计算机组织与系统结构

设计流水线处理器

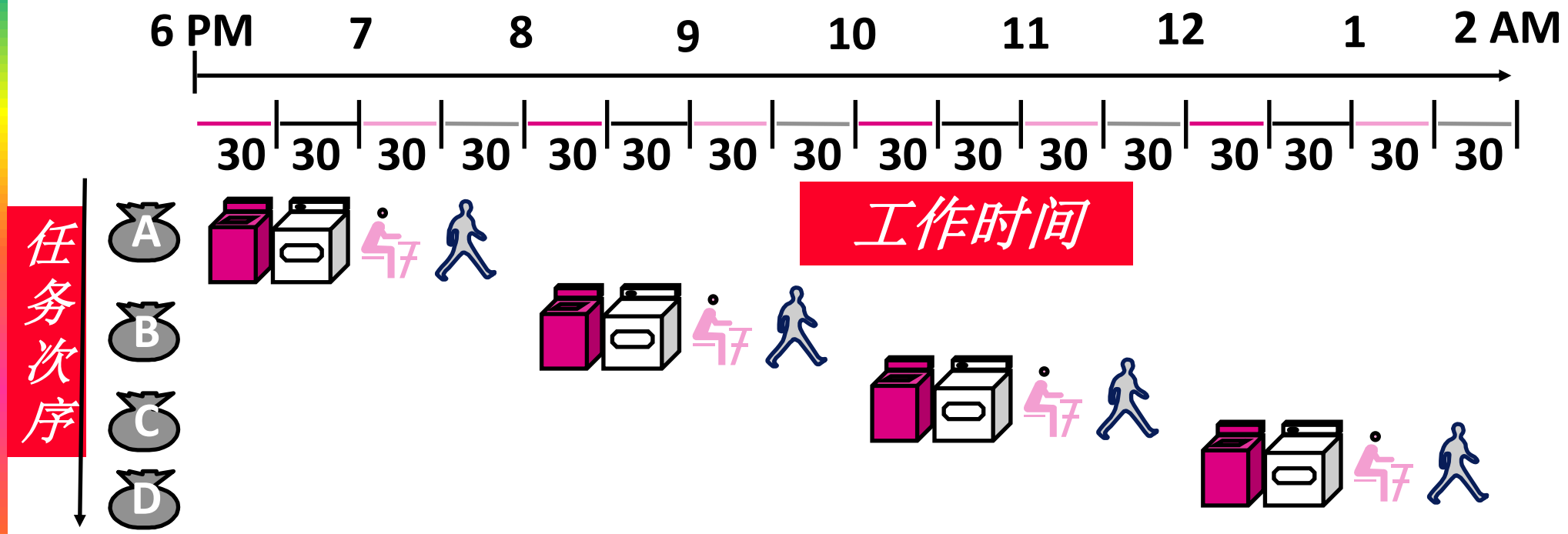
Designing a Pipeline Processor

(第十二讲)

程 旭

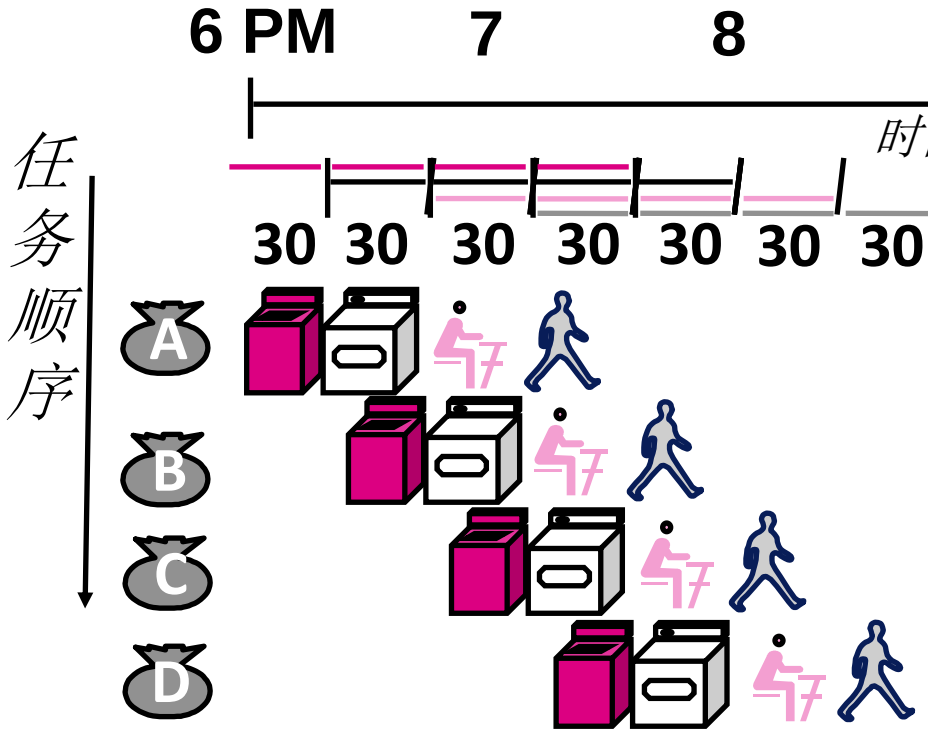
2020.12.10

串行洗衣店



- 串行洗衣店需要8个小时完成4个工作量
- 如果他们了解流水技术，那么需要多长时间完成上述工作呢？

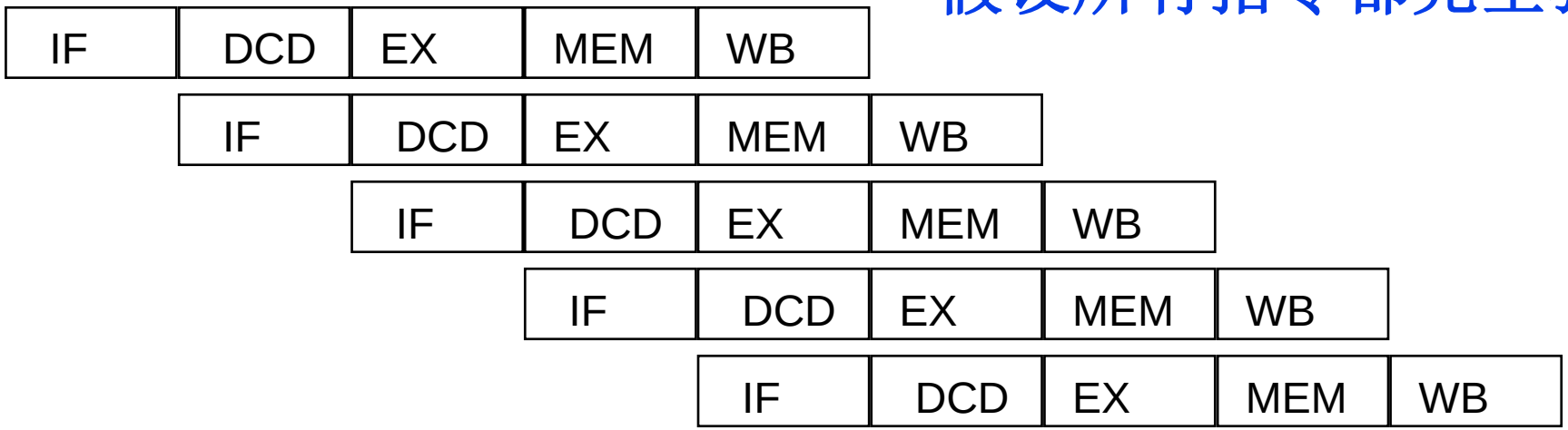
流水技术性质



- 流水技术无助于减少单个任务的 处理延迟(latency)，但有助于提高整体工作负载 的吞吐率
- 多个不同任务同时操作, 使用不同资源
- 潜在加速比 = 流水线级数
- 流水线的速率受限于最慢的流水段
- 流水段的执行时间如果不均衡, 那么加速比就会降低
- 开始填充流水线的时间 和 最后排放流水线的时间 降低加速比
- 相关将导致流水线暂停

理想流水线

假设所有指令都完全独立!



最大加速比 ≤ 流水线段数

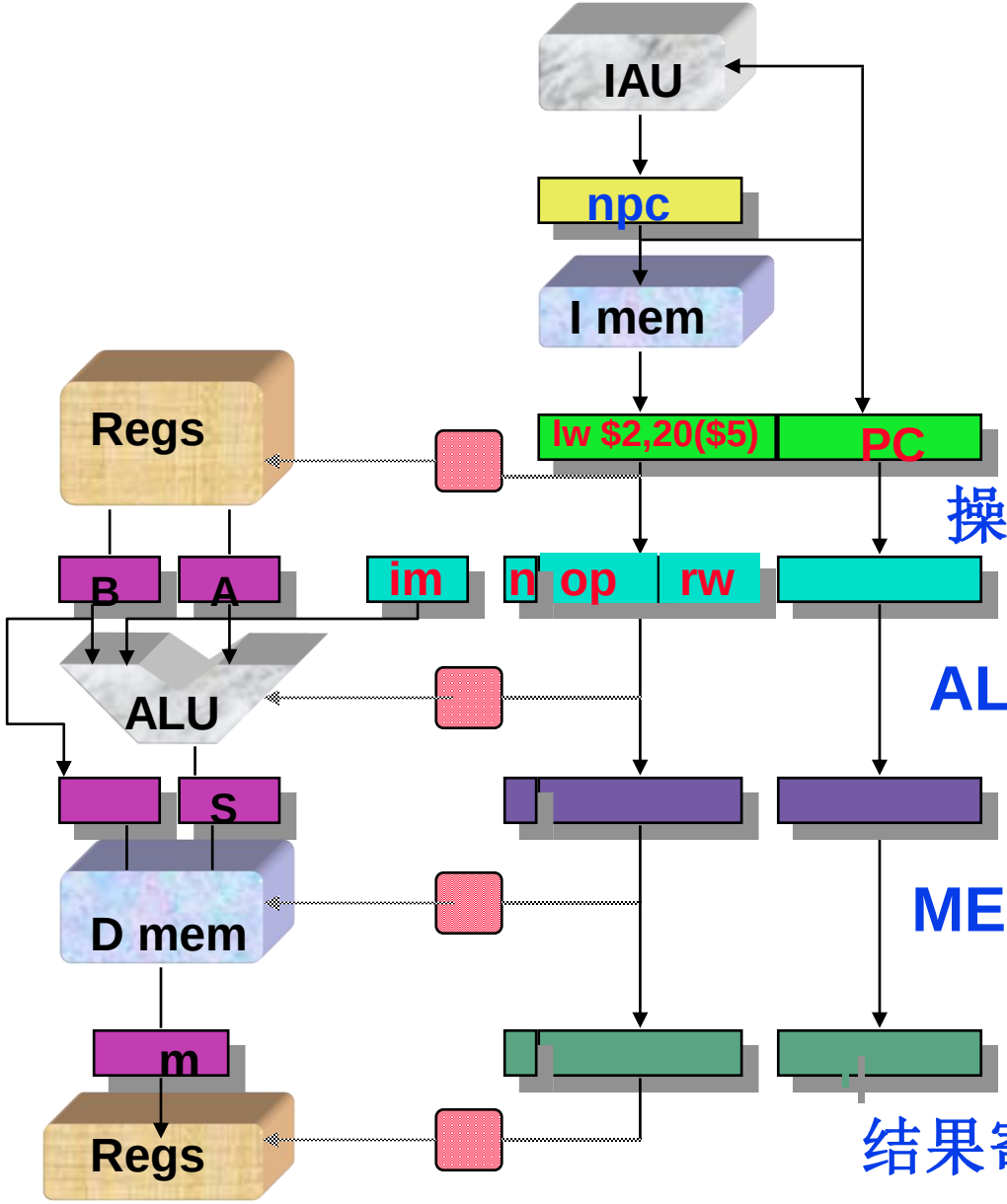
$$\text{加速比} \le \frac{\text{非流水化操作的时间}}{\text{最长段的时间}}$$

例如: 40ns数据通路, 5段, 最长段为10 ns, 加速比 ≤ 4

流水技术会产生哪些问题?

- 流水线冒险(Pipeline Hazards)
  - 结构冒险(structural hazards)：试图同时以两种不同的方式使用同一资源
    - 例如，多次存储器访问、多次寄存器写
    - 解决方案：多个存储器、暂停
  - 数据冒险(data hazards)：在产生数据之前，就试图使用它们
    - 例如， **add r1,r2,r3; sub r4, r1 ,r5; lw r6, 0(r7); or r8, r6 ,r9**
    - 解决方案： 前递/旁路、暂停/气泡
  - 控制冒险(control hazards):在判定转移条件之前，就试图决策转移方向
    - 例如，条件转移
    - 解决方案：预测、延迟转移

具有数据-固定控制的流水化数据通路



就象 时间-状态!

操作数寄存器选择

ALU Op ⇐ *PC + 4 + immed*

MEM Op

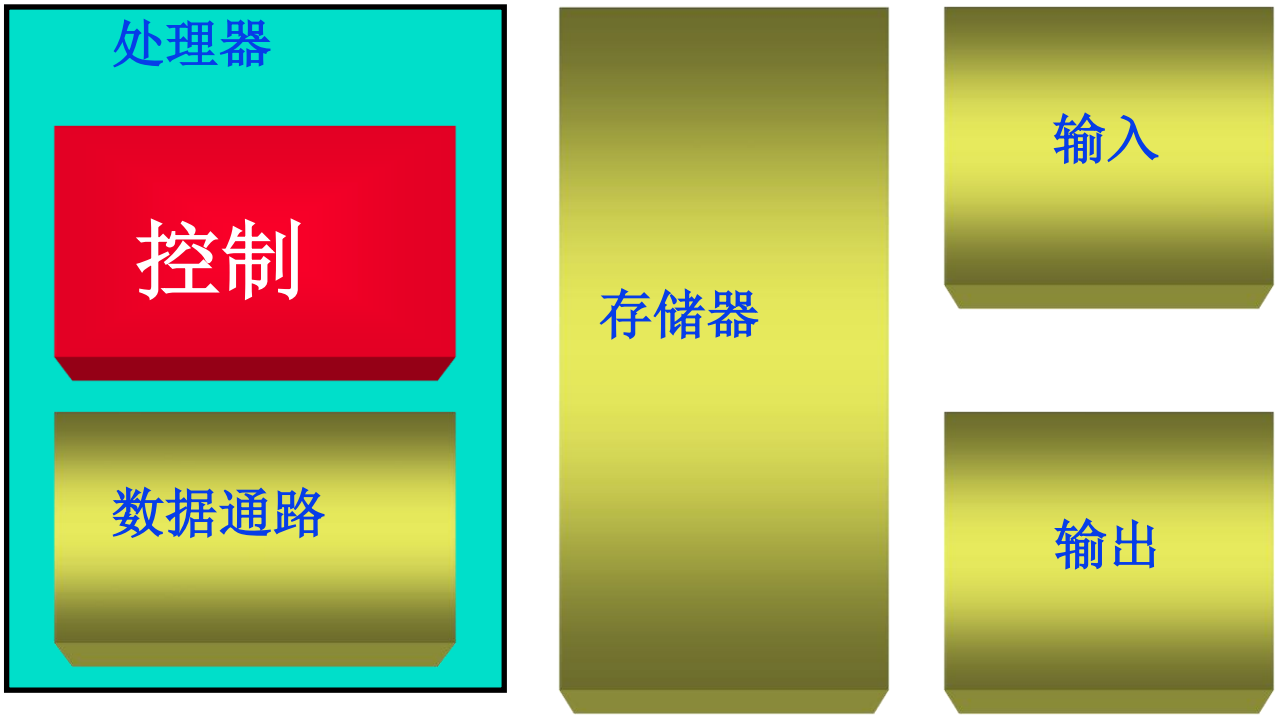
结果寄存器选择和使能

总结

- 流水处理是非常基本的概念
  - 使用不同资源的多个处理步骤
- 通过流水化指令处理利用数据通路的处理能力
  - 在处理当前指令的同时，开始处理下一指令
  - 受最长段的时间限制
  - 需要检测和处理冒险
- 哪些策略使得流水容易实现?
  - 所有的指令长度相同
  - 只有少量的指令格式
  - **Ld/St结构**
- 相关冒险问题是难点

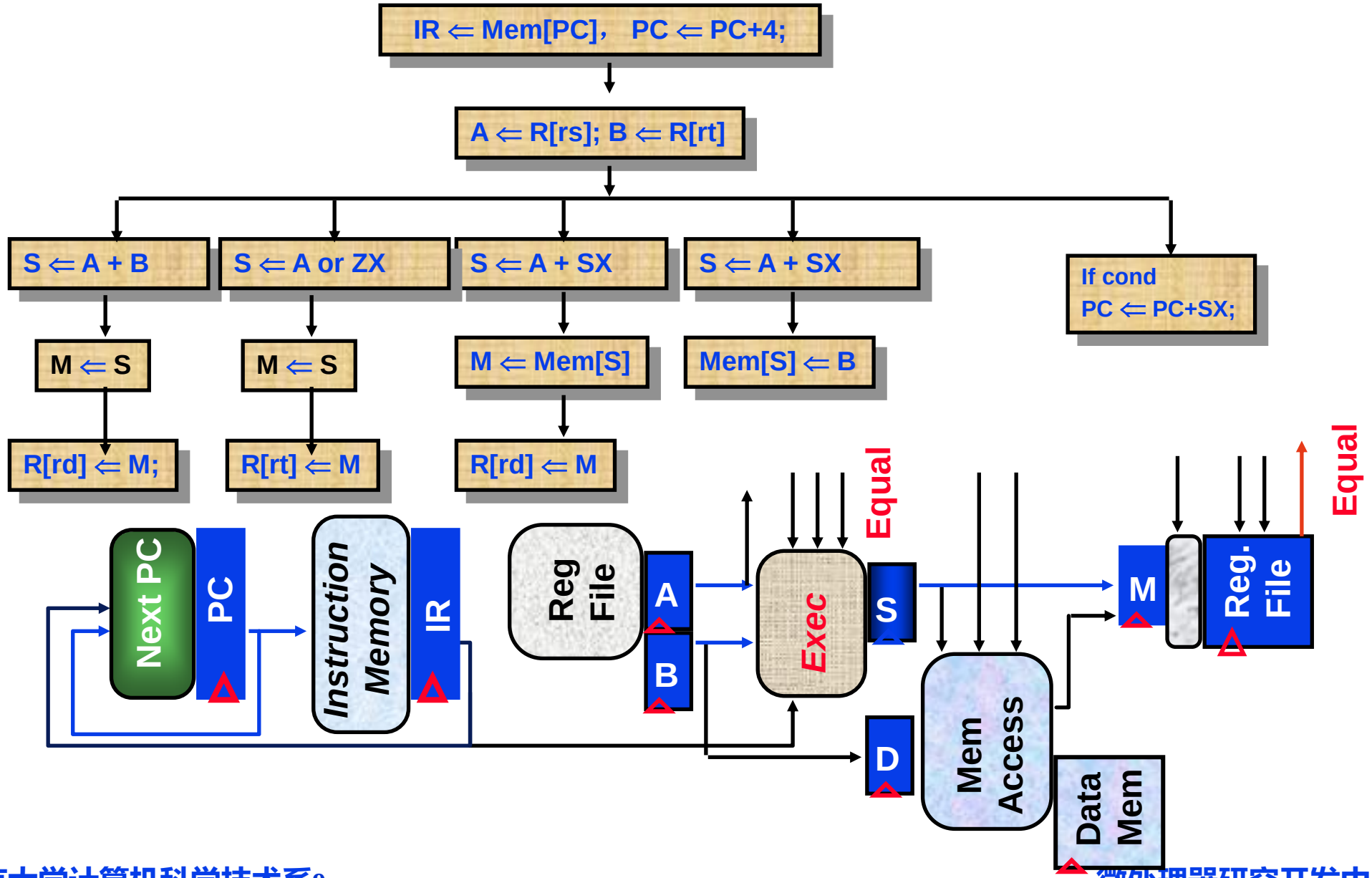
教学目标：已经掌握的内容

- 计算机的五个基本部件



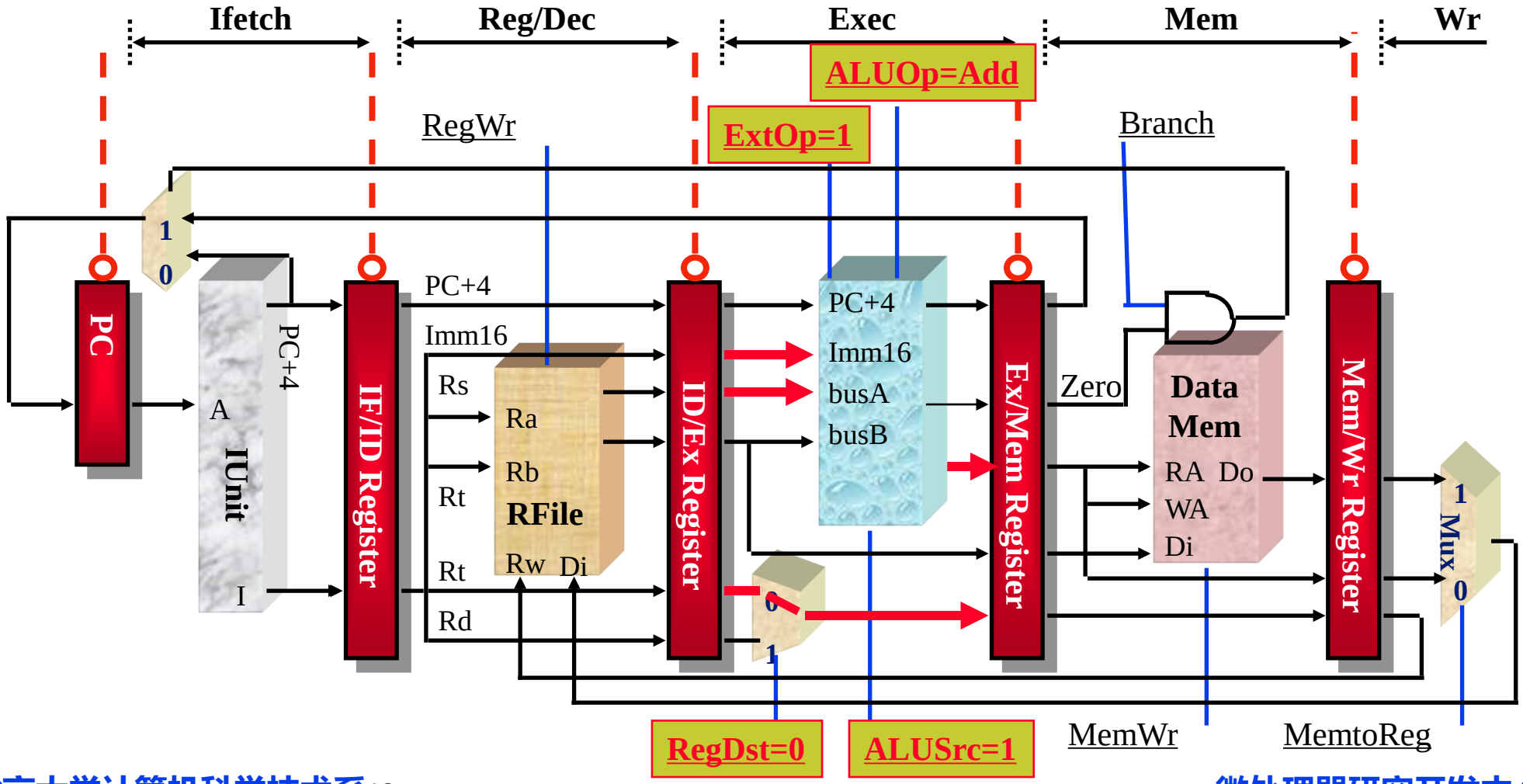
- 本讲主题:为流水化数据通路设计控制

控制图


IR ⇐ Mem[PC]， PC ⇐ PC+4;
A ⇐ R[rs]; B ⇐ R[rt]
S ⇐ A + B
S ⇐ A or ZX
S ⇐ A + SX
S ⇐ A + SX
If cond
PC ⇐ PC+SX;
M ⇐ S
M ⇐ S
M ⇐ Mem[S]
Mem[S] ⇐ B
R[rd] ⇐ M;
R[rt] ⇐ M
R[rd] ⇐ M
Next PC
PC
Instruction Memory
IR
Reg File
A
B
Exec
S
Equal
D
Mem Access
M
Reg. File
Equal
Data Mem

