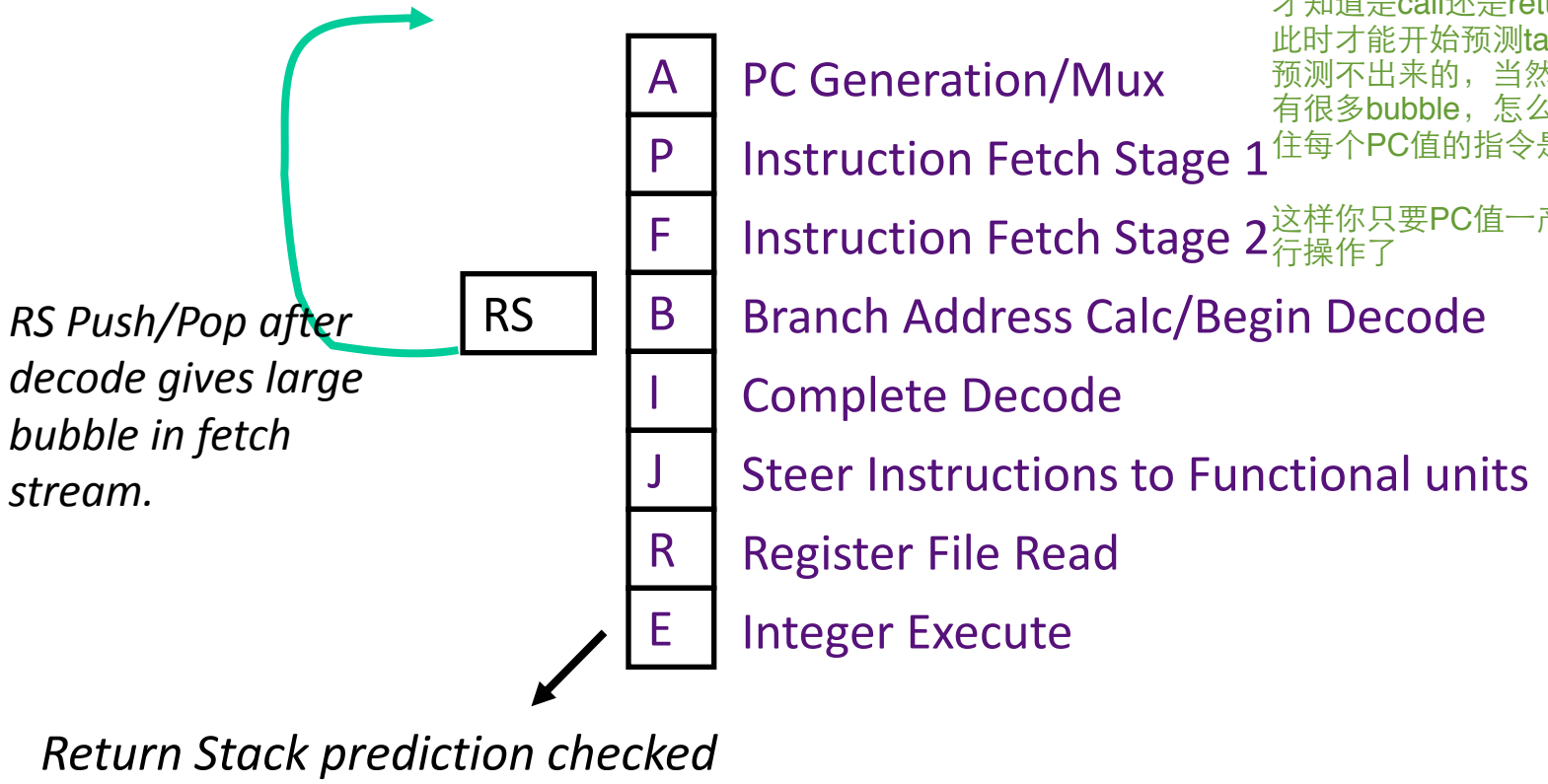
k entries (typically k=8-16)

Return Stack in Pipeline

1. How to use return stack (RS) in deep fetch pipeline?
2. Only know if subroutine call/return at decode

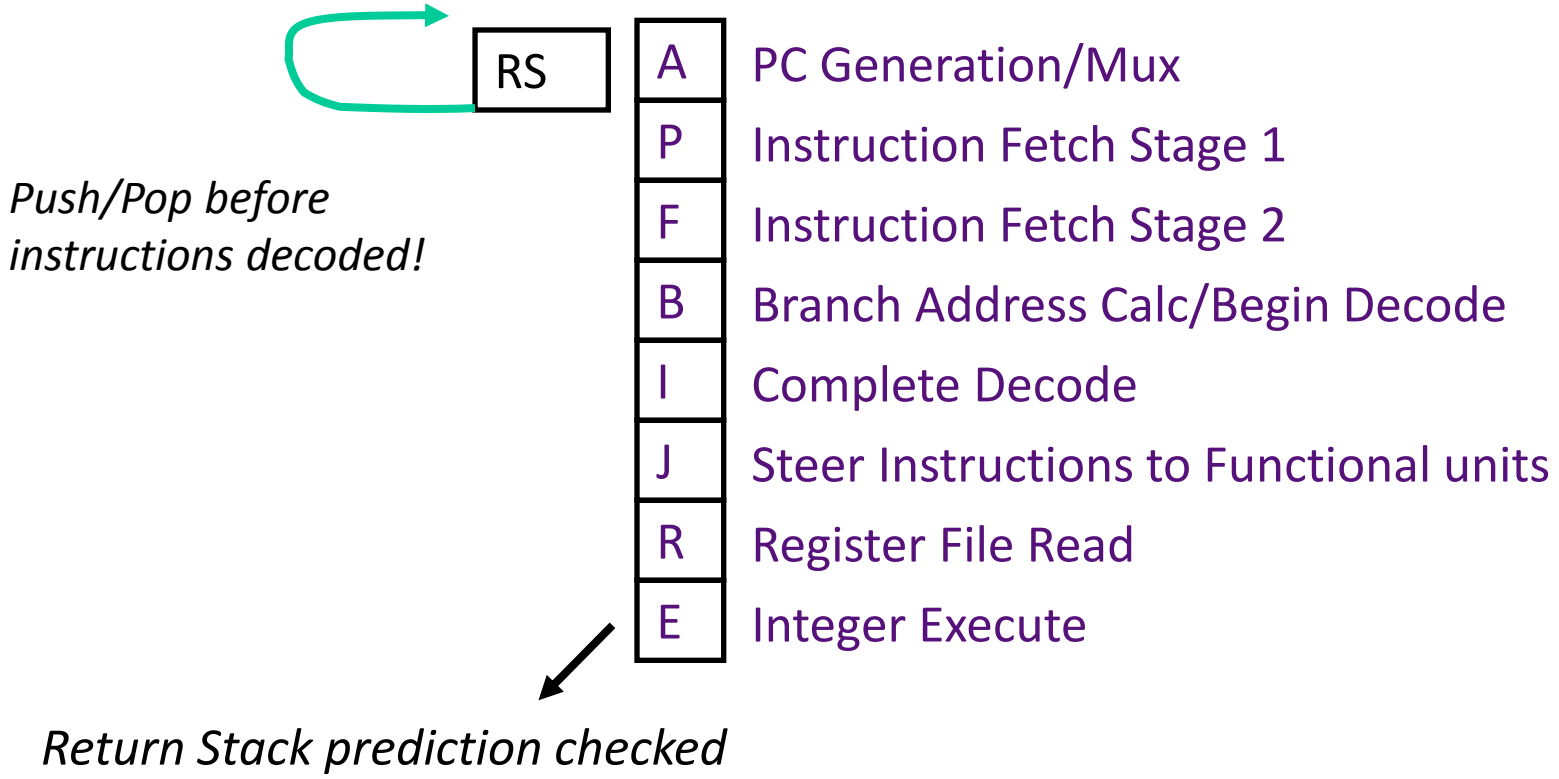
就是说在深度流水线中，只有当decode之后才知道是call还是return还是其它指令，直到此时才能开始预测target，在此之前target是预测不出来的，当然也就没法取指，于是就有很多bubble，怎么解决这个问题呢？你记住每个PC值的指令是return还是call不就行了

这样你只要PC值一产生，立马就可以对RS进行操作了

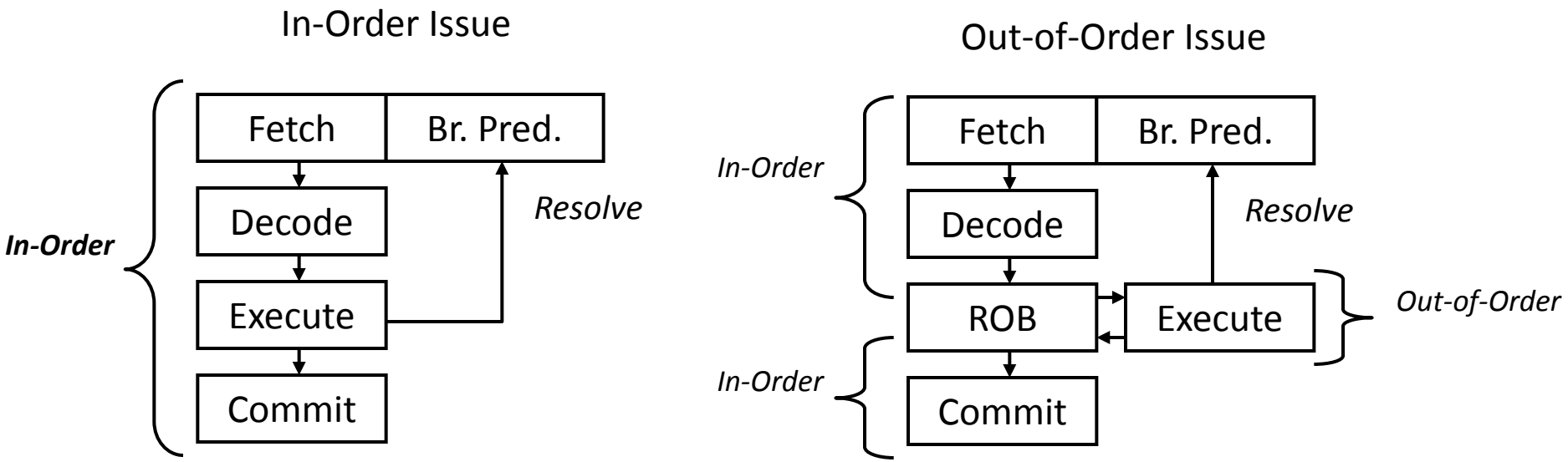


Return Stack in Pipeline

- Can remember whether PC is subroutine call/return using BTB-like structure
- Instead of target-PC, just store push/pop bit



