

 北京大学计算机本科生课程

计算机组成与 系统结构实习

 **Review 3&4**

北京大学微处理器研发中心

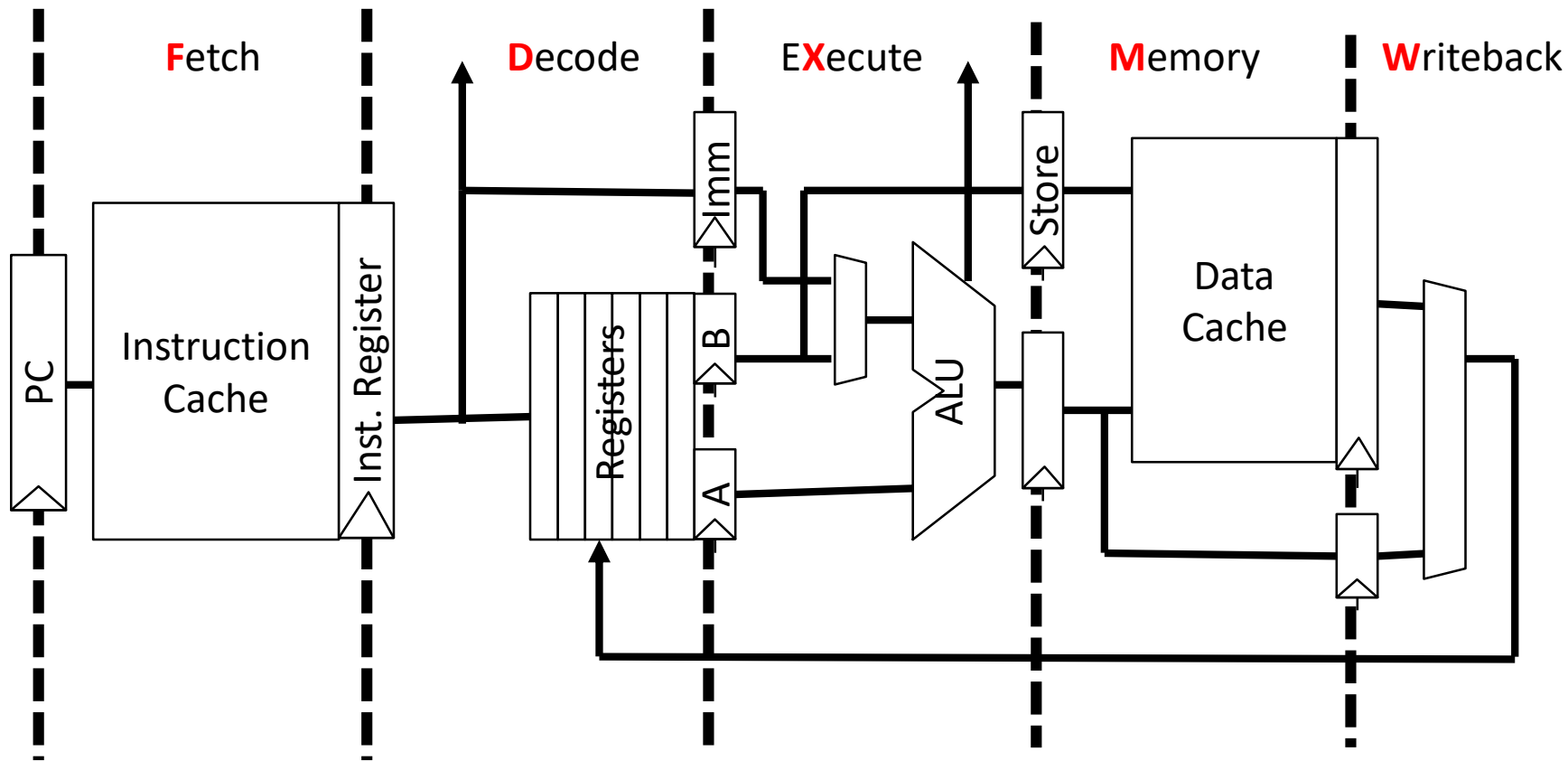
易江芳

2022-10-17

主要内容

- 重温流水线
- 超深流水线
- 挖掘指令级并行
 - 数据流：超标量，乱序
 - 控制流：分支预测

重温流水线



- 经典的RISC五级流水线实现

重温流水线

- 流水线中的三种冒险
- 流水线中的数据冒险
- 解决数据冒险的三种策略
- 延迟转移 和 延迟装入



上述流水线能不能每个周期都执行一条指令？

■ 流水线中的三种冒险 (hazard)

- 相关 (relative) vs. 冒险 (hazard)
 - 相关：程序数据之间的固有关系
 - 冒险：相关在特定微结构中引发的资源冲突
- 简单流水线中的三种冒险
 - 结构冒险
 - 数据冒险
 - 控制冒险

流水线中的数据冒险

考虑如下的操作类型

$$r_k \leftarrow r_i \text{ op } r_j$$

真相关 (数据相关, Data-dependence)

$$\begin{array}{l} r_3 \leftarrow r_1 \text{ op } r_2 \\ r_5 \leftarrow r_3 \text{ op } r_4 \end{array} \quad \longrightarrow \quad \text{Read-after-Write (RAW) hazard}$$

反相关 (Anti-dependence)

$$\begin{array}{l} r_3 \leftarrow r_1 \text{ op } r_2 \\ r_1 \leftarrow r_4 \text{ op } r_5 \end{array} \quad \longrightarrow \quad \text{Write-after-Read (WAR) hazard}$$

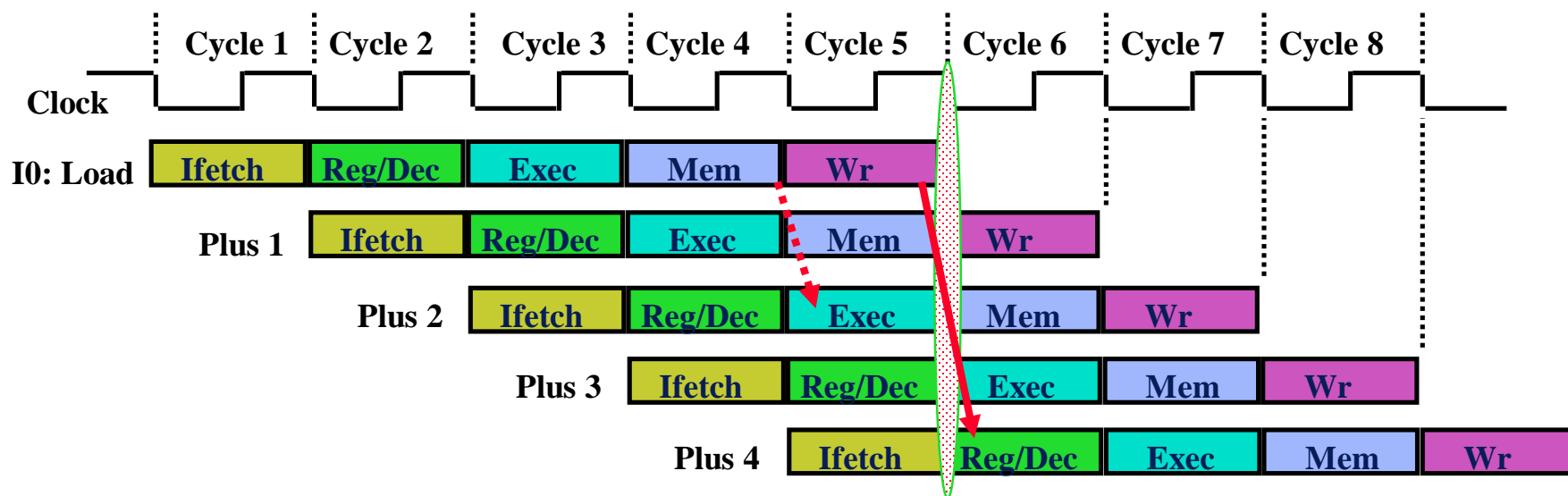
输出相关 (Output-dependence)

$$\begin{array}{l} r_3 \leftarrow r_1 \text{ op } r_2 \\ r_3 \leftarrow r_6 \text{ op } r_7 \end{array} \quad \longrightarrow \quad \text{Write-after-Write (WAW) hazard}$$

解决数据冒险的三个策略

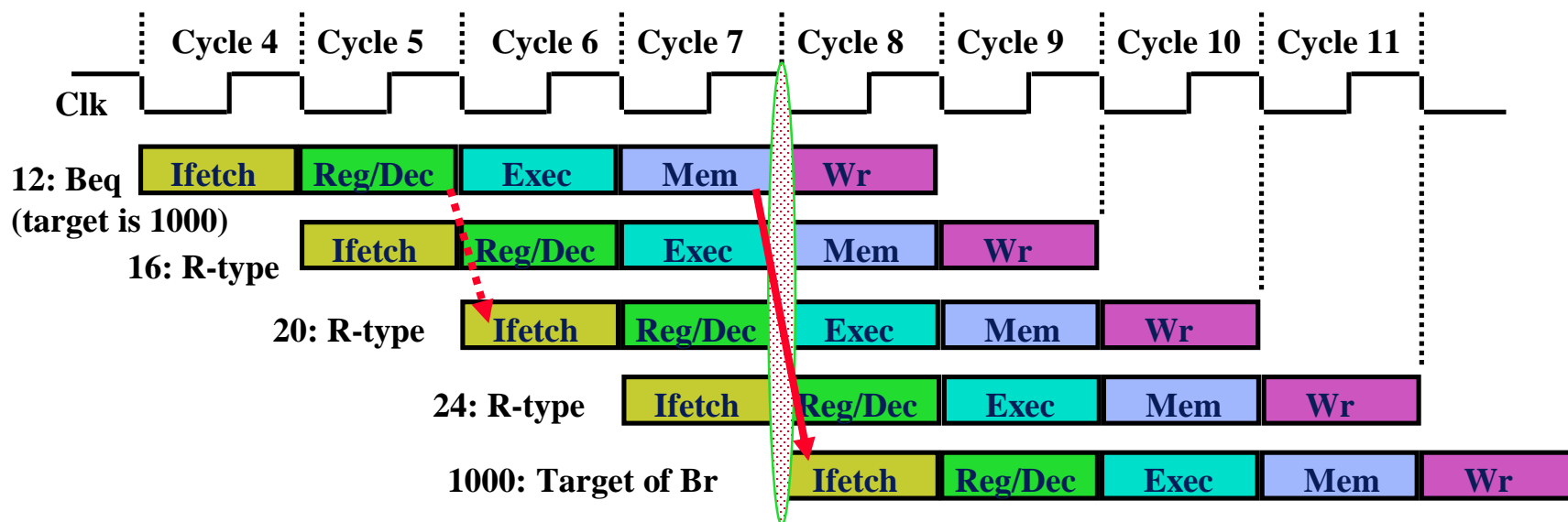
- 停顿 (stall)
- 加气泡 (bubble)
- 旁路 (bypassing)

延迟装入 (delayed load)