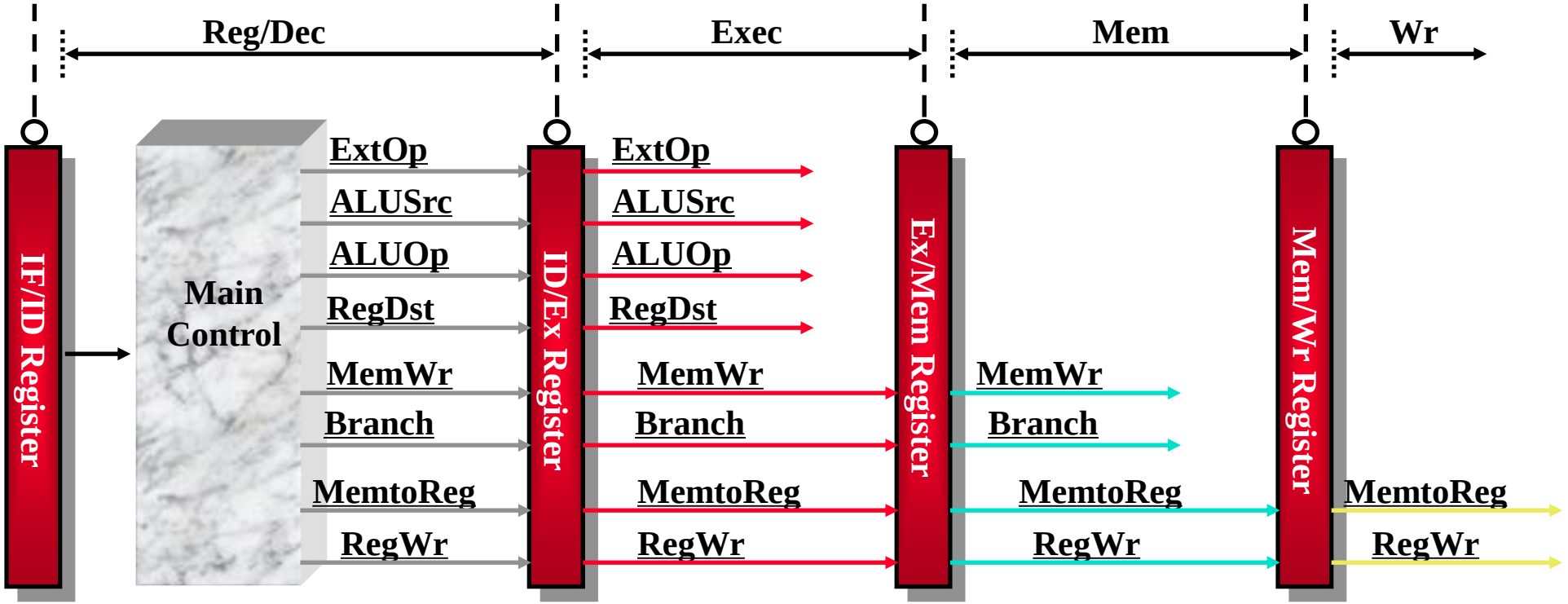
控制信号

- 重要发现：N段的控制信号 = Func (N段的指令)
  - N = Exec、Mem或 Wr
- 示例: Exec段的控制信号 = Func(Load的Exec)

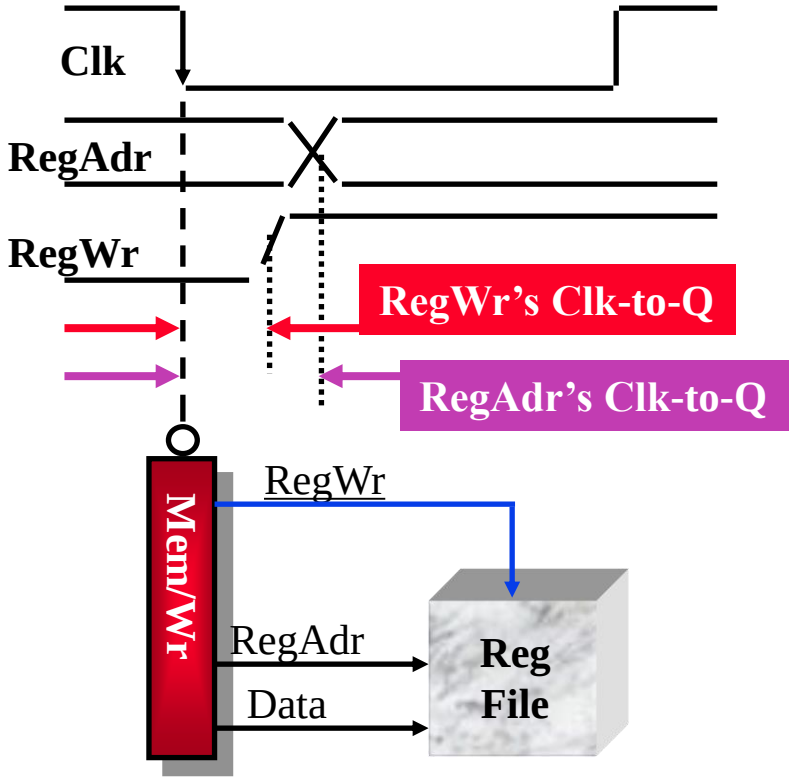


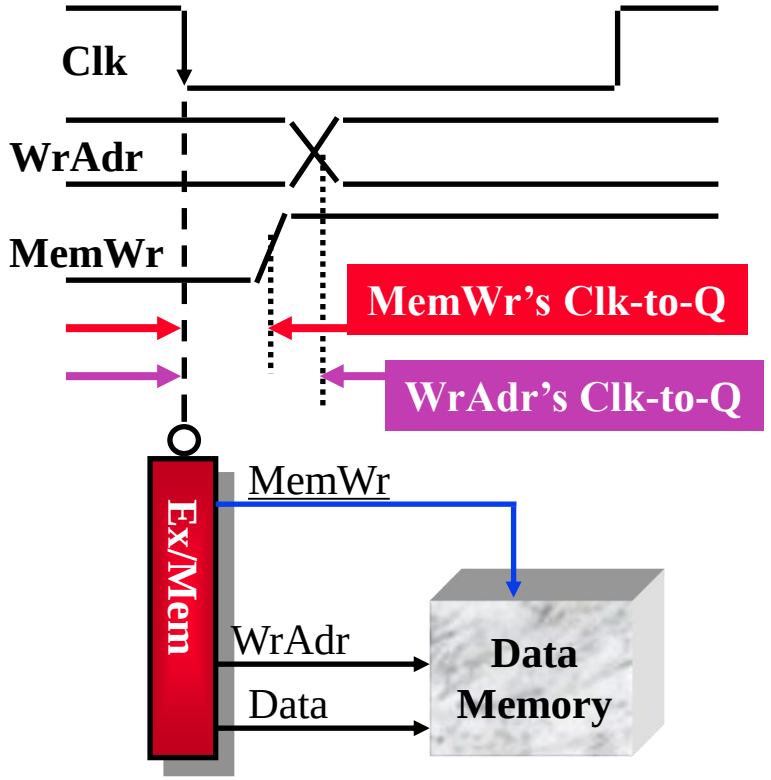
流水线控制

- 在Reg/Dec段，主控产生控制信号
  - Exec (ExtOp, ALUSrc, ...)的控制信号在1个周期后使用
  - Mem (MemWr Branch)的控制信号在2个周期后使用
  - Wr (MemtoReg MemWr)的控制信号在3个周期后使用



回写段的开始：现实中的问题

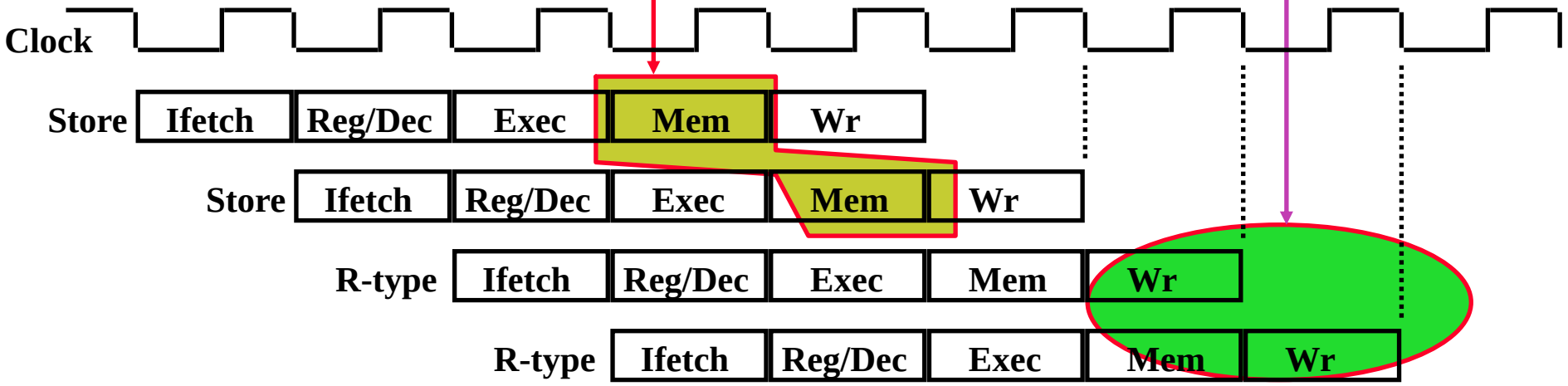




- 在Wr段的开始, 如果下式成立，现实中就会出现问题:
  - RegAdr's (Rd or Rt) Clk-to-Q > RegWr's Clk-to-Q
  - 在Address 和 Write Enable之间出现了竞争!
- 同样, 在Mem段的开始,如果下式成立，现实中就会出现问题:
  - WrAdr's Clk-to-Q > MemWr's Clk-to-Q
  - 在Address 和 Write Enable之间出现了竞争!

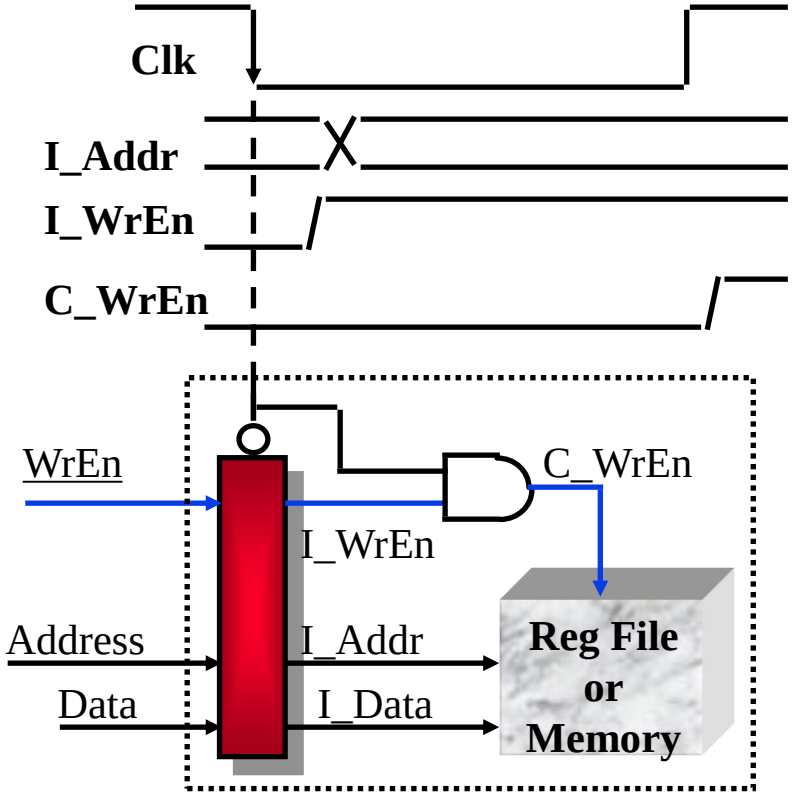
流水线的问题

- 多周期设计是如何防止Addr和WrEn之间的竞争的？
  - 确保在第N个周期结束时地址是稳定的
  - 在第N+1个周期使 WrEn有效
- 这种方法在流水线设计中不能使用！这是因为
  - 在每个周期，都必须能够写寄存器堆
  - 在每个周期，都必须能够写数据存储器



同步寄存器堆 和 同步存储器

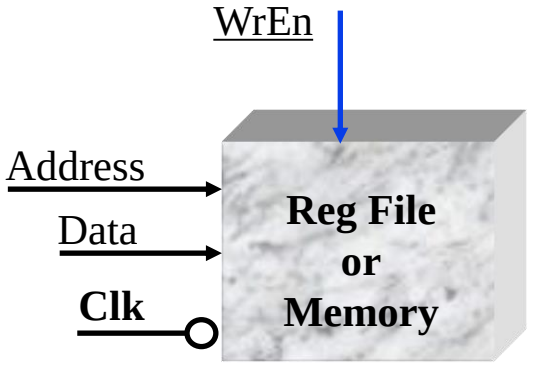
- 解决方案: 将 **Write Enable**信号 和 时钟 进行逻辑与
  - 只有这里可以使用门控的时钟信号
  - 必须 咨询电路专家，确保没有定时违例:
    - 例如: **Clock High Time > Write Access Delay**



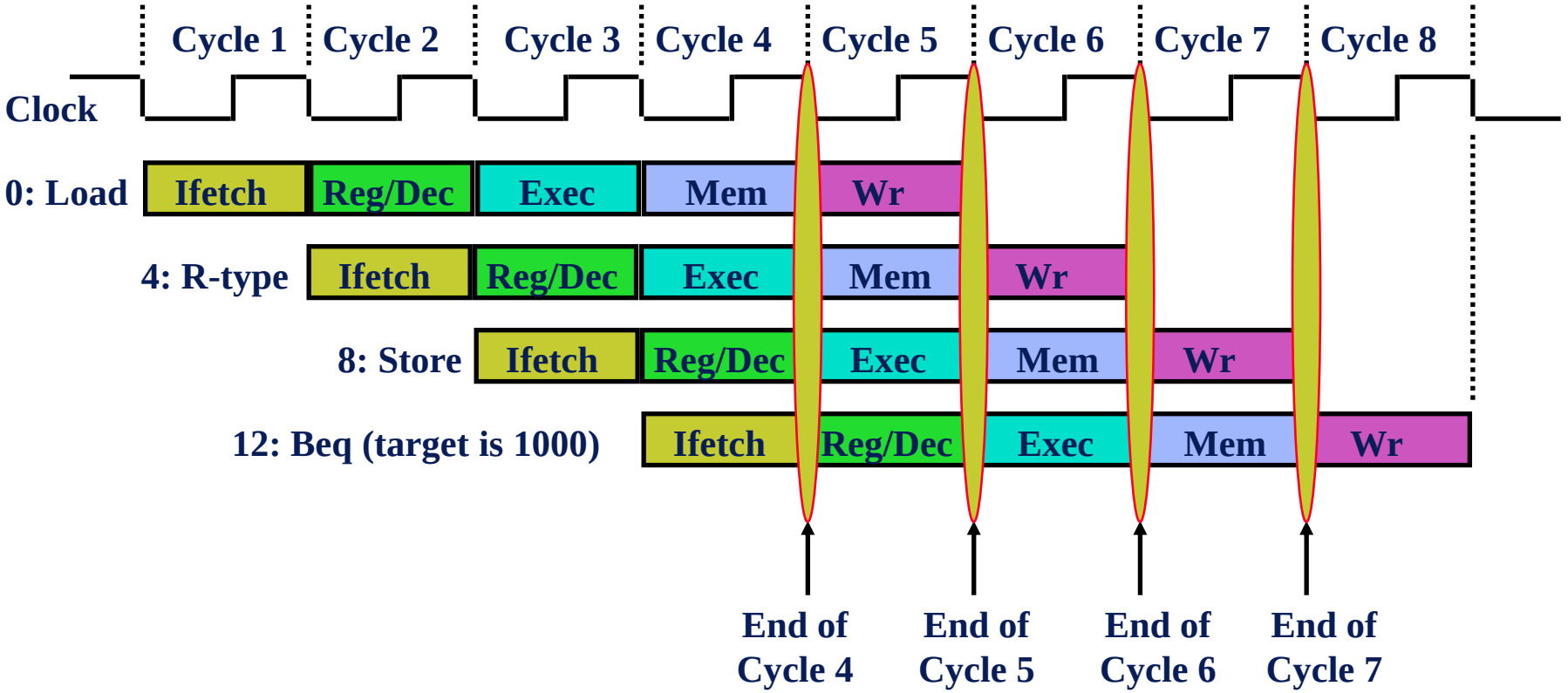
同步存储器 和 同步寄存器

至少在时钟沿前的建立时间之间，Address、Data和 WrEn必须稳定

在捕获这些信号的时钟沿之后的时钟周期，发生写操作



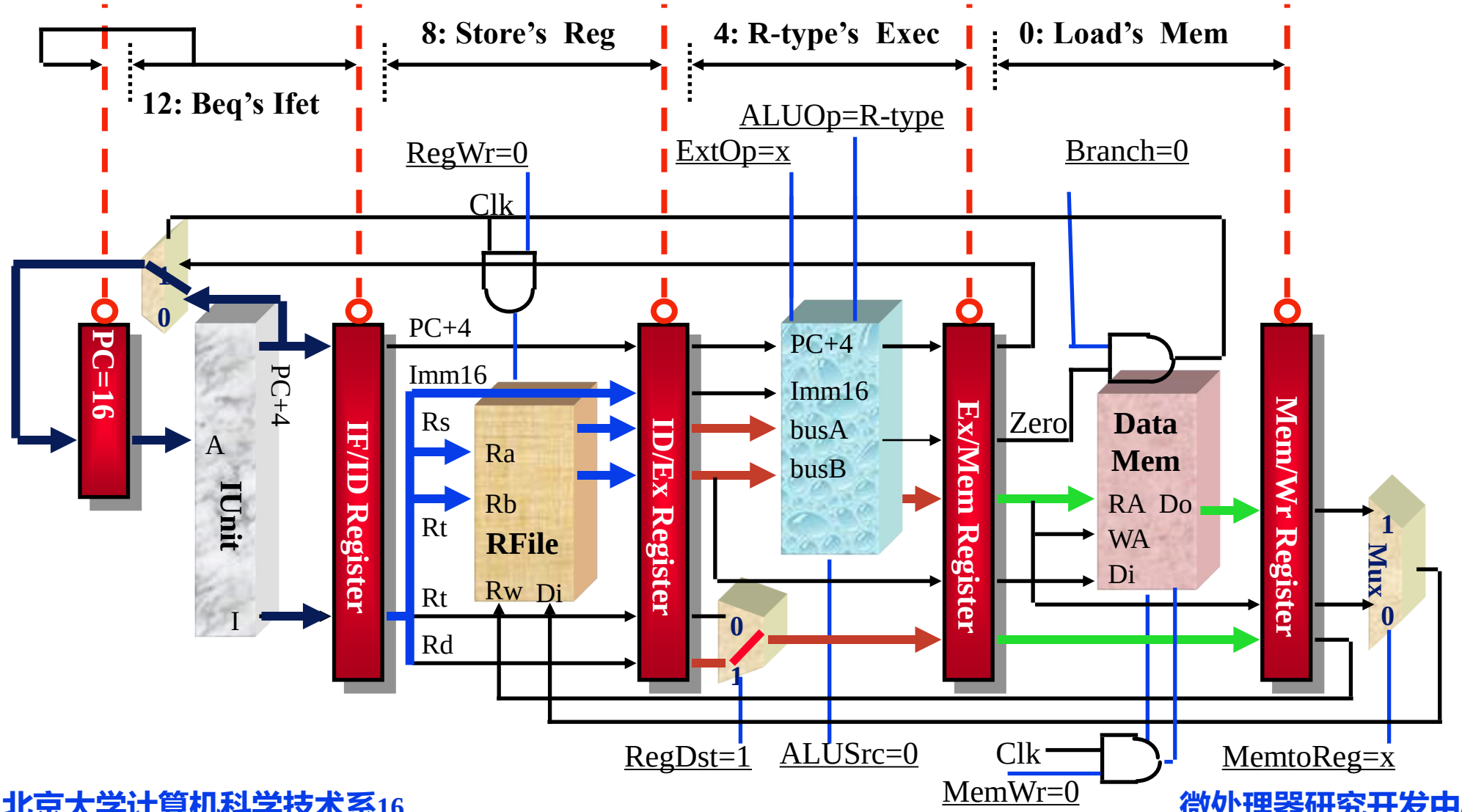
流水示例



- End of Cycle 4: Ld's Mem, R-tpe's Exec, St's Reg, Beq's Ifetch
- End of Cycle 5: Ld's Wr, R-type's Mem, St's Exec, Beq's Reg
- End of Cycle 6: R-type's Wr, St's Mem, Beq's Exec
- End of Cycle 7: Store's Wr, Beq's Mem

