高级计算机系统结构笔记

Mobyw

Created by ElegantLaTeX

版本:1.1

更新:2023 年 2 月 28 日

1 量化设计与分析基础

1.1 计算机的分类

Flynn's 分类:基于指令流和数据流数量的计算机结构分类

- SISD: 串行计算机
  - 任一时钟周期只有单个指令流在 CPU 执行,单个数据流用作输入
  - 确定执行: 给定输入条件下多次运行的执行流程和结果一致
- SIMD: 处理数据级并行
  - 任一时钟周期所有处理单元执行相同指令
  - 每个处理单元能对不同数据进行操作
- MISD: 处理数据级并行
  - 单个数据流进入多个处理单元
  - 每个处理单元的指令流对数据独立进行操作
- MIMD: 任务级并行
  - 每个处理器可以执行不同的指令流,也可以对不同数据流进行操作

Flynn 分类模型是抽象和粗略的,现代的不少并行处理器是 SISD、SIMD 和 MIMD 的混合类型.

市场分类

- 个人移动设备
- 桌面计算机
  - 个人计算机
  - 工作站

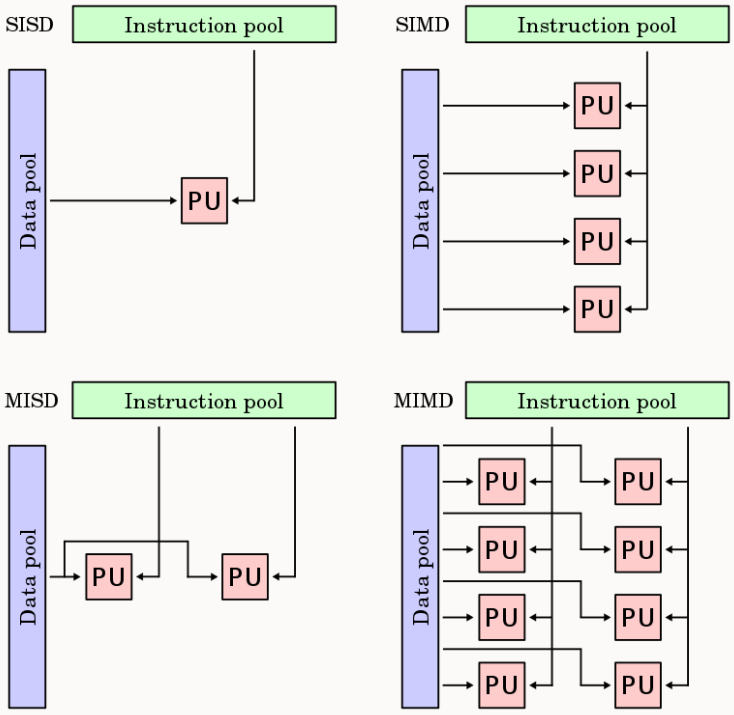


图 1: Flynn's Taxonomy

- 服务器：多处理器结构
  - 可靠性
  - 可扩展性
  - 吞吐量
- 集群/仓库级计算机
- 嵌入式计算机：实时性能
  - 实时性能
  - 软件固化

1.2 计算机系统结构定义

计算机系统结构：机器语言程序员所看到的传统机器级所具有的属性. 它确定计算机系统的软、硬件界面.

计算机组成：计算机系统结构的逻辑实现，包括五大功能部件组成以及逻辑设计等. 它着眼于机器级内各事件的排序方式与控制方式，各部件的功能以及各部件的联系.

计算机实现：计算机组成的物理实现，包括处理机、主存等部件的物理结构，器件的集成度和速度功耗，模块、插件、底板的划分与连接，信号传输，电源、冷却及整机装配技术等. 它着眼于器件技术和微组装技术，其中器件技术在实现技术中占主导作用.

计算机系统结构的现代定义：在满足功能、性能和价格目标的条件下，设计、选择和互连硬件部件构成计算机.

指令集结构（ISA）是硬件与软件之间的接口. 其特征有：

- ISA 的类型：现代通用寄存器结构、早期累加器结构
- 存储器访问：按字节访问
- 寻址方式
- 操作数类型和大小：8 位字符、32 位整型数
- 操作类型：数据传输、算术/逻辑
- 控制流指令：转移、子程序调用/返回
- ISA 编码：固定长度、可变长度

1.3 实现技术的趋势

摩尔定律（Moore's Law）：集成电路上可容纳的晶体管数目每隔两年就会增加一倍.

性能趋势：带宽改进优先于时延.

经验法则：成本减少速度与密度增加速度成比例，带宽增加速度与时延平方改进速度成比例.

1.4 集成电路功耗的趋势

经验法则：电压减少 10%，功耗减少 30%，性能减少 < 10%.

单核与双核：单核性能与面积增加倍数的一半成正比；多核有更高的功率利用率.

1.5 可靠性

提高可靠性的方法：冗余

- 时间冗余：重复操作直到无错
- 资源冗余：配置另外的相同部件，有错时用于替代出错部件

1.6 计算机主要性能指标

响应时间（墙钟时间）：从用户发出请求到得到响应的时间，包括等待 I/O 的时间. 是用户感觉到的系统速度.

