	ALU或branch指令	数据传输指令	时钟周期
Loop:		ld x31, 0(x20)	1
	addi x20, x20, -8		2
	add x31, x31, x21		3
	blt x22, x20, Loop	sd x31, 8(x20)	4

图 4-67 针对 RISC-V 双发射流水线进行调度后的代码。空发射槽中是 nop 指令。需要注意的是，我们将 addi 指令调度在 sd 指令之前执行，因此需要将 sd 指令的访存地址重新加上 8

循环展开（loop unrolling）是一种专门针对循环体提高程序性能的重要编译技术。它将循环体展开多遍，从不同循环中寻找可以重叠执行的指令来挖掘更多的指令级并行性。

循环展开：一种针对数组访问循环体的提高程序性能的技术。它将循环体展开多遍，对不同循环内的指令进行统一调度。

例题 | 面向多发射流水线进行循环展开

上文所示的例子中，循环展开和指令调度互相配合，可提升处理器性能。为简化起见，假设循环间隔为 4 的倍数。

答案 | 为实现无延迟的循环操作，我们需要将循环体展开 4 遍。展开后还需要消除不必要的循环开销，循环体内将包含 ld、add 和 sd 的 4 次拷贝，再加上一条 addi 和一条 blt。图 4-68 中给出了循环展开并调度后的代码。

	ALU或分支指令	数据传输指令	时钟周期
Loop:	addi x20, x20, -32	ld x28, 0(x20)	1
		ld x29, 24(x20)	2
	add x28, x28, x21	ld x30, 16(x20)	3
	add x29, x29, x21	ld x31, 8(x20)	4
	add x30, x30, x21	sd x28, 32(x20)	5
	add x31, x31, x21	sd x29, 24(x20)	6
		sd x30, 16(x20)	7
	blt x22, x20, Loop	sd x31, 8(x20)	8

图 4-68 针对 RISC-V 静态双发射流水线进行循环展开和调度后的代码。空指令槽中的是 nop 指令。因为循环体的第一条指令是 x20 减去 32，load 指令的访存地址先使用 x20 的原值，然后是地址减 8、减 16 和减 24

在循环展开的过程中，编译器使用了额外的寄存器（x28、x29 和 x30），这样的过程称为**寄存器重命名**（register renaming）。寄存器重命名的目标是，除了真数据相关，消除指令间存在的其他数据相关。这些数据相关将会导致潜在的冒险，或者妨碍编译器进行灵活的代码调度。如果只使用 x31，考虑展开后的代码将会如何：ld x31, 0(x20), add x31, x31, x21, 之后跟着 sd x31, 8(x20)，这样的指令序列不断重复，除了都使用 x31，这些指令实际上是相互独立的。也就是说，不同循环的指令之间是没有数据依赖的。这种情况称为**反相关**（antidependence）或**名字相关**（name

寄存器重命名：编译器或硬件对寄存器进行重命名，消除指令序列中的反相关。

反相关：也称为名字相关，由于名字复用（典型的就是寄存器）被迫导致的顺序排列。这并不是一种指令间真实的数据相关。

dependence)，是一种由于名称复用而被迫导致的顺序排列，并不是真正的数据相关（即真相关）。

在循环展开时对寄存器进行重命名，可以允许编译器移动不同循环中的指令，以更好地调度代码。重命名的过程可以消除名字相关，但不能消除真相关。

注意，此时循环体中的14条指令中12条指令是可以成对执行的，4遍循环花费了8个时钟周期，因此IPC为14/8=1.75。对于4遍循环，之前需要20个时钟周期，现在只需要8个时钟周期。循环展开和指令调度配合，流水线性能提高2倍以上。这些性能提升一部分来自于循环控制语句的减少，一部分来自于双发射执行。代价是使用了4个而非1个临时寄存器，同时单个循环中的代码长度也增长了一倍以上。

4.10.3　动态多发射处理器

动态多发射处理器也称为**超标量**处理器或朴素的超标量处理器。在最简单的超标量处理器中，指令按序发射，由硬件来判断当前周期可以发射的指令数：一条还是更多，或者停顿发射。显然，如果想让这样的处理器获得更好的性能，仍然需要编译器进行指令调度，消除掉指令间的相关，提高指令的发射率。不过，即使编译器配合进行了指令调度，在这个简单的超标量处理器和超长指令字处理器之间仍然存在一个重要的差别，即不论软件调度与否，硬件必须保证代码运行的正确性。此外，编译生成代码的运行正确性应该与发射率或处理器的流水线结构无关。但是，在一些超长指令字处理器中，情况却不一样。代码需要重新编译才能正确运行在不同处理器实现上。还有一些静态多发射处理器，虽然代码在不同的处理器实现上应该能运行正确，但实际情况经常会比较糟糕，仍然可能需要编译器的支持。

超标量：一种高级流水线技术，指处理器能够在动态执行时选择指令，并在一个周期内执行一条以上的指令。

许多超标量处理器扩展了动态发射逻辑的基础框架，形成了**动态流水线调度**技术。动态流水线调度技术由硬件逻辑选择当前周期内执行的指令，并尽量避免流水线的冒险和停顿。我们使用一个简单的例子来说明它是如何避免数据冒险的。请考虑下面的代码序列：

动态流水线调度：指一种为避免停顿流水线，对指令执行顺序进行重排的硬件技术。

```
ld    x31, 0(x21)
add   x1,  x31, x2
sub   x23, x23, x3
andi  x5,  x23, 20
```

即使sub指令已经可以执行，它也必须等待ld和add指令先完成。其中ld指令需要访存，可能会花费大量的时间（第5章中会解释缓存失效，这是存储访问有时候会变慢的重要原因）。采用动态流水线调度技术可以部分或者完全避免这样的数据冒险。

动态调度流水线

动态调度流水线由硬件选择后续执行的指令，并对指令进行重排来避免流水线的停顿。在这样的处理器中，流水线被分成三个主要部分：取指和发射单元、多功能部件（在2015年的高端处理器设计中，功能部件的数量达到十几个甚至更多）以及**提交单元**。图4-69中给出了流水线模型。取指和发射单元负责取指令、译码、将各指令发送到相应的功能单元上执行。每一个功能单元前都有若

提交单元：动态调度或乱序执行的流水线中判定指令何时提交的功能单元。指令一旦被提交，将会更新程序员可见的寄存器和存储器。

干缓冲区，称为**保留站**。保留站中存放指令的操作和所需的操作数。（在下一节中，我们将讨论保留站的另一种替代选择，这种方式被许多当今主流处理器使用。）只要缓冲区中指令所需操作数准备好，并且功能单元就绪，就可以执行指令。一旦指令执行结束，结果将被传送给保留站中正在等待使用该结果的指令，同时也传送到提交单元中进行保存。提交单元中保存了已完成指令的执行结果，并在指令真正提交时才使用它们更新寄存器或者写入内存。这些位于提交单元的缓冲区，通常被称为**重排序缓冲**。和静态调度流水线中的前递逻辑一样，重排序缓冲也可以用来为其他指令提供操作数。一旦指令提交，寄存器得到更新，就和正常流水线一样直接从寄存器获取最新的数据。

