Pipline

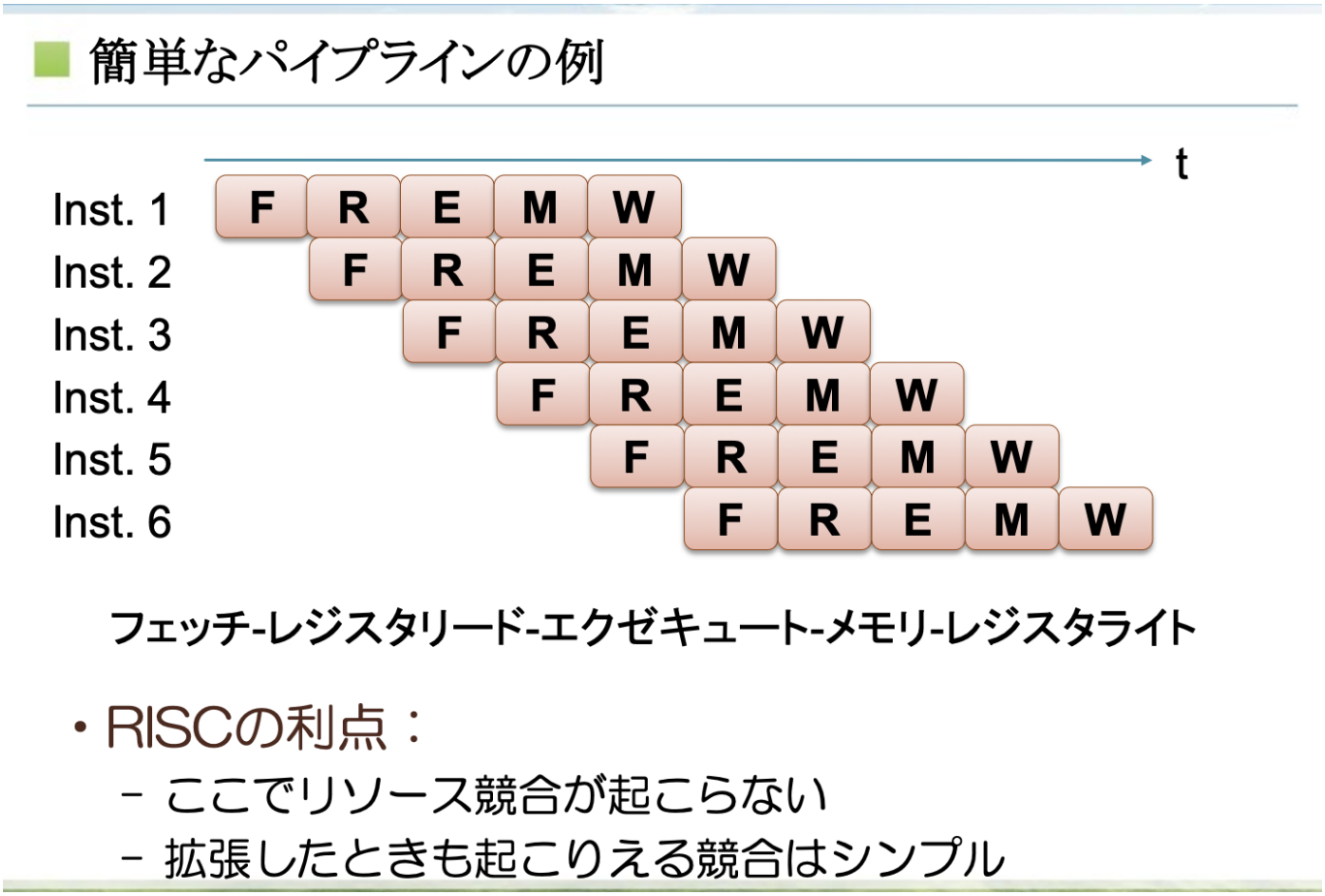
パイプライン化" (Pipelining)

- 将多条指令错开，使其重叠执行 (Overlap Execution)。
- 这是一种在不复制硬件资源的前提下提高吞吐量 (Throughput) 的经典方法。

