

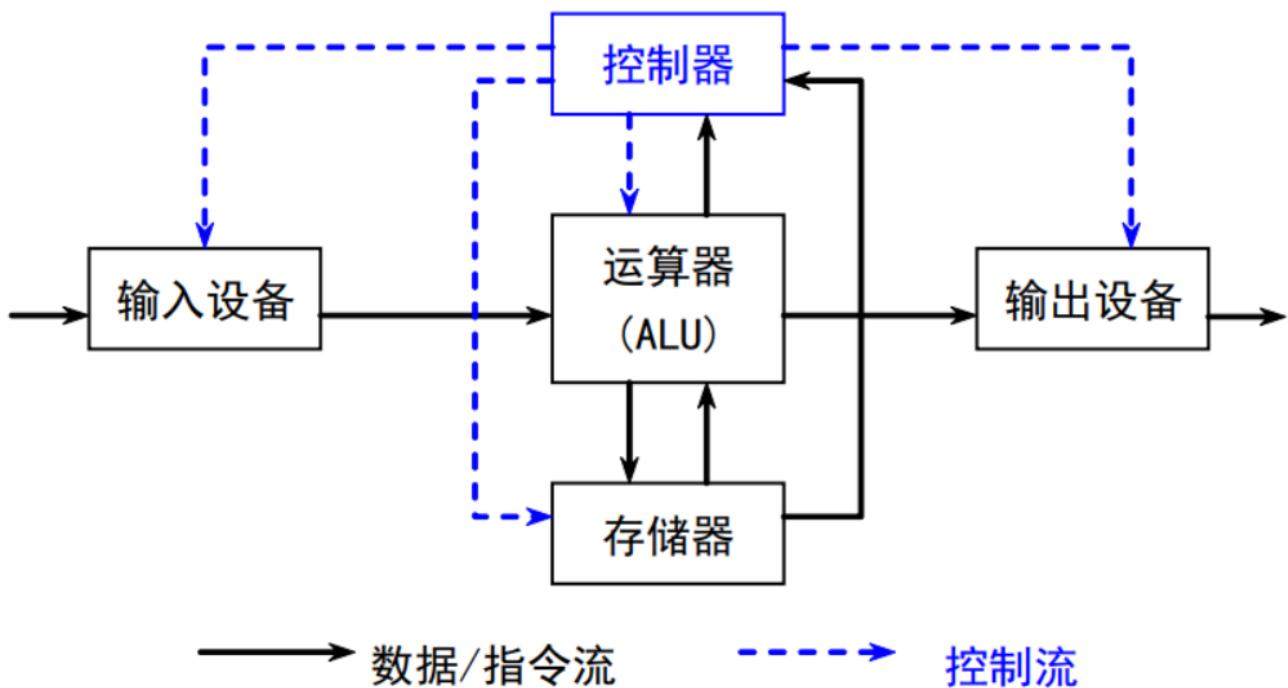
哈工大计算机体系结构笔记

第一章-计算机体系结构的基本概念

1.1 计算机体系结构的概念

1.1.1 存储程序计算机

存储程序机器的结构



四个组成部分：

- 运算器
- 存储器
- 输入输出设备
- 控制器

主要特点：

1. 以运算器为中心

2. 采用存储程序原理
3. 存储器是按地址访问的
4. 控制流由指令流产生
5. 指令由操作码和地址码组成
6. 数据以二进制编码表示

一个机器周期里面安排的操作序列

1. 计算机从存储器中取出一条指令
 2. 对这条指令进行译码
 - 分解并确定这条指令所指示的操作
 - 确定操作对象（操作数）所在的位置
 - 某个寄存器单元、存储器单元或者输入设备
 3. 取操作数并送到运算器
 4. 运算器按照译码确定的操作进行运算
 5. 运算结束后，将结果送到指定的位置
- 计算机准备执行下一条指令

1.1.2 计算机体系结构、组成和实现

计算机体系结构的概念由阿姆道尔于 1964 年首次明确：程序员所看到的计算机的属性，即概念性结构与功能特性。

对于通用寄存器型的机器，这些属性主要是指：

1. 数据表示
2. 寻址规则
3. 寄存器定义
4. 指令系统
5. 中断系统
6. 机器工作状态的定义和切换
7. 存储系统
8. 信息保护
9. I/O 结构

计算机体系结构包括以下三个方面：

1. 计算机指令系统 ISA
2. 计算机组装
3. 计算机硬件实现

1.1.3 系列机和兼容

系列机：具有相同体系结构，但组成和实现不同的一系列不同型号的计算机系统

一种体系结构可以有多种组织、多种物理实现。

软件兼容：

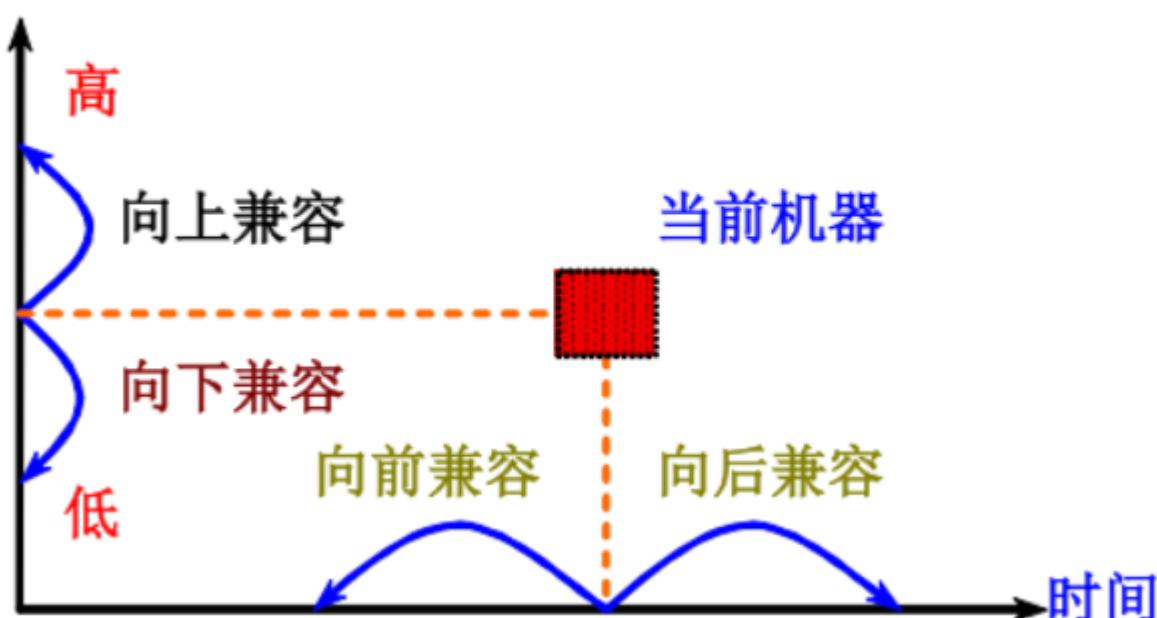
- 系列机具有相同的体系结构，软件可以在系列计算机的各档机器上运行。
- 同一个软件可以不加修改地运行于体系结构相同的各档机器，而且它们所获得的结果一样，差别只在于有不同的运行时间。
- 兼容分为二进制级兼容、汇编级兼容、高级语言兼容、数据级兼容等等。

兼容机：不同厂家生产的具有相同体系结构的计算机

兼容性：

- 向上(下)兼容指的是按某档机器编制的程序，不加修改的就能运行于比它高(低)档的机器
- 向前(后)兼容指的是按某个时期投入市场的某种型号机器编制的程序，不加修改地就能运行于在它之前(后)投入市场的机器

机器档次



向下兼容和向后兼容比向上兼容和向前兼容更重要

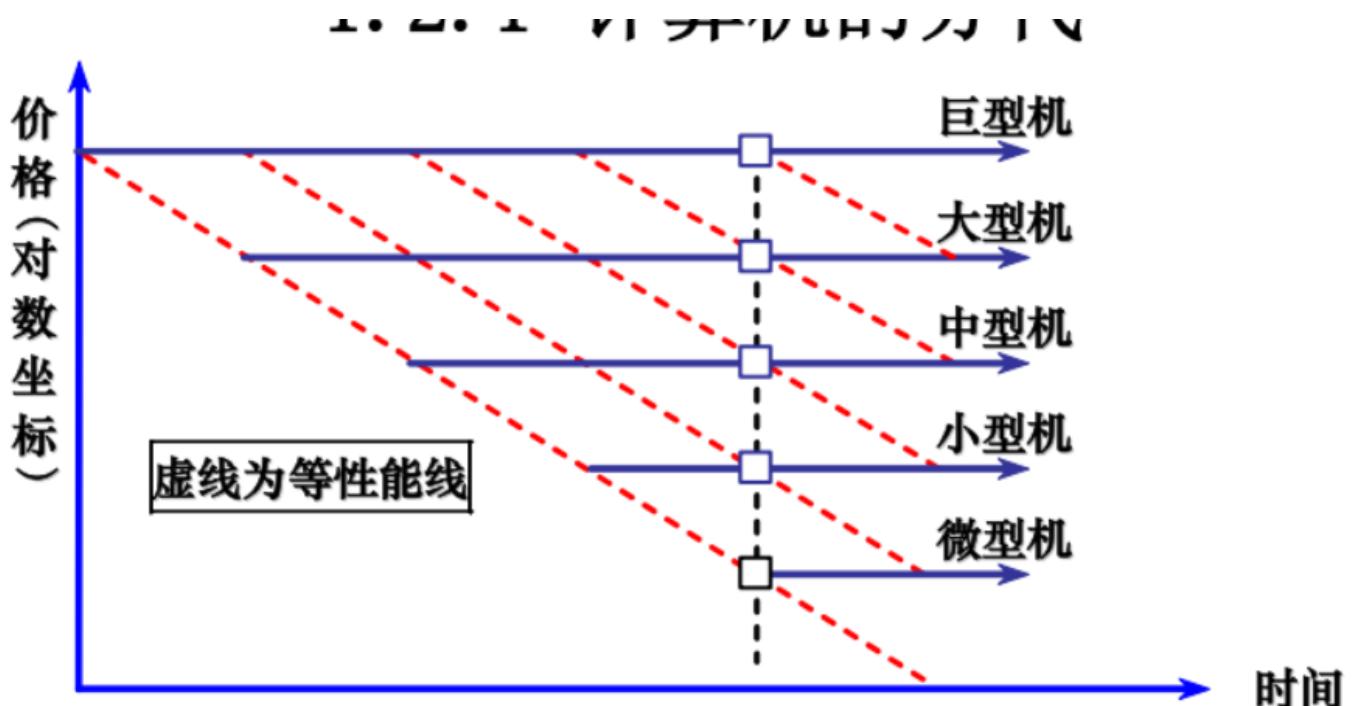
1.2 计算机体系结构的发展

1.2.1 计算机的分代

一般认为到目前为止计算机已经发展了五代

分代	器件	体系结构技术	软件技术	典型机器
第一代 (1945-1954)	电子管和继电器	存储程序计算机、 程序控制I/O	机器语言和汇编语言	普林斯顿ISA、 ENIAC IBM701
第二代 (1955-1964)	晶体管、磁芯、印刷电路	浮点数据表示、寻址技术、中断、 I/O处理机	高级语言和编译、批处理监控系统	Univac LARC CDC1604 IBM7030
第三代 (1965-1974)	SSI 和 MSI 、多层印刷电路、微程序	流水线、 Cache 、 先行处理、系列计算机	多道程序和分时操作系统	IBM360/370CDC 6600/7600 、 DEC PDP-8
第四代 (1974-1990)	LSI 和 VLSI 、半导体存储器	向量处理、分布式存储器	并行与分布处理	Cray-1 、 IBM 3090 、 DEC VAX9000 、 Convax-1
第五代 (1991-)	高性能微处理器、 大规模高密度电路	指令级并行、 SMP 、 MP 、 MPP 、网络	可扩展并行与分布处理	SGI Cray T3E IBM xServer Sun E10000

计算机曾根据价格分为 5 个档次：巨型机、大型机、中型机、小型机、微型机。



技术和性能的“下移”。

新型体系结构的设计一方面是合理地增加计算机系统中硬件的功能比例、另一方面则是通过多种途径提高计算机体系结构中的并行性。

1.2.2 软件的发展

最重要的发展趋势之一：程序和数据使用的存储器容量不断扩大。

1. 计算机语言和编译技术
2. 操作系统
3. 软件工具和中间件

1.2.3 应用的发展

计算机技术和市场分为嵌入式计算机、掌上计算机、台式计算机、服务器、数据中心。

1.2.4 相关核心产品技术的发展

1. 逻辑电路 摩尔定律
2. DRAM 单个 DRAM 模块的容量每年增加 25-40%，增速逐年下降。
3. 闪存 近几年，每年约 50-60% 的速度增长，大约每两年翻一番，大量用于便携式设备。
4. 磁盘 从 04 年开始，大约每年增长 40%，约每 3 年翻一番。SSD 的成本的迅速下降。
5. 网络

1.2.5 体系结构的发展

1. 分布的 I/O 处理能力
2. 保护的存储器空间
3. 存储器组织结构的发展
4. 并行处理技术
5. 指令集的发展

1.2.6 并行处理技术的发展

并行性：计算机系统在同一时刻或者同一时间间隔内进行多种运算或操作。只要时间上相互重叠，就存在并行性。

- 同时性：两个及以上的事件在同一时刻发生
- 并发性：两个及以上的事件在同一时间间隔内发生

从执行程序的角度来看，并行性等级从低到高可分为：

- 指令内部并行：单条指令中各微操作之间的并行。
- 指令级并行：并行执行两条或两条以上的指令。
- 线程级并行：并行执行两个或两个以上的线程。通常是以一个进程内派生的多个线程为调度单位。
- 任务级或过程级并行：并行执行两个或两个以上的过程或任务（程序段），以子程序或进程为调度单元。
- 作业或程序级并行：并行执行两个或两个以上的作业或程序。

从处理数据的角度来看，并行性等级从低到高可分为：

- 字串位串：每次只对一个字的一位进行处理。最基本的串行处理方式，不存在并行性。
- 字串位并：同时对一个字的全部位进行处理，不同字之间是串行的。开始出现并行性。
- 字并位串：同时对许多字的同一位（称为位片）进行处理。具有较高的并行性。
- 全并行：同时对许多字的全部位或部分位进行处理。最高一级的并行。

计算机系统结构的 Flynn 分类法：按照指令和数据的关系，把计算机系统的结构分为 4 类：

- 单指令流单数据流 SISD (Single Instruction stream Single Data stream)
- 单指令流多数据流 SIMD (Single Instruction stream Multiple Data stream)
- 多指令流单数据流 MISD (Multiple Instruction stream Single Data stream)
- 多指令流多数据流 MIMD (Multiple Instruction stream Multiple Data stream)

提高并行性的途径：

1. 时间重叠
2. 资源重复
3. 资源共享

在单机系统并行性的发展中：

- 起主导作用的是时间重叠，基础是部件功能专用化
- 资源重复运用也十分普遍

- 多体存储器
- 多操作部件
 - 通用部件被分解成若干个专用部件，如加法部件、乘法部件、除法部件、逻辑运算部件等，而且同一种部件也可以重复设置多个。
 - 只要指令所需的操作部件空闲，就可以开始执行这条指令（如果操作数已准备好的话）。
- 阵列处理机（并行处理机）

更进一步，设置许多相同的处理单元，让它们在同一个控制器的指挥下，按照同一条指令的要求，对向量或数组的各元素同时进行同一操作，就形成了阵列处理机。

- 资源共享的实质上是用单处理机模拟多处理机的功能

在多级系统并行性的发展中，遵循时间重叠、资源重复、资源共享原理，发展为 3 种不同的多处理机：同构型多处理机、异构型多处理机、分布式系统。

耦合度：反映多机系统中各机器之间物理连接的紧密程度和交互作用能力的强弱。

- **最低耦合：**耦合度最低，除通过某种中间存储介质之外，各计算机之间没有物理连接，也无共享的联机硬件资源。
- **松散耦合系统（间接耦合系统）：**一般是通过通道或通信线路实现计算机之间的互连，可以共享外存设备（磁盘、磁带等）。机器之间的相互作用是在文件或数据集一级上进行。
- **紧密耦合系统（直接耦合系统）：**在这种系统中，计算机之间的物理连接的频带较高，一般是通过总线或高速开关互连，可以共享主存。

1.3 计算机系统设计和分析

1.3.1 成本和价格

商品的标价由这样一些因素构成：原料成本、直接成本、毛利和折扣。

性能/成本设计：设计者需取得性能与成本之间的平衡。

对计算机系统成本产生影响的主要因素有：时间、产量、商品化等。最直接的影响是时间。

1.3.2 基准测试程序

五类测试程序：

1. 真实程序
2. 修正的（或者脚本化）应用程序
3. 核心测试程序
4. 小测试程序
5. 合成测试程序

测试程序包

1.3.1 量化设计基本原则

大概率事件优先原则：对于大概率事件(最常见的事件)，赋予它优先的处理权和资源使用权，以获得全局的最优结果

Amdahl 定律：

$$\text{系统加速比} = \frac{\text{总执行时间}_{\text{加速前}}}{\text{总执行时间}_{\text{加速后}}} = \frac{1}{(1 - \text{可改进比例}) + \frac{\text{可改进比例}}{\text{部件加速比}}}$$

1. 性能增加的递减规则

- 仅仅对计算机中的一部分做性能改进，则改进越多，系统获得的效果越小

2. Amdahl定律的一个重要推论

- 针对整个任务的一部分进行优化，则**最大加速比**不大于

$$\frac{1}{1 - \text{可改进比例}}$$

3. Amdahl定律衡量一个“好”的计算机系统

- 具有高性能价格比的计算机系统是一个**带宽平衡**的系统，而不是看它使用的某些部件的性能

程序的局部性原理：程序总是趋向于使用最近使用过的数据和指令。

CPU 的性能公式：

$$T_{CPU} = CPI \times IC \times T_{CLK}$$

进一步细化：一共有 n 条指令，第 i 条指令处理时间为 CPI_i ，出现次数为 IC_i

$$T_{CPU} = \sum_i IC_i \times CPI_i \times T_{CLK}$$
$$CPI = \frac{1}{IC} \times \sum_i IC_i \times CPI_i$$

第二章 指令系统

2.1 指令系统概述

指令系统是计算机系统中软件与硬件交界的一个主要标志：

- 软件设计人员利用指令系统编制各种应用软件和系统软件
- 硬件设计人员采用各种手段实现指令系统

2.2 指令集结构的分类

一般来说，可以从如下五个因素考虑对计算机指令集结构进行分类，即：

- 在 CPU 中操作数的存储方法
 - 堆栈
 - 累加器
 - 寄存器
- 指令中显式表示的操作数个数
- 操作数的寻址方式
- 指令集所提供的操作类型
- 操作数的类型和大小

指令中的操作数可以被明确地显式给出，也可以按照某种约定隐式地给出。

$Z=X+Y$ 在这三种指令集结构上的实现方法

堆栈	累加器	寄存器 (寄存器-存储器)	寄存器 (寄存器-寄存器)
PUSH X PUSH Y <u>ADD</u> POP Z	LOAD X <u>ADD Y</u> Store Z	LOAD R1,X <u>ADD R1,Y</u> Store R1,Z	LOAD R1,X LOAD R2,Y <u>ADD R3,R1,R2</u> Store R3,Z

通用寄存器型指令集结构的优点：

- 在表达式求值方面，比其它类型指令集结构都具有更大的灵活性
- 寄存器可以用来存放变量
 - 寄存器比存储器快，减少了存储器的通信量，加快程序执行速度
 - 可以用更少的地址位来寻址寄存器，有效改进程序代码大小。

通用寄存器型指令集结构可以进一步细分：

- 寄存器-寄存器型 R-R
- 寄存器-存储器型 R-M
- 存储器-存储器型 M-M

ALU指令中 存储器操作 数的个数	ALU指令中 操作数的最 多 个数	结构 类型	机器实例
0	3	RR	MIPS, SPARC, Alpha, PowerPC, ARM
1	2	RM	IBM 360/370, Intel 80x86, Motorola 6800
	3	RM	IBM 360/370
2	2	MM	VAX
3	3	MM	VAX

三类指令集的优缺点：

指令集结 构类型	优点	缺点
寄存器- 寄存器 (0, 3)	指令字长固定，指令结构简洁，是一 种简单的代码生成模型，各种指 令的执行时钟周期数相近	与指令中含存储器操作数的指令系统 结构相比，指令条数多，目标代码不 够紧凑，因而程序占用的空间比较大
寄存器- 存储器型 (1, 2)	可以在ALU指令中直接对存储器操 作数进行引用，而不必先用load指 令进行加载，容易对指令进行编码， 目标代码比较紧凑	由于有一个操作数的内容将被破坏， 所以指令中的两个操作数不对称。在 一条指令中同时对寄存器操作数和存 储器操作数进行编码，有可能限制指 令所能够表示的寄存器个数。指令的 执行时钟周期因操作数的来源（寄存 器或存储器）的不同而差别比较大
存储器- 存储器 (2, 2) 或 (3, 3)	目标代码最紧凑，不需要设置存储 器来保存变量	指令字长变换很大，特别是3个操作 数指令。而且每条指令完成的工作也 差别很大。对存储器的频率访问会使 存储器成为瓶颈。这种类型的指令系 统现在已经不用了

