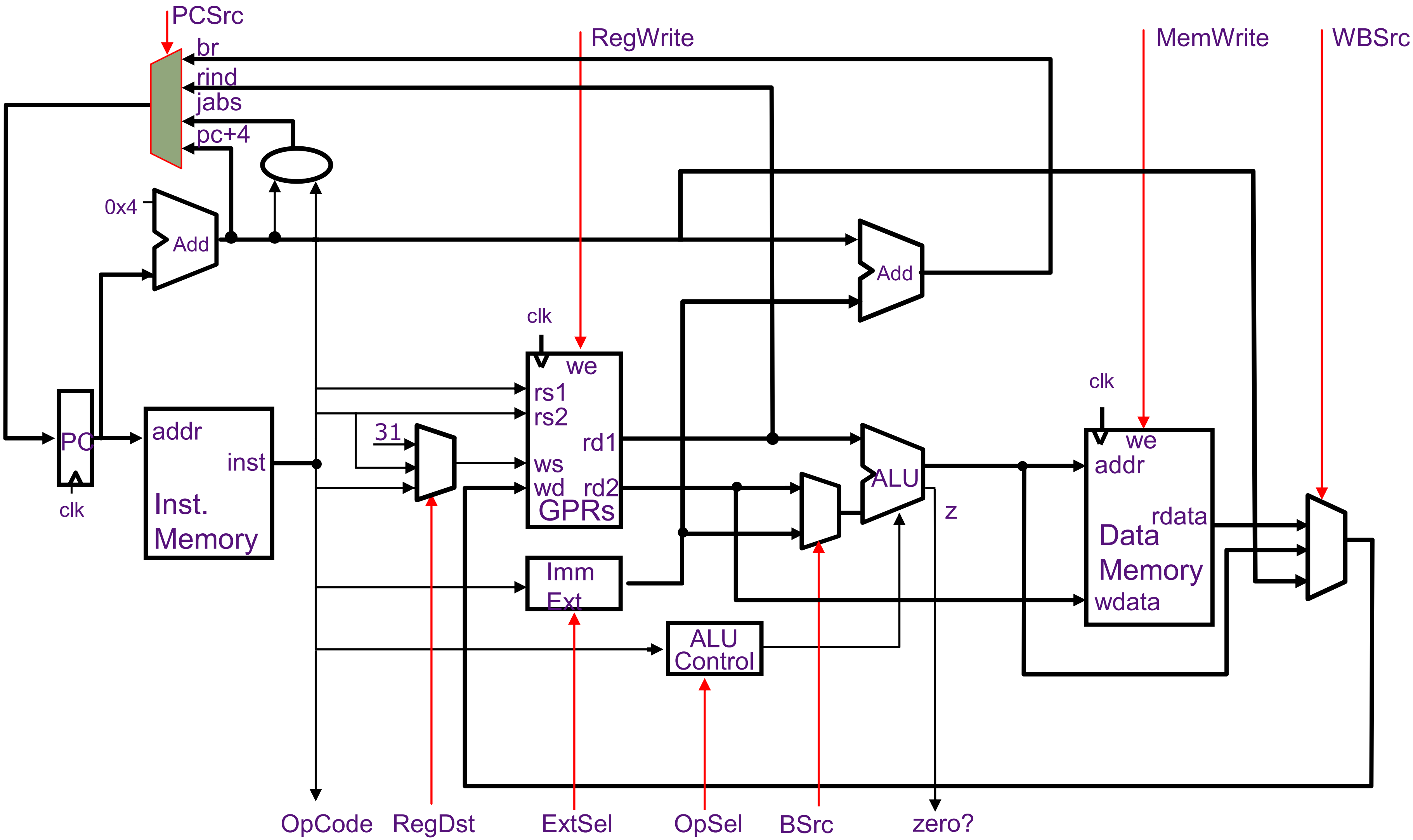
Instruction Pipelining and Hazards

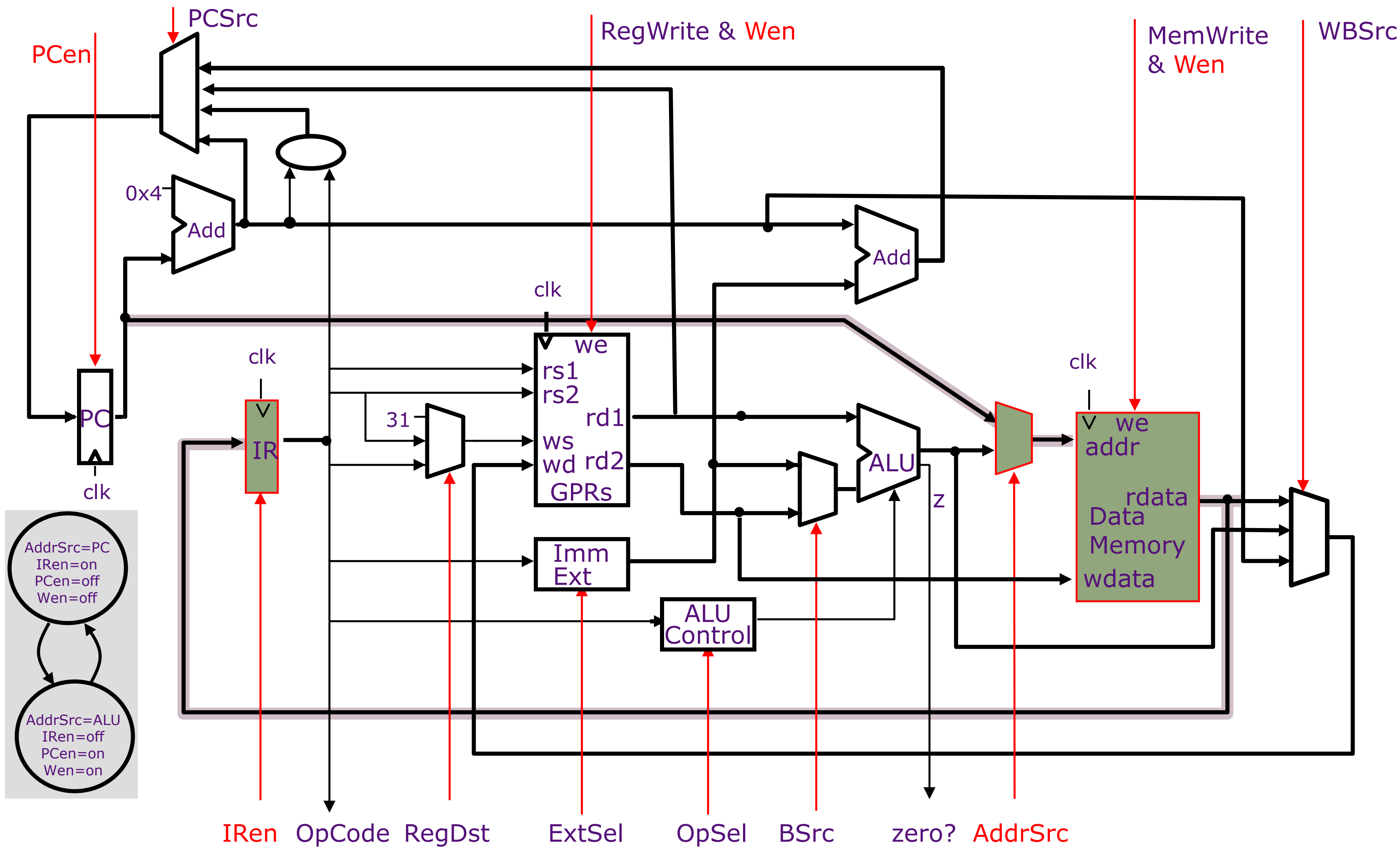
Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Reminder: Harvard-Style Single-Cycle Datapath for MIPS

