

计算机组成和体系结构

第六讲

四川大学网络空间安全学院

2020年3月30日

封面来自wccftech.com

版权声明

课件中所使用的图片、视频等资源版权归原作者所有。

课件原创内容采用 [创作共用署名—非商业使用—相同方式共享4.0国际版许可证\(Creative Commons BY-NC-SA 4.0 International License\)](#) 授权使用。

Copyright@四川大学网络空间安全学院计算机组成与体系结构课程组，2020



上期内容回顾

- 数据的存储
 - 大端存储/小端存储
- 指令格式
 - 堆栈型、累加器型、通用寄存器型
 - 存储器—存储器、寄存器—存储器、取—存
 - 影响操作数存储方式的因素：性能、指令长度
 - 扩展操作码
- 内存寻址
 - 立即寻址、直接寻址、间接寻址、寄存器寻址、变址寻址、基址寻址

本期学习目标

- 指令流水线
 - 指令流水线的性能分析
 - 流水线冒险
 - 流水线对于指令集架构设计的影响
- 现实中的指令集架构

中英文缩写对照表

英文缩写	英文全称	中文全称
ILP	Instruction-level Parallelism	指令级并行
ISA	Instruction Set Architecture	指令集架构
MIPS	Microprocessors without Interlocked Pipeline Stages	MIPS指令集

指令集架构2

流水线：厨房

- 做一道菜需要多个步骤：
 - 洗菜、切菜、炒菜、摆盘
- 一个厨房有多个师傅
 - 分别负责洗菜、切菜、炒菜、摆盘
- 假设每个单位时间每个师傅可以处理一份菜
- 厨房依次收到许多订单：如何保证正确性的前提下尽快完成所有订单



图片来源：flaticon.com

流水线：厨房

- 做一道菜需要多个步骤：
 - 洗菜、切菜、炒菜、摆盘
- 一个厨房有多个师傅
 - 分别负责洗菜、切菜、炒菜、摆盘
- 假设每个单位时间每个师傅可以处理一份菜
- 厨房依次收到许多订单：如何保证正确性的前提下尽快完成所有订单



采用流水线的厨房

图片来源：flaticon.com

流水线在生产生活中的例子



图片来源：<http://www.swop-online.com/news/info/458.html?mid=84>



图片来源：<https://club.geely.com/>



图片来源：<http://www.ccaonline.cn/news/hqtx/569587.html>

指令流水线：可行性

- 做菜分为洗、切、炒、摆
- 所有的菜都按照洗、切、炒、摆的顺序执行
- 由专门的师傅分别负责洗、切、炒、摆
- 每个师傅的工作是独立的

指令流水线：可行性

- 做菜分为洗、切、炒、摆
 - **指令的执行不是原子的：一条指令的执行通常分为多个步骤**
 - 所有的菜都按照洗、切、炒、摆的顺序执行
-
- 由专门的师傅分别负责洗、切、炒、摆
 - 每个师傅的工作是独立的

指令流水线：可行性

- 做菜分为洗、切、炒、摆
- 指令的执行不是原子的：一条指令的执行通常分为多个步骤
- 所有的菜都按照洗、切、炒、摆的顺序执行
- 所有的指令都可以划分为类似的几个步骤，并且各步骤之间的顺序一致
- 由专门的师傅分别负责洗、切、炒、摆
- 每个师傅的工作是独立的

指令流水线：可行性

- 做菜分为洗、切、炒、摆
- **指令的执行不是原子的：一条指令的执行通常分为多个步骤**
- 所有的菜都按照洗、切、炒、摆的顺序执行
- **所有的指令都可以划分为类似的几个步骤，并且各步骤之间的顺序一致**
- 由专门的师傅分别负责洗、切、炒、摆
- **不同的阶段涉及到的CPU数据通路组件不同**
- 每个师傅的工作是独立的

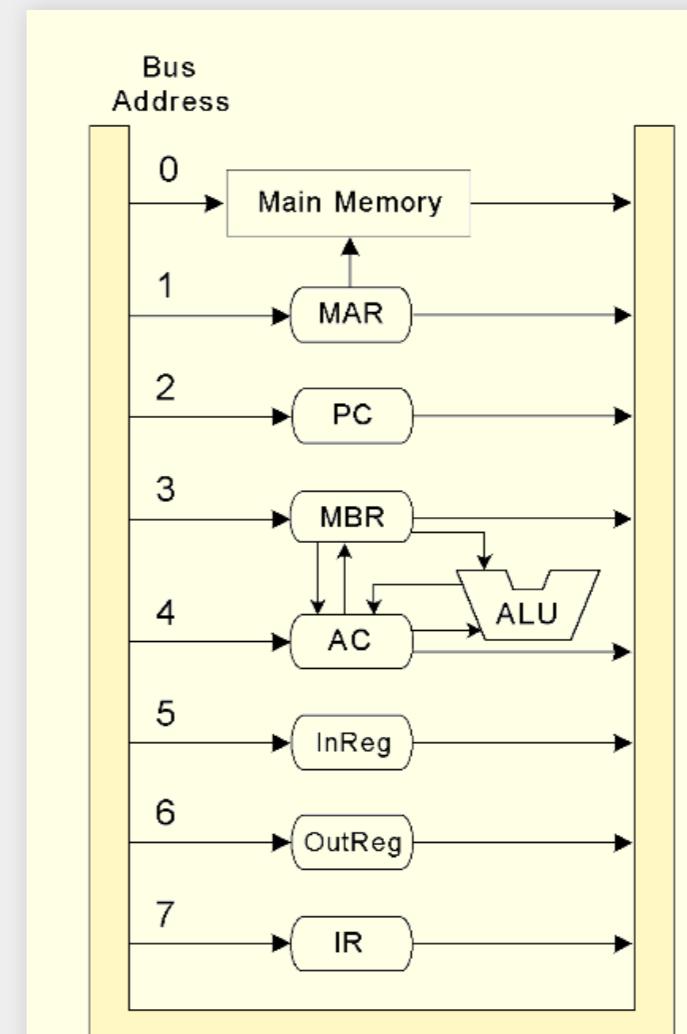
指令流水线：可行性

- 做菜分为洗、切、炒、摆
- **指令的执行不是原子的：一条指令的执行通常分为多个步骤**
- 所有的菜都按照洗、切、炒、摆的顺序执行
- **所有的指令都可以划分为类似的几个步骤，并且各步骤之间的顺序一致**
- 由专门的师傅分别负责洗、切、炒、摆
- **不同的阶段涉及到的CPU数据通路组件不同**
- 每个师傅的工作是独立的
- **不同阶段的组件可以同时运行**

