结构冲突和数据冲突

2022年秋

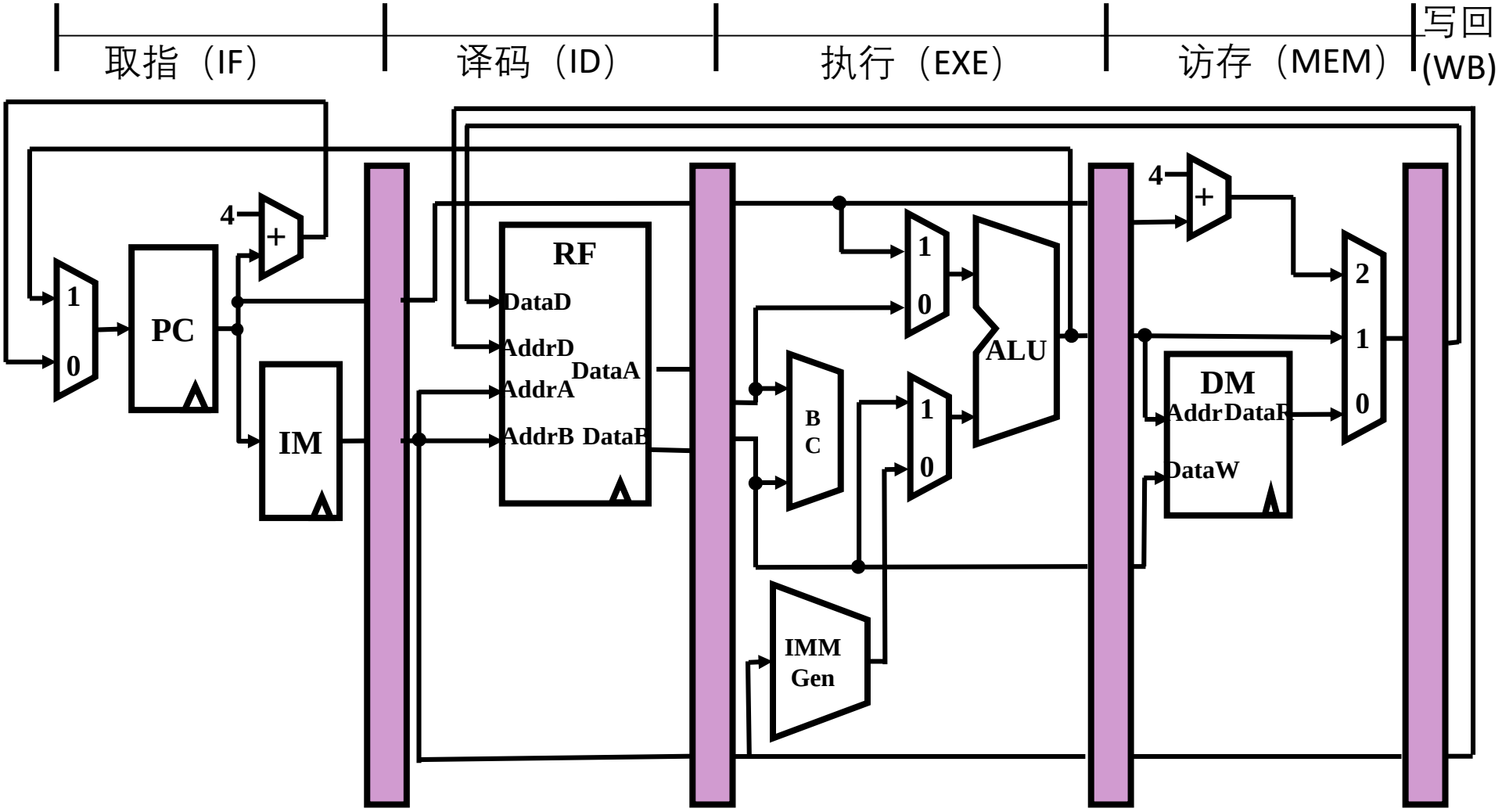
本讲概要

- 指令流水实现原理
- 结构冲突
  - 检测
  - 避免
- 数据冲突
  - 检测
  - 避免
- 小结

流水线的实现原理

- 指令流水的简单实现
- 每一条指令的实现至多需要5个时钟周期。这5个时钟周期如下：
  - 取指令周期（**IF**）
  - 指令译码 / 读寄存器周期（**ID**）
  - 执行 / 有效地址计算周期（**EX**）
  - 存储器访问 / 分支完成周期（**MEM**）
  - 写回周期（**WB**）
- 不同类型的指令在以上5个时钟周期中进行的操作各不相同。

流水线的实现原理


取指（IF）
译码（ID）
执行（EXE）
访存（MEM）
写回（WB）
4
+
1
0
PC
IM
RF
DataD
AddrD
DataA
AddrA
AddrB
DataB
B
C
1
0
ALU
IMM
Gen
4
+
DM
Addr
DataR
DataW
2
1
0

流水线的实现原理

- 简单的基本流水线

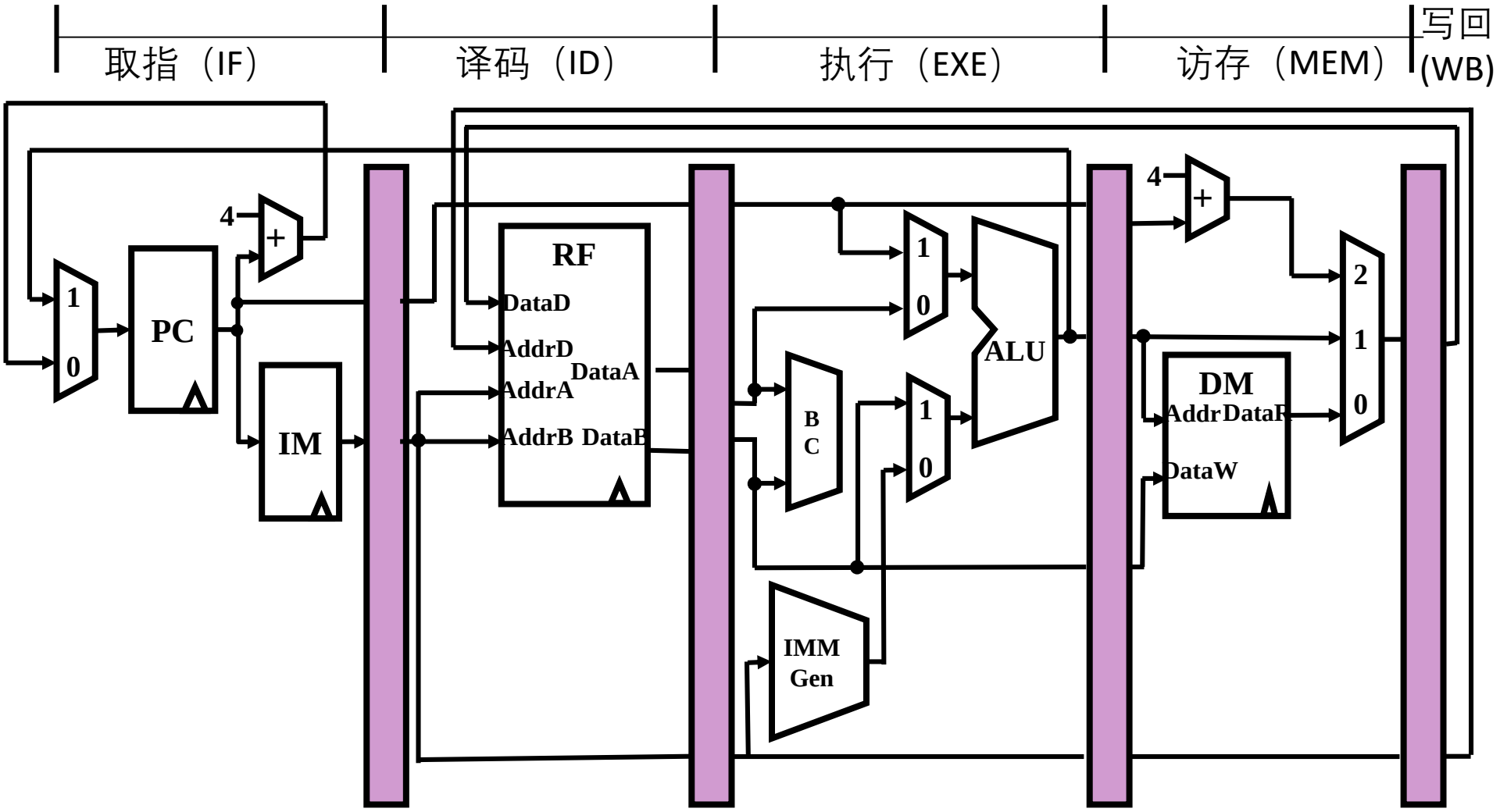
指令	时钟								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

- 每一个时钟周期启动一条新的指令，就可以使数据通路成功流水，每一个时钟周期就是流水线的一个流水段。每一条指令经过5个时钟周期执行完成，而在每一个时钟周期内，硬件将启动一条新的指令并执行5条不同指令的某个部分。

