

Instruction Pipelining: Hazard Resolution, Timing Constraints

Mengjia Yan

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

Pipeline Diagram – Ideal Pipelining

<i>time</i>	t0	t1	t2	t3	t4	t5	t6	t7	...
(I ₁) a1 ← a0 + 10	IF ₁	ID ₁	EX ₁	MA ₁	WB ₁				
(I ₂) a3 ← a2 + 12		IF ₂	ID ₂	EX ₂	MA ₂	WB ₂			
(I ₃) a5 ← a4 + 14			IF ₃	ID ₃	EX ₃	MA ₃	WB ₃		
(I ₄) a7 ← a6 + 16				IF ₄	ID ₄	EX ₄	MA ₄	WB ₄	
(I ₅) s1 ← s0 + 18					IF ₅	ID ₅	EX ₅	MA ₅	WB ₅

Over long term, i.e., in steady state, what are the ...

Throughput (T)

Latency (L)

CPI?

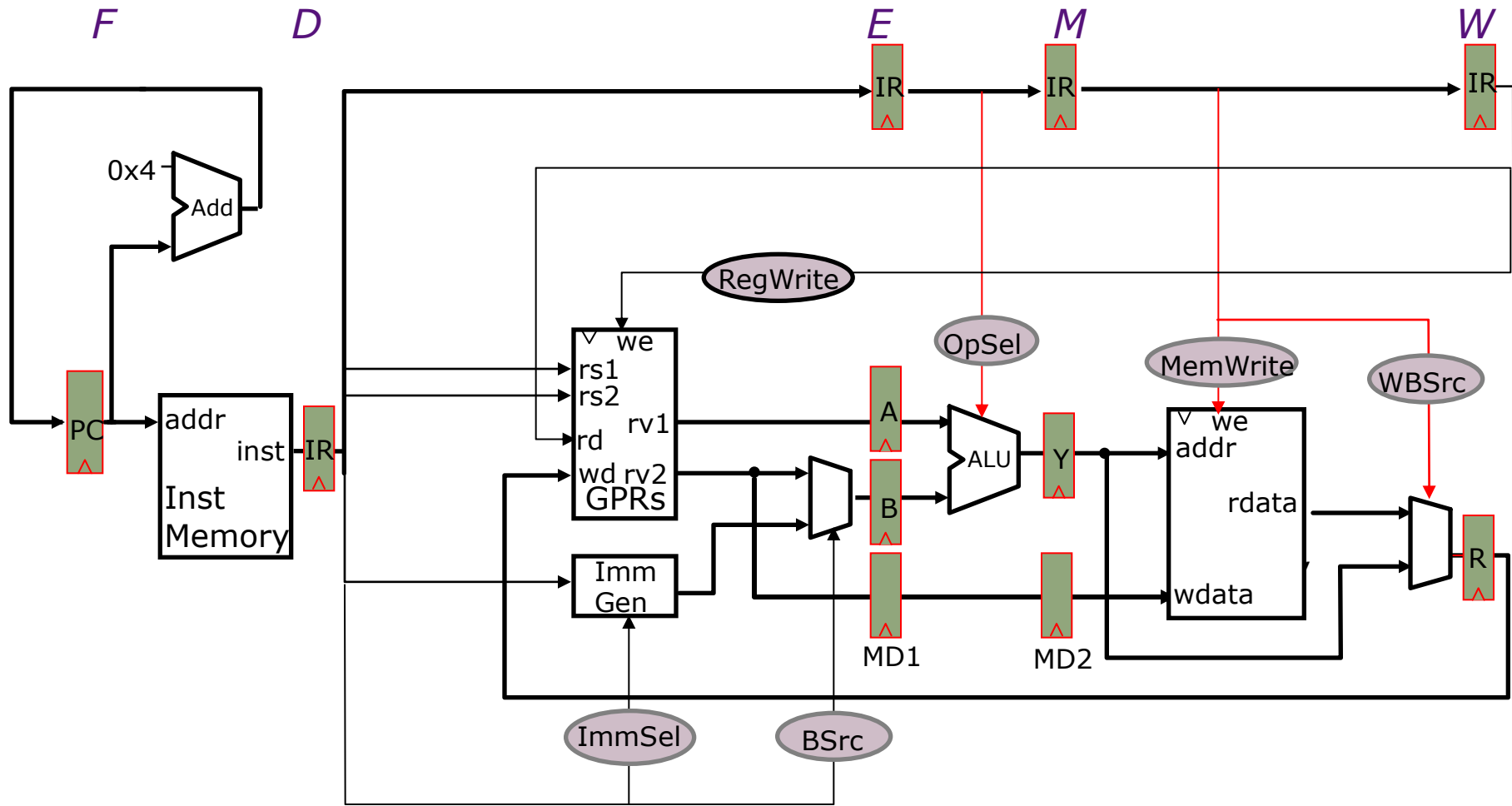
IPC?

Inst exec time?

N

Num inst in flight?

Reminder: Pipelined RISC-V Datapath *without jumps*



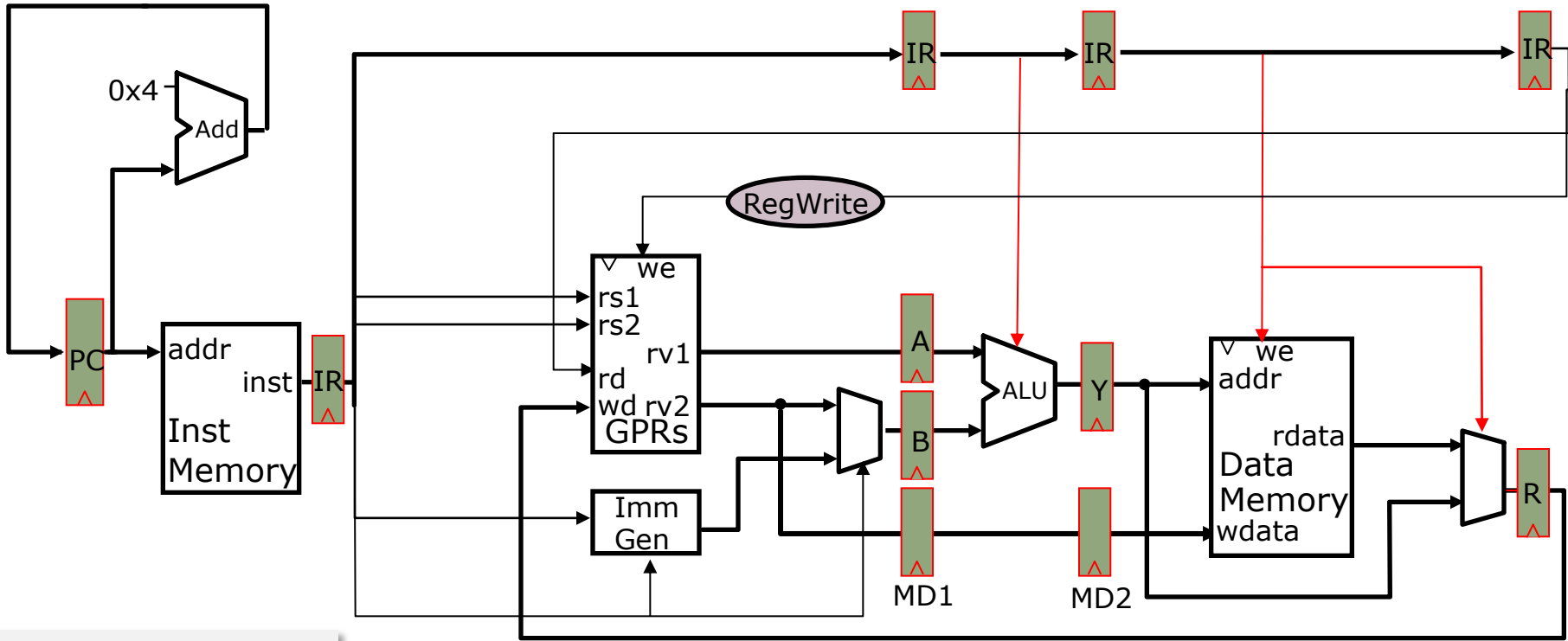
Pipelining increases clock frequency,
but instruction dependences may increase CPI

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

$a3 \leftarrow t1 \dots$



Cycle N+1

a1 is stale. Oops!

...
 $a1 \leftarrow a0 + 10$
 $a3 \leftarrow a1 + 12$
 ...

Pipeline Diagram – Hazard

<i>time</i>	t0	t1	t2	t3	t4	t5	t6	t7	...
(I ₁) a1 ← a0 + 10	IF ₁	ID ₁	EX ₁	MA ₁	WB ₁				
(I ₂) a3 ← a1 + 12		IF ₂	ID ₂	EX ₂	MA ₂	WB ₂			

a1 is stale. Oops!



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

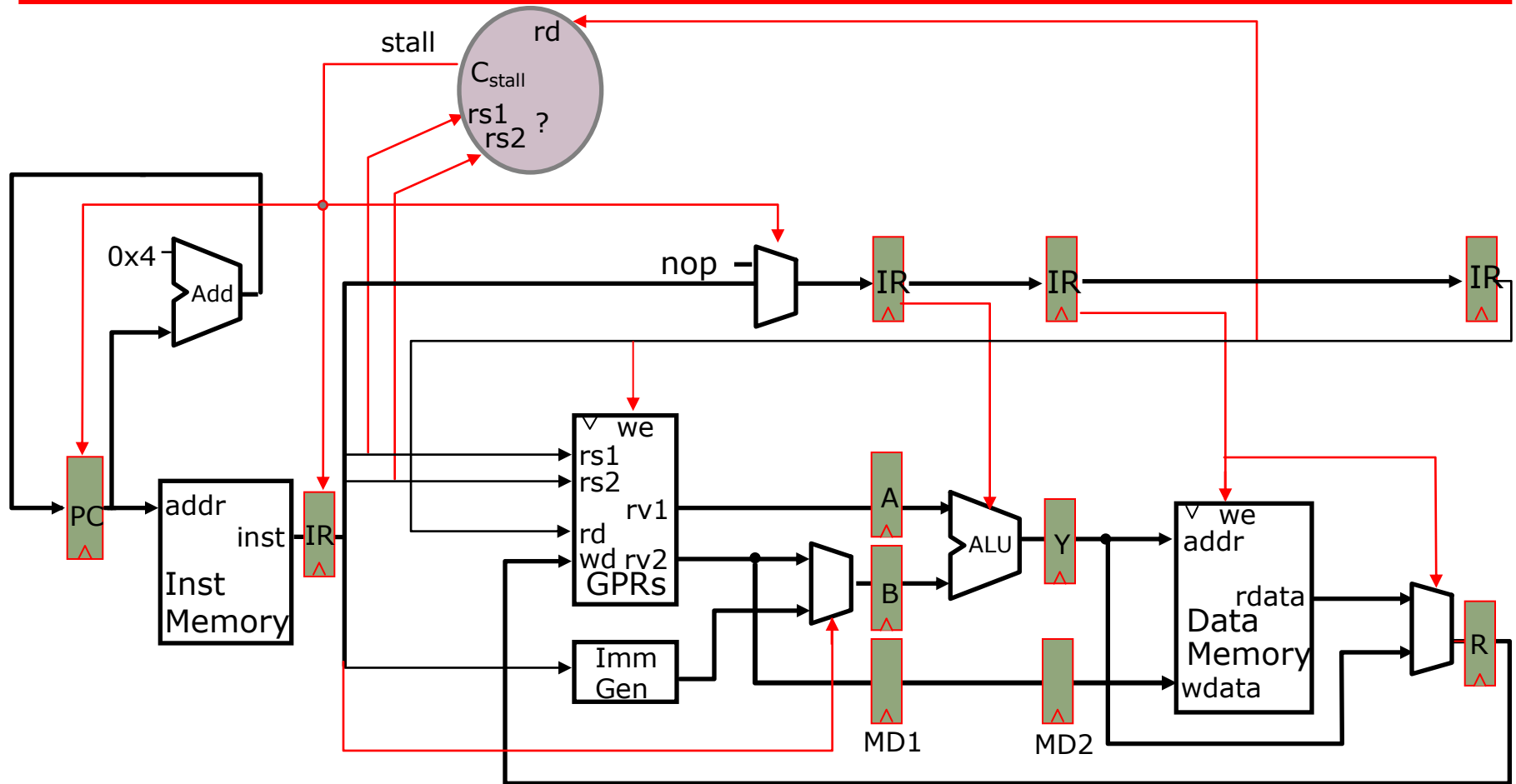
Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages → stall*

