

EE 660: Computer Architecture Pipelining Review

Yao Zheng

Department of Electrical Engineering

University of Hawai'i at Mānoa



UNIVERSITY
of HAWAI'I®
MĀNOA

Based on the slides of Prof. David Wentzlaff

Agenda

- Microcoded Microarchitectures
- Pipeline Review
 - Pipelining Basics
 - Structural Hazards
 - Data Hazards
 - Control Hazards

Agenda

- **Microcoded Microarchitectures**
- Pipeline Review
 - Pipelining Basics
 - Structural Hazards
 - Data Hazards
 - Control Hazards

What Happens When the Processor is Too Large?

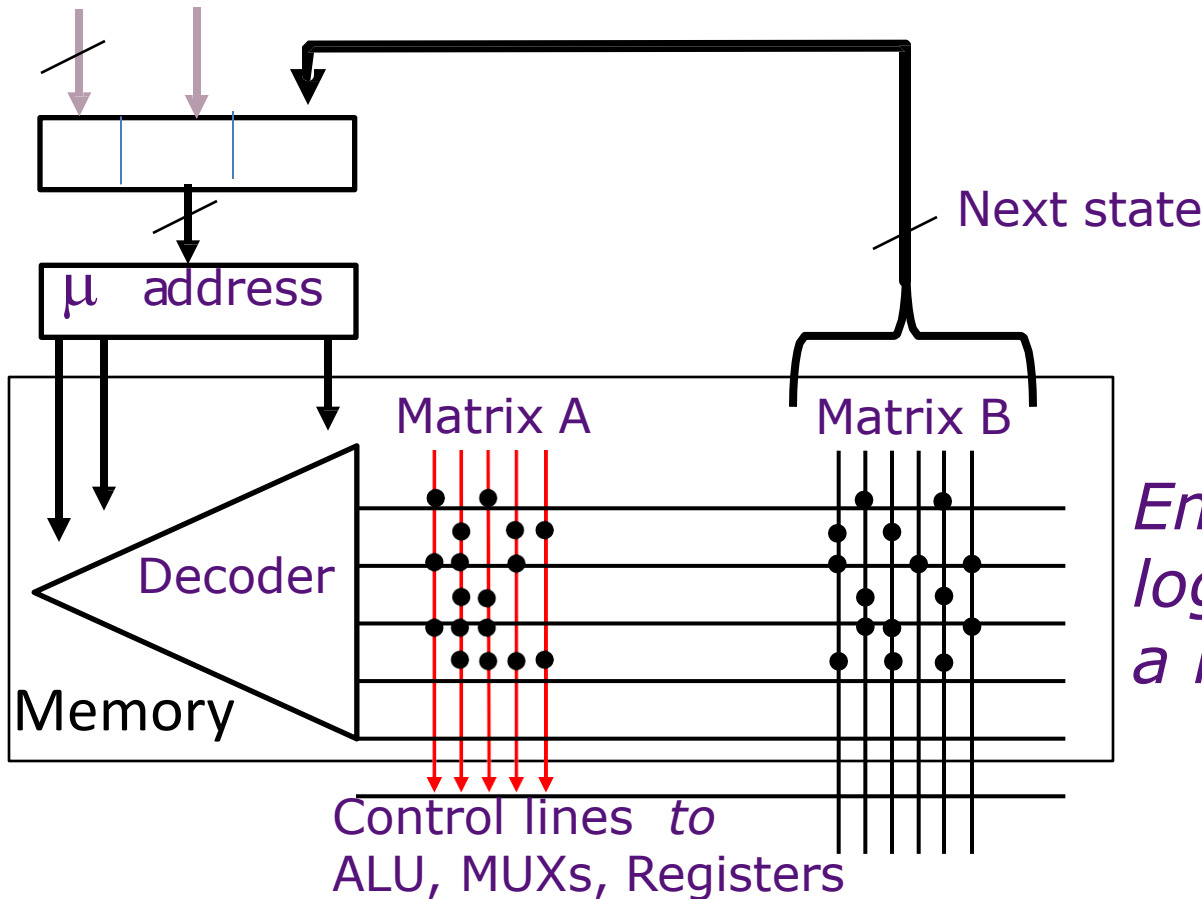
What Happens When the Processor is Too Large?

- Time Multiplex Resources!

