Lecture 10 指令级并行

1. 多发射 Multiple issue

多发射介绍

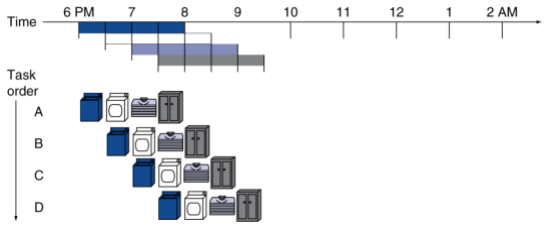
流水线挖掘了之零件的潜在并行性，这种并行性被称为**指令级并行 instruction-level parallelism ILP**

有两种方法可以增加指令潜在的并行性

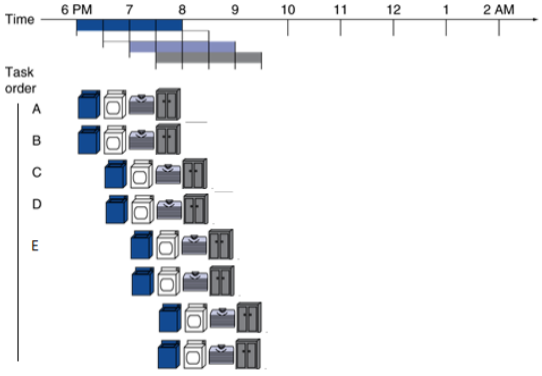
- 增加流水线的深度以重叠更多的指令
    - 每级更少的工作，可以获得更短的时钟周期数
- 复制计算机内部部件的数量，使得每个流水级可以启动多条指令，这种技术一般被称为**多发射 multiple issue**
    - 每个阶段同时启动多条指令允许指令执行速率超过时钟速率，即 CPI < 1，所以有时候也用 IPC 表示

多发射的效果如下

- 单发射流水线



- 多发射流水线



多发射总结

1. 实现方式
    a. 静态多发射 static multiple issue
    b. 动态多发射 dynamic multiple issue
2. 多发射流水线需要解决的目标
    a. 往**发射槽 issue slot** 中发射多条指令
    b. 处理**数据冒险和控制冒险**
3. 推测的实现机制
    a. 推测是一种编译器或处理器推测指令结果以消除执行其它指令对该结果依赖的方法
4. 推测错误的恢复机制
    a. 推测的问题在于可能会猜错，所以，推测技术必须包含一种机制
5. 由于推测引起的异常处理
    a. 推测还可能引入新的异常，对于某些指令的推测会导致原本不存在的异常发生
    b. 当异常真正发生的时候，就会执行异常处理程序

	静态	动态
时间	编译时	在执行时
调度者	编译器（软件）	处理器（硬件）
往发射槽中发射多条指令	一般是由编译器处理的	一般是由处理器在运行中处理（编译器可以调整指令顺序加以协助）
处理数据冒险和控制冒险	部分甚至全部的数据冒险和控制冒险是由编译器静态处理的	通过硬件技术在执行时消除某些类别的冒险
推测	编译器可以重新排序指令 改变取数指令和其他指令的顺序 改变分支指令和其他指令的顺序 包括一些从错误推测中恢复的“修复”指令	硬件可以预先查找要执行的指令 缓冲结果，直到确定它们确实需要 对不正确的推测刷新缓冲区
推测引起的异常	为了延迟异常可以提供一些新的 ISA 支持	处理器缓存异常，直到导致异常的指令确定会被执行（因为有的不一定会出现）

将在下面的部分介绍静态多发射和动态多发射

	静态多发射	动态多发射
决策由	编译器决定（软件）	处理器决定（硬件）
也叫	Very long instruction VILW	Superscaler
避免冒险的方式	循环展开 / 寄存器重命名	乱序执行

误区

1. 流水线很容易？不一定
    a. 基本的想法很简单
    b. 细节决定成败
2. 流水线与技术不相关？错
    a. 更多的晶体管使更先进的技术成为可能
    b. 与流水线相关的 ISA 设计需要考虑技术趋势
3. 糟糕的 ISA 设计会使流水线更加困难！
    a. 因为流水线 RISC 的性能才能比 CISC 好