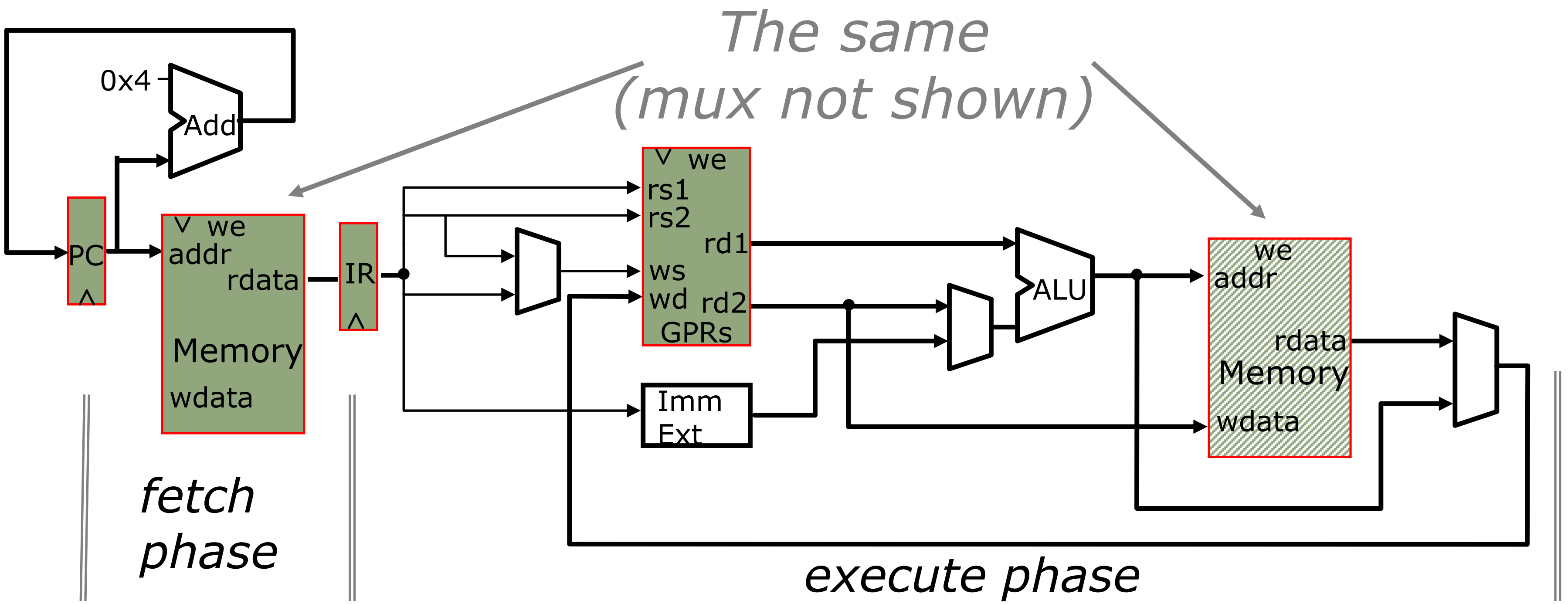


Reminder: Princeton Microarchitecture

Datapath & Control for 2 cycles-per-instruction



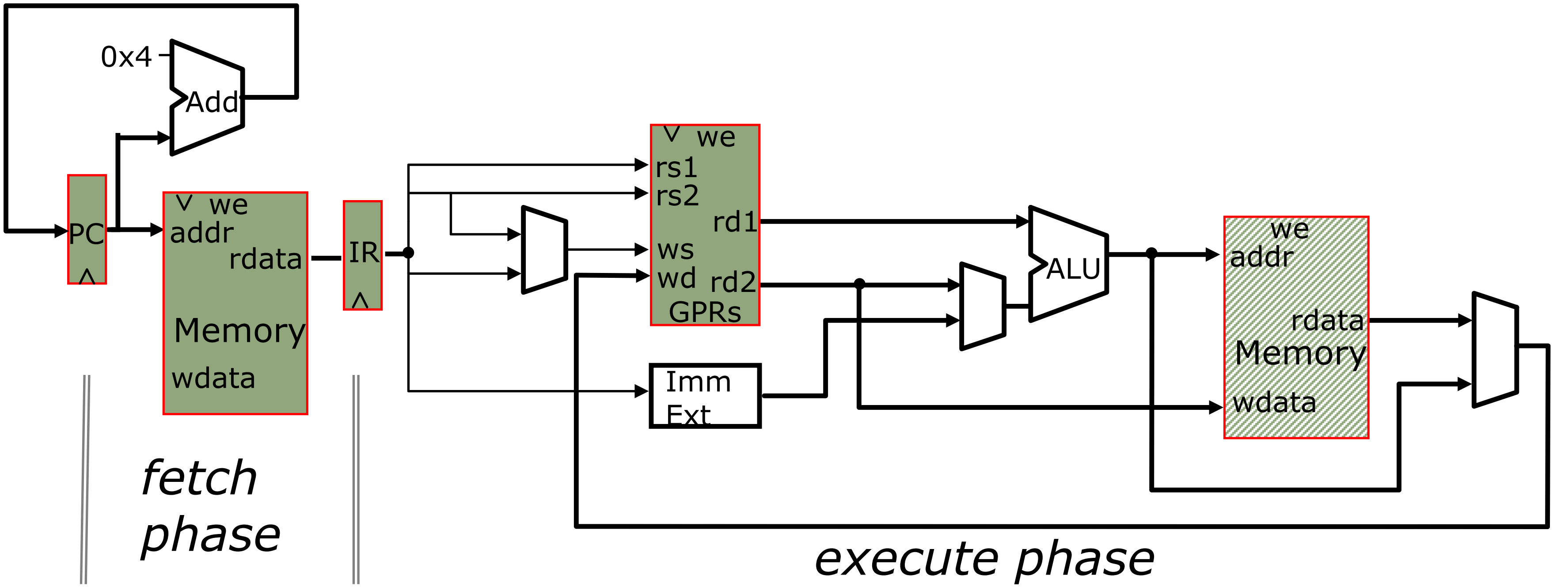
Princeton Microarchitecture
(redrawn)



Only one of the phases is active in any cycle
⇒ a lot of datapath not used at any given time

Princeton Microarchitecture

Overlapped execution



Can we overlap instruction fetch and execute?

