《计算机系统结构》课程直播
2020. 4.28

听不到声音请及时调试声音设备，可以下课后补签到

请将ZOOM名称改为“姓名”；

本节内容

- 动态调度 Tomasulo Algorithm 回顾
- Branch Prediction

From : H&P Computer Architecture: A Quantitative Approach,
Fifth Edition, （5th edition)

动态调度回顾

取指时猜测

按序、多发射

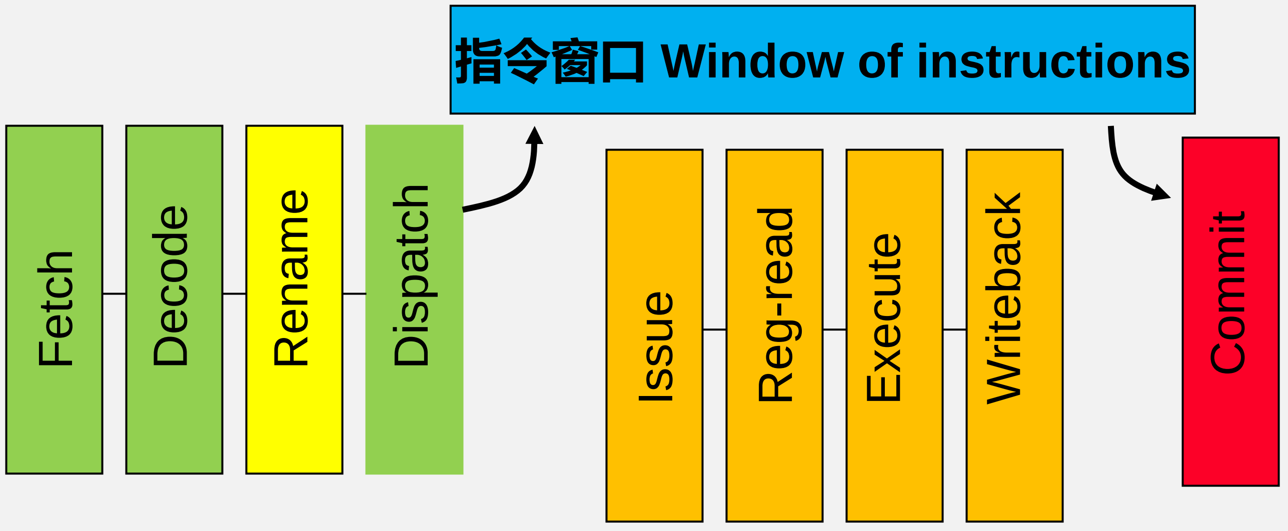
动态（硬件）调度

乱序执行

按序提交

Tomasulo Algorithm with ROB

动态调度乱序指令流水线

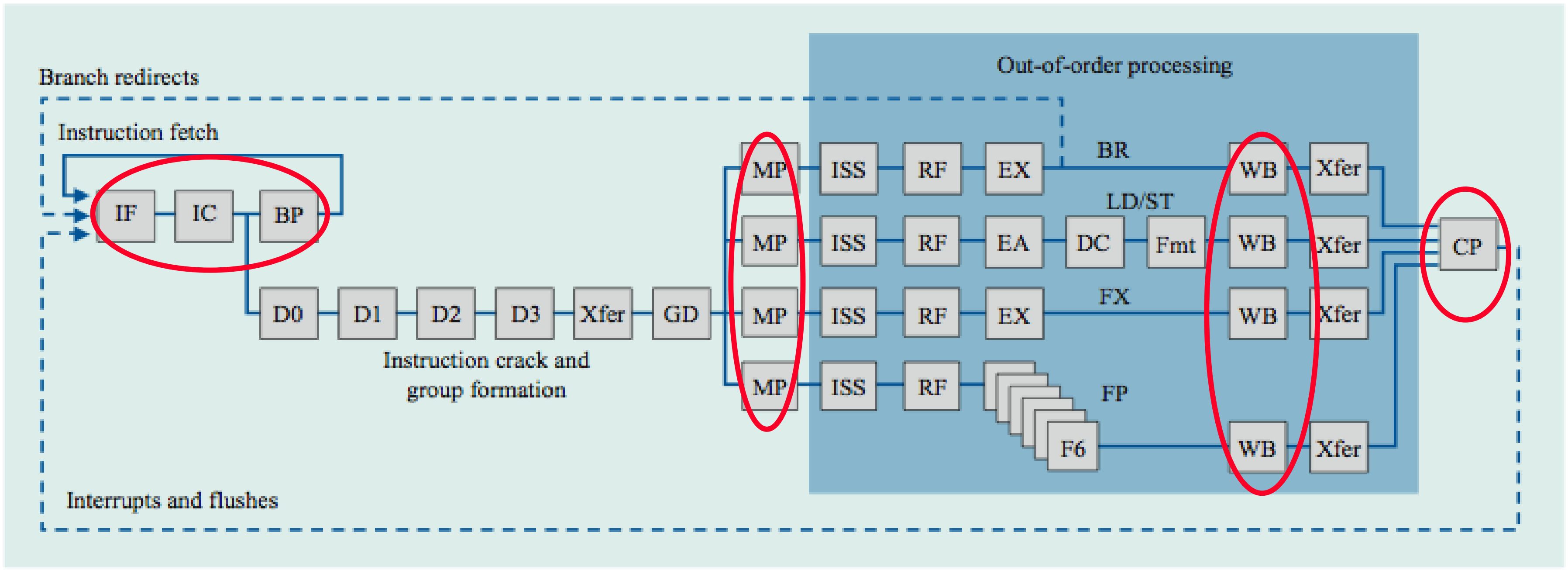


按序取指、译码
In-order front end

乱序执行
Out-of-order execution

按序提交
In-order commit

实例：IBM Power4



Instruction pipeline (IF: instruction fetch, IC: instruction cache,
BP: branch predict, D0: decode stage 0, Xfer: transfer, GD: group dispatch,
MP: mapping, ISS: instruction issue, RF: register file read, EX: execute,
EA: compute address, DC: data caches,
F6: six-cycle floating-point execution pipe,
Fmt: data format, WB: write back, and CP: group commit)

为什么要使用硬件调度（动态调度）？

- 为什么要使用硬件调度方案?
  - 在编译时无法确定的相关，可以通过硬件调度来优化
  - 编译器简单
  - 代码在不同组织结构的机器上，同样可以有效的运行

- 基本思想: 允许 stall后的指令继续向前流动

```
DIVD    F0,F2,F4
ADDD    F10,F0,F8
SUBD    F12,F8,F14
```

  - 允许乱序执行（out-of-order execution）

为什么顺序发射?

- 顺序发射使我们可以进行程序的数据流分析
  - 我们可以知道某条指令的结果会流向哪些指令
  - 如果我们乱序发射，可能会混淆RAW和WAR相关

- 每一周期发射多条指令也使用该原则将会正确地工作:
  - 需要多端口的 “rename table” ,以同时对一组指令所用的寄存器重命名
  - 需要在单周期内发射到多个RS中.
  - 寄存器文件需要有2x 个读端口和x个写端口(x：并行发射数)

为什么要按序提交?

- 乱序完成加大了实现精确中断的难度
  - 前面指令还没有完成时，寄存器文件中可能会有后面指令的运行结果
  - 如果这些前面的指令执行时有中断产生，怎么办?
  - 例如：DIVD F10, F0, F2
    - SUBD F4, F6, F8
    - ADDD F12, F14, F16
- 需要“rollback” 寄存器文件到原来的状态:
  - 精确中断的含义: 返回地址为:
    - 该地址之前的所有指令都已完成
    - 其后的指令还都没有完成
- 实现精确中断的技术：顺序完成（或提交）
  - 即提交指令完成的顺序必须与指令发射的顺序相同

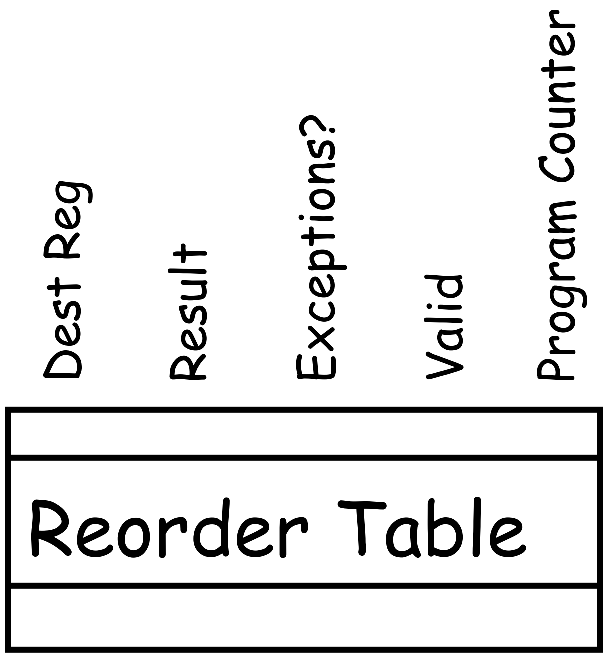
为什么要猜测执行?

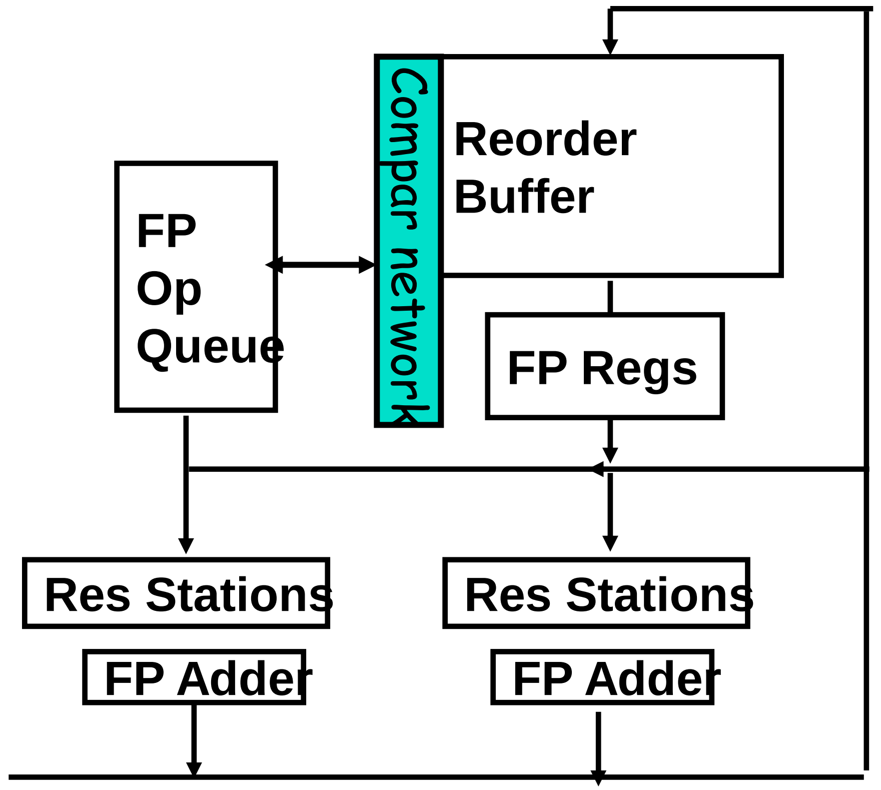
- 进行循环重叠执行需要尽快解决分支问题!
- 在循环展开的例子中，我们假设整数部件可以快速解决分支问题，以便进行循环重叠执行!

```
Loop:   LD      F0   0     R1
        MULTD   F4   F0    F2
        SD      F4   0     R1
        SUBI    R1   R1    #8
        BNEZ    R1   Loop
```

- 如果分支依赖于multd,怎么办??
  - 需要能预测分支方向
  - 如果分支成功，我们就可以重叠执行循环
- 对于superscalar机器这一问题更加突出

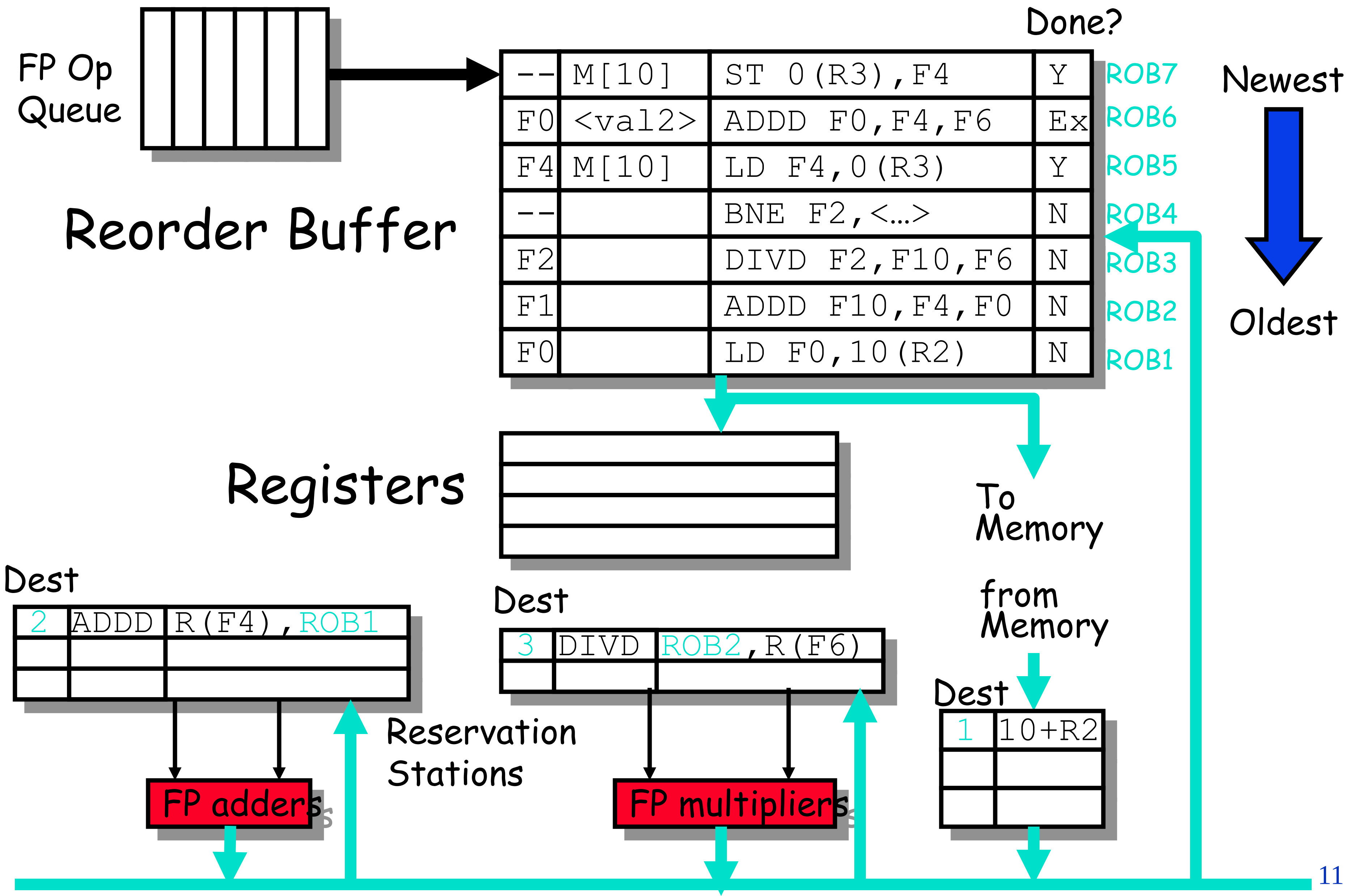
Tomasulo with reorder buffer (ROB)





