MIPS Pipeline

Introduction to MIPS ISA

MIPS R2000

- 32 bit 的 PC
- 4 GB 内存空间
- 32 个 32 bit 的 GPR
- 支持 32 位有符号/无符号整数，以及 IEEE 754 标准的单双精度浮点数

指令可以分为三类

- R-type：三个寄存器操作数
- I-type：两个寄存器操作数，一个 16 bit 立即数
- J-type：26 bit 立即数操作数

指令定长，32 bit，load/store 指令采用寄存器+偏移的方式

Single and Multi-Cycle Microarchitecture

Processing Instruction

执行指令就是将当前架构的状态（AS, Architectural State）转换到新的状态，而具体的转换规约由 ISA 给出（FSM），以 ISA 的视角来看的话 AS 直接的转换是没有中间状态的

对于微架构来说，可以有多种实现

- 在一个周期内完成转换
- 在多个周期中完成转换

Single-cycle machine：每个指令耗时一个时钟周期，在指令执行结束更新状态，最大的缺点是短板效应：最慢的指令决定了时钟周期的长度

Multi-cycle machine：将指令的执行分为多个阶段，在指令执行过程中状态即可发生变化，而 AS 只在指令执行结束后变化

执行指令也可以看作将旧的数据变为新的数据，一个执行引擎包含两个部分

- datapath：用于处理和传送数据信号的硬件部分，包括 function unit，总线，选择器，存储设施等
- 控制逻辑：决定控制信号的硬件部分

Instruction Processing for MIPS

MIPS 指令的执行由五个部分组成

- IF: instruction fetch
- ID: instruction decode and register operand fetch
- EX: execute/evaluate memory address
- MEM: memory operand fetch
- WB: store/writeback result

Performance Analysis

指令执行时间：CPI $\times$ clock cycle time

程序执行时间：$\sum$ CPI $\times$ clock cycle time

> Iron law
>
> # of instructions $\times$ Average CPI $\times$ clock cycle time

对于 single-cycle machine，所有指令 CPI 都是 1，而 clock cycle time 会比较长

对于 multi-cycle machine，指令 CPI 各不相同，而 clock cycle time 较短

决定执行时间的关键要素就是 CPI 和 clock cycle time

Single-cycle uArch

critical path 在于访存，最慢的指令会拖长所有指令的执行时间

类似 x86 中 REP 之类的指令难以实现

难以优化：优化普通情况没有提升，只能瞄准最差情况优化

控制信号和数据信号在同一时钟周期产生

Multi-cycle microarchitecture

时钟周期时间和指令时间独立，每条指令按其需要决定要执行几个周期

实现类似于 FSM

可以再添加一个 ISA 和硬件间的抽象层：microcode

- 将 ISA 翻译成微码的序列
- 微码可以看作一个用户不可见的 ISA
- 可以支持 ISA 的扩展，只需要修改微码即可
- 可以支持非常复杂的指令

Microarchitecture Design Principles

critical path：减少最大的组合逻辑延迟，如果一条通路延迟太长则考虑拆分成多个周期

bread and butter design：在最重要的部分花费时间做优化：common case

balanced design：平衡设计，减少瓶颈

如果采用 multi-cycle 的设计，可以利用上述原则

- 减少 critical path 与指令的 worst case 无关
- 可以优化执行重要指令的阶段数
- 不需要提供超出需要的资源，如果一个指令在不同的阶段需要同一个数据，只需要提供一次，复用即可

MIPS Pipeline

Limitation of multi-cycle

并发度低：大部分硬件资源在不同的阶段是空闲的

解决方法：提高并发度，当处理一条指令时，用空闲的资源处理其他指令的其他阶段

> throughput 和 latency 的区别
>
> - throughput：给定时间能执行多少任务
> - latency：执行一个任务要多少时间

提高并发可以提高指令的吞吐量——执行一条指令的时间没有缩短，但同一时间可以执行更多指令了

并行可分为不同种类

- 应用层的并行
  - DLP (Data-Level Parallelism)：同时处理多个数据
  - TLP (Task-Level Parallelism)：同时处理多个任务
- 硬件层的并行
  - ILP (Instruction-Level Parallelism)：流水线，超标量，预测执行等
  - 向量机，GPU
  - TLP (Thread-Level Parallelism)：同时多线程（SMT）
  - RLP (Request-Level Parallelism)：云计算

