硬件投机及多发射

浙大计算机 陈文智
chenwz@zju.edu.cn
2014年11月

3.7 Reducing Branch Costs with Dynamic Hardware Prediction(2.3)

- 1-bit Branch-Prediction Buffer
- 2-bit Branch-Prediction Buffer
- Correlating Branch Prediction Buffer
- Branch Target Buffer
- Return Address Predictors

概述

- 本节内容: 通过硬件动态预测转移指令的行为来减少转移代价
- 基本思想: **设置一预测branch指令行为(即转移成功与否)的硬件,取出Branch指令的同时,取出其预测结果(转移成功或不成功).**
  - 若预测为不成功,则下一节拍就立即取出下一条指令,无任何停顿.
  - 若预测为成功,则继续执行Branch指令,计算出转移地址,此时有一个stall,存在delay slot.

- **Branch动态预测特别适用于用静态调度方法即在compiling阶段无法解决(预测)的情况。**
  - 如本次转移方向取决于其它的**branch**结果，即存在关联（**correlation**）关系的时候。
- 转移预测的效率与下列因素有关：
  - 预测的正确率
  - 转移的代价，指预测正确时的代价，和预测出错时的代价。
- 转移的代价与下列因素有关：
  - 流水线的结构（如硬件安排方式等）
  - 预测器的类型（将在下面介绍不同预测器的效率）
  - 预测失败时，恢复的策略。

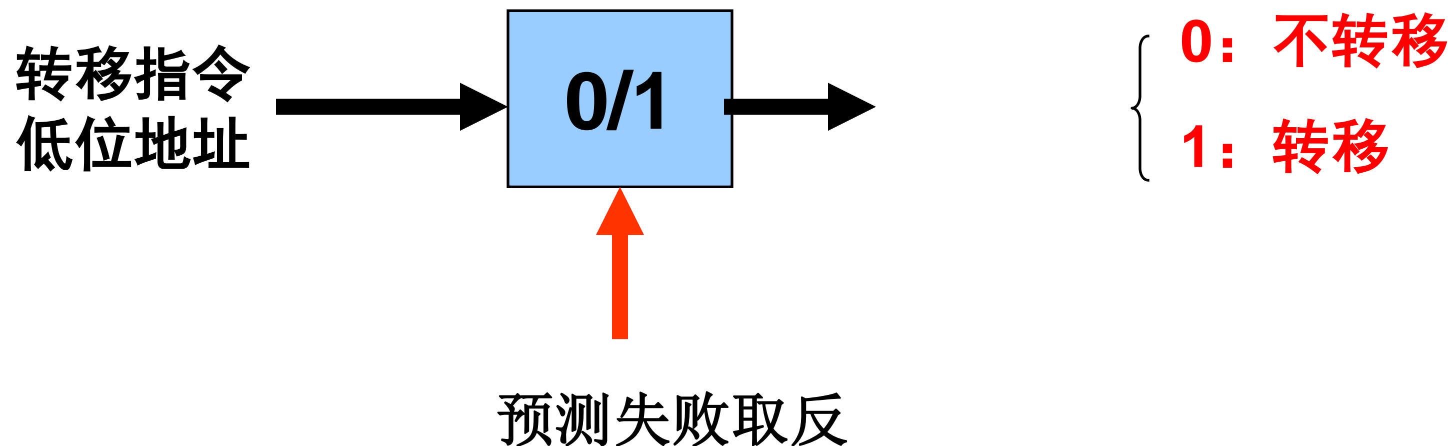
3.7.1 Basic Branch predication and Branch-Predication Buffers

最简单的动态转移预测器就是采用转移预测缓冲器，缓冲器中储存预测信息，又可称为转移历史表。这是一个小的存储器，由转移指令地址的低位来索引。

介绍两种转移预测缓冲器方案：一位预测器（one-bit)和两位预测器（two bits)

一、一位转移预测缓冲器 one-bit predictor

- 建立一个只有一位的缓冲器，存放当前程序的转移行为，并用转移指令的低位读取；根据其中的值预测当前转移指令的行为；预测命中值不变，预测失败则修改缓冲器内的值



存在问题：

- 由于预测缓冲器单元，是由转移指令的低位地址索引的，因此该单元的信息可能由另一条低位地位与本条Branch指令相同的branch指令的转移历史纪录，并非本条branch指令上一次转移行为的历史纪录。
- 这实际上是没有关系的。因为我们把预测仅仅是看作一种提示（预测），若预测是正确的，则按预测方向取指令；若预测是错误的，程序仍按正确方向执行，同时将预测位置反即可。也就是说branch指令仍然是在执行的，一旦branch指令的判断与预测结果相矛盾时，仍按实际Branch的实际结果执行。

一位转移预测器性能

- 转移预测缓冲器从硬件角度来看类似于一个每次都命中的Cache.同时该缓冲器的性能是与下列因素有关的。
  - 预测位的结论属于我们感兴趣转移指令的频度，即预测结论与转移指令是否匹配。
  - 预测的正确性（一旦匹配的话）。
- 我们可以利用Cache技术来提高匹配率。

例：某循环体转移行为是循环体一次迭代有9次是转移成功，紧接后面一次转移不成功。假定用一位转移预测缓冲器来控制转移，对此循环的预测正确率是多少？

解：

设缓冲器初值为：**0**

预测第一次转移不发生，实际转移，**预测失败**，修改为：**1**

后连续**8**次成功转移，并预测成功，预测位指向成功：**1**

预测第**10** 次转移成功，实际不转移，**预测失败**，修改为：**0**

程序本身的转移成功率是**90%**

预测成功率是**80%**

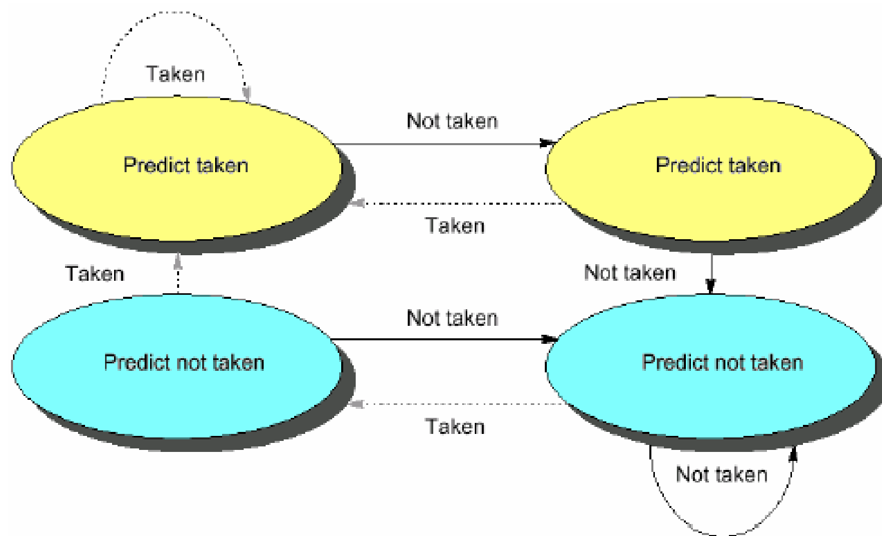
这是较理想的情况

一位预测器的缺点：

- **如果程序转移行为是间隔变换一次，则预测命中率为 0%**
- 改进的方法是：采用两位预测器。

二、二位转移预测器（two-bit predictor)