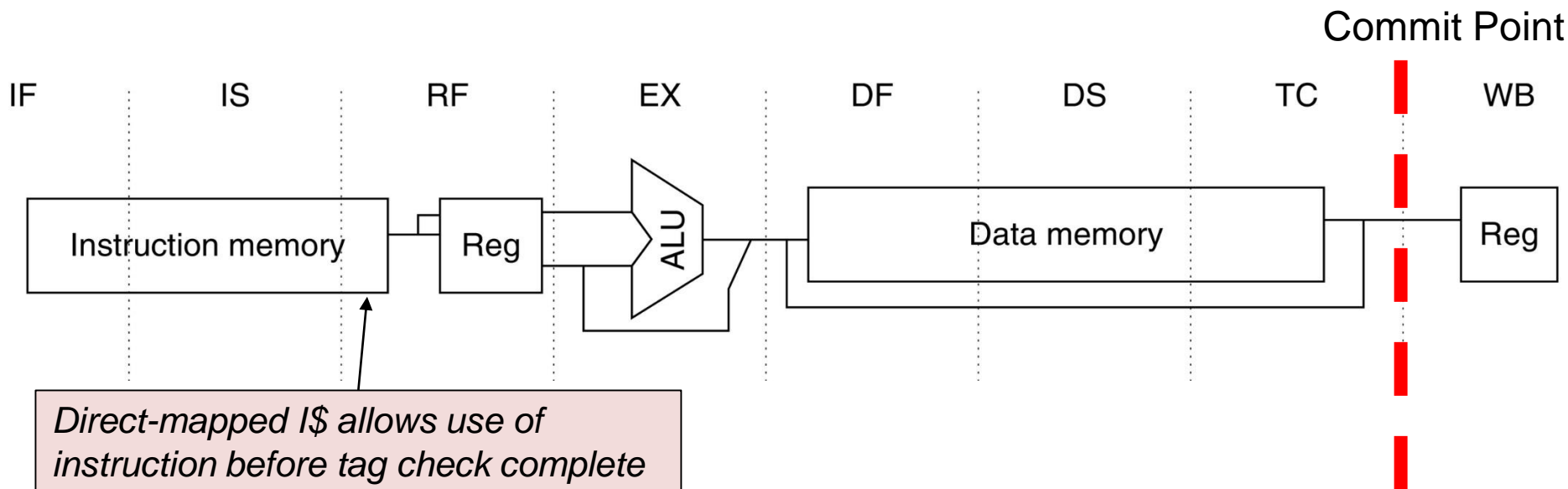
- 通过编译器来减少数据冒险的代价
- MIPS ISA设计理念：微结构信息软件可见
- 虽然Load在第一个周期就被取指：
 - 它的数据直到第五个周期结束时才被写入到寄存器堆
 - 在第六个周期之前, 不可能从寄存器堆中读取这一数值
 - 在装入指令结束之前, 有3条指令被延迟

延迟转移 (delayed branch)



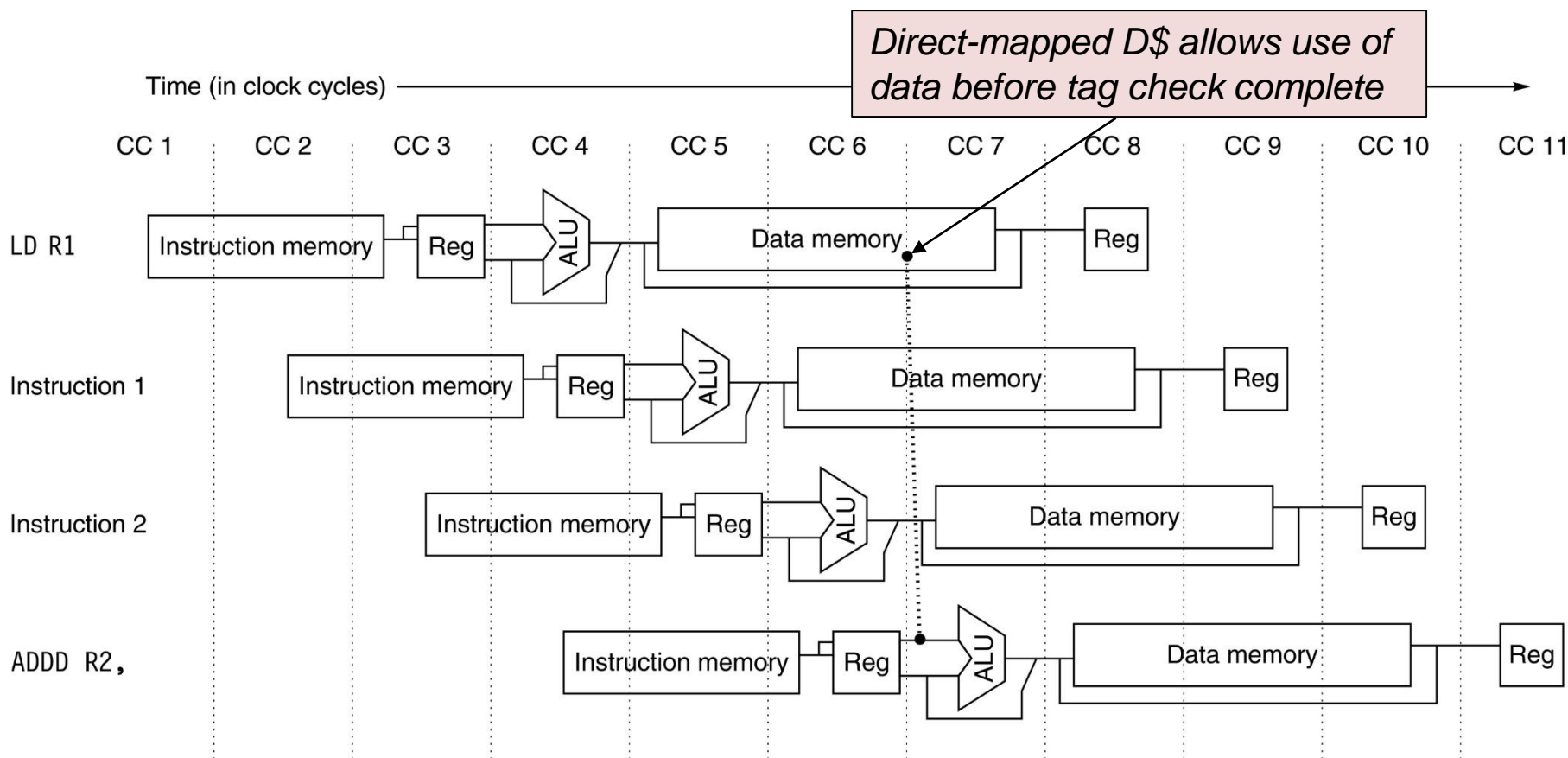
- 通过编译器来减少控制冒险的代价
- 虽然Load在第一个周期就被取指
 - 它的数据直到第五个周期结束时才被写入到寄存器堆
 - 在第六个周期之前, 不可能从寄存器堆中读取这个数
 - 在装入指令结束之前, 有3条指令被延迟

超深流水线：以MIPS R4000为例



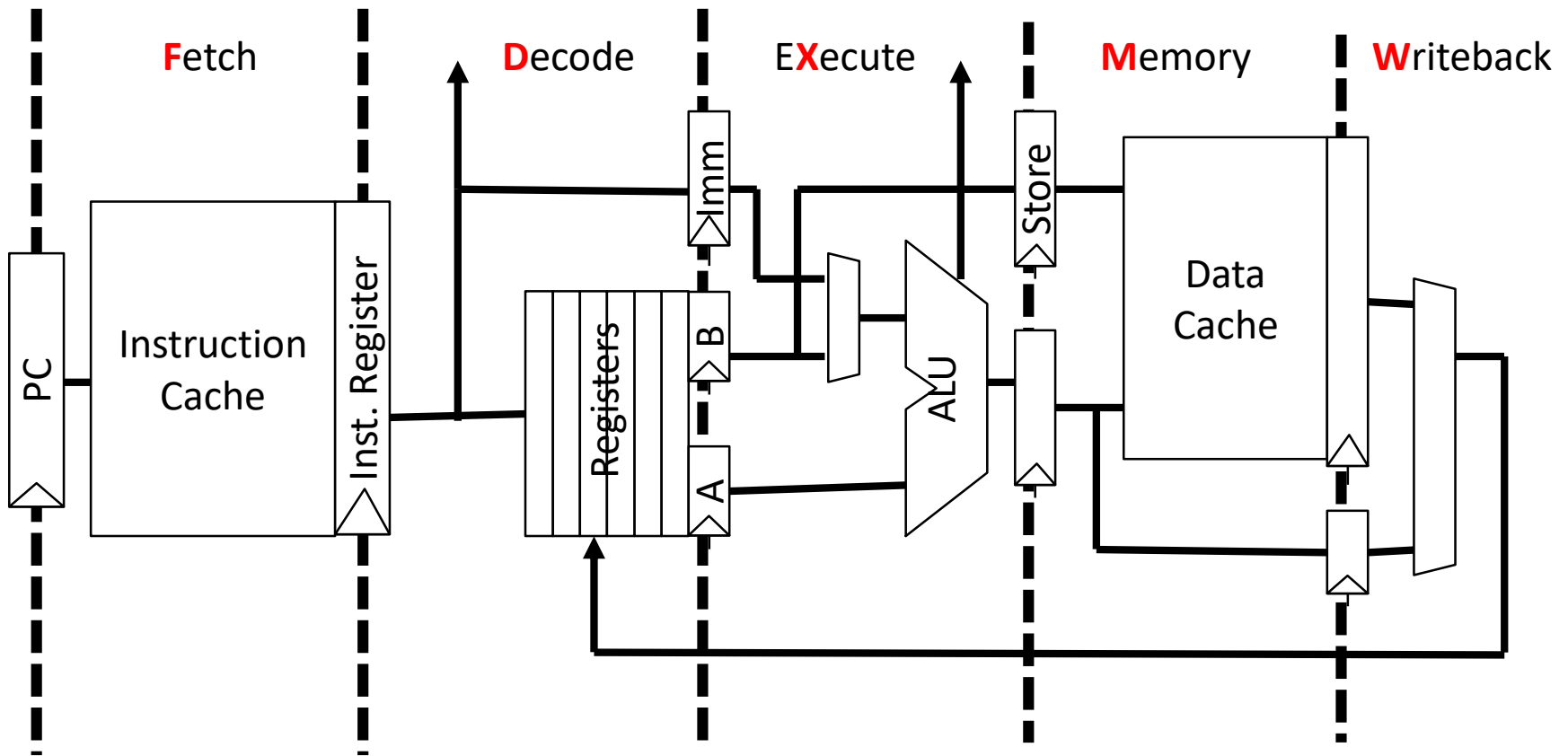
- Figure C.36 MIPS R4000的8级流水线结构，其中I\$和D\$也是流水的
- IS 阶段的末尾，指令被取出可用，但指令地址Tag比对需在RF阶段完成，因此，实际上对 I\$的访问贯穿到了RF阶段。
- 但是，对于D\$，必须在TC阶段完成访问，因为Cache是否命中决定了是否能够改写寄存器堆

R4000的 load-use 延迟



- 可以考虑在 load 指令的 DS 阶段末尾增加 bypass，旁路到寄存器堆的读口上。如果 load 指令在 TC 阶段发现 Cache miss，则整个流水线需要进行恢复（ADDD 指令的 ALU 操作无效）

