第10章 指令级并行

授课教师：车喜龙

chexilong@jlu.edu.cn

Outline

指令级并行的概念

- 当指令之间不存在相关时，它们在流水线中是可以重叠起来并行执行的。这种指令序列中存在的潜在并行性称为指令级并行。

  （ILP：Instruction-Level Parallelism）

- 本章研究：如何通过各种可能的技术，在流水线思想的基础上，获得更多的指令级并行性。

  硬件技术(动态)＋软件技术（静态）

  必须要两种技术互相配合，才能够最大限度地挖掘出程序中存在的指令级并行。

相关与冲突

- 相关：两条指令之间存在某种依赖关系。

  如果两条指令相关，则它们就有可能不能在流水线中重叠执行(1拍延后)或者只能部分重叠执行(多拍延后)。
- 冲突：由于相关的存在，使得指令流中的下一条指令不能在顺次重叠的时钟周期进行。
- 相关有3种类型，因此冲突也有3种类型
  - 结构相关（导致结构冲突）
  - 数据相关（导致数据冲突）
  - 控制相关（导致控制冲突）

结构相关与结构冲突

- 结构相关：如果某种指令组合因资源冲突而不能顺利重叠执行，则称该机器具有结构相关。
- 结构冲突：因硬件资源满足不了指令重叠执行的要求而发生的冲突。
- 常见的导致结构相关的原因：
  - 功能部件不是全流水
  - 重复设置的资源的份数不够
- 结构相关导致结构冲突的例子
  - 指令与数据存在同一存储器
  - IF段的访存（取指令）与MEM段的访存（读/写数据）发生冲突。
- 结构冲突的解决方法
  - 允许结构冲突发生，发生冲突时插入流水线气泡
  - 从根本上消除结构冲突的发生，设置足够多的硬件资源

数据相关与数据冲突

- 数据相关：当指令在流水线中重叠执行时，流水线有可能改变指令读/写操作数的顺序，使之不同于它们在非流水实现时的顺序，从而因读写顺序冲突而不能顺利重叠执行，则称该指令集合具有数据相关。
- 指令集中的数据操作顺序导致相关的情况：
  - 写后读相关（RAW），又称真数据相关，生产者消费者相关
  - 读后写相关（WAR），又称反相关
  - 写后写相关（WAW），又称输出相关
  - 读后读相关（RAR），其实不相关

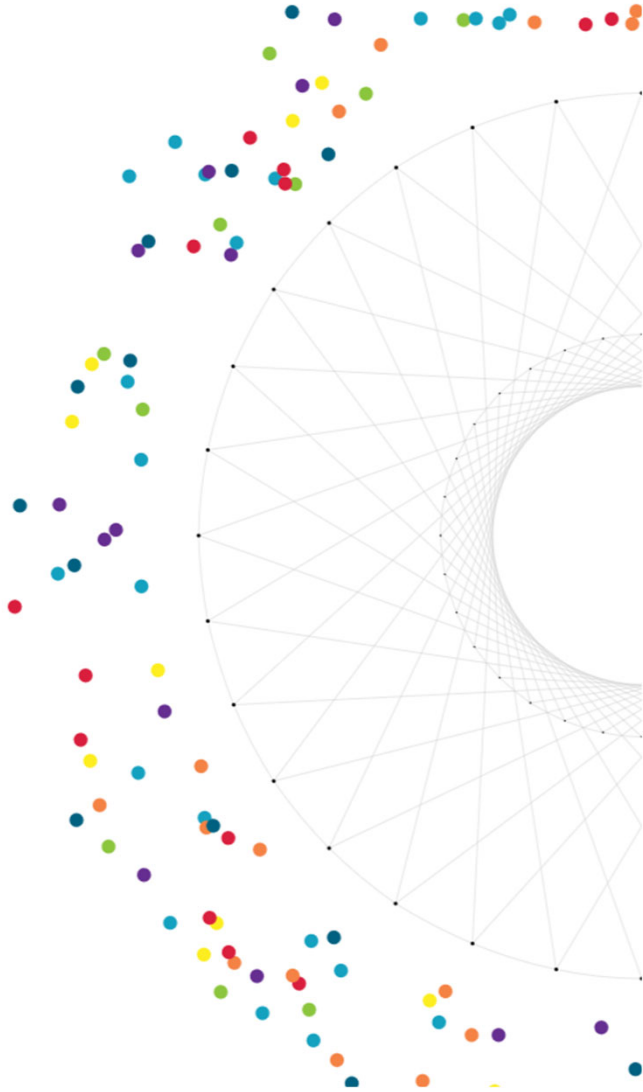
名相关={读后写相关, 写后写相关}

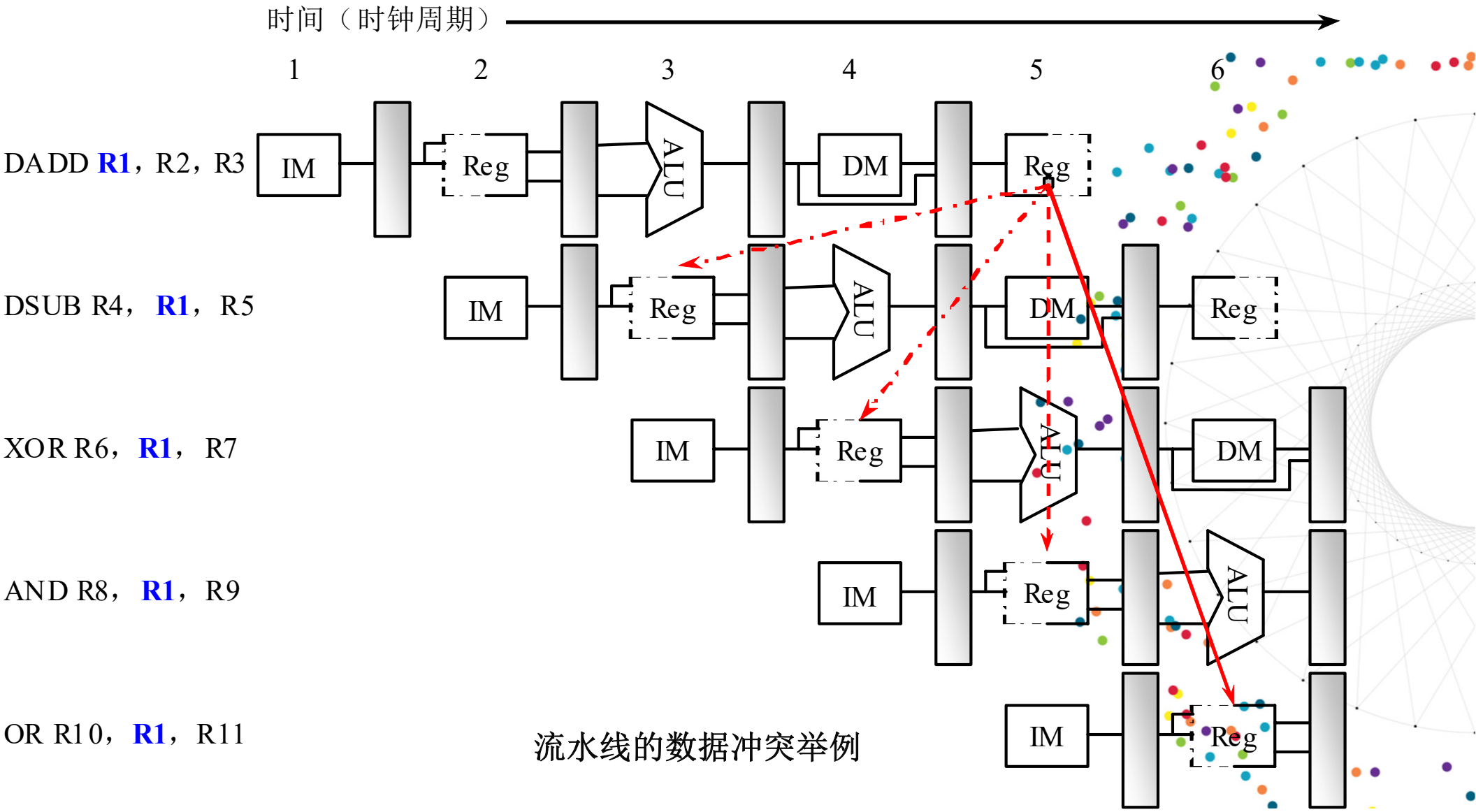
- 名相关定义：如果两条指令使用相同的名，但是它们之间并没有数据流动，则称这两条指令存在名相关。
- 读后写相关与写后写相关均为名相关
- 名相关的消除（换名）可以通过改变指令中操作数的名来消除，即换名技术。
- 换名过程既可以用编译器静态实现，也可以用硬件动态完成。

数据冲突的例子

```
DADD   R1, R2, R3
DSUB   R4, R1, R5
XOR    R6, R1, R7
AND    R8, R1, R9
OR     R10, R1, R11
```

分析见下页




时间（时钟周期）
1
2
3
4
5
6
DADD R1，R2，R3
DSUB R4，R1，R5
XOR R6，R1，R7
AND R8，R1，R9
OR R10，R1，R11
IM
Reg
ALU
DM
Reg

流水线的数据冲突举例

- 数据冲突的解决方法
  - 硬件中采用定向技术减少数据冲突引起的停顿
  - 硬件中采用流水线互锁机制，发生冲突时插入流水线气泡，直到数据冲突消失
  - 软件中（编译器）采用指令调度解决数据冲突

控制相关与控制冲突

- 控制相关：是指由分支指令引起的相关，它需要根据分支指令的执行结果来确定后续指令是否执行。
- 控制冲突：流水线遇到分支指令和其他会改变PC值的指令所引起的冲突。
- 控制相关的例子：“if-then”结构

```
if p1 {
        S1;          S1与p1控制相关;
    };
S;                   S2与p2控制相关;
if p2 {
        S2;          S与p1和p2无关。
    };
```

流水线冲突

$$CPI_{流水线} = CPI_{理想} + 停顿_{结构冲突} + 停顿_{数据冲突} + 停顿_{控制冲突}$$

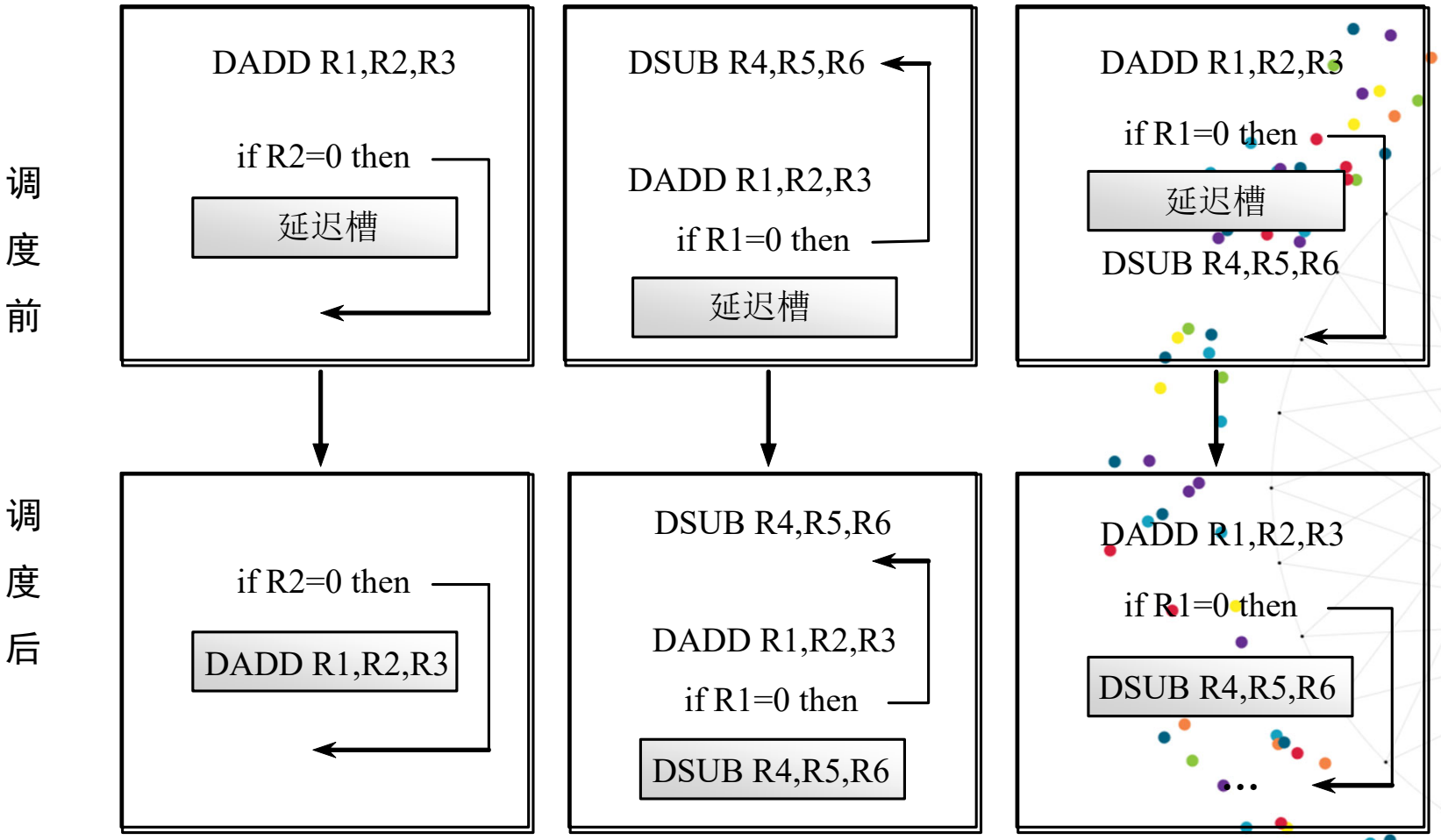
- 理想CPI是衡量流水线最高性能的一个指标，减少任何一种停顿，都可以有效地减少实际CPI。
- 相关是程序固有属性，它反映程序中指令之间的相互依赖关系。
- 一次相关是否会导致冲突发生以及该冲突会带来多长的停顿，则是流水线的属性。
- 可以从两个方面来解决相关：
  - 硬件解决：多倍资源，定向路径，插入气泡，冻结/排空策略，分支电路前移等
  - 软件解决：指令调度，预测成功，预测失败，延迟槽，分支取消等


开发ILP的方法
基于软件的静态开发方法
编译器调度
分支预测（成功/失败）
分支延迟槽
循环展开
基于硬件的动态开发 方法
记分牌算法
Tomasulo算法
分支历史表
分支历史缓冲器
基于硬件的前瞻算法

- 分支预测
  - 总是预测分支失败
  - 总是预测分支成功
  - 基于延迟槽的指令调度
  - 基于分支取消的指令调度
- 这几种方法对分支的处理方法在程序的执行过程中始终是静态的，要么总是预测分支成功，要么总是预测分支失败。

- **基于延迟槽的指令调度**
  - 编译器每当遇到分支指令，就在其后连续生成k个空指令位，称为延迟槽，并根据不同的调度规则将原始代码中分支指令附近的指令放进延迟槽中，如果延迟槽没有填满，剩余部分用nop填充，即气泡。
  - 延迟槽中的指令同分支指令都看做普通的指令顺序流水，无论分支指令成功与否，都先按顺序执行延迟槽中的指令。延迟槽中的指令“掩盖”了流水线原来必须插入的暂停周期，减少了分支指令带来的延迟。
  - 无论分支是否成功，在执行延迟槽中的指令过程中，分支后继指令的PC值都已经准备好，因此延迟槽中的指令一旦都进入流水，下一条正确的指令可以马上进入流水。

- **在延迟槽中放入有用的指令**
  - 由编译器完成。延迟分支能否带来好处取决于编译器能否把有用的指令调度到延迟槽中。
- **三种常用的调度方法** 图在下页
  - 从前调度（最佳方法）：从分支指令之前找一条独立的指令移动到延迟槽中。
    - 延迟槽指令i+1=指令t （t<i）
  - 从目标处调度：把分支成功的目标地址指令复制到延迟槽中，并把分支目标地址改成分支后继地址。永远猜测分支是成功的，如果猜错，则丢弃延迟槽中指令的结果。
    - 延迟槽指令i+1=指令j，指令j=指令j+1。
  - 从失败处调度：把分支失败的目标地址指令移动到延迟槽中。永远猜测分支是失败的。如果猜错，则丢弃延迟槽中指令的结果。
    - 延迟槽指令i+1=指令i+2，指令t=指令t+1 （t>i+1）。



(a) 从前调度　(b) 从目标处调度　(c) 从失败处调度

- 基于分支取消的指令调度
  - 编译时，对每个分支在编译的时候给出一个成功与否的预测，并根据这个预测来调度预测路径上的指令进入延迟槽。
    - 如果预测分支成功，则将分支目标地址指令放入延迟槽；
    - 如果预测分支失败，则将分支后继地址指令放入延迟槽。
  - 执行时，当分支执行结果和预测相符，则延迟槽指令的执行有效；当分支执行结果和预测不符，则延迟槽指令的执行取消。
    - 如果猜对，下条指令的PC根据延迟槽指令更新；
    - 如果猜错，下条指令的PC根据分支指令的ID段计算更新；
  - 这种方法统一了延迟槽的指令调度条件，便于编译器调度，支持对分支成功的预测和对分支失败的预测。

指令的动态调度

- 静态调度：在出现数据相关时，为了消除或者减少流水线空转，编译器确定并分离出程序中存在相关的指令，然后进行指令调度，并对代码进行优化。
  - 在编译期间静态完成，通过把相关的指令拉开距离来减少可能产生的停顿。优化通常是面向特定体系结构的。
- 动态调度：通过硬件重新调度相关的指令的执行顺序，减少冲突导致的处理器空转。
  - 在程序的执行过程中动态完成，能够处理在编译时情况不明的相关（比如涉及到存储器访问的相关），以增加硬件复杂性为代价简轻编译器的负担（软件硬化）。

动态调度的基本思想

1. 基本MIPS流水线的局限性

- 指令必须按序流出和执行，一旦一条指令受阻，其后的指令都将停顿，不管与受阻的指令有没有关系。
- 考虑下面一段代码：

  指令i　DIV.D　F4，F0，F2

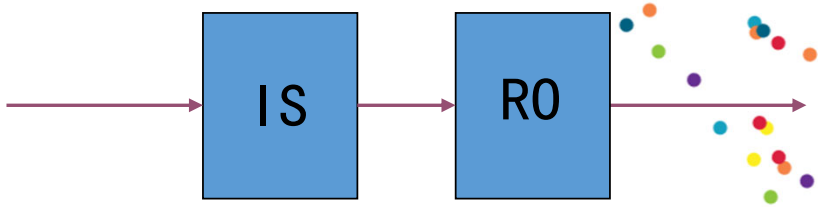
  指令i+1 SUB.D　F10，F4，F6

  指令i+2 ADD.D　F12，F6，F14

  - 指令i+1与指令i关于F4相关，导致流水线停顿。指令i+2与前两条指令都无关，但也因指令i+1而无辜受阻。
  - 能否让无辜指令先执行？
- 解决办法：允许乱序执行与乱序完成

2. 在前述基本流水线中：ID段负责检测结构冲突与数据冲突。为了支持乱序执行，我们将ID段再分为两个阶段.