- How do you find the latest version of a register?
  - As specified by Smith paper, need associative comparison network
  - Could use future file or just use the register result status buffer to track which specific reorder buffer has received the value
- Need as many ports on ROB as register file

Tomasulo With Reorder buffer:



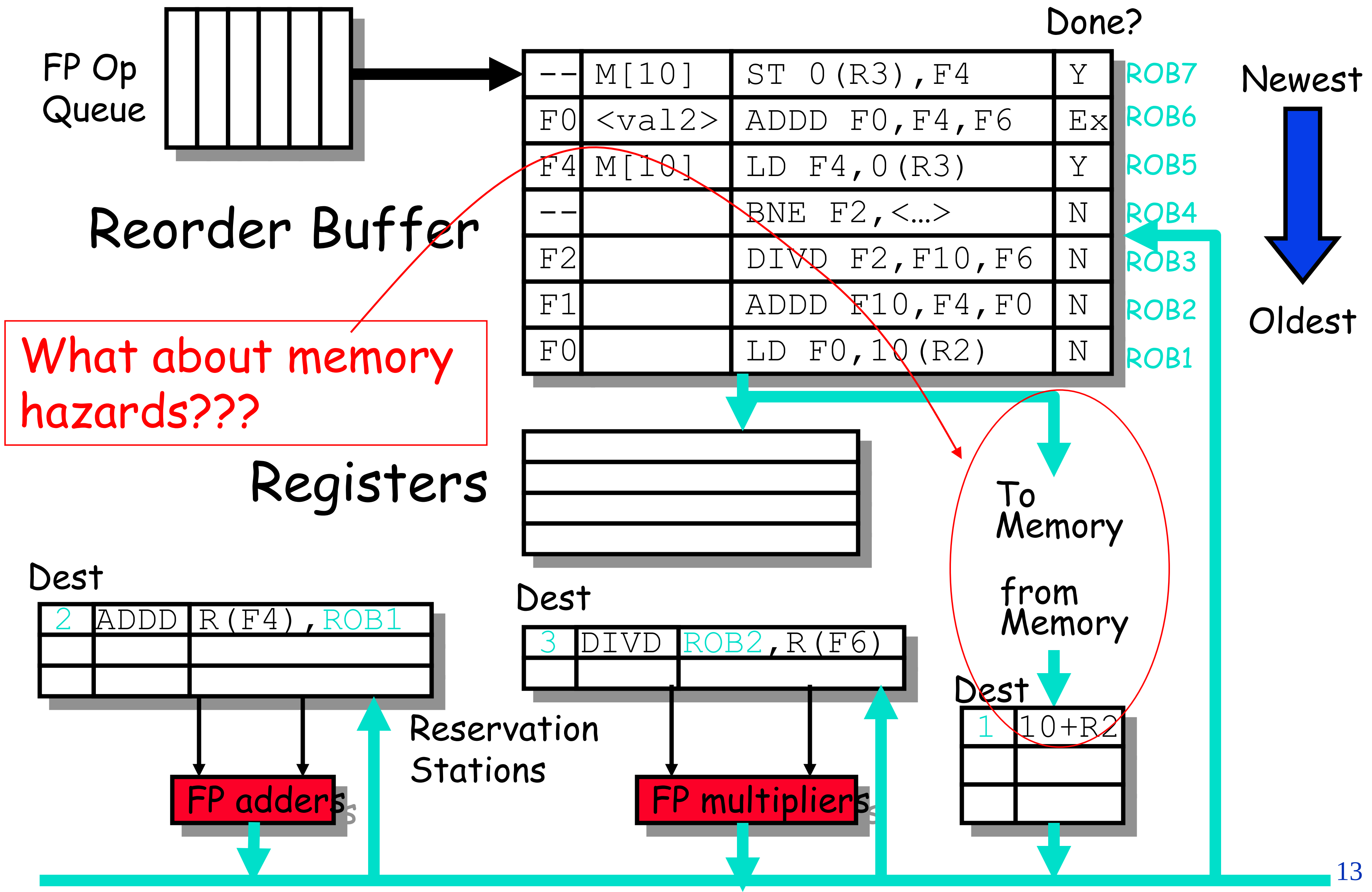
精确中断与猜测执行之间的关系

- Speculation is a form of guessing
  - Branch prediction, data prediction
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  - This is exactly same as precise exceptions!

- Branch prediction is a very important!
  - Need to “take our best shot” at predicting branch direction.
  - If we issue multiple instructions per cycle, lose lots of potential instructions otherwise:

- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*
  - This is why reorder buffers in all new processors

Tomasulo With Reorder buffer: 解决存储地址冲突？



FP Op Queue
Done?
Newest
Oldest
Reorder Buffer
-- | M[10] | ST 0(R3),F4 | Y | ROB7
F0 | <val2> | ADDD F0,F4,F6 | Ex | ROB6
F4 | M[10] | LD F4,0(R3) | Y | ROB5
-- | | BNE F2,<…> | N | ROB4
F2 | | DIVD F2,F10,F6 | N | ROB3
F1 | | ADDD F10,F4,F0 | N | ROB2
F0 | | LD F0,10(R2) | N | ROB1
What about memory hazards???
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
Dest
3 DIVD ROB2,R(F6)
Dest
1 10+R2
Reservation Stations
FP adders
FP multipliers

解决存储地址冲突 : Sorting out RAW Hazards in memory

- Question: Given a load that follows a store in program order, are the two related?
  - (Alternatively: is there a RAW hazard between the store and the load)?

Eg:
```
st      0(R2),R5
ld      R6,0(R3)
```

- Can we go ahead and start the load early?
  - Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
  - We might want to issue/begin execution of both operations in same cycle.
  - "no speculation": Answer is that we are not allowed to start load until we know that address 0(R2) ≠ 0(R3)
  - "dependence speculation": we might guess at whether or not they are dependent (called and use reorder buffer to fixup if we are wrong).

Hardware Support for Memory Disambiguation

- Need buffer to keep track of all outstanding stores to memory, in program order.
  - Keep track of address (when becomes available) and value (when becomes available)
  - FIFO ordering: will retire stores from this buffer in program order
- When issuing a load, record current head of store queue (know which stores are ahead of you).
- When have address for load, check store queue:
  - If *any* store prior to load is waiting for its address, stall load.
  - If load address matches earlier store address (associative lookup), then we have a *memory-induced RAW hazard*:
    - store value available $\Rightarrow$ return value
    - store value not available $\Rightarrow$ return ROB number of source
  - Otherwise, send out request to memory
- Actual stores commit in order, so no worry about WAR/WAW hazards through memory.

Tomasulo 算法的特点

- 控制和缓存分布在各部件中
  - FU 缓存称“reservation stations(RS) ”; 保存待用操作数
- 指令中的寄存器在RS中用寄存器值或指向RS的指针代替（称为register renaming）
  - 避免 WAR, WAW hazards
  - RS多于寄存器，因此可以做更多编译器无法做的优化
- 传给FU的结果从RS来而不是从寄存器来，FU的计算结果通过Common Data Bus 以广播方式发向所有功能部件
- Load和Store部件也看作带有RS的功能部件
- 可以跨越分支，允许FP操作队列中FP操作不仅仅局限于基本块

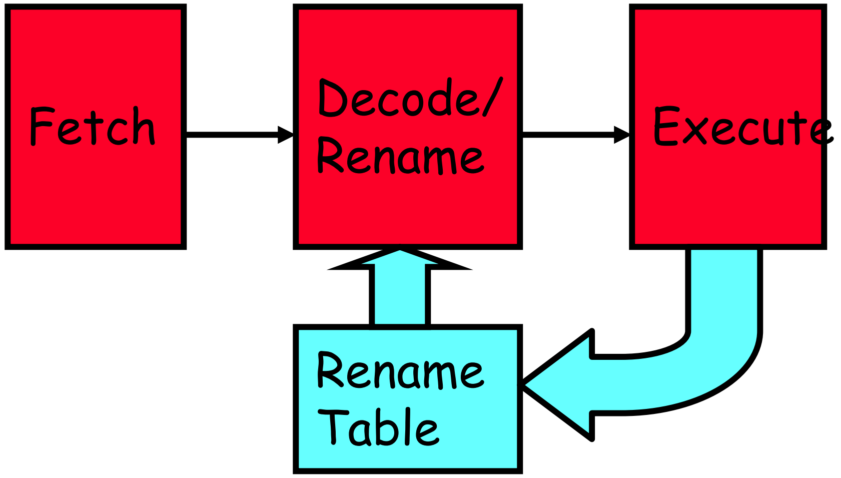
Tomasulo 缺陷

- 复杂
  - delays of 360/91, MIPS 10000, IBM 620?
- 要求高速CDB
  - 性能受限于Common Data Bus

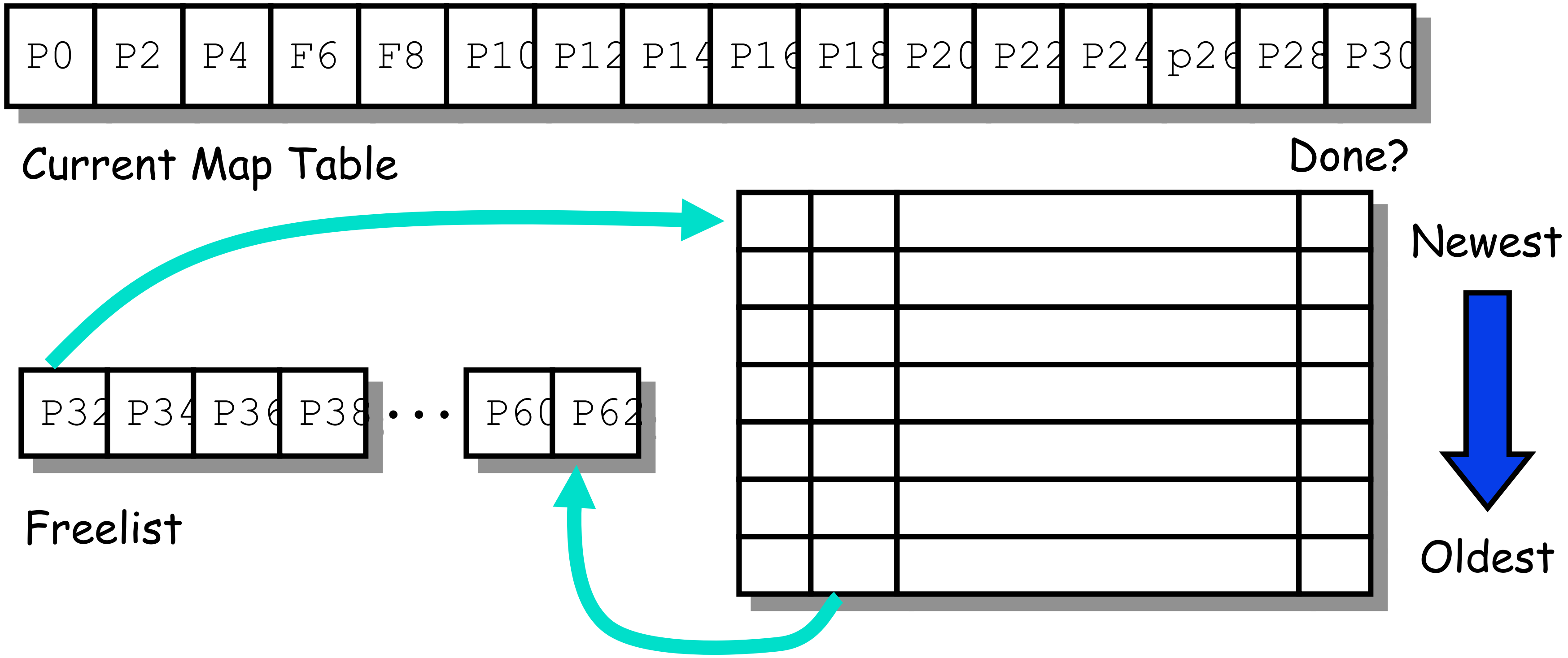
教材：Ch. 3.4-3.5

Tomasulo with Explicit Register Renaming

- Make use of a *physical* register file that is larger than number of registers specified by ISA
- Keep a translation table:
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.

