分支预测与推测执行

主讲教师：胡 淼

中山大学计算机学院

2024 年 秋季

- 第 1 章 绪论
- 第 2 章 基准评测集
- 第 3 章 并行计算机的体系结构
- **<u>第 4 章 高性能处理器的并行计算技术</u>**
- 第 5 章 并行计算机的存储层次
- 第 6 章 并行计算机的互连网络
- 第 7 章 异构计算体系结构
- 第 8 章 领域专用体系结构

第 4 章 高性能处理器的并行计算技术

4.1 指令级并行

part2 分支预测与推测执行

参考资料

- Computer Architecture, A Quantitative Approach, 6th Edition.
  - Appendix C. “Pipelining: Basic and Intermediate Concepts”
  - Chapter 3. “Instruction-Level Parallelism and Its Exploitation”

内容纲要

- 分支预测
- 推测执行

为什么需要分支预测

- 典型的MIPS程序中，每3-6条指令出现一次分支
  - 动态调度的窗口太小
- 迄今为止讲的调度方法都没有涉及分支
  - 循环展开
  - 计分板
  - Tomasulo算法
- 指令级并行发掘的越多，分支带来的性能损失越大

Branch Hazards

- Control hazards can cause a greater performance loss than data hazards
  - 10% - 30% performance degression
  - ***cycle stall* in the pipeline**
    - Example: the instruction after the branch is fetched, but the instruction is ignored
      - the fetch is restarted once the branch target is known
      - It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant

Branch Hazards

- Reducing Pipeline Branch Penalties
  - The simplest scheme to handle branches is to **freeze** (or flush) the pipeline
  - **holding or deleting any instructions** after the branch until the branch destination is **known**
    - simplicity both for hardware and software
    - It is the solution used earlier in the pipeline
  - In this case, the *branch penalty* is fixed and cannot be reduced by software

Branch Hazards

- Reducing Pipeline Branch Penalties
  - A higher-performance, and only slightly more complex, scheme is to **treat every branch as** **not taken**
    - simply allowing the hardware to continue
    - as if the branch were not executed
  - having to know **when** the state might be changed by an instruction and **how to “back out” such a change**？

Branch Hazards

- Reducing Pipeline Branch Penalties

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target $+1$				IF	ID	EX	MEM	WB	
Branch target $+2$					IF	ID	EX	MEM	WB

Figure C.10 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

Branch Hazards

- An alternative scheme is to **treat every branch as** **taken**
- This buys us a one-cycle improvement when the branch is actually taken
  - because we know the target address at the end of ID
  - one cycle before we know whether the branch condition is satisfied in the ALU stage

Delayed Branch

- A fourth scheme, which was heavily used in early RISC processors is called **delayed branch**
  - In a delayed branch, the execution cycle with a branch delay of one is:

branch instruction
sequential successor
branch target if taken

Executed whether or not the branch is taken

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.11 The behavior of a delayed branch is the same whether or not the branch is taken.

- the delay slot is scheduled with an independent instruction
- architectures with delay branches often disallow putting a branch in the delay slot
    - When the instruction in the branch delay slot is also a branch, the meaning is unclear

延迟分支的缺点

- 延迟分支需要重新定义架构
- 延迟分支会导致轻微的代码扩展
- 中断处理变得更加困难
  - 因为延迟槽中的指令引起的中断请求必须与“正常”指令引起的中断请求不同地处理
- 需要额外的硬件来实现延迟分支

Reducing the Cost of Branches Through Prediction

- As pipelines get deeper and the potential penalty of branches increases, **using delayed branches and similar schemes becomes insufficient**
- Instead, we need to turn to more aggressive means for predicting branches
- Such schemes fall into two classes:
  - **Static branch prediction**: low-cost static schemes that rely on information available at compile time
  - **Dynamic branch prediction**: strategies that predict branches dynamically based on program behavior

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Static Branch Prediction

- the behavior of branches is often *bimodally distributed*
  - Means that an individual branch is often *highly* ***biased*** *toward taken or untaken*

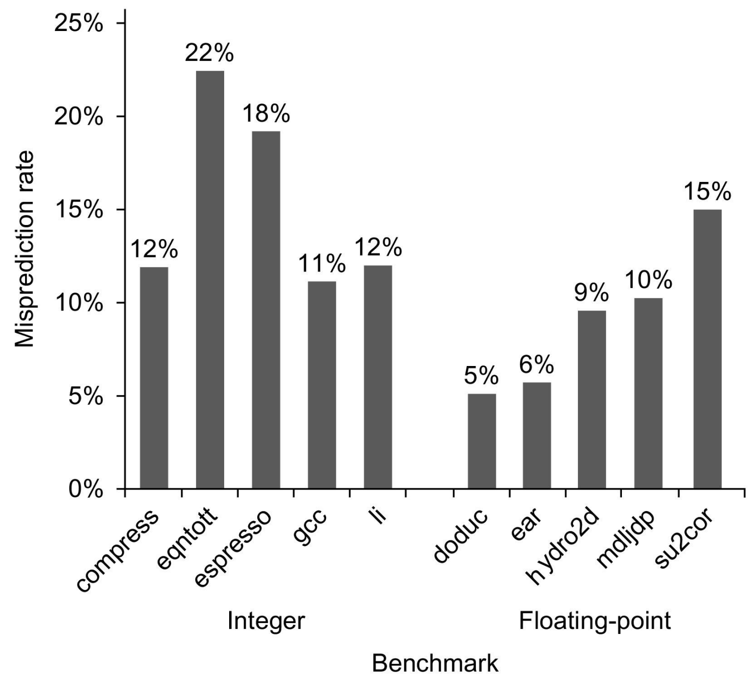
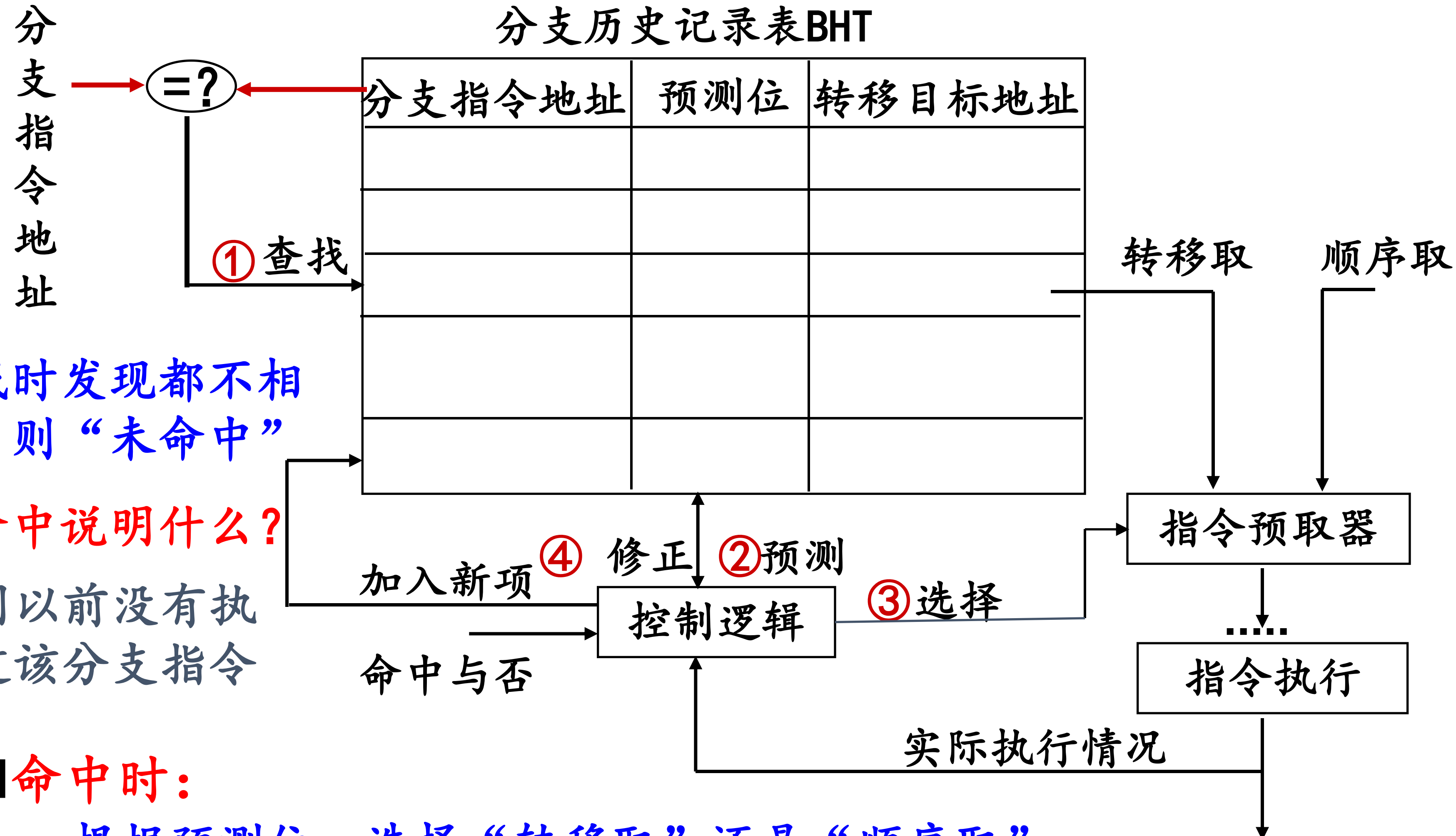


Figure C.14 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.

动态分支预测的基本原理

- 基本思想：
  - 利用最近转移发生的情况，来预测下一次可能发生的转移
  - 预测后，在实际发生时验证并调整预测
  - 转移发生的历史情况记录在BHT中(有多个不同的名称)
    - 分支历史记录表BHT(Branch History Table)
      - 对于SPEC89测试程序而言，具有4K的BHT的预测准确率为82%~99%，可存放在指令cache或者专门硬件实现
    - 分支预测缓冲器BPB(Branch Prediction Buffer)
    - 分支目标缓冲器BTB(Branch Target Buffer)
  - 每个表项由分支指令地址低位作索引，故在IF阶段就可以取到预测位
    - 低位地址相同的分支指令共享一个表项，可能存在冲突
    - 由于仅用于预测，所以不影响执行结果

动态分支预测的基本原理



查找时发现都不相等，则“未命中”

未命中说明什么？

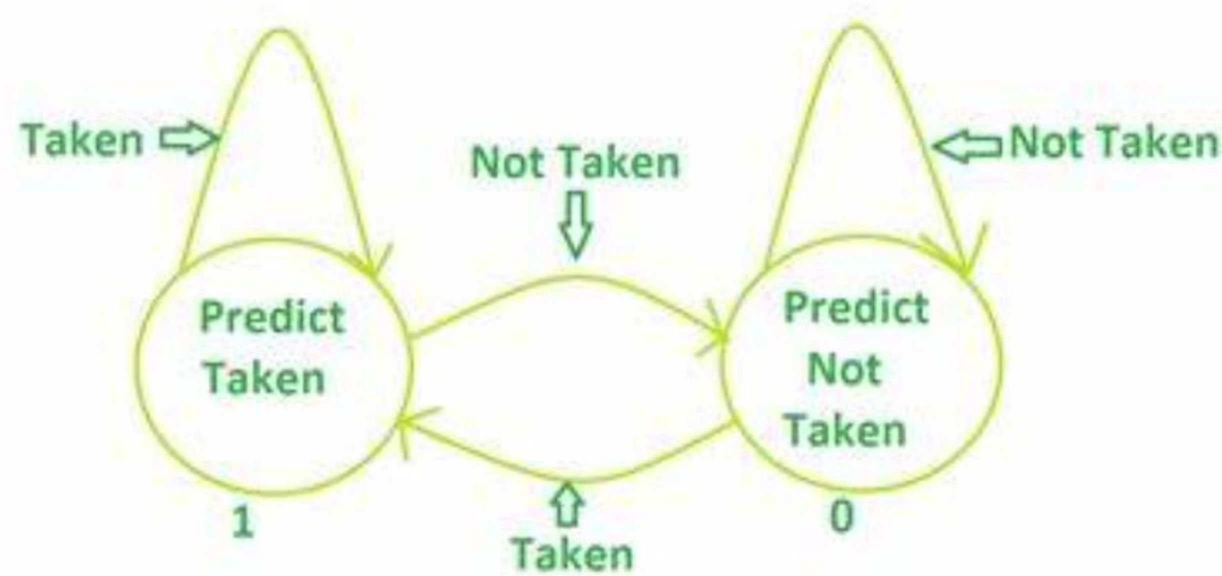
说明以前没有执行过该分支指令

- **命中时：**
  根据预测位，选择“转移取”还是“顺序取”
- **未命中时：**
  加入新项，并填入指令地址和转移目标地址、初始化预测位

中山大學 SUN YAT-SEN UNIVERSITY

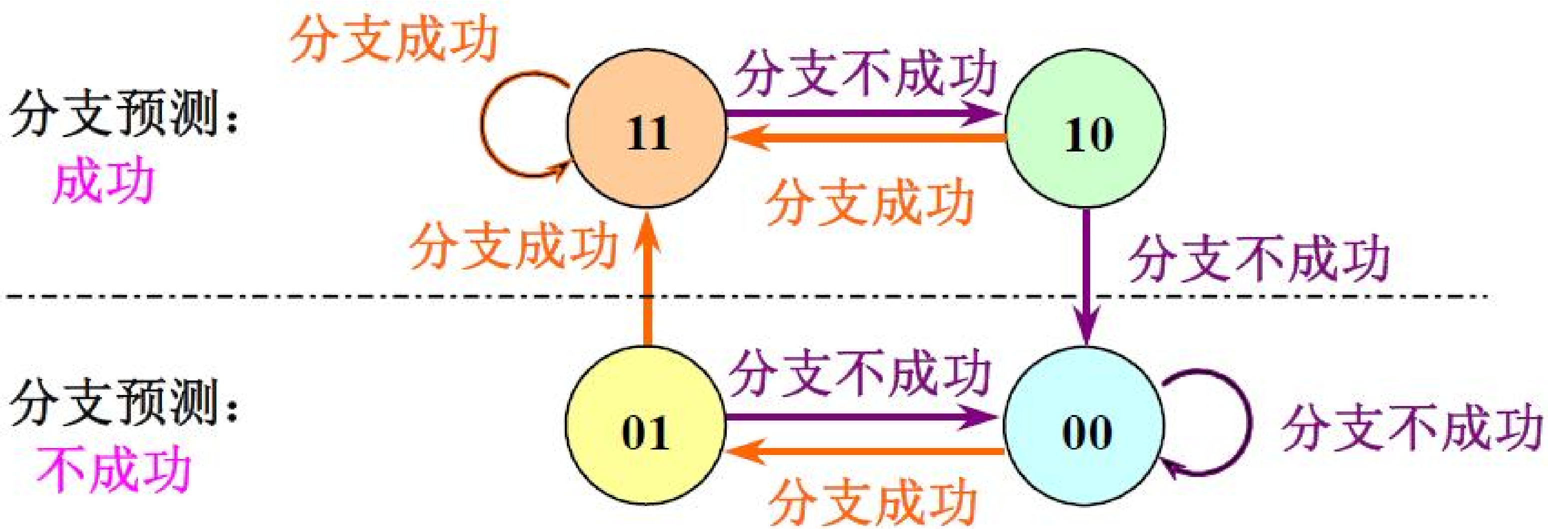
简单的1位预测器

- 只有1个预测位的分支预测缓冲
  - 记录分支指令最近一次的历史，BHT中只需要1位二进制位



简单的2位预测器

- 采用2位来记录历史：与n位的预测器效果差不多
  - 采用有限状态机记录分支是否成功的历史情况
  - 根据状态机的状态做出预测
  - 根据真实分支情况修正预测器



Branch Prediction

- **Basic 2-bit predictor:**
  - The 2-bit predictor use only the recent behavior of a single branch to predict the future behavior of that branch

How about looking at the recent behavior of other branches?

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Correlating Branch Predictors

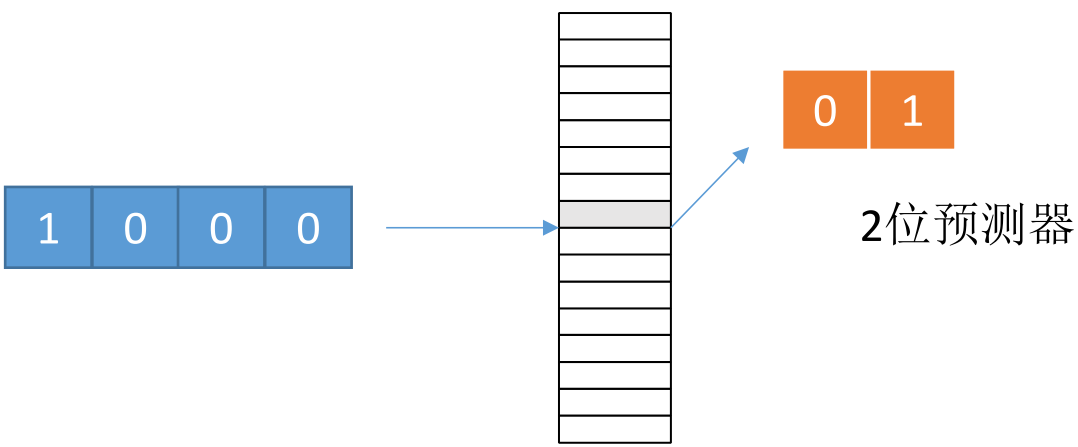
- **Key observation:**
  - the behavior of branch b3 is correlated with the behavior of branches b1 and b2

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

```
     addi  x3,x1,-2
     bnez  x3,L1      //branch b1   (aa!=2)
     add   x1,x0,x0   //aa=0
L1:  addi  x3,x2,-2
     bnez  x3,L2      //branch b2   (bb!=2)
     add   x2,x0,x0   //bb=0
L2:  sub   x3,x1,x2   //x3=aa-bb
     beqz  x3,L3      //branch b3   (aa==bb)
```

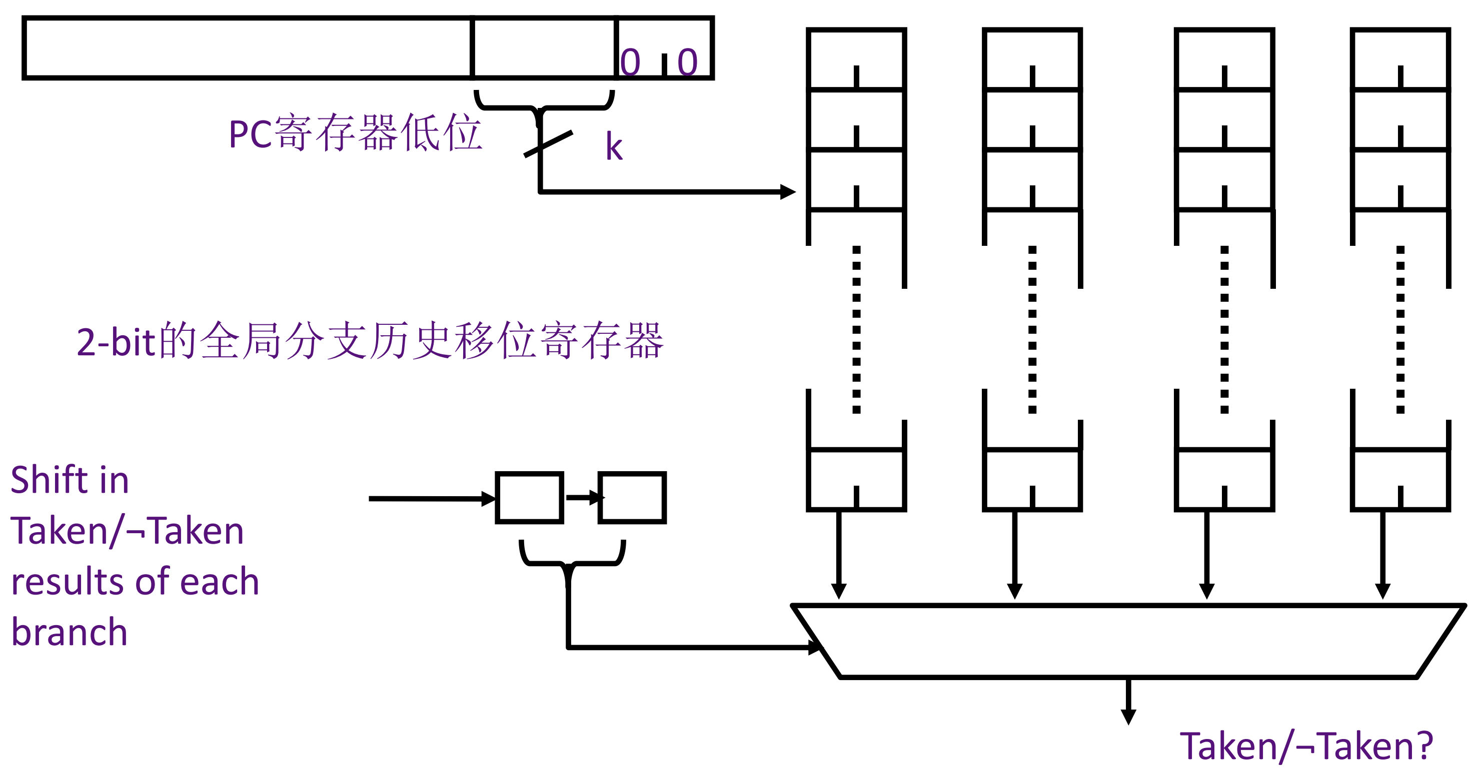
相关预测器（两级预测器）

- 相关预测器——（m，n）相关预测器
  - 用移位寄存器记录最近m个分支的转移情况
    - 转移成功置为1，转移失败置为0
  - 根据这m位可以寻址2^m个预测器
    - 每个预测器n位
  - 分支指令地址低位+寄存器m位，查BPB