In-Order vs. Out-of-Order Branch Prediction



第二条还没执行的时候，上一条已经确定结果了，推测还没执行就知道对错了

- Speculative fetch but not speculative execution - branch resolves before later instructions complete
- Completed values held in bypass network until commit

- Speculative execution, with branches resolved after later instructions complete
- Completed values held in rename registers in ROB or unified physical register file until commit

- Both styles of machine can use same branch predictors in front-end fetch pipeline, and both can execute multiple instructions per cycle
- Common to have 10-30 pipeline stages in either style of design

错误预测的恢复

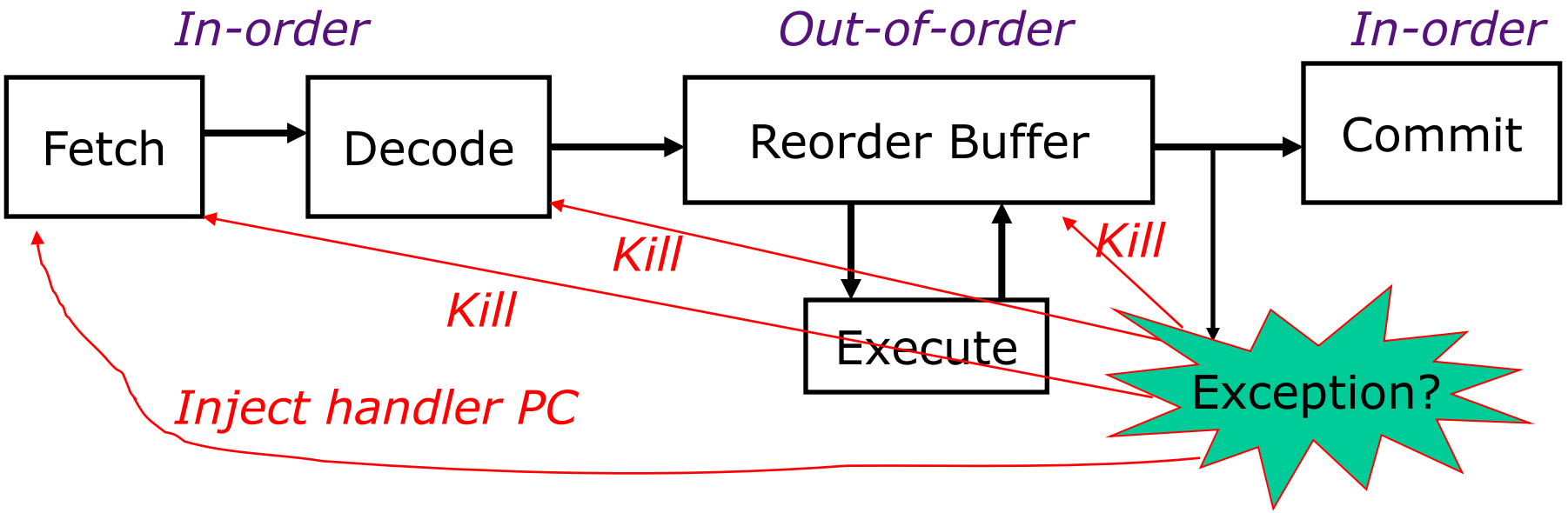
按序执行机器：

- 假设在转移解决之前，没有该转移之后发射的指令会回写结果（write-back）
- 将错误预测转移之后的所有指令都删除

乱序执行?

- 在转移解决之前，转移之后的多条指令（按串行程序序）可能均已完成

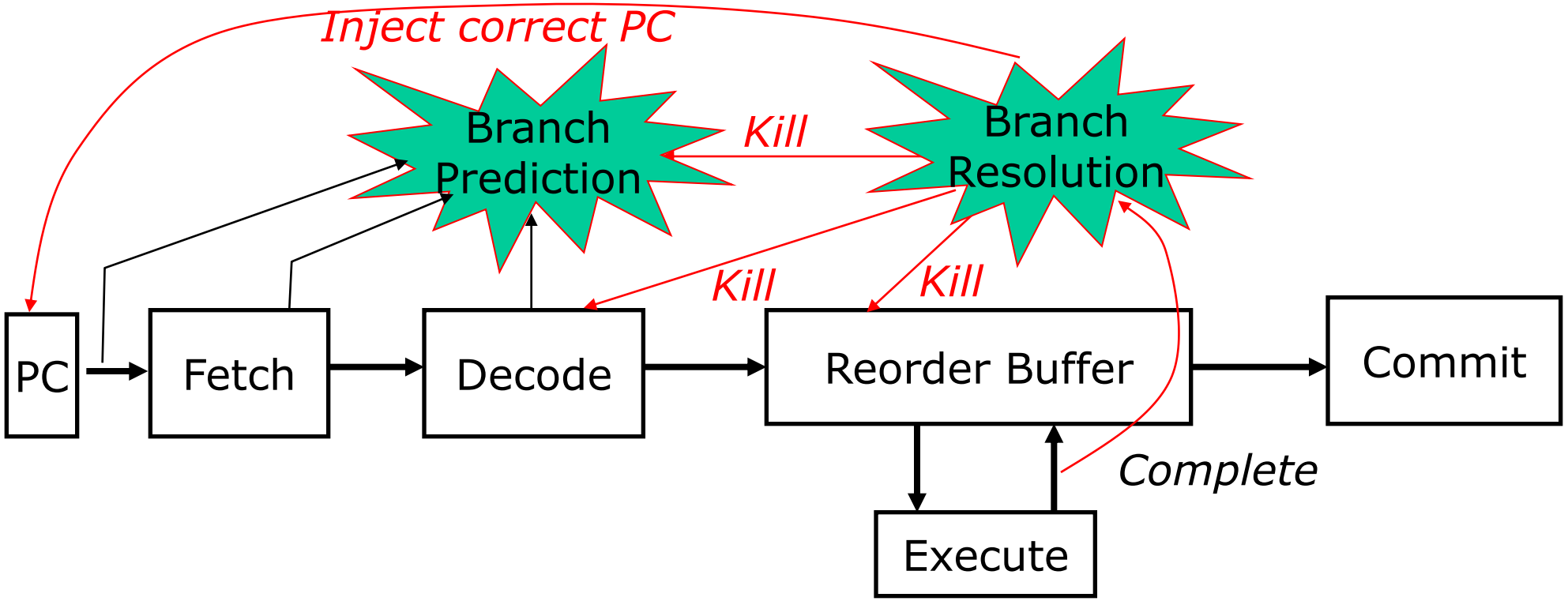
支持精确中断的按序提交



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order

Temporary storage needed in ROB to hold results before commit

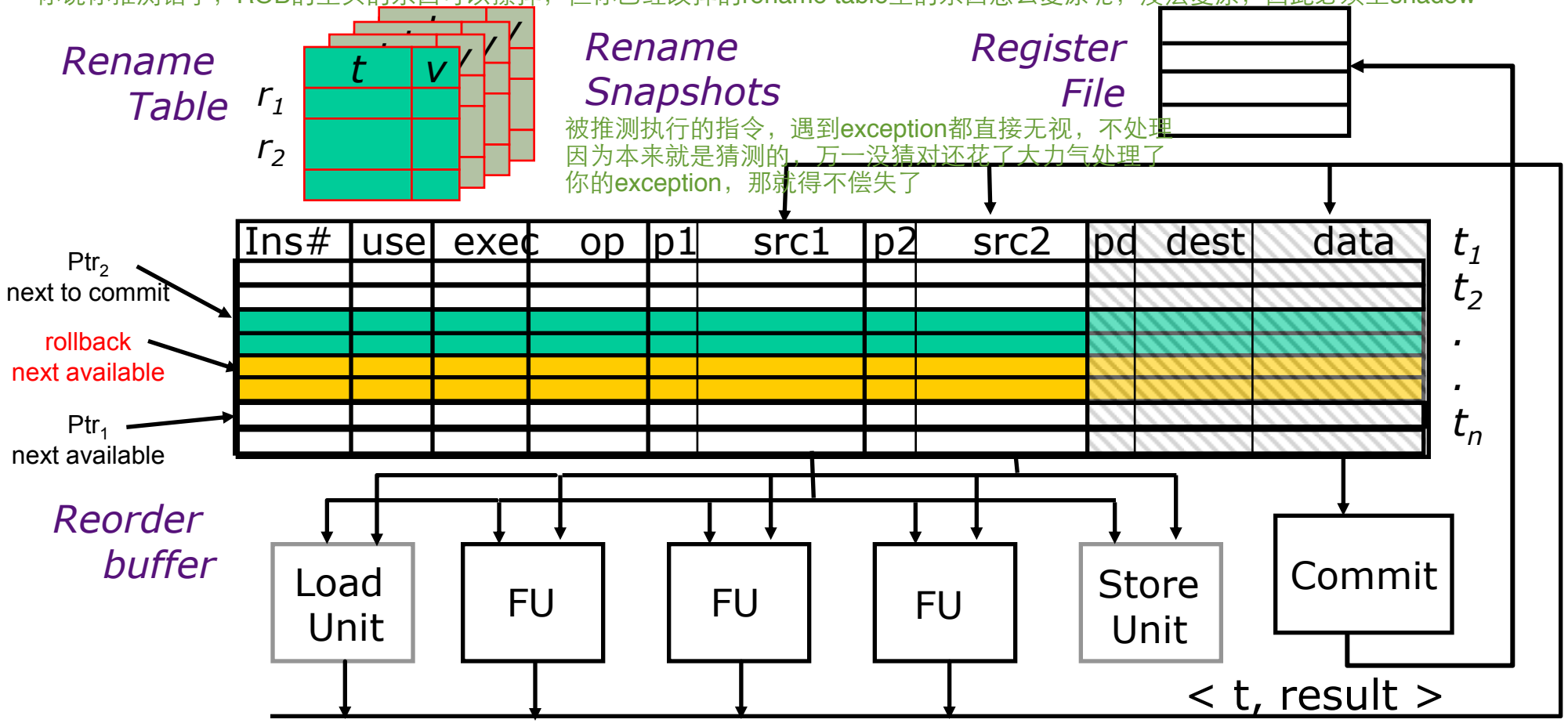
流水线中的转移错误预测



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch

Recovering ROB/Renaming Table

你一走不同的路径，就得有不同的renaming table。每次推测都把推测之前的rename table快速生出一个shadow，你在shadow上去推测
你说你推测错了，ROB的里头的东西可以擦掉，但你已经改掉的rename table里的东西怎么复原呢，没法复原，因此必须上shadow



Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted

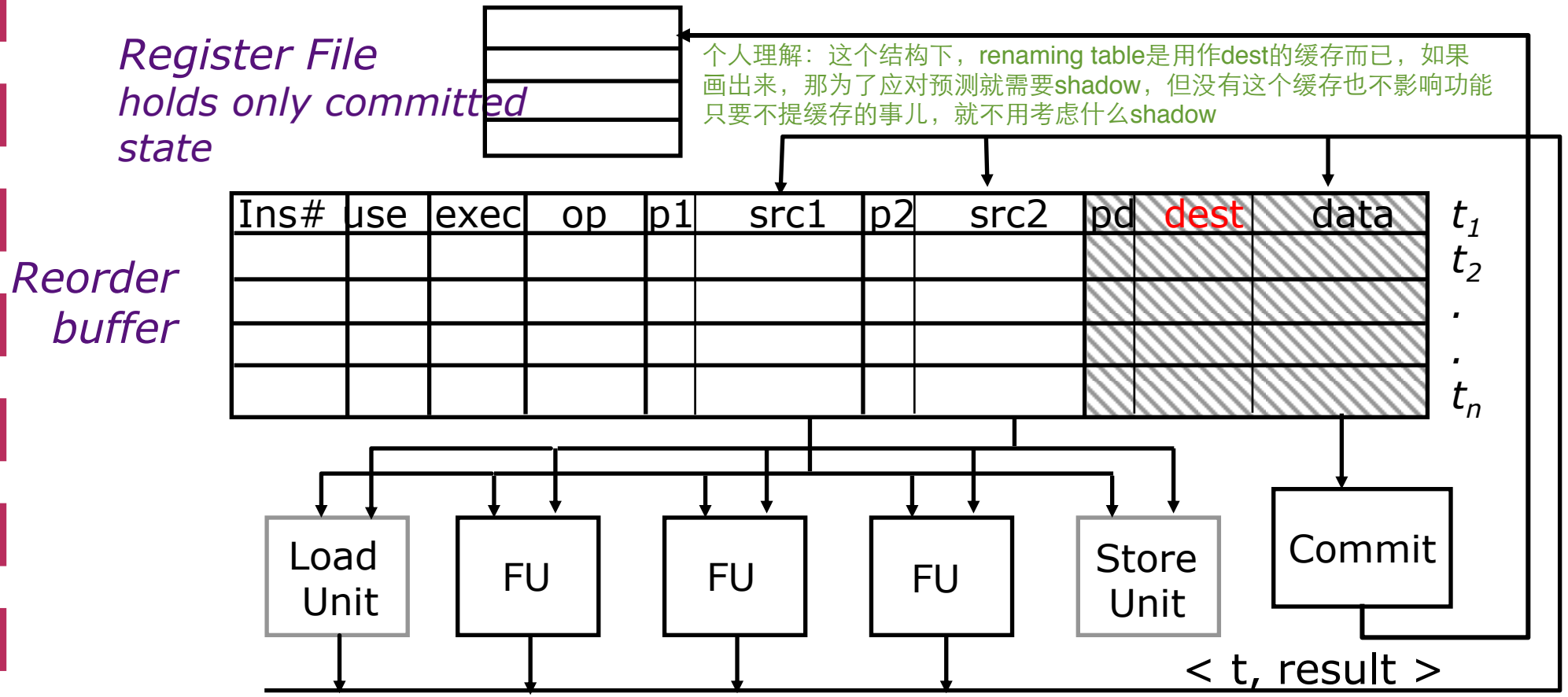
双向推测执行

与转移预测不同，还可以对转移个两条可能的方向同时进行推测执行

- 所需的资源数与并发推测执行的指令流数目成正比
- 当同时对一条转移的两条可能指令流进行推测执行时，只有一半的资源真正用于了有用工作
- 基于转移预测的推测执行比对转移的所有方向都进行推测执行需要更少的资源

当转移预测率很高时，将所有的资源都用于预测的方向是效率很高（cost effective）的方案

"Data in ROB" Design
(HP PA8000, Pentium Pro, Core2Duo)

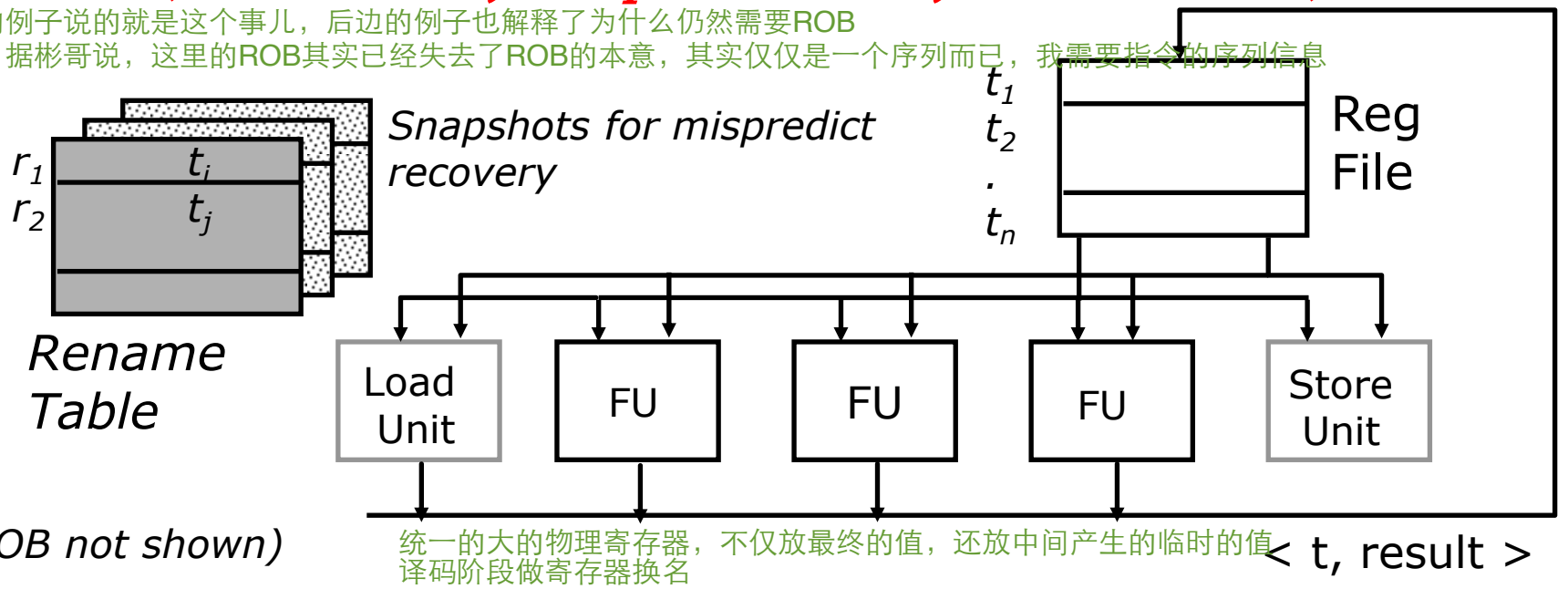


- On dispatch into ROB, ready sources can be in regfile or in ROB dest (copied into src1/src2 if ready before dispatch)
- On completion, write to dest field and broadcast to src fields.
- On issue, read from ROB src fields

Unified Physical Register File

(MIPS R10K, Alpha 21264, Pentium 4)

后边的例子说的就是这个事儿，后边的例子也解释了为什么仍然需要ROB
当然，据彬哥说，这里的ROB其实已经失去了ROB的本意，其实仅仅是一个序列而已，我需要指令的序列信息