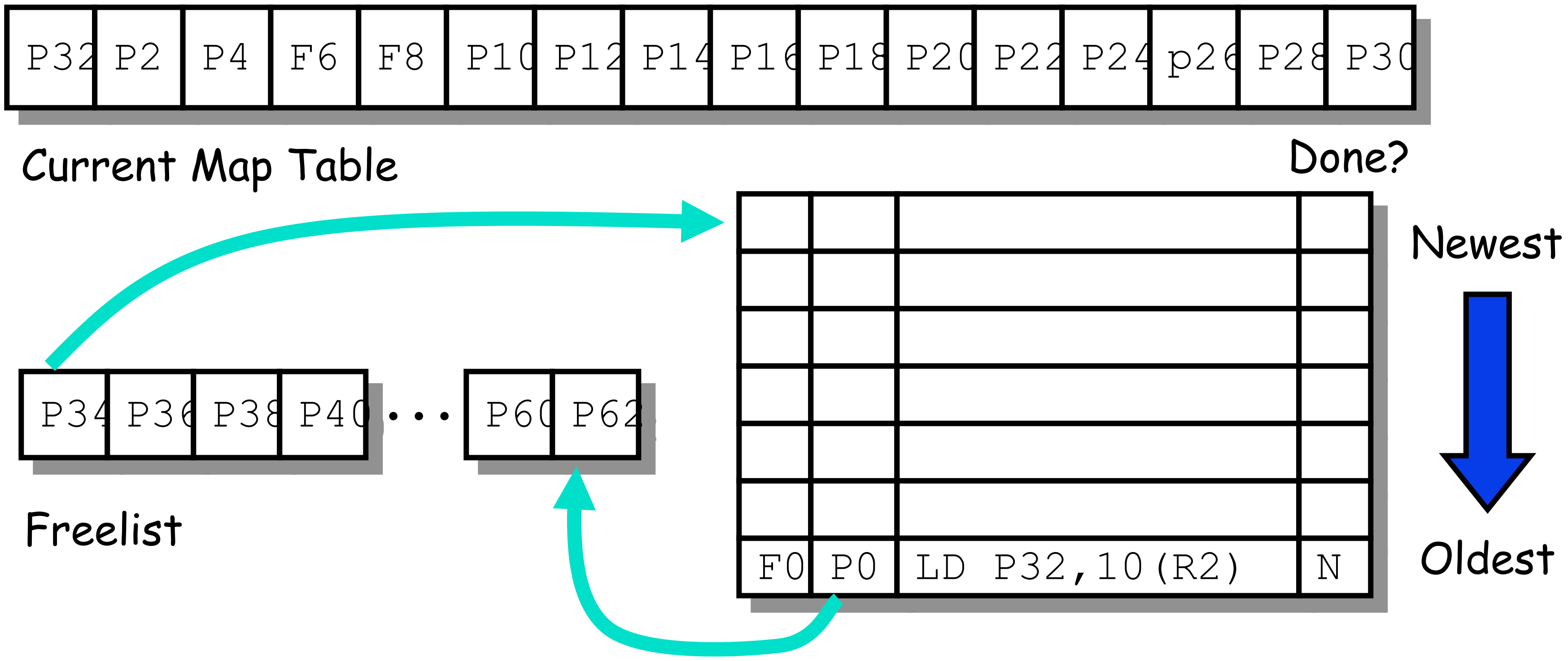


Explicit register renaming:
R10000 Freelist Management



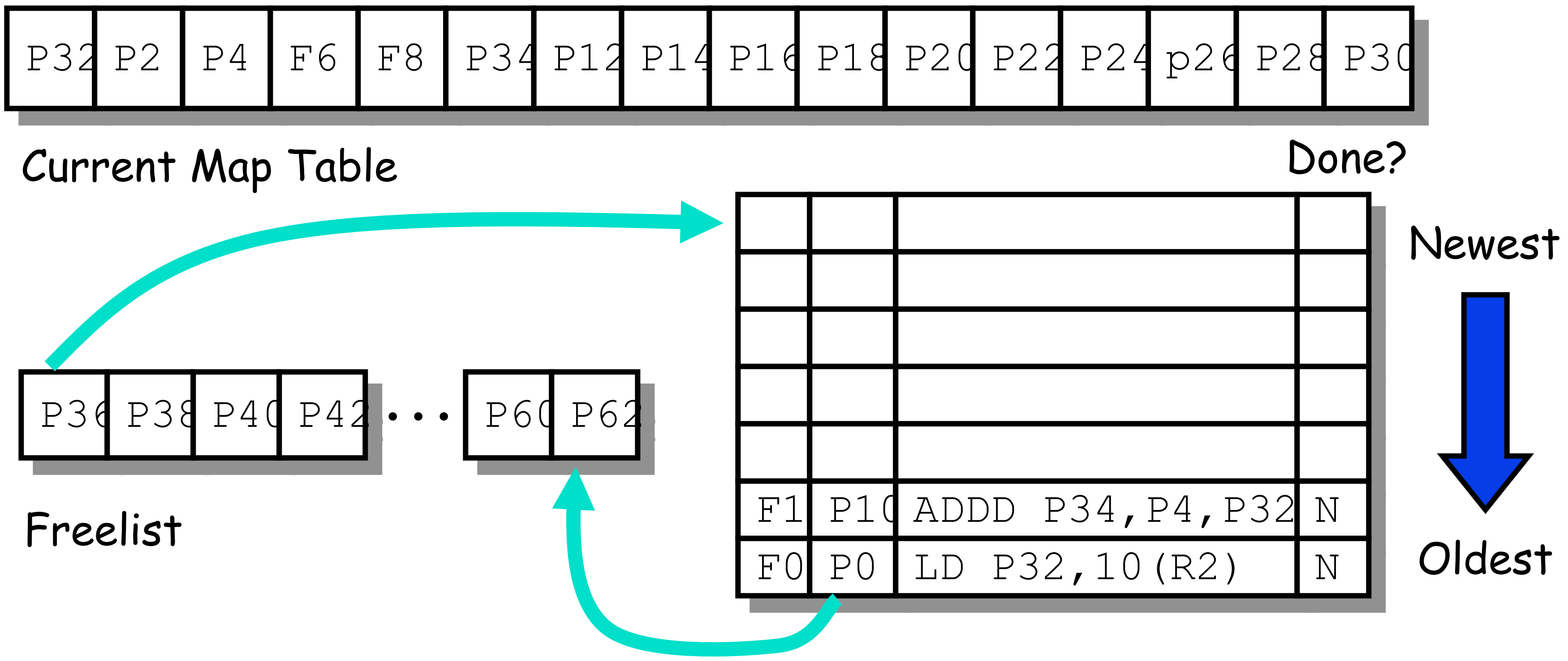
- Physical register file larger than ISA register file
- On issue, each instruction that modifies a register is allocated new physical register from freelist
- Used on: R10000, Alpha 21264, HP PA8000

Explicit register renaming:
R10000 Freelist Management

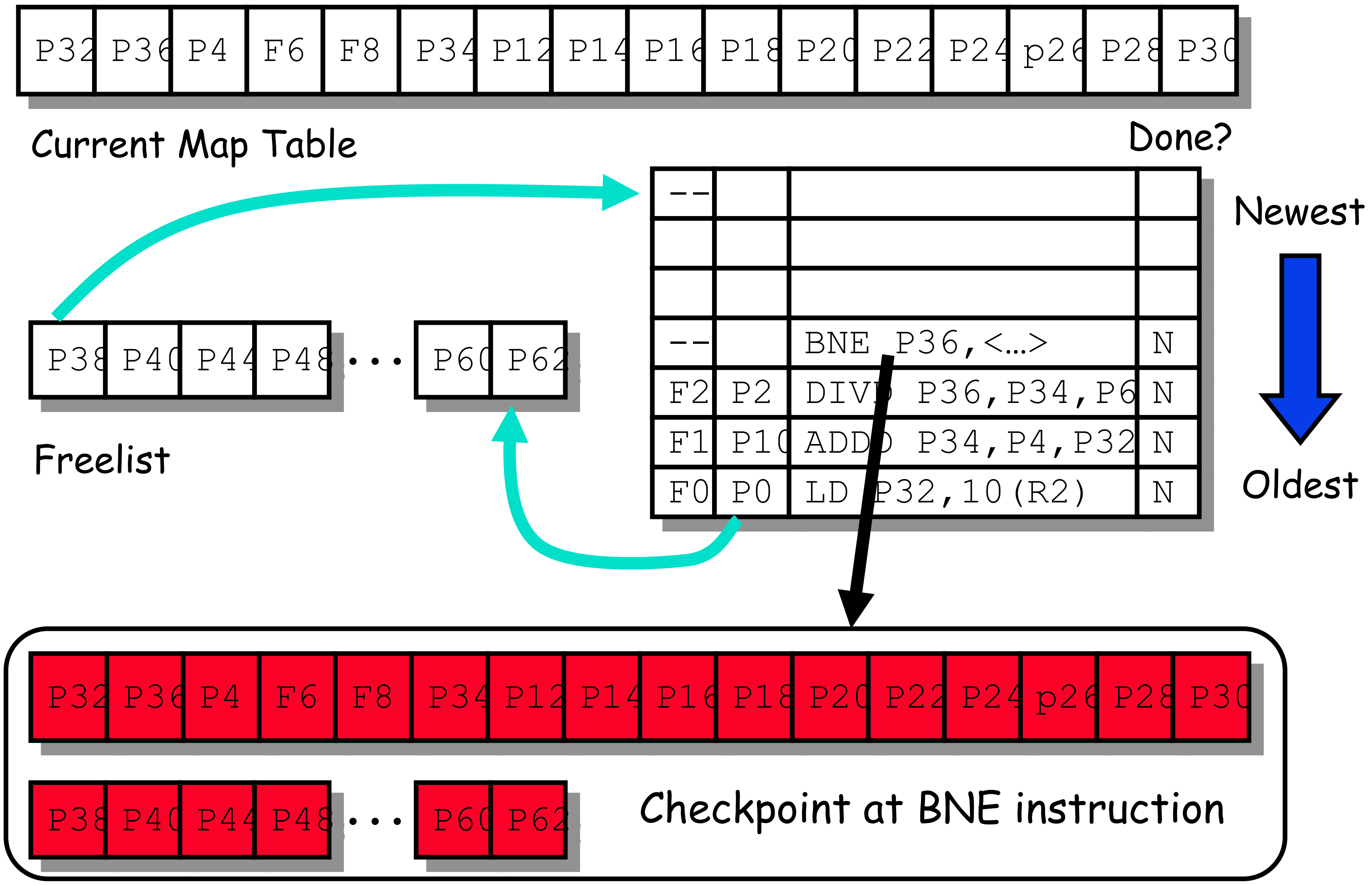


- Note that physical register P0 is “dead” (or not “live”) past the point of this load.
  - When we go to commit the load, we free up

Explicit register renaming:
R10000 Freelist Management

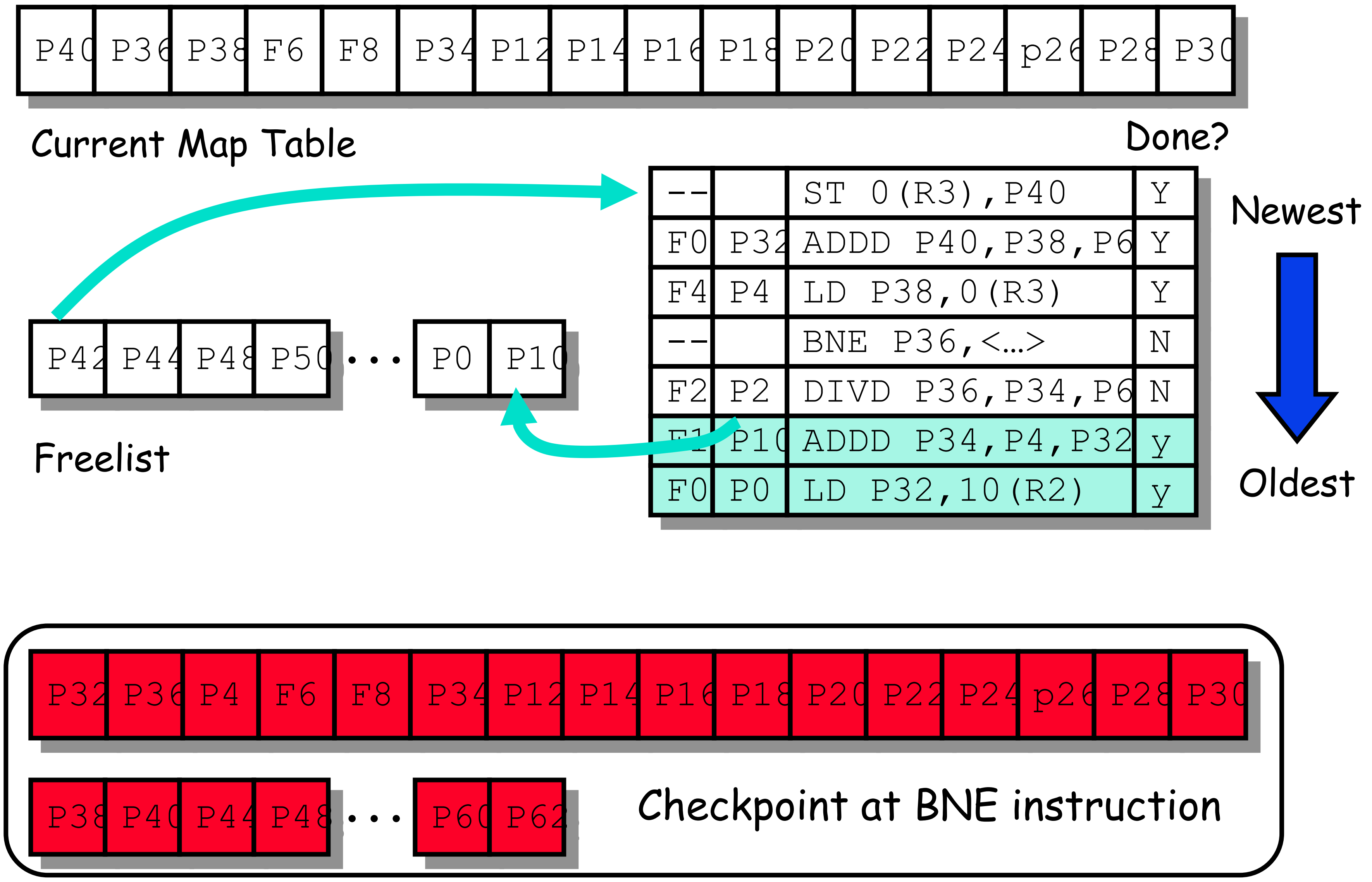


Explicit register renaming:
R10000 Freelist Management



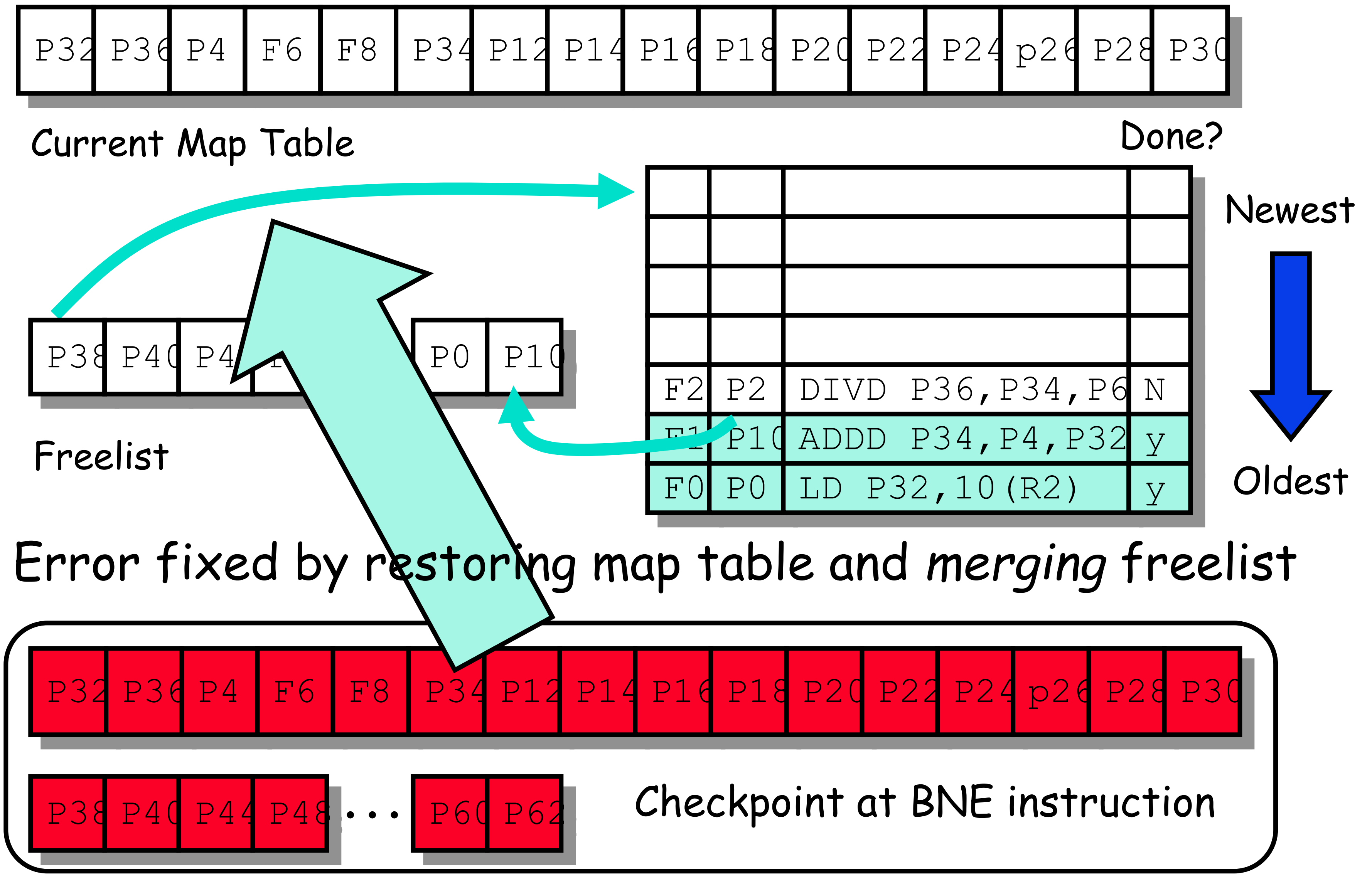
P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
Newest
Oldest
-- | | BNE P36,<…> | N
F2 | P2 | DIVD P36,P34,P6 | N
F1 | P10 | ADDD P34,P4,P32 | N
F0 | P0 | LD P32,10(R2) | N
P38 P40 P44 P48 … P60 P62
Freelist
Checkpoint at BNE instruction