相关预测器（两级预测器）

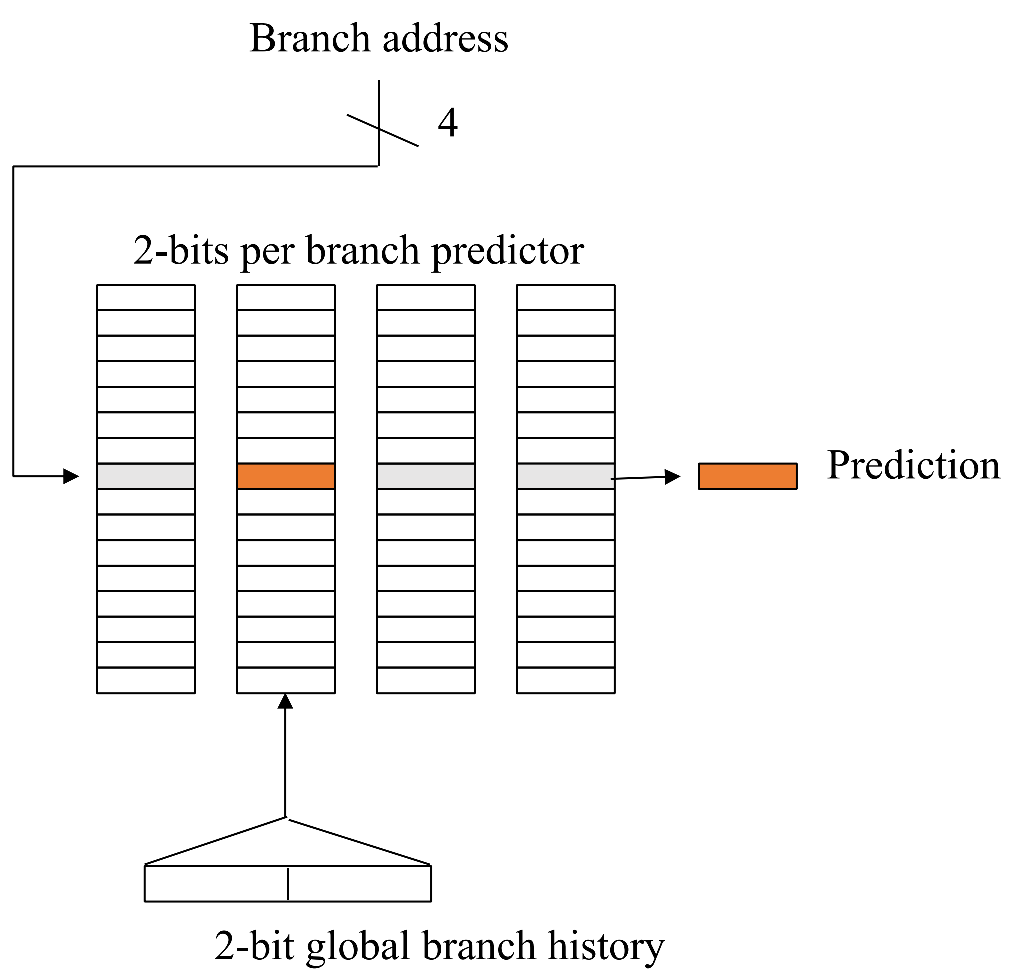
- 两级的（2,2）预测器



Pentium Pro 通过最近的两次分支历史从4个预测结果中选择能达到大约**95%**的准确率

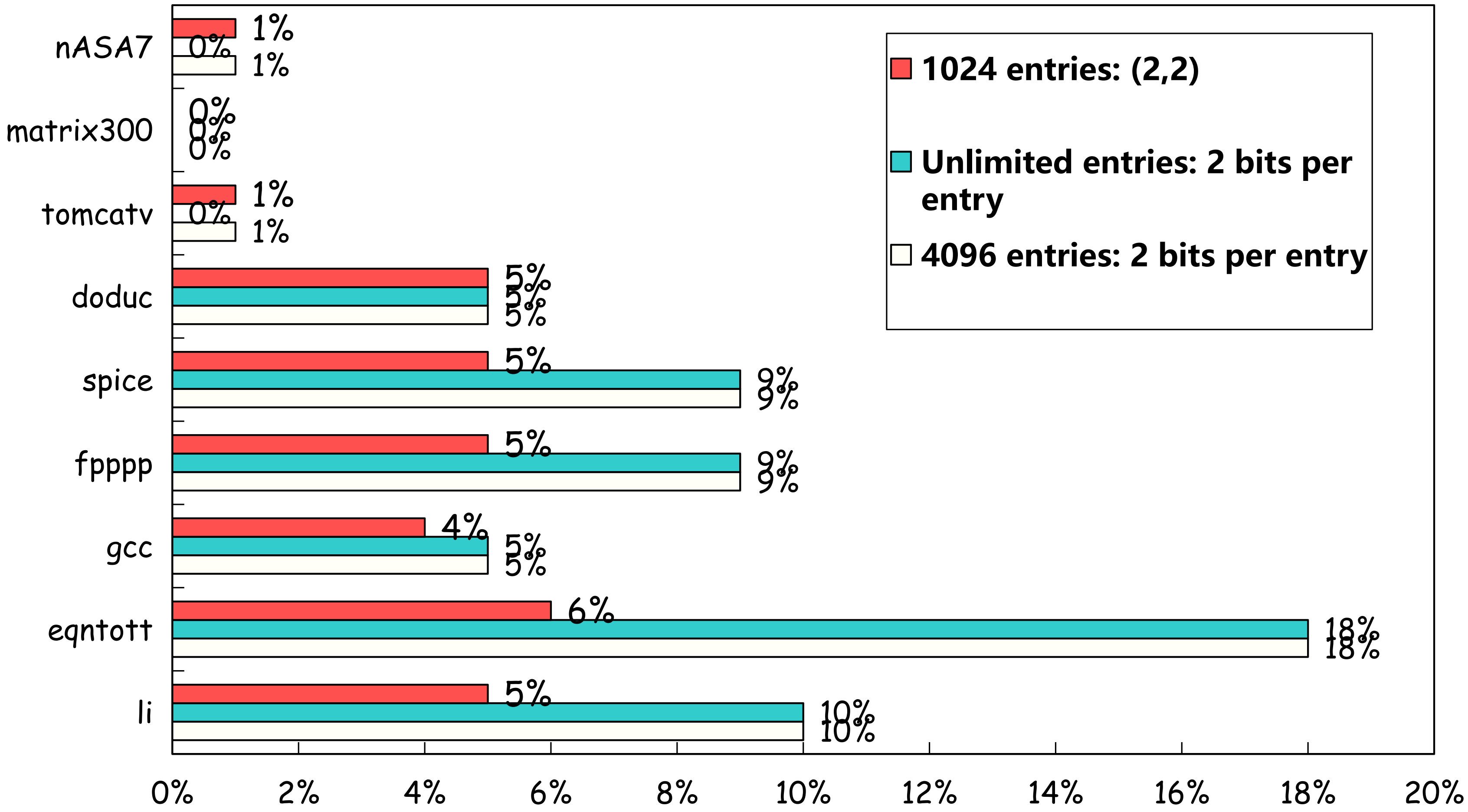
相关预测器（两级预测器）

- 从另一个视角看两级的（2, 2）预测器



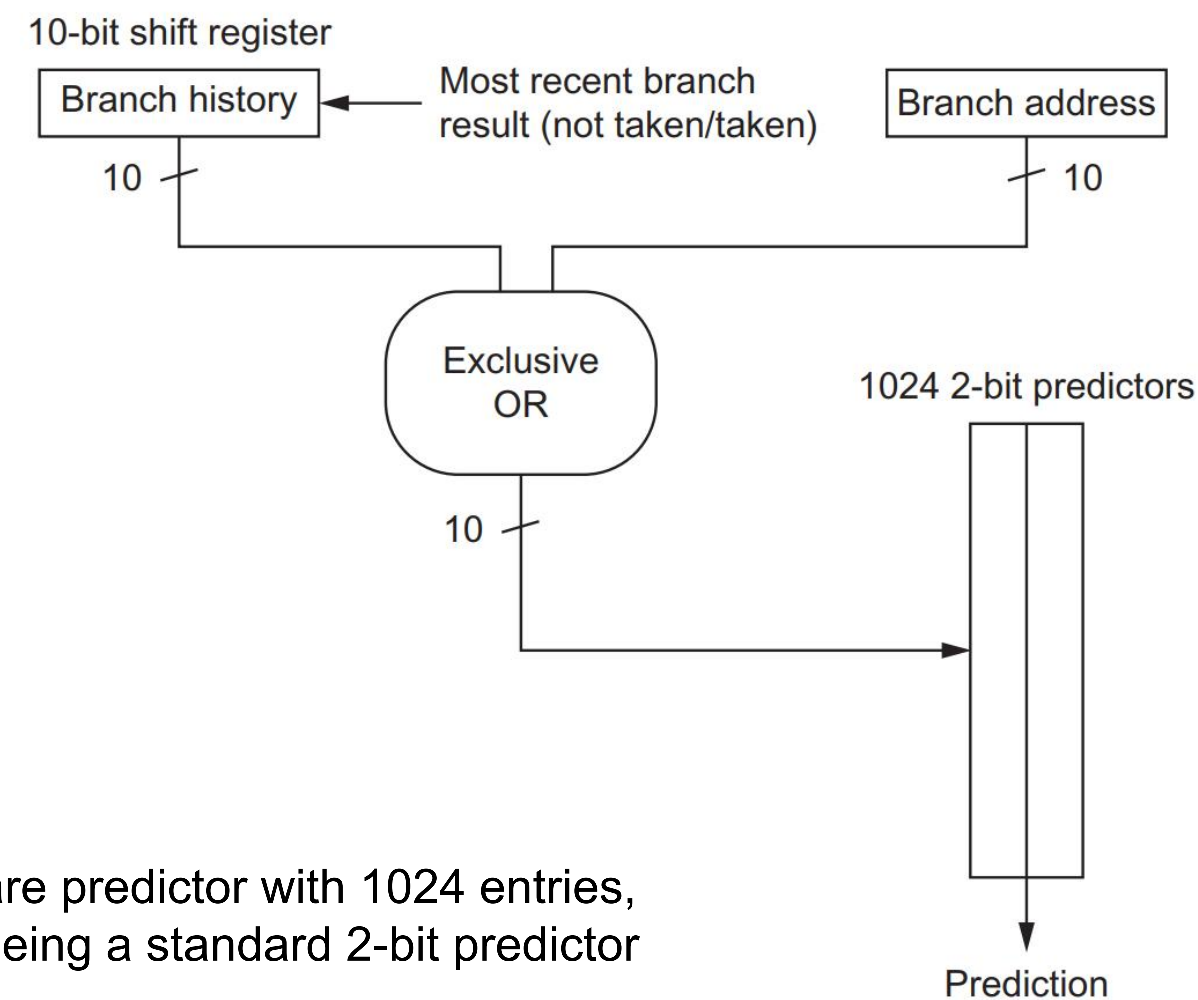
相关预测器（两级预测器）

- （2,2）预测器与简单2位预测器的性能比较（失败率）



中山大學 SUN YAT-SEN UNIVERSITY

gshare predictor

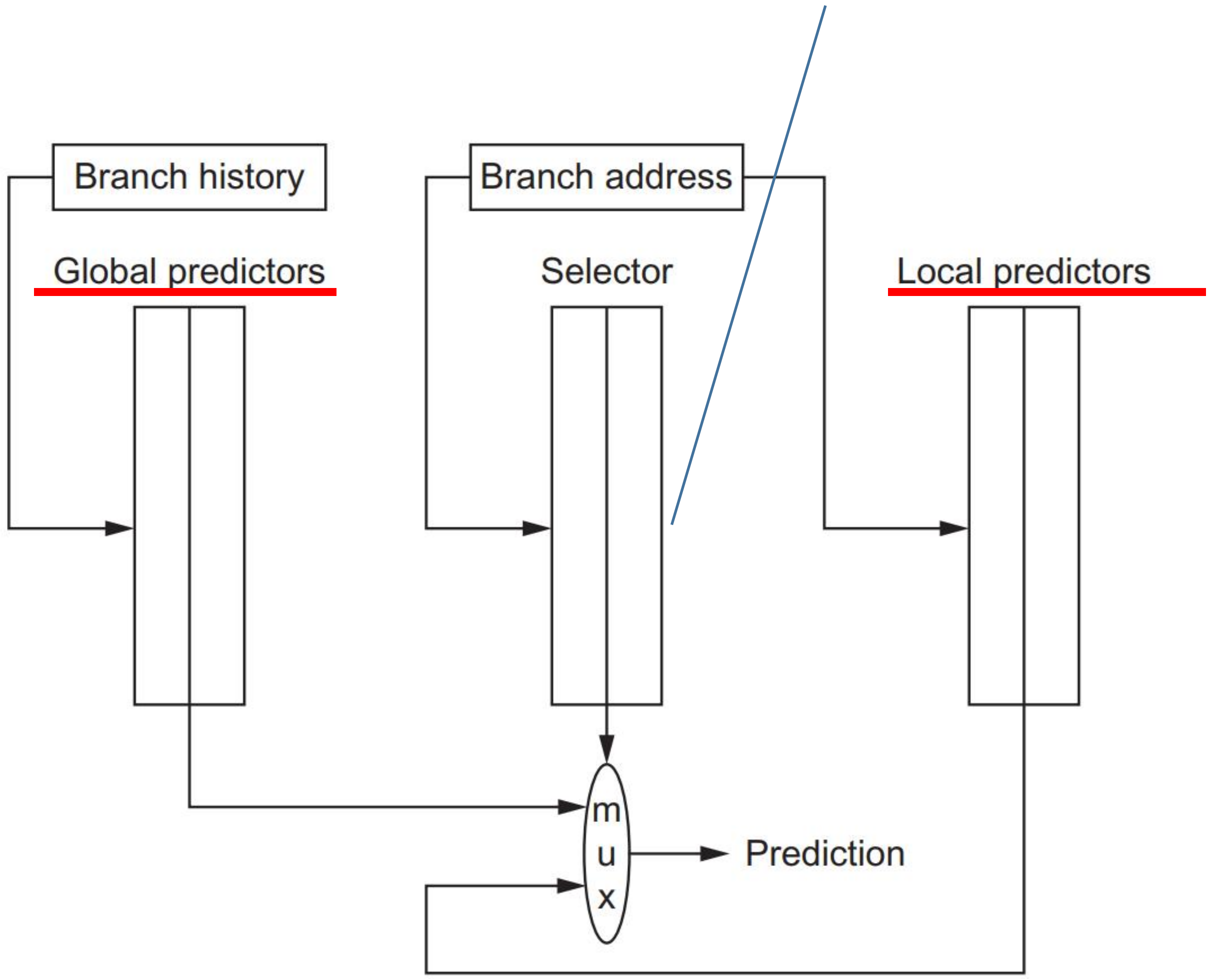


A gshare predictor with 1024 entries, each being a standard 2-bit predictor

Tournament predictor

- **Tournament predictor（竞赛预测器）：**
  - Combine correlating predictor with local predictor
  - A global predictor uses the most recent branch history to index the predictor
  - A local predictor uses the address of the branch as the index

The selector acts like a 2-bit predictor, changing the preferred predictor for a branch address when two mispredicts occur in a row

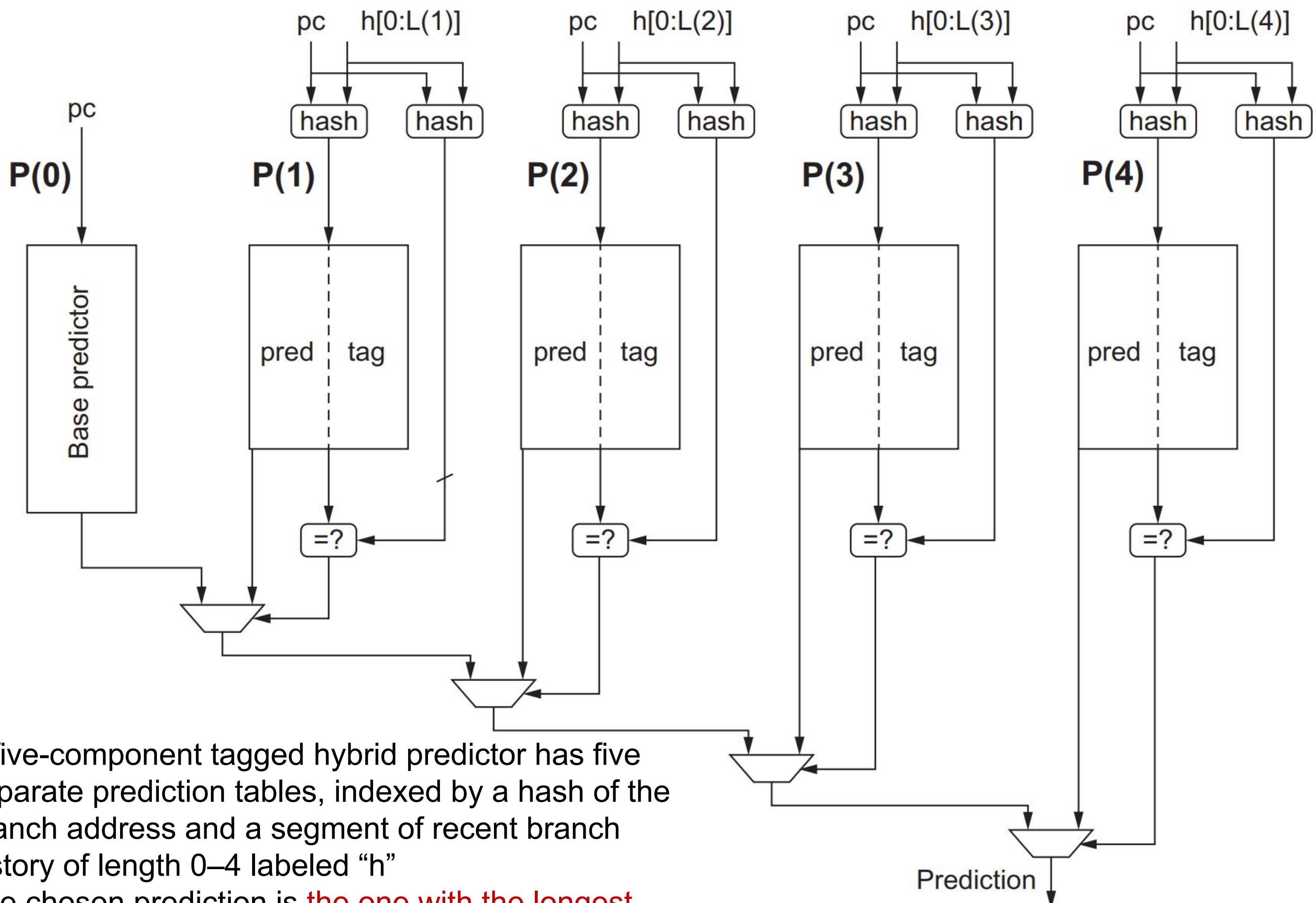


A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor.

Tagged Hybrid Predictors

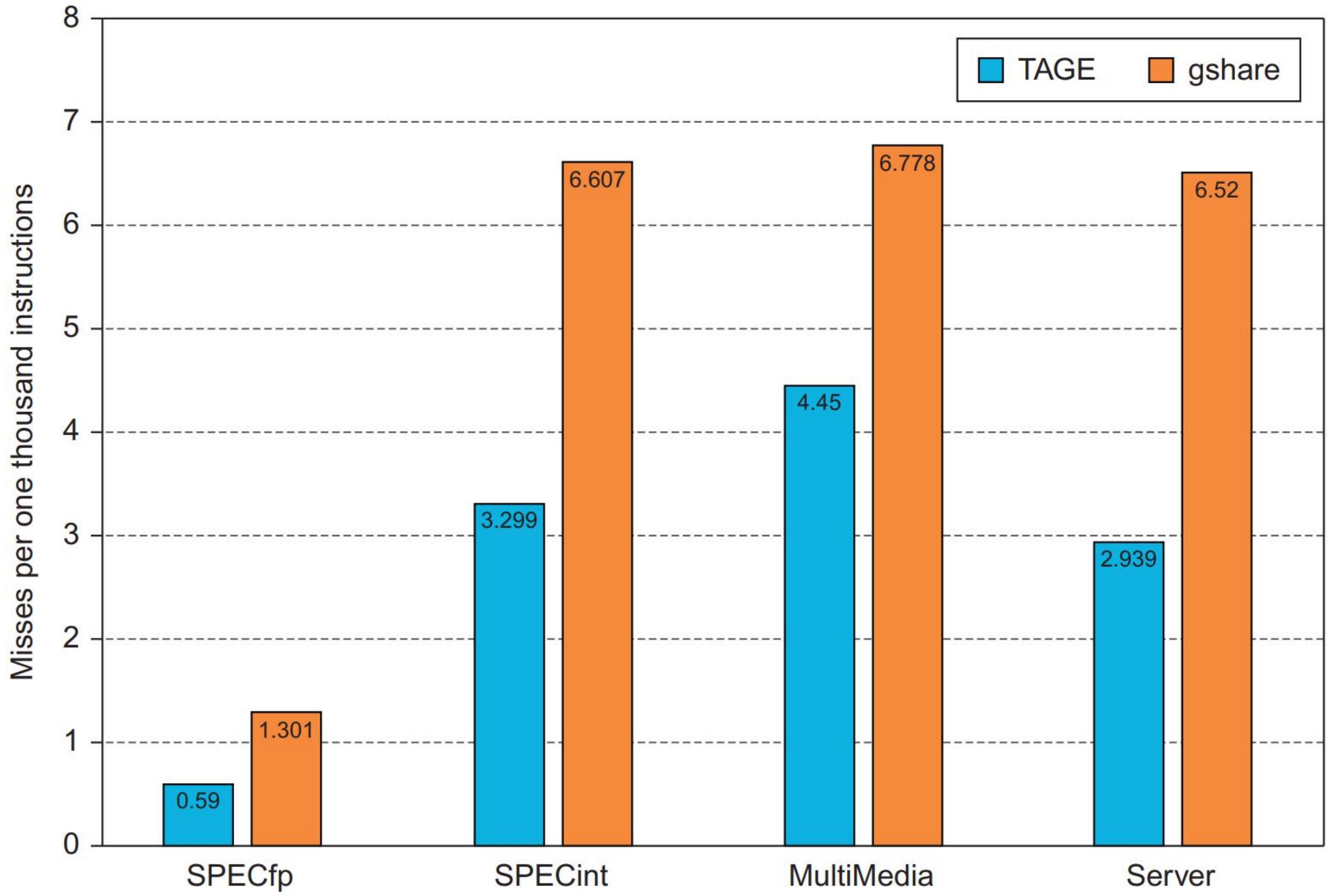
- **Need to have predictor for each branch and history**
  - Problem: **this implies huge tables**
  - **Solution**:
    - Use **hash tables**, whose hash value is based on branch address and branch history
    - Longer histories may lead to increased chance of hash collision, so use multiple tables with increasingly shorter histories

Tagged Hybrid Predictors



- A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h"
- The chosen prediction is the one with the longest history where the tags also match

Tagged Hybrid Predictors

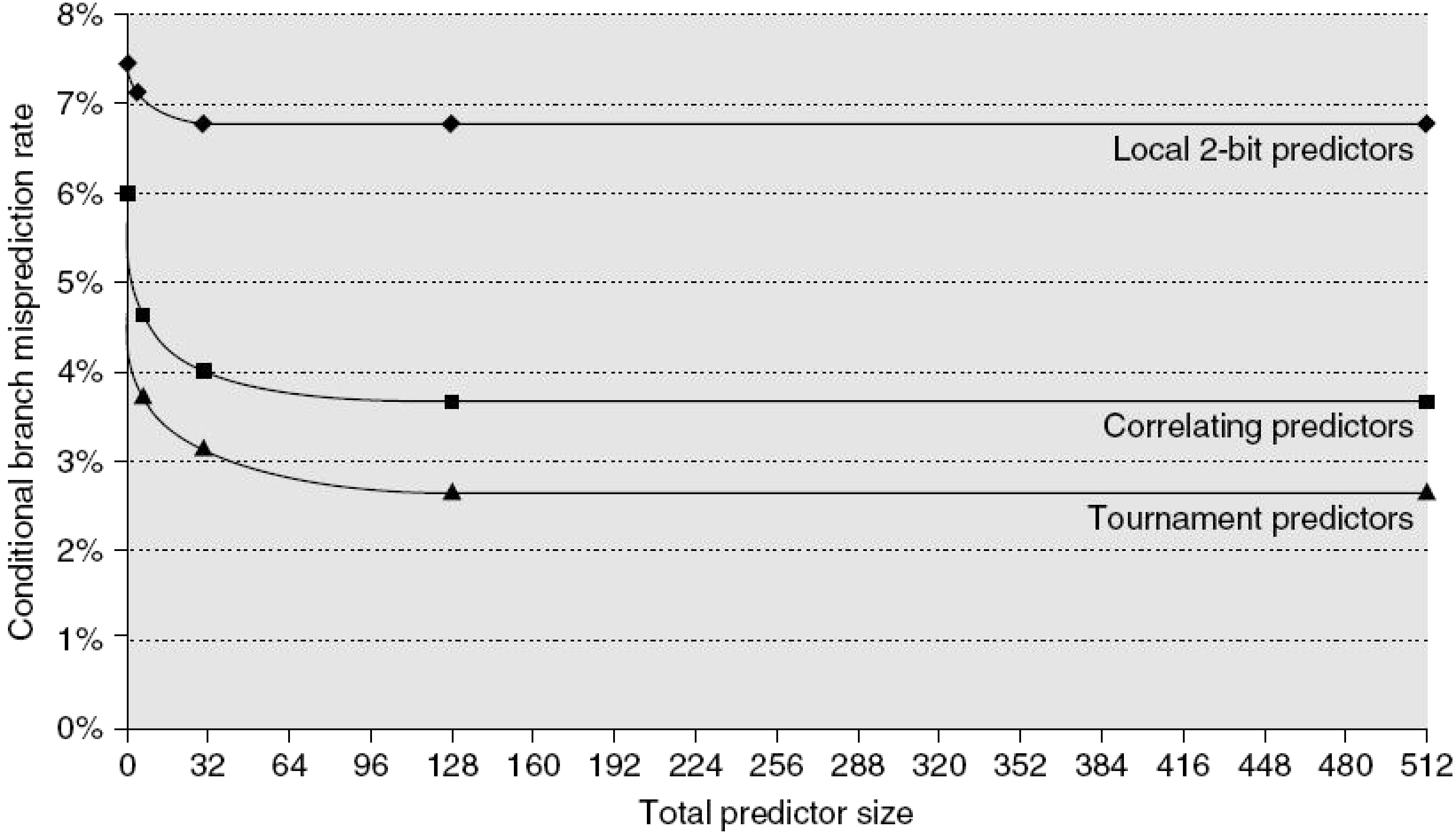


Tagged hybrid predictors (sometimes called TAGE—TAgged Geometic predictors)

