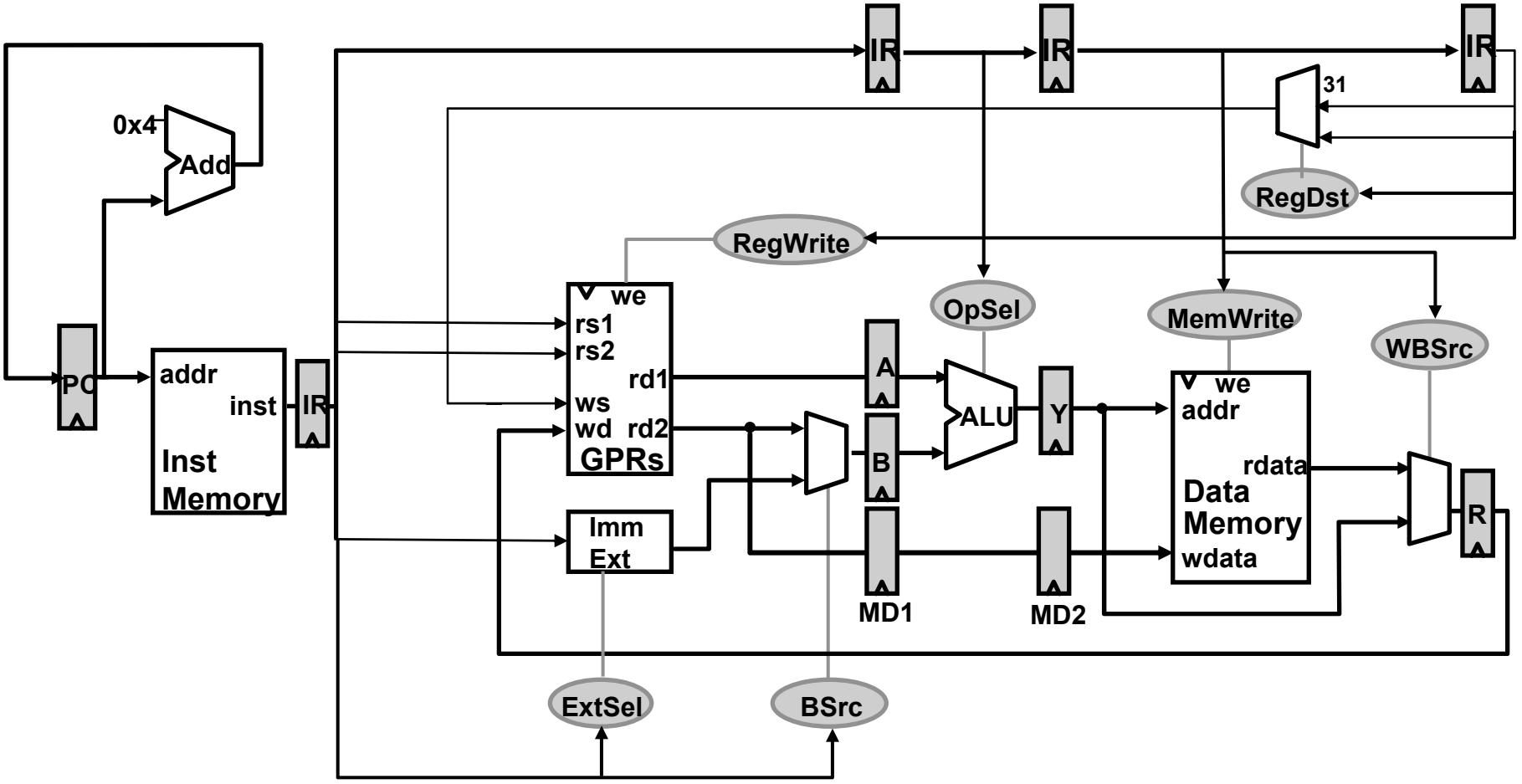
Pipeline Hazards

Krste Asanovic
Laboratory for Computer Science
M.I.T.

Pipelined DLX Datapath

without interlocks and jumps



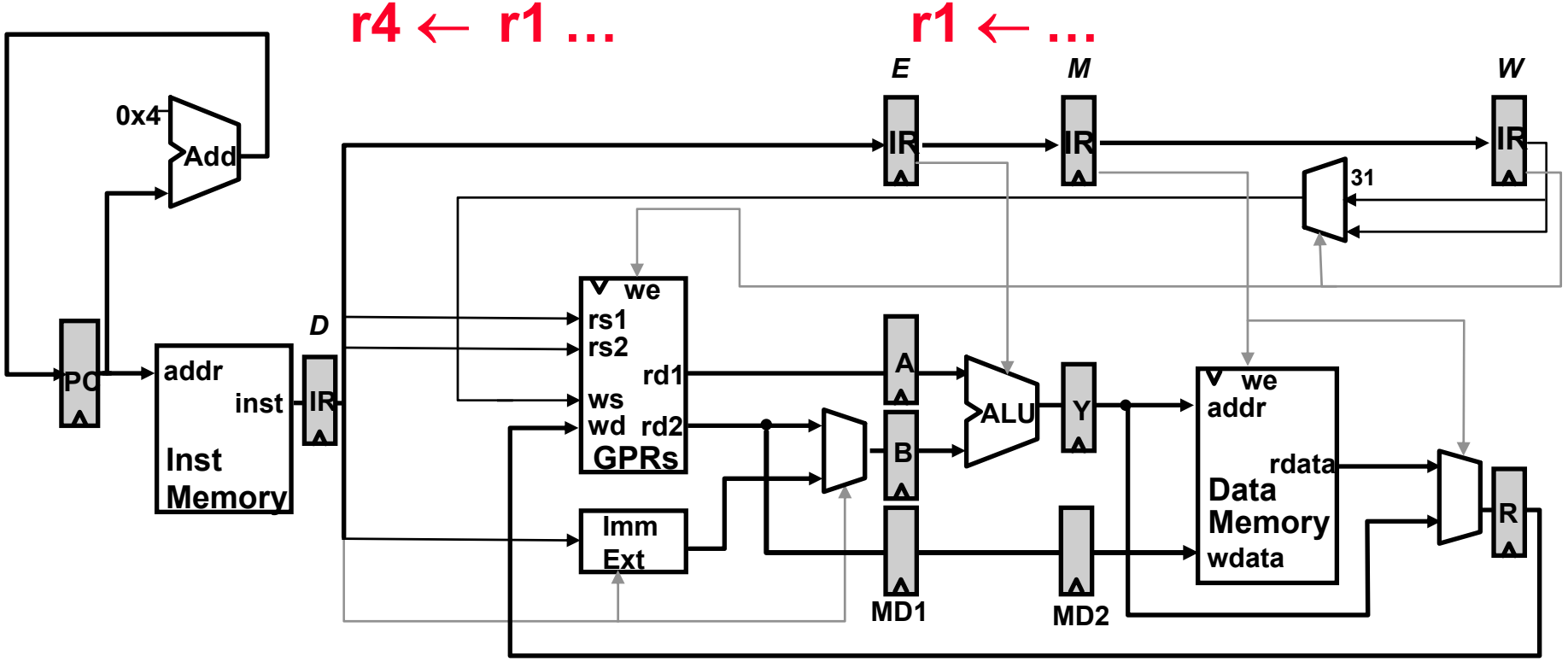
Data Hazards

...

r1 ← r0 + 10

r4 ← r1 + 17

...



Oops!