CPU 时间：从程序开始执行到程序结束的时间，不包括等待 I/O 的时间. 测量设计者感觉到的 CPU 速度.

- 用户 CPU 时间：花费在用户模式的时间
- 系统 CPU 时间：花费在内核模式的时间

吞吐量：单位时间内处理的任务数.

处理器用更快的型号替换可以改善响应时间和吞吐量；增加额外的处理器可以改善吞吐量.

工业性能指标 MIPS：每秒百万条指令.

总执行时间：n 个测试程序的执行时间 T 的算术平均值为 $\frac{1}{n}\sum_{i=1}^{n} T_i$；如果性能用 Rate R（如 MIPS）表示，那么总的执行时间平均值就是调和平均值 $\frac{n}{\sum_{i=1}^{n}\frac{1}{R_i}}$.

通常，机器运行某些程序更频繁，则应该给这些程序更大的权值 W，算术平均值为 $\frac{1}{n}\sum_{i=1}^{n} W_i \times T_i$；调和平均值为 $\frac{n}{\sum_{i=1}^{n}\frac{W_i}{R_i}}$. 通常，$W_i$ 为程序的执行频率.

实用基准测试程序集 SPEC：以 CPU 时间为基准，以不同的工作负载为测试对象.

SPEC 率（SPEC Ratio）是一个测试程序在参考计算机上的执行时间与在被测计算机上的执行时间的比值，SPEC Ratio 越高，说明被测计算机的性能越好.

$$\text{SPEC Ratio} = \frac{t_{ref}}{t_{test}} = \frac{\text{参考计算机的执行时间}}{\text{被测计算机的执行时间}}$$

被测试计算机的性能用执行 n 个基准测试程序分别得到的 SPEC Ratio 的几何平均值来衡量（PPT 中表示为 SM）：

$$\sqrt[n]{\prod_{i=1}^{n} \text{SPEC Ratio}_i}$$

应用程序	Opteron SPEC Ratio	Itanium2 SPEC Ratio	Itanium/Opteron
mesa	21.67	12.99	0.6
applu	22.34	41.25	1.85
lucas	22.52	18.76	0.83
...	...	...	...
几何平均值	20.86	27.12	1.3

例：

假如你的公司需要选购是 Opteron 还是 Itanium2. 公司的应用情况是：50% 的时间运行类似于 mesa 的应用程序，25% 的时间运行类似于 applu 应用程序，25% 的时间运行类似于 lucas 的应用程序. 上表提供了 Opteron 和 Itanium 的 SPEC Ratio 信息.

（1）如果仅根据SPEC总体性能进行选择，你选择哪一种微处理器？为什么？

（2）计算公司混合应用程序的 Itanium/Opteron SPEC Ratio 加权平均值是多少？按照这个结果应该选择哪一种微处理器？

解：

（1）

选择 Itanium2，因为其几何平均值的值是 Opteron 的 1.3 倍，表明它的总的性能更好.

（2）

$0.6 \times 50\% + 1.85 \times 25\% + 0.83 \times 25\% = 0.97$，表明针对公司的应用程序情况，选择 Opteron 有更好的性能.

1.7 计算机设计的量化原则

Amdahl 定律：采用更快的执行方式后所获得的系统性能提高，与这种执行方式的使用频率或占总执行时间的比例有关.

Amdahl 定律定义了一台计算机采用某种改进措施所取得的系统加速比：

$$\text{加速比} = \frac{\text{改进后的计算机性能}}{\text{改进后的计算机性能}} = \frac{\text{改进前的执行时间}}{\text{改进后的执行时间}}$$

改进比例：

$$F_e = \frac{\text{改进部分的执行时间}}{\text{改进前的执行时间}}$$

改进加速比：

$$S_e = \frac{\text{改进部分改进前的执行时间}}{\text{改进部分改进后的执行时间}}$$

设改进后执行时间为 T_e，改进前的执行时间为 T_0，则：

$$T_e = T_0 \times (1 - F_e) + \frac{T_0 F_e}{S_e} = T_0 \left(1 - F_e + \frac{F_e}{S_e}\right)$$

改进后整个系统的加速比为：

$$S_n = \frac{T_0}{T_e} = \frac{T_0}{T_0\left(1 - F_e + \frac{F_e}{S_e}\right)} = \frac{1}{1 - F_e + \frac{F_e}{S_e}}$$

提高改进比例 F_e 或改进加速比 S_e 可以提高系统加速比 S_n. 但是 F_e 对 S_n 的影响更大.

例：

设一台计算机运行某程序的 CPU 时间如下：

- 浮点指令：60 s
- 整数指令：100 s
- 读写指令：40 s
- 分支指令：40 s
- 总时间：240 s

（1）如浮点指令执行时间减少50%，总时间减少百分之多少？减少后加速比为多少？

（2）如总时间减少15%，只减少整数指令时间，整数指令时间减少百分之多少？

（3）如只减少分支指令时间，总时间能否减少20%？

解：

（1）

浮点指令减少 50% 时间：$60 \times 50\% = 30$ s

总时间减少：$30/240 = 12.5\%$；

加速比：$240/210 = 1.143$

（2）

总时间减少 15% 时间：$240 \times 15\% = 36$ s

减少的整数时间比例：$36/100 = 36\%$

（3）

不能，因为分支指令时间减少后，总时间减少的比例不够.

