

# Instruction Pipelining: Hazard Resolution, Timing Constraints

*Daniel Sanchez*

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

# Resolving Data Hazards

---

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *stall*

Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage* → *bypass*

Strategy 3: *Speculate on the dependence*

*Two cases:*

*Guessed correctly* → no special action required

*Guessed incorrectly* → kill and restart

# Resolving Data Hazards (1)

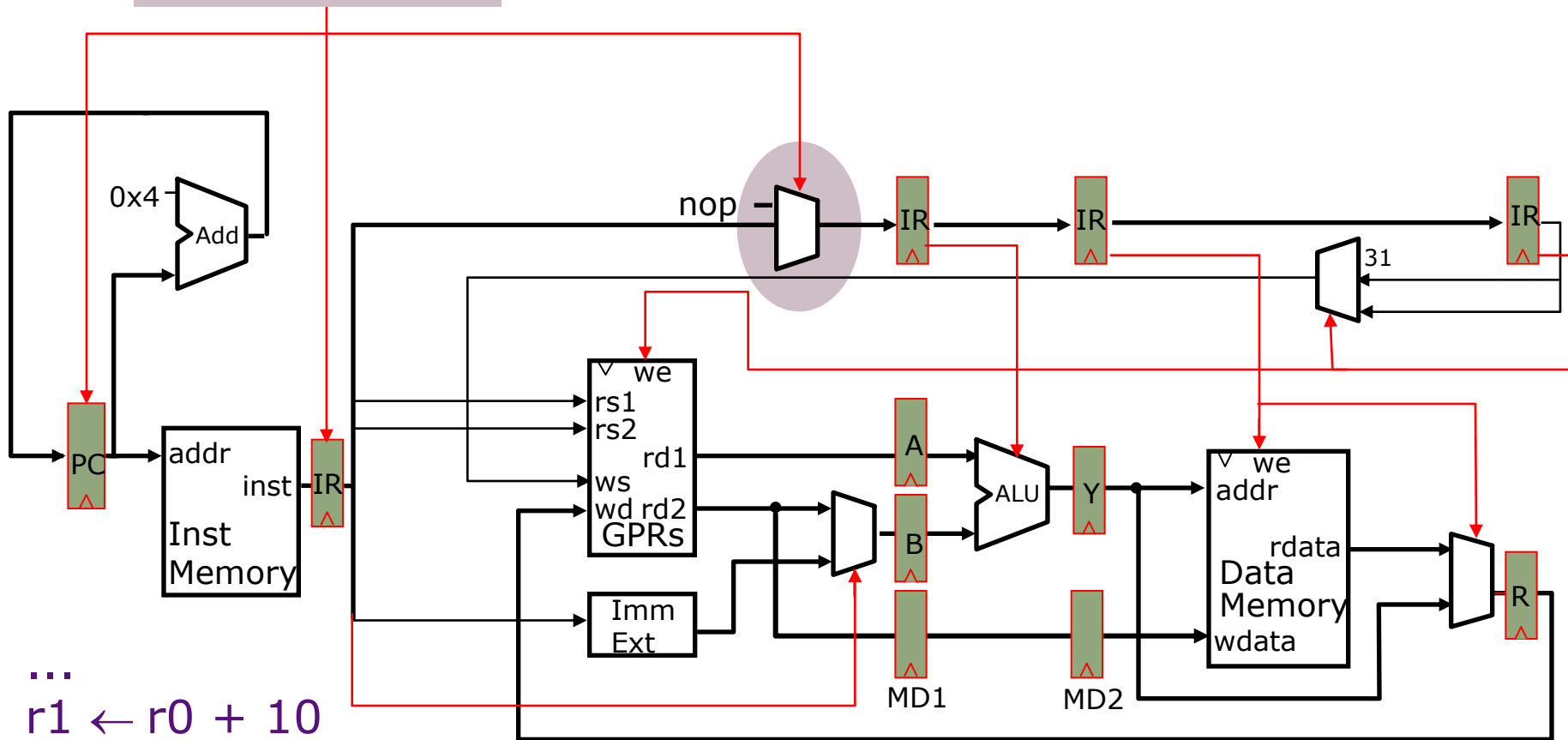
---

*Strategy 1:*

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

# Resolving Data Hazards by Stalling

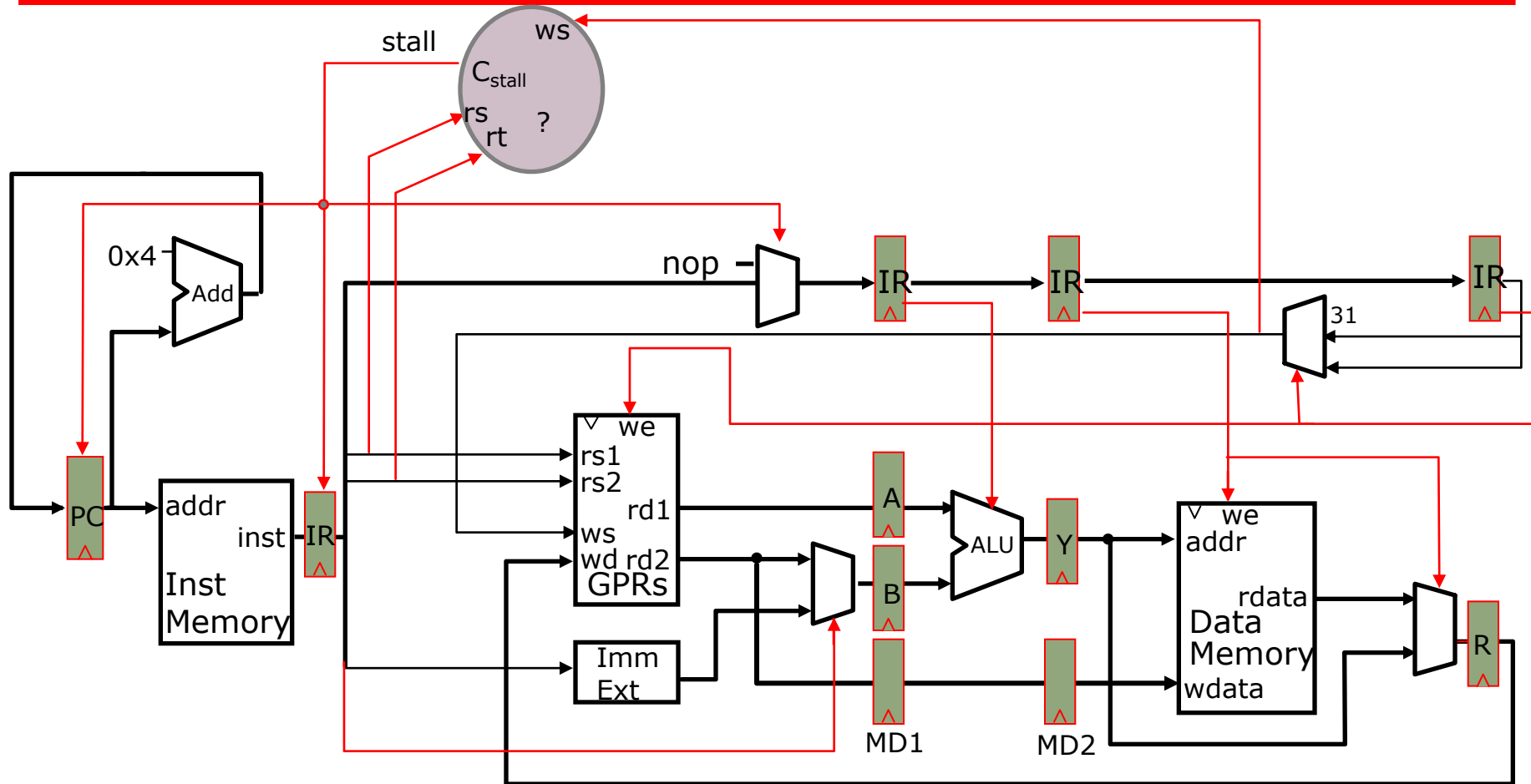
*Stall Condition*



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

How do we know when to stall?

