CS 152 Computer Architecture and Engineering

Lecture 4 - Pipelining

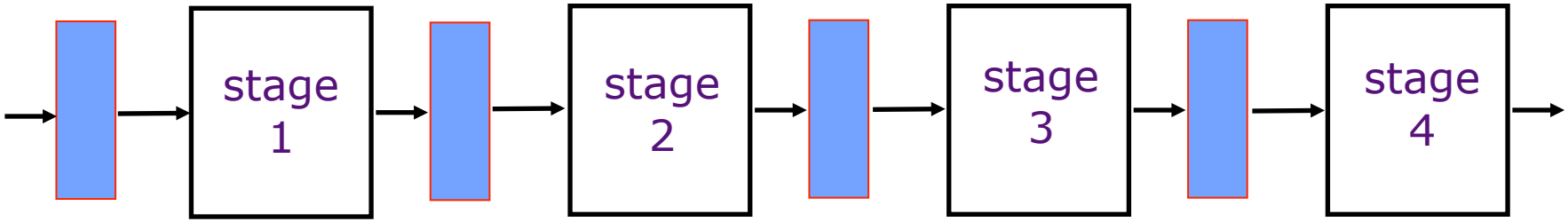
Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Last time in Lecture 3

- Microcoding became less attractive as gap between RAM and ROM speeds reduced
- Complex instruction sets difficult to pipeline, so difficult to increase performance as gate count grew
- Iron-law explains architecture design space
  - Trade instructions/program, cycles/instruction, and time/cycle
- Load-Store RISC ISAs designed for efficient pipelined implementations
  - Very similar to vertical microcode
  - Inspired by earlier Cray machines (more on these later)

An Ideal Pipeline



- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

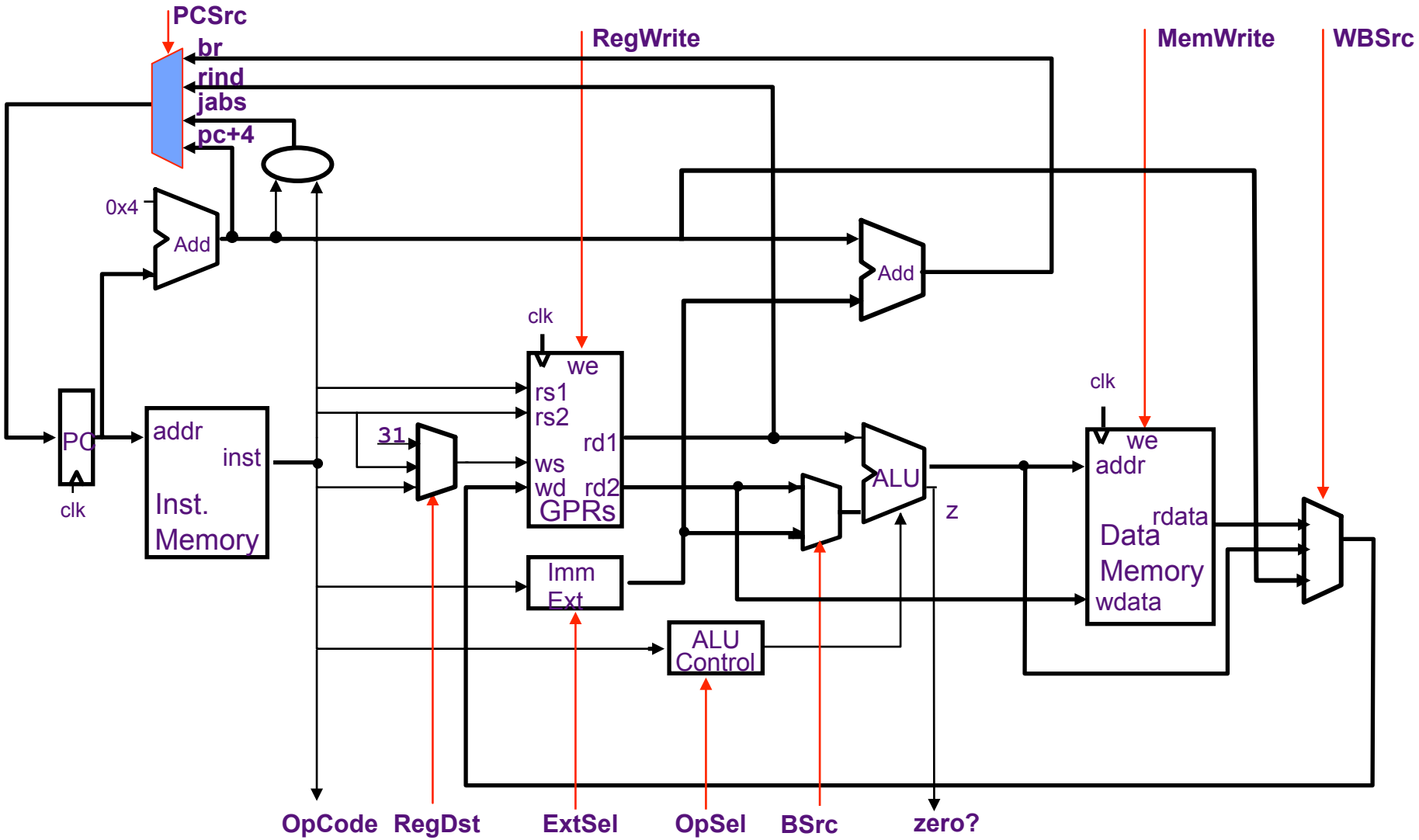
These conditions generally hold for industrial assembly lines, but instructions depend on each other!

Pipelined MIPS

To pipeline MIPS:

- First build MIPS without pipelining with CPI=1
- Next, add pipeline registers to reduce cycle time while maintaining CPI=1