2 指令系统原理与示例

2.1 指令集系统结构的分类

不同指令集系统结构最根本的区别在于处理器内部数据的存储结构不同.

存储结构：堆栈、累加器或一组寄存器. 操作数可以显式指定或者隐含指定.

- **堆栈结构：**操作数隐含地位于栈顶
- **累加器结构：**操作数隐含地位于累加器

- **通用寄存器结构：**只能明确地指定操作数（寄存器/存储器地址）

通用寄存器按照访问方式划分，主要有以下两种系统结构：

- **寄存器-存储器结构：**一般指令都可以访问存储器
- **寄存器-寄存器结构：**只能通过 load 和 store 指令访问存储器

2.2 存储器寻址

大小端模式：存储器中的数据在存储器中的存放顺序.

- **大端模式：**高位字节存放在低地址
- **小端模式：**低位字节存放在低地址

地址对齐：地址对齐是为了提高存储器的访问效率，使得存储器的访问速度与存储器的容量成正比.

假设一个 s 字节数据的地址是 A，如果 A 是 s 的整数倍，那么这个数据就是地址对齐的.

2.3 MIPS 指令集结构

MIPS 系统结构是一种寄存器-寄存器结构（load-store 结构），指令集结构是 RISC 结构.

- 固定长度指令编码：32 位
- 32 个 64 位通用寄存器：R0-R31，其中 R0 的值永远是 0
- 32 个浮点寄存器：F0-F31
- 定点数据类型有 8 位字节、16 位半字、32 位字和 64 位双字
- 浮点数有 32 位单精度和 64 位双精度

MIPS 的指令格式：

- I 类型：操作码（OP）+源寄存器（RS）+目的寄存器（RT）+立即数/偏移量（IMM）
- R 类型：操作码（OP）+源寄存器（RS）+源寄存器（RT）+目的寄存器（RD）+移位量（SHAMT）+功能码（FUNCT）
- J 类型：操作码（OP）+目的地址（ADDR）

MIPS 操作：载入和存储、ALU操作、分支与跳转、浮点操作.

3 单周期 MIPS 处理器的设计

3.1 单周期实现

单周期 CPU 设计：将每条指令的执行周期设为一个时钟周期，时钟周期需要满足所有指令的执行时间.

要实现的 MIPS 的指令子集（9 种）：

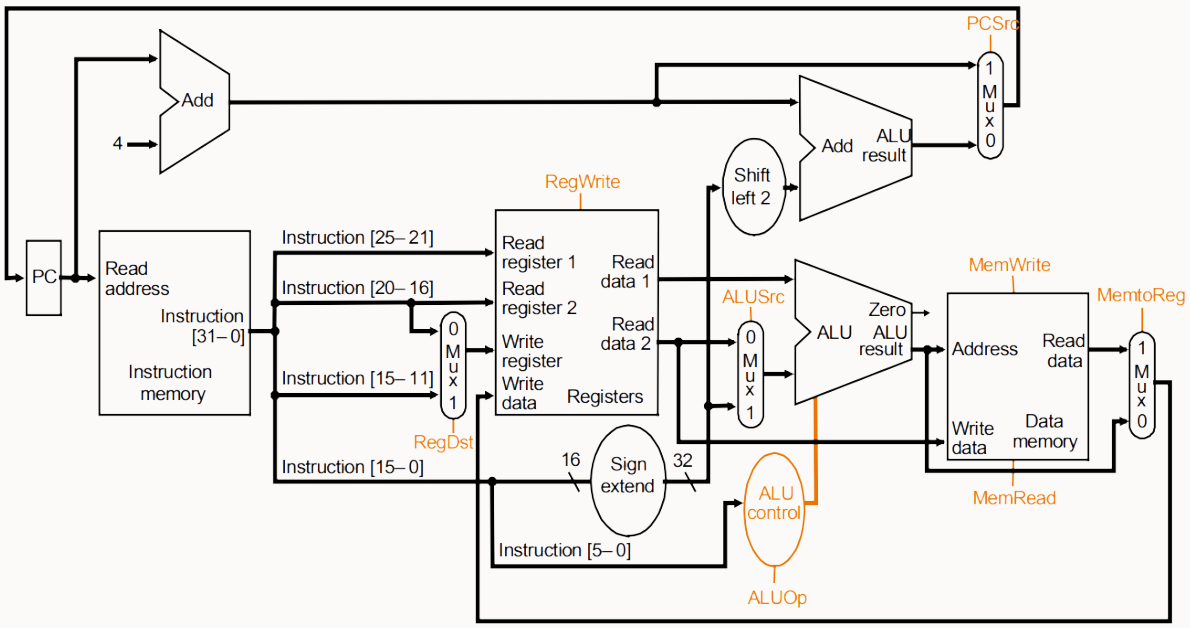
- R 型指令：`add`、`sub`、`and`、`or`、`slt`、`beq`
- I 型指令：`lw`、`sw`
- J 型指令：`j`

以上指令集按功能可归纳为：

- 存储器访问指令（I 型指令）：`lw`、`sw`
- 算术逻辑指令（R 型指令）：`add`、`sub`、`and`、`or`、`slt`
- 分支指令（R 型指令）：`beq`
- 跳转指令（J 型指令）：`j`

