计算机组织与系统结构

流水技术引论

Introduction to Pipelining

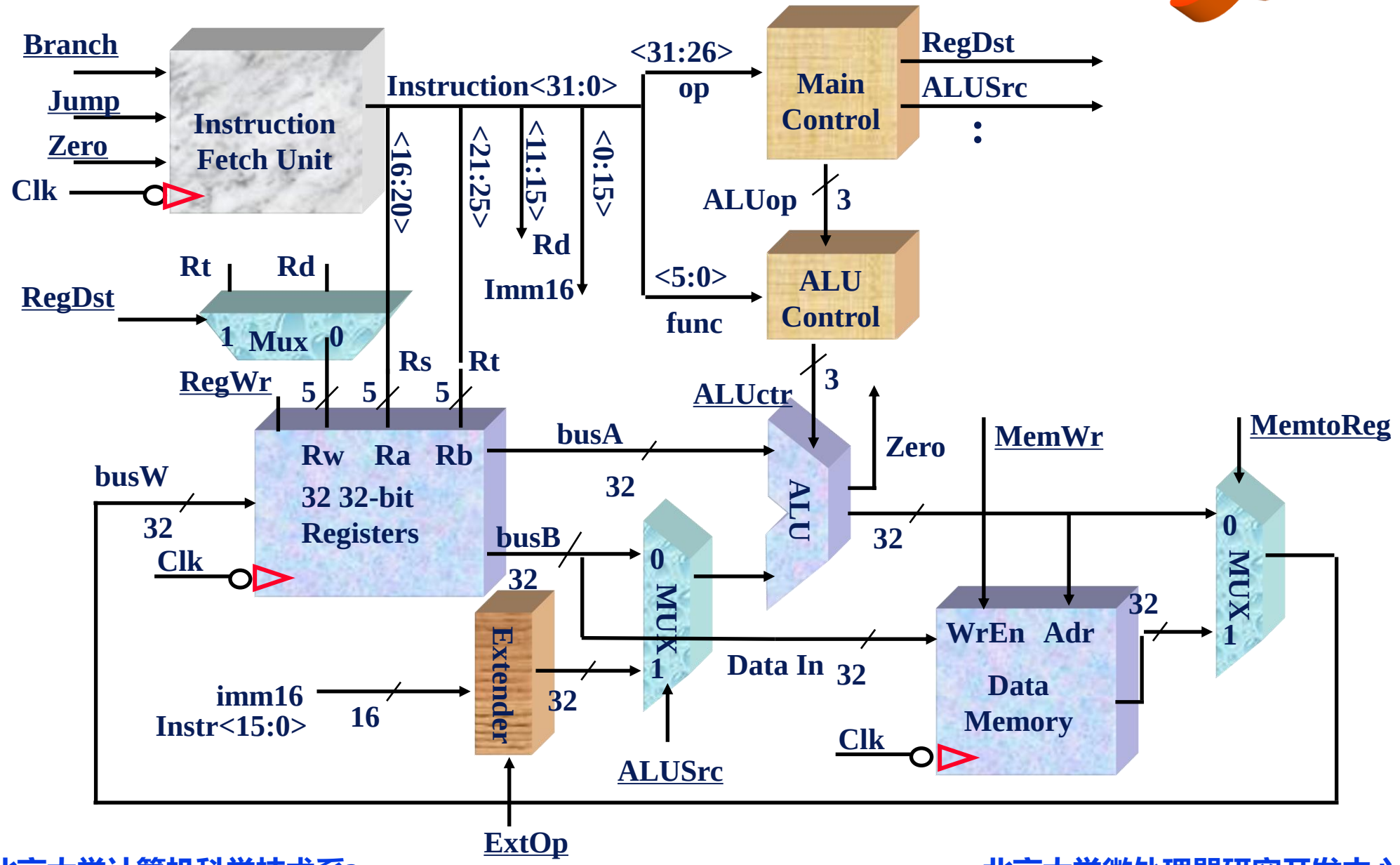
（第十一讲）

程 旭

2020.12.10

单周期处理器


复习


Branch
Jump
Zero
Clk
Instruction Fetch Unit
Instruction<31:0>
<31:26>
op
Main Control
RegDst
ALUSrc
:
<16:20>
<21:25>
<11:15>
Rd
<0:15>
Imm16
ALUop
3
<5:0>
func
ALU Control
Rt
Rd
RegDst
1 Mux 0
RegWr
5
5
5
Rs
Rt
ALUctr
3
busA
32
Rw Ra Rb
32 32-bit Registers
busW
32
Clk
busB
32
0 MUX 1
ALU
Zero
MemWr
MemtoReg
32
0 MUX 1
32
WrEn Adr
Data In
32
Data Memory
Clk
imm16
Instr<15:0>
16
Extender
32
ALUSrc
ExtOp

该单周期处理器的缺陷

- 时钟周期时间长:
  - 对于装入指令，周期时间必须足够长:

    PC的Clock -to-Q +

    指令存储器访问时间 +

    寄存器堆访问时间 +

    ALU延迟（地址计算） +

    数据存储器访问时间 +

    寄存器对建立时间 +

    时钟纽斜

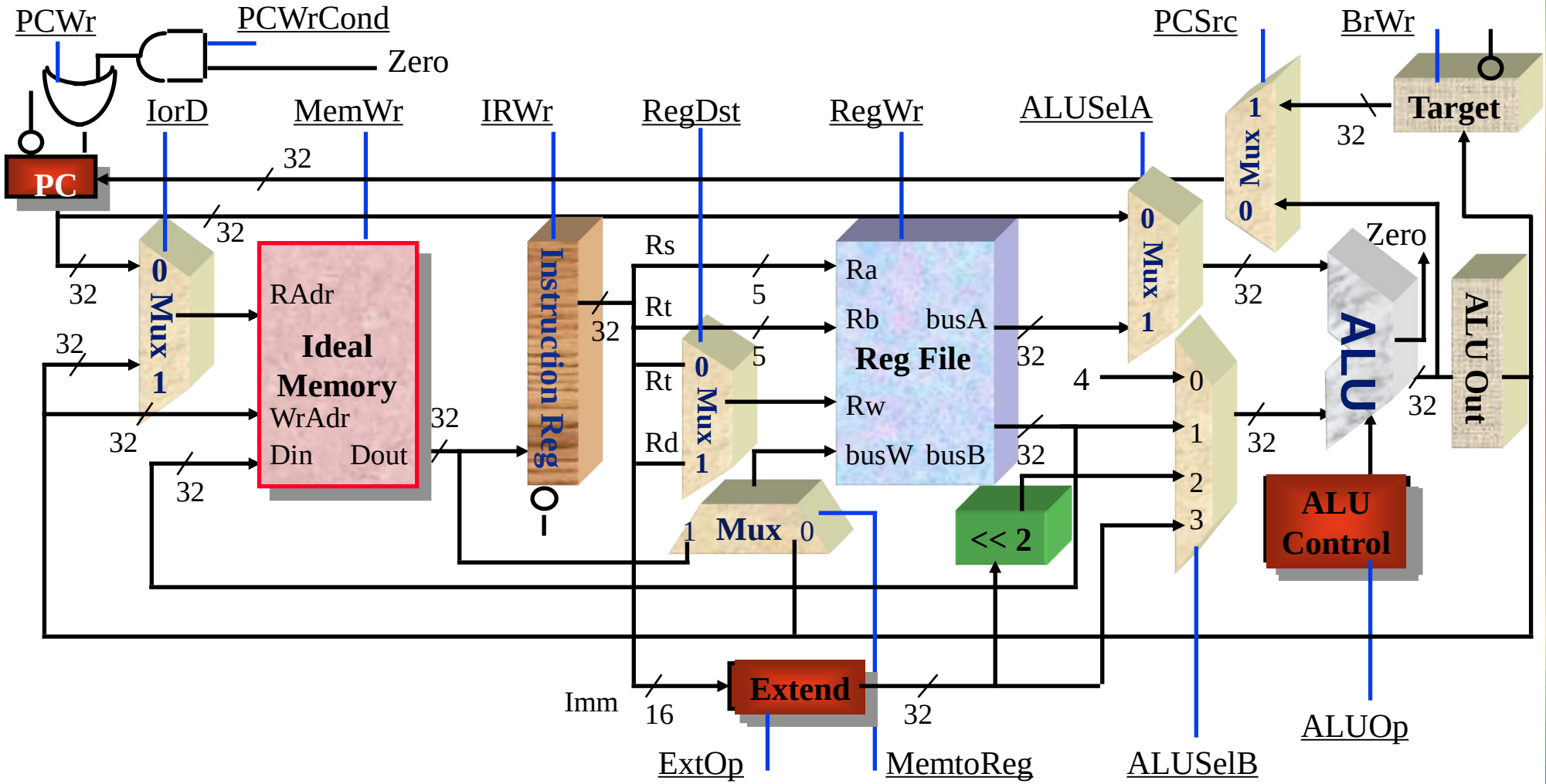
- 对于所有其他指令，周期时间都比所需的要长很多!

多周期实现概述

- 单周期处理器的问题根源：
  - 对于最慢的指令，周期时间必须足够长
- 解决方案：
  - 将指令处理分为更小的步骤
  - 每个周期执行一步（而不是整个指令）
    - 周期时间： 执行最长步所需的时间
    - 使所有的步骤尽量具有相同的长度
  - 这个多周期处理器的本质所在
- 多周期处理器的优点：
  - 周期时间非常短
  - 不同的指令需要不同的周期数来完成
    - 装入需要 5 个周期
    - 跳转仅仅需要3个周期
  - 允许每条指令多次使用同一个功能部件

多周期处理器

- 在每条指令的执行过程中，可以多次使用同一功能部件



微程序设计

- 精心设计的状态图易于用微序列器实现
  - 简单地递增 和 转移场位
  - 数据通路控制场位
- 控制设计可以简化为微程序设计
- 微程序设计是一个基本概念
  - 通过建造一个非常简单地处理器，并对指令进行 解释执行 ， 来实现指令系统
  - 特别适用于非常复杂的指令，以及寄存器传输比较少的情况

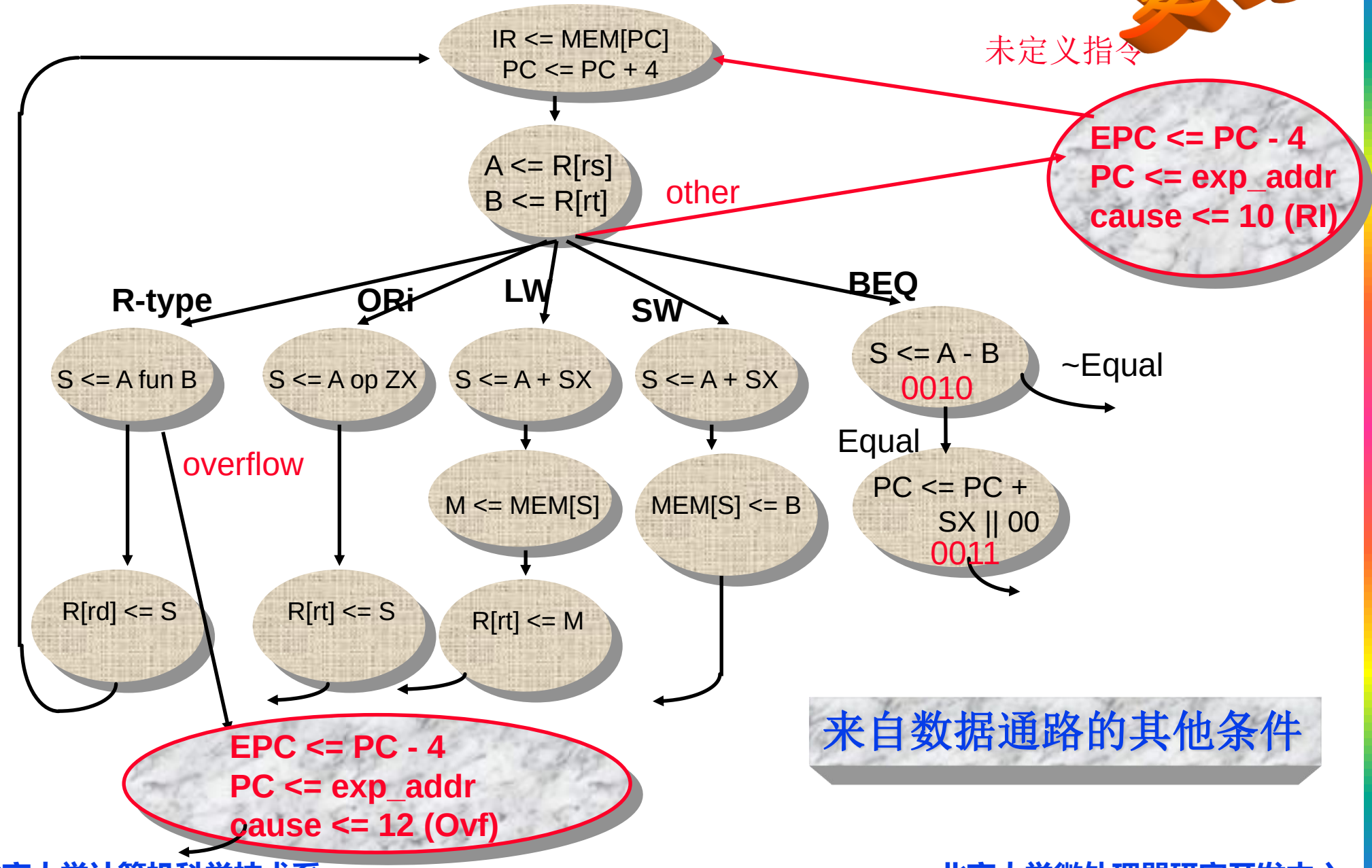
微程序设计对RISC产生的灵感

- 如果简单指令能够以很高的时钟频率执行
- 如果人们有能力编写产生微指令的编译器
- 如果绝大多数程序仅仅使用简单指令和简单的寻址方式
- 如果微码保存在RAM中，而非ROM中，那么就易于排错
- 如果用于控制存储器的同一存储器还可被用为宏指令的 cache
- 那么，为什么不用微程序和可直接产生机器最低级语言的编译器来跨越指令解释器呢？

意外事件和中断

- 意外处理是控制部分的难点
- 需要找到可以保存PC、并激活操作系统功能的方便地方来放置检测意外事件、转移到状态或微指令
- 当以后我们学习支持存储访问出现页面失效的流水化CPU时，由于指令不能完成，并且 恰好要在产生意外事件的指令处重启程序，这使得控制的设计难上加难！

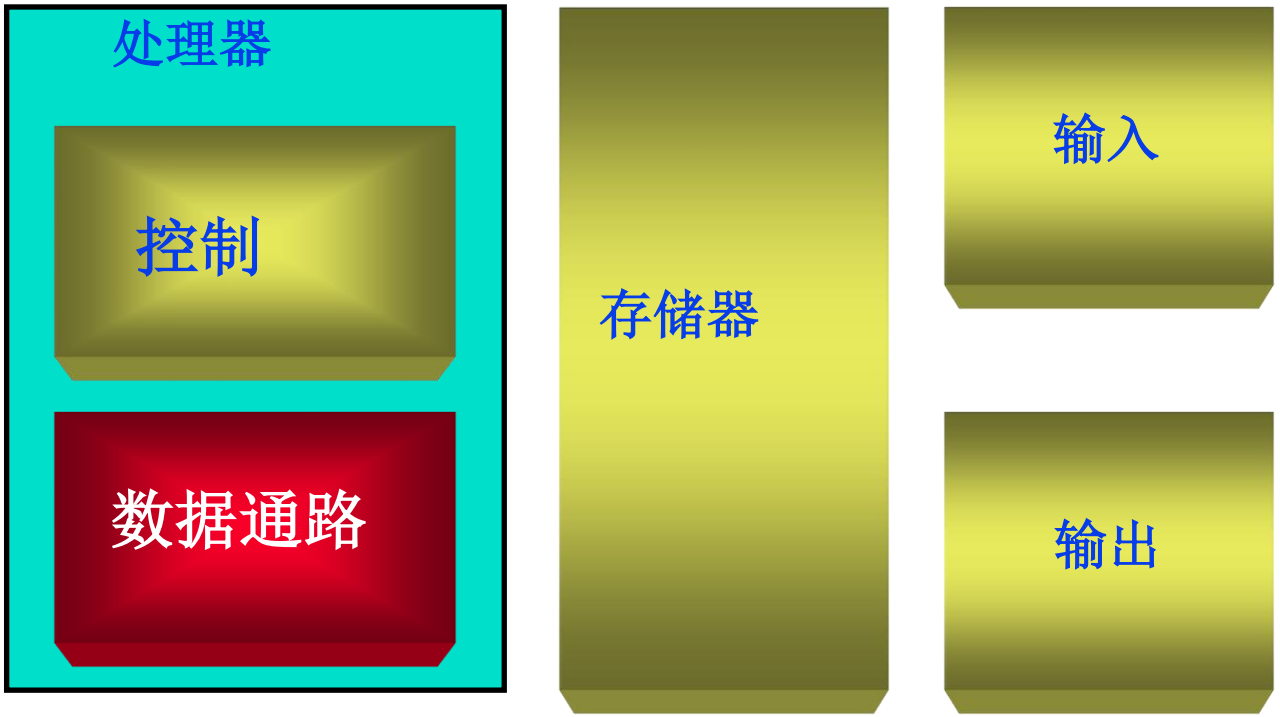
对控制描述进行修改



来自数据通路的其他条件

教学目标：已经掌握的内容

- 计算机的五个基本部件

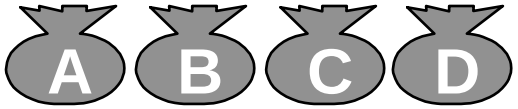


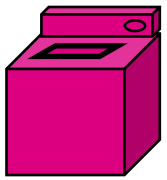
- 本讲主题：流水技术

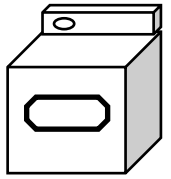
流水技术的思想非常自然！

洗衣房

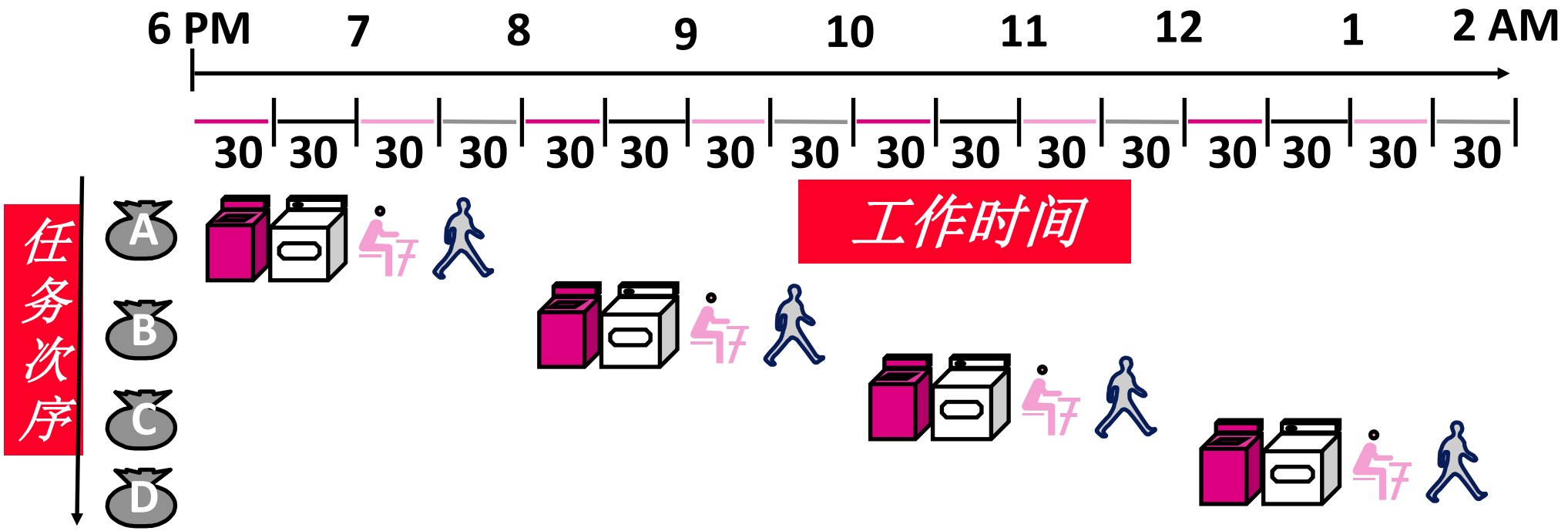
- 张三、李四、王五、赵六每人有一包衣服需要洗涤、烘干、熨整
- 洗衣机需要 30分钟
- 烘干机需要30分钟
- 熨斗需要30分钟
- 洗衣工需要30分钟将衣物放到抽屉里





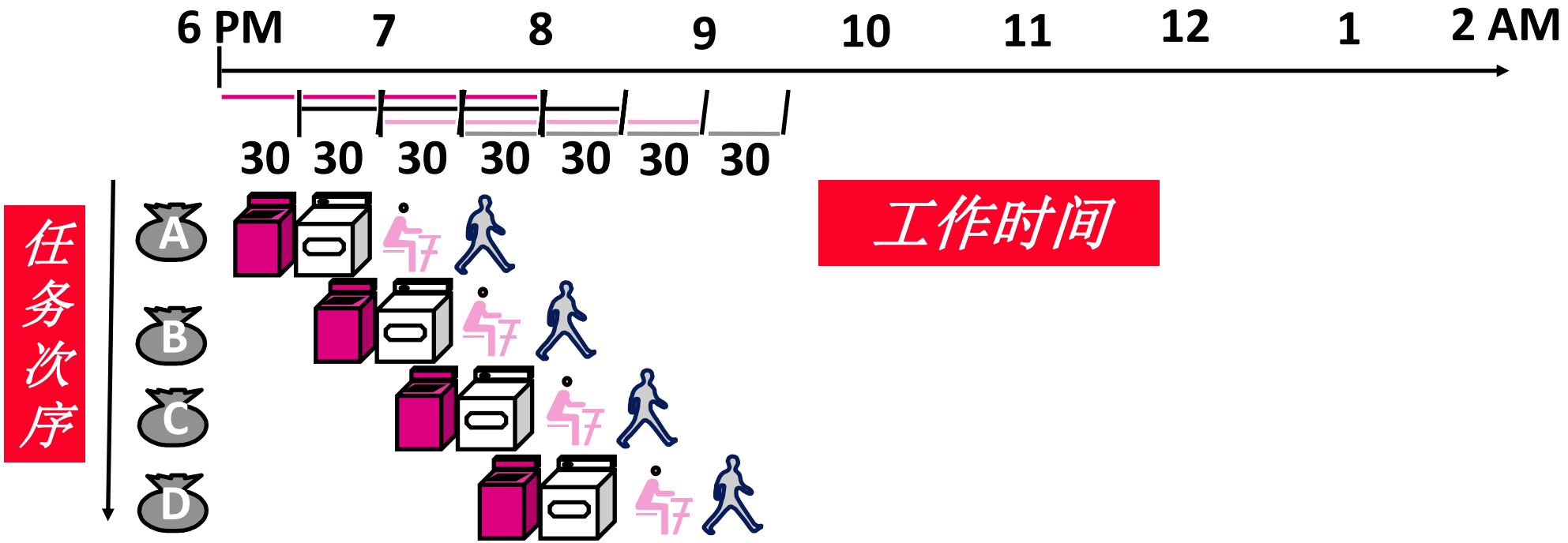


串行洗衣店



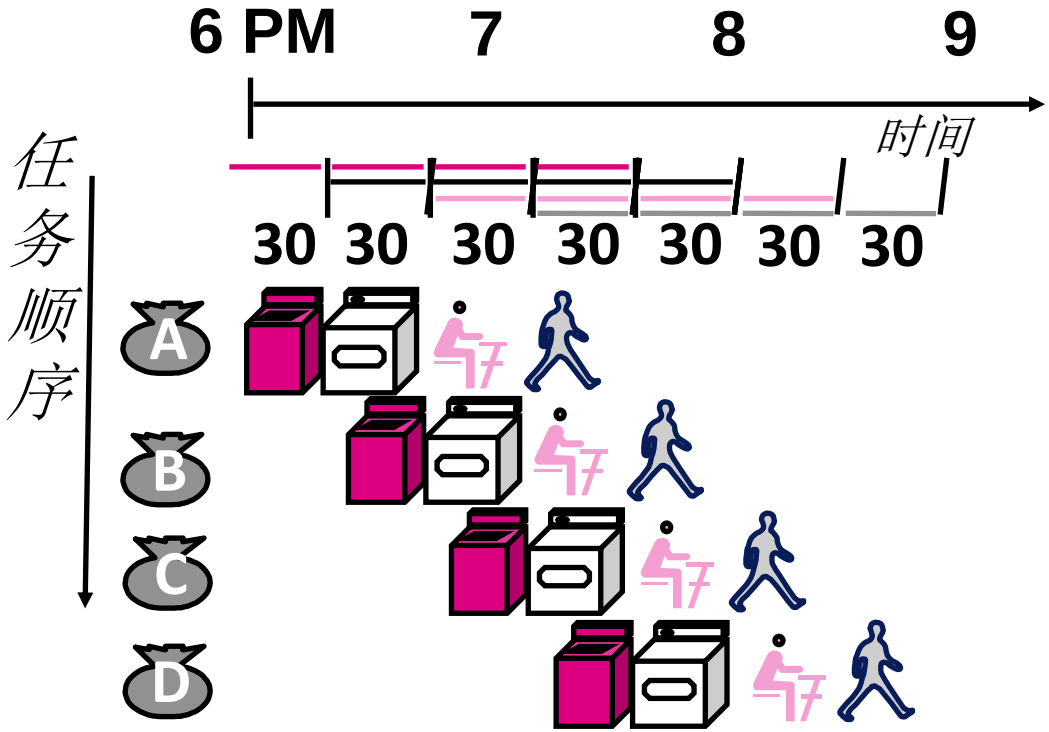
- 串行洗衣店需要8个小时完成4个工作量
- 如果他们了解流水技术，那么需要多长时间完成上述工作呢？