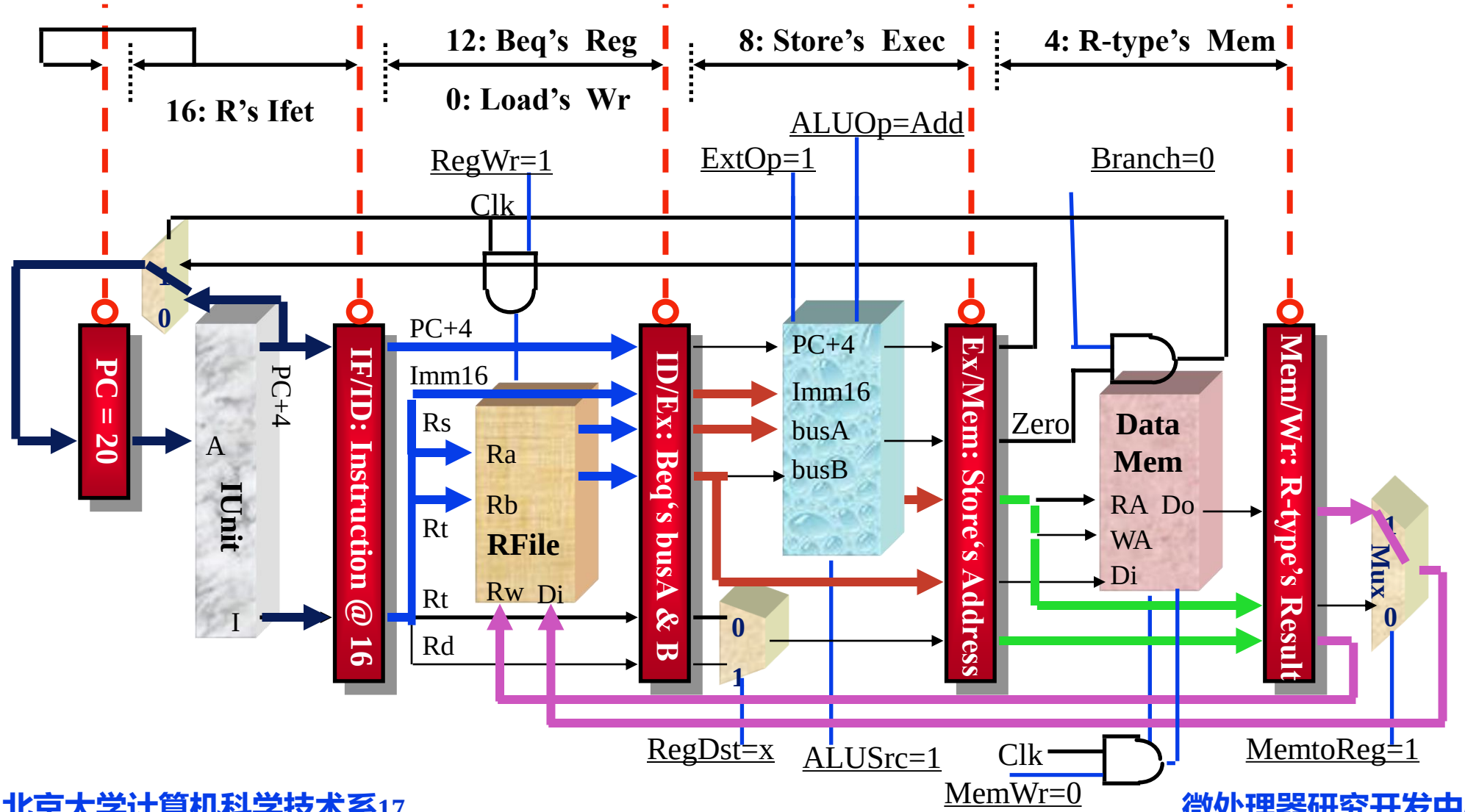
流水示例：第四周期结束

0: Load's Mem 4: R-type's Exec 8: Store's Reg 12: Beq's Ifetch



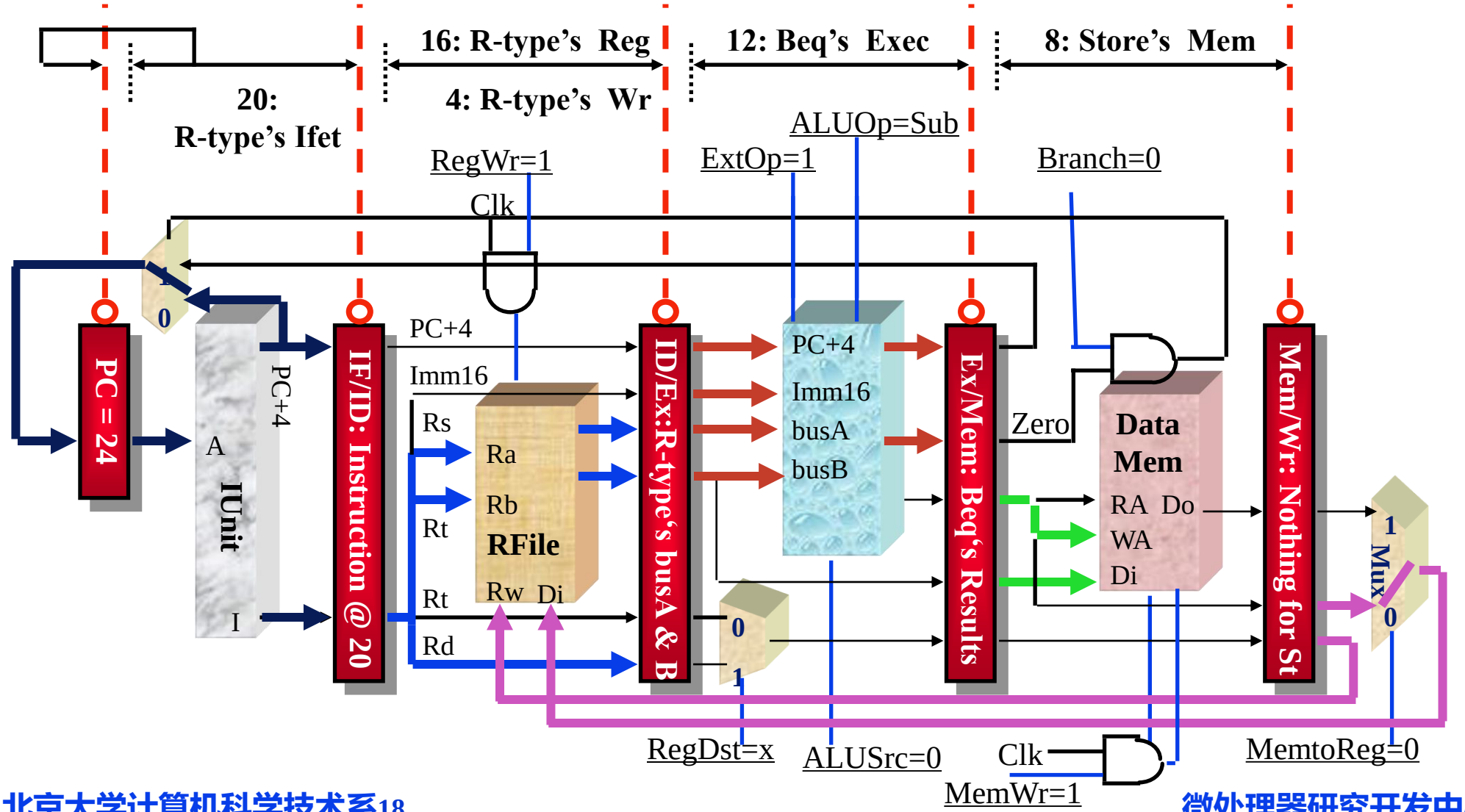
流水示例：第五周期结束

0: Lw's Wr 4: R's Mem 8: St's Exec 12: Beq's Reg 16: R's Ifetch



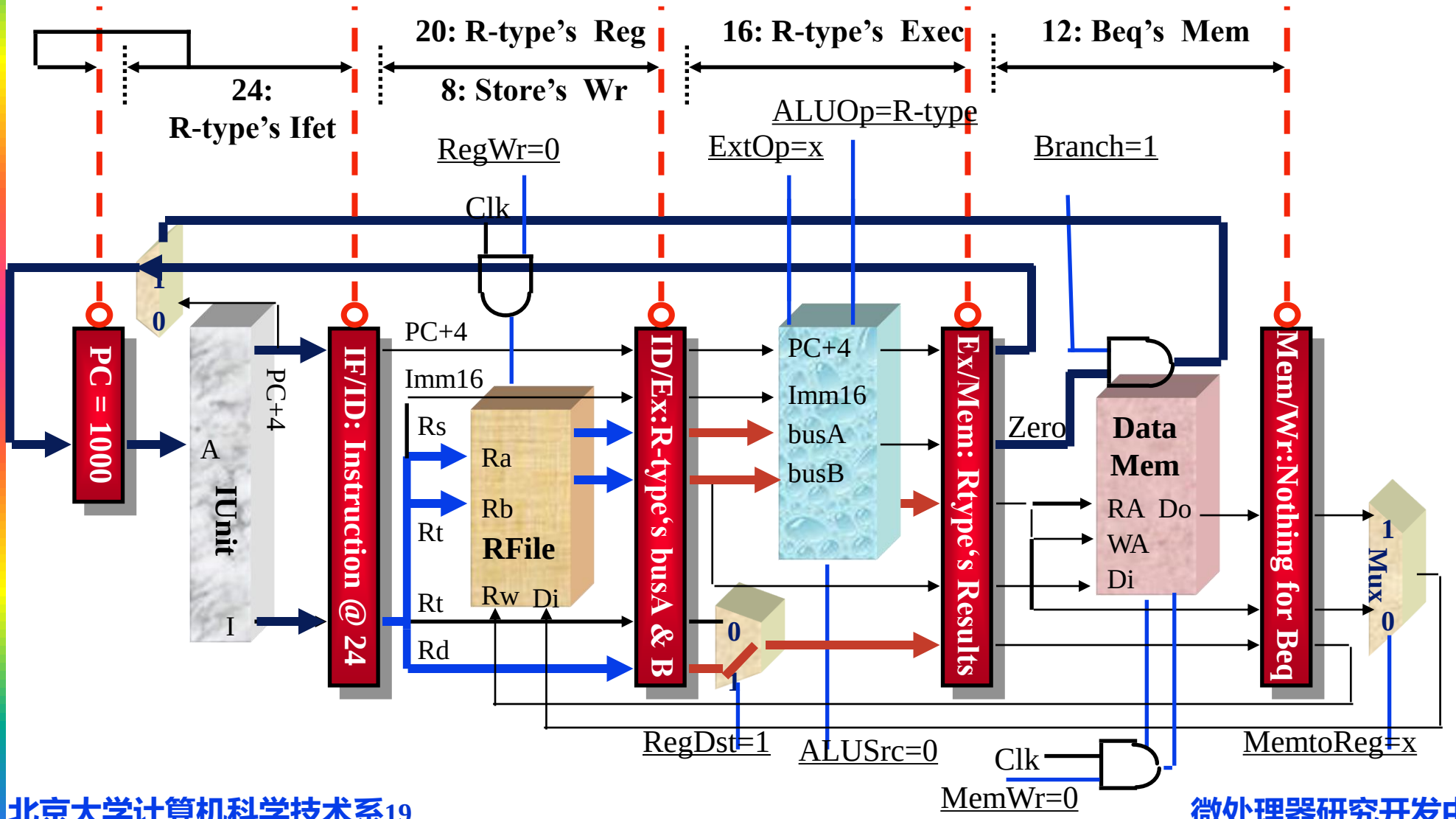
流水示例： 第六周期结束

4: R's Wr 8: St's Mem 12: Beq's Exec 16: R's Reg 20: R's Ifet

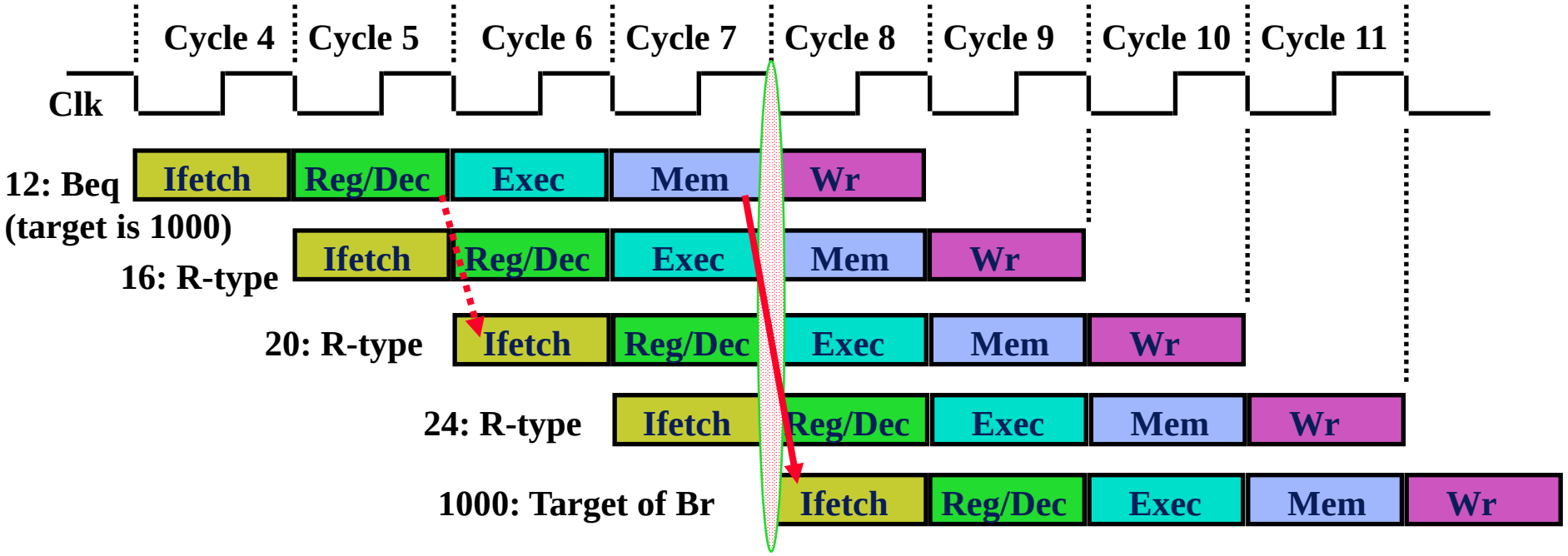


流水示例：第七周期结束

8: St's Wr 12: Beq's Mem 16: R's Exec 20: R's Reg 24: R's Ifet

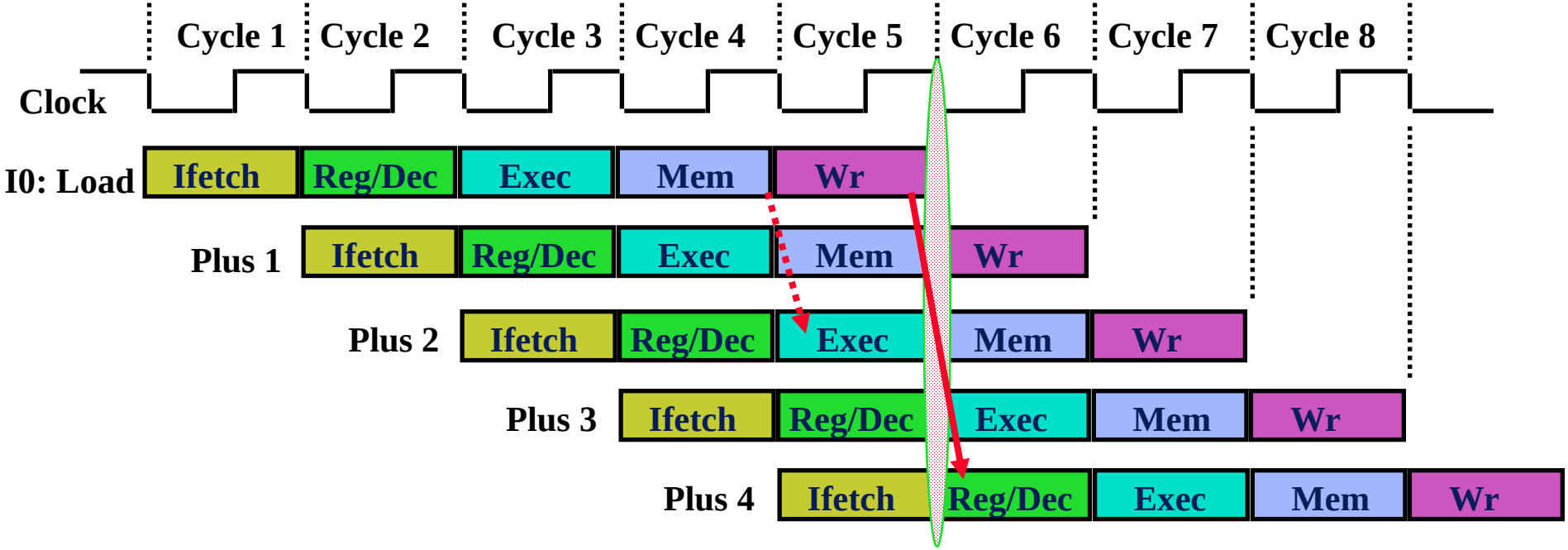


延迟转移现象



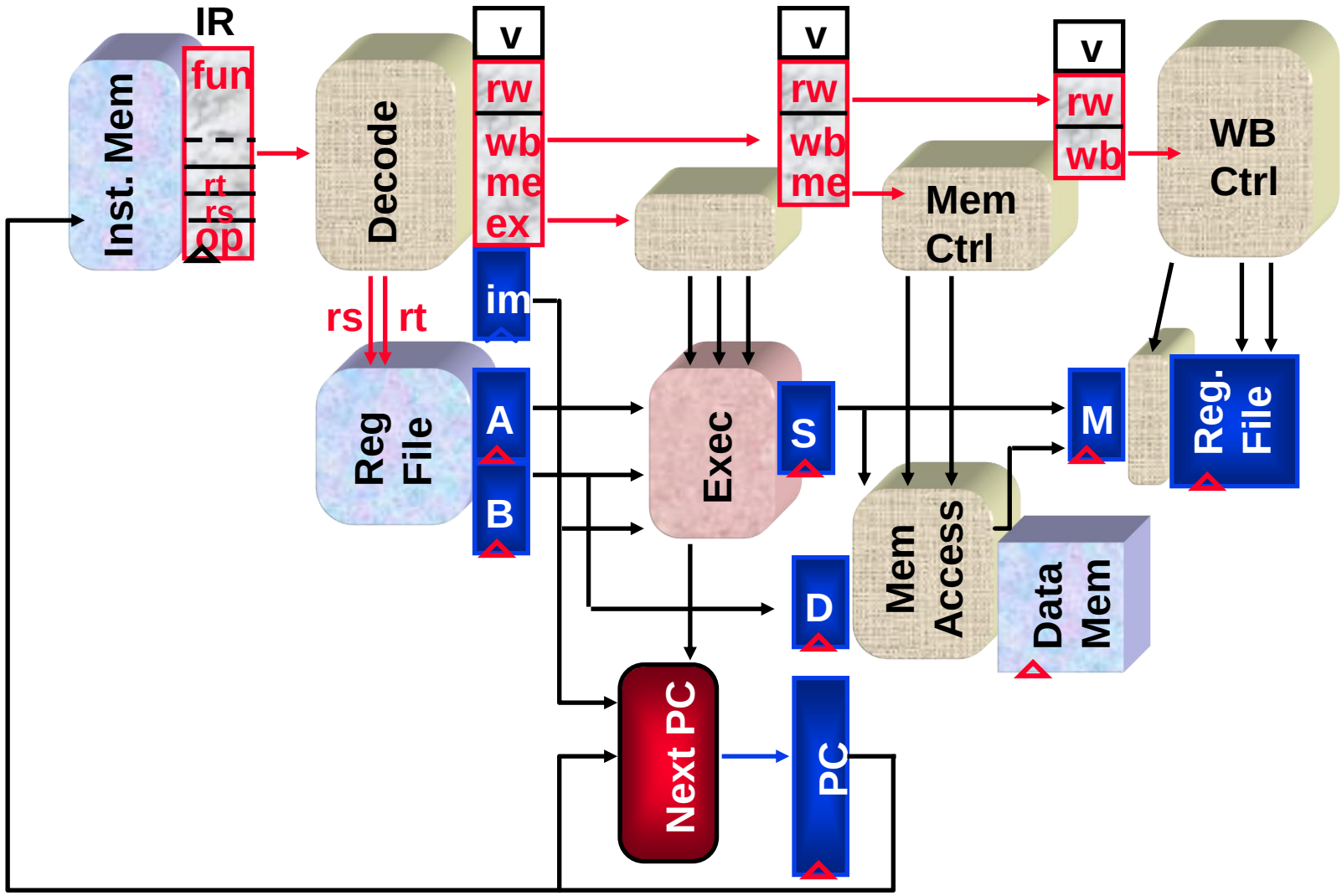
- 虽然Beq在第 4周期取指:
  - 目标地址在第7个周期结束时才能写入PC
  - 转移的目标在第8个周期前不能被取指
  - 在转移发生效果之前, 有3条指令被延迟
- 这也就是转移冒险(Branch Hazard):
  - 非常精致的设计有可能将这一延迟减小至一条指令

延迟装入现象



- 虽然Load在第一个周期就被取指:
  - 它的数据直到第五个周期结束时才被写入到寄存器堆
  - 在第六个周期之前, 不可能从寄存器堆中读取这一数值
  - 在装入指令结束之前, 有3条指令被延迟

数据通路 + 数据-固定控制

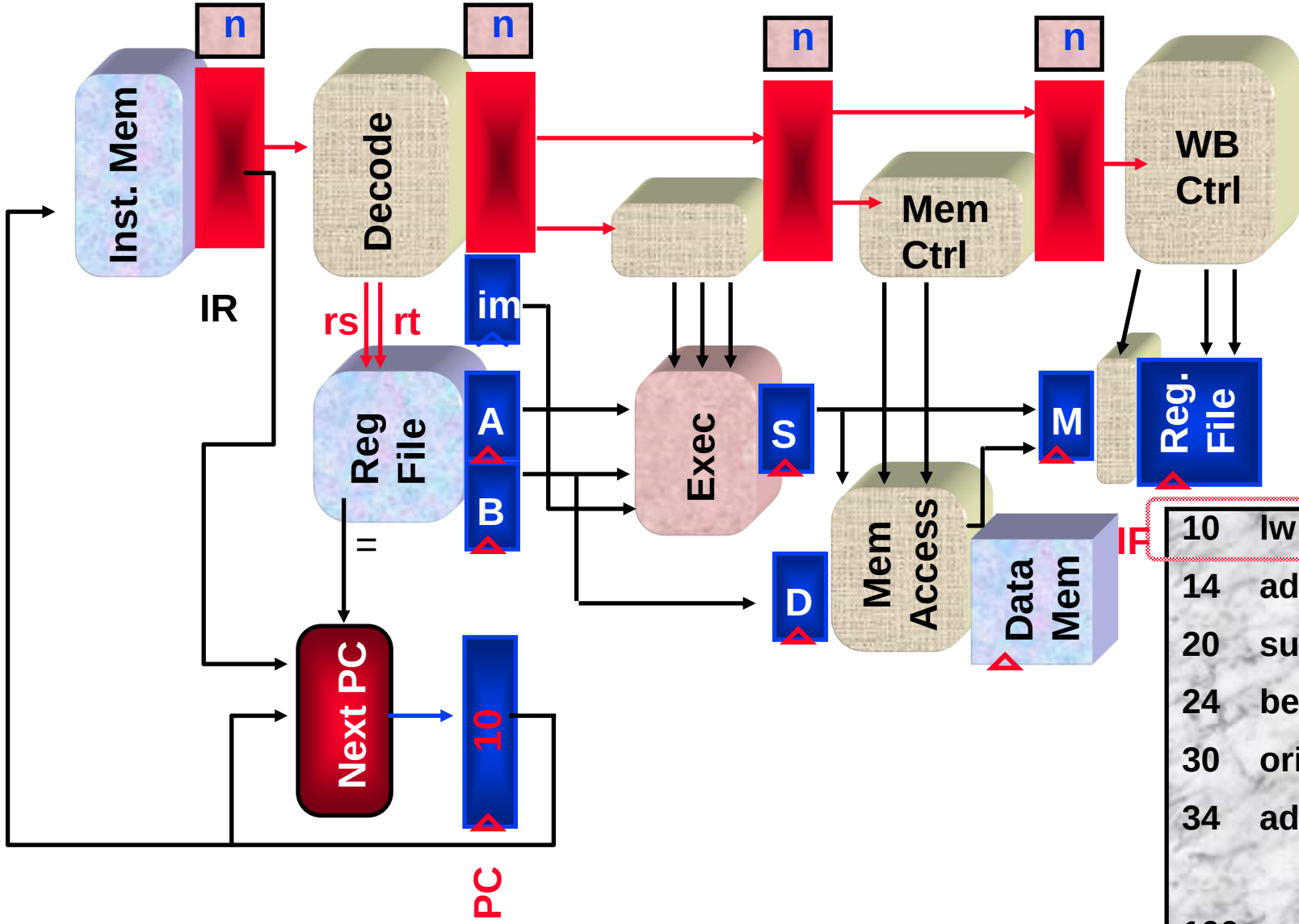

IR
Inst. Mem
fun
rt
rs
op
Decode
v
rw
wb
me
ex
im
rs
rt
Reg File
A
B
Exec
S
D
Mem Access
Data Mem
M
Mem Ctrl
WB Ctrl
Reg. File
Next PC
PC