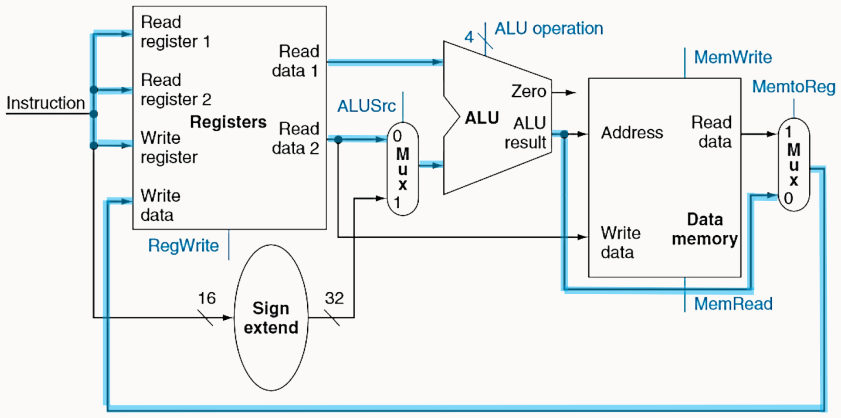
使用该指令子集可以说明在建立数据通路和控制部件时的关键原理（该指令集没有包含乘、除和移位指令、浮点指令）.

构建数据通路和控制部件：

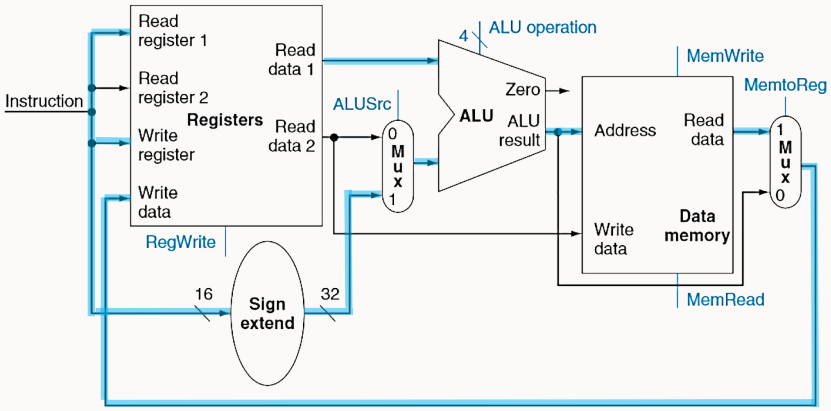


3.2 数据通路

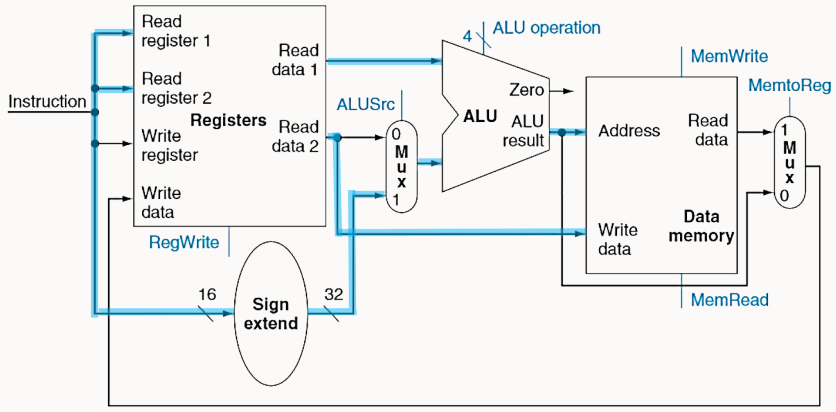
R 型指令的数据通路：



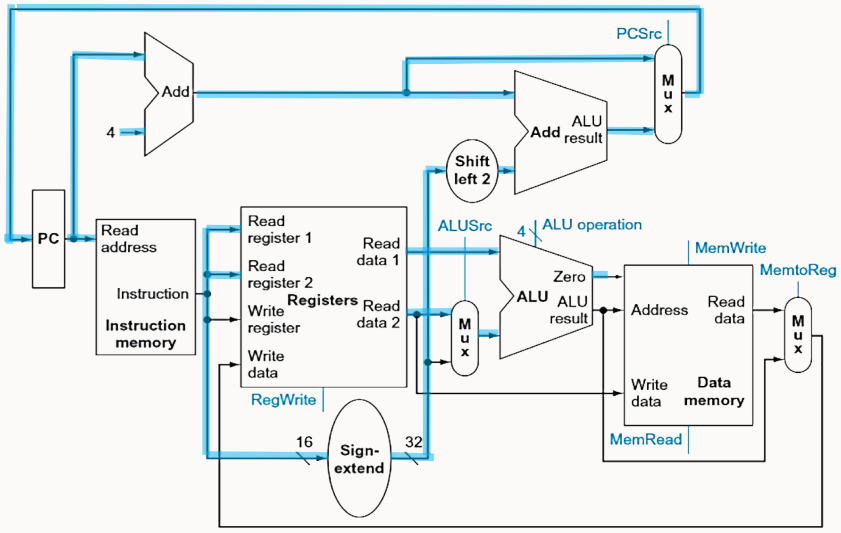
`lw` 指令的数据通路：



`sw` 指令的数据通路：



`beq` 指令的数据通路：



3.3 控制信号

各指令的控制信号：

	RegDst	ALUSrc	Mem2Reg	RegWr	MemRd	MemWr	Branch	ALUOp
R	1	0	0	1	0	0	0	10
lw	0	1	1	1	1	0	0	00
sw	X	1	X	0	0	1	0	00
beq	X	0	X	0	0	0	1	01