流水化的洗衣店：尽可能早地开始工作



- 流水化洗衣店需要3.5个小时完成4个工作量

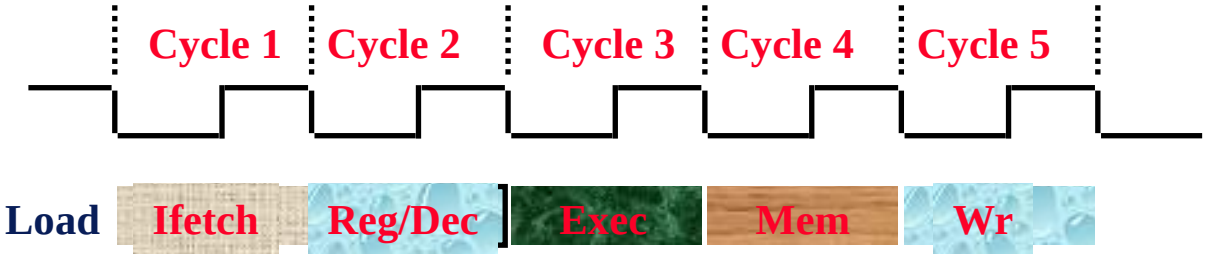
流水技术性质



- 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

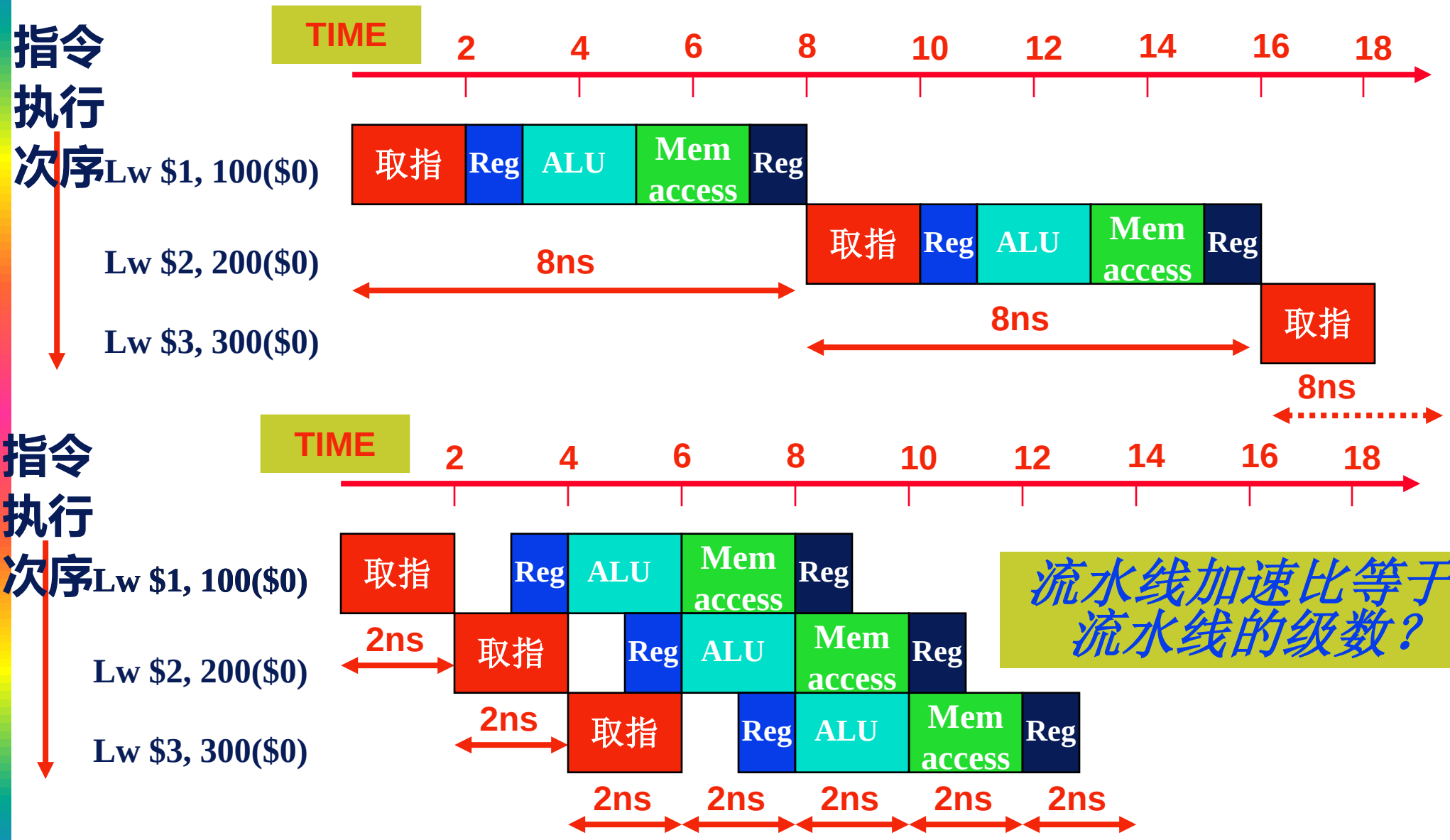
装入操作的五级

- **Ifetch:** 取指
  - 从指令存储器中取出指令
- **Reg/Dec:**取操作数和译码
  - 取寄存器和指令译码
- **Exec:** 执行
  - 计算存储器的地址
- **Mem:** 存储器访问
  - 从数据存储器中读取数据
- **Wr:** 写回
  - 将数据写回到寄存器堆

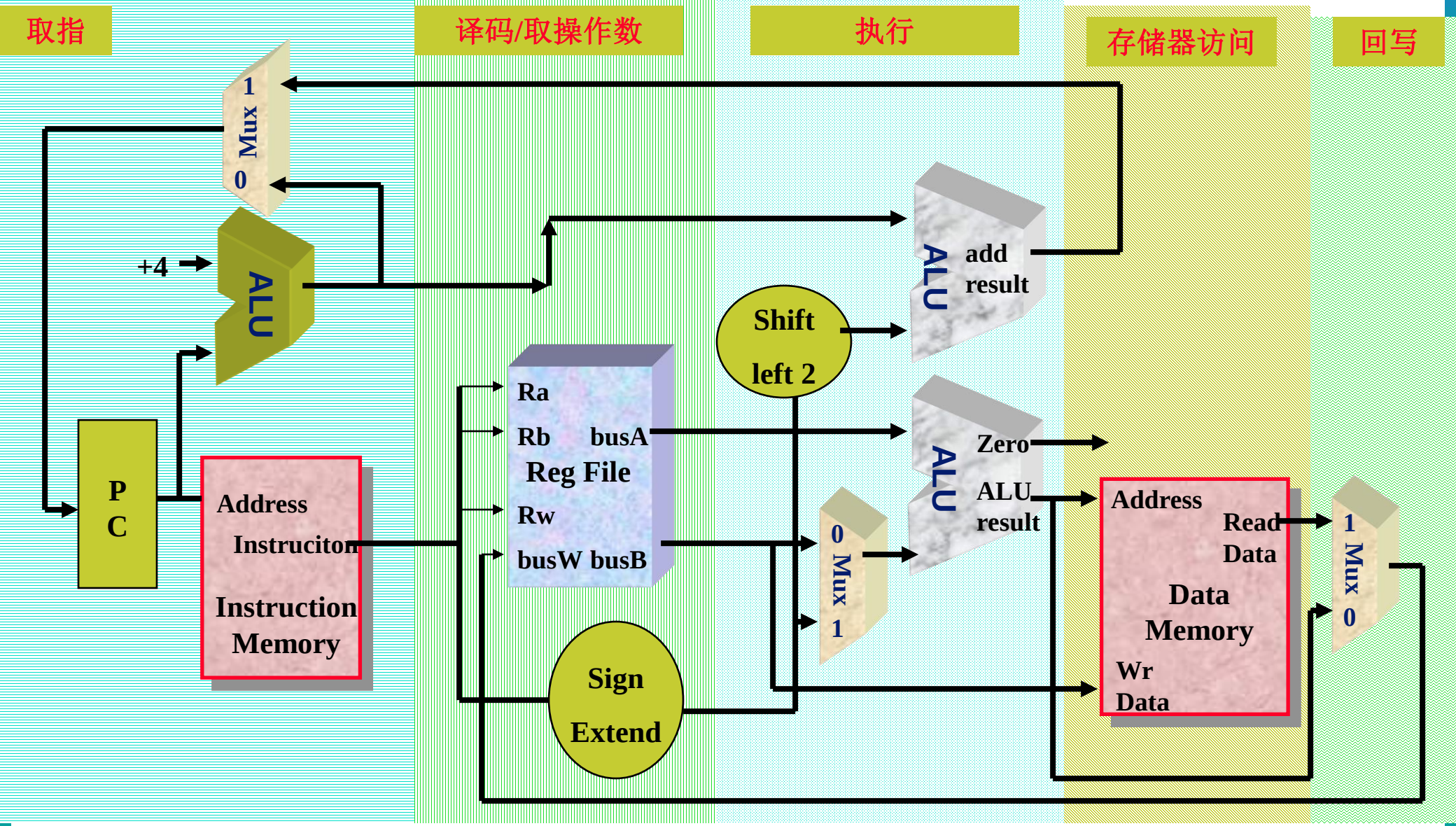


流水技术

- 通过增加指令吞吐率来改进性能

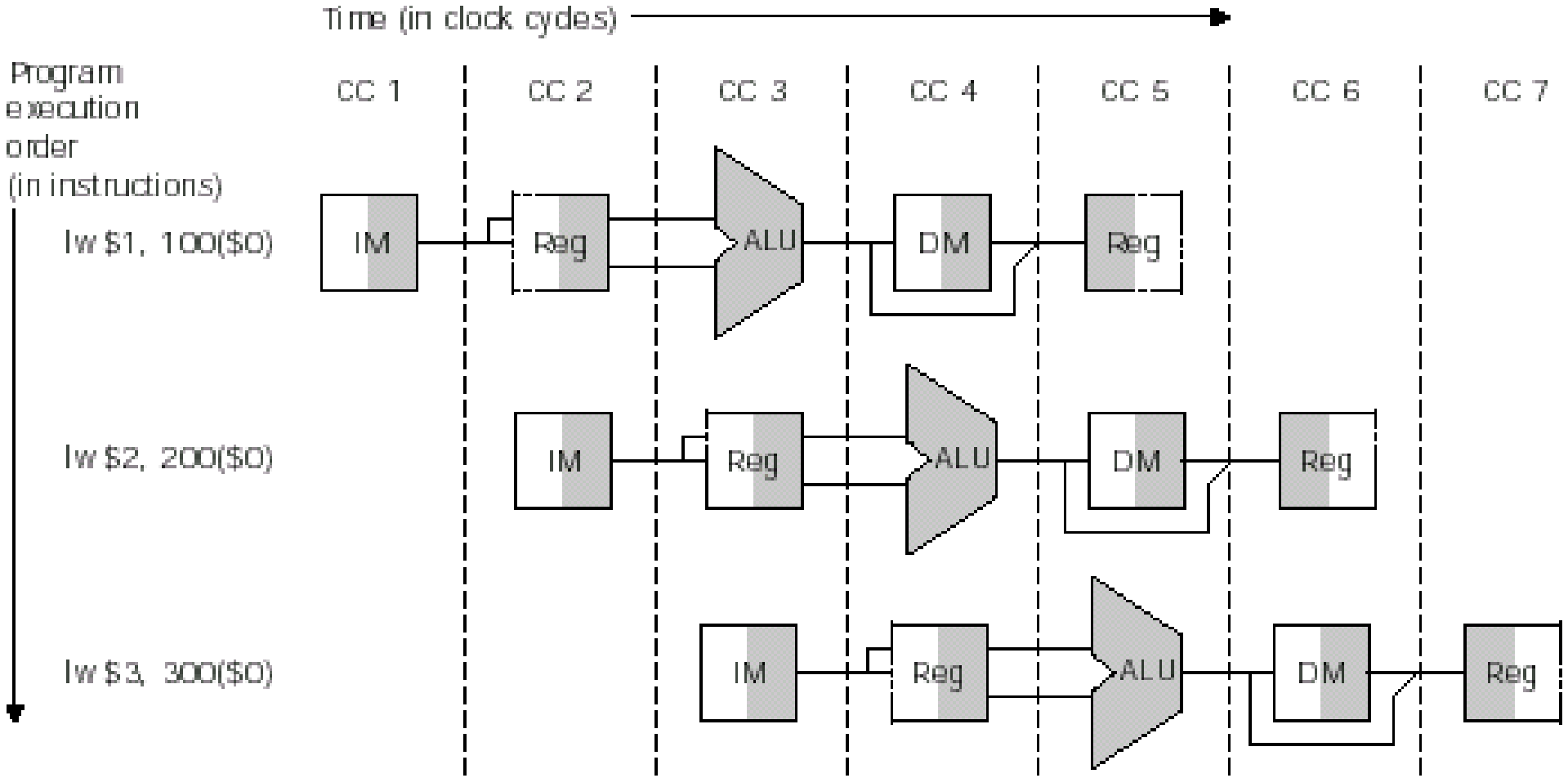


基本思路



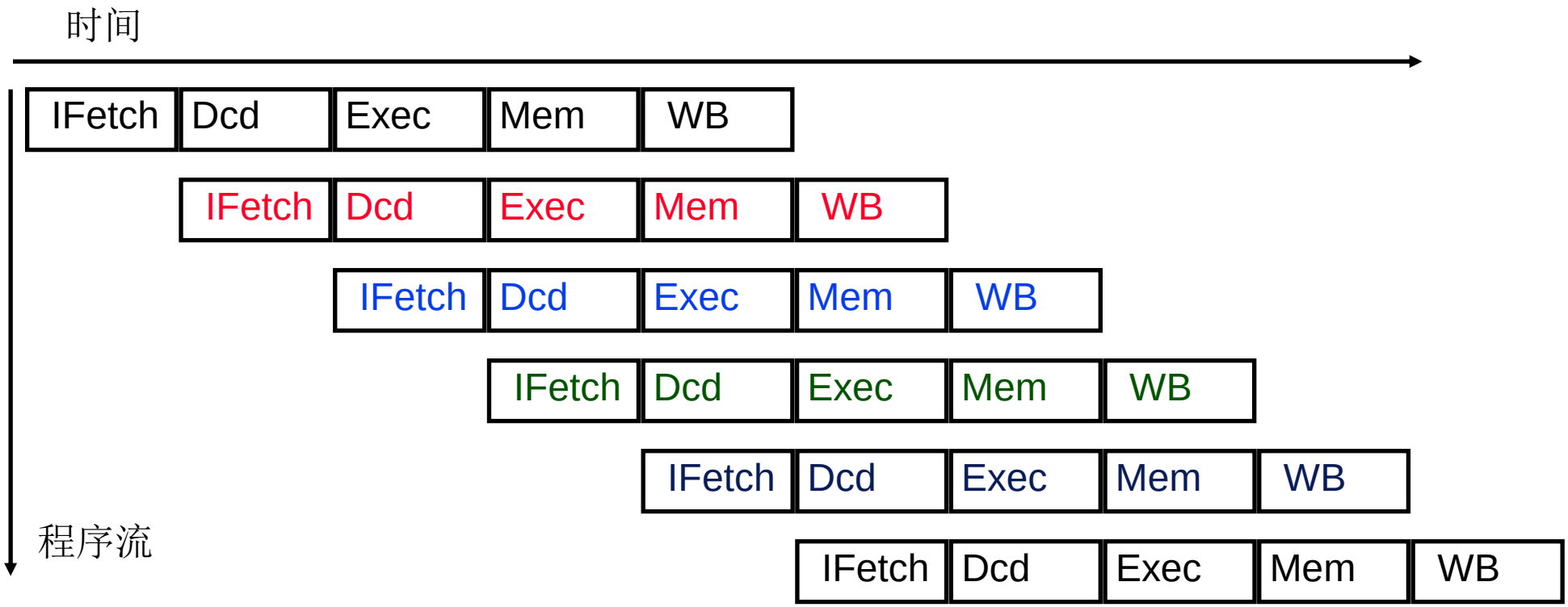
- *将数据通路实际划分成流水段，需要增加什么？*

流水线的图形化表示

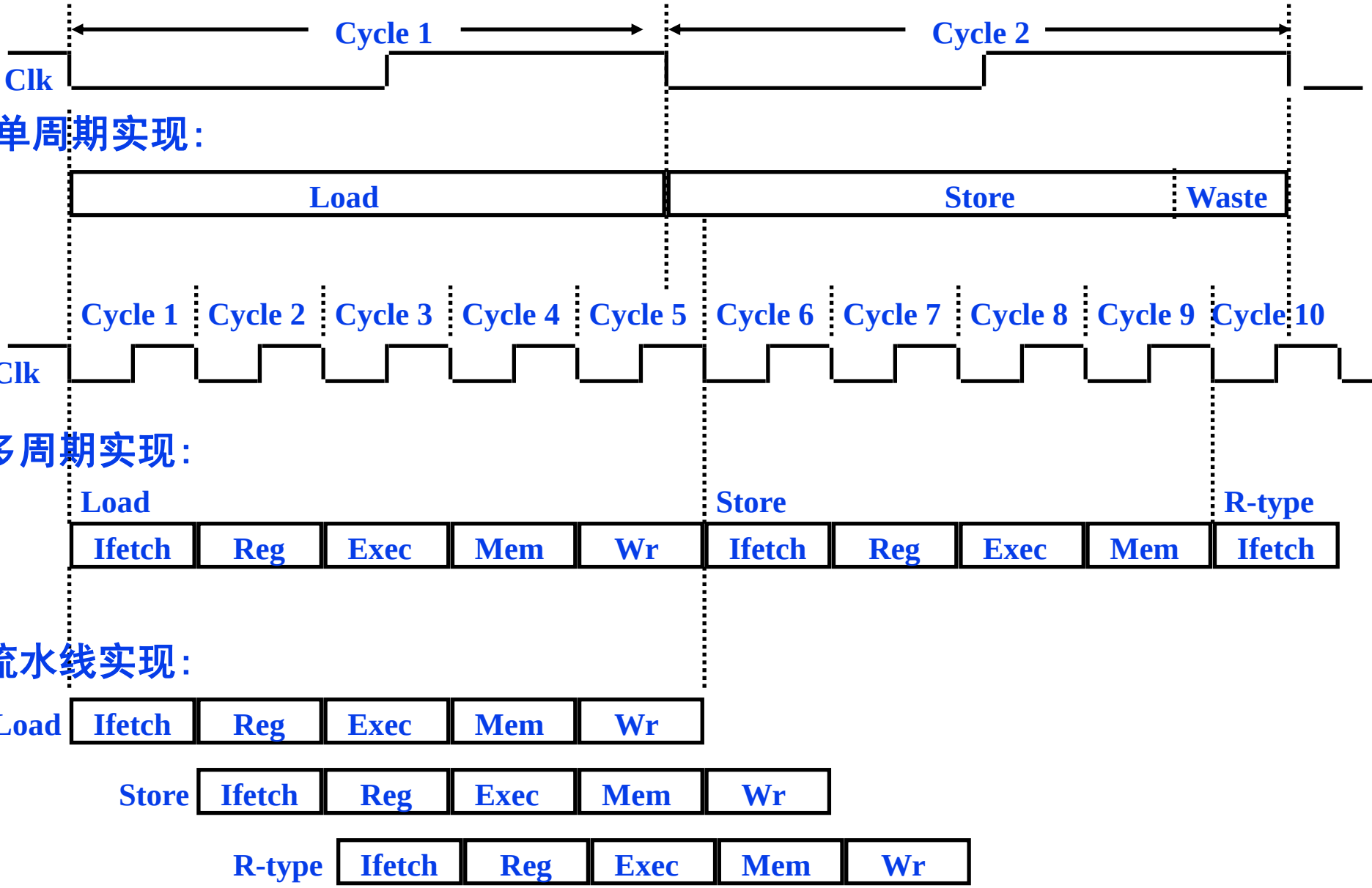


- 执行某段代码需要多少周期?
- 在某一周期，ALU正在做什么?
- 该表示有助于加强对数据通路的理解

常规流水化执行的表示


时间
IFetch
Dcd
Exec
Mem
WB
IFetch
Dcd
Exec
Mem
WB
IFetch
Dcd
Exec
Mem
WB
IFetch
Dcd
Exec
Mem
WB
IFetch
Dcd
Exec
Mem
WB
IFetch
Dcd
Exec
Mem
WB
程序流

单周期、多周期 与 流水线


Cycle 1
Cycle 2
Clk
单周期实现:
Load
Store
Waste
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Cycle 10
Clk
多周期实现:
Load
Store
R-type
Ifetch
Reg
Exec
Mem
Wr
Ifetch
Reg
Exec
Mem
Ifetch
流水线实现:
Load
Ifetch
Reg
Exec
Mem
Wr
Store
Ifetch
Reg
Exec
Mem
Wr
R-type
Ifetch
Reg
Exec
Mem
Wr

为什么需要流水线?