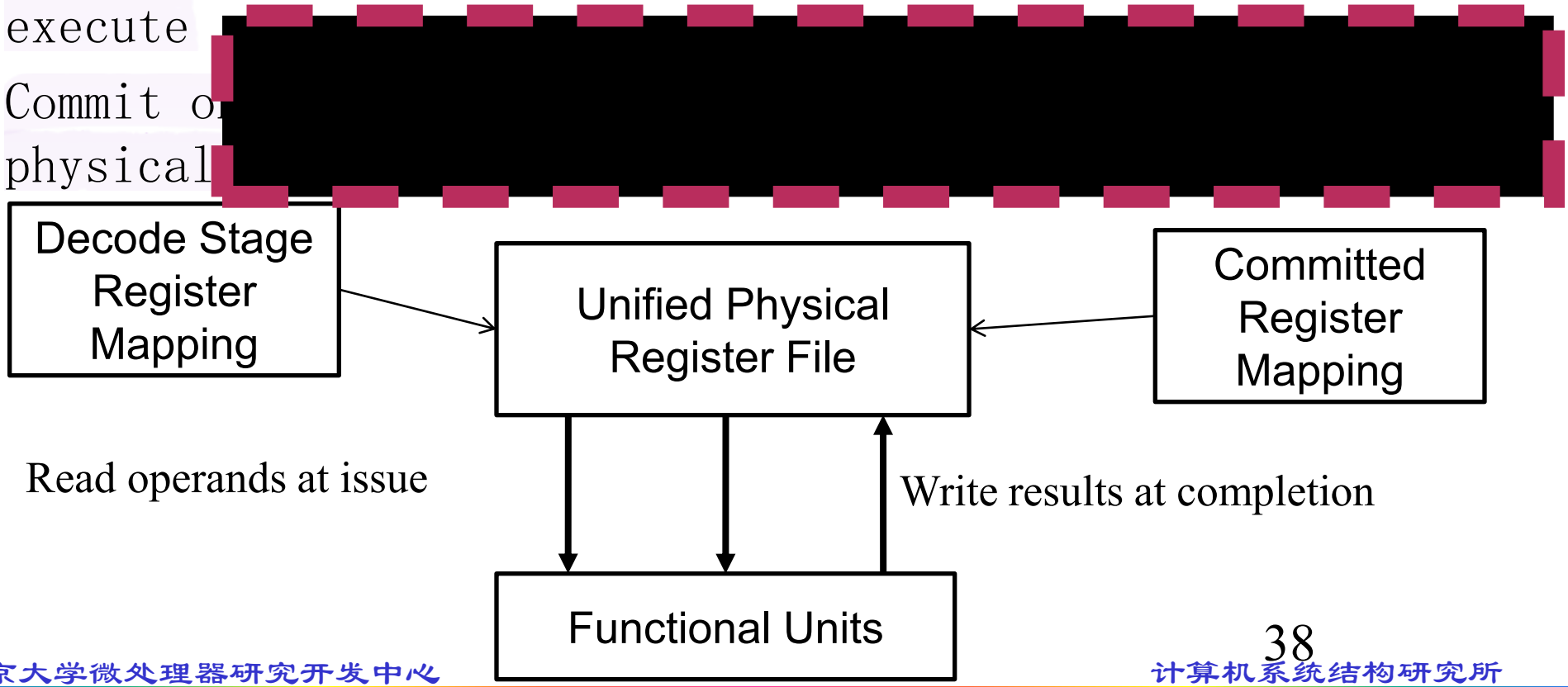
统一的大的物理寄存器，不仅放最终的值，还放中间产生的临时的值
译码阶段做寄存器换名

- One regfile for both *committed* and *speculative* values (no data in ROB)
- During decode, instruction result allocated new physical register, source regs translated to physical regs through rename table
- Instruction reads data from regfile at start of execute (not in decode)
- Write-back updates reg. busy bits on instructions in ROB (assoc. search)
- Snapshots of rename table taken at every branch to recover mispredicts
- On exception, renaming undone in reverse order of issue *(MIPS R10000)*

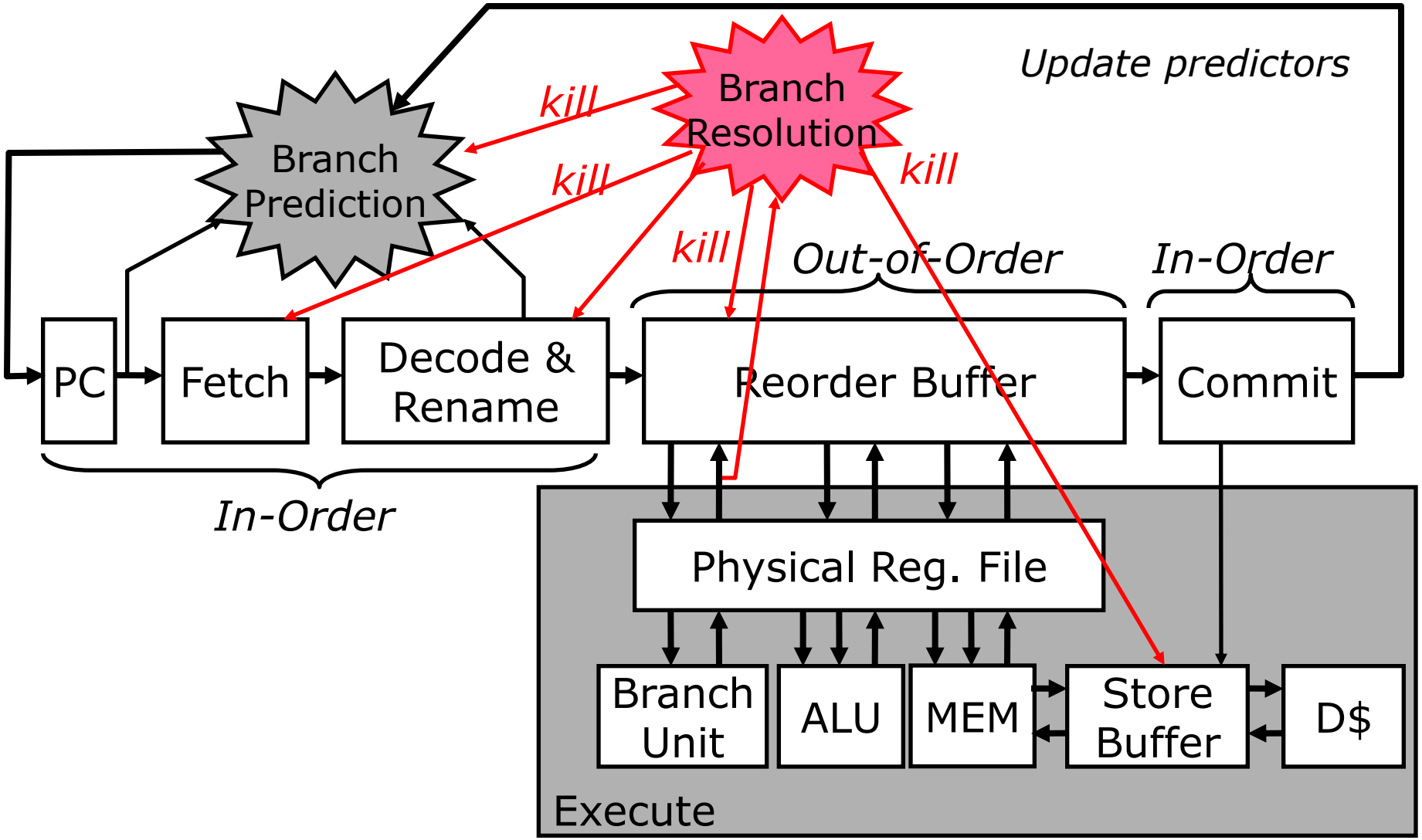
Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy Bridge)

- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit o
physical



Pipeline Design with Physical Regfile

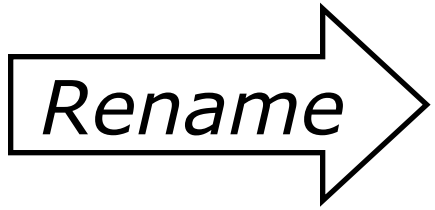


物理寄存器的生命期

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*

renaming之后，只要有足够的物理寄存器，就可以为这些换名之后多出来的寄存器分配物理寄存器
换名之后没有这些name dep了。程序员可见的寄存器不可太多，因为指令不能太长

```
ld r1, (r3)                ld P1, (Px)
add r3, r1, #4             add P2, P1, #4
sub r6, r7, r9             sub P3, Py, Pz
add r3, r3, r6             add P4, P2, P3
ld r6, (r1)                ld P5, (P1)
add r6, r6, r3             add P6, P5, P4
st r6, (r1)                st P6, (P1)
ld r6, (r11)               ld P7, (Pw)
```

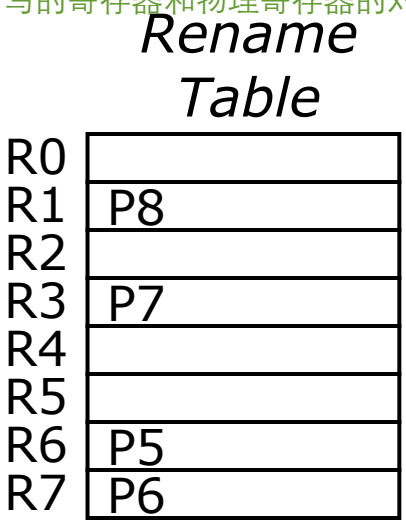


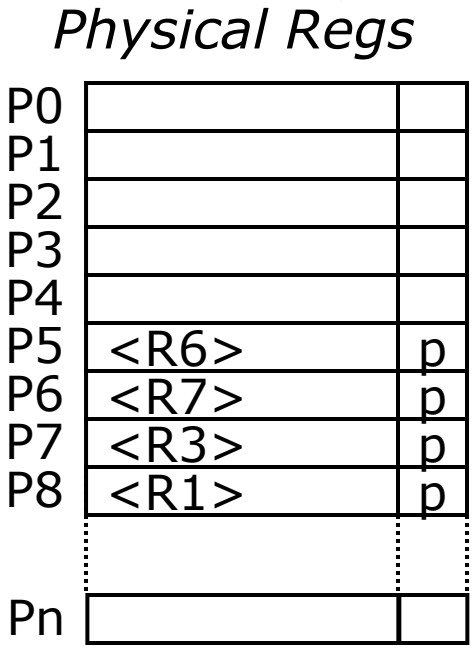
When can we reuse a physical register?