分支预测器总结

- 简单的2位预测器
  - 根据2位饱和计数器的值选择Taken和¬Taken
  - 连续错误两次则改变预测结果
- 相关预测器
  - 为每个分支设置2^m个n位的预测器
  - 根据最近全局发生的m次分支从2^m个n位的预测器中选出一个
- 局部预测器
  - 为每个分支设置2^m个2位的预测器
  - 根据最近本身发生的m次分支从2^m个n位的预测器中选出一个
- 竞赛预测器（Tournament Predictor）
  - 在局部预测器和相关预测器之间动态选择
  - 采用一个饱和计数器（如2位计数器），在两者之间选择
    - 计数器值为00、01，选择局部预测器
    - 计数器值为10、11，选择相关预测器

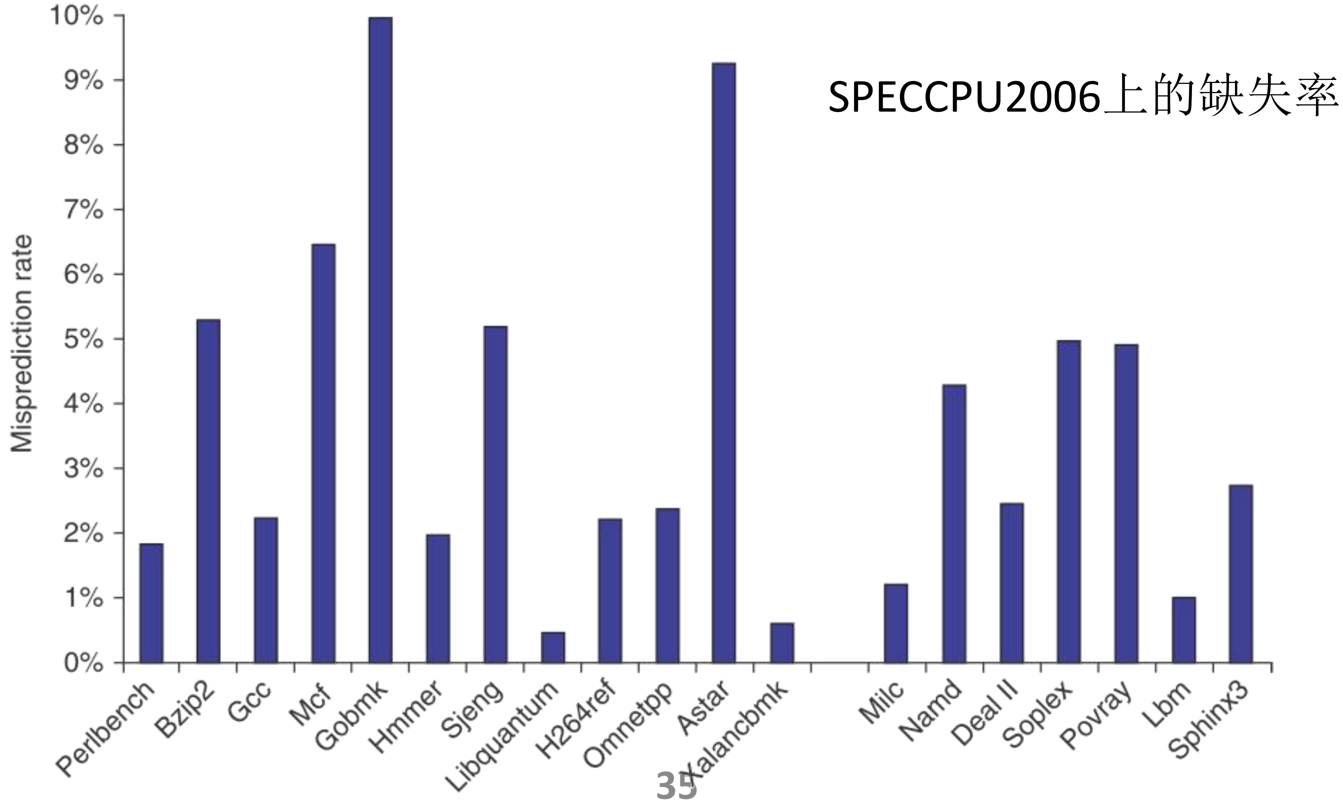
多种预测器性能比较



8%
7%
6%
5%
4%
3%
2%
1%
0%
Conditional branch misprediction rate
Local 2-bit predictors
Correlating predictors
Tournament predictors
0
32
64
96
128
160
192
224
256
288
320
352
384
416
448
480
512
Total predictor size

Intel Core i7的分支预测方法

- 综合三种预测器
  - 简单的2位预测器
  - 基于全局历史的相关预测器
  - 循环跳出（Loop Exit）预测器
    - 当一个分支被判断为一个循环时，用一个计数器记录循环次数

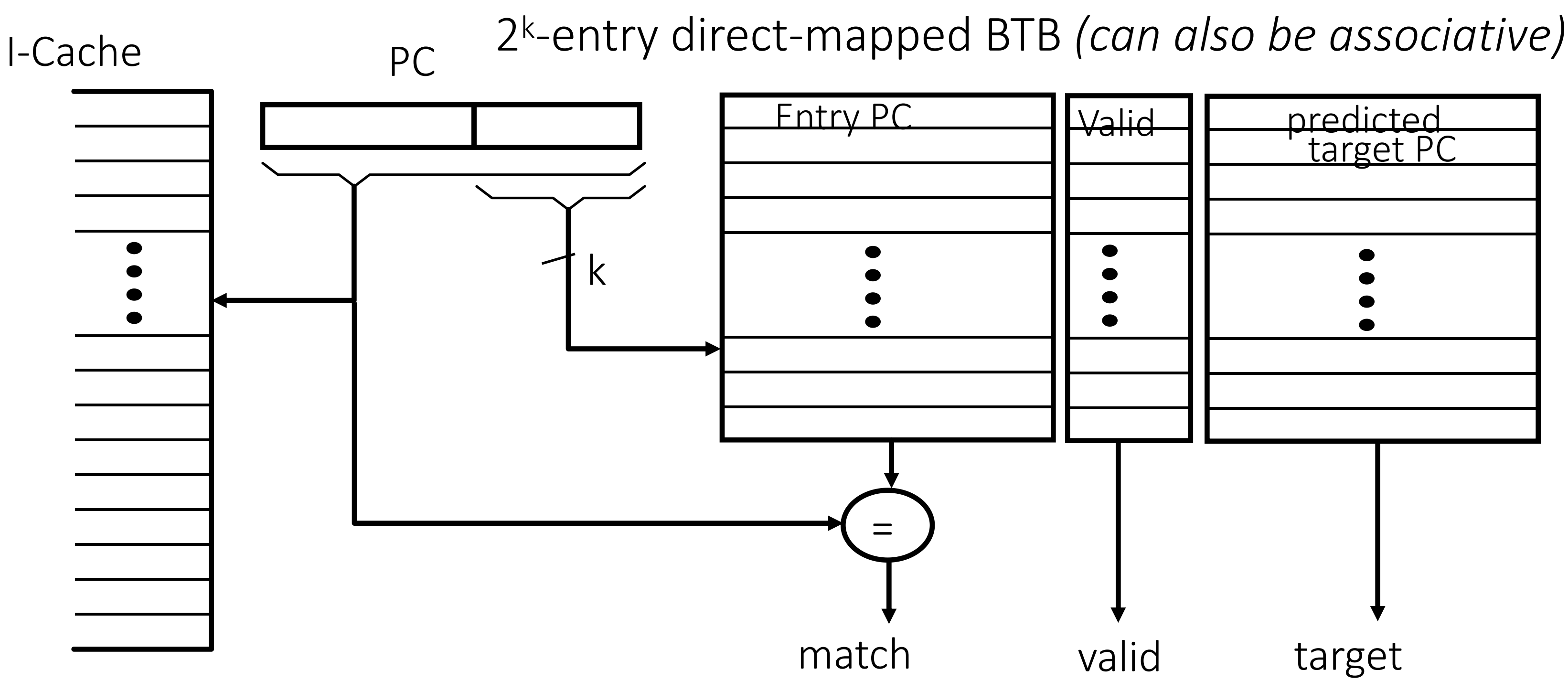


分支目标缓存Branch Target Buffer

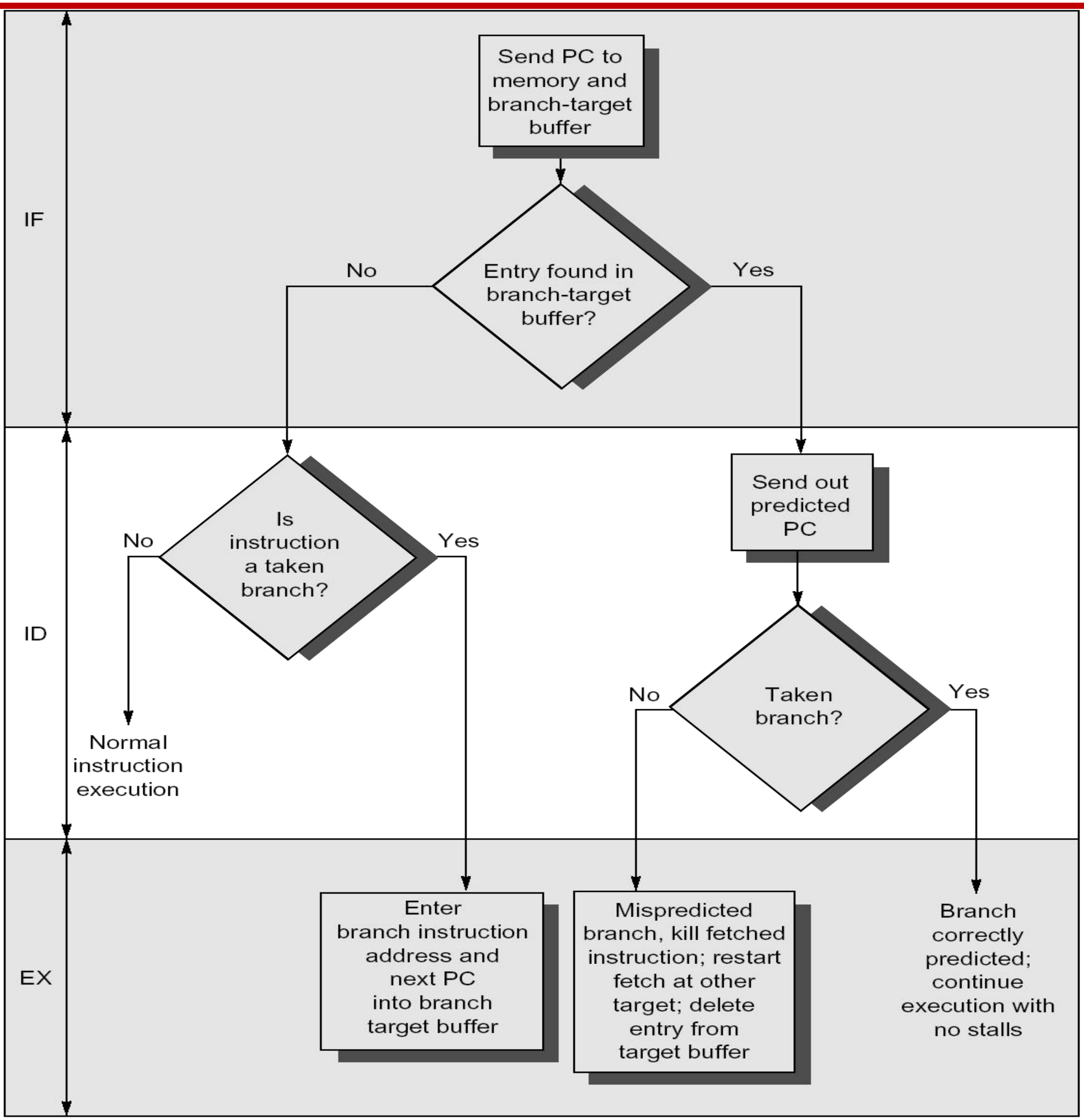
- 目的：在取指阶段（译码之前）就能知道：
  - 本指令是否为分支指令
  - 如果是分支指令，是否发生转移
  - 如果发生转移，转移目标是哪里
- BTB，一个高速缓存，其必要字段包括：
  - 分支指令的PC值保存在BTB中
  - 一个Taken的分支指令的转移地址保存在BTB中
- BTB的工作原理
  - 在取指阶段，将PC与BTB中的条目比较，是否出现当前PC
  - 如果当前PC出现在BTB中，则返回所保存的转移目标地址
  - 返回的转移目标地址用于下一条指令取指

分支目标缓存

- 分支指令的PC和对应转移目标的PC均保存在BTB
- 仅有taken的分支或者无条件跳转指令保存在BTB中
- 如果BTB不命中，则根据PC+4取指
- 在分支指令取指阶段即可知道转移目标



分支目标缓存工作流程


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

分支目标缓存失败的代价

Instruction in BTB	Prediction	Actual branch	Penalty cycle
yes	Taken	Taken	0
yes	Taken	Not Taken	2
no		Taken	2
no		Not Taken	0

中山大學 SUN YAT-SEN UNIVERSITY

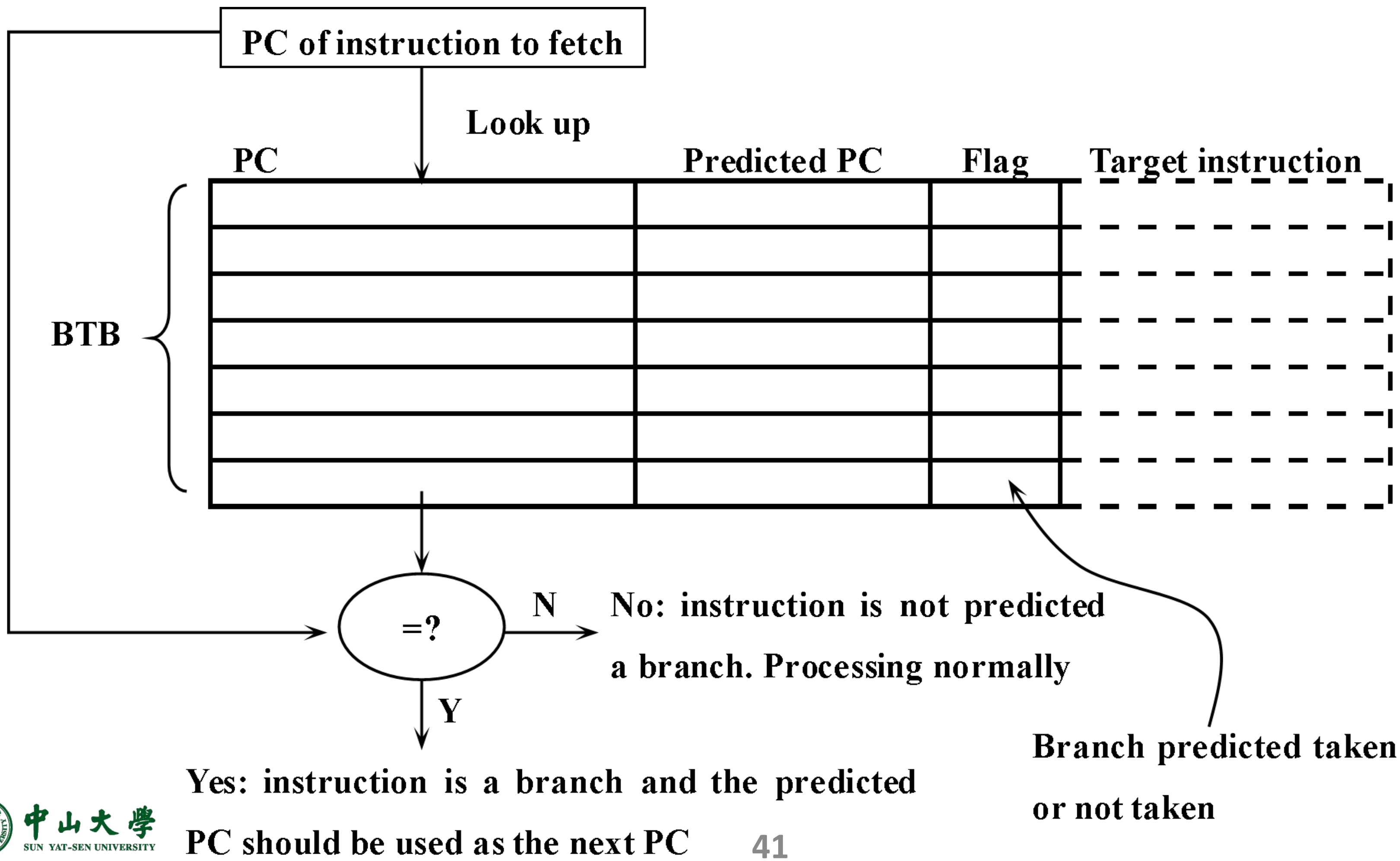
Observation

- The improvement from dynamic branch prediction will grow as the pipeline length, and thus the branch delay grows
- Better predictors will yield a greater performance advantage
- Modern high-performance processors have branch misprediction delays on the order of 15 clock cycles

- **Clearly, accurate prediction is critical!**

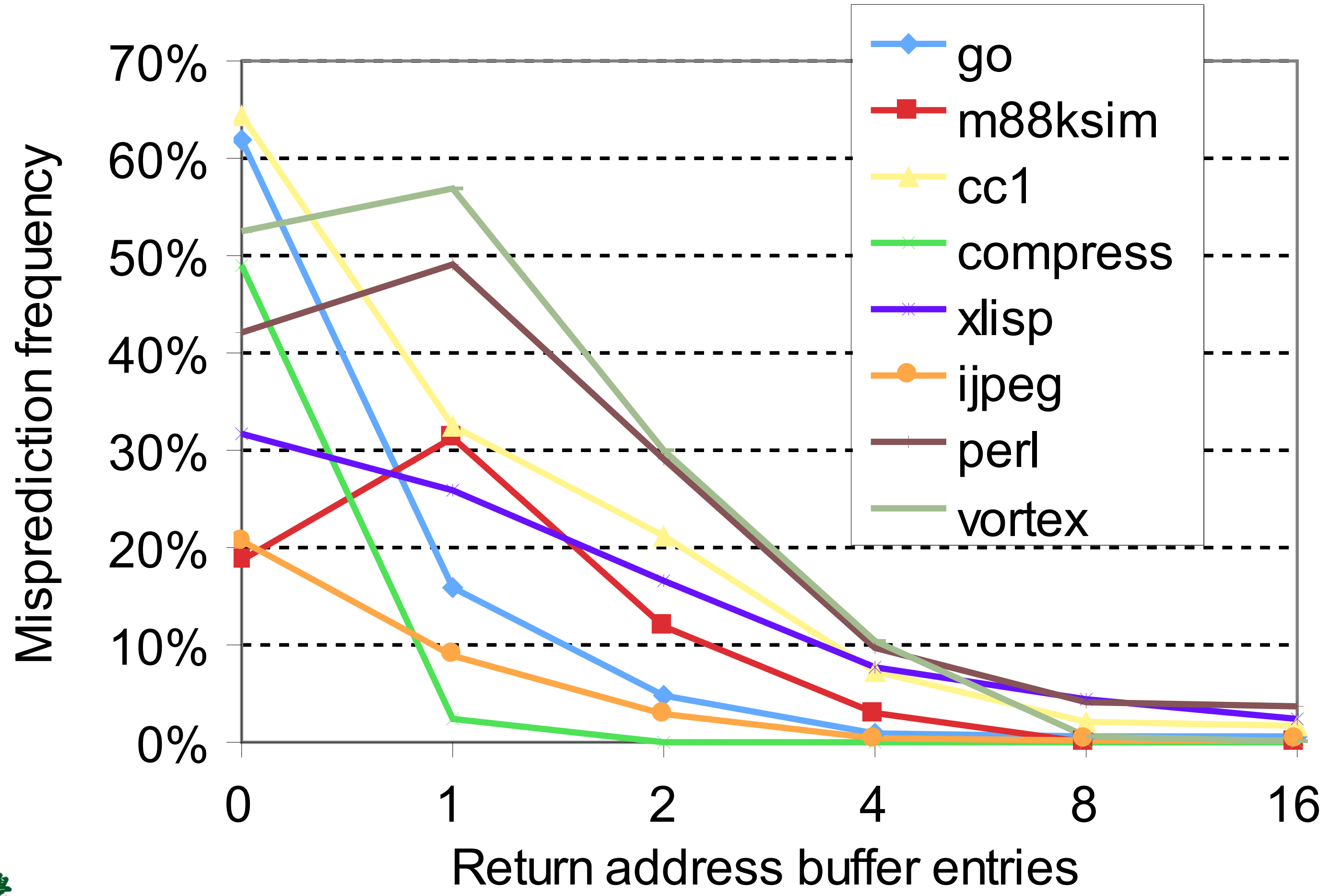
一些小的改进 (Branch Folding)

- 将目标指令也保存在BTB中
  - 在取目标PC的同时，将目标指令也取出来，减少一次访存
    - 可以将BTB做的更大



再来一些小的改进

- 记录返回地址
  - 将Call指令的返回地址（Call指令的下一指令）记录在一个小的栈中



分支预测总结

- BPB/BHT：预测分支是否Taken
  - 根据同一分支以前是否Taken预测当前执行是否Taken
  - 根据最近的几个分支预测当前的分支是否Taken
- BTB
  - 包含分支预测
  - 还要给出转移目标PC
  - 甚至给出转移目标指令
- 预测的准确度
  - 程序的特性
  - 预测缓存的大小

内容纲要

- 分支预测
- 推测执行

推测执行

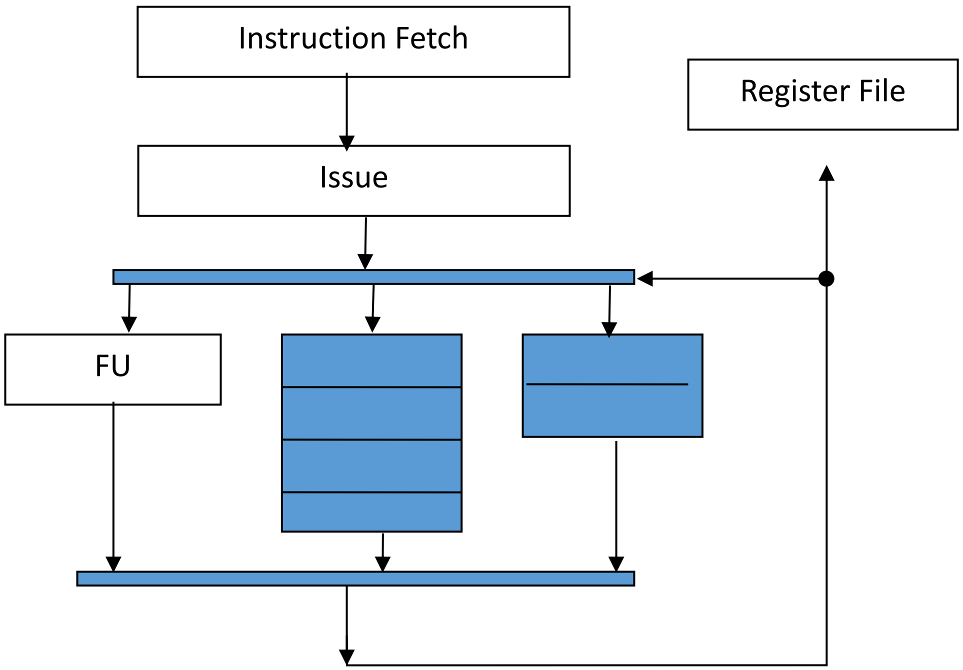
- 分支预测的目的何在
  - 推测执行
- 推测执行的基本理念
  - 假定分支预测永远正确，按预测结果发射指令
  - 对发射的指令动态调度
  - 设计一定的机制容忍预测错误
- 推测执行的挑战
  - 分支预测错误
    - 可以按照分支预测执行，但是必须保证分支结果确定之后再提交
  - 精确中断
    - 一条指令发生中断/异常时，其后的指令不能已经提交
  - 计分板和Tomasulo算法都不是按序提交
    - 按序发射，乱序提交