# Stall Control Logic



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



# Source & Destination Registers

*R-type:*

op	rs	rt	rd		func
----	----	----	----	--	------

*I-type:*

op	rs	rt	immediate16
----	----	----	-------------

*J-type:*

op	immediate26
----	-------------

*source(s) destination*

ALU	rd $\leftarrow$ (rs) func (rt)	rs, rt	rd
ALUi	rt $\leftarrow$ (rs) op imm	rs	rt
LW	rt $\leftarrow$ M [(rs) + imm]	rs	rt
SW	M [(rs) + imm] $\leftarrow$ (rt)	rs, rt	
BZ	<i>cond</i> (rs)		
	<i>true:</i> PC $\leftarrow$ (PC) + imm	rs	
	<i>false:</i> PC $\leftarrow$ (PC) + 4	rs	
J	PC $\leftarrow$ (PC) + imm		
JAL	r31 $\leftarrow$ (PC), PC $\leftarrow$ (PC) + imm		31
JR	PC $\leftarrow$ (rs)	rs	
JALR	r31 $\leftarrow$ (PC), PC $\leftarrow$ (rs)	rs	31

# Deriving the Stall Signal

 $C_{\text{dest}}$ 

ws = Case opcode

ALU  $\Rightarrow$  rd  
 ALUi, LW  $\Rightarrow$  rt  
 JAL, JALR  $\Rightarrow$  R31

we = Case opcode

ALU, ALUi, LW  $\Rightarrow$  (ws  $\neq$  0)  
 JAL, JALR  $\Rightarrow$  on  
 ...  $\Rightarrow$  off

 $C_{\text{re}}$ 

re1 = Case opcode

ALU, ALUi,  
 LW, SW, BZ,  
 JR, JALR  $\Rightarrow$  on  
 J, JAL  $\Rightarrow$  off

re2 = Case opcode

ALU, SW  $\Rightarrow$  on  
 ...  $\Rightarrow$  off

 $C_{\text{stall}}$ 

$$\begin{aligned} \text{stall} = & ((rs_D = ws_E) \cdot we_E + \\ & (rs_D = ws_M) \cdot we_M + \\ & (rs_D = ws_W) \cdot we_W) \cdot re1_D + \\ & ((rt_D = ws_E) \cdot we_E + \\ & (rt_D = ws_M) \cdot we_M + \\ & (rt_D = ws_W) \cdot we_W) \cdot re2_D \end{aligned}$$

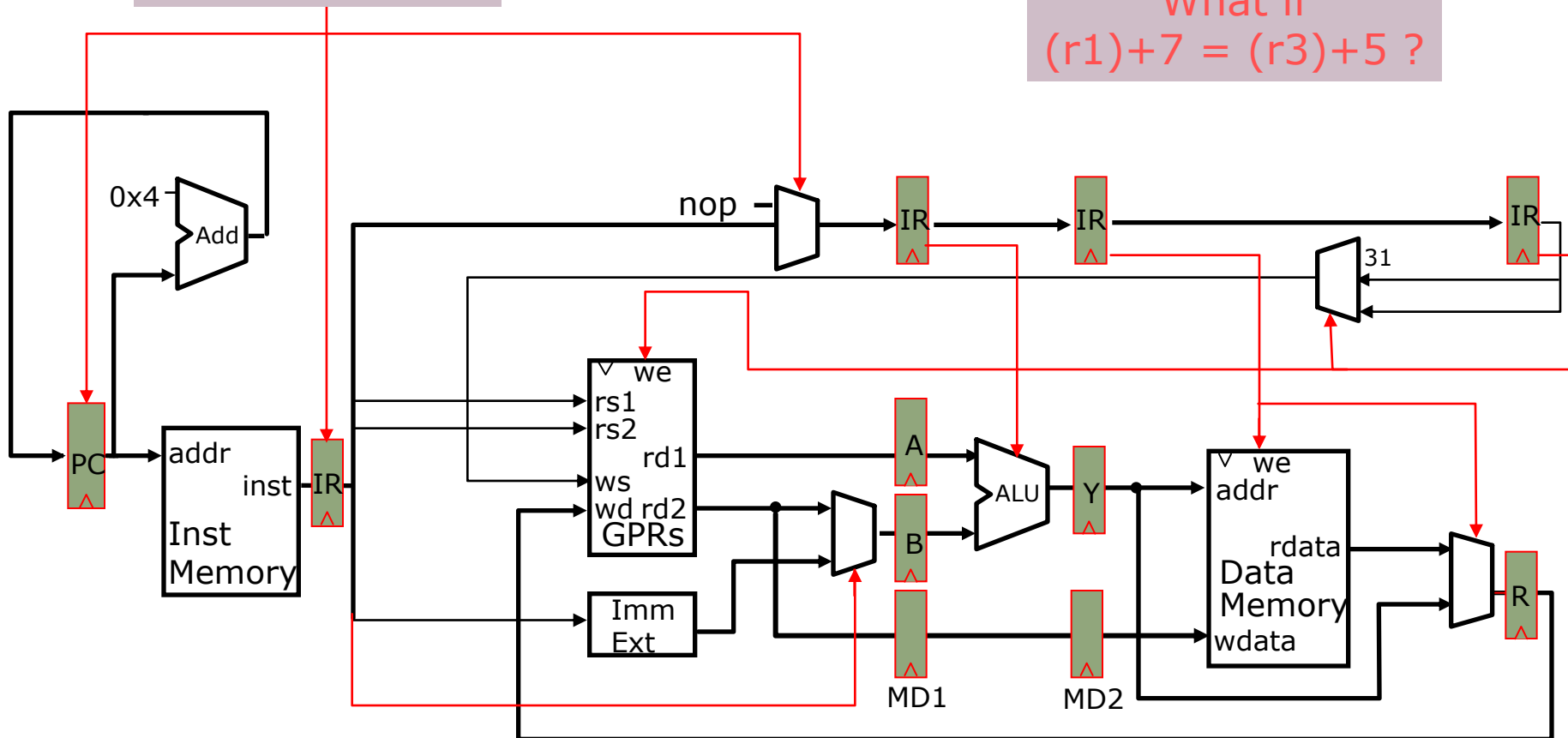
*This is not  
the full story!*



# Hazards due to Loads & Stores

*Stall Condition*

What if  
 $(r1)+7 = (r3)+5$  ?



...  
 $M[(r1)+7] \leftarrow (r2)$   
 $r4 \leftarrow M[(r3)+5]$

*Is there any possible data hazard  
in this instruction sequence?*

# Load & Store Hazards

---

```
...  
M[(r1)+7] ← (r2)  
r4 ← M[(r3)+5]  
...
```

$(r1)+7 = (r3)+5 \Rightarrow$  *data hazard*

However, the hazard is avoided because *our memory system completes writes in one cycle!*

Load/Store hazards are sometimes resolved in the pipeline and sometimes in the memory system itself.

