Lecture 26: VLIW & Superscalar

高级流水线技术

- 高性能流水线 - 指令级并行(ILP)技术
  - 超流水线
  - 多发射流水线
    - 静态多发射（VLIW处理器+编译器静态调度）
    - 动态多发射（超标量处理器+动态流水线调度）
- 静态多发射（VLIW-超长指令字）
  - 编译器静态推测完成“指令打包”和“冒险处理”
  - MIPS 2-发射流水线数据通路
  - 循环展开指令调度
  - IA-64的EPIC技术
- 动态多发射
  - 动态多发射流水线的通用模型
  - 动态多发射流水线的执行模式
    - 按序发射、按序完成
    - 按序发射、无序完成
    - 无序发射、无序完成
  - Pentium 4 动态多发射流水线
    - 超流水、超标量、动态调度、无序发射、无序完成

提高性能措施—实现指令级并行

- 流水线实现了指令流内部的并行，这种并行称为指令级并行（ILP）
- 有两种指令级并行策略
  - 超流水线（Super- pipelining）
    - 级数更多的流水线　　CPI=？　　CPI=1
    - 理想情况下，流水线的加速比与流水段的数目成正比

    （即：理想情况下，流水段越多，时钟周期越短，指令吞吐率越高）

    但是，它是有极限的！可以怎样突破极限呢？
  - 多发射流水线（Multiple issue pipelining ）
    - 多条指令(如整数运算、浮点运算、装入/存储等) 同时启动并独立运行
    - 前提：有多个执行部件。如：定点、浮点、乘/除、取数/存数部件等
    - 结果：能达到小于1的CPI，定义CPI的倒数为IPC

    （例如：四路多发射流水线的理想IPC为4）
    - 需要实现以下两个主要任务
      - 指令打包：分析每个周期发射多少条？哪些指令可以同时发射？
      - 冒险处理：由编译器静态调整指令或在运行时由硬件处理
    - 实现上述两个主要任务的基础—推测技术
    - 两种实现方法
      - 静态多发射：由编译器在编译时静态完成指令打包或冒险处理
      - 动态多发射：由硬件在执行时动态完成指令打包或冒险处理

实现多发射技术的基础—推测

- 推测技术：由编译器或处理器猜测指令执行结果，并以此来调整指令执行顺序，使指令的执行能达到最大可能的并行
  - 指令打包的决策依赖于“推测”的结果
    - 可根据指令间的相关性来进行推测
      - 与前面指令不相关的指令可以提前执行
    - 可对分支指令进行推测
      - 可提前执行分支目标处的指令
    - 预测仅是“猜测”，有可能推测错误，故需有推测错误检测和回滚机制
    - 推测错误时，会增加额外开销
  - 有“软件推测”和“硬件推测”两种
    - 软件推测：编译器通过推测来静态重排指令（一定要正确！）
    - 硬件推测：处理器在运行时通过推测来动态调度指令

BACK

静态多发射处理器

- 由编译器在编译时，进行相关性分析和静态分支预测，以静态完成“指令打包”或“冒险处理”
  - 指令打包（将同时发射的多条指令合并到一个长指令中）
    - 将一个周期内发射的多个指令看成一条多个操作的长指令，称为一个“发射包”
    - “静态多发射”最初被称为“超长指令字”（VLIW-Very Long Instruction Word），采用这种技术的处理器被称为VLIW处理器
    - 在同一个周期内发射的指令类型是受限制的

      （例如，只能是一条ALU指令/分支指令、一条Load/Store指令）
    - IA-64采用这种方法，Intel称其为EPIC（Explicitly Parallel Instruction Computer—显式并行指令计算机）
  - 冒险处理（主要是数据冒险和控制冒险）
    - 做法1：完全由编译器通过代码调度和插入nop指令来消除所有冒险，无需硬件实现冒险检测和流水线阻塞
    - 做法2：由编译器通过静态分支预测和代码调度来消除同时发射指令间内部依赖，由硬件检测数据冒险并进行流水线阻塞

      即：保证打包指令内部不会出现冒险！

静态多发射处理器实例

- 实例：**MIPS ISA** 指令集的静态多发射----2发射处理器

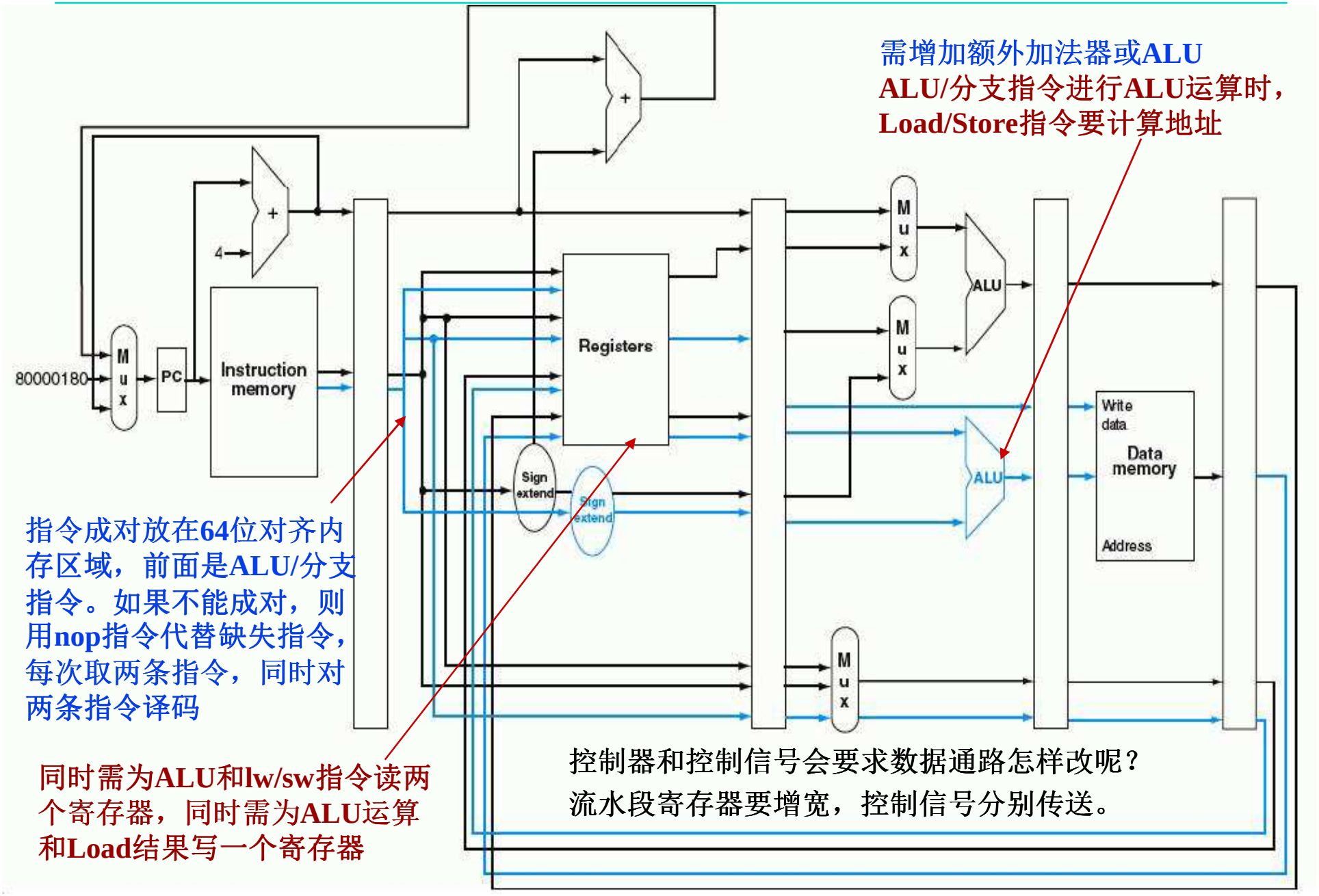
Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM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

要使原来的**MIPS**处理器能够同时处理两条流水线，数据通路需要做哪些改进？

1. 同时要能取并译码两条指令，怎么办？
   - 将两条指令打包成64位长指令，前面为ALU/Branch，后面为lw/sw
   - 没有配对指令时，就用nop指令代替
   - 将64位长指令中的两个操作码同时送到控制器（指令译码器）进行译码
2. 两条指令同时要读两个寄存器或写寄存器（和lw配对时），怎么办？
   - 增加两个读口和一个写口
3. 两条指令同时要使用ALU进行运算，怎么办？
   - 增加一个ALU（包括2组输入总线和1组输出总线）