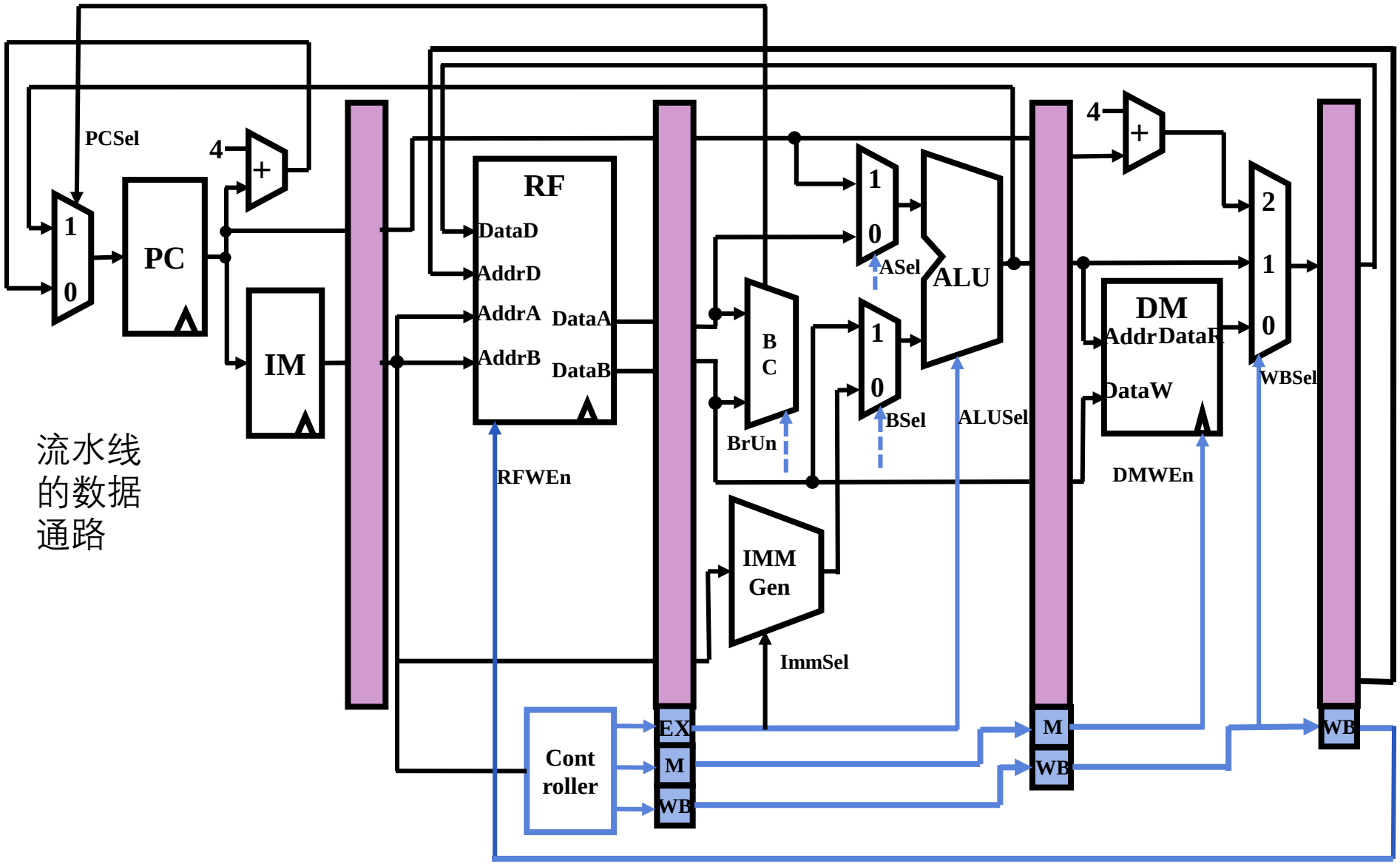
流水线的实现原理

- 在流水线的各个流水段之间加入了被称为流水线寄存器（流水线锁存器）的寄存器堆，并在这些寄存器堆上标明所连接的流水段。
  - 所有用于在同一条指令的各个时钟周期之间保存临时数据的寄存器，都归入流水线寄存器这一类中。
  - 流水线寄存器保存着从一个流水段传送到下一个流水段的所有数据和控制信号。
- PC值多路选择器被移到IF段，这样做的目的是保证对PC值的写操作只出现在一个流水段内，否则当分支转移成功的时候，流水线中两条指令都试图在不同的流水段修改PC值，从而发生写冲突。
- 每个时刻，每条指令都只在一个流水段上是活动的，因此，任何指令所作的任何动作都发生在一对流水线寄存器之间。

流水线的实现原理


取指（IF）
译码（ID）
执行（EXE）
访存（MEM）
写回（WB）
4
+
1
0
PC
IM
RF
DataD
AddrD
DataA
AddrA
AddrB
DataB
B
C
1
0
ALU
IMM
Gen
4
+
DM
Addr
DataR
DataW
2
1
0

流水线
的数据
通路


PCSel
4
+
1
0
PC
IM
RF
DataD
AddrD
AddrA
DataA
AddrB
DataB
RFWEn
B
C
BrUn
1
0
ASel
ALU
1
0
BSel
ALUSel
IMM
Gen
ImmSel
EX
M
WB
Cont
roller
M
WB
4
+
DM
Addr
DataR
DataW
DMWEn
2
1
0
WBSel
WB

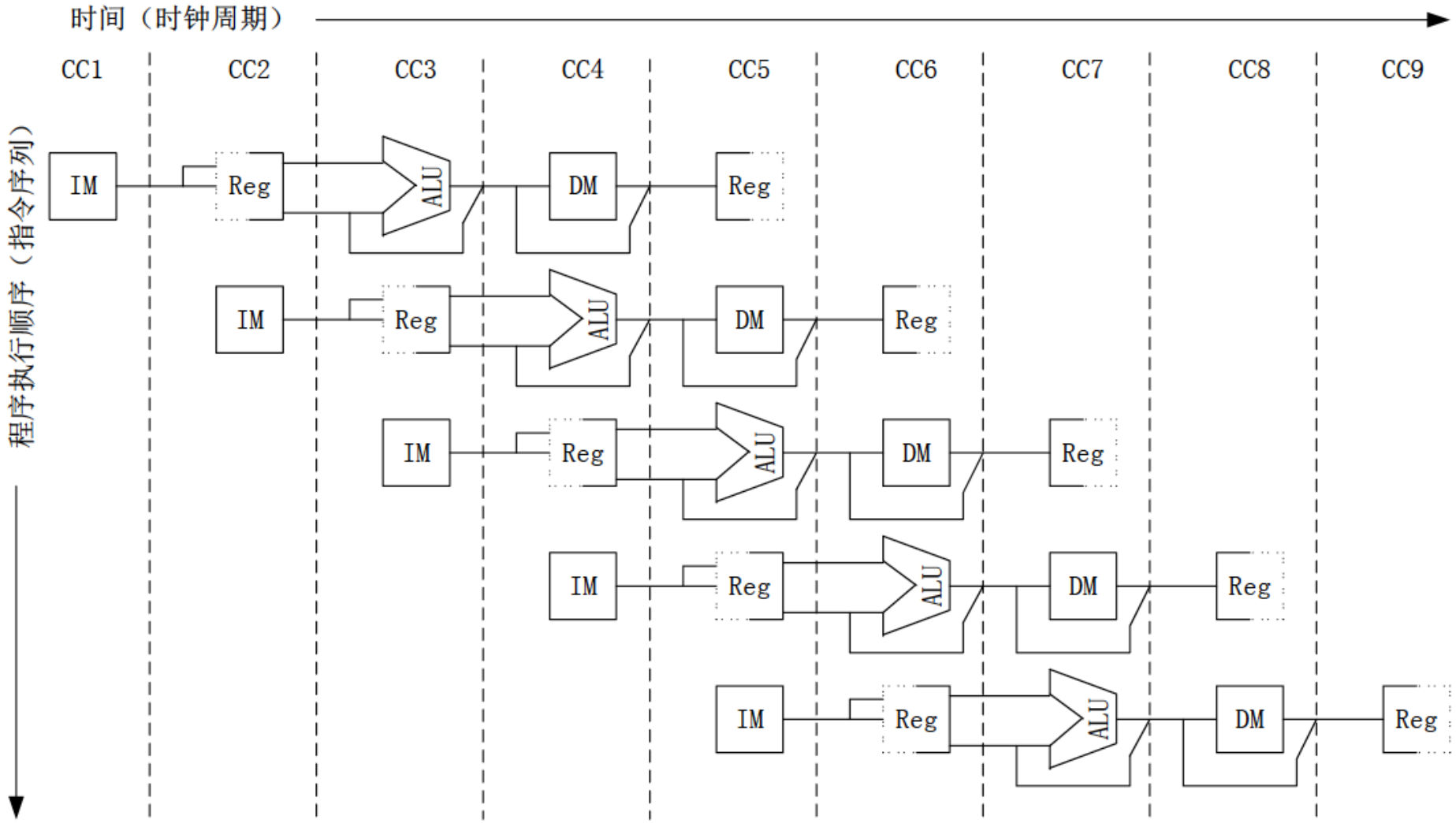
各阶段寄存器保存的值

- IF/ID
  - PC
  - IR
- ID/EXE
  - A、B、PC
  - Inst (imm, dst)
  - 所有控制信号
- EXE/MEM
  - PC、ALU运算结果、B
  - Inst
  - 控制信号：MEM及WB
- MEM/WB
  - DataD: 目的寄存器、ALU结果、存储器读出的结果三者之一
  - AddrD (inst)
  - WB阶段控制信号

流水线的实现原理

- 实际的流水线就这么简单吗？——NO！
  - 必须保证在指令重叠执行时不会存在任何流水线资源冲突问题，即要保证流水线的各段在同一个时钟周期内不会使用相同的数据通路资源。
- 简化的流水线数据通路
  - 下图从使用流水线资源的角度描述上述流水线的流水过程，这张图显示了不同数据通路的重叠，其中周期5表示稳定状态。
  - 在包围每个流水段的线框中，如果实线在右侧，说明是读操作；如果实线在左侧，说明是写操作；其他部分用虚线。
  - 主要的功能部件都在不同的时钟周期内使用，因而多条指令重叠执行时引入的冲突很少。
    - 分开的指令存储器（IM）和数据存储器（DM）。
    - 在两个流水线段都使用了寄存器：ID段读，WB段写。
    - 没有考虑PC的问题，流水要求IF段要形成新的PC值。

流水线的实现原理


时间（时钟周期）
CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
CC9
程序执行顺序（指令序列）
IM
Reg
ALU
DM
Reg

