Instruction Pipelining: Hazard Resolution, Timing Constraints

Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory

M.I.T.

http://www.csg.csail.mit.edu/6.823

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*

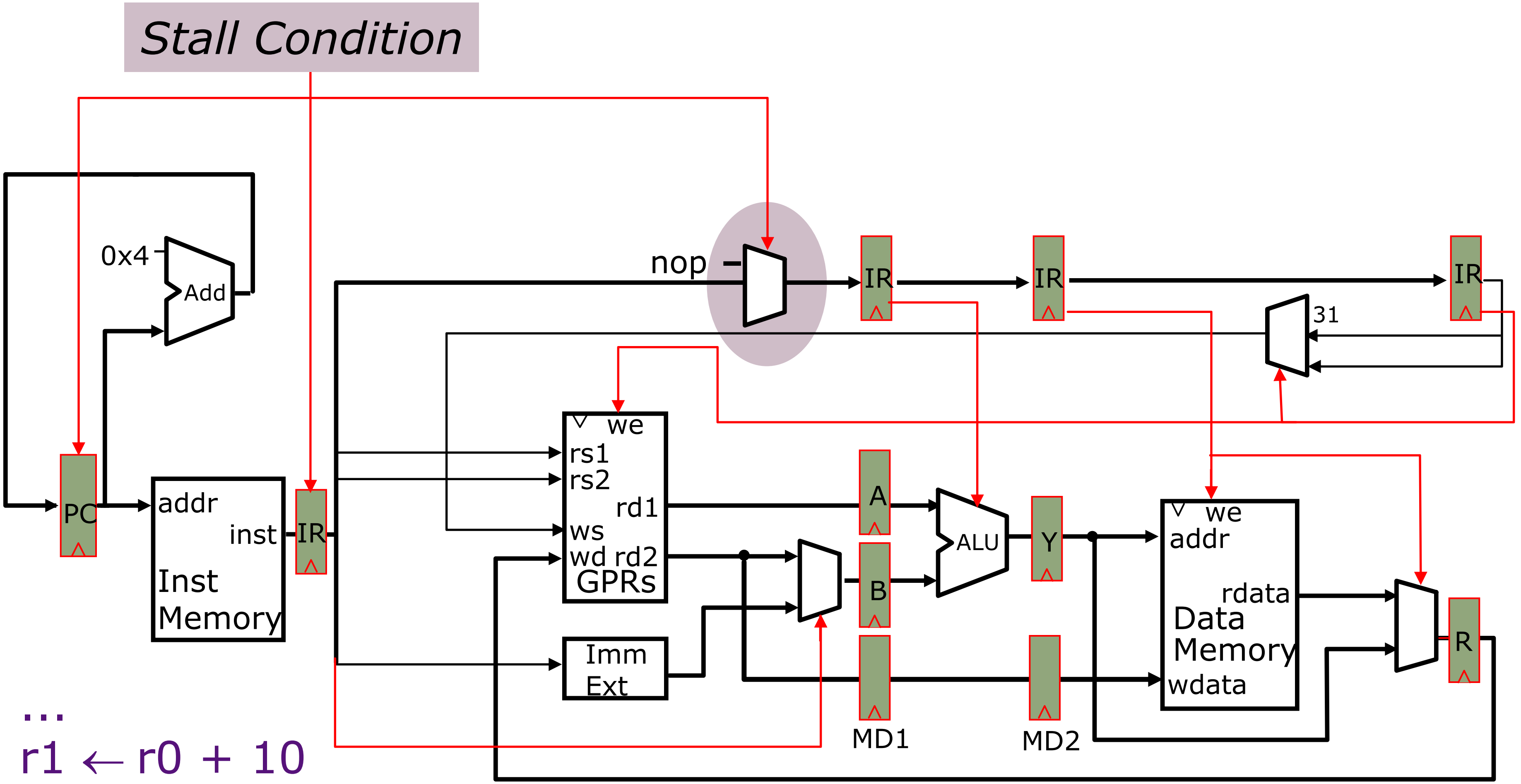
Strategy 3: *Speculate on the dependence*
Two cases:
Guessed correctly → no special action required
Guessed incorrectly → kill and restart

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



...

