并行时代的处理器

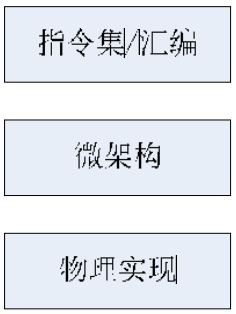
kiterunner_t
TO THE HAPPY FEW.

在传统顺序化程序中，从程序源代码到执行代码完成某个任务，编译器、加载器和操作系统不断的提供了若干抽象，使得程序员可以指挥硬件完成日益复杂的任务。而在连串复杂的转化过程中，若程序员不了解底层的硬件习性，各个抽象处理环节就可能不能充分高效的利用硬件。现在并行时代，并行已经深入到计算机的各个层级中了，如硬件指令级并行、数据并行，再到更高层的线程并行，顺序编程模型中封装良好的底层硬件特性不再对程序员透明，更需要程序员了解各个层面的工作。

对于多核时代的并行编程技术来说，显而易见的正确性不见了，我们为了保证程序能够正确的执行，开发了若干模型来保证底层的硬件特性满足规定的正确性。

从计算机执行代码的最底层来说，处理器是完成这些工作的重心。我需要从不同的视角来了解处理器。处理器可以按照下图的层级来进行划分，指令集是程序和处理器的接口，微架构是处理器的设计，最终会将微架构化为电子电路之类的物理实现（这块对于程序员来说不用太关心）。因此，按照这种划分，我需要完成以下几篇小结：

1) 现代处理器微架构的一些技术。
2) 常见的指令集有 5 种，x86、ARM、MIPS、Power、C6000(DSP)等。我主要在 x86 体系上开发，因此以 x86-64 为主，会涉及到一些 Power 体系。
3) 多核时代并发编程的指令集和微架构一些知识。



本文简单介绍了现代计算机硬件的重要部件 CPU 的习性。首先简单的介绍了现代计算机的三个部件及其相互连接的方式，计算机基本结构。然后重点放在当代处理器微架构的一些技术。从某个角度来说，这些技术都是为了提高并行性，缓解处理器和主存之间速度的鸿沟而设计的。因此我试图从各个角度来理解并行时代程序员需要理解的关于处理器的知识。

1. 计算机的基本结构

计算机的基本结构由 3 部分组成：内存（或主存），处理器，输入输出设备，它们彼此交互的信息包括数据、指令和地址，以及控制信息。计算机从输入输出设备载入数据和指令到主存中，CPU 从主存中获取数据和指令并处理，将结果写回主存，最后再把结果写到输入输出设备。（冯 • 诺依曼结构）

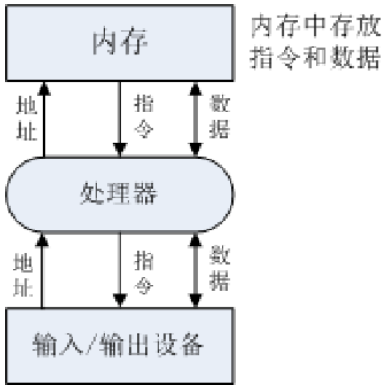


图 1-1 计算机的基本结构

这三个部分在传统上使用**通用总线**连接起来的，所有内存、IO 和 CPU 的数据、指令、地址、控制信息的交互都是通过这一组总线仲裁电路互斥完成操作的。通用总线结构简单，但是一次只能使用一个设备，速度最高只能达到 1GHz 左右。

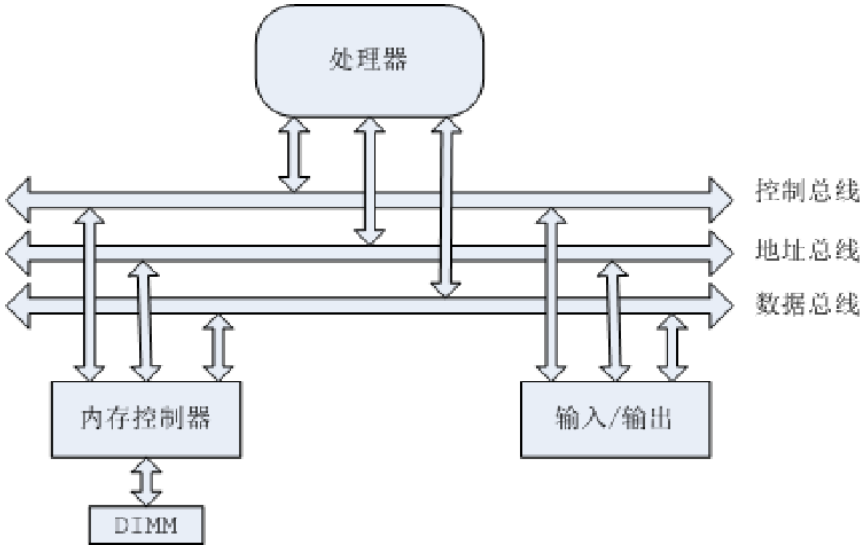


图 1-2 CPU、内存和 IO 的互联

现代处理器对通用总线结构做出了改变，对速度要求高的主存和高速 IO 直接连接到 CPU，CPU 内置内存控制器，低速 IO 通过拥有南桥功能的 IO 控制器芯片连接。

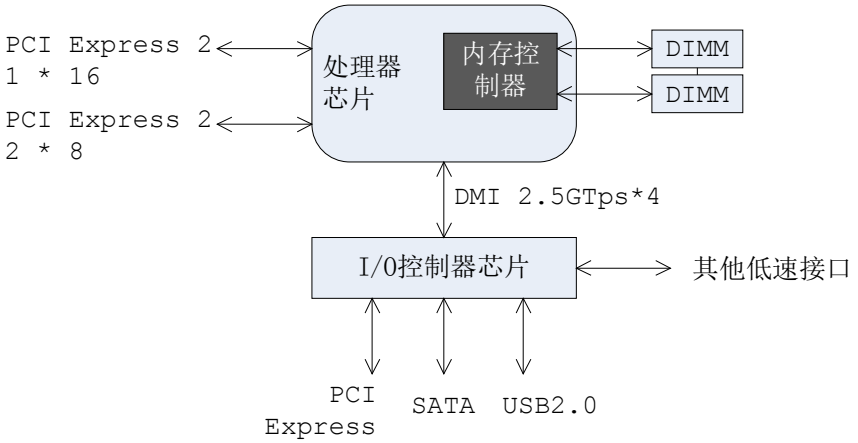


图 1-3 Core i7 的内存与 IO 连接

随着处理器技术的高速发展，主存与处理器的距离越来越大，处理器访问主存的代价越来越大，因此现代计算机都广泛使用了缓存来弥补二者之间的差距。

处理器与 IO 设备之间的交互有以下几种可能的方式：内存映射，IO 专用地址空间 (intel x86)，DMA，命令链（IO 通道，用于高速 IO）。

并行已经深入到计算机体系结构了，处理器提供了指令级并行、数据级并行、线程级并行。

在正式介绍之前，先了解一下常用的一些度量单位，如图 1-4 所示：

Unit	Abbreviation	Fraction of 1s
Minute	m	60
Second	s	1
Millisecond	ms	10^(-3)
Microsecond	us	10^(-6)
Nanosecond	ns	10^(-9)
Picosecond	ps	10^(-12)

图 1-4 时间单位

2. 线程并行（多处理器）

并行的概念已经深入到计算机体系结构的各个方面了，在 20 世纪 60 年代，Flynn 对所有计算机给出了一个简单的分类模型，他根据处理器中调用的指令和数据的并行度把计算机归为 4 类：