2.3 寻址方式

寻址方式	指令实例	含 义
寄存器寻址	ADD R1 , R2	$Regs[R1] \leftarrow Regs[R1] + Regs[R2]$
立即值寻址	ADD R3 , #6	$Regs[R3] \leftarrow Regs[R3] + 6$
偏移寻址	ADD R3 , 120(R2)	$Regs[R3] \leftarrow Regs[R3] + Mem[120+Regs[R2]]$
寄存器间接寻址	ADD R4 , (R2)	$Regs[R4] \leftarrow Regs[R4] + Mem[Regs[R2]]$
索引寻址	ADD R4 , (R2 + R3)	$Regs[R4] \leftarrow Regs[R4] + Mem[Regs[R2]+Regs[R3]]$
直接寻址或绝对寻址	ADD R4 , (1010)	$Regs[R4] \leftarrow Regs[R4] + Mem[1010]$
存储器间接寻址	ADD R2 , @(R4)	$Regs[R2] \leftarrow Regs[R2] + Mem[Mem[Regs[R4]]]$
自增寻址	ADD R1 , (R2)+	$Regs[R1] \leftarrow Regs[R1] + Mem[Regs[R2]]$ $Regs[R2] \leftarrow Regs[R2] + d$
自减寻址	ADD R1, -(R2)	$Regs[R2] \leftarrow Regs[R2] - d$ $Regs[R1] \leftarrow Regs[R1] + Mem[Regs[R2]]$
缩放寻址	ADD R1 , 80(R2)[R3]	$Regs[R1] \leftarrow Regs[R1] + Mem[80 + Regs[R2] * d]$

两种表示寻址方式的方法：

- 将寻址方式编码与操作码中，由操作码描述相应操作的寻址方式。
 - 适合：采用 load-store 结构的处理机，寻址方式只有很少几种。MIPS 指令集
- 在指令字中设置专门的寻址字段，直接指出寻址方式。
 - 灵活，操作码短，但需要设置专门的寻址方式字段，而且操作码和寻址方式字段合起来所需要的总位数可能会比隐含方法的总位数多。
 - 适合：有多种寻址方式的处理机，且指令有多个操作数。VAX11

数据在内存中存放要对齐：信息在主存中存放的起始地址必须是该信息宽度（字节数）的整数倍。

• 满足以下条件

- 字节信息的起始地址为： $\times \dots \times \times \times \times$
- 半字信息的起始地址为： $\times \dots \times \times \times 0$
- 单字信息的起始地址为： $\times \dots \times \times 0 0$
- 双字信息的起始地址为： $\times \dots \times 0 0 0$

存在空间浪费，但保证访问速度。

2.4 操作数类型和大小

- 操作数类型和操作数表示也是软硬件主要界面之一。
- 操作数类型是面向应用、面向软件系统所处理的各种数据结构。
- 操作数表示是硬件结构能够识别、指令系统可以直接调用的那些结构。

- 操作数表示所表征的那些操作数类型，是应用软件和系统软件所处理的操作数类型的子集。



- 整数（定点）：二进制补码表示；其大小可以是字节（8位）、半字（16位）或单字（32位）。
- 浮点：可以分为单精度浮点（单字大小）和双精度浮点（双字大小）。当前普遍采用的是IEEE 754浮点操作数表示标准。
- 字符和字符串：8位ASCII码表示。

- 十进制：面向商业应用，通常采用“**压缩十进制**”或“**二进制编码十进制（BCD）**”表示。压缩十进制数据表示用**4位**编码数字0~9，然后将两个十进制数字压缩在一个字节中存储。如果将十进制数字直接用字符串来表示，就叫做“**非压缩十进制**”表示法。
- 操作数类型的表示主要有如下两种方法：
 - 操作数的类型可以**由操作码的编码指定**，这也是最常见的一种方法；
 - 数据可以附上**由硬件解释的标记（tag）**，由这些标记指定操作数的类型，从而选择适当的运算。
然而有标记数据的机器却非常少见。

操作数大小主要有：字节、半字 16、单字 32、双字 64。

2.5 指令系统的设计和优化

2.5.1 指令系统设计的基本原则

指令系统的设计：

- 指令的功能设计
- 指令的格式设计

确定哪些功能由硬件实现，主要考虑三个因素：速度，成本，灵活性。

- 硬件实现：快、高、差
- 软件实现：慢、低、好

对指令系统的基本要求：

- 完整性：在一个有限可用的存储空间内，对于任何可解的问题，编制计算程序时，指令系统所提供的指令足够使用。
- 规整性：主要包括对称性和均匀性。
 - 对称性：所有与指令系统有关的存储单元的使用、操作码的设置等都是对称的。
 - 均匀性：指对于各种不同的操作数类型、字长、操作种类和数据存储单元，指令的设置都要同等对待。
- 正交性：在指令中各个不同含义的字段，如操作类型、数据类型、寻址方式字段等，在编码时应互不相关、相互独立。
- 高效率：指令的执行速度快、使用频度高。
- 兼容性：主要是要实现向后兼容，指令系统可以增加新指令，但不能删除指令或更改指令的功能。

2.5.2 控制指令

控制指令是用来改变控制流的，有四种：

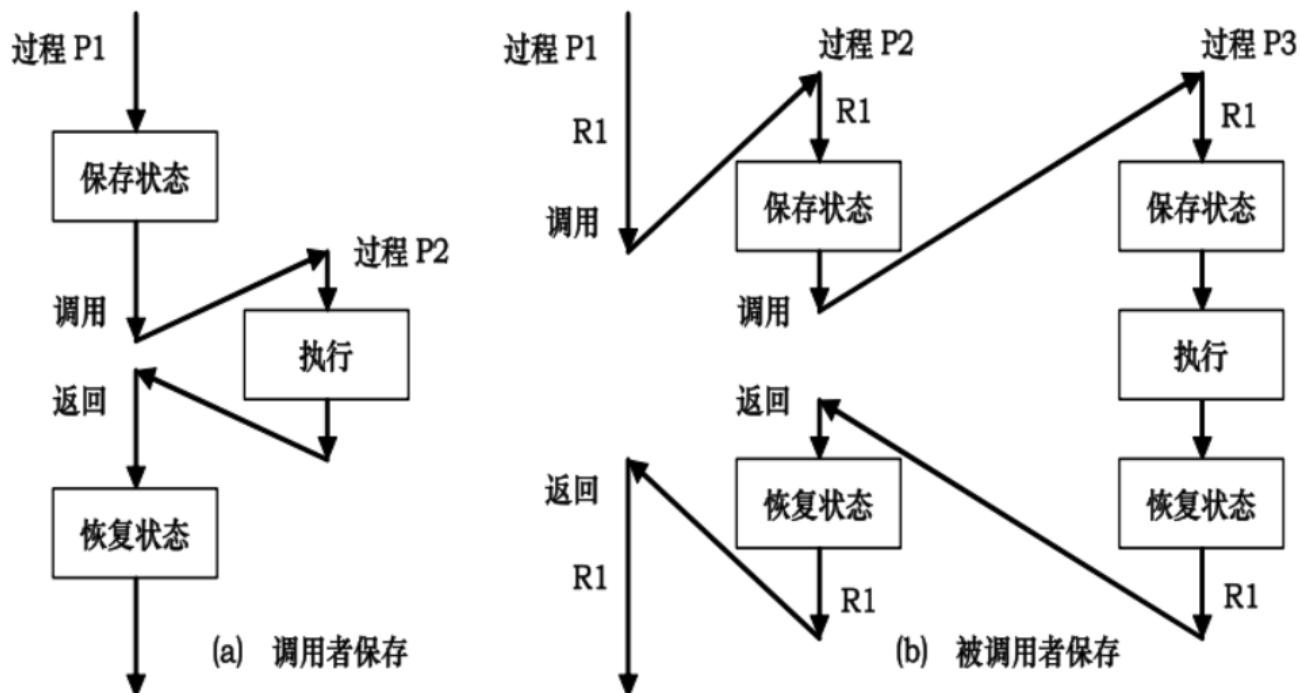
- 分支：有条件跳转（大部分）
- 跳转：无条件跳转
- 过程调用
- 过程返回

条件分支指令的表示：

分支条件表示	优 点	缺 点
条件码 (CC) : 在程序的控制下, 由 ALU 操作设置特殊的位	可以自由设置分支条件	必须从一条指令将分支条件信息传送到分支指令, 所以 CC 是额外状态, 条件码限制了指令执行顺序
条件寄存器 : 比较指令把比较结果放入任何一个寄存器, 检测时就检测该寄存器	简单	占用了一个寄存器
比较分支 : 比较操作是分支指令的一部分, 比较受限制	一条指令完成了两条指令的功能	分支指令的操作增多

过程调用和返回的状态保存策略:

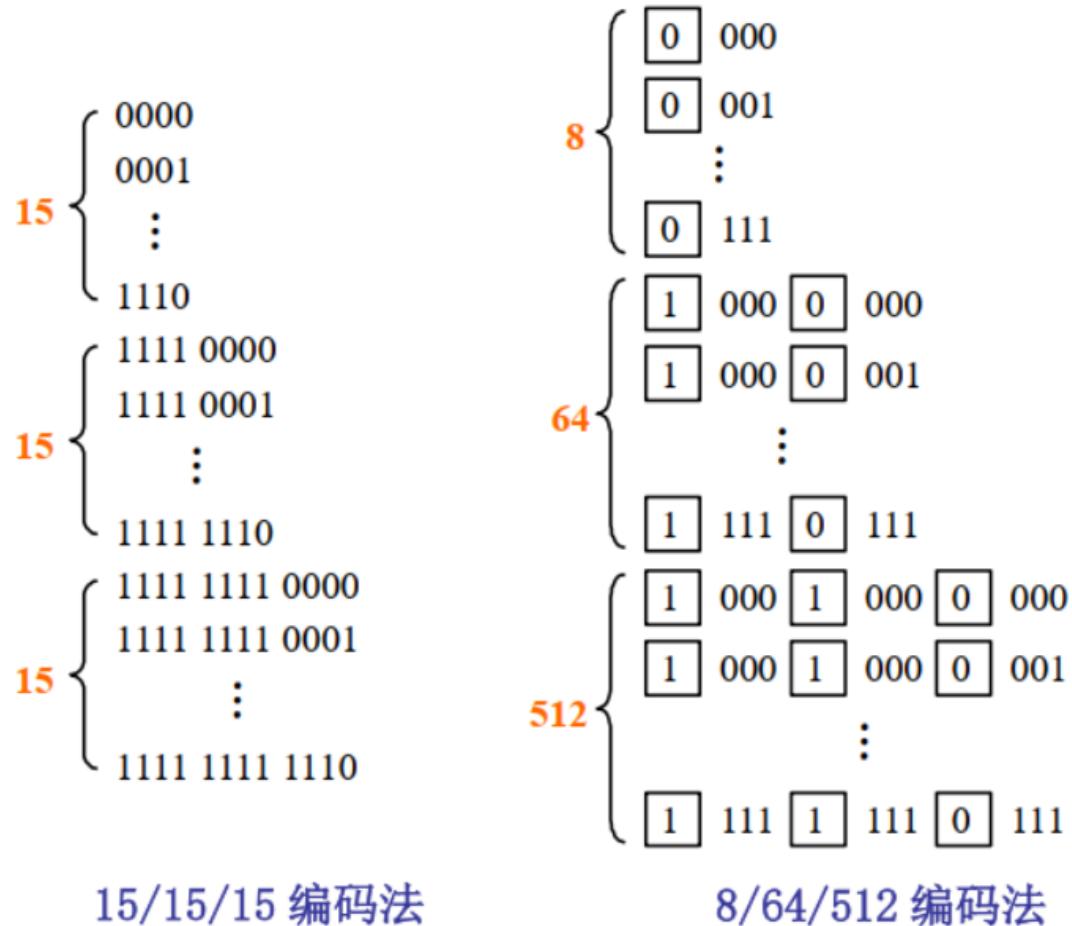
- 调用者保存
- 被调用者保存



2.5.3 指令操作码的优化

如何用最短的位数表示指令的操作信息和地址信息

等长拓展码:



定长拓展码：操作码长度固定。

2.6 指令系统的发展和改进

2.6.1 沿 CISC 方向改进

CISC（复杂指令计算机）

- 面向目标程序增强指令功能：对于使用频度高的指令，用硬件加快其执行；对于使用频度高的指令串，用一条新的指令来替代。
 - 增强运算型指令的功能
 - 增强数据传送指令的功能
 - 增强程序控制指令的功能
- 面向高级语言的优化改进指令系统
 - 增强对高级语言和编译器的支持
 - 高级语言计算机：间接或直接执行高级语言的机器
- 面向操作系统的优化改进指令系统

- 处理机工作状态和访问方式的切换
- 进程的管理和切换
- 存储管理和信息保护
- 进程的同步与互斥，信号灯的管理

2.6.2 沿 RISC 方向改进

RISC（精简指令计算机）

CISC 的缺点：

- 各指令的使用频率相差悬殊
- 指令系统的复杂性使得控制器硬件变得十分复杂
 - 占用大量芯片面积
 - 增加研发成本和时间，容易造成设计错误
- 其 CPI 值比较大，执行速度慢
- 规整性不好，不利于使用流水线技术提高性能

RISC 指令集设计原则：

- 选取使用频率最高的指令，并补充一些最有用的指令；
- 每条指令的功能应尽可能简单，并在一个机器周期内完成（采用流水线技术后）；
- 所有指令长度均相同；
- 只有**load**和**store**操作指令才访问存储器，其它指令操作均在寄存器之间进行；
- 大多数指令都采用硬连线技术；
- 以简单有效的方式支持高级语言；
- 充分利用流水线技术。

2.7 MIPS 指令系统结构

寄存器：

- 32 个 32 位通用寄存器，R0 恒为 0
- 32 个 32 位单精度浮点寄存器

数据类型

- 整型数据：8、16、32 位
- 浮点数据：32 位单精度、64 位双精度，IEEE754 标准

寻址方式：

- 寄存器寻址
- 立即寻址
- 偏移寻址
 - 寄存器间接寻址
 - 直接寻址

指令格式：

- I 类指令

I 类型指令

6	5	5	16
操作码	rs1	rd	立即值

字节、半字、字的载入和存储；
 $rd \leftarrow rs1 \ op \ 立即值。$

- R 类指令

R 类型指令

6	5	5	5	11
操作码	rs1	rs2	rd	Func

寄存器—寄存器 ALU 操作: $rd \leftarrow rs1 \text{ func } rs2$;
函数对数据的操作进行编码: 加、减、...;
对特殊寄存器的读/写和移动。

- J 类指令

J 类型指令

6	26
操作码	与 PC 相加的偏移量

跳转, 跳转并链接, 从异常 (exception) 处自陷和返回。

操作类型:

- Load 和 store 操作
- ALU 操作
- 分支和跳转指令
- 浮点操作

第三章 流水线技术

3.1 流水线概述

3.1.1 流水线基本概念

流水线技术: 将一重复的时序过程分解为若干子过程, 每个子过程都可有效地在其专用功能段上与其它子过程同时执行, 这种技术称为流水技术。

时空图：从时间和空间两个方面描述流水线的工作过程，横坐标表示时间，纵坐标表示各流水段。

流水线特点：

- 流水过程由多个相关的子过程组成，这些子过程称为流水线的“级”或“段”。段的数目称为流水线的“深度”
- 每个子过程由专用的功能段实现
- 各功能段的时间应基本相等，通常为 1 个时钟周期（1 拍）
- 流水线需要经过一定的通过时间才能稳定
- 流水技术适合于大量重复的时序过程

3.1.2 流水线分类

按功能分类：

- 单功能流水线
- 多功能流水线

按同一时间各段之间的连接方式分类：

- 静态流水线
- 动态流水线

按流水的级别分类：

- 部件级流水线
- 处理机级流水线
- 处理机间流水线

按数据表示分类：

- 标量流水处理机
- 向量流水处理机

按是否有反馈回路分类：

- 线性流水线
- 非线性流水线，存在流水线调度问题

3.2 MIPS 基本流水线

3.2.1 基本 MIPS 流水线

分为五个周期：取值、译码、执行/有效地址计算、访存/分支、写回

流水段	任何指令类型		
IF	$IF/ID. IR \leftarrow Mem[PC]$ $IF/ID. NPC, PC \leftarrow (if EX/MEM. cond \{EX/MEM. ALUOutput\} else \{PC+4\})$;		
ID	$ID/EX. A \leftarrow Regs[IF/ID. IR_{6\dots 10}]$; $ID/EX. B \leftarrow Regs[IF/ID. IR_{11\dots 15}]$; $ID/EX. NPC \leftarrow IF/ID. NPC$; $ID/EX. IR \leftarrow IF/ID. IR$; $ID/EX. Imm \leftarrow (IR_{16})^{16\#} IR_{16\dots 31}$;		
EX	ALU 指令	Load/Store 指令	分支指令
	$EX/MEM. IR \leftarrow ID/EX. IR$; $EX/MEM. ALUOutput \leftarrow ID/EX. A op ID/EX. B$ 或 $EX/MEM. ALUOutput \leftarrow ID/EX. A op ID/EX. Imm$; $EX/MEM. cond \leftarrow 0$;	$EX/MEM. IR \leftarrow ID/EX. IR$; $EX/MEM. B \leftarrow ID/EX. B$ $EX/MEM. ALUOutput \leftarrow ID/EX. A + ID/EX. Imm$; $EX/MEM. cond \leftarrow 0$;	$EX/MEM. ALUOutput \leftarrow ID/EX. NPC + ID/EX. Imm$; $EX/MEM. cond \leftarrow (ID/EX. A op 0)$;

3.2.2 流水线性能分析

三项性能指标：吞吐率、加速比、效率

吞吐率：单位时间内流水线所完成的任务数或输出结果的数量（计算公式：完成的指令条数/完成这些指令条数所需的时间）

- 最大吞吐率：
 - 如果各段时间相等，均为 Δt_0 ，则 $TP_{MAX} = \frac{1}{\Delta t_0}$
 - 如果各段时间不等，第*i*段时间为 Δt_i ，则 $TP_{MAX} = \frac{1}{\max \Delta t_i}$
 - 最大吞吐率取决于流水线中最慢的一段所需时间，该段成为流水线性能瓶颈。
- 消除瓶颈的方法：
 - 细分瓶颈段
 - 重复设置瓶颈段
- 实际吞吐率：m 段流水，n 条指令

- 如果各段时间相等，均为 Δt_0 ，那么完成时间 $T = m\Delta t_0 + (n - 1)\Delta t_0$ ，则实际吞吐率 $TP = \frac{n}{T} = \frac{TP_{MAX}}{1 + \frac{m-1}{n}}$
- 如果各段时间不等，第*i*段时间为 Δt_i ，那么完成时间 $T = \sum_1^m \Delta t_i + (n - 1) \max \Delta t_i$ ，即第一条指令流出时间+后续指令流出时间。实际吞吐率 $TP = \frac{n}{\sum_1^m \Delta t_i + (n-1) \max \Delta t_i}$

加速比：加速比是指流水线速度与等功能的非流水线速度之比。 $S = \frac{T_{非流水}}{T_{流水}}$

效率：效率指流水线的设备利用率。

- 由于流水线有通过时间和排空时间，所以效率 $E < 1$
- 效率 $E = \frac{S}{m}$
- 也可以通过时空图占用面积与总面积之比得出

有关流水线性能的若干问题：

- 流水线并不能减少（而且一般是增加）单条指令的执行时间，但能够提高吞吐率
- 增加流水线的深度可以提高流水线性能
- 流水线深度受限于流水线的延迟和额外开销
- 需要用高速锁存器作为流水线寄存器- Earle 锁存器
- 指令之间存在的相关，产生了流水线的冲突，进而限制了流水线的性能

3.3 流水线中的冲突

流水线冲突：相邻或相近的两条指令因存在某种关联，后一条指令不能在原先指定的时钟周期开始执行。

消除冲突基本方法：暂停

3.3.1 结构冲突

原因：

- 功能部件不是全流水
- 重复设置资源不足

避免方法：

- 所有功能部件全部流水化
- 设置足够多硬件资源，代价很大

3.3.2 数据冲突

原因：当指令在流水线中重叠执行时，流水线有可能改变指令读/写操作数的顺序，使之不同于它们在非流水实现时的顺序，这将导致数据冲突。

消除方法：插入暂停周期，通过定向技术减少暂停

数据冲突分类：

- 写后读冲突(RAW): 最常见
- 写后写冲突(WAW): MIPS 不会出现
- 读后写冲突(WAR): MIPS 不会出现
- 读后读冲突(RAR): 不引起冲突