- 每个预测器采用2位，**只有**预测连续出错两次后才改变预测方向。



对偶然一次预测出错，不会出现两次预测错

多位预测缓冲器
增加缓冲器容量
预测方案：当前转移行为修改

•转移预测缓冲器的实现技术

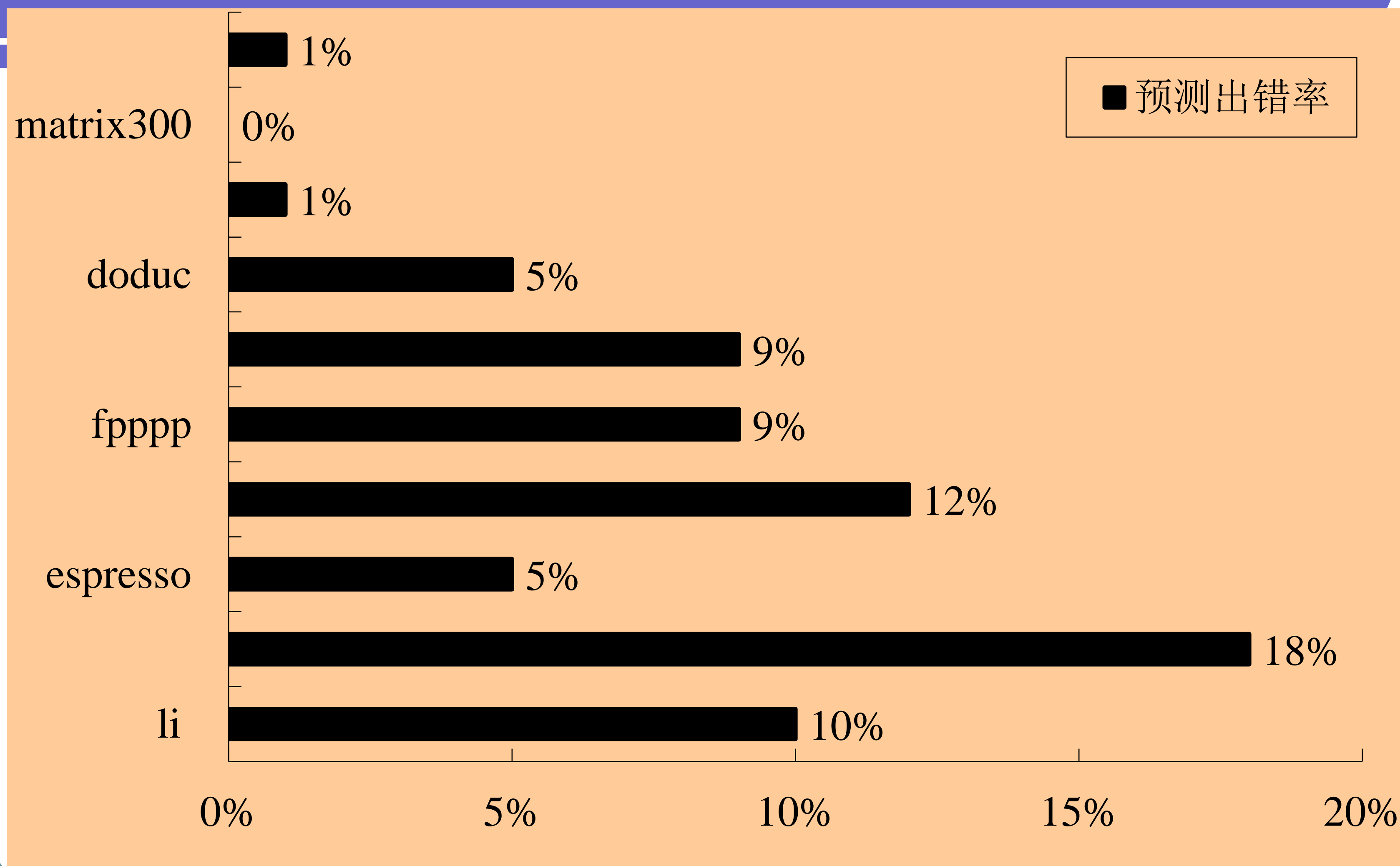
两种实现方法：

- 该缓冲器作为一种专门的**Cache**, 在**IF**节拍取指时，用指令地址访问这一缓冲器。即取出指令时，同时取出预测值。
- 在指令**Cache**每一**Block**中附加两个预测位，从而与指令一起读出。

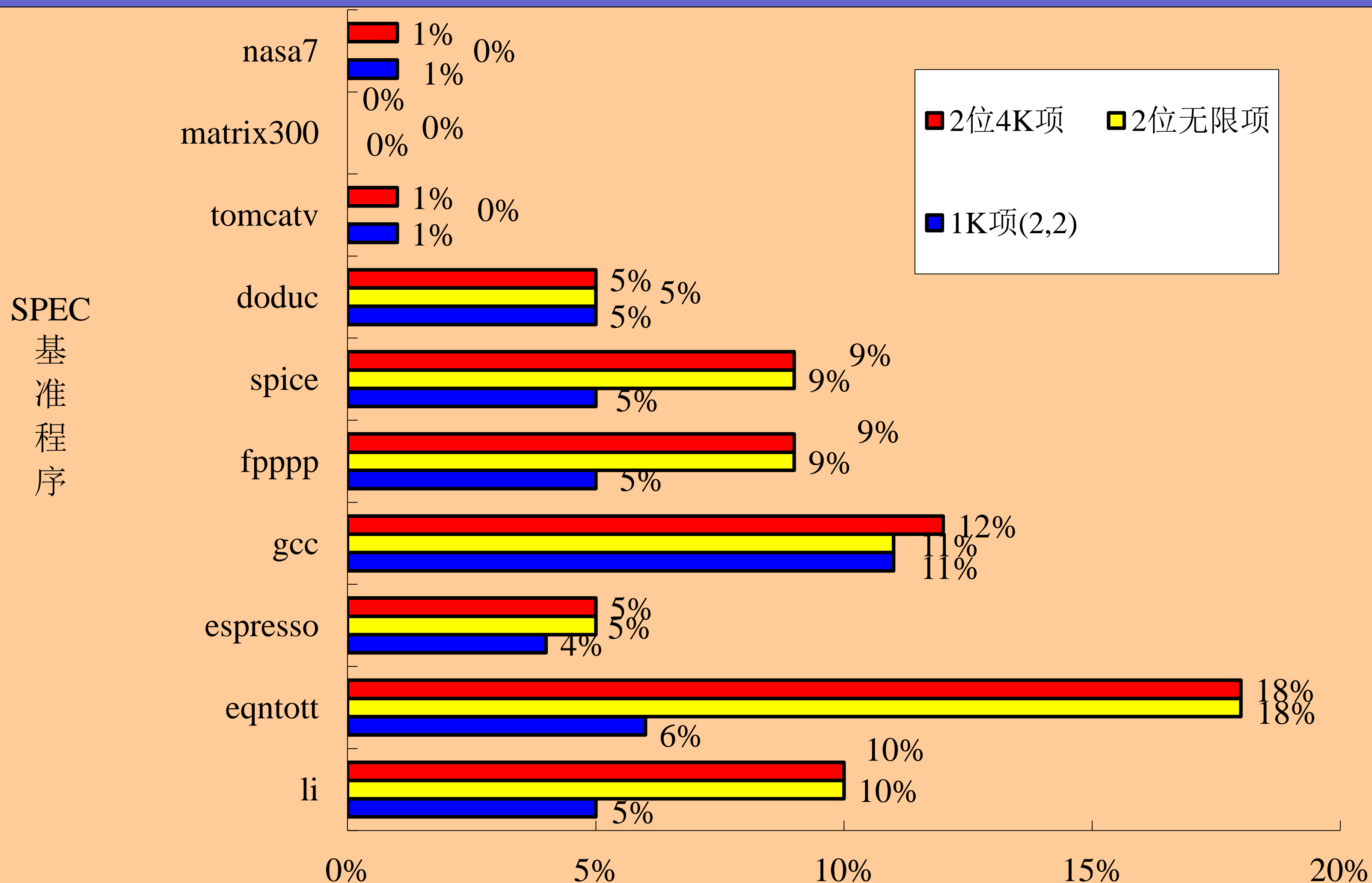
转移预测的正确率

- 测试条件：
  - 转移预测缓冲器有4096个entries，每个entry含 2位预测位；
  - 对SPEC89 的banchmark进行测试

转移预测的正确率



缓冲大小对转移预测出错率的影响



两位预测器测试结果：

- 测试结果：
  - 预测正确率：**82%－99%**
  - 预测出错率：**1%－18%**
  - 小**buffer**结果差，**4K**的**buffer**足够大
  - 浮点测试程序的预测正确率高于整数测试程序。因为浮点程序中**loop**出现次数多。
  - 由图说明：**buffer**大于**4K**已对提高预测正确率无益。同样，增加预测位（即大于**2 bits)**也无益。

三、相关转移预测缓冲器

如何进一步提高预测的正确率？

- 迄今为止，我们根据转移指令最近的转移行为预测当前的转移行为。进一步提高预测正确率的出路在于根据多个相关转移指令的行为来预测我们感兴趣branch指令的行为，即称为相关预测（correlating prediction)或两级预测(two-level prediction)。