保留站：功能部件前的缓冲区，用来存放指令的操作和所需操作数。

重排序缓冲：动态调度处理器中用来保存指令执行结果的缓冲区。一旦指令确认将被提交，将会把缓冲区中的结果写入内存或者寄存器中。

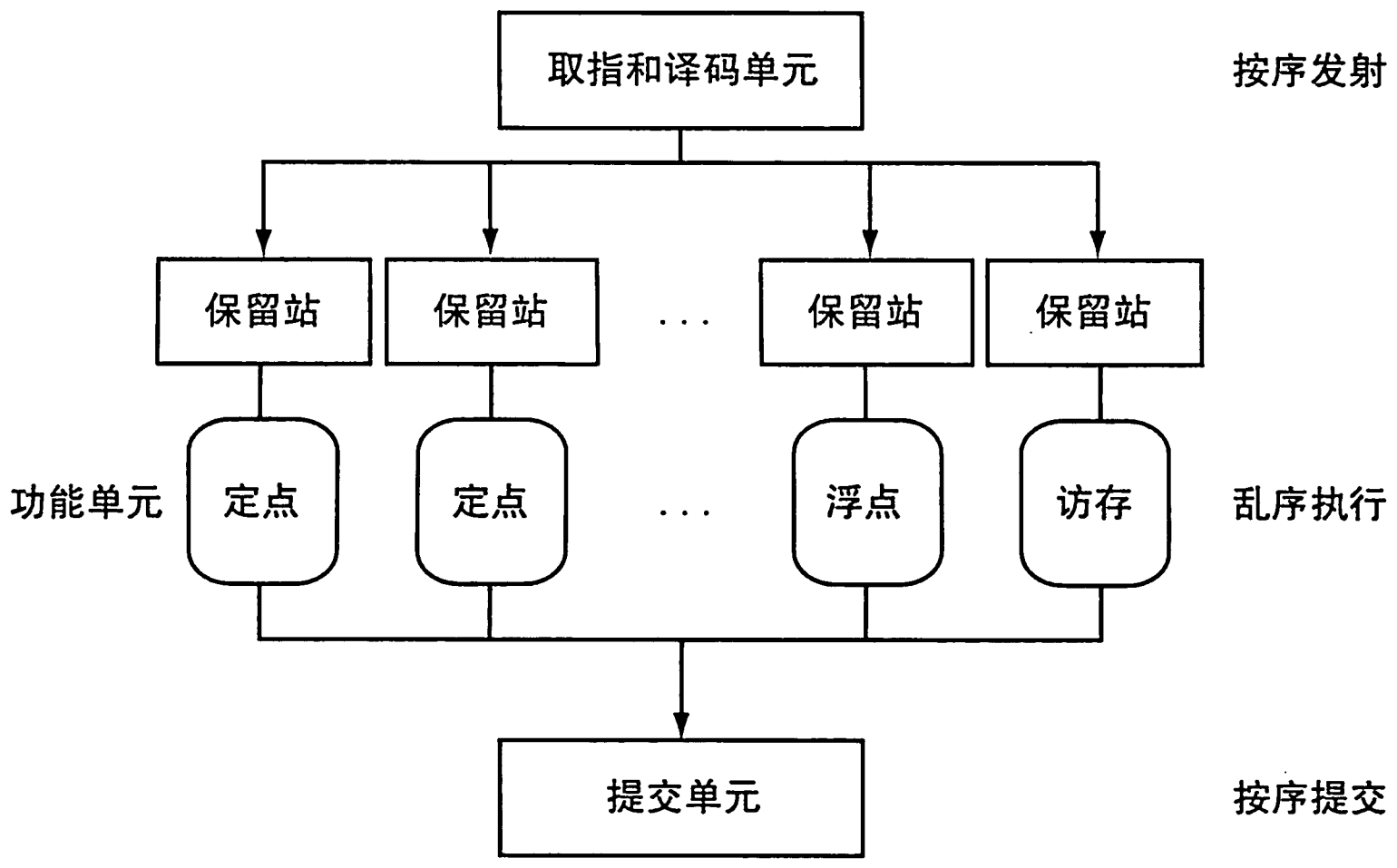

图 4-69 动态调度流水线的三个主要部分。最后一步更新处理器状态的阶段也被称为提交或者完成

在保留站中保存操作数，以及在重排序缓冲中保存运算结果，两者共同提供了一种寄存器重命名方式，有点类似之前的循环展开例子中编译器使用的技术。要了解这个概念是如何工作的，考虑以下步骤：

1. 发射指令时，指令会被拷贝到相应功能单元的保留站中。同时，如果指令所需的操作数已准备好，也会从寄存器堆或者重排序缓冲中拷贝到保留站中。指令会一直保存在保留站中，直到所需的操作数全部准备好，并且相应功能部件可用。对于处在发射阶段的指令，由于可用操作数已被拷贝至保留站中，它们在寄存器堆中的副本就无须保存了，如果出现相应寄存器的写操作，那么该寄存器中的数值将被更新。

2. 如果操作数不在寄存器堆或者重排序缓冲中，那它一定在等待某个功能单元的计算结果。该功能单元的名字将被记录。当最终结果计算完毕，将会直接从功能单元拷贝到等待该结果的保留站中，旁路了寄存器堆。

这些步骤充分利用了重排序缓冲和保留站来实现寄存器重命名。

从概念上讲，可以把动态调度流水线看作程序的数据流结构分析。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指令，被称为**乱序执行**。这是因为这样执行的指令顺序和取指的顺序是不同的。

乱序执行：流水线处理器执行过程中的一种情况，即如果当前执行的指令停顿，并不会引起后续无关指令的等待。

为使得程序行为与简单的按序单发射流水线一致，乱序执行流水线的取指和译码都需要按序进行，以便正确处理指令间的相关。同样，提交阶段也需要按照取指的顺序依次将指令执行的结果写入寄存器和存储中。这种保守的处理方法称为**按序提交**。如果发生例外，处理器很容易就能找到例外前的最后一条指令，也会保证只更新在此之前的指令需要改写的寄存器。虽然流水线的前端（取指和译码阶段）和后端（提交阶段）都是按序执行，但是功能部件是允许乱序执行的。任何时候只要所需数据准备好，指令就可以被发射到功能部件上开始执行。目前，所有动态调度的流水线都是按序提交的。

按序提交：流水线处理器的一种提交方式。指的是按照取指的顺序更新程序员可见的处理器状态。

更为高级的动态调度技术还包括基于硬件的推测式执行，特别是基于分支预测。通过预测分支指令的转移方向，动态调度处理器能够沿着预测路径不间断地取指和执行指令。由于指令是按序提交的，在预测路径上的指令提交之前就已经知道分支指令是否预测成功。支持推测执行的动态调度流水线还可以支持 load 指令访存地址的推测。这将允许乱序执行 load-store 指令，并使用提交单元来避免不正确的推测。在下一节中，我们将介绍在 Intel Core i7 设计中的动态调度和推测式执行技术。

理解程序性能　既然编译器也能进行指令调度来解决数据相关，读者可能会问为什么超标量处理器还需要使用动态调度技术。这主要有三个原因。首先，不是所有的流水线停顿都是可预测的。特别是，存储层次中的缓存失效就能引起流水线中不可预测的停顿（具体见第 5 章）。动态调度允许处理器通过执行其他指令来隐藏这些停顿。

其次，如果处理器中使用动态分支预测技术来推测分支指令的执行结果，我们在编译程序时是无法知道指令的真实执行顺序的，这依赖于分支指令的预测结果和执行结果。仅仅使用推测式执行技术去挖掘程序的指令级并行性，而不与动态调度相结合，这显然会影响推测式执行的效果。

最后，不同的流水线实现具有不同的延迟和发射宽度，这会改变编译代码的最佳配置。例如，对具有相关的指令序列如何进行调度受到流水线的发射宽度和延迟的双重影响。流水线结构也会影响为了避免停顿而展开的循环体遍数，也就影响了基于编译器的寄存器重命名过程。而动态调度技术可以隐藏以上大多数硬件细节。因此，用户和软件发行商无须担心为相同指令系统的不同实现维护多个程序版本。类似的，之前的旧代码也可以不用重新编译就运行在新的硬件实现上，从中获得更多的好处。

重点　流水线和多发射技术尝试挖掘程序的指令级并行性，提高了指令执行的峰值吞吐率。但是，由于处理器总是需要等待冒险的解决，因此程序中的数据和控制相关限制了性能的可达上限。以软件为中心的指令级并行开发技术依靠编译器来寻找这些依赖关系，并减少它们带来的不良影响。而以硬件为中心的指令级并行开发技术依赖于流水线结构和指令发射机制的扩展。不管是基于编译器还是硬件，推测式执行都能通过预测提高指令级并行度。不过，由于推测错误很可能会降低性能，使用时需要小心。

硬件/软件接口　现代高性能处理器能够单周期发射多条指令，但是一直保持高发射率是困难的。例如，尽管存在四发射或者六发射的处理器，但很少有应用可以一直保持两条以上的发射率。这主要是由以下两个原因造成的。

首先，在流水线内部，主要的性能瓶颈在于指令之间的依赖关系。这种依赖关系无法消除，降低了指令间的并行性，也降低了流水线的发射率。对于真正的数据相关，我们确实无能为力。但是，有时候却是由于编译器或者硬件的能力有限，并不能准确知道这种数据相关

是否存在，因此不得不先保守地假设指令序列中存在真数据相关。例如，程序的代码中常使用指针，这种数据结构特别容易产生存储器别名，这会导致潜在的数据相关。相反，对于数组访问，由于有更强的规律性，编译器可以直接判断出指令之间不存在依赖关系。同样，对于分支指令来说，那些无法在运行或者编译时准确预测出跳转方向的分支指令，将会对深入挖掘流水线中的指令级并行能力产生不良影响。通常，指令级并行是有提升空间的，但由于那些影响性能的因素分布非常广泛（有时是在成千上万条指令的执行中），编译器和硬件就显得能力不足了。