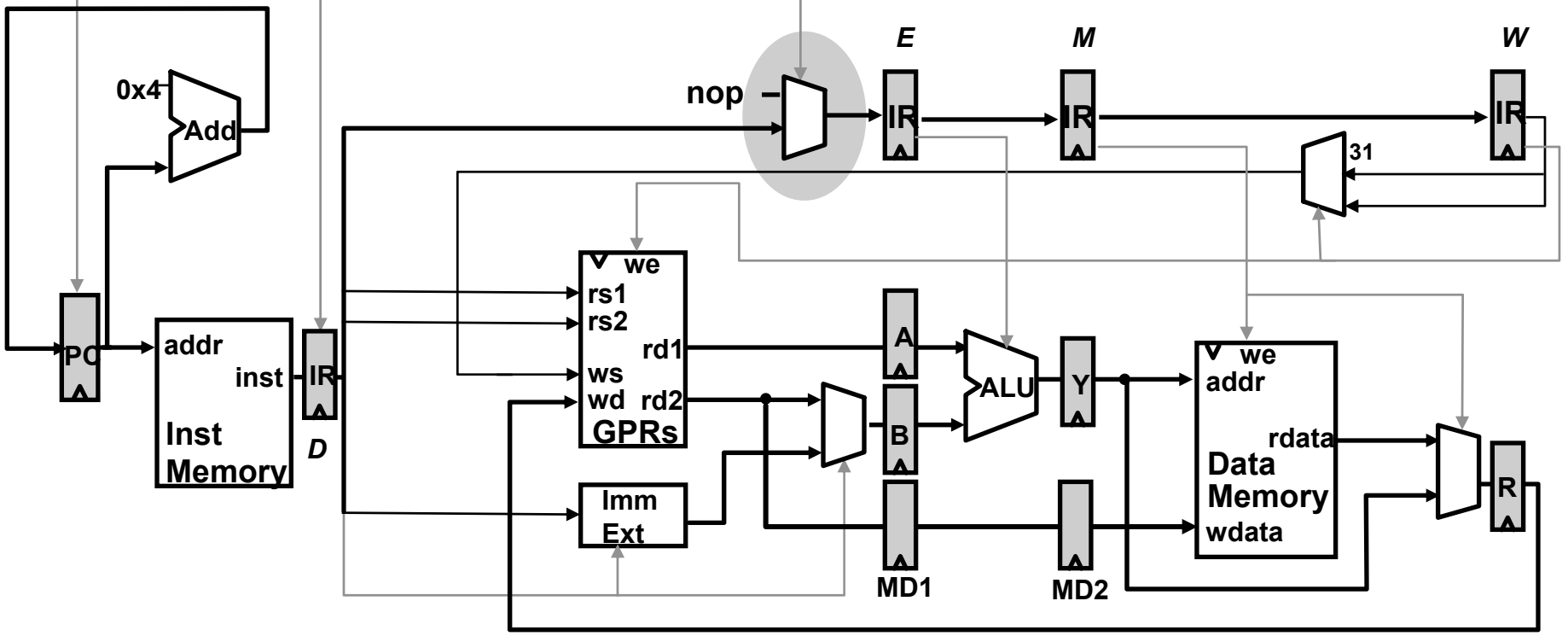
Resolving Data Hazards

1. Interlocks
   Freeze earlier pipeline stages until data becomes available

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

Interlocks to resolve Data Hazards

Stall Condition



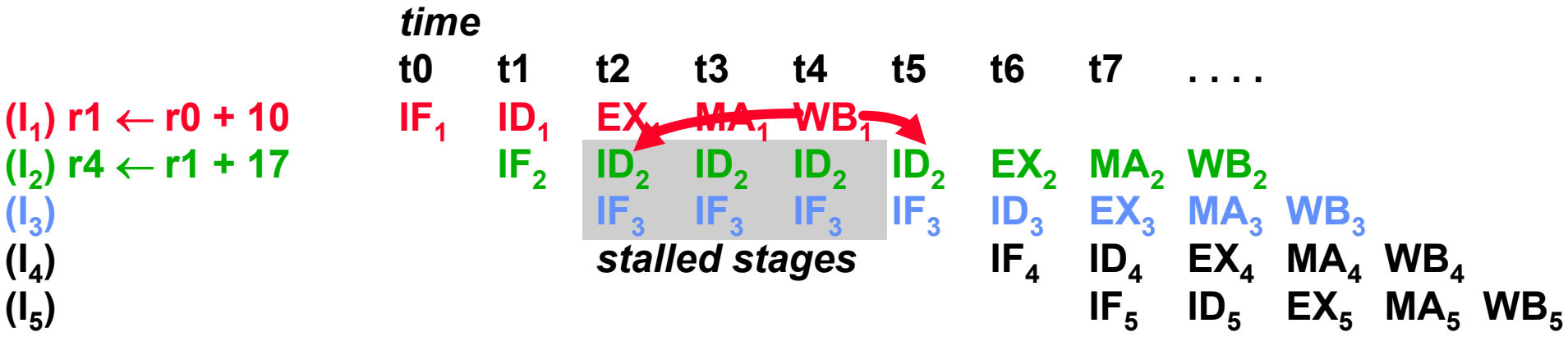
...

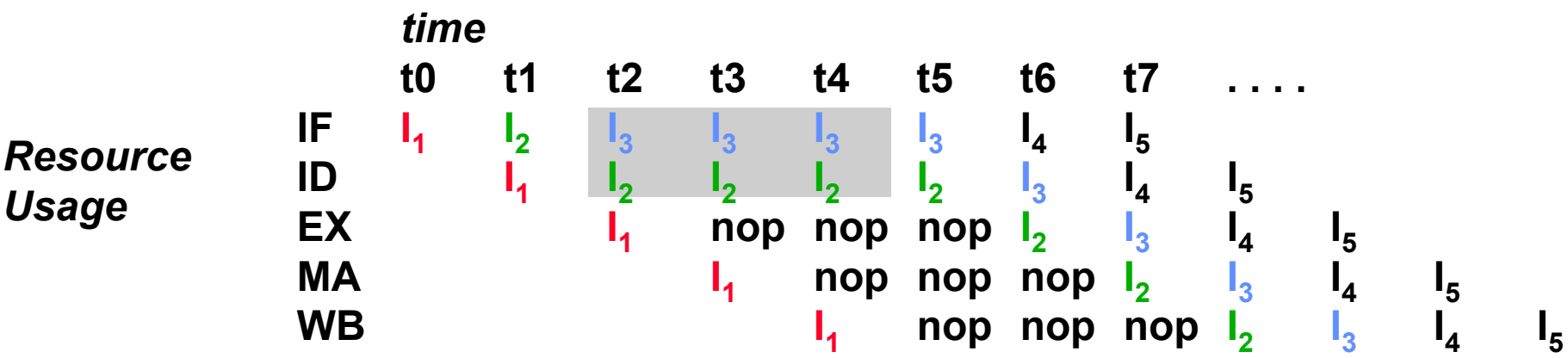
r1 ← r0 + 10

r4 ← r1 + 17

...