1) SISD：单指令流，单数据流，是单处理器。
2) SIMD：单指令流，多数据流，同一条指令被多个使用不同数据流的处理器执行。如多媒体扩展就属于该类型。
3) MISD：多指令流，单数据流，目前还没有这种类型的机器。
4) MIMD：多指令流，多数据流，每个处理器取自己的指令和数据分别进行操作。这是通常多处理器的模型。

在单处理系统上，操作系统在软件层面通过时分复用的方法进行并发，但并发会有较耗时的上下文切换。而现在的计算机都支持硬件级的多线程。处理器只要能够有寄存器保存多个线程的状态，即可直接支持硬件多线程，这样线程上下文切换耗几乎不需要什么时间。硬件级多线程有两种：粗粒度和细粒度多线程。粗粒度多线程指当处理器发现一个线程被长时间中断时，如缓存未命中，则处理器就切换到另一个线程。细粒度多线程指每个时钟周期处理器轮流发射不同线程的指令。细粒度的硬件多线程优势在于多线程之间的指令是不相关的，可以乱序执行。

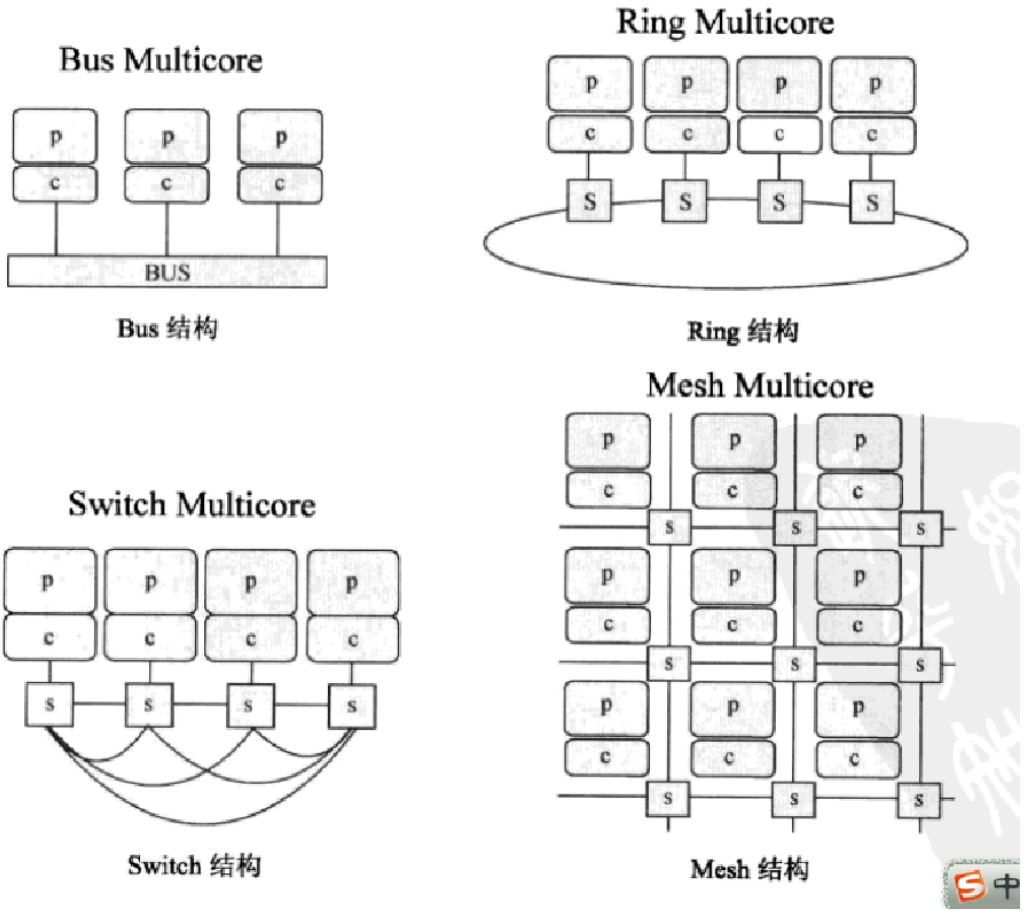
在超标量处理器中，处理器一次能发射多条指令，如果这些指令来自不同线程，那么就被成为同时多线程 SMT。在 Intel 中，该技术又被称为超线程。

线程级并行真正的威力来自于多核处理器。多核处理器常用的互联结构有：

1) 总线结构，是较简单的核间通信方式，内核直接挂在通信总线上，实现简单，缺点

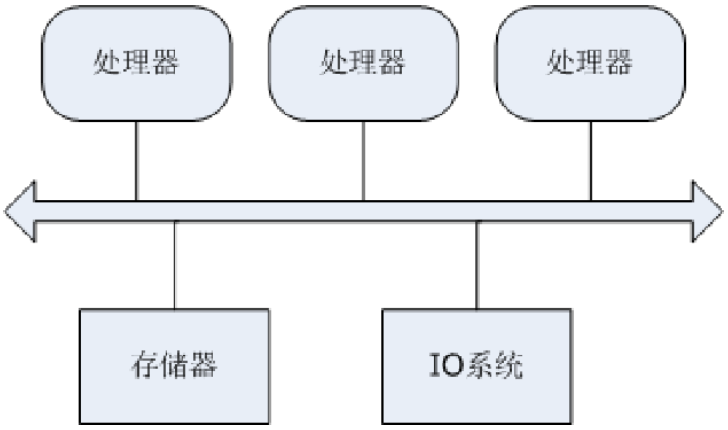
是每个内核通信需要占用总线，效率低，不利于多核扩展。

2) 交换结构，内核直接直接互连，通信效率最好，但是硬件设计复杂，常用于核较少的处理器。
3) 环结构，介于总线和交换结构之间，通信效率较高，硬件不复杂，适用于较少的核的处理器。
4) Mesh 结构，类似于二维的环结构，利于扩展，众核多采用该结构。

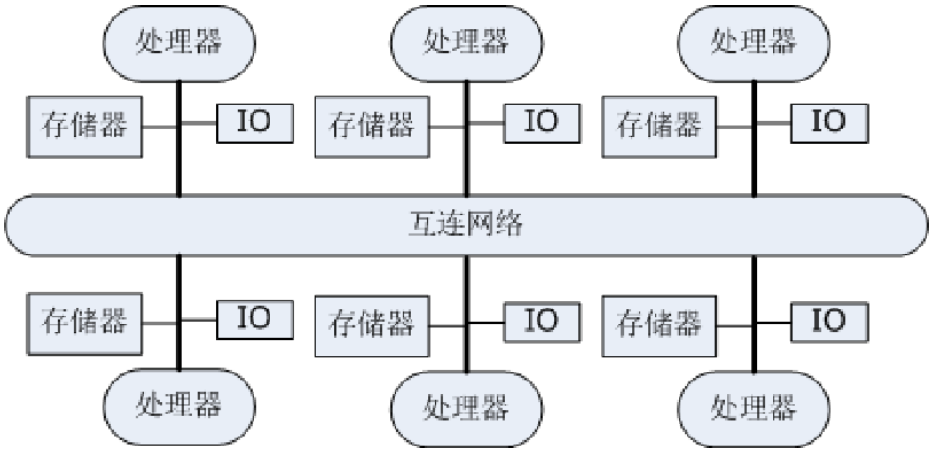


多处理器根据存储器的组织方式和互连策略（不是处理器的组织方式）分为两大类：**集中式共享存储器系统**和**分布式存储器系统**。

集中式共享存储器系统中，所有处理器共享单个集中式存储器，每个处理器访问主存的延时相同。通常使用多个点对点、交换机等技术进行互连。该方案对于大规模的处理器来说并不适用，通常用于小规模的单个主机系统。由于对主存来说，每个处理器是对等的，故更常见的叫法是**对称多处理器系统** SMP，也叫**均匀存储器访问系统** UMA。