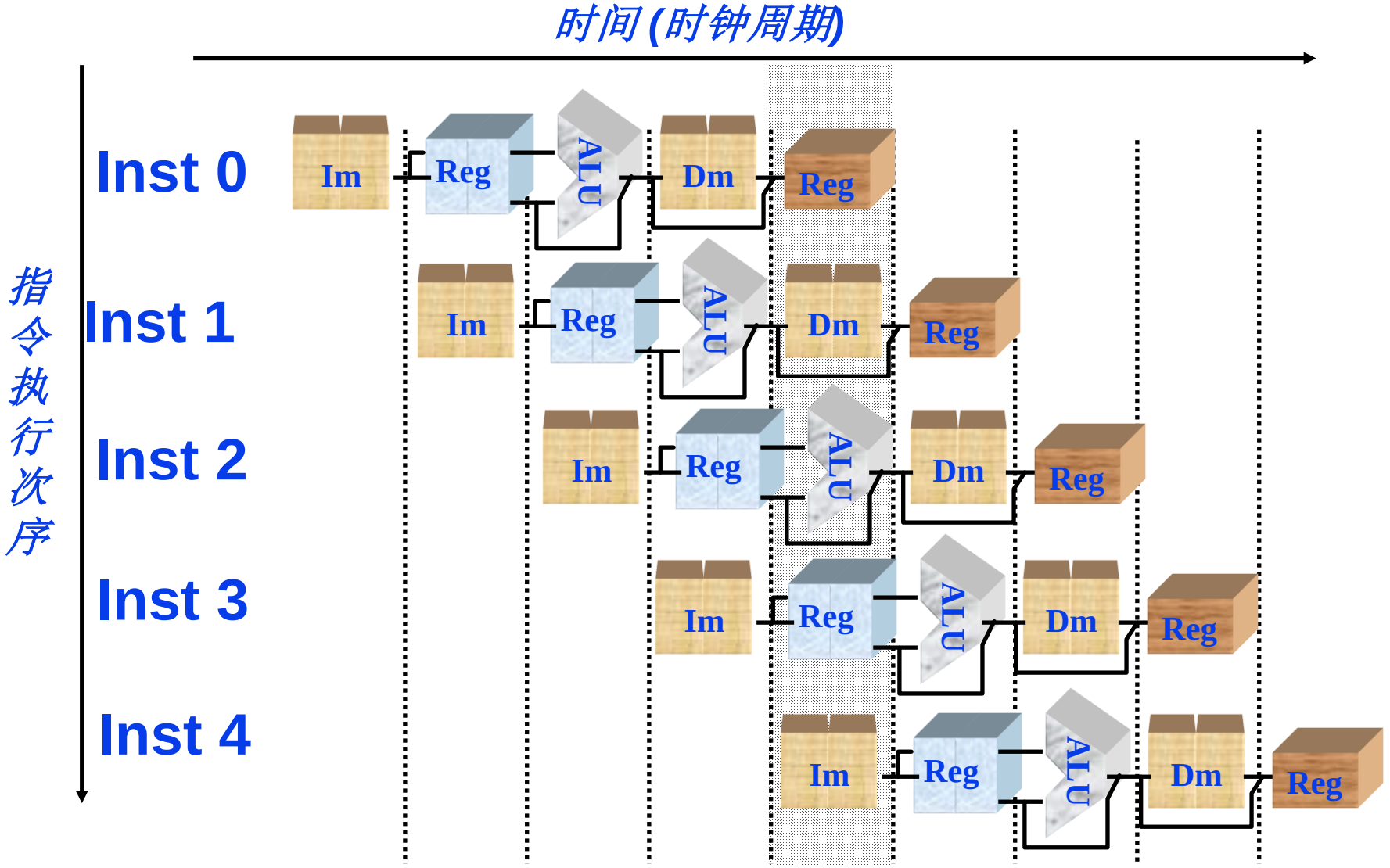
- 假设我们需要执行100条指令

- 单周期机器
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns

- 多周期机器
  - 10 ns/cycle x 4.6 CPI (due to inst mix) x 100 inst = 4600 ns

- 理想的流水化机器
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

为什么需要流水线？
因为需要提高资源利用率

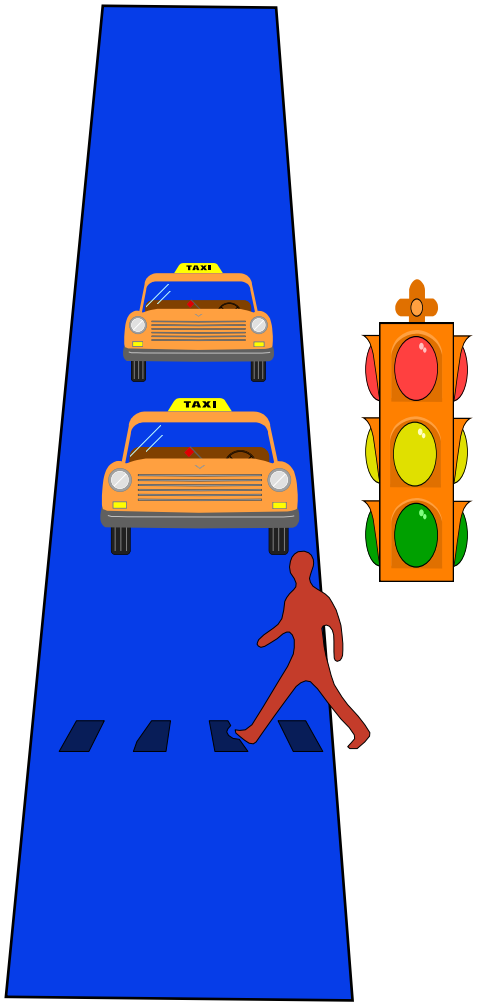


流水技术会产生哪些问题?

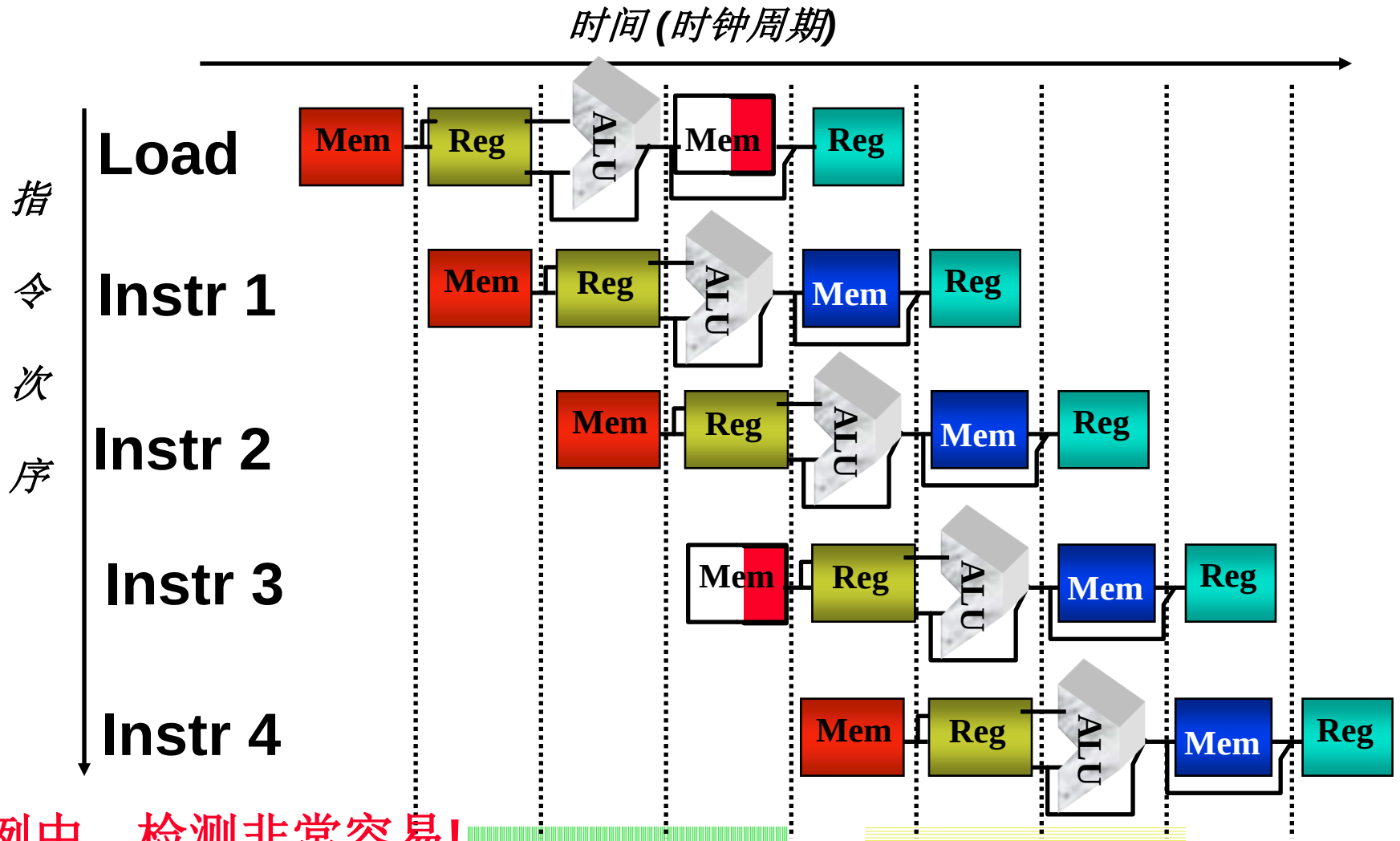
- 流水线冒险(Pipeline Hazards)
  - 结构冒险(structural hazards): 试图同时以两种不同的方式使用同一资源
    - 例如, 组合式洗衣机/烘干机可能产生结构冒险 或者熨整工工作时偷看电视
  - 数据冒险(data hazards): 在产生数据之前,就试图使用它们
    - 例如，一包衣物在洗衣机中, 另一包衣物在干衣机中, 那么就不能对这些衣物进行熨整。
    - 流水线中的一条指令等待使用上一条指令的结果。
  - 控制冒险(control hazards):在判定转移条件之前，就试图决策转移方向
    - 例如，在洗一件非常脏的衣物之前，需要判定加多少洗衣粉;
    - 转移指令

流水技术会产生哪些问题（续）？

- 等待 总是可以解决冒险问题
  - 流水线控制必须检测冒险
  - 采取必要措施（或延迟等待）来解决冒险



单存储器是一种结构冒险

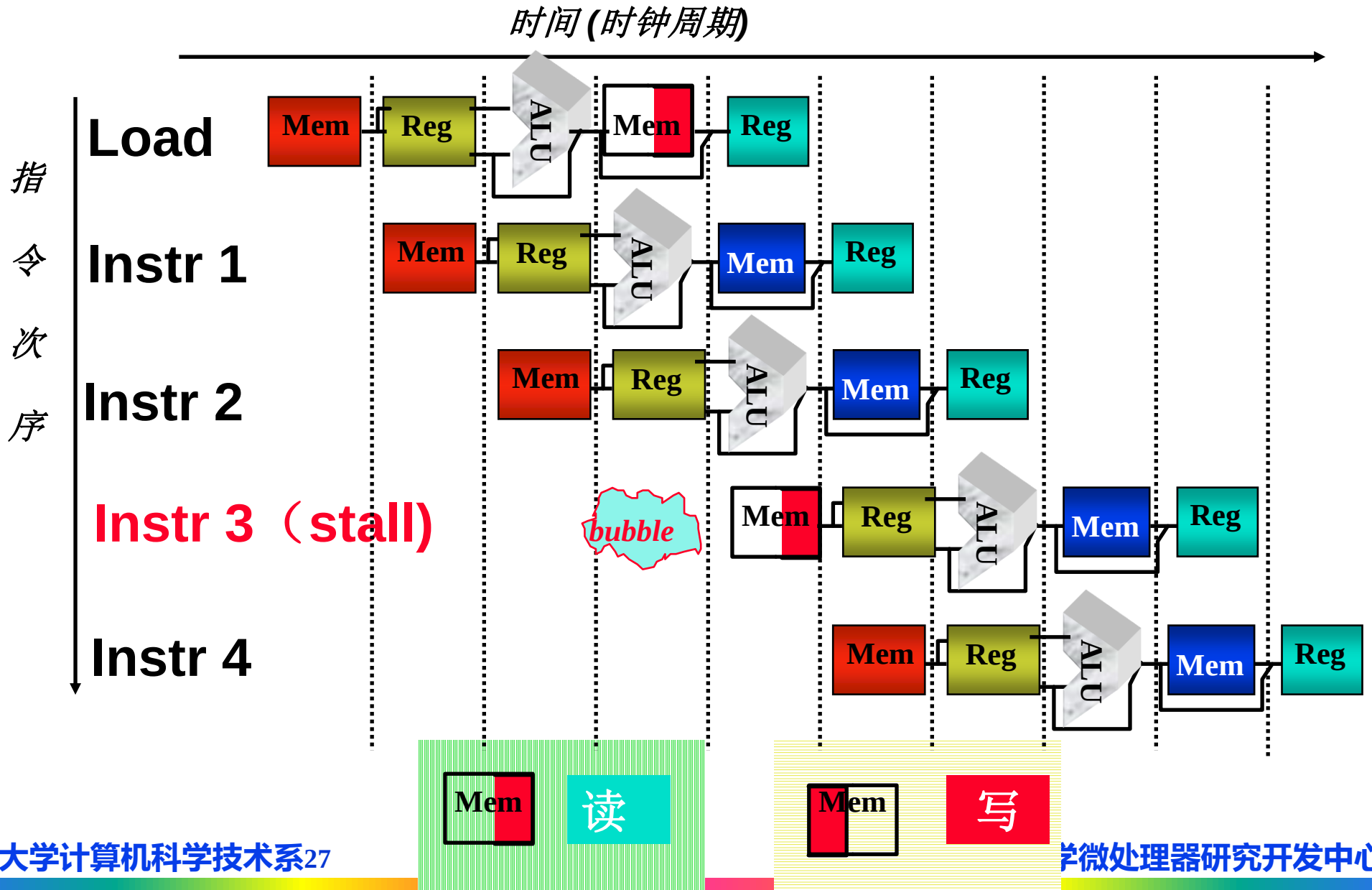


此例中，检测非常容易!

结构冒险限制性能

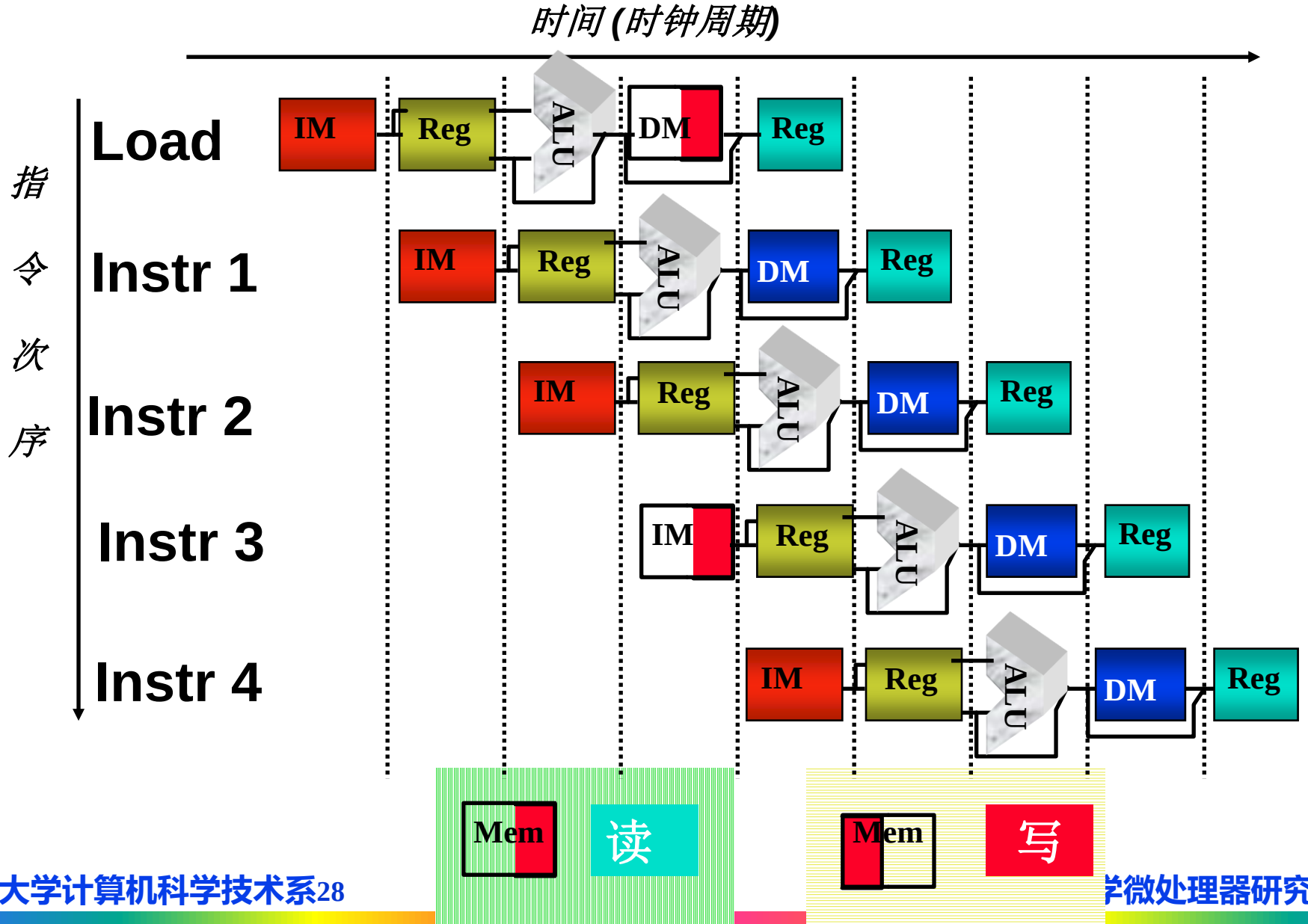
- 示例: 如果每条指令平均访问 **1.3** 次存储器，而每个周期只能完成一次存储器访问，那么
  - 平均 **CPI ≥ 1.3**
  - 否则，资源的利用率 **> 100%**

解决方案一：暂停解决单存储器结构冒险



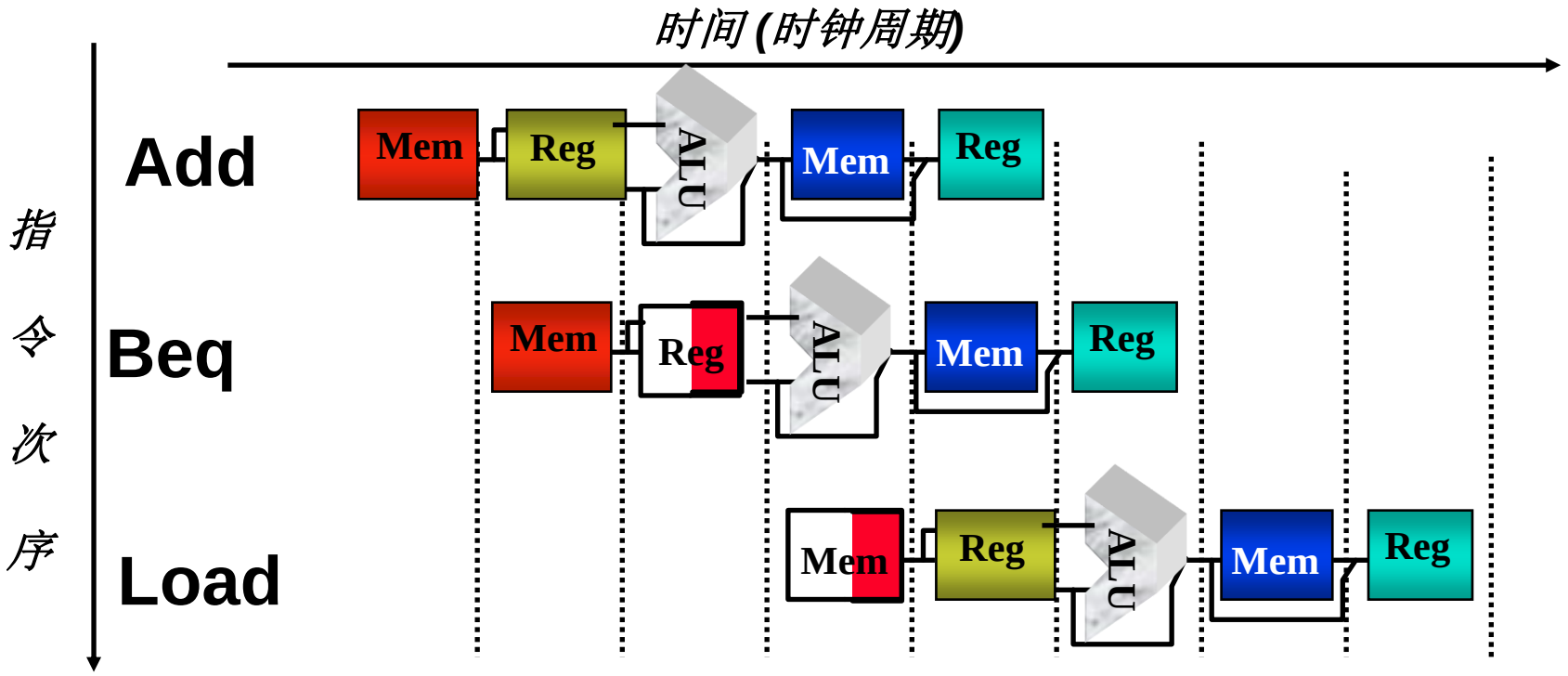
解决方案二：资源复置解决存储器结构冒险

- 分离 指令Cache (Im) 和 数据Cache (Dm)



控制冒险的解决策略

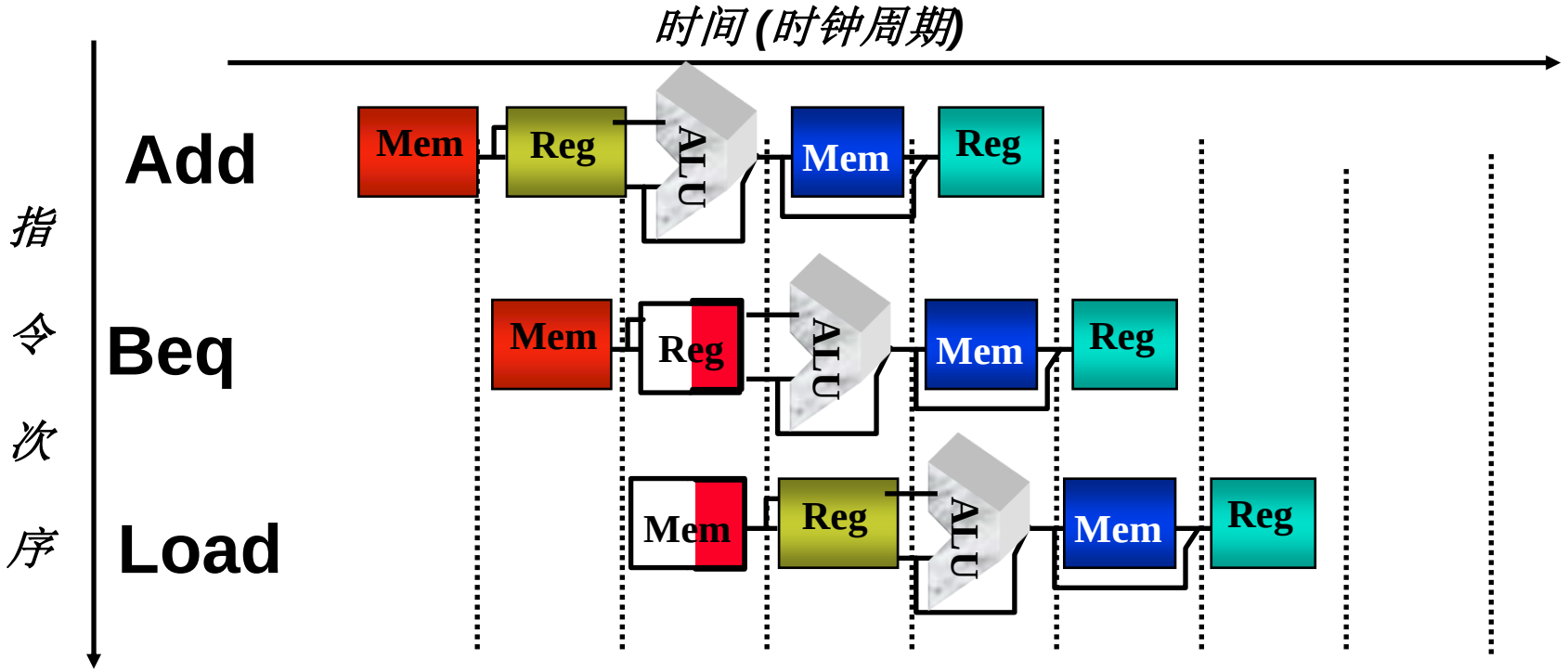
- 暂停(Stall): 等待直到决策明朗
  - 通过增设硬件在读取寄存器的周期就对转移方向进行判定，从而可以在第二流水段完成转移方向的判定