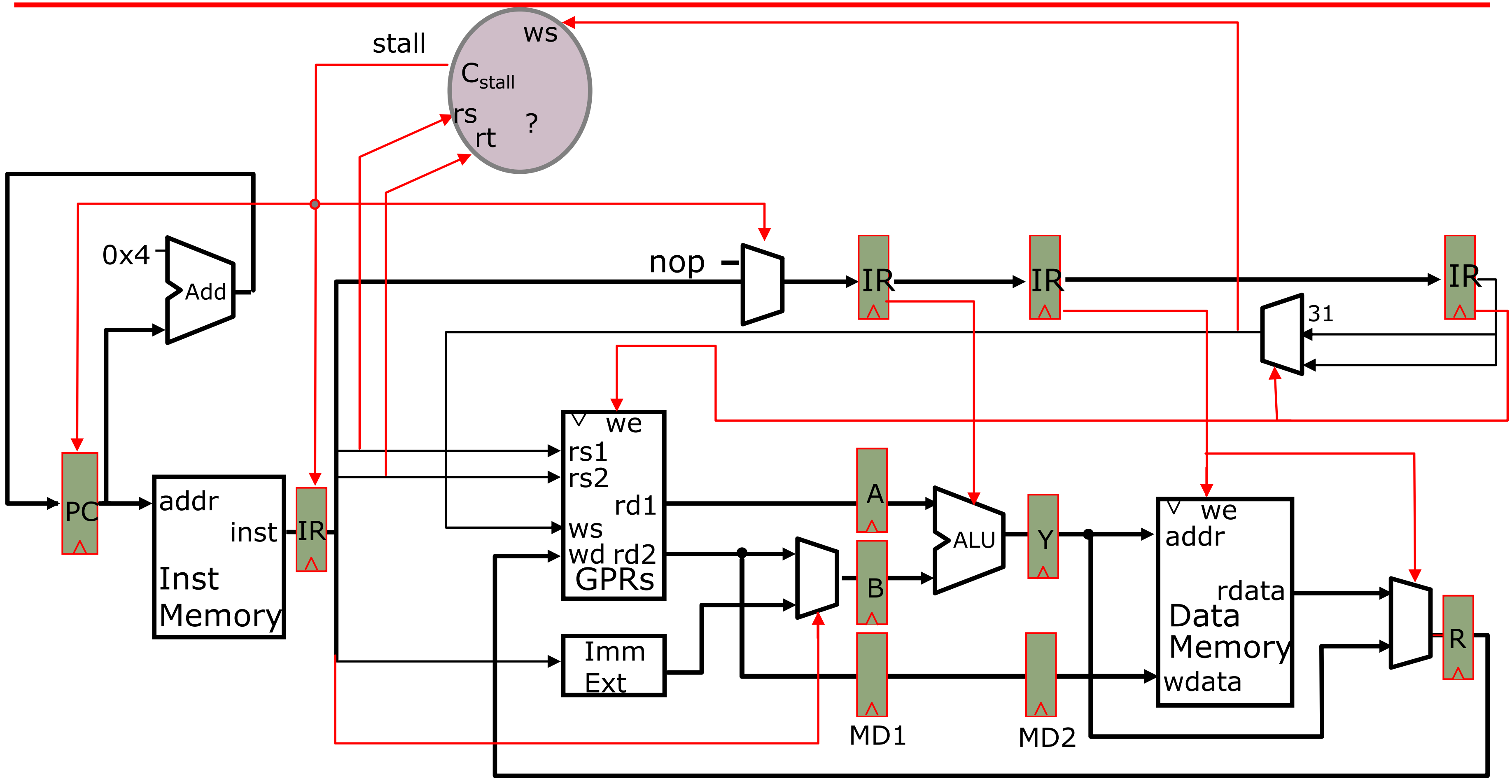
r1 ← r0 + 10

r4 ← r1 + 17

...

How do we know when to stall?

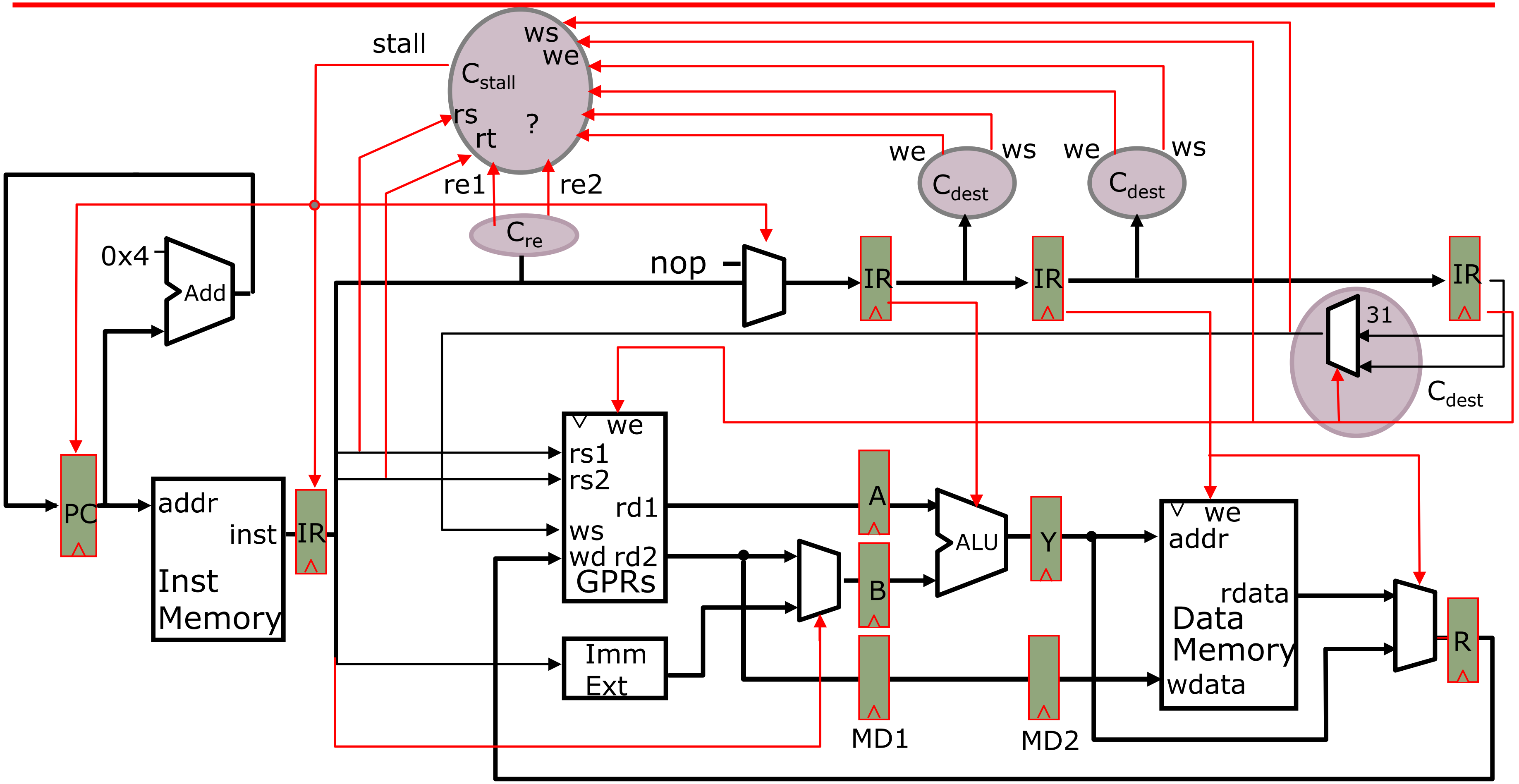
Stall Control Logic



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

Stall Control Logic

ignoring jumps & branches

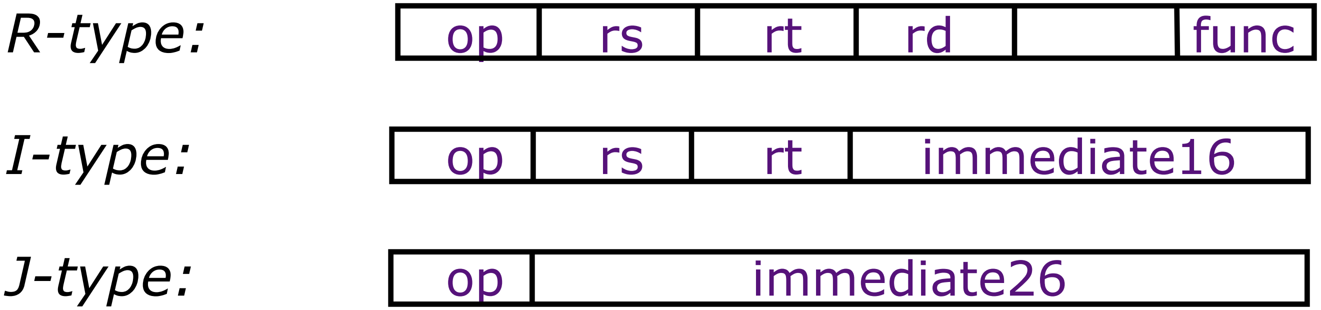


Should we always stall if the rs field matches some rd?

not every instruction writes a register ⇒ we

not every instruction reads a register ⇒ re

Source & Destination Registers



R-type: | op | rs | rt | rd | | func |

I-type: | op | rs | rt | immediate16 |

J-type: | op | immediate26 |

		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 ⇒ $(ws \neq 0)$
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}

