处理器体系结构

第四章 处理器的微架构E

--多发射技术

(Micro-processor Architecture)

目录

4.5.1 多发射技术基础--流水线小结与比较

指令	IF	ID	EX	MEM	WB	单周期处理器	流水线处理器
lw	200ps	100 ps	200ps	200ps	100ps	**800ps**	**1000ps**
sw	200ps	100 ps	200ps	200ps		**700ps**	**1000ps**
R-format	200ps	100 ps	200ps		100ps	**600ps**	**1000ps**
beq	200ps	100 ps	200ps			**500ps**	**1000ps**

单周期处理器：时钟周期800ps，CPI=1

流水线处理器：

- **不存在“取数-使用”相关，Load指令CPI为1；存在“取数-使用”相关，则Load指令CPI=2；若两种情况各占50%，则Load指令CPI？：1.5**
- **Store指令CPI=1**
- **Branch指令预测失败25%，失败时CPI=2，成功时CPI=1，所以平均1.25**
- **Jump指令CPI=2**

4.5.1 多发射技术基础--流水线的平均指令周期

MIPS指令在IntSPEC中出现的频率：

25%为Loads
10%为Stores
11%为Branches
2%为Jumps
52%为ALU指令

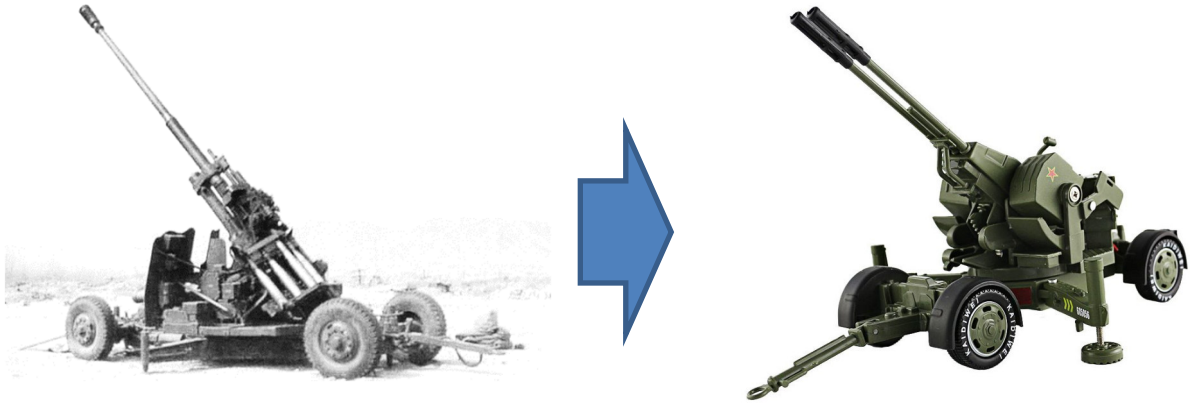
平均指令周期：1.5*25%+1*10%+1.25*11%+2*2%+1*52%=1.17

	CPI	周期长度	指令执行时间
单周期处理器	1	800ps	800ps
流水线处理器	1.17	200ps	234ps

思考：评价性能好坏的唯一标准?

->CPI能否更小? or IPC能否更高?

(如何进一步提高指令的并行性？)



指令级并行

- 增加流水线的深度，以能容纳更多的指令
- 复制处理器的资源，在每一个流水线段发射更多的指令，也称之为多发射

  -允许多发射，意味着指令的执行速度超过处理器的时钟频率 CPI<1 ，反过来，IPC>1

  -高端处理器可以每个周期发射3~8条指令

一个6GHz 4 路多发射处理器，指令执行速度是多少(每秒执行多少指令)？其IPC 或是CPI 多少？假定流水线是5 级流水，问在一个周期内有多少指令被处理?

(1) 指令的执行速度

6GHz ×4 = 24G 条指令/ 秒 = 24 billion/ 秒

(2) IPC=4/1=4 条指令/cycle， CPI = 1/4=0.25cycle/ 条指令;

(3) 一个周期内有4 ×5 = 20

4.5.1 多发射技术基础--实现多发射的2种方法

问题：发射几条指令和发射哪些指令?

静态多发射（Static multiple issue）：在程序执行前，由编译器做出决定

动态多发射（Dynamic multiple issue）：在程序执行中，由处理器做出决定

4.5.1 多发射技术基础--实现多发射的2种方法

发射槽：一个位置，在这个位置每个时钟周期能发射多条指令。

将指令打包到发射槽中：

- **在静态发射的处理器中，“发射几条指令和发射哪些指令？”的决定是由编译器来完成。**
- **在动态发射的处理器中，“发射几条指令和发射哪些指令？”是由处理器硬件完成的，编译器只是帮助形成一个有利于多发射的指令序列。**

数据和控制冒险的处理：

- **在静态发射的处理器中，一些数据和控制冒险由编译器来处理**
- **在动态发射的处理器中，处理器在执行时，利用硬件技术，如Forwarding or Bypass来至少减轻或缓和一些种类的冒险**

尽管2 种发射各有特点，但在实际的应用中，互相借鉴；实际上，一种方法会被另一种方法所借用，因为没有一种方法可称之为完美的。

4.5.1 多发射技术基础--推测恢复机制

推测：编译器或处理器“猜测”指令的特性，从而在执行该指令的时候，开始执行其他可能与其相关的指令

例如:

- 我们可能推测分支的结果，从而允许分支后的指令更早地执行
- 我们可能推测store-load的数据地址不同，从而允许load在store前执行

推测的难度在于：推测可能会错误，所以所有推测机制都必须包含检查和恢复方法，从而增加处理器复杂度

4.5.1 多发射技术基础--推测恢复机制

推测可以通过**编译器**或**硬件**实现：编译器可以通过推测来重新排序指令，硬件也可以实时调整指令顺序(后面再讨论)

恢复机制的软件/硬件实现则非常不同：

- 对于软件推测，**编译器**通常插入额外指令来检查推测精度，并提供修复线程
- 对于硬件推测，处理器**硬件**缓冲(buffer)推测执行的结果，直到其确定所执行的指令不再是推测的：如果推测正确，则将buffer结果写入寄存器或内存；如果推测错误，则Flush掉buffer，重新执行正确的指令

4.5.1 多发射技术基础--推测所引起的异常问题

推测可能会产生一个潜在问题：一些指令的推测可能会产生异常（原本不会发生）。比如：假设一个load指令经推测被移动了，但推测错误导致其执行的时候地址非法，从而形成本不该产生的异常

正确地使用推测机制会提高处理器的性能，反之，会使处理器的性能下降。

目录

4.5.2 静态多发射

静态多发射处理器使用编译器来打包指令，处理冒险

在一个周期内发射的指令束（若干条指令），称之为发射包。静态多发射指令通常限制发射的指令类型，可以把一个发射包看作为含有某些预定义字段的多个操作的长指令。

- 超长指令字 VLIW （Very Long Instruction Word ）就是典型的静态多发射处理器技术
- Intel IA-64 采用了这种VLIW 结构， 最先实现在安腾和安腾2 处理器，也叫显式并行指令计算机（EPIC-Explicitly Parallel Instruction Computer ）

大多数静态处理器都使用编译器来处理部分数据和控制冒险，如分支预测或代码调度，来减少和预防冒险。

4.5.2 静态多发射--MIPS ISA下的静态多发射

MIPS 的2发射处理器，发射的是2条指令的组合：

- 一条指令是ALU或是Branch
- 另一条指令是load或store

->2 条指令同时取指和译码需要64位的宽度

在静态多发射或VLIW 处理器中，同时发射2 条指令的格局应给予限制以简化译码和指令发射的设计

- 指令成双并且以64 位为界
- ALU 或branch 指令打头，load 或store指令随后
- 如果某条指令不可用，就用nop指令替换

4.5.2 静态多发射--静态2发射流水线操作

ALU 或branch 指令打头

load 或store指令随后

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 6.44 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

其中的一对发射指令

4.5.2 静态多发射--静态多发射的冒险处理

静态多发射处理器对潜在的数据冒险和控制冒险，有不同的处理方式：

方法1 ： 编译器承担了全部的责任：如避开所有的冒险、调度代码、插入nop 指令使得代码的执行不需要检测冒险或产生硬件停顿

方法2 ： 利用硬件检测冒险并在2 个“发射包”间产生停顿，同时利用编译器检测，以避免在“发射的指令对”中的所有相关

4.5.2 静态多发射--静态多发射的硬件配置

采用方法2：用硬件检测冒险，用编译器避开 “发射的指令对“的所有相关

- **首先，需要检测冒险和停顿逻辑**
- **其次，指令cache 需要具有2 个读端口**
- **再次，寄存器堆需要具有4 读2 写**
- **最后，还要增加一个ALU 单元，用来计算load 或store 指令的地址**

