



计算机体系结构概览

周学海

xhzhou@ustc.edu.cn

0551-63492149

中国科学技术大学



Summary-计算部分

- Part1: 指令级并行
- Part2: 数据级并行
- Part3: 任务级并行
- **核心目标:**
 - 如何提高计算机信息处理的效率?
 - 信息处理: 计算、访存、通信
 - 如何有效提高单条/多条指令流的执行效率?





系统结构的Flynn分类 (1966)

- **SISD: Single instruction stream, single data stream**
 - 单处理器模式
- **SIMD: Single instruction stream, multiple data streams**
 - 相同的指令作用在不同的数据
 - 可用来挖掘数据级并行(Data Level Parallelism)
 - 如：Vector processors, SIMD instructions, and Graphics processing units
- **MISD: Multiple instruction streams, single data stream**
 - No commercial implementation
- **MIMD: Multiple instruction streams, multiple data streams**
 - 通用性最强的一种结构，可用来挖掘线程级并行、数据级并行.....
 - 组织方式可以是松耦合方式也可以是紧耦合方式

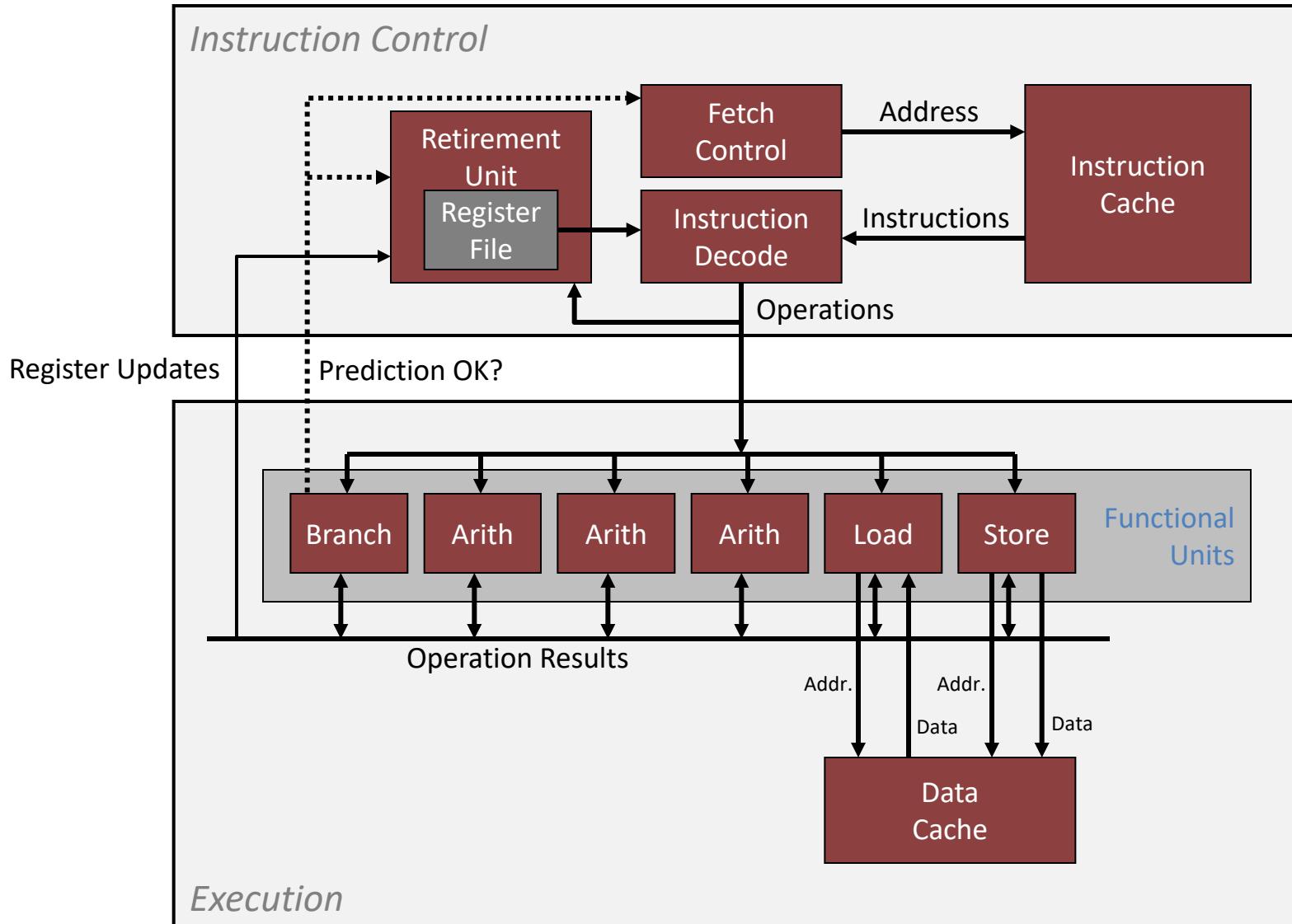


Levels of Parallelism

- **请求级并行**
 - 多个任务可被分配到多个计算机上并行执行
- **进程级并行**
 - 进程可被调度到不同的处理器上并行执行
- **线程级并行**
 - 任务被组织成多个线程，多个线程共享一个进程的地址空间
 - 每个线程都有自己独立的程序计数器和寄存器文件
- **数据级并行**
 - 单线程（逻辑上）中并行处理多个数据 (SIMD/Vector execution)
 - 一个程序计数器，多个执行部件
- **指令级并行**
 - 针对单一指令流，多个执行部件并行执行不同的指令



Modern CPU Design





Part1：指令级并行

- **指令级并行：提高程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - 流水线的扩展：指令执行周期数(可能) 不同
 - 指令级并行优化：编译优化和硬件优化
 - 动态调度的“漏网之鱼”——控制相关与异常处理问题
 - 分支预测技术
 - 推断执行技术
- **多发射流水线：ILP的突破 (CPI<1)**
 - 每个cycle发射多条指令执行
- **多线程技术：单线程ILP瓶颈的突破**
 - 流水线执行的指令来源于多条指令流
- **其他问题**



Part1：指令级并行

- **指令级并行：提高单条指令流（单线程）程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - 流水线的扩展：指令执行周期数(可能) 不同
 - 指令级并行优化：编译优化和硬件优化
 - 动态调度的“漏网之鱼”——控制相关与异常处理问题
 - 分支预测技术
 - 推断执行技术
- **多发射流水线：ILP的突破 (CPI<1)**
 - 每个cycle发射多条指令执行
- **多线程技术：单线程ILP瓶颈的突破**
 - 流水线执行的指令来源于多条指令流
- **其他：“发射”指每个cycle可进入功能部件（执行部件）的指令条数**



指令流水线技术起源

- **基本思想源于：**
 - 工业上的一种生产方式：流水线又称装配线。指每一个生产单位只专注处理某一个片段的工作，以提高工作效率及产量
 - 1769年，英国人乔赛亚·韦奇伍德开办埃特鲁利亚陶瓷工厂，在场内实行精细的劳动分工，把原来由一个人从头到尾完成的制陶流程分成几十道专门工序，分别由专人完成。
 - 另一说法：20世纪初亨利·福特发明了流水线装配工艺
- **指令流水线起源于IBM7030 (Stretch) , 1961年发布 (1956-61)**
 - Stretch是IBM7030的别名，希望在当时的计算机技术上有大踏步的创新（性能是IBM704的100倍）
 - 四级流水
- **20世纪60年代早期开发的CDC6600，还引入了一些流水线方面增强技术**
- **70年代末和80年代初的有关流水线的术语和描述，涵盖了简单流水线中使用的基本技术。**
- **RISC机器设计最初考虑：易于流水线的实现**
 - 第一个RISC项目是在70年代末和80年代初来自IBM、斯坦福和加州大学伯克利分校。IBM 801, Stanford MIPS和Berkeley RISC 1和RISC 2
- **Intel CPU i486 (1989) 引入5级流水线**

采用流水线技术实现ISA的思想，早于RISC



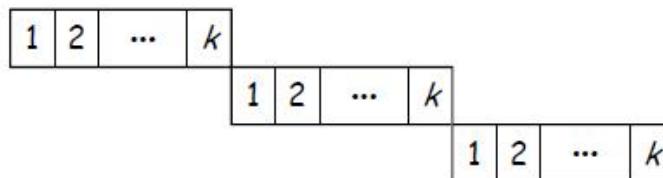
流水线的基本概念

- **一个任务可以分解为 k 个子任务**

- K 个子任务在 K 个不同阶段（**使用不同的资源**）运行
- 每个子任务执行需要 1 个单位时间
- 整个任务的执行时间为 K 倍单位时间

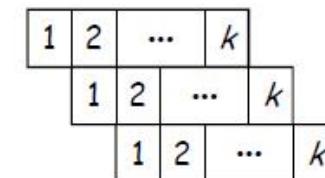
- **流水线执行模式是重叠执行模式**

- K 个流水段并行执行 K 个不同任务
- 每个单位时间进入/离开流水线一个任务



Serial Execution

One completion every k time units

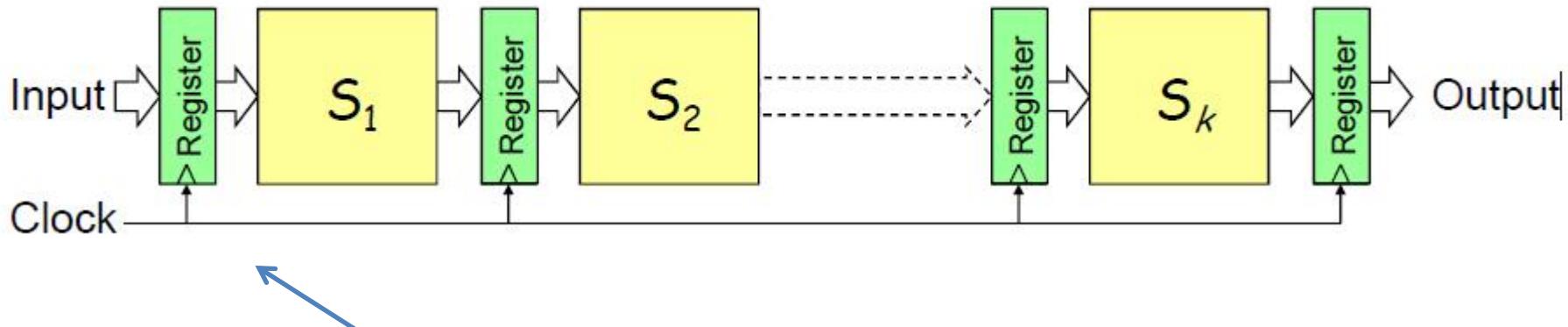


Pipelined Execution

One completion every 1 time unit

同步流水线

- 流水段之间采用时钟控制的寄存器文件 (clocked registers)
- 时钟上升沿到达时...
 - 所有寄存器同时保存前一流水段的结果
- 流水线设计中希望各段相对平衡
 - 即所有段的延迟时间大致相等
- 时钟周期取决于延迟最长的流水段



为什么要插入段间寄存器文件?



流水线的性能

- 设 τ_i = time delay in stage $S_i, i=1..k$
- 时钟周期 $\tau = \max(\tau_i)$ 为最长的流水段延迟
- 时钟频率 $f = 1/\tau = 1/\max(\tau_i)$
- 流水线可以在 $k+n-1$ 个时钟周期内完成 n 个任务
 - 完成第一个任务需要 k 个时钟周期
 - 其他 $n-1$ 个任务需要 $n-1$ 个时钟周期完成
- K-段流水线的理想加速比（相对于串行执行）

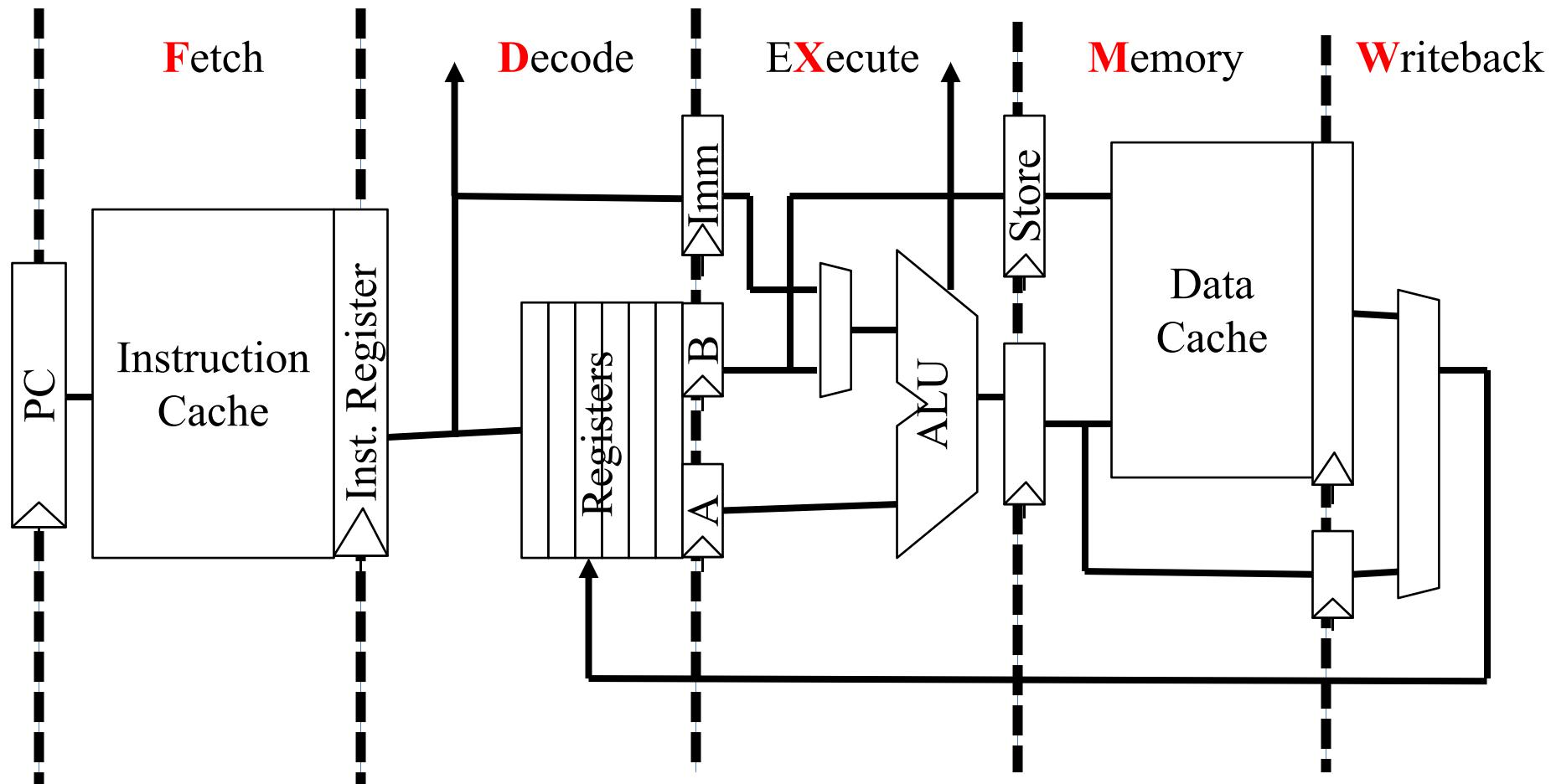
$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k + n - 1} \quad S_k \rightarrow k \text{ for large } n$$



典型的RISC 5段指令流水线

- **5个流水段，每段的延迟为1个cycle**
- **IF: 取值阶段**
 - 选择地址：下一条指令地址、转移地址
- **ID: 译码阶段**
 - 确定控制信号 并从寄存器文件中读取寄存器值
- **EX: 执行**
 - Load、Store：计算有效地址
 - Branch：计算转移地址并确定转移方向
- **MEM: 存储器访问（仅Load和Store）**
- **WB: 结果写回**

典型的 RISC 5段流水线



*This version designed for regfiles/memories
with synchronous reads and writes.*

流水线的可视化表示

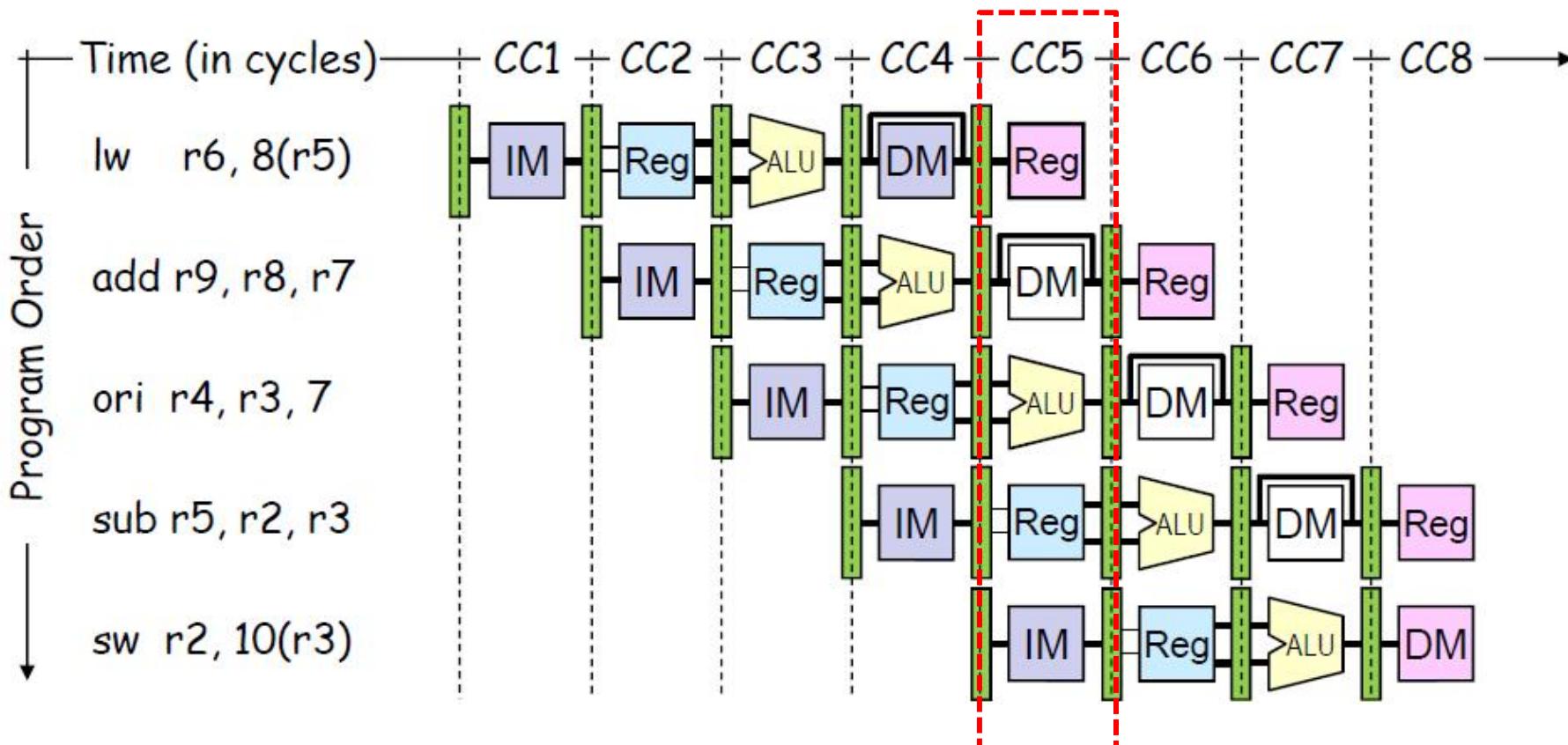
- 多条指令执行多个时钟周期**

- 指令按程序序从上到下排列
- 图中展示了每一时钟周期资源的使用情况
- 不同指令相邻阶段之间没有干扰

基本假设：

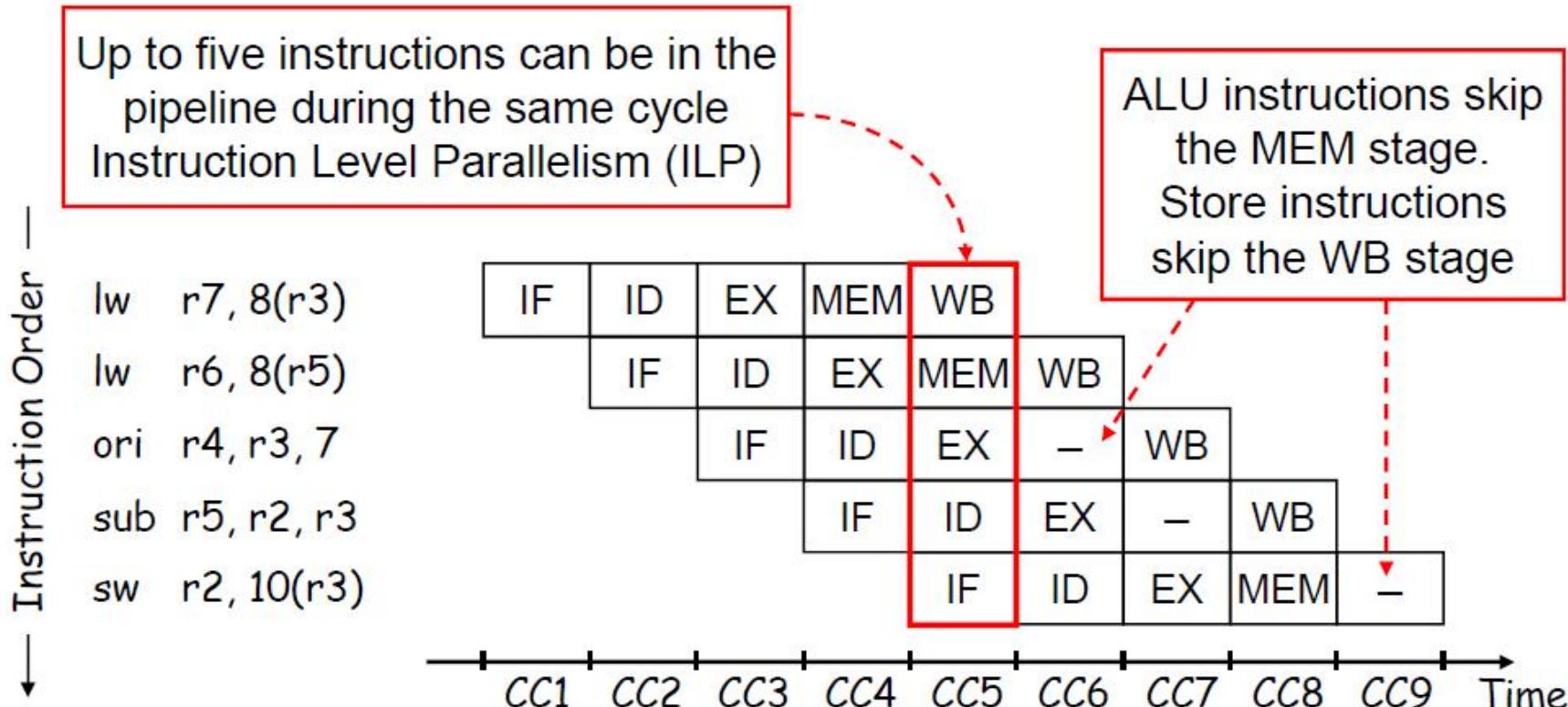
- 存储器分为IM和DM
- 寄存器文件的写-读同一寄存器时，将要写的值直接传递

Why?



指令流时序

- 时序图展示：
 - 每个时钟周期指令所使用的流水段情况
- 指令流在采用5段流水线执行模式的执行情况





单周期、多周期、流水线控制性能比较

- 假设5段指令执行流水线

Instruction	Fetch	Reg Read	ALU	Memory	Reg Wr	Time
Load	350 ps	250 ps	350 ps	350 ps	250 ps	1550 ps
Store	350 ps	250 ps	350 ps	350 ps		1300 ps
ALU	350 ps	250 ps	350 ps		250 ps	1200 ps
Branch	350 ps	250 ps	350 ps			950 ps

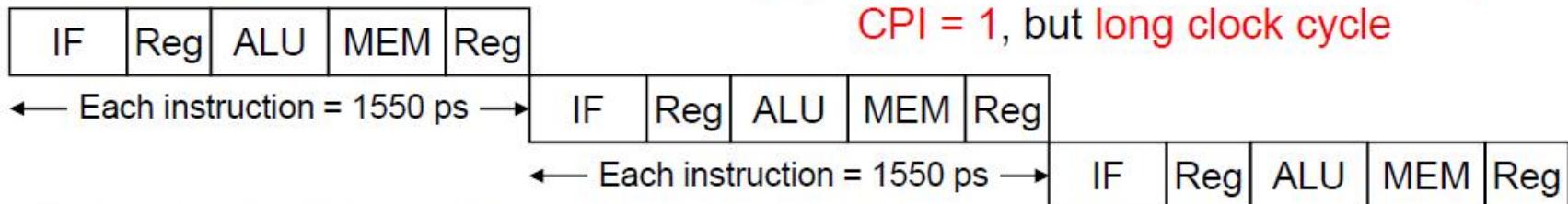
- 某一程序段假设：
 - 20% load, 10% store, 40% ALU, and 30% branch
- 比较三种执行模式的性能

单周期、多周期、流水线控制性能比较

Single-Cycle Execution:

$$T_{clock} = 350 + 250 + 350 + 350 + 250 = 1550 \text{ ps}$$

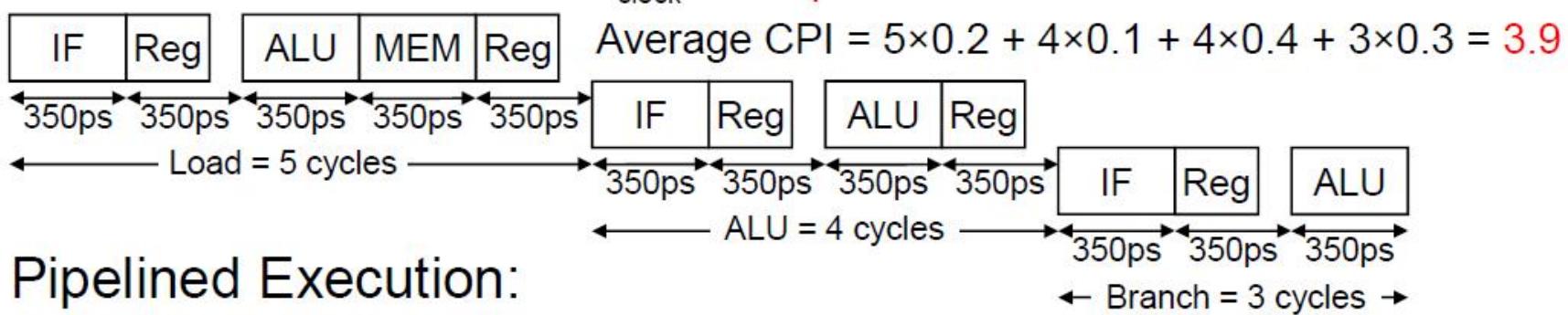
CPI = 1, but long clock cycle



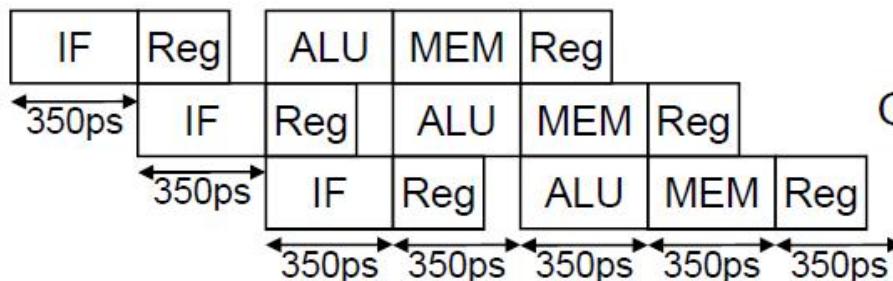
Multi-Cycle Execution:

$$T_{clock} = 350 \text{ ps}$$

Each instruction = 1550 ps



Pipelined Execution:



$$T_{clock} = 350 \text{ ps} = \max(350, 250)$$

One instruction completes each cycle

$$\text{Average CPI} = 1$$

Ignore time to fill pipeline



流水线技术要点

- **流水线的基本概念**

- 多个任务重叠（并发/并行）执行，但使用不同的资源
 - 流水线技术提高整个系统的吞吐率，不能缩短单个任务的执行时间
 - 其潜在的加速比 = 流水线的级数

- **流水线正常工作的基本条件**

- 增加寄存器文件保存当前段传送到下一段的数据和控制信息
 - 需要更高的存储器带宽

- **流水线的相关 (hazards)问题**

- 由于存在相关(hazards)问题，会导致流水线停顿
 - Hazards 问题：流水线的执行可能会导致对资源的访问冲突，或破坏对资源的访问顺序（相对于串行执行指令流：顺序发射-顺序执行-顺序完成）

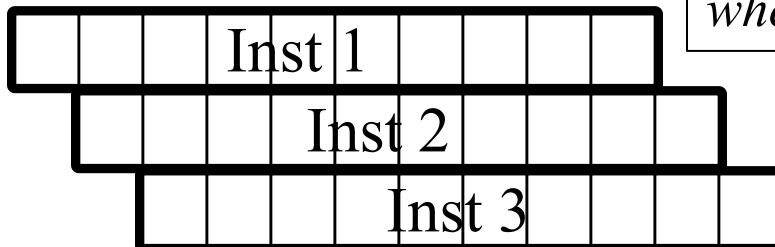


流水线的相关 (Hazards)

- **结构相关**: 流水线中一条指令可能需要另一条指令使用的资源
- **数据和控制相关**: 一条指令可能依赖于先前的指令生成的内容
 - 数据相关: 依赖先前指令产生的结果 (数据) 值
 - 控制相关: 依赖关系是如何确定下一条指令地址 (branches, exceptions)
- **处理相关的一般方法是插入bubble, 导致 $CPI > 1$ (单发射理想CPI)**

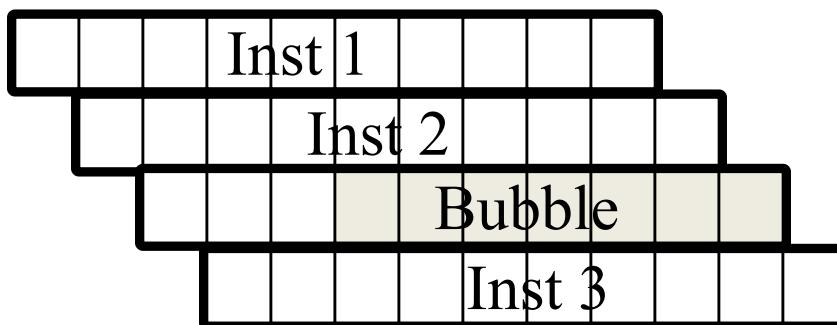
Pipeline CPI Examples

Time →

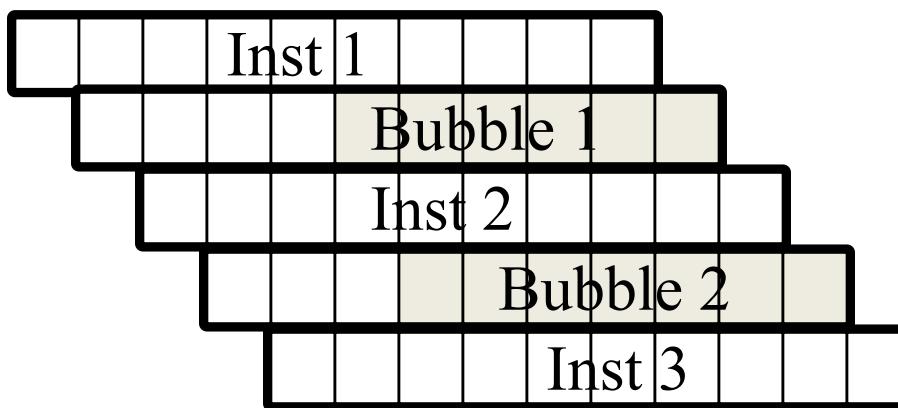


Measure from when first instruction finishes to when last instruction in sequence finishes.

3 instructions finish in 3 cycles
 $CPI = 3/3 = 1$



3 instructions finish in 4 cycles
 $CPI = 4/3 = 1.33$



3 instructions finish in 5cycles
 $CPI = 5/3 = 1.67$



消减结构相关

- **当两条指令同时需要相同的硬件资源时，就会发生结构相关（冲突）**
 - 方法1：通过将新指令延迟到前一条指令执行完（释放资源后）执行
 - 方法2：增加新的资源
 - E.g., 如果两条指令同时需要操作存储器，可以通过增加到两个存储器操作端口来避免结构冲突
- **经典的 RISC 5-段整型数流水线通过设计可以没有结构相关**
 - 很多RISC实现在（执行阶段）**多周期操作**时存在结构相关
 - 例如多周期操作的multipliers, dividers, floating-point units等，由于没有多个寄存器文件写端口 导致 结构冲突



三种基本的数据相关

- **写后读相关(Read After Write (RAW))**

- 由于实际的数据交换需求而引起的

I: add x_1, x_2, x_3
J: sub x_4, x_1, x_3

- **读后写相关 (Write After Read (WAR))**

I: sub x_4, x_1, x_3
J: add x_1, x_2, x_3

- 编译器编写者称之为“anti-dependence”（反相关），是由于重复使用寄存器名“ x_1 ”引起的。

- **写后写相关 (Write After Write (WAW))**

I: sub x_1, x_4, x_3
J: add x_1, x_2, x_3

- 编译器编写者称之为“output dependence”，也是由于重复使用寄存器名“ x_1 ”引起的。
 - 在后面的复杂的流水线中我们将会看到 WAR 和 WAW 相关

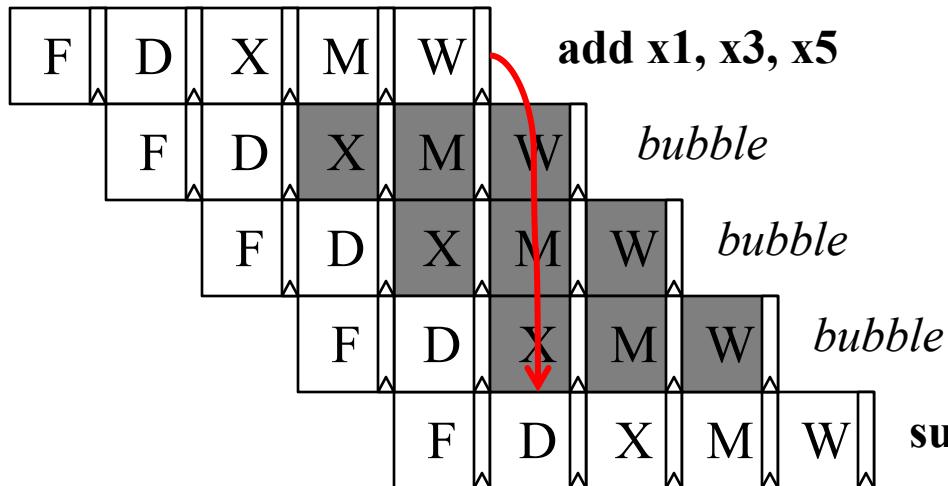


消减数据相关的三种策略

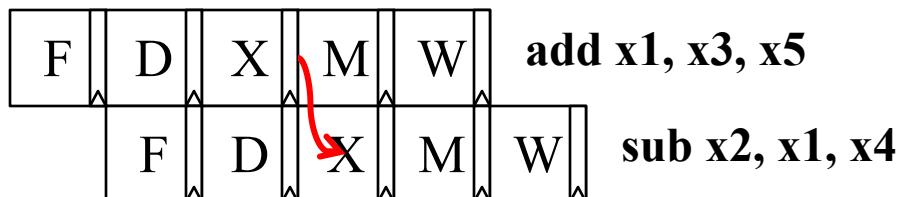
- **联锁机制 (Interlock)**
 - 在issue阶段保持当前相关指令，等待相关解除
- **设置旁路定向路径 (Bypass or Forwarding)**
 - 只要结果可用，通过旁路尽快传递数据
- **投机 (Speculate)**
 - 猜测一个值继续，如果猜测错了再更正

Interlocking Versus Bypassing

add x1, x3, x5
sub x2, x1, x4

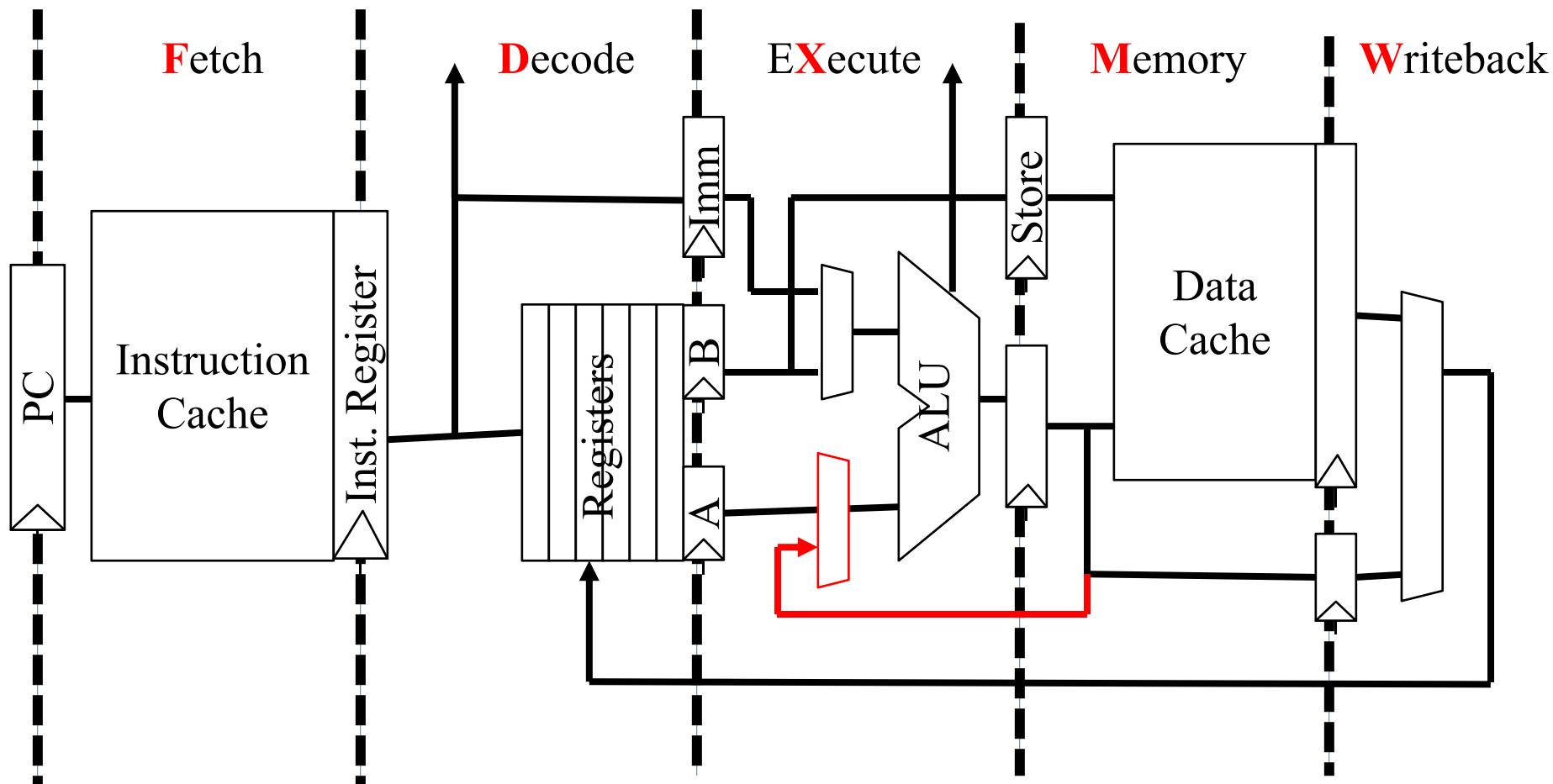


Instruction interlocked
in decode stage

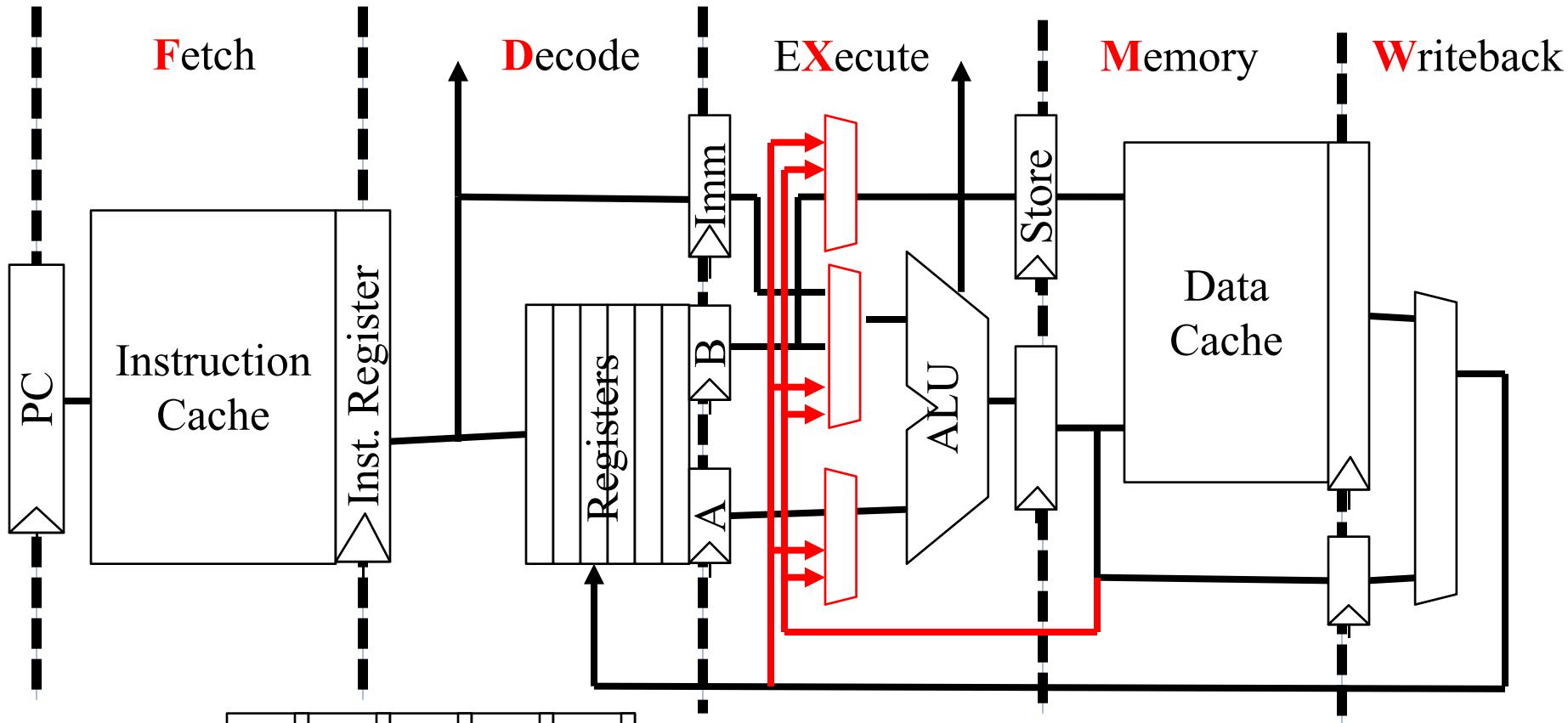


Bypass around ALU
with no bubbles

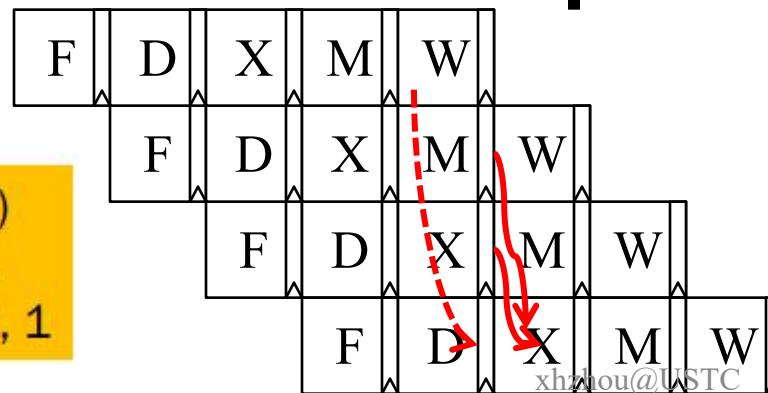
Example Bypass Path



Fully Bypassed Data Path



```
Iw $t0, 4($t1)
addi $t1, $t1
addi $t0, $t0, 1
```



[Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible.]