其次，存储层次（详见第 5 章）中的各级失效会使得流水线不能满负荷运转。虽然通过流水线的指令调度可以隐藏某些存储系统的延迟，但是，程序中有限的指令级并行会限制可被调度的指令数量，从而使得隐藏延迟的能力受限。

4.10.4 高级流水线和能效

通过动态多发射和推测式执行深度挖掘指令级并行能力也会带来负面影响，其中最重要的就是降低了处理器的能效。每一个技术上的创新都可能会产生新的结构，使用更多的晶体管来获取更高的性能。但是这种做法可能很低效。目前，我们已经撞上了功耗墙，因此转向设计单芯片多处理器架构，这样就无须像之前那样设计更深的流水线或者采用更激进的推测机制。

我们都相信，虽然简单处理器运行速度不如复杂处理器，但是相同的性能下它们的能耗更低。因此，当结构设计受限于能量而非晶体管数量时，简单处理器能够在单芯片上获得更高的性能。

图 4-70 中给出了 Intel 系列处理器的参数比较，包括流水线级数、发射宽度、推测策略、时钟频率、单芯片上的核心数目以及功耗等。特别需要注意的是，从单核设计转向多核设计后，流水线级数和功耗都有明显的下降。

微处理器	年份	时钟频率	流水线级数	发射宽度	乱序/推测	单芯片核数	功耗
Intel 486	1989	25 MHz	5	1	否	1	5W
Intel Pentium	1993	66 MHz	5	2	否	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	是	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	是	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	是	1	103W
Intel Core	2006	2930 MHz	14	4	是	2	75W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	是	2～4	87W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	是	8	77W

图 4-70 Intel 系列处理器参数比较，包括流水线复杂度、核数和功耗。其中，奔腾 4 的流水线级数不包括提交阶段。如果把它考虑进来，奔腾 4 的流水级数将会更深

详细阐述 提交单元控制了寄存器堆和存储器的更新。一类动态调度处理器在执行期间就更新寄存器堆，使用额外的寄存器实现重命名功能，并保存寄存器旧值直到指令提交。其他处理器则将执行结果保存在上文提到的重排序缓冲中，并在提交阶段才真正更新寄存器堆。对于存储器的写操作，必须保存在重排序缓冲中，或者写入缓冲区（store buffer）中（详见第 5 章）。当缓冲区中的地址和数据都准备好，并且 store 指令不在任何推测路径上时，提交单元允许 store 指令向存储器发出写操作。

详细阐述　存储器访问还可以采用非阻塞高速缓存（nonblocking cache）。该结构支持在缓存失效时继续提供缓存访问服务（详见第5章）。乱序执行处理器需要在发生缓存失效时继续执行指令。

自我检测　说明下列挖掘指令级并行的技术或模块主要是基于硬件还是基于软件的。对某些项来说两者都有可能。

1. 分支预测　　4. 超标量　　7. 推测
2. 多发射　　5. 动态调度　　8. 重排序缓冲
3. 超长指令字　　6. 乱序执行　　9. 寄存器重命名

4.11　实例：ARM Cortex-A53 和 Intel Core i7 流水线结构

图4-71中是本节要讨论的两个微处理器，它们是后PC时代两个具有代表性的结构。

处理器	ARM A53	Intel Core i7 920
市场	个人移动设备	服务器，云计算
设计功耗	100mW（单核频率1GHz）	130W
时钟频率	1.5GHz	2.66GHz
单芯片核数	4（可配置）	4
是否有浮点?	有	有
多发射?	动态	动态
峰值IPC	2	4
流水线级数	8	14
流水线调度	静态按序	动态乱序+推测式执行
分支预测	混合	2级
单核内L1 Cache	ICache：16～64KB DCache：16～64KB	ICache：32KB DCache：32KB
单核内L2 Cache	共享，128～2048KB	每个核256KB
共享L3 Cache	由平台决定	2～8MB

图4-71　ARM Cortex-A53 和 Intel Core i7 920 的技术参数

4.11.1　ARM Cortex-A53

ARM Cortex-A53是一款基于ARMv8指令系统的八级流水线结构处理器，主频为1.5GHz。该处理器采用动态多发射技术，每周期发射两条指令；采用静态按序流水线，指令按序发射、按序执行、按序提交。流水线分为三个部分：取指、译码和执行。图4-72显示其完整的流水线结构。

流水线的前三级每周期取两条指令，尽可能保持指令队列中有足够多的指令。Cortex-A53采用6kb大小的混合条件分支预测器、256表项的间接转移预测器以及一个用来预测函数返回地址的8表项返回地址栈。间接转移预测需要占用额外的一个流水级。如果无法使用指令队列对取指阶段和译码、执行阶段解耦合的话，这样的设计选择将会带来额外的性能开销。特别是在分支预测错误或者指令缓存失效的情况下，分支预测错误使得流水线被清空，这将导致8个时钟周期的性能开销。

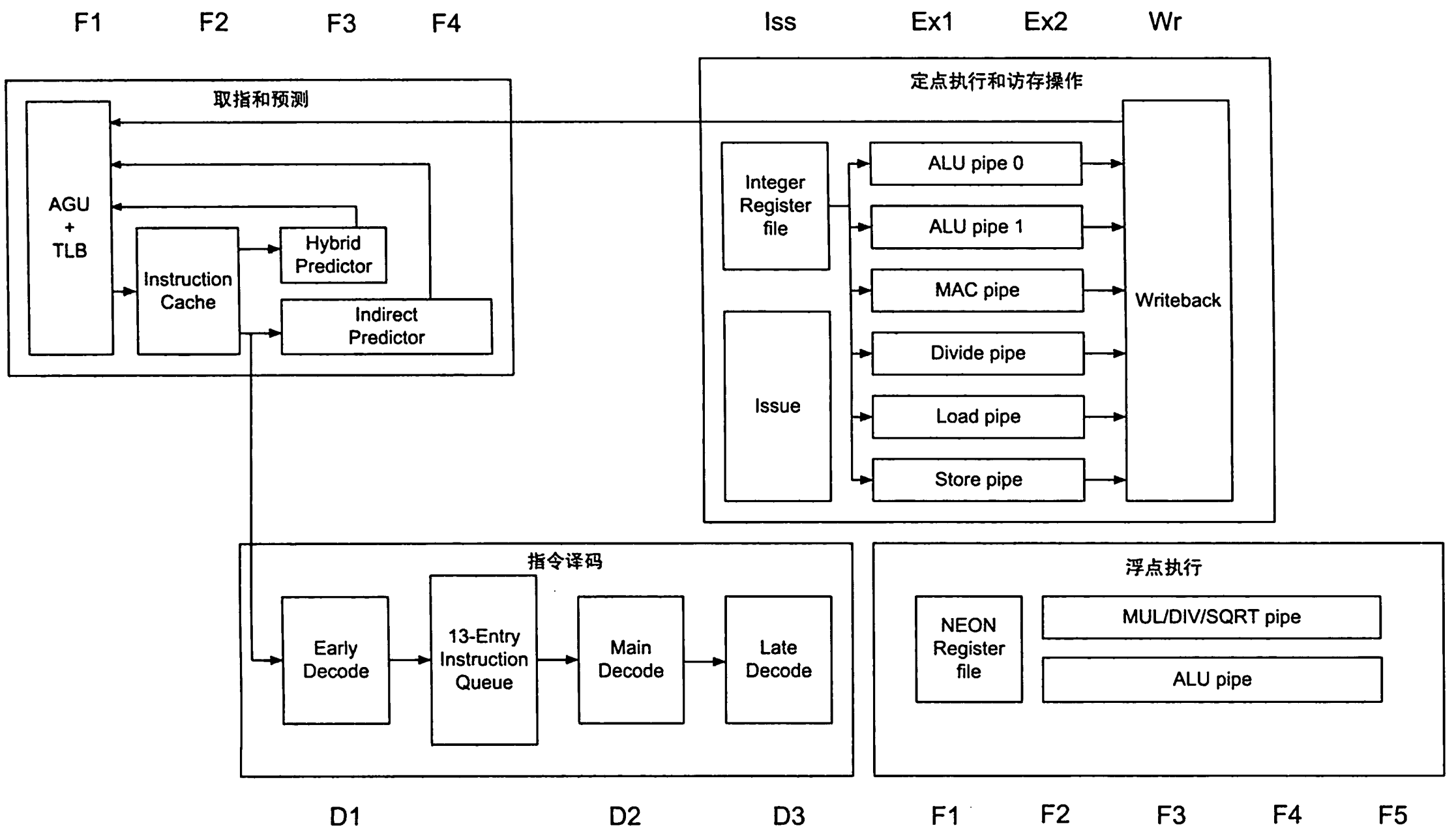

图 4-72 Cortex-A53 流水线结构。前三级流水线将指令取到表项数为 13 的指令队列中。地址产生单元（AGU）使用混合预测器、间接转移预测器和返回栈对各类分支转移指令进行预测，保持指令队列中有足够多的指令。指令译码和执行分别占用了三级流水线，浮点和 SIMD 操作则占用了五级流水线