- 影响: 每条转移指令需要两个时钟周期完成 => 速度慢

控制冒险的解决策略

- 预测：猜测某一转移方向，如果猜测错误，就用备份恢复
  - 猜测转移不发生

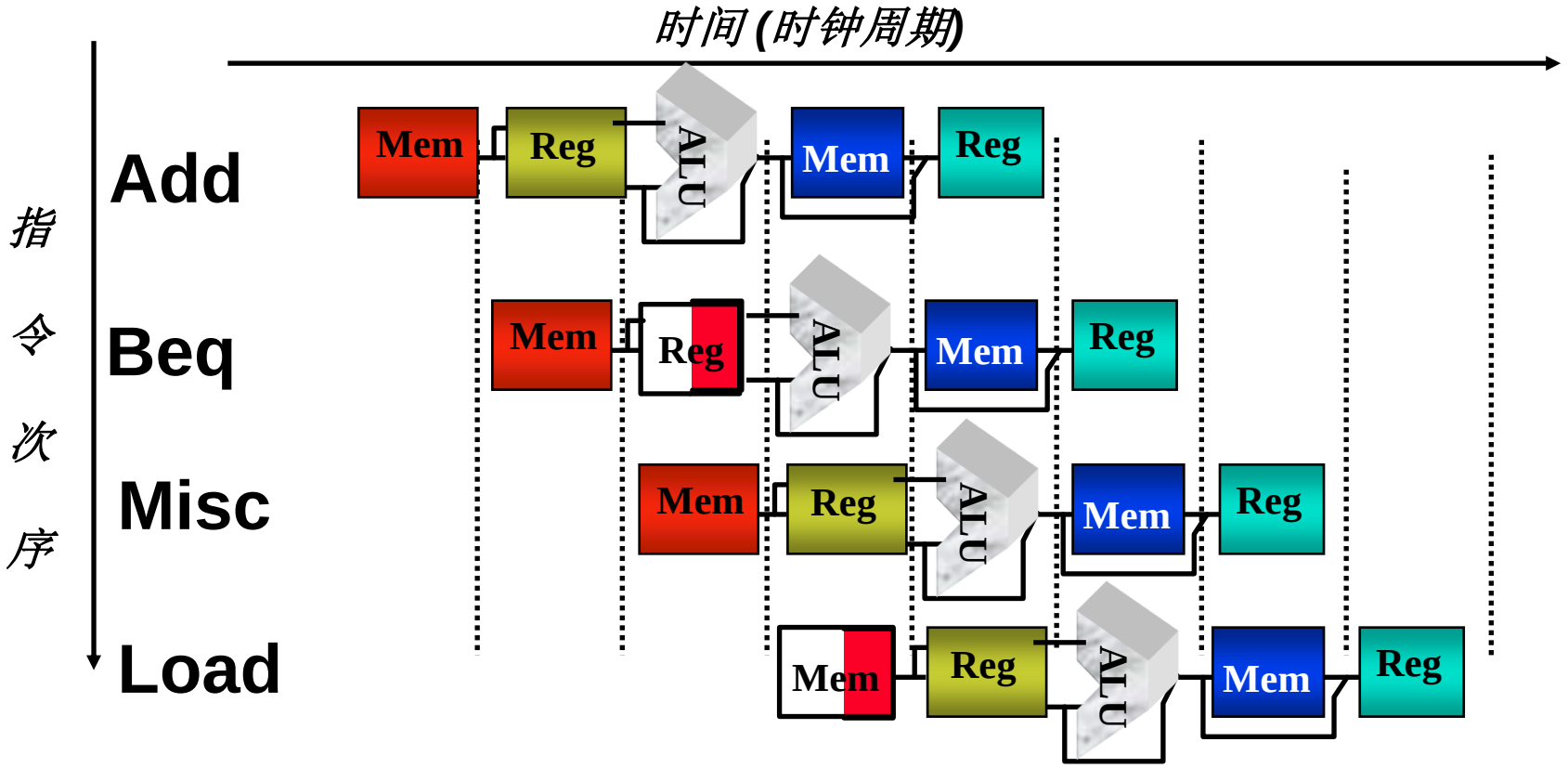


- 效果：如果猜测正确，每条指令一个时钟周期完成，如果猜错每条转移指令两个周期（正确率 ≈ 50%）

- 较复杂的硬件预测：一条指令的历史（(正确率 ≈ 90%）

控制冒险的解决策略

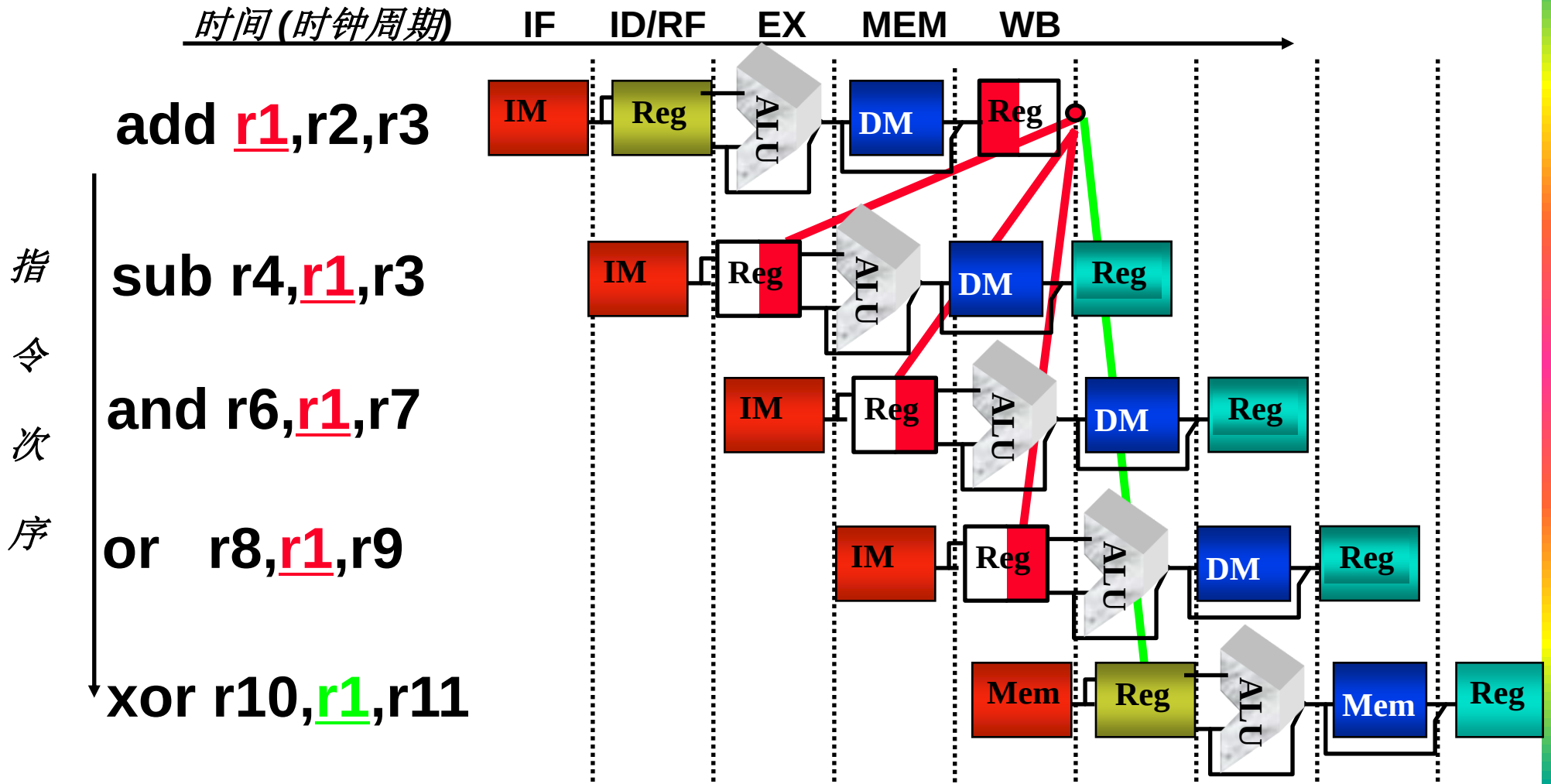
- 重新定义转移指令的行为
  - 延迟转移：转移行为在下一条指令之后发生



- 影响：如果能够将有效指令调度到延迟槽中，每条转移指令的完成将不占用时钟周期(≈ 50%的可能)

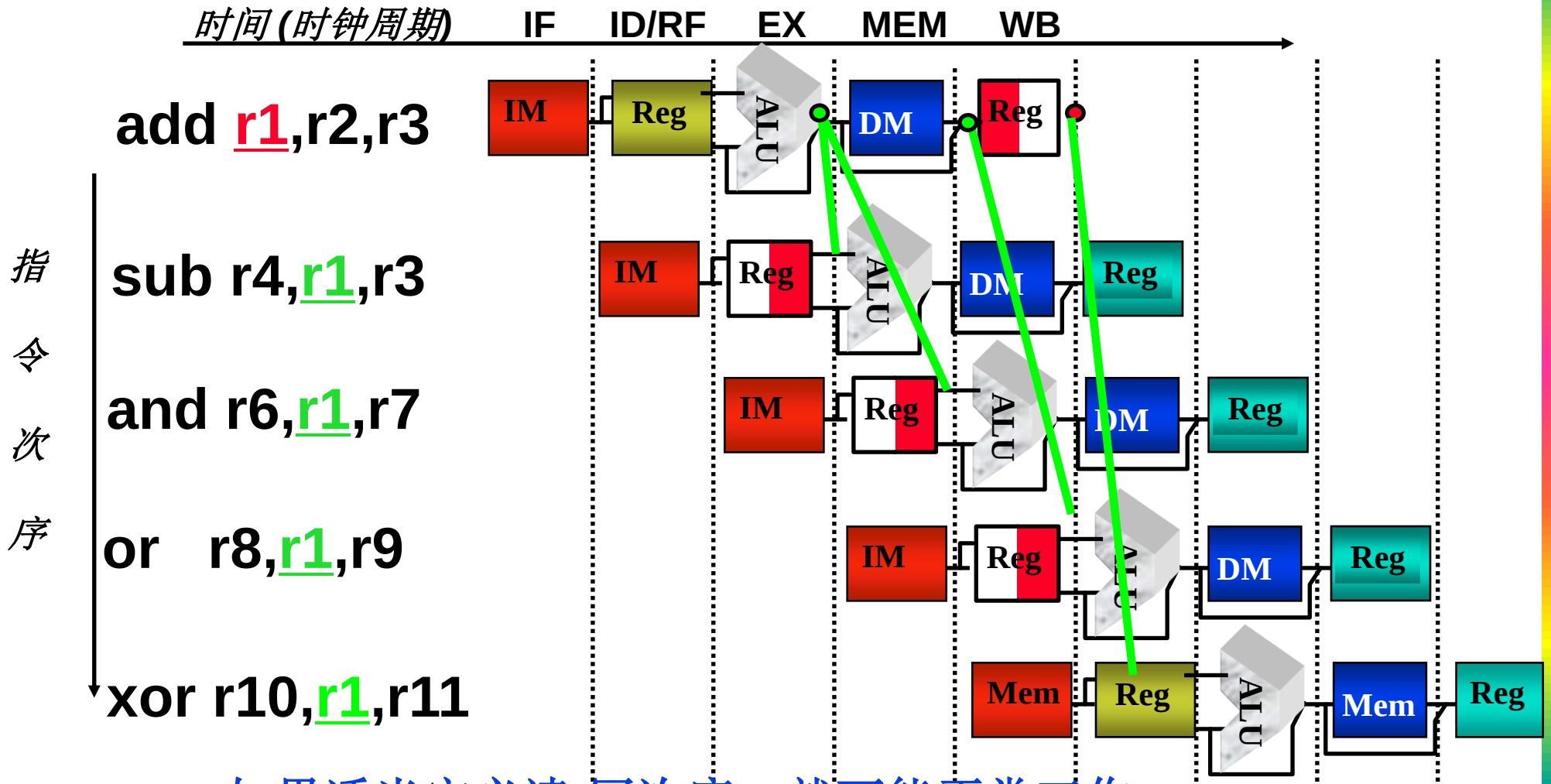
关于rl的数据冒险

- 立即向后相关就可能出现冒险



数据冒险解决策略

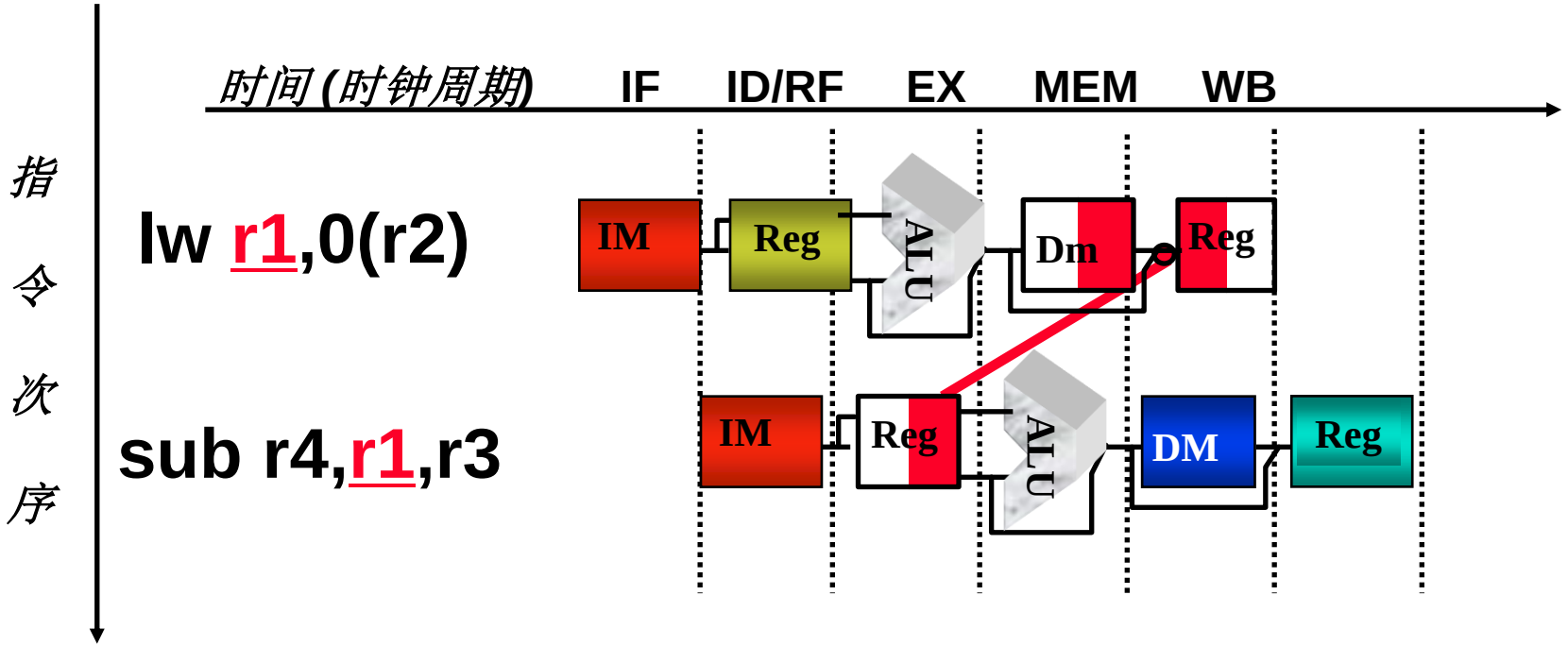
- 从某流水段向其他流水段前递(forward)结果



- 如果适当定义读/写次序，就可能正常工作

前递或旁路(forward or bypass):装入指令

- 立即向后相关就可能出现冒险

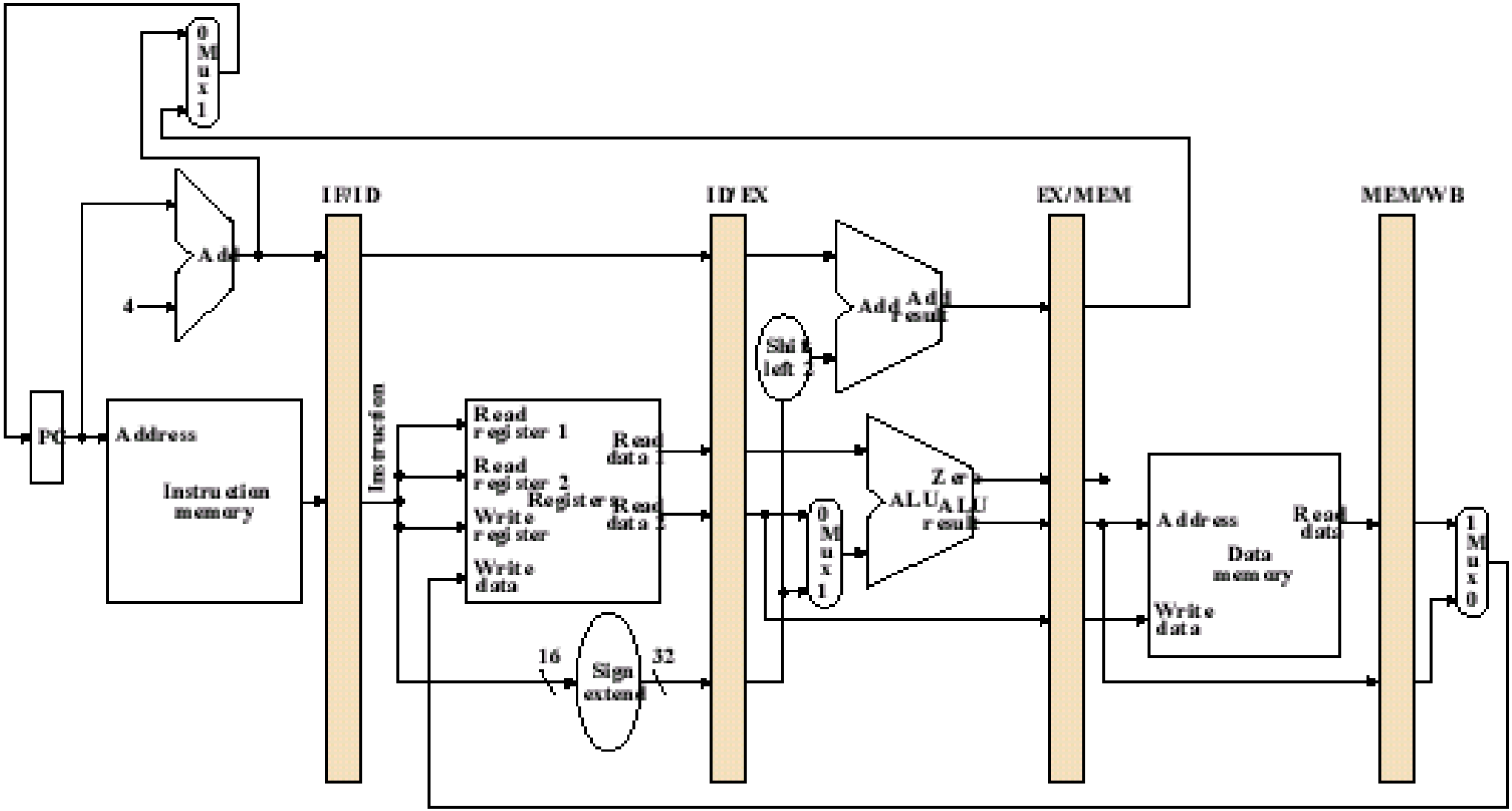


- 不能通过前递技术解决
- 必须延迟/暂停与装入指令相关的指令

设计一个流水化的处理器

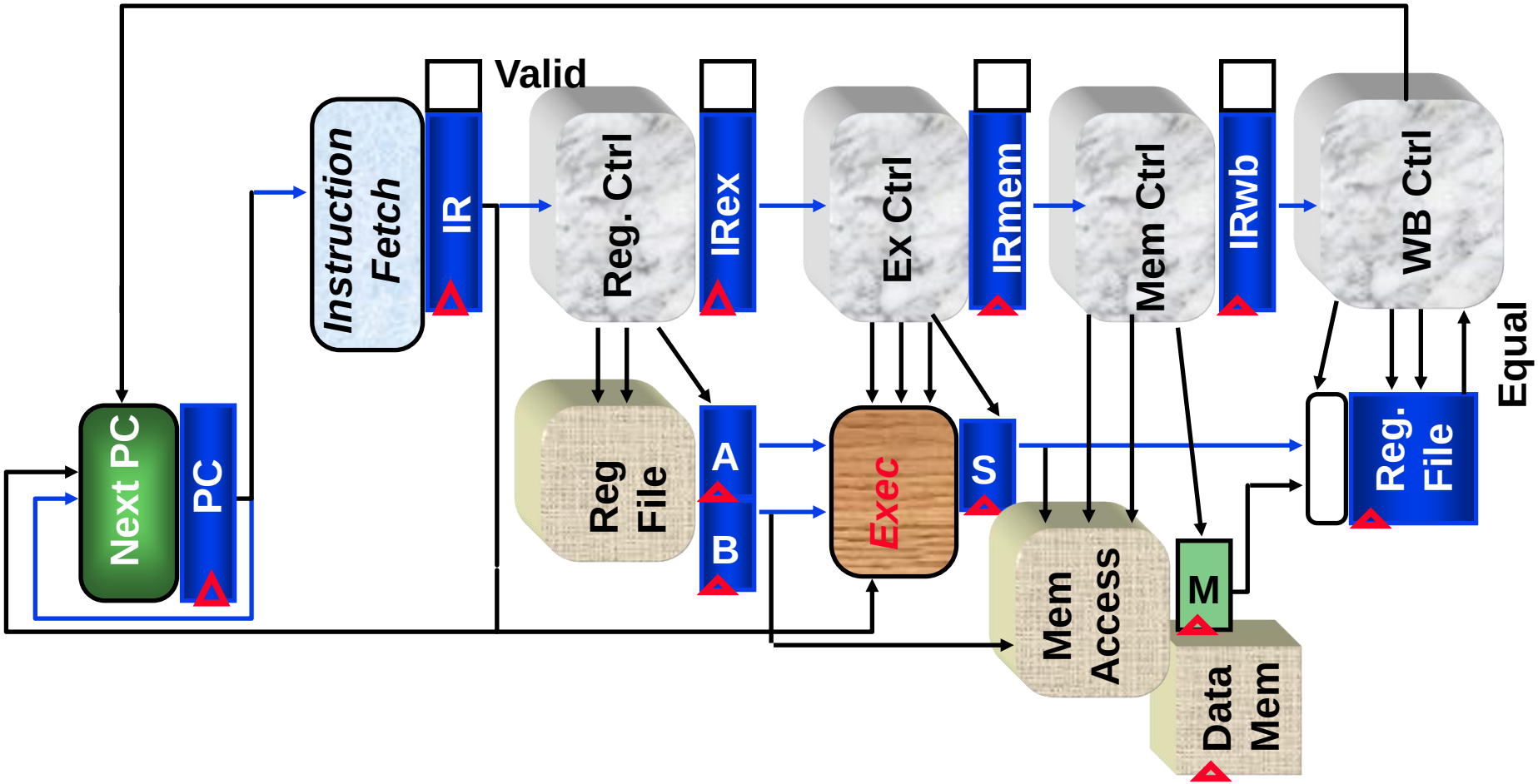
- 回顾我们设计的数据通路和控制图
- 将资源和状态联系起来
- 保证处理流程没有冲突，或考虑出解决方案
- 在适当的流水段发出正确的控制信号