需要暂停的数据冲突：`load` 指令后紧跟一条使用寄存器的指令时，需要暂停一个周期才能使用定向解决冲突

通过编译器调度消除冲突：

请为下列表达式生成没有暂停的指令序列：

a = b - c ;

d = e - f ;

假设载入延迟为1个时钟周期。

调度前的代码	调度后的代码
LW Rb, b	LW Rb, b
LW Rc, c	LW Rc, c
ADD Ra, Rb, Rc	LW Re, e
SW Ra, a	ADD Ra, Rb, Rc
LW Re, e	LW Rf, f
LW Rf, f	SW Ra, a
SUB Rd, Re, Rf	SUB Rd, Re, Rf
SW Rd, d	SW Rd, d

3.3.3 控制冲突

分支指令转移成功在访存段才能得到正确的跳转地址，简单暂停后续指令的指令会导致流水线暂停 3 个周期

减少分支开销的途径：

- 尽早判断是否分支成功
- 分支成功时尽早计算得出转移目标地址

优化时两种方法都要采用

对于 MIPS 流水线的改进：

- 分支测试提前到 ID 段
- 分支地址计算也提前到 ID 段

- 开销降低到 1 拍

减少流水线分支损失的方法：

- 冻结或排空流水线：简单的暂停后续指令的执行
- 预测分支失败：流水线照常流动，如果分支成功则要清空流水线，重新取值执行
- 预测分支成功：始终预测分支成功，直接从目标处取值执行
- 延迟分支：分支开销为 n 的分支指令后紧跟有 n 个延迟槽，遇到分支指令时正常处理并执行延迟槽中指令，减少分支开销。
 - 从前调度
 - 从目标处调度
 - 从失败处调度

表 3.7 调度分支延迟指令的三种常用方法的特点及其局限性

调度策略	对调度的要求	对流水线性能改善的影响
从前调度	分支必须不依赖于被调度的指令	总是可以有效提高流水线性能
从目标处调度	如果分支转移失败，必须保证被调度的指令对程序的执行没有影响，可能需要复制被调度指令	分支转移成功时，可以提高流水线性能。但由于复制指令，可能加大程序空间
从失败处调度	如果分支转移成功，必须保证被调度的指令对程序的执行没有影响	分支转移失败时，可以提高流水线性能

取消分支：分支指令中包含预测方向，若预测正确，正常执行延迟槽中的指令，否则将其转换为空操作

分支处理方法的性能：

- 假设理想 $CPI = 1$ ，则加速比 $S = \frac{D}{1+C} = \frac{D}{1+f \times p_{\text{分支}}}$ ， D 为流水线深度， $p_{\text{分支}}$ 为分支开销， C 为分支引起的流水线暂停时钟周期数（平均值）， f 为分支指令出现的频率

调度方法	每条条件分支指令的分支损失		每条无条件分支指令的损失	每条分支指令的平均分支损失		具有分支暂停的有效 CPI	
	整型平均	浮点平均		整型平均	浮点平均	整型平均	浮点平均
暂停流水线	1.00	1.00	1.00	1.00	1.00	1.17	1.15
预测分支成功	1.00	1.00	1.00	1.00	1.00	1.17	1.15
预测分支失败	0.62	0.70	1.00	0.69	0.74	1.12	1.11
延迟分支	0.25	0.35	0.00	0.21	0.30	1.04	1.04

3.4 MIPS 流水线分析（略）

3.5 向量处理机

处理 $d = a * (b + c)$, a、b、c、d 均为水平向量

- 水平处理方式，一位一位计算
- 垂直处理方式， $k = b + c, d = a * k$
- 分组处理方式，将长度为 N 的向量分为 m 组，每组 n 个元素，组内纵向处理，依次处理各组

向量机速度评价方法：

- 标量机通常用每秒执行多少指令衡量
- 每秒取得多少浮点运算结果衡量向量机速度

CRAY-1 实例分析（略）

向量流水线：向量链接技术

链接只有在第一个结果流出之后才可以建立。

第四章 指令级并行

4.1 指令级并行的概念

指令级并行：当指令之间不存在相关时，它们在流水线中是可以重叠起来并行执行的。这种指令序列中存在的潜在并行性称为指令级并行，ILP

流水线处理器的实际CPI（平均每条指令使用的周期数）等于理想流水线的CPI加上各类停顿引起的周期数的总和

$$\begin{aligned} \text{CPI}_{\text{流水线}} &= \text{CPI}_{\text{理想}} \\ &+ \text{停顿}_{\text{结构冲突}} \\ &+ \text{停顿}_{\text{先写后读}} \\ &+ \text{停顿}_{\text{先读后写}} \\ &+ \text{停顿}_{\text{写后写}} \\ &+ \text{停顿}_{\text{控制冲突}} \end{aligned}$$

- 基本程序块：一段除了入口和出口以外不包含其他分支的线性代码段
- 循环级并行：开发循环体的不同迭代之间存在的并行性
 - 指令调度
 - 循环展开
 - 换名

4.1.1 循环展开调度的基本方法

循环展开：展开循环体若干次，将循环级并行转化为指令级并行的技术

编译器在完成这种指令调度时，受限于以下两个特性：

- 程序固有的指令级并行性
- 流水线功能部件的执行延迟

产生结果指令	使用结果指令	延迟时钟数
浮点计算	另外的浮点计算	3
浮点计算	浮点数据存操作 (SD)	2
浮点数据取操作 (LD)	浮点计算	1
浮点数据取操作 (LD)	浮点数据存操作 (SD)	0

- 整数流水线采用改进的MIPS整数流水线
 - 载入延迟为1个节拍
 - 由于数据的取操作的结果可以毫无停顿的通过相关通路机制传送到数据存部件，所以延迟为0
 - 有定向通道或旁路机制
- 分支指令，由整数流水线执行
 - 分支条件检测调整到ID段
 - 如果分支指令使用上一条指令的结果作为分支条件，将要延迟1节拍
 - 分支指令有1个节拍的延迟槽
- 浮点运算一般为64位

分析以下代码

```
for(i=1;i<1000;i++){
    x[i]=x[i]+s;
}
```

转变成汇编语言：

Loop: LD F0,0(R1) ;F0为向量元素
 ADDD F4,F0,F2 ;加常数F2
 SD 0(R1),F4 ;保存结果
 SUBI R1,R1,#8 ;修改指针
 BNEZ R1,Loop ;循环控制

无调度:

时钟	无调度	
1	LD	F0 , 0 (R1)
2	<i>stall</i>	
3	ADDD	F4 , F0 , F2
4	<i>stall</i>	
5	<i>stall</i>	
6	SD	0 (R1) , F4
7	SUBI	R1 , R1 , #8
8	<i>stall</i>	
9	BNEZ	R1 , LOOP
10	<i>stall</i>	

进行调度之后:

调度后

LD	F0 , 0 (R1)
SUBI	R1 , R1 , #8
ADDD	F4 , F0 , F2
<i>Stall</i>	
BNEZ	R1 , Loop
SD	8 (R1) , F4

循环展开三次得到四个循环体（无调度）：

Loop:	LD	F0,0(R1)	1
	ADDD	F4,F0,F2	2,3
	SD	0(R1),F4	4,5,6
	LD	F6,-8(R1)	7
	ADDD	F8,F6,F2	8,9
	SD	-8(R1), F8	10,11,12
	LD	F10,-16(R1)	13
	ADDD	F12,F10,F2	14,15
	SD	-16(R1), F12	16,17,18
	LD	F14,-24(R1)	19
	ADDD	F16,F14,F2	20,21
	SD	-24(R1), F16	22,23,24
	SUBI	R1,R1,#-32	25
	BNEZ	R1,Loop	26,27
	stall		28

循环展开+调度：

1. Loop:	LD	F0,0(R1)
2.	LD	F6,-8(R1)
3.	LD	F10,-16(R1)
4.	LD	F14,-24(R1)
5.	ADDD	F4,F0,F2
6.	ADDD	F8,F6,F2
7.	ADDD	F12,F10,F2
8.	ADDD	F16,F14,F2
9.	SD	0(R1),F4
10.	SD	-8(R1),F8
11.	SUBI	R1,R1,#-32
12.	SD	16(R1),F12
13.	BNEZ	R1,R2,Loop
14.	SD	8(R1),F16

这里需要注意，BNEZ 分支指令具有一个分支延迟槽

循环展开和指令调度总结：

- 保证正确性
- 注意有效性
- 使用大量不同寄存器
- 减少循环控制中的测试指令和分支指令
- 注意 LOAD/STORE 的内存地址
- 注意新的相关性

4.1.2 相关性

相关：两条指令之间存在某种依赖关系，如果两条指令相关则它们有可能不能再流水线中重叠执行或只能部分重叠执行

相关的类型：

- 数据相关（真数据相关、数据流相关）

- 名相关（反相关、输出相关）
- 控制相关

数据相关：

- 对于两条指令 i (在前) 和 j (在后) 如果下述条件之一成立，则称指令 j 与指令 i 数据相关
 - 指令 j 使用指令 i 产生的结果
 - 指令 j 与指令 k 数据相关，而指令 k 与指令 i 数据相关 (数据相关具有传递性)
- 数据相关并不一定代表冲突，也不一定就会在流水线中引起暂停
- 数据相关的解决办法：
 - 硬件：采用互锁机制，检测到相关就插入暂停周期
 - 软件：使用编译器在相关处插入空操作
- 对于寄存器的相关判断比较简单，但是存储器就相对复杂

名相关：如果两条指令使用相同的名，但是它们之间并没有数据流动，则称这两条指令存在名相关。

- 分类：
 - 反相关：指令 j 写的名和指令 i 读的名相同 (读后写相关)
 - 输出相关：指令 j 和指令 i 写相同的名 (写后写相关)
- 解决办法：换名
 - 编译器静态完成或硬件动态完成

控制相关：由分支指令引起的相关

- 两个原则：
 - 与控制相关的指令不能移到分支指令之前，即控制有关的指令不能调度到分支指令的控制范围之外。
 - 与控制无关的指令不能移到分支指令之后，即控制无关的指令不能调度到分支指令的控制范围以内。

4.2 指令的动态调度

动态调度

- 目的
 - 在程序执行的时候，解决 WAW、 WAR、 RAW 带来的流水线冲突
- 优点
 - 处理在编译的时候未知的相关，简化编译器
 - 在不同的流水线上都能有效的运行
- 缺点：
 - 很大增加了硬件复杂性

4.2.1 动态调度的原理

到目前为止流水线最大的局限性：指令按序流出和按序执行

乱序执行：

- 指令的执行顺序和程序顺序是不同的
- 指令的完成也是乱序完成的
- 为了支持乱序执行将 5 段流水线的译码阶段再分为两个阶段
 - 流出 IS：指令译码，检查是否存在结构冲突
 - 读操作数 RO：等待数据冲突消失，然后读操作数

4.2.2 记分牌算法

两个浮点乘（10 拍延迟）、一个浮点除（40 拍延迟）、一个浮点加（2 拍延迟）、一个整数运算部件

执行过程：

- 流出
 - 是否存在空闲部件（结构冲突）
 - 正在执行的指令使用的目的寄存器和本指令不同（WAW 冲突）
 - 如果不能满足，阻塞本条指令以及后续指令的流出，直到条件都能满足
- 读操作数
 - 前面正在执行的指令不对本指令的源操作数寄存器进行写操作
 - 一个正在工作的功能部件已经完成了对这个寄存器的写操作（RAW 相关）
- 执行
 - 开始于取到操作数之后
 - 结果产生后修改记分牌
- 写结果：检查 WAR 相关，当出现以下情况时，不允许写结果
 - 前面的指令还没有取得操作数而且其源寄存器与本指令目的寄存器相同

记分牌结构:

- 指令状态表: 记录指令执行到那个阶段
- 功能部件状态表:
 - Busy: 指示功能部件是否工作
 - Op: 功能部件当前执行的操作
 - Fi: 目的寄存器编号
 - Fj, Fk: 源寄存器编号
 - Qj, Qk: 向源寄存器 Fj, Fk 中写结果的功能部件
 - Rj, Rk: 标示 Fj, Fk 是否就绪, 是否已经被使用
- 结果寄存器状态表: 记录写入寄存器的功能部件编号

实例:

- 指令序列

LD	F6, 34(R2)
LD	F2, 45(R3)
MULTD	F0, F2, F4
SUBD	F8, F6, F2
DIVD	F10, F0, F6
ADDD	F6, F8, F2

- 相关性
 - 先写后读相关: 第二个 LD 和 MULID、SUBD, MULTD 和 DIVD, SUBD 和 ADDD
 - 先读后写相关: SUBD、DIVD 和 ADDD
 - 写后写相关: 第一条 LD 和 ADDD 之间
 - 结构冲突: SUBD 和 ADDD 浮点加部件
- 第一拍: 第一条 LD 指令流出, 占用整数部件计算目的地址

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fi?	Fk?
		Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer		Yes	Load	F6		R2			Yes
Mult1		No							
Mult2		No							
Add		No							
Divide		No							

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
1									
FU									Integer

- 第二拍：第二个 LD 无法流出，因为它也需要整数部件，第一个 LD 完成 RO 段

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

• Issue 2nd LD?

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fi?	Fk?
		Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer		Yes	Load	F6		R2			Yes
Mult1		No							
Mult2		No							
Add		No							
Divide		No							

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
2									
FU									Integer

- 第三拍，第二条指令依旧阻塞，这里将 Rk 该为 No 并不是说未准备，而是说已经使用完了，同时记分牌依然是顺序流出的所以 MULTD 并不流出。

Instruction	j	k	Read		Exec	Write
			Issue	Open	Comp	Result
LD	F6	34+ R2	1	2	3	
LD	F2	45+ R3				
MULTD	F0	F2 F4				
SUBD	F8	F6 F2				
DIVD	F10	F0 F6				
ADDD	F6	F8 F2				

• Issue MULT?

Functional unit status:			dest	S1	S2	FU	FU	Fj?	Fk?	
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Integer							

- 第四拍，第一条指令写回结果，修改记分牌，结束对整数部件的占用，注意：这时第二条指令仍然不能流出，因为对整数部件的占用在这一拍的后半段才被删除

Instruction status:			Read	Exec	Write
Instruction	j	k	Issue	Open	Comp Result
LD	F6	34+ R2	1	2	3
LD	F2	45+ R3			4
MULTD	F0	F2 F4			
SUBD	F8	F6 F2			
DIVD	F10	F0 F6			
ADDD	F6	F8 F2			

Functional unit status:			dest	S1	S2	FU	FU	Fj?	Fk?	
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Integer							

- 5，第二条指令因为结构冲突消除所以可以流出，修改记分牌

Instruction	<i>j</i>	<i>k</i>	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	Yes	Load	F2		R3			Yes
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Integer							

- 6, 第二条指令完成 RO 段，第三条指令不存在结构冲突和 WAW 冲突所以可以流出

Instruction	<i>j</i>	<i>k</i>	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5	6	
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj
	Integer	Yes	Load	F2		R2			Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No Yes
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1 Integer							

- 7, 因为第二条指令对 MULTD 的源操作数进行写操作，所以 MULTD 指令无法进入 RO 段，而第二条指令完成 EX 段，SUBD 不存在结构冲突和 WAW 相关所以可以流出

Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6		
SUBD	F8	F6	F2	7		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

- Read multiply operands?

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Rj	Rk
	Integer	Yes	Load	F2		R3		No	
	Mult1	Yes	Mult	F0	F2	F4	Integer	No	Yes
	Mult2	No							
	Add	Yes	Sub	F8	F6	F2	Integer	Yes	No
	Divide	No							

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	Integer						

- 8 前半拍, LD 完成写回操作, 修改记分牌, DIVD 不存在结构冲突和 WAW 冲突, 流出

Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6		
SUBD	F8	F6	F2	7		
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Rj	Rk
	Integer	Yes	Load	F2		R3		No	
	Mult1	Yes	Mult	F0	F2	F4	Integer	No	Yes
	Mult2	No							
	Add	Yes	Sub	F8	F6	F2	Integer	Yes	No
	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	Integer						

- 8 后半拍, 因为 LD 指令已经写回, 所以 MULTD 和 SUBD 的源寄存器可以使用, 修改记分牌

Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6		
SUBD	F8	F6	F2	7		
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?
		Busy	Op	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Integer	No							
1	Mult1	Yes	Mult	F0	F2	F4		Yes	Yes
2	Mult2	No							
3	Add	Yes	Sub	F8	F6	F2		Yes	Yes
4	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU	Mult1			Add	Divide			
8									

- 9, ADDD 存在结构相关所以不能流出, DIVD 存在 RAW 冲突所以不能进入 RO 段, MULTD 和 SUBD 完成 RO 段

Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

• Issue ADDD?

Functional unit status:

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?
		Busy	Op	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Integer	No							
1	Mult1	Yes	Mult	F0	F2	F4		Yes	Yes
2	Mult2	No							
3	Add	Yes	Sub	F8	F6	F2		Yes	Yes
4	Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU	Mult1			Add	Divide			
9									

- 10, MULTD 和 SUBD 开始执行, 因为这两个操作都不能在一拍内完成, 所以这一拍只是在执行这两条指令

Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?	
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
9	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
1	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU	Mult1			Add	Divide			
10									

- 11, SUBD 指令执行完毕

Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?	
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
8	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
0	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU	Mult1			Add	Divide			
11									

- 12, SUBD 指令写回, ADDD 指令不能流出的原因和第 4 拍相同

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2			

- Read operands for DIVD?

Functional unit status:

Time	Name	Busy	Op	dest	SI	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
7	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	No								
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	Mult1							

- 13, ADDD 流出

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13		

Functional unit status:

Time	Name	Busy	Op	dest	SI	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
6	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
13	FU	Mult1							

- 14, ADDD 完成 RO 段，开始执行

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	

Functional unit status:

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?	
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
5	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
2	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU	Mult1			Add		Divide		
14									

- 15, ADDD 指令执行
- 16, ADDD 指令执行完成

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?	
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
3	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
0	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU	Mult1			Add		Divide		
16									

- 17, ADDD 无法写回，因为前面 DIVD 使用了 F6 并且还没有取出
- 18, 等待 MULTD 执行完成
- 19, MULTD 执行完成

Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	19
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	dest		SI	S2	FU	FU	Fj?	Fk?	
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
0	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock	F0 F2 F4 F6 F8 F10 F12 ... F30							
	FU	Mult1		Add		Divide		
19								

- 20, MULTD 写回, 修改记分牌

Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	5	6	7 8
MULTD	F0	F2	F4	6	9	19 20
SUBD	F8	F6	F2	7	9	11 12
DIVD	F10	F0	F6	8		
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	dest		SI	S2	FU	FU	Fj?	Fk?	
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status:

Clock	F0 F2 F4 F6 F8 F10 F12 ... F30							
	FU		Add		Divide			
20								

- 21, DIVD 的 RAW 冲突解除, 完成 RO 段

Instruction status:

Instruction	j	k	Issue	Read		Exec		Write	
				Oper	Comp	Result			
LD	F6	34+	R2	1	2	3	4		
LD	F2	45+	R3	5	6	7	8		
MULTD	F0	F2	F4	6	9	19	20		
SUBD	F8	F6	F2	7	9	11	12		
DIVD	F10	F0	F6	8	21				
ADDD	F6	F8	F2	13	14	16			

- WAR Hazard is now gone...