流水线化是一种通过将指令执行过程拆分成多个阶段，并让多条指令在这些阶段中并行/重叠执行的技术，从而在不增加核心硬件数量的情况下，显著提高处理器吞吐量的优化方法。

简单的流水线示例 Simple Pipeline Example

1. 流水线阶段划分： F：フェッチ (Fetch) - 取指令 R：レジスタリード (Register Read) - 译码/读寄存器 E：エグゼキュート (Execute) - 执行 (主要由 ALU 完成运算) M：メモリ (Memory) - 访存 (对于 Load/Store 指令) W：レジスタライト (Register Write) - 写回 (将结果写回寄存器)



RISC V的利点：

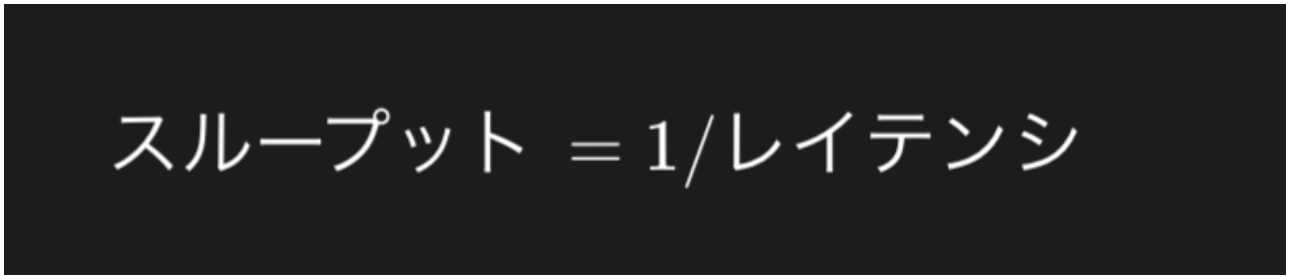
- 在这里（指 5 级流水线设计中），不会发生资源竞争。
- 解释：在 RISC 架构中，指令的结构简单且长度一致，使得流水线阶段的划分相对均匀和独立，取指令 (F) 和访存 (M) 阶段通常不会在同一个周期内竞争同一个资源（例如，指令访问内存和数据访问内存被划分到不同阶段）。
- 即使进行扩展时，可能发生的竞争也是简单的。
- 解释：RISC 的简单性使得它在扩展流水线级数或增加功能单元时，所需的控制逻辑和解决冲突（Hazard）的机制相对简单。

"効果の見積もり" (Estimating the Effect/Performance)，即估算流水线技术的性能效果。

- 朴素结论： 在理想情况下，将指令执行分成 n 级流水线，理论上可以使处理器的时钟频率提高 n 倍，从而使吞吐量也提高 n 倍。
- 分段延迟不均匀： 如果 F、R、E、M、W 各阶段的时间长度不相等，则流水线周期必须由最慢的那个阶段决定，所以实际提速将小于 n 倍。
- 流水线寄存器开销： 在各级之间插入流水线寄存器会引入额外的延迟（Delay latch），这也会降低实际可达到的频率。
- 冲突/冒险 (Hazards)： 实际程序中存在的数据冒险和控制冒险会导致流水线停顿 (Stall)，降低实际的指令吞吐率。

吞吐量与延迟 (Throughput and Latency)

