Chapter 4

The Processor

- 流水线CPU的设计

Outline

- **流水线CPU概述**
- 流水线CPU datapath与controller
- 流水线CPU 数据冒险的检测与处理
- 流水线CPU 控制冒险的检测与处理
- 中断与异常
- 多发射

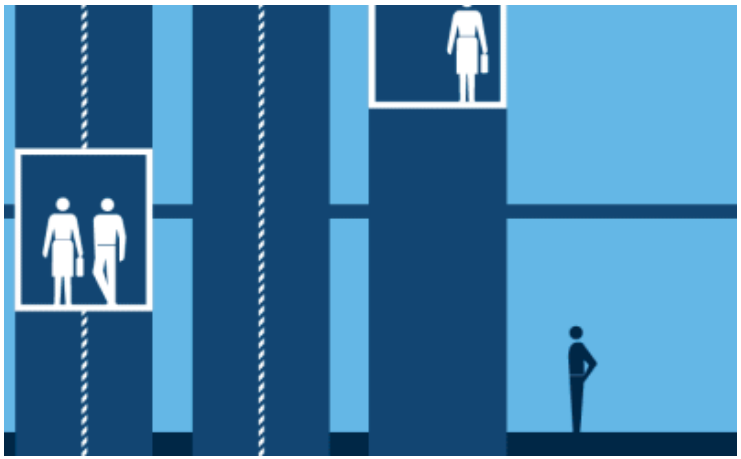
性能问题

- Longest delay 决定时钟周期
  - 关键路径: load指令指令，会使用所有5个阶段
  - Instruction memory → register file → ALU → data memory → register file
  - 不同指令使用不同的时钟，在物理实现上是不可行的
- 但是，不同延时的指令共用时钟周期，是一个次优的方案
  - 那些快速的指令会浪费时间去等待
  - 此时，CPU的某些阶段会空闲

- 如何改善性能——什么是性能？
  - 响应时间
    - 单个任务更快结束?
  - 吞吐量:
    - 单位时间内完成更多任务 (比如：Web服务器响应更多页面请求)?
  - 更长的电池续航?
- 提升吞吐量
  - By keeping datapath as busy as possible

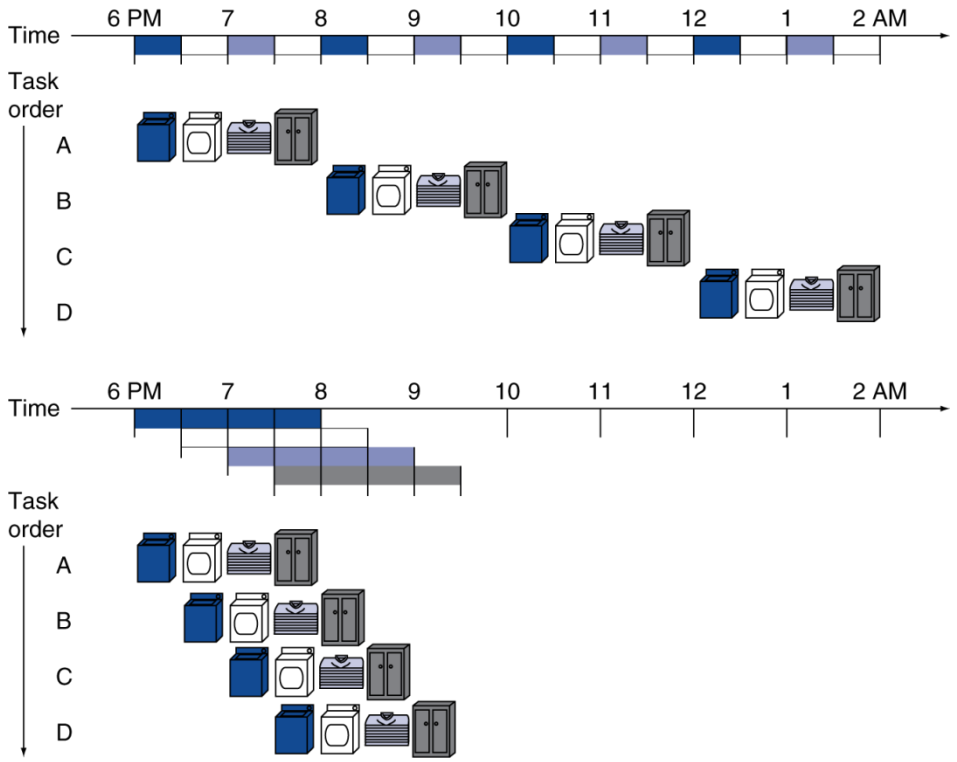
流水线并行——例子

- 为什么教学楼不使用垂直电梯?
  - 垂直电梯 （不考虑启动时间，假设6米高）
    - 假设：耗时4s，12人
    - 吞吐量：12p/3s = 4p/s
  - 扶手电梯（或楼梯）
    - 假设：耗时10s，40级台阶(15 cm)，每台阶2人，
    - 吞吐量：40*2p/10s= 8p/s（满流水）



流水线并行——例子

- 流水线洗衣机: 重叠运行
  - 通过并行提升性能



- 假设总共4个任务:
  - 加速比
    = 8/3.5 = 2.3
- 假设持续不停工作:
  - 加速比
    = 2n/0.5n + 1.5 ≈ 4
    = 流水线的阶段数

如何保持数据通路的忙碌?