有如下特征：

- `sw` 和 `beq` 指令不写任何寄存器
- `lw` 和 `sw` 指令使用常数字段，它们还依赖 ALU 计算有效内存地址
- R 类型指令的 ALUOp 依赖于指令的 func 字段
- `beq` 指令的 PCSrc 控制信号（未列出）为 1，ALU 有 Zero 输出

4 流水线技术及指令级并行

4.1 流水线的概念

流水线技术：将指令的执行周期分为多个时钟周期，每个时钟周期执行一部分指令.

流水线的分类：

- 按功能分类：单功能流水线、多功能流水线
- 按工作方式分类：静态流水线、动态流水线

4.2 流水线的时空图及性能分析

时空图：时空图从时间和空间两个方面描述了流水线的工作过程. 时空图中，横坐标代表时间，纵坐标代表流水线的各个段（或级，即完成一条指令的一部分操作）.

吞吐率：在单位时间内流水线所完成的任务数量或输出结果的数量.

$$\mathrm{TP} = \frac{n}{T}$$

各段时间均相等的 k 段线性流水线的实际吞吐率：

$$\mathrm{TP} = \frac{n}{T} = \frac{n}{(k+n-1)\Delta t}$$

最大吞吐率：

$$\mathrm{TP}_{\max} = \frac{1}{\Delta t}$$

各段时间不完全相等的流水线的最大吞吐率：

$$\mathrm{TP}_{\max} = \frac{1}{\max(\Delta t_i)}$$

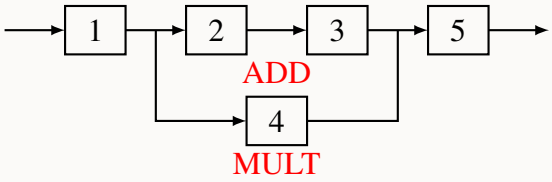
解决流水线瓶颈问题的常用方法：细分瓶颈段、重复设置瓶颈段.

衡量流水线性能的主要指标有：吞吐率、加速比、效率.

多功能流水线的时空图：静态/动态.

例：

计算 $\sum_{i=1}^{4} a_i b_i$，画出下图所示多功能流水线在静态和动态下的时空图，并分析其性能.



解：

计算过程分解为 4 次乘法和 3 次加法：

$$①\mathrm{MUL}_1 : a_1 \times b_1$$

$$②\mathrm{MUL}_2 : a_2 \times b_2$$

$$③\mathrm{MUL}_3 : a_3 \times b_3$$

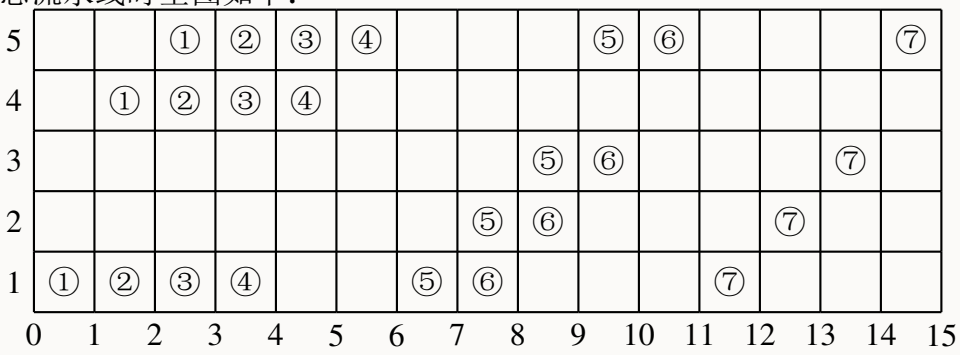
$$④\mathrm{MUL}_4 : a_4 \times b_4$$

$$⑤\mathrm{ADD}_1 : \mathrm{MUL}_1 + \mathrm{MUL}_2$$

$$⑥\mathrm{ADD}_2 : \mathrm{MUL}_3 + \mathrm{MUL}_4$$

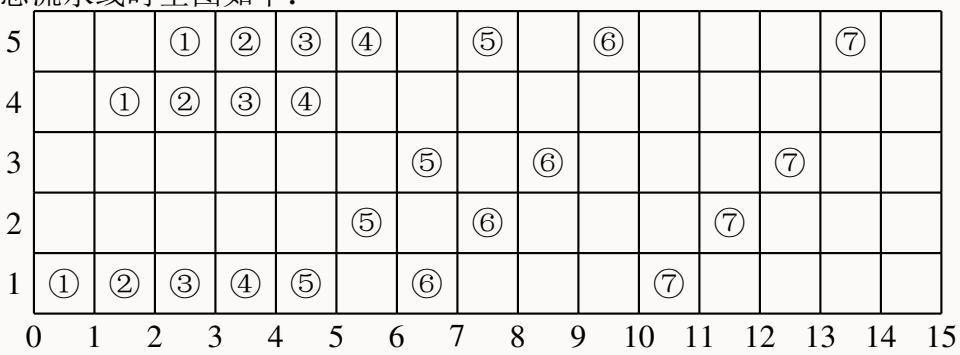
$$⑦\mathrm{ADD}_3 : \mathrm{ADD}_1 + \mathrm{ADD}_2$$