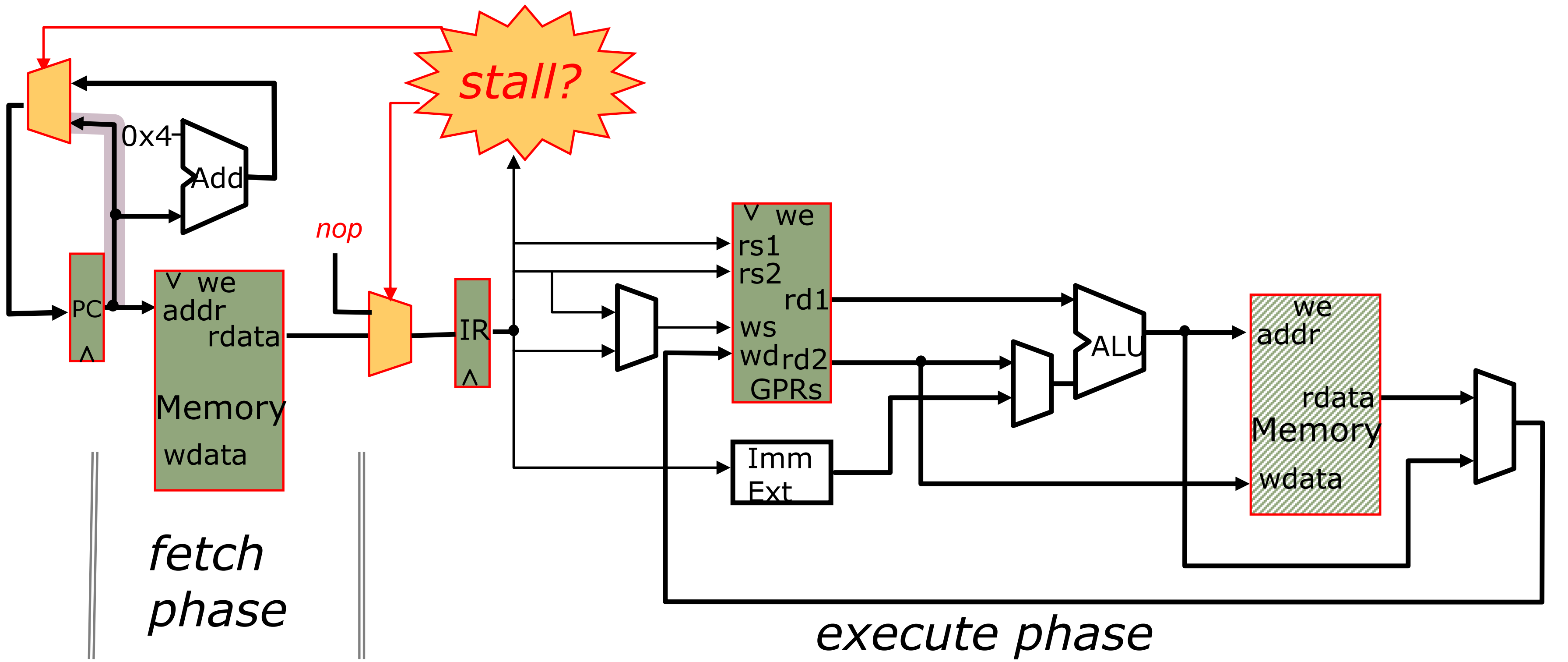
Yes, unless IR contains a Load or Store

Which action should be prioritized? *Execute*

What do we do with Fetch? *Stall it* *How?*

Stalling the instruction fetch

Princeton Microarchitecture

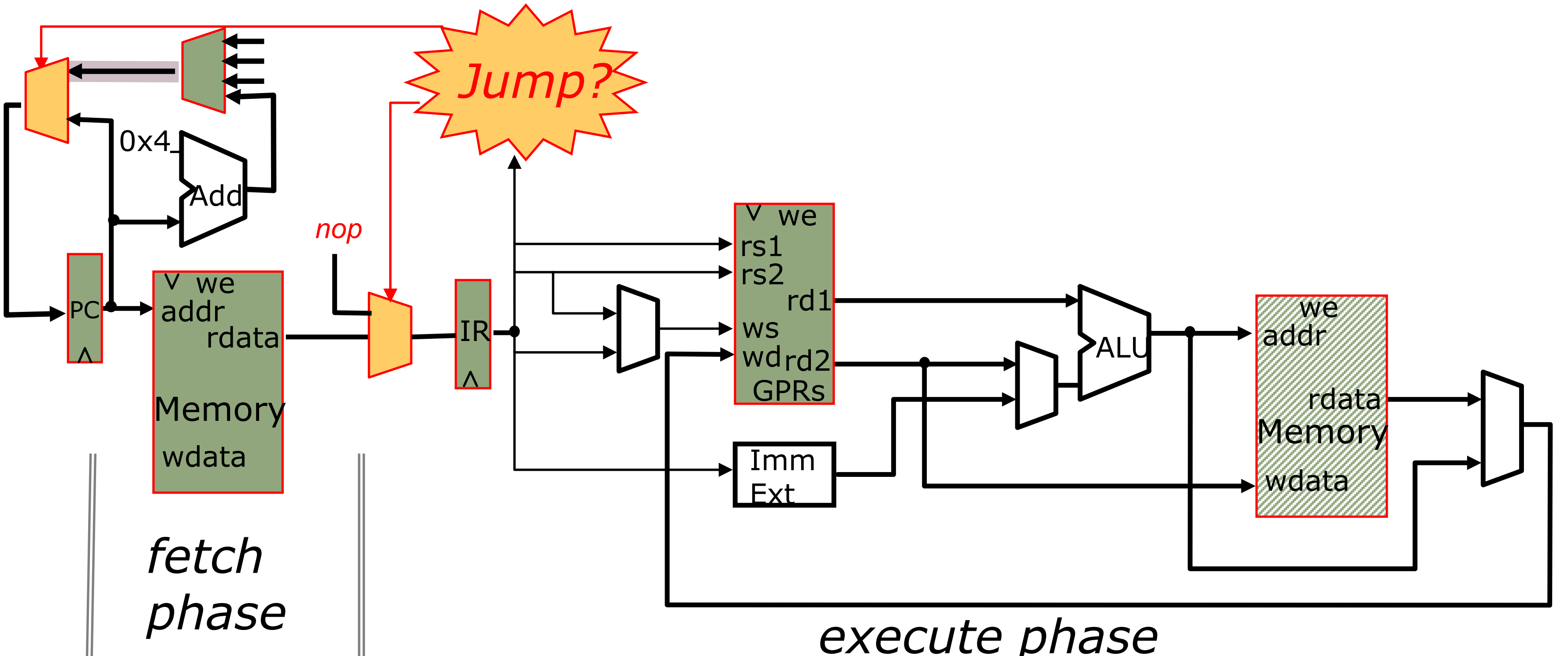


When stall condition is indicated

- *don't fetch a new instruction and don't change the PC*
- *insert a nop in the IR*
- *set the Memory Address mux to ALU (not shown)*

What if IR contains a jump or branch instruction?

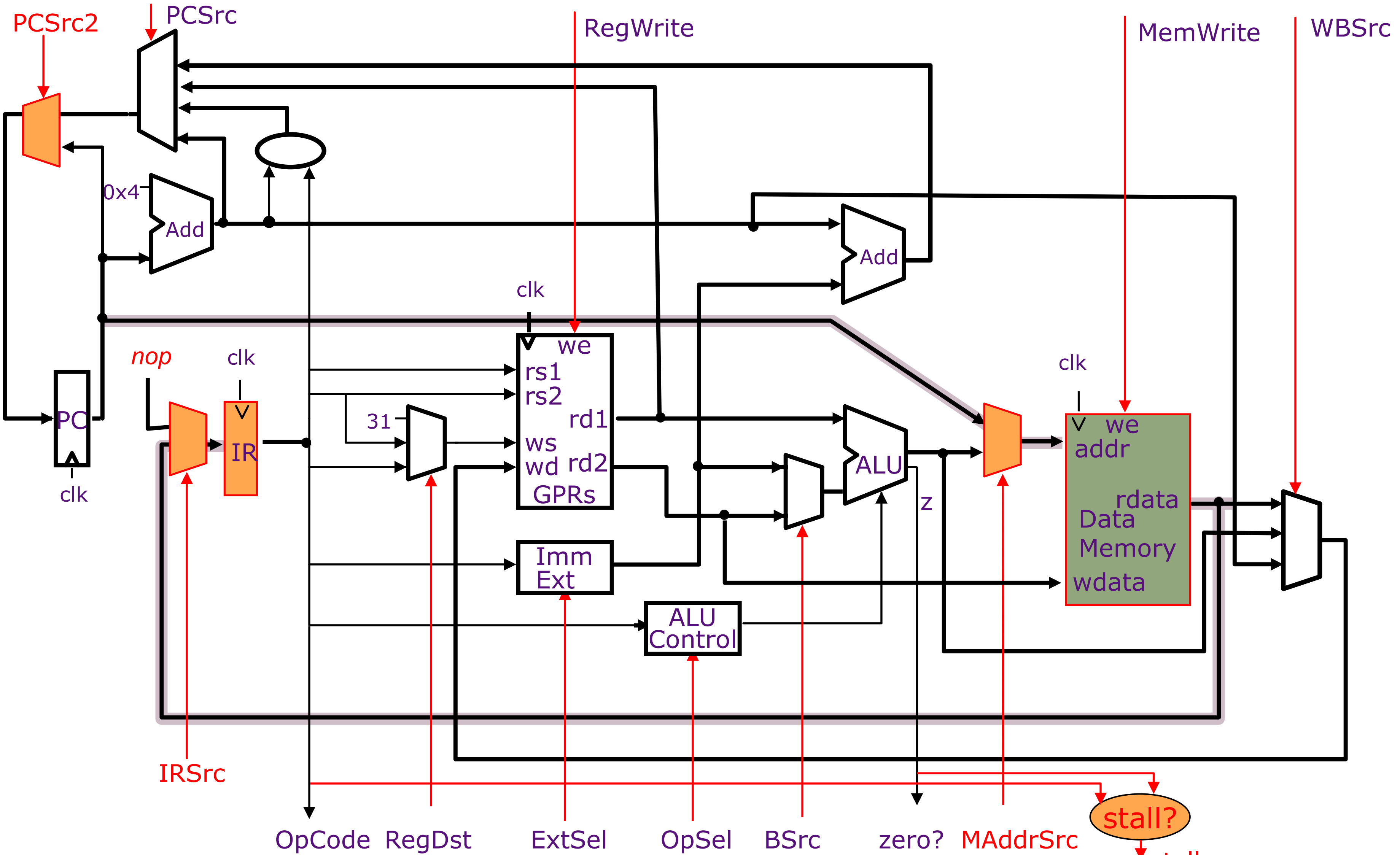
Need to stall on branches
Princeton Microarchitecture



When IR contains a jump or taken branch

- *no "structural conflict" for the memory*
- *but we do not have the correct PC value in the PC*
- *memory cannot be used – Address Mux setting is irrelevant*
- *insert a nop in the IR*
- *insert the nextPC (branch-target) address in the PC*

Pipelined Princeton Microarchitecture



Pipelined Princeton: Control Table

Opcode	Stall	Ext Sel	B Src	Op Sel	Mem W	Reg W	WB Src	Reg Dst	PC Src1	PC Src2	IR Src	MAddr Src
ALU	no	*	Reg	Func	no	yes	ALU	rd	pc+4	npc	mem	pc
ALUi	no	sE_{16}	Imm	Op	no	yes	ALU	rt	pc+4	npc	mem	pc
ALUiu	no	uE_{16}	Imm	Op	no	yes	ALU	rt	pc+4	npc	mem	pc
LW	yes	sE_{16}	Imm	+	no	yes	Mem	rt	pc+4	pc	nop	ALU
SW	yes	sE_{16}	Imm	+	yes	no	*	*	pc+4	pc	nop	ALU
$BEQZ_{z=1}$	yes	sE_{16}	*	0?	no	no	*	*	br	npc	nop	*
$BEQZ_{z=0}$	no	sE_{16}	*	0?	no	no	*	*	pc+4	npc	mem	pc
J	yes	*	*	*	no	no	*	*	jabs	npc	nop	*
JAL	yes	*	*	*	no	yes	PC	R31	jabs	npc	nop	*
JR	yes	*	*	*	no	no	*	*	rind	npc	nop	*
JALR	yes	*	*	*	no	yes	PC	R31	rind	npc	nop	*
NOP	no	*	*	*	no	no	*	*	pc+4	npc	mem	pc

BSrc = Reg / Imm ; WBSrc = ALU / Mem / PC; IRSrc = nop/mem; MAddSrc = pc/ALU
RegDst = rt / rd / R31; PCSrc1 = pc+4 / br / rind / jabs; PCSrc2 = pc/nPC

stall & IRSrc columns are identical

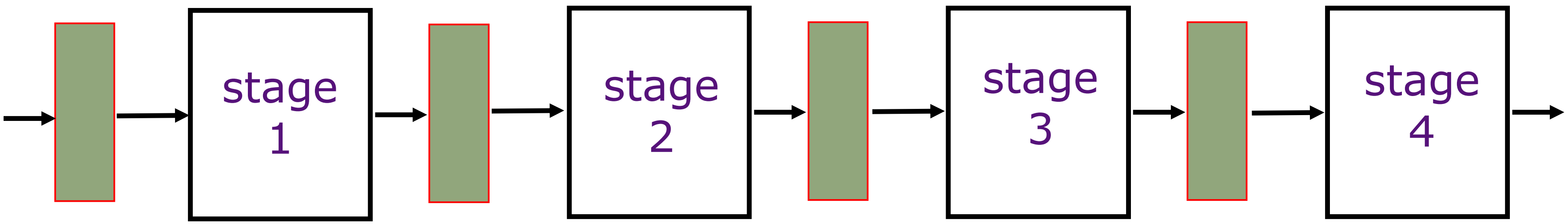
Pipelined Princeton Architecture

Clock: $t_{C\text{-}Princeton} > t_{RF} + t_{ALU} + t_M + t_{WB}$

CPI: $(1 - f) + 2f$ cycles per instruction
where f is the fraction of
instructions that cause a stall

What is a likely value of f?