Microcontrol Unit *Maurice Wilkes, 1954*

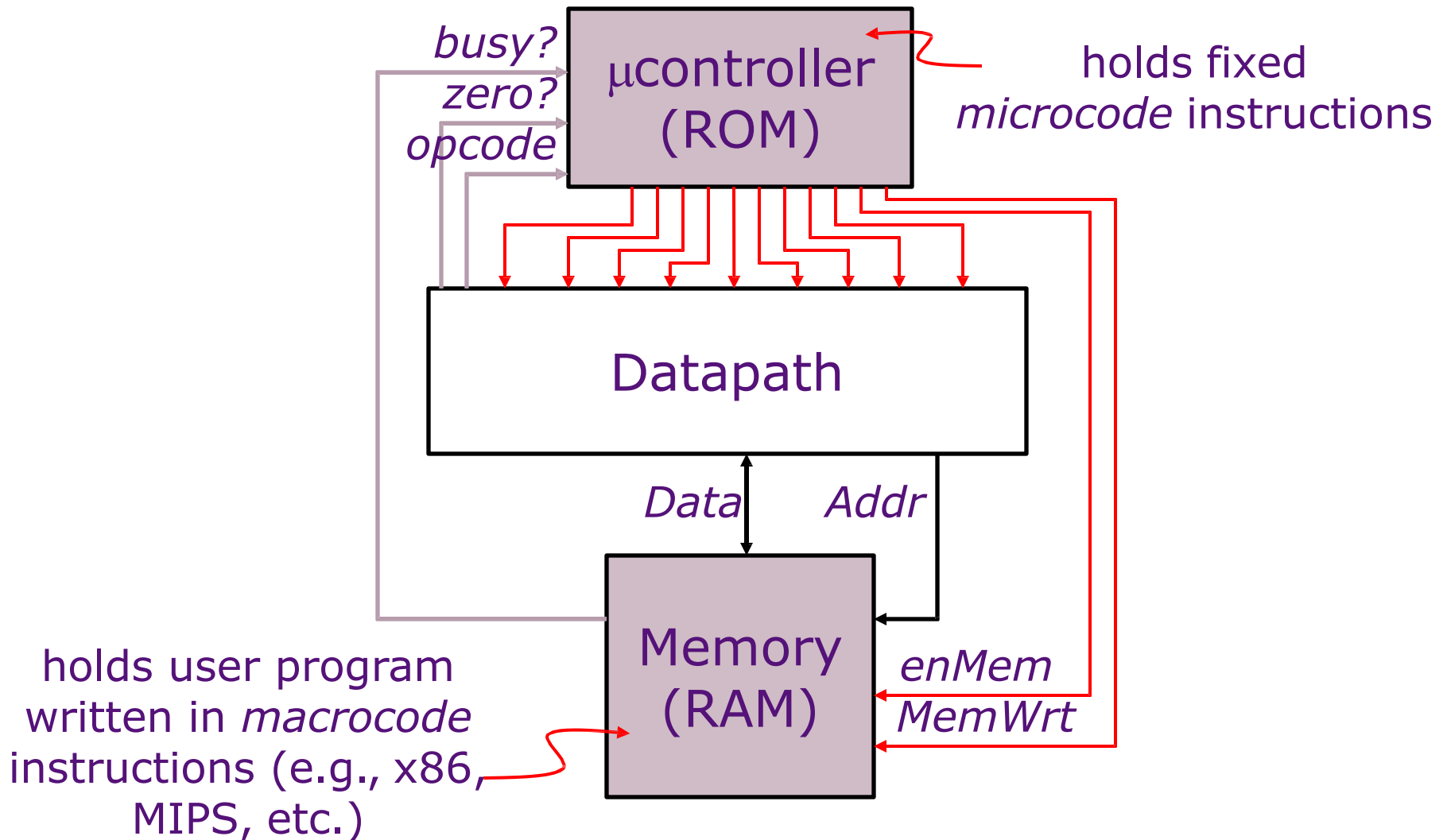
op code conditional
code flip-flop

*First used in EDSAC-2,
completed 1958*

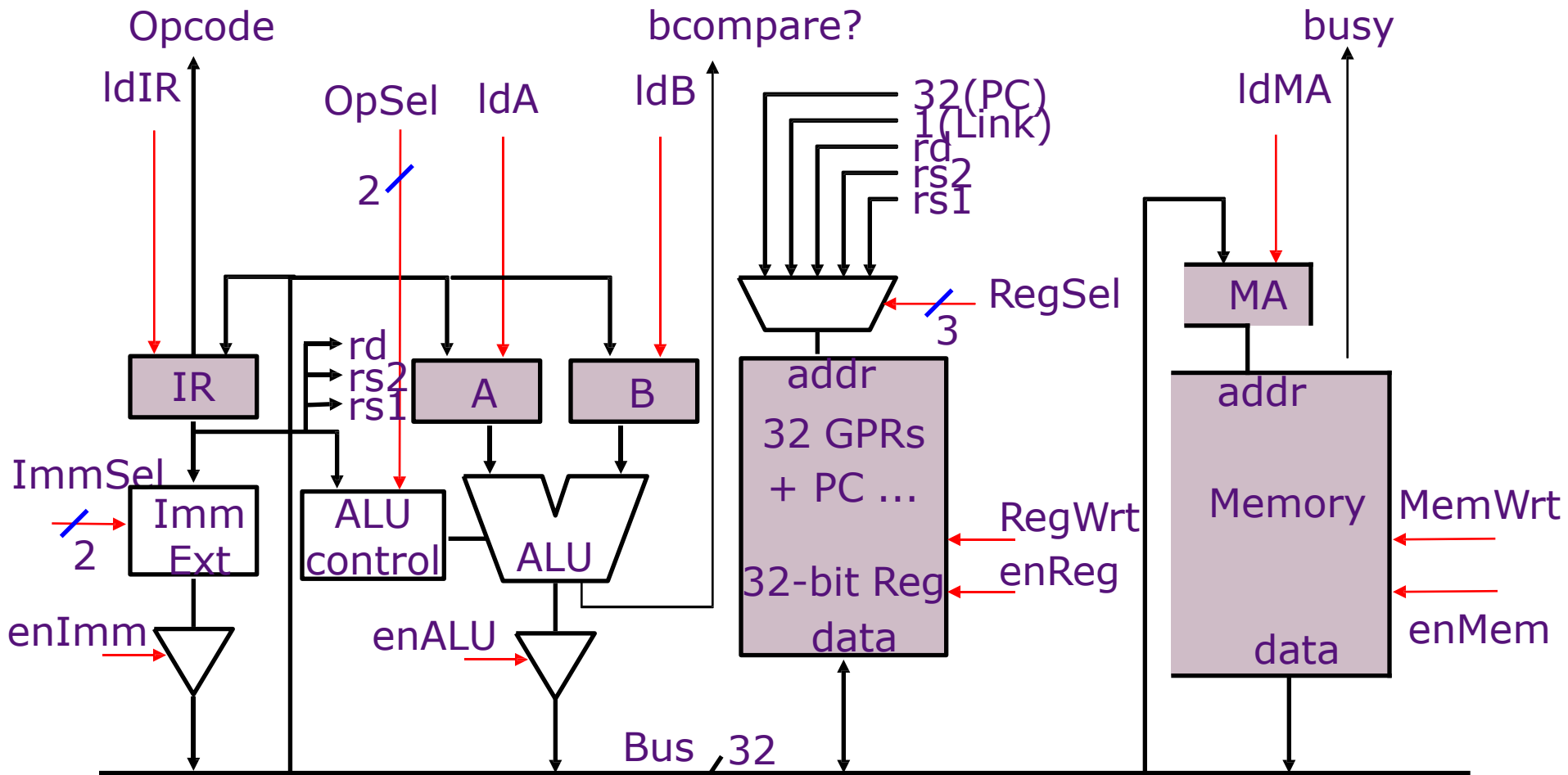


*Embed the control
logic state table in
a memory array*

Microcoded Microarchitecture