静态流水线时空图如下：



吞吐率 $\mathrm{TP} = \dfrac{7}{15\Delta T}$；效率 $\mathrm{EFF} = \dfrac{3\times 4 + 4\times 3}{5\times 15}$；加速比 $\mathrm{SP} = \dfrac{7\times 5}{15}$.

动态流水线时空图如下：



吞吐率 $\mathrm{TP} = \dfrac{7}{14\Delta T}$；效率 $\mathrm{EFF} = \dfrac{3\times 4 + 4\times 3}{5\times 14}$；加速比 $\mathrm{SP} = \dfrac{7\times 5}{14}$.

4.3 流水线中的冒险问题

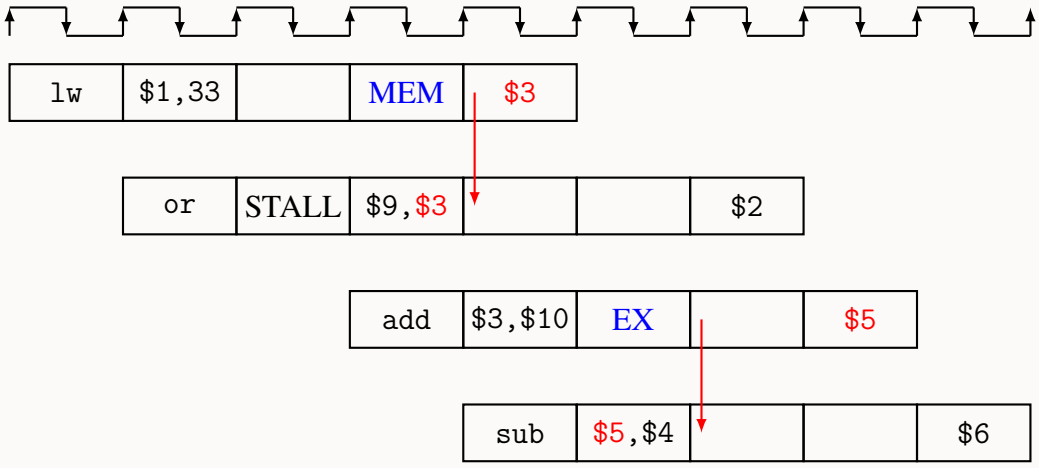
数据冒险（Data Hazard）：指令的操作数尚未准备好，而该指令已经进入流水线的下一阶段而导致的冒险.

结构冒险（Structural Hazard）：某些指令组合在流水线中重叠执行时产生了资源冲突而导致的冒险.

控制冒险（Control Hazard）：因为程序的执行方向可能被改变而存在的冒险.

数据冒险的解决：

数据前推：将数据从流水线的前一阶段传递到后一阶段. `lw` 数据冒险需要额外的暂停周期.



控制冒险的解决：提前计算分支目标地址、提前判断分支条件，以及提前预测分支的执行方向.

在 ID 级中已形成了PC值和地址偏移量，因此将分支地址计算从 EX 级前移到 ID 级即可. 分支预测错误的代价减小到 1 条指令.

- R 型指令 + `beq` 指令：需暂停 1 个时钟周期
- `lw` + `beq` 指令：需暂停 2 个时钟周期

分支指令 `beq` 与其前面 R 型指令的数据冒险处理：

- 无数据冒险时：IF.Flush 冲掉 IF/ID 寄存器，预取到的指令变成 `nop`
- 有数据冒险时：保持 PC 和 IF/ID 不变，指令编码不变，并将该指令的所有写信号全部关闭

4.4 流水线中异常的处理

停止和重新开始执行的步骤：

- 强制一个 trap 指令进入流水线
- 关掉出错指令及后面指令的所有写操作，直到 trap 指令开始执行
- 当 trap 指令开始执行，唤醒 OS ，OS 保存出错指令的 PC 值
- OS 试图修复异常，然后重新执行出错指令

异常在 MIPS 体系结构中的处理：

- 异常发生时：在异常程序计数器（EPC）中保存出错指令的地址，并把控制权转交给 OS
- 在完成处理异常所需动作后，OS 可以终止程序，也可以继续执行程序，此时由 EPC 决定重新开始执行的地方

4.5 动态调度算法

见记分牌算法与 Tomasulo 算法专题.

5 存储系统

5.1 存储器

存储器的分类与参数：

- 半导体存储器：容量、存储周期
  - 静态随机存储器（SRAM）
  - 动态随机存储器（DRAM）
  - 闪存（Flash Memory）
- 磁表面存储器：容量、转速
- 光盘存储器
  - 只读性光盘（CD-ROM）
  - 写入式光盘（WORM）
  - 可擦写光盘（CD-RW）

存储器的存取方式：

- 随机存取存储器（RAM）

- 顺序存取存储器（SAM）

存储器的主要性能指标：

- 存储容量
- 存取周期

5.2 Cache 基本原理

Cache 是按**块（block）**进行管理的. Cache 和主存均被分割成大小相同的块. 对 Cache 和主存的访问是以块为基本单位.