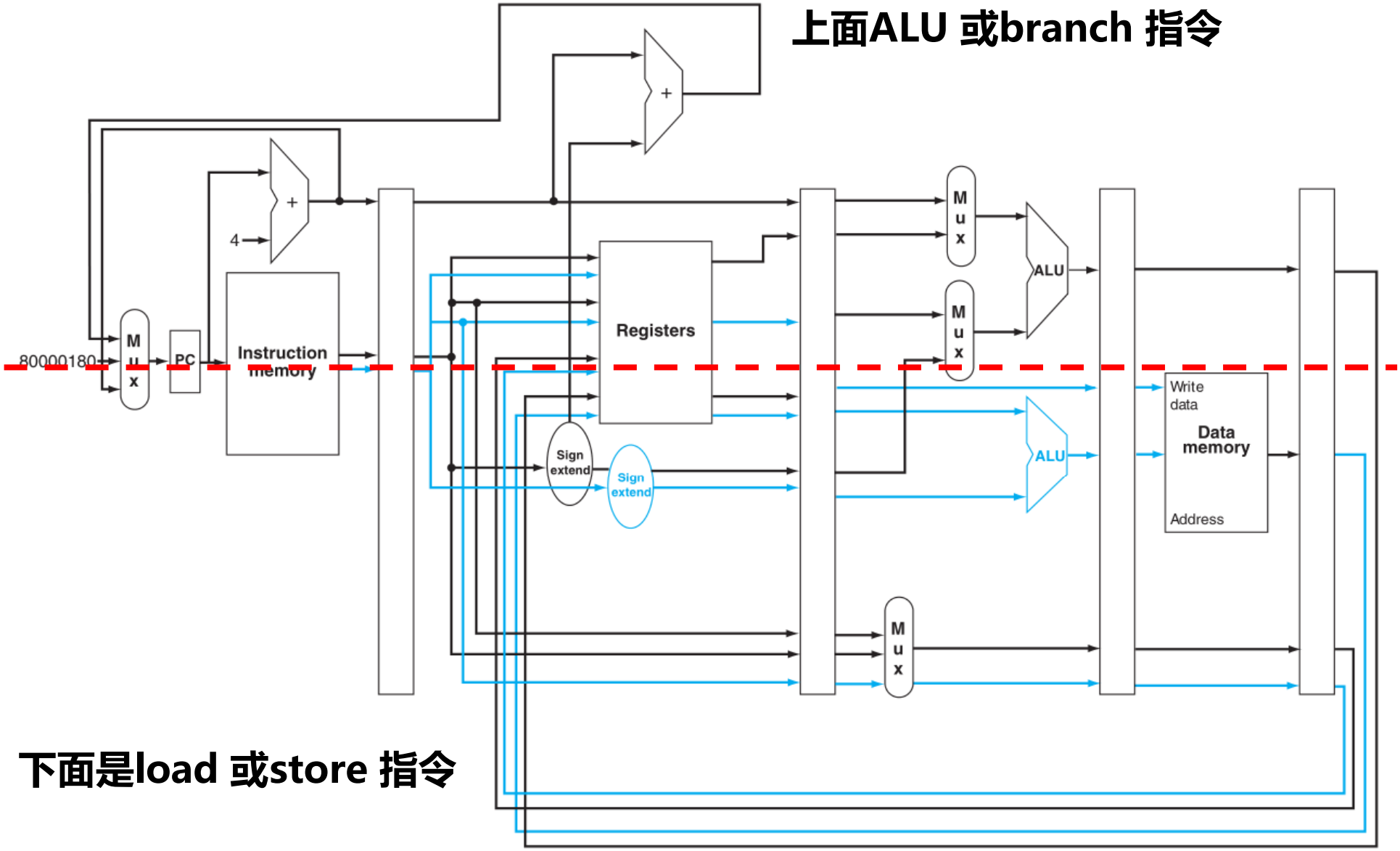


FIGURE 6.45 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

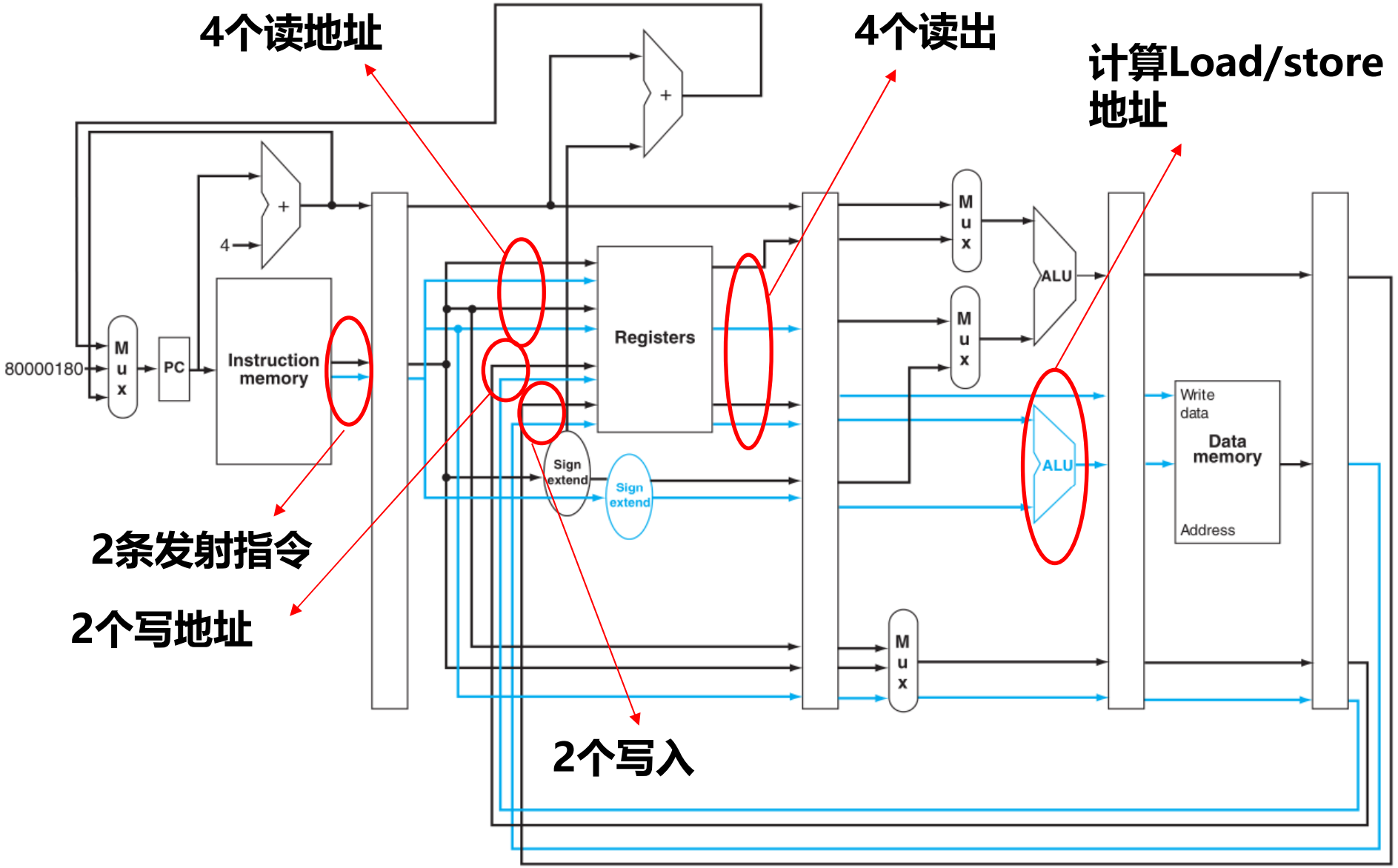


FIGURE 6.45 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

4.5.2 静态多发射-- 2发射所带来的问题

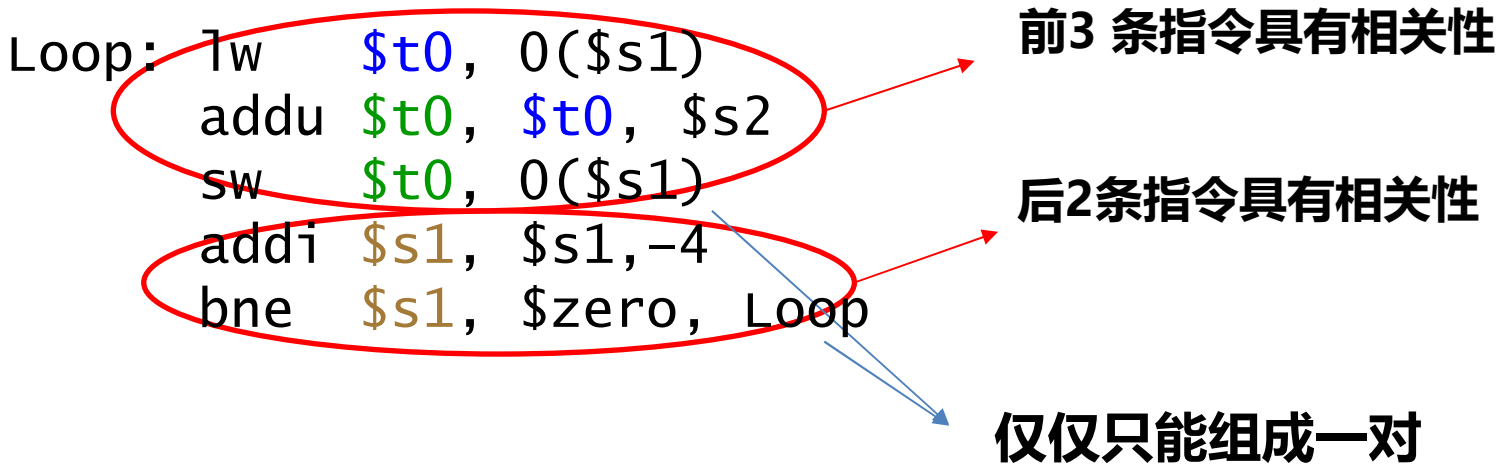
2 发射可提高处理器的性能2 倍，前提条件是有2个可重叠的指令，这势必会增加由**数据和控制冒险**所带来的相对的性能损失，例如：

- 对于2发射的5级流水线处理器，取数指令后面的两条指令都不能使用取数结果(“取数-使用”型冒险)
- 由于所发射的指令对中的load 或store 指令，不可能利用与其配对的ALU 的运算结果，所以需要延迟

为了有效发挥多发射处理器的并行性，需要开发更强大的**编译器或硬件调度技术**

4.5.2 静态多发射--举例：简单的多发射代码调度

例：如何调整下列代码序列，使其达到最佳次序以适应静态2 发射MIPS



4.5.2 静态多发射--调整后的指令序列

	ALU/branch	Load/store	cycle
Loop:	nop	①lw $t0, 0($s1)	1
	④addi $s1, $s1,-4	nop	2
	②addu $t0, $t0, $s2	nop	3
	⑤bne $s1,$zero, Loop	③sw $t0, 4($s1)	4

发射包：ALU 指令或Branch打头，load或store垫后
4 对指令中，只有一对指令占满了2个发射槽

性能改进不明显：

- **单发射处理器需要5个时钟周期， 2发射处理器需要4个时钟周期；**
