南京大学
NANJING UNIVERSITY

指令流水线

殷亚凤
智能软件与工程学院
苏州校区南雍楼东区225
yafeng@nju.edu.cn，https://yafengnju.github.io/

指令流水线

- 流水线概述
- 流水线处理器的实现
- **流水线冒险及其处理**
- 高级流水线技术

更加复杂的数据冒险问题

- 考察以下指令序列，采用前述**转发条件**会发生什么情况？

add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
……

本条指令

ForwardA (ForwardB) = { 01 当c2(a)=1 or c2(b)=1 时 ; 10 当c1(a)=1 or c1(b)=1 时

对于左边的指令序列，C1和C2的值各是什么？
C1=C2=1，使得Forward信号取值不确定！

可能会使转发到第3条指令的操作数是第1条指令结果，而不是第2条指令的结果！

怎样改写“转发”检测条件：改C1还是改C2?

应该让C1=1,C2=0!

需要改写“转发”条件C2(a)和C2(b)为：

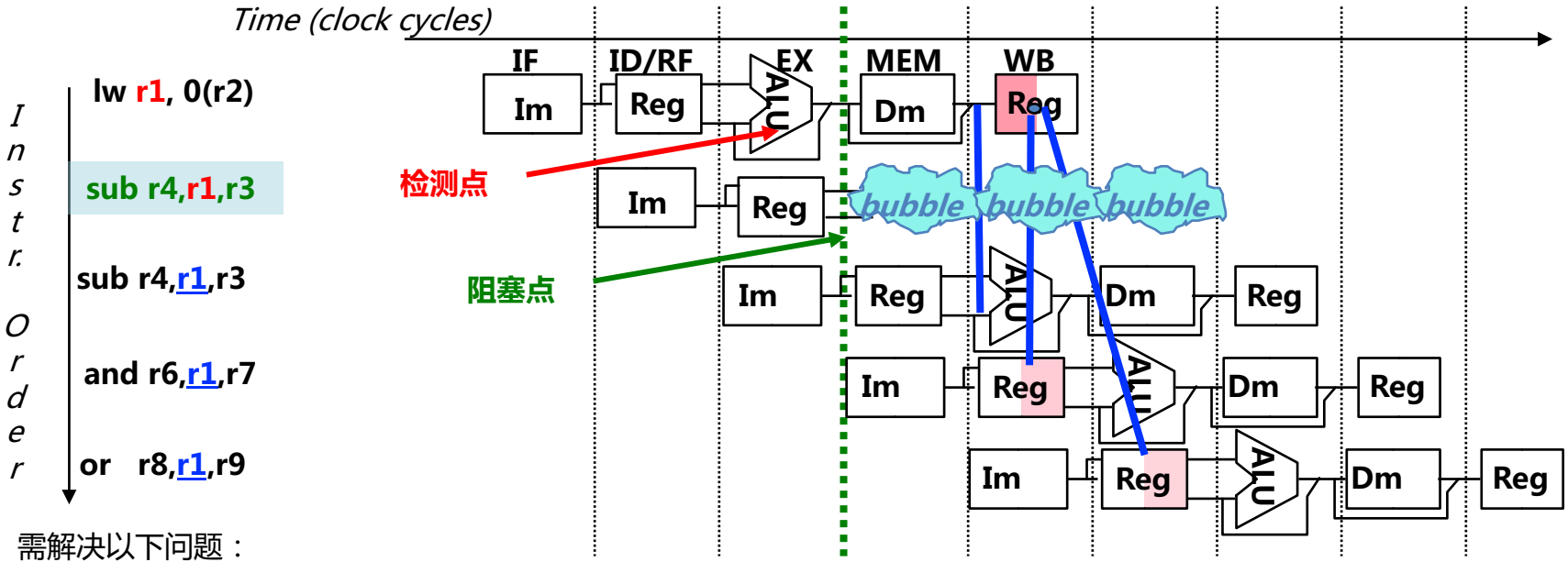
C2(a)=MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) **and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)** and (MEM/WB.RegisterRd=ID/EX.RegisterRs)

C2(b)=MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) **and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)** and (MEM/WB.RegisterRd=ID/EX.RegisterRt)

上述公式相当于加了一个条件限制：
如果本条指令源操作数和上条指令的目的寄存器一样，则不转发上上条指令的结果，
而转发上条指令的结果（即：此时的C1=1而C2=0）

至此，解决了RAW数据冒险的“转发”处理

硬件阻塞方式(Load-use Data Hazard)



需解决以下问题：

(1) 判断什么条件下需要阻塞

ID/EX.MemRead

and (ID/EX.RegisterRt=IF/ID.RegisterRs

or ID/EX.RegisterRt=IF/ID.RegisterRt)

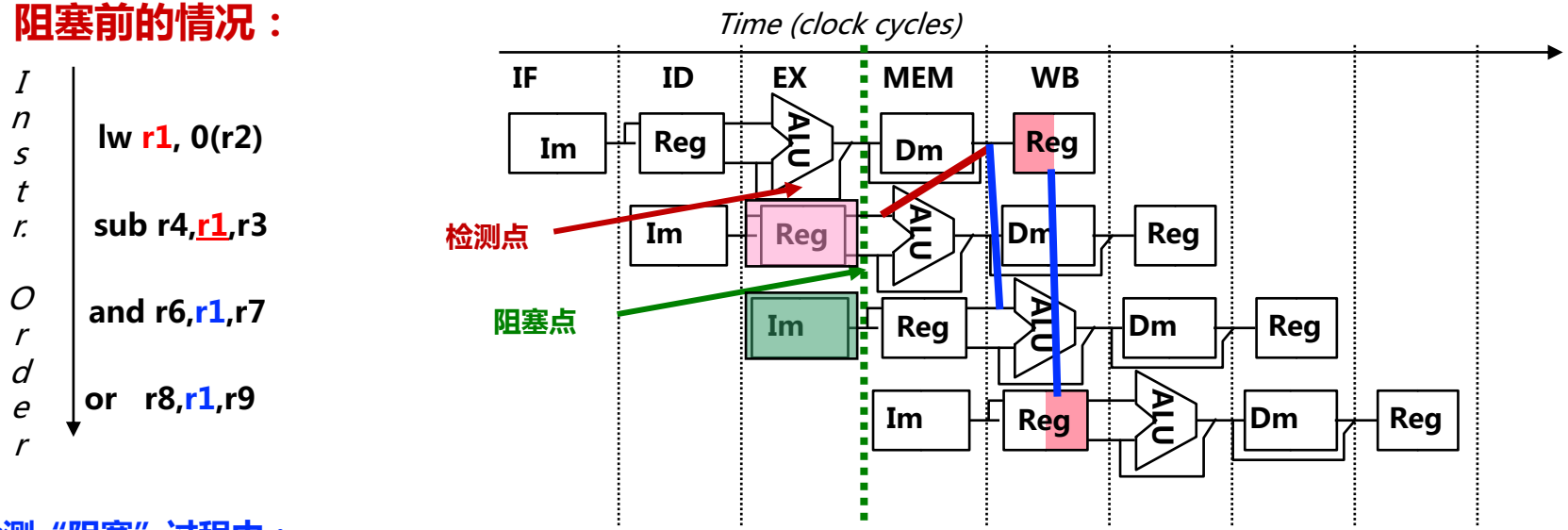
→ 前面指令为Load 并且

→ 前面指令的目的寄存器等于当前刚取出指令的源寄存器

(2) 如何修改数据通路来实现阻塞

硬件阻塞方式(Load-use Data Hazard)

