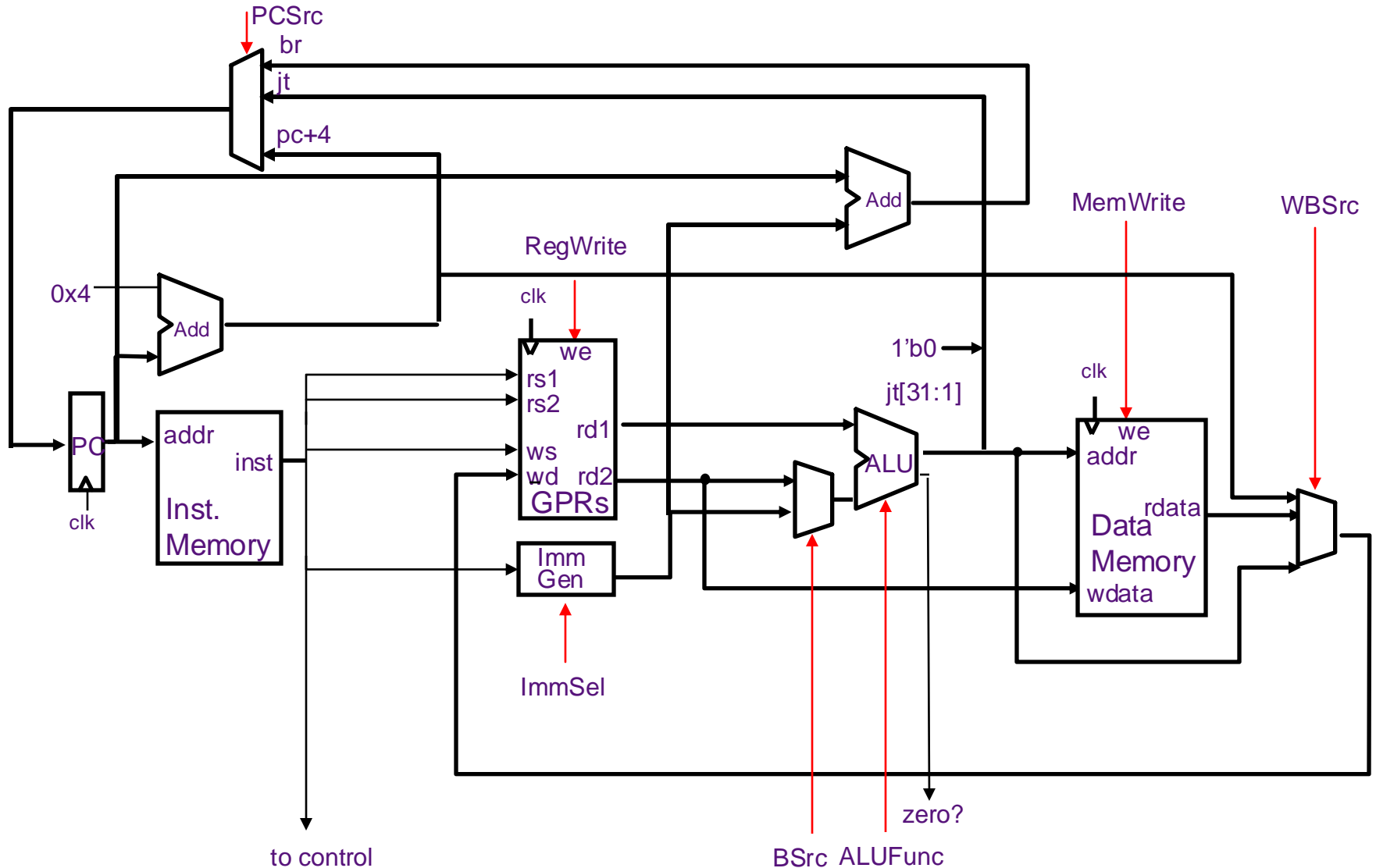# Instruction Pipelining

Axel Feldmann

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

Announcements

- Lab 1 released later today
  - Designing 3 different cache models using Pin
  - Due Sept. 27

- Please view the Pin tutorial video posted on Piazza

- Contact me if you cannot get access to lab machines

Reminder: Harvard-Style Single-Cycle Datapath for RISC-V

Single-Cycle Hardwired Control:
Harvard architecture

We will assume

- Clock period is sufficiently long for all of the following steps to be "completed":

    1. instruction fetch
    2. decode and register fetch
    3. ALU operation
    4. data fetch if required
    5. register write-back setup time

$$\Rightarrow\ t_C >\ t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$$

- At the rising edge of the following clock, the PC, the register file and the memory are updated

Datapath for Memory Instructions

Should program and data memory be separate?

Harvard style: separate (Aiken and Mark 1 influence)
- read-only program memory
- read/write data memory

- Note:
  There must be a way to load the program memory

Princeton style: the same (von Neumann's influence)
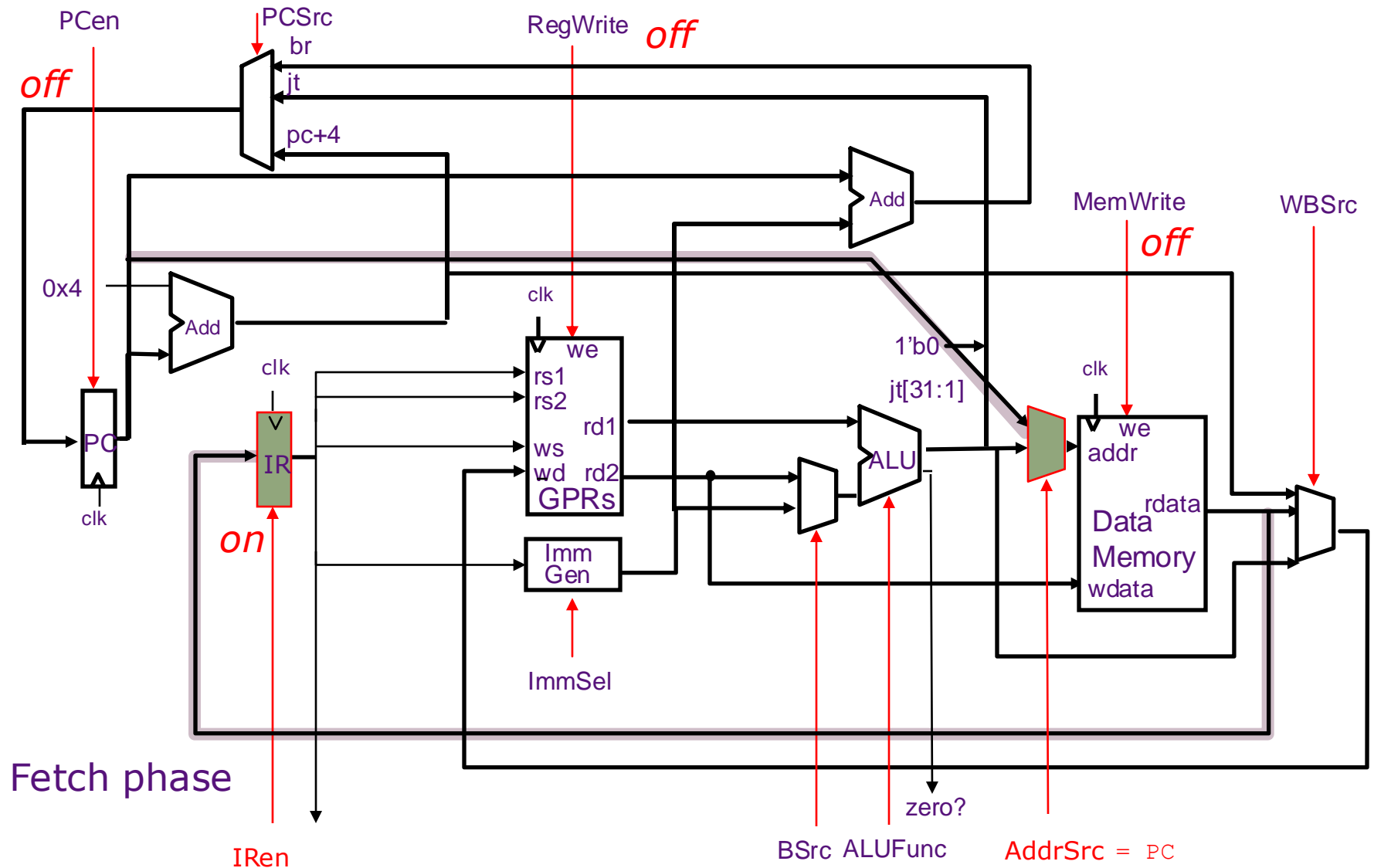- single read/write memory for program and data

- Note:
  Executing a Load or Store instruction requires accessing the memory more than once

Princeton challenge

- What problem arises if instructions and data reside in the same memory?

Princeton Microarchitecture
Datapath & Control

PCen

PCSrc
br
off
jt
pc+4

off

RegWrite
off

MemWrite
off

WBSrc

0x4

Add

clk

clk

IR

on

we
rs1
rs2
rd1
ws
wd rd2
GPRs

clk

Add

1'b0

clk

jt[31:1]

ALU

Imm
Gen

Data
Memory

we
addr

rdata

wdata

ImmSel

Fetch phase

IRen

zero?