- IS段：指令流出Issue。包括译码及检测结构冲突，如果无结构冲突，就允许指令流出。（顺序从IS进入RO）
- RO段：读操作数Read Operands。检测数据冲突，如果有数据冲突就等待冲突消失，然后才能读操作数，并将指令送入EX段（乱序从RO进入EX段）。
- 这样无辜的指令i+2可以先执行了，但是乱序执行后怎样保证数据相关正确性？



解决结构冲突　　解决数据冲突

3. 在经典5段流水线中，是不会发生WAR冲突和WAW冲突的。但乱序执行就使得它们可能发生了。

- 例如，考虑下面的代码

```
DIV.D    F10, F0, F2
SUB.D    F10, F4, F6
ADD.D    F6, F8, F14
```

- DIV和SUB输出相关，SUB和ADD反相关，如果SUB早于DIV完成，就会使得后面使用F10的程序取到错误结果。
- 如果能消除这两种相关，乱序执行也不会导致WAR冲突和WAW冲突，这就需要用到寄存器换名技术。
- 乱序执行也会导致多条指令都处于执行状态，我们假设流水线具有多个功能部件，多条指令可同处EX段。

通过寄存器换名可消除WAR/WAW冲突。考虑以下代码：

```
DIV.D    F0，F2，F4
ADD.D    F6，F0，F8
S.D   F6，0（R1）
SUB.D    F8，F10，F14
MUL.D    F6，  F10，F8
```

引入 S消除ADD与MUL的WAW冲突　T消除ADD与SUB的WAR冲突

- 消除名相关
  - 把这段代码改写为：

```
DIV.D    F0，F2，F4
ADD.D    S，F0，F8
S.D      S，0（R1）
SUB.D    T，F10，F14
MUL.D    F6，F10，T
```

4. 指令乱序完成增加了异常处理的复杂度

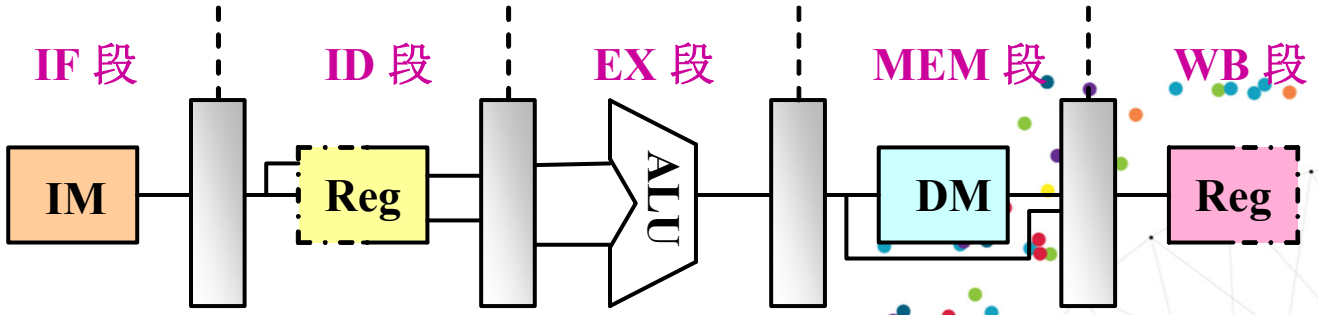
- 精确异常：当执行指令i导致发生异常时， 处理机的现场跟严格按程序顺序执行时指令i的现场相同。
- 不精确异常：当执行指令i导致发生异常时，处理机的现场跟严格按程序顺序执行时指令i的现场不同。
- 动态调度必须要保持正确的异常行为
  - 动态调度情况下，对于一条会产生异常的指令来说，只有当处理机确切地知道该指令将被执行后，才允许它产生异常。
- 即使保持了正确的异常行为，动态调度处理机仍可能发生不精确异常。
  - 不精确异常通常是乱序完成导致的，它使得在异常处理后难以接着继续执行程序。

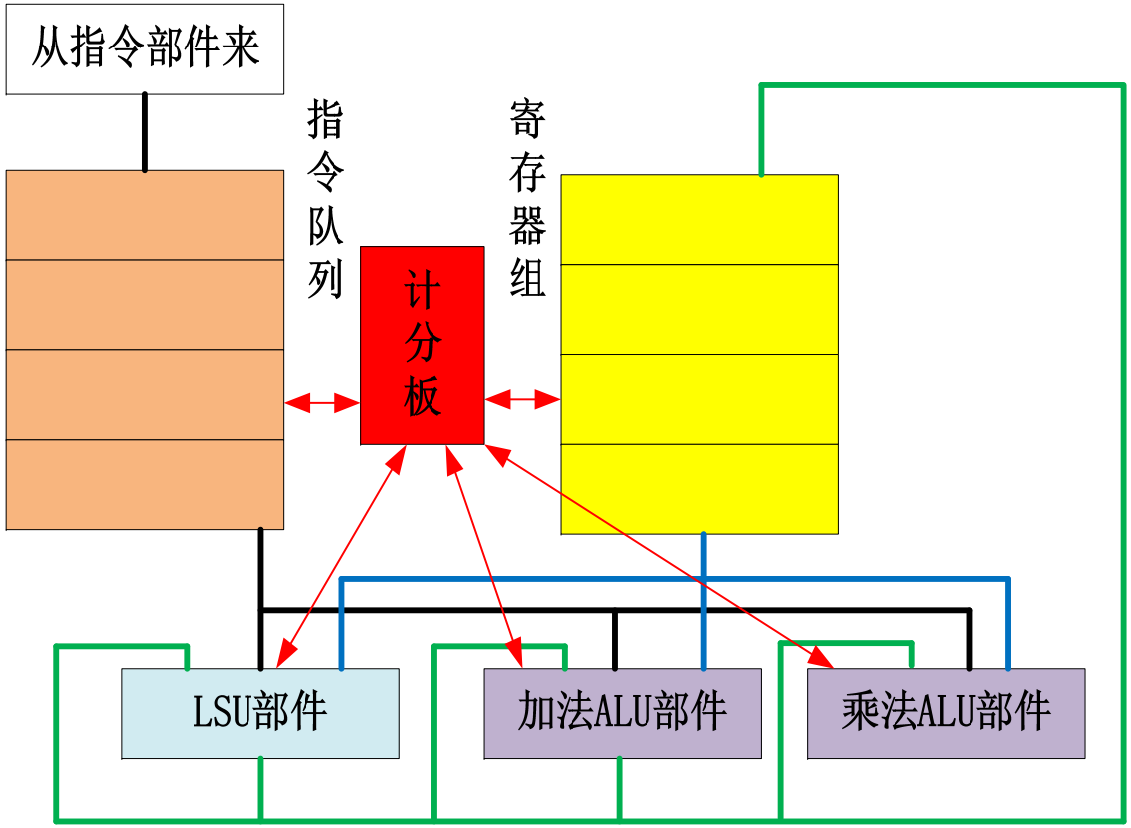
Scoreboard算法

- 1964年CDC61350提出，需求引导技术：数据相关分为RAW、RAR、WAR、WAW这四类，其中的RAR不是真正的数据相关，因此寄存器不会因为读而改变。真正需要处理的是RAW以及WAW和WAR这三种。在顺序执行的指令流水线中，WAR和WAW是不会出现问题的，因为指令顺序指令，后面的W不会覆盖前面的R或者W；但是对于乱序执行就不一样了，需要有算法解决；而对于RAW，道理也一样。
- 核心思想：用一个核心调度部件“记分牌”监控并调度所有的指令、数据和部件，从而允许指令乱序执行的技术。通过流出操作消除结构冲突和WAW冲突，通过读数操作消除RAW冲突，通过写结果操作消除WAR冲突。让与当前已发射指令不相关的后续指令尽早执行，从而减少总执行时间。

Scoreboard算法特点

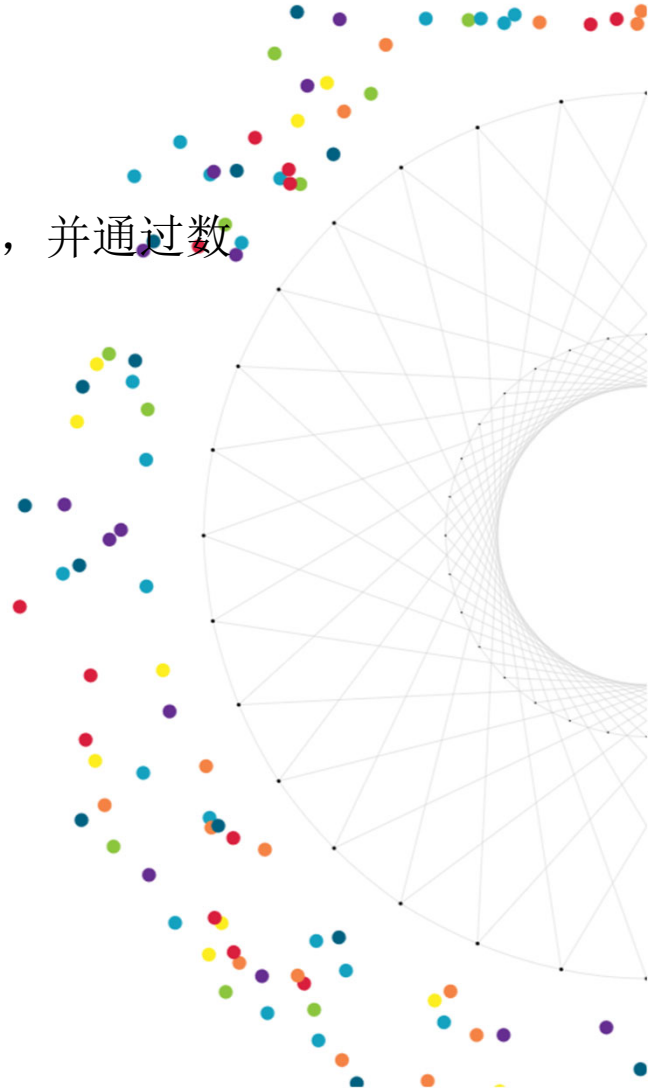
- 记分牌是一集中控制部件，其功能是控制数据寄存器与处理部件之间的数据传送。
- 在记分牌中保存有与各个处理部件相联系的寄存器中的数据装载情况。当一个处理部件所要求的数据都已就绪（装载完毕），记分牌允许处理部件开始执行。当执行完成后，处理部件通知记分牌释放相关资源。
- 记分牌中记录了数据寄存器和多个处理部件状态的变化情况，通过它来检测和消除或减少数据相关性，加快程序执行速度。





基于Scoreboard算法的处理器基本结构

- 寄存器组
  - 它们通过一对控制总线和数据总线连接到功能部件，并通过数据总线连接到写回接口。
- 指令队列
  - 指令部件送来的指令放入指令队列
  - 指令队列中的指令按先进先出的顺序流出 FIFO
- 执行部件
  - ALU负责LSU之外的所有指令
  - LSU内部含有加法器，用于执行Load/Store指令



使用Scoreboard算法的流水线需4个步骤：

- 流出
  - 流出条件：指令所需功能部件不忙（消除结构冲突），且目标寄存器没有被其他指令预约（消除WAW冲突）。
  - 流出动作：指令可从指令队列流出到功能部件(设为f)，更新记分牌内部数据表（更新操作后面讲）。（IS段）
- 读数
  - 读数条件：指令所需操作数全部就绪（消除RAW冲突），则指令可读取操作数，否则该功能部件挂起。
  - 读数动作：读取操作数，更新记分牌内部数据表（更新操作后面讲）。（RO段）

使用Scoreboard算法的流水线需4个步骤：

- 执行
  - 执行条件：功能部件获得执行所需全部操作数。
  - 执行动作：完成计算功能，产生计算结果，通知记分牌更新内部数据表（更新操作后面讲）（EX段，Load指令EX+MEM段）
- 写结果
  - 写结果条件：该功能部件中的指令与其他执行中的指令间无关于目标寄存器的WAR冲突（消除WAR冲突），且数据总线就绪
  - 写结果动作：将计算结果放到数据总线上，写入寄存器组，释放功能部件，记分牌更新内部数据表（更新操作后面讲）（WB段，Store指令WB+MEM段）

Scoreboard算法需要的各信息表

- 指令执行状态表：每条2项={指令字段,状态字段IREW}
- 寄存器预约表：每条2项={寄存器号,功能部件预约字段Qi}
- 功能部件状态表：每条10项
  ={L,B,O,Fi,Fj,Fk,Qj,Qk,Rj,Rk}
  - L：label，功能部件标识字段。
  - B：Busy，功能部件是否“忙”。
  - O：Operation，功能部件中指令的操作码。
  - Fi：目的操作数的寄存器编号。
  - Fj，Fk：源操作数的寄存器编号。
  - Qj，Qk：将要产生源操作数的功能部件标识。
  - Rj，Rk：源操作数是否已经在Fj，Fk中就绪。

符号说明： Qi,Fi的i Fj,Rj的j Fk,Rk的k

MUL.D F4，F0，F2

i j k

L.D F2，45（R3）

k j

S.D F3，40（R4）

k j

完整的Scoreboard算法（1/4）：指令的流出

```
Wait until (f.busy=No)&&(Qi[rd]=null)
// 若功能部件不忙且目标寄存器无WAW冲突
Do{
f.state=Issued;   [rd].Qi=f;   f.busy=Yes;
    f.op=op;
f.Fi=[Rd];   f.Fj=[Rs];   f.Fk=[Rt];
f.Qj=[Rs].Qi;   f.Qk=[Rt].Qi;   f.Rj=!Qj;
    f.Rk=!Qk;
}
//更新指令执行状态表，寄存器预约表，
//更新功能部件状态表；
```

完整的Scoreboard算法（2/4）：指令的读数

Wait until (Rj•Rk)

//指令所需操作数全部就绪，无RAW冲突

Do{

f.state=Read;

f.Qj=0;　f.Qk0;　f.Rj=0;　f.Rk=0;

}

//更新指令执行状态表，

//更新功能部件状态表；

完整的Scoreboard算法（3/4）：指令的执行

f.state=Exec

对于计算指令：

Result = f.Reg[rs] op f.Reg[rs] ;

对于ST指令：

Address = f.Reg[rs] + ##; //计算有效地址

对于LD指令：

Address = f.Reg[rs] + ##; //计算有效地址

Result = Mem[Address] ; //访存读数据

对于分支指令：

PC = newPC; //计算后继指令地址

完整的Scoreboard算法（4/4）：指令的写结果

```
Wait until {
for all F in flight: f.Fi ≠ F.[rs];
or
for all F in flight: f.Fk ≠ F.[rt];
or
A F in flight: f.Fi =
    F.[rs]&&F.Rj=0;
or
A F in flight: f.Fi =
    F.[rt]&&F.Rk=0;
}
```

```
do{
f.state=Write
[rd]=[Result]
ST指令数据写入存储器;
任意F，if (F.Qj=f) {
                F.Rj=1;F.Qj=0;
                }
任意F，if (F.Qk=f) {
                F.Rk=1;F.Qk=0;
                }
f.Qi=0;
f.busy=No;
}
```

例 单流出处理器采用基于scoreboard算法进行指令调度。有一个LSU部件（内部自带加法器），1个加法ALU部件，1个乘法ALU部件。指令序列执行前，指令均未流出。

（1）各个硬件操作及指令执行的时钟周期如下表

Issue	ReadOperand	Mem	Writeback	Execute				
				LD	ST	SUB	ADD	MUL
1	1	3	1	1	1	4	4	10

（2）待执行指令序列如下：

指令	对应变量
LD R2, (R1)	Rt=R2, Rs=R1, Imm=0
MUL R2, R2, #2	Rd=R2, Rs=R2, Rt=#2
ST R2, (R1)	Rt=R2, Rs=R1, Imm=0
SUB R1, R1, #4	Rd=R1, Rs=R1, Rt=#4
ADD R5, R4, R3	Rd=R5, Rs=R3, Rt=R4

问：

（1）请给出指令执行状态时钟周期表

（2）请给出第18周期末尾各状态表内容

Scoreboard 习题答案

解：指令状态表、功能部件状态表、REG状态表 内容如下

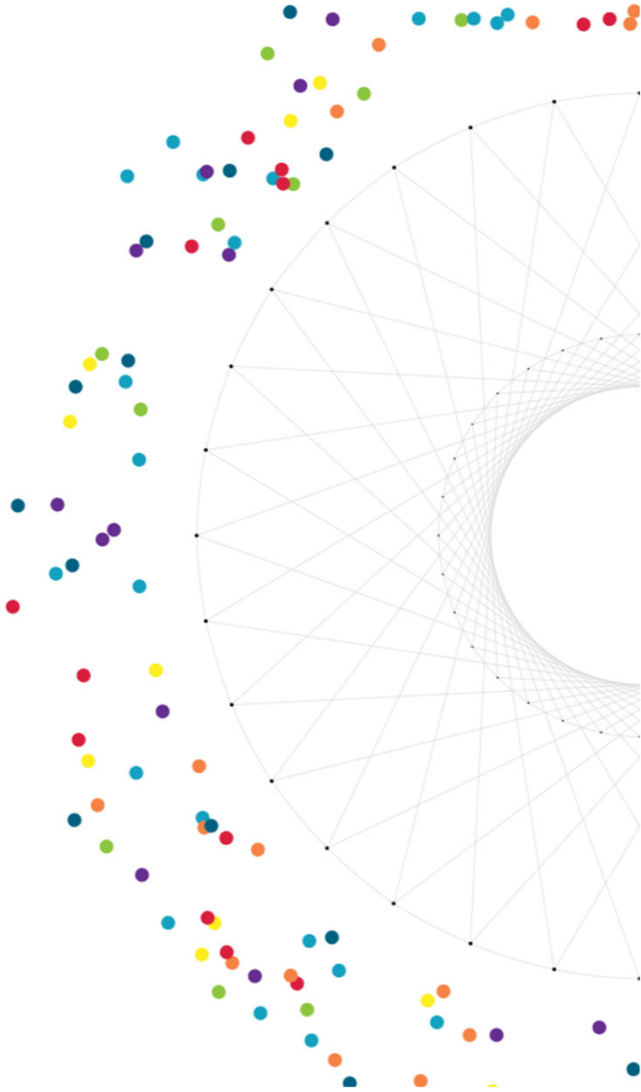
指令	IS	RO	EX	MEM	WB	备注	18末
LD R2, (R1)	1	2	3	4-6	7		已写结果
MUL R2, R2, #2	2	8	9-18	Null	19	RAW冲突	已执行
ST R2, (R1)	8	20	21	Null	22	结构+RAW	已流出
SUB R1, R1, #4	3	4	5-8	Null	21	WAR冲突	已执行
ADD R5, R3, R4	20	21	22-25	Null	26	结构冲突	未流出

label	Busy	Op	Fi(rd)	Fj(rs)	Fk(rt)	Qj(rs)	Qk(rt)	Rj(rs)	Rk(rt)
LSU	Yes	ST		R1			ALU1	1	0
ALU1	Yes	MUL	R2	R2	#2			0	0
ALU2	Yes	SUB	R1	R1	#4			0	0

	R1	R2	R3	R4	R5
Qi	ALU2	ALU1			

Scoreboard的功能

- 检测冲突并消除
  - IS段，消除结构冲突和WAW冲突
  - RO段，消除RAW冲突
  - WB段，消除WAR冲突
  - 数据相关，结构相关，控制相关？
- 记录软硬件状态
  - 指令状态，功能单元状态，寄存器预约状态
- 集中式指令动态调度
  - 所有指令都经过记分牌，
  - 所有功能部件都由记分牌监控，
  - 指令的每个阶段都更新记分牌