流水线的译码阶段用来判断“指令对”内部是否存在依赖关系，并决定指令将被发送到哪类功能部件上。这种依赖关系可能导致“指令对”内部顺序执行。

指令执行阶段占用了三个流水级。Cortex-A53 为 load 和 store 指令分别提供单独的执行流水线，为定点算术指令提供两条执行流水线，为定点乘法和除法操作分别提供单独的执行流水线。“指令对”中的任意一条指令都可以发射到 load 或者 store 流水线中。执行阶段的不同流水线之间具有全旁路机制。

为实现浮点和 SIMD 指令，在指令的执行阶段增加两个流水级。使用其中一级实现浮点的乘法 / 除法 / 开方操作，另一流水级实现浮点的其他算术操作。

图 4-73 给出了 Cortex-A53 上运行 SPEC2006 测试程序的 CPI。理想 CPI 为 0.5，实际上最小 CPI 达到了 1.0，平均 CPI 为 1.3，最坏情况为 8.6。在一般情况下，60% 的停顿都是由于流水线中的冒险引起的，另外 40% 是由于存储访问造成的。流水线停顿可能是由于分支预测错误、结构冒险以及指令对之间的数据相关造成的。ARM Cortex-A53 使用静态调度的流水线结构，想要避免结构冒险和数据相关只能依靠编译器。

详细阐述 Cortex-A53 是一款支持 ARMv8 指令系统体系结构的可配置处理器核，它以 IP（Intellectual Property，知识产权）核方式交付使用。IP 核是一种用于交付的主要模式，被广泛运用在嵌入式、个人移动设备以及其他相关方向的市场上。数以十亿计的 ARM 和 MIPS 处理器就是从这些 IP 核中衍生出来的。

注意，上述的处理器 IP 核与 Intel Core i7 多核计算机上的处理器核不尽相同。处理器 IP 核（本身可能就是个多核）需要与其他逻辑协同设计（因为它是整个芯片的核心），并共同生产出一款针对特定应用进行优化的微处理器芯片。这里的其他逻辑包括特定应用处理器

（比如视频编码器或者解码器）、I/O 接口和存储控制器。虽然处理器核的逻辑结构是确定的，但是最终的芯片实现却千差万别。例如，对于 L2 Cache 的容量可以有 16 倍的差别。

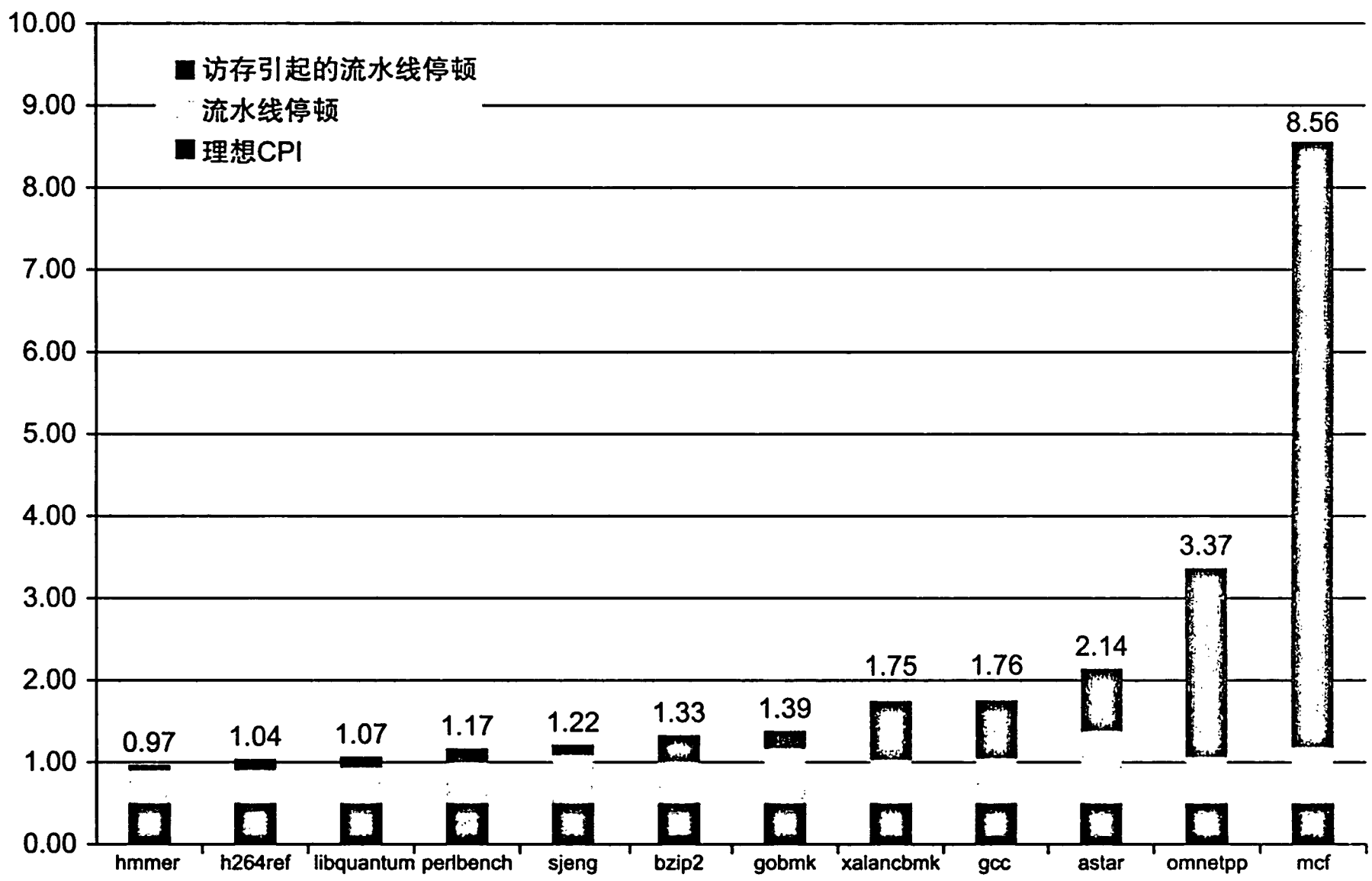

图 4-73　ARM Cortex-A53 上运行 SPEC2006 定点测试程序的 CPI

4.11.2　Intel Core i7 920

x86 微处理器采用了大量复杂的流水线技术，包括动态多发射、动态流水线调度、乱序执行以及流水线的推测式执行。但是，这些处理器仍然需要面对挑战，实现复杂的 x86 指令系统（如第 2 章中所述）。Intel 处理器按照 x86 指令格式进行取指，并将它们转换成内部的类似 RISC-V 风格的指令，Intel 称之为微操作（micro-operation）。之后，指令以微操作的形式在上述复杂流水线中执行、动态调度并推测，保持着每周期 6 条微操作以上的执行效率。本节我们将重点考虑微操作流水线。

微结构：处理器的组织结构，包括主要功能部件、它们之间的互连结构以及控制逻辑

当我们考虑设计这样的处理器时，功能单元的设计、高速缓存和寄存器堆的设计、指令发射以及整个流水线控制的设计，各种因素混合在一起，这使得数据通路很难从流水线中分离出来。因此，很多工程师和研究者采用了**微结构**（microarchitecture）这一术语来指代处理器内部详细的体系结构。