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
		Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	Yes	Add	F6	F8	F2		No	No
	Divide	Yes	Div	F10	F0	F6		Yes	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
21	FU									
					Add			Divide		

- 22, ADDD 的 WAR 相关解除，写回

Instruction status:

Instruction	j	k	Issue	Read		Exec		Write	
				Oper	Comp	Result			
LD	F6	34+	R2	1	2	3	4		
LD	F2	45+	R3	5	6	7	8		
MULTD	F0	F2	F4	6	9	19	20		
SUBD	F8	F6	F2	7	9	11	12		
DIVD	F10	F0	F6	8	21				
ADDD	F6	F8	F2	13	14	16	22		

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
		Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	No							
39	Divide	Yes	Div	F10	F0	F6		No	No

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
22	FU							Divide		

- 直到 61 拍，DIVD 操作完成

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6	9	19
SUBD	F8	F6	F2	7	9	11
DIVD	F10	F0	F6	8	21	61
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	dest	S1	S2	FU	FU	Fj?	Fk?		
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	Yes	Div	F10	F0	F6			No	No

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU								Divide
61									

- 62, DIVD 写回完成程序运行

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Oper	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	5	6	7
MULTD	F0	F2	F4	6	9	19
SUBD	F8	F6	F2	7	9	11
DIVD	F10	F0	F6	8	21	61
ADDD	F6	F8	F2	13	14	16

Functional unit status:

Time	Name	dest	S1	S2	FU	FU	Fj?	Fk?		
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU								
62									

记分牌流水线控制：

指令状态	进入条件	记分牌内容
流出	not Busy(FU) and not result ('D')	Busy(FU)← yes; Op(FU)← op; Fi(FU)← 'D'; Fj(FU)← 'S1'; Fk(FU)← 'S2'; Qj← Result('S1'); Qk← Result('S2'); Rj← not Qj; Rk← not Qk; Result('D')← FU;
读操作数	Rj and Rk	Rj← No; Rk← No Qj← 0; Qk← 0;
执行	功能部件操作	
写结果	$\forall f((F_j(f) \neq F_i(FU) \text{ or } R_j(f) = \text{No})$ and $(F_k(f) \neq F_i(FU) \text{ or } R_k(f) = \text{No}))$	$\forall f(\text{if } Q_j(f) = FU \text{ then } R_j(f) \leftarrow \text{Yes});$ $\forall f(\text{if } Q_k(f) = FU \text{ then } R_k(f) \leftarrow \text{Yes});$ Result(Fi(FU))← 0; Busy(FU)← No

性能受限于以下几个方面：

- 程序代码中可开发的并行性，即是否存在可以并行执行的不相关的指令
- 记分牌的容量
 - 记分牌的容量决定了流水线能在多大范围内寻找不相关指令。流水线中可以同时容纳的指令数量称为指令窗口
- 功能部件的数目和种类
 - 功能部件的总数决定了结构冲突的严重程度
- 反相关和输出相关
 - 它们引起计分牌中更多的 WAR 和 WAW 冲突

4.2.3 Tomasulo 算法

与记分牌的异同：

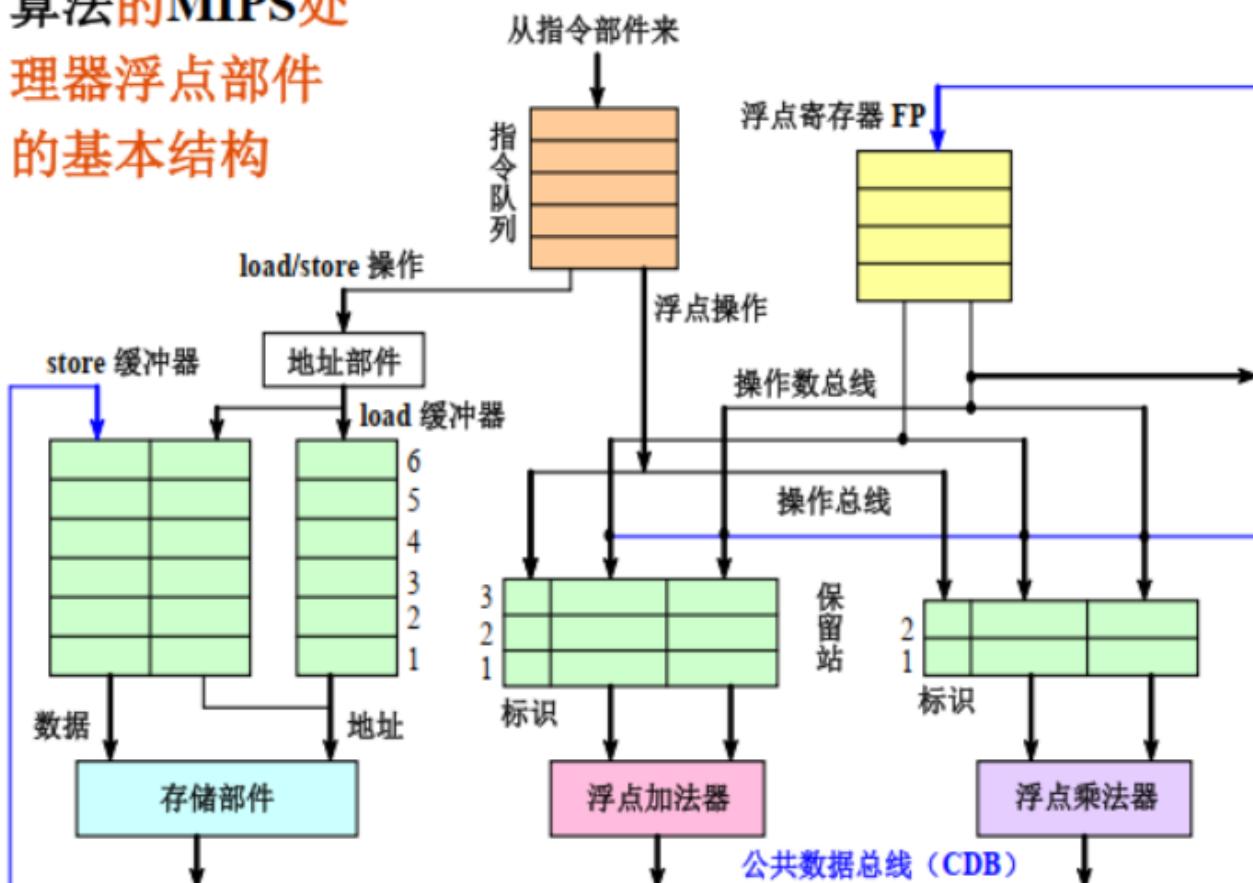
- 采用许多记分牌中的理念
- 在 Tomasulo 算法中，冲突检测和执行控制是分布的，利用该保留站实现
- Tomasulo 算法不检查 WAR 和 WAW 相关，通过算法本身消除。计算结果通过专用通道直接从功能部件进入到保留站进行缓冲，而不是直接写回到寄存器

核心思想：

- 记录并检测指令的 RAW 相关，操作数一旦准备就绪就立即执行
- 不检查 WAR 和 WAW 相关，通过换名技术消除（保留站实现）

基本结构：

➤ 基于Tomasulo 算法的MIPS处理器浮点部件的基本结构



算法流程：

- 算法流水段
 - 流出 (Issue)
 - 如果存在一个空闲的保留站，则发射指令和操作数，消除 WAR 和 WAW 相关
 - 执行 (Execute)
 - 如果两个操作数准备好就可以指令
 - 如果没有准备好，检测 CDB 以获取结果，通过推迟指令执行避免 RAW 相关
 - 结果写回 (Write result)
 - 结果通过 CDB 传给所有等待该结果的部件
- 保留站结构
 - op: 对源操作数的操作
 - Qj, Qk: 产生源操作数的保留站号

- 没有记分牌中的准备就绪标志，为零表示就绪
- **Store** 缓存区只有 **Qk** 表示产生结果的保留站号
- **Vj, Vk**: 两个源操作数的值。**Store** 缓冲区只有 **Vk** 域，用于存放要写入存储器的值。**V** 域和 **Q** 域不同时有效。
- **Busy**: 表示本保留站和相应功能部件是否空闲
- 每个寄存器和存缓冲有一个 **Qi** 域：结果要写入本寄存器或存缓冲的保留站号
- 存缓冲和取缓冲还各有一个 **Busy** 和 **Address** 域
 - **busy**: 表示缓冲是否空闲
 - **A**: 地址域，记录存或取的存储器地址
 - 存缓冲还有一个 **V** 域：写入存储器的值

实例：

- 代码：

1. LD	F6, 34(R2)
2. LD	F2, 45(R3)
3. MULTD	F0, F2, F4
4. SUBD	F8, F6, F2
5. DIVD	F10, F0, F6
6. ADDD	F6, F8, F2

- 操作延迟：

加法：2个时钟周期

乘法：10个时钟周期

除法：40个时钟周期

- 运行：

- 1, 第一个 **LD** 流出，功能部件保留站无变化，取缓冲记录，寄存器记录，
(这里隐含一个加法操作延迟)

Instruction status:

Instruction	j	k	Issue	Exec	Write
				Comp	Result
LD	F6	34+	R2	1	
LD	F2	45+	R3		
MULTD	F0	F2	F4		
SUBD	F8	F6	F2		
DIVD	F10	F0	F6		
ADDD	F6	F8	F2		

Busy	Address
Yes	34+R2
No	
No	

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU									
							Load1			

- 2, 第一个 LD 地址计算仍未结束，第二个 LD 由于取缓冲保留站还有空闲，可以流出，这里寄存器加入记录，取缓冲保留站加入记录

Instruction status:

Instruction	j	k	Issue	Exec	Write
				Comp	Result
LD	F6	34+	R2	1	
LD	F2	45+	R3	2	
MULTD	F0	F2	F4		
SUBD	F8	F6	F2		
DIVD	F10	F0	F6		
ADDD	F6	F8	F2		

Busy	Address
Yes	34+R2
Yes	45+R3
No	

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU									
						Load2	Load1			

- 3, 第一个 LD 的地址计算结束，第二个 LD 的地址计算仍在进行，MULTD 由于乘法保留站有空闲所以可以流出，其所需操作数 F4 可以直接取，而 F2 正在被占用

Instruction status:

Instruction	j	k	Exec Write		
			Issue	Comp	Result
LD	F6	34+	R2	1	3
LD	F2	45+	R3	2	
MULTD	F0	F2	F4	3	
SUBD	F8	F6	F2		
DIVD	F10	F0	F6		
ADDD	F6	F8	F2		

Busy	Address
Yes	34+R2
Yes	45+R3
No	

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- 4, 第一个 LD 写回数据, 通过 CDB 送到 F6 和刚流出的 SUBD 中, 第二个 LD 地址计算完毕, SUBD 所需操作数一个从 CDB 中送入, 一个等待 LD2 的送出

Instruction status:

Instruction	j	k	Exec Write		
			Issue	Comp	Result
LD	F6	34+	R2	1	3
LD	F2	45+	R3	2	4
MULTD	F0	F2	F4	3	
SUBD	F8	F6	F2	4	
DIVD	F10	F0	F6		
ADDD	F6	F8	F2		

Busy	Address
No	
Yes	45+R3
No	

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	Yes	SUBD	M(A1)		Load2	
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Load1				

- 5, 第二个 LD 写回数据, 通过 CDB 送入 F2, MULTD1, ADD1 中, DIVD 由于乘法保留站有空闲所以流出操作数未就绪记录占用信息, 并且 MULTD 和 ADD 所需数据都准备完毕, 立即执行, 这一点和记分牌不同

Instruction status:

Instruction	<i>j</i>	<i>k</i>	Exec Write			Busy	Address	
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4				
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2					

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				<i>V_j</i>	<i>V_k</i>	<i>Q_j</i>	<i>Q_k</i>
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
	5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2		

- 6, 加法保留站有空闲所以 ADDD 可以流出

Instruction status:

Instruction	<i>j</i>	<i>k</i>	Exec Write			Busy	Address	
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4				
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6				

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				<i>V_j</i>	<i>V_k</i>	<i>Q_j</i>	<i>Q_k</i>
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
	6	FU	Mult1	M(A2)		Add2	Add1	Mult2		

- 7, SUBD 执行完成

Instruction status:

Instruction	j	k	Issue	Exec		Write		Busy	Address
				Comp	Result				
LD	F6	34+	R2	1	3	4		Load1	No
LD	F2	45+	R3	2	4	5		Load2	No
MULTD	F0	F2	F4	3				Load3	No
SUBD	F8	F6	F2	4	7				
DIVD	F10	F0	F6	5					
ADDD	F6	F8	F2	6					

Reservation Stations:

Time	Name	Busy	Op	S1		S2		RS	
				Vj	Vk	Qj	Qk		
0	Add1	Yes	SUBD	M(A1)	M(A2)				
	Add2	Yes	ADDD		M(A2)	Add1			
	Add3	No							
8	Mult1	Yes	MULTD	M(A2)	R(F4)				
	Mult2	Yes	DIVD		M(A1)	Mult1			

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- 8, SUBD 写回通过 CDB 送入寄存器和 ADDD 指令，ADDD 指令由于操作数都准备好了所以开始执行

Instruction status:

Instruction	j	k	Issue	Exec		Write		Busy	Address
				Comp	Result				
LD	F6	34+	R2	1	3	4		Load1	No
LD	F2	45+	R3	2	4	5		Load2	No
MULTD	F0	F2	F4	3				Load3	No
SUBD	F8	F6	F2	4	7	8			
DIVD	F10	F0	F6	5					
ADDD	F6	F8	F2	6					

Reservation Stations:

Time	Name	Busy	Op	S1		S2		RS	
				Vj	Vk	Qj	Qk		
	Add1	No							
2	Add2	Yes	ADDD	(M-M)	M(A2)				
	Add3	No							
7	Mult1	Yes	MULTD	M(A2)	R(F4)				
	Mult2	Yes	DIVD		M(A1)	Mult1			

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

- 9, 进行操作

Instruction status:

Instruction	j	k	Issue	Exec Write			Busy	Address
				Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6				

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
	9	FU	Mult1	M(A2)		Add2	(M-M)	Mult2		

- 10, ADDD 操作执行完毕

Instruction status:

Instruction	j	k	Issue	Exec Write			Busy	Address
				Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10			

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
	10	FU	Mult1	M(A2)		Add2	(M-M)	Mult2		

- 11, ADDD 指令写回，因为 DIVD 指令通过保留站换名已经解除了与 ADDD 的 WAR 相关

Instruction status:

Instruction	j	k	Exec Write			Busy	Address	
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1	M(A2)			M-M+N	(M-M)	Mult2		

- 一直到 15 拍，MULTD 执行完毕

Instruction status:

Instruction	j	k	Exec Write			Busy	Address	
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3	15		Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1	M(A2)			M-M+N	(M-M)	Mult2		

- 16，MULTD 写回，通过 CDB 送入寄存器和 DIVD，DIVD 开始执行

Instruction status:

Instruction	j	k	Issue	Exec Write			Busy	Address
				Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3	15	16	Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	M*F4	M(A2)		(M-M+V(M-M))	Mult2				

- 直到 56 拍， DIVD 执行完毕

Instruction status:

Instruction	j	k	Issue	Exec Write			Busy	Address
				Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3	15	16	Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5	56			
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+V(M-M))	Mult2				

- 57， DIVD 写回，通过 CDB 送寄存器

Instruction status:

Instruction	j	k	Issue	Exec	Write
				Comp	Result
LD	F6	34+	R2	1	3 4
LD	F2	45+	R3	2	4 5
MULTD	F0	F2	F4	3	15 16
SUBD	F8	F6	F2	4	7 8
DIVD	F10	F0	F6	5	56 57
ADD	F6	F8	F2	6	10 11

Busy	Address
Load1	No
Load2	No
Load3	No

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD M*F4 M(A1)				

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)	(M-M+N)	(M-M)	Mult2			

循环实例这里不再过多赘述，只用记住一点：当分支指令没有执行完时，后续指令不允许流出

优缺点：

- 优点
 - 分布式硬件冲突检测。
 - 利用寄存器换名，彻底消除 WAW 和 WAR 这两种名相关
 - 如果多个保留站等待同一个操作数，当操作数在 CDB 上广播时，他们可以同时获得所需的数据
 - 对于存储器访问，动态存储器地址判别技术可解决 RAW 冲突（取操作数时判断）、WAR 和 WAW 冲突（存操作数时判断）
 - 能够达到很高的性能。
- 缺点
 - 高复杂性：需要大量硬件
 - 存在瓶颈：单个公共数据总线（CDB）引发竞争
 - 额外的 CDB：在每个保留站上需要为每条 CDB 设置重复的硬件接口
 - 为了保证正确的异常行为，对指令的执行有一个限制：一旦有一条分支指令还没有执行完，其后的指令是不允许进入执行段

指令乱序增大了异常处理的复杂度

- 不精确异常（Imprecise Exception）：当指令 i 导致发生异常时，处理机的现场（状态）与严格按程序顺序执行不同。
- 精确异常（Precise Exception）：处理机现场跟严格按程序顺序执行时指令 i 的现场相同。
- 不精确异常产生的原因：

- 流水线可能已经执行完按程序顺序是位于指令 i 之后的指令
- 流水线可能还没完成按程序顺序是指令 i 之前的指令

4.3 控制相关的动态解决技术

动态预测的理由

- n 流出的处理器加速上限为 n
- Amdahl 定律提示：在较低 CPI 机器上，控制相关导致的空转对机器性能影响大

前述的静态策略：分支延迟槽

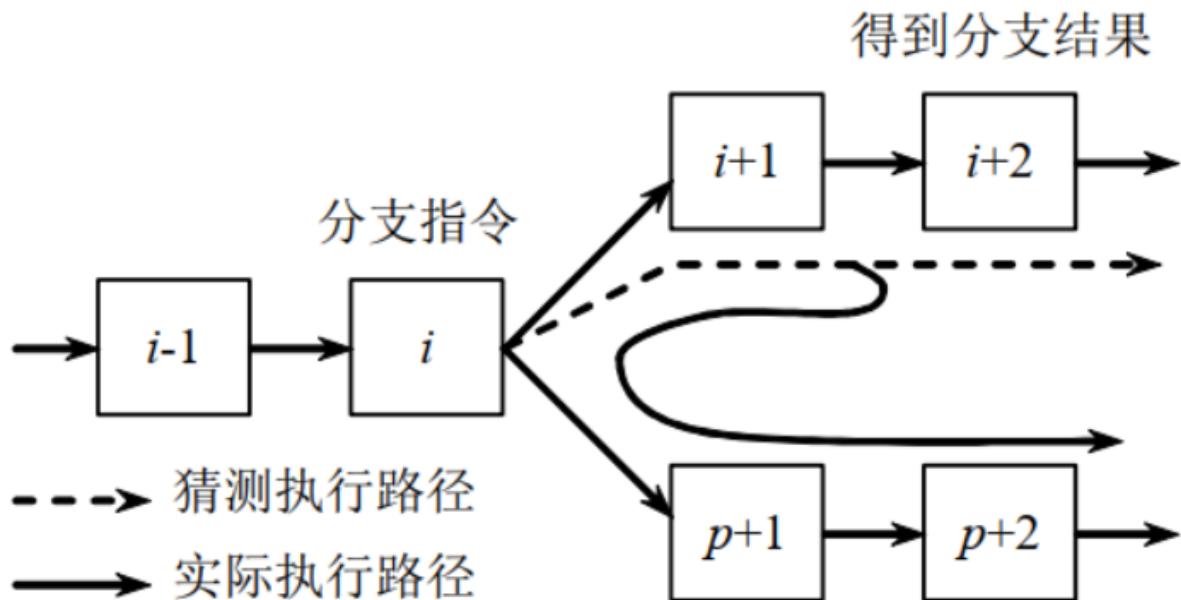
分支预测：

- 预测分支是否成功
- 执行目标指令

4.3.1 分支预测缓冲 BPB

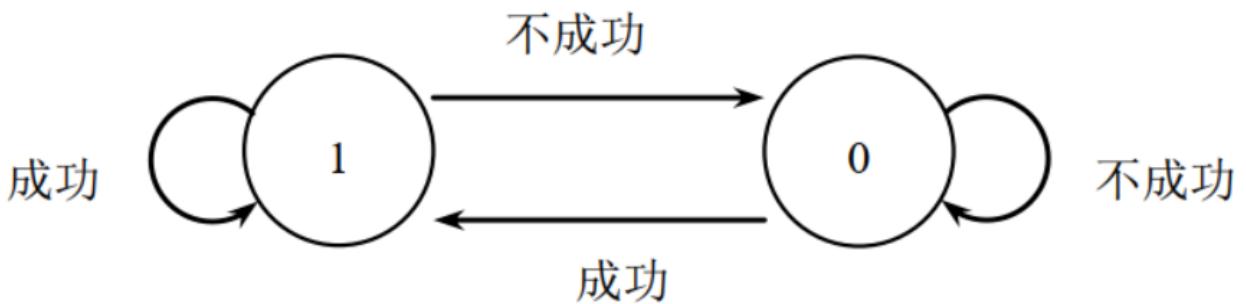
分支预测缓冲是一个小的存储器阵列

- 每个单元最小可以只有 1 位，记录最近一次分支是否成功的信息
- 预测位为 1 时，表示预测分支成功，并从目标位置开始取指令
- 在预测错误时，要作废已经预取和分析的指令，恢复现场，并从另一条分支路径重新取指令。



一位 BPB:

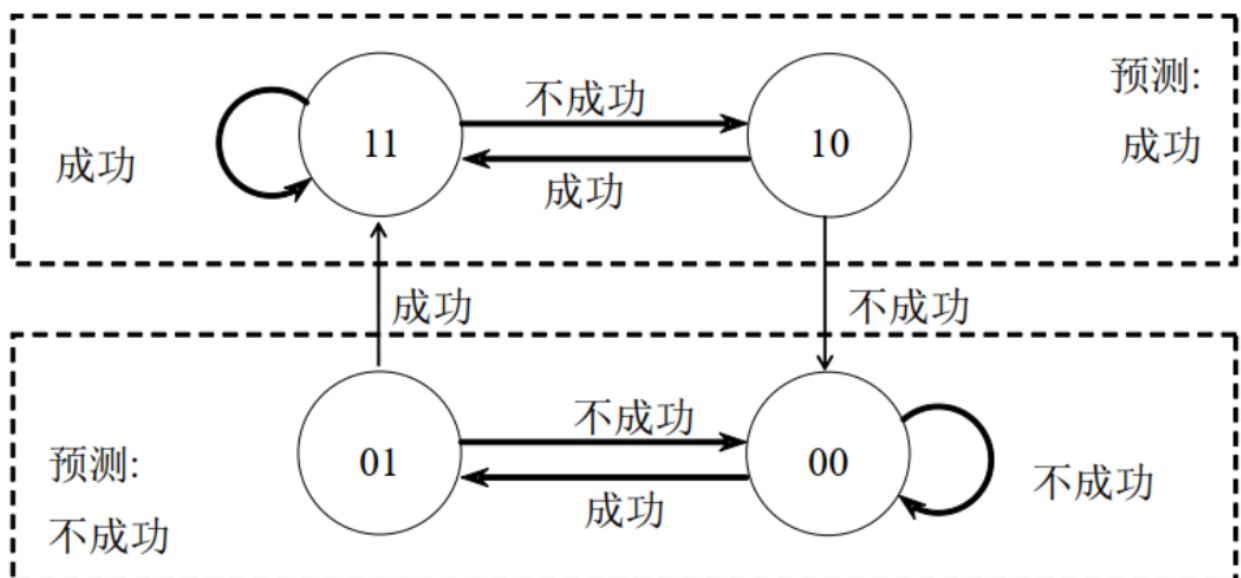
- 状态转移图



- 当分支不成功时，将会发生连续两侧预测出错

2 位 BPB

- 状态转移图



- 容忍出两次错误

实现方案：

- 实现一个小而特殊的“cache”，利用指令地址进行索引，在 IF 流水段访问
- 为指令 cache 中每一块增加附加位，与指令一起取出

对于改进 MIPS，BPB 不起作用

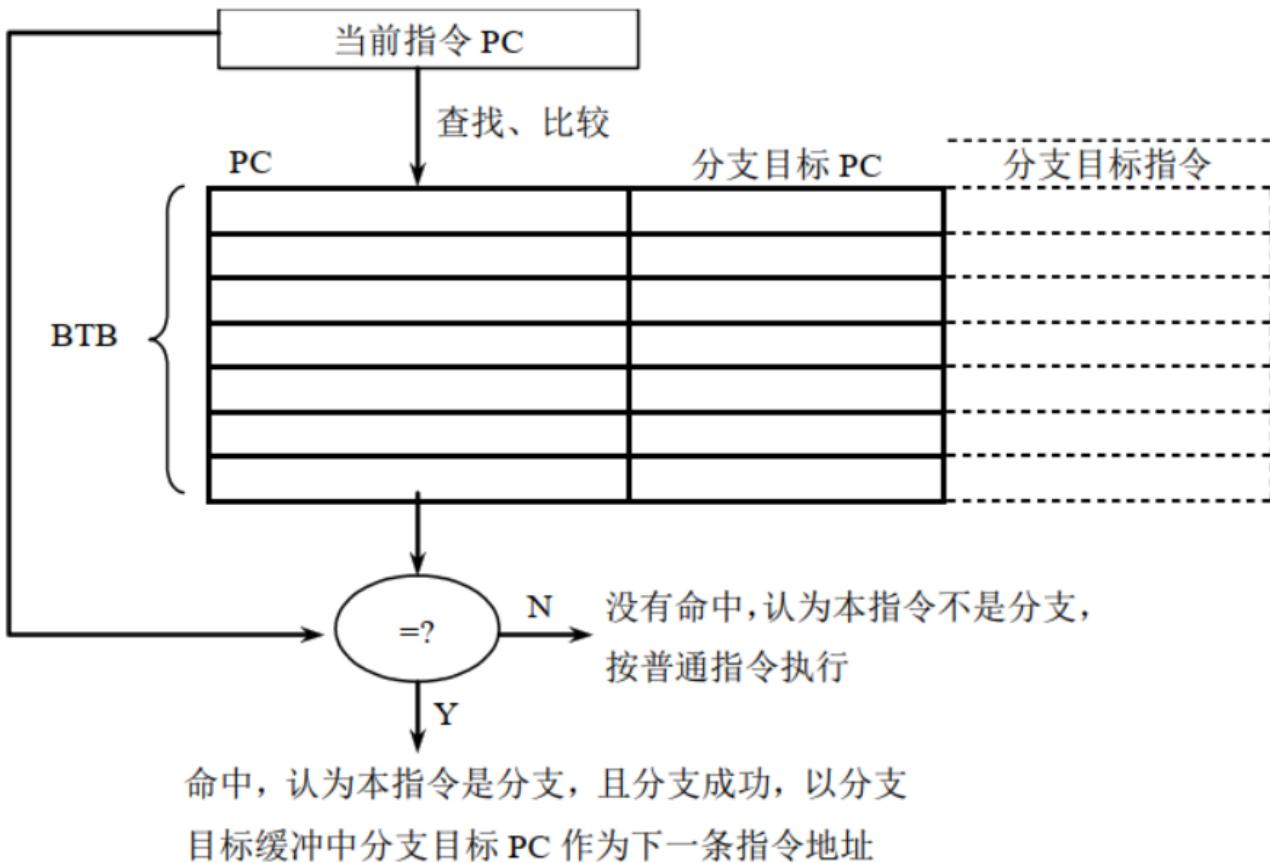
4.3.2 分支目标缓冲 BTB

BTB 每个单元包括：

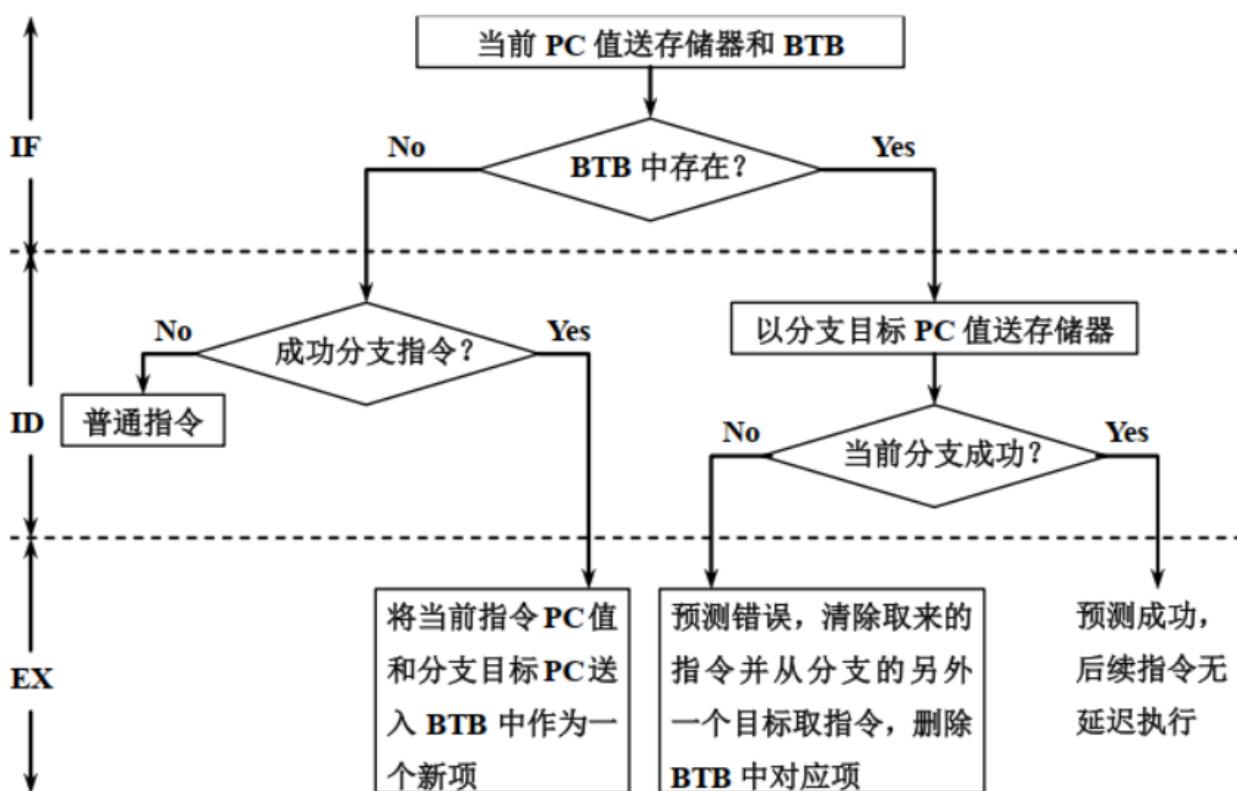
- 分支指令地址

- 分支目标地址
- 分支预测标识

结构:



执行过程:



分支预测局限性：

- 准确性 80%-90%
- 性能依赖于
 - 程序类型
 - 缓冲区大小
- 预测失效开销的优化
 - 预取不同分支路径指令：但是会引入其他开销，比如存储系统端口加倍

4.3.3 基于硬件的前瞻执行

基本思想：

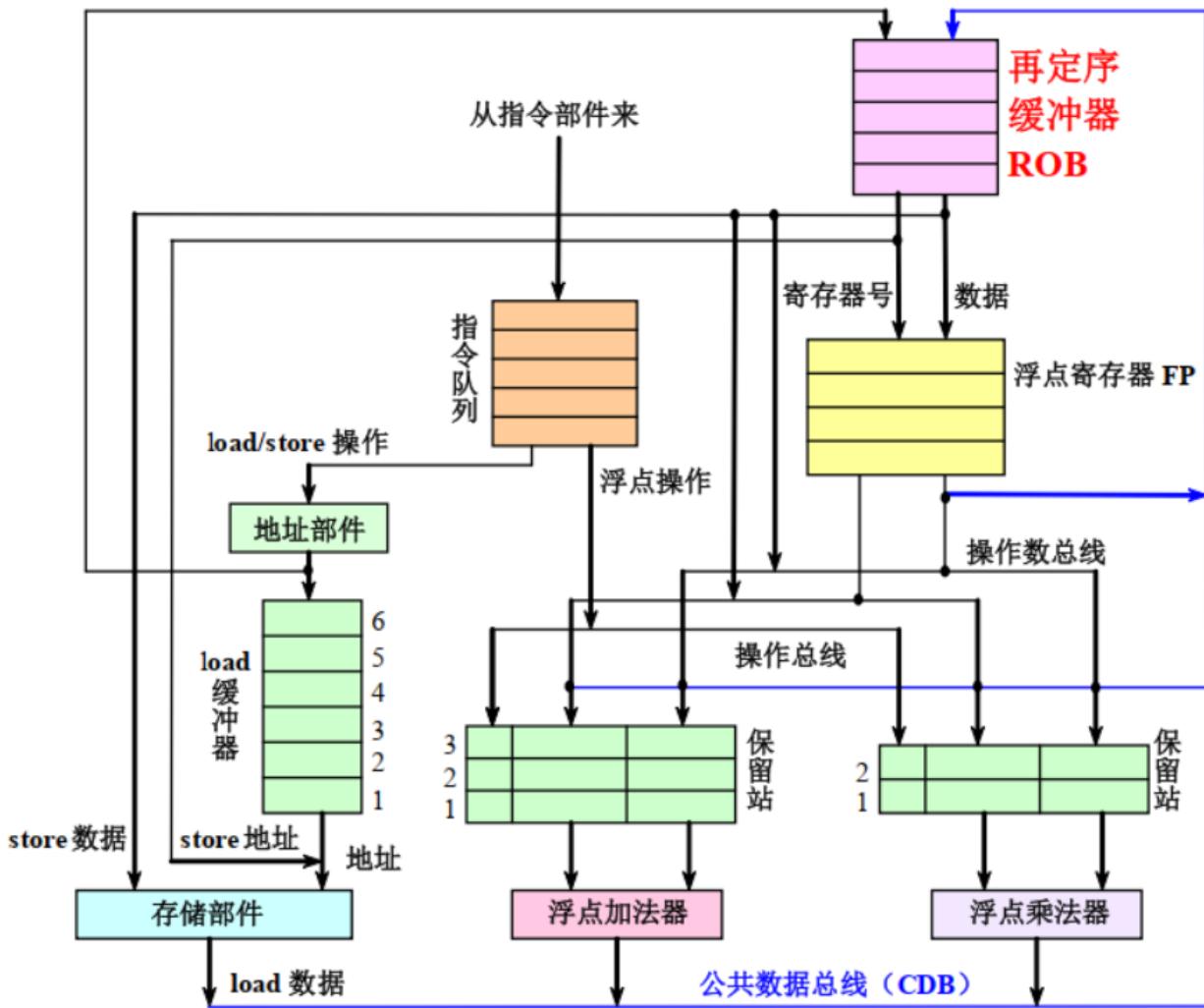
- 对分支指令的结果进行猜测，并假设这个猜测总是对的，然后按这个猜测结果继续取、流出和执行后续的指令。
- 执行指令的结果不是写回到寄存器或存储器，而是写入一个称为再定序缓冲器 ROB (ReOrder Buffer) 中。等到相应的指令得到“确认”(commit)（即确实是应该执行的）之后，才将结果写入寄存器或存储器。

结合了三种思想：

- 采用动态预测选择后续执行的指令
- 在控制相关的结果还未出来时，前瞻的执行后续指令
- 对基本块采用动态调度

扩充 tomasulo 算法就可以支持前瞻执行

- 把写结果再分成写结果和指令确认段
 - 写结果段：
 - 把结果写入 ROB 中
 - 通过 CDB 传送数据
 - 确认段：
 - 如果猜测正确：把 ROB 中结果写入寄存器
 - 猜测错误：刷新 ROB 从另一条路径重新执行
- 允许乱序执行，但必须顺序确认
- 部件扩充



- ROB 字段

- 指令类型: 分支、**store**、寄存器操作
- 状态: 支出指令是否完成以及数据已经就绪
- 目标地址: 写入的寄存器号或内存地址
- 数据值: 要写入的值

- 保留站增加目标地址字段: 对应 ROB 的编号

- 算法阶段:

- 流出: **RS** 和 **ROB** 有空闲就发射指令
- 执行
- 写结果
- 确认

实例:

- 指令

LD	F6, 34(R2)
LD	F2, 45(R3)
MULTD	F0, F2, F4
SUBD	F8, F6, F2
DIVD	F10, F0, F6
ADDD	F6, F8, F2

- 时延
 - LD: 1
 - ADDD/SUBD: 2
 - MULTD: 9
 - DIVD: 40
- 1, 第一个 LD 流出, 写入 ROB 中

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Reservation Stations
0	Add1	No							
0	Add2	No							
0	Add3	No							
0	Mult1	No							
0	Mult2	No							

Entry	Busy	Instruction	State	Destination	Value	Busy	Address
1	Yes	LD F6, 34(R2)	Issue	F6		Load1	Yes 34+Regs[R2]
2						Load2	
3						Load3	
4							
5							
6							
7							
8							
9							
10							

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #				#1					
Busy	no	no	no	Yes	no	no	no		no

- 2, 第二个 LD 流出

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	
0	Add1	No							Reservation Stations
0	Add2	No							
0	Add3	No							
0	Mult1	No							
0	Mult2	No							

Entry	Busy	Instruction	State	Destination	Value	Busy	Address
head → 1	Yes	LD F6, 34(R2)	Ex1	F6		Load1	Yes 34+Regs[R2]
tail → 2	Yes	LD F2, 45(R3)	Issue	F2		Load2	Yes 45+Regs[R3]
3						Load3	
4							
5							
6							
7							
8							
9							
10							

Reorder Buffer

F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#2		#1					
Busy	no	Yes	no	Yes	no	no	no	no

- 3, 第一个 LD 写回, 注意: 这里写入到 ROB 中, 并不写入寄存器中, 因为还未确认。MULTD 流出

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	
0	Add1	No							Reservation Stations
0	Add2	No							
0	Add3	No							
0	Mult1	Yes	Mult		Regs[F4]	#2		#3	
0	Mult2	No							

Entry	Busy	Instruction	State	Destination	Value	Busy	Address
head → 1	Yes	LD F6, 34(R2)	write	F6	Mem[load1]	Load1	No
2	Yes	LD F2, 45(R3)	Ex1	F2		Load2	Yes 45+Regs[R3]
tail → 3	Yes	MULT F0, F2, F4	Issue	F0		Load3	
4							
5							
6							
7							
8							
9							
10							

Reorder Buffer

F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3	#2		#1				
Busy	Yes	Yes	no	Yes	no	no	no	no

- 4, 第一个 LD 确认, 结果从 ROB 中写入寄存器中。第二个 LD 写回, 数据通过 CDB 送入 ROB 和 MULTD 和流出的 SUBD, MULTD 开始执行, SUBD 不执行的

原因是这时处于流出段，下一拍才开始执行

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]			#4	Reservation Stations
0	Add2	No							
0	Add3	No							
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3	
0	Mult2	No							

Entry	Busy	Instruction	State	Destination	Value	Load1	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	
head	→ 2	Yes	LD F2, 45(R3)	write	F2	Mem[load2]	No
tail	→ 4	Yes	MULT F0, F2, F4	EX1	F0		
5							
6							
7							
8							
9							
10							

F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3	#2		#4				
Busy	Yes	Yes	no	no	Yes	no	no	no

- 5, 第二个 LD 确认数据从 ROB 写回寄存器，DIVD 流出，SUBD 开始执行

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]			#4	Reservation Stations
0	Add2	No							
0	Add3	No							
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3	
0	Mult2	Yes	DIV		Regs[F6]			#5	

Entry	Busy	Instruction	State	Destination	Value	Load1	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	
head	→ 3	Yes	MULT F0, F2, F4	Ex2	F0	Load3	
tail	→ 5	Yes	SUBD F8, F6, F2	Ex1	F8		
6							
7							
8							
9							
10							

F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#4		#5		
Busy	Yes	no	no	no	Yes	Yes	no	no

- 6, ADDD 流出，SUBD 执行完毕

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]		#4	Reservation Stations
0	Add2	Yes	Add		Regs[F2]	#4		#6
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]		#3	
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Entry	Busy	Instruction	State	Destination	Value	Load1	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	Load1	No
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	Load2	No
head	→ 3	Yes MULT F0, F2, F4	Ex3	F0		Load3	
tail	→ 6	Yes SUBD F8, F6, F2	Ex2	F8			
7							
8							
9							
10							
		F0 F2 F4	F6	F8 F10	F12 ... F30		
Reorder #		#3		#6	#4	#5	
Busy	Yes	no no	Yes	Yes Yes	Yes	no	no

- 7, SUBD 写回, 通过 CDB 送 ROB 和 ADDD, ADDD 开始执行

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						Reservation Stations
0	Add2	Yes	Add	#4	Regs[F2]			#6
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]		#3	
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Entry	Busy	Instruction	State	Destination	Value	Load1	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	Load1	No
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	Load2	No
head	→ 3	Yes MULT F0, F2, F4	Ex4	F0		Load3	
tail	→ 6	Yes SUBD F8, F6, F2	write	F8	F6 - #2		
7							
8							
9							
10							
		F0 F2 F4	F6	F8 F10	F12 ... F30		
Reorder #		#3		#6	#4	#5	
Busy	Yes	no no	Yes	Yes Yes	Yes	no	no

- 8, SUBD 无法确认, 因为前面存在未确认的指令, 而确认必须按序, ADDD 执行完毕

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Busy	Address
0	Add1	No								Reservation Stations
0	Add2	Yes	Add	#4	Regs[F2]			#6		
0	Add3	No								
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3		
0	Mult2	Yes	DIV		Regs[F6]	#3		#5		

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No		
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No		
head → 3	Yes	MULT F0, F2, F4	Ex5	F0				
4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2			
5	Yes	DIVD F10, F0, F6	Issue	F10				
tail → 6	Yes	ADDD F6, F8, F2	Ex2	F6				
7								
8								
9								
10								

F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5		
Busy	Yes	no	no	Yes	Yes	Yes	no	no

- 9, ADDD 写回

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Busy	Address
0	Add1	No								Reservation Stations
0	Add2	Yes	Add	#4	Regs[F2]			#6		
0	Add3	No								
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3		
0	Mult2	Yes	DIV		Regs[F6]	#3		#5		

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No		
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No		
head → 3	Yes	MULT F0, F2, F4	Ex6	F0				
4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2			
5	Yes	DIVD F10, F0, F6	Issue	F10				
tail → 6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2			
7								
8								
9								
10								

F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5		
Busy	Yes	no	no	Yes	Yes	Yes	no	no

- 直到第 12 拍，MULTD 执行完毕

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Busy	Address
0	Add1	No								Reservation Stations
0	Add2	No								
0	Add3	No								
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3		
0	Mult2	Yes	DIV		Regs[F6]	#3		#5		
head →										
tail →										
F0 F2 F4 F6 F8 F10 F12 ... F30										
Reorder #3 #6 #4 #5										
Busy Yes no no Yes Yes Yes no no										

- 13, MULTD 写回, 通过 CDB 送 DIVD 和 ROB, DIVD 开始执行

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Busy	Address
0	Add1	No								Reservation Stations
0	Add2	No								
0	Add3	No								
0	Mult1	No								
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5		
head →										
tail →										
F0 F2 F4 F6 F8 F10 F12 ... F30										
Reorder #3 #6 #4 #5										
Busy Yes no no Yes Yes Yes no no										