2. 静态多发射处理器 static multiple issue

介绍

静态多发射处理器使用**编译器**来帮助封装多条指令并处理冒险

- 编译器将同时发出的指令分组，称为**发射包 issue packet**
- 将它们打包到**发射槽 issue slot** 中
- 编译器检测并避免冒险

发射包 issue packet

- 在静态发射处理器中，可以在给定时钟周期内发射多条指令
- 它可以被视为一条完成多个操作的长指令，所以最初叫做**超长指令字 Very Long Instruction Word，VLIW**

编译器的任务主要包括

- 静态分支预测
- 代码调度

以减少冒险或组织所有冒险

静态双发射

考虑简单的 MIPS 处理器中的简单的双发射指令，其中

- 一条指令是整型 ALU 操作/ 分支之后零
- 一条指令是装载 / 存储指令

要求两条指令成对地放在一个 64-bit 对齐的内存区域中，并且 ALU 指令或分支指令必须放在前面

指令类型	流水线阶段							
ALU 或分支	IF	ID	EX	MEM	WB			
load 或 store	IF	ID	EX	MEM	WB			
ALU 或分支		IF	ID	EX	MEM	WB		
load 或 store		IF	ID	EX	MEM	WB		
ALU 或分支			IF	ID	EX	MEM	WB	
load 或 store			IF	ID	EX	MEM	WB	
ALU 或分支				IF	ID	EX	MEM	WB
load 或 store				IF	ID	EX	MEM	WB

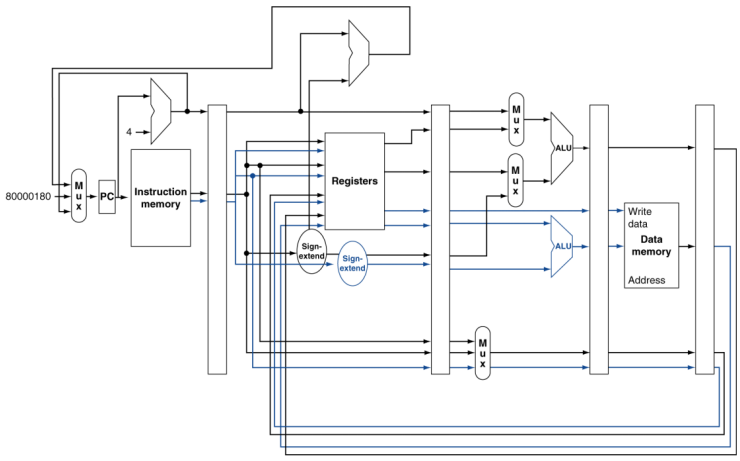
调度静态多发射

编译器有不同的处理数据冒险和控制冒险的方式，必须删除一些 / 所有的冒险

- 将指令重新排序到发射包中
- 包内指令没有依赖关系
- 包之间可能存在一些依赖关系
    - 不同的指令集有区别，编译器必须直到
- 如果找不到另一条同时发射的指令，可以用 `nop` 指令代替它

静态双发射数据通路

静态双发射数据通路如下：



为了实现上述情况，首先需要增加一些硬件

- 冒险检测和阻塞逻辑
- 寄存器堆的额外端口
- ALU 操作读两个以上寄存器
- 存储操作读两个以上寄存器
- 因为 ALU 要用来进行 ALU 指令的操作，所以需要一个额外的 ALU 来计算数据传输的有效地址

静态双发射的冒险

双发射处理器最多将性能提升两倍，实时上，额外的重叠使用数据冒险和控制冒险带来的相对性能损失也增加了

EX 的数据冒险

- 在单发射中，可以通过前推来避免
- 在双发射中，如果有这样的指令集

  ```
  add  $t0, $s0, $s1
  load $s2, 0($t0)
  ```

  它们不能放在一个发射包中，必须分开，实际上是阻塞

取数-使用冒险

- 与单发射一样，有一个时钟周期的**使用延迟 use latency**（在装载指令与可以无阻塞使用其结果的指令间相隔的时钟周期数）
- 因为发射包，下一发发射包的两条指令同时被阻塞