Explicit register renaming:
R10000 Freelist Management



P40 P36 P38 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
-- | | ST 0(R3),P40 | Y
F0 | P32 | ADDD P40,P38,P6 | Y
F4 | P4 | LD P38,0(R3) | Y
-- | | BNE P36,<…> | N
F2 | P2 | DIVD P36,P34,P6 | N
F10 | P10 | ADDD P34,P4,P32 | y
F0 | P0 | LD P32,10(R2) | y
Newest
Oldest
P42 P44 P48 P50 … P0 P10
Freelist
P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
P38 P40 P44 P48 … P60 P62
Checkpoint at BNE instruction

Explicit register renaming:
R10000 Freelist Management

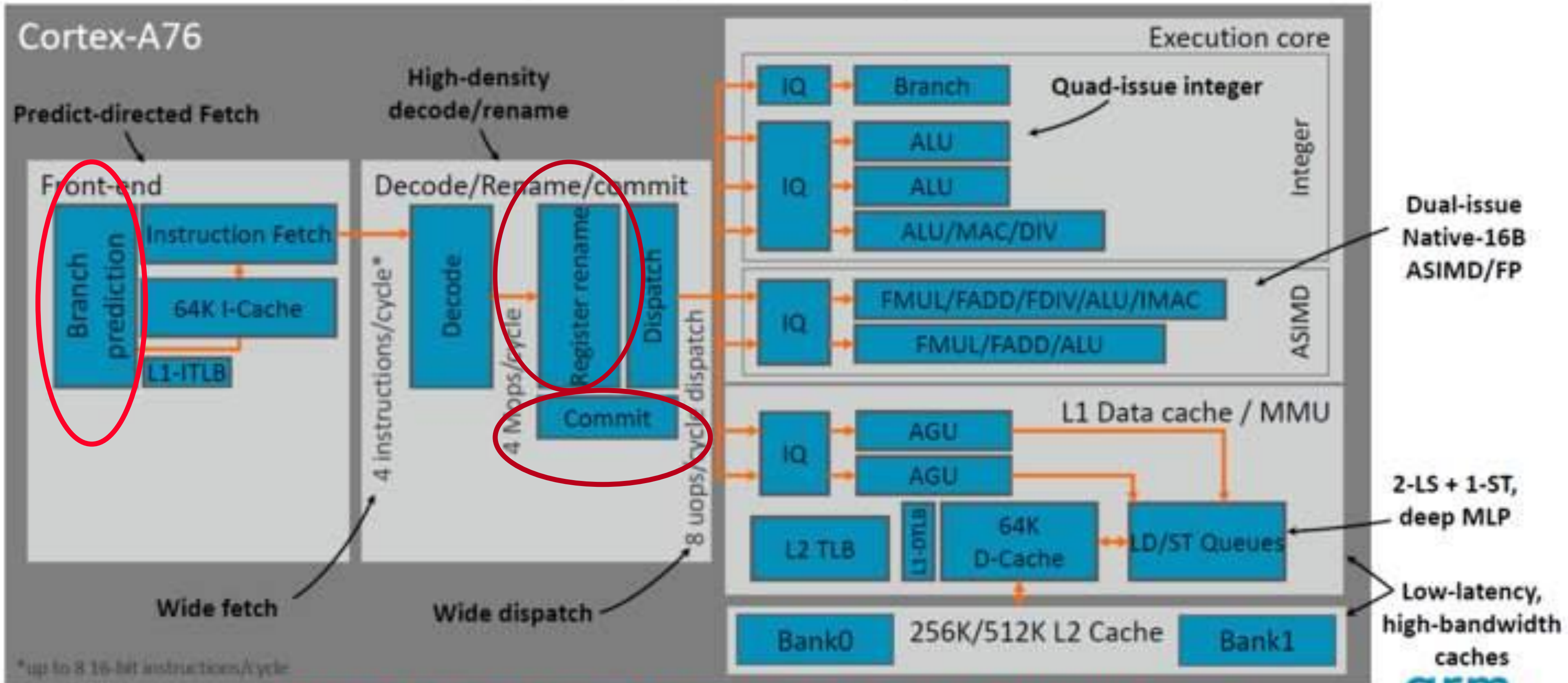


P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
Newest
Oldest
P38 P40 P4 P0 P10
Freelist
F2 P2 DIVD P36,P34,P6 N
F1 P10 ADDD P34,P4,P32 y
F0 P0 LD P32,10(R2) y
P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
P38 P40 P44 P48 . . . P60 P62
Checkpoint at BNE instruction

Error fixed by restoring map table and *merging* freelist

Cortex-A76: Microarchitecture overview

The foundation of a new family of high-performance products



The embargo for this content presented at Arm Tech Day will lift on Thursday, May 31st at 12pm Pacific Standard Time. Corresponding UK and Japan times are: Thursday, May 31st 8pm BST/Friday, June 1st at 4am JST

arm

转移预测

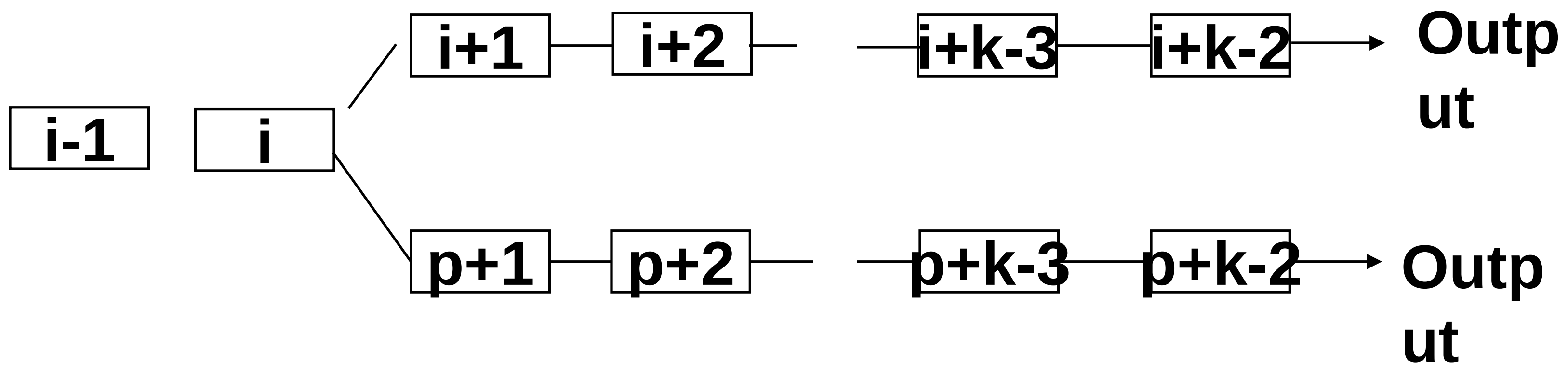
Branch Prediction

控制相关的动态解决技术

- 控制相关：
  - 由条件转移或程序中断引起的相关，也称全局相关。
  - 控制相关对流水线的吞吐率和效率影响相对于数据相关要大得多
    - 条件指令在一般程序中所占的比例相当大
    - 中断虽然在程序中所占的比例不大，但中断发生在程序中的哪条指令，发生在一条指令执行过程中的哪个功能段都是不确定的
- 处理条件转移和中断引起的控制相关的关键问题：
  - 要确保流水线能够正常工作
  - 减少因断流引起的吞吐率和效率的下降

分支对性能的影响

- 假设在一条有K段的流水线中，在最后一段才能确定目标地址



- 当分支方向预测错误时
  - 流水线中有多个功能段要浪费
  - 可能造成程序执行结果发生错误
  - 因此当程序沿着错误方向运行后，作废这些程序时，一定不能破坏通用寄存器和主存储器的内容。

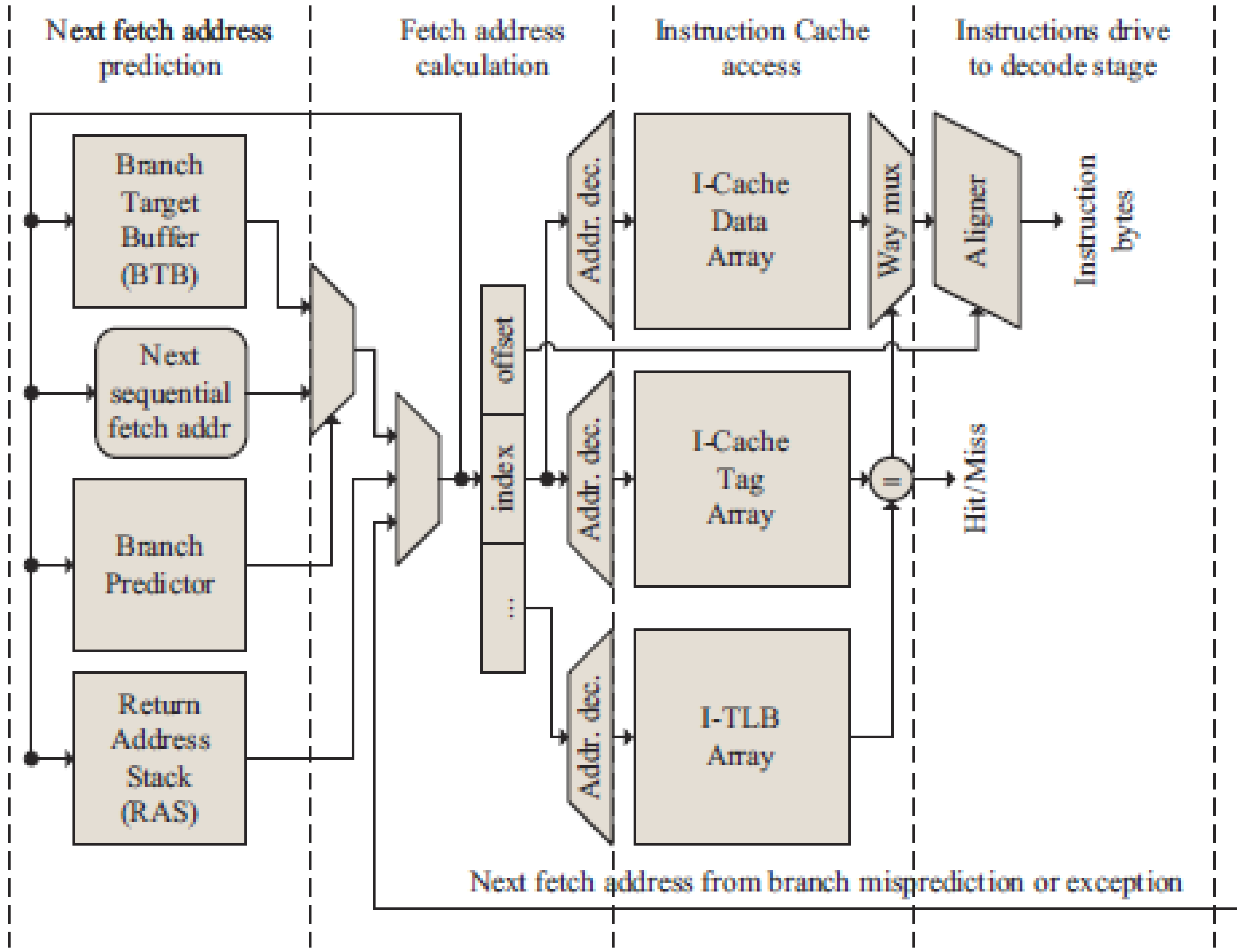
条件转移指令对流水线性能的影响

- 假设对于一条有K段的流水线，由于条件分支的影响，在最坏情况下，每次条件转移将造成k-1个时钟周期的断流。
- 假设条件分支在一般程序中所占的比例为p, 采用分支预测失败策略，条件成功的概率为q。试分析分支对流水线的影响。

- 结论：条件转移指令对流水线的影响很大，必须采取相关措施来减少这种影响。

- 预测可以是静态预测“Static” (at compile time) 或动态预测“Dynamic” (at runtime)
  - 例如：一个循环供循环10次，它将分支成功9次，1次不成功。
  - 动态分支预测 vs. 静态分支预测，哪个好?

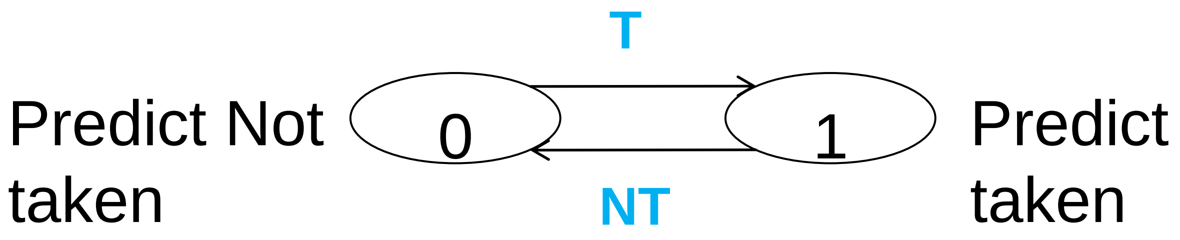
Instruction Fetch Unit


Next fetch address prediction
Fetch address calculation
Instruction Cache access
Instructions drive to decode stage
Branch Target Buffer (BTB)
Next sequential fetch addr
Branch Predictor
Return Address Stack (RAS)
offset
index
...
Addr. dec.
Addr. dec.
Addr. dec.
I-Cache Data Array
I-Cache Tag Array
I-TLB Array
Way mux
Aligner
=
Instruction bytes
Hit/Miss
Next fetch address from branch misprediction or exception