5.2.1 映像规则

直接映像：块只能放在 Cache 中唯一的位置.

全相联：块可以放在 Cache 中的任意位置.

组相联：块能够放在 Cache 一组中任意一块位置. 如果一组有 n 块，则称为 n 路组相联.

5.2.2 块的识别

物理地址的格式：**标识 Tag + 索引 Index + 字节位移量 Byte Offset**.

标识 Tag：用于查找在 Cache 或一组中的匹配块.

索引 Index：选择块（直接映像 Cache）或组（组相联 Cache）. 位数是 $\log_2$(块数/组数).

字节位移量 Byte Offset：选择块中的某个字节. 位数是 $\log_2$(块字节数).

例：
数据 Cache 容量为 16KB，块大小为 32B，最小寻址单位 1 字节，采用两路组相联映像方式，Cache 的物理地址为 36 位. 计算索引、标识、块内偏移量的位数.

解：
块内偏移：$32 = 2^5$，占 5 位（低 5 位）.
索引：$\dfrac{\text{Cache 容量}}{\text{组数} \times \text{块大小}} = \dfrac{16 \times 1024}{2 \times 32} = 2^8$，占 8 位（中间 8 位）.
标识：$36 - 8 - 5 = 23$，占 23 位（高23位）.

5.2.3 块的替换策略

随机替换：

- 随机选择一组中的一块作为被替换块
- 硬件容易实现，需要随机数产生器
- 均匀使用一组中的各块
- 可能替换即将被访问的那一块

最近最少使用（LRU）：

- 选择一组中最近最少被访问的块作为被替换块
- 假定最近被访问的块很可能会一再访问
- 需要额外的位记录访问历史

先进先出（FIFO）：

- 选择一组中最先进入 Cache 的一块

例：
Cache-主存存储层次中，主存有 0-15 共 16 块，Cache 为 8 块，块号为 C0-C7，采用 2 路组相联映像. 设 Cache 初始为空，现访存块地址流为 1, 1, 2, 4, 13, 3, 9, 7, 3, 13 时：
（1）分别画出用 LRU 替换算法和 FIFO 替换算法，Cache 内各块的实际替换过程图，并标出命中时刻.
（2）计算命中率.
解：
（1）见下页表
（2）3/10 = 0.3，3/10 = 0.3.

5.2.4 块的写策略

写直达策略：写 Cache 的同时也写主存，读缺失不会导致替换时的写操作，保持了数据一致性，实现简单.

- Cache 中的数据可以随时丢弃
- 只需要 valid 控制位
- 主存（或其他处理器）总是有最新的数据

写回策略：写 Cache 时不写主存，速度更快，主存带宽更低.

- 不能丢弃 Cache 中的数据

Cache 块	主存块地址流									
	1	1	2	4	13	3	9	7	3	13
C0				4*	4*	4*	4*	4*	4*	4*
C1										
C2	1*	1*	1*	1*	1*	1*	9	9	9	9*
C3					13	13	13*	13*	13*	13
C4			2*	2*	2*	2*	2*	2*	2*	2*
C5										
C6						3*	3*	3*	3	3
C7								7	7*	7*
命中		H							H	H

Cache 块	主存块地址流									
	1	1	2	4	13	3	9	7	3	13
C0				4*	4*	4*	4*	4*	4*	4*
C1										
C2	1*	1*	1*	1*	1*	1*	9	9	9	9
C3					13	13	13*	13*	13*	13*
C4			2*	2*	2*	2*	2*	2*	2*	2*
C5										
C6						3*	3*	3*	3*	3*
C7								7	7	7
命中		H							H	H

- 需要 valid 位和 dirty 位
- 对同一块的多次写仅需要对主存写一次

按写分配：

- 写失效时，把所写单元所在的块调入 Cache，然后再进行写命中操作
- 与读失效类似
- 写回 Cache 通常采用

不按写分配：

- 写失效时，直接将值写入下一级存储器而不将相应的块调入 Cache
- 写的值不在 Cache 中
- 写直达 Cache 通常采用

例：

假设有一个全相联映射有多个项的 Cache，采用写回策略，刚开始时 Cache 为空. 有下面 5 个存储器操作：

```
write Mem[100];
write Mem[100];
read Mem[200];
write Mem[200];
write Mem[100];
```

分别求按写分配和不按写分配情况下的命中次数和缺失次数.

解：

按写分配：

- `write Mem[100];`:Cache 缺失，调入 Cache.
- `write Mem[100];`:Cache 命中.
- `read Mem[200];`:Cache 缺失，调入 Cache.
- `write Mem[200];`:Cache 命中.
- `write Mem[100];`:Cache 命中.

不按写分配：

- `write Mem[100];`:Cache 缺失.
- `write Mem[100];`:Cache 缺失.
- `read Mem[200];`:Cache 缺失，调入 Cache.
- `write Mem[200];`:Cache 命中.
- `write Mem[100];`:Cache 缺失.