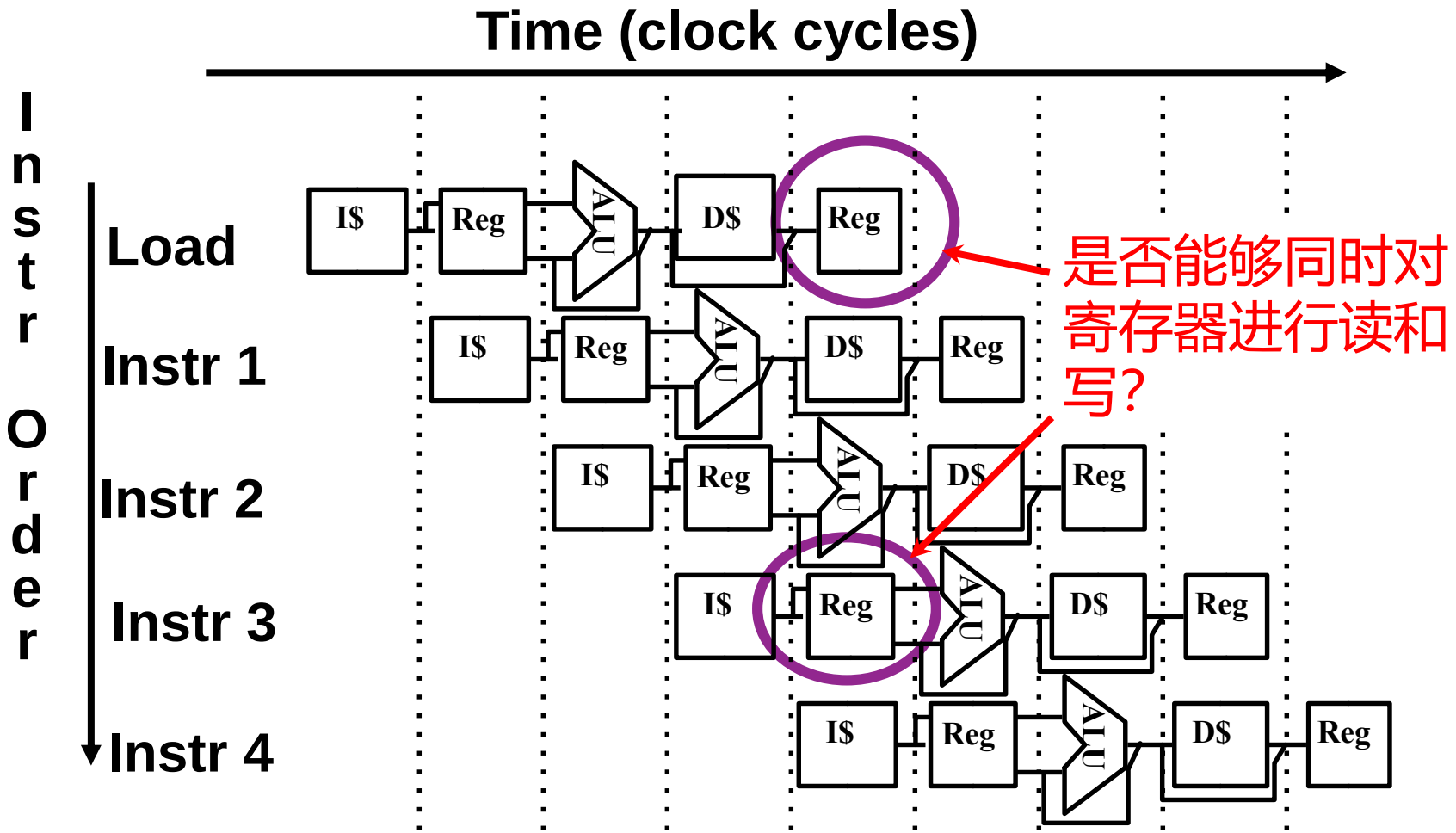
流水线的冲突问题

- 什么是流水线中的“冲突”？
  - 在流水线中经常有一些被称为“冲突”的情况发生，它使得指令序列中下一条指令无法按照设计的时钟周期执行，这些“冲突”可能会降低流水线可以获得的理想性能。
- 流水线中的冲突可以分为以下三种类型
  - 结构冲突：指令在重叠执行的过程中，硬件资源满足不了指令重叠执行的要求，发生硬件资源冲突而产生的冲突。
  - 数据冲突：是指在同时重叠执行的几条指令中，一条指令依赖于前面指令执行结果数据，但是又在指定位置得不到时发生的冲突。
  - 控制冲突：是指流水线中的分支指令或者其他需要改写PC的指令造成的冲突。
- 流水线冲突问题是流水线执行过程中的主要障碍，会给流水线中指令序列的顺利执行带来许多不利的影响。如果不能较好地处理流水线冲突问题，就可能影响流水线的性能，甚至使程序运行产生错误的结果。

结构冲突

- **问题:** 两条或者多条在流水线中的指令去访问相同的物理资源
- **解决方法 1:** 指令依次使用资源，一些指令暂停stall
- **解决方法 2:** 增加硬件资源
- 可以增加更多的硬件资源来解决结构冲突

寄存器结构冲突


Time (clock cycles)
Instr Order
Load
Instr 1
Instr 2
Instr 3
Instr 4
I$
Reg
ALU
D$
Reg
是否能够同时对寄存器进行读和写?

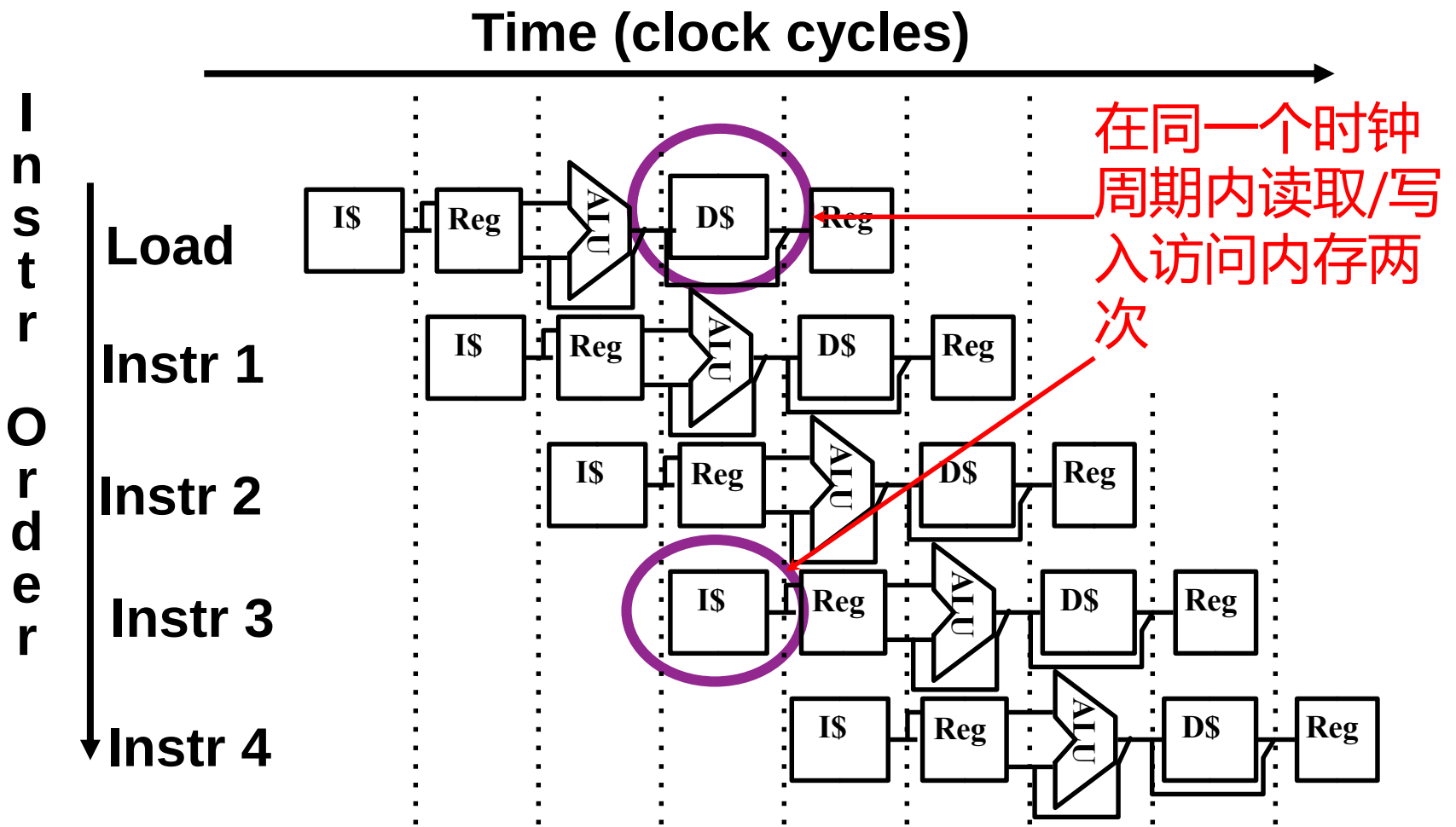
寄存器结构冲突

- 对于每一条指令来说:
  - 在译码阶段最多读两个操作数
  - 在写回阶段写入一个值
- 设置不同的“端口”来避免结构冲突
  - 两个独立的读端口和一个独立的写端口
- 在一个时钟周期同时完成三次访问

寄存器结构冲突

- 两个可选的解决方案:
  1) 通过独立的读和写端口进行支持 (在实验中会采用这种办法，方便单个流水级中使用)
  2) 双沿访问Double Pumping: 寄存器访问分成两步，第一步在前半个时钟周期准备写入；然后在下降沿写入；第二部在后半个时钟周期读出。
     - 可以节约一个时钟周期
     - 能这么做的原因是寄存器文件的访问非常快速，比ALU阶段的访问时间的一半还短
- **在相同的时钟内同时完成对于寄存器的读写是可以的**
- **注意，上面的（2）在FPGA中做不到，一定用（1）这个方式实现**

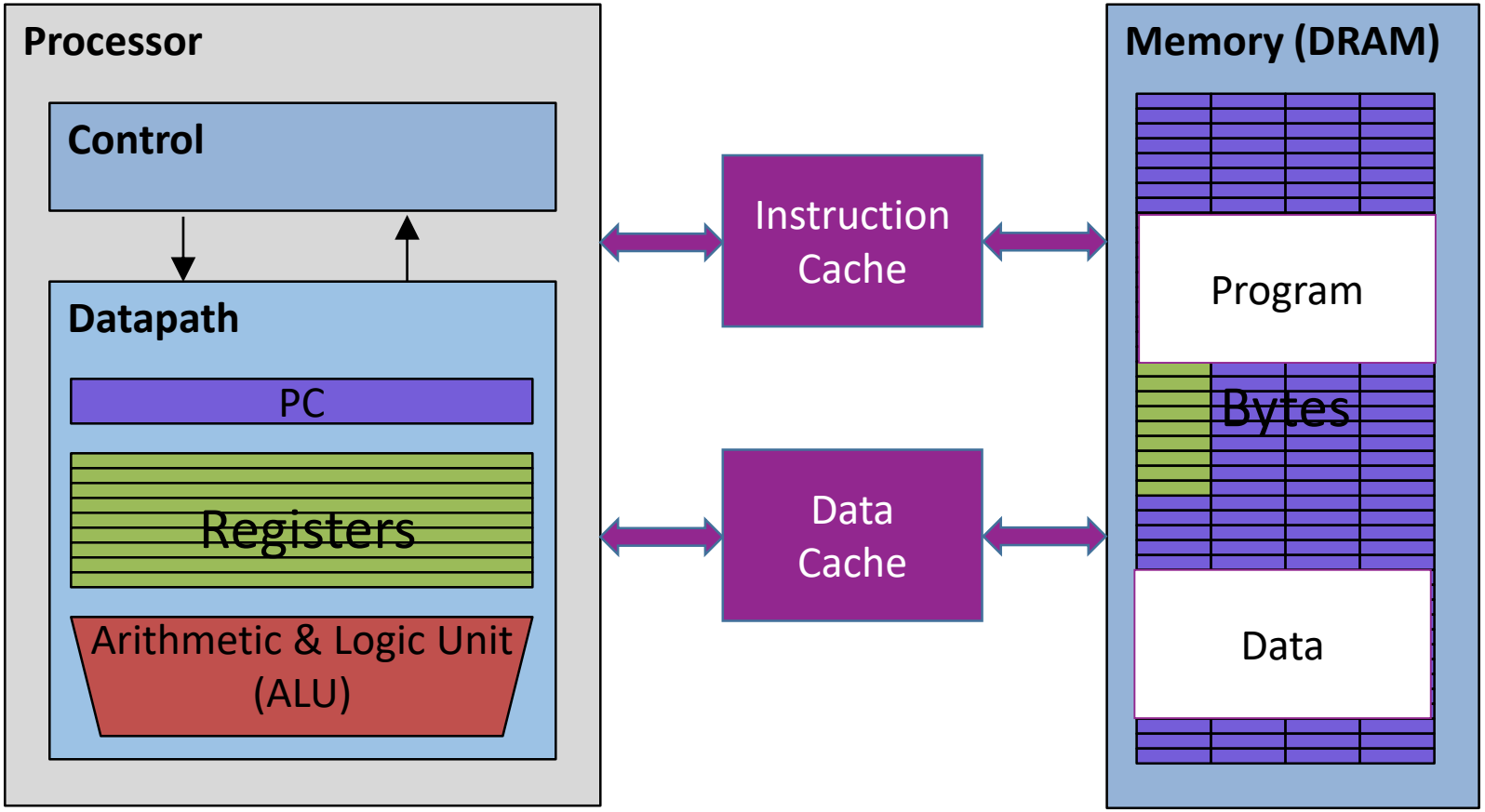
内存结构冲突


Time (clock cycles)
Instr Order
Load
Instr 1
Instr 2
Instr 3
Instr 4
I$
Reg
ALU
D$
Reg
在同一个时钟周期内读取/写入访问内存两次

