

# Instruction Pipelining

*Ryan Lee*

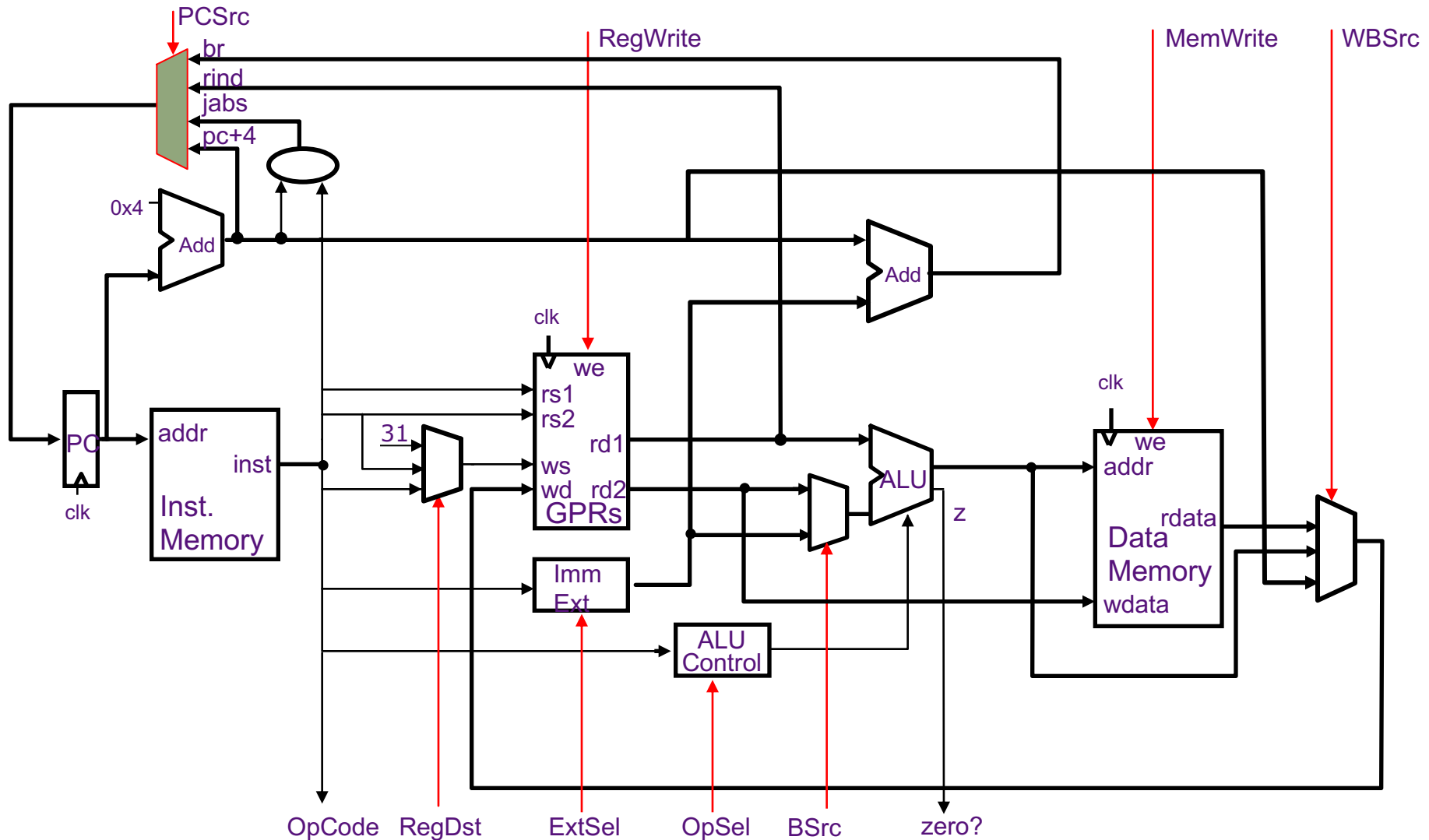
Computer Science and Artificial Intelligence Laboratory  
M.I.T.

# Announcements

---

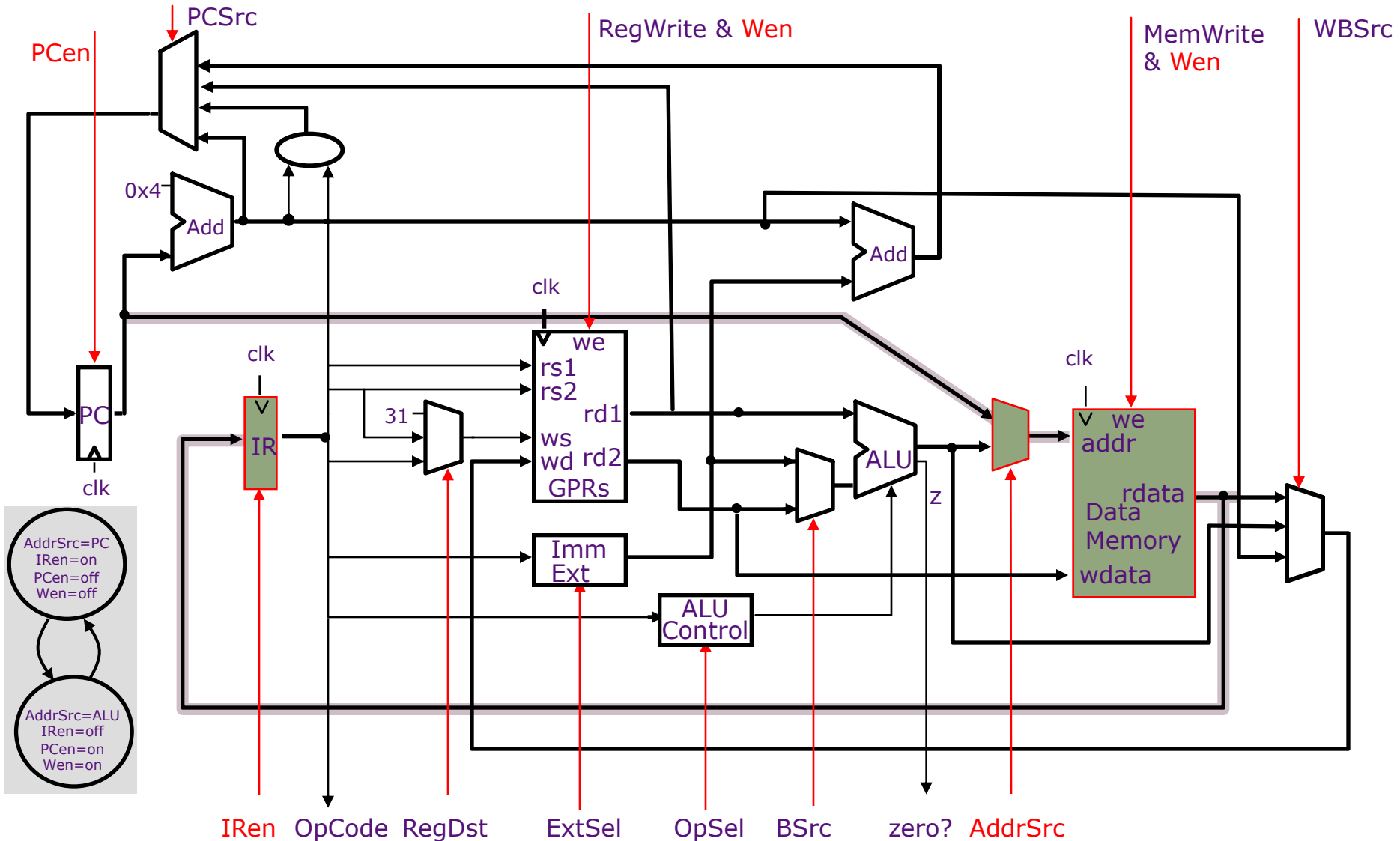
- Lab 1 released later today
  - Designing 3 different cache models using Pin
  - Due Oct. 1
- Please view the Pin tutorial video posted on the course website (under Recitation tab)
- Contact me if you cannot get access to lab machines