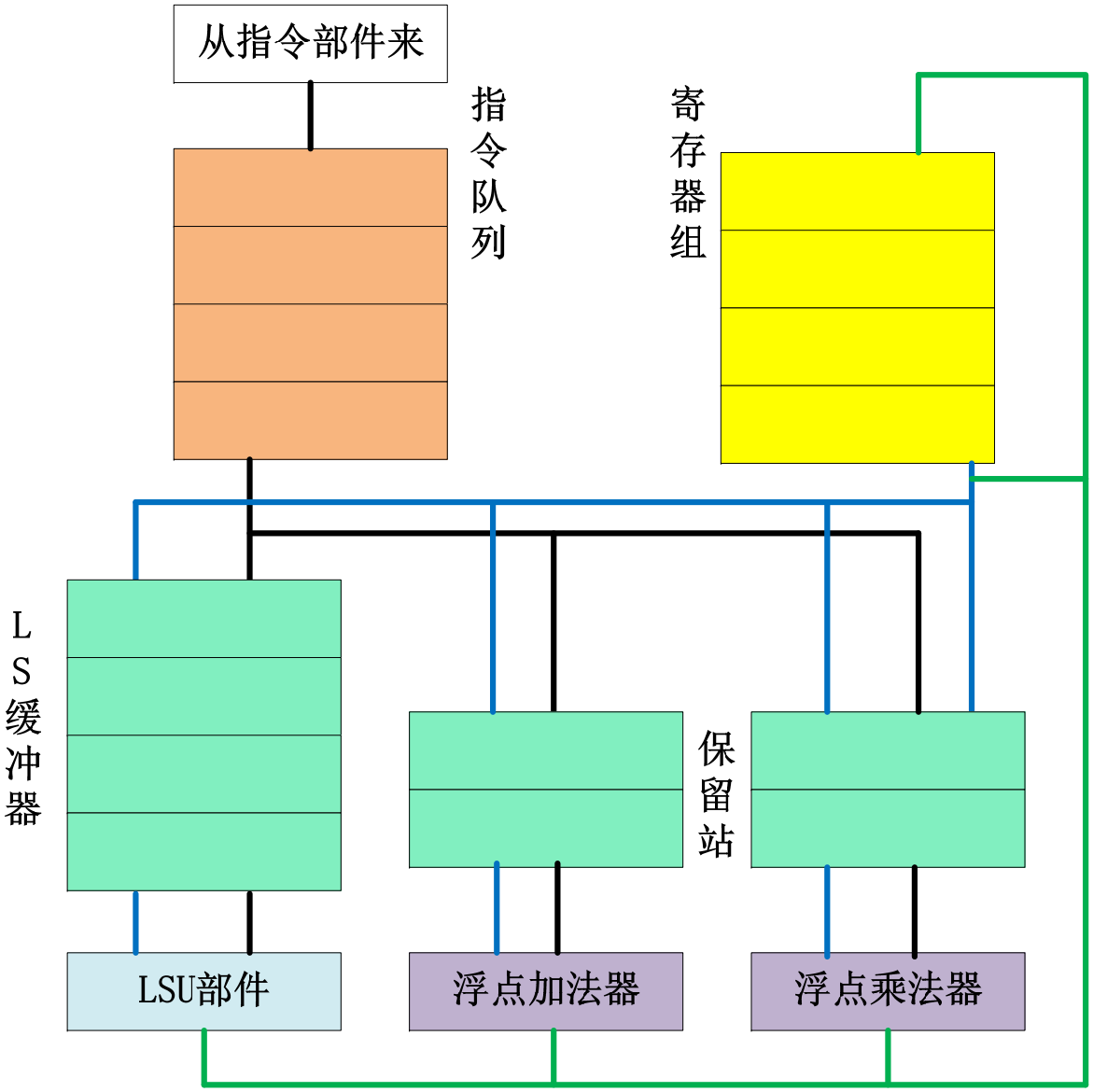


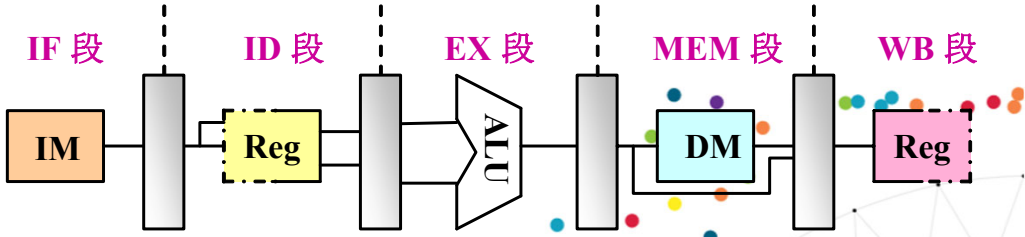
限制Scoreboard性能的因素

- 指令中可开发的并行性（决定了有多少独立的指令能够被发掘出来并行执行）
- 记分牌表项的入口条数（影响到流水线可以提前查找独立不相关指令的能力）
- 功能部件的类型与数量（影响到冲突的类型和数量）
- 真数据相关导致的停顿，有些通过乱序执行可以消除，有些仍然无法消除。
- 名相关导致的停顿理论上可以消除，记分牌却无能为力，对性能影响较大

Tomasulo算法

- Robert Tomasulo发明，IBM 3135/91首先采用。
- 核心思想：通过引入保留站（Reservation Station）和相关逻辑硬件来实现寄存器换名技术。完全消除名相关，并尽最大可能减少真数据相关引发的流水线阻塞。
- 每条指令根据不同的类型流出到不同的保留站中。保留站保存已流出并等待到对应的功能部件进行执行的指令及其相关信息。在一条指令流出到保留站时：
  - 如果该指令的源操作数已经在寄存器中就绪，则将该操作数读出来保存到保留站中。（寄存器换成数据）
  - 如果操作数还没有计算出来，则把将会产生这个操作数的其他保留站的标识保存到保留站中。（寄存器换成保留站号）





基于Tomasulo算法的MIPS处理器浮点部件的基本结构

- 浮点部件
  - 浮点加法器：浮点加减法
  - 浮点乘法器：浮点乘除法
  - 浮点存储单元：浮点Load/Store
- 保留站
  - 浮点加减法操作有3个保留站：ADD1，ADD2，ADD3
  - 浮点乘除法操作有2个保留站：MULT1，MULT2
  - Load/Store操作有6个LS缓冲器，每个都可看做保留站；
  - 每个保留站都有一个标识字段，唯一地标识了该保留站。

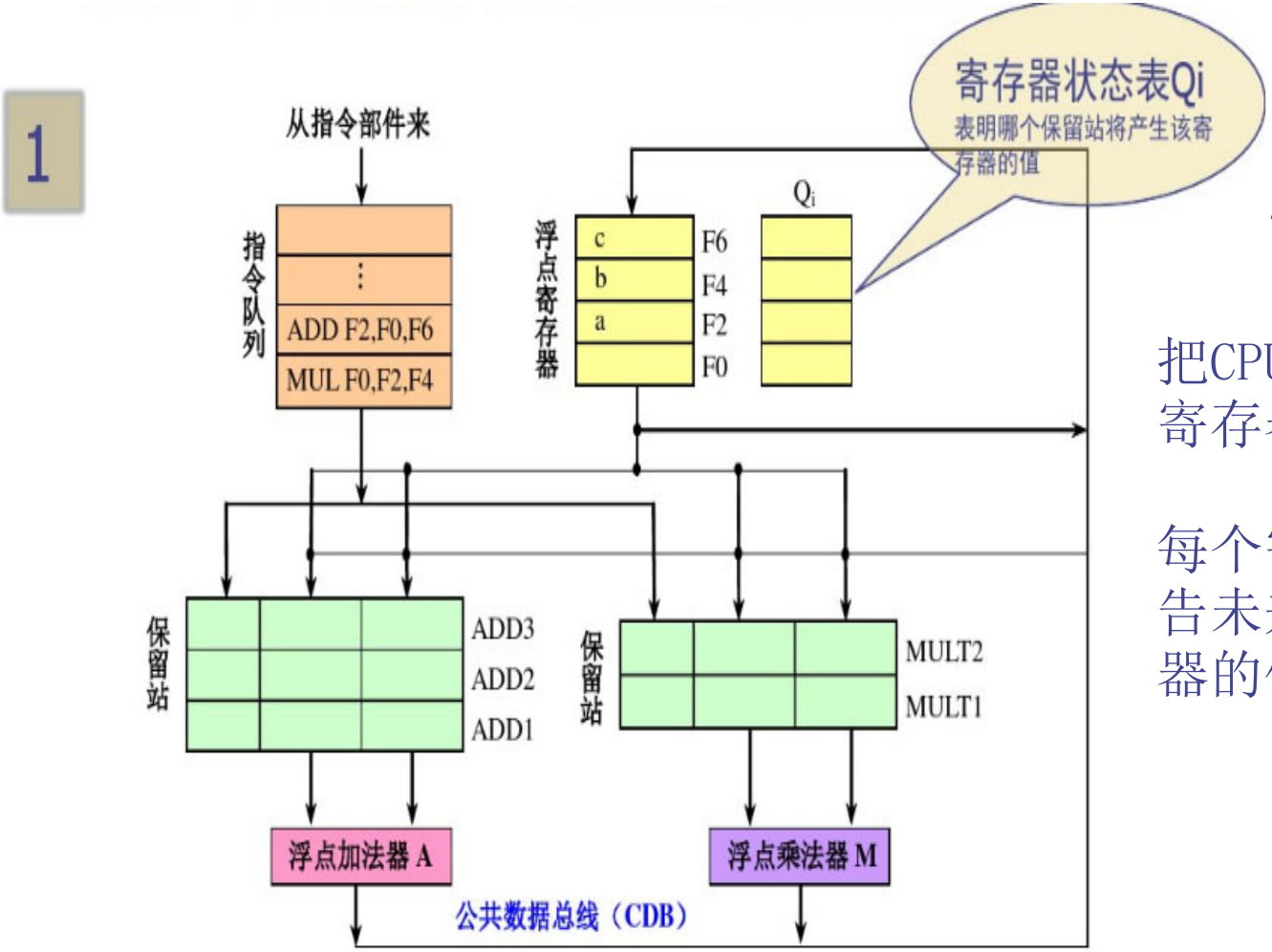
- 公共数据总线CDB（一条重要的数据通路）
  - 所有功能部件的计算/访存结果都是送到CDB上，由它把这些结果直接送到（广播到）各个需要该结果的地方。 CDB连到除了load缓冲器以外所有的部件入口。
  - 在具有多个执行部件且采用多流出（即每个时钟周期流出多条指令）的流水线中，需要采用多条CDB。
- 浮点寄存器FP
  - 共有16个浮点寄存器：F0，F2，F4，…，F30。
  - 它们通过一对控制总线和数据总线连接到功能部件，并通过CDB连接到store缓冲器。
- 指令队列
  - 指令部件送来的指令放入指令队列
  - 指令队列中的指令按先进先出的顺序流出

- LSU部件
  - 内部含有加法器，用于地址计算
- load/store缓冲器
  - 存放读/写存储器的数据或地址
  - load/store缓冲器的作用有5个：
    - 存放用于计算有效地址的分量Imm；
    - 保存正在进行的load访存的目标地址，等待存储器的响应；
    - 保存已经完成了的load的结果（即从存储器取来的数据），等待CDB传输。
    - 保存正进行的store访存的目标地址，等待被存储数据的到达
    - 保存要被存储的数据，直到存储部件接收。

使用Tomasulo算法的流水线需3个步骤，各段中指令执行步骤如下：

- 流出
  - 流出条件：指令进入指令队列的头部，且有对应该指令的空闲保留站，则指令可从指令队列的头部流出到保留站(设为r)。
  - 流出动作：缓冲操作数，完成换名，预约目标寄存器。（IS流水段）
- 执行 （流水中没有待执行的分支指令）
  - 浮点指令执行条件：两个操作数就绪，无论r是否成为保留站的头部，且计算部件就绪（乱序计算）。
  - 浮点指令执行动作：计算浮点操作，产生计算结果（EX流水段）
  - LD/ST指令执行条件：地址操作数就绪，无论r是否成为缓冲器队列的头部。
  - LD/ST指令执行动作：计算有效地址，并将有效地址放入r（EX流水段），LD数据（必须满足当前LD指令为缓冲器队列头部且存储器就绪，保证按序访存，MEM流水段）
- 写结果
  - 浮点或LD指令写结果条件： r执行结束，且CDB就绪
  - 浮点或LD指令写结果动作：将计算结果放到CDB上，送给所有等待该结果的寄存器和保留站（包括store缓冲器），释放产生结果的保留站。（WB流水段）
  - ST指令写结果条件：r执行结束，待存数据就绪，当前ST指令为缓冲器队列头部，存储器就绪
  - ST指令写结果动作：ST数据，释放产生结果的保留站。（MEM流水段）

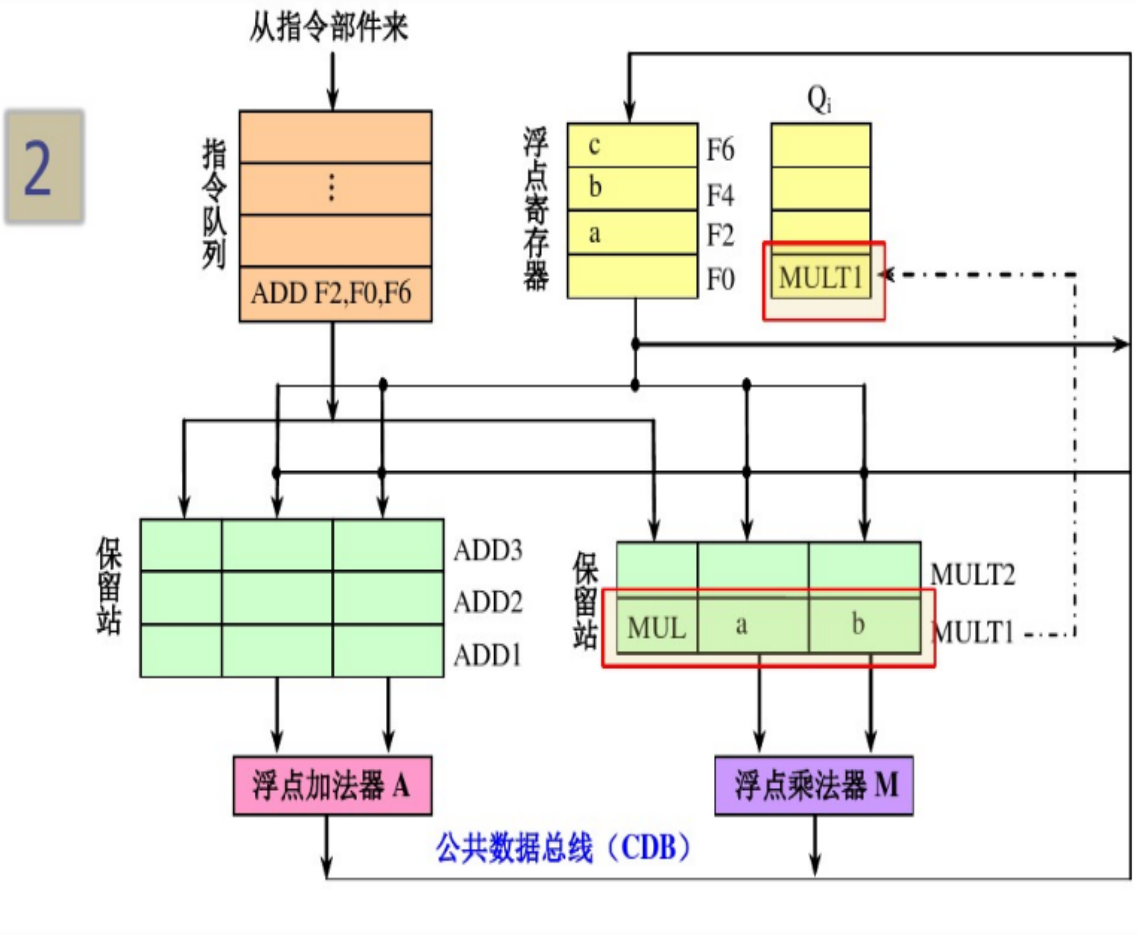
一个简单的例子说明 Tomasulo算法的基本思想。



MUL F0,F2,F4
ADD F2,F0,F6

把CPU简化为图A，并且增加了一个寄存器状态表Q_i。

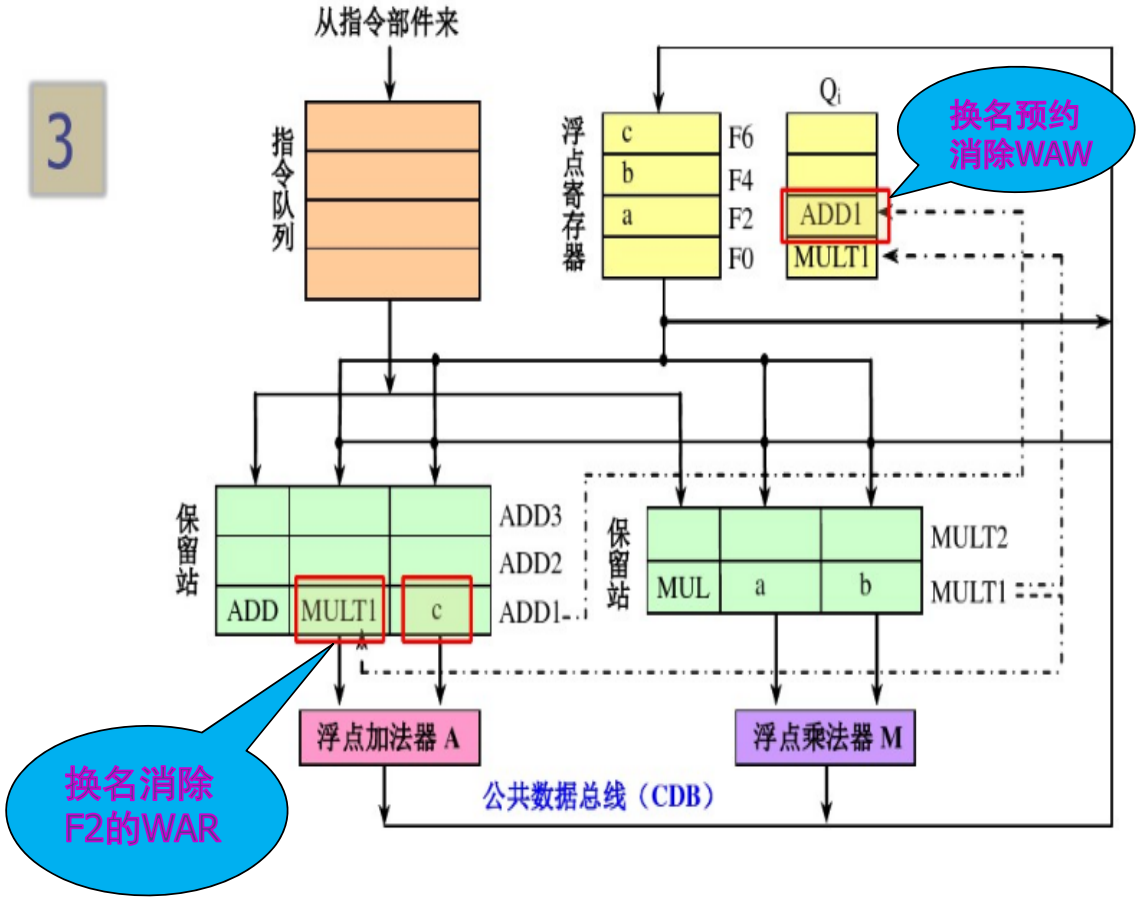
每个寄存器在Q_i中有一项，用于公告未来哪个保留站将生产该寄存器的值。



图(b)

MUL F0,F2,F4
ADD F2,F0,F6

图b，当指令MULF0，F2，F4流出到保留站Mult1时，由于其操作数a和b就绪(在F2和F4中)，就将它们从寄存器取到保留站，这样该指令以后就跟F2和F4没有关系了，执行时直接从保留站中取操作数。
同时，将目的寄存器F0对应的Q_i标志置为Mut1，表示该寄存器的内容将由保留站 Mult1提供，对同一个寄存器的多个预约不断覆盖更新，只会留下最后一个，消除WAW。