指令流水示例

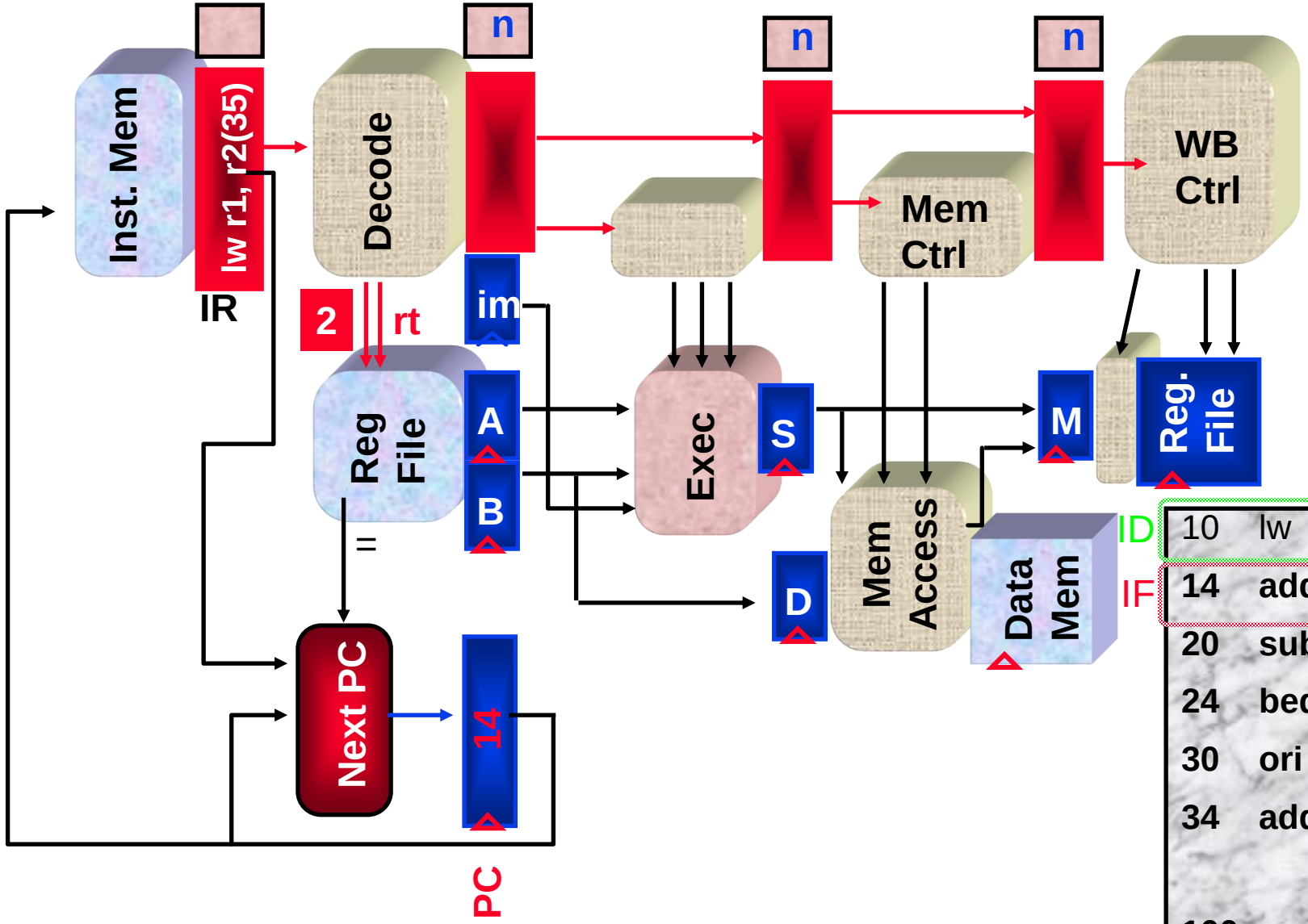
```
10     lw     r1, r2(35)
14     addi   r2, r2, 3
20     sub    r3, r4, r5
24     beq    r6, r7, 100
30     ori    r8, r9, 17
34     add    r10, r11, r12


100    and    r13, r14, 15
```

地址以八进制表示

开始：从地址10取指

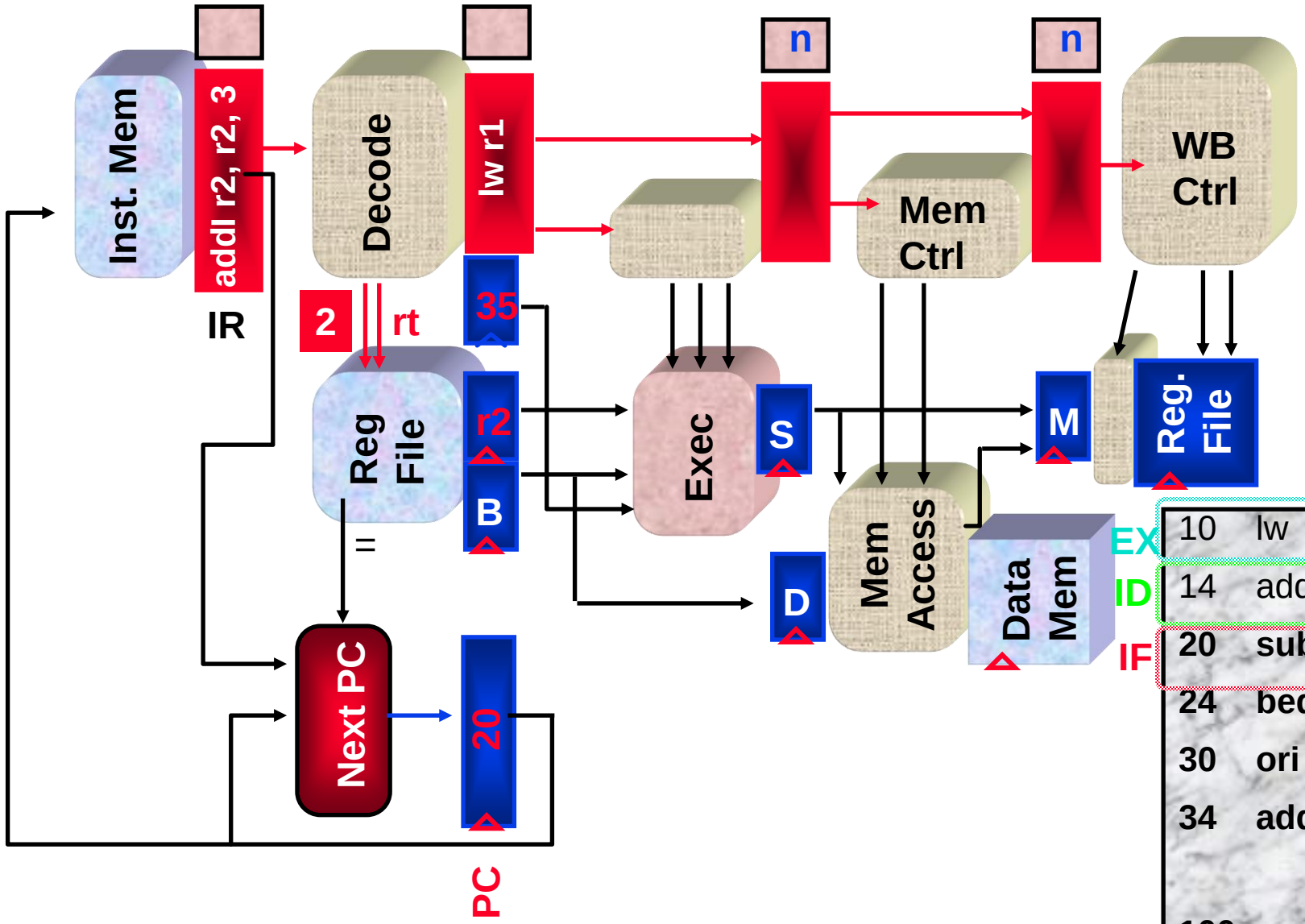

n
Inst. Mem
IR
Decode
rs
rt
im
A
B
Reg File
=
Next PC
10
PC
Exec
S
D
Mem Ctrl
Mem Access
Data Mem
M
Reg. File
WB Ctrl
IR
10 lw r1, r2(35)
14 addI r2, r2, 3
20 sub r3, r4, r5
24 beq r6, r7, 100
30 ori r8, r9, 17
34 add r10, r11, r12
100 and r13, r14, 15

Fetch14 # Decode10



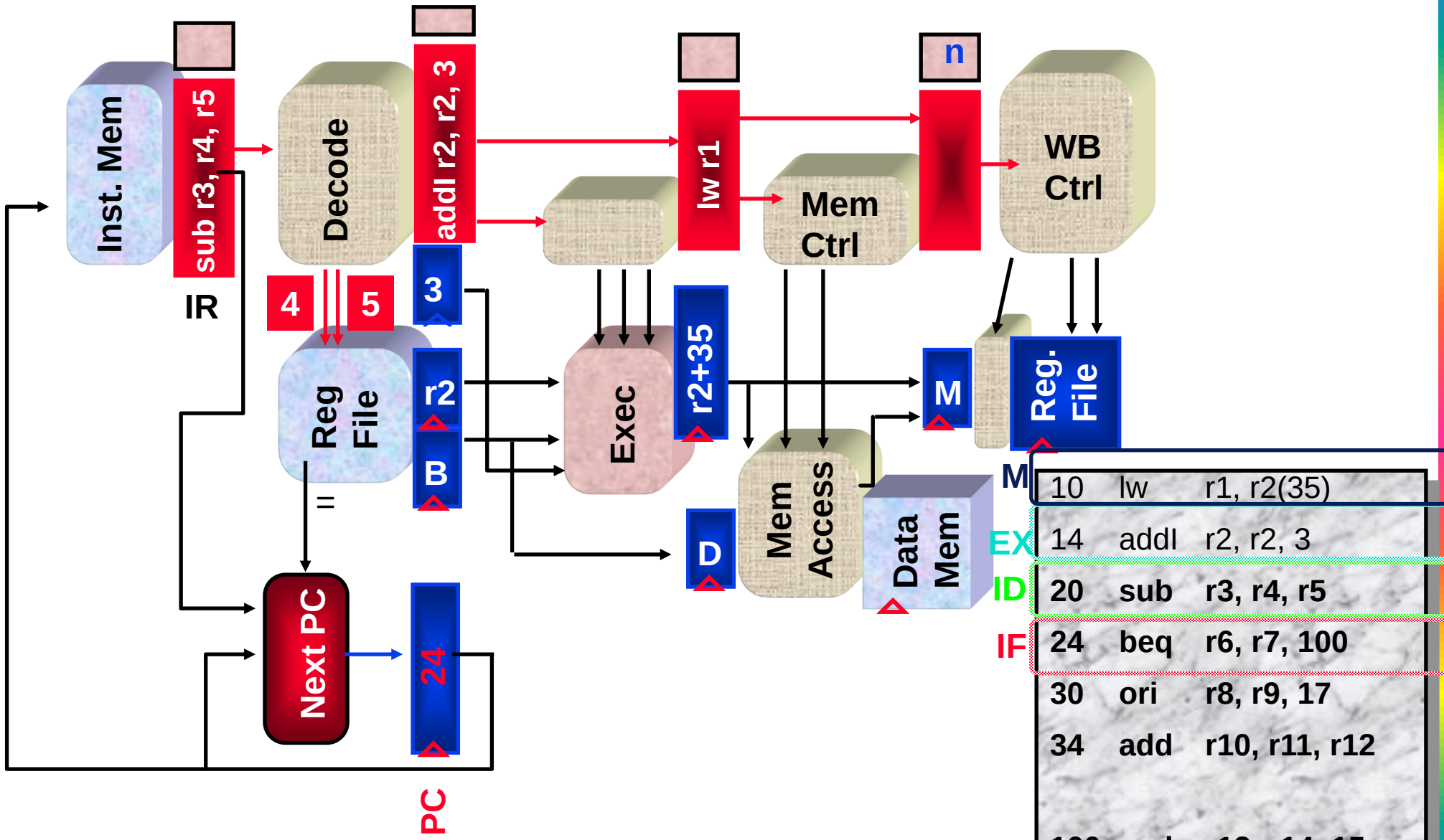

```
10   lw     r1, r2(35)
14   addI   r2, r2, 3
20   sub    r3, r4, r5
24   beq    r6, r7, 100
30   ori    r8, r9, 17
34   add    r10, r11, r12


100  and    r13, r14, 15
```

Fetch20 #Decode 14 # Exec10

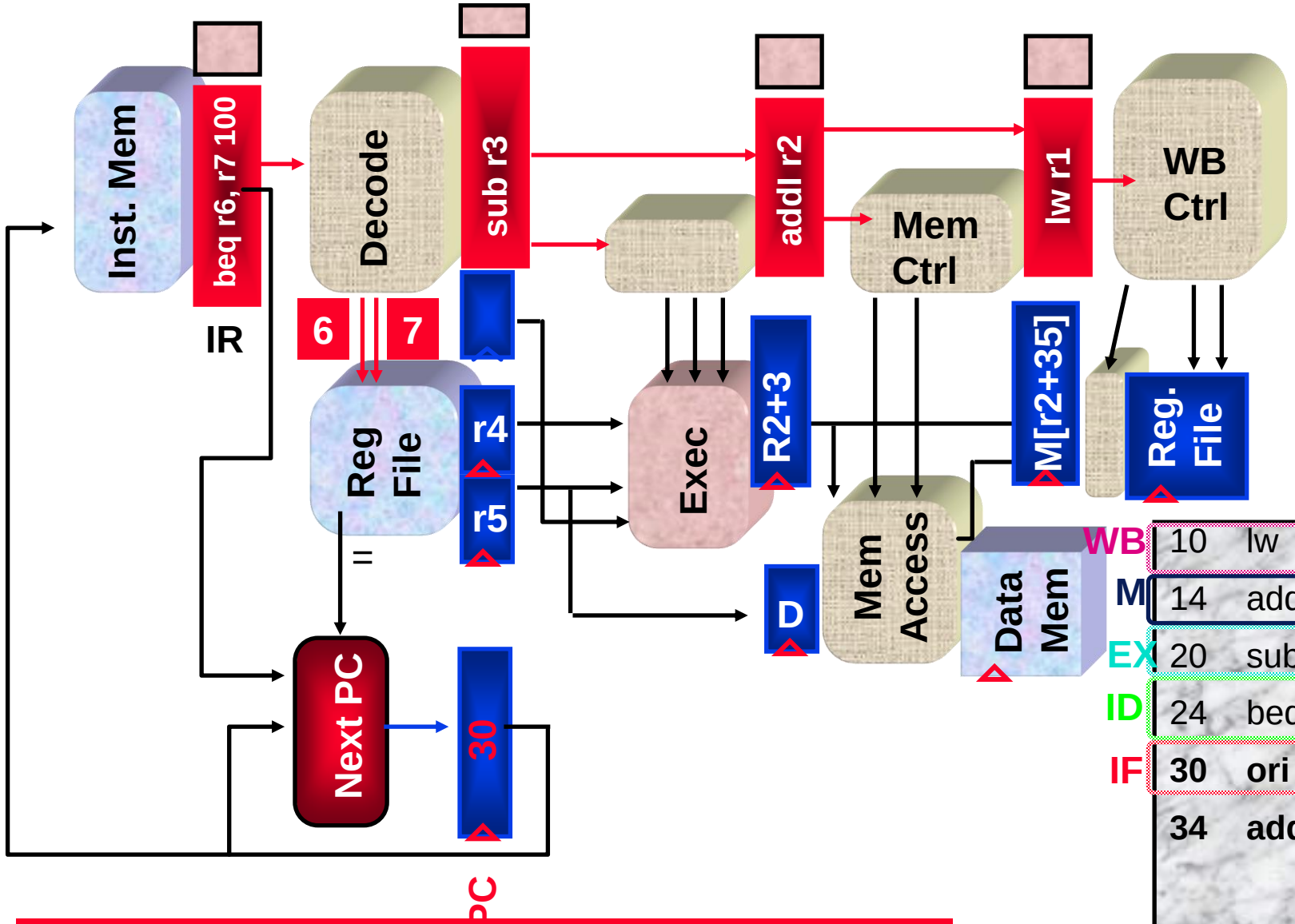



```
10   lw     r1, r2(35)
14   addl   r2, r2, 3
20   sub    r3, r4, r5
24   beq    r6, r7, 100
30   ori    r8, r9, 17
34   add    r10, r11, r12

100  and    r13, r14, 15
```