分布式存储器系统中，存储器在物理上是分布的，每个处理器构成一个节点，节点间通过直接互连网络（如交换机）或间接互连网络（如多维网状结构）进行互连。该方案利于扩展，但处理器访问存储的延时不同。该方案的处理器之间的通信结构更复杂，常用于复杂系统的构建。根据处理器之间数据传递的方法，又分为两大类：**分布式共享存储器系统** DSM 和**消息传递多处理器系统**。DSM 系统通过共享的逻辑地址空间进行通信，任何一个处理器都能通过直接逻辑地址访问任意节点上的存储器，也就是说这里的存储不是一个单一的集中式存储，与 UMA 不同，故通常又被成为**非均匀存储器访问** NUMA。消息传递多处理器系统中每个处理器拥有自己私有的地址空间，不能直接被其他节点的处理器访问，通信需要通过某种数据消息进行，常见的**集群**就属于这种类型。



SMP
NUMA
集群

3. 高速缓存

处理器与内存的速度差异较大，看下面这个对比（图 3-1 访问延迟）就知道对于处理器来说，内存访问是多么慢了（主频为 3.3GHz，图片来自”Systems Performance: Enterprise and the Clouds”），因此引入了不同级别的高速缓存。

Event	Lantency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s

Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50-150 us	2-6 days
Rotational disck I/O	1-10 ms	1-12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdon	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1-3 s	105-317 years
OS Virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

图 3-1 访问延迟

缓存是收益和成本的权衡，速度与单位容量的成本成正比的。如（图 3-2 速度与单位容量成本）所示。

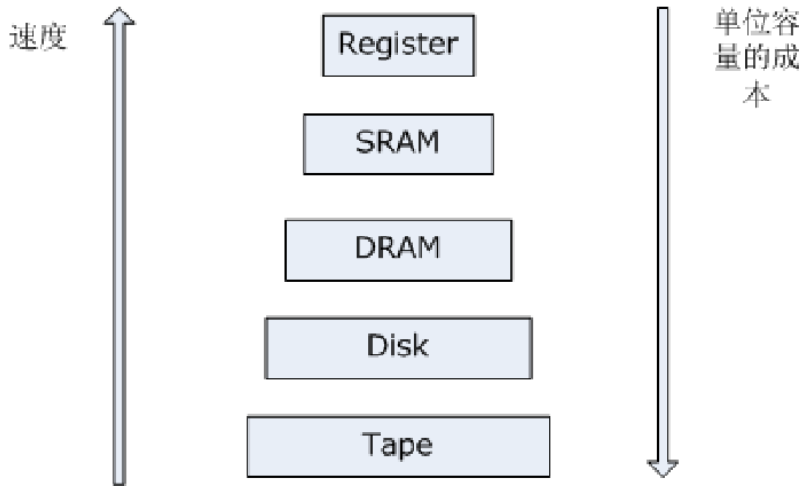


图 3-2 速度与单位容量成本

cache 是利用局部性原理，某个数据被访问，在不久的将来它很可能被再次访问，就是时间局部性，如循环中的代码和数据；其相邻的数据在不久的将来很可能被访问，就是空间局部性，如数组中的元素。

Cache 也有层级，如图 3-3 所示，SMP 处理器上每个核都有单独的 L1 和 L2 缓存，两个核共享 L3 缓存。通常，L1 与处理器核速度相同，而 L2 和 L3 则容量更大，但速度较慢，功耗也更低。

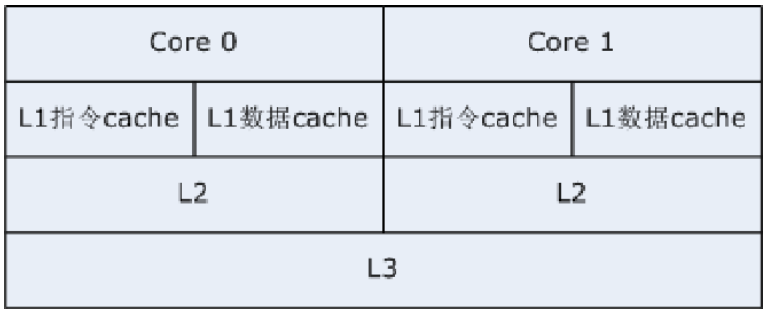


图 3-3 cache 层次结构

主存远大于缓存空间，因此主存与缓存行是多对一的映射。主存被划分成缓存行大小的数据块，映射方式有以下 3 种：

1) 全关联缓存：主存中的每个缓存行大小的数据能被映射到任意缓存行中；缺点是判断数据是否在缓存中时，需要同时匹配所有缓存行的 tag 和 valid，硬件实现成本较高。在 32 位系统中，地址线是 32 位，假设缓存行是 64 字节，则 tag 只需 26 位。
2) 直接映射缓存：主存中的数据块与缓存的映射是唯一的，数据块 0 只能映射到缓存行 0，取余依次类推。这也就意味着知道了主存地址，就可以知道该地址的数据被缓存在哪个行中。tag 记录数据块的索引即可。直接映射的问题在于某些情况下，缓存未命中的情况较严重，如下述代码，如果数组 a 和数组 b 刚好被映射到同一个缓存行，则 a, b 的每次数据访问都会发生缓存未命中。

```
for (i = 0; I < 10; ++i)
  c += a[i] + b[i];
```

3) 组关联缓存：由于全关联硬件实现复杂，而直接映射可能存在严重冲突，因此组关联是二者的优缺点的折中。它将缓存分成若干路，每路结构完全相同。每路采用全关联的方式，而每路中的缓存则采用直接映射。直接映射缓存就相当于一路组关联映射了。

当缓存行需要被置换的时候，有几种策略：随机置换，FIFO 置换，LRU 置换。通常处理器采用 LRU 方式，基于时间局部性，性能较好。

缓存工作方式

缓存读：cache hit/miss。

整个 cache 空间按照 cacheline 分成了若干行。cacheline 是缓存和主存交换数据的最小单位，通常为 32 字节或 64 字节等。cacheline 结构如图 3-4，valid 表示该行数据是否有效，tag 表示数据块在内存的地址，block 中存储的是主存中的数据。

cacheline 0	valid	tag	block
cacheline 1	valid	tag	block
cacheline 2	valid	tag	block
cacheline 3	valid	tag	block

图 3-4 cacheline 结构

高速缓存工作方式用缓存命中或未命中来描述。当处理器访问数据时，首先在 L1 数据

缓存中寻找，第一次没有找到，于是产生**缓存未命中**，相应的内存数据已 cacheline 大小被导入到某个缓存行中，并将地址写入 tag，置 valid 为 1。当再次访问时，处理器将访问的内存地址与 tag 进行匹配，且 valid 标志合法，则从缓存中读取数据，即**缓存命中**。