例

```
IF ( aa == 2)
        aa = 0;

IF ( bb == 2)
        bb = 0;

IF (aa !== bb ) {
  ……
  }
```

```
     DSUBI   R3, R1, #2
     BNEZ    R3, L1          ; br.b1 (aa!=2)
     DADD    R1, R0, R0  ; aa==0
L1:  DSUBI   F3, R2, #2
     BNEZ    R3, L2          ; br.b2 (bb!=2)
     DADD    R2, R0, R0 ; bb==0
L2:  DSUB    R3, R1, R2;  R3=aa-bb
     BEQZ    R3, L3        ; br.b3 (aa==bb)
```

例子说明：

- 这里b3的行为与b1,b2两条转移指令相关，即当b1，b2不成功时，b3会成功。
- 如果我们仅根据b3过去行为来预测b3当前行为是不可能抓住这个特点的 。
- 如何预测此类转移指令？参看下例：

例

```
If (d==0)
        d=1;
if (d==1){
    ……
}
```

- **设Reg[R1] = d**

```
    BNEZ     R1, L1       ; br b1, (d!=0)
    DADDIU   R1, R0, #1   ; d==0, so d=1
L1: DADDIU   R3, R1, #-1  ;
    BNEZ     R3, L2       ; br b2, (d!=1)
       ……
L2:
```

设d的初值为0,1,2,上述代码段的转移特征如下:

d 的初值	d==0?	B1	在 b2 以前的 d 值	d==1?	b2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	no	Taken

结论: **当b1 not taken, b2也为not taken.**

所以, 若利用相关预测器,就能成功作出预测.

若利用传统的one-bit预测器,则无法利用这一相关性,且预测总是错的。

d=?	b1 预测	b1 动作	新的 b1 预测	b2 预测	b2 动作	新的 b2 预测
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

预测与实际动作总是相反

预测与实际动作总是相反

引入相关性的预测器：

- 设branch的预测器由两位(两部分)组成：

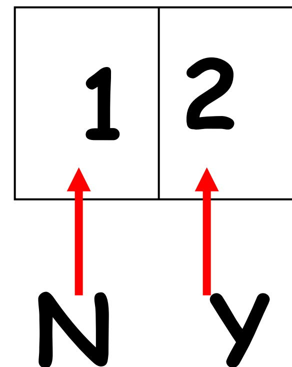
第一位是上次br.为NT时的预测，

即上次br.为NT时取第一位作预测值；

第二位是上次br.为T时的预测，

即上次br.为T时取第二位作预测值；

这样就有四种预测组合。



四种组合的含义：

预测组合	上一次 Br 为 NT,预测本次位(看第一位)	上一次 Br 为 T,预测本次为(看第二位)
NT/NT	**NT**	**NT**
NT/ T	**NT**	**T**
T /NT	**T**	**NT**
T / T	**T**	**T**

- **注意：**
  - 这里体现了相关性
  - 虽然上一次Br指令，并非一定是本次br指令，但在简单的loop中是可能的，如简单loop中不含其它br指令。

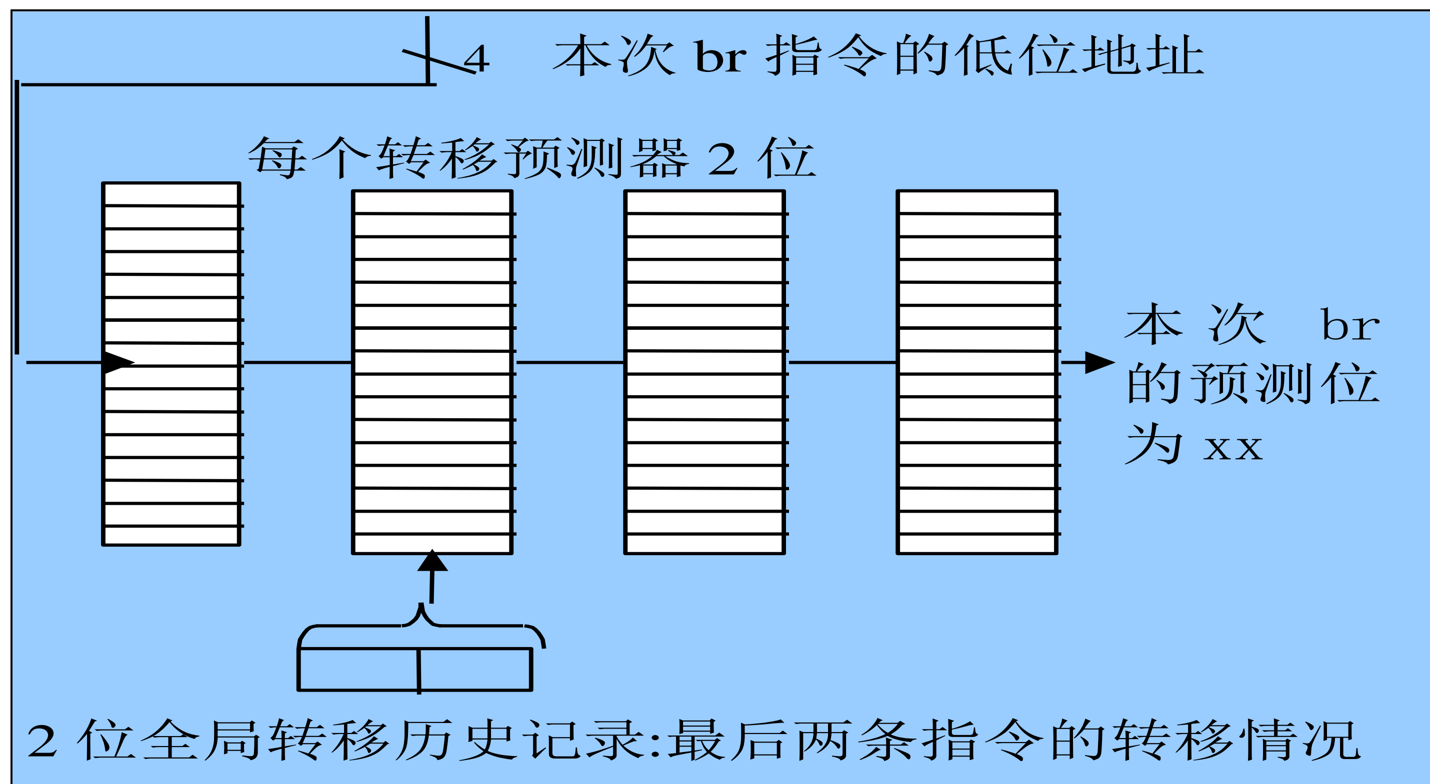
用这种相关预测器来预测上述例子。初值为NT/NT.（第一次迭代预测错，其余均正确）

d=?	b1 预测	b1 动作	新的 b1 预测	b2 预测	b2 动作	新的 b2 预测
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