Dynamic Branch Prediction

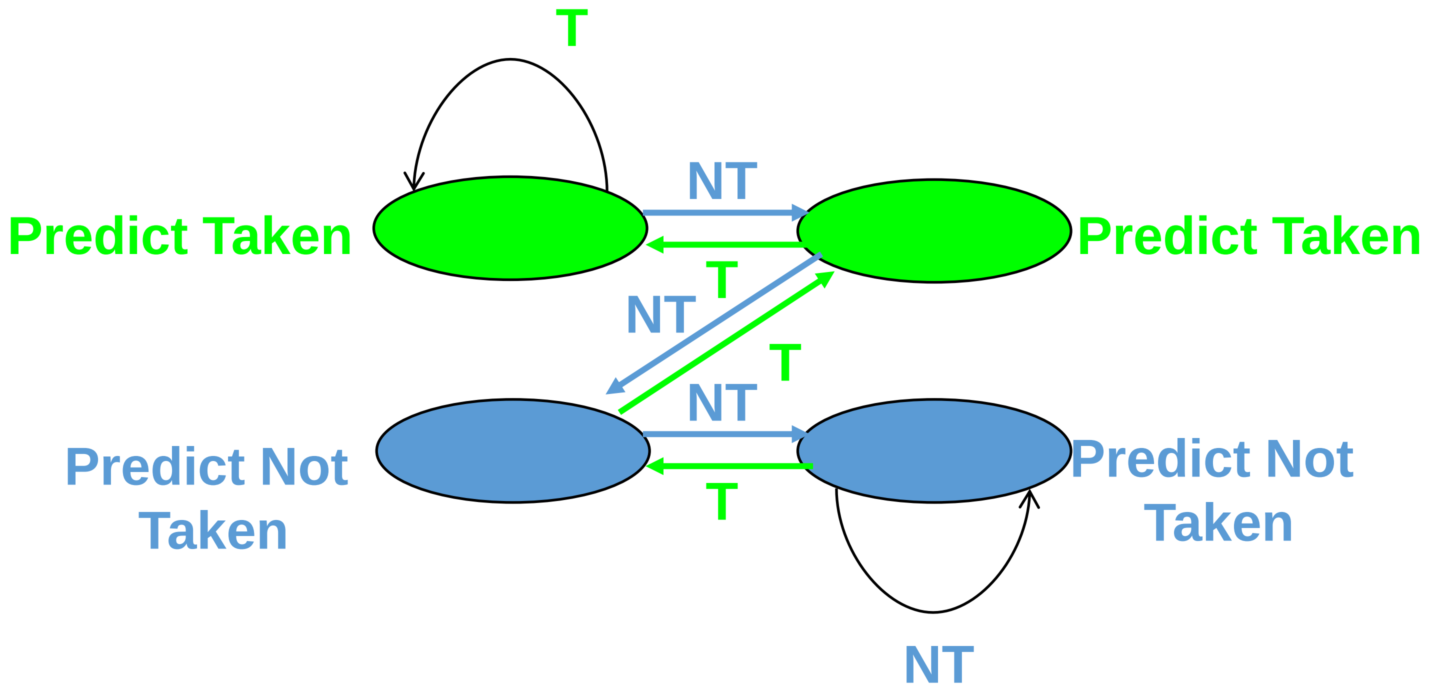
- 动态分支预测：预测分支的方向在程序运行时刻动态确定
- 需解决的关键问题是：
  - 如何记录转移历史信息
  - 如何根据所记录的转移历史信息，预测转移的方向
- 主要方法
  - 基于BPB(Branch Prediction Buffer)或BHT(Branch History Table)
    - 1-bit BHT和2-bit BHT
    - Correlating Branch Predictors
    - Tournament Predictors: Adaptively Combining Local and Global Predictors
  - High Performance Instruction Delivery
    - BTB
    - Integrated Instruction Fetch Units
    - Return Address Predictors
- Performance = *f*(accuracy, cost of misprediction)
  - Misprediction : Flush Reorder Buffer

1-bit BHT



- Branch History Table:
  - 分支指令的PC的低位索引
  - 该表记录上一次转移是否成功
  - 1-bit BHT
- 问题: 在一个循环中, 1-bit BHT 将导致2次分支预测错误
  - 假设一循环次数为10次的简单程序段
  - 最后一次循环
    - 前面为预测成功，最后一次需要退出循环
  - 首次循环
    - 前面为预测为失败，这次实际上为成功

2-bit BHT



- 解决办法: 2位记录分支历史
- Blue: stop, not taken
- Green: go, taken

2-bit BHT 预测的错误率

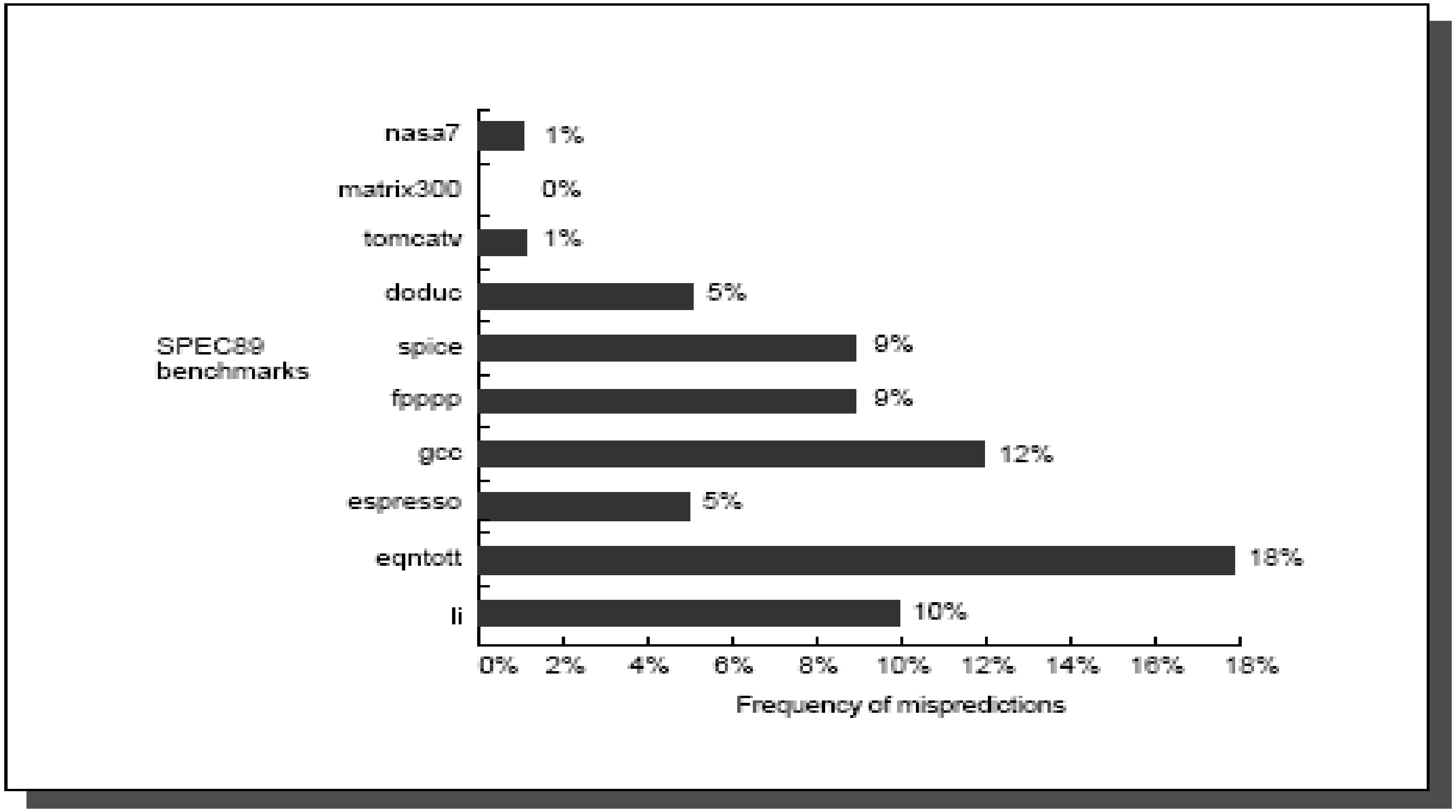


FIGURE 3.8 Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the FP programs (average of 4%). Even omitting the FP kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch prediction study done using the IBM Power architecture and optimized code for that system. See Pan et al. [1992].

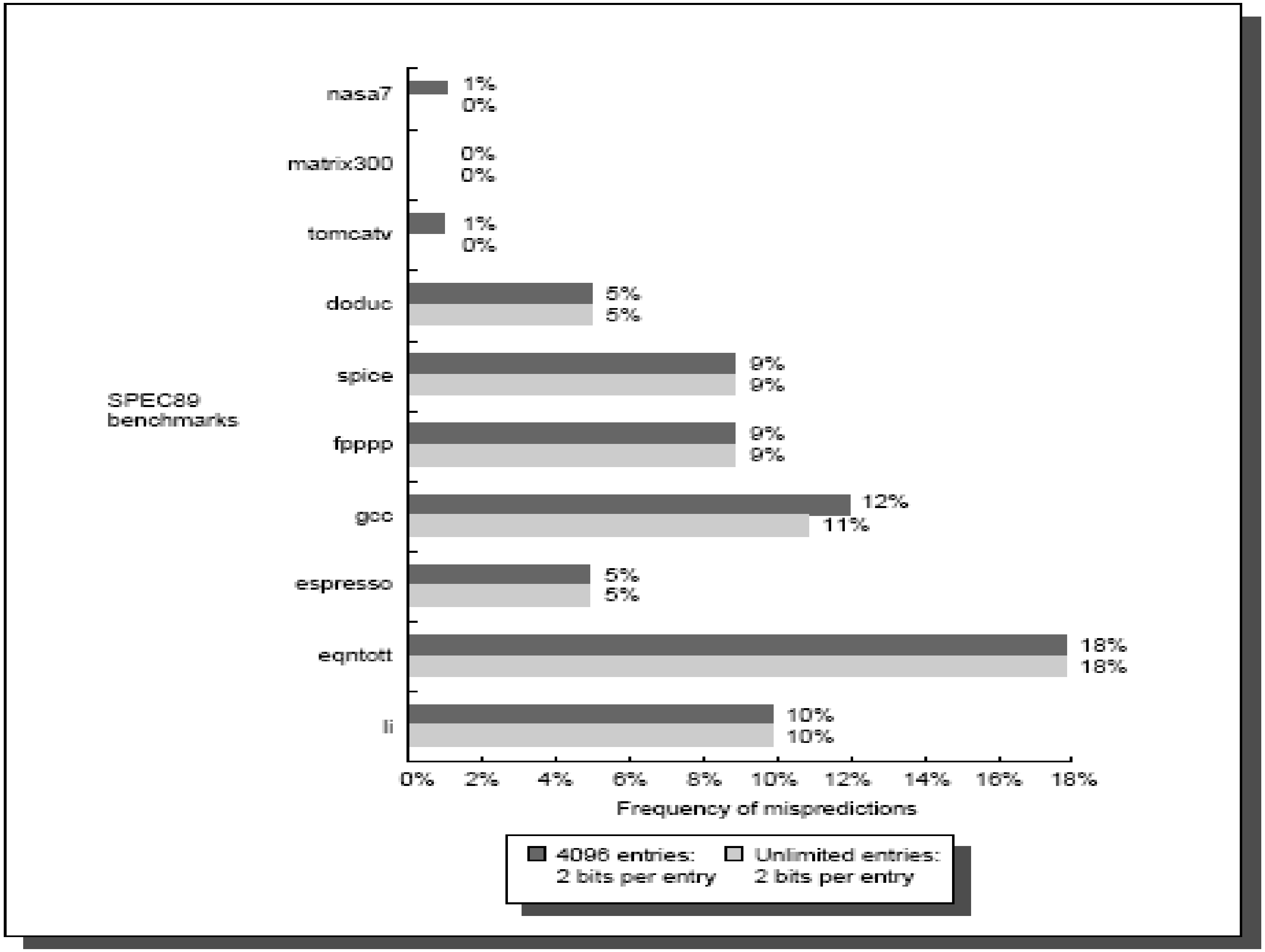


FIGURE 3.9 Prediction accuracy of a 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks.

BHT Accuracy

- 分支预测错误的原因:
  - 预测错误
  - 由于使用PC的低位查找BHT表，可能得到错误的分支历史记录
- BHT表的大小问题
  - 4096 项的表分支预测错误的比例为1%（nasa7, tomcatv) to 18% (eqntott)，spice at 9% and gcc at 12%
  - 再增加项数，对提高预测准确率几乎没有效果 (in Alpha 21164)

Correlating Branch Predicator

- 例如：

```
        if (aa==2)
aa=0;
        if (bb==2)
bb=0;
        if (aa!=bb) {
```

- 翻译为MIPS

```
      SUBI R3,R1,#2
      BNEZ R3,L1          ; branch b1 (aa!=2)
      ADDI  R1,R0,R0      ;aa=0
L1:   SUBI R3,R2,#2
      BNEZ R3,L2          ;branch b2(bb!=2)
      ADDI R2,R0,R0       ; bb=0
L2:   SUBI R3,R1,R2       ;R3=aa-bb
      BEQZ R3,L3       ;branch b3 (aa==bb)
```

- 观察结果：

  b3 与分支b2 和b1相关。

  如果b1和b2都分支失败， 则b3一定成功。

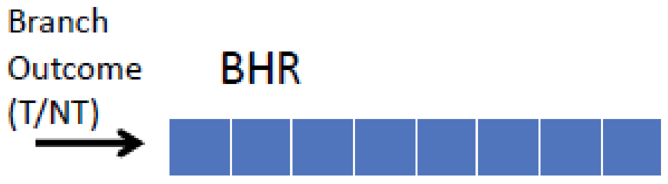
Sometimes 2-bit Saturating Counter Can't Work Well

- Local prediction

  For (i=1; i<=4; i++) {……}

  - Branch history pattern
    - 1110 1110 1110 1110 …

History pattern	prediction
111	0
110	1
101	1
011	1



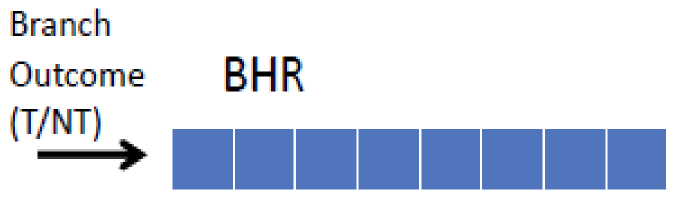
Example

- Global prediction

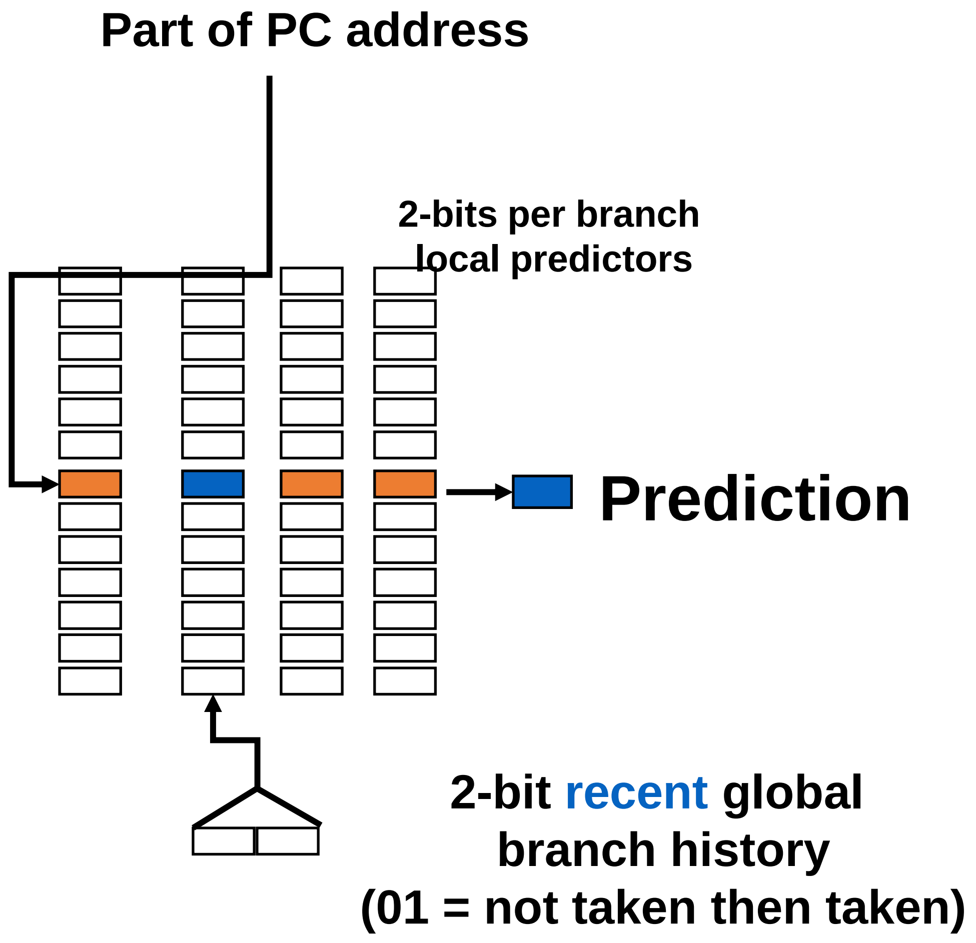
```
        if (aa==2)
                aa=0;
        if (bb==2)
                bb=0;
        if (aa!=bb) {
```

is translated as

```
        DSUBUIR3,R1,#2
        BNEZ    R3,L1           ;branch b1 (aa!=2)
        DADD    R1,R0,R0        ;aa=0
L1:     DSUBUIR3,R2,#2
        BNEZ    R3,L2           ;branch b2 (bb!=2)
        DADD    R2,R0,R0        ;bb=0
L2:     DSUBU R3,R1,R2          ;R3=aa-bb
        BEQZ    R3,L3           ;branch b3
```