BSrc ALUFunc

AddrSrc = PC

September 13, 2024

MIT 6.5900 Fall 2024

Two-State Controller:
Princeton Architecture

fetch phase

AddrSrc=PC
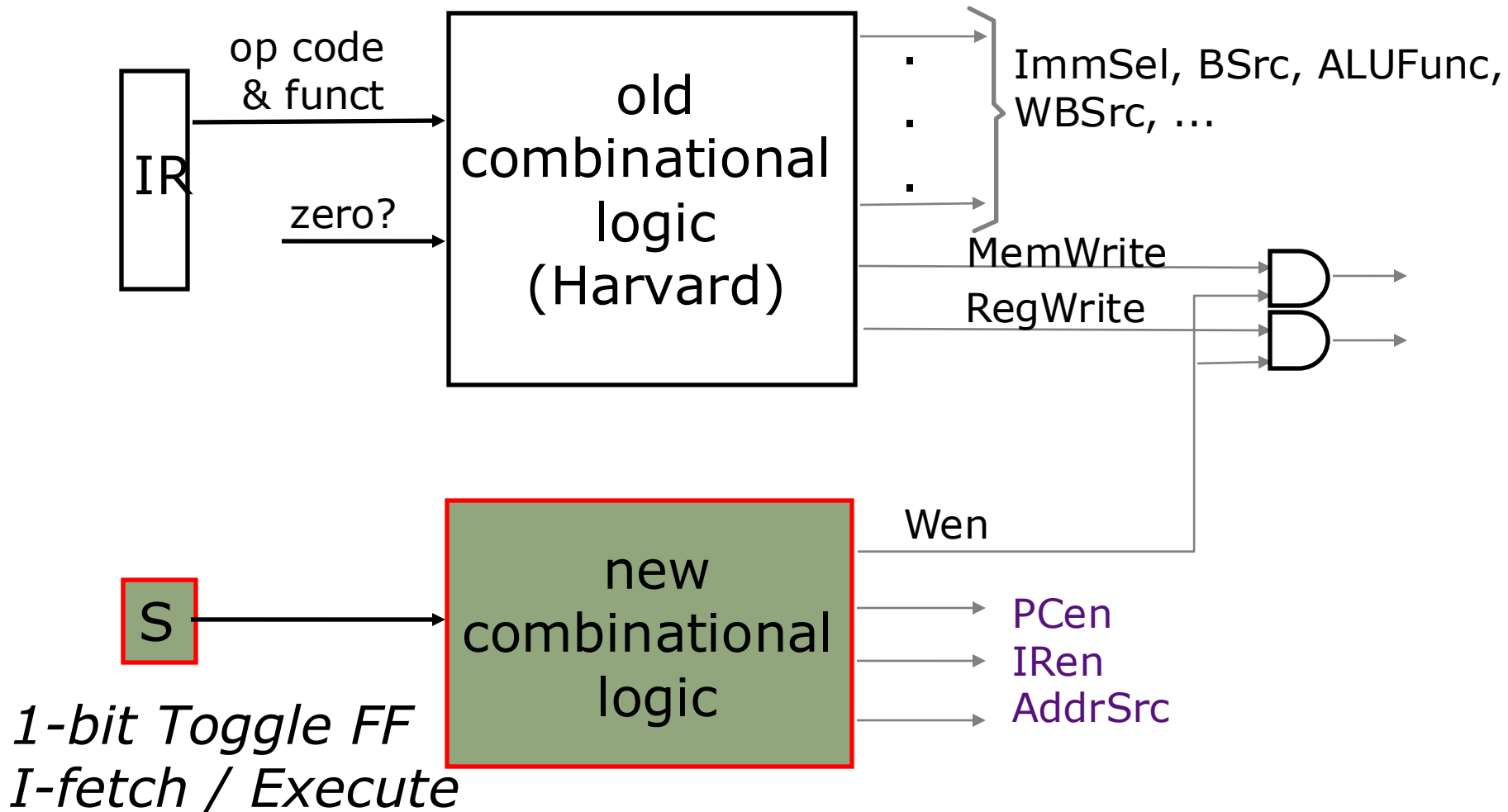IRen=on
PCen=off
Wen=off

execute phase

AddrSrc=ALU
IRen=off
PCen=on
Wen=on

A flipflop can be used to remember the phase

Hardwired Controller:
Princeton Architecture

Clock Rate vs CPI

$t_{C\text{-}Princeton} > \max \{ t_M , t_{RF} + t_{ALU} + t_M + t_{WB} \}$

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

$t_{C\text{-}Harvard} > t_M + t_{RF} + t_{ALU} + t_M + t_{WB}$

Suppose $t_M \gg t_{RF} + t_{ALU} + t_{WB}$

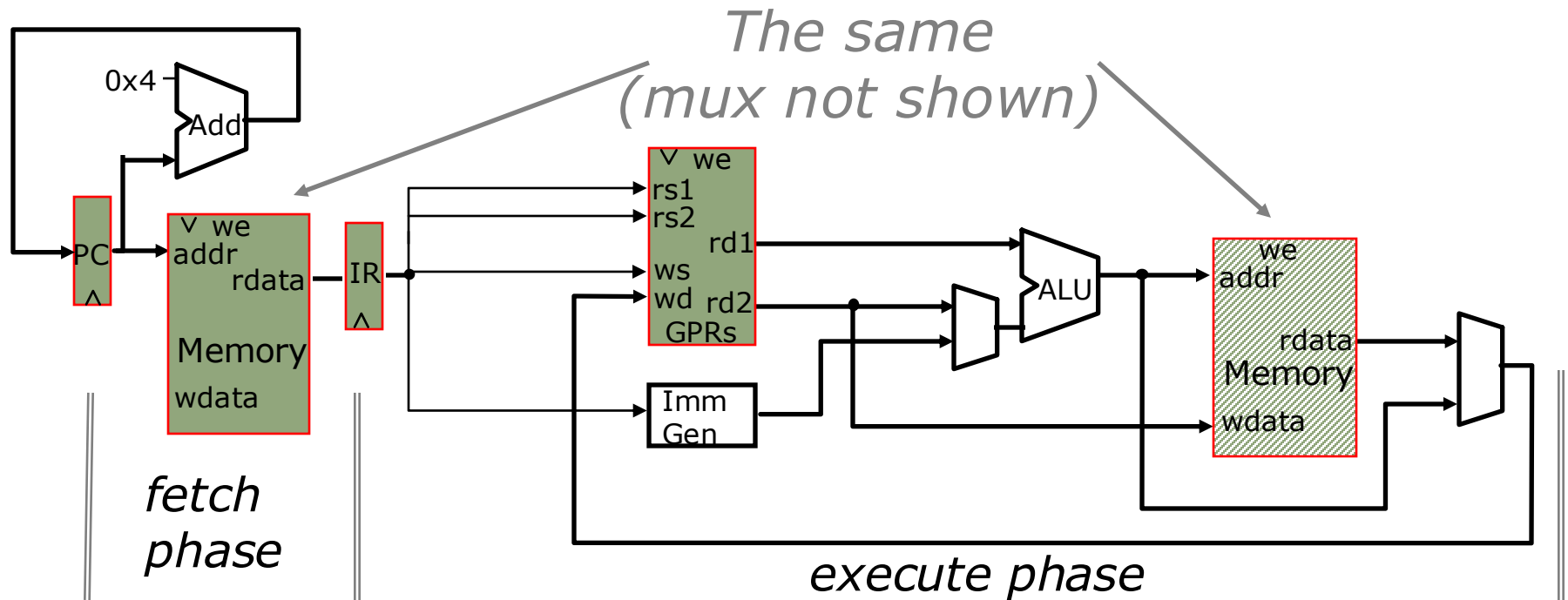$t_{C\text{-}Princeton} = 0.5 * t_{C\text{-}Harvard}$

$CPI_{Princeton} = 2$

$CPI_{Harvard} = 1$

No difference in performance!

Is it possible to design a controller for the Princeton architecture with CPI < 2 ?

CPI = Clock cycles Per Instruction

Princeton Microarchitecture
(redrawn)

The same
(mux not shown)