An Ideal Pipeline

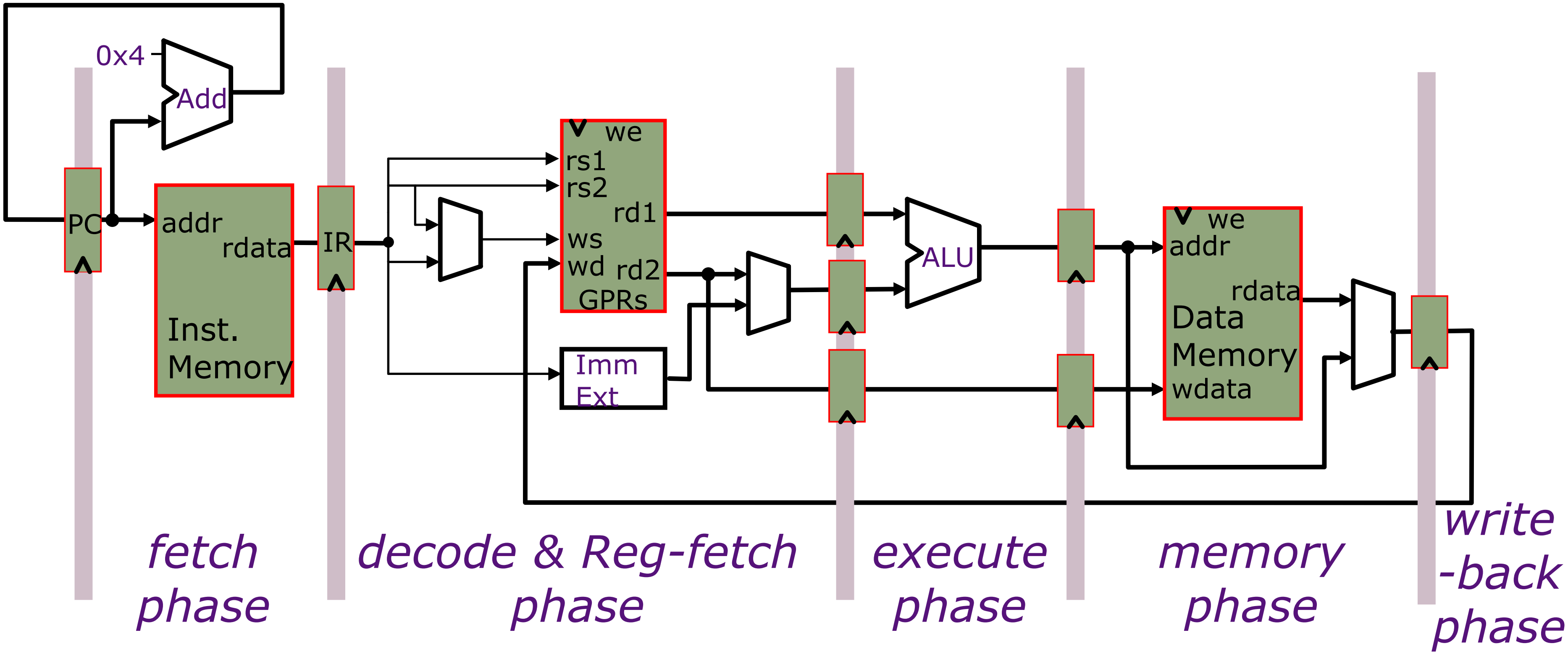


- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

These conditions generally hold for industrial assembly lines.

But what about an instruction pipeline?

Pipelined Datapath



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

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

However, CPI will increase unless instructions are pipelined

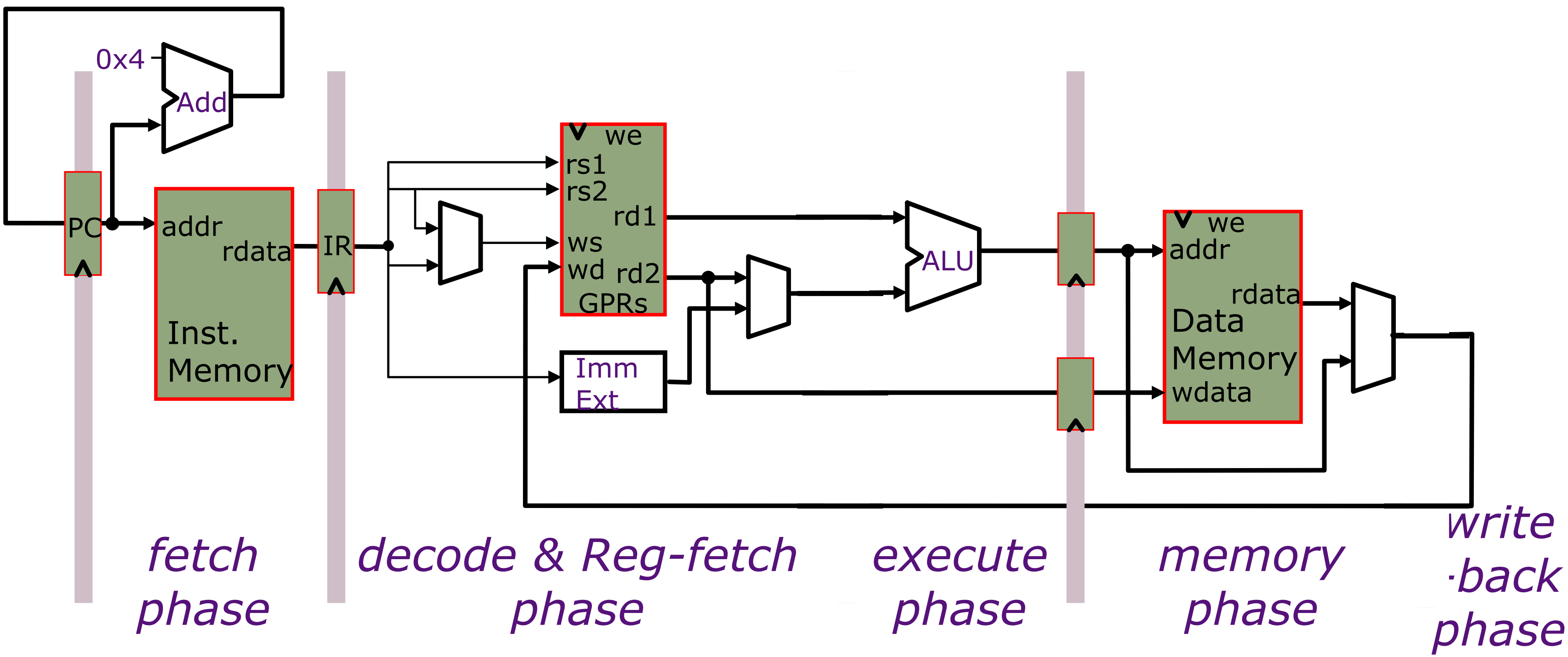
How to divide datapath into stages

Suppose memory is significantly slower than other stages. For example, suppose

t_{IM} = 10 units
t_{DM} = 10 units
t_{ALU} = 5 units
t_{RF} = 1 unit
t_{RW} = 1 unit

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance

Alternative Pipelining



$t_C > \max \{t_{IM}, t_{RF}+t_{ALU}, t_{DM}+t_{RW}\} = t_{DM}+ t_{RW}$

$\Rightarrow$ *increase the critical path by 10%*

Write-back stage takes much less time than other stages. Suppose we combined it with the memory phase

