Out-of-Order Execution

Out-of-Order Execution

Revisit Data Dependency

有三种数据依赖：RAW, WAW, WAR

其中 RAW 通过乱序执行和 scoreboarding 解决，WAW 和 WAR 通过寄存器重命名解决

对于一个按序的流水线，如果指令因为 RAW stall 在流水线，后续指令的分发会被 stall，这样的 stall 是没有必要的

Out-of-Order Execution

基本思路就是按照数据流图的顺序执行指令，当指令的操作数未就绪可以先停留在 reservation station，而无关指令可以先执行，直至其操作数准备好

要注意乱序执行和超标量没有必然的联系

Scoreboard

基本思路：指令可以乱序执行，只要其资源就绪且没有数据依赖

根据 CDC 6600 的算法命名

指令执行可以分为四个阶段

- issue
- dispatch
- execute
- write result

对于 RAW 依赖，指令等待源操作数就绪后再分发执行

对于 WAW 和结构冒险，指令按序分发，如果有结构冒险则 stall，没有 WAW，因为一次只有一个指令试图写寄存器

对于 WAR，指令等待其之前的指令读寄存器，直到寄存器可用后再写

Stages of scoreboard

issue

取指令，以及一部分解码

- 检查结构冒险：是否该指令需要的 function unit 均不可用
- 检查 WAW：是否有活跃指令要写相同的寄存器

如果存在上述情况，stall，直到没有冒险，然后继续 issue，更新 scoreboard

read operand

检查 RAW 冒险：是否所有的源操作数均可用

- 如果否，将指令存储在 pre-execution buffer 中
- 如果是，由 scoreboard 告知 function unit 读寄存器并开始执行

execution

开始执行，完成后通知 scoreboard

write result

检查 WAR 冒险：是否有之前的指令要读目的寄存器且还未读

如果检测到 WAR，将指令 stall，结果仍存在 function unit 中，否则将结果写入目的寄存器

Scoreboard Implementation

function unit 的状态有

- busy：指示是否被占用
- Op：执行的操作
- F_i ： 目的寄存器
- F_j, F_k ： 源寄存器
- Q_j, Q_k ： 产生源寄存器值的 function unit
- R_j, R_k ： 指示 F_j, F_k 是否已就绪并未读，如果读取了则设为 NO

register result 状态指示哪个 function unit 要写该寄存器

Summary

scoreboard 算法按序 issue 指令，乱序分发并执行

- 优点
  - 基于数据流执行
  - 实现简单

- 缺点
  - 会在 WAR 和 WAW 上 stall
  - function 的等待空间有限

Tomasulo's Algorithm

Register Renaming

WAW 和 WAR 不是真正的依赖，因为可以用寄存器命名解决。产生 WAR，WAW 的根本原因是寄存器不足

寄存器重命名就是用某些条目的 ID 来代替寄存器的 ID，根据该条目的 ID 来访问寄存器的值。可以消除所有 WAW 和 WAR

一个更好的乱序执行模型需要

- 将数据的 consumer 和其 producer 联系起来：通过寄存器重命名
- 将未就绪的指令缓存起来直到其就绪：reservation station
- 指令需要直到源值的就绪情况：当值就绪时和其 tag 一起广播
- 当所有源值就绪，需要将指令分发至 function unit 执行

Tomasulo's algorithm

乱序执行 + register renaming

通过 CDB (Common Data Bus) 实现 bypass，由寄存器重命名消除 WAW，WAR

scoreboard 用 RS (Reservation Station) 代替

- 操作数就绪后立即 fetch 并缓存
- 执行时不需要从寄存器获取值
- 如果有多条指令写同一个寄存器，只有最后一条会更新寄存器的值

Tomasulo's Implementation

Reservation Station 的状态有

- Busy：指示是否被占用
- Op：要执行的操作
- V_j, V_k：源操作数的值
- Q_j, Q_k：产生源操作数的 reservation station
- A：load/store 的地址