例题3 双发射流水线调度

在一个 MIPS 静态双发射流水线中，下面这个循环将如何调度？

```
Loop:
    lw   $t0, 0($s1)
    addu $t0, $t0, $s2
    sw   $t0, 0($s1)
    addi $s1, $s1, -4
    bne  $s1, $zero, Loop
```

对上述指令进行重排序，以尽可能地避免流水线阻塞，假设分支是可预测的，即控制冒险由硬件处理

前三条指令和后两条指令存在数据相关性，我们的安排如下

	ALU 或分支指令	数据传输指令	时钟周期
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, -4	`nop`	2
	addu $t0, $t0, $s2	`nop`	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

5 条指令每次循环花费 4 个时钟周期，与最好情况下 0.5 CPI/ 2.0 IPC 相比，最佳情况下 CPI = 0.8 / IPC = 1.25

注意，`nop` 指令不用算到有效指令中，它没有办法提高真实的性能

循环展开 loop unrolling

有一种重要的从循环中获得更多性能的编译技术叫**循环展开 loop unrolling**，循环体被复制多份，通过重叠不同循环体中的指令可以获更高的指令级并行 ILP

- 复制循环体以实现更多的并行性
- 每次复制使用不同的寄存器
  称为**“寄存器重命名” register renaming**
- 减少了循环控制开销

例题4 双发射流水线的循环展开

试对例题 3 的例子进行循环展开和调度

为了无延迟地循环的调度，我们把循环体指令复制 4 份，在展开和消除不必要的循环体开销指令后，将得到 4 给备份

	ALU 或分支指令	数据传输指令	时钟周期
Loop:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
	`nop`	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
	`nop`	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

在循环展开的过程中，编译器使用寄存器重命名，引入了几个临时的寄存器（$t1，$t2，$t3），消除一些虚假的数据依赖，即一个指令序列与下一个指令序列之前没有数据流动，这叫做反相关 **antidependence** 或被称为名字相关 **name dependence**

循环指令中 14 条指令花费了 8 次时钟周期，即 $CPI = \frac{8}{14} = 0.57$ ，$IPC = \frac{14}{8} = 1.75$ 双发射加上循环展开与调度使得性能提升了将近两倍

这种性能的提高代价是使用了 4 个临时寄存器，同时增长了代码长度

3. 动态多发射处理器 dynamic multiple issue

介绍

动态多发射处理器通常也称为超标量处理器，或者简称**超标量 supersclar**

- CPU 检查指令流并选择每个周期发射的指令
- 编译器可以帮助重新排序指令，但不一定需要帮忙调度
- CPU 在运行时使用高级技术解决冒险
- 最简单的超标量处理器中，指令顺序发射，每个周期处理器决定是发射 0 条、1 条还是多条指令

        - 避免结构和数据危险
    - 硬件支持指令执行时候的重新排序
    - 允许 CPU **不按顺序**执行指令，以避免中断
        - 但是必须**按顺序提交**结果到寄存器

许多超标量处理器扩展了基本的动态发射决策，将**动态流水线调度 dynamic pipline scheduling** 也包含在内，动态流水线调度选择某个时钟周期内将执行的指令，约束条件是尽量不产生冒险和阻塞，它是对指令进行重排序以避免阻塞的硬件支持

考虑下面的指令序列

```
1 lw   $t0, 20($s2)
2 addu $t1, $t0, $t2
3 sub  $s4, $s4, $t3
4 slti $t5, $s4, 20
```

- sub 指令其实可以先执行，但是由于前面的 addu 在等待 lw，sub 也被阻塞，如果内存很慢的话，sub 指令可能会等待很多个时钟周期，动态流水线调度可以完全避免这种冒险