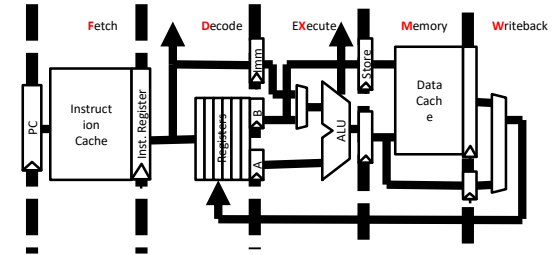
简单流水线



- 经典的RISC五级流水线实现

挖掘指令级并行

- 体系课上的经典例子



```
#      for(i=0; i<N; i++)  
#          A[i] = B[i]+C[i];
```

```
loop: fld f0, 0(x2) // x2 points to B  
      fld f1, 0(x3) // x3 points to C  
      fadd.d f2, f0, f1  
      fsd f2, 0(x1) // x1 points to A  
      addi x1, x1, 8// Bump pointer  
      addi x2, x2, 8// Bump pointer  
      addi x3, x3, 8// Bump pointer  
      bne x1, x4, loop // x4 holds end
```

流水线性能分析

现象：

- 访存操作、浮点计算操作都是多个执行周期（3个），这使得指令执行过程中的停顿过多
- 浮点和定点操作交替出现
- 同一类功能的指令同时出现多个

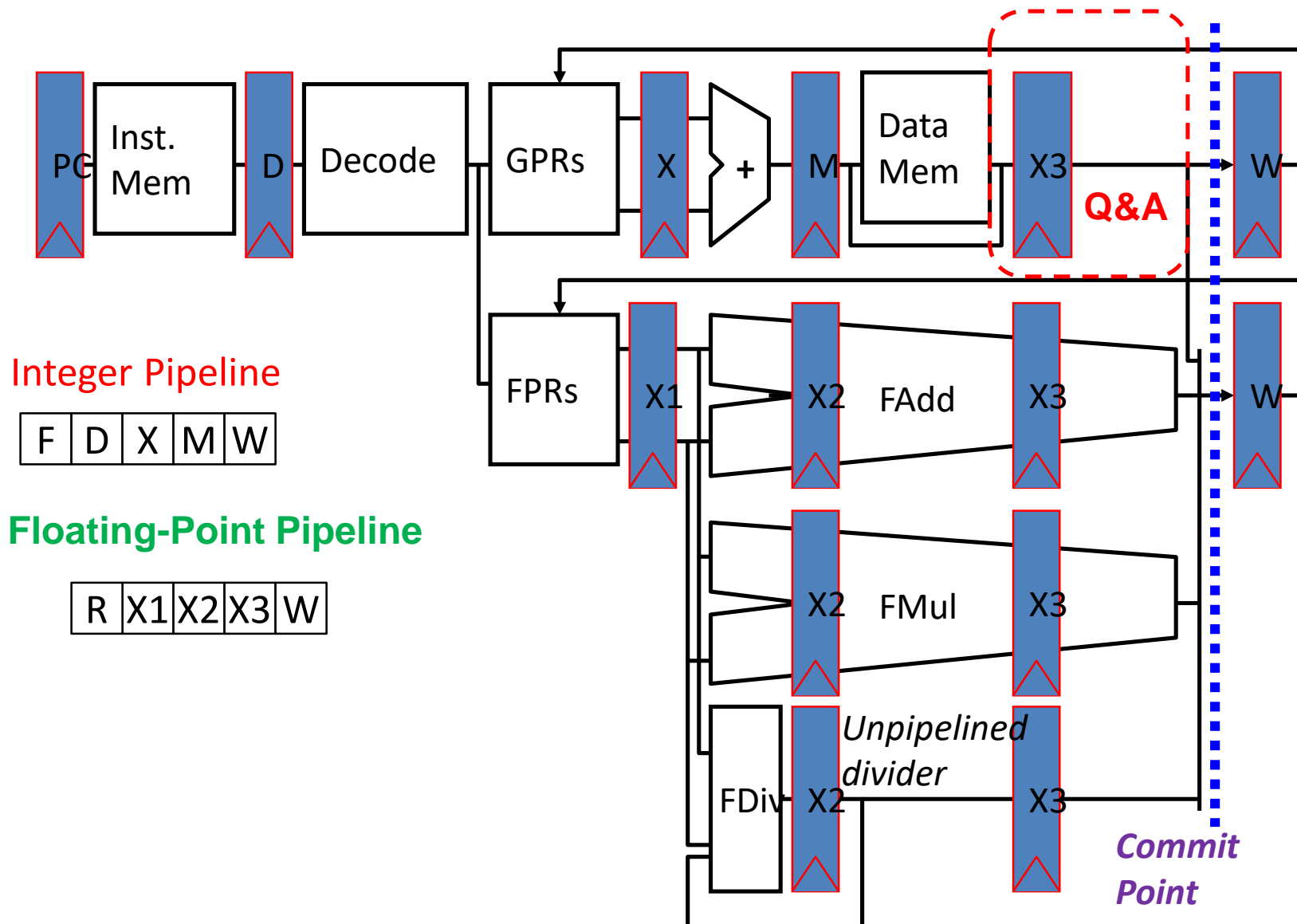
再仔细观察一下：

- 两个 fld 之间其实没有关系，可以考虑优化
- 三个 addi 之间也没有关系，可以考虑优化
- fadd 和 fsd 之间，寄存器 f2 可以考虑最后一个周期读出来

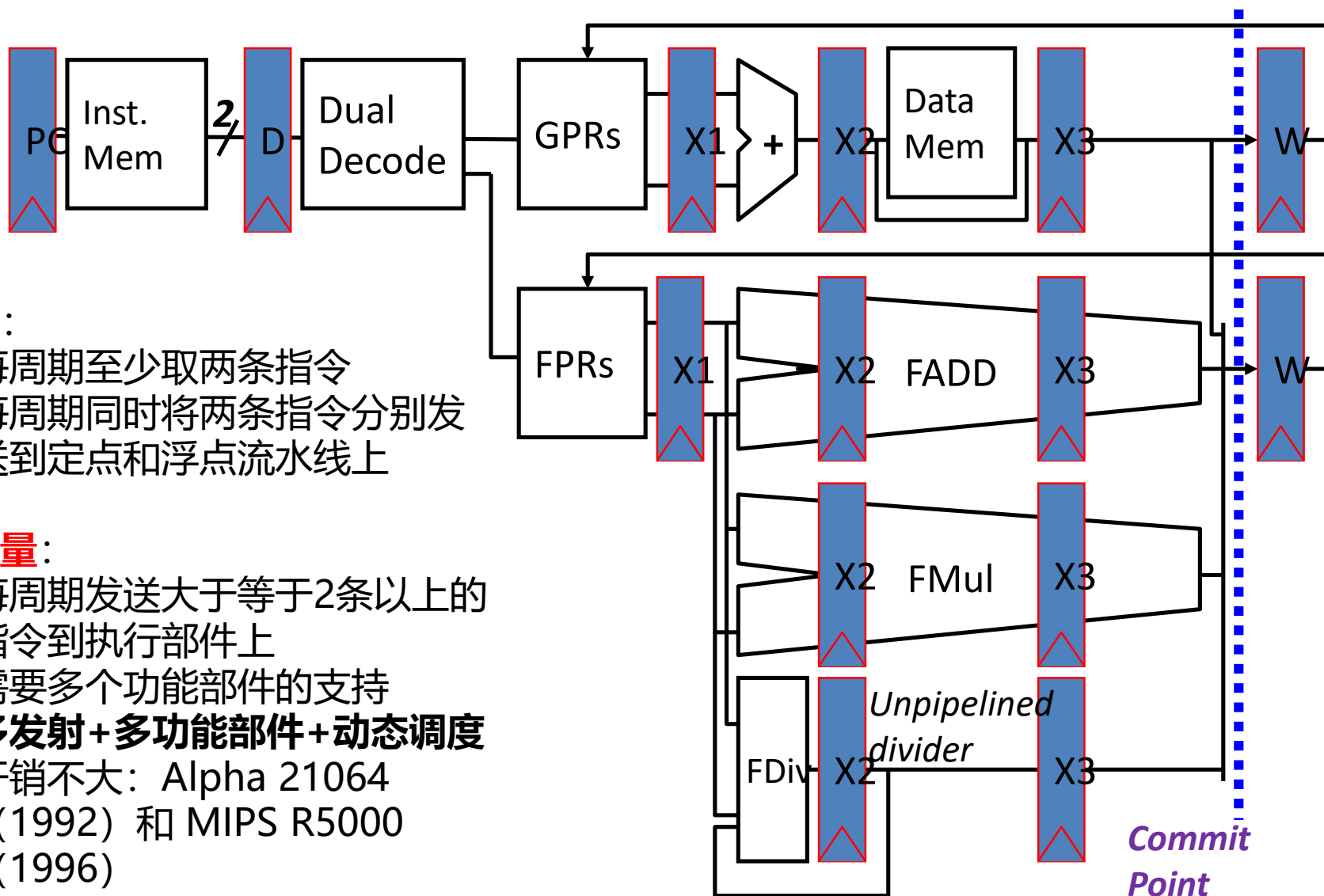
问题：

- 指令功能各不相同，是否可以将他们区分开来？（之前就是一个ALU）
- 同一功能的部件是否只能有一个？（之前只有一个）
- 功能部件的设计是否都是单周期完成的？（之前都是）

1、复杂的按序单发射流水线



2、按序超标量流水线微结构



软件调度：循环展开(Loop Unrolling)

double A = B + C

```
loop:  fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fadd.d f2, f0, f1
      fsd f2, 0(x1) // x1 points to A
      addi x1,x1,8    // Bump pointer
      addi x2,x2,8    // Bump pointer
      addi x3,x3,8    // Bump pointer
      bne x1, x4, loop // x4 holds end
```

```
loop:  fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fld f10, 8(x2)
      fld f11, 8(x3)
      addi x3,x3,16    // Bump pointer
      addi x2,x2,16    // Bump pointer
      fadd.d f2, f0, f1
      fadd.d f12, f10, f11
      addi x1,x1,16    // Bump pointer
      fsd f2, -16(x1) // x1 points to A
      fsd f12, -8(x1)
      bne x1, x4, loop // x4 holds end
```

- 观察1：循环展开的程度受体系结构寄存器数量的影响
- 观察2：占用指令缓存的空间
- 观察3：对编译器的能力提出挑战，特别是对指针类型的识别和处理

3、动态调度减少冒险

```
loop: fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fadd.d f2, f0, f1
      fsd f2, 0(x1) // x1 points to A
      addi x1, x1, 8 // Bump pointer
      addi x2, x2, 8 // Bump pointer
      addi x3, x3, 8 // Bump pointer
      bne x1, x4, loop // x4 holds end
```

- **发现：**当一个循环体内存在真相关的长延迟操作时，指令执行的并行度严重下降。
- 采用超标量技术：多发射+多功能部件+动态调度





动态调度？

动态调度造成乱序

```
loop: fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fadd.d f2, f0, f1
      fsd f2, 0(x1) // x1 points to A
      addi x1, x1, 8 // Bump pointer
      addi x2, x2, 8 // Bump pointer
      addi x3, x3, 8 // Bump pointer
      bne x1, x4, loop // x4 holds end
```