Lecture 3: Unpipelined Datapath for MIPS



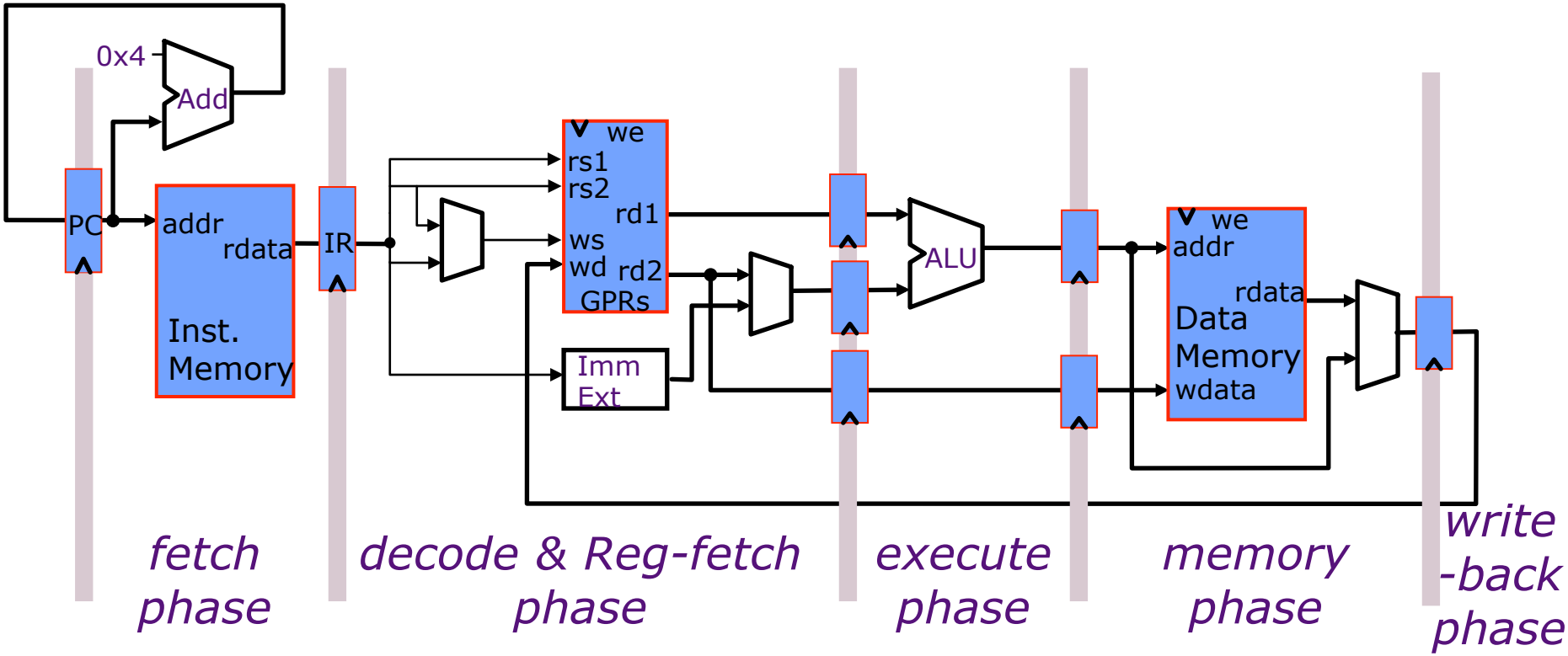
Lecture 3: Hardwired Control Table

Opcode	ExtSel	BSrc	OpSel	MemW	RegW	WBSrc	RegDst	PCSrc
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	$sExt_{16}$	Imm	Op	no	yes	ALU	rt	pc+4
ALUiu	$uExt_{16}$	Imm	Op	no	yes	ALU	rt	pc+4
LW	$sExt_{16}$	Imm	+	no	yes	Mem	rt	pc+4
SW	$sExt_{16}$	Imm	+	yes	no	*	*	pc+4
$BEQZ_{z=0}$	$sExt_{16}$	*	0?	no	no	*	*	br
$BEQZ_{z=1}$	$sExt_{16}$	*	0?	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	R31	jabs
JR	*	*	*	no	no	*	*	rind
JALR	*	*	*	no	yes	PC	R31	rind

BSrc = Reg / Imm
RegDst = rt / rd / R31

WBSrc = ALU / Mem / PC
PCSrc = pc+4 / br / rind / jabs

Pipelined Datapath



Clock period can be reduced by dividing the execution of an instruction into multiple cycles

$$t_C > \max \{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} \; (= t_{DM} \textit{ probably})$$

However, CPI will increase unless instructions are pipelined

"Iron Law" of Processor Performance

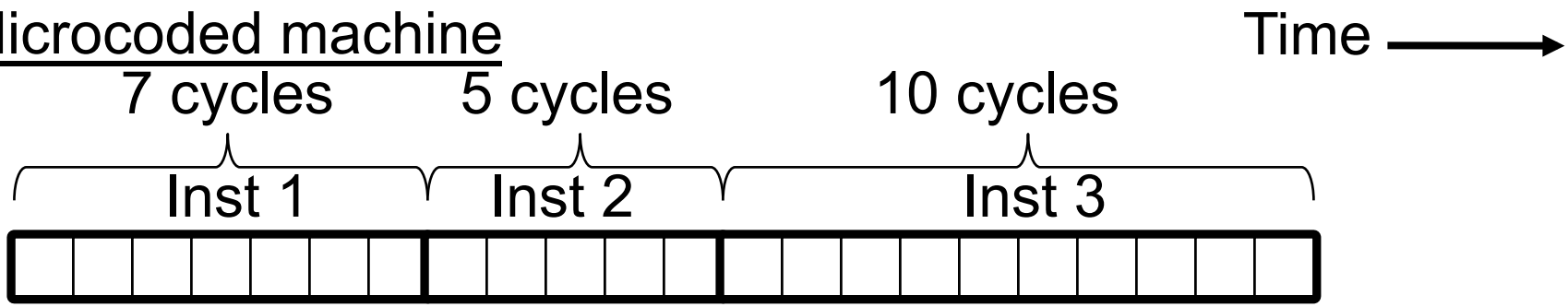
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

	Microarchitecture	CPI	cycle time
Lecture 2	Microcoded	>1	short
Lecture 3	Single-cycle unpipelined	1	long
Lecture 4	Pipelined	1	short

CPI Examples

Microcoded machine

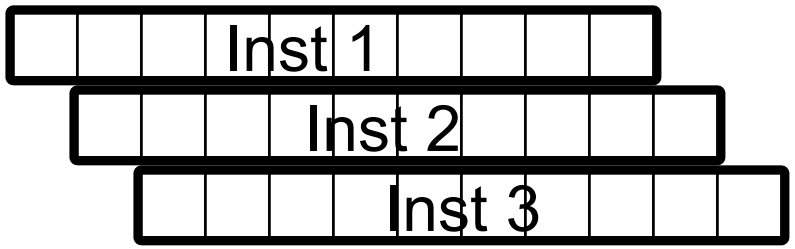


3 instructions, 22 cycles, CPI=7.33

Unpipelined machine

3 instructions, 3 cycles, CPI=1

Pipelined machine



3 instructions, 3 cycles, CPI=1

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!)

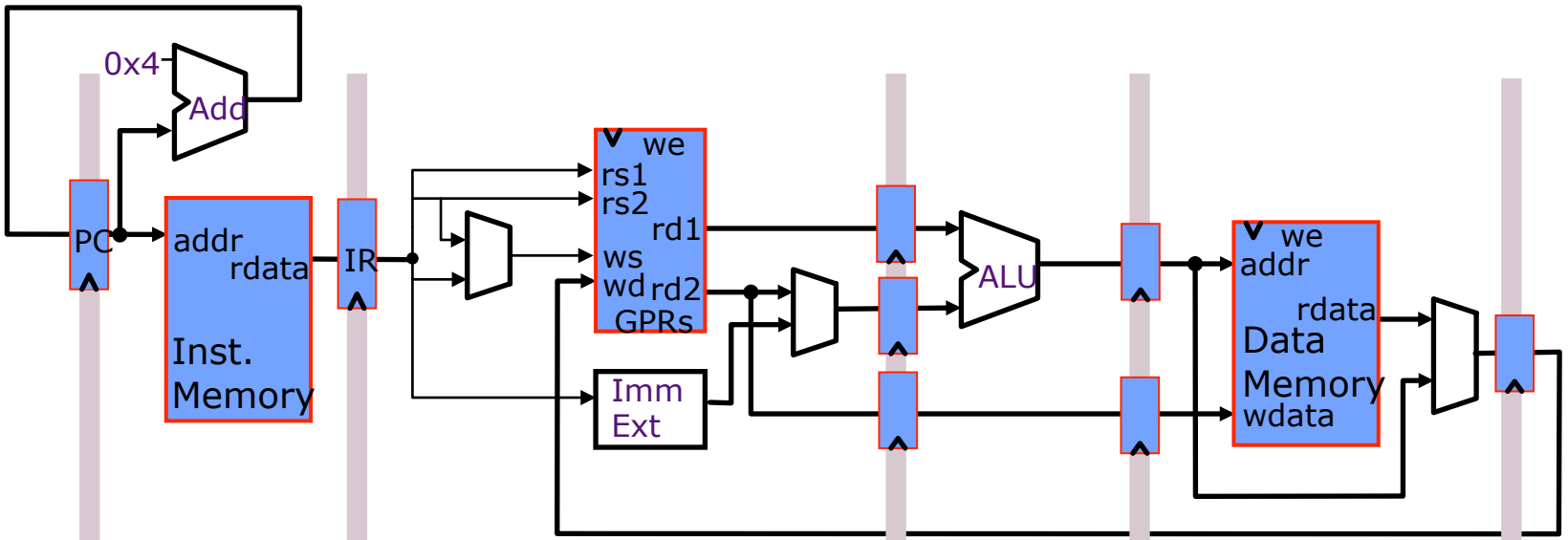
Thus, the following timing assumption is reasonable

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

A 5-stage pipeline will be the focus of our detailed design

- some commercial designs have over 30 pipeline stages to do an integer add!