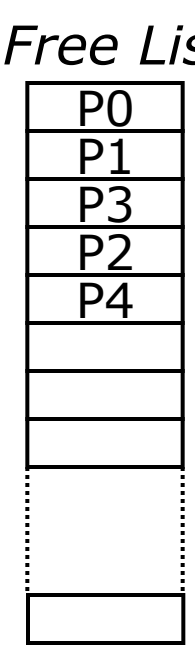
When next write of same architectural register commits

物理寄存器的管理

程序中写的寄存器和物理寄存器的对应关系







记录可用的物理寄存器，即目前未被占用的

```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

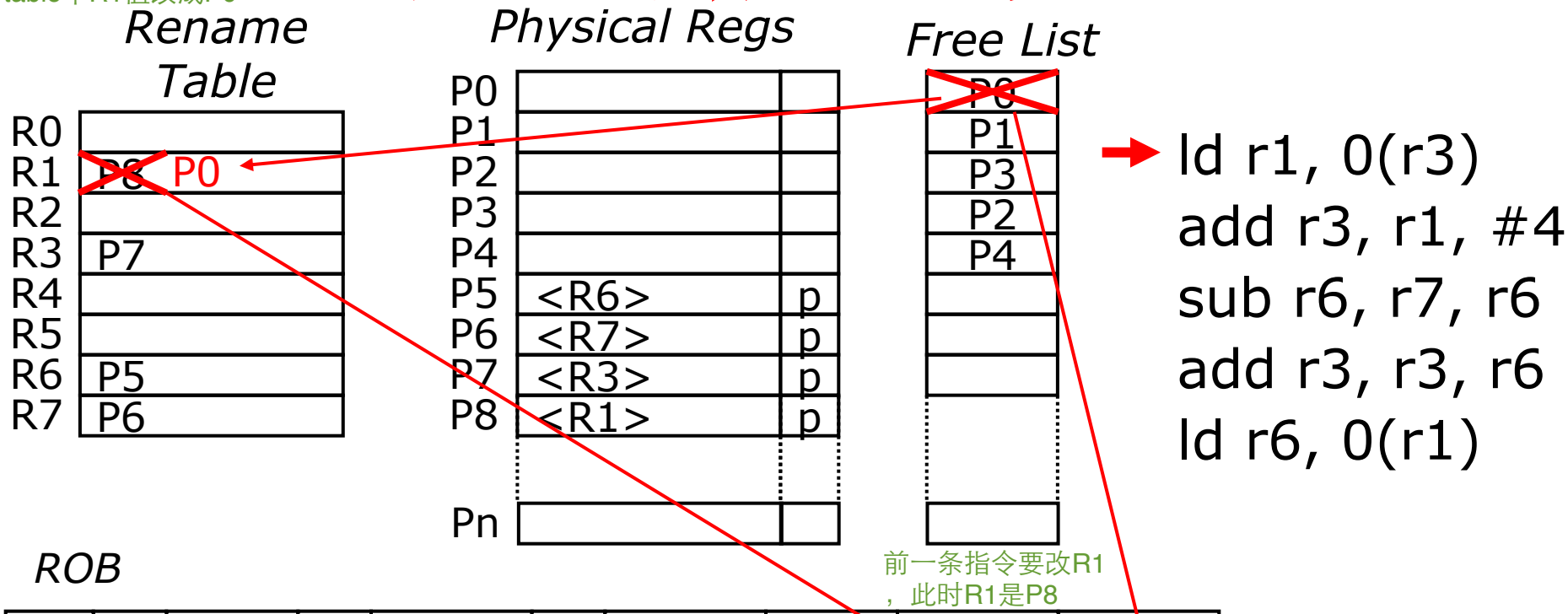
PRd:该指令将改的值对应的物理寄存器号

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

(LPRd requires third read port on Rename Table for each instruction)

为R1重新申请一个物理寄存器即P0，后续指令再要用R1就应该到P0中读，因此将renaming table中R1值改成P0

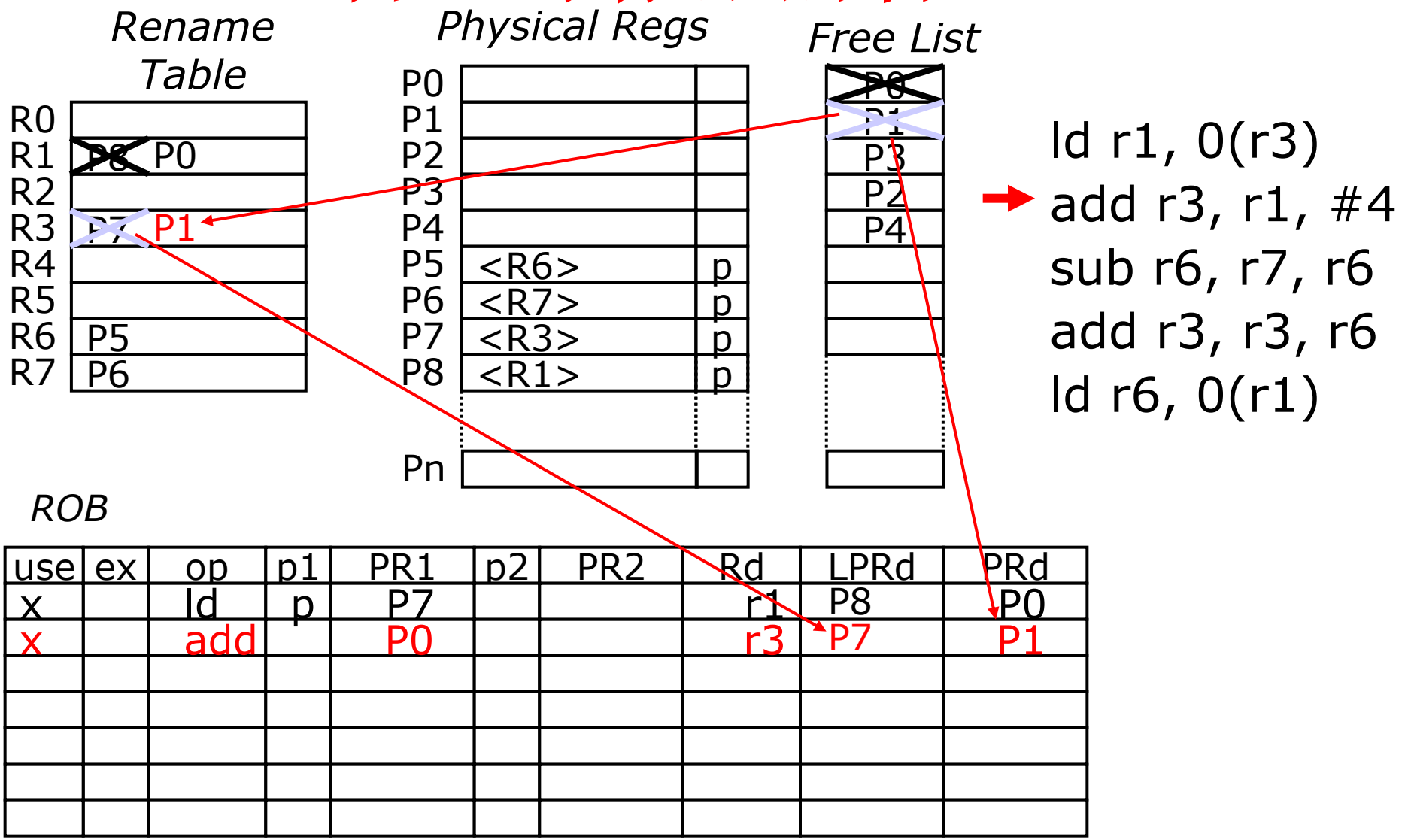
物理寄存器的管理



ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0

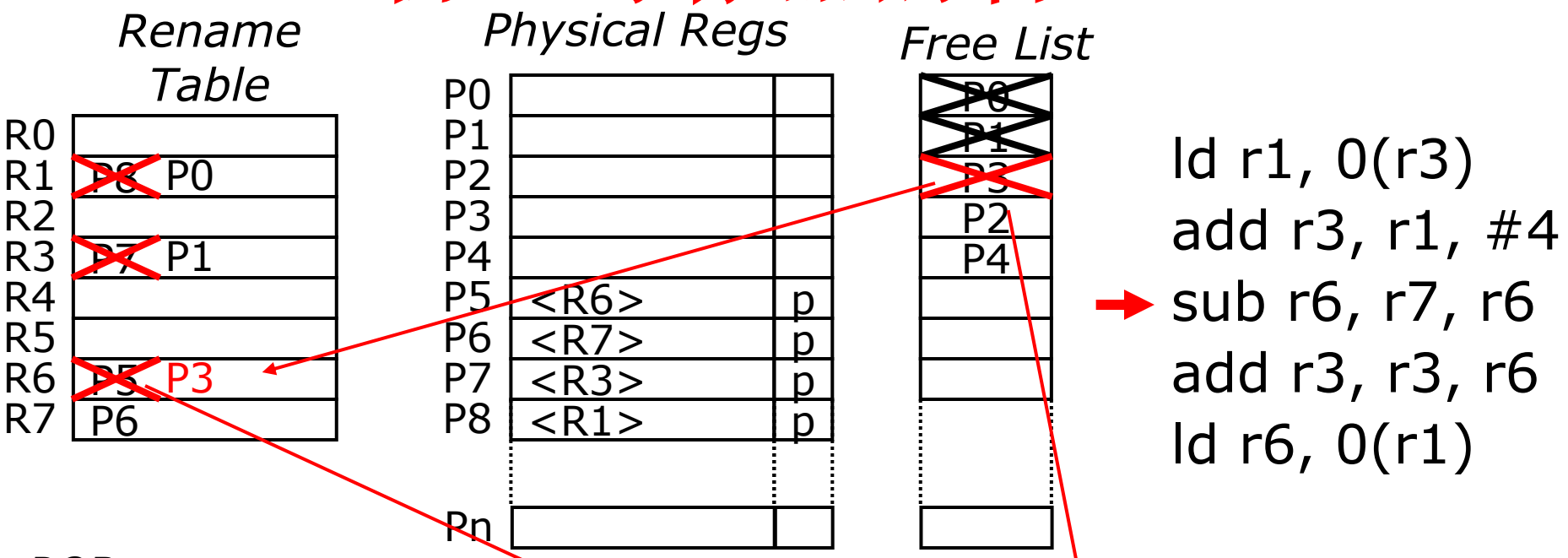
物理寄存器的管理



ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1

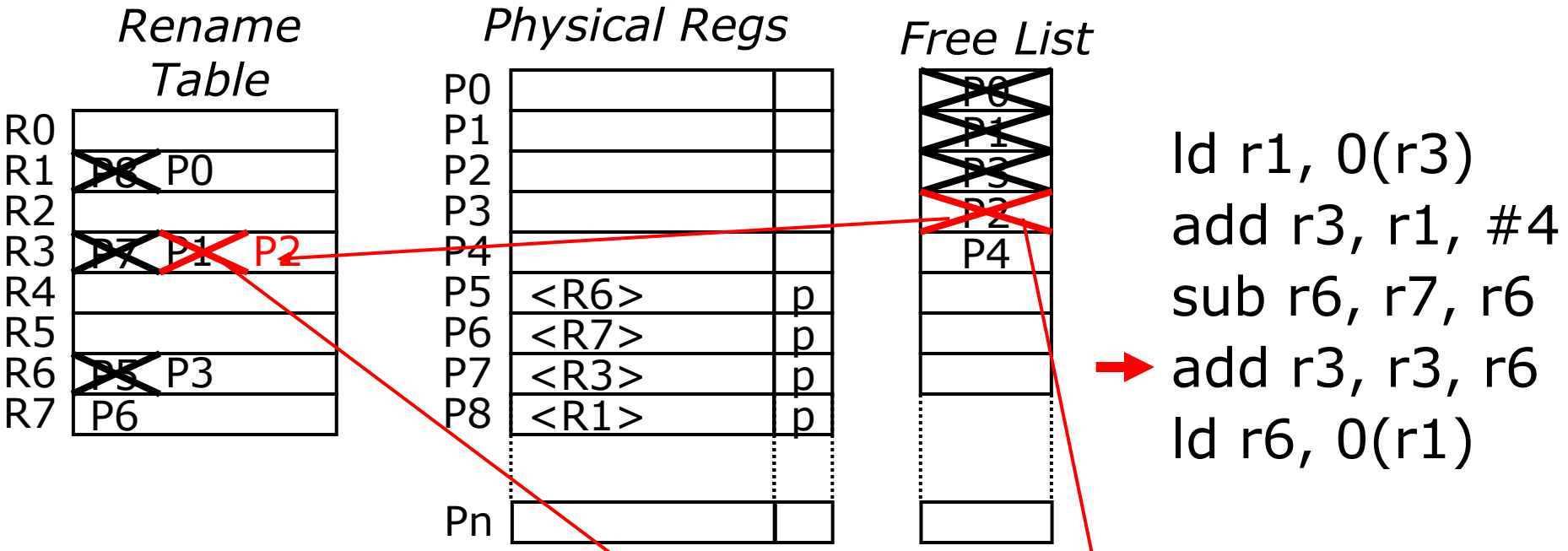
物理寄存器的管理



ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3

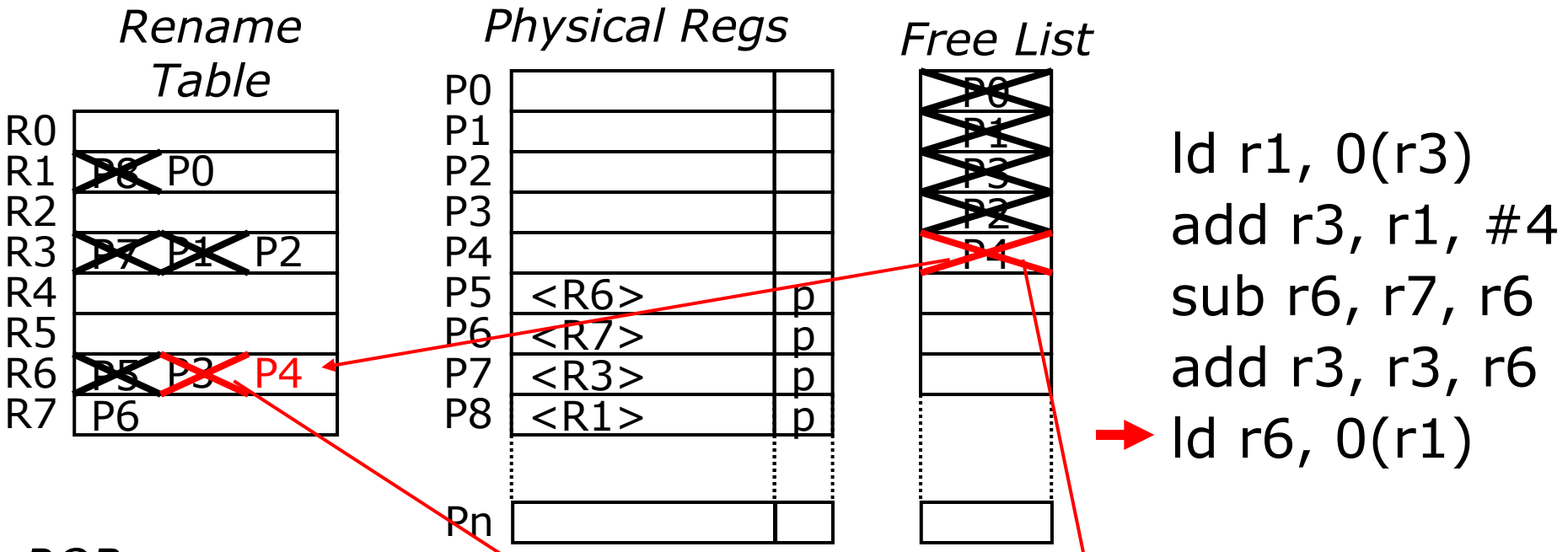
物理寄存器的管理



ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2

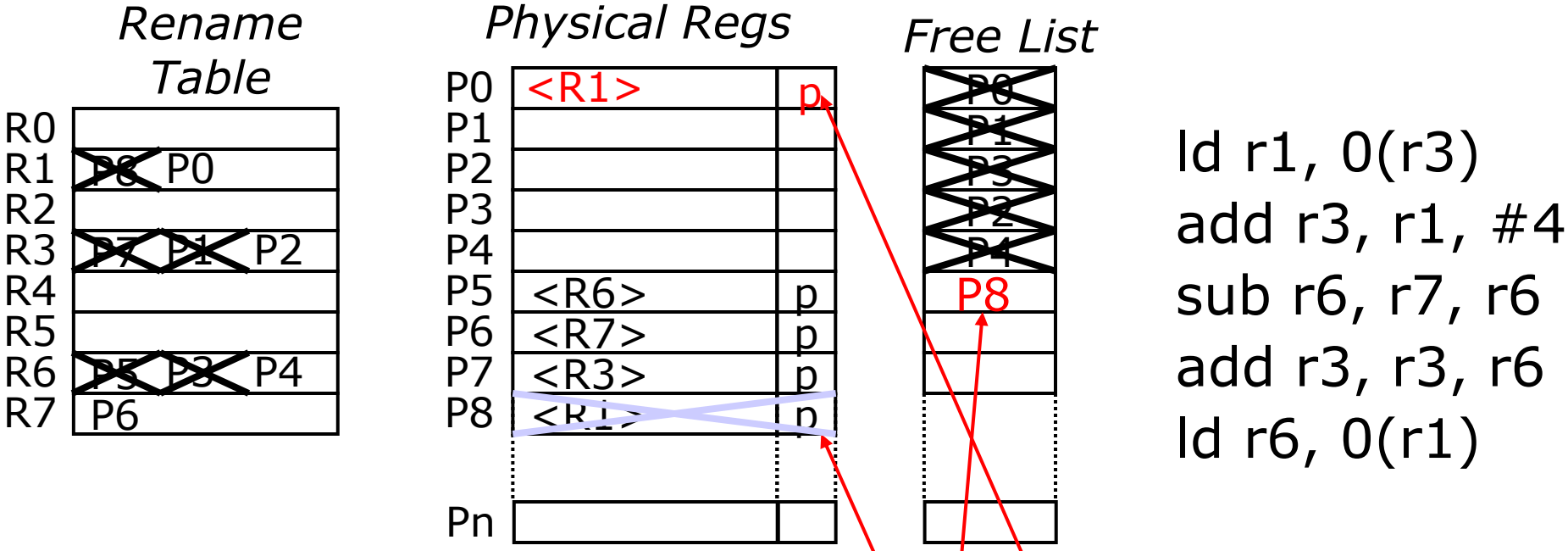
物理寄存器的管理



ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2
x		ld		P0			r6	P3	P4

物理寄存器的管理



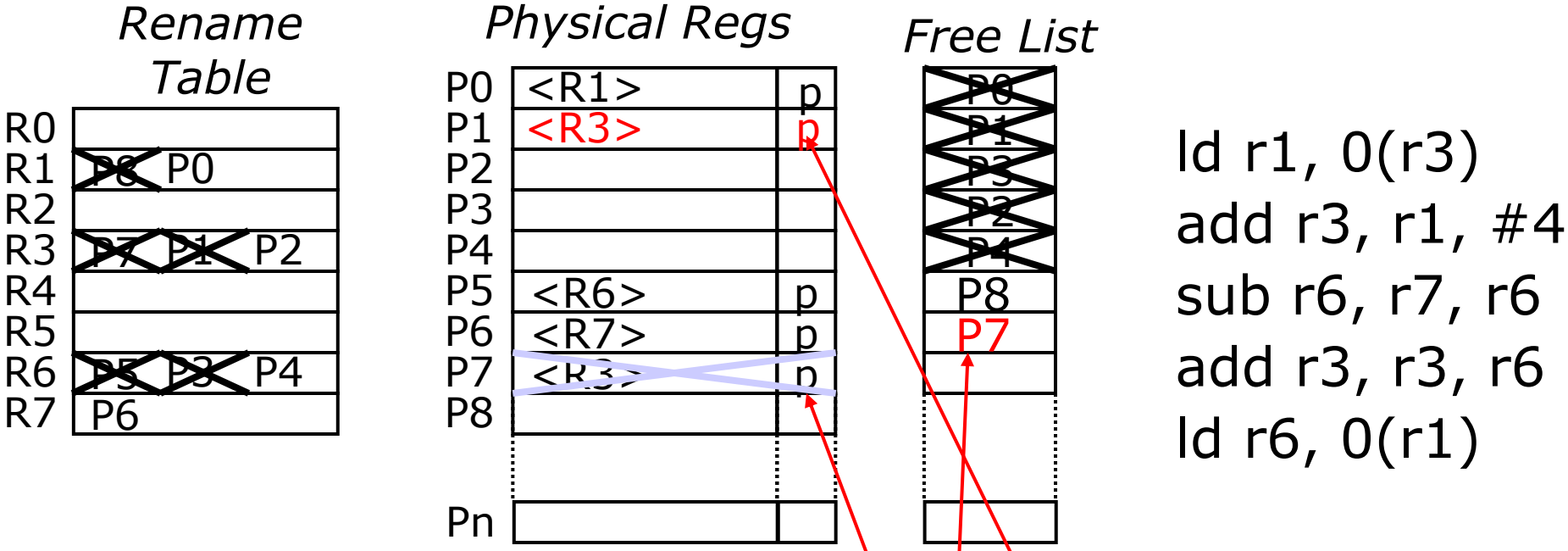