A Bus-based Datapath for RISC

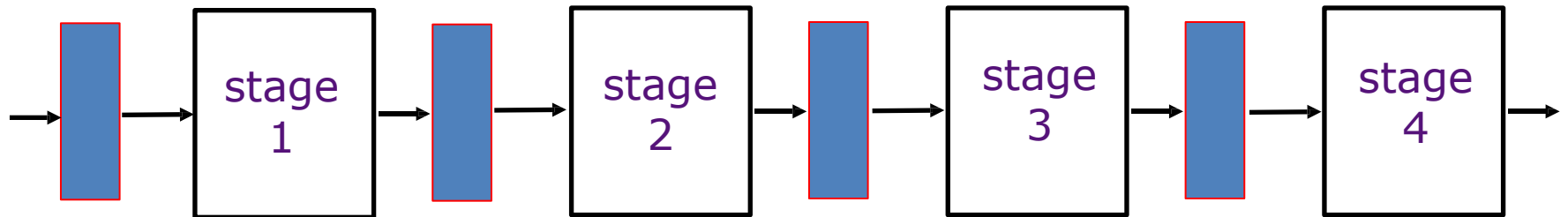


Microinstruction: register to register transfer (17 control signals)

Agenda

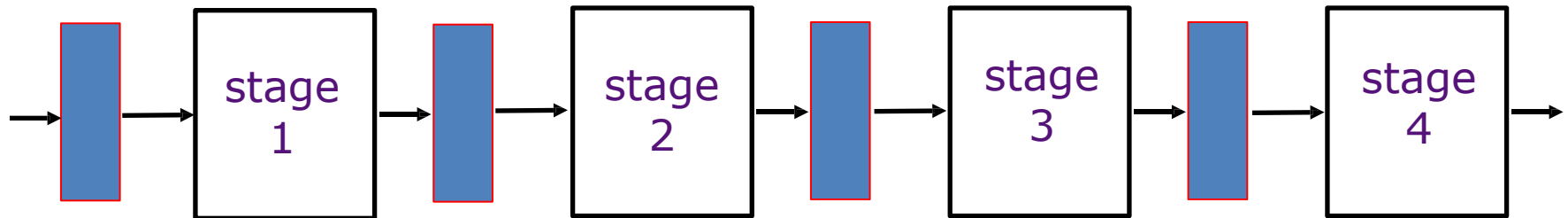
- Microcoded Microarchitectures
- Pipeline Review
 - Pipelining Basics
 - Structural Hazards
 - Data Hazards
 - Control Hazards

