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

流水线概述

- 如果将指令的每个阶段看成相应的流水段，则指令的执行过程构成了一条指令流水线。

- **一条指令流水线由如下5个流水段组成**：
  - **取指令(IF)**：从存储器取指令；
  - **指令译码(ID)**: 产生指令执行所需的控制信号；
  - **取操作数(OF)**: 读取操作数；
  - **执行(EX)**: 对操作数完成指定操作；
  - **写回(WB)**: 将结果写回。

流水线概述

- 当后一条指令的第i步与前一条指令的第i+1步同时进行，可以使**一串指令总的完成时间大为缩短**。

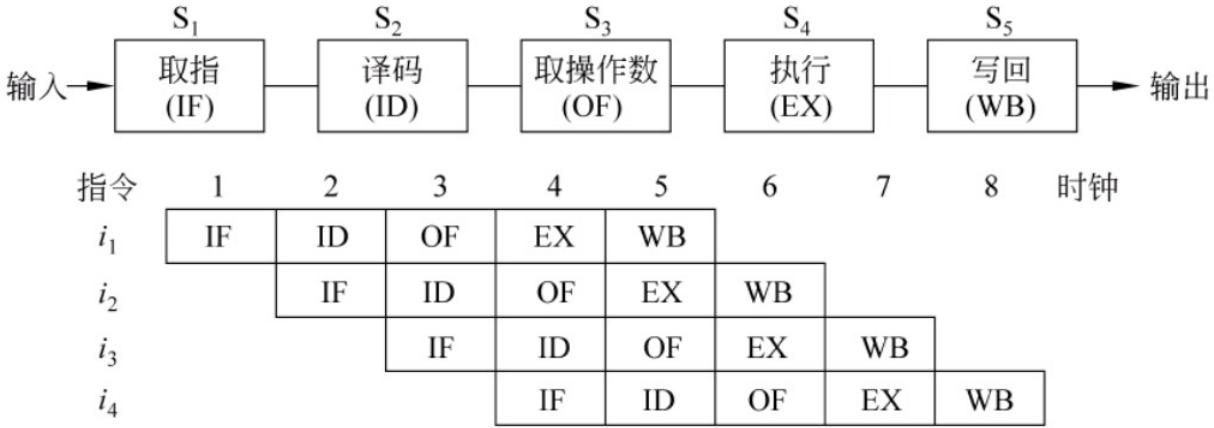


图 6.1　一个 5 段指令流水线

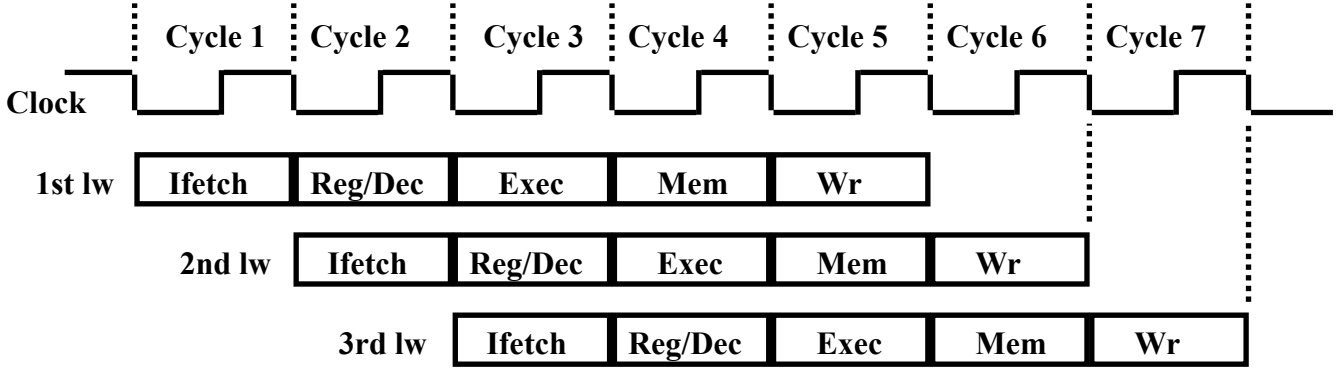
- 理想情况下，每个时钟都有一条指令进入流水线，每个时钟周期都有一条指令完成，每条指令的时钟周期数（即CPI）都为1。

回顾：Load指令的5个阶段

- **Ifetch (取指)**：取指令并计算PC+4 (用到哪些部件？) **指令存储器、Adder**
- **Reg/Dec (取数和译码)**：取数同时译码 (用到哪些部件？) **寄存器堆读口、指令译码器**
- **Exec (执行)**：计算内存单元地址 (用到哪些部件？) **扩展器、ALU**
- **Mem (读存储器)**：从数据存储器中读 (用到哪些部件？) **数据存储器**
- **Wr(写寄存器)**: 将数据写到寄存器中 (用到哪些部件？) **寄存器堆写口**

这里寄存器堆的读口和写口可看成两个不同的部件。

Load指令的流水线



- **每个周期有五个功能部件同时在工作**
- 后面指令在前面完成取指后马上开始
- 每个load指令仍然需要五个周期完成
- **但是，吞吐率(throughput)提高许多，理想情况下，有：**
  - 每个周期有一条指令进入流水线
  - 每个周期都有一条指令完成
  - 每条指令的有效周期(CPI)为1

单周期指令模型与流水模型的性能比较

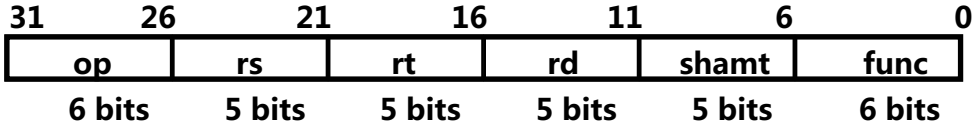
- **假定以下每步操作所花时间为：**
  - 取指：2ns
  - 寄存器读：1ns
  - ALU操作：2ns
  - 存储器读：2ns
  - 寄存器写：1ns

  Load指令执行时间总计为：8ns
  (假定控制单元、PC访问、信号传递等没有延迟)
- **单周期模型**
  - 每条指令在一个时钟周期内完成
  - 时钟周期等于最长的lw指令的执行时间，即：8ns
  - 串行执行时，N条指令的执行时间为：8Nns
- **流水线性能**
  - 时钟周期等于最长阶段所花时间为：2ns
  - 每条指令的执行时间为： 2nsx5=10ns
  - N条指令的执行时间为：(4+N)x2ns
  - **在N很大时，比串行方式提高约 4 倍**
  - 若各阶段操作均衡(例如，各阶段都是2ns)，则提高倍数为5倍。

流水线方式下，单条指令执行时间不能缩短，但能大大提高指令吞吐率！

适合流水线的指令集特征

- **具有什么特征的指令集有利于流水线执行呢？**
  - **长度尽量一致**，有利于简化取指令和指令译码操作
    - MIPS指令32位，下址计算方便: PC+4
    - X86指令从1字节到17字节不等，使取指部件极其复杂
  - **格式少，且源寄存器位置相同**，有利于在指令未知时就可取操作数
    - MIPS指令的rs和rt位置一定，在指令译码时就可读rs和rt的值



若位置随指令不同而不同，则需先确定指令类型才能取寄存器编号

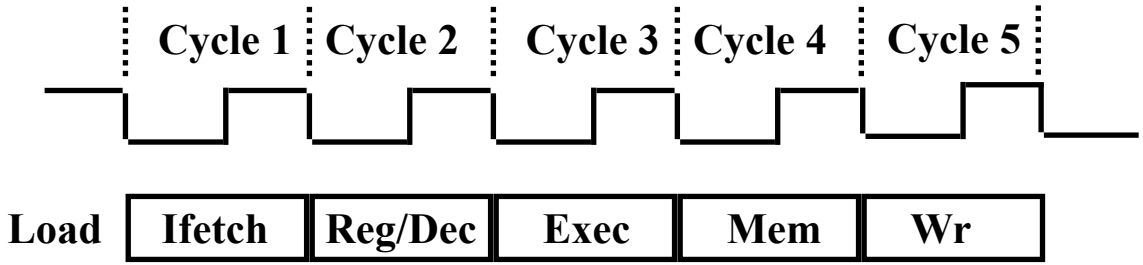
  - **load / Store指令才能访问存储器**，有利于减少操作步骤，规整流水线
    - lw/sw指令的地址计算和运算指令的执行步骤规整在同一个周期
    - X86运算类指令操作数可为内存数据，需计算地址、访存、执行
  - **内存中”对齐”存放**，有利于减少访存次数和流水线的规整

总之，规整、简单和一致等特性有利于指令的流水线执行

指令流水线

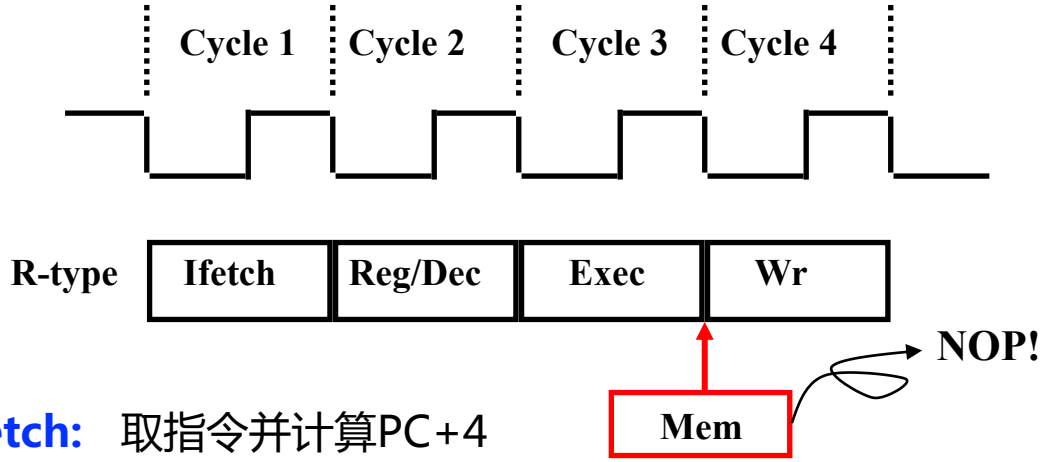
- 流水线概述
- **流水线处理器的实现**
- 流水线冒险及其处理
- 高级流水线技术

Load指令的流水线



- **Ifetch**：取指令并计算PC+4
- **Reg/Dec**： 从寄存器取数，同时指令在译码器进行译码
- **Exec**：16位立即数符号扩展后与寄存器值相加，计算主存地址
- **Mem**：从存储器中读数据；
- **Wr**：将数据写入寄存器。

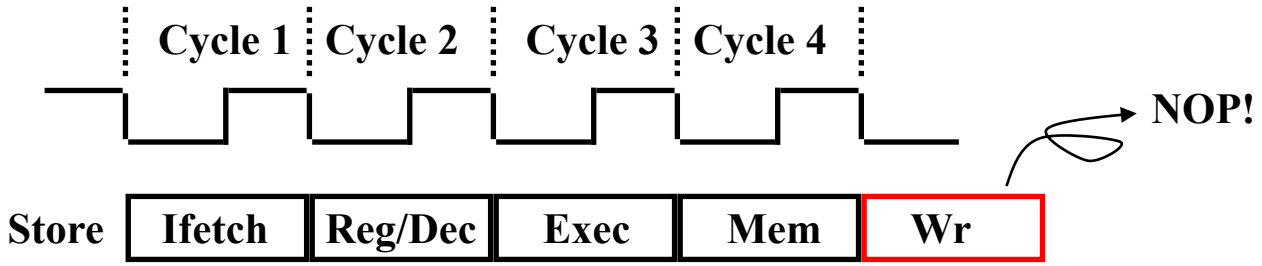
R型指令功能段划分



- **Ifetch:** 取指令并计算PC+4
- **Reg/Dec:** 从寄存器取数，同时指令在译码器进行译码
- **Exec:** 在ALU中对操作数进行计算
- **Wr:** ALU计算的结果写到寄存器
- **Mem: 在Write前加一个空的Mem段，使流水线更规整！**