*More on this later in the course.*

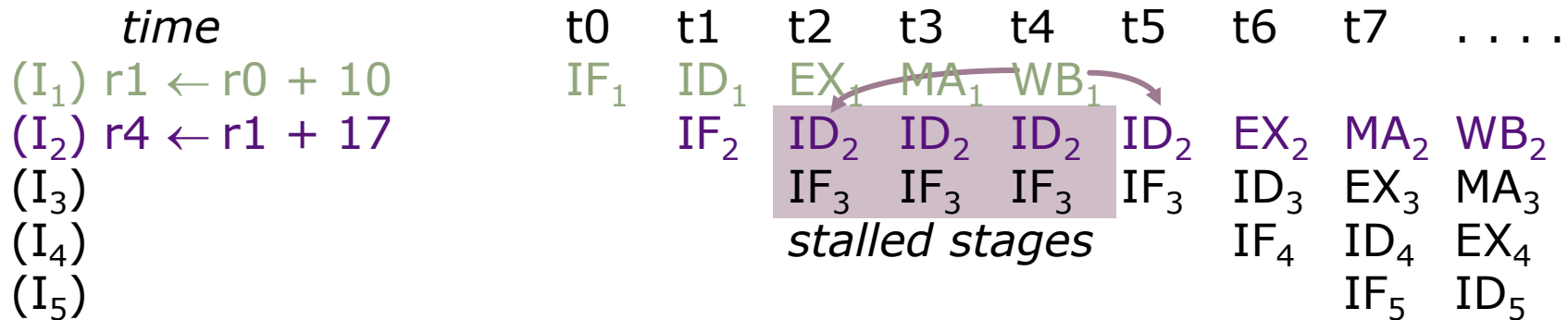
# Resolving Data Hazards (2)

---

## Strategy 2:

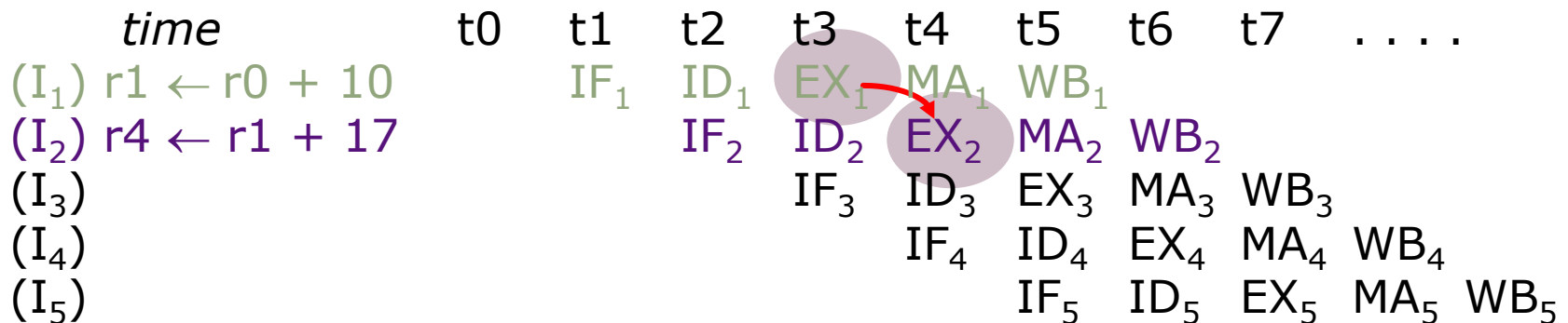
Route data as soon as possible after it is calculated to the earlier pipeline stage → *bypass*

# Bypassing



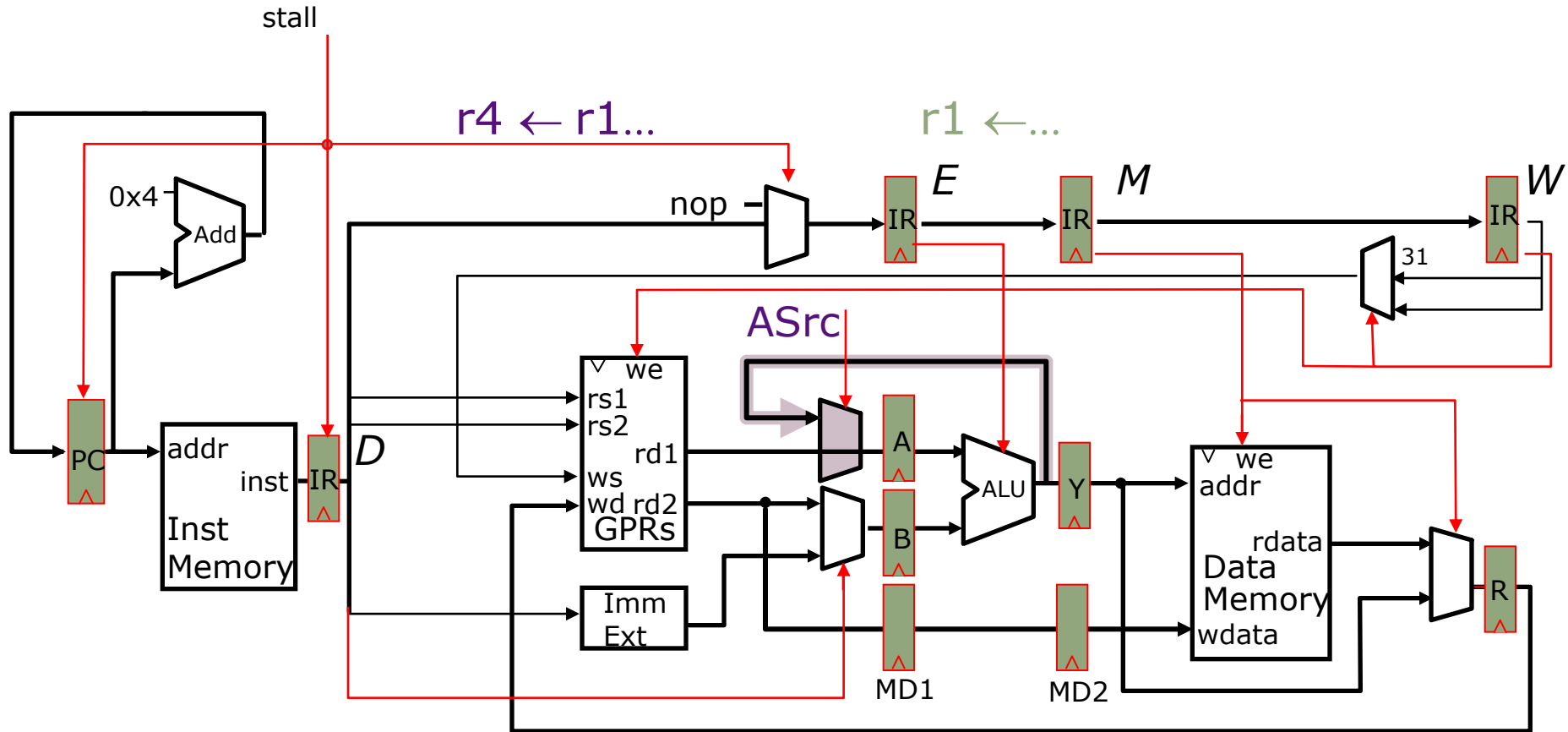
Each *stall or kill* introduces a bubble  $\Rightarrow CPI > 1$

When is data actually available? **At Execute**



A new datapath, i.e., a *bypass*, can get the data from the output of the ALU to its input

# Adding a Bypass



When does this bypass help?