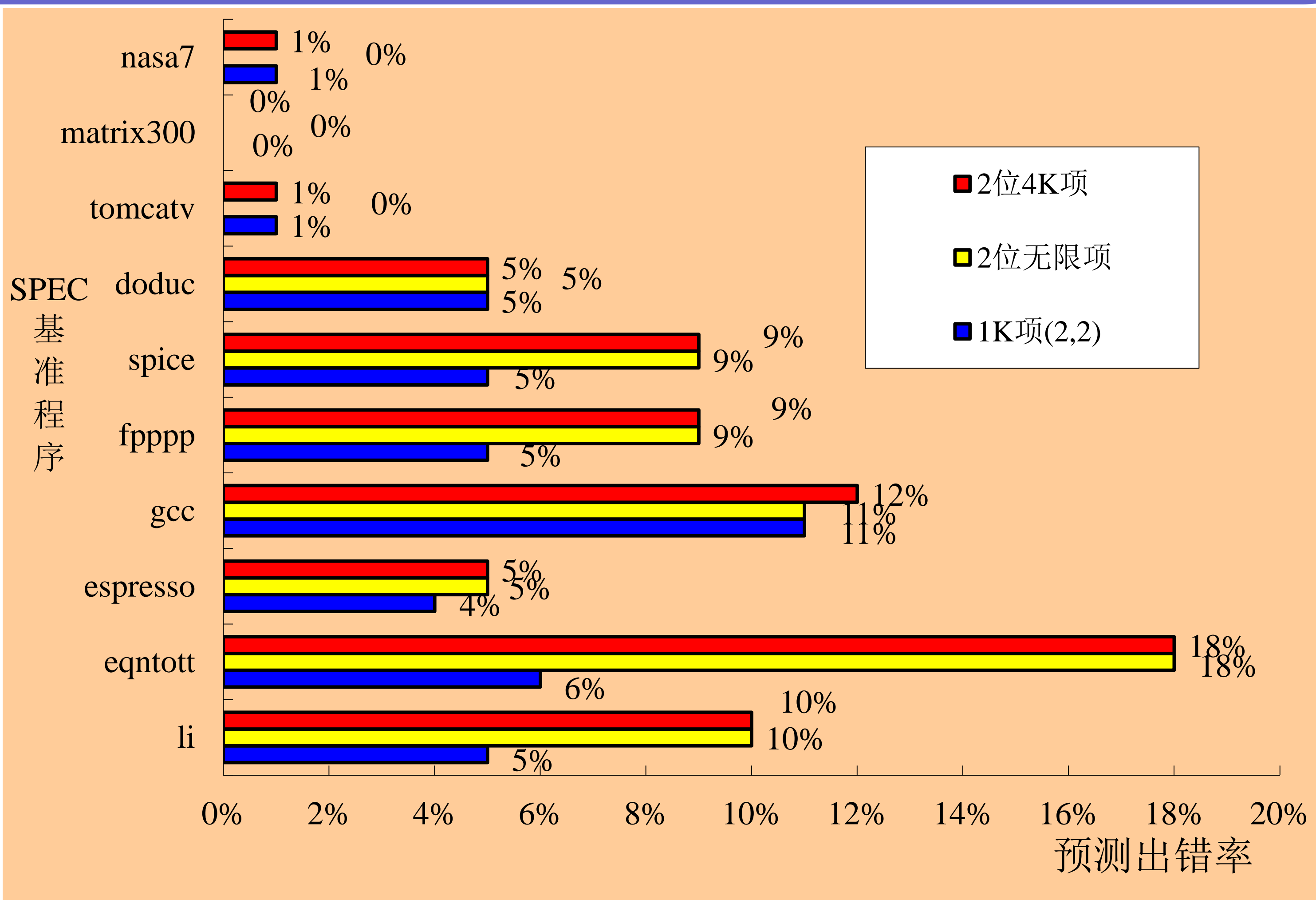
- 上面相关预测器称为(1,1)维转移预测器,即根据前一次br指令的执行情况,以一对预测位选择预测值。
- 一般情况的相关预测器应为（m,n)维，即根据前m条br指令的转移历史纪录，从2^m个预测器中选择一个预测器，每个预测器有n位来预测本次br指令的行为。

（2，2）相关预测器的硬件框图：

可寻址的预测单元为：2^{4+2}=64个



相关预测器与简单预测器性能比较（前提是总预测容量相等）



比较结果

- 预测容量

(0,2)4K entries: $2^0 \times 2 \times 4K = 8K$

(2,2)1K entries: $2^2 \times 2 \times 1K = 8K$

- 结论

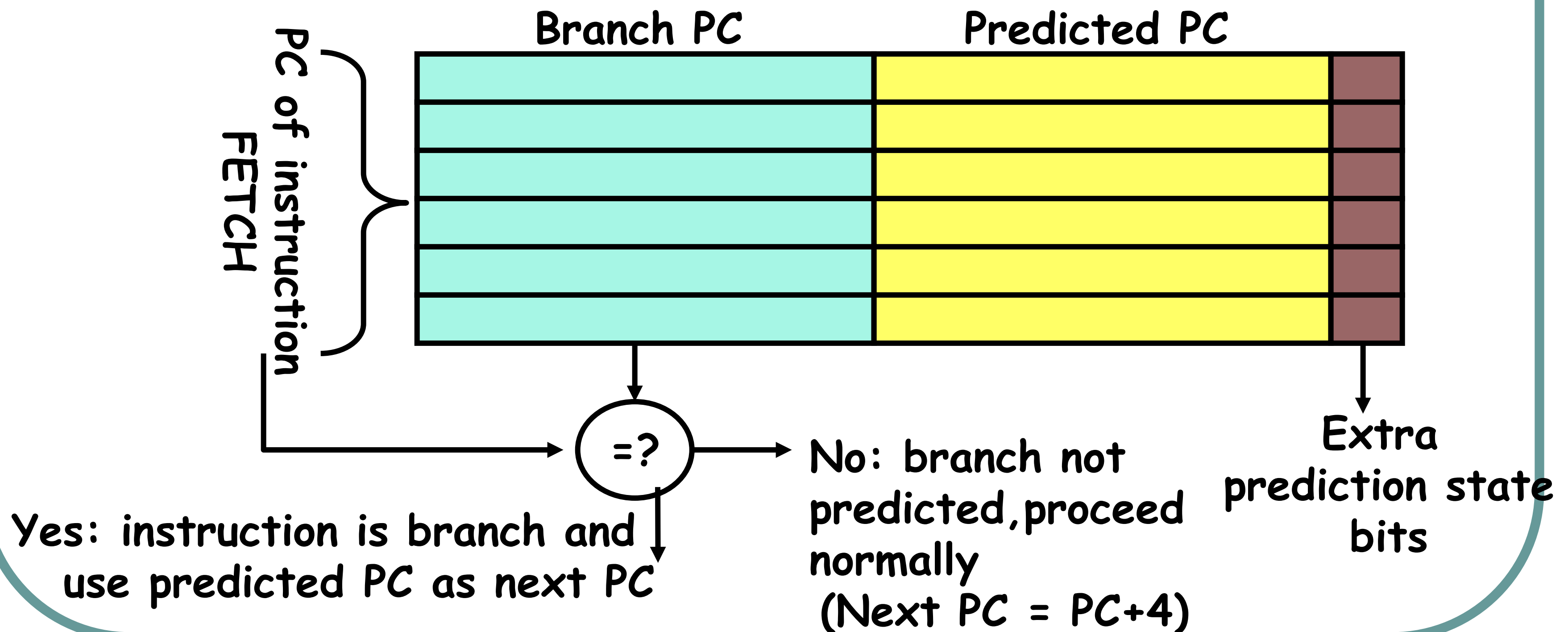
相关预测器的性能明显优于简单预测器。

3.7.2 转移目标缓冲器(Branch Target Buffer) ----进一步减少控制竞争带来的延迟

- 转移目标缓冲器
  - **如果一个转移预测缓冲器存储了被调用转移指令的下一条要执行的****预测指令的地址****，则称之为**转移目标缓冲器，**或**转移目标Cache.
    - 希望在IF结束时知道下条指令的指针，即在译码前知道是否是转移指令及转移行为。
    - 建立缓冲器用来存放转移指令地址和该指令的转移行为

转移目标缓冲器结构:

- Branch Target Buffer (Branch Target Cache):
  - Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address