Fetch24 #Decode20 # Exec14 # Mem10

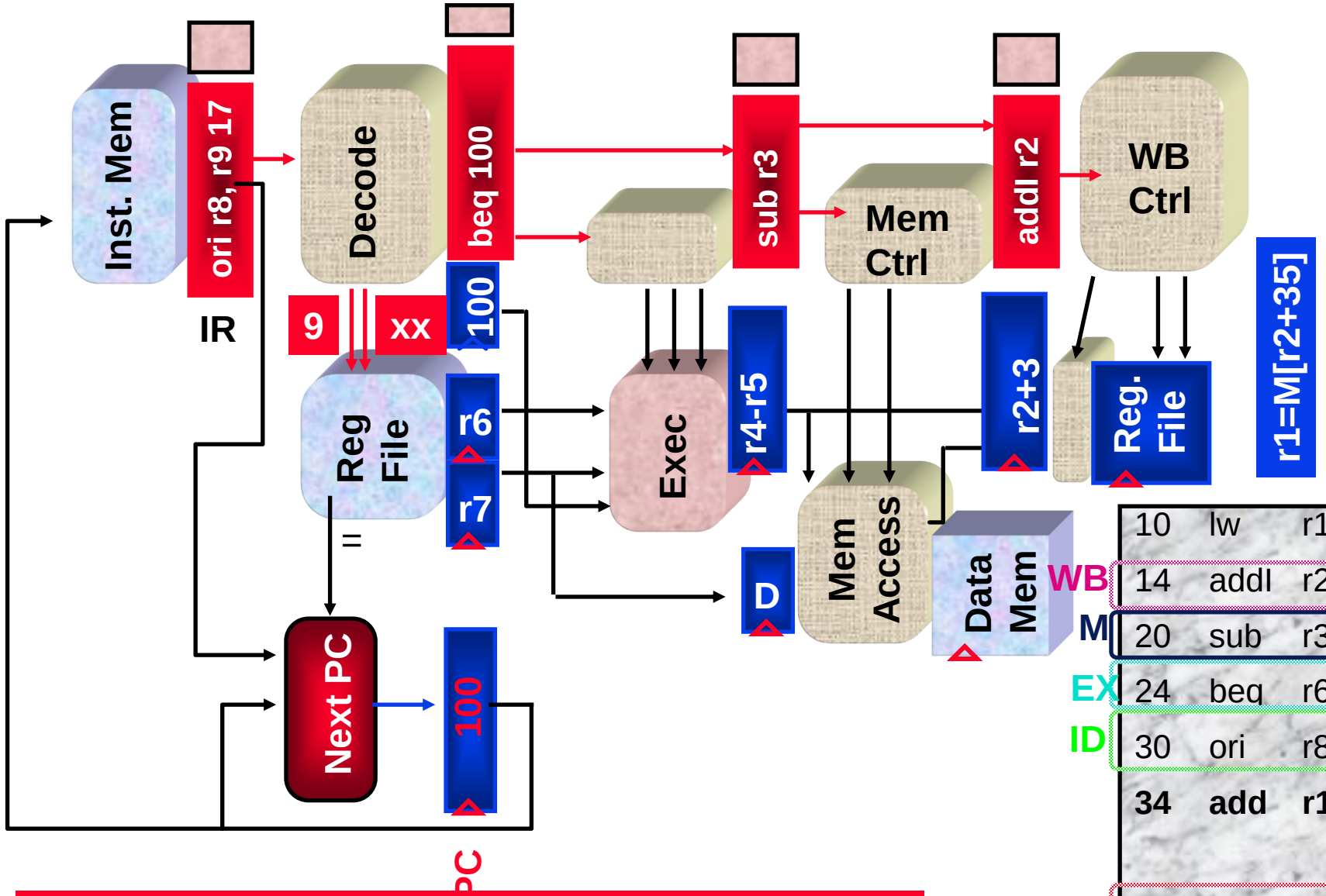


Fetch30并Decode24并 Exec20并Mem14并WB10



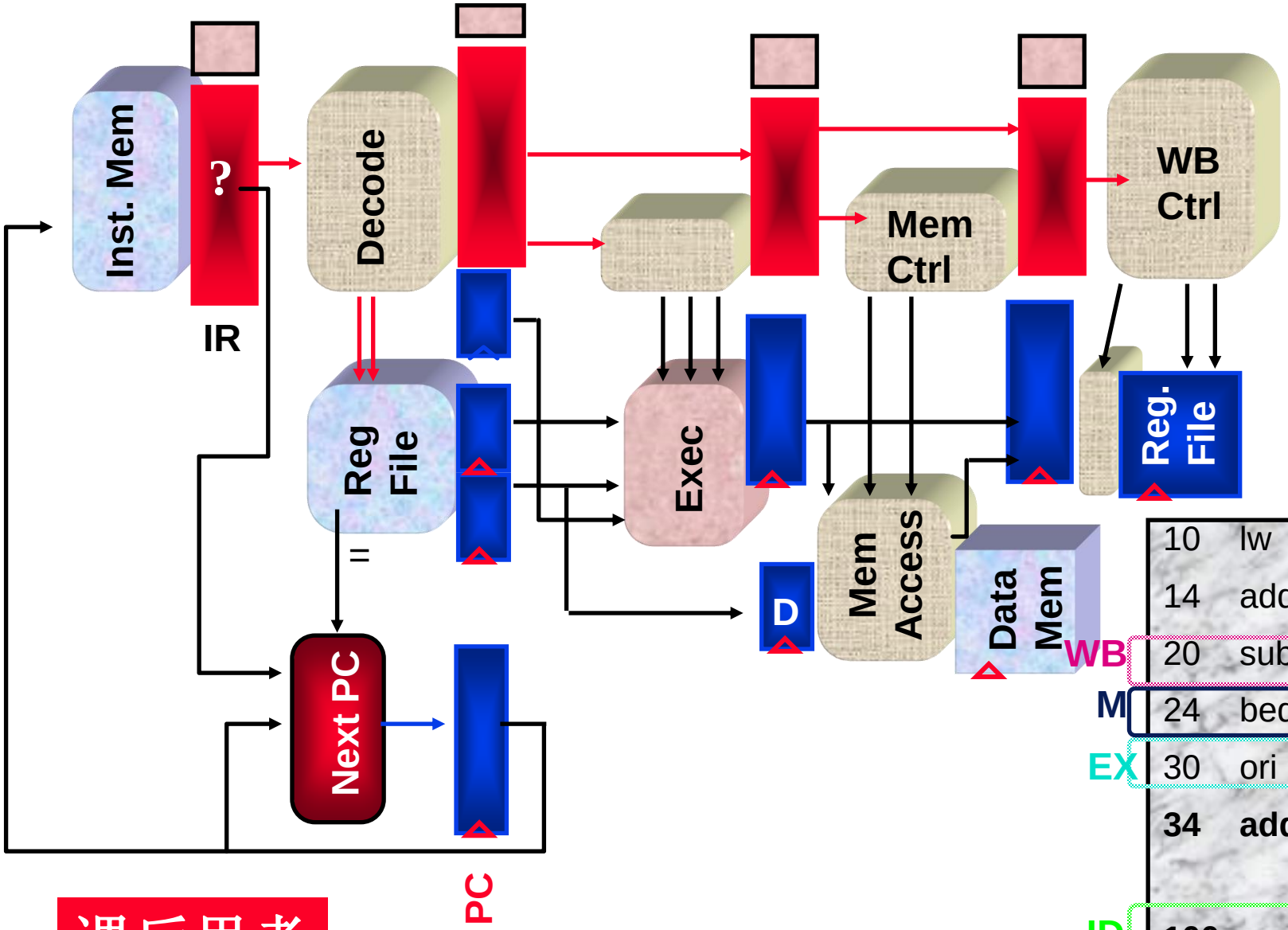
注意延迟转移: 在Beq之后总是执行 ori

Fetch100并Decode30并Exec24并Mem20并WB14



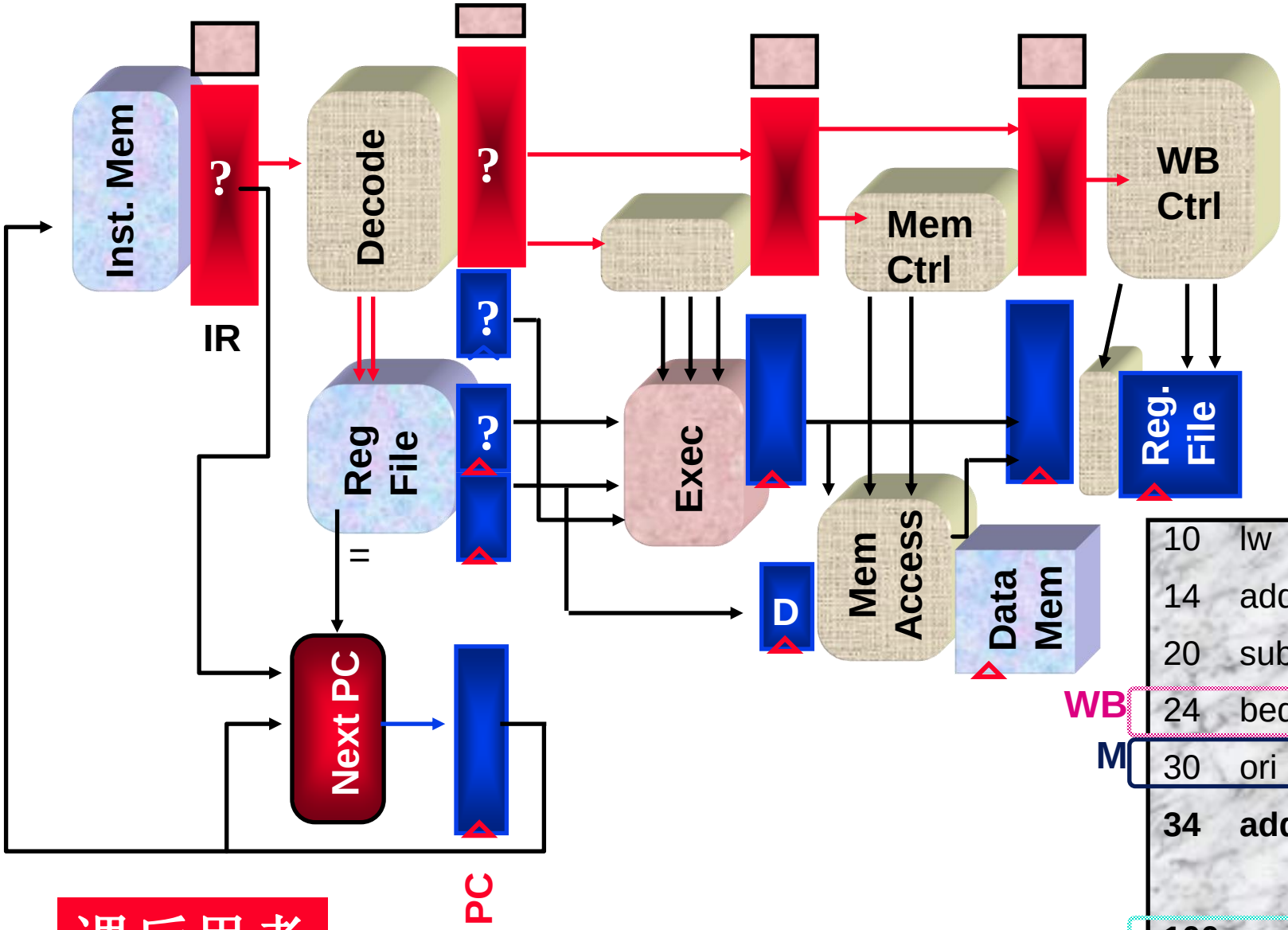
注意延迟转移: 在Beq之后总是执行 ori

Fetch104并Decode100并Exec30并Mem24并WB20



课后思考

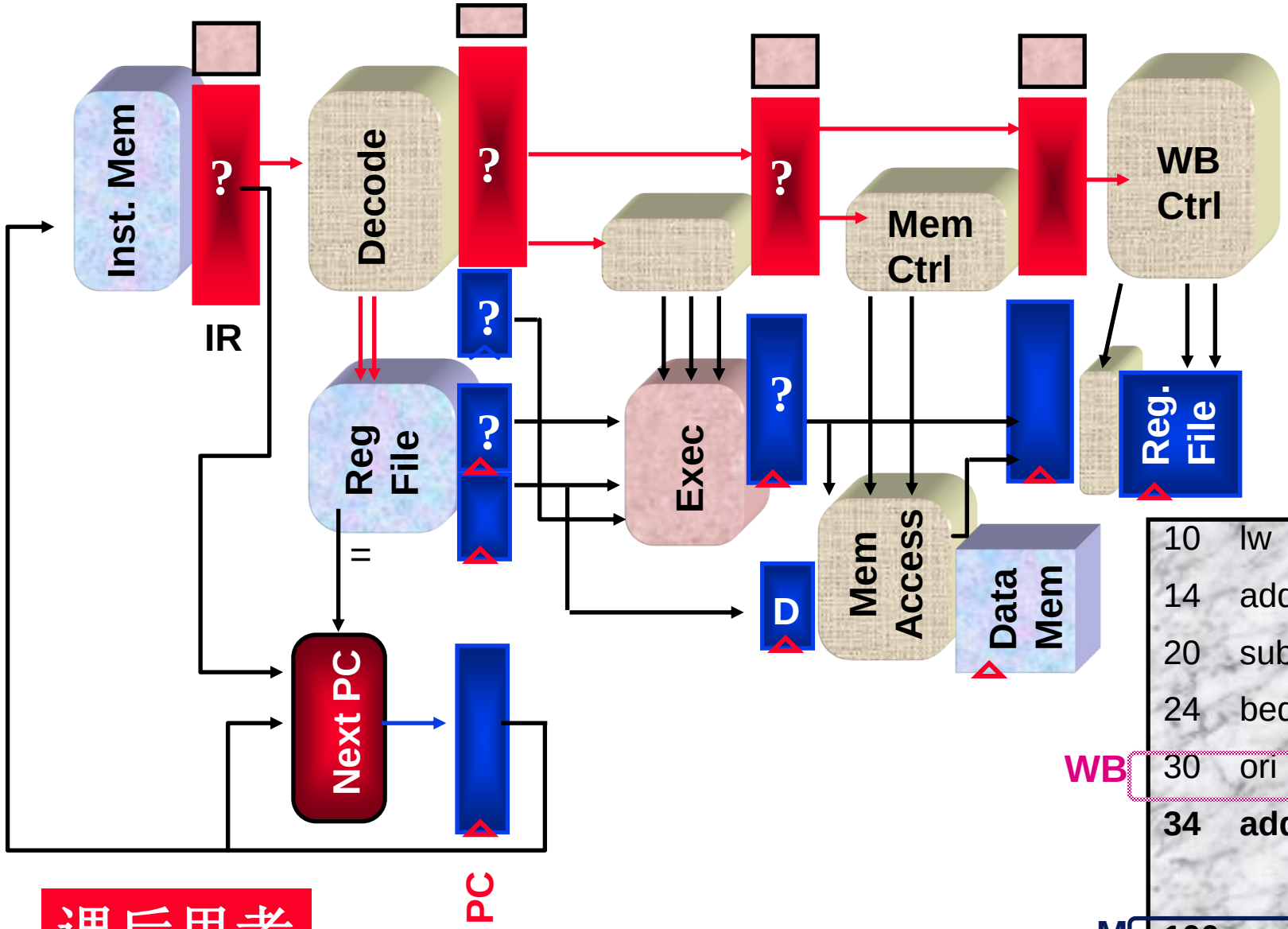
Fetch110并Decode104并Exec100并Mem30并WB24



	10	lw	r1, r2(35)
	14	addI	r2, r2, 3
	20	sub	r3, r4, r5
WB	24	beq	r6, r7, 100
M	30	ori	r8, r9, 17
	34	**add**	**r10, r11, r12**
EX	**100**	**and**	**r13, r14, 15**

课后思考

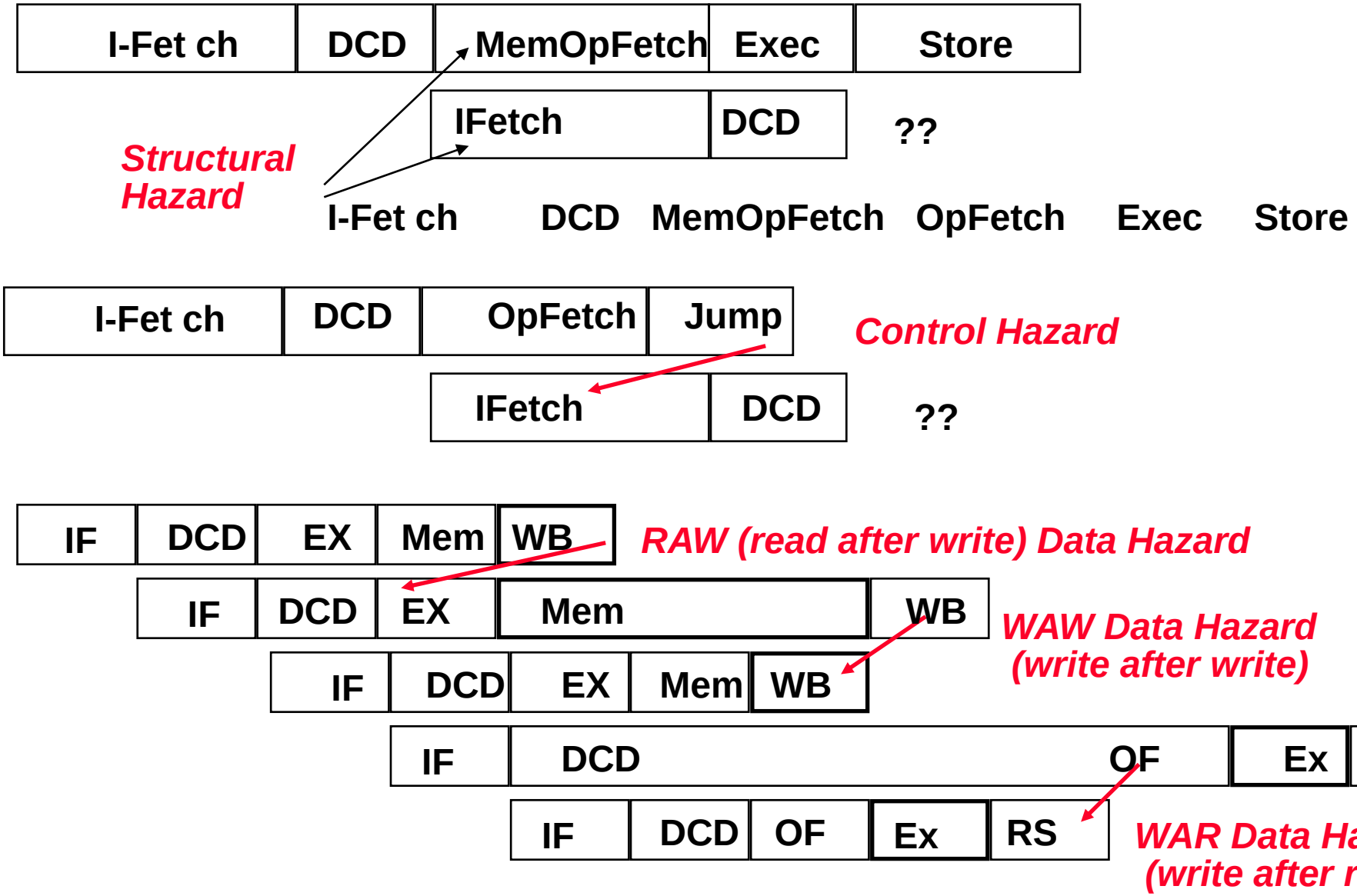
Fetch114并Decode110并Exec104并Mem100并WB30



	10	lw	r1, r2(35)
	14	addl	r2, r2, 3
	20	sub	r3, r4, r5
	24	beq	r6, r7, 100
WB	30	ori	r8, r9, 17
	34	**add**	**r10, r11, r12**
M	**100**	**and**	**r13, r14, 15**

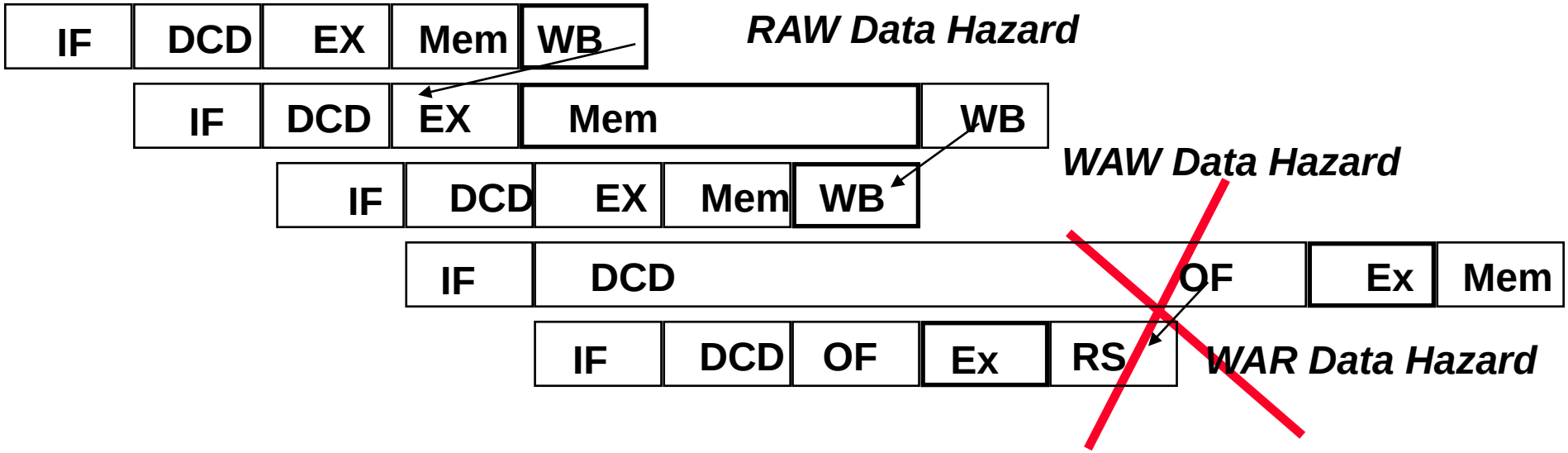
课后思考

再谈流水线冒险


I-Fet ch
DCD
MemOpFetch
Exec
Store
IFetch
DCD
??
Structural Hazard
I-Fet ch
DCD
MemOpFetch
OpFetch
Exec
Store
I-Fet ch
DCD
OpFetch
Jump
Control Hazard
IFetch
DCD
??
IF
DCD
EX
Mem
WB
RAW (read after write) Data Hazard
IF
DCD
EX
Mem
WB
WAW Data Hazard
(write after write)
IF
DCD
EX
Mem
WB
IF
DCD
OF
Ex
Mem
IF
DCD
OF
Ex
RS
WAR Data Hazard
(write after read)

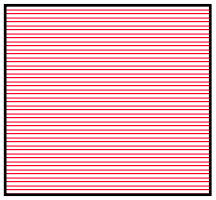
数据冒险

- 如何避免一些冒险
  - 通过总是在流水线的前段(DCD)取操作数, 来消除WAR
  - 通过按序完成所有回写操作(在最后一级,静态), 来消除WAW
- 检测并解决RAW
  - 暂停, 并尽可能前递



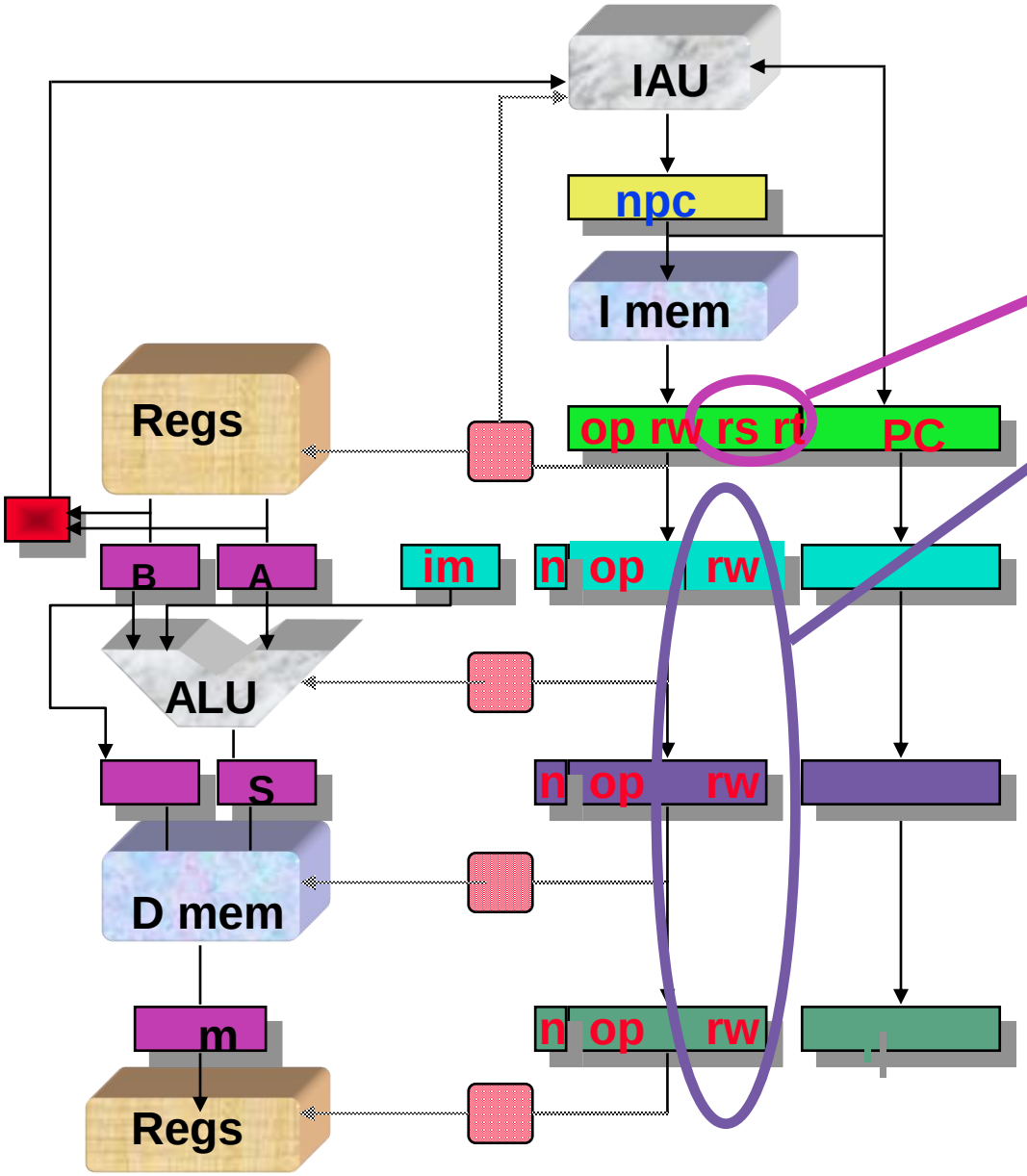
冒险检测

- 假设指令*i*将被发射，它的前指令 *j* 在指令流水线中
- 关于寄存器r存在RAW冒险，如果r ∈ Rregs(*i*) ∩ Wregs(*j*)
  - 保存流水线中指令的尚未完成写的纪录信息,并与当前指令的操作数寄存器进行比较
  - 当发送指令时,预留它的结果寄存器
  - 当完成操作后，删除它的写预留(write reservation)



- 关于寄存器r存在WAW冒险，如果r ∈ Wregs(*i*) ∩ Wregs(*j*)
- 关于寄存器r存在WAR冒险，如果r ∈ Wregs(*i*) ∩ Rregs(*j*)

未完成写的纪录



当前操作数寄存器

未完成写操作

冒险 <=

$((rs == rw_{ex)} \ \& \ regW_{ex}) \ OR$

$((rs == rw_{mem)} \ \& \ regW_{me}) \ OR$

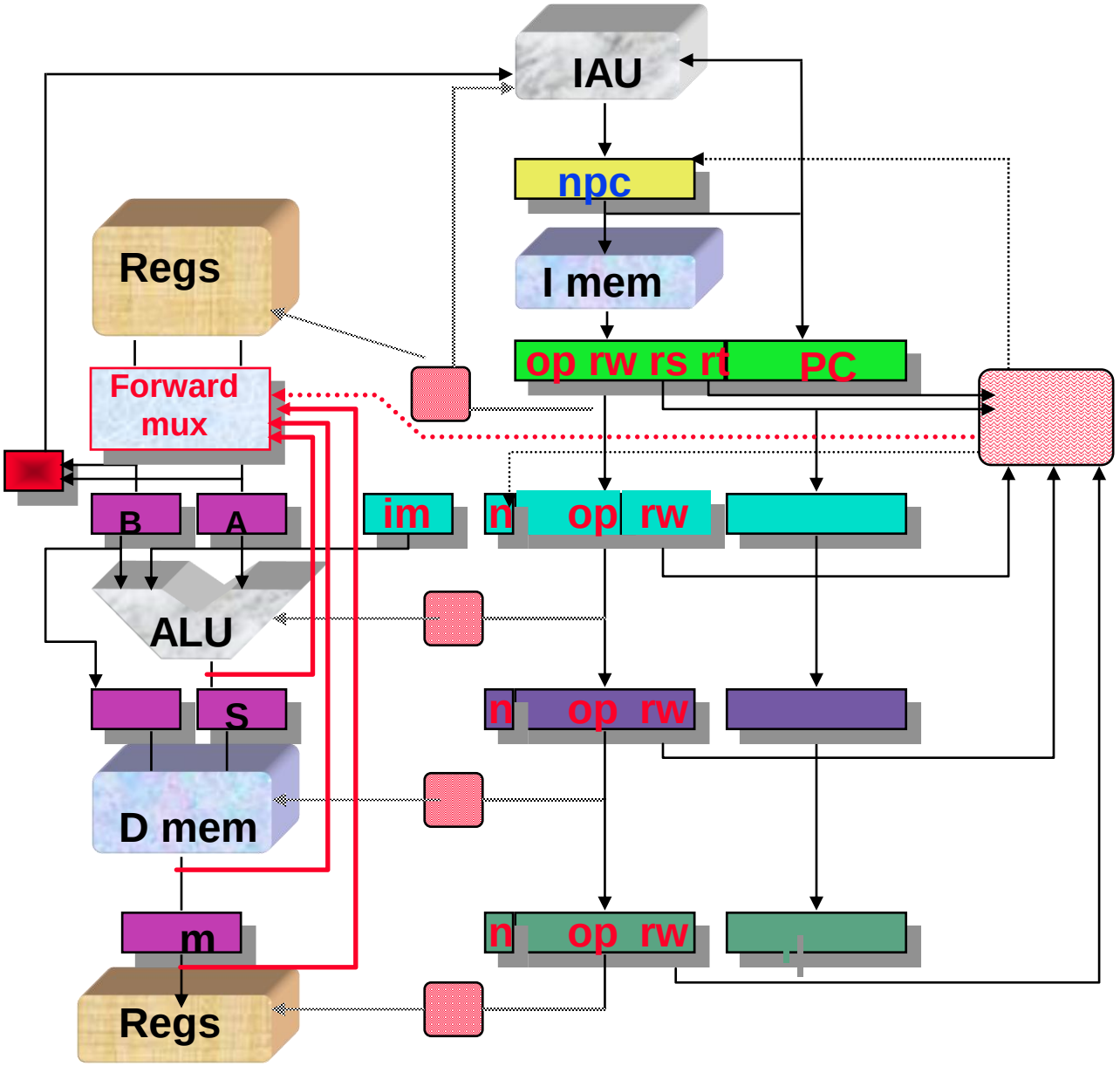
$((rs == rw_{wb)} \ \& \ regW_{wb}) \ OR$

$((rt == rw_{ex)} \ \& \ regW_{ex}) \ OR$

$((rt == rw_{mem)} \ \& \ regW_{me}) \ OR$

$((rt == rw_{wb}) \ \& \ regW_{wb})$

通过前递解决RAW冒险



检测最近的 valid 写操作操作数寄存器 并 forward 到 op锁存器，bypassing 流水线中的其他部分

增加多路选择器来增设来自流水线寄存器的通路

Data Forwarding = Data Bypassing

存储器操作

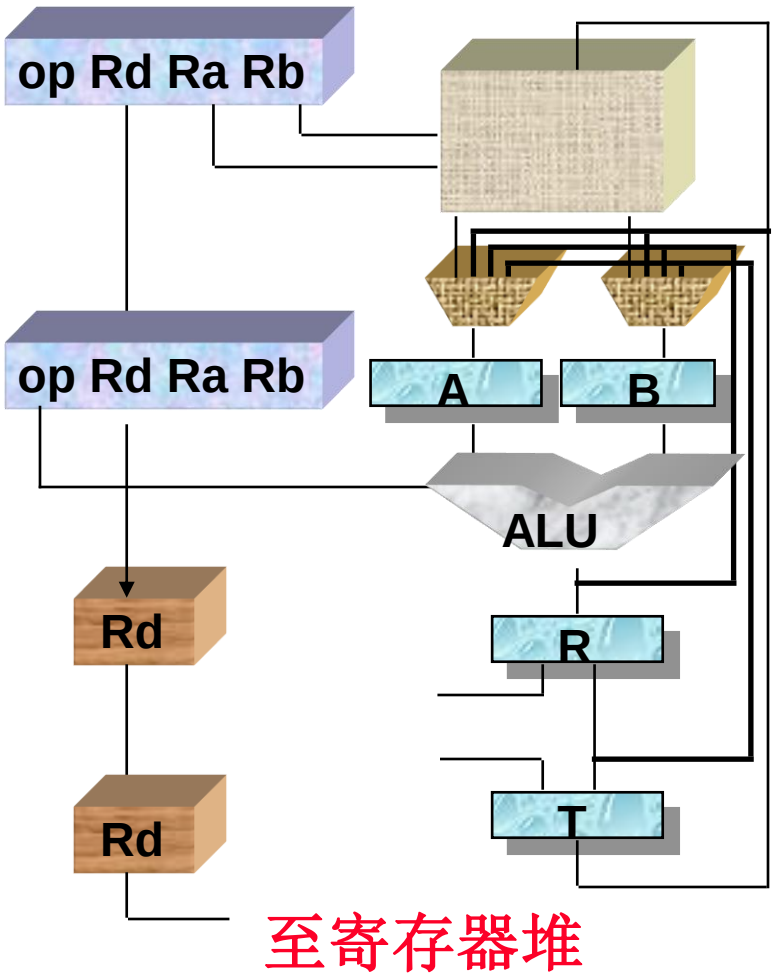
- 如果指令按序开始处理, 并且在相同段执行操作, 那么存储器操作之间就没有冒险!