BACK

2发射流水线数据通路（蓝色是增加部分）



2发射流水线的特点

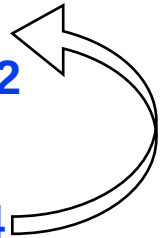
- 优点：潜在性能将提高大约2倍 （实际上达不到！为什么？ ）
- 缺点：
  - 为消除结构冒险，需增加额外部件
  - 增加潜在的由数据冒险和控制冒险导致的性能损失
    - 例1：对于Load-use数据冒险
      - 单发射流水线：只有一条指令延迟
      - 2发射流水线：有一个周期（2条指令）延迟
    - 例2：对于ALU-Load/Store数据冒险
      - 单发射流水线：可用“转发”技术使ALU结果直接转发到Load/Store指令的EXE阶段
      - 2发射流水线：两条指令同时进行，ALU的结果不能直接转发，因而不能提供给与其配对的Load/Store指令使用，只能延迟一个周期

为了更有效地利用多发射处理器的并行性，必须有更强大的编译器，能够充分消除指令间的依赖关系，使指令序列达到最大的并行性！

例：2发射MIPS指令调度

- 以下是一段循环代码段

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



前三条和后两条各具有相关性

可把第四条指令调到第一条后面

sw指令是否有问题？怎么办？

$s1减4，故sw指令偏移改为4

能否把addi和lw配成一对？

冒险！对同一个寄存器同时读，且读后要写，取决于寄存器如何设计

(能看出这段程序的功能吗？) 循环内进行的是数组访问！

- 为了能在2发射MIPS流水线中有效执行，该怎样重新排列指令
  - 调度方案如下：没有指令配对时，用nop指令代替

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

一个循环内，五条指令在四个时钟内完成，实际CPI为0.8，即: IPC=1.25

在循环中访问数组的更好的调度技术是“循环展开”

用“循环展开”技术进行指令调度

- 基本思想：展开循环体，生成多个副本，在展开的指令中统筹调度
- 上例采用“循环展开”后的指令序列是什么？
  - 为简化起见，假定循环执行次数是4的倍数
  - “循环展开”4次后循环内每条指令（lw, addu, sw，与数组访问相关）有4条再加上1条addi和1条bne, 共14条指令
  - 指令最佳调度序列如下：为何第一条指令将$s1减16？与$t0关联的指令偏移为何不同？

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi $s1,$s1,-16	lw $t0, 0($s1)	1
		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
		sw $t2, 8($s1)	7
	bne $s1,$zero,Loop	sw $t3, 4($s1)	8

14条指令用了8个时钟，CPI达到8/14=0.57。好处：充分利用并行，并消除部分循环分支！

需要用到“重命名寄存器”技术，多用了三个临时寄存器$t1,$t2,$t3，消除了名字依赖关系（非真实依赖，只是寄存器名相同而已）

代价是什么？ 多用了三个临时寄存器，并增加了代码大小（存储空间变大）

实例：Intel IA-64架构

- IA-64类似于64位MIPS架构，是Register-Register型的RISC风格指令集
- 但有独特性：要求编译器显式地给出指令级的并行性，Intel称其为EPIC

  Explicitly Parallel Instruction Computer—显式并行指令计算机
- 与MIPS-64架构的区别
  - 更多寄存器：128个整数、128个浮点数、8个专用分支、64个1位谓词
  - 支持寄存器窗口重叠技术
  - 同时发射的指令组织在指令包（bundle）中
  - 引入特殊的谓词化技术，以支持推测执行和消除分支，提高指令级并行度
- EPIC的实现技术
  - 指令组（Instruction Group）：相互间没有寄存器级数据依赖的指令序列
    - 指令组长度任意，用“停止标记”在指令组之间明显标识
    - 指令组内部的所有指令可并行执行，只要有足够硬件且无内存操作依赖
  - 指令包：同时发射的指令重新编码并形成指令包　　14+3x7+6=41
    - 长度为128，由5位长的模板字段、三个41位长的指令组成
    - 模板字段对应于以下五类功能部件中的三条指令

      整数ALU、非整数ALU（移位和多媒体）、访存、浮点、分支
  - 谓词化：将指令的执行与谓词相关联，而不是与分支指令关联

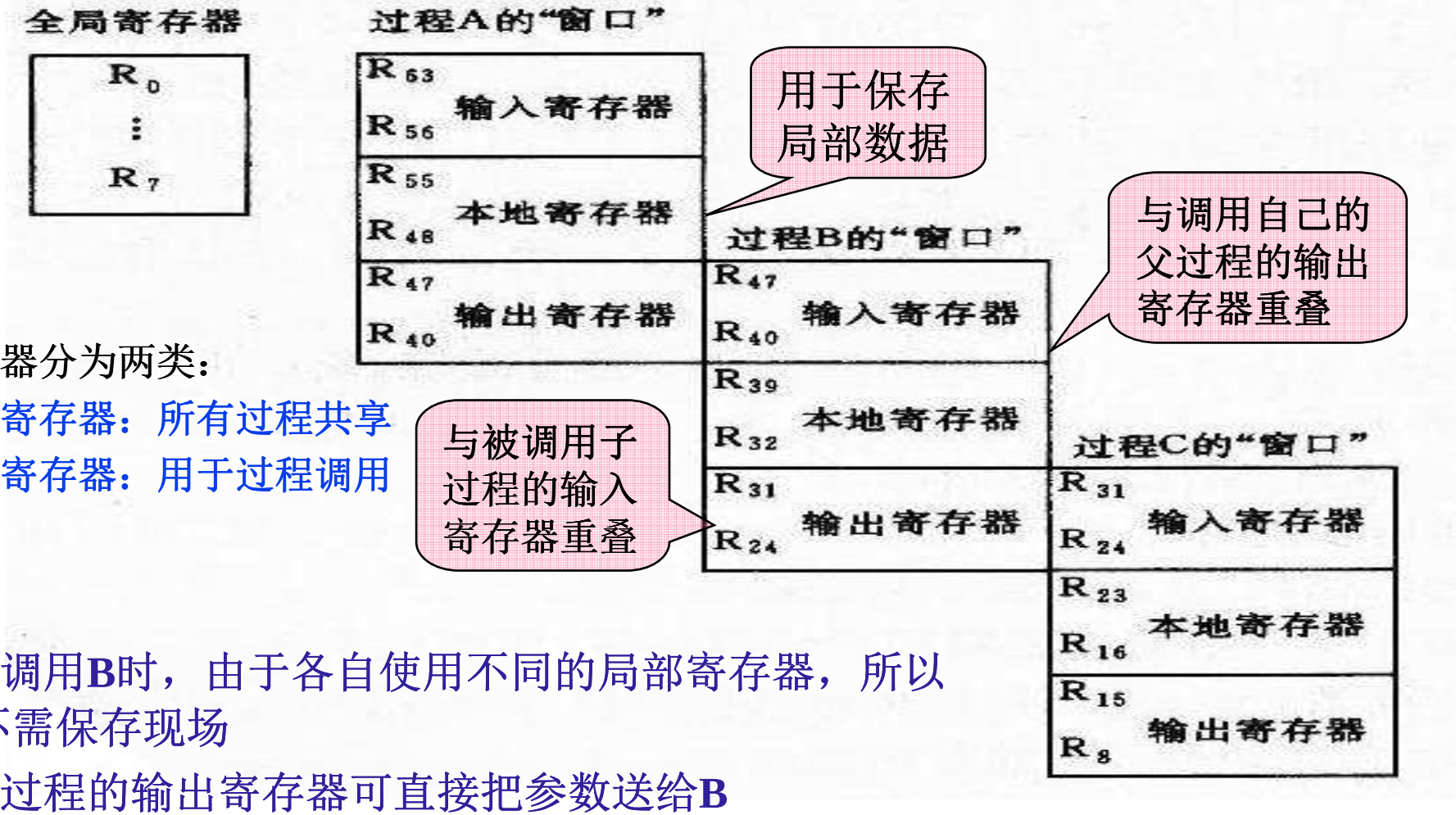
IA-64是?-发射流水线？　　3-发射流水线！　　BACK

RISC的通用寄存器

- RISC机采用大量寄存器
- 其目的：
  - 减少程序访问存储器的次数
- RISC机寄存器的组织方式有两种：
  - 重叠寄存器窗口技术ORW（硬件方法）
    - 执行过程调用和返回时，利用寄存器组而不是存储器来完成参数传递
    - 通过重叠窗口技术，使得不再需要保存和恢复寄存器内容
    - 可大大提高了程序执行的速度
  - 优化寄存器分配技术（软件方法）
    - 规定一套寄存器分配算法
    - 通过编译程序的优化处理来充分利用寄存器资源
    - 编译器为那些在一定的时间内使用最多的变量分配寄存器