Correlating Branches


Part of PC address
2-bits per branch local predictors
Prediction
2-bit recent global branch history
(01 = not taken then taken)

- (2,2) Correlating predictor

Gshare predictor

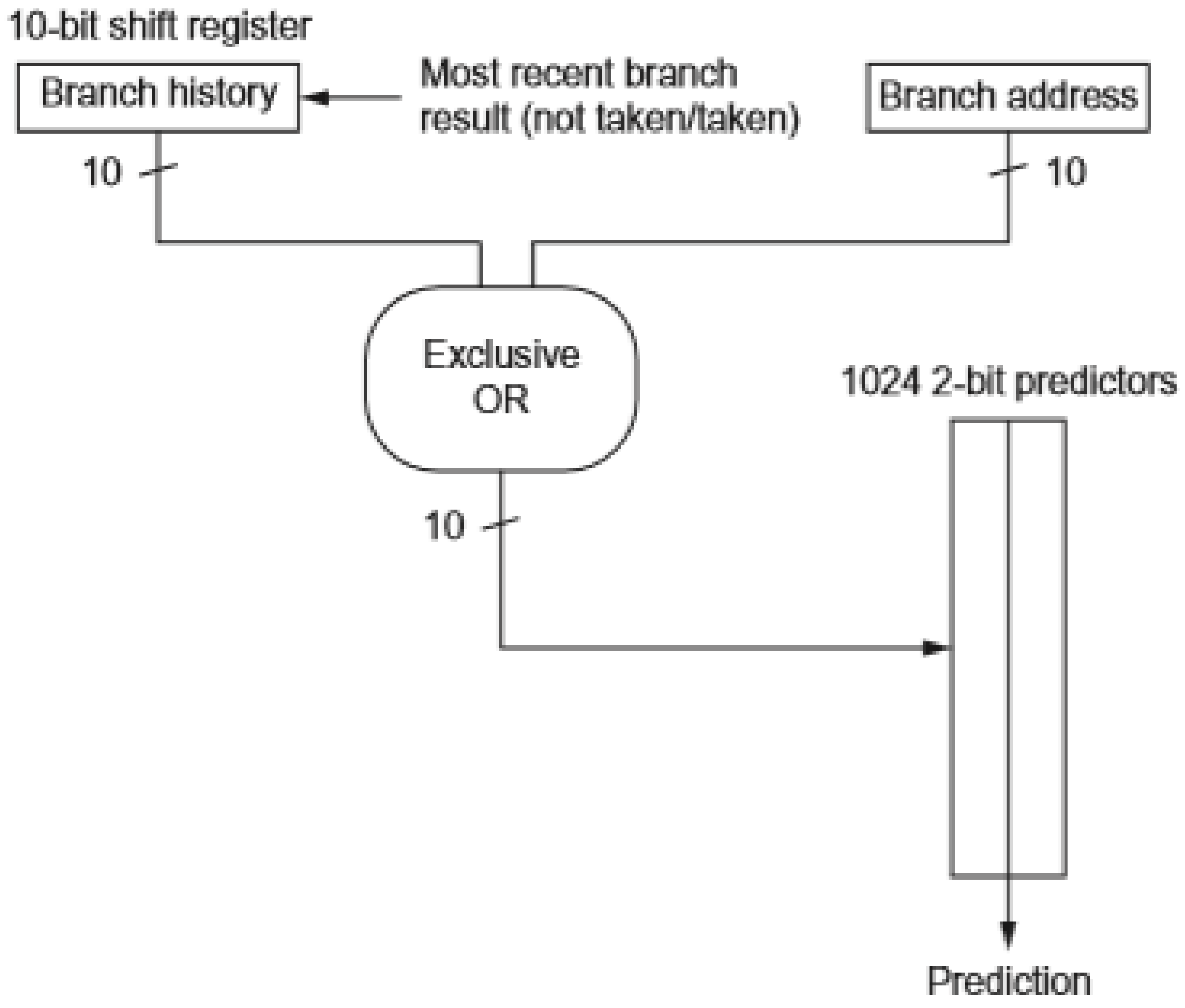


Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

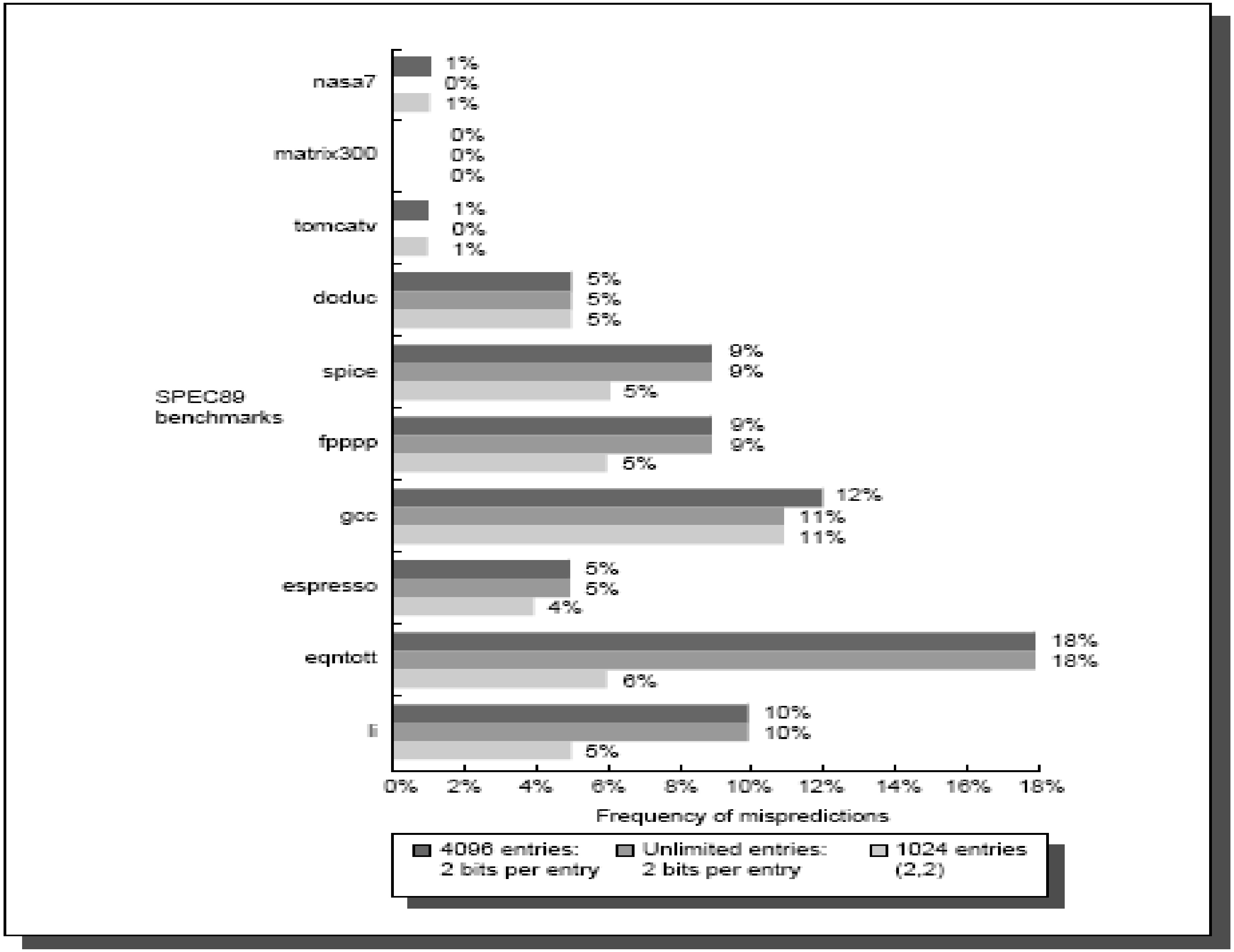
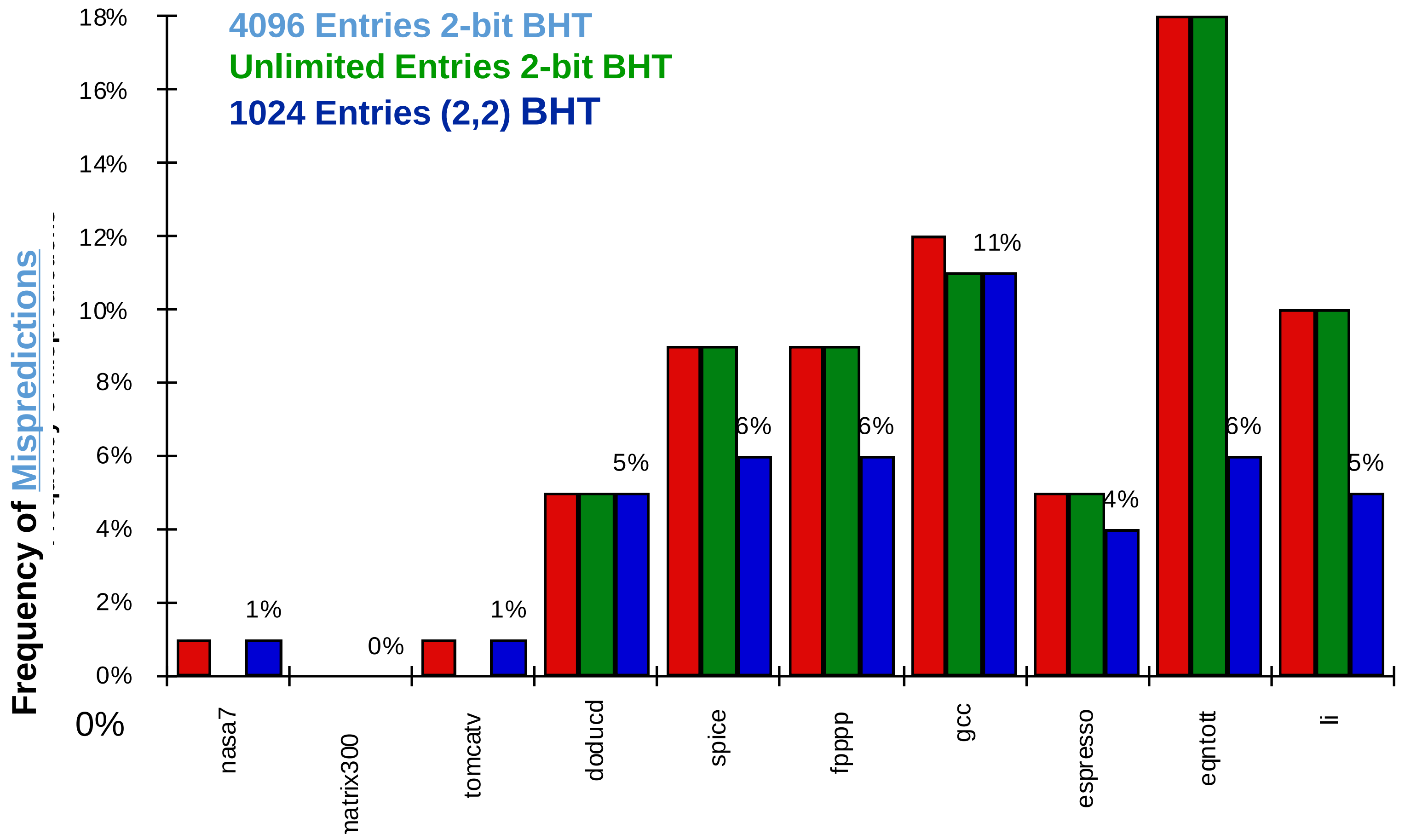


FIGURE 3.15 Comparison of two-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating two-bit predictor with unlimited entries and a two-bit predictor with two bits of global history and a total of 1024 entries.