An Ideal Pipeline



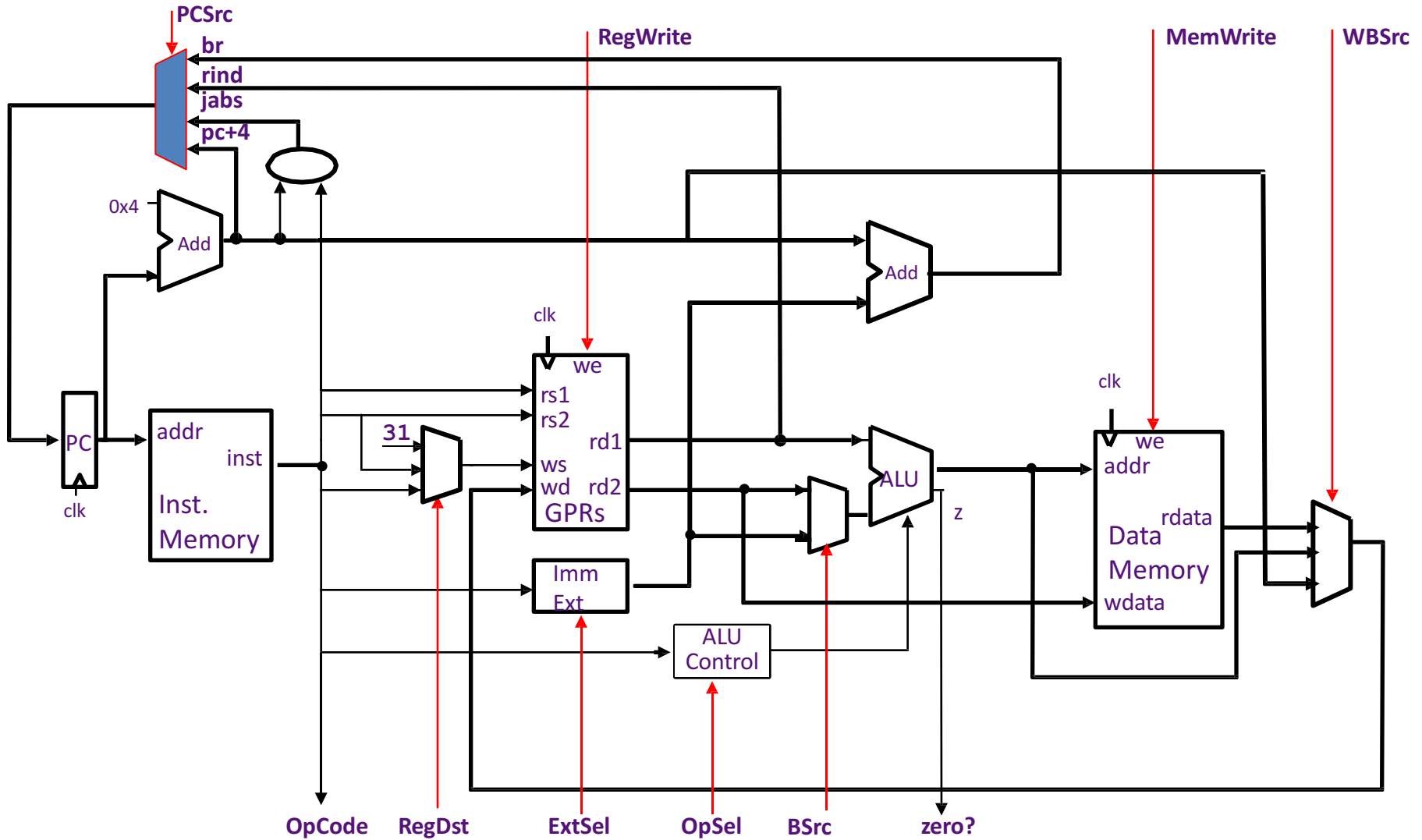
- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- Scheduling of a transaction entering the pipeline is not affected by the transactions in other stages

An Ideal Pipeline

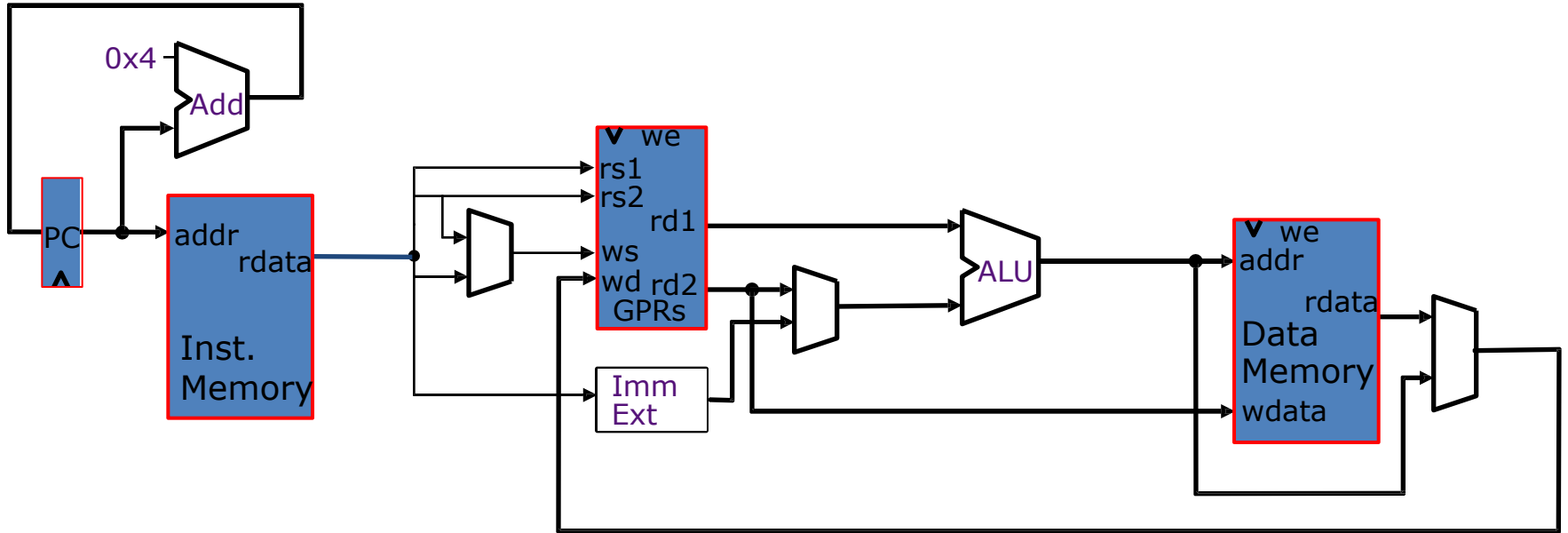


- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- Scheduling of a transaction entering the pipeline is not affected by the transactions in other stages
- These conditions generally hold for industry assembly lines, but instructions depend on each other causing various hazards