阻塞前的情况：



检测“阻塞”过程中：

1) sub指令在IF/ID寄存器中，**并正被译码**，控制信号和Rs/Rt的值**将被写到**ID/EX段寄存器

2) and指令地址在PC中，**正被取出**，取出的指令**将被写到**IF/ID段寄存器中

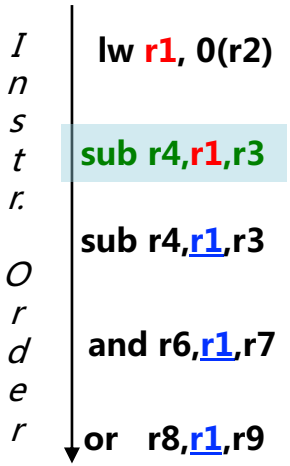
在阻塞点，必须将上述两条指令的执行结果清除，并延迟一个周期执行这两条指令

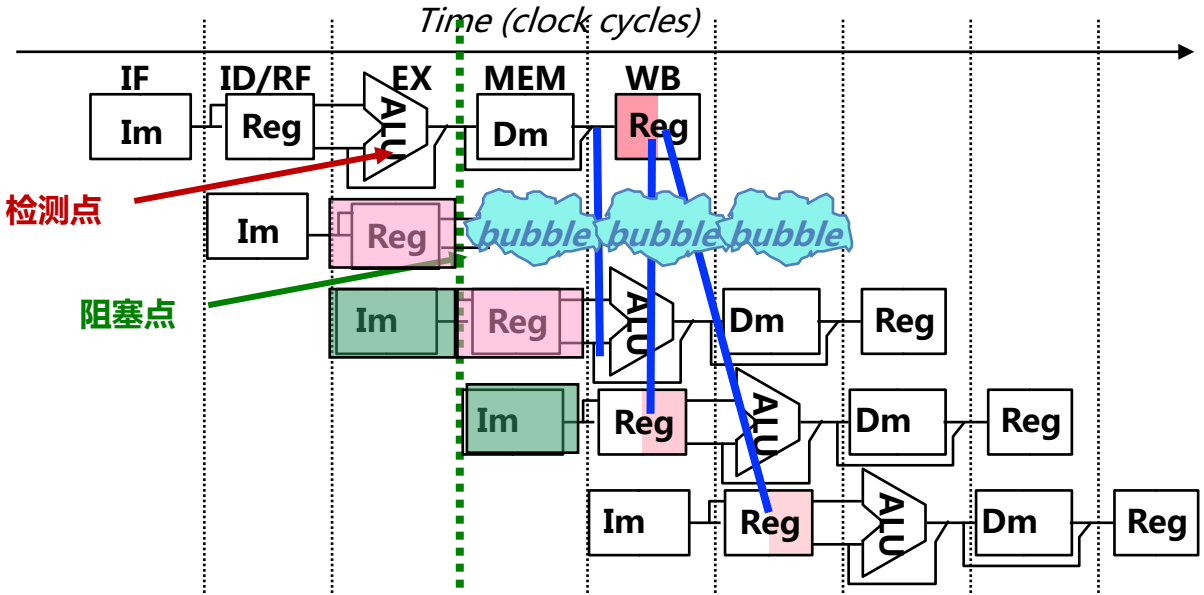
延迟一个周期执行后面指令，相当于把阻塞点前面一个周期的状态再保持一个周期

lw指令还是继续正常执行下去

硬件阻塞方式(Load-use Data Hazard)

阻塞后的情况：





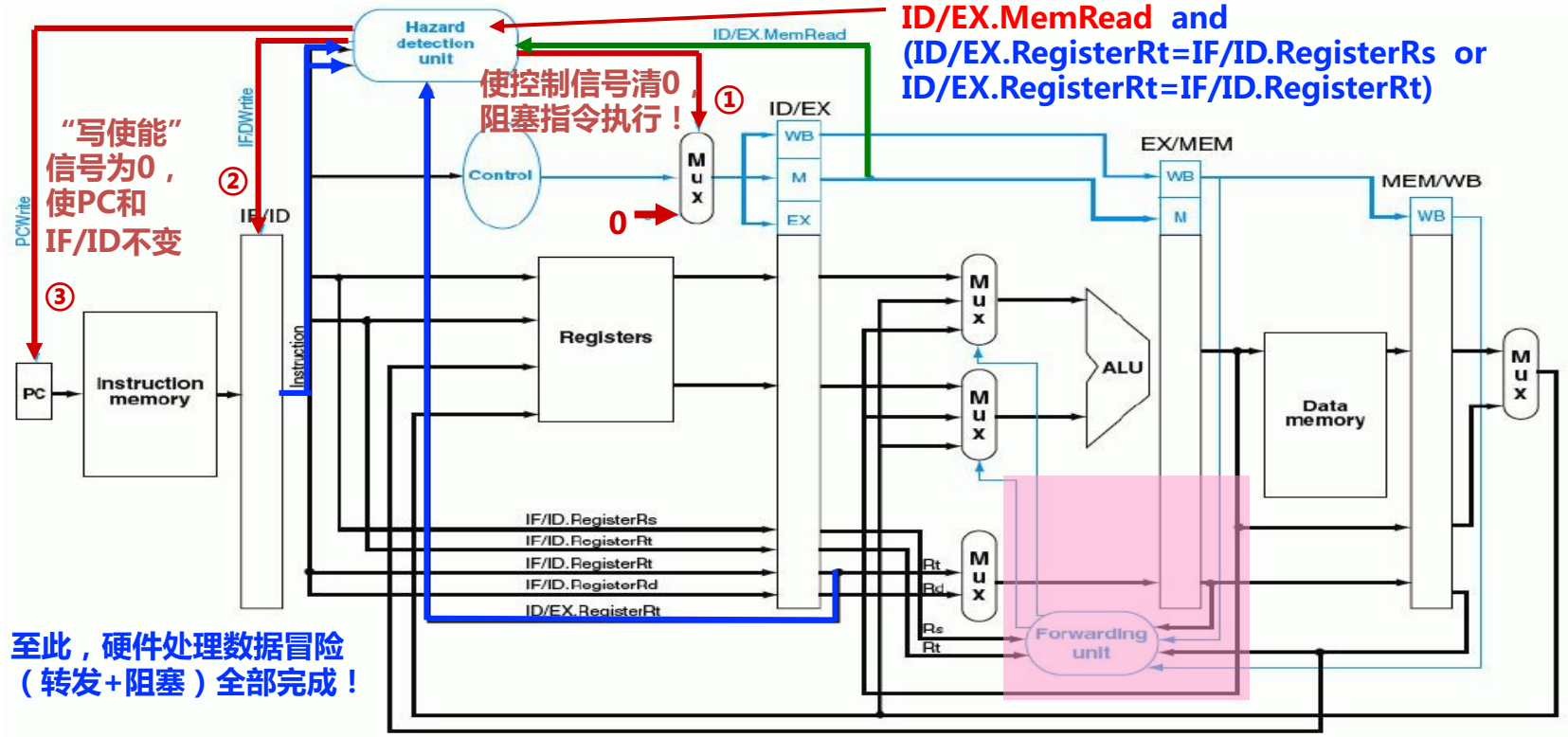
在阻塞点，将上述两条指令的执行结果清除，并延迟一个周期执行这两条指令

① 将ID/EX段寄存器中所有控制信号清0 ，插入一个“气泡”

② IF/ID寄存器中的信息不变（还是sub指令），sub指令重新译码执行

③ PC中的值不变（还是and指令地址），and指令重新被取出执行

带“转发”和“阻塞”检测的流水线数据通路



数据冒险-方案5：编译器进行指令顺序调整来解决数据冒险

以下源程序可生成两种不同的代码，优化的代码可避免Load阻塞

```
a = b + c;
d = e – f;
```

假定 a, b, c, d ,e, f 在内存

编译器的优化很重要！

Slow code:

```
lw    $2, b
lw    $3, c
add   $1, $2, $3
sw    $1, a
lw    $5, e
lw    $6, f
sub   $4, $5, $6
sw    $4, d
```

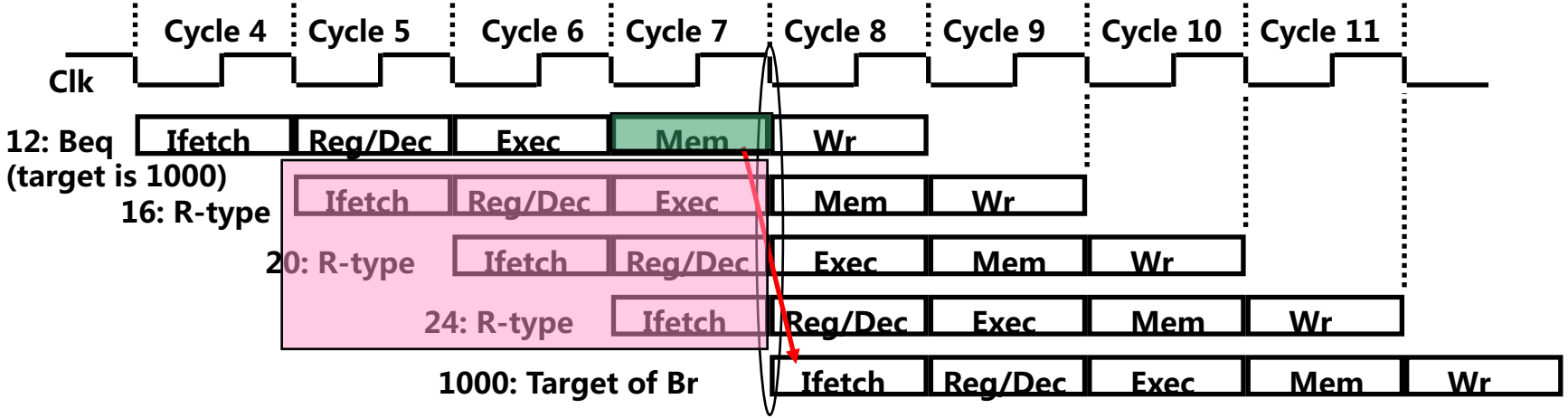
调整后

Fast code:

```
lw    $2, b
lw    $3, c
lw    $5, e
add   $1, $2, $3
lw    $6, f
sw    $1, a
sub   $4, $5, $6
sw    $4, d
```

如果硬件不支持阻塞处理的话，则编译器可以将顺序调整和插入NOP指令结合起来，在找不到可插入的指令时，插入NOP指令！

控制冒险现象



- 虽然Beq指令在第四周期取出，但:
  - “是否转移”在Mem阶段确定，目标地址在第七周期才被送到PC输入端
  - 第八周期才取出目标地址处的指令执行

  结果：在取目标指令之前，已有三条指令被取出，取错了三条指令！
- 发生转移时，要在流水线中清除Beq后面的三条指令，分别在EXE 、ID、 IF段中
- **延迟损失时间片C**：发生转移时，给流水线带来的延迟损失 **这里 C=3**