<i>time</i>	t0	t1	t2	t3	t4	t5	t6	t7	...
(I ₁) a1 ← a0 + 10	IF ₁	ID ₁	EX ₁	MA ₁	WB ₁				
(I ₂) a3 ← a1 + 12		IF ₂	ID ₂	ID ₂	ID ₂	ID ₂	EX ₂	MA ₂	WB ₂

Stall DEC & IF when instruction in DEC reads a register that is written by any earlier in-flight instruction (in EXE, MEM, or WB)

Reminder: Stall Control Logic

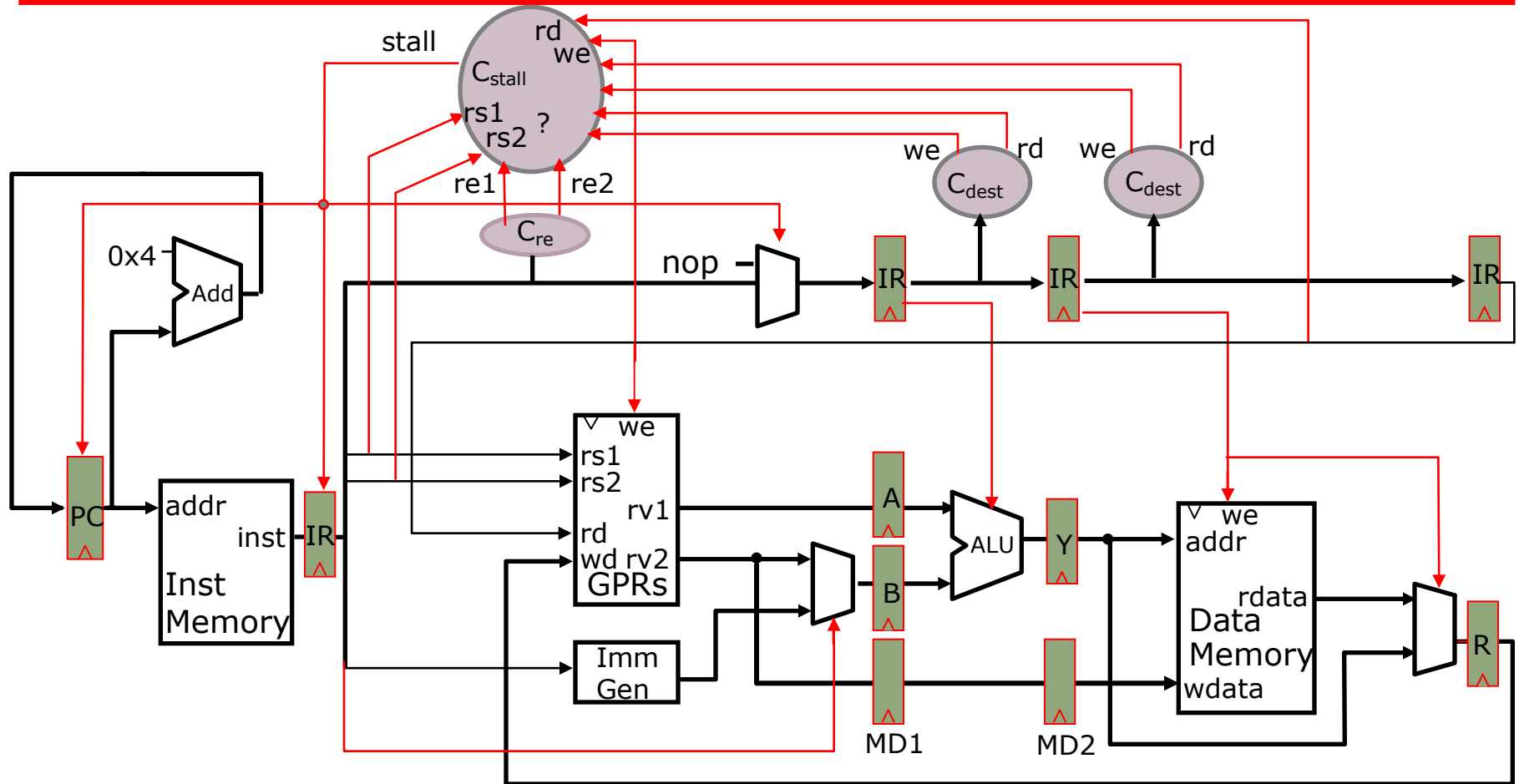
ignoring jumps & branches



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

Reminder: Stall Control Logic

ignoring jumps & branches



Should we always stall if the rs field(s) matches some rd?

Source & Destination Registers

R-type:

funct7	rs2	rs1	funct3	rd	opcode
--------	-----	-----	--------	----	--------

I-type:

imm[11:0]	rs1	funct3	rd	opcode
-----------	-----	--------	----	--------

imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
-----------	-----	-----	-------	----------	--------

		<i>source(s)</i>	<i>destination</i>
ALU	$rd \leftarrow (rs1) \text{ func } (rs2)$	rs1, rs2	rd
ALUi	$rd \leftarrow (rs1) \text{ funct SXT}(imm)$	rs1	rd
LW	$rd \leftarrow M [(rs1) + imm.disp]$	rs1	rd
SW	$M [(rs1) + imm.disp] \leftarrow (rs2)$	rs1, rs2	
BEQ	$cond (rs1) == (rs2)$		
	<i>true:</i> $PC \leftarrow (PC) + SXT(imm)$	rs1, rs2	
	<i>false:</i> $PC \leftarrow (PC) + 4$	rs1, rs2	
JAL	$rd \leftarrow (PC+4), PC \leftarrow (PC) + imm$		rd
JALR	$rd \leftarrow (PC+4), PC \leftarrow (rs1) + imm^*$	rs1	rd

*More precise: $pc \leftarrow \{(reg[rs1] + imm)[31:1], 1'b0\}$

Deriving the Stall Signal

C_{dest}

$we = \text{Case opcode}$
ALU, ALUi, LW, JAL, JALR
 $\Rightarrow (rd \neq 0)$
...
 $\Rightarrow \text{off}$

C_{re}

$re1 = \text{Case opcode}$
ALU, ALUi,
LW, SW, BEQ,
JR, JALR $\Rightarrow \text{on}$

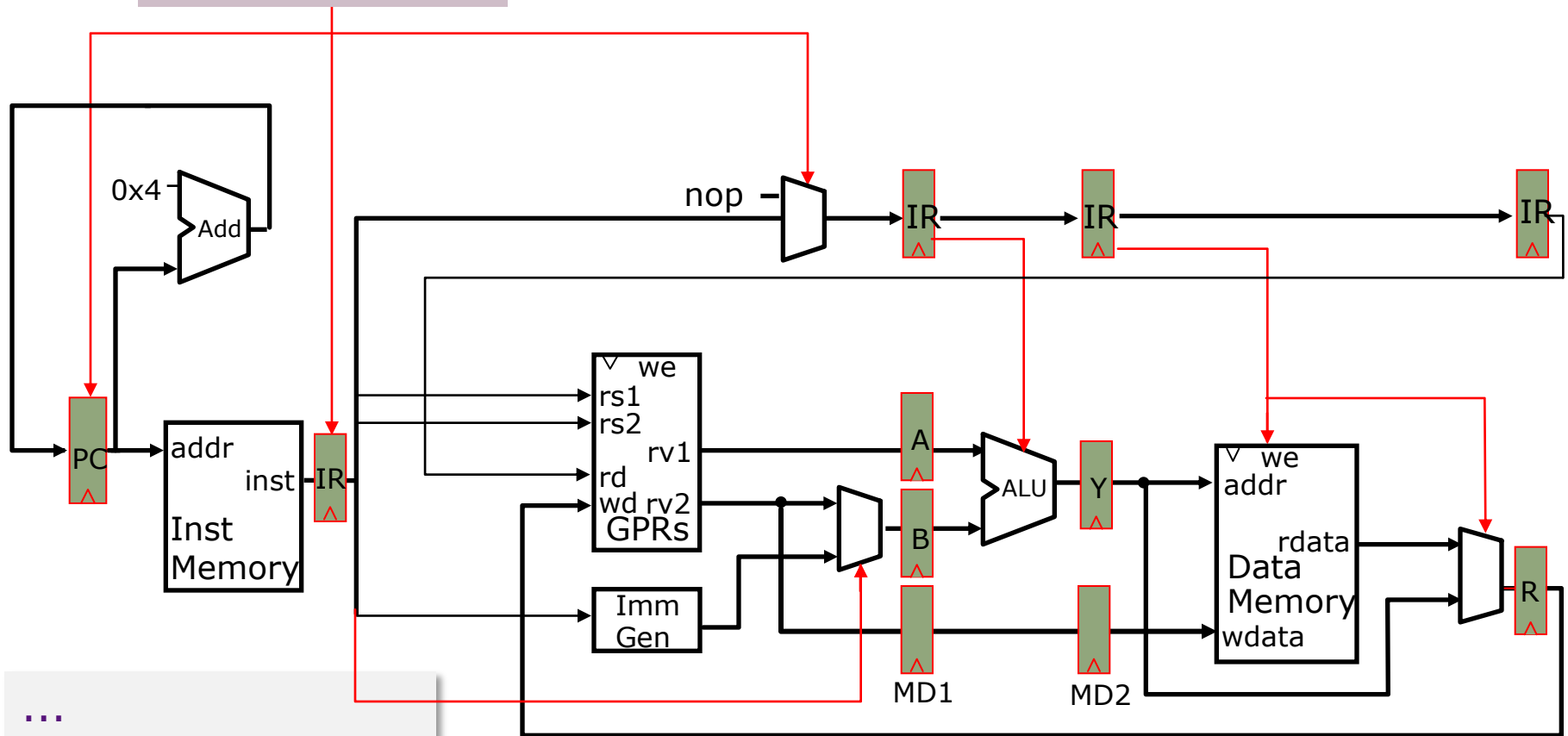
$re2 = \text{Case opcode}$
ALU, SW, BEQ $\Rightarrow \text{on}$
... $\Rightarrow \text{off}$

$$C_{\text{stall}} \quad \text{stall} = ((rs1_D == rd_E) \cdot we_E + (rs1_D == rd_M) \cdot we_M + (rs1_D == rd_W) \cdot we_W) \cdot re1_D + ((rs2_D == rd_E) \cdot we_E + (rs2_D == rd_M) \cdot we_M + (rs2_D == rd_W) \cdot we_W) \cdot re2_D$$

This is not the full story !

Hazards due to Loads & Stores

Stall Condition



...
 $M[(a1)+7] \leftarrow (a2)$
 $a4 \leftarrow M[(a3)+5]$
 ...

Is there any possible data hazard in this instruction sequence?