Time											
Space		cycle	cycle	cycle	cycle	cycle	cycle	cycle	cycle	cycle	cycle
	IF stage	load IF					add IF				
	ID stage		load ID					add ID			
	EX stage			load EX					add EX		
	MEM stage				load MEM						
	WB stage					load WB					add WB

Time											
Space		cycle	cycle	cycle	cycle	cycle	cycle	cycle	cycle	cycle	cycle
	IF stage	load IF	add IF				add IF				
	ID stage		load ID	add ID				add ID			
	EX stage			load EX	add EX				add EX		
	MEM stage				load MEM						
	WB stage					load WB	add WB				add WB

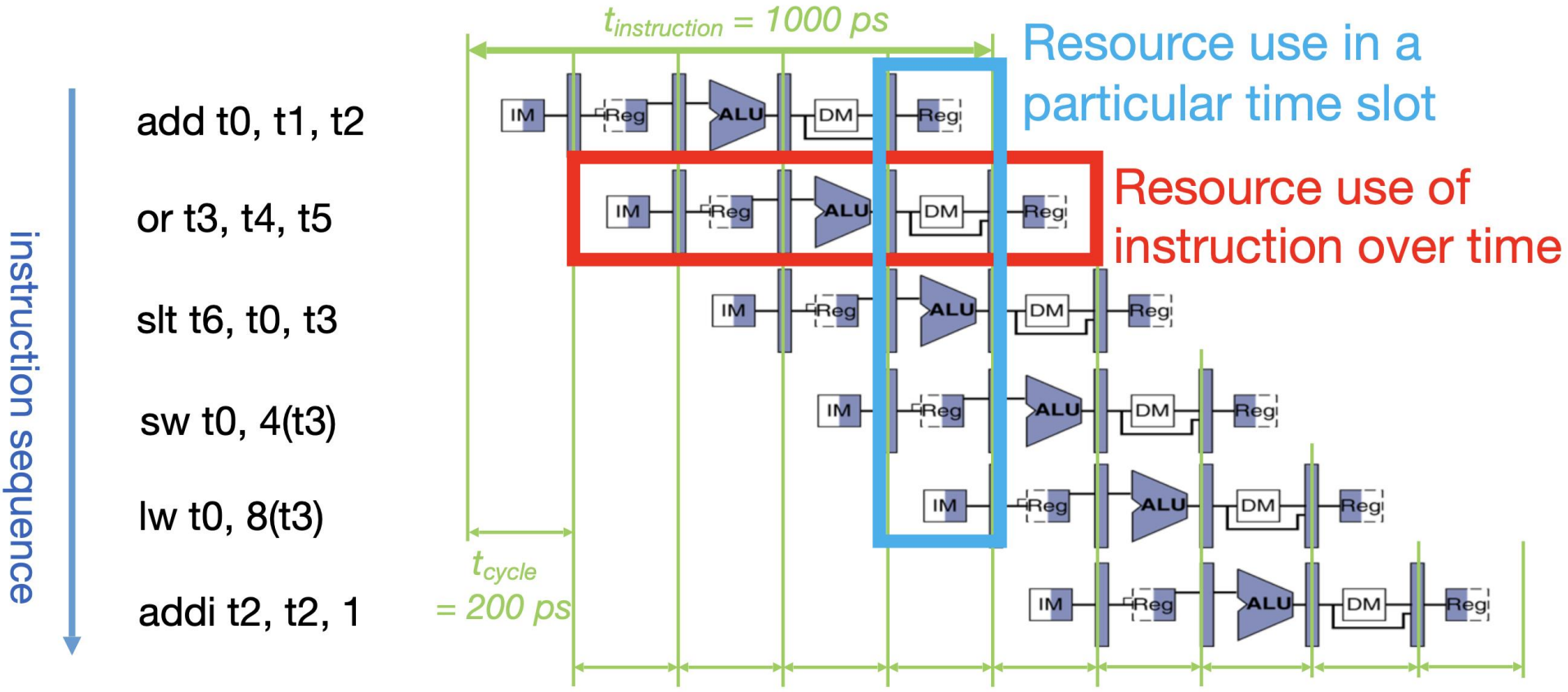
流水线并行

- 非流水线CPU：
  - 同一时刻每条指令独占整个数据通道
  - 每个时钟完成一条指令，但是时钟周期很长
- 流水线CPU
  - 将一个过程拆分成k个阶段
    - 同一时刻每条指令只占datapath的一部分
    - 同一时刻多条指令分别占用datapath的不同部分
  - 最长阶段决定时钟周期，即时钟频率
- 理想加速比？
  - k？