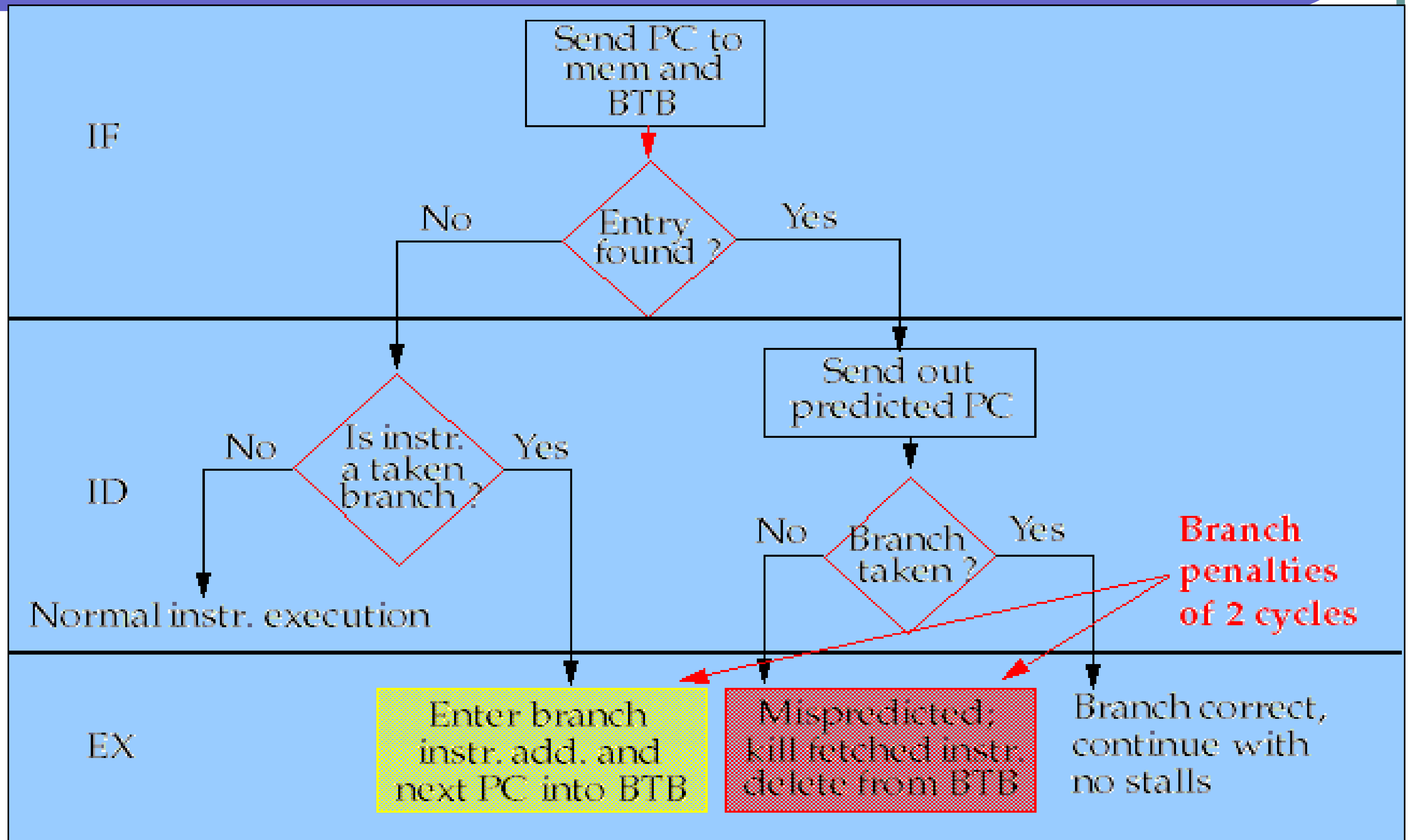
转移预测缓冲器与转移目标缓冲器的差别

- 在IF级访问转移目标缓冲器，在IF级结束前就能得到转移目标地址。

  一般，在ID级访问转移预测缓冲器，在ID级结束前得到转移目标地址；
- 访问转移目标缓冲器时，还无法判定是否是转移指令，所以必须进行PC值的匹配。

  而转移预测缓冲器是按地址访问的
- 转移目标缓冲器中只需存放预测转移成功的转移指令，无需存放预测不成功的转移指令。

采用转移目标缓冲器时的指令流水处理过程



转移目标缓冲器的几种变形

- 在转移目标缓冲器中直接存放转移目标指令而不是转移目标指令地址
- 同时存放转移目标指令和转移目标地址
- 设置很大的目标缓冲器，即存放预测路径的转移目标指令，也同时存放非预测路径的转移目标指令。这要求存储器系统必须是双端口的，cache是以并行交叉方式工作的。

3.8 Hardware-Based Speculation (2.6)

一、基于硬件投机技术概述

- 基本概念：**基于硬件的投机技术实质上是综合了下述三种技术的一种集成技术，它们是：**
  - **应用动态转移预测技术选择投机指令；**
  - **应用投机技术达到在控制相关性消除以前就执行投机指令；**
  - **应用动态调度技术来调度程序基本块的不同组合。**

实际上就是动态投机和动态调度相结合的一种技术。

基于硬件的投机技术的优点（1）

1、便于扩大投机指令的范围

- 例如，程序很难在编译时明确存储器访问的地址，而基于硬件的投机技术是在程序动态执行时确定访问存储器地址的，因而有利于扩大投机指令的范围；

2、在硬件转移预测上实现的基于硬件投机技术比在软件转移预测上实现的基于编译投机的效率更高。因为硬件转移预测的正确率要高于静态转移预测。

基于硬件的投机技术的优点（2）

3、基于硬件的投机能保证完全精确的中断处理模式，即使是投机产生中断也是一样。其理由将进一步介绍；

4、基于硬件的投机不需要补偿或纪录代码；

5、采用动态调度的基于硬件投机技术在体系结构的不同实现方案中可以不用不同的编译器就能保证其性能。基于编译的投机和调度通常在体系结构的不同实现机种中，要求对代码序列作适当调整才能确保其性能不变，通常老的程序代码的性能会低一些。

基于硬件的投机技术的缺点

1、硬件代价高

2、硬件复杂

本节将介绍已被众多著名微处理器（如 **PowerPC 603/604/G3/G4, MIPS R10000/R12000, Intel II/III/P4/, Alpha 21264,and AMD K5/K6/Athlon**等）采用的基于**Tomasulo**动态调度的基于硬件的投机技术。

二、基于Tomasulo动态调度的硬件投机

- Tomasulo算法的基本思想：
  - 针对数据相关性而提出；
  - 容许指令不按序执行，只要操作数就绪就可以执行；
  - 容许指令不按序结束。

将基于Tomasulo算法的硬件经过扩充用来支持投机执行：

1、从解决数据相关性进一步扩充到解决控制相关性；

2、容许指令，包括提前到转移指令前执行的投机指令，在操作数就绪后，就可以执行，即容许指令不按序执行，从而进一步提高调度性能，开发出更多的ILP;

3、所有指令必须按序结束。

实际执行过程是：经过投机和动态调度以后，指令乱序执行，乱序得到其结果，并被其他指令所应用，但是不能更新指令的目的寄存器、或写入存储单元，因此指令实际并未结束。

所谓按序结束是指：指令必须按源代码顺序更新其目的寄存器或写入存储器单元。由此可见，在执行指令结束这一步时，实际上投机已经成功。这样做的目的是为了一旦投机失败时，可以恢复代码段的原始数据，不至于因投机失败而造成错误的纪录。

结论（1）

1、必须把指令乱序执行与实际结束分离开来，成为两步实现；

- 乱序执行是动态调度的需要，必须把指令的执行结果，包括投机指令的结果，通过旁路方法，随时提供给其它指令使用；
- 按序结束是为了确保不因投机失败而造成出错的需要，也是为了确保实现精确中断的需要（能确保恢复中断前的状态）。

结论（2）

2、为此，在Tomasulo算法把指令分为Issue, Execute, 和Write result三步的基础上，增加一步，称为Commit（交付，后提交）。Commit的功能（将在下面作进一步介绍）是指令将其结果交付给（写入）目的寄存器或存储单元；

3、必须增加一硬件缓冲存储器（buffer)，供Write Result这一步存放已获得的结果，并可以提供给其它指令应用这些结果。当指令进入Commit这一步时，将结果从buffer中拷贝到目的寄存器或存储单元。这一硬件缓冲存储器称为重构序缓冲存储器（Reorder Buffer),或简称重组缓存。