体系结构寄存器：处理器指令系统中的可见寄存器。例如，在 RISC-V 中有 32 个定点寄存器和 32 个浮点寄存器。

Intel Core i7 使用重排序缓冲结合寄存器重命名机制来解决指令中存在的反相关和推测错误。寄存器重命名技术显式地将处理器中的**体系结构寄存器**（x86 的 64 位指令系统中，体系结构寄存器数目是 16）通过换名技术映射到一个更大的物理寄存器集合上。Intel Core i7 处理器使用重命名技术来解决数据间的反相关。寄存器重命名技术需要处理器来维护体系结构寄存器与物理寄存器之间的映射关系，这种映射关系保存了每个体系结构寄存器最新映射到的物理寄存器编号。通过跟踪已经发生的重命名，寄存器重命名技术提供了另一种推测错误时的流水线恢复方法：从错误的推测路径上的第一条指令开始，撤销后续所有的寄存器映射操作。这个撤销操作将会使处理器状态恢复到指令正确执

行时的最终状态，并保持体系结构寄存器和物理寄存器之间的正确映射关系。

图 4-74 显示了 Intel Core i7 的整个组织结构和流水线结构。下面是 x86 指令在该流水线上执行的 8 个阶段。

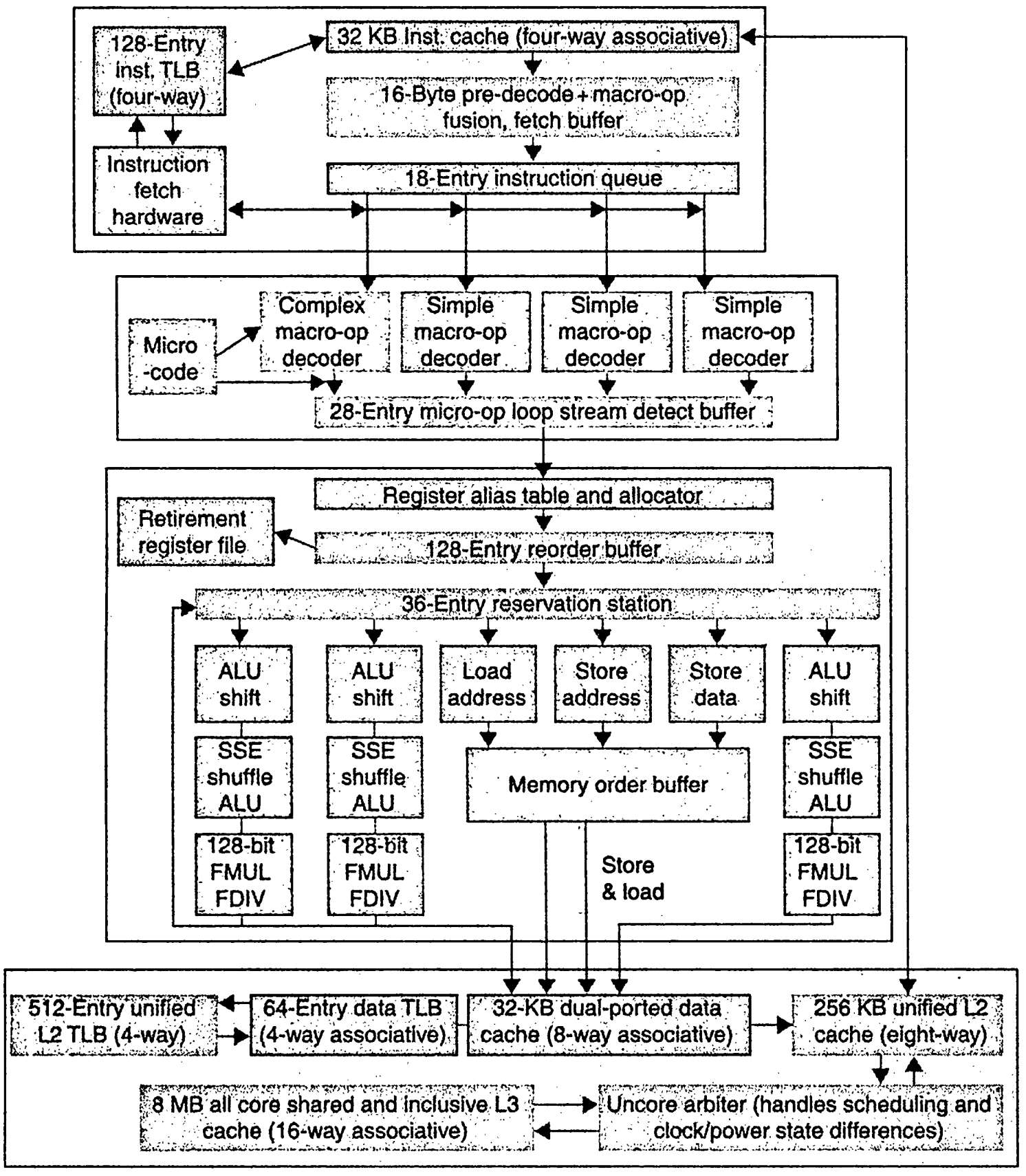

图 4-74 Intel Core i7 流水线和访存部件。整个流水线深度为 14 级，分支预测错误的开销为 17 个时钟周期。该设计能够缓存 48 条 load 指令和 32 条 store 指令，具有 6 个独立的功能单元，每周期可以执行 6 个微操作

1. 取指——处理器采用了多层次分支目标缓存结构，以此来平衡分支预测正确率和处理器频率之间的关系。还有一个返回地址栈（RAS）用来加速函数调用的返回。预测错误会引起大约 15 个时钟周期的性能开销。使用了预测产生的地址，取指单元可保证从指令缓存中每周期取出 16 字节的指令。

2. 这 16 个字节的指令将放入预译码指令缓冲中——在预译码阶段，将把这 16 字节的指令转换成单个的 x86 指令。由于 x86 指令是变长的，从 1 到 15 个字节都有可能，因此预译码阶段必须扫描多个字节以确定指令的长度。预译码后的单个 x86 指令放入 18 个表项的指令队列中。

3. 微操作译码——单条 x86 指令将被转换成微操作，Intel Core i7 使用三个译码器进行直接转换。对于语义复杂的 x86 指令，使用微码引擎产生对应的微操作序列，该引擎能连续每周期产生 4 个以上的微操作。这些微操作按照 x86 的指令顺序存放在 28 表项的微操作缓冲中。

4. 在微操作缓冲中进行循环流检测——循环流检测可以在具有循环体的动态指令序列（少于28条指令或者指令长度小于256字节）中发现循环，并直接从微操作缓冲中发射后续操作，无须启动取指和译码。

5. 进行基本指令发射——在寄存器表中查找寄存器位置，对寄存器进行重命名，为指令在重排序缓冲中分配表项，将微操作发送到保留站中，并从寄存器堆或重排序缓冲中取出操作数。

6. Intel Core i7采用36表项的集中式保留站，由6个功能部件共享。每周期将分发6个以上的微操作到不同的功能部件上。

7. 单个功能部件执行对应的微操作，并把执行结果送回到等待该数据的保留站和寄存器提交单元。一旦指令不在任何推测路径上就及时更新相关寄存器的状态，并将该指令在重排序缓冲中的相应状态标注为“完成”。

8. 当一条或多条被标注为“完成”的指令到达重排序缓冲的顶端，寄存器提交单元执行真正的写操作，指令将从重排序缓冲中移除。

详细阐述 上述第2阶段和第4阶段的硬件能够组合或者合并操作，以此来减少执行的操作数量。第2阶段中的宏操作（macro-ops）合并是对x86指令进行组合。例如，比较指令的后面跟着分支指令，可以将其变为一个操作。第4阶段的微操作合并（microfusion）是对load+ALU和ALU+store这样的“微操作对”进行组合，并发射到一个保留站中（在这里它们可以被独立发射），以提高微操作缓冲的使用率。在对Intel Core架构中的微操作合并和宏操作合并的研究中，Bird等人[2007]发现微操作合并对性能几乎没有影响，宏操作合并对定点测试程序的性能有一定提升，但对浮点测试程序的性能几乎没有影响。

4.11.3　Intel Core i7处理器的性能

图4-75显示了Intel Core i7处理器运行SPEC2006测试程序的CPI。理想CPI为0.25，事实上最小CPI为0.44，平均CPI为0.79，最大为2.67。