针对数据相关的值猜测执行

- 不等待产生结果的指令产生值，直接猜测值继续
- 这种技术，仅在某些情况下可以使用：
 - 分支预测
 - 堆栈指针更新
 - 存储器地址消除歧义 (Memory address disambiguation)



采用软件方法避免数据相关

Try producing fast code for

$$a = b + c;$$

$$d = e - f;$$

assuming a, b, c, d ,e, and f in
memory.

Slow code:

LW	Rb,b
LW	Rc,c
ADD	Ra,Rb,Rc
SW	a,Ra
LW	Re,e
LW	Rf,f
SUB	Rd,Re,Rf
SW	d,Rd

Fast code:

LW	Rb,b	LW	Rb,b
LW	Rc,c	LW	Re,e
ADD		ADD	Ra,Rb,Rc
LW		LW	Rf,f
SW		SW	a,Ra
LW		SUB	Rd,Re,Rf
SUB		nop	
SW		SW	d,Rd

假设:

- 1、使用Load的结果中间需要至少1条指令的间隔
- 2、运算的结果，写入内存至少需要1条指令的间隔



review

• 流水线技术要点

- 多个任务重叠（并发/并行）执行，但使用不同的资源
- 流水线技术提高整个系统的吞吐率，不能缩短单个任务的执行时间
- 其潜在的加速比 = 流水线的级数
- 由于存在相关(hazards)问题，会导致流水线停顿
 - Hazards 问题：流水线的执行可能会导致资源访问冲突，或破坏对资源的访问顺序

• 指令流水线

- 通过指令重叠减小 CPI
- 充分利用数据通路
 - 当前指令执行时，启动下一条指令
 - 其性能受限于花费时间最长的段
- 检测和消除相关
- 正常工作的基本条件
 - 增加寄存器文件保存当前段传送到下一段的数据和控制信息
 - 需要更高的存储器带宽



Control Hazards

如何计算下一条指令地址 (next PC)

- **无条件直接转移**
 - Opcode, PC, and offset
- **基于基址寄存器的无条件转移**
 - Opcode, Register value, and offset
- **条件转移**
 - Opcode, Register (for condition), PC and offset
- **其他指令**
 - Opcode and PC (and have to know it's not one of above)

Control flow information in pipeline

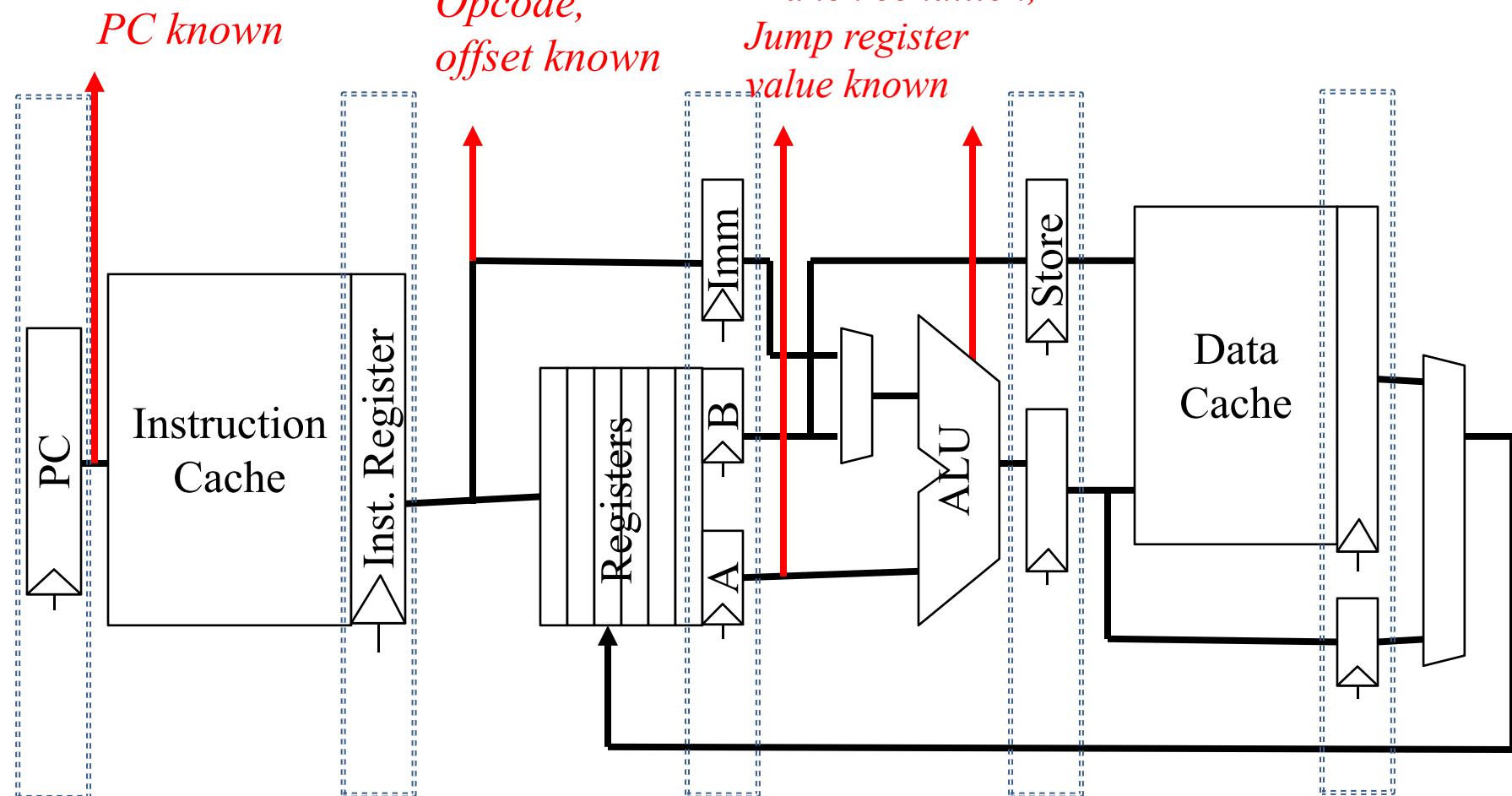
Fetch

Decode

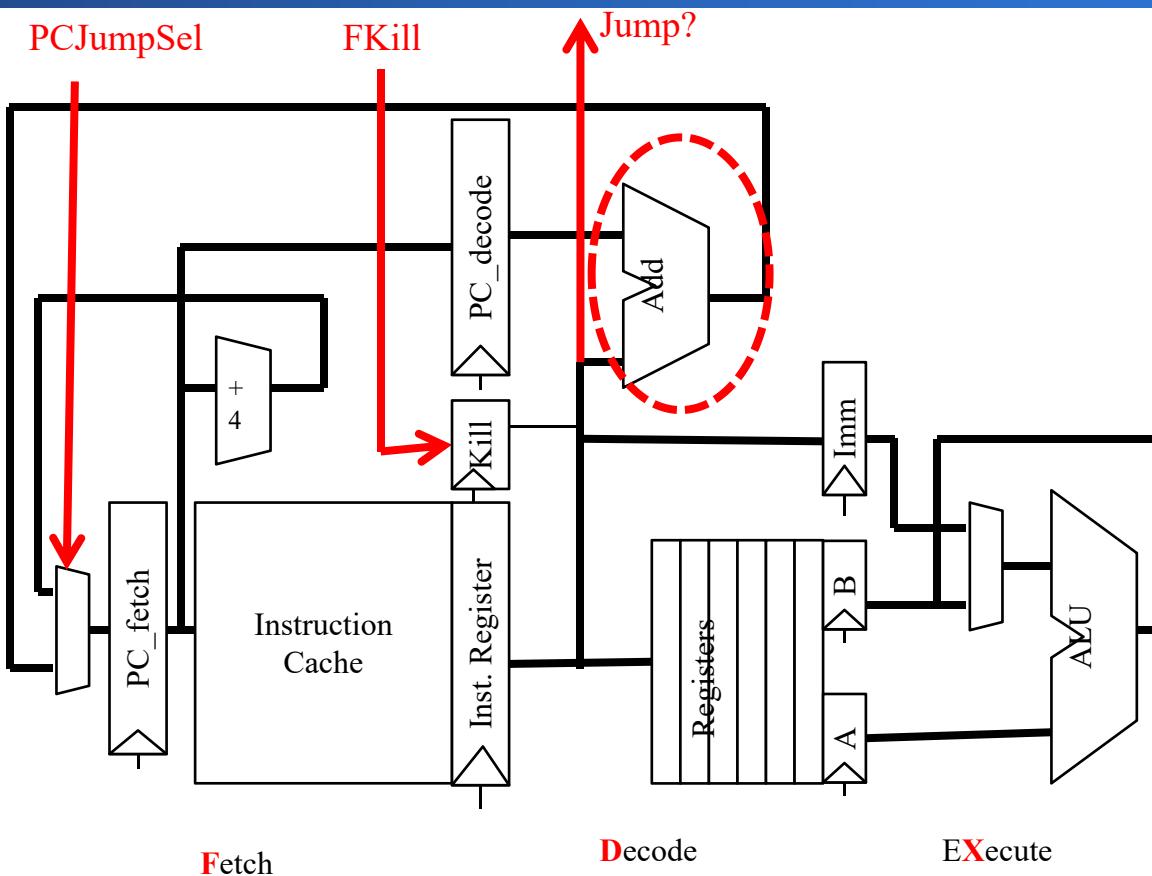
EXecute

Memory

Writeback

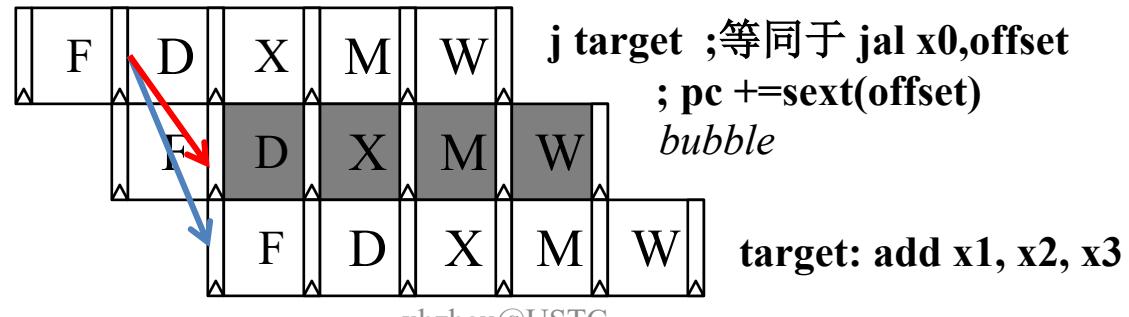


RISC-V Unconditional PC-Relative Jumps



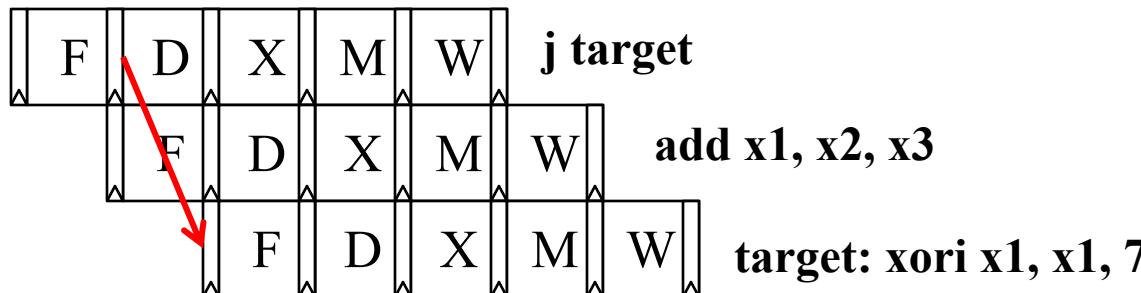
改进的数据通路

[Kill bit turns instruction into a bubble]



Branch Delay Slots

- 早期的RISC机器的延迟槽技术—改变ISA语义，在分支/跳转后的**延迟槽**中指令总是在控制流发生变化之前执行：
 - 0x100 j target
 - 0x104 add x1, x2, x3 // Executed before target
 - ...
 - 0x205 target: xori x1, x1, 7
- 软件必须用有用的工作填充延迟槽（delay slots），或者用显式的NOP指令填充延迟槽





Post-1990 RISC ISAs 取消了延迟槽

- **性能问题**
 - 当延迟槽中填充了NOPs指令后，增加了I-cache的失效率
 - 即使延迟槽中只有一个NOP， I-cache失效导致机器等待
- **使先进的微体系架构复杂化**
 - 例如4发射30段流水线
- **分支预测技术的进步减少了采用延迟槽技术的动力**



小结：解决控制相关的方法

- #1: Stall 直到分支方向确定
 - 可通过修改数据通路，减少stall
- #2: 预测分支失败
 - 直接执行后继指令
 - 如果分支实际情况为分支成功，则撤销流水线中的指令对流水线状态的更新
 - 要保证：分支结果出来之前不会改变处理机的状态，以便一旦猜错时，处理机能够回退到原先的状态。
- #3: 预测分支成功
 - 前提：先知道分支目标地址，后知道分支是否成功
- #4: 延迟转移技术



流水线的加速比计算

$$CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$



Exception、Interrupt和Trap

- 广义上的异常机制：处理器在执行指令流的过程中遇到异常事件而中止执行当前程序，转而去处理**异常事件（特殊事件）**
- 异常事件：内部异常事件和外部请求事件
- 根据异常事件的来源可分为：
 - 内部异常 (Exception)：处理器内部事件或程序执行过程中的事件引起的异常
 - 例如：硬件故障、Page fault、arithmetic underflow等
 - 中断 (Interrupt)：处理器外部请求事件引起的异常
- 根据异常事件发生在程序执行中的位置，可分为：
 - 同步异常：在每次执行时异常事件发生在相同位置。即异常原因可精确定位到某指令
 - 异步异常：在每次执行时异常事件发生在不同位置。如定时器过期、I/O设备中相关事件的信号、硬件故障、电源故障等等，不能够被精确定位到某条指令
- 根据异常处理后是否能够精确地返回到引起异常的指令，可分为：
 - 精确异常和非精确异常
- **陷阱 (Trap)**：因执行指令引起异常 (Exception) 情况而被迫将控制权移交给管理员环境 (监控程序，运行级别的变化)
 - Not all exceptions cause traps (c.f. IEEE 754 floating-point standard)
- **中断响应：响应（处理）外部请求事件**，导致控制权转移到管理员环境
- **陷阱和中断响应通常由同样的流水线机制处理**



异常处理

• 广义的异常：异常和中断

- 异常：程序运行过程中产生的**内部事件**，称为异常事件，因异常事件将控制权转移到监控程序，称为陷阱（Trap）
- 中断：响应（处理）程序之外的**外部事件**，导致控制权转移到监控程序

• 同步异常 vs. 异步异常

- 同步：在每次执行时异常事件发生在相同位置
- 异步：在每次执行时异常事件可能发生在不同位置

• 精确异常 vs. 非精确异常

- 精确：响应异常后，可精确返回引起异常的指令位置
- 非精确：响应异常后，无法精确返回引起异常的指令位置

• 异常处理：

- 异常处理的时机：指令commit阶段 即最后完成阶段
- 优先级：外部事件引起的异常，内部事件引起的异常（同一条指令按时间顺序）
- 过程：保存现场、处理、恢复现场



小结

• 流水线技术要点

- 多个任务重叠（并发/并行）执行，但使用不同的资源
- 流水线技术提高整个系统的吞吐率，不能缩短单个任务的执行时间
- 其潜在的加速比 = 流水线的级数
- 由于存在相关(hazards)问题，会导致流水线停顿
 - Hazards 问题：流水线的执行可能会导致资源访问冲突，或破坏对资源的访问顺序
- **由于存在异常问题，会影响流水线的执行**

• 指令流水线

- 通过指令重叠减小 CPI
- 充分利用数据通路
 - 当前指令执行时，启动下一条指令
 - 其性能受限于花费时间最长的段
- 检测和消除相关
- 正常工作的基本条件
 - 增加寄存器文件保存当前段传送到下一段的数据和控制信息
 - 需要更高的存储器带宽



小结

- **流水线的性能评估**

- 实际吞吐率：假设 k 段，完成 n 个任务，单位时间所实际完成的任务数。
 - 加速比： k 段流水线的速度与等功能的非流水线的速度之比。
 - 效率：流水线的设备利用率

- **影响流水线性能的因素**

- 流水线中的瓶颈——最慢的那一段
 - 流水段所需时间不均衡将降低加速比
 - 流水线存在装入时间和排空时间，使得加速比降低

- **如何有利于流水线技术的应用**

- 指令格式简单
 - 固定长度、很少的指令格式
 - 只用Load/Store来进行存储器访问

- **基本5段流水线执行模式：顺序发射、顺序执行、顺序完成**



Part1：指令级并行

- **指令级并行：提高程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - 浮点数操作打破了基本流水线平衡。怎么办？
 - **流水线的扩展：指令执行周期数(可能) 不同**
- **多发射流水线**
 - 每个cycle发射多条指令执行
- **多线程技术**
 - 流水线执行的指令来源于多条指令流
- **分支预测技术**
 - 提高ILP
- **其他问题**



Part1：指令级并行

- **指令级并行：提高单条指令流（单线程）程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - **流水线的扩展：指令执行周期数(可能) 不同**
 - 指令级并行优化：编译优化和硬件优化
 - 动态调度的“漏网之鱼”——控制相关与异常处理问题
 - 分支预测技术
 - 推断执行技术
- **多发射流水线：ILP的突破 ($CPI < 1$)**
 - 每个cycle发射多条指令执行
- **多线程技术：单线程ILP瓶颈的突破**
 - 流水线执行的指令来源于多条指令流
- **其他：“发射”指每个cycle可进入功能部件（执行部件）的指令条数**



多周期操作的处理

- **问题**

- 目前浮点操作还不能像整型操作在1 ~ 2个cycles完成，一般要花费较长时间

如何处理？

- **在1到2个cycles时间内完成的处理方法**

- 采用较慢的时钟源，或
 - 在FP部件中延迟其EX段

- **现假设FP指令与整数指令采用相同的流水线，那么**

- EX 段需要循环多次来完成FP操作，循环次数取决于操作类型
 - 有多个FP功能部件，如果发射出的指令导致结构或数据相关，需暂停

- **执行阶段(EX)分为多个周期 (多段)**



支持浮点数操作的流水线

- 与基本流水线的显著区别
 - EX段是多周期的，**不同操作EX段的周期数不同**
 - EX段周期数不同→**乱序完成**→WAR相关
- 增加流水线级数在提高性能的同时，会增加相关产生的可能性
- 浮点运算使得流水线控制更加复杂
 - 两个重要参数：Latency & Repeat Interval
- 问题：
 - 结构相关（增多）；
 - 数据相关、控制相关引起的stall增多；
 - 有新的冲突源产生；定向路径增多；异常处理复杂
 - 以MIPS 扩展为例
- MIPS R4000 8级整型数流水线
 - 存储器操作分阶段 – load延迟为2个cycles
 - Branch操作在EX段确定分支方向- 3个cycles的延迟
 - 多个定向源： EX/DF, DF/DS, DS/TC, TC/WB
- MIPS R4000的浮点数操作

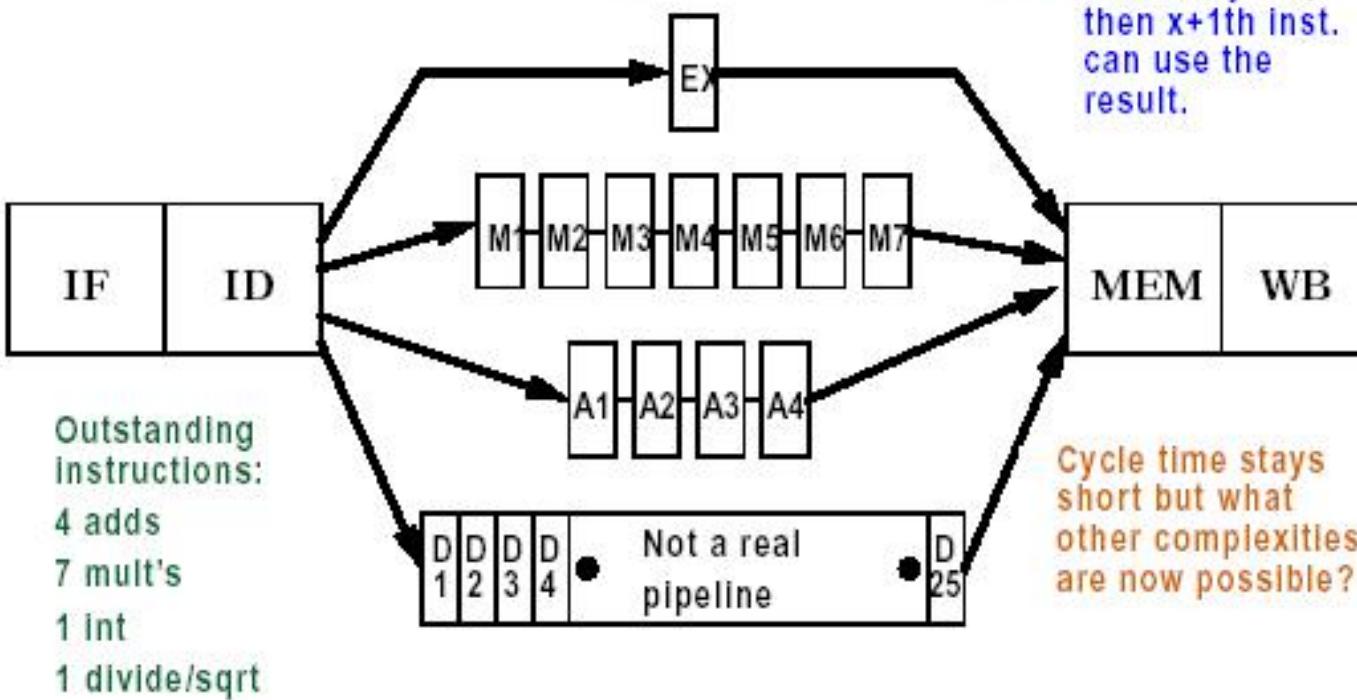


以MIPS为例：对MIPS的扩充

- **四个功能部件**
- **Integer 部件处理**
 - Loads, Store, Integer ALU操作和Branch
- **FP/Integer 乘法部件**
 - 处理浮点数和整数乘法
- **FP加法器**
 - 处理FP加, 减和类型转换
- **FP/Integer除法部件**
 - 处理浮点数和整数除法
- **这些功能部件未流水化**

将部分执行部件流水化后的MIPS流水线

Note # of stages are $1 + \text{latency}_{\text{EX}}$



部件流水化后，执行阶段周期数不同产生的新问题？

Function Unit	Latency	Repeat Interval
Integer ALU	0	1
Data Memory (Integer and FP loads(1 less for store latency))	1	1
FP Add	3	1
FP multiply	6	1
FP Divide (also integer divide and FP sqrt)	24	25



新的相关和定向问题

- **结构冲突增多**
 - 非流水的Divide部件，使得EX段增长24个cycles
- **EX段不同周期数**
 - 在一个周期内可能有多个寄存器写操作
 - 可能指令**乱序完成**（乱序到达WB段）有可能存在**WAW**
- **由于在ID段读，还不会有 WAR 相关**
- **乱序完成导致异常处理复杂**
- **由于指令的延迟加大导致RAW相关的stall数增多**
- **需要付出更多的代价来增加定向路径**

顺序发射、顺序执行、乱序完成



新的结构相关

Instruction	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
LD F8, 0(R2)							IF	ID	EX	MEM	WB

- 纵向检查指令所使用的资源
 - 第10个cycle，三条指令同时进入MEM，但由于MULTD和ADDD在MEM段没有实际动作，这种情况没有关系
 - 第11个cycle，三条指令同时进入WB段，存在结构相关



解决方法

- **Option 1**

- 在ID段跟踪写端口的使用情况，以便能暂停该指令的发射
 - 一旦发现冲突，暂停当前指令的发射

- **Option 2**

- 在进入MEM或WB段时，暂停冲突的指令，让有较长延时的指令先做。这里假设较长延时的指令，可能会更容易引起其他RAW相关，从而导致更多的stalls



新的冲突源

- **GPR与FPR间的数据传送造成的数据相关**
 - MOVI2FP and MOVFP2I instructions
- **如果在ID段进行相关检测，指令发射前须做如下检测：**
 - 结构相关
 - 循环间隔检测
 - 确定寄存器写端口是否可用
 - RAW相关
 - 列表所有待写的目的寄存器
 - 不发射以待写寄存器做为源寄存器的指令，插入latency个stall
 - WAW相关
 - 仍然使用上述待写寄存器列表
 - 不发射那些目的寄存器与待写寄存器列表中的指令有WAW冲突的指令，延迟1个cycle发射。



MIPS中的异常

- **IF**
 - page fault, misaligned address, memory protection violation
- **ID**
 - undefined or illegal opcode
- **EX**
 - arithmetic exception
- **MEM**
 - page fault, misaligned address, memory protection violation
- **WB**
 - none



处理异常的4种可能的办法

- **方法1：忽略这种问题，当非精确处理**
 - 原来的supercomputer的方法
 - 但现代计算机对IEEE 浮点标准的异常处理，虚拟存储的异常处理要求必须是精确异常。
- **方法2：缓存操作结果，直到早期发射的指令执行完。**
 - 当指令运行时间较长时，Buffer区较大
 - Future file (Power PC620 MIPS R10000)
 - 缓存执行结果，按指令序确认
 - history file (CYBER 180/990)
 - 尽快确认
 - 缓存区存放原来的操作数，如果异常发生，回卷到合适的状态



第3 & 4种方法

- **方法3：以非精确方式处理，用软件来修正**

- 为软件修正保存足够的状态
 - 让软件仿真尚未执行完的指令的执行
 - 例如
 - Instruction 1 – A 执行时间较长，引起中断的指令
 - Instruction 2, instruction 3,instruction n-1 未执行完的指令
 - Instruction n 已执行完的指令
 - 由于第n条指令已执行完，希望中断返回后从第n+1条指令开始执行，如果我们保存所有的流水线的PC值，那么软件可以仿真Instruction1 到Instruction n-1 的执行

- **方法4：暂停发射，直到确定先前的指令都可无异常的完成，再发射下面的指令。**

- 在EX段的前期确认（MIPS流水线在前三个周期中）
 - MIPS R2K to R4K 以及Pentium使用这种方法



Part1：指令级并行

- **指令级并行：提高单条指令流（单线程）程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - 流水线的扩展：指令执行周期数(可能) 不同
 - **指令级并行优化：编译优化和硬件优化**
 - 动态调度的“漏网之鱼”——控制相关与异常处理问题
 - 分支预测技术
 - 推断执行技术
- **多发射流水线：ILP的突破 (CPI<1)**
 - 每个cycle发射多条指令执行
- **多线程技术：单线程ILP瓶颈的突破**
 - 流水线执行的指令来源于多条指令流
- **其他：这里的“发射”指流水线每个cycle所取的指令条数**



提高指令级并行 (ILP)

- **软件方法：**
 - 循环展开
 - 静态指令流调度（指令重排）
 -
- **动态指令流调度**
 - 记分牌和基本Tomasulo算法



回顾: 基本流水线

- **流水线提高的是指令带宽（吞吐率）, 而不是单条指令的执行速度**
- **相关限制了流水线性能的发挥**
 - 结构相关: 需要更多的硬件资源
 - 数据相关: 需要定向, 编译器调度
 - 控制相关: 尽早检测条件和计算目标地址, 延迟转移, 预测
- **增加流水线的级数会增加相关产生的可能性**
- **异常, 浮点运算使得流水线控制更加复杂**
- **编译器可降低数据相关和控制相关的开销**
 - Load 延迟槽
 - Branch 延迟槽
 - Branch预测



指令级并行的基本概念及挑战

- ILP: (无关的) 指令重叠 (并行) 执行
- 流水线的平均CPI

Pipeline CPI = Ideal Pipeline CPI

- + Struct Stalls
- + RAW Stalls + WAR Stalls + WAW Stalls
- + Control Stalls
- + Memory Stalls

- **减少停顿 (stalls)数的基本途径**

- 软件方法：静态指令流调度 (指令重排)
 - Gcc: 17%控制类指令, 5 instructions + 1 branch;
 - 在基本块上, 得到更多的并行性
 - 挖掘循环级并行
- 硬件方法：动态指令流调度方法
- 以MIPS的浮点数操作为例



采用的基本技术

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	A.2
Delayed branches and simple branch scheduling	Control hazard stalls	A.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	A.8
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences	3.2
Dynamic branch prediction	Control stalls	3.4
Issuing multiple instructions per cycle	Ideal CPI	3.6
Speculation	Data hazard and control hazard stalls	3.5
Dynamic memory disambiguation	Data hazard stalls with memory	3.2, 3.7
Loop unrolling	Control hazard stalls	4.1
Basic compiler pipeline scheduling	Data hazard stalls	A.2, 4.1
Compiler dependence analysis	Ideal CPI, data hazard stalls	4.4
Software pipelining, trace scheduling	Ideal CPI, data hazard stalls	4.3
Compiler speculation	Ideal CPI, data, control stalls	4.4



本章遵循的指令延时

产生结果的指令	使用结果的指令	所需延时
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- 当使用结果的指令为BRANCH指令时除外
- FP 操作 浮点加减运算



循环展开+静态指令流调度

- **优化对象（热点代码）：程序中的循环**
- **基本块的定义：**
 - 直线型代码，无分支；单入口；程序由分支语句连接基本块构成
- **循环级并行**
 - $\text{for } (i = 1; i \leq 1000; i++) \quad x(i) = x(i) + s;$
 - 计算 $x(i)$ 时**没有数据相关**；可以并行产生1000个数据；
 - 问题：在生成代码时会有Branch指令 - **有控制相关**
 - 分支预测相对比较容易
 - 向量处理机模型
 - load vectors x and y (up to some machine dependent max)
 - then do result-vec = $x\text{vec} + y\text{vec}$ in a single instruction



简单循环及其对应的汇编程序

```
for (i=1; i<=1000; i++)  
    x(i) = x(i) + s;
```

Loop:	LD	F0,0(R1)	;F0=vector element
	ADDD	F4,F0,F2	;add scalar from F2
	SD	0(R1),F4	;store result
	SUBI	R1,R1,8	;decrement pointer 8B (DW)
	BNEZ	R1,Loop	;branch R1!=zero
	NOP		;delayed branch slot



FP 循环中的相关

Loop: LD F0,0(R1) ;F0=vector element
ADD D F4,F0,F2 ;add scalar from F2
SD 0(R1),F4 ;store result
SUB I R1,R1,8 ;decrement pointer 8B (DW)
BNEZ R1,Loop ;branch R1!=zero
NOP ;delayed branch slot

产生结果的指令	使用结果的指令	所需延时
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0



FP 循环中的Stalls

1 Loop: LD F0,0(R1) ;F0=vector element
2 stall
3 ADDD F4,F0,F2 ;add scalar in F2
4 stall
5 stall
6 SD 0(R1),F4 ;store result
7 SUBI R1,R1,8 ;decrement pointer 8B (DW)
8 stall ; Notice
9 BNEZ R1,Loop ;branch R1!=zero
10 stall ;delayed branch slot

产生结果的指令	使用结果的指令	所需延时
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

10 clocks: 是否可以通过调整代码顺序使stalls减到最小



FP 循环中的最少Stalls数

1 Loop:	LD	F0,0(R1)
2	SUBI	R1,R1,8
3	ADDD	F4,F0,F2
4	stall	
5	BNEZ	R1,Loop ;delayed branch
6	SD	8(R1),F4 ;altered when move past SUBI

Swap BNEZ and SD by changing address of SD

6 clocks: 通过循环展开4次是否可以提高性能?



循环展开4次(straight forward way)

1	Loop:	LD	F0,0(R1)	stall
2		ADDD	F4,F0,F2	stall stall
3		SD	0(R1),F4	;drop SUBI & BNEZ
4		LD	F6,-8(R1)	stall
5		ADDD	F8,F6,F2	stall stall
6		SD	-8(R1),F8	;drop SUBI & BNEZ
7		LD	F10,-16(R1)	stall
8		ADDD	F12,F10,F2	stall stall
9		SD	-16(R1),F12	;drop SUBI & BNEZ
10		LD	F14,-24(R1)	stall
11		ADDD	F16,F14,F2	stall stall
12		SD	-24(R1),F16	
13		SUBI	R1,R1,#32	stall ;alter to 4*8
14		BNEZ	R1,LOOP	
15		NOP		Rewrite loop to minimize stalls?

15 + 4 x (1+2) + 1 = 28 cycles, or 7 per iteration

Assumes R1 is multiple of 4



Stalls数最小的循环展开

1 Loop:	LD	F0,0(R1)	
2	LD	F6,-8(R1)	
3	LD	F10,-16(R1)	
4	LD	F14,-24(R1)	
5	ADDD	F4,F0,F2	
6	ADDD	F8,F6,F2	
7	ADDD	F12,F10,F2	
8	ADDD	F16,F14,F2	
9	SD	0(R1),F4	
10	SD	-8(R1),F8	
11	SUBI	R1,R1,#32	
12	SD	16(R1),F12	; 16-32 = -16
13	BNEZ	R1,LOOP	
14	SD	8(R1),F16	; 8-32 = -24

• 代码移动后的注意事项

- SD移动到SUBI后，注意偏移量的修改
- LD移动到SD前，注意偏移量的修改

14 clock cycles, or 3.5 per iteration



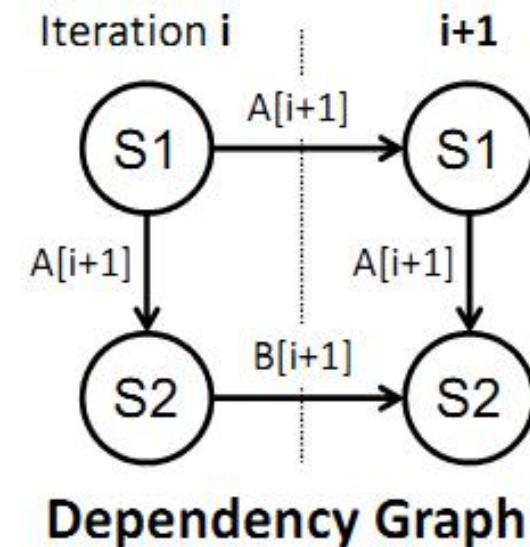
循环展开示例小结

- 循环展开对循环间无关的程序是有效降低stalls的手段(**循环级并行**) .
- **指令调度**, 必须保证程序运行的结果不变
- 注意循环展开中的**Load**和**Store**
 - 不同次循环的Load 和Store 是相互独立的。需要分析对存储器的引用, 保证他们没有引用同一地址. (**检测存储器访问冲突**)
- **不同次的循环, 使用不同的寄存器** (**寄存器重命名**)
- **删除不必要的测试和分支后, 需调整循环步长等控制循环的代码.** (**步长调整**)
- **移动SD到SUBI和BNEZ后, 需要调整SD中的偏移**

循环间相关 (1/4)

Example: 下列程序段存在哪些数据相关?
(A,B,C 指向不同的存储区且不存在覆盖区)

```
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
}
```



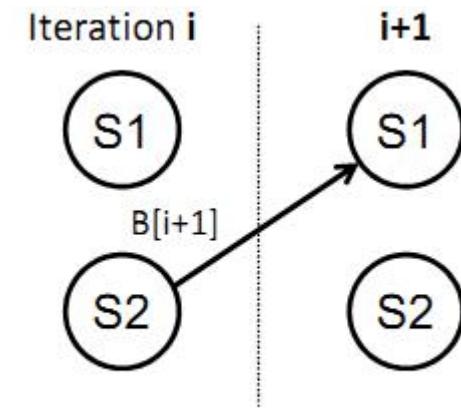
1. S2 使用由 S1 在同一循环计算出的 $A[i+1]$.
 2. S1 依赖于前一次循环的 S1; S2 也依赖于前一次循环的 S2。
- 这种存在于循环间的相关，称为 “loop-carried dependence” 表示循环间存在相关，不能乱序执行，它与我们前面的例子中循环间无关是有区别的

无回路的循环间相关 (2/4)-

Example: A,B,C,D distinct & nonoverlapping
for ($i=1$; $i \leq 100$; $i=i+1$) {

```
A[i] = A[i] + B[i]; /* S1 */  
B[i+1] = C[i] + D[i];} /* S2 */
```

Non-Circular Loop-Carried Dependence



Dependency Graph

1. S1和S2没有相关， S1和S2互换不会影响程序的正确性
2. 在第一次循环中， S1依赖于前一次循环的B[1].
3. S1依赖上一次循环的S2，但S2不依赖S1



循环间相关 (3/4) - 循环变换

OLD:

```
for (i=1; i<=100; i=i+1) {  
    A[i] = A[i] + B[i]; /* S1 */  
    B[i+1] = C[i] + D[i];} /* S2 */
```

NEW:

```
A[1] = A[1] + B[1];  
for (i=1; i<=99; i=i+1) {  
    B[i+1] = C[i] + D[i];  
    A[i+1] = A[i+1] + B[i+1];  
}  
B[101] = C[100] + D[100];
```

循环间无相关



循环间相关 (4/4) -Dependence Distance

通常循环间相关呈现为递推关系

for ($i=1; i < N; i++$) $A[i] = A[i-1] + B[i];$

相关的距离可能大于1

for ($i=4; i < N; i++$) $A[i] = A[i-4] + B[i];$

可以通过循环展开增加循环内的并行性

for ($i=4; i < N; i=i+4$)

$\{ \quad A[i] = A[i-4] + B[i];$

$A[i+1] = A[i-3] + B[i+1];$

$A[i+2] = A[i-2] + B[i+2];$

$A[i+3] = A[i-1] + B[i+3];$

$\}$



循环展开示例小结

- 循环展开对循环间无关的程序是有效降低stalls的手段(**循环级并行**) .
- **指令调度**, 必须保证程序运行的结果不变
- 注意循环展开中的Load和Store
 - 不同次循环的Load 和Store 是相互独立的。需要分析对存储器的引用, 保证他们没有引用同一地址. (**检测存储器访问冲突**)
- 不同次的循环, 使用不同的寄存器 (**寄存器重命名**)
- 删除不必要的测试和分支后, 需调整循环步长等控制循环的代码. (**步长调整**)
- 移动SD到SUBI和BNEZ后, 需要调整SD中的偏移



小结

- **指令级并行(ILP)：流水线的平均CPI**
 - Pipeline CPI = Ideal Pipeline CPI + Struct Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls +.....
 - 提高指令级并行的方法
 - 软件方法：指令流调度，循环展开，软件流水线，trace scheduling
 - 硬件方法
- **软件方法：指令流调度-循环展开**
 - 指令调度，必须保证程序运行的结果不变
 - 寄存器的重命名
 - 循环步长的调整
 - 偏移量的修改
 - **保证存储器访问无冲突**



硬件方案: 指令级并行

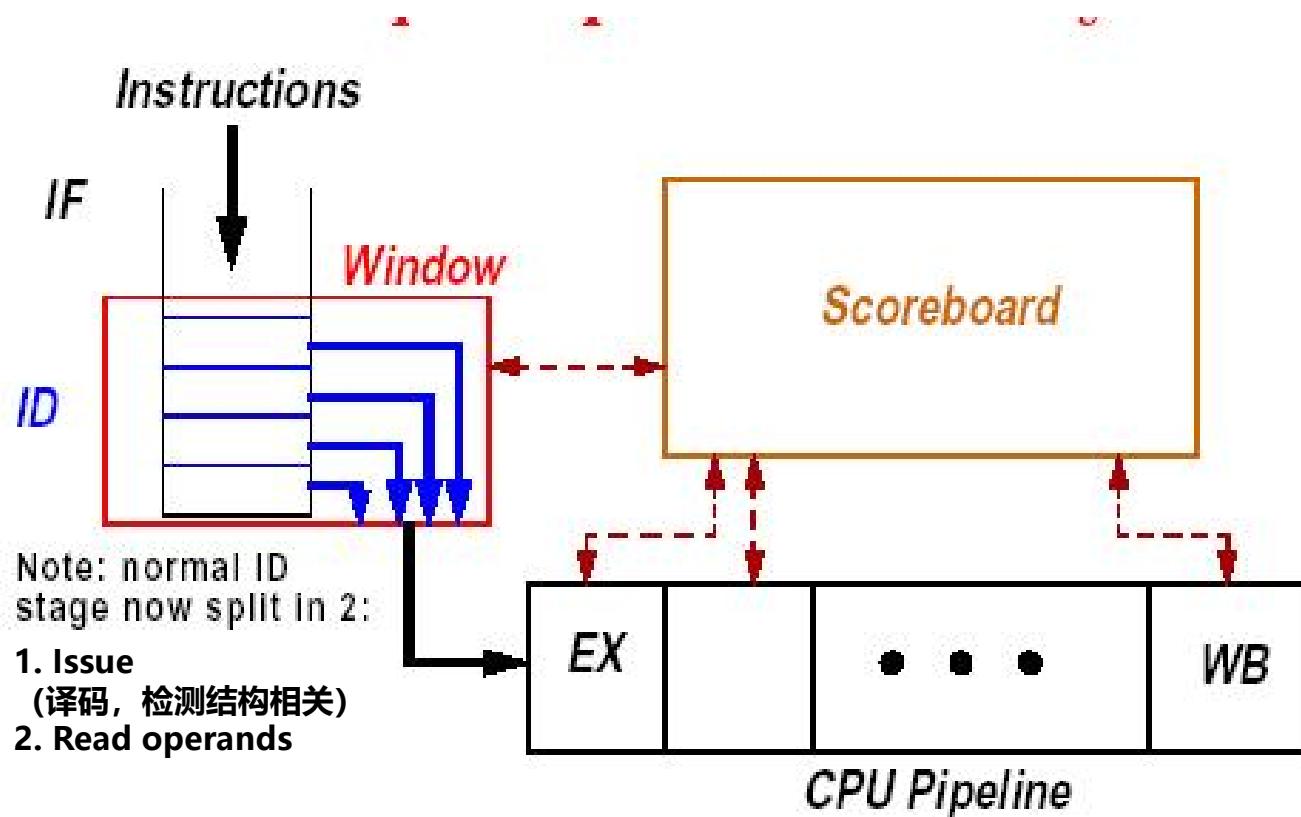
- 为什么要使用硬件调度方案? 突破软件优化的局限
 - 在编译时无法确定的相关, 可以通过硬件调度来优化
 - 编译器简单
 - 代码在不同组织结构的机器上, 同样可以有效的运行
- 基本思想: 允许stall后的指令继续向前流动

DIVD F0,F2,F4
ADDL F10,F0,F8
SUBD F12,F8,F14

- 允许乱序执行 (out-of-order execution) => 乱序完成 (out-of-order completion)
- 经典方法:
 - Scoreboard
 - Tomasulo