Stalled Stages and Pipeline Bubbles

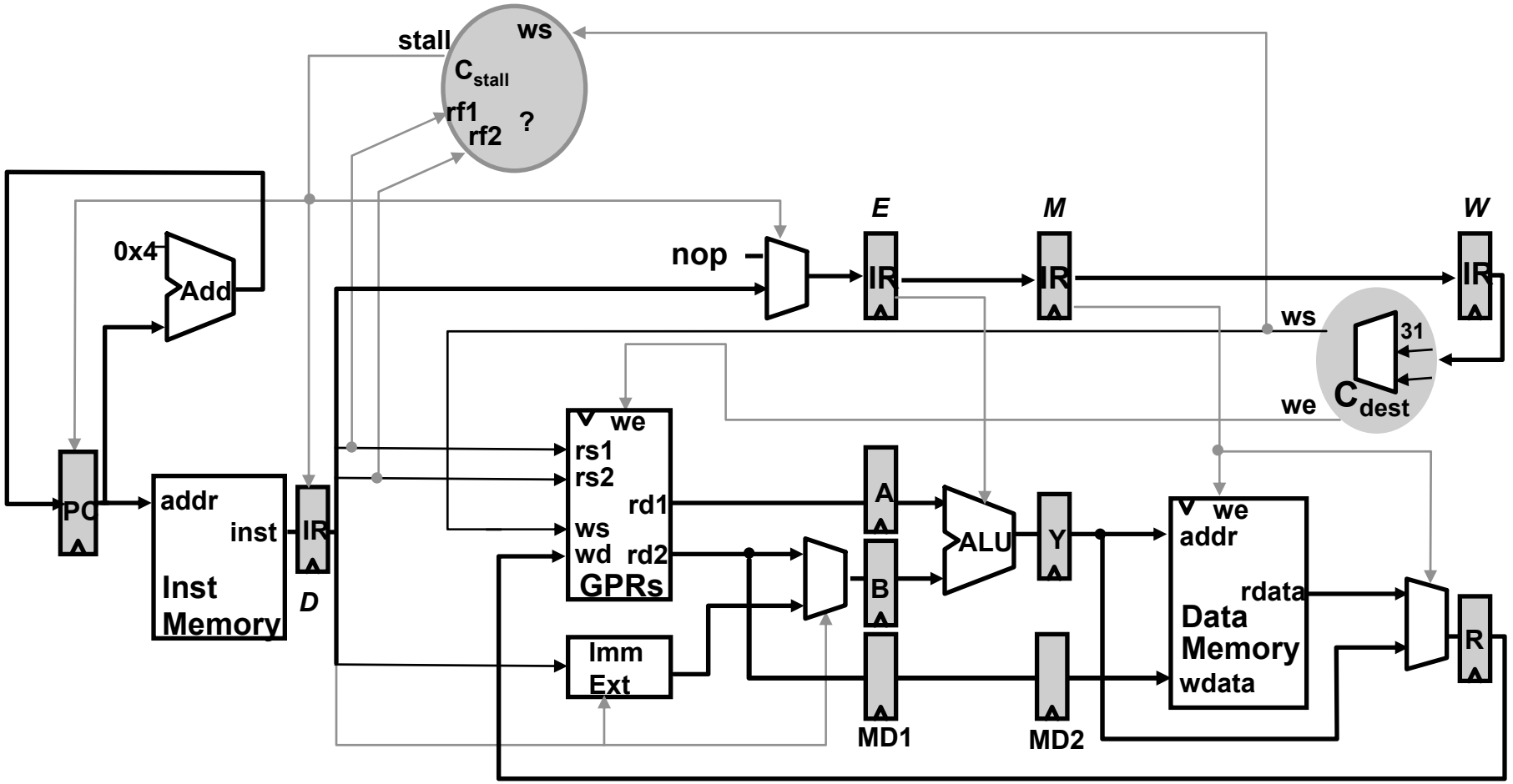




nop ⇒ *pipeline bubble*

Interlock Control Logic

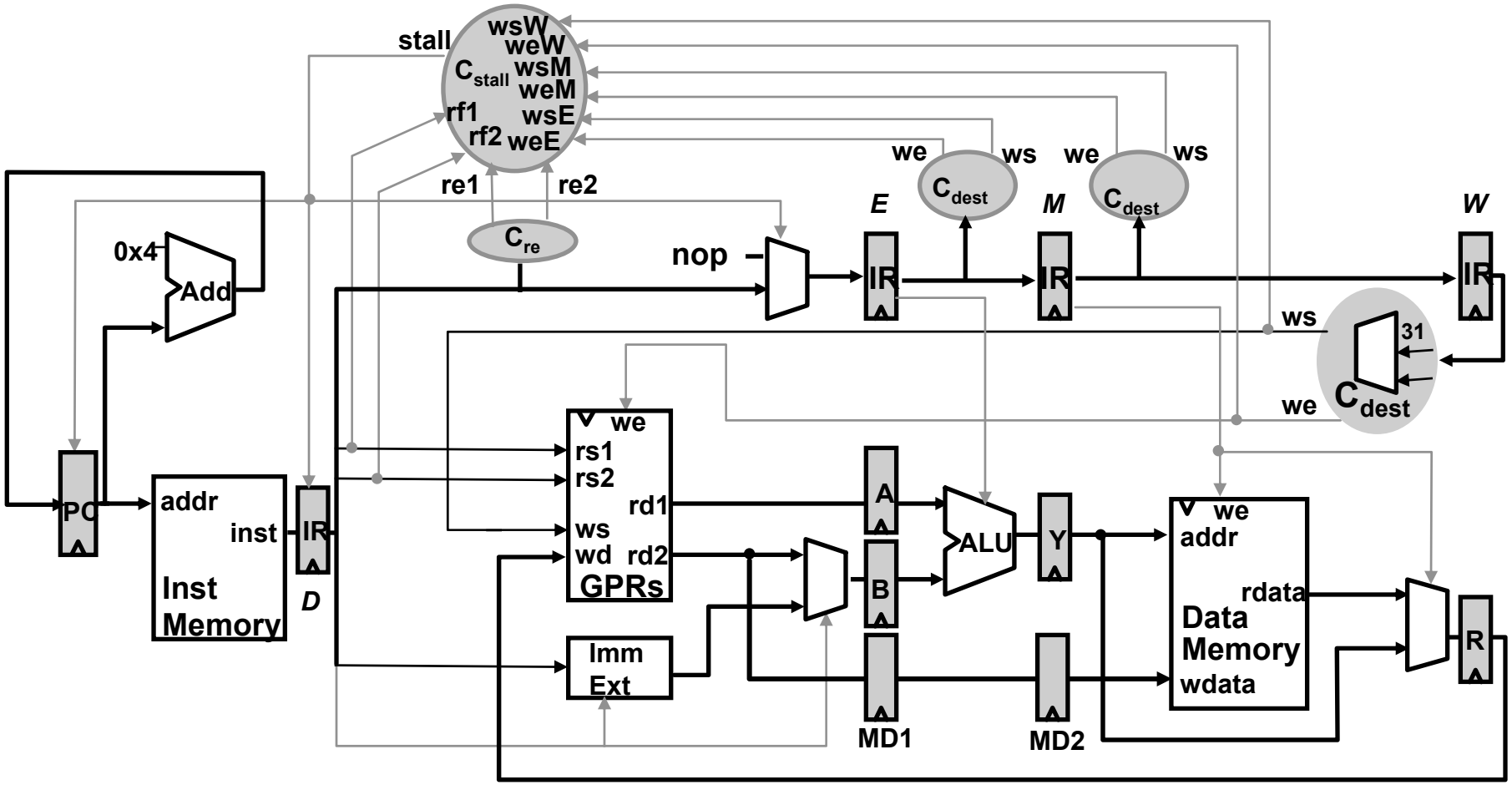
worksheet



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

Interlock Control Logic

ignoring jumps & branches

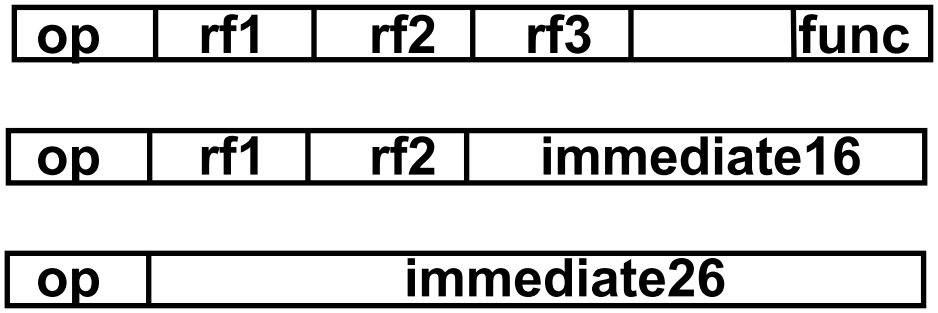


Should we always stall if the rs field matches some rd?

not every instruction writes registers ⇒ we

not every instruction reads registers ⇒ re

Source & Destination Registers



		source(s)	*destination*
ALU	rf3 ← rf1 func rf2	rf1, rf2	rf3
ALUi	rf2 ← rf1 op imm	rf1	rf2
LW	rf2 ← M [rf1 + imm]	rf1	rf2
SW	M [rf1 + imm] ← rf2	rf1, rf2	
B__Z	*cond* rf1		
	true: PC ← PC + 4 + imm	rf1	
	false: PC ← PC + 4	rf1	
J	PC ← PC + 4 + imm		
JAL	r31 ← PC, PC ← PC + 4 + imm	S	31
JR	PC ← rf1	rf1	
JALR	r31 ← PC, PC ← rf1	rf1	31

Deriving the Stall Signal

C_{dest}

ws = *Case* opcode

ALU	⇒ rf3
ALUi, LW	⇒ rf2
JAL, JALR	⇒ 31

we = *Case* opcode

ALU, ALUi, LW, JAL, JALR	⇒ on
...	⇒ off

C_{re}

re1 = *Case* opcode

ALU, ALUi,	⇒ on
	⇒ off

re2 = *Case* opcode

	⇒ on
	⇒ off

stall =

Stall if the *source registers* of the instruction in the decode stage matches the *destination register* of the *uncommitted* instructions.

The Stall Signal

C_{dest}

ws = *Case* opcode

ALU	$\Rightarrow$ rf3
ALUi, LW	$\Rightarrow$ rf2
JAL, JALR	$\Rightarrow$ 31

we = *Case* opcode

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

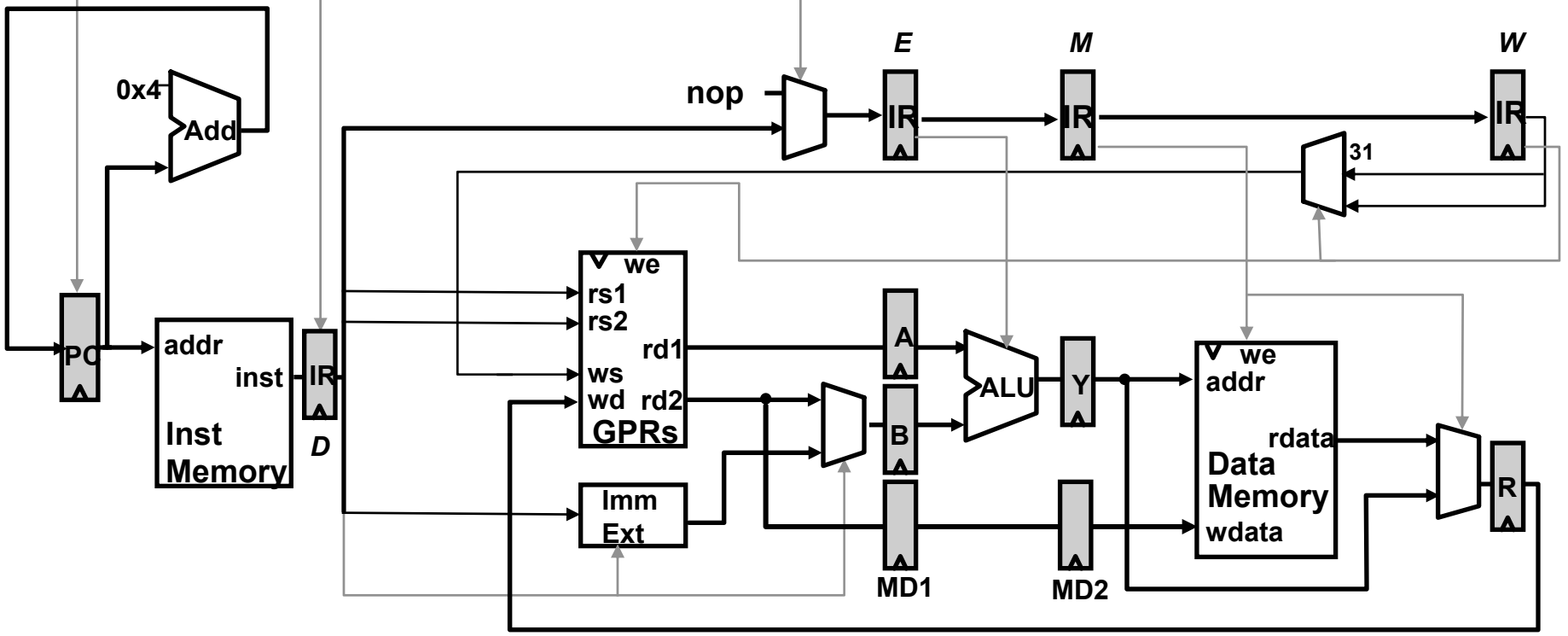
C_{stall}

$$stall = ((rf1_D = ws_E).we_E + (rf1_D = ws_M).we_M + (rf1_D = ws_W).we_W) \,.\, re1_D + ((rf2_D = ws_E).we_E + (rf2_D = ws_M).we_M + (rf2_D = ws_W).we_W) \,.\, re2_D$$

This is not the full story !

Hazards due to Loads & Stores

Stall Condition



...

M[r1+7] ← r2

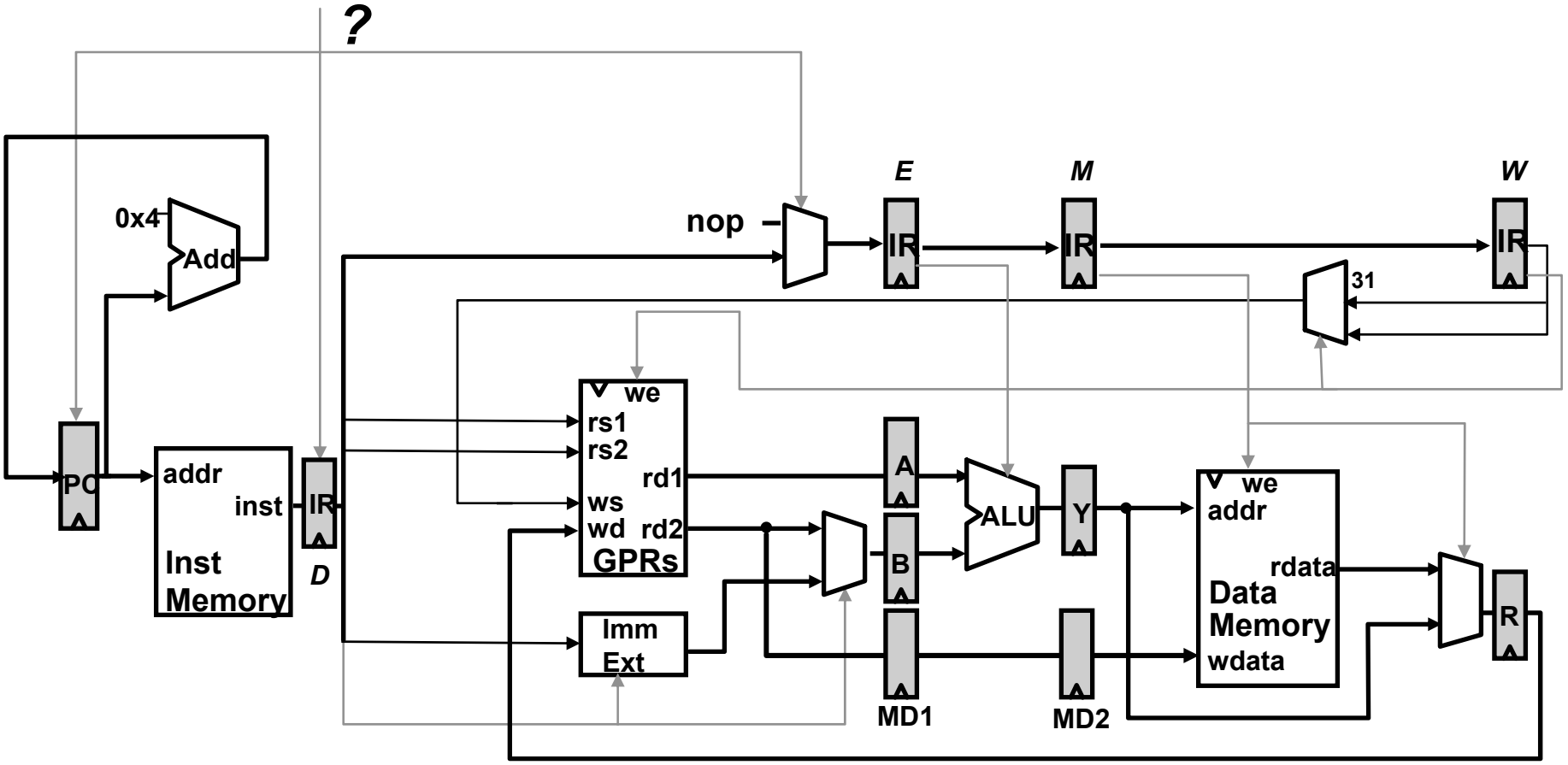
r4 ← M[r3+5]

...