Load & Store Hazards

```
...  
M[(a1)+7] ← (a2)  
t4 ← M[(a3)+5]  
...
```

$(a1)+7 = (a3)+5 \Rightarrow \textit{data hazard}$

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.

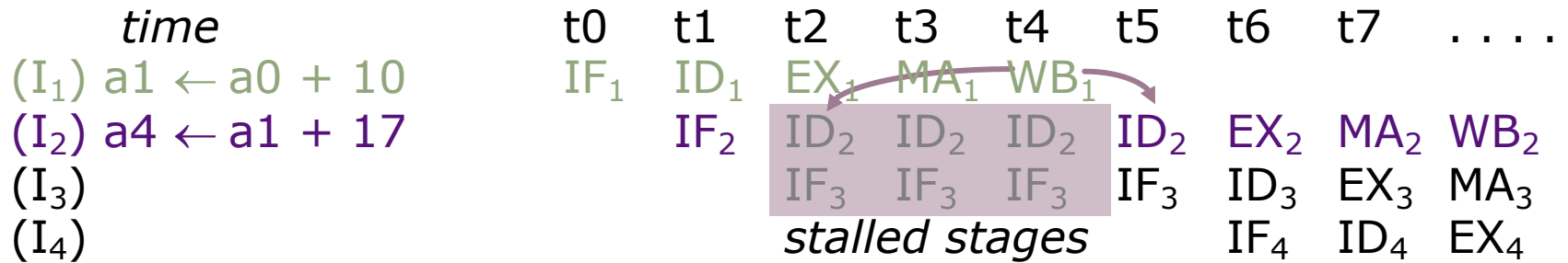
More on this later in the course.

Resolving Data Hazards (2)

Strategy 2:

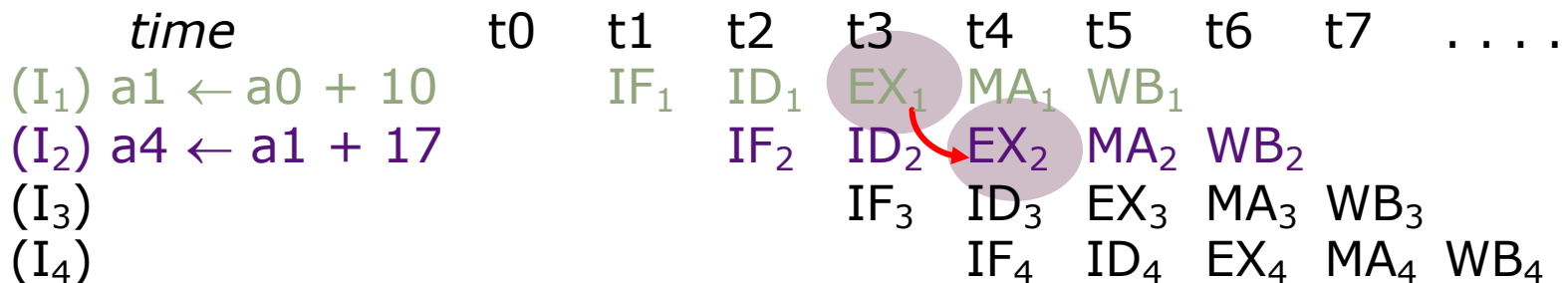
Route data as soon as possible after it is calculated to the earlier pipeline stage → *bypass*

Bypassing



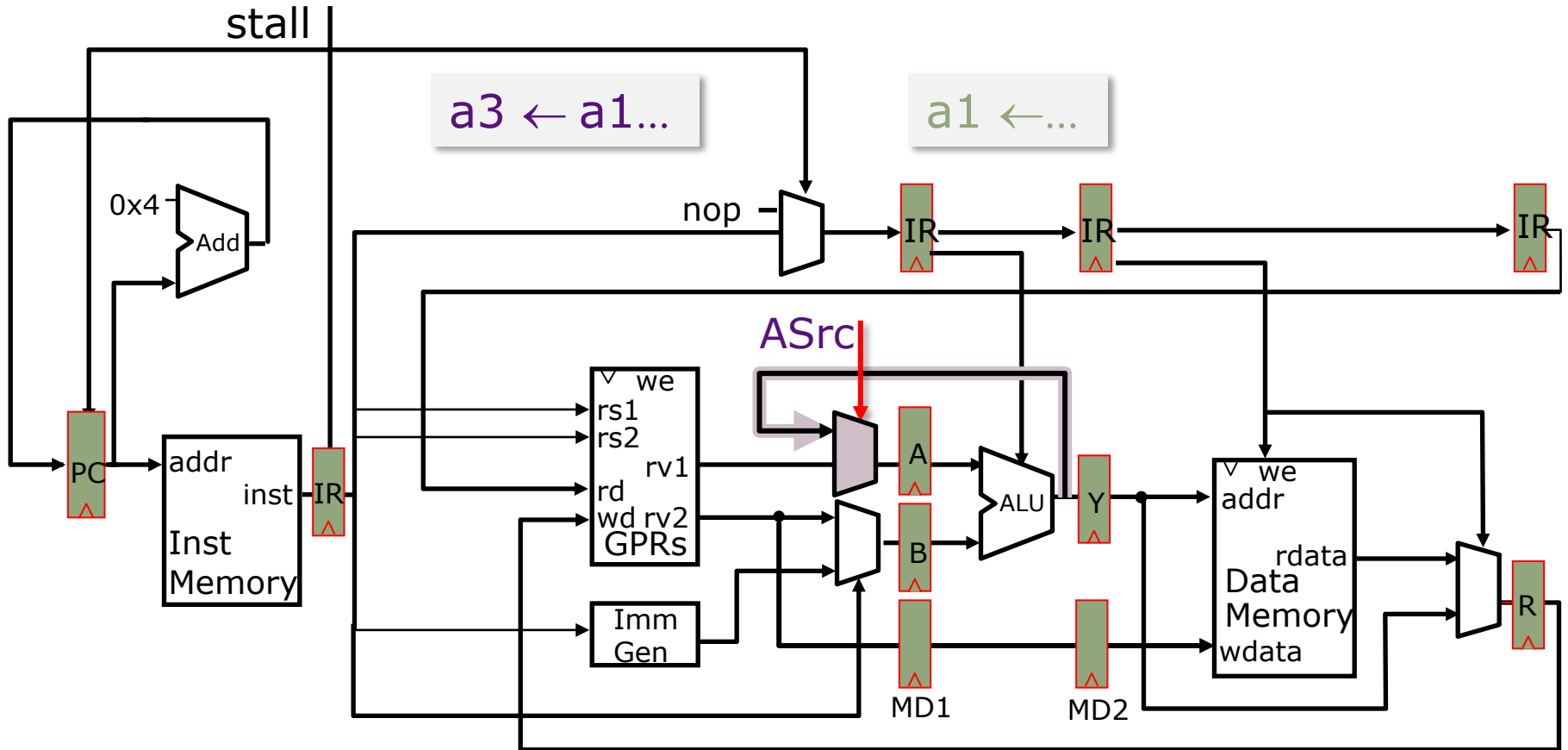
Each *stall or kill* introduces a bubble $\Rightarrow CPI > 1$

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?

(I₁) a1 ← a0 + 10
 (I₂) a3 ← a1 + 12

a1 ← M[a0 + 10]
 a3 ← a1 + 12

JAL ra 500
 t3 ← ra + 12

The Bypass Signal

Deriving it from the Stall Signal

$$\text{stall} = \cancel{((rs1_D == rd_E) \cdot we_E)} + (rs1_D == rd_M) \cdot we_M + (rs1_D == rd_W) \cdot we_W \cdot re1_D \\ + ((rs2_D == rd_E) \cdot we_E) + (rs2_D == rd_M) \cdot we_M + (rs2_D == rd_W) \cdot we_W \cdot re2_D$$

we = Case opcode
ALU, ALUi, LW, JAL, JALR
 $\Rightarrow (rd \neq 0)$
... \Rightarrow off

$$\text{ASrc} = (rs1_D == rd_E) \cdot we_E \cdot re1_D$$

Is this correct?

How might we address this?

Bypass and Stall Signals

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

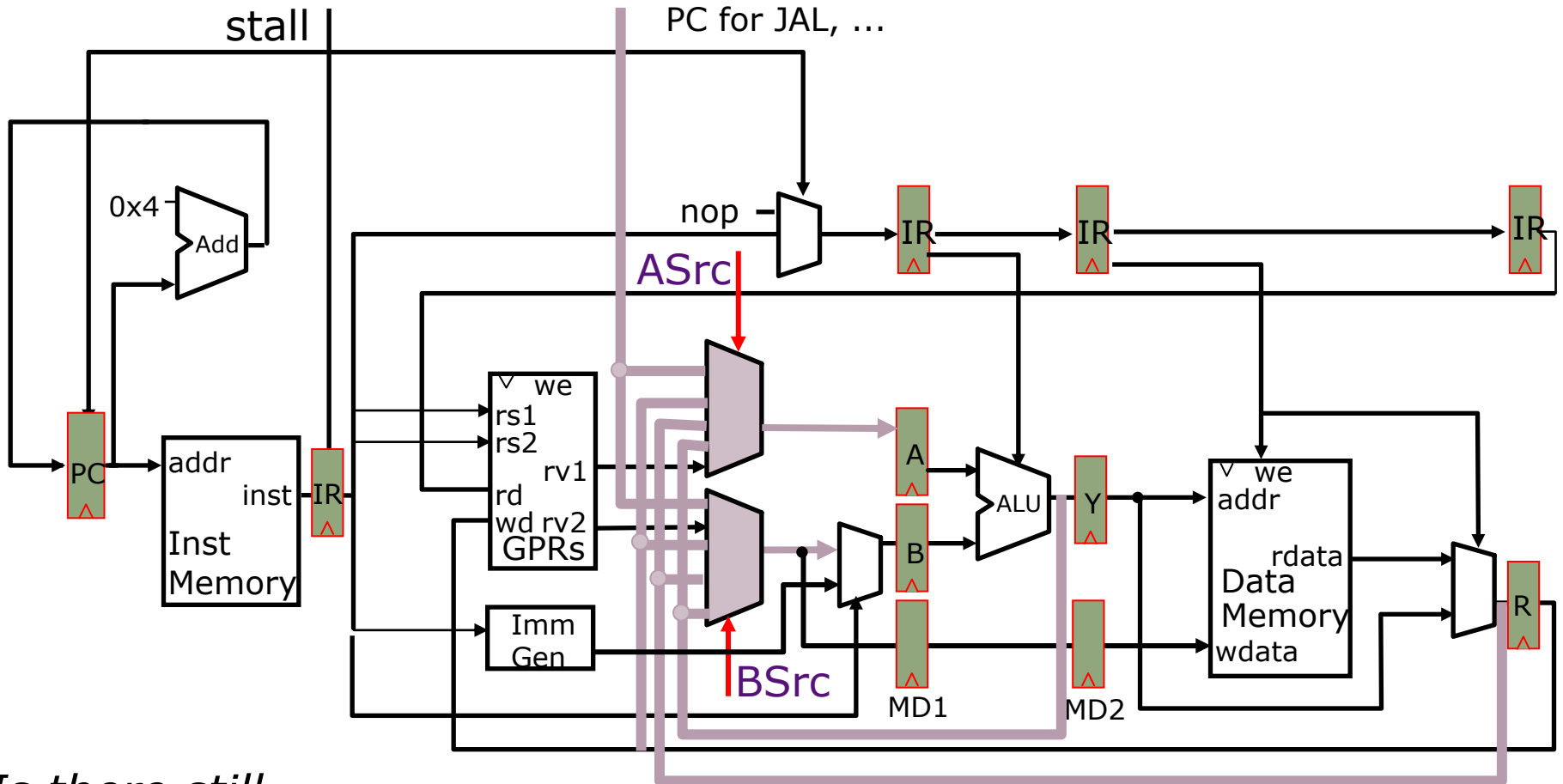
$we\text{-}bypass_E = \text{Case opcode}_E$
ALU, ALUi $\Rightarrow (rd \neq 0)$
... $\Rightarrow \text{off}$