BACK

重叠寄存器窗口技术（Overlapped Register Window ）



寄存器分为两类：

全局寄存器：所有过程共享

窗口寄存器：用于过程调用

A调用B时，由于各自使用不同的局部寄存器，所以不需保存现场

A过程的输出寄存器可直接把参数送给B

从B返回时，B将返回结果送到其输入寄存器，A可直接得到B返回的结果

BACK

动态多发射处理器

- 由硬件在执行时动态完成指令打包或冒险处理
- 通常被称为超标量处理器（**Superscalar**）
  - 在一个周期内执行一条以上指令
- 与**VLIW**处理器的不同点：
  - **VLIW**处理器：编译结果与机器结构密切相关，结构有差异的机器上要重新编译
  - 超标量处理器：编译器仅进行指令顺序调整，但不进行指令打包，由硬件根据机器的结构来决定一个周期发射哪几条指令。因此，编译后的代码能够被不同结构的机器正确执行
- 多数超标量处理器都结合动态流水线调度（**Dynamic pipeline scheduling**）技术
  - 通过指令相关性检测和动态分支预测等手段，投机性地不按指令顺序执行，当发生流水线阻塞时，可以到后面找指令来执行
  - 举例说明动态流水线调度技术：

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

左边指令序列中，哪条指令可以提前执行？

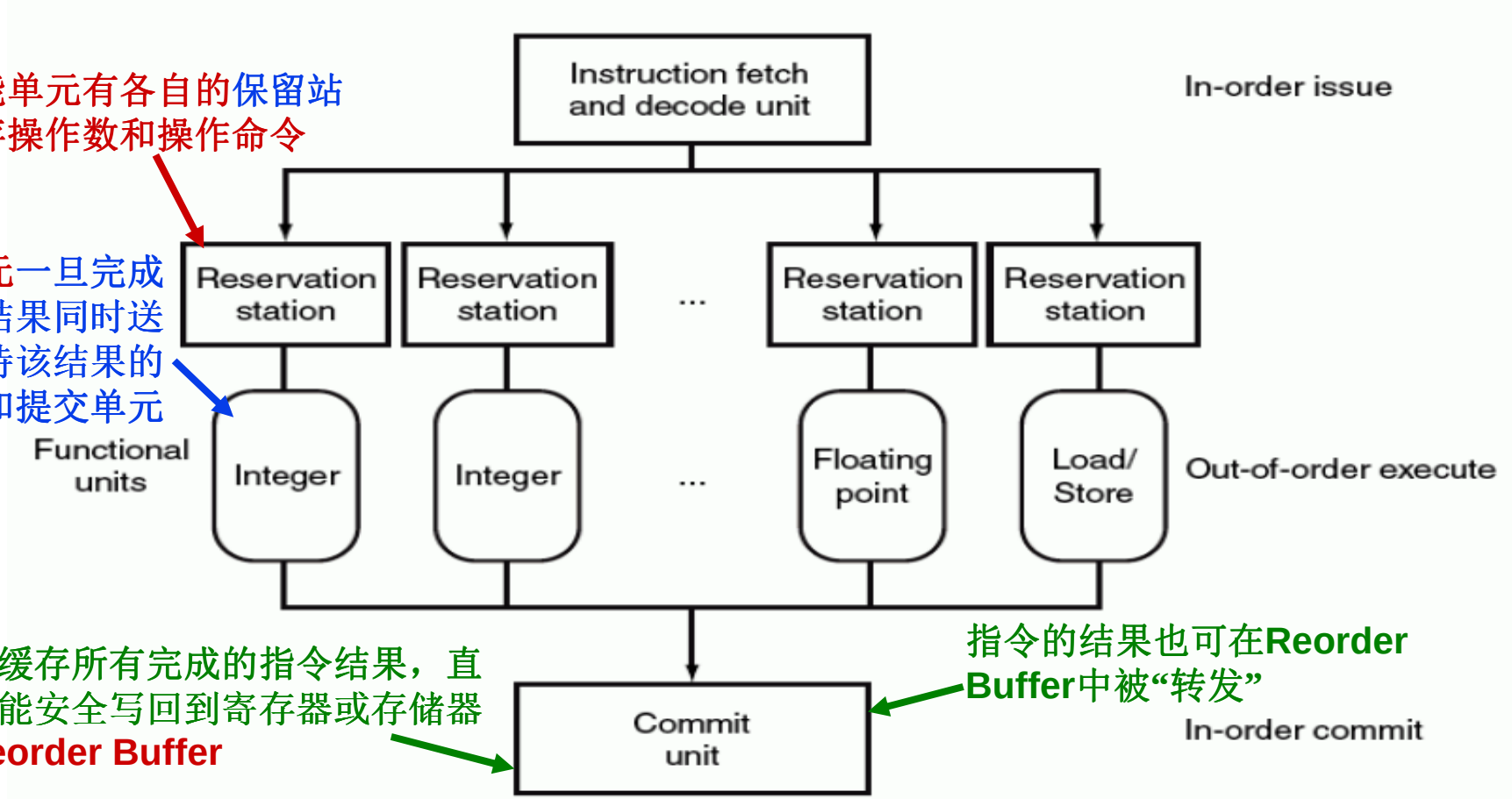
sub指令可以提前执行，不需等**lw**和**addu**指令执行完

如果不将**sub**调到前面，可能要等很长时间（**lw**指令的访存操作耗时较长！），从而影响**slti**指令的执行

最佳的方案是什么？

动态流水线调度的通用模型

- 动态流水线的一个重要的思想：在等待解决阻塞时，到后面找指令提前执行！
- 动态流水线的通用模型：
  - 一个指令预取和译码单元：有序发射
  - 多个并列执行的功能单元：乱序执行
  - 一个提交单元：有序提交



功能单元的性能

- 功能：用来执行特定类型的操作
- 性能：每个功能单元具有基本的操作性能，用两个周期数来刻画
  - 执行周期（**Latency**）：完成特定操作所花的周期数
  - 发射时间（**Issue Time**）：连续、独立操作之间的最短周期数

以下是**Pentium III** 算术功能部件的性能

Operation	Latency	Issue Time
Integer Add	1	1
Integer Multiply	4	1
Integer Divide	36	36
Floating-Point Add	3	1
Floating-Point Multiply	5	2
Floating-Point Divide	38	38
Load (Cache Hit)	3	1
Store (Cache Hit)	3	1

从上述图中看出，哪些功能部件是流水化的？哪些是非流水化的？

- 整数加、整数乘、浮点加、装入、存储这五种部件是流水化的
- 浮点乘部件是部分流水化
- 整数除和浮点除是完全没有流水化

CPU设计的一个原则：有限的芯片空间应该在各功能部件之间进行平衡！尽量让大多数资源用于最关键的操作（对大量基准程序进行评估）

从上述图中能否看出：哪些是最重要的操作？哪些是不常用的？

整数加法和乘法、浮点数加法和乘法是重要的操作

除法相对来说不太常用，而且本身难以实现流水线

动态流水线的几种执行模式

- 根据动态流水线指令发射和完成顺序，可分为三种执行模式：
  - 按序发射按序完成（Pentium）
  - 按序发射无序完成（Pentium II和Pentium III）
  - 无序发射无序完成（Pentium 4）