寄存器的状态指示哪个 function unit 将写寄存器

Stages of Tomasulo

issue

如果有可用的 reservation station，将指令和重命名后的指令加入 reservation station

否则 stall

execute

reservation station 中的每条指令监视 CDB

如果 CDB 上的数据和需要的源操作数的 tag 相符，则将其值存入 reservation station

如果源值均就绪，则分发指令到 function unit

write result

执行完成后将值和 tag 在 CDB 上广播

寄存器堆和 CDB 相连，每个寄存器都有一个 tag 指示最后一个写该寄存器的指令，如果 tag 符合则更新寄存器的值，并且设有效位

Data buses

一般的数据总线是数据 + 目的

CDB 是数据 + 源，由 64 bit 的数据和 4 bit 的 function unit 地址组成，数据在总线上广播

Review

Tomasulo 中

- RAW：在 RS 中记录未准备好的操作数的 tag，当数据就绪时广播数据和 tag，指令就绪时分发，CDB 实现了 forwarding 的功能
- WAR：由 RS 实现了重命名的功能
- WAW：寄存器由最后一个写的 function unit 更新
- 结构冒险：一个 function unit 有多个 RS 条目

如果同一周期有多个指令就绪，从中选择一个分发，如果同一周期有多个指令完成，一般是等待 CDB

增加寄存器的读写 port 和 function unit 可以实现多条指令同时 issue 和完成

如果 load 和 store 地址相同，load 会检查所有执行中的 store，而 store 会检查更早的 load 和 store

Tomasolu 没有提供精确中断的解决方案

tradeoff：

- 优点
  - 分布式的依赖检测，不是通过中心的寄存器堆检测数据依赖，而是通过单次的广播来通知多条指令
  - 解决了 WAW 和 WAR
- 缺点
  - 性能受限于 CDB，包括
    - tag 比较
    - 大量的读写
    - 如果单 CDB，每周期只有一条指令能完成
  - 没有精确中断
  - RS 中数据冗余

Alternative Methods for Register Renaming

Two sets of register file

维护两个寄存器堆

- ARF (Arhitected Register File)：程序员可见的寄存器堆
- RRF (Rename Register File)：用于暂时存储的寄存器

当读取源值时，如果 ARF 中条目非占用（没有指令要写），则从中读取，否则检测 RRF，如果其中数据有效则读取，否则等待

分配目的值时，选择一个空闲的 RRF 条目，然后将其分配给指令，修改对应 ARF 的映射关系，然后将 RRF 和 ARF 条目均设为占用

更新寄存器时，将 RRF 值拷贝到对应 ARF，然后将条目置为非占用

One more step

可以在硬件上仅保存物理寄存器，结构寄存器只是映射关系，不需要将数据拷贝到 RS，只用拷贝物理寄存器的 tag

物理寄存器的数目大于结构寄存器

> e. g. Alpha 21264
>
> 寄存器重命名的过程大概是这样的，首先有一个寄存器映射状态表存储结构寄存器与物理寄存器的映射关系，当译码指令时，处理器访问映射状态表获得源寄存器对应的物理寄存器和其中的值，然后寻找一个空闲的物理寄存器，作为目的结构寄存器对应的物理寄存器，将映射关系存储到 map 中，当一条指令完

> 成时，将结果写入物理寄存器，将该寄存器标为 valid，然后将物理寄存器号和结果数据广播出去，当一条指令 retire 时，将该物理寄存器标记为以提交，解除原本的目的寄存器与对应的物理寄存器的映射关系
>
> 初始时所有与结构寄存器对应的物理寄存器处于 AR 状态，其余寄存器处于 Available 状态，译码指令时一个 available 的寄存器被选为目的寄存器，其状态转换为 rename buffer，not valid，如果指令执行完成则会转换为 valid，当指令提交时，该物理寄存器转换为 AR，同时原本的目的结构寄存器对应的物理寄存器转换为 available