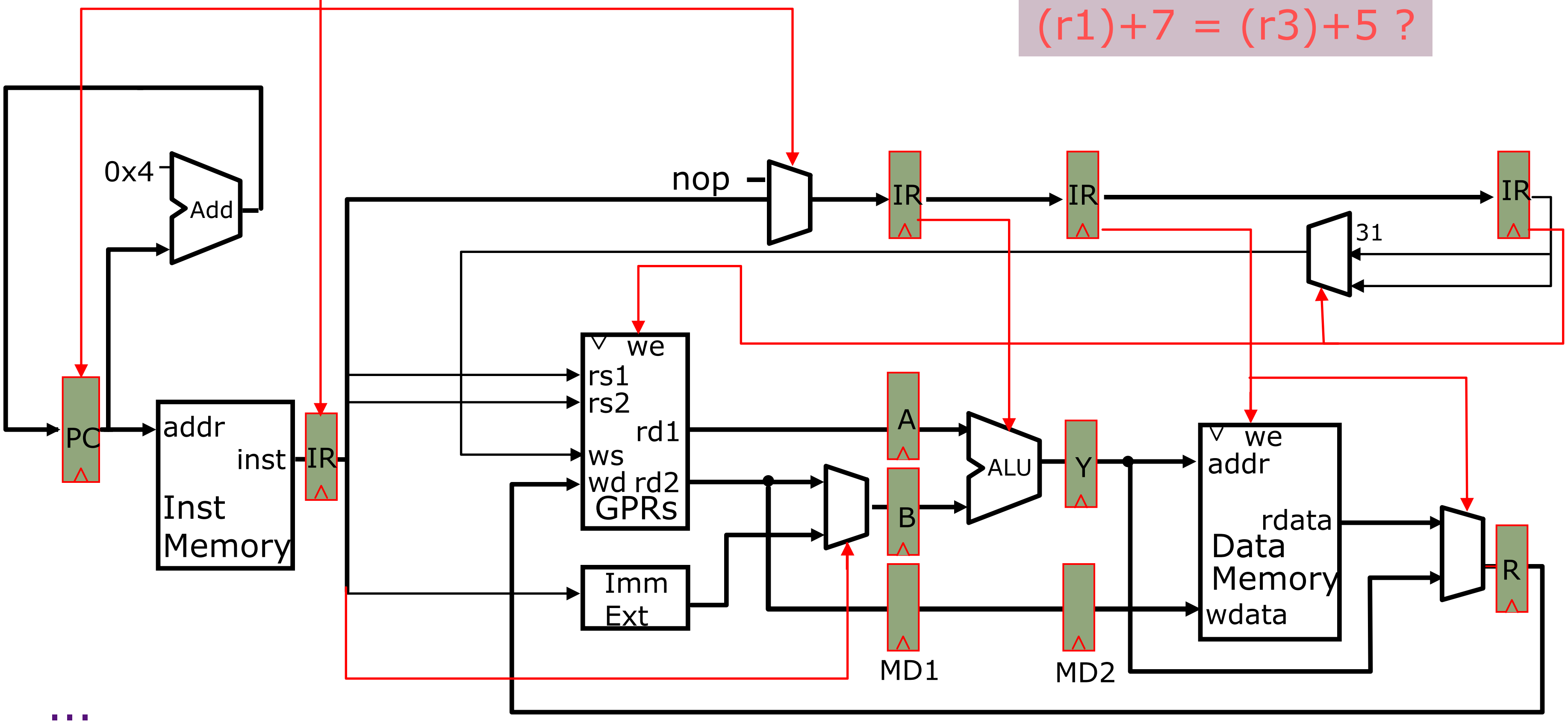
$$\begin{aligned}\text{stall} = ((&rs_D = ws_E)\cdot we_E + \\ &(rs_D = ws_M)\cdot we_M + \\ &(rs_D = ws_W)\cdot we_W)\cdot re1_D \quad + \\ ((&rt_D = ws_E)\cdot we_E + \\ &(rt_D = ws_M)\cdot we_M + \\ &(rt_D = ws_W)\cdot we_W)\cdot re2_D\end{aligned}$$

This is not the full story !

Hazards due to Loads & Stores

Stall Condition

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



...
M[(r1)+7] ← (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 are sometimes resolved in the pipeline and sometimes in the memory system itself.

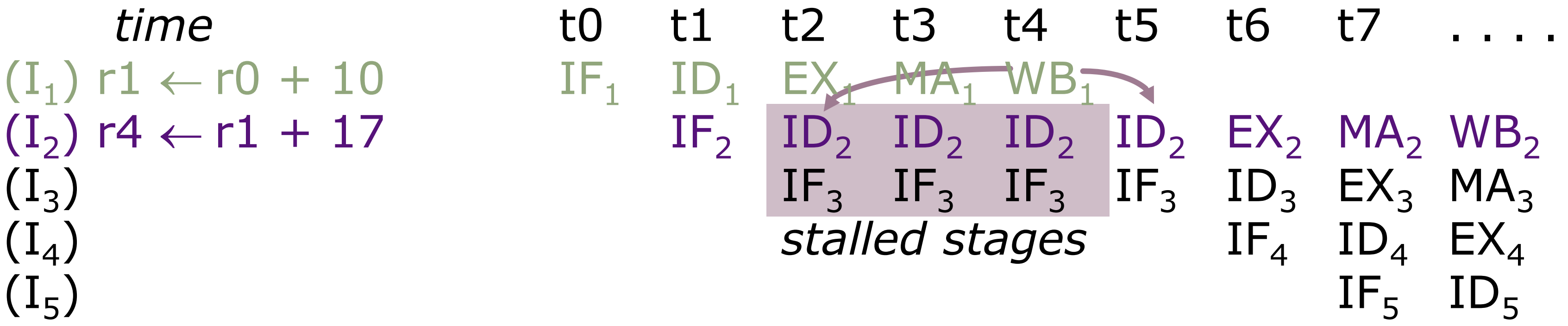
More on this later in the course.

Resolving Data Hazards (2)

Strategy 2:

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

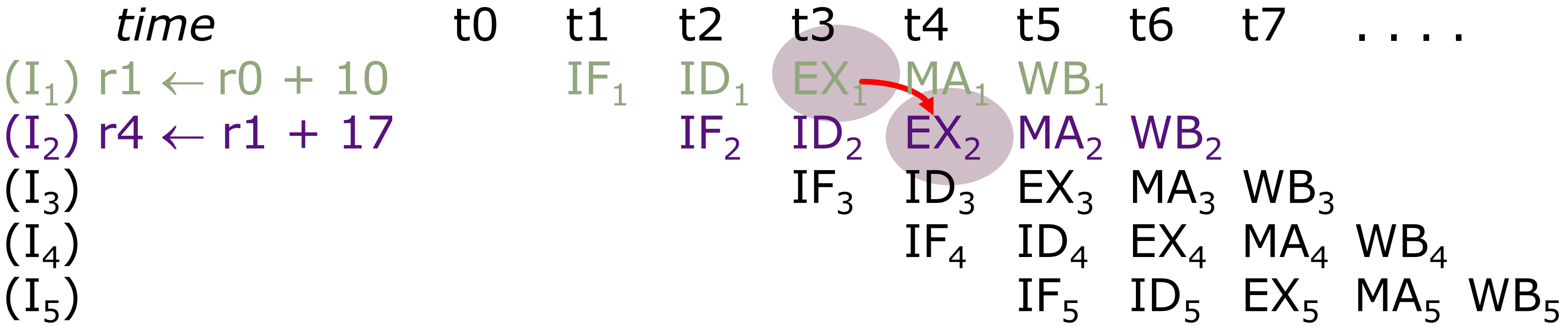
Bypassing



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 ⇒ *CPI > 1*

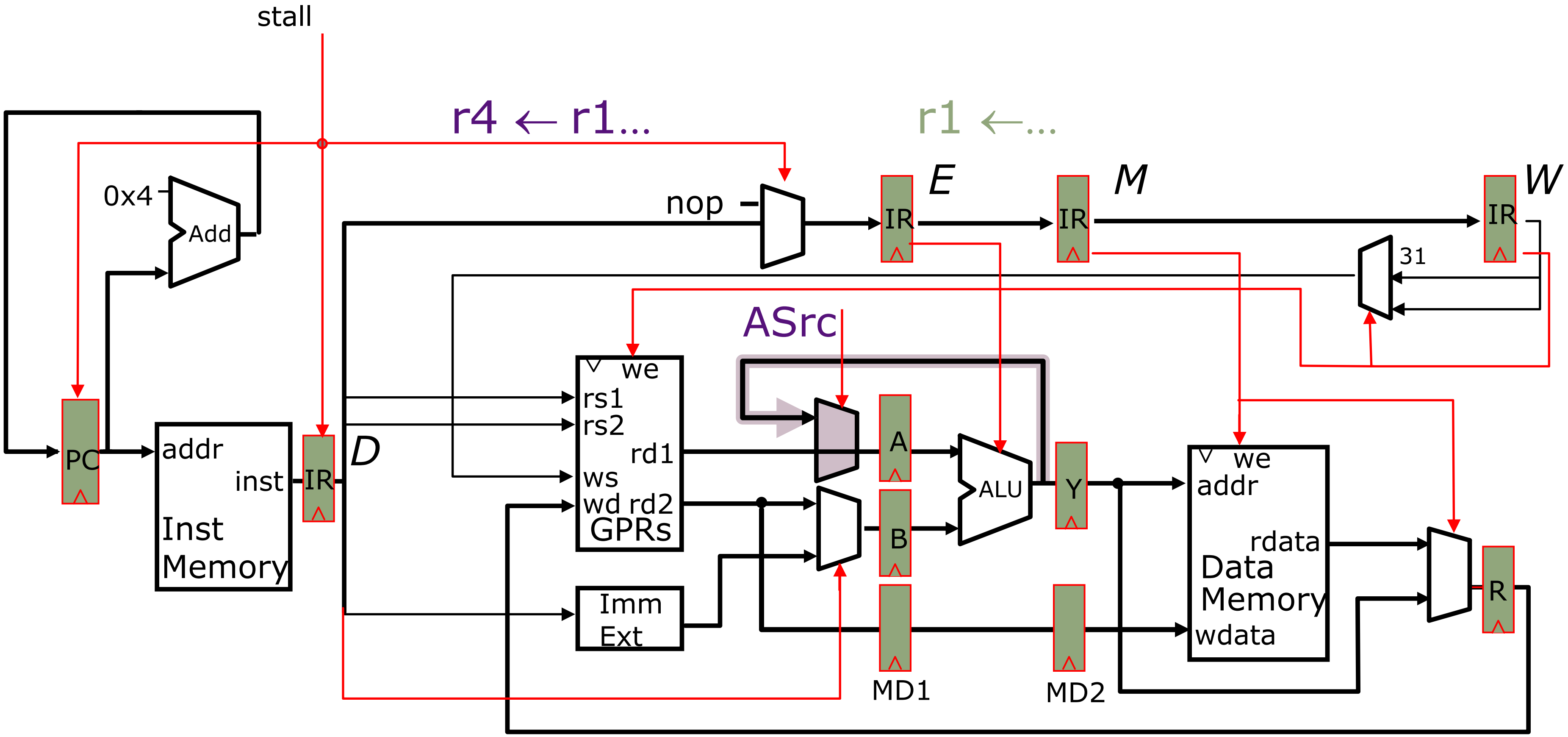
When is data actually available? At Execute



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

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) r1 ← r0 + 10

(I_2) r4 ← r1 + 17

yes

r1 ← M[r0 + 10]

r4 ← r1 + 17

no

JAL 500

r4 ← r31 + 17

no

The Bypass Signal

Deriving it from the Stall Signal

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

$+((rt_D = ws_E)\cdot we_E + (rt_D = ws_M)\cdot we_M + (rt_D = ws_W)\cdot we_W)\cdot re2_D$