0x4	Add		

∨ we
rs1
rs2
rd1
ws
wd rd2
GPRs

∨ we
addr
rdata
PC
∧
Memory
wdata
IR
∧

Imm
Gen

ALU

we
addr
rdata
Memory
wdata

fetch
phase

execute phase

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

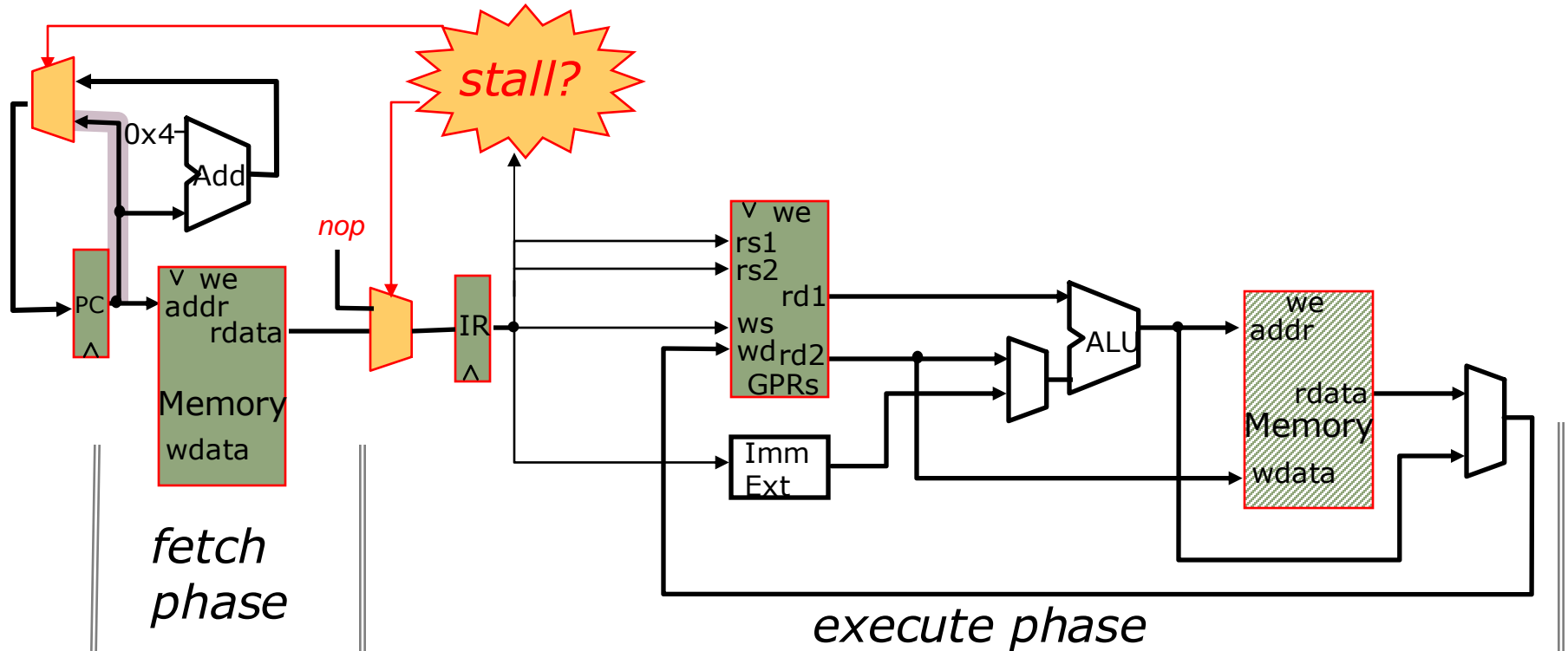# Princeton Microarchitecture
Overlapped execution
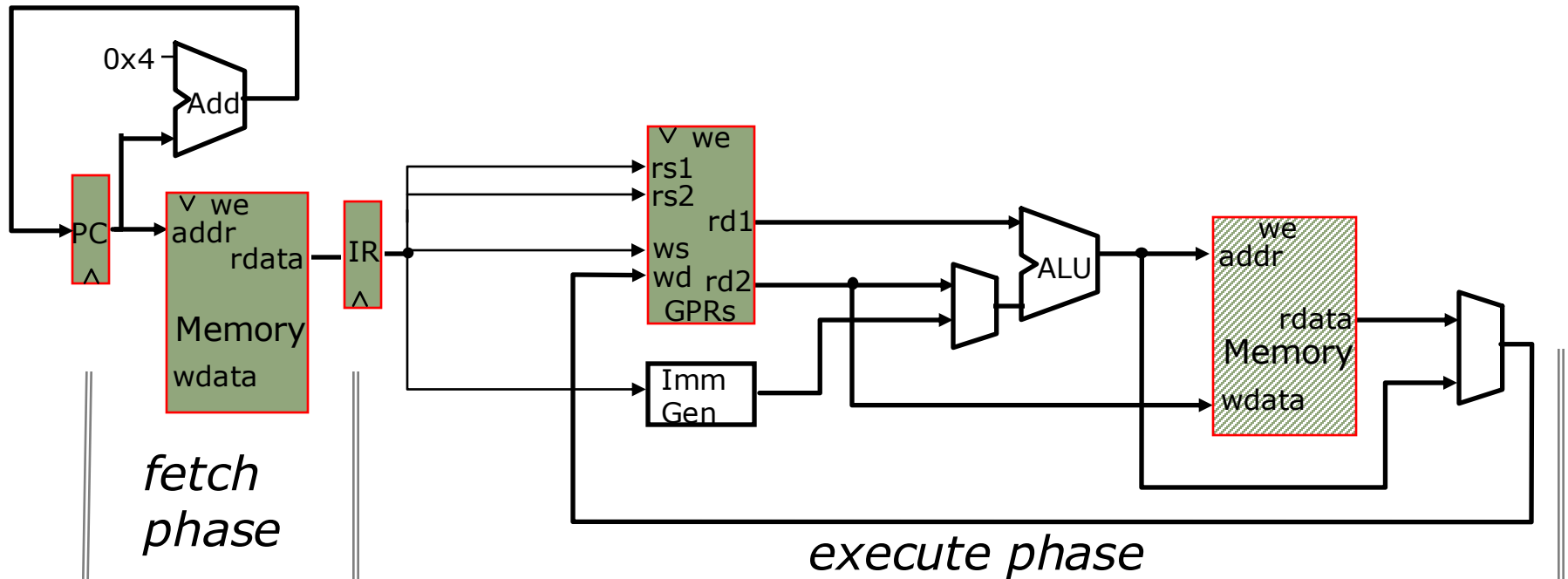
Can we overlap instruction fetch and execute?

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
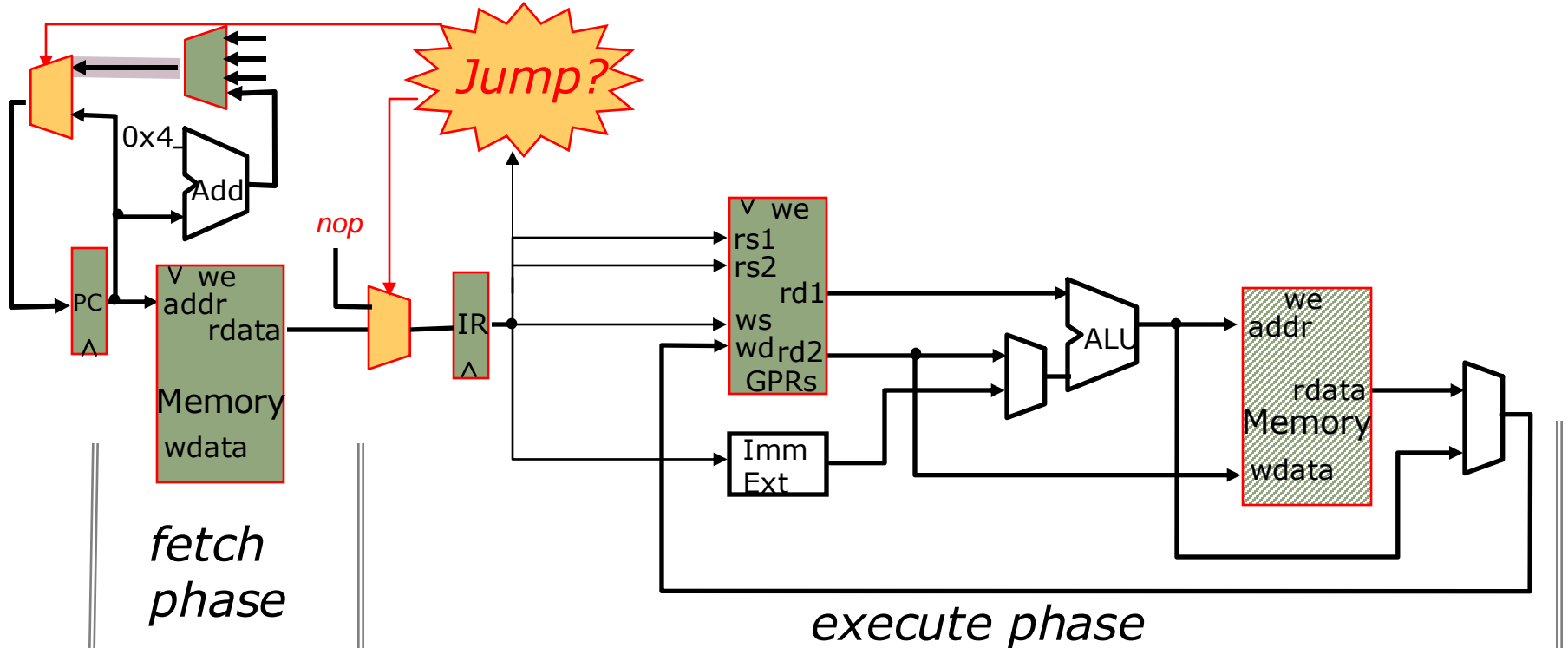## *Princeton Microarchitecture*

fetch phase

execute phase

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

Jump?