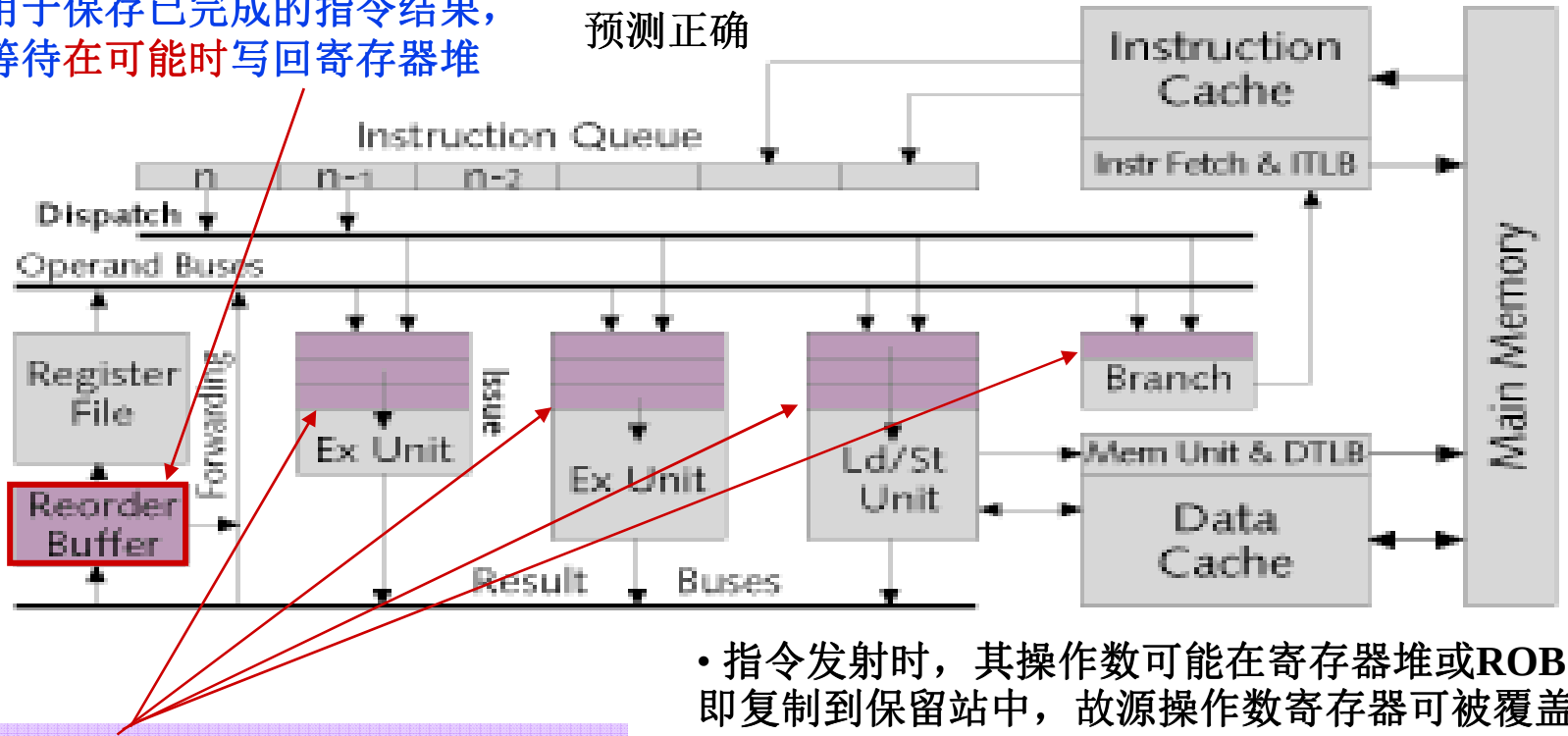
最保守的方案是顺序完成，好处：

(1) 简化异常检测和异常处理

(2) 能在被推测指令完成前得知推测结果的正确性

ReOrder Buffer 重排序缓冲:
用于保存已完成的指令结果，
等待在可能时写回寄存器堆

写回条件：与前面的所有指令结果不相关、并预测正确



保留站：存放操作数和操作命令

- 指令发射时，其操作数可能在寄存器堆或ROB中，可立即复制到保留站中，故源操作数寄存器可被覆盖
- 若操作数不在寄存器堆或ROB中，则一定在某个时刻被一个功能单元计算出来，硬件将定位该功能单元，并将结果从旁路寄存器复制到保留站

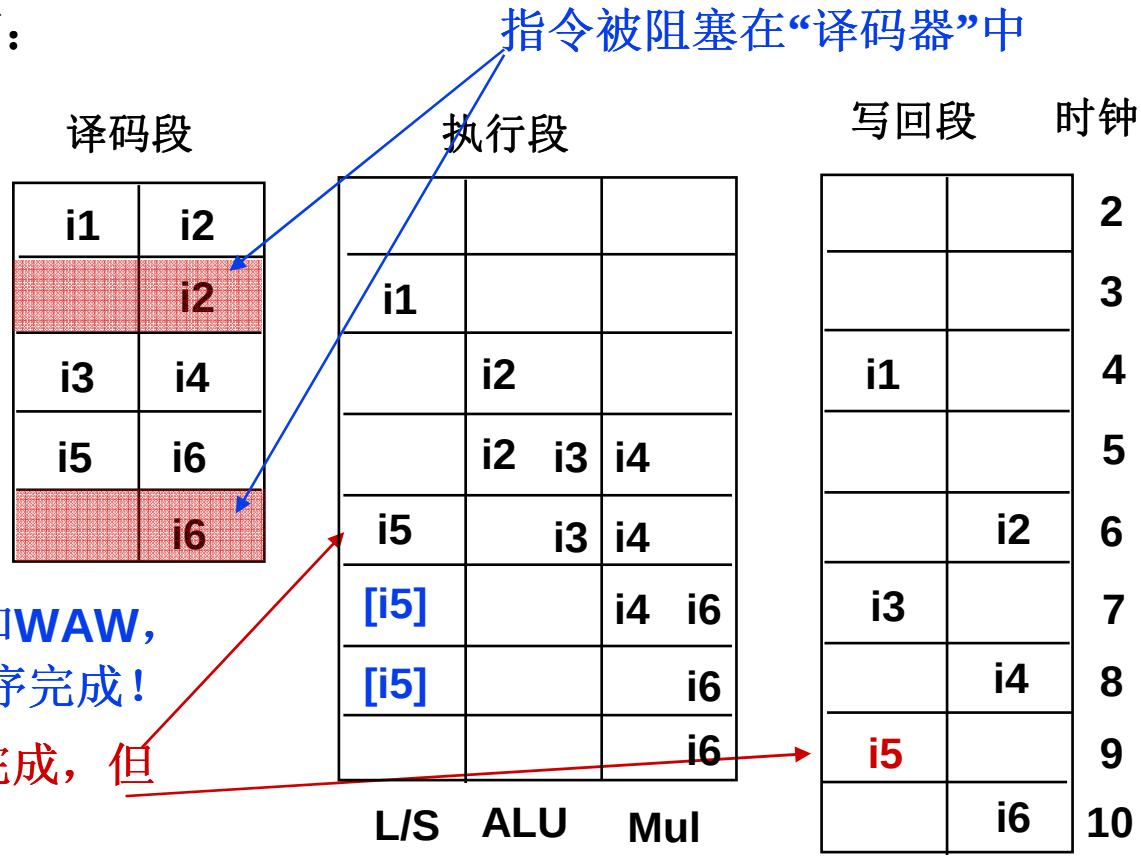
SKIP

按序发射按序完成

- 举例：2发射超标量，分为取指（F）、译码（D）、执行（E）、写回（W）。F、D、W段在一个时钟周期内完成（可同时有两条指令在这三个阶段）；E段有三个执行部件：Load/Store部件完成数据Cache访问需要1个时钟，整数ALU完成简单ALU操作需2个时钟，整数乘法器完成乘法运算需要3个时钟。执行部件采用流水化方式。

  按序发射按序完成的过程如下：

i1	lw $1, A
i2	add $2, $2, $1
i3	add $3, $3, $4
i4	mul $4, $5, $4
i5	lw $6, B
i6	mul $6, $6, $7