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

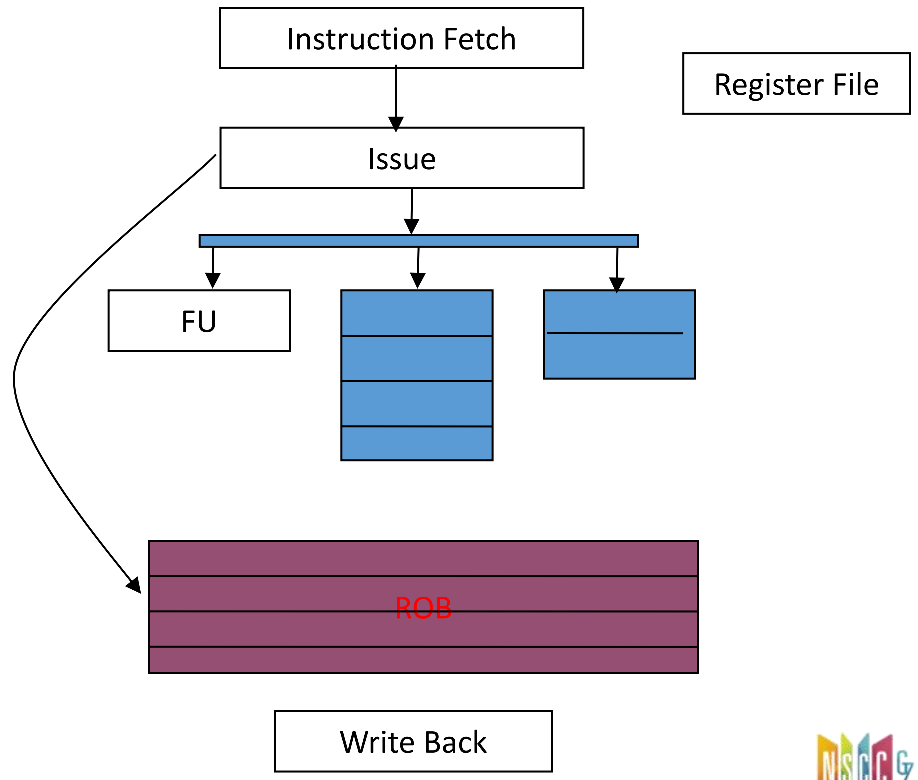
推测执行的按序提交

- 回顾Tomasulo算法
  - 指令发射到保留栈
  - 保留栈动态调度功能单元执行
  - 功能单元发送结果到CDB
  - CDB广播结果
  - 寄存器组**乱序**提交结果



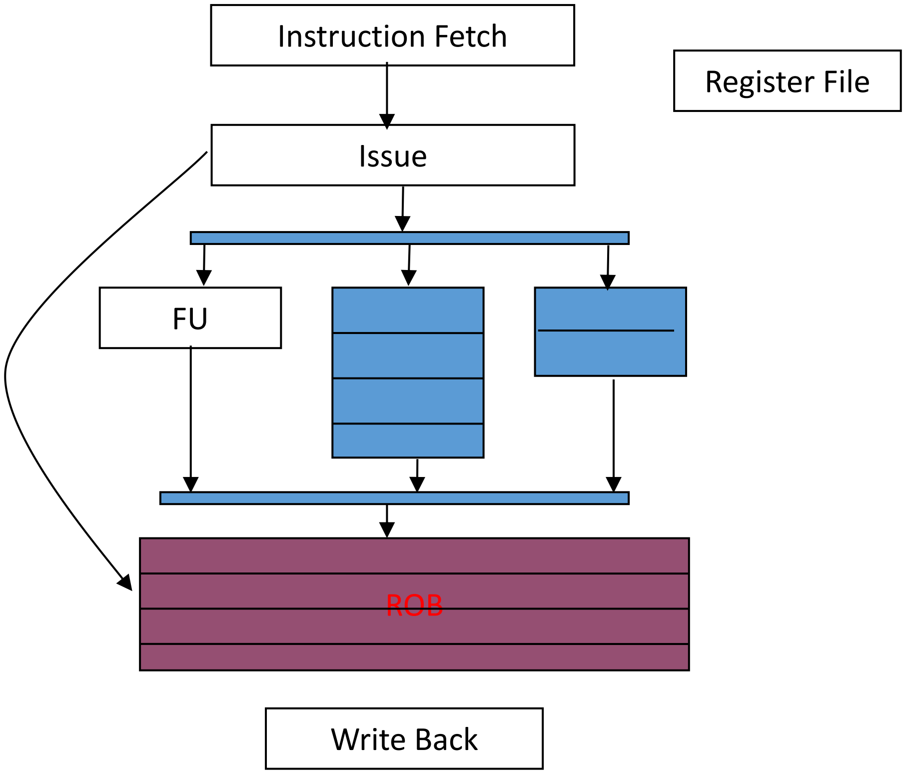
推测执行——引入重排序缓存（ROB）

- 引入重排序缓存（Reorder Buffer）的指令发射
  - 将发射的指令按序保存在ROB中
  - 记录指令的目的寄存器、PC值



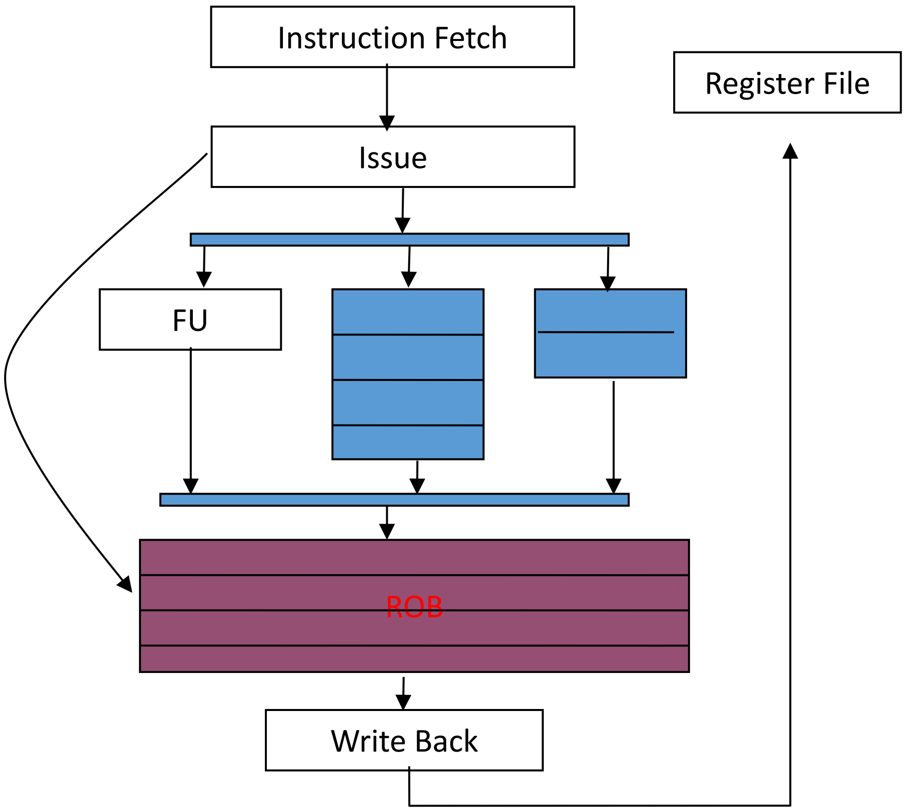
推测执行——引入重排序缓存（ROB）

- 引入重排序缓存（Reorder Buffer）的指令执行
  - 将指令执行结果保存在ROB中
    - 不提交
    - 但可广播到保留栈各个等待该结果的单元
  - 记录可能发生的中断、异常



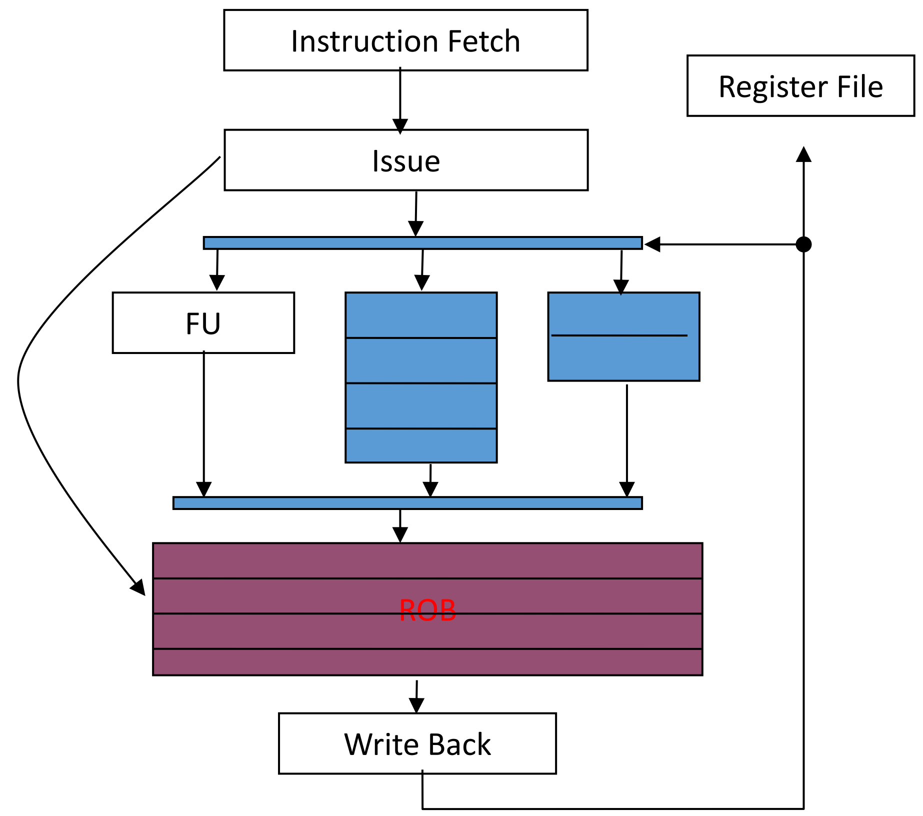
推测执行——引入重排序缓存（ROB）

- 引入重排序缓存（Reorder Buffer）的结果写回
  - 将ROB头部的指令结果提交
    - 写寄存器
    - 写存储器
  - 处理发生的中断、异常



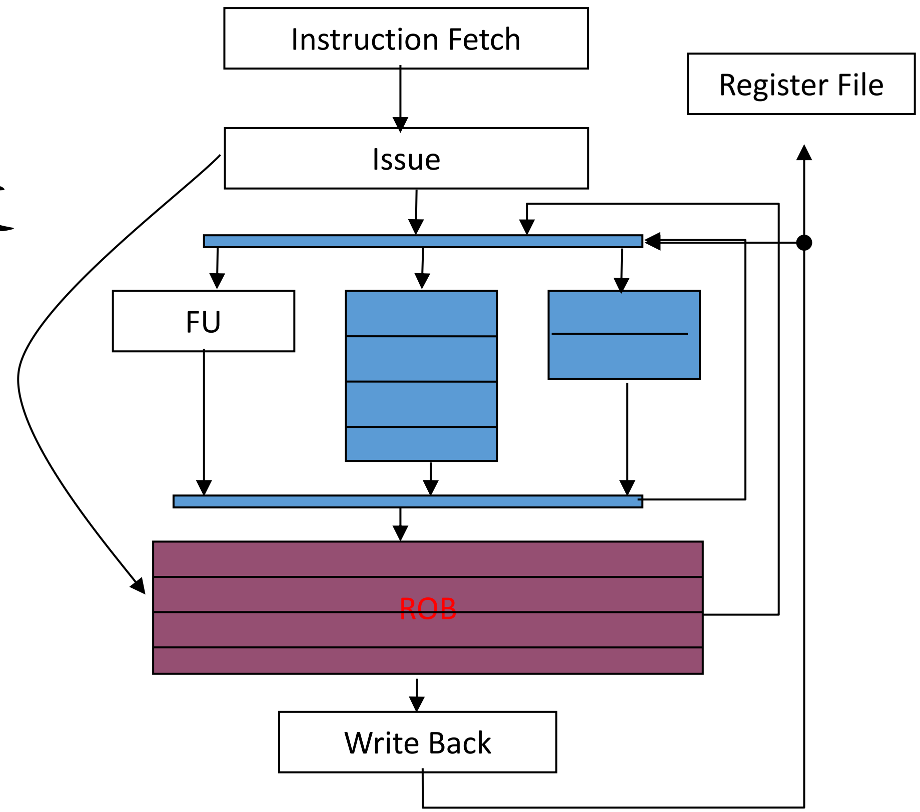
推测执行——引入重排序缓存（ROB）

- 优化1：重排序缓存+Forwarding
  - 将已经确定提交的结果直接Forward到需要该结果的指令



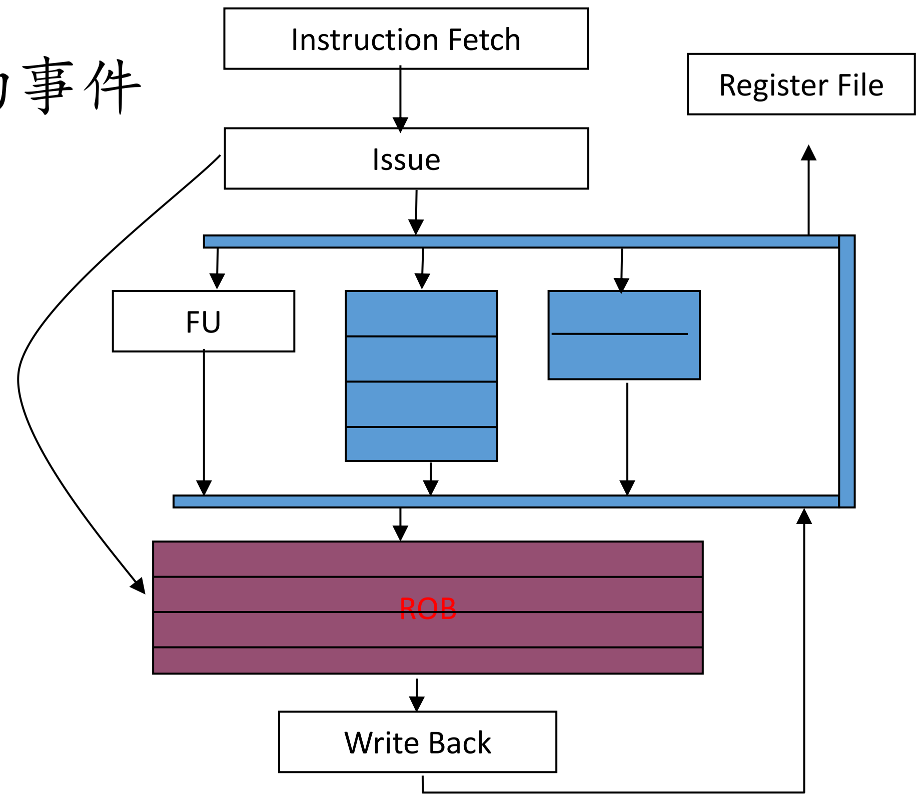
推测执行——引入重排序缓存（ROB）

- 优化2：重排序缓存+Forwarding+推测执行
  - 发射分支指令到ROB
    - 但须标记这是猜测执行的指令
  - 正常执行分支指令
    - 但谨慎提交
  - 分支确定之后决定是否提交
  - 预测正确
    - 后续指令都可提交
  - 预测错误
    - 清除ROB中的后续指令



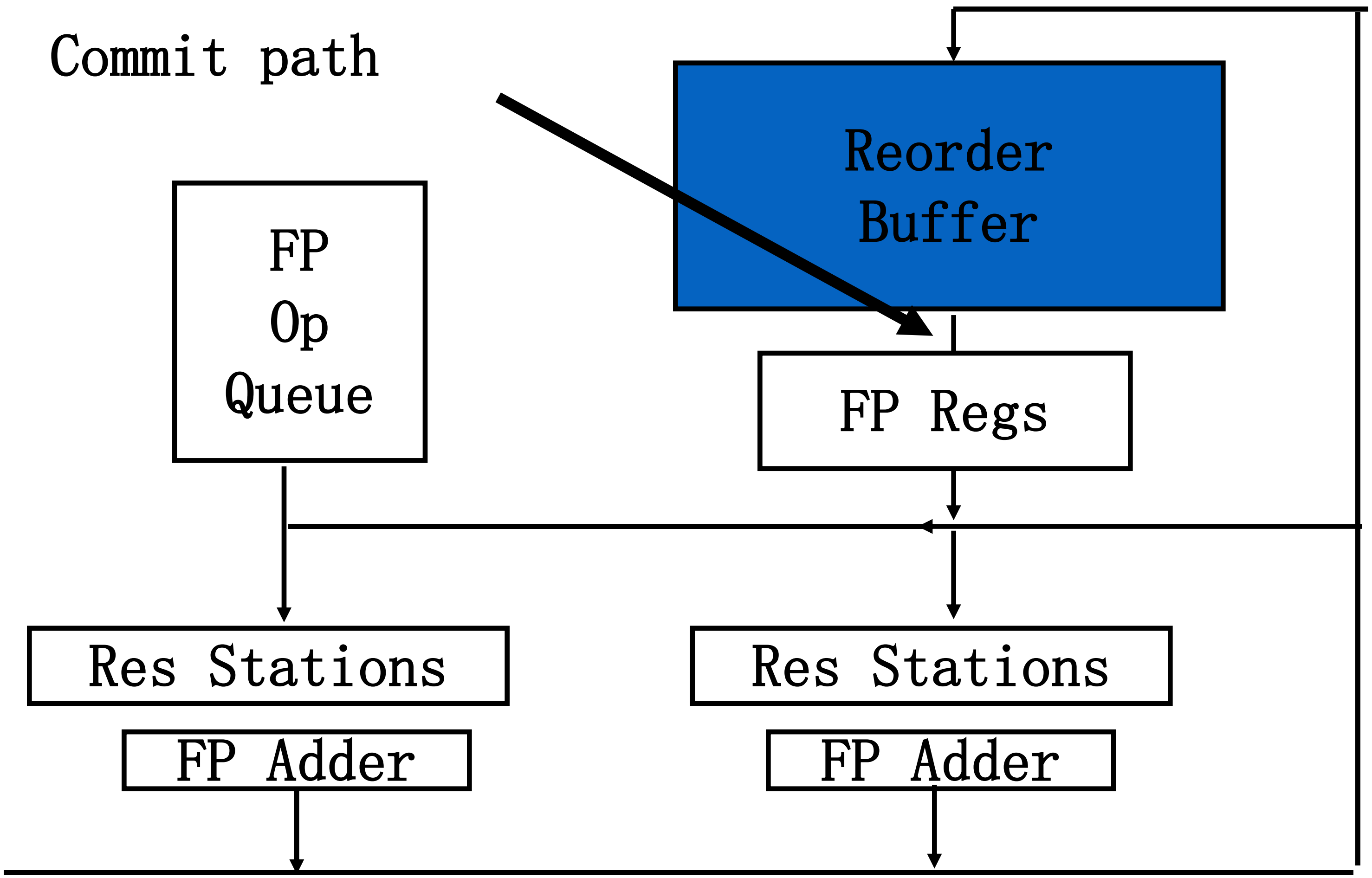
推测执行——引入重排序缓存（ROB）

- 推测执行的核心思想
  - 按序发射
  - 乱序执行
  - 按序提交
  - 提交前阻止一切不可逆转的事件
    - 中断
    - 异常
- 硬件推测执行所需的组件
  - 动态分支预测
  - 动态指令调度
    - 跨基本块的调度
  - 指令推测执行
  - 结果UNDO模块



推测执行——引入重排序缓存（ROB）

- 基于重排序缓存的结果提交
  - 将ROB头部的指令结果提交



推测执行——引入重排序缓存（ROB）

- 重排序缓存的各个字段
  - **指令类型**
    - 分支指令：无需提交结果
    - Store指令：需要写内存
    - 写寄存器指令：需要写寄存器
  - **目标域**
    - 寄存器编号
    - 内存地址
  - **Value**
  - **Ready**
    - 指令已经执行，随时准备提交

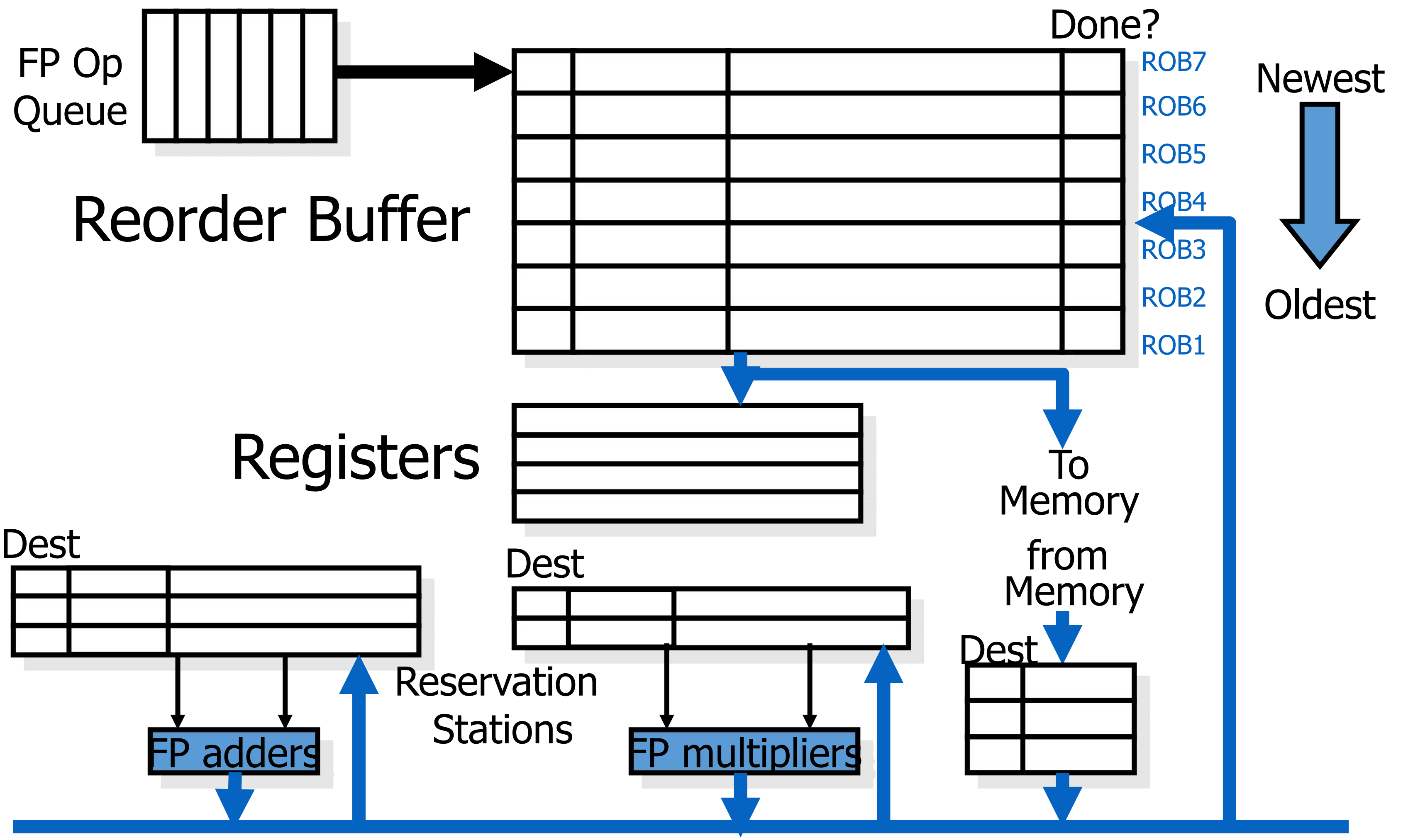
推测执行——引入重排序缓存（ROB）

- Issue:
  - Allocate RS and ROB, read available operands
- Execute:
  - Begin execution when operand values are available
- Write result:
  - Write result and ROB tag on CDB[1]
- Commit:
  - When ROB reaches head of ROB, update register
  - When a mispredicted branch reaches head of ROB, discard all entries