- **对于2发射处理器，理想CPI为0.5，然而上述实际CPI为0.8；**

->如何解决?

循环展开技术：编译器通过循环展开，即把n次循环展开成一种顺序执行的代码，这样，就可从展开的代码序列中，发现并挖掘更多的并行的指令。

4.5.2 静态多发射--循环展开技术

假设循环4次：

```
loop:  lw $t0, 0($s1) ;
       addu $t0, $t0, $s2;
       sw $t0, 0($t1);
       addi $s1, $s1, -4;
       bne $s1, $zero, loop;
```

改变偏移量

×4

```
loop:  addi $s1, $s1, -16;
       lw $t0, 16($s1) ;
①      addu $t0, $t0, $s2;
       sw $t0, 16($s1);
       lw $t0, 12($s1) ;
②      addu $t0, $t0, $s2;
       sw $t0, 12($s1);
       lw $t0, 8($s1) ;
③      addu $t0, $t0, $s2;
       sw $t0, 8($s1);
       lw $t0, 4($s1) ;
④      addu $t0, $t0, $s2;
       sw $t0, 4($t1);
       bne $s1, $zero, loop;
```

循环展开后，①~④中的$t0并不是真正的数据依赖，即：可将②~④的$t0用其它寄存器替代，而不会影响结果

①->②、②->③、③->④之间，并没有真正的数据流存在，这种关系叫作名字相关(name dependence)，也叫反相关(anti-dependence)

->名字相关与真正的相关混杂在一起，怎么解决?