# Reminder: Harvard-Style Single-Cycle Datapath for MIPS

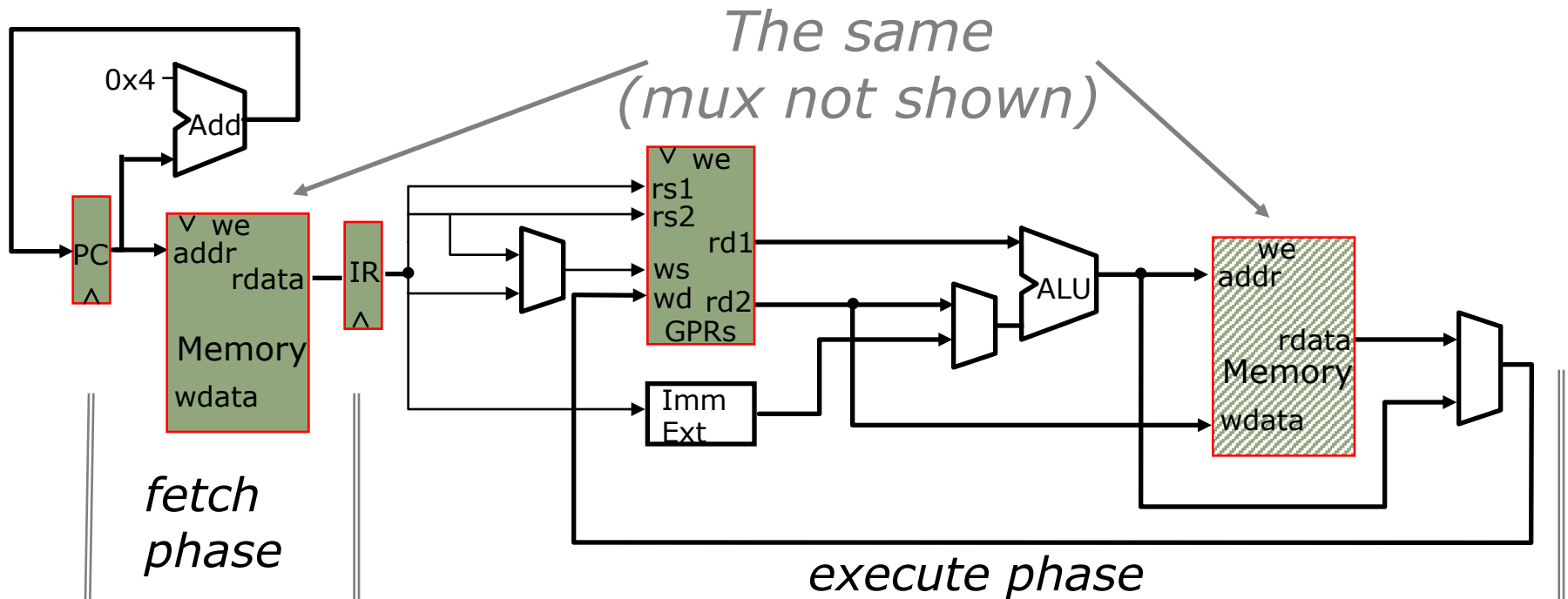


# Reminder: Princeton Microarchitecture

## Datapath & Control for 2 cycles-per-instruction



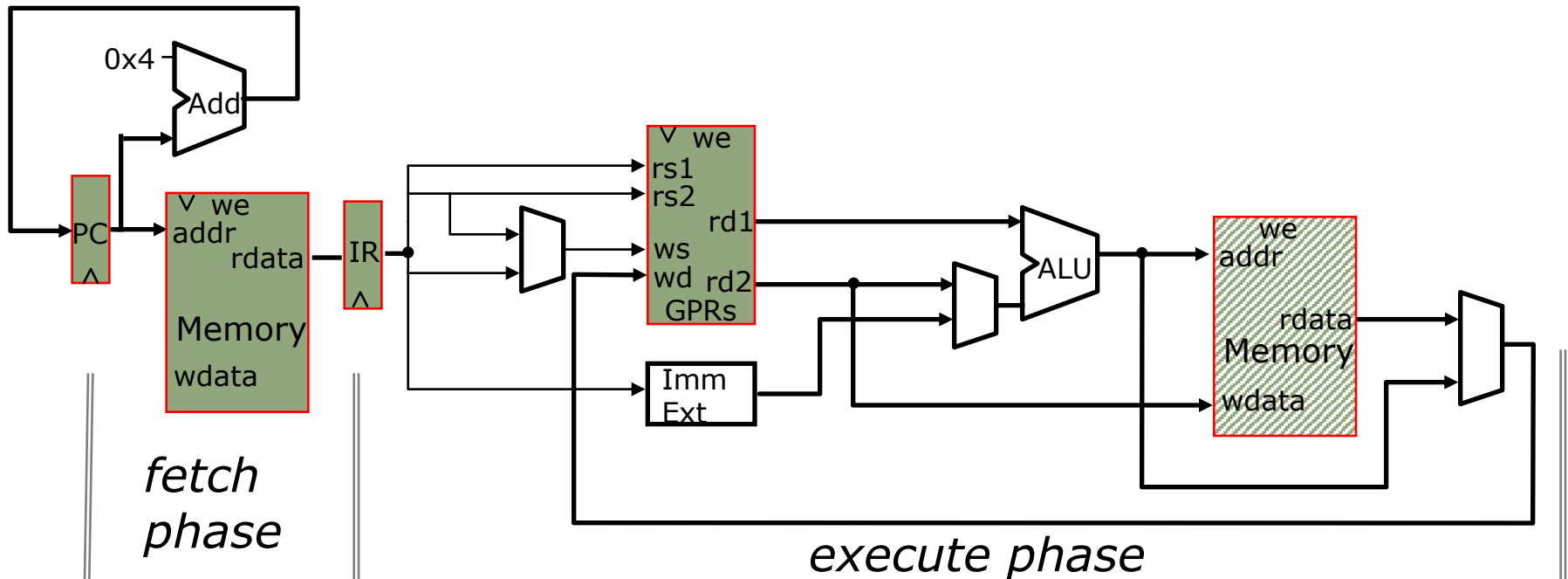
# Princeton Microarchitecture (redrawn)



Only one of the phases is active in any cycle  
⇒ a lot of datapath not used at any given time

# Princeton Microarchitecture

## *Overlapped execution*



*Can we overlap instruction fetch and execute?*

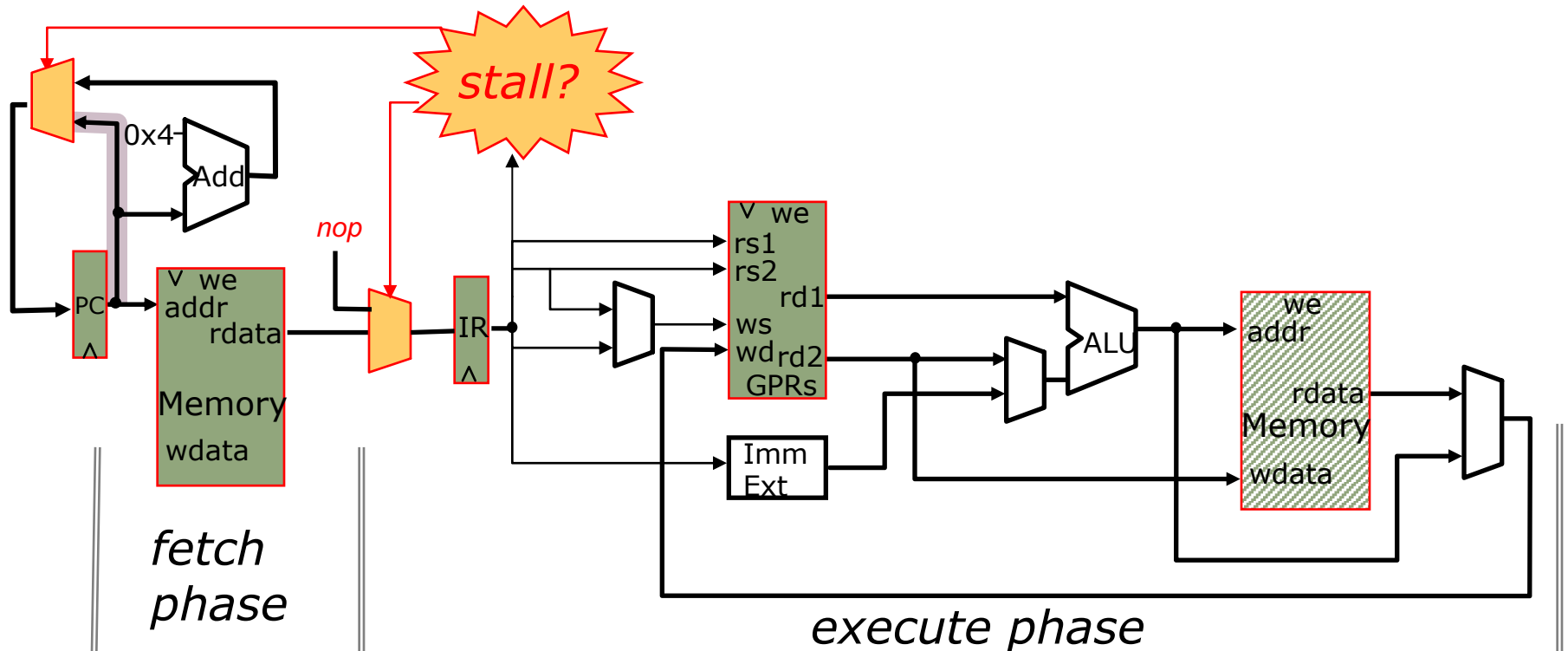
*Yes, unless IR contains a Load or Store*