- 延迟算术操作的回写会产生哪些开销?
  - 时钟周期?
  - 硬件 ?

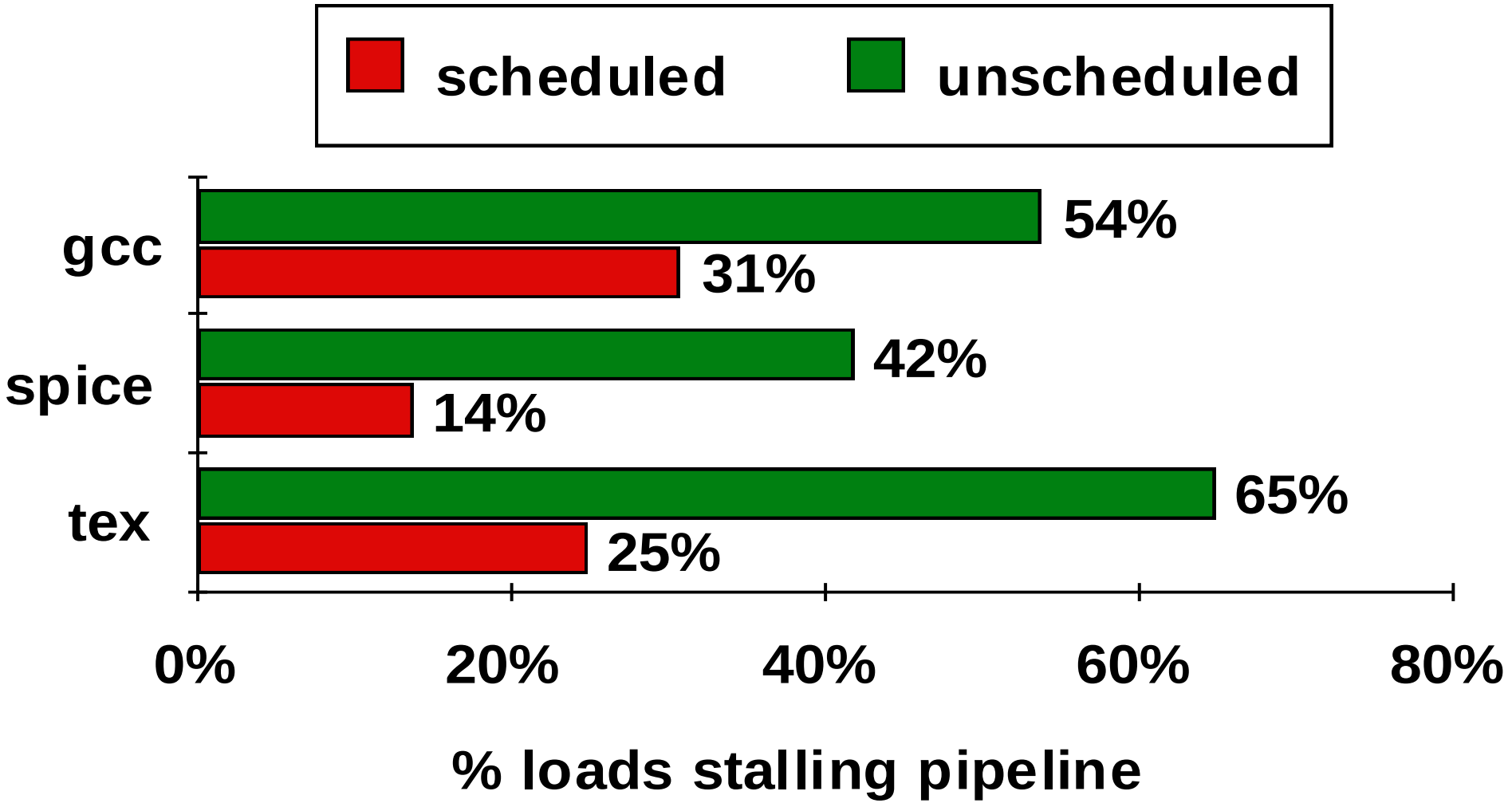
- 关于装入指令的数据相关如何?
  - R1 ⇐ R4 + R5
  - R2 ⇐ Mem[R2 + i]
  - R3 ⇐ R2 + R1

  =>

"Delayed Loads"



编译消除装入暂停的效果

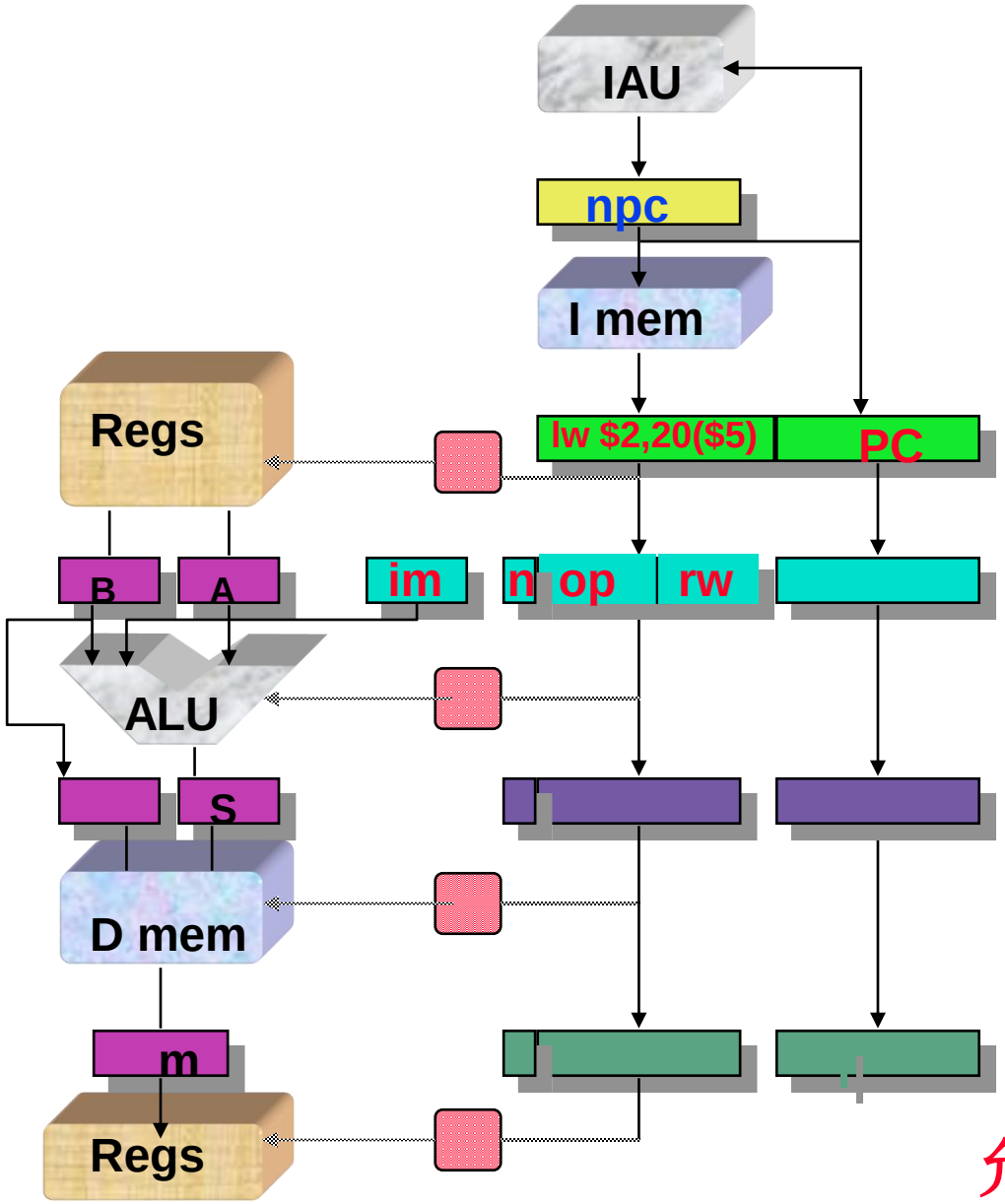


如何处理中断、自陷、故障？

- 外部中断:
  - 允许流水线排空,
  - PC ⇐ 中断地址
- 故障(指令内部、可重启)
  - 强制自陷指令进入IF
  - 禁止写操作直到自陷指令达到WB
  - 必须保存多个PCs 或者 PC + state

Refer to MIPS solution

意外事件处理



检测错误指令地址

检测错误指令

检测溢出

检测错误数据地址

允许意外事件生效！

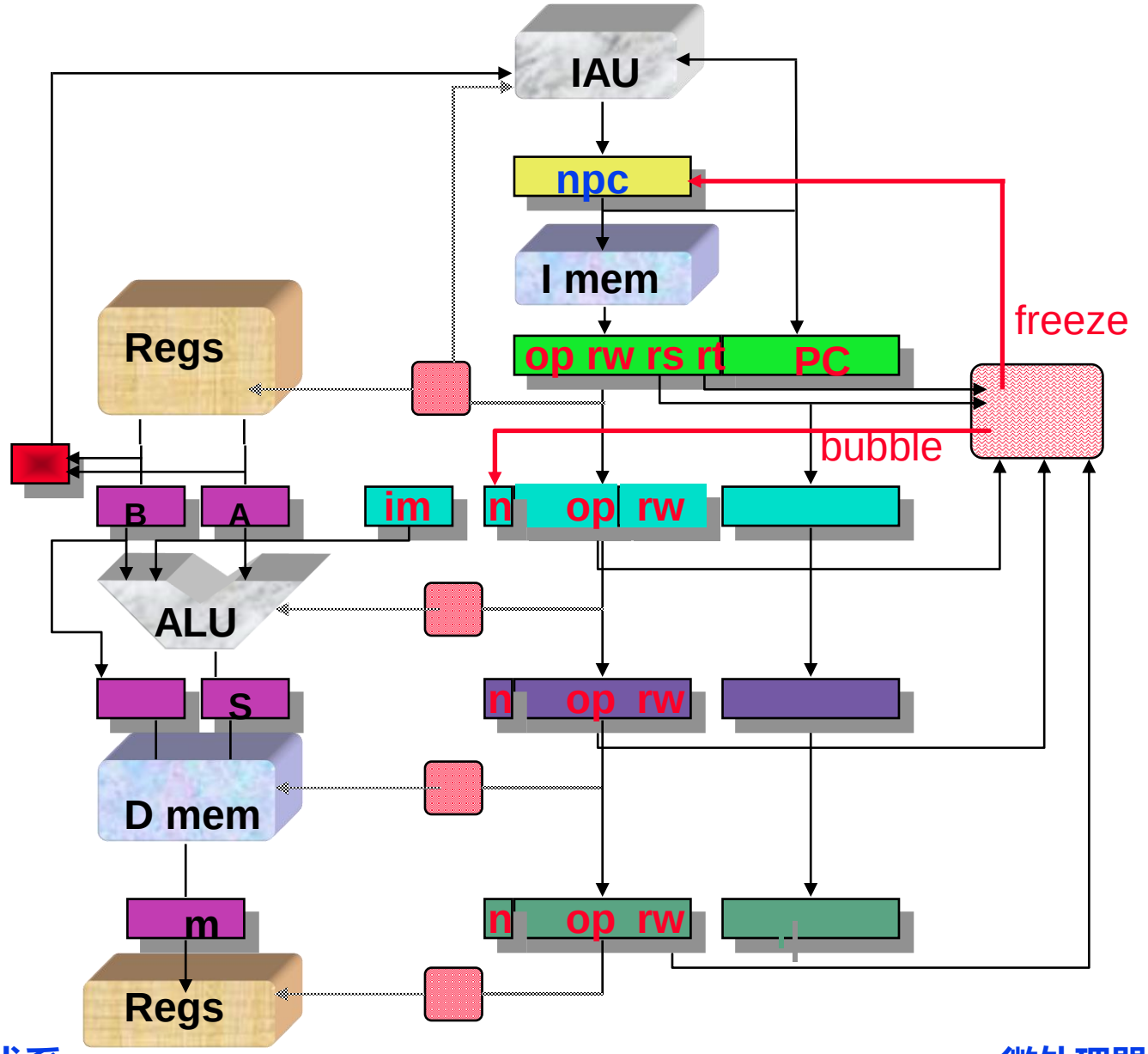
意外事件中的问题

- 意外事件/中断: 在5段流水线中执行着5条指令
  - 如何停止流水线?
  - 重启?
  - 哪些问题产生中断?

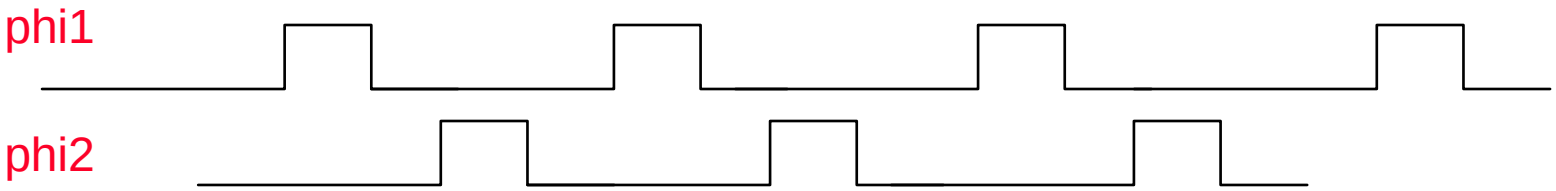
段名	*可能出现的中断问题*
IF	取指页失效、未对准存储器访问、存储保护违例
ID	未定义或非法操作码
EX	算术意外事件
MEM	取数据页失效、未对准存储器访问、存储保护违例、存储器错误

- 产生数据页失效的Load指令、产生指令页失效的 Add指令?
- 解决方案1: 中断向量/指令
- 解决方案2: 尽可能早地中断执行，之后，重启所有未执行完的操作

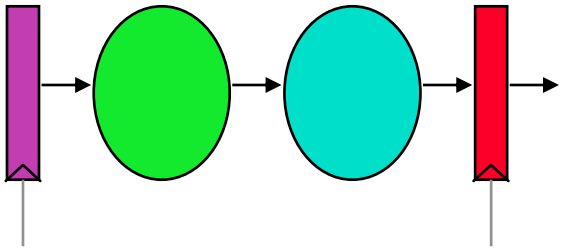
解决方案：上部冻结 & 下部加空泡



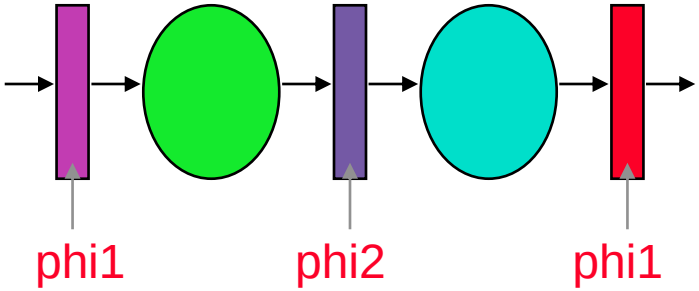
参考：MIPS R3000的时钟定时策略



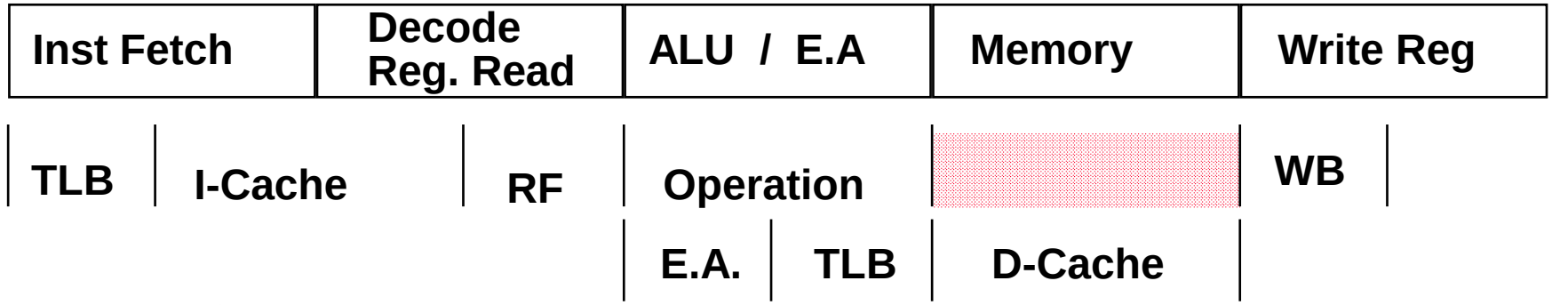
- 双相无重叠时钟（**2-phase non-overlapping clocks**）
- 流水线段两级锁存（电平使能）