5-Stage Pipelined Execution

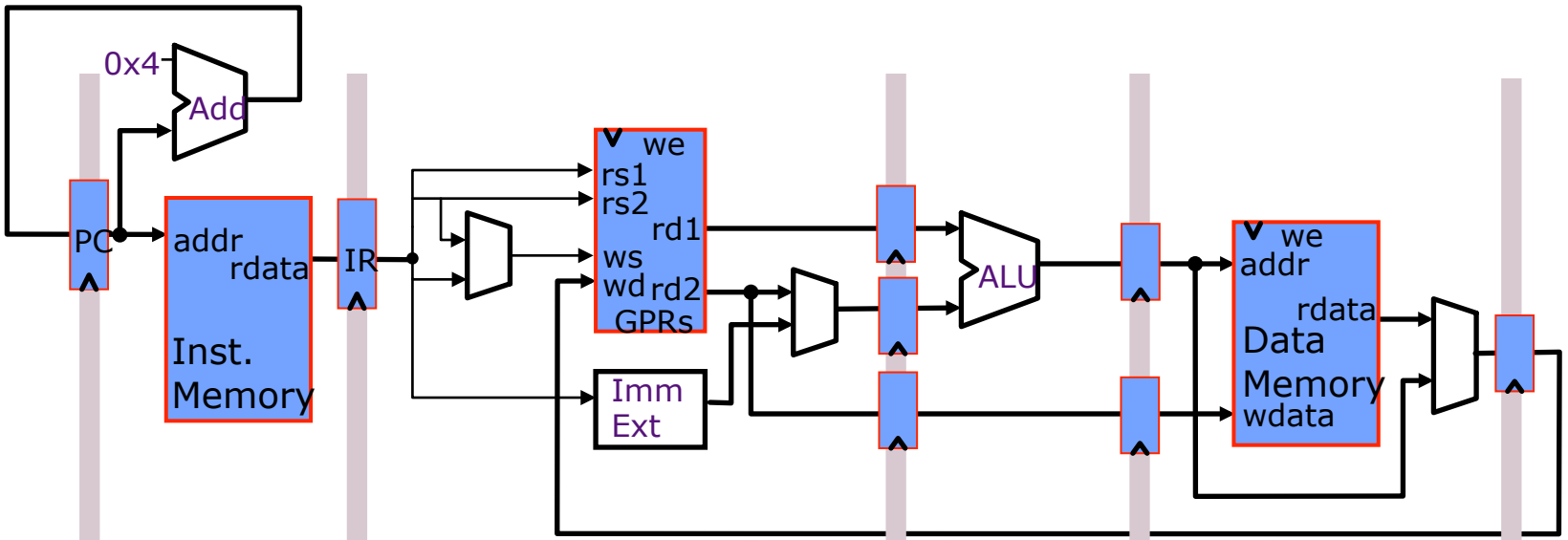


I-Fetch (IF) | *Decode, Reg. Fetch (ID)* | *Execute (EX)* | *Memory (MA)* | *Write-Back (WB)*

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

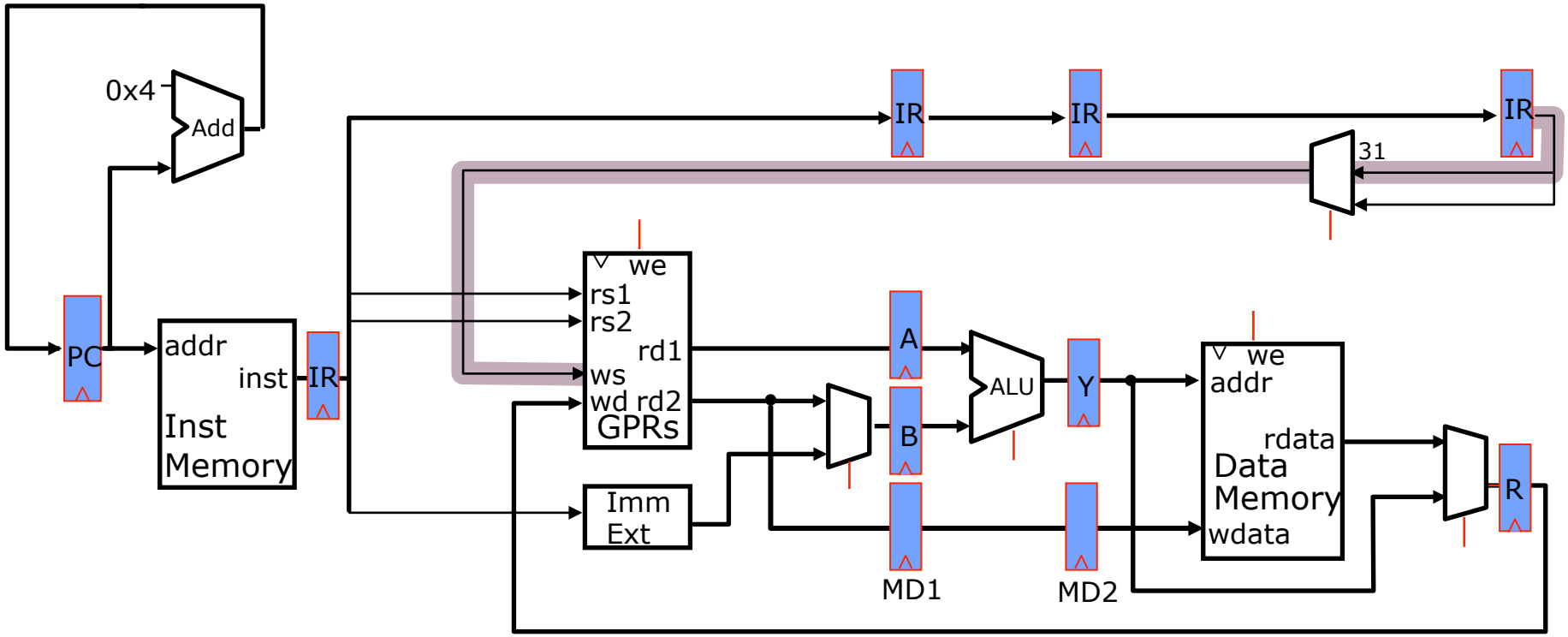


I-Fetch (IF) | *Decode, Reg. Fetch (ID)* | *Execute (EX)* | *Memory (MA)* | *Write -Back (WB)*