- 乱序：指令顺序不按照程序顺序操作
- 可以乱序的阶段：发射，执行（访存），完成，提交
- 微结构设计需要保证乱序不会带来程序执行错误

4*. 访存操作解耦合

- 进一步分析，访存指令可以拆分为 { 访存地址计算 + 真正的访存操作 } 两个微操作（简称 μop ），其中访存地址计算与浮点计算结果没有关系，可以提前执行。
- 这样，就将流水线中各操作划分成如下微操作（ μop ）
 - 浮点计算 μop  浮点操作
 - Load数据写回寄存器 μop （浮点 or 定点）  共用定点流水线
 - Store数据 μop （浮点 or 定点）  共用定点流水线
 - 计算访存地址 μop  定点操作

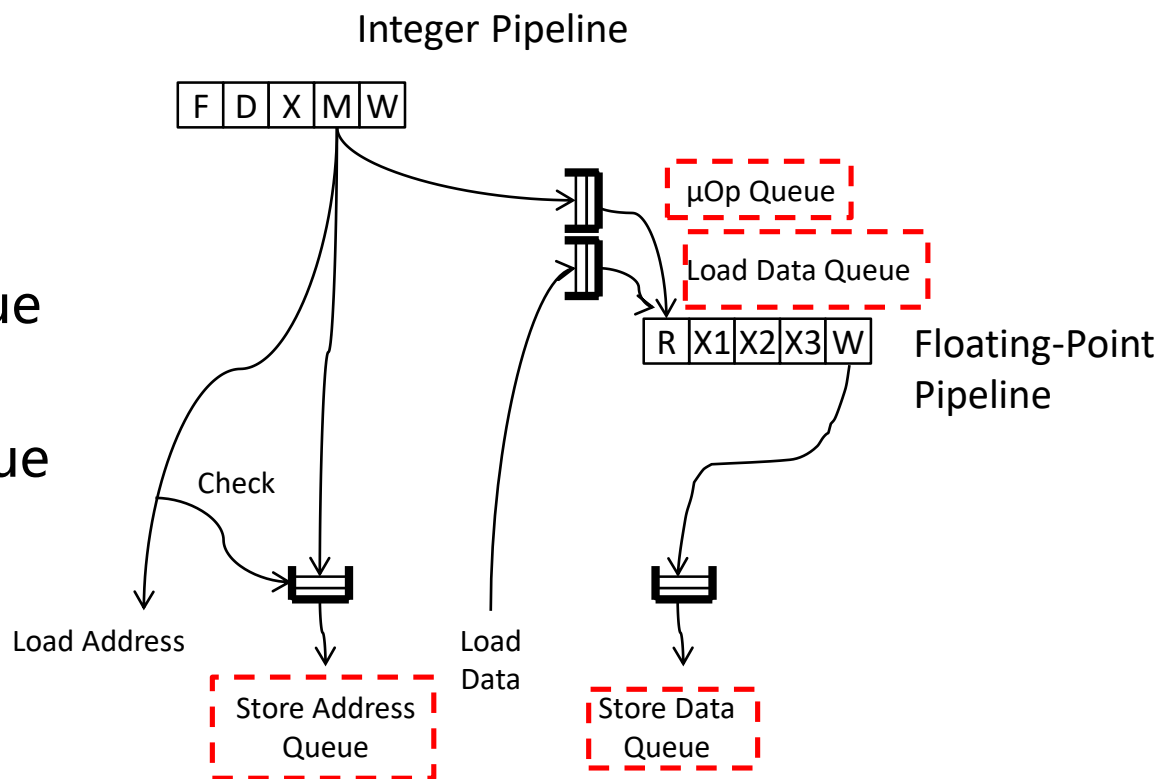
与浮点操作无关，可以提前完成
- 这样就足够了吗？？？



访存顺序问题

简单的解耦合

- 解决方案：增加4个Queue
 - μ op Queue
 - Load Address Queue
 - Store Address Queue
 - Store Data Queue



乱序执行

Cycle 1

<code>fld f0</code>

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f0

```
loop:    fld f0, 0(x2) // x2 points to B
         fld f1, 0(x3) // x3 points to C
         fadd.d f2, f0, f1
         fsd f2, 0(x1) // x1 points to A
         addi x1,x1,8      // Bump pointer
         addi x2,x2,8      // Bump pointer
         addi x3,x3,8      // Bump pointer
         bne x1, x4, loop // x4 holds end
```

μ op Queue

fld(f0)					
---------	--	--	--	--	--

Load Data Queue

--	--	--	--	--	--

Store Data Queue

--	--	--	--	--	--

Store Address Queue

--	--	--	--	--	--

乱序执行

Cycle 1

```
loop:    fld f0, 0(x2) // x2 points to B
         fld f1, 0(x3) // x3 points to C
         fadd.d f2, f0, f1
         fsd f2, 0(x1) // x1 points to A
         addi x1,x1,8      // Bump pointer
         addi x2,x2,8      // Bump pointer
         addi x3,x3,8      // Bump pointer
         bne x1, x4, loop // x4 holds end
```

<code>fld f0</code>

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f0



Q: 如果load数据没返回，怎么办？

乱序执行

Cycle 2

```
loop:    fld f0, 0(x2) // x2 points to B
         fld f1, 0(x3) // x3 points to C
         fadd.d f2, f0, f1
         fsd f2, 0(x1) // x1 points to A
         addi x1,x1,8      // Bump pointer
         addi x2,x2,8      // Bump pointer
         addi x3,x3,8      // Bump pointer
         bne x1, x4, loop // x4 holds end
```

fld f0
fld f1

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f1

μ op Queue

fld(f0)	fld(f1)				
---------	---------	--	--	--	--

Load Data Queue

--	--	--	--	--	--

Store Data Queue

--	--	--	--	--	--

Store Address Queue

--	--	--	--	--	--

乱序执行

Cycle 3

fld f0
fld f1
fadd.d

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f1

浮点计算 μ op 插入队列

```
loop:    fld f0, 0(x2) // x2 points to B
         fld f1, 0(x3) // x3 points to C
         fadd.d f2, f0, f1
         fsd f2, 0(x1) // x1 points to A
         addi x1,x1,8      // Bump pointer
         addi x2,x2,8      // Bump pointer
         addi x3,x3,8      // Bump pointer
         bne x1, x4, loop // x4 holds end
```

μ op Queue

fld(f0)	fld(f1)	Fadd			
---------	---------	------	--	--	--

Load Data Queue

--	--	--	--	--	--

Store Data Queue

--	--	--	--	--	--

Store Address Queue

--	--	--	--	--	--

乱序执行

Cycle 3

```
loop:    fld f0, 0(x2) // x2 points to B
         fld f1, 0(x3) // x3 points to C
         fadd.d f2, f0, f1
         fsd f2, 0(x1) // x1 points to A
         addi x1,x1,8      // Bump pointer
         addi x2,x2,8      // Bump pointer
         addi x3,x3,8      // Bump pointer
         bne x1, x4, loop // x4 holds end
```

<code>fld f0</code>
<code>fld f1</code>
<code>fadd.d</code>

计算load地址，向Mem发出读请求， μop 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求， μop 插入队列，等待数据返回写入f1

浮点计算 μop 插入队列



Q: 如果load数据
都没返回，怎么办？

乱序执行

Cycle 4

```
loop:      fld f0, 0(x2) // x2 points to B
           fld f1, 0(x3) // x3 points to C
           fadd.d f2, f0, f1
           fsd f2, 0(x1) // x1 points to A
           addi x1,x1,8      // Bump pointer
           addi x2,x2,8      // Bump pointer
           addi x3,x3,8      // Bump pointer
           bne x1, x4, loop // x4 holds end
```

fld f0
fld f1
fadd.d
fsd f2

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f1

浮点计算 μ op 插入队列

计算store地址，向Mem发出写请求， μ op 插入队列，等待写数据准备好

μ op Queue

fld(f0)	fld(f1)	Fadd	fsd		
---------	---------	------	------------	--	--

Load Data Queue

--	--	--	--	--	--

Store Data Queue

f2					
-----------	--	--	--	--	--

Store Address Queue

0(x1)					
--------------	--	--	--	--	--

乱序执行

Cycle 5

fld f0
fld f1
fadd.d
fsd f2
addi

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f1

浮点计算 μ op 插入队列

计算store地址，向Mem发出写请求， μ op 插入队列，等待写数据准备好
更新指针 x1

```
loop:      fld f0, 0(x2) // x2 points to B
           fld f1, 0(x3) // x3 points to C
           fadd.d f2, f0, f1
           fsd f2, 0(x1) // x1 points to A
           addi x1,x1,8          // Bump pointer
           addi x2,x2,8          // Bump pointer
           addi x3,x3,8          // Bump pointer
           bne x1, x4, loop // x4 holds end
```

μ op Queue

fld(f0)	fld(f1)	Fadd	fsd		
---------	---------	------	-----	--	--

Load Data Queue

--	--	--	--	--	--

Store Data Queue

f2					
----	--	--	--	--	--

Store Address Queue

0(x1)					
-------	--	--	--	--	--

乱序执行

Cycle 6

```

loop:      fld f0, 0(x2) // x2 points to B
           fld f1, 0(x3) // x3 points to C
           fadd.d f2, f0, f1
           fsd f2, 0(x1) // x1 points to A
           addi x1,x1,8          // Bump pointer
           addi x2,x2,8          // Bump pointer
           addi x3,x3,8          // Bump pointer
           bne x1, x4, loop // x4 holds end
    
```

fld f0
fld f1
fadd.d
fsd f2
addi
addi

计算load地址，向Mem发出读请求，μop 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求，μop 插入队列，等待数据返回写入f1

浮点计算μop 插入队列

计算store地址，向Mem发出写请求，μop 插入队列，等待写数据准备好

更新指针 x1

更新指针 x2

μop Queue

fld(f0)	fld(f1)	Fadd	fsd		
---------	---------	------	-----	--	--

Load Data Queue

--	--	--	--	--	--

Store Data Queue

f2					
----	--	--	--	--	--

Store Address Queue

0(x1)					
-------	--	--	--	--	--

乱序执行

Cycle 7

```
loop:      fld f0, 0(x2) // x2 points to B
           fld f1, 0(x3) // x3 points to C
           fadd.d f2, f0, f1
           fsd f2, 0(x1) // x1 points to A
           addi x1,x1,8          // Bump pointer
           addi x2,x2,8          // Bump pointer
           addi x3,x3,8          // Bump pointer
           bne x1, x4, loop // x4 holds end
```

fld f0
fld f1
fadd.d
fsd f2
addi
addi
addi

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f1

浮点计算 μ op 插入队列

计算store地址，向Mem发出写请求， μ op 插入队列，等待写数据准备好

更新指针 x1

更新指针 x2

更新指针 x3

μ op Queue

fld(f0)	fld(f1)	Fadd	fsd		
---------	---------	------	-----	--	--

Load Data Queue

--	--	--	--	--	--

Store Data Queue

f2					
----	--	--	--	--	--

Store Address Queue

0(x1)					
-------	--	--	--	--	--

乱序执行

Cycle 8

```

loop:      fld f0, 0(x2) // x2 points to B
           fld f1, 0(x3) // x3 points to C
           fadd.d f2, f0, f1
           fsd f2, 0(x1) // x1 points to A
           addi x1,x1,8           // Bump pointer
           addi x2,x2,8           // Bump pointer
           addi x3,x3,8           // Bump pointer
           bne x1, x4, loop // x4 holds end
    
```

fld f0
fld f1
fadd.d
fsd f2
addi
addi
addi
bne

计算load地址，向Mem发出读请求，μop 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求，μop 插入队列，等待数据返回写入f1