nop

fetch phase

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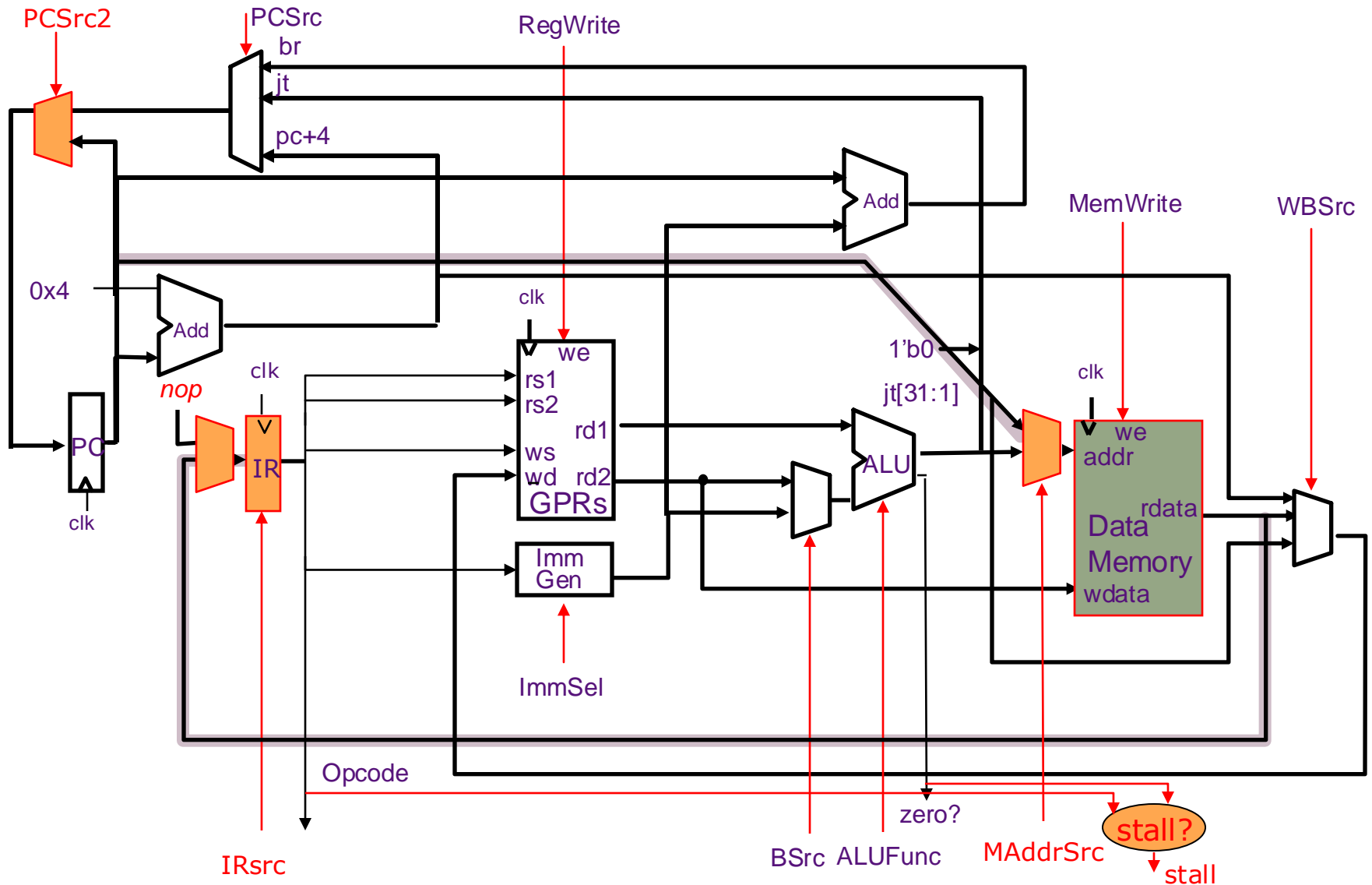
When IR contains a jump or taken branch
- *no "structural conflict" for the memory*
- *but we do not have the correct PC value in the PC*
- *memory cannot be used – Address Mux setting is irrelevant*
- *insert a nop in the IR*
- *insert the nextPC (branch-target) address in the PC*

Pipelined Princeton Microarchitecture

Hardwired Control Table

Opcode	Stall	Imm Sel	BSrc	ALU	MemW	WBSrc	RegWr	PC Src1	PC Src2	IRSrc	MAddr Src
ALU	no	*	Reg	funct	no	ALU	yes	pc+4	npc	mem	pc
ALUi	no	SXT(imm [11:0])	Imm	funct	no	ALU	yes	pc+4	npc	mem	pc
LW	yes	SXT(imm [11:0])	Imm	+	no	Mem	yes	pc+4	pc	nop	ALU
SW	yes	SXT(imm [11:0])	Imm	+	yes	*	no	pc+4	pc	nop	ALU
BRtaken	yes	ImmB	Reg	funct	no	*	no	br	npc	nop	*
BRnotTaken	no	ImmB	Reg	funct	no	*	no	pc+4	npc	mem	pc
JALR	yes	ImmI	Imm	+	no	pc+4	yes	jt	npc	nop	*
JAL	yes	ImmU	Imm	*	no	pc+4	yes	br	npc	nop	*
NOP	no	*	*	*	no	*	no	pc+4	npc	mem	pc

BSrc = Reg / Imm; IRSrc = nop/mem; MAddSrc = pc/ALU; WBSrc = ALU / Mem / pc+4
PCSrc1 = pc+4 / br / rind; PCSrc2 = npc/pc;

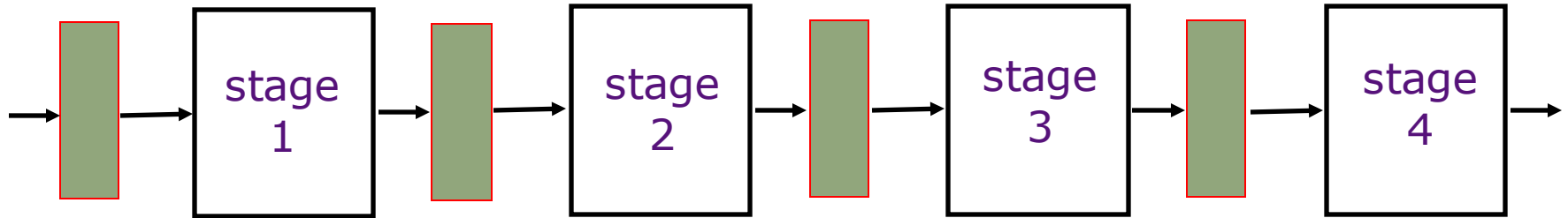# Pipelined Princeton Architecture

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

fetch phase   *decode & Reg-fetch phase*   *execute phase*   *memory phase*   *write-back phase*
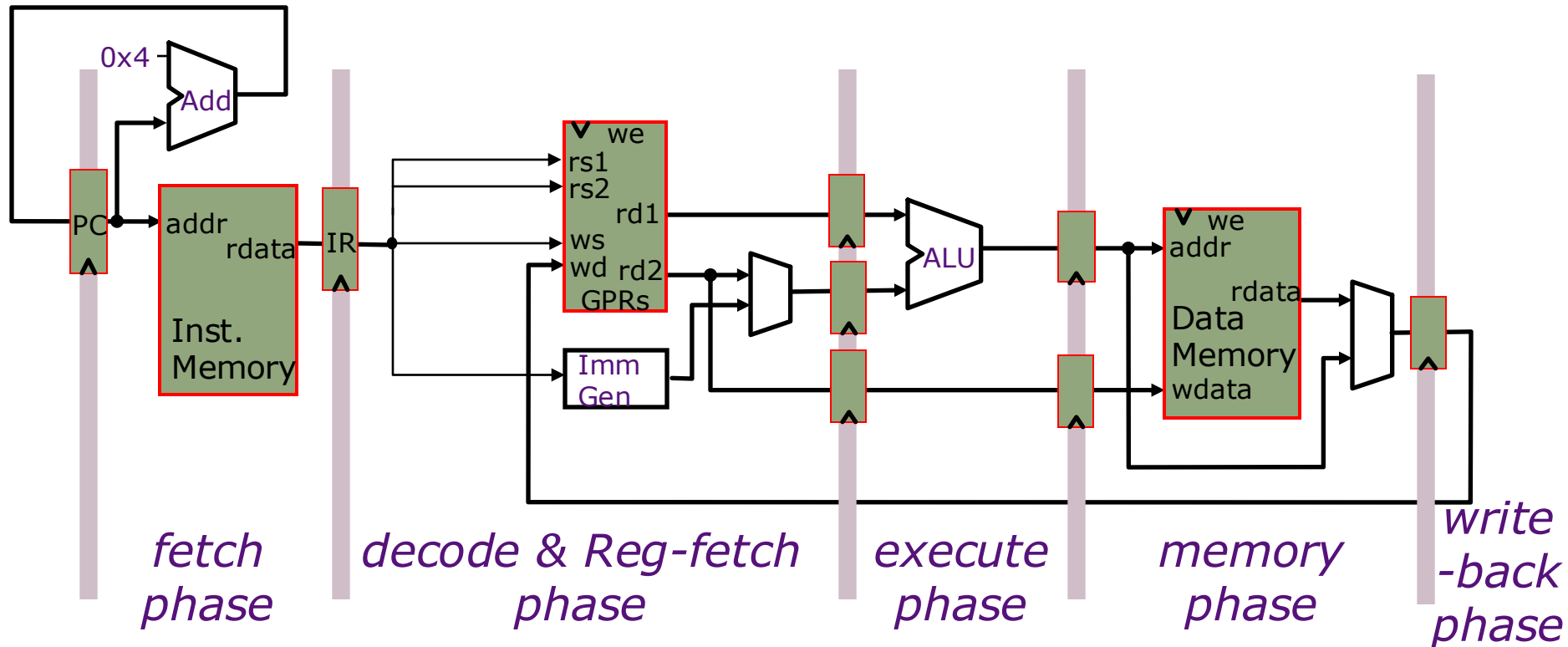
Clock period can be reduced by dividing the execution of an instruction into multiple cycles