Resources

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

Pipelined Execution:
ALU Instructions

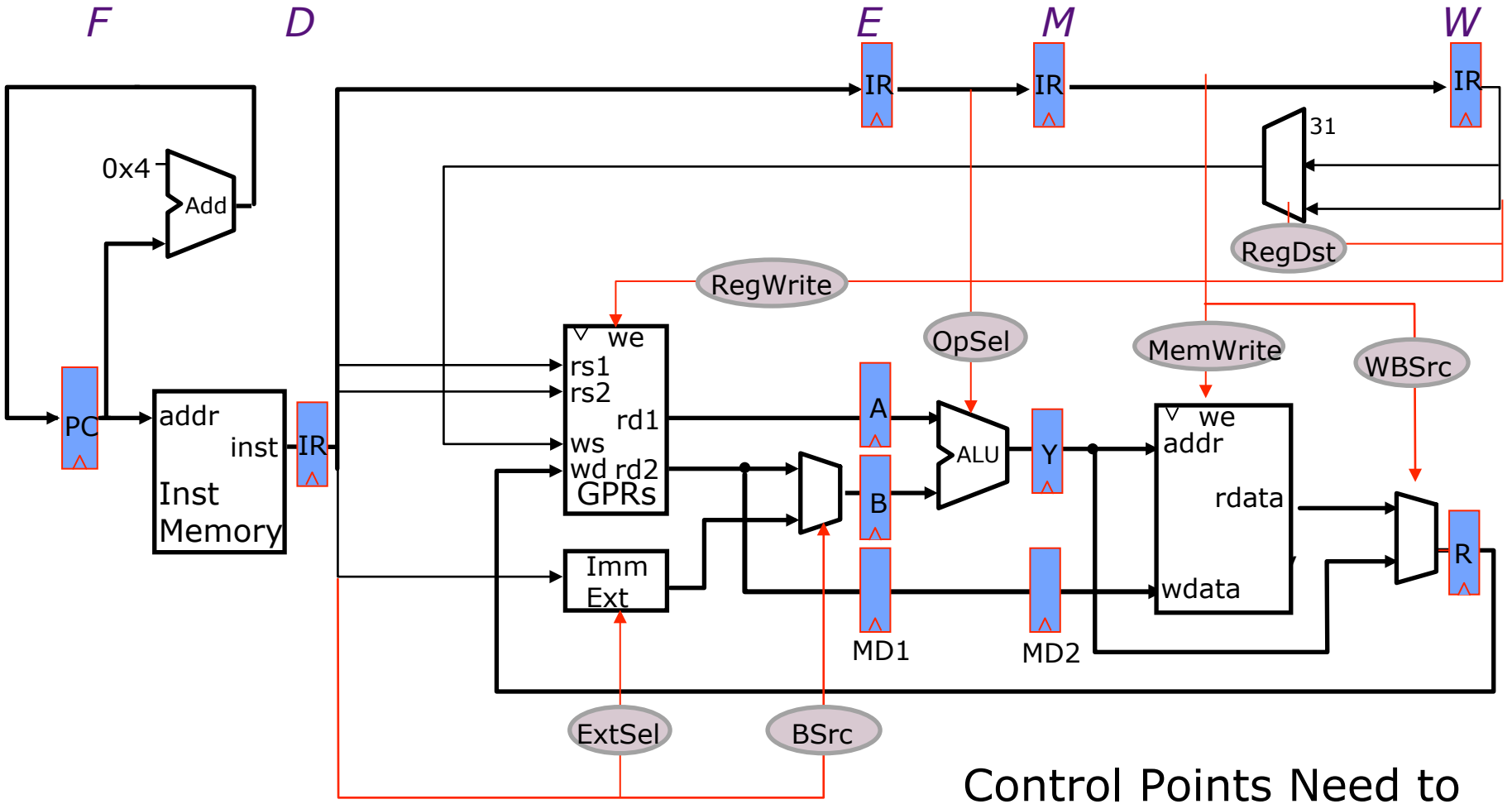


Not quite correct!

We need an Instruction Reg (IR) for each stage

Pipelined MIPS Datapath

without jumps



Control Points Need to Be Connected

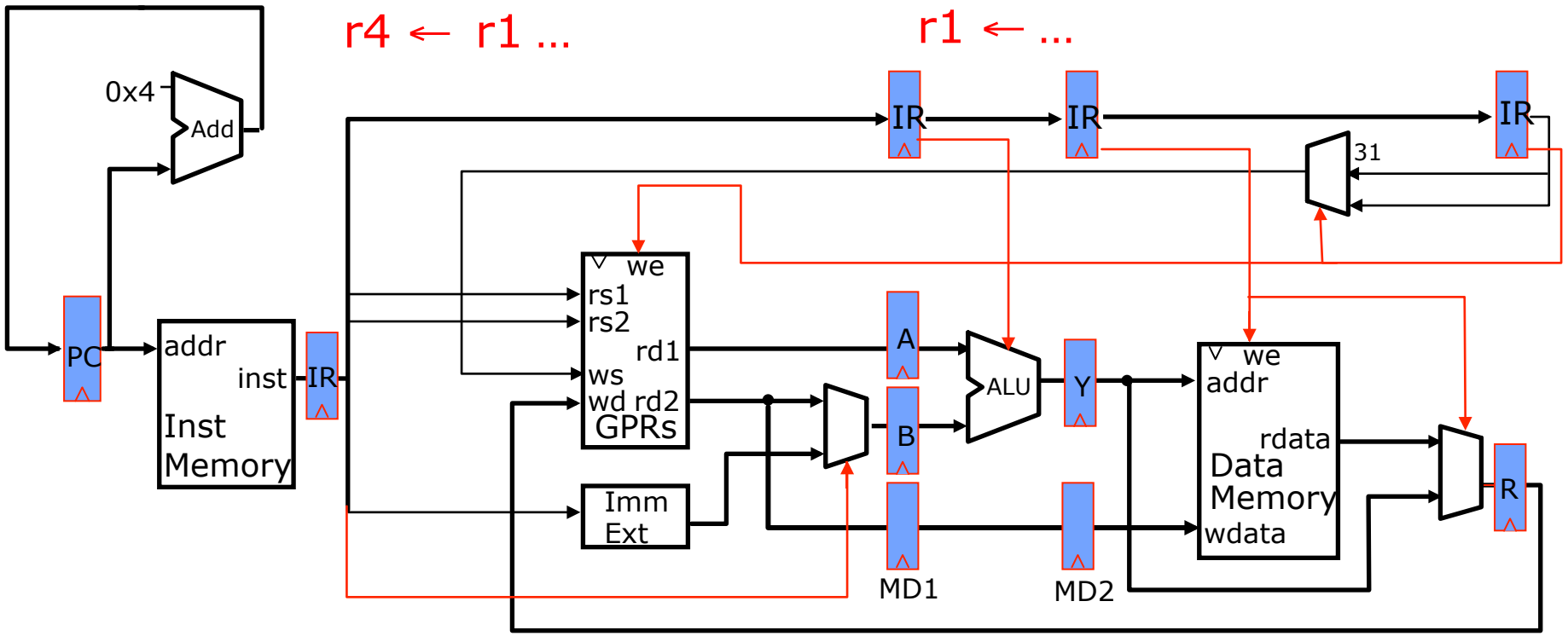
Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → data hazard
  - Dependence may be for the next instruction's address
    → control hazard (branches, exceptions)

Resolving Structural Hazards

- Structural hazards occurs when two instruction need same hardware resource at same time
  - Can resolve in hardware by stalling newer instruction till older instruction finished with resource
- A structural hazard can always be avoided by adding more hardware to design
  - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory
- Our 5-stage pipe has no structural hazards by design
  - Thanks to MIPS ISA, which was designed for pipelining

Data Hazards



...
r1 ← r0 + 10
r4 ← r1 + 17
...

r1 is stale. Oops!

CS152 Administrivia

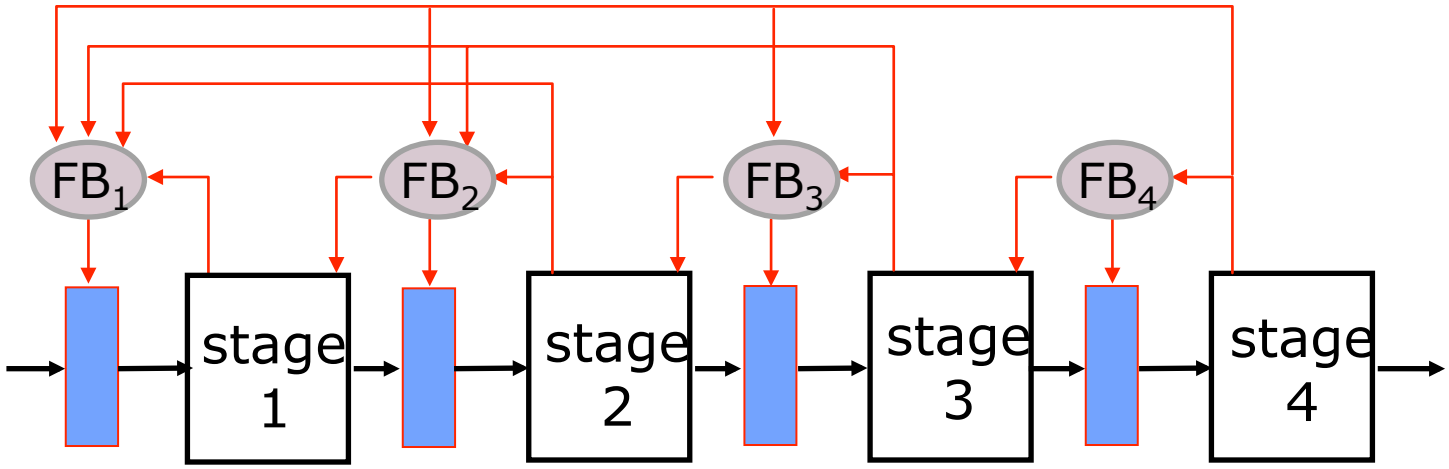
- PS 1 out Tuesday
- Lab 1 out today
- Scott Beamer will run section reviewing lab 1 at 2pm in 320 Soda

Resolving Data Hazards (1)

Strategy 1:

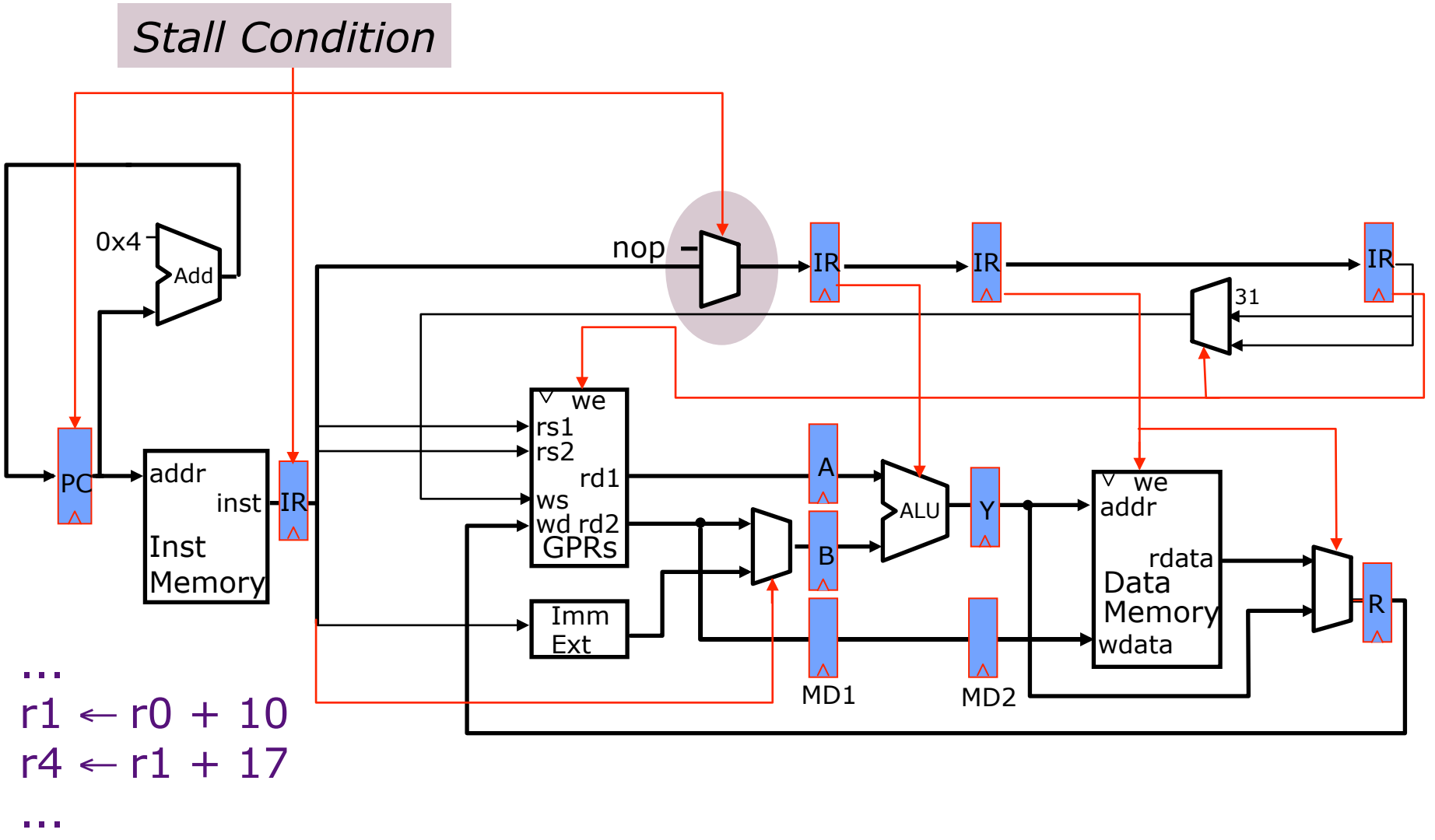
Wait for the result to be available by freezing earlier pipeline stages ➔ interlocks

Feedback to Resolve Hazards

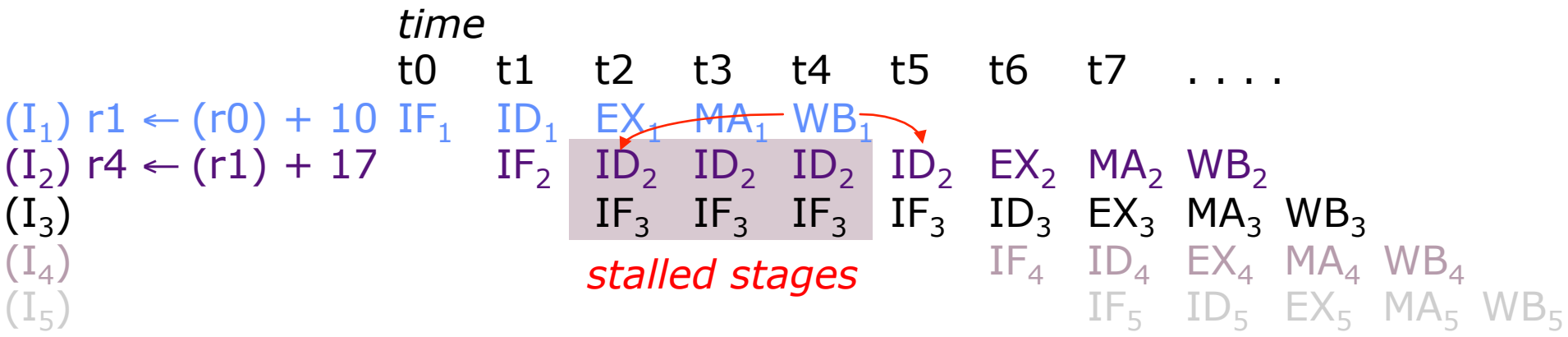


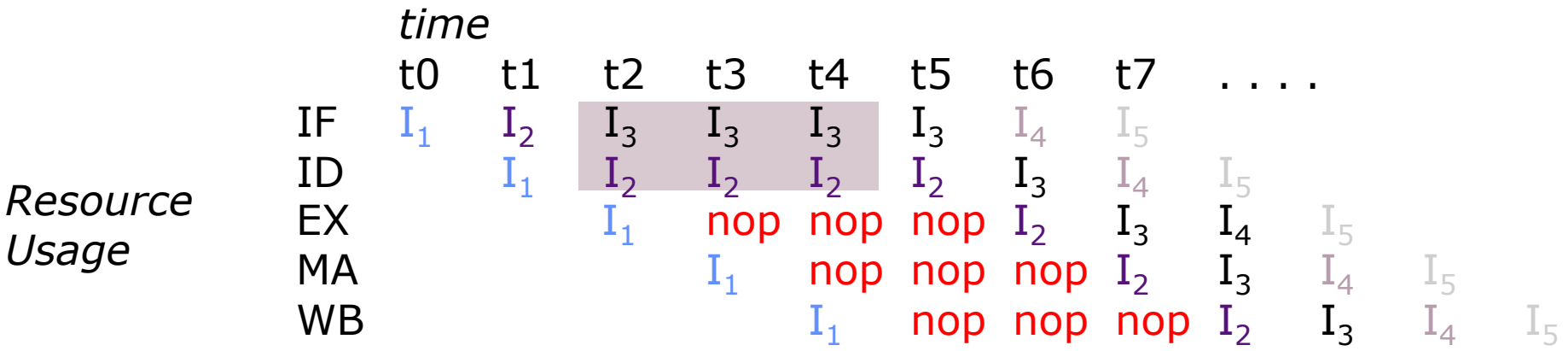
- Later stages provide dependence information to earlier stages which can *stall (or kill) instructions*
- Controlling a pipeline in this manner works provided *the instruction at stage i+1 can complete without any interference from instructions in stages 1 to i* (otherwise deadlocks may occur)

