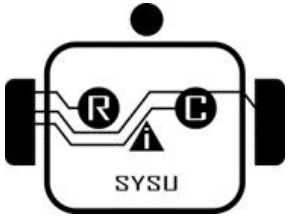
中山大學
SUN YAT-SEN UNIVERSITY



计算机组成原理

第四章：MIPS CPU逻辑设计

中山大学计算机学院
陈刚

2022年秋季

流水线竞争

在流水线各个阶段,指令之间可能存在关联

✓– 下一条指令可能需要某些无法获得的资源
(访问权?新的数值?到底该怎么办的指示等等)

✓– If all instructions are dependent

- No advantage of a shorter cycle (since all must wait)
- This will limit clock cycle time

这些限制就是

——**流水线的竞争Pipeline Hazards**

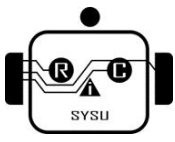
流水线竞争

- **Structural Hazard (Resource Conflict)**

  ——结构竞争

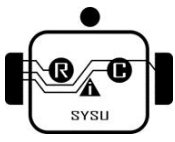
  - 两条指令需要使用同样的硬件块
- **Data Hazard——数据关联竞争**
  - 指令依赖于流水线中运行的其他指令结果
- **Control Hazard控制竞争**
  - 取指令依赖于流水线中指令的结果

Control Hazards: Deal with instructions fetched before a branch or jump determines the next PC.



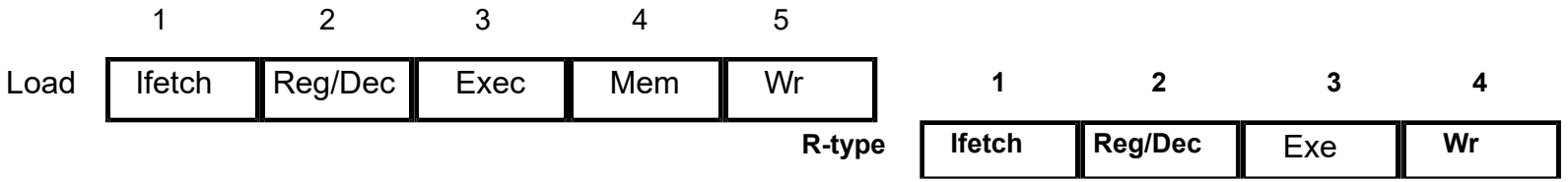
流水线冒险的处理

- 流水线冒险的几种类型
- 数据冒险的现象和对策
  - 数据冒险的种类
    - 相关的数据是ALU结果：可以通过转发解决
    - 相关的数据是DM读出的内容：随后的指令需被阻塞一个时钟
  - 数据冒险和转发
    - 转发检测 / 转发控制
  - 数据冒险和阻塞
    - 阻塞检测 / 阻塞控制
- 控制冒险的现象和对策
  - 静态分支预测技术
  - 动态分支预测技术
  - 缩短分支延迟技术
- 流水线中对异常和中断的处理
- 访问缺失对流水线的影响

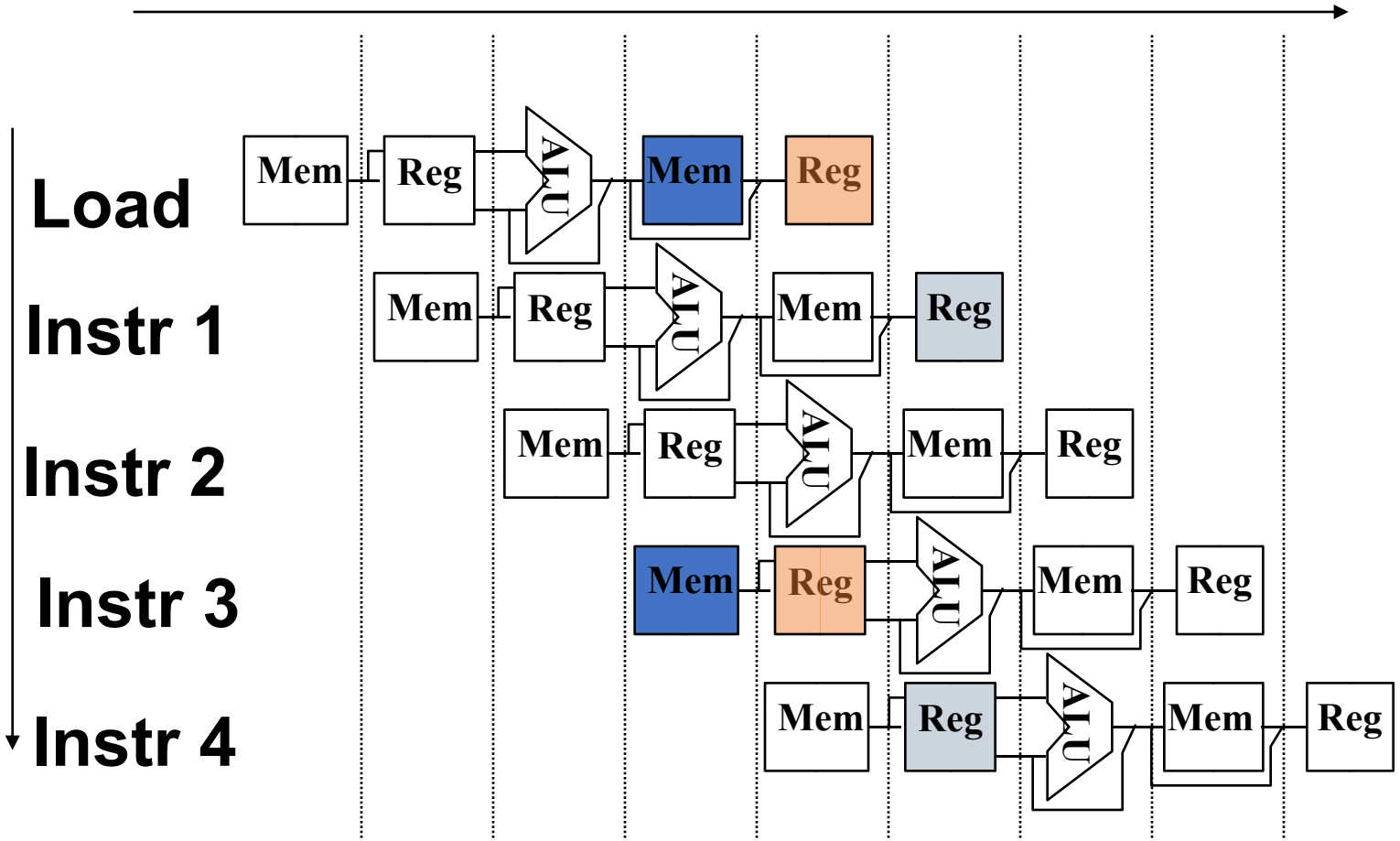


结构流水线竞争

- **资源使用冲突**
  - ✓定义为: 当两条指令需要同时访问相同硬件资源时
  - ✓经常当某个单元未完全按照流水线方式实现时发生
- **为避免结构竞争**
  - ✓限制资源使用方式,每个指令中只对某资源使用一次
  - ✓对相同资源的访问对于不同类型的指令都在同一周期进行
- **引起结构竞争的例子:**
  - ✓对load指令采用5级流水第5级回写寄存器
  - ✓对R类指令采用4级流水第4级回写寄存器



Structural Hazard（结构冒险）现象

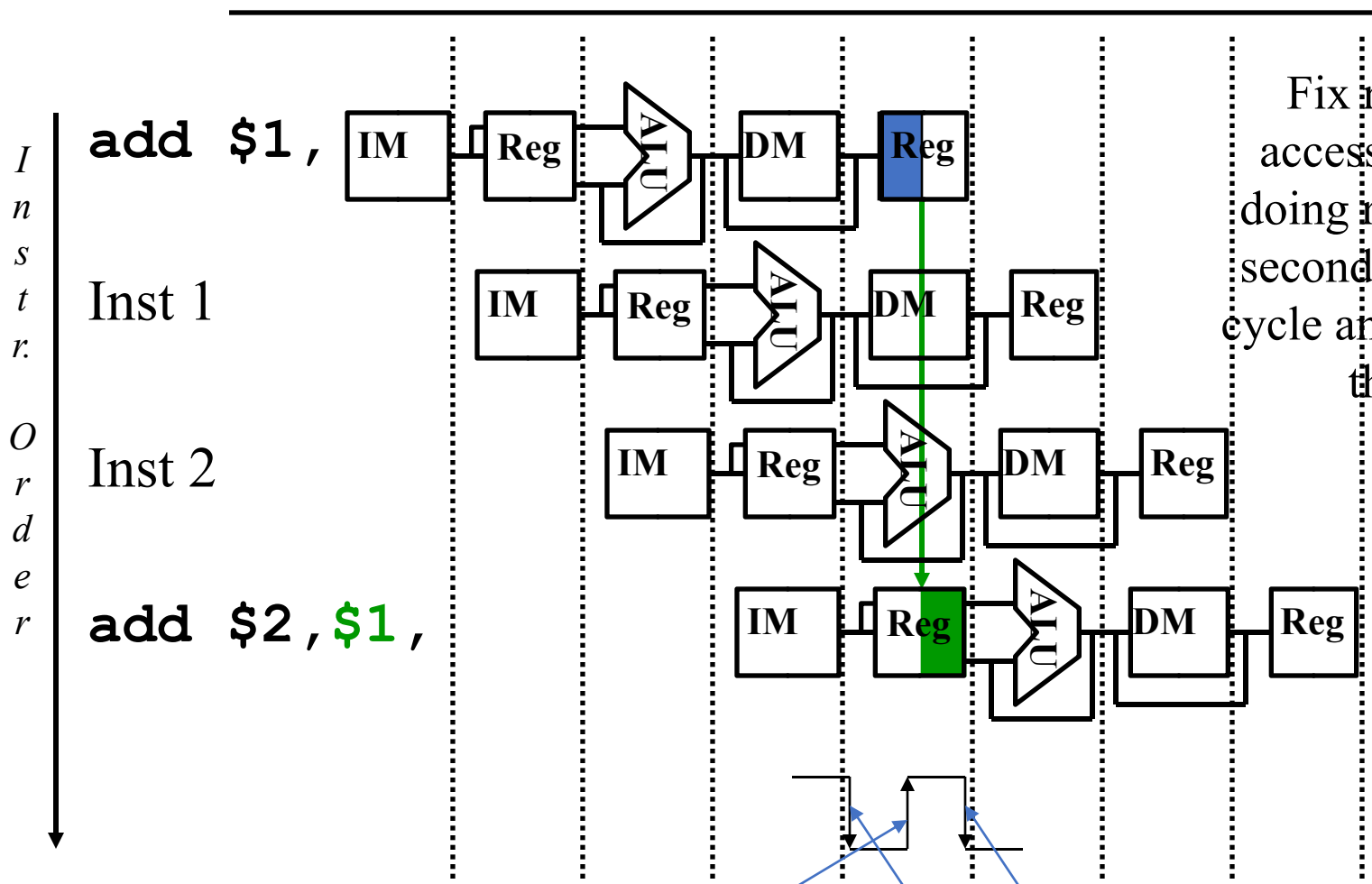


如果只有一个存储器，则在Load指令取数据同时又取指令的话，则发生冲突！

如果不对寄存器堆的写口和读口独立设置的话，则发生冲突！

结构冒险也称为硬件资源冲突：同一个执行部件被多条指令使用。

How About Register File Access?



Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

通过使写入操作（上升沿触发）总在一个周期的半段完成而读出（下降沿触发）总在半段完成来避免此问题.

clock edge that controls register writing

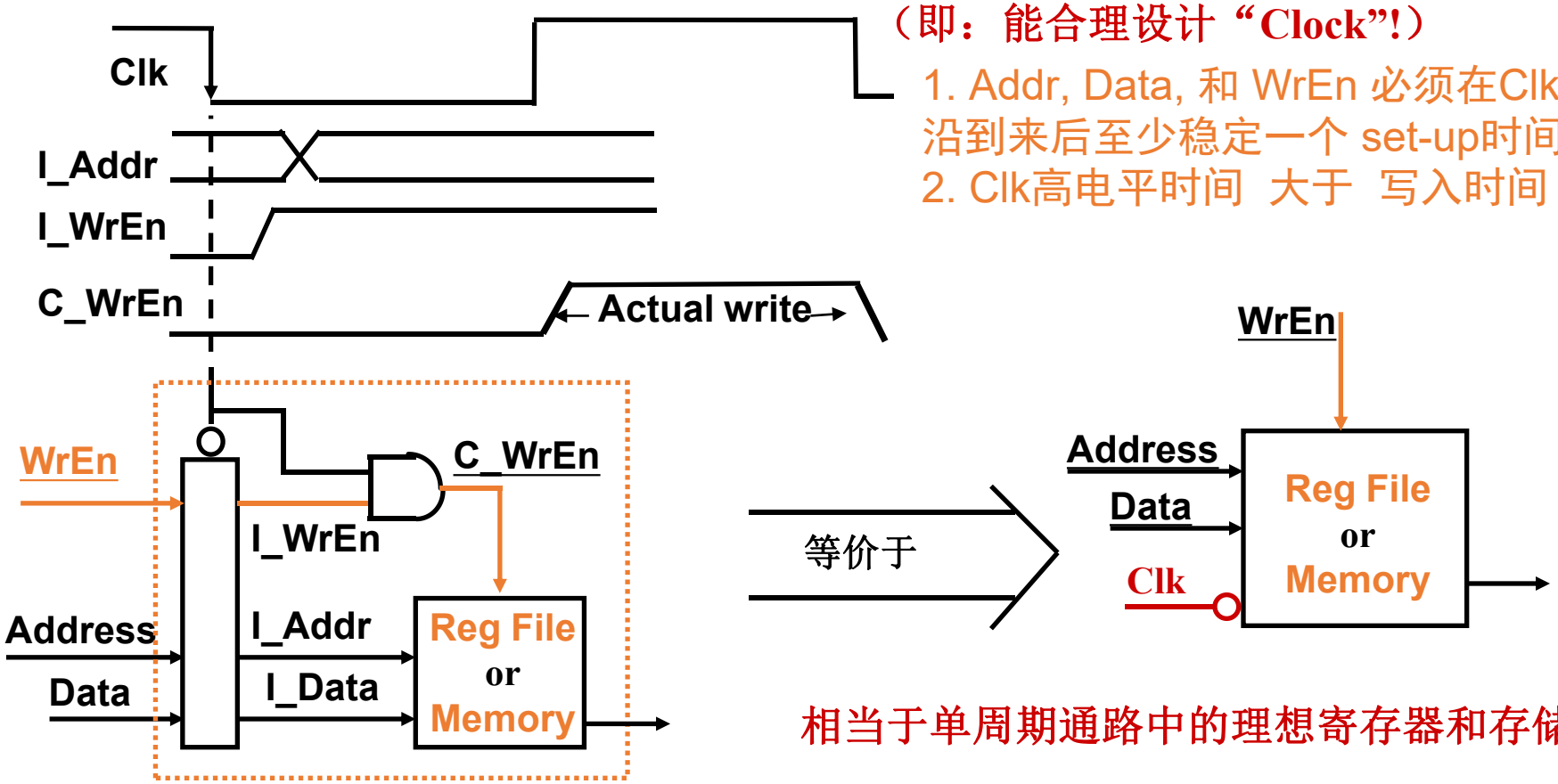
clock edge that controls loading of pipeline state registers

寄存器组的同步和存储器的同步

- 解决方案： 将Write Enable和时钟信号“与”

须由电路专家确保不会发生“定时错误”
（即：能合理设计“Clock”!）

1. Addr, Data, 和 WrEn 必须在Clk边沿到来后至少稳定一个 set-up时间
2. Clk高电平时间 大于 写入时间



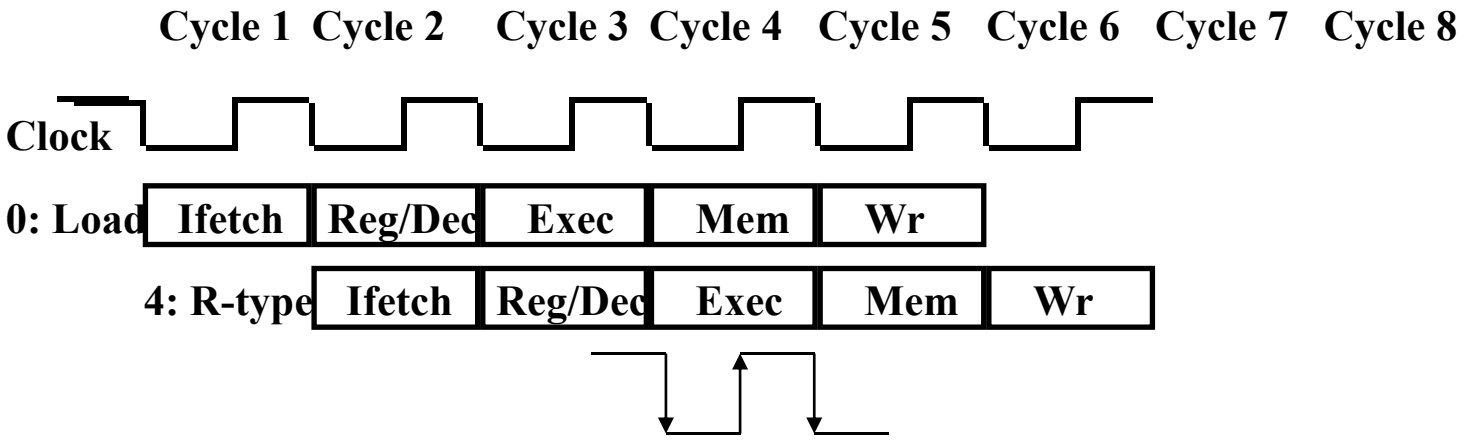
相当于单周期通路中的理想寄存器和存储器

Load 指令采用第五周期上升沿写入的状态：

Clk4上升沿后
指令流：结果存入MEM/WB

Clk5 开始的下降沿从MEM/WB读取到的addr和data有关数据送到寄存器，
Clk5 上升沿 结果存入寄存器

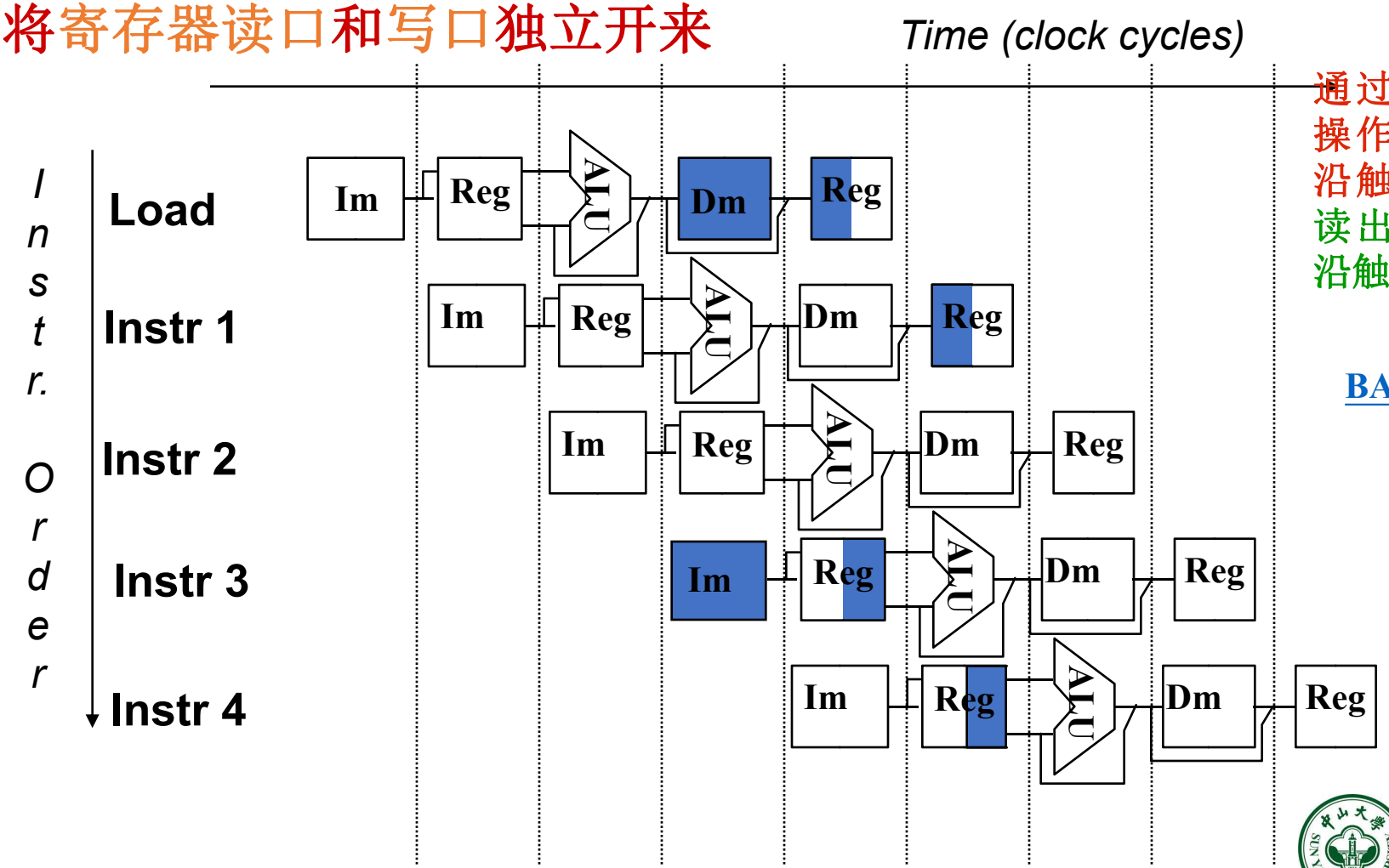
如果采用转发技术，在Clk5 开始的下降沿即可从MEM/WB读取到有关数据



仍需延迟一个时钟周期

Structural Hazard的解决方法

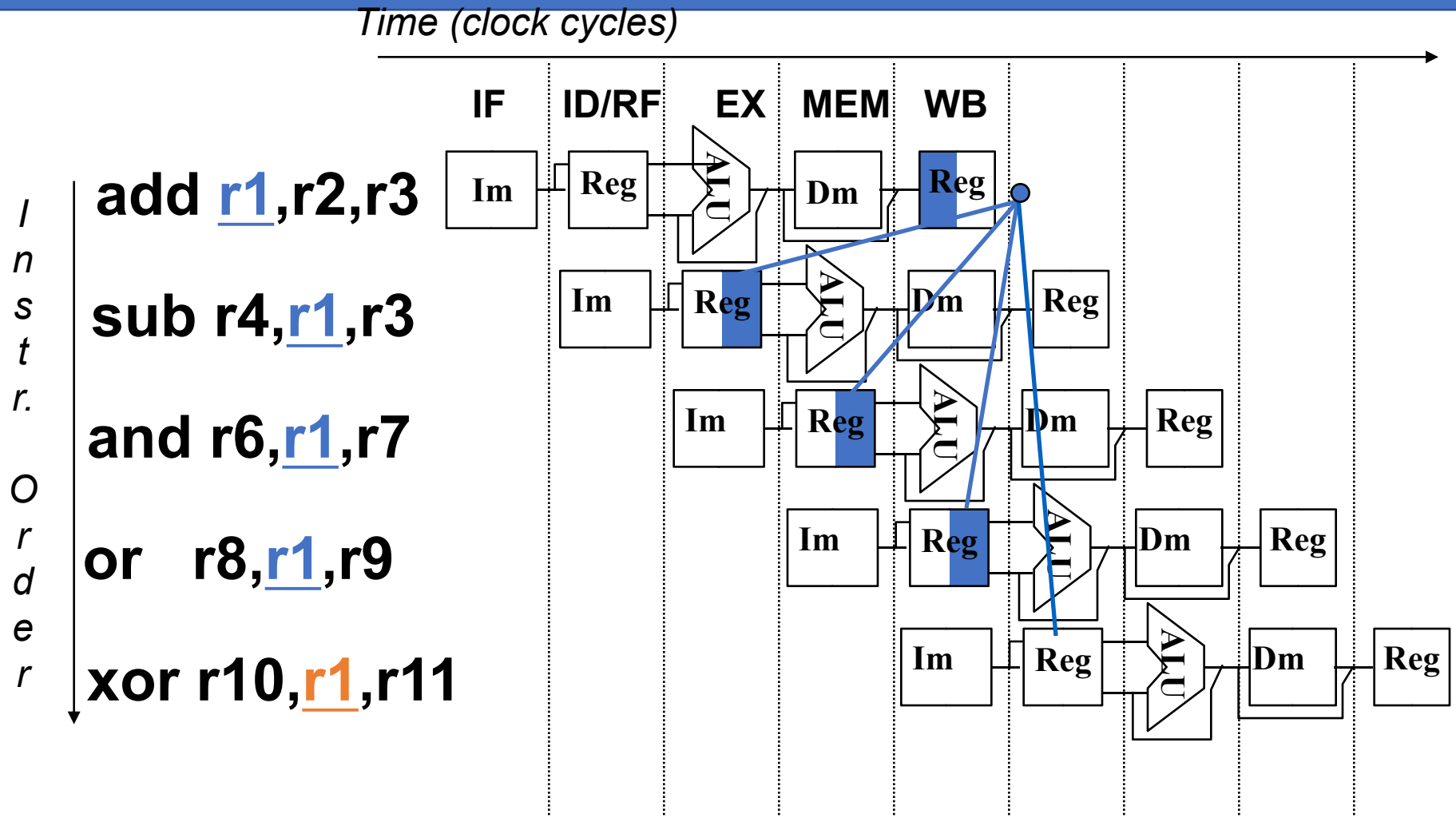
将Instruction Memory (Im) 和
Data Memory (Dm)分开
将寄存器读口和写口独立开来



通过使写入操作（上升沿触发）而读出（下降沿触发）

BACK

Data Hazard on r1



最后一条指令的r1才是新的值！

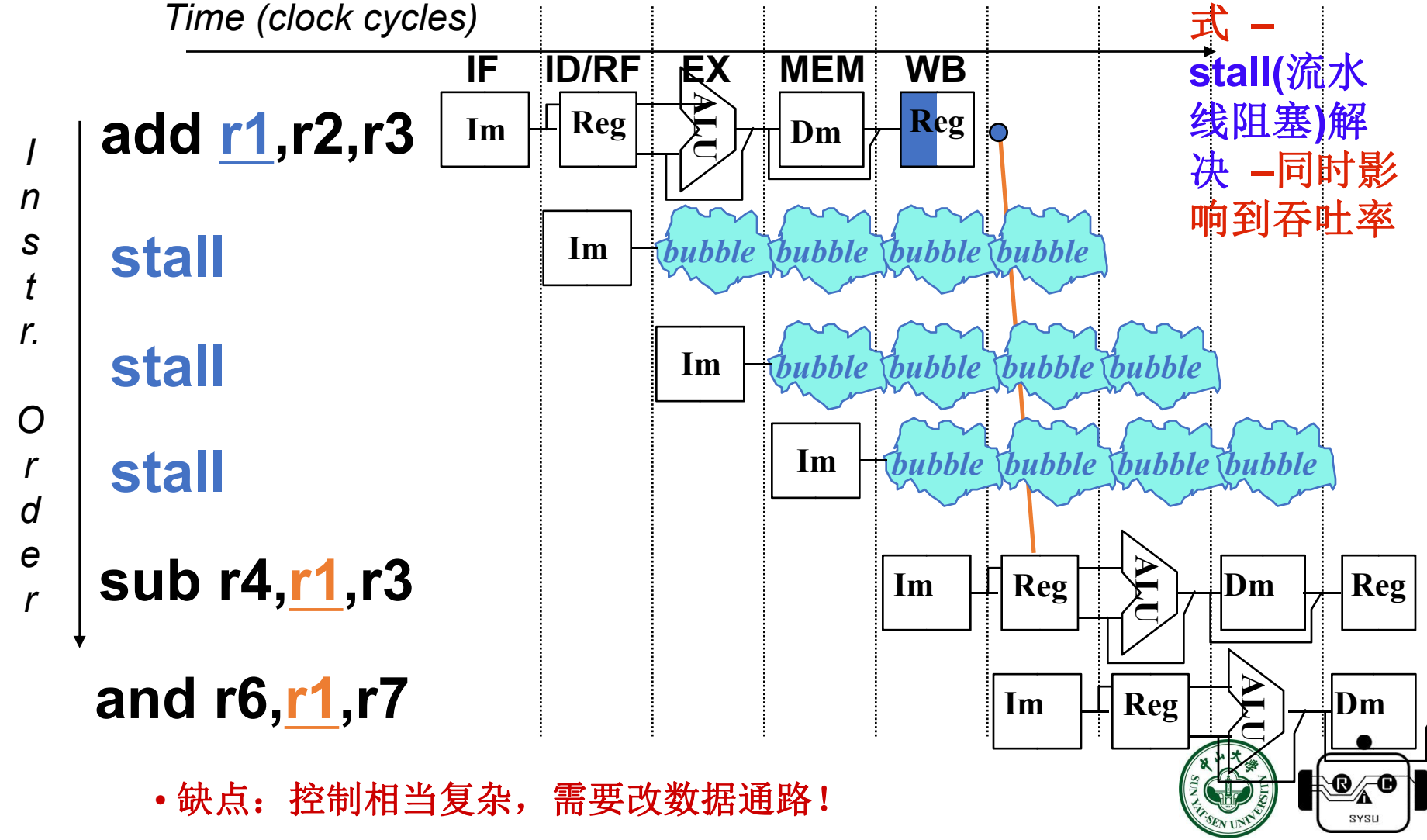
如何解决这个问题？

方案1：在硬件上采取措施，使相关指令延迟执行

- 硬件上通过阻塞(stall)方式阻止后续指令执行，延迟到有新值以后！

这种做法称为流水线阻塞，也称为“气泡Bubble”

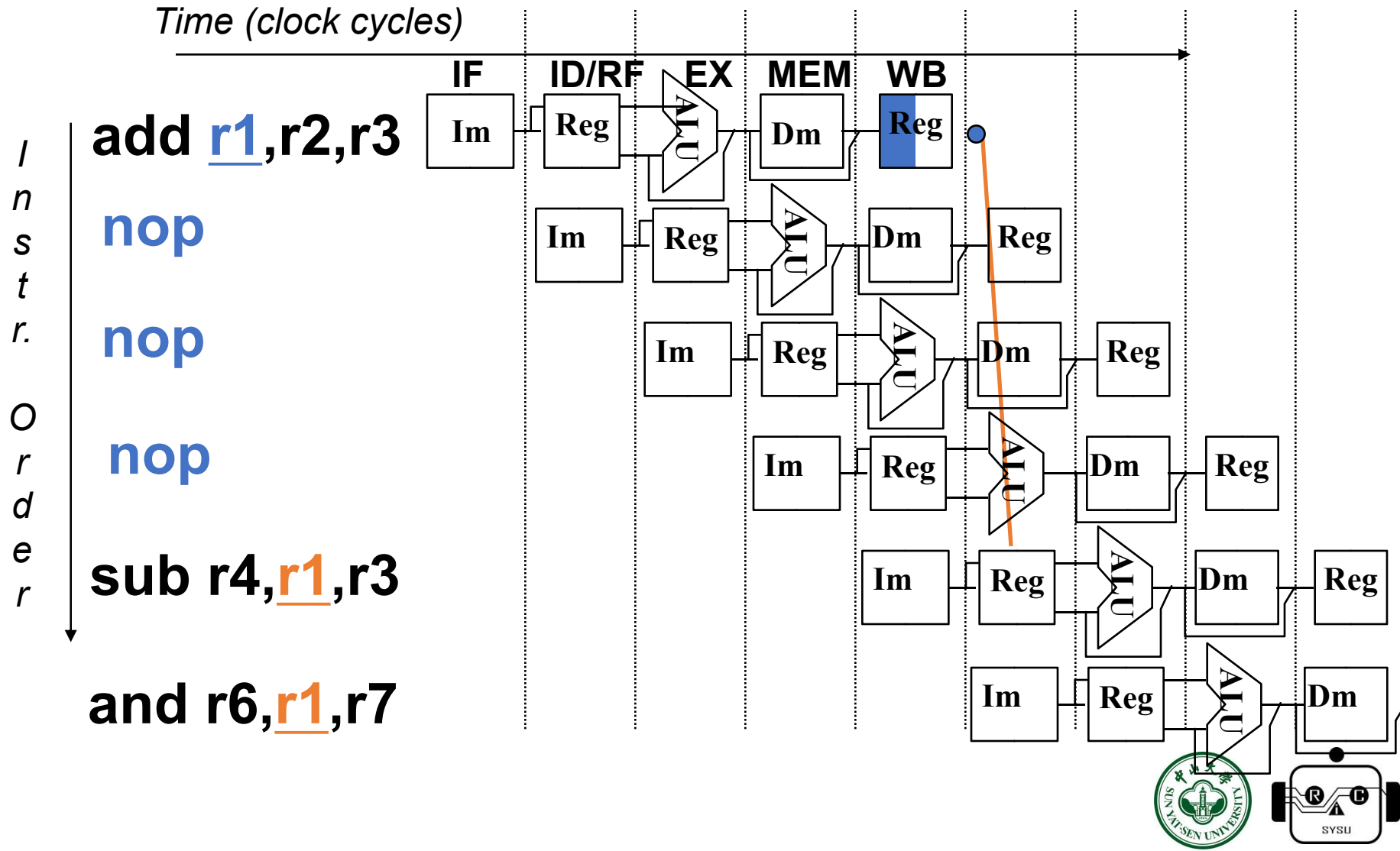
数据竞争可用等待的方式 –stall(流水线阻塞)解决 –同时影响到吞吐率



- 缺点：控制相当复杂，需要改数据通路！

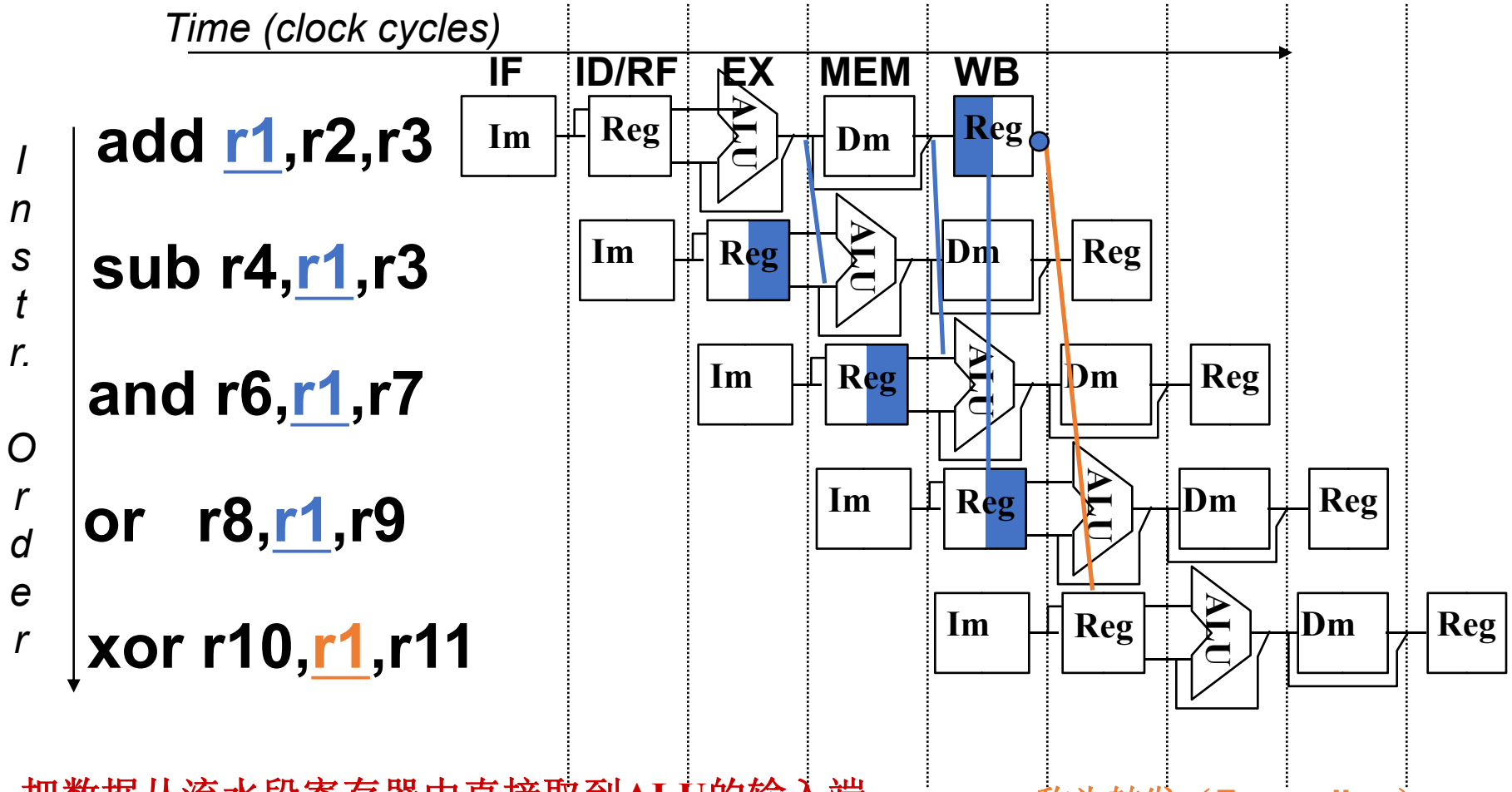
方案 2：软件上插入无关指令

- 最差的做法：由编译器插入三条NOP指令，浪费三条指令的空间和时间



方案3： 利用DataPath中的中间数据

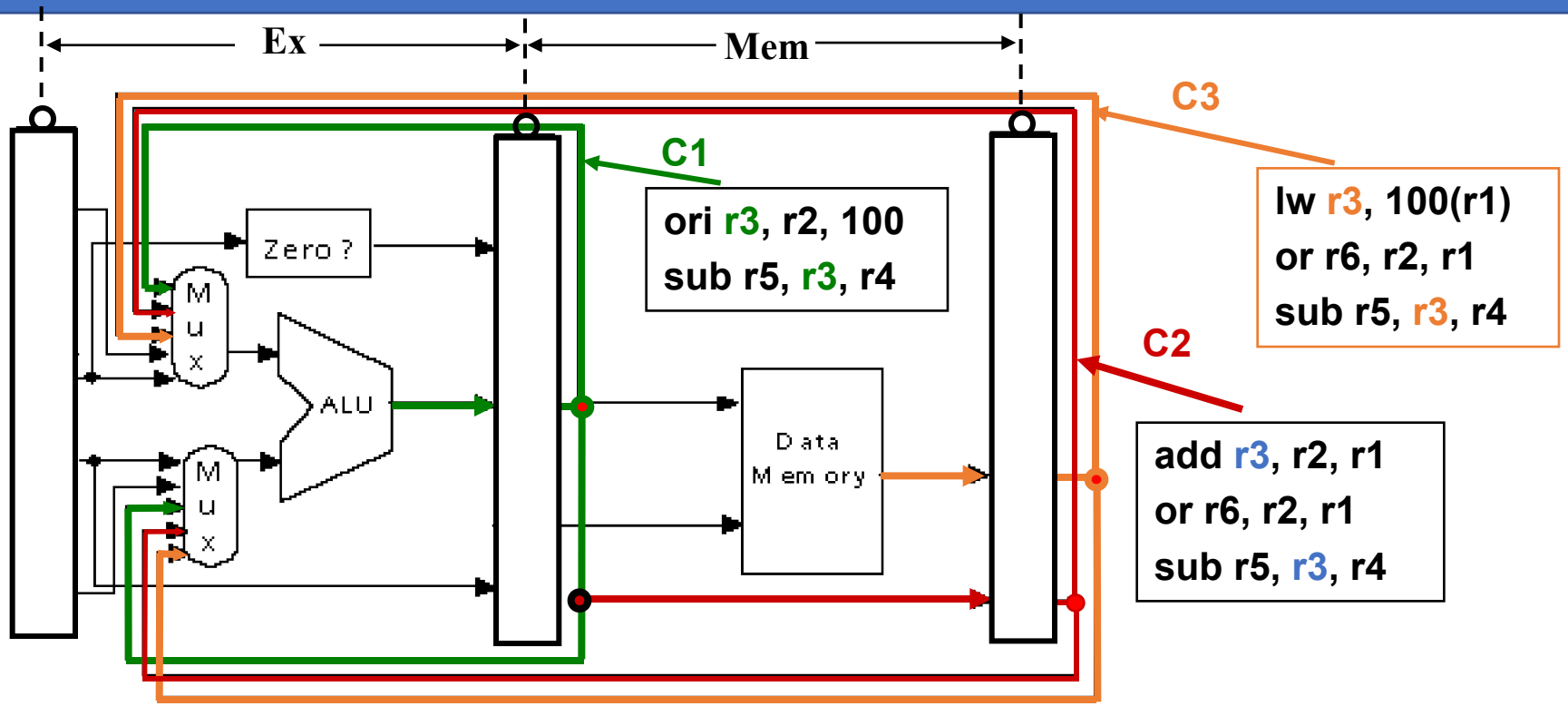
- 仔细观察后发现：流水段寄存器中已有需要的值r1！ 在哪个流水段R中？



1. 把数据从流水段寄存器中直接取到ALU的输入端
2. 寄存器写/读口分别在前/后半周期，使写入被直接读出

称为转发（Forwading）
或旁路（Bypassing）

RAW（写后读）数据冒险的“转发”条件



后面指令需用ALU输出结果

C1: 目寄是后一条指令的源寄

C2: 目寄是后第二条指令的源寄

(例如：R-Type后跟R- / lw / sw / beq等)

后面指令需用从DM读出的结果

C3: 目寄是后第二指令的源寄

(例如：load指令后跟R-Type / beq等)

用流水段寄存器来表示转发条件（C3以后考虑）

C1(a): EX/MEM. RegisterRd=ID/EX. RegisterRs

C1(b): EX/MEM. RegisterRd=ID/EX. RegisterRt

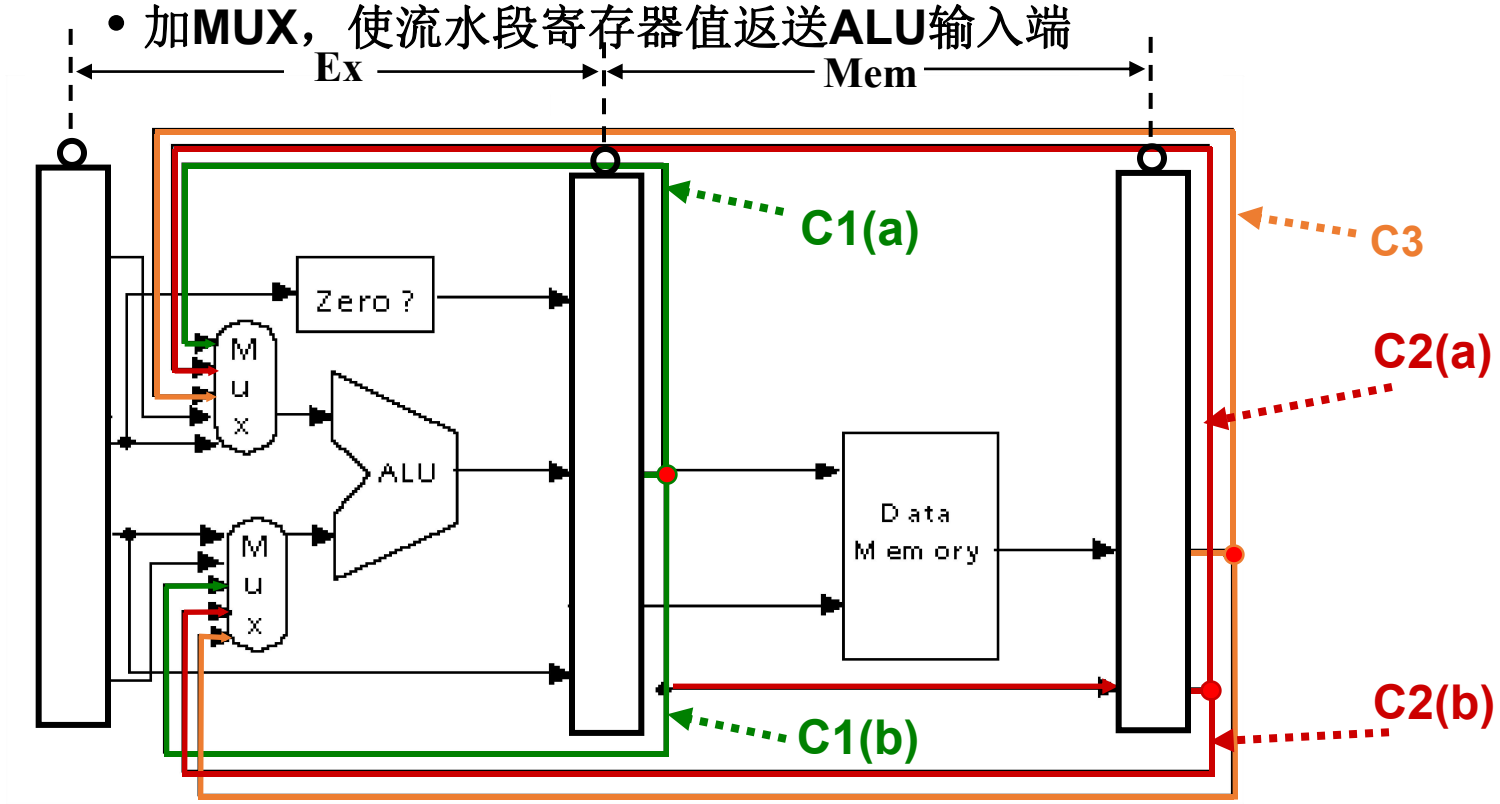
C2(a): MEM/WB. RegisterRd=ID/EX. RegisterRs

C2(b): MEM/WB. RegisterRd=ID/EX. RegisterRt

这里的RegisterRd是指目的寄存器

实际上是R-type的Rd 或 I-Type的rt

转发路径和转发条件



C1反映本条指令和随后指令间的相关关系

C2反映本条指令和随后第二条指令间的相关关系

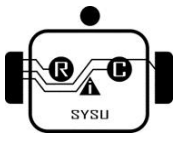
C1(a)和C1(b)可以合并为一个条件C1，并把转发线合一起后同时送A口和B口

即：C1=C1(a) or C1(b)，同样：C2=C2(a) or C2(b)，转发线合起来

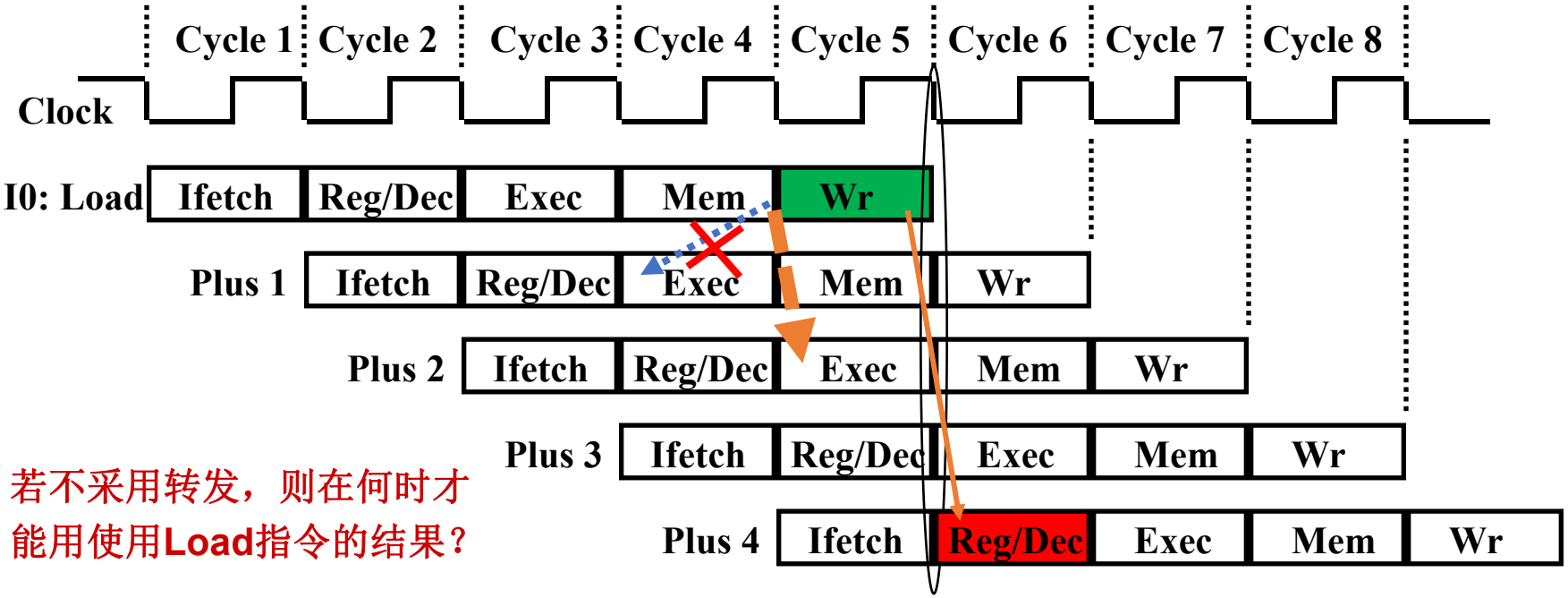
实际上红线和兰线可以合并，而且在原数据通路中是合并在一起的。记得吗？

由一个二路选择器（控制端为MemtoReg）合并输出到寄存器堆！

所以不需另外有一个检测条件C3!　　C1和C2分别反映哪两条指令的关系呢



复习：Load指令引起的延迟现象



若不采用转发，则在何时才能用使用Load指令的结果？

延迟2或3条指令！

- Load指令最早在哪个流水线寄存器中开始有后续指令需要的值？

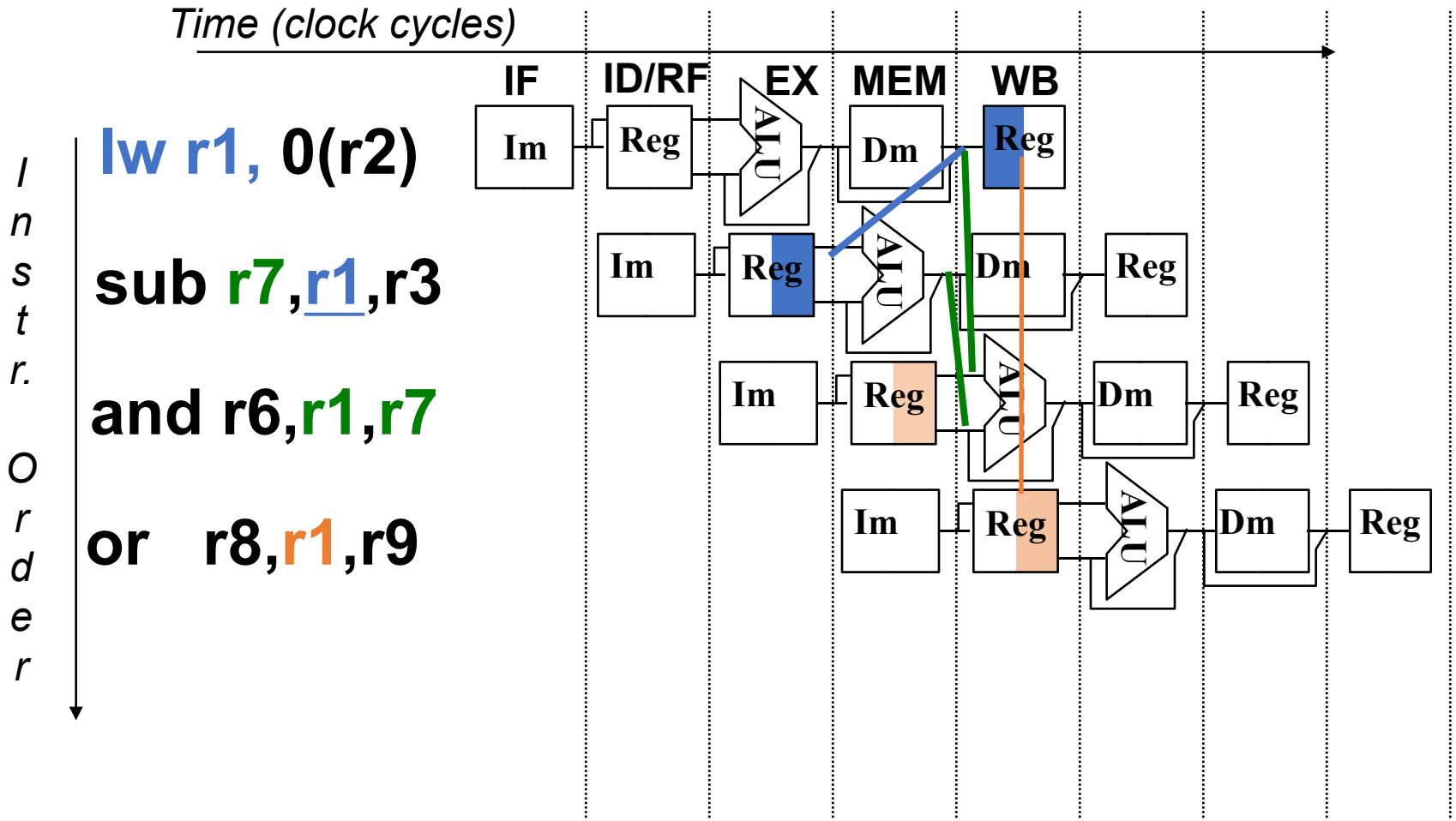
实际上，在第四周期结束时，数据在流水段寄存器中已经有值。

采用数据转发技术可以使load指令后面第二条指令得到所需的值

但不能解决load指令和随后的第一条指令间的数据冒险，要延迟执行一条指令！

这种load指令和随后指令间的数据冒险，称为“装入- 使用数据冒险(load- use Data Hazard)”

“Forwarding”技术使Load-use冒险只需延迟一个周期



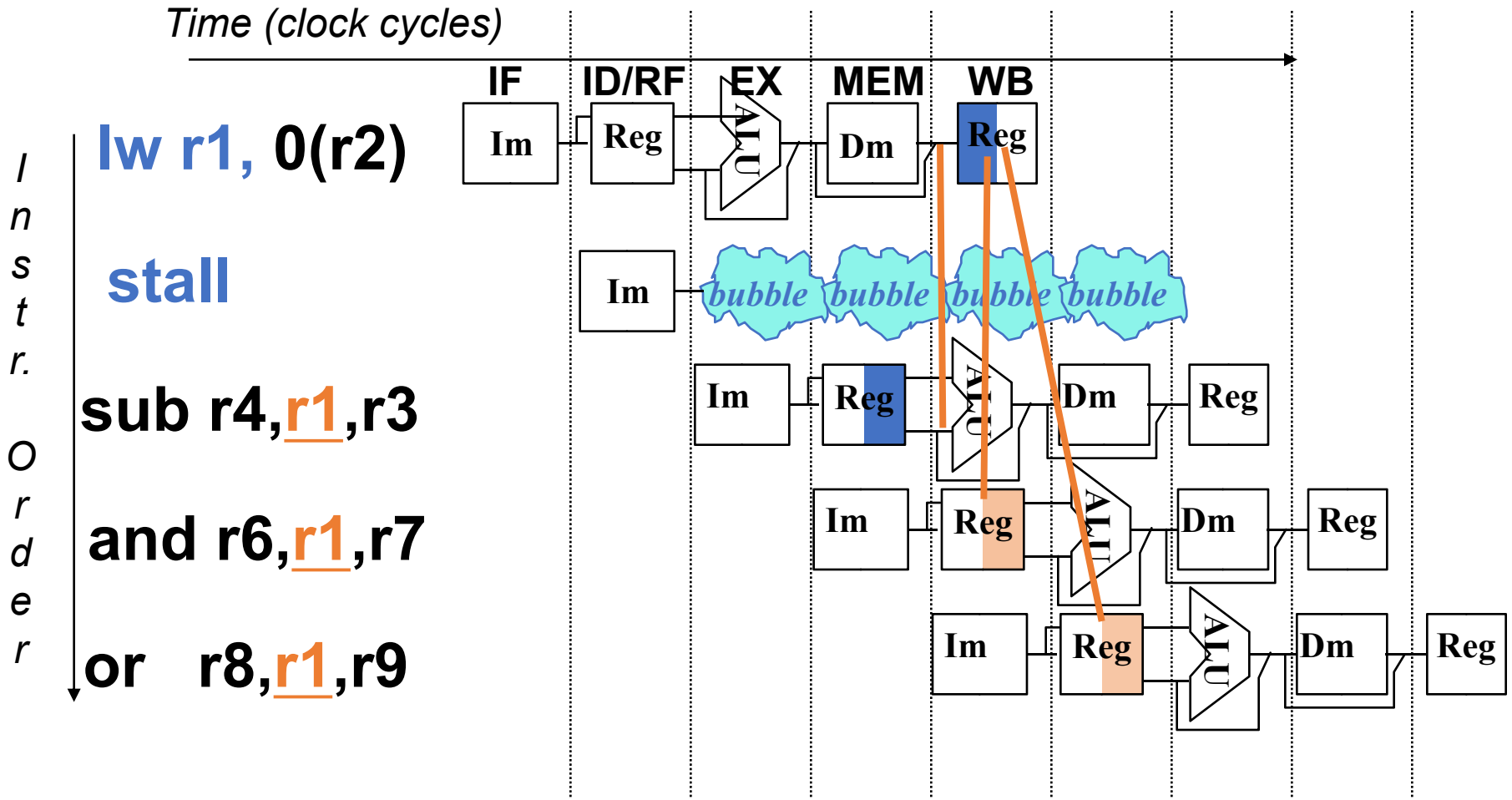
采用“转发”后仅第二条指令 **SUB r7,r1,r3** 不能按时执行！需要阻塞一个周期。

（其它后续指令顺延）

发生“装入- 使用数据冒险”时，需要对load后的指令阻塞一个时钟周期！

方案1：硬件阻止指令执行来解决load-use

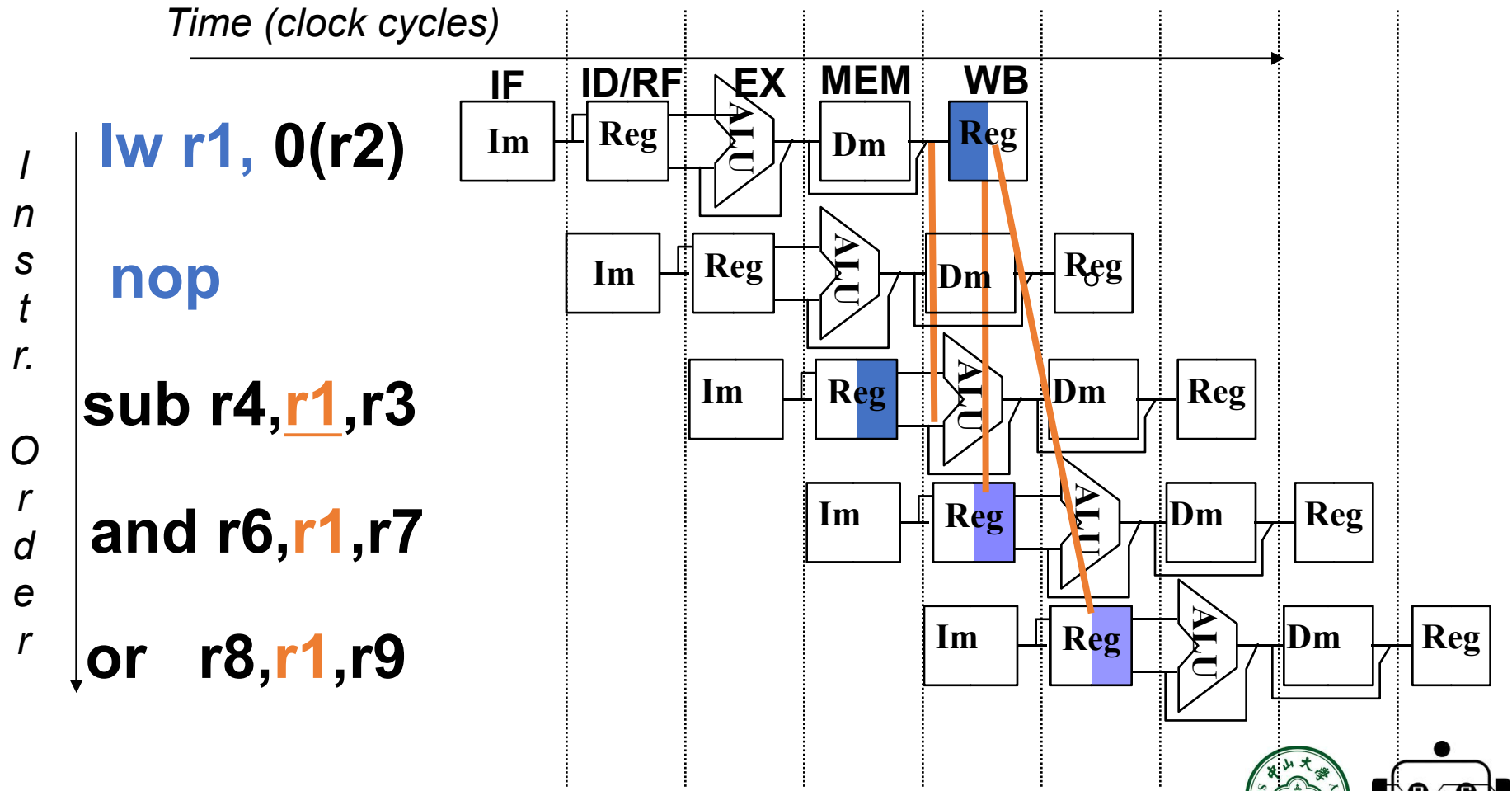
用硬件阻塞一个周期



方案2：软件上插入NOP指令来解决load-use

- 用软件插入一条NOP指令！（有些处理器不支持硬件阻塞处理）

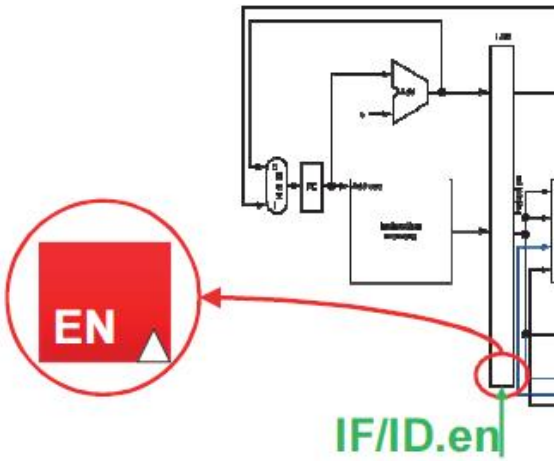
例如：MIPS 1 处理器没有硬件阻塞处理，而由编译器（或汇编程序员） 来处理。



如何插入NOP指令？

- 检测条件：IF/ID 的前序是lw指令，并且lw的rt寄存器与IF/ID的rs或rt相同
- 执行动作：
  - 1)冻结IF/ID:sub继续被保存
  - 2)清除ID/EX:指令全为0，等价于插入NOP
  - 3)禁止PC 防止PC继续计数，PC保持PC+4

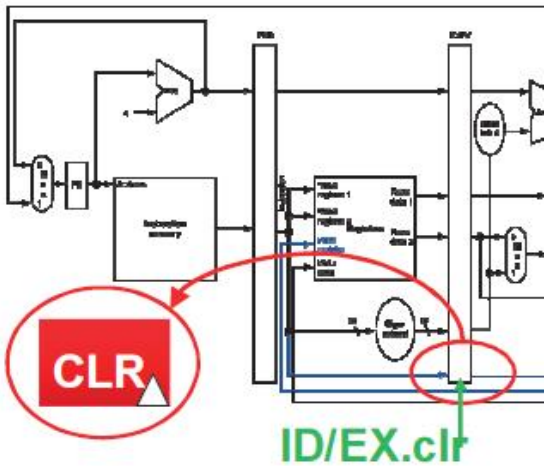
- 数据通路:将IF/ID修改为使能型寄存器
- 控制系统：增加IF/ID.en控制信号
  - 当IF/ID.en为0时， IF/ID在下一个clock上升沿
  - 到来时保持不变



如何插入NOP指令？

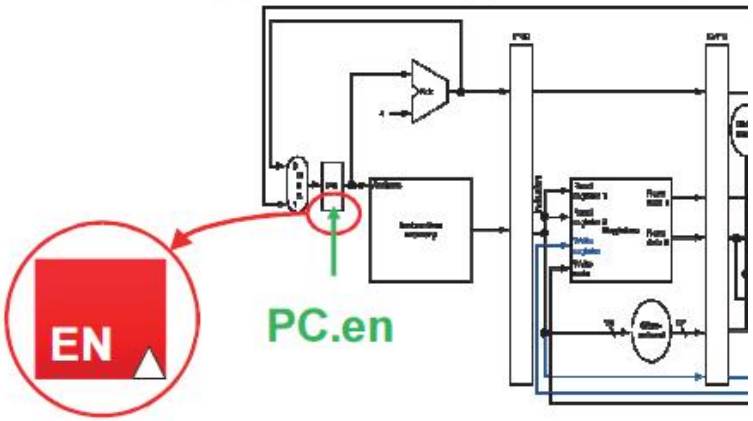
- 检测条件：IF/ID 的前序是lw指令，并且lw的rt寄存器与IF/ID的rs或rt相同
- 执行动作：
  - 1)冻结IF/ID:sub继续被保存
  - 2)清除ID/EX:指令全为0，等价于插入NOP
  - 3)禁止PC 防止PC继续计数，PC保持PC+4

- 数据通路:将ID/EX修改为复位型寄存器
- 控制系统：增加ID/EX.ctr控制信号
  - 当ID/EX.ctr为0时， ID/EX在下一个clock上升沿
  - 到来时被清除为0.



如何插入NOP指令？

- 检测条件： IF/ID 的前序是lw指令，并且lw的rt寄存器与IF/ID的rs或rt相同
- 执行动作：
  - 1)冻结IF/ID:sub继续被保存
  - 2)清除ID/EX:指令全为0，等价于插入NOP
  - 3)禁止PC 防止PC继续计数，PC保持PC+4



- 数据通路:将PC修改为使能型寄存器
- 控制系统： 增加PC. en控制信号
  - 当PC. en为0时， ID/EX在下一个clock上升沿
  - 到来时保持不变.

方案3：编译器进行指令顺序调整来解决load-use

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

Slow code:

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

Fast code:

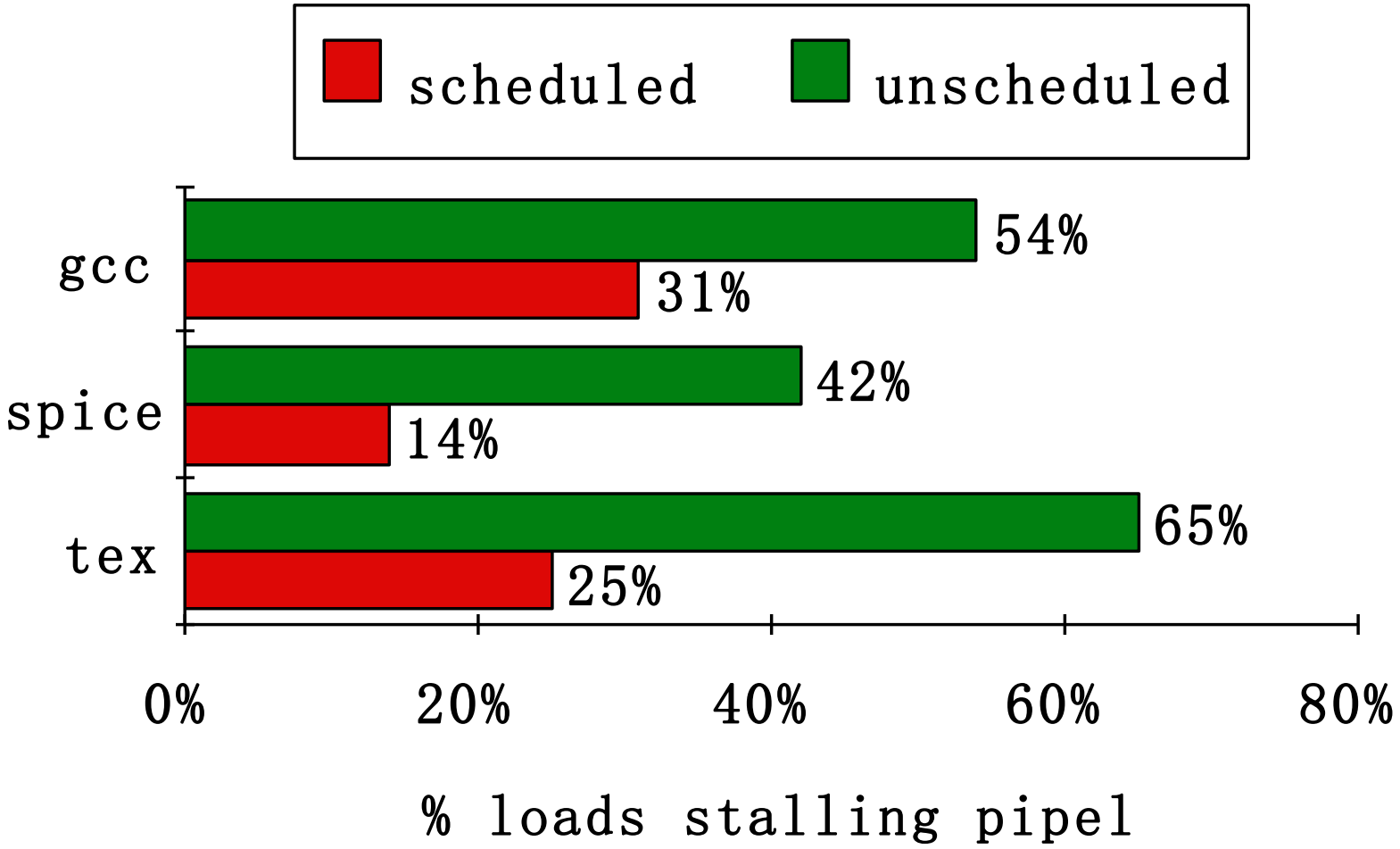
```
lw   $2, b
lw   $3, c
lw   $5, e
add  $1, $2, $3
lw   $6, f
sw   a, $1
sub  $4, $5, $6
sw   d, $4
```

真正执行时需阻塞2次

编译器的优化很重要！

真正执行时无需阻塞

编译器优化以避免阻塞的情况调查:



由此可见，优化调度后load阻塞现象大约降低了1/2~1/3

数据冒险的解决方法

- 方法1：硬件阻塞（stall）
- 方法2：软件插入“NOP”指令
- 方法3：编译优化：调整指令顺序，能解决所有数据冒险吗？
- 方法4：合理实现寄存器堆的读/写操作，能解决所有数据冒险吗？
  - 前半时钟周期写，后半时钟周期读，若同一个时钟内前面指令写入的数据正好是后面指令所读数据，则不会发生数据冒险
- 方法5：转发（Forwarding或Bypassing 旁路）技术，能解决所有数据冒险吗？
  - 若相关数据是ALU结果，则如何？

    可通过转发解决
  - 若相关数据是上条指令DM读出内容，则如何？

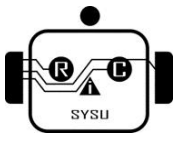
    不能通过转发解决，随后指令需被阻塞一个时钟 或 加NOP指令

称为Load-use数据冒险！

实现“转发”和“阻塞” 要修改数据通路：

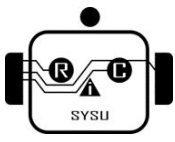
（1）检测何时需要“转发”，并控制实现“转发”

（2）检测何时需要“阻塞”，并控制实现“阻塞”

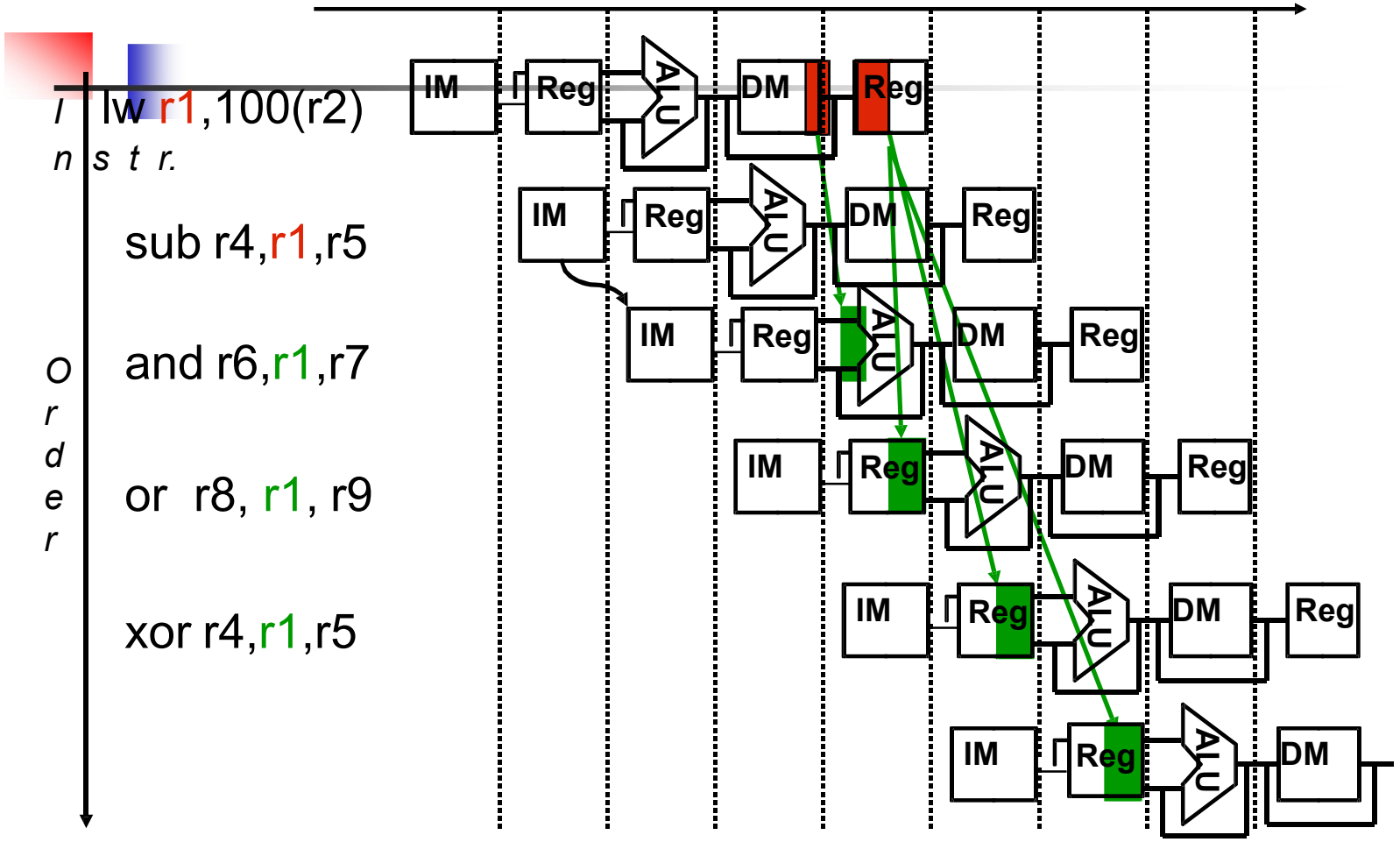


今天所有处理器都采用流水线技术

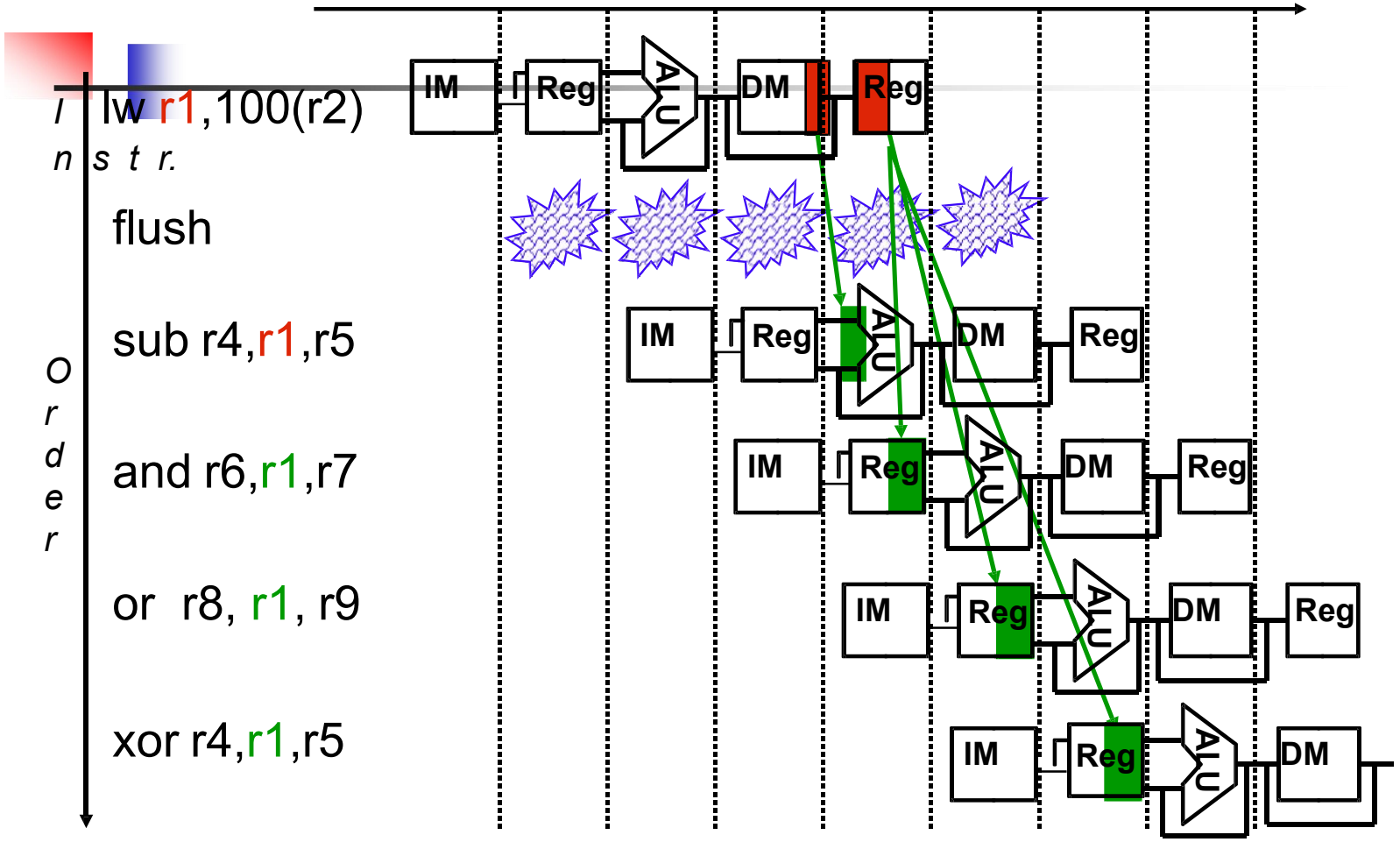
- 流水线提高吞吐率但是恶化延迟
- 流水线级数反映了潜在的吞吐率提升倍数
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced lengths of pipe stages reduces speedup
  - Time to “fill” pipeline and time to “drain” it reduces speedup
- 需要能够检测和处理竞争问题
  - 结构竞争—硬件资源使用冲突
  - 数据竞争—数据存储器或者寄存器内容因指令关联而不可获得ReadAfterWrite，load-use
  - 控制竞争—因跳转分支指令使得PC无法及时确定



Load-use Data Hazards的datapath forwarding



Load-use Data Hazards的datapath forwarding



Load-use Hazard 检测

- 因为load-use hazard无法通过forward完全解决，即便加了forward电路仍然存在hazard, 必须加一个阻塞周期
- **ID Hazard Detection**

if (ID/EX.MemRead

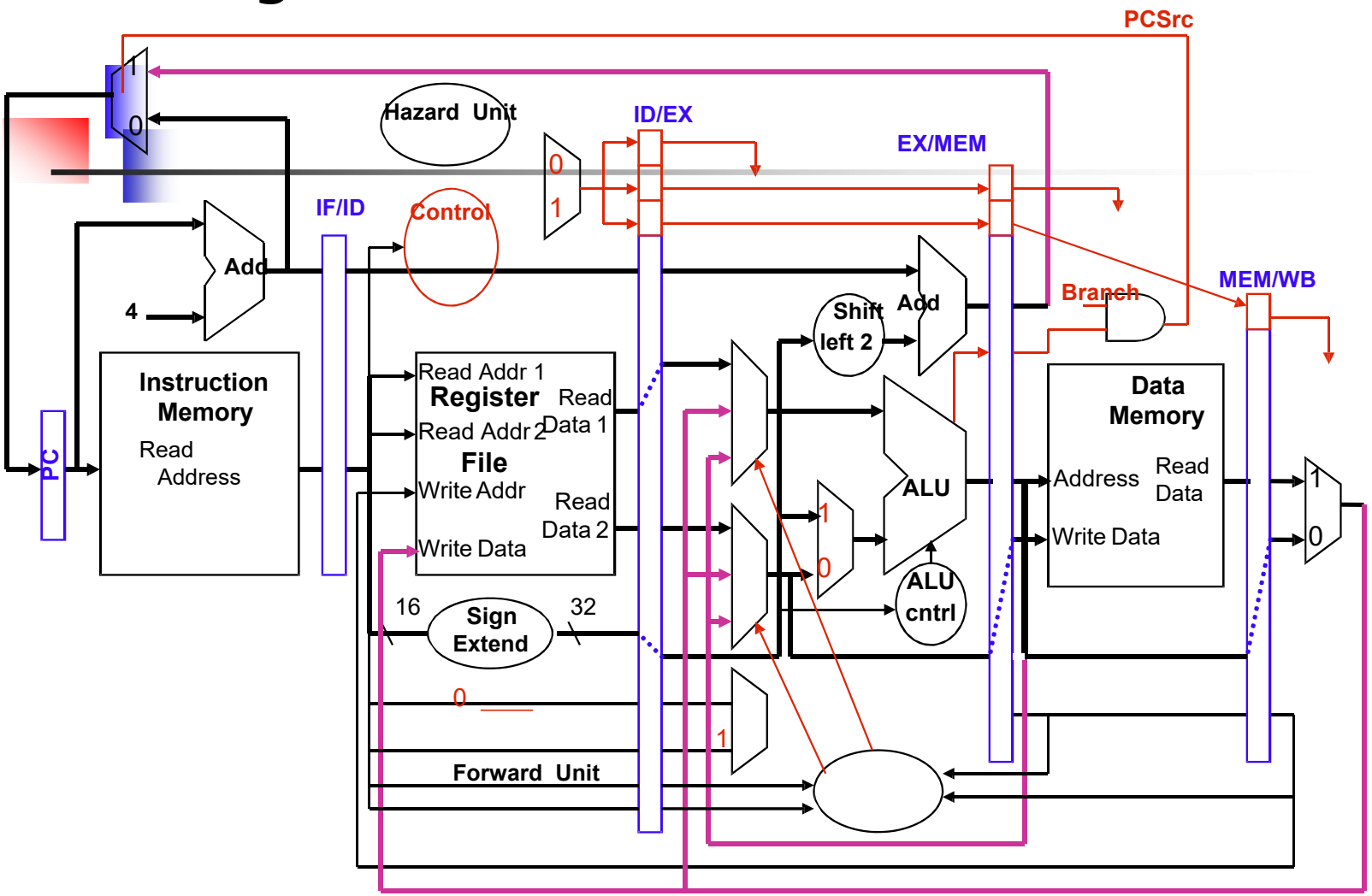
and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))) stall the pipeline

- 首先检查是否是load指令; 然后检查EX stage load指令的destination是否与ID阶段的两个source registers 是一样的
- 经过这个检测导致的阻塞, forwarding 逻辑就可以处理剩下的竞争问题了

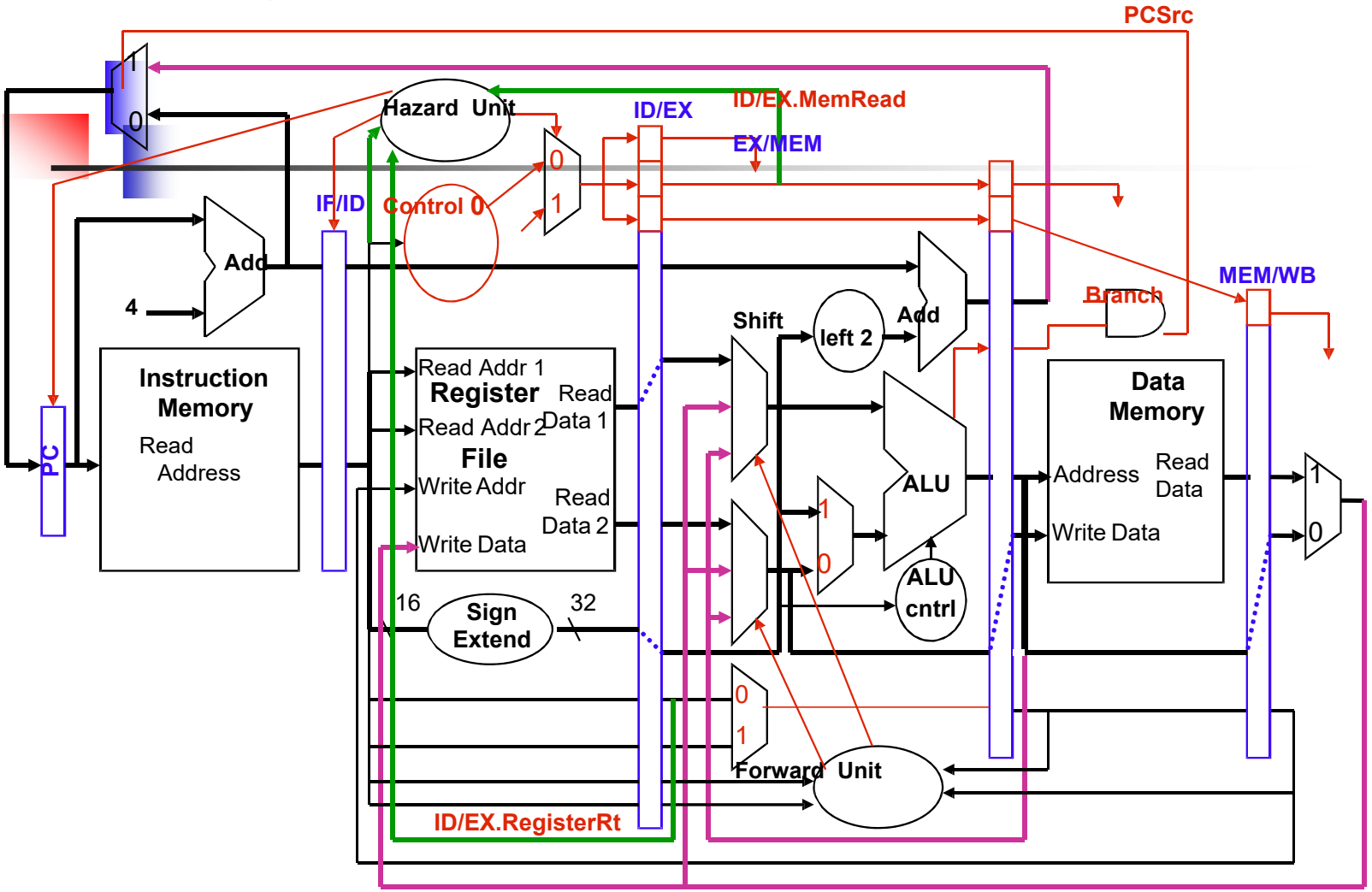
如何实现Stall 阻塞?

- 必须实现阻塞功能才能使forward发挥作用
- 阻止IF and ID 阶段的指令进入EX阶段执行, 只要不让PC寄存器变化和IF/ID流水线状态寄存器保持不变就可以了
  - 只要让前面的hazard检测电路来控制这些寄存器的写入就可以了
- 从EX开始的后半段在阻塞时应处于清空状态:执行NOP
  - 将 ID/EX pipeline register中与EX,MEM,WB有关的控制信号清零,不工作状态
  - 由Hazard检测电路控制选择使用正常的控制信号还是直接为0.
  - 假定控制信号为0总是使电路不工作或者不影响电路状态的

Adding the Hazard Hardware

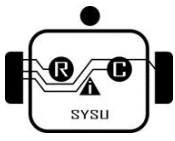


Adding the Hazard Hardware



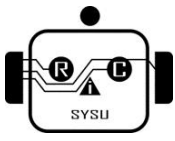
找到所有的数据竞争和控制竞争问题

- 如果是寄存器关联引起的=> **data hazard**: forward or stall to resolve them
- 如果是和**PC**有关的竞争=> **control hazard**: we’ll see next



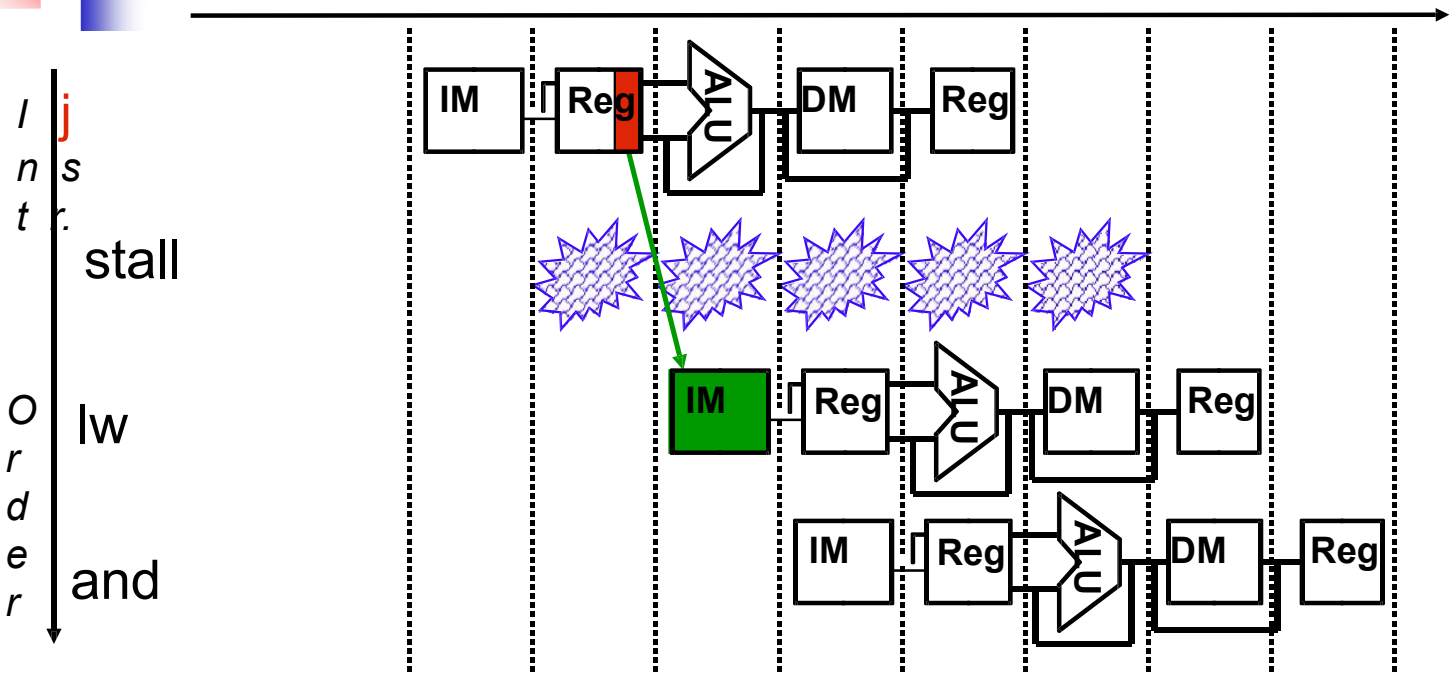
Control Hazards控制竞争

- 当指令流向不是预期的方式
  - 有条件的分支 (beq, bne)需阻塞3个周期
  - 无条件的跳转 (j)
- 可能的解决方案
  - Stall
  - 将判断提前
  - 预测
  - 延迟决定(即延迟分支)(requires compiler support)
- 控制竞争不像数据竞争那样频繁的发生，但是也没有像forwarding这样解决它的那么有效的方式



Jumps 引起一个阻塞

- Jumps 在ID阶段可以被判断出来这时只需要阻塞一个周期



- 实际情况下,只有2% 指令是jump(SPEC int instruction mix)

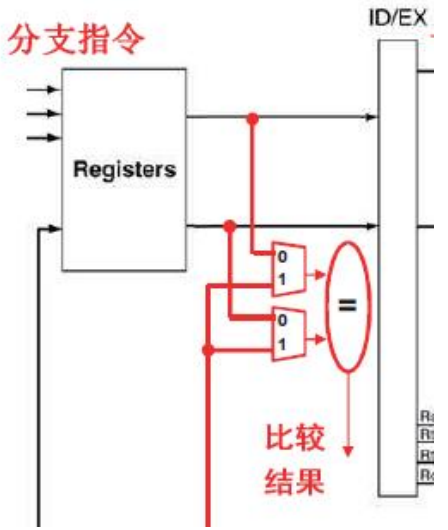
在流水线中提前进行分支决定

- 将分支判断放在**EX**阶段中
  - 可以将STALL减少为2个
  - Adds an and gate and a 2x1 mux to the EX timing path

- 增加硬件以在**ID**阶段就进行比较运算和有效地址的运算,
  - 这样可以使STALL减少为1
  - 与RF的读操作同时
  - 需要一个比较器，多路选择器等电路
  - 需要在ID阶段引入forwarding hardware 电路

将分支判断放在ID阶段中

- 在ID阶段放置比较器，尽快得到指令结果
- 分支指令结果可以提前2个clock得到
  - 分支指令后继可能被废弃的指令减少为1条
  - 当需要转移时，清除IF/ID即可

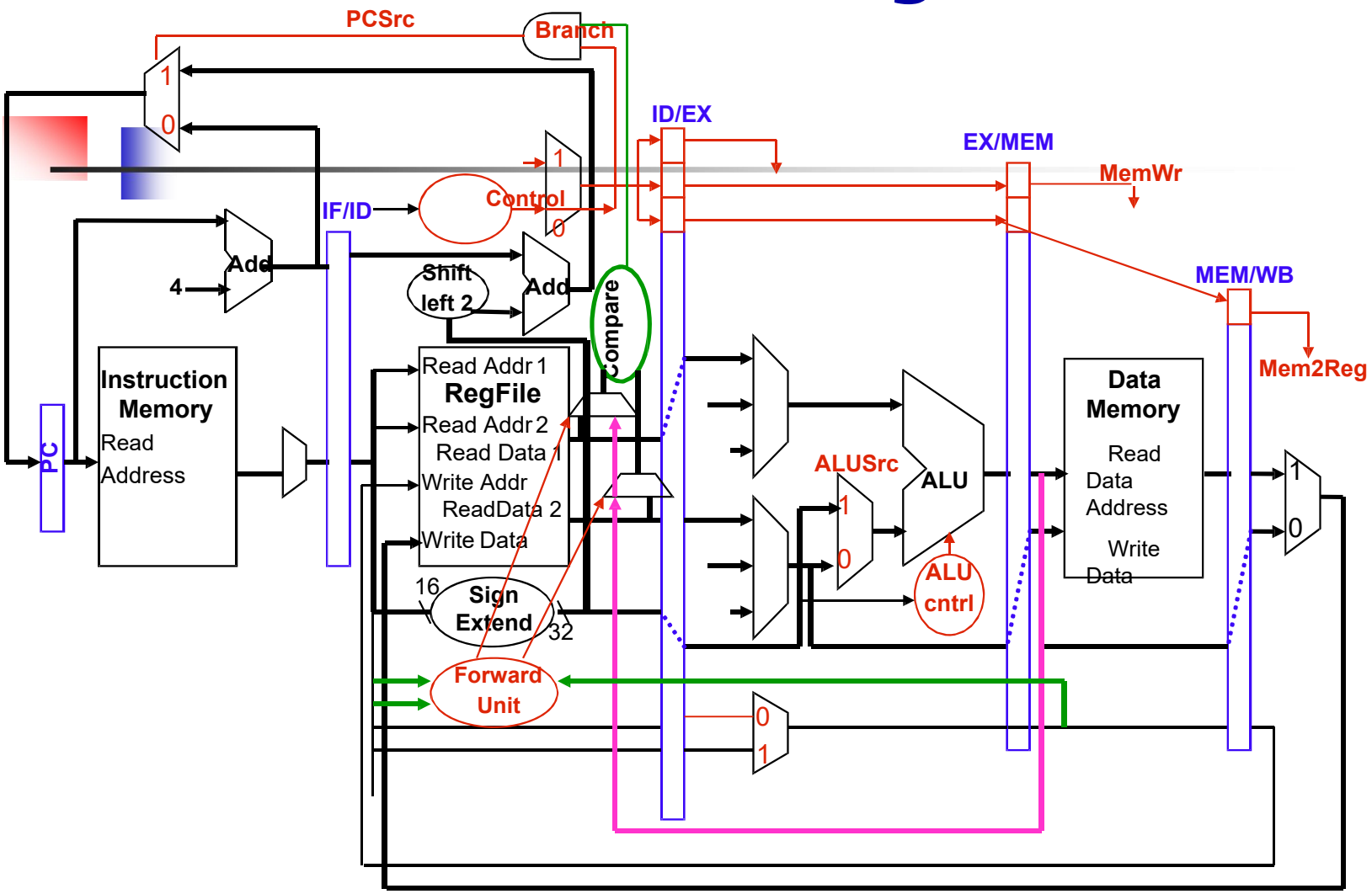


- 需要一个比较器，多路选择器等电路
- 需要在ID阶段引入forwarding hardware 电路

问题：如果比较的数据取决于前面指令的运算结果怎么办？数据相关

如果可以利用ALU运算结果可以转发

ID Stage 分支电路



在流水线中提前进行分支决定

从EX/MEM中旁路源操作数

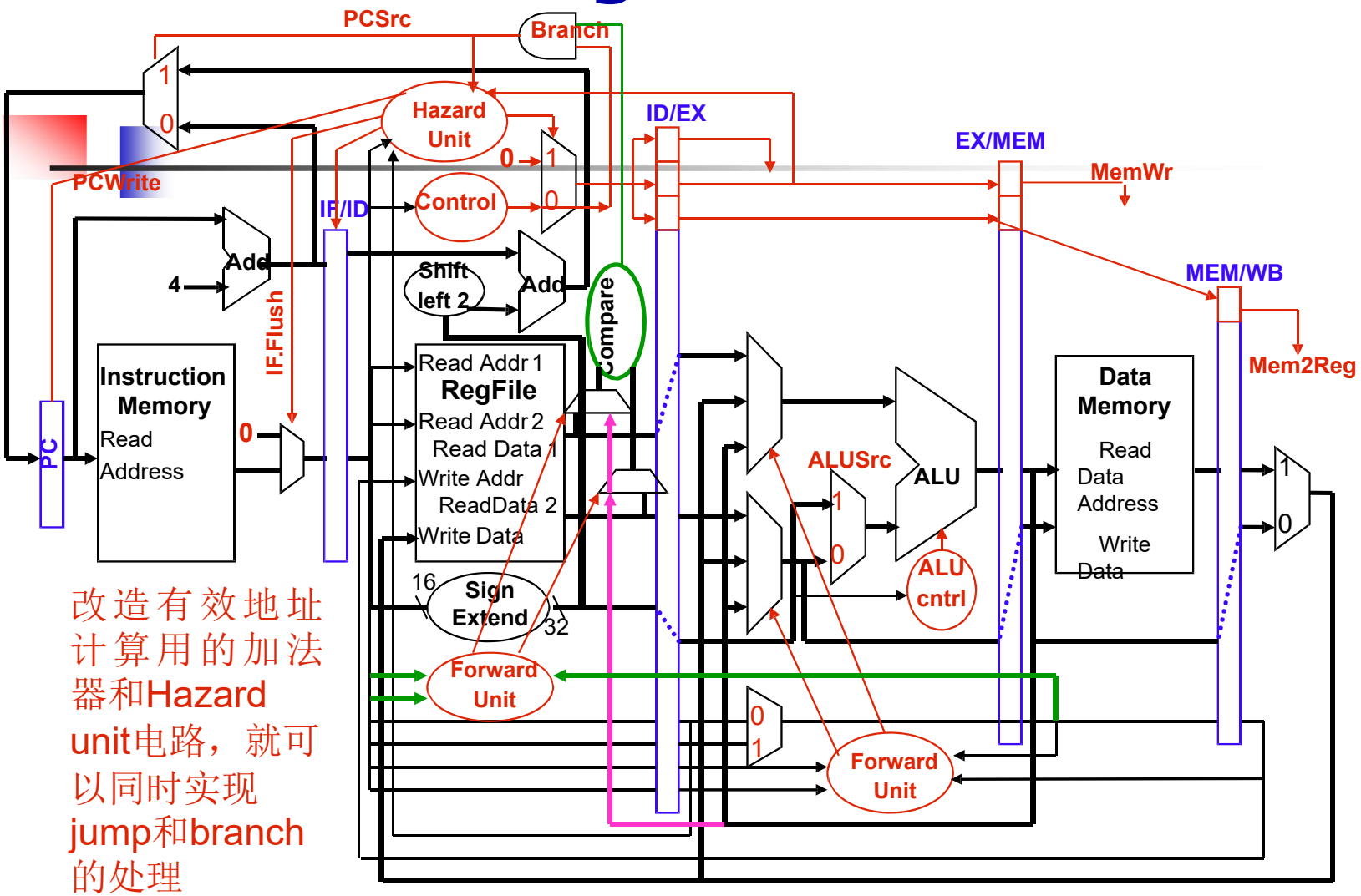
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
ForwardC = 1 if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
ForwardD = 1

从前条指令的关联寄存器结果forward到ID阶段的比较器中

- MEM/WB “forwarding” 是考虑到正常的寄存器关联引起的竞争
- 如果分支指令中参与比较的寄存器是前一条指令的输出，则必须阻塞一个周期后再判断以等待前一条指令的结果

(因为比较运算在ID阶段与前一条指令的EX阶段是同时的)

ID Stage 分支电路



改造有效地址计算用的加法器和Hazard unit电路，就可以同时实现jump和branch的处理

分支预测

- 假定一个分支的结果，不等待实际的分支判断，直接根据假定的分支结果执行流程

（1）总假设分支不发生

（2）总假设分支会发生

（3）动态预测：

分支决策移到了IF段，用于转发和阻塞的多路复用器和控制也不得不复制到解码段

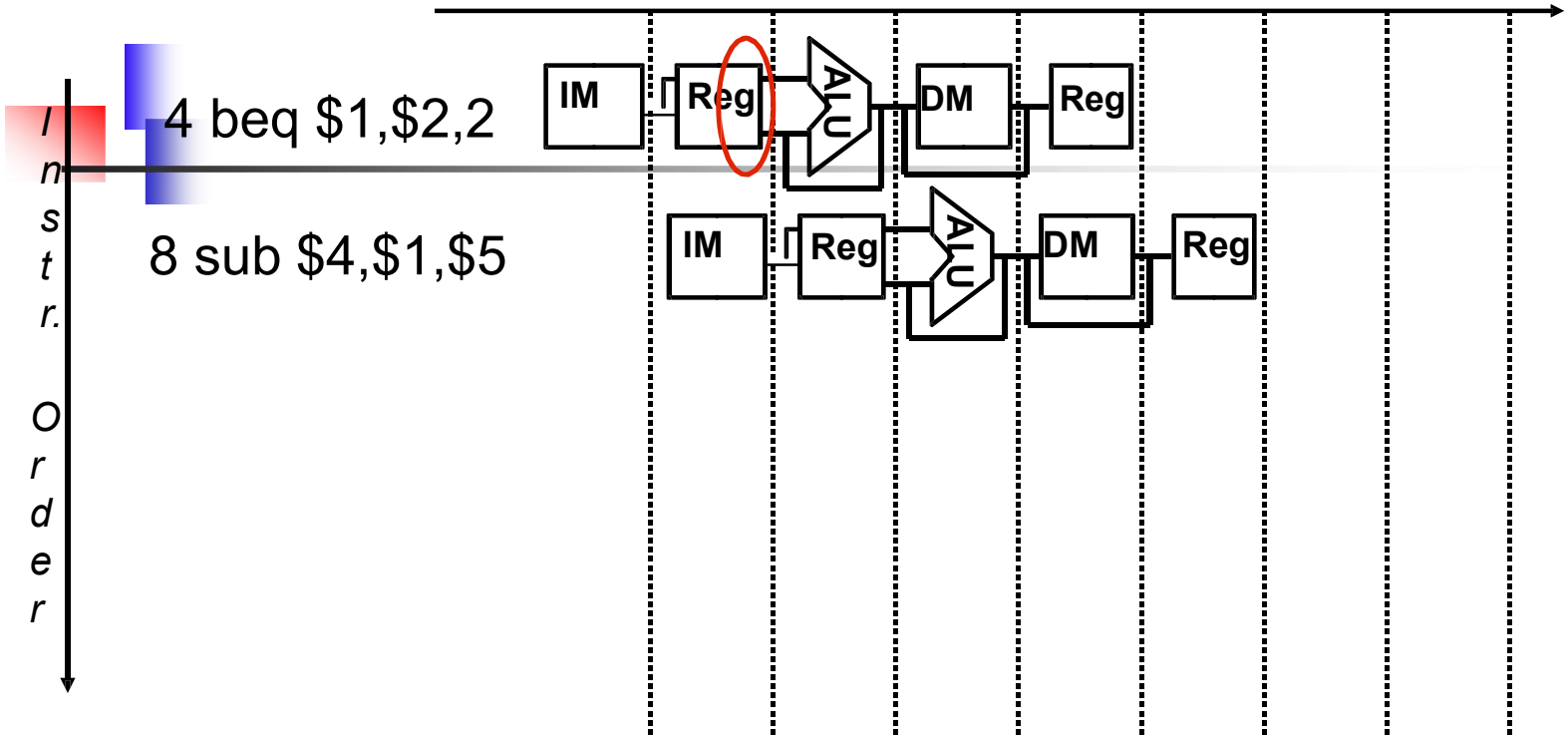
- 延迟分支

分支预测——不发生

（1）总假设分支不发生 – 遇到分支指令仍然按照正常的流程取指令,如果在流水线后期确实分支发生,则阻塞并取消已进入流水线指令

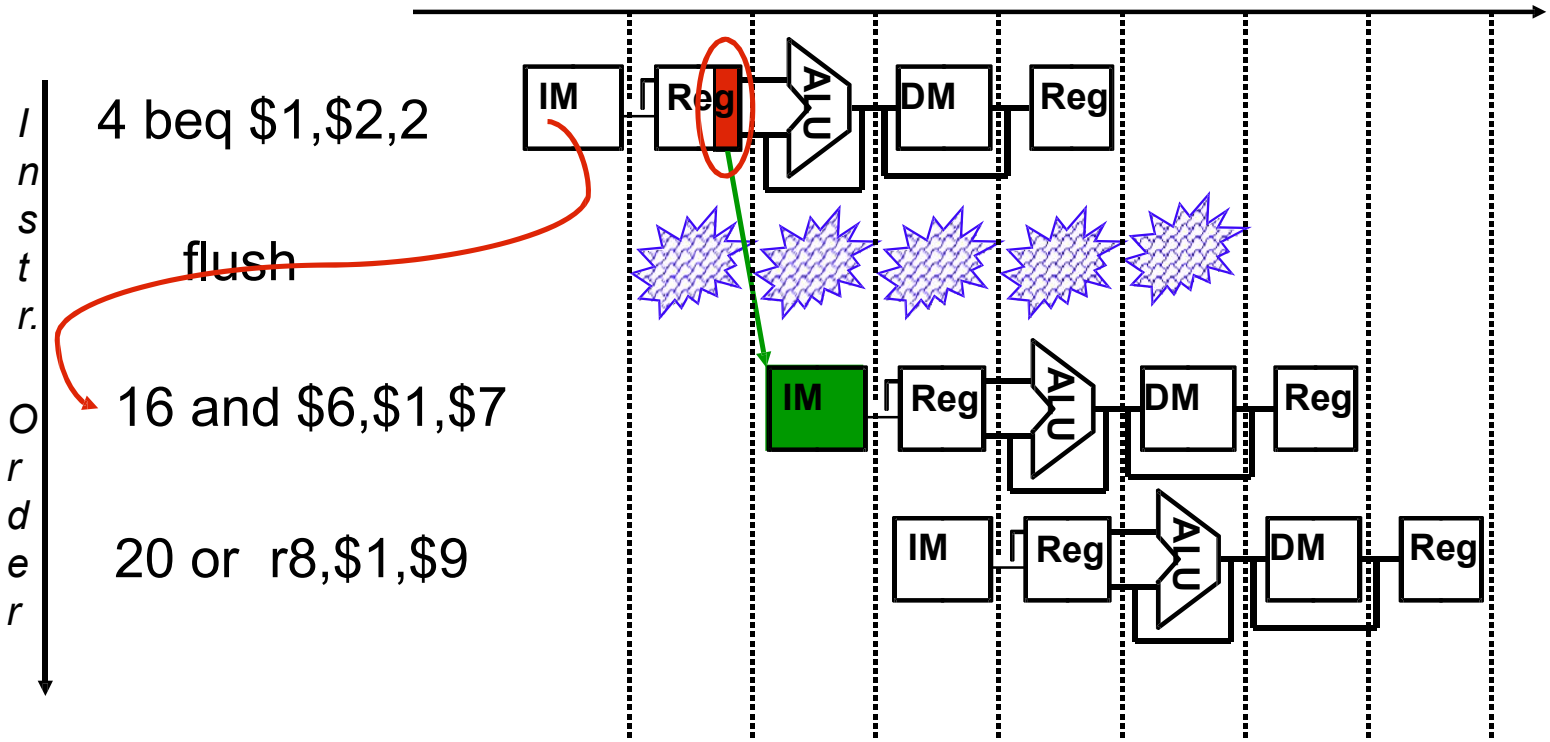
- 如果分支发生, 清除分支指令后的流水线内指令
  - IF, ID, and EX if branch logic in MEM – three stalls
  - IF if branch logic in ID – one stall
- 确保这些被清除的指令未改变机器状态– 由于机器状态是被(MemWrite or RegWrite) 控制信号在MEM和WB阶段改写的，因此这点可以保证
- 分支后继续流水线执行

预测错误时清除



- 为了实现IF阶段指令清零, 增加IF.Flush 控制线于IF/ID pipeline register,这个功能事实上在control hazard改造时已经加了.

预测错误时清除



- 为了实现IF阶段指令清零, 增加IF.Flush 控制线于IF/ID pipeline register,这个功能事实上在control hazard改造时已经加了.

分支预测——假设总发生

- 通过静态的假定一个分支判断结果的输出解决分支竞争

2 Predict taken – 总假定分支会发生

  - 发生分支引起一个周期的阻塞 (假定ID阶段的分支判断电路已经加上了)

- 这种策略损失效率

- 另外的解决方案就是再增加硬件进行动态预测

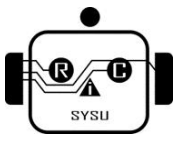
（3）Dynamic branch prediction – 利用运行时的信息进行动态预测

分支预测——动态预测

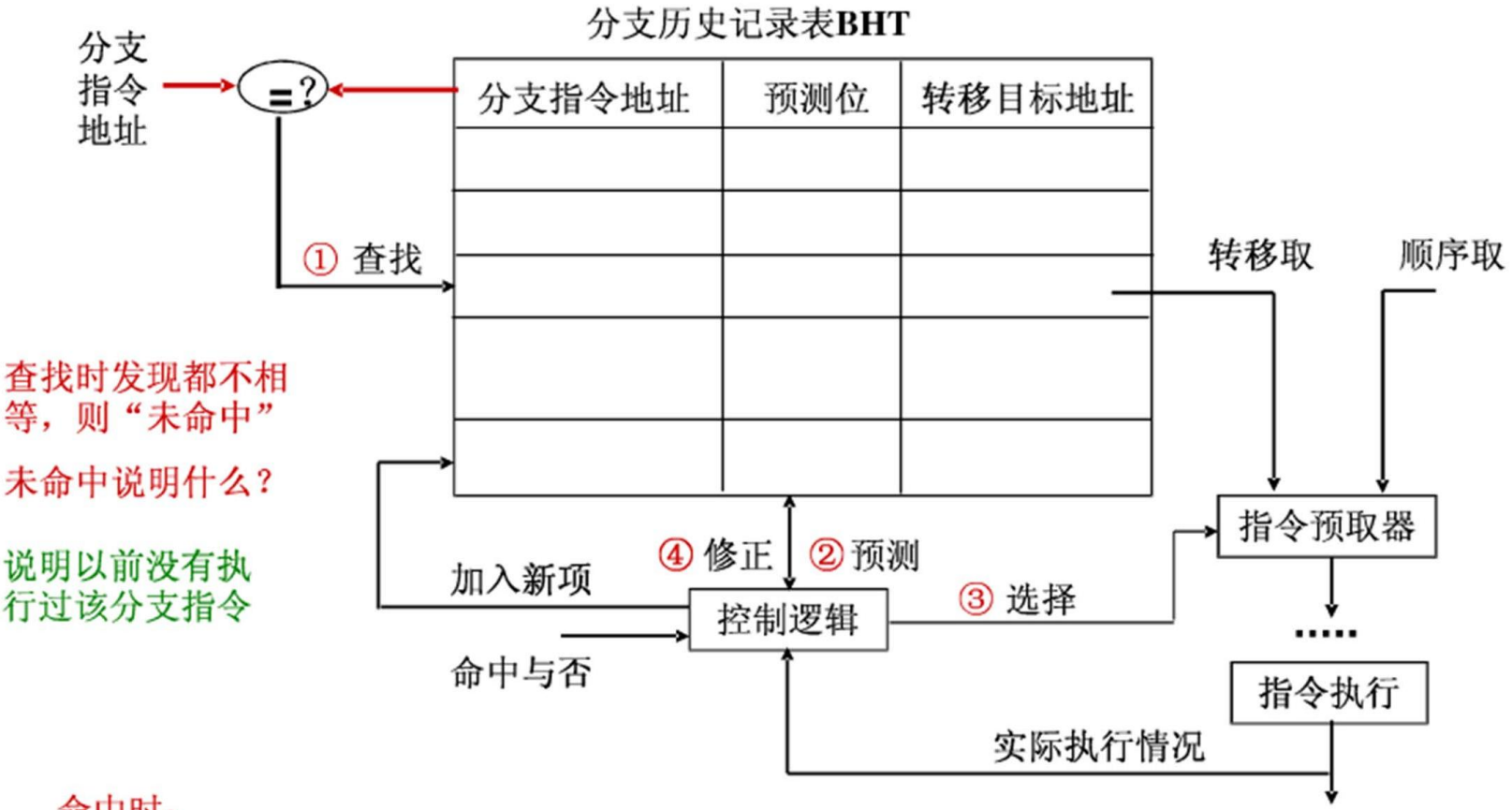
- 再增加硬件进行动态预测

3 Dynamic branch prediction – 利用运行时的信息进行动态预测

- 需要实现一个IF阶段分支预测缓存(即 branch history table (BHT)), 可以以*PC的低位地址*为索引,以1bit记录上次是否发生了转移
- *why?*



分支历史记录表BHT（或BTB、BPB）



命中时：

根据预测位，选择“转移取”还是“顺序取”

未命中时：

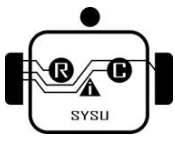
加入新项，并填入指令地址和转移目标地址、初始化预测位

分支预测——动态预测

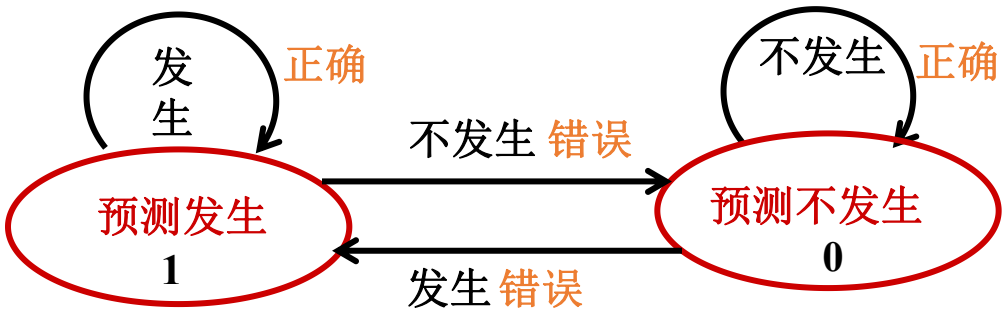
分支历史记录表**BHT(Branch History Table）**

- 每个表项有分支地址的地位索引，故在**IF**段可以取到预测位：
  - 低位地址相同的分支指令共享一个表项
    - 可能取得时其它指令的预测位，会不会有问题？
  - 仅用于预测，不影响执行结果

现在几乎所有的处理器都采用动态预测。



一位预测状态图



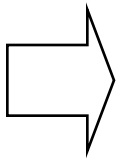

```
Loop:    g = g +A[i];
         i = i+ j;
         if (i != h) go to Loop:
```

- 指令预取时，按照预测读取相应分支的指令
  - ‘1’：选择“转移取”
  - ‘0’：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

```
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:  … …
```

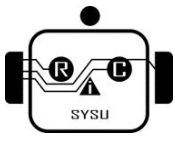
外循环中的分支指令共执行N+1次，
内循环中的分支指令共执行N×(N+1)次。

N=10, 分别90.9%和82.7%

N=100, 分别99%和98%

预测位初始为0，外循环只有最后1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，内循环有1+2×(N–1)次预测错误。

N越大准确率越高！



延迟分支

- 首先还是要将分支的判断电路放在ID阶段
- 延迟分支技术使得分支指令的含义与原来不同,该机制规定分支指令后面紧跟的指令必须被执行，无论条件判断结果如何
  - MIPS支持该机制, 于是MIPS 编译器将一条不受分支条件影响的指令放在分支指令之后，这样可以隐藏该指令的执行时间
- 处理器流水线级数增加以后需要更多的延迟分支槽数目.
  - 延迟分支毕竟不如动态分支灵活目前已经不那么广泛使用,但是这是个简单的解决方案

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操作填充

举例：如何对以下程序段进行分支延迟调度？

（假定时间片C=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C=1，则可以：

```
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，降低了分支延迟损失

Control Hazard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)或满足，

即：继续执行分支指令的后续指令

  - 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：采用分支预测方式时，流水线控制必须确保错误预测指令的执行结果不能生

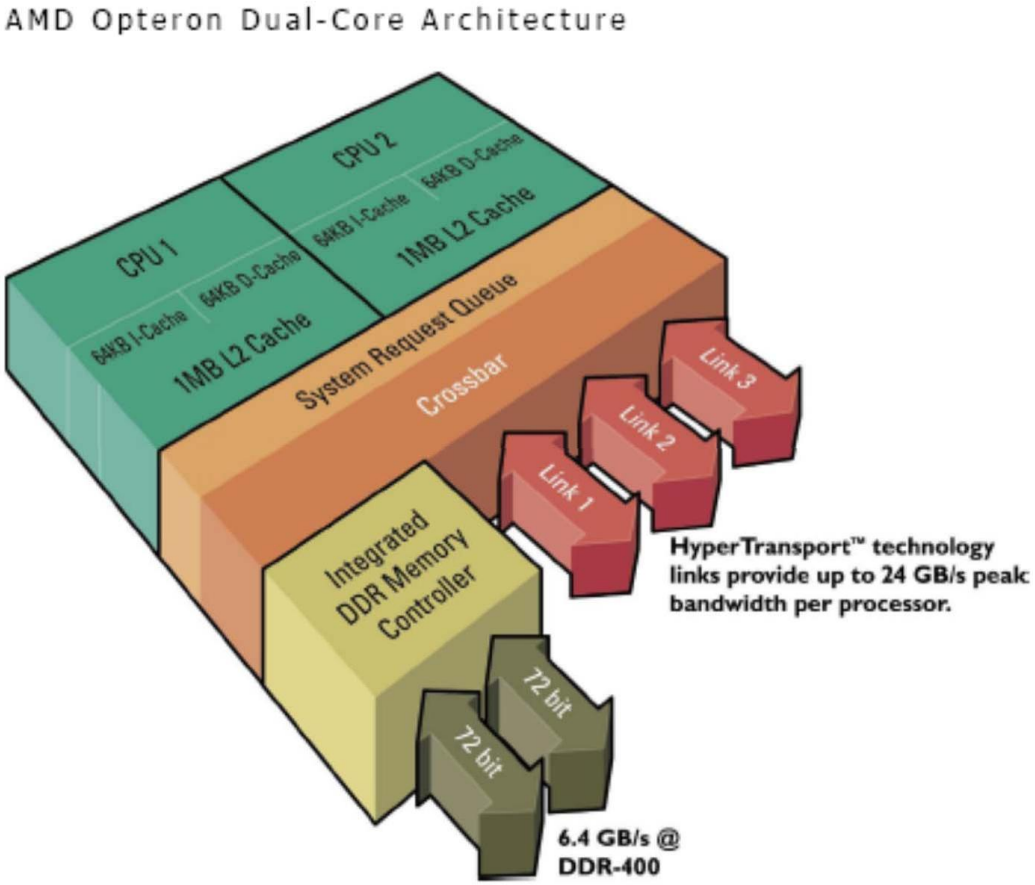
效，而且要能从正确的分支地址处重新启动流水线工作

- 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面执行，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

What is next?

- Multi-Core
  - AMD X2 (2004)
  - Intel Montecito(late 2005)



Multi-core mapps multi-thread applications to real parallelism

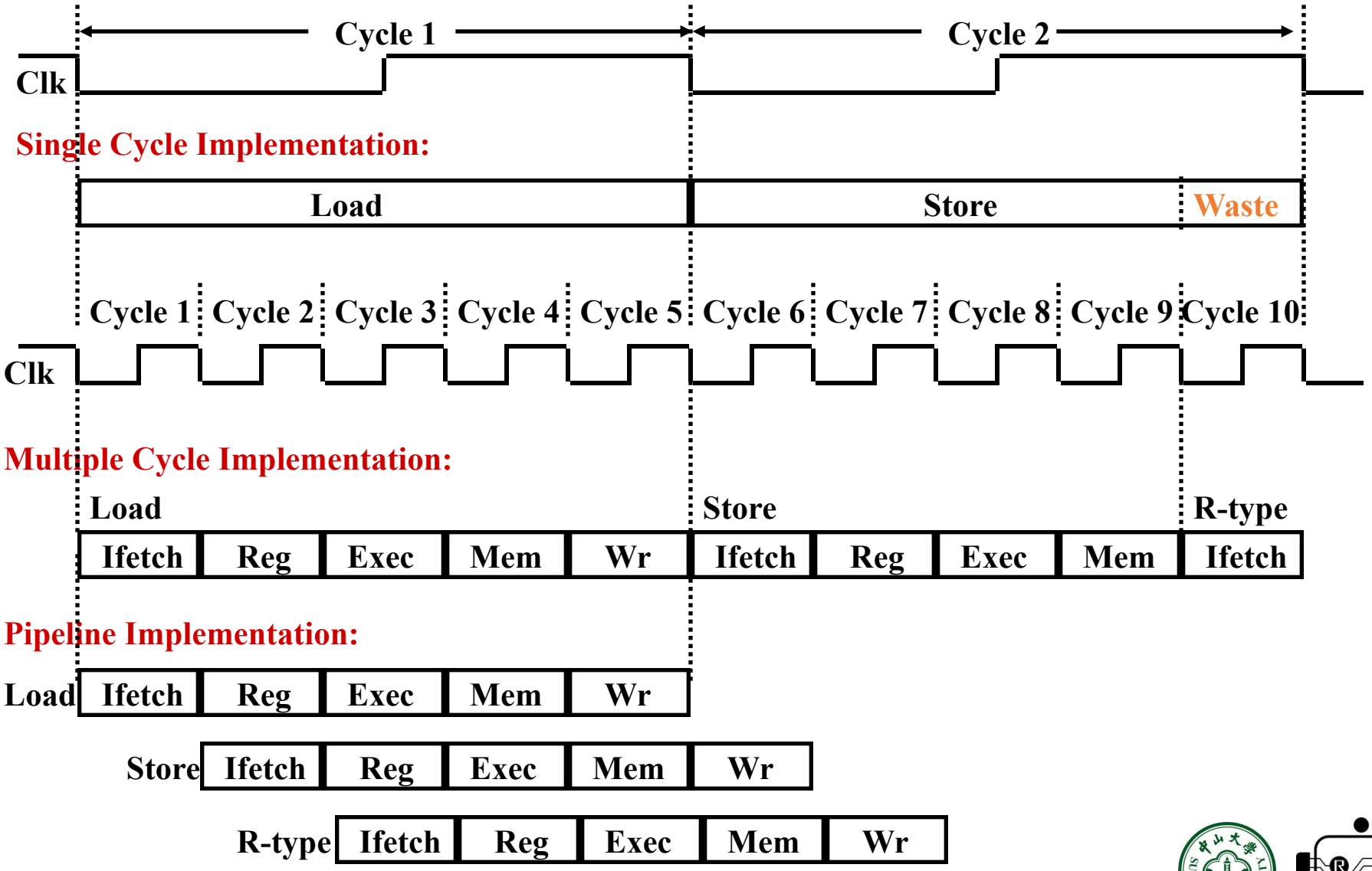
- **Already available in RISC platform;**
- **IBM Power5, Sum Ulta SPARC IV, HP PA-RISC 8800**

本讲小结

- 有以下两种指令级并行(ILP)技术（即：高性能流水线形式）
  - 超流水线：更多的流水线级数
  - 多发射流水线：同时发射多个指令，有多条流水线同时进行
    - 静态多发射（VLIW处理器+编译器静态调度）
    - 动态多发射（超标量处理器+动态流水线调度）

本章总结1

单周期, 多周期 和 流水线比较

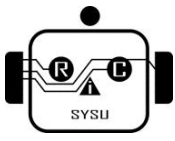


本章总结2

- 指令流水线的设计
  - 将每条指令的执行规整化为若干个同样的流水阶段
  - 每个流水阶段的执行时间一样，都等于一个时钟
  - 理想情况下，每个时钟有一条指令进入流水线，也有一条指令执行结束
  - 每个流水段中的部件都是组合逻辑加寄存器，组合逻辑中产生的结果在时钟到来时被存储到寄存器（如：程序计数器、条件码寄存器、流水段寄存器）
  - 每两个相邻流水段之间的流水线寄存器，用以记录所有在后面阶段要用到的各种信息，有哪些呢？
    - 控制信号、指令的代码、参加运算的操作数、指令运算结果、指令异常信息、寄存器读口地址、寄存器写口地址、存储器地址、新的PC值等。
  - 指令译码得到的控制信号通过流水线寄存器传送到后面各个流水段中

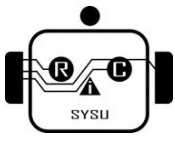
本章总结3

- 指令流水线的局限性
  - 并不是每个流水段都一样长
  - 随着流水线深度的增加，流水线寄存器的额外开销比例也增大
  - 指令在资源冲突、数据相关或控制相关时会发生流水线冒险
- 指令流水线的执行效率
  - 吞吐率：比非流水线方式下大大提高
  - 指令执行时间：相对于非流水线方式，一条指令的执行时间延长了
- 提高流水线指令效率的高级流水线技术
  - 超流水线：级数更多的流水线
  - 多发射流水线：同时发射多条指令的流水线
    - 静态多发射：VLIW结构、编译器静态推测
    - 动态多发射：超标量结构、硬件动态推测调度



本章总结4

- 结构冒险（资源冲突）：多条指令同时使用同一个功能部件
  - 规定每个功能部件在一条指令中只能被用一次
  - 规定每个功能部件只能在某个特定的阶段被用
  - 指令存储器(Code Cache)和数据存储器(Data Cache)分开
- 数据冒险（数据相关）：前面指令的结果是后面指令的操作数
  - 软件阻塞：（如：编译器）在后面的数据相关指令前插入nop指令
  - 硬件阻塞：在后面数据相关指令的特定流水段插入“气泡”以“阻塞”指令继续执行，直到取得所需数据为止
  - “转发”（旁路）：把前面指令执行过程中得到的数据直接传送到后面指令。
  - 对于取数后直接使用的情况（如：Load指令取出的数据是随后的运算指令的操作数），则采用“阻塞加转发”的方式解决数据冒险



本章总结5

- 控制冒险（控制相关）：返回指令、分支指令等可能改变顺序增量的PC值，由于获取转移目标地址的时间较长，使得在目标地址产生前已经有指令被取到流水线中，如果已经取出执行的指令不是正确的指令，则发生控制冒险。
  - 软件阻塞：（如：编译器）在控制相关指令后面插入nop指令
  - 硬件阻塞：在控制相关指令后面的指令被取出前插入“气泡”，使流水线停顿若干时钟，直到控制相关指令得到正确的PC值为止
  - 采用“分支预测”技术。简单（静态）地预测每次分支结果都一样，或根据分支指令执行历史进行动态预测，动态预测能达到90%以上的成功率
  - 采用延迟分支技术。将前面一条与分支指令无关的指令放到分支指令后面执行，这样，流水线不会发生阻塞现象。这种对指令顺序进行调整的工作在程序编译阶段完成

联系方式

- Acknowledgements:
- This slides contains materials from following lectures:
  - Computer Architecture (ETH，NUDT，USTC)

- Research Area：
  - 计算机视觉与机器人应用计算加速，
  - 人工智能和深度学习芯片及智能计算机

- Contact：
  - 中山大学计算机学院
  - 管理学院D101（图书馆右侧）
  - 机器人与智能计算实验室
  - cheng83@mail.sysu.edu.cn