教科书上的流水化的数据通路

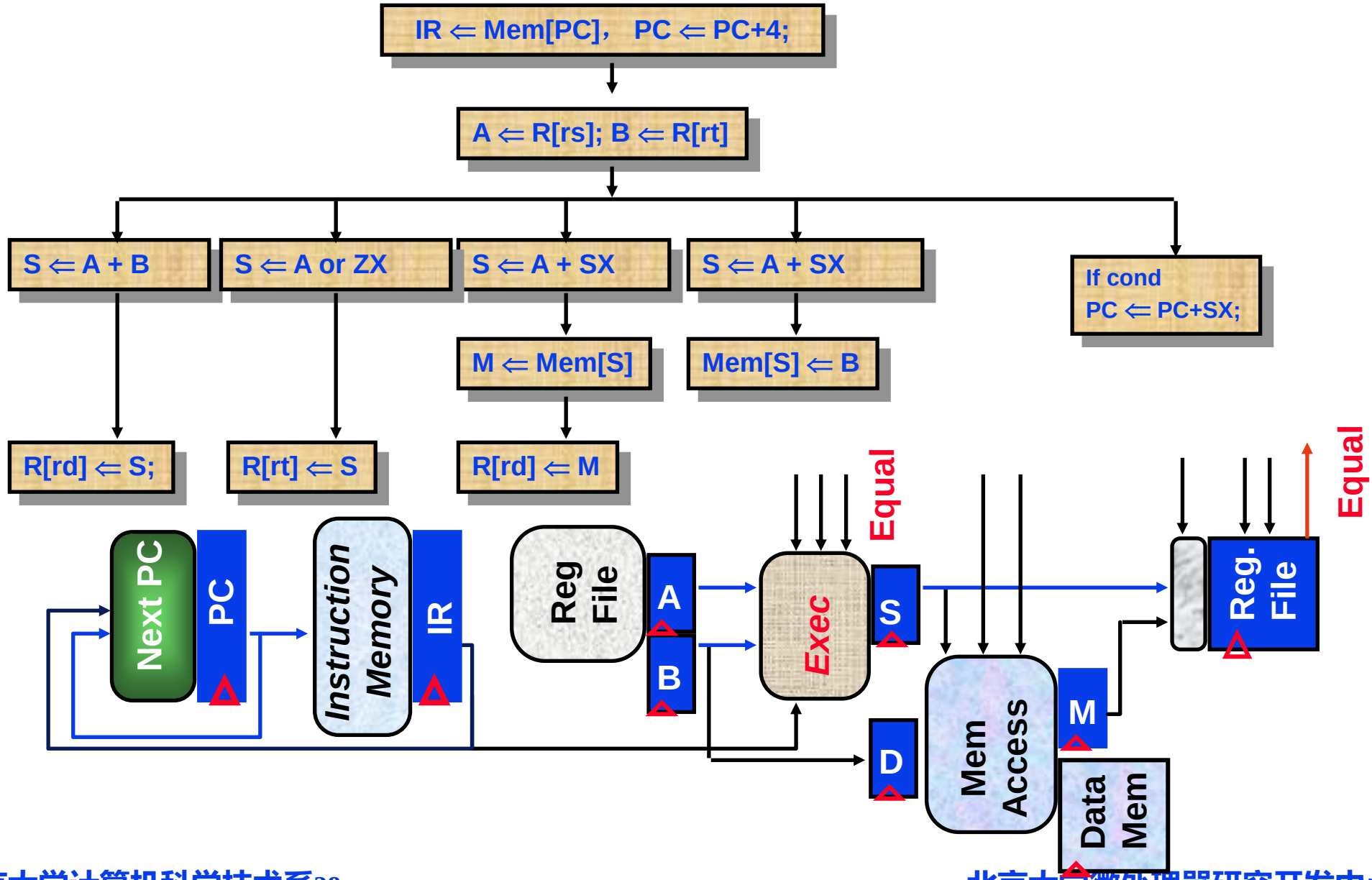


多周期数据通路中介绍的基本流水化的处理器

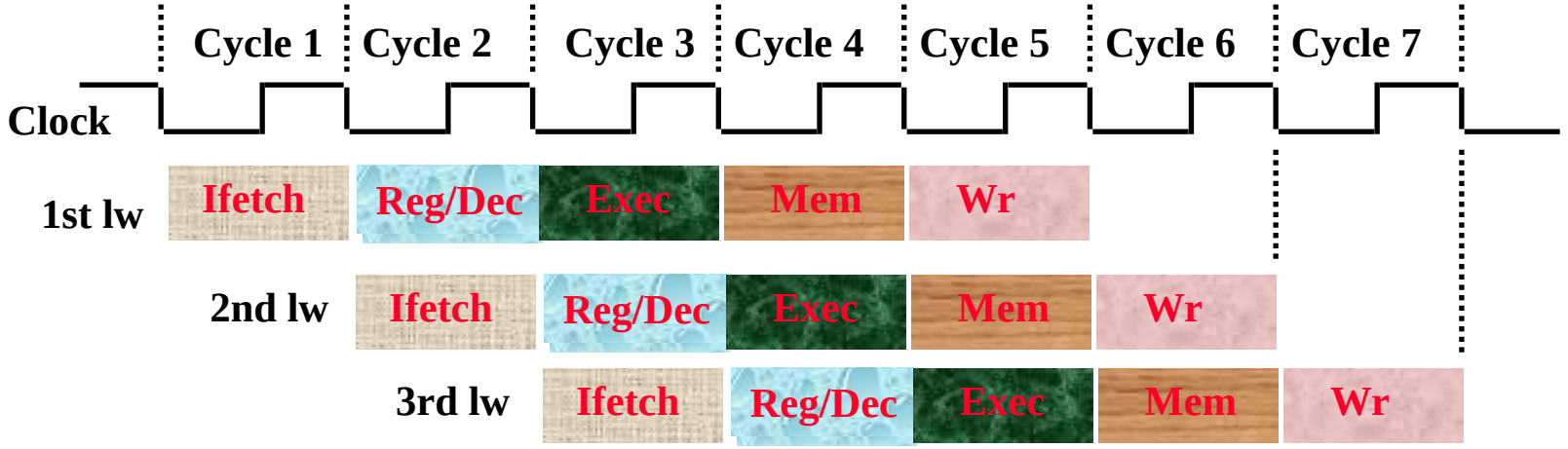
- 如果每周期，开始一条新指令的处理，会发生什么情况？



控制和数据通路

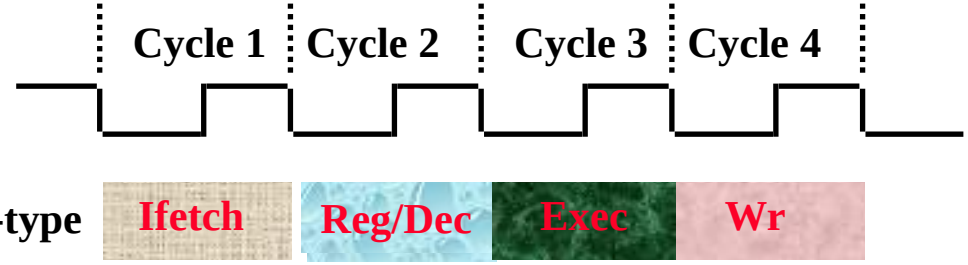

IR ⇐ Mem[PC]， PC ⇐ PC+4;
A ⇐ R[rs]; B ⇐ R[rt]
S ⇐ A + B
S ⇐ A or ZX
S ⇐ A + SX
S ⇐ A + SX
If cond
PC ⇐ PC+SX;
M ⇐ Mem[S]
Mem[S] ⇐ B
R[rd] ⇐ S;
R[rt] ⇐ S
R[rd] ⇐ M
Next PC
PC
Instruction Memory
IR
Reg File
A
B
Exec
S
Equal
D
Mem Access
M
Data Mem
Reg. File
Equal

流水化装入指令



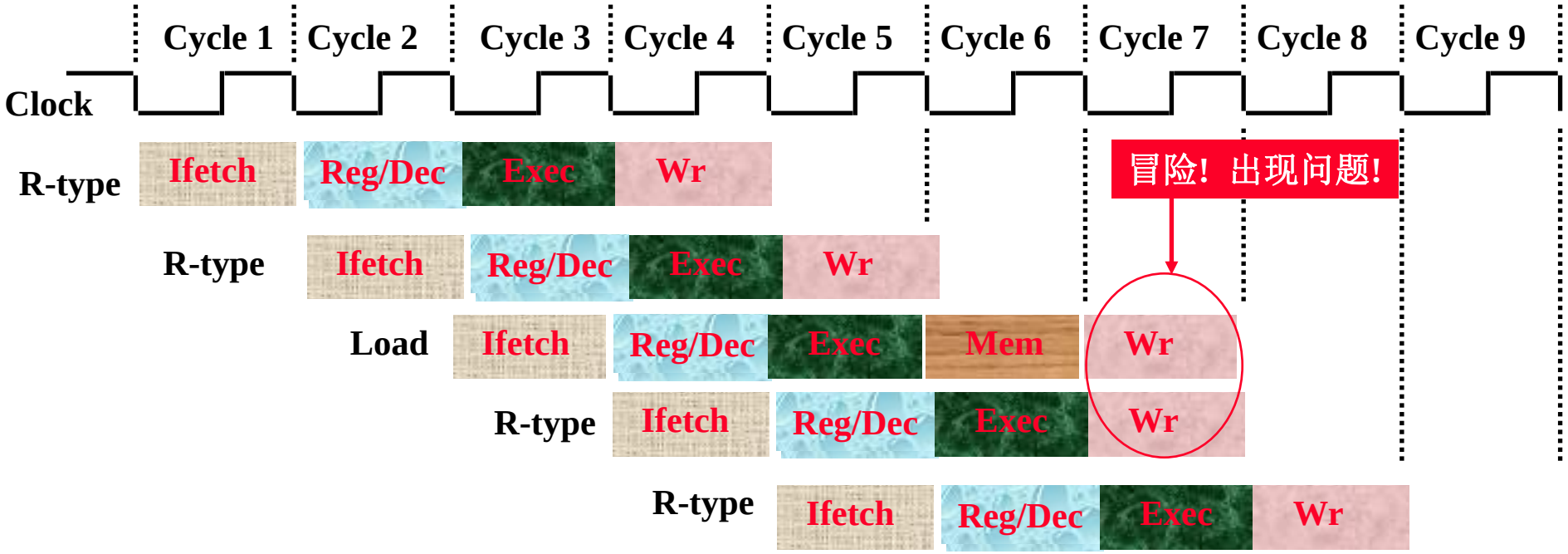
- 流水线数据通路中的五个独立功能部件是:
  - 取指段(Ifetch)的指令存储器
  - 寄存器/译码段(Reg/Dec)的寄存器堆的读端口 (busA 和 busB)
  - 执行段(Exec)的ALU
  - 存储器段(Mem)段的数据存储器
  - 回写段(Wr)的寄存器堆的写端口 (busW)

R型指令的四段



- **Ifetch:** 取指
  - 从指令存储器中读取指令
- **Reg/Dec:** 取寄存器和指令译码
- **Exec:**
  - 对两个寄存器操作数执行ALU操作
- **Wr:** 将ALU的输出回写到寄存器堆

流水化R型指令和装入指令



- 产生了流水线冲突或者结构冒险:
  - 两条指令都试图同时回写寄存器堆!
  - 寄存器堆只有一个写入端口

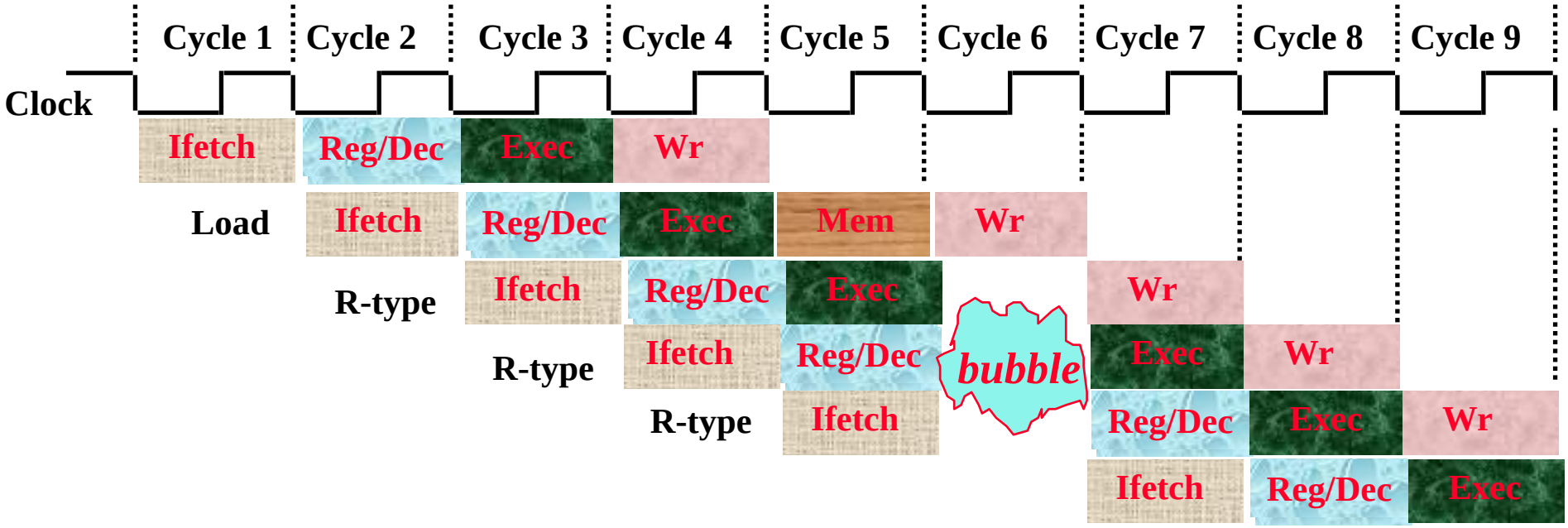
重要发现

- 每个功能部件只能被每条指令使用一次
- 每个功能部件必须被所有指令在相同的流水段使用:
  - 装入指令在它的第五级使用寄存器堆的写端口

  - R型指令在它的第四级使用寄存器堆的写端口

- 有两种方法解决流水线冒险

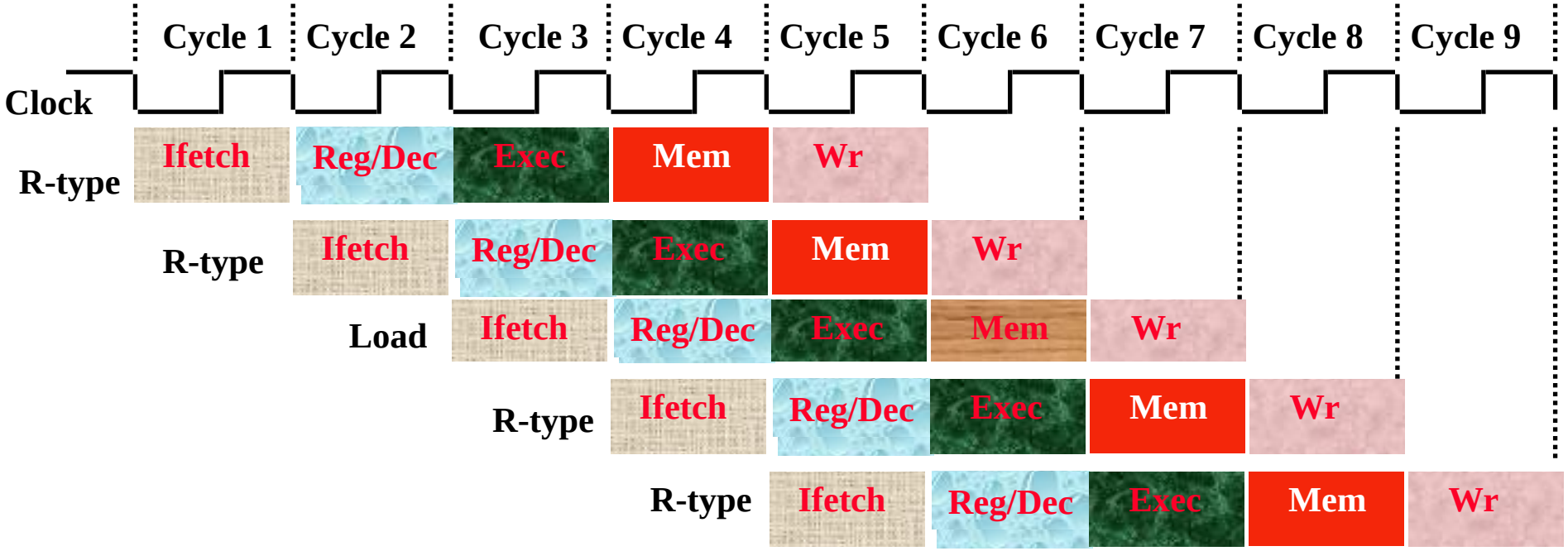
第一种解决方案：在流水线中插入空泡



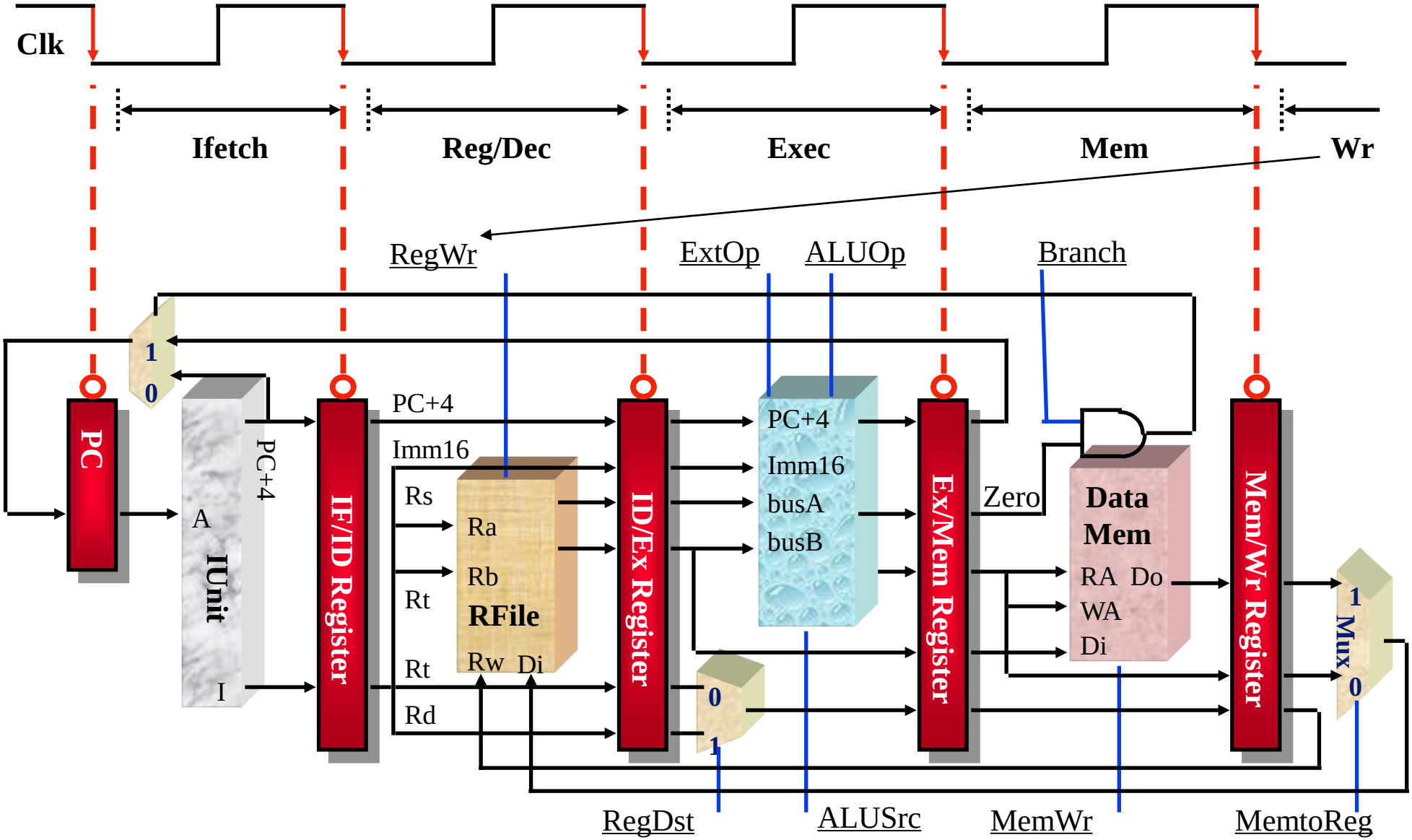
- 在流水线中插入空泡从而避免了在同一周期出现两次写入操作
  - 控制逻辑变得更加复杂
  - 失去了取指和发送的机会
- 在第六周期没有任何指令启动!

第二种解决方案： 将R型指令的回写延迟一个周期

- 将R型指令的写寄存器操作延迟一个周期:
  - 现在,R型指令也在第五段才使用寄存器堆的写端口
  - 存储器段是一个空段: 无所事事

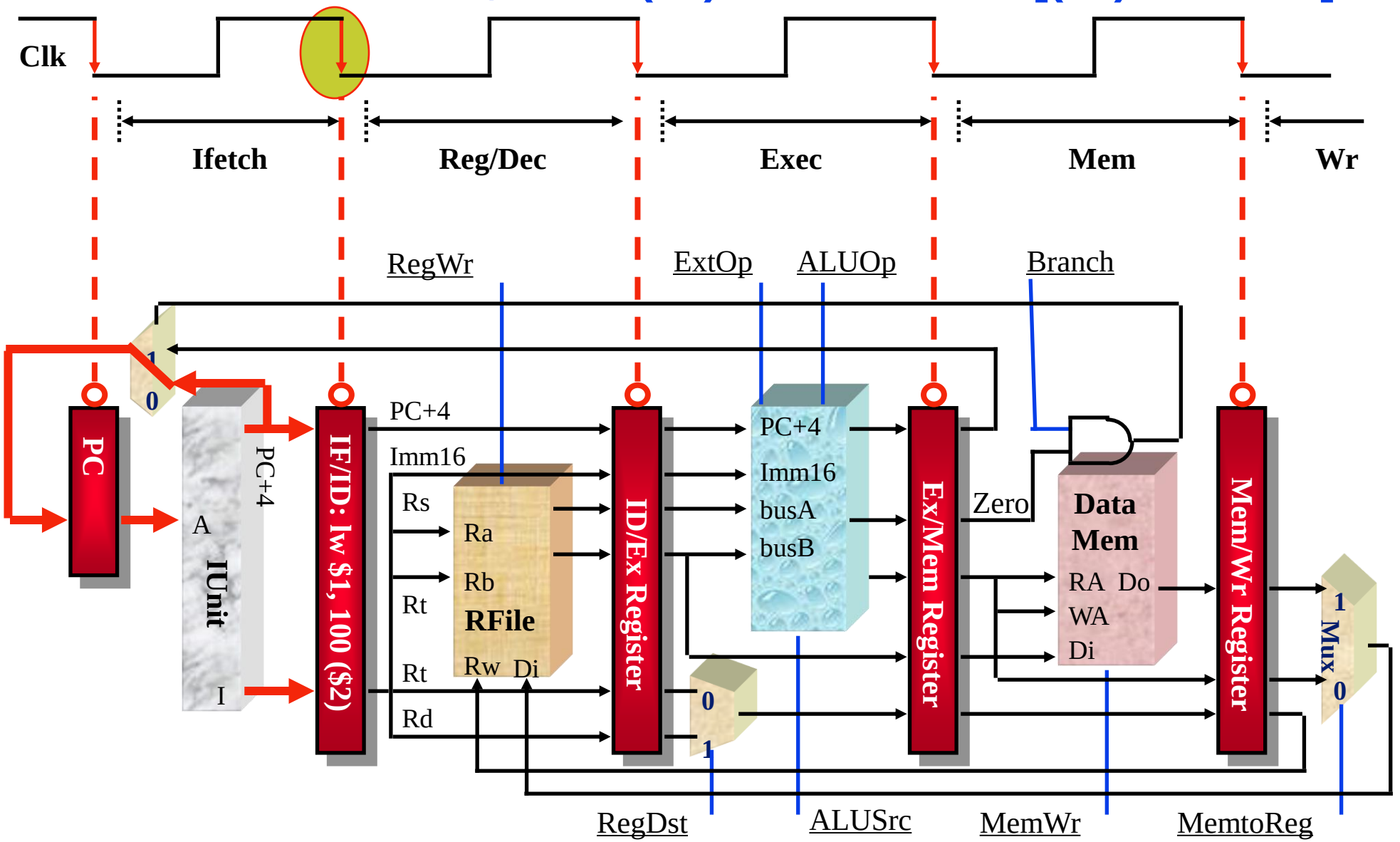


流水化的数据通路


Clk
Ifetch
Reg/Dec
Exec
Mem
Wr
RegWr
ExtOp
ALUOp
Branch
1
0
PC
A
IUnit
I
PC+4
IF/ID Register
PC+4
Imm16
Rs
Rt
Rt
Rd
Ra
Rb
RFile
Rw
Di
ID/Ex Register
PC+4
Imm16
busA
busB
0
1
Ex/Mem Register
Zero
Data
Mem
RA
Do
WA
Di
Mem/Wr Register
1
Mux
0
RegDst
ALUSrc
MemWr
MemtoReg

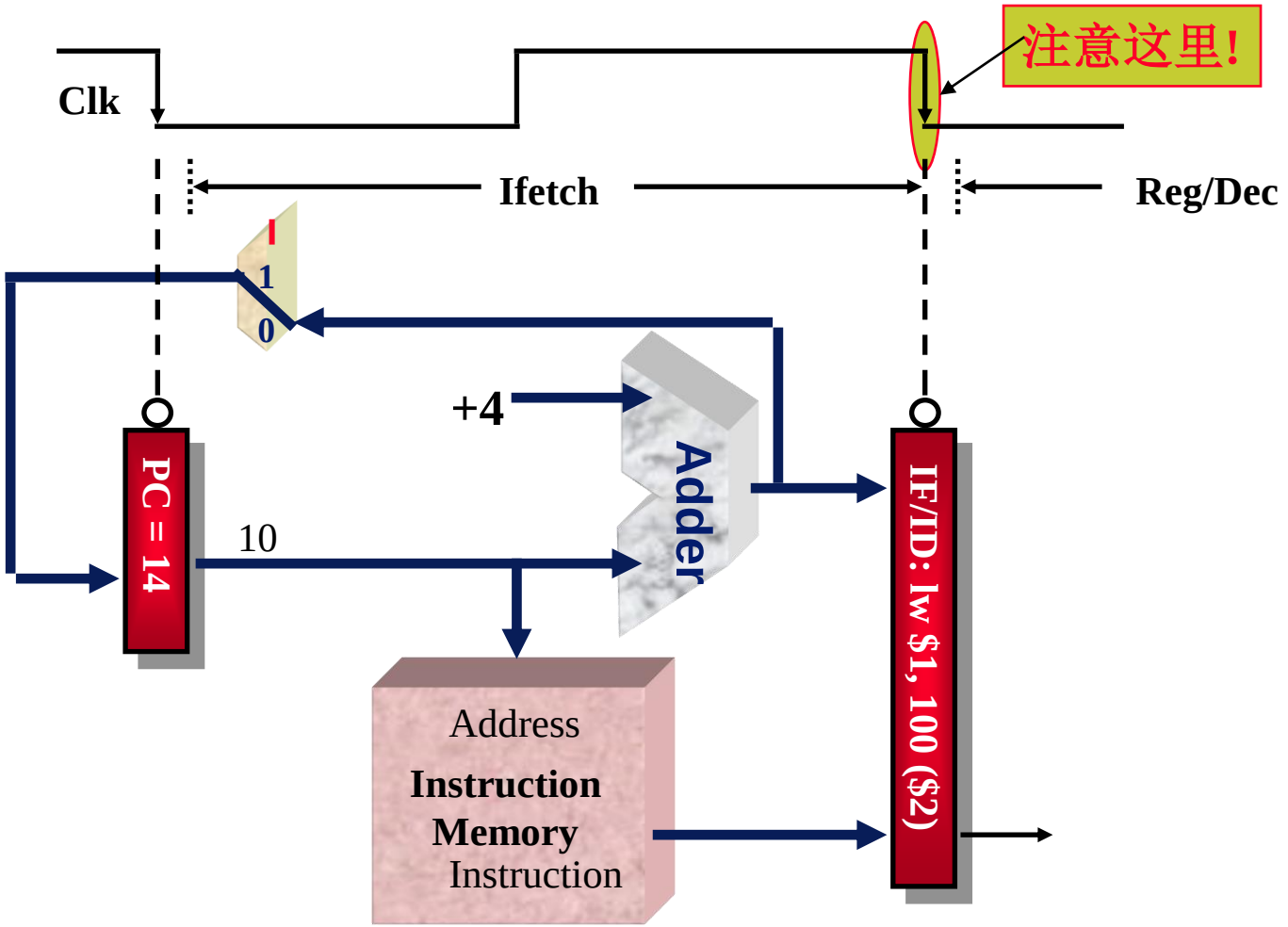
取指段

- Location 10: lw $1, 0x100($2) $1 ⇐ Mem[($2) + 0x100]