Maximum Speedup by Pipelining

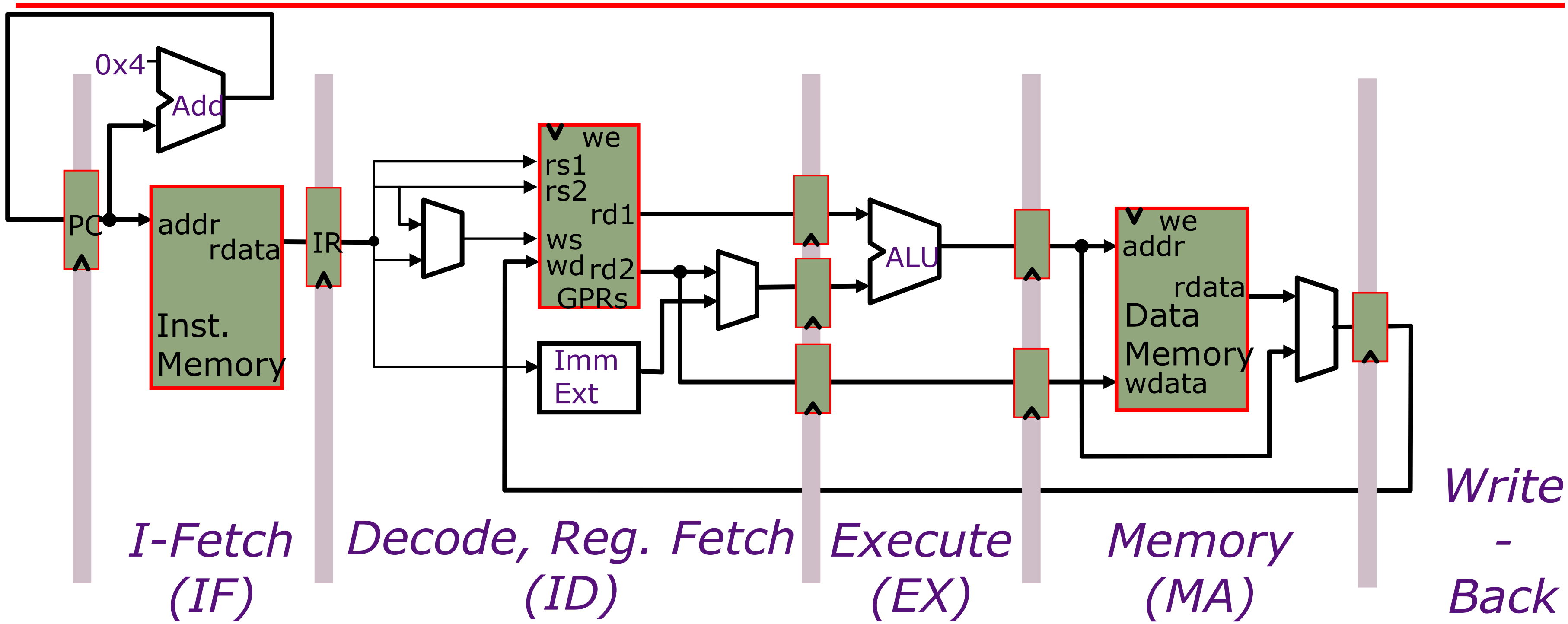
Assumptions	Unpipelined t_C	Pipelined t_C	Speedup
1. $t_{IM} = t_{DM} = 10$, $t_{ALU} = 5$, $t_{RF} = t_{RW} = 1$ 4-stage pipeline	27	10	2.7
2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 4-stage pipeline	25	10	2.5
3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 5-stage pipeline	25	5	5.0

What seems to be the message here?