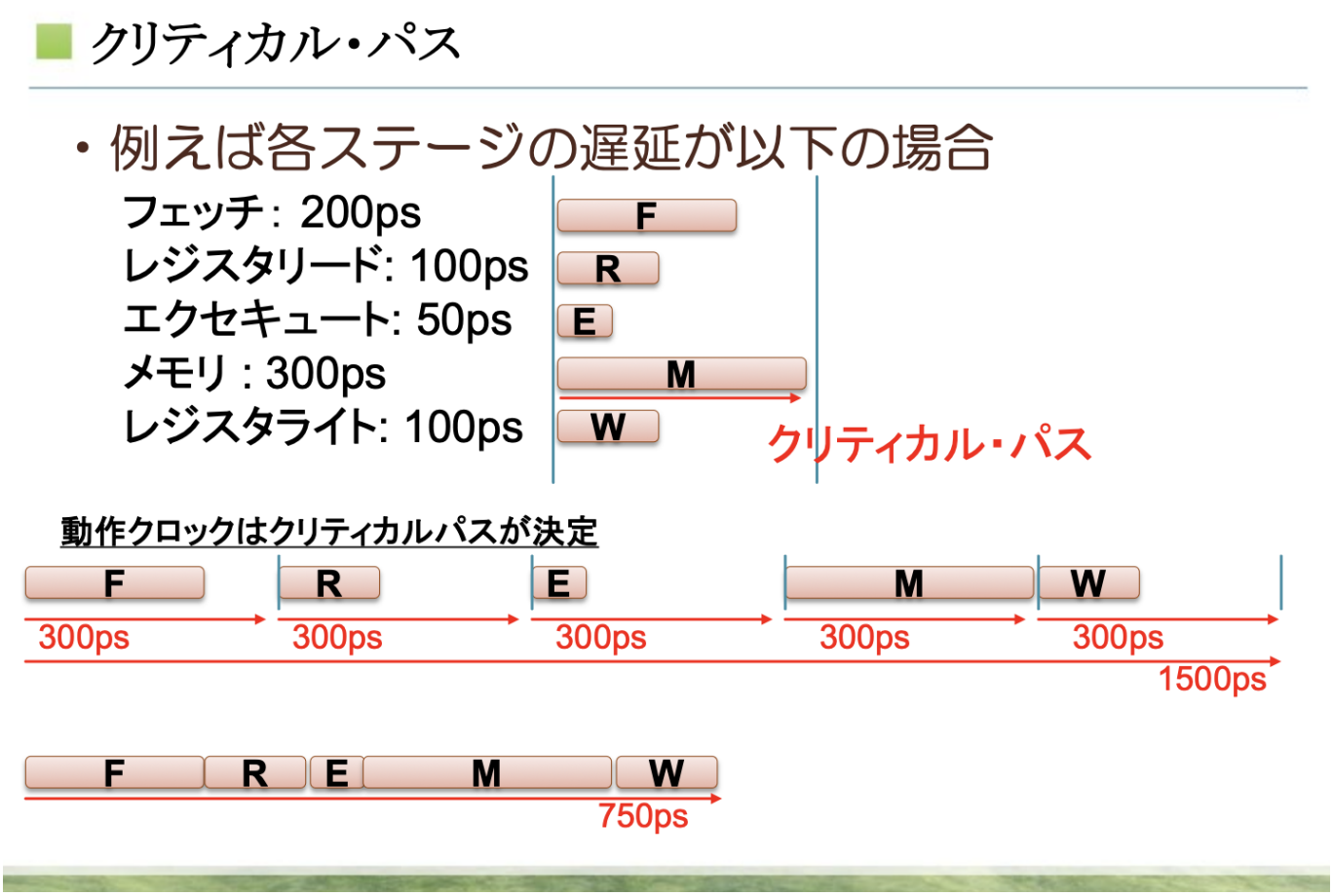
- レイテンシ (Latency):即：延迟。指单个任务从开始到结束所花费的总时间。
- スループット (Throughput):即：吞吐量。指单位时间内系统能完成的任务总量。
- もし、オーバーラップや並列処理がなければ (If there is no overlapping or parallel processing):



降低单个任务的延迟通常意味着需要提高单个晶体管或电路的开关速度，以及减小信号传播的距离。这受到物理极限（如半导体技术和光速）的制约，难度非常大，提高吞吐量通常可以通过架构上的创新来实现。

尽管实际影响并非理想中的完美 n 倍加速，但通过流水线化，吞吐量的提升远大于延迟的微小增加，因此流水线化仍是提高处理器整体性能的最有效方法之一。

Critical Path



- 关键路径是流水线所有阶段中最耗时的阶段-M
- 它决定了流水线处理器的最短时钟周期（即最高时钟频率）。
- 虽然流水线中单个指令的总延迟 (1500ps) 增加了 (由于阶段时间拉齐)，但时钟频率 (≈3.33GHz) 远高于单周期处理器 (≈1.33GHz)，从而使吞吐量大幅提高。

流水线阶段的划分 (Stage Splitting)

提高效果的目标和方法：

1. 分割每个阶段，使所有阶段的长度（延迟）尽可能相等。
2. 然后持续地分割 ⟹ 超级流水线 (Superpipelining)。

- 以上两点只是为了获取更短的Critical Path

3. 与冒险 (Hazards) 和关键循环 (Critical Loop) 的权衡。

- 阶段越多，指令之间的依赖关系（数据冒险、控制冒险）需要更长的周期才能解决，解决这些冒险所需的硬件（转发路径、停顿逻辑）也更复杂，导致流水线气泡增多，可能抵消频率提高带来的益处。