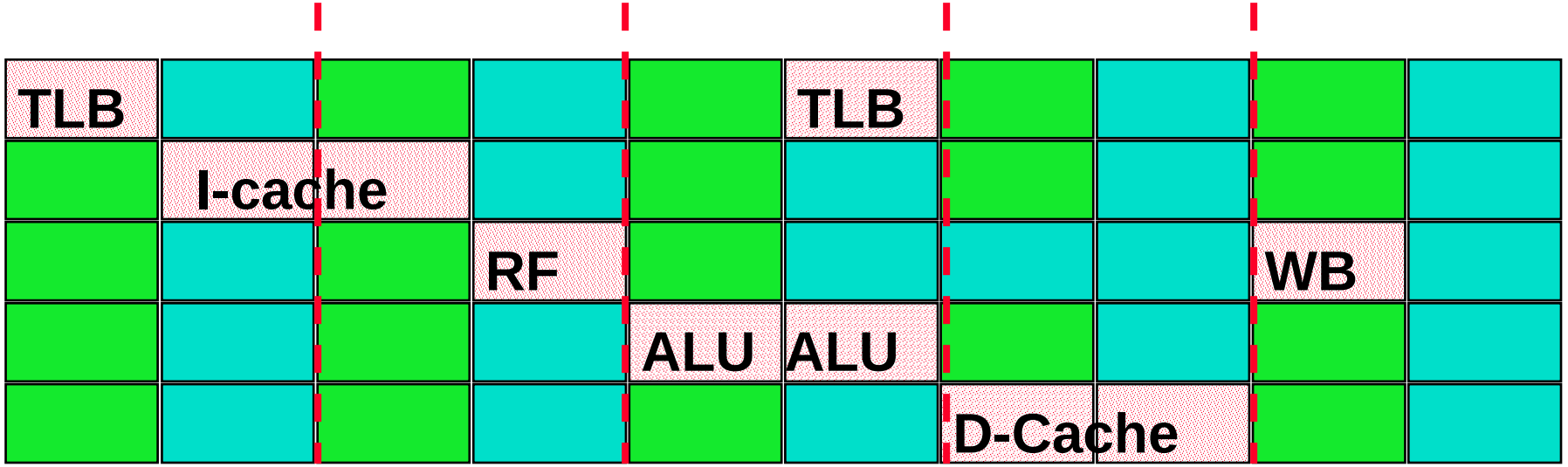
边沿触发



MIPS R3000 指令流水线



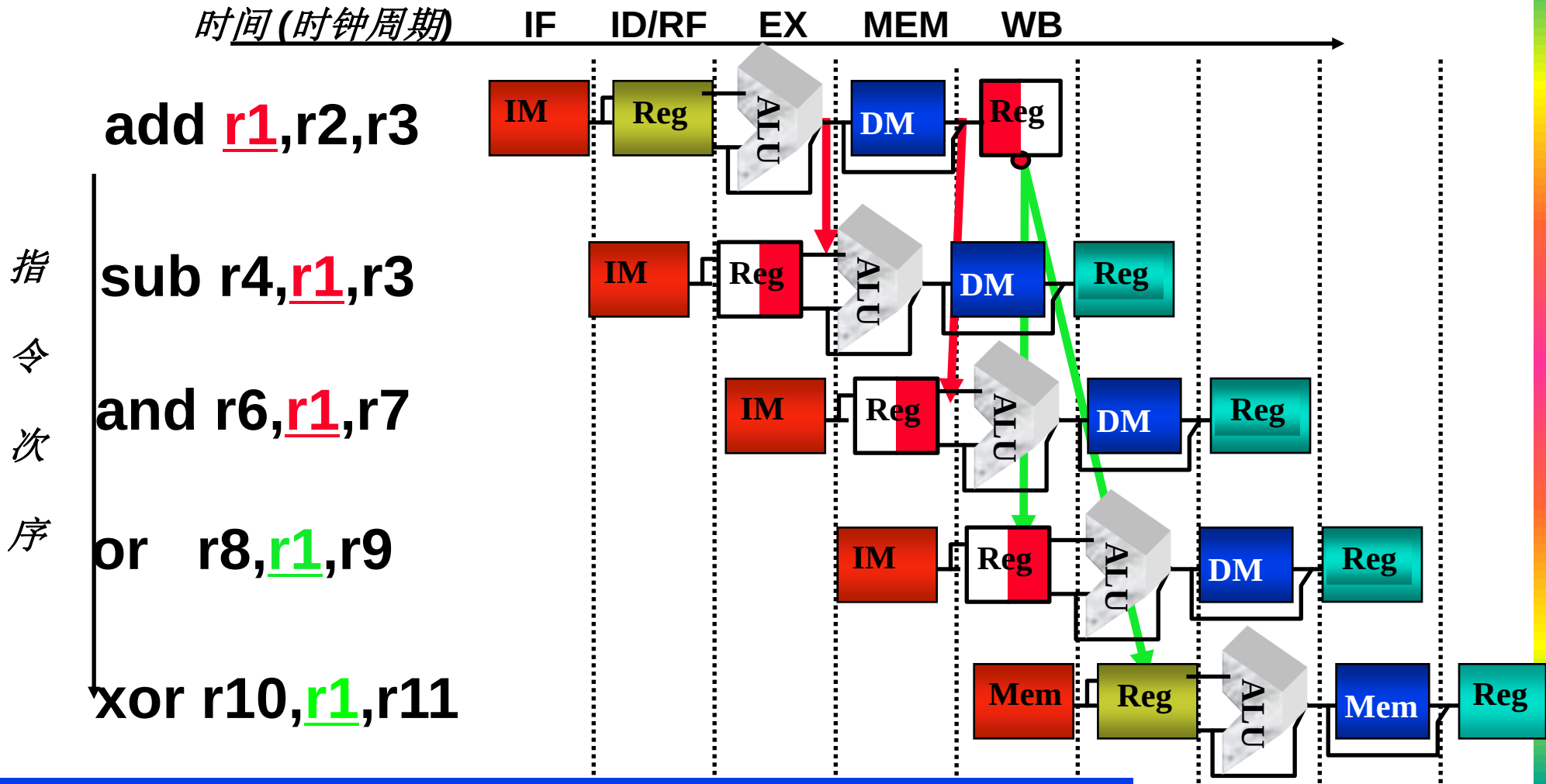
使用资源情况



在第一相进行写，在第二相进行读 => 消除了从WB段的旁路

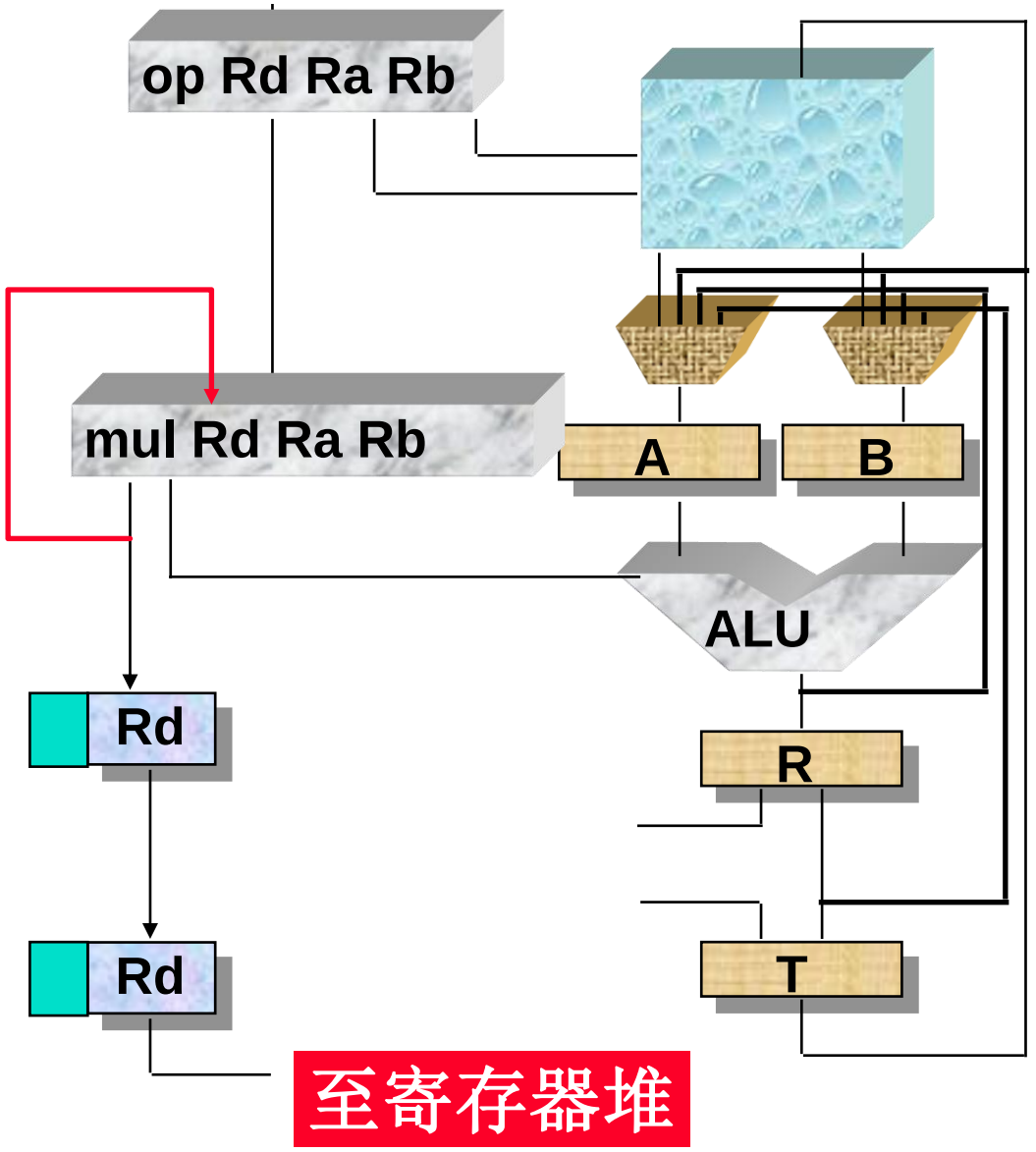
关于rl的数据冒险

- 立即向后相关就可能出现冒险



在MIPS R3000流水线中， 无需从WB段进行前递！

MIPS R3000 的多周期操作



例如: 乘法、除法、Cache失效

暂停流水线中多周期操作之上的所有流水段

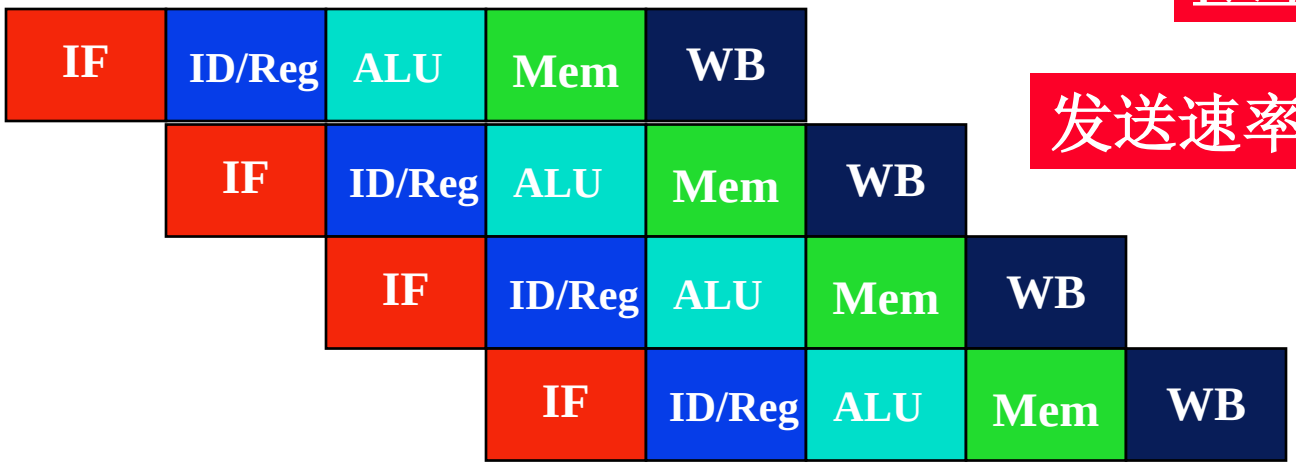
排空（空泡）它之下的所有段

使用本地流水段状态的控制字来一步步执行多周期操作

流水化设计中的指令发送

基本流水线

限制

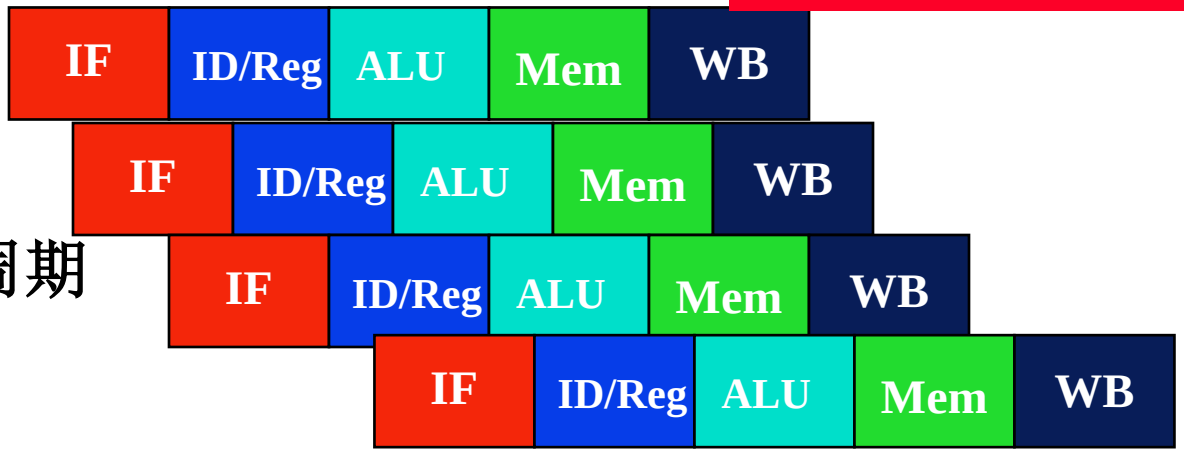


发送速率、FU 暂停、FU 深度

超级流水线

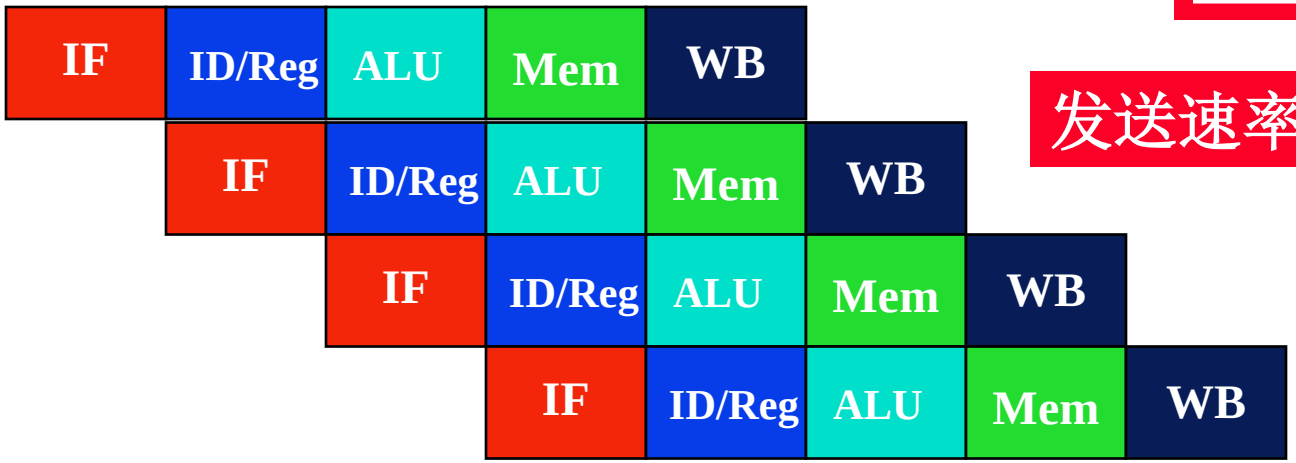
时钟纽斜、FU暂停、FU深度

- 每个内部周期发送一条指令
- ALU 需要多个周期



流水化设计中的指令发送（续一）

基本流水线



限制

发送速率、FU 暂停、FU 深度

超标量

- 每个周期发送多条标量指令

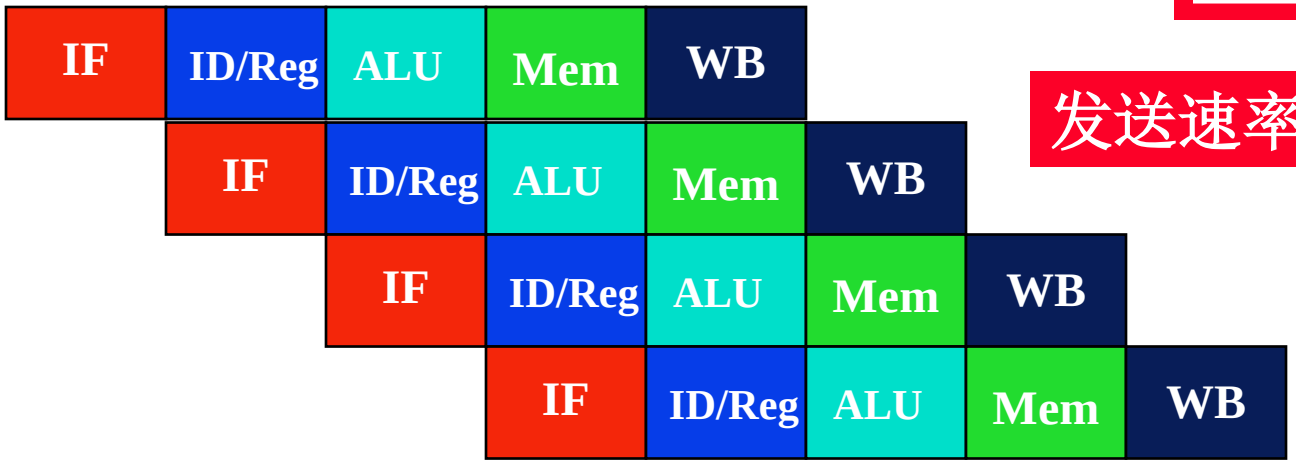
指令间相关处理

流水化设计中的指令发送（续二）

基本流水线

限制

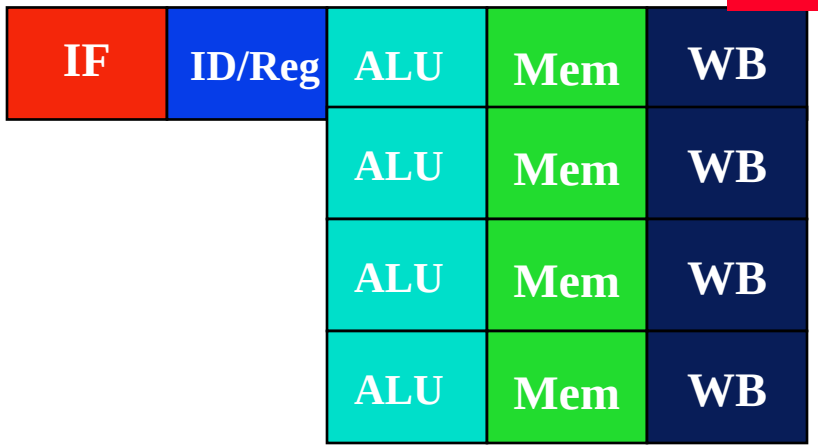
发送速率、FU 暂停、FU 深度



超长指令字

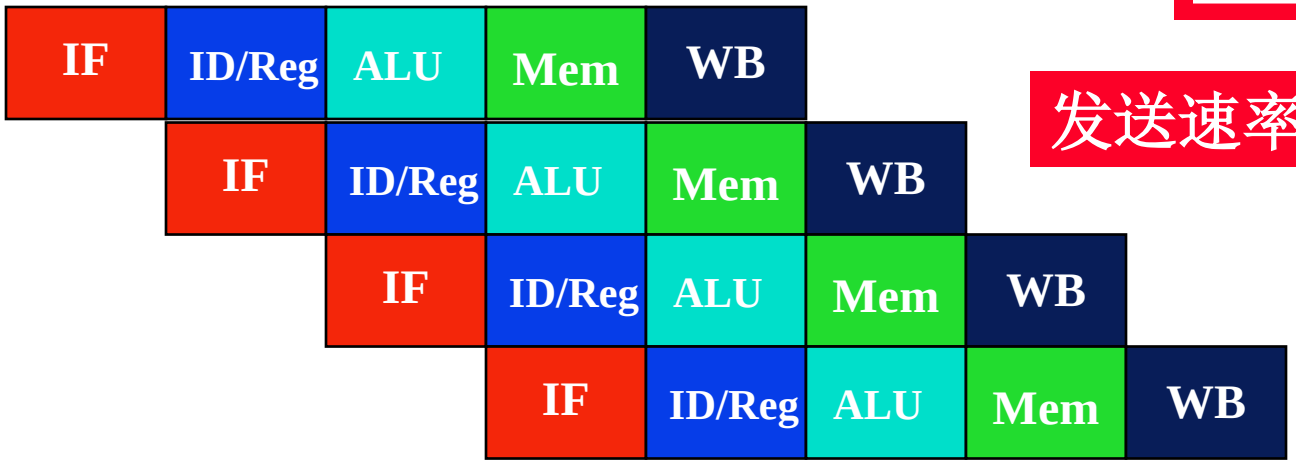
指令封装

- 每条指令指明多个标量操作
- 编译程序判定指令的并行情况



流水化设计中的指令发送（续三）

基本流水线

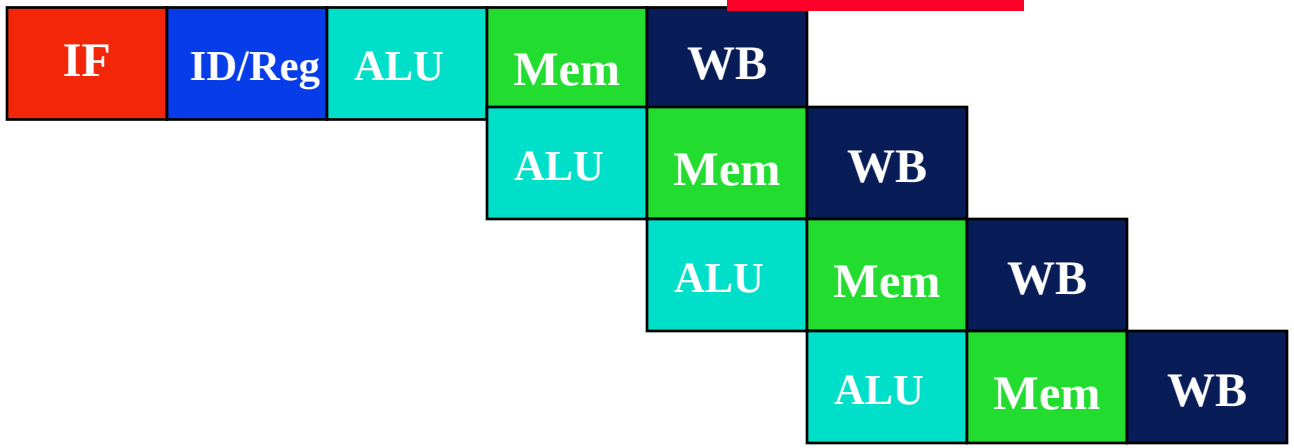


限制

发送速率、FU 暂停、FU 深度

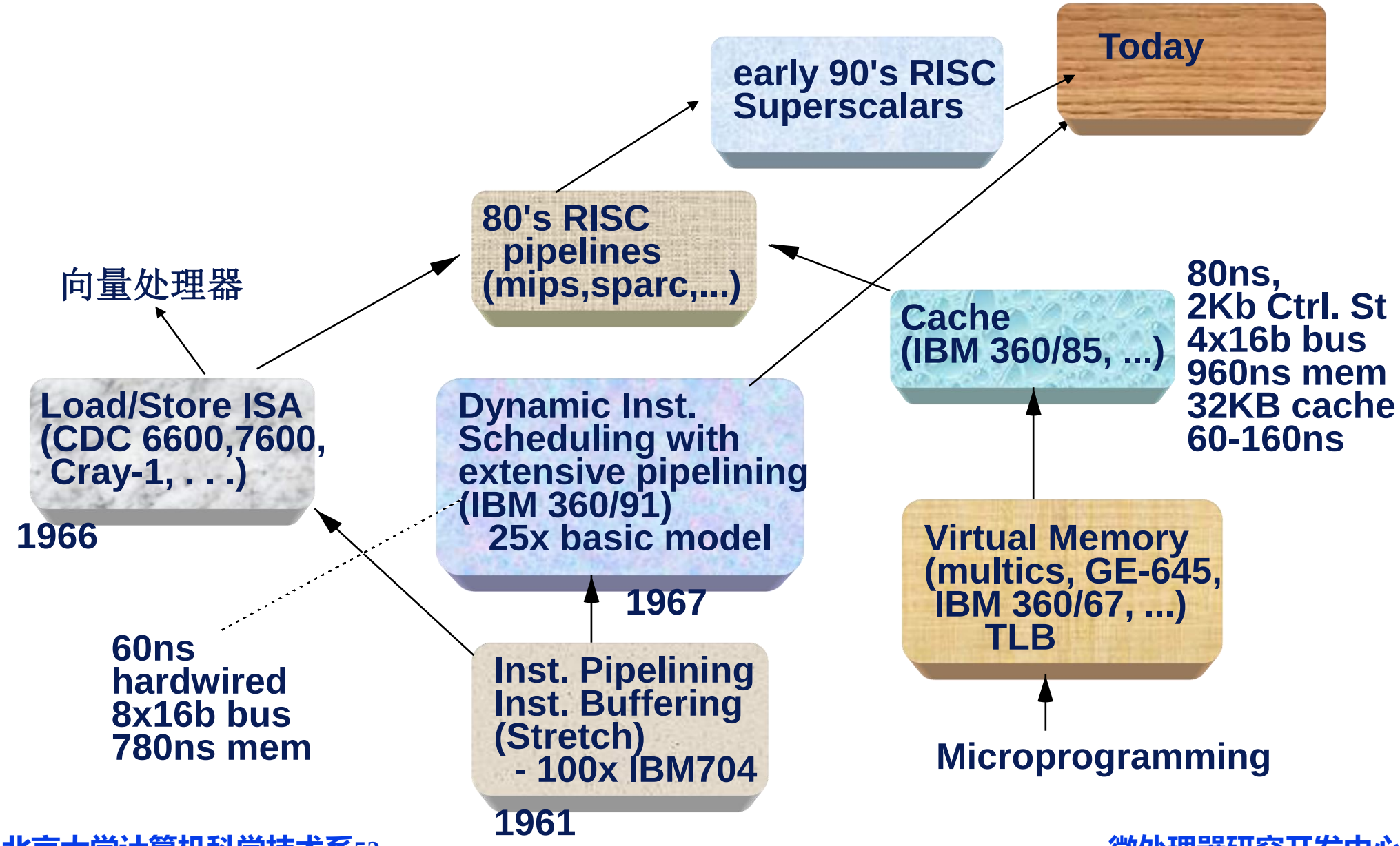
向量流水线

应用领域



- 每条指令指令一串相同的操作

发展历史


Today
early 90's RISC Superscalars
80's RISC pipelines (mips,sparc,...)
向量处理器
Load/Store ISA (CDC 6600,7600, Cray-1, . . .)
1966
Dynamic Inst. Scheduling with extensive pipelining (IBM 360/91) 25x basic model
1967
Cache (IBM 360/85, ...)
80ns, 2Kb Ctrl. St 4x16b bus 960ns mem 32KB cache 60-160ns
Virtual Memory (multics, GE-645, IBM 360/67, ...) TLB
Microprogramming
60ns hardwired 8x16b bus 780ns mem
Inst. Pipelining Inst. Buffering (Stretch) - 100x IBM704
1961

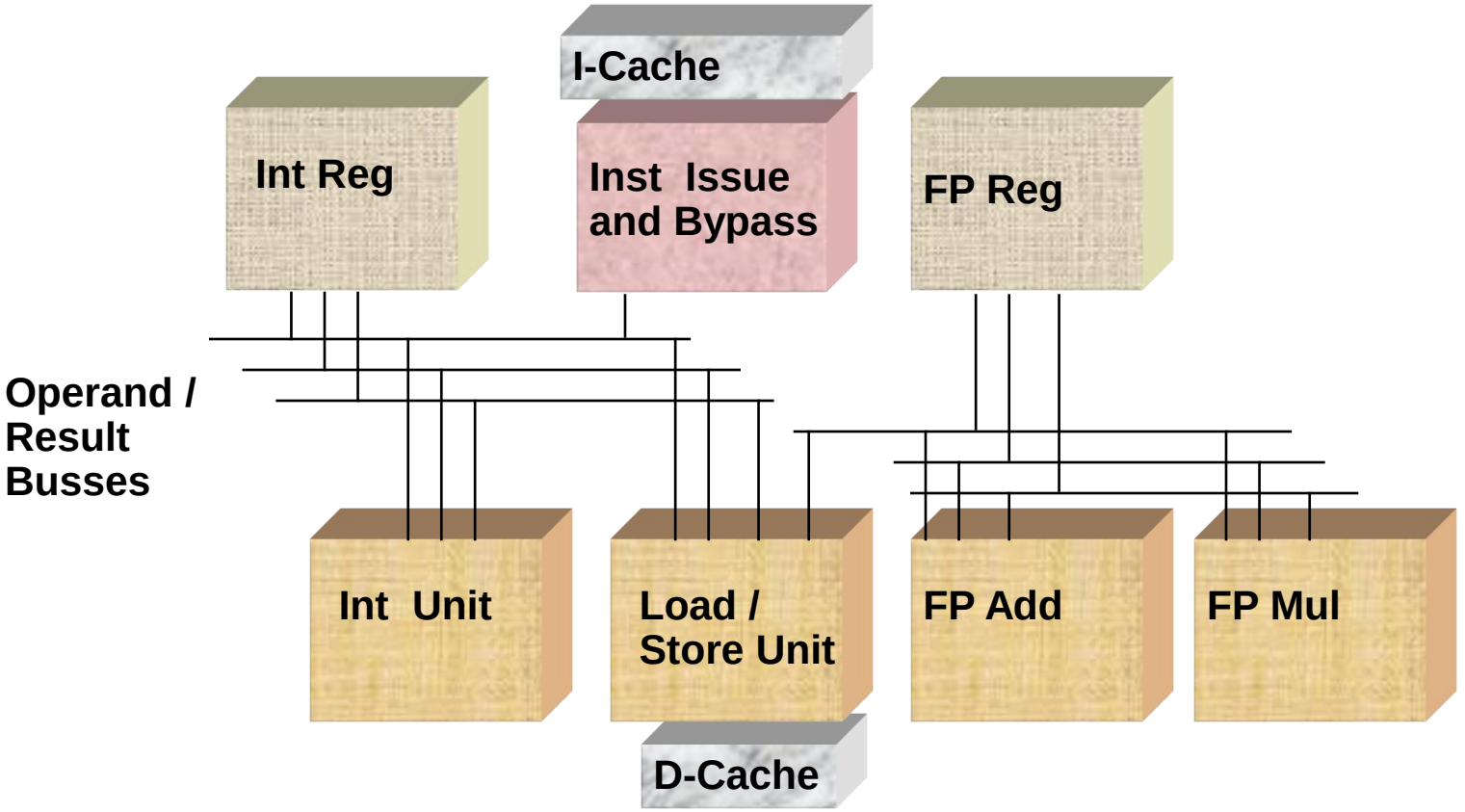
技术工艺的发展

存储器容量 60%/年

—每三年，翻两番

多指令发送（简单的超标量）

不相关整数指令和浮点指令被发送到不同的流水线



单发射机制的总执行时间 = Int Time + FP Time

最大加速比：
$$\frac{\text{单发射机制的总执行时间}}{\text{MAX(Int Time, FP Time)}}$$

示例： DAXPY

Basic Loop:

load	Ra ⇐ Ai
load	Ry ⇐ Yi
fmult	Rm ⇐ Ra×Rx
fadd	Rs ⇐ Rm+Ry
store	Ai ⇐ Rs
inc	Yi
dec	i
inc	Ai
branch	

单发射机制的总执行周期:
19（7整数, 12浮点)
双发射机制的最小周期:
12
潜在加速比:
1.6 !!!