One can achieve higher speedup with more pipeline stages

5-Stage Pipelined Execution

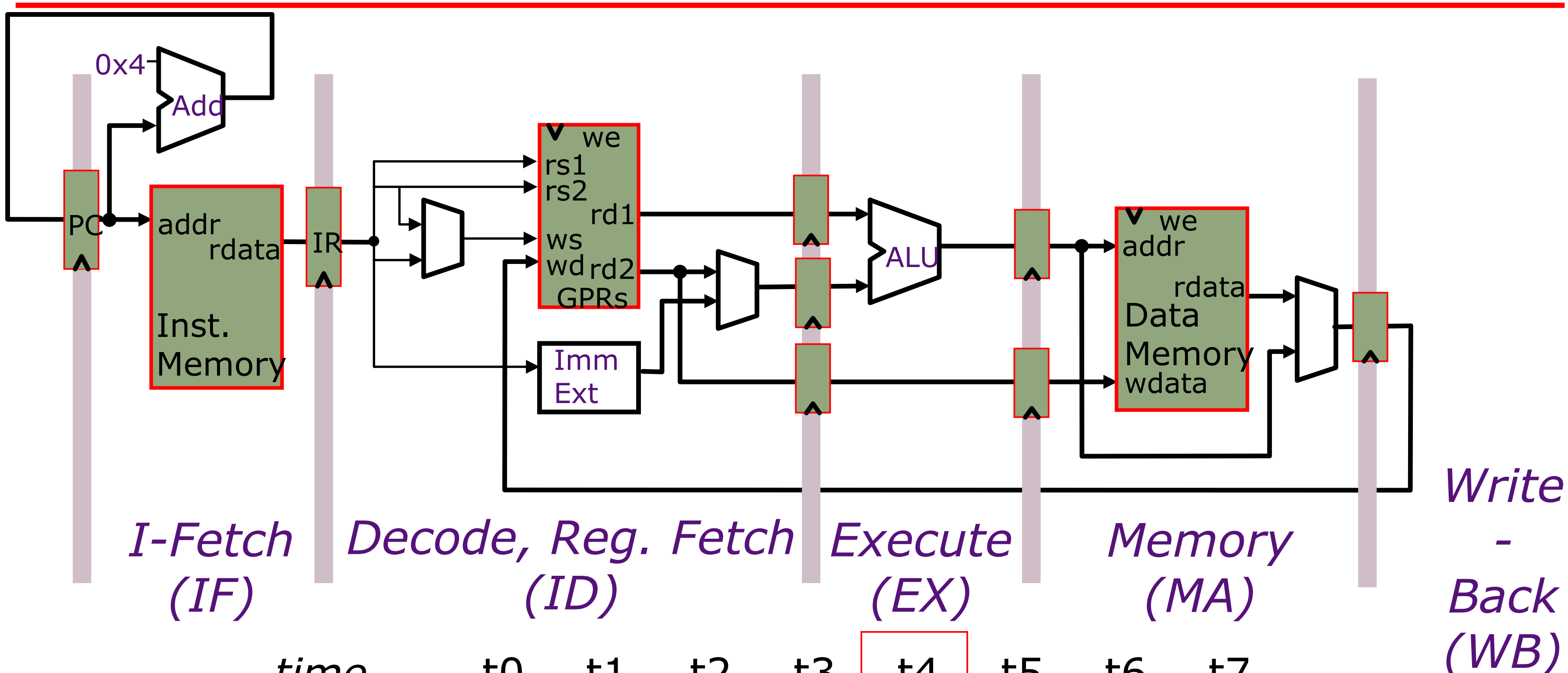
Instruction Flow Diagram



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

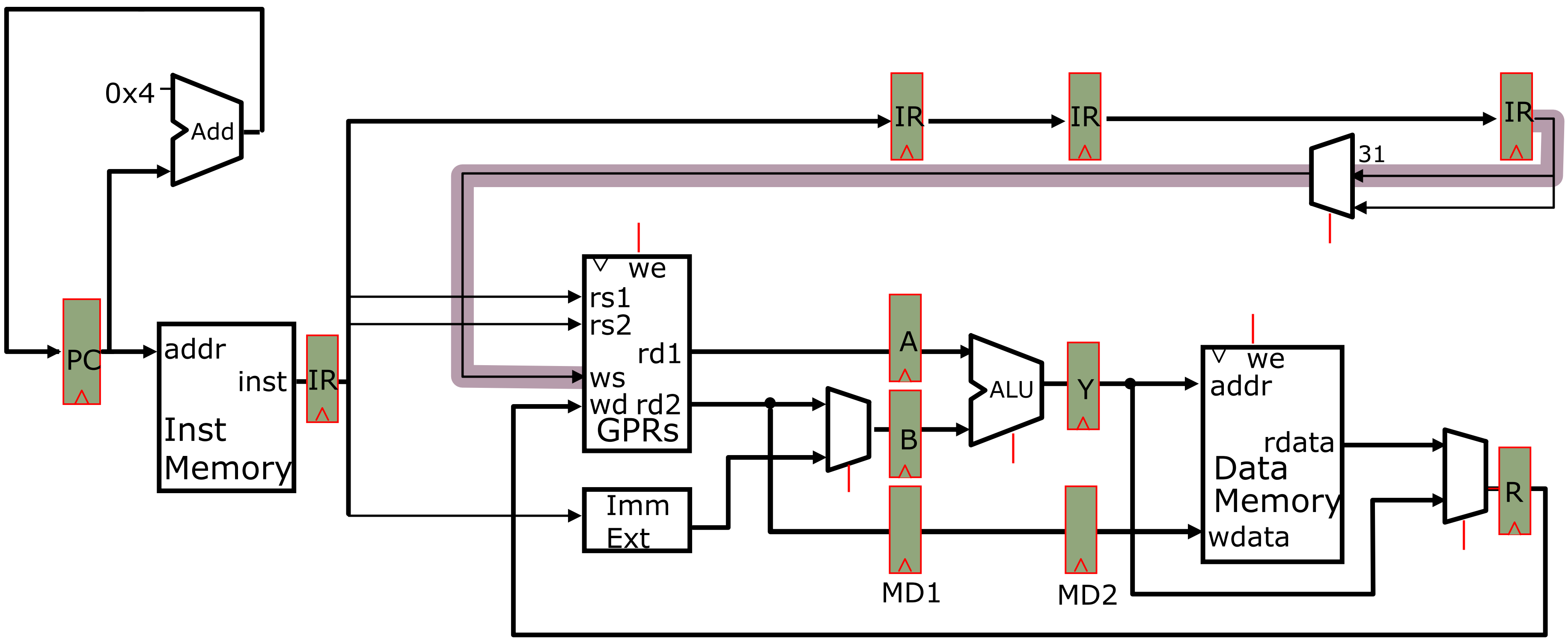
Resource Usage Diagram



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

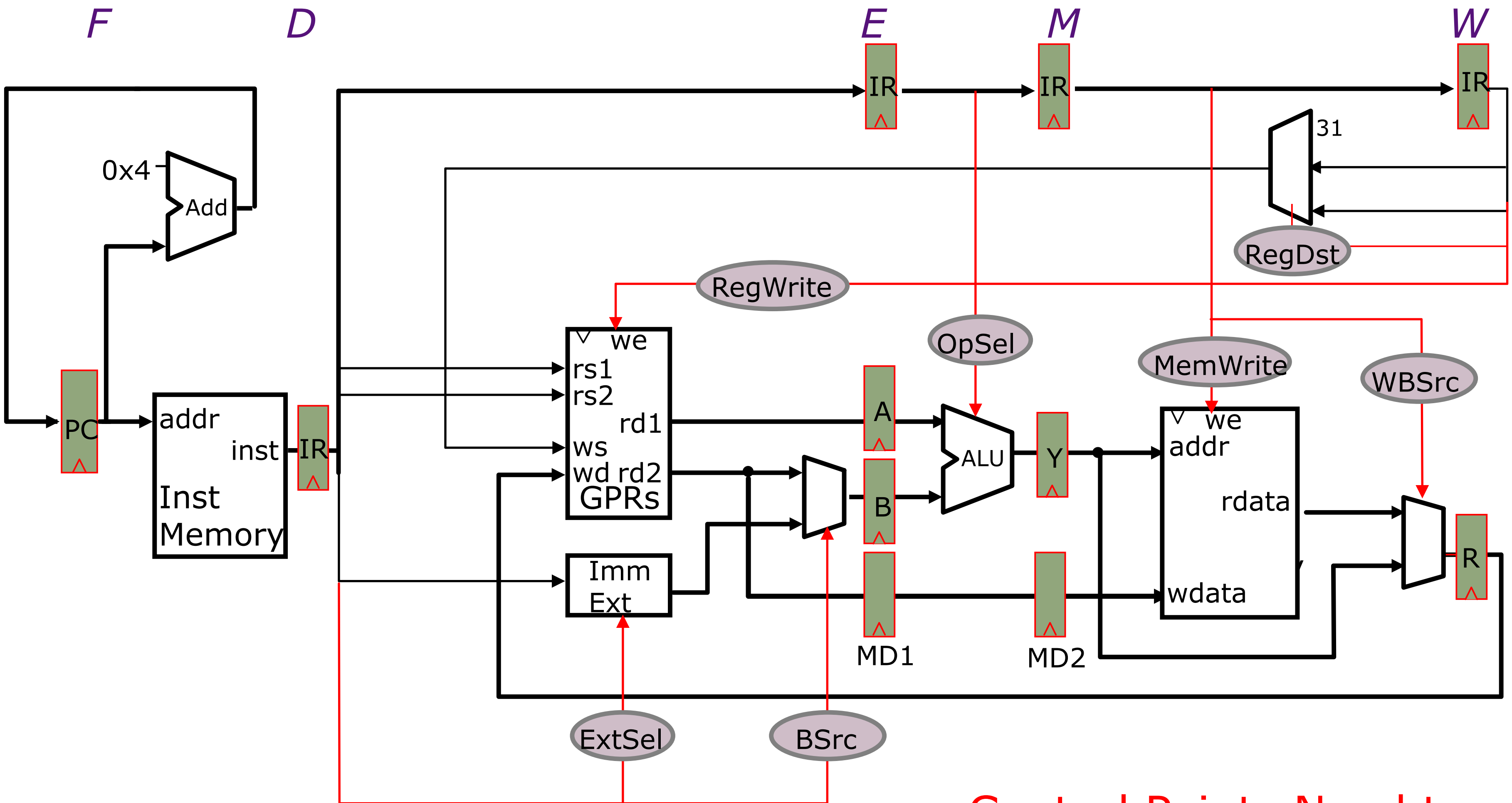
Pipelined Execution
ALU Instructions



Not quite correct!

We need an Instruction Reg (IR) for each stage

Pipelined MIPS Datapath
without jumps



Control Points Need to Be Connected

What else is needed?

How instructions can interact with each other in a pipeline

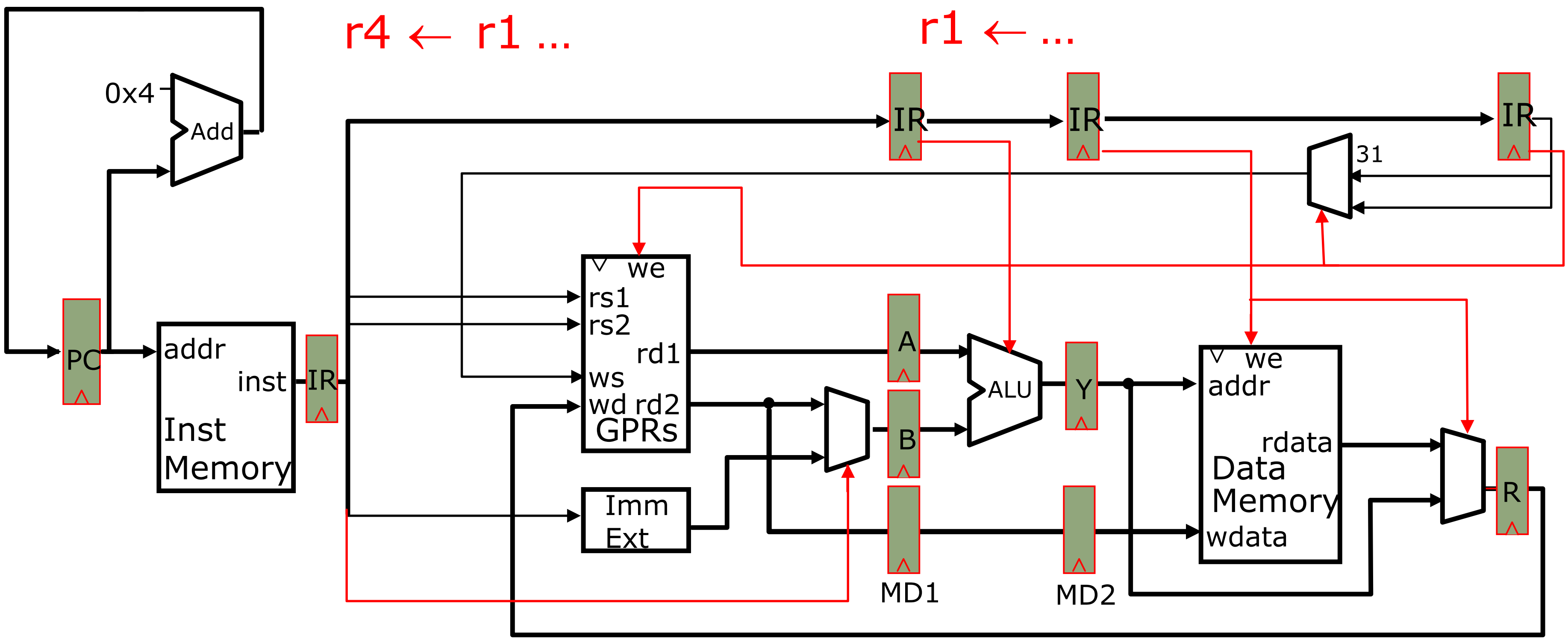
- An instruction in the pipeline may need a resource being used by another instruction in the pipeline
  → *structural hazard*

- 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



…

r1 ← r0 + 10

r4 ← r1 + 17

…

r1 is stale. Oops!