流水线就属于 ILP 的范畴

Basic Idea for Pipeline

思路

- 将指令执行分为不同的阶段
- 确保硬件资源足够在每个阶段执行一条指令
- 在每个阶段执行不同的指令（按照程序中的顺序）

指令执行就好像工厂的流水线装配一样

流水线的优点是可以提升 throughput，理想的执行可以达到一个周期一条指令

流水线要求

- 不同的指令间没有依赖
- 不同的阶段不共享资源

同样，最慢的阶段决定了 throughput 的大小

理想的流水线：指令独立无依赖，且指令能被分成独立且平均的多个阶段

但是指令流水线不是一个理想的流水线

- 指令不同，所需要的阶段也不尽相同，会产生外部碎片（对某些指令来说有些阶段是空闲的）
- 不同的阶段 latency 不同，但需要以同一个时钟周期控制，会产生内部碎片（过快的阶段会空闲）
- 指令间有依赖，需要解决依赖以确保结果正确

在设计流水线时需要解决下述问题

- 平衡各个阶段的任务
- 保证流水线能够正确运行，持续运行，以及满载
- 处理异常和中断
- 最小化流水线的 stall

Pipeline Hazard

流水线中会出现三种冒险

- structural hazard：硬件不能满足所有情况下指令的组合，会出现竞争资源的情况
- data hazard：指令依赖的还在流水线中的指令的结果
- control hazard：取指令和改变控制流之间的延迟

Structural Hazards

在不同 stage 的指令需求同一资源

一般有两种解决方案

- 减少争抢的原因，如增加资源
- 检测资源争抢并且将其中一个指令 stall

Data Hazards

可以分为三类

- flow dependence: RAW, true data dependence
- Output dependence: WAW
- Anti dependence: WAR

RAW 依赖是程序本身的值依赖决定的，而 WAW 和 WAR 是因为寄存器数量有限导致的，是名字的依赖而非值的依赖

在按序执行的机器中不需要处理 WAW 和 WAR 依赖，因为按照指令顺序写入寄存器

处理 RAW 有五种基本方法

- detect and wait：等待至值就绪
- detect and forward/bypass：使用硬件结构来转发
- detect and eliminate：在软件层面（编译）消除
- predict：预测性执行，然后验证
- do something else：如细粒度多线程

detect and wait

即 interlocking，在检测到 RAW 时 stall 流水线，可以基于硬件或者软件实现

> Scoreboard
>
> - 每个寄存器有一个有效位
> - 写寄存器的指令重置有效位
> - 解码时检查指令的操作数是否有效，无效则 stall 到有效为止

优点是简单，但缺点是浪费时间，且对于所有的依赖都要 stall

- WAR：乱序执行时会 stall
- WAW：两条 write 写同一个寄存器，硬件无法区分哪个指令重置了有效位

当两条指令的距离小于等于读阶段和写阶段的距离时就要 stall，stall 的方法有

- 停止取指，让 stall 的指令留在原本的 stage
- 插入无效指令，如 nop

data forwarding

基本思路：使用额外的数据通路，在数据准备好时直接转发到需要的位置，一般路径有

- EX/MEM to EX
- MEM/WB to EX
- 寄存器堆内部的转发，即在同一周期内读写

需要额外注意指令的顺序，且不能完全避免 stall，即 lw 之后的指令

detect and eliminate

可以使用静态的方法调度，在编译阶段实现，消除依赖

也可以采用动态的方法调度，由硬件实现，乱序执行

静态调度的优缺点是

- 优点
  - 编译器更了解数据依赖
  - 硬件简单
- 缺点
  - 硬件不同导致调度不同，如不同的流水线和不同的 latency
  - 有些信息编译器难以获得，如运行时确定的值，这样需要 profiling

value prediction

预测式地执行，执行后验证

但是难以预测正确的值，并且需要回滚以解决错误

fine-grained multithreading

轮流执行不同线程的指令，避免依赖导致的 stall

优点是不需要线程内的处理依赖的逻辑

缺点是会影响单线程的性能，且需要额外的逻辑保证不同线程的上下文