控制冒险的解决方法

- **方法1：硬件上阻塞（stall）分支指令后三条指令的执行**
  - 使后面三条指令清0或 其操作信号清0，以插入三条NOP指令
- **方法2：软件上插入三条“NOP”指令**

（以上两种方法的效率太低，需结合分支预测进行）

- **方法3：分支预测（Predict）**
  - **简单（静态）预测**：
    - 总是预测条件不满足(not taken)，即：继续执行分支指令的后续指令

      (可加启发式规则：在特定情况下总是预测满足(taken)，其他情况总是预测不满足。如：循环顶部（底部）分支总是预测为不满足（满足）。能达65%-85%的预测准确率)
  - **动态预测**：
    - 根据程序执行的历史情况进行动态预测调整，能达90%的预测准确率

    (注：流水线控制必须确保被错误预测指令的执行结果不能生效，而且要能从正确的分支地址处重新启动流水线工作)
- **方法4：延迟分支（Delayed branch）（通过编译程序优化指令顺序！）**
  - 把分支指令前面与分支指令无关的指令调到分支指令后执行，也称延迟转移

另一种控制冒险：异常或中断控制冒险的处理

简单（静态）分支预测方法

- **基本做法**
  - 总预测条件不满足(not taken)，即：继续执行分支指令的后续指令

    可加启发式规则：在特定情况下总是预测满足(taken)，其他情况总是预测不满足
  - 预测失败时，需把流水线中三条错误预测指令丢弃掉
    - 将被丢弃指令的控制信号值或指令设置为0

      （注：涉及到当时在IF、ID和EX三个阶段的指令）
- **性能**
  - 如果转移概率是50%，则预测正确率仅有50%
- **预测错误的代价**
  - 预测错误的代价与何时能确定是否转移有关。越早确定代价越少
  - 是否可把判断转移的工作提前，而不等到MEM阶段才确定？

可以！　那最早可以提前到哪个阶段呢？

简单（静态）分支预测方法

- **缩短分支延迟，减少错误预测代价**
  - 可以将 “转移地址计算”和“分支条件判断”操作调整到ID阶段来缩短延迟
    - **将转移地址生成从MEM阶段移到ID阶段，可以吗？为什么？**

    （是可能的：IF/ID流水段寄存器中已经有PC的值和立即数）
    - **将“判0”操作从EX阶段移到ID阶段，可以吗？为什么？**

    （用逻辑运算(如，先按位异或，再结果各位相或)来直接比较Rs和Rt的值）

    （简单判断用逻辑运算，复杂判断可以用专门指令生成条件码）

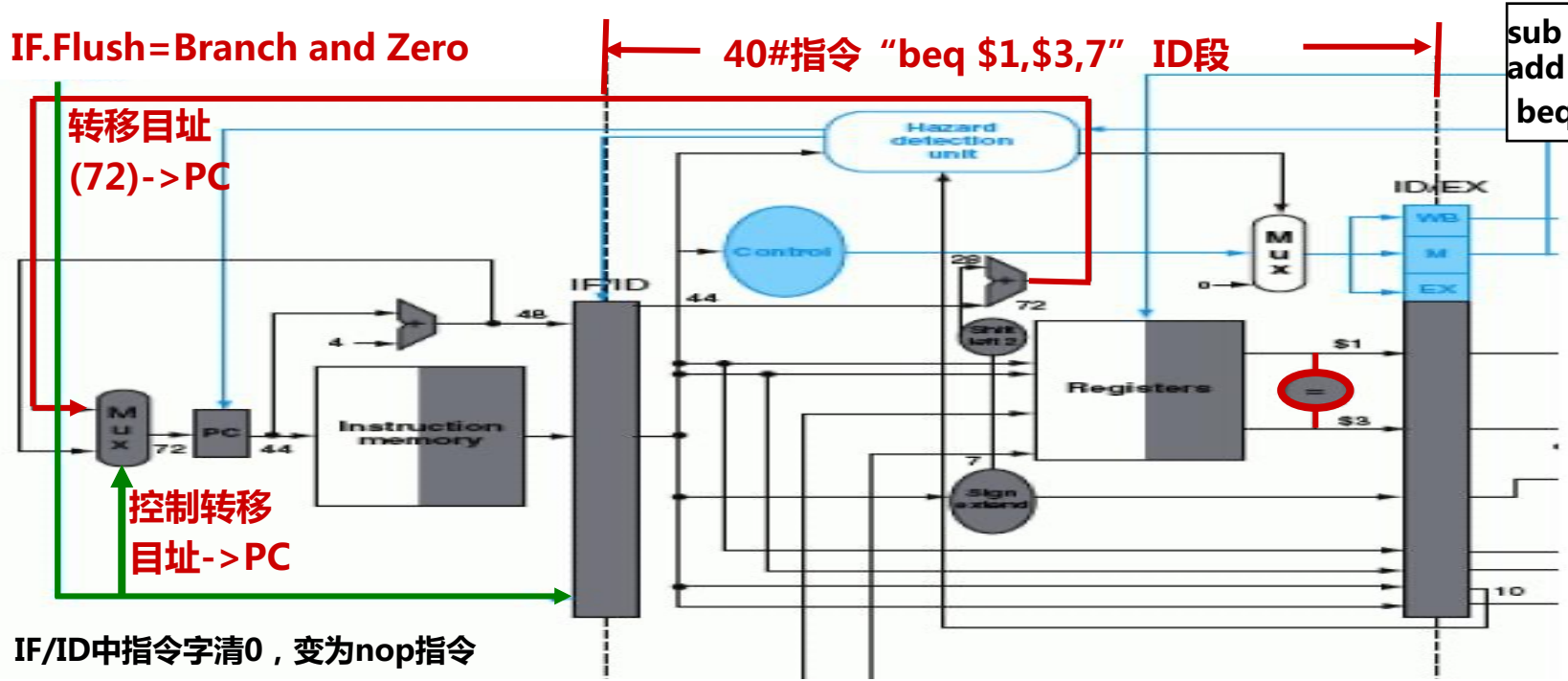
    （许多条件判断都很简单）
- **预测错误的检测和处理（称为“冲刷、冲洗” -- Flush）**
  - 当Branch=1并且Zero=1时，发生转移（taken）
  - **增加控制信号：IF.Flush=Branch and Zero**，取值为1时，说明预测失败
  - 预测失败 (条件满足) 时，完成以下两件事（延迟损失时间片C=1时）：

    ① 将转移目标地址->PC

    ② 清除IF段中取出的指令，即：将IF/ID中的指令字清0，转变为nop指令

原来要清除三条指令，调整后只需要清除一条指令，因而只延迟一个时钟周期，每次预测错误减少了两个周期的代价！ 即：这里 C=1

带静态分支预测处理的数据通路



IF/ID中指令字清0，变为nop指令

若$1或$3和前面指令数据相关，会怎么样？

- 上上条指令的EXE段结果可转发回来进行判断
- 上条指令的EXE段结果来不及转发回来，引起1次阻塞!

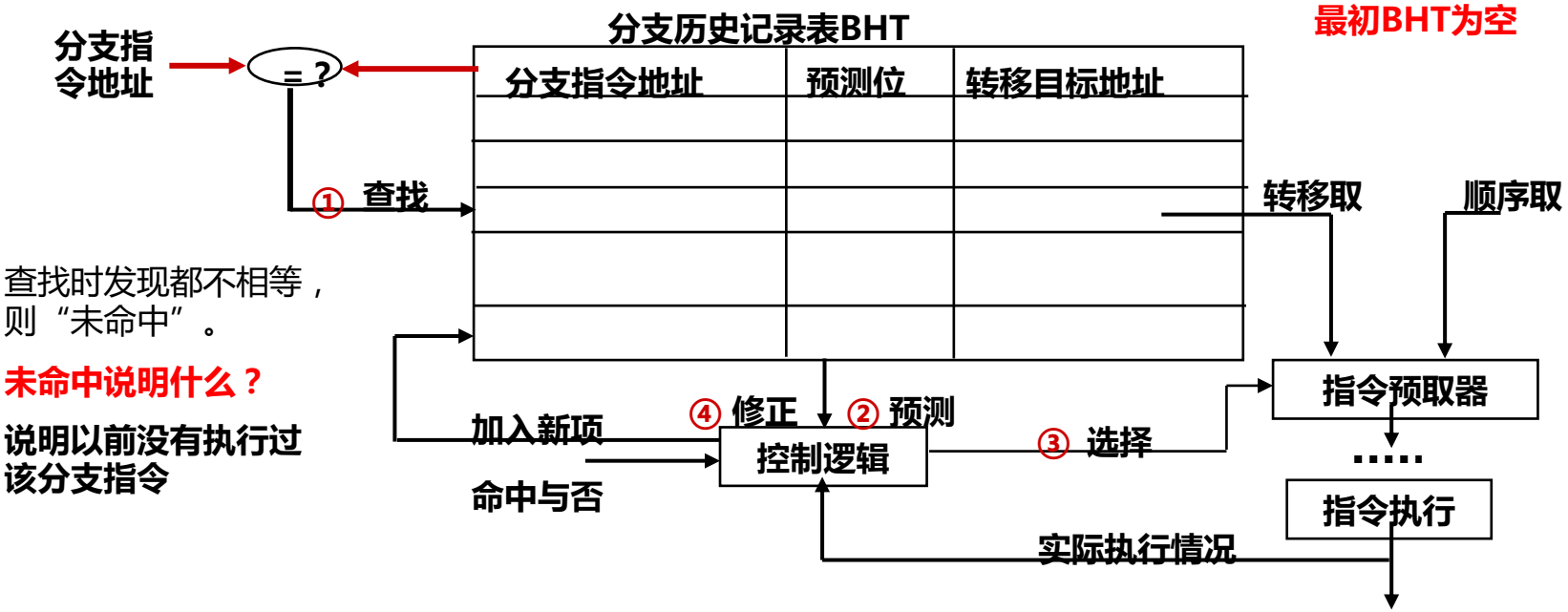
需重新改“转发”条件和转发线路！

动态分支预测方法

- 简单的静态分支预测方法的预测成功率不高，应考虑动态预测
- **动态预测基本思想**：
  - 利用**最近转移发生的情况**，来预测下一次可能转移还是不转移
  - 根据实际情况来调整预测
  - 转移发生的历史情况记录在BHT中（有多个不同的名称）
    - 分支历史记录表BHT（Branch History Table）
    - 分支预测缓冲BPB（Branch Prediction Buffer）
    - 分支目标缓冲BTB（Branch Target Buffer）
  - 每个表项由分支指令地址低位作索引，故在IF阶段就可以取到预测位
    - 低位地址相同的分支指令共享一个表项，所以，可能取的是其他分支指令的预测位。会不会有问题？
    - 由于仅用于预测，所以不影响执行结果

现在几乎所有的处理器都采用动态预测（dynamic predictor）

分支历史记录表BHT



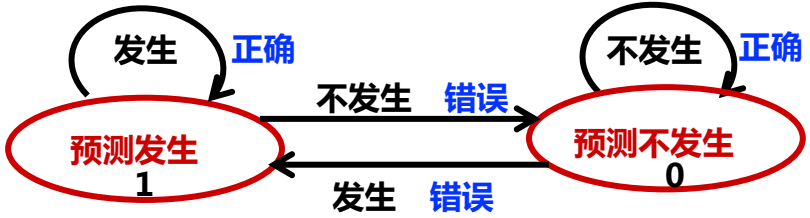
- **命中时**：根据预测位，选择“转移取”还是“顺序取”
- **未命中时**：加入新项，并填入指令地址和转移目标地址、初始化预测位

动态预测基本方法

- **采用一位预测位：总是按上次实际发生的情况来预测下次**
  - 1表示最近一次发生过转移（taken），0表示未发生（not taken）
  - 预测时，若为1，则预测下次taken，若为0，则预测下次not taken
  - 实际执行时，若预测错，则该位取反，否则，该位不变
  - 可用一个简单的预测状态图表示
  - 缺点：当连续两次的分支情况发生改变时，预测错误
    - 例如，循环迭代分支时，第一次和最后一次会发生预测错误，因为循环的第一次和最后一次都会改变分支情况，而在循环中间的各次总是会发生分支，按上次的实际情况预测时，都不会错。
- **采用二位预测位**
  - 用2位组合四种情况来表示预测和实际转移情况
  - 按照预测状态图进行预测和调整
  - 在连续两次分支发生不同时，只会有一次预测错误

采用较多的是二位或二位以上预测位。如：Pentium 4 的BTB2采用4位预测位

一位预测状态图



- 指令预取时，按照预测位读取相应分支的指令
  - 预测发生时，选择“转移取”
  - 预测不发生时，选择“顺序取”
- 指令执行时，按实际执行结果修改预测位
  - 对照状态转换图来进行修改
  - 例如：对于一个循环分支
  - 若初始状态为0(再次循环时为0)，则第一次和最后一次都错
  - 若初始状态为1，则只有最后一次会错。（再次循环时又改为0，还是有两次错）

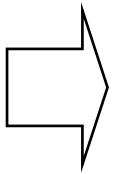
即：只要本次和上次的发生情况不同，就会出现一次预测错误。

```
Loop:    g = g +A[i];
         i = i+ j;
         if (i != h) go to Loop:
Assuming variables g, h, i, j  ~ $1, $2, $3,
$4 and base address of array is in $5
```

```
Loop:  add $7, $3, $3        ; i*2
       add $7, $7, $7        ; i*4
       add $7, $7, $5
       lw $6, 0($7)          ; $6=A[i]
       add $1, $1, $6        ; g= g+A[i]
       add $3, $3, $4
       bne $3, $2, Loop
          … …
```

举例：双重循环的一位动态预测

```
into sum (int N)
{
int i, j, sum=0;
for (i=0; i < N; i++)
    for (j=0; j <N; j++)
      sum=sum+1;
return sum;
}
```



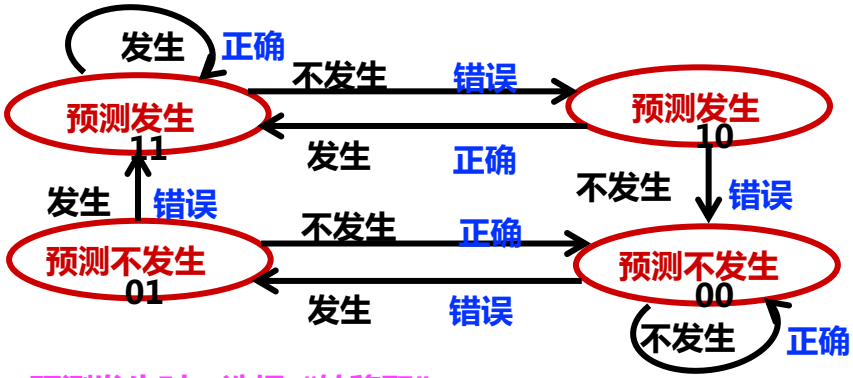
```
… …
Loop-i:  beq $t1,$a0, exit-i    # 若( i=N)则跳出外循环
         add $t2, $zero, $zero  #j=0
Loop-j:  beq $t2, $a0, exit-j   # 若(j=N)则跳出内循环
         addi $t2, $t2, 1                  # j=j+1
         addi $t0, $t0, 1                  #sum=sum+1
         j  Loop-j
exit-j:  addi $t1, $t1, 1       # i=i +1
         j  Loop-i
exit-i:  … …
```

- 外循环中的分支指令共执行N+1次，
- 内循环中的分支指令共执行N×(N+1)次。

N=10, 分别90.9%和82.7%
N=100, 分别99%和98%

若预测位初始为0，则外循环只有最后一次预测错误；跳出内循环时预测位变为1，再进入内循环时，第一次总是预测错误，并且任何一次循环的最后一次总是预测错误，因此，总共有1+2×(N–1)次预测错误（第一次循环有1次预测错，后面(N-1)次循环每次有2次预测错）。 **N越大预测准确率越高！**

两位预测状态图



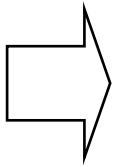
预测发生时，选择“转移取”
预测不发生时，选择“顺序取”

```
Loop:   add $7, $3, $3          ; i*2
        add $7, $7, $7          ; i*4
        add $7, $7, $5
        lw $6, 0($7); $6=A[i]
        add $1, $1, $6          ; g= g+A[i]
        add $3, $3, $4
        bne $3, $2, Loop
          … …
```

- **基本思想：只有两次预测错误才改变预测方向**
  - 11状态时预测发生（强转移），实际不发生时，转到状态10（弱转移），下次仍预测为发生，如果再次预测错误（实际不发生），才使下次预测调整为不发生00
- **好处：连续两次发生不同的分支情况时，会预测正确**
  - 例如，对于循环分支的预测（假定预测初始位为11）
  - 第一次：初始态为11（再次进入循环时为10），预测发生，实际也发生，正确
  - 中间：状态为“11”，预测发生，实际也发生，正确
  - 最后一次：状态为“11”，预测发生，但实际不发生，错

举例：双重循环的两位动态预测

```
into sum (int N)
{
int i, j, sum=0;
for (i=0; i < N; i++)
    for (j=0; j <N; j++)
       sum=sum+1;
return sum;
}
```



```
… …
Loop-i:  beq $t1,$a0, exit-i    # 若( i=N)则跳出外循环
         add $t2, $zero, $zero  #j=0
Loop-j:  beq $t2, $a0, exit-j   # 若(j=N)则跳出内循环
         addi $t2, $t2, 1                  # j=j+1
         addi $t0, $t0, 1                  #sum=sum+1
         j  Loop-j
exit-j:  addi $t1, $t1, 1       # i=i +1
         j  Loop-i
exit-i:  … …
```

- 外循环中的分支指令共执行N+1次，
- 内循环中的分支指令共执行N×(N+1)次。

N=10, 分别90.9%和90.9%

N=100, 分别99%和99%

若预测位初始为00，外循环只有最后一次预测错误；跳出内循环时预测 位变为01，再进入内循环时，第一次预测正确，只有最后一次预测错误，因此，总共有N次预测错误。 **N越大准确率越高！**

分支延迟时间片的调度

- 属于静态调度技术，由编译程序重排指令顺序来实现
- **基本思想：把分支指令前面的与分支指令无关的指令调到分支指令后面执行，以填充延迟时间片（也称分支延迟槽Branch Delay slot），不够时用nop填充**

举例：如何对以下程序段进行分支延迟调度？（假定时间片为2）

```
lw $1, 0($2)
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 2
add $3, $3,$2
sw $1, 0($2)
……
```

调度后

```
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 2
lw $1, 0($2)
nop
add $3, $3,$2
sw $1, 0($2)
```

若分支条件判断和目标地址计算提前到ID阶段，则分支延迟时间片减少为1

```
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 2
lw $1, 0($2)
add $3, $3,$2
sw $1, 0($2)
……
```

调度后可能带来其他问题：产生新的load-use数据冒险

调度后，无需在硬件线路中阻塞branch指令后面指令的执行

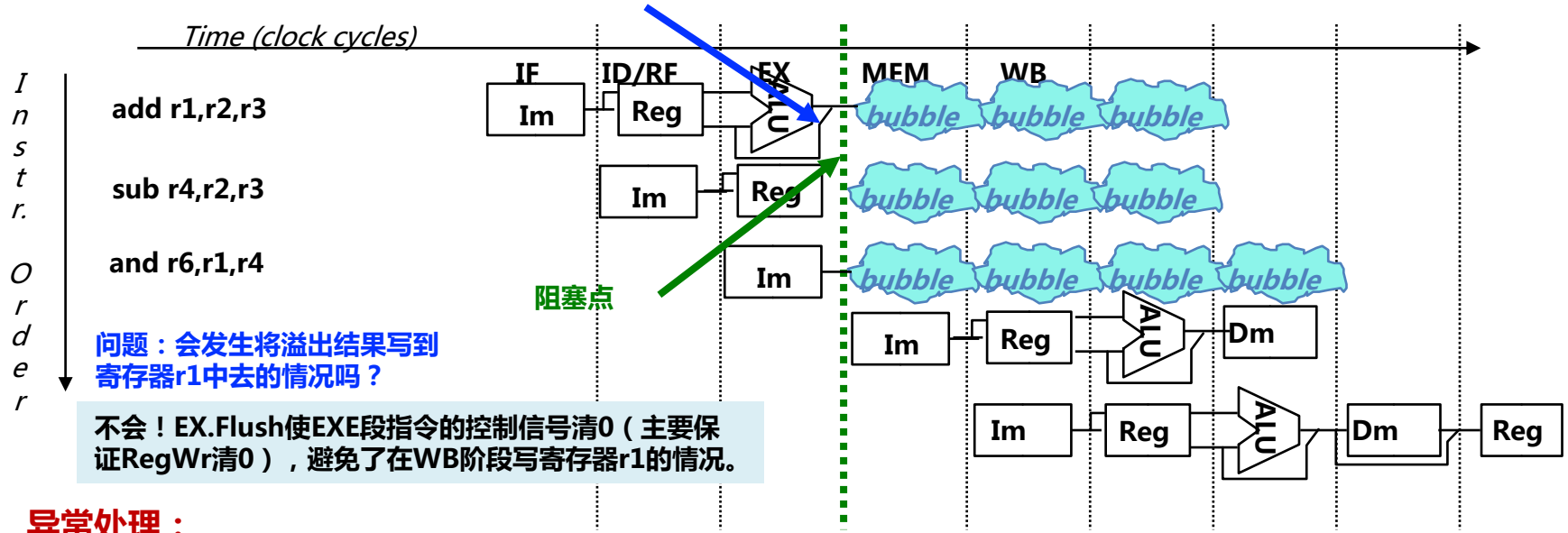
另一种控制冒险：异常和中断

- **异常和中断会改变程序的执行流程**
- **某条指令发现异常时，后面多条指令已被取到流水线中正在执行**
  - 例如ALU指令发现“溢出”时，已经到EX阶段结束了，此时，它后面已有两条指令进入流水线了
- **流水线数据通路如何处理异常? (举例说明)**
  - 假设指令add r1,r2,r3产生了溢出

    （记住：MIPS异常处理程序的首地址为0x8000 0180）
  - **处理思路：**
    - ✓ 清除add指令以及后面的所有已在流水线中的指令
    - ✓ 关中断（将中断允许触发器清0）
    - ✓ 保存PC或PC-4（断点） 到 EPC
    - ✓ 0x8000 0180送PC（从0x8000 0180处开始取指令）

异常的处理

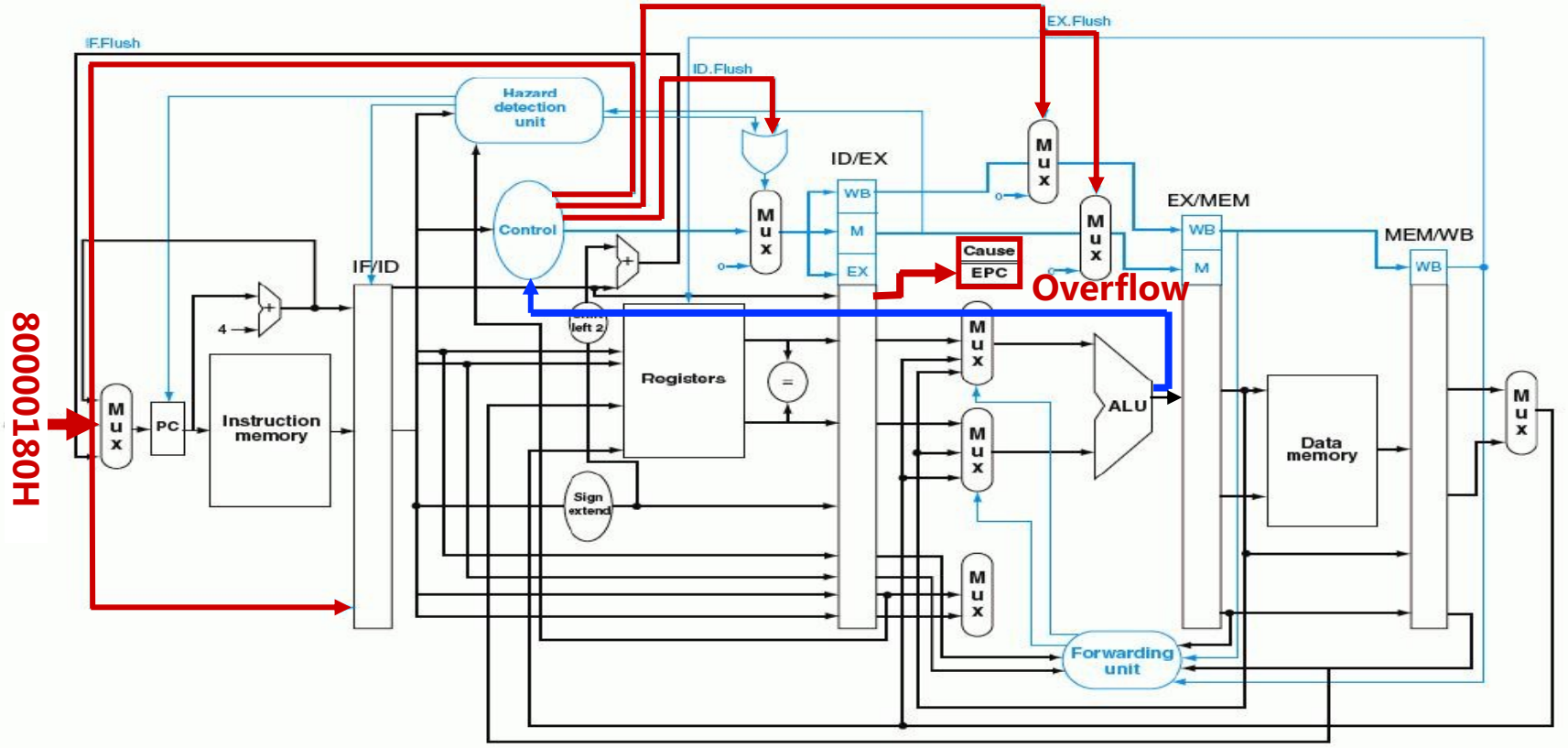
- 异常（溢出）在第一条指令的EXE阶段被检出



异常处理：

- IF.Flush使IF段指令在IF/ID寄存器中清为0，变成nop指令
- ID.Flush与数据冒险阻塞检测信号相或(or)后，使ID段指令的控制信号清0
- EX.Flush使EX段指令的控制信号清0
- 关中断，并将断点（可能是PC、可能是PC-4）保存到EPC中
- 将0x8000 0180作为PC的一个输入，并控制PC输入端的多路选择器

带异常处理的流水线数据通路



流水线方式下的异常处理的难点问题

- **流水线中同时有5条指令，到底是哪一条发生异常？**
  - 根据异常发生的流水段可确定是哪条指令，因为各类异常发生的流水段不同
    - ✓ “溢出”在EXE段检出
    - ✓ “无效指令”在ID段检出
    - ✓ “除数为0”在ID段检出
    - ✓ “无效指令地址”在IF段检出
    - ✓ “无效数据地址”在Load/Store指令的EXE段检出
- **外部中断与特定指令无关，如何确定处理点？**
  - 可在IF段或WB段中进行中断查询，需要保证当前WB段的指令能正确完成，并在有中断发生时，确保下个时钟开始执行中断服务程序
- **检测到异常时，指令已经取出多条，当前PC的值已不是断点，怎么办？**
  - 指令地址存放在流水段R，可把这个地址送到EPC保存，以实现精确中断
    非精确中断不能提供准确的断点，而由操作系统来确定哪条指令发生了异常
- **一个时钟周期内可能有多个异常，该先处理哪个？**
  - 异常：检出异常后，其原因存到专门寄存器中并流到最后阶段处理，使前面指令的异常优先级高于后面指令
  - 中断：在中断查询程序或中断优先级排队电路中按顺序查询
- **系统中只有一个EPC，多个中断发生时，一个EPC不够放多个断点，怎么办？**
  - 总是把优先级最高的送到EPC中
- **在异常处理过程中，又发生了新的异常或中断，怎么办？**
  - 利用中断屏蔽和中断嵌套机制来处理

指令流水线

- 流水线概述
- 流水线处理器的实现
- 流水线冒险及其处理
- **高级流水线技术**

提高性能措施—实现指令级并行

- **实现指令流内部的并行流水线称为指令级并行（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，它会增加开销，且是有极限的！可以怎样突破极限呢？

增加的开销体现在哪里？

流水段寄存器！

  - **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：理想的四路多发射流水线的IPC为4）
    - 需要实现以下两个主要任务 **一条流水线变成多条流水线！**
      - **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和冒险处理**
      - **动态多发射**：**由硬件在执行时动态完成指令打包和冒险处理**

N段流水线说明一个时钟周期内最多有几条指令同时并行执行？

N条！故N越大并行度越高！

实现多发射技术的基础—推测

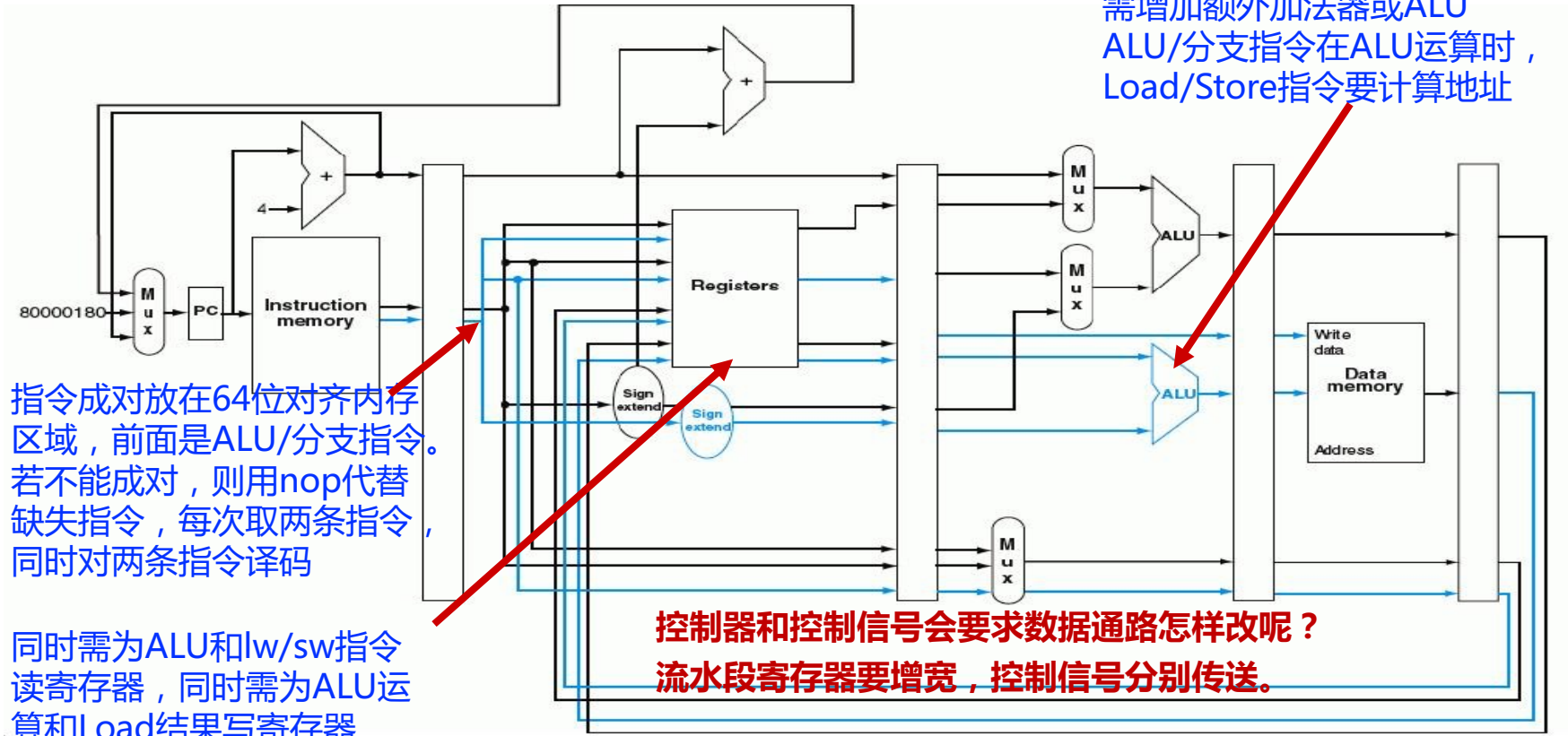
- **推测技术：由编译器或处理器猜测指令执行结果，并以此来调整指令执行顺序，使指令的执行能达到最大可能的并行**
  - **指令打包的决策依赖于“推测”的结果**
    - 可根据指令间的相关性来进行推测
      - **与前面指令不相关的指令可以提前执行**
    - 可对分支指令进行推测
      - **可提前执行分支目标处的指令**
    - 预测仅是“猜测”，可能推测错误，故需有推测错误检测和回滚机制
    - 推测错误时，会增加额外开销
  - **有“软件推测”和“硬件推测”两种**
    - 软件推测：编译器通过推测来静态重排指令（一定要正确！）
    -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

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”和“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）
    - 将同一个时钟周期内发射的多个指令看成是一条多个操作的长指令，称为一个**“发射包”**
    - **“静态多发射”**也被称为“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**（举例:干洗/水洗）
      (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！

2发射流水线数据通路


需增加额外加法器或ALU
ALU/分支指令在ALU运算时，
Load/Store指令要计算地址
指令成对放在64位对齐内存区域，前面是ALU/分支指令。若不能成对，则用nop代替缺失指令，每次取两条指令，同时对两条指令译码
同时需为ALU和lw/sw指令读寄存器，同时需为ALU运算和Load结果写寄存器
控制器和控制信号会要求数据通路怎样改呢？
流水段寄存器要增宽，控制信号分别传送。
80000180
M u x
PC
Instruction memory
4
+
Registers
Sign extend
Sign extend
ALU
ALU
Write data
Data memory
Address

2发射流水线的特点

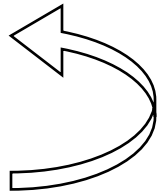
- **优点：**潜在性能将提高大约2倍（**实际上达不到！为什么？**）
- **缺点：**
  - 为消除结构冒险，需增加**额外部件**
  - 增加了潜在的由于数据冒险和控制冒险导致的**性能损失**
    - **例1：对于Load-use数据冒险**
      - 单发射流水线：只有一条指令延迟
      - 2发射流水线：有一个周期（即2条指令）延迟
    - **例2：对于ALU-Load/Store数据冒险**
      - 单发射流水线：可用“转发”技术使ALU结果直接转发到Load/Store指令的EXE阶段
      - 2发射流水线：两条指令同时进行，ALU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
```



(能看出这段程序的功能吗？) 循环内进行的是数组访问！

- 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？
- $s1减4，故sw指令偏移改为4
- 能否把addi和lw配成一对？
- 寄存器$s1被同时读，并读后写，两条指令的操作不会相互影响。

- 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:	addi $s1, $s1, -4 ? ?	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次后循环内指令（lw, addu, sw与数组访问相关）各有4条，再加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

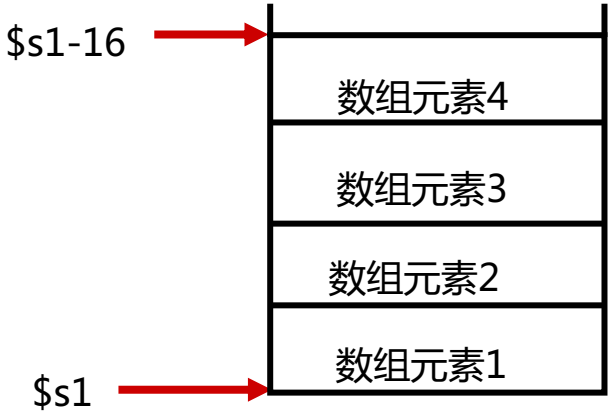
- 14条指令用了8个时钟，CPI达到8/14=0.57。
- 需要用到“**重命名寄存器**”技术，多用了三个临时寄存器$t1,$t2,$t3，消除了名字依赖关系（非真实依赖，只是寄存器名相同而已）

- **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）
- **好处**：充分利用并行，并消除部分循环分支！

循环展开后的偏移量

- 第一条指令将$s1减16，指令执行后，$s1的值变成了循环结束时$s1的值
- **所以循环体内各数组元素的访问指令的偏移量依次为**：
  16 - 数组元素1，12 - 数组元素2 ，8 - 数组元素3，4 - 数组元素4

新的$S1



- **为什么第一个周期中的lw指令的偏移量为0?**
  - 因为第一个周期中的lw指令进行地址计算时，addi指令的执行结果还没有写到$s1中，所以，此时$s1中还是原来的值?
- **为什么第一条addu指令不放在第二周期?**
  - 为了避免load-use数据冒险！
- 当循环次数不是4的倍数时这样做就有问题！

编译器和机器结构密切相关！系统程序员必须非常了解机器结构！编译器的好坏直接影响程序执行快慢！

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）
  - 同一个时钟动态发射多条指令，一个周期内可执行一条以上指令
- **与VLIW处理器的不同点：**
  - **VLIW处理器**：与机器结构密切相关，在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（还是串行序列），不进行指令打包，而是由硬件根据机器结构决定同时发射哪几条指令。因此，编译后的代码能够被不同结构的机器正确执行
- **超标量处理器多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指令可提前到addu指令前执行

如果不将sub调到前面，则会影响slti指令的执行，而且还会发生load-use冒险

动态多发射处理器的通用模型

指令预取和译码单元：预取的指令经译码后，放到指令队列中；

指令分派器：确定何时发射哪条指令到哪个功能单元中；

功能单元：如整数部件、浮点部件、存/取部件；

重排序缓冲：保存已完成的指令结果，等待在可能时写回寄存器堆。

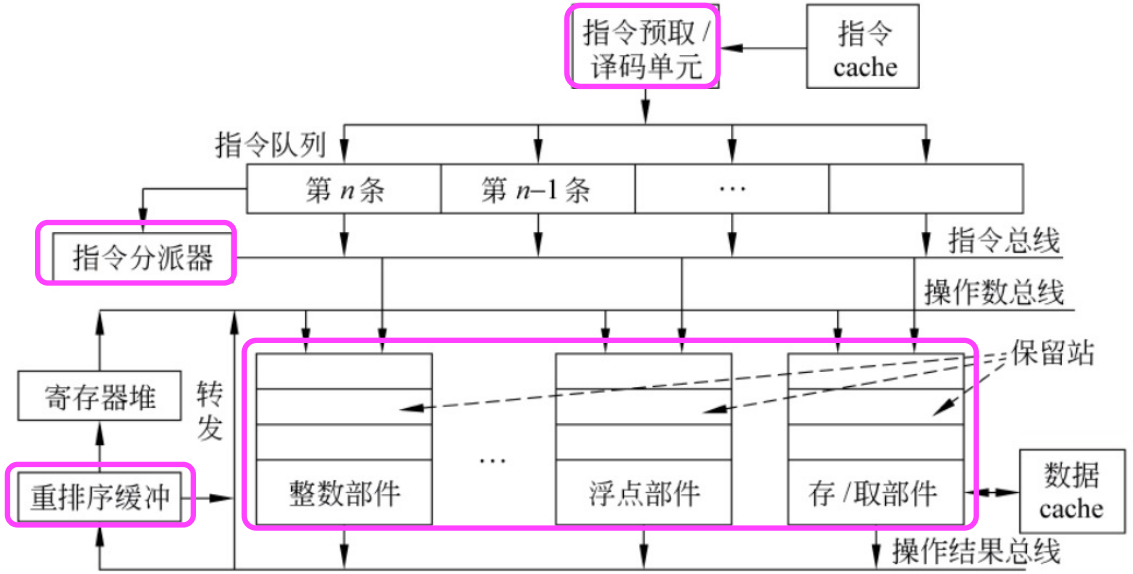


图 6.28　动态多发射流水线处理器的通用模型

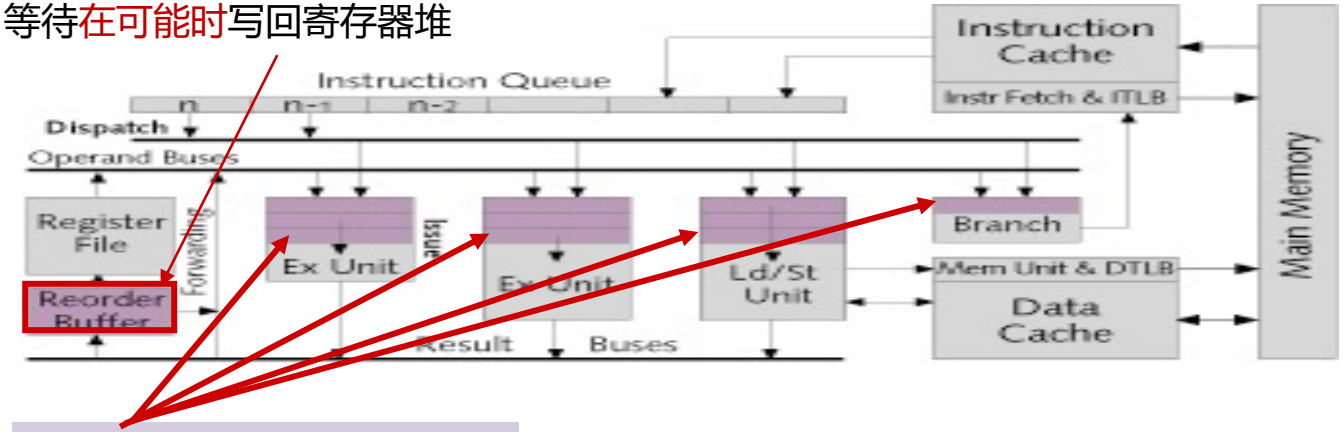
动态流水线的几种执行模式

- 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）

保守的是顺序完成，好处：
(1) 简化异常检测及其处理
(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