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 → do nothing

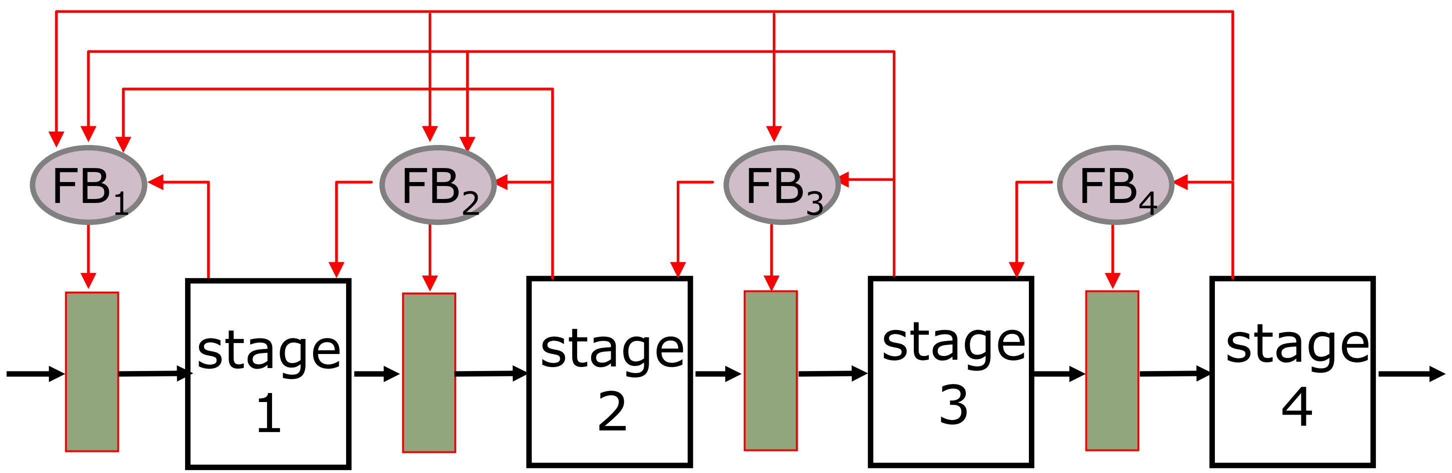
Guessed incorrectly → kill and restart

Resolving Data Hazards (1)

Strategy 1:

Wait for the result to be available by freezing earlier pipeline stages → stall

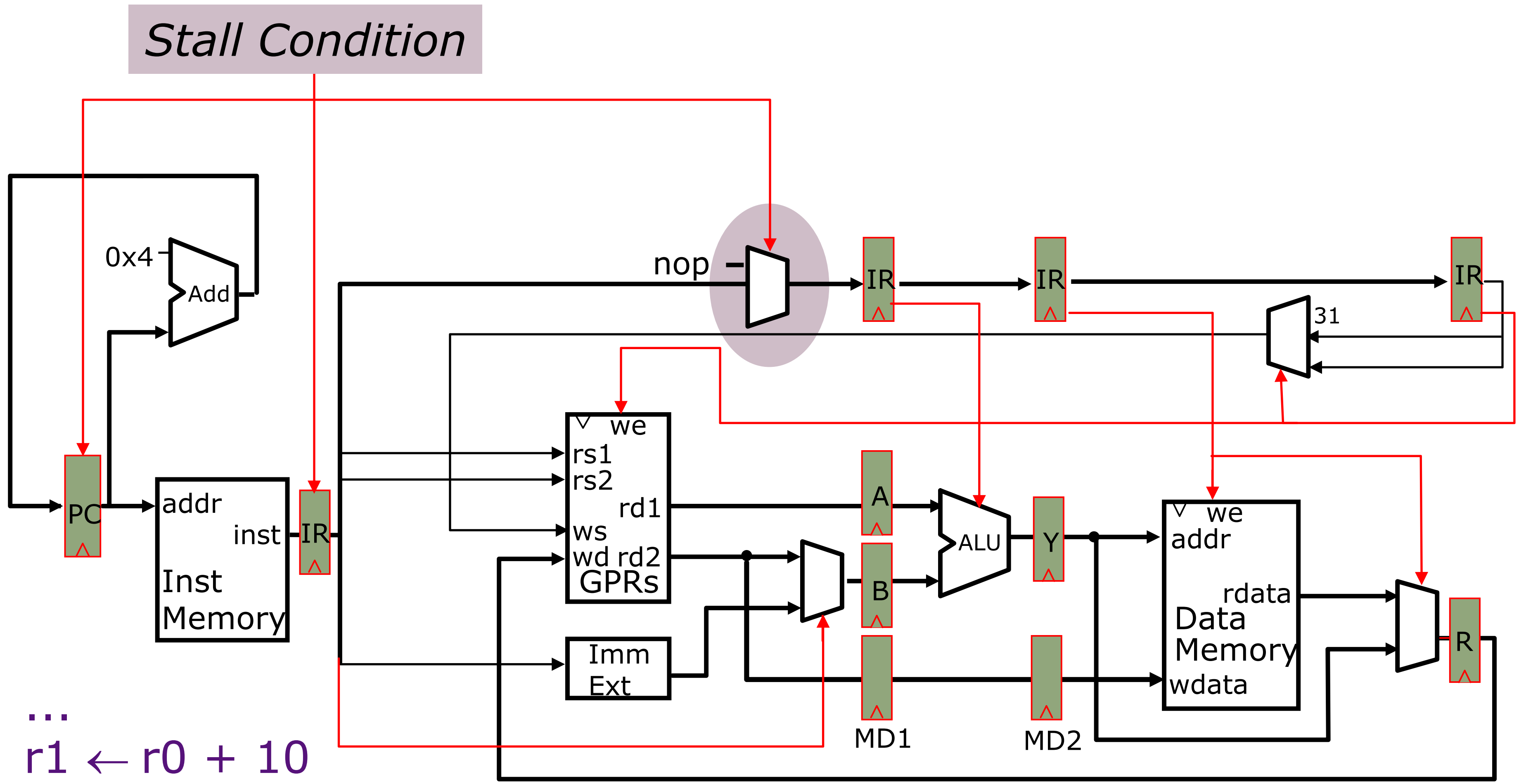
Feedback to Resolve Hazards



- 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)

Resolving Data Hazards by Stalling



...

r1 ← r0 + 10

r4 ← r1 + 17

...

Stalled Stages and Pipeline 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) 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	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