i1、i2间RAW; i5、i6间RAW和WAW，需阻塞一个时钟周期，并须按序完成！

为按序完成，虽i5在时钟6已完成，但一直推迟到i4写回后才写回i5

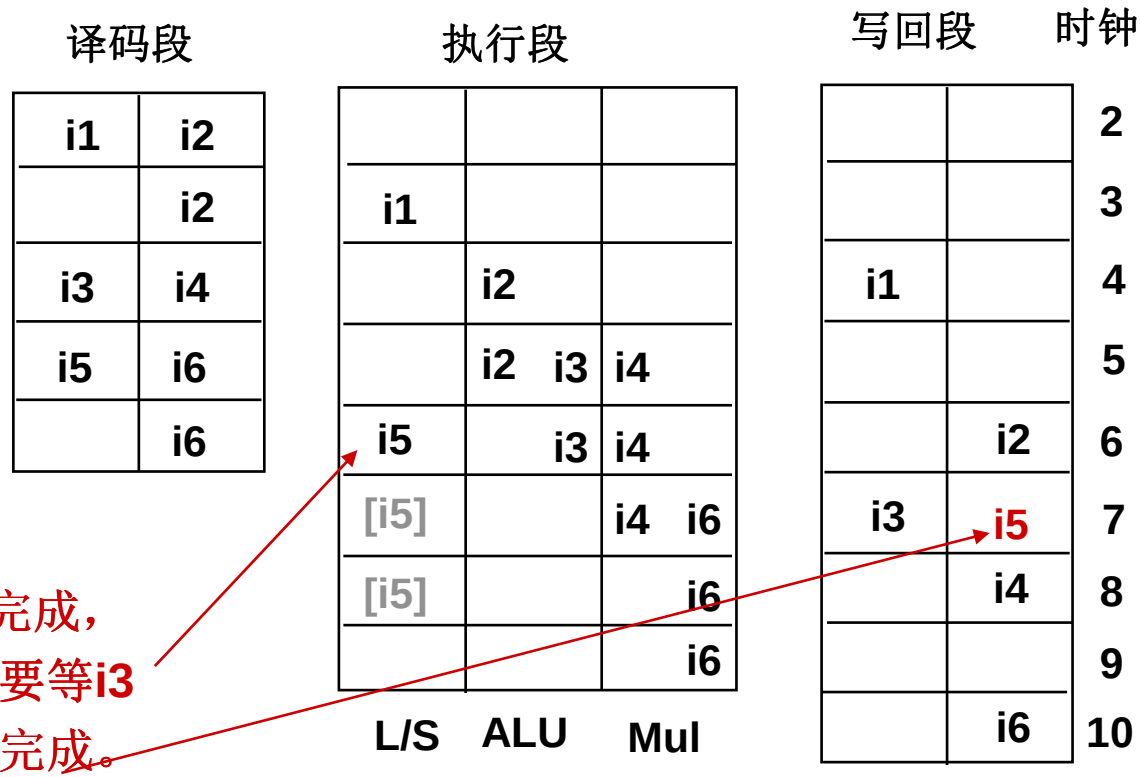
如果还有一条乘法指令，则最多可有三条乘法指令同时在执行

按序发射无序完成

- 举例：2发射超标量，分为取指（F）、译码（D）、执行（E）、写回（W）。F、D、W段在一个时钟周期内完成（可同时有两条指令在这三个阶段）；E段有三个执行部件：Load/Store部件完成数据Cache访问需要1个时钟，整数ALU完成简单ALU操作只需2个时钟，整数乘法器完成乘法运算需要3个时钟。执行部件采用流水化方式。

  按序发射无序完成的过程如下：

i1	lw $1, A
i2	add $2, $2, $1
i3	add $3, $3, $4
i4	mul $4, $5, $4
i5	lw $6, B
i6	mul $6, $6, $7



无序完成时，因为i5在时钟6已完成，和i3、i4没有相关性，可以不需要等i3、i4写回后再写回，故可先于i4完成。

BACK

无序发射无序完成

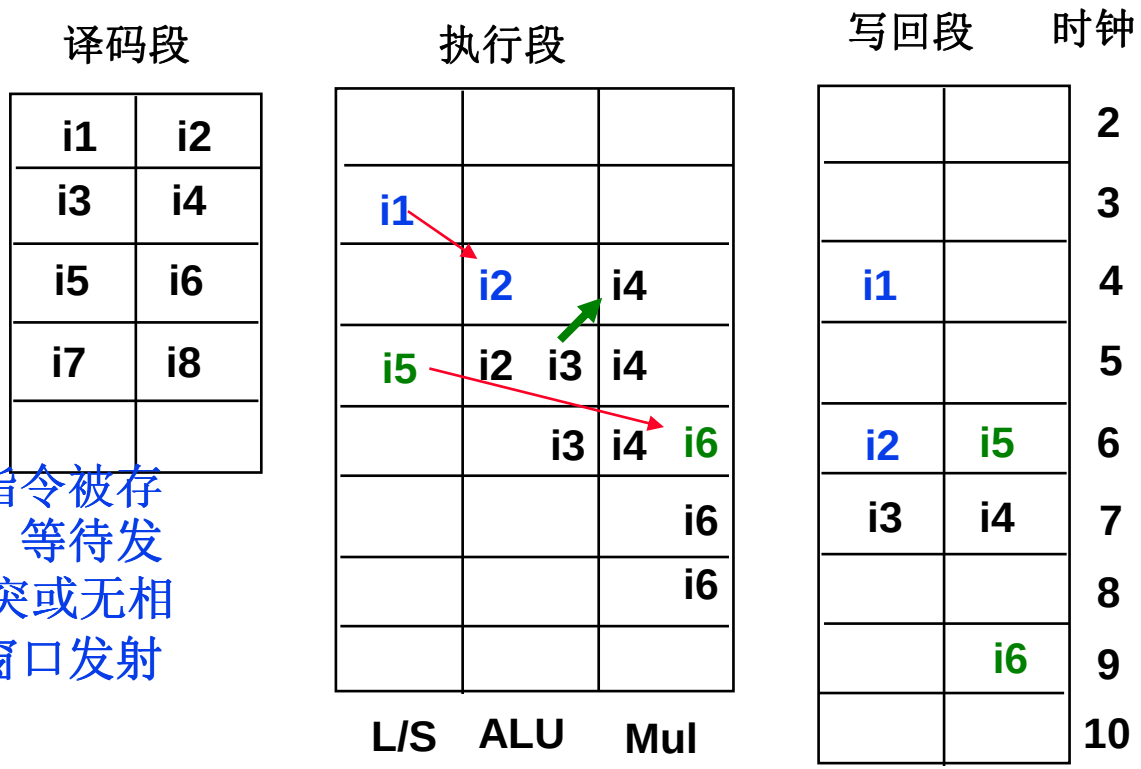
- 举例：2发射超标量，分为取指（F）、译码（D）、执行（E）、写回（W）。F、D、W段在一个时钟周期内完成（可同时有两条指令在这三个阶段）；E段有三个执行部件：Load/Store部件完成数据Cache访问需要1个时钟，整数ALU完成简单ALU操作只需2个时钟，整数乘法器完成乘法运算需要3个时钟。执行部件采用流水化方式。

取指和译码按顺序进行，发射前进行相关性检测，无关指令可先行发射和先行完成!

无序发射无序完成的过程如下：

例如：i4在i3前面发射！

```
i1    lw  $1, A
i2    add $2, $2, $1
i3    add $3, $3, $4
i4    mul $4, $5, $4
i5    lw $6, B
i6    mul $6, $6, $7
```



无序发射的超标量中，译码后的指令被存放在一个“指令窗口”的缓冲器中，等待发射.当所需功能部件可用、且无冲突或无相关性阻碍指令执行时，就从指令窗口发射，与取指和译码的顺序无关。

只要保证i1和i2、i5和i6之间的发射和完成顺序即可！

BACK

动态流水线调度的必要性

- 编译器可以依据数据依赖关系来调度代码，为什么还要超标量处理器来动态调度？
  - 并不是所有阻塞都能事先确定，动态调度可在阻塞时，提前执行无关指令
    - 例如，**Cache**缺失是不可预见的阻塞
  - 动态分支预测需要根据执行的真实情况进行预测
  - 采用动态调度使得硬件将处理器细节屏蔽起来

（不同处理器的发射宽度、流水线延时等可能不同，流水线的结构也会影响循环展开的深度。通过动态调度使得处理器细节被屏蔽起来，软件发行商无需针对同一指令集的不同处理器发行相应的编译器，并且，以前的代码也可在新的处理器上运行，无需重新编译）