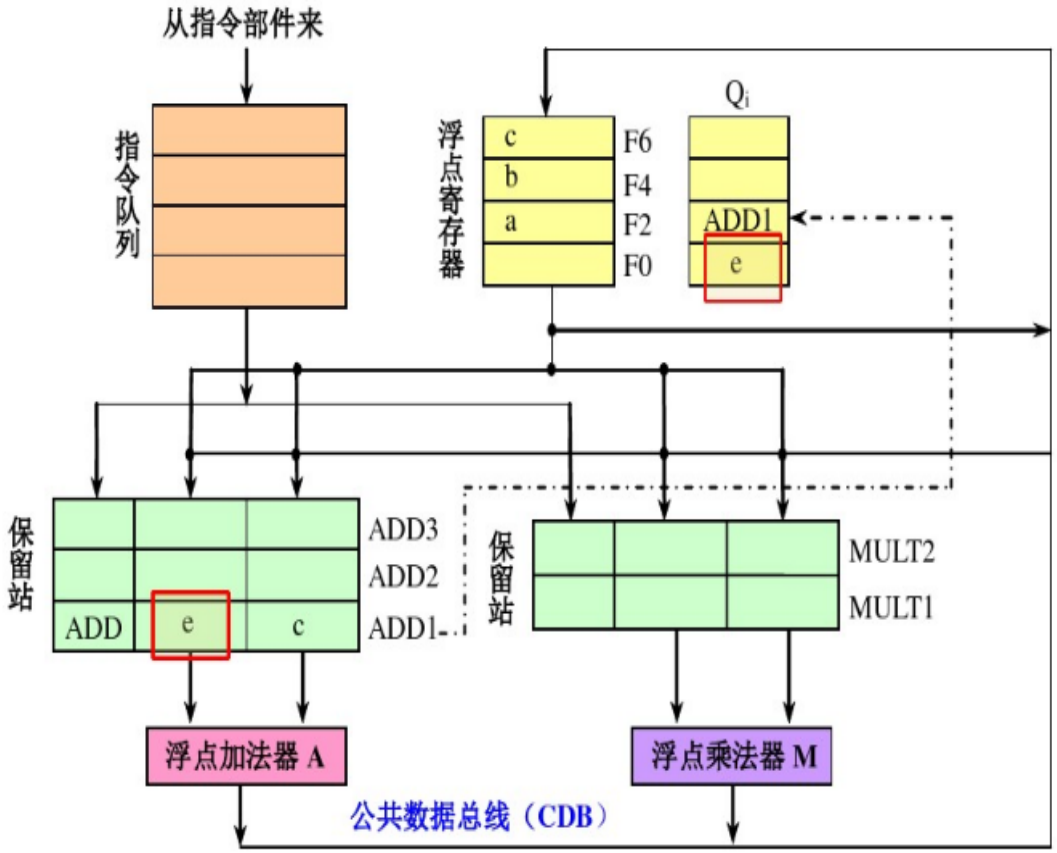


图(c)

MUL F0,F2,F4
ADD F2,F0,F6

图(c)，当指令ADDF2,F0,F6流出到保留站Add1时，也将操作数c取到保留站，但发现F0中的操作数还没有就绪，于是就把其提供者 Mult1的标识取到保留站中。这样就有两个地方在等 Mult1的结果，**其中一个写F2就被替换成写ADD1，与后续指令写F2的WAW被消除了**。同时，它将目的寄存器F2对应的Q_i标志置为Add1表示该寄存器的内容将由保留站Add1提供。**ADD F2,F0,F6变成了ADD F2,Mul,F6，后续指令如果要写F0，就与ADD的读入无关了，消除了WAR**

4



图(d)

MUL F0,F2,F4
ADD F2,F0,F6

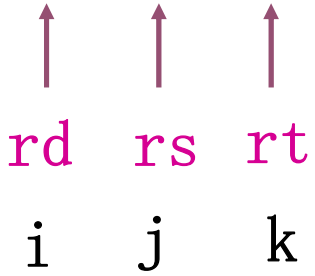
图(d)，当 Mult1的运算结果产生后(设为e)，就把数据放到总线上(广播)，所有等待该数据的地方都会自动把数据取走。Add1中的ADD指令得到该数据后，马上就可以开始执行，不用从寄存器读取，最大程度减少RAW冲突。

Tomasulo算法需要的各信息表

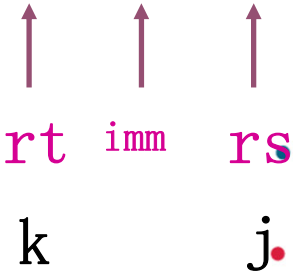
- 指令执行状态表：每条2项={指令字段,状态字段}
- 寄存器状态表：每条2项={寄存器号,保留站预约字段Qi}
- 保留站状态表：每条8项={L,B,O,Vj,Vk,Qj,Qk,A}
  - L：label，保留站标识字段。
  - B：Busy，保留站或缓冲单元是否“忙”。
  - O：Operation，流出指令的操作码。
  - A：地址计算前存Imm，地址计算后存有效地址。
  - Qj，Qk：将要产生源操作数的保留站号。
  - Vj，Vk：源操作数的数值。
  - 注：仅LD和ST缓冲器有A字段，ST缓冲器待存数字段Vk；Qj对应rs，为0表示Vj就绪；Qk对应rt，为0表示Vk就绪；Qi的预约字段为0表示该寄存器中的数据就绪。

符号说明： Qi的i Vj,Vk,Qj,Qk的j,k

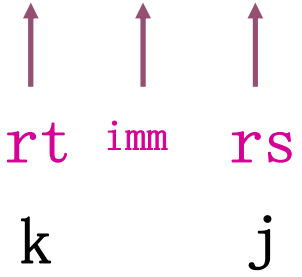
MUL.D F4, F0, F2



L.D F2, 45 (R3)



S.D F3, 40 (R4)



完整的Tomasulo算法（1/5）：浮点指令的流出

```
if （Qi[rs] = 0） { RS[r].Vj = Regs[rs]； RS[r].Qj = 0 }；
// 若第一操作数就绪，取到保留站r的Vj，置Qj为0，表示Vj操作数就绪
else { RS[r].Qj = Qi[rs] }
// 若第一操作数没有就绪，把将产生该操作数的保留站编号放入r的Qj。
if （Qi[rt] = 0） { RS[r].Vk = Regs[rt]； RS[r].Qk = 0 }
// 若第二操作数就绪，取到保留站r的Vk，置Qk为0，表示Vk操作数就绪
Else { RS[r].Qk = Qi[rt] }
 //若第二操作数没有就绪，把将产生该操作数的保留站编号放入r的Qk。
RS[r].Busy = yes；RS[r].Op = Op；//置当前保留站为“忙”并记录操作码
Qi[rd] = r；
//把当前保留站的编号r放入rd对应寄存器状态表项，以便rd将来接收结果。
```

完整的Tomasulo算法（2/5）：LD/ST指令的流出

if （Qi[rs] = 0） {RS[r].Vj = Regs[rs]；RS[r].Qj = 0 }；

//若第一操作数就绪，取到保留站r的Vj，置Qj为0，表示Vj操作数就绪

else {RS[r].Qj = Qi[rs] }

//若第一操作数没有就绪，就把将产生该操作数的保留站编号放入r的Qj。

RS[r].Busy = yes；　// 置当前缓冲器单元为“忙”

RS[r].Op = Op；　//设置操作码

RS[r].A = Imm；　// 把符号位扩展后的偏移量放入当前保留站的A

对于load指令：

Qi[rt] = r；// 把当前缓冲器的编号r放入rt对应寄存器状态表项，以便rt将来接收数据

对于store指令：

if （Qi[rt] = 0） {RS[r].Vk = Regs[rt]；RS[r].Qk = 0 }；

// 若待存数据就绪，从R[rt]取到r的Vk，置Qk为0，表示Vk操作数就绪

else {RS[r].Qk = Qi[rt] }

// 若待存数据没有就绪，把将产生该数据的保留站编号放入r的Qk。

完整的Tomasulo算法（3/5）：指令的执行

对于浮点指令：

Result = RS[r].Vj op RS[r].Vk; //计算浮点操作

对于ST指令：

RS[r].A = RS[r].Vj + RS[r].A; //计算有效地址,EX流水段

对于LD指令：

RS[r].A = RS[r].Vj + RS[r].A; //计算有效地址，EX流水段

Result = Mem[RS[r].A] ; //访存读数据，MEM流水段

完整的Tomasulo算法（4/5）：浮点与LD指令的写结果

```
任意x （if（Qi[x] = r） { Regs[x] = result；Qi[x] = 0 }；
// 对于任何正在等该结果的浮点寄存器x，
    向x写入结果，把x的预约状态置为数据就绪
任意x （if（RS[x].Qj = r） {RS[x].Vj = result；RS[x].Qj = 0 }；
// 对于任何正在等该结果作为第一操作数的保留站x，
    向x的Vj写入结果，置Qj为0，表示x的Vj操作数就绪
任意x （if（RS[x].Qk = r） {RS[x].Vk = result；RS[x].Qk = 0 }；
// 对于任何正在等该结果作为第二操作数的保留站x，
    向x的Vk写入结果，置Qk为0，表示x的Vk操作数就绪
RS[r].Busy = no；  // 释放当前保留站，将之置为空闲状态。
```

完整的Tomasulo算法（5/5）：ST指令的写结果

Mem[RS[r].A] = RS[r].Vk

// 数据写入存储器，地址由store缓冲器单元的A字段给出。MEM流水段

RS[r].Busy = no;

//释放当前缓冲器单元，将之置为空闲状态。

算法结束，例题

例 单流出处理器采用基于Tomasulo算法进行指令调度。有一个LSU部件（内部自带加法器），2个LS缓冲器（私有时间戳实现FIFO），1个加法ALU部件，1个乘法ALU部件，2个ALU保留站。指令序列执行前，指令均未流出，所有缓冲器/保留站均空闲。

（1）各个硬件操作流水段及指令通过的时钟周期如下表

Issue	WtCDB	Mem	Execute				
			LD	ST	SUB	ADD	MUL
1	1	3	1	1	4	4	10

（2）待执行指令序列如下：

指令	对应变量
LD R2, (R1)	Rt=R2, Rs=R1, Imm=0
MUL R2, R2, #2	Rd=R2, Rs=R2, Rt= #2
ST R2, (R1)	Rt=R2, Rs=R1, Imm=0
SUB R1, R1, #4	Rd=R1, Rs=R1, Rt= #4
ADD R5, R3, R4	Rd=R5, Rs=R3, Rt=R4

问：

（1）请给出指令执行状态时钟周期表

（2）请给出第14周期末尾各个状态表内容

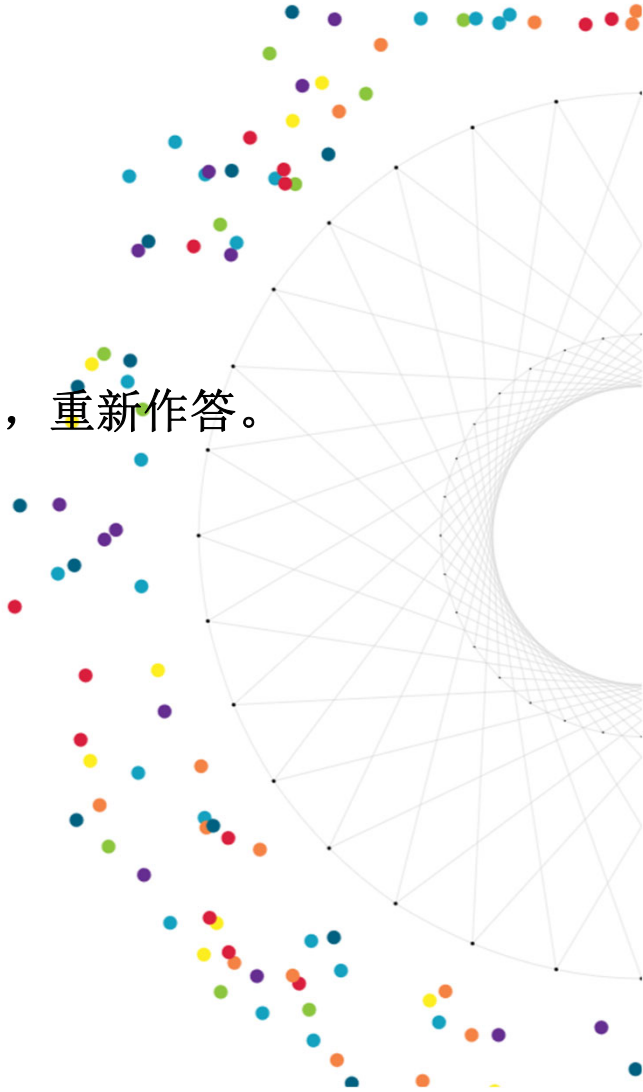
指令	IS	EX	MEM	WtCDB	备注	14末
LD R2, (R1)	1	2	3-5	6		已写结果
MUL R2, R2, #2	2	7-16	Null	17	数据相关	已执行
ST R2, (R1)	3	4	18-20	Null	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	R1已换名	已写结果
ADD R5, R3, R4	10	11-14	Null	15	保留站冲突	已执行

Label	Busy	Op	Vj(rs)	Vk(rt)	Qj(rs)	Qk(rt)	A(Imm)
Load1	N						
Load2	Y	ST	Reg[R1]			ALU1	Reg[R1]
ALU1	Y	MUL	Reg[R2]	#2			
ALU2	Y	SUB->ADD	Reg[R3]	Reg[R4]			

	R1	R2	R3	R4	R5
Qi		ALU1			ALU2

习题 3.1 上述例题的MEM段由3个cycles变成5个cycles，重新作答。

要求：非手写答案零分。



Tomasulo算法总结

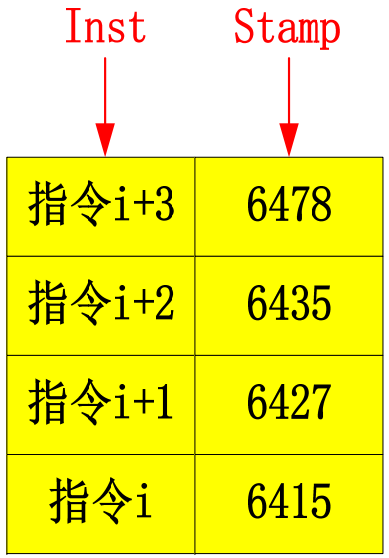
- 寄存器换名是通过保留站和流出逻辑来共同完成的。指令流出到保留站后，其操作数寄存器号要么换成了数据本身，要么换成了生产者保留站的标识，不再与寄存器有关系，消除了名相关导致的冲突。
- 指令执行控制是分布的。每个功能部件的保留站中的信息决定了什么时候指令可以在该功能部件开始执行，乱序计算，但**真数据相关必须保持，如果操作数没有准备好，指令不能执行，只能等待**。
- CDB是一对多的定向技术，计算结果通过CDB直接从生产它的保留站传送到所有需要它的功能部件，使得等待这一运算结果的多个指令可以同时获得而不用等待访问寄存器，最大限度减少了真数据相关导致的停顿。

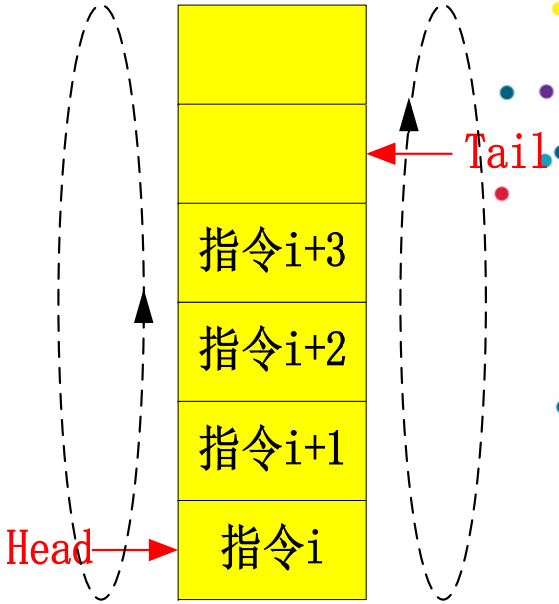
Tomasulo算法总结

- 指令的流出是顺序的，指令队列是FIFO队列，后继指令从尾部进入，前驱指令从头部取出；
- 存储器的访问是顺序的，LS保留站是FIFO队列，后继指令从尾部进入，前驱指令从头部取出；
- 保留站和执行部件的数量是有限的，如果部件冲突，指令不能进入下一个步骤，只能等待。
- 执行步骤中，Load指令经过EX段和MEM段，Store指令/ALU指令只经过EX段；
- 写回步骤中，Store指令经过WB段和MEM段，Load指令/ALU指令只经过WB段。

Tomasulo算法总结

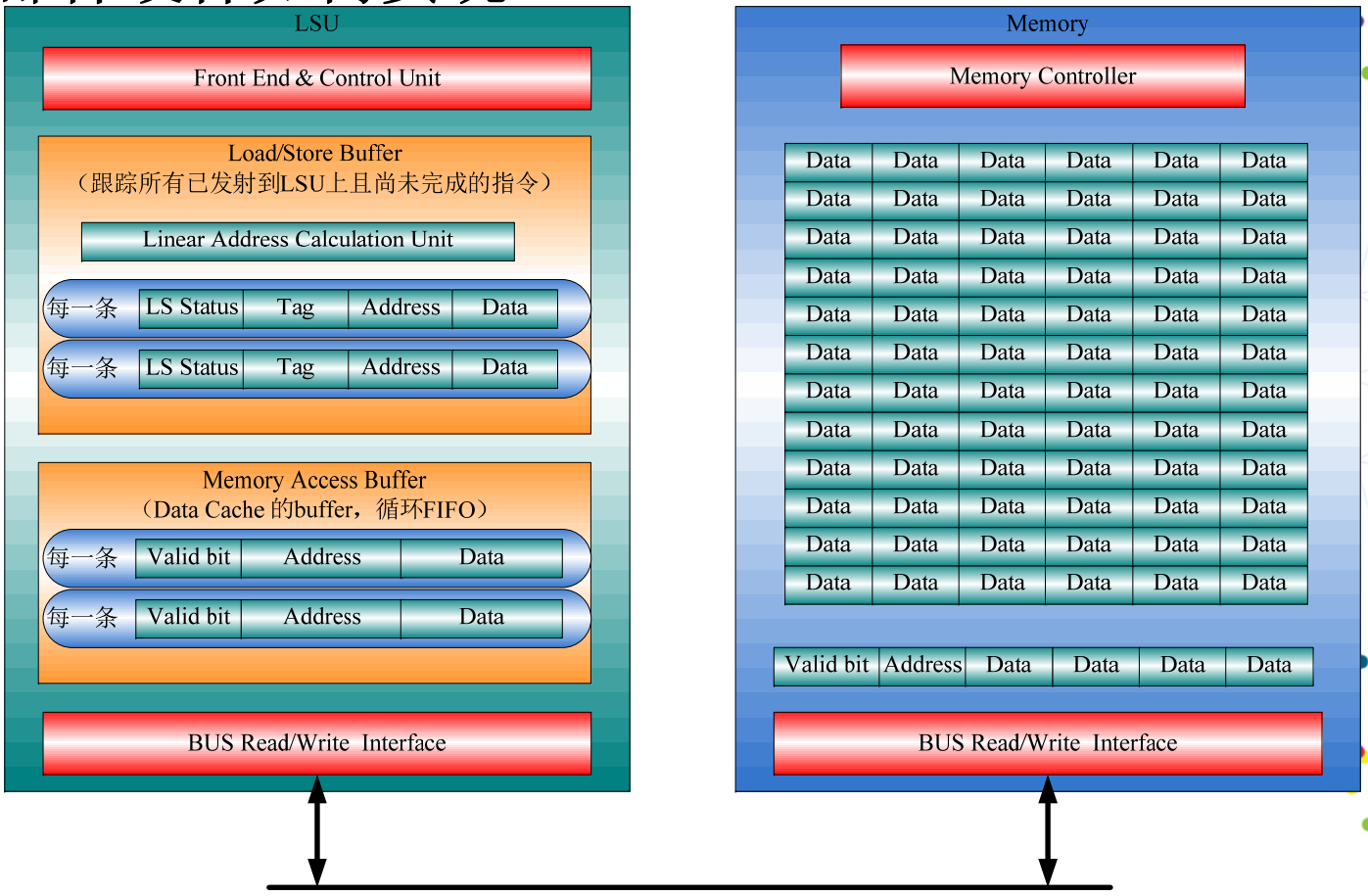
- FIFO队列硬件如何实现？
- 考虑到队列内指令批量前移和名称变换的复杂性，可以用私有时间戳或者循环head指针。