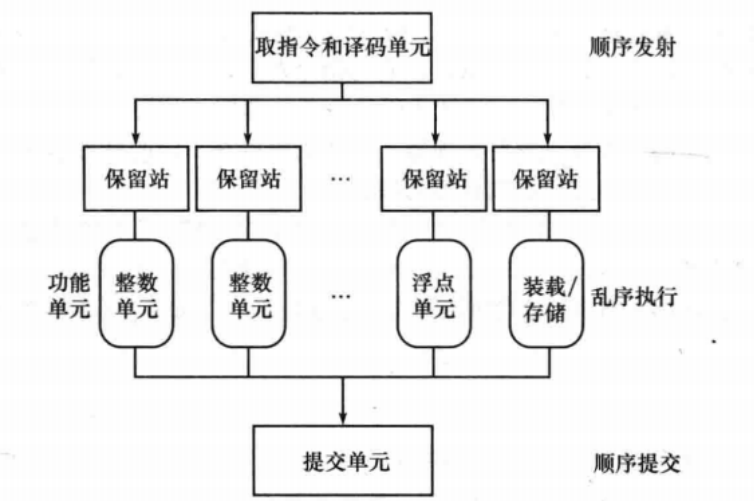
动态流水线单元介绍

动态流水线调度选择下一条要执行的指令，可能的话会重排指令以避免阻塞

在这种处理器中，流水线被划分为 3 个主要单元

- 取指和发射单元
- 多个功能单元
- 提交单元 commit

下图描述了这种构造



1. 第一个单元**取指并译码**，然后将每条指令发送到**相应的功能单元**执行
    a. 不同类型的指令被发送到不同的功能单元去
2. 每个功能单元都有自己的缓冲区，被称为**保留站 reservation station**，用来**保存操作数和操作**
    a. 如果一个指令需要的数据还没有被计算好，它会先在保留站中等待
3. 当缓冲区**包含了所有的操作数，且功能单元就绪**时，结果就被计算出来
4. 结果得到后，被发送到等待该结果的**保留站和提交单元**
5. 提交单元**缓存**这个结果，在确定安全时，再将这个结果写回寄存器或存储器
    a. 提交单元的缓冲区称为**重排序缓冲区 recorder buffer**，它也可以用来提供操作数，工作方式类似于静态调度流水线中的旁路逻辑
    b. 因为要按照顺序提交，一些提前被计算出来的值会存在重排序缓冲区中
6. 一旦结果写回寄存器堆，其又可以从寄存器堆中直接被取出，和一般的流水线完全一样

发射方式

顺序发射 in-order issue

- 取值和译码单元按顺序发射指令，以记录程序中的依赖关系

乱序执行 out-of-order execution

- 处理器在不违背程序原有的数据流顺序的前提下以某种顺序执行各条指令
- 执行的顺序可以和取指的顺序不同

顺序提交 in-order commit

- 提交单元按照程序顺序将结果写回寄存器堆和存储器

为什么需要动态流水线调度

- 不是所有的阻塞都可以在编译时被预测
    - 比如 cache miss 的时候的调度问题
- 不能总是围绕分支安排时间
- ISA 的不同实现具有不同的延迟和冒险

4. 能耗效率

动态调度和推测的复杂性需要强大的能力

下图给出了一些处理器的流水线级数、发射宽度、推测级别、时钟频率、每芯片的核数和功耗等

- 流水线级数多，则时钟频率高，功率高

从单核发展到多核时流水线级数和功耗逐渐减少

处理器	年份	时钟频率	流水线级数	发射宽度	乱序/推测	核数目/片	功耗	
Intel 486	1989	25MHz	5	1	No	1	5	W
Intel Pentium	1993	66MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2 000MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3 600MHz	31	3	Yes	1	103	W
Intel Core	2006	2 930MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3 300MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3 400MHz	14	4	Yes	8	77	W

ARM Cortex-A8 和 Intel Core i7 流水线

ARM Cortex-A8 和 Intel Core i7 是后 PC 时代的标志性产品

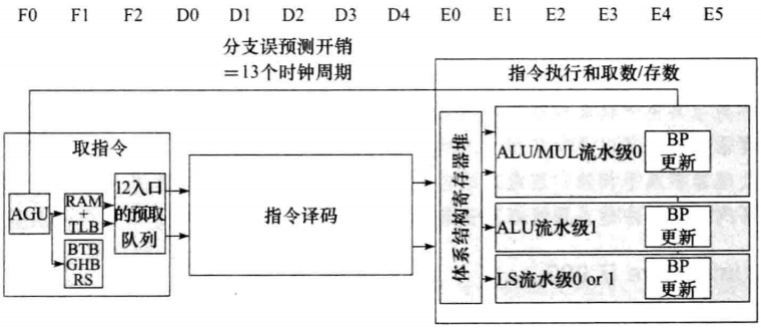
处理器	ARM A8	Intel Core i7 920
市场	个人移动设备	服务器，云计算
功耗	2W	130W
时钟频率	1GHz	2.66GHz
核/芯片	1	4
浮点	无	有
多发射	动态	动态
峰值指令数/周期	2	4
流水线级数	14	14
流水线调度	静态顺序	动态乱序猜测执行
分支预测	2 级	2 级
1 级 cache/核	32KiB 指令，32KiB 数据	32KiB 指令，32KiB 数据
2 级 cache/核	128～1 024KiB	256KiB
3 级 cache（共享）		2～8MiB