*Which action should be prioritized? Execute*

*What do we do with Fetch? Stall it How?*

# Stalling the instruction fetch

## Princeton Microarchitecture



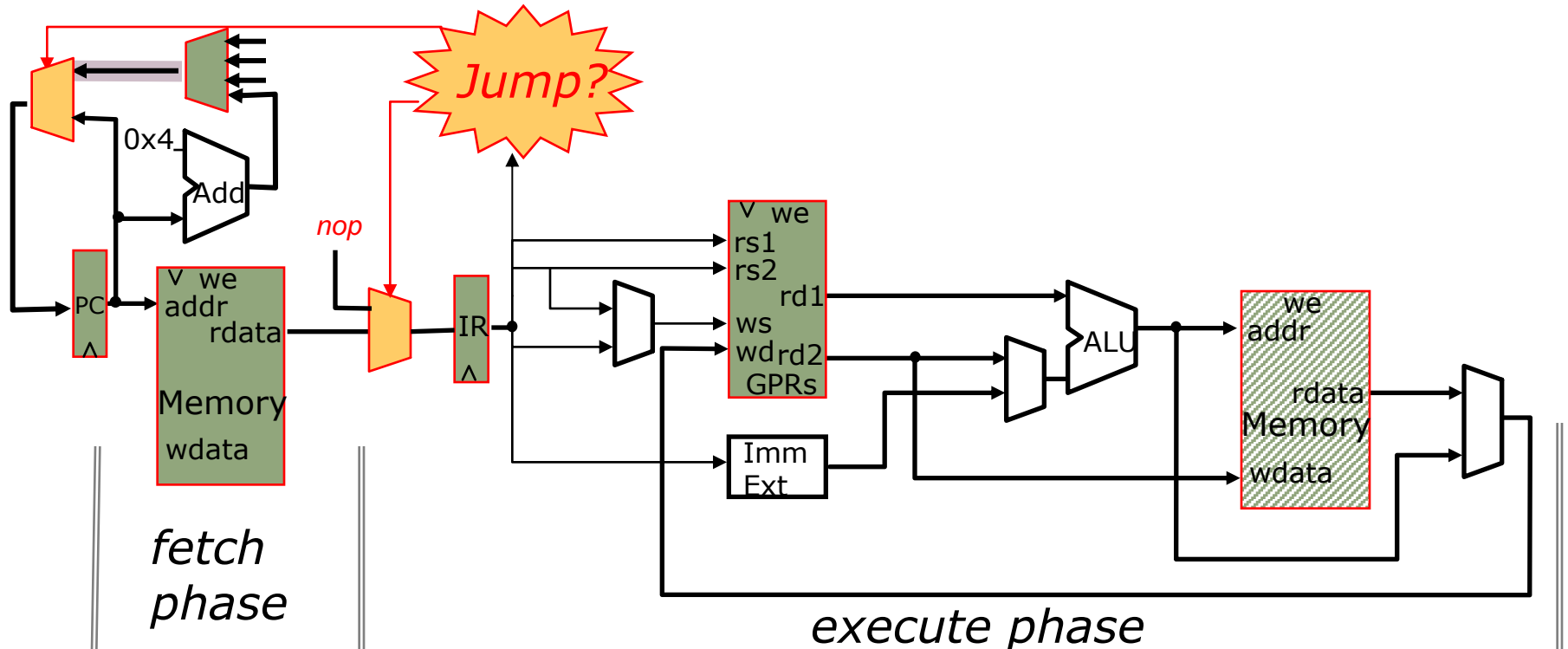
When stall condition is indicated

- *don't fetch a new instruction and don't change the PC*
- *insert a nop in the IR*
- *set the Memory Address mux to ALU (not shown)*

*What if IR contains a jump or branch instruction?*

# Need to stall on branches

## Princeton Microarchitecture

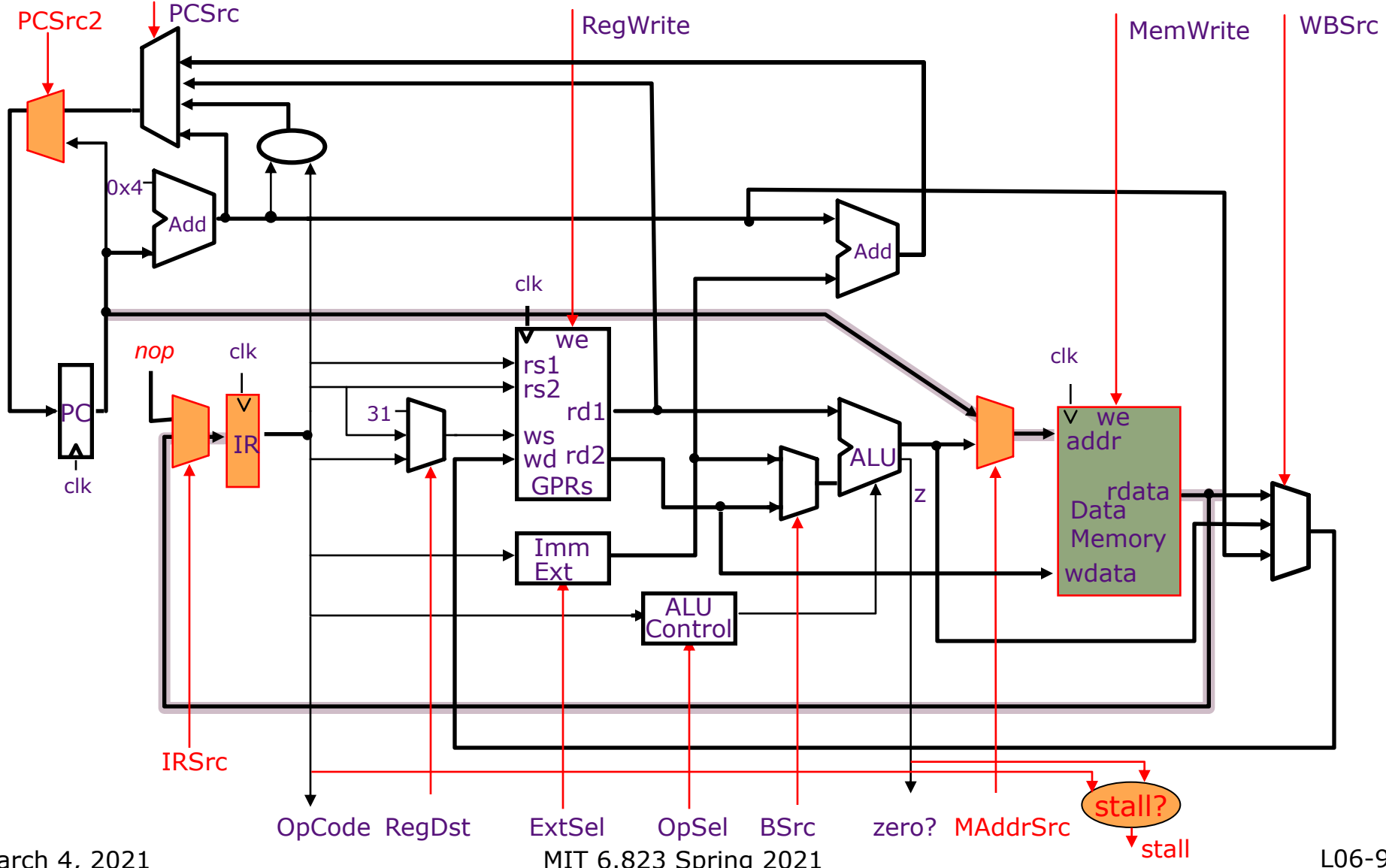


When IR contains a jump or taken branch

- *no "structural conflict" for the memory*
- *but we do not have the correct PC value in the PC*
- *memory cannot be used – Address Mux setting is irrelevant*
- *insert a nop in the IR*
- *insert the nextPC (branch-target) address in the PC*



# Pipelined Princeton Microarchitecture



# Pipelined Princeton: Control Table

Opcode	Stall	Ext Sel	B Src	Op Sel	Mem W	Reg W	WB Src	Reg Dst	PC Src1	PC Src2	IR Src	MAddr Src
ALU	no	*	Reg	Func	no	yes	ALU	rd	pc+4	npc	mem	pc
ALUi	no	sE <sub>16</sub>	Imm	Op	no	yes	ALU	rt	pc+4	npc	mem	pc
ALUiu	no	uE <sub>16</sub>	Imm	Op	no	yes	ALU	rt	pc+4	npc	mem	pc
LW	yes	sE <sub>16</sub>	Imm	+	no	yes	Mem	rt	pc+4	pc	nop	ALU
SW	yes	sE <sub>16</sub>	Imm	+	yes	no	*	*	pc+4	pc	nop	ALU
BEQZ <sub>Z=1</sub>	yes	sE <sub>16</sub>	*	0?	no	no	*	*	br	npc	nop	*
BEQZ <sub>Z=0</sub>	no	sE <sub>16</sub>	*	0?	no	no	*	*	pc+4	npc	mem	pc
J	yes	*	*	*	no	no	*	*	jabs	npc	nop	*
JAL	yes	*	*	*	no	yes	PC	R31	jabs	npc	nop	*
JR	yes	*	*	*	no	no	*	*	rind	npc	nop	*
JALR	yes	*	*	*	no	yes	PC	R31	rind	npc	nop	*
NOP	no	*	*	*	no	no	*	*	pc+4	npc	mem	pc