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

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

$$t_{IM} \quad = 10 \text{ units}$$
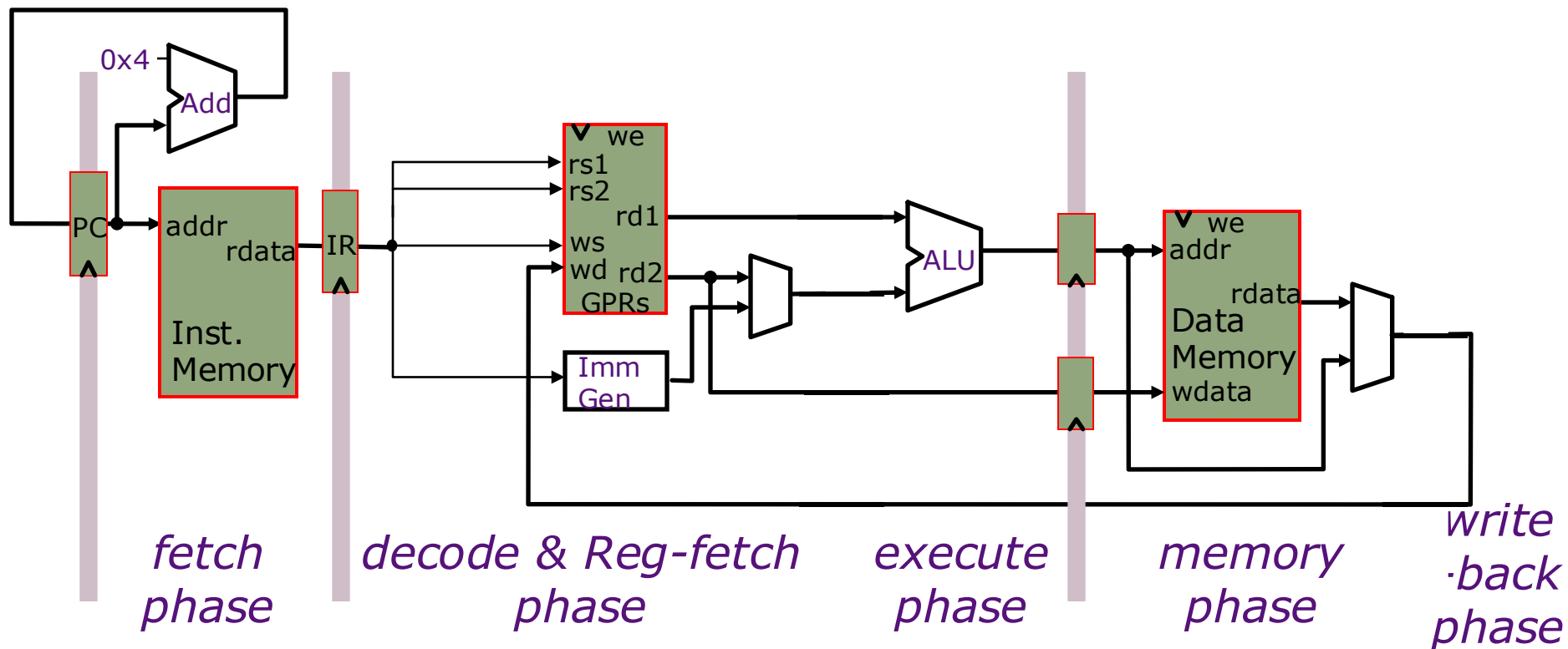$$t_{DM} \quad = 10 \text{ units}$$
$$t_{ALU} \quad = 5 \text{ units}$$
$$t_{RF} \quad = 1 \text{ unit}$$
$$t_{RW} \quad = 1 \text{ unit}$$

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

Alternative Pipelining

fetch phase
decode & Reg-fetch phase
execute phase
memory phase
write-back phase

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

⇒ 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?
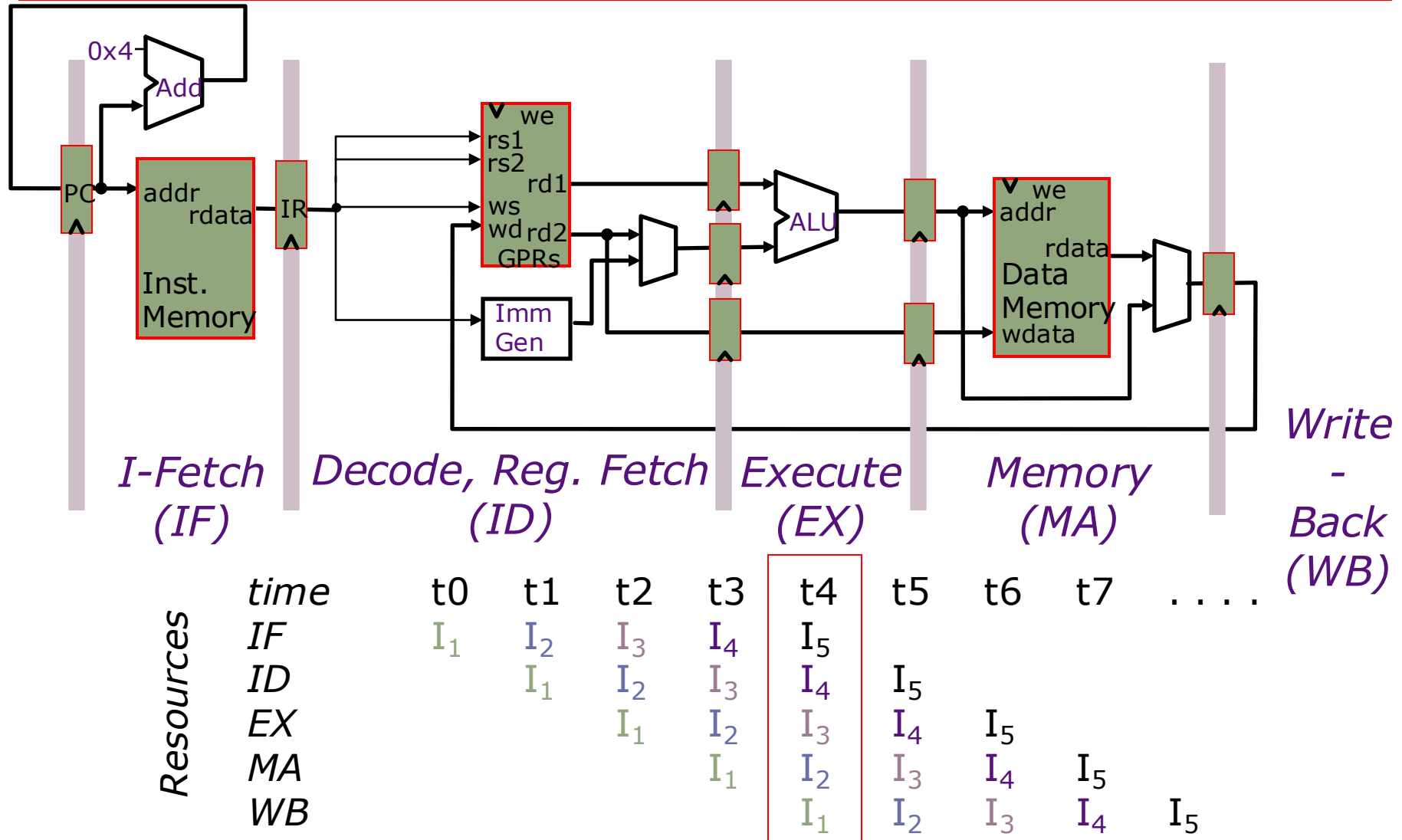
One can achieve higher speedup with more pipeline stages

5-Stage Pipelined Execution
Instruction Flow Diagram

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
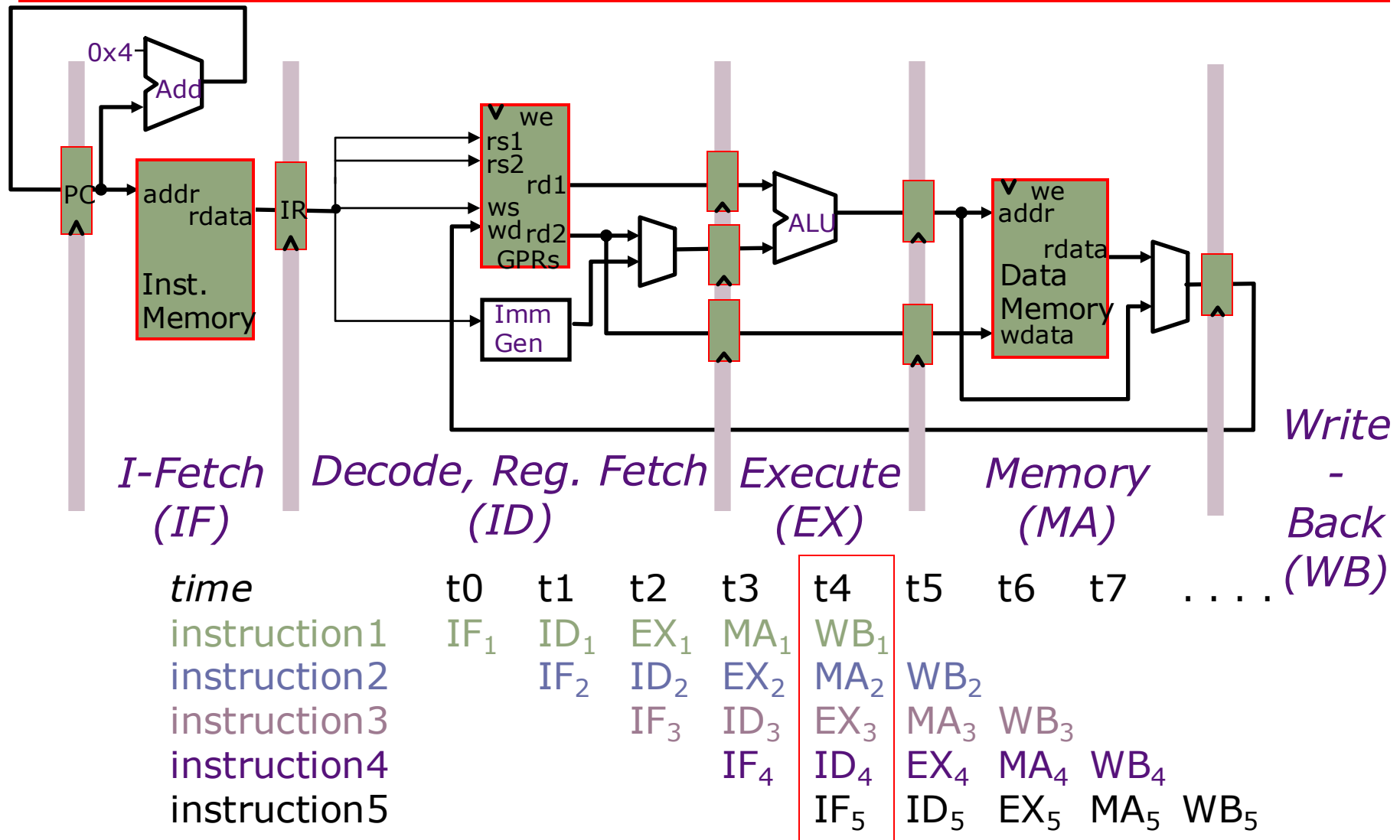
5-Stage Pipelined Execution
Resource Usage Diagram