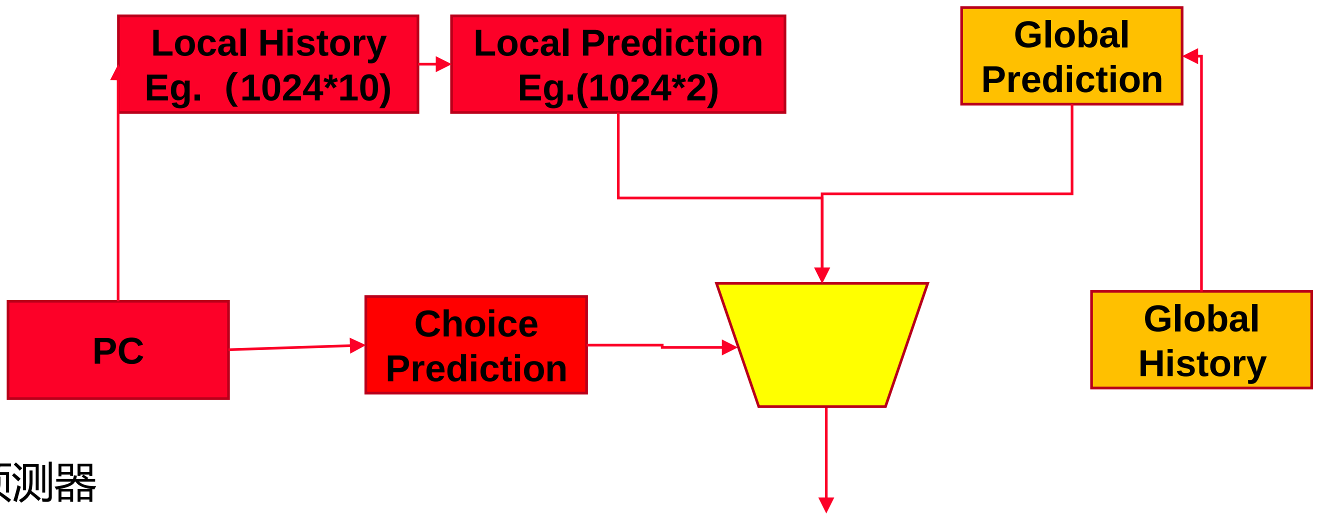
Accuracy of Different Schemes



Branch Prediction

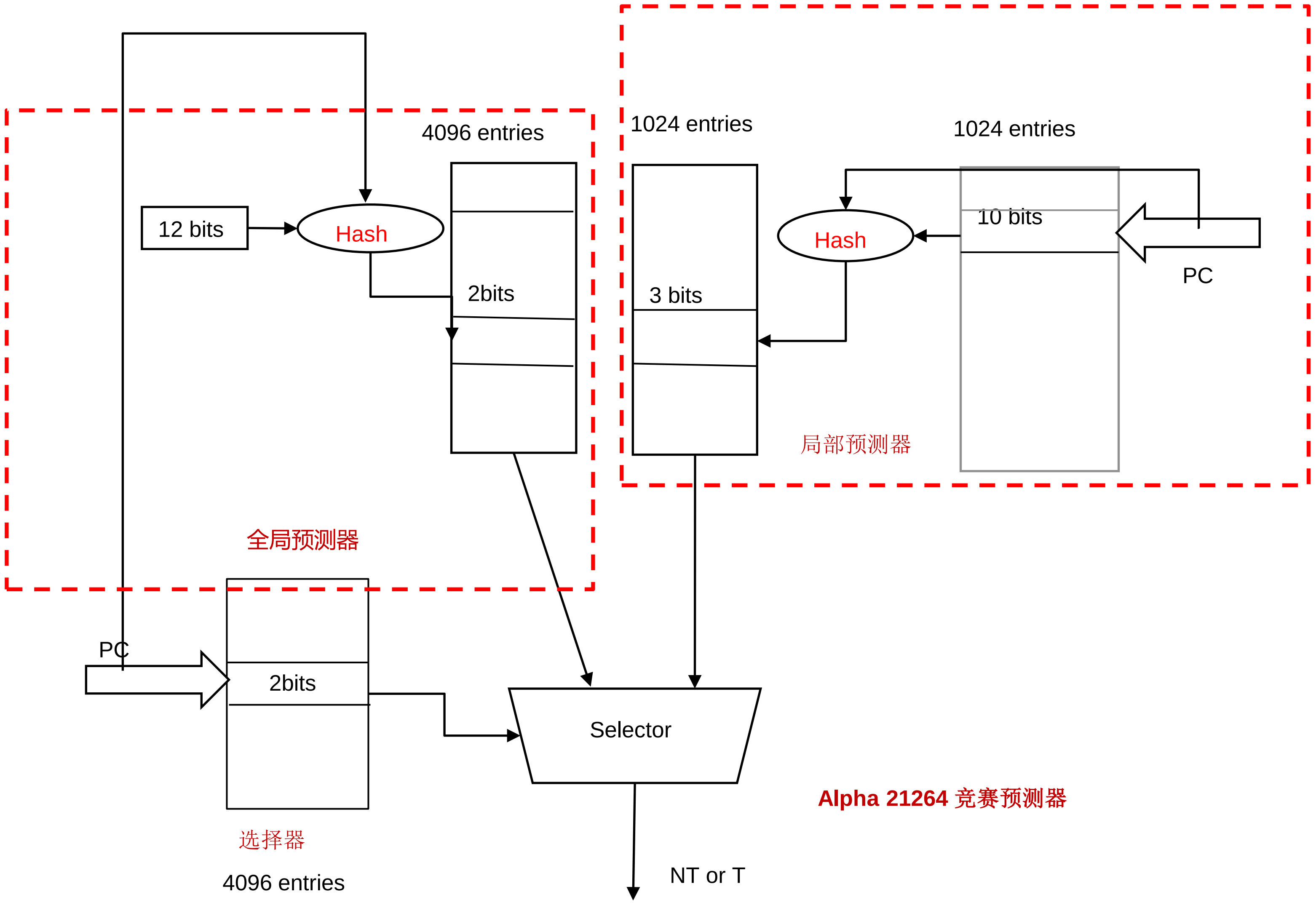
- Basic 2-bit predictor:
- 关联预测器(n,2):
  - 每个分支有多个 2-bit 预测器
  - 根据最近n次分支的执行情况从2^n中选择预测器
- 两级局部预测器(Local predictor):
  - 每个分支有多个2-bit 预测器
  - 根据该分支的最近n次分支的执行情况从2^n中选择预测器
- 竞赛预测器(Tournament predictor):
  - 结合关联预测器和两级局部预测器

竞赛预测器

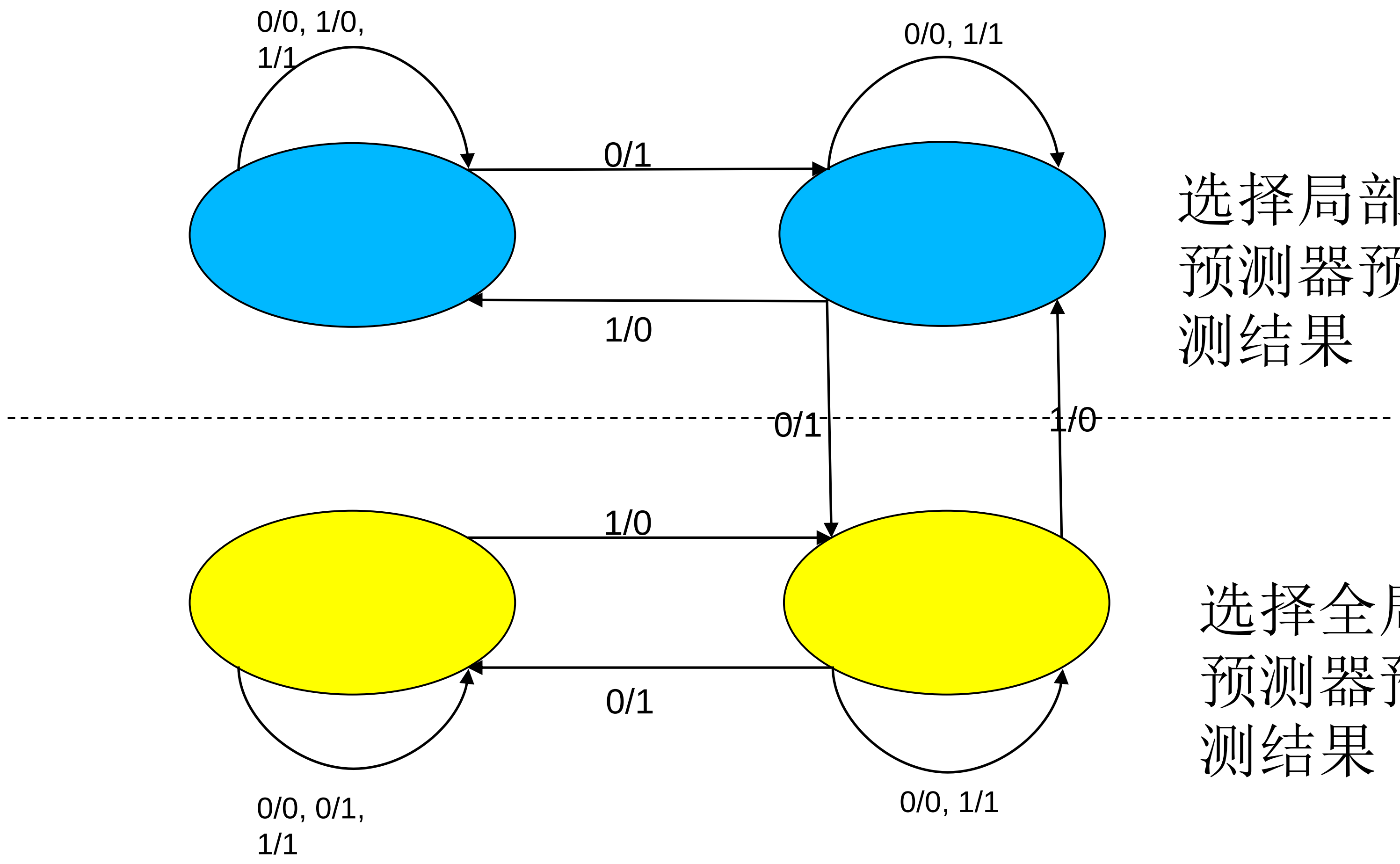


- 全局预测器
  - 使用最近n次分支跳转情况来索引，即全局预测器入口数：2^n每个入口是一个标准的2位预测器
- 局部预测器：两层
  - 一个局部历史记录,使用指令地址的低m位进行索引，每个入口k位，分别对应这个入口最近的k次分支，即最近k次分支的 跳转情况
  - 从局部历史记录选择出的入口对一个2^K的入口表进行索引，这些入口由2位计数器构成，以提供本地预测。
- 选择器：
  - 使用分支局部地址的低m位分支局部地址索引，每个索引得到一个两位计数器，用来选择使用局部预测器还是使用全局预测器的预测结果。
  - 在设计时默认使用局部预测器，当两个预测器都正确或都不正确时，不改变计数器；当全局预测器正确而局部预测器预测错误时，计数器加1，否则减1。

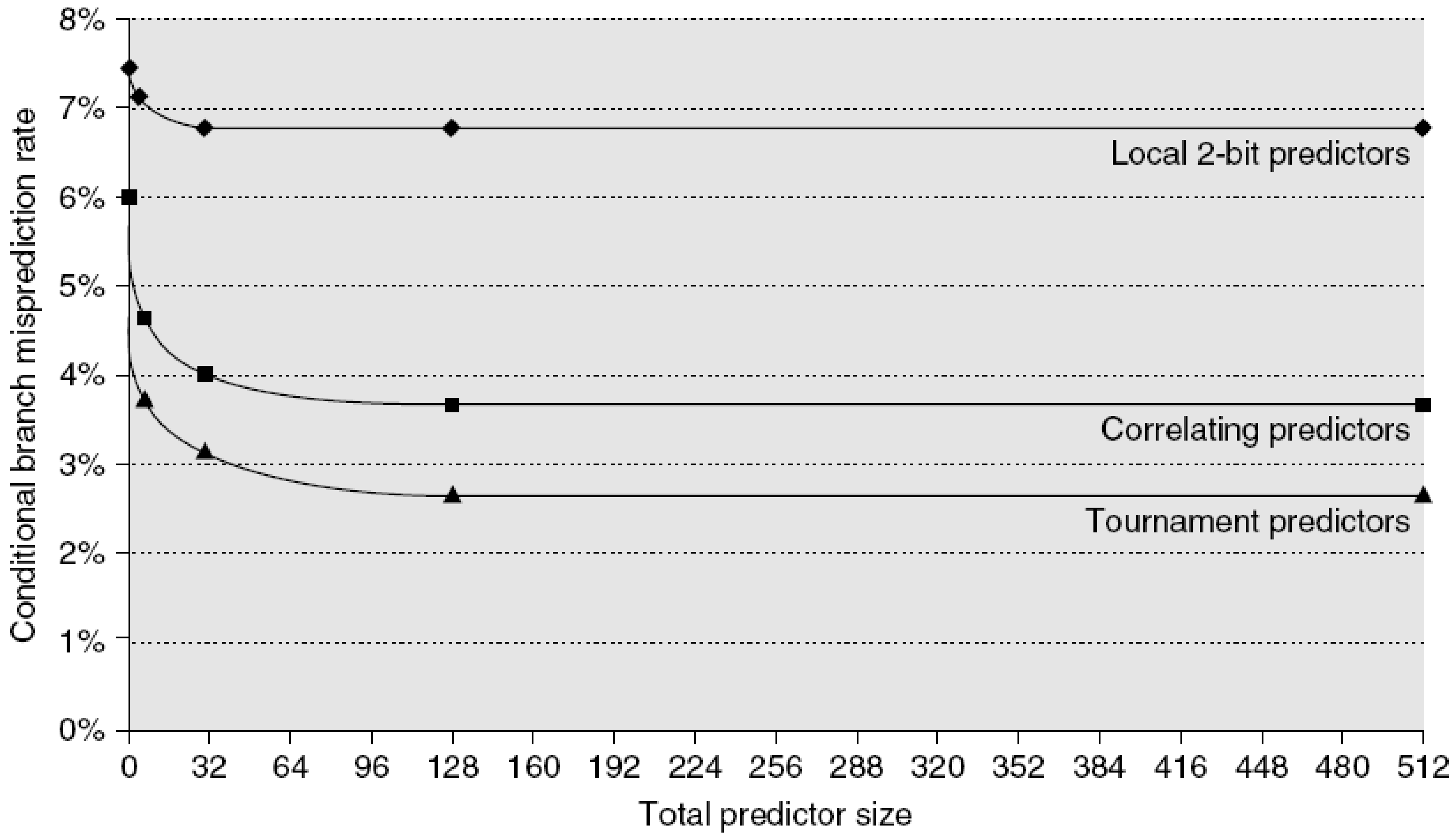
Alpha 21264


4096 entries
12 bits
Hash
2bits
全局预测器
1024 entries
3 bits
1024 entries
10 bits
Hash
PC
局部预测器
PC
2bits
选择器
4096 entries
Selector
NT or T
Alpha 21264 竞赛预测器

选择器状态转移图


0/0, 1/0,
1/1
0/0, 1/1
0/1
1/0
选择局部预测器预测结果
0/1
1/0
1/0
0/1
选择全局预测器预测结果
0/0, 0/1,
1/1
0/0, 1/1