BSrc = Reg / Imm ; WBSrc = ALU / Mem / PC; IRSrc = nop/mem; MAddrSrc = pc/ALU

RegDst = rt / rd / R31; PCSrc1 = pc+4 / br / rind / jabs; PCSrc2 = pc/nPC

stall & IRSrc columns are identical

# Pipelined Princeton Architecture

---

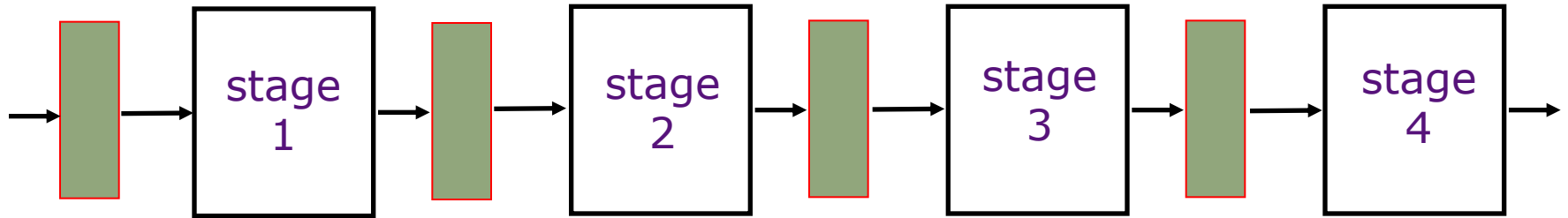
*Clock:*  $t_{\text{C-Princeton}} > t_{\text{RF}} + t_{\text{ALU}} + t_{\text{M}} + t_{\text{WB}}$

*CPI:*  $(1 - f) + 2f$  cycles per instruction  
where  $f$  is the fraction of  
instructions that cause a stall

*What is a likely value of  $f$ ?*

# An Ideal Pipeline

---

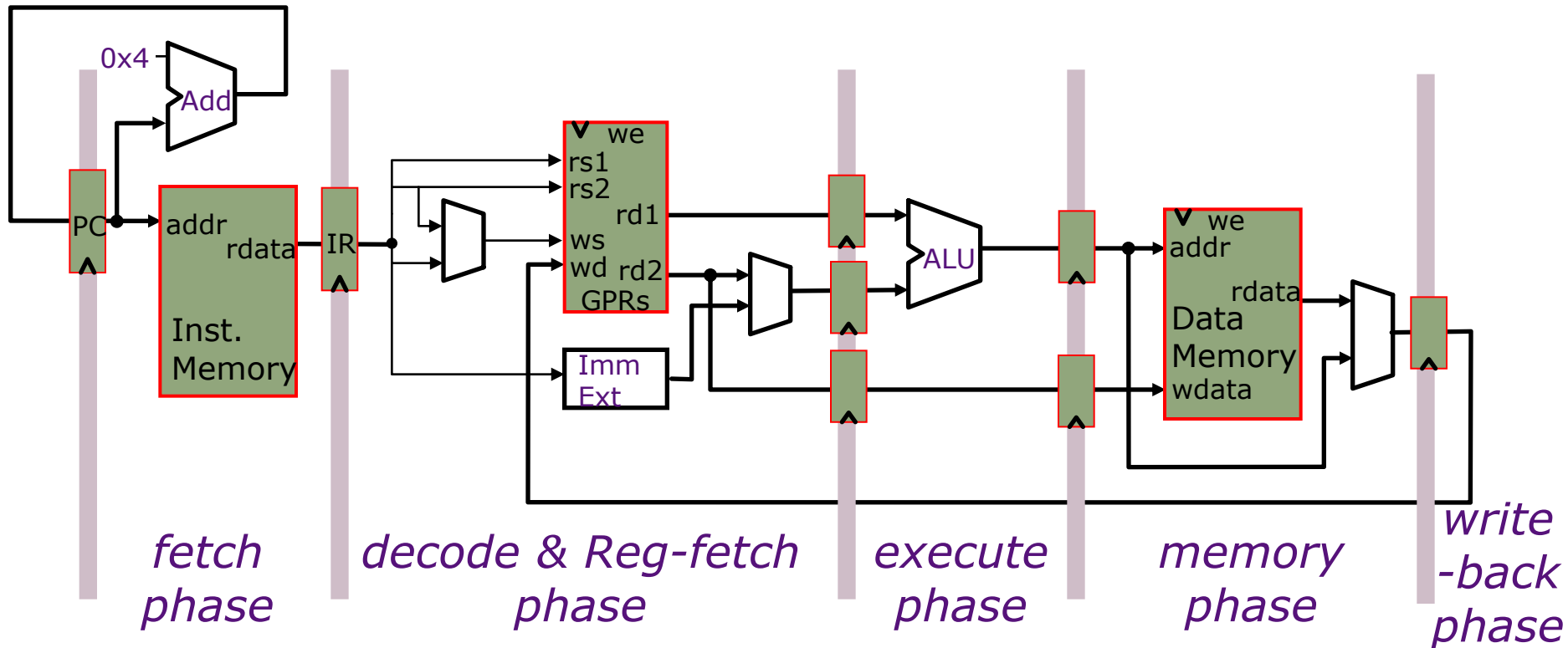


- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

*These conditions generally hold for industrial assembly lines.*

*But what about an instruction pipeline?*

# Pipelined Datapath



Clock period can be reduced by dividing the execution of an instruction into multiple cycles

$$t_c > \max \{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} \quad (= t_{DM} \text{ probably})$$

However, CPI will increase unless instructions are pipelined

# How to divide datapath into stages

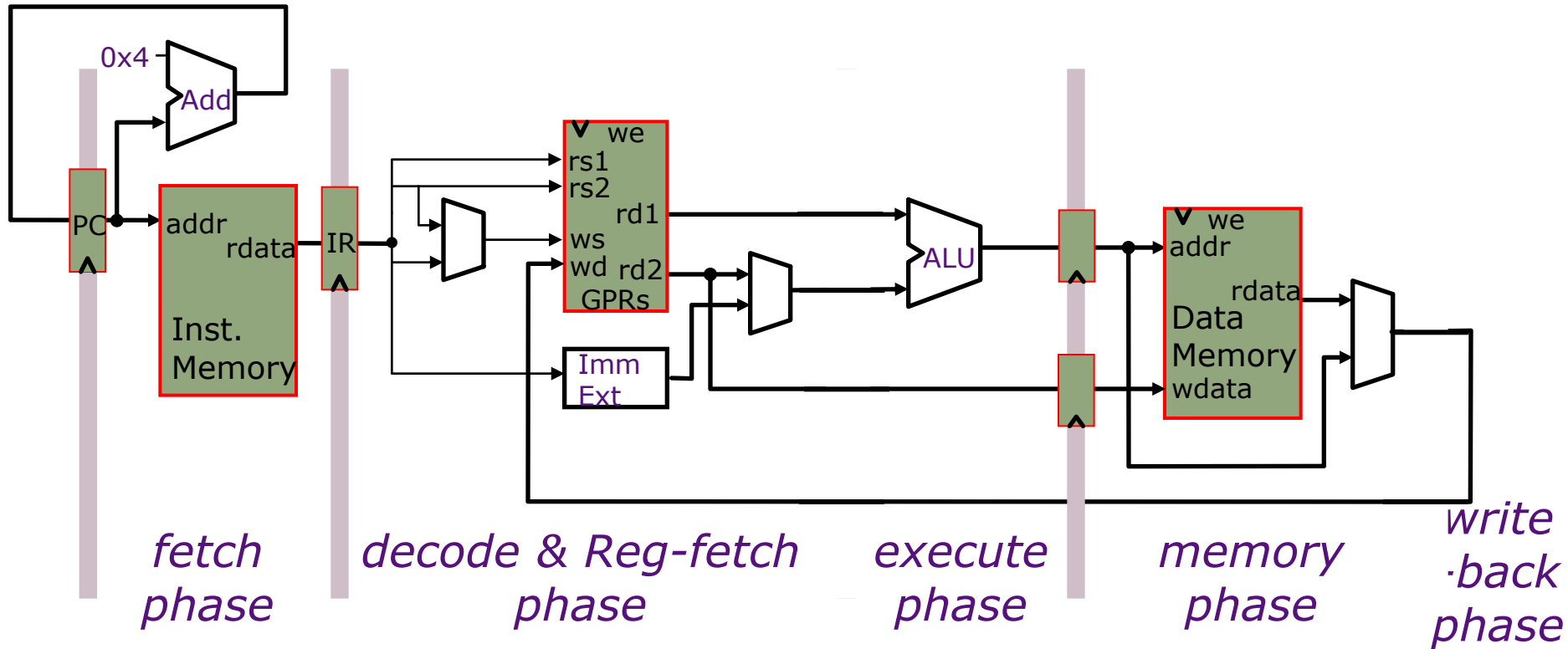
---

Suppose memory is significantly slower than other stages. For example, suppose

$$\begin{aligned}t_{IM} &= 10 \text{ units} \\t_{DM} &= 10 \text{ units} \\t_{ALU} &= 5 \text{ units} \\t_{RF} &= 1 \text{ unit} \\t_{RW} &= 1 \text{ unit}\end{aligned}$$

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance

# Alternative Pipelining



$$t_C > \max \{t_{IM}, t_{RF}+t_{ALU}, t_{DM}+t_{RW}\} = t_{DM} + t_{RW}$$

*⇒ increase the critical path by 10%*

Write-back stage takes much less time than other stages.  
 Suppose we combined it with the memory phase

# Maximum Speedup by Pipelining

---

Assumptions	Unpipelined $t_c$	Pipelined $t_c$	Speedup
1. $t_{IM} = t_{DM} = 10,$ $t_{ALU} = 5,$ $t_{RF} = t_{RW} = 1$ 4-stage pipeline	27	10	2.7
2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 4-stage pipeline	25	10	2.5
3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 5-stage pipeline	25	5	5.0

