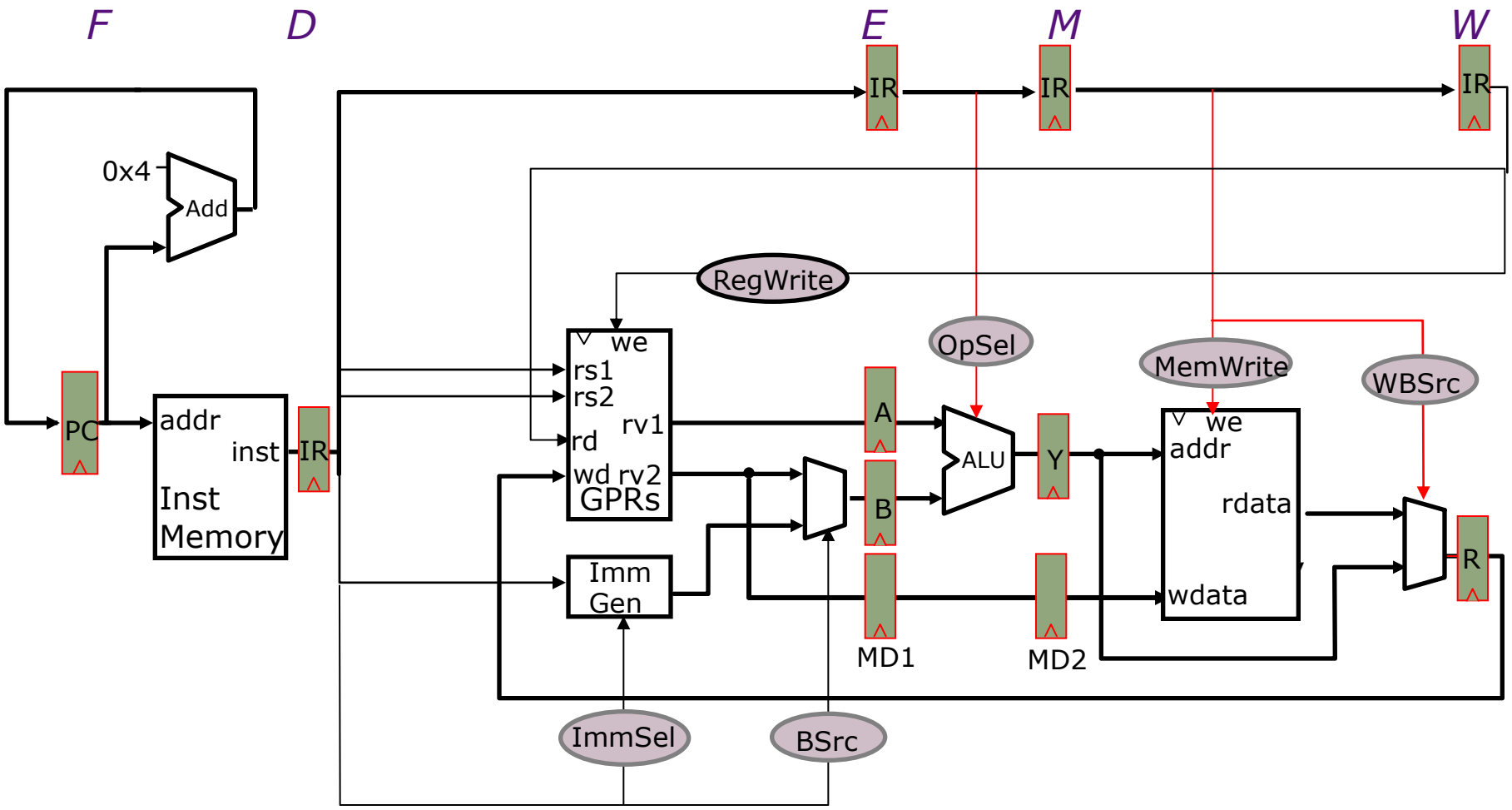
Instruction Pipelining: Hazard Resolution, Timing Constraints

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

Reminder: Pipelined RISC-V Datapath
without jumps



Pipeline Diagram – Ideal Pipelining

time	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

In steady state, what are the …

CPI? IPC? Cycles to exec each instr?

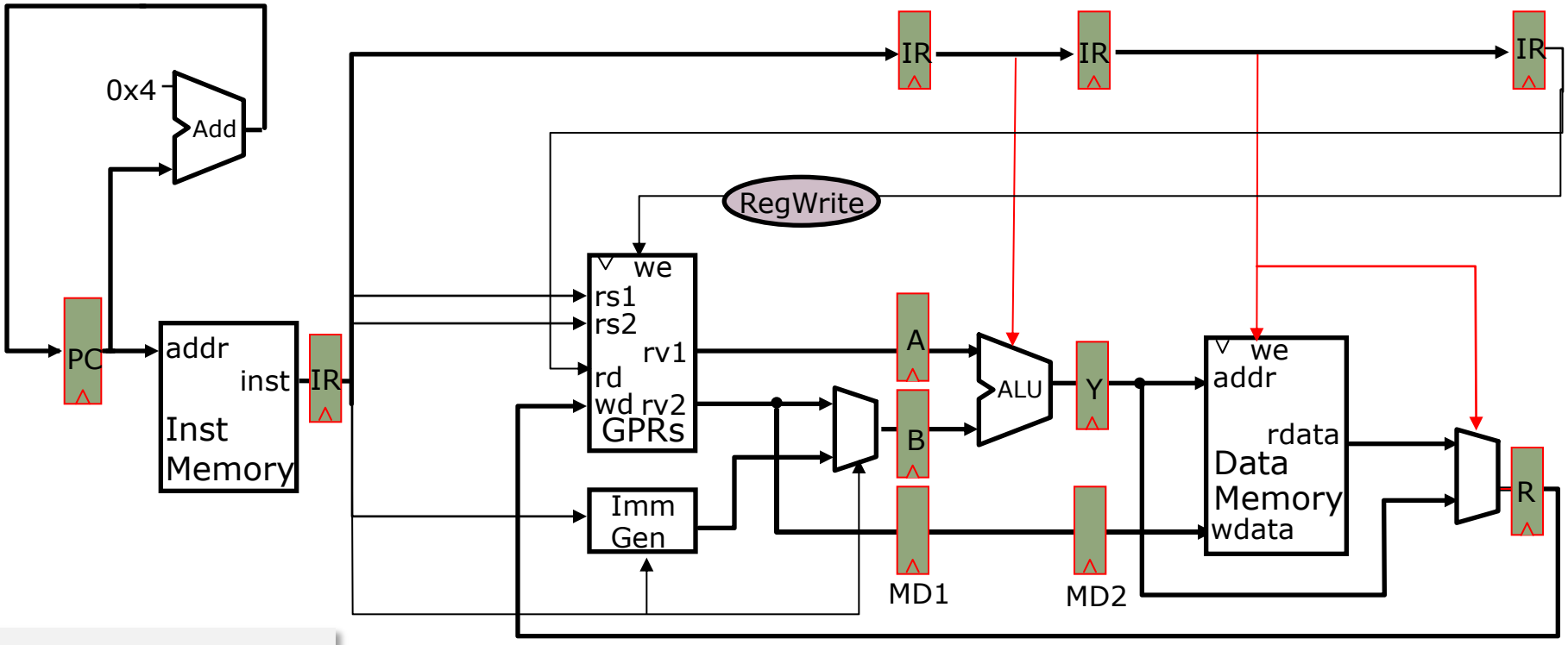
Num instrs in flight?

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



...
a1 ← a0 + 10
a3 ← a1 + 12
...

Pipeline Diagram – Hazard

time	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			

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

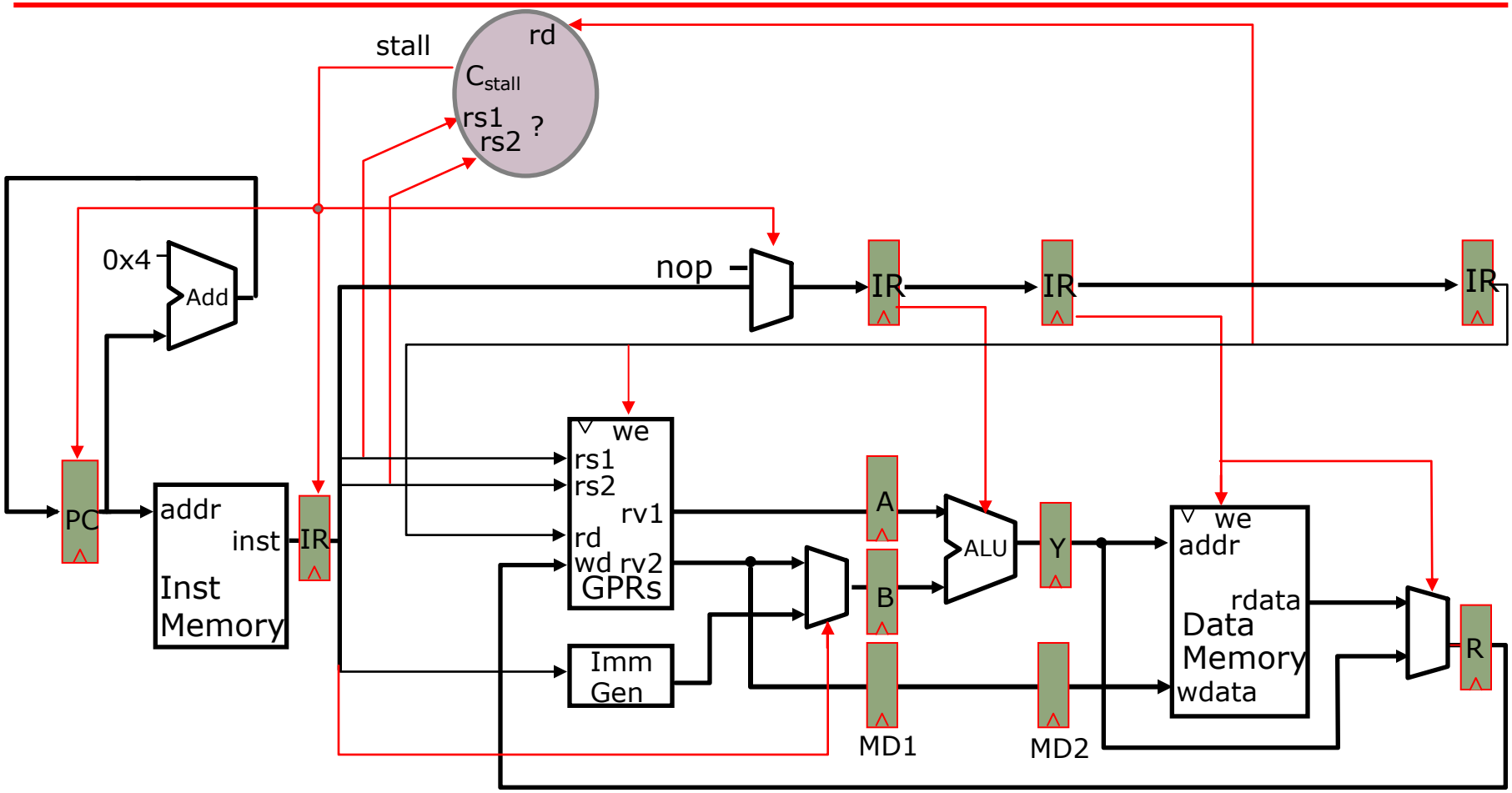
Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *stall*

time	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

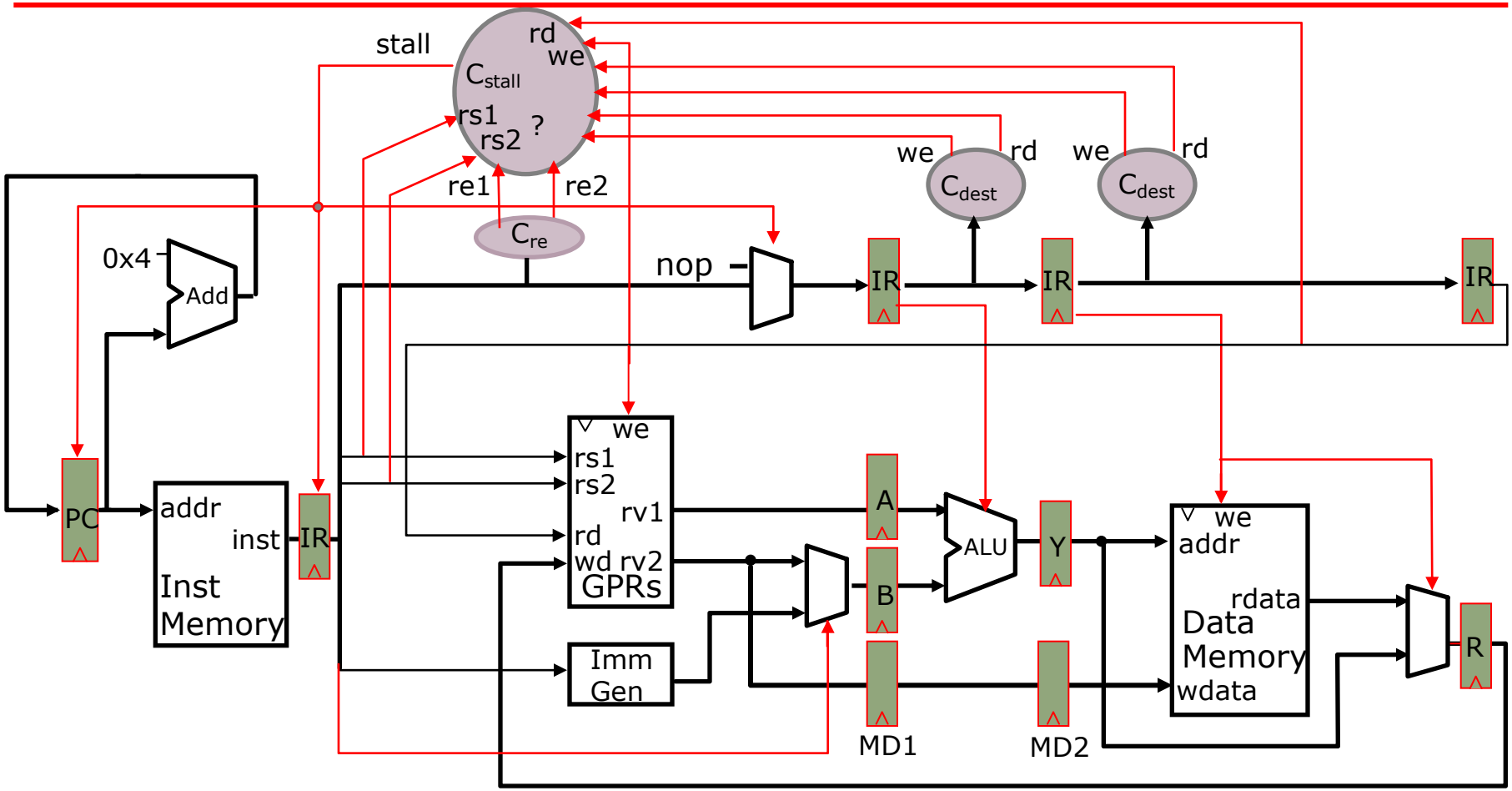
ignoring jumps & branches



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

Reminder: Stall Control Logic

ignoring jumps & branches

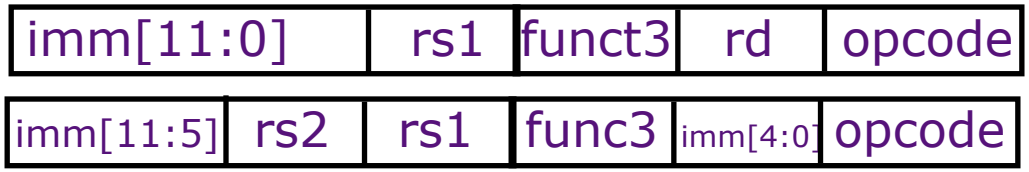


Should we always stall if the rs field(s) matches some rd?

Source & Destination Registers

R-type:

I-type:



		source(s)	*dest*
ALU	rd ← (rs1) func (rs2)	rs1, rs2	rd
ALUi	rd ← (rs1) func imm	rs1	rd
LW	rd ← M [(rs1) + imm]	rs1	rd
SW	M [(rs1) + imm] ← (rs2)	rs1, rs2	
BRANCH	*cond* (rs1) func (rs2)		
	true: PC ← (PC) + imm	rs1, rs2	
	false: PC ← (PC) + 4	rs1, rs2	
JAL	rd ← (PC+4), PC ← (PC) + imm		rd
JALR	rd ← (PC+4), PC ← (rs1) + imm*	rs1	rd

**More precisely: pc ← {(reg[rs1] + imm)[31:1], 1'b0}*

Deriving the Stall Signal

C_{dest}

we = *Case* opcode

ALU, ALUi, LW, JAL, JALR $\Rightarrow (rd \neq 0)$

... $\Rightarrow$ off

C_{re}

re1 = *Case* opcode

ALU, ALUi, LW, SW, BRANCH, JR, JALR $\Rightarrow$ on

re2 = *Case* opcode

ALU, SW, BRANCH $\Rightarrow$ on

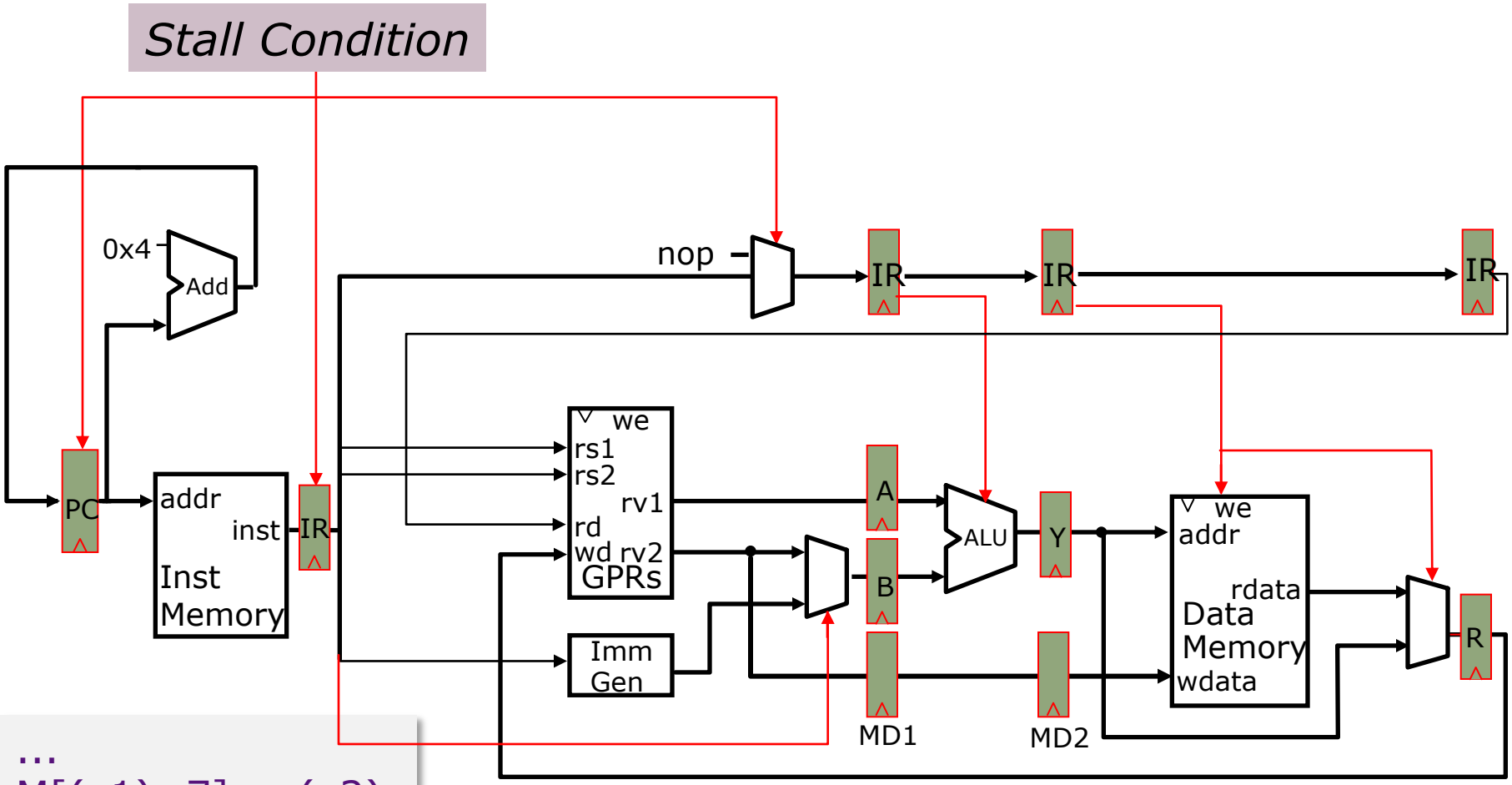
... $\Rightarrow$ off

C_{stall}

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

This is not the full story !

Hazards due to Loads & Stores




```
...
M[(a1)+7] ← (a2)
a4 ← M[(a3)+5]
...
```

Is there any possible data hazard in this instruction sequence?

Load & Store Hazards

```
...
M[(a1)+7] ← (a2)
a4 ← M[(a3)+5]
...
```

(a1)+7 = (a3)+5 ⇒ *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.

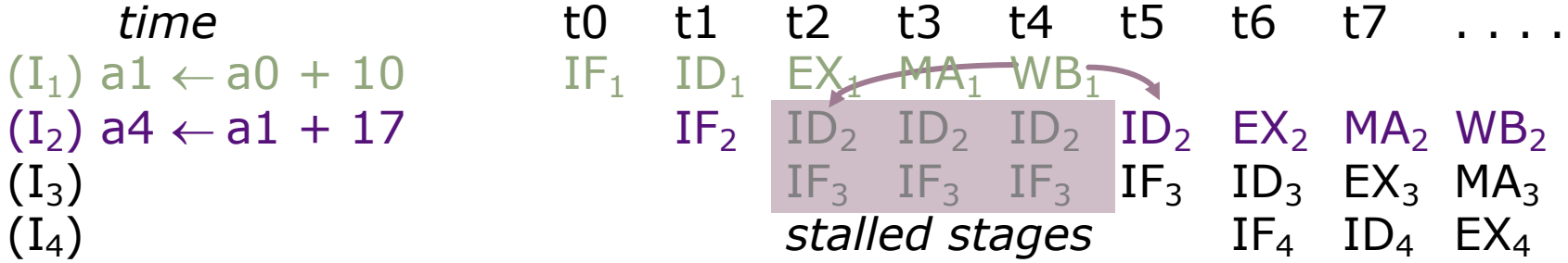
More on this later in the course.

Resolving Data Hazards (2)

Strategy 2:

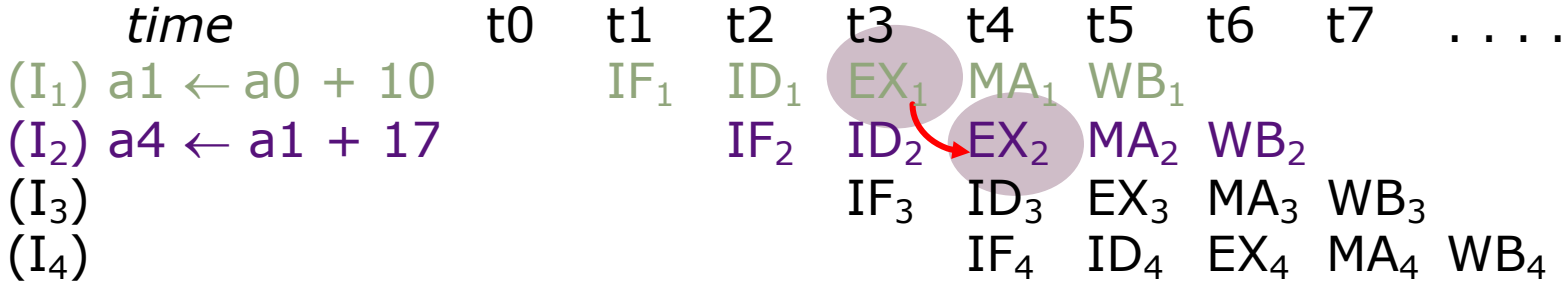
Route data as soon as possible after it is calculated to the earlier pipeline stage → *bypass*

Bypassing



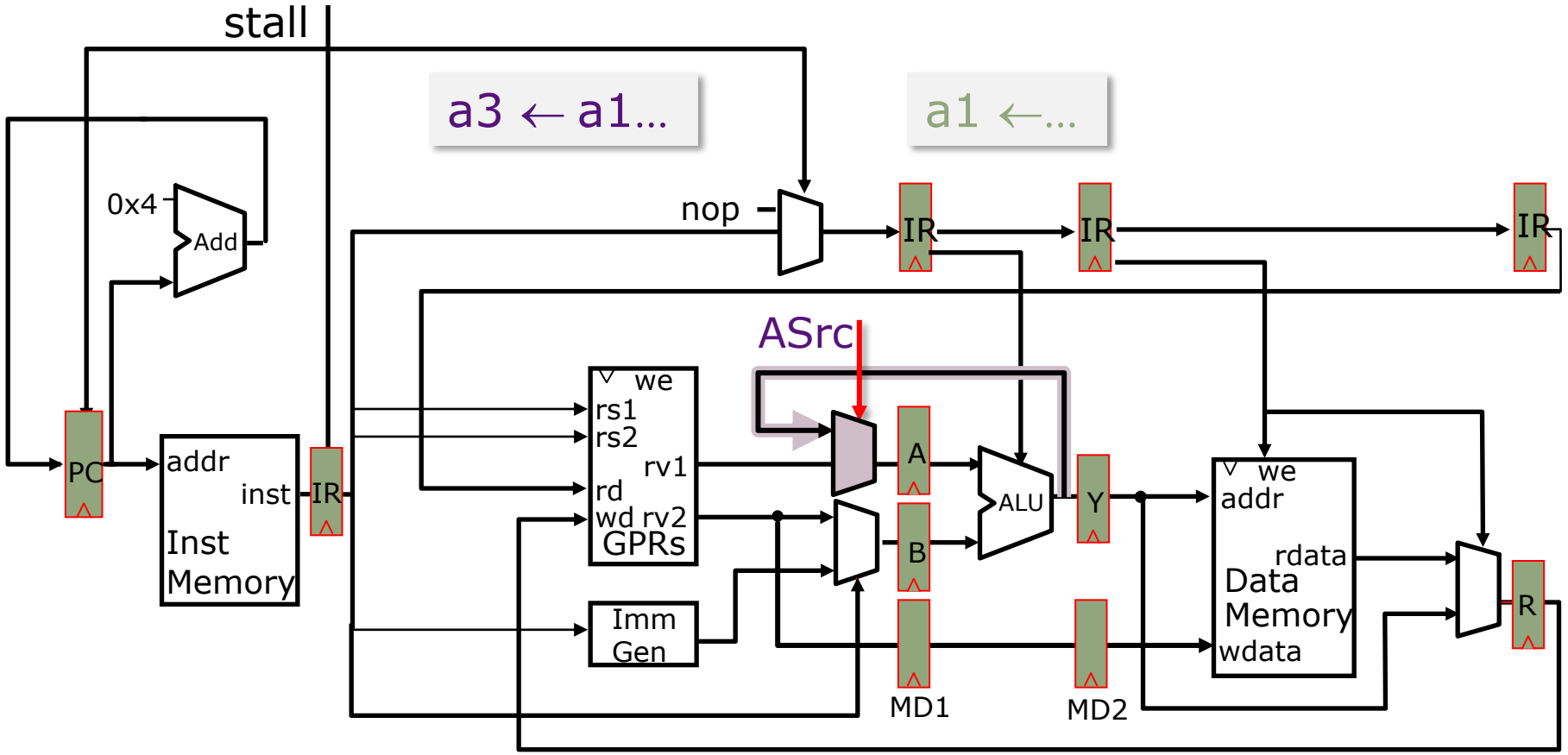
Each *stall* introduces a bubble $\Rightarrow$ *CPI* > *1*

When is data actually available?



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 <u>this</u> bypass help?

(I_1) a1 ← a0 + 10 (I_2) a3 ← a1 + 12	*yes*	a1 ← M[a0 + 10] a3 ← a1 + 12	*no*	JAL ra 500 t3 ← ra + 12	*no*

The Bypass Signal

Deriving it from the Stall Signal

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

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

$$ASrc = (rs1_D == rd_E)\cdot we_E \cdot re1_D$$

Is this correct?

How might we address this?

Bypass and Stall Signals

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

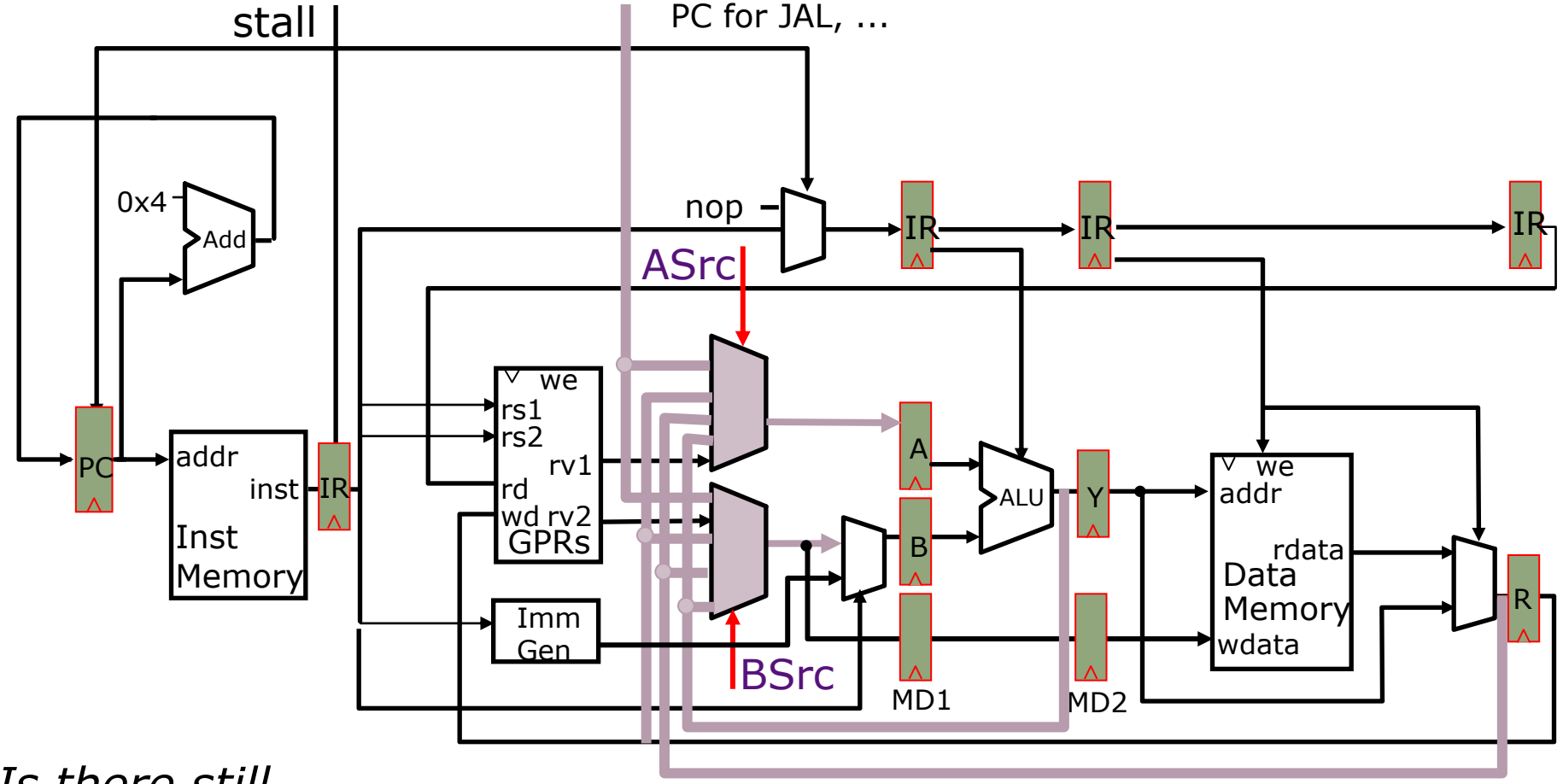
$we\text{-}bypass_E$ = *Case* $opcode_E$
ALU, ALUi $\Rightarrow (rd \neq 0)$
... $\Rightarrow$ off

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

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

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

Fully Bypassed Datapath



Is there still a need for the stall signal?

$$\text{stall} = (rs1_D == rd_E)\cdot(opcode_E == LW_E)\cdot(rd_E \neq 0)\cdot re1_D + (rs2_D == rd_E)\cdot(opcode_E == LW_E)\cdot(rd_E \neq 0)\cdot re2_D$$

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 (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

	time 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) + 17		IF_2	IF_2	ID_2	EX_2	MA_2	WB_2		
(I_3)				IF_3	IF_3	ID_3	EX_3	MA_3	WB_3
(I_4)						IF_4	IF_4	ID_4	EX_4 MA_4 WB_4

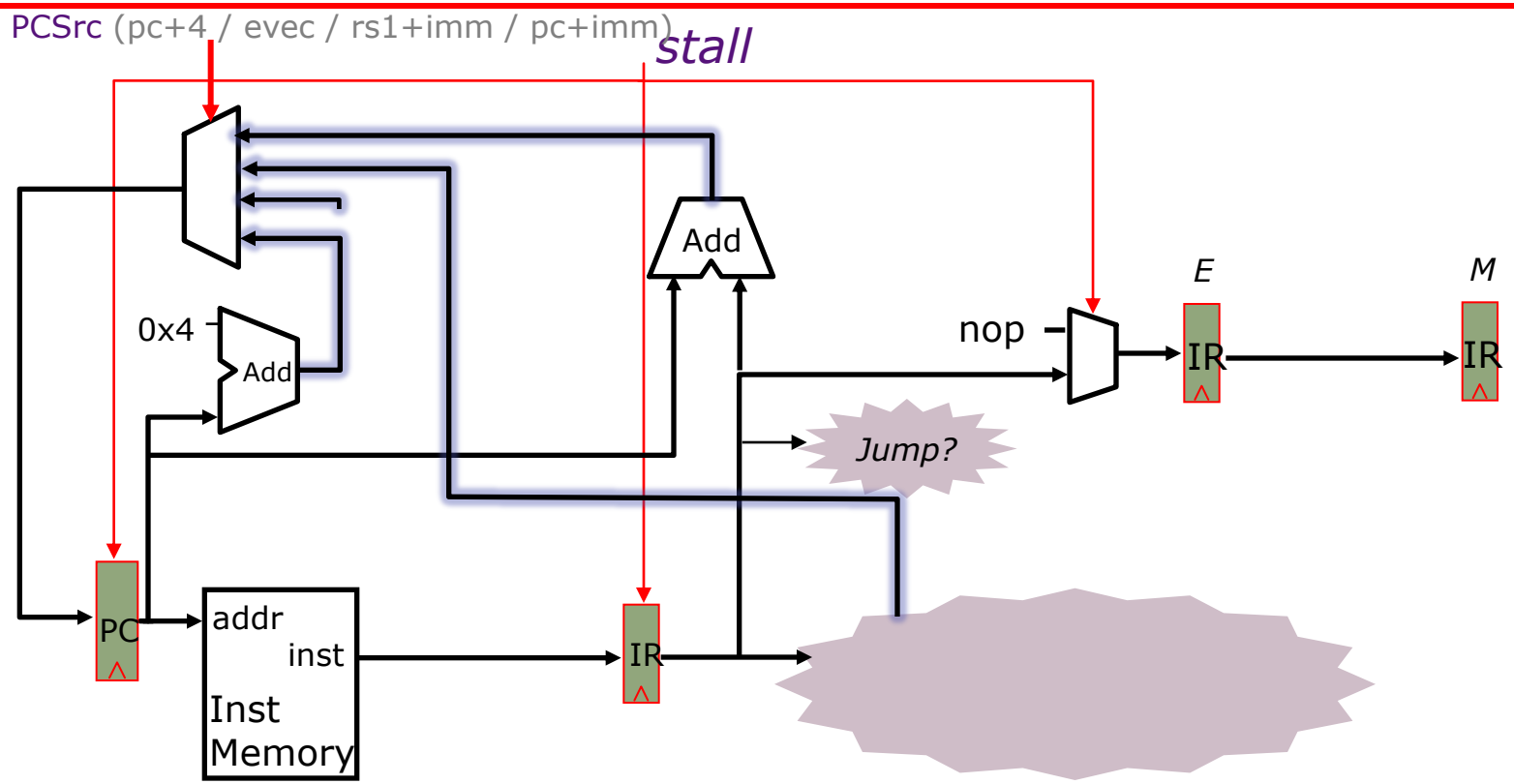
Resource Usage Diagram

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	nop	I_2	nop	I_3	nop	I_4		
ID		I_1	nop	I_2	nop	I_3	nop	I_4	
EX			I_1	nop	I_2	nop	I_3	nop	I_4
MA				I_1	nop	I_2	nop	I_3	nop I_4
WB					I_1	nop	I_2	nop	I_3 nop I_4

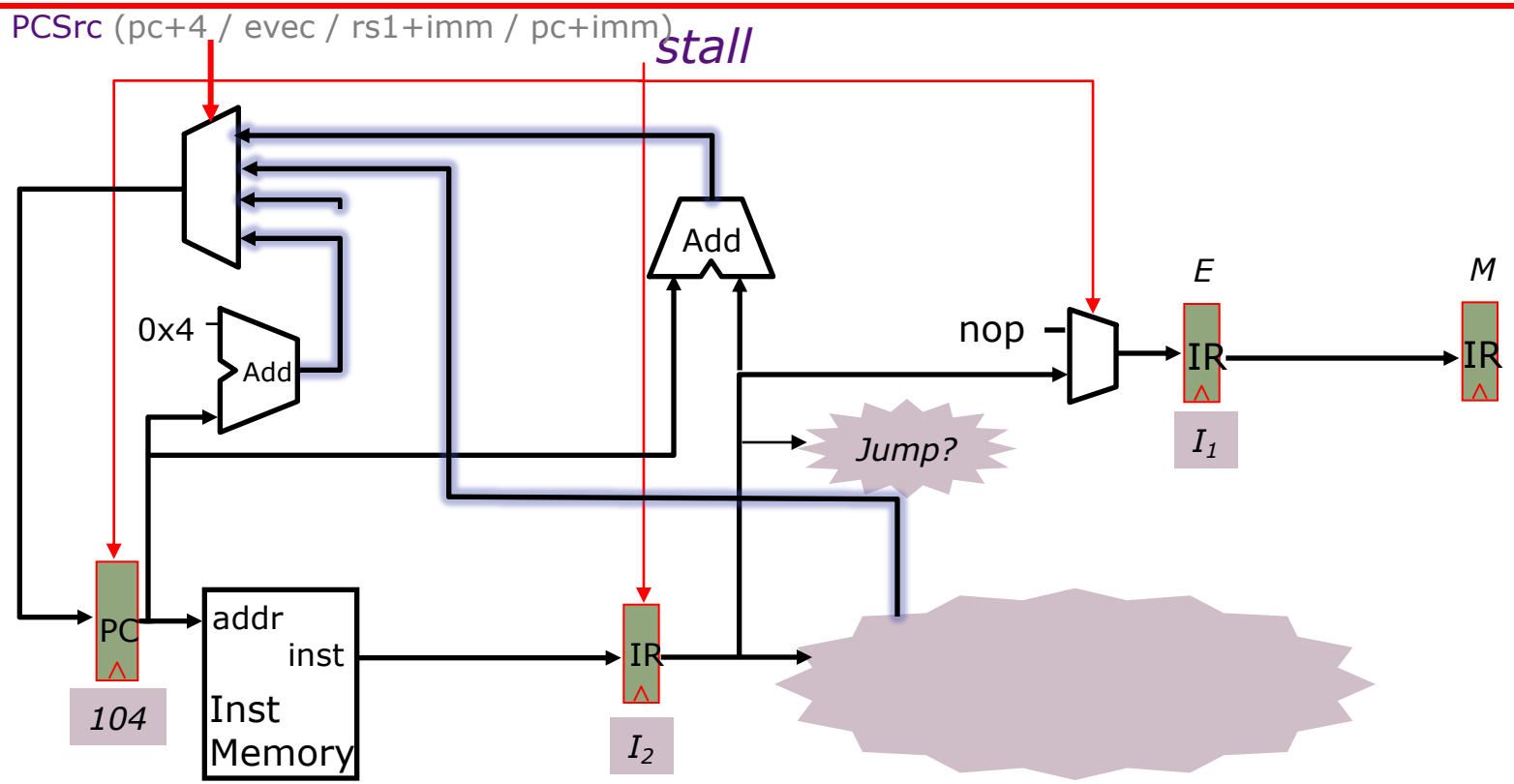
nop ⇒ pipeline bubble

What's a good guess for next PC?

Speculate NextPC is PC+4



Speculate NextPC is PC+4

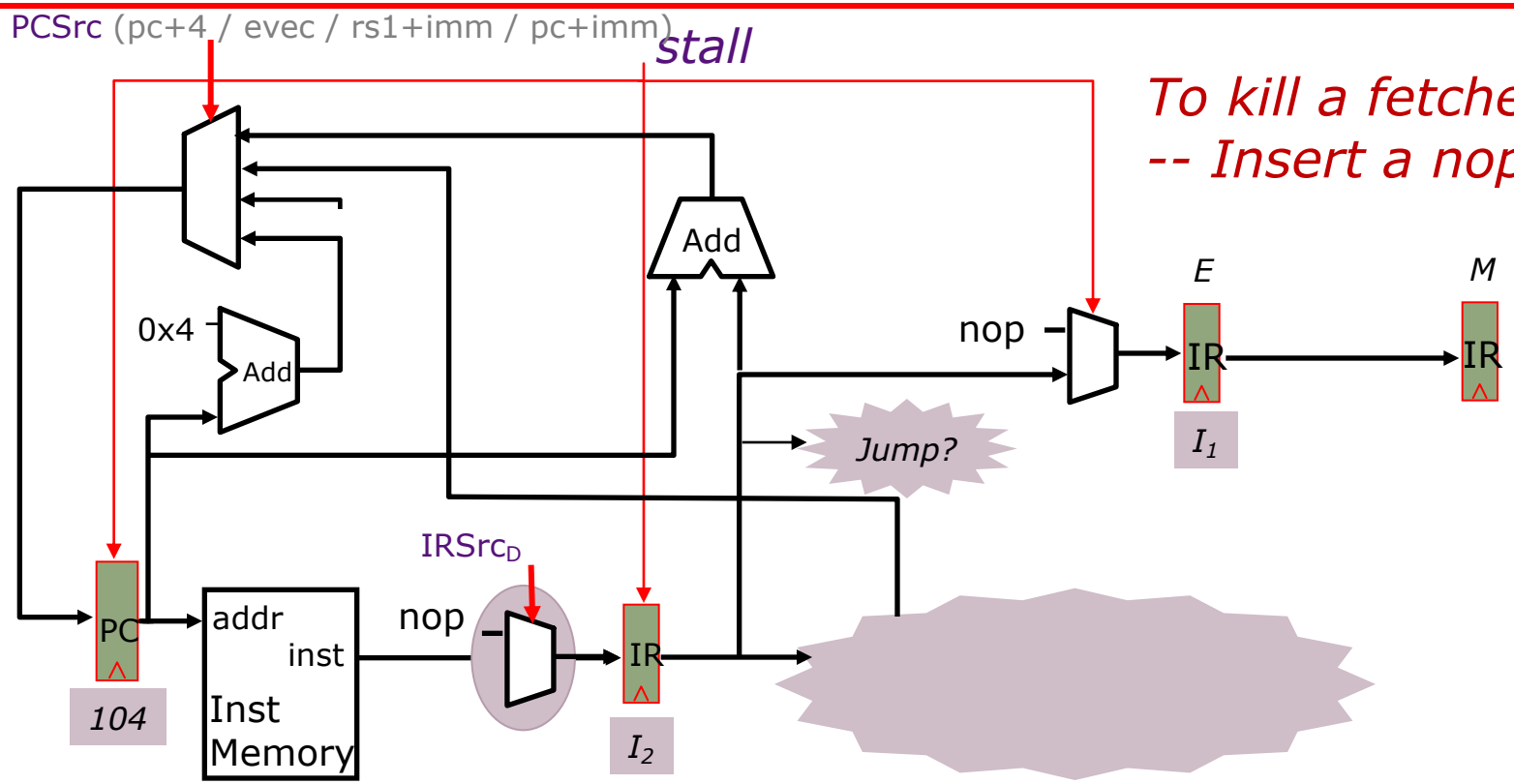


I_1	096	ADD
I_2	100	JAL ra 200
I_3	~~104~~	~~ADD~~ *kill*
I_4	304	ADD

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

How?

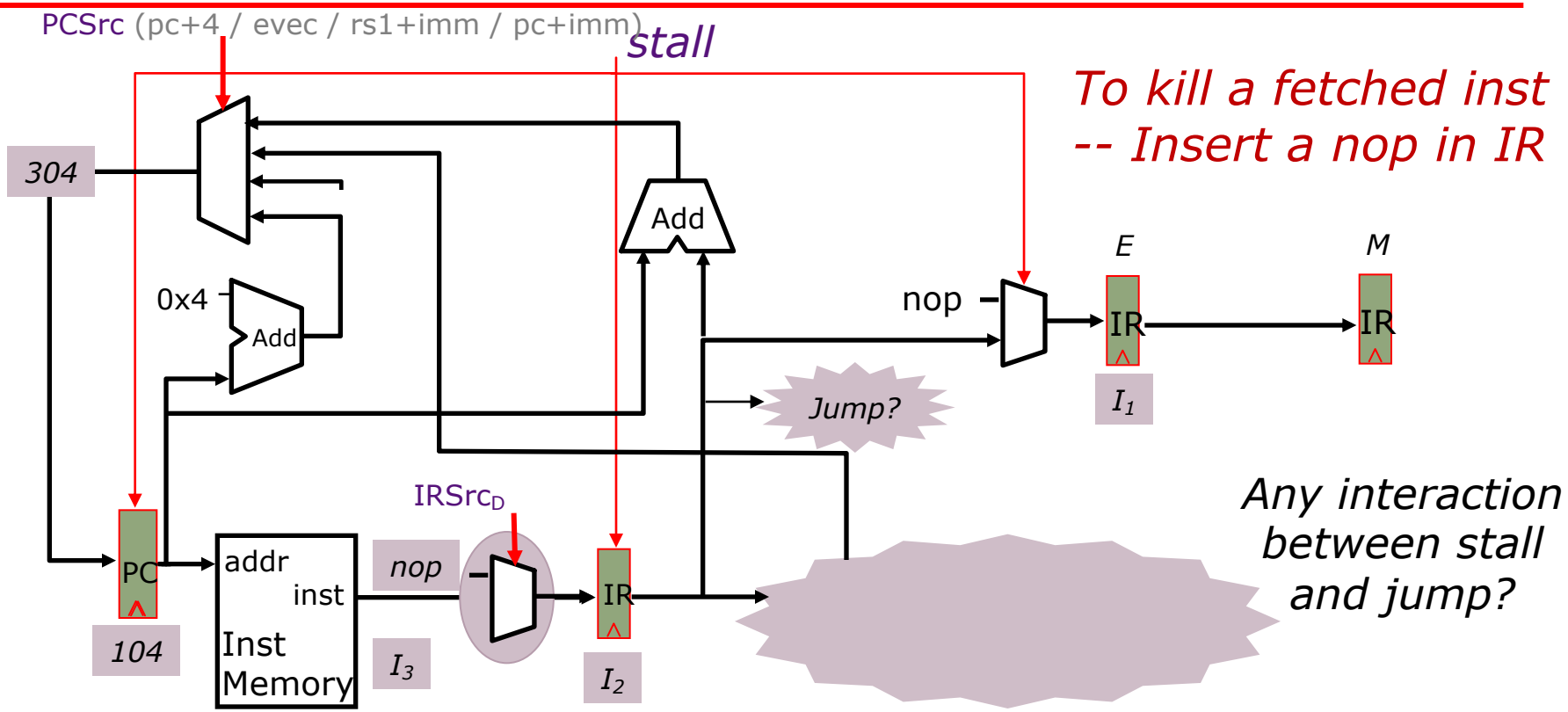
Pipelining Jumps



I_1	096	ADD
I_2	100	JAL ra 200
I_3	~~104~~	~~ADD~~ *kill*
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

$IRSrc_D$ = *Case* $opcode_D$
JAL, JALR $\Rightarrow$ nop
... $\Rightarrow$ IM

Jump Pipeline Diagrams

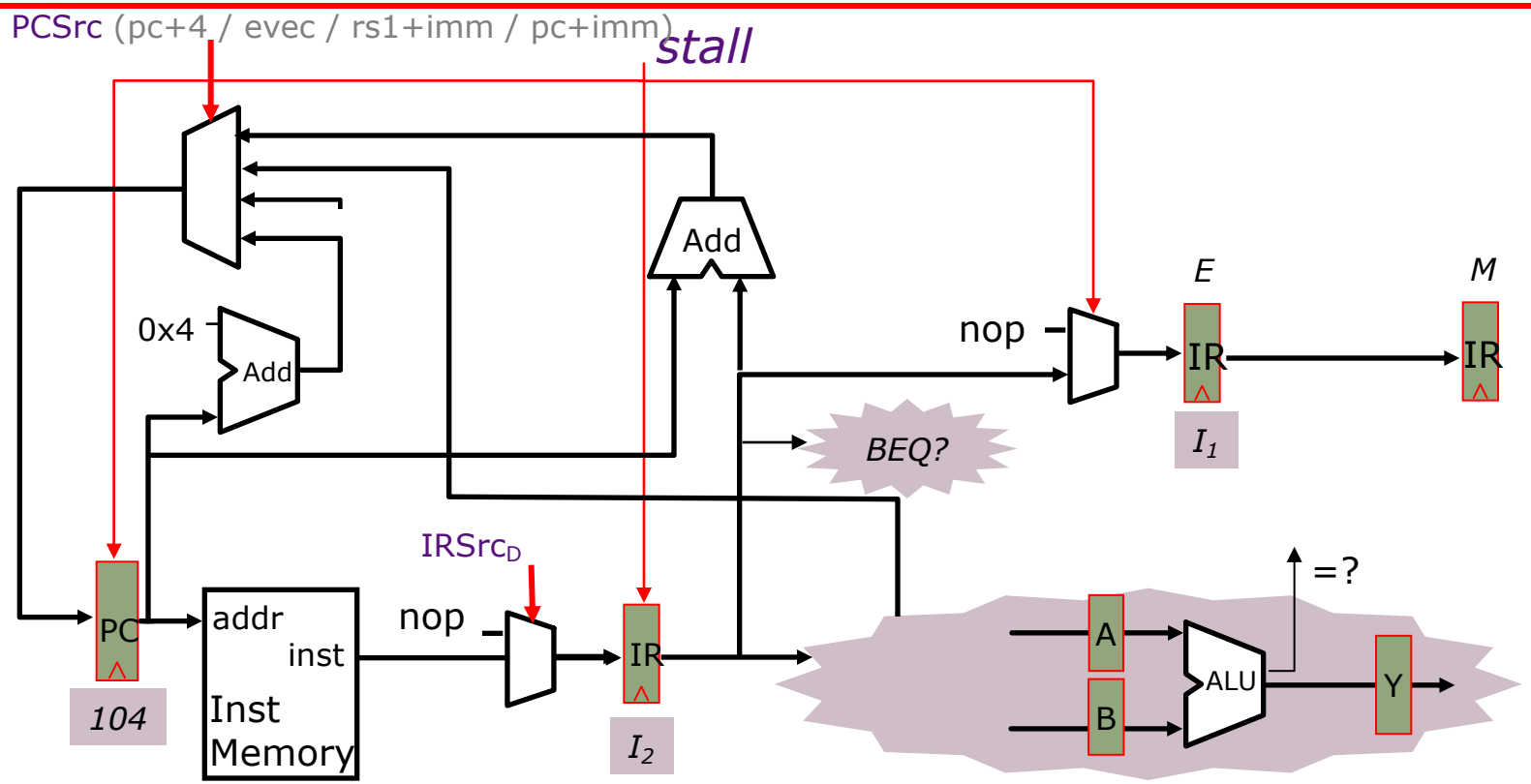
	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: JAL ra 200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	nop	nop	nop	nop		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

Resource Usage Diagram

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4	I_5				
ID		I_1	I_2	nop	I_4	I_5			
EX			I_1	I_2	nop	I_4	I_5		
MA				I_1	I_2	nop	I_4	I_5	
WB					I_1	I_2	nop	I_4	I_5

nop ⇒ *pipeline bubble*

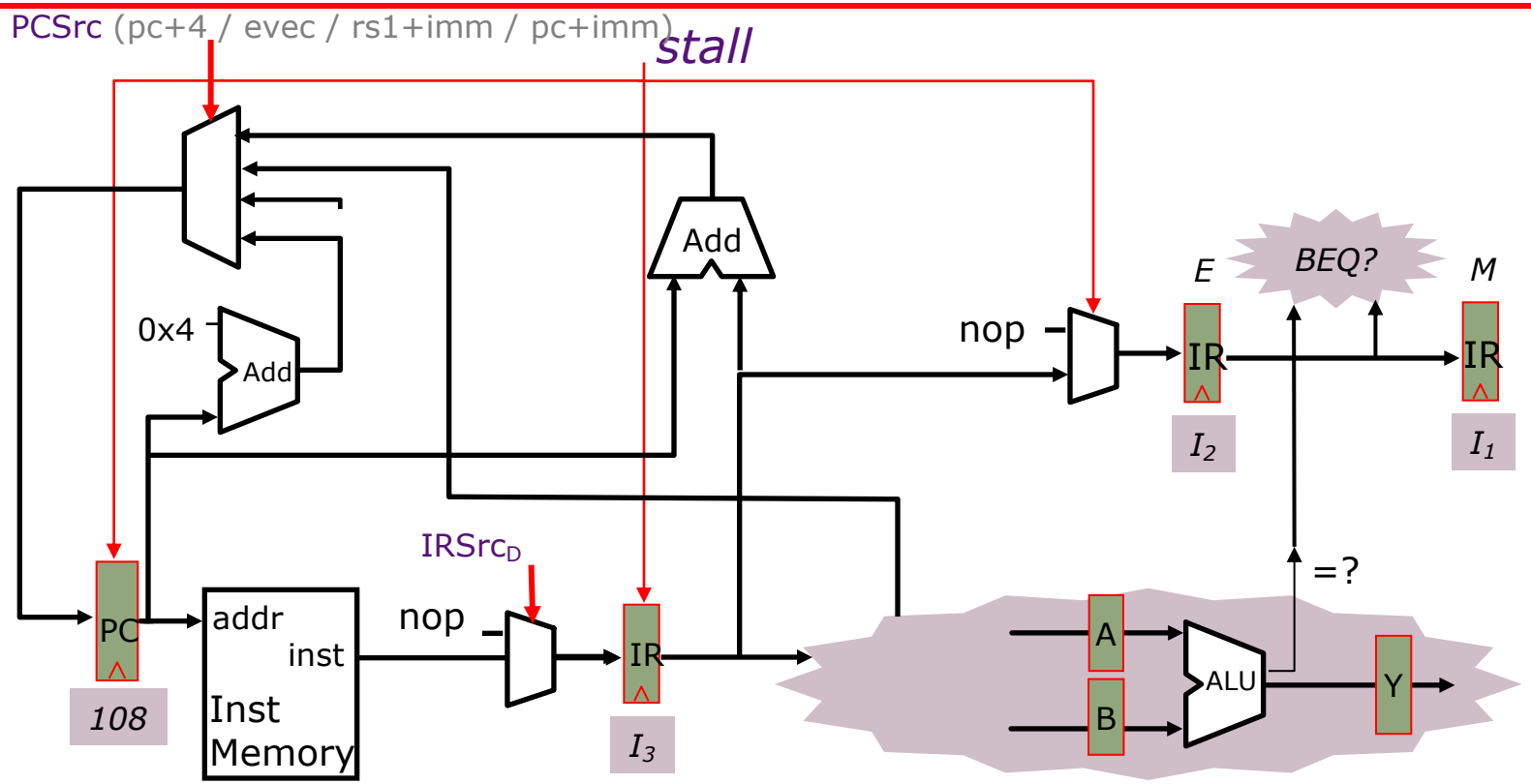
Pipelining Conditional Branches



I_1	096	ADD
I_2	100	BEQ a1 a2 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?*

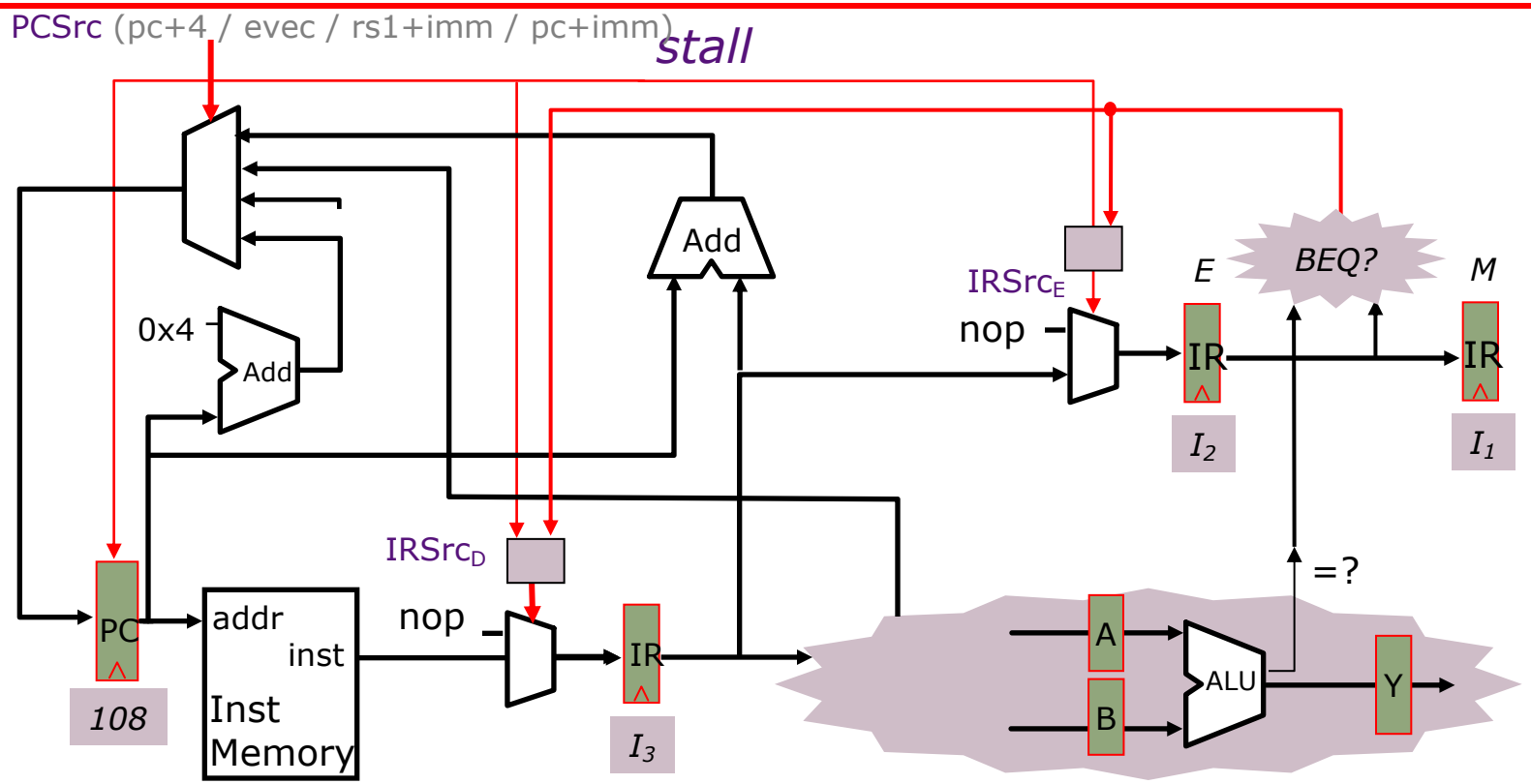
Pipelining Conditional Branches



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

Pipelining Conditional Branches



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

$IRSrc_D$ = *Case* $opcode_E$
- Taken branch ⇒ nop
- *Case* $opcode_D$
  - JAL, JALR ⇒ nop
  - ... ⇒ IM

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

$IRSrc_E$ = *Case* $opcode_E$
- Taken branch ⇒ nop
- ... ⇒ stall·nop + !stall·IR_D

PCSrc = *Case* $opcode_E$
- Taken branch ⇒ pc + imm (from E)
- *Case* $opcode_D$
  - JAL ⇒ pc + imm (from D)
  - JALR ⇒ rs1 + imm (from D)
  - ... ⇒ pc+4

pc+4 is a speculative guess

nop ⇒ Kill
pc+imm /rs1+imm⇒ Restart
pc+4 ⇒ Speculate

Branch Pipeline Diagrams
(resolved in execute stage)

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: BEQ a1 a2 200	IF_2	ID_2	EX_2	MA_2	WB_2				
(I_3) 104: ADD			IF_3	ID_3	nop	nop	nop		
(I_4) 108:				IF_4	nop	nop	nop	nop	
(I_5) 304: ADD					IF_5	ID_5	EX_5	MA_5	WB_5

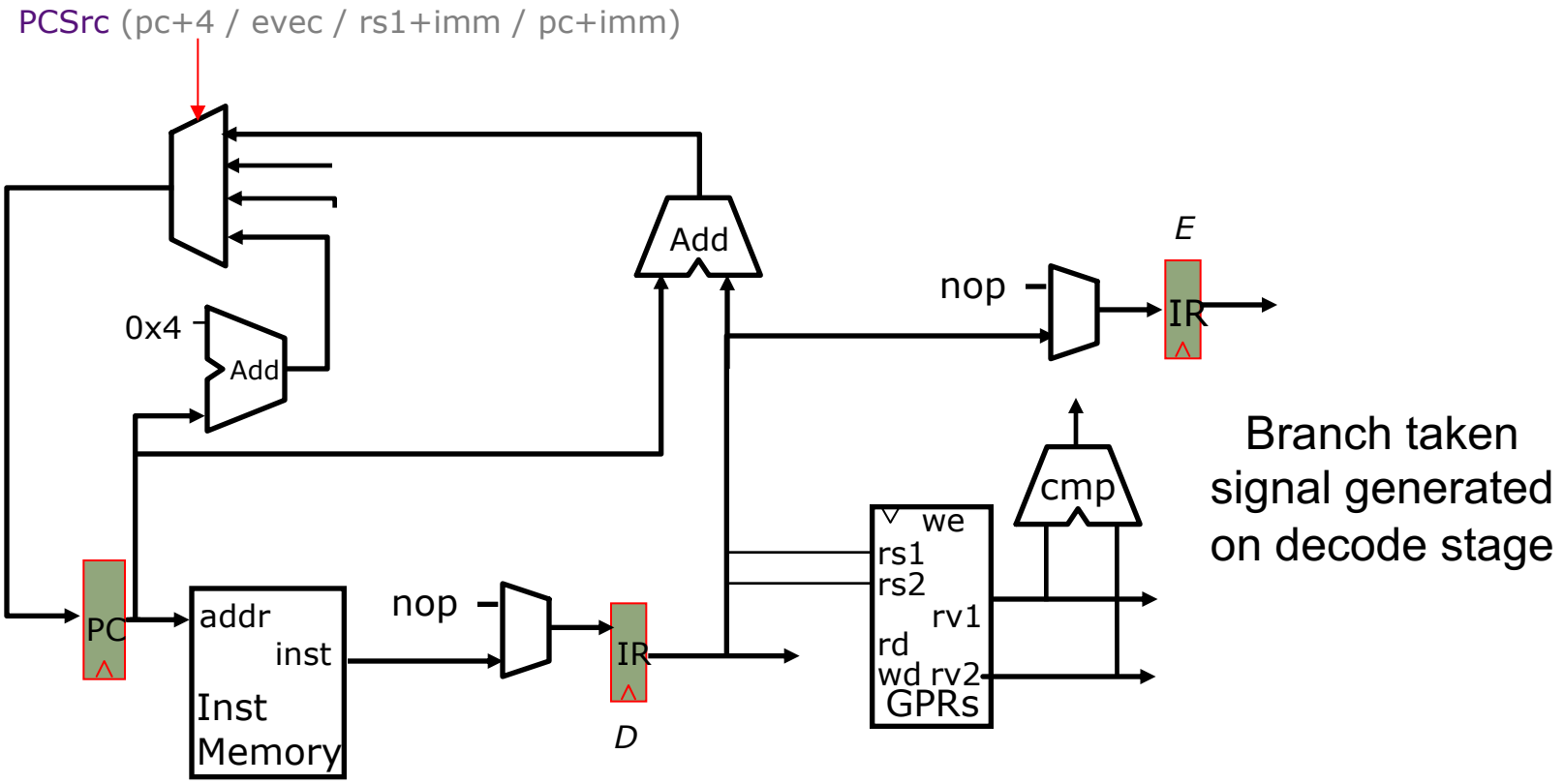
Resource Usage Diagram

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4	I_5				
ID		I_1	I_2	I_3	nop	I_5			
EX			I_1	I_2	nop	nop	I_5		
MA				I_1	I_2	nop	nop	I_5	
WB					I_1	I_2	nop	nop	I_5

nop ⇒ *pipeline bubble*

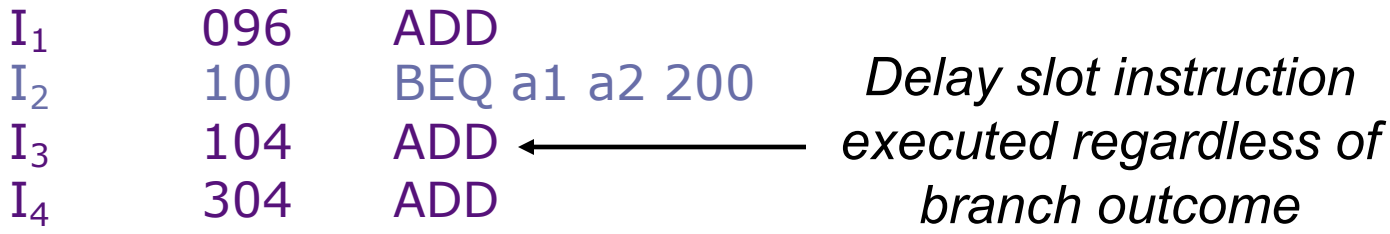
Reducing Branch Penalty (resolve in decode stage)

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



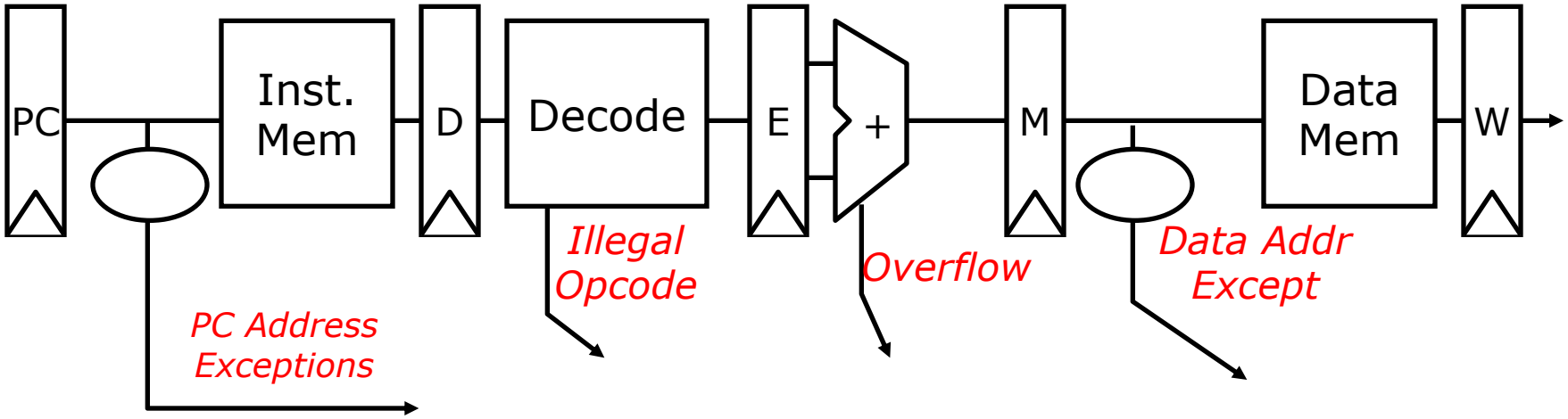
Branch Delay Slots
(expose control hazard to software)

- 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.



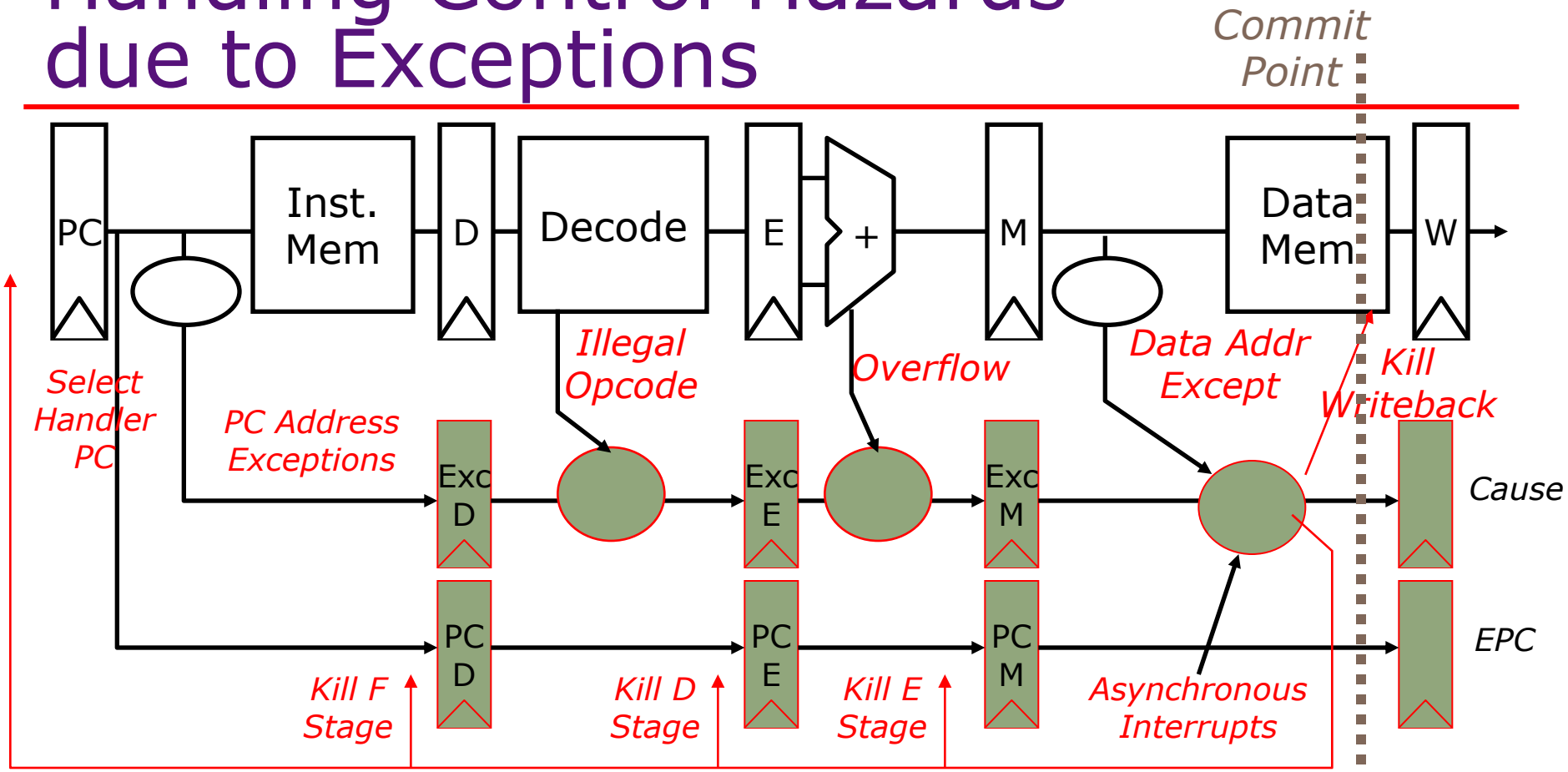
- Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*

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, as in MIPS)

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

Next lecture: Superscalar & Scoreboarded Pipelines