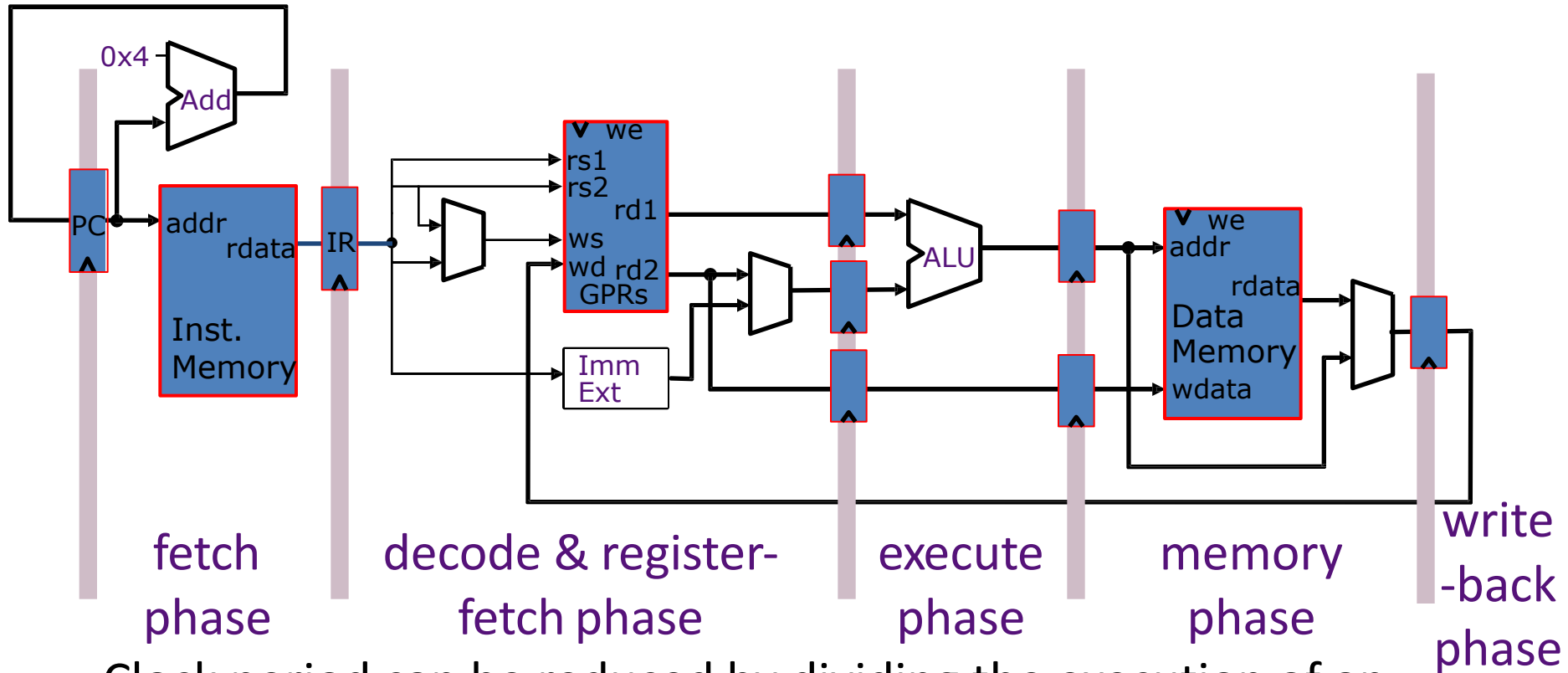
Unpipelined Datapath for MIPS



Simplified Unpipelined Datapath



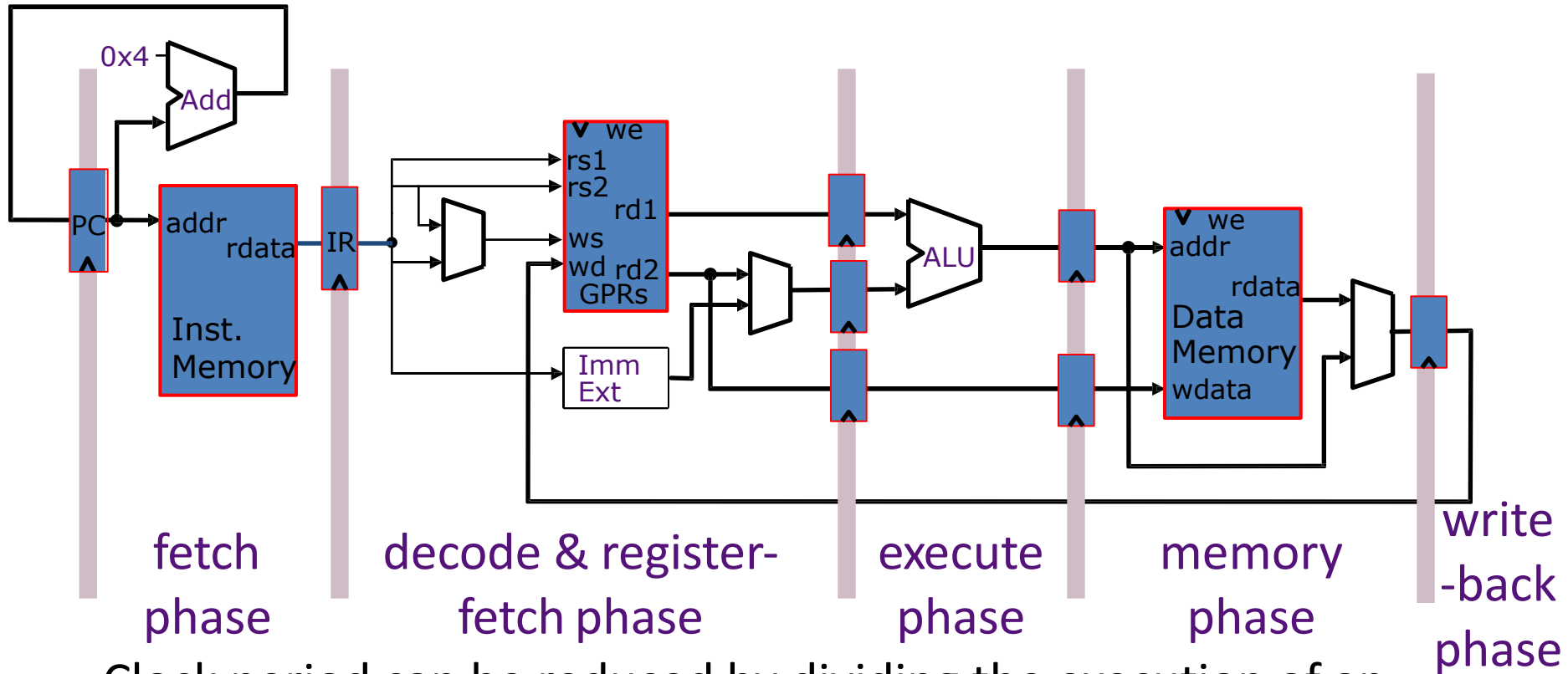
Pipelined Datapath



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

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

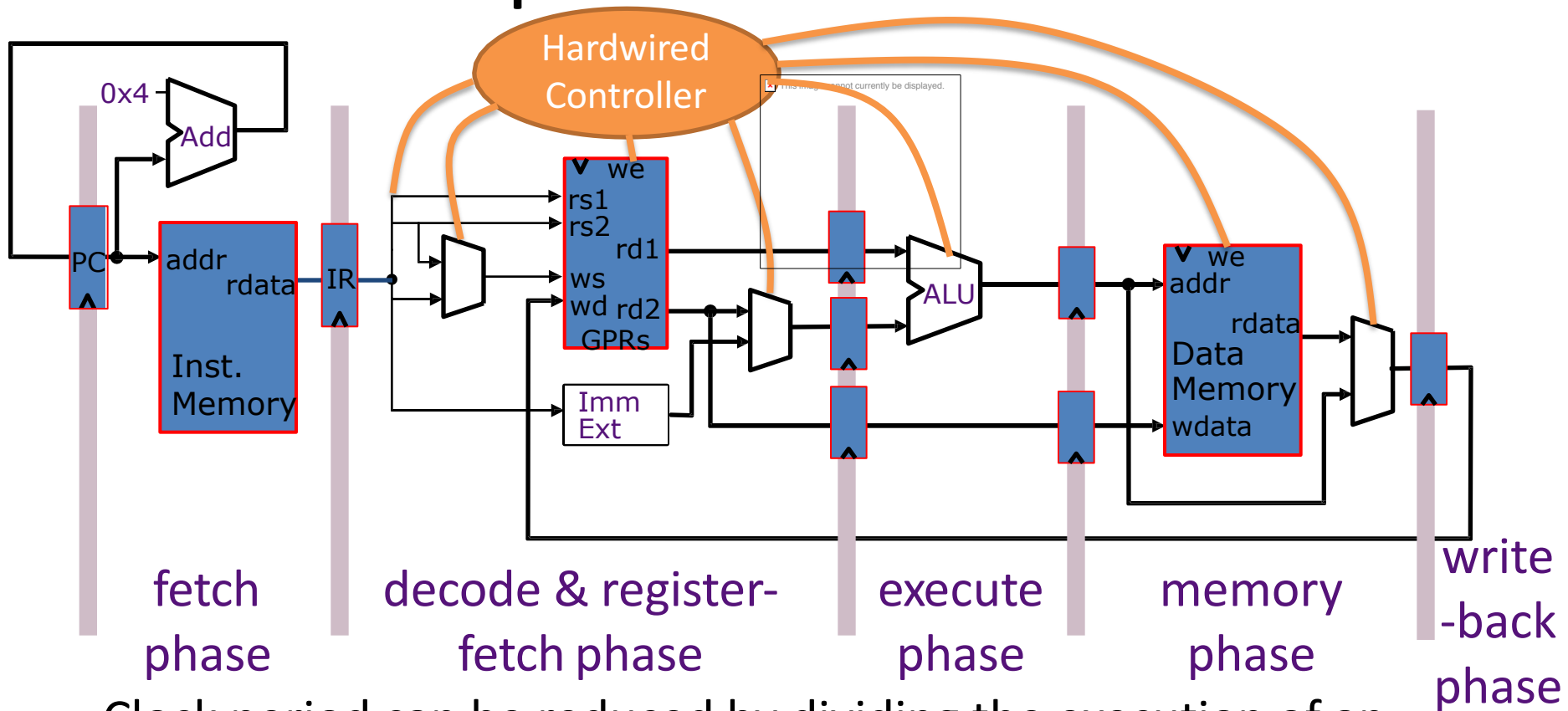
Pipelined Control



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

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