RISC-V 流水线

- 5阶段, 每阶段完成一个步骤
  1. IF: 从指令内存中读取指令
  2. ID: 指令的解码 & 读取寄存器的值
  3. EX: ALU完成计算（ld/sd计算地址）
  4. MEM: 访问数据内存（仅ld/sd指令）
  5. WB: 将结果写回寄存器

RISC-V 流水线

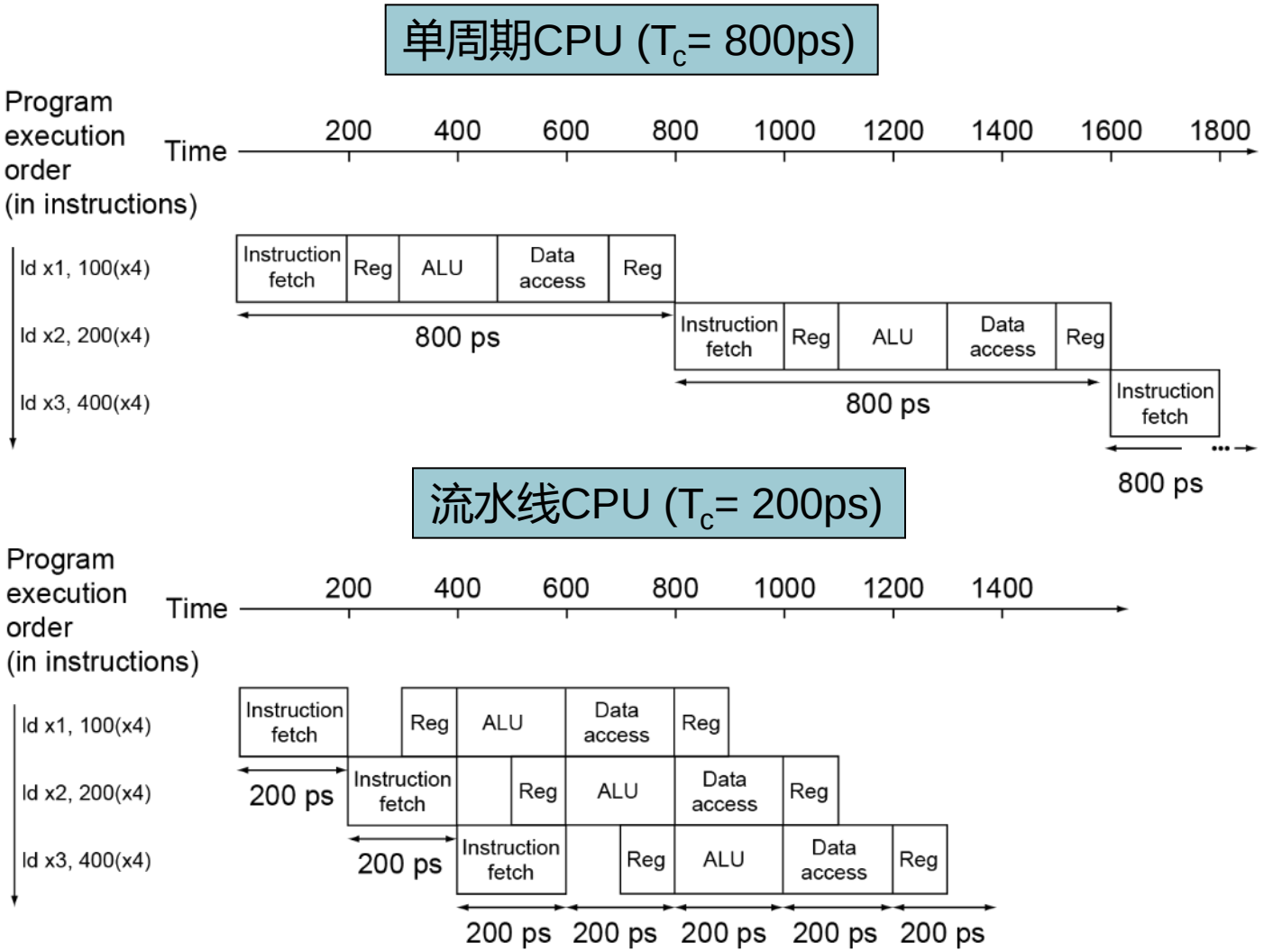

add t0, t1, t2
or t3, t4, t5
slt t6, t0, t3
sw t0, 4(t3)
lw t0, 8(t3)
addi t2, t2, 1
instruction sequence
$t_{instruction}$ = 1000 ps
t_{cycle} = 200 ps
IM
Reg
ALU
DM
Reg
Resource use in a particular time slot
Resource use of instruction over time