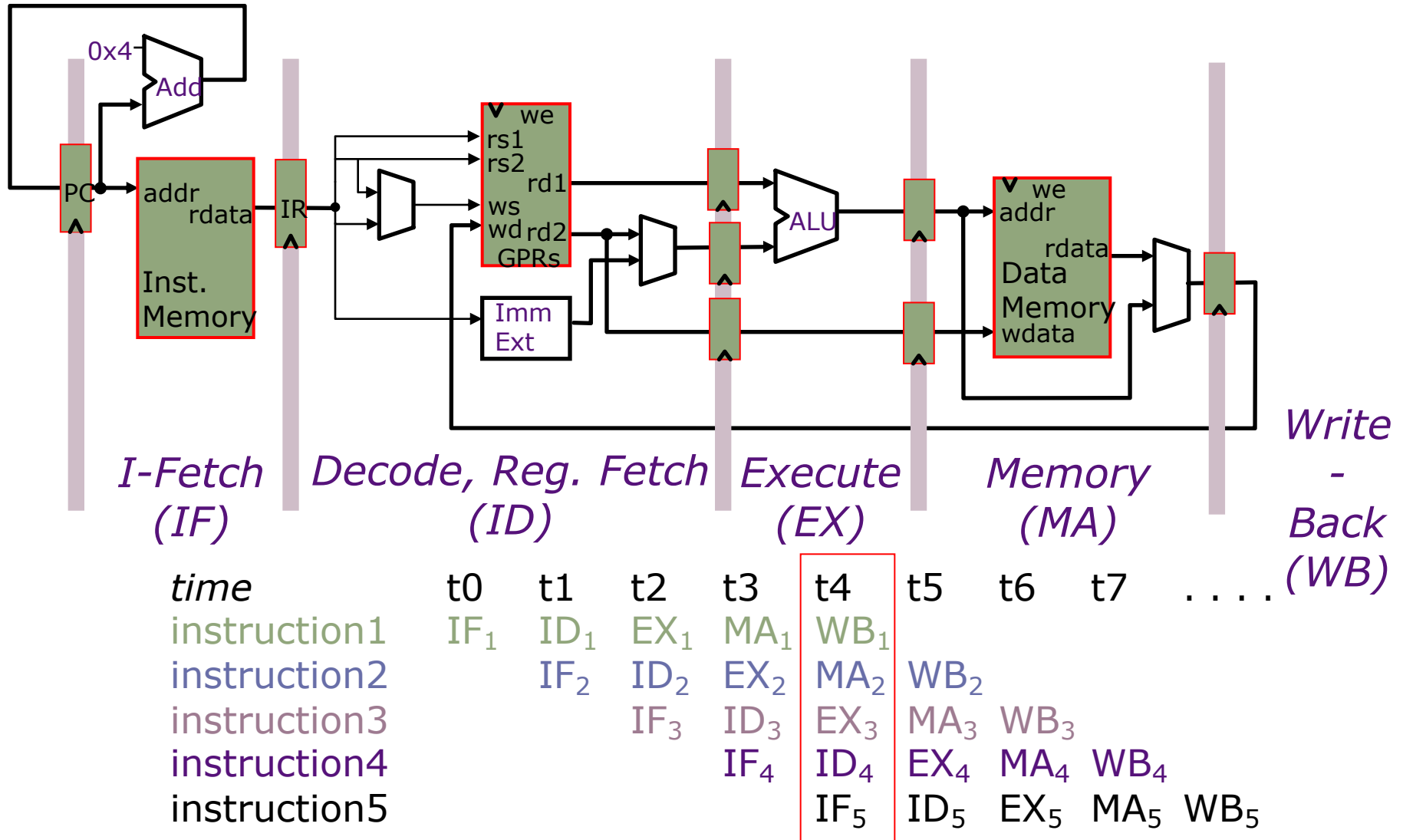
*What seems to be the message here?*

*One can achieve higher speedup with more pipeline stages*



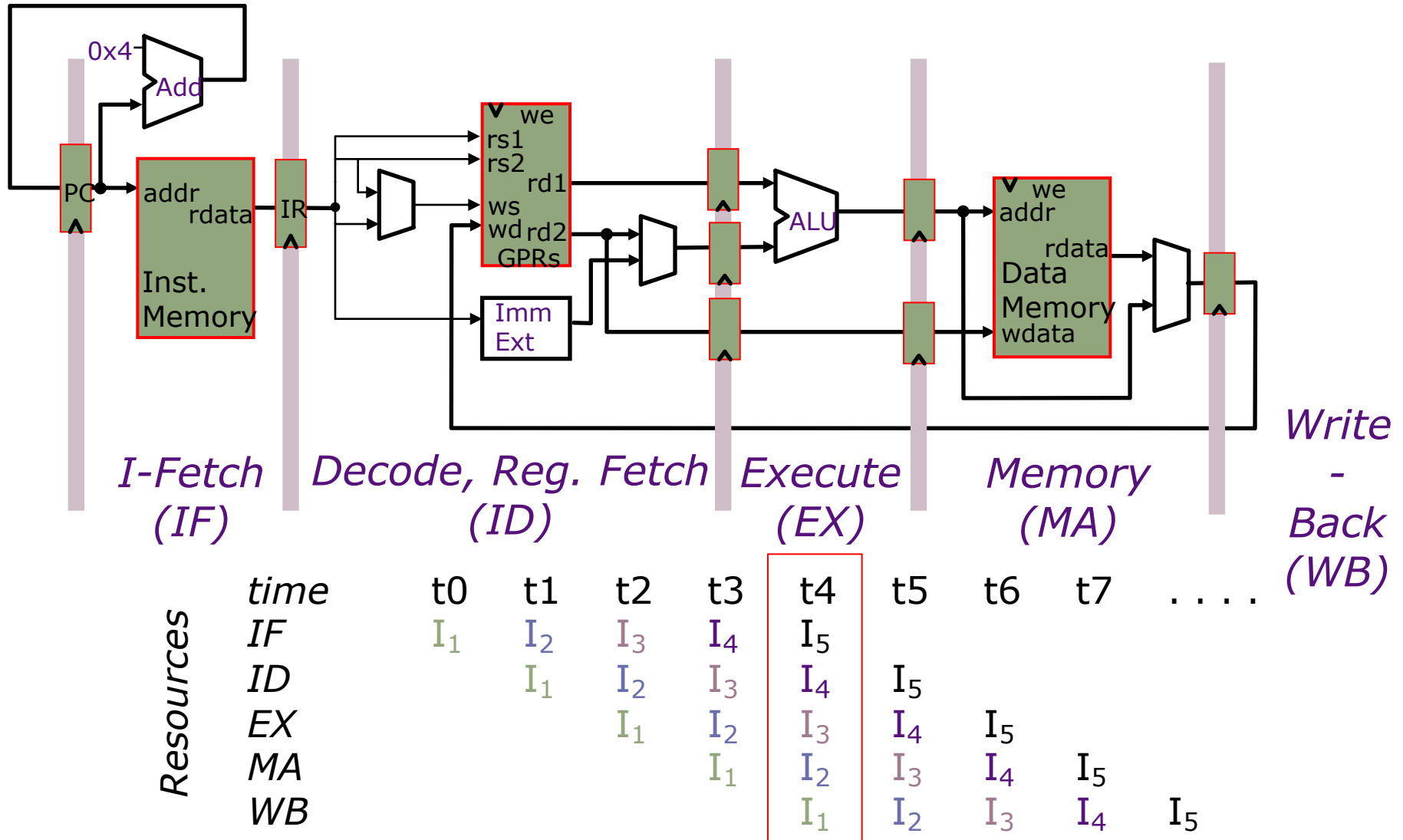
# 5-Stage Pipelined Execution

## Instruction Flow Diagram



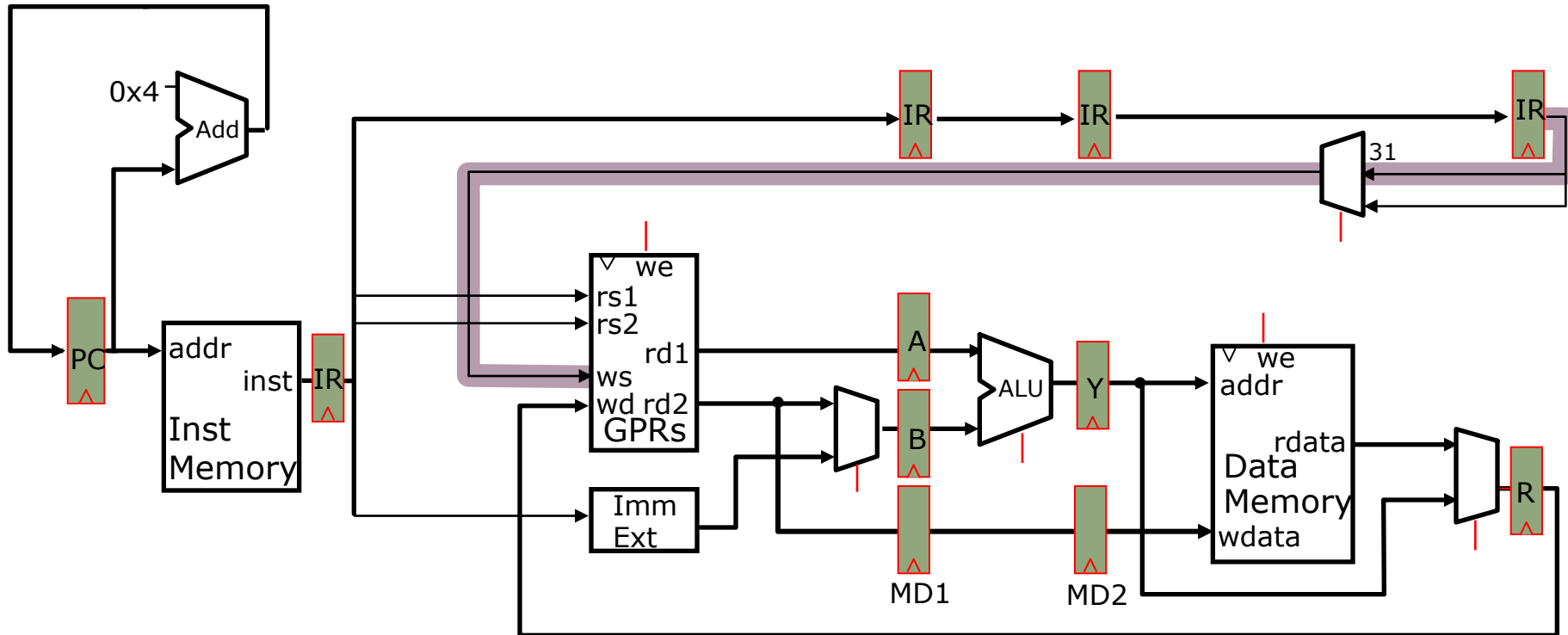
# 5-Stage Pipelined Execution

## Resource Usage Diagram



# Pipelined Execution

## *ALU Instructions*

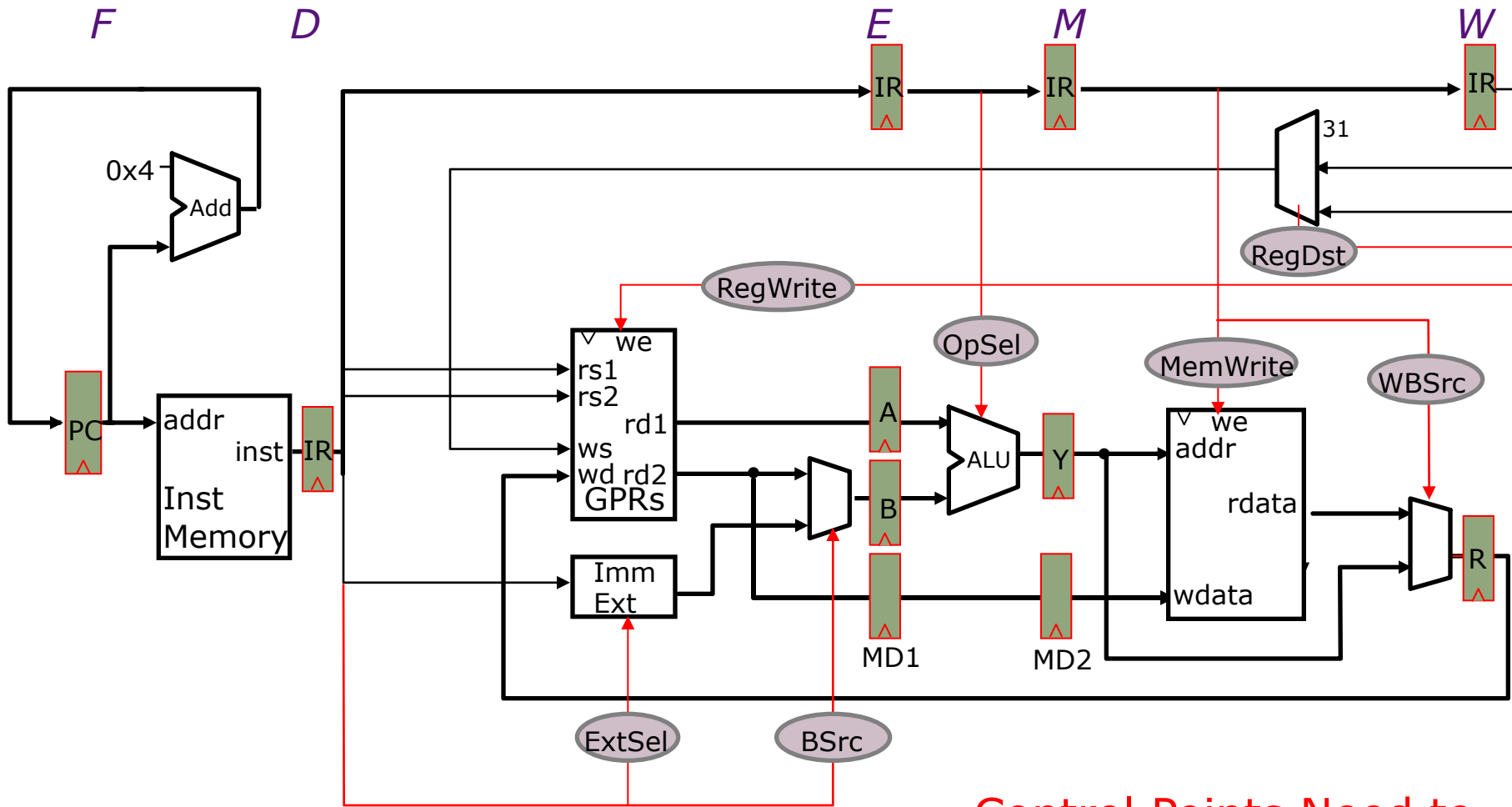


*Not quite correct!*

*We need an Instruction Reg (IR) for each stage*

# Pipelined MIPS Datapath

*without jumps*