Tomasulo算法总结

- LSU部件硬件如何实现？

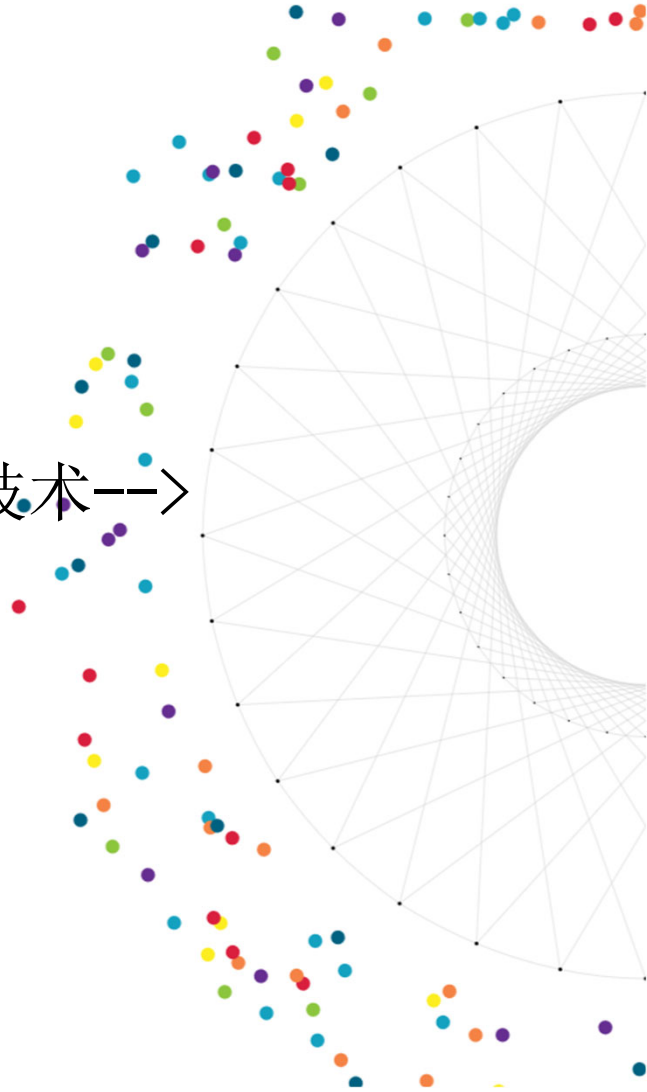


动态解决控制冲突的技术

结构冲突

数据冲突

控制冲突-->解决控制冲突-->静态分支预测技术-->动态分支预测技术



什么是动态分支预测

动态分支预测技术：

通过硬件技术，在程序执行时根据每一条转移指令过去的转移历史记录来预测下一次转移的方向。通过提前预测分支方向，减少或消除控制相关导致的流水线停顿。

优点：

- 根据程序的执行过程动态地改变转移的预测方向，因此有更好的准确度和适应性。
- 程序每次执行时，可能预测的分支方向与前次相同或不同。

从简单到复杂的动态转移预测技术如下：

预测分支执行是否成功？分支预测缓存器（BHT）

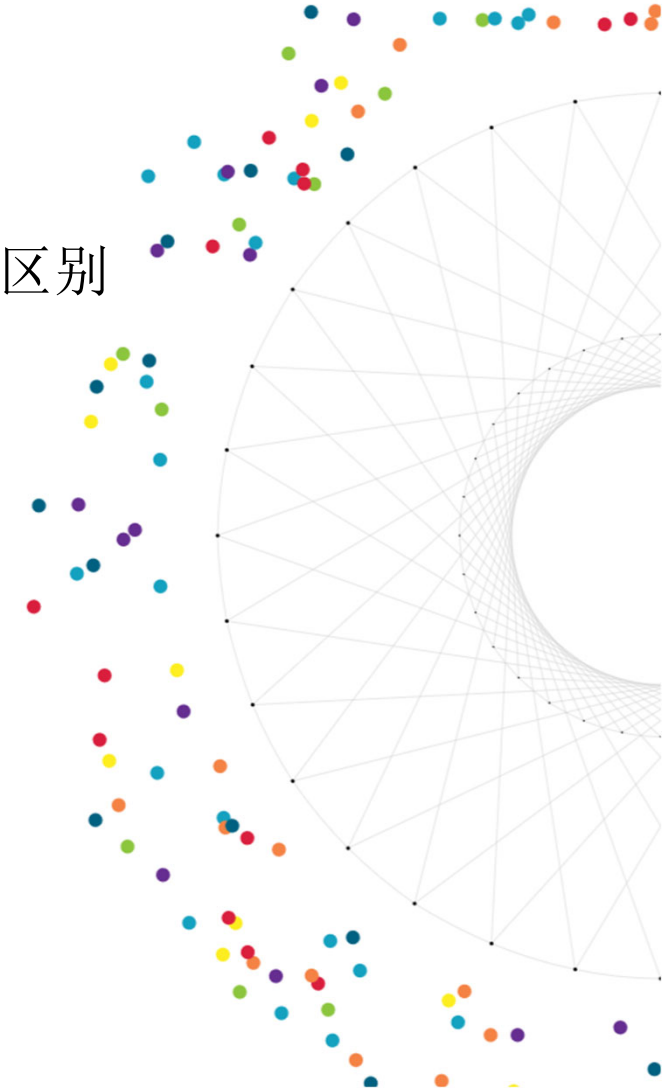
预测分支后继的指令？分支目标缓冲器（BTB）

分支预测+乱序执行=？ 基于硬件的前瞻执行(ROB)

性能不断提高，代价是控制电路的复杂性不断提高。

要点：

- 动态分支预测技术和静态分支预测技术的区别
- 什么叫“分支指令过去的表现”？
- 如何衡量分支预测的有效性？
- 分支预测的目的？
- 需要解决的关键问题？



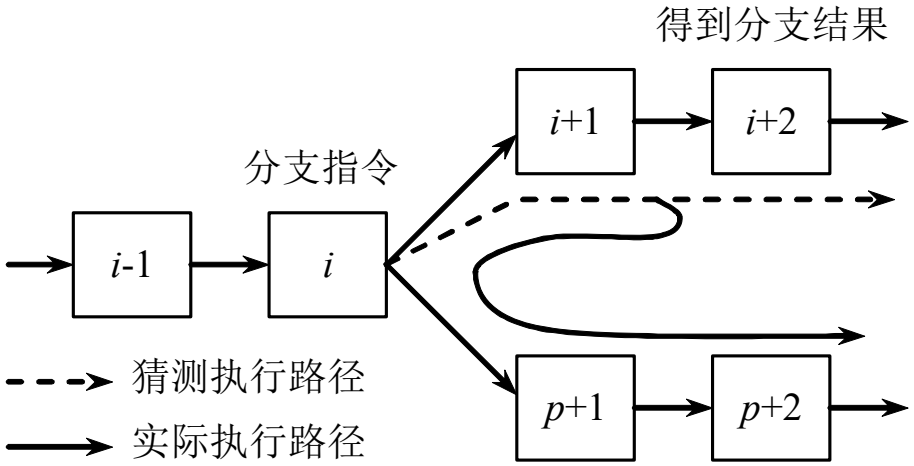
1. 动态分支预测技术和静态分支预测技术的区别

静态分支预测技术所进行的操作事先预定好的 ，与分支的实际执行情况无关；

动态分支预测技术的方法在程序运行时根据分支执行过去的表现预测其将来的行为（如果分支行为发生了变化，预测结果也跟着改变，此外有更好的预测准确度和适应性）。

2. 什么是“分支指令过去的表现”？

就是记录分支的历史信息，在预测错误时，要作废已经预取和分析的指令，恢复现场，为了恢复现场，需要在执行预测的目标指令之前将现场保存起来。



3. 分支预测的有效性取决于：

预测的准确性

预测正确和不正确两种情况下的分支开销

决定分支开销的因素：

流水线的结构（5段流水，分支处理有MEM段提到ID）

预测的方法

预测错误时的恢复策略等

4. 采用动态分支预测技术的目的

预测分支是否成功

尽快找到分支目标地址（或指令）

（避免控制相关造成流水线停顿）

5. 需要解决的关键问题

如何记录分支的历史信息；

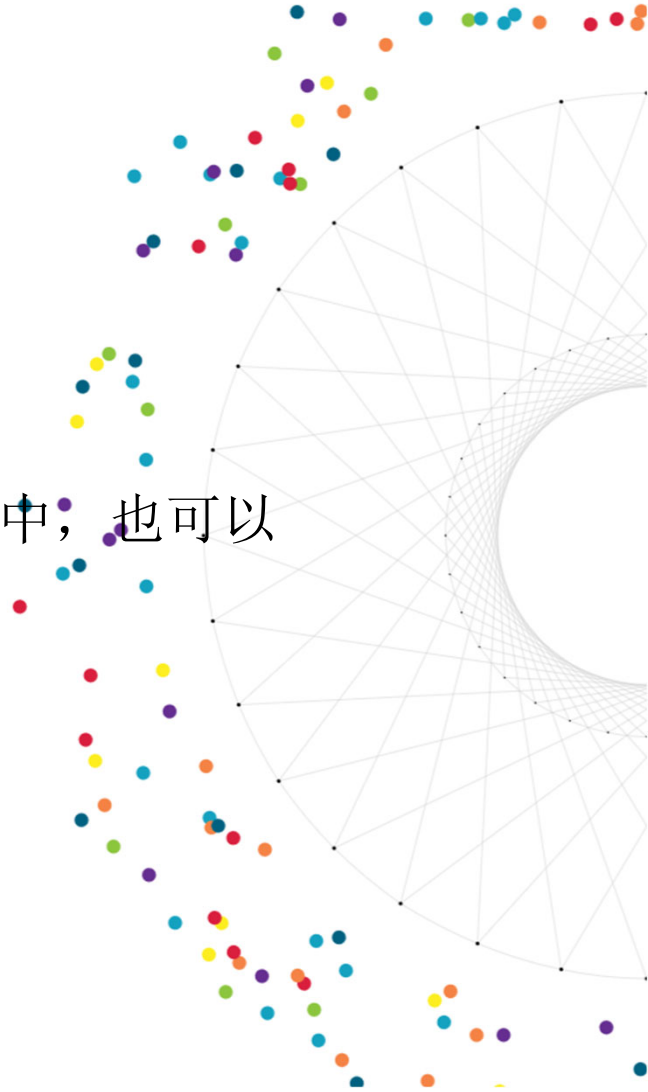
如何根据这些分支的历史信息来预测分支的去向（甚至取到指令）。

采用分支历史表 BHT

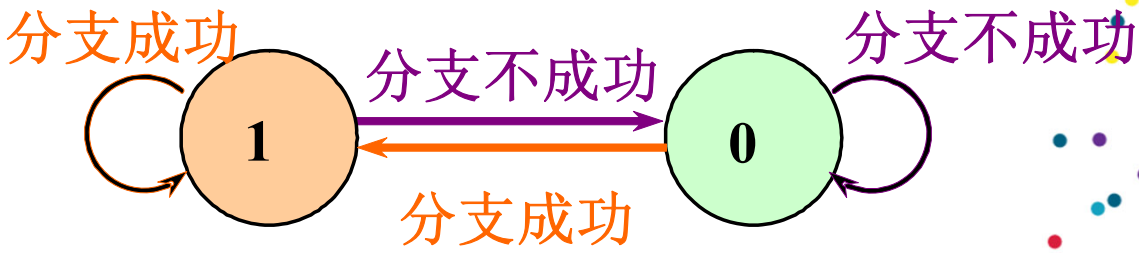
- 分支历史表BHT（Branch History Table）或分支预测缓冲器（Branch Prediciton Buffer）
  - 最简单的动态分支预测方法。
  - 用BHT来记录分支指令最近一次或几次的执行情况（成功或不成功），并据此进行预测。

只有1个预测位的分支预测缓冲

- BHT只需要1位
- BHT可以跟分支指令一起存放在指令Cache中；也可以用一个专门的硬件来实现。



1位分支预测的状态转换



分支预测：
成功

分支预测：
不成功

例题 一个循环体N共循环10次（分支成功9次，失败1次）。假设使用1位分支预测，那么分支预测的准确性是多少？

解：这种预测方式将会在第一次和最后一次循环中出现预测错误：

（1）循环体N最后一次预测错误是不可避免的，因为最后一次之前所有的9次循环中，分支都是成功的。

（2）第一次预测错误，是源于前面执行的另一个循环体M，M的最后一次循环导致预测位被置0，该预测值保持到了循环体N的第一次预测。

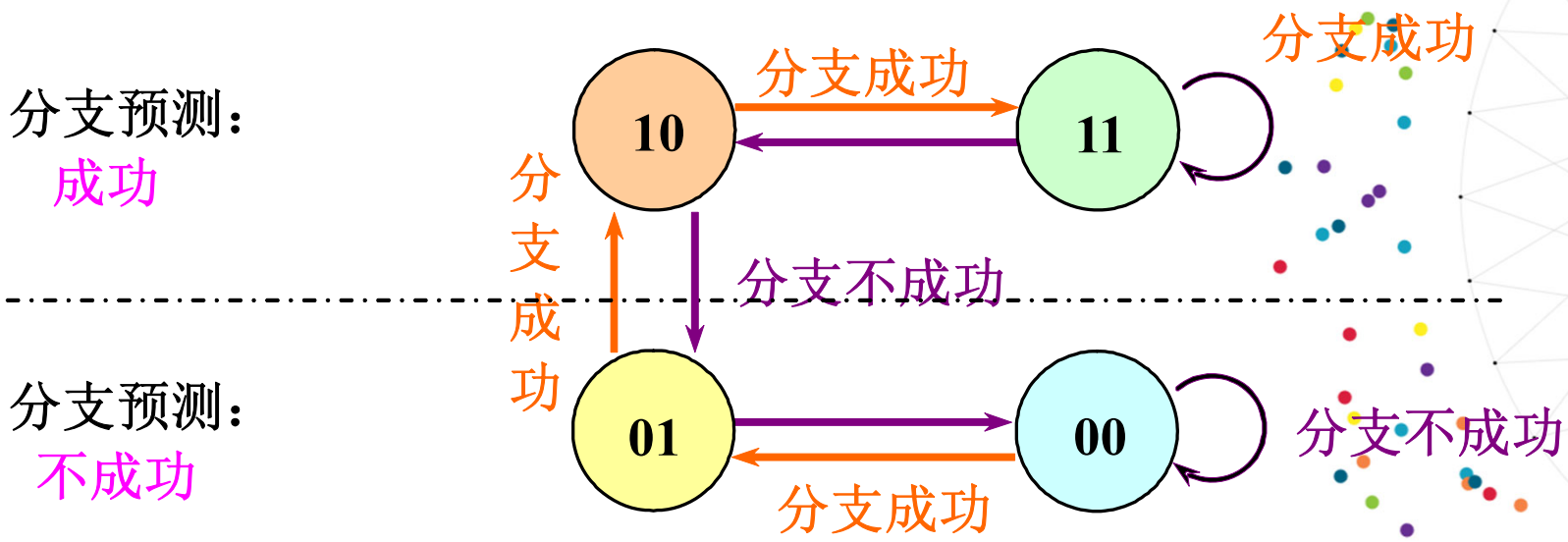
（3）因此，尽管分支成功的比例率是90%，但分支预测的准确性为101%（两次不正确，8次正确）。

解毕。

1位分支预测机制的缺点与改进

- 缺点：只要预测出错，往往是连续两次预测错误，而不是一次。
- 改进：采用2个预测位。
  - 提高预测的准确度
  - 研究结果表明：两位分支预测的性能与n位（n>2）分支预测的性能差不多。

BHT采用2个二进制预测位



例题 一个循环体N共循环10次（分支成功9次，失败1次）。
假设使用2位分支预测，那么分支预测的准确性是多少？

解：这种预测方式只会在最后一次循环中出现预测错误：
（1）循环体N最后一次预测错误是不可避免的，因为最后一次之前所有的9次循环中，分支都是成功的。最后一次循环中预测错误，导致预测位由11变为10.
（2）前面执行的另一个循环体M，M的最后一次循环导致预测位由11变为10 ，该预测值保持到了循环体N的第一次预测，10导致此次预测正确。
（3）因此，分支成功的比例率是90%，分支预测的准确性为90%（1次不正确，9次正确）。
解毕。

两位分支预测中的操作有两个步骤：

- 分支预测。
  - 当分支指令到达译码段（ID）时，根据从BHT读出的信息进行分支预测 。
  - 若预测正确，就继续处理后续的指令，流水线没有断流。否则，就要作废已经预取和分析的指令，恢复现场，并从另一条分支路径重新取指令。
- 状态修改。