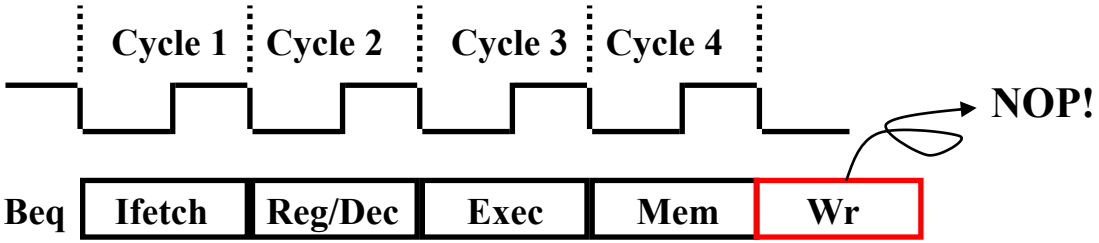
I型指令功能段划分，与R型指令相同。

Store指令功能段划分



- **Ifetch**：取指令并计算PC+4
- **Reg/Dec**： 从寄存器取数，同时指令在译码器进行译码
- **Exec**：16位立即数符号扩展后与寄存器值相加，计算主存地址
- **Mem**：将寄存器读出的数据写到主存
- **Wr：加一个空的写阶段，使流水线更规整！**

Beq指令功能段划分



- **Ifetch:** 取指令并计算PC+4
- **Reg/Dec:** 从寄存器取数，同时指令在译码器进行译码
- **Exec:** 执行阶段
  - ALU中比较两个寄存器的大小（做减法）
  - Adder中计算转移地址
- **Mem:** 如果比较相等, 则转移目标地址写到PC
- **Wr: 加一个空写阶段，使流水线更规整！**

按照上述方式，把所有指令都按照最复杂的“load”指令所需的五个阶段来划分，不需要的阶段加一个“NOP”操作

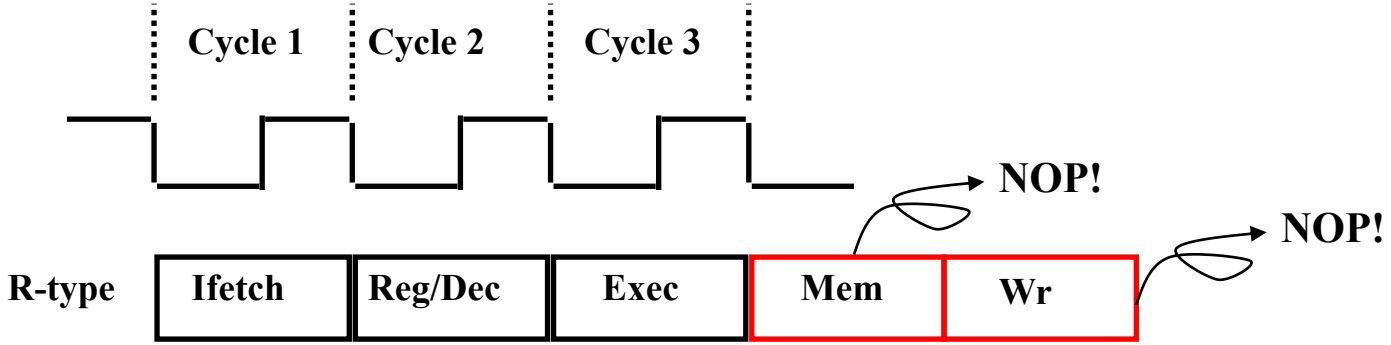
与多周期通路有什么不同？

多周期通路中，在Reg/Dec阶段投机进行了转移地址的计算！可以减少Branch指令的时钟数

为什么流水线中不进行“投机”计算？

因为流水线中所有指令的执行阶段一样多，Branch指令无需节省时钟，因为有比它更复杂的指令。

J指令功能段划分



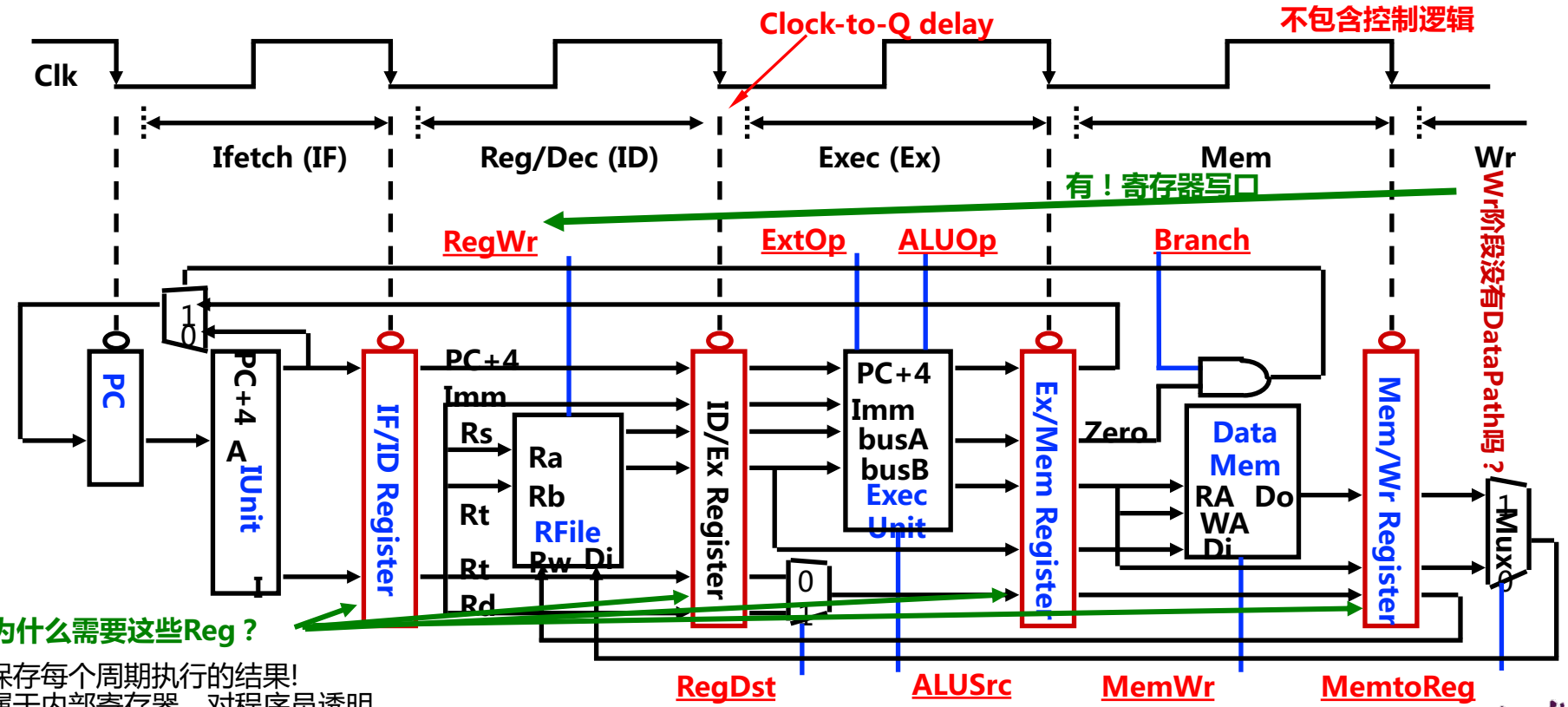
- **Ifetch**：取指令并计算PC+4
- **Reg/Dec**： 从寄存器取数，同时指令在译码器进行译码
- **Exec**：将目标地址送PC
- **Mem：加一个空的Mem阶段，使流水线更规整！**
- **Wr：加一个空的写阶段，使流水线更规整！**

插入“空”段时：

1. 每个功能部件每条指令只能用一次（如寄存器写口不能用两次或以上）；

2. 每个功能部件必须在相同的阶段被使用（如寄存器写口总是在第5阶段被使用）。

5段流水线数据通路



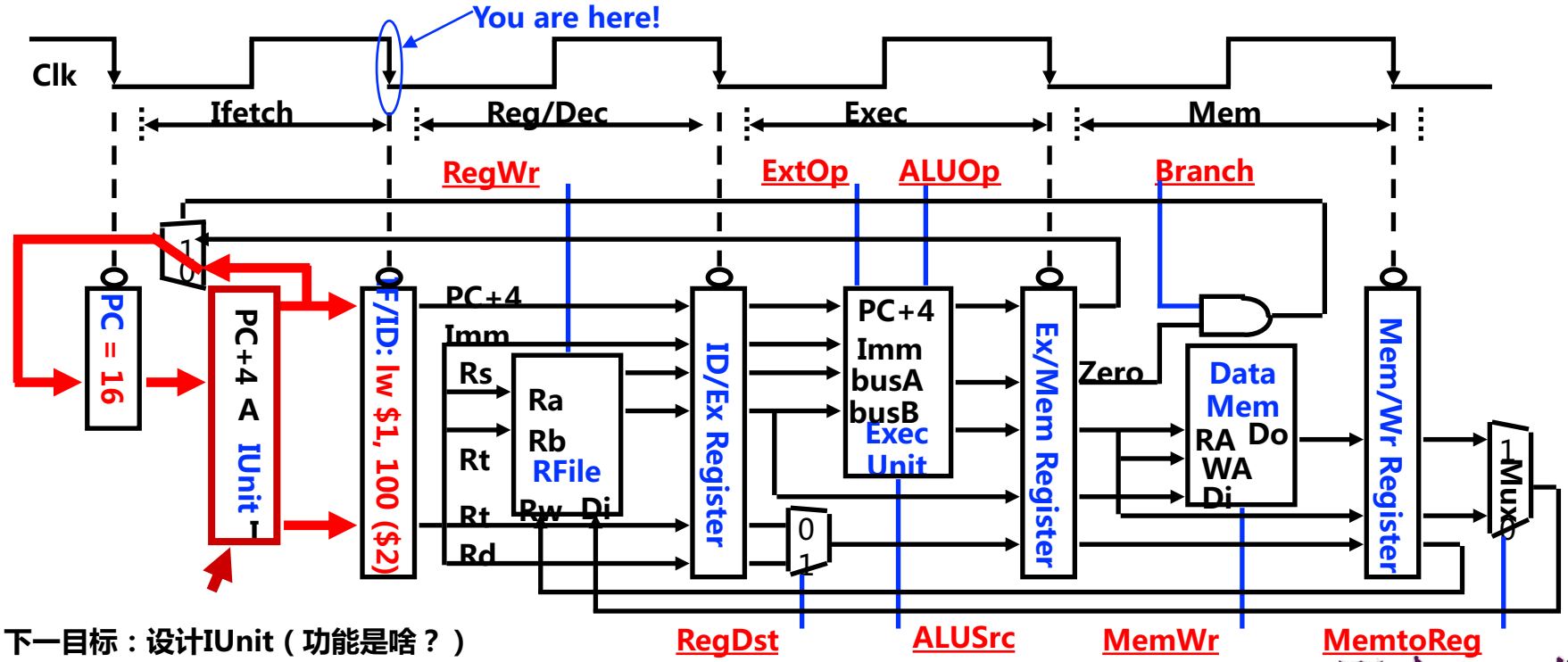
为什么需要这些Reg？

保存每个周期执行的结果!
属于内部寄存器，对程序员透明，
不需作为现场保存。

下面看每条指令在流水线通路中的执行过程

取指令(Ifetch)阶段

- 第12单元指令: lw $1, 0x100($2)　　功能：$1 <- Mem [($2) + 0x100]