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
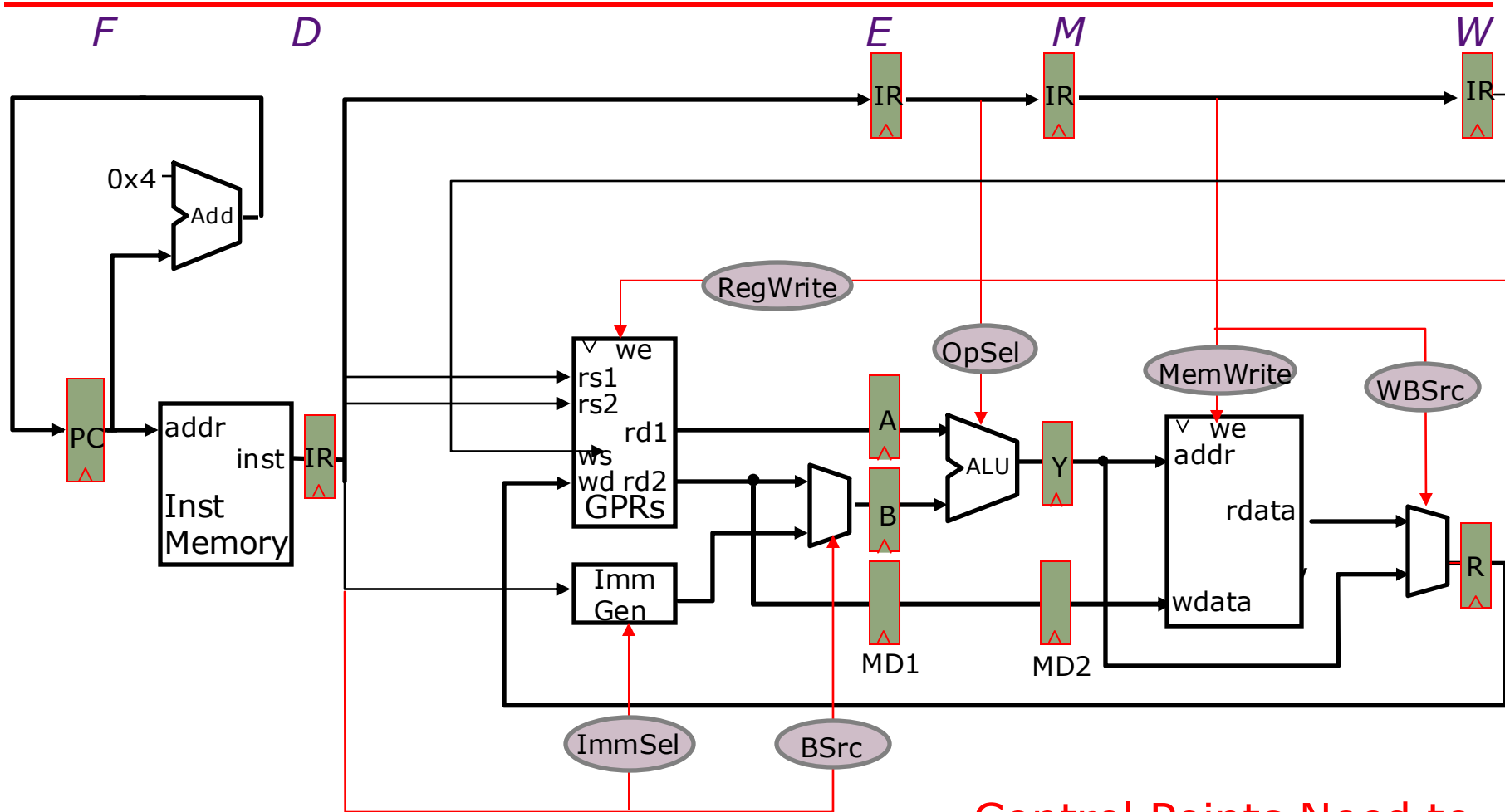## ALU Instructions
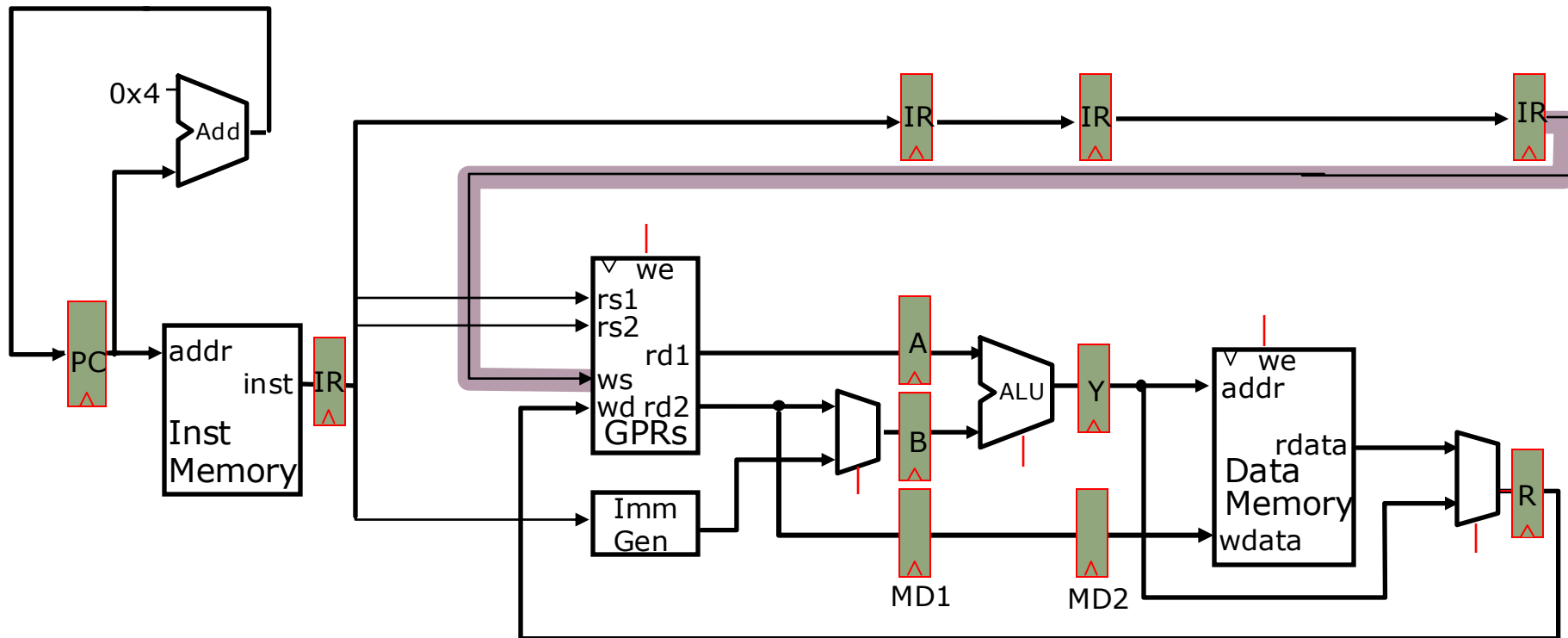
Not quite correct!

We need an Instruction Reg (IR) for each stage

Pipelined RISC-V Datapath
## *without jumps*

What else is needed?