BHT方法只在以下情况下才有用：

- 判定分支是否成功所需的时间大于确定分支目标地址所需的时间。

5段经典流水线：由于判定分支是否成功和计算分支目标地址都是在ID段完成，所以BHT方法不会给该流水线带来好处。

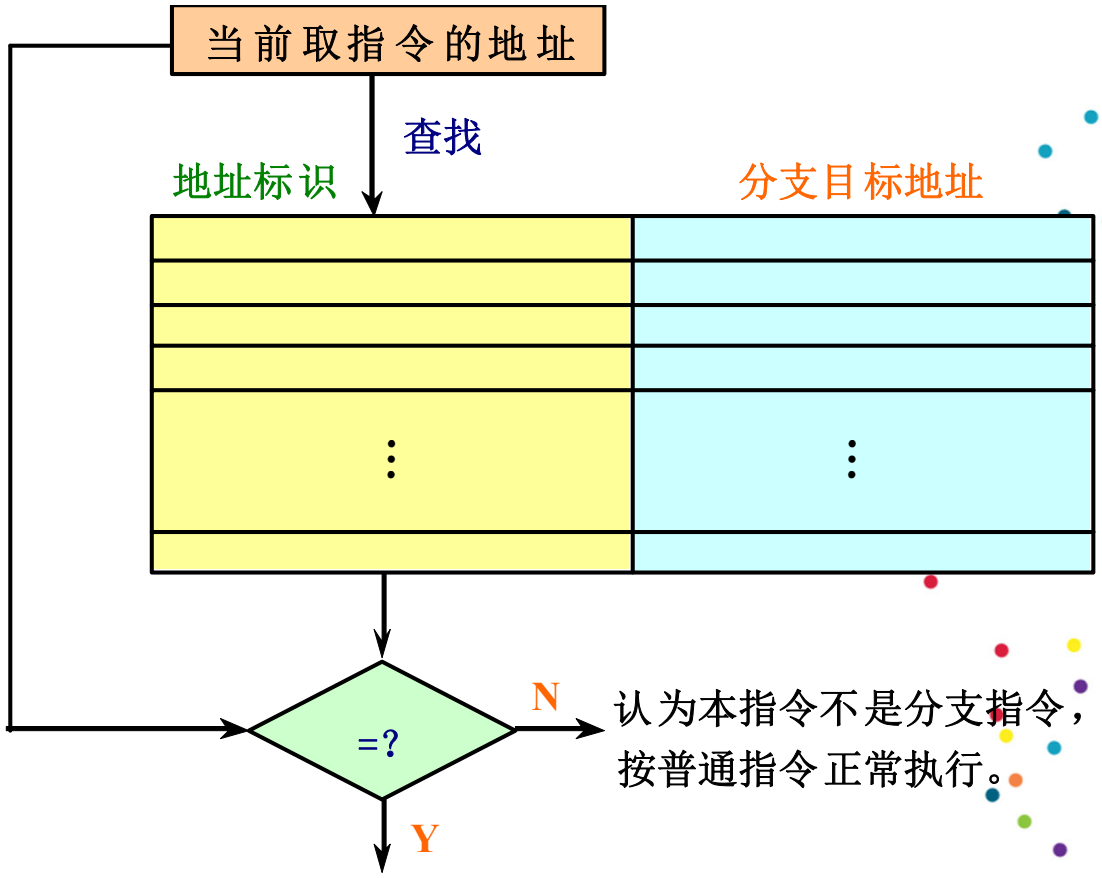
采用分支目标缓冲器BTB

目标：将分支的开销降为0

方法：分支目标缓冲

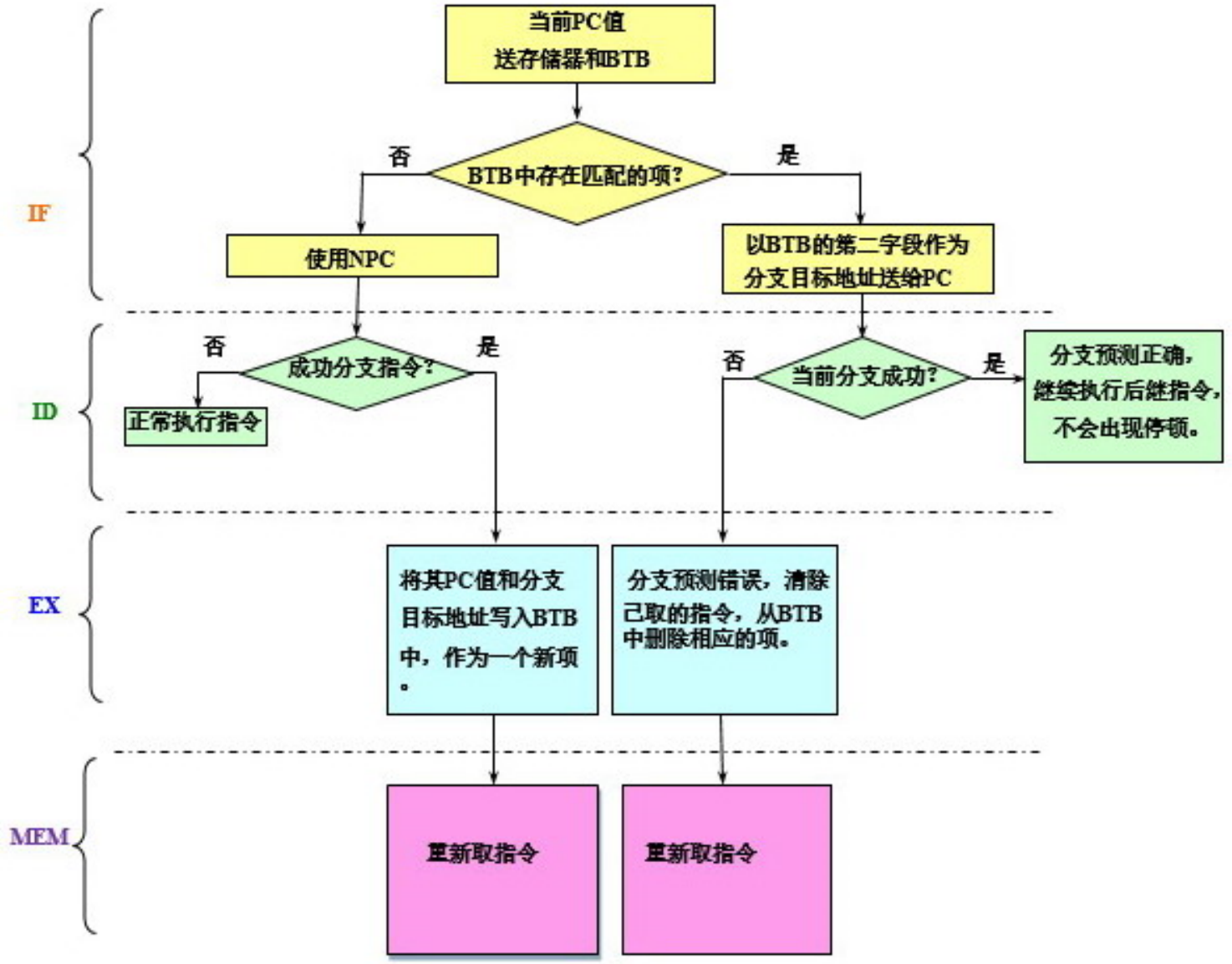
- 将分支成功的分支指令的地址和它的分支目标地址都放到一个缓冲区中保存起来，缓冲区以分支指令的地址作为标识。
- 这个缓冲区就是分支目标缓冲器（Branch-Target Buffer，简记为BTB，或者Branch-Target Cache）。

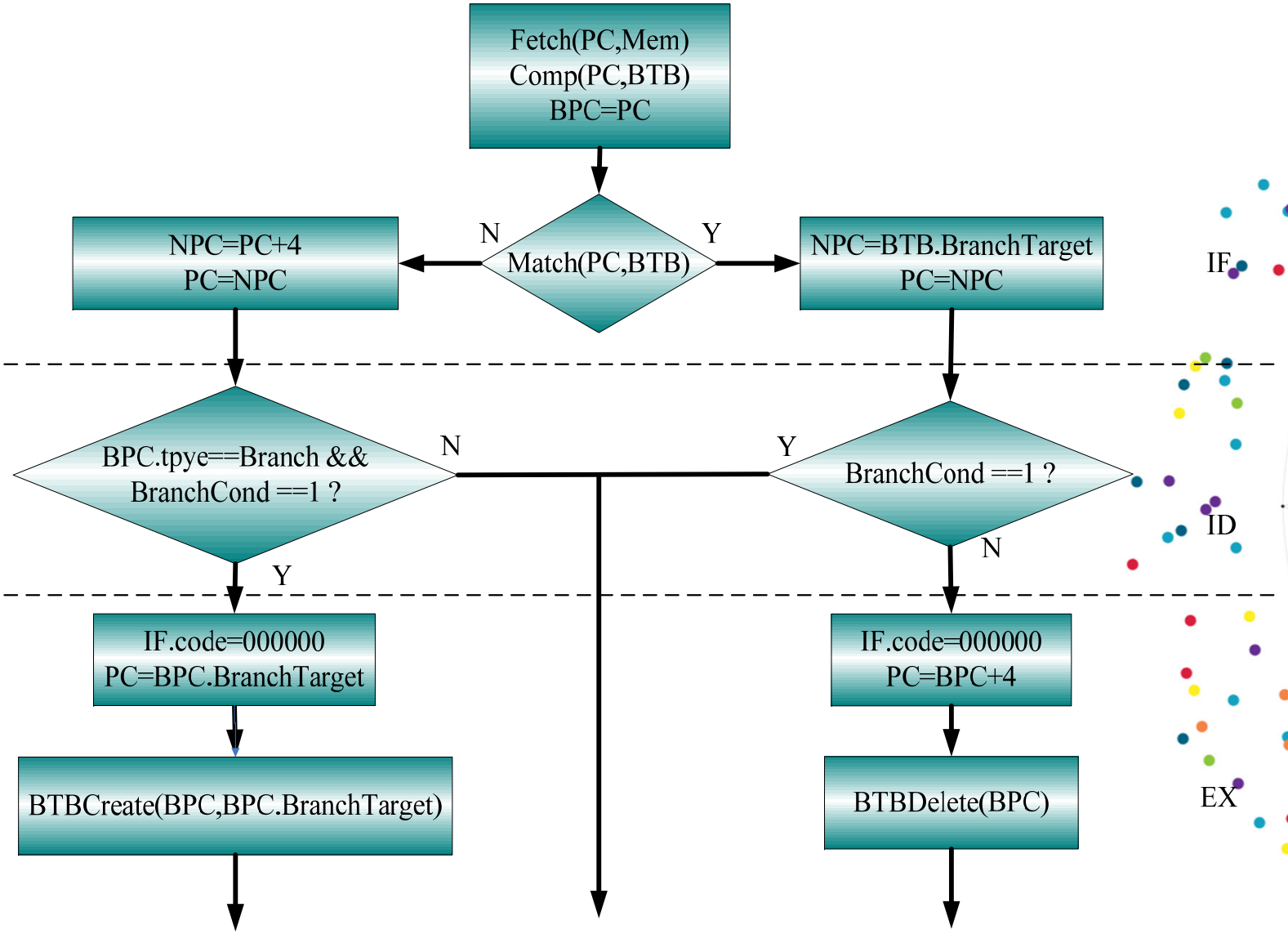
BTB的结构


当前取指令的地址
查找
地址标识
分支目标地址
=?
N
认为本指令不是分支指令，
按普通指令正常执行。
Y
认为该指令是成功的分支指令，用
分支目标地址作为下一条指令的 PC 值。

- 看成是用专门的硬件实现的一张表格。
- 表格中的每一项至少有两个字段：
  - 执行过的成功分支指令的地址；
    （作为该表的匹配标识）
  - 预测的分支目标地址。

- 采用BTB后，在流水线各个阶段所进行的相关操作：


当前PC值
送存储器和BTB
否
BTB中存在匹配的项?
是
IF
使用NPC
以BTB的第二字段作为
分支目标地址送给PC
否
成功分支指令?
是
否
当前分支成功?
是
分支预测正确，
继续执行后继指令，
不会出现停顿。
ID
正常执行指令
EX
将其PC值和分支
目标地址写入BTB
中，作为一个新项
。
分支预测错误，清除
已取的指令，从BTB
中删除相应的项。
MEM
重新取指令
重新取指令

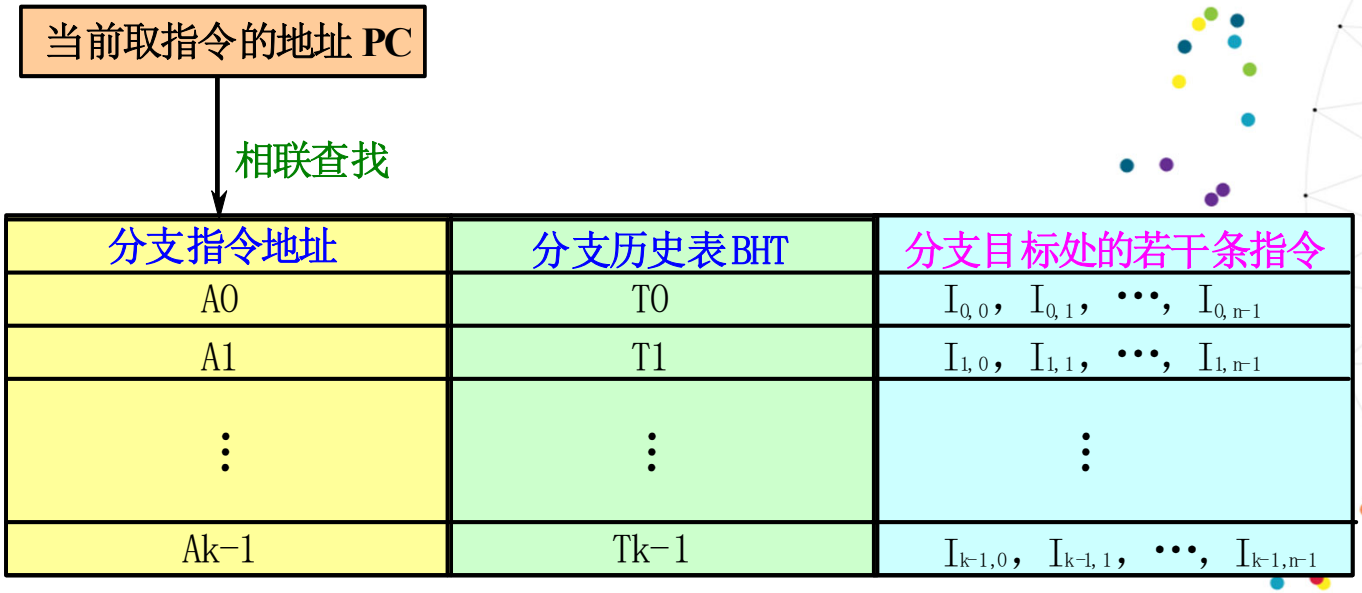

Fetch(PC,Mem)
Comp(PC,BTB)
BPC=PC
Match(PC,BTB)
N
Y
NPC=PC+4
PC=NPC
NPC=BTB.BranchTarget
PC=NPC
IF
BPC.tpye==Branch &&
BranchCond ==1 ?
N
Y
BranchCond ==1 ?
ID
Y
N
IF.code=000000
PC=BPC.BranchTarget
IF.code=000000
PC=BPC+4
BTBCreate(BPC,BPC.BranchTarget)
BTBDelete(BPC)
EX

采用BTB后，各种可能情况下的延迟 ：

指令在BTB中？	预测	实际情况	延迟周期
是	成功	成功	0
是	成功	不成功	2
不是		成功	2
不是		不成功	0

BTB的另一种形式

BTB不存地址了，存指令，一条或者多条分支目标处的指令。



BTB的另一种形式

有三个潜在的好处：

- 更快地获得分支目标处的指令。节省出来的时间可以消耗在BTB的检索中，这样就能够支持更大的BTB。
- 可以一次提供分支目标处的多条指令，这对于多流出处理器是很有必要的；
- 可以支持分支折叠技术（branch folding）：流水线正常处理分支指令导致后继指令等待的时间内，可以从BTB中取出指令送入流水线占据气泡，使得分支路径上的指令串和非分支指令上的指令串同时被执行。

  （硬件版分支延迟槽）

BTB v.s. BPB

	BTB	BPB
缓冲内容	分支成功路径上的 后继指令地址或后继指令	分支是否成功的预测值
分支预测	预测分支成功	预测值动态改变
作用位置	IF段	ID段

指令前瞻执行 IS

前瞻执行（speculation）的基本思想：

对分支指令的结果进行猜测，并假设这个猜测总是对的，然后按这个猜测结果继续取、流出和执行后续的指令。只是执行指令的结果不是写回到寄存器或存储器，而是放到一个称为ROB（ReOrder Buffer）的缓冲器中。等到相应的指令得到“确认”（commit）（即确实是应该执行的）之后，才将结果写入寄存器或存储器。如果分支预测错误，清空缓冲区；如果产生异常，清空缓冲区。

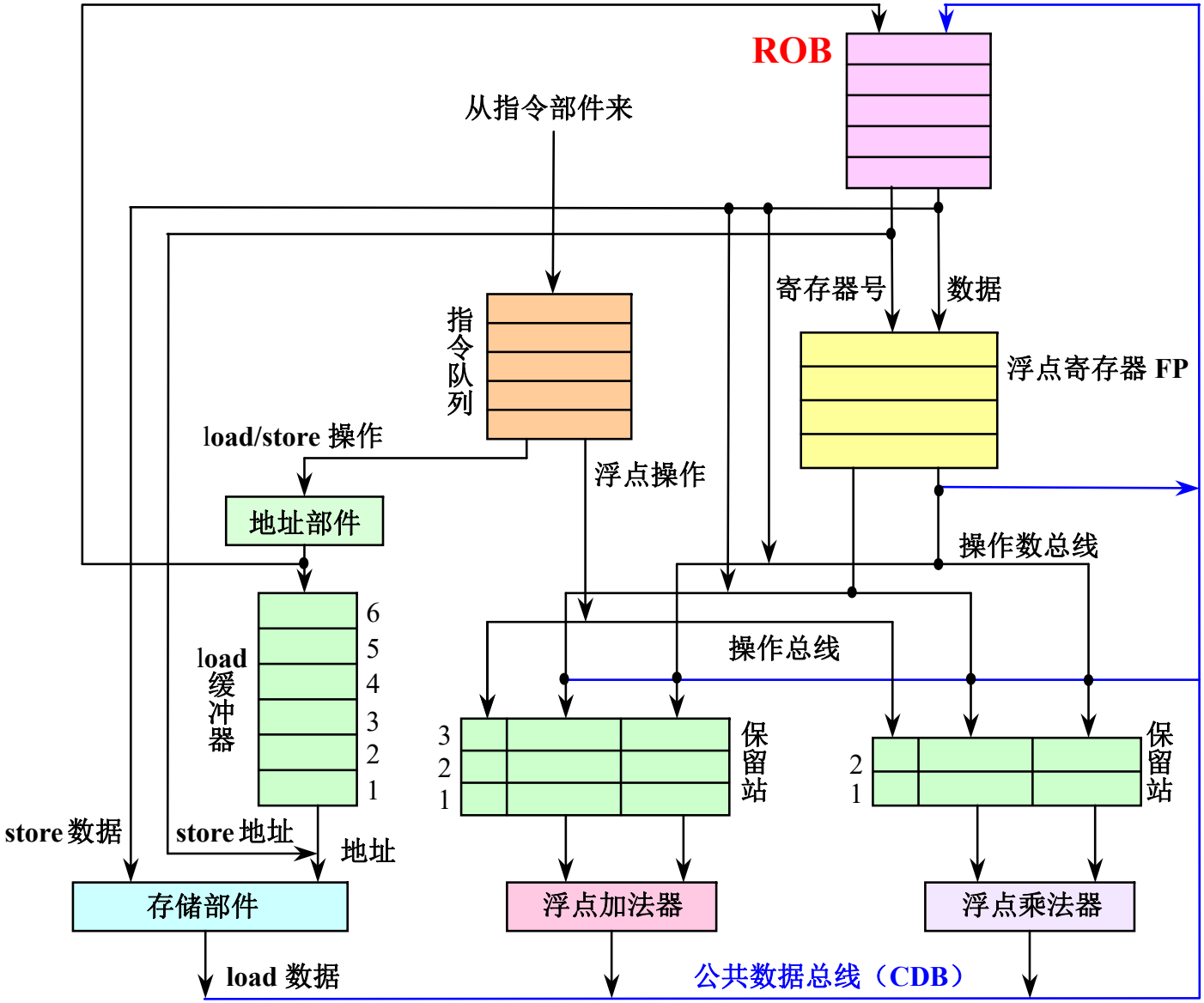
- 基于硬件的前瞻执行结合了三种技术：
  - 动态分支预测。用来选择后续执行的指令。
  - 在控制相关的结果尚未出来之前，前瞻地执行后续指令。
  - 用动态调度对基本块的各种组合进行跨基本块的调度。
- 对Tomasulo算法加以扩充，就可以支持前瞻执行。

  加入再定序缓冲器ROB，把Tomasulo算法的写结果和指令完成加以区分，分成两个不同的段：写结果，指令确认
- 实现前瞻的关键原理：

  允许指令乱序执行，但必须顺序确认。不被确认的指令不能改变机器状态和引发异常。

IS的特点

- 在大多数指令前瞻正确的前提下，前瞻就可以有效地加快分支处理的速度。以大幅度提高硬件复杂性为代价。
- 动态地根据数据相关性来选择指令和指令的执行时间。其实质是数据流驱动运行（data-flow execution），只要操作数有效，指令就可以执行。
- 前瞻执行的指令产生的结果要一直到指令处于非前瞻执行状态时，才能被确认。不被确认的指令不能改变机器状态和引发异常。能够实现精确异常。
- Tomasulo算法中保留站的换名功能是由ROB来完成的。
- Intel Pentium 2/3/4， Alpha 21264，SW11350，SW213510，AMD K5/K6/Athlon，MIPS R10000/R12000，PowerPC 1353/1354/G3/G4



注：

LD缓冲器和ST缓冲器，甚至所有保留站可以都合并到ROB中，原理相同。

IS机制需要的各信息表（1/2）

- 指令执行状态表：每条2项={指令字段,状态字段}
- 寄存器状态表：每条3项={寄存器号,ROB预约字段,Busy}
- 保留站状态表：每条8项={L,B,O,Vj,Vk,Qj,Qk,A,Dest}
  - L：label，保留站标识字段。
  - B：Busy，保留站或缓冲单元是否“忙”。
  - O：Operation，流出指令的操作码。
  - A：地址计算前存Imm，地址计算后存有效地址。
  - Qj，Qk：将要产生源操作数的ROB号。
  - Vj，Vk：源操作数的数值。
  - Dest：指明哪个ROB项b将接收该保留站产生的结果

IS机制需要的各信息表（2/2）

- **ROB状态表：每条6项={L, B, O, Target, Value, State}**
  - L：label，ROB标识字段。
  - B：Busy，保留站或缓冲单元是否“忙”。
  - O：Operation，流出指令的操作码
  - T: Target，目的寄存器标识
    - 分支指令：Target为pc值，value为分支条件码0/1
    - RR-ALU指令：Target = R[rd]
    - RI-ALU指令 & Load指令：Target = R[rt]
    - Store指令：Target=MemoryAddress
  - Value：保存前瞻执行结果，直到指令得到确认
  - State：标识指令是否已完成且数据已就绪
    - 已流出/已执行/已写结果/已确认