[1] Operand source is now reorder buffer instead of functional unit

推测执行——引入重排序缓存（ROB）

- 一个MIPS下的举例

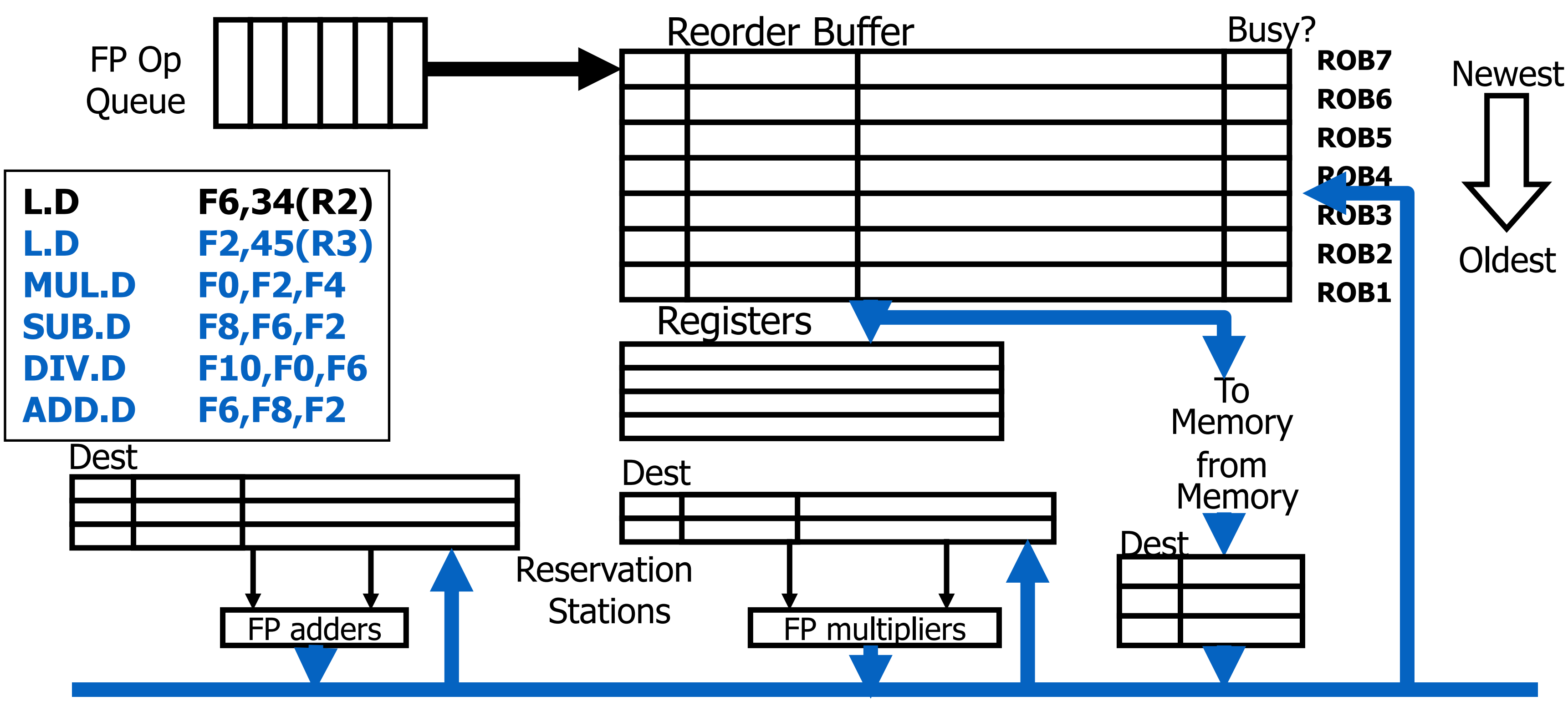


推测执行——引入重排序缓存（ROB）

- 带推测执行的Tomasulo算法
  - Issue
  - Execute
  - Write result
  - Commit
    - Additional stage
- 一些基本假设
  - Add: 2 cycles
  - Multiply: 10 cycles
  - Divide: 40 cycles

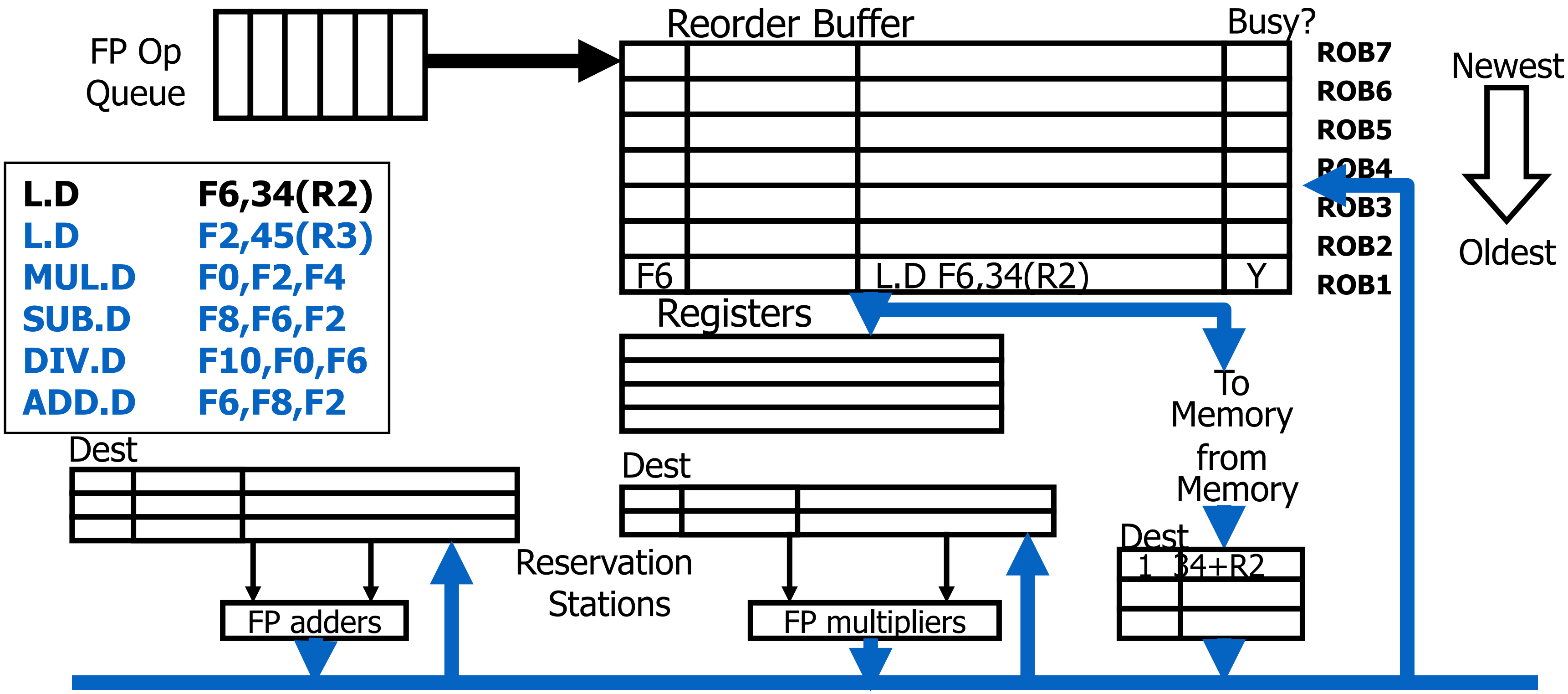
L.D	F6,34(R2)
L.D	F2,45(R3)
MUL.D	F0,F2,F4
SUB.D	F8,F6,F2
DIV.D	F10,F0,F6
ADD.D	F6,F8,F2

推测执行（初始化）



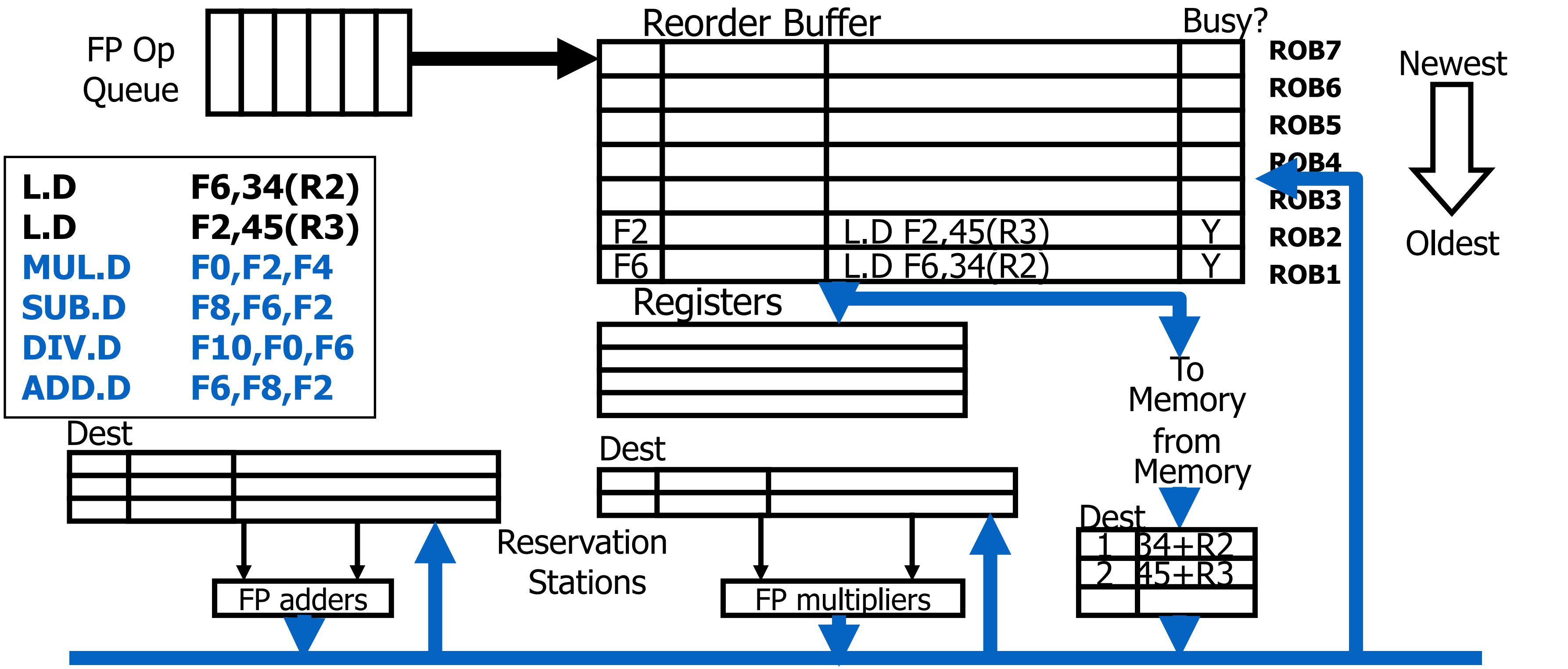
FP Op
Queue
Reorder Buffer
Busy?
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Registers
To
Memory
from
Memory
Dest
Dest
Dest
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟1）

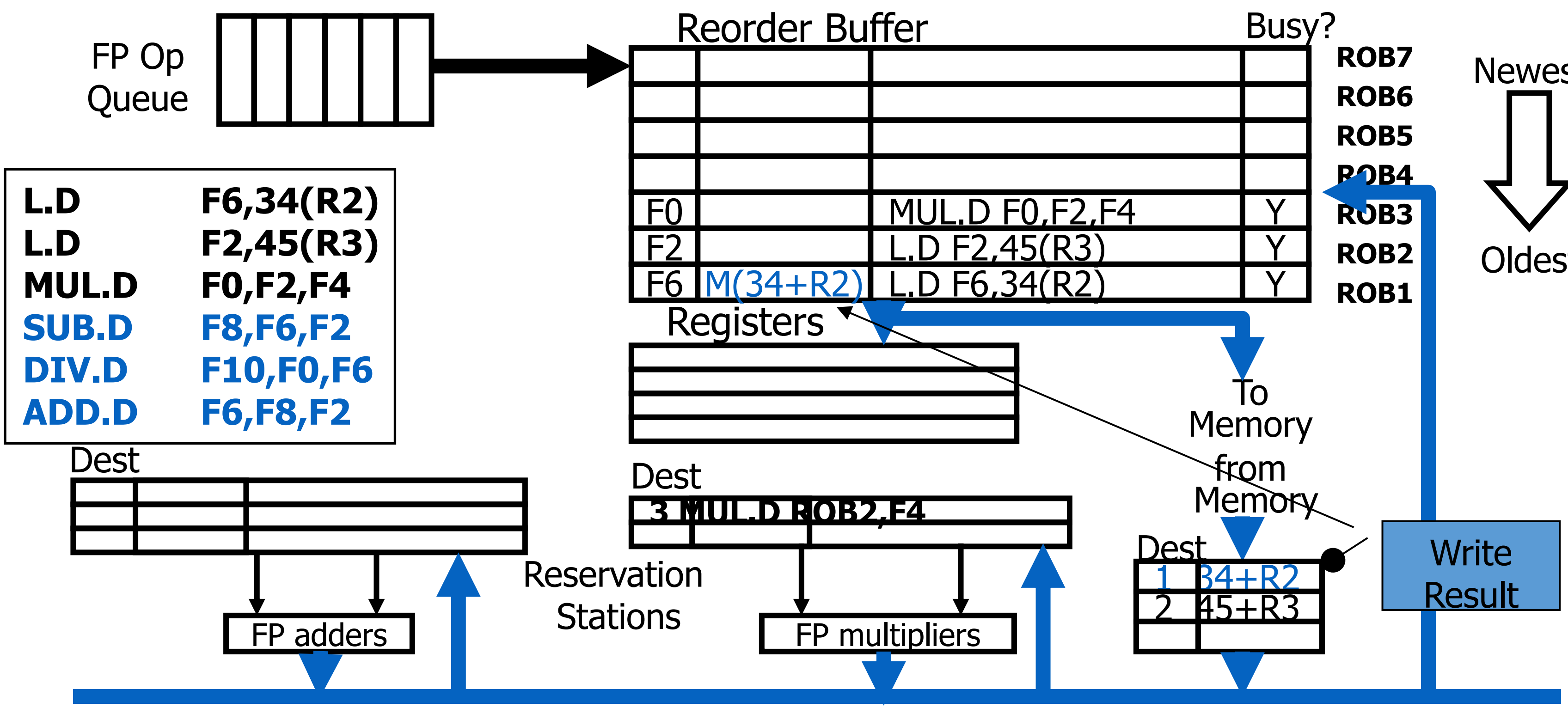


FP Op
Queue
Reorder Buffer
Busy?
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F6
L.D F6,34(R2)
Y
Newest
Oldest
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Registers
To
Memory
from
Memory
Dest
Dest
Dest
1 34+R2
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟2）

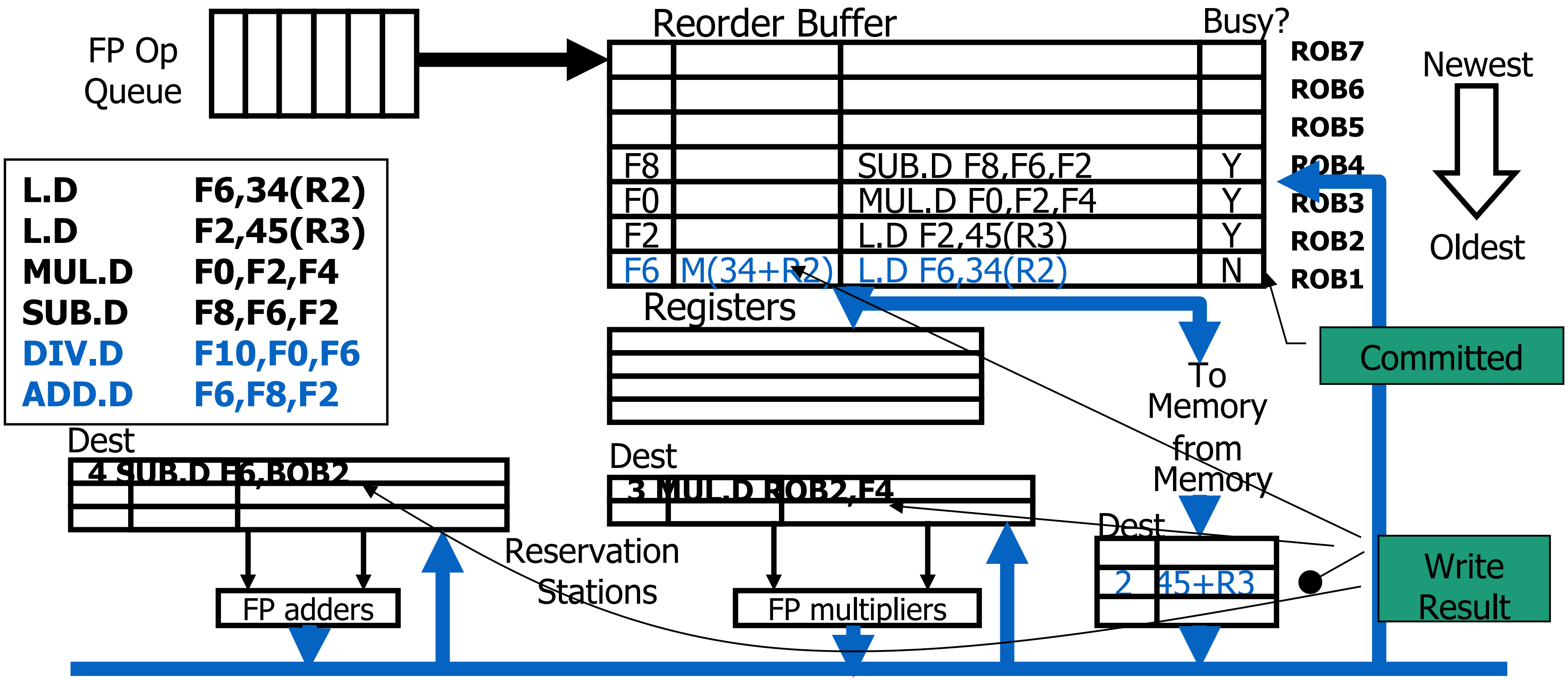

FP Op
Queue
Reorder Buffer
Busy?
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F2
L.D F2,45(R3)
Y
F6
L.D F6,34(R2)
Y
Newest
Oldest
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Registers
To
Memory
from
Memory
Dest
Dest
Dest
1 34+R2
2 45+R3
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟3）



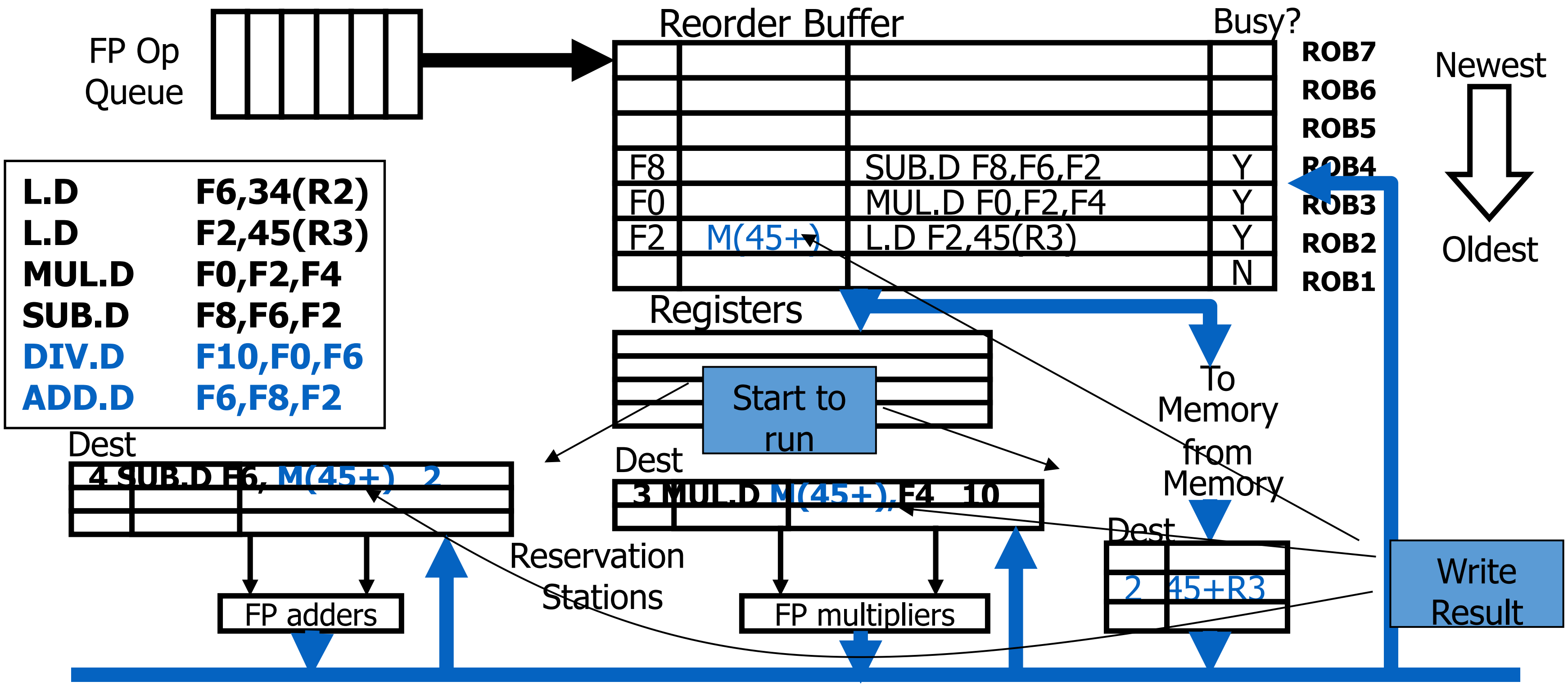
FP Op Queue
Reorder Buffer
Busy?
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F0 MUL.D F0,F2,F4 Y
F2 L.D F2,45(R3) Y
F6 M(34+R2) L.D F6,34(R2) Y
Newest
Oldest
Registers
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Dest
Dest
3 MUL.D ROB2,F4
To Memory
from Memory
Dest
1 34+R2
2 45+R3
Reservation Stations
FP adders
FP multipliers
Write Result

推测执行（时钟4-1）



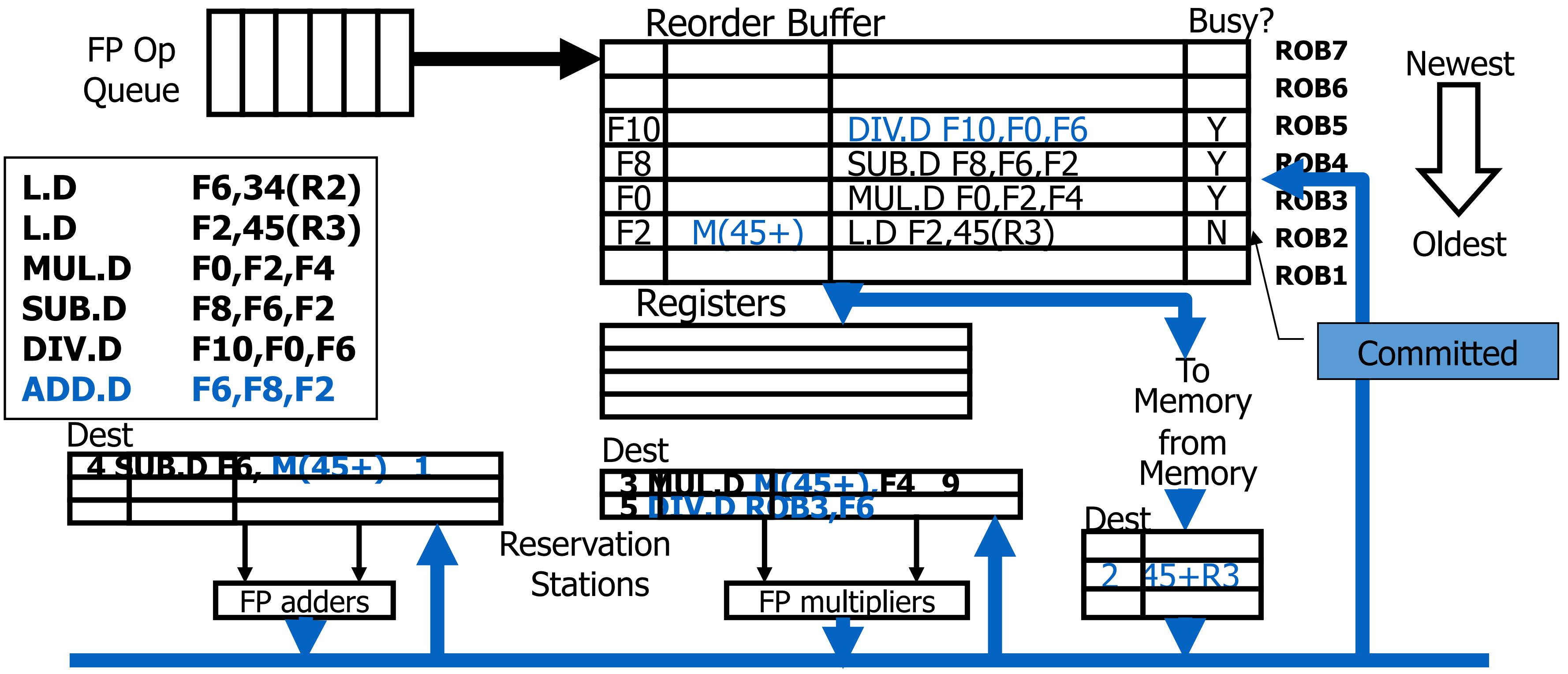
FP Op Queue
Reorder Buffer
Busy?
F8 SUB.D F8,F6,F2 Y
F0 MUL.D F0,F2,F4 Y
F2 L.D F2,45(R3) Y
F6 M(34+R2) L.D F6,34(R2) N
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
Committed
To Memory
from Memory
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Dest
4 SUB.D F6,ROB2
Dest
3 MUL.D ROB2,F4
Dest
2 45+R3
Reservation Stations
FP adders
FP multipliers
Write Result