$we\text{-}stall_E = \text{Case opcode}_E$
LW, JAL, JALR $\Rightarrow (rd \neq 0)$
... $\Rightarrow \text{off}$

$ASrc = (rs1_D == rd_E) \cdot re1_D \cdot we\text{-}bypass_E$

$stall = ((rs1_D == rd_E) \cdot we\text{-}stall_E +$
 $(rs1_D == rd_M) \cdot we_M + (rs1_D == rd_W) \cdot we_W) \cdot re1_D$
 $+ ((rs2_D == rd_E) \cdot we_E + (rs2_D == rd_M) \cdot we_M + (rs2_D == rd_W) \cdot we_W) \cdot re2_D$

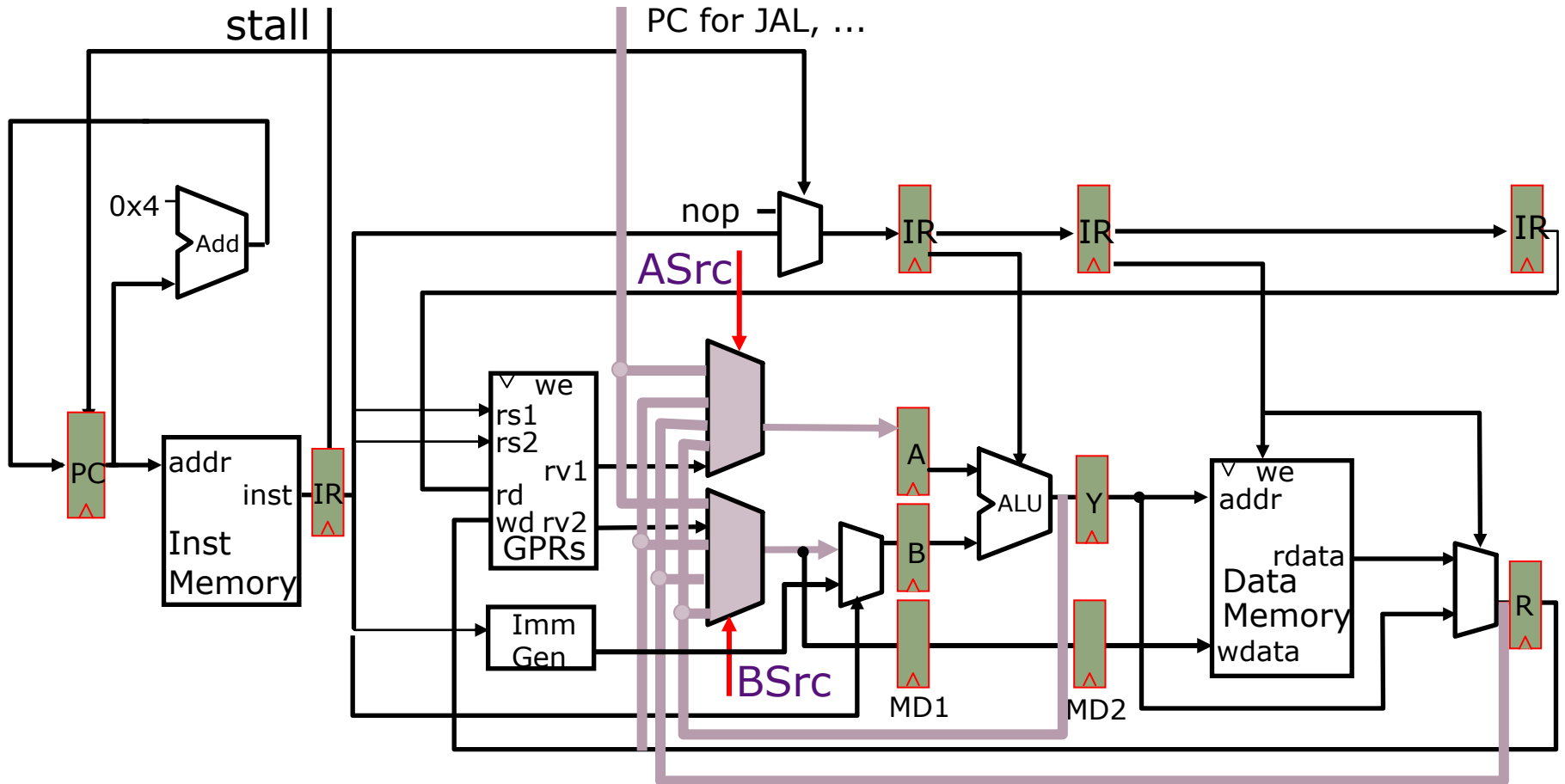
Full Bypass Datapath



*Is there still
a need for the
stall signal?*

$$\text{stall} = (\text{rs1}_D == \text{rd}_E) \cdot (\text{opcode}_E == \text{LW}_E) \cdot (\text{rd}_E \neq 0) \cdot \text{re1}_D \\ + (\text{rs2}_D == \text{rd}_E) \cdot (\text{opcode}_E == \text{LW}_E) \cdot (\text{rd}_E \neq 0) \cdot \text{re2}_D$$

Full Bypass Datapath



Challenge: How to scale bypass datapaths for more pipeline stages? More complex ISAs?

Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

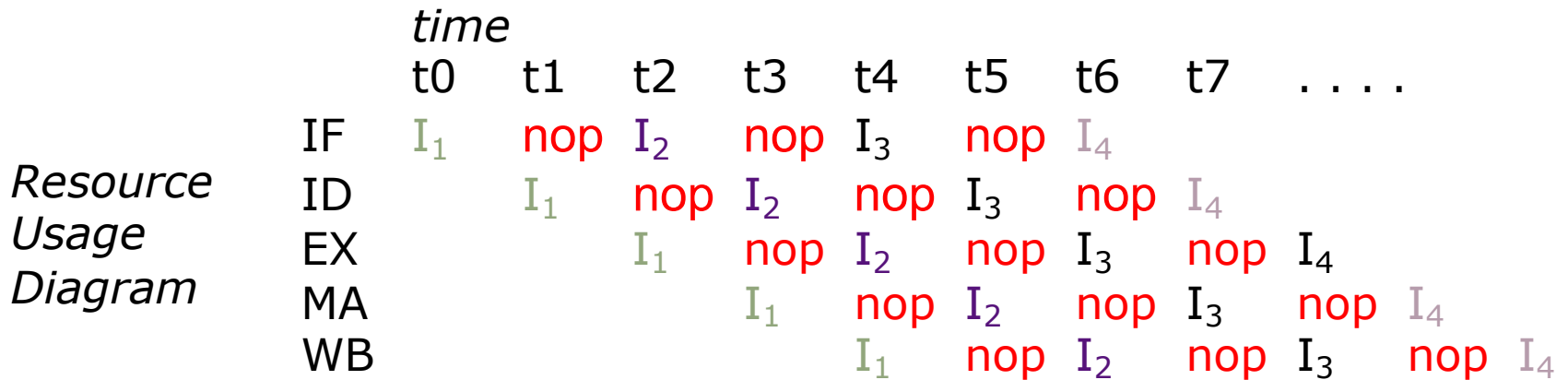
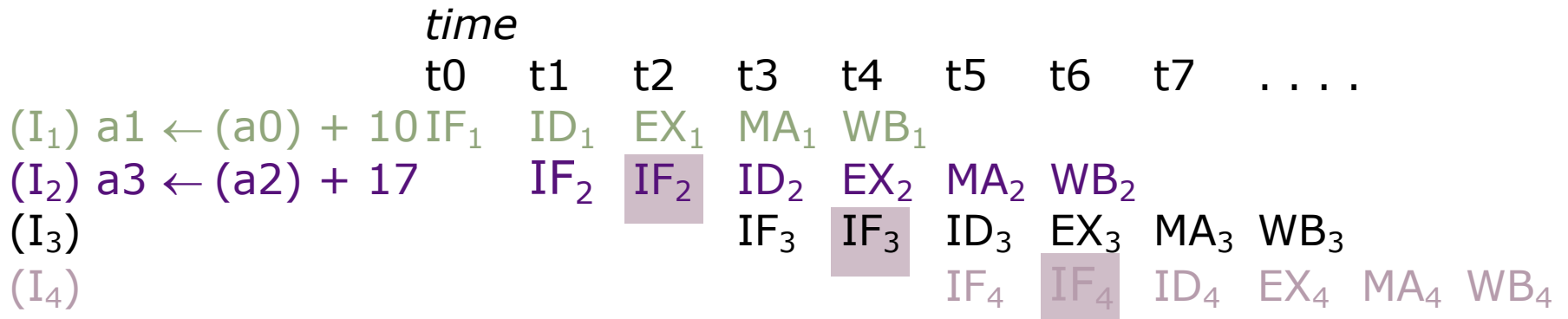
Guessed correctly → no special action required

Guessed incorrectly → kill and restart

Instruction to Instruction Dependence

- What do we need to calculate next PC?
 - For Jumps (JAL)
 - For Jump Register (JALR)
 - For Conditional Branches (e.g., BEQ)
 - For all others (e.g., arithmetic insts)
- In what stage do we know these?
 - PC →
 - Opcode, offset →
 - Register value →
 - Branch condition $((rs1) == (rs2))$ →