...  
 $(I_1) \quad r1 \leftarrow r0 + 10$   
 $(I_2) \quad r4 \leftarrow r1 + 17$   
*yes*

$r1 \leftarrow M[r0 + 10]$   
 $r4 \leftarrow r1 + 17$   
*no*

JAL 500  
 $r4 \leftarrow r31 + 17$   
*no*

# The Bypass Signal

## *Deriving it from the Stall Signal*

$$\text{stall} = \cancel{((rs_D = ws_E) \cdot we_E)} + (rs_D = ws_M) \cdot we_M + (rs_D = ws_W) \cdot we_W \cdot re1_D \\ + ((rt_D = ws_E) \cdot we_E) + (rt_D = ws_M) \cdot we_M + (rt_D = ws_W) \cdot we_W \cdot re2_D$$

ws = Case opcode  
 ALU            ⇒ rd  
 ALUi, LW      ⇒ rt  
 JAL, JALR     ⇒ R31

we = Case opcode  
 ALU, ALUi, LW ⇒ (ws ≠ 0)  
 JAL, JALR     ⇒ on  
 ...            ⇒ off

$$\text{ASrc} = (rs_D = ws_E) \cdot we_E \cdot re1_D$$

Is this correct?

No because only ALU and ALUi instructions can benefit from this bypass

How might we address this?

Split  $we_E$  into two components: we-bypass, we-stall

# Bypass and Stall Signals

Split  $we_E$  into two components: we-bypass, we-stall

$we\_bypass_E =$	Case opcode <sub>E</sub>
ALU, ALUi	$\Rightarrow (ws \neq 0)$
...	$\Rightarrow$ off

$we\_stall_E =$	Case opcode <sub>E</sub>
LW	$\Rightarrow (ws \neq 0)$
JAL, JALR	$\Rightarrow$ on
...	$\Rightarrow$ off

ASrc	$= (rs_D = ws_E) \cdot we\_bypass_E \cdot re1_D$
------	--

stall	$= ((rs_D = ws_E) \cdot we\_stall_E +$
	$(rs_D = ws_M) \cdot we_M + (rs_D = ws_W) \cdot we_W) \cdot re1_D$
	$+ ((rt_D = ws_E) \cdot we_E + (rt_D = ws_M) \cdot we_M + (rt_D = ws_W) \cdot we_W) \cdot re2_D$





# Resolving Data Hazards (3)

---

*Strategy 3:*

*Speculate on the dependence. Two cases:*

*Guessed correctly* → no special action required

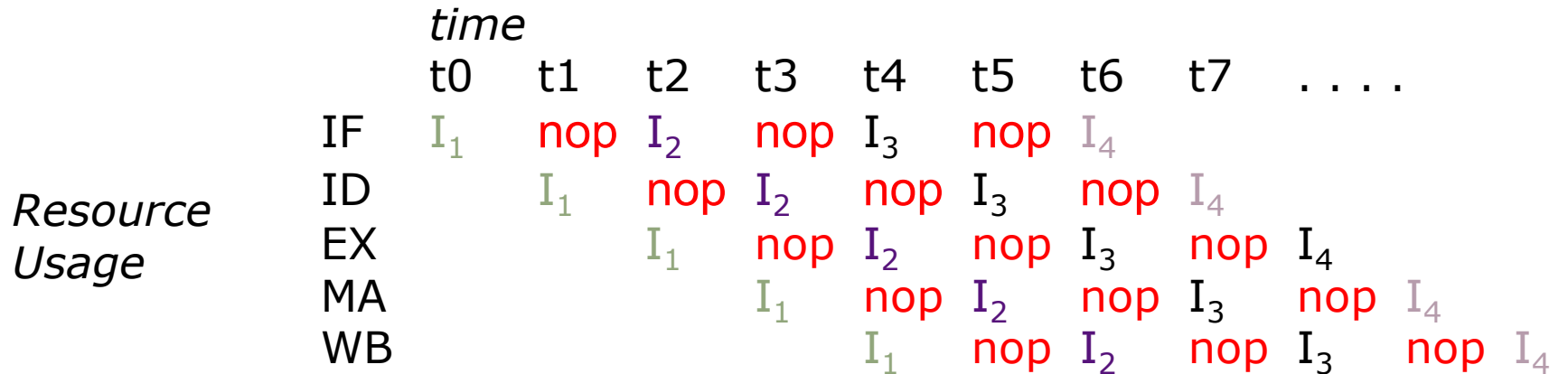
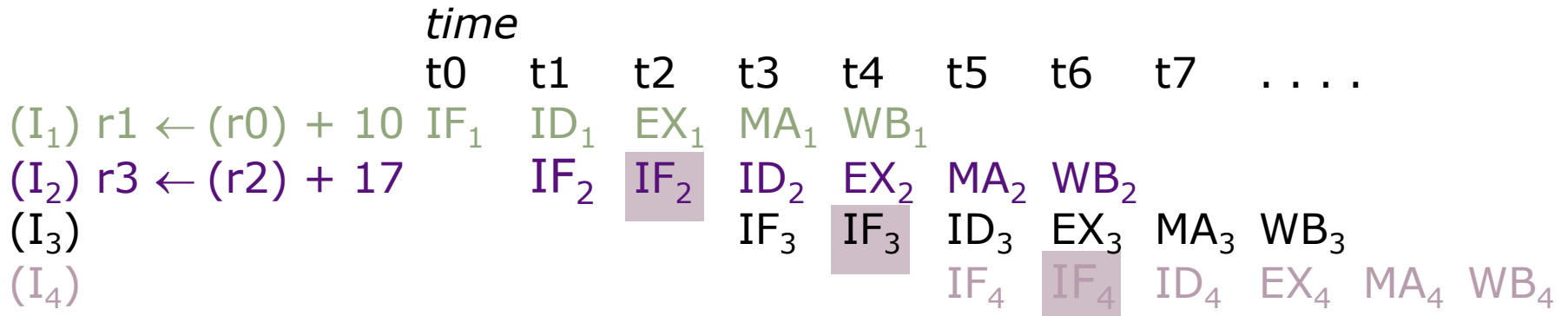
*Guessed incorrectly* → kill and restart

# Instruction to Instruction Dependence

---

- What do we need to calculate next PC?
  - For Jumps
    - Opcode, offset, and PC
  - For Jump Register
    - Opcode and register value
  - For Conditional Branches
    - Opcode, offset, PC, and register (for condition)
  - For all others
    - Opcode and PC
- In what stage do we know these?
  - PC → Fetch
  - Opcode, offset → Decode (or Fetch?)
  - Register value → Decode
  - Branch condition ((rs)==0) → Execute (or Decode?)