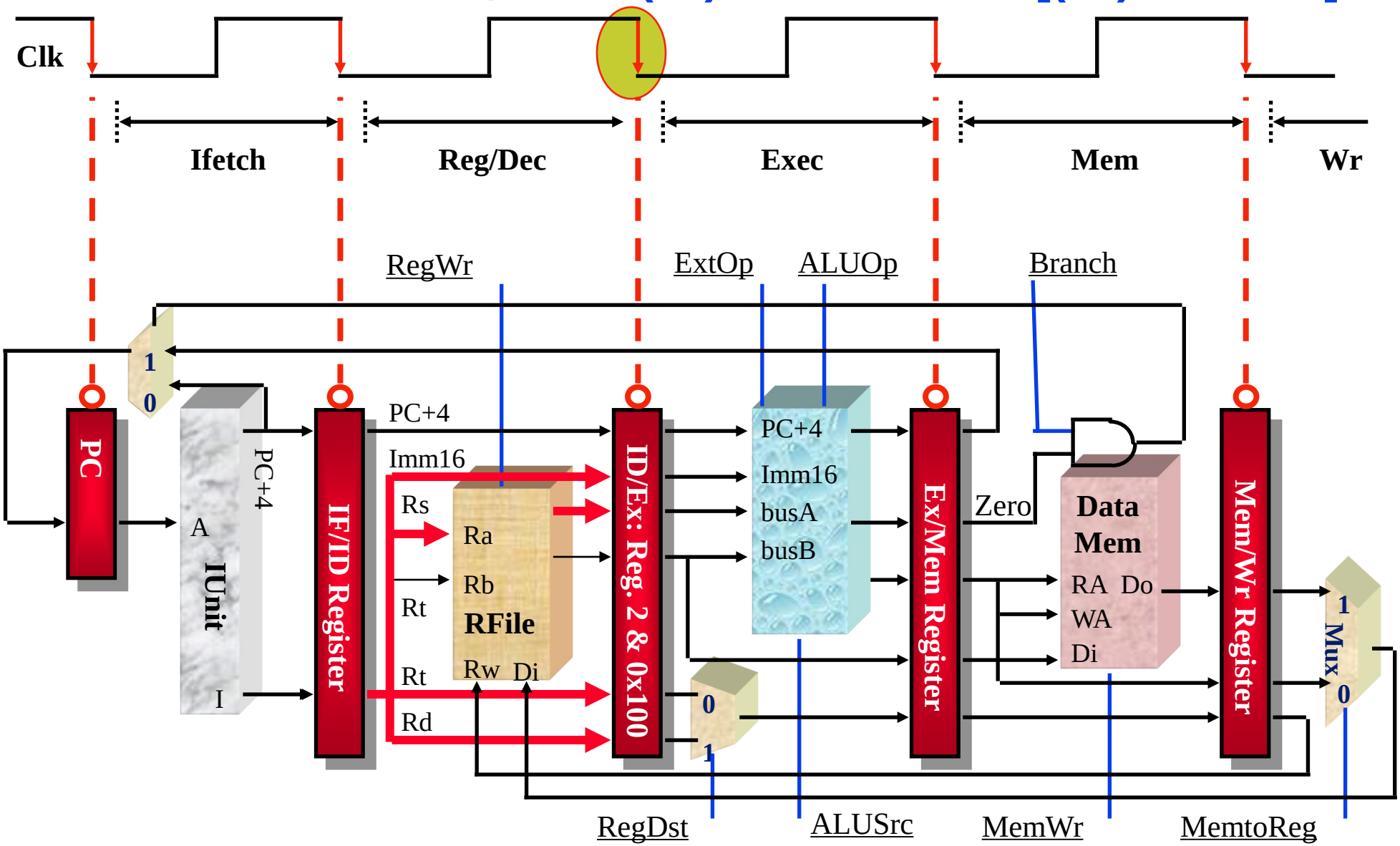
指令部件探秘

- Location 10: lw $1, 0x100($2)



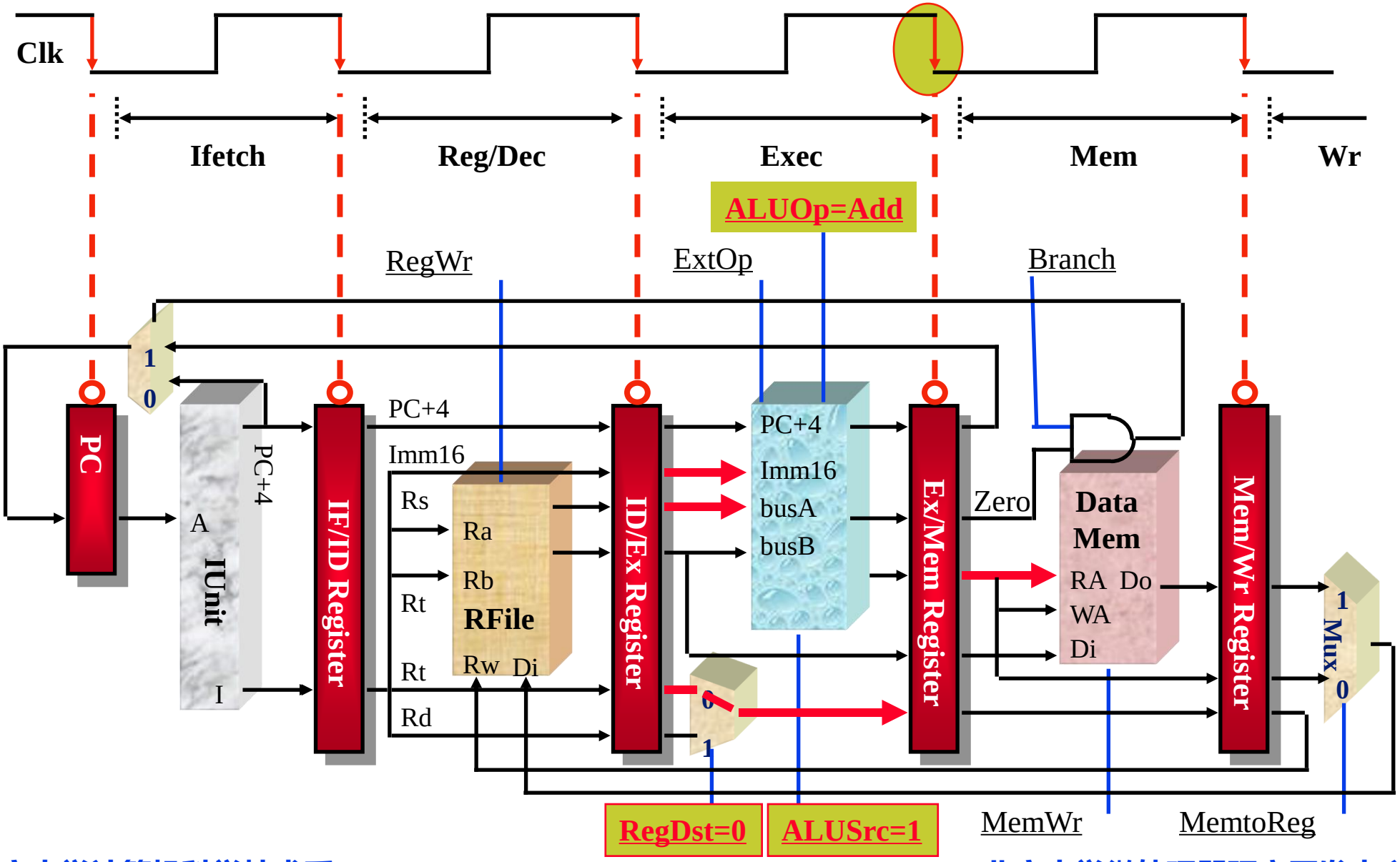
译码/寄存器访问段

- Location 10: lw $1, 0x100($2) $1 ⇐ Mem[($2) + 0x100]

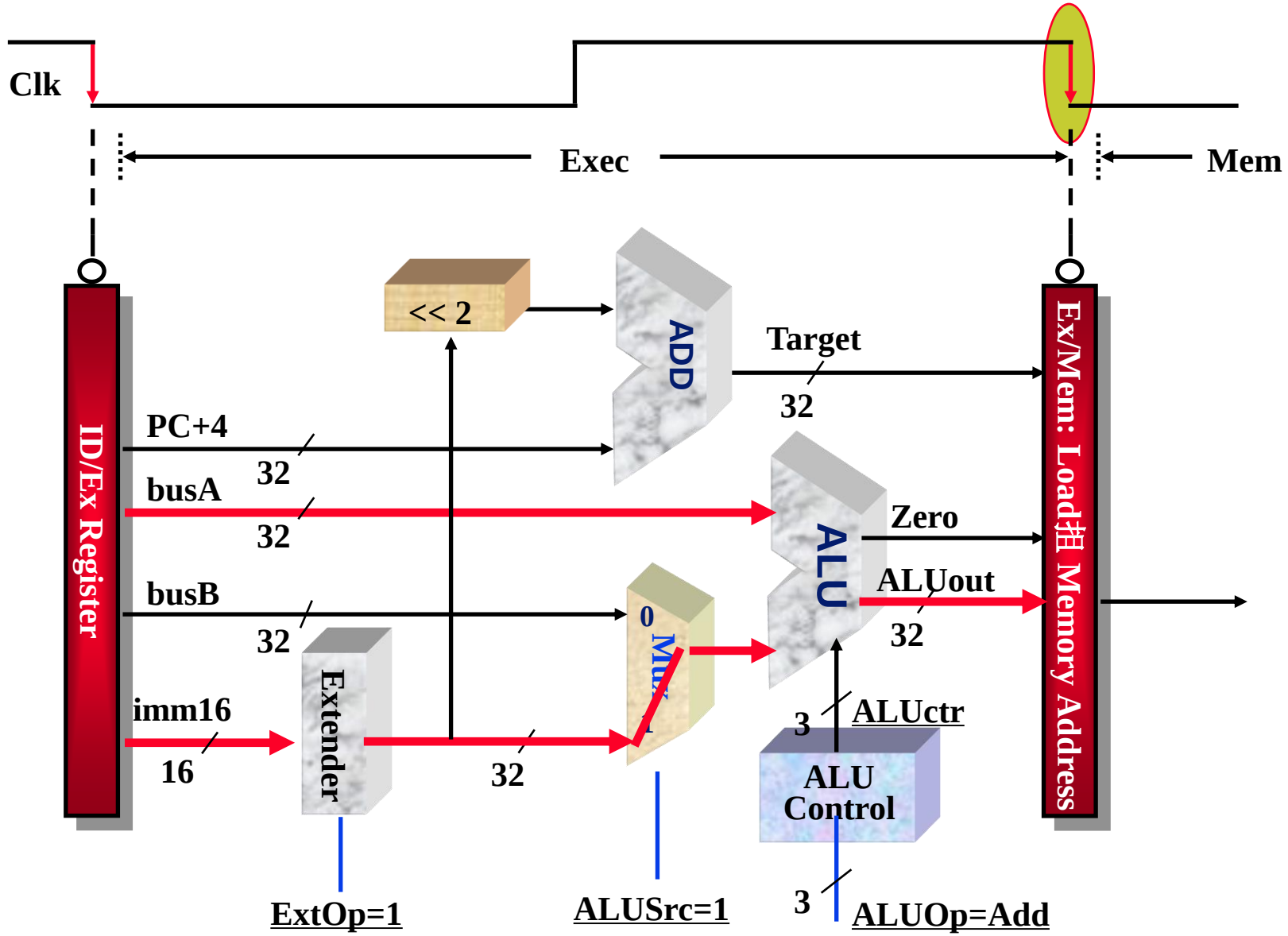


装入指令的地址计算段

- Location 10: lw $1, 0x100($2) $1 ⇐ Mem[($2) + 0x100]

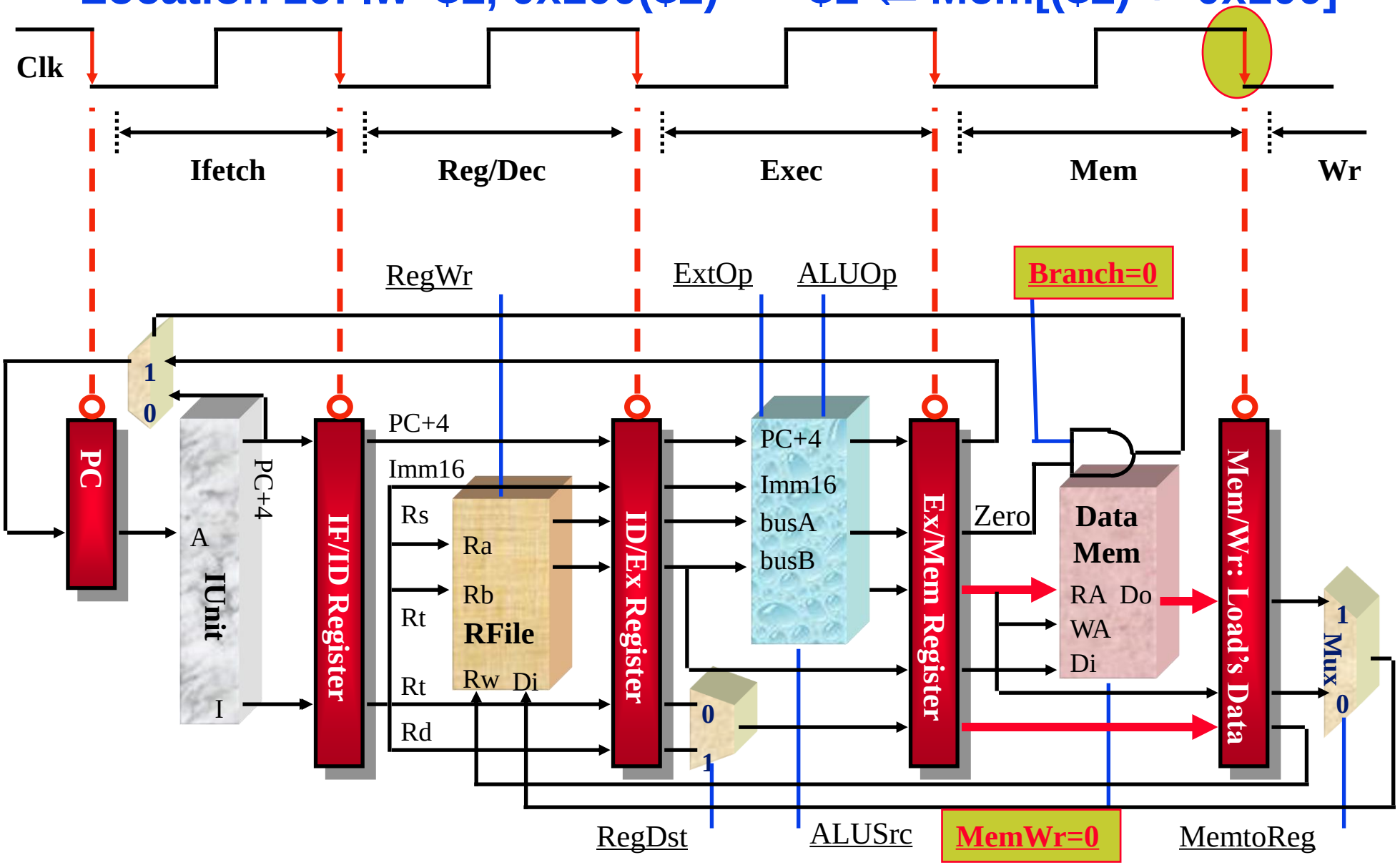


执行部件探秘


Clk
Exec
Mem
ID/Ex Register
<< 2
ADD
Target
32
Ex/Mem: Load指 Memory Address
PC+4
32
busA
32
ALU
Zero
ALUout
32
busB
32
0
Mux
1
imm16
16
Extender
32
3
ALUctr
ALU
Control
3
ExtOp=1
ALUSrc=1
ALUOp=Add

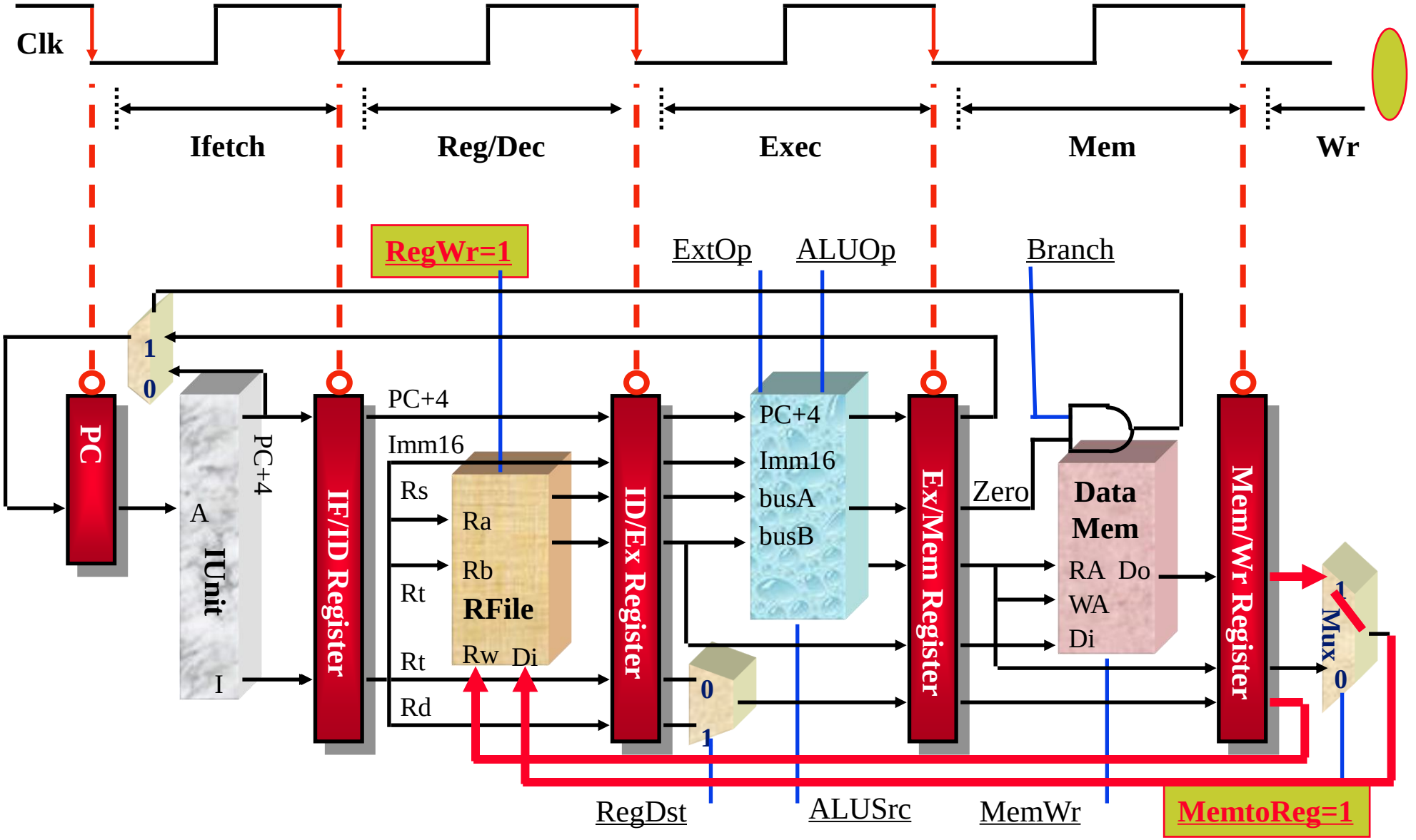
装入指令的存储器访问段

- Location 10: lw $1, 0x100($2) $1 ⇐ Mem[($2) + 0x100]

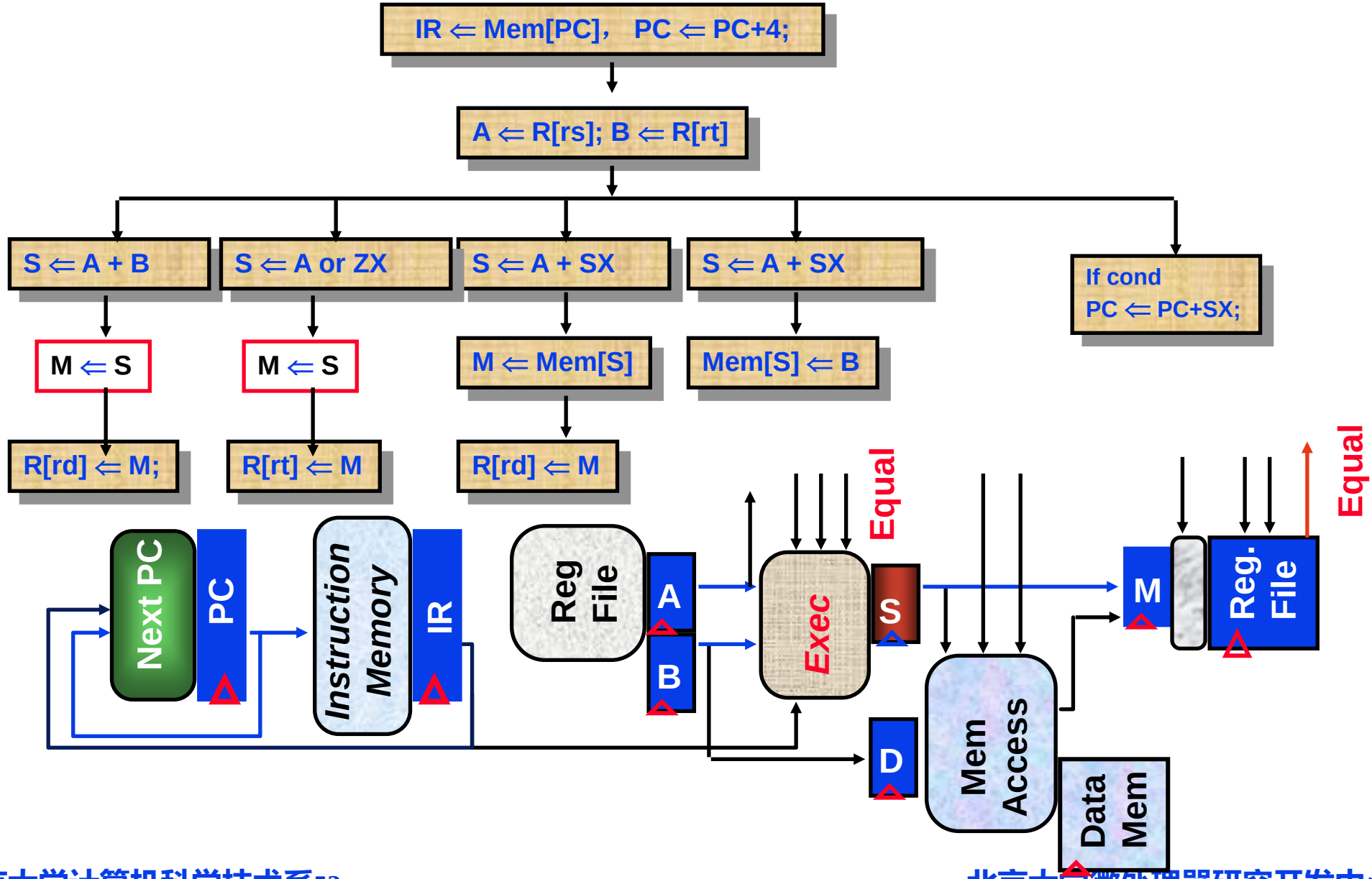


装入指令的回写段

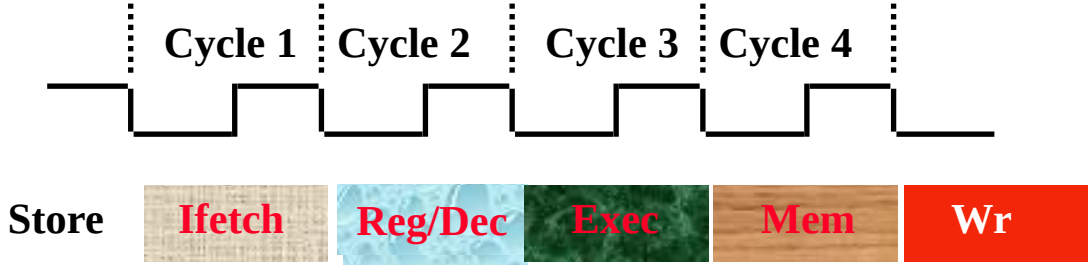
- Location 10: lw $1, 0x100($2) $1 ⇐ Mem[($2) + 0x100]