*What else is needed?*

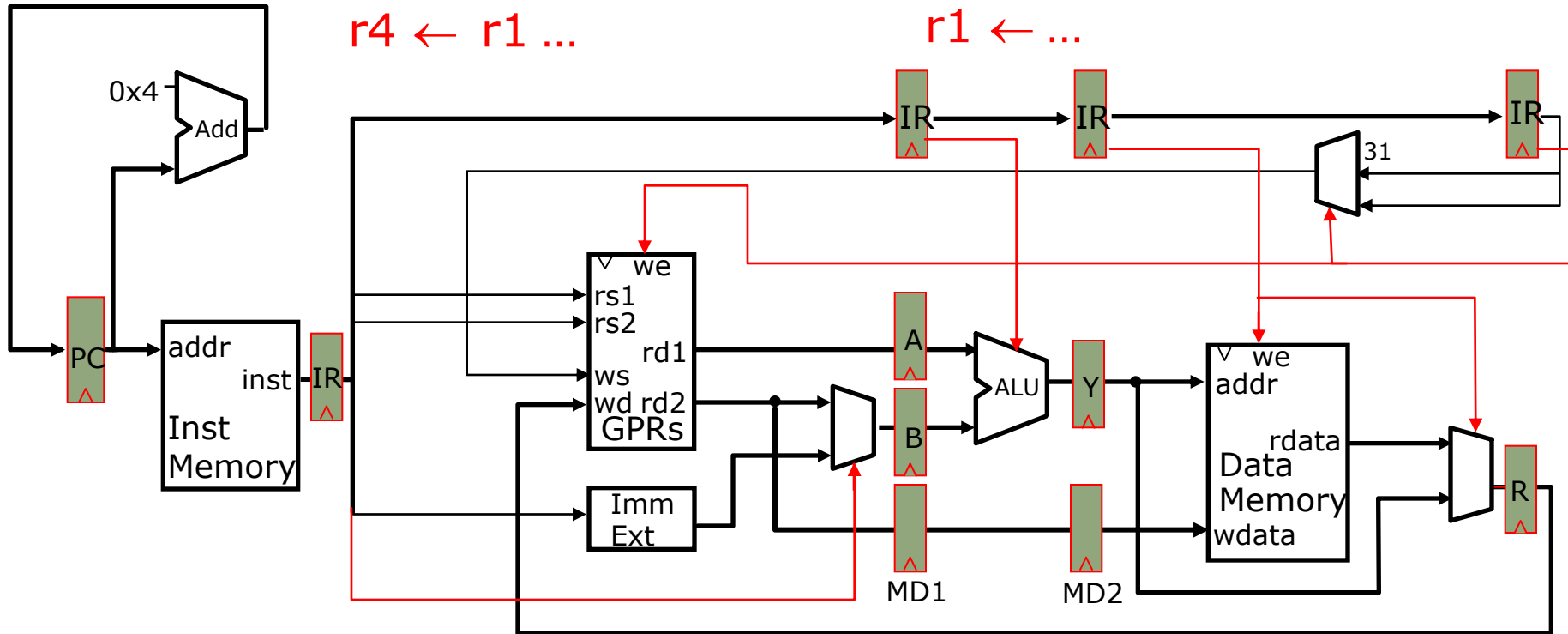
**Control Points Need to Be Connected**

# How instructions can interact with each other in a pipeline

---

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline  
→ *structural hazard*
- An instruction may depend on a value produced by an earlier instruction
  - Dependence may be for a data calculation  
→ *data hazard*
  - Dependence may be for calculating the next PC  
→ *control hazard (branches, interrupts)*

# Data Hazards



...  
 $r1 \leftarrow r0 + 10$   
 $r4 \leftarrow r1 + 17$   
 ...