```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	**ld**	**p**	**P7**			**r1**	P8	**P0**
x		add	p	P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2
x		ld	p	P0			r6	P3	P4

Execute & Commit

物理寄存器的管理



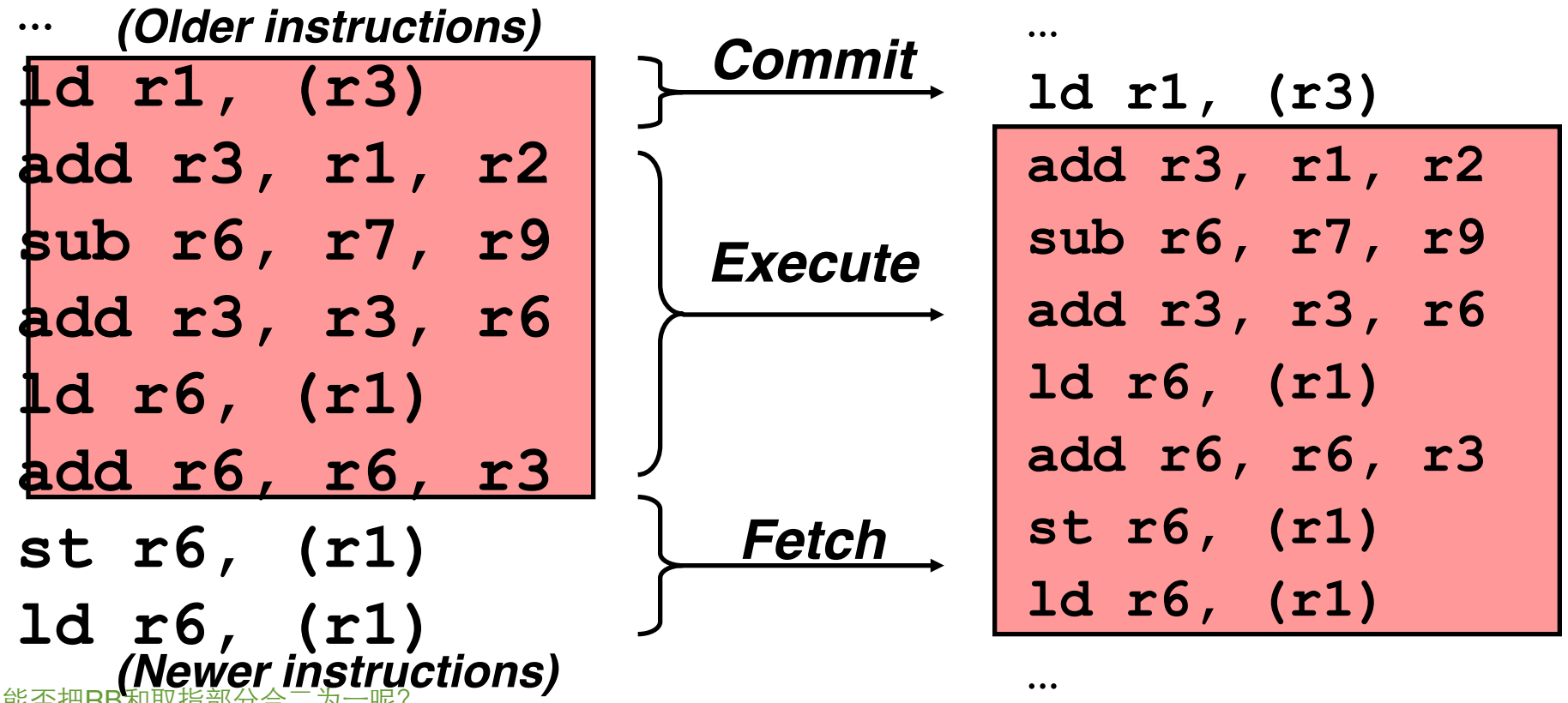

```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			r1	P8	P0
x	x	add	p	P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add	p	P1		P3	r3	P1	P2
x		ld	p	P0			r6	P3	P4

Execute & Commit

Reorder Buffer Holds Active Instruction Window

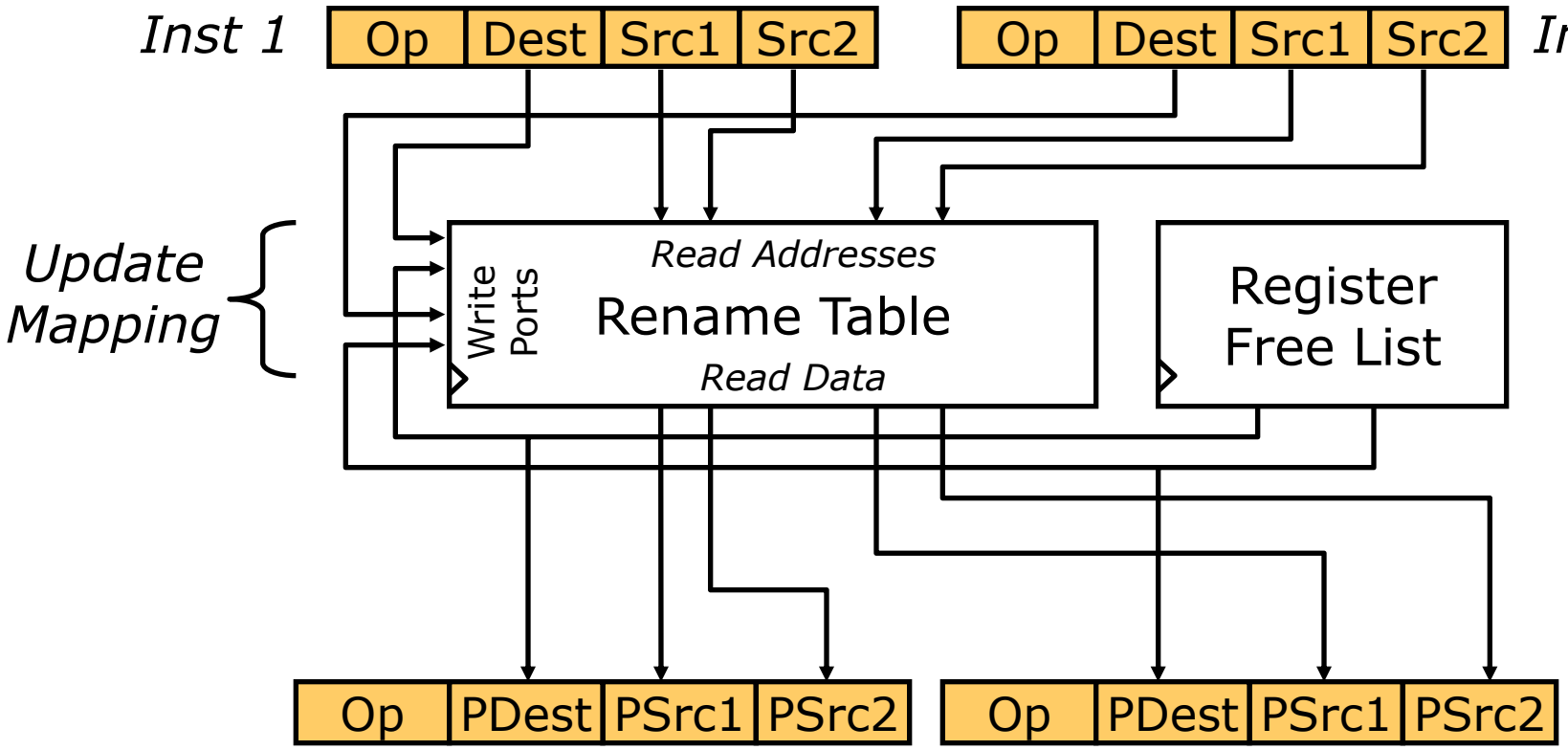


能否把RB和取指部分合二为一呢?
instruction window是取指完之后待处理的指令，这些指令只要包括一部分信息即可，后续通过译码继续得到信息
而RB中包含的是已译码后被扔出去，准备被处理以及正在被处理还没commit的指令，每个entry包含相当多的信息，它已经很大了
如果你要合二为一，你以为将其空间生出来一部分，实际上在RB里头引入了更多的域，而RB本是很复杂，使得RB的空间利用率稀释了

超标量处理器的寄存器重命名

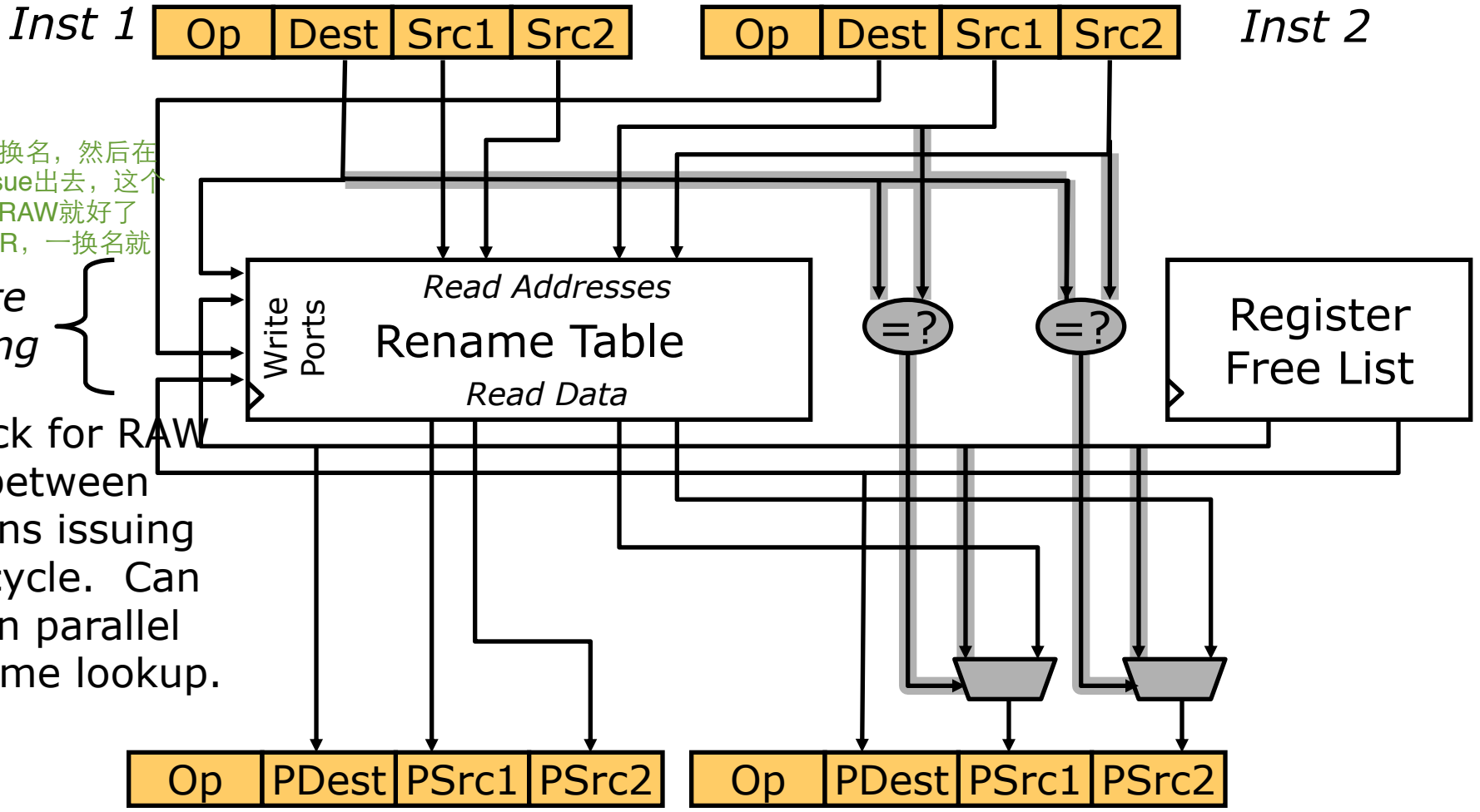
- 在译码阶段，指令被重新分配新的物理目标寄存器
- 源操作数被重命名为具有最新数据的物理寄存器
- 执行部件仅能看到物理寄存器号

希望一个周期同时译码多条指令，把它们同时issue出去
原来一个周期issue一条指令的情况下后续处理却未必一个周期处理完一条指令
有可能一个周期处理完多条指令，因此一瓶颈在于如何一个周期发射多条指令



Does this work?

超标量处理器的寄存器重命名



MIPS R10K renames 4 serially-RAW-dependent insts/cycle

存储器相关

我们前边费了很多事，终于用换名的方法去掉了伪数据相关，用推测式执行的方法尽量减少了控制相关的影响，剩下一个重要的问题就是如何让存取指令能够正确地工作，即如何解决存储相关。
存储相关也解决之后，superscalar还有啥重要问题呢？那就是现有的程序本身到底存在多少指令级的并行性？一般来说，科学计算程序比较