NextPC Calculation Bubbles

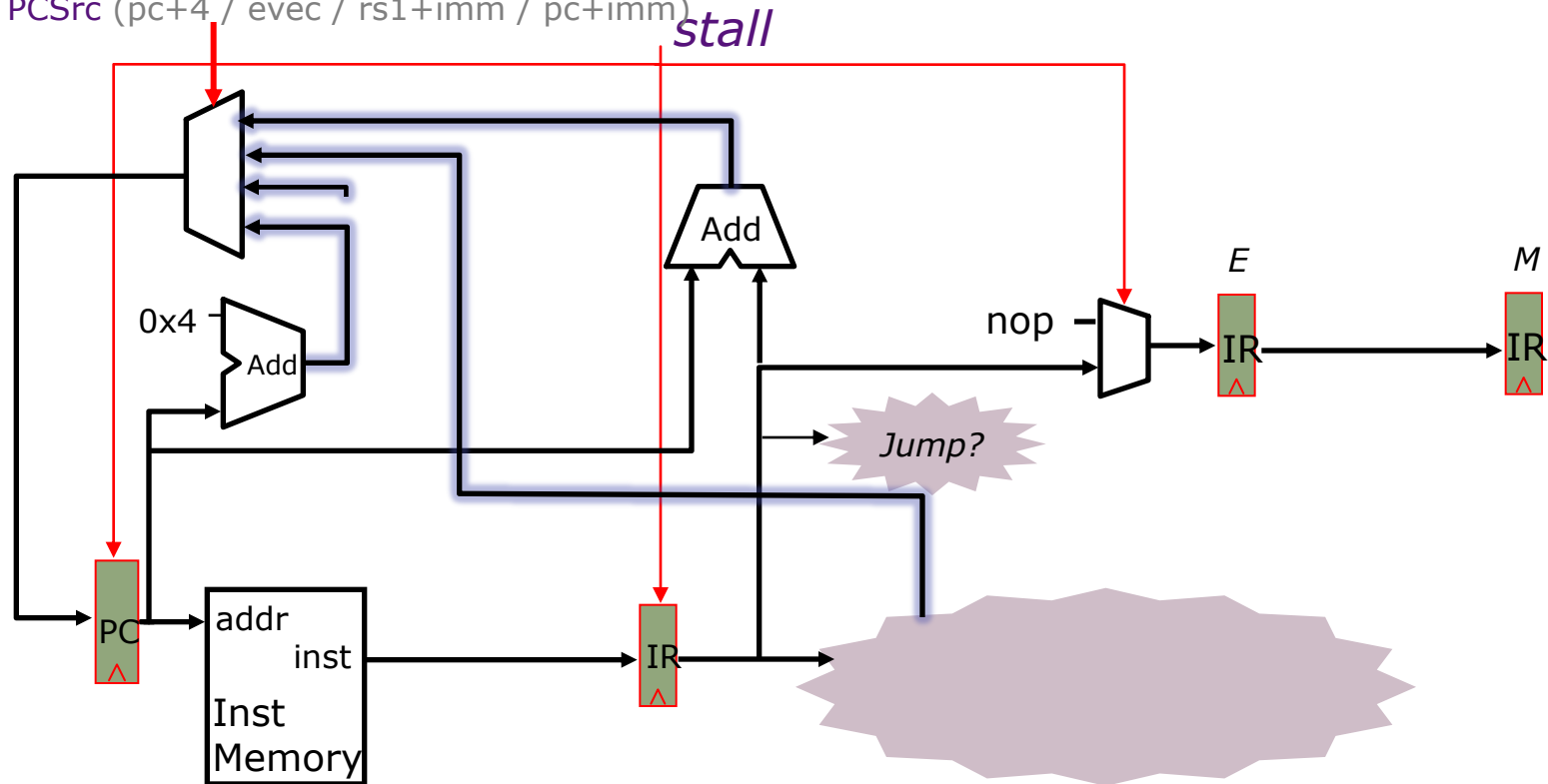


nop ⇒ *pipeline bubble*

What's a good guess for next PC?

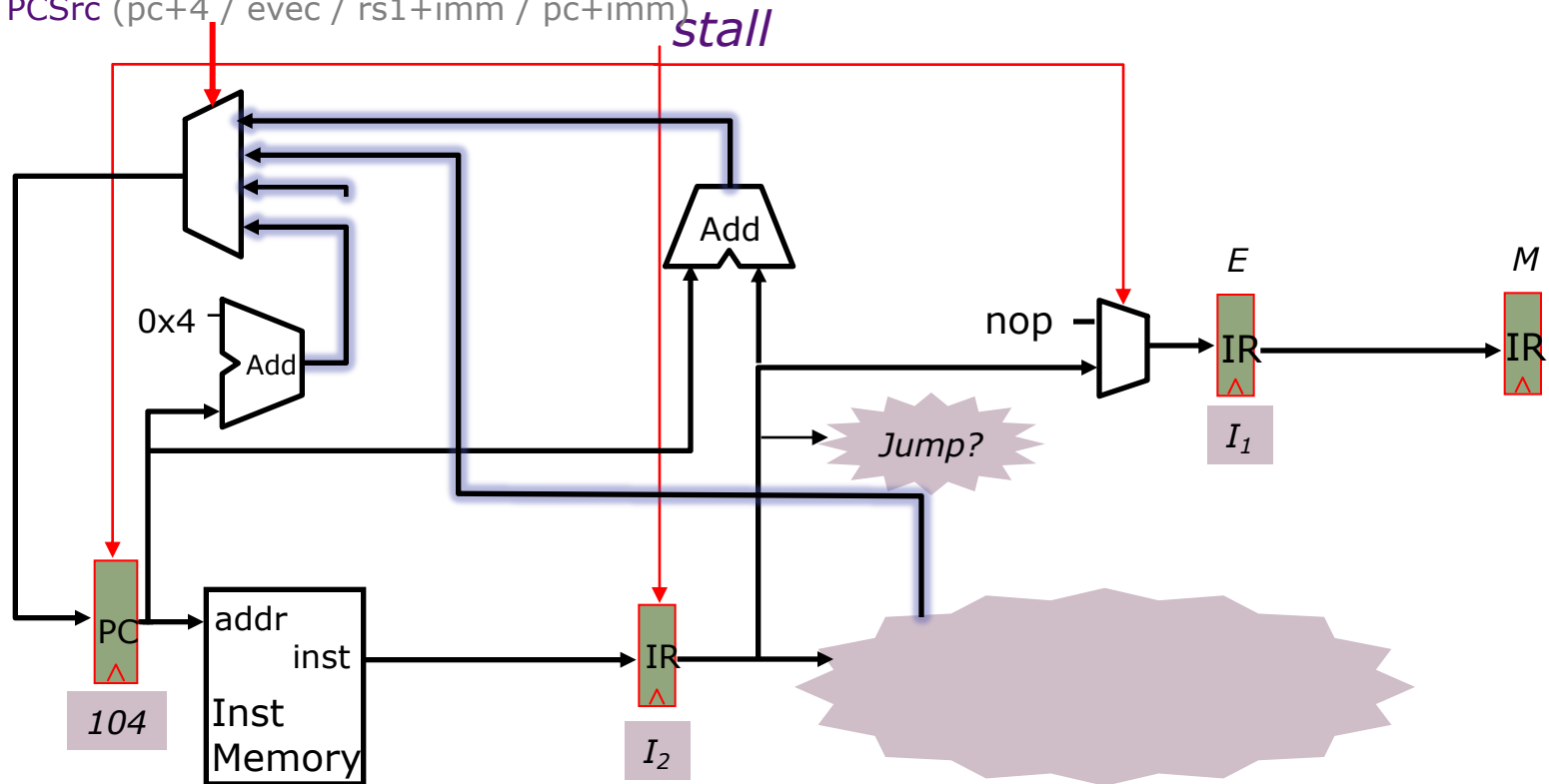
Speculate NextPC is PC+4

PCSrc (pc+4 / evec / rs1+imm / pc+imm)



Speculate NextPC is PC+4

PCSrc (pc+4 / evec / rs1+imm / pc+imm)



I_1	096	ADD	
I_2	100	JAL ra 200	
I_3	104	ADD	<i>kill</i>
I_4	304	ADD	

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

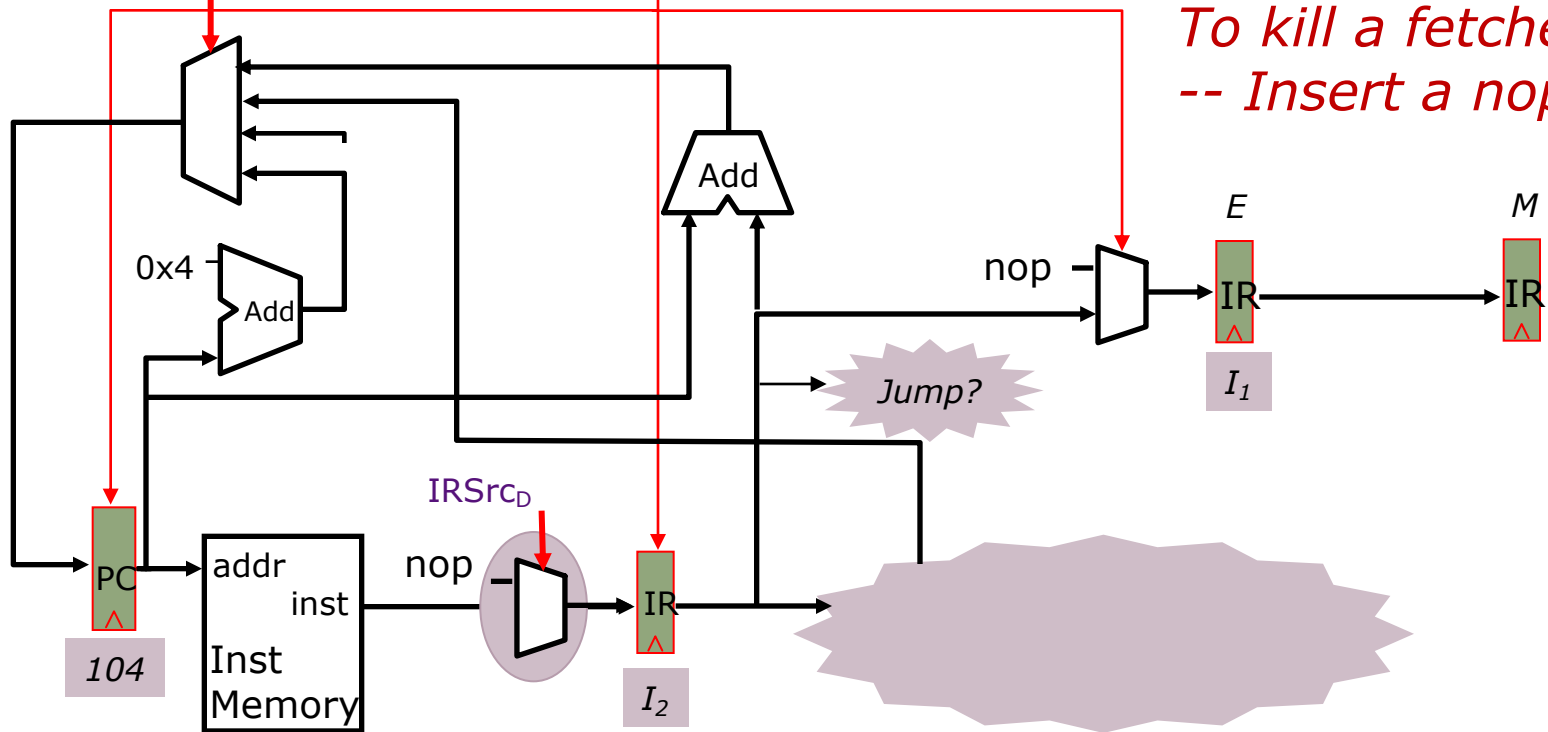
How?