*r1 is stale. Oops!*

# Resolving Data Hazards

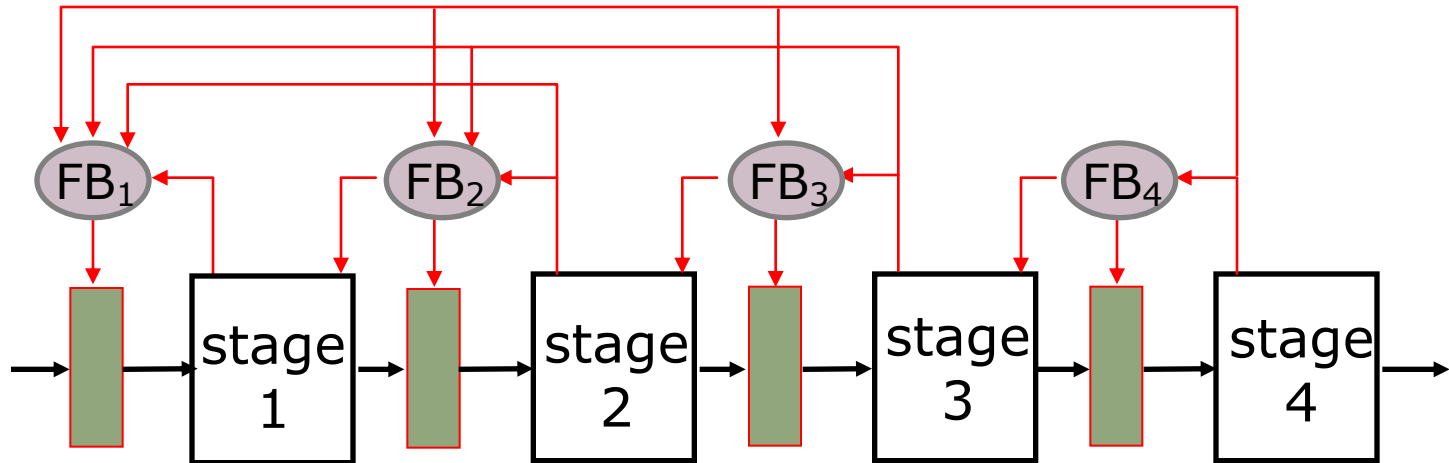
---

*Use strategy from Princeton Pipeline:*

*Wait for the result to be available by freezing earlier pipeline stages → **stall***

# Feedback to Resolve Hazards

---

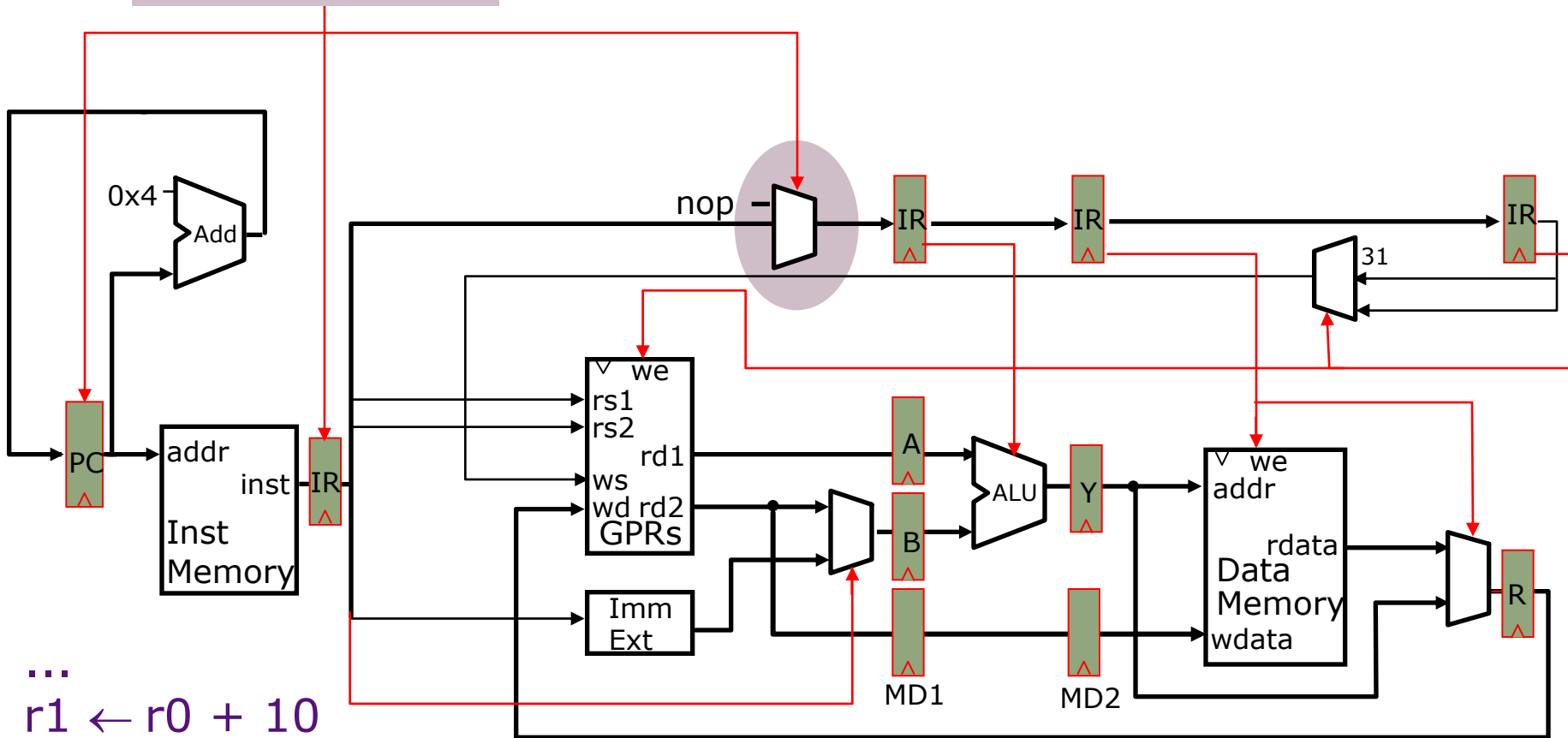


- Later stages provide dependence information to earlier stages which can *stall instructions*
- Controlling a pipeline in this manner works provided *the instruction at stage  $i+1$  can complete without any interference from instructions in stages 1 to  $i$*  (otherwise deadlocks may occur)



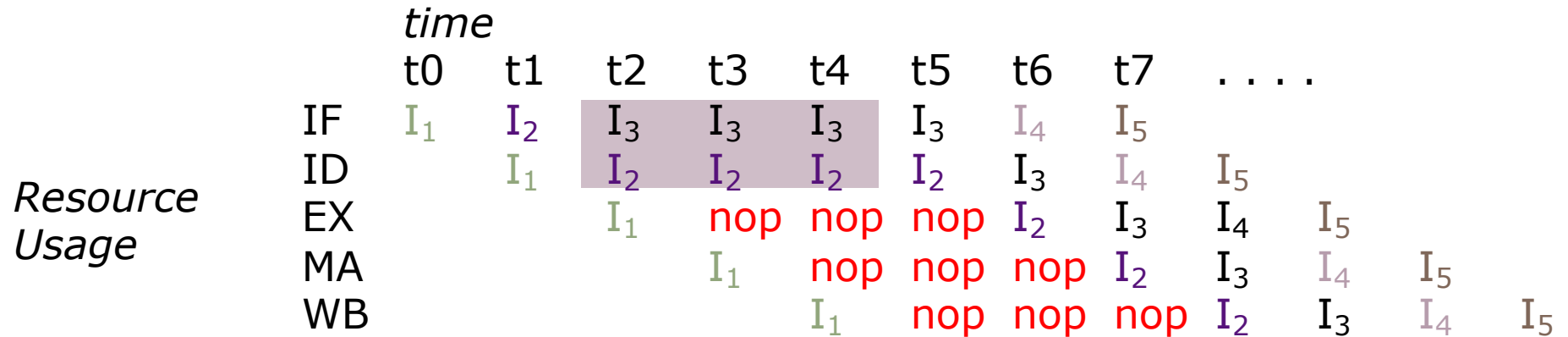
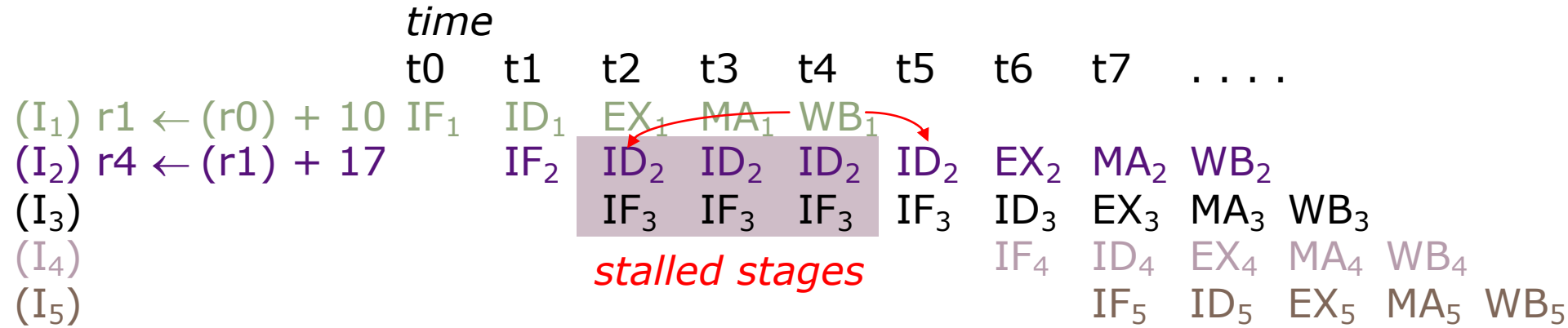
# Resolving Data Hazards by Stalling