Is there any possible data hazard in this instruction sequence?

Hazards due to Loads & Stores

depends on the memory system



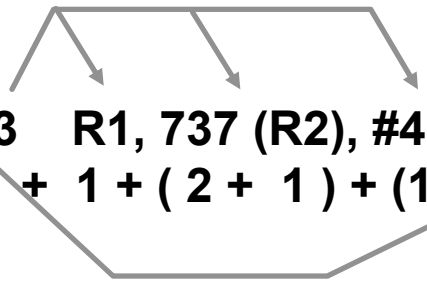
M[r1+7] ← r2
r4 ← M[r3+5]
...

r1+7 = r3+5 ⇒ *data hazard*

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

VAX: A Complex Instruction Set

Addressing Mode	*Syntax*	*Length in bytes*
Literal	#value	1 (6-bit signed value)
Immediate	#value	1 + |immediate|
Register	Rn	1
Register deferred	(Rn)	1
Byte/word/long displacement	disp(Rn)	1 + |displacement|
Byte/word/long displacement deferred	@disp(Rn)	1 + |displacement|
Scaled (Indexed)	base mode (Rx)	1 + |base addressing mode|
Autoincrement	(Rn)+	1
Autodecrement	-(Rn)	1
Autoincrement deferred	@(Rn)+	1



ADDL3 R1, 737 (R2), #456 ≡ R1 <-- M[(R2)+737] + 456

1 + 1 + (2 + 1) + (1+ 4) = 10 bytes !

VAX Extremes

Long instructions, e.g.,

```
emodh #1.2334, 70000(r1)[r2],
      #2.3456, 80000(r1)[r2],
      90000(r1)[r2]
```

- 54 byte encoding (rumors of even longer instruction sequences...)

Memory referencing behavior

- In worst case, a single instruction could require at least 41 different memory pages to be resident in memory to complete execution
- Doesn't include string instructions which were designed to be interruptable

Reduced Instruction Set Computers

(Cocke, IBM; Patterson, UC Berkeley; Hennessy, Stanford)

Compilers have difficulty using complex instructions

- VAX: 60% of microcode for 20% of instructions, only responsible for 0.2% execution time
- IBM experiment retargets 370 compiler to use simple subset of ISA

=> Compiler generated faster code!

Simple instruction sets don't need microcode

- Use fast memory near processor as cache, not microcode storage

Design ISA for simple pipelined implementation

- Fixed length, fixed format instructions
- Load/store architecture with up to one memory access/instruction
- Few addressing modes, synthesize others with code sequence
- Register-register ALU operations
- Delayed branch

MIPS R2000
(One of first commercial RISCs, 1986)

Load/Store architecture

- 32x32-bit GPR (R0 is wired), HI & LO SPR (for multiply/divide)
- 74 instructions
- Fixed instruction size (32 bits), only 3 formats
- PC-relative branches, register indirect jumps
- Only base+displacement addressing mode
- No condition bits, compares write GPRs, branches test GPRs
- Delayed loads and branches

Five-stage instruction pipeline

- Fetch, Decode, Execute, Memory, Write Back
- CPI of 1 for register-to-register ALU instructions
- 8 MHz clock
- Tightly-coupled off-chip FP accelerator (R2010)

RISC/CISC Comparisons
(late 80s/early 90s)

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

R2000 vs VAX 8700 [Bhandarkar and Clark, '91]

- R2000 has ~2.7x advantage with equivalent technology

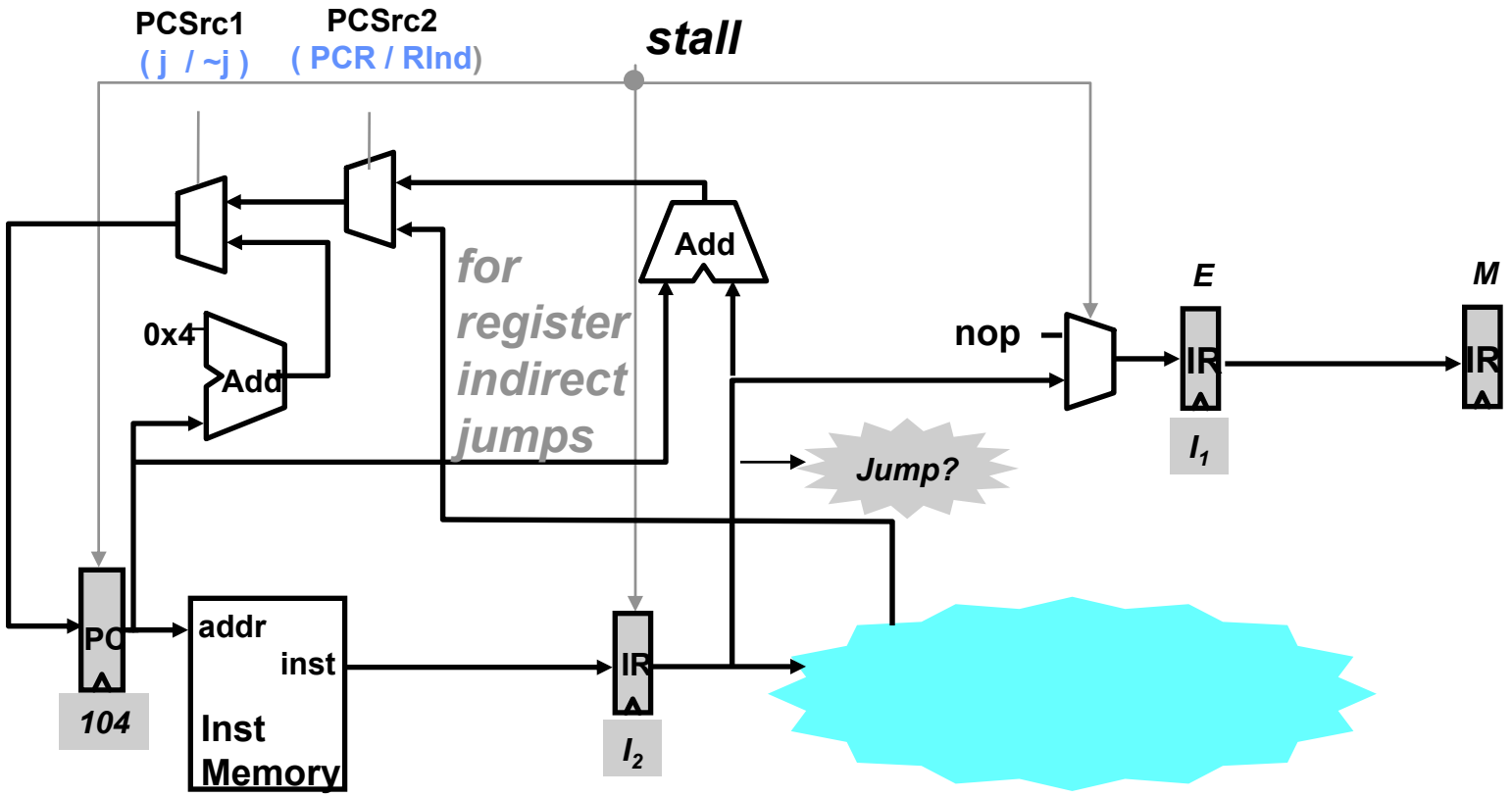
Intel 80486 vs Intel i860 (both 1989)

- Same company, same CAD tools, same process
- i860 2-4x faster - even more on some floating-point tasks

DEC nVAX vs Alpha 21064 (both 1992)

- Same company, same CAD tools, same process
- Alpha 2-4x faster

Complications due to Jumps



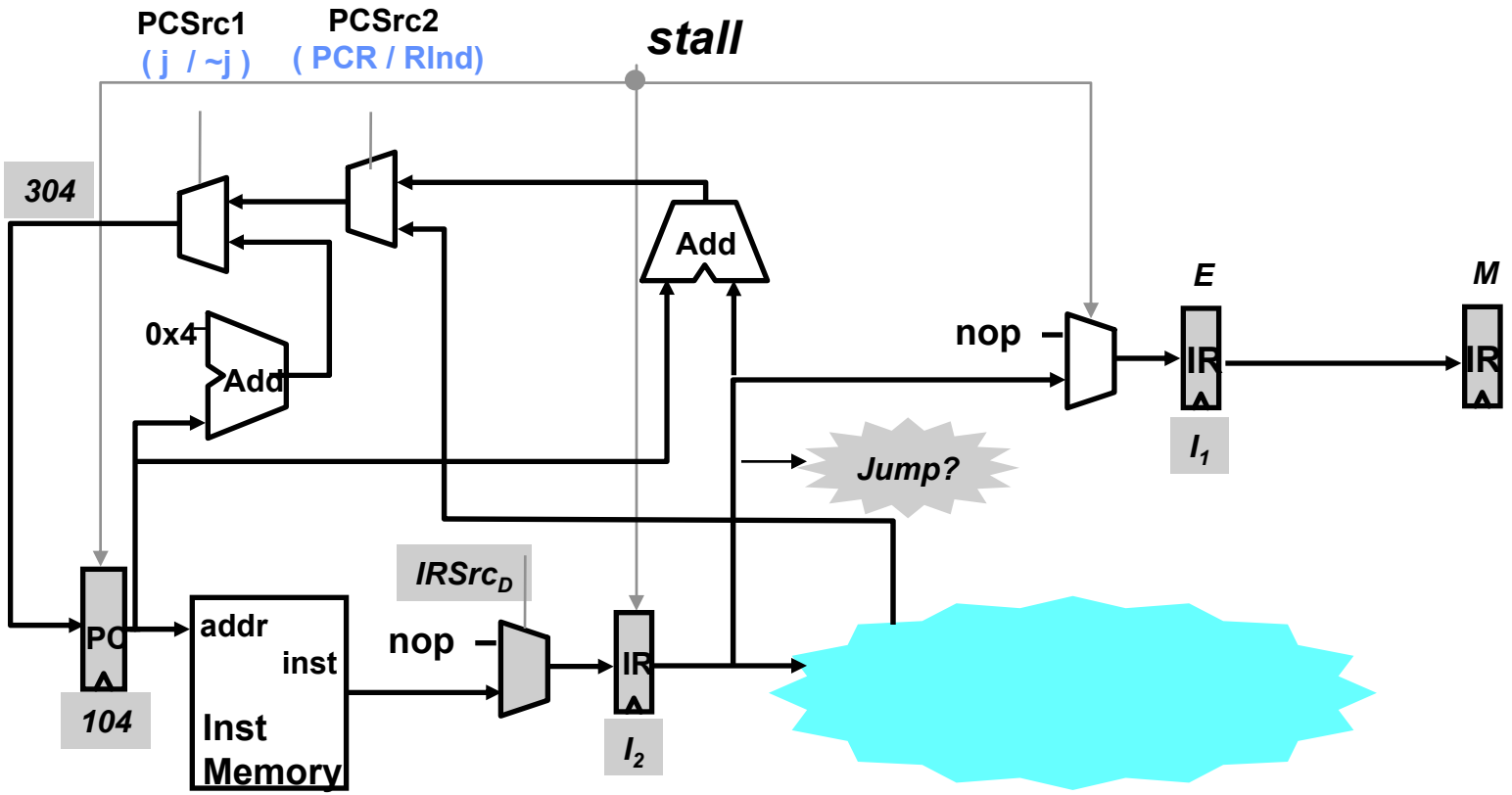
I_1	096	ADD
I_2	100	J +200
~~I_3~~	~~104~~	~~ADD~~
I_4	304	SUB

kill

A jump instruction kills (not stalls) the following instruction

How?

Pipelining Jumps



Killing the fetched instruction:
Insert a mux before IR

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

kill

$IRSrc_D$ = *Case* $opcode_D$
J, JAL ⇒ nop
... ⇒ IM

Any interaction between stall and jump?

Jump Pipeline Diagrams

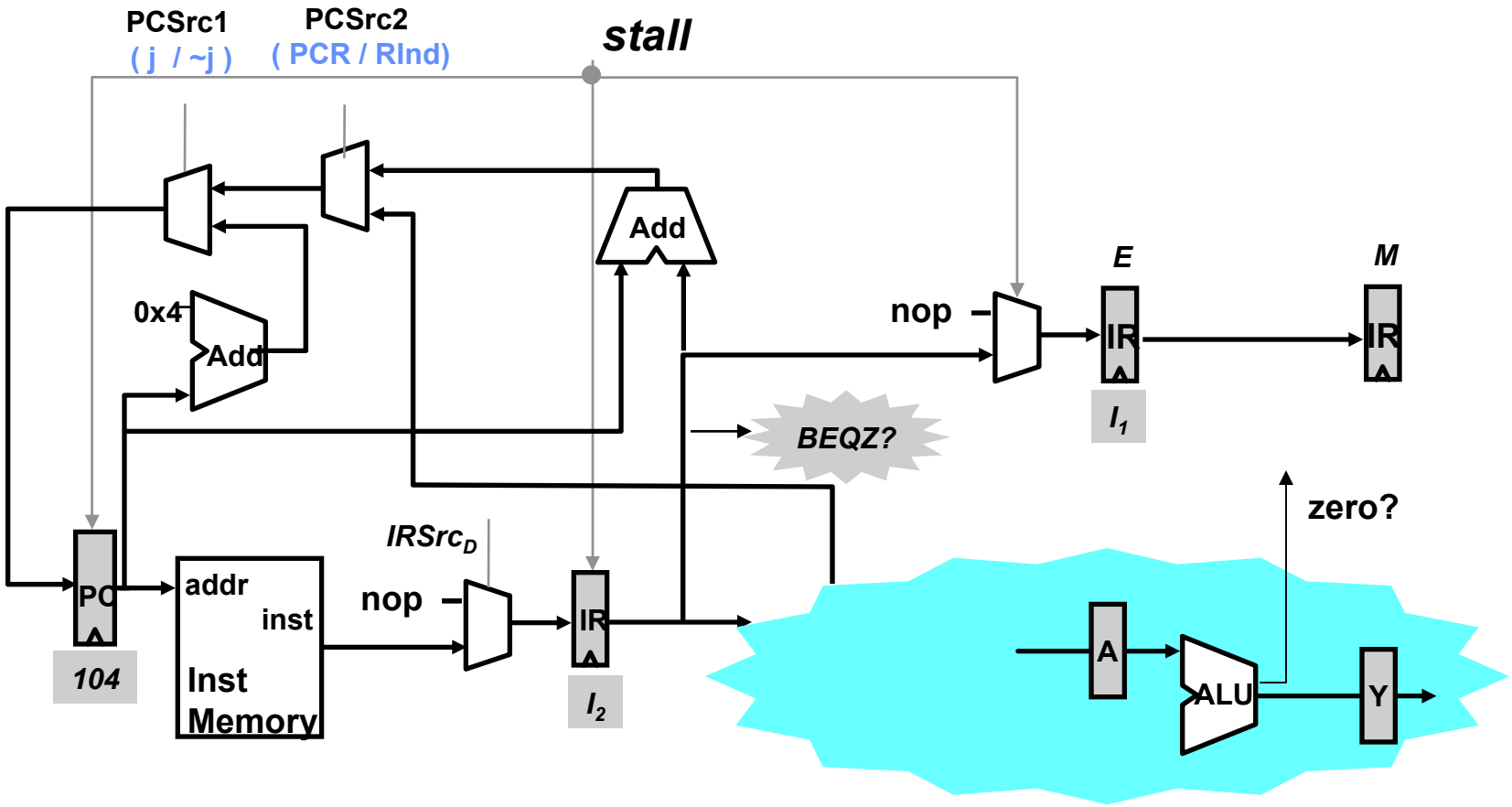
		time							
		t0	t1	t2	t3	t4	t5	t6	t7 . . . .
I_1 096	ADD	IF_1	ID_1	EX_1	MA_1	WB_1			
I_2 100	J +200		IF_2	ID_2	EX_2	MA_2	WB_2		
I_3 104	~~ADD~~ *kill*			IF_3	nop	nop	nop	nop	
I_4 304	SUB				IF_4	ID_4	EX_4	MA_4	WB_4

Resource Usage

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

nop ⇒ *pipeline bubble*

Pipelining Conditional Branches

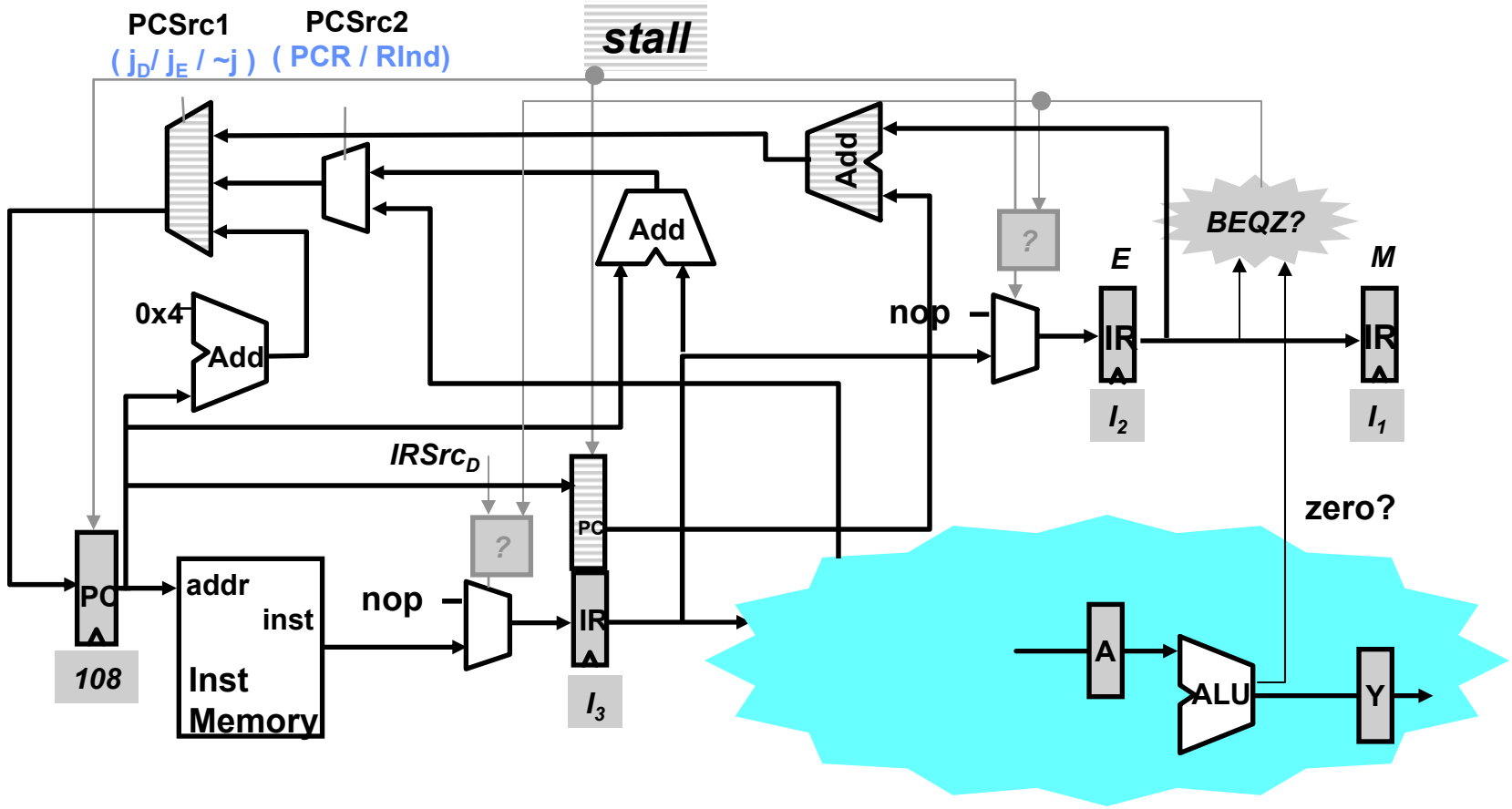


I_1 096 ADD
I_2 100 BEQZ r1, +200
I_3 104 ADD
I_4 304 ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Conditional Branches: *solution 1*



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*

New Stall Signal

$$stall = (((rf1_D = ws_E).we_E + (rf1_D = ws_M).we_M + (rf1_D = ws_W).we_W).re1_D$$
$$+ ((rf2_D = ws_E).we_E + (rf2_D = ws_M).we_M + (rf2_D = ws_W).we_W).re2_D)$$
$$. !((opcode_E = BEQZ).z + (opcode_E = BNEZ).!z)$$

Control Equations for PC Muxes

Solution 1

PCSrc1 = *Case* $opcode_E$
- BEQZ.z, BNEZ.!z ⇒ j_E
- ... ⇒ *Case* $opcode_D$
  - J, JAL, JR, JALR ⇒ j_D
  - ... ⇒ ~j

PCSrc2 = *Case* $opcode_D$
- J, JAL ⇒ PCR
- JR, JALR ⇒ RegI

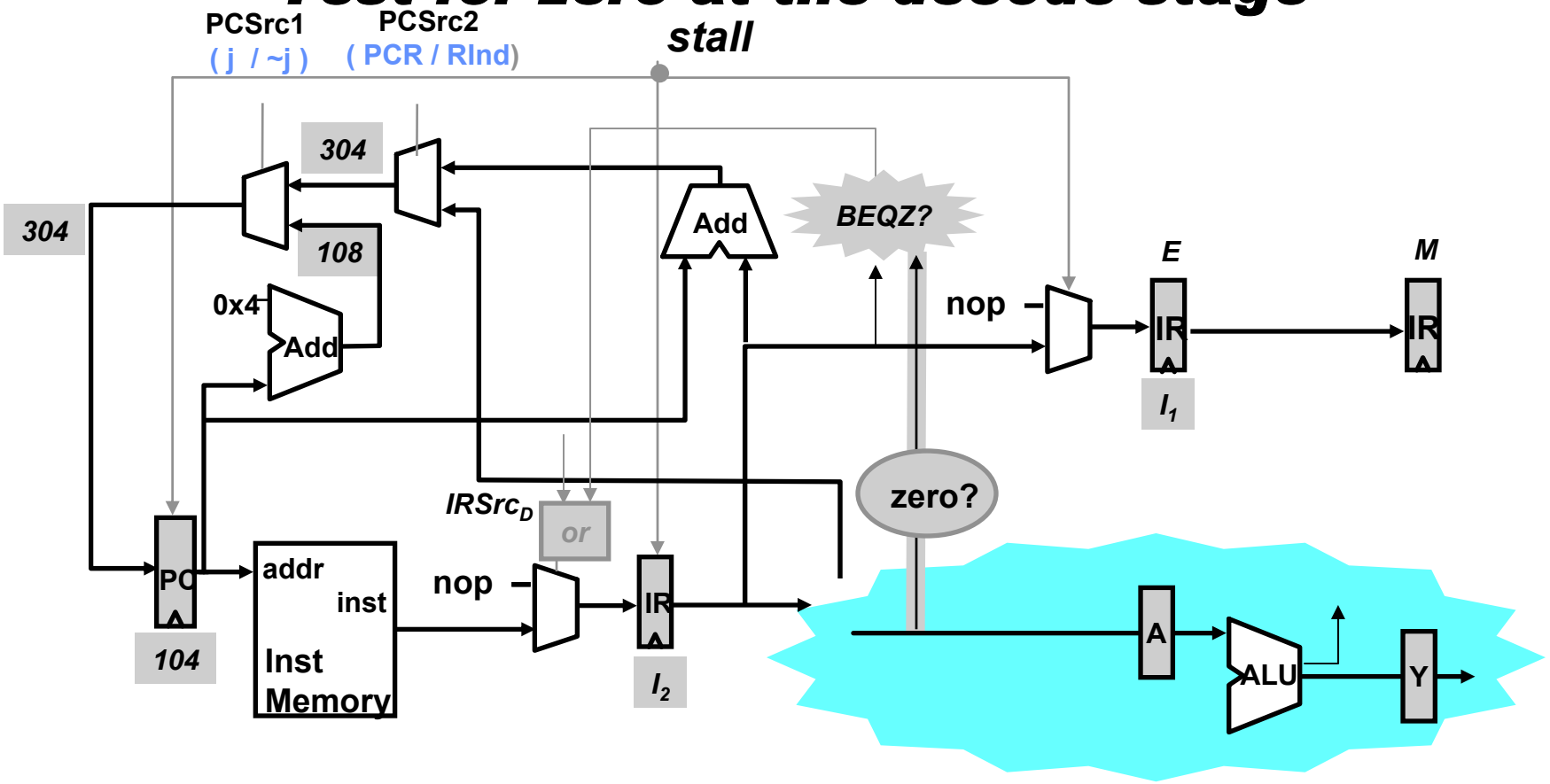
$IRSrc_D$ = *Case* $opcode_E$
- BEQZ.z, BNEZ.!z ⇒ nop
- ... ⇒ (*Case* $opcode_D$
  - J, JAL, JR, JALR ⇒ nop
  - ... ⇒ IM)

$IRSrc_E$ = *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

Conditional Branches: *solution 2*

Test for zero at the decode stage



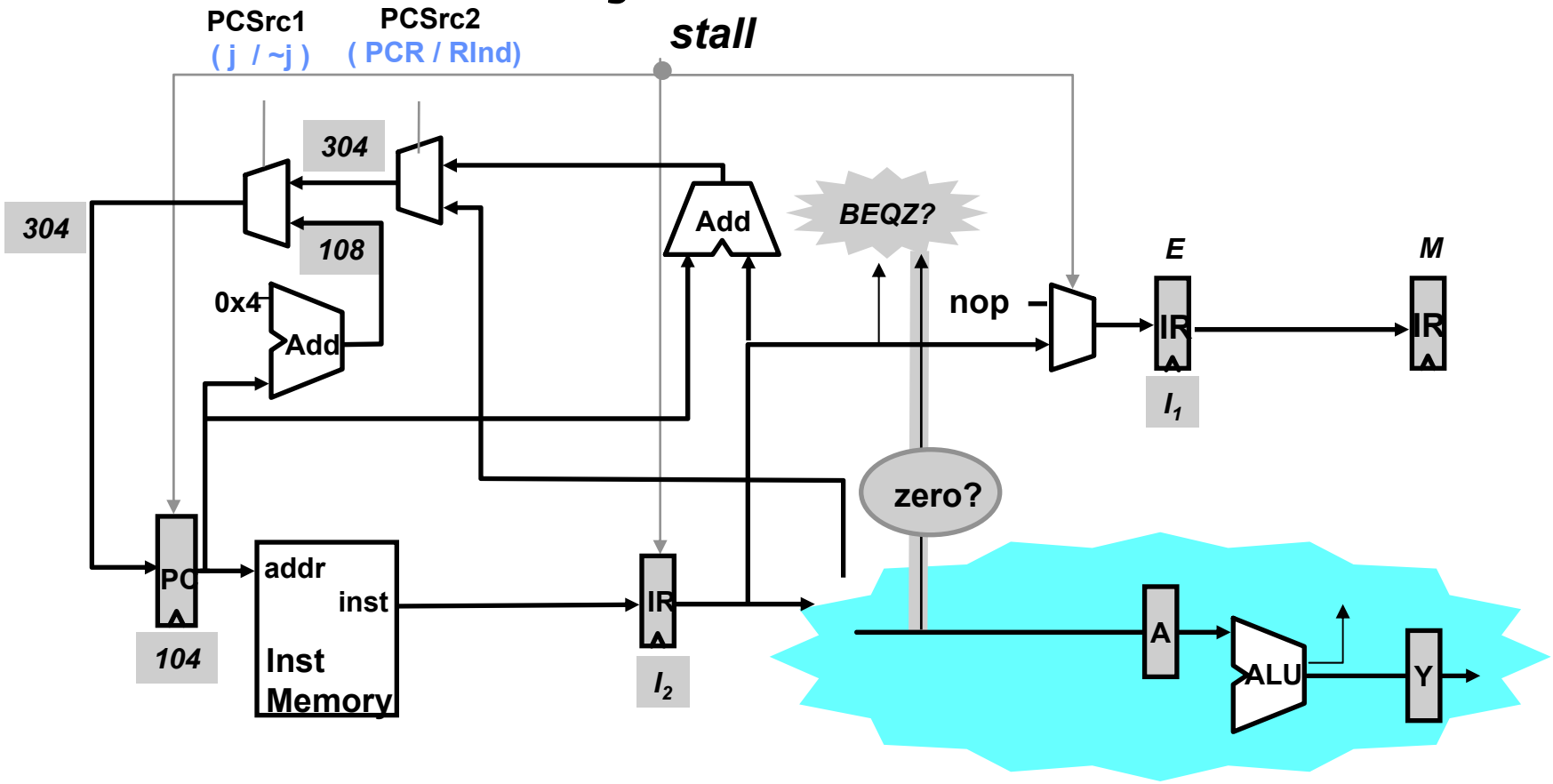
I_1 096 ADD
I_2 100 BEQZ r1, +200
I_3 104 ADD
I_4 304 ADD

Need to kill only one instruction !

Wouldn't work if DLX had general branch conditions (i.e., r1>r2)?

Conditional Branches: *solution 3*

Delayed Branches



I_1 096 ADD
I_2 100 BEQZ r1, +200
I_3 104 ADD (delay slot)
I_4 304 ADD

Change the semantics of branches and jumps

⇒ Instruction after branch *always* executed, regardless if branch taken or not taken.

Need not kill any instructions !

Annulling Branches

- Plain delayed branches only allow compiler to fill delay slot with instructions that will *always* be executed:

```
        ADD R1, R2, R3
        BEQZ R3, target
        NOP (delay slot)
```

```
        BEQZ R3, target
        ADD R1, R2, R3 (delay slot)
```

- Annulling branches kill delay slot if branch is *not taken*. Compiler can therefore fill delay slot with copy of branch target:

```
        ADD R1, R2, R3
        BEQZ R1, target
        NOP (delay slot)
        ...
target:
        ADDI R2, R2, #1
        SUB R5, R6, R8
```

```
        ADD R1, R2, R3
        BEQZL R1, target+4 (annulling)
        ADDI R2, R2, #1 (delay slot)
        ...
target:
        ADDI R2, R2, #1
        SUB R5, R6, R8
```

- Can also have variant of annulling branch that kills delay slot if branch is *taken*

Branch Delay Slots

First introduced in pipelined microcode engines.
Adopted for early RISC machines with single-issue pipelines, made *visible* to user-level software.

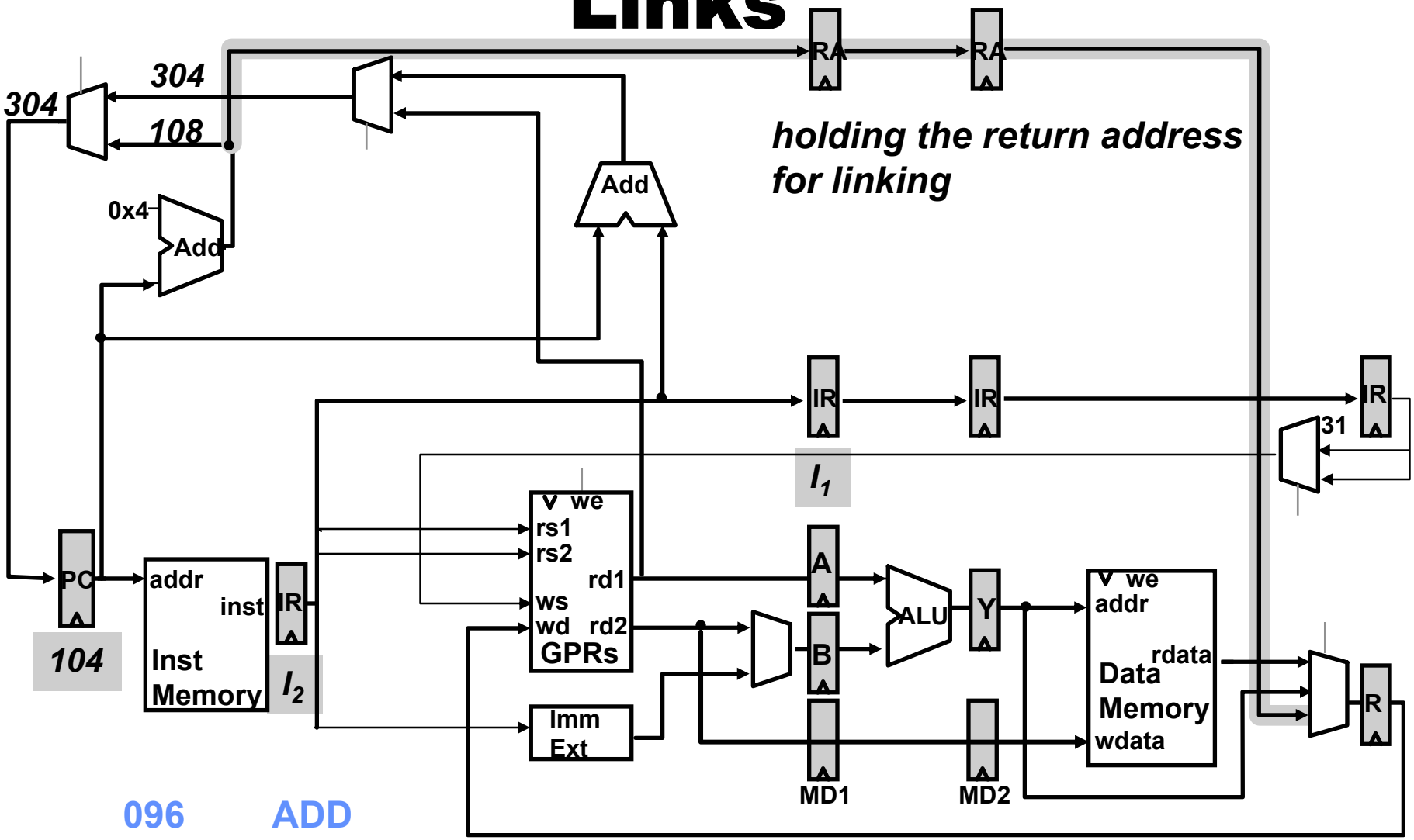
Advantages
- simplifies control logic for simple pipeline
- compiler helps reduce branch hazard penalties
  - ~70% of single delay slots usefully filled

Disadvantages
- complicates ISA specification and programming
  - adds extra "next-PC" state to programming model
- complicates control logic for more aggressive implementations
  - e.g., out-of-order superscalar designs

⇒ *Out of favor for new (post-1990) general-purpose ISAs*

Later lectures will cover advanced techniques for control hazards:
- dynamic (run-time) schemes, e.g., branch prediction
- static (compile-time) schemes, e.g., predicated execution

Pipelining Delayed Jumps & Links



I_1	096	ADD
I_2	100	JALR +200
I_3	104	ADD (delay slot)
I_4	304	ADD

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

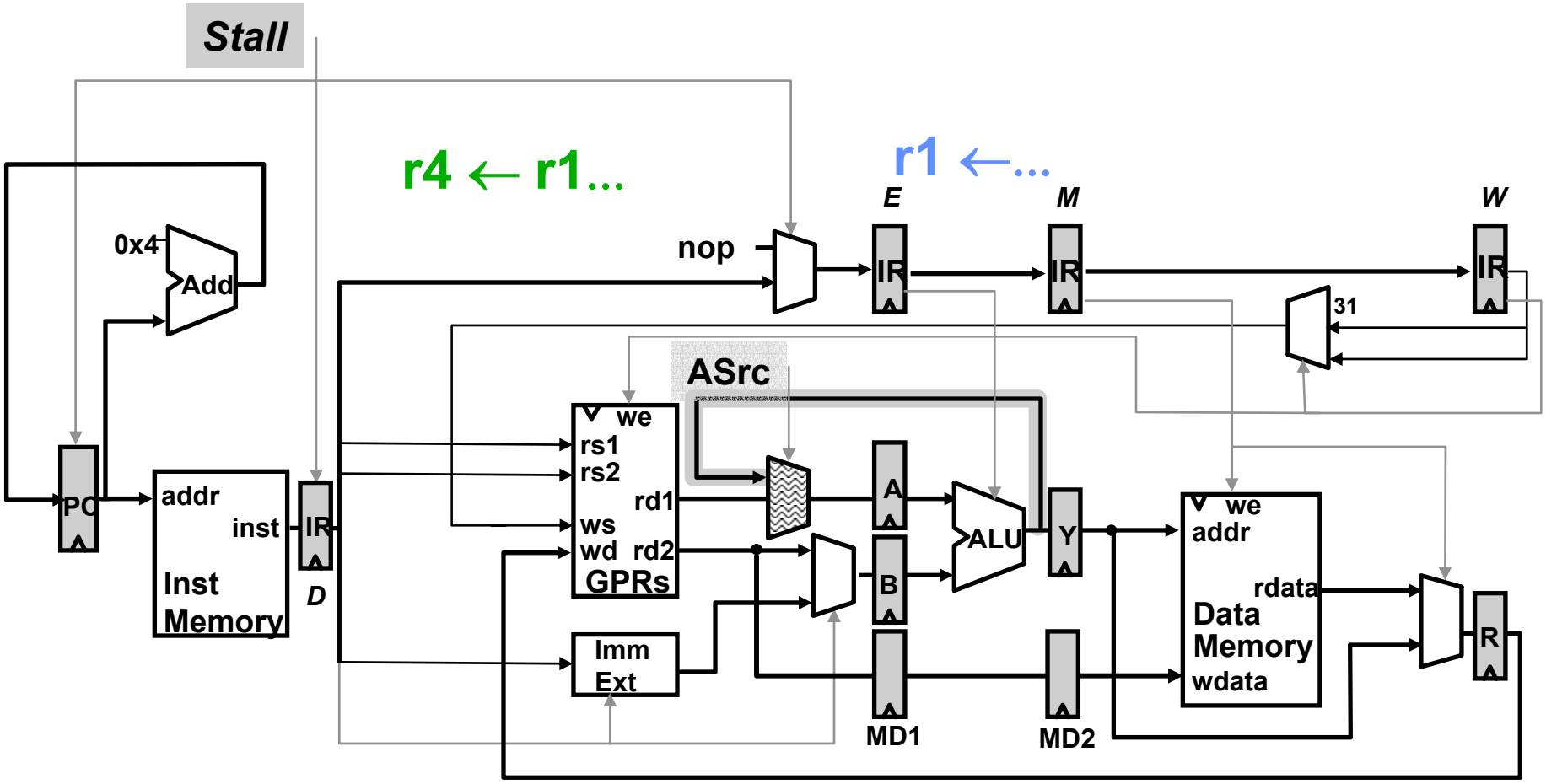
Each *stall or kill* introduces a bubble in the pipeline

$\Rightarrow$ *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 Bypasses



...

(I_1) r1 ← r0 + 10

(I_2) r4 ← r1 + 17

...

Of course you can add many more bypasses!

The Bypass Signal

deriving it from the stall signal

C_{dest}

ws = *Case* opcode
- ALU ⇒ rf3
- ALUi, LW ⇒ rf2
- JAL, JALR ⇒ R31

we = *Case* opcode
- ALU, ALUi, LW ⇒ (ws ≠ R0)
- 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