硬件方案之一：记分牌

• 记分牌的基本概念示意图

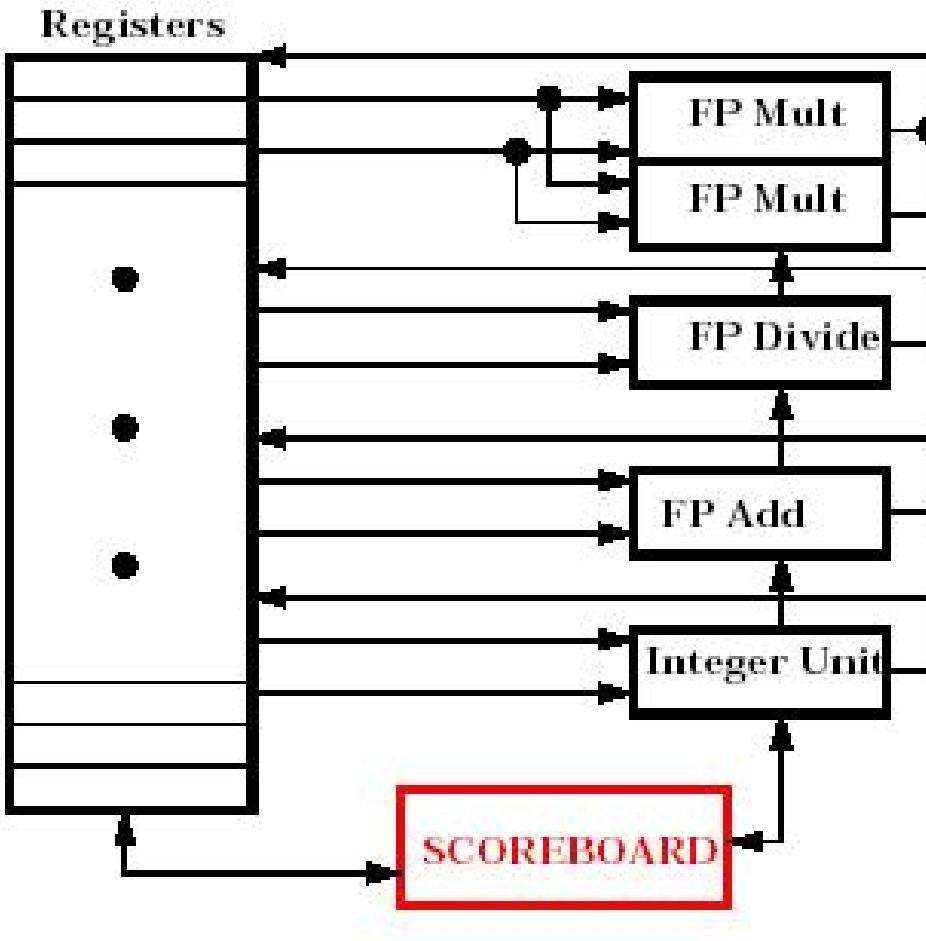




记分牌技术要点(1/2)

- Scordboard技术将ID段分为:
 - Issue—译码, 检测结构相关
 - Read operands—**等待到无数据相关时, 读操作数**
- 起源于1964年Control Data Corporation推出的CDC6600: **顺序发射, 乱序执行, 乱序完成**, 没有采用定向技术, 只实现非精确中断
 - Ten Functional Unit
 - Floating ADD, Floating Multiply(2), Floating Divide
 - Fix ADD, Increment (2), Boolean, Shift, Branch
 - Ten Peripheral Processors for Input/Output: a fast time-shared 12-bit integer ALU
 - Load /store结构
- 集中相关(冲突)检查(**Scoreboard**), 互锁机制(**interlock**)解决相关
 - **Interlock:**一种装置, 其中一个部分或机构的运转自动地导致或阻止另一个部分的运转
- 采用这种技术的微处理器企业
 - MIPS, HP, IBM
 - Sun公司的UltraSparc
 - DEC Alpha

带有记分牌控制的MIPS



Note: this model could support both single or multi-issue

Exception is that one multiply will be issued per cycle

All depends on bus/trunk structure



记分牌技术要点(2/2)

- **Out-of-order completion => WAR, WAW hazards?**
- **WAR:** 一般解决方案 (在不采用寄存器重命名的情况下)
 - 对操作排队
 - 仅在读操作数阶段读寄存器
- **WAW:** 检测到相关后，停止发射前一条指令，直到前一条指令完成
- **提高效率的前提：**需要有多条指令进入执行阶段=>必须有多个执行部件或执行部件是流水化的
- 记分牌保存相关操作和状态
- 指令执行过程： IF, ISSUE, RO, EX, WR



记分牌控制的四阶段(1/2)

- **1. Issue—指令译码，检测结构相关**

- **准入条件**: 顺序发射，并且没有结构相关，没有WAW相关
 - 如果当前指令所使用的**功能部件空闲**，并且没有其他**活动的（已进入Issue）** 指令使用相同的目的寄存器 (**WAW**)，记分牌发射该指令到功能部件，并更新记分牌内部数据，如果有结构相关或WAW相关，则该指令的发射暂停，并且也不发射后继指令，直到相关解除。

- **2. Read operands—没有数据相关时，读操作数**

- **准入条件**: 没有**RAW**相关。
 - 如果先前已Issue的正在运行的指令不对当前指令的源操作数寄存器进行写操作，或者一个正在工作的功能部件已经完成了对该寄存器的写操作，则该操作数有效。操作数有效时，记分牌控制功能部件读操作数，准备执行。
 - 记分牌在这一步动态地解决了**RAW相关**，指令可能会乱序执行。



记分牌控制的四阶段 (2/2)

- **3.Execution—取到操作数后执行 (EX)**
 - **准入条件:** RO后准入
 - 接收到操作数后，功能部件开始执行
 - 当计算出结果后，通知记分牌，可以结束该条指令的执行。
- **4.Write result—finish execution (WR)**
 - **准入条件:** 没有WAR相关
 - 一旦记分牌得到功能部件执行完毕的信息后，记分牌**检测WAR相关**，如果没有WAR相关，就写结果，如果有WAR 相关，则暂停该条指令。
 - Example:
 - DIVD F0,F2,F4
 - ADDD F10,F0,F8
 - SUBD F8,F8,F14
 - CDC 6600 scoreboard 将暂停 SUBD 直到ADDD 读取操作数后，才进入WR段处理。



记分牌的结构

1. **Instruction status**—记录正在执行的各条指令的状态步
2. **Functional unit status**—记录功能部件(FU)的状态。用9个域记录每个功能部件的9个参量：

Busy—指示该部件是否空闲

Op—该部件所完成的操作

Fi—其目的寄存器编号

Fj, Fk—源寄存器编号

Qj, Qk—产生源操作数Fj, Fk的功能部件

Rj, Rk—标识源操作数Fj, Fk是否就绪的标志

3. **Register result status**—如果存在功能部件对某一寄存器进行写操作，在寄存器结果状态表中记录该功能部件。如果没有指令对该寄存器进行写操作，则该域为Blank

寄存器索引	F0	F2	F4	F6									
部件编号														



记分牌流水线控制

Instruction status	Wait until	Bookkeeping
Issue	Not busy (FU) and not Result(D)	$\text{Busy(FU)} \leftarrow \text{yes}; \text{Op(FU)} \leftarrow \text{op};$ $\text{Fi(FU)} \leftarrow 'D'; \text{Fj(FU)} \leftarrow 'S1';$ $\text{Fk(FU)} \leftarrow 'S2'; \text{Qj} \leftarrow \text{Result}('S1');$ $\text{Qk} \leftarrow \text{Result}('S2'); \text{Rj} \leftarrow \text{not Qj};$ $\text{Rk} \leftarrow \text{not Qk}; \text{Result}('D') \leftarrow \text{FU};$
Read operands	Rj and Rk	$\text{Rj} \leftarrow \text{No}; \text{Rk} \leftarrow \text{No}$
Execution complete	Functional unit done	
Write result	$f((\text{Fj}(f) \neq \text{Fi(FU)}) \text{ or } (\text{Rj}(f) = \text{No})) \text{ & }$ $(\text{Fk}(f) \neq \text{Fi(FU)}) \text{ or } (\text{Rk}(f) = \text{No}))$	$f(\text{if Qj(f)=FU then Rj(f) \leftarrow Yes});$ $f(\text{if Qk(f)=FU then Rk(f) \leftarrow Yes});$ $\text{Result}(\text{Fi(FU)}) \leftarrow 0; \text{Busy(FU)} \leftarrow \text{No}$

Result(): 寄存器结果状态表, FU: 功能部件(编号)

Scoreboard算法小结

- **硬件方法挖掘ILP**
 - 编译阶段无法确定的相关性，在程序执行时，用硬件方法判定
 - 可以使得程序代码在其他机器上有效地执行
- **记分牌的主要思想：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

- 乱序执行(out-of-order execution) => 乱序完成(out-of-order completion) CPI ↓
- 发射前检测结构相关和WAW相关
- 读操作数前检测RAW相关
- 写结果前处理WAR相关





指令流动态调度方法： Scoreboard

A large blue double-headed arrow graphic spans the width of the slide. On the left side, the text "记分牌技术要点" is displayed in white. On the right side, the word "示例" is displayed in red.

记分牌
技术要点

示例



Scoreboard Example

Instruction status

Instruction	j	k
LD	F6	34+
LD	F2	45+
MULTD	F0	F2
SUBD	F8	F6
DIVD	F10	F0
ADDD	F6	F8

Read Executi Write
Issue operana completer

Functional unit status

Time	Name	Busy	Op	dest	S1	S2	FU for j	FU for k	Fj?	Fk?
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30

基本假设:

- 1、5个FU
- 2、Load - 1个Cycle
Multi - 10个Cycle
Add - 2个Cycle
Divide - 40个Cycle



Scoreboard Example: Cycle 1

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	dest	S1	S2	FU	FU	Fj?	Fk?	
		Busy	Op	F i	F j	F k	Q j	Q k	R j
	Integer	Yes	Load	F6		R2			Yes
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU				Integer				

Scoreboard Example: Cycle 2

Instruction status:

Instruction	j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
2									

FU

Integer

- Issue 2nd LD?



Scoreboard Example: Cycle 3

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Read Open</i>	<i>Exec Comp</i>	<i>Write Result</i>
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
3	<i>FU</i>								

• Issue MULT?



Scoreboard Example: Cycle 4

Instruction status:

Instruction	j	k	Read			Exec	Write
			Issue	Op	Comp	Result	
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?	
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU								



Scoreboard Example: Cycle 5

Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write			
			<i>Issue</i>	<i>Op</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	<i>dest</i>		<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
		<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Integer	Yes	Load	F2		R3			Yes
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
5	<i>FU</i>	Integer							



Scoreboard Example: Cycle 6

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5	6	
MULTD	F0	F2	F4	6		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F2		R2			Yes	
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	Integer						

Scoreboard Example: Cycle 7

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write
				<i>Oper</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6		
SUBD	F8	F6	F2	7		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	Yes	Load	F2		R3			No	
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2	Integer	Yes	No	
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	Integer			Add			

- Read multiply operands?



Scoreboard Example: Cycle 8a (First half of clock cycle)

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6		
SUBD	F8	F6	F2	7		
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F2		R3			No	
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2	Integer	Yes	No	
	Divide	Yes	Div	F10	F0	F6	Mult1	No		Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	Integer		Add	Divide			

Issue DIVD



Scoreboard Example: Cycle 8b (Second half of clock cycle)

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6		
SUBD	F8	F6	F2	7		
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1			Add	Divide			

2nd LD 写结果，并通知相关指令结果可用，释放IU部件



Scoreboard Example: Cycle 9

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write
				<i>Oper</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	dest	<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
			<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
10	Integer	No							
10	Mult1	Yes	Mult	F0	F2	F4		Yes	Yes
	Mult2	No							
2	Add	Yes	Sub	F8	F6	F2		Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
9	<i>FU</i>	Mult1			Add	Divide			

- Read operands for MULT & SUB? Issue ADDD?



Scoreboard Example: Cycle 10

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	No							
9	Mult1	Yes	Mult	F0	F2	F4		No	No
	Mult2	No							
1	Add	Yes	Sub	F8	F6	F2		No	No
	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
10	FU	Mult1			Add	Divide			



Scoreboard Example: Cycle 11

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	No							
8	Mult1	Yes	Mult	F0	F2	F4		No	No
	Mult2	No							
0	Add	Yes	Sub	F8	F6	F2		No	No
	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1			Add	Divide			



Scoreboard Example: Cycle 12

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	No							
7	Mult1	Yes	Mult	F0	F2	F4		No	No
	Mult2	No							
	Add	No							
	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	Mult1					Divide		

- Read operands for DIVD?



Scoreboard Example: Cycle 13

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13		

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
6	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
13	FU	Mult1			Add		Divide		



Scoreboard Example: Cycle 14

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
5	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
2	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1		Add			Divide		



Scoreboard Example: Cycle 15

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?	
			Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
4	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
1	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1			Add		Divide		

Scoreboard Example: Cycle 16

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write
				<i>Oper</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	Busy	dest	<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
			<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Integer	No							
3	Mult1	Yes	Mult	F0	F2	F4		No	No
	Mult2	No							
0	Add	Yes	Add	F6	F8	F2		No	No
	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
16	<i>FU</i>	Mult1			Add		Divide		

Scoreboard Example: Cycle 17

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write
				<i>Op</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	16

WAR Hazard!

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	No								
2	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
17	FU	Mult1			Add				Divide

- Why not write result of ADD???

Scoreboard Example: Cycle 18

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write
				<i>Oper</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Integer	No							
1	Mult1	Yes	Mult	F0	F2	F4		No	No
	Mult2	No							
	Add	Yes	Add	F6	F8	F2		No	No
	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
18	FU	Mult1		Add			Divide		



Scoreboard Example: Cycle 19

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write
				<i>Op</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	19
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	dest		<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
		<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Integer	No							
0	Mult1	Yes	Mult	F0	F2	F4			No No
	Mult2	No							
	Add	Yes	Add	F6	F8	F2			No No
	Divide	Yes	Div	F10	F0	F6	Mult1		No Yes

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
19	<i>FU</i>	Mult1			Add		Divide		



Scoreboard Example: Cycle 20

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	19 20
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	Yes	Add	F6	F8	F2		No	No
	Divide	Yes	Div	F10	F0	F6		Yes	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
20	FU				Add				Divide

Scoreboard Example: Cycle 21

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write
				<i>Op</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	19 20
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8	21	
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	Busy	dest	<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
			<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	Yes	Add	F6	F8	F2		No	No
	Divide	Yes	Div	F10	F0	F6		Yes	Yes

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
21	<i>FU</i>				Add				Divide

- WAR Hazard is now gone...



Scoreboard Example: Cycle 22

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	19 20
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8	21	
ADDD	F6	F8	F2	13	14	16 22

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	No							
39	Divide	Yes	Div	F10	F0	F6		No	No

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
22	FU								Divide



Continue.....



Scoreboard Example: Cycle 61

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	19 20
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8	21	61
ADDD	F6	F8	F2	13	14	16 22

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	No							
0	Divide	Yes	Div	F10	F0	F6		No	No

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
61	FU								Divide



Scoreboard Example: Cycle 62

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6	9	19
SUBD	F8	F6	F2	7	9	11
DIVD	F10	F0	F6	8	21	61
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
62	FU								

Review: Scoreboard Example: Cycle 62

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write
				<i>Oper</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6	9	19
SUBD	F8	F6	F2	7	9	11
DIVD	F10	F0	F6	8	21	61
ADDD	F6	F8	F2	13	14	16
						22

Functional unit status:

Time	Name	dest		<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
		<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
62	<i>FU</i>								

- 顺序issue; 乱序 execute & 乱序 commit
- 问题: Branch指令怎么办?



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**
 - 寄存器文件至少需要有 $2x$ 个读端口和 x 个写端口.
 - 当有寄存器重命名时，例如：Tomasulo 还需要**
 - 多端口的“rename table”，以同时对一组指令所用的寄存器重命名
 - 在单周期内发射到多个RS中

Summary

- **硬件方法挖掘ILP**
 - 编译阶段无法确定的相关性，在程序执行时，用硬件方法判定
 - 可以使得程序代码在其他机器上有效地执行
- **记分牌的主要思想：允许stall后的指令继续**
 - 乱序执行(out-of-order execution) => 乱序完成(out-of-order completion)
 - 发射前检测结构相关和WAW相关
 - 读操作数前检测RAW相关
 - 写结果前处理WAR相关





CDC 6600 Scoreboard

CDC 6600 scoreboard的主要缺陷：

- 没有定向数据通路
- 指令窗口较小，仅局限于基本块内的调度
 - 基本块按序执行
- 功能部件数较少
- 结构冲突时不能发射
- WAR相关是通过等待解决的
- WAW相关时，不会进入Issue阶段



CDC 6600 Scoreboard

CDC 6600 scoreboard的主要缺陷：

- 功能部件数较少，指令窗口较小
- 没有定向数据通路
- 仅局限于基本块内的动态指令流调度
 - Branch类指令执行完成后，才能Issue下一条指令
- 结构冲突时不能发射
- WAR相关是通过等待解决的
- WAW相关时，不会进入Issue阶段



动态调度方案之二：Tomasulo Algorithm

- 该算法首次在 IBM 360/91 (1968/01) 上使用
(CDC6600, 1964, Scoreboard)
- 目标：在没有专用编译器的情况下，提高系统性能
- IBM 360 & CDC 6600 ISA的差别
 - IBM360只有 2位寄存器描述符 vs. CDC 6600寄存器描述符3位
 - IBM360 4个FP 寄存器 vs. CDC 6600 8个
 - IBM 360 有memory-register 操作
- Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604, ...



Tomasulo Algorithm vs. Scoreboard

- **控制和缓存:** 分布在各部件中 vs. 集中在记分牌
 - FU 缓存称“Reservation Stations”; 保存待用操作数
- **寄存器重命名:** Tomasulo 有 vs. Scoreboard 无
 - 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
- **定向路径:** Tomasulo 有 vs. Scoreboard 无
 - 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
- **控制相关处理:** Tomasulo 分支可跨越 vs. Scoreboard 不可跨越
 - 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
- **Load和Store部件也看作带有RS的功能部件**



Tomasulo Organization

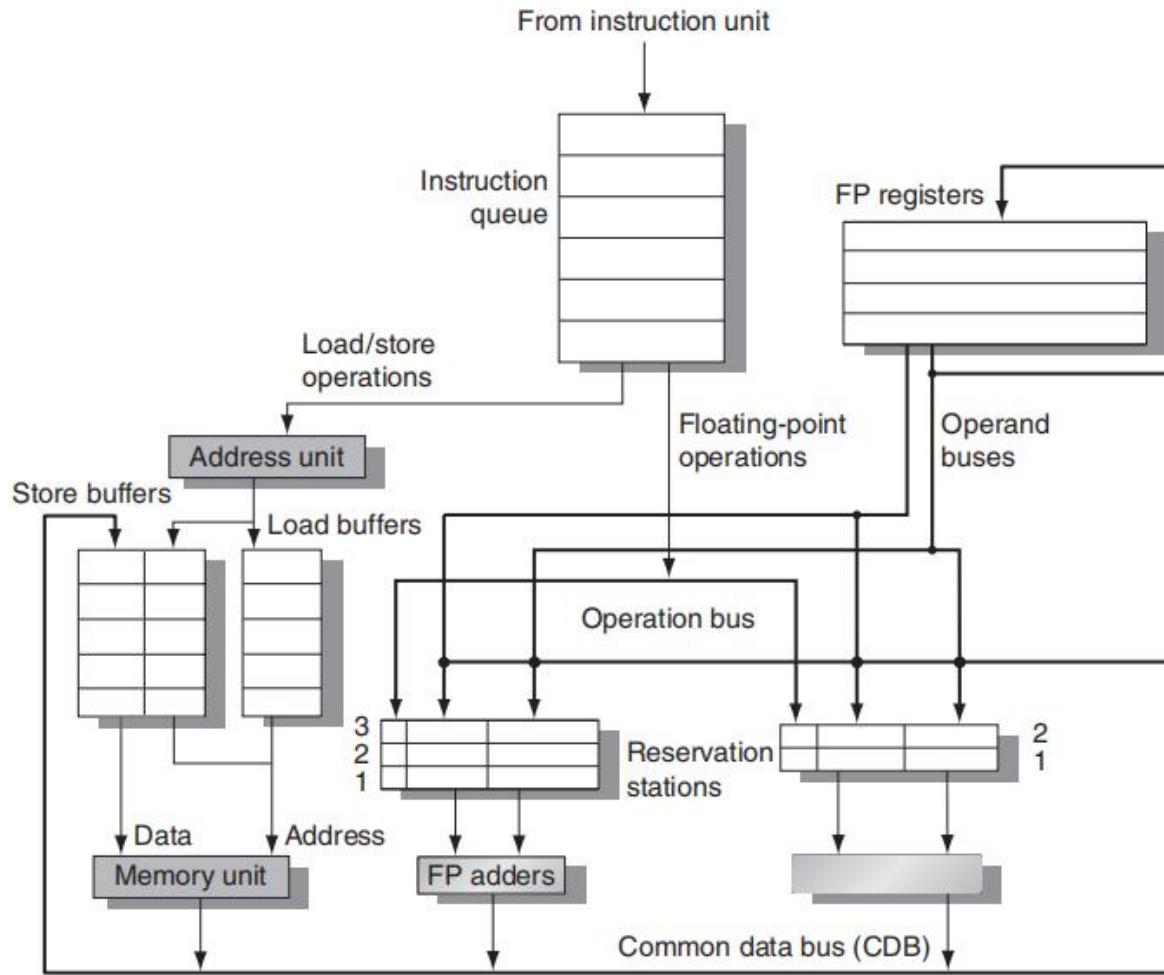


Figure 3.6 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm.



Reservation Station 结构

Op: 部件所进行的操作

V_j, V_k: 源操作数的值。Store 缓冲区有V_k域，用于存放要写入存储器的值

A: 存放存储器地址。开始存立即数，计算出有效地址后，存放有效地址

Qj, Qk : 产生源操作数的RS

注：没有记分牌中的准备就绪标志， $Qj, Qk=0 \Rightarrow$ ready

Store 缓存区中Qk表示产生结果的RS

Busy: 标识RS或FU是否空闲

Register result status—如果存在对寄存器的写操作，指示对该寄存器进行写操作的部件。

Qi: 保留站的编号



Tomasulo 算法的三阶段

- 1. Issue—从FP操作队列中取指令
 - 如果RS空闲(no structural hazard), 则控制发射指令和操作数(renames registers). **消除WAR, WAW相关**
- 2. Execution—operate on operands (EX)
 - 当两操作数就绪后, 就可以执行
如果没有准备好, 则监测Common Data Bus 以获取结果。通过推迟指令执行**避免RAW相关**
- 3. Write result—finish execution (WB)
 - 将结果通过Common Data Bus传给所有等待该结果的部件;
标识RS可用
- **数据通信: 功能部件产生结果的传送**
 - 通常的数据总线: data + destination (“go to” bus)
 - Common data bus: data + source (“come from” bus)
 - 64 bits 数据线 + 4 bits 功能部件源地址 (FU source address)
 - 产生结果的部件如果与RS中等待的部件匹配, 就接收数据
 - 广播方式传送



Tomasulo 算法流水线控制

1、Issue

FP Operation:

Wait until : Station r empty

Action or bookkeeping:

```

if(RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi}
else {RS[r].Vj ← Reg[rs]; RS[r].Qj ← 0 }

if(RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi}
else {RS[r].Vk ← Reg[rt]; RS[r].Qk ← 0 }

RS[r].Busy ← yes; RegisterStat[rd].Qi = r;

```

Load or Store:

Wait until: Buffer r empty

Action or bookkeeping:

```

if(RegisterStat[rs].Qi≠0)
    {RS[r].Qj←RegisterStat[rs].Qi}
else {RS[r].Vj←Reg[rs]; RS[r].Qj ← 0 }

RS[r].A ← imm; RS[r].Busy ← yes;

```

Load only: RegisterStat[rt].Qi = r;

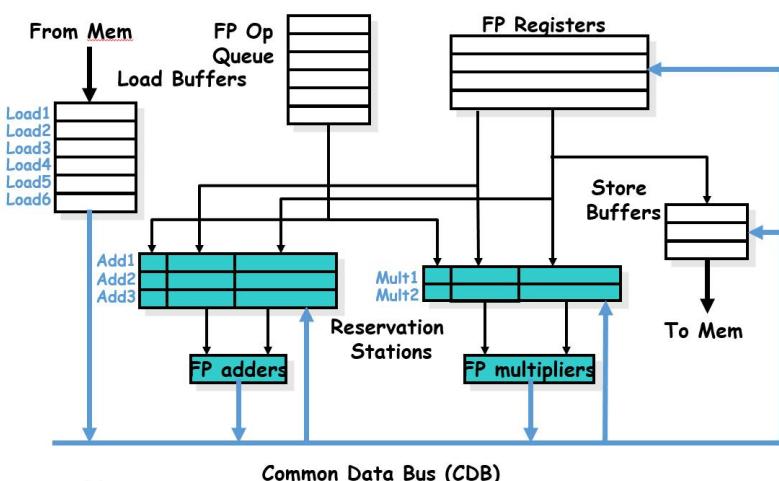
Store only:

```

if(RegisterStat[rt].Qi≠0) {RS[r].Qk←RegisterStat[rt].Qi}
else {RS[r].Vk←Reg[rt]; RS[r].Qk ← 0 }

```

rs, rt : 源寄存器名; **rd**: 目的寄存器名
RS: 保留站数据结构; **r**: 保留站编号
RegisterStat: 寄存器结果状态表
Qi: 记录写结果的部件编号 (保留站编号)
Reg: 寄存器组





注意：Load操作在EXE阶段分两步

2、Execute

FP Operation

wait until: $(RS[r].Qj=0)$ and $(RS[r].Qk=0)$

Action or bookkeeping:

computer result: Operands are in V_j and V_k

Load-store step1

wait until: $RS[r].Qj = 0 \& r$ is head of load-store queue

//按照程序序计算访存地址

Action or bookkeeping:

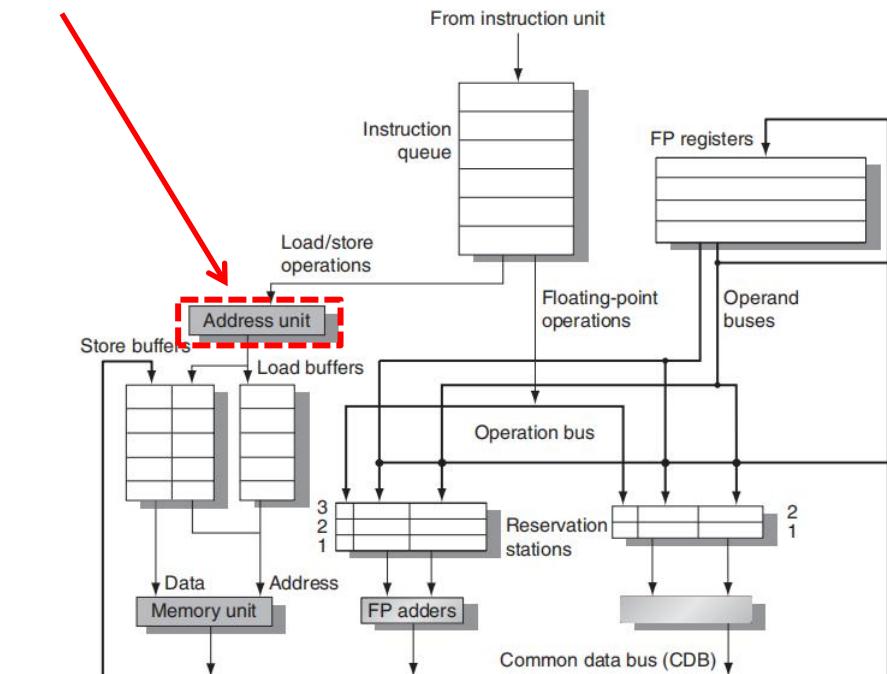
$RS[r].A \leftarrow RS[r].V_j + RS[r].A;$

Load step2

wait until: Load Step1 complete

Action or bookkeeping:

Read from Mem[$RS[r].A$]





3、Write result

FP Operation or Load

Wait until: Execution complete at r & CDB available

Action or bookkeeping

$\forall x \text{ (if } (\text{RegisterStat}[x].Qi=r) \{ \text{Regs}[x] \leftarrow \text{result}; \text{RegisterStat}[x].Qi \leftarrow 0 \})$

$\forall x \text{ (if } (\text{RS}[x].Qj=r) \{ \text{RS}[x].Vj \leftarrow \text{result}; \text{RS}[x].Qj \leftarrow 0 \});$

$\forall x \text{ (if } (\text{RS}[x].Qk=r) \{ \text{RS}[x].Vk \leftarrow \text{result}; \text{RS}[x].Qk \leftarrow 0 \});$

$\text{RS}[r].\text{Busy} \leftarrow \text{no};$

Store

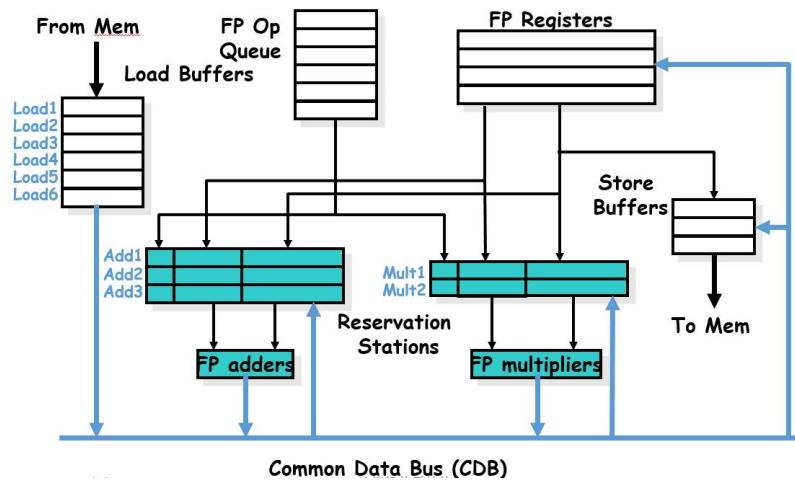
wait until:

Execution complete at r & $\text{RS}[r].Qk = 0$

Action or bookkeeping

$\text{Mem}[\text{RS}[r].A] \leftarrow \text{RS}[r].Vk;$

$\text{RS}[r].\text{Busy} \leftarrow \text{no};$





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
- **传给FU的值从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操作队列中操作不仅仅局限于基本块**



No.	Inst.	i	j	k	Issue	Exec-start	Exec-End	Cache	WR(CDB)
1	LD	F6	34+	R2	1	2		3	4
2	LD	F2	45+	R3	2	3		4	5
3	MULTD	F0	F2	F4	3	6	15		16
4	SUBD	F8	F6	F2	4	6	7		8
5	DIVD	F10	F0	F6	5	17	56		57
6	ADDD	F6	F8	F12	6	9	10		11

- **TIPS：访存顺序约定不同，结果会不同**
- **本例中的访存约定**
 - 分别有LoadBuffer和StoreBuffer，计算地址和实际访存buffer合并；
 - 顺序Load访存、顺序Store访存，Load访存可以跨越Store访存先行
- **其他访存约定：**
 - 所有访存指令1个计算地址队列（Memory队列）；实际访存操作时分为load队列和store队列；顺序计算访存地址

Compare to Scoreboard Cycle 62

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	Read	Exec	Write	<i>Issue</i>	<i>Comp</i>	<i>Result</i>
				<i>Oper</i>	<i>Comp</i>	<i>Result</i>			
LD	F6	34+	R2	1	2	3	4	1	3
LD	F2	45+	R3	5	6	7	8	2	4
MULTD	F0	F2	F4	6	9	19	20	3	15
SUBD	F8	F6	F2	7	9	11	12	4	7
DIVD	F10	F0	F6	8	21	61	62	5	56
ADDD	F6	F8	F2	13	14	16	22	6	10

- 为什么scoreboard/6600所需时间较长?
 - 结构冲突
 - WAR,WAW冲突
 - 没有定向技术



Tomasulo v. Scoreboard (IBM 360/91 v. CDC 6600)

IBM 360/91 (Tomasulo)	CDC 6600 (Scoreboard)
流水化的功能部件 (6 load, 3 store, 3 +, 2 x/÷)	多个功能部件 (1 load/store, 1 +, 2 x, 1 ÷,...)
指令窗口较大	指令窗口较小
有结构冲突时不发射	有结构冲突时不发射
WAR: 用寄存器重命名避免	WAR: 用stall来避免
WAW: 用寄存器重命名避免	停止发射
从FU广播结果	写寄存器方式
分散Control: RS	集中式控制scoreboard



Tomasulo 算法的特点

- **控制和缓存分布在各部件中**
 - FU 缓存称“reservation stations”; 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
- **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 Loop Example

Loop:	LD F0, 0(R1)
	MULTD F4, F0, F2
	SD F4, 0(R1)
	SUBI R1,R1,#8
	BNEZ R1 Loop

- 假设循环3次，设Multiply执行阶段4 cycles
- 访存操作分为计算地址和访存两阶段
 - 计算地址需1个cycle
 - 第1次load时Cache未命中，访存需7个cycles (cache miss), 第2次以后均命中，访存操作需1cycles
- **访存顺序的约定：**
 - 所有访存指令1个计算地址队列，实际访存操作时分为load队列和store队列
 - 计算访存地址按序，Load操作之间按序，Store操作按序
 - Load访存操作如果与store访存操作没有冲突，可以先行
- **为清楚起见，下面我们也列出SUBI, BNEZ的时钟周期**



Summary-Loop Example

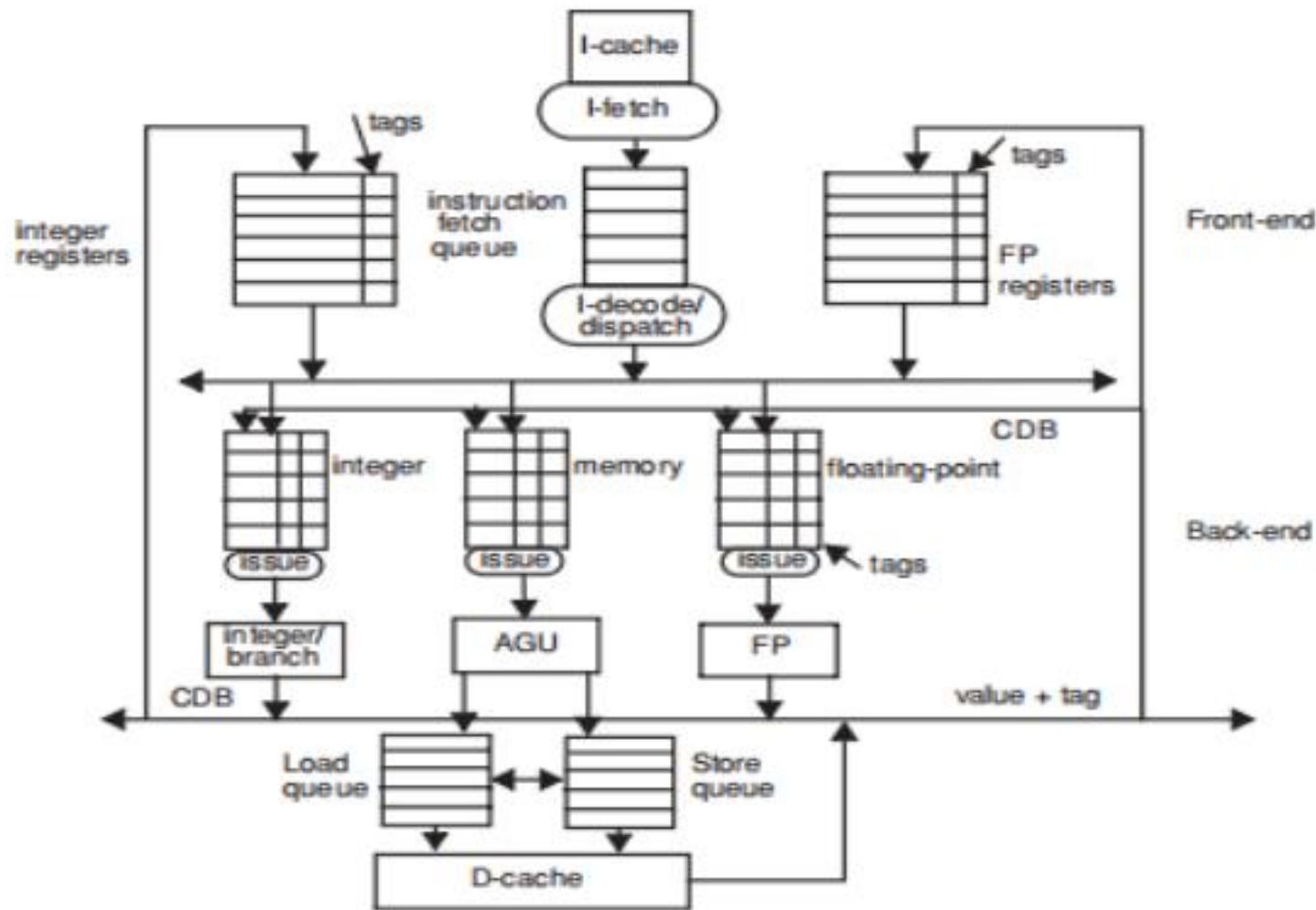
ITER	Inst.	i	j	k	Issue	Exec-start	Exec-End	Cache	WR(CDB)
1	LD	F0	0	R1	1	2		3~9	10
1	MULTD	F4	F0	F2	2	11	14		15
1	SD	F4	0	R1	3	4	-	16	
1	SUB				4				
1	BNEZ				5				
2	LD	F0	0	R1	6	11		11	12
2	MULTD	F4	F0	F2	7	13	16		17
2	SD	F4	0	R1	8	17		18	
2	SUB				9				
2	BNEZ				10				
3	LD	F0	0	R1	11	13 (18)		13 (18)	14 (19)
3	MULTD	F4	F0	F2	16	17 (20)	20 (23)		21 (24)
3	SD	F4	0	R1	17	19		22 (25)	
3								

- 本例访存约定：
- Load buffer
 - Store buffer
 - 顺序Load访存
 - 顺序Store访存
 - **Load访存可以跨越Store访存先行** (前提：Load的地址与Store地址不冲突时)

- **TIPS：不同的存储器访问序的约定会产生不同结果。**
- **其他约定？**
 - 例如：简单约定所有访存指令按序执行（计算地址队列和实际访存buffer合并，并且顺序访存）；



Tomasulo Organization



说明：1、Tomasulo 算法中的ISSUE 对应图中的dispatch，该图中的issue指数据准备好了可送到执行部件执行。2、memory访问分为两部（1）AGU 计算地址（2）访存



Summary-Loop Example

ITER	Inst.	i	j	k	Issue	Exec-start	Exec-End	D-Cache	WR(CDB)
1	LD	F0	0	R1	1	2	2	3~9	10
1	MULTD	F4	F0	F2	2	11	14		15
1	SD	F4	0	R1	3	4	4	16	
1	SUB				4				
1	BNEZ				5				
2	LD	F0	0	R1	6	7	7	11	12
2	MULTD	F4	F0	F2	7	13	16		17
2	SD	F4	0	R1	8	9	9	18	
2	SUB	R1	R1		9	10	11		12
2	BNEZ				10	13			
3	LD	F0	0	R1	11	12	12	13	14
3	MULTD	F4	F0	F2	16	17	20		21
3	SD	F4	0	R1	17	18	18	22	
3								

- 本例访存约定：
- Memory队列
 - Load buffer
 - Store buffer
- 顺序计算访存地址
- 顺序Load访存
- 顺序Store访存
- Load访存可以跨越 Store访存先行 (前提：Load的地址与 Store地址不冲突时)

- TIPS: 不同的存储器访问序的约定会产生不同结果。
- 其他约定?
 - 例如：简单约定所有访存指令按序执行 (计算地址队列和实际访存buffer合并，并且顺序访存)；



小结

- **Tomasulo Algorithm 三阶段**
 - **1. Issue—从FP操作队列中取指令**
 - 如果RS空闲(no structural hazard), 则控制发射指令和操作数 (renames registers).
 - **2. Execution—operate on operands (EX)**
 - 当两操作数就绪后, 就可以执行
如果没有准备好, 则监测Common Data Bus 以获取结果
 - **3. Write result—finish execution (WB)**
 - 将结果通过Common Data Bus传给所有等待该结果的部件;
表示RS可用
- **基本数据结构**
 - **1. Instruction Status**
 - **2. Reservation Station**
 - **3. Register Result Status**
- **注意:**
 - CDB冲突、Loadbuffer和Storebuffer操作冲突



小结

- **Reservations stations: 寄存器重命名，缓冲源操作数**
 - 避免寄存器成为瓶颈
 - 避免了Scoreboard中无法解决的 WAR, WAW hazards
 - 允许硬件做循环展开
- **不限于基本块(分支指令后的指令可以继续发射)**
- **贡献**
 - Dynamic scheduling
 - Register renaming
 - Load/store disambiguation
- **360/91 后 Pentium II; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264使用这种技术**



Summary: Tomasulo算法实现循环重叠执行?

- **寄存器重命名技术**
 - 不同的循环使用不同的物理寄存器 (dynamic loop unrolling).
 - 将代码中的静态寄存器名修改为动态寄存器指针 “pointers”
 - 有效地增加了寄存器文件的大小
- **关键: 分支指令后的指令可以继续发射, 以便能发射多个循环中的操作**



Part1：指令级并行

- **指令级并行：提高单条指令流（单线程）程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - 流水线的扩展：指令执行周期数(可能) 不同
 - 指令级并行优化：编译优化和硬件优化
 - 动态调度的“漏网之鱼”——控制相关与异常处理问题
 - 分支预测技术
 - 推断执行技术
- **多发射流水线：ILP的突破 ($CPI < 1$)**
 - 每个cycle发射多条指令执行
- **多线程技术：单线程ILP瓶颈的突破**
 - 流水线执行的指令来源于多条指令流
- **其他：“发射”指每个cycle可进入功能部件（执行部件）的指令条数**



分支预测方法

控制相关对
性能的影响

基于BHT的
分支预测

基于BTB的
分支预测

- 1、基本2-bit预测器 (c-27)
- 2、关联预测器 (两级预测器) (3.3)
- 3、组合预测器 (3.3)

- 1、分支目标缓冲区(3.9)
- 2、Return Address预测器(3.9)



?

- 动态指令流调度硬件方案可以用硬件进行循环展开
- 如何处理精确中断?
 - Out-of-order execution -> out-of-order completion!
- 如何处理分支?
 - **用硬件做循环展开必须快速解决控制相关问题**



关于异常处理???