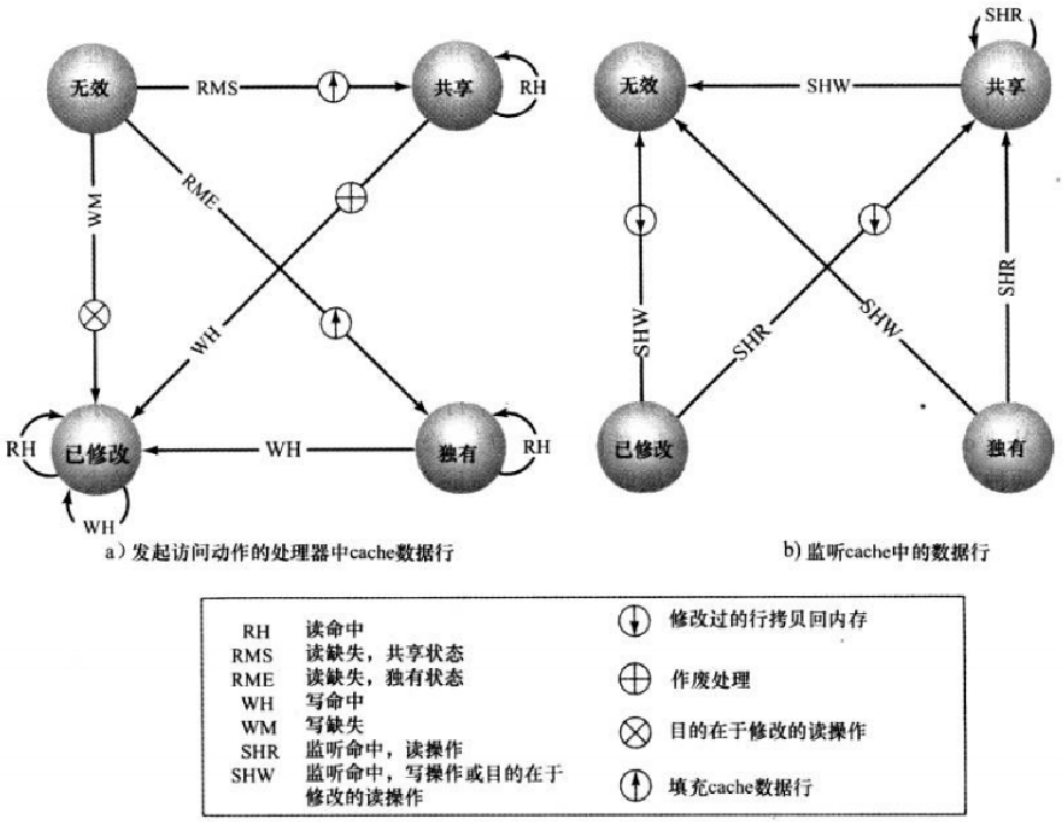
缓存写
当处理器修改缓存中的数据，通常的处理方式有通写 write through 和回写 write back。通写指每次处理器更新了缓存中的数据时，缓存立即更新主存中的值。回写是指处理器修改缓存中的内容后，并不立即更新主存中的值，而是等到该缓存行因为某种原因需要从缓存中移除时，才更新主存的内容。写通存在更多的内存访问，效率较低，因此大多数处理器都采用了回写策略。为了支持回写，缓存行增加了一个标志 dirty，1 表示该缓存行的内容与主存不同，0 表示相同。

在单核处理器上，缓存不存在一致性问题。但是对于多核处理器，数据被修改时就存在不一致的可能了。多核处理器需要协议和相关硬件来保证多核的缓存一致性。提供的硬件底层支持有 2 种方式：写无效 write invalidate 和写更新 write update。写无效指当一个处理器核修改了一份数据，其他处理器核的该数据副本的 valid 标志被写为无效。写更新则是指直接更新其他处理器核上的副本。

写无效不用更新数据，较简单，但会影响整个缓存行，导致本来有效的数据也变成无效了；而写更新会产生更多的更新操作。因此，多数处理器选用写无效方式。

为了保证缓存的一致性，除了硬件操作的支持，还需要相关的一致性协议。常用的有目录协议和监听协议，目录协议通过集中控制管理哪些缓存持有共享数据，在数据发生变更时，只针对持有的该数据的缓存发出控制命令，而监听协议则是广播到所有缓存。这里主要讨论采用写无效的监听协议。

在监听协议中，缓存行中包括一个两个状态位的标记：
M modified 该缓存行数据与主存不同，仅有该缓存行的数据是有效的
E Exclusive 该缓存行数据与主存相同，且在此缓存行中
S Shared 该缓存行数据与主存相同，数据存在于多个缓存中
I Invalid 该缓存行数据无效



对于不同的操作，状态转换描述如下

1) 读未命中，当处理器的缓存出现读未命中时，处理器在总线上发出一个信号，通知所有其他处理器的缓存单元该读事务。此时可能有以下情况
   a) 若另一处理器有处于专有状态 E 的该行副本，则该缓存行转换为共享状态 S，并通知发出事务的处理器共享该行数据；发出事务的处理器则从主存中读入数据，将该行数据由无效状态 I 转换为共享状态 S。
   b) 若有其他处理器处于共享状态 S 的该行副本，则它们都发出通知指示发出事务的处理器共享该行数据；启动事务的处理器则从主存中读入数据，并将状态转换为共享 S。
   c) 若另一处理器有处于修改状态 M 的该行副本，则阻塞存储器读，并将该行由处于 M 行的数据发给发出事务的处理器，该行转为共享 S；而发出事务的处理器接收数据行，将该行置为共享，并更新主存中的数据行。
   d) 如其他缓存都没有该行数据的副本，则没有信号返回；发出事务的处理器启动读主存操作，此行状态由无效转为专有 E。
2) 读命中，处理器直接取缓存数据，状态不发生变化。
3) 写未命中，此时处理器在总线上发送一个“用于修改的读”信号，根据其他处理器是否拥有该行被修改过的缓存数据，有 2 种情况，比读未命中时简单
   a) 有，则通知发出事务的处理器，发出事务的处理器放弃总线并等待，该处理器则将数据写回主存，此行状态变为无效 I；然后发出事务的处理器再次在总线上发出“用于修改的读”信号，然后从主存中读入该行数据，修改，状态变为修改 M。
   b) 没有，此时无信号返回。发出事务的处理器从主存读入数据并置为修改状态 M，并将其他处理器的该行缓存置为无效 I。

4) 写命中，若该行数据处于共享，则通过总线发出通知说我要修改该行，其他缓存将自身数据置为无效状态 I，然后再修改数据，并转换为修改状态 M；若为专有或修改，则状态不变，直接修改数据。

片内寻址存储器

缓存未命中分析
有 3 种：必须的未命中，第一次访问指令或数据时，这种未命中时不可避免的；容量未命中，是由于缓存容量有限导致的被置换的缓存行未命中；冲突未命中，是指虽然缓存虽然还有空闲空间，但映射方式导致该行已经被占用。

还有一种需要注意的情况是伪共享。

linux 中内存如何要求内存对齐？
```
int x __attribute__((aligned(64))) = 0;
struct foo {
   int x[2] __attribute__((aligned(64)));
};
```

动态内存分配时使用 memalign 或 posix_memalign

内存对齐
通常，编译器会按照一定的规则使内存对齐，有助于内存访问。但目前的 intel 处理器已经能够在不损失效率的情况下访问任何字节地址的内存。因此，程序员需要进行内存对齐的更好理由是充分利用缓存，缓存未命中的损失极大。