```
st r1, (r2)
ld r3, (r4)
```

习惯上，流水线超过6级就叫超流水 super pipeline

何时能够执行load指令?

load、store涉及程序正确性问题，你打乱顺序可能程序正确性都不能保证。
最笨的方法就是顺序一致性原则：所有的存储顺序要保证不能打乱

按序存储队列（In-Order Memory Queue）

- 按程序序执行所有的load和store操作

=> 在之前的所有存储和装入指令完成之前，Load和store指令不能离开ROB执行

- 希望对Load和Store指令进行推测执行，并与其他指令乱序执行

保守的Load乱序执行

```
st r1, (r2)
ld r3, (r4)
```

- 将Store指令的执行分解为两个阶段：地址计算、数据写入
- 如果地址已知，并可以确认r4 != r2，就可以在Store之前执行load指令
- 每个load地址都需要与所有之前未确认的STORE之前的地址进行比较*(可以使用部分保守比较，例如地址的低12位)*
- 如有之前的任何STORE指令有地址尚不确定，就不能执行load

(MIPS R10K, 16 entry address queue)

地址推测（Address Speculation）

```
st r1, (r2)
ld r3, (r4)
```

- 猜设 r4 != r2
- 在store地址未知情况先，执行load指令
- 需要按程序序保存所有完成但未提交的load/store地址
- 如果后续发现 r4==r2，碾压掉load及后续所有的指令
  => 如果地址猜测不准确，损失可能会很大!

Memory Dependence Prediction

(Alpha 21264)

```
st r1, (r2)
ld r3, (r4)
```

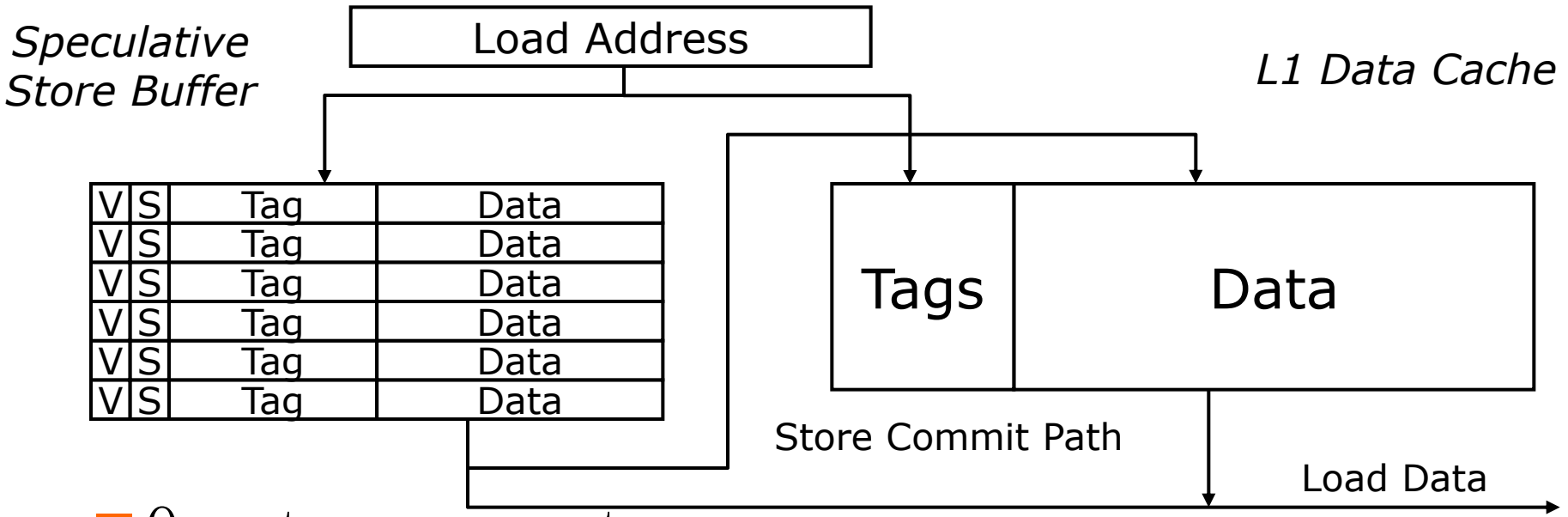
- Guess that r4 != r2 and execute load before store
- If later find r4==r2, squash load and all following instructions, but mark load instruction as *store-wait*
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear *store-wait* bits

推测式Loads / Stores

与寄存器的变更相同，在之前的所有指令都提交之后，store指令才能修改存储器

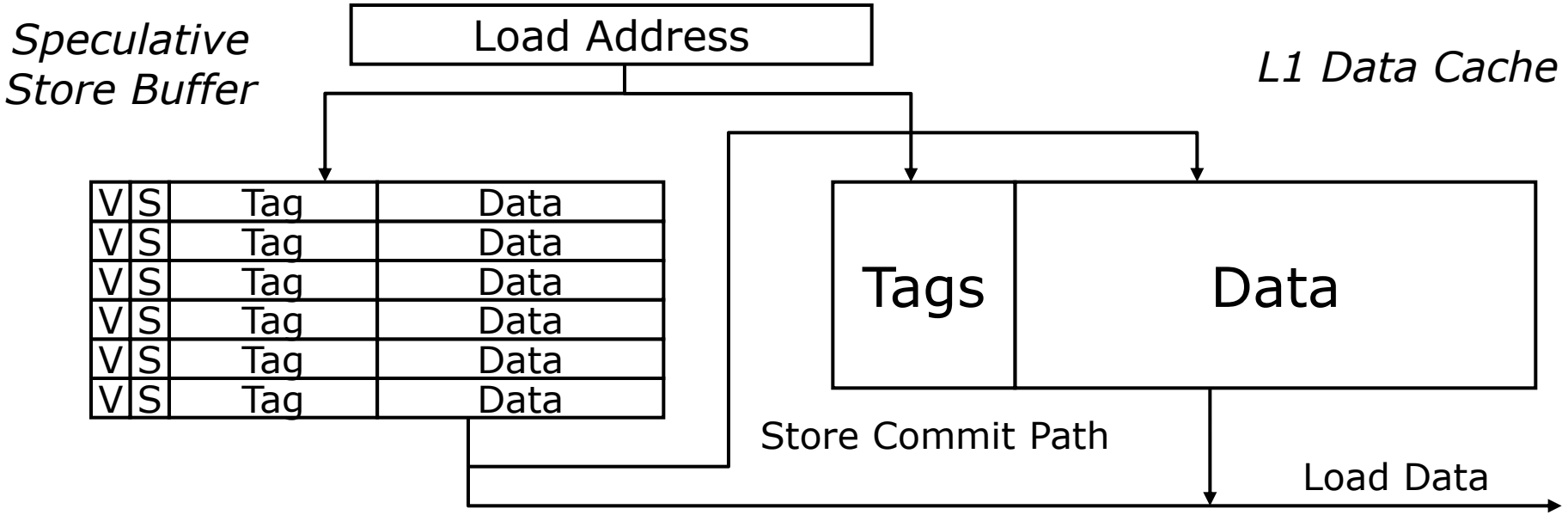
- 需要引进一种新的结构：推测式存储缓冲器（speculative store buffer） 来保存推测式store的数据

Speculative Store Buffer



- On store execute:
  - mark entry valid and speculative, and save data and tag of instruction.
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Speculative Store Buffer



- If data in both store buffer and cache, which should we use:

  Speculative store buffer

- If same address in store buffer twice, which should we use:

  Youngest store older than load

Datapath: Branch Prediction and Speculative Execution

至此，spuerscalar就讲完了