- **乱序完成加大了实现精确异常的难度**
 - 在前面指令还没有完成时，寄存器文件中可能有后面指令的运行结果.
 - 如果这些前面的指令执行时有异常产生，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：**
 - 精确异常的含义：
 - 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
- **实现精确异常的技术：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



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

- 如果分支与其他指令有依赖关系,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



控制相关的动态解决技术

- **控制相关：**

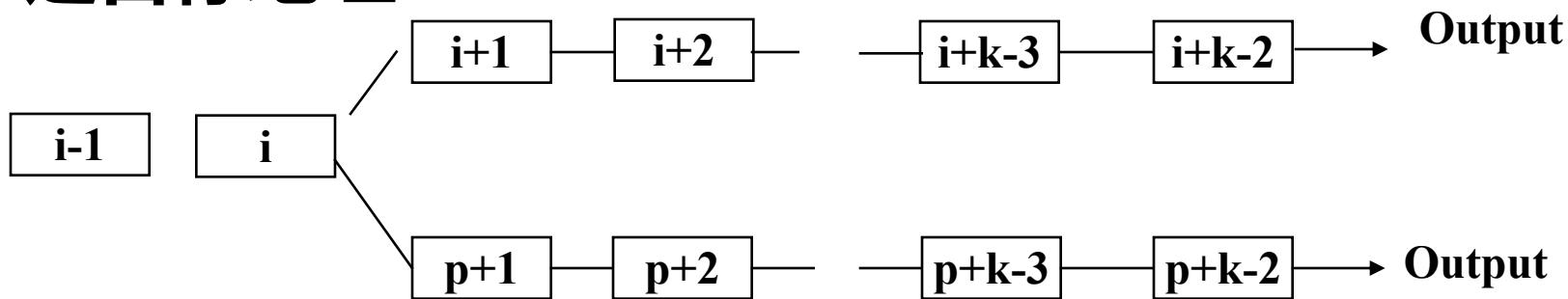
- 由条件转移或程序中断引起的相关，也称全局相关。
- 控制相关对流水线的吞吐率和效率影响相对于数据相关要大得多
 - 条件指令在一般程序中所占的比例相当大
 - 中断虽然在程序中所占的比例不大，但中断发生在程序中的哪条指令，发生在一条指令执行过程中的哪个功能段都是不确定的

- **处理条件转移和异常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，每次分支“跳转”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 , 采用静态分支预测“不跳转”策略，条件“跳转”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？



分支预测方法

控制相关对
性能的影响

基于BHT的
分支预测

基于BTB的
分支预测

- 1、基本2-bit预测器
- 2、关联预测器（两级预测器）
- 3、组合预测器

- 1、分支目标缓冲区
- 2、Return Address预测器



分支预测概览

- **分支预测的核心目标：**解决分支指令的“流水线陷阱”
- **分支预测的核心任务：**方向预测和目标地址预测
- **预测器的分类：**基于BHT表的预测器 和 优化取指令的带宽

- **基于BHT表的预测器：方向预测**
 - 基本的2-bit预测器（饱和预测器）
 - 关联预测器（Correlating predictor） or 2级预测器：
 - GAp (Global History table and per-address predictor table)
 - 每条分支有多个2-bit预测器
 - **由最近的n次分支**的结果来选择对应的2-bits预测器
 - PAp (Per-address history table and per-address predictor table)
 - 每条分支有多个2-bit预测器
 - **由该分支最近的n次分支**结果来选择对应的2-bits预测器
 - Tournament predictor (竞赛预测器) : 组合GAp和PAp
- **优化取指令的带宽：目标地址预测**
 - Branch Target Buffer
 - Return Address Predictor



Instruction Fetch Unit

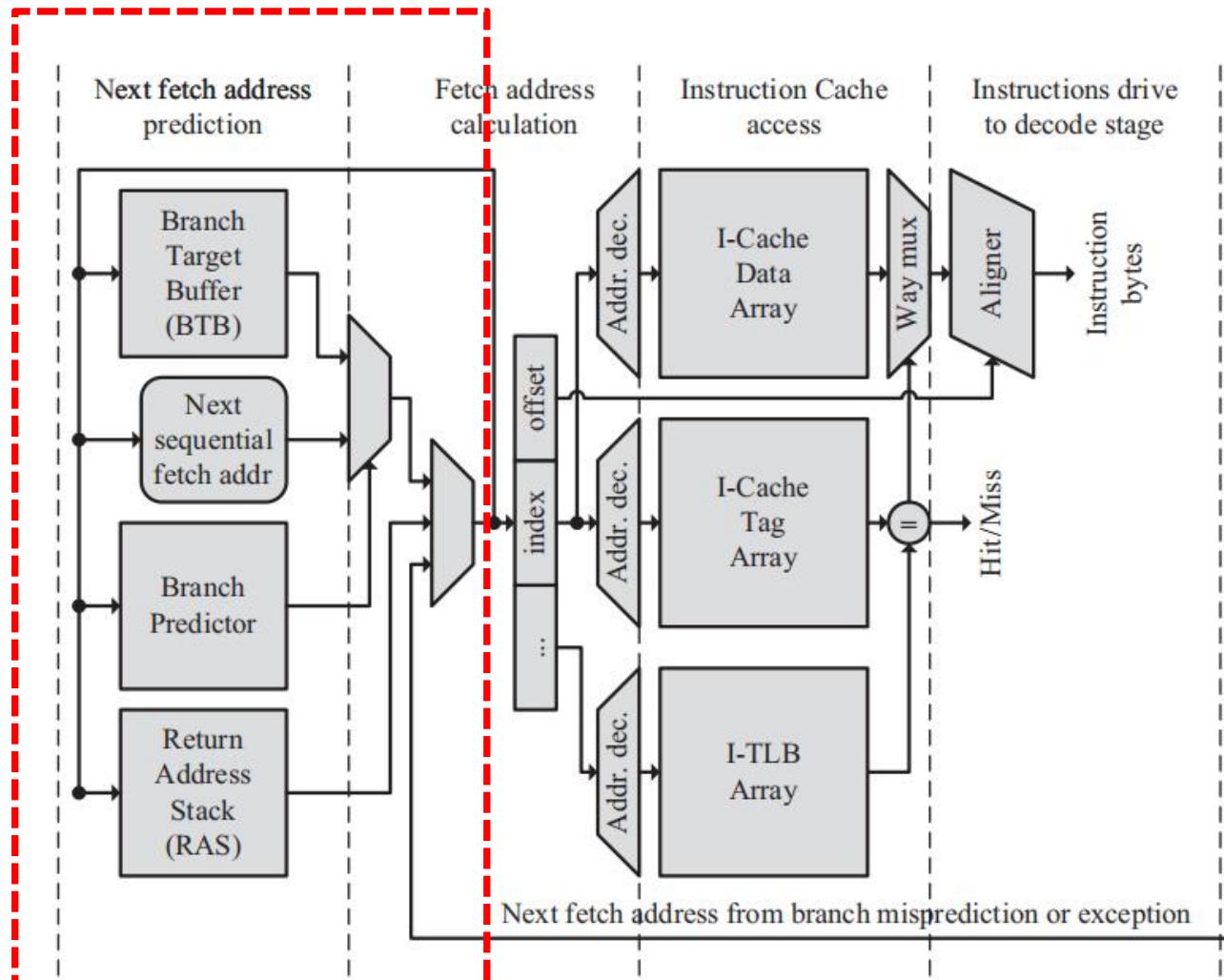
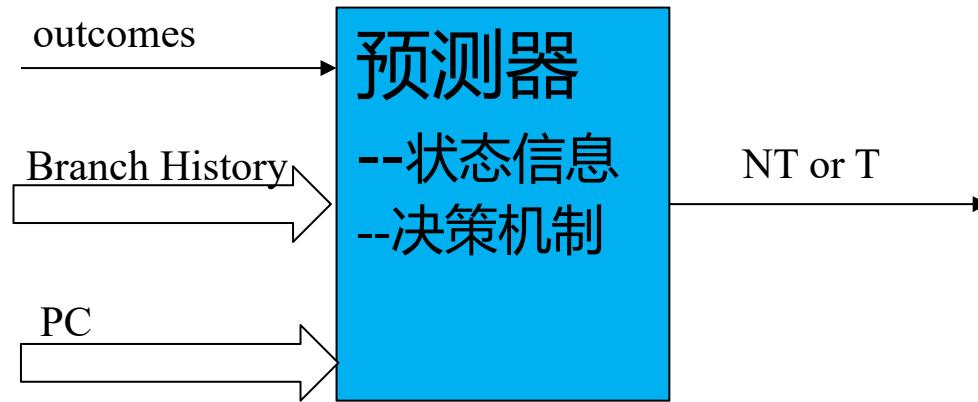


FIGURE 3.1: Example fetch pipeline.

分支预测在哪个阶段完成?

预测器的基本结构及输入输出



- **根据转移历史(和PC)来选择预测器(状态)**
- **由预测器的状态决定预测值(输出)**
- **根据实际结果(outcomes)更新预测器的状态信息**



从简单到复杂的预测器演进

• 静态预测：无历史依赖的简单规则

- 编译器或硬件按固定规则预测，无需历史信息，适用于早期简单处理器：
- 总是不跳转（Not Taken）：默认分支不执行（如if条件通常为假）；
- 总是跳转（Taken）：默认循环分支执行（如for循环多数情况继续）；
- 向后跳转预测为Taken，向前跳转预测为Not Taken：利用指令地址规律（循环分支通常向后跳转，if-else的异常分支通常向前跳转）。

• 静态预测缺陷：错误率高，仅适用于早期流水线 (如5段RISC)

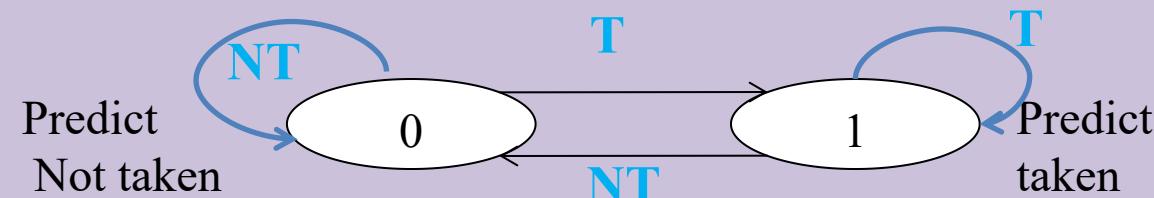
• 动态预测：



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（跳转或不跳转）
- **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 (GAp or PAp)
 - Tournament Predictors: Adaptively Combining Local and Global Predictors
 - 现代高级预测器：TAGE预测器
 - High Performance Instruction Delivery (优化取指令带宽)
 - BTB
 - Return Address Predictors
 - Integrated Instruction Fetch Units (单独的取指部件连接到流水线的其他部分，其中集成了分支预测器、指令预取、指令Cache的存取和缓存等)
- **Performance = $f(\text{accuracy}, \text{cost of misprediction})$**
 - Misprediction Flush Reorder Buffer

基本预测器：1-bit BHT



- 术语：

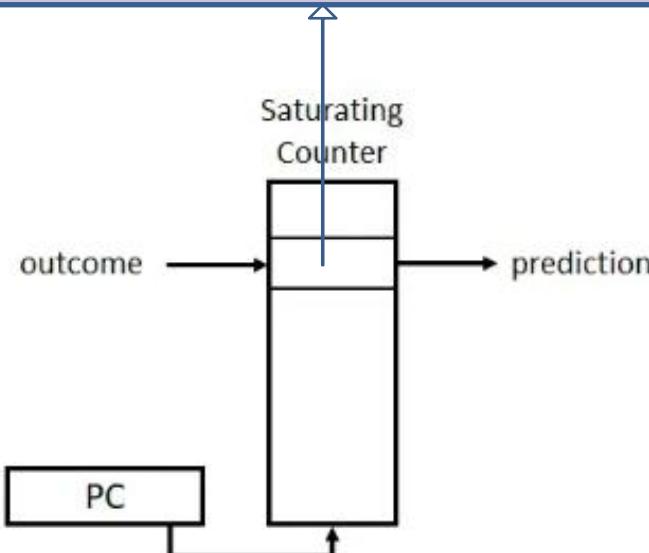
- Not taken | taken 跳转|不跳转 (成功|失败)
- 预测准确率 (Accuracy), 预测错误率(Misprediction)

- Branch History Table:

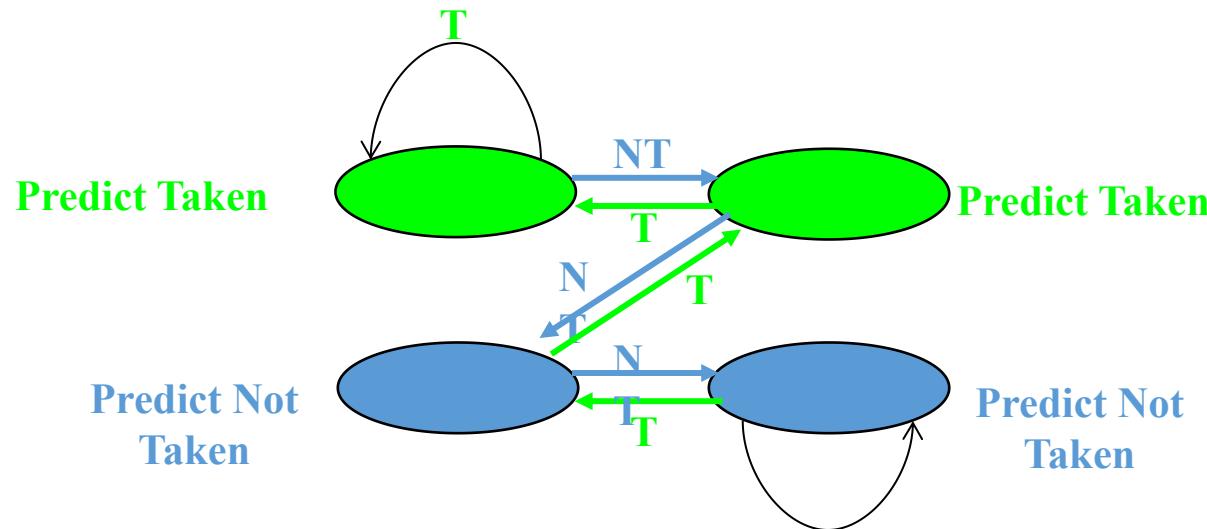
- 分支指令的PC的低位索引
- 该表记录上一次转移是否成功
- 不做地址检查
- 1-bit BHT

- 问题: 在一个循环中, 1-bit BHT 将导致2次分支预测错误

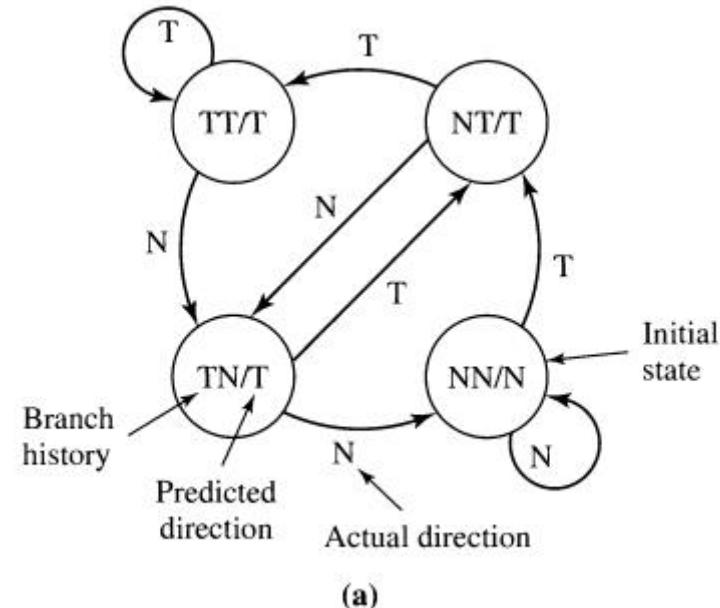
- 假设一循环次数为10次的简单程序段
- 最后一次循环：前面预测“跳转”，最后一次需要退出循环
- 首次循环：前面预测为“不跳转”，这次实际上为成功



基本预测器：2-bit BHT



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



预测器 (2-bits) 预测错误率

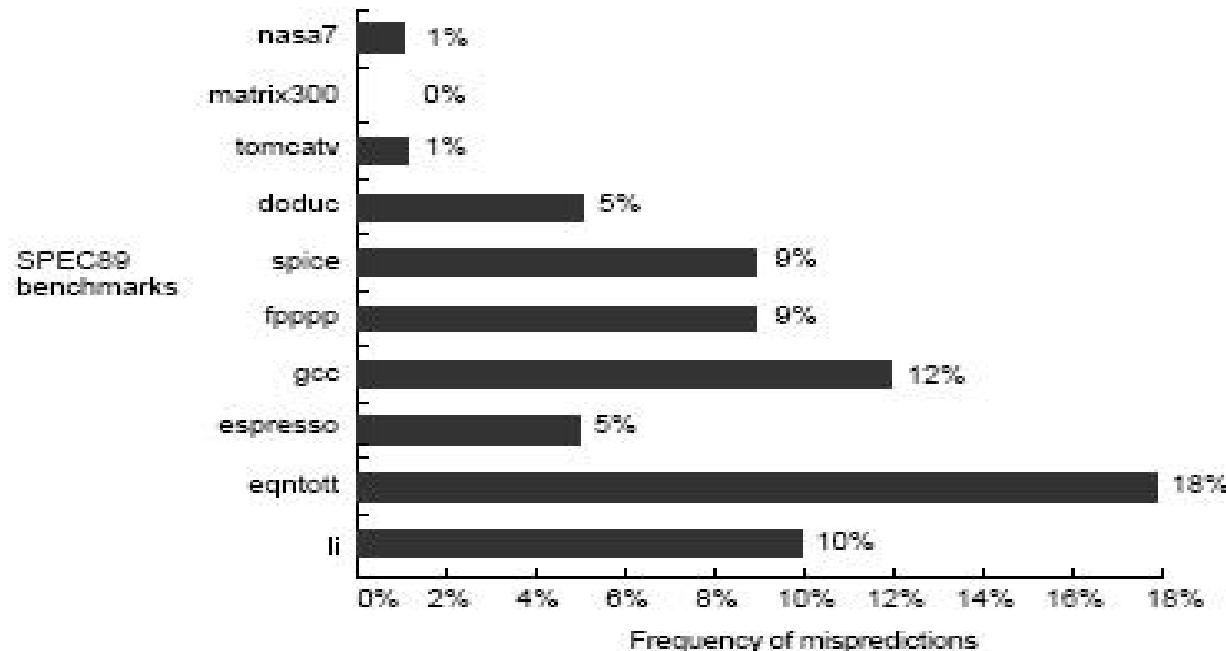


FIGURE 3.8 Prediction accuracy of a 4096-entry two-bit prediction buffer for the **SPEC89 benchmarks**. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the FP programs (average of 4%). Even omitting the FP kernels (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].

预测器 (2-bits) 预测错误率

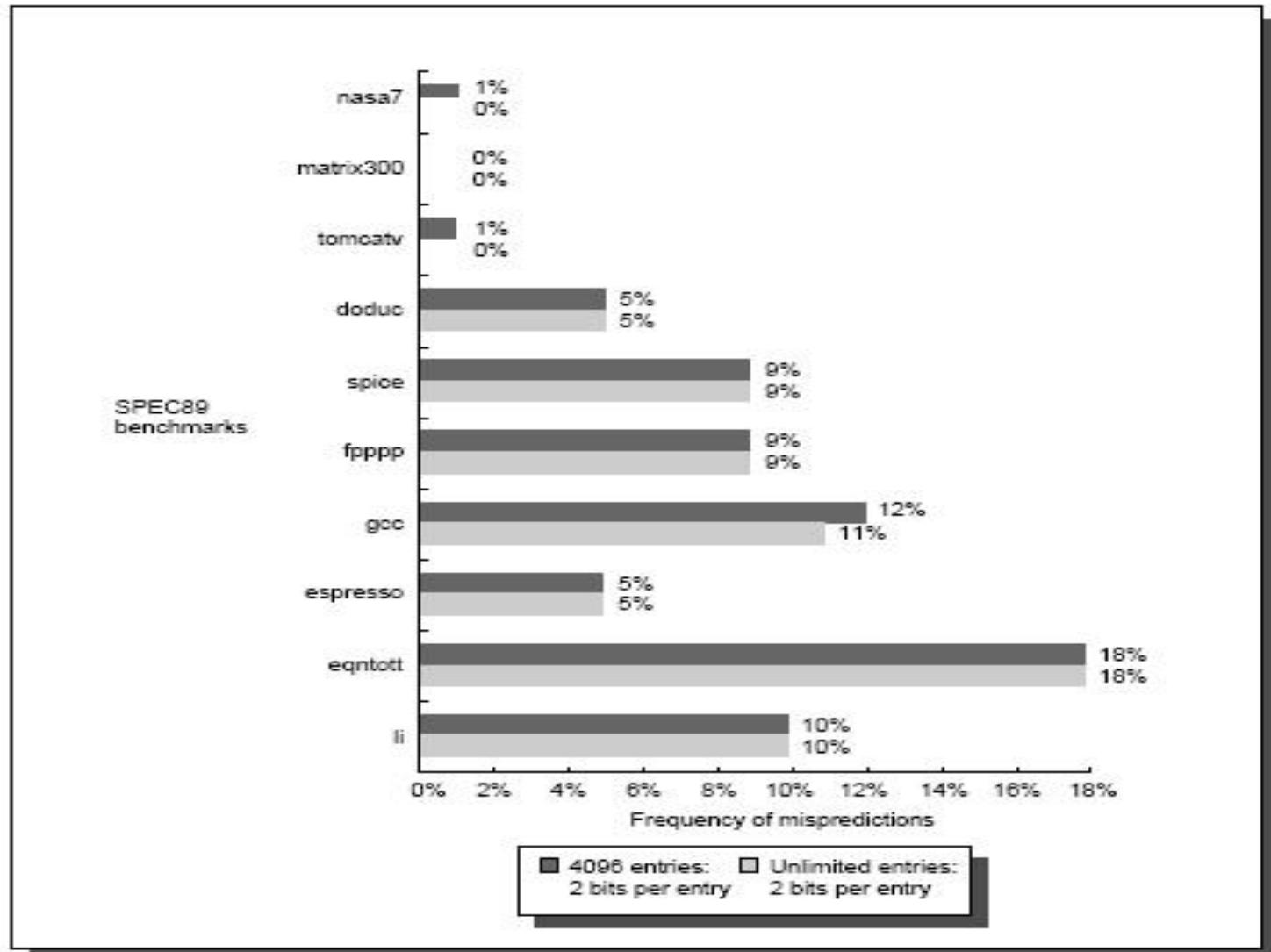


FIGURE 3.9 Prediction accuracy of a 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks.



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 Predictor

例如：

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

翻译为汇编指令

```
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 “不跳转” , 则 b3一定成功。即：
(aa ==2; bb == 2, 则 aa == bb)



分支间存在关联

- Correlating predictors 或 两级预测器：
 - 设计两级预测器，考虑“其他”分支行为
 - “其他”包括：全局或局部
- 工作原理：
 - 根据一个简单的例子来看其基本原理

存在关联的分支

```
if (d==0) d=1;  
if (d==1) d=0;
```

```
BNEZ R1,L1      ;branch b1(d!=0)  
ADDI R1,R0,#1    ;d==0, so d=1  
L1: ADDI R3,R1,#-1  
      BNEZ R3,L2  ;branch  
b2(d!=1)
```

...

L2:

分支间存在关联的情况举例(1/2)

- 假设d的初始值序列为0, 1, 2
- b1 如果分支 “不跳转” , b2一定也分支 “不跳转” 。
- 前面基本的1-bit 2-bit预测器都没法利用这一点



两级预测器