IS机制的指令执行步骤（1/4）

- **IS流出** (分支指令流出到加法保留站)
  - 流出条件：指令进入指令队列的头部，且有对应该指令的空闲保留站r，且有空闲的ROB项b，则指令可从指令队列的头部流出到保留站r和ROB项b。
  - 流出动作：缓冲操作数，完成换名（记录将会产生该操作数的ROB项标识），预约目标寄存器。（IS流水段）
  - If Regs[rs].B=N, r.Vj= Regs[rs].
  - If (Regs[rs].B==Y&&Regs[rs].rob.Value!=Null); r.Vj= Regs[rs].rob.Value;
  - If (Regs[rs].B=Y&&Regs[rs].rob.Value==Null); r.Qj= Regs[rs].rob;
- **Tomasulo流出(比较)**
  - 流出条件：指令进入指令队列的头部，且有对应该指令的空闲保留站r，则指令可从指令队列的头部流出到保留站r。
  - 流出动作：缓冲操作数，完成换名（记录将会产生该操作数的保留站标识），预约目标寄存器。分支指令的流出不操作任何信息表。

IS机制的指令执行步骤（2/4）

- IS执行 (分支指令计算有效地址和分支条件)
  - 浮点指令执行条件：两个操作数就绪，无论r是否为保留站的头部，且计算部件空闲。
  - 浮点指令执行动作：计算浮点操作，产生计算结果（EX流水段）
  - LD/ST指令执行条件：地址操作数就绪，无论r是否成为缓冲器队列的头部。
  - LD/ST指令执行动作：计算有效地址，并将有效地址放入r，LD数据
  - LD经过EX和MEM段，ST只经过EX段
- Tomasulo执行(比较)
  - 浮点指令执行条件：两个操作数就绪，无论r是否为保留站的头部，且计算部件空闲。
  - LD/ST指令执行条件：地址操作数就绪，无论r是否成为缓冲器队列的头部。
  - 浮点指令执行动作：计算浮点操作，产生计算结果
  - LD/ST指令执行动作：计算有效地址，并将有效地址放入r，LD数据（...）

IS机制的指令执行步骤（3/4）

- IS写结果 （分支指令将有效地址和分支条件码通过CDB写入ROB）
  - 浮点或LD指令写结果条件： r执行结束，且CDB就绪
  - 浮点或LD指令写结果动作：将计算结果[b][Data]放到CDB上，送给所有等待该结果的ROB项和保留站，释放产生结果的保留站。（WB段）
  - ST指令写结果条件：r执行结束，待存数据就绪，当前ST指令为缓冲器队列头部，存储器就绪
  - ST指令写结果动作：将待存数据和目标地址写入对应该指令的ROB项（WB段），释放产生结果的保留站。
- Tomasulo写结果 (比较)
  - 浮点或LD指令写结果条件： r执行结束，且CDB就绪
  - 浮点或LD指令写结果动作：将计算结果[r][Data]放到CDB上，送给所有等待该结果的寄存器和保留站（包括store缓冲器），释放产生结果的保留站。（WB段）
  - ST指令写结果条件：r执行结束，待存数据就绪，当前ST指令为缓冲器队列头部，存储器就绪
  - ST指令写结果动作：将待存数据写入存储器（MEM段），释放产生结果的保留站。（WB段）

IS机制的指令执行步骤（4/4）

- IS指令确认（Tomasulo相当于在写结果同时完成了确认）
  - 每个周期检测处于ROB头部的指令，看是否已经完成写结果，若完成，则取出该指令I，进行确认操作后，释放该ROB项b。
  - 若I为浮点或LD指令：计算结果从b中取出，写入目标寄存器（CMT段）
  - 若I为ST指令：将待存数据从b中取出，写入目标存储器单元（CMT段+MEM段）
  - 若I为分支指令：
    - 猜对，则前瞻执行均有效，释放b。（CMT段）
    - 猜错，则前瞻执行均无效，清空ROB和所有保留站，且用真后继指令地址更新PC，重新流出
  - 若任何一条指令执行过程中引发异常，则清空ROB和所有保留站。

例 单流出处理器采用基于Tomasulo的IS算法进行指令调度。有一个LSU部件（内部自带加法器），2个LS缓冲器（私有时间戳实现FIFO），1个加法ALU部件，1个乘法ALU部件，2个ALU保留站。总是预测分支失败。指令序列执行前，指令均未流出，所有缓冲器/保留站均空闲。分支指令由加法ALU部件执行。ROB空间足够。

（1）各个硬件操作及指令执行的时钟周期如下表

Issue	WtCDB	Mem	Commit	Execute					
				LD	ST	SUB	BNEZ	ADD	MUL
1	1	3	1	1	1	4	4	4	10

（2）待执行指令序列如下：

指令	对应变量
Loop: LD R2, (R1)	Rt=R2, Rs=R1, Imm=0
MUL R2, R2, #2	Rd=R2, Rs=R2, Rt= #2
ST R2, (R1)	Rt=R2, Rs=R1, Imm=0
SUB R1, R1, #4	Rd=R1, Rs=R1, Rt= #4
BNEZ R1, Loop	Rt=Loop, Rs=R1
ADD R5, R3, R4	Rd=R5, Rs=R3, Rt=R4

问：

（1）请给出指令执行状态时钟周期表

（2）给出第16周期末尾各状态表内容

指令	IS	EX	MEM	WtCDB	Cmt	备注	16末状态
LD R2, (R1)	1	2	3-5	6	7		已确认
MUL R2, R2, #2	2	7-16	Null	17	18	数据相关	已执行
ST R2, (R1)	3	4	20-22	Null	19	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	20	R1已换名	已写结果
BNEZ R1, Loop	10	11-14	Null	15	21	分支失败	已写结果
ADD R5, R3, R4	16	17-20	Null	21	22	保留站冲突	已流出

Label	Busy	Op	Vj(rs)	Vk(rt)	Qj(rs)	Qk(rt)	A(Imm)	Dest
Load1	N							
Load2	Y	ST	Reg[R1]			ROB2	Reg[R1]	ROB3
ALU1	Y	MUL	Reg[R2]	#2				ROB2
ALU2	Y	SUB ->BNEZ ->ADD	Reg[R3]	Reg[R4]				ROB4 ->ROB5 ->ROB6

保留站状态表

指令	IS	EX	MEM	WtCDB	Cmt	备注	16末状态
LD R2, (R1)	1	2	3-5	6	7		已确认
MUL R2, R2, #2	2	7-16	Null	17	18	数据相关	已执行
ST R2, (R1)	3	4	20-22	Null	19	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	20	R1已换名	已写结果
BNEZ R1, Loop	10	11-14	Null	15	21	分支失败	已写结果
ADD R5, R3, R4	16	17-20	Null	21	22	保留站冲突	已流出

ROB状态表

Label	Busy	Op	Target	Value	State
ROB1	N	LD			已确认
ROB2	Y	MUL	R2		已执行
ROB3	Y	ST	Mem(Reg[R1])		已执行
ROB4	Y	SUB	R1	Reg[R1]-4	已写结果
ROB5	Y	BNEZ	#Loop	Reg[R1]-4	已写结果
ROB6	Y	ADD	R5		已流出

指令	IS	EX	MEM	WtCDB	Cmt	备注	16末状态
LD R2, (R1)	1	2	3-5	6	7		已确认
MUL R2, R2, #2	2	7-16	Null	17	18	数据相关	已执行
ST R2, (R1)	3	4	20-22	Null	19	数据相关	已执行
SUB R1, R1, #4	4	5-8	Null	9	20	R1已换名	已写结果
BNEZ R1, Loop	10	11-14	Null	15	21	分支失败	已写结果
ADD R5, R3, R4	16	17-20	Null	21	22	保留站冲突	已流出

寄存器状态表

	R1	R2	R3	R4	R5
ROB		ROB2			ROB6
Busy		Y			Y

习题 上述例题的MEM段由3个cycles变成5个cycles，重新作答。要求：

非手写答案零分。

多指令流出技术

1. 从单流出到多流出

- 单流出处理器每个周期只能流出一条指令，理想CPI等于1，因此实际CPI>1。
- 如果想进一步提高性能（CPI<1），就要允许每个周期流出多条指令，即多指令流出技术。
- 如果处理机每个周期最多流出n条指令，就称该处理机为n流出。

2. 典型的多指令流出技术：

- 静态超标量（Static Superscalar）
  - 每个时钟周期流出的指令数不固定(有上限）
  - 采用编译器静态指令调度，顺序执行
- 动态超标量（Dynamic Superscalar）
  - 每个时钟周期流出的指令数不固定(有上限）
  - 采用硬件动态指令调度（Tomasulo或IS），乱序执行
- 超长指令字（Very Long Instruction Word）
  - 每个时钟周期流出的指令数是固定的，它们构成一条长指令，或说是一个混合指令包，指令包中指令间的并行性是显式的。
  - 这种处理器目前只能通过编译器静态调度。
- 超流水（Super Pipeline）
  - 在每个流水段进一步实现流水，特别是IF段或IS段被分解为多个子段，使得一个段在一个周期内可以流水的处理多条指令。

静态超标量

- 典型的超标量处理器每个周期可流出1到8条指令。
- 指令按序流出，在流出时由硬件进行冲突检测，判断能否流出。
  - 检测包括结构冲突和数据冲突，通常在不同的段检测。
  - 如果没有冲突，n流出处理器每周期可以流出n条指令，
  - 如果其中r条指令与正在运行的指令存在冲突，则该周期可以流出n−r条指令。
- 对n流出处理器，如果允许任意类型的指令n流出，则硬件开销过大，且利用率低。因此，通常限制同时流出的指令的类型。
  - 举例：2流出处理器，通常每周期可以处理一条整型指令和一条浮点指令，因此每个周期最多可以处理这样的组合指令包，而不能处理2条浮点或2条整型指令。

静态超标量MIPS处理机

假设：每个时钟周期可流出 1条整型指令＋1条浮点指令

- LD/ST、分支归入整型指令，浮点LD/ST虽然属于整型指令，但是也要操作浮点寄存器，如果刚好1条浮点LD/ST与一条浮点指令同时流出，两条指令都要访问浮点寄存器。
  - 可以增设一个浮点寄存器的读/写端口以避免冲突
  - 也可以只允许浮点LD/ST单独流出与执行
- 为实现双流出，IF段每周期取2条指令，ID段每周期译码2条指令，因此IR应该具有64bit(即指令Cache宽度)
- 由于流水线的指令宽度增加了1倍，定向路径也要增加。
- 编译要求指令序列按类型组合成对，且与64位边界对齐，每对中整数指令顺序在前，且严格优先流出。（一列指令编译为两列例子）

静态超标量MIPS处理机例题

下面的循环程序段如果要工作在2流出静态超标量MIPS流水线上，如何进行编译器静态调度？

```
For(i=1; i<=1000; i++)
    x[i]=x[i]+s;
```

假设流水线的延迟如下：

生产者指令	消费者指令	延迟时钟周期数
浮点计算	另一个浮点计算	3
浮点计算	浮点ST	2
浮点LD	浮点计算	1
浮点LD	浮点ST	0

解:

先把高级语言程序翻译成MIPS汇编语言代码

```
For(i=1; i<=1000; i++)
    x[i]=x[i]+s;
```

翻译后

```
Loop:   LD F0,0(R1);        取一个向量元素放入F0
        ADDD F4,F0,F2;      加上在F2中的标量
        ST 0(R1),F4;        存结果
        SUBI R1,R1,#8;      指针减少8(一个DWord)
        BNEZ R1,R2,Loop;    R1不等于R2，转移到Loop
```

根据延迟表，给出一次循环内部各汇编指令的流出时钟延迟

标号	指令	流出	备注
Loop:	LD F0,0(R1);	1	浮点LD后接浮点计算，1个周期延迟
	空转	2	
	ADDD F4,F0,F2;	3	浮点计算后接浮点ST，2个周期延迟
	空转	4	
	空转	5	
	ST 0(R1),F4;	6	
	SUBI R1,R1,#8;	7	减法指令在EX开始定向，此时BNEZ已经进入ID段但当前周期错过了定向结果，因此在ID段多等一个周期读到定向结果，此时减法指令已经进入M段，因此有1个周期的定向延迟
	空转	8	
	BNEZ R1,R2,Loop;	9	分支在ID段完成，因此下轮循环延迟为1个周期
	空转	10	

使用循环展开技术展开循环

流出	第一轮	第二轮	第三轮	第四轮	第五轮
1	LD				
2	空转	LD			
3	ADDD	空转	LD		
4	空转	ADDD	空转	LD	
5	空转	空转	ADDD	空转	LD
6	ST	空转	空转	ADDD	空转
7		ST	空转	空转	ADDD
8			ST	空转	空转
9				ST	空转
10					ST
11	SUBI R1,R1,#40;				
12	空转				
13	BNEZ R1,R2,Loop;				
14	空转				

展开后的指令直接紧凑

流出	整数指令	浮点指令
1	LD	
2	LD	
3	LD	ADDD
4	LD	ADDD
5	LD	ADDD
6	ST	ADDD
7	ST	ADDD
8	ST	
9	ST	
10	ST	
11	SUBI R1,R1,#40;	
12	空转	
13	BNEZ R1,R2,Loop;	
14	空转	

不同的颜色，使用不同的寄存器

不同的颜色，使用不同的地址偏移进行LD/ST，修正后的地址：
0(R1)，
-8(R1)，
-16(R1)，
-24(R1)，
-32(R1)

五个SUBI合并成一个
因此8*5=40

展开后的指令进行调度

流出	整数指令	浮点指令
1	LD	
2	LD	
3	LD	ADDD
4	LD	ADDD
5	LD	ADDD
6	ST	ADDD
7	ST	ADDD
8	ST	
9	ST	
10	ST	
11	SUBI R1,R1,#40;	
12	空转	
13	BNEZ R1,R2,Loop;	
14	空转	

调度过程：

ST的作用就是将算好的值写回存储器，因此可以向后移动，填充到空转的位置。
注意：由于ST的地址是由基址R1计算出来的，SUBI将R1减少了40，因此一旦某个ST移到SUBI之后，要把这40加回来，才能保证ST的目标地址仍旧为指令移动前的数值。

由此，将9号指令填充12号空转，10号指令填充13号空转，9号10号新空出来的位置由后面所有的指令前移补足，结果见下页

展开后的指令进行调度

流出	整数指令	浮点指令
1	LD	
2	LD	
3	LD	ADDD
4	LD	ADDD
5	LD	ADDD
6	ST	ADDD
7	ST	ADDD
8	ST	
9	SUBI R1,R1,#40;	
10	ST	
11	BNEZ R1,R2,Loop;	
12	ST	

调度过程：

SUBI之前的LD/ST指令的地址如下：
0(R1),
-8(R1),
-16(R1),
-24(R1),
-32(R1)

SUBI之后的2个ST指令的地址如下：
16(R1),
8(R1)

为什么最好是5轮循环展开？

流出	整数指令	浮点指令
1	LD	
2	LD	
3	LD	ADDD
4	LD	ADDD
5		ADDD
6	ST	ADDD
7	ST	
8	SUBI R1,R1,#40;	
9	ST	
10	BNEZ R1,R2,Loop;	
11	ST	

如果是4轮循环展开，结果如左图所示，第5个周期有一个空转无法填满，只有大于等于5轮展开，才能填上这个空转。

如果展开超过5轮，前两个周期浮点指令的空转仍旧不能填满，而展开的循环越多，占用的寄存器总量就越大，有可能导致寄存器不足而出错。

因此，5轮最佳。

静态指令调度的性能分析：

- 指令展开与指令调度之前，每轮循环10个周期，5轮循环需要50个时钟周期
- 指令展开与指令调度之后，5轮循环一共需要12个周期
- 优化加速比：50/12=4
- 在硬件支持2路超标量的情况下，性能提高了4倍。

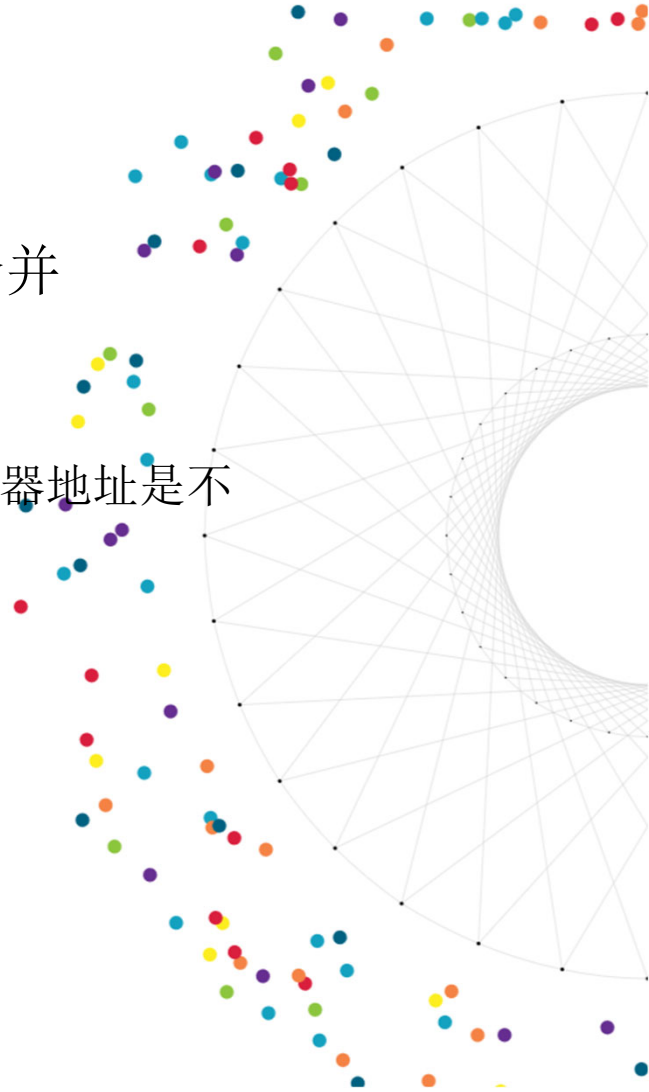
解毕。

影响静态超标量MIPS性能的因素

- load指令
  - 编译器编译好的指令序列中，load指令后续的3条指令都不能使用该load的结果，否则就会引起流水线停顿。
- 分支指令
  - 分支指令为整数指令，因此一定为指令对的第一条指令
  - 即使分支延迟为1个时钟周期，由于每个周期流出2条指令，因此分支指令影响的是其后的3条指令可能进入流水线后由于分支成功而被作废。
- 多流出指令的效率平衡问题
  - 由于指令分布的不均衡性，导致n流出处理器的n条流水线得不到足够的指令来达到饱和。

循环展开和静态指令调度时要注意

- 注意分支条件的修改和控制指令的合并
- 注意LD/ST指令地址偏移量的修正
- 注意对代码的相关性分析
  - 不同循环迭代中使用的寄存器地址和存储器地址是不相关的
  - 循环展开可能引入新的相关性



动态超标量

- 指令按序流出，根据保留站空闲情况判断能否流出。
- 对于n流出处理器，每个周期可以流出n条指令，只要对应保留站空闲。一般每个周期最多有n条指令处于执行状态。
- 2流出处理器的设计实例：
  - 一个流出段，每半个周期流出一条非分支指令，则每个周期最多可以流出2条非分支指令，分支指令单独流出。
  - 两个执行段，一个整数ALU，一个浮点ALU，最多可以接受一条整型指令和一条浮点指令同时执行，整数运算需1个周期，浮点运算需3个周期，分支指令和LD/SD指令视为整数运算指令
  - 一个写结果段，写结果1个周期
  - 一个确认段，指令确认1个周期
  - 采用IS机制进行指令动态调度
  - 指令使用ROB按序确认，保留站都合并到ROB中
  - 无定向路径