4. 指令级并行

流水线

流水线冒险
hazard
实际中，流水线的每个节拍都被充分利用并不可能，总会存在某些因素让流水线停顿。这就是流水线冒险。

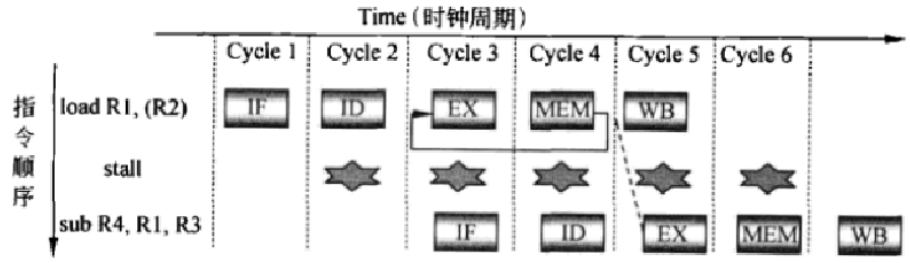
结构冒险
由于处理器资源冲突，而无法满足某些指令节拍的的组合被称为结构冒险。如在 MIPS 流水线中，指令和数据都放在存储器中，IF 需要访问存储器，MEM 阶段也需要访问存储器，因此当这两个指令位于同一个时钟周期时，其中一个指令需要等待。（当前的计算机在 L1cache 中，指令和数据是独立的，因此不存在这个问题）。

数据冒险 （延时是整条指令的延时？而不是某个节拍的延时？也就是指令开始执行前就已经知道冲突了？）
有数据依赖的指令出现的某些节拍组合导致使用错误的数据被称为数据冒险。
寄存器访问冲突

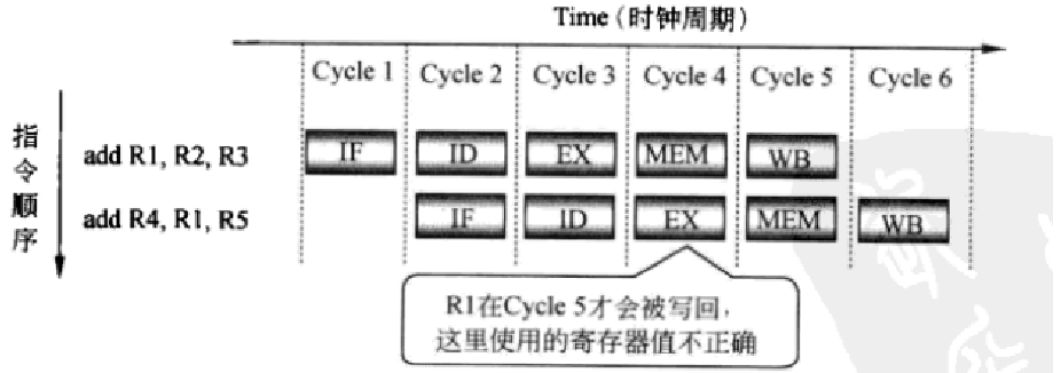
add r1, r2, r3
add r4, r1, r5

内存访问冲突
load r1, (r2)
store r1, (r3)

有些数据冒险冲突的解决，必然会引起延时

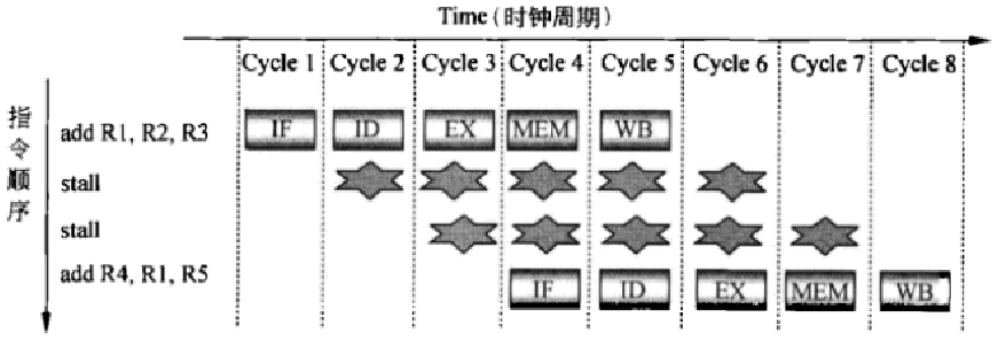


仍然需要 stall 的直通示例

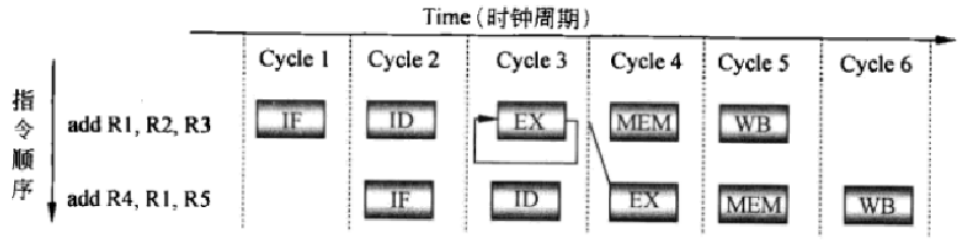


寄存器访问的数据冒险

解决方法有两种，延时和直通



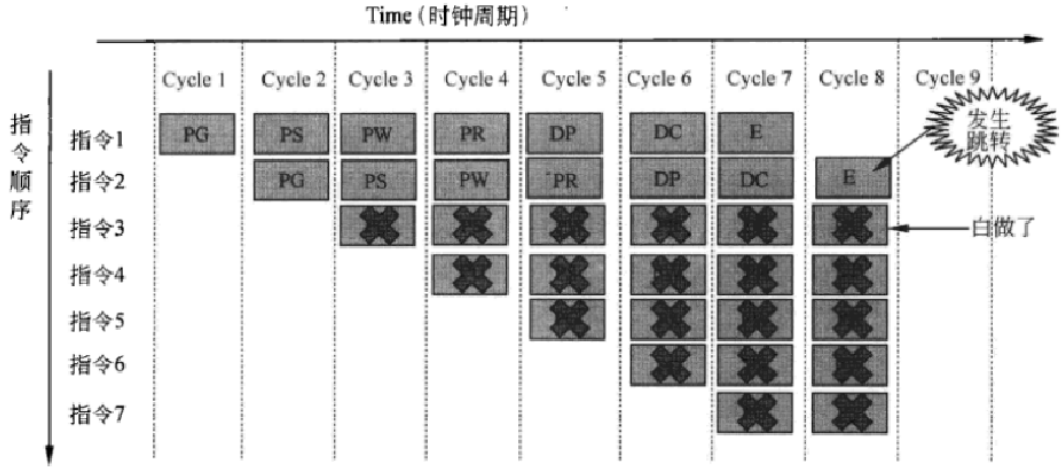
通过增加等待来消除数据冒险



使用 Forwarding，解决数据冒险

控制冒险

指令的节拍是并行处理的，然而在程序中存在很多跳转条件语句，如果程序的实际执行路径是要跳到其他地方去执行，那么流水线中的 IF、ID 等工作就白做了，被称为控制冒险。此时，处理器需要排空流水线，跳转到新的地址重新进入流水线，这对流水线的性能损失是很大的。