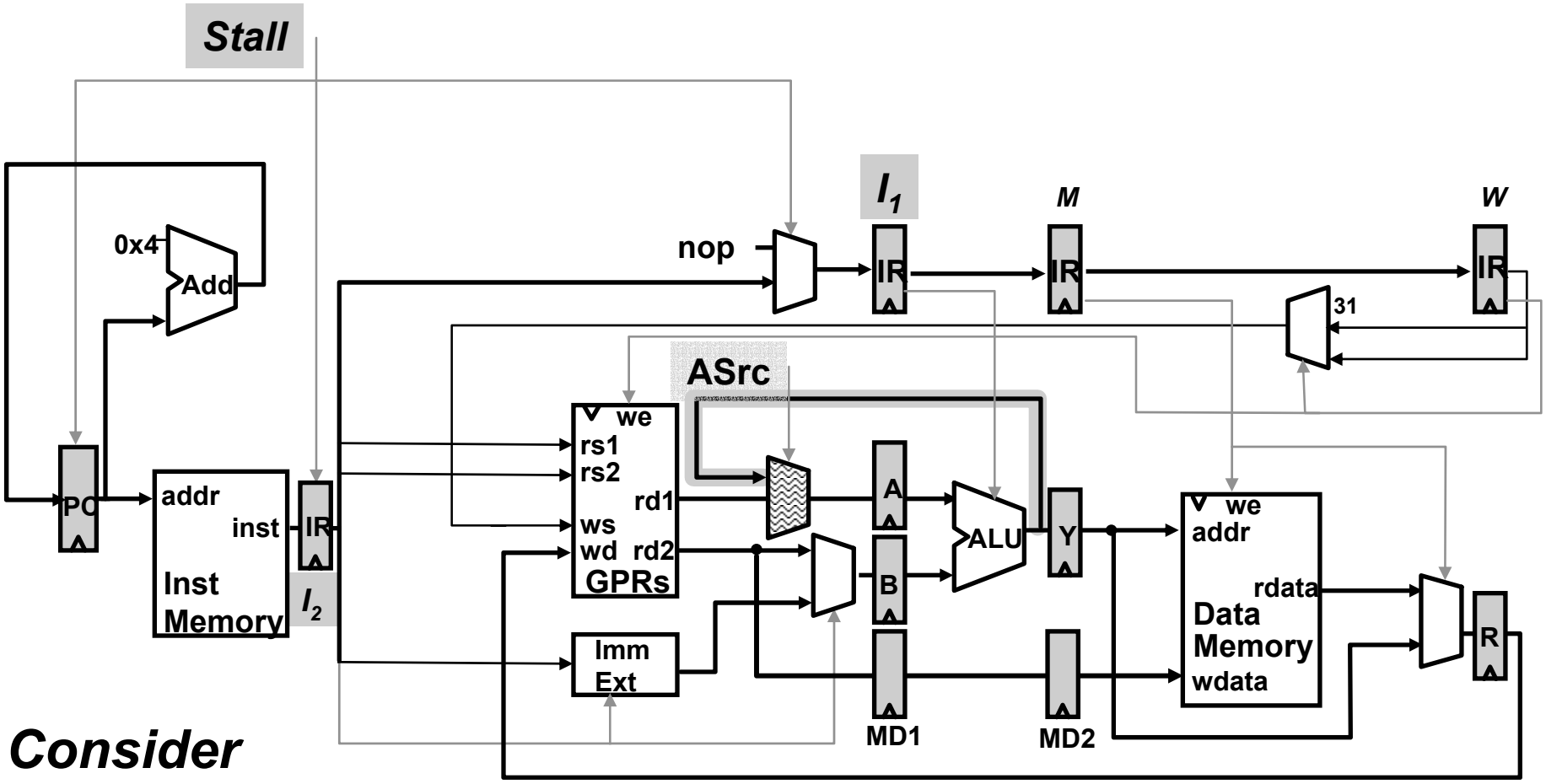
C_{stall}

$$stall = ((rf1_D = ws_E).we_E + (rf1_D = ws_M).we_M + (rf1_D = ws_W).we_W) . re1_D + ((rf2_D = ws_E).we_E + (rf2_D = ws_M).we_M + (rf2_D = ws_W).we_W) . re2_D$$

C_{bypass} $ASrc = (rf1_D = ws_E).we_E . re1_D$

Is this correct ?

Usefulness of a Bypass

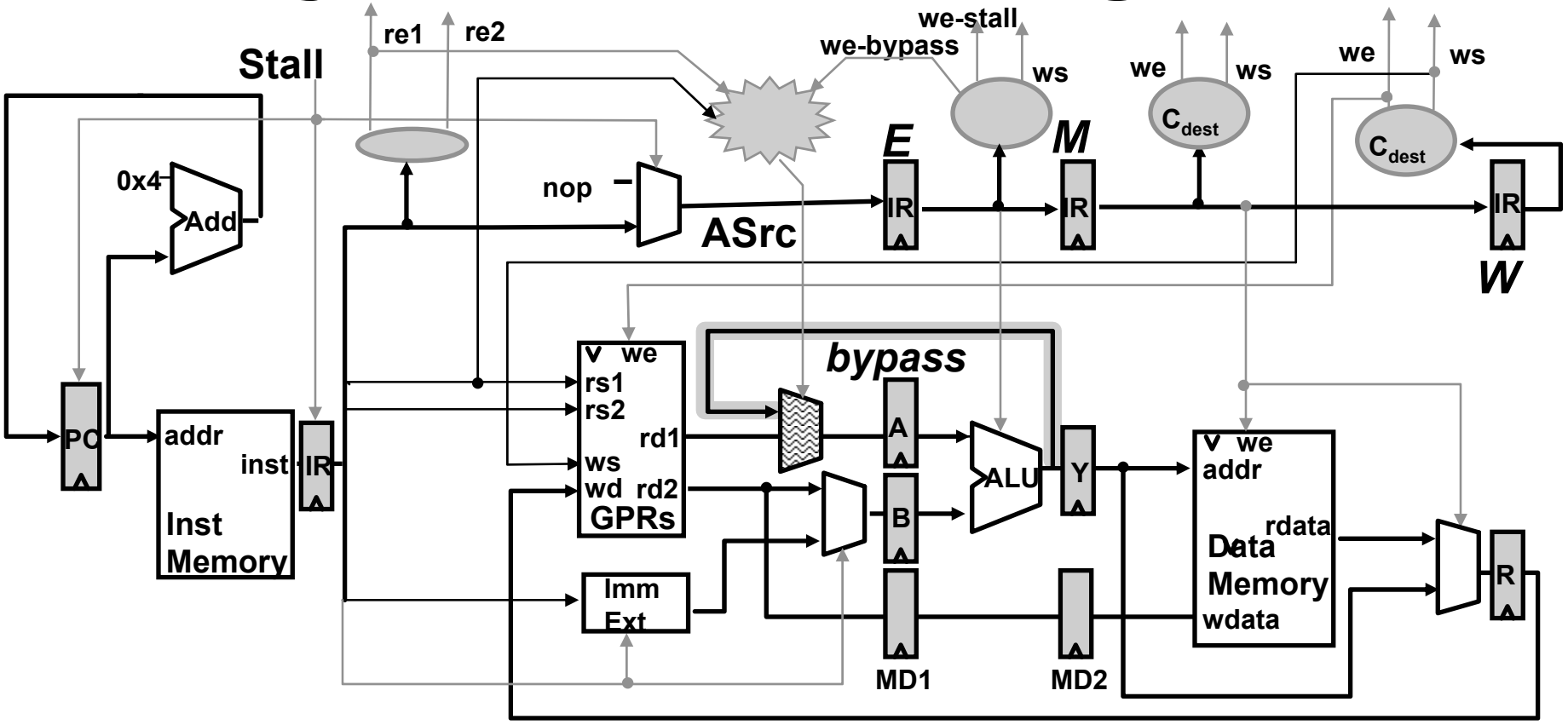


Consider

	...	...	...
(I_1)	r1 ← r0 + 10	r1 ← M[r0 + 10]	JAL 500
(I_2)	r4 ← r1 + 17	r4 ← r1 + 17	r4 ← r31 + 17

Where can this bypass help?

Bypass and Stall Signals



we_E has to be split into two components

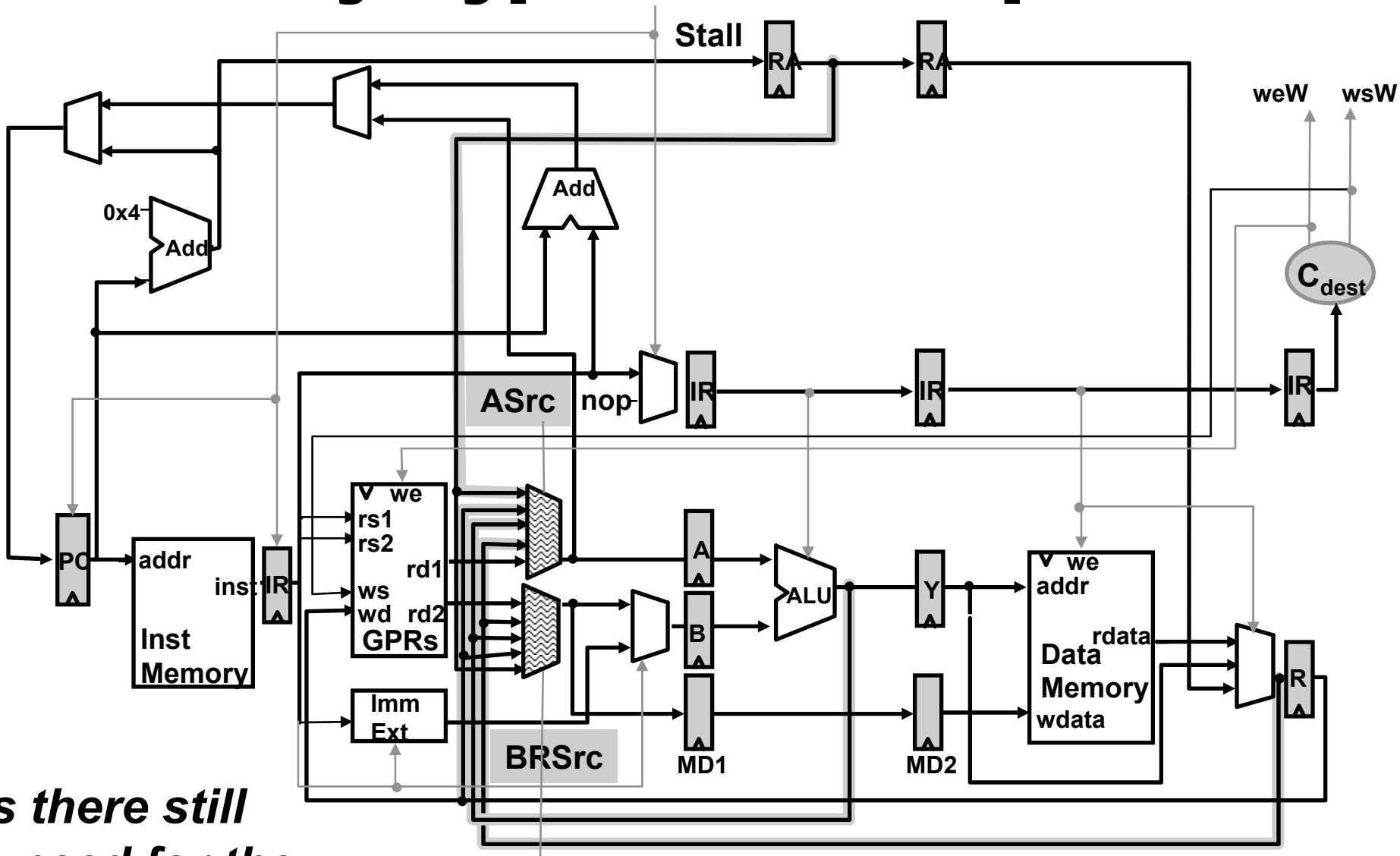
$we\text{-}bypass_E = ((opcode_E = ALU) + (opcode_E = ALUi_E)) \,.\, (ws_E \neq R0)$

$we\text{-}stall_E = (opcode_E = LW_E).(ws_E \neq R0) + (opcode_E = JAL_E) + (opcode_E = JALR_E)$

$ASrc = (rf1_D = ws_E).we\text{-}bypass_E \,.\, re1_D$

$Stall = ((rf1_D = ws_E).we\text{-}stall_E) + (rf1_D = ws_M).we_M + (rf1_D = ws_W).we_W).\, re1_D + ((rf2_D = ws_E).we_E + (rf2_D = ws_M).we_M + (rf2_D = ws_W).we_W).\, re2_D$

Fully Bypassed Datapath



Is there still a need for the Stall signal ?

$$\text{Stall} = (rf1_D = ws_E).(opcode_E = LW_E).(ws_E \neq R0).re1_D + (rf2_D = ws_E).(opcode_E = LW_E).(ws_E \neq R0).re2_D$$

Why an Instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
  - typically all frequently used paths provided
  - some infrequently used bypass paths may increase cycle time and/or cost more than their benefit in 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 compiler code scheduling)