指令流水线：可行性

- 做菜分为洗、切、炒、摆
- 指令的执行不是原子的：一条指令的执行通常分为多个步骤
- 所有的菜都按照洗、切、炒、摆的顺序执行
- 所有的指令都可以划分为类似的几个步骤，并且各步骤之间的顺序一致
- 由专门的师傅分别负责洗、切、炒、摆
- 不同的阶段涉及到的CPU数据通路组件不同
- 每个师傅的工作是独立的
- 不同阶段的组件可以同时运行

MARIE能否利用流水线？



流水线术语

- 阶段(Stage): 每个阶段对应了一个特定的步骤
 - N 级流水线包含了 N 个阶段
- 时延(Latency): 完整执行一条指令的时间
- 吞吐(Throughput): 单位时间CPU可以执行的指令数量
- 加速比(Speedup): 使用流水线之后的性能提升比例

流水线性能定性分析

采用流水线



时延=4, 利用率=3/6=0.5, 吞吐=3/6 = 0.5, 处理速度=1/6



时延=4, 利用率=3/12=0.25, 吞吐=3/12 = 0.25, 处理速度=1/12

流水线性能定性分析

采用流水线

- **不能**提高每一条指令的执行速度/降低指令的时延(Latency)



时延=4, 利用率=3/6=0.5, 吞吐=3/6 = 0.5, 处理速度=1/6



时延=4, 利用率=3/12=0.25, 吞吐=3/12 = 0.25, 处理速度=1/12

流水线性能定性分析

采用流水线

- **不能**提高每一条指令的执行速度/降低指令的时延(Latency)
- 提高了各组件的利用率(Utilization)



时延=4, 利用率=3/6=0.5, 吞吐=3/6 = 0.5, 处理速度=1/6



时延=4, 利用率=3/12=0.25, 吞吐=3/12 = 0.25, 处理速度=1/12

流水线性能定性分析

采用流水线

- **不能**提高每一条指令的执行速度/降低指令的时延(Latency)
- 提高了各组件的利用率(Utilization)
- 提高了指令的吞吐率(Throughput)



时延=4, 利用率=3/6=0.5, 吞吐=3/6 = 0.5, 处理速度=1/6



时延=4, 利用率=3/12=0.25, 吞吐=3/12 = 0.25, 处理速度=1/12

流水线性能定性分析

采用流水线

- **不能**提高每一条指令的执行速度/降低指令的时延(Latency)
- 提高了各组件的利用率(Utilization)
- 提高了指令的吞吐率(Throughput)
- 提高了整个程序的处理速度



时延=4, 利用率=3/6=0.5, 吞吐=3/6 = 0.5, 处理速度=1/6



时延=4, 利用率=3/12=0.25, 吞吐=3/12 = 0.25, 处理速度=1/12

流水线性能定量分析(1)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间都相等，并且等于时钟周期 t
- 对于 n 条指令，该流水线的加速比

$$\text{加速比} = \frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$$



流水线性能定量分析(1)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间都相等，并且等于时钟周期 t
- 对于 n 条指令，该流水线的加速比

加速比 = $\frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$

$$= \frac{nst}{st + (n - 1)t} = \frac{ns}{s + n - 1}$$



流水线性能定量分析(1)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间都相等，并且等于时钟周期 t
- 对于 n 条指令，该流水线的加速比

加速比 = $\frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$

$$= \frac{n st}{st + (n - 1)t} = \frac{ns}{s + n - 1}$$

理论加速比 = $\lim_{n \rightarrow +\infty}$ 加速比

$$= \lim_{n \rightarrow +\infty} \frac{ns}{s + n - 1} = s$$



流水线性能定量分析(1)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间都相等，并且等于时钟周期 t
- 对于 n 条指令，该流水线的加速比

加速比 = $\frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$

$$= \frac{nst}{st + (n - 1)t} = \frac{ns}{s + n - 1}$$

理论加速比 = $\lim_{n \rightarrow +\infty} \text{加速比}$

$$= \lim_{n \rightarrow +\infty} \frac{ns}{s + n - 1} = s$$



理论加速比等于流水线级数

流水线性能定量分析(1)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间都相等，并且等于时钟周期 t
- 对于 n 条指令，该流水线的加速比

$$\text{加速比} = \frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$$

$$= \frac{nst}{st + (n - 1)t} = \frac{ns}{s + n - 1}$$

$$\text{理论加速比} = \lim_{n \rightarrow +\infty} \text{加速比}$$

$$= \lim_{n \rightarrow +\infty} \frac{ns}{s + n - 1} = s$$



理论加速比等于流水线级数 为什么？

流水线性能定量分析(1)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间都相等，并且等于时钟周期 t
- 对于 n 条指令，该流水线的加速比

$$\text{加速比} = \frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$$

$$= \frac{nst}{st + (n - 1)t} = \frac{ns}{s + n - 1}$$

$$\text{理论加速比} = \lim_{n \rightarrow +\infty} \text{加速比}$$

$$= \lim_{n \rightarrow +\infty} \frac{ns}{s + n - 1} = s$$



理论加速比等于流水线级数 为什么？

s 级流水线最多可以并行执行 s 条指令 - 指令级并行

流水线性能定量分析(2)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间不都相等，第 i 个阶段需要 t_i ，故时钟周期 $t = \max_i^s t_i$
- 对于 n 条指令，该流水线的加速比

$$\text{加速比} = \frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$$

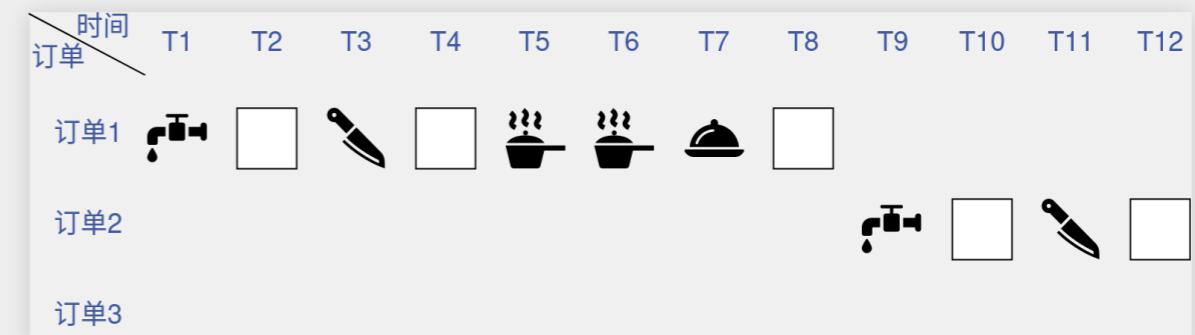


流水线性能定量分析(2)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间不都相等，第 i 个阶段需要 t_i ，故时钟周期 $t = \max_i^s t_i$
- 对于 n 条指令，该流水线的加速比