理解程序的性能：

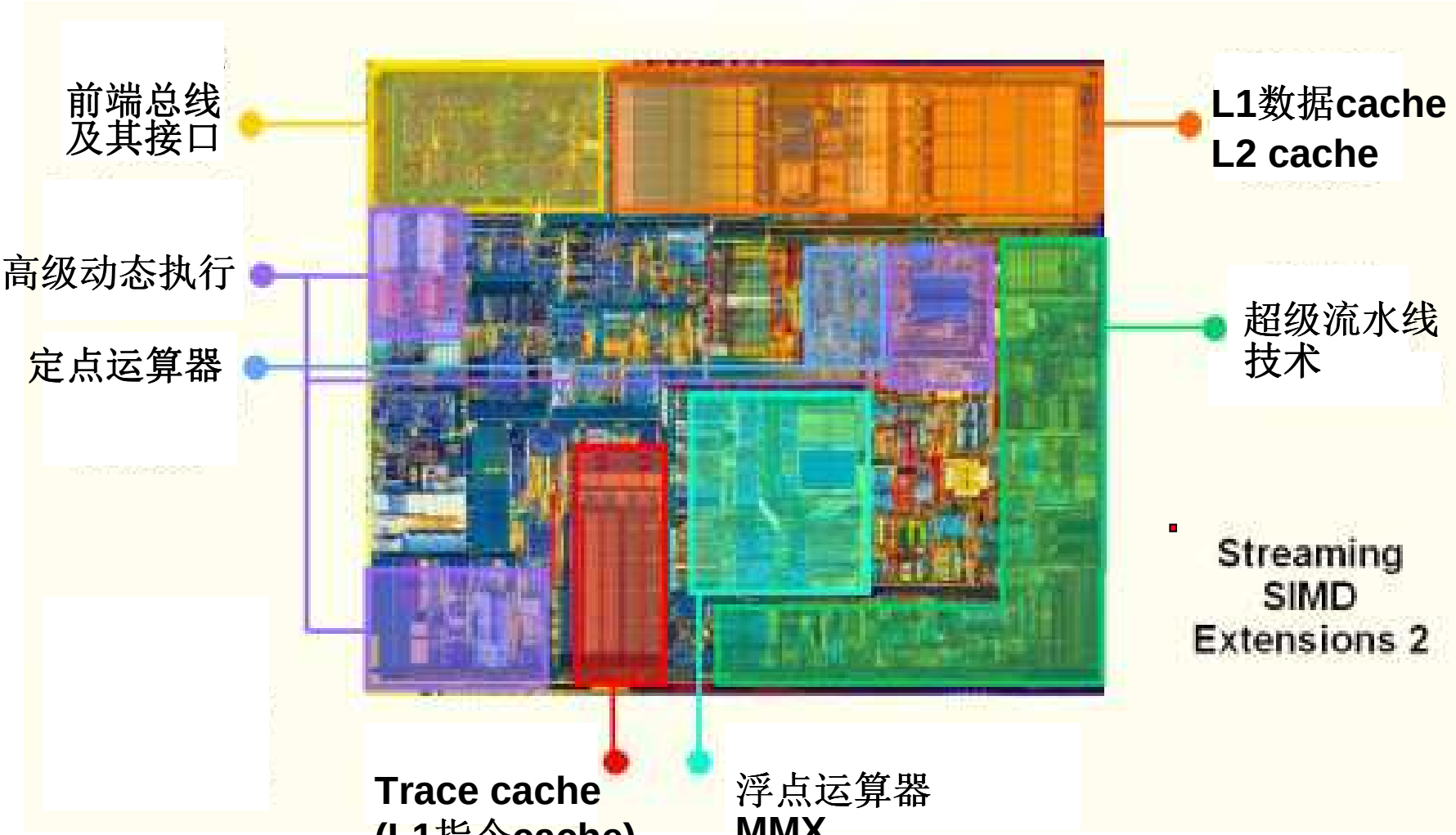
高性能微处理器并不能持续进行多条指令的发射，原因有：

(1) 指令间的高度依赖关系限制了指令之间的并行执行，特别是隐含依赖关系的存在。例如，使用指针的代码段，存在隐含依赖。

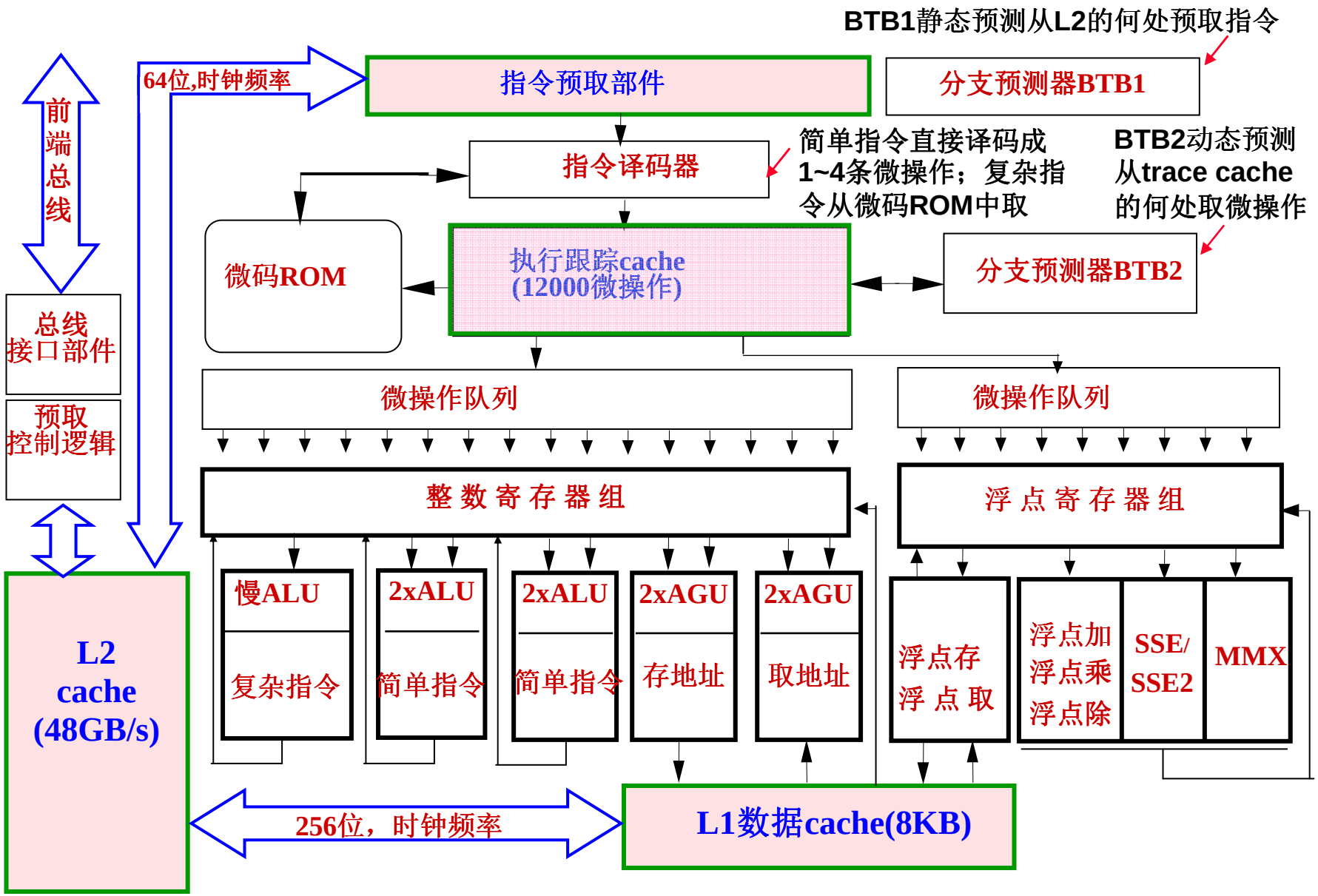
(2) 分支指令预测错误。

(3) 内存访问引起的阻塞（**Cache**缺失、缺页等）使得流水线难以满负荷运转。

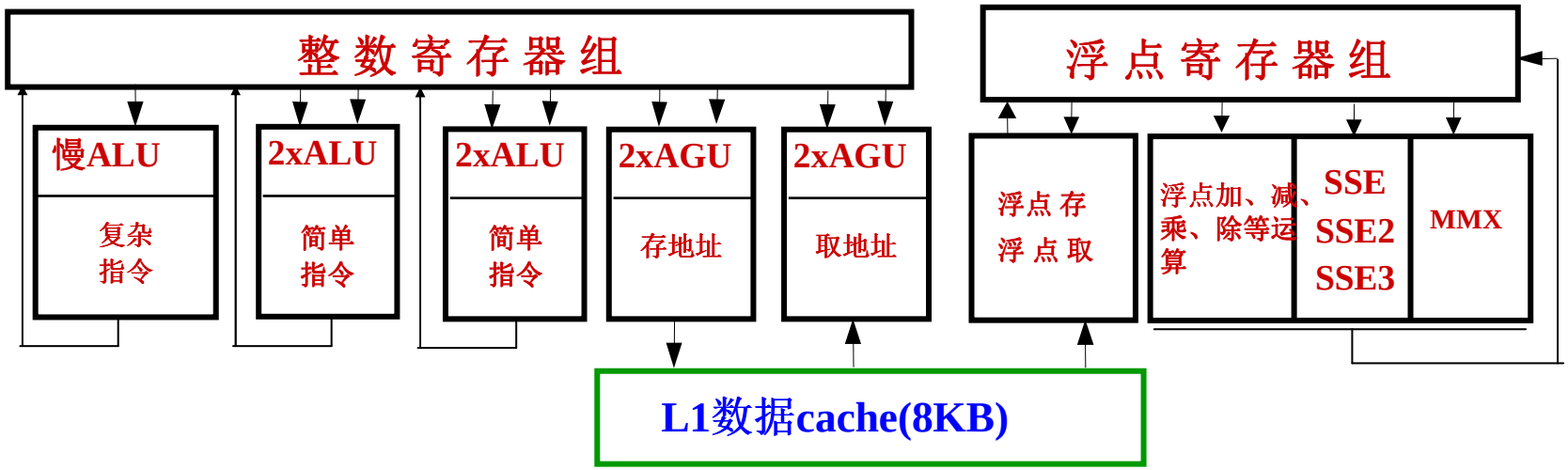
回顾：Pentium 4 处理器的芯片布局



回顾：Pentium 4 处理器的逻辑结构



Pentium 4的超标量结构运算器