```
addu  $t0, $t0, $s2
addi   $s1, $t0, -4
beq    $s1, $t3, exit
mul    $t1, $t1
mflo   $s3
```

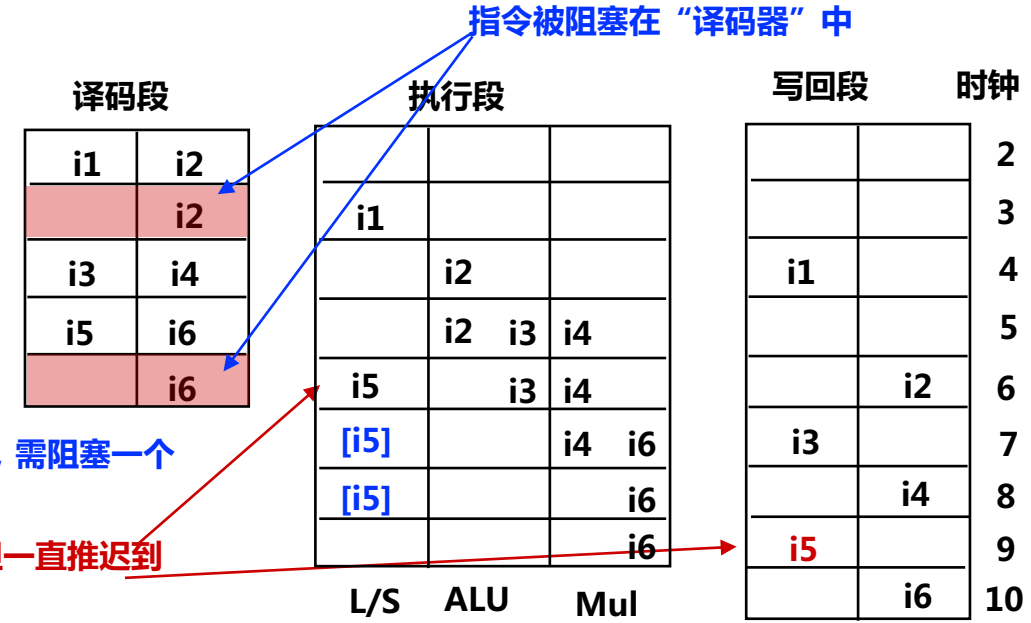
保留站：存放操作数和操作命令

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！

为了按序完成，虽i5在时钟6已完成，但一直推迟到i4写回后才写回i5

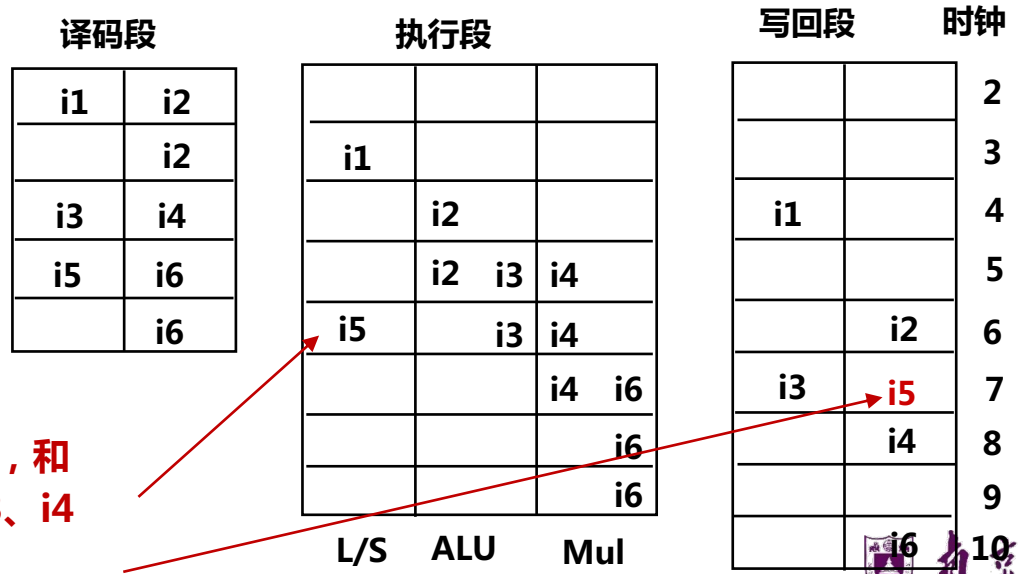
如果还有一条乘法指令，则最多可有三条乘法指令同时在执行

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。

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。

无序发射无序完成的过程如下：

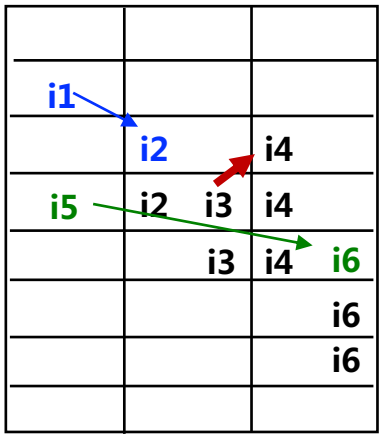
i1	lw $1, A
i2	add $2, $2, $1
i3	add $3, $3, $4
i4	mul $4, $5, $4
i5	lw $6, B
i6	mul $6, $6, $7

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!
(例如：i4在i3前面发射！)

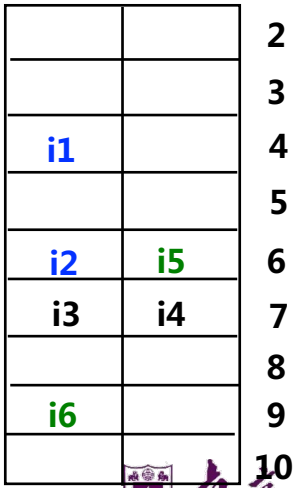
译码段

i1	i2
i3	i4
i5	i6

执行段



写回段 **时钟**



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！

动态流水线调度的必要性

- **编译器可依据数据依赖关系调度代码，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。

课程习题（作业）——截止日期：11月25日晚23:59

- **课本203-205页**：第4、5、6、7、9、10、11、12题
- 提交方式：https://selearning.nju.edu.cn/（教学支持系统）

第6章-指令流水线-课后习题

课本203-205页：第4、5、6、7、9、10、11、12题

- 命名：学号+姓名+第*章。
- 若提交遇到问题请及时发邮件或在下一次上课时反馈。

课程习题（作业）——截止日期：11月25日晚23:59

4. 假定某计算机工程师想设计一个新的CPU，一个典型程序的核心模块有一百万条指令，每条指令执行时间为100ps。请问：

（1）在非流水线处理器上执行该程序需要花多长时间？

（2）若新CPU采用20级流水线，执行上述同样的程序，理想情况下，它比非流水线处理器快多少？

（3）实际流水线并不是理想的，流水段之间的数据传送会有额外开销。这些开销是否会影响指令执行时间和指令吞吐率？

5. 假定最复杂的一条指令所用的组合逻辑分成6部分，依次为A～F，其延迟分别为80ps、30ps、60ps、50ps、70ps、10ps。在这些组合逻辑块之间插入必要的流水段寄存器就可实现相应的指令流水线，寄存器延迟为20ps。理想情况下，以下各种方式所得到的时钟周期、指令吞吐率和指令执行时间各是多少？应该在哪里插入流水段寄存器？

（1）插入1个流水段寄存器，得到一个两级流水线。

（2）插入2个流水段寄存器，得到一个三级流水线。

（3）插入3个流水段寄存器，得到一个四级流水线。

（4）吞吐量最大的流水线。

课程习题（作业）——截止日期：11月25日晚23:59

6. 以下指令序列中，哪些指令对之间发生数据相关？假定采用“取指、译码/取数、执行、访存、写回”5段流水线方式，如果不用“转发”技术，需要在发生数据相关的指令前加入几条 nop 指令才能使这段程序避免数据冒险？如果采用“转发”是否可以完全解决数据冒险？不行的话，需要在发生数据相关的指令前加入几条 nop 指令才能使这段 MIPS 程序不发生数据冒险？

```
addu    $s3, $s1, $s0
addu    $t2, $s3, $s3
lw      $t1, 0($t2)
add     $t3, $t1, $t2
```

7. 假定以下 MIPS 指令序列在图 6.18 所示的流水线数据通路中执行：

```
addu    $s3, $s1, $s0
subu    $t2, $s3, $s3
lw      $t1, 0($t2)
add     $t3, $t1, $t2
add     $t1, $s4, $s5
```

请问：

(1) 上述指令序列中，哪些指令的哪个寄存器需要转发？转发到何处？

(2) 上述指令序列中，是否存在 load-use 数据冒险？

(3) 第 5 周期结束时，各指令执行状态是什么？哪些寄存器的数据正被读出？哪些寄存器将被写入？

课程习题（作业）——截止日期：11月25日晚23:59

9. 在一个带转发的5段流水线中执行以下MIPS程序段，怎样调整指令序列使其性能达到最好？

```
lw    $2, 100($6)
add   $2, $2, $3
lw    $3, 200($7)
add   $6, $4, $7
sub   $3, $4, $6
lw    $2, 300($8)
beq   $2, $8, Loop
```

10. 在一个采用“取指、译码/取数、执行、访存、写回”的5段流水线中，若检测结果是否为“0”和将转移目标地址(Btarg和Jtarg)送PC的操作在执行阶段进行，则分支延迟损失时间片(即分支延迟槽)为多少？在带转发的5段流水线中，对于以下MIPS指令序列，哪些指令执行时会发生流水线阻塞？各需要阻塞几个时钟周期？

```
Loop:  add    $t1, $s3, $s3
       add    $t1, $t1, $t1
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       add    $s3, $s3, $s4
       j      Loop
Exit:
```

课程习题（作业）——截止日期：11月25日晚23:59

11. 假设数据通路中各主要功能部件的操作时间是：存储单元为 200ps；ALU 和加法器为 100ps；寄存器堆读口或写口为 50ps。程序中指令的组成比例为：取数为 25%、存数为 10%、ALU 为 52%、分支为 11%、跳转为 2%。假设控制单元和传输线路等延迟都忽略不计，则以下实现方式中哪个更快？快多少？

（1）单周期方式。每条指令在一个固定长度的时钟周期内完成。

（2）多周期方式。时钟周期取存储单元操作时间的一半，每类指令时钟数是：取数为 7、存数为 6、ALU 为 5、分支为 4、跳转为 4。

（3）流水线方式。时钟周期取存储操作时间的一半，采用“取指 1、取指 2、取数/译码、执行、存取 1、存取 2、写回”7 段流水线；没有结构冒险；数据冒险采用“转发”技术处理；load 指令与后续各指令之间存在依赖关系的概率分别 1/2，1/4，1/8，…；分支延迟损失时间片为 2，预测准确率为 75%；不考虑异常、中断和访问缺失引起的流水线冒险。

课程习题（作业）——截止日期：11月25日晚23:59

12. 有一段程序的核心模块中有 5 条分支指令，该模块将会被执行成千上万次，在其中一次执行过程中，5 条分支指令的实际执行情况如下（T：taken；N：not taken）。

分支指令 1：T-T-T。

分支指令 2：N-N-N-N。

分支指令 3：T-N-T-N-T-N。

分支指令 4：T-T-T-N-T。

分支指令 5：T-T-N-T-T-N-T。

假定各个分支指令在每次模块执行过程中实际执行情况都一样，并且动态预测时每个分支指令都有自己的预测表项，每次执行该模块时的初始预测位都相同。请分析并给出以下几种预测方案的预测准确率。

（1）静态预测，总是预测转移（taken）。

（2）静态预测，总是预测不转移（not taken）。

（3）一位动态预测，初始预测转移（taken）。

（4）二位动态预测，初始预测弱转移（taken）。

提问

Q & A

殷亚凤
智能软件与工程学院
苏州校区南雍楼东区225
yafeng@nju.edu.cn，https://yafengnju.github.io/

南京大学 NANJING UNIVERSITY