加速比 = $\frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$

$$= \frac{n \sum_{i=1}^s t_i}{st + (n-1)t}$$



流水线性能定量分析(2)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间不都相等，第 i 个阶段需要 t_i ，故时钟周期 $t = \max_i^s t_i$
- 对于 n 条指令，该流水线的加速比

加速比 = $\frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$

$$= \frac{n \sum_{i=1}^s t_i}{st + (n-1)t}$$

$$\lim_{n \rightarrow +\infty} \frac{n \sum_{i=1}^s t_i}{st + (n-1)t} = \frac{\sum_{i=1}^s t_i}{t} = \frac{\sum_{i=1}^s t_i}{\max_{i=1}^s t_i}$$



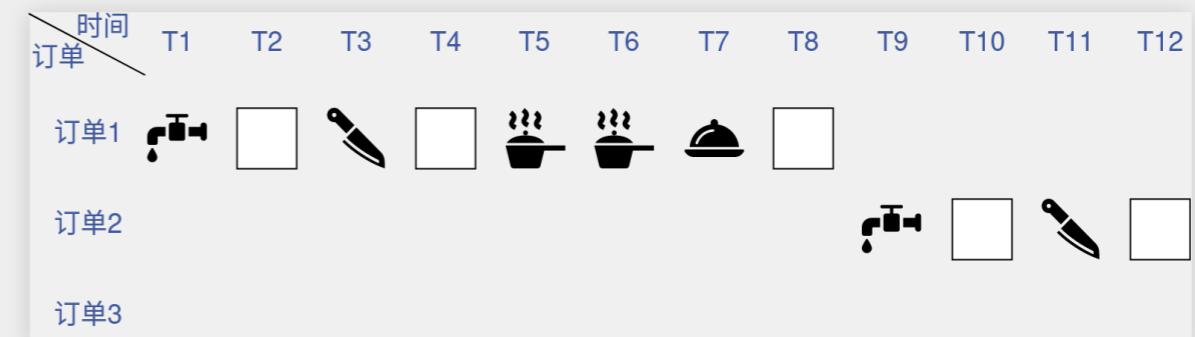
流水线性能定量分析(2)

- 假设有一条 s 阶段流水线，假设所有阶段都在一个时钟周期内完成(单周期模型)
- 假设各阶段执行时间不都相等，第 i 个阶段需要 t_i ，故时钟周期 $t = \max_i^s t_i$
- 对于 n 条指令，该流水线的加速比

加速比 = $\frac{\text{未使用流水线的时间}}{\text{使用了流水线的时间}}$

$$= \frac{n \sum_{i=1}^s t_i}{st + (n-1)t}$$

$$\lim_{n \rightarrow +\infty} \frac{n \sum_{i=1}^s t_i}{st + (n-1)t} = \frac{\sum_{i=1}^s t_i}{t} = \frac{\sum_{i=1}^s t_i}{\max_{i=1}^s t_i}$$



单周期模型中，理论加速比等于非流水线和流水线处理的**时钟周期的比值**

流水线冒险

目标：在**保证正确性**的基础上**提高处理性能**

- 通过对加速比的分析，可以发现流水线可以提高处理性能

问题：流水线是否会导致不正确的执行结果？

流水线冒险

目标：在**保证正确性**的基础上**提高处理性能**

- 通过对加速比的分析，可以发现流水线可以提高处理性能

问题：流水线是否会导致不正确的执行结果？

流水线冒险(Hazard)：

- 结构冒险
- 数据冒险
- 控制冒险

流水线冒险

目标：在**保证正确性**的基础上**提高处理性能**

- 通过对加速比的分析，可以发现流水线可以提高处理性能

问题：流水线是否会导致不正确的执行结果？

流水线冒险(Hazard)：

- 结构冒险
- 数据冒险
- 控制冒险

下面我们介绍不同流水线冒险的具体内容、产生原因和解决方案

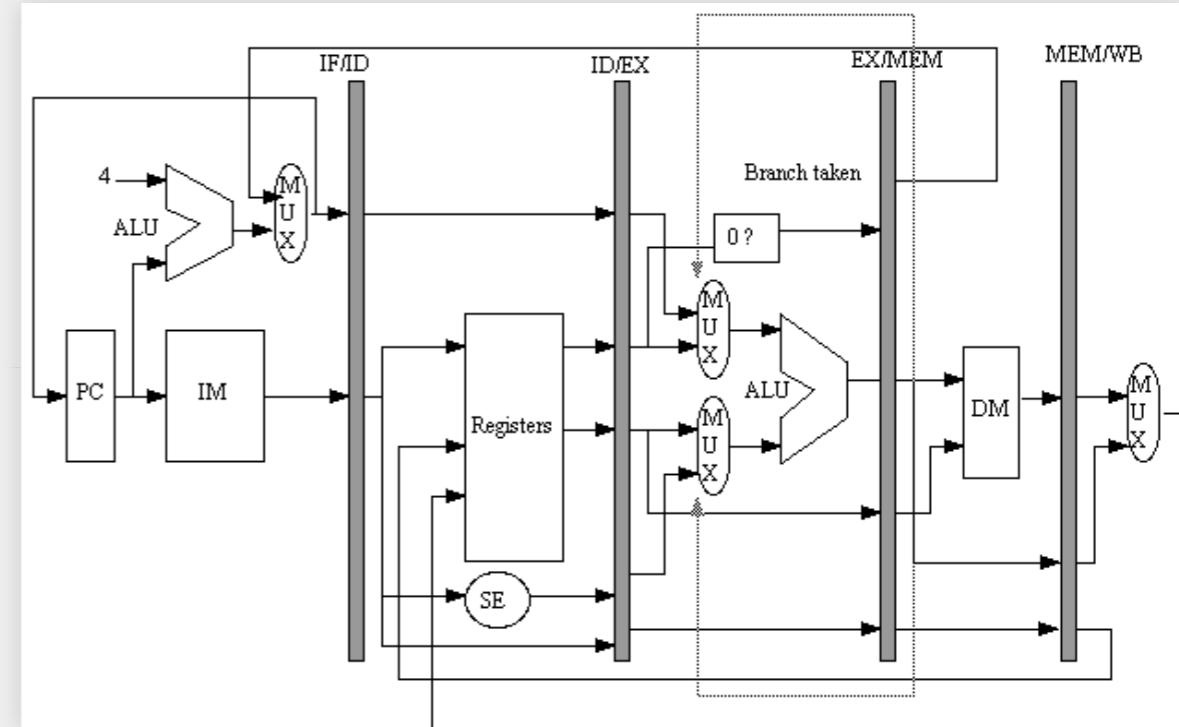
结构冒险

结构冒险(Structure Hazard): 硬件不满足流水线的条件

- 硬件无法支持多阶段并行
 - 例: MARIE的取指令和寄存器之间的数据转移使用同一条总线: 取指令阶段和执行指令无法同时进行
- 指令无法按照阶段依次执行
 - 例: MARIE 的指令中涉及多次内存访问

