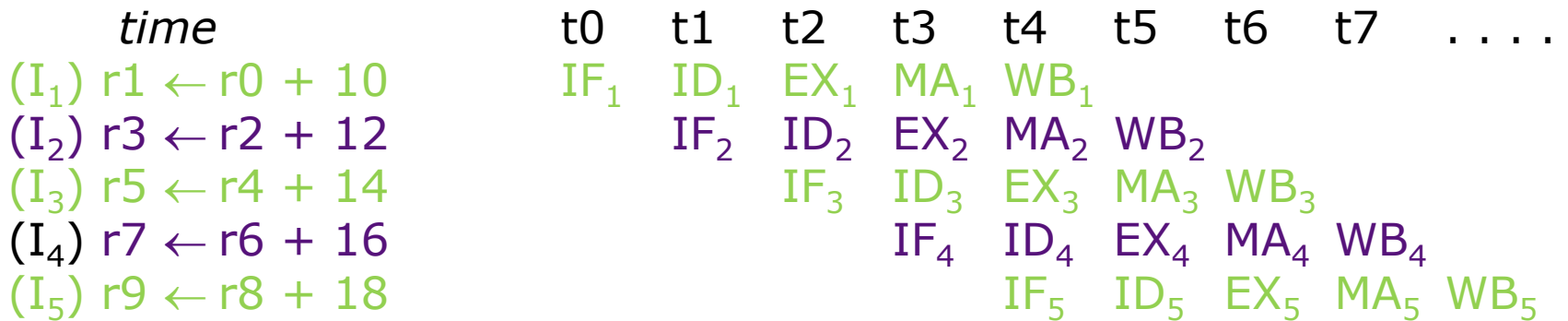


Instruction Pipelining: Hazard Resolution, Timing Constraints

Joel Emer

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

Pipeline Diagram – Ideal Pipelining



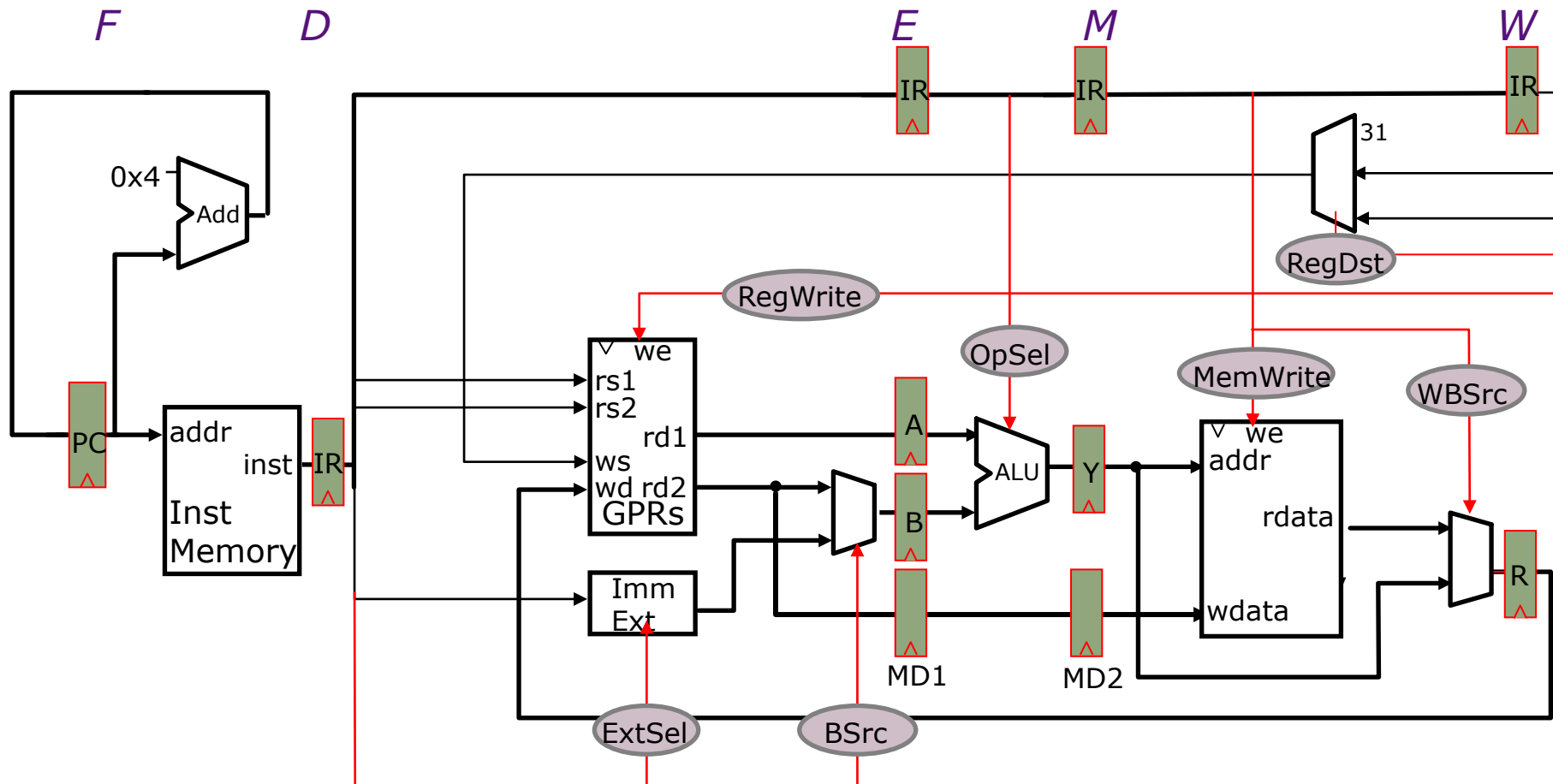
Over long term, i.e., in steady state, what is...

T
L
 CPI? 1 IPC? 1 Inst exec time? 5

N
 Num inst in flight? T = N/L -> 5

Reminder: Pipelined MIPS Datapath

without jumps

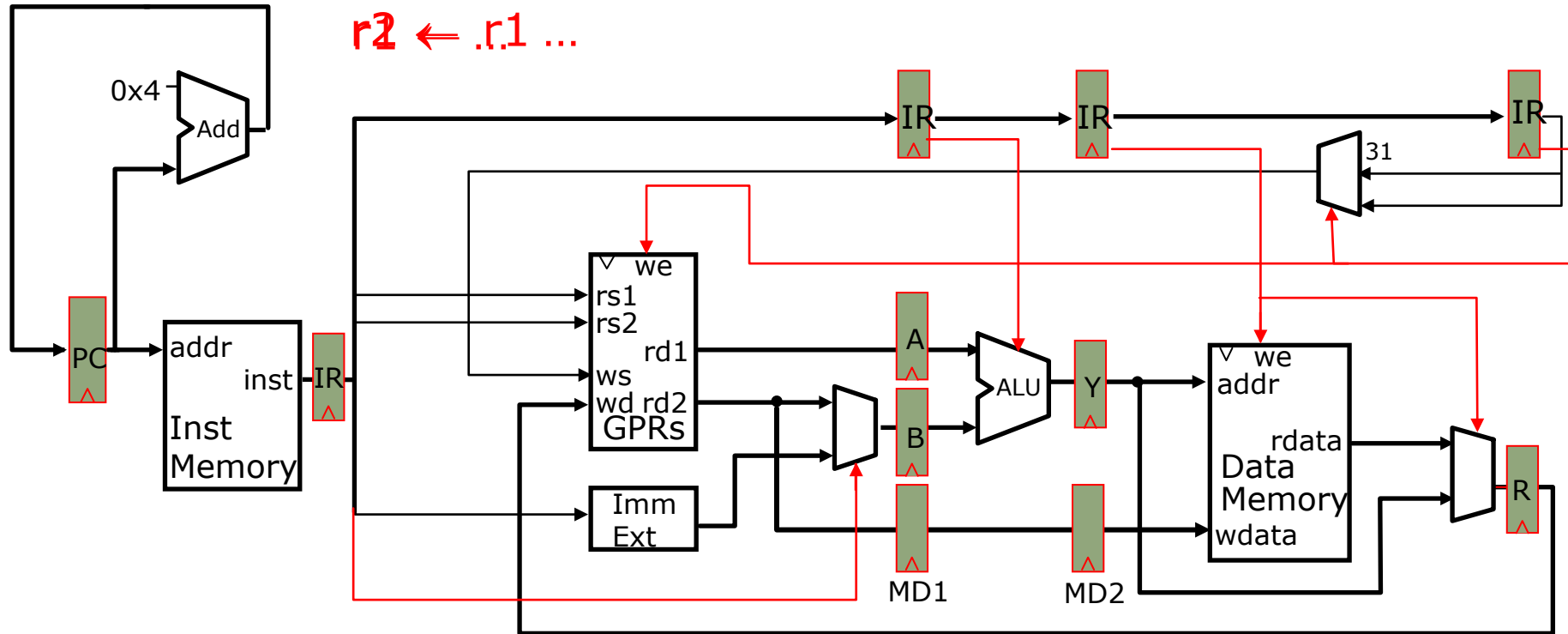


Pipelining increases clock frequency,
but instruction dependences may increase CPI

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



...