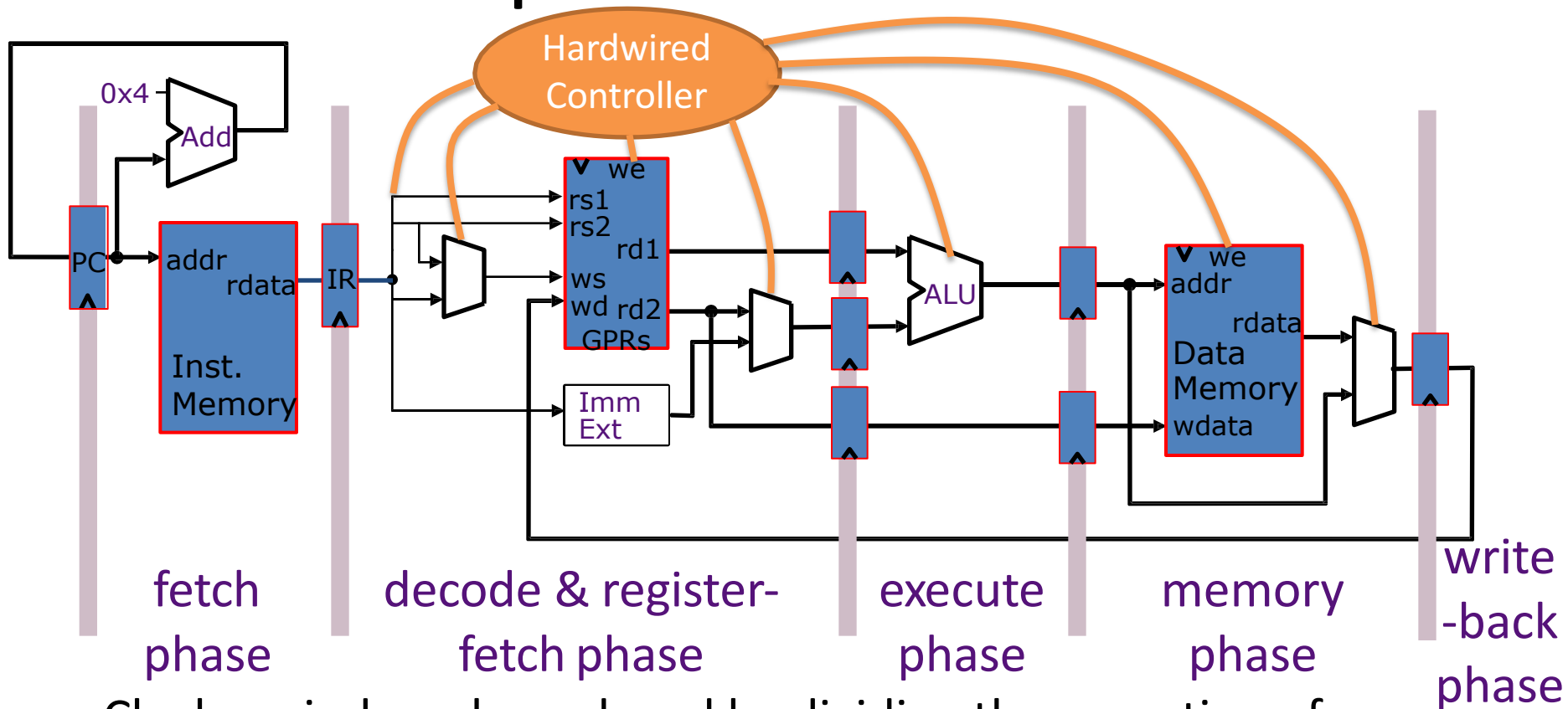
Pipelined Control



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

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

Pipelined Control

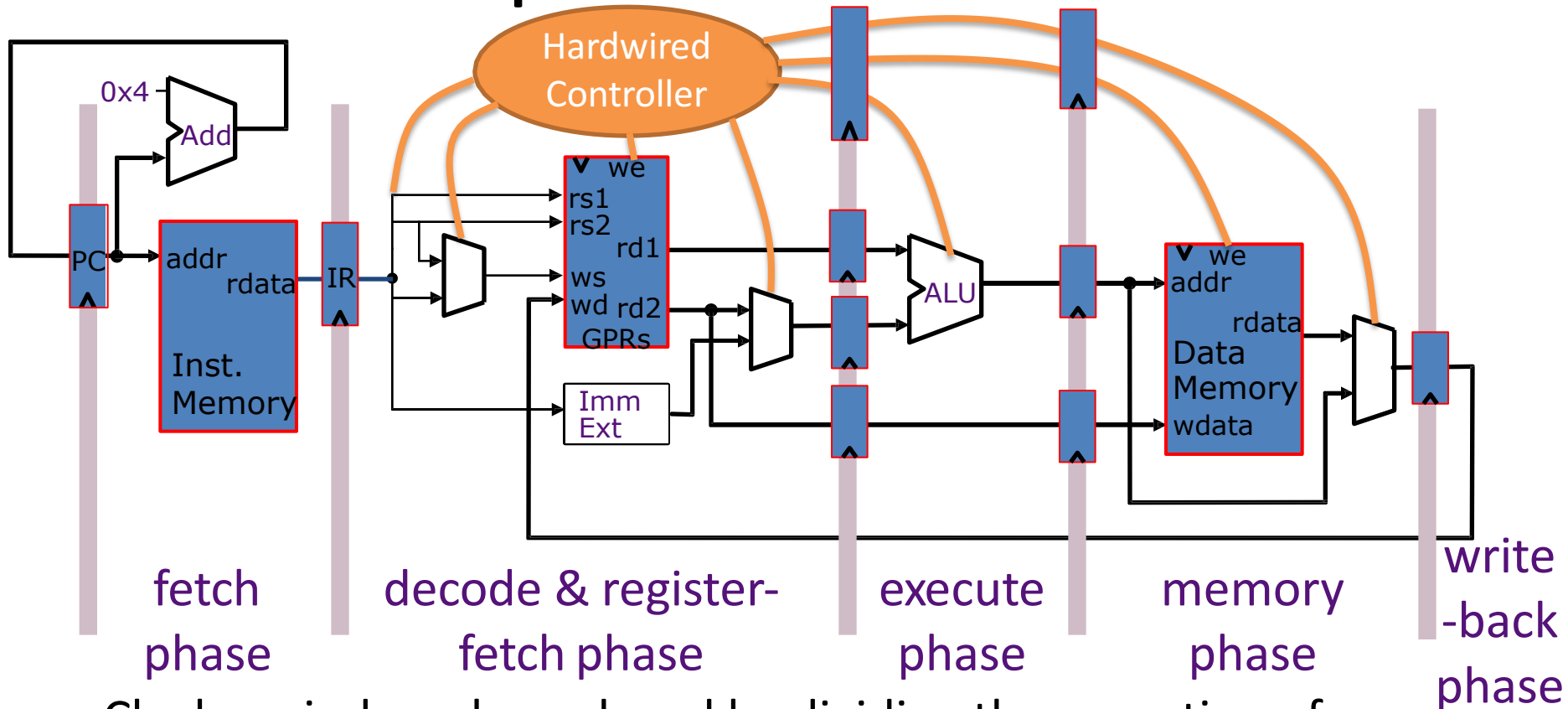


Clock period can be reduced by dividing the execution of an instruction into multiple cycles

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

However, CPI will increase unless instructions are pipelined