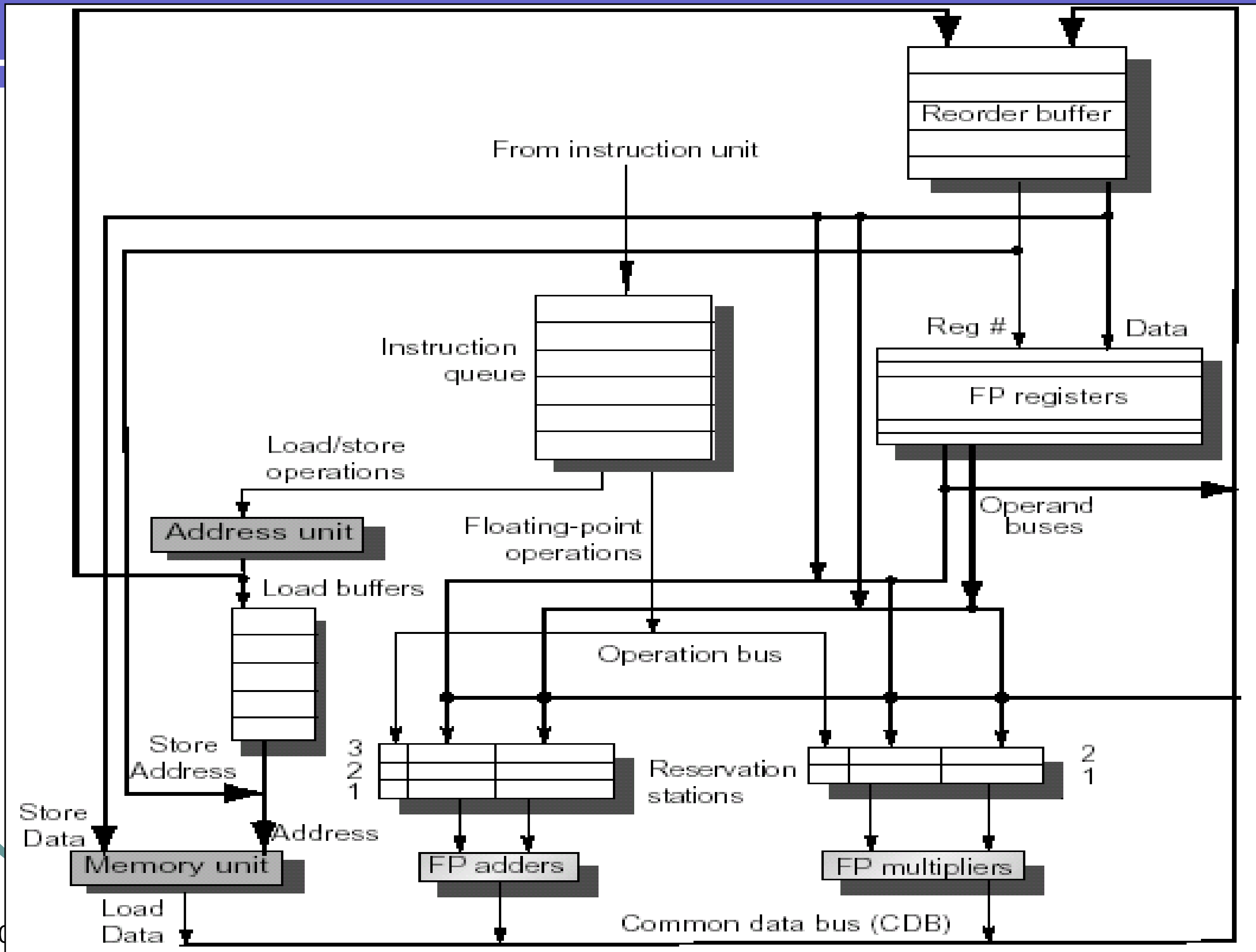
重构序缓存的作用

- 相当于一个额外虚拟寄存器，相当于**Tomasulo**算法中的保存站、**Load buffer**和**store buffer**等的功能。（注意，在基于**Tomasulo**的投机硬件中，取消了**Store buffer**部件**)**。
- 重组缓存在指令完成操作之后直到交付之前这段时间里保存该指令的结果，作为其它指令操作数的源，类似于**Tomasulo**算法中保留站作用。不同之处在于：**Tomasulo**算法中在**Write Result**这一拍中就可以更新**register file,**而这里只能等到进入**Commit**节拍才能更新**register file**。

重构序缓存单元的结构：由四个域组成

- **指令类型域**：用来说明指令类型
  - 转移指令——无目标结果
  - Store——以存储器地址作为目标结果
  - Reg操作（ALU或Load)：以Reg.作为目标结果
- **目标域**：
  - 寄存器号（针对ALU和Load操作指令）
  - 存储器地址（针对Store指令）
- **值域**：用来存放指令的结果，直到指令进入交付节拍。
- **就绪域**:表明指令已经执行完毕,值已经就绪.

基于Tomasulo算法的投机技术的硬件结构



与Tomasulo算法硬件结构的不同之处

- 增加了重组缓存；
- 撤销了store buffer；
- 寄存器改名功能由重构序缓存（重组缓存号）来实现，而不再由保留站来完成；
- 保留站的功能仅为在指令发射到开始执行这段时间内保存指令的操作码和操作数；
- 用重组缓存单元号来标识指令的结果，而不再用保留站号来标识，因为每一指令在其交付前均在重构序缓存中有一单元。

指令执行四个节拍的功能（1）

1、Issue——

- **Get**指令，**Issue**指令进入保留站和重组缓存（如果都有空的话）；
- **send**操作数进入保留站，如果它们已在**FP Reg**或重组缓存就绪；
- **Update**控制项，指示**buffers**正在使用；

若保留站和重组缓存之一无空，则**Issue step is stalled**，直到两者都有空为止。

指令执行四个节拍的功能（2）

2、Execute——

- **waiting操作数，直到它们准备就绪；**
- **check是否存在RAW;**
- **Execute操作，当两个操作数都在保留站中就绪。**

3、Write result——

- **结果一旦就绪, Write 到CDB ；**
- **Send结果到重组缓存以及所有等待这一结果的保留站；**
- **Mark所相应的保留站空闲就绪。**

指令执行四个节拍的功能（3）

4、Commit——

进入**commit**节拍的条件

- 对于非投机指令，按**code**顺序进入（即什么时候可进入**Commit**节拍）；
- 对于投机指令，当它确认不再是投机指令时（实际上也是按**code**的顺序）。

进入Commit节拍后做什么？

- 对于非branch指令：当该指令在重组缓存中的位置移到顶部时（说明按代码顺序轮到它进入commit时），Update相应的目的寄存器（用其存储在值域里的结果值），或write存储单元，如果是Store指令的话。从重组缓存中Remove该指令；
- 对于预测branch出错的：当错误分支指令移到重组缓存顶部时，由于预测出错，指出已投机的指令出错，清除重组缓存中位于该Branch指令后的所有指令，重新根据Br.转移方向启动代码；
- 对于预测正确的Branch:当该指令移到重组缓存的顶部时，由于预测正确，Branch指令结束；

小结：

- 当指令交付时
  - **它所占据的重组缓存单元被收回，即该指令从重构序缓存中撤销；**
  - **该指令的目的寄存器，或存储单元被更新。对于预测出错，即投机失败指令，将重组缓存中的Br.后的指令均清除，重新按正确转移方向启动代码；**

三、实例一

- 已知：

```
L.D      F6, 34(R2)
L.D      F2, 45(R3)
MUL.D    F0,  F2, F4
SUB.D    F8,   F6, F2
DIV.D    F10, F0, F6
ADD.D    F6,   F8, F2
```

给出当MUL.D 可进入Commit节拍时，硬件中状态表的结果。

保留站表（Ep110, Fig 2.15; Cp231, 表4-42)

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
Add1	No						
Add2	No						
Add3	No						
Mult1	No	MULT	Mem[45 +Regs[R3]	Regs[F4]			#3
Mult2	Yes	DIV		Mem[34 +Regs[R2]	#3		#5

重构序缓存状态

Entry	Busy	Instruction	State	Dest.	Value
1	No	L.D F6, 34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	NO	L.D F2, 45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	Yes	MUL.D F0, F2, F4	Wirte Result	F0	#2 * Regs[F4]
4	Yes	SUB.D F8, F6, F4	Write Result	F8	#1-#2
5	Yes	DIV.D F10, F0, F6	Execute	F10	
6	Yes	ADD.D F6, F8, F2	Write Result	F6	#4+#2

寄存器状态

Register Status									
Field	F0	F2	F4	F6	F8	F10	F12	…	F30
Reorder#	3			6	4	5			
Busy	Yes			Yes	Yes	Yes	No	…	No

注意（1）：

- 重构序缓存号的作用：**对应Qj和Qk，这里标明重组缓存号而不再是保留站的tag；**
- **Fig3.30中，实际已经完成Write result的有MUL.D, SUB.D和ADD.D，但由于MUL.D尚未完成Commit，故SUB.D和ADD.D必须等待MUL.D交付后才能依次Commit。这就是允许精确中断的原因（因为可以恢复）。**

注意（2）

- 动态调度行为：**SUB.D和ADD.D可早于MUL.D完成Write result;**
- 中断处理：**假设MUL.D指令产生一中断事件，这时只有等待MUL.D到达重组缓存顶部，才可以处理这一中断事件，同时，将缓存中其他指令清除掉，这样做的中断处理是精确的。**

在Tomasulo算法中，由于SUB.D和ADD.D可早于MUL.D结束，因此当处理MUL.D的中断时，由于F8,F6均已更新过，从而无法实现精确的中断处理。

小结

- 在指令Commit以前，不处理该指令的异常事件；
- 对正常指令和投机成功指令，在进入Commit后，处理中断；
- 对投机失败指令，则清除重组缓存中Br.指令以后的所有指令，重新按正确方向取指，重新开始执行。