推测执行（时钟4-2）

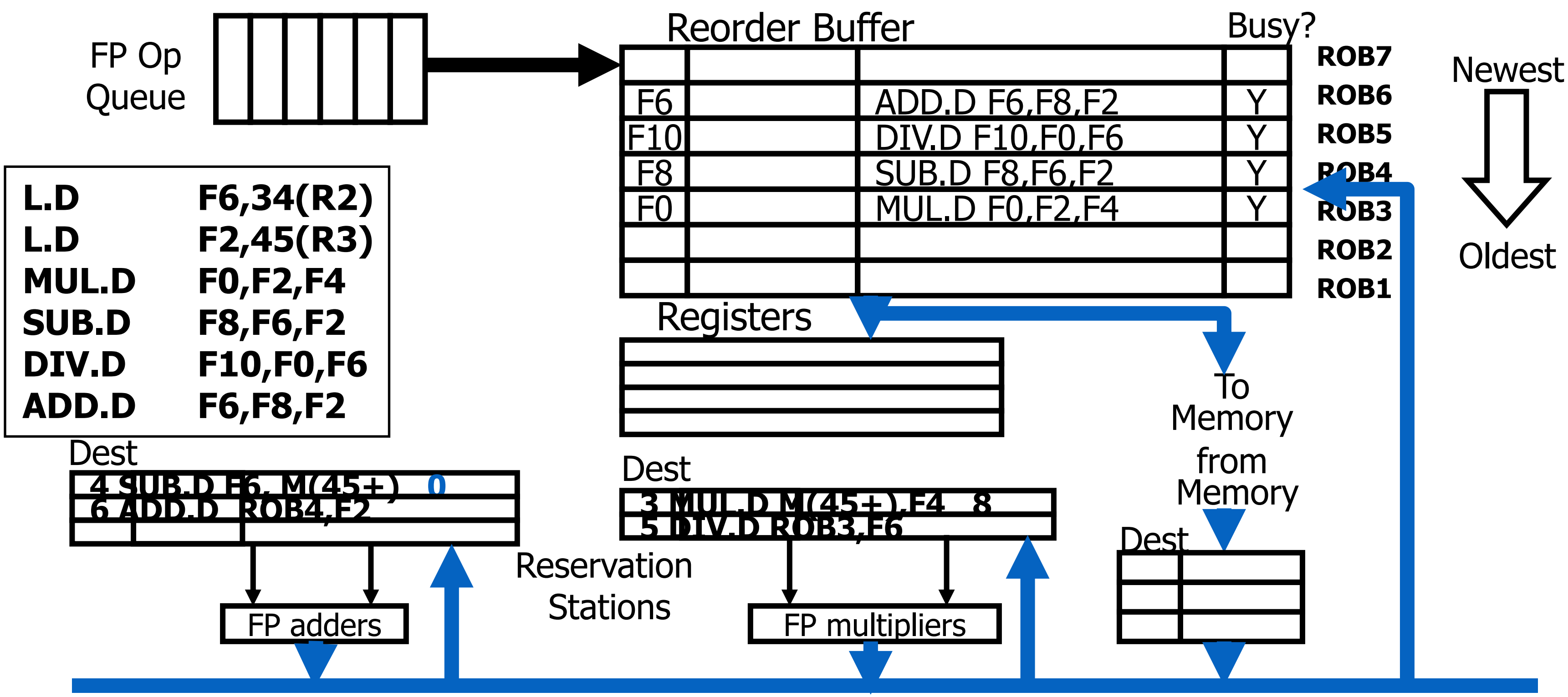


FP Op Queue
Reorder Buffer
Busy?
F8
SUB.D F8,F6,F2
Y
F0
MUL.D F0,F2,F4
Y
F2
M(45+)
L.D F2,45(R3)
Y
N
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Registers
Start to run
Dest
4 SUB.D F6, M(45+) 2
Dest
3 MUL.D M(45+),F4 10
To Memory
from Memory
Dest
2 45+R3
Reservation Stations
FP adders
FP multipliers
Write Result

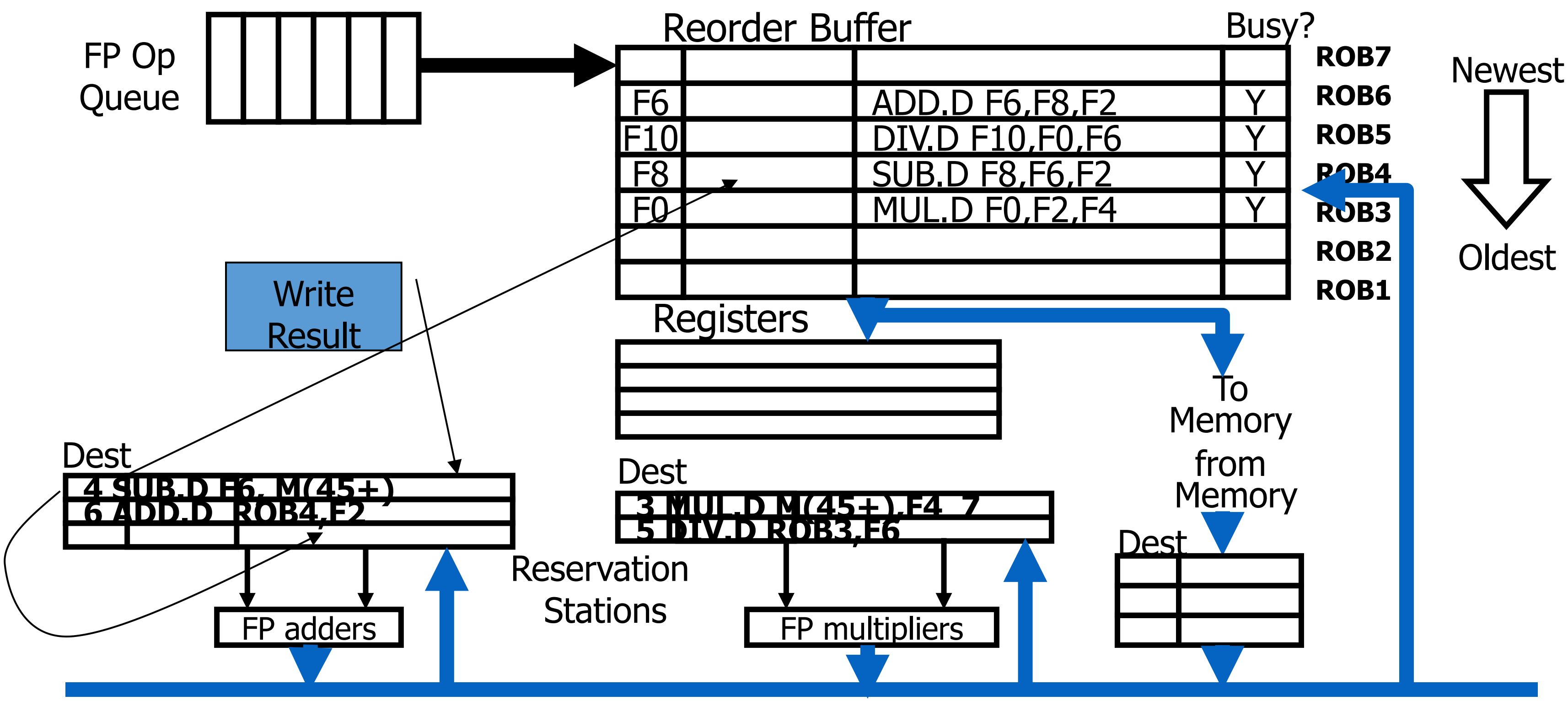
推测执行（时钟5）


FP Op
Queue
Reorder Buffer
Busy?
F10 DIV.D F10,F0,F6 Y
F8 SUB.D F8,F6,F2 Y
F0 MUL.D F0,F2,F4 Y
F2 M(45+) L.D F2,45(R3) N
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Committed
Registers
To
Memory
from
Memory
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Dest
4 SUB.D F6, M(45+) 1
Dest
3 MUL.D M(45+),F4 9
5 DIV.D ROB3,F6
Dest
2 45+R3
FP adders
Reservation
Stations
FP multipliers

推测执行（时钟6）

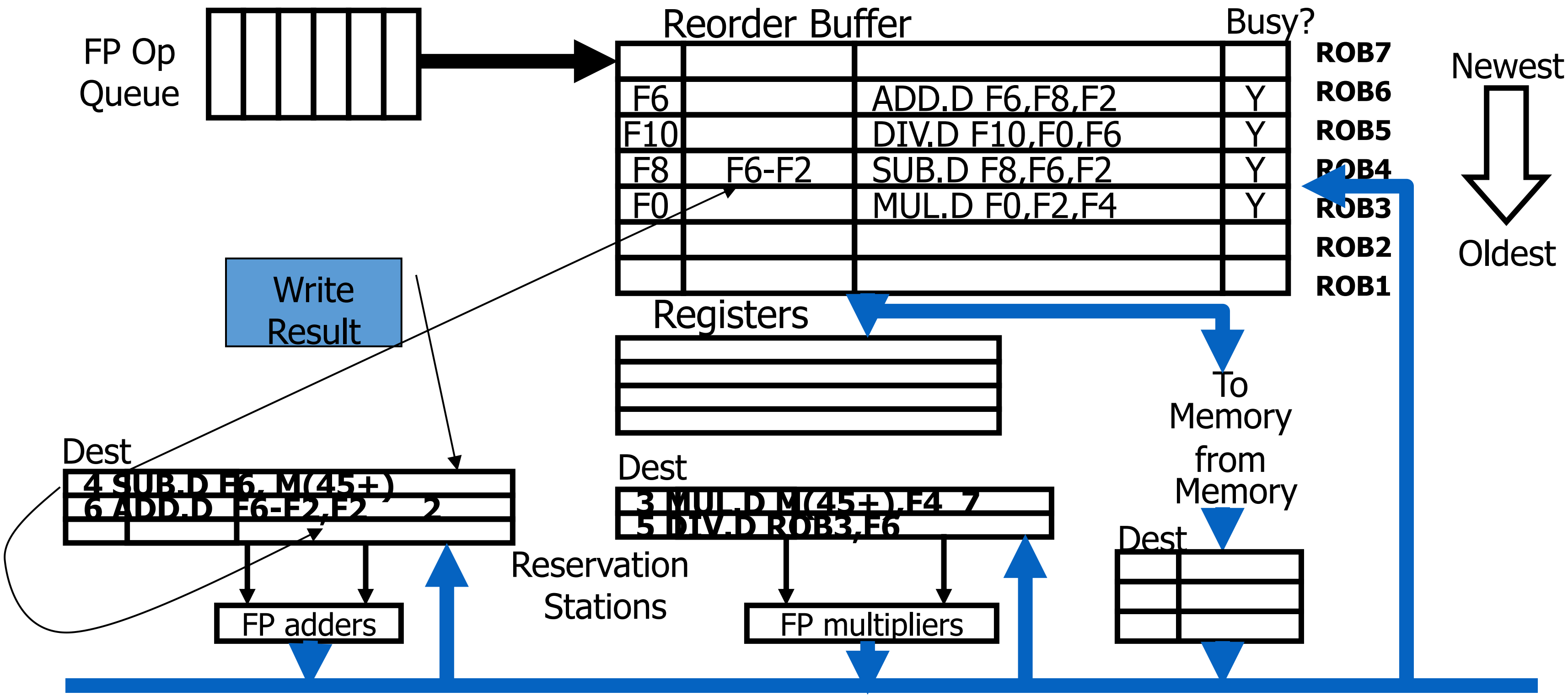

FP Op
Queue
Reorder Buffer
Busy?
F6 | ADD.D F6,F8,F2 | Y
F10 | DIV.D F10,F0,F6 | Y
F8 | SUB.D F8,F6,F2 | Y
F0 | MUL.D F0,F2,F4 | Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Registers
To
Memory
from
Memory
Dest
4 SUB.D F6, M(45+) 0
6 ADD.D ROB4,F2
Dest
3 MUL.D M(45+),F4 8
5 DIV.D ROB3,F6
Dest
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟7-1）

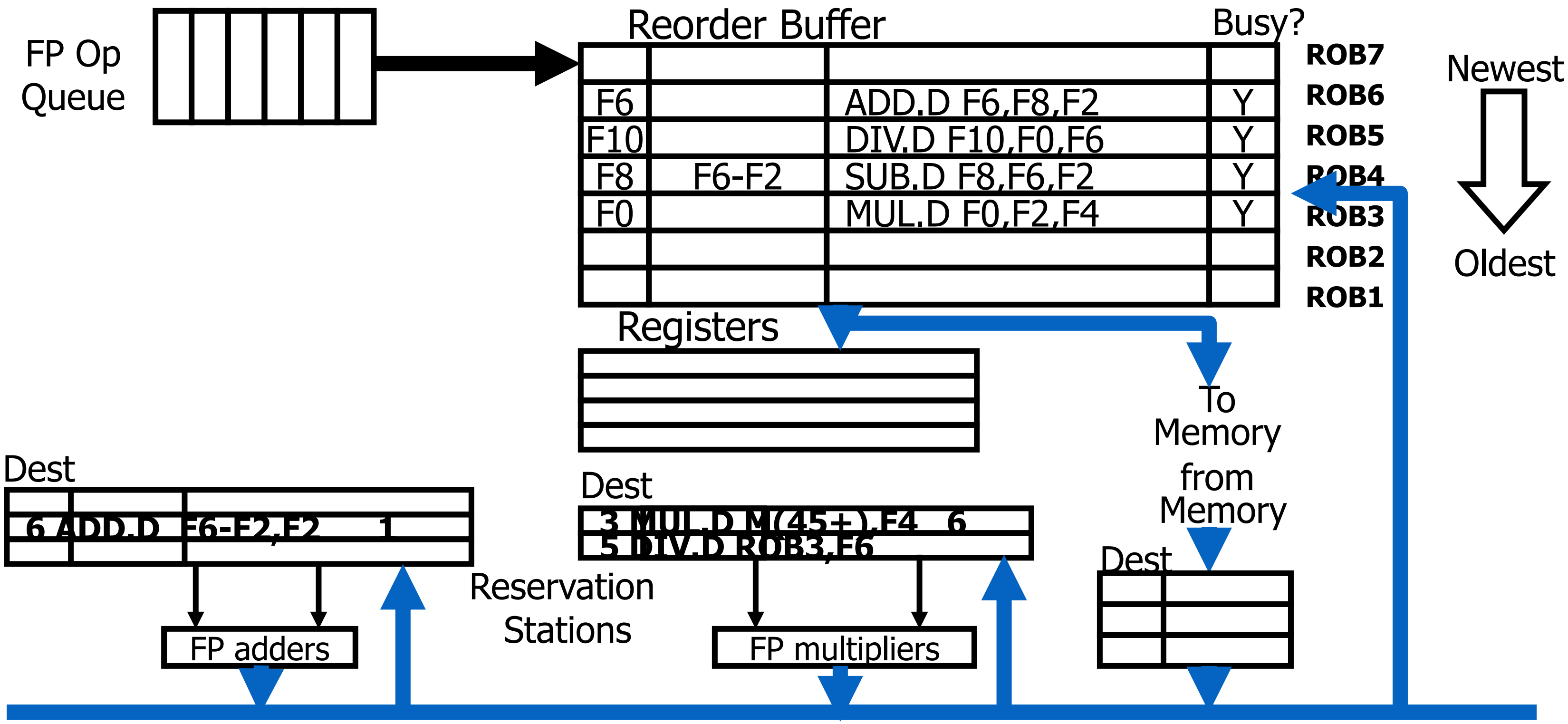


FP Op Queue
Reorder Buffer
Busy?
F6
ADD.D F6,F8,F2
Y
F10
DIV.D F10,F0,F6
Y
F8
SUB.D F8,F6,F2
Y
F0
MUL.D F0,F2,F4
Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Write Result
Registers
To Memory
from Memory
Dest
4 SUB.D F6, M(45+)
6 ADD.D ROB4,F2
Dest
3 MUL.D M(45+),F4 7
5 DIV.D ROB3,F6
Dest
Reservation Stations
FP adders
FP multipliers

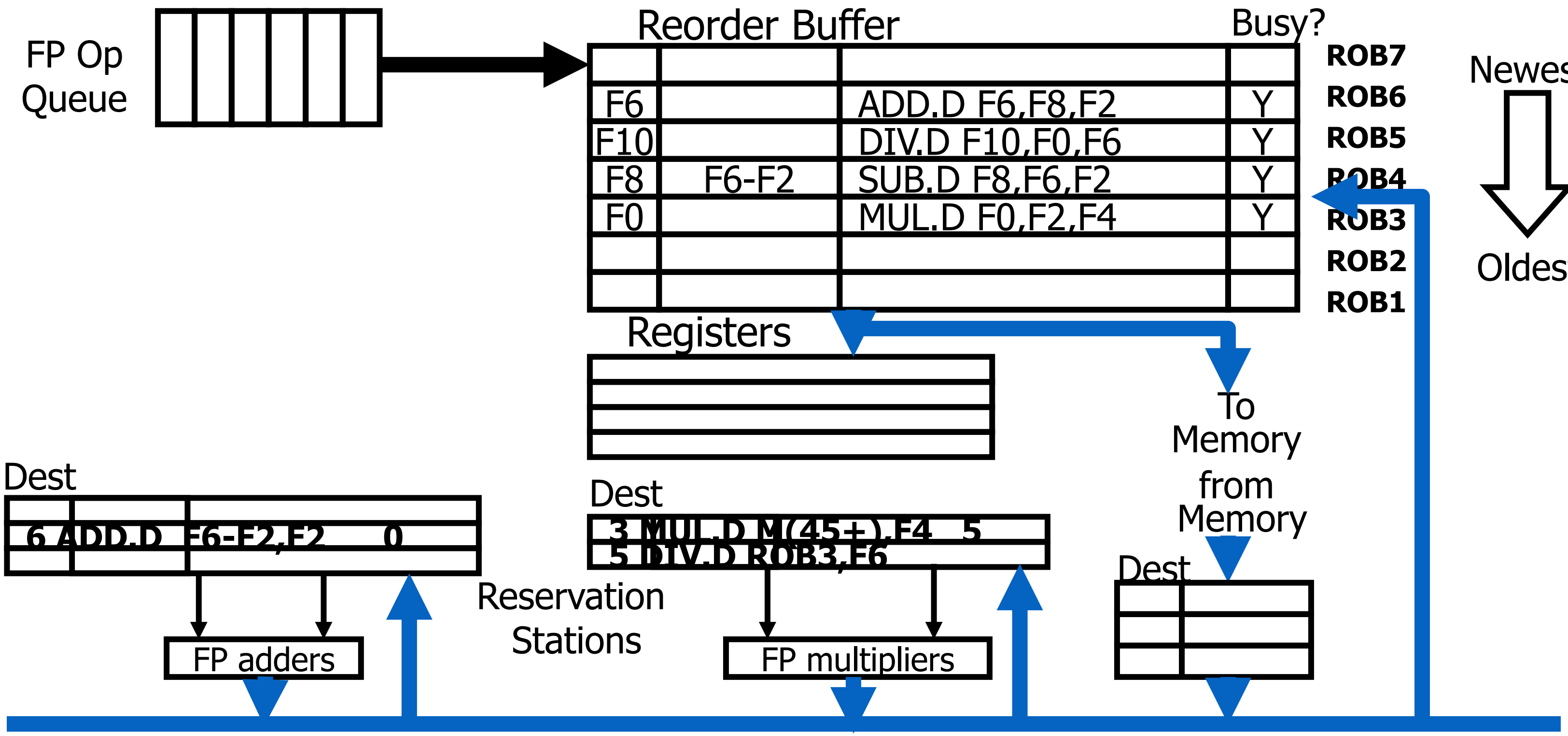
推测执行（时钟7-2）


FP Op Queue
Reorder Buffer
Busy?
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F6 ADD.D F6,F8,F2 Y
F10 DIV.D F10,F0,F6 Y
F8 F6-F2 SUB.D F8,F6,F2 Y
F0 MUL.D F0,F2,F4 Y
Newest
Oldest
Write Result
Registers
To Memory
from Memory
Dest
4 SUB.D F6, M(45+)
6 ADD.D F6-F2,F2 2
Dest
3 MUL.D M(45+),F4 7
5 DIV.D ROB3,F6
Dest
Reservation Stations
FP adders
FP multipliers

推测执行（时钟8）

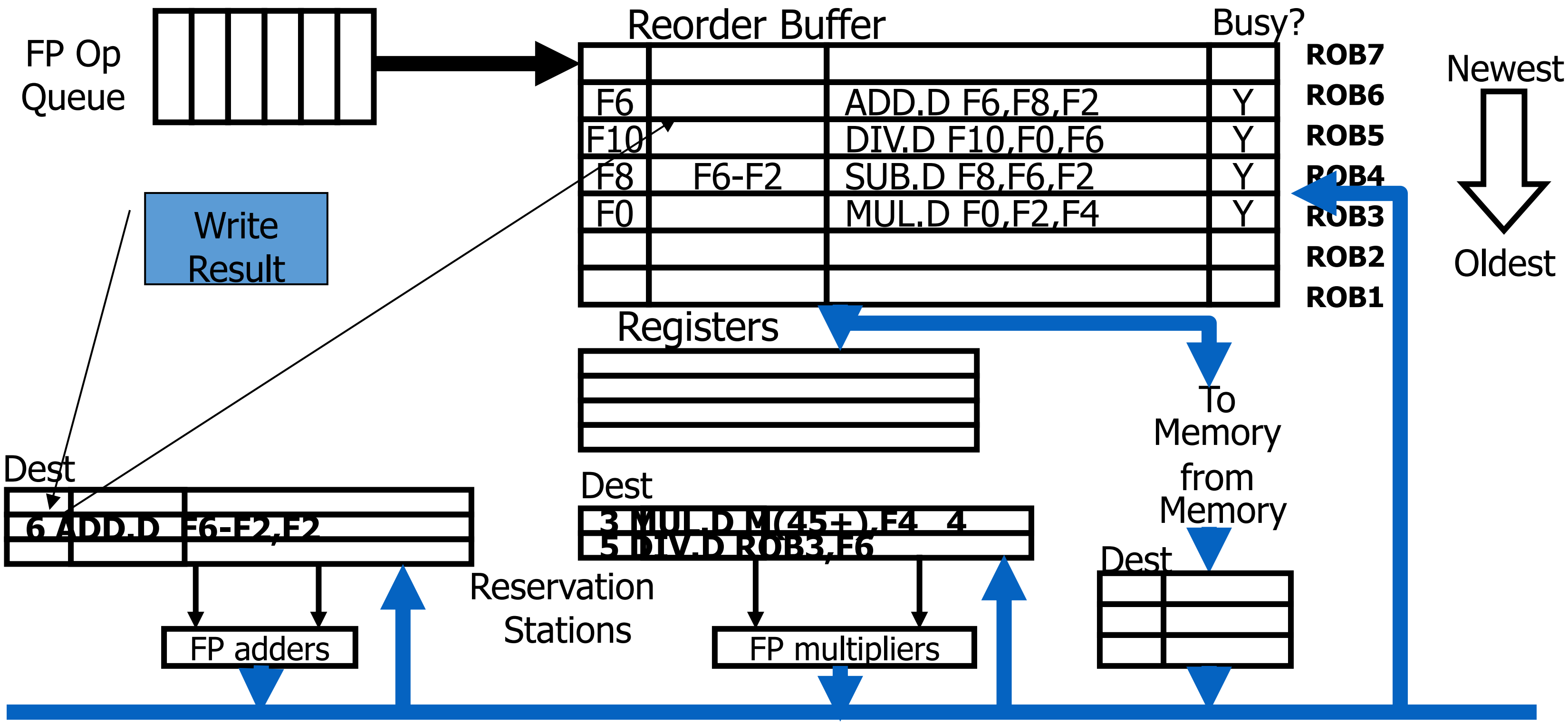

FP Op
Queue
Reorder Buffer
Busy?
F6
ADD.D F6,F8,F2
Y
F10
DIV.D F10,F0,F6
Y
F8
F6-F2
SUB.D F8,F6,F2
Y
F0
MUL.D F0,F2,F4
Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
To
Memory
from
Memory
Dest
6 ADD.D F6-F2,F2 1
Dest
3 MUL.D M(45+),F4 6
5 DIV.D ROB3,F6
Dest
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟9）