ARM Cortex-A8

ARM Cortex-A8处理器主频为 1GHz，具有 14 级流水线。它采用动态多发射技术,每个时钟周期可以发射两条指令，其流水线为**静态顺序流水线**，指令发射、执行和提交顺序执行。流水线包含**取指令、指令译码和执行**三个阶段

流水线介绍

下图给出了流水线的整体情况



- 取指令有 3 个流水级
    - 12 入口的指令预取缓存取入指令
    - 地址产生单元 AGU
    - 分支预测，使用分支目标缓存 BTB、全局历史缓存 GHB 和返回栈 RS
- 指令译码有 5 个流水级
- 指令执行有 6 个流水级，提供 3 条流水线，3 条流水线间在在执行级具有全旁路机制
    - 1 条用于 load 和 store 指令的流水线（任何指令都可以发射到 load-store 流水线）

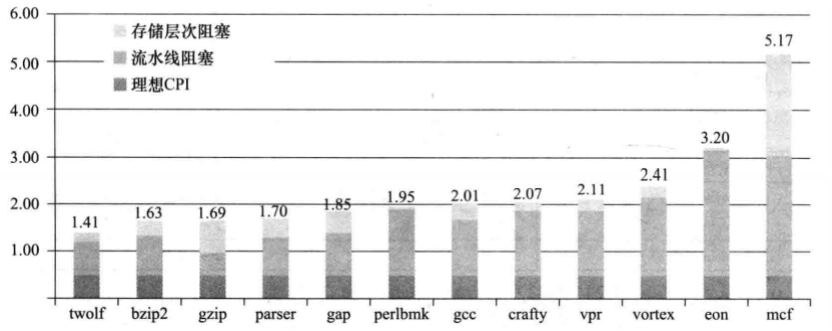
- 2 条算数操作流水线，其中只有第一对可以处理乘法

执行流程

1. 取指令的 3 个流水级每次取入两条指令，使得能够存放 12 条指令的预取缓存区维持满状态
2. 使用一个两级分支预测器（分支目标缓存 BTB、全局历史缓存 GHB）和一个用于预测未来返回操作的返回栈，进行预测
3. 当分支预测错误时清空流水线，导致 13 个时钟周期的误预测开销
4. 译码阶段的 5 个流水级确定一对指令间是否存在可导致顺序执行的相关，并且确定将指令送往哪条执行流水线
5. 执行阶段的 6 个流水级执行指令，进行相应的操作