4.5.2 静态多发射--循环展开技术

寄存器重命名：通过编译器或硬件删除名字相关，而保留了真正的相关

```
loop:   addi $s1, $s1, -16;
        lw $t0, 16($s1) ;
        addu $t0, $t0, $s2;
        sw $t0, 16($s1);
        lw $t0, 12($s1) ;
        addu $t0, $t0, $s2;
        sw $t0, 12($s1);
        lw $t0, 8($s1) ;
        addu $t0, $t0, $s2;
        sw $t0, 8($s1);
        lw $t0, 4($s1) ;
        addu $t0, $t0, $s2;
        sw $t0, 4($t1);
        bne $s1, $zero, loop;
```

```
loop:   addi $s1, $s1, -16;
        lw $t0, 16($s1) ;
        addu $t0, $t0, $s2;
        sw $t0, 16($s1);
        lw $t1, 12($s1) ;
        addu $t1, $t1, $s2;
        sw $t1, 12($s1);
        lw $t2, 8($s1) ;
        addu $t2, $t2, $s2;
        sw $t2, 8($s1);
        lw $t3, 4($s1) ;
        addu $t3, $t3, $s2;
        sw $t3, 4($t1);
        bne $s1, $zero, loop;
```

在循环展开的过程中，寄存器重命名可以使这些独立的指令达到更好的调度代码

4.5.2 静态多发射--循环展开技术

	ALU/branch	Load/store	cycle
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, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

循环不展前：20 条指令，有4条组成一对，花费16个周期，CPI=16/20=0.8

循环展开后：14 条指令，有12 条组成一对，花费8 个周期，CPI = 8/14=0.57

成本：增加了代码量（长度）

4.5.2 静态多发射--循环展开技术

	ALU/branch	Load/store	cycle
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, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

思考：把指令调到该位置行吗？为什么？

不行，因为发生load-use 相关，必须停顿1 拍

目录

动态多发射处理器

- 动态多发射处理器，也称之为超标量处理器，或简称超标量
- 超标量：一种先进的流水线技术，这种技术可以使处理器在一个时钟周期内执行多于1条以上的指令
- 超标量处理器仍需要编译器协助来移开数据相关的指令，以达到超标处理器较高的处理效率

4.5.3 动态多发射(超标量)处理器--超标量处理器和VLIW处理器的区别

VLIW处理器：采用静态多发射，“发射几条指令和发射哪些指令？”的决定是由编译器来完成，编译器完成代码的调度

->同样的代码在不同结构的处理器上不能运行，或执行效率非常低，需要重新编译

超标量处理器：采用动态多发射，“发射几条指令和发射哪些指令？”的决定是由硬件来完成，硬件完成代码的调度 (动态流水线调度)

4.5.3 动态多发射(超标量)处理器

动态流水线调度是超标量处理器动态多发射框架的基本组成部分，可以在给定的一个时钟周期内选取要执行的指令，从而避开冒险和停顿

考虑下面的代码：

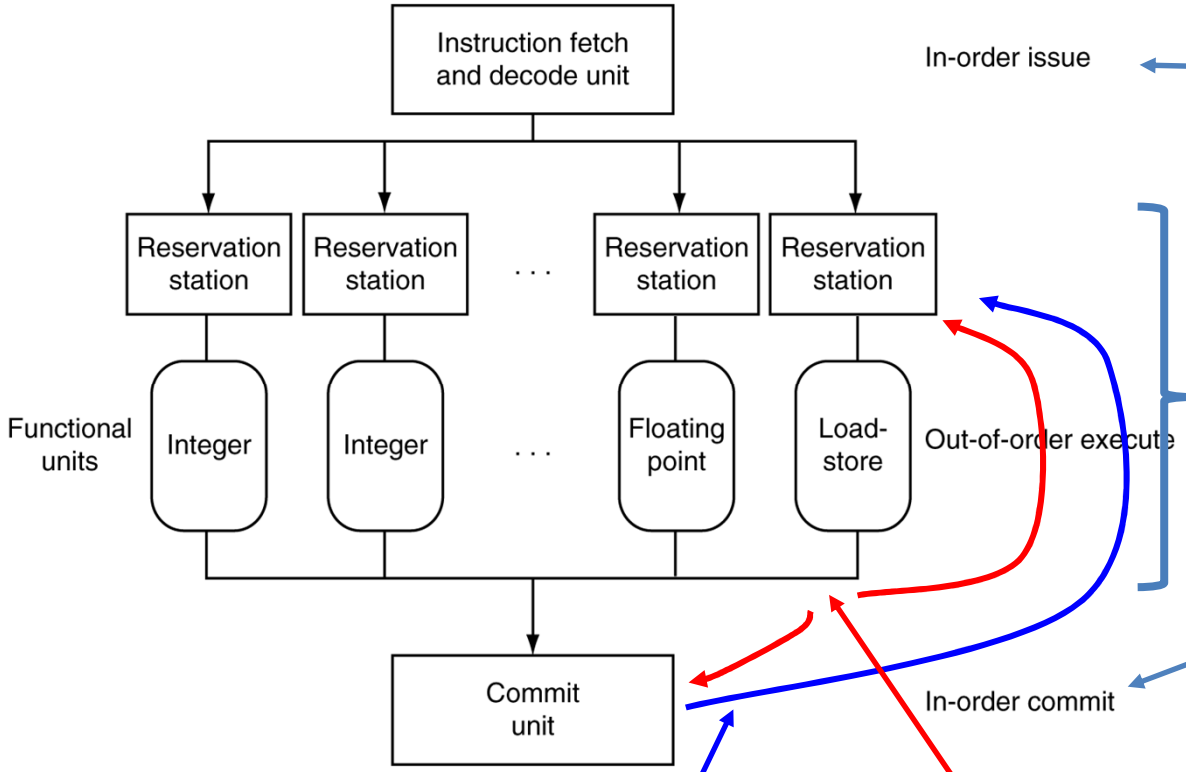
```
lw $t0,20($s2)
addu $t1,$t0,$t2
sub $s4,$t1,$s3
slti $t5,$s4,2
```