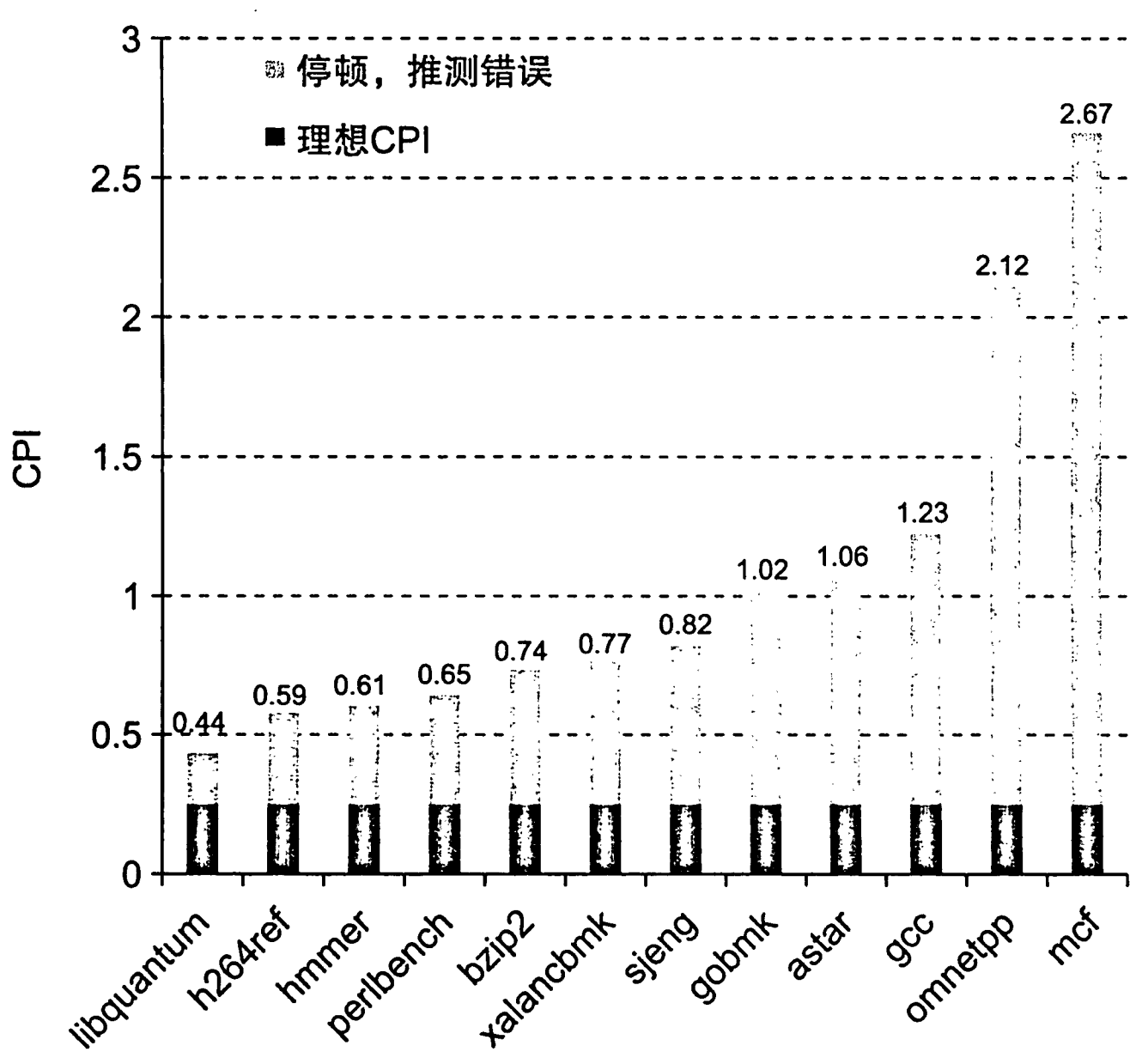

图4-75　Intel Core i7 920处理器运行SPEC2006测试程序的CPI

在一个动态乱序执行的流水线中，虽然很难区分开流水线阻塞和访存阻塞，但我们可以展示分支预测和推测执行的效果。图 4-76 显示了分支预测错误所占的百分比，以及未提交的微操作数（即执行结果被置为无效的操作）占已分发的微操作总数的百分比。分支预测错误的最小值、平均值和最大值分别为 0%、2%、10%。对于无效操作来说，最小值、平均值和最大值分别为 1%、18% 和 39%。

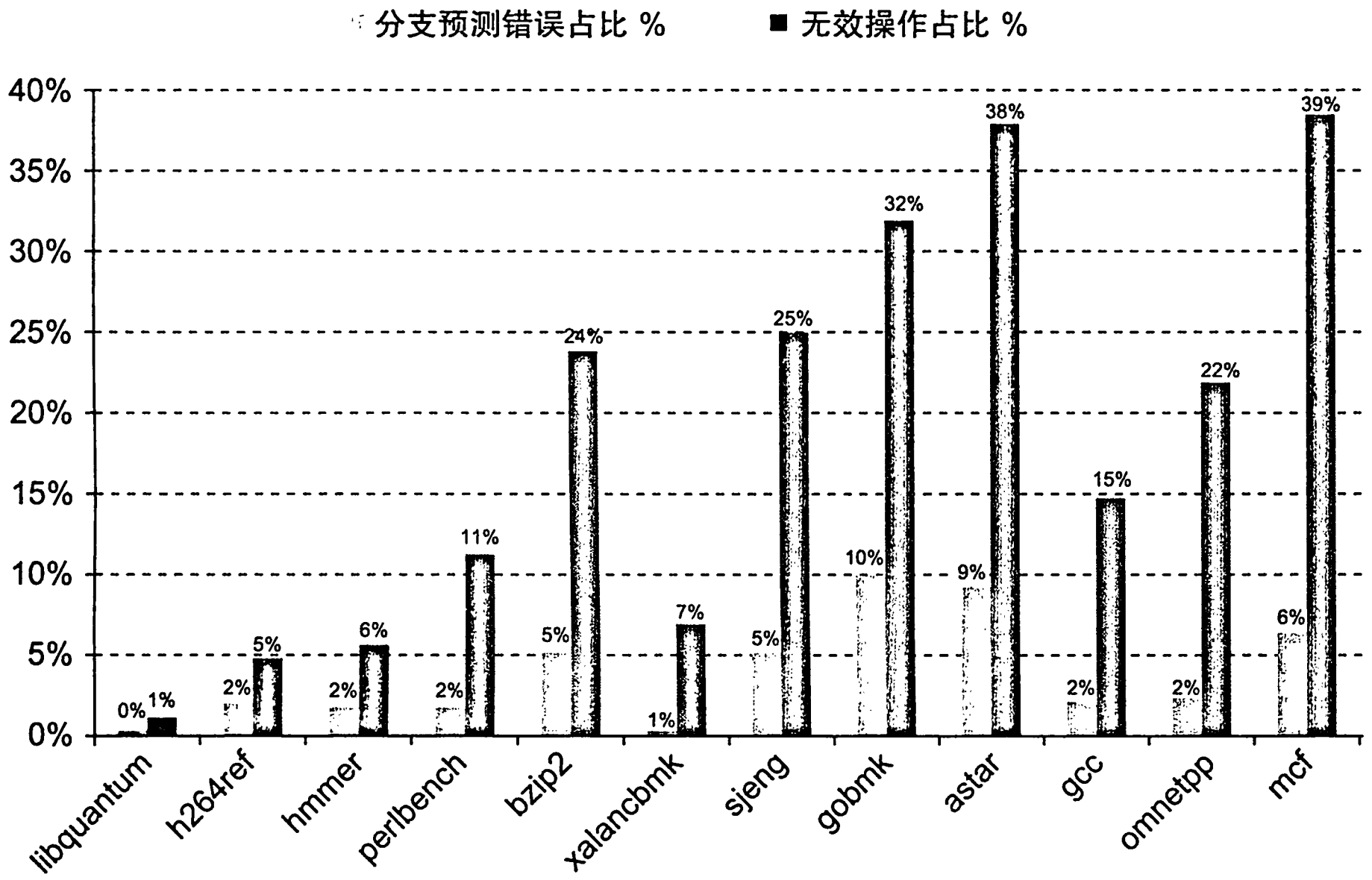

图 4-76 Intel Core i7 920 处理器运行 SPEC2006 测试程序时，分支预测错误率和无效操作比例

在某些情况下，无效操作的比例接近于分支预测错误率，比如测试程序 gobmk 和 astar。在一些实例中，例如 mcf 中，无效操作所占比例比分支预测错误率要高。区别可能就在于访存行为。由于数据缓存失效率很高，mcf 程序在推测错误和拥有充足保留站的情况下分发了大量的指令。如果分支指令最终预测错误，与此相关的所有指令的微操作都将被清除。