$r1 \leftarrow r0 + 10$

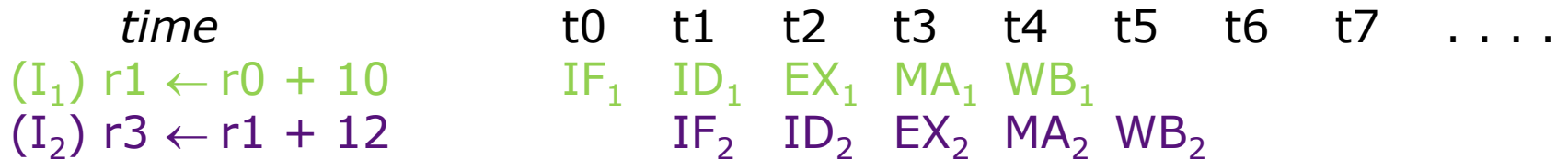
$r3 \leftarrow r1 + 12$

...

Cycle 0

$r1$ is stale. Oops!

Pipeline Diagram – Hazard



r1 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:

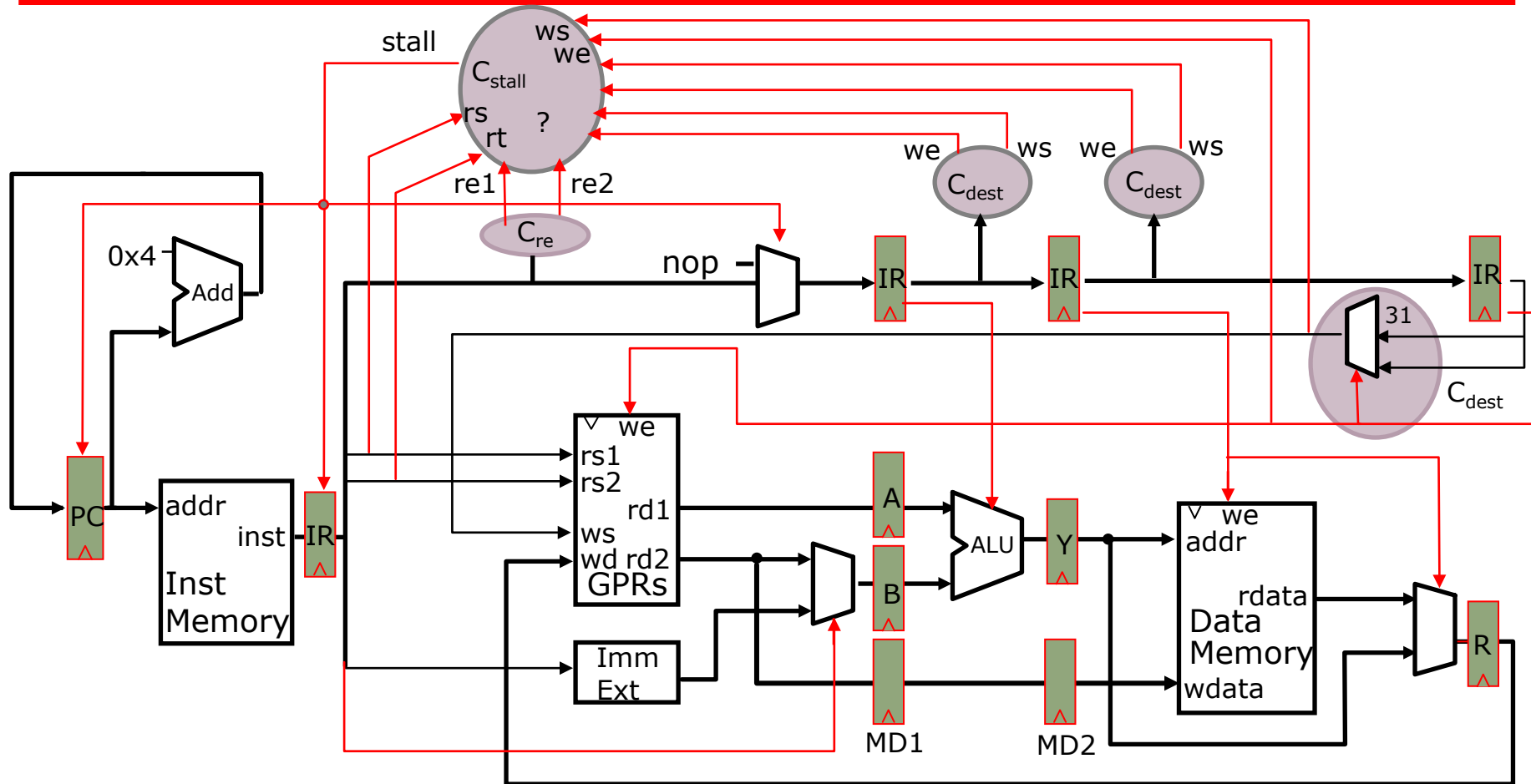
Guessed correctly → no special action required
Guessed incorrectly → kill and restart

Resolving Data Hazards

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

Reminder: Stall Control Logic

ignoring jumps & branches



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

Source & Destination Registers

R-type:

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

I-type:

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

J-type:

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

source(s) destination

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

Deriving the Stall Signal

C_{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_{re}

re1 = Case opcode

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

re2 = Case opcode

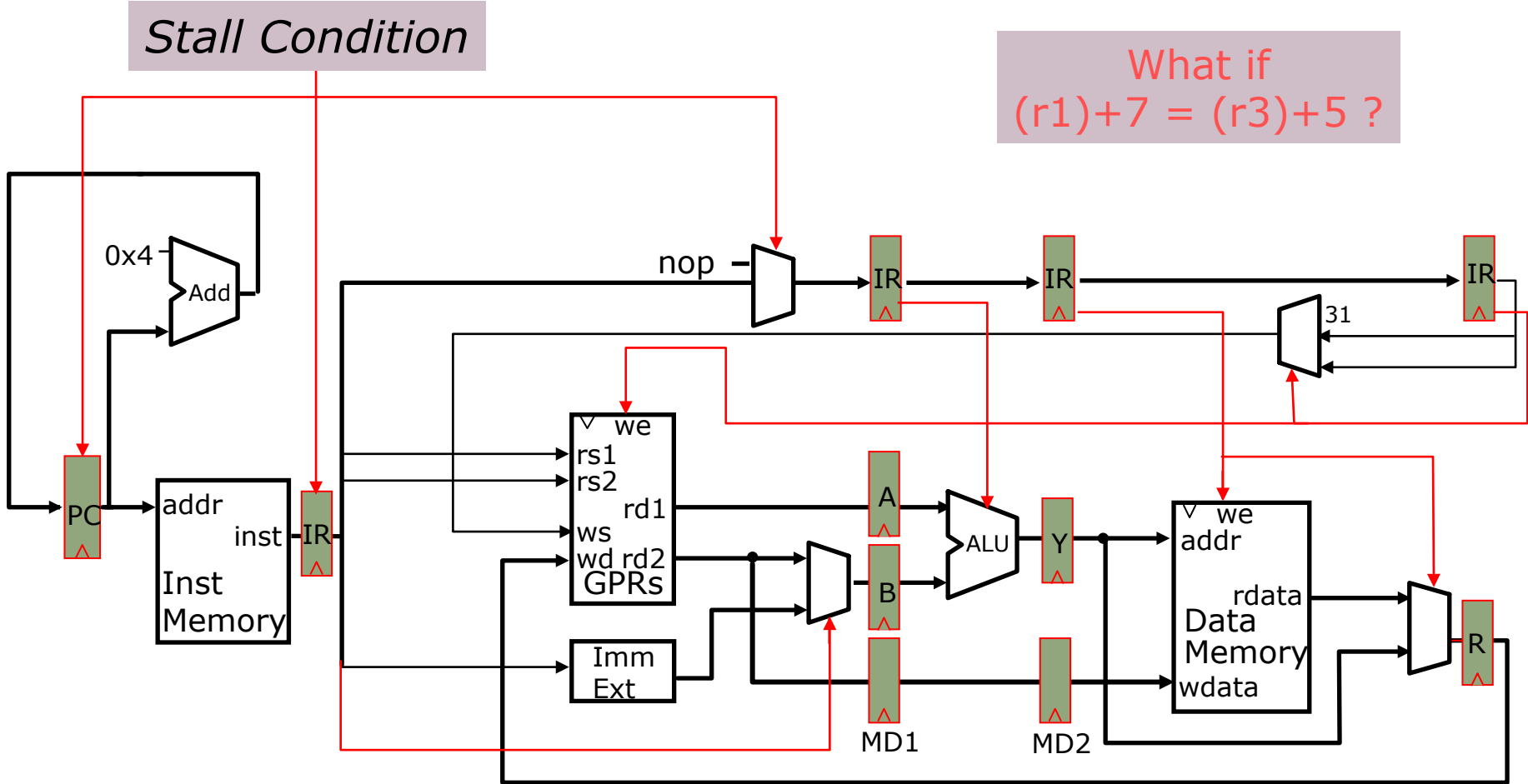
ALU, SW $\Rightarrow$ on
 ... $\Rightarrow$ off

C_{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



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

Is there any possible data hazard in this instruction sequence?