Control Points Need to Be Connected

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

r4 ← r1 …
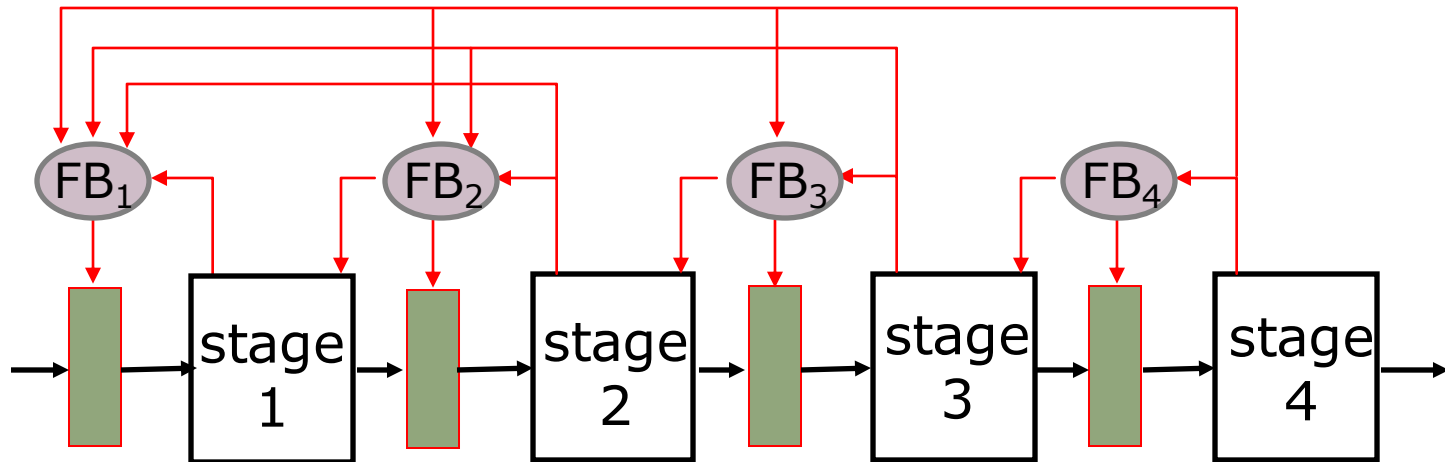
r1 ← …
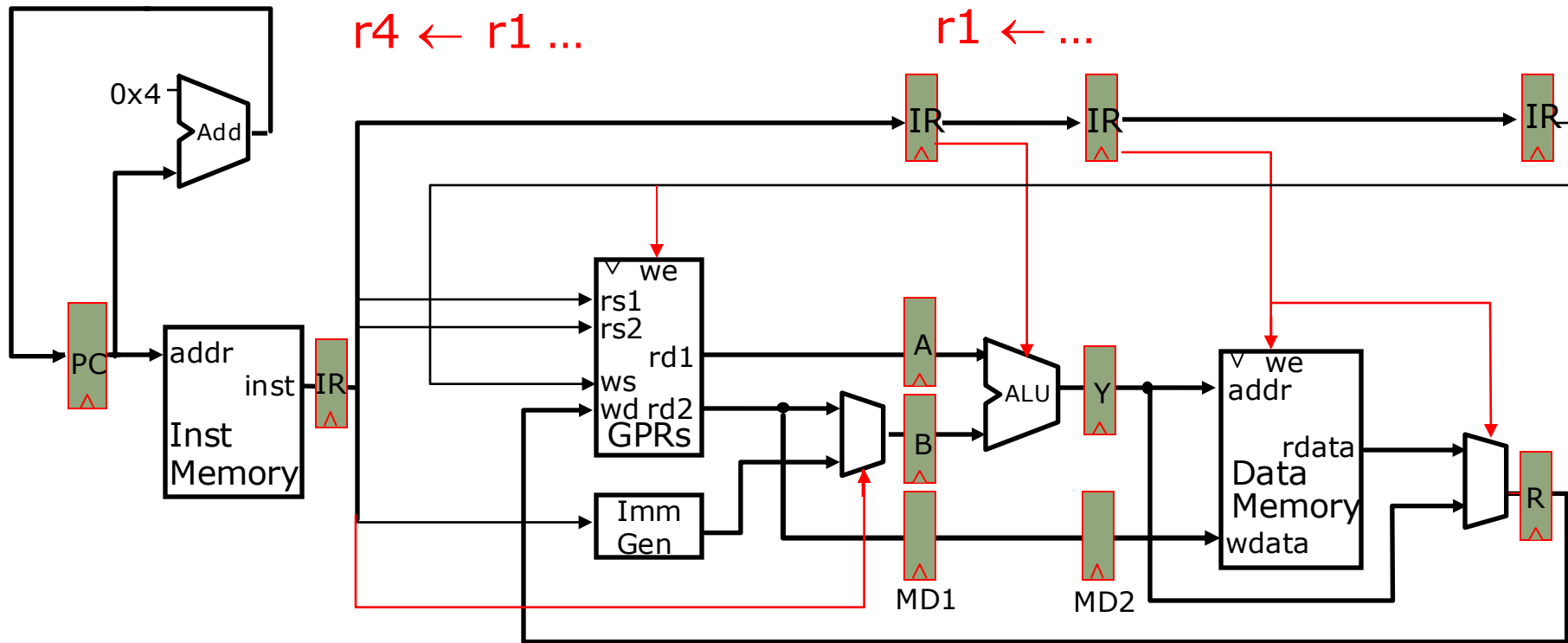
...
r1 ← r0 + 10
r4 ← r1 + 17
...

r1 is stale. Oops!

Resolving Data Hazards

Use strategy from Princeton Pipeline:

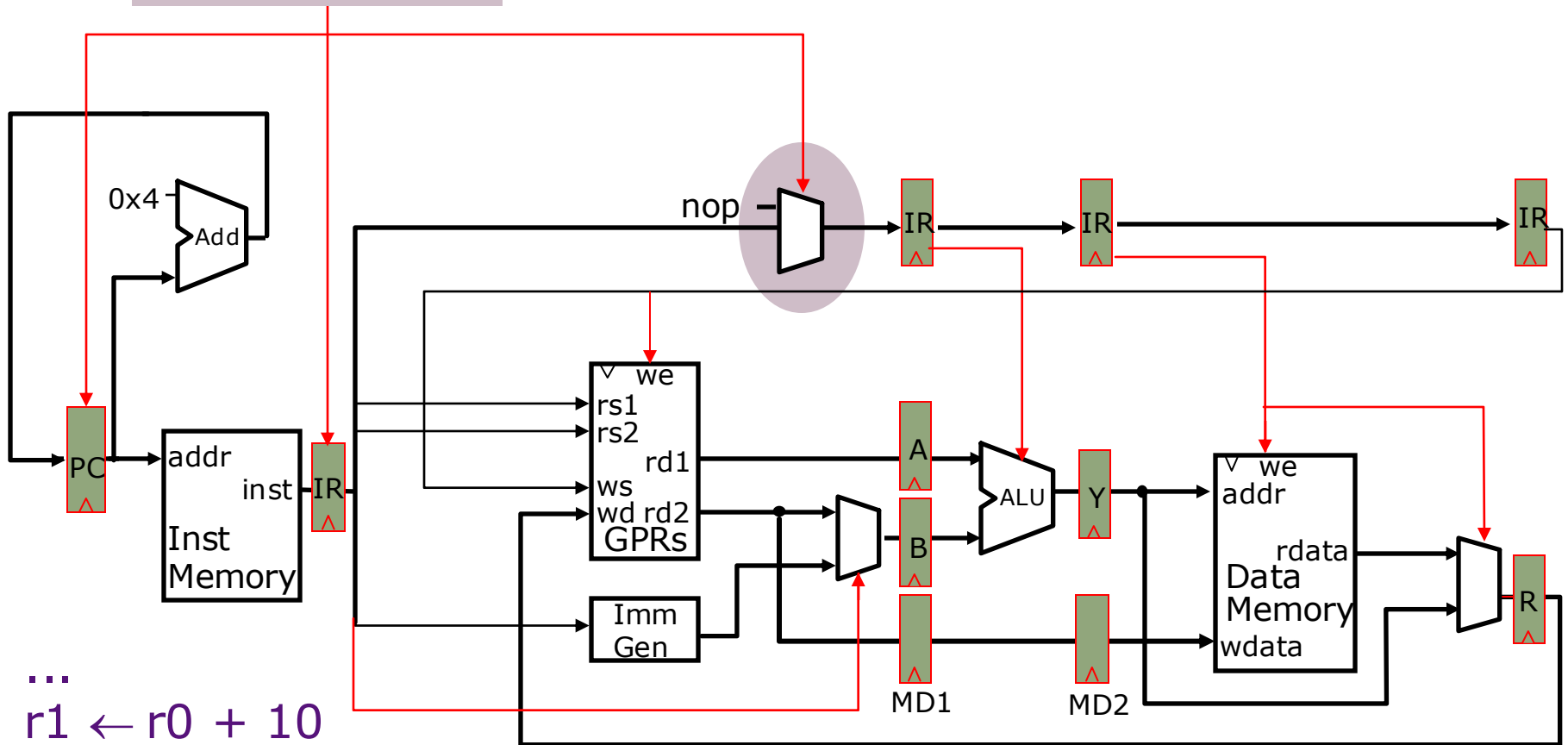*Wait for the result to be available by freezing earlier pipeline stages → stall*