```
if (d==0)d=1;
```

```
if (d==1) d=0;
```

翻译为汇编指令

```
BNEZ R1,L1 ;branch b1(d!=0)
```

```
ADDI R1,R0,#1 ;d==0, so d=1
```

```
L1: ADDI R3,R1,#-1
```

```
BNEZ R3,L2 ;branch b2(d!=1)
```

Initial value
of d

d==0?

b1

Value of d
before b2

d==1?

b2

0	yes	not taken	1	yes	not taken
---	-----	-----------	---	-----	-----------

1	no	taken	1	yes	not taken
---	----	-------	---	-----	-----------

2	no	taken	2	no	taken
---	----	-------	---	----	-------

FIGURE 3.10 Possible execution sequences for a code fragment.

分支间存在关联的情况举例 (2/2)

- 假设d的初始值在2和0之间切换。T: “跳转”， NT: “不跳转”
- 用1-bit预测器， b1和b2的初始设置为预测NT

BNEZ R1,L1 ;branch b1(d!=0)
 ADDI R1,R0,#1 ;d==0, so d=1

L1: ADDI R3,R1,#-1
 BNEZ R3,L2 ;branch b2(d!=1)

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

FIGURE 3.11 Behavior of a one-bit predictor initialized to not taken. T stands for taken, NT for not taken.

- 结论：这样的序列每次预测都错，预测错误率100%。
- 问题出在哪里？有办法改善吗？



Correlating Branches

- **基本思想：记为 (1, 1)**
 - 用1位作为correlation位。记录最近一次执行的分支
 - 每个分支都有两个相互独立的预测位：一个预测位假设最近一次执行的分支“不跳转”时的预测位，另一个预测位是假设最近一次执行的分支“跳转”时的预测位。
- **最近一次执行的分支与要预测的分支可能不是同一条指令**

Prediction bits	Prediction if last branch	
	not taken	Prediction if last branch taken
NT/NT	not taken	not taken
NT/T	not taken	taken
T/NT	taken	not taken
T/T	taken	taken

FIGURE 3.12 Combinations and meaning of the taken/not taken prediction bits. T stands for taken, NT for not taken.



Correlating Branches

- Correlating 预测器的预测和执行情况
- 显然只有在第一次 $d=2$ 时，预测错误，其他都预测正确
- 记为 (1, 1) 预测器，即根据最近一次分支行为来选择一对1-bit预测器中的一个。
- 更一般的表示为 (m, n) ，即根据最近的 m 个分支，从 2^m 个分支预测器中选择预测器，每个预测器的位数为 n

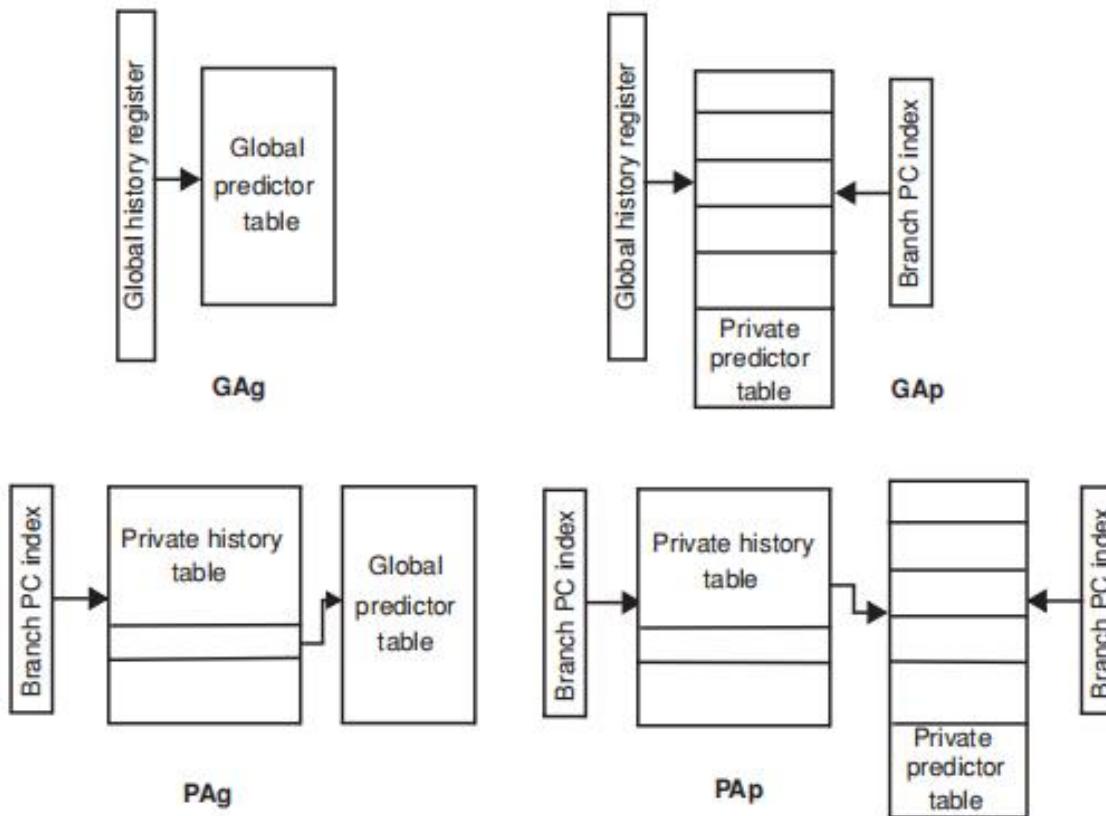
BNEZ R1,L1 ;branch b1($d!=0$)
ADDI R1,R0,#1 ; $d==0$, so $d=1$
L1: ADDI R3,R1,#-1
BNEZ R3,L2 ;branch b2($d!=1$)

$d=?$	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

FIGURE 3.13 The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken. T stands for taken, NT for not taken. The prediction used is shown in bold.

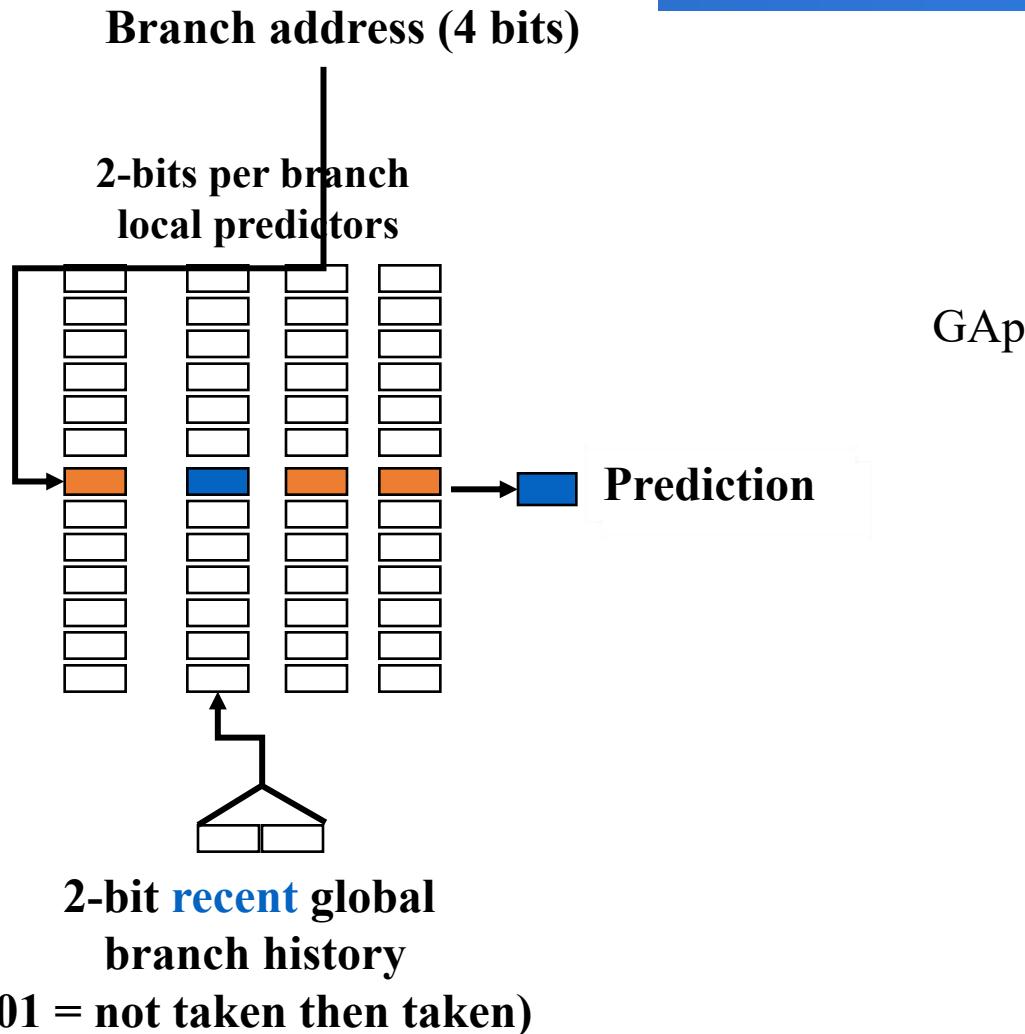


两级预测器（枚举）



- **关联预测器也称为两级预测器**
- **两级预测器的四种组合：**依据两级的全局或局部属性
 - ①GAg：全局历史表和全局预测表；②GAp：全局历史表和单地址预测表
 - ③PAg：单地址历史表和全局预测器表；④PAp：单地址历史表和单地址预测器表

两级全局预测器 (GAp)



- (2,2) predictor: 2-bit global, 2-bit local

Gshare predictor

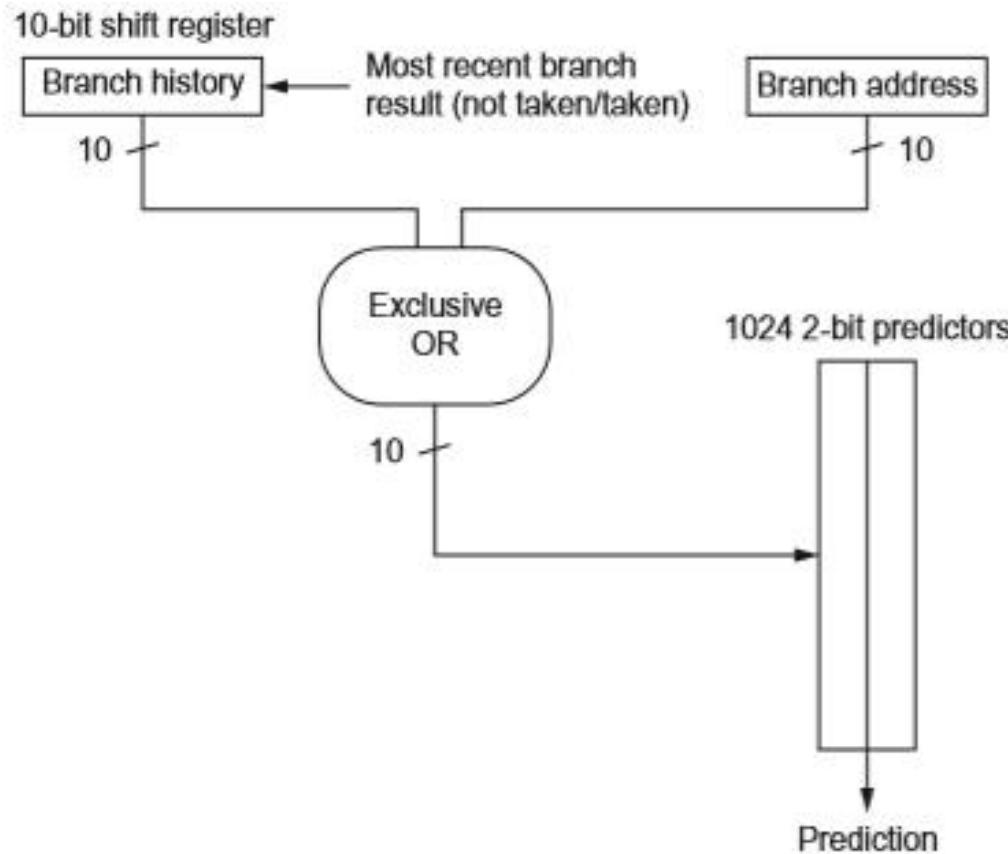
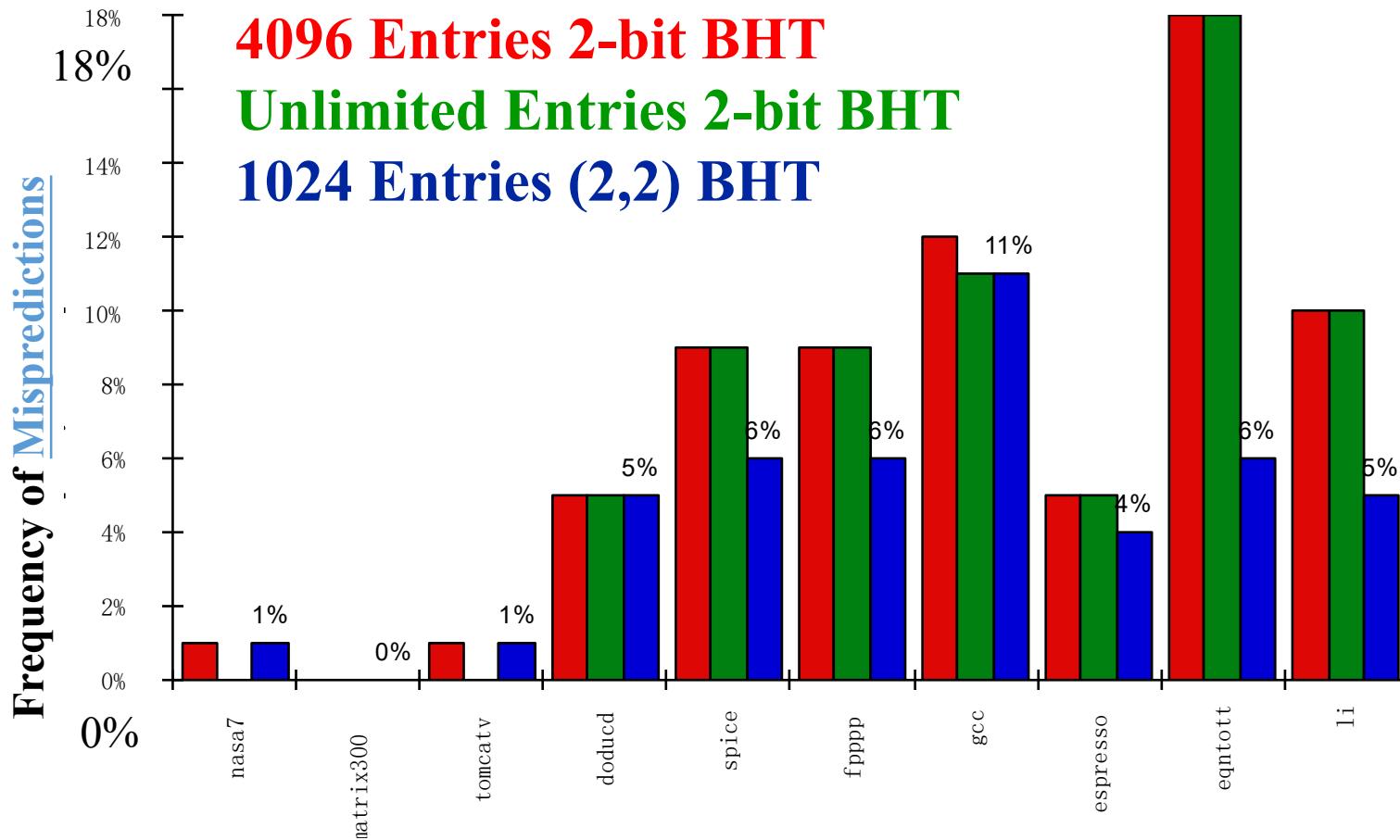


Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

带有全局分支历史的分支预测器Gshare（一种GAp的实现）

Accuracy of Different Schemes





Branch Prediction

- **Basic 2-bit predictor:**
- **关联预测器(n,2):**
 - 两级全局预测器 (GAp)
 - 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) PAp
 - 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 (组合) 预测器(Tournament predictor):**
 - 例如：结合两级全局预测器和两级局部预测器



竞赛 (组合) 预测器

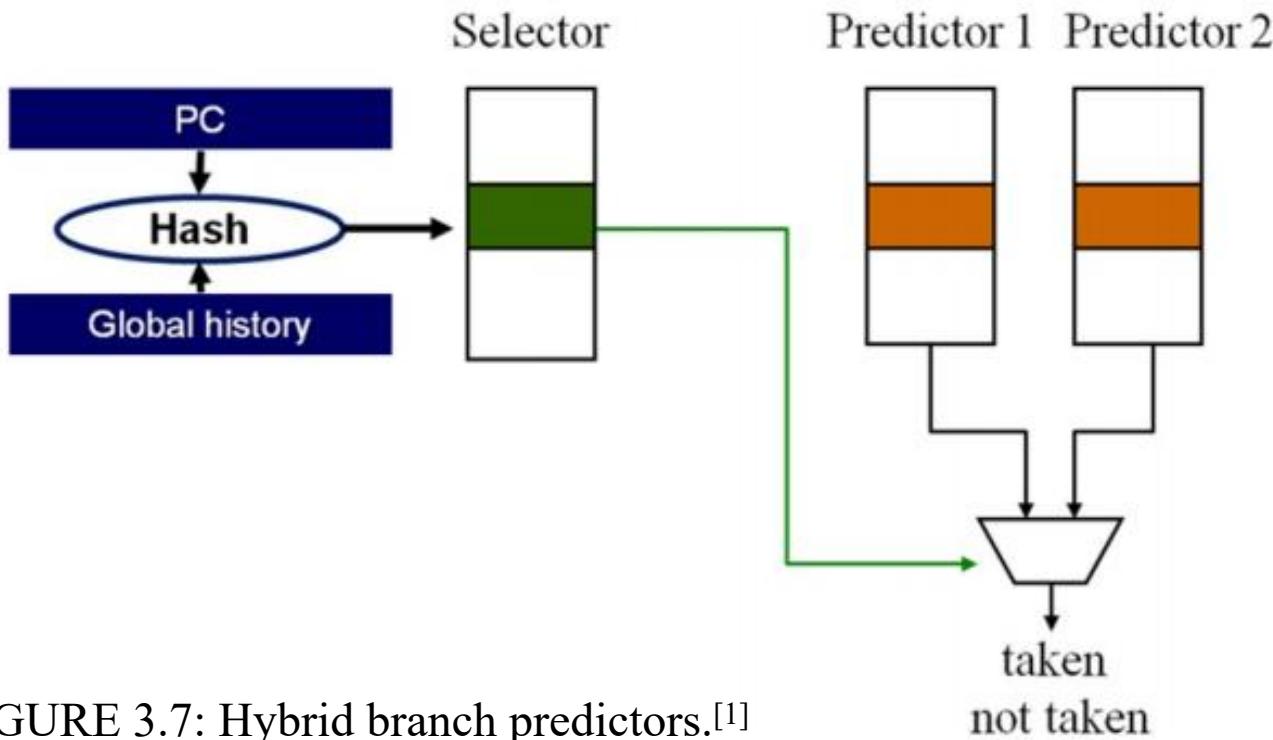
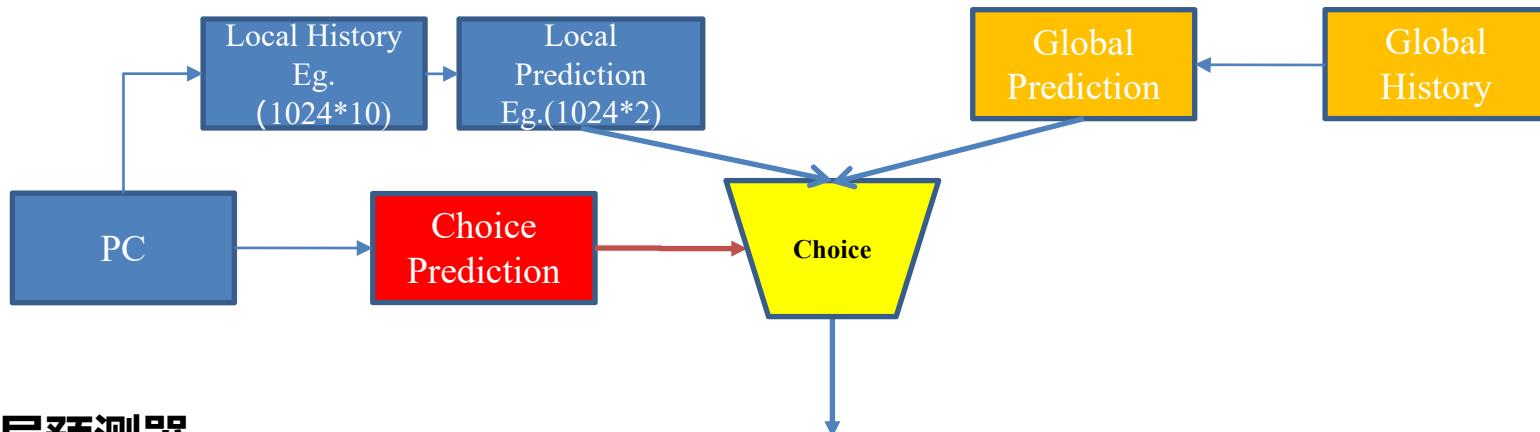


FIGURE 3.7: Hybrid branch predictors.^[1]

- [1] A González, Latorre F , Magklis G . Processor Microarchitecture: An Implementation Perspective[J]. Synthesis Lectures on Computer Architecture, 2010, 5(1).
- Tournament predictor 也称 Hybrid branch predictors
- 例如：两级局部预测器（预热时间较短）与全局预测器（预热时间较长）的组合

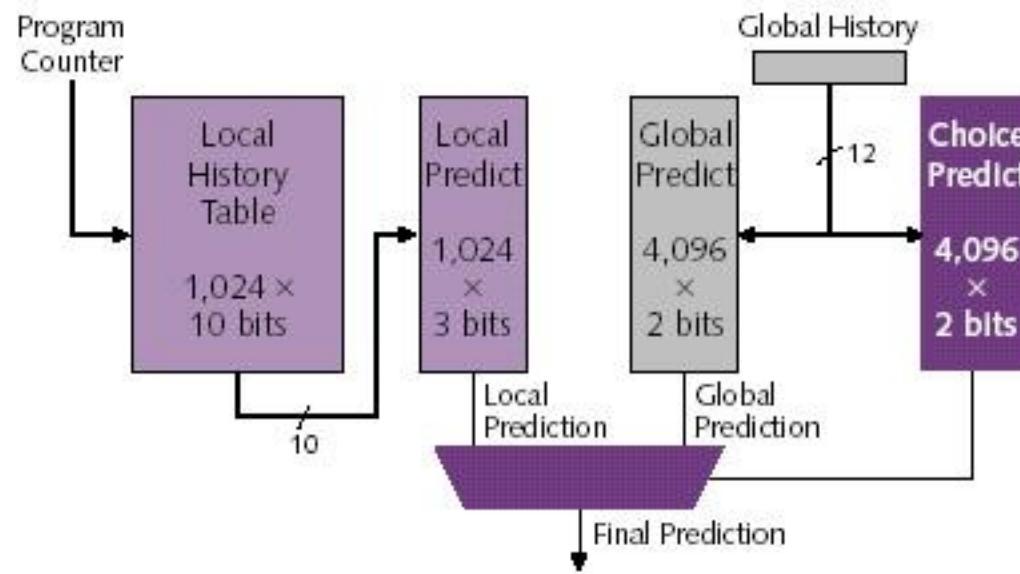
竞赛预测器（举例）



- **全局预测器**
 - 使用最近 n 次分支跳转情况来索引 (2^n 个entries)，每个Entry是一个标准的2位预测器
- **两级局部预测器**
 - 一个局部历史记录表 (**Local History**)：使用PC的低 m 位索引 (2^m 个entries)，每个entry有 k 位，记录该指令最近的 k 次分支跳转情况
 - 根据Local History选择的entry的 k 位，索引选择下一级 (**Local Prediction**) 的entries，这些entries由2位计数器构成，以提供本地预测。
- **选择器：**
 - 使用PC低 m 位索引，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。



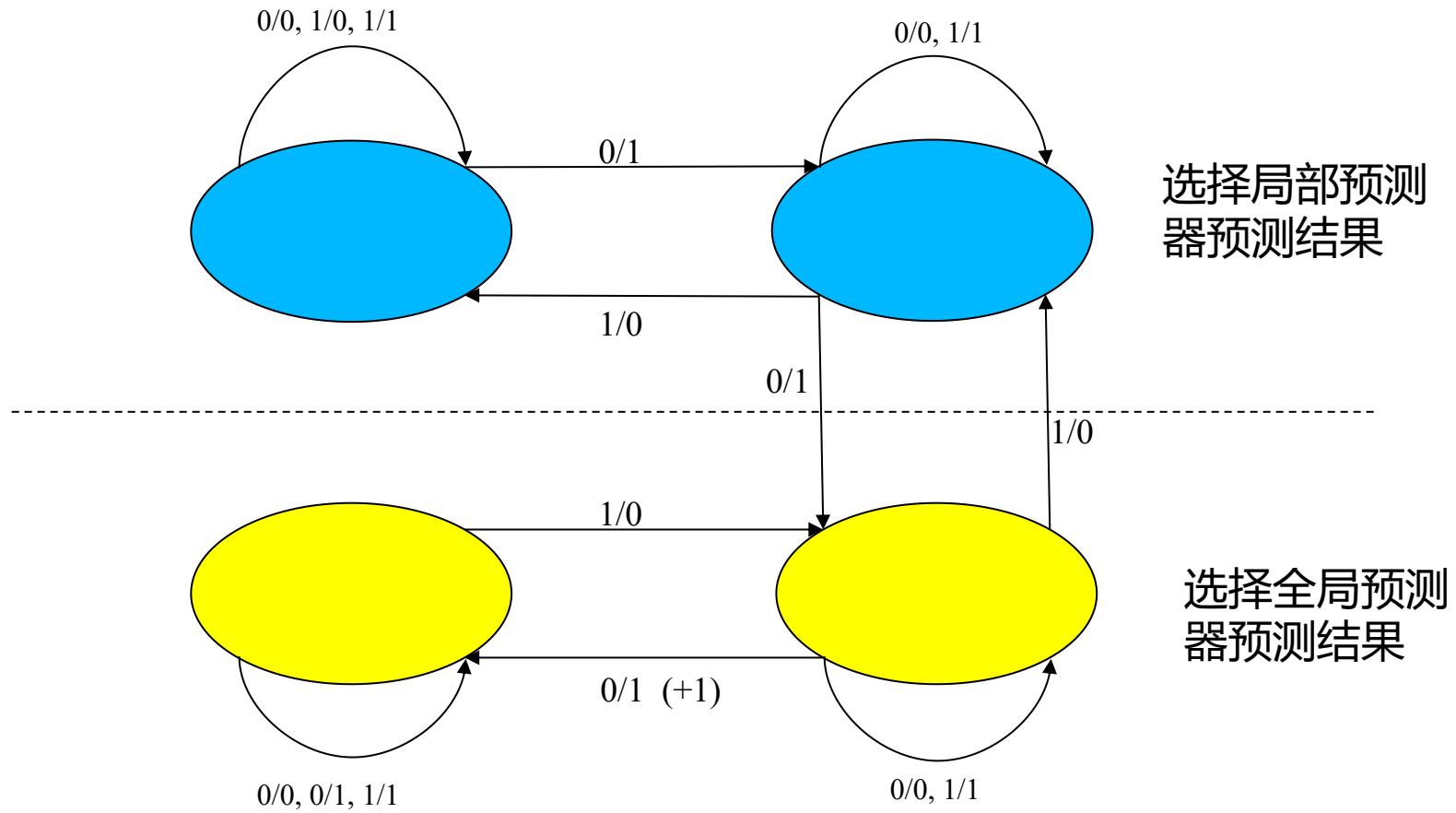
Alpha 21264 Branch Prediction Mechanism



Source: Microprocessor Report, 10/28/96

- **Minimum branch penalty: 7 cycles**
- **Typical branch penalty: 11+ cycles (IQ delay)**
- **48K bits of target addresses stored in I-cache**
- **32-entry return address stack**
- **Predictor tables are reset on a context switch**

选择器状态转移图

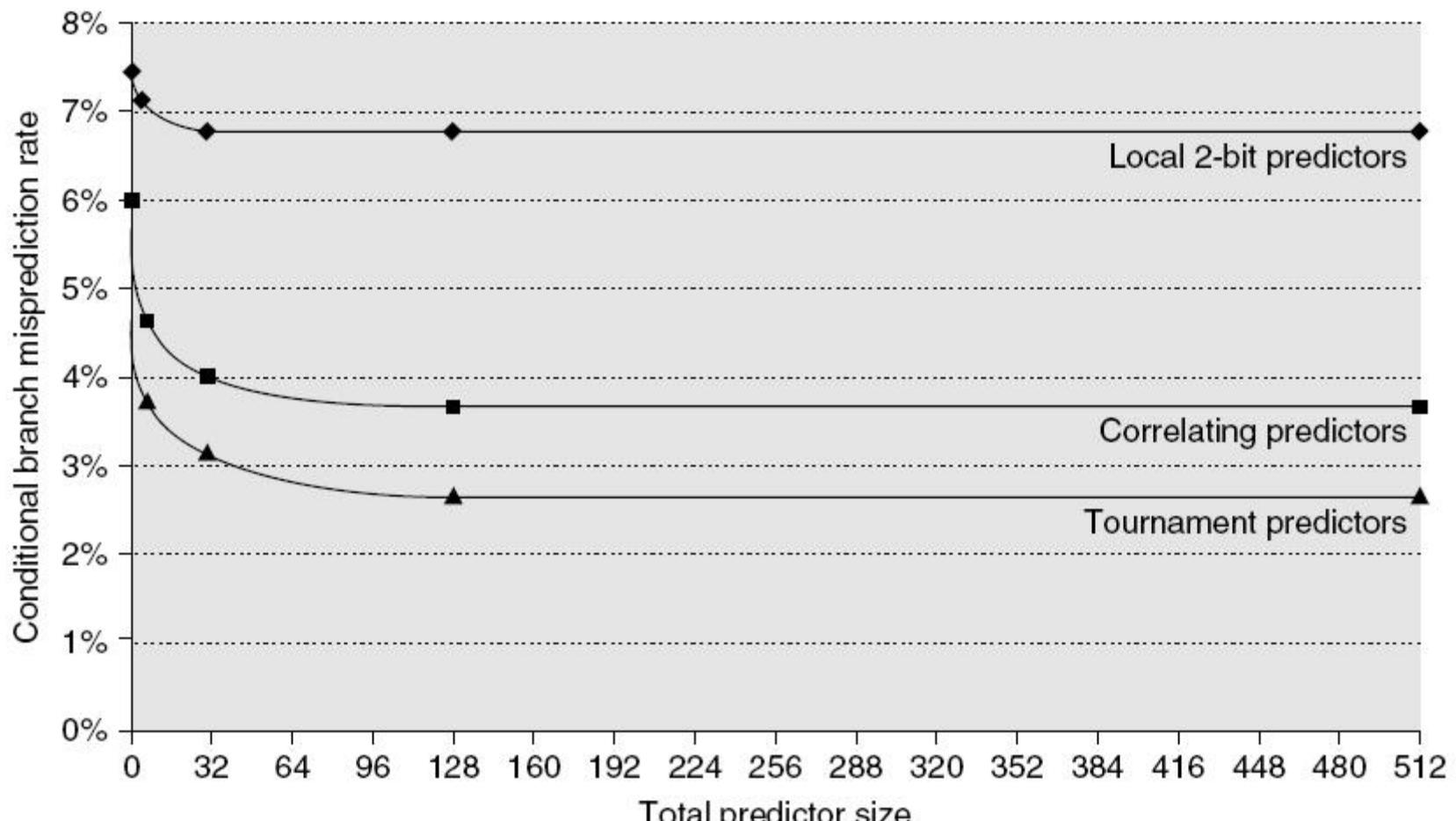


- **局部预测器/全局预测器：**

- 0/1 : 局部预测器预测错误, 全局预测器预测正确 (+1)
- 1/0 : 局部预测器预测正确, 全局预测器预测错误 (-1)



Branch Prediction Performance



Branch predictor performance



分支预测技术

控制相关对
性能的影响

基于BHT的
分支预测

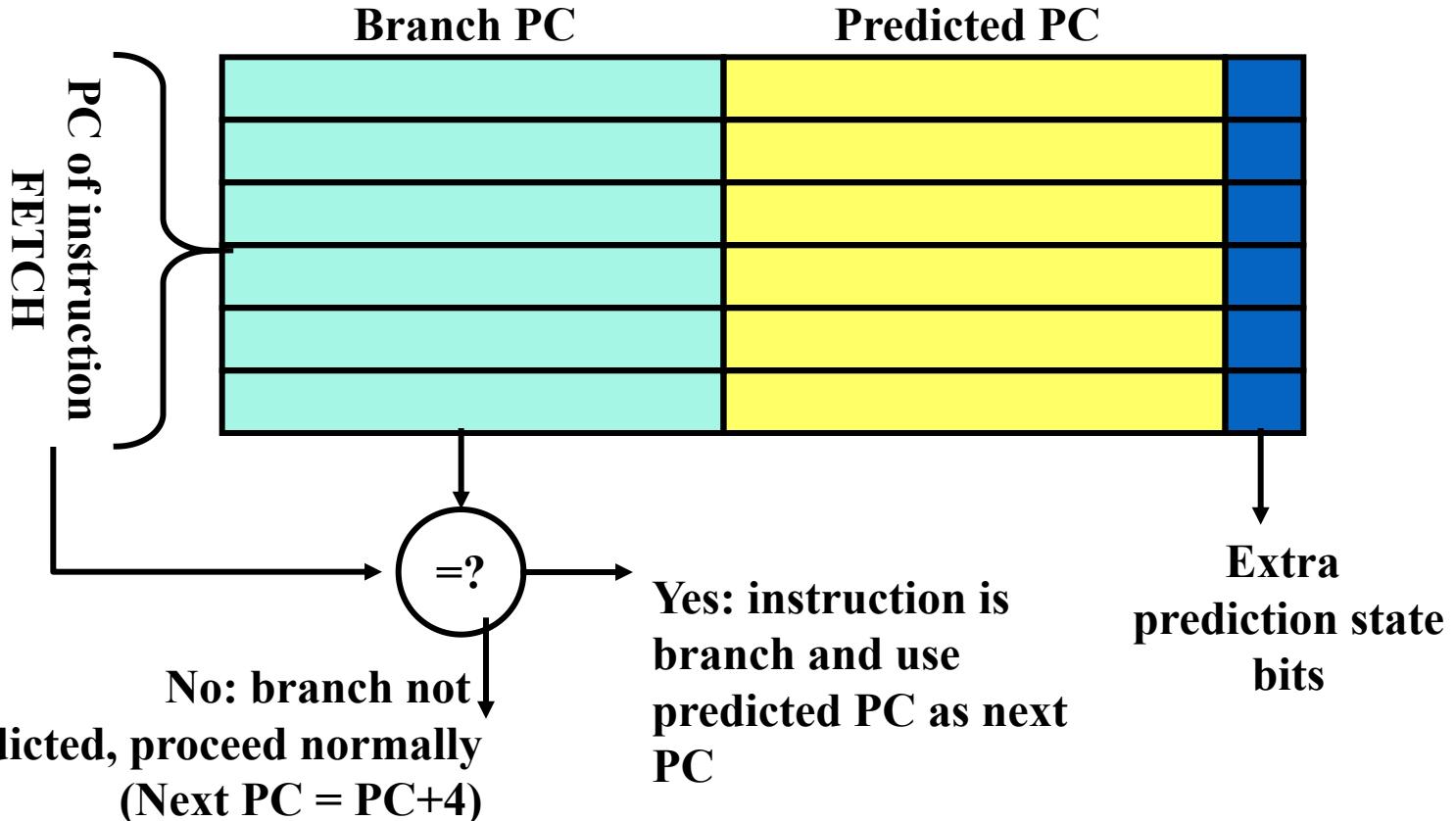
基于BTB的
分支预测

- 1、基本2-bit预测器
- 2、关联预测器（两级预测器）
- 3、组合预测器

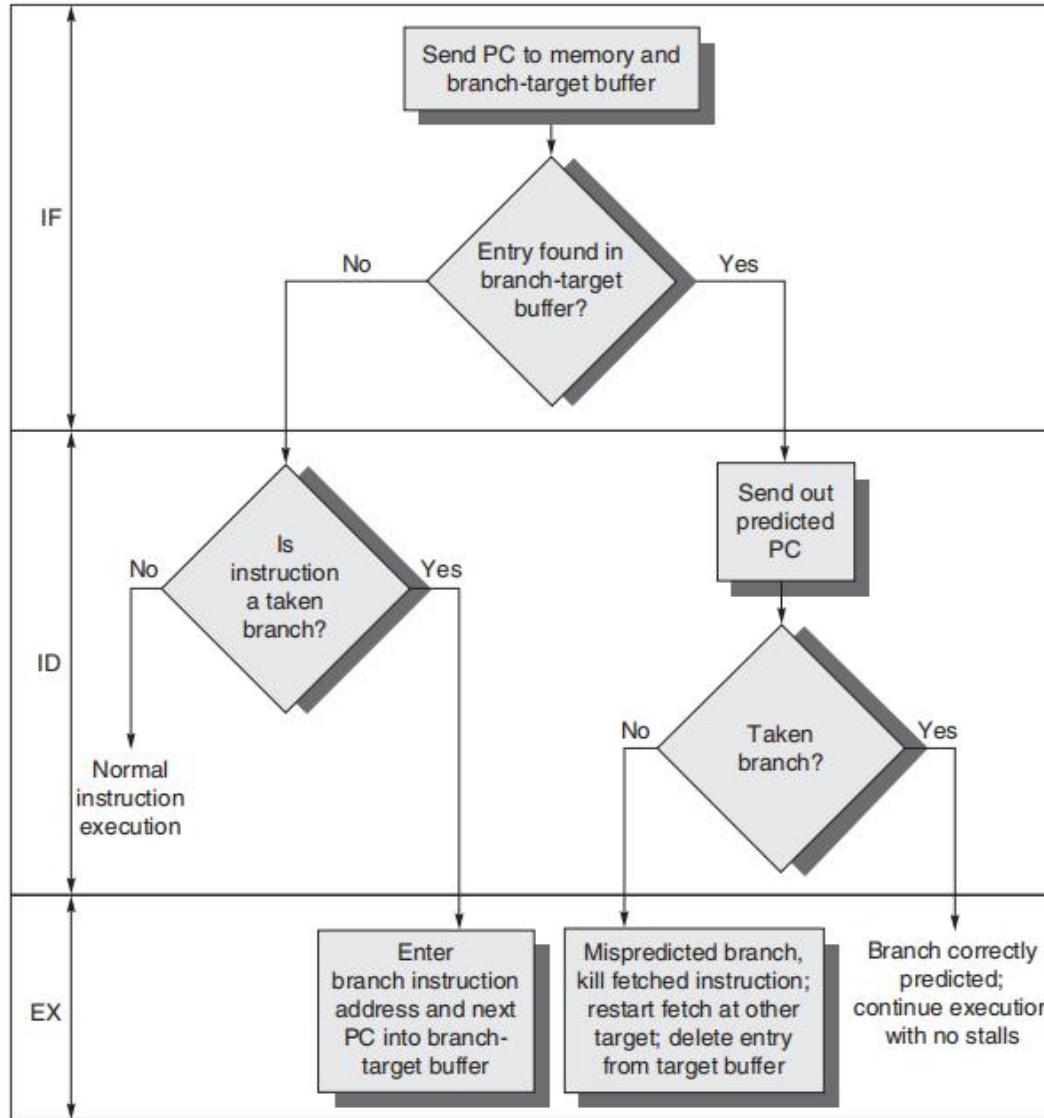
- 1、分支目标缓冲区
- 2、Return Address预测器

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



BTB的换入换出

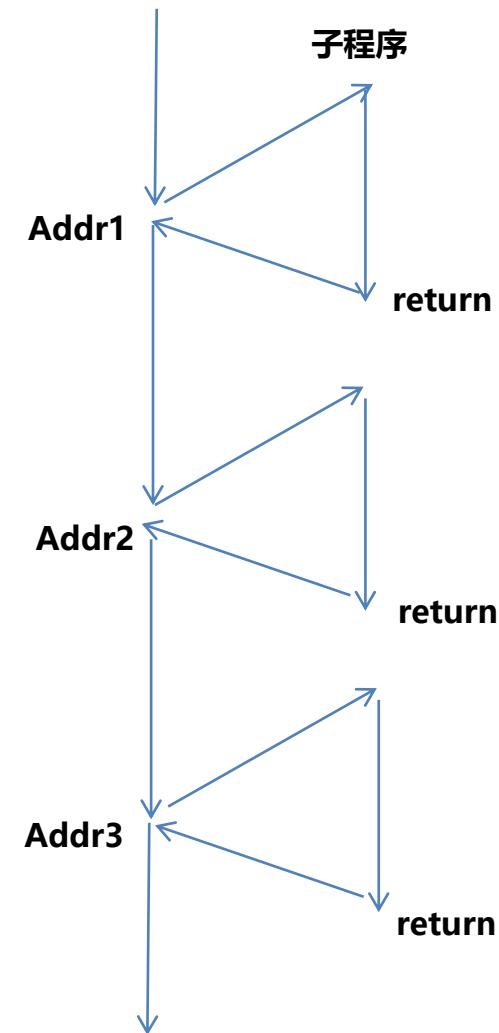


例如：基本模型
• 简单的五段流水
• ID段 确认 是否 可以 跳转
• BTB预测器 分支目标缓存的换入换出

Figure 3.22 The steps involved in handling an instruction with a branch-target buffer.

Return Address Predictors

- **投机执行面临的挑战：预测间接跳转**
 - 运行时才能确定分支目标地址
- **多数间接跳转来源于Procedure Return**
 - 采用BTB时，过程返回的预测精度较低
 - 如果采用BTB，BTB中存放的信息包括：return指令本身地址 (PC) ## 返回地址
 - 函数在不同位置调用，return指令本身地址不变，但返回地址不同
 - SPEC CPU95测试，这类分支预测的准确性不到60%
- **使用一个小的缓存(栈) 存放 Return Address**
 - 过程调用时将返回地址压入该栈
 - 过程返回时通过弹栈操作获得转移地址



Return Address Predictors

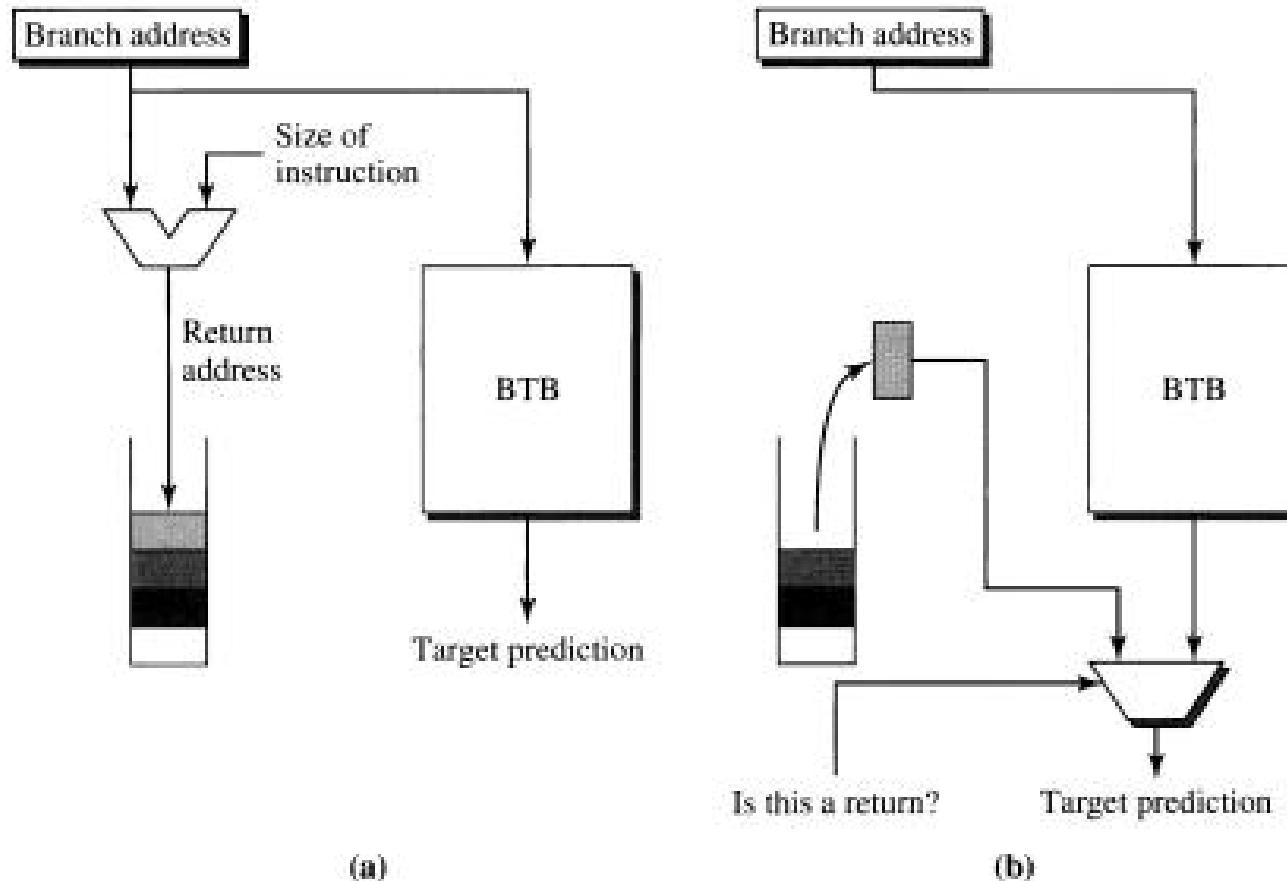


Figure 9.31

(a) Return Address Push on Jump to Subroutine. (b) Return Address Pop on Subroutine Return.

Return Address Buffer entries

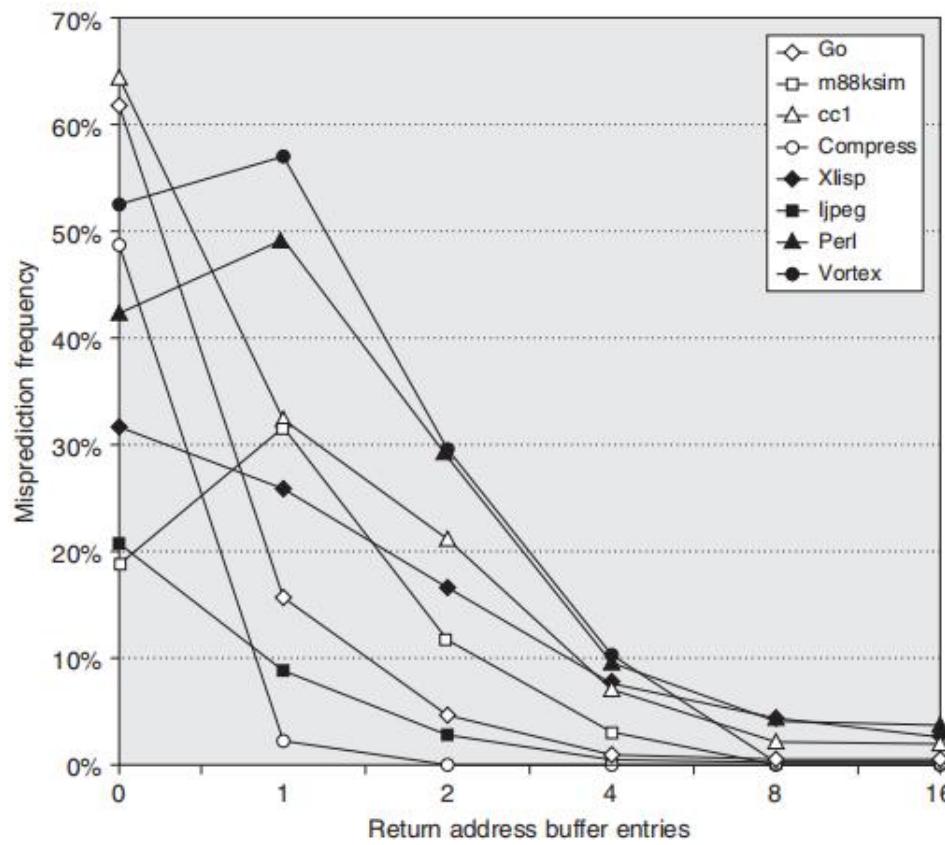


Figure 3.24 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Since call depths are typically not large, with some exceptions, a modest buffer works well. These data come from Skadron et al. [1999] and use a fix-up mechanism to prevent corruption of the cached return addresses.

- 返回栈 (Return Address Buffer)中表项数 (entries)与预测精度的关系

其他预测间接跳转目标地址方法

Case (a)

- 1: 跳转到目标地址1
- 2: 跳转到目标地址2
- 3: 跳转到目标地址3
-
- 9: 跳转到目标地址X

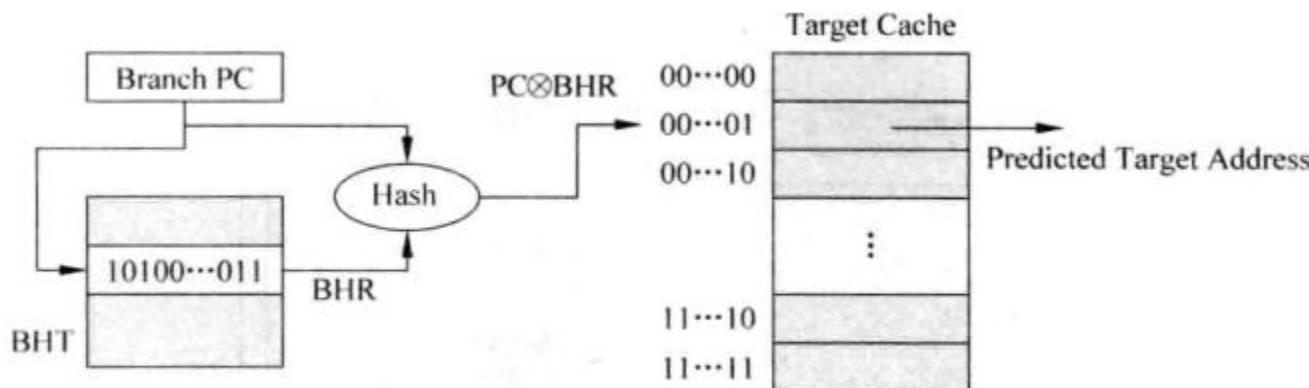


图 4.48 使用基于局部历史的分支预测方法对目标地址进行预测

其他预测器

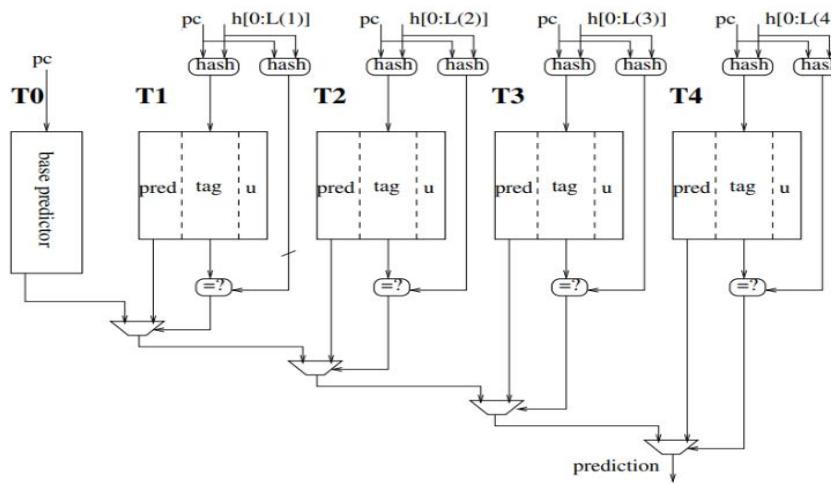


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

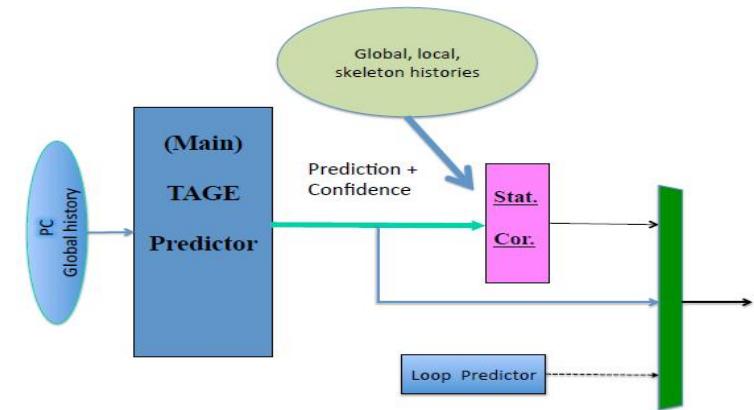


Figure 1. The TAGE-SC-L predictor: a TAGE predictor backed with a Statistical Corrector predictor and a loop predictor

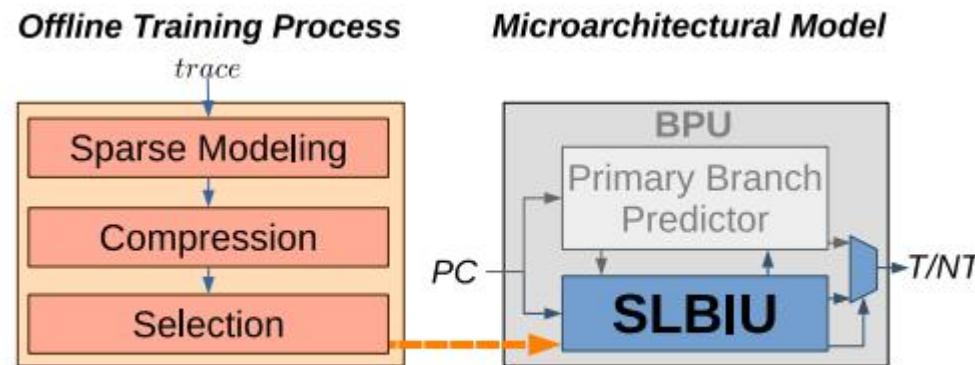


Figure 4: Sparsity-aware branch prediction overview.

- 1、Seznec A, Michaud P. A case for (partially) TAGged GEometric history length branch prediction[J]. The Journal of Instruction-Level Parallelism, 2006, 8: 23
- 2、Seznec A. Tage-sc-l branch predictors[C]//JILP-Championship Branch Prediction. 2014
- 3、Zouzias A, Kalaitzidis K, Berestizshevsky K, et al. Identifying and Exploiting Sparse Branch Correlations for Optimizing Branch Prediction[J]. arXiv preprint arXiv:2207.14033, 2022



Instruction Fetch Unit

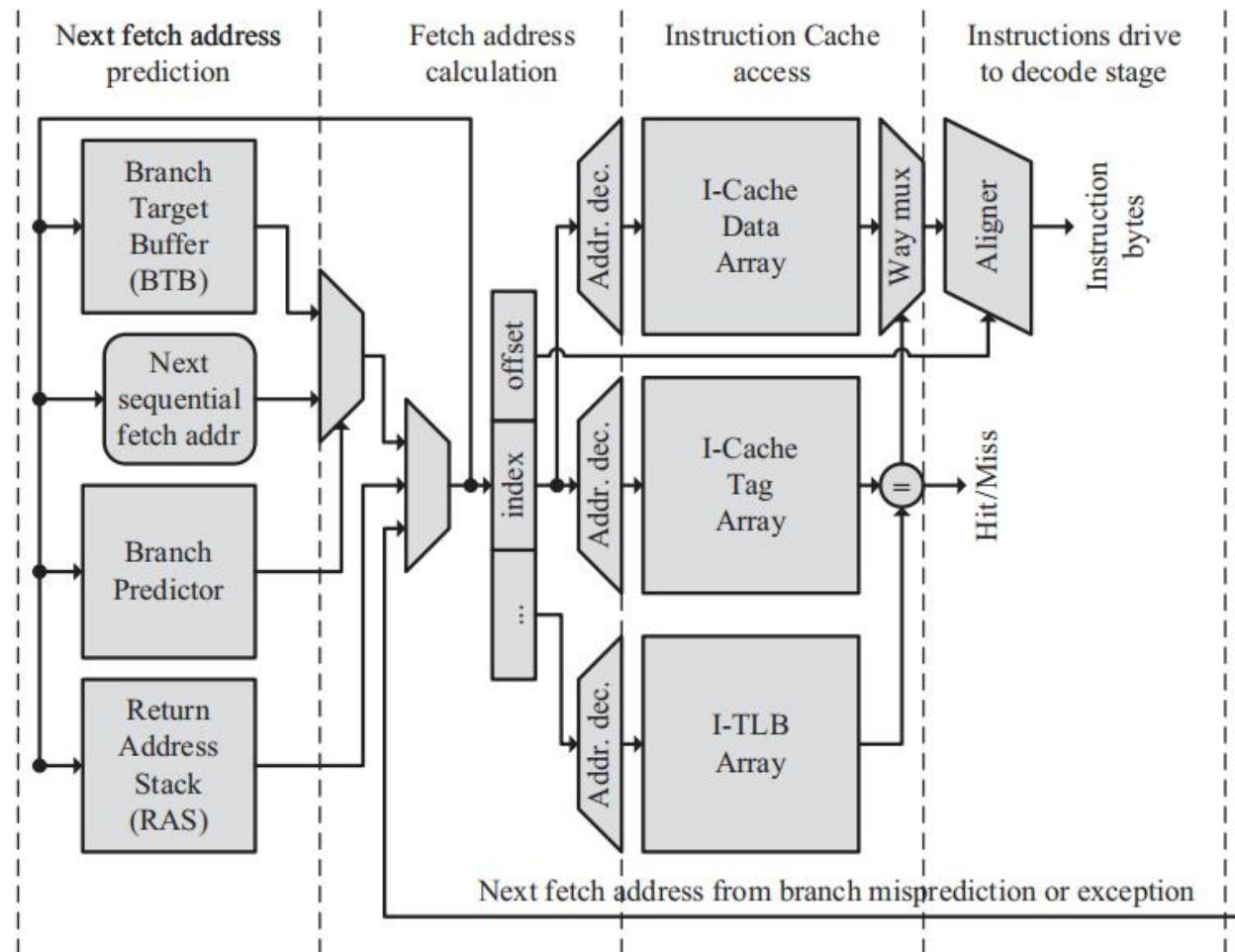


FIGURE 3.1: Example fetch pipeline.

分支预测小结

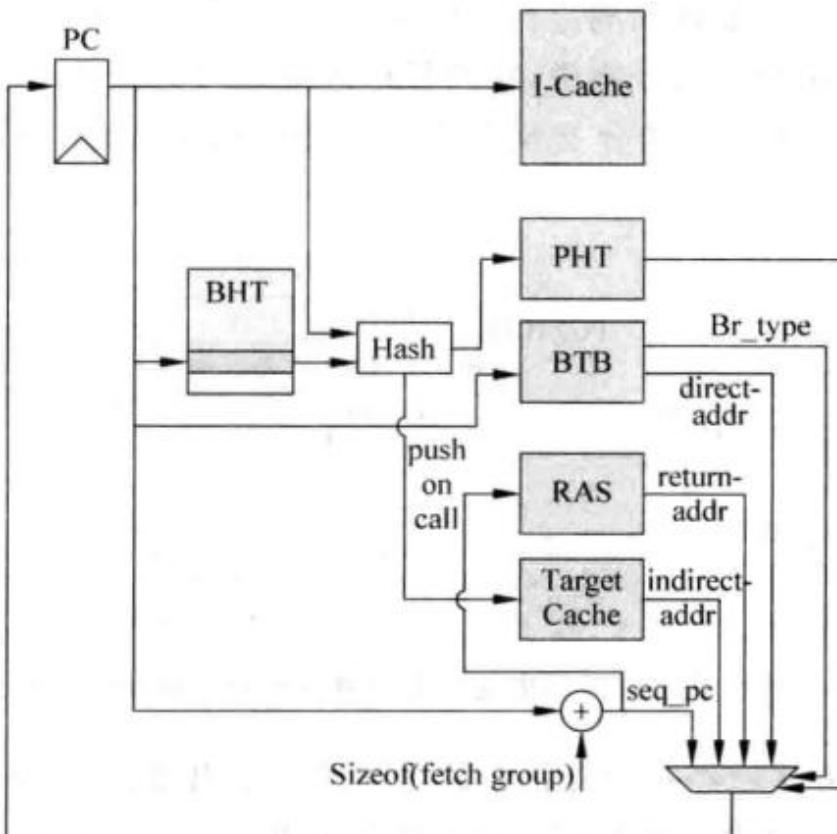


图 4.49 一种完整的分支预测方法

分支预测的核心价值与趋势：

分支预测是现代处理器性能的“隐形支柱”：

- 没有分支预测，超标量处理器的ILP会因频繁排空损失50%以上性能；
- 从1位预测到TAGE，错误率从60%降至1%，直接推动处理器IPC（每周期指令数）从1提升到8以上。

未来可能趋势：结合机器学习（如神经网络预测器）捕捉更长程、更复杂的分支模式，进一步降低错误率。



Part1：指令级并行

- **指令级并行：提高单条指令流（单线程）程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - 流水线的扩展：指令执行周期数(可能) 不同
 - 指令级并行优化：编译优化和硬件优化
 - 动态调度的“漏网之鱼”——控制相关与异常处理问题
 - 分支预测技术
 - **推断执行技术**
- **多发射流水线：ILP的突破 ($CPI < 1$)**
 - 每个cycle发射多条指令执行
- **多线程技术：单线程ILP瓶颈的突破**
 - 流水线执行的指令来源于多条指令流
- **其他：“发射”指每个cycle可进入功能部件（执行部件）的指令条数**



基于硬件的推断执行

推断执行

支持推断执行
的Tomasulo

代码执行
示例

Tomasulo
小结

- 1. 带有ROB的机器结构
- 2. 四阶段算法描述

- 1. 简单代码示例
- 2. 推断执行示例

- 1. ROB的作用
- 2. 动态内存歧义消除

分支预测失败时的恢复



精确异常与长流水线

- **例如**
 - DIVF F0,F2,F4
 - ADDF F10,F10,F8
 - SUBF F12,F12,F14
- **ADDF 和SUBF都在DIVF前完成**
- **如果DIVF导致异常，会如何？**
 - 非精确异常
- **Ideas???**



精确异常与非精确异常

- **精确异常**

- 如果流水线可以控制使得引起异常的指令前序指令都执行完，故障后的指令可以重新执行，则称该流水线支持精确异常
- 按照指令的逻辑序处理异常
- 理想情况，引起故障的指令没有改变机器的状态
- 要正确的处理这类异常请求，必须保证故障指令不产生副作用

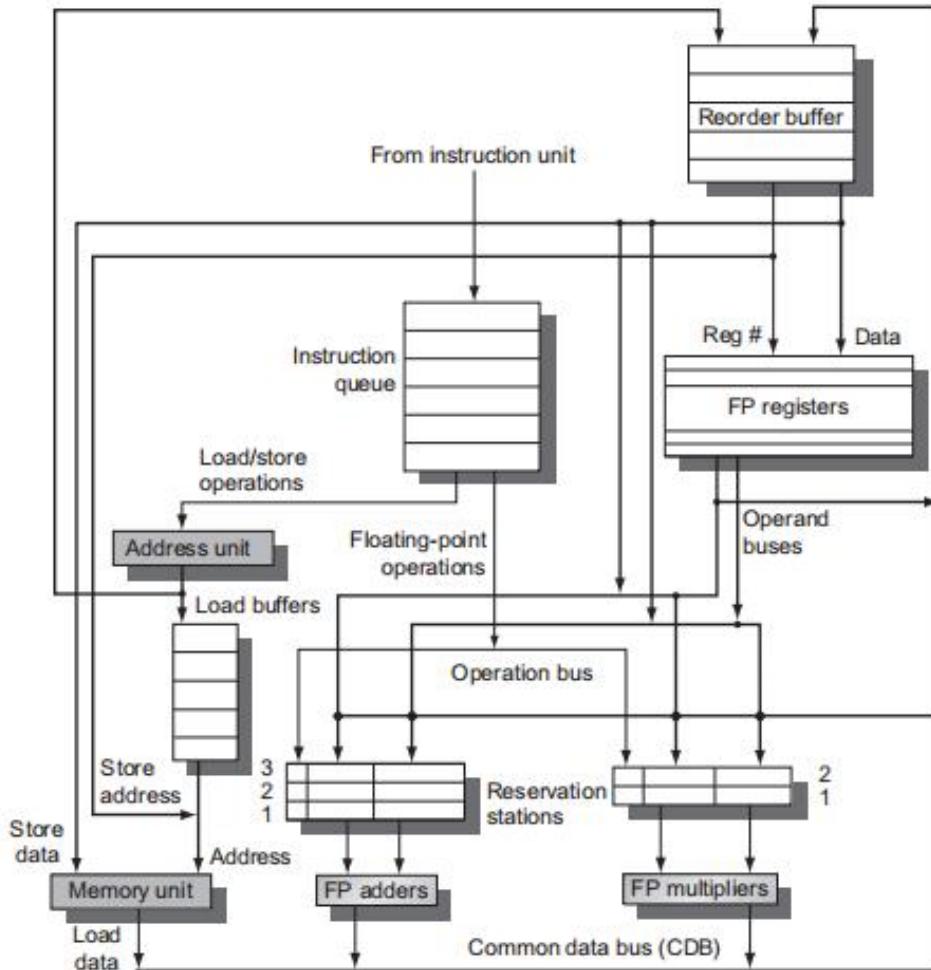
- **在有些机器上，浮点数异常**

- 流水线段数多，在发现故障前，故障点后的指令就已经写了结果，在这种情况下，必须有办法处理。
- 很多高性能计算机，Alpha 21164，MIPS R10000等支持精确中断，但精确模式要慢10+倍，一般用在代码调试时，很多系统要求精确异常模式，如IEEE FP标准处理程序，虚拟存储器等。

- **精确异常对整数流水线而言，不是太难实现，但注意：**

- 指令执行的中途改变机器的状态
- 例如IA-32 的自动增量寻址模式

使用Tomasulo算法，支持推断执行的基本结构



主要差异：

- 增加了 Reorder buffer
- 删除了 store buffer，其功能集成在 ROB 中

Figure 3.15 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.10 on page 198, which implemented Tomasulo's algorithm, we can see that the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to allow multiple issues per clock by making the CDB wider to allow for multiple completions per clock.



硬件支持推断执行以及精确异常

- **支持推断执行的条件：具有“恢复”能力**
- **(硬件) 缓存没有提交的指令结果: reorder buffer (ROB)**
 - 4个域: 指令类型, 目的地址, 值, ready域
 - Reorder buffer 可以作为操作数源 => 就像有更多的寄存器 (与RS类似)
 - 当指令执行阶段完成后, 用ROB的编号代替RS中的值
 - 增加指令提交阶段 (Commit)
 - ROB提供执行完成阶段和提交阶段的操作数
 - 一旦结果提交, 结果就写入寄存器
 - 在预测错误时, 容易恢复推断执行的指令, 或发生异常时, 容易恢复状态



支持推断执行的 Tomasulo 算法的四阶段

- **1. Issue—get instruction from FP Op Queue**
 - 如果RS和ROB有空闲单元就发射指令。如果寄存器或ROB中源操作数可用，就将其发送到RS，目的地址的ROB编号也发送给RS
- **2. Execution—operate on operands (EX)**
 - 当操作数就绪后，开始执行。如果没有就绪，监测CDB，检查RAW相关
- **3. Write result—finish execution (WB)**
 - 将运算结果通过CDB传送给所有等待结果的FU以及**ROB单元**，标识RS可用
- **4. Commit—update register with reorder result**
 - 按ROB表中顺序，如果结果已有，就更新寄存器（或存储器），并将该指令从ROB表中删除
 - 预测错误或有异常（中断）时，刷新ROB
 - P191 Figure 3.14 (英文版), P141 Figure 3-9 (中文版)