内存结构冲突

- 内存物理结构的限制
  - 在同一个时钟周期内无法完成两次读或者一次读一次写
  - load/store指令需要使用访问内存中的数据，所有的代码需要从访问内存中的指令
  - 哪怕是对于单周期的处理器也是结构冲突的
- 流水线处理器可以通过暂停流水线的办法来解决，对于每一次取指在出现内存结构冲突的时候需要等待一个时钟周期（流水线中的气泡）
- 另外的解决办法：将指令内存和数据内存分开，这样取指和数据内存访问不会发生结构冲突

Instruction and Data Caches

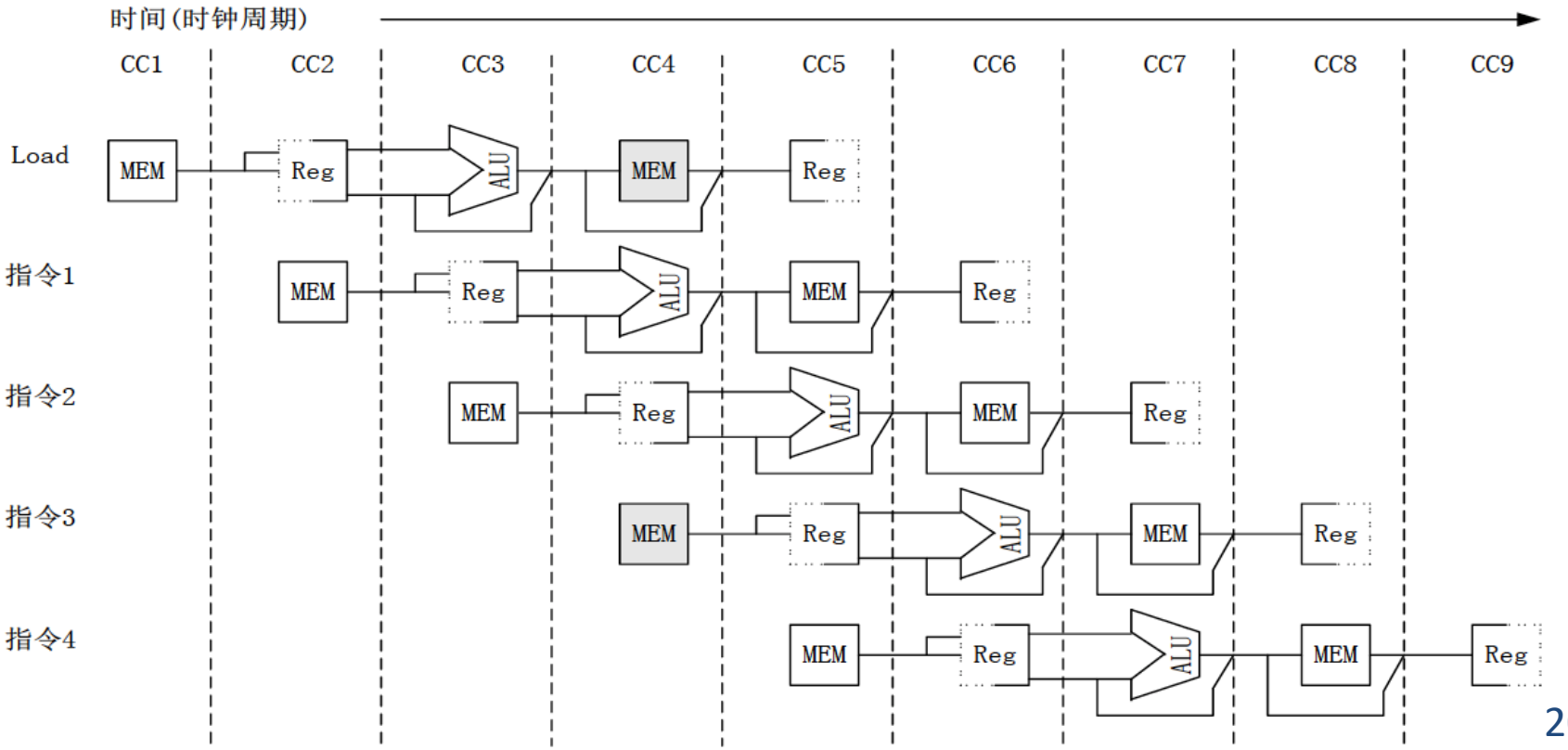


Caches: small and fast “buffer” memories

哈佛结构，通常使用在L1缓存中

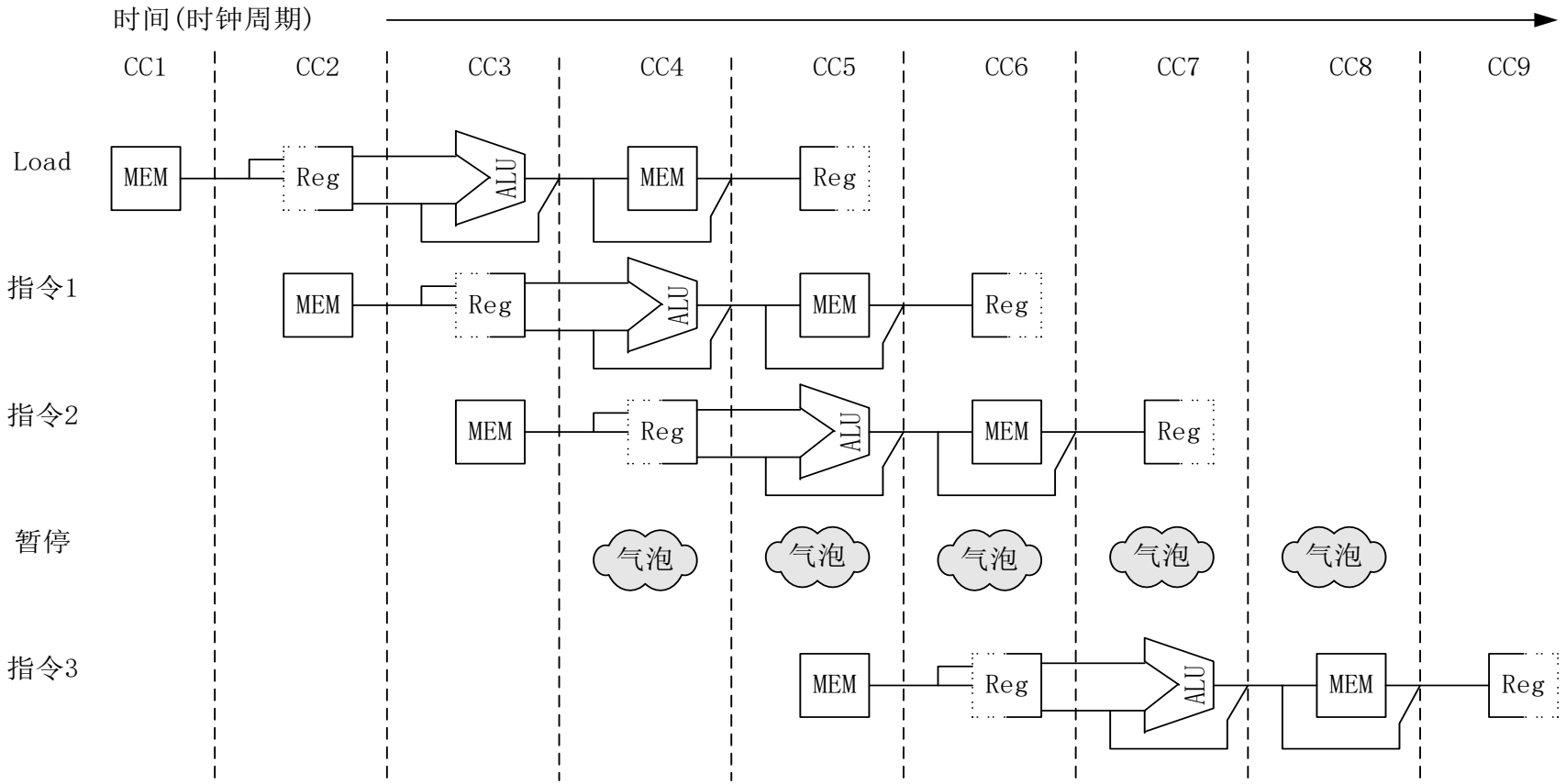
结构冲突和相应解决方法

- 如果因资源冲突而无法使用某种指令组合，那么就称流水线产生了结构冲突。
  - 暂停流水线执行，插入等待周期
  - 增加资源，解决资源冲突



结构冲突和相应解决方法

- 消除结构冲突的最简单方法就是引入暂停周期，这必然要降低流水线的性能。

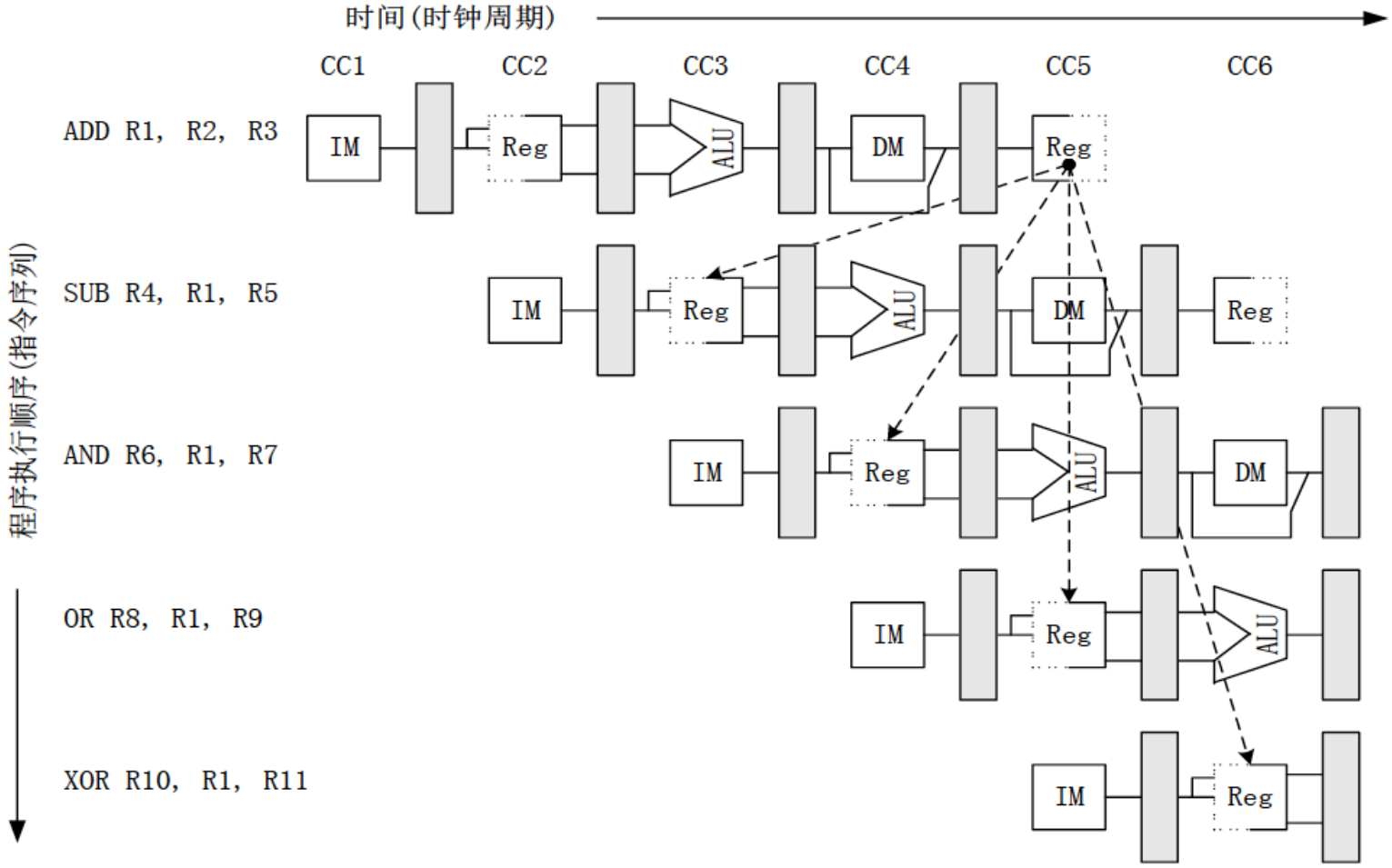


结构冲突和相应解决方法

- 解决结构冲突的基本方法
  - 结构冲突的起因是资源争用，那么可以考虑采用资源充分重复设置的方法来避免结构冲突。
- 解决存储器争用冲突的办法
  - 如果指令和数据放在同一个存储器，可使用双端口存储器，其中一个端口存取数据，另一个端口取指令。
  - 设置两个存储器，其中一个作为数据存储器，另一个作为指令存储器。
  - 上述两种方案中，取指令和访问数据可以并行进行，不会发生结构冲突。

数据冲突和相应解决方法

- 流水线技术可以通过指令的重叠执行来改变指令的相对执行时间，这就可能导致流水线中的指令序列读写操作数的顺序发生改变，而不同于非流水线时的指令序列读写操作数的顺序。



数据冲突的分类

- 按照指令读写寄存器顺序对数据冲突分类
  - 对于两条指令i和j，假设指令i在j之前进入流水线，下面讨论几种不同的数据冲突。