解决方法

- 设计数据通路使得各阶段可以并行执行
 - 例: MIPS数据通路包含指令存储器和数据存储器
- 设计指令使得指令可以按照确定的阶段顺序执行



图片来源: <https://user.eng.umd.edu/~yavuz/enee446/images-446/lecture446-10.htm>

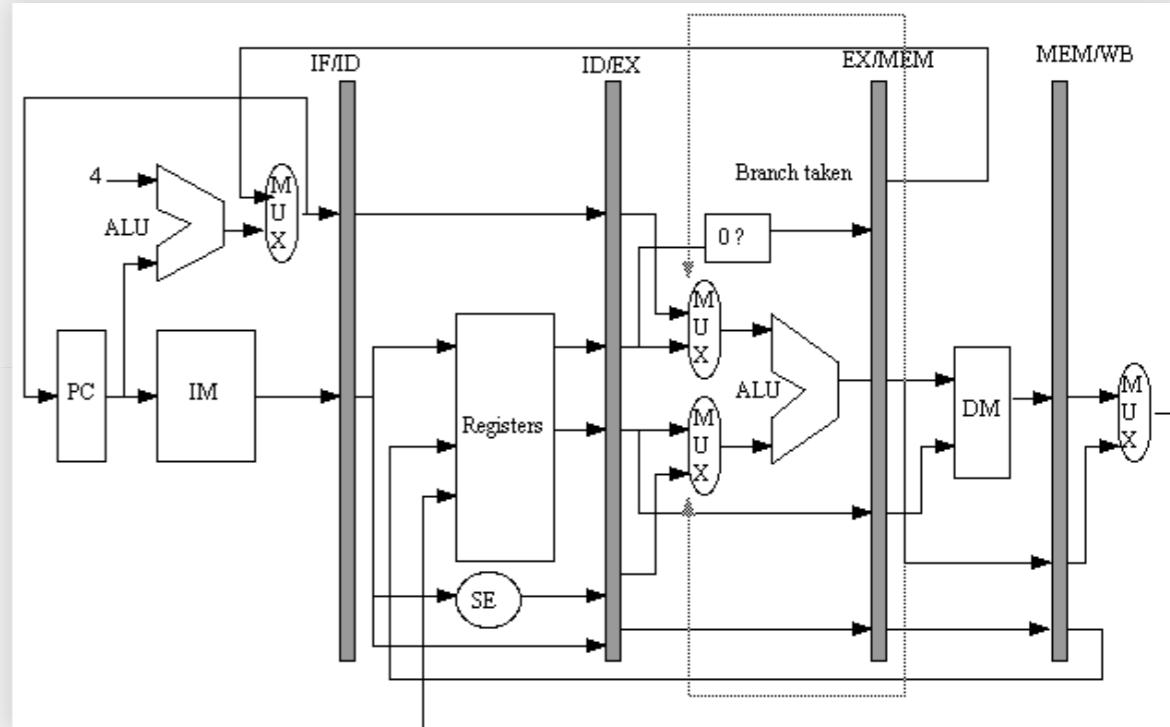
数据冒险

数据冒险(Data Hazard): 指令之间存在数据依赖

- 例: MIPS指令间存在读-写依赖(Read-after-Write, RAW)

```
ADD r0, r1, r2  
ADD r3, r0, r4
```