实例2：

- 设有某Loop： L.D F0, 0(R1)

```
L.D     F0, 0(R1)
MUL.D   F4, F0, F2
S.D     0(R1), F4
DADDIU  R1, R1, #8
BNE     R1, R2,Loop
```

并且已经将两次迭代的指令都已issue到保留站和重构序缓存器中。这样做法意味着已经在进行投机，即已假设BNE将成功。

保留站表（Ep111, Fig 2.16; Cp231, 表4-43)

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
Mult1	No	MULT	Mem[0+Regs[R1]]	Regs[F2]			#2
Mult2	No	MULT	Mem[0+Regs[R1]]	Regs[F2]			#7

重构序缓存状态

Entry	Busy	Instruction	State	Dest.	Value
1	No	L.D F0, 0(R1)	C.	F0	Mem[0+Regs[R1]]
2	NO	MUL.D F4,F0,F2	C.	F4	#1*Regs[F2]
3	Yes	S.D 0(R1), F4	W. R.	0+Regs[R1]	#2
4	Yes	DADDIU R1,R1, #-8	W. R.	R1	Regs[R1]-8
5	Yes	BNE R1,R2, Loop	W. R.		
6	Yes	L.D F0, 0(R1)	W. R.	F0	Mem[#4]
7	Yes	MUL.D F4,F0,F2	W. R.	F4	#6*Regs[F2]
8	Yes	S.D 0(R1), F4	W. R.	0+ #4	#7
9	Yes	DADDIU R1,R1, #-8	W. R.	R1	#4-8
10	Yes	BNE R1,R2, Loop	W. R.		

寄存器状态

Register Status									
Field	F0	F2	F4	F6	F8	F10	F12	…	F30
Reorder#	6		7						
Busy	Yes	No	Yes	No	No	No	No	…	No

四、关于投机失败处理

- **如果前一个BNEZ为不成功，即预测出错，这时如何处理？**
- 等到该BNEZ移到重组缓存顶部时，
  - **将整个重组缓存清零，**
  - **处理器重新按正确转移路径取指，重新开始执行，而实际做法是尽可能提早把该BNEZ指令后的所有指令清除掉，提早从新的转移方向上取指令。这里所谓提早是指不必等到BNEZ移到重组缓存顶部在开始上述两步。**

基于Tomasulo算法投机技术的形式化描述

- 见p113, Fig2.17/ Cp233, 表4-44。

3.9 Taking Advantage of More ILP with Multiple Issue(2.7)

3.9.1 指令多发射技术的基本概念

一、基本概念

- 迄今为止介绍的各类提高性能的技术都是围绕使**CPI=1**这一目标展开的。
  - 如：流水线中消除数据相关、控制相关、静态调度、动态调度等
- 根据公式**CPUtime=IC×CPI ×cycle time**，进一步提高性能的启发是使**CPI <1**

CPI=1 ➡ Multiple-Issue ➡ CPI<1

基本概念

- 在传统每一周期发射一条指令的系统中,是无法实现**CPI<1**的。也就是说，要达到**CPI<1**,必须要求实现在一个时钟周期里发射多条指令，即指令的多发射技术。
- 多发射技术的两种方法(Two basic flavors)：
  - **Superscalar(超标量）**方法
  - **VLIW(超长指令字）**方法
- 实现指令多发射技术的前提：
  - 有足够硬件，即功能单元、寄存器、及存储器带宽的基础上。也就是说不存在结构竞争。

二、Superscalar的基本概念

- 在一个周期里能发射可变数量的指令，通常为**1-8**条指令**/cycle**；
- 同时发射的指令按一定规律搭配，即有一定限制，不能自由搭配；
- 用静态调度（**compiler**完成）和**/**或动态调度（硬件完成）方法确定可同时发射的指令条数。
  - Statically scheduled—in-order execution
  - Dynamically scheduled—out-of-order execution

三、VLIW的基本概念

- 在一个时钟周期里发射固定数量的指令，实际为一条长指令，或固定的指令包；
- **VLIW也是按固定格式组织的；**
- **VLIW是由Compiler组织的，（将在后面分析）**
  - Issue a fixed number of instruction formatted as one large instruction
  - A fixed instruction packet with a parallelism among instructions explicitly indicated by the instruction (EPIC—explicitly parallel instruction computer)

3.9.2 Statically-Scheduled Superscalar Processors

一、基本概念

- **Typical issue 0~8 instructions in a clock cycle with the hardware**
- **In a statically-scheduled superscalar**
  - **instructions issue in order**
  - **all pipeline hazards are checked for at issue**
- **The issue checks are sufficiently complex**
  - **performing the works/1CLK could mean that the issue logic determined the minimum clock cycle length**
  - **the issue stage is split and pipelined, so that it can issue instructions every clock cycle(2 stage)**
    - **to be higher branch penalties**

二、A Statically Scheduled Superscalar MIPS Processor

1. 结构

设：**2-issue** 整数指令：*Load/store,Branch,ALU*

浮点指令：*Fp*

激发条件：两条同时发射的指令必须是独立的

无数据竞争

无结构竞争

激发过程：从**cache**取**2**条指令

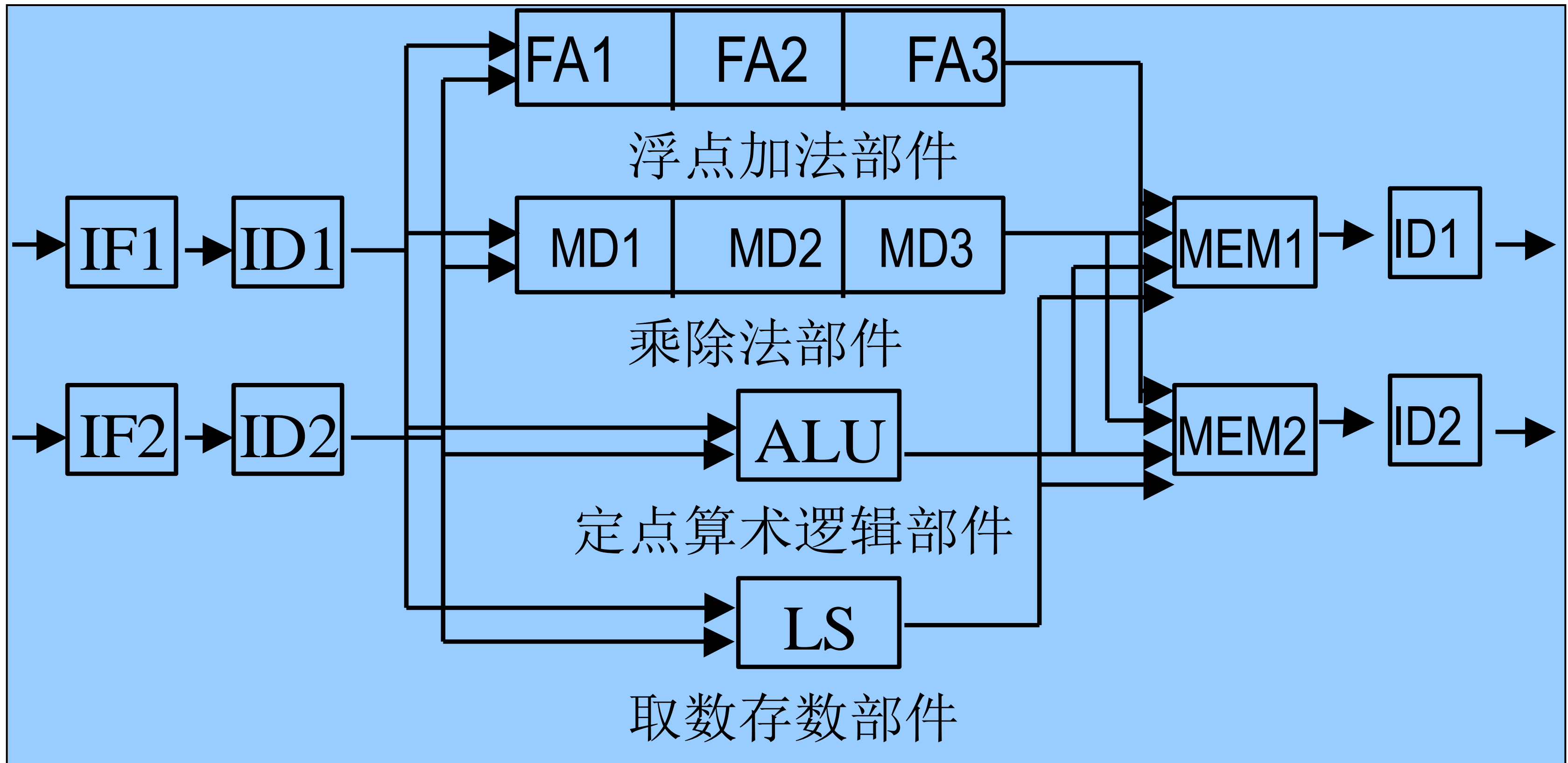
决定有几条指令可发射

激发到正确的功能单元

2. 双发射处理器的流水时序

指令类型	Pipe stages							
整数指令	IF	ID	EX	MEM	WB			
浮点指令	IF	ID	EX	MEM	WB			
整数指令		IF	ID	EX	MEM	WB		
浮点指令		IF	ID	EX	MEM	WB		
整数指令			IF	ID	EX	MEM	WB	
浮点指令			IF	ID	EX	MEM	WB	
整数指令				IF	ID	EX	MEM	WB
浮点指令				IF	ID	EX	MEM	WB