# NextPC Calculation Bubbles

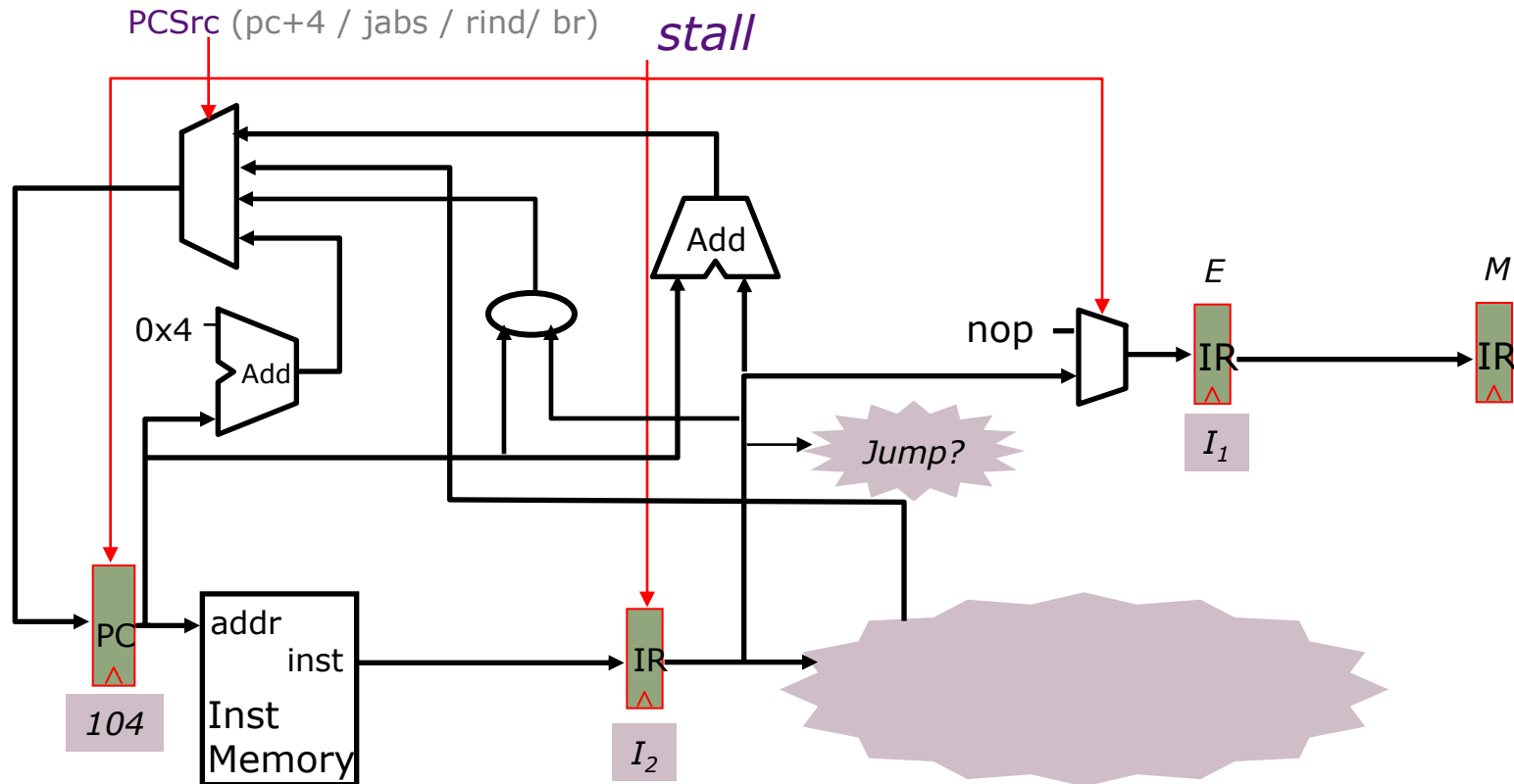


*nop* ⇒ *pipeline bubble*

What's a good guess for next PC?

PC+4

# Speculate NextPC is PC+4

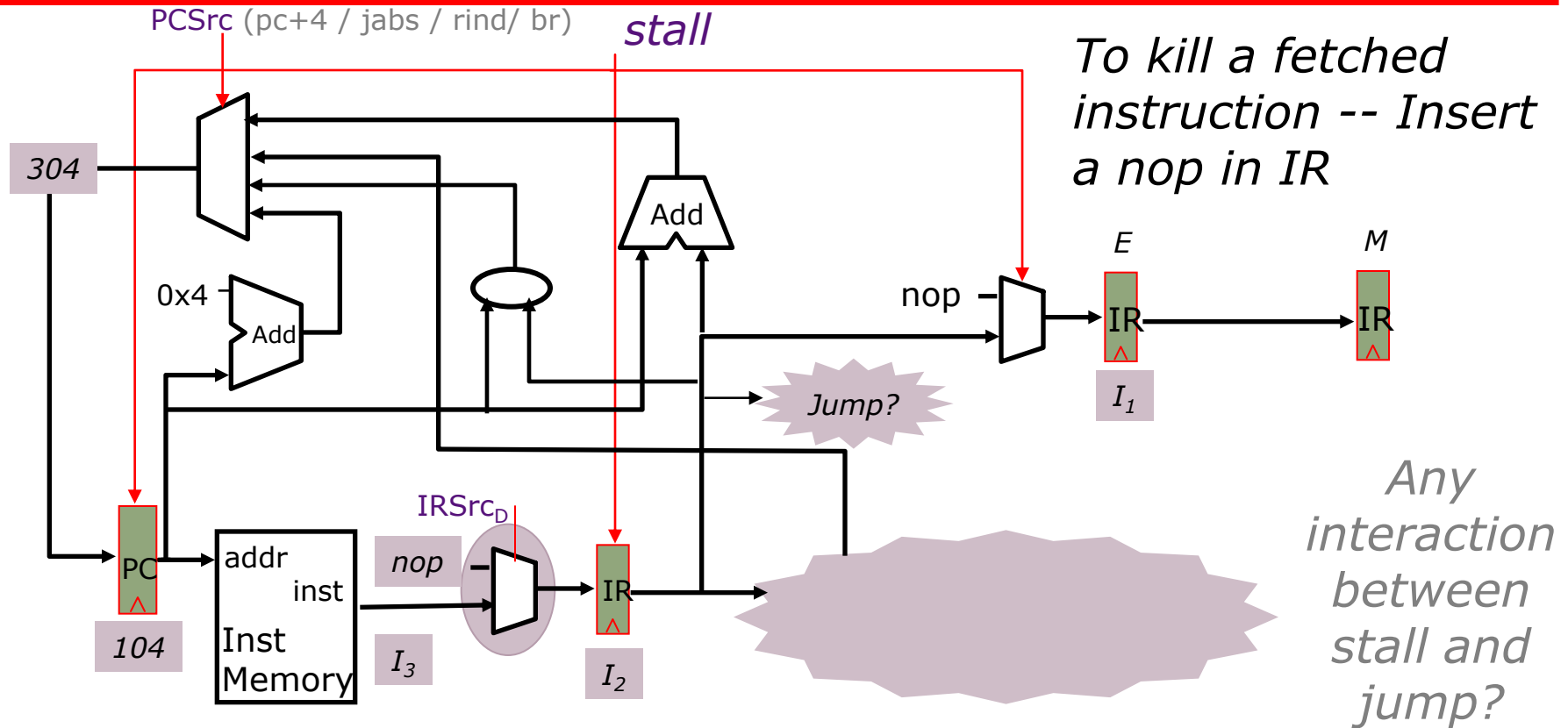


I <sub>1</sub>	096	ADD	
I <sub>2</sub>	100	J	200
I <sub>3</sub>	<del>104</del>	<del>ADD</del>	<i>kill</i>
I <sub>4</sub>	304	ADD	

What happens on mis-speculation, i.e., when next instruction is not PC+4?