流水线CPU的性能

- 假设各阶段的时间是：
  - 寄存器的读写为：100ps
  - 其余阶段：200ps
- 比较流水线CPU和单周期CPU的性能

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

流水线CPU的性能


单周期CPU (T_c= 800ps)
Program execution order (in instructions)
Time
200 400 600 800 1000 1200 1400 1600 1800
ld x1, 100(x4)
ld x2, 200(x4)
ld x3, 400(x4)
Instruction fetch
Reg
ALU
Data access
Reg
800 ps
流水线CPU (T_c= 200ps)
200 ps

流水线CPU的加速比

- 如果所有阶段是均衡的
  - 即, 每阶段的速度一样
  - 两条指令的间隔$_{pipelined}$
    = $\frac{\text{两条指令的间隔}_{nonpipelined}}{\text{阶段的数目}}$
- 如果阶段是不平衡的，加速比会下降
  - 此外, 流水线需要额外的开销
- 加速比源自吞吐量的提升
  - 单条指令的延时并没有下降, 反而通常会上升

ISA设计对流水线的影响

- RISC-V ISA 为流水线而设计
  - 所有指令都是32位宽
    - 更容易在一个周期内完成取指或者解码
    - 反例：x86中的指令宽度为1至17字节
  - 更少、更规整的指令格式
    - 可以在同一个周期完成解码和读取寄存器
  - 专门的访存指令——load/store
    - 可以在第3阶段计算地址，第4阶段访问内存
    - 反例：x86指令[Mem] + R3 -> R4
      - 依次需要Ex（地址计算）、Mem、Ex（算术运算），导致流水线变长

流水线冒险（妨碍流水线的因素）

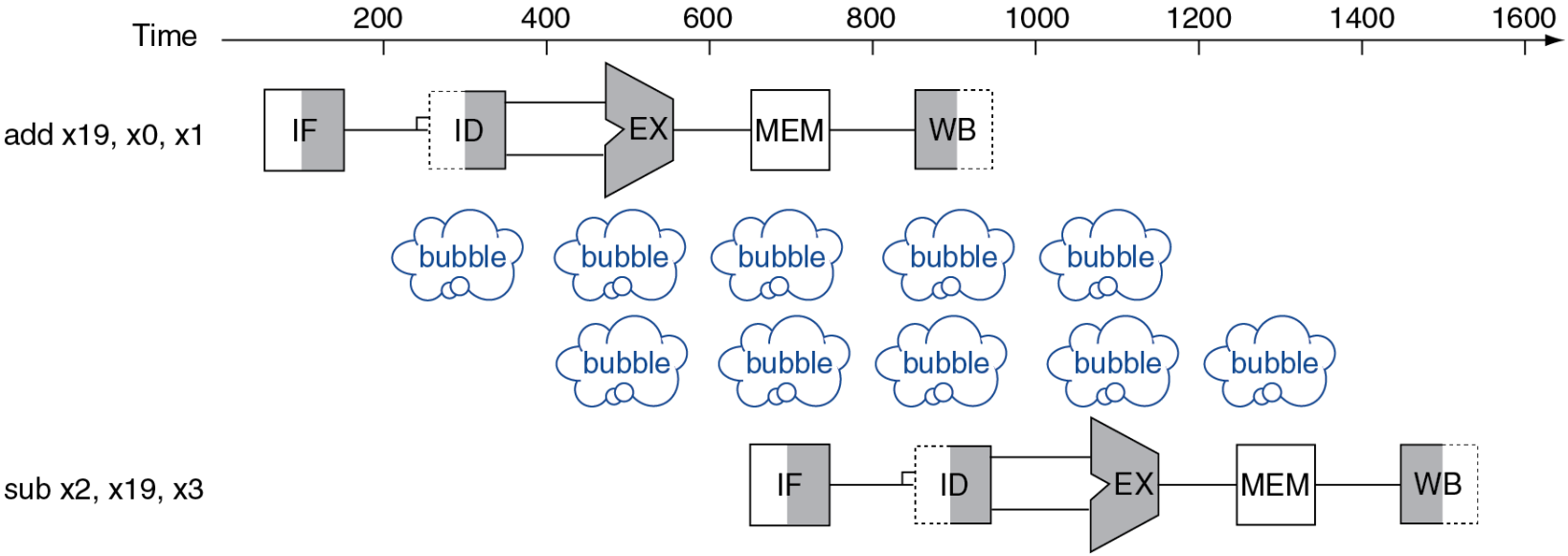
- 妨碍在下个周期连续开始下条指令的情形
- 结构冒险——物理资源竞争
  - 需要等待物理资源空闲
  - 比如，前面的指令返过来使用后面的硬件资源
- 数据冒险——数据依赖
  - 需要等待前面的指令完成读/写操作
- 控制冒险——控制依赖
  - 需要等待前面指令的运算结果，才能决定后续执行哪条指令

结构冒险

- 资源使用的冲突
- 如果RISC-V只有一个内存
  - Load/store指令需要访问内存（第1条指令）
  - 后面指令需要停顿后，才能取指令（第4条指令）
    - 因为内存不支持同时多个访问
    - 等待会导致流水中的“气泡”
- 因此，流水线CPU的通路需要分开的指令/数据内存
  - 或者分开的 指令/数据 缓存