- 14, MULTD 确认, ROB 写入寄存器

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	
0	Add1	No							Reservation Stations
0	Add2	No							
0	Add3	No							
0	Mult1	No							
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5	
		Busy	Address						
Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3	
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No			
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No			
3	No	MULT F0, F2, F4	commit	F0	#2 x Regs[F4]				
head → 4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2				
tail → 5	Yes	DIVD F10, F0, F6	Ex2	F10					
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2				Reorder Buffer
7									
8									
9									
10									
F0	F2	F4	F6	F8	F10	F12	...	F30	
Reorder #			#6	#4	#5				
Busy	No	no	no	Yes	Yes	Yes	no	no	

- 15, SUBD 确认
- 直到 52 拍, DIVD 执行完毕
- 53, DIVD 写回
- 54, DIVD 确认
- 55, ADDD 确认

4.4 多指令流出技术

- 多指令流出处理器
 - 实现一个时钟周期内流出多条指令时
 - 达到 CPI 小于 1
- 多流出处理器 2 种基本结构
 - 超标量 (Superscalar)
 - 超标量每个时钟周期流出的指令数不定
 - 可以编译器静态调度, 也可以硬件动态调度
 - 超长指令字 (VLIW, Very long Instruction Word)
 - 每个时钟周期流出的指令数是固定的, 只能通过编译静态调度

技术	流出结构	冲突检测	调度	主要特点	处理机实例
超标量 (静态)	动态	硬件	静态	按序执行	Sun UltraSPARCII/III
超标量 (动态)	动态	硬件	动态	部分乱序执行	IBM Power2
超标量 (前瞻)	动态	硬件	带有前瞻的动态调度	带有前瞻的乱序执行	Pentium III/4, MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64III
VLIW /LIW	静态	软件	静态	流出包指令之间没有冲突	Trimedia, i860
EPIC	主要是静态	主要是软件	主要是静态	相关性被编译器显式地标记出来	Itanium (IA64)

4.4.1 静态超标量

每个周期发送 1-8 条不存在相关的指令

假设有一个超标量机器

- 每个周期可以流出两条指令
 - 一条取、存、分支、整数指令
 - 一条浮点指令
- 理想情况

指令	流水线工作情况							
	1	2	3	4	5	6	7	8
整数指令	IF	ID	EX	MEM	WB			
浮点指令	IF	ID	EX	MEM	WB			
整数指令		IF	ID	EX	MEM	WB		
浮点指令		IF	ID	EX	MEM	WB		
整数指令			IF	ID	EX	MEM	WB	
浮点指令			IF	ID	EX	MEM	WB	
整数指令				IF	ID	EX	MEM	WB
浮点指令				IF	ID	EX	MEM	WB

- 存在的技术问题
 - 每个时钟周期流出两条指令意味着同时取两条指令（64位），译码两条指令（64位）
 - 假设：指令按要求组合成对，且与64位边界对齐，整数指令顺序在前
 - 需要使得浮点部件流水化或增加相关检测部件来减少结构相关
 - 另一个限制超标量流水线性能发挥的障碍是取操作和分支操作的延迟
 - 取操作指令的结果不能在本周期和下一个周期使用
 - 分支指令肯定是指令组合的第一条指令，影响配对指令和后续两条指令，分支延迟也变为3条指令
- 实例：一般需要更有效的编译技术、硬件调度技术和更复杂的指令译码技术才能有效利用超标量技术的并行度
 - 代码

LOOP: LD F0, 0(R1) ;F0=数组元素
ADDD F4, F0, F2 ;与F2寄存器中的标量相加
SD 0(R1), F4 ;存结果
SUBI R1, R1, #8 ;R1寄存器值减8
BNEZ R1, LOOP ;如果R1值不为0，则跳转

- 循环展开5次进行调度

整数指令		浮点指令		时钟周期	
Loop:	LD	F0(R1)		1	
	LD	F6,-8(R1)		2	
	LD	F10,-16(R1)	ADDD	F4,F0,F2	3
	LD	F14,-24(R1)	ADDD	F8,F6,F2	4
	LD	F18,-32(R1)	ADDD	F12,F10,F2	5
	SD	0(R1).F4	ADDD	F16,F14,F2	6
	SD	-8(R1).F8	ADDD	F20,F18,F2	7
	SD	-16(R1).F12		8	
	SD	-24(R1).F16		9	
	SUBI	R1,R1,#40		10	
	BNEZ	R1,Loop		11	
	SD	8(R1).F20		12	

超标量可以获得更好的性能，但是硬件复杂性大幅增加

4.4.2 动态多指令流出

多指令流出+动态调度

对于采用了Tomasulo算法和多流出技术的MIPS流水线，考虑以下简单循环的执行，列表表示前面三遍循环的各指令流出、开始执行、访存、结果写到CDB的时间。

LOOP:	LD	F0, 0(R1)	;F0=数组元素.
	ADDD	F4, F0, F2	;与F2寄存器中的标量相加
	SD	F4, 0(R1)	;存结果
	SUBI	R1, R1, #8	;R1寄存器值减8
	BNEZ	R1, LOOP	;如果R1值不为0，则跳转

- (1) 无论是否相关，每个时钟周期流出一条整数指令和一条浮点数指令；
- (2) 有一个整数部件，用于整数运算和地址计算；有一个独立的浮点功能部件；
- (3) 指令流出和写结果各占用1个时钟周期；
- (4) 有1个具有独立分支预测能力的分支预测部件，分支指令只能单独流出，没有分支延迟；
- (5) 因为写结果占用1个时钟周期，所以产生结果的延迟为：整数运算1个周期，存储器取数操作2个周期，浮点运算3个周期。

遍数	指令	流出	执行	访存	写CDB	说明
1	LD F0,0(R1)	1	2	3	4	流出第一条指令
1	ADDD F4,F0,F2	1	5		8	等待LD的结果
1	SD 0(R1),F4	2	3	9		等待ADDD的结果
1	SUBI R1,R1,#-8	2	4		5	等待计算的ALU
1	BNEZ R1,Loop	3	6			等待SUBI的结果
2	LD F0,0(R1)	4	7	8	9	等待BNEZ完成
2	ADDD F4,F0,F2	4	10		13	等待LD的结果
2	SD 0(R1),F4	5	8	14		等待ADDD的结果
2	SUBI R1,R1,#-8	5	9		10	等待ALU
2	BNEZ R1,Loop	6	11			等待SUBI的结果
3	LD F0,0(R1)	7	12	13	14	等待BNEZ完成
3	ADDD F4,F0,F2	7	15		18	等待LD的结果
3	SD 0(R1),F4	8	13	19		等待ADDD的结果
3	SUBI R1,R1,#-8	8	14		15	等待ALU
3	BNEZ R1,Loop	9	16			等待SUBI的结果

可以看出：

- 每3个时钟周期就执行一个新循环迭代，每个循环5条指令。

$$IPC = 5/3 = 1.67 \text{ 条/拍}$$

- 虽然指令的流出率比较高，但是执行效率并不是很高。
 - 16拍共执行15条指令，
 - 平均指令执行速度为 $15/16 = 0.94 \text{ 条/拍}$ 。
- 原因是浮点运算少，ALU部件成了瓶颈。

解决方法：增加一个加法器，把ALU功能和地址运算功能分开。

- 上述双流出动态调度流水线的性能受限于以下3个因素：
 - 整数部件和浮点部件的工作负载不平衡，没有充分发挥出浮点部件的作用。
 - 应该设法减少循环中整数型指令的数量。
 - 每个循环迭代中的控制开销太大。
 - 5条指令中有两条指令是辅助指令；
 - 应该设法减少或消除这些指令。
 - 控制相关使得处理器必须等到分支指令的结果出来后才能开始下一条LD指令的执行。

4.4.3 超长指令字

一条指令包含多个操作

例子

- 每个指令字包含
 - 两个访存操作
 - 两个浮点操作
 - 一个整数或分支操作
- 代码

LOOP:	LD	F0, 0(R1)	;F0=数组元素.
	ADDD	F4, F0, F2	;与F2寄存器中的标量相加
	SD	F4, 0(R1)	;存结果
	SUBI	R1, R1, #8	;R1寄存器值减8
	BNEZ	R1, LOOP	;如果R1值不为0，则跳转

- 循环展开五次

访存操作 1	访存操作 2	浮点操作 1	浮点操作 2	整数操作/分支
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14,-24(R1)			
LD F18,-32(R1)		ADDD F4,F0,F2	ADDD F8,F6,F2	
		ADDD F12,F10,F2	ADDD F16,F14,F2	
		ADDD F20,F18,F2		
SD 0(R1),F4	SD -8(R1),F8			
SD -16(R1),F12	SD -24(R1),F16			SUBI R1,R1,#40
SD 8(R1),F20				BNEZ R1,Loop

□ 5条结果在8个时钟周期中完成计算

- 每条结果花费1.3时钟周期
- 有17/40的指令槽被放入了有效的操作

- 展开 7 次

Mem Ref1	Mem Ref2	FP1	FP2	Int/branch
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14,-24(R1)			
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2	
		ADDD F20,F18,F2	ADDD F24,F22,F2	
SD F4,0(R1)	SD F8,-8(R1)	ADDD F28,F26,F2		
SD F12,-16(R1)	SD F16,-24(R1)			SUBI R1,R1,#56
SD F20,24(R1)	SD F24,16(R1)			
SD F28,8(R1)				BNEZ R1,Loop

- 9拍产生7个结果
 - 每个结果1.29拍
 - 比前面的超标量，每个结果2.4拍，快接近2倍
- 9拍里面执行了23个操作
 - 2.5操作/拍
 - 指令槽利用率不高，只有51% (23/45)
- 使用大量寄存器
 - 7个循环，共计使用 $2 \times 7 + 1 = 15$ 个64位浮点寄存器

- 展开三次无调度

Mem Ref1	Mem Ref2	FP1	FP2	Int/branch
LD F0, 0(R1)	LD F6,-8(R1)			
LD F10, -16(R1)				
		ADDD F4,F0,F2	ADDD F8,F6,F2	
		ADDD F12,F10,F2		
				SUBI R1, R1, #24
SD F4,24(R1)	SD F8,16(R1)			
SD F12,8(R1)				
				BNEZ R1, Loop

- 8拍3个结果，每个结果2.66拍
- 指令槽利用率: $11/40 = 27.5\%$
- 寄存器： $3 \times 2 + 1 = 7$ 个64位浮点寄存器

- 展开 10 次有调度

Mem Ref1	Mem Ref2	FP1	FP2	Int/branch
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14,-24(R1)			
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	
LD F26,-48(R1)	LD F30,-56(R1)	ADDD F12,F10,F2	ADDD F16,F14,F2	
LD F34,-64(R1)	LD F38,-72(R1)	ADDD F20,F18,F2	ADDD F24,F22,F2	SUBI R1,R1,#80
SD F4,80(R1)	SD F8,72(R1)	ADDD F28,F26,F2	ADDD F32,F30,F2	
SD F12,64(R1)	SD F16,56(R1)	ADDD F36,F34,F2	ADDD F40,F38,F2	
SD F20,48(R1)	SD F20,40(R1)			
SD F20,32(R1)	SD F20,24(R1)			
SD F20,16(R1)	SD F20,8(R1)			BNEZ R1,Loop

- 10拍10个结果，每个结果使用1拍
- 指令槽利用率: $36/50 = 72\%$
- 寄存器需求: $10 \times 2 + 1 = 21$ 个64位浮点寄存器

技术难题

- 从线性代码片段中产生足够的操作需要进行激进的循环展开，这增大了代码大小
- 无论指令是否被充满，没有被使用的功能单元也在指令字编码过程中占据了相应的位。将近一半的指令是被浪费掉的
 - 在主存中压缩指令，在 cache 中解压缩指令
- VLIW 带来了二进制代码兼容性问题

- 采用机器代码翻译或仿真模拟的方法解决移植的问题

多指令流出的技术难题

- 程序所固有的指令级并行性
- 硬件实现上的困难
- 超标量和超长指令字处理器固有的技术限制
- 设计难点
 - 访存开销
 - 硬件复杂性
 - 编译器技术

第五章 存储层次

5.1 存储器的层次结构

5.1.1 多级存储层次

存储器的三个主要指标：

- 容量、大
- 速度、快
- 每位价格、低

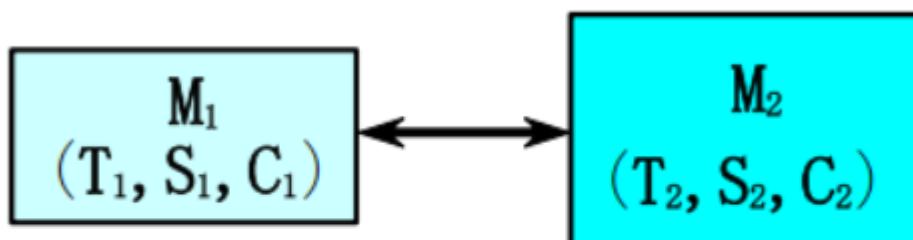
这三个要求时互相矛盾的

- 假设第*i*个存储器 M_i 的访问时间为 T_i , 容量为 S_i , 平均每位价格为 C_i , 则
 - 访问时间: $T_1 < T_2 < \dots < T_n$
 - 容量: $S_1 < S_2 < \dots < S_n$
 - 平均每位价格: $C_1 > C_2 > \dots > C_n$
- 整个存储系统要达到的目标: 从CPU来看, 该存储系统的速度接近于 M_1 的, 而容量和每位价格都接近于 M_n 的。
 - 存储器越靠近CPU, 则CPU对它的访问频度越高, 而且最好大多数的访问都能在 M_1 完成。

5.1.2 存储层次的性能参数

考虑两级存储层次

- M_1 的参数: S_1, T_1, C_1
- M_2 的参数: S_2, T_2, C_2



其中, **S**: 存储容量
T: 访问时间
C: 每位价格

- 存储容量 S 整个系统的存储容量 $S = S_2$
- 每位价格 C $C = \frac{C_1S_1 + C_2S_2}{S_1 + S_2}$

- 命中率 H 和失效率 F
 - 命中率: CPU 访存时, 在 M_1 中找到信息的概率 $H = \frac{N_1}{N_1+N_2}$
 - 失效率: $F = 1 - H$
- 平均访存时间 T_A $T_A = HT_1 + (1 - H)(T_1 + T_M) = T_1 + (1 - H)T_M = T_1 + FT_M$
 - 第一次访存时间
 - 命中: T_1
 - 不命中: $T_1 + T_M, T_M = T_2 + T_B$
 - T_M : 失效开销, 从向 M_2 发出请求到数据块调入 M_1 中的时间
 - T_2 : M_2 的访问时间
 - T_B : 从 M_2 把一个数据送入 M_1 中的时间

5.1.3 两种存储层次的关系

- 三级存储系统
 - Cache
 - 主存
 - 辅存
- 两种存储层次
 - Cache-主存
 - 主存-辅存

比较项目 存储层次	“Cache - 主存” 层次	“主存 - 辅存” 层次
目的	为了弥补主存速度的不足	为了弥补主存容量的不足
存储管理实现	主要由专用硬件实现	主要由软件实现
访问速度的比值 (第一级和第二级)	几比一	几百比一
典型的块(页)大小	几十个字节	几百到几千个字节
CPU对第二级的 访问方式	可直接访问	均通过第一级
失效时CPU是否切换	不切换	切换到其他进程

5.1.4 存储层次的 4 个问题

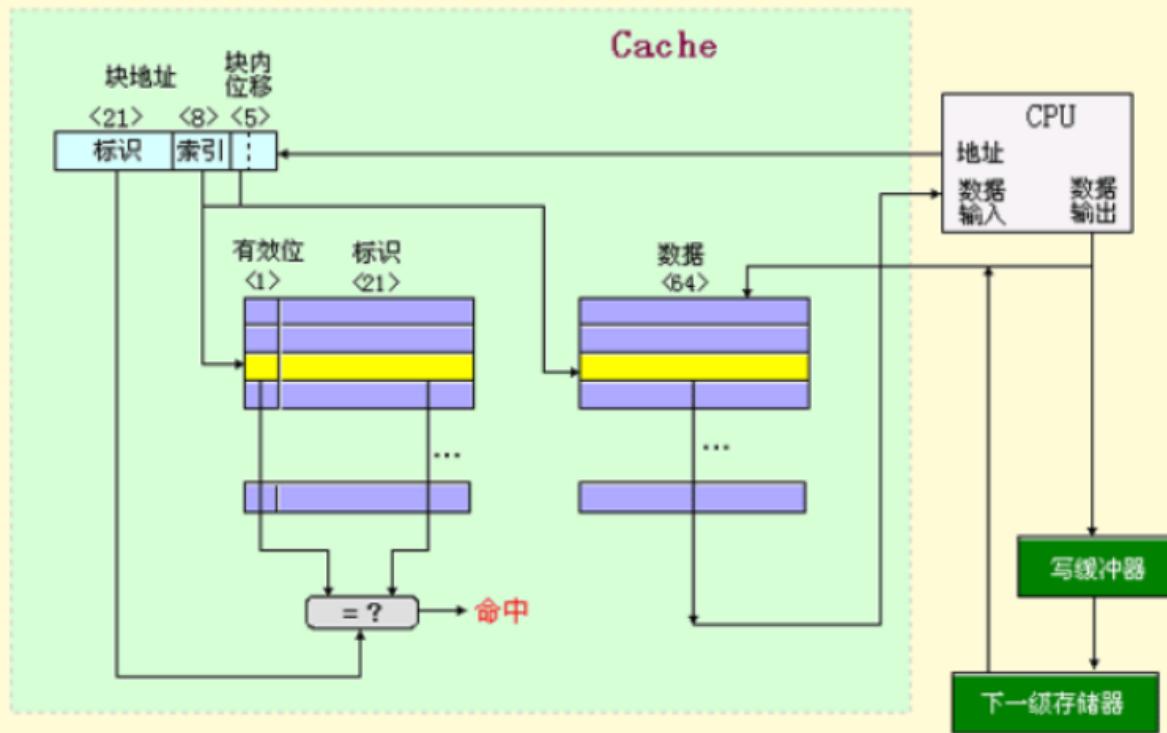
- 映像规则
- 查找算法
- 替换算法
- 写策略

5.2 Cache 基本知识

5.2.5 Cache 结构

- 例子
 - 容量: 8KB
 - 块大小: 32B
 - 块数: 256
 - 直接映射
 - 写直达-不按写分配
 - 写缓冲器大小: 4 个块
 - 内存地址: 34 位 (29 块地址 (21tag+8index)、5 块内地址)
- 结构图

Alpha AXP 21064中数据Cache的结构



- 读命中工作过程

(1) 处理器传送给Cache物理地址

(2) 由索引选择标识的过程

- 根据索引从目录项中读出相应的标识和有效位

(3) 从Cache中读出标识之后，用来同从CPU发来的块地址
中标志域部分进行比较

- 为了保证包含有效的信息，必须要设置有效位

(4) 如果有一个标识匹配，且标志位有效，则此次命中
• 通知CPU取走数据

- 写命中工作过程

- 前三步一样，只有在确认标识匹配后才把数据写
- 设置了一个**写缓冲器**
(提高“写”访问的速度)
 - 按字寻址的，它含有4个块，每块大小为4个字。
 - 当要进行写入操作时，如果写缓冲器不满，那么就把数据和完整的地址写入缓冲器。**对CPU而言，本次“写”访问已完成，**CPU可以继续往下执行。**由写缓冲器负责把该数据写入主存。**
 - 在写入缓冲器时，要进行写合并检查。即检查本次写入数据的地址是否与缓冲器内某个有效块的地址匹配。如果匹配，就把新数据与该块合并。
- 读失效
 - 读失效：向CPU发出一个暂停信号，通知它等待，并从下一级存储器中新调入一个数据块(32字节)
 - Cache与下一级存储的数据通路宽度为16B，传送一次需5个周期，因此，一次传送需要10个周期
- 写失效
 - 写失效：将使数据“绕过”Cache，直接写入主存。
 - 写直达-不按写分配
 - 因为是写直达，所以替换时不需要写回

分离 cache 和混合 cache：指令和数据是保存在两个 cache 中还是单独各有一个 cache
分离 cache 失效率：各自失效率乘上访问概率再相加

5.2.6 cache 性能分析

- 平均访存时间：命中时间+不命中率*不命中开销
- CPU 时间
 - (CPU 执行周期数+存储器停顿周期数) *时钟周期时间
 - $IC \times (CPI_{exe} + \text{每条指令平均存储器停顿周期数}) \times \text{时钟周期时间}$
 - 存储器停顿周期数=访存次数*失效率*失效开销