浮点计算μop 插入队列

计算store地址，向Mem发出写请求，μop 插入队列，等待写数据准备好

更新指针 x1

更新指针 x2

更新指针 x3

执行转移相关操作

μop Queue

fld(f0)	fld(f1)	Fadd	fsd		
---------	---------	------	-----	--	--

Load Data Queue

--	--	--	--	--	--

Store Data Queue

f2					
----	--	--	--	--	--

Store Address Queue

0(x1)					
-------	--	--	--	--	--

乱序执行

Cycle 9

fld f0
fld f1
fadd.d
fsd f2
addi
addi
addi
bne
fld f0

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f0

计算load地址，向Mem发出读请求， μ op 插入队列，等待数据返回写入f1

浮点计算 μ op 插入队列

计算store地址，向Mem发出写请求， μ op 插入队列，等待写数据准备好

更新指针 x1

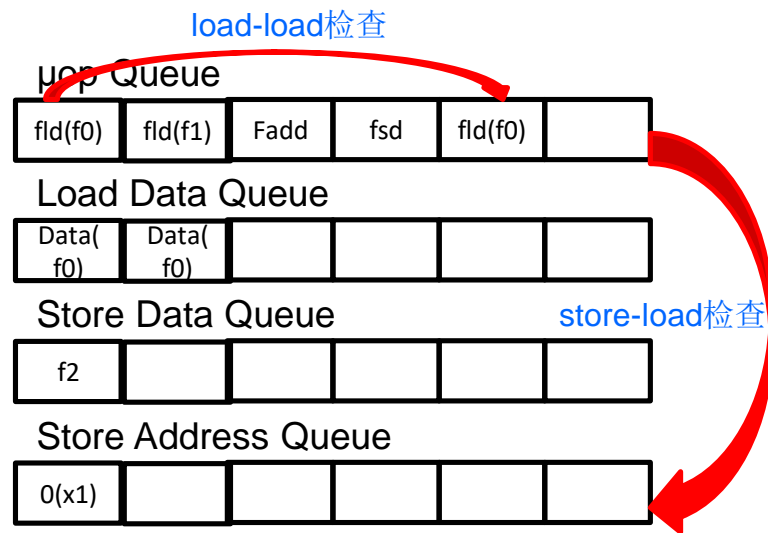
更新指针 x2

更新指针 x3

执行转移相关操作

**保证了每周期都可以
执行至少一条指令！**

```
loop:      fld f0, 0(x2) // x2 points to B
           fld f1, 0(x3) // x3 points to C
           fadd.d f2, f0, f1
           fsd f2, 0(x1) // x1 points to A
           addi x1,x1,8           // Bump pointer
           addi x2,x2,8           // Bump pointer
           addi x3,x3,8           // Bump pointer
           bne x1, x4, loop // x4 holds end
```



Thinking

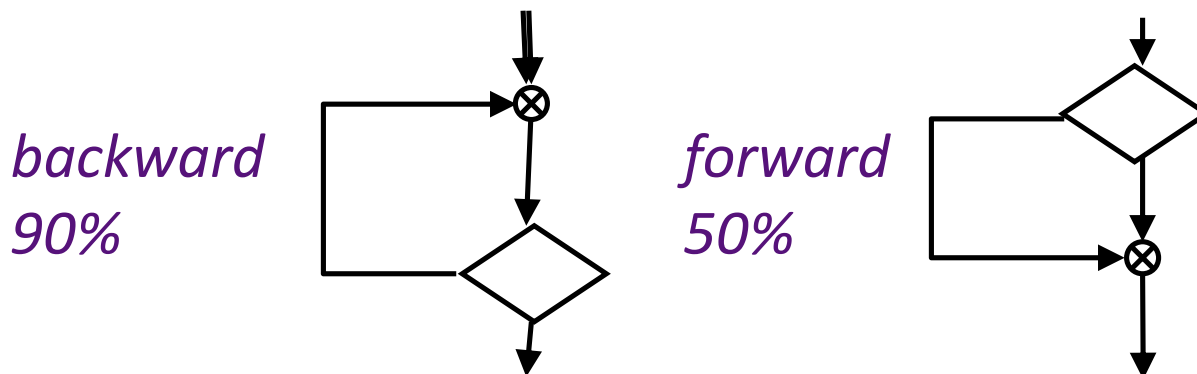
- 主流高性能处理器 =
 - + 复杂流水线（多功能部件）
 - + 超标量
 - + 乱序
- 当今主流高性能处理器基本上都采用超标量乱序的微结构
- 存在的问题：WAW、WAR、RAW

挖掘指令级并行：分支预测

- 分支预测：branch prediction
 - 分支方向预测
 - 分支目标地址预测
- 动机：对于超深流水线，分支指令的错误执行浪费了取指带宽，限制了流水线的性能
- 进一步，超标量流水线结构使得取指带宽的浪费更为严重
 - 主流高性能处理器参数：8级流水线，4发射
 - 从取指到获得分支目标地址，至少3个周期
 - 把分支预测的准确率提高到90% ~ 95%，就会减少近乎50%的错误预测，将错误预测之间的距离扩大一倍（指令数增多一倍）

静态分支预测

- 虽然分支方向不易预测，但也是有规律可循



- 可以使用 ISA 来传递分支方向信息，如Motorola MC88110
 - *bne0 (preferred taken), beq0 (not taken)*
- 某些 ISA提供静态分支方向预测的动态配置
 - 如 HP PA-RISC, Intel IA-64
 - 预测准确率可以达到80%