Pipelining Jumps

PCSrc (pc+4 / evec / rs1+imm / pc+imm)

stall

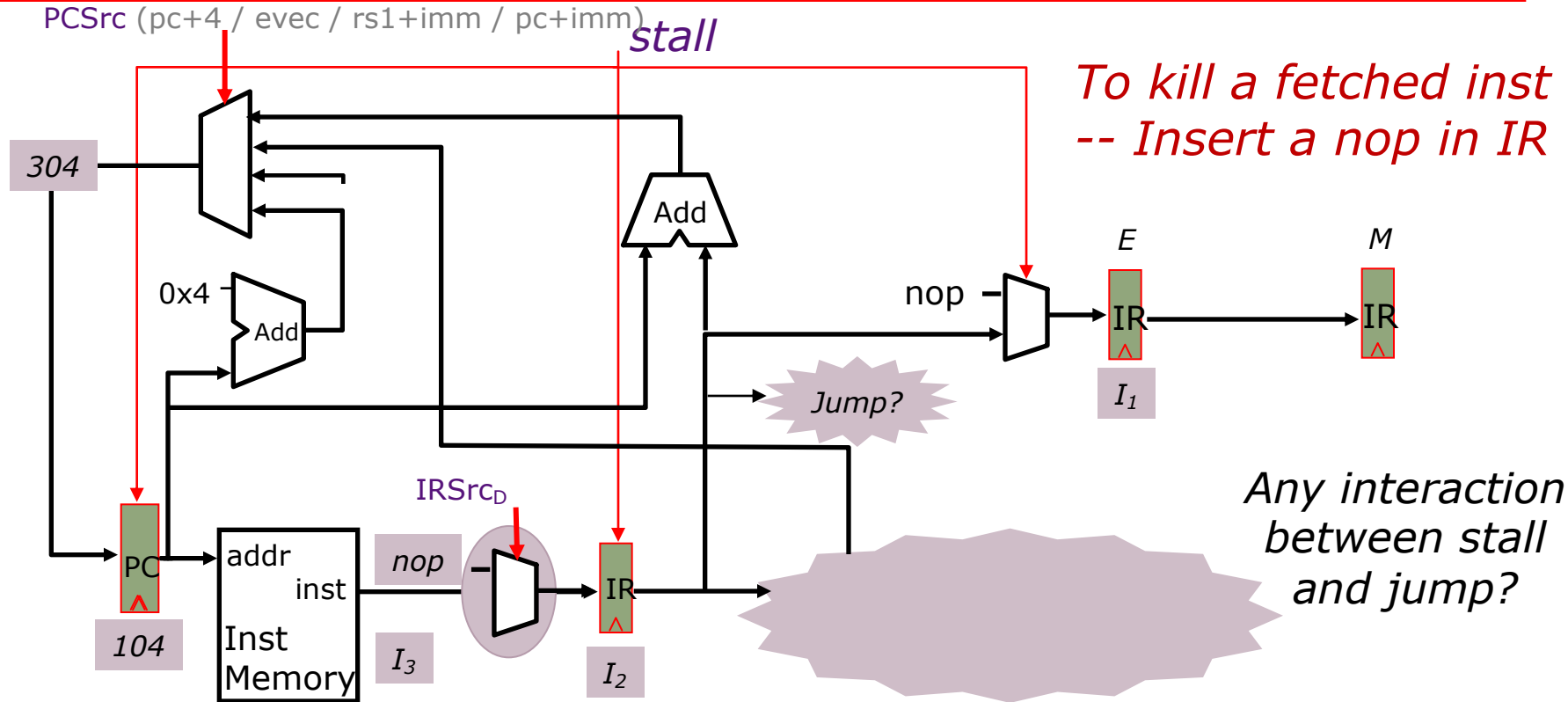
*To kill a fetched inst
-- Insert a nop in IR*



I ₁	096	ADD	
I ₂	100	JAL ra 200	
I ₃	104	ADD	<i>kill</i>
I ₄	304	ADD	

IRSrc_D = Case opcode_D
 JAL, JALR ⇒ nop
 ... ⇒ IM

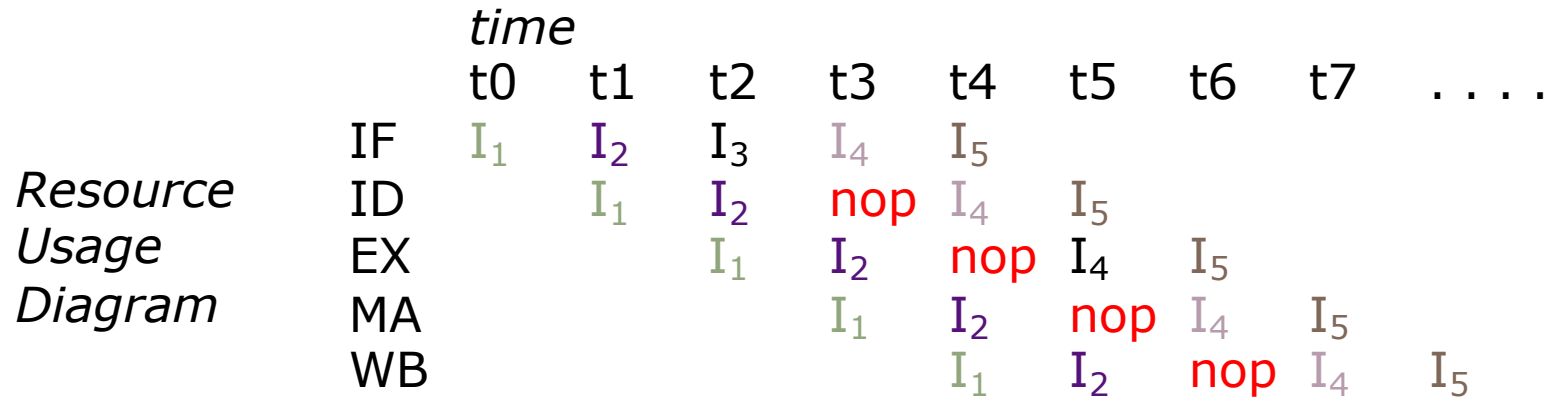
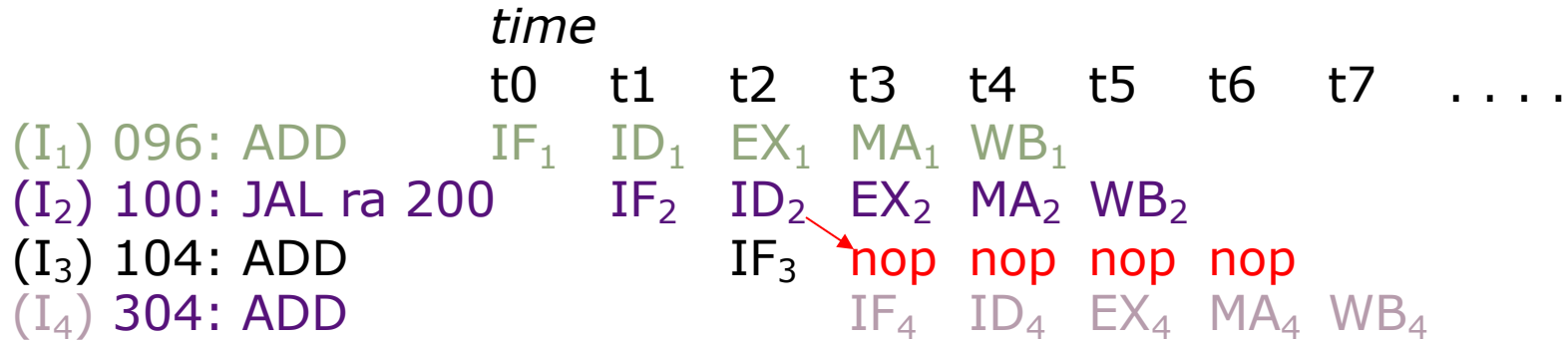
Pipelining Jumps



I_1	096	ADD	
I_2	100	JAL ra 200	
I_3	104	ADD	kill
I_4	304	ADD	

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

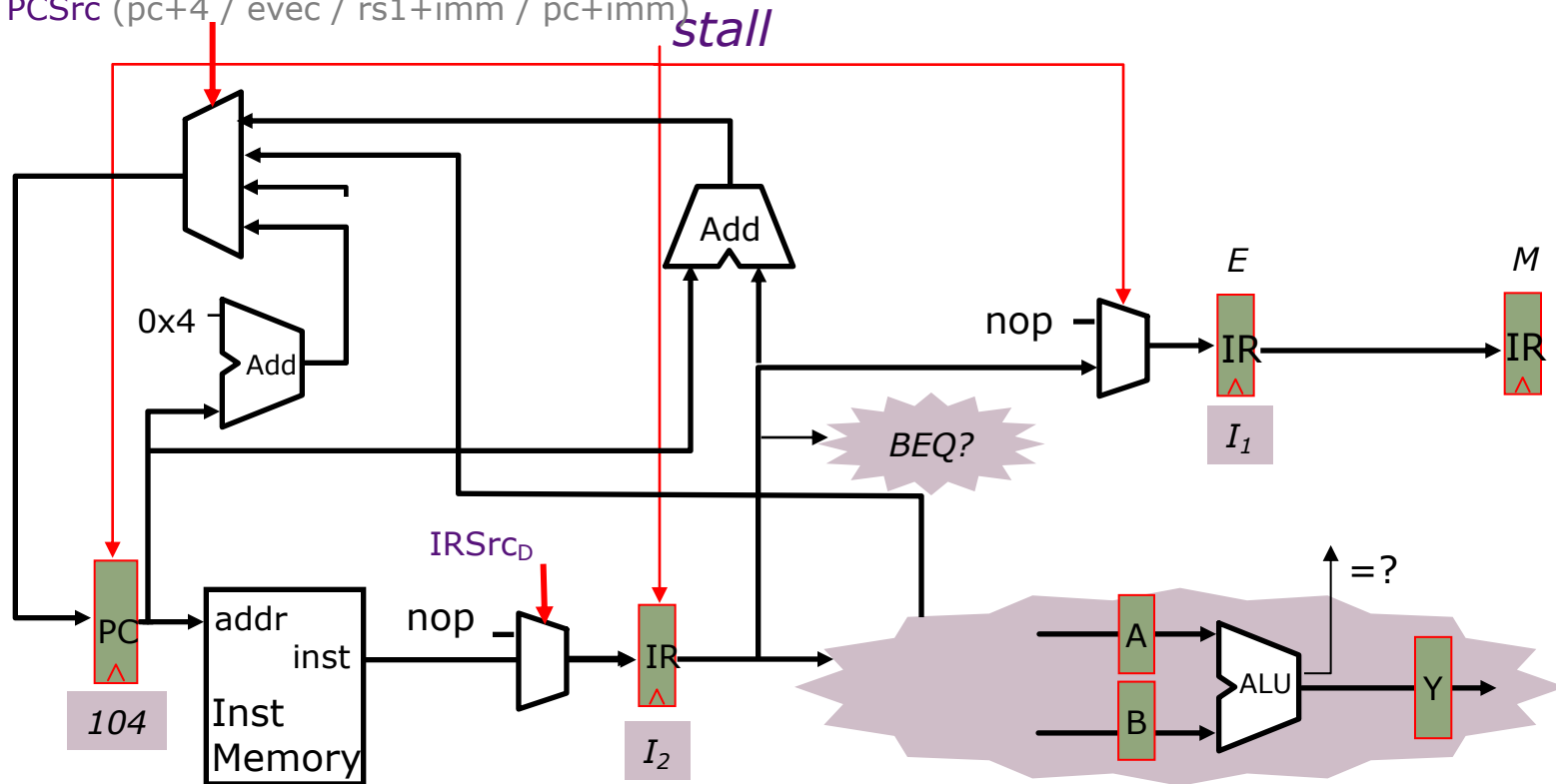
Jump Pipeline Diagrams



nop ⇒ *pipeline bubble*

Pipelining Conditional Branches

PCSrc (pc+4 / evec / rs1+imm / pc+imm)