*Stall Condition*



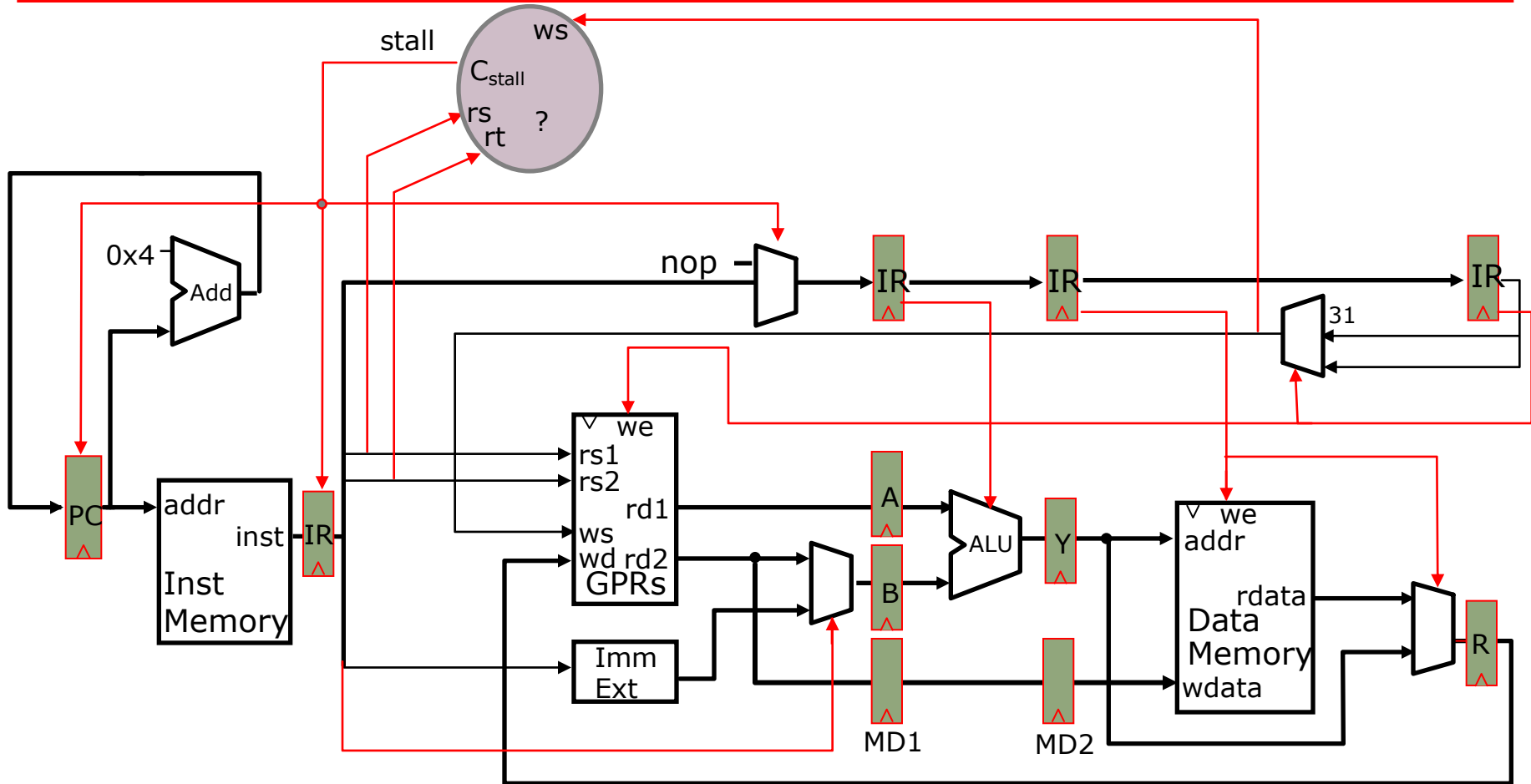
...  
 $r1 \leftarrow r0 + 10$   
 $r4 \leftarrow r1 + 17$   
...

# Stalled Stages and Pipeline Bubbles



*nop* ⇒ *pipeline bubble*

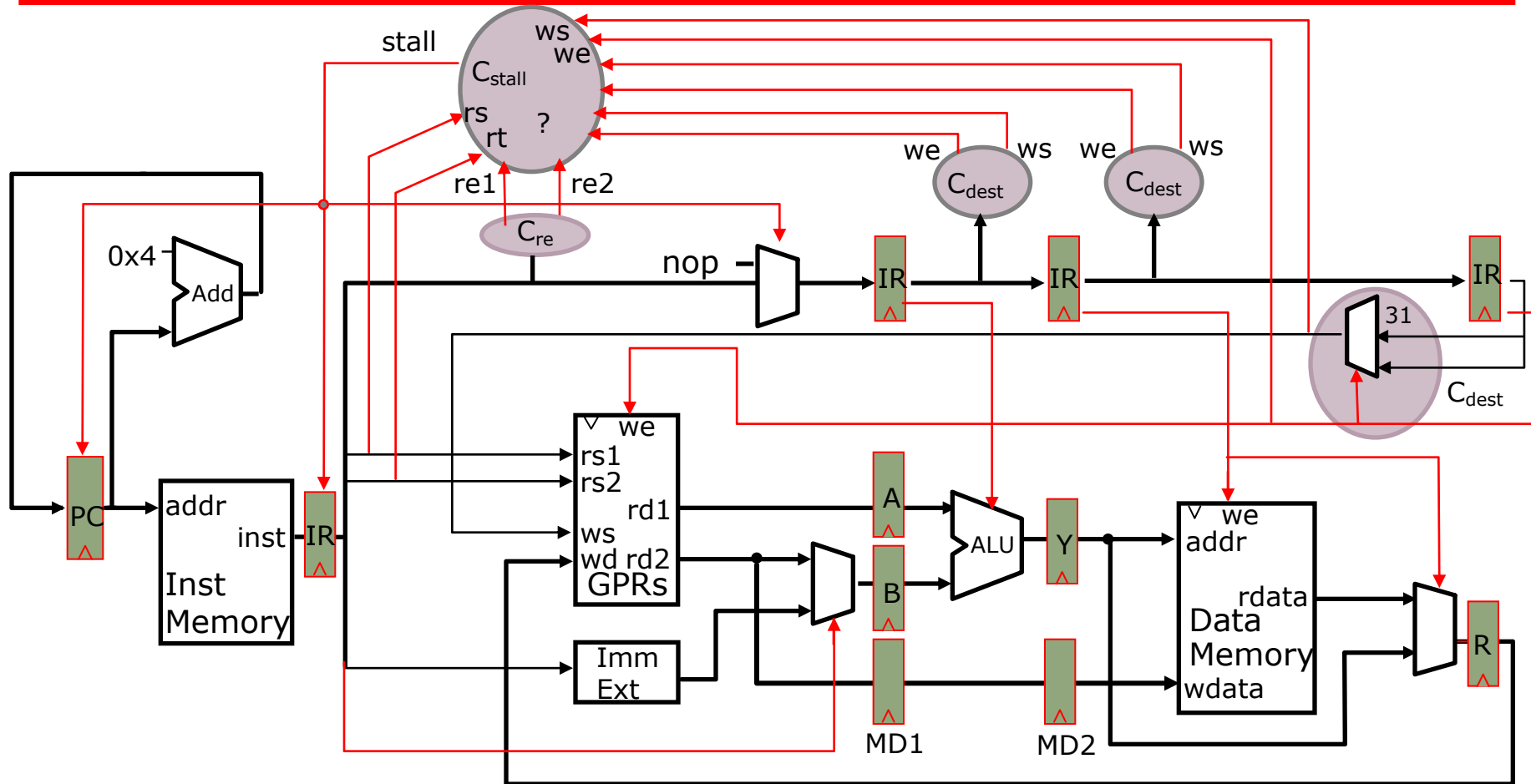
# Stall Control Logic



Compare the *source registers* of the instruction in the decode stage with the *destination register* of the *uncommitted instructions*.

# Stall Control Logic

*ignoring jumps & branches*



Should we always stall if the rs field matches some rd?  
 not every instruction writes a register  $\Rightarrow$  we  
 not every instruction reads a register  $\Rightarrow$  re

Thank you!