- **执行过程中需要检测CDB冲突**

	F0	F2	F4	F6	F8	F10	F12	F30
Reorder#									
Busy	No	No	No	No	No	No	No		No



Issue

Status	Wait until	Action or bookkeeping
Issue all instructions		<pre>if (RegisterStat[rs].Busy) /*in-flight instr. writes rs*/ { h ← RegisterStat[rs].Reorder; if (ROB[h].Ready) /* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd; ROB[b].Ready ← no;</pre>
FP operations and stores	Reservation station (r) and ROB (b) both available	<pre>if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ { h ← RegisterStat[rt].Reorder; if (ROB[h].Ready) /* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};</pre>
FP operations		<pre>RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;</pre>
Loads		<pre>RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;</pre>
Stores		<pre>RS[r].A ← imm;</pre>

rs: FP操作指令源操作数寄存器，Load/store指令的基址寄存器

rt: FP指令的源操作数寄存器，store操作的待写入的寄存器，load操作的目的寄存器

h: ROB中当前指令所依赖的指令对应的ROB编号；

b: 当前指令对应的ROB编号； **r:**当前指令对应的保留站编号



Execute

Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	$RS[r].A \leftarrow RS[r].Vj + RS[r].Ak;$
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj == 0) and store at queue head	$ROB[h].Address \leftarrow RS[r].Vj + RS[r].A;$

h: store对应的ROB entry编号
是store操作队列的head



Write result & Commit

Write result all but store	Execution done at r and CDB available	$b \leftarrow RS[r].Dest; RS[r].Busy \leftarrow no;$ $\forall x (\text{if } (RS[x].Qj == b) \{RS[x].Vj \leftarrow \text{result}; RS[x].Qj \leftarrow 0\});$ $\forall x (\text{if } (RS[x].Qk == b) \{RS[x].Vk \leftarrow \text{result}; RS[x].Qk \leftarrow 0\});$ <u>$ROB[b].Value \leftarrow \text{result}; ROB[b].Ready \leftarrow yes;$</u>
Store	Execution done at r and $(RS[r].Qk == 0)$	$ROB[h].Value \leftarrow RS[r].V_k;$ h: store对应的ROB entry编号
Commit	Instruction is at the head of the ROB (entry h) and $ROB[h].ready ==$ yes h: head of the ROB entry	$d \leftarrow ROB[h].Dest; /* register dest, if exists */$ $\text{if } (ROB[h].Instruction == \text{Branch})$ {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;};} $\text{else if } (ROB[h].Instruction == \text{Store})$ {Mem[ROB[h].Destination] $\leftarrow ROB[h].Value;$; } $\text{else /* put the result in the register destination */}$ {Regs[d] $\leftarrow ROB[h].Value;$; } $ROB[h].Busy \leftarrow no; /* free up ROB entry */$ /* free up dest register if no one else writing it */ $\text{if } (\text{RegisterStat}[d].Reorder == h) \{\text{RegisterStat}[d].Busy \leftarrow no;\};$



推断执行

支持推断执行
的Tomasulo

代码执行
示例

Tomasulo
小结

- 1. 带有ROB的机器结构
- 2. 四阶段算法描述

- 1. 简单代码示例
- 2. 推断执行示例

- 1. ROB的作用
- 2. 动态内存歧义消除



例如：

LD	F6, 34(R2)
LD	F2, 45(R3)
MULT	F0, F2, F4
SUBD	F8, F6, F2
DIVD	F10, F0, F6
ADDD	F6, F8, F2

假设：执行阶段的周期数

LD: 1 cycles MULT: 10 cycles

SUBD/ADDD: 2cycles DIVD: 40 cycles



Tomasulo With Reorder Buffer-Summary

Instruction	Issue	Exec Comp	WriteBack	Commit
LD F6,34(R2)	1	2	3	4
LD F2,45(R3)	2	3	4	5
MULT F0,F2,F4	3	5~14	15	16
SUBD F8,F6,F2	4	5~6	7	17
DIVD F10,F0,F6	5	16~55	56	57
ADDD F6,F8,F2	6	8~9	10	58

顺序发射、乱序执行、乱序完成、顺序提交



两种Tomasulo算法比较 (三阶段vs.四阶段)

Loop	L.S F0, 0(R1)
	L.S F1, 0(R2)
	ADD.S F2, F1, F0
	S.S F2, 0(R1)
	ADDI R1,R1, #4
	ADDI R2,R2, #4
	SUBI R3,R3,#1
	BNEZ R3, Loop

- 假设：

- Load和store部件：计算访存地址需要 2 cycle；对Cache访问 需要 1个cycle
- 浮点ADD执行：需要6个cycle
- Store操作内部分解为两个操作操作：**S.S-A 计算访存地址； S.S-D 对Cache访问**
- 其他整型类执行：需要2个cycle



Tomsasulo算法执行示例（无预测）

		Issue	Exe Start	Exe End	Cache	CDB	备注
I1	L.S F0, 0(R1)	1	2	3	(4)	(5)	
I2	L.S F1, 0(R2)	2	3	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	12	---	(13)	等待F1
I4	S.S-A F2, 0(R1)	4	5	6	---	---	
I5	S.S-D F2,0(R1)	5	14	15	(16)	---	等待F2
I6	ADDI R1,R2, #4	6	7	8	---	(9)	
I7	ADDI R2, R2,#4	7	8	9	---	(10)	
I8	SUBI R3, R3, #1	8	9	10	---	(11)	
I9	BNEZ R3, Loop	9	12	13	---	(14)	等待R3的值
I10	L.S F0, 0(R1)	15	16	17	(18)	(19)	等待I9
I11	L.S F1, 0(R2)	16	17	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	26	---	(27)	等待F1



Tomsasulo算法执行示例 (有预测)

		Issue	Exe Start	Exe End	Cache	CDB	Commit	备注
I1	L.S F0, 0(R1)	1	2	3	4	(5)	6	
I2	L.S F1, 0(R2)	2	3	4	5	(6)	7	
I3	ADD.S F2,F1,F0	3	7	12	---	(13)	14	等待F1
I4	S.S-A F2, 0(R1)	4	5	6	---	---		
I5	S.S-D F2,0(R1)	5	14	15	16	---	(17)	等待F2
I6	ADDI R1,R2, #4	6	7	8	---	(9)	(18)	
I7	ADDI R2, R2,#4	7	8	9	---	(10)	(19)	
I8	SUBI R3, R3, #1	8	9	10	---	(11)	(20)	
I9	BNEZ R3, Loop	9	14	15	---	(16)	(21)	等待R3的值，若第12拍或第13拍进入EXE段，则WR阶段 (CDB争用) 分别与I10, I11存在冲突
I10	L.S F0, 0(R1)	10	11	12	13	(14)	(22)	
I11	L.S F1, 0(R2)	11	12	13	14	(15)	(23)	
I12	ADD.S F2,F1,F0	12	16	21	---	(22)	(24)	等待F1



Memory Disambiguation： 处理对存储器引用的数据相关

- 内存访问消除歧义 (Memory Disambiguation)：计算机体系结构中用于确定内存操作 (load和store) 是否访问相同内存地址的一项技术
- Question: 给定一个指令序列, store, load 是否有关? 即下列代码是否有相关问题?
Eg: st 0(R2),R5
•
- ld R6,0(R3)
- 我们是否可以较早启动ld?
 - Store的地址可能会延迟很长时间才能得到.
 - 我们也许想在同一个周期开始这两个操作的执行.
- 两种方法:
 - No Speculation: 不进行load操作, 直到我们确信地址 $0(R2) \neq 0(R3)$
 - Speculation: 我们可以假设他们相关还是不相关 (called “dependence speculation”), 如果推测错误通过ROB来修正
- 参考书: Gonzalez, A., et al. (2011). “Processor Microarchitecture: An Implementation Perspective.” Synthesis Lectures on Computer Architecture #12, Morgan & Claypool Publishers



Memory Disambiguation

TABLE 6.1: Memory disambiguation schemes.

NAME	SPECULATIVE	DESCRIPTION
Total Ordering	No	All memory accesses are processed in order.
Partial Ordering	No	All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address.
Load Ordering	No	Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them.
Store Ordering	Yes	Stores execute in order, but loads execute completely out of order.

- **非投机方式的基本原则：当前存储器指令之前的store指令计算存储器地址后，才能执行当前的存储器操作**



Tomasulo Loop Example

Loop:	LD F0, 0(R1)
	MULTD F4, F0, F2
	SD F4, 0(R1)
	SUBI R1,R1,#8
	BNEZ R1 Loop

- **假设**

- Load和store部件：计算访存地址需要 2 cycle；对Cache访问 需要 1个 cycle
- 浮点操作执行：需要6个cycle
- Store操作内部分解为两个操作操作：SD-A 计算访存地址； SD-D 对 Cache访问
- 其他整型类执行：需要2个cycle



Tomsasulo算法执行示例 (Total Ordering - 无推断)

		Issue	Exe Start	Exe End	Cache	CDB	备注
I1	LD F0, 0(R1)	1	2	3	(4)	(5)	
I2	MULTD F4, F0, F2	2	6	11	--	(12)	等待F0 (I1)
I3	SD-A, F4, 0(R1)	3	4	5	---	---	
I4	SD-D F4, 0(R1)	4	13	14	(15)	---	等待F4 (I2)
I5	SUBI R1, R1, #8	5	6	7	---	(8)	
I6	BNEZ R1, Loop	6	9	10		(11)	等待R1 (i5)
I7	LD F0, 0(R1)	12	14	15	(16)	(17)	考虑CDB冲突(与SD-D指令)，从14拍开始计算地址
I8	MULTD F4, F0, F2	13	18	23	--	(24)	
I9	SD-A, F4, 0(R1)	14	15	16	---	---	
I10	SD-D F4, 0(R1)	15	25	26	(27)	(28)	
I11	SUBI R1, R1, #8	16	17	18	---	(19)	
I12	BNEZ R1, Loop	17	20	21	---	(22)	

- Load和store: 计算访存地址 2 cycle; 对Cache访问 1个cycle
- 浮点操作执行: 6个cycle ; 其他整型类: 2个cycle



Load ordering, store ordering - 分支预测

		Issue	Exe Start	Exe End	Cache	CDB	commit	备注
I1	LD F0, 0(R1)	1	2	3	(4)	(5)	6	
I2	MULTD F4, F0, F2	2	6	11	--	(12)	13	等待F0 (I1)
I3	SD-A, F4, 0(R1)	3	4	5	---	---	14	
I4	SD-D F4, 0(R1)	4	13	14	(15)	---	16	等待F4 (I2)
I5	SUBI R1, R1, #8	5	6	7	---	(8)	17	
I6	BNEZ R1, Loop	6	11	12		(13)	18	CDB冲突
I7	LD F0, 0(R1)	7	8	9	(10)	(11)	19	
I8	MULTD F4, F0, F2	8	12	17	--	(18)	20	
I9	SD-A, F4, 0(R1)	9	10	11	---	---	21	
I10	SD-D F4, 0(R1)	10	19	20	(21)	(22)	23	等待F4 (I8)
I11	SUBI R1, R1, #8	11	12	13	---	(14)	24	
I12	BNEZ R1, Loop	12	15	16	---	(17)	25	

- Load和store: 计算访存地址 2 cycle; 对Cache访问 1个cycle
- 浮点操作执行: 6个cycle ; 其他整型类: 2个cycle



推断执行

支持推断执行
的Tomasulo

代码执行
示例

Tomasulo
小结

1. 带有ROB的机器结构
2. 四阶段算法描述

1. 简单代码示例
2. 推断执行示例

1. ROB的作用
2. 动态内存歧义消除



使用ROB保持机器的精确状态

- **ROB维持了机器的精确状态，允许投机（推测）执行**
 - 直到确认无异常 然后进入提交阶段
 - 直到确定分支预测正确进入提交阶段
 - 如果有异常或预测错误
 - 刷新ROB、RS和寄存器结果状态表
- **存储器操作使用类似的方法**
 - Memory Ordering Buffer (MOB)
 - Store操作的结果先存放到MOB中，然后提交阶段按存储操作的程序序提交



Summary-Tomasulo小结 #1/3

- **Reservations stations:** 寄存器重命名，缓冲源操作数
 - 避免寄存器成为瓶颈
 - 避免了Scoreboard中无法解决的 WAR, WAW hazards
 - 允许硬件做循环展开
 - 不限于基本块(快速解决控制相关)
- **Reorder Buffer:**
 - 提供了撤销指令运行的机制
 - 指令以发射序存放在ROB中
 - 指令顺序提交
- **分支预测对提高性能是非常重要的**
 - 推断执行: 在控制相关还没有解决情况下, 就开始执行
 - 推断执行利用了ROB撤销指令执行的机制
 - 处理预测错误时, 撤销 推测执行的指令
 - 基于BHT的分支预测技术 (预测分支方向)
 - 基于BTB的分支预测技术 (预测分支目标地址)



Summary-Tomasulo小结

#2/3

- **贡献**
 - Dynamic scheduling
 - Register renaming
 - Load/store disambiguation
- **360/91 后 Pentium II; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264使用这种技术**
- **不足之处:**
 - Too many value copy operations
 - Register File → RS → ROB → Register File
 - Too many muxes/busses (CDB)
 - Values are from everywhere to everywhere else!
 - Reservation Stations mix values(data) and tags (control)
 - Slow down max clock frequency



- **存储器访问的冲突消解**

- 非投机方式的冲突消解

- Total Ordering
 - Partial Ordering

- Load指令前的store指令已经完成了地址计算，有可能乱序执行存储器load操作

- Load Ordering, Store Ordering
 - Load指令前的存储器访问指令已经完成了地址计算，load队头的load操作有可能在store指令之前执行访存操作。

- 投机方式的执行

- Store Ordering
 - 假设Load操作与之前未计算出有效地址的store操作无关。



Summary

- **基于BHT表的预测器:**

- Basic 2-bit predictor:
- Global 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：两级全局预测器与两级局部预测器

- **优化取指令的带宽**

- 基于BTB的分支预测器
- Return Address Stack
- 集成的独立的取指部件



存储器访问冲突消解



Memory Disambiguation

TABLE 6.1: Memory disambiguation schemes.

NAME	SPECULATIVE	DESCRIPTION
Total Ordering	No	All memory accesses are processed in order.