使用 SPEC2000 衡量性能

下图给出了 ARM Cortex-A8 使用 SPEC2000 基准程序中衍生出来的一个小版本程序集的 CPI



流水线阻塞的情况

- 80% 的流水线阻塞源于冒险（分支误预测、结构冒险和之指令对间数据相关）
- 20% 源于存储器层次

Intel Core i7

介绍

x86 采用了复杂流水线的技术，其 14 级流水线综合使用了动态多发射、乱序执行和推测执行的动态流水线调度技术

由于 x86 指令级是 CISC，处理器面临着实现复杂 x86 指令级的挑战

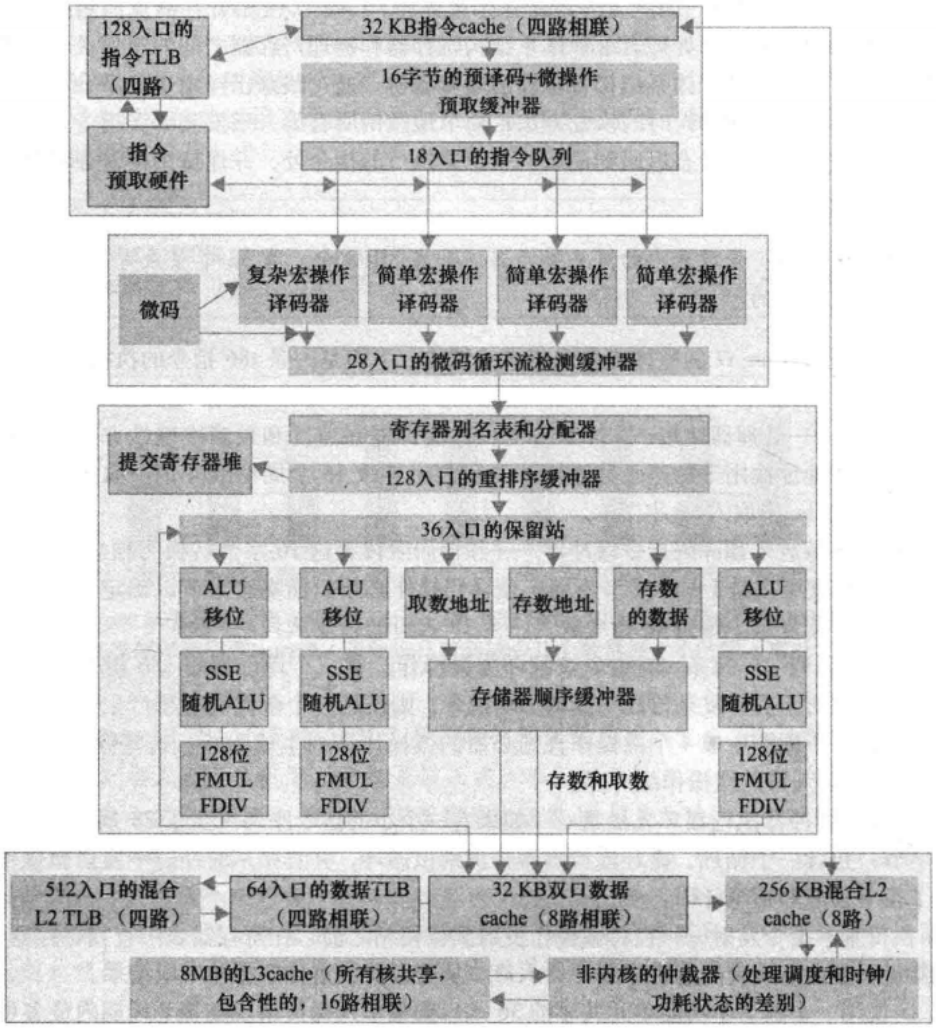
Intel 取入 x86 指令，将其翻译为类 MIPS 指令，Intel 称之为**微操作**，它由复杂的基于推测执行的动态调度流水线执行，该流水线每时钟周期最多可执行 6 个微操作

由于动态调度处理器设计中，功能单元、cache 和寄存器堆、指令发射和整个流水控制的设计混杂在一起，使得把数据通路和流水线分开变得很困难，因此我们用**微体系结构 microarchitecture** 来描述处理器内部体系结构的细节

Intel Core i7 使用**重排序缓冲区**和**寄存器重命名技术**来解决反相关和推测错误，寄存器重命名显式地将处理器中的**体系结构寄存器 architectural register**（处理器中可间的寄存器） 重命名为一组**更大的物理寄存器集合**

Core i7 使用寄存器重命名技术来消除反相关，它需要处理器维护体系结构寄存器和物理寄存器之间的映射关系，这也提供了一种预测错误时的恢复方法，即简单的**撤销所有第一条推测错误指令后所建立的所有映射**即可