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

we = *Case* opcode
ALU, ALUi, LW ⇒ $(ws \neq 0)$
JAL, JALR ⇒ on
... ⇒ off

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

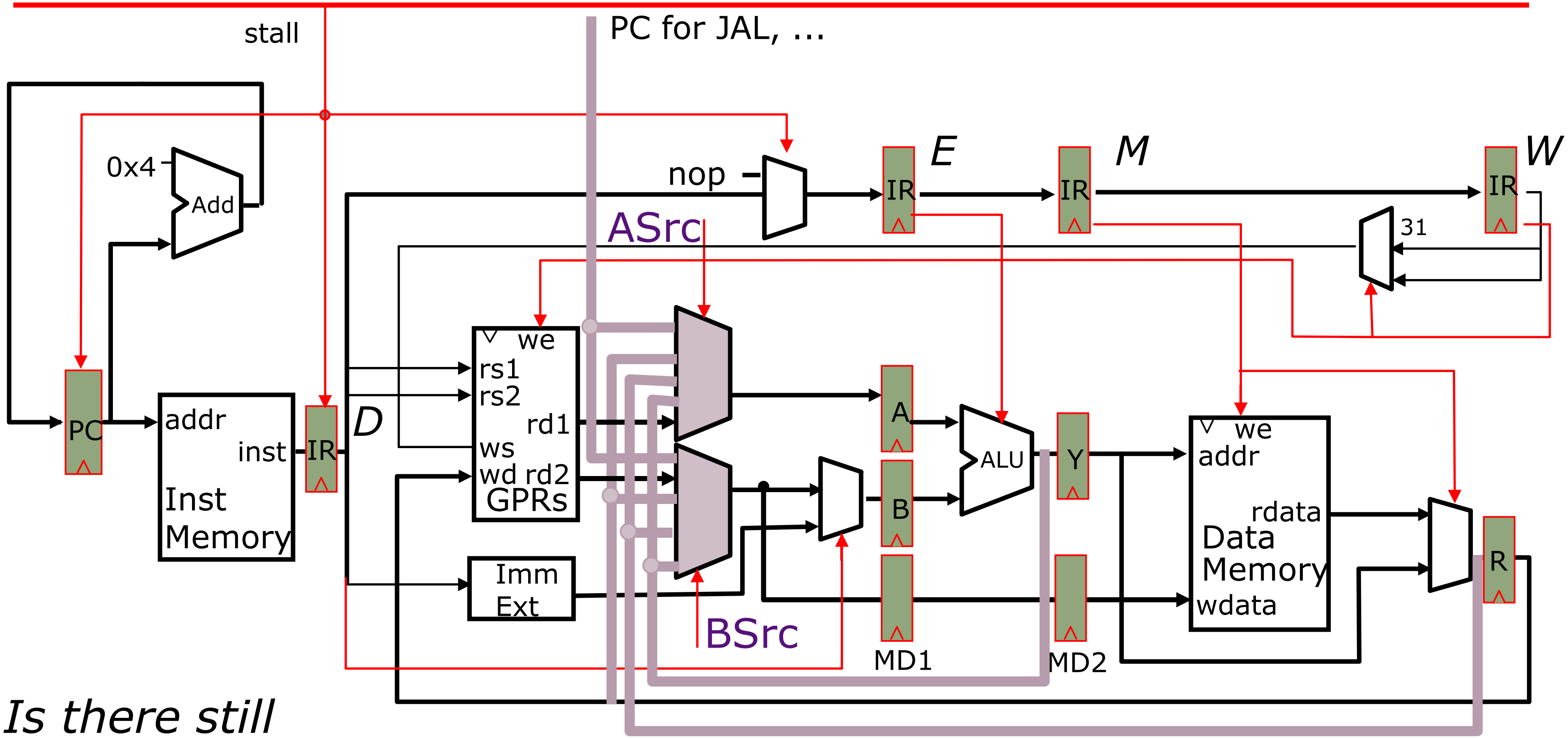
$$\begin{aligned} \text{we-bypass}_E &= \textit{Case}\ \text{opcode}_E \\ &\quad \text{ALU, ALUi} \Rightarrow (ws \neq 0) \\ &\quad \ldots \Rightarrow \text{off} \end{aligned}$$

$$\begin{aligned} \text{we-stall}_E &= \textit{Case}\ \text{opcode}_E \\ &\quad \text{LW} \Rightarrow (ws \neq 0) \\ &\quad \text{JAL, JALR} \Rightarrow \text{on} \\ &\quad \ldots \Rightarrow \text{off} \end{aligned}$$

$$\text{ASrc} = (rs_D = ws_E)\cdot\text{we-bypass}_E\cdot re1_D$$

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

Fully Bypassed Datapath



Is there still a need for the stall signal ?

$$\boxed{\begin{aligned} stall = \; & (rs_D = ws_E)\cdot(opcode_E = LW_E)\cdot(ws_E \neq 0)\cdot re1_D \\ & + (rt_D = ws_E)\cdot(opcode_E = LW_E)\cdot(ws_E \neq 0)\cdot re2_D \end{aligned}}$$

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

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) r3 ← (r2) + 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$_4$