通常，sub指令的执行需要等待lw和addu指令执行完毕。如果lw访问存储器很慢（可能发生cache miss），那么sub指令会等很长时间

动态流水线调度会重新排序指令的执行，从而部分地或全部地避开这种停顿

4.5.3 动态多发射(超标量)处理器--动态流水线调度结构

动态调度的流水线可分为3 个部分：



取指和发射单元：主要负责取指令，译码，派送指令给相应的功能单元去执行

多功能单元：每个功能单元包含一个功能部件和一个buffer；这个buffer叫保留站，用于保存操作数和操作类型；保留站中的操作数一旦就绪，就立马启动后面的功能单元

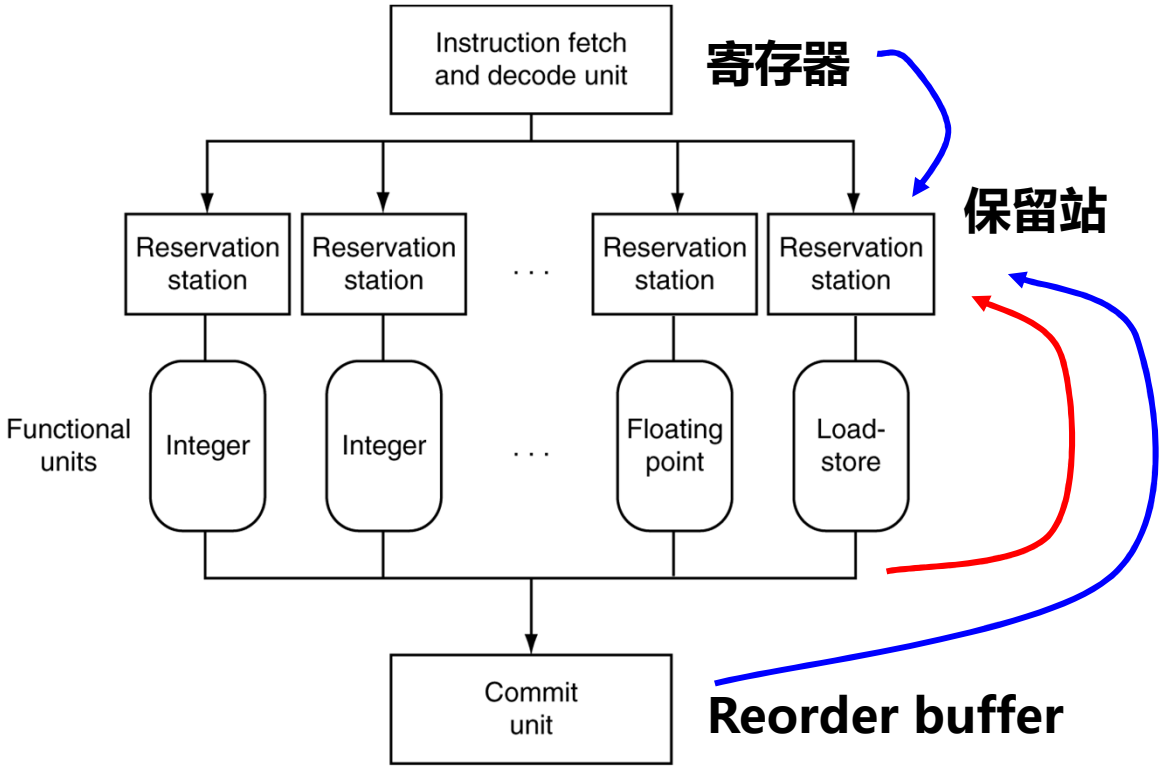
提交单元：缓存结果直到安全的时候再写回寄存器堆或存储器。提交单元中buffer，又称之为reorder buffer

可充当forwarding logic 的作用，为其它的功能单元提供操作数。

功能单元计算出结果后，就会将其发送到任何等待该结果的保留站或提交单元

4.5.3 动态多发射(超标量)处理器--动态流水线中的寄存器重命名

保留站中的操作数和提交单元中的计算结果的结合，事实上起到寄存器重命名的作用。



发射一条指令后，若操作数在寄存器堆或在提交单元的reorder buffer 中，该操作数就立即拷贝到对应的保留站中，进行缓存，直至其它所有的操作数就绪和功能单元可用为止。一旦计算完毕，保留站中备份的操作数就不再需要了。