理解程序性能 Intel Core i7 将 14 级流水线与激进的多发射技术相结合以获取更高的性能。通过保持背靠背（back-to-back）操作的低延迟，数据相关的影响将被减小。对于运行在该款处理器上的程序来说，哪些才是潜在的影响性能的瓶颈呢？下述列表包括了一些可能的性能问题，最后三个问题会在任何高性能流水线处理器中以某种形式出现。

- 采用 x86 指令，并且不映射到简单的微操作上。
- 很难预测的分支，会引起预测错误时的流水线停顿，以及推测失败后的流水线重启。
- 长相关——典型的长延迟指令或者存储层次引起的流水线停顿。
- 存储访问延迟增大（具体见第 5 章）引起的处理器停顿。

4.12 加速：指令级并行和矩阵乘法

以第 3 章中提到的 DGEMM 为例，多发射和乱序执行处理器通过循环展开可以获得更多可供调度的指令，这体现出指令级并行技术的影响。图 4-77 给出了图 3-22 中的 C 程序进行循环展开后的版本。该版本利用 C 程序的内在特征来生成对应的 AVX 指令。

和图 4-68 中的循环展开示例一样，对以上程序循环展开四次。与之前不同，图 3-22 的

示例中手动将循环体中的每条语句复制四次，本次使用 gcc 编译器中的 -O3 编译优化选项完成循环展开。(在 C 代码中使用常数 UNROLL 来控制循环展开的次数，以便于进行不同数值的尝试。) 使用一个简单的 4 次迭代循环体 (具体见第 9、15 和 20 行)，并使用一个包含 4 个元素的数组 c[ ] 来代替原来的标量 C0 (具体见第 8、10、16 和 21 行)。

```
1  //include <x86intrin.h>
2  //define UNROLL (4)
3
4  void dgemm (int n, double* A, double* B, double* C)
5  {
6      for ( int i = 0; i < n; i+=UNROLL*4 )
7          for ( int j = 0; j < n; j++ ) {
8              __m256d c[4];
9              for ( int x = 0; x < UNROLL; x++ )
10                 c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12             for( int k = 0; k < n; k++ )
13             {
14                 __m256d b = _mm256_broadcast_sd(B+k+j*n);
15                 for (int x = 0; x < UNROLL; x++)
16                 c[x] = _mm256_add_pd(c[x],
17                     _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18             }
19
20             for ( int x = 0; x < UNROLL; x++ )
21                 _mm256_store_pd(C+i+x*4+j*n, c[x]);
22             }
23     }
```

图 4-77　DGEMM（图 3-22）优化后的 C 程序版本。使用 C 程序的内在特征产生 x86 指令系统中的 AVX 扩展指令 (子字并行)，通过循环展开产生更多的指令级并行的机会。图 4-78 是针对图 4-77 中的内部循环使用编译器生成对应的汇编程序，对其中的三个循环体进行了展开，来发现更多的指令级并行

图 4-78 中是循环展开后的汇编代码。正如所料，图 3-23 中的每条 AVX 指令都有对应的四个版本。只有一个例外，我们只需要一条 vbroadcastsd 指令，因为可以在整个循环体内部重复使用 B 元素在寄存器 %ymm0 中的 4 个拷贝。因此，图 3-23 中的 5 条 AVX 指令在图 4-78 中变成了 17 条 AVX 指令和 7 条定点指令。注意，程序中的常数和地址计算要根据循环展开情况进行相应变化。尽管循环展开了 4 次，循环体中的指令数量也仅增加了 1 倍：从 12 条变为 24 条。

图 4-79 中，相比未优化版本，DGEMM (矩阵规模为 32×32) 中使用 AVX、AVX 和循环展开得到了性能提升。循环展开使得性能增加了 1 倍，从 6.4 GFLOPS 到 14.6GFLOPS。相比图 3-21 中的未优化版本，子字并行 和 指令级并行 的优化技术带来了 8.53 倍的性能提升。

详细阐述　正如本书 3.8 节中的详细阐述，这些评测都未打开 Turbo 模式。如果像第 3 章中那样打开 Turbo 模式，可以通过时钟频率的暂时提高 (3.3 / 2.6 = 1.27) 获得性能提升。未优化的 DGEMM 将从 1.7GFLOPS 变为 2.1GFLOPS。在 3.8 节中我们提到，Turbo 模式特别适合这种情况，因为它只使用片上八个核心中的一个。

详细阐述　在图 4-78 中，第 9 行到第 17 行之间反复使用了寄存器 %ymm5，但这并不会导致流水线停顿。这是因为 Intel Core i7 流水线会对这些寄存器进行换名。

```
 1  vmovapd (%r11),%ymm4            // Load 4 elements of C into %ymm4
 2  mov    %rbx,%rax                // register %rax = %rbx
 3  xor    %ecx,%ecx                // register %ecx = 0
 4  vmovapd 0x20(%r11),%ymm3        // Load 4 elements of C into %ymm3
 5  vmovapd 0x40(%r11),%ymm2        // Load 4 elements of C into %ymm2
 6  vmovapd 0x60(%r11),%ymm1        // Load 4 elements of C into %ymm1
 7  vbroadcastsd (%rcx,%r9,1),%ymm0    // Make 4 copies of B element
 8  add    $0x8,%rcx                // register %rcx = %rcx + 8
 9  vmulpd (%rax),%ymm0,%ymm5       // Parallel mul %ymm1,4 A
10  vaddpd %ymm5,%ymm4,%ymm4        // Parallel add %ymm5, %ymm4
11  vmulpd 0x20(%rax),%ymm0,%ymm5 // Parallel mul %ymm1,4 A
12  vaddpd %ymm5,%ymm3,%ymm3        // Parallel add %ymm5, %ymm3
13  vmulpd 0x40(%rax),%ymm0,%ymm5 // Parallel mul %ymm1,4 A
14  vmulpd 0x60(%rax),%ymm0,%ymm0 // Parallel mul %ymm1,4 A
15  add    %r8,%rax                 // register %rax = %rax + %r8
16  cmp    %r10,%rcx                // compare %r8 to %rax
17  vaddpd %ymm5,%ymm2,%ymm2        // Parallel add %ymm5, %ymm2
18  vaddpd %ymm0,%ymm1,%ymm1        // Parallel add %ymm0, %ymm1
19  jne    68 <dgemm+0x68>          // branch if %r8 !=   %rax
20  add    $0x1,%esi                // register % esi = % esi + 1
21  vmovapd %ymm4,(%r11)            // Store %ymm4 into 4 C elements
22  vmovapd %ymm3,0x20(%r11)        // Store %ymm3 into 4 C elements
23  vmovapd %ymm2,0x40(%r11)        // Store %ymm2 into 4 C elements
24  vmovapd %ymm1,0x60(%r11)        // Store %ymm1 into 4 C elements
```