stalled stages

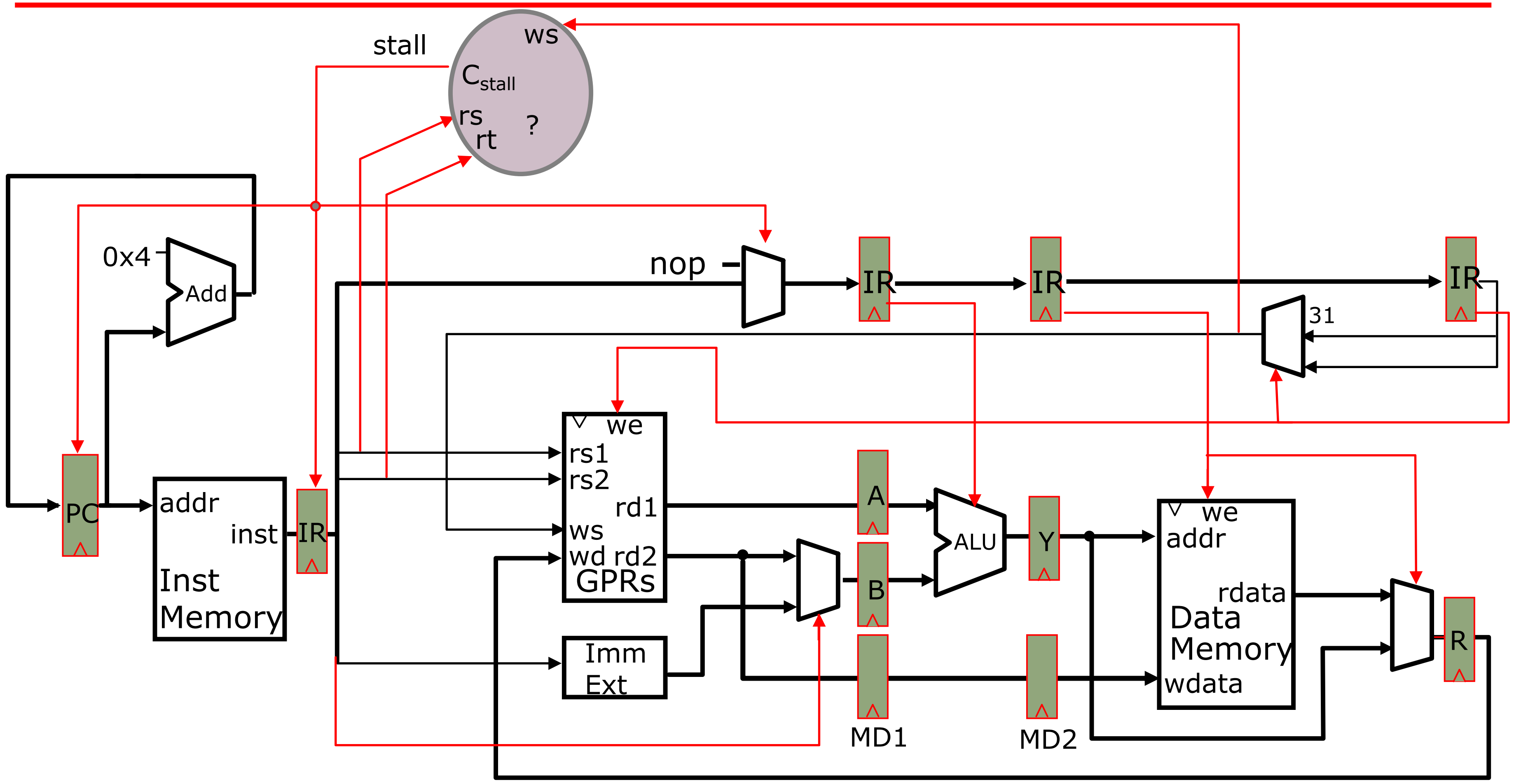
Resource Usage

time

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

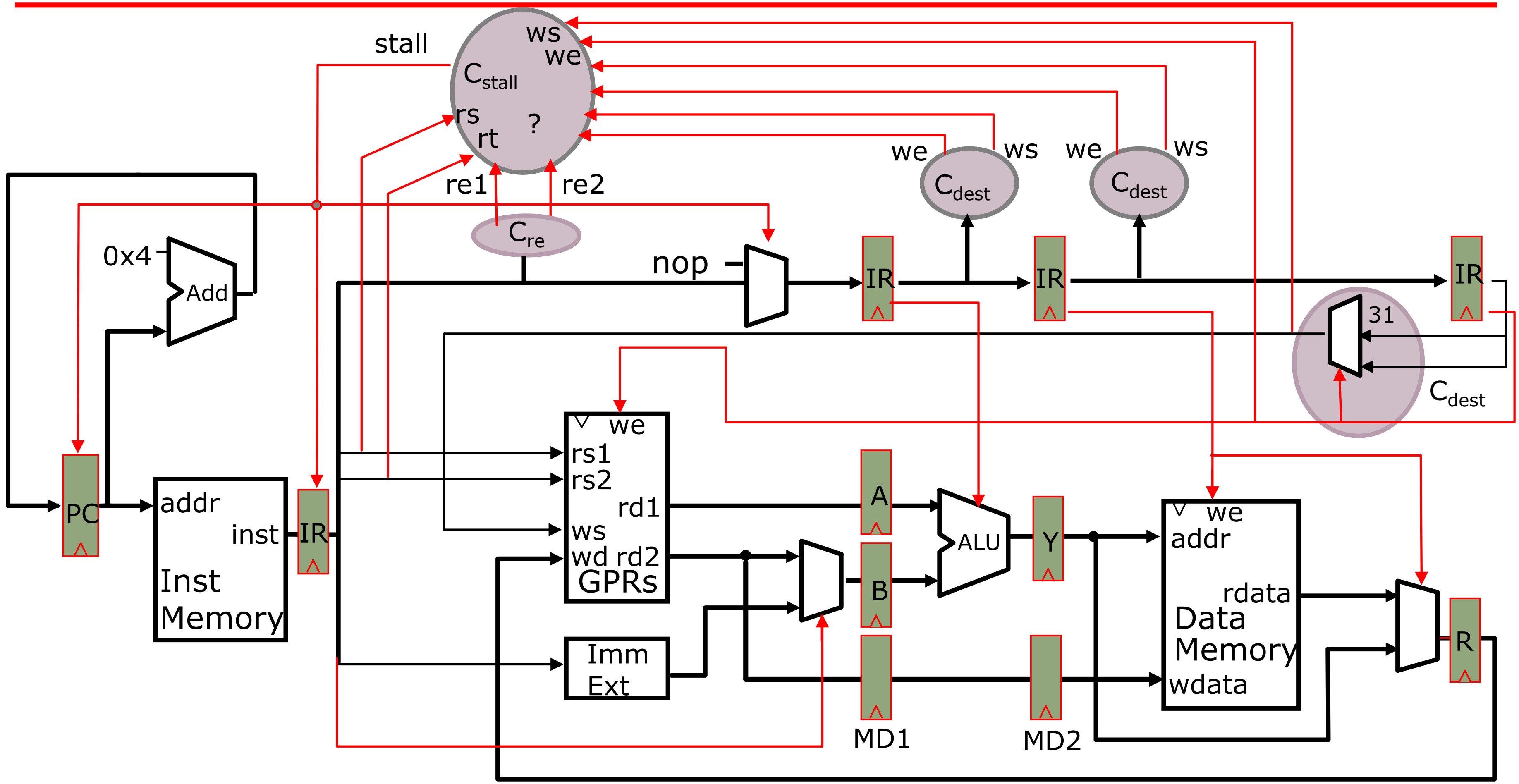
nop ⇒ *pipeline bubble*

Stall Control Logic



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

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

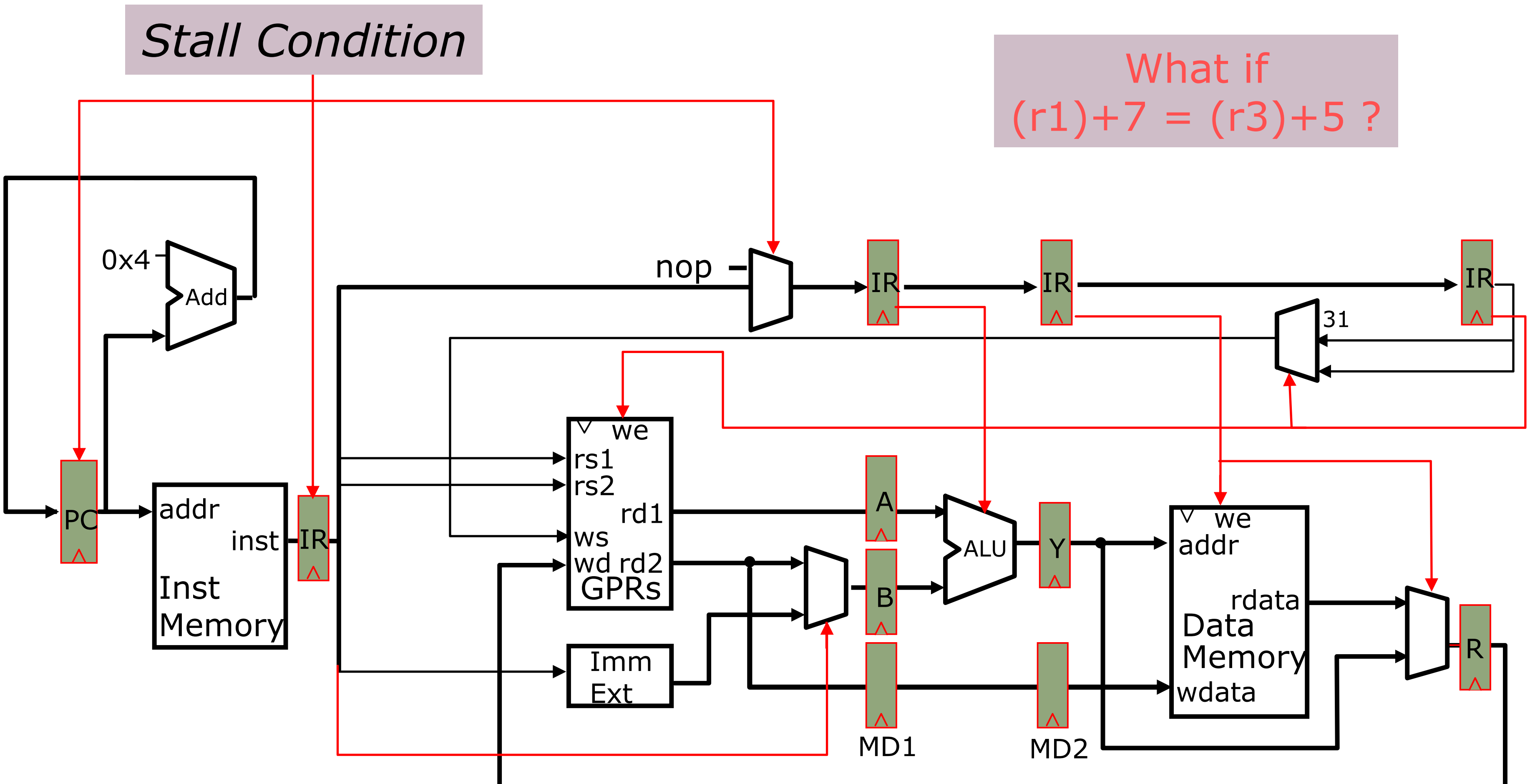
re2 = *Case* opcode
- ALU, SW ⇒ on
- ... ⇒ off

C_{stall}

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

This is not the full story !

Hazards due to Loads & Stores



...
M[(r1)+7] ← (r2)
r4 ← M[(r3)+5]
...

Is there any possible data hazard in this instruction sequence?

Load & Store Hazards

...
M[(r1)+7] $\leftarrow$ (r2)
r4 $\leftarrow$ M[(r3)+5]
...

(r1)+7 = (r3)+5 $\Rightarrow$ *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.

Next lecture: Control Hazards, Bypassing, and Speculation