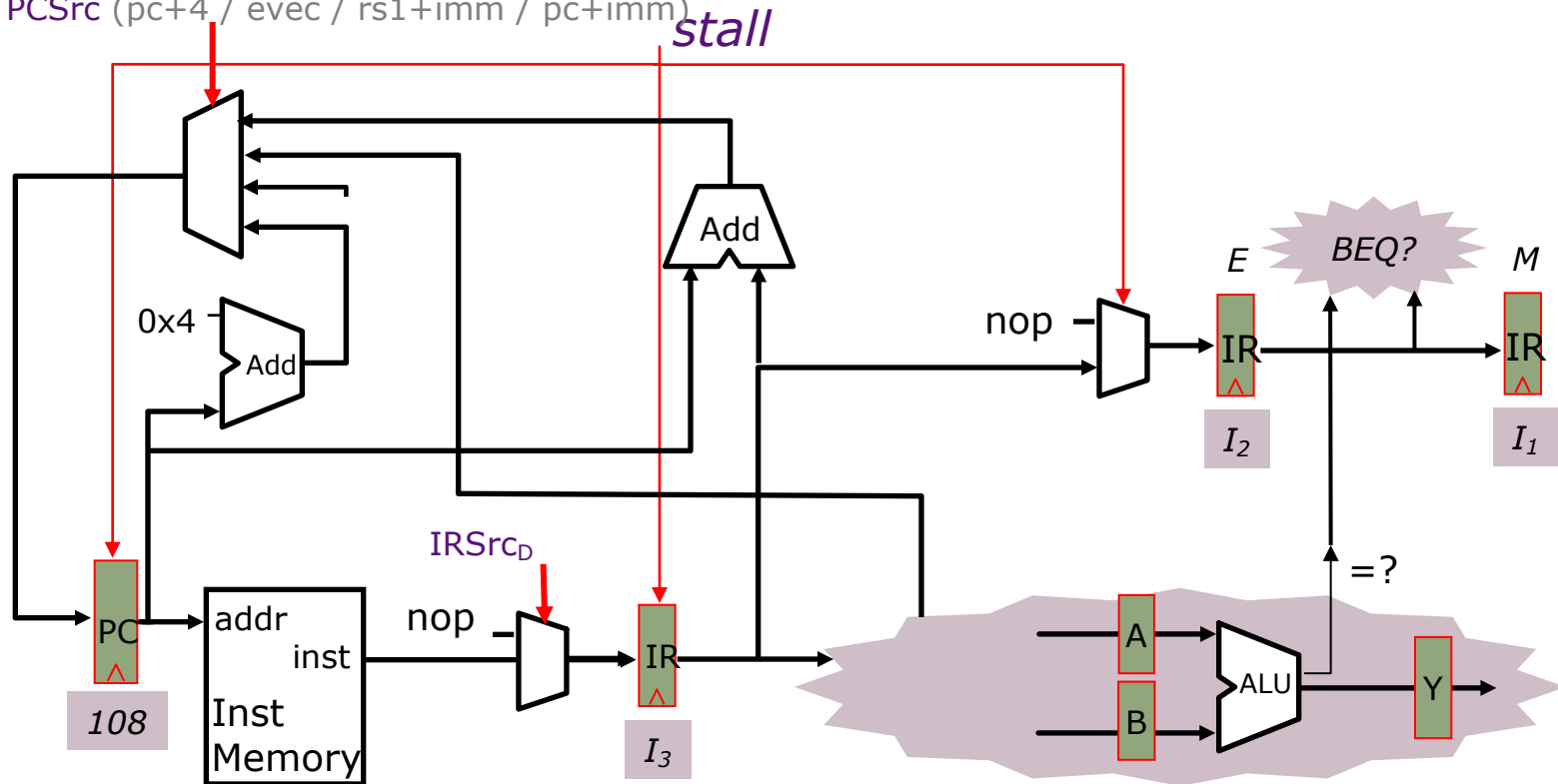


Branch condition is not known until the execute stage. *What action should be taken in the decode stage?*

I_1	096	ADD
I_2	100	BEQ a1 a2 200
I_3	104	ADD
I_4	304	ADD

Pipelining Conditional Branches

PCSrc (pc+4 / evec / rs1+imm / pc+imm)



I_1	096	ADD
I_2	100	BEQ a1 a2 200
I_3	104	ADD
I_4	304	ADD

If the branch is taken

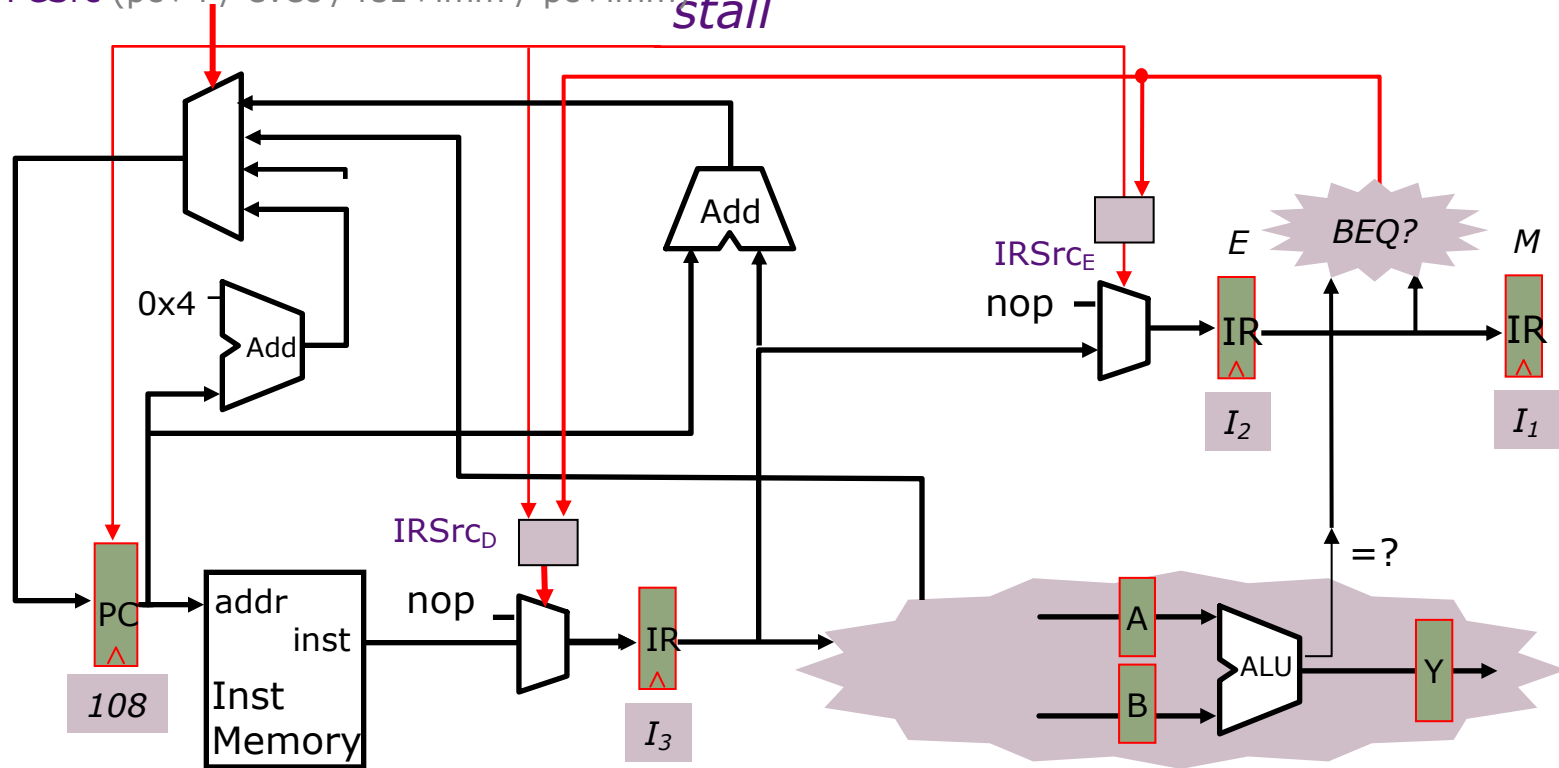
- kill the two following instructions
- the instruction at the decode stage is not valid

⇒ stall signal is not valid

Pipelining Conditional Branches

PCSrc (pc+4 / evec / rs1+imm / pc+imm)

stall



I_1	096	ADD
I_2	100	BEQ a1 a2 200
I_3	104	ADD
I_4	304	ADD

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

Control Equations for PC and IR Muxes

$IRSrc_D = \text{Case opcode}_E$
BEQ·z \Rightarrow nop
... \Rightarrow Case opcode_D
JAL, JALR \Rightarrow nop
... \Rightarrow IM

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

$IRSrc_E = \text{Case opcode}_E$
BEQ·z \Rightarrow nop
... \Rightarrow stall·nop + !stall·IR_D

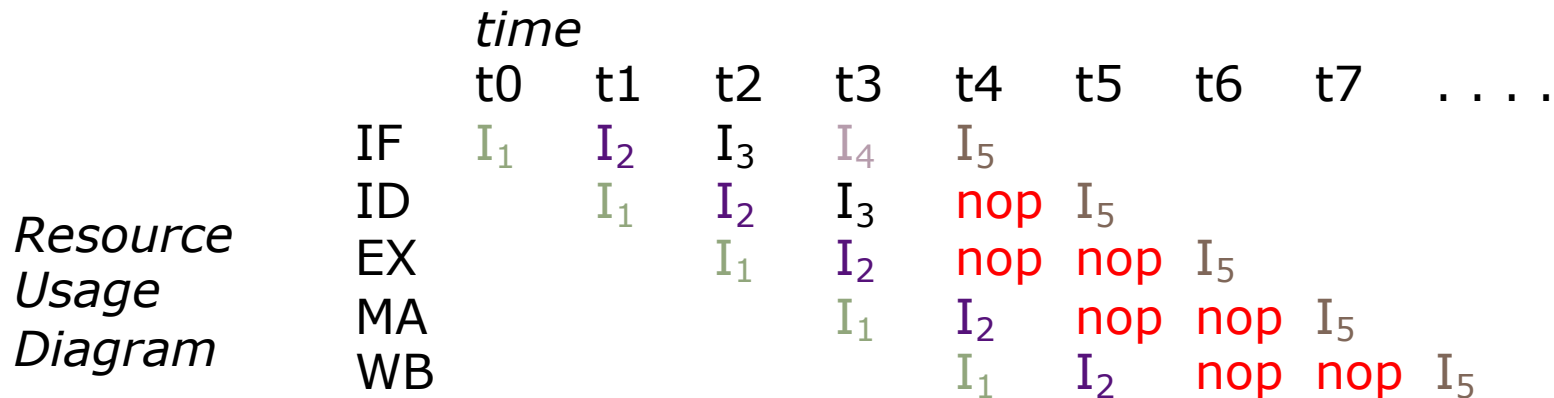
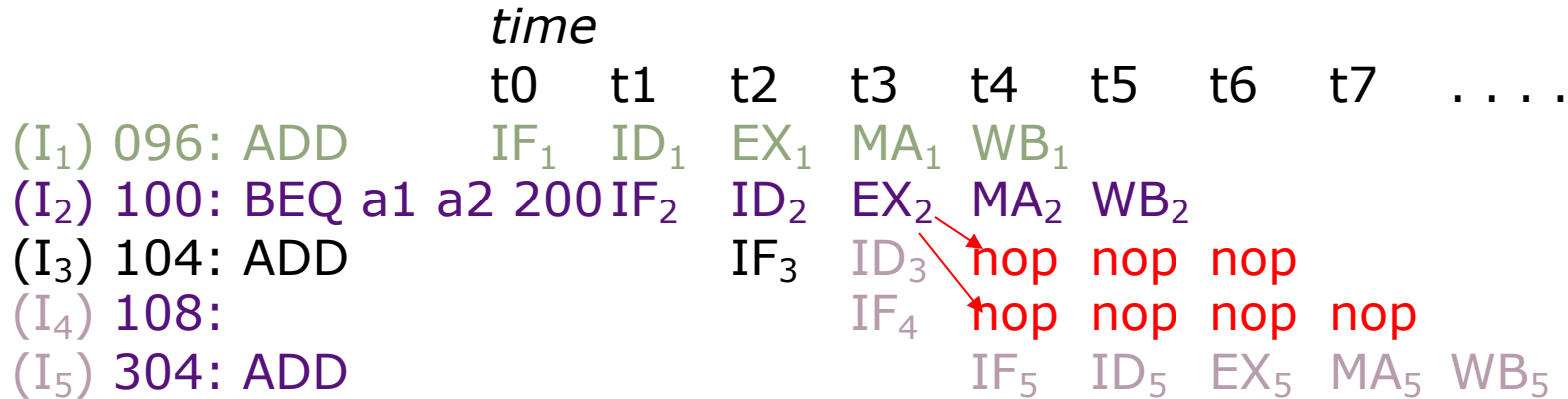
$PCSrc = \text{Case opcode}_E$
BEQ·z \Rightarrow pc + imm
... \Rightarrow Case opcode_D
JAL \Rightarrow pc + imm
JALR \Rightarrow rs1 + imm
... \Rightarrow pc+4

pc+4 is a speculative guess

nop \Rightarrow Kill
pc+imm / rs1+imm \Rightarrow Restart
pc+4 \Rightarrow Speculate