*How?*

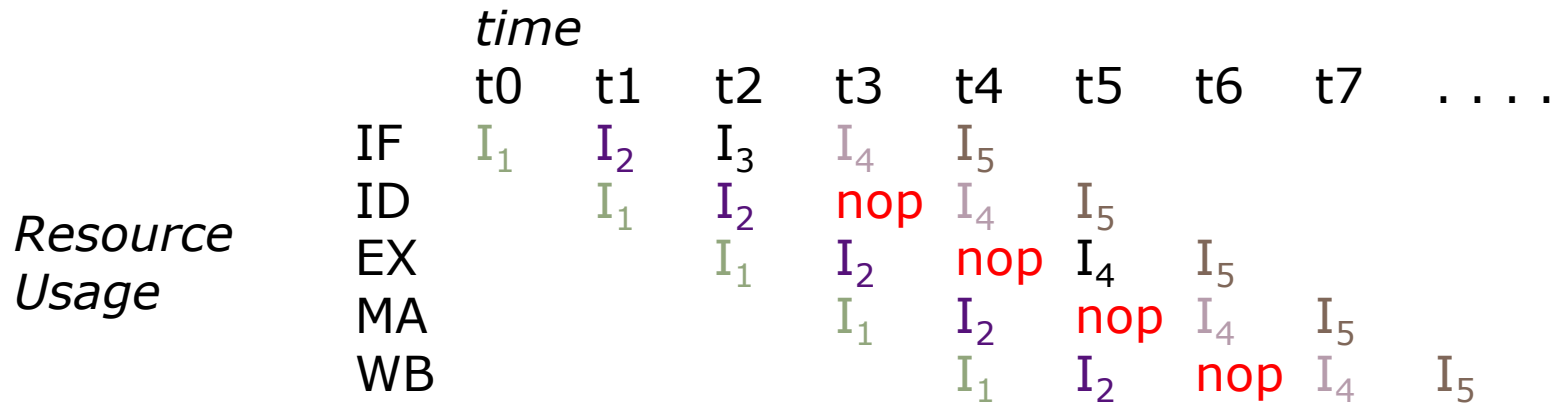
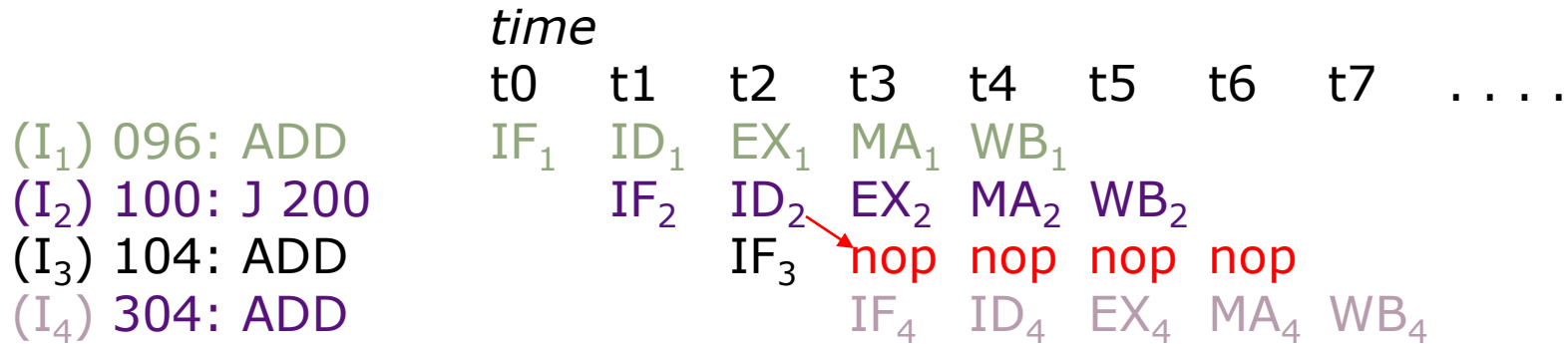
# Pipelining Jumps



$I_1$	096	ADD	
$I_2$	100	J	200
$I_3$	<del>104</del>	<del>ADD</del>	<i>kill</i>
$I_4$	304	ADD	

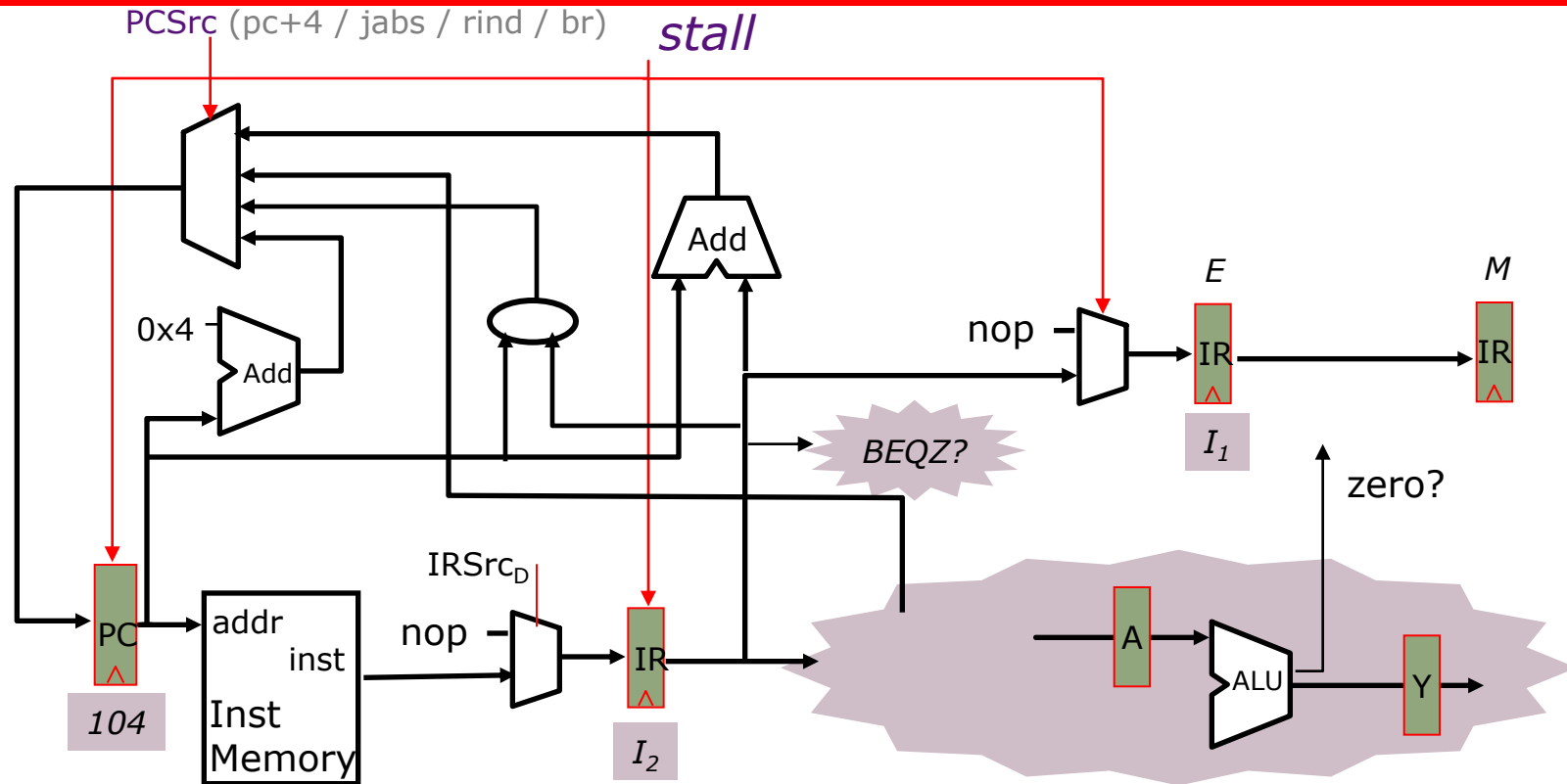
$IRSrc_D = \text{Case opcode}_D$   
 J, JAL  $\Rightarrow$  nop  
 ...  $\Rightarrow$  IM