°采用超标量（superscalar）结构，一共包含9个运算部件，可同时工作，所花时钟不同

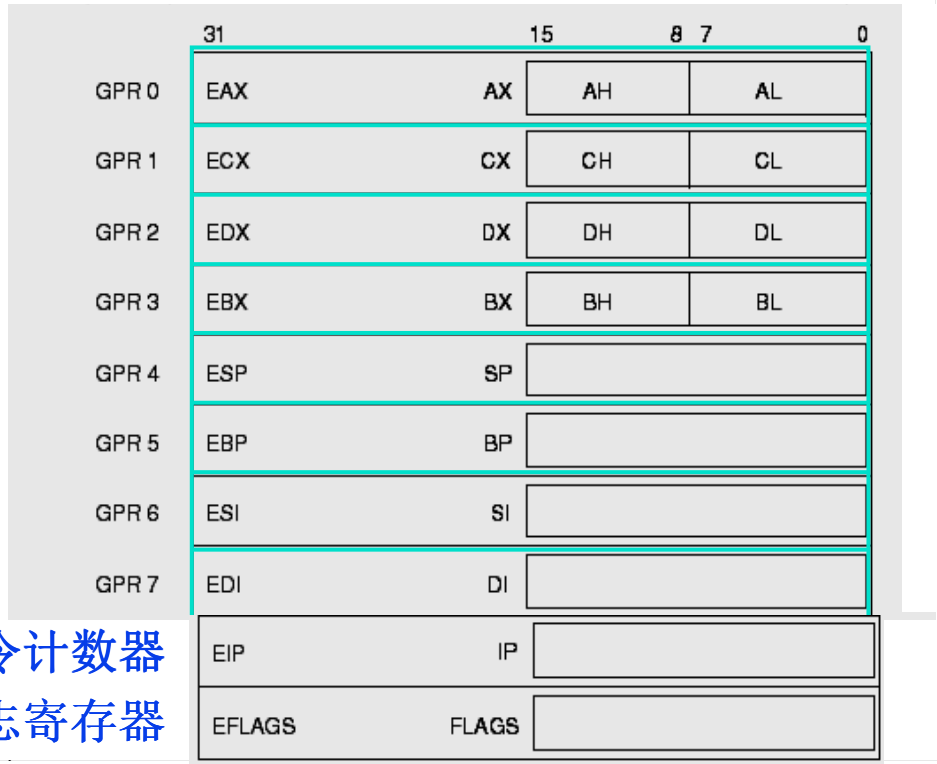
- 2个高速整数ALU(每个时钟周期进行2次操作)， 用于完成简单的整数运算(如加、减法)
- 1个慢速整数ALU(需要多个时钟周期才能完成1次操作)，用于完成整数乘、除法运算
- 2个地址生成部件（AGU），用于计算操作数的有效地址，所生成的地址分别用于从内存取操作数或向内存保存操作结果
- 1个运算部件用于完成浮点操作数地址的计算
- 1个运算部件用于完成浮点加法、乘法和除法运算
- 1个运算部件用于执行流式的SIMD处理（SSE/SSE2/SSE3指令）
- 1个运算部件用于完成多媒体信号处理（MMX指令）