动态分支预测

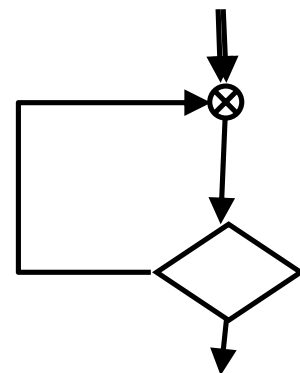
机器学习
😊

- 根据历史行为进行学习并预测未来
- 两类历史信息
 - 时间相关：同一条分支指令，行为相似
 - 空间相关：不同的分支指令之间是有关联的

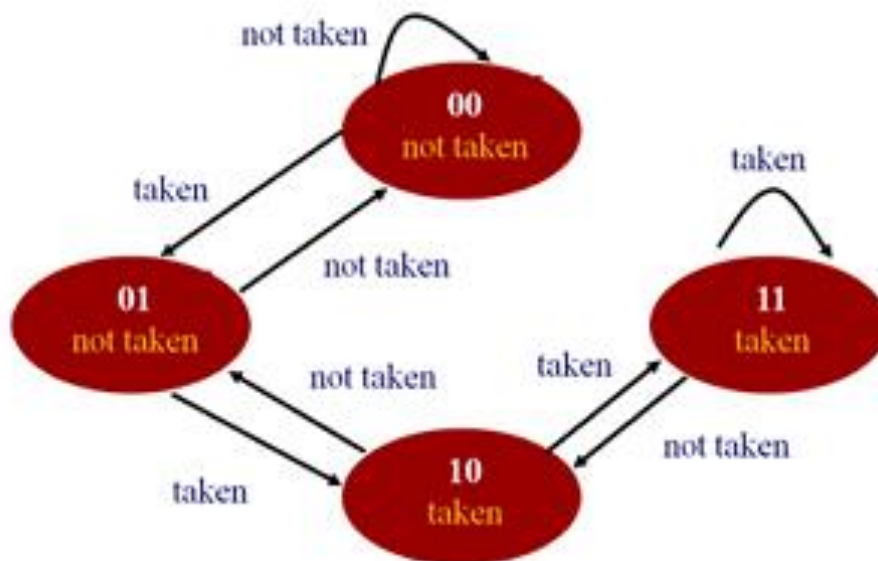
分支预测器

- 对每一条分支指令，使用1位计数器记录上一次的方向
 - 问题：如图循环体一定会错两次

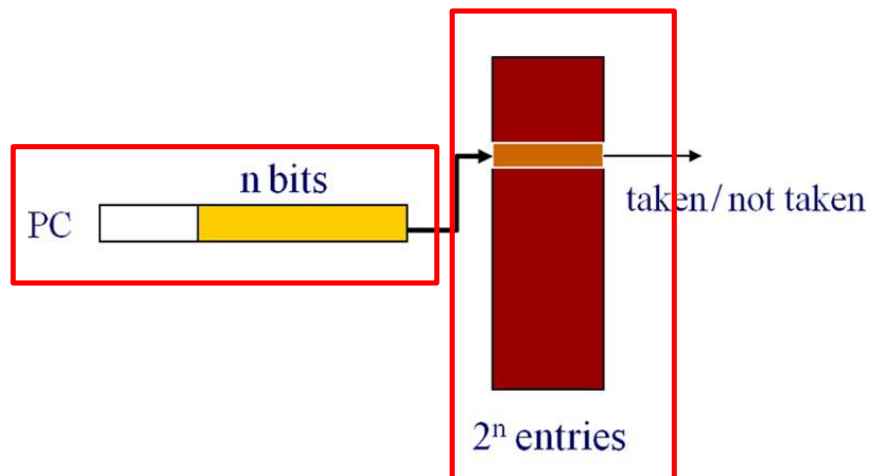
backward



- 考虑使用2位计数器记录历史
 - 连续的两次错误才会改变预测器的方向



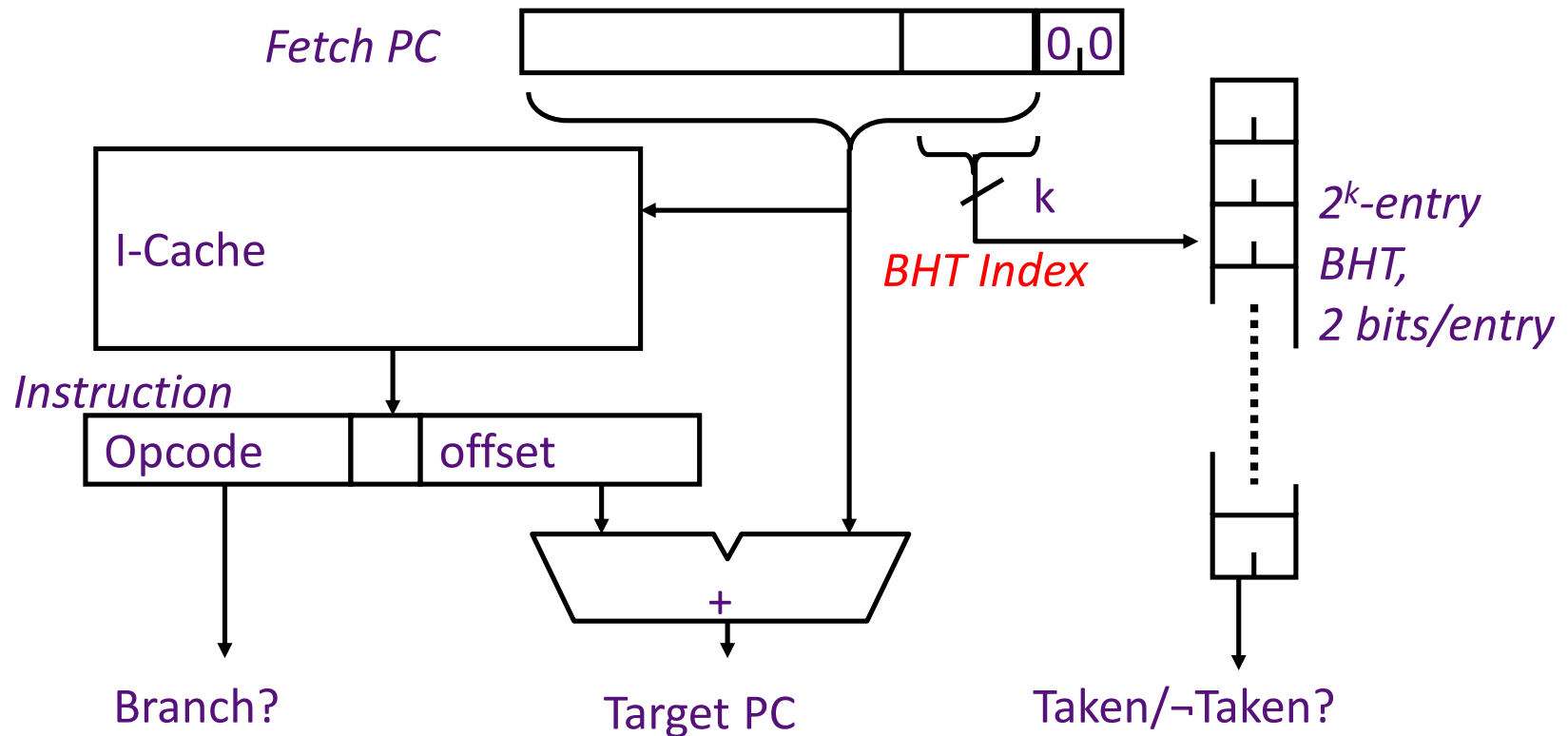
两位饱和计数器



- 2-bit本地转移预测器
- 拥有一个含 2^n 个表项的表
- 每个表项含2-bit饱和计数器，记录历史信息
- 通过PC的低n位进行索引
- 仅仅记录该分支（local）有限的历史信息，没有考虑分支执行的上下文信息

Branch History Table (BHT)

- 分支历史表



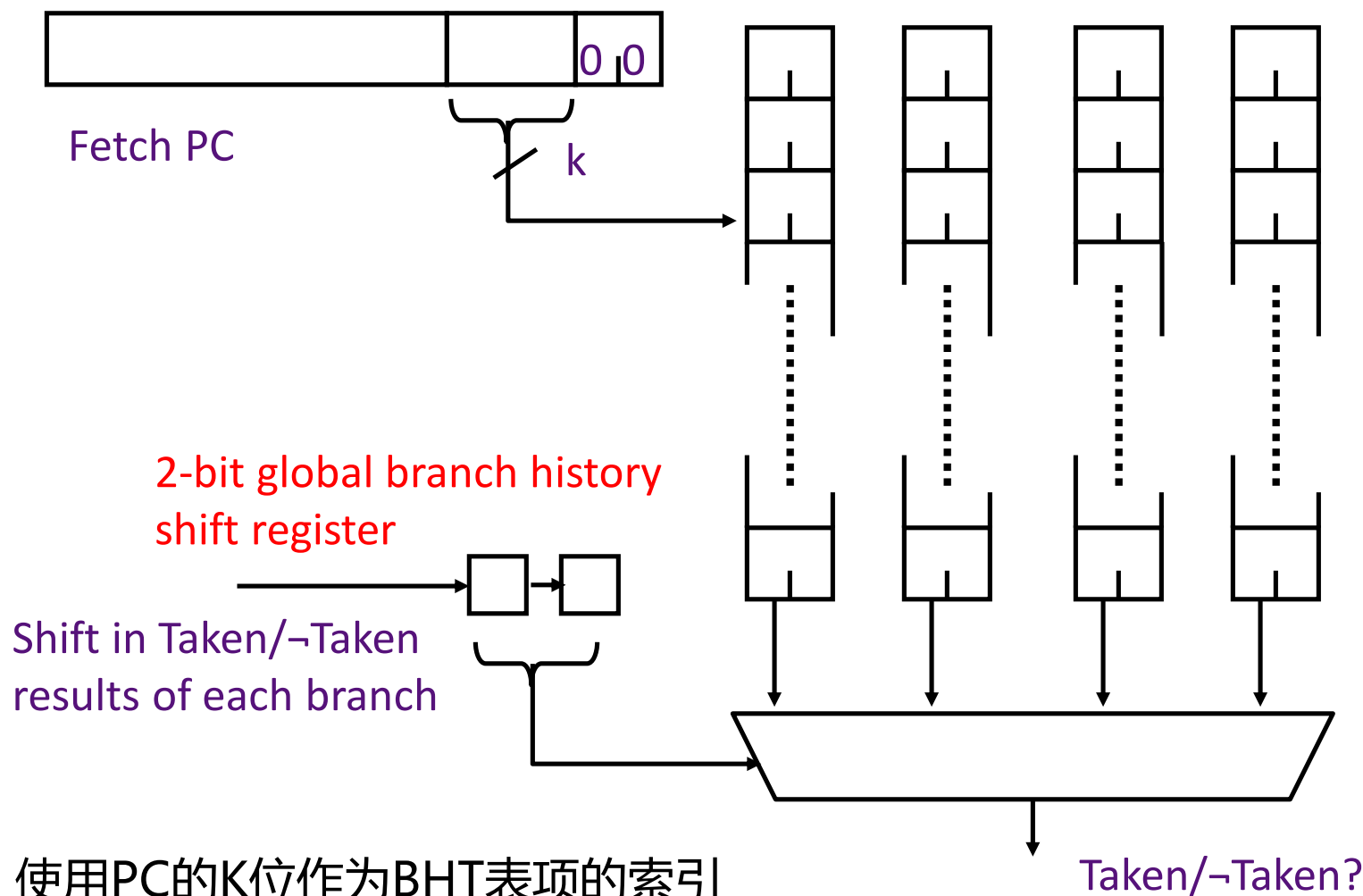
4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