实际执行周期: 18

循环展开 (Unrolling)

Basic Loop:

load	a ⇐ Ai
load	y ⇐ Yi
fmult	m ⇐ a × s
fadd	r ⇐ m+y
store	Ai ⇐ r
inc	Ai
inc	Yi
dec	i
branch	

9 Insts Per
2 FP ops

6 Insts Per
2 FP ops

Unrolled Loop:

load ;load;
fmult;fadd;
store;
load ;load;
fmult;fadd;
store;
load ;load;
fmult;fadd;
store;
load ;load;
fmult;fadd;
store;
inc;inc;dec;
branch

指令
相关
不变

循环展开（续）

Unrolled Loop:

load ;load;
fmult;fadd;
store;
load ;load;
fmult;fadd;
store;
load ;load;
fmult;fadd;
store;
load ;load;
fmult;fadd;
store;
inc;inc;dec;
branch

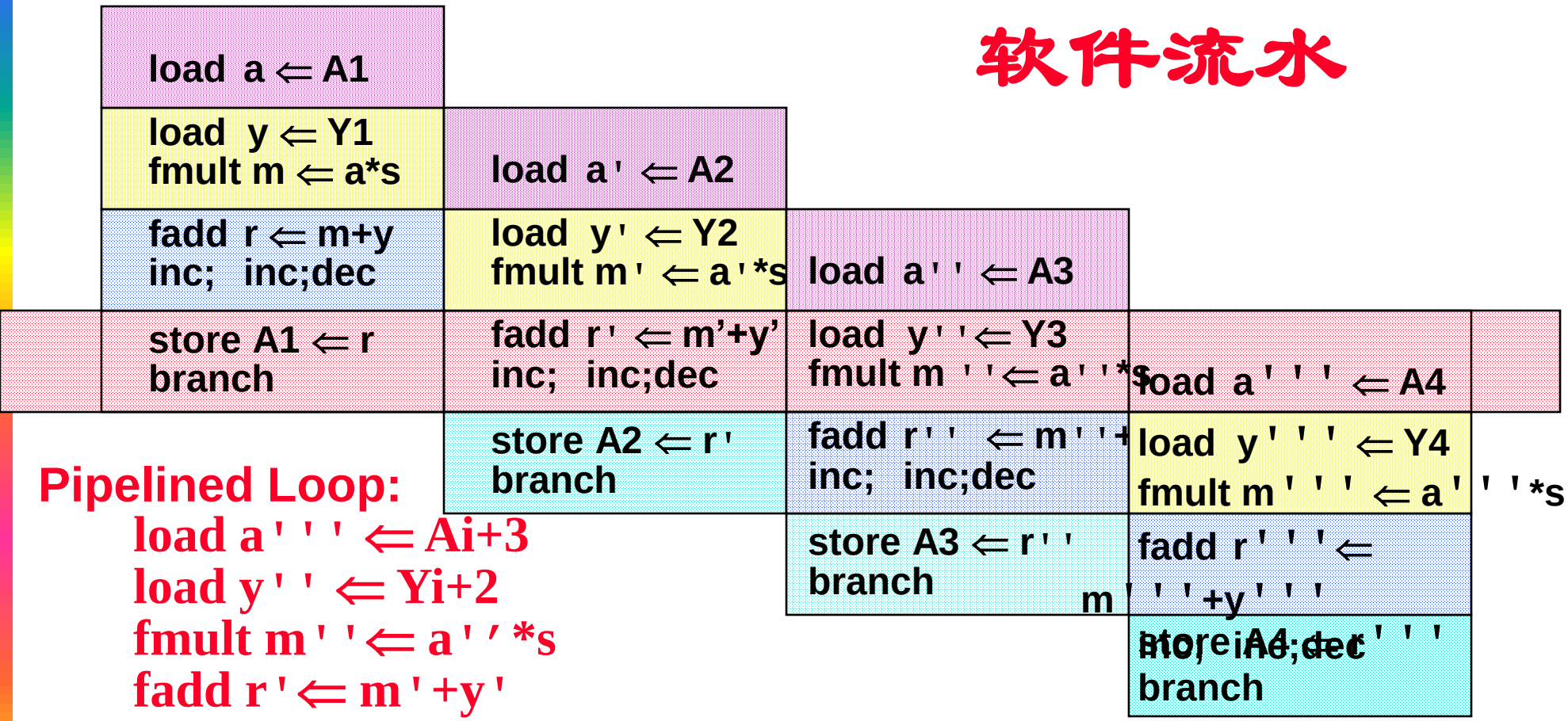
6 Insts Per
2 FP ops

Reordered Unrolled Loop:

load ;load;
load ;load;
load ;load;
load ;load;
fmult;fmult;
fmult;fmult;
fadd;fadd;
fadd;fadd;
store;store;
store;store;
inc;inc;dec;
branch

基本块的大小为24条指令

软件流水

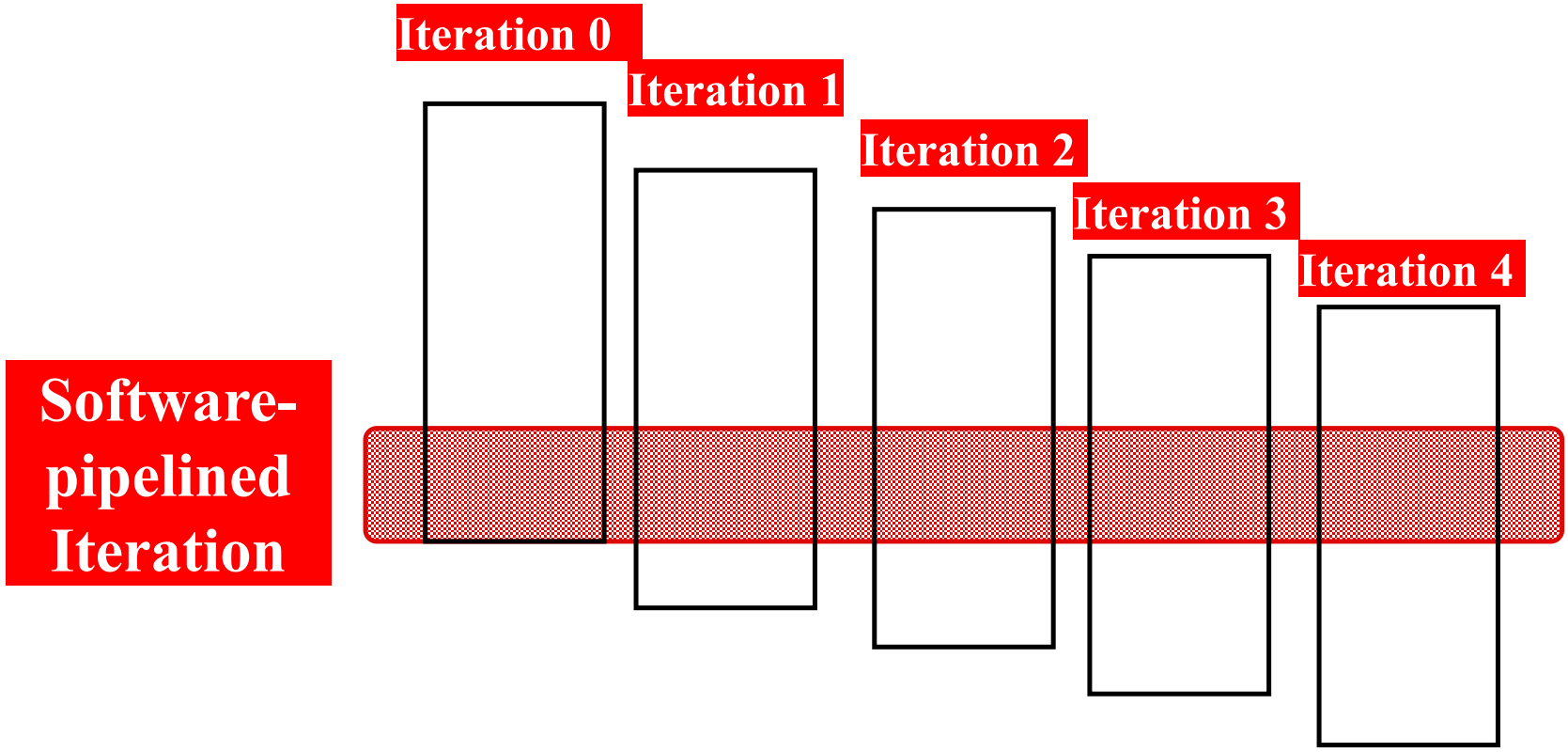


Pipelined Loop:

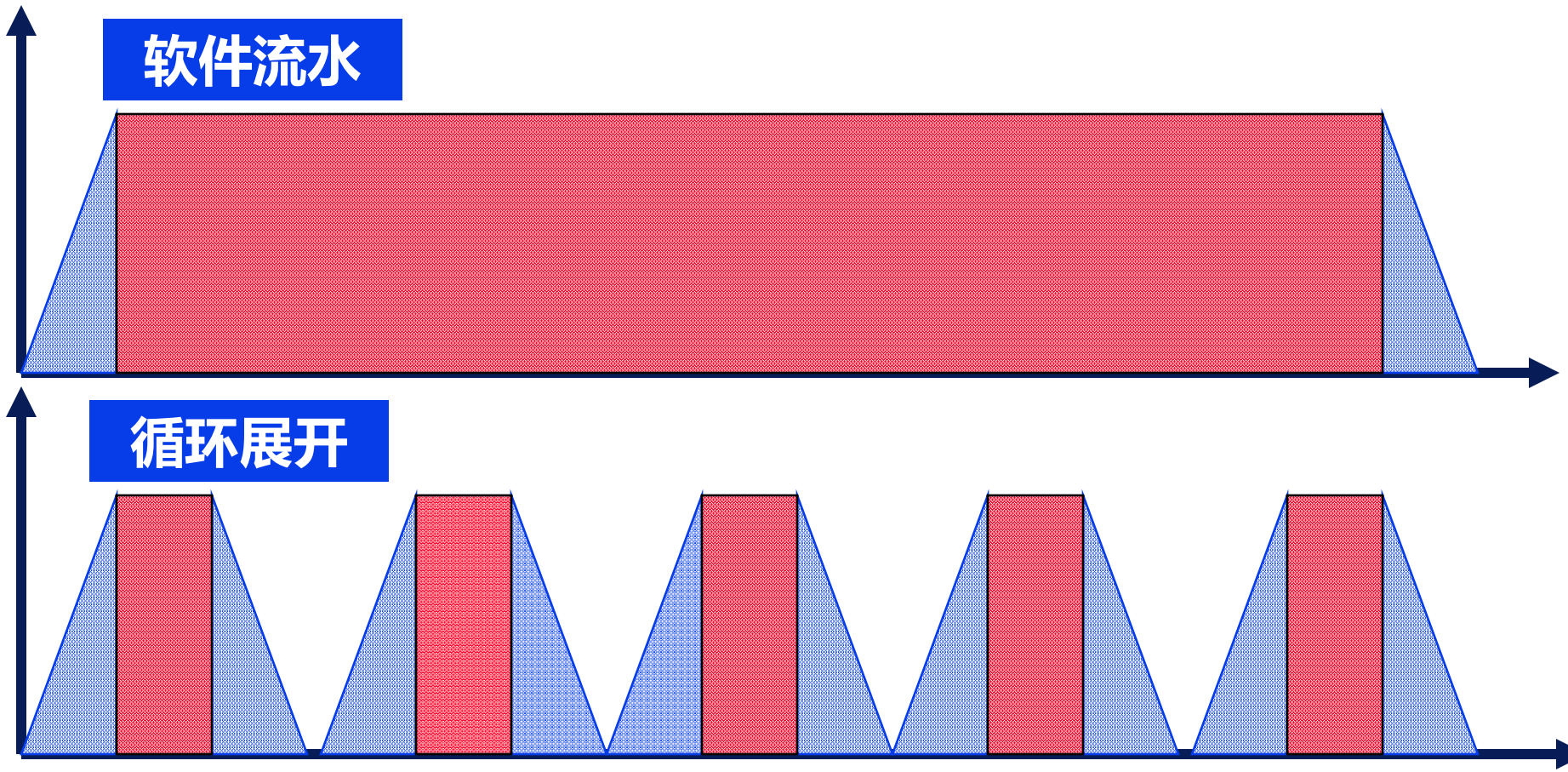
load a''' ⇐ Ai+3
load y'' ⇐ Yi+2
fmult m''⇐ a''*s
fadd r'⇐ m'+y'
store Ai ⇐ r
inc Ai+3
inc Yi
dec i
a'' ⇐ a'''; Y' ⇐ y''; m'⇐m''; r ⇐r'
branch

软件流水技术

- 发现: 如果循环的每次迭代之间没有相关, 那么通过从不同的迭代中抽取指令来获得更高的指令级并行性。
- 软件流水：对循环进行重构，使得每次迭代执行的指令是属于原循环的不同迭代过程的。

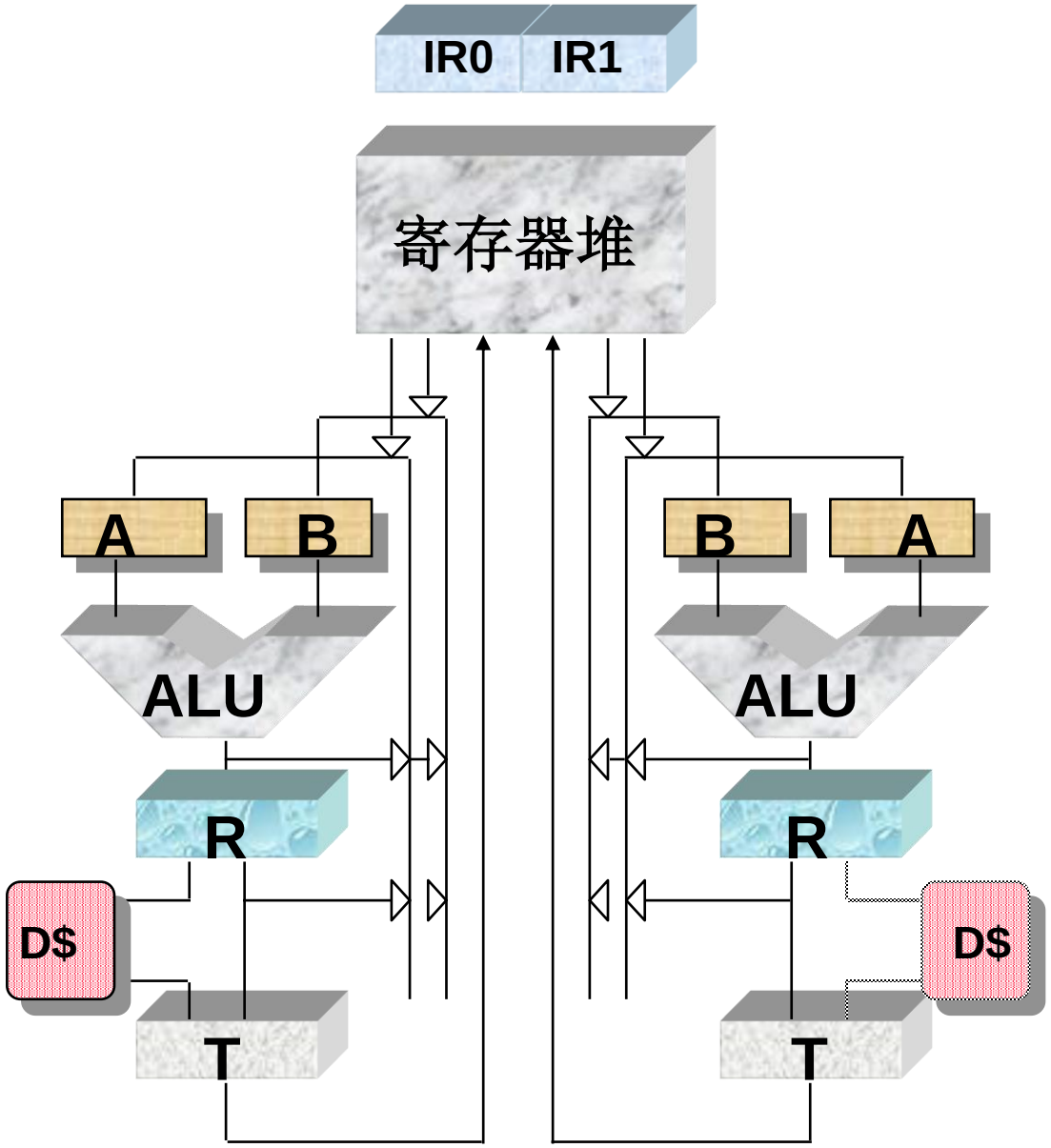


循环展开和软件流水的示意图



- 软件流水
  - 代码空间较小
  - 只需填充和排空流水线一次
    而循环展开每次迭代就需要一次

多流水线/较难的超标量



问题:

寄存器堆端口

检测数据相关

旁路

RAW冒险

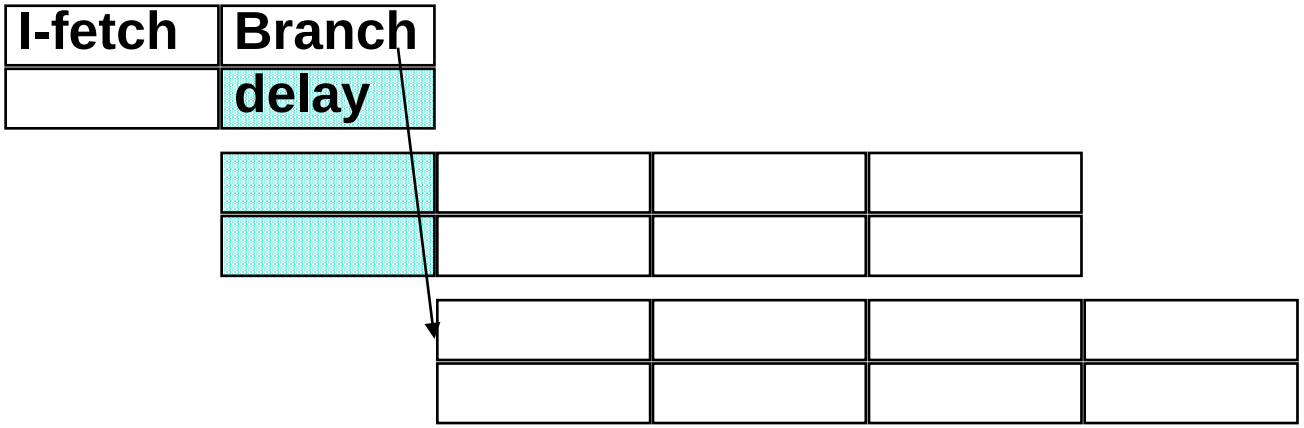
WAR冒险

多装入/存储操作?

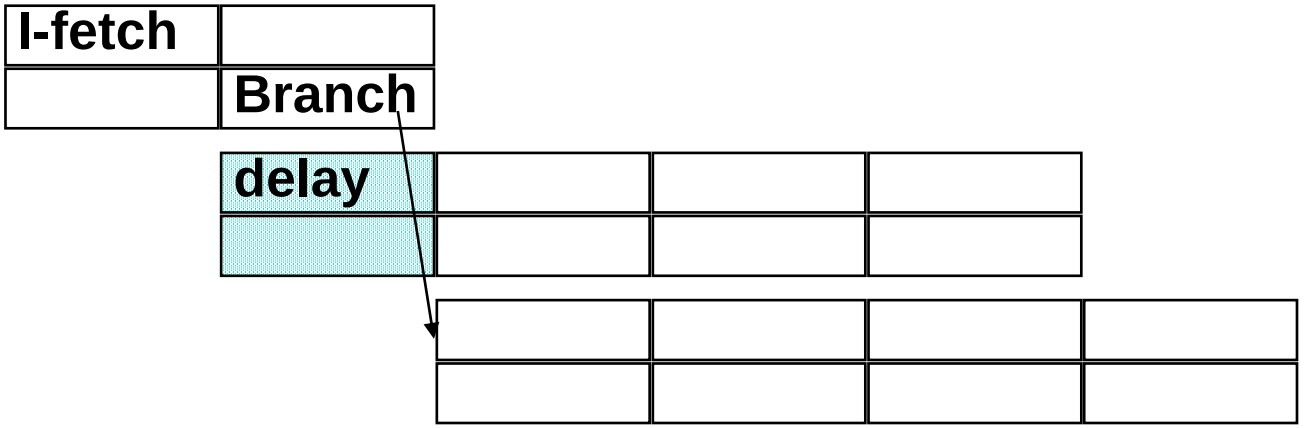
转移

超标量结构中的转移损失

例如: 在取操作数阶段解决，单延迟槽



Squash 2



Squash 1

总结

- 流水线向下传递控制信息，就象向下传递数据一样
- 通过局部控制解决 前递/暂停
- 意外事件会导致流水线停止
- MIPS指令系统体系结构中流水线是可见的(延迟转移、延迟装入)
- 更深的流水线、更多的并行度可能获得出更高的性能