Partial Ordering	No	All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address.
Load Ordering	No	Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them.
Store Ordering	Yes	Stores execute in order, but loads execute completely out of order.

- **非投机方式的基本原则：当前存储器指令之前的store指令计算存储器地址后，才能执行当前的存储器操作**

Load Ordering. Store Ordering

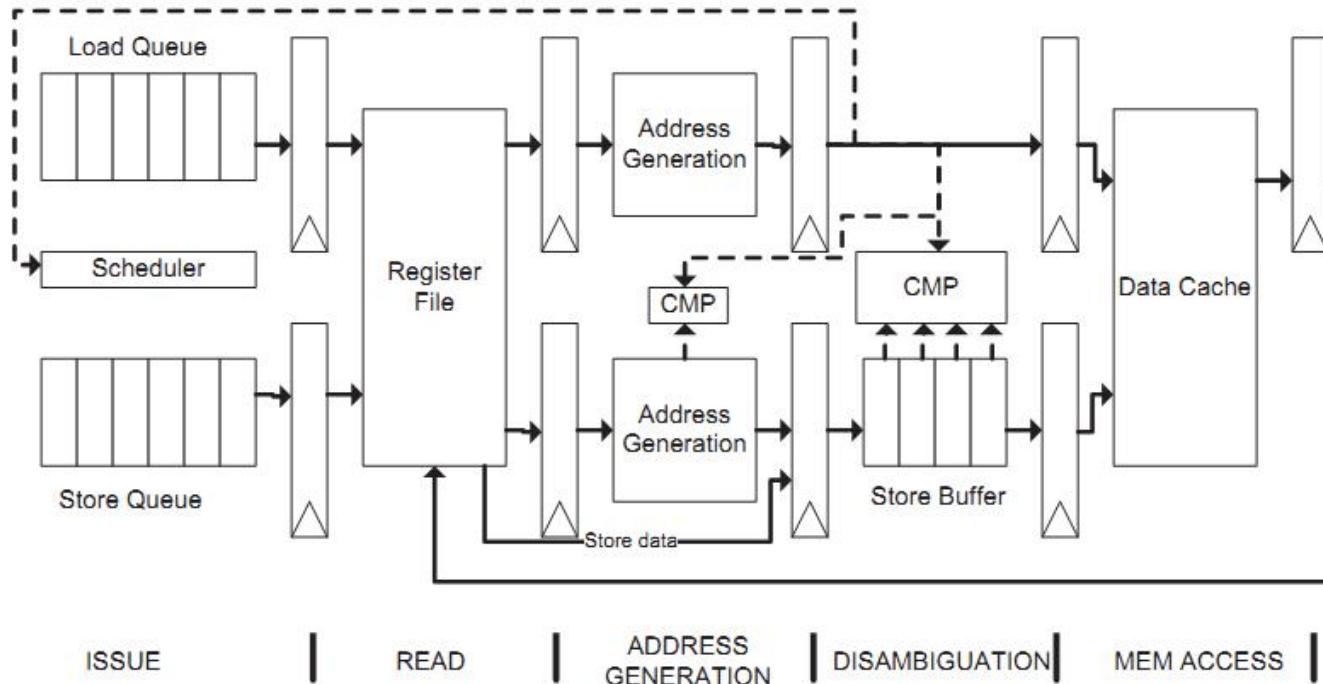


FIGURE 6.7: Schematics of the AMD K6 pipeline to implement a load-ordering store-ordering memory disambiguation policy.

Load queue: 以程序序存储load指令，按照FIFO方式流出

Address generation: 生成存储器访问有效地址

Store queue: 以程序序存储store指令，按照FIFO方式流出

Store buffer: 以程序序保留store操作（有效地址，值），**一直到其成为最早的store操作，才实际更新存储器**

Load 操作

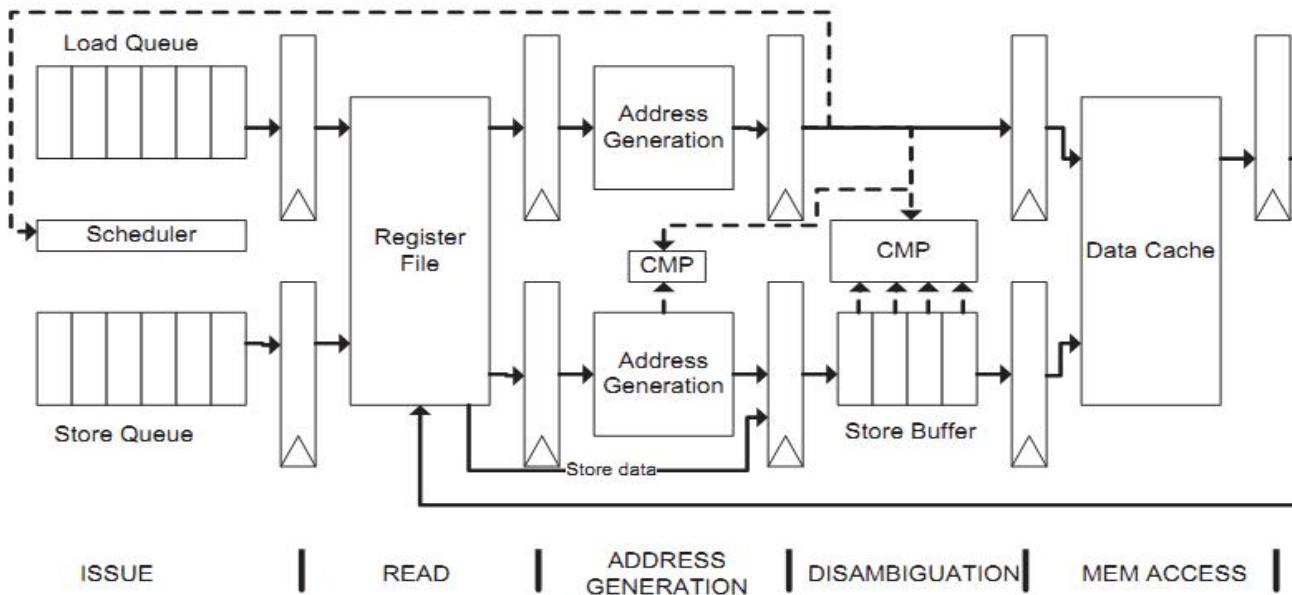


FIGURE 6.7: Schematics of the AMD K6 pipeline to implement a load-ordering store-ordering memory disambiguation policy.

- 处于Load队列对头，并且计算地址的操作数准备好，就向前流动；在Address Generation阶段计算地址
- **Disambiguation阶段：** Load操作可以继续前行需同时满足的三个条件
 - 在StoreBuffer中比该Load操作早的Store的地址与该Load地址不同
 - 如果正在进行地址计算的Store操作比该Load操作早，进行部分地址比较时，部分地址不同
 - 在StoreQueue中不存在比该Load早的Store操作。

Store 操作

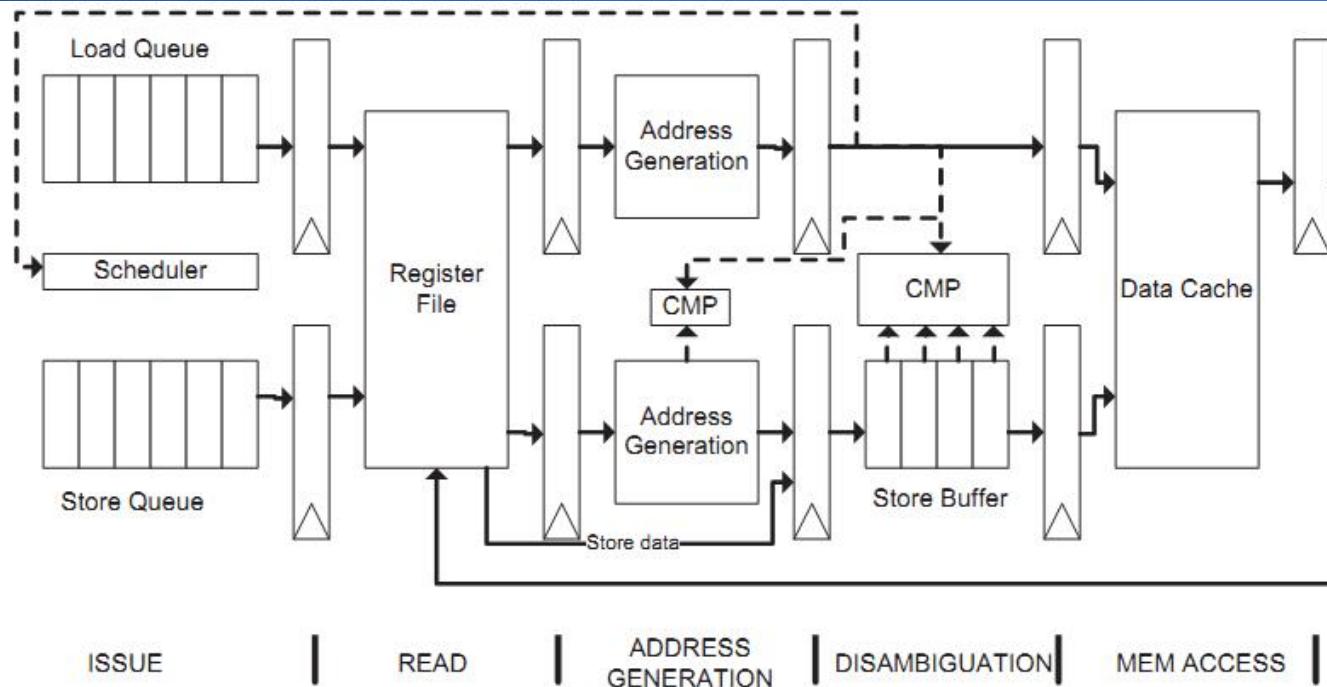


FIGURE 6.7: Schematics of the AMD K6 pipeline to implement a load-ordering store-ordering memory disambiguation policy.

- 处于Store队列对头，并且计算地址的操作数准备好，就向前流动，在Address Generation阶段计算地址
- 若要存储的数据准备好，则读该数据，若未准备好，则流水线停顿
- Disambiguation阶段：StoreBuffer中按程序序，执行最先的Store操作

Partial Ordering (MIPS R10000)

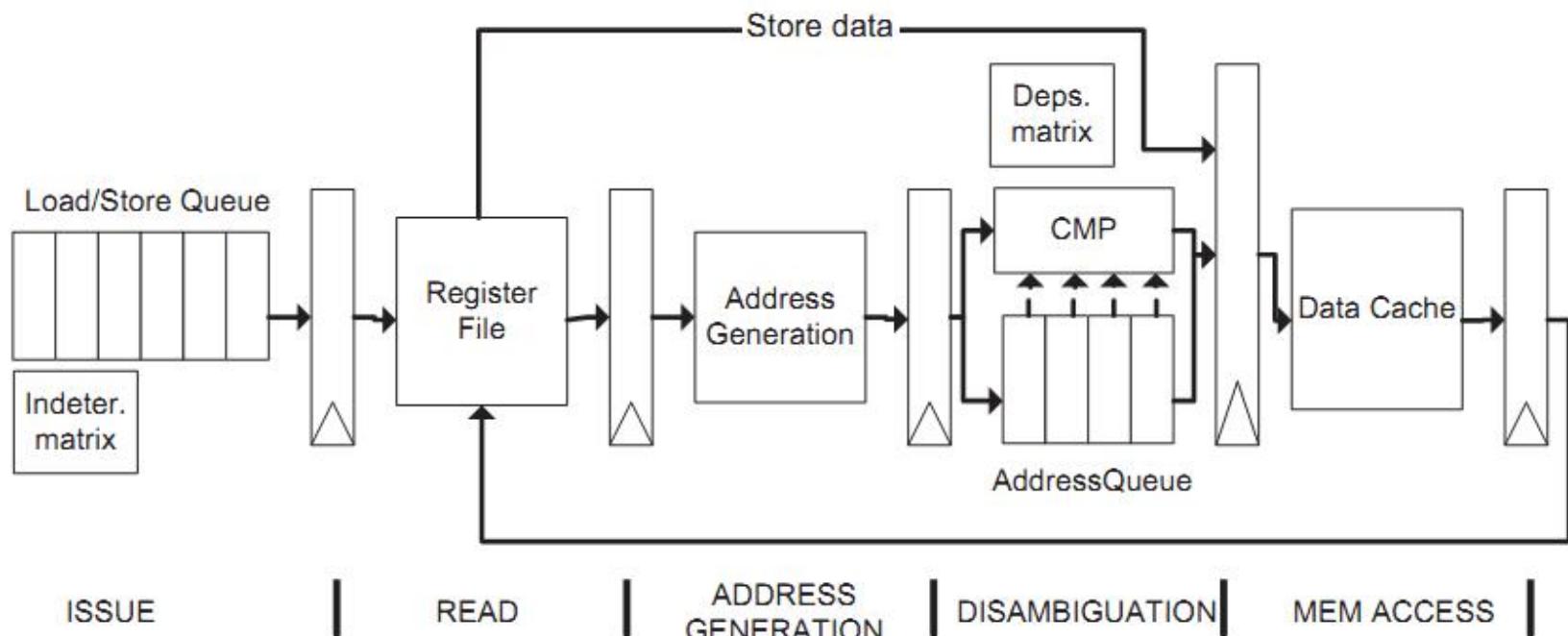


FIGURE 6.8: Schematics of the MIPS R10000 pipeline to implement the partial ordering memory disambiguation policy.

Load/store queue: 长度为16的队列，存储load/store指令，直到其操作数准备好。

Address generation: 计算存储器操作的有效地址

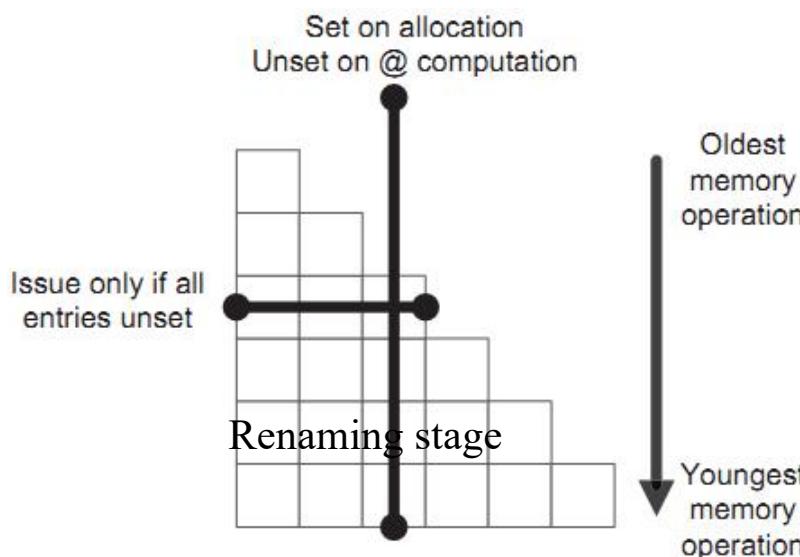


FIGURE 6.9: Example of a 6-entry indetermination matrix.

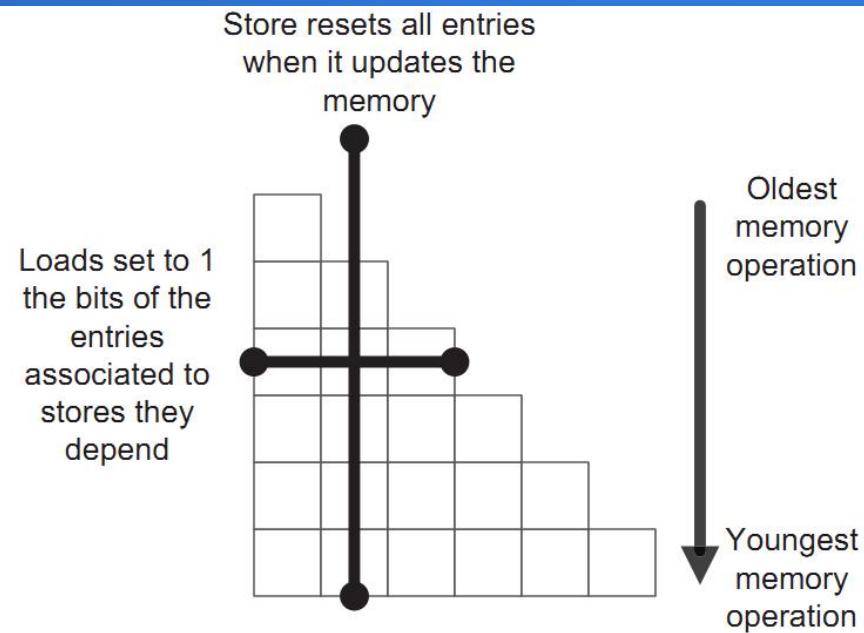


FIGURE 6.10: Example of a 6-entry dependency matrix.

- **Indetermination matrix:** 16x16 矩阵 (half), 每行每列代表队列中的存储器操作。当有存储器操作进入队列时，对应列置位。当存储器指令计算有效地址时，对应列复位。计算有效地址的条件之一，该操作对应行中非对角线元素为0 (按序计算有效地址)
- **Dependency matrix:** 16x16 矩阵，每行每列对应load/store队列的存储器操作。Load操作如果依赖于前面的store，则将该load操作的行中对应该store的列置位。按程序序执行Store操作，Store操作更新存储器时，将该store操作对应的列复位。只有当load操作对应的行全0时，方可执行load操作
- **Address queue:**保存访问cache的loads/store操作的地址。如果是load操作，除了保存地址，还需要比较所有在该load操作之前的store的地址，如果匹配，则对dependency matrix对应位置位。

Speculative Memory Disambiguation

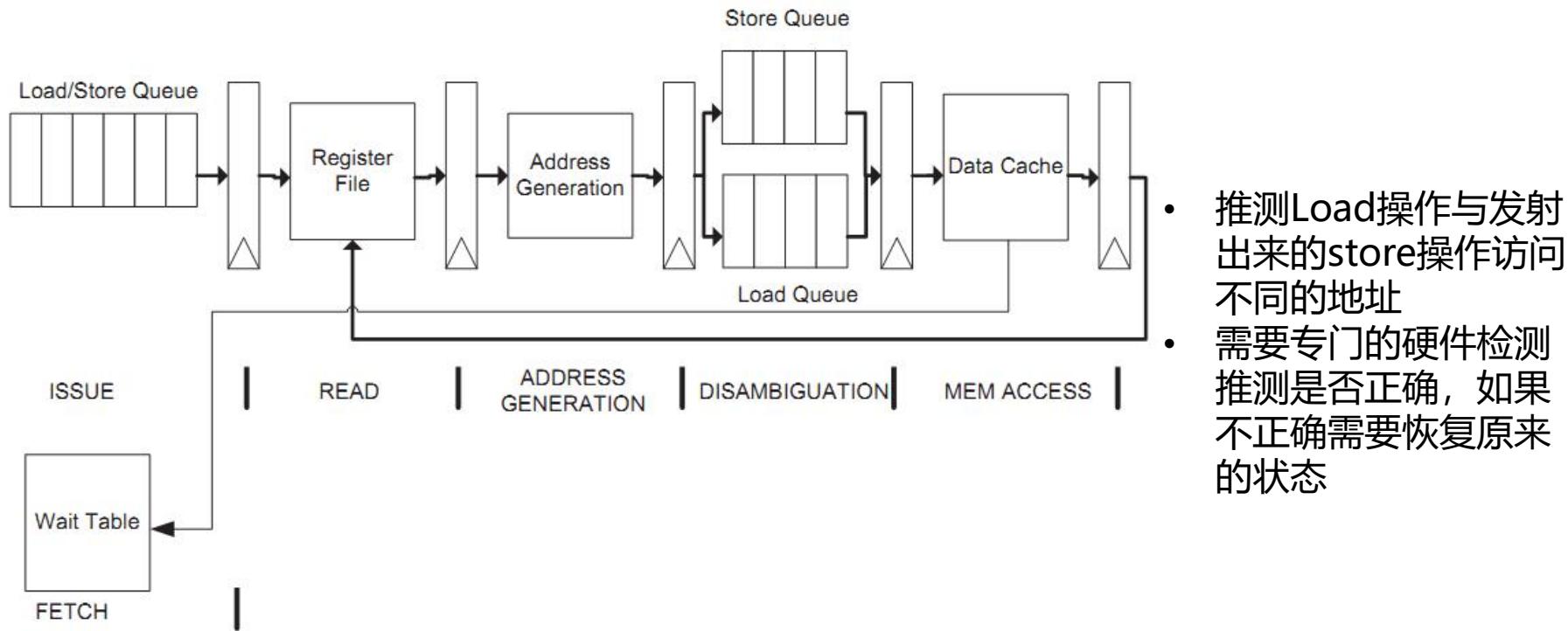


FIGURE 6.11: Schematics of the Alpha 21264 pipeline to implement a speculative memory disambiguation policy.

Load/Store Queue: 保存存储器操作的队列,直到可以计算有效地址

Load Queue: 该队列以程序序存储load操作的存储器物理地址。该队列有32项

Store Queue: 存储store操作的存储器物理地址以及要存储的值。该对列32项

Wait Table: 具有1024 项, 每项1位的表, 由存储器指令虚拟地址索引. 当我们检测到一个load操作超越了与它相关的store操作时, 该load对应项置1, 取指部件读取该信息, 将不会再提前发射出去。该表每隔16384周期复位一次, 否则该表可能所有项均为1

Alpha 21264 or Intel Core Architecture 实现了这种投机方式的存储器访问歧义消解机制



Part1：指令级并行

- **指令级并行：提高单条指令流（单线程）程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - 流水线的扩展：指令执行周期数(可能) 不同
 - 指令级并行优化：编译优化和硬件优化
 - 动态调度的“漏网之鱼”——控制相关与异常处理问题
 - 分支预测技术
 - 推断执行技术
- **多发射流水线：ILP的突破 ($CPI < 1$)**
 - 每个cycle发射多条指令执行
- **多线程技术：单线程ILP瓶颈的突破**
 - 流水线执行的指令来源于多条指令流
- **其他：“发射”指每个cycle可进入功能部件（执行部件）的指令条数**



如何使CPI < 1 ? (1/2)

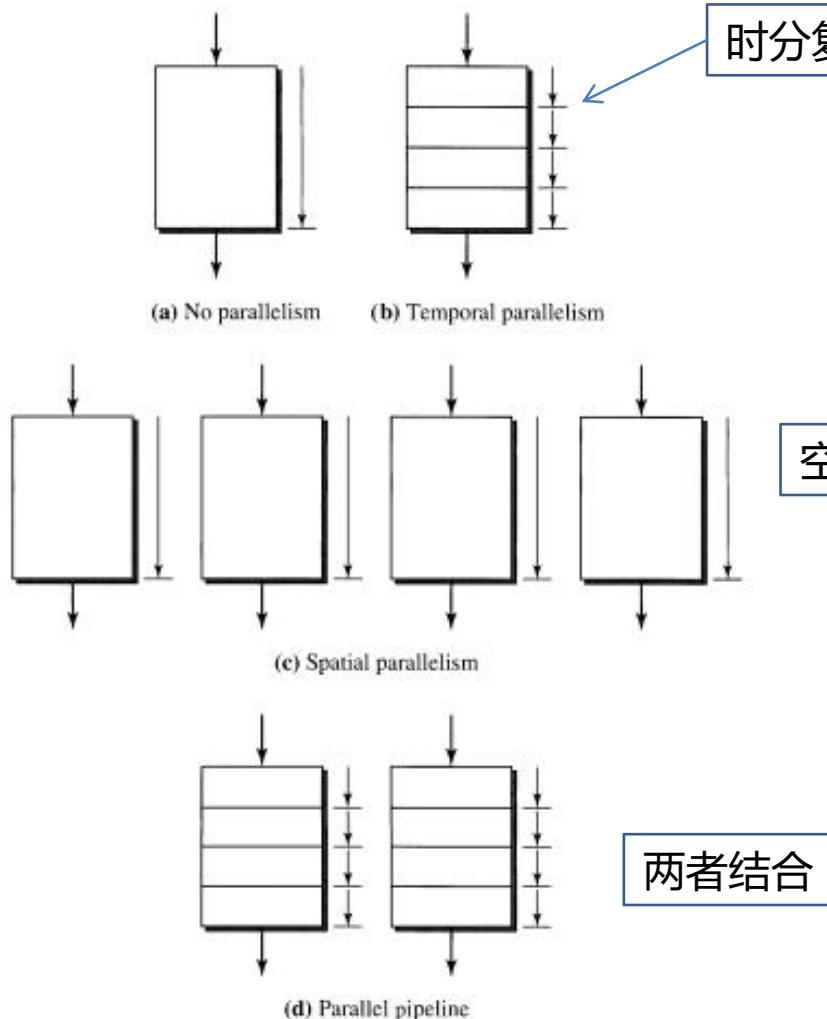
- 前面所述的各种技术主要通过减少数据相关和控制相关，使得CPI → 1
- 是否能够使CPI < 1? **多发射处理器**
- 两种基本方法: **Superscalar**、**VLIW**
- **Superscalar:**
 - 每个时钟周期所发射的指令数不定 (1 - 8条)
 - 由编译器或硬件完成调度
 - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
 - 该方法对目前通用计算是最成功的方法
- **Instructions Per Clock (IPC) vs. CPI**



如何使 CPI < 1? (2/2)

- **(Very) Long Instruction Words (V)LIW:**
 - 每个时钟周期流出的指令数（操作的数量）固定 (4-16)
 - 由编译器调度，实际上由多个单操作指令构成一个超长指令
 - 目前比较成功的应用于DSP, 多媒体应用
 - DSA (Domain Specific Architecture)
 - 1999/2000 HP和Intel达成协议共同研究VLIW
 - Intel Architecture-64 (Merced/A-64) 64-bit address
 - Style: “Explicitly Parallel Instruction Computer (EPIC)”

Machine Parallelism



时分复用，提高资源利用率

指令级并行
：时空结合

空间并行，部件并行

两者结合

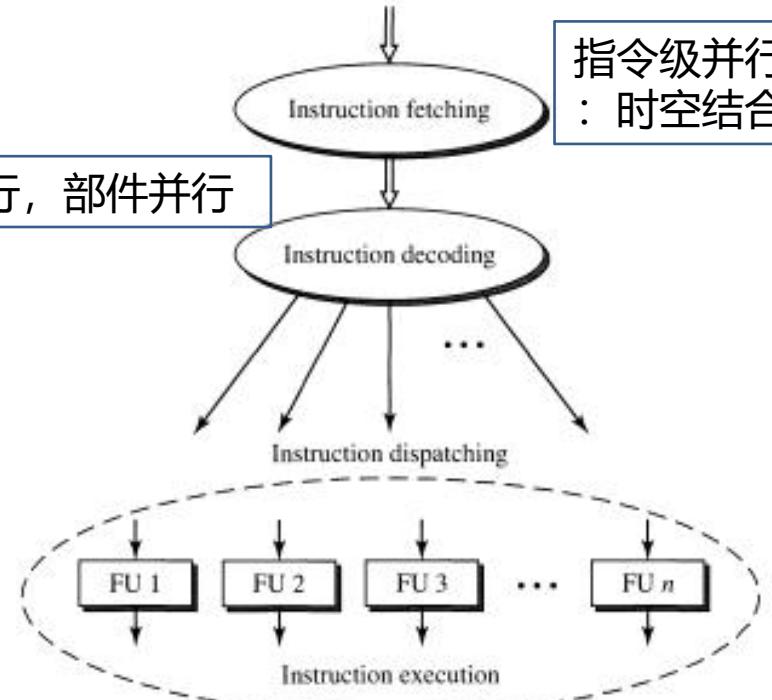


Figure 4.16

The Necessity of Instruction Dispatching in a Superscalar Pipeline.

Figure 4.2

Machine Parallelism: (a) No Parallelism (Nonpipelined); (b) Temporal Parallelism (Pipelined); (c) Spatial Parallelism (Multiple Units); (d) Combined Temporal and Spatial Parallelism.

简单的超标量流水线

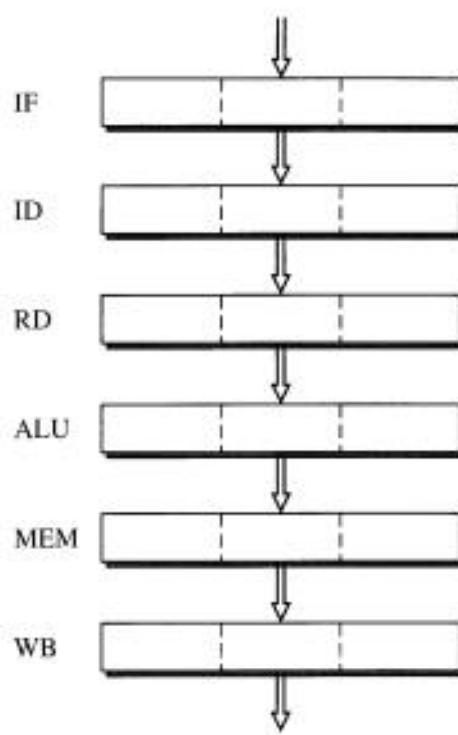


Figure 4.3

A Parallel Pipeline of Width $s=3$.

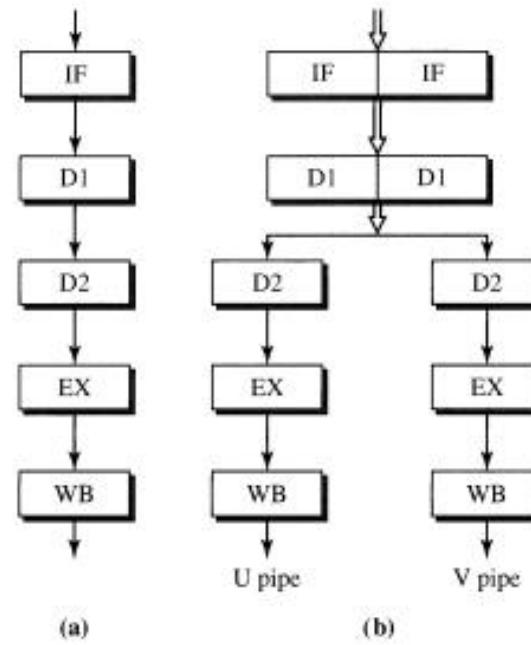


Figure 4.4

(a) The Five-Stage i486 Scalar Pipeline;
(b) The Five-Stage Pentium Parallel Pipeline
of Width $s = 2$.

具有多种执行部件

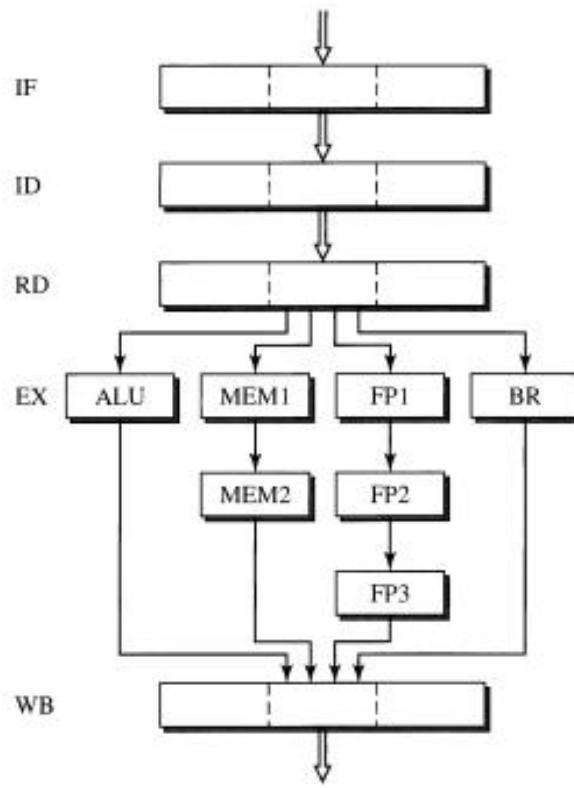


Figure 4.5

A Diversified Parallel Pipeline with Four Execution Pipes.

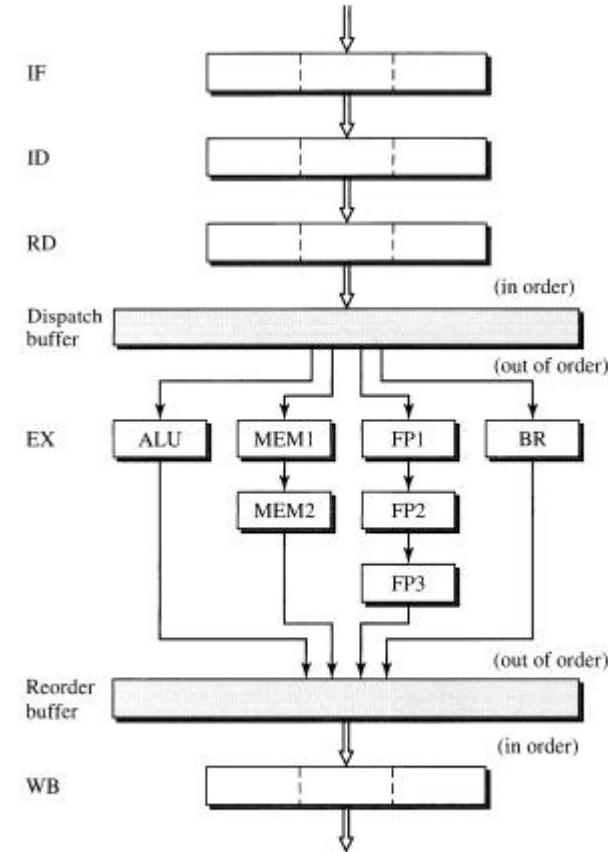


Figure 4.9

A Dynamic Pipeline of Width $s = 3$.

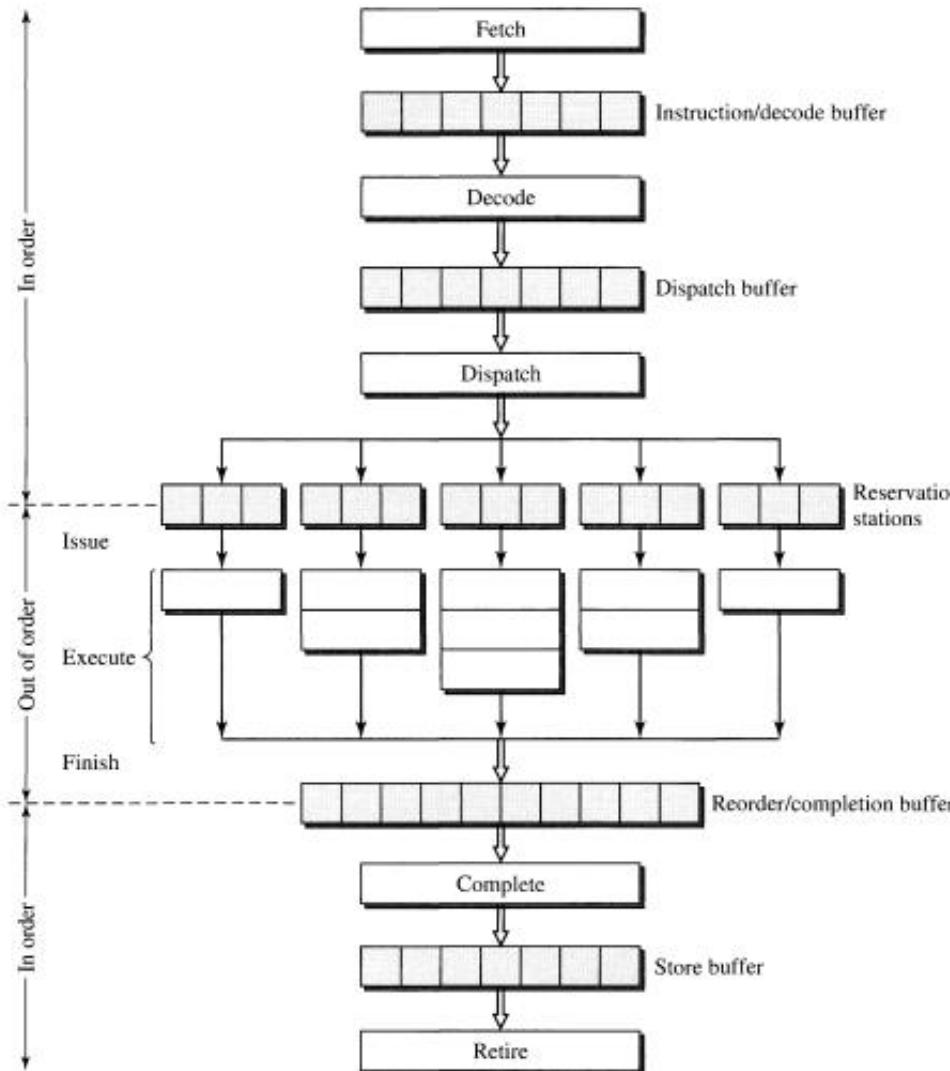
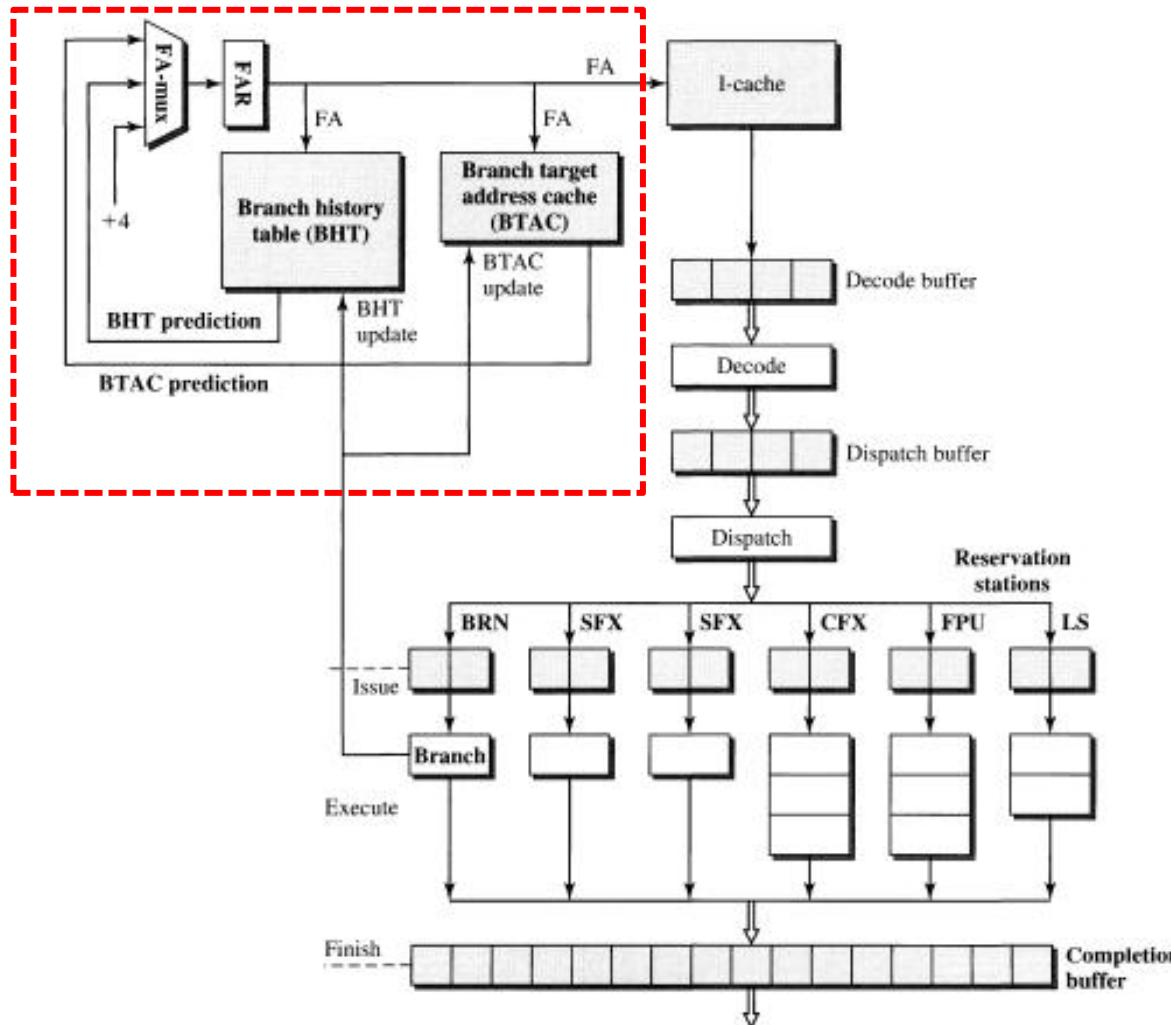


Figure 4.20

A Dynamic Pipeline with Reservation Station and Reorder Buffer.

超标量流水线中的分支预测



MODERN PROCESSOR DESIGN
Fundamentals of Superscalar Processors

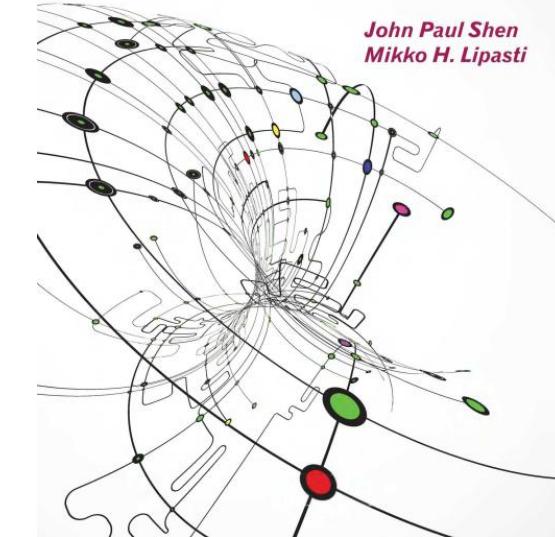


Figure 5.10

Branch Prediction in the PowerPC 604 Superscalar Microprocessor.



用于多发射处理器的五种主要方法

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.19 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.



Superscalar MIPS

- **Superscalar MIPS: 每个时钟周期发射2条指令，1条FP指令和一条其他指令**
 - 每个时钟周期取64位; 左边为Int, 右边为FP
 - 只有第一条指令发射了, 才能发射第二条
 - 需要更多的寄存器端口, 因为如果两条指令中第一条指令是对FP的load操作(通过整数部件完成), 另一条指令为浮点操作指令, 则都会有对浮点寄存器文件的操作
- 原来1 cycle load 延时在Superscalar中扩展为3条指令

Type	Pipe Stages				
Int. instruction	IF	ID	EX	MEM	WB
Fp. Instruction	IF	ID	EX	MEM	WB
Int. instruction	IF	ID	EX	MEM	WB
Fp. Instruction	IF	ID	EX	MEM	WB
Int. Instruction		IF	ID	EX	MEM
Fp. Instruction		IF	ID	EX	MEM



Review: 具有最小stalls数的循环展开优化

1 Loop:	LD	F0,0(R1)
2	LD	F6,-8(R1)
3	LD	F10,-16(R1)
4	LD	F14,-24(R1)
5	ADDD	F4,F0,F2
6	ADDD	F8,F6,F2
7	ADDD	F12,F10,F2
8	ADDD	F16,F14,F2
9	SD	0(R1),F4
10	SD	-8(R1),F8
11	SUBI	R1,R1,#32
12	SD	16(R1),F12
13	BNEZ	R1,LOOP
14	SD	8(R1),F16 ; 8-32 = -24

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

14 clock cycles, or 3.5 per iteration



采用Superscalar技术的循环展开

<i>Integer instruction</i>	<i>FP instruction</i>	<i>Clock cycle</i>
Loop: LD F0 ,0(R1)		1
LD F6,-8(R1)		2
LD F10,-16(R1)	ADDD F4,F0 ,F2	3
LD F14,-24(R1)	ADDD F8,F6,F2	4
LD F18,-32(R1)	ADDD F12,F10,F2	5
SD 0(R1), F4	ADDD F16,F14,F2	6
SD -8(R1),F8	ADDD F20,F18,F2	7
SD -16(R1),F12		8
SUBI R1,R1,#40		9
SD +16(R1),F16		10
BNEZ R1,LOOP		11
SD +8(R1),F20		12

- 循环展开5次以消除延时 (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)



多发射的问题

- 如果Integer和FP操作很容易区分组合，那么对这类程序在下列条件满足的情况下**理想CPI= 0.5**：
 - 程序中50% 为FP 操作
 - 没有任何相关
- 如果在同一时刻发射的指令越多，**译码和发射就越困难**
 - 即使是同一时刻发射2条 => 需检查2个操作码，6个寄存器描述符，检查是发射1条还是2条指令。
- **VLIW**
 - 指令字较长可以容纳较多的操作
 - 根据定义,VLIW中的所有操作是由编译时刻组合的，并且是相互无关的，也就是说：可以并行执行
 - 例如 2 个整数操作，2个浮点操作，2个存储器引用，1个分支指令
 - 每一个操作用16 到 24 位 表示 => 共 $7 \times 16 = 112$ bits 到 $7 \times 24 = 168$ bits wide
 - 需要用编译技术调度来解决分支问题



基于VLIW的循环展开

Memory reference 1	Memory reference 2	FP operation 1	FP op. 2	Int. op/branch	Clock
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16		SUBI R1,R1,#48		7
SD 16(R1),F20	SD 8(R1),F24				8
SD -0(R1),F28			BNEZ R1,LOOP		9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

注: 在VLIW中, 一条超长指令有更多的读写寄存器操作(15 vs. 6 in SS)



Example

Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

Loop: LD R2,0(R1) ; R2=array element

DADDIU R2,R2,#1 ; increment R2

SD R2,0(R1) ;store result

DADDIU R1,R1,#8 ;increment pointer

BNE R2,R3,LOOP ;branch if not last element

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. **Assume that up to two instructions of any type can commit per clock.**

Performance of Dynamic SS without speculation

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

Figure 2.20 The time of issue, execution, and writing result for a dual-issue version of our pipeline without speculation. Note that the LD following the BNE cannot start execution earlier because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch-condition evaluation allow multiple instructions to execute in the same cycle. Figure 2.21 shows this example with speculation,

Performance of Dynamic SS with speculation

Iteration number	Instructions	Issues at clock number	Executes at clock number	Readaccess at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Figure 2.21 The time of issue, execution, and writing result for a dual-issue version of our pipeline with speculation. Note that the LD following the BNE can start execution early because it is speculative.



多发射处理器受到的限制 (1/2)

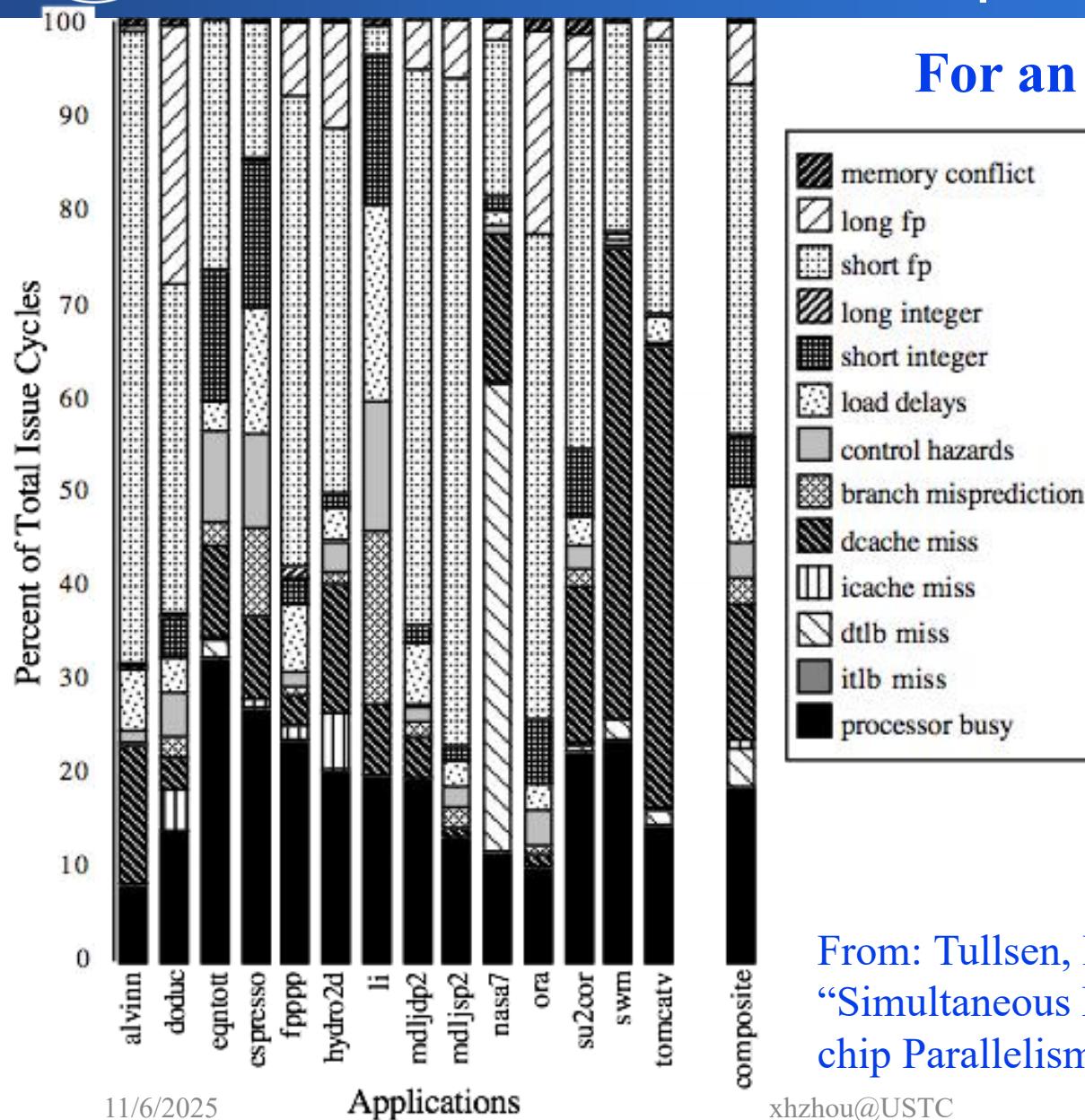
- **程序内在的ILP的限制**
 - 如果每5条指令有1条相关指令：如何保持5-路VLIW 并行？
 - 部件的操作延时：许多操作需要调度，使部件延时加大
- **多指令流出的处理器需要大量的硬件资源**
 - 需要多个功能部件来使得多个操作并行(Easy)
 - 需要更大的指令访问带宽(Easy)
 - 需要增加寄存器文件的端口数（以及通信带宽） (Hard)
 - 增加存储器的端口数（带宽） (Harder)



多发射处理器受到的限制 (2/2)

- **一些由Superscalar或VLIW实现带来的特殊问题**
 - Superscalar的译码、发射问题
 - 到底发射多少条指令?
 - VLIW 代码量问题: 循环展开 + VLIW中无用的区域
 - VLIW 互锁 => 1 个相关导致所有指令停顿
 - 静态调度, 顺序执行
 - VLIW 的二进制兼容问题
 - 微体系结构 (计算机组织结构) 不同, 二进制代码不兼容