第二条ADD指令在读取寄存器r0的时候，寄存器r0的值还没有写入



图片来源: <https://user.eng.umd.edu/~yavuz/enee446/images-446/lecture446-10.htm>

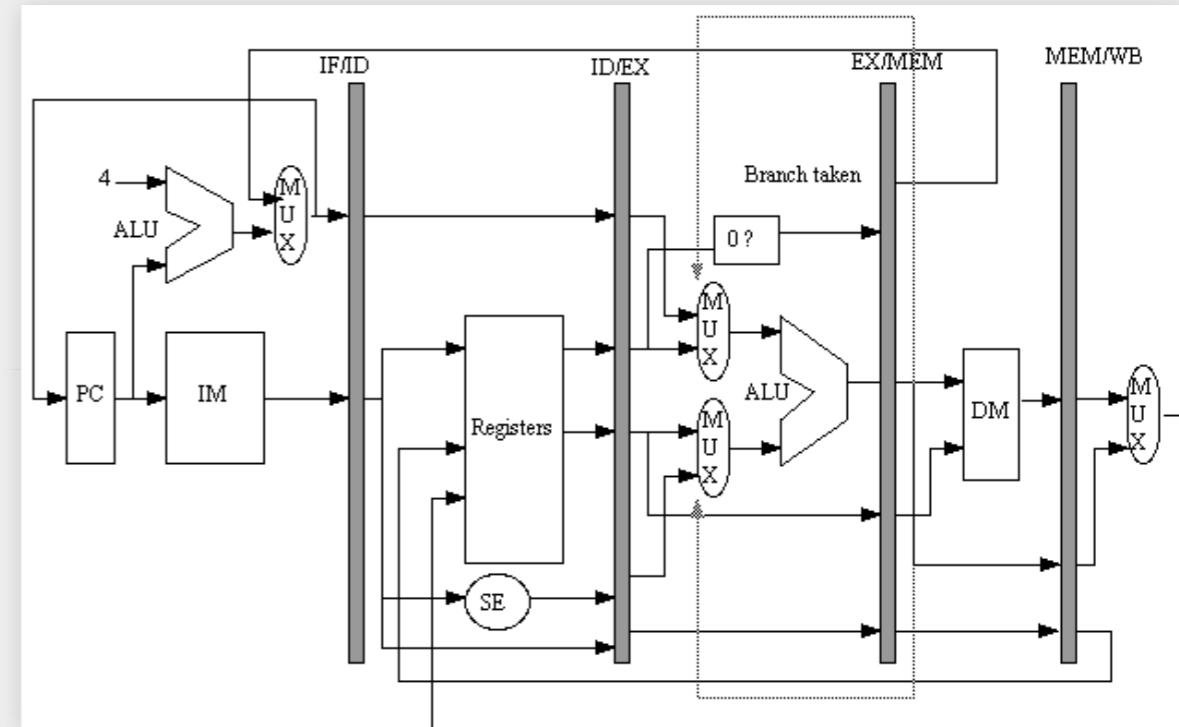
数据冒险

数据冒险(Data Hazard): 指令之间存在数据依赖

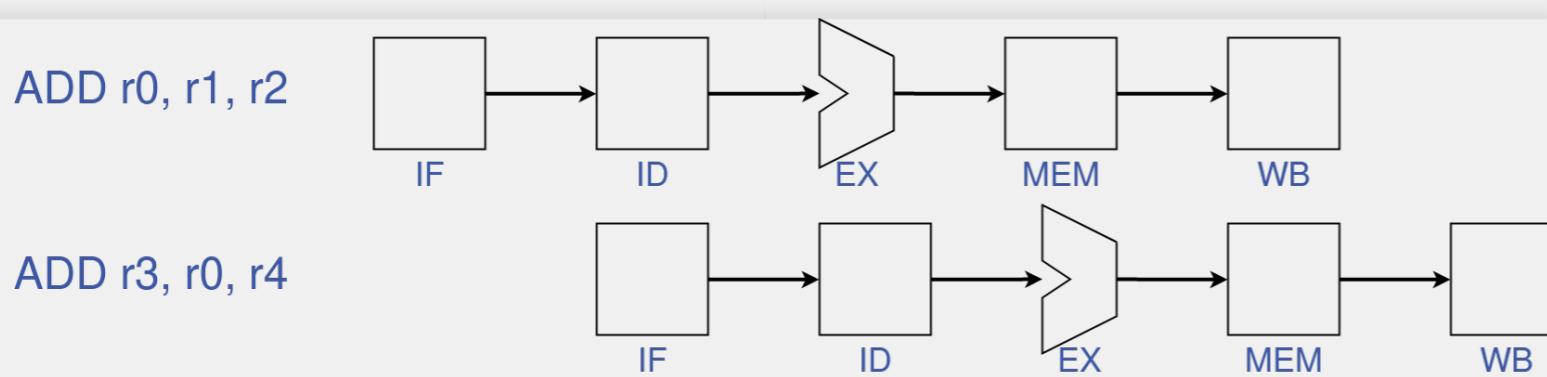
- 例: MIPS指令间存在读-写依赖(Read-after-Write, RAW)

```
ADD r0, r1, r2  
ADD r3, r0, r4
```

第二条ADD指令在读取寄存器r0的时候, 寄存器r0的值还没有写入



图片来源: <https://user.eng.umd.edu/~yavuz/enee446/images-446/lecture446-10.htm>



数据冒险

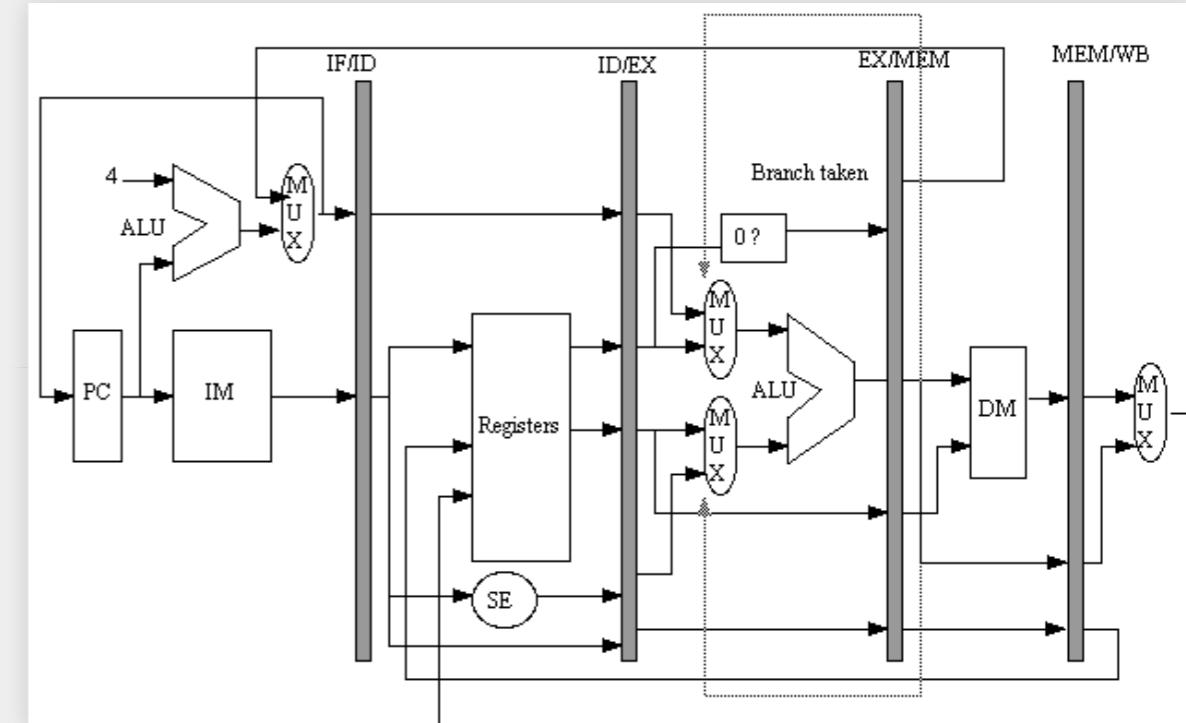
数据冒险(Data Hazard): 指令之间存在数据依赖

- 例: MIPS指令间存在读-写依赖(Read-after-Write, RAW)