在运算部件中执行的是微操作，而不是指令！运算器中的操作采用流水方式！

回顾：Pentium 4 的用户可见寄存器组

P4 Pentium 80386 80486 8086 8088

整数寄存器组



在**Pentium4**内部，整数和浮点数各有**128**个物理寄存器

寄存器换名操作：将用户可见的逻辑寄存器换成内部的物理寄存器

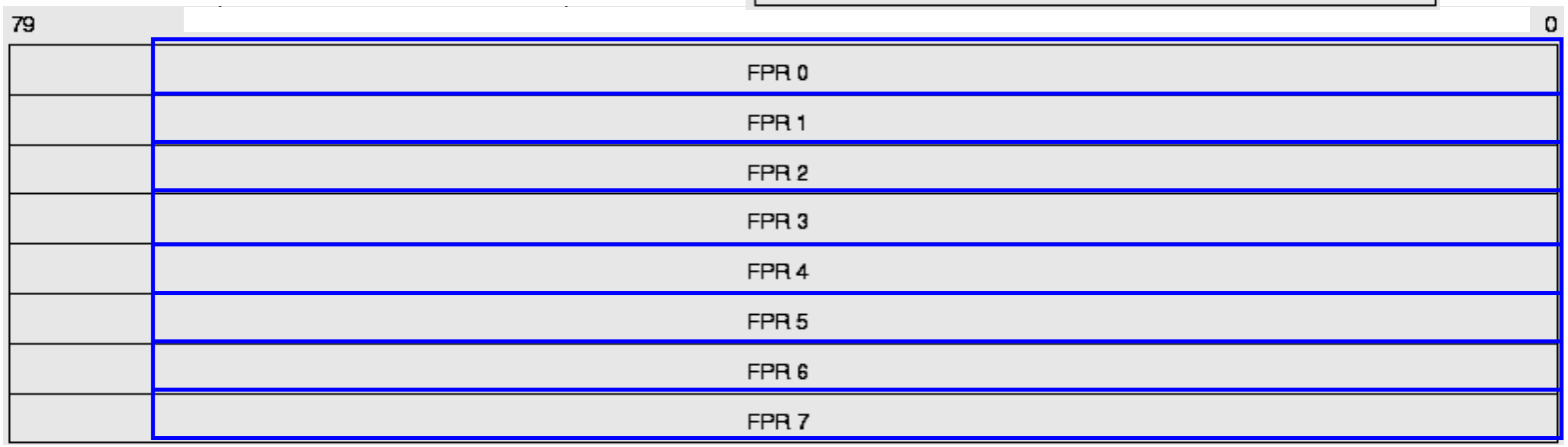
寄存器换名时，要确定是真实依赖还是名字依赖（反依赖）

名字依赖时可用不同的物理寄存器替换相同的逻辑寄存器

指令计数器

标志寄存器

浮点寄存器组



Pentium4 流水线结构部分

每个MOP相当于一条RISC指令
但其格式没有公开

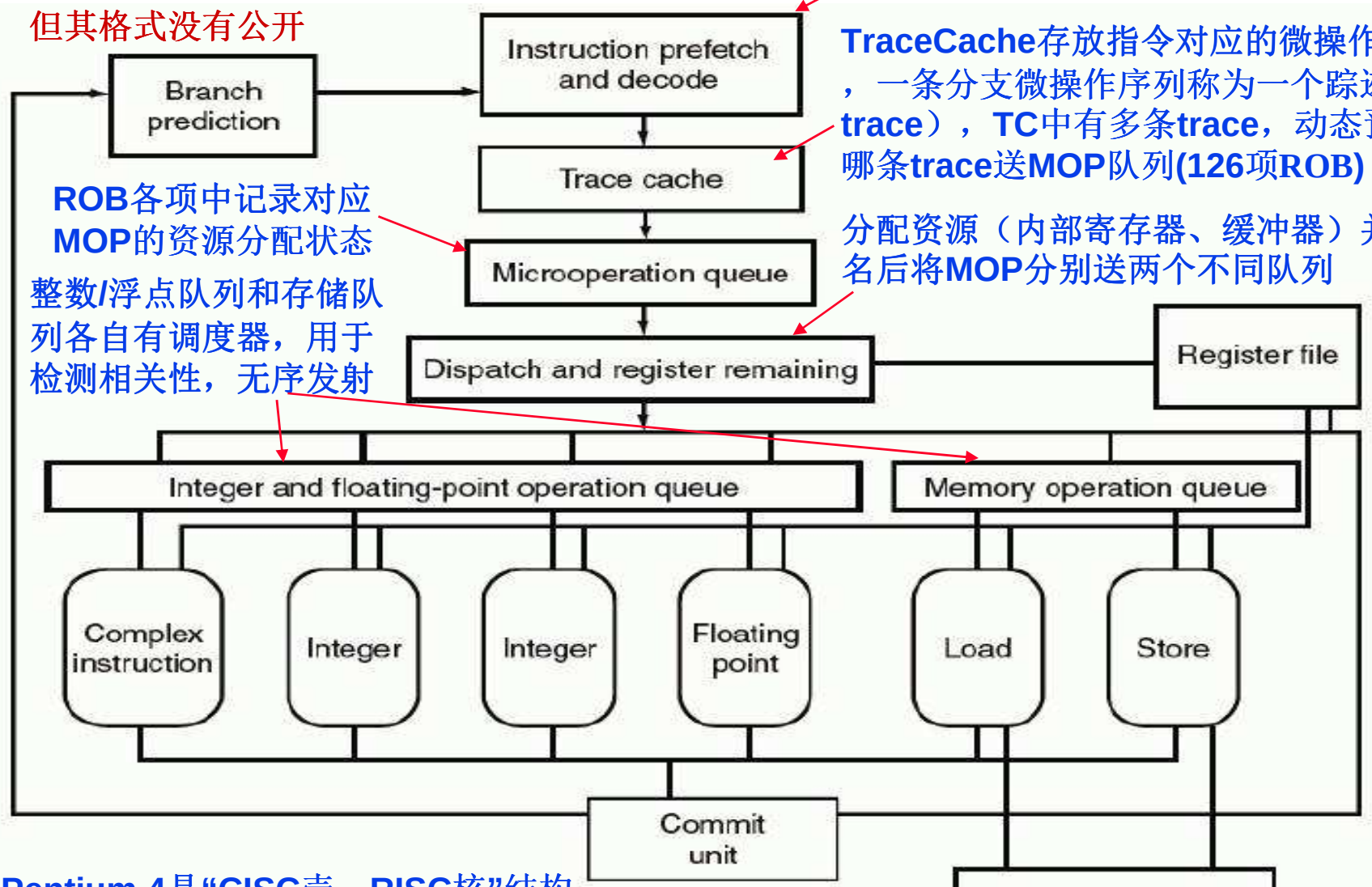
指令译码结果是对应的若干条微操作MOP

TraceCache存放指令对应的微操作序列，一条分支微操作序列称为一个踪迹（trace），TC中有多条trace，动态预测将哪条trace送MOP队列(126项ROB)

ROB各项中记录对应MOP的资源分配状态

分配资源（内部寄存器、缓冲器）并重命名后将MOP分别送两个不同队列

整数/浮点队列和存储队列各自有调度器，用于检测相关性，无序发射



Pentium 4是“CISC壳、RISC核”结构
从TC中取MOP开始，进入MOP的20级超标量超流水线

Pentium 4的指令译码 – 对指令功能进行分解

- 指令译码逻辑：
  - 功能：将指令转换为一组基本操作，称为微操作MOP
  - 输入：程序中的指令
  - 输出：微操作（简单计算任务）

例1：addl　%eax, %edx 的译码结果为什么？

一个“加法”操作（对应一个微操作MOP）

例2：addl　%eax, 4 (%edx) 的译码结果呢？

四个简单操作（对应四个微操作MOP）：

“地址计算”：Reg[%edx]+4->addr

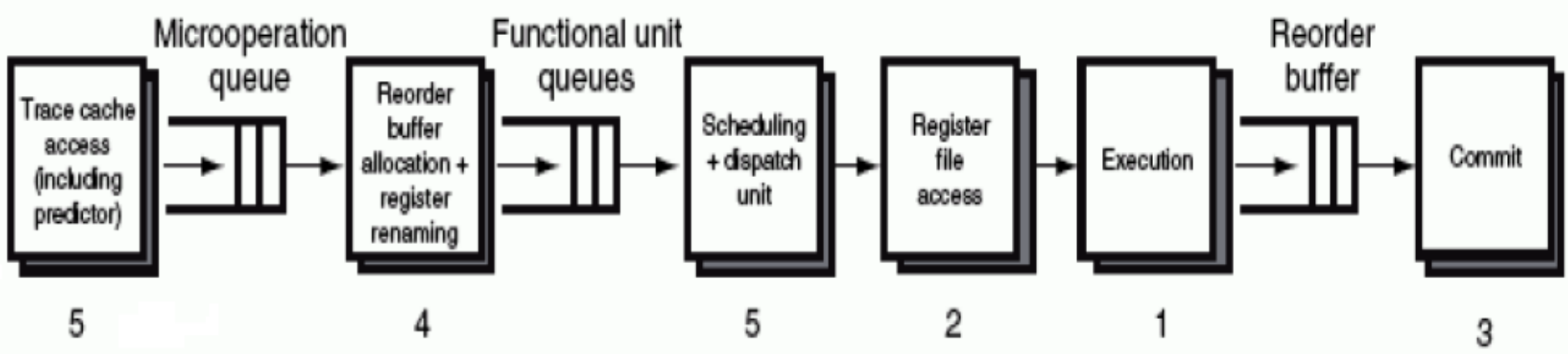
“装入”：Mem[addr]->Reg[Rtemp]

“加法”：Reg[Rtemp]+Reg[%eax]->Reg[Rtemp]

“存数”：Reg[Rtemp]->Mem[addr]

一个微操作相当于一条RISC指令，译码生成的微操作序列被存放到Trace Cache中

Pentium4 的20级超流水线(Hyper-pipeline)



整数运算微操作流水线为20级，浮点为29级（执行阶段的长度不同）

两个drive段用于芯片内传输信号的驱动，使其保证长距离传输

沿一个踪迹顺序取MOP，直到遇到一条转移MOP，通过BTB2预测下个踪迹开始点，继续取MOP送ROB/Alloc/Ren部件。预测目标处MOP不在时，要通知指令预取器，快从L2中取指令并译码。

一个周期3条MOP送ROB。ROB有126项，记录每个MOP及分配的资源和执行状态，根据资源分配情况进行寄存器重命名后，分别送两个MOP队列中进行排队。

每个队列按FIFO将MOP送到各自的调度器，在调度器中进行数据相关性检测，当所有源操作都就绪时，将MOP发射到对应的执行部件。是“无序”发射。

被发射的MOP开始读取物理寄存器中的源操作数，或从旁路由L1-D Cache读取。

在不同的执行部件中执行。每个部件执行时间长短不同

建立标志信息ZF/CF等，并将执行结果写入物理寄存器。对BTB2中预测是否正确进行确认及相应处理

本讲小结

- 有以下两种指令级并行(ILP)技术（即：高性能流水线形式）
  - 超流水线：更多的流水线级数
  - 多发射流水线：同时发射多个指令，有多条流水线同时进行
    - 静态多发射（VLIW处理器+编译器静态调度）
    - 动态多发射（超标量处理器+动态流水线调度）
- 静态多发射（VLIW（超长指令字）处理器）
  - 由编译器静态推测来完成“指令打包”和“冒险处理”
  - MIPS 2-发射Datapath中有2个执行部件，将2条指令打包，并同时译码执行
  - 采用循环展开进行指令调度，能得到很好的性能
  - IA-64采用VLIW级数，Intel特称其为EPIC技术，3条指令打包
- 动态多发射（超标量处理器）
  - 指令执行时由硬件动态推测，多个执行部件，同时发射多个指令到执行部件
  - 3种动态多发射流水线的执行模式
    - 按序发射按序完成、按序发射无序完成、无序发射无序完成
  - Pentium 4 动态多发射流水线（无序发射、无序完成）
    - 简单指令由硬件译码器 + 复杂指令由微操作ROM 产生 MOP
    - 指令对应的MOP存放在 trace cache中，按一条条trace存放 （同一条指令对应的若干微操作可能存放在不同的trace中）
    - 20级以上超流水线、3发射超标量、2个队列动态调度、126条微操作同时执行
    - 指令静态预测 + 微操作动态预测