5.2.7 改进 cache 性能

- 降低失效率
- 减少失效开销
- 减少命中时间

调节Cache块的大小

提高相联度

Victim Cache

硬件预取

编译器控制的预取

编译器优化

降低失效率

写缓冲及写合并

让读失效优先于写

请求字处理

多级Cache

非阻塞Cache

减少失效开销

容量小、结构简单Cache

虚拟化Cache 多体Cache

路预测

Trace Cache

访问流水化

减少命中时间

5.3 降低 cache 失效率的办法

三种失效

- 强制性失效：第一次访问

- 容量失效：因为容量不足被替换后重新访问。随着容量增加而减少，不受相联度影响
- 冲突失效：被冲突后重新访问。相联度越高，冲突失效越少

2:1 的 cache 经验规则：大小为 N 的直接映像 cache 失效率约等于大小为 $N/2$ 的两路组相联 cache 失效率

减少三种失效的方法：

- 强制性失效：增加块大小，预取
- 容量失效：增加容量
- 冲突失效：提高相联度

许多降低失效率的方法都会增加失效开销或命中时间

5.3.1 调节 cache 块大小

失效率与块大小关系

- 对于给定 cache 容量，当块大小增加失效率开始下降，后来反而上升
- cache 容量越大，失效率达到最低的块大小就越大

增加块大小会增加失效开销

5.3.2 提高相联度

相联度超过 8 之后意义不大

2:1 的 cache 经验规则：大小为 N 的直接映像 cache 失效率约等于大小为 $N/2$ 的两路组相联 cache 失效率

提高相联度会增加命中时间

Cache 容量 (KB)	相联度 (路)			
	1	2	4	8
4	3.44	3.25	3.22	3.28
8	2.69	2.58	2.55	2.62
16	2.23	2.40	2.46	2.53
32	2.06	2.30	2.37	2.45
64	1.92	2.14	2.18	2.25
128	1.52	1.84	1.92	2.00
256	1.32	1.66	1.74	1.82
512	1.20	1.55	1.59	1.66

5.3.3 Vivtim cache

基本思想：在 Cache 和它从下一级存储器调数据的通路之间设置一个全相联的小 Cache，用于存放被替换出去的块(称为 Victim)，以备重用。失效时先检查 Vivtim 中是否有要请求的块，如果有直接调入 cache，没有再访问下一级存储器

作用：减小冲突失效

伪相联 cache：采用和 vivtim cache 相近的思想

- 在逻辑上把直接映象 Cache 的空间上下平分为两个区。对于任何一次访问，伪相联 Cache 先按直接映象 Cache 的方式去处理。若命中（快命中），则其访问过程与直接映象 Cache 的情况一样。若不命中，则再到另一区相应的位置去查找。若找到，则发生了伪命中（慢命中），否则就只好访问下一级存储器。
- 命中时间小，失效率低
- 多种命中时间会使 CPU 流水线设计复杂

5.3.4 硬件预取

在 CPU 提出对指令或数据的请求之前进行预取，放入 Cache 或者外缓冲器中。硬件预取通常由 cache 之外的硬件完成

比如：指令失效时，取被请求的指令块放入 cache 和顺序的下一指令块放入缓冲器中，如果某次被请求的指令快在缓冲器中，则取消对该块的访存请求，直接从缓冲器中读取，同时发出对下一指令块的预取请求。

5.3.5 编译器控制的预取

由编译器加入预取指令，在数据被用到之前发出预取请求

预取类型

- 预取数据存放位置
 - 寄存器预取
 - cache 预取
- 预取的故障处理方式
 - 故障性预取：出现虚地址故障或访问越界，抛出异常
 - 非故障性预取：出现虚地址故障或访问越界，不抛出异常，只是转为不预取

只有在预取数据同时，CPU 还能正常执行才有意义：Cache 在等待数据返回时，仍能接受请求。非阻塞 cache

循环是预取优化的主要对象

- 失效开销小：展开 1~2 次
- 失效开销大：展开许多次

每次预取还会产生一条指令的开销

- 保证这种开销不会超过预取带来的收益
- 编译器可以把重点放在可能会导致不命中的访问上，减少不必要的预取

5.3.6 编译器优化

基本思想：在编译时，对程序中的指令和数据进行重新组织，降低 cache 的失效 rate（指令失效和数据失效）

- 重新组织程序而不影响正确性：把其中的几个重要过程重新排序，可能减少冲突失效，降低指令的不命中率

- 如果编译器知道一个分支指令很可能会成功转移，那么它就可以通过以下两步来改善空间局部性
 - 将转移目标处的基本块和紧跟着该分支指令后的基本块进行对调
 - 把该分支指令换为操作语义相反的分支指令
- 降低数据失效主要是对数组的优化
 - 数组合并

/* 修改前 */

```
int val [SIZE];
int key [SIZE];
```

/* 修改后 */

```
struct merge {
    int val ;
    int key ;
} ;
struct merge merged_array[size];
```

- 内外循环交换

/* 修改前 */

```
for (j = 0 ;j<100 ;j = j + 1)
    for (i = 0 ;i<5000 ;i = i + 1)
        x[i][j] = 2*x[i][j];
```

/* 修改后 */

```
for (i = 0 ;i<5000 ;i = i + 1)
    for (j = 0 ;j<100 ;j = j + 1)
        x[i][j] = 2*x[i][j];
```

- 循环融合

```
for (i = 0 ; i<N ;i = i + 1)
    for (j = 0 ; j<N ; j = j + 1)
        a[i][j] = 1/b[i][j]*c[i][j];
for (i = 0 ;i<N ;i = i + 1)
    for (j = 0 ; j<N ;j = j + 1)
        d[i][j] = a[i][j] + c[i][j];
```

/* 修改后 */

```
for (i = 0 ; i < N ; i = i + 1)
    for (j = 0 ; j < N ; j = j + 1) {
        a[i][j] = 1/b[i][j]*c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
```

◦ 分块

5.3.7 总结

优化技术	不命中率	不命中开销	命中时间	硬件复杂度	说明
增加块大小	+	-		0	实现容易；Pentium 4 的第二级Cache采用了128字节的块
提高相联度	+		-	1	被广泛采用
牺牲Cache	+			2	AMD Athlon采用了8个项的Victim Cache
伪相联Cache	+			2	MIPS R10000的第二级Cache采用
硬件预取指令和数据	+			2~3	许多机器预取指令，UltraSPARC III预取数据
编译器控制的预取	+			3	需同时采用非阻塞Cache；有几种微处理器提供了对这种预取的支持
用编译技术减少Cache不命中次数	+			0	向软件提出了新要求；有些机器提供了编译器选项

5.4 减少 cache 失效开销

5.4.1 写缓冲及写合并

在写直达 cache 中，写请求都发送到下一级存储层次中，所以经常使用写缓冲来提高写效率

如何提高写缓冲的效率

- 写合并

如果写缓冲器中已经有了待写入的数据，就要把这次的写入地址与写缓冲器中已有的所有地址进行比较，看是否有匹配的项。如果有地址匹配而对应的位置又是空闲的，就把这次要写入的数据与该项合并。这就叫写缓冲合并。

如果写缓冲器满且又没有能进行写合并的项，就必须等待。

5.4.2 让读失效优先于写

读不命中时，所读单元最新值可能在写缓冲器中

解决办法

- 推迟读失效的处理：增加读失效的开销
- 让读失效优先于写
 - 读失效时先检查写缓冲器中内容：增加硬件

5.4.3 请求字处理

尽早把请求字发送给 CPU

- 尽早重启动：调块时，从块的起始位置开始读取，一旦请求字到达立即送 CPU
- 请求字优先：调块时，从请求字所在位置读起，第一个读出请求字后立即送 CPU

在以下情况效果不大

- cache 块比较小
- 下一条指令正好访问同一 cache 块的另一部分

5.4.4 多级 cache

$$\begin{aligned}\text{平均访问时间} &= \text{命中时间}_{L_1} + \text{失效率}_{L_1} \times \text{失效开销}_{L_1} \\ &= \text{命中时间}_{L_1} + \text{失效率}_{L_1} \times \\ &\quad (\text{命中时间}_{L_2} + \text{失效率}_{L_2} \times \text{失效开销}_{L_2})\end{aligned}$$

局部失效率与全局失效率

➤ 局部失效率 = 该级 Cache 的失效次数 / 到达该级 Cache 的访问次数

(例如：上述式子中的失效率_{L₂})

➤ 全局失效率 = n 级 Cache 的失效次数 / CPU 发出的访存的总次数

➤ 全局失效率 = 失效率_{L₁} × 失效率_{L₂}

当第二级 cache 比第一级大得多时，两级 cache 的全局失效率和第二级 cache 相同的单级 cache 的失效率非常接近

第二级 cache 的参数

- 容量：一般比第一级 cache 大得多
- 相联度：可以采用较高的相联度或伪相联
- 块大小：可以采用较大的块
- 多级包容和多级互斥：第一级的数据是否总是同时存在于二级 cache 中

5.4.5 非阻塞 cache

cache 失效时仍允许 CPU 进行其他的命中访问，允许失效下命中

- 进一步提升：多重失效下命中
- 可同时处理的不命中次数越多，所带来的性能提升越大

5.4.6 总结

优化技术	不命中率	不命中开销	命中时间	硬件复杂度	说明
写缓冲写合并		+		1	与写直达合用，广泛应用，例如21164, UltraSPARC III
使读失效优先于写		+	-	1	在单处理机上实现容易，被广泛采用
请求字处理		+		2	被广泛采用
两级Cache		+		2	硬件代价大；两级Cache的块大小不同时实现困难；被广泛采用
非阻塞Cache		+		3	在支持乱序执行的CPU中使用

5.5 减少命中时间

5.5.1 容量小、结构简单的 cache

- 硬件越简单，速度越快
- 应使 cache 足够小，可以和 CPU 放在同一块芯片上
- 一种折中方案：Cache 标识放在片内，数据放在片外

5.5.2 虚拟 cache

物理 cache：地址转换和访问 cache 串行进行，访问速度慢

虚拟 cache：直接使用虚拟地址访问 cache

- 优点：命中时不需要进行地址转换。不命中地址转换和访问 cache 也是并行进行的
- 缺点
 - 不同进程可能使用相同虚地址指向不同的物理地址
 - 进程切换时清空 cache
 - 在地址字段中增加 PID 字段
 - 同一物理地址可能有不同形式的虚拟地址
 - 硬件检查并作废相同物理地址的副本

- 页着色，通过软件强制别名的某些位相同
- 虚拟索引+物理标识

**利用虚拟地址找到索引读cache，同时进行虚实地址转换：
前提是虚拟索引=物理索引**

而：页内位移在虚实地址变换时不变

优点：兼得虚拟Cache和物理Cache的好处

局限性：Cache容量受限 \leq 页大小 \times 相联度

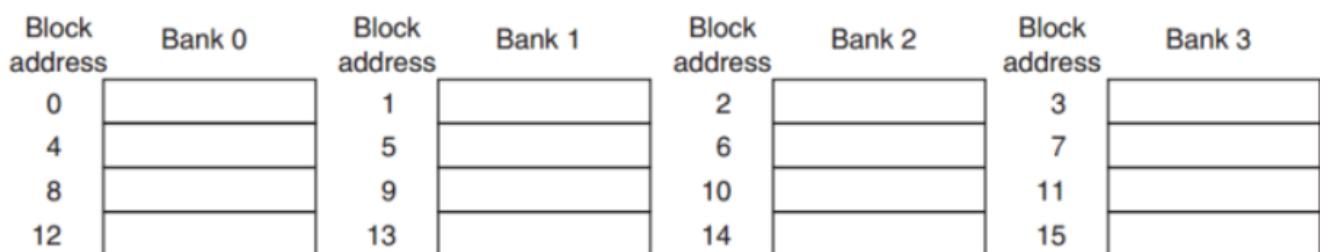
5.5.3 cache 访问流水化

当第一级 cache 的命中时间是多个时钟时，将其访问按流水方式组织

**访问Cache周期数的增加使得流水线的站数也增加了，
进一步增加了分支预测失败的代价，也增加了从load指令
发射到使用该数据间的时间间隔**

5.5.4 多体 cache

将 cache 组织为多个独立的 banks，支持并行访问



作用

- 当多个体利用同一个组索引进行访存时，它可以使组相联结构的 Cache 中的一组数据和 Tag 同时被访问，加快查找时间
- 当多个体可以利用不同地址进行独立访问时，它可以更有效地支持 CPU 发出的多个访存请求，减小 Cache 的平均命中时间，增大 Cache 的吞吐率

5.5.5 路预测

每一组 cache 保存一些预测位，标识下次访问本组时应命中的块。命中时只用比较一个块的 tag，不命中再比较其他的

一旦预测错误会有更长的命中时间

5.5.6 Trace cache

开发指令级并行的挑战：当要每个时钟周期流出超过 4 条指令时，要提供足够多条彼此互不相关的指令是很困难的

trace cache：

- 保存的是 CPU 实际执行的动态指令序列
- 地址映像机制复杂
- 相同的指令序列有可能被当作条件分支的不同选择而重复存放
- 能提高指令 cache 的空间利用率

5.5.7 总结

优化技术	不命中率	不命中开销	命中时间	硬件复杂度	说明
容量小且结构简单的Cache	-		+	0	实现容易，被广泛采用
虚拟Cache			+	2	对于小容量Cache来说实现容易，已被Alpha 21164和UltraSPARC III采用
流水化Cache访问			+	1	被广泛采用
多体Cache			+	1	Operon和Niagara L2 Cache
路预测			+	1	Pentium 4 采用
Trace Cache			+	3	Pentium 4 采用

5.6 主存

性能指标：

- 延迟
- 带宽

假设基本存储器结构的性能为：

- 送地址需4个时钟周期
- 每个字的访问时间为24个时钟周期
- 传送一个字（4个字节）的数据需4个时钟周期

如果Cache大小为4个字，则：

$$\text{失效开销} = 4 \times (4 + 24 + 4)$$

$$= 4 \times 32$$

$$= 128 \text{ (时钟周期)}$$

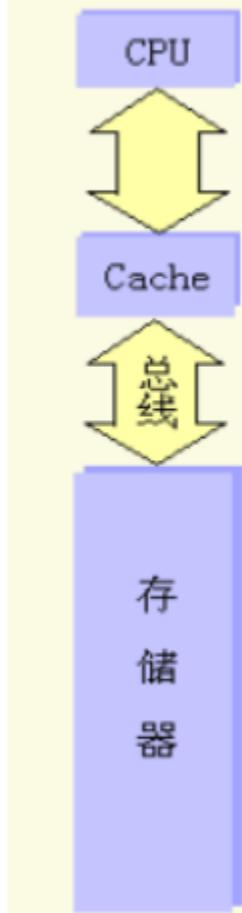
$$\text{带宽} = 16 / 128$$

$$= 0.0125 \text{ (字节/时钟周期)}$$

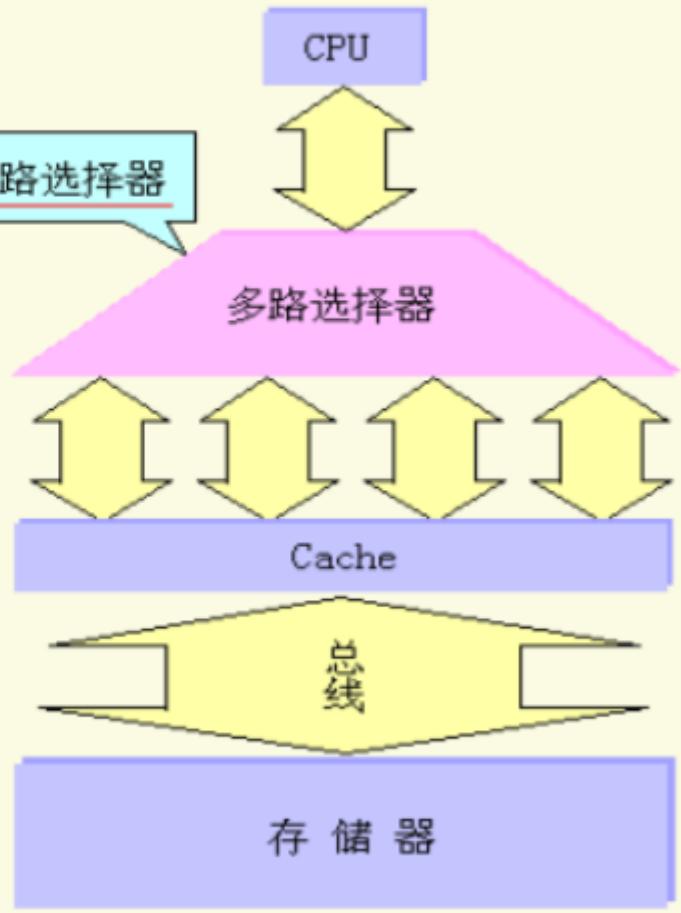
5.6.1 存储器组织技术

增加存储器宽度：一次送多个字

单字宽存储器



多字宽存储器



并增设一个
多路选择器

➤ 性能分析（参照前面的假设）

- 当宽度为4个字时：失效开销=1×32(周期)
- 带宽=16/32=0.5(字节/周期)