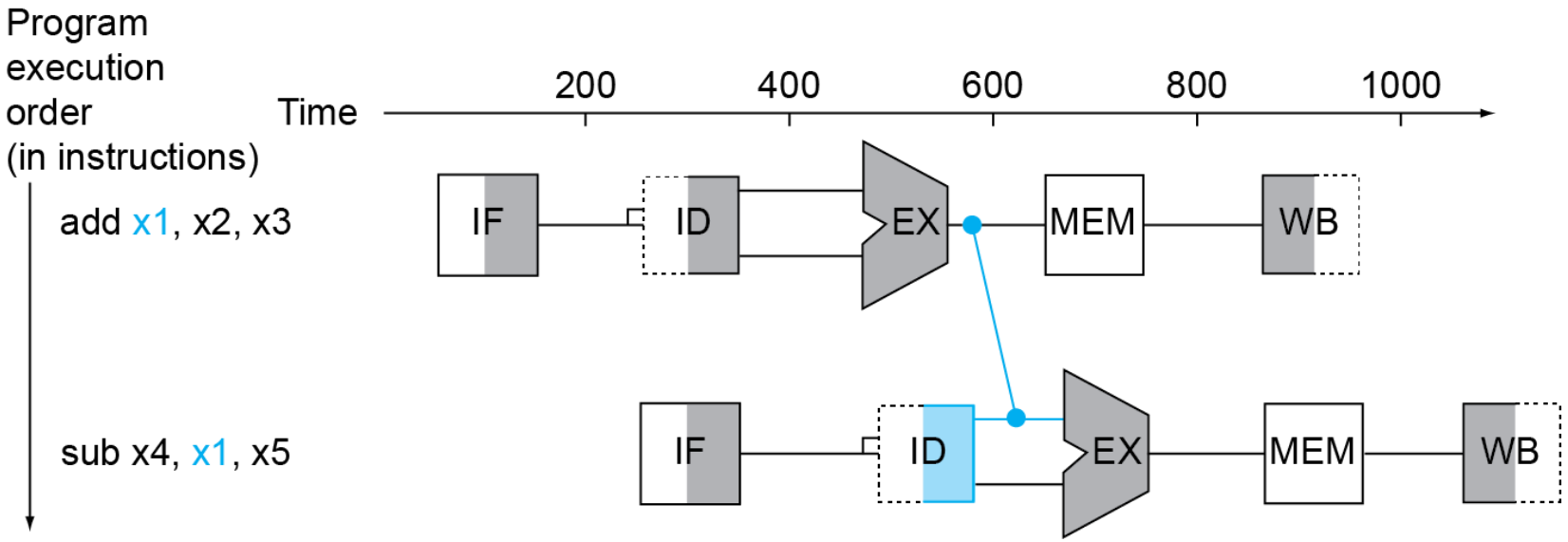
数据冒险

- 一条指令的数据，依赖于前面指令计算的值
  - add x19, x0, x1
    sub x2, x19, x3



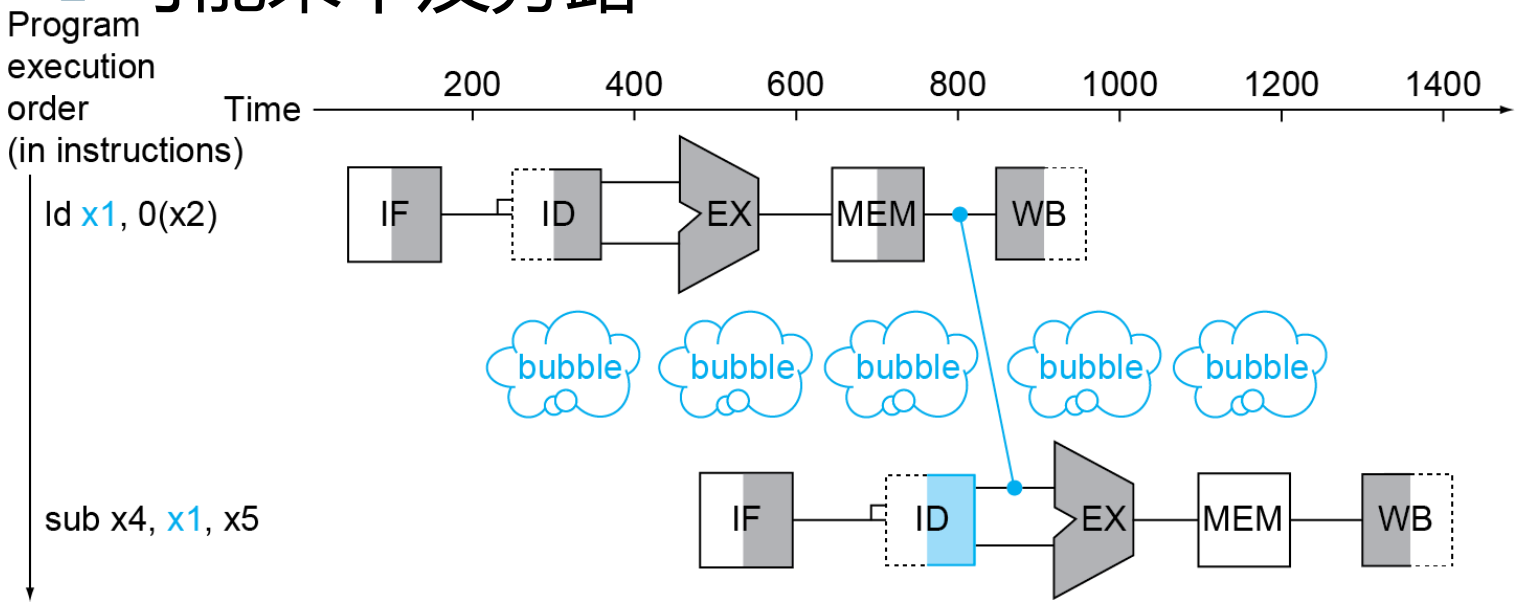
数据冒险的解决方法1——旁路/转发（图4.31）

- 前面指令一旦计算出值，就直接使用
  - 不用等待回写到寄存器
  - 需要数据通路上增加额外的连线