# Jump Pipeline Diagrams



*nop* ⇒ *pipeline bubble*

# Pipelining Conditional Branches



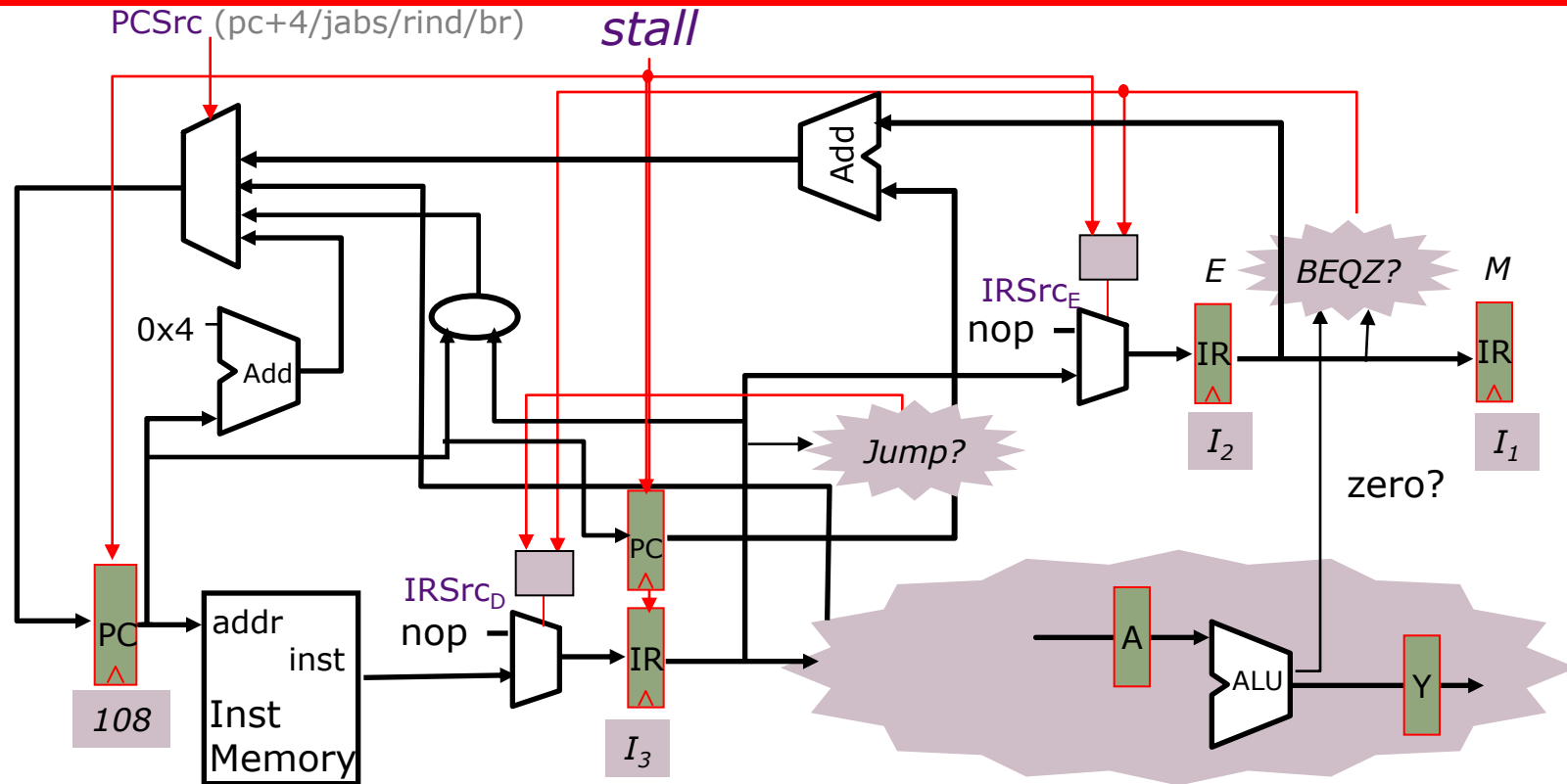
I <sub>1</sub>	096	ADD
I <sub>2</sub>	100	BEQZ r1 200
I <sub>3</sub>	104	ADD
I <sub>4</sub>	304	ADD

Branch condition is not known until the execute stage  
*what action should be taken in the decode stage?*





# Pipelining Conditional Branches



If the branch is taken

- kill the two following instructions
- the instruction at the decode stage is not valid

⇒ *stall signal is not valid*

I <sub>1</sub>	096	ADD
I <sub>2</sub>	100	BEQZ r1 200
I <sub>3</sub>	104	ADD
I <sub>4</sub>	304	ADD

# New Stall Signal

---

$$\text{stall} = ( ((rs_D = ws_E) \cdot we_E + (rs_D = ws_M) \cdot we_M + (rs_D = ws_W) \cdot we_W) \cdot re1_D \\ + ((rt_D = ws_E) \cdot we_E + (rt_D = ws_M) \cdot we_M + (rt_D = ws_W) \cdot we_W) \cdot re2_D \\ ) \cdot !((opcode_E = BEQZ) \cdot z + (opcode_E = BNEZ) \cdot !z)$$

Don't stall if the branch is taken. Why?

Instruction at the decode stage is invalid

# Control Equations for PC and IR Muxes

$$\begin{aligned} \text{IRSrc}_D &= \text{Case opcode}_E \\ \text{BEQZ}\cdot z, \text{BNEZ}\cdot !z &\Rightarrow \text{nop} \\ \dots &\Rightarrow \\ &\text{Case opcode}_D \\ \text{J, JAL, JR, JALR} &\Rightarrow \text{nop} \\ \dots &\Rightarrow \text{IM} \end{aligned}$$

*Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction*

$$\begin{aligned} \text{IRSrc}_E &= \text{Case opcode}_E \\ \text{BEQZ}\cdot z, \text{BNEZ}\cdot !z &\Rightarrow \text{nop} \\ \dots &\Rightarrow \text{stall}\cdot \text{nop} + !\text{stall}\cdot \text{IR}_D \end{aligned}$$