- 写后读冲突（RAW：Read After Write）
  - 指令j的执行需要使用指令i的计算结果，但是当它们在流水线中重叠执行时，指令j可能在指令i将其计算结果写入之前就先行对保存该计算结果的寄存器进行了读操作，这样指令j读出的寄存器值就是错误的。
  - 这是最常见的一种数据冲突。

数据冲突分类

- 写后写冲突（WAW：Write After Write）
  - 指令j和指令i的目的操作数相同，但是当它们在流水线中重叠执行时，指令j可能在指令i将其计算结果写入之前就先行对保存该计算结果的寄存器进行了写操作，这样就导致了寄存器写入顺序的错误，此时，目的寄存器的内容是指令i写入的值，而不是指令j写入的值。
- RISC-V指令流水不会发生WAW冲突
  - 如果在流水线中不只一个流水段可以进行写操作，或者当流水线暂停某条指令的执行时，允许该指令之后的其他指令继续执行，就可能发生这种数据冲突。但是RISC-V流水线中的指令不会发生这种数据冲突，因为流水中只有WB段才会写寄存器。
  - 如果对流水线进行改变，将ALU结果的写回操作移到MEM段进行，因为这时计算结果已经有效，同时再假定访问数据存储器需要两个流水段，那么流水线中执行的指令就可能发生WAW冲突。

LW R1,0(R2)	IF	ID	EX	MEM1	MEM2	WB
Add R1,R2,R3		IF	ID	EX	WB	

数据冲突分类

- 读后写冲突（WAR：Write After Read）
  - 指令j可能在指令i读取某个源寄存器的内容之前就对该寄存器进行了写操作，结果就是导致了指令i后来读取的值是错误的。
- RISC-V指令流水中不会发生WAR冲突
  - 因为RISC-V流水线在ID段完成所有的读操作，而在WB段完成所有的写操作。

数据冲突举例 (1/2)

- 考虑下面的指令序列

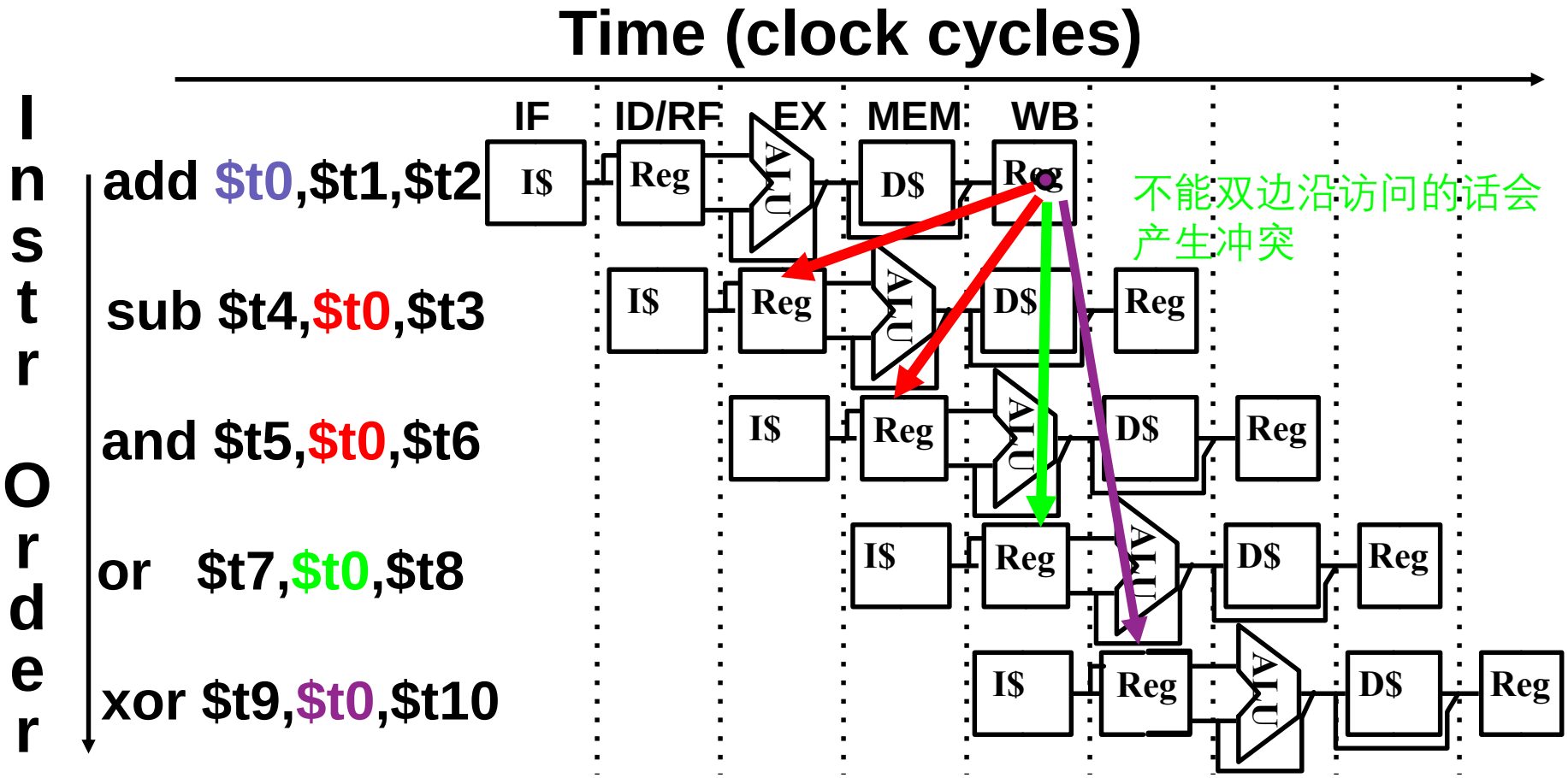
```
add $t0, $t1, $t2
sub $t4, $t0, $t3
and $t5, $t0, $t6
or  $t7, $t0, $t8
xor $t9, $t0, $t10
```