Pipelined Control



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

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

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
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

“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
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short
Multi-cycle, unpipelined control		

“Iron Law” of Processor Performance

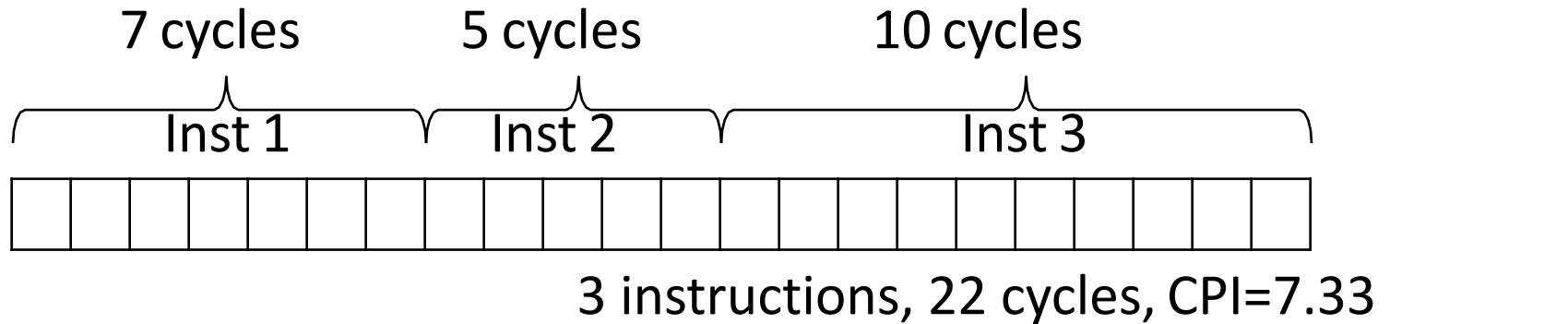