如果操作数不在寄存器堆或reorder buffer中，那么就要等待某个功能单元产生所要的操作数。在此期间，硬件会跟踪或监控产生此操作数的功能单元，一旦监控的功能单元产生出操作数后，就绕过寄存器直接将操作数拷贝到对应的保留站

4.5.3 动态多发射(超标量)处理器--动态调度的流水线一些策略

动态调度流水线中的要点：

- **取指和译码是按程序顺序执行，同时跟踪数据的相关性　　有序发射**
- **功能单元的执行是乱序执行，只要操作数就绪就立马执行　乱序执行**
- **提交单元写寄存器和写存储器也是按程序的执行序列执行　有序提交**

若发生异常，处理器会指向最后执行的一条指令，并且保证只有发生异常指令以前的指令可以修改寄存器（只有发生异常以前的指令有效）

->如何实现推测/恢复?

4.5.3 动态多发射(超标量)处理器--动态调度中的硬件推测机制

回顾：推测/恢复机制可以由编译器或硬件实现

- **动态调度会沿着推测的路径不断取指和执行指令**
- **由于是有序提交，所以在提交前可以知道推测成功与否**

  方法1：在功能单元执行期间就立即修改寄存器，而使用额外的寄存器实现重命名和保存老的寄存器值，直到更新寄存器的指令不再是预测性的

  方法2：先把结果缓存在reorder buffer，实际寄存器的更新作为提交单元的任务在后来更新

- **动态调度支持推测load 和store 是否会发生地址冲突，以便重排load - store 的指令序列，而使用提交单元避开不正确的推测**

*发掘ILP的辩证观点

发掘ILP的潜力，有2种方法：

- **基于软件的方法，依赖于编译器来寻找和减少这种相关性的影响**
- **基于硬件的方法，依赖于动态调度流水线和发射机制**

推测可以由编译器或硬件来完成，这都可增加ILP 并行度，但要小心推测失败时所带来的性能损失

流水线和多发射执行可以提高指令吞吐率的峰值，并可开发出更多 ILP （指令级并行），然而，处理器数据相关的问题限制了其速度。现代高性能的处理器能够每个周期发射若干条指令，不幸的是，维持这种发射速率很难的，没有多少应用程序能支撑每周期2 条以上的指令发射。原因有2 点：

- 流水线的瓶颈来源于不能缓解的相关性，这就降低了程序的并行度和发射速率。常常是编译器或硬件不可能知道相关是否存在，只能保守的认为相关存在。程序中总是存在ILP ，但这些ILP 存在比较分散，编译器和硬件找到这些ILP 的能力是有限的
- 存储器系统的性能损失也会限制流水线全速地运转