  - 连线叫做旁路（或转发）forwarding



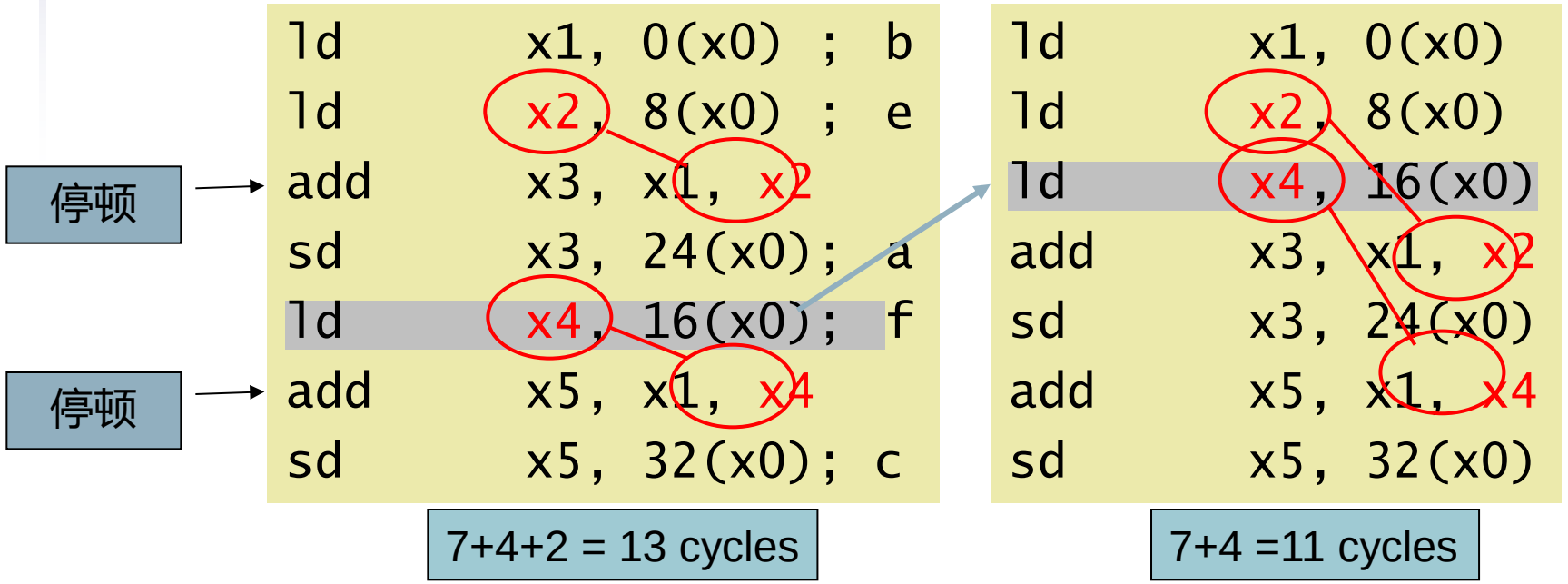
特殊的数据冒险：load-use（图4.32）

- 旁路也不能避免停顿
  - 后面指令需要值时，前面指令还没完成计算，只能停顿
  - 可能来不及旁路



解决方法2——通过代码调度来避免停顿

- 通过重排代码，避免在load指令后面的指令立即use
- C代码例子：a = b + e; c = b + f;