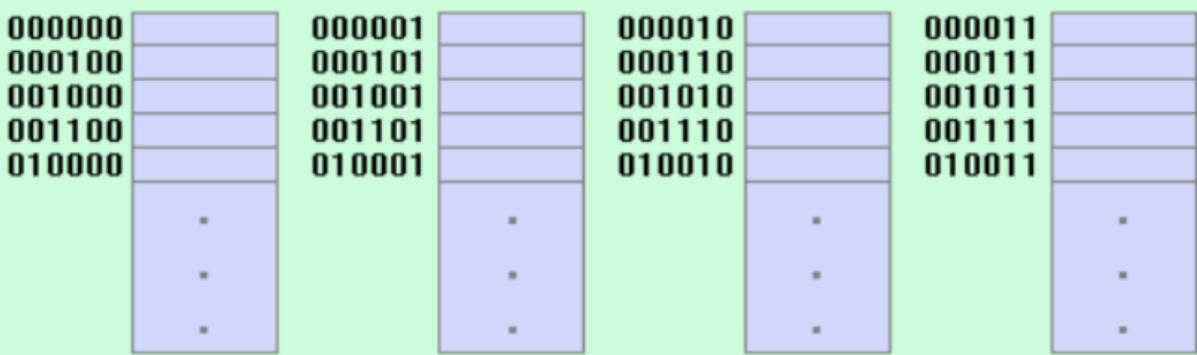
➤ 缺点：

- 增加CPU和存储器之间的连接通路宽度
- CPU和Cache之间有一个多路选择器
- 扩充主存的最小增量增加了相应的倍数
- 写入有可能变得复杂

采用简单的多体交叉存储器

- 高位交叉和低位交叉

- 低位交叉编址

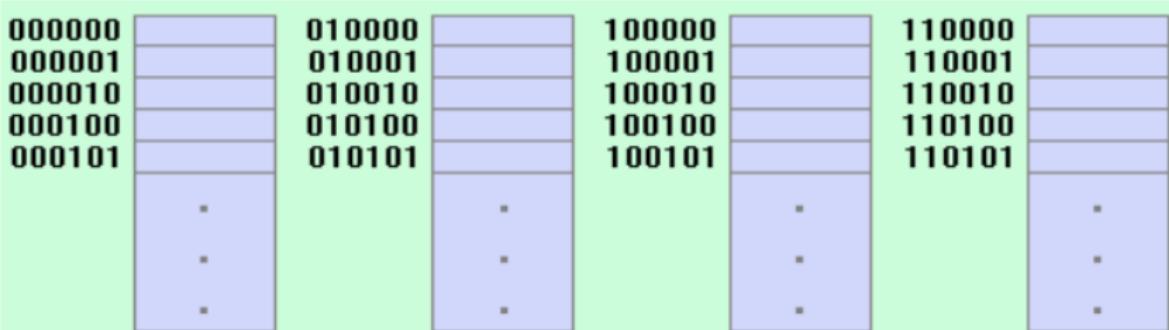


给定一定单元，假设其地址的二进制形式为：

一般地，若有 2^m 个体，则：

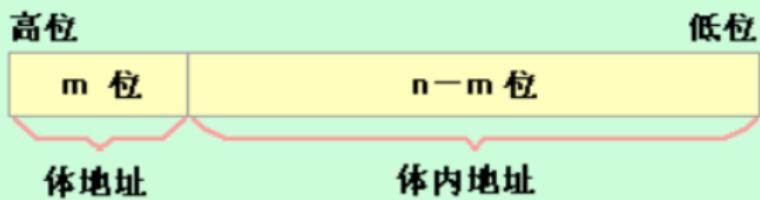


- 高位交叉编址



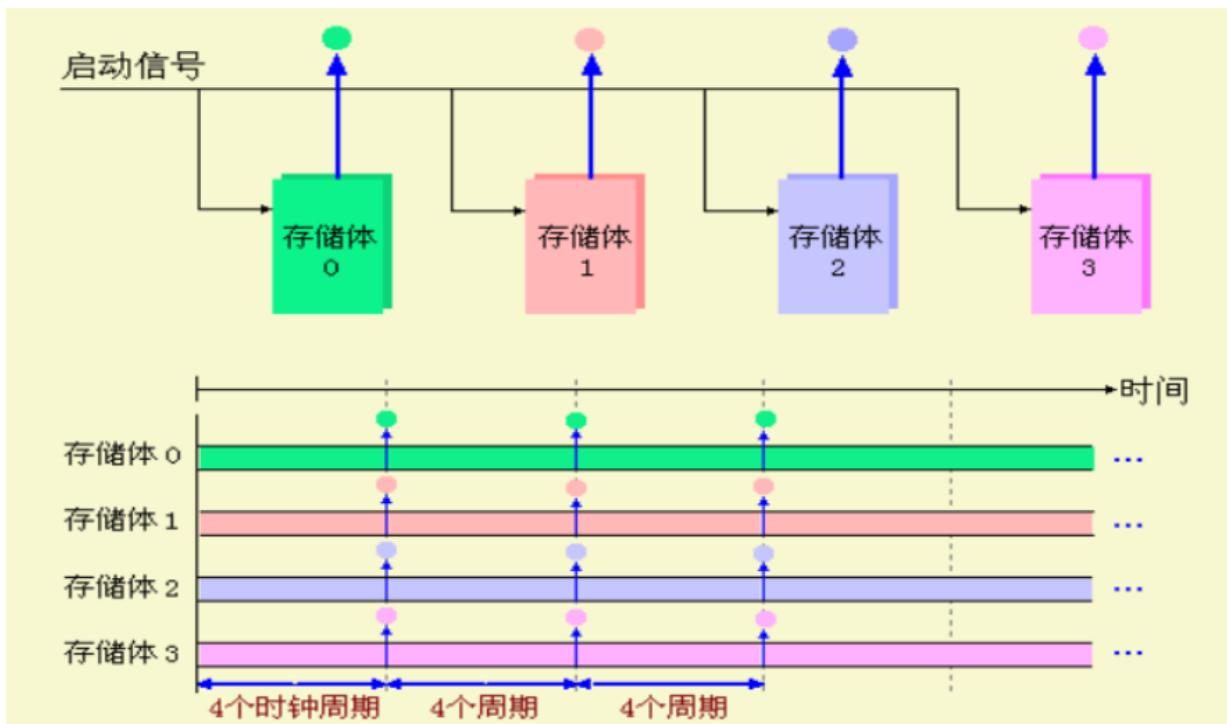
给定一定单元，假设其地址的二进制形式为：

一般地，若有 2^m 个体，则

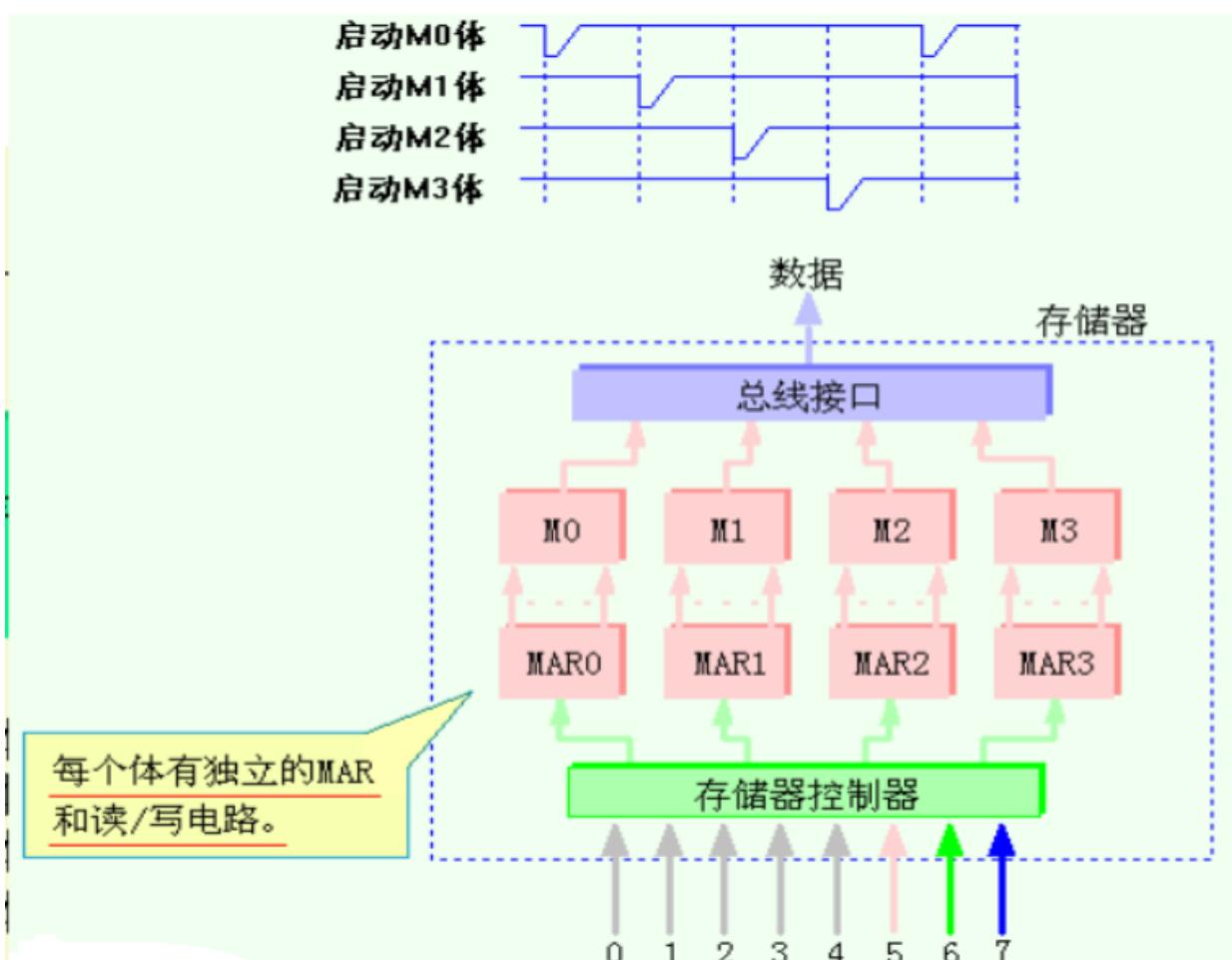


- 同时启动和分时启动

- 同时启动



- 分时启动



- 失效开销：地址传送和数据访问都是并行进行的，但是数据的传送是串行的，所以一共有 44 个周期

独立存储体：设置多个存储控制器，使多个体能独立操作，以便同时进行多个独立的访问

5.6.2 存储器芯片技术

DRAM 和 SRAM

容量: 4~8: 1

存储周期: 8~16:1

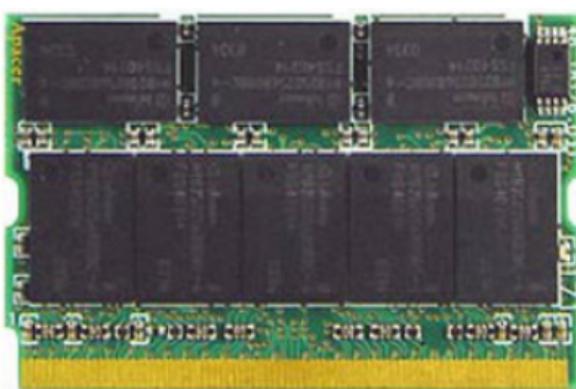
价格: 1: 8~16

一般来说: 主存: DRAM Cache: SRAM

DIMM

多个DRAM芯片经常被组装在称为条的小型板上，构成“双列直插式存储模块”。

一个DIMM通常包含4~16片DRAM芯片，这些芯片常被组织成8字节宽的主存（带ECC校验）



DRAM 芯片优化技术

- 芯片内部优化技术是提高主存系统性能的一个重要方面
- SDRAM: , DRAM 接口增加一个时钟信号可使 DRAM 能针对一个请求连续同步地传输多个数据而不需同步开销
- DDR: 在 DRAM 时钟的上沿和下沿都进行数据传输, 可把数据传输率提高一倍
- CDRAM: 带 Cache 的 DRAM, DRAM 芯片里集成一个小的 SRAM, 暂存最后读出行数据

5.7 虚拟存储器

5.7.3 虚存和 cache 关系的例子

CPU 使用虚地址发出请求，TLB 通过虚地址中的虚页号查出 TLB 数据，L1cache 使用虚拟 cache 通过虚地址页内偏移查出 tag，比较 tag 和 TLB 数据是否相同，相同则将 L1cache 中读出的数据送入 CPU，如果不同，则利用物理地址查找 L2cache。

5.8 进程保护和虚存实例

进程保护

1. 界地址寄存器

基地址，上界地址

检测条件：(基地址 + 地址) ≤ 上界地址

2. 虚拟存储器

给每个页面增加访问权限标识

3. 环形保护

4. 加锁和解锁

第六章 输入输出系统

6.1 引言

输入输出系统 (I/O 系统)

- 包括

- I/O 设备
- I/O 设备与处理机的连接
- 分类
 - 存储 I/O 系统
 - 通信 I/O 系统
- 性能衡量
 - 系统响应时间：从用户输入命令开始，到得到结果花费的时间
 - I/O 相应时间
 - CPU 处理时间
- 使用多进程技术不能忽略 I/O 性能对系统性能的影响
 - 多进程只能提高系统吞吐率，不能减少相应时间
 - 进程切换需要增加 I/O 操作
- 性能评价参数
 - 连接特性
 - I/O 系统的容量
 - 相应时间和吞吐率等

6.2 外部存储设备（略）

6.3 I/O 系统分析与评测

6.3.1 I/O 性能与系统相应时间

性能评价参数

- 连接特性
- I/O 系统的容量
- 响应时间和吞吐率等

响应时间和吞吐率是一对矛盾

6.3.2 Little 定律

➤ 黑箱 (Black Box)

- 假定I/O系统具有几个独立的请求源，I/O系统具有足够的服务能力使得系统达到平衡的状态：系统的输入速率与输出速率相等，I/O请求的服务速率与请求等待的时间无关。

➤ Little定律

- 系统中的平均任务数 = 到达率 × 平均响应时间

6.3.3 M/M/1 排队系统

➤ M/M/1排队系统一般假设为：

- 系统为一个平衡系统
- 任务的到达率 λ ，连续两个到达请求的时间间隔服从指数分布，其均值为平均到达时间。
- 请求的个数不受限制
- 如果排队中有任务，服务员服务完当前任务后立即服务下一个。
- 任务的服务速率 μ
- 队列无限长，FIFO规则
- 系统只有一个服务员
- 服务员利用率（服务强度） $\rho=(\lambda/\mu)$

□ 相关结论

- 系统中没有任务的概率 $P_0 = 1 - \rho$
- 系统中有 n 个任务的概率 $P_n = (1 - \rho)\rho^n, n = 0, 1, \dots$
- 系统中平均任务数 $E(n) = \rho / (1 - \rho)$
- 队列中平均任务数 $E(n_q) = \rho^2 / (1 - \rho)$
- 系统平均响应时间 $E(R) = (1/u) / (1 - \rho)$
- 任务在队列中的平均等待时间
 - $E(W) = \rho \frac{1-\mu}{1-\rho}$

6.3.4 M/M/m 排队系统

将 M/M/1 排队系统服务员增加到 m 个

6.3.5 I/O 基准测试程序

- 使用 I/O 基准测试程序来反映响应时间和吞吐率之间的平衡关系
- TPC (Transaction Processing Performance Council)
 - 事务处理委员会
 - 发布 9 个事务处理基准测试程序
 - 高端商业应用中，通常采用 TPC-C 测试程序进行测试

- TPC具有一些独特的性质
 - 测试结果中**给出系统的价格因素**
 - TPC**模拟的是实际系统**
 - 测试结果经过TPC审核后才能发布
 - 吞吐率指标受到响应时间的限制
 - 通过独立的机构来维护

6.4 I/O 系统的可靠性、可用性和可信性

故障、错误、失效

- 故障引起错误，错误引起失效
- 故障分类
 - 瞬时性
 - 间歇性
 - 永久性
- 故障产生原因
 - 硬件
 - 设计
 - 操作
 - 环境

反映外设可靠性能的参数

- 可靠性：系统从某个初始参考点开始一直连续提供服务的能力
 - 用平均无故障时间 MTTF 衡量
 - MTTF 的倒数就是系统的失效率
 - 系统中断服务的时间用平均维修时间 MTTR 衡量
- 可用性：系统正常工作的时间在连续两次正常服务间隔时间中所占的比率

- $\frac{MTTF}{MTTF+MTTR}$
- MTTF+MTTR: 平均失效间隔时间 MTBF

- 可信性: 服务的指令, 在多大程度上可以合理地认为服务是可靠的 (不可度量)

提高系统组成部件可靠性的方法:

- 有效构建方法: 在构建系统的过程中将故障隐患消除
- 纠错方法: 构建系统时采用容错方法

提高系统可靠性的方法:

- 故障避免技术: 通过合理构建系统来避免故障
- 故障容忍技术: 采取冗余措施
- 错误消除技术: 通过验证, 最大限度地减少潜在的错误
- 错误预报技术: 通过分析, 预报错误的出现, 以便提前采取应对措施

6.5 廉价磁盘冗余阵列

RAID级别	可以容忍的故障个数以及当数据盘为8个时, 所需要的检测盘的个数	优点	缺点	应用场合
0 非冗余, 条带存放	0个故障; 0个检测盘	没有空间开销	没有纠错能力	视频处理等高带宽场合
1 镜像	1个故障; 8个检测盘	不需要计算奇偶校验, 数据恢复快, 读数据快。而且其小规模写操作比更高级别的RAID快	检测空间开销最大 (即需要的检测盘最多)	金融等需要高可用性领域
2 存储器式ECC	1个故障; 4个检测盘	不依靠故障盘进行自诊断	检测空间开销的级别是 $\log_2 m$ 级 (m 为数据盘的个数)	无
3 位交叉奇偶校验	1个故障; 1个检测盘	检测空间开销小 (即需要的检测盘少), 大规模读写操作的带宽高	对小规模、随机的读写操作没有提供特别的支持	视频编辑等高吞吐率场合

RAID级别	可以容忍的故障个数以及当数据盘为8个时，所需要的检测盘的个数	优 点	缺 点	公司产品
4 块交叉奇偶校验	1个故障； 1个检测盘	检测空间开销小，小规模的读操作带宽更高	阵列控制器设计复杂，写数据传输率差	文件服务器
5 块交叉分布 奇偶校验	1个故障； 1个检测盘	检测空间开销小，小规模的读写操作带宽更高	阵列控制器设计复杂	应用广泛
6 双维奇偶校验	2个故障； 2个检测盘	具有容忍2个故障的能力	阵列控制器设计复杂；冗余信息设计复杂	文件服务器

实现盘阵列的方式

- 软件方式：成本低，过多占用主机时间、带宽指标上不去
- 阵列卡方式
- 子系统方式

磁盘阵列技术研究的主要热点问题：

- 新型阵列体系结构
- RAID结构与其所记录文件特性的关系
- 在RAID冗余设计中，综合平衡性能、可靠性和开销的问题
- 超大型磁盘阵列在物理上如何构造和连接
- 如何利用新型存储部件构建RAID

6.6 总线（略）

6.7 通道

- 通道处理机能够负担外围设备的大部分I/O工作。
- 通道处理机（简称通道）：专门负责整个计算机系统的输入/输出工作。通道处理机只能执行有限的一组输入/输出指令。

功能：

- 接受 CPU 发来的 I/O 指令，根据指令要求选择一台指定的外围设备与通道相连接
- 执行 CPU 为通道组织的通道程序，从主存中取出通道指令，对通道指令进行译码，并根据需要向被选中的设备控制器发出各种操作命令
- 为主存和外设设置传输控制信息，包括
 - 给出外设的有关地址，即进行读/写操作的数据所在的位置。如磁盘存储器的柱面号、磁头号、扇区号等
 - 给出主存缓冲区的首地址，这个缓冲区用来暂时存放从外围设备上输入的数据，或者暂时存放将要输出到外围设备中去的数据
 - 控制外围设备与主存缓冲区之间数据交换的个数，对交换的数据个数进行计数，并判断数据传送工作是否结束
- 指定传送工作结束时要进行的操作。例如，将外围设备的中断请求及通道的中断请求送往 CPU 等
- 检查外围设备的工作状态，是正常或故障。根据需要将设备的状态信息送往主存指定单元保存
- 在数据传输过程中完成必要的格式变换，例如，把字拆卸为字节，或者把字节装配成字等

主要硬件

➤ 寄存器

- 数据缓冲寄存器
- 主存地址计数器
- 传输字节数计数器
- 通道命令字寄存器
- 通道状态字寄存器

➤ 控制逻辑

- 分时控制
- 地址分配
- 数据传送、装配和拆分等

通道对外设的控制通过输入/输出接口和设备控制器进行:

- 通道与设备控制器之间一般采用标准的输入/输出接口来连接
- 通道通过标准接口把操作命令送到设备控制器，设备控制器解释并执行这些通道命令，完成命令指定的操作
- 设备控制器能够记录外设的状态，并把状态信息送往通道和 CPU

通道的工作过程

- 在用户程序中使用访管指令进入管理程序，由管理程序生成一个通道程序，并启动通道
 - 用户在目标程序中设置一条广义指令，通过调用操作系统的管理程序来实现
 - 管理程序根据广义指令提供的参数来编制通道程序
 - 启动输入/输出设备指令是一条主要的输入/输出指令，属于特权指令
- 通道处理机执行通道程序，完成指定的数据输入/输出工作
 - 通道处理机执行通道程序与 CPU 执行用户程序是并行的
- 通道程序结束后向 CPU 发中断请求

通道的种类

- **字节多路通道 (Byte multiplexer channel)** : 简单的共享通道，为多台低速或中速的外围设备服务。采用分时方式工作。
- **选择通道 (Selector channel)** : 为高速外围设备（如磁盘存储器等）服务。在传送数据期间，只能为一台高速外围设备服务，在不同的时间内可以选择不同的设备。
- **数组多路通道 (Block multiplexer channel)** : 为高速设备服务，各台高速设备重迭操作。

6.8 I/O与操作系统

设计I/O系统需要注意操作系统的因素:

- 在用硬件实现的I/O技术中，哪些会实际被采用，是由操作系统来决定的。
- I/O操作主要是在外设和存储器之间进行，所以操作系统必须保证这些I/O操作的安全性。

6.8.1 DMA和虚拟存储器

DMA是使用虚拟地址还是物理地址？

1. 使用物理地址进行DMA传输，存在以下两个问题：

- 对于超过一页的数据缓冲区，由于缓冲区使用的页面在物理存储器中不一定是连续的，所以传输可能会发生问题。
- 如果DMA正在存储器和缓冲区之间传输数据时，操作系统从存储器中移出（或重定位）一些页面，那么，DMA将会在存储器中错误的物理页面上进行数据传输。

2. 解决这些问题的方法

- 使操作系统在I/O的传输过程中确保DMA设备所访问的页面都位于物理存储器中，这些页面被称为是**钉在了主存中**。
- “**虚拟DMA**”技术
 - 允许DMA设备直接使用虚拟地址，并在DMA期间由硬件将虚拟地址转换为物理地址。
 - 在采用**虚拟DMA**的情况下，如果进程在内存中被移动，操作系统应该能够及时地修改相应的**DMA地址表**。

6.8.2 I/O和Cache的数据一致性

1. Cache会使一个数据出现两个副本：
一个在Cache中，另一个在主存中。
2. 数据不一致问题有两个方面：
 - 存储器中可能不是CPU产生的最新数据，所以I/O系统从存储器中取出来使用的是陈旧数据；
 - I/O与存储器交换数据之后，在Cache中被CPU使用的可能就是陈旧数据。

3. I/O与计算机的连接有两种方式：

① 把I/O直接连接到Cache上

- 不会产生由I/O导致的数据不一致的问题。

所有I/O设备和CPU都能在Cache中看到最新的数据

- I/O会跟CPU竞争访问Cache，在进行I/O时，会造成CPU的停顿。
- I/O还可能会破坏Cache中CPU访问的内容，因为I/O操作可能导致一些新数据被加入Cache，而这些新数据可能在近期内并不会被CPU访问。

② 把I/O连接到存储器上

- CPU修改了Cache的内容后，由于存储器的内容跟不上Cache内容的变化，I/O系统进行输出操作时所看到的数据是旧值（写直达Cache没有这样的问题）。
- I/O系统进行输入操作后，存储器的内容发生了变化，但CPU在Cache中所看到的内容依然是旧值。

4. 解决内容一致性问题的方法

➤ 确保I/O从存储器中取出的是正确的数据

- 写直达Cache可以保证存储器和Cache中有相同的数据
- 写回Cache则需要操作系统帮助进行数据块检查：
 - ✓ 操作系统将I/O使用的存储器地址与Cache中的tag进行比较
 - ✓ 如果数据块未被修改， Cache和存储器数据一致
 - ✓ 如果数据块已被修改， Cache中是最新的数据，需要将Cache中的数据块写回

➤ I/O与存储器交换数据之后，确保Cache中的数据及时更新

- 操作系统保证I/O操作的数据块不在Cache中
- 如果发现I/O地址在Cache中有匹配的项，就把相应的Cache块设置为“无效”