|| 不足

```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

- 例如
 - c1 和 c2 是有关系的
 - 如果c1 不成立，则 c2 也不成立
- 如何将上下文信息考虑进来呢？
 - 使用分支历史寄存器，记录前N 条分支指令的方向历史

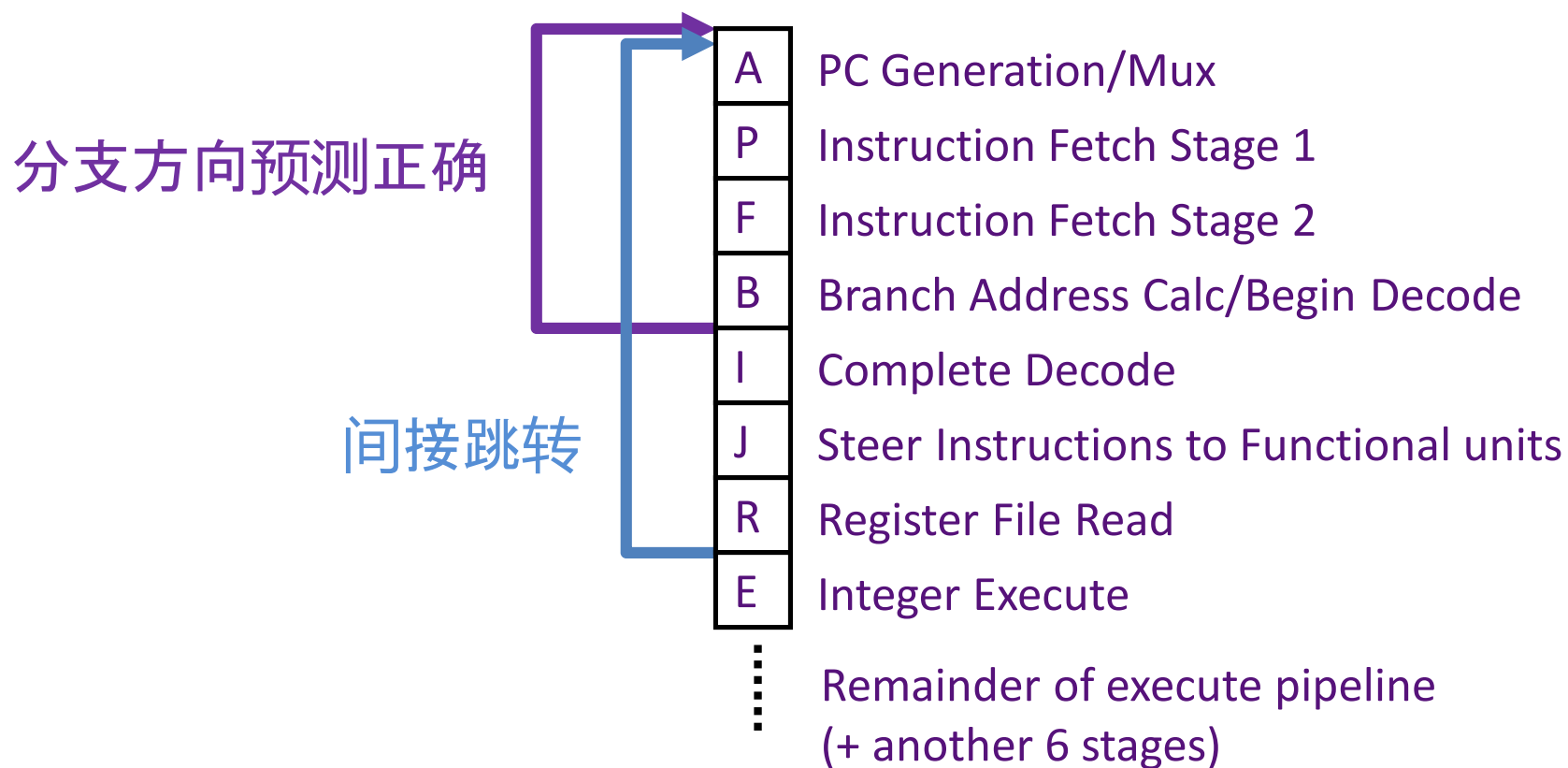
两级分支预测器



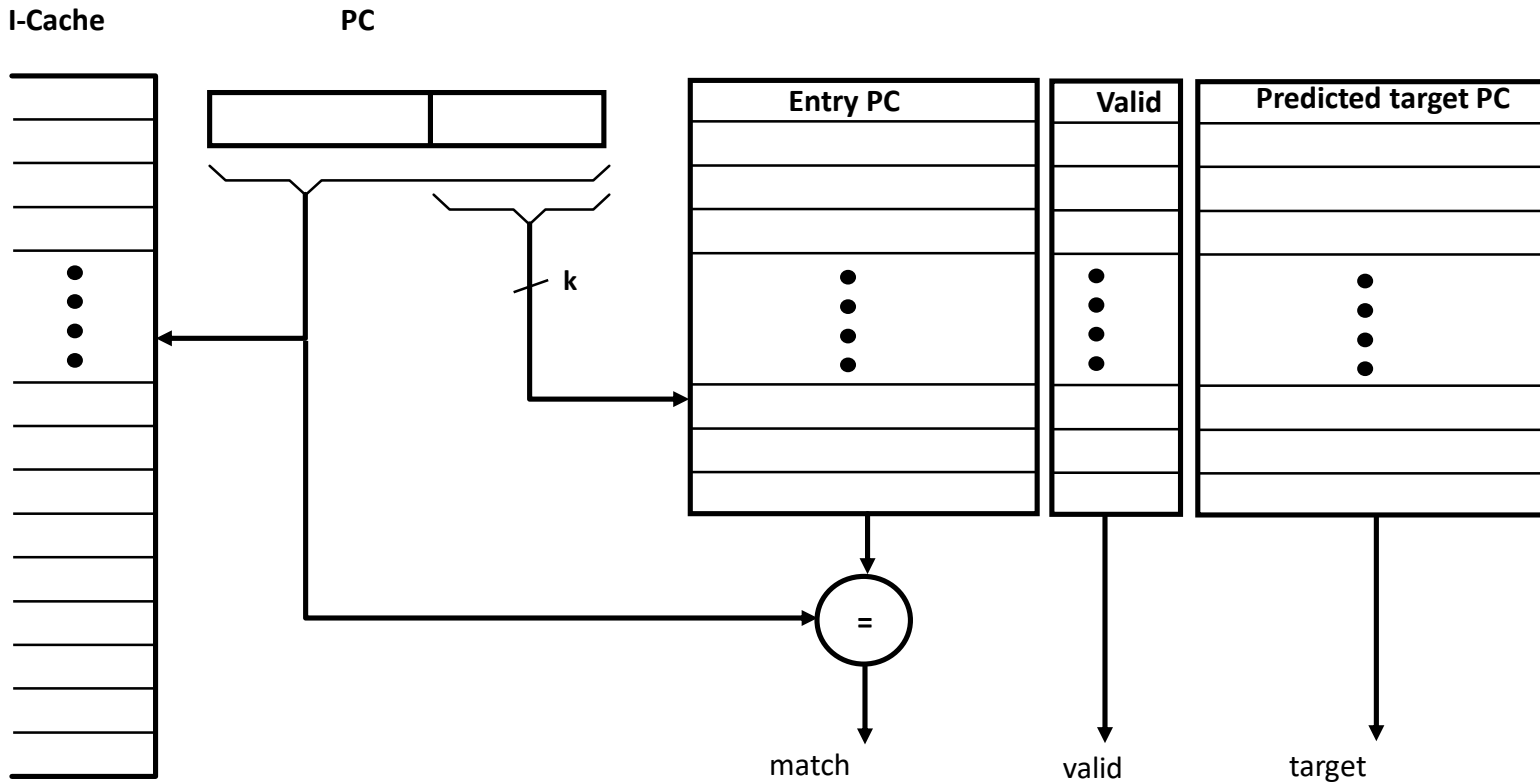
- 使用PC的K位作为BHT表项的索引
- 使用2位全局分支历史移位寄存器（BHSR）记录之前分支指令的历史
- 使用该2位BHSR作为不同BHT的索引

使用两级分支预测的局限性

- 只能预测方向，在未得知正确的分支目标地址之前仍然无法取指
- 以UltraSparc III的取指流水线为例



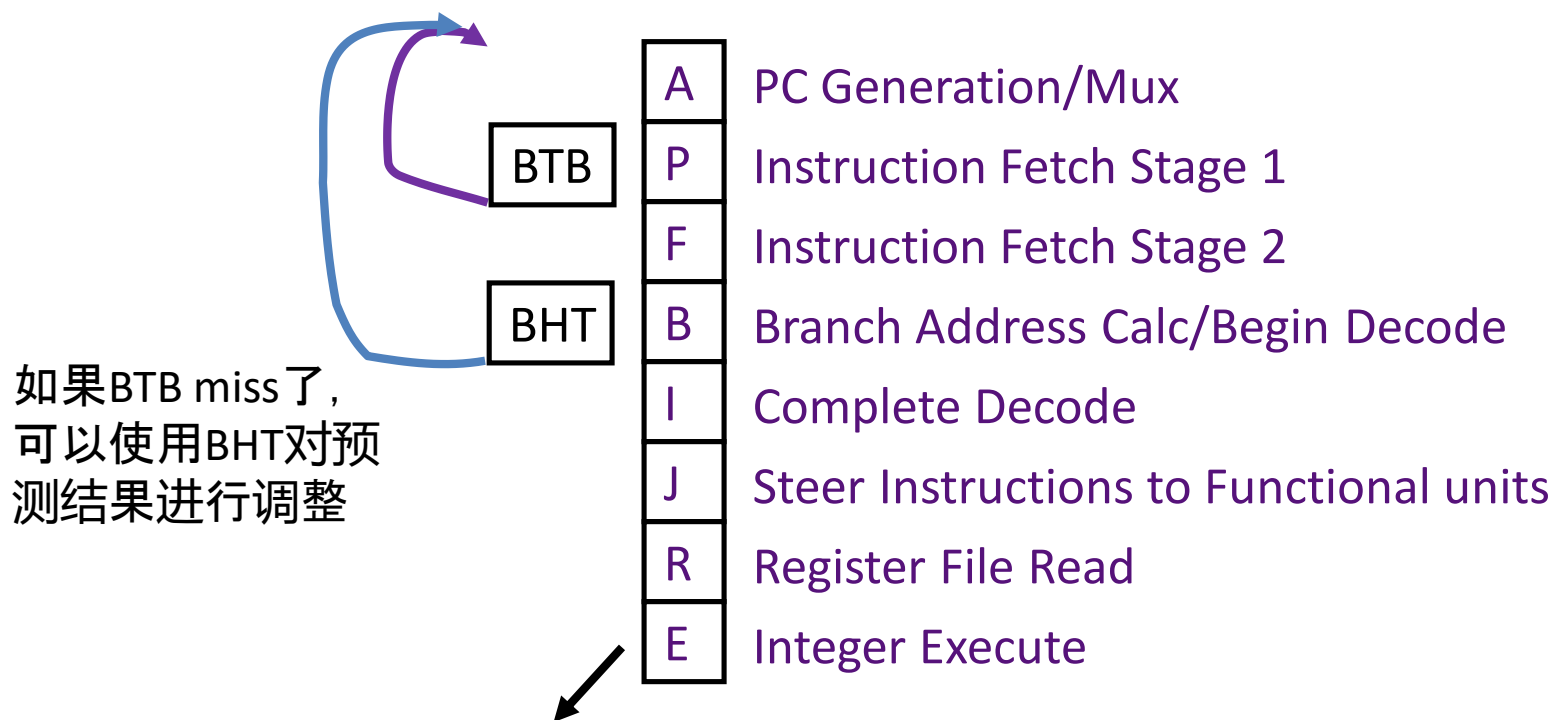
分支目标缓冲 (BTB)



- 需同时记录 PC 和预测的目标地址
- 只记录发生跳转 (Taken) 的分支指令和跳转指令的目标地址
- 下一条指令取指之前完成

BHT + BTB

- BTB表项记录的信息比BHT多，因此表项数不能太多，这使得BTB的miss率高于BHT
- 可以使用BHT + BTB的方式来进行调整



BTB/BHT 都需要在分支/跳转指令执行后进行更新

■ 跳转指令

- Jump, 跳转指令
 - 直接跳转, jump+PC相对偏移
 - 间接跳转, jump+寄存器
- BTB可以加速对**间接跳转指令**的目标地址的预测
- switch语句
 - 如果重复相同的情况, 非常适合BTB
- 动态函数调用
 - 运行时才能确定函数入口
 - 如果总是调用相同的函数, 非常适合BTB
 - 例如, C++中相同类型的虚函数调用
- 例程调用
 - 比如, 从许多不同的地方调用同一个函数

■ RAS (Return Address Stack)

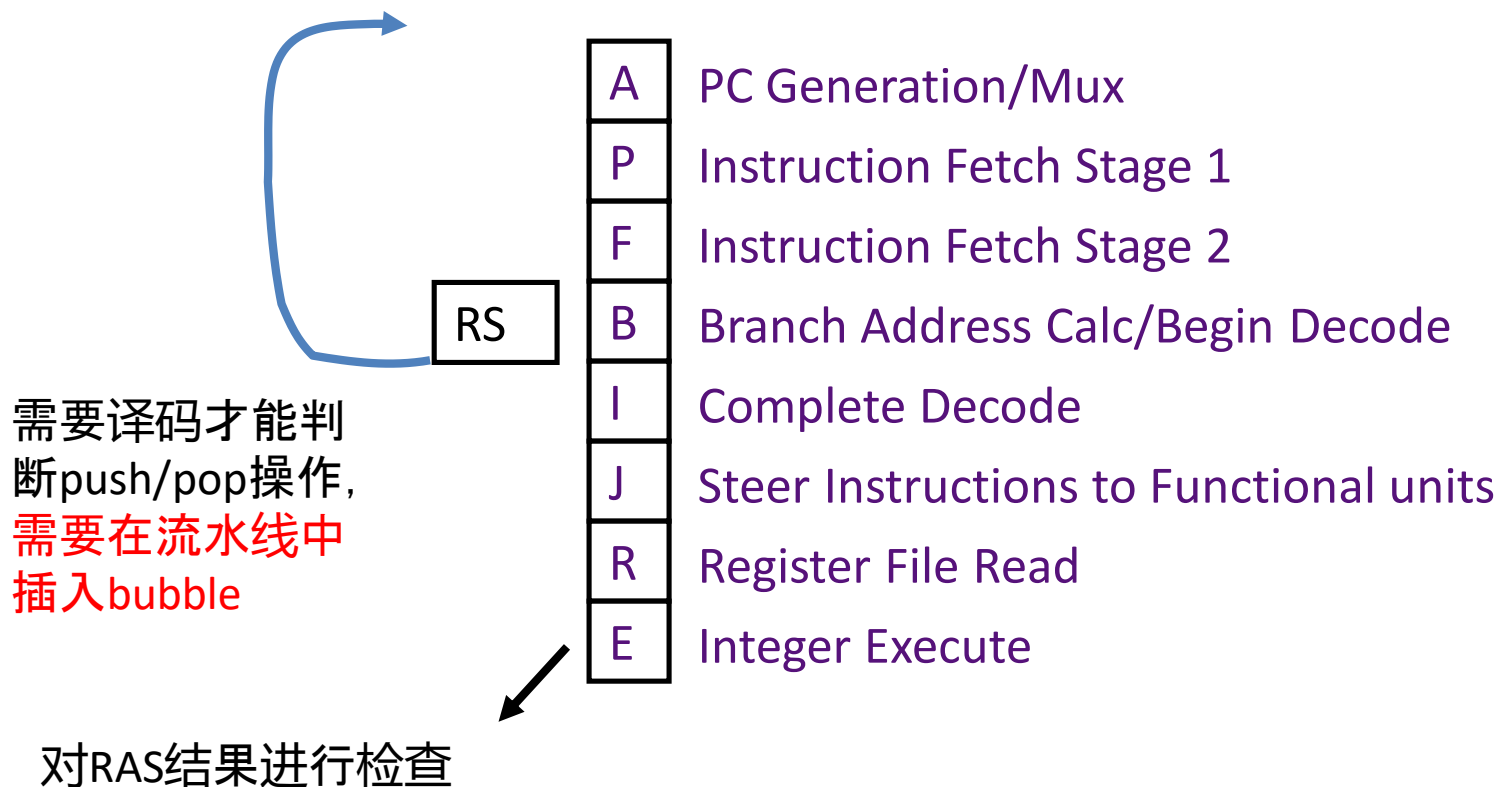
- 用来加速间接跳转目标地址预测的小结构
- 对于例程调用，比BTB更有效
- 当调用例程时，返回地址压栈
- 当从例程返回时，返回地址出栈



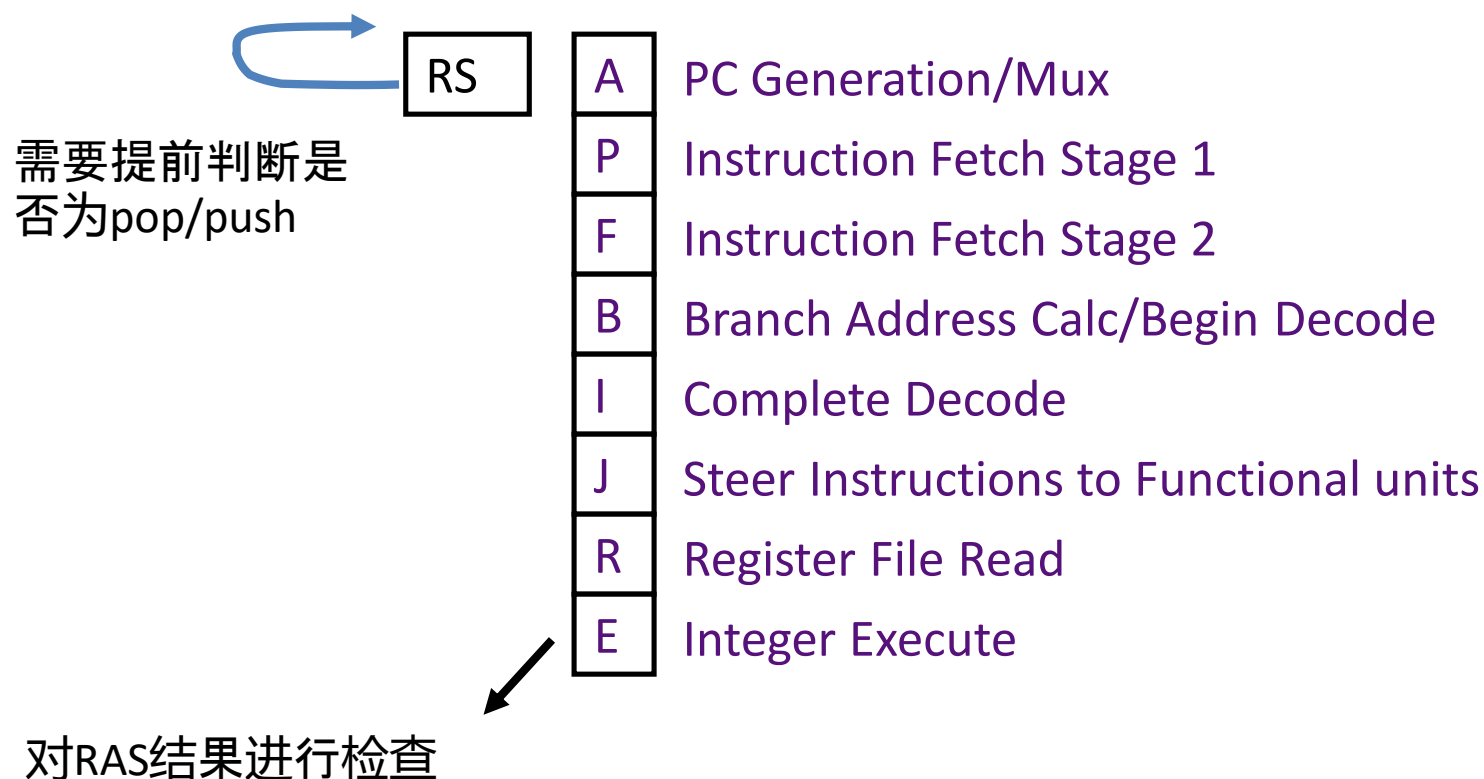
k entries
(typically k=8~16)