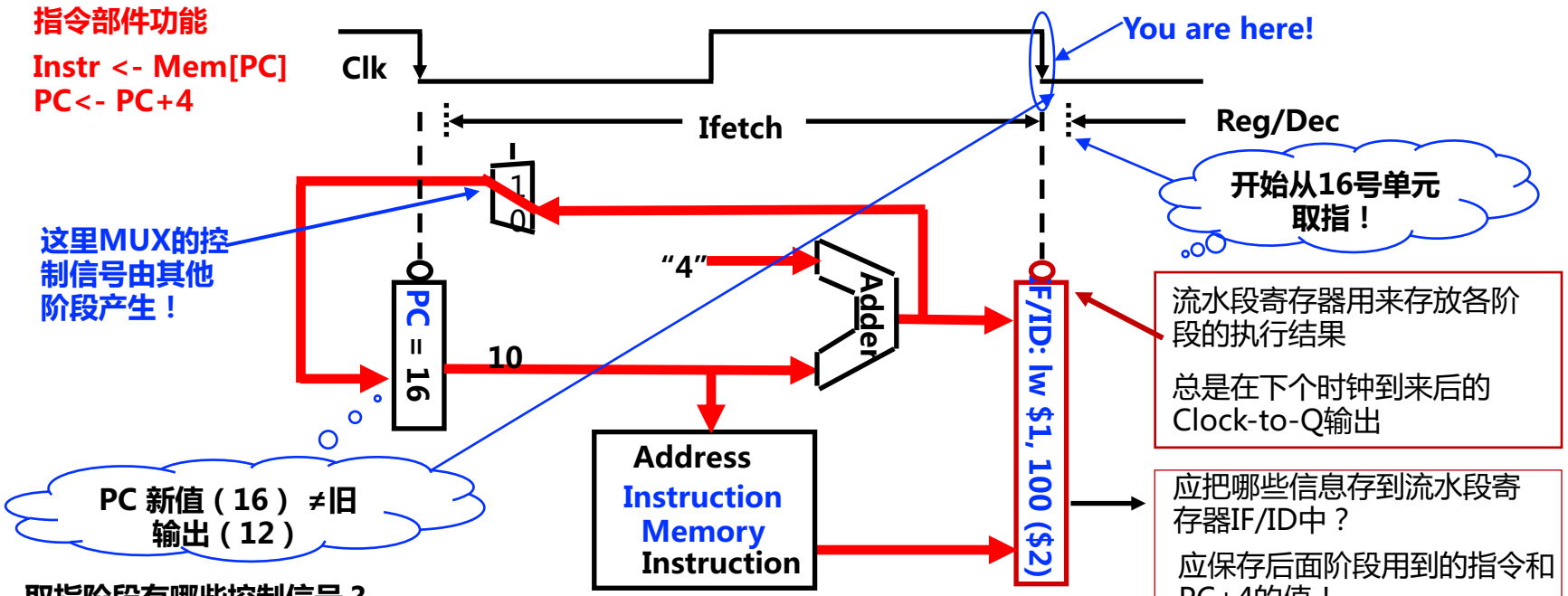
下一目标：设计IUnit（功能是啥？）

MIPS指令的地址可能是12吗？可以！

先猜一下IUnit中有哪些功能部件？

指令部件IUnit的设计

- **第12单元指令:** lw $1, 0x100($2)　　**随后的指令在16号单元中！**



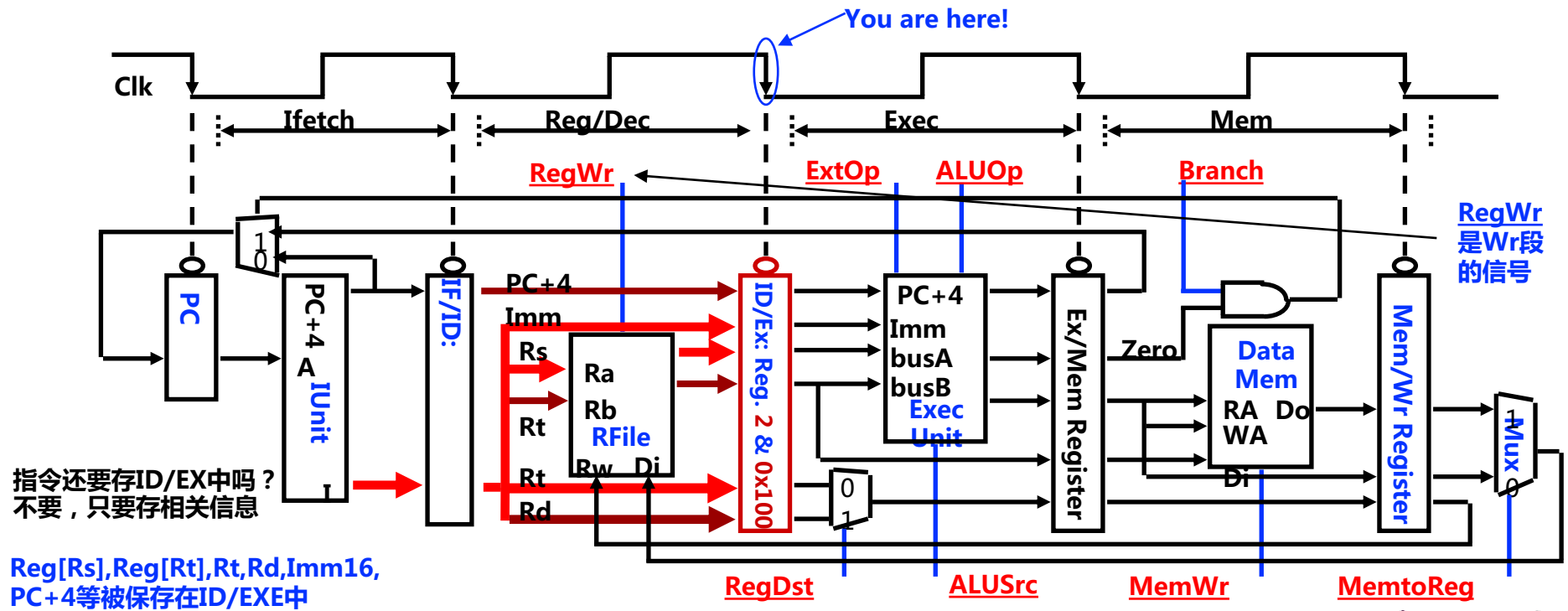
取指阶段有哪些控制信号？

不需控制信号，因为每条指令执行功能一样，是确定的，无需根据指令的不同来控制执行不同的操作！

指令在随后阶段被送出译码！
PC+4用来计算转移目标地址

译码/取数(Reg/Dec)阶段

- **Location 12:** lw $1, 0x100($2)　　**功能：**$1 <- Mem[($2) + 0x100]

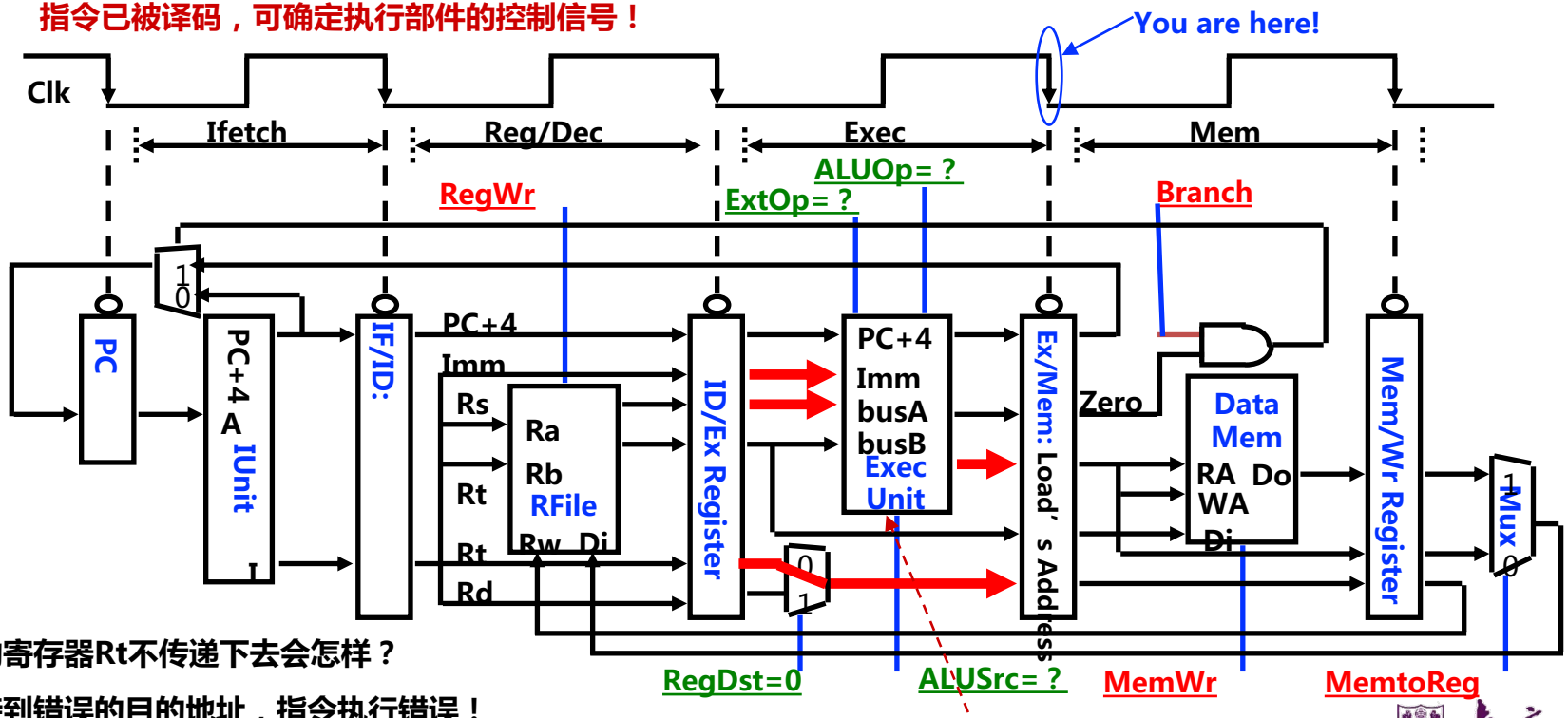


该阶段有哪些控制信号？ 没有！因是所有指令的公共操作，故无控制信号！

执行(Exec)阶段：Load指令的地址计算阶段

- **Location 12:** lw $1, 0x100($2)　　**功能：**$1 <- Mem[($2) + 0x100]