time

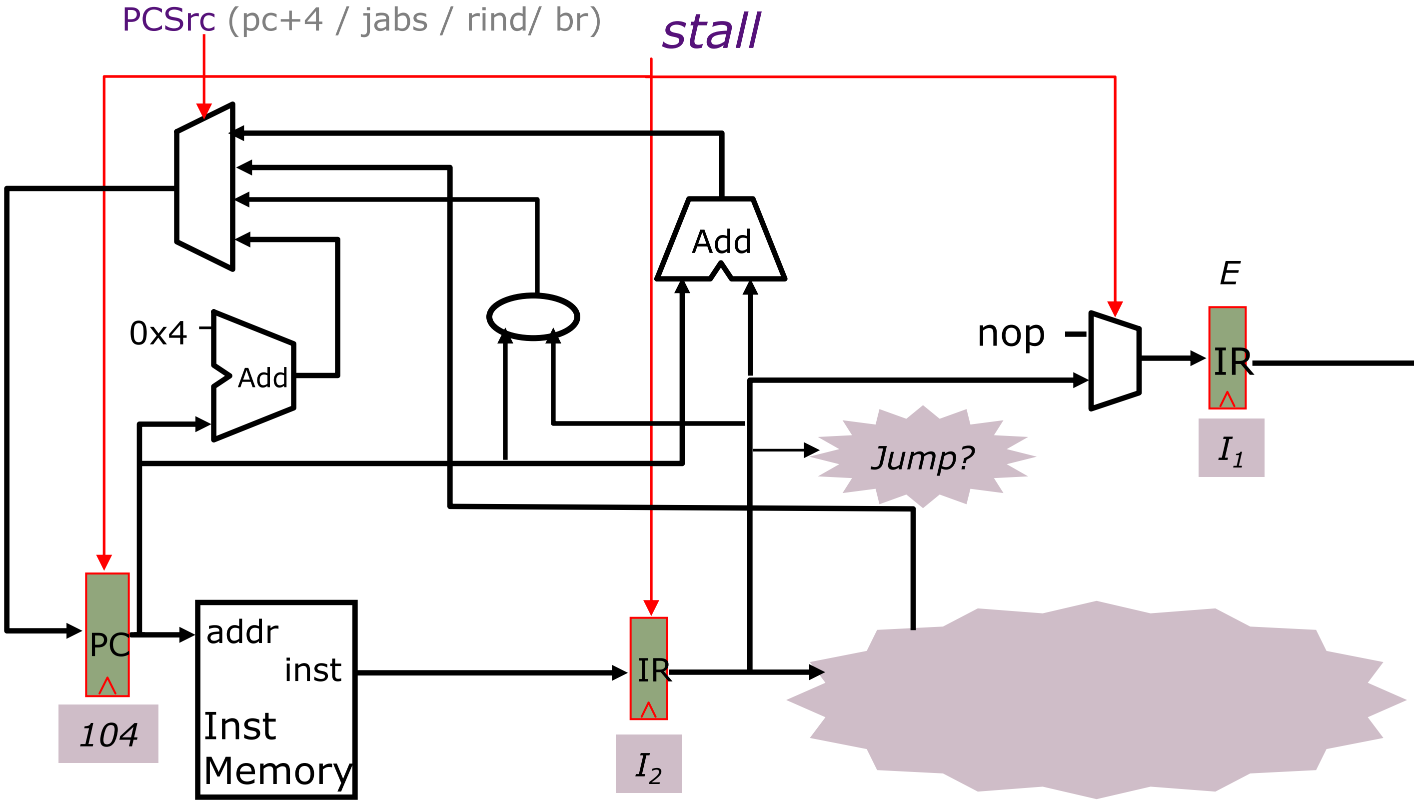
Resource Usage

	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? PC+4

Speculate NextPC is PC+4

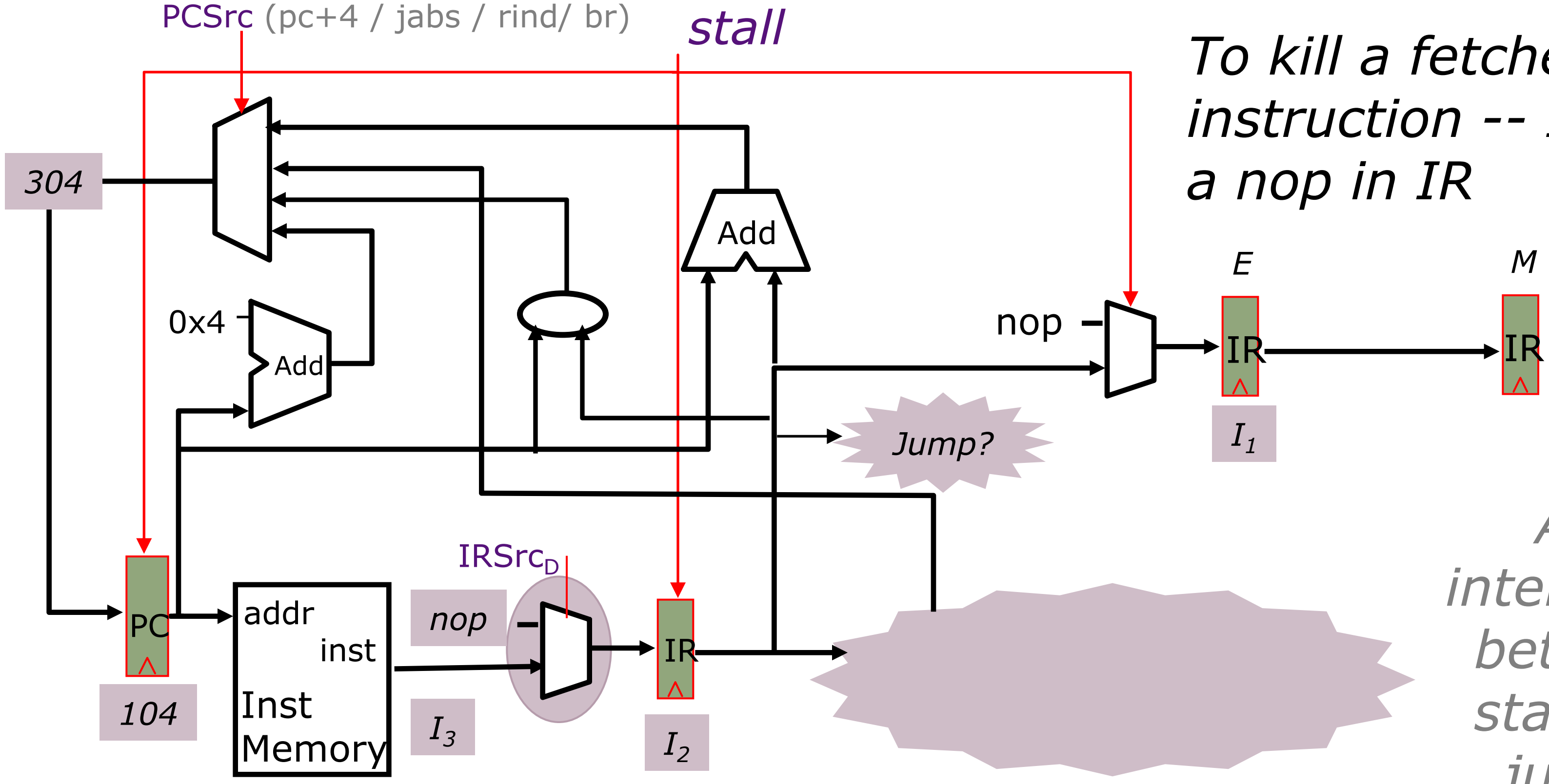


I_1	096	ADD	
I_2	100	J	200
I_3	~~104~~	~~ADD~~	kill
I_4	304	ADD	

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

How?

Pipelining Jumps



To kill a fetched instruction -- Insert a nop in IR

Any interaction between stall and jump?

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

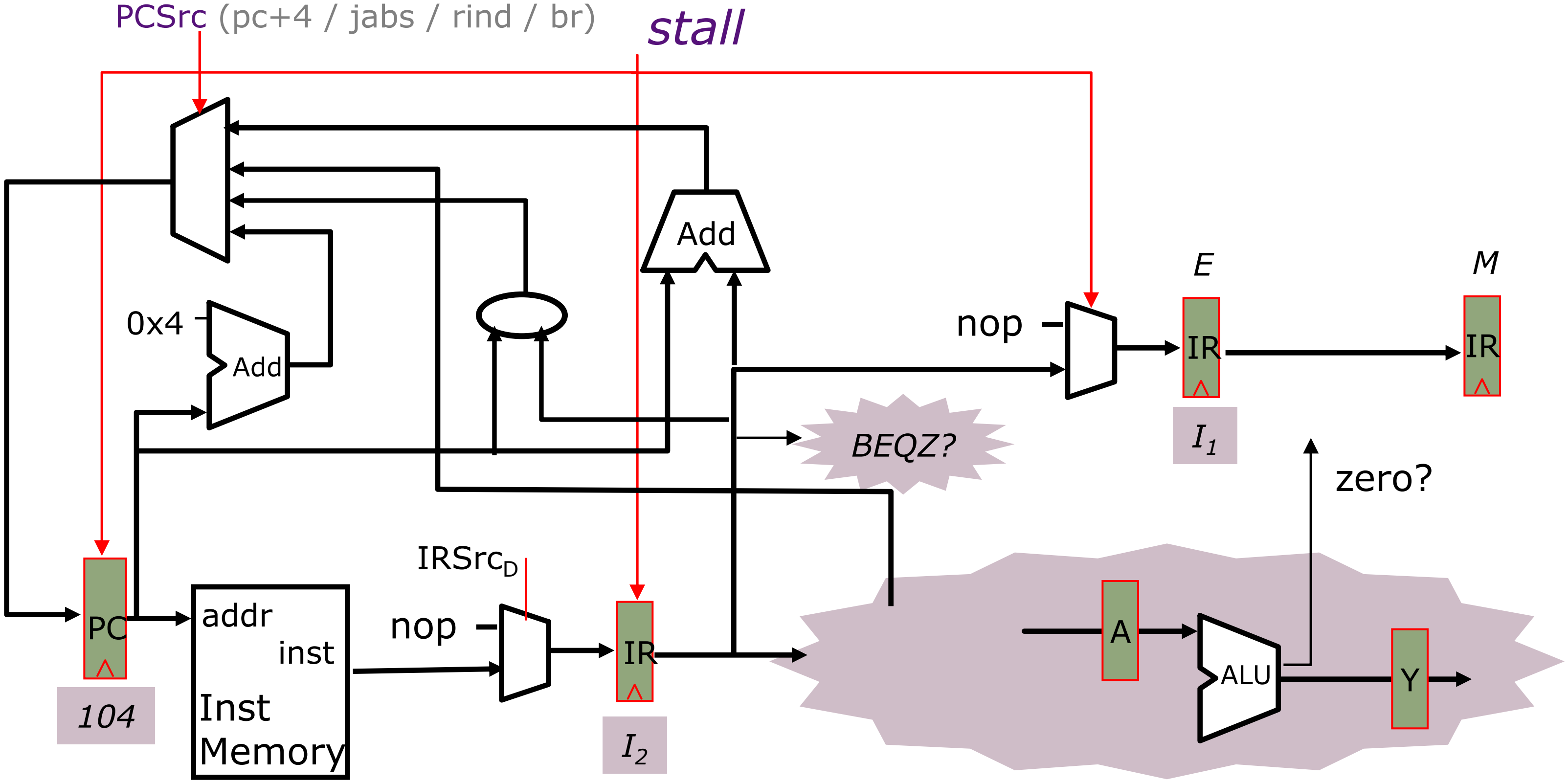
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: J 200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	nop	nop	nop	nop		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