发生跳转时的性能损失

x86 中，发生跳转时使用硬件冲刷流水线；而 DSP 中，硬件不能处理，只能使用软件来冲刷流水线，通常在跳转语句后使用 nop 指令进行排空流水线。

分支预测

分支预测

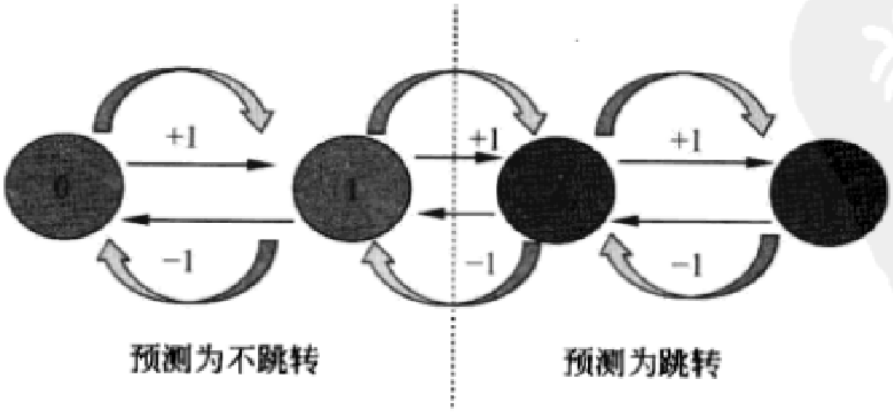
CPU 通过分支预测来避免控制冒险带来的损失。处理器使用算法猜测进入哪个分支，如果猜测正确，就节省时间，猜测错误，则排空流水线。

常用分支预测算法有 1 位预测，2 位预测。

1 位预测，如果跳转指令上一次发生跳转，就预测下一次也会跳转；否则不跳转。这对循环来说是很好，但对于 if 语句来说，50%的猜中率，性能不足。

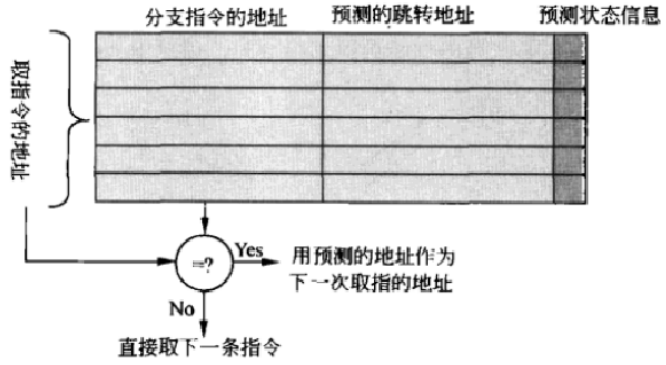
2 位预测，每个跳转指令预测的状态有 2 位，跳转执行，则加 1，加到 3 就不再增加；

如果跳转不执行，则减 1，减到 0 时不再减。计数器值为 0 和 1 时，预测分支不执行，为 2 和 3 时，预测分支执行。该算法比 1 位预测有更好的稳定性。



2 位预测

分支预测的实现
Intel 包含 3 个单元
分支目标缓冲区 BTB (Branch Target Buffer)
静态预测器 static predictor
返回栈 Return Stack



BTB(Branch Target Buffer)

分支指令在执行后，会将这条指令的地址及它的跳转信息记录在 BTB 中。BTB buffer 不会太大，不能将所有的分支指令都存进去，通常采用 Hash 表的方式存入。在取指时，先将 PC（程序指针）和 BTB 中的分支指令的地址进行比较，如果找到了，说明这条指令是分支指令，并且在 BTB 中有记录，就使用 BTB 预测出来的跳转地址。如果没有记录，就不能使用 BTB 的信息了，取指下一条指令。

The Static Predictor

当分支指令在 BTB 中记录了历史信息才能使用 BTB 进行预测，当分支在 BTB 中找不到记录时，可以使用 The Static Predictor（静态预测器）。人们将分支指令的执行情况做了大量的统计，从中总结出一些特征，并将这些特征总结为一些固定的策略，这就是静态预测器。

当指令被解码后，它是不是分支指令，以及要跳转的地方就知道了，只是不知道是否该跳。一般来说，向上的跳转，常用来组织成循环，这个跳转应该被预测为执行。

静态预测器通常的策略是：向下跳转预测为不跳转，向上跳转预测为跳转。

Return Stack

函数调用在程序中大量出现，函数调用与返回也都是通过跳转来实现的。例如，有 3 个函数调用了 printf 函数，printf 函数地址固定，调用时知道地方，但是在返回时，并不知道该返回到哪个地方，Return Stack（返回栈）可以用于解决这个问题。在函数调用时，将函数的返回地址压栈到 Return Stack 中，当遇到函数返回指令时，就从 Return Stack 中取出地址。

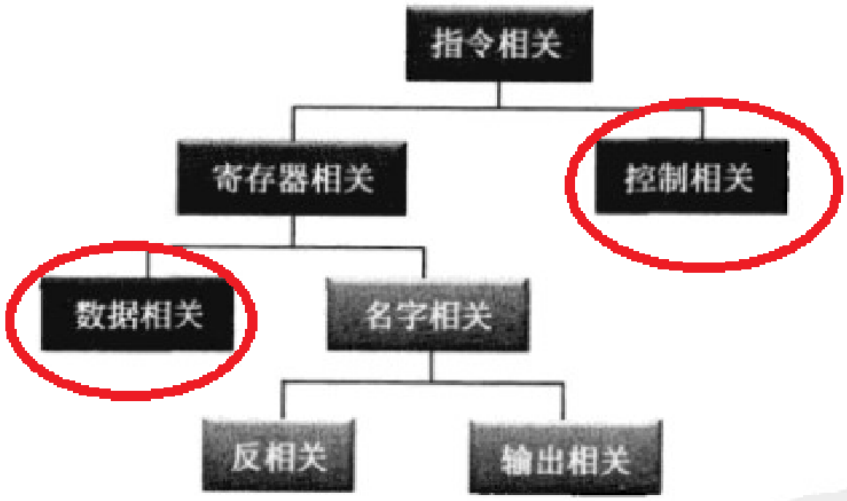
分支预测会消耗大量资源，许多低功耗的处理器没有分支预测，通常采用指令的条件执行来减少（不能消除复杂的跳转）指令的跳转。

乱序

指令执行时常因一些限制而等待，如缓存未命中（通常长达数百个时钟周期），导致后面的指令得不到执行，这是对 CPU 极大的浪费。此时处理器通常会先执行不依赖该数据的指令。这就是乱序执行。

乱序可能发生在编译器编译时以及处理器执行指令时。

指令的相关性分为寄存器相关和控制相关两大类。



指令的相关性

寄存器相关

两条指令共用寄存器时就可能存在寄存器相关。

```
; 先读后读，实际无相关性
add bx, ax
add cx, ax

; 先写后读 RAW Read After Write，数据相关
add bx, ax
add cx, bx

; 先读后写 WAR Write After Read，反相关，这是由于寄存器太少，指令不得不共用寄存器导致的
add bx, ax
mov ax, cx

; 先写后写 WAW Write After Write，输出相关，没有数据关联
mov ax, bx
mov ax, cx
```

WAR 和 WAW 没有逻辑上的相关性，只是由于共用了寄存器而导致相关，被称为伪相关或名字相关。

控制相关
如 xor 的执行结果依赖于 jnz 的结果，这种相关性是由指令的控制流决定的，因此被称为控制相关。

```
  cmp
  jnz LABEL
  ; …
LABEL:
  xor
```