指令已被译码，可确定执行部件的控制信号！



目的寄存器Rt不传递下去会怎样？

连接到错误的目的地址，指令执行错误！

下一目标：设计执行部件(Exec Unit)　猜有哪些部件？

执行部件(Exec Unit)的设计

执行部件功能？

- 计算内存地址
- 计算转移目标地址
- 一般ALU运算

Load指令的各控制信号取值？

RegDst=0, ALUSrc=1
ALUop=addu, Extop=1

Store指令呢？

RegDst=x, ALUSrc=1
ALUop=addu, Extop=1

Branch指令呢？

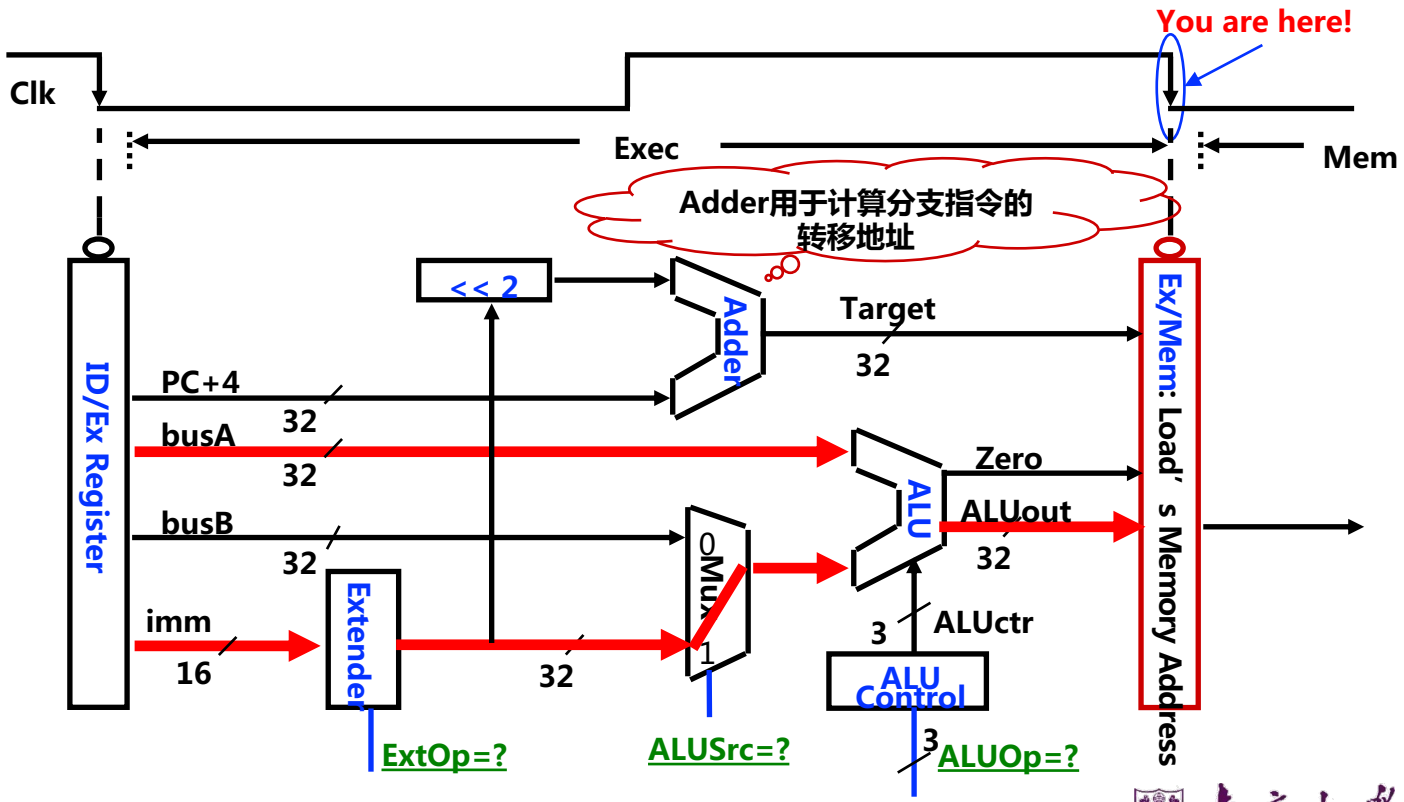
RegDst=x, ALUSrc=0
ALUop=subu, Extop=1

Ori指令呢？

RegDst=0, ALUSrc=1
ALUop=or, Extop=0

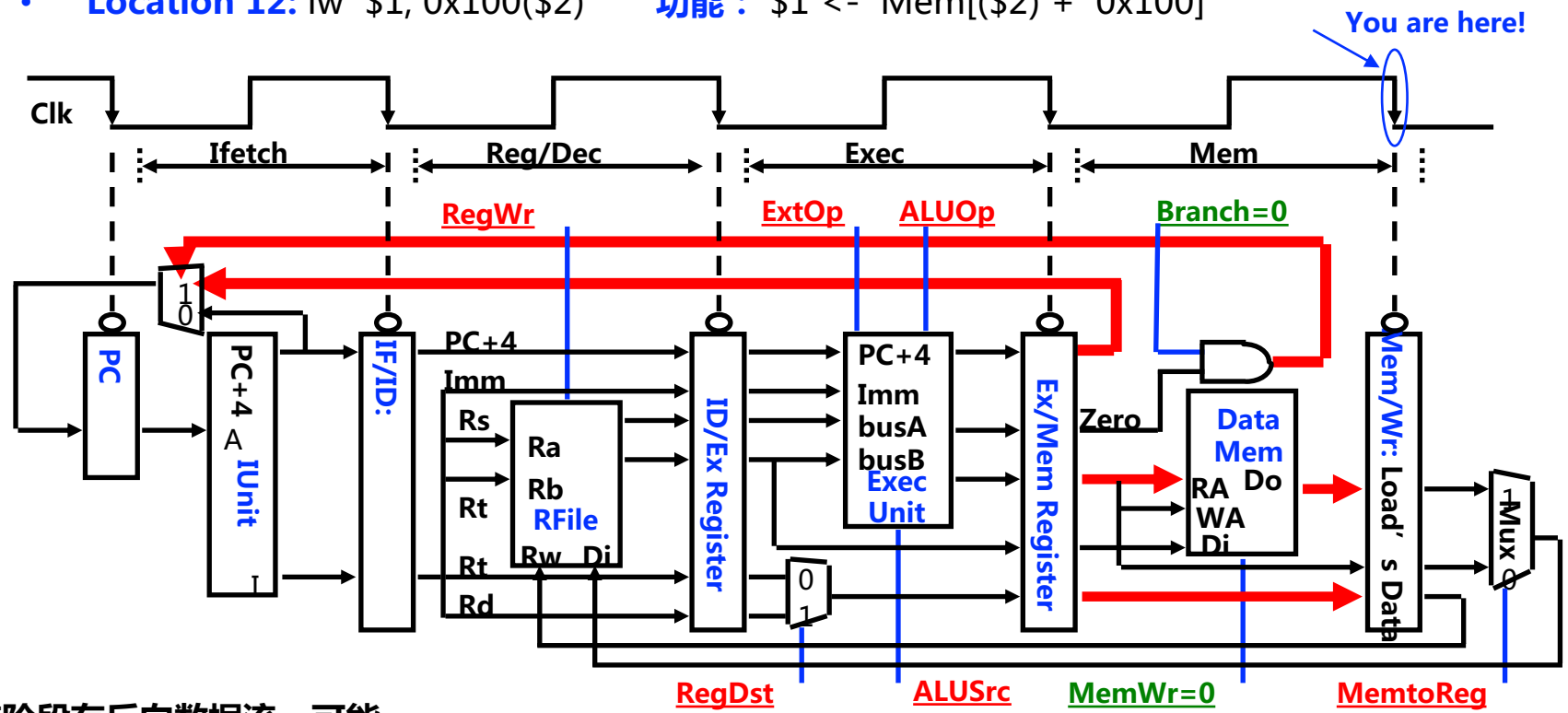
R型指令呢？

RegDst=1, ALUSrc=0
ALUop= ‘func’, Extop=x



Mem阶段：Load指令的存储器读(Mem)周期

- **Location 12:** lw $1, 0x100($2)　　**功能：** $1 <- Mem[($2) + 0x100]

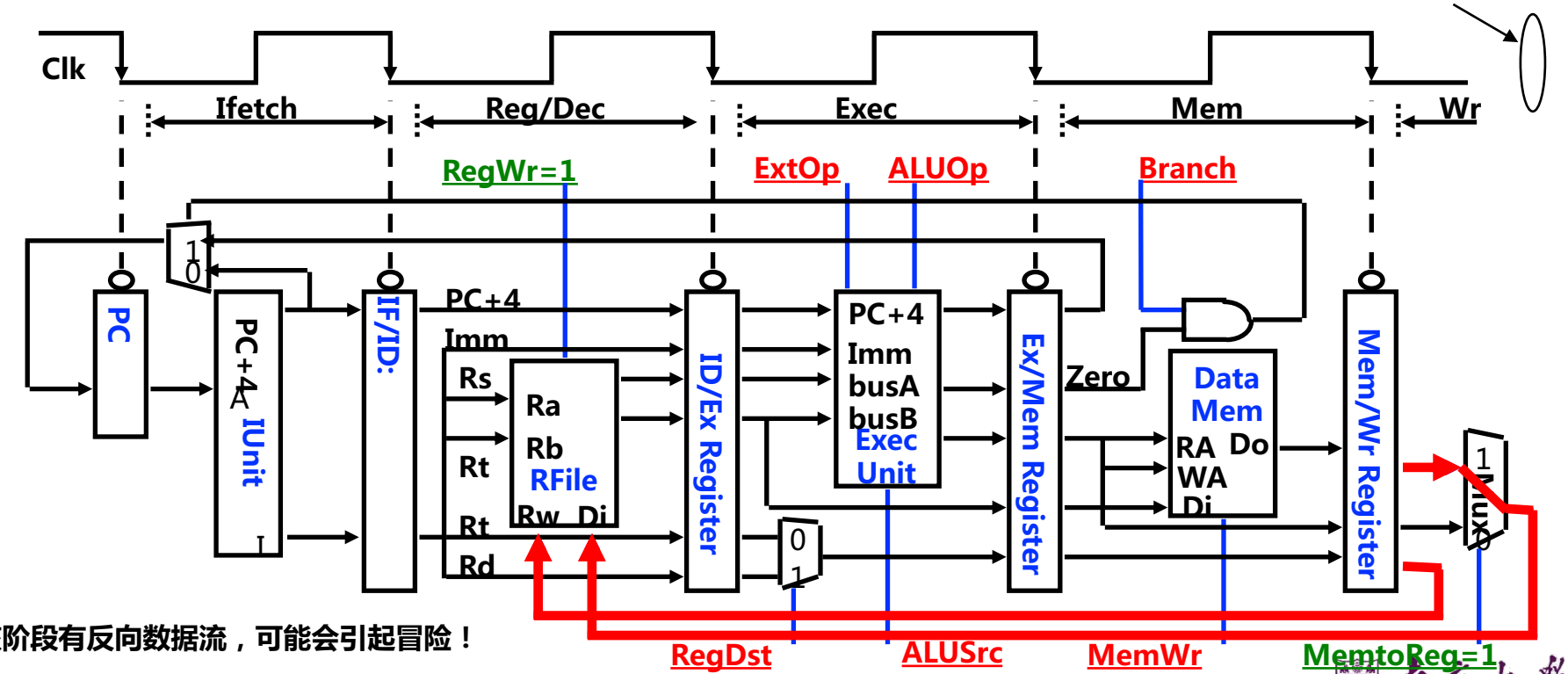


该阶段有反向数据流，可能会引起冒险！以后介绍。

周期以最长操作为准设计Cycle > T_{read}

Wr段：Load指令的回写(Write Back)阶段

- **Location 12:** lw $1, 0x100($2)　　**功能：**$1 <- Mem[($2) + 0x100]



该阶段有反向数据流，可能会引起冒险！

各阶段所经DataPath已有，控制信号如何得到？

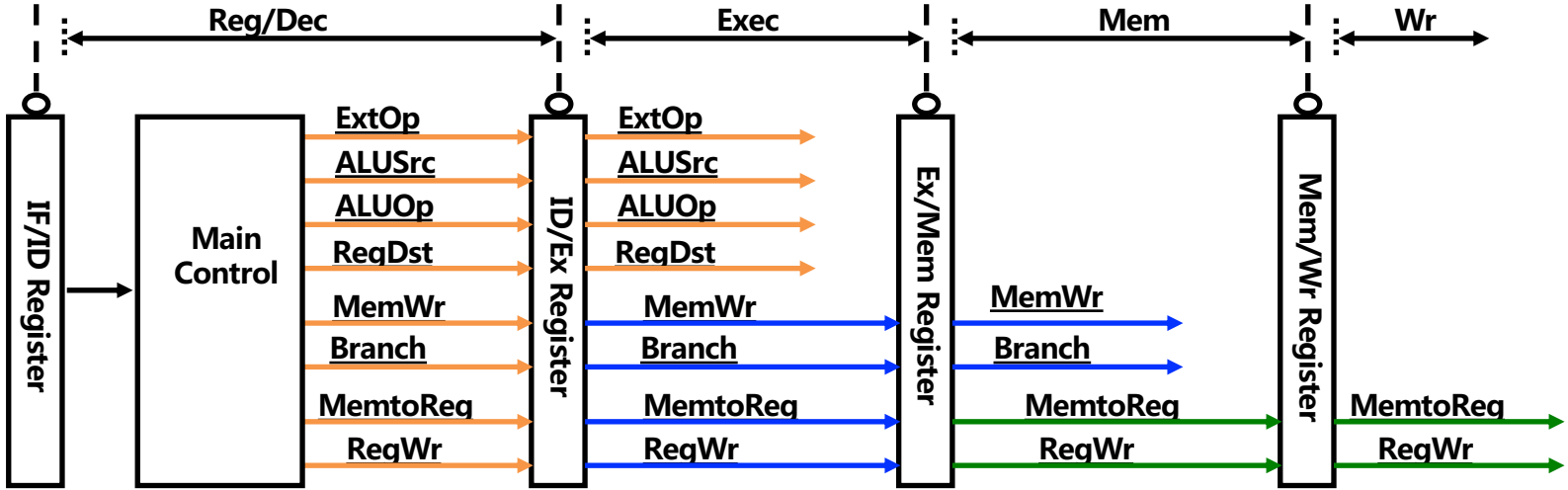
流水线中的控制信号

- 在**取数/译码（Reg/Dec）阶段**产生本指令每个阶段的所有控制信号
  - **Exec信号 (ExtOp, ALUSrc, ...)** 在1个周期后使用
  - **Mem信号 (MemWr, Branch)** 在2个周期后使用
  - **Wr信号 (MemtoReg, RegWr)** 在3个周期后使用

所以，控制信号也要保存在流水段寄存器中！

为什么 第1、2阶段没有控制信号?

IF和ID阶段的功能对每条指令来说都一样



各流水段部件在一个时钟内完成某条指令的某个阶段的工作！

在下个时钟到达时，把执行结果以及前面传递来的后面各阶段要用到的所有数据（如：指令、立即数、目的寄存器等）和控制信号保存到流水线寄存器中！

流水线中的控制信号

- 通过对前面流水线数据通路的分析，得知：
  - PC需要写使能吗？ 每个时钟都会改变PC，故不需要!
  - 流水段寄存器需要写使能吗？ 每个时钟都会改变流水段寄存器，故不需要!
  - **Ifecth阶段和Dec/Reg阶段都没有控制信号**
  - **Exec阶段的控制信号有四个**
    - ExtOp (扩展器操作)：1- 符号扩展；0- 零扩展
    - ALUSrc (ALU的B口来源)：1- 来源于扩展器；0- 来源于BusB
    - ALUOp (主控制器输出，用于辅助局部ALU控制逻辑来决定ALUCtrl)
    - RegDst (指定目的寄存器)：1- Rd；0- Rt
  - **Mem阶段的控制信号有两个**
    - MemWr (DM的写信号)：Store指令时为1，其他指令为0
    - Branch (是否为分支指令)：分支指令时为1，其他指令为0
  - **Wr阶段的控制信号有两个**
    - MemtoReg (寄存器的写入源)：1- DM输出；0- ALU输出
    - RegWr (寄存器堆写信号)：结果写寄存器的指令都为1，其他指令为0

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

流水线冒险及其处理

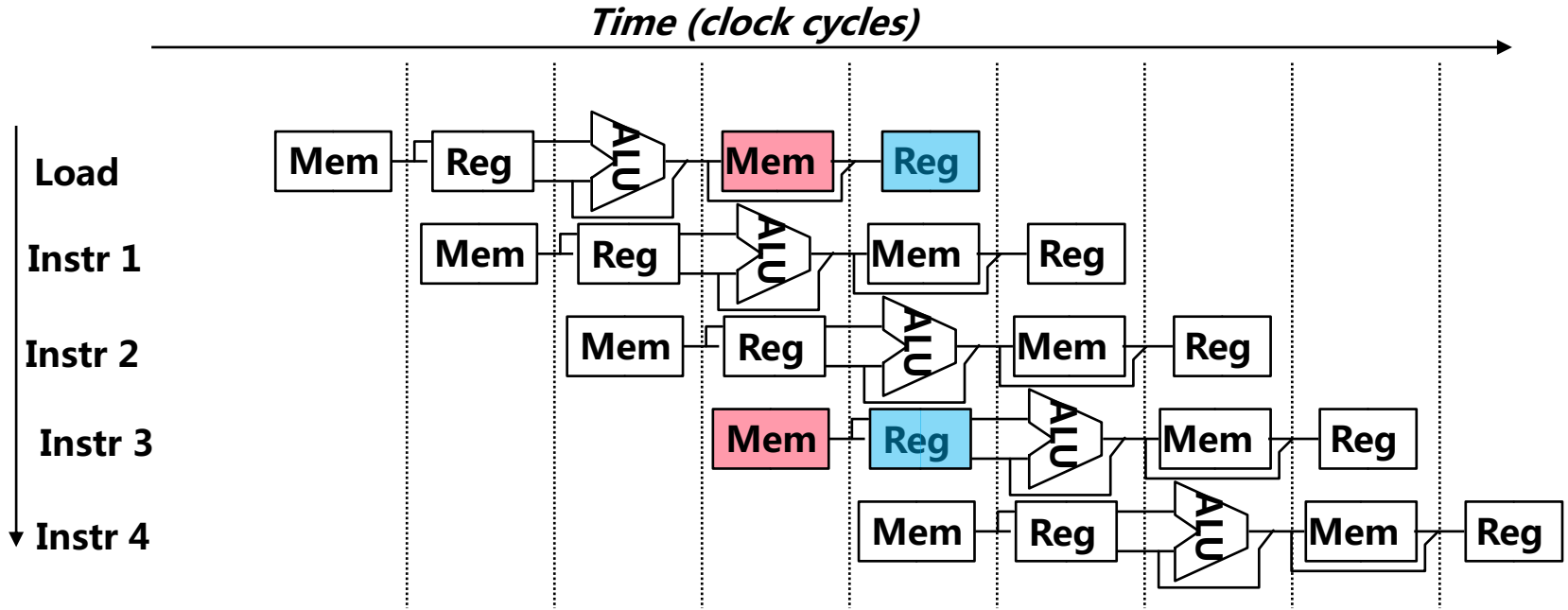
- **冒险(Hazards)：指流水线遇到无法正确执行后续指令或执行了不该执行的指令**
  - **结构冒险(Structural hazards)** (硬件资源冲突):

    现象：同一个部件同时被不同指令所使用
    - 一个部件每条指令只能使用1次，且只能在特定周期使用
    - 设置多个部件，以避免冲突。如指令存储器IM 和数据存储器DM分开
  - **数据冒险(Data hazards)** (数据冲突):

    现象：后面指令用到前面指令结果数据时，前面指令的结果还没产生
    - 采用转发(Forwarding/Bypassing)技术
    - Load-use冒险需要一次阻塞(stall)
    - 编译程序优化指令顺序
  - **控制冒险(Control hazards)** (指令执行顺序改变):

    现象：转移或异常改变执行流程，后继指令在目标地址产生前已被取出
    - 采用静态或动态分支预测
    - 编译程序优化指令顺序(分支延迟)

结构冒险现象



只有一个存储器时，在Load指令取数据同时又取指令的话，则发生冲突！

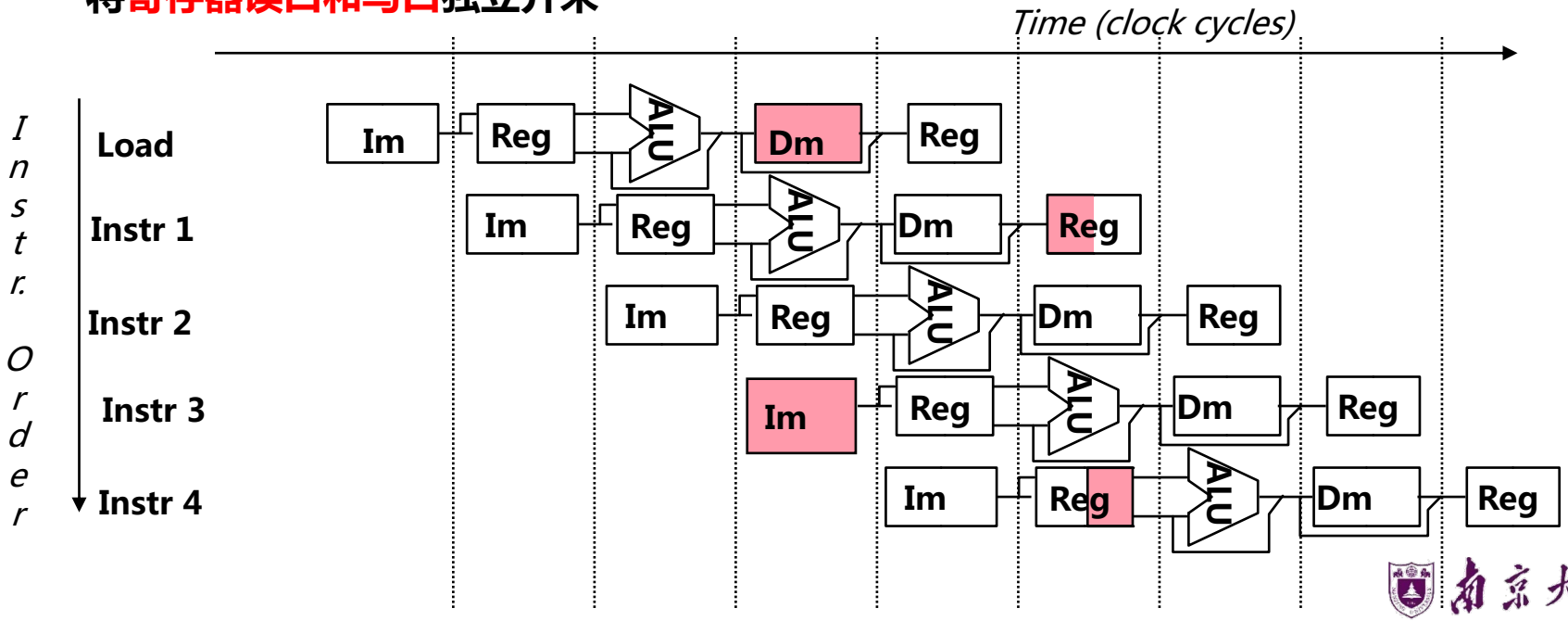
如果不对寄存器堆的写口和读口独立设置的话，则发生冲突！

结构冒险也称硬件资源冲突：同一个执行部件被多条指令使用。

结构冒险的解决方法

为了避免结构冒险，规定**流水线数据通路中功能部件的设置原则为**：

- **每个部件在特定的阶段被用！**（如：ALU总在第三阶段被用！）
- **将指令存储器(Im)和数据存储器(Dm)分开**
- **将寄存器读口和写口独立开来**



数据冒险现象

举例说明：以下指令序列中，寄存器r1会发生数据冒险

想一下，哪条指令的r1是老的值？哪条是新的值？

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1 ,r11
```

读r1时，add指令正在执行加法(EXE)，老值!

读r1时，add指令正在传递加法结果(MEM), 老值!

读r1时，add指令正在写加法结果到r1(WB), 老值!

读r1时，add指令已经把加法结果写到r1, 新值

画出流水线图能很清楚理解！

三类数据冒险现象:

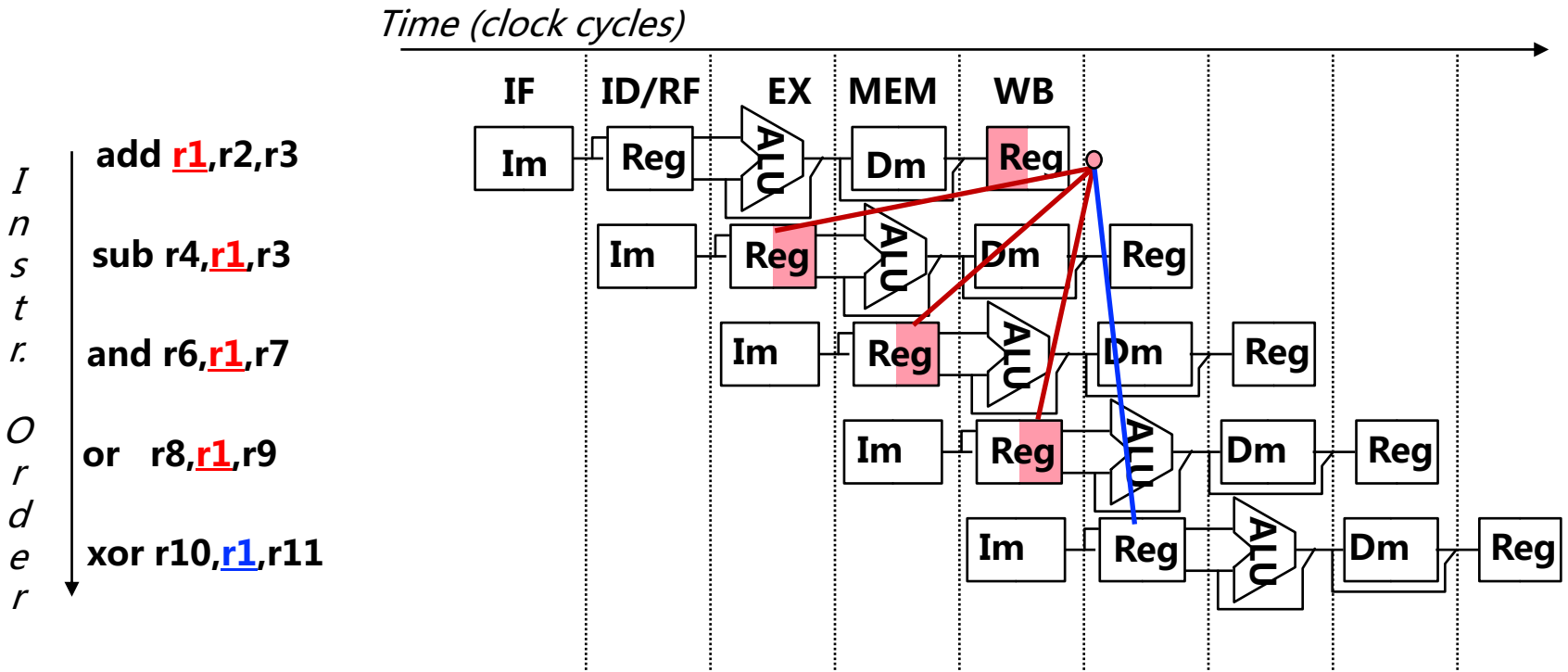
RAW: 写后读（基本流水线中经常发生，如上例）

WAR：读后写（基本流水线中不会发生，乱序执行时会发生）

WAW：写后写（基本流水线中不会发生，乱序执行时会发生）

本讲介绍基本流水线，所以仅考虑RAW冒险

数据冒险：r1



最后一条指令的r1才是新的值！

如何解决这个问题？

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：合理实现寄存器堆的读/写操作（不能解决所有数据冒险）
  - 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4：转发（Forwarding或Bypassing 旁路）技术
  - 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！)**
- 方法5：编译优化：调整指令顺序（不能解决所有数据冒险）

实现“转发”和“阻塞” 要修改数据通路：

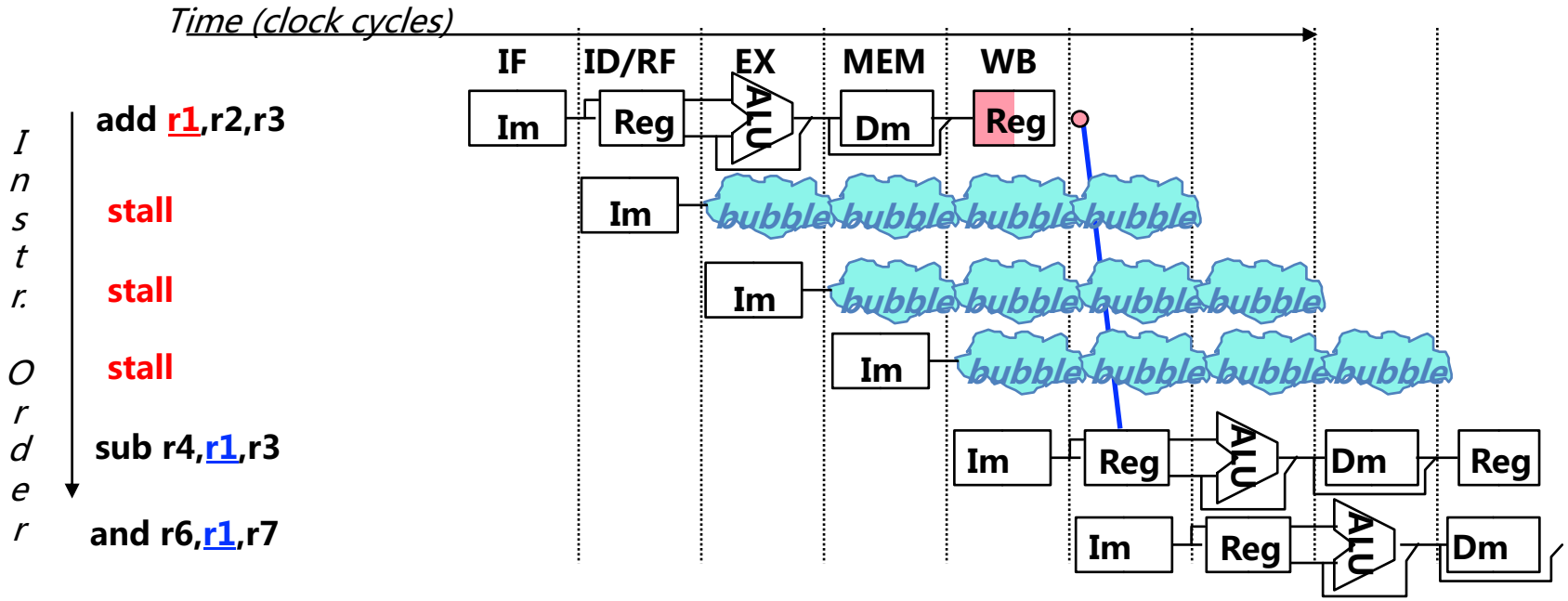
（1）检测何时需要“转发” ，并控制实现“转发”

（2）检测何时需要“阻塞” ，并控制实现“阻塞”

数据冒险-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！**

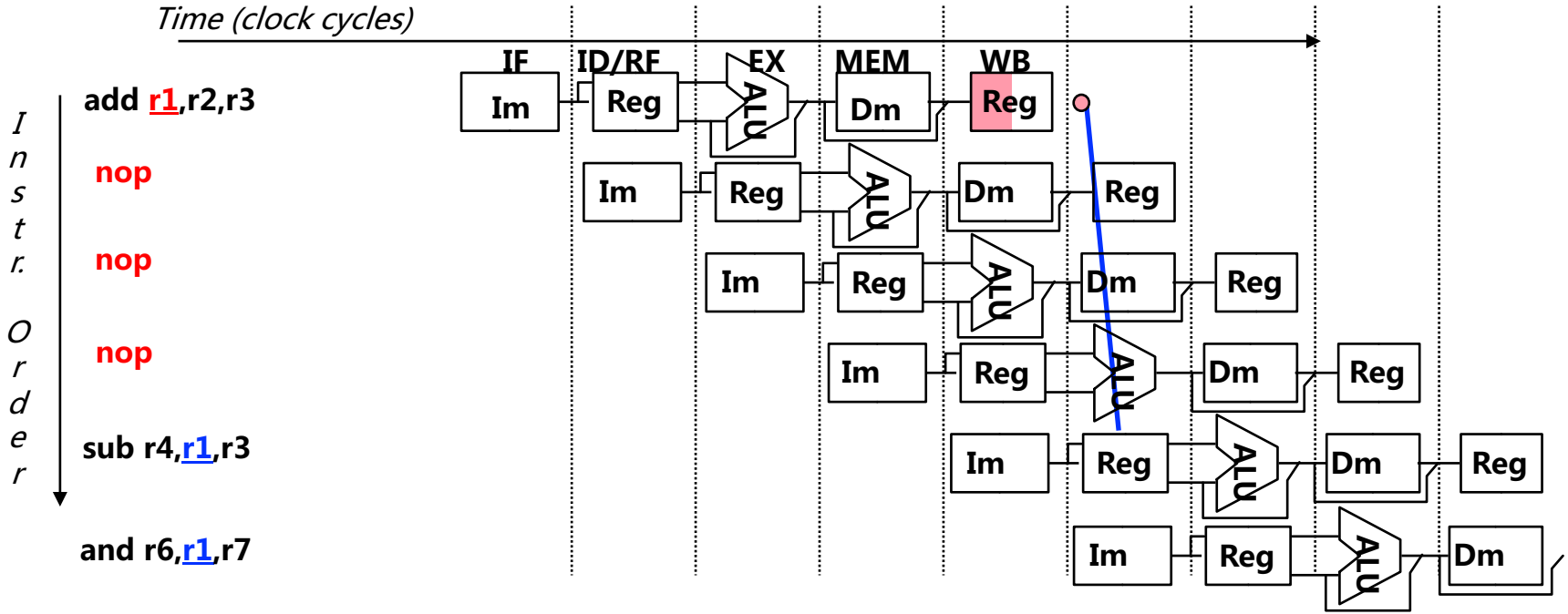
  这种做法称为流水线阻塞，也称为**插入“气泡Bubble”**



- 缺点：控制比较复杂，需要改数据通路；指令被延迟三个时钟执行。

数据冒险-方案2：软件上插入无关指令

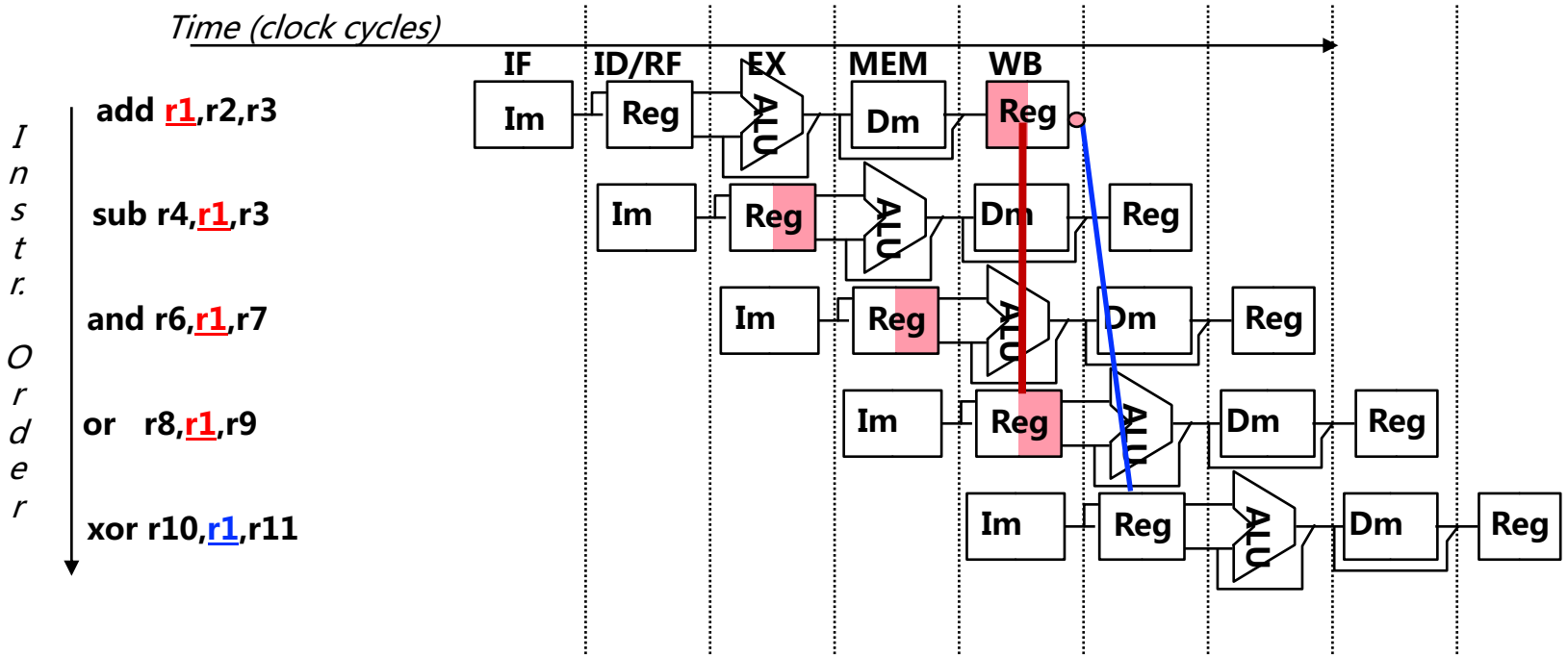
- **由编译器插入三条NOP指令，浪费三条指令的空间和时间，是最差的做法。**
  (好处：数据通路简单，即无需改数据通路)



与方案1比，哪个更快？　一样，都是多三个时钟周期！

数据冒险-方案3：同一周期内寄存器堆先写后读

- 寄存器堆的读口和写口是相互独立的部件！

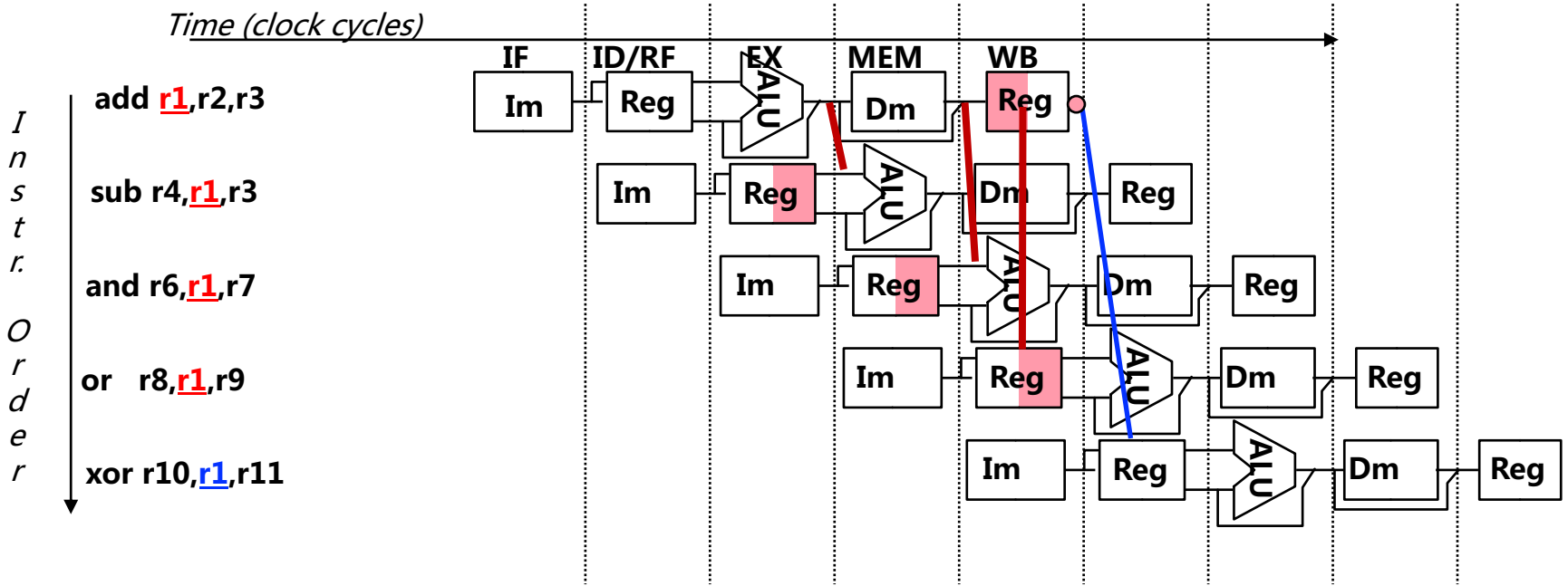


寄存器**写口/读口**分别在**前/后半周期**进行操作，使写入数据被直接读出。

但是，只能解决部分数据冒险！

数据冒险-方案4：利用数据通路中的中间数据：转发+阻塞

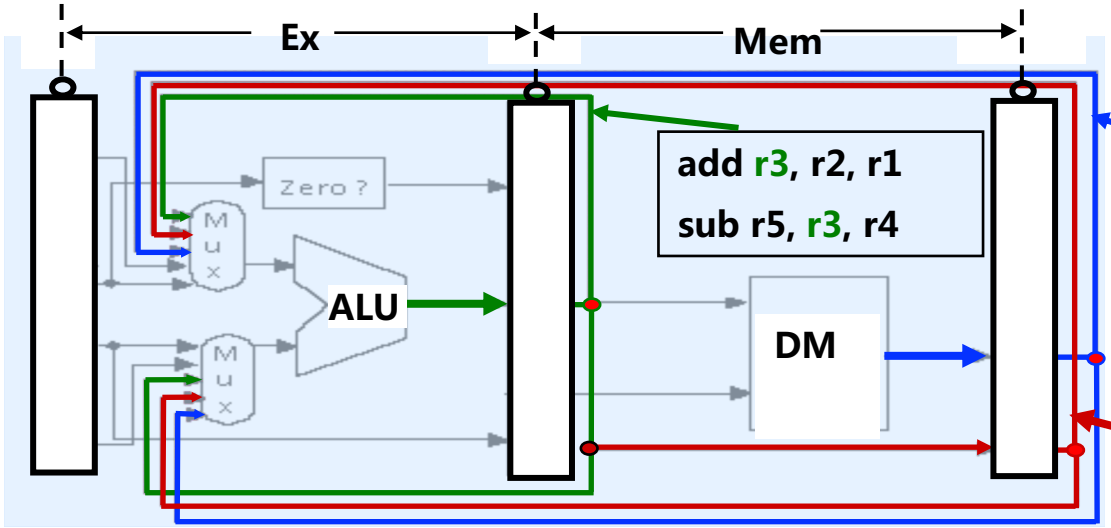
- 仔细观察后发现：流水段寄存器中已有需要的值r1！ 在哪个流水段R中？



把数据从流水段寄存器中直接取到ALU的输入端 ← 称为转发或旁路

硬件上的改动以支持“转发”技术

- 加MUX，使流水段寄存器值返送ALU输入端

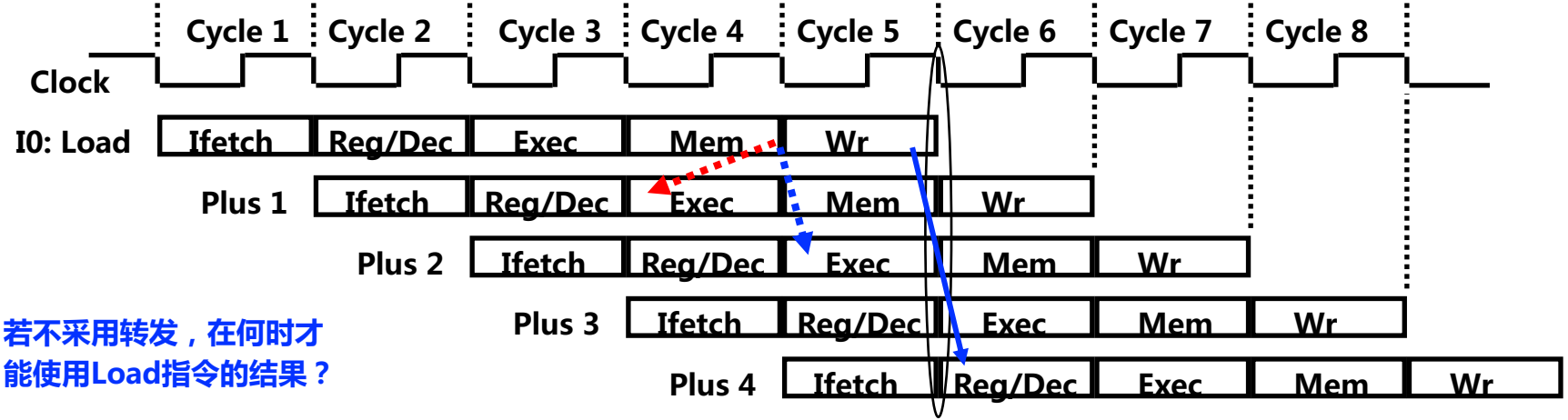


如果指令序列为

lw r3, 100(r1)
or r6, r3, r1
sub r5, r3, r4

能用“转发”技术解决第1、2两条指令间的数据冒险吗？
（不能！）

回顾：Load指令引起的延迟现象

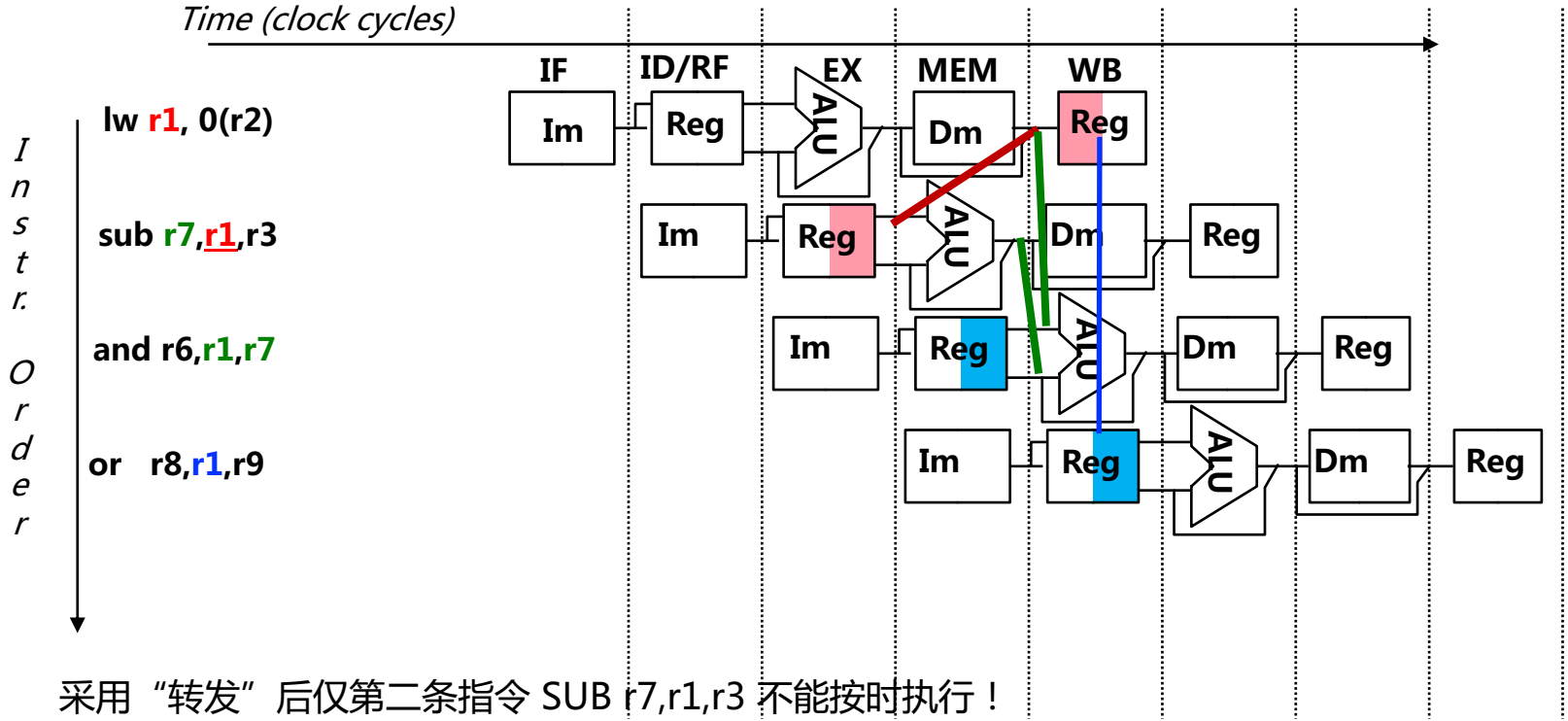


- **Load指令最早在哪个流水线寄存器中开始有后续指令需要的值？**

  - 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)”

“转发”技术使Load-use冒险只需延迟一个周期

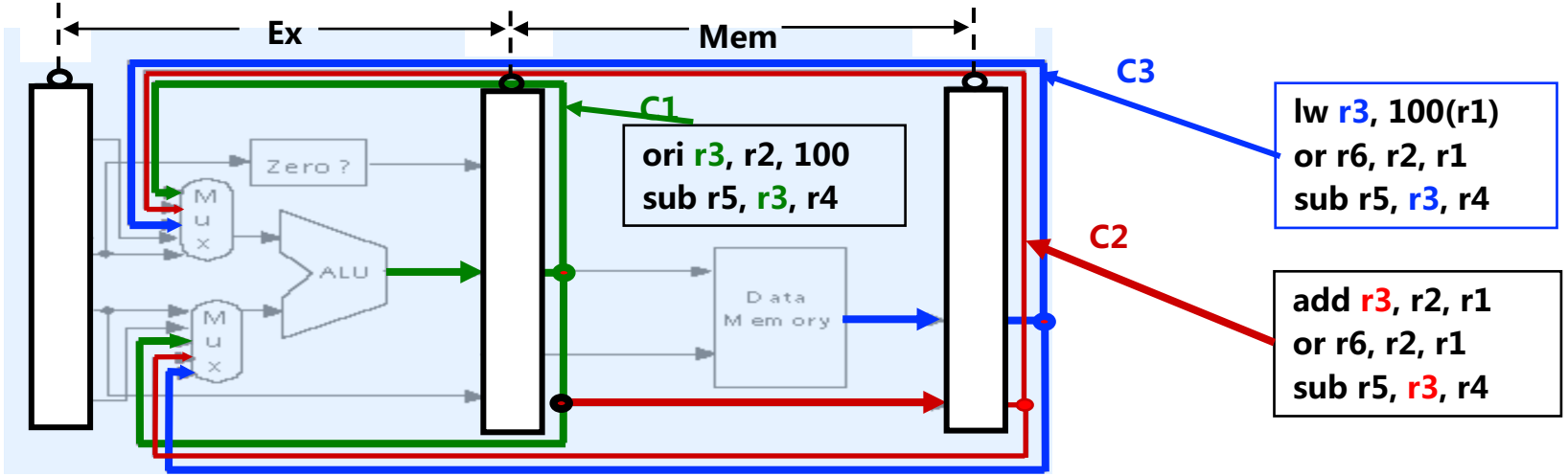


采用“转发”后仅第二条指令 SUB r7,r1,r3 不能按时执行！

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！**

数据冒险处理最佳方案：“转发” + “Load-use阻塞”

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

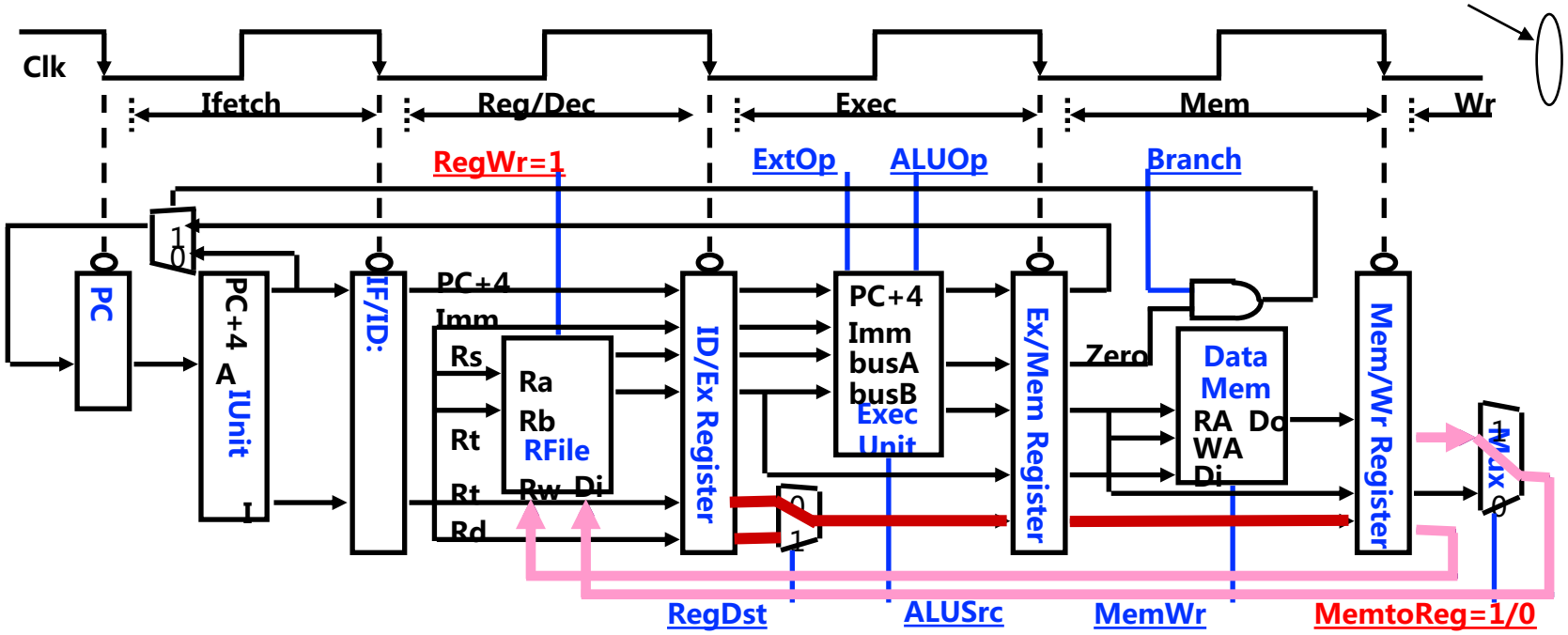
C2(a): MEM/WB. RegisterRd=ID/EX. RegisterRs

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

指令的回写（Write Back）阶段



- Rd还是Rt取决于是R-型指令，还是I-型指令！
- 若是beq指令会怎样？

```
beq r3, r2, 100
add r4, r3, r2
sub r5, r3, r2
```

因为beq指令无需写结果，故不能进行转发！

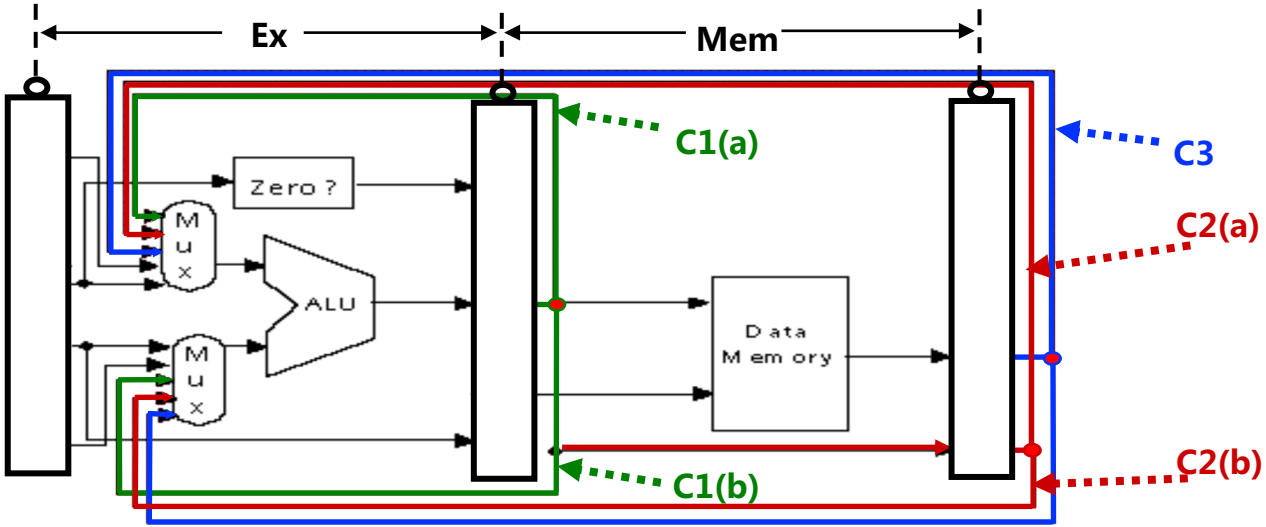
转发条件的进一步完善

- 以下两种情况下，转发会发生错误
  - **指令的结果不写入目的寄存器Rd时**
    - 例如，Beq指令只对rs和rt相减，不写结果到目的寄存器Rt
    - 即：EX / MEM 或 MEM / WB 流水段寄存器的RegWr信号为0
  - **Rd等于$0时**
    - 例如，指令 sll $0, $1, 2 的转发结果为(R[$1]<<2) ，但实际上应该是0
- 因此，修改转发条件为：
  - **C1(a)**: EX/MEM.RegWr
    and EX/MEM. RegisterRd ≠ 0
    and EX/MEM. RegisterRd=ID/EX. RegisterRs
  - **C1(b)**: EX/MEM.RegWr
    and EX/MEM. RegisterRd ≠ 0
    and EX/MEM. RegisterRd=ID/EX. RegisterRt
  - **C2(a)**: MEM/WB.RegWr
    and MEM/WB. RegisterRd ≠ 0
    and MEM/WB. RegisterRd=ID/EX. RegisterRs
  - **C2(b)**: MEM/WB.RegWr
    and MEM/WB. RegisterRd ≠ 0
    and MEM/WB. RegisterRd=ID/EX. RegisterRt

```
beq r3, r2, 100
add r4, r3, r2
sub r5, r3, r2
```

转发路径和转发条件

- 加MUX，使流水段寄存器值返送ALU输入端



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

红线和蓝线可以合并，在原数据通路中确实是合并在一起的。记得吗？

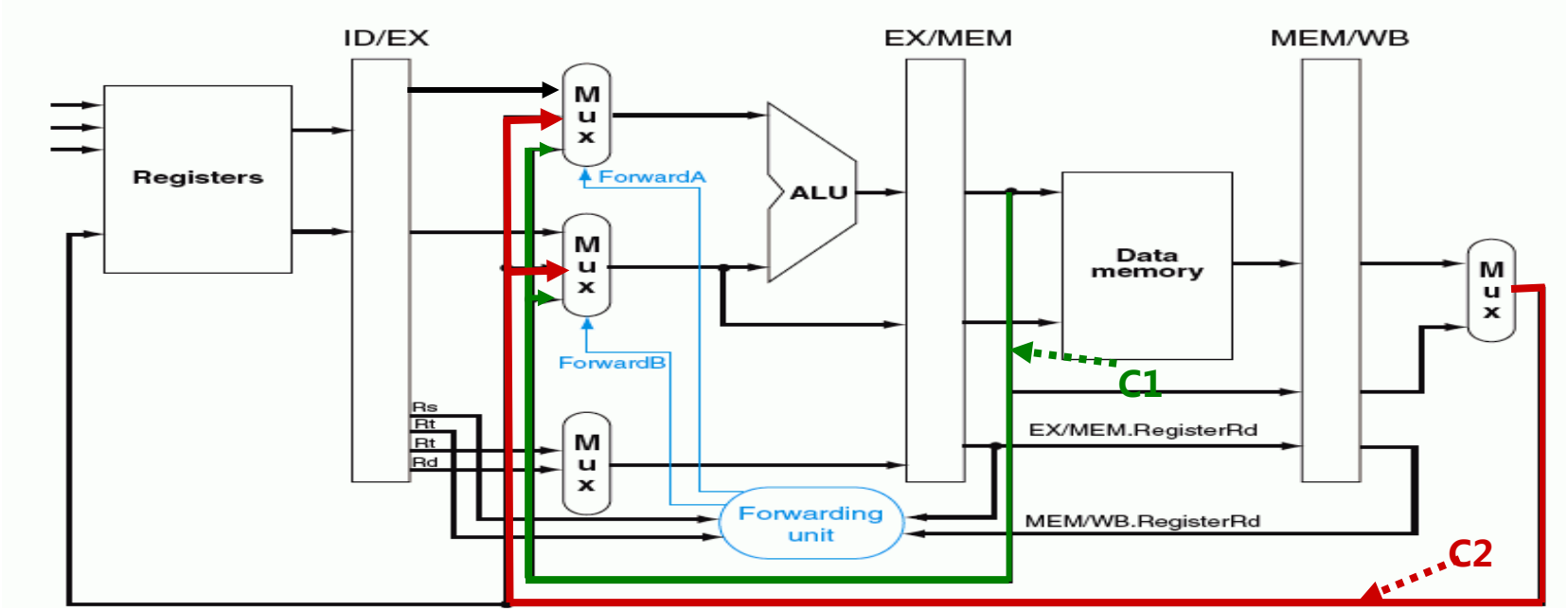
由一个二路选择器（控制端为MemtoReg）合并输出到寄存器堆！

所以, 无需另有一个检测条件C3! C3就是C2！

C1和C2分别反映哪两条指令的关系呢？

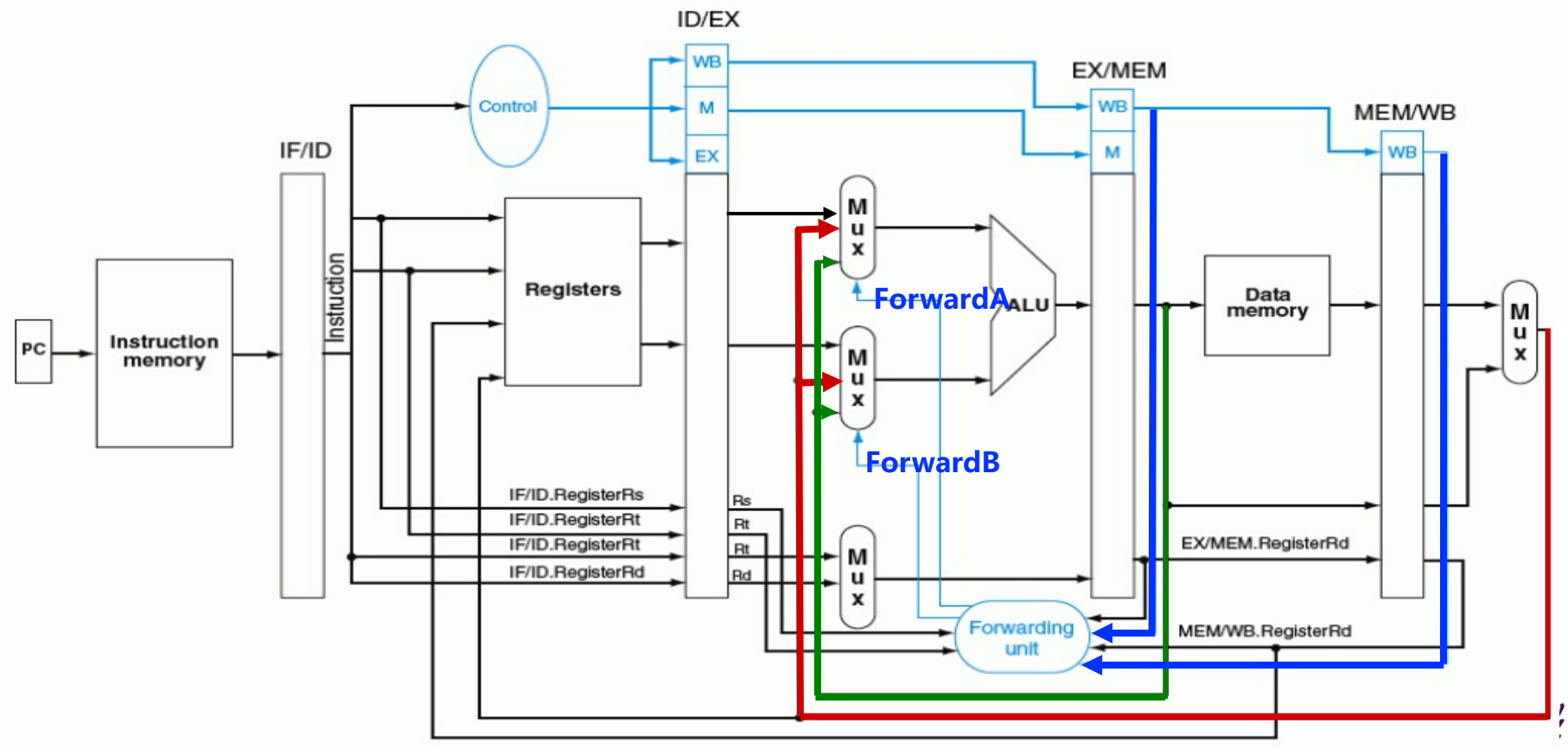
转发路径和转发条件

ForwardA (ForwardB) = { 01 当c2(a)=1 或C2(b)=1 时
10 当c1(a)=1或C1(b)=1 时



“转发检测”部件中缺何条件？ MEM/WB.RegWr=1、EX/MEM.RegWr=1

带转发的流水线数据通路



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