Branch Pipeline Diagrams

(resolved in execute stage)

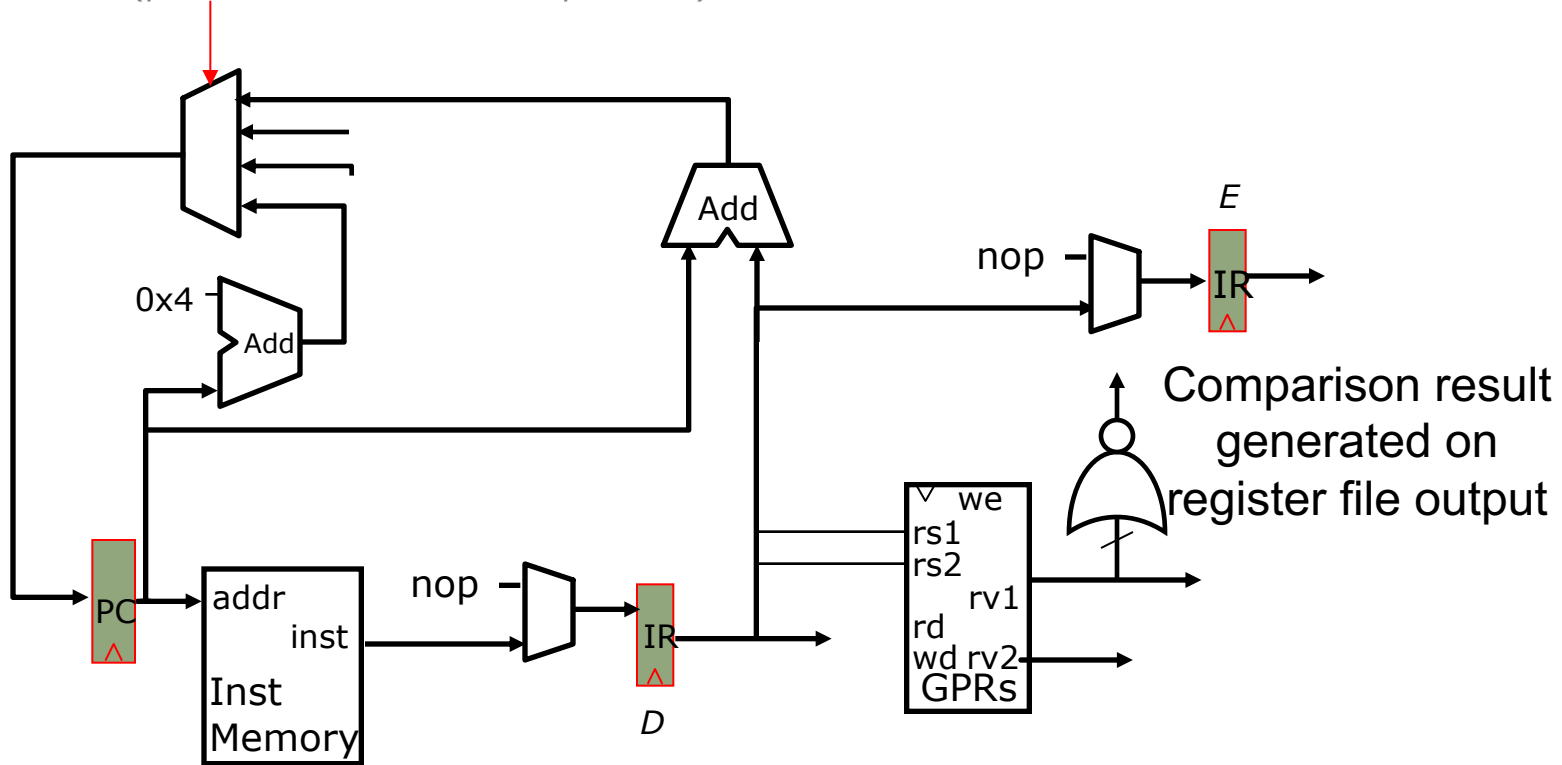


nop ⇒ *pipeline bubble*

Reducing Branch Penalty (resolve in decode stage)

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

PCSrc (pc+4 / evec / rs1+imm / pc+imm)



Pipeline diagram now same as for jumps

Branch Delay Slots

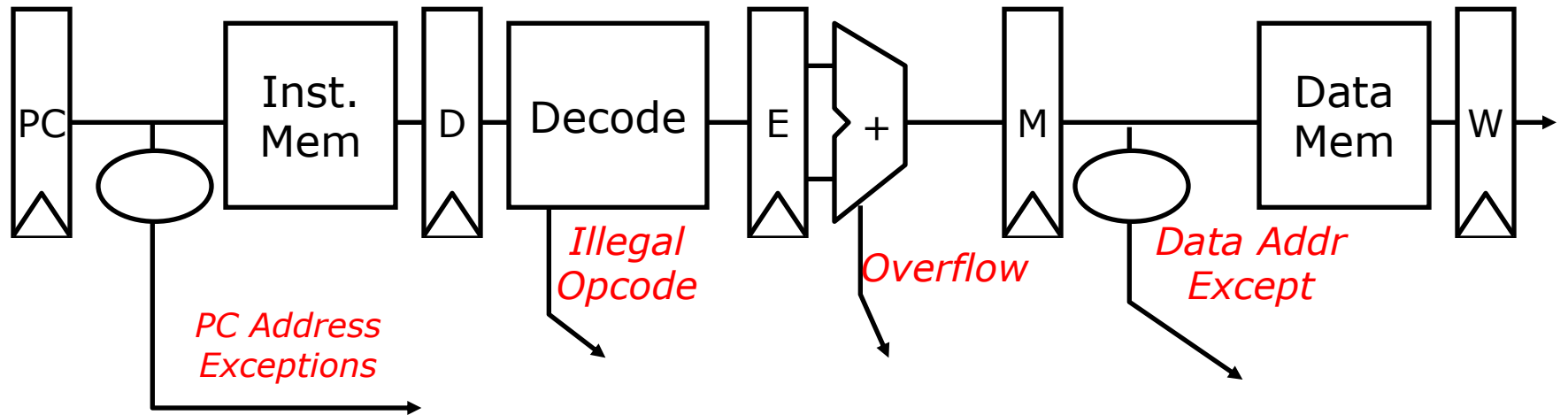
(expose control hazard to software)

- 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 ₁	096	ADD		
I ₂	100	BEQ a1 a2 200		<i>Delay slot instruction</i>
I ₃	104	ADD	←	<i>executed regardless of</i>
I ₄	304	ADD		<i>branch outcome</i>

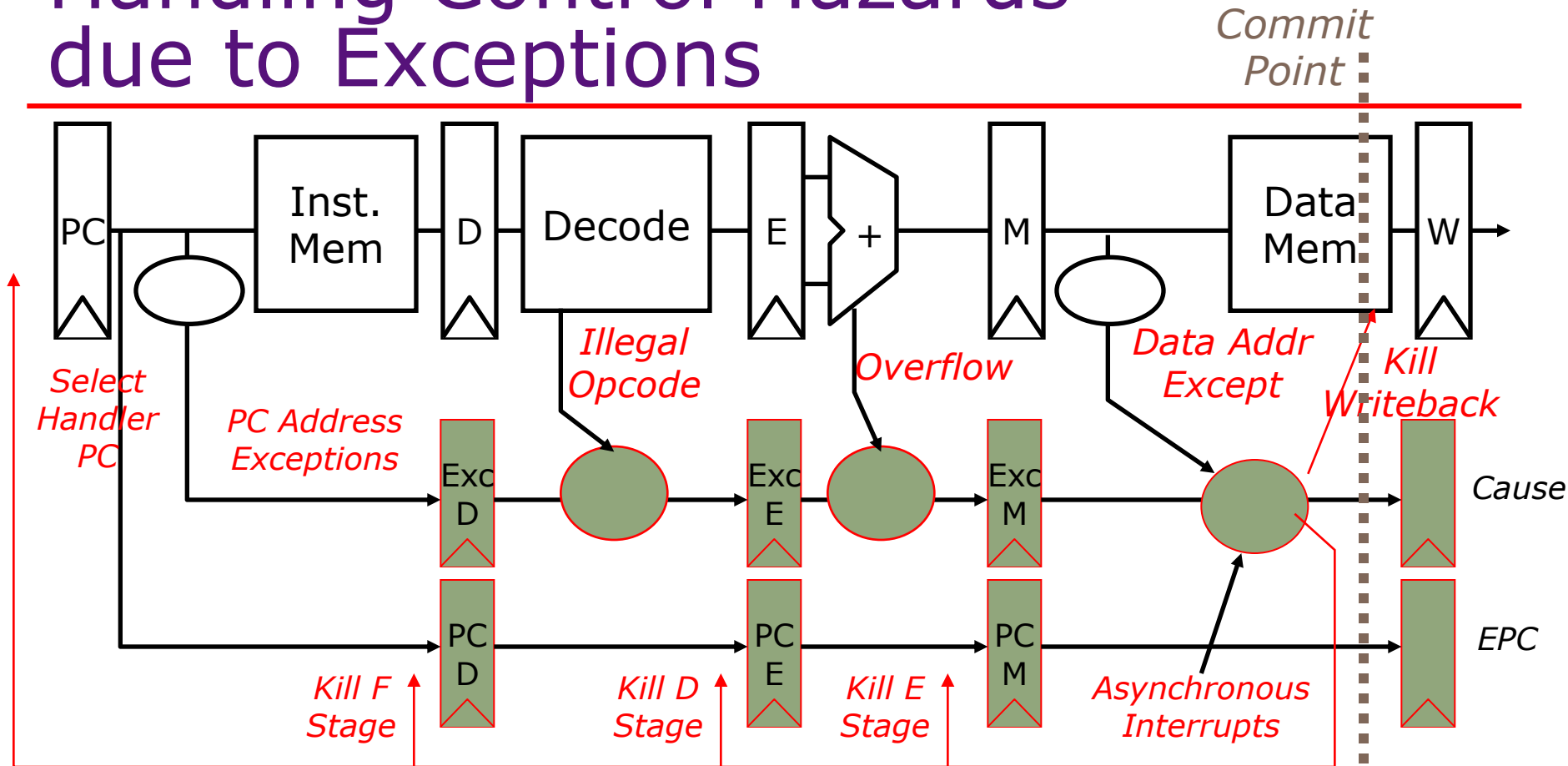
- Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*

Handling Control Hazards due to Exceptions



- Instructions may suffer exceptions in different pipeline stages
- Must prioritize exceptions from earlier instructions

Handling Control Hazards due to Exceptions



- Typical strategy: Record exceptions, process the first one to reach commit point (i.e., the point where architectural state is modified)
 - *Pros/cons vs handling exceptions eagerly, like branches?*

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, jumps, and exceptions may cause bubbles
 - Kill instruction(s) following branch if no delay slots

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

Next lecture:
Superscalar & Scoreboarded
Pipelines