time

Resource Usage		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

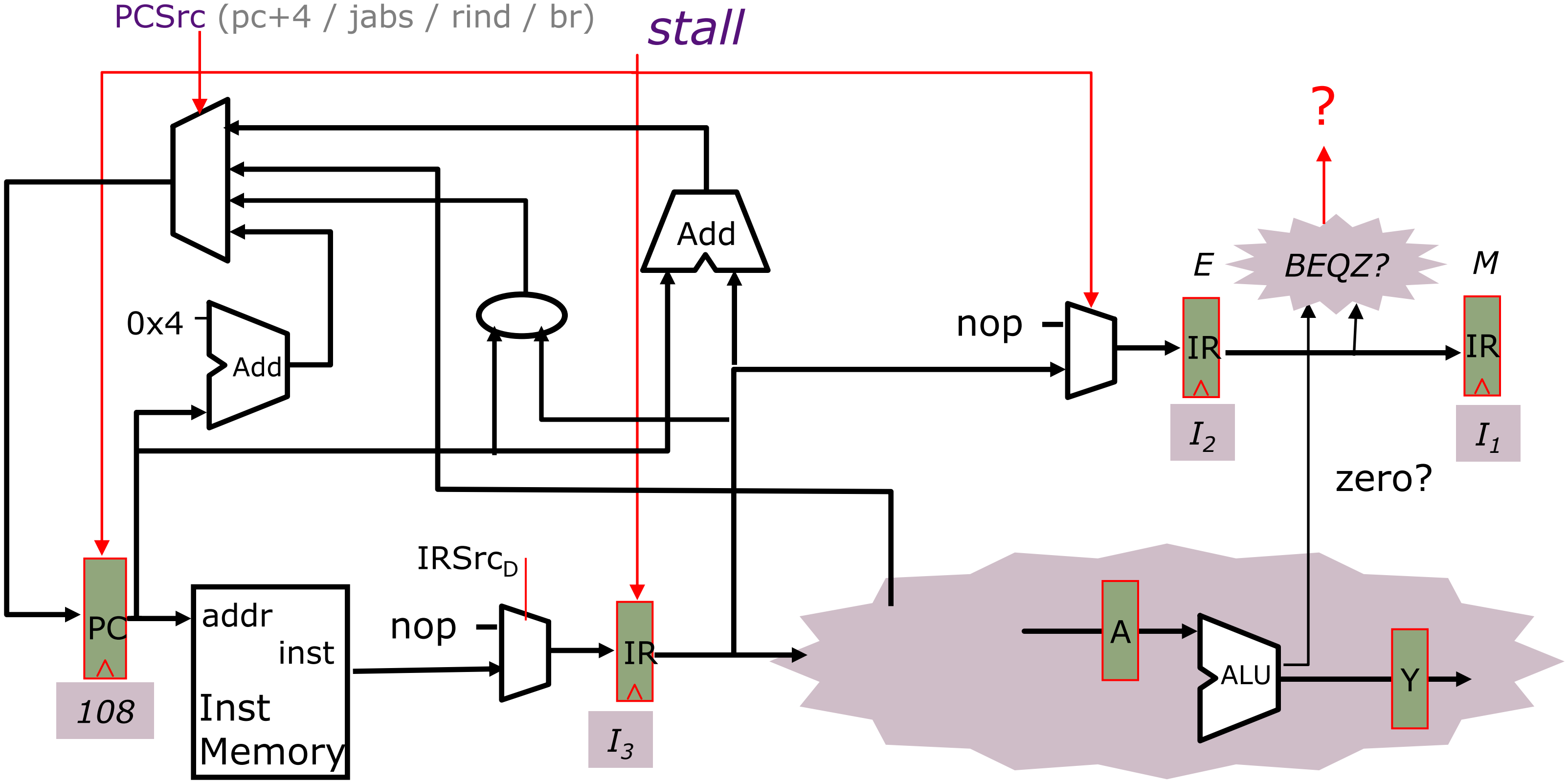


I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage?