修改控制和数据通路

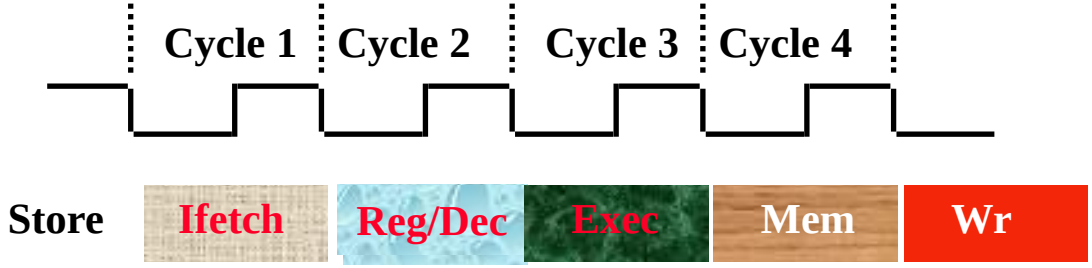

IR ⇐ Mem[PC]， PC ⇐ PC+4;
A ⇐ R[rs]; B ⇐ R[rt]
S ⇐ A + B
S ⇐ A or ZX
S ⇐ A + SX
S ⇐ A + SX
If cond
PC ⇐ PC+SX;
M ⇐ S
M ⇐ S
M ⇐ Mem[S]
Mem[S] ⇐ B
R[rd] ⇐ M;
R[rt] ⇐ M
R[rd] ⇐ M
Next PC
PC
Instruction Memory
IR
Reg File
A
B
Exec
S
Equal
D
Mem Access
Data Mem
M
Reg. File
Equal

存储指令的四段



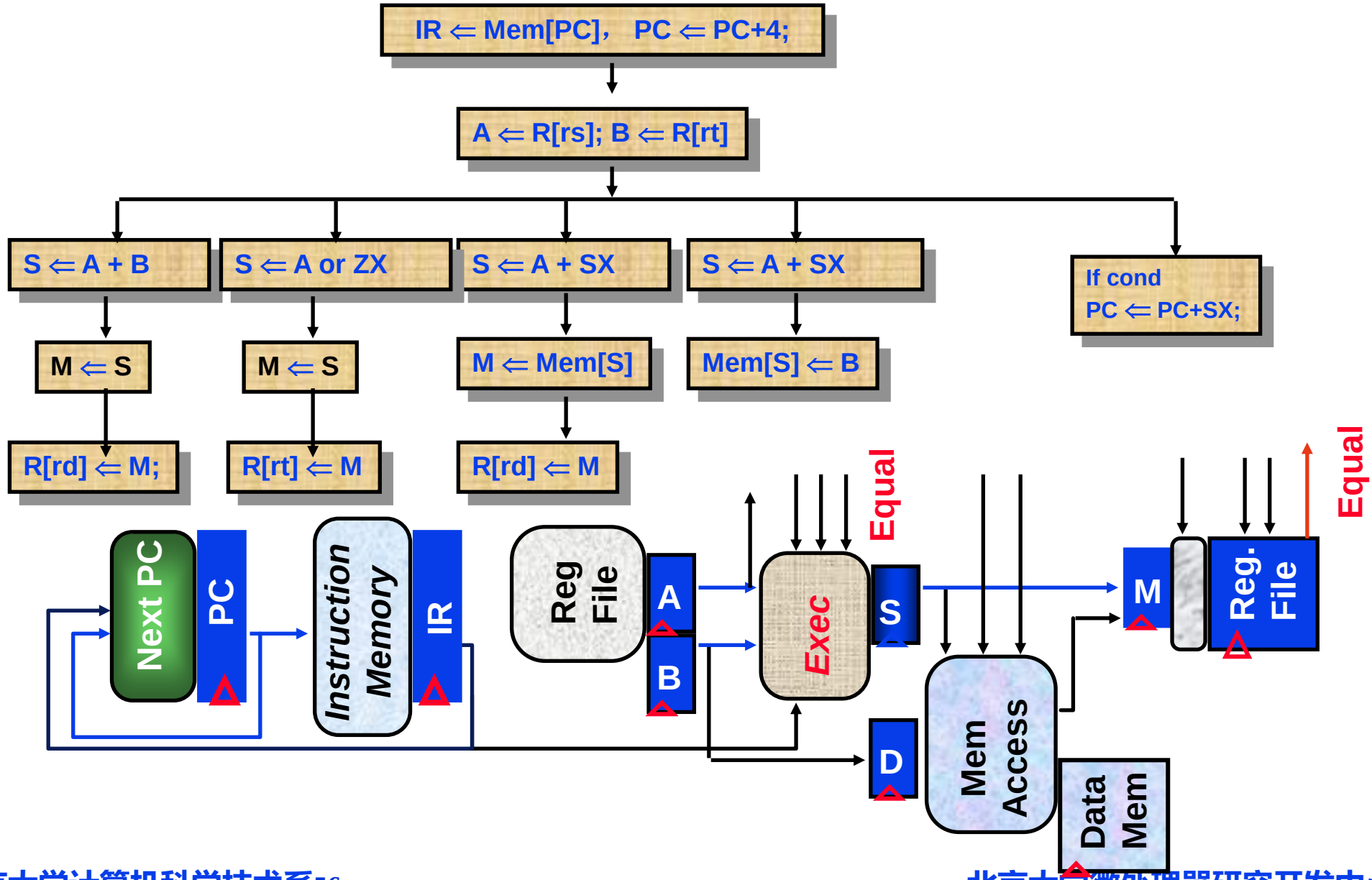
- **Ifetch:** 取指
  - 从指令存储器中读取指令
- **Reg/Dec:** 取寄存器和指令译码
- **Exec:**
  - 计算存储访问的地址
- **Mem:** 将数据写入到数据存储器

Beq指令的三段

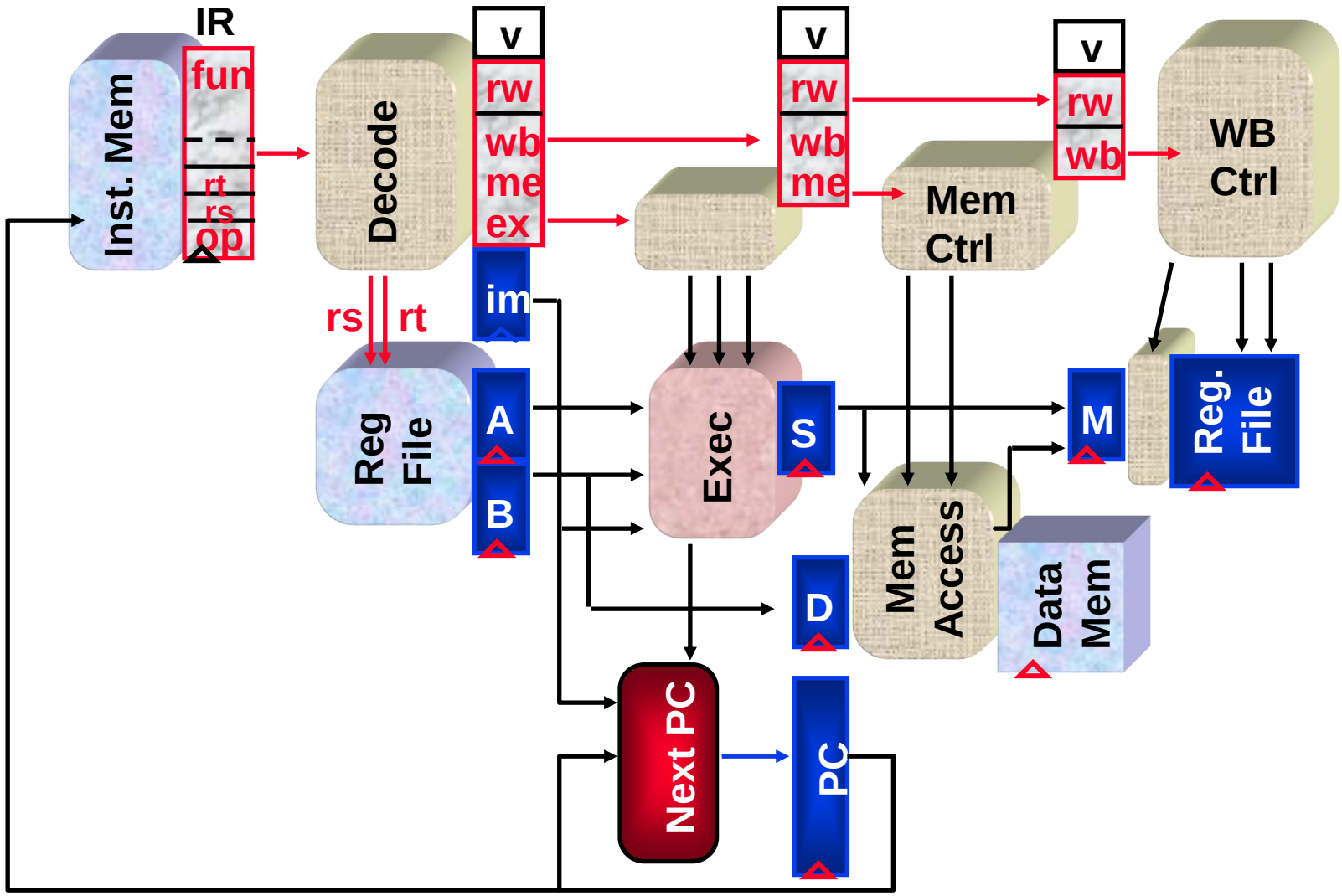


- **Ifetch:** 取指
  - 从指令存储器中读取指令
- **Reg/Dec:** 取寄存器和指令译码
- **Exec:**
  - 比较两个寄存器操作数
  - 择取正确的转移目标地址
  - 锁存到PC

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

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
v
rw
wb
me
Mem Ctrl
v
rw
wb
WB Ctrl
M
Reg. File
Mem Access
Data Mem
Next PC
PC

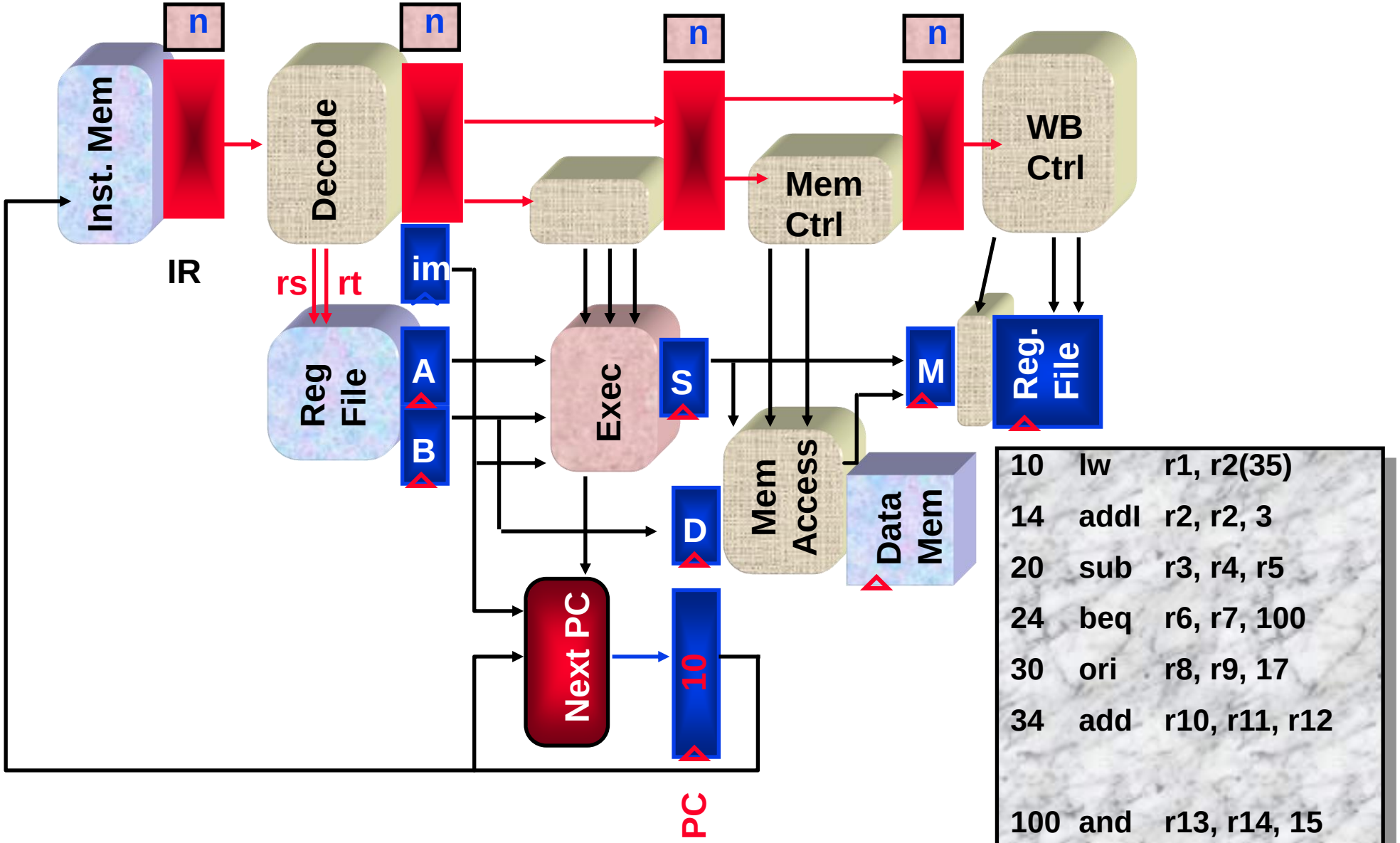
指令流水示例

```
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

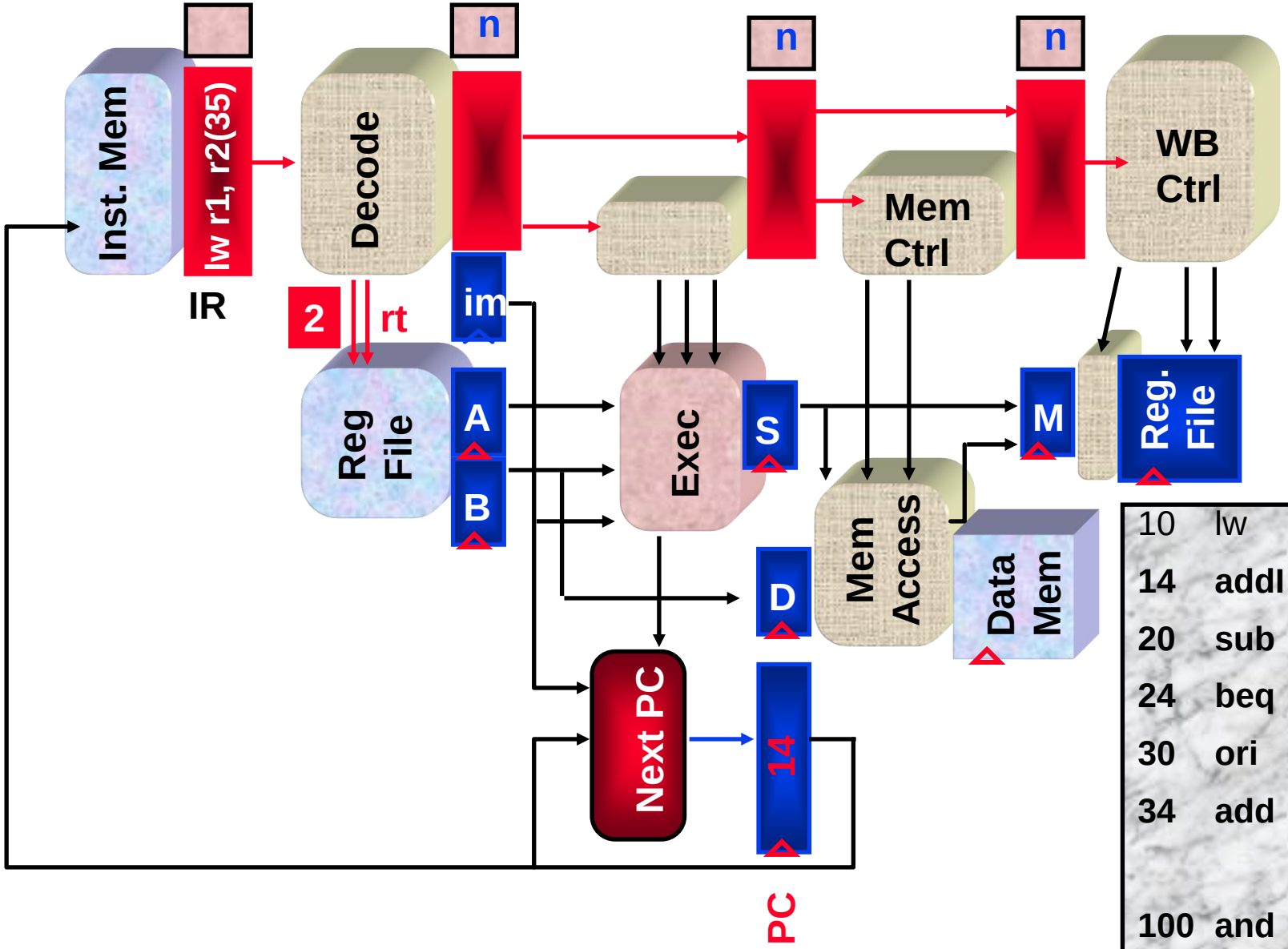
开始：从地址10取指




```
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
```

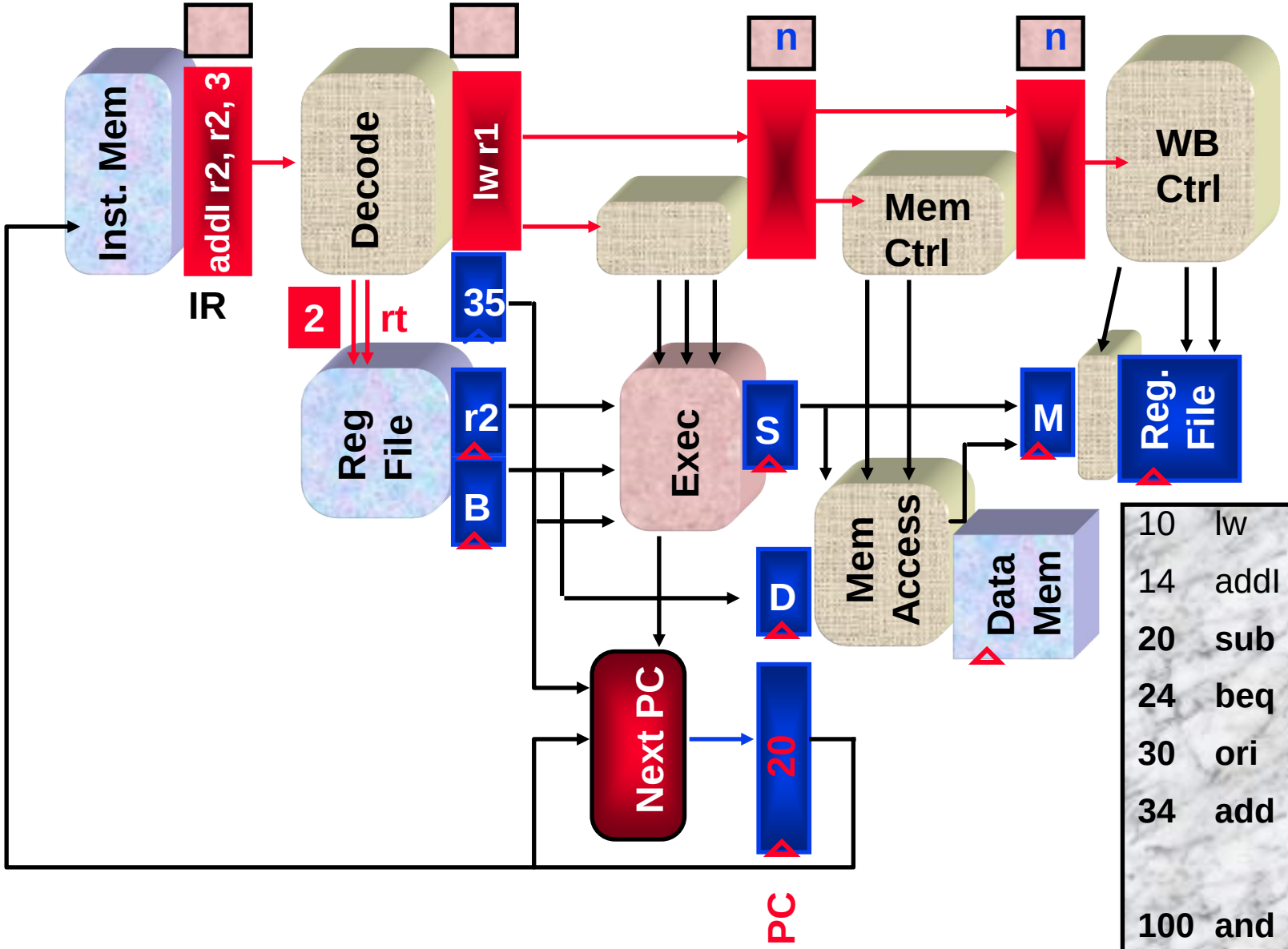
Fetch14 # Decode10




```
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
```

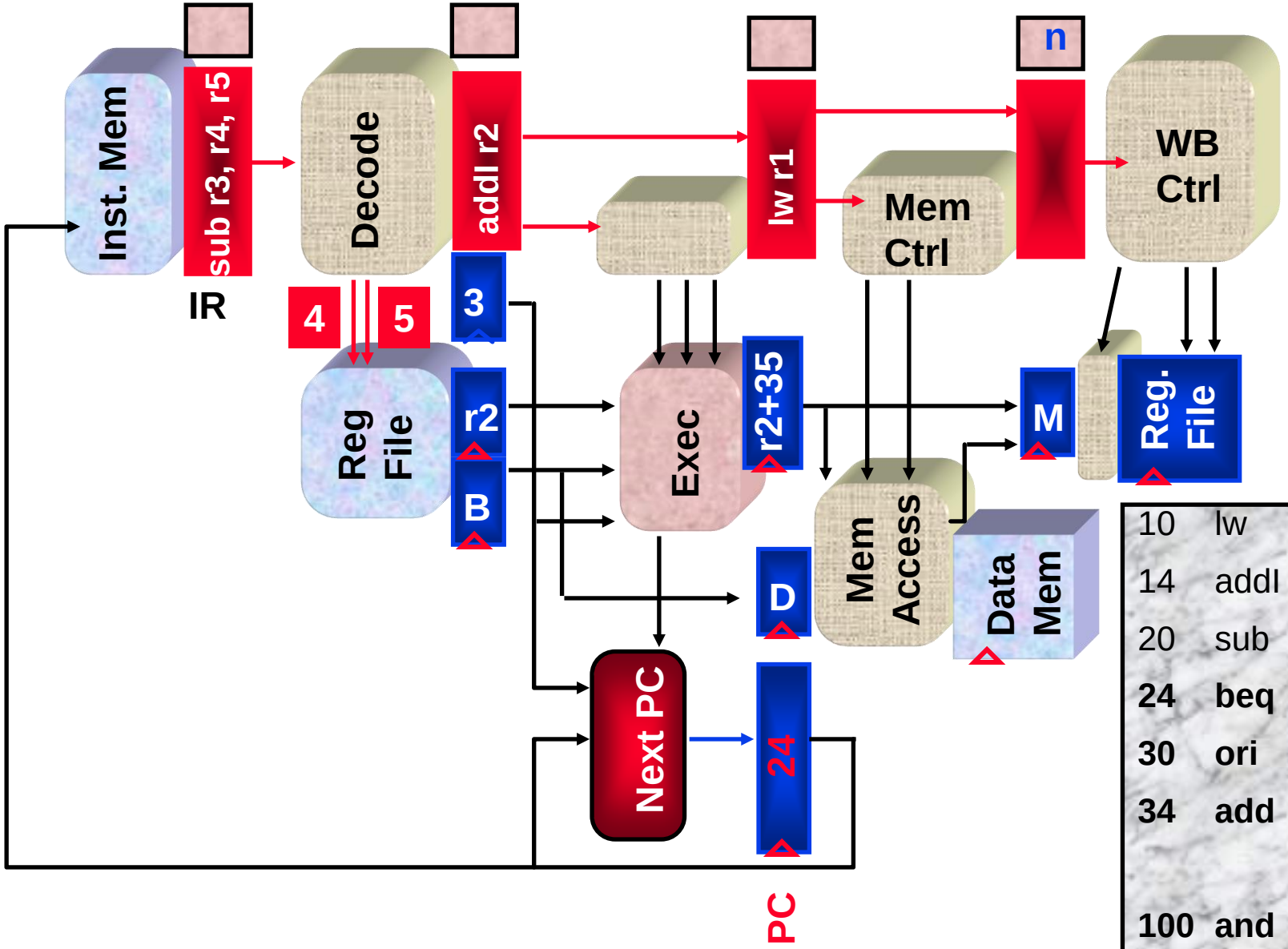
Fetch20 #Decode 14 # Exec10