整体组合和流水线结构

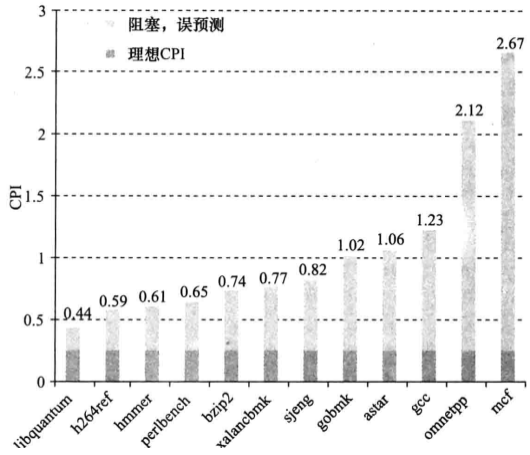


它的步骤如下

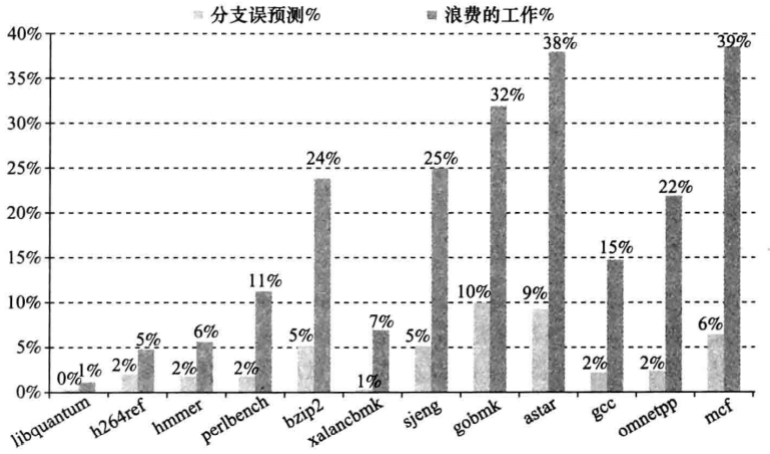
1. 取指令——处理器使用一个多级分支目标缓冲器在速度和预测准确性方面做平衡，另外还有一个返回地址栈用于加速函数返回。误预测将导致 15 个周期的开销，取指部件使用预测地址从指令 cache 中取入 16 字节
2. 这 16 字节放入预译码指令缓冲器——预译码阶段将这 16 字节转换为独立的 x86 指令，因为 x86 指令长度可以是 1-15 字节不等,预译码操作必须扫描多个字节以确定指令长度，所以预译码操作非常复杂，每条 x8 指令放入一个 18 入口的指令队列
3. 微操作译码——每条 x86 指令被翻译为微操作，有三个译码器将 x86 指令直接翻译为一个微操作，而对于具有复杂语法功能的x86指令，则使用一个微代码引擎产生一个微操作序列；它可以在每个周期生成 4 个微操作直到必需的微操作序列生成为止，这些微操作按照 x86 指令顺序放入 28 入口的微操作缓冲器
4. 微操作缓冲器执行循环流检测——如果有一个小的指令序列（少于 28 条指令或长度小于 256 字节）包含一个循环，循环流检测器将识别该循环，并直接从缓冲器中发射微操作，从而减少了指令预取和指令译码
5. 执行基本指令发射——在将微操作发射到保留站之前，在寄存器表中查找寄存器位置、对寄存器进行重命名、分配重排序缓冲器入口、从寄存器或重排序缓冲中取结果
6. i7 使用一个被 6 个功能单元共享的 36 入口的集中式保留站，在每个周期内最多可以向功能单元分派 6 个微操作
7. 各个功能单元执行微操作，执行结果不但送往寄存器提交部件，在已知指令将不再预测的情况下更新寄存器状态，还可送往任何一个等待的保留站，重排序缓冲中与指令对应的入口标记为完成
8. 当前面的一条或多条指令已经被标记为完成，则执行寄存器提交部件中未决的写操作，指令从重排序缓冲器中移走

使用 SPEC2006 衡量性能

下图给出了 Intel Core i7 使用 SPEC2006 基准程序中衍生出来的一个小版本程序集的 CPI



下图显示了 Intel Core i7 分支误预测的比例和所有微操作分派相关的工作中未提交（即它们的结果被取消）的比例


分支误预测%
浪费的工作%
40%
35%
30%
25%
20%
15%
10%
5%
0%
0%
1%
2%
5%
2%
6%
2%
11%
5%
24%
1%
7%
5%
25%
10%
32%
9%
38%
2%
15%
2%
22%
6%
39%
libquantum
h264ref
hmmer
perlbench
bzip2
xalancbmk
sjeng
gobmk
astar
gcc
omnetpp
mcf