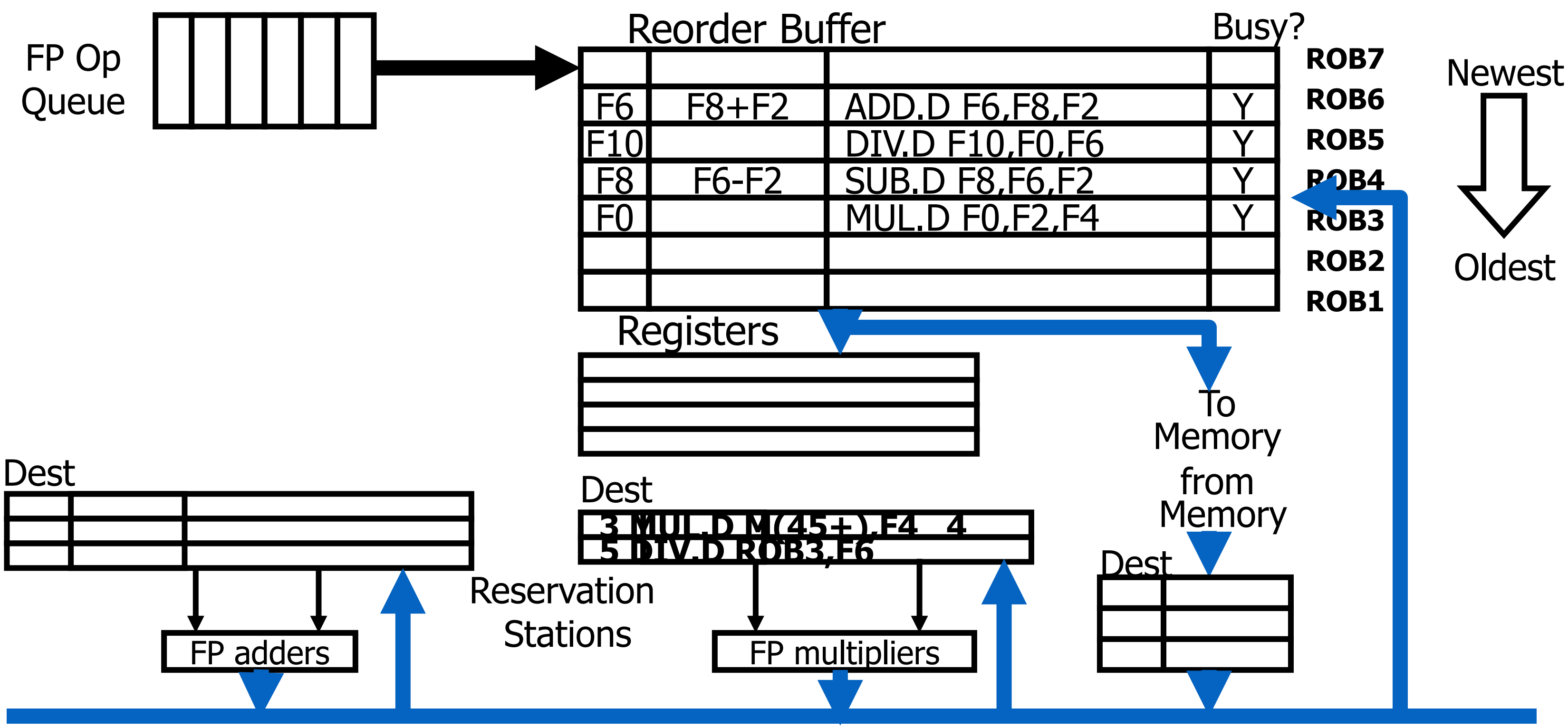
FP Op Queue
Reorder Buffer
Busy?
F6 ADD.D F6,F8,F2 Y
F10 DIV.D F10,F0,F6 Y
F8 F6-F2 SUB.D F8,F6,F2 Y
F0 MUL.D F0,F2,F4 Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
To Memory
from Memory
Dest
6 ADD.D F6-F2,F2 0
Dest
3 MUL.D M(45+),F4 5
5 DIV.D ROB3,F6
Reservation Stations
FP adders
FP multipliers
Dest

推测执行（时钟10-1）



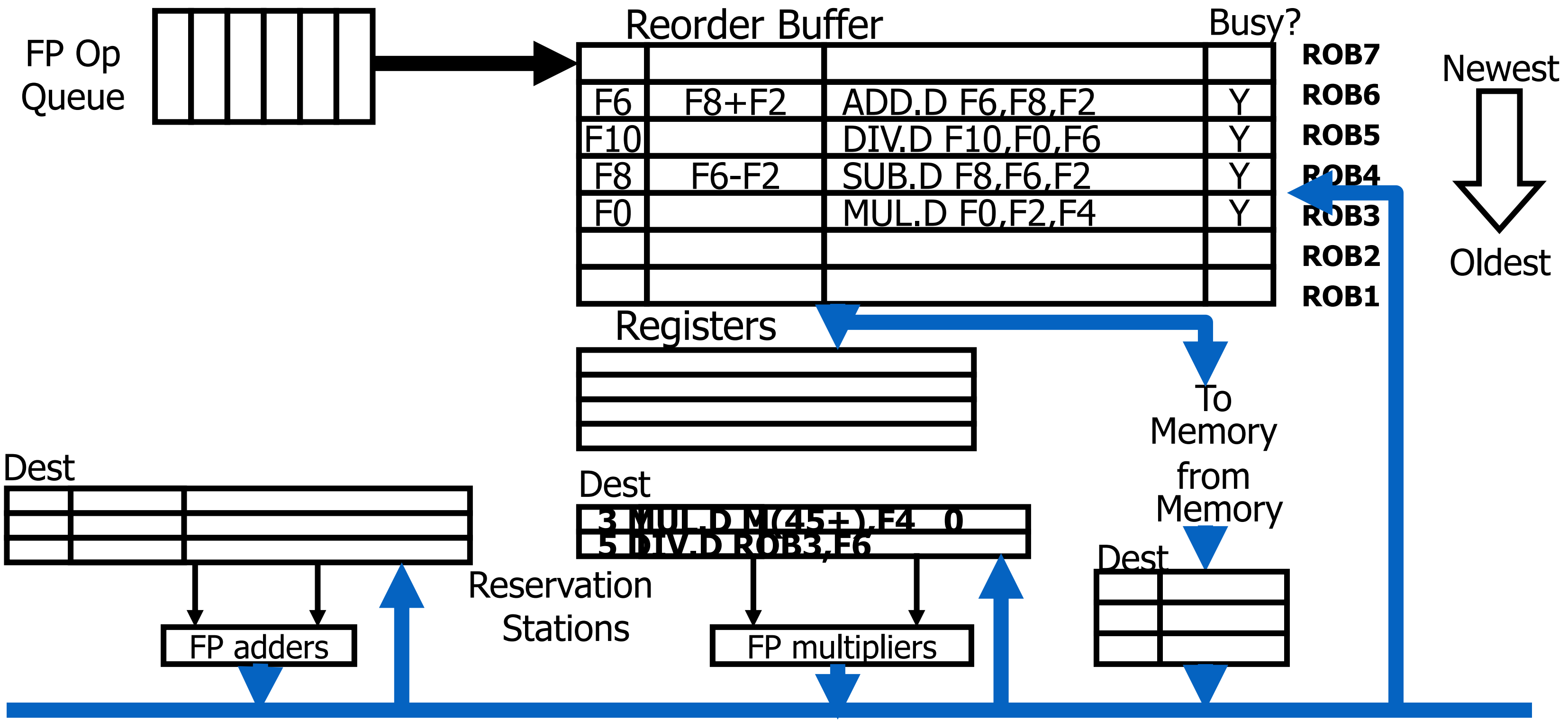
FP Op Queue
Reorder Buffer
Busy?
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F6 ADD.D F6,F8,F2 Y
F10 DIV.D F10,F0,F6 Y
F8 F6-F2 SUB.D F8,F6,F2 Y
F0 MUL.D F0,F2,F4 Y
Newest
Oldest
Write Result
Registers
To Memory
from Memory
Dest
6 ADD.D F6-F2,F2
Dest
3 MUL.D M(45+),F4 4
5 DIV.D ROB3,F6
Dest
Reservation Stations
FP adders
FP multipliers

推测执行（时钟10-2）



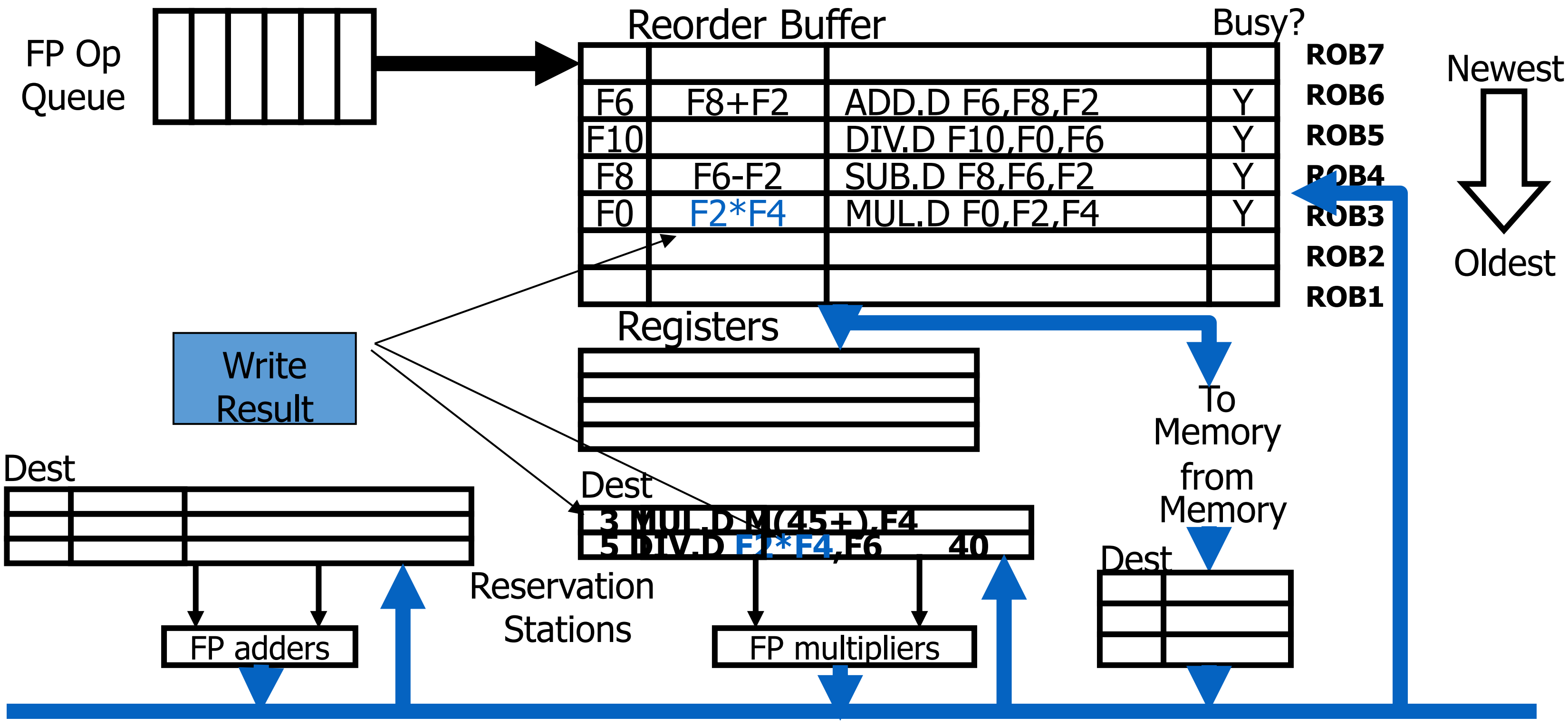
FP Op Queue
Reorder Buffer
Busy?
F6 | F8+F2 | ADD.D F6,F8,F2 | Y
F10 | | DIV.D F10,F0,F6 | Y
F8 | F6-F2 | SUB.D F8,F6,F2 | Y
F0 | | MUL.D F0,F2,F4 | Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
To Memory
from Memory
Dest
Dest
3 MUL.D M(45+),F4 4
5 DIV.D ROB3,F6
Dest
Reservation Stations
FP adders
FP multipliers

推测执行（时钟10-14）



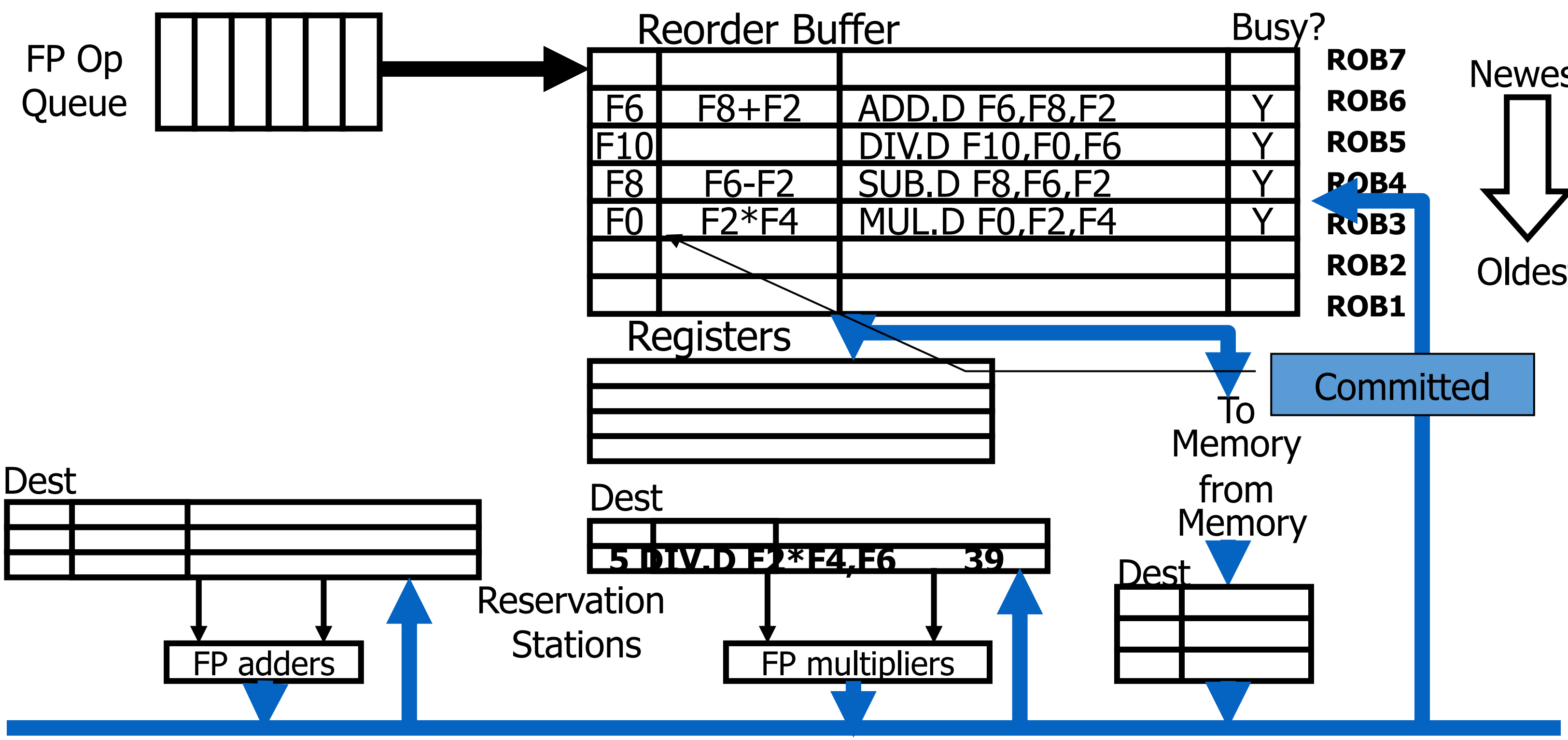
FP Op
Queue
Reorder Buffer
Busy?
F6
F8+F2
ADD.D F6,F8,F2
Y
F10
DIV.D F10,F0,F6
Y
F8
F6-F2
SUB.D F8,F6,F2
Y
F0
MUL.D F0,F2,F4
Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
To
Memory
from
Memory
Dest
Dest
3 MUL.D M(45+),F4 0
5 DIV.D ROB3,F6
Dest
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟15）



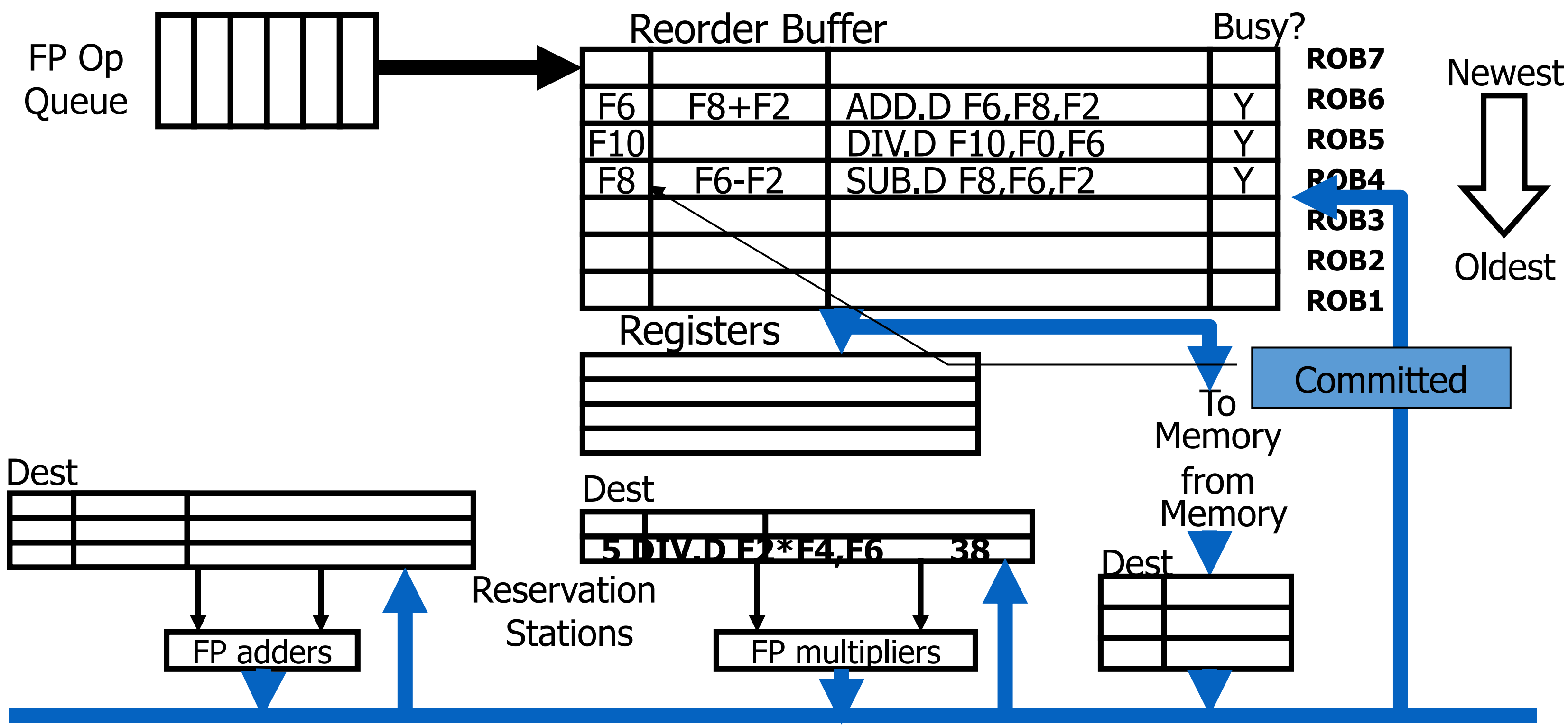
FP Op
Queue
Reorder Buffer
Busy?
F6 F8+F2 ADD.D F6,F8,F2 Y
F10 DIV.D F10,F0,F6 Y
F8 F6-F2 SUB.D F8,F6,F2 Y
F0 F2*F4 MUL.D F0,F2,F4 Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
Write
Result
To
Memory
from
Memory
Dest
Dest
3 MUL.D M(45+),F4
5 DIV.D F2*F4,F6 40
Dest
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟16）



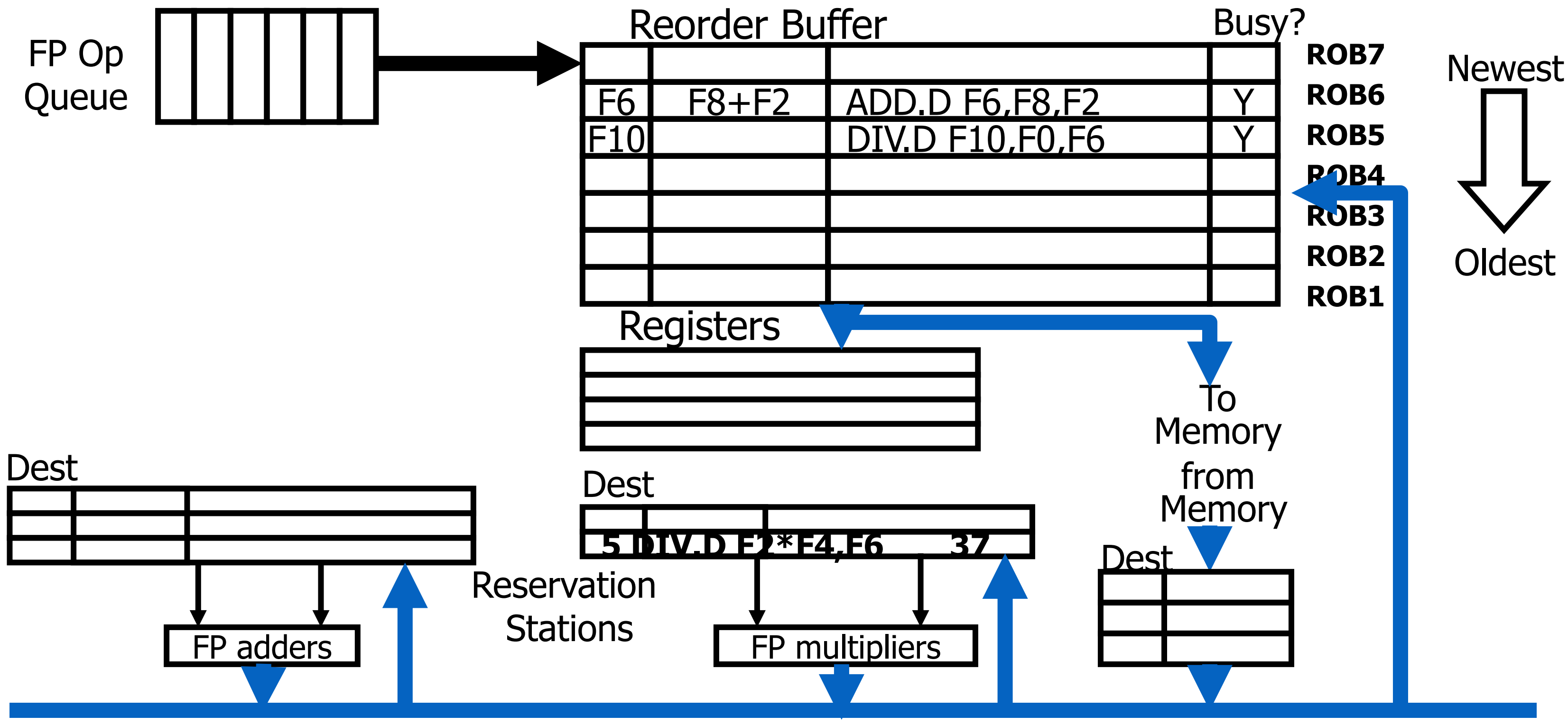
FP Op
Queue
Reorder Buffer
Busy?
ROB7
F6 F8+F2 ADD.D F6,F8,F2 Y
ROB6
F10 DIV.D F10,F0,F6 Y
ROB5
F8 F6-F2 SUB.D F8,F6,F2 Y
ROB4
F0 F2*F4 MUL.D F0,F2,F4 Y
ROB3
ROB2
ROB1
Newest
Oldest
Registers
Committed
To
Memory
from
Memory
Dest
Dest
5 DIV.D F2*F4,F6 39
Dest
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟17）