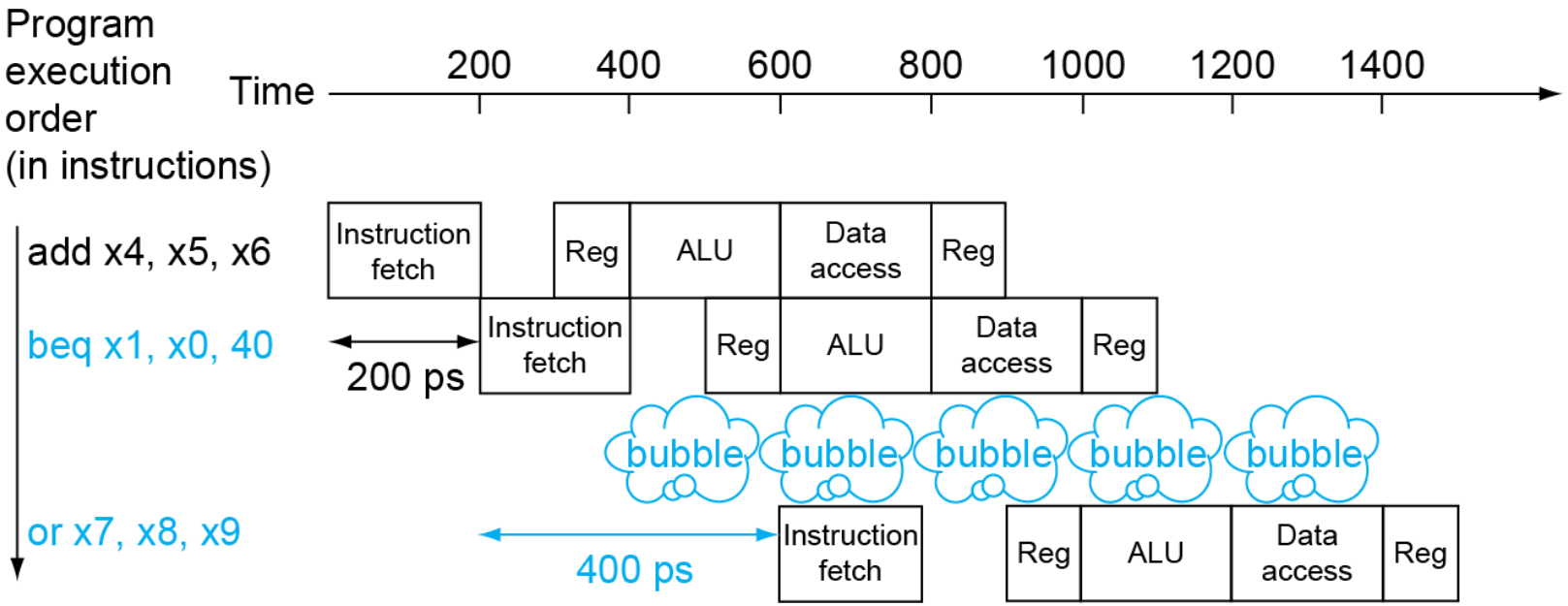
控制冒险

- 分支指令决定了控制流
  - 下一条指令取决于分支结果
  - 流水线取指令时，等不及前面分支指令的结果
    - 前面分支指令可能仍在ID阶段

- 在RISC-V 流水线中，一种优化是
  - 在流水线的早期阶段比较寄存器并计算出分支的目标地址
  - 增加硬件在ID阶段计算分支结果

分支导致的停顿（图4.33）

- 下一条指令的IF阶段必须停顿等待，直到分支结果计算完成

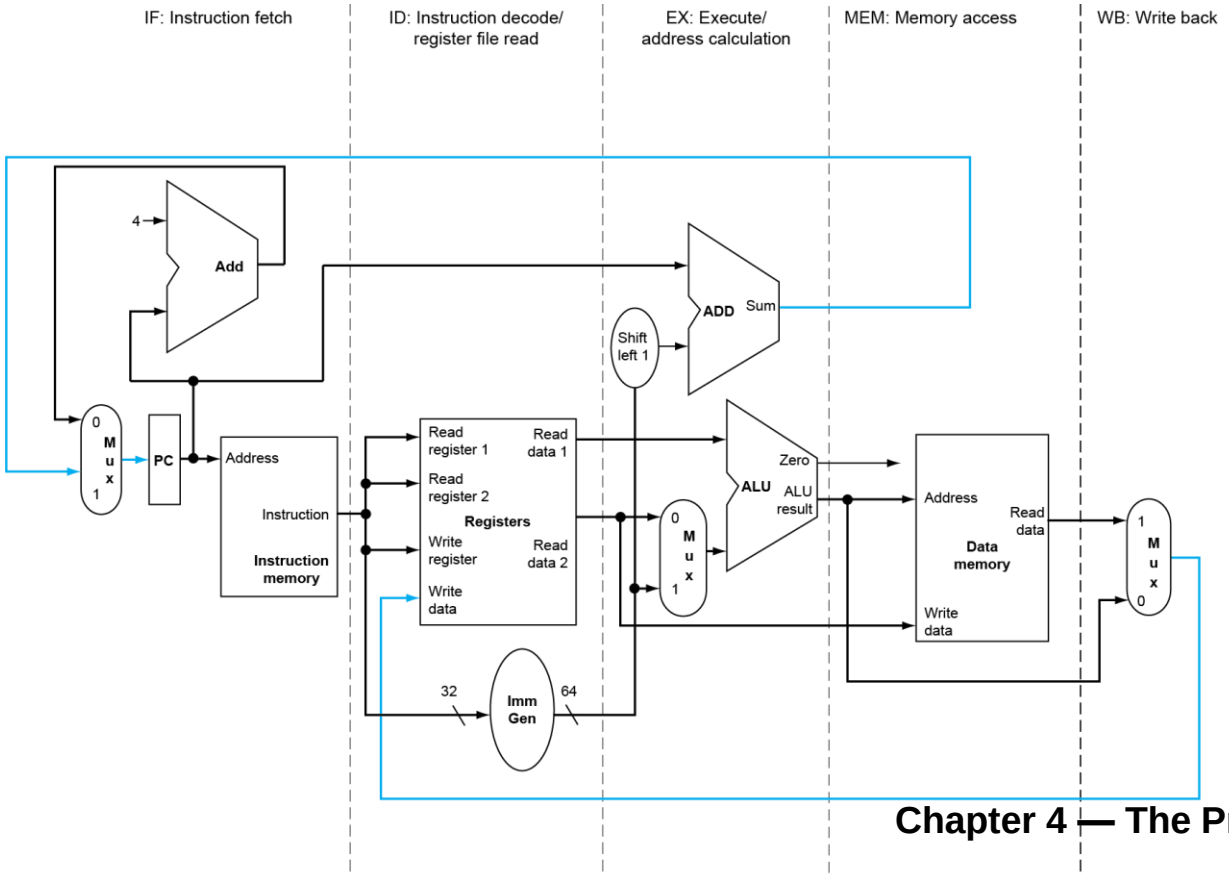


见教材图4.29

此处为**临时的、新增的**假设：增加了额外硬件，以便在ID阶段就可以更新PC

如果在EX阶段更新PC，则需要停顿2个周期

- 事实上，在ID阶段获得分支结果，设计上有挑战
  - https://www.cs.umd.edu/~meesh/411/CA-online/chapter/handling-control-hazards/index.html
- 本书中通常在MEM阶段完成



分支预测

- 对于长流水线CPU来说，很难在早期确定分支结果
  - 会导致巨大的停顿代价
- 可以预测分支的结果
  - 只有预测错误的时候，才停顿
- 在RISC-V流水线中
  - 可以预测分支为：不跳转（ not taken）
  - 立即取指分支指令的下一条指令

循环语句的编译

- C 代码:

```
while (save[i] == k) i += 1;
```

  - i in x22, k in x24, save数组的地址 in x25
- 对应的RISC-V汇编指令:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit   (预测不跳转?)
       addi x22, x22, 1
      beq  x0, x0, Loop    (预测跳转?)
Exit: …
```

更实际的分支预测器

- 静态分支预测
  - 基于典型的分支行为
  - 比如: 循环和条件分支指令