Load & Store Hazards

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

$(r1)+7 = (r3)+5 \Rightarrow \text{data hazard}$

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.

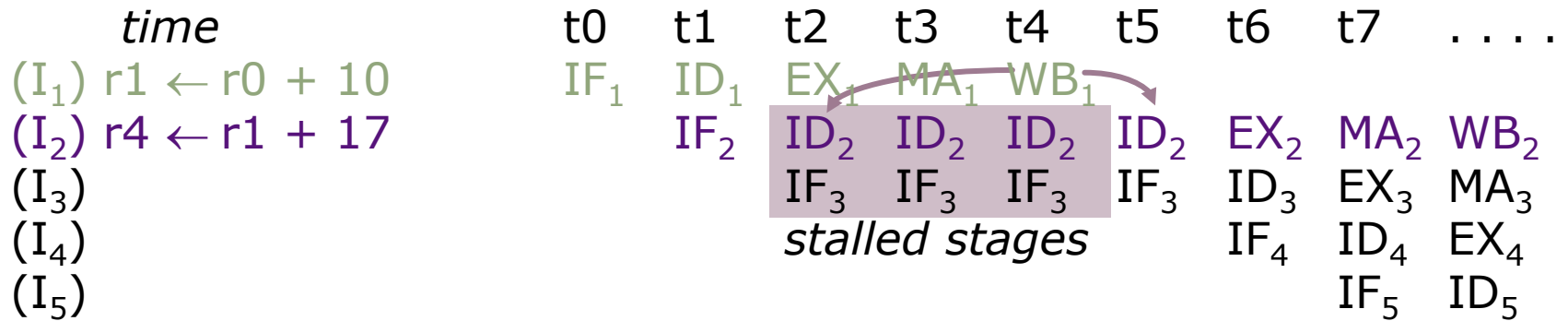
More on this later in the course.

Resolving Data Hazards (2)

Strategy 2:

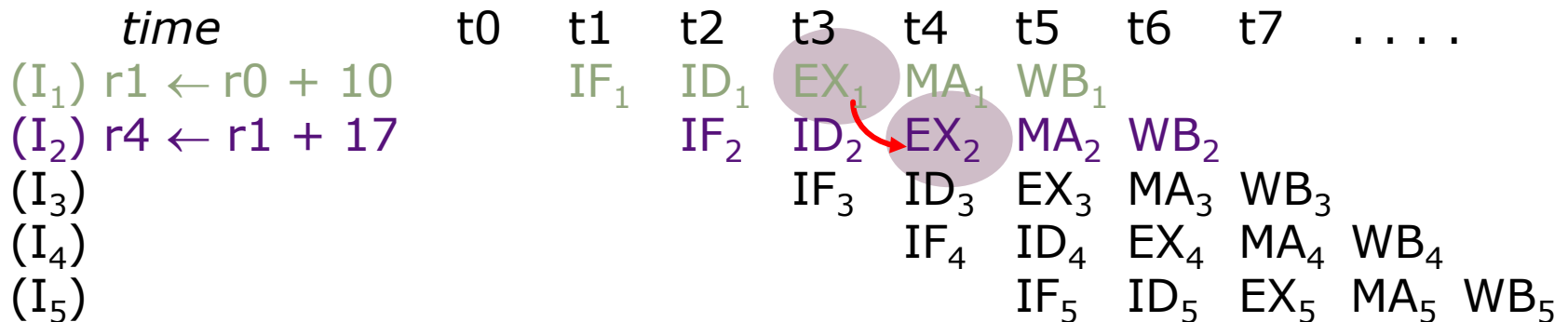
Route data as soon as possible after it is calculated to the earlier pipeline stage → *bypass*

Bypassing



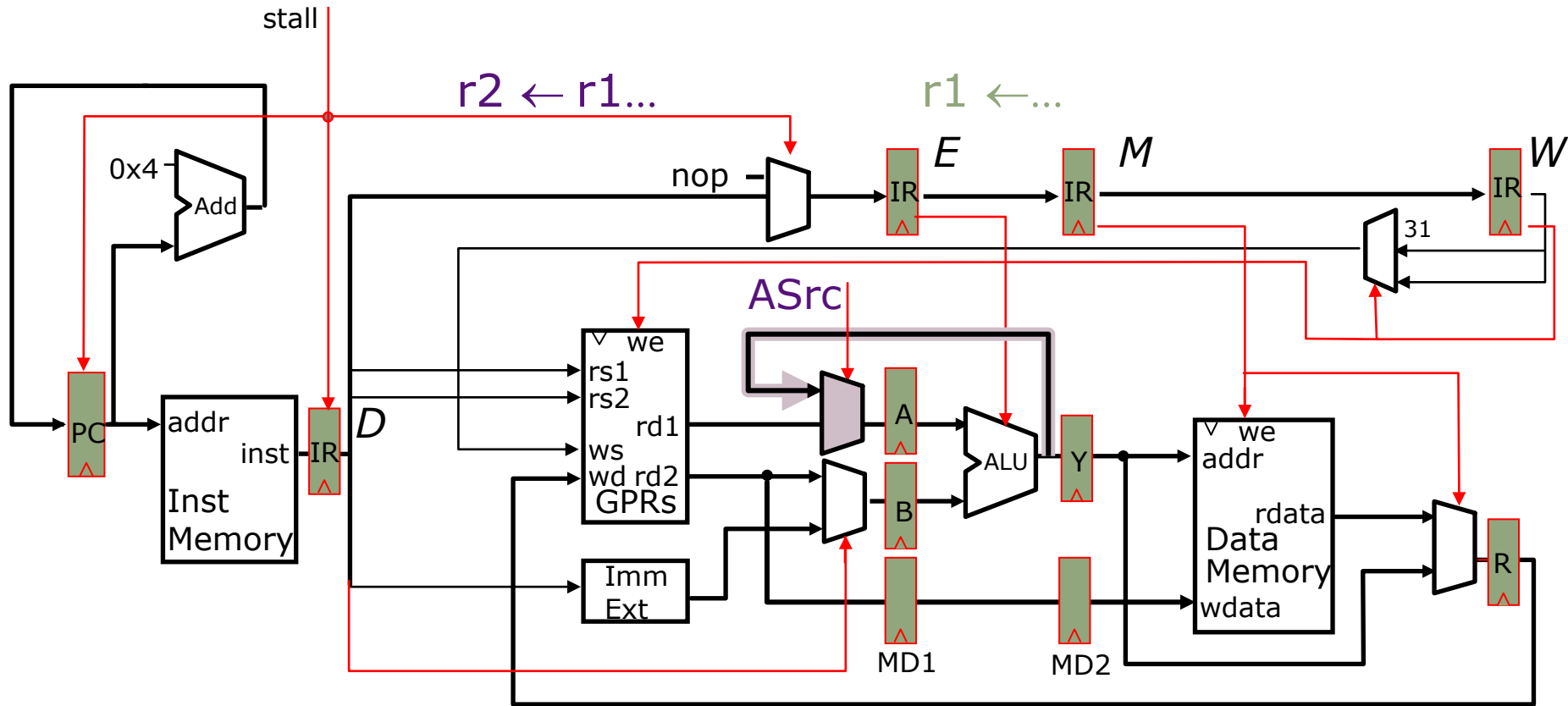
Each *stall or kill* introduces a bubble $\Rightarrow CPI > 1$

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)	$r1 \leftarrow r0 + 10$	$r1 \leftarrow M[r0 + 10]$	JAL 500
(I_2)	$r2 \leftarrow r1 + 12$	$r2 \leftarrow r1 + 12$	$r2 \leftarrow r31 + 12$
	yes	no	no

The Bypass Signal

Deriving it from the Stall Signal

$$\text{stall} = ((\text{rs}_D == \text{ws}_E) \cdot \text{we}_E + (\text{rs}_D == \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D == \text{ws}_W) \cdot \text{we}_W) \cdot \text{re1}_D \\ + ((\text{rt}_D == \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D == \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D == \text{ws}_W) \cdot \text{we}_W) \cdot \text{re2}_D$$

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

$$\text{ASrc} = (\text{rs}_D == \text{ws}_E) \cdot \text{we}_E \cdot \text{re1}_D$$

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

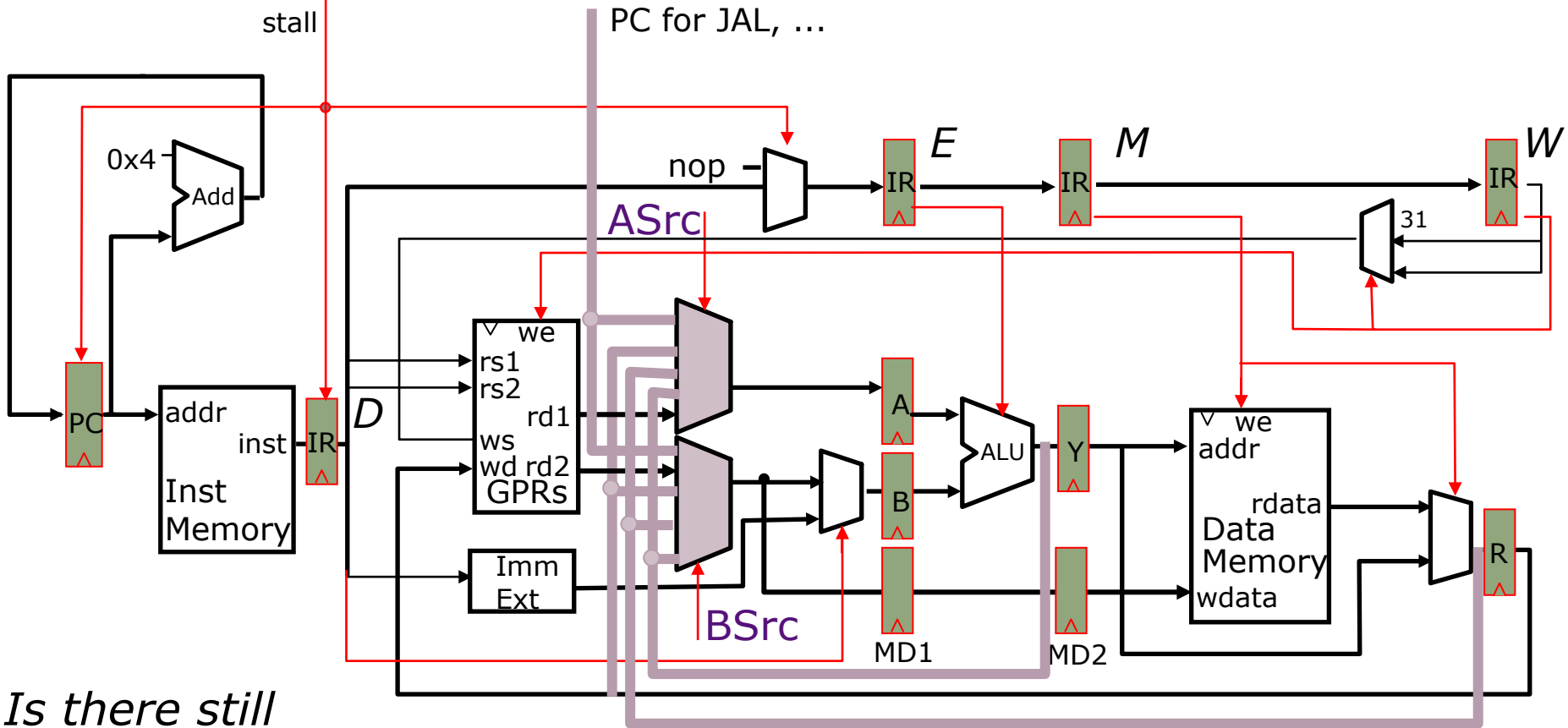
$we_bypass_E = \text{Case opcode}_E$
ALU, ALUi $\Rightarrow (ws \neq 0)$
... $\Rightarrow \text{off}$

$we_stall_E = \text{Case opcode}_E$
LW $\Rightarrow (ws \neq 0)$
JAL, JALR $\Rightarrow \text{on}$
... $\Rightarrow \text{off}$

$ASrc = (rs_D == ws_E) \cdot we_bypass_E \cdot re1_D$

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

Fully Bypassed Datapath



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

$$\text{stall} = (\text{rs}_D == \text{ws}_E) \cdot (\text{opcode}_E == \text{LW}_E) \cdot (\text{ws}_E \neq 0) \cdot \text{re1}_D \\ + (\text{rt}_D == \text{ws}_E) \cdot (\text{opcode}_E == \text{LW}_E) \cdot (\text{ws}_E \neq 0) \cdot \text{re2}_D$$

Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

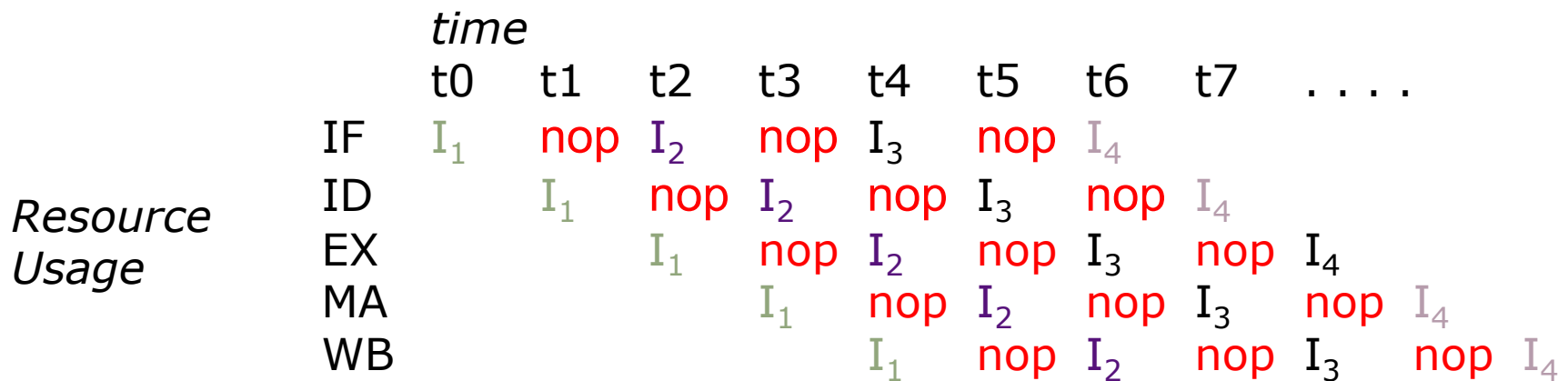
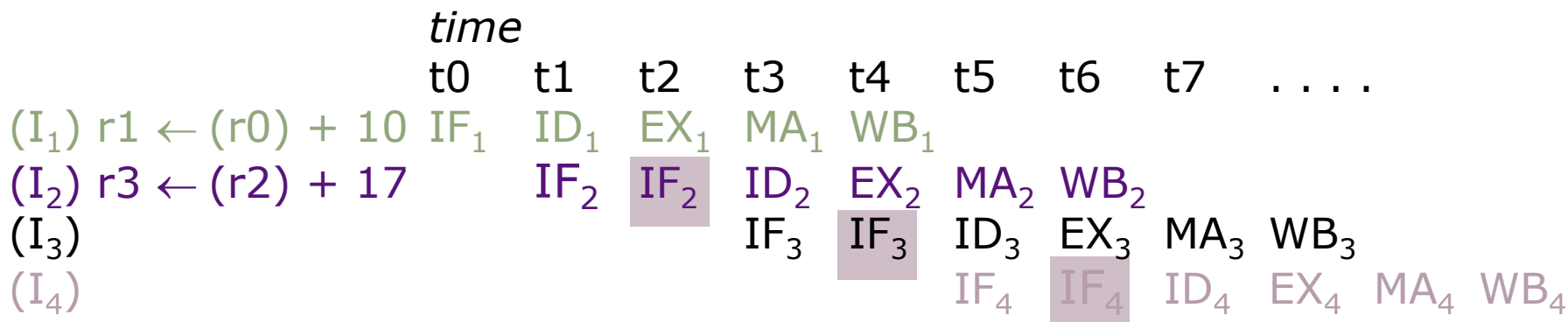
Guessed correctly → no special action required

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

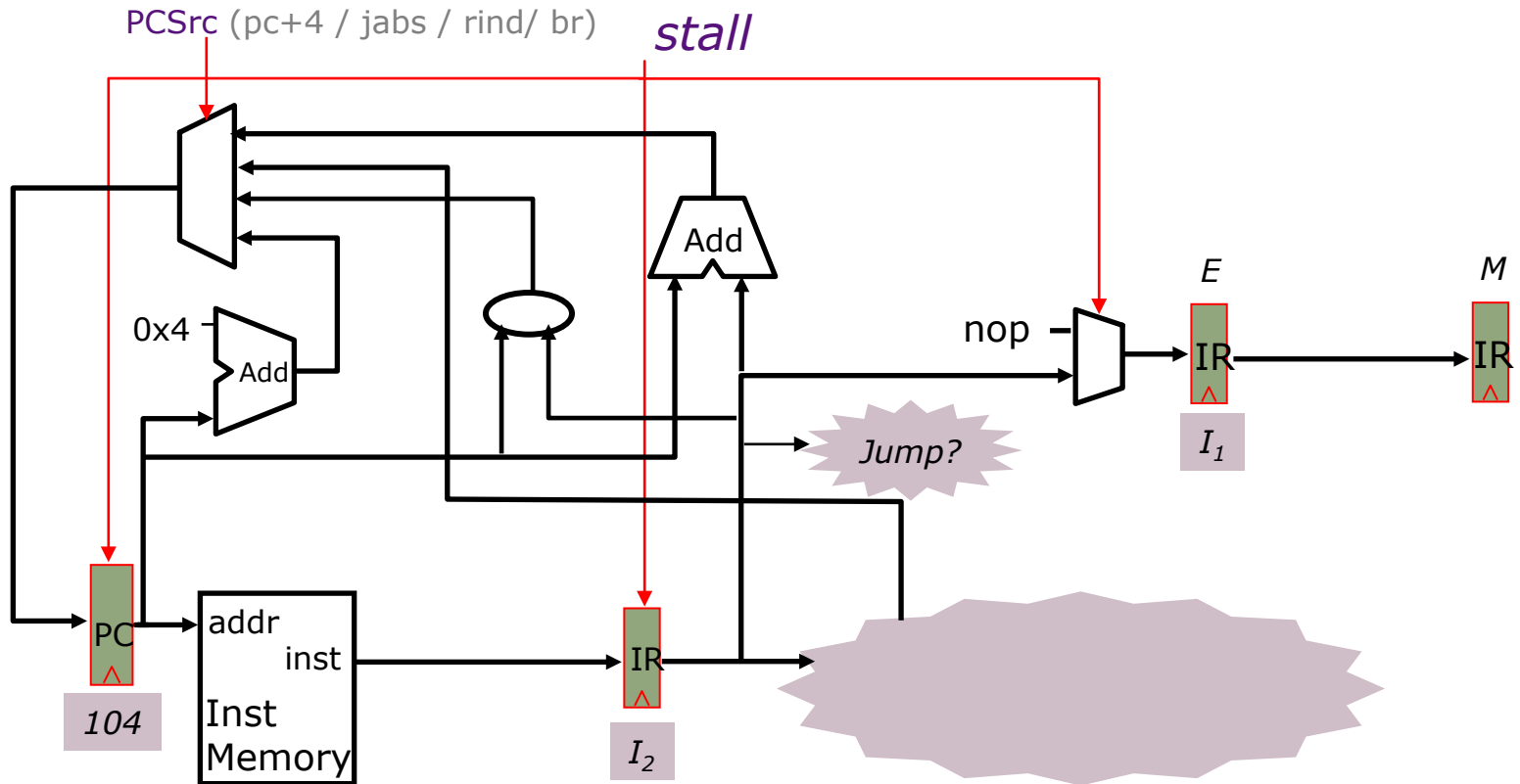


nop $\Rightarrow$ *pipeline bubble*

What's a good guess for next PC?

PC+4

Speculate NextPC is PC+4

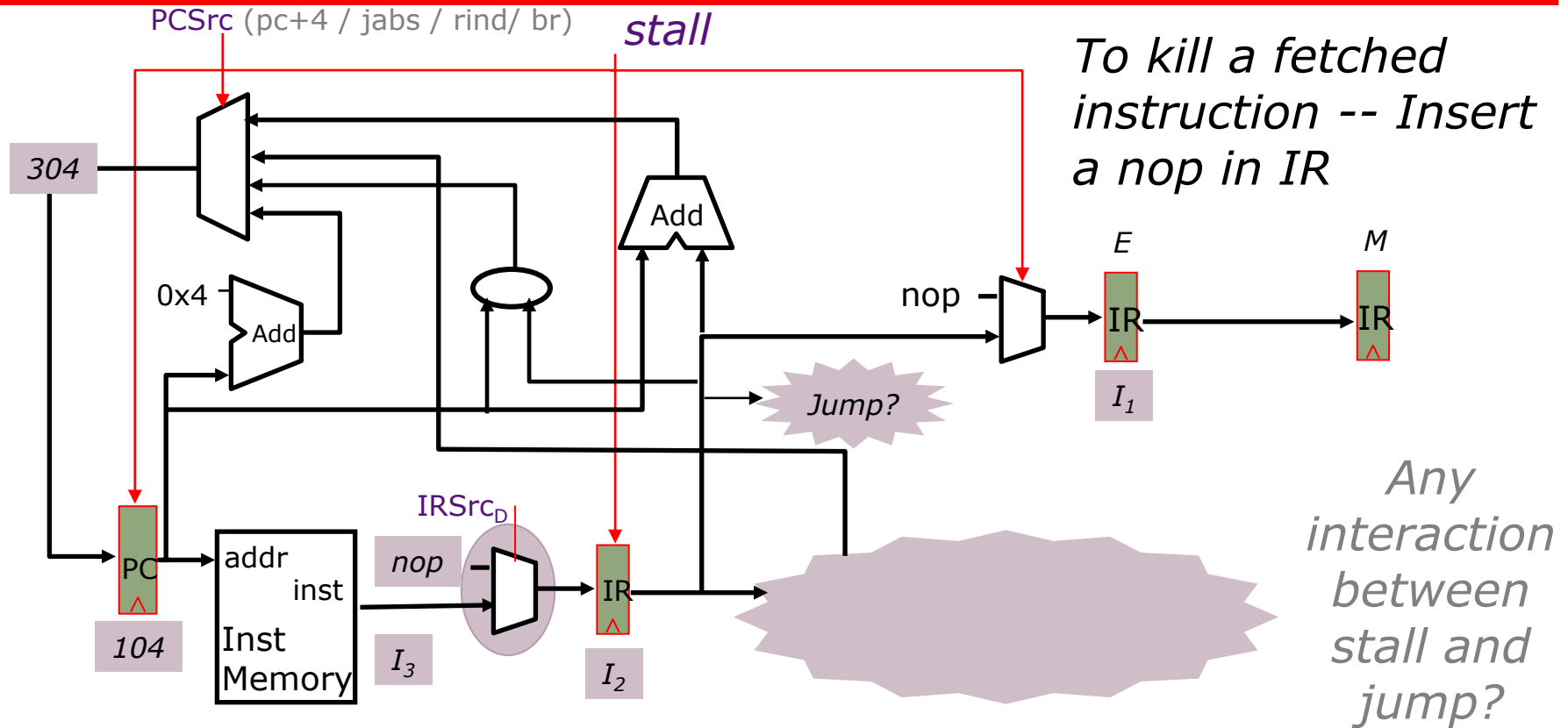


I ₁	096	ADD	
I ₂	100	J	200
I ₃	104	ADD	<i>kill</i>
I ₄	304	ADD	

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

How?

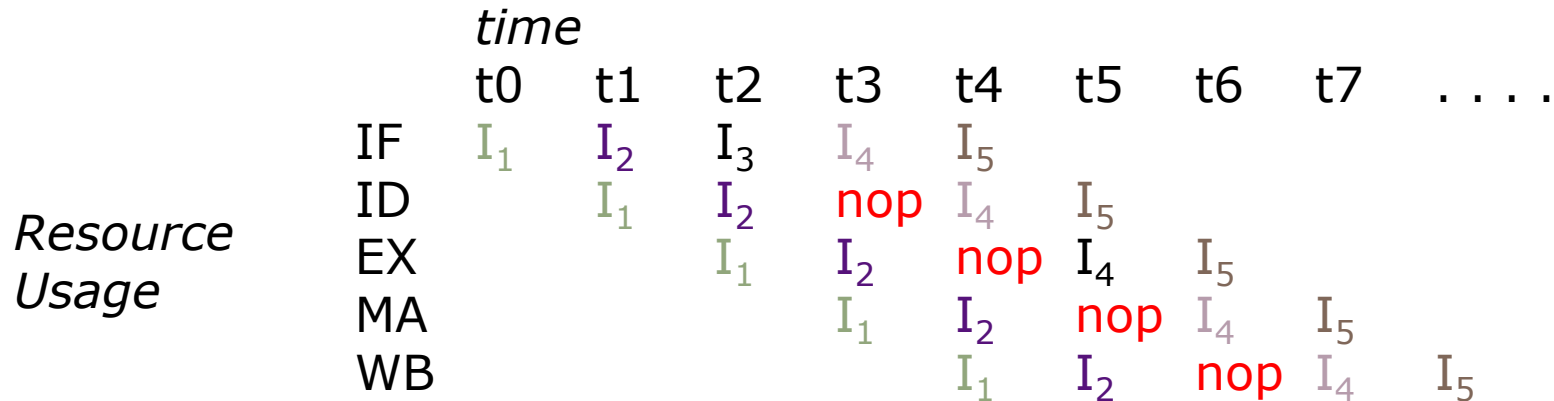
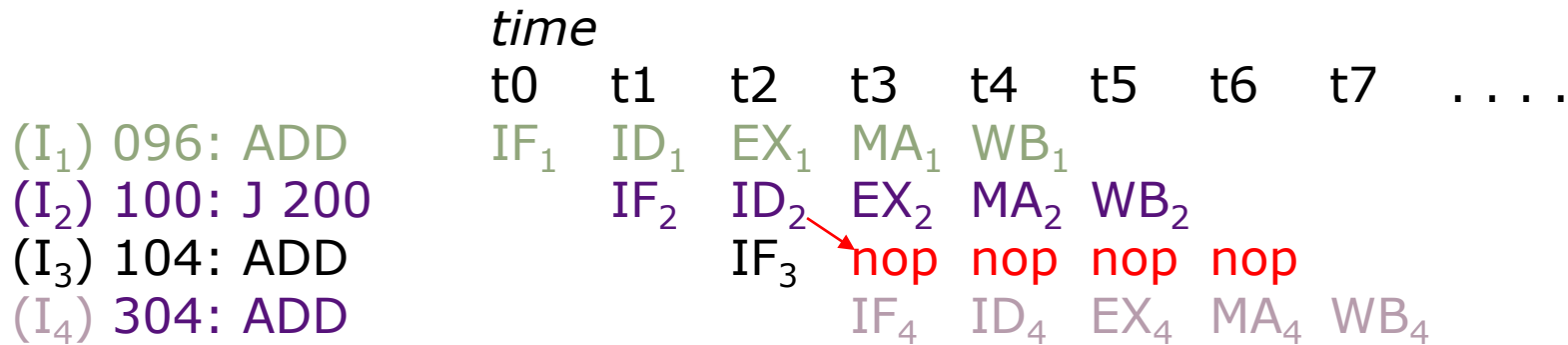
Pipelining Jumps



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

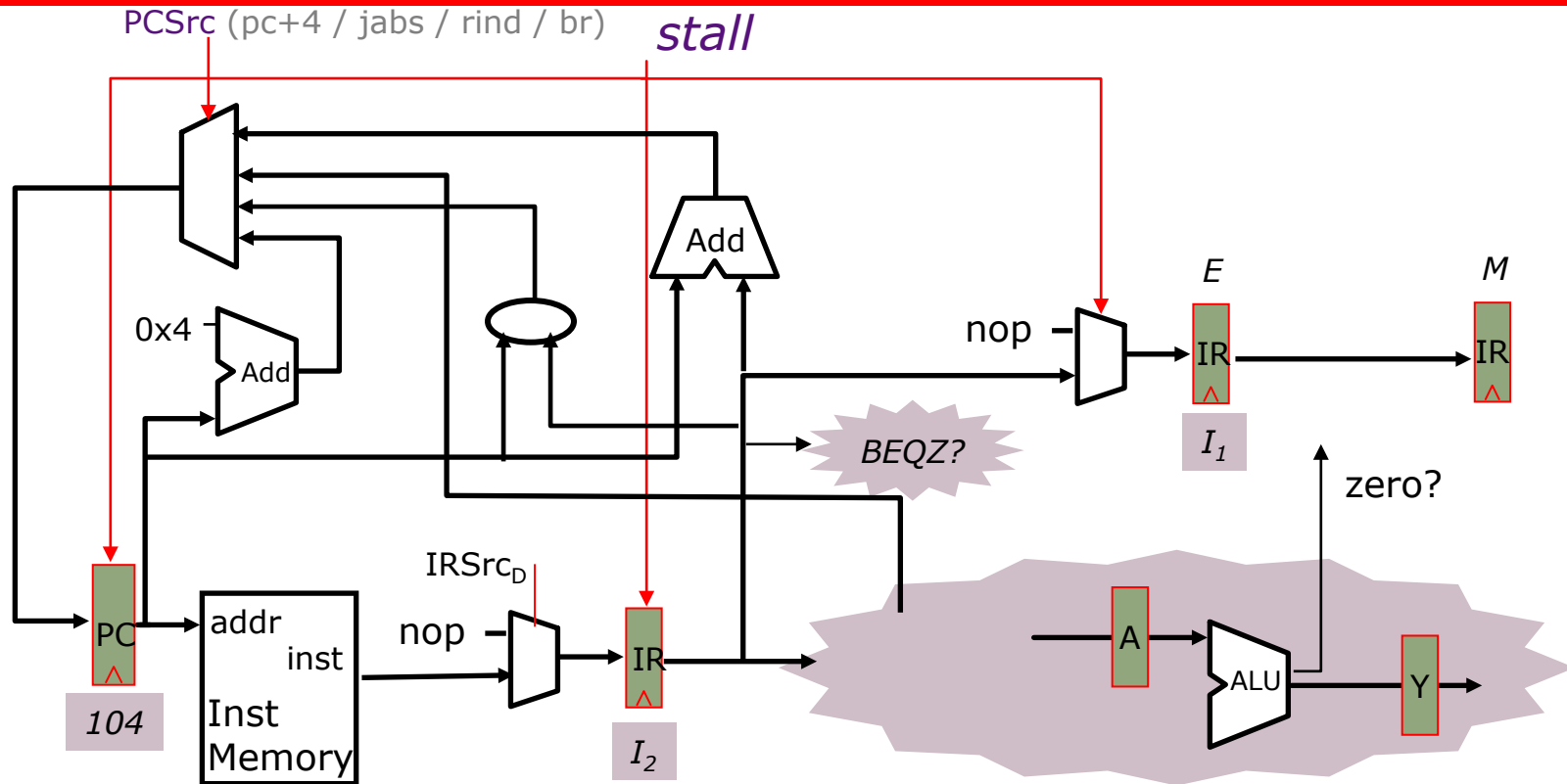
$IRSrc_D = \text{Case opcode}_D$
 J, JAL $\Rightarrow$ nop
 ... $\Rightarrow$ IM

Jump Pipeline Diagrams



nop ⇒ *pipeline bubble*

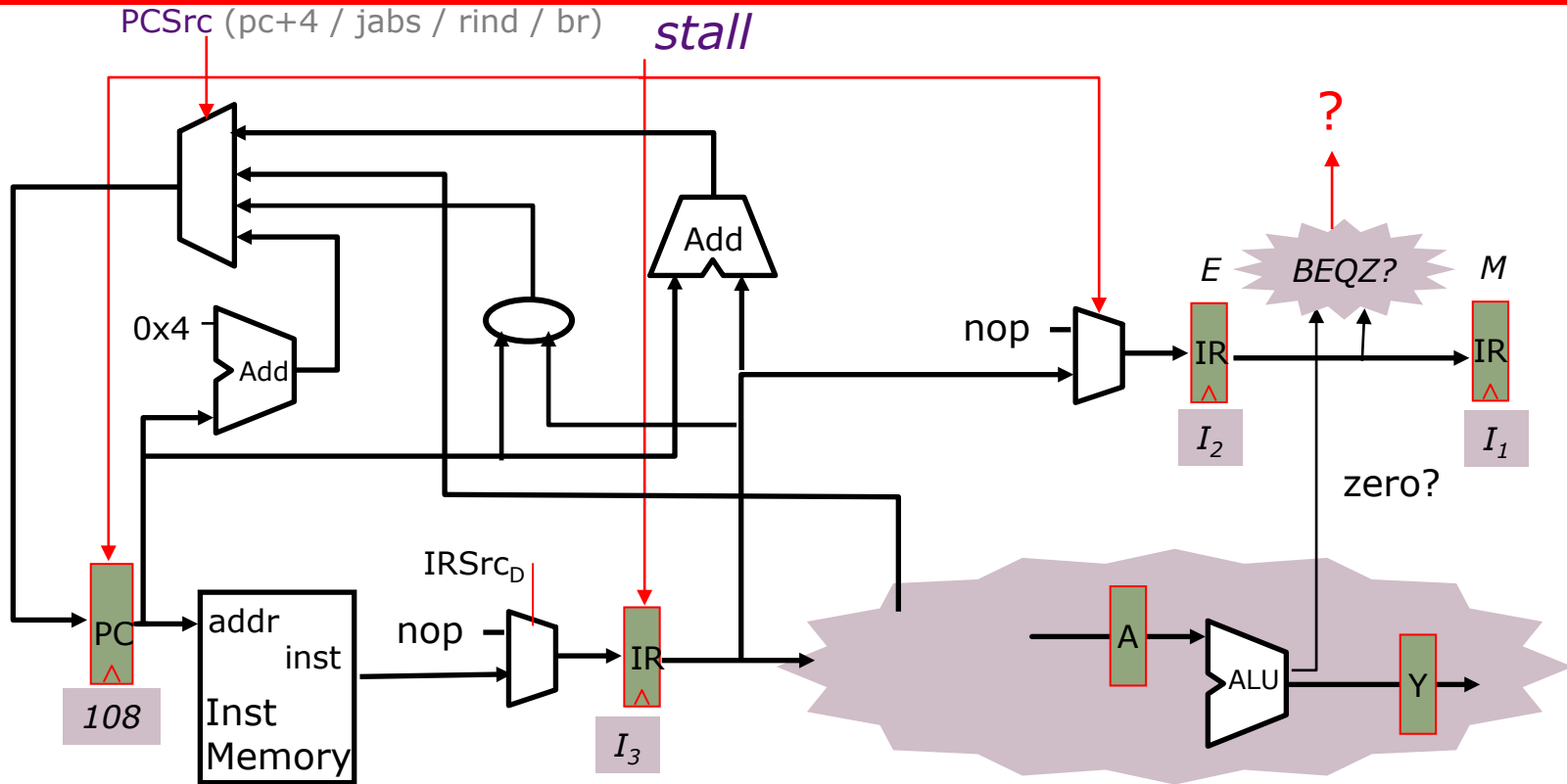
Pipelining Conditional Branches



I ₁	096	ADD
I ₂	100	BEQZ r1 200
I ₃	104	ADD
I ₄	304	ADD

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

Pipelining Conditional Branches



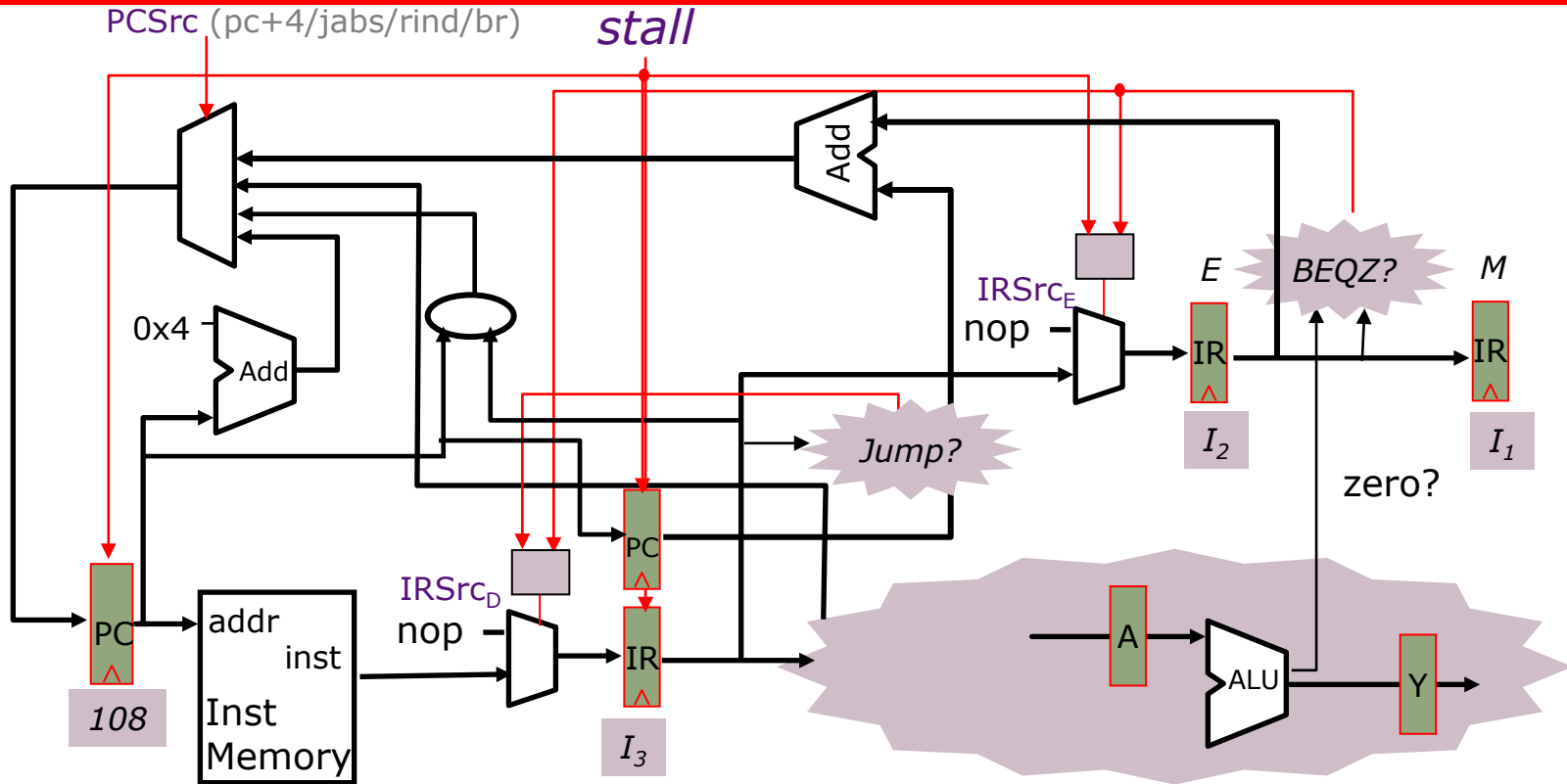
If the branch is taken

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

⇒ stall signal is not valid

I ₁	096	ADD
I ₂	100	BEQZ r1 200
I ₃	104	ADD
I ₄	304	ADD

Pipelining Conditional Branches



If the branch is taken

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

$\Rightarrow$ stall signal is not valid

I ₁	096	ADD
I ₂	100	BEQZ r1 200
I ₃	104	ADD
I ₄	304	ADD

New Stall Signal

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

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

Instruction at the decode stage is invalid

Control Equations for PC and IR Muxes

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

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

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

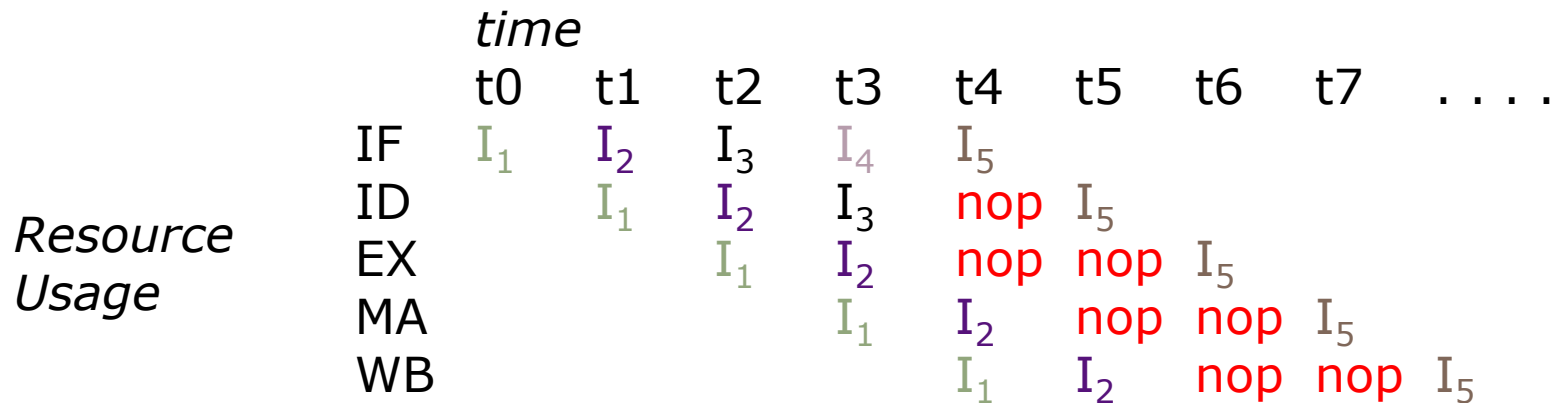
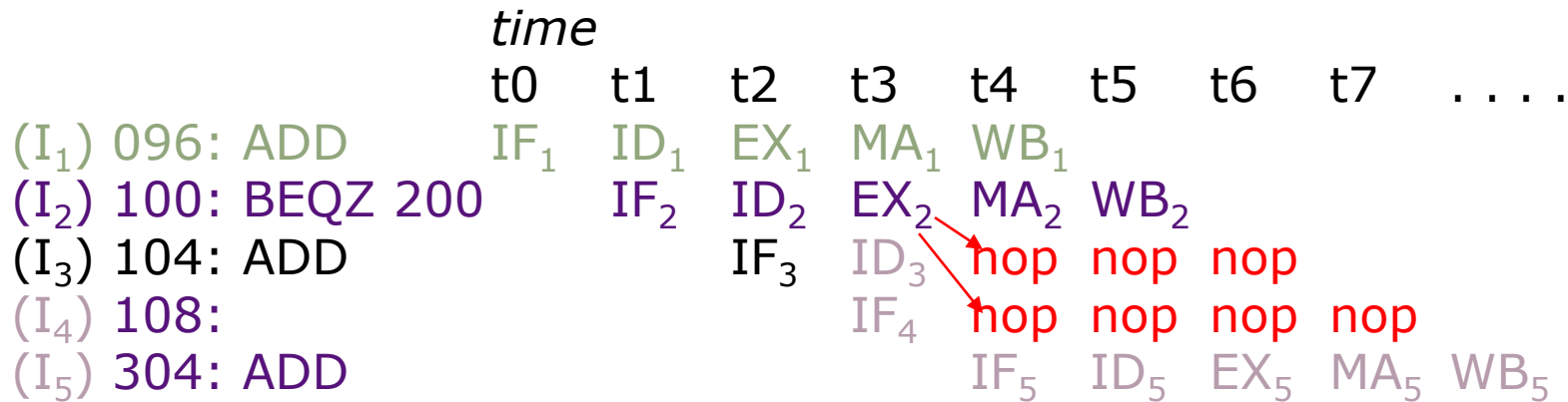
$$\begin{aligned} \text{PCSrc} &= \text{Case opcode}_E \\ &\quad \text{BEQZ}\cdot z, \text{BNEZ}\cdot !z \Rightarrow \text{br} \\ &\quad \dots \Rightarrow \\ &\quad \quad \text{Case opcode}_D \\ &\quad \quad \text{J, JAL} \Rightarrow \text{jabs} \\ &\quad \quad \text{JR, JALR} \Rightarrow \text{rind}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

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

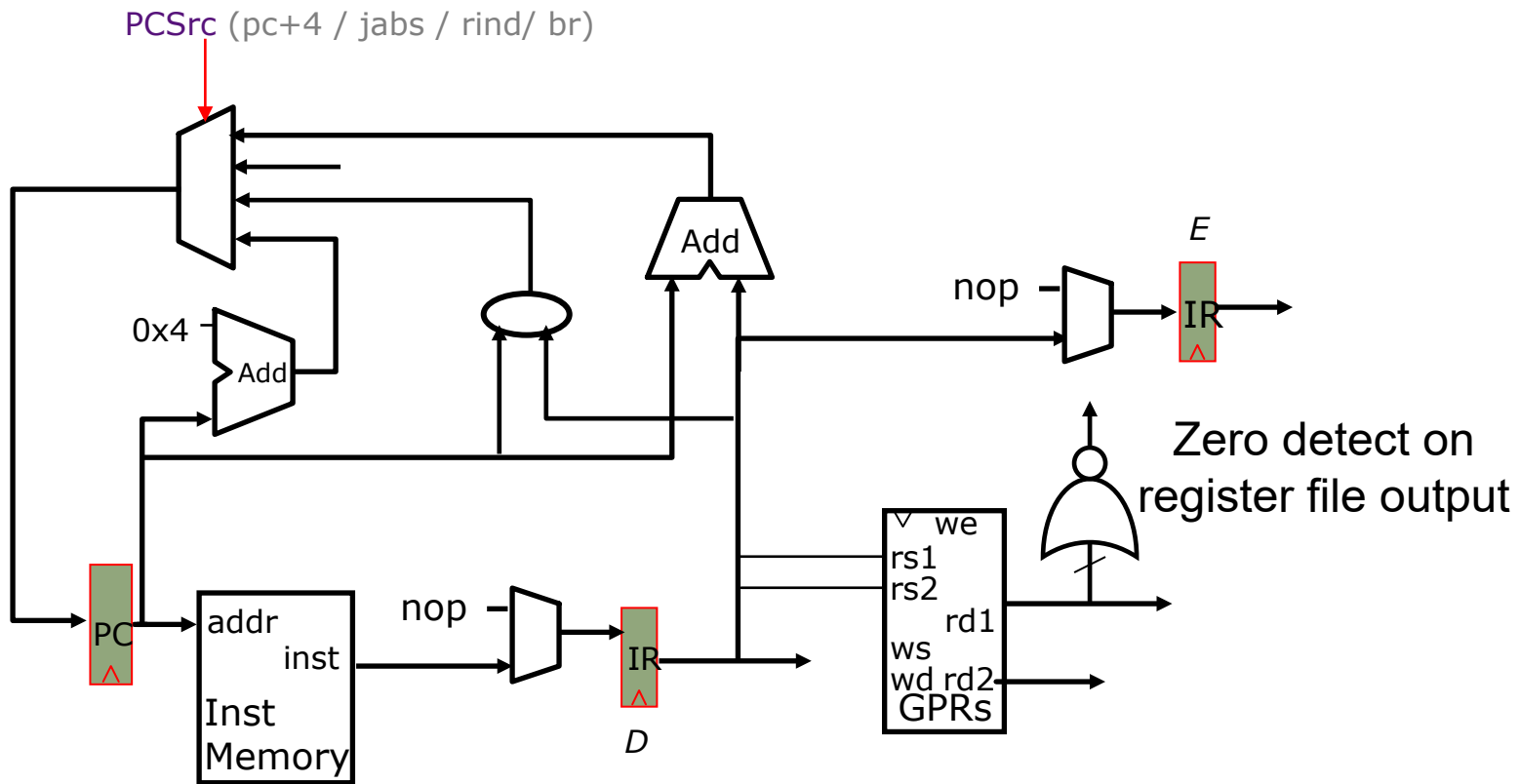
(resolved in execute stage)



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

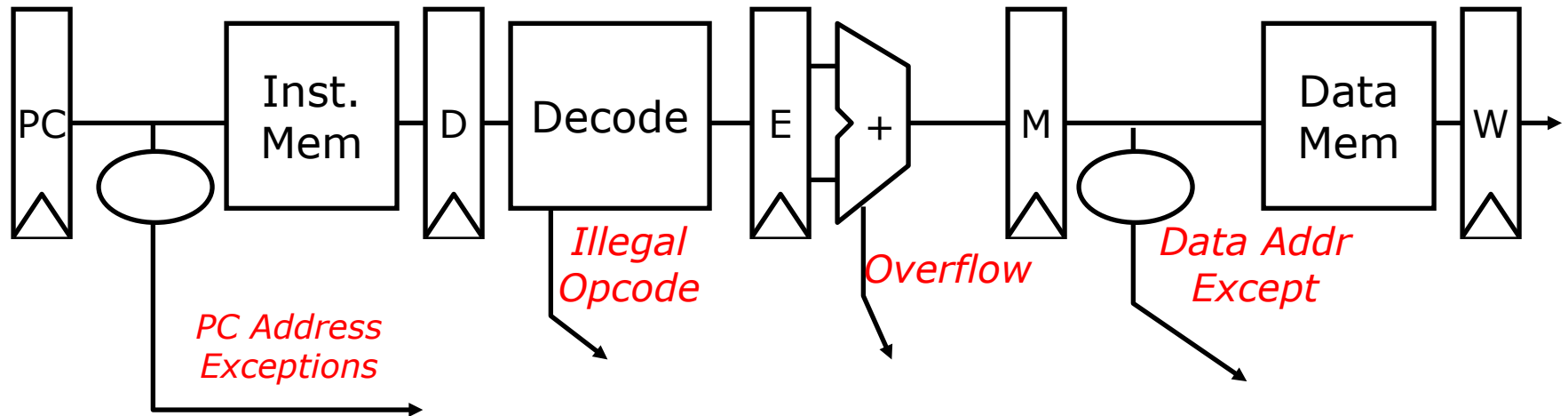
(expose control hazard to software)

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

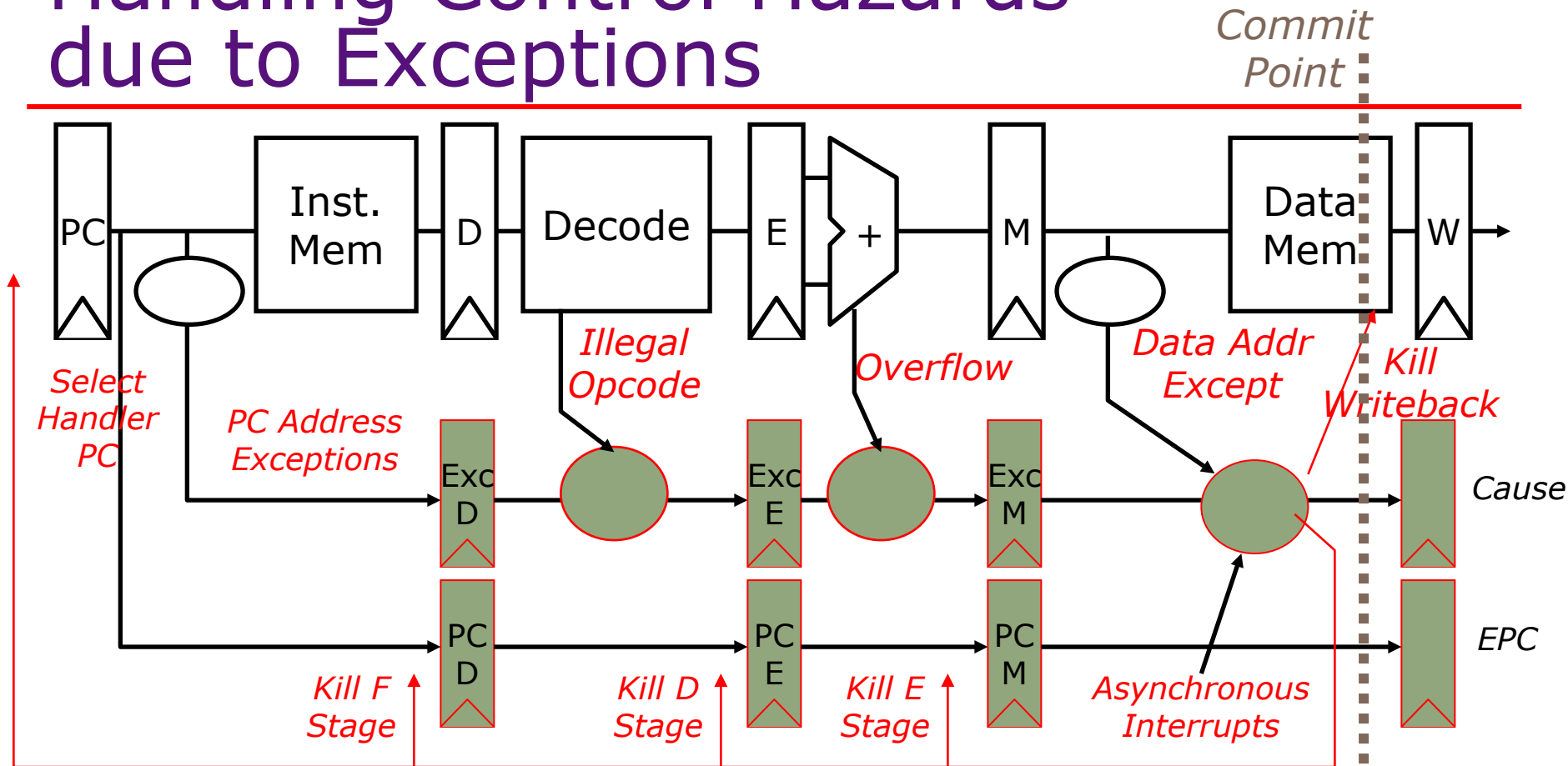
- Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*

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