去相关性
1．减少数据相关，虽然不能消除。这是程序员或编译器的工作。

```
x = a + b
y = x + c
z = y + d
```

可以修改成

```
x = a + b
y = c + d
z = x + y
```

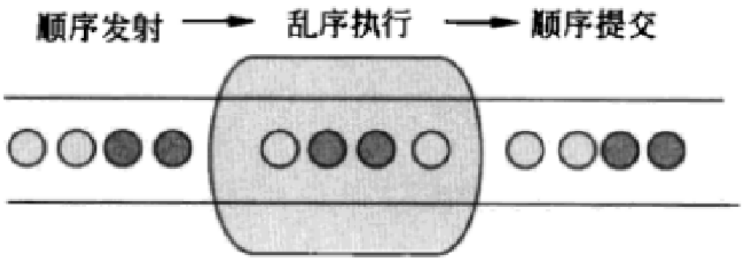
2．去控制相关
通常有两种策略，投机执行 speculative execution 和 eager execution。前者是利用分支预测执行，而后者将分支的两条路径都执行。后者资源消耗过高，一般乱序都采

用投机执行。

3. 去伪相关
ISA 寄存器不足，在处理器内部有更多的物理寄存器（对程序员不可见），通过将相同的 ISA 寄存器映射到不同的物理寄存器，去除相关性。

寄存器重命名的策略（参见《大话处理器》P.131）
??? ISA 寄存器和处理器物理寄存器书中没有介绍，只是带出了这么一个概念，在处理器中，这些究竟是怎么安排的？

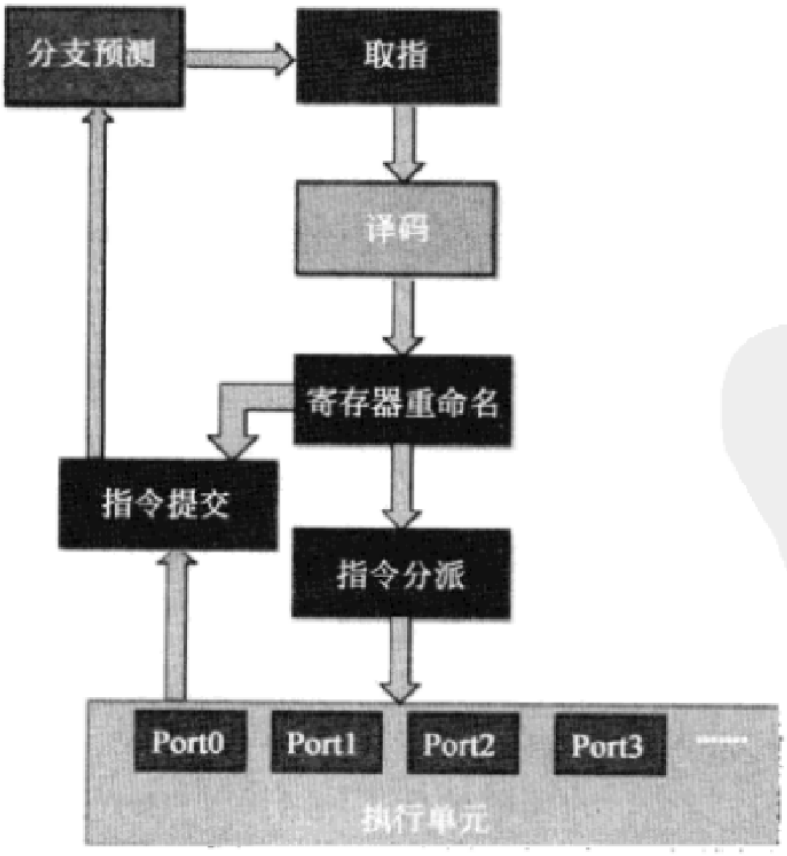
乱序执行基本方法



指令乱序执行步骤

之所以需要顺序提交是由于精确中断、投机执行出错时需要只执行中断前、分支前的指令，而其后的指令不能执行。为了完成这个步骤，处理器提供了指令缓冲区、重排缓冲区 ROB Re-Order Buffer，调度单元从指令缓冲区中选择就绪的指令执行，乱序执行完的指令并不立即生效（即写回 ISA 寄存器，而是仍在物理寄存器中），直到重排缓冲区顺序提交。

乱序执行基本结构



乱序执行内核结构图

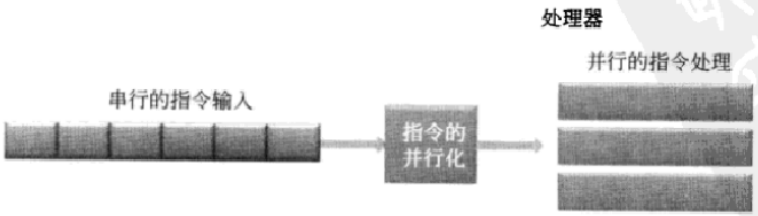
缓存和乱序执行对 IO 的影响

处理器和 IO 控制器之间有缓存，读出和写入命中缓存时，命令就无法传给 IO 控制器。考虑这种情况：为了启动输入设备，存储指令写入指令寄存器，然后用加载指令从数据寄存器中读取数据。此时若乱序执行颠倒了加载和存储指令的顺序，则无法得到预期的结果。因此 IO 控制器所在的物理页面必须设置不使用缓存，并利用内存屏障指令保证指令执行顺序不会颠倒。

超标量

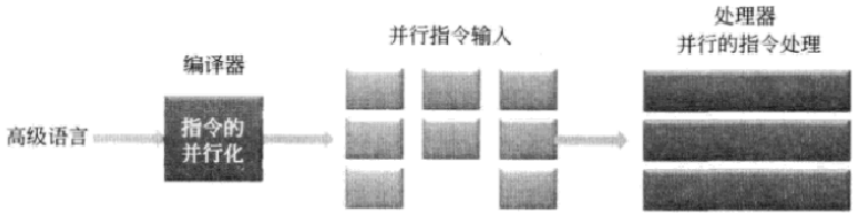
超标量

Superscalar 是由 super（超）+ scalar（标量）组成，标量处理器时代的指令都是串行执行的，处理器为了兼容原有的程序，但同时又要提高程序执行效率，就在处理器内部做了指令的并行化处理。这就是超标量处理器的基本原型。



Superscalar 的指令并行化在处理器内部实现

如果将指令的并行化显示的声明在指令格式中，处理器只是傻呼呼的执行，这种方式称为 VLIW（Very Long Instruction Word）。指令的并行化可由编译器完成，也可以由程序员手工写并行汇编代码实现。