    - 后向分支：预测为跳转
    - 前向分支：预测为不跳转
- 动态分支预测
  - 用硬件测量实际的分支行为
    - 即，记录每个分支的近期是否跳转的历史
  - 假设历史可以预测未来
    - 如果预测错误，则：停顿，重新取新的目标指令，更新历史

流水线总结

The BIG Picture

- 流水线通过提升指令吞吐量来提升性能
  - 并行执行多条指令
  - 每条指令需要同样的延时
- 流水线会被冒险妨碍
  - 结构、数据、控制冒险
- 指令集设计会影响流水线实现的复杂度

Exercise 1:

请回答下列问题。

(1)若 int 型变量 x 的值为-513,存放在寄存器 R1 中,则执行指令"SHR R1"后,R1 的内容是多少?(用十六进制表示)

(2)若某个时间段中,有连续的 4 条指令进入流水线,在其执行过程中没有发生任何阻塞,则执行这 4 条指令所需的时钟周期数为多少?

(3)若高级语言程序中某赋值语句为 x=a+b,x、a 和 b 均为 int 型变量,它们的存储单元地址分别表示为[x]、[a]和[b]。该语句对应的指令序列及其在指令流水线中的执行过程如下图所示。

I_1 LOAD R1,[a]

I_2 LOAD R2,[b]

I_3 ADD R1,R2

I_4 STORE R2,[x]

指令	时间单元													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I_1	IF	ID	EX	M	WB									
I_2		IF	ID	EX	M	WB								
I_3			IF				ID	EX	M	WB				
I_4							IF				ID	EX	M	WB

则这 4 条指令执行过程中,I_3的 ID 段和 I_4的 IF 段被阻塞的原因各是什么?

(4)若高级语言程序中某赋值语句为 x=x*2+a,x 和 a 均为 unsigned int 类型变量,它们的存储单元地址分别表示为[x]、[a],则执行这条语句至少需要多少个时钟周期?要求模仿上图画出这条语句对应的指令序列及其在流水线中的执行过程示意图。