For most apps, most execution units lie idle in an OoO superscalar



For an 8-way superscalar.



Sources of all unused issue cycles in an 8-issue superscalar processor. **Processor busy** represents the utilized issue slots; all others represent wasted issue slots.

From: Tullsen, Eggers, and Levy,
“Simultaneous Multithreading: Maximizing On-chip Parallelism”, ISCA 1995.



Summary #1/3

- **Reservations stations:** 寄存器重命名，缓冲源操作数
 - 避免寄存器成为瓶颈
 - 避免了Scoreboard中无法解决的 WAR, WAW hazards
 - 允许硬件做循环展开
 - 不限于基本块(IU先行，解决控制相关)
- **Reorder Buffer:**
 - 提供了撤销指令运行的机制
 - 指令以发射序存放在ROB中
 - 指令顺序提交
- **分支预测对提高性能是非常重要的**
 - 推断执行: 在控制相关还没有解决情况下，就开始执行
 - 推断执行利用了ROB撤销指令执行的机制
 - 处理预测错误时，撤销 推测执行的指令
 - 基于BHT的分支预测技术
 - 基于BTB的分支预测技术



Summary #2/3

- **贡献**
 - Dynamic scheduling
 - Register renaming
 - Load/store disambiguation
- **360/91 后 Pentium II; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264 使用这种技术**
- **不足之处:**
 - Too many value copy operations
 - Register File → RS → ROB → Register File
 - Too many muxes/busses (CDB)
 - Values are from everywhere to everywhere else!
 - Reservation Stations mix values(data) and tags (control)
 - Slow down max clock frequency



Summary #3/3

- **存储器访问的冲突消解**
 - 非投机方式的冲突消解
 - Total Ordering
 - Partial Ordering
 - Load指令前的store指令已经完成了地址计算，有可能乱序执行存储器load操作
 - Load Ordering, Store Ordering
 - Load指令前的存储器访问指令已经完成了地址计算，load队头的load操作有可能在store指令之前执行访存操作。
 - 投机方式的执行
 - Store Ordering
 - 假设Load操作与之前未计算出有效地址的store操作无关。
- **Superscalar and VLIW: CPI < 1 (IPC > 1)**
 - Dynamic issue vs. Static issue
 - 同一时刻发射更多的指令 => 导致更大的冲突开销



Part1：指令级并行

- **指令级并行：提高单条指令流（单线程）程序执行的性能**
 - 评估指标：CPI or IPC、CPUtime
 - 核心技术：流水线
- **单发射流水线：每个cycle发射一条指令执行**
 - 基本流水线：指令执行的周期数相同
 - 流水线的扩展：指令执行周期数(可能) 不同
 - 指令级并行优化：编译优化和硬件优化
 - 动态调度的“漏网之鱼”——控制相关与异常处理问题
 - 分支预测技术
 - 推断执行技术
- **多发射流水线：ILP的突破 ($CPI < 1$)**
 - 每个cycle发射多条指令执行
- **多线程技术：单线程ILP瓶颈的突破**
 - 流水线执行的指令来源于多条指令流
- **其他：“发射”指每个cycle可进入功能部件（执行部件）的指令条数**



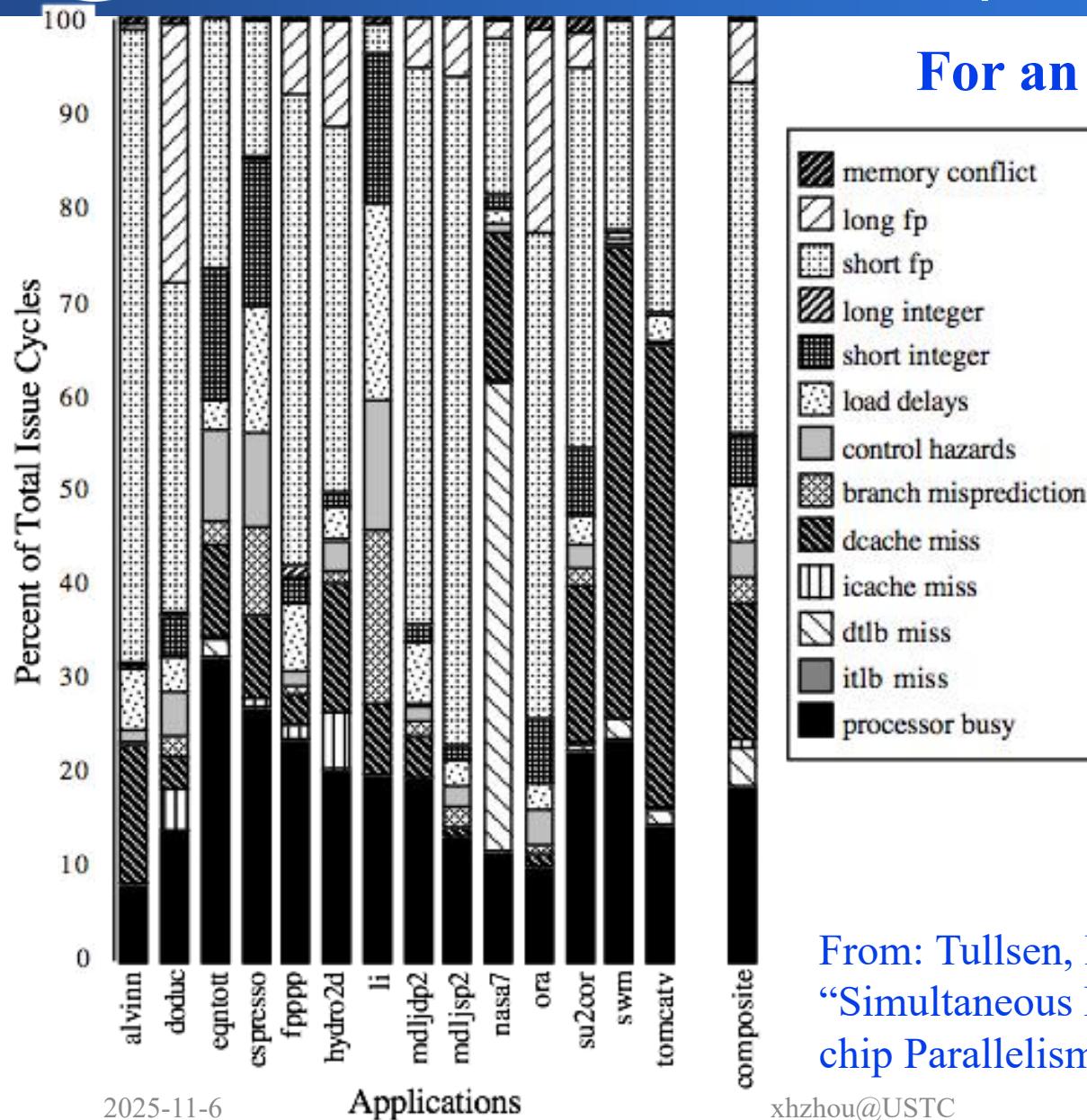
多线程技术

多发射处理器局限性

多线程处理器基本思想

多线程处理器分类

For most apps, most execution units lie idle in an OoO superscalar



For an 8-way superscalar.

Sources of all unused issue cycles in an 8-issue superscalar processor. **Processor busy** represents the utilized issue slots; all others represent wasted issue slots.

From: Tullsen, Eggers, and Levy,
“Simultaneous Multithreading: Maximizing On-chip Parallelism”, ISCA 1995.

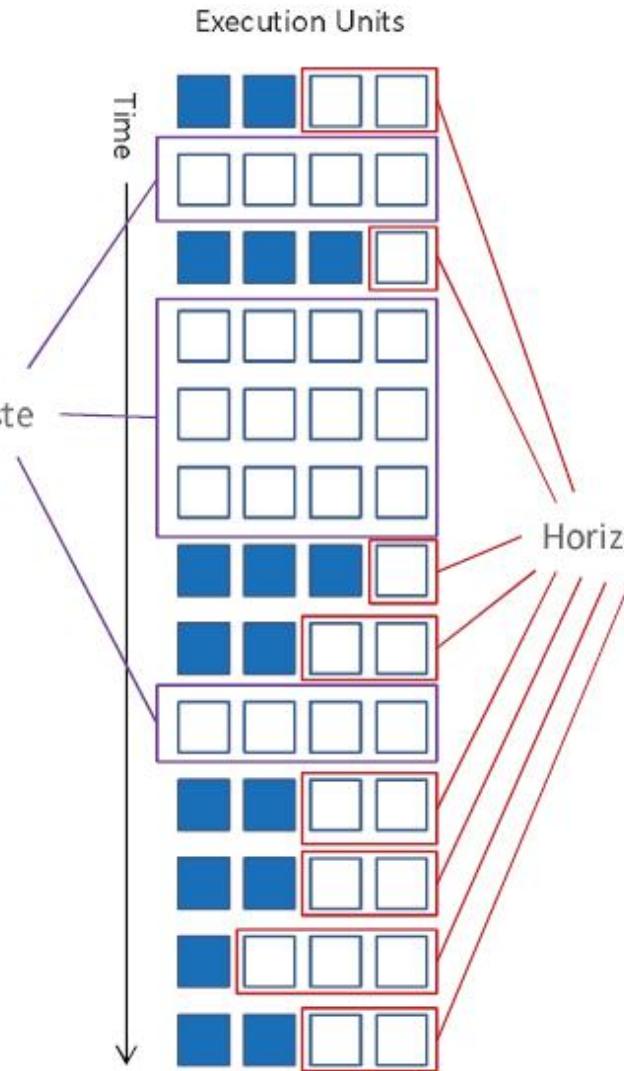
Superscalar Machine Efficiency

*Completely idle cycle
(vertical waste)*

时间维度上的浪费

*Partially filled cycle,
i.e., $IPC < 4$
(horizontal waste)*

空间维度上的浪费





多线程技术

多发射处理器局限性

多线程处理器基本思想

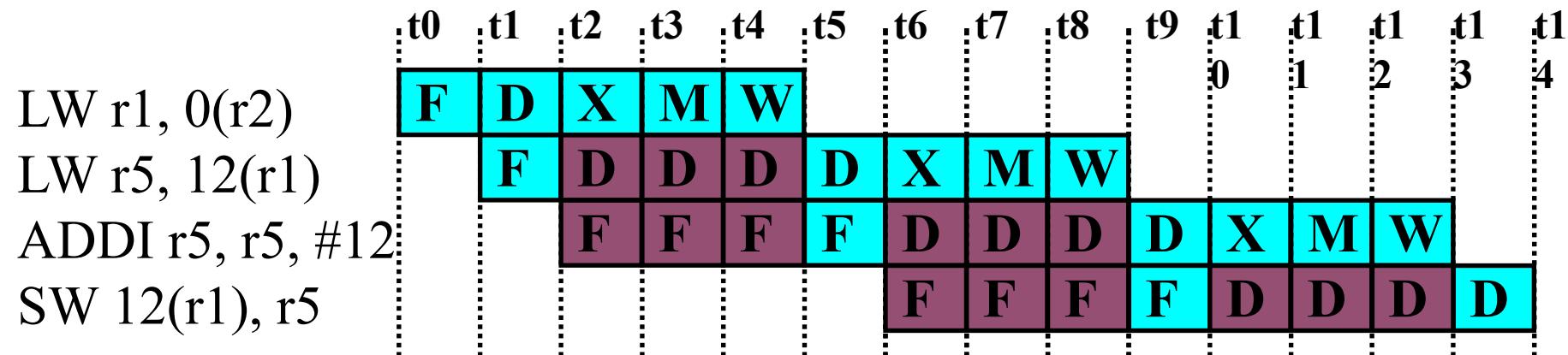
多线程处理器分类



Multithreading

- **背景：**从单线程程序挖掘指令集并行越来越困难，是否有其他途径？→ Multithreading
- **前提：**许多工作任务可以使用线程级并行来完成
 - 线程级并行来源于多道程序设计
 - 线程级并行的基础是多线程应用，即一个任务可以用多个线程并行来加速
- **基本思想：**多线程应用可以用线程级并行来提高**单个处理器的利用率**
 - 针对单个处理器：**多个线程以时分复用方式共享单个处理器的功能单元，填补硬件吞吐率与软件单线程ILP之间的差异。**

Pipeline Hazards

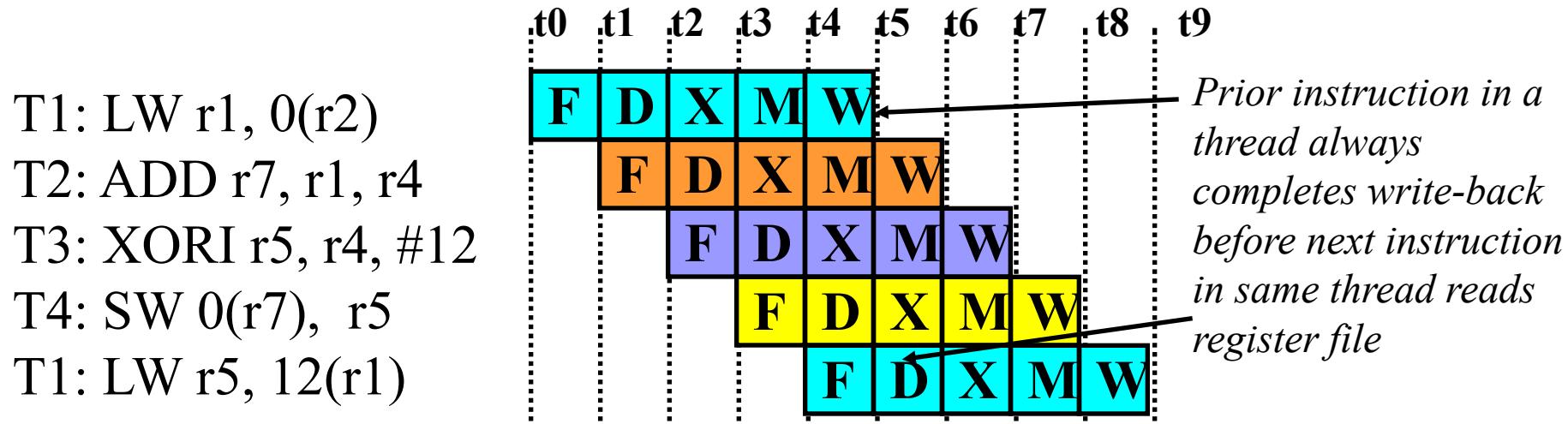


- 每条指令与前一条指令存在RAW相关
- 如何处理相关?
 - 使用interlock机制(slow)
 - 或定向路径

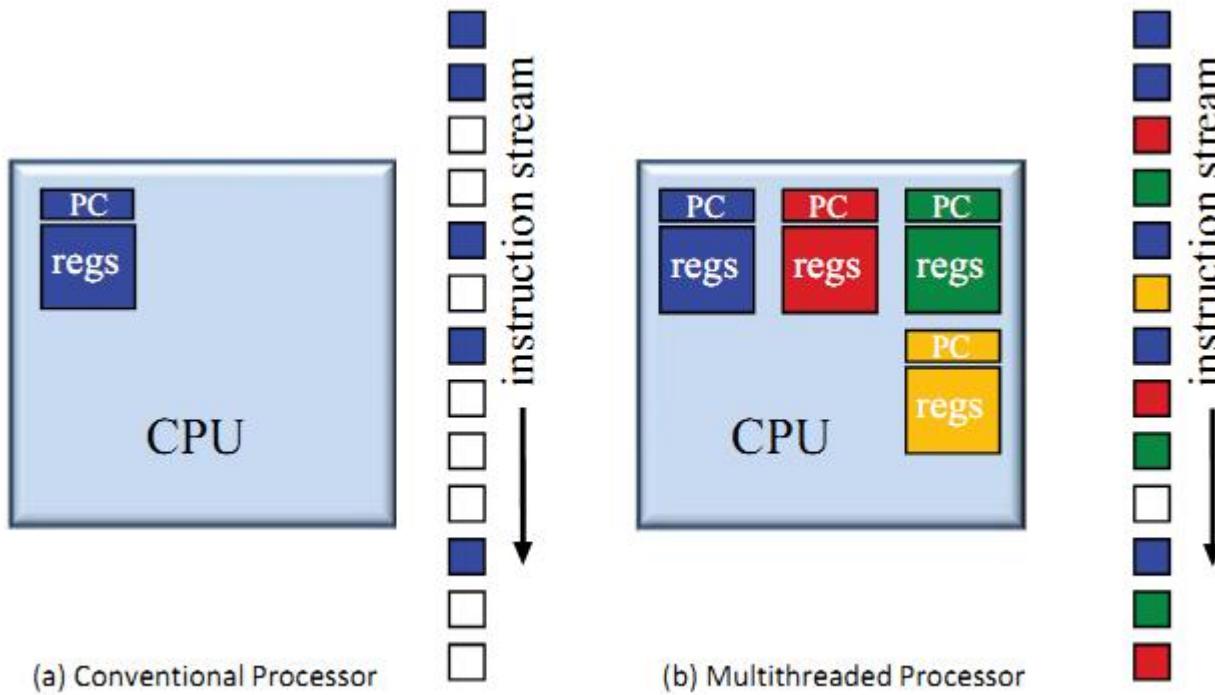
Multithreading

- 如何保证流水线中指令间无数据依赖关系?
- 一种办法: 在相同的流水线中交叉执行来自不同线程的指令

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe



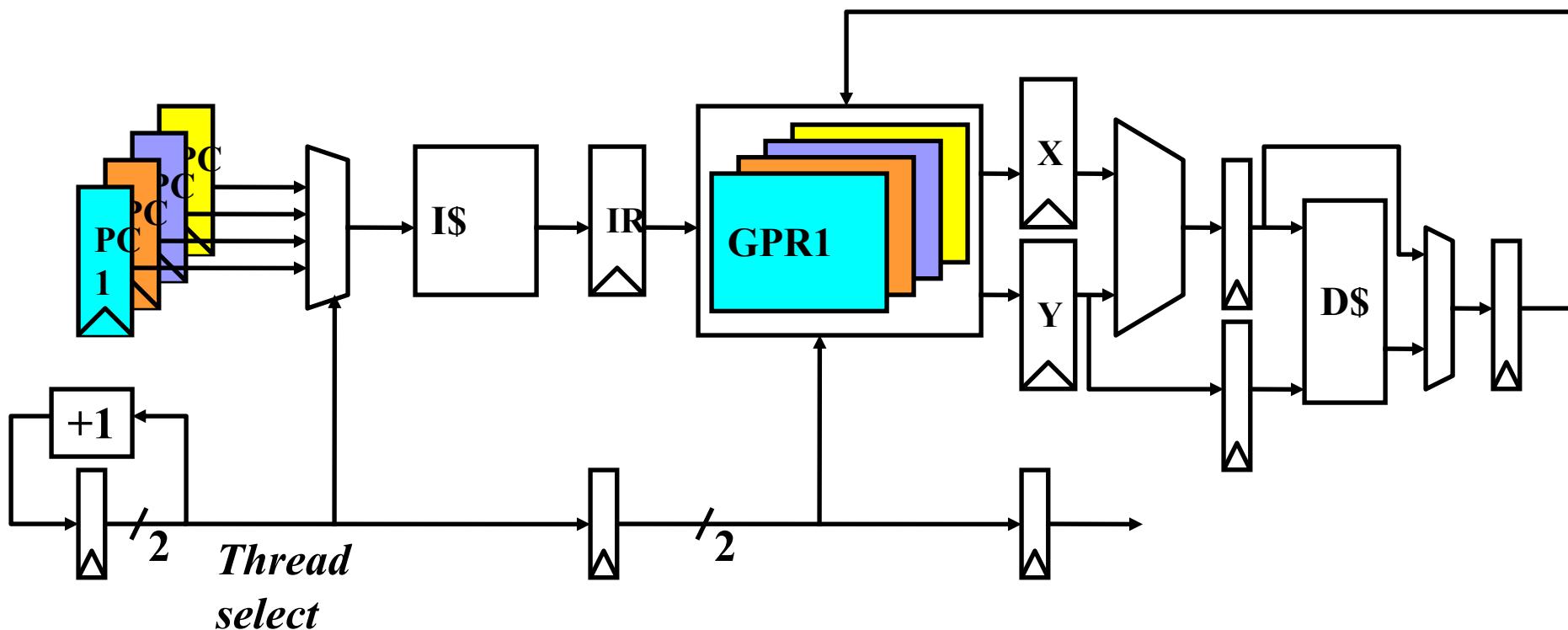
单线程处理器与多线程处理器



(传统) 单线程处理器与多线程处理器

Simple Multithreaded Pipeline

- 必须传递线程选择信号以保证各流水段读写的正确性
- 从软件（包括OS）的角度看 好像存在多个CPU（针对每个线程，CPU似乎运行的慢一些）





Multithreading Costs

- **硬件上下文切换开销：保存和恢复相关状态信息**
- **硬件上下文切换需要为每个线程保存和恢复的信息**
 - 用户态信息 (user state)：包括PC、GPRs
 - 系统态信息 (system state)：
 - 虚拟存储的页表基地址寄存器 (Virtual-memory page-table-base register)
 - 异常处理寄存器 (Exception-handling registers)
- **其他开销：**
 - 需要处理由于线程竞争导致的Cache/TLB冲突 或 需要更大的cache/TLB 容量
 - 调度器 (OS 或硬件) 管理线程

Thread Scheduling Policies

- **硬件固定交叉模式 (CDC 6600 PPUs, 1964) (静态)**
 - 针对N个线程，每个线程每隔N个周期执行一条指令
 - 如果流水线的某一时隙(slot)其对应线程未就绪，插入pipeline bubble
- **硬件控制的线程调度 (HEP, 1982) (动态)**
 - 硬件跟踪哪些线程处于ready状态
 - 根据优先级方案选择线程执行
- **软件控制的交叉模式 (TI ASC PPUs, 1971)**
 - PPU- Peripheral processing Unit
 - OS (软件) 显式地控制线程交叉
 - 多个线程分时共享功能部件
 - 例如：





多线程技术

多发射处理器局限性

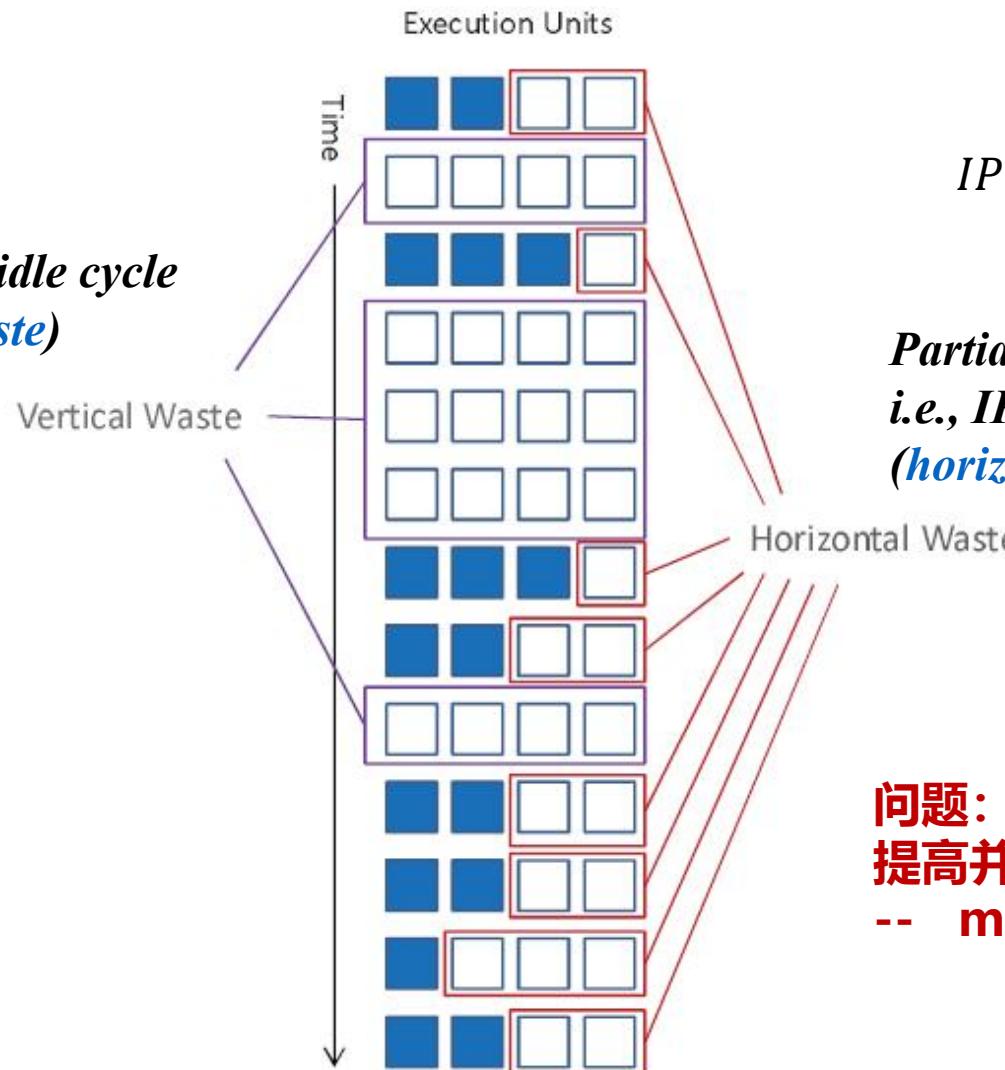
多线程处理器基本思想

多线程处理器模型

1. Chip Multiprocessing
2. Coarse-Grain Multithreading
3. Fine-Grain Multithreading
4. Simultaneous Multithreading

Superscalar Machine Efficiency

*Completely idle cycle
(vertical waste)*



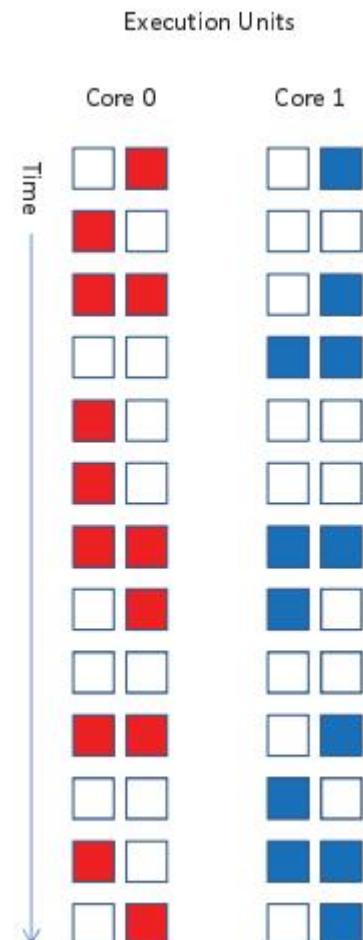
$$IPC = \frac{1}{CPI}$$

*Partially filled cycle,
i.e., $IPC < 4$
(horizontal waste)*

问题：如何充分利用资源，
提高并行度？
-- multithreading

Chip Multiprocessing (CMP)

- **如何充分利用资源，提高并行度？片上集成多个处理器核**
- **分成多个处理器后的效果？**
 - 同一时间周期不同核可以运行不同线程的指令
 - 单个处理器核同一时间周期无法执行不同线程指令
 - 从单个核看，没有减少水平浪费和垂直浪费。
 - 由于发射宽度在核间进行了静态分配，导致水平和垂直方向浪费减少
- **例如：2核2发射的CMP**
 - 发射宽度为4，即同时可以发射4条指令
 - 当1个线程stall，那么垂直浪费最多为2条指令



Vertical(Coarse-Grain) Multithreading

- **如何充分利用资源，提高并行度？粗粒度多线程**

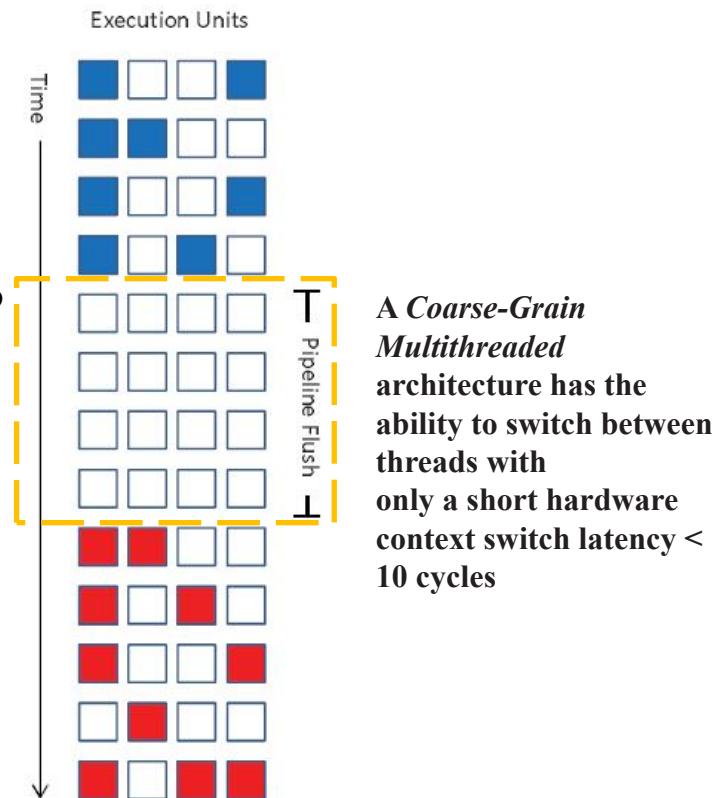
- 基本思想源于分时系统
- 当线程运行时存在较长延时时切换到另一线程
 - 例如：Cache失效时
 - 等待同步结束
- 同一线程指令间的较短延时不切换

- **如果基于粗粒度的线程交叉运行模式，结果怎样？**

- 减少了垂直方向的（时间）浪费，但仍然存在垂直方向的浪费（流水线Flush）
- 减少水平方向的（空间）浪费？

- **最早的CGMT系统：**

- DYSEAC [Leiner, 1954]
 - 美国国家标准局为美军通信兵团设计建造
- TX-2 [Forgie, 1957]
 - MIT Lincoln Laboratory.



线程间切换速度快！<10 cycles

Coarse-Grain Multithreading参考设计

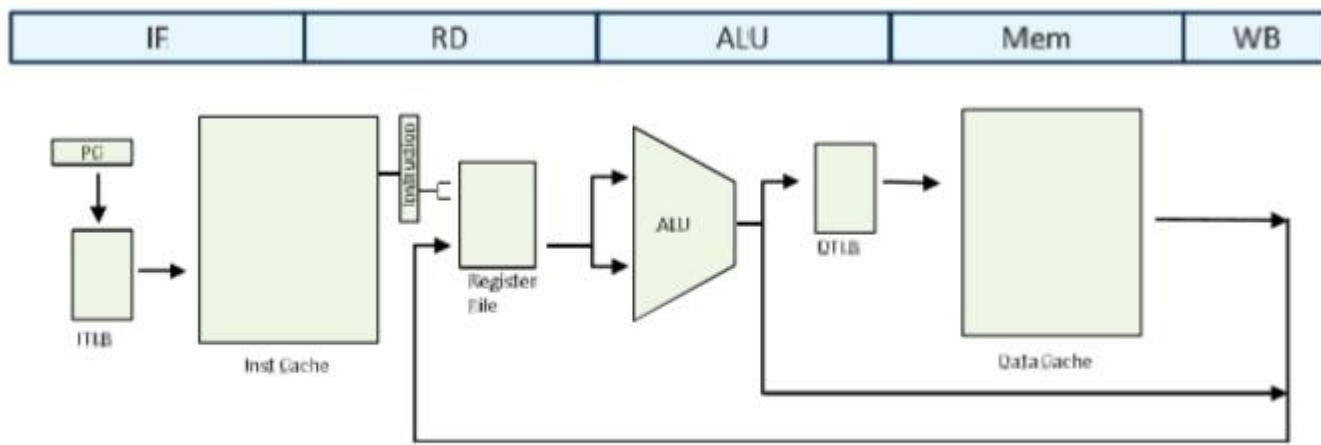


Figure 3.1: The MIPS R3000 Pipeline. This pipeline does not support multithreading.

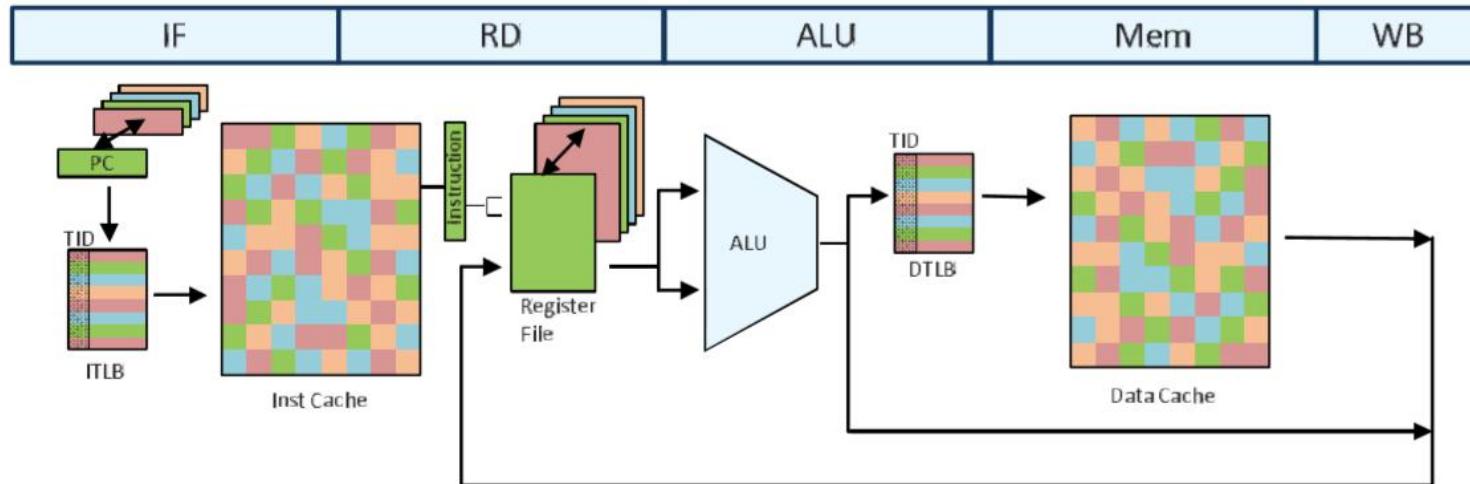


Figure 3.2: The MIPS R3000 Pipeline with support for Coarse-Grain Multithreading.



CGMT 上下文切换 (线程切换)

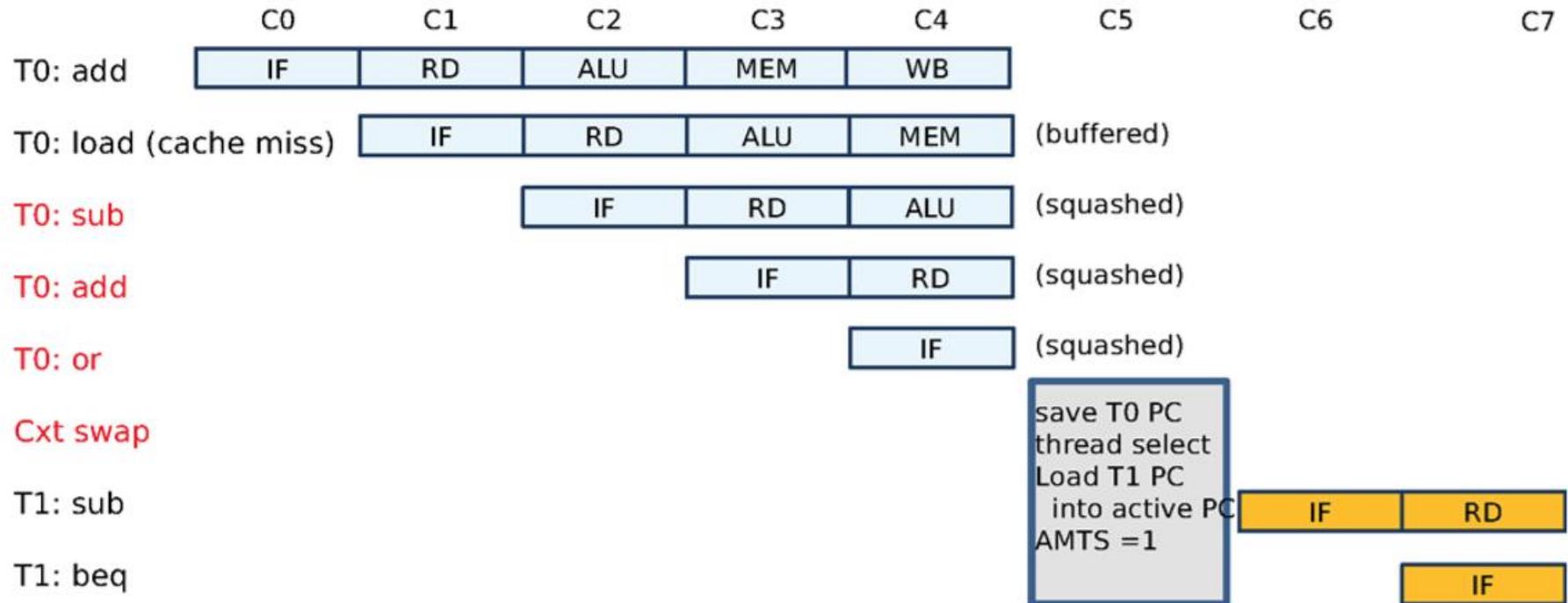


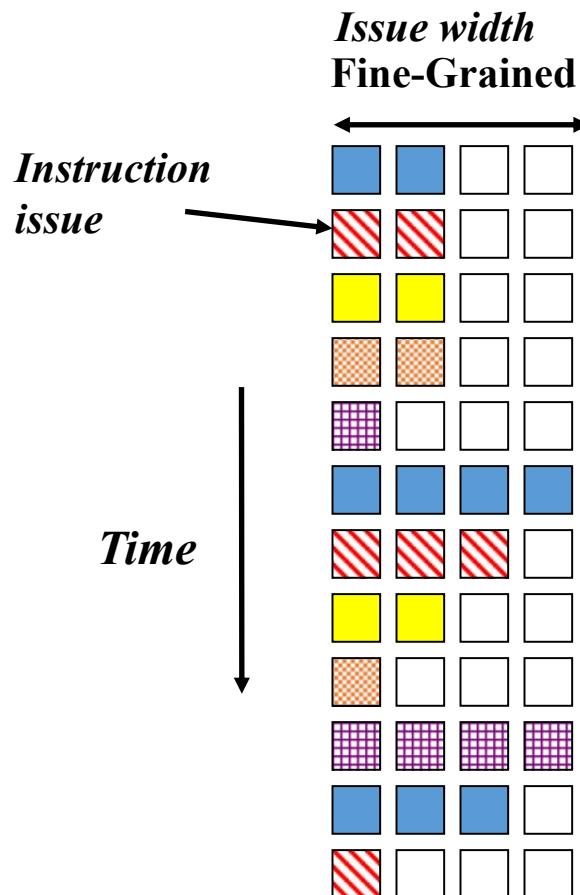
Figure 3.3: The operation of a CGMT context swap in our reference design.



CGMT优劣势

- **与单线程设计相比：CGMT的主要优势是它可执行快速的（硬件完成）上下文切换，以隐藏内存访问延迟**
 - 其他多线程模型（FGMT）和同时多线程（SMT）也具有该优势，并且有可能更彻底地隐藏延迟
 - 实现这一收益代价是额外的存储—需要每个线程能够运行的完整硬件上下文
- **与其他多线程模型相比：CGMT优势在于它与现有的单线程处理器的差异最小，在原有单线程处理器上改进的代价较小**
- **CGMT负面影响可能会降低这些好处，在极端情况下会导致性能下降**
 - 在CGMT中，唯一的负面影响来自对缓存、分支预测器和TLB表项的竞争，可能会导致命中率，预测准确性的降低
 - 其他多线程模型也存在相同的问题

Fine-Grain Multithreading



- 如何充分利用资源，提高并行度？
细粒度多线程
 - 多个线程的指令交叉执行
- 如果基于细粒度的线程交叉运行模式，结果怎样？
 - 减少垂直方向的浪费
 - 当线程数足够多时，可消除垂直方向的浪费
 - 仍然存在水平方向的浪费
- **CDC 6600：第一次实现FGMT**
 - ALU部件100ns vs. 存储器访问1000ns

Fine-Grain Multithreading参考设计

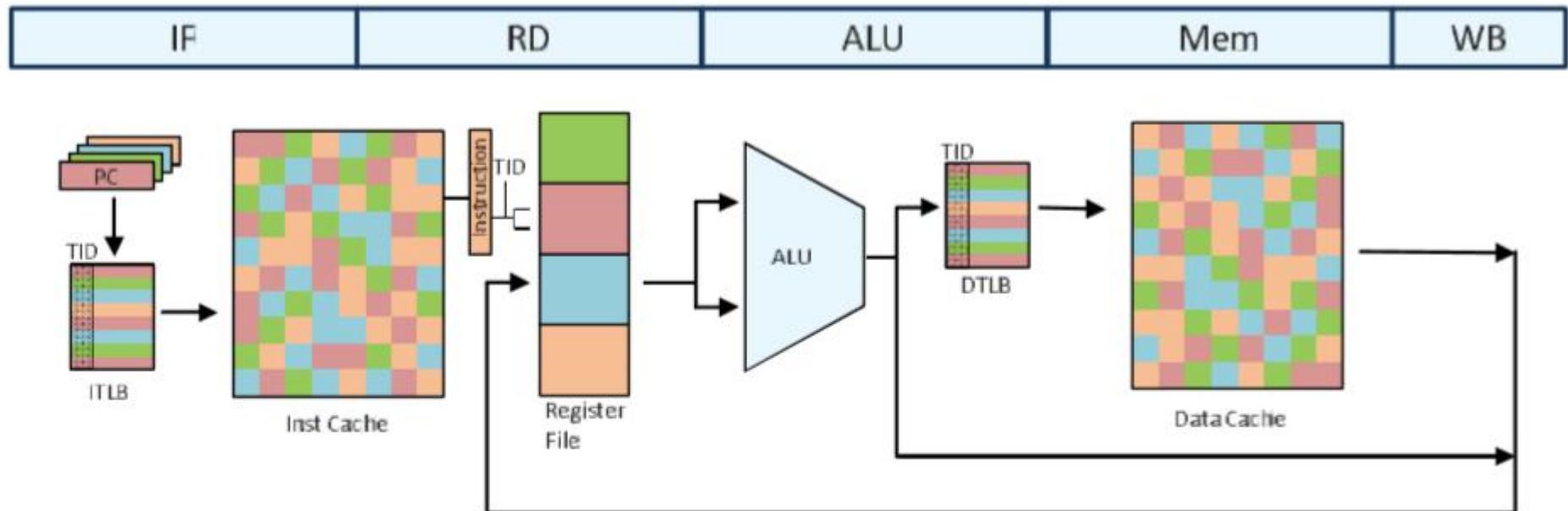


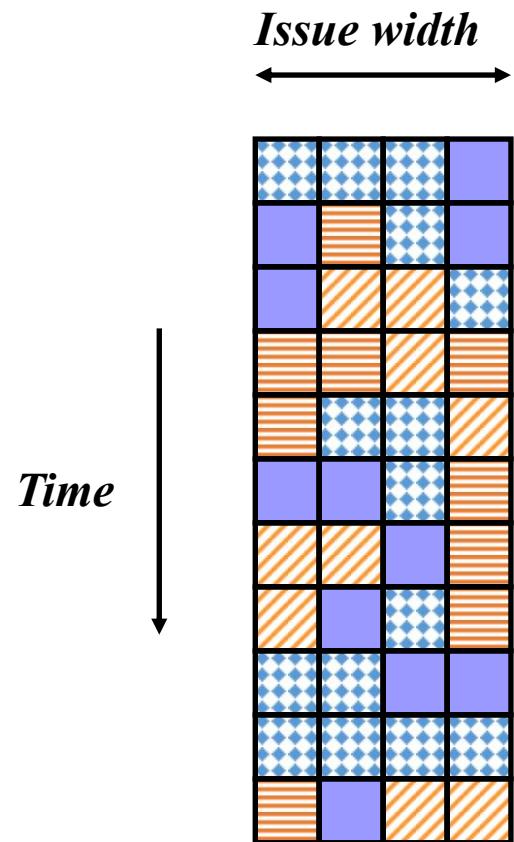
Figure 4.1: The MIPS R3000 Pipeline with support for Fine-Grain Multithreading. Although our target design would support 8 threads to maximize throughput, we show a 4-thread implementation for simplicity.

要点：硬件实现更快的上下文切换

Ideal Superscalar Multithreading

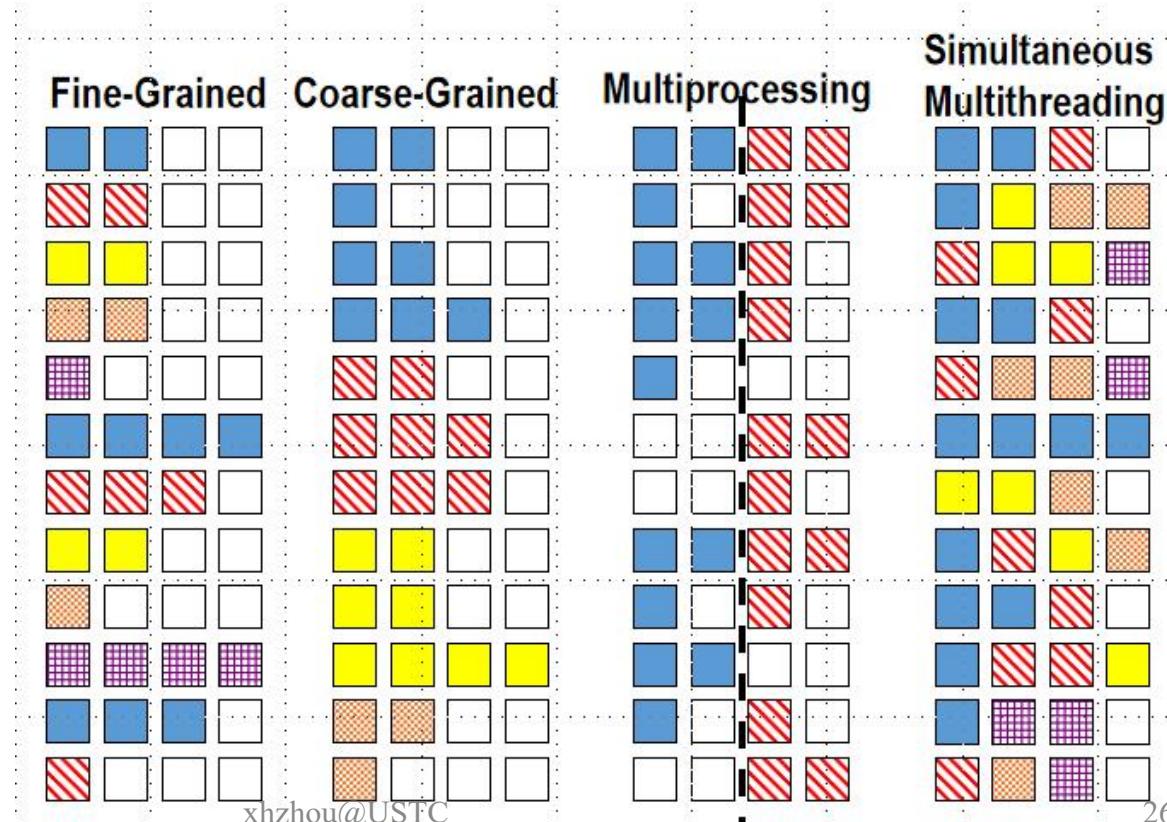
[Tullsen, Eggers, Levy, UW, 1995]

- 如何充分利用资源，提高并行度？
- 采用多线程交叉模式使用多个issue slots
 - 允许在水平方向同时处理来自多个线程的指令
 - Simultaneous Multithreading (SMT)



Simultaneous Multithreading (SMT) for OoO Superscalars

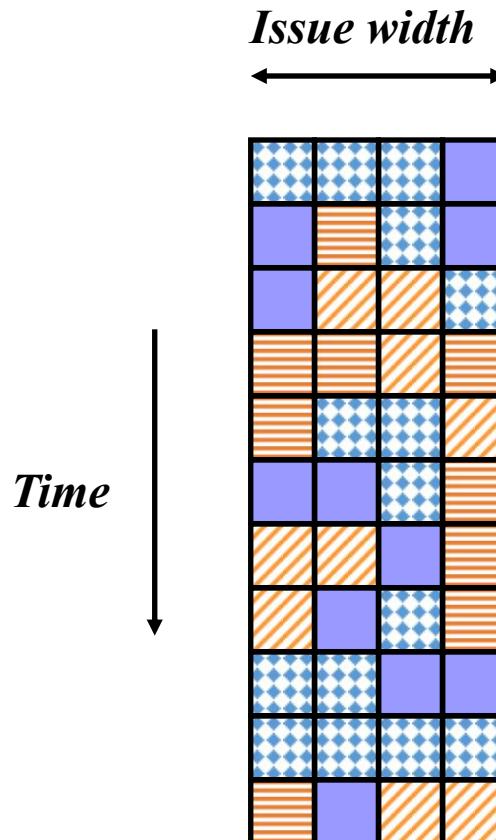
- CMP、CGMT、FGMT都是“vertical”多线程：多条流水线同一时钟周期只处理一个线程的指令
- SMT 使用OoOSuperscalar细粒度控制技术在同一时钟周期处理多个线程的指令，以更好的利用系统资源
 - Alpha AXP 21464
 - Intel Pentium 4, Intel Nehalem i7
 - IBM Power5



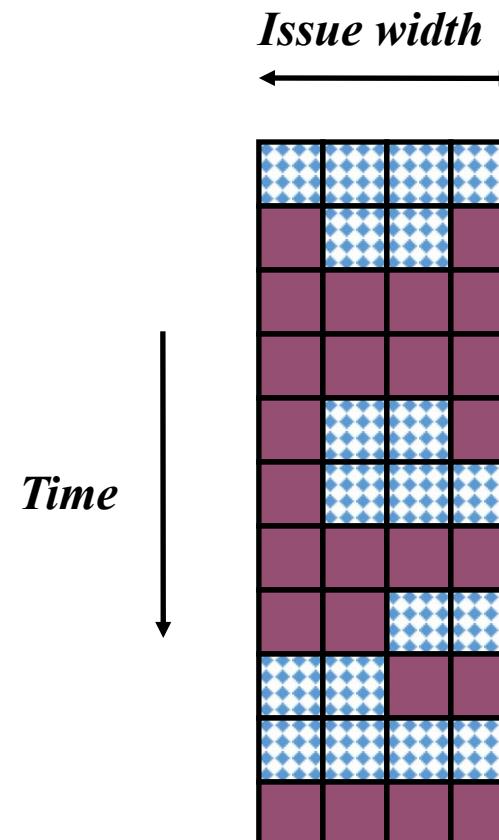
SMT adaptation to parallelism type

对于具有高线程级别并行性(TLP)的SW区域，所有线程共享整个发射宽度

对于具有低线程级并行度(TLP)的SW区域，整个发射宽度可用于指令级并行度(ILP)



同时有4个线程并行执行



同时有2个线程并行执行



O-o-O Simultaneous Multithreading

[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

- 考虑多上下文以及取指引擎可以从多个线程取指令，并可同时发射
- 使用OoO (Out-of-Order) superscalar处理器的发射宽度，从发射队列中选择指令发射，这些指令可来源于多个线程
- OoO 指令窗口已具备从多个线程调度指令的绝大多数电路
- 任何单线程程序可以使用整个系统资源



仅支持单线程的MIPS R1000

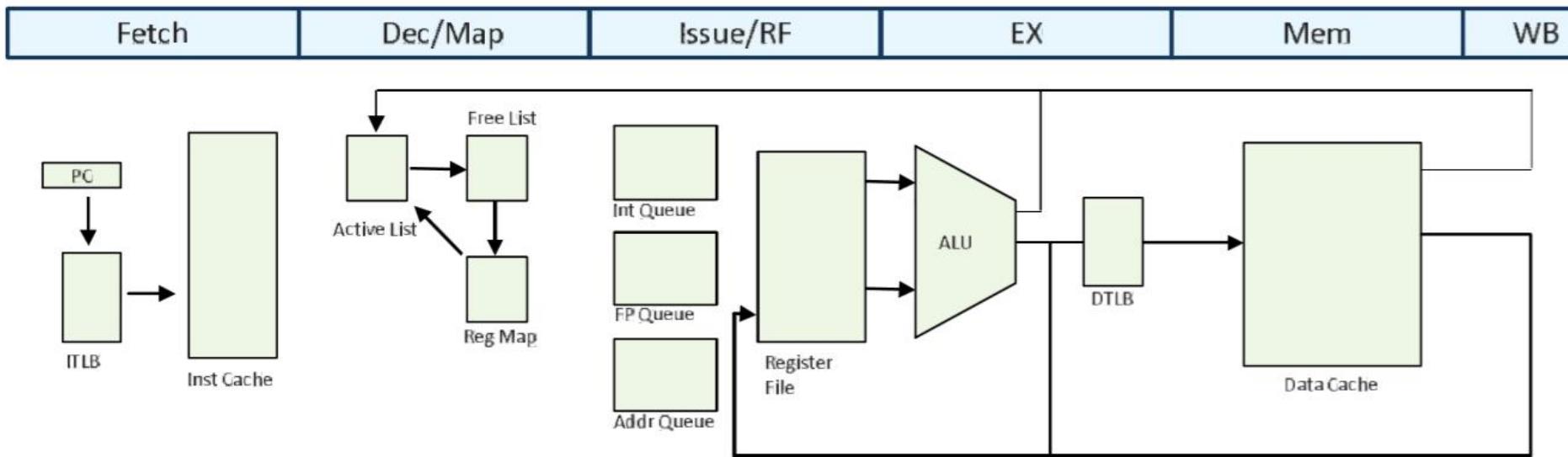


Figure 5.1: The pipeline of the MIPS R10000 processor, as seen by a load instruction.

支持同时多线程的MIPS R10000参考设计

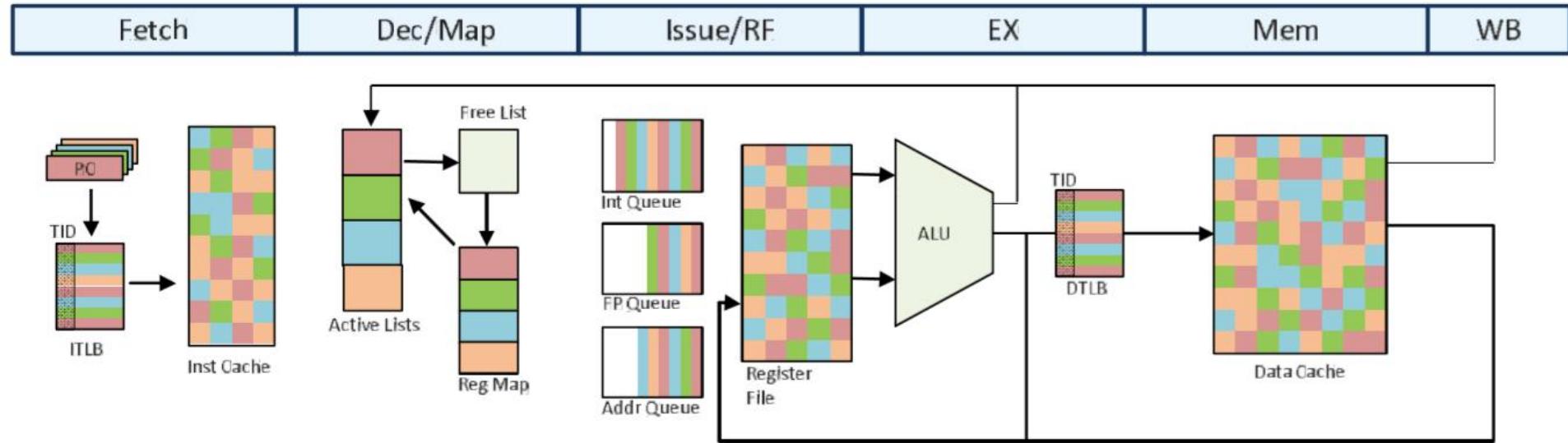


Figure 5.2: The MIPS R10000 pipeline with support added for simultaneous multithreading.

Performance

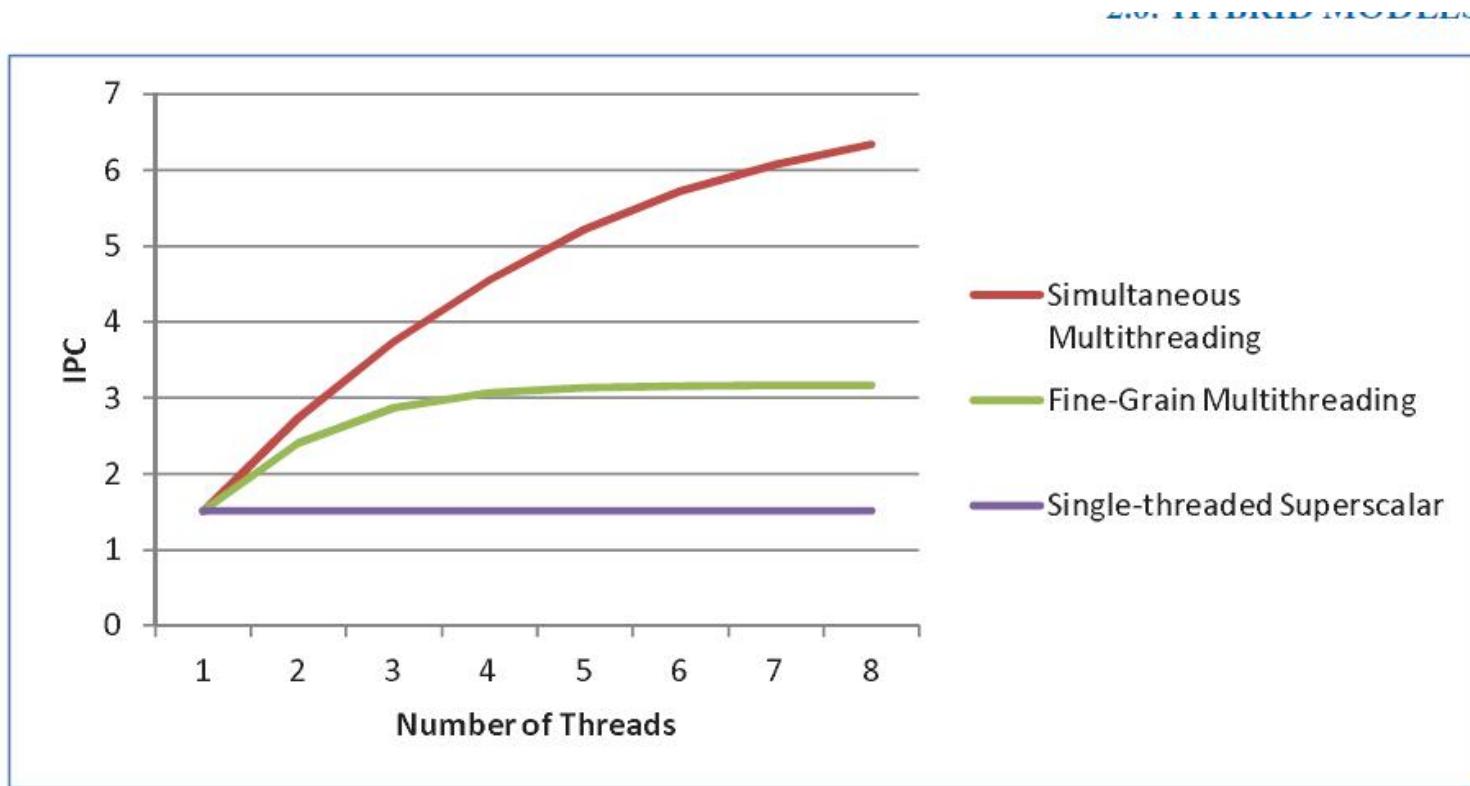
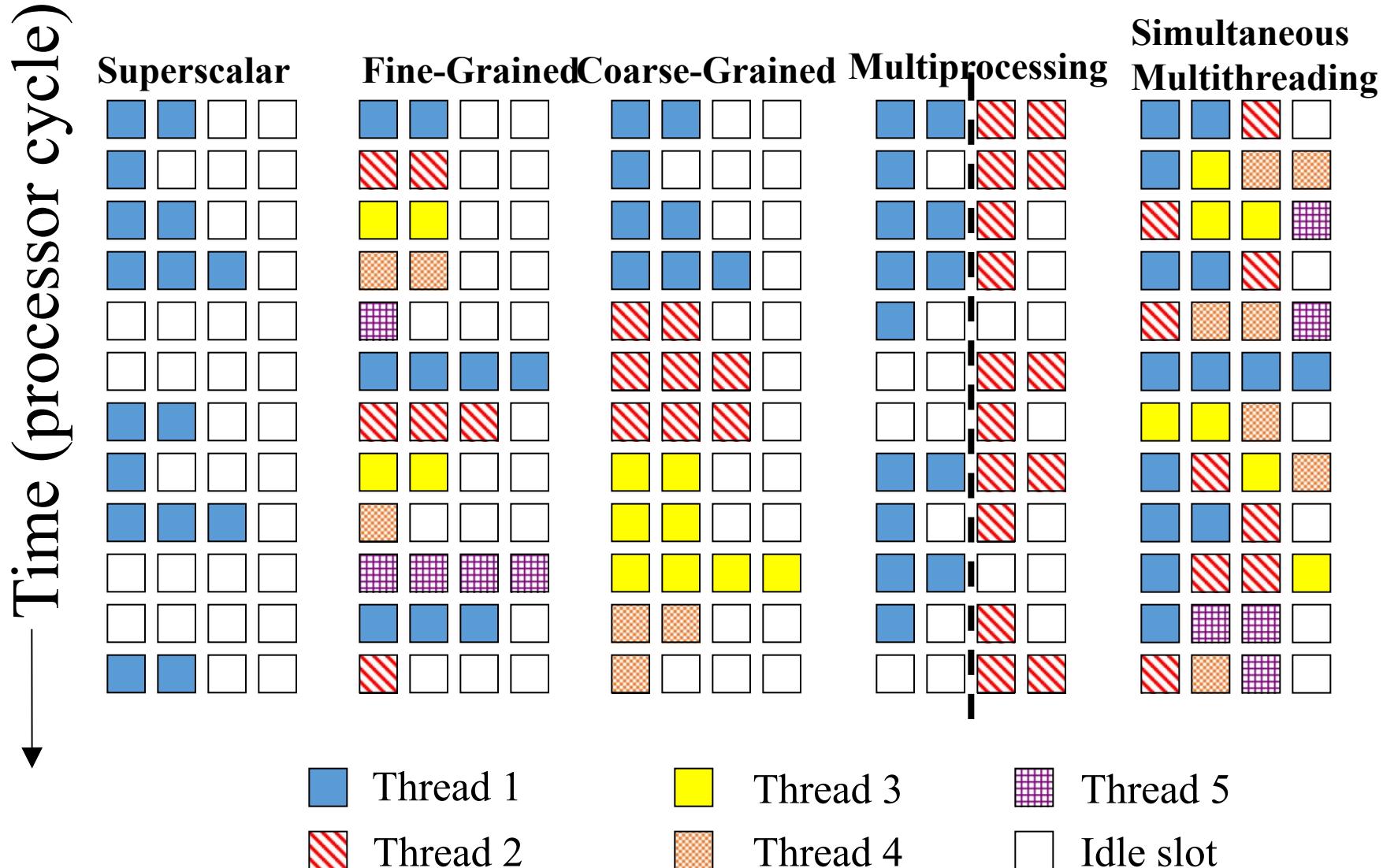


Figure 2.7: The performance of fine-grain and simultaneous multithreading models on a wide superscalar processor, as the number of threads increases. Reprinted from Tullsen et al. [1995].

Summary: Multithreaded Categories



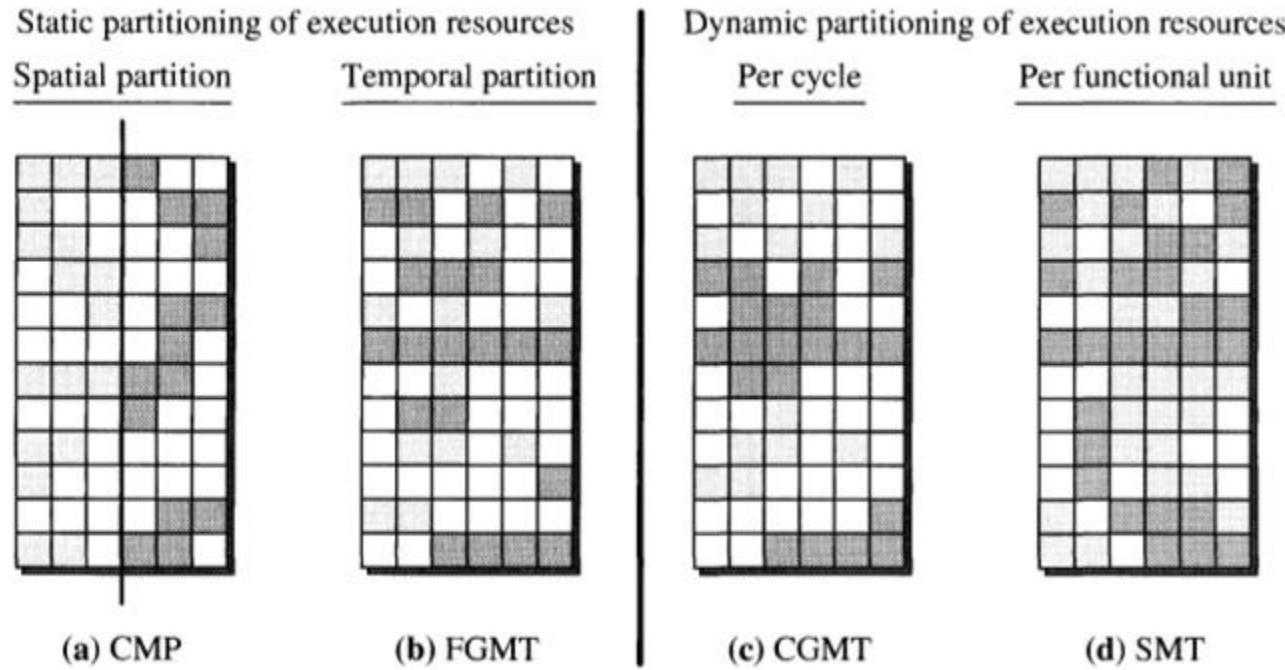


各种多线程技术资源使用情况 (1/2)

多线程技术	线程间共享的资源	上下文切换机制
None	共享：所有资源	操作系统负责切换
FGMT	独占：寄存器文件，控制逻辑/状态	每个Cycle切换
CGMT	独占：I-fetch buffer, 寄存器文件，控制逻辑/状态	流水线较长stall时切换
SMT	独占：I-fetch buffer, return address stack, 寄存器文件，控制逻辑/状态, reorder buffer, store queue等	所有上下文处于活跃状态，没有切换问题
CMP	共享：二级Cache, 系统互联	所有上下文处于活跃状态，没有切换问题

John Paul Shen, Mikko H. Lipasti; Modern Processor Design: Fundamentals of Superscalar Processors ; 2013, Waveland Press

各种多线程技术资源使用情况 (2/2)



(a) CMP	空间维度划分发射宽度；静态划分执行所需资源
(b) FGMT	时间维度划分发射宽度；静态划分执行所需资源
(c) CGMT	长延时时切换线程；每个cycle动态划分执行所需的资源
(d) SMT	每个cycle发射的指令可来自不同线程，动态划分执行所需的资源



Acknowledgements

- **These slides contain material developed and copyright by:**
 - John Kubitowicz (UCB)
 - Krste Asanovic (UCB)
 - David Patterson (UCB)
 - Chenxi Zhang (Tongji)
- **UCB material derived from course CS152, CS252, CS61C**
- **KFUPM material derived from course COE501, COE502**