Pipelining Conditional Branches



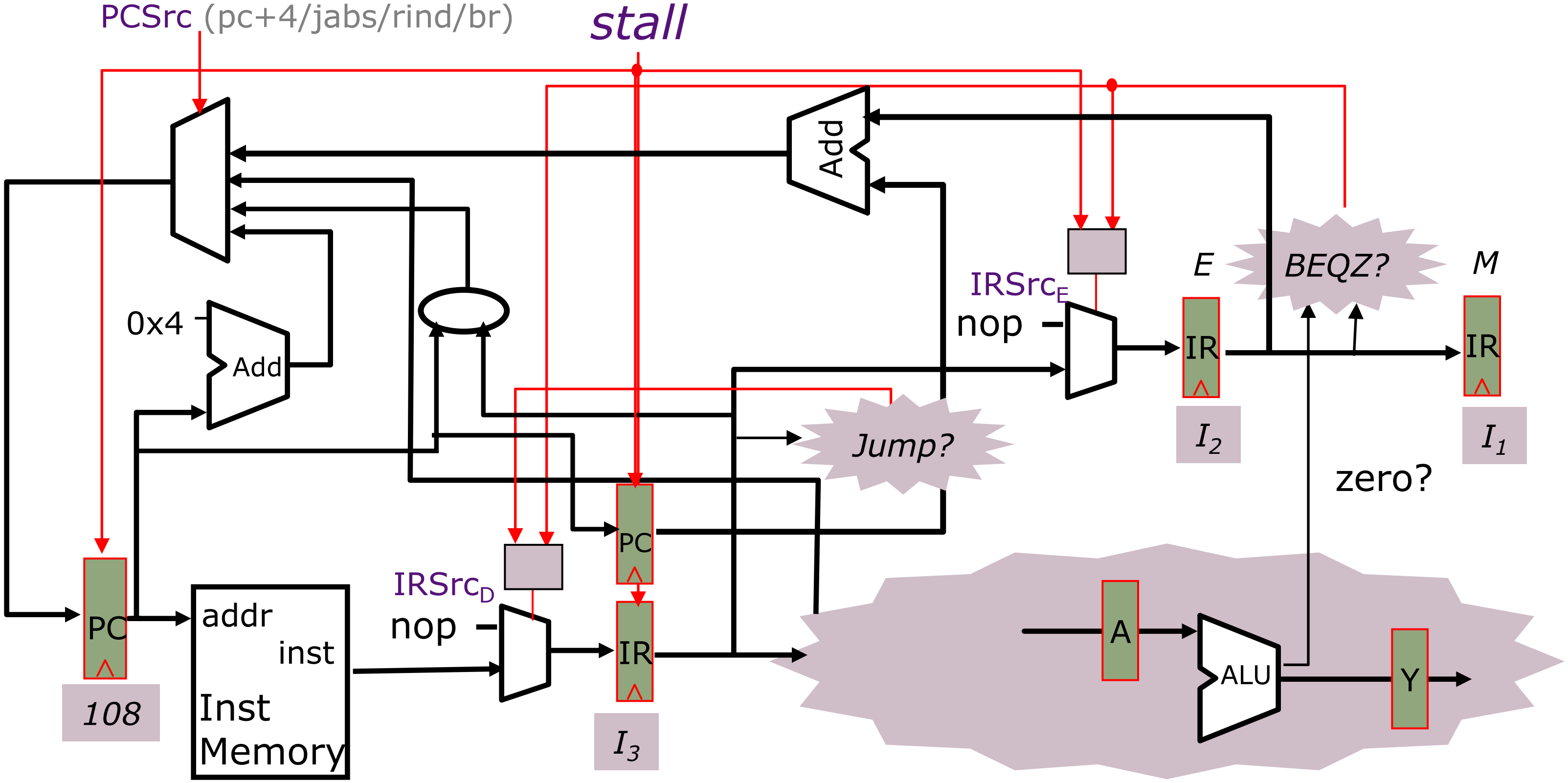
I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

If the branch is taken

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

⇒ *stall signal is not valid*

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

⇒ *stall signal is not valid*

New Stall Signal

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

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

Instruction at the decode stage is invalid

Control Equations for PC and IR Muxes

```
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$_D$
```

```
PCSrc = Case opcode$_E$
    BEQZ·z, BNEZ·!z     ⇒ br
    ...                 ⇒
            Case opcode$_D$
                J, JAL       ⇒ jabs
                JR, JALR     ⇒ rind
                ...          ⇒ pc+4
```

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

pc+4 is a speculative guess

nop ⇒ Kill
br/jabs/rind ⇒ 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: 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

time

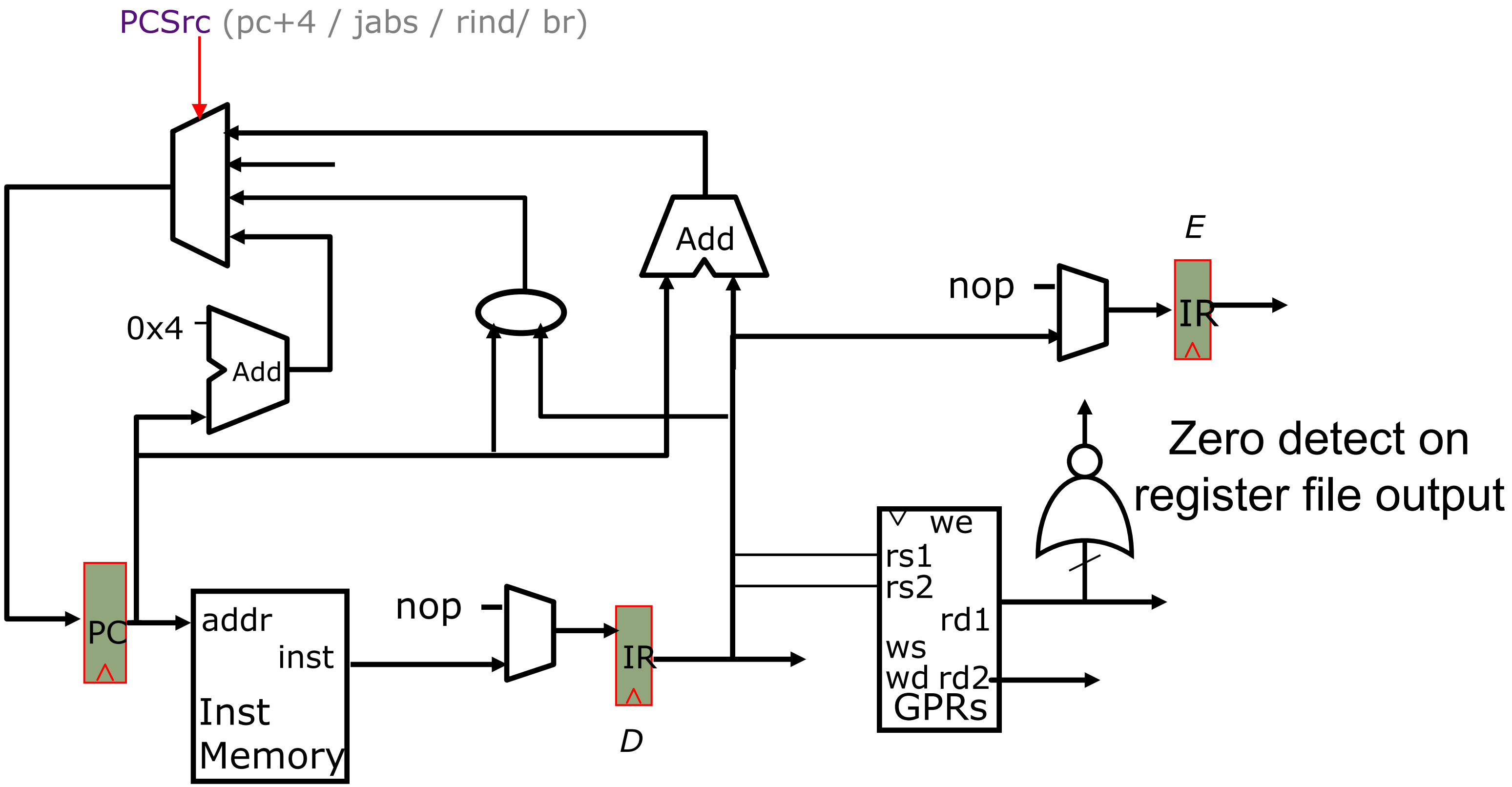
Resource Usage

	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
I_3	104	ADD
I_4	304	ADD

Delay slot instruction executed regardless of branch outcome

- 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.