Branch Prediction Performance



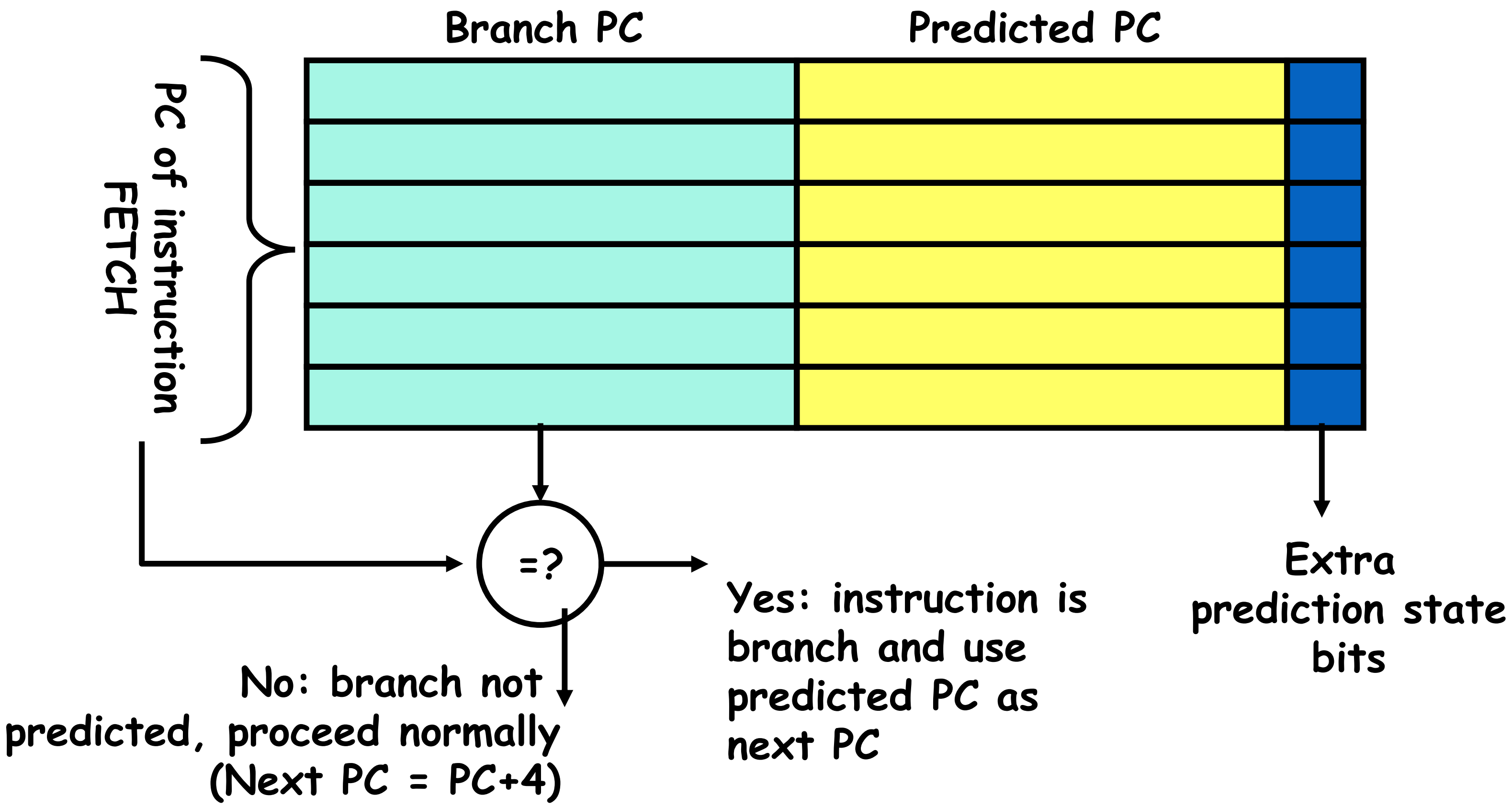
Branch predictor performance

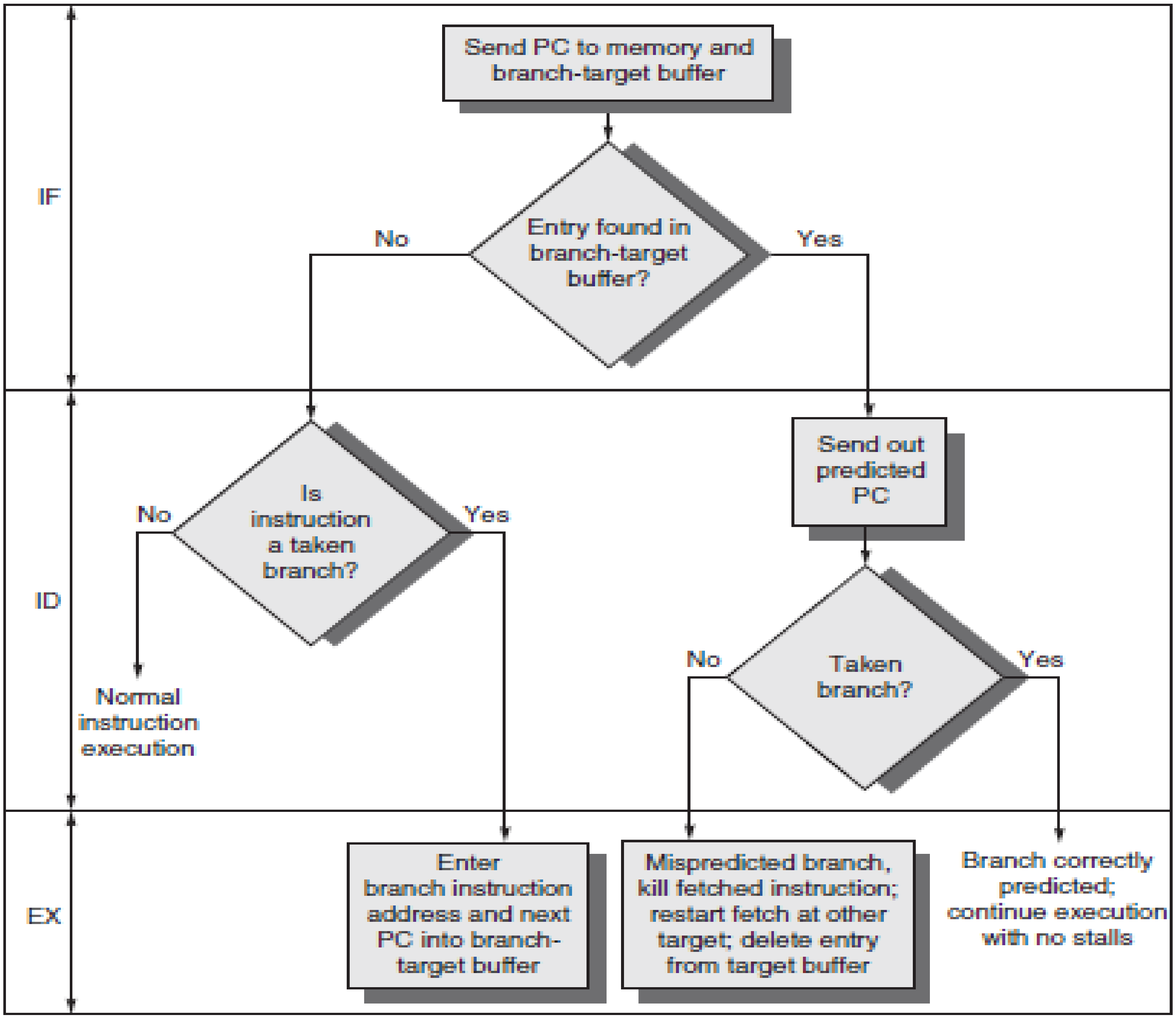
基于BHT表的预测器 review

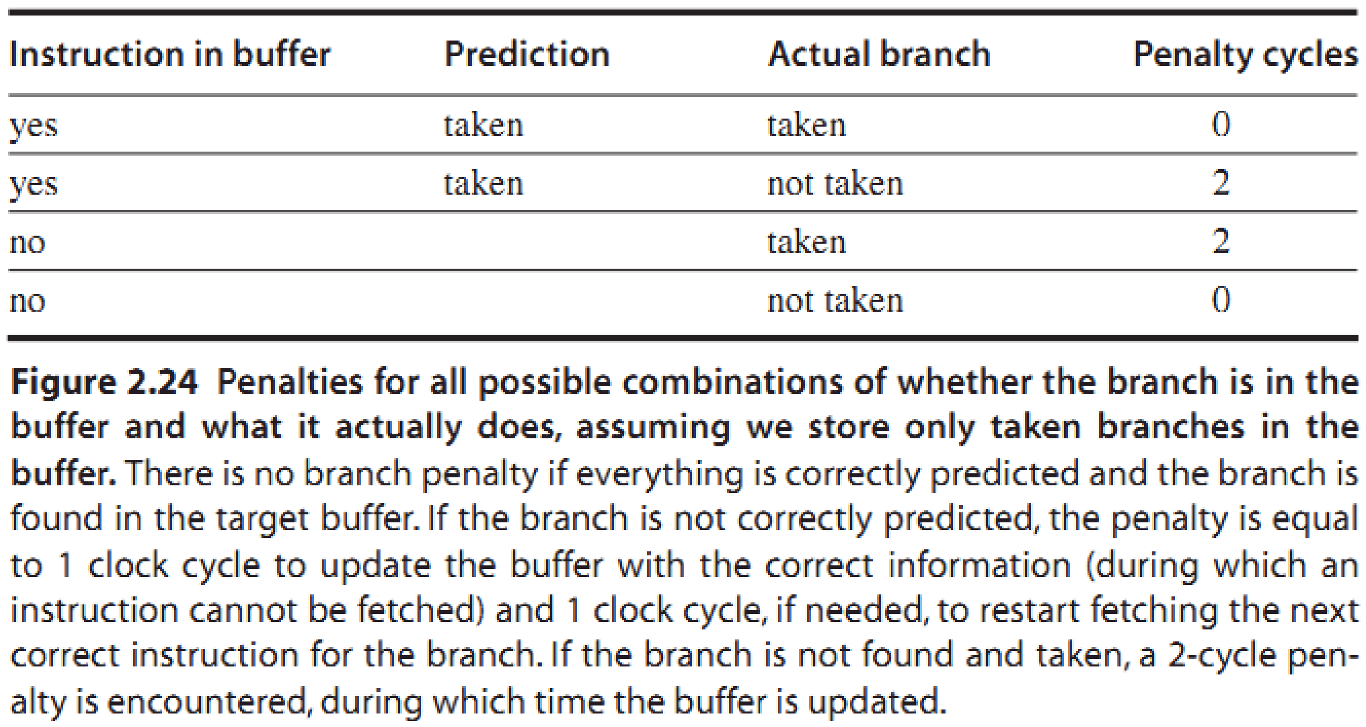
- 基于BHT表的预测器:
  - Basic 2-bit predictor:
  - Correlating predictor:
    - 每个分支对应多个m-bit预测器
    - 最近n次的分支转移的每一种情况分别对应其中一个预测器
  - Local predictor:
    - 每个分支对应多个m-bit预测器
    - 该分支最近n次分支转移的每一种情况分别对应其中一个预测器
  - Tournament predictor:
    - 从多种预测器的预测结果中选择合适的预测结果。
    - 例如：Combine correlating predictor with local predictor

简单的动态预测：Branch Target Buffer (BTB)

- 分支指令的地址作为BTB的索引，以得到分支预测地址
  - 必须检测分支指令的地址是否匹配，以免用错误的分支地址
  - 从表中得到预测地址
  - 分支方向确定后，更新预测的PC


Branch PC
Predicted PC
PC of instruction FETCH
=?
Yes: instruction is branch and use predicted PC as next PC
No: branch not predicted, proceed normally (Next PC = PC+4)
Extra prediction state bits


IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls



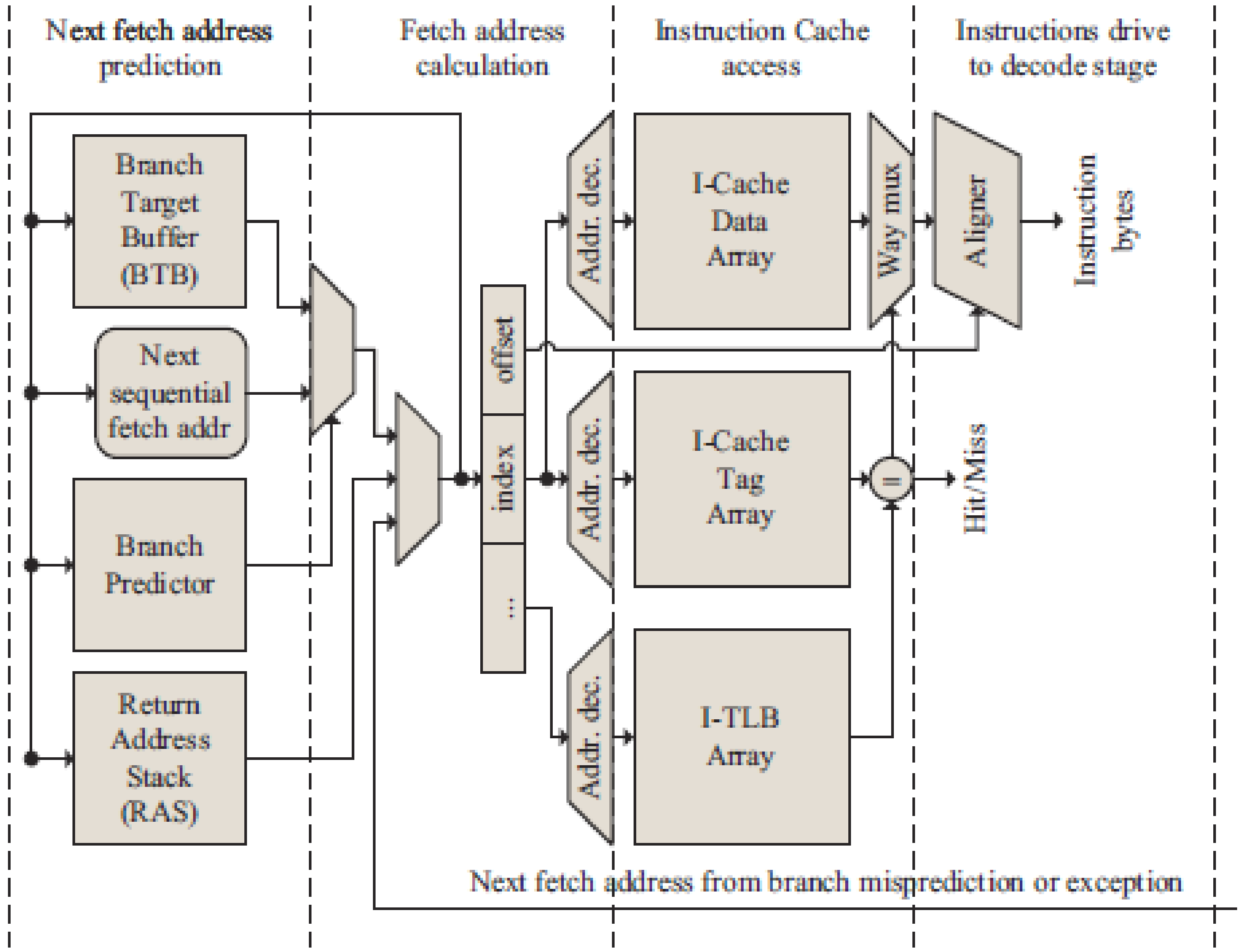
Instruction in buffer	Prediction	Actual branch	Penalty cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

Figure 2.24 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to 1 clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and 1 clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a 2-cycle penalty is encountered, during which time the buffer is updated.

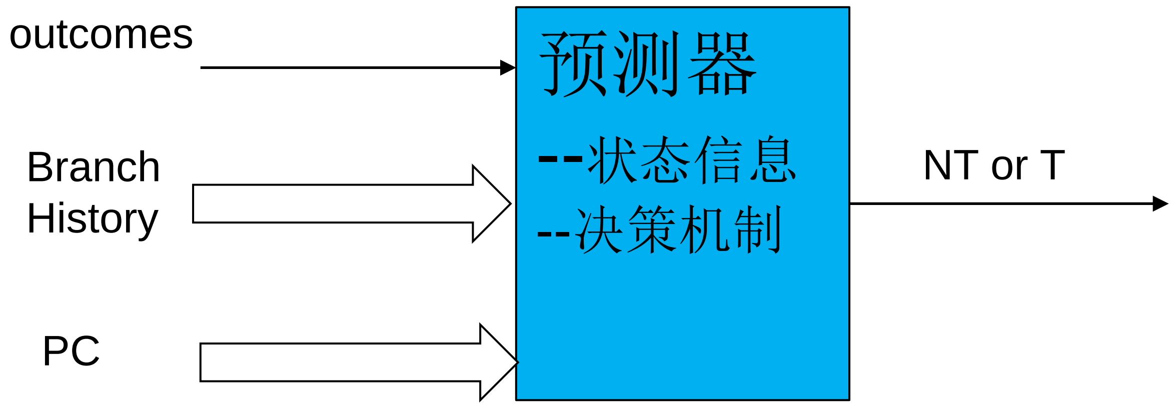
Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from Figure 2.24. Make the following assumptions about the prediction accuracy and hit rate:

- Prediction accuracy is 90% (for instructions in the buffer).
- Hit rate in the buffer is 90% (for branches predicted taken).

Instruction Fetch Unit


Next fetch address prediction
Fetch address calculation
Instruction Cache access
Instructions drive to decode stage
Branch Target Buffer (BTB)
Next sequential fetch addr
Branch Predictor
Return Address Stack (RAS)
offset
index
...
Addr. dec.
Addr. dec.
Addr. dec.
I-Cache Data Array
I-Cache Tag Array
I-TLB Array
Way mux
Aligner
Instruction bytes
=
Hit/Miss
Next fetch address from branch misprediction or exception

预测器的基本结构及输入输出



- 使用FSM来完成决策
  - 根据Branch History和PC来选择状态
  - 由状态决定输出
- 根据实际结果修改状态

下一节内容

- ILP（指令级并行性）的局限性
- 并发多线程处理器
  - 用ILP开发TLP（线程级并行性）

From : H&P Computer Architecture: A Quantitative Approach, Fifth Edition, （5th edition)

谢　谢！