图 4-78　编译器产生的嵌套循环体的 x86 汇编程序，C 程序见图 4-77

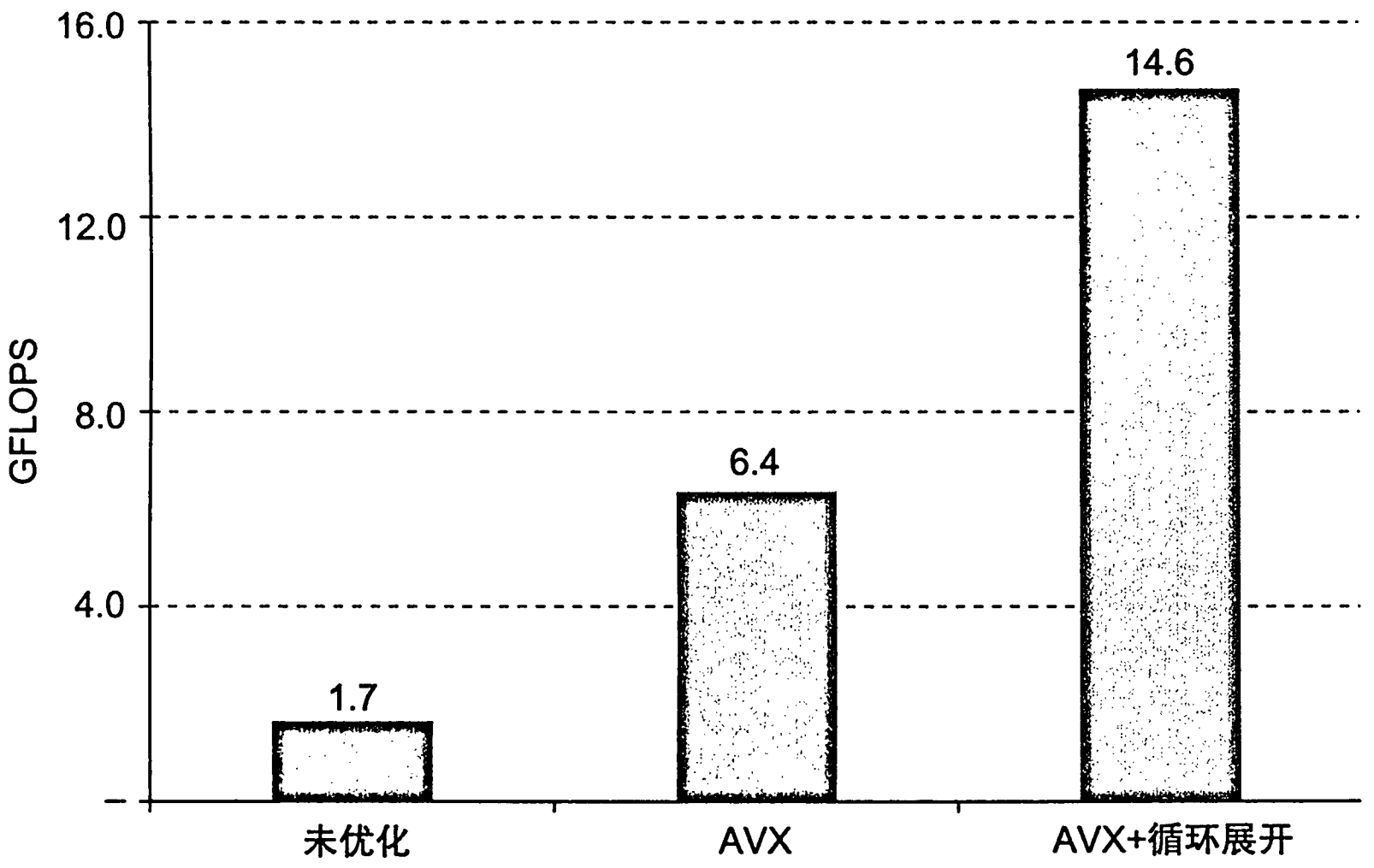

图 4-79　DGEMM（矩阵规模为 32 × 32）的性能。相比图 3-21 中的未优化版本，子字并行和指令级并行带来了几乎 9 倍的性能提升

自我检测　以下这些描述哪些正确？哪些错误？

1. Intel Core i7 处理器使用了多发射流水线，可以直接执行 x86 指令。
2. Cortex-A53 和 Core i7 都使用了动态多发射。
3. 相比 x86 架构所需，Core i7 微结构中实现了更多的寄存器。
4. 相比早期的 Intel Pentium 4 Prescott 微结构，Intel Core i7 使用了不到一半的流水线级数（见图 4-70）。

4.13　高级专题：数字设计概述——使用硬件设计语言进行流水线建模以及更多流水线示例

借助于硬件描述语言和现代计算机辅助综合工具，现代数字电路可以使用逻辑综合和标准库单元根据具体描述实现详细的硬件设计。关于这类语言和它们在数字设计中的使用，已有很多书籍说明。本节（作为网络在线内容）仅给出概述，并以 Verilog（一种硬件设计语言）为例说明如何从行为和硬件可综合两个角度来描述处理器控制逻辑。之后给出五级流水线处理器的一系列 Verilog 行为级模型。最初的模型不考虑冒险，不考虑各种前递逻辑、数据冒险和控制冒险带来的变化（在模型中高亮表示）。

之后对流水线每周期的状态变化给出一系列图示，让读者可以更详细地了解 RISC-V 指令序列在流水线上的工作细节。

4.14　谬误与陷阱

谬误：流水线是简单的。

本书用来证明正确设计流水线需要非常细致。本书的第 1 版中存在一个流水线的设计错误。虽然本书已经被 100 多位专业人士审查过，也被 18 所大学选为教材，但直到有人按照它去真正实现一个计算机时，这个错误才被发现。而在现实中，像 Intel Core i7 那样的流水线结构仅 Verilog 代码就有成千上万行，复杂性可见一斑。务必当心！

谬误：对于流水线等结构设计，可以与工艺无关。

当片上晶体管的数量和速度决定了五级流水线结构是最佳解决方案时，延迟转移（具体见 4.5.2 节的详细阐述）就是一种解决控制冒险的简单方法。随着流水线级数的加深、超标量执行以及动态分支预测技术的发展，延迟转移技术就变得有些多余了。在 20 世纪 90 年代早期，动态流水线调度占用越来越多的资源，却并没有获得相应的性能提升。但由于摩尔定律的影响，晶体管数量成倍增长，处理速度远超存储，多个功能部件和动态流水线技术也就越来越关键。目前，对功耗的密切关注将使得结构设计不会过于激进，更注重能效性。

陷阱：缺乏对指令系统设计的考虑反过来会影响流水线的实现。

许多流水线设计的困难是由指令系统的复杂性引起的。以下举例说明：

- 可变的指令长度和不确定的执行时间会造成流水线各级不均衡，从而使得设计中的冒险检测逻辑变得特别复杂。最初在 20 世纪 80 年代的 DEC VAX 8500 机器中使用微操作和微流水线技术解决了上述问题。今天该技术也被运用到 Intel Core i7 中。当然，在微操作和真正的指令之间还需要进行转换并维护对应关系，这必然引入一定的开销。
- 复杂的寻址模式会引起不同类型的问题。更新寄存器的寻址模式会让冒险检测更加复

杂。还有一些寻址模式需要进行多次内存访问，这会让流水线控制更加复杂，并使流水线难以保持不间断的流动。

- 也许最佳例子就是 DEC Alpha 和 DEC NVAX。如果使用相似的工艺节点，Alpha 的最新指令系统体系结构可以实现性能超过 NVAX 的两倍。另一个例子是，Bhandarkar 和 Clark[1991] 对 MIPS M/2000 和 DEC VAX 8700 进行了比较。他们针对运行 SPEC 基准程序的时钟周期数进行统计，结论是，虽然 MIPS M/2000 执行了更多的指令，但在 VAX 上运行的时钟周期数却是 MIPS 的 2.7 倍，当然 MIPS 更快。

4.15 本章小结

明智的百分之九十，在于明智得及时。
美国谚语

正如我们在本章所见，处理器的数据通路和控制逻辑设计都是从分析指令系统和了解工艺技术的基本特性开始的。在 4.3 节中，我们学习了如何针对设计目标（实现一个单周期处理器）基于指令系统构建 RISC-V 处理器的数据通路。当然，底层的工艺技术也会影响设计决策，例如数据通路中使用何种功能部件、单周期实现是否有意义等。

指令延迟：指令固有的执行时间。

流水线技术改善的是吞吐率，而不是指令固有的执行时间，也称为**指令延迟**。对于一些指令，其延迟数与单周期设计类似。多发射机制添加额外的数据通路硬件，每周期发射出多条指令，但这会增加有效延迟。流水线技术的提出可以提高单周期数据通路的时钟频率，相应的，多发射技术则聚焦于降低 CPI（每条指令执行周期数）。

流水线技术和多发射机制都在尝试挖掘指令间的并行程度。程序中的数据和控制相关之后会变成各种冒险，它们的存在是开发更高指令级并行的主要限制因素。借助于硬件和软件，通过预测来调度和推测执行指令，是降低冒险造成的性能影响的主要技术。

在本章中，对 DGEMM 进行循环展开四遍来挖掘更多的指令，利用 Core i7 处理器的乱序执行机制可以提升一倍以上的性能。

20 世纪 90 年代，更长的流水线、多发射和动态调度技术有助于维持从 80 年代早期开始的每年 60% 的处理器性能提升。正如在第 1 章中提到的，这些微处理器一直保持着串行编程模型，但是最终它们撞上了“功耗墙”。因此，工业界被迫转向多处理器，以开发更粗粒度的指令级并行（具体见第 6 章）。这个趋势也引发设计者对 90 年代中期以来的一些发明中的能量 – 性能的含义进行重新评估。这导致了最近一些处理器在微体系结构上对流水线进行简化。

为了保持通过并行处理器带来的性能提升，Amdahl 定律表明系统的另一部分将会成为瓶颈。这将是下一章的主要内容：层次化存储。

4.16 历史视角和拓展阅读

本章节为在线部分，阐述第一个流水线处理器、最早超标量处理器的历史，讨论乱序执行和推测技术的发展以及同时期编译技术的重要进展。

4.17 练习

4.1 考虑如下指令：

指令：`and rd, rs1, rs2`