实现阶段分割的途径：

1. 組み合わせ回路の分割 (Splitting Combinational Circuits)

- パイプラインレジスタ挿入 (Insertion of Pipeline Registers)

2. メモリ回路 (Memory Circuits)

- ウェーブパイプライン, バーストパイプライン (Wave Pipelining, Burst Pipelining)

- 解释： 内存访问本身延迟较高。可以采用更高级的技术（如波形流水线，利用信号传播时间差异；或突发流水线，一次性访问多个数据）来优化内存子系统的吞吐量，使其更好地融入流水线。

3. 分割したければ基本的に分割可能 (If you want to split it, it is basically possible)

- 结论： 从技术上讲，几乎所有长延迟的电路路径都可以通过插入流水线寄存器的方式进行分割，以缩短时钟周期。但关键在于权衡其带来的复杂度和冒险开销。

通过**平均化**和**细分流水线阶段**来最大化时钟频率的方法。尽管分割阶段可以在理论上获得更高的频率（即更高的吞吐量），但冒险（如停顿和转发）的增加、以及流水线寄存器本身引入的开销，使得性能提升并非线性关系，设计者需要在高频率和低复杂度/少停顿之间找到最佳平衡点。

クロックのオーバーヘッド" (Clock Overhead)

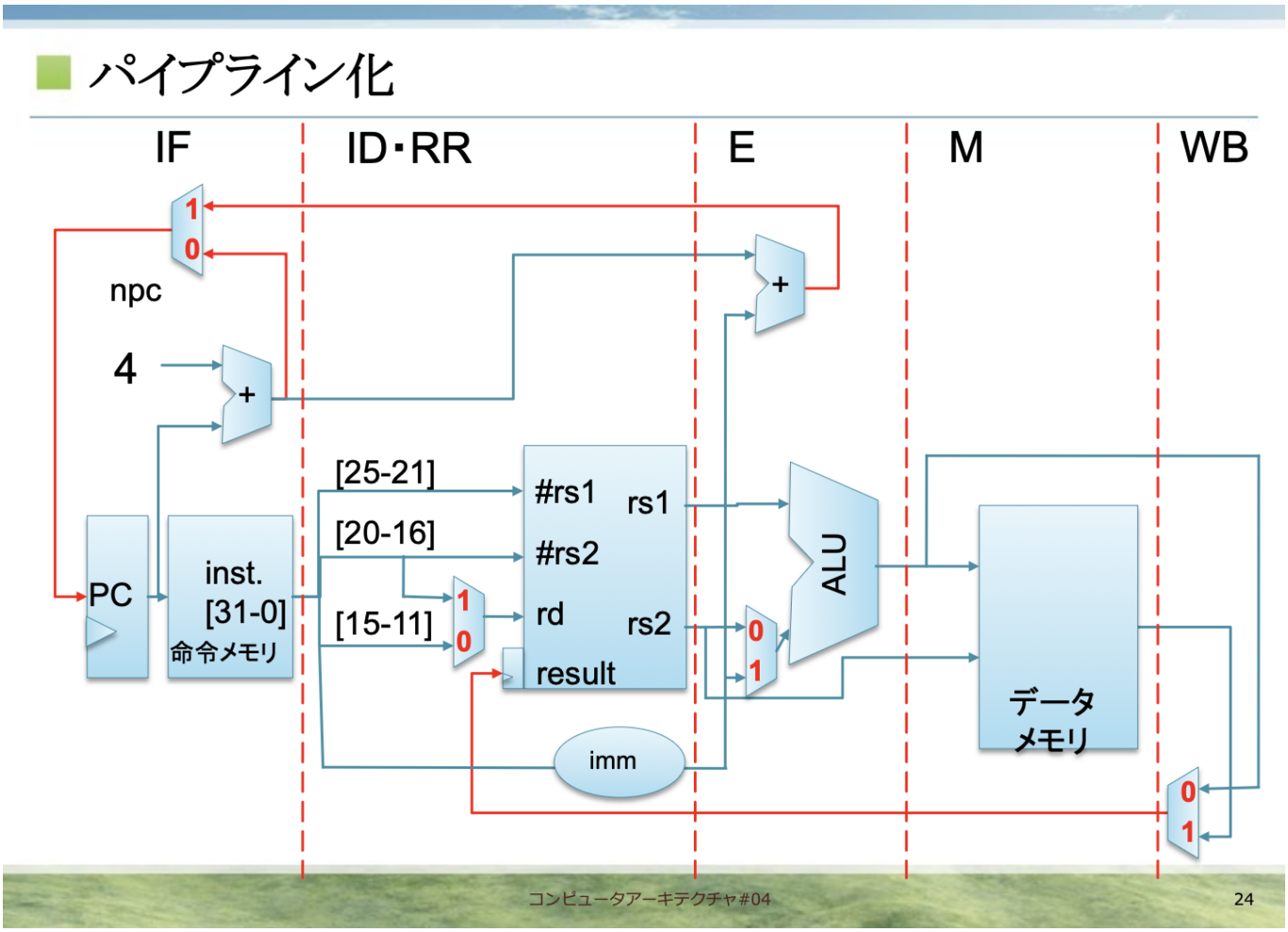
时钟频率越高 (时钟周期越短)，开销所占的比例越大。

主要开销类型：

- 时钟信号到达芯片上不同位置的寄存器的时间差异。由于布线长度、电容和电阻的不同，时钟信号不是同时到达所有地方，这个差异必须计入时钟周期内。
- 单个时钟周期长度（脉冲宽度）在不同周期之间发生的随机变化。这也是由电源噪声、温度变化等因素引起的，必须在设计时预留安全裕度。
- 为了确保芯片在最恶劣的条件（高温、低压、最差的制造工艺）下仍能稳定工作，设计者必须预留额外的安全裕度。

当阶段被分得越细，这些开销（skew, jitter, margin 和寄存器本身的延迟）所占的比例就会越高，最终限制了时钟周期的进一步缩短。

簡単なパイプラインプロセッサの設計



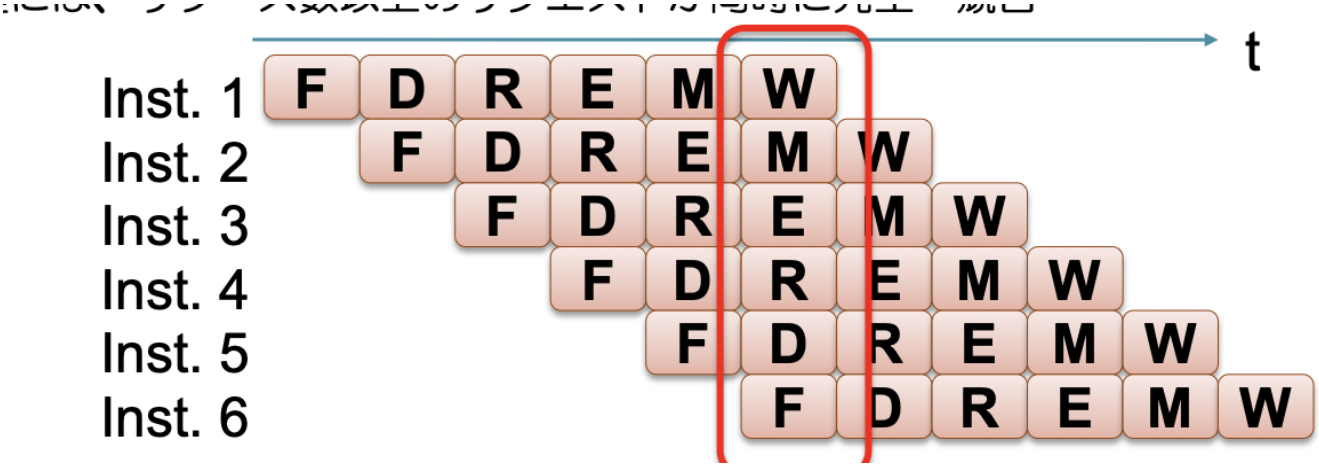
パイプライン高性能化手法

パイプラインの性能低下要因

- ナイーブにはn段に刻めばスループットn倍を期待
- クロックに起因するオーバヘッド
- バブル

構造ハザード

同じ資源を同時に使おうとして不足



- 准确地说，是同时发生的请求数量超过了资源的实际数量，从而发生竞争。
- 易于流水线化是 RISC (精简指令集计算机) 的优点。
- 解决方法可以插入Bubble
- 结构冒险的解决办法之一是增加资源数量。