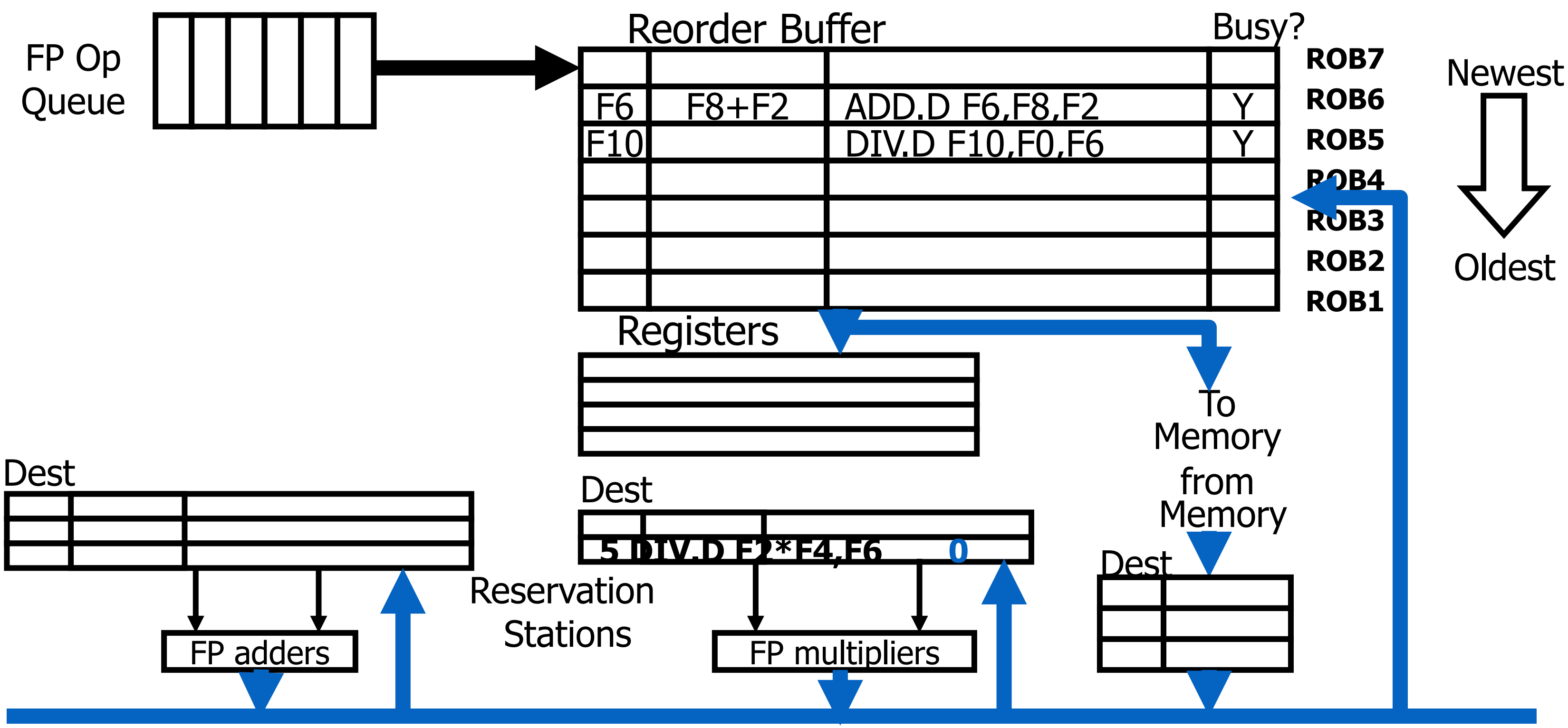
FP Op Queue
Reorder Buffer
Busy?
F6 F8+F2 ADD.D F6,F8,F2 Y
F10 DIV.D F10,F0,F6 Y
F8 F6-F2 SUB.D F8,F6,F2 Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
Committed
To Memory
from Memory
Dest
Dest
5 DIV.D F2*F4,F6 38
Dest
Reservation Stations
FP adders
FP multipliers

推测执行（时钟18）



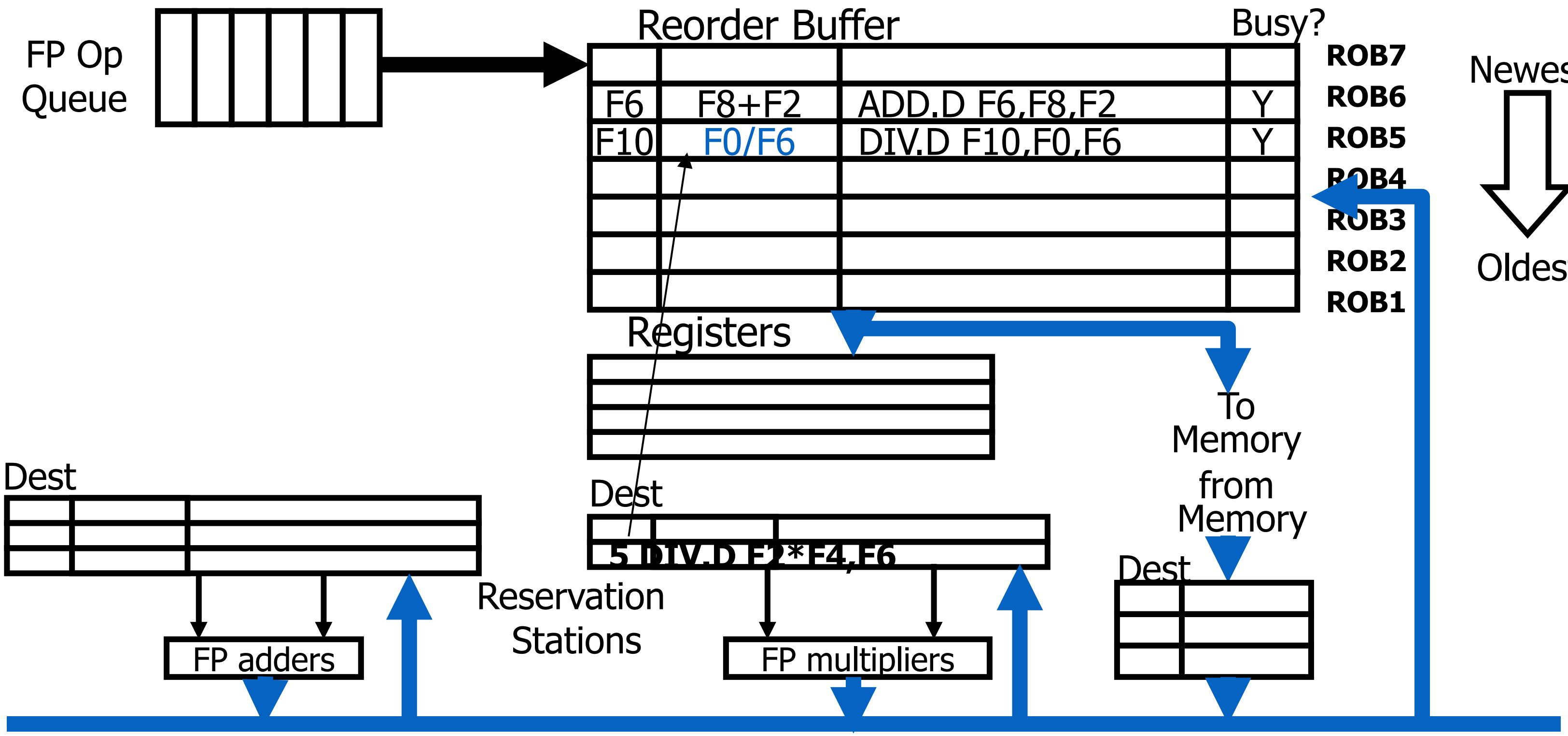
FP Op Queue
Reorder Buffer
Busy?
F6 F8+F2 ADD.D F6,F8,F2 Y
F10 DIV.D F10,F0,F6 Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
To Memory
from Memory
Dest
Dest
5 DIV.D F2*F4,F6 37
Dest
Reservation Stations
FP adders
FP multipliers

推测执行（时钟55）



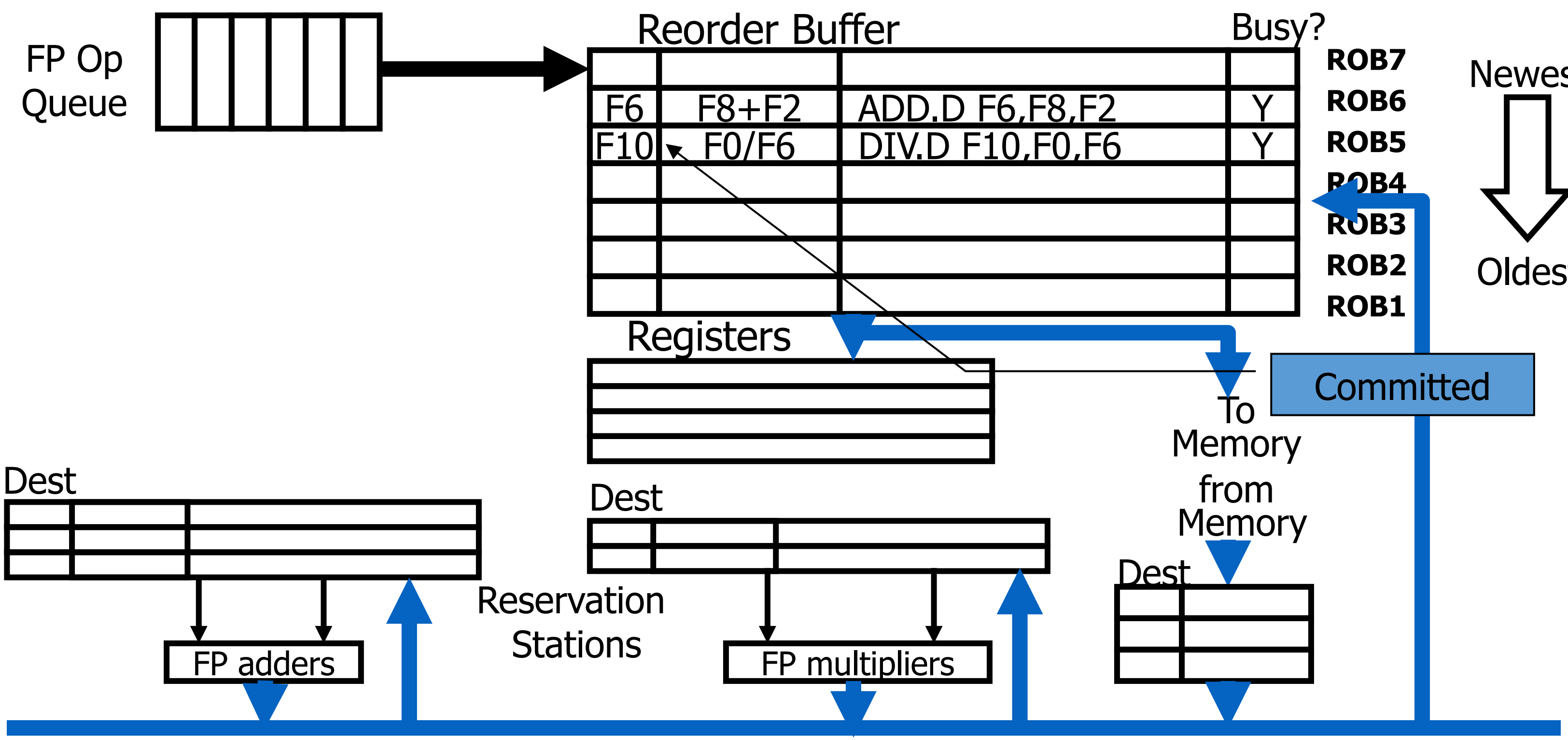
FP Op Queue
Reorder Buffer
Busy?
F6
F8+F2
ADD.D F6,F8,F2
Y
F10
DIV.D F10,F0,F6
Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
To Memory
from Memory
Dest
Dest
5 DIV.D F2*F4,F6
0
Dest
Reservation Stations
FP adders
FP multipliers

推测执行（时钟56）



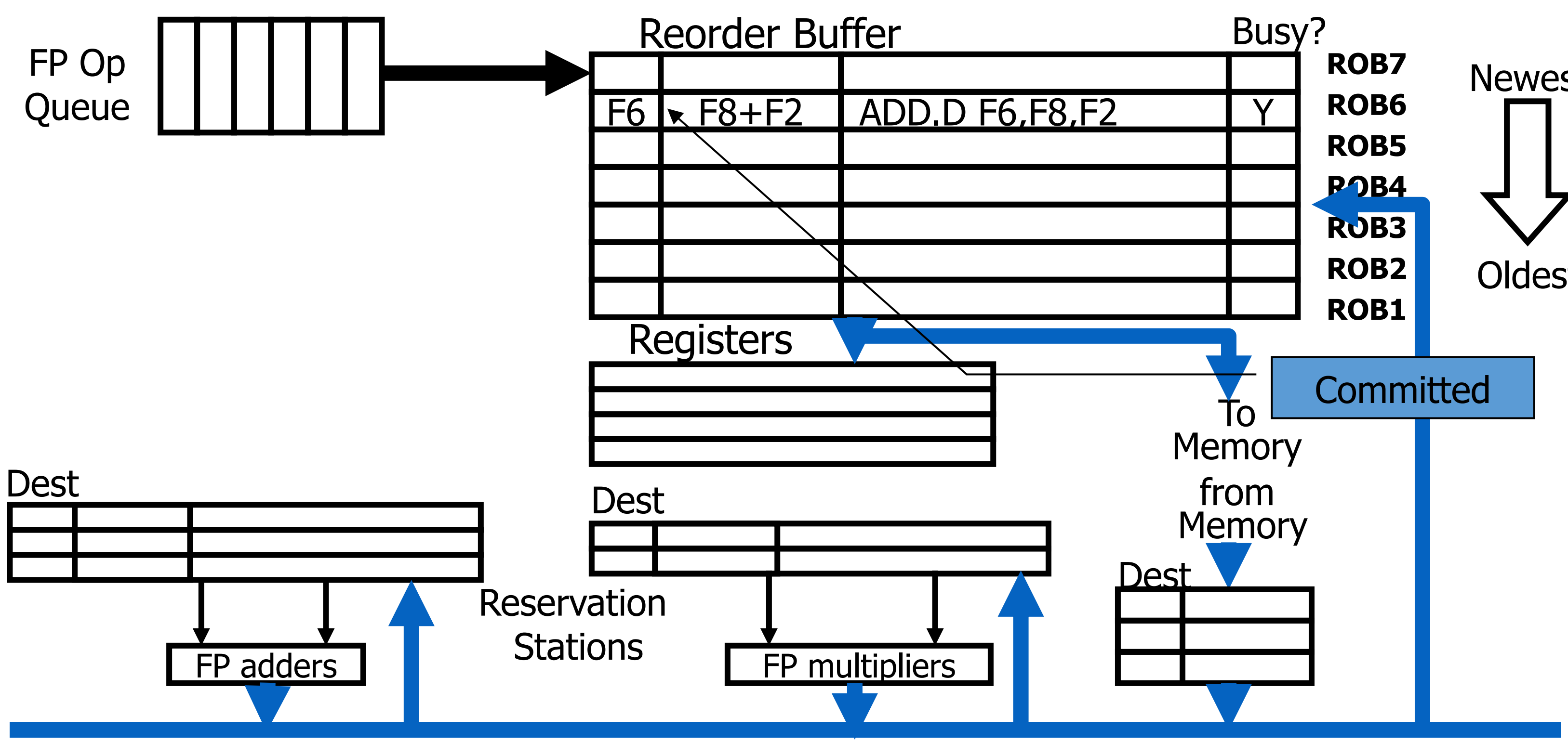
FP Op
Queue
Reorder Buffer
Busy?
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F6
F8+F2
ADD.D F6,F8,F2
Y
F10
F0/F6
DIV.D F10,F0,F6
Y
Newest
Oldest
Registers
To
Memory
from
Memory
Dest
Dest
Dest
5 DIV.D F2*F4,F6
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟57）



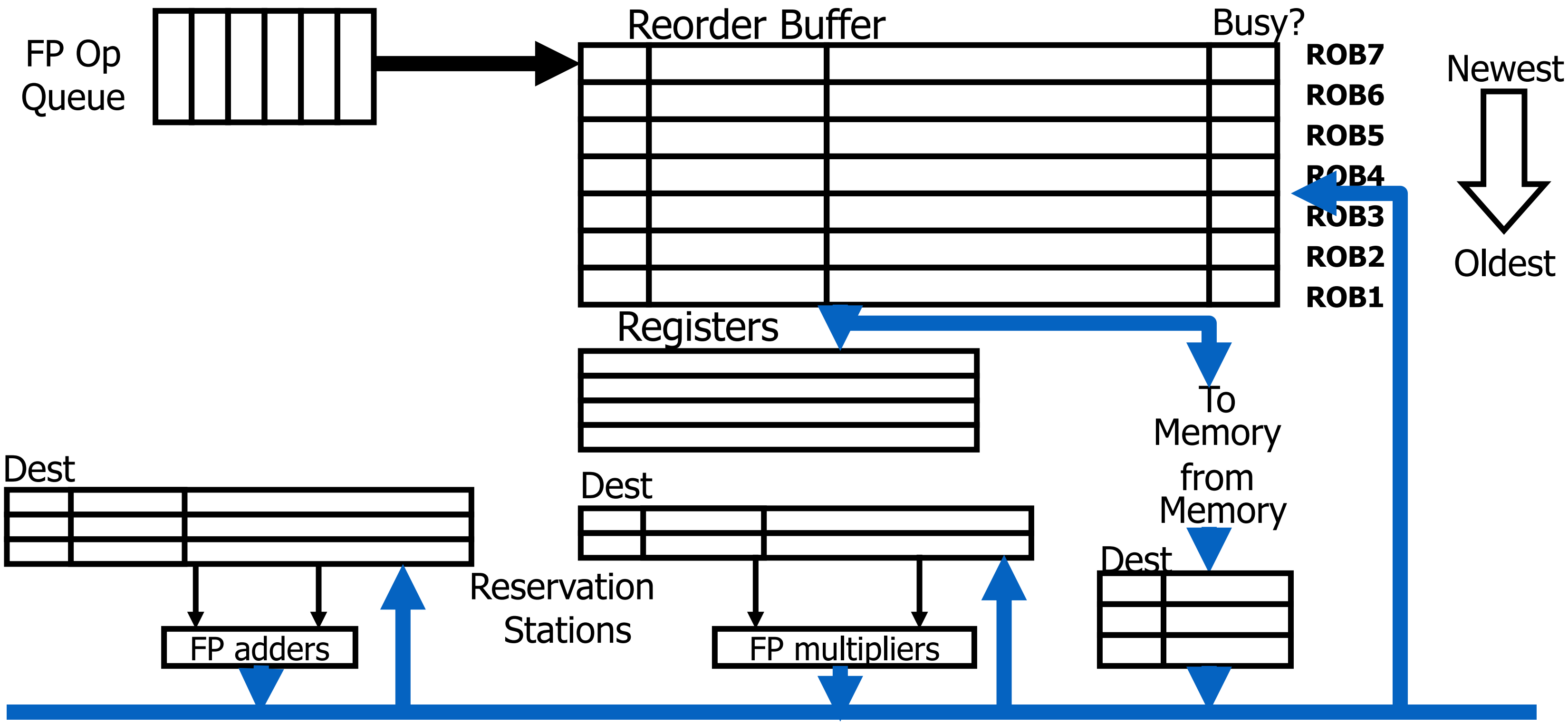
FP Op
Queue
Reorder Buffer
Busy?
F6
F8+F2
ADD.D F6,F8,F2
Y
F10
F0/F6
DIV.D F10,F0,F6
Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
Committed
To
Memory
from
Memory
Dest
Dest
Dest
Reservation
Stations
FP adders
FP multipliers

推测执行（时钟58）



FP Op Queue
Reorder Buffer
Busy?
F6
F8+F2
ADD.D F6,F8,F2
Y
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
Committed
To Memory
from Memory
Dest
Dest
Dest
Reservation Stations
FP adders
FP multipliers

推测执行（时钟59）



FP Op
Queue
Reorder Buffer
Busy?
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Newest
Oldest
Registers
To
Memory
from
Memory
Dest
Dest
Dest
Reservation
Stations
FP adders
FP multipliers

推测执行

乘法指令执行完，但还未提交的瞬间

Reservation stations								
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**Dest**	**A**
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	MUL.D	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	Yes	DIV.D		Mem[34+Regs[R2]]	#3		#5	

Reorder buffer						
Entry	**Busy**	**Instruction**		**State**	**Destination**	**Value**
1	No	L.D	F6,34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	No	L.D	F2,45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	Yes	MUL.D	F0,F2,F4	Write result	F0	#2 x Regs[F4]
4	Yes	SUB.D	F8,F6,F2	Write result	F8	#1 – #2
5	Yes	DIV.D	F10,F0,F6	Execute	F10	
6	Yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

FP Register status										
Field	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F10**
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

中山大學 SUN YAT-SEN UNIVERSITY

推测执行

- 推测执行与动态调度对比总结
  - 推测执行
    - SUB.D 和 ADD.D 一定在 MUL.D 之后提交
  - 动态调度
    - MUL.D 完成时 SUB.D 和 ADD.D早已结束
      - 如果MUL.D处发生了中断怎么办
      - 如果MUL.D是DIV.D指令，且除数为0，怎么办
- 推测执行的特点：按序提交
  - 维持**精确的中断**和**精确的异常**

推测执行（一个基于循环的例子）

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      0(R1),F4
        DADDIU   R1,R1,#-8
        BNE      R1,R2,Loop
```

- 一些基本假设
  - 第二次循环的所有指令发射到ROB中
  - 第一次循环的L.D 和 MUL.D 已经提交
  - 其他的指令均已经完成
  - Store指令既要等待地址计算、也要等待Value，才能提交

推测执行（一个基于循环的例子）

- 第一条乘法指令执行完，刚提交的瞬间

Reservation station

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
Mult1	No	MUL.D	Mem[0+Regs[R1]]	Regs[F2]			#2
Mult2	Yes	MUL.D	Mem[0+Regs[R1]]	Regs[F2]			#7

Reorder Buffer

Entry	Busy	Instruction		State	Dest	Value
1	No	L.D	F0,0(R1)	Commit	F0	Mem[0+Regs[R1]]
2	No	MUL.D	F4,F0,F2	Commit	F4	#1×Regs[F2]
3	Yes	S.D	0(R1),F4	WR	0+Regs[R1]	#2
4	Yes	DADDIU	R1,R1,#-8	WR	R1	Regs[R1]-8
5	Yes	BNE	R1,R2,Loop	WR		
6	Yes	L.D	F0,0(R1)	WR	F0	Mem[#4]
7	Yes	MUL.D	F4,F0,F2	WR	F4	#6×Regs[F2]
8	Yes	S.D	0(R1),F4	WR	0+#4	#7
9	Yes	DADDIU	R1,R1,#-8	WR	R1	#4-8
10	Yes	BNE	R1,R2,Loop	WR		

FP register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	6		7					...	
Busy	Yes	No	Yes	No	No	No	No	...	No

指令调度技术的发展历程

- 指令调度
- 循环展开
- 计分板
- Tomasulo算法
- 推测执行