*小结

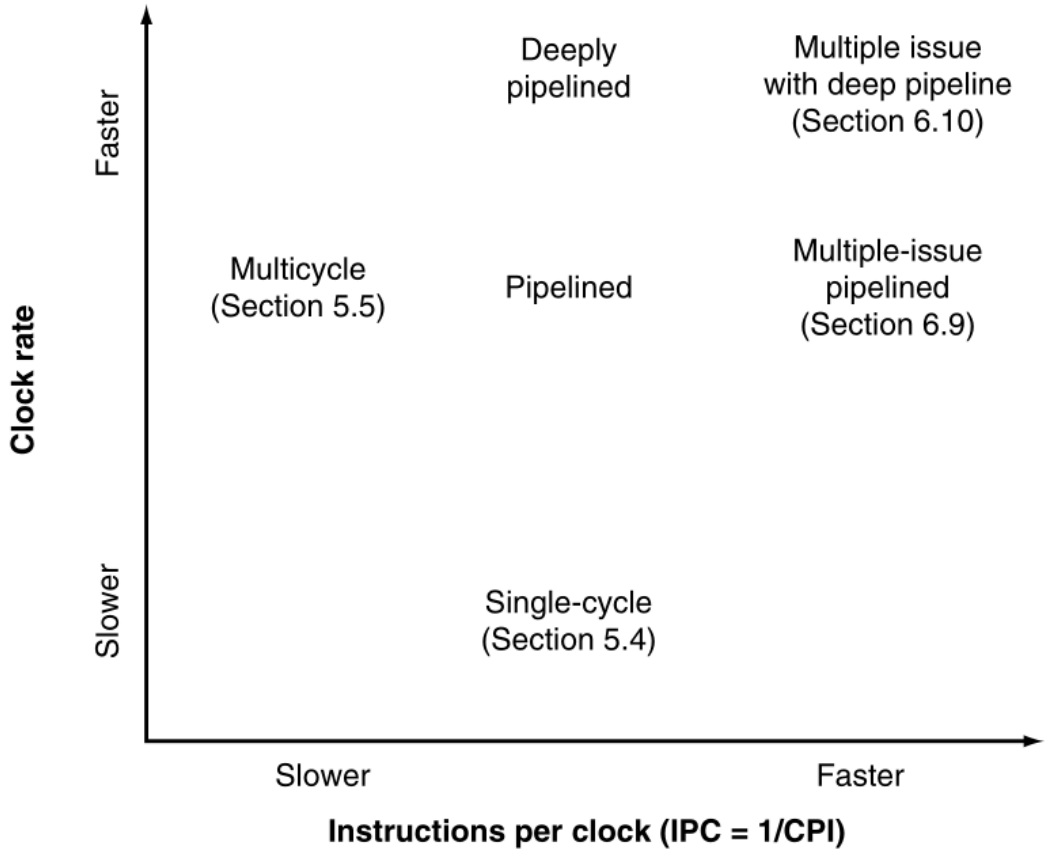


FIGURE 6.52 The performance consequences of simple (single-cycle) datapath and multicycle datapath from Chapter 5 and the pipelined execution model in Chapter 6. Remember that CPU performance is a function of IPC times clock rate, and hence moving to the upper right increases performance. Although the instructions per clock cycle is slightly larger in the simple datapath, the pipelined datapath is close, and it uses a clock rate as fast as the multicycle datapath.

*小结

流水线的每级都有专用硬件支持

多发射需要添加更多的专用硬件支持

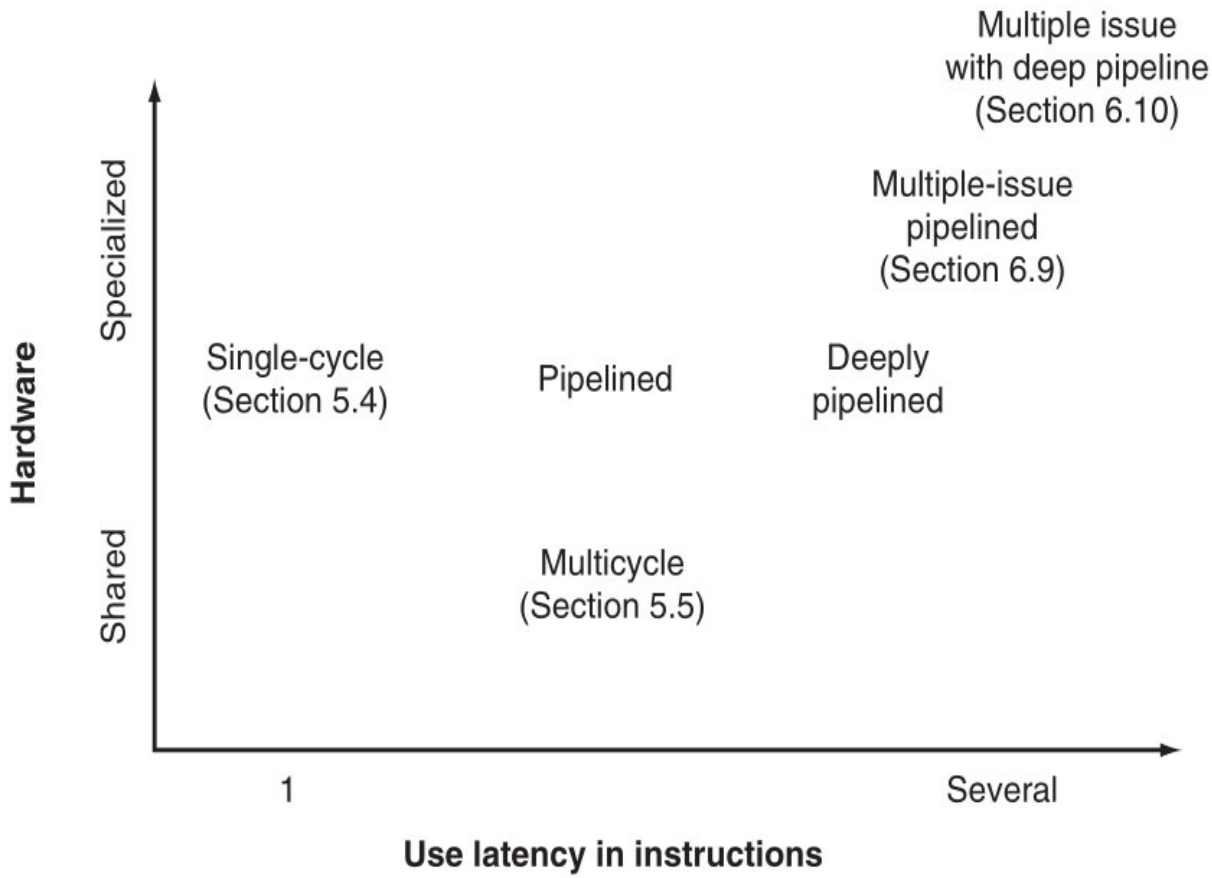


FIGURE 6.53 The basic relationship between the datapaths in Figure 6.52. Notice that the *x*-axis is use latency in instructions, which is what determines the ease of keeping the pipeline full. The pipelined datapath is shown as multiple clock cycles for instruction latency because the execution time of an instruction is not shorter; it's the instruction throughput that is improved.

*小结

流水线和多发射技术企图来提高指令级的并行度，但是数据和控制的相关性限制了这种并行度的挖掘。从硬件和软件上，使用调度或推测技术可降低相关对性能的影响

- 长流水线、多指令发射和动态调度在20世纪90年代维持了每年60%微处理器性能的增加
- 过去，性能的改善主要依靠提高时钟频率和实现复杂的超标量处理器，如Pentium 4 处理器就应用了上述2 点，达到了瞩目的高性能
- 今天，采用多核技术（multi-core processor）开发更粗粒度的并行性