Data Hazrad

- 数据冒险发生在指令之间存在数据依赖关系，而流水线无法按顺序正确执行这些依赖操作时。

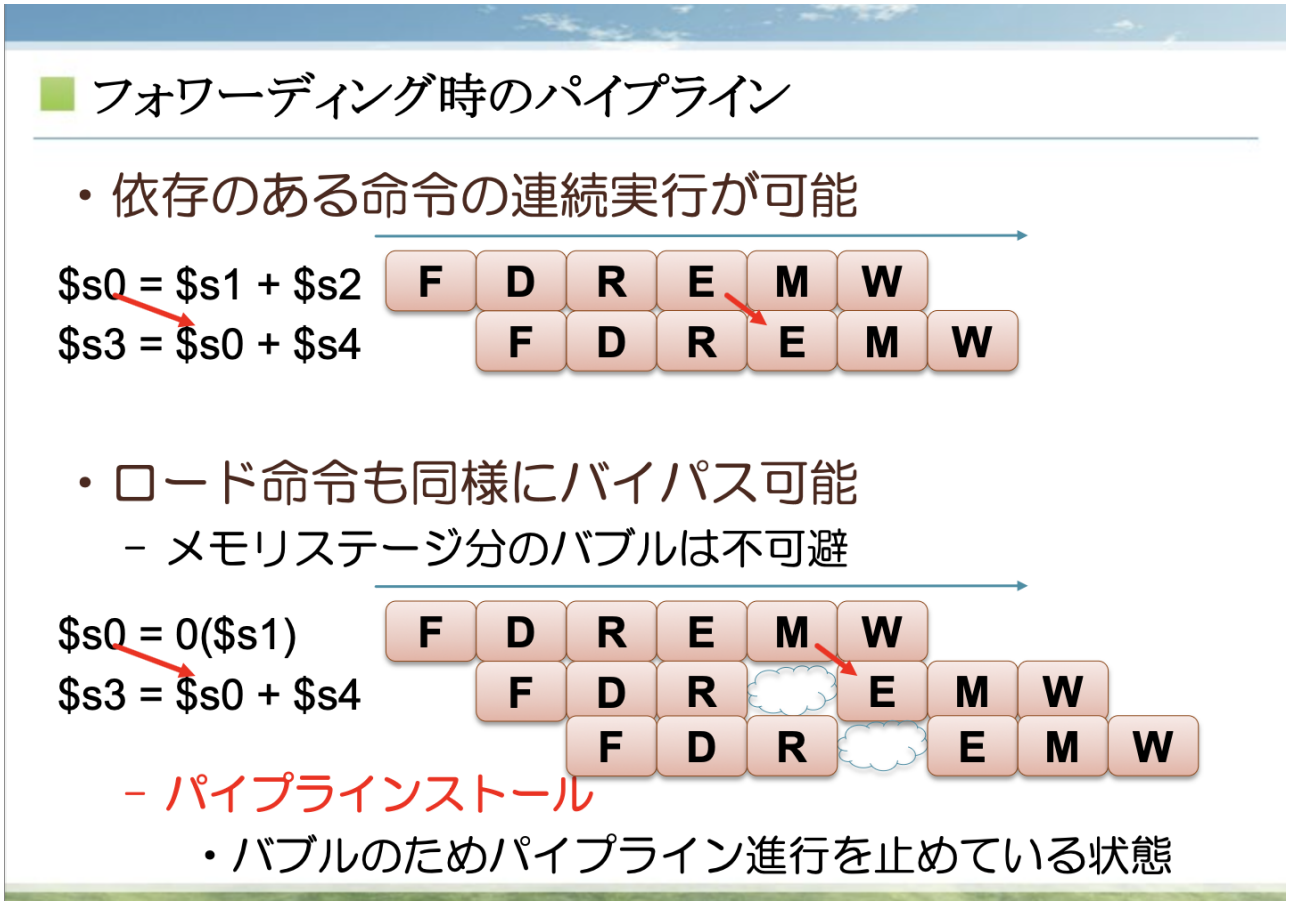
解决方法：

- 如果编译器努力 (编译器优化)，可以减少此类情况。
- 然而，由于这种代码经常出现，通常由硬件来处理：フォワーディング (バイパシング) (Forwarding / Bypassing)
- 解释： 这是最主要的硬件解决方式。它通过内部旁路，直接从流水线内部的寄存器（例如 ALU 的输出）将尚未写回寄存器堆的最新计算结果，转发给需要该数据的后续指令的 ALU 输入端。这样可以避免停顿，显著提高性能。