```
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
```

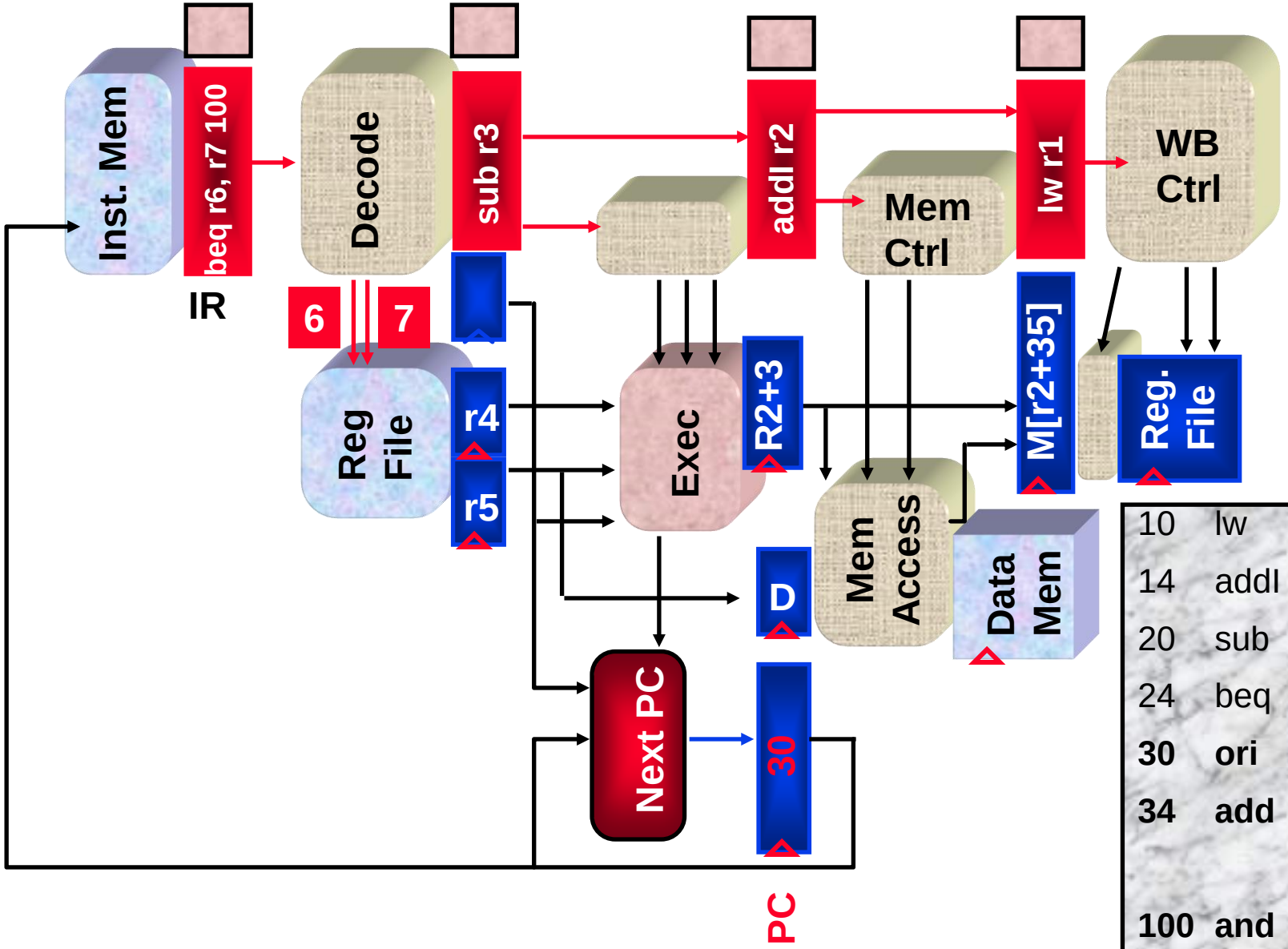
Fetch24 #Decode20 # Exec14 # Mem10




```
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
```

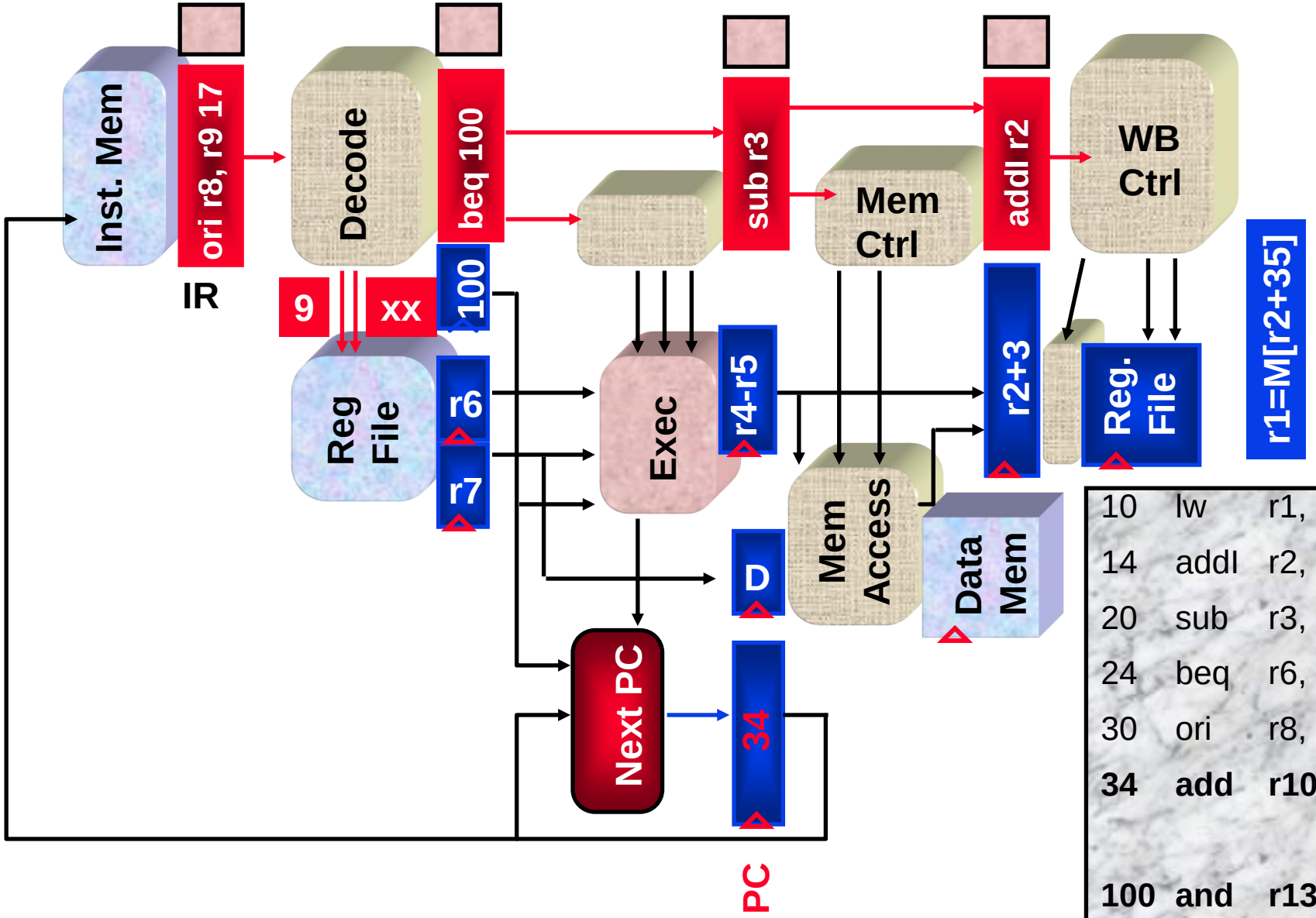
Fetch30#Decode24# Exec20#Mem14#WB10




```
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
```

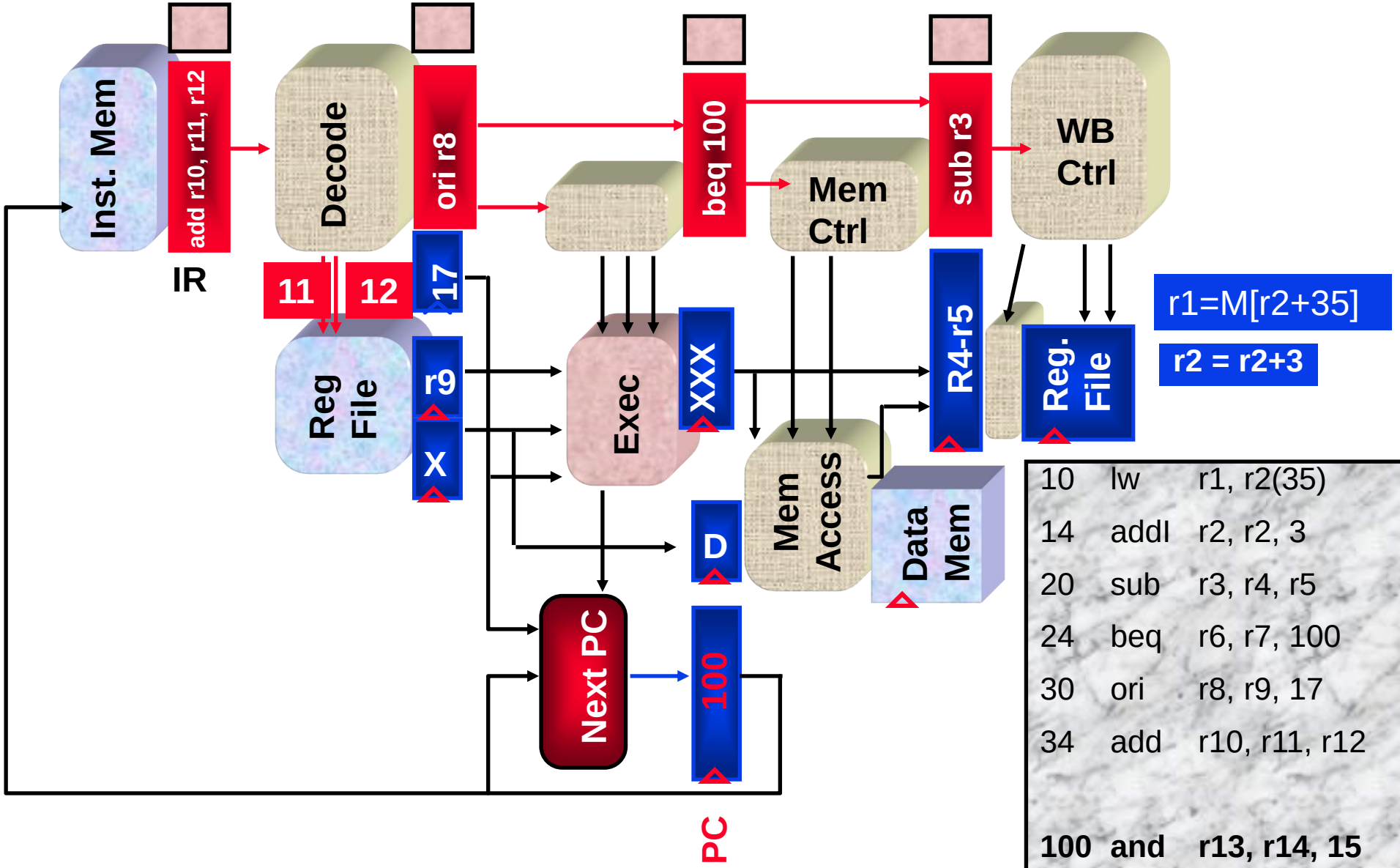
Fetch34#Decode30#Exec24#Mem20#WB14




```
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
```

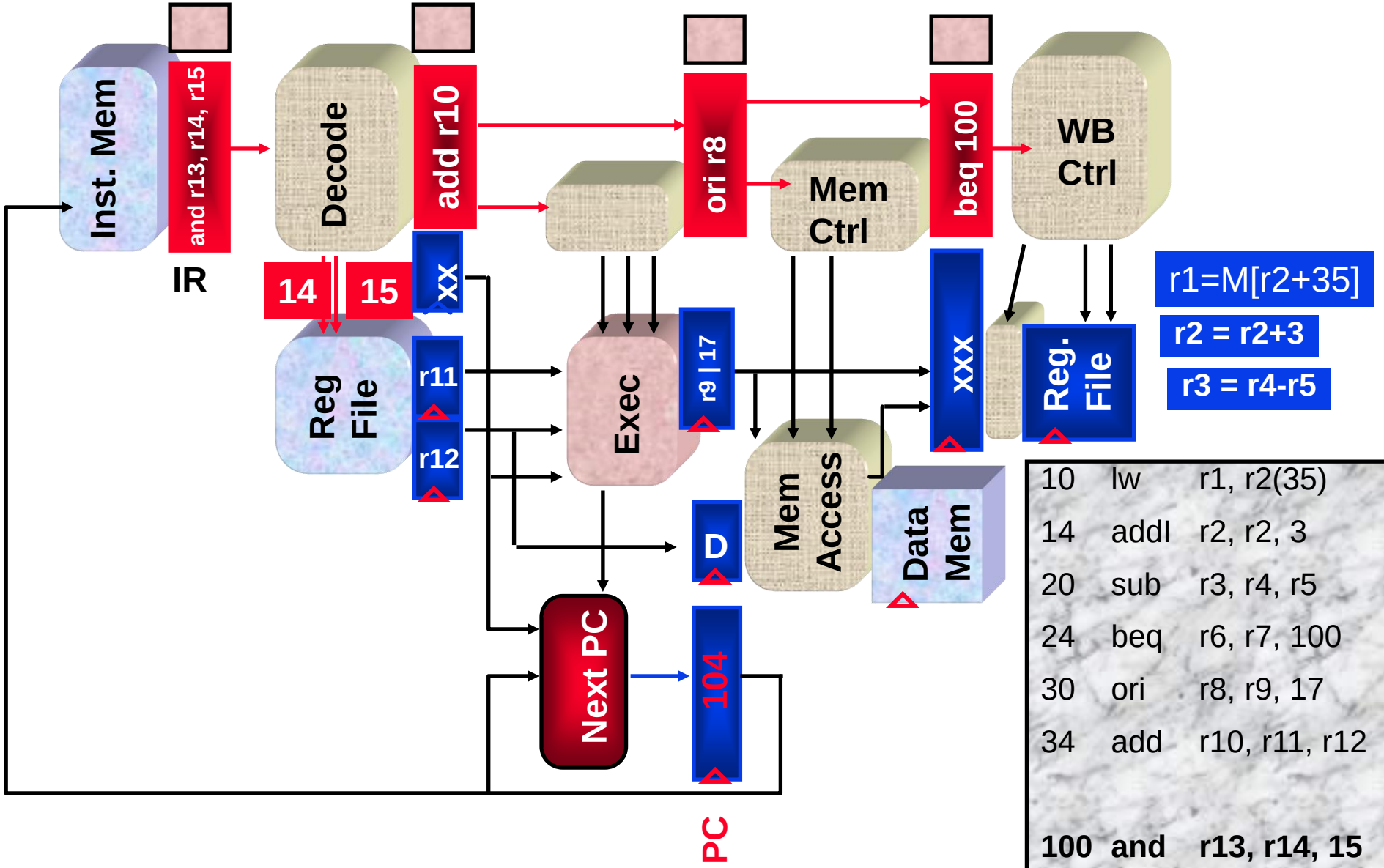
Fetch100#Decode34#Exec30#Mem24#WB20



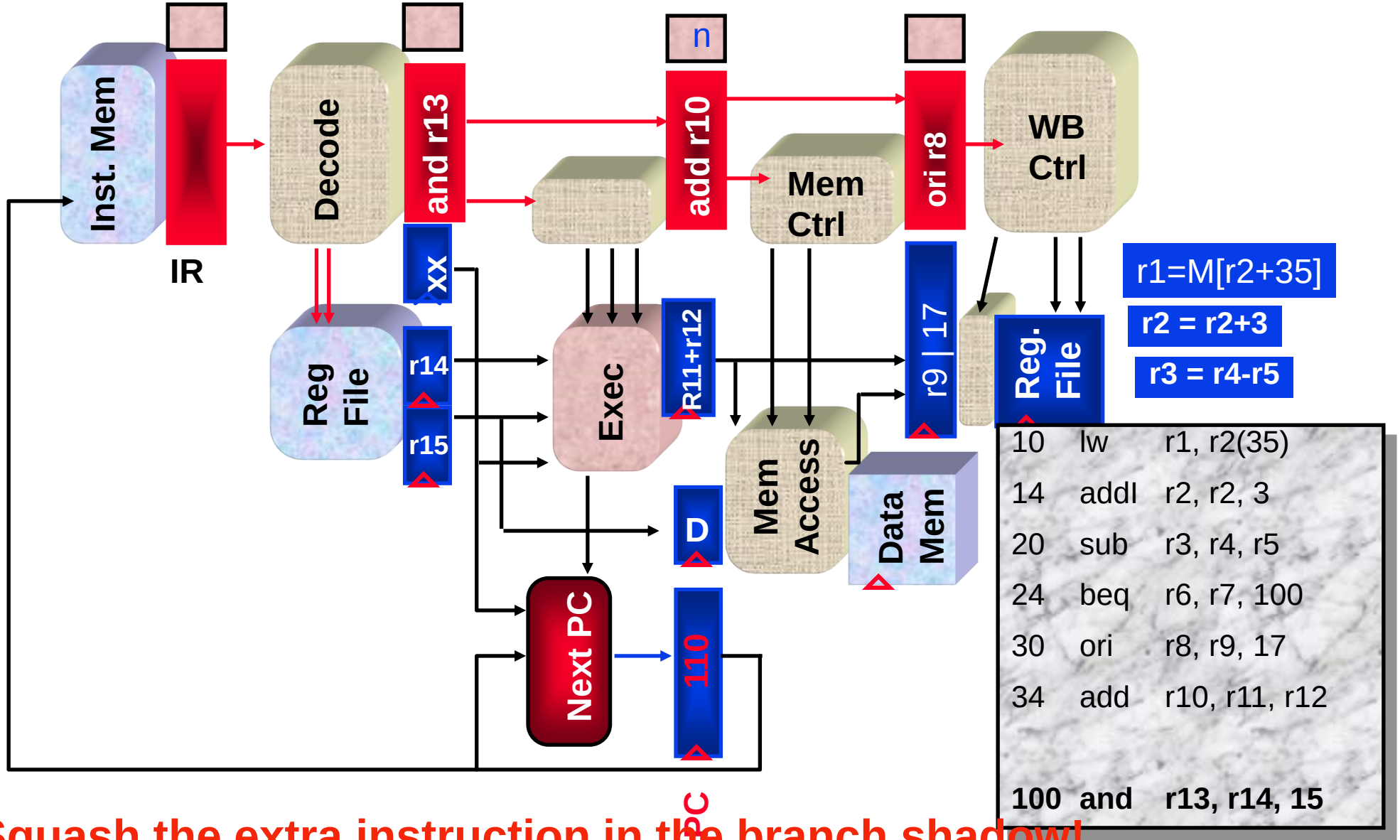

```
10   lw     r1, r2(35)
14   addI   r2, r2, 3
20   sub    r3, r4, r5
24   beq    r6, r7, 100
30   ori    r8, r9, 17
34   add    r10, r11, r12

100  and    r13, r14, 15
```

Fetch104#Decode100#Exec34#Mem30#WB24

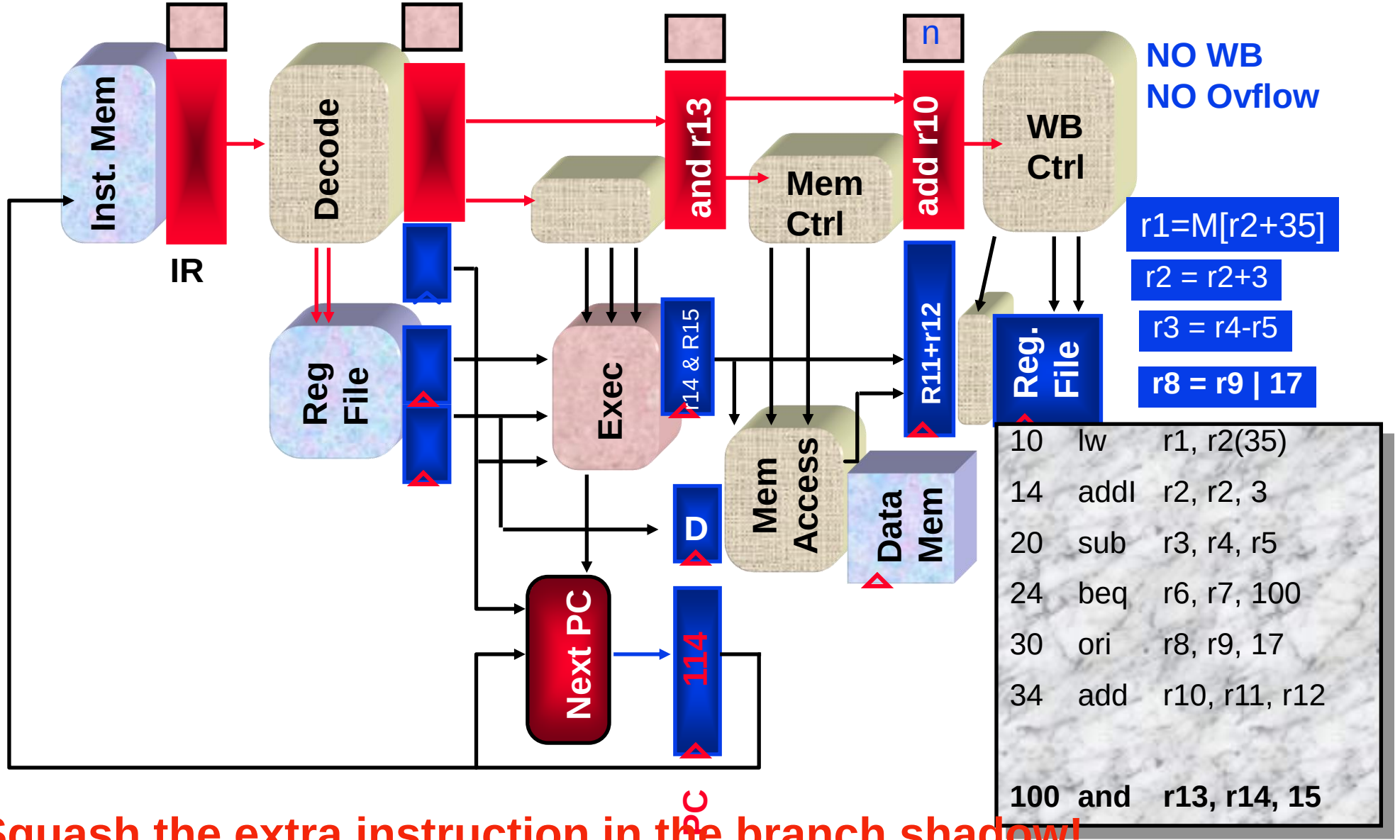


Fetch110并Decode104并Exec100并Mem34并WB30



Squash the extra instruction in the branch shadow!

Fetch114#Dcd110#Exec104#Mem100#WB34



Squash the extra instruction in the branch shadow!

总结

- 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**
- 哪些方面使得流水线难以实现?
  - 结构冒险: 假设我们只有一个存储器
  - 控制冒险: 需要考虑转移指令
  - 数据冒险: 一条指令依赖于上一条指令的结果

总结

- 我们将建造一条简单的流水线, 并考虑上述因素
- 我们将讨论现代微处理器, 并分析它的真正难点:
  - 意外事件处理
  - 利用乱序执行等技术改进性能