Stored during WB

Read during ID

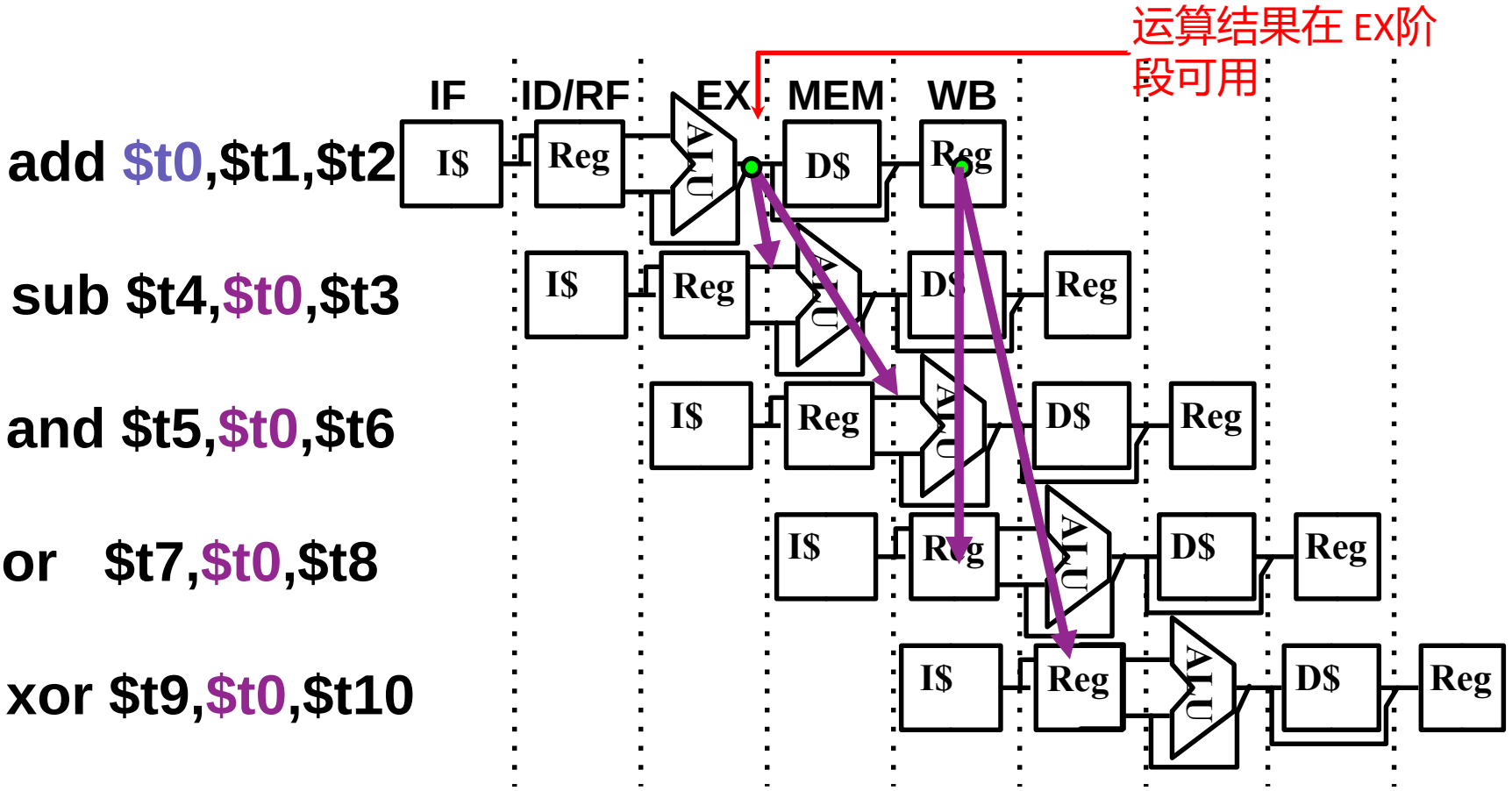
数据冲突举例(2/2)

- 数据后传产生冲突



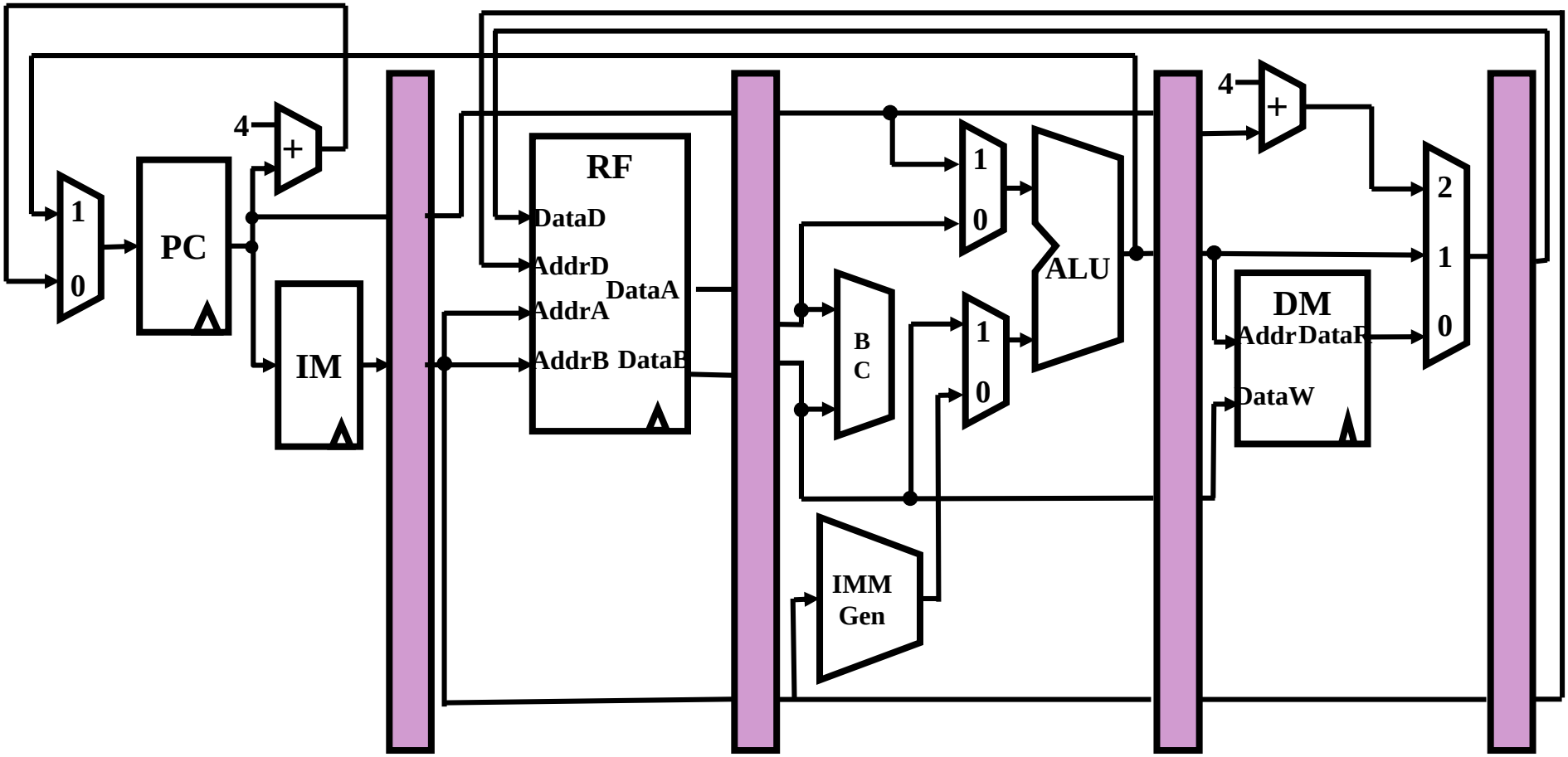
数据冲突解决办法: 数据旁路

- 结果可用的时候即可前传，无需先保存到RegFile



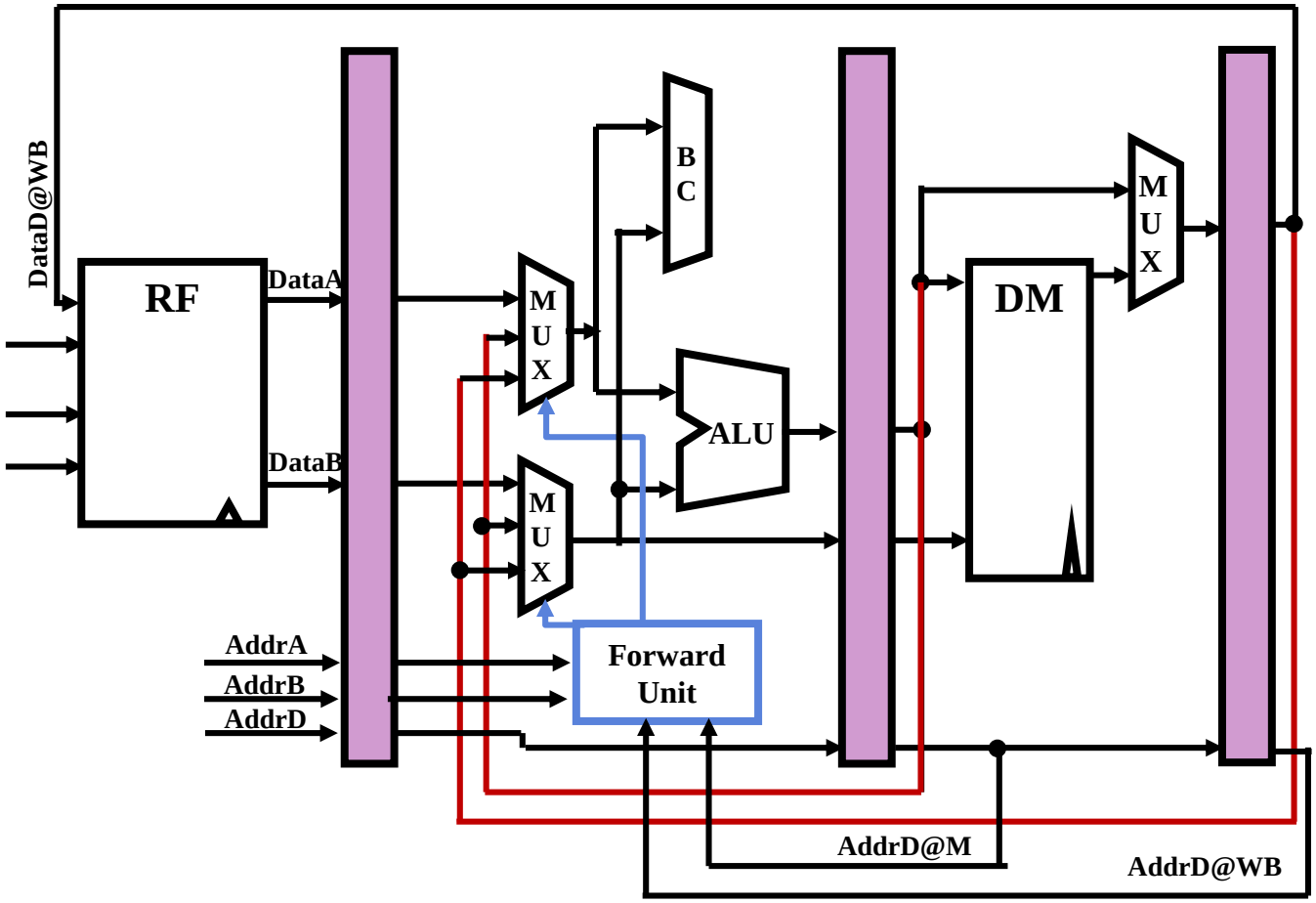
支持旁路的数据通路 (1/2)

- 数据通路需要做什么样的修改?



支持旁路的数据通路 (2/2)

- *使用前传部件传输数据*
  - Pictured: ALU → ALU in one and 2 cycles



检测数据冲突

- 数据冲突的类型
  - EXE段冲突
  - MEM段冲突
- EXE段数据冲突的检测
  - EXE冲突检测点：当前指令的ID/EX段和上一指令的EX/MEM段
    - 本条指令的源寄存器之一和上一条指令的目的寄存器相同
    - 上一条指令需要改写目的寄存器，且不是0寄存器
    - RegWrite@MEM AND AddrD@MEM != 0
    - AND AddrD@MEM = AddrA(B)@EX
- MEM段数据冲突的检测

Forwarding实现示例

```
always_comb begin: reg1_forward
    if (raddr1 == 5'b0) begin
        true_gpr_rdata1 = '0;
    end else if (mem_gpr_we && mem_gpr_waddr == raddr1) begin
        true_gpr_rdata1 = mem_gpr_wdata;
    end else if (wb_gpr_we && wb_gpr_waddr == raddr1) begin
        true_gpr_rdata1 = wb_gpr_wdata;
    end else begin
        true_gpr_rdata1 = gpr_rdata1;
    end
end

always_comb begin: reg2_forward
    if (raddr2 == 5'b0) begin
        true_gpr_rdata2 = '0;
    end else if (mem_gpr_we && mem_gpr_waddr == raddr2) begin
        true_gpr_rdata2 = mem_gpr_wdata;
    end else if (wb_gpr_we && wb_gpr_waddr == raddr2) begin
        true_gpr_rdata2 = wb_gpr_wdata;
    end else begin
        true_gpr_rdata2 = gpr_rdata2;
    end
end
```

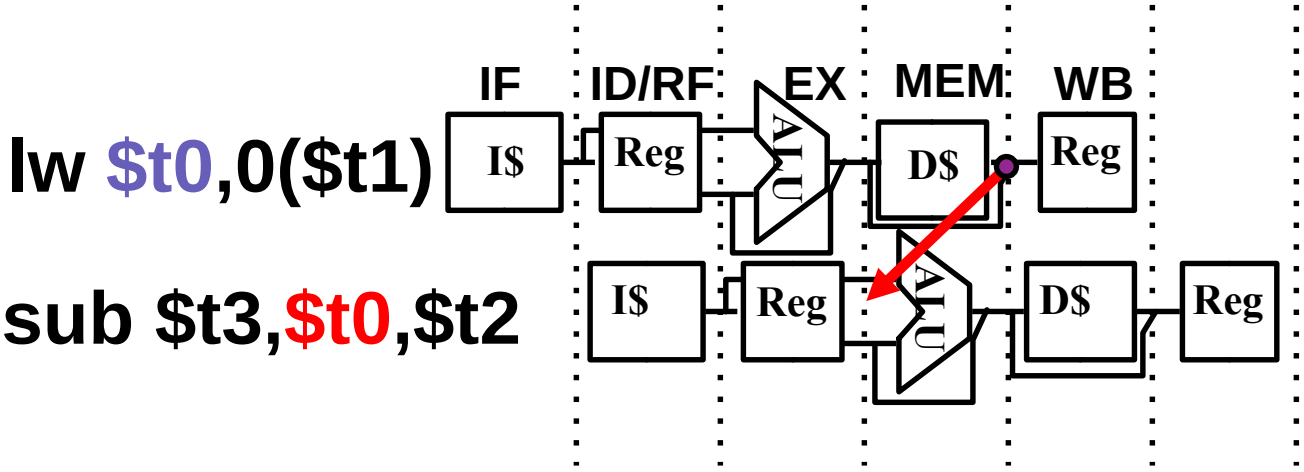
寄存器数据冲突

```
// read port 1
always_comb begin : read_port1
    if (gpr_we && gpr_raddr1 == gpr_waddr) begin
        gpr_rdata1 = gpr_wdata;
    end else begin
        gpr_rdata1 = registers[gpr_raddr1];
    end
end

// read port 2
always_comb begin : read_port2
    if (gpr_we && gpr_raddr2 == gpr_waddr) begin
        gpr_rdata2 = gpr_wdata;
    end else begin
        gpr_rdata2 = registers[gpr_raddr2];
    end
end
```

数据的装入使用冲突 (1/4)

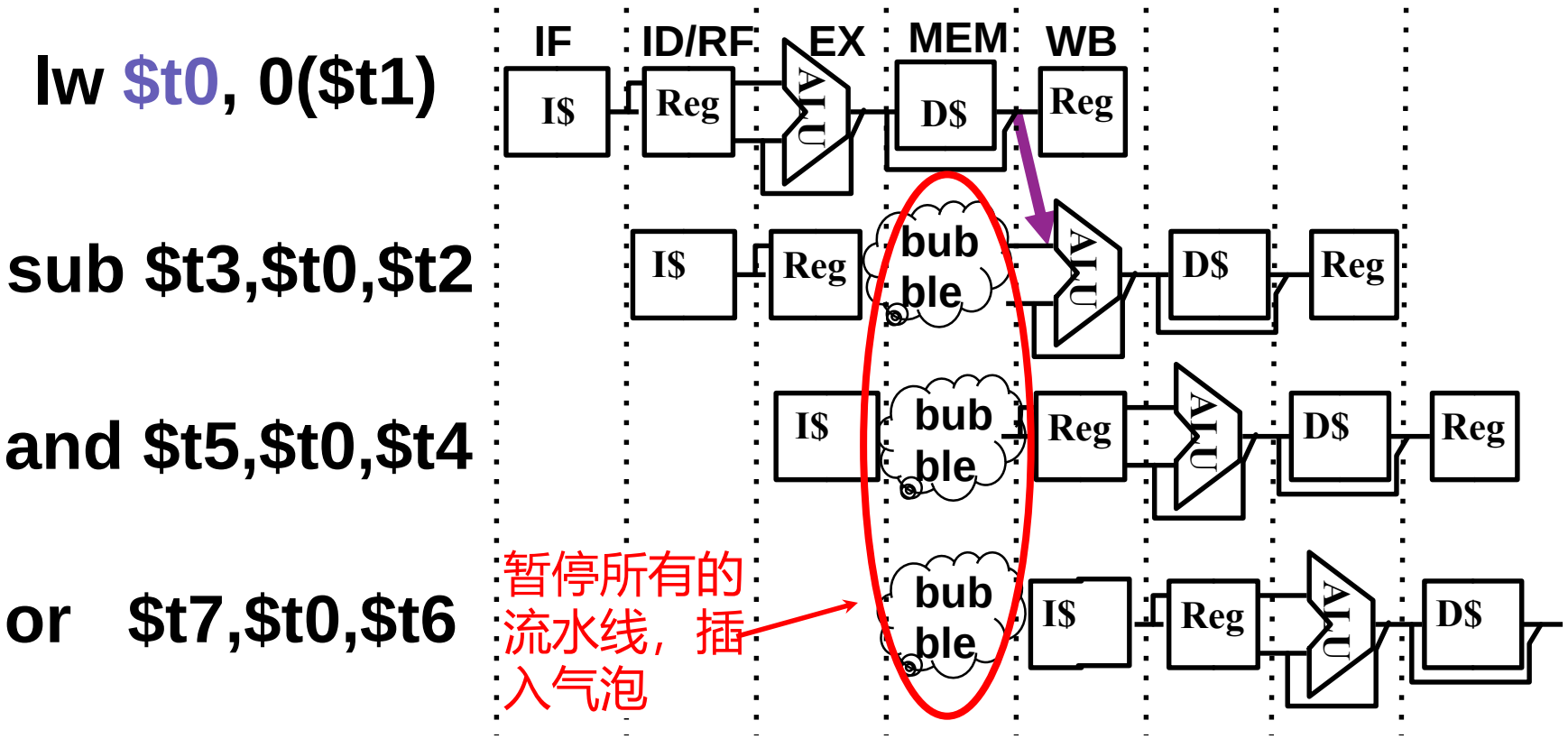
- 数据装入之后立即使用会产生数据冲突



- 不能通过数据旁路来解决所有的数据冲突
  - 此时必须暂停依赖于load的指令，等装入数据之后进行数据旁路（前传）

数据的装入使用冲突 (2/4)

- *硬件*暂停流水线



数据的装入使用冲突 (3/4)

- 流水线暂停等价于 nop指令

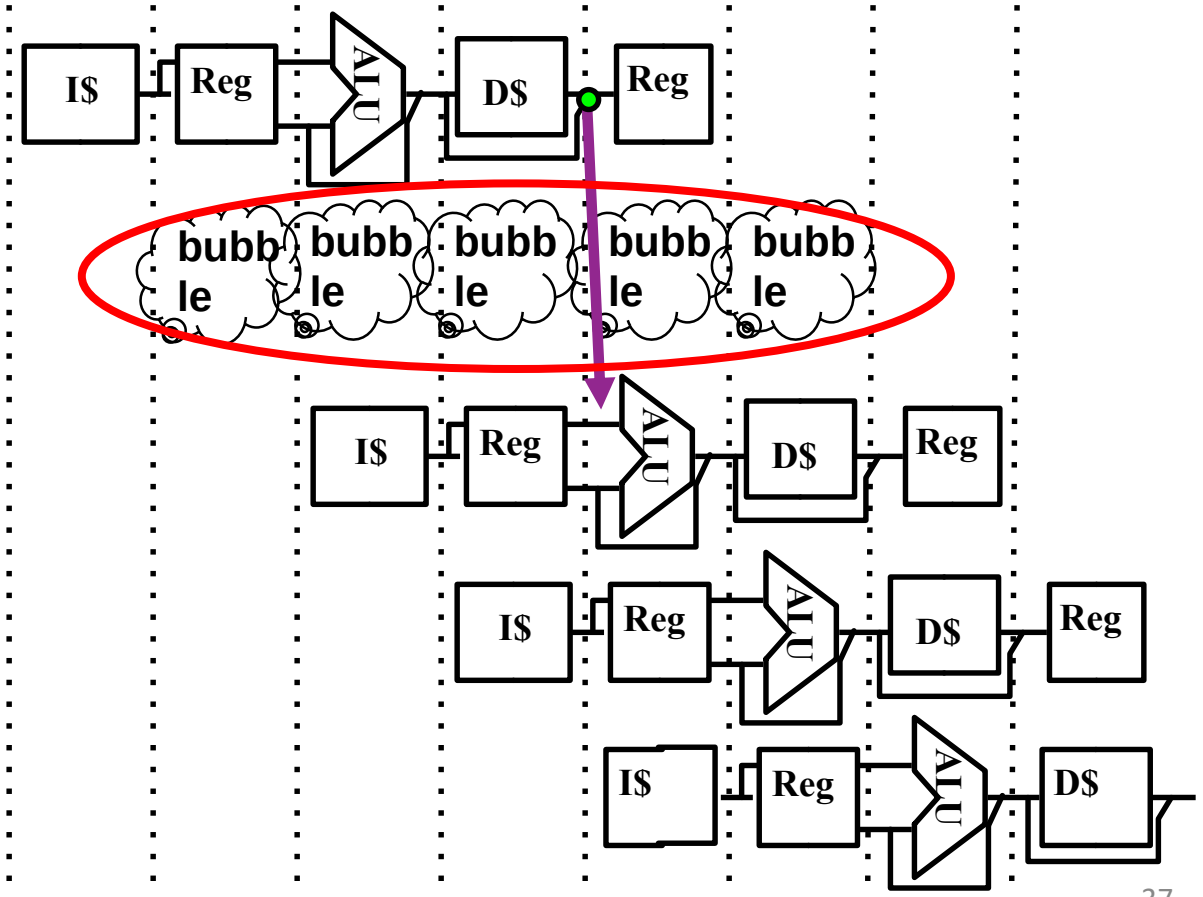
lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



检测

- 判断条件
  - 检测点：
    - 指令译码阶段
  - 检测条件
    - 上一指令是Load指令
    - 且它的写入寄存器和当前指令的某一源寄存器相同
    - isMemRead@EX AND
    - （AddrD@EX= AddrA@ID OR AddrD@EX= AddrB@ID)

暂停流水线

- 一旦发生此类冲突
  - 暂停流水线一个时钟
    - 让当前指令的控制信号全部为0，即不进行任何写入操作
    - 让PC值保持不变
    - 让IF/ID段寄存器保持不变
  - 将LW指令的结果通过旁路送到ALU输入端
    - Forwarding逻辑需要增加：???