Interlocks to resolve Data Hazards



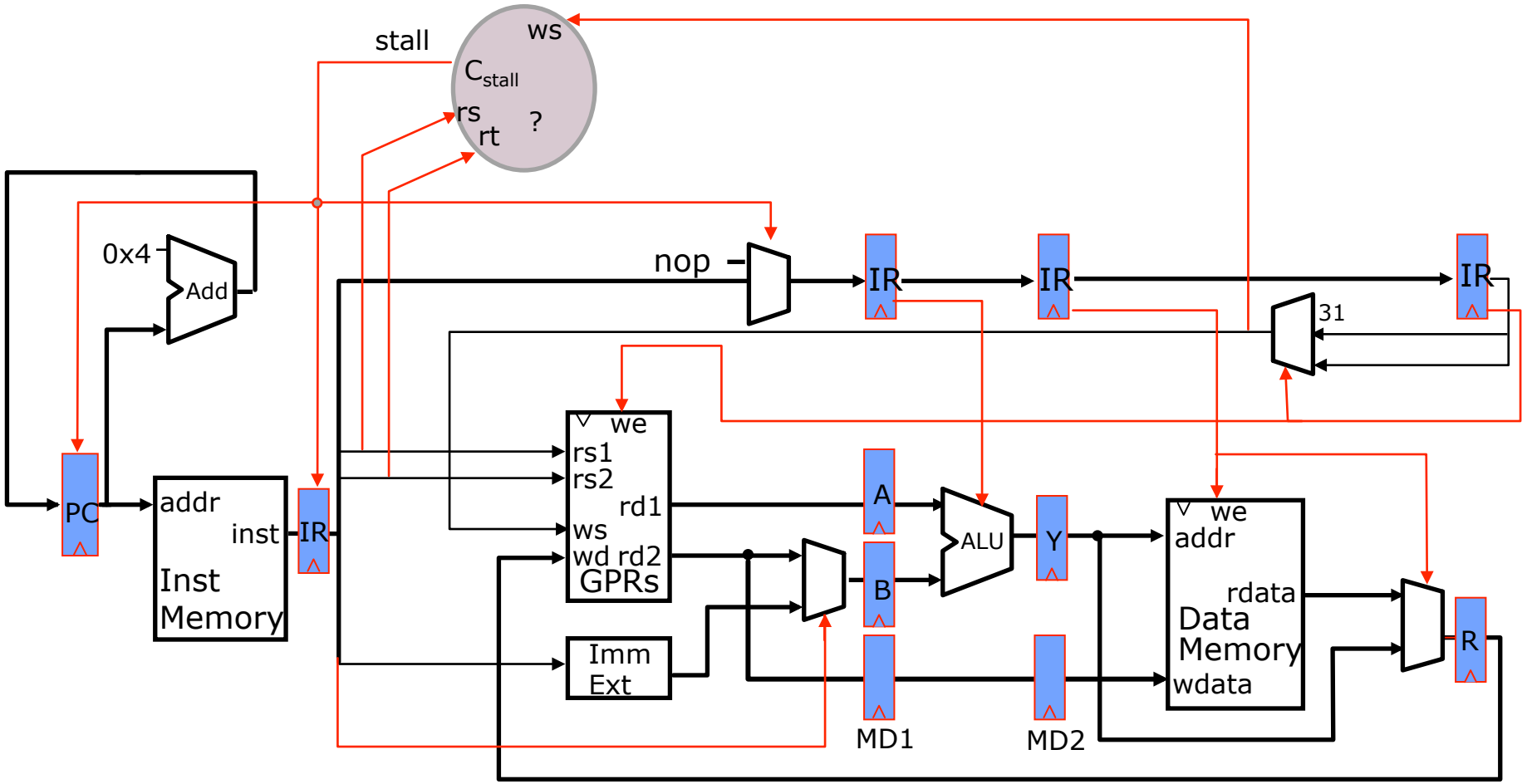
Stalled Stages and Pipeline Bubbles





nop ⇒ *pipeline bubble*

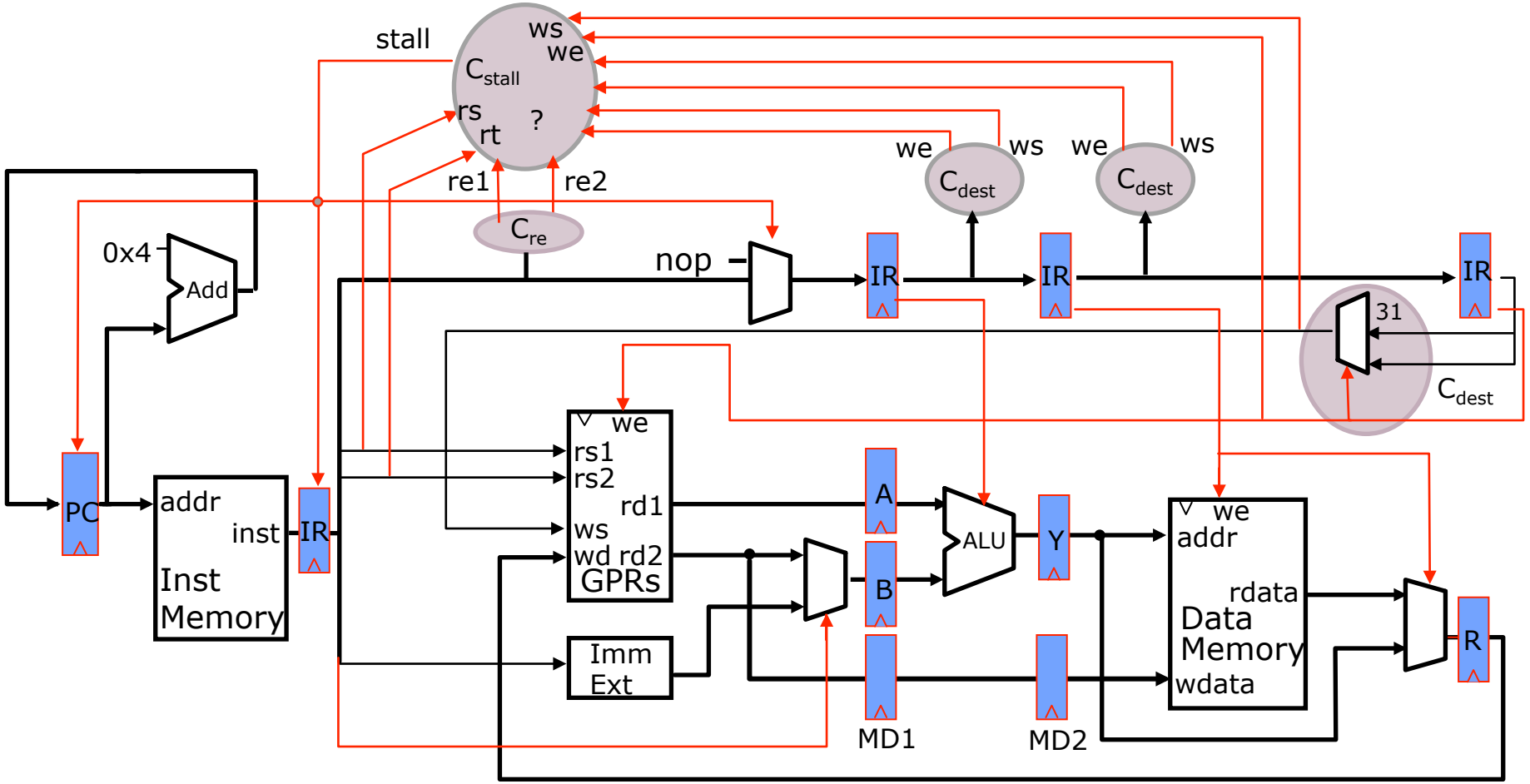
Interlock Control Logic



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

Interlock Control Logic

ignoring jumps & branches

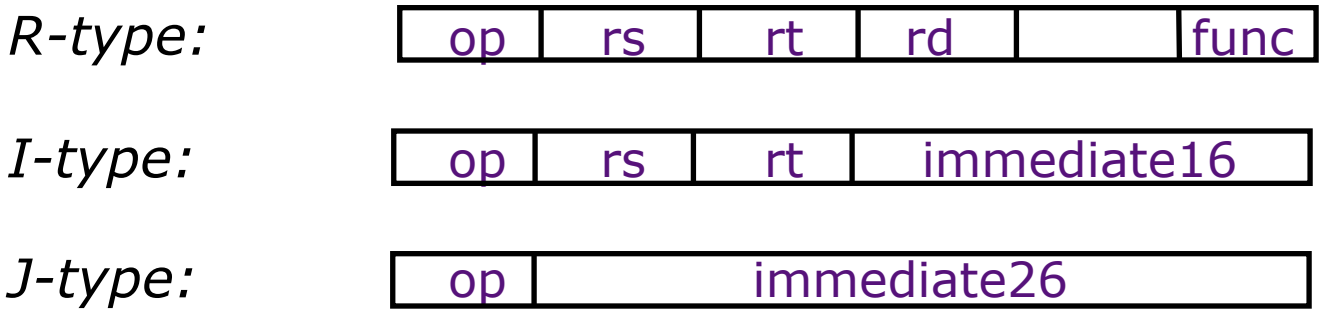


Should we always stall if the rs field matches some rd?