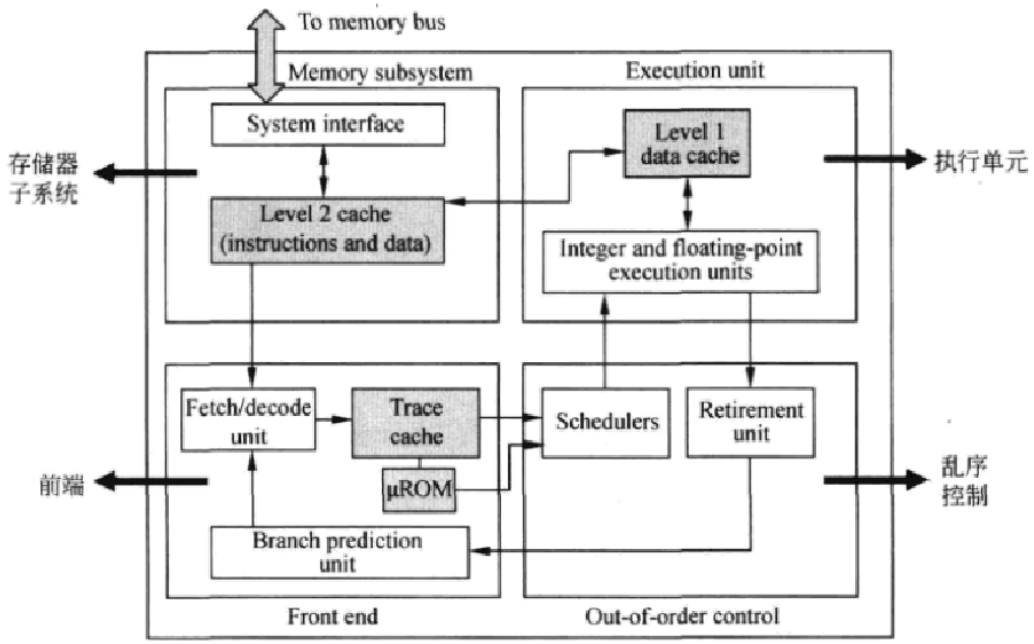
VLIW 的指令并行化在处理器外部实现

RISC 和 CSIC

Intel 中采用了 CSIC 指令，但是在其微架构中却转化为 RISC 的微指令进行执行。获取兼容性。

5. 数据并行

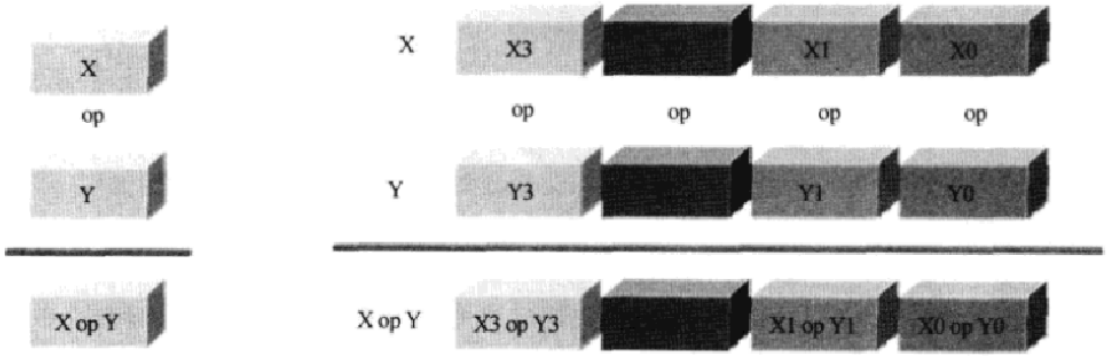
奔 4 的微架构 NetBurst



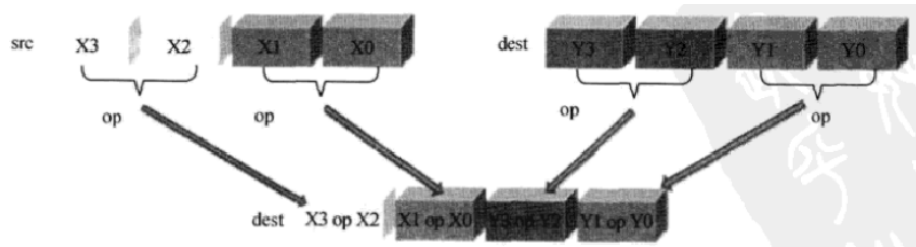
奔 4 处理器微架构的基本结构

左边是普通指令的计算形式，右面是垂直计算形式的 SSE 指令。

在垂直计算形式的 SSE 指令中，X 和 Y 寄存器被看成是一个向量，它包含多个标量数据，每个标量数据分别运算，这也是 SIMD 指令最常用的计算形式。



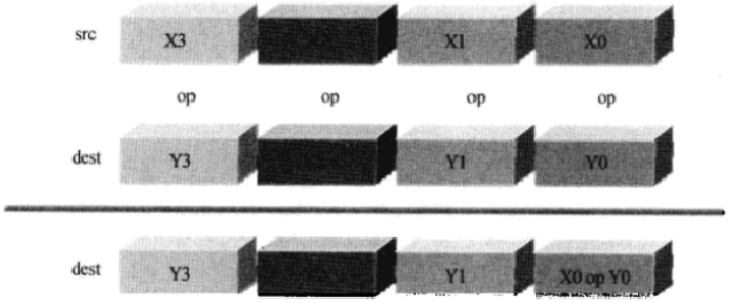
普通指令　　垂直计算形式的 SSE 指令



水平计算形式

在水平计算形式中，两个操作数均来自于同一个源，而不像垂直计算形式，两个操作

数来自于不同的源。

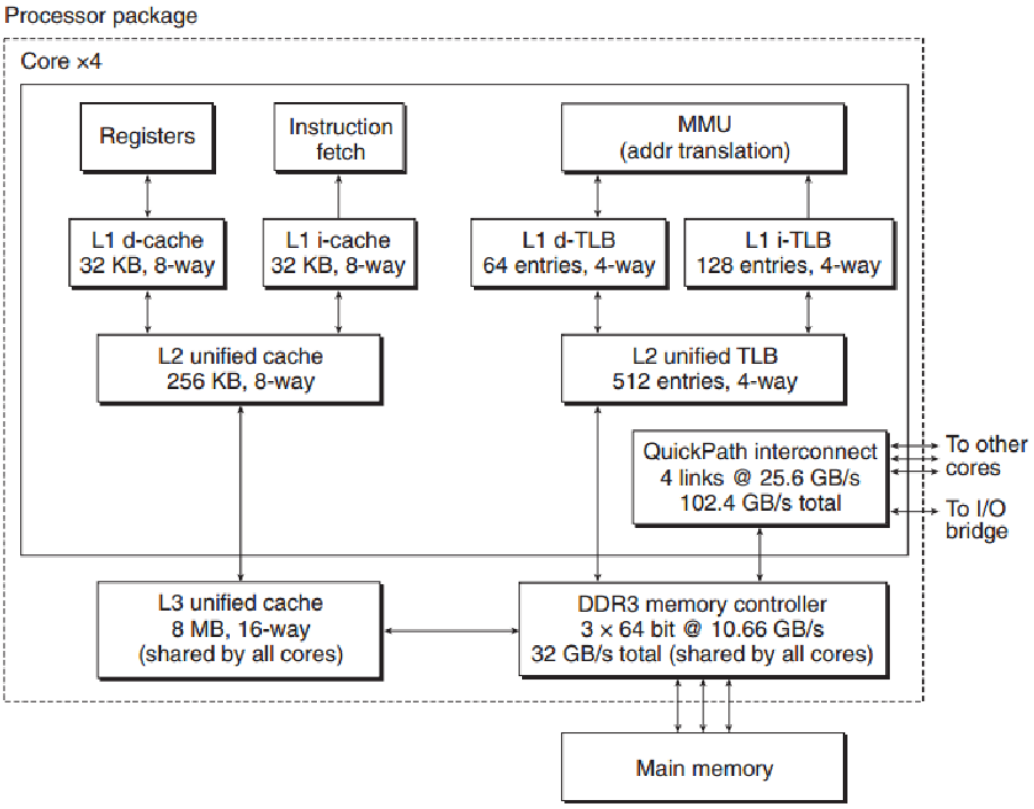


标量计算形式

SSE 指令还支持标量的运算方式，如上图所示：只有 x0 和 y0 进行操作，其他的元素保持不变。

GPU

6. core i7



7. 参考资料

[1] 大话处理器——处理器基础知识读本，万木杨。
[2] 支撑处理器的技术——永无止境地追求速度的世界，（日）Hisa Ando，李剑译。
[3] 计算机组成与体系结构——性能设计，第 8 版，William Stalling，彭蔓蔓等译。
[4] 计算机系统结构——量化研究方法，第 4 版，John L.Hennessy, David A.Patterson，白跃彬译。