Load-use Hazard

Load-Use 冒险 (Load-Use Hazard): 这是唯一无法通过简单的单周期转发完全解决的数据冒险。

- Load-Use 冒险 (Load-Use Hazard): 这是唯一无法通过简单的单周期转发完全解决的数据冒险。
- Load 指令产生结果（M 阶段）和使用结果的指令需要结果（E 阶段）之间存在一个时间差，即加载指令的数据回来得太晚，无法赶上下一条指令的 ALU 输入。
- 解决方案： 硬件必须插入一个气泡/停顿 (Stall)，延迟使用数据的指令一个时钟周期，以等待 Load 指令从内存中取出数据，再通过转发网络将其传递给后续指令。



如何stall方法：

1. 停顿的判断 (D ステージで判断 - Decided in the D stage):

- 刚刚译码的指令是 Load 指令 (Inst. 1)，并且它的目标 (Destination) 寄存器编号...与当前指令 (Inst. 2) 所需的源 (Source) 寄存器编号一致。

2. 插入气泡 (D ステージの出力をNOPに - Set D Stage Output to NOP): 当检测到冒险时，处理器会将 Inst. 2 在 D 阶段的输出（即送往 E 阶段的指令和控制信号）替换成一条 NOP (空操作指令)。这条 NOP 会沿着流水线前进，占用一个周期，但它不会改变任何寄存器或内存状态。
3. 冻结 IF 和 ID 阶段 (F ステージとDステージを止める - Stop F and D Stages):简单来说，冻结 F/D 阶段和 PC 的作用就是让流水线暂停接收新的指令 (Inst. 3)，并让 Inst. 2 留级一个周期，直到 Inst. 1 的数据及时赶到。

制御ハザード

- 次の命令が不明な場合

What is the Effect?

1. 核心评估指标：CPI
2. 一个处理器在每次遇到分支指令时都会停顿 2 个周期，它执行的代码中，有 5% 的指令是分支指令。

制御ハザードへの対策

1. 延迟分支 (遅延分岐 - Delayed Branch)

- 在分支指令之后的 n 条指令 (延迟槽) 执行完后，才应用分支指令的控制。
- 编译器将合适的指令填充到延迟槽中。

2. 分支预测 (分岐予測 - Branch Prediction)

- 预测下一条指令的地址，并连续取指 (Fetch)。
- 与预测错误时的恢复机制配套使用。

Summary：

- 延迟分支 (Delayed Branch): 是一种软件/硬件结合的静态解决方案，要求编译器优化代码。它在早期的 RISC 架构中很流行，但随着流水线级数增多，它的效率变低。

- 分支预测 (Branch Prediction): 是一种硬件的动态解决方案，旨在通过高准确率的猜测来避免停顿。这是现代高性能处理器解决控制冒险的主要方法。

分歧预测实装

1. 预测的原理和目标：

- 在分支指令执行之前，预测下一条指令的地址。
- 即使是简单的预测，命中率也出奇地高（90%以上）。
- 通过 PC 预测下一个 PC。

2. 实现连续执行

3. 预测失败时的处理：

- 立即清空/冲刷 (Flush) 流水线中所有错误的投机性指令（将它们替换为 NOP）。
- 将 PC 重置为正确的分支目标地址。
- 从正确的地址重新开始取指令。

スカラプロセッサ

- 标量处理器： 一次处理一个数据元素，通过流水线提高吞吐量。
- 超标量处理器： 具有多个执行单元，可以在一个周期内并行处理多条不相关的指令，是更高级的加速技术。
- Out-of-Order (OoO)： 通过乱序执行不相关指令来隐藏延迟，是超标量处理器实现高性能的关键技术。

The author is 11_sum