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- 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

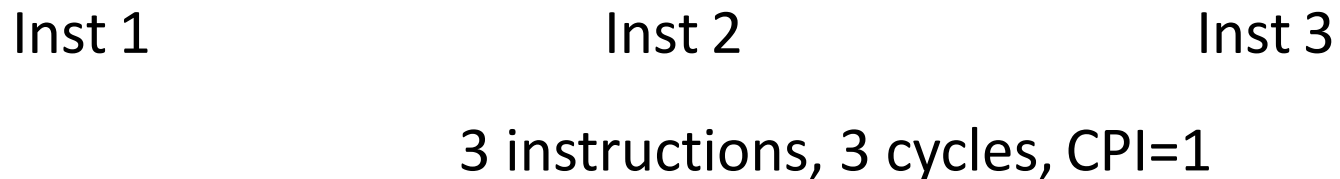
Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short
Multi-cycle, unpipelined control	>1	short

CPI Examples

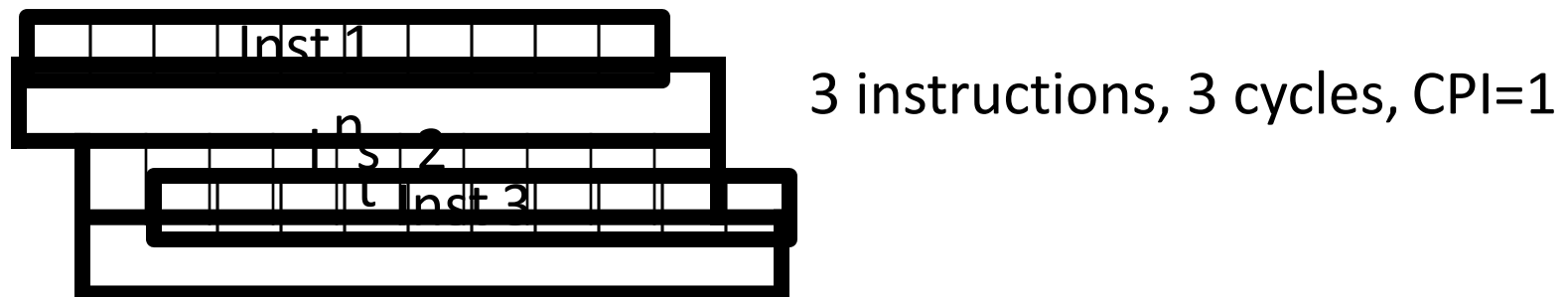
Microcoded machine



Unpipelined machine



Pipelined machine



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!)