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

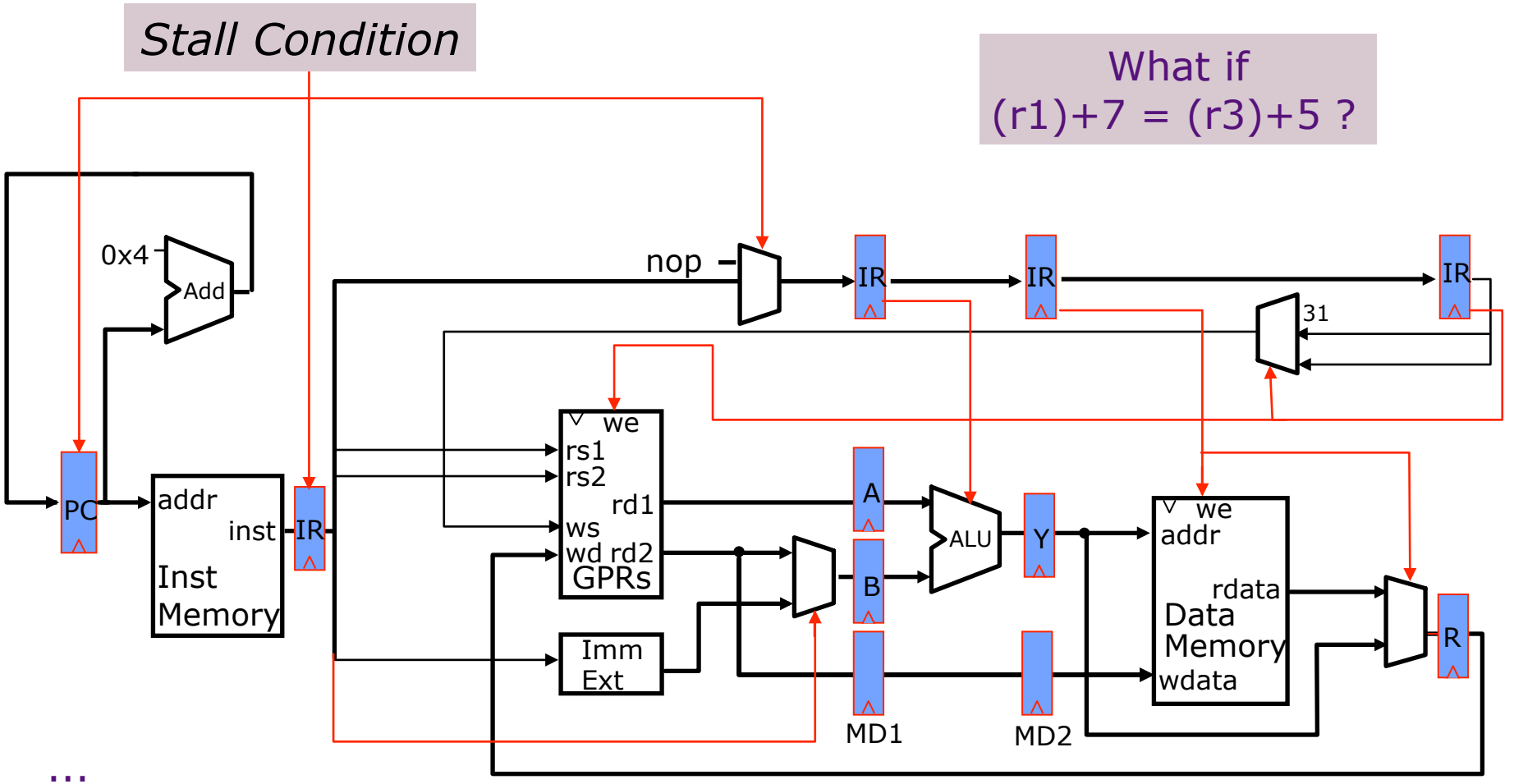
ALU, SW	⇒ on
...	⇒ off

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 are sometimes resolved in the pipeline and sometimes in the memory system itself.

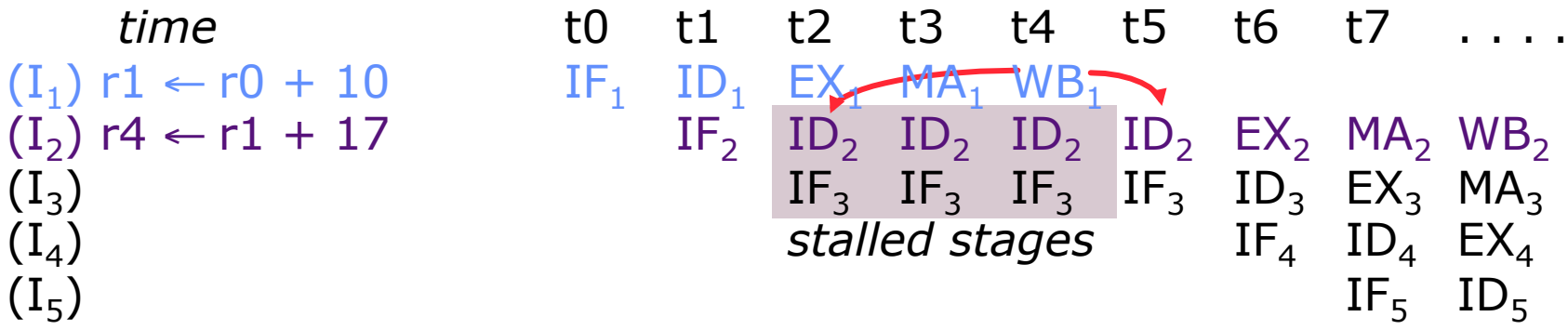
More on this later in the course.

Resolving Data Hazards (2)

Strategy 2:

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

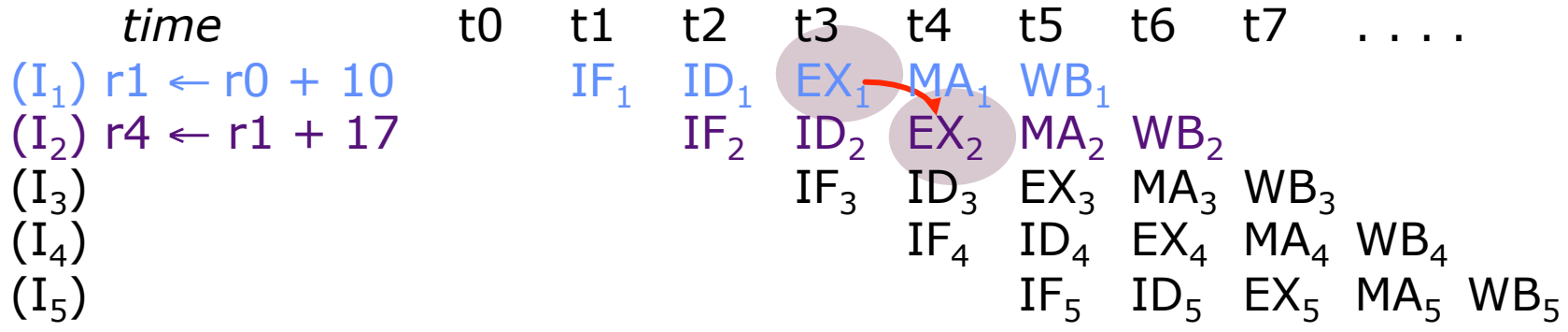
Bypassing



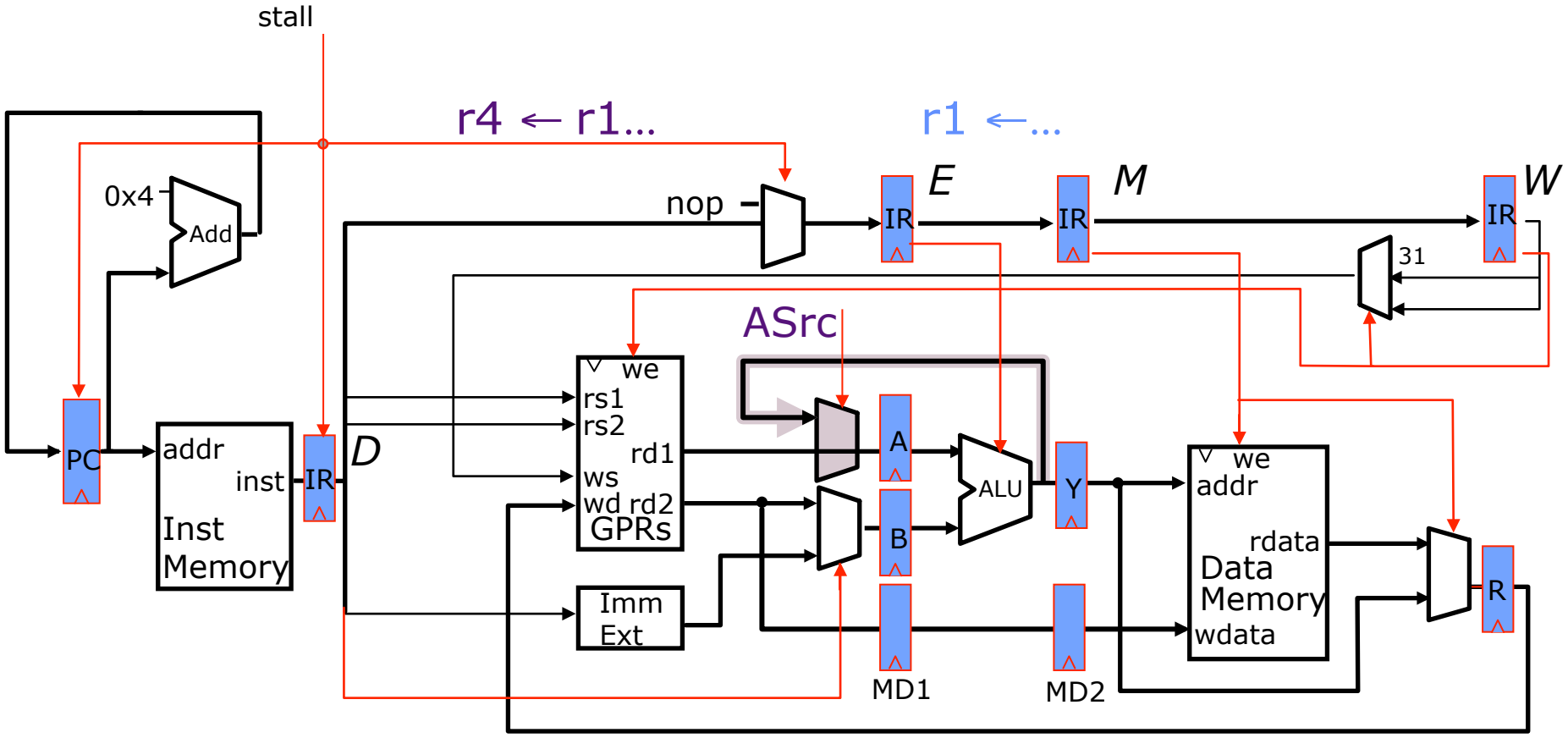
Each *stall or kill* introduces a bubble in the pipeline

$$\Rightarrow CPI > 1$$

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



Adding a Bypass



When does <u>this</u> 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	⇒ rd
ALUi, LW	⇒ rt
JAL, JALR	⇒ R31

we = *Case* opcode

ALU, ALUi, LW	⇒ (ws ≠ 0)
JAL, JALR	⇒ on
...	⇒ 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

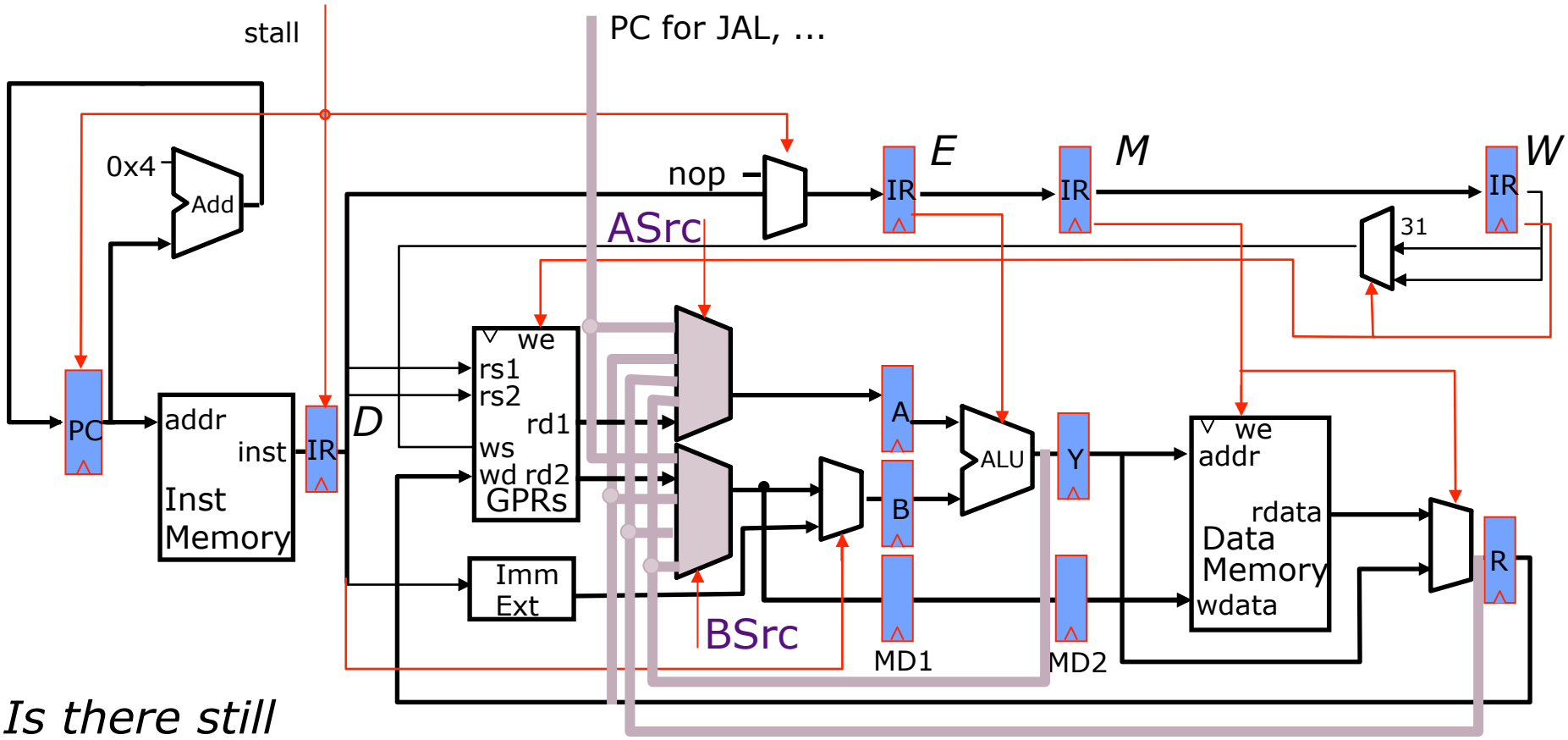
$we\text{-}bypass_E$ = *Case* $opcode_E$
- ALU, ALUi ⇒ $(ws \neq 0)$
- ... ⇒ off

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

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

Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

Guessed correctly ➔ do nothing

Guessed incorrectly ➔ kill and restart

…. We'll later see examples of this approach in more complex processors.

Next Time: Control Hazards

- Branches/Jumps
- Exceptions/Interrupts

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252