动态超标量MIPS处理机例题

下面的循环程序段如果要工作在采用IS机制的2流出动态超标量MIPS流水线上，如何进行指令的动态调度？

（假设分支预测完美，且ROB足够大）

Loop:	LD F0,0(R1);	取一个向量元素放入F0
	ADDD F4,F0,F2;	加上在F2中的标量
	ST 0(R1),F4;	存结果
	SUBI R1,R1,#8;	指针减少8(一个DWord)
	BNEZ R1,R2,Loop;	R1不等于R2，转移到Loop

解：

在这里，不能使用静态的循环展开技术。

使用指令执行状态表表示指令的调度执行过程。

表格见下页

遍数	指令	流出	执行	写结果	确认	说明
1	LD F0,0(R1);	1	2	3	4	流出第一条指令
1	ADDD F4,F0,F2;	1	4	7	8	数据相关，浮点延迟
1	ST 0(R1),F4;	2	3	8	9	数据相关
1	SUBI R1,R1,#8;	2	4	5	10	R1操作数已缓冲，ALU冲突
1	BNEZ R1,R2,Loop;	3	6		11	数据相关
2	LD F0,0(R1);	4	7	8	12	R1被预约导致寄存器换名， ALU冲突+数据相关
2	ADDD F4,F0,F2;	4	9	12	13	数据相关+ALU冲突， 浮点延迟
2	ST 0(R1),F4;	5	8	13	14	R1被预约导致寄存器换名， ALU冲突，数据相关
2	SUBI R1,R1,#8;	5	9	10	15	R1被预约导致寄存器换名， ALU冲突
2	BNEZ R1,R2,Loop;	6	11		16	数据相关

动态指令调度的性能分析：

- 借用上一个例题的结果，调度优化之前，每轮循环10个周期
- 指令动态调度之后，1轮循环一共需要11个周期，2轮循环需要11+5个周期，n轮循环需要6+5n个周期
- 优化加速比： p=10n/(6+5n)
- 在硬件支持2路超标量的情况下，
- n=1, p=0.91； n=2, p=1.25； n=3, p=1.43；
- 性能随循环次数而提高，加速比极值为2倍。

解毕。

思考：若计算过程中除法指令导致除零异常，如何处理？

影响动态超标量MIPS性能的因素

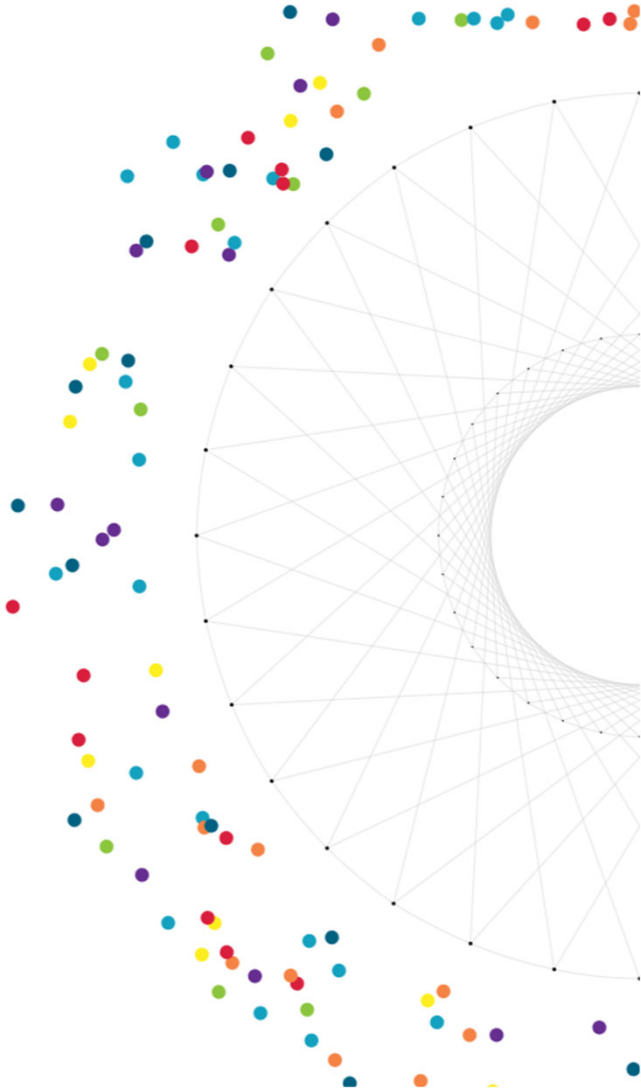
- 整数部件和浮点部件的工作负载不平衡，没有充分发挥出浮点部件的作用。
  - 应该设法减少循环中整数型指令的数量，或者增加整数处理部件
  - 例如增加用于计算LD/ST指令地址的加法器
- 每轮循环中的控制开销太大。
  - 5条指令中有两条指令是辅助指令，应该设法减少或消除这些指令
  - 例如再加入循环展开技术，将控制指令合并
- 分支指令控制相关导致的延迟
  - 处理机必须等到分支指令的结果出来后才能开始下一轮循环首条指令的执行。

超长指令字 VLIW

- VLIW处理器是一种特殊的n流出超标量处理器。在VLIW中，每次指令调度与执行的最小单位不再是一条一条的32bit指令，而是多条可并行的指令组成的长指令，每条长指令的平均长度为一百到几百比特。
- VLIW处理器中，IR含有n个操作槽，每个操作槽固定的服务于一个独立的功能部件，并控制该功能部件执行操作槽中的指令。
- 每个操作槽对应的功能部件类型都是暴露给编译器的，编译器据此来进行指令调度，将可以并行执行且拼接起来满足操作槽类型要求的指令拼接为长指令。静态指令调度通常和循环展开技术共同使用。
- 不同的长指令之间是按序流出和执行的。同一个操作槽内部的所有并行指令都是同时流出和并行执行的。一旦一条长指令不能流出，其后所有的长指令都不能流出和执行。
- 由于相关等原因，编译器调度优化后的长指令序列，不能保证每条长指令都能填满n个操作槽。

5流出VLIW的设计实例

- 一个IF段，每个周期取一条长指令
- 一个ID段，每个周期流出一条长指令
- 一个EX段，含有5个独立处理部件
  - **2个LSU**（浮点**LD/ST**）
  - **2个浮点ALU**（浮点计算）
  - **1个整数ALU**（整数计算和分支）



VLIW例题

下面的循环程序段如果要工作在5流出VLIW流水线上循环展开5次，如何进行指令的编译器静态调度？

```
Loop:   LD F0,0(R1);        取一个向量元素放入F0
        ADDD F4,F0,F2;      加上在F2中的标量
        ST 0(R1),F4;        存结果
        SUBI R1,R1,#8;      指针减少8(一个DWord)
        BNEZ R1,R2,Loop;    R1不等于R2，转移到Loop
```

假设：

生产者指令	消费者指令	延迟时钟周期数
浮点计算	另一个浮点计算	3
浮点计算	浮点ST	2
浮点LD	浮点计算	1
浮点LD	浮点ST	0

根据延迟表，给出一次循环内部各汇编指令的流出时钟延迟

标号	指令	流出	备注
Loop:	**LD F0,0(R1);**	**1**	**浮点LD后接浮点计算，1个周期延迟**
	空转	**2**	
	ADDD F4,F0,F2;	**3**	**浮点计算后接浮点ST，2个周期延迟**
	空转	**4**	
	空转	**5**	
	ST 0(R1),F4;	**6**	
	SUBI R1,R1,#8;	**7**	**浮点计算EX段尾出结果定向给BNEZ的ID段，BNEZ在ID段开始读结果时浮点计算已经进入M段，因此有1个周期的定向延迟**
	空转	**8**	
	BNEZ R1,R2,Loop;	**9**	**分支延迟为1个周期，之后才能进入下轮循环**
	空转	**10**	

使用循环展开技术展开循环

流出	第一轮	第二轮	第三轮	第四轮	第五轮
1	LD	LD			
2	空转	空转	LD	LD	
3	ADDD	ADDD	空转	空转	LD
4	空转	空转	ADDD	ADDD	空转
5	空转	空转	空转	空转	ADDD
6	ST	ST	空转	空转	空转
7			ST	ST	空转
8					ST
9	SUBI R1,R1,#40;				
10	空转				
11	BNEZ R1,R2,Loop;				
12	空转				

展开后的指令直接紧凑

流出	浮点ALU	浮点ALU	LSU	LSU	整数ALU
1			LD	LD	
2			LD	LD	
3	ADDD	ADDD	LD		
4	ADDD	ADDD			
5	ADDD				
6			ST	ST	
7			ST	ST	
8			ST		
9					SUBI R1,R1,#40;
10					空转
11					BNEZ R1,R2,Loop;
12					空转

展开后的指令进行调度

流出	浮点ALU	浮点ALU	LSU	LSU	整数ALU
1			LD	LD	
2			LD	LD	
3	ADDD	ADDD	LD		
4	ADDD	ADDD			
5	ADDD				SUBI R1,R1,#40;
6			ST	ST	空转
7			ST	ST	BNEZ R1,R2,Loop;
8			ST		空转
9					SUBI R1,R1,#40;
10					空转
11					BNEZ R1,R2,Loop;
12					空转

调度过程：

不同颜色的指令，使用不同的寄存器和不同的偏移，和前面例题相同，此略。

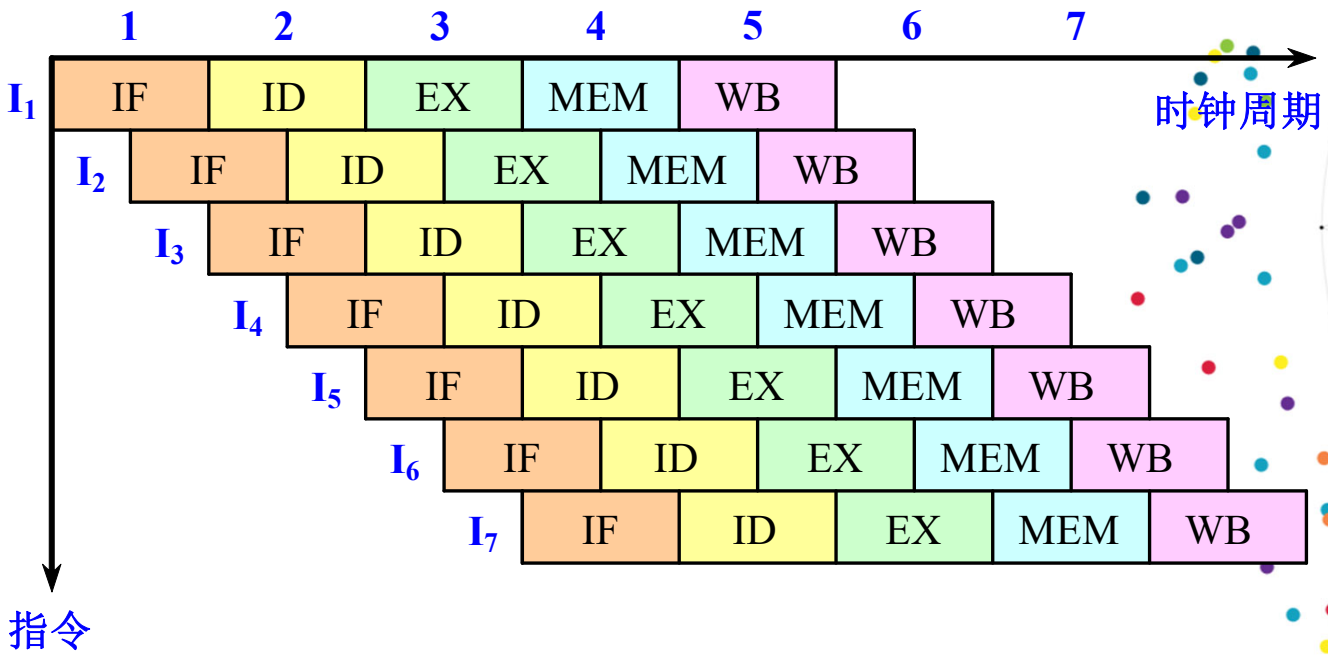
9-12的整数指令提前到5-8，由于所有ST都从SUBI前移到SUBI后，所以偏移量都加40.

解毕。

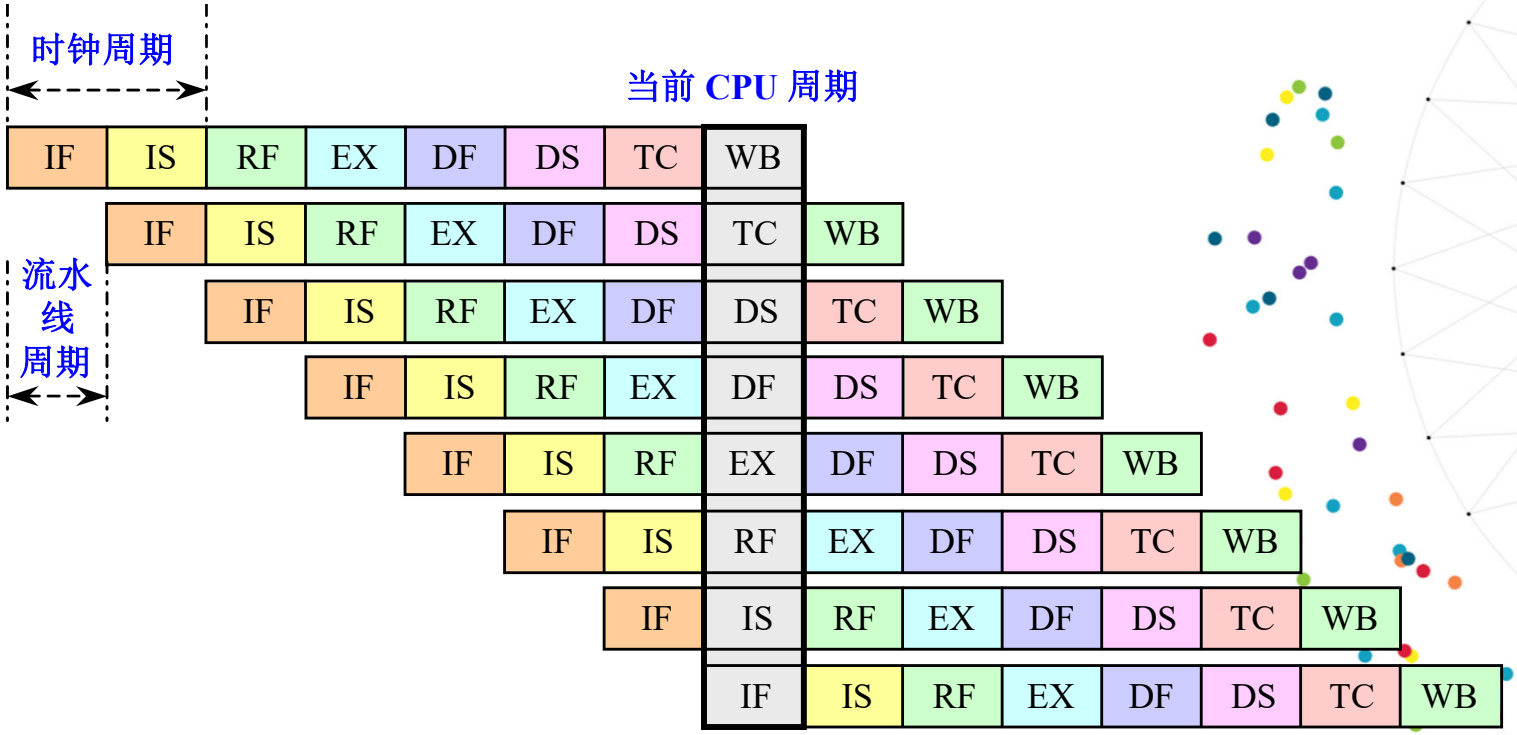
超流水

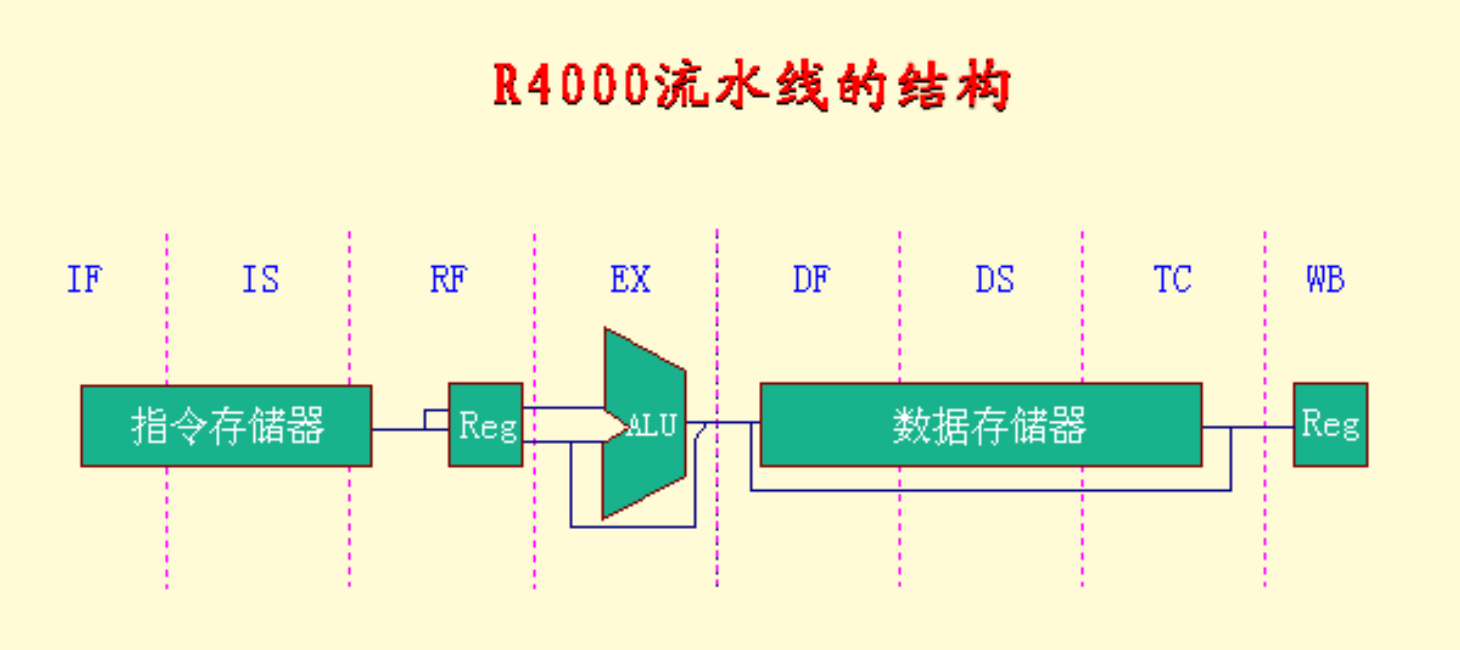
- 将每个流水段进一步细分，这样在一个时钟周期内能够分时流出多条指令。这种处理机称为超流水线处理机。
- 对于每个时钟周期能流出n条指令的超流水线计算机来说，这n条指令不是同时流出的，而是每隔1/n个时钟周期流出一条指令。实际上该超流水线计算机的流水线周期为1/n个时钟周期。

一台每周期分时流出两条指令的超流水线计算机的时空图



一个典型超流水产品(MIPS R4000)指令流水线时空图




R4000流水线的结构
IF
IS
RF
EX
DF
DS
TC
WB
指令存储器
Reg
ALU
数据存储器
Reg

- 各级的功能
  - IF：取指令的前半步，根据PC值去启动对指令Cache的访问。
  - IS：取指令的后半步，在这一级完成对指令Cache的访问。
  - RF：指令译码，访问寄存器组读取操作数，冲突检测，并判断指令Cache是否命中。
  - EX：指令执行。包括有效地址计算，ALU操作，分支目标地址计算，条件码测试。
  - DF：取数据的前半步，启动对数据Cache的访问。
  - DS：取数据的后半步，在这一级完成对数据Cache的访问。
  - TC：标识比较，判断对数据Cache的访问是否命中。
  - WB：load指令或运算型指令把结果写回寄存器组。