5.3 Cache 的性能分析

5.3.1 Cache 的命中率

平均访存时间：

$$
\begin{aligned}
\text{AMAT} &= \text{命中时间} + \text{失效率} \times \text{失效开销} \\
&= \text{访问指令所占的百分比} \times (\text{指令命中时间} + \text{指令失效率} \times \text{失效开销}) \\
&\quad + \text{访问数据所占的百分比} \times (\text{数据命中时间} + \text{数据失效率} \times \text{失效开销})
\end{aligned}
$$

例：

假设 Cache 的命中时间为一个时钟周期，失效开销为 50 个时钟周期，在混合 Cache 中一次 LOAD 或 STORE 操作访问 Cache 的命中时间都要增加一个时钟周期. 又假设采用写直达策略，且有一个写缓冲器，并且忽略写缓冲器引起的等待. 约 75% 的访存为取指令，则剩下 25% 的访存是取数据. 请问指令 Cache 和数据 Cache 容量均为 16 KB 的分离 Cache 和容量为 32 KB 的混合 Cache 的平均访存时间各是多少？

解：

$$\begin{aligned}
\text{平均访存时间}_{\text{分离}} &= 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50) \\
&= (75\% \times 1.32) + (25\% \times 4.325) = 0.990 + 1.059 = 2.05 \\
\text{平均访存时间}_{\text{混合}} &= 75\% \times (1 + 1.99\% \times 50) + 25\% \times (1 + 1 + 1.99\% \times 50) \\
&= (75\% \times 1.995) + (25\% \times 2.995) = 1.496 + 0.749 = 2.24
\end{aligned}$$

CPU 执行时间：

$$\begin{aligned}
\text{CPU时间} &= \text{IC} \times \left(\text{CPI}_{\text{EX}} + \frac{\text{访存次数}}{\text{指令数}} \times \text{失效率} \times \text{失效开销周期} \right) \times \text{时钟周期} \\
&= \text{IC} \times \left(\text{CPI}_{\text{EX}} \times \text{时钟周期} + \frac{\text{访存次数}}{\text{指令数}} \times \text{失效率} \times \text{失效开销时间} \right)
\end{aligned}$$

例：

给定以下的假设，试计算直接映象 Cache 和两路组相联 Cache 的平均访问时间以及 CPU 的性能.

（1）理想 Cache 情况下的 CPI 为 2.0，时钟周期为 2 ns，平均每条指令访存 1.2 次.

（2）两者 Cache 容量均为 64KB，块大小都是 32B.

（3）两路组相联 Cache 中的多路选择器使 CPU 的时钟周期增加了 12%.

（4）这两种 Cache 的失效开销都是 80 ns.

（5）命中时间为 1 个时钟周期.

（6）直接映象 Cache 的失效率为 1.4%，两路组相联 Cache 的失效率为 1.0%.

解：

$$\begin{aligned}
\text{平均访存时间}_{\text{直接}} &= 2 + 1.4\% \times 80 = 3.12 \\
\text{平均访存时间}_{\text{两路组相联}} &= 2 \times (1 + 12\%) + 1.0\% \times 80 = 3.04 \\
\text{CPU时间}_{\text{直接}} &= \text{IC} \times (2.0 \times 2 + 1.2 \times 1.4\% \times 80) = 5.344\text{IC} \\
\text{CPU时间}_{\text{两路组相联}} &= \text{IC} \times (2.0 \times 2 \times (1 + 12\%) + 1.2 \times 1.0\% \times 80) = 5.44\text{IC}
\end{aligned}$$

5.4 降低 Cache 缺失率

Cache 失效分为 3 类：强制性失效、容量失效、冲突失效.

- 减少冲突失效，可以增加组的数目，全相联不会发生冲突失效
- 减小容量失效，可以增大 Cache 的容量
- 减少强制性失效，可以增大 Cache 块的容量

许多降低失效率的方法会增加命中时间或失效开销，降低失效率的方法：

- 增大 Cache 块容量：减少了强制性失效，但增加了容量失效
- 提高相联度：减少了冲突失效，但增加了命中时间
- 增大 Cache 容量：减少了容量失效和冲突失效
- 编译器优化：合并数组、循环交换、循环融合

减小 Cache 失效开销：采用两级 Cache、关键字优先和提前重启动、让读失效优先于写、牺牲缓存、非阻塞 Cache、预取技术.

5.5 快表 TLB

虚拟地址到物理地址的转换需要 2 次访存，一次是访问页表取得物理地址，一次是访问物理内存取得数据.

快表 TLB 是一种高速缓存，用于存储虚拟地址到物理地址的映射.

快表 TLB 的缺失：TLB 中没有一个表项能匹配虚拟地址.

- 页在主存中，只需要创建缺失的表项
- 页不在主存中，需要将控制权交给操作系统来处理

当处理缺失时，需要查找页表项，如果匹配的页表项的有效位无效，则该页就不在主存中. 如果有效位有效，则取回所需的表项.

访存时间：

- 最好情况：虚拟地址由 TLB 转换，然后送到 Cache，找到相应的数据，然后取回并送入处理器
- 最坏情况：TLB、页表、Cache 三个部件都发生缺失