Feedback to Resolve Hazards

- Later stages provide dependence information to earlier stages which can *stall 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
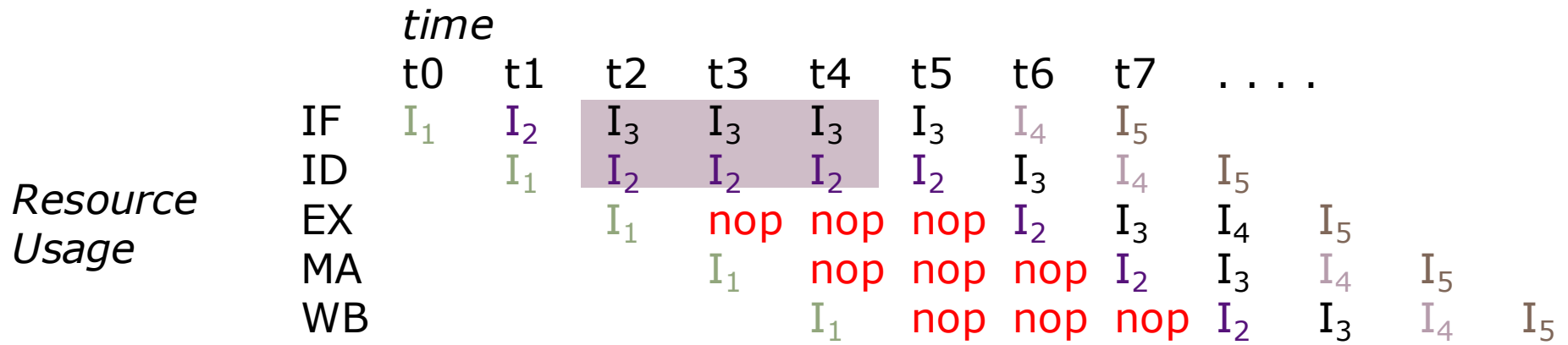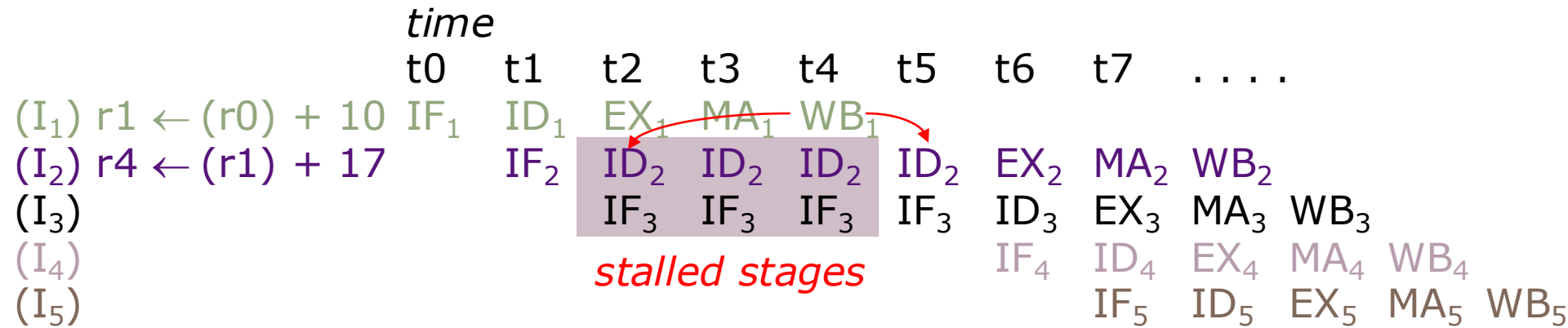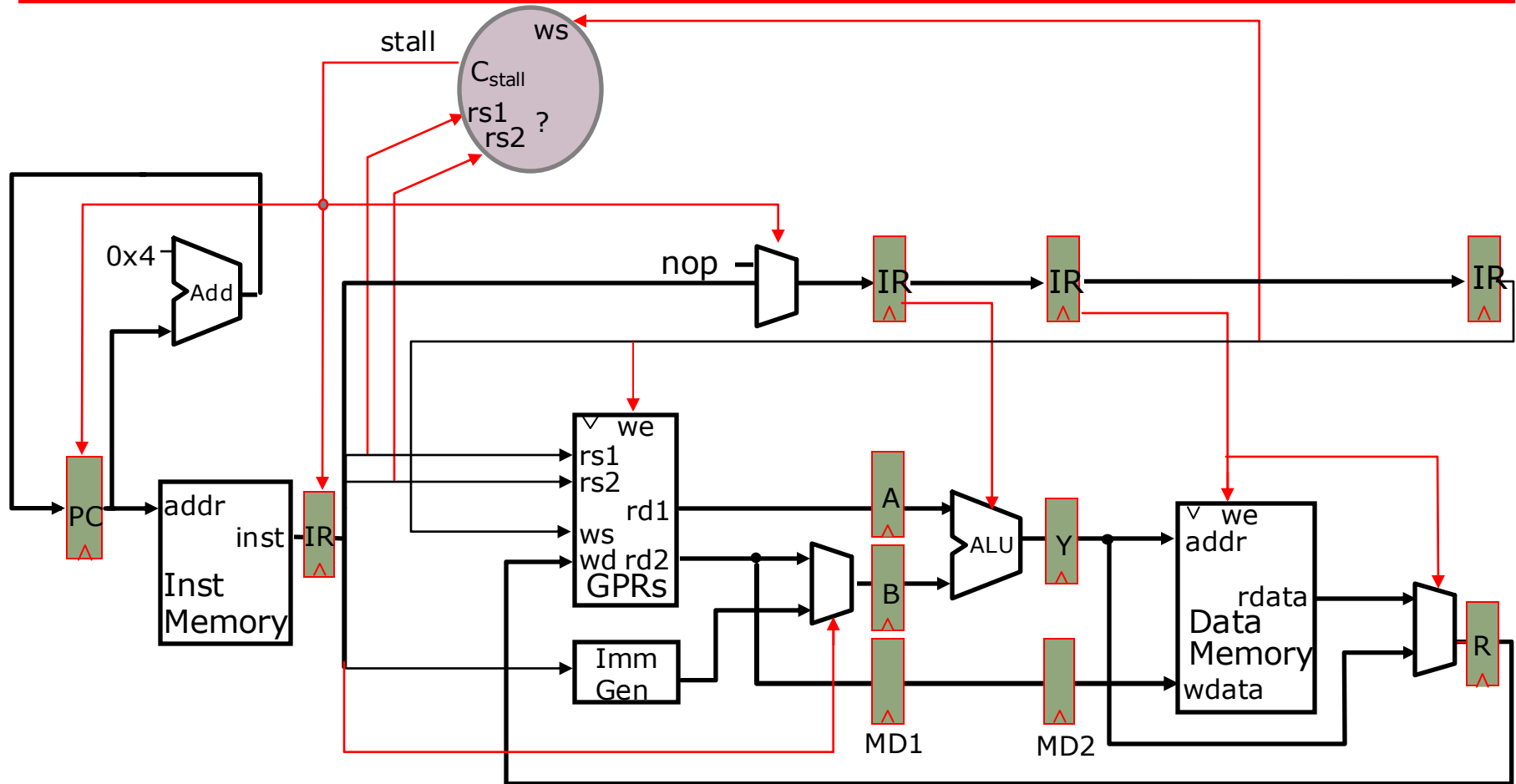

Stall Condition

0x4
Add

nop

IR   IR   IR

PC   addr   inst   IR

Inst
Memory

we
rs1
rs2   rd1   A
ws   ALU   Y
wd rd2   B
GPRs

Imm
Gen

we
addr
Data
Memory
rdata
wdata   R

MD1   MD2

...
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

time

Resource Usage

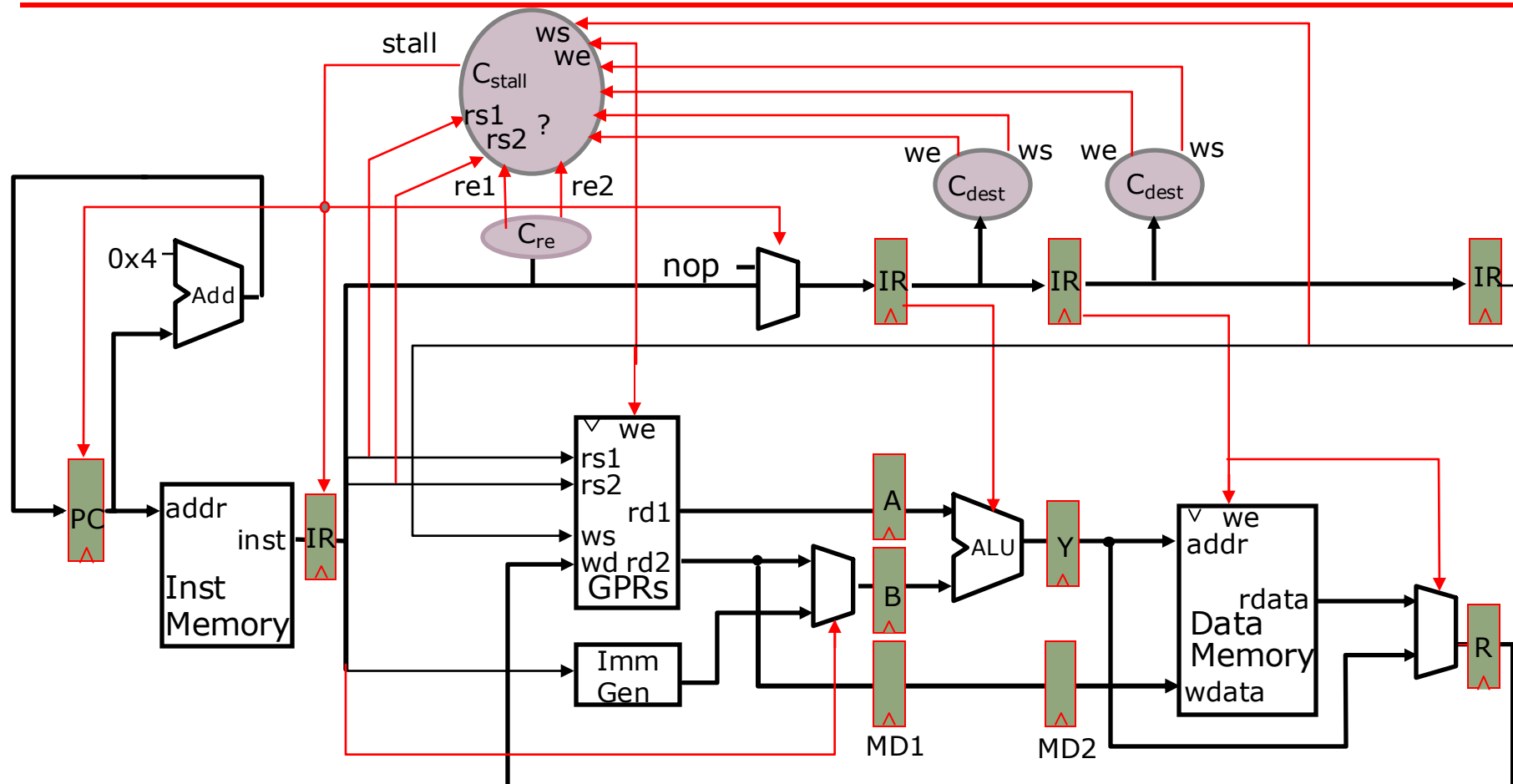|	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_3	I_3	I_3	I_4	I_5	
ID		I_1	I_2	I_2	I_2	I_2	I_3	I_4	I_5
EX			I_1	nop	nop	nop	I_2	I_3	I_4 I_5
MA				I_1	nop	nop	nop	I_2	I_3 I_4 I_5
WB					I_1	nop	nop	nop	I_2 I_3 I_4 I_5

nop ⇒ *pipeline bubble*

Stall Control Logic

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

Stall Control Logic
ignoring jumps & branches

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

not every instruction writes a register $\Rightarrow$ we

not every instruction reads a register $\Rightarrow$ re

Thank you!