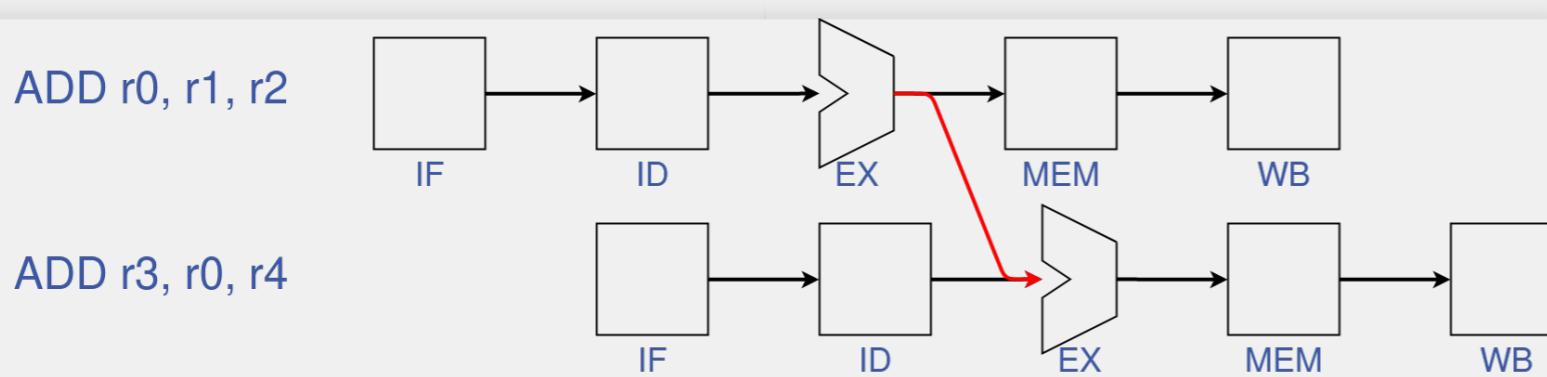
```
ADD r0, r1, r2  
ADD r3, r0, r4
```

第二条ADD指令在读取寄存器r0的时候, 寄存器r0的值还没有写入

- 解决方案: 数据定向(Data Forwarding)/数据旁路(Data Bypassing)



图片来源: <https://user.eng.umd.edu/~yavuz/enee446/images-446/lecture446-10.htm>



数据冒险(2)

加载-使用型数据冒险(Load-use data hazard):

```
LOAD r0, x  
ADD r2, r0, r1
```

数据冒险(2)

加载-使用型数据冒险(Load-use data hazard):

```
LOAD r0, x  
ADD r2, r0, r1
```

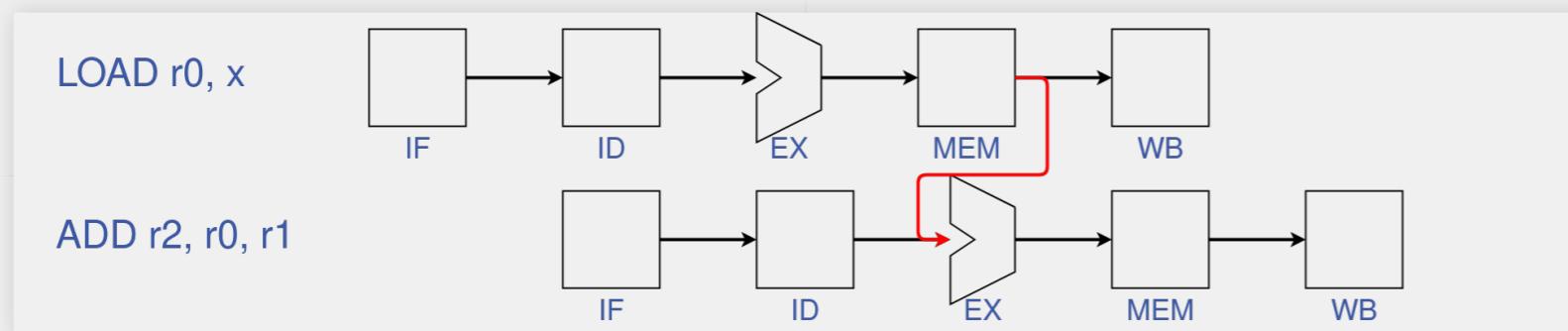
- 是否可以用数据旁路解决?

数据冒险(2)

加载-使用型数据冒险(Load-use data hazard):

```
LOAD r0, x  
ADD r2, r0, r1
```

- 是否可以用数据旁路解决? **不能, 计算所需的数据的值在计算阶段仍然未知**

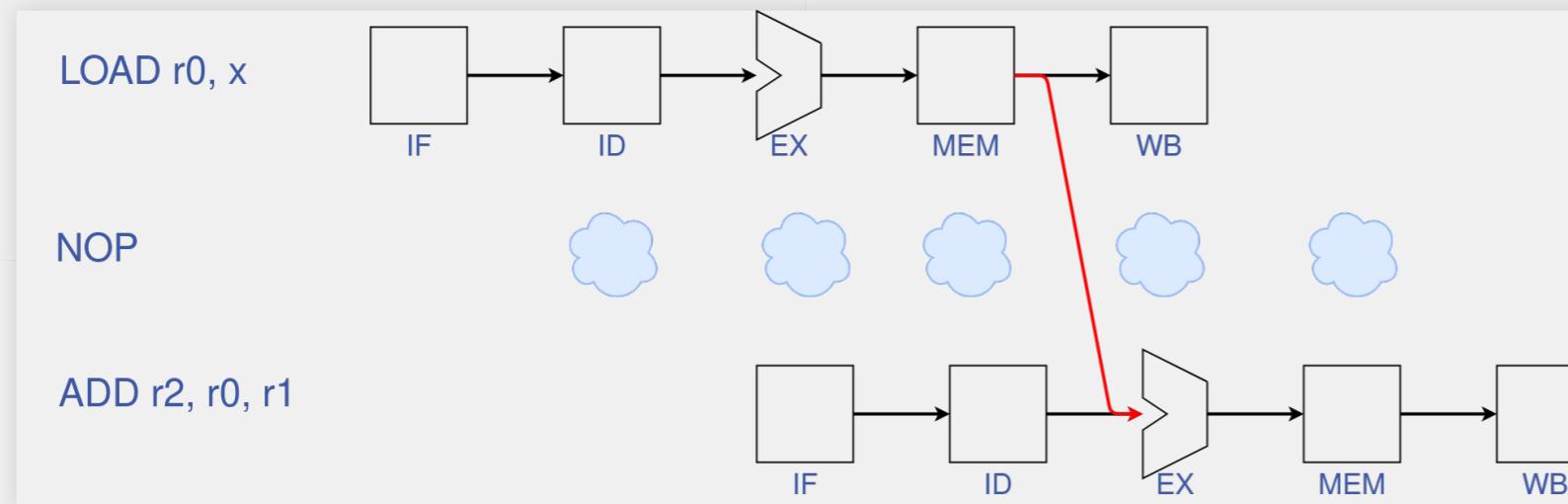


数据冒险(2)

加载-使用型数据冒险(Load-use data hazard):

```
LOAD r0, x  
ADD r2, r0, r1
```

- 是否可以用数据旁路解决? **不能, 计算所需的数据的值在计算阶段仍然未知**
- 流水线阻塞(Stalling): 插入一条NOP指令

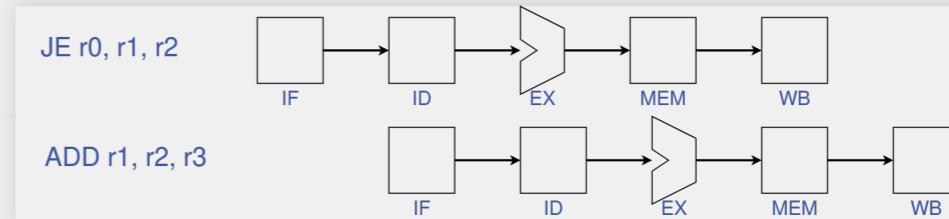


控制冒险

控制冒险(Control Hazard): 指令直接存在控制依赖，前一条指令的结果决定了后一条指令是否需要执行

- 主要由于分支指令造成，因此也被称为分支冒险 (Branch Hazard)

```
JE r0, r1, r2  
ADD r1, r2, r3  
...  
ADD r1, r2, r4 <- r0
```



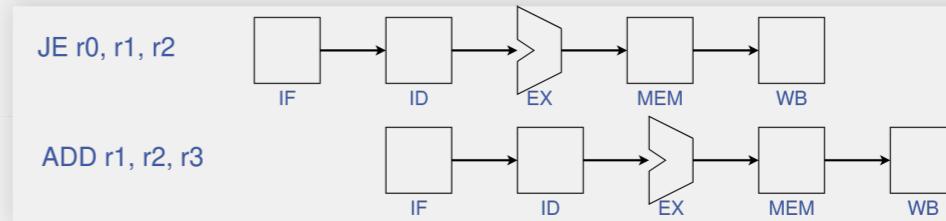
控制冒险

控制冒险(Control Hazard): 指令直接存在控制依赖，前一条指令的结果决定了后一条指令是否需要执行

- 主要由于分支指令造成，因此也被称为分支冒险 (Branch Hazard)

常见的解决方案

```
JE r0, r1, r2  
ADD r1, r2, r3  
...  
ADD r1, r2, r4 <- r0
```



控制冒险

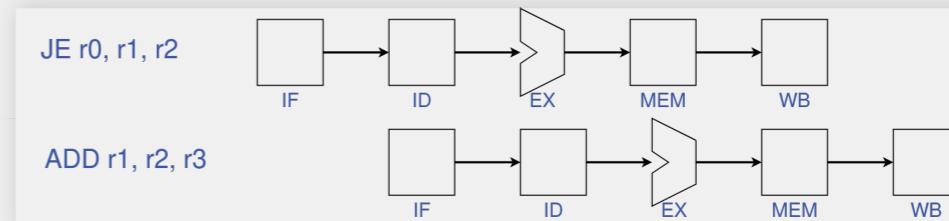
控制冒险(Control Hazard): 指令直接存在控制依赖，前一条指令的结果决定了后一条指令是否需要执行

- 主要由于分支指令造成，因此也被称为分支冒险 (Branch Hazard)

常见的解决方案

- 方案1：阻塞流水线，执行完分支指令后再进行取指令
 - 优点：简单
 - 缺点：不管是分支预测正确还是失败，性能都较低

```
JE r0, r1, r2  
ADD r1, r2, r3  
...  
ADD r1, r2, r4 <- r0
```



控制冒险

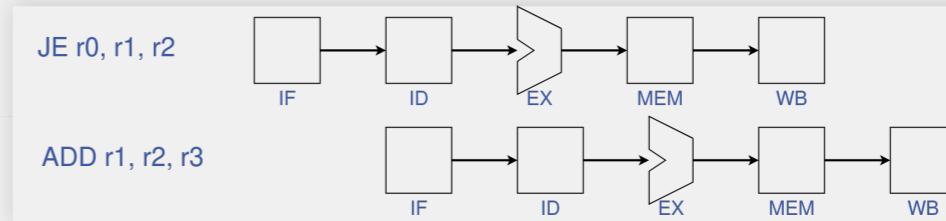
控制冒险(Control Hazard): 指令直接存在控制依赖，前一条指令的结果决定了后一条指令是否需要执行

- 主要由于分支指令造成，因此也被称为分支冒险 (Branch Hazard)

常见的解决方案

- 方案1：阻塞流水线，执行完分支指令后再进行取指令
 - 优点：简单
 - 缺点：不管是分支预测正确还是失败，性能都较低
- 方案2：预测分支的执行情况
 - 优点：如果预测正确，则流水线性能较高
 - 缺点：电路实现更复杂，如果预测失败则执行的指令无效

```
JE r0, r1, r2  
ADD r1, r2, r3  
...  
ADD r1, r2, r4 <- r0
```



分支预测

- 预测分支未执行：假设分支不会执行，正常载入下一条指令并执行
- 预测分支执行：假设分支会执行，**读取到跳转的指令地址后**载入指令并执行

```
JE r0, r1, r2  
ADD r1, r2, r3  
...  
ADD r1, r2, r4 <- r0
```

分支预测

- 预测分支未执行：假设分支不会执行，正常载入下一条指令并执行
- 预测分支执行：假设分支会执行，**读取到跳转的指令地址后**载入指令并执行

看似对称的这两种预测有什么区别？

```
JE r0, r1, r2  
ADD r1, r2, r3  
...  
ADD r1, r2, r4 <- r0
```

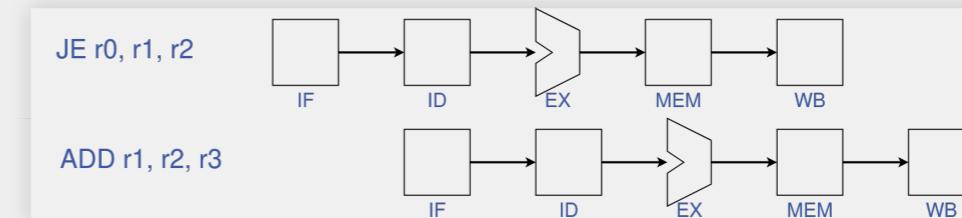
分支预测

- 预测分支未执行：假设分支不会执行，正常载入下一条指令并执行
- 预测分支执行：假设分支会执行，**读取到跳转的指令地址后**载入指令并执行

看似对称的这两种预测有什么区别？

- 预测分支未执行**不阻塞流水线**

```
JE r0, r1, r2  
ADD r1, r2, r3  
...  
ADD r1, r2, r4 <- r0
```



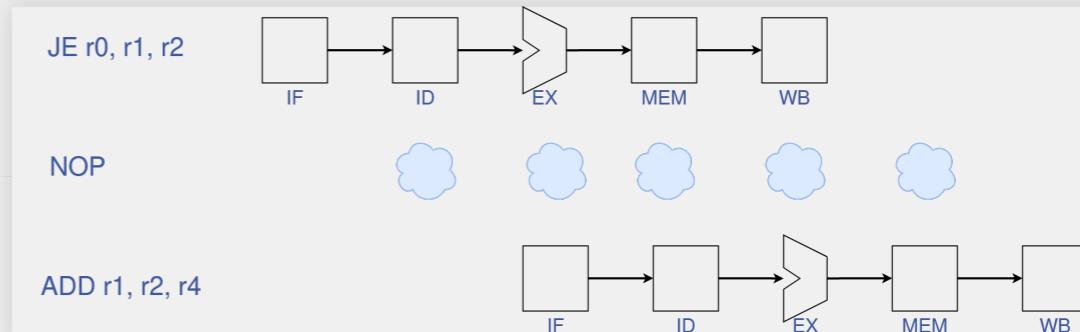
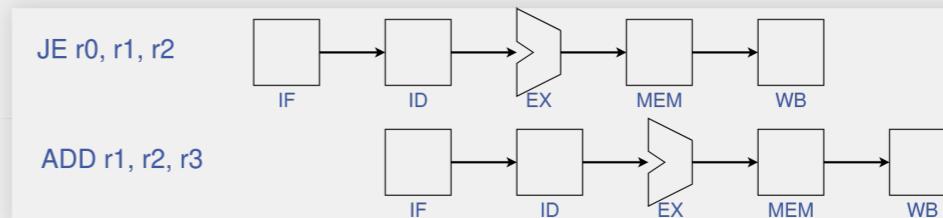
分支预测

- 预测分支未执行：假设分支不会执行，正常载入下一条指令并执行
- 预测分支执行：假设分支会执行，**读取到跳转的指令地址后**载入指令并执行

看似对称的这两种预测有什么区别？

- 预测分支未执行**不阻塞流水线**
- 预测分支执行**需要阻塞流水线**

```
JE r0, r1, r2  
ADD r1, r2, r3  
...  
ADD r1, r2, r4 <- r0
```



利用编译器优化流水线执行

利用编译器优化流水线执行

通过调度指令顺序避免流水线阻塞

```
LOAD r1, x  
LOAD r2, y  
ADD r4, r1, r2  
LOAD r3, z  
ADD r5, r4, r3
```

```
LOAD r1, x  
LOAD r2, y  
LOAD r3, z  
ADD r4, r1, r2  
ADD r5, r4, r3
```

加载-使用型数据冒险需
要插入一条NOP指令 读写型数据冒险不需要插
入流水线阻塞

超标量计算机

超标量(Superscalar)计算机又称为**动态多发射**(Dynamic Multiple Issue)，其核心思想是**允许一个流水线阶段同时执行多条指令**

类比：厨房每个步骤都请了两个师傅

现实中的指令集架构

- Intel
- MIPS
- Java
- LLVM
- ARM

- Intel在奔腾(Pentium)系列芯片引入了流水线
- Pentium I具有五级流水线，Pentium IV具有24级流水线
- Pentium采用8086指令集，支持17种内存寻址模式(向后兼容性)
- Itanium系列芯片具有10级流水线，支持IA-64指令集，只支持寄存器间接寻址

Basic Pentium® III Processor Misprediction Pipeline																			
1 Fetch	2 Fetch	3 Decode	4 Decode	5 Decode	6 Rename	7 ROB Rd	8 Rdy/Sch	9 Dispatch	10 Exec										
Basic Pentium® 4 Processor Misprediction Pipeline																			
1 TC Nxt IP	2 TC Fetch	3 Drive Alloc	4 Rename	5 Que	6 Sch	7 Sch	8 Disp	9 Disp	10 RF	11 RF	12 Ex	13 Flgs	14 Br Ck	15 Drive	16 Drive	17 Drive	18 Drive	19 Drive	20 Drive

<http://www.owchallie.com/systems/cache-pentium4.php>

MIPS

- MIPS采用三地址定长指令
- 采用取-存模式，只支持基址寻址
- R2000/R3000具有5级流水线，R4000/R4400具有8级流水线



MIPS R4000 Die Photo by Pauli Rautakorpi

ARM

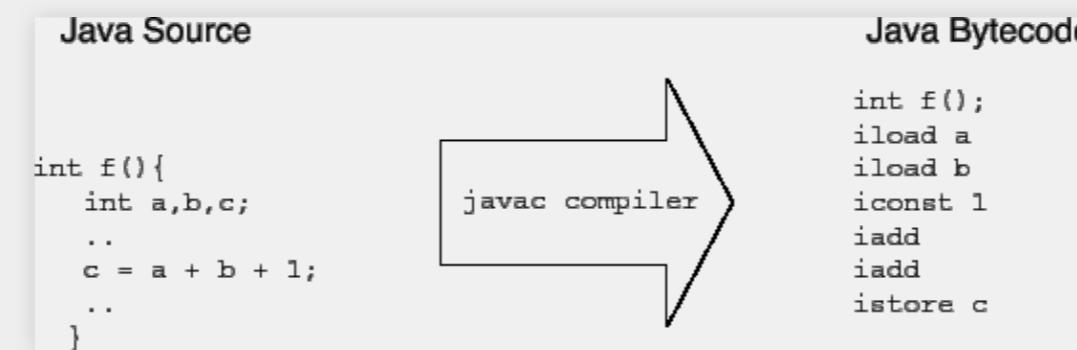
- ARM(Advanced RISC Machine)采用取-存模式
- 支持定长三地址指令
- 至少包含3级流水线，最多的实现具有13级流水线



<https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/5219/how-long-are-the-cortex-m7-pipeline-stages>

JAVA

- Java Bytecode是基于堆栈的指令集
- JVM有4个地址寄存器，可以访问5类地址
- 地址引用采用偏移值



<http://www.cs.toronto.edu/~matz/dissertation/matzDissertation-latex2html/node5.html>

LLVM

- LLVM(Low-level Virtual Machine)是一个通用的编译框架
- LLVM的IR(Intermediate Representation)类似Java Bytecode，提供了一种可移植的中间表示
- LLVM IR支持三地址指令，可使用无限个寄存器

```
; Function Attrs: noinline nounwind optnone uwtable
define i32 @sum(i32, i32) #0 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, i32* %3, align 4
    store i32 %1, i32* %4, align 4
    %5 = load i32, i32* %3, align 4
    %6 = load i32, i32* %4, align 4
    %7 = add nsw i32 %5, %6
    ret i32 %7
}
```



第六讲结束

本期内容回顾

- 指令流水线
 - 指令流水线的性能分析
 - 流水线冒险
 - 流水线对于指令集架构设计的影响
- 现实中的指令集架构

扩展阅读

- David A. Patterson, John L. Hennessy著, 计算机组成与设计: 硬件/软件接口, 原书第3版, 机械工业出版社



Q & A