双发射流水线结构示意图



三、竞争的处理

1.当整数指令为Load/Store/Move浮点数时，可能造成

- **在FP register file处出现**寄存器口的竞争
- **与下一条FP操作指令可能有**RAW数据竞争

解决方法

- **当出现上述情况时，把它作为结构竞争处理，即不允许出现此类搭配**
- **将FP register file做成Read/Write双口寄存器，允许同时访问不同FP registers.**

2. **Load浮点数的结果不能被同一周期的另一条指令所应用，实际上也不能被下一周期的两条指令所应用。**因为**LD后跟FP操作之间有一个Stall周期。由于这里每一个周期发射两条指令，因此LD的结果不能被紧接的三条指令所用。**

3. **同理，由于Br.指令后存在一个周期的Br. Delay，所以也要影响三条指令，即要用三条不相关指令来填充这一延时槽，而不是传统处理器中只需填一条指令。**

所以需要更强的编译调度和硬件调度策略

4. Larger set of bypass paths will be needed

5. 一对指令可能来自不同的cache块

- 要用独立的取指单元

6. 中断：精确中断的困难

- A floating point instruction can finish execution after an integer instruction that is later in program order
- The floating point instruction exception could be detected after the integer instruction completed
- Solutions
  - Early detection of FP exceptions
  - The use of software mechanisms to restore a precise exception state
  - Delaying instruction completion until we know an exception is impossible（the speculation approach）

3.9.3 Multiple Instruction Issue with Dynamic Scheduling

多发射技术也可采用动态调度方法（**Scoreboard**、**Tomasulo)**来决策可同时发射的指令。

现将**Tomasulo**动态调度算法扩展到支持每个周期同时发射两条指令的多发射机制。

约定

- 两条同时发射的指令搭配：一条为整数操作，一条为**FP**操作；
- 按序发射，而非乱序发射；
- 采用独立的整数寄存器堆和**FP**寄存器堆，使同时发射的两条指令可同时进入对应的保留站，可分别同时访问对应的寄存器堆。

如何处理相邻的两条相关指令？

- 在非动态调度的多发射处理器钟，由compiler作静态调度，选择两条非相关指令同时发射同时执行。
- 在采用Tomasulo动态调度算法时，可以按序，按搭配规定同时发射两条指令，由硬件（保留站等）自动解决相关性问题，即乱序执行，乱序结束。

例

```
Loop:  L.D       F0, 0(R1)
       ADD.D F4, F0, F2
       S.D       0(R1), F4
       DADDIU  R1, R1, #-8
       BNE  R1,R2, Loop
```

设:　2-issue，branches单一发射，预测正确

"latency":ALU—1,load—2,Fp.ADD—3

CDB: 2个

ALU和有效地址运算用同一个整数单元

双发射Tomasulo流水线

Iter. #	Instructions	Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First issue
1	ADD.D F4,F0,F2	1	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1,R1,#-8	2	4		5	Wait for ALU
1	BNE R1,R2,Loop	3	6			Wait for DADDIU
2	L.D F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D F4,F0,F2	4	10		13	Wait for L.D
2	S.D F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU R1,R1,#-8	5	9		10	Wait for ALU
2	BNE R1,R2,Loop	6	11			Wait for DADDIU
3	L.D F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D F4,F0,F2	7	15		18	Wait for L.D
3	S.D F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU R1,R1,#-8	8	14		15	Wait for ALU
3	BNE R1,R2,Loop	9	16			Wait for DADDIU

结果

- **注意：**
  - 这里是按序发射的，且由硬件进行组合，按搭配要求进行多发射；
  - 由于FP操作指令少，所以多发射次数少；
  - 这里是不按顺序执行，也不按顺序写结果。
- **结论：**
  - **9 cycles/3 iteration = 3 cycles/iteration**
  - **one iteration every three cycles would lead to an IPC=5/3=1.67**
  - **Instruction completion rate is 15/16 = 0.94.**
  - 如果增加整数部件，则可提高多发射机会，达到加速目的。

Clock #	Integer ALU	FP ALU	Data Cache	CDB
2	1/ L.D			
3	1 / S.D		1/ L.D	
4	1 / DADDIU			1/ L.D
5		1 / ADD.D		1 / DADDIU
6				
7	2 / L.D			
8	2 / S.D		2 / L.D	1 / ADD.D
9	2 / DADDIU		1 / S.D	2 / L.D
10		2 / ADD.D		2 / DADDIU
11				
12	3 / L.D			
13	3 / S.D		3 / L.D	2 / ADD.D
14	3 / DADDIU		2 / S.D	3 / L.D
15		3 / ADD.D		3 / DADDIU
16				
17				
18				3 / ADD.D
19			3 / S.D	
20				

Iter. #	Instructions	Issues at	Executes		Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	2	3	4	First issue
1	ADD.D F4,F0,F2	1	5	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	3	9		Wait for ADD.D
1	DADDIU R1,R1,#-8	2	3	4		4	Executes earlier
1	BNE R1,R2,Loop	3	5	6			Wait for DADDIU
2	L.D F0,0(R1)	4	6	7	7	8	Wait for BNE complete
2	ADD.D F4,F0,F2	4	9	10		12	Wait for L.D
2	S.D F4,0(R1)	5	7	8	13		Wait for ADD.D
2	DADDIU R1,R1,#-8	5	6	9		7	Executes earlier
2	BNE R1,Loop	6	8	11			Wait for DADDIU
3	L.D F0,0(R1)	7	9	12	10	11	Wait for BNE complete
3	ADD.D F4,F0,F2	7	12	15		15	Wait for L.D
3	S.D F4,0(R1)	8	10	13	16		Wait for ADD.D
3	DADDIU R1,R1,#-8	8	9	14		10	Executes earlier
3	BNE R1,Loop	9	11	16			Wait for DADDIU

FIGURE： The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline with separate functional units for integer ALU operations and effective address calculation, which also uses a wider CDB. **The extra integer ALU allows the DADDIU to execute earlier, in turn allowing the BNE to execute earlier, and, thereby, starting the next iteration earlier.**

- Three factors limit the performance of the two-issue dynamically scheduled pipeline:
  - **There is an imbalance between the functional unit structure of the pipeline and the example loop. This imbalance means that it is impossible to fully use the FP units. To remedy this, we would need fewer dependent integer operations per loop.**
  - **The amount of overhead per loop iteration is very high: two of out of five instructions (the `DADDIU` and the `BNE`) are overhead. In the next chapter we look at how this overhead can be reduced.**
  - **The control hazard, which prevents us from starting the next `L.D` before we know whether the branch was correctly predicted, causes a one-cycle penalty on every loop iteration. The next section introduces a technique that addresses this limitation.**(投机)

3.9.4 Multiple Issue with Speculation

Iter. #	Instructions	Issues at clock cycle #	Executes at clock cycle #	Memory access at clock cycle #	Write CDB at clock cycle #	Comment
1	LW R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SW 0(R1),R2	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LW R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SW 0(R1),R2	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LW R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SW 0(R1),R2	8	19	20		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

- **FIGURE： The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation.**

FIGURE： The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation.

Note that the L.D following the BNE can start execution early, because it is speculative.

Iter. #	Instructions	Issues at clock #	Executes at clock #	Read access at clock #	Write CDB at clock #	Commits at clock #	Comment
1	LW R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SW 0(R1),R2	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for ADDDI
2	LW R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SW 0(R1),R2	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LW R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SW 0(R1),R2	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU