Pipeline Hazards

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

Based on the material prepared by
Arvind and Krste Asanovic

Technology Assumptions

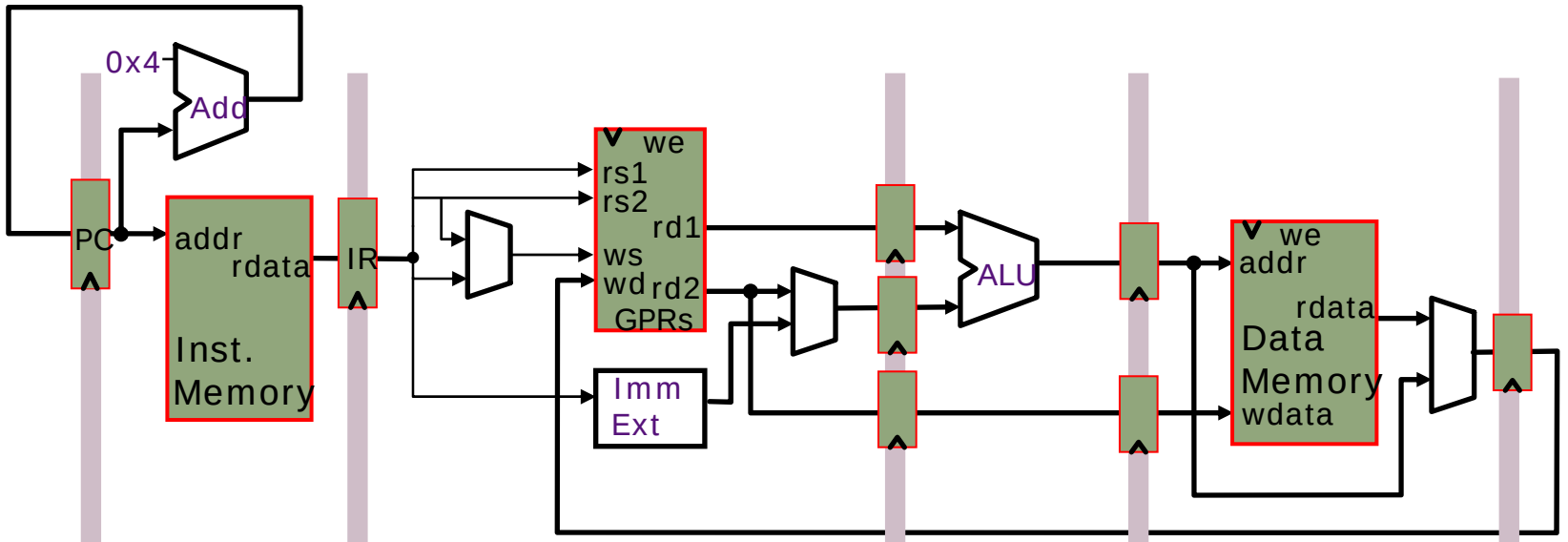
- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

It makes the following timing assumption valid

$$t_{IM} \approx t_{RF} \approx t_{ALU} \approx t_{DM} \approx t_{RW}$$

A 5-stage pipelined Harvard architecture will be the focus of our detailed design

5-Stage Pipelined Execution

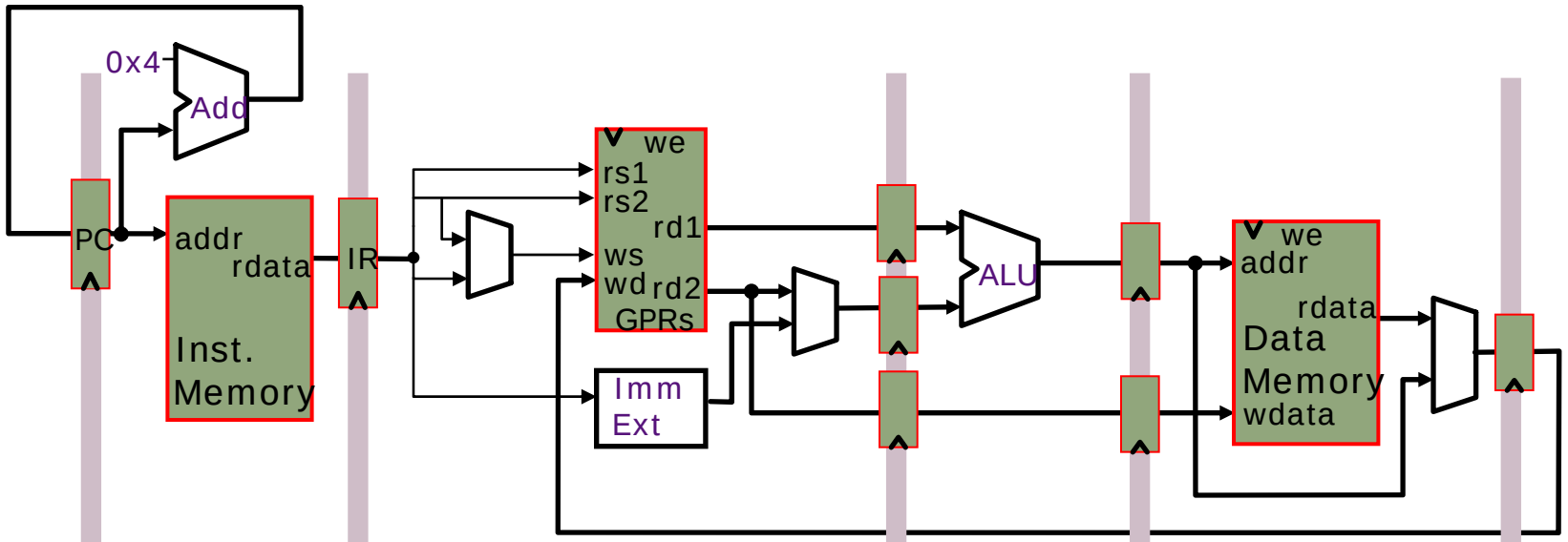


I-Fetch (IF) | *Decode, Reg. Fetch (ID)* | *Execute (EX)* | *Memory (MA)* | *Write -Back (WB)*

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

5-Stage Pipelined Execution

Resource Usage Diagram

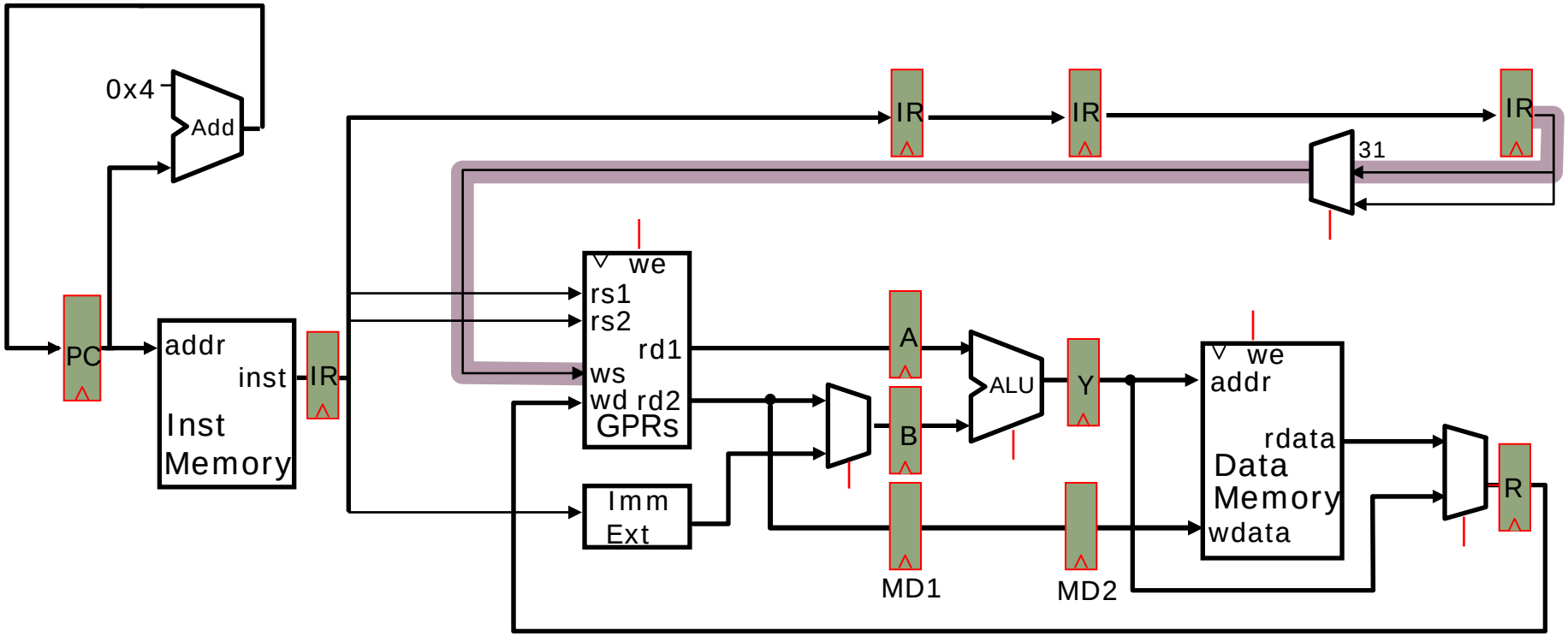


I-Fetch (IF) | *Decode, Reg. Fetch (ID)* | *Execute (EX)* | *Memory (MA)* | *Write -Back (WB)*

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	I_4	I_5			
EX			I_1	I_2	I_3	I_4	I_5		
MA				I_1	I_2	I_3	I_4	I_5	
WB					I_1	I_2	I_3	I_4	I_5

Resources

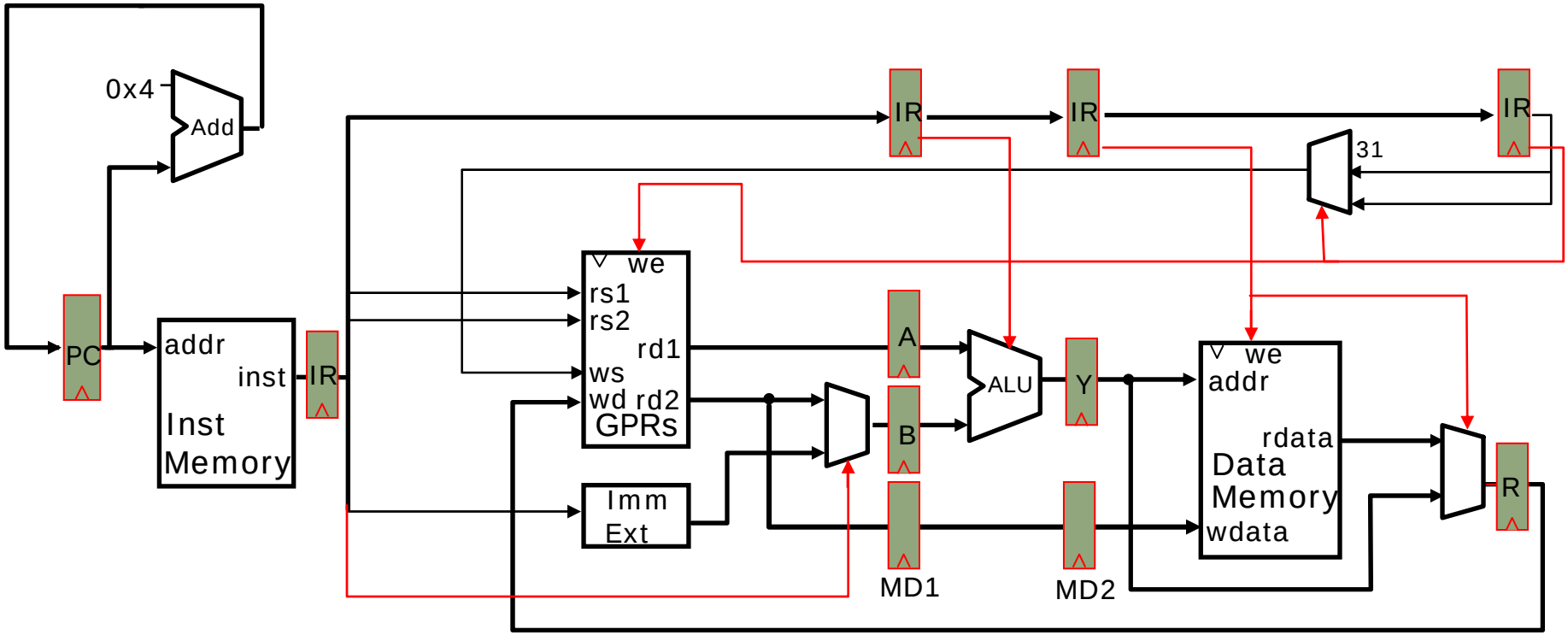
Pipelined Execution: ALU Instructions



Not quite correct!

We need an Instruction Reg (IR) for each stage

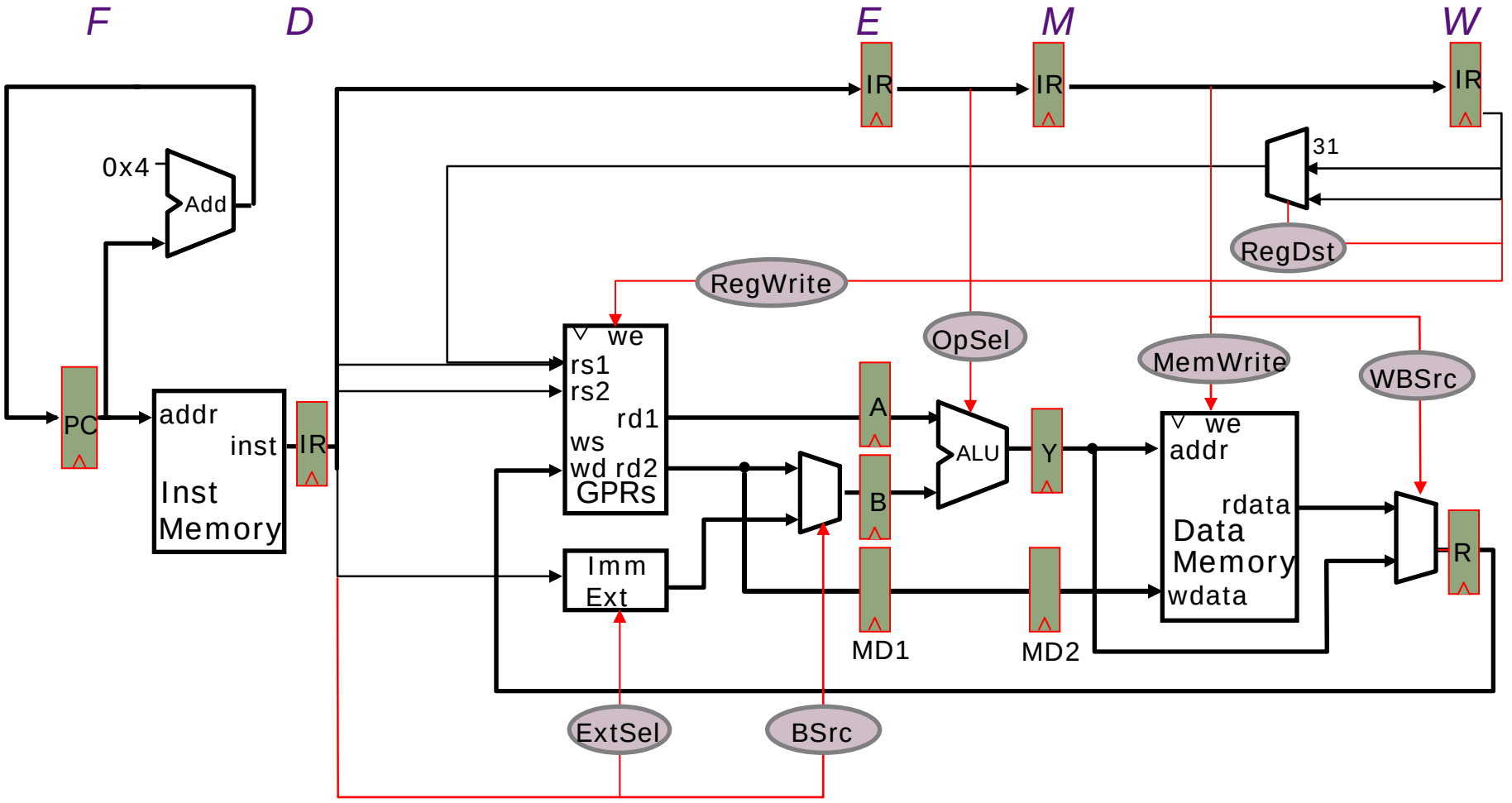
IR's and Control points



Are control points connected properly?

- ALU instructions
- Load/Store instructions
- Write back

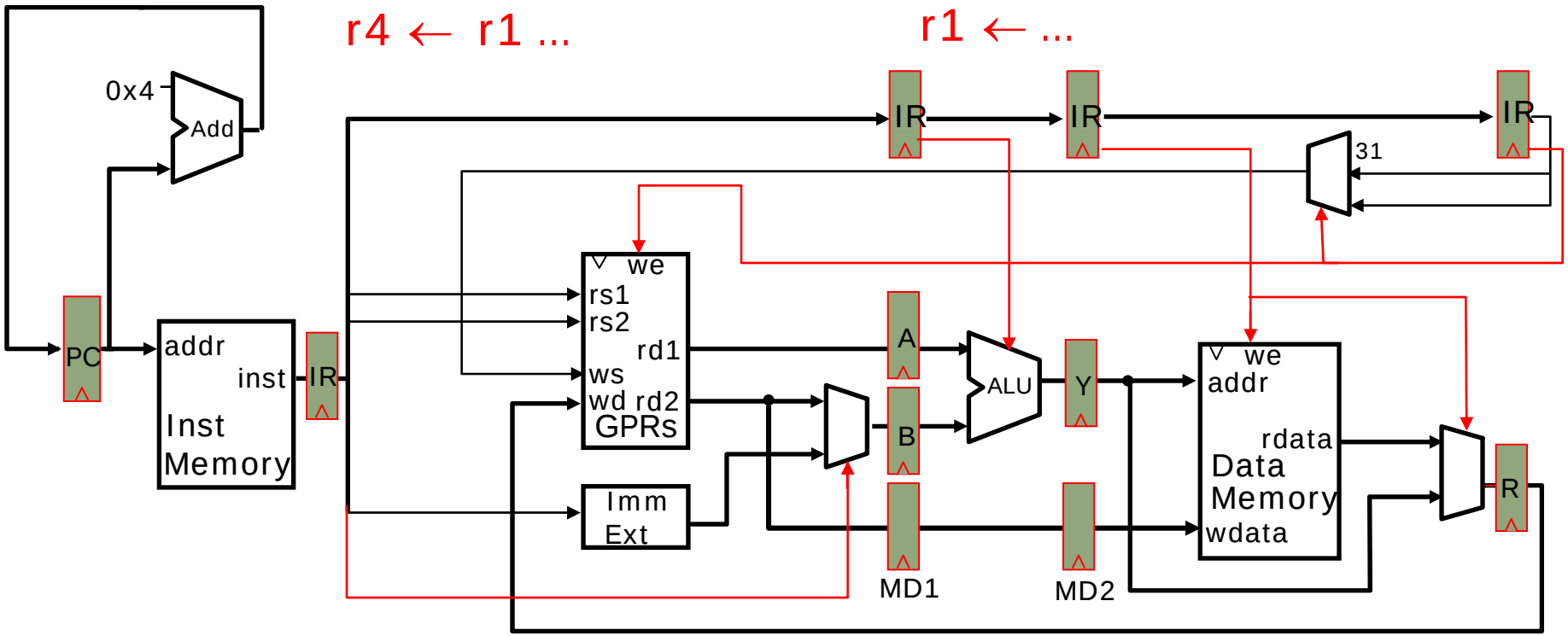
Pipelined MIPS Datapath
without jumps



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 produce data that is needed by a later instruction
  - *data hazard*
- In the extreme case, an instruction may determine the next instruction to be executed
  - *control hazard (branches, interrupts,...)*

Data Hazards



...
r1 ← r0 + 10
r4 ← r1 + 17
...

r1 is stale. Oops!

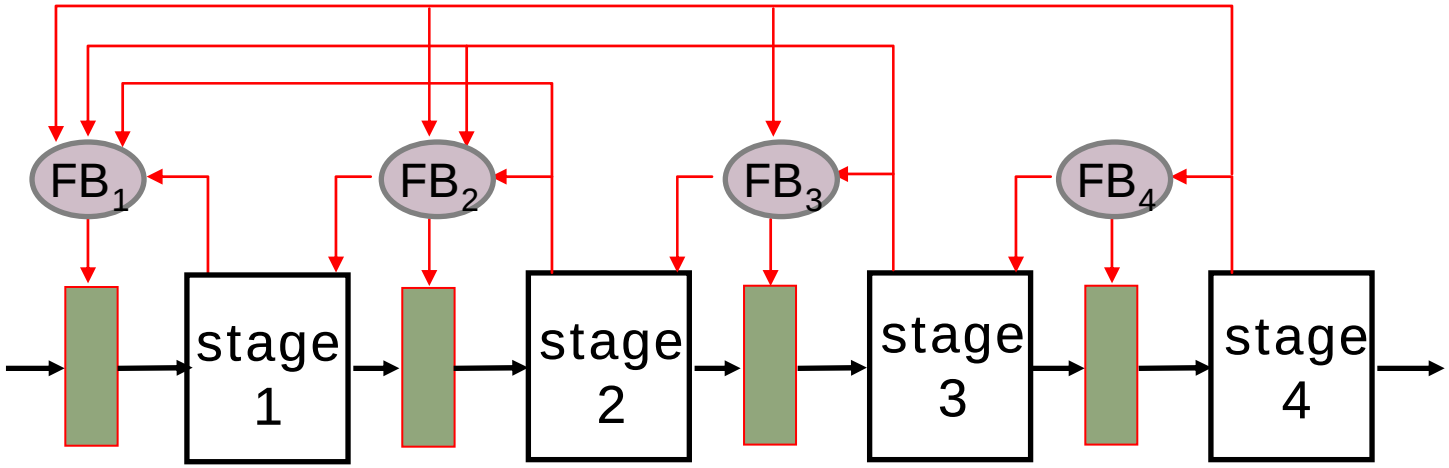
Resolving Data Hazards

Freeze earlier pipeline stages until the data becomes available ⇒ *interlocks*

If data is available somewhere in the datapath provide a *bypass* to get it to the right stage

Speculate about the hazard resolution and *kill* the instruction later if the speculation is wrong.

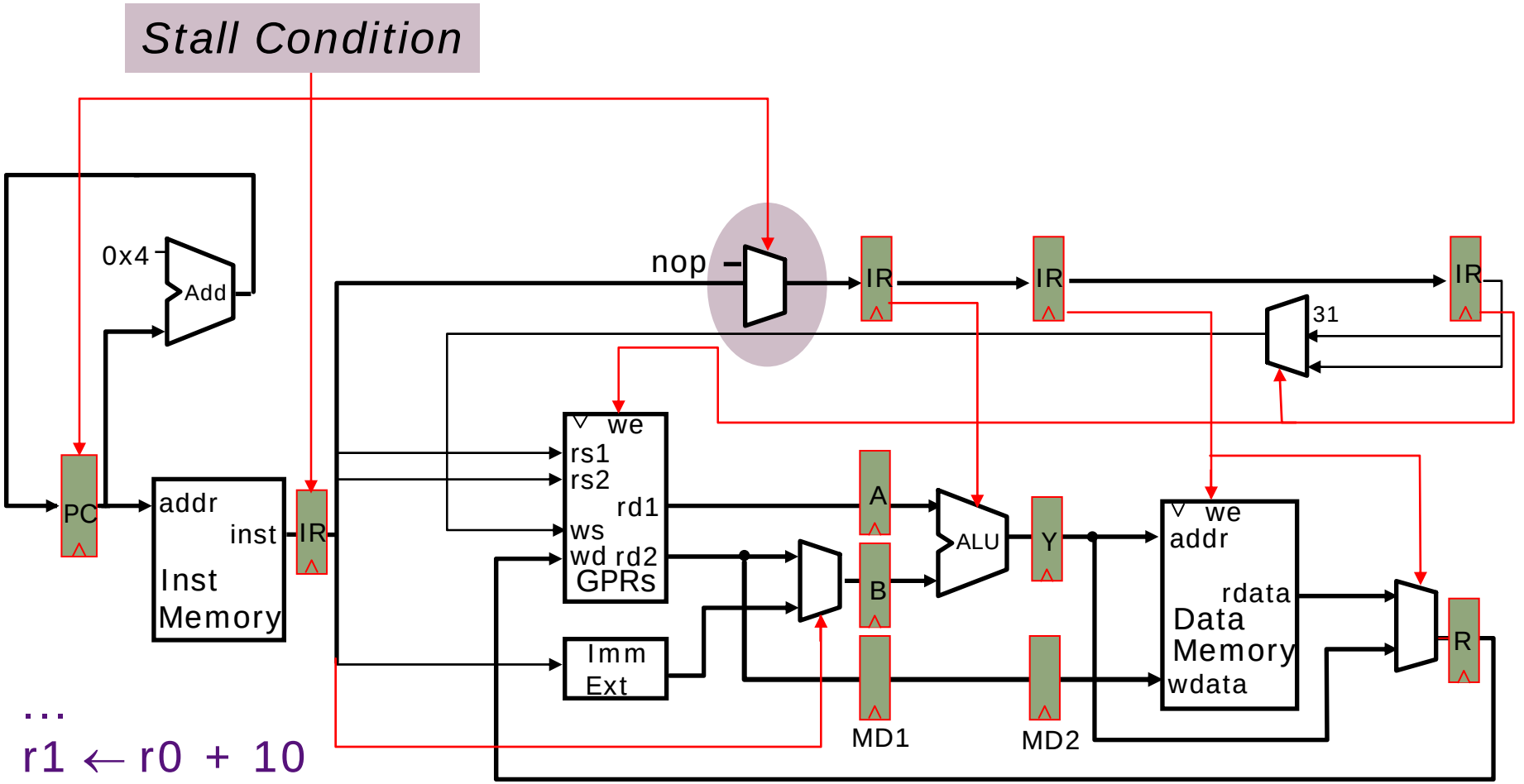
Feedback to Resolve Hazards



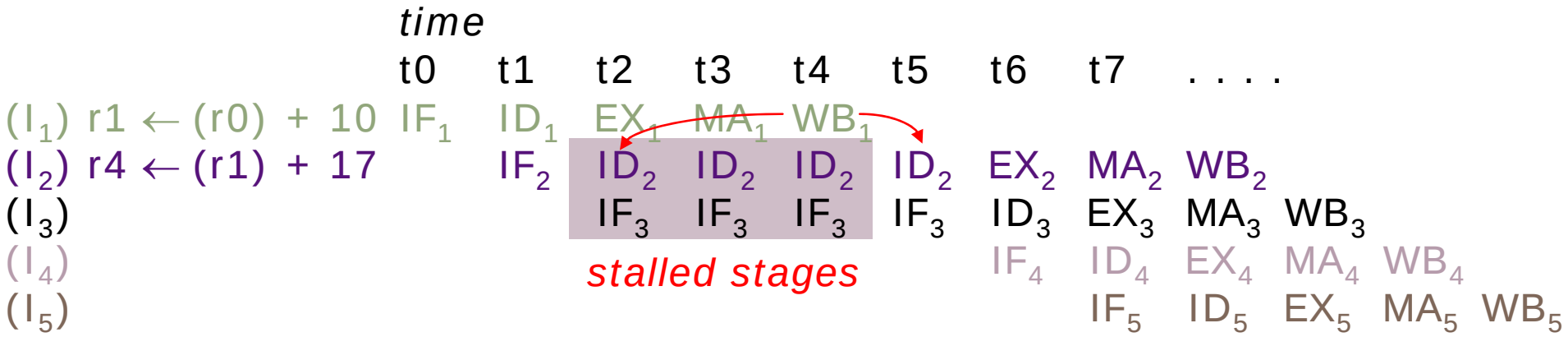
- Detect a hazard and provide feedback to previous stages to *stall or kill instructions*

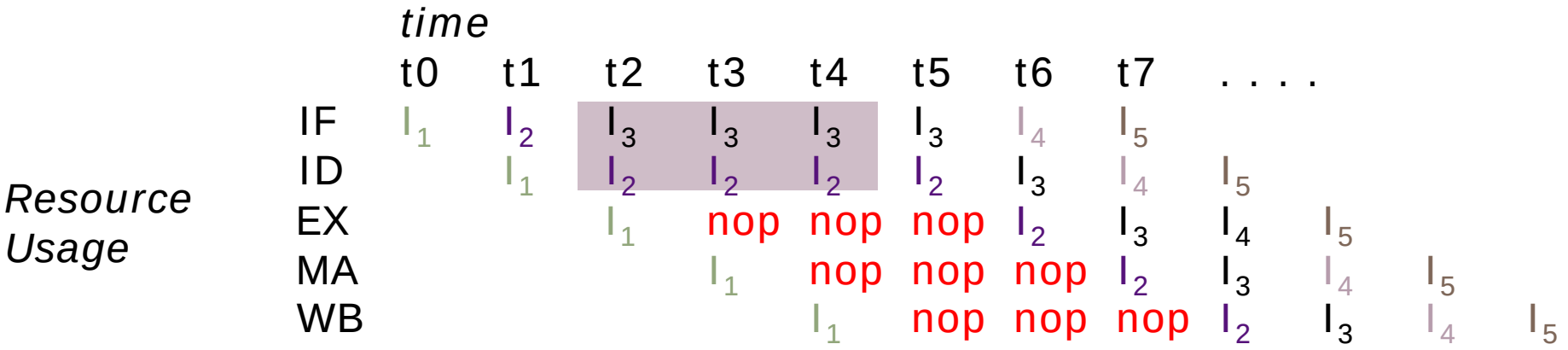
- Controlling a pipeline in this manner works provided *the instruction at stage i+1 can complete without any interference from instructions in stages 1 to i* (otherwise deadlocks may occur)

Interlocks to resolve Data Hazards



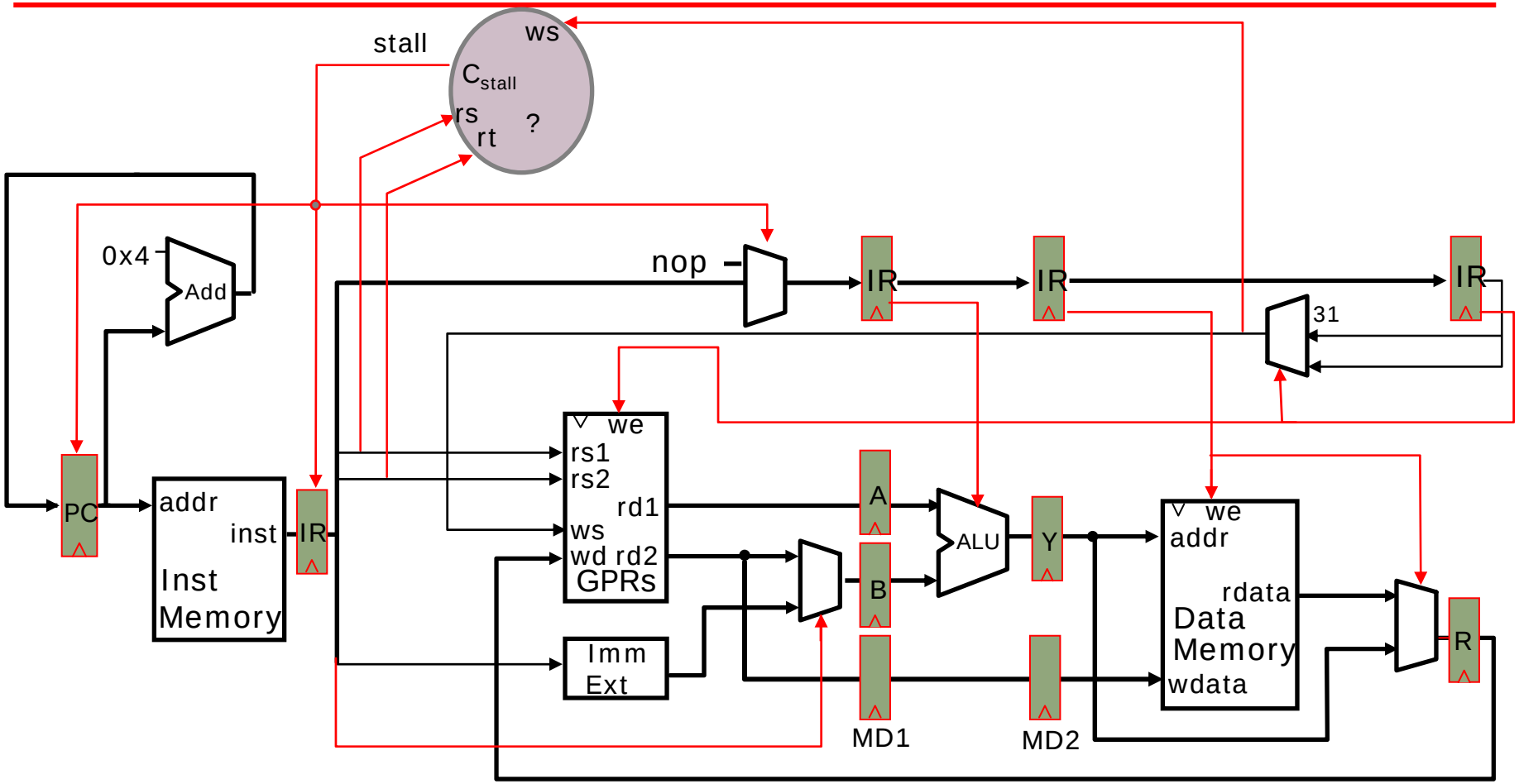
Stalled Stages and Pipeline Bubbles





nop ⇒ *pipeline bubble*

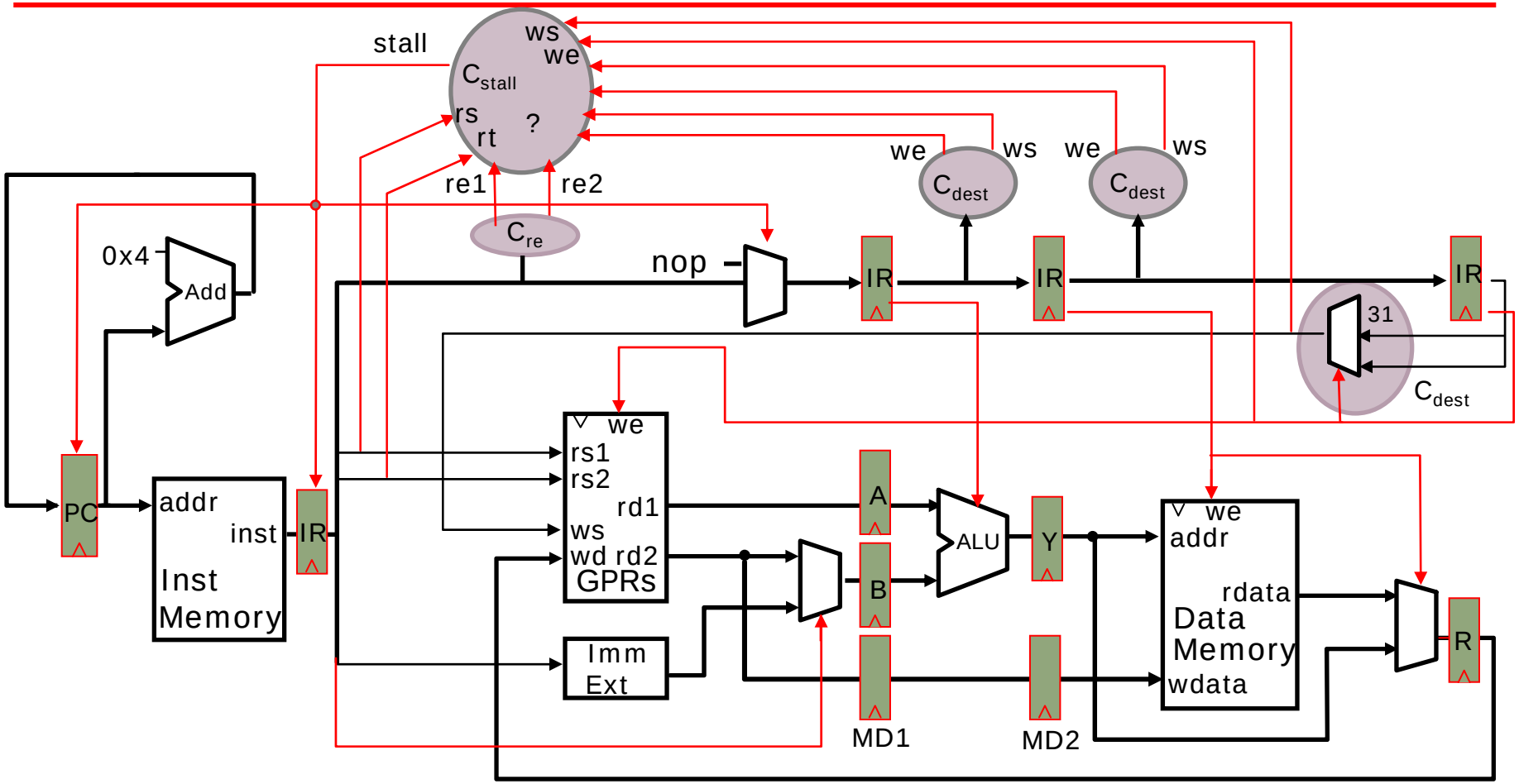
Interlock Control Logic



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

Interlocks Control Logic

ignoring jumps & branches

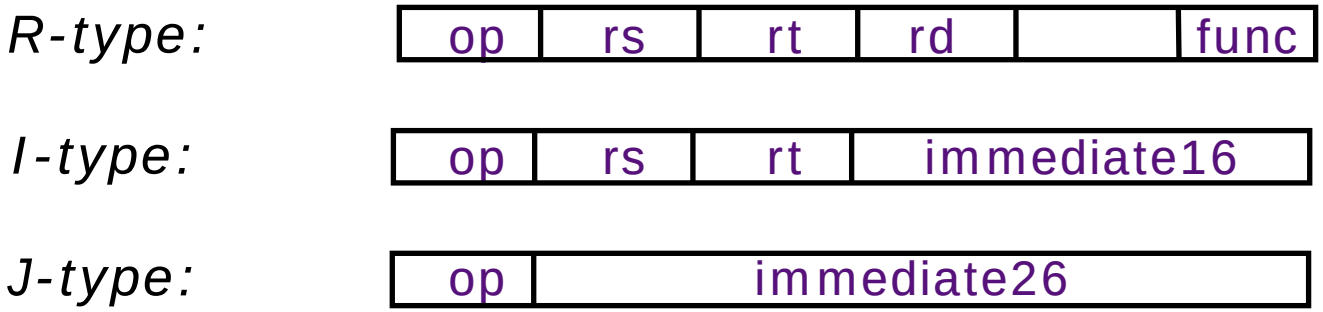


Should we always stall if the rs field matches some rd?

not every instruction writes a register ⇒ we

not every instruction reads a register ⇒ re

Source & Destination Registers



		source(s)	*destination*
ALU	rd ← (rs) func (rt)	rs, rt	rd
ALUi	rt ← (rs) op imm	rs	rt
LW	rt ← M [(rs) + imm]	rs	rt
SW	M [(rs) + imm] ← (rt)	rs, rt	
BZ	*cond* (rs)		
	true: PC ← (PC) + imm	rs	
	false: PC ← (PC) + 4	rs	
J	PC ← (PC) + imm		
JAL	r31 ← (PC), PC ← (PC) + imm		31
JR	PC ← (rs)	rs	
JALR	r31 ← (PC), PC ← (rs)	rs	31

Deriving the Stall Signal

C_{dest}

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

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

C_{re}

re1 = *Case* opcode
- ALU, ALUi, LW, SW, BZ, JR, JALR ⇒ on
- J, JAL ⇒ off

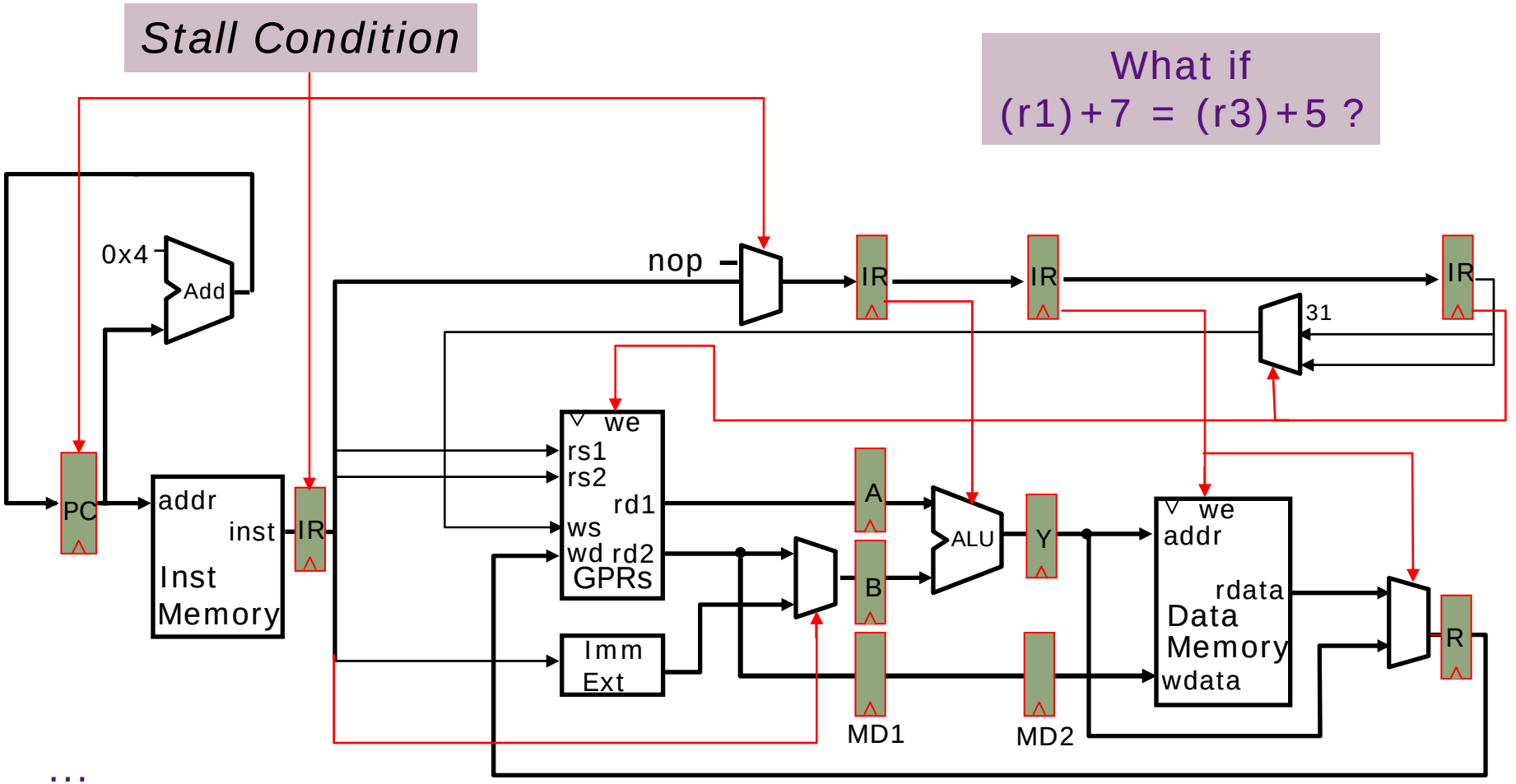
re2 = *Case* opcode
- ALU, SW ⇒ on
- … ⇒ off

C_{stall}

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

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 ⇒ *data hazard*

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

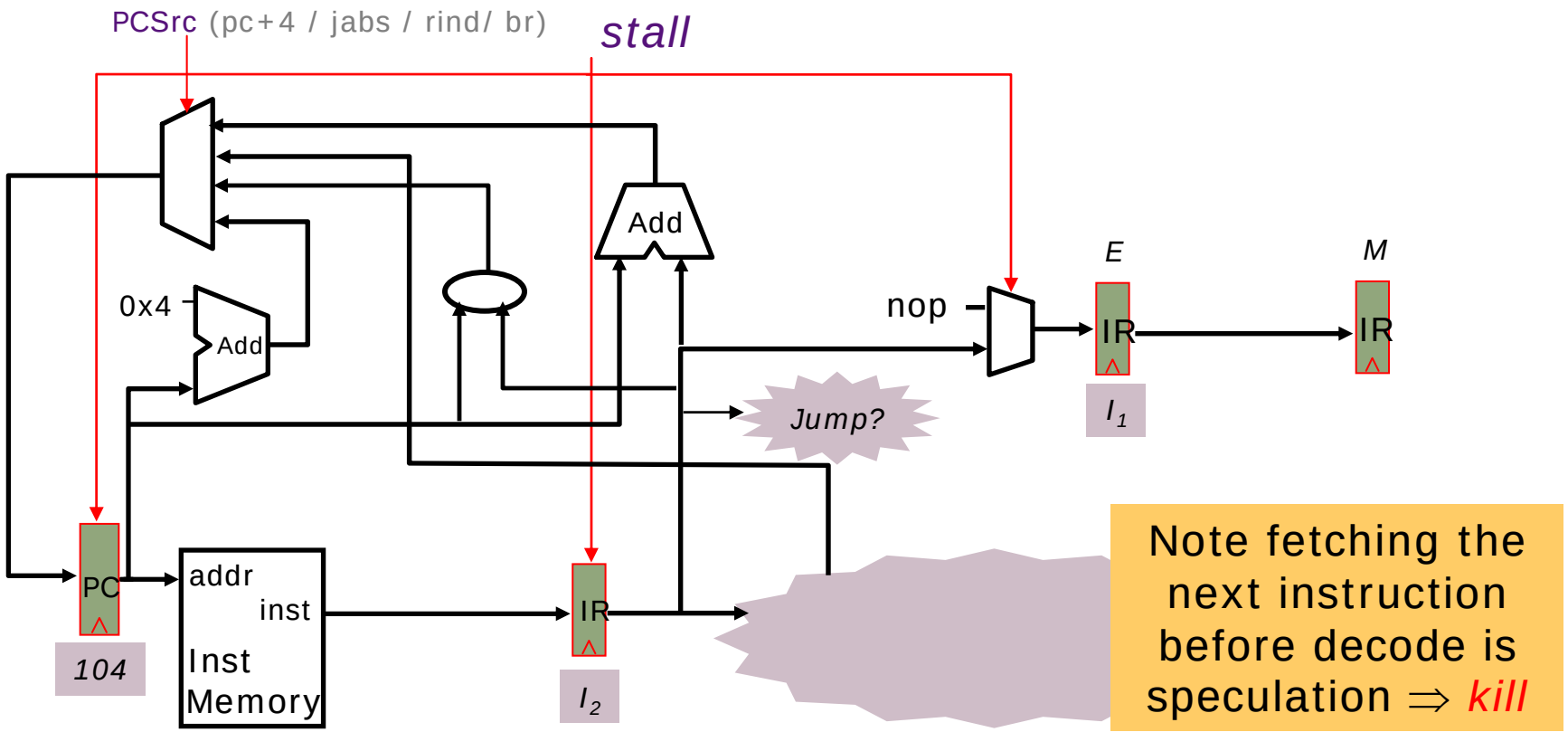
Load/Store hazards, even when they do exist, are often resolved in the memory system itself.

More on this later in the course.

CSAIL

Five-minute break to stretch your legs

Complications due to Jumps

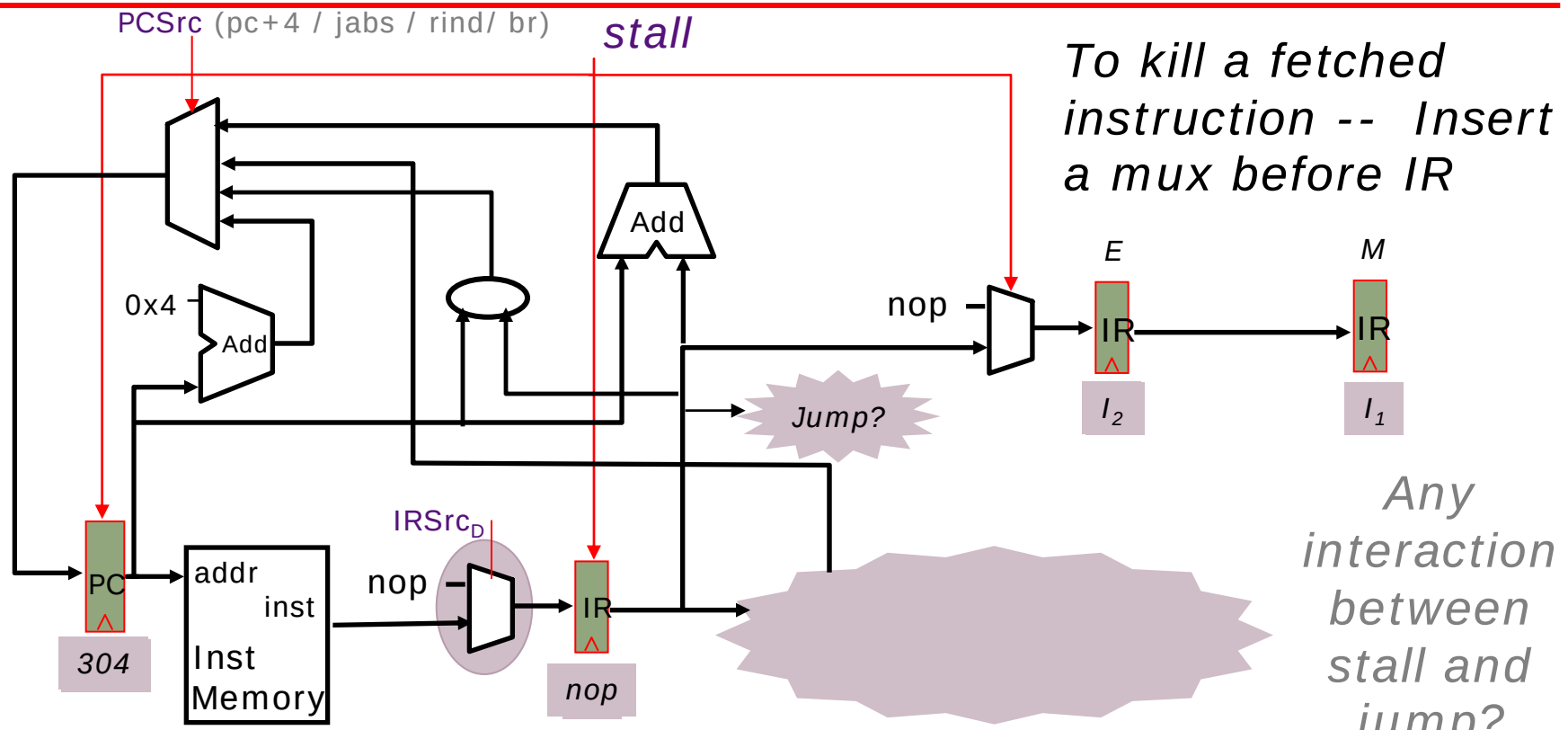


I_1	096	ADD	
I_2	100	J	200
I_3	~~104~~	~~ADD~~	*kill*
I_4	304	ADD	

A jump instruction kills (not stalls) the following instruction

How?

Pipelining Jumps



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

IRSrc$_D$ = *Case* opcode$_D$
J, JAL ⇒ nop
… ⇒ IM

Jump Pipeline Diagrams

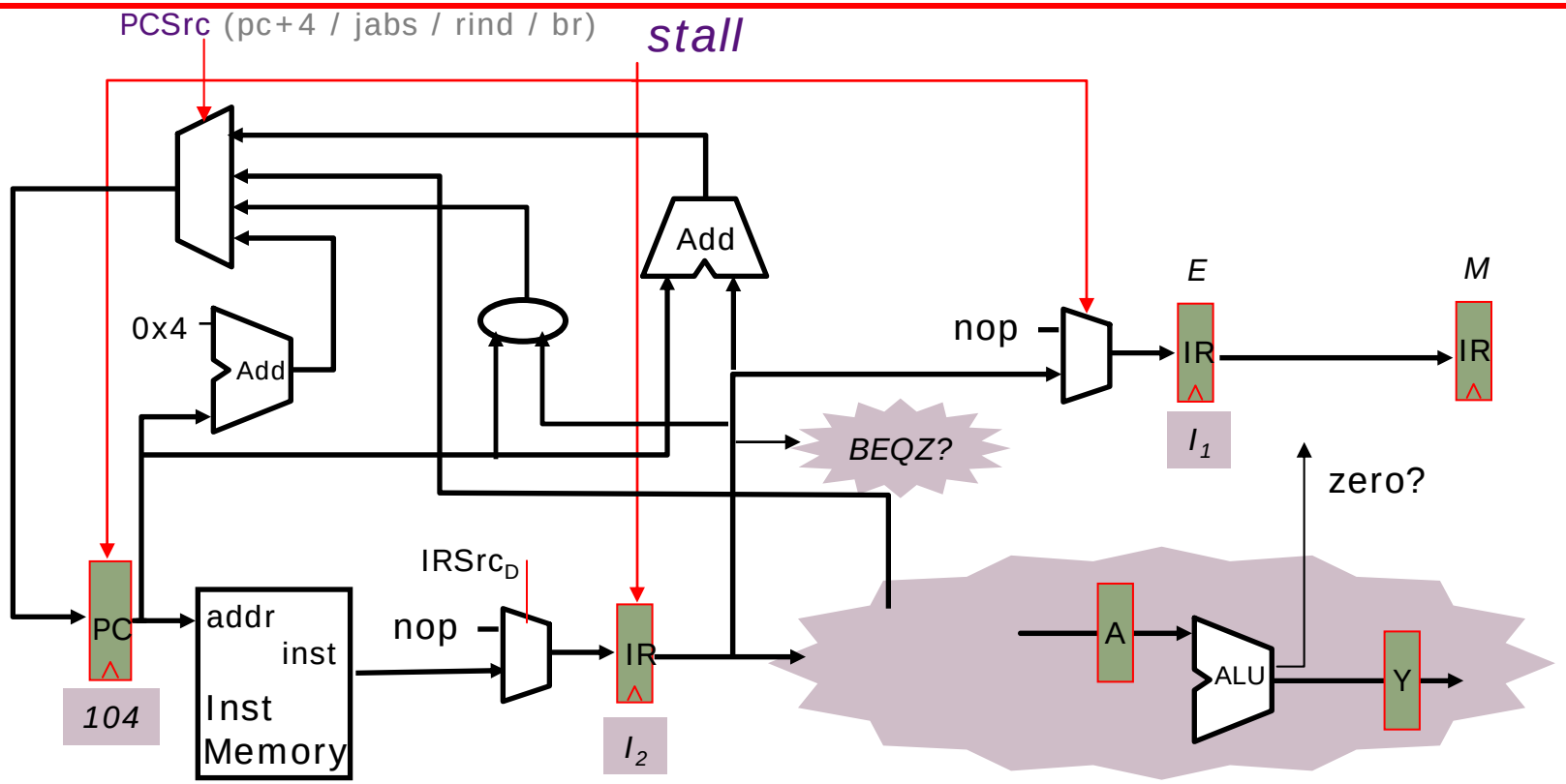
	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: J 200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	nop	nop	nop	nop		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

Resource Usage

	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

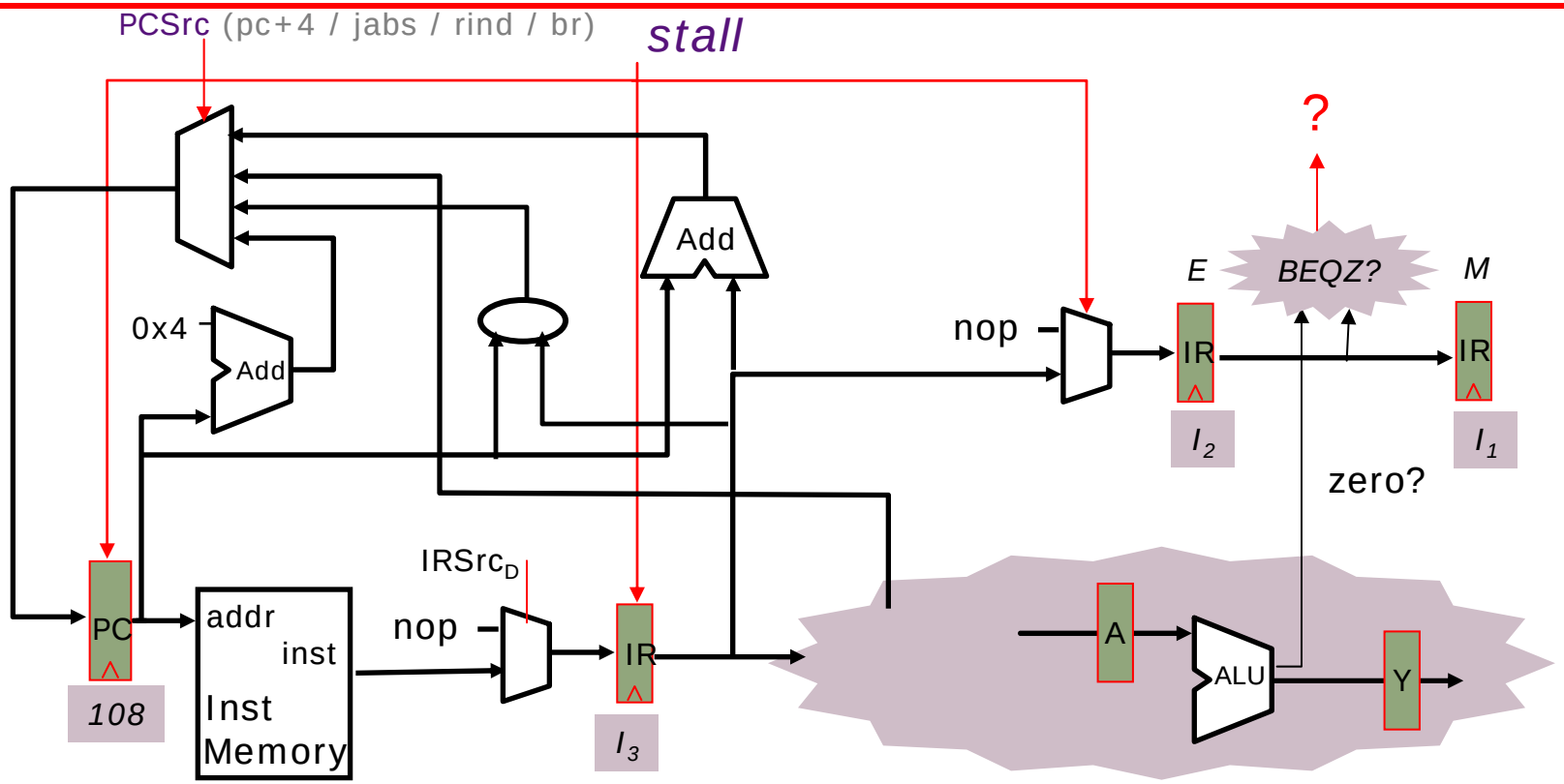


I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Pipelining Conditional Branches



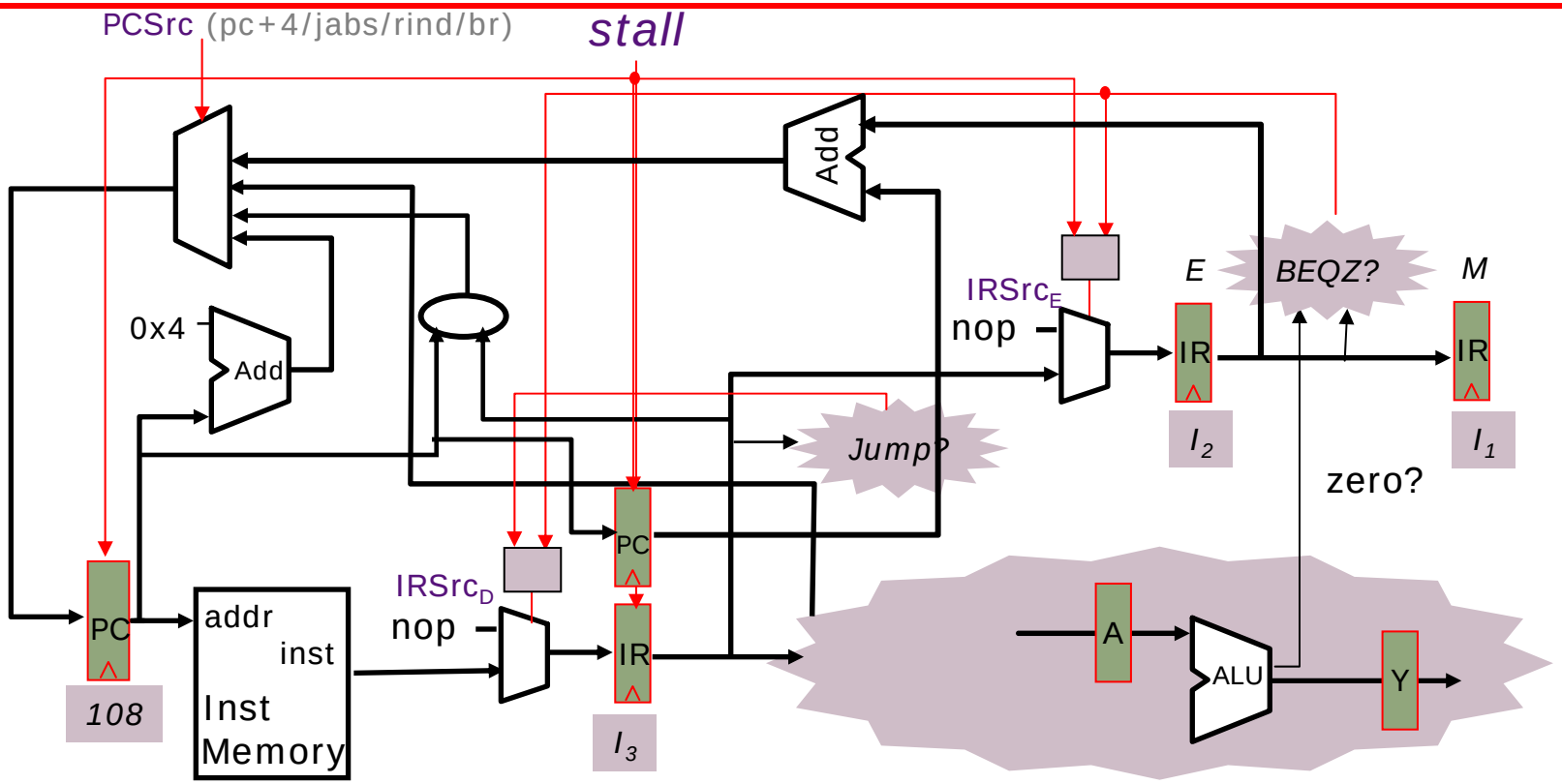
I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

If the branch is taken

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

$\Rightarrow$ *stall signal is not valid*

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

New Stall Signal

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

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

Instruction at the decode stage is invalid

Control Equations for PC and IR Muxes

PCSrc = *Case* $opcode_E$
- BEQZ.z, BNEZ.!z ⇒ br
- ... ⇒
  - *Case* $opcode_D$
    - J, JAL ⇒ jabs
    - JR, JALR ⇒ rind
    - ... ⇒ pc+4

$IRSrc_D$ = *Case* $opcode_E$
- BEQZ.z, BNEZ.!z ⇒ nop
- ... ⇒
  - *Case* $opcode_D$
    - J, JAL, JR, JALR ⇒ nop
    - ... ⇒ IM

$IRSrc_E$ = *Case* $opcode_E$
- BEQZ.z, BNEZ.!z ⇒ nop
- ... ⇒ stall.nop + !stall.IR_D

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

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: BEQZ 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

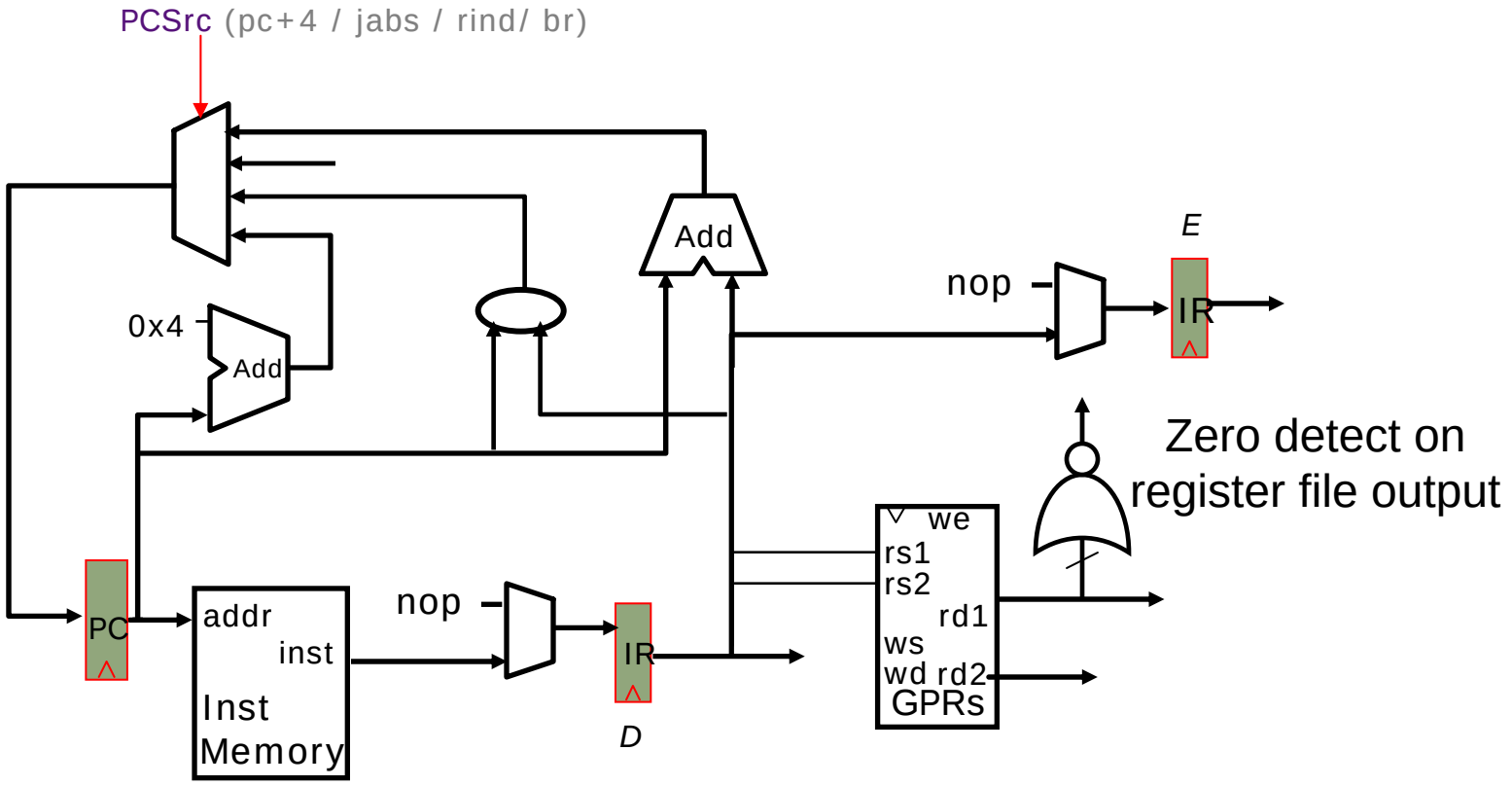
Resource Usage

	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



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	*Delay slot instruction executed regardless of branch outcome*
I_3	104	ADD	←
I_4	304	ADD	

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

Bypassing

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) r1 ← r0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) r4 ← r1 + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	MA_3
(I_4)			*stalled stages*				IF_4	ID_4	EX_4
(I_5)								IF_5	ID_5

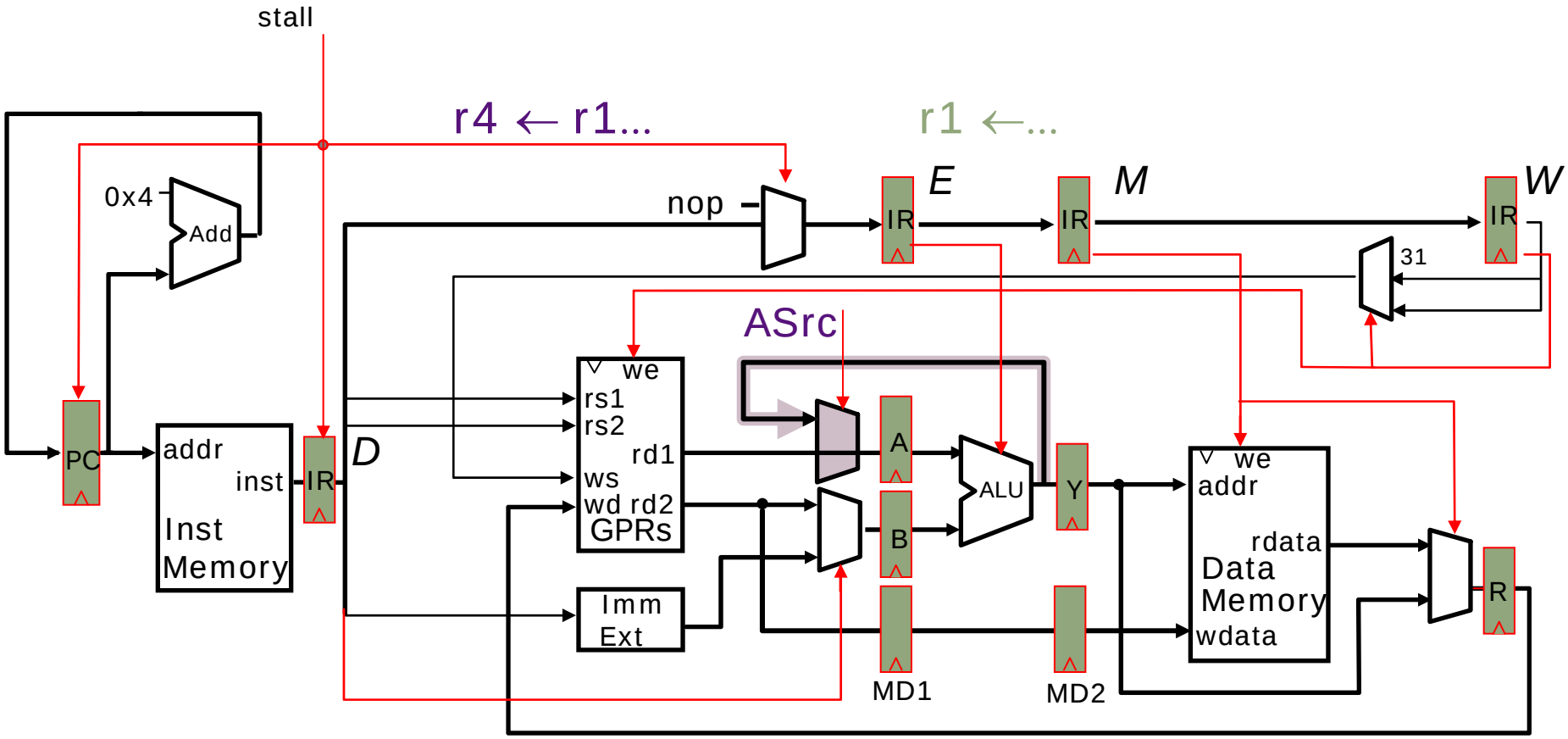
Each *stall or kill* introduces a bubble in the pipeline

$$\Rightarrow CPI > 1$$

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

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .	
(I_1) r1 ← r0 + 10		IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) r4 ← r1 + 17			IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3)				IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4)					IF_4	ID_4	EX_4	MA_4	WB_4	
(I_5)						IF_5	ID_5	EX_5	MA_5	WB_5

Adding a Bypass



When does this bypass help?

...

(I_1)	r1 ← r0 + 10	r1 ← M[r0 + 10]	JAL 500
(I_2)	r4 ← r1 + 17	r4 ← r1 + 17	r4 ← r31 + 17
	yes	*no*	*no*

The Bypass Signal

Deriving it from the Stall Signal

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

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

$$ASrc = (rs_D = ws_E).we_E.re1_D$$

Is this correct?

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

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

Bypass and Stall Signals

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

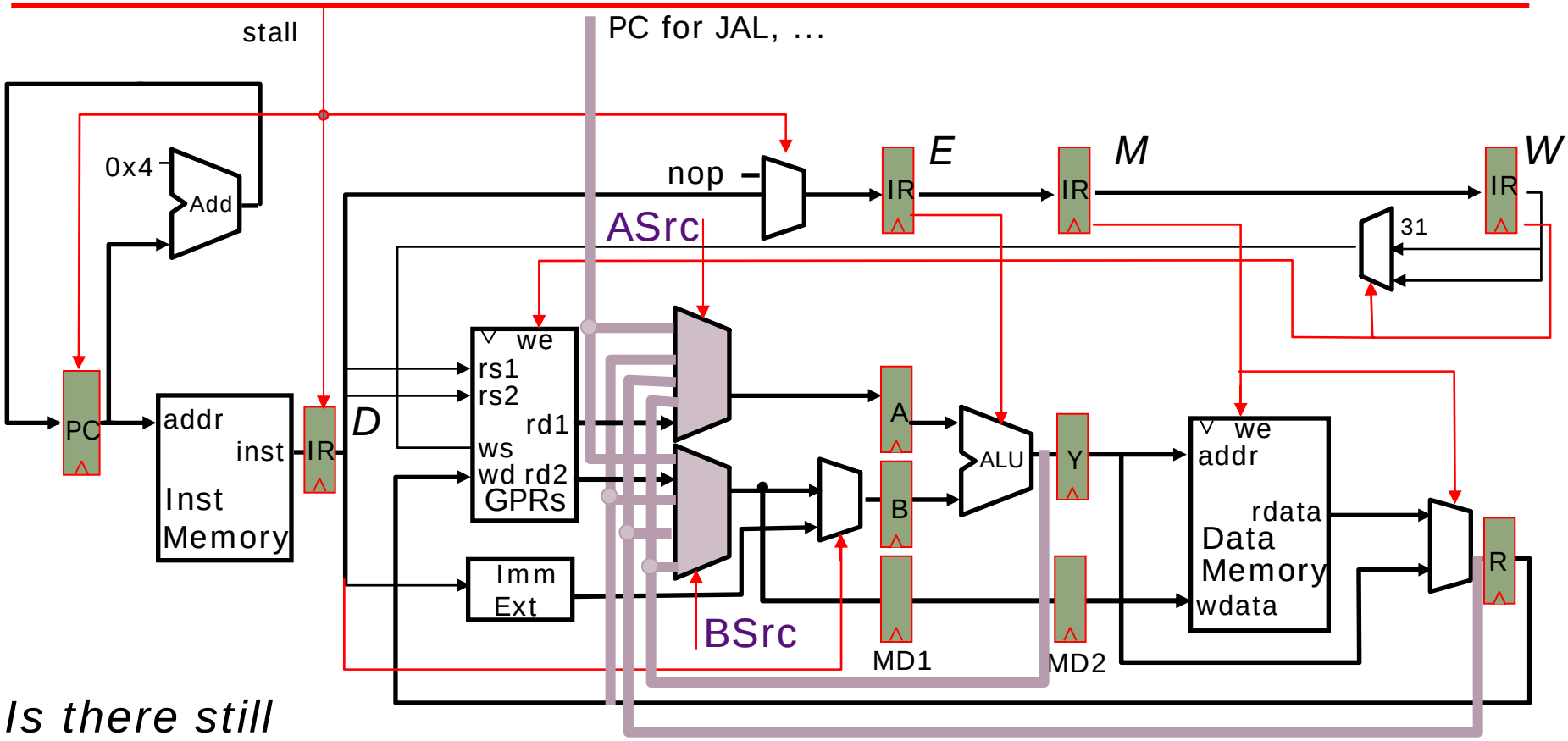
$we\text{-}bypass_E$ = *Case* $opcode_E$
- ALU, ALUi $\Rightarrow (ws \neq 0)$
- ... $\Rightarrow$ off

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

$$ASrc = (rs_D = ws_E).we\text{-}bypass_E . re1_D$$

$$stall = ((rs_D = ws_E).we\text{-}stall_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re1_D + ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re2_D$$

Fully Bypassed Datapath



Is there still a need for the stall signal ?

$$stall = (rs_D = ws_E).(opcode_E = LW_E).(ws_E \neq 0).re1_D + (rt_D = ws_E).(opcode_E = LW_E).(ws_E \neq 0).re2_D$$

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.

Thank you !