```
fa() { fb(); }  
fb() { fc(); }  
fc() { fd(); }
```

RAS在流水线中的位置



RAS在流水线中的位置



小结

- 流水线
 - 不断加深流水线，并不总能带来性能提升
 - 解决真数据相关是提高性能的关键，但非常难解决，要么减少延迟，要么容忍延迟（意味着需要挖掘更多的LIP）
- 挖掘指令级并行
 - 超标量/乱序，需要有复杂流水线（多功能部件）、多发射、动态调度等做支持
 - 分支预测的目的是提高取指带宽，间接跳转是最难预测的
 - 流水线就像一个大蓄水池，进水量和出水量要保持均衡，否则就会出现水流不畅或者水流枯竭的现象



欢迎提问

简单的分离

- 动机：当真相关的长延迟的 load 指令和长延迟的浮点计算指令在一个循环体内的時候，指令执行的并行度下降。
- 进一步分析：如果将 长延迟load指令 和 长延迟浮点计算指令 **进行分离**并解耦合，是否可以提高指令执行的并行度？
 - 分离：可以考虑两条流水线，一条处理定点操作，一条处理浮点操作

```
loop:  fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fadd.d f2, f0, f1
      fsd f2, 0(x1) // x1 points to A
      addi x1,x1,8 // Bump pointer
      addi x2,x2,8 // Bump pointer
      addi x3,x3,8 // Bump pointer
      bne x1, x4, loop // x4 holds end
```

红色：定点操作

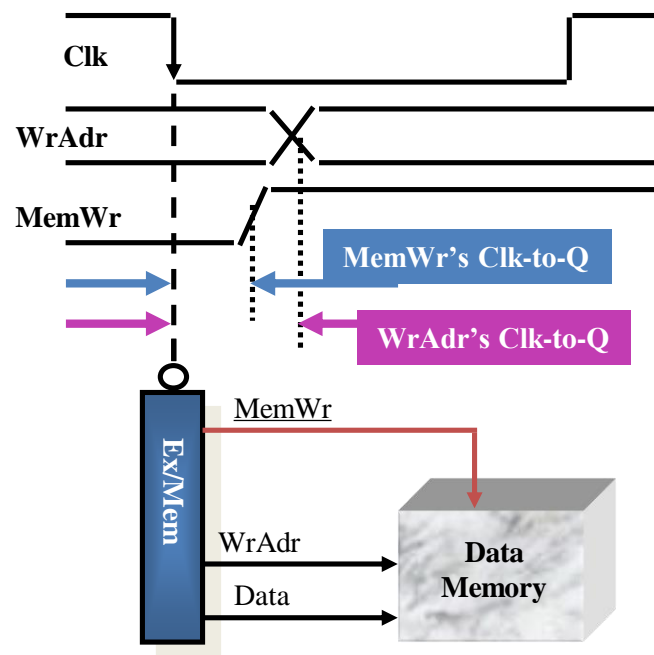
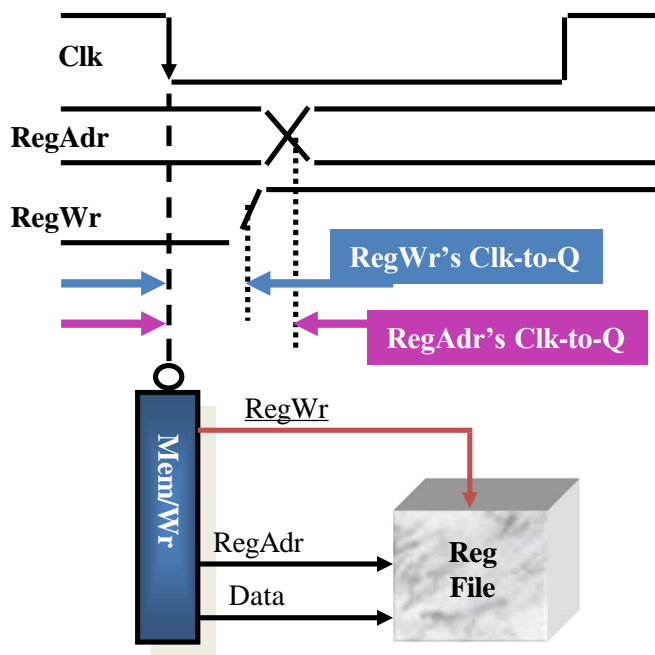
绿色：浮点操作

简单的解耦合



- 动机：当长延迟的 load 指令和长延迟的浮点计算指令在一个循环体内的时候，指令执行的并行度下降。
- 进一步分析：如果将长延迟的 load 指令 和 长延迟的浮点计算指令 进行分离并**解耦合**，是否可以提高指令执行的并行度？
 - 解耦合怎么做？
 - 解耦合的目的，降低真相关带来的影响

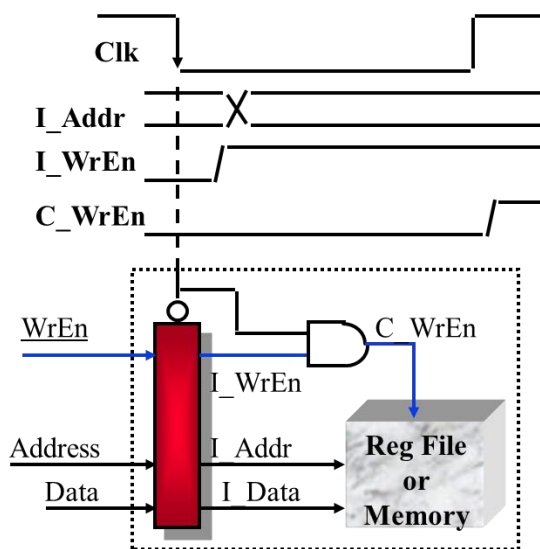
寄存器堆的写操作



- 在Wr段的开始, 如果下式成立, 现实中就会出现问题:
 - **RegAdr's (Rd or Rt) Clk-to-Q > RegWr's Clk-to-Q**
 - 在Address 和 Write Enable之间出现了竞争!
- 同样, 在Mem段的开始, 如果下式成立, 现实中就会出现问题:
 - **WrAdr's Clk-to-Q > MemWr's Clk-to-Q**
 - 在Address 和 Write Enable之间出现了竞争!

寄存器堆的写操作

- 多周期处理器是如何防止 Addr 和 WrEn 之间的竞争的？
 - 确保在第N个周期结束时地址是稳定的
 - 在第N+1个周期使 WrEn有效
- 那么，这种方法在流水线设计中能不能使用？为什么？
- 解决方案：门控时钟
 - 将 WrEn信号 和 时钟 进行逻辑与
 - 咨询电路专家，确保没有时序违例



同步存储器 和 同步寄存器

至少在时钟沿前的建立时间之间，**Address、Data和 WrEn**必须稳定

在捕获这些信号的时钟沿之后的时钟周期，发生写操作