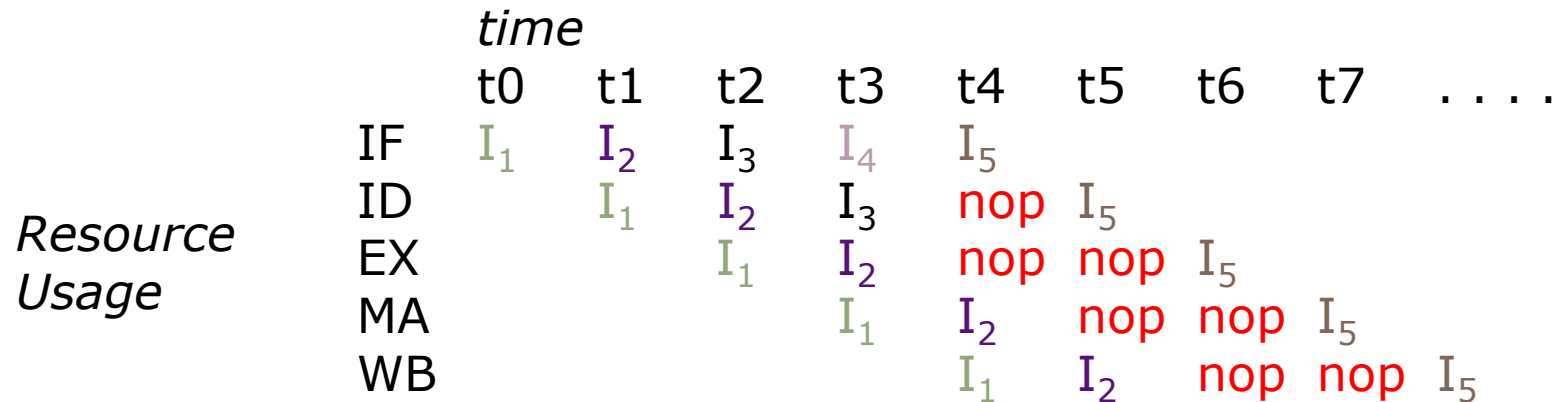
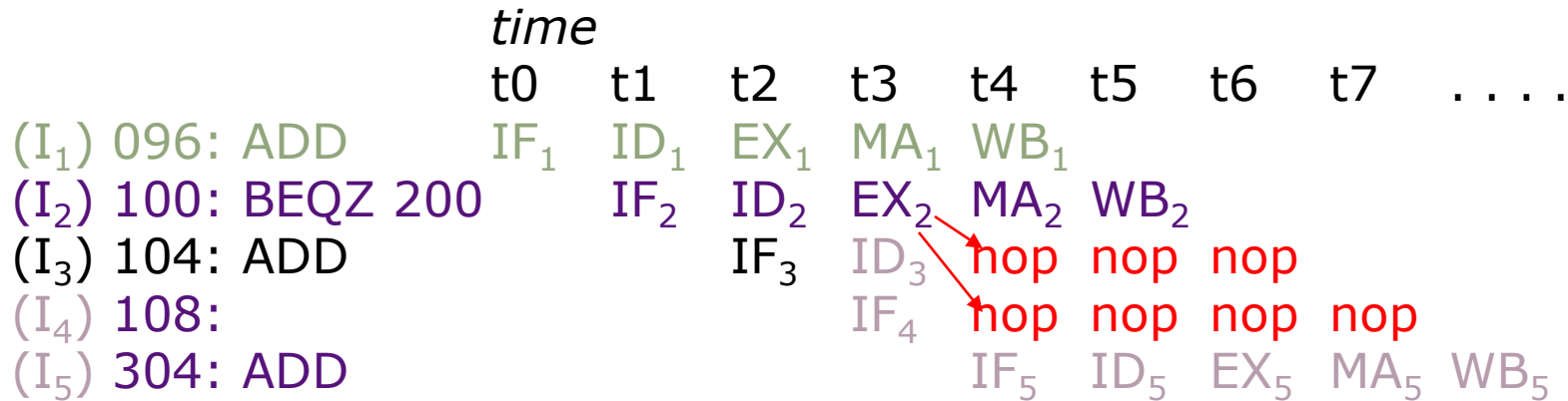
$$\begin{aligned} \text{PCSrc} &= \text{Case opcode}_E \\ \text{BEQZ}\cdot z, \text{BNEZ}\cdot !z &\Rightarrow \text{br} \\ \dots &\Rightarrow \\ &\text{Case opcode}_D \\ \text{J, JAL} &\Rightarrow \text{jabs} \\ \text{JR, JALR} &\Rightarrow \text{rind} \\ \dots &\Rightarrow \text{pc}+4 \end{aligned}$$

*pc+4 is a speculative guess*

$\text{nop} \Rightarrow \text{Kill}$   
 $\text{br/jabs/rind} \Rightarrow \text{Restart}$   
 $\text{pc}+4 \Rightarrow \text{Speculate}$

# Branch Pipeline Diagrams

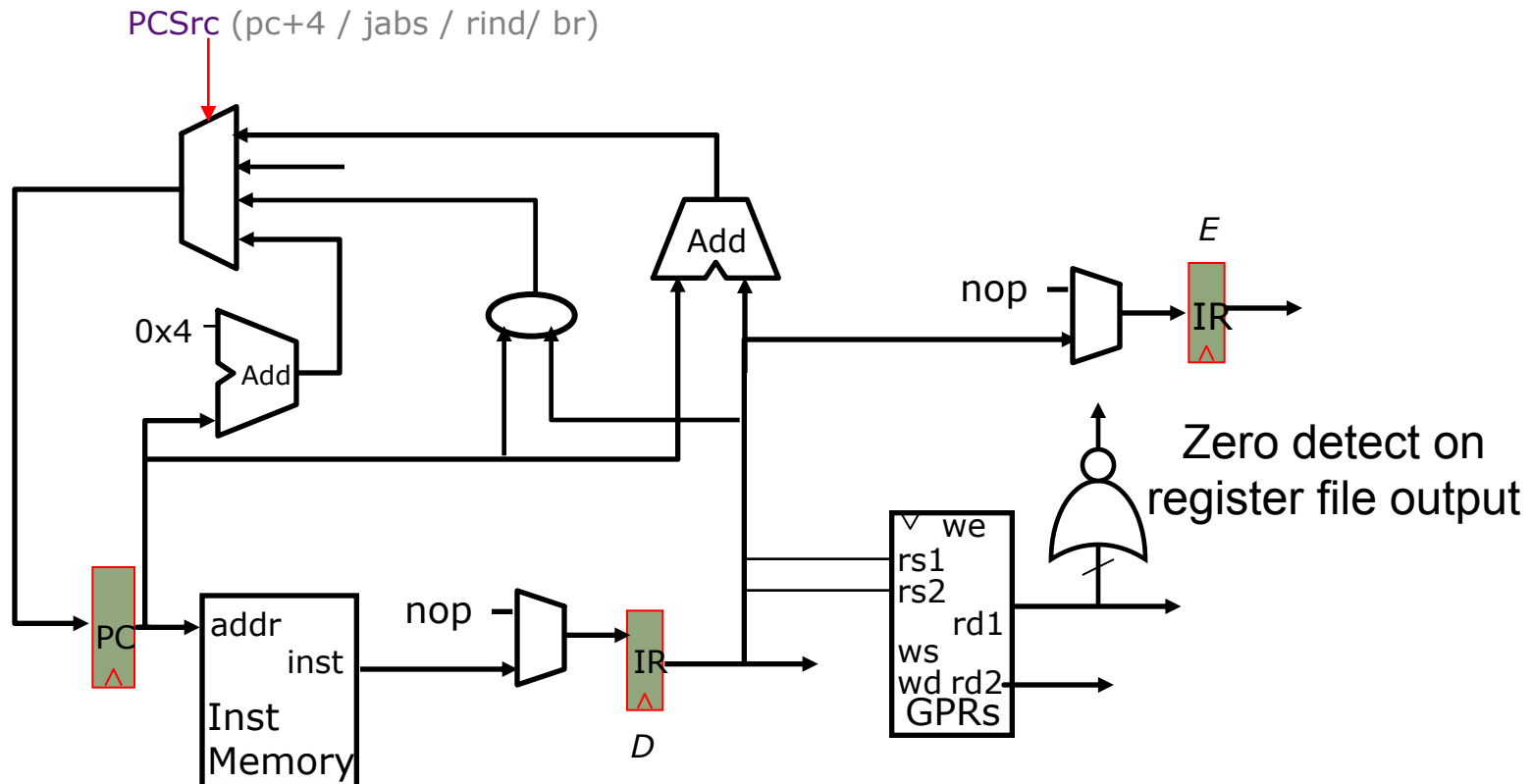
## (resolved in execute stage)



*nop* ⇒ *pipeline bubble*

# Reducing Branch Penalty (resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage



*Pipeline diagram now same as for jumps*

# Branch Delay Slots (expose control hazard to software)

---

- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

I <sub>1</sub>	096	ADD	
I <sub>2</sub>	100	BEQZ r1 200	<i>Delay slot instruction</i>
I <sub>3</sub>	104	ADD	← <i>executed regardless of branch outcome</i>
I <sub>4</sub>	304	ADD	

- Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*

# Why an Instruction may not be dispatched every cycle (CPI>1)

---

- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two cycle latency
  - Instruction after load cannot use load result
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- Conditional branches may cause bubbles
  - kill following instruction(s) if no delay slots

*Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.*