暂停流水线（部分示例代码）

```
always_ff @(posedge clk or posedge rst) begin
    if (rst) begin
        id_pc <= '0;
        id_inst <= `NOP_INST;
    end else if (branch_flag | (stall_if && ~stall_id)) begin
        id_pc <= '0;
        id_inst <= `NOP_INST;
    end else if (~stall_if) begin
        id_pc <= if_pc;
        id_inst <= if_inst;
    end
end
```

数据的装入使用冲突 (4/4)

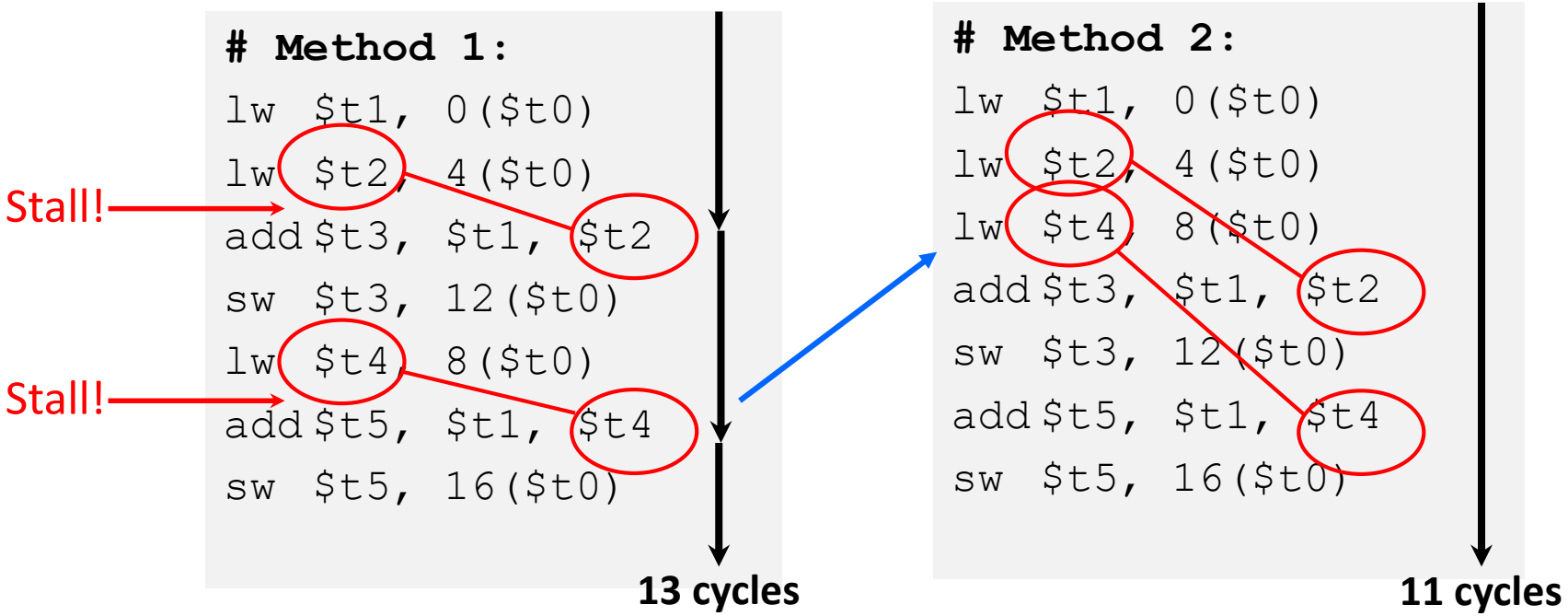
- 在数据装入之后的那个周期被称为是 *load delay slot*
  - 一些硬件的实现：
    - 可以检测到后一条使用装入的结果
    - 暂停流水线一个时钟周期
  - 等价于在周期中间插入一条 `nop` 指令

- **另一个解决办法**：让汇编器assemble或者程序员放一条不相关的指令，避免冲突 → 无需暂停!

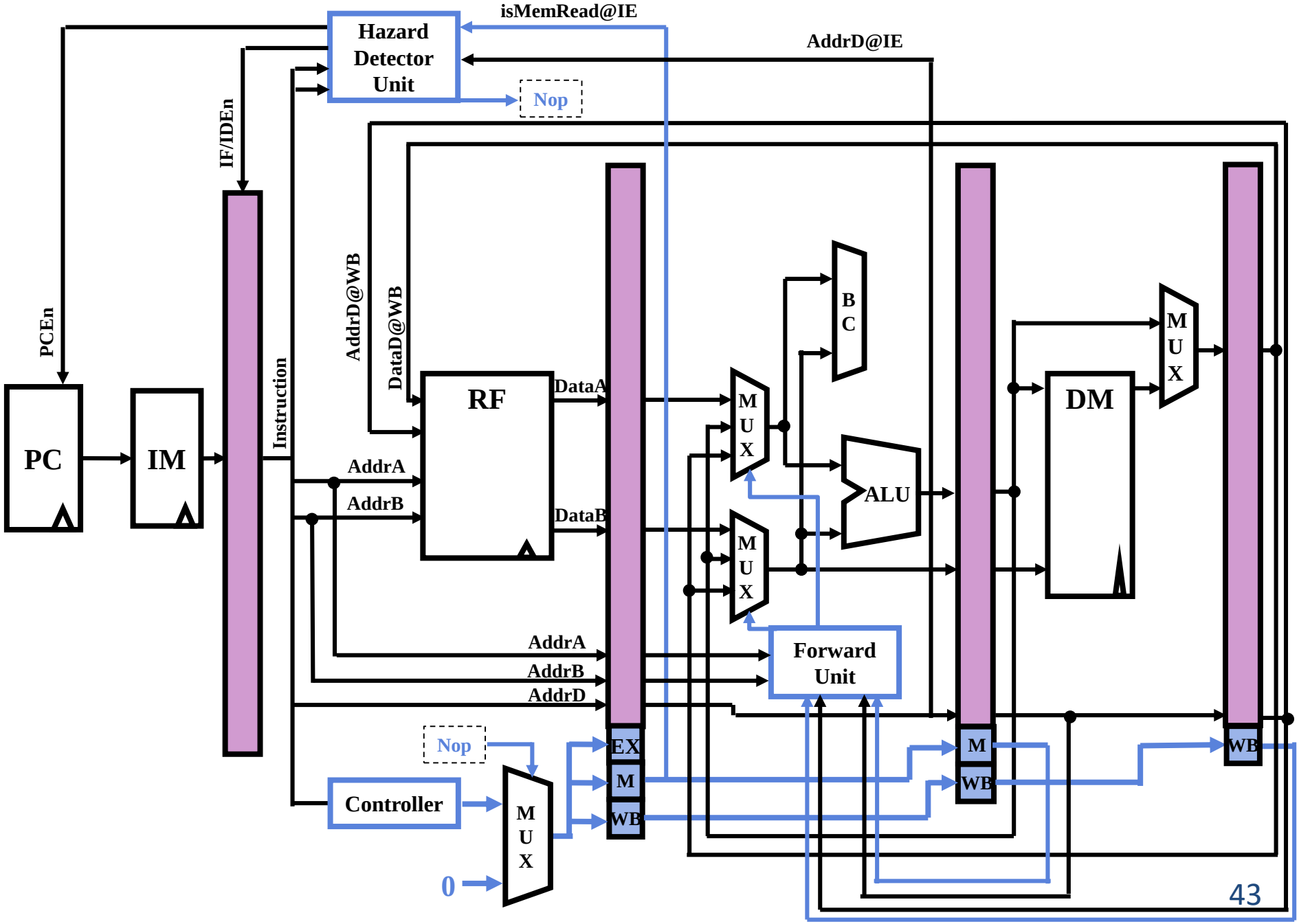
代码顺序调换来避免流水线暂停

Assembler Update!

- 通过汇编器避免下一条指令结果依赖装入指令!（汇编器调度，静态调度）
- 代码 `D=A+B; E=A+C;`



数据冲突处理通路


Hazard Detector Unit
isMemRead@IE
AddrD@IE
Nop
IF/IDEn
PCEn
PC
IM
Instruction
AddrD@WB
DataD@WB
RF
DataA
DataB
AddrA
AddrB
B
C
MUX
ALU
DM
Forward Unit
AddrA
AddrB
AddrD
EX
M
WB
Controller
0

数据冲突和相应解决方法

- 数据冲突的动态调度
  - 这种方法是由硬件动态调整指令执行顺序以减少暂停的影响，能够简化编译器设计。
  - 动态调度并不能真正消除数据冲突，但它能在出现数据冲突时尽量避免出现处理器暂停。而静态调度方法则是尽量通过分离有冲突问题的指令使它们不会导致冲突，从而减少暂停的影响。
  - 动态调度的主要思想：
    - 指令顺序发射——乱序执行——指令乱序流出
  - 动态调度的问题：异常处理的不精确性。在采用动态调度方法的处理机中，在某条指令产生异常情况时，有可能出现其后面的指令已经执行完成的情况，这样异常处理是不精确的。

小结

- 结构冲突
  - 资源发生冲突
  - 增加资源
  - 暂停流水线
- 数据冲突
  - 指令需要使用的操作数还没有保存到寄存器组中
    - 暂停流水线
    - 使用旁路技术
    - 静态调度（汇编器调度）
    - 动态调度（处理器调度）

阅读和思考

- 阅读

- 思考
  - 在THINPAD硬件组成上，如何发现各指令流水的冲突并进行避免？
- 实践
  - 根据ThinPAD RISC-V指令系统的要求，确定数据通路的组成以及各组成部件所需要完成的具体功能
  - 现有的实验硬件平台上，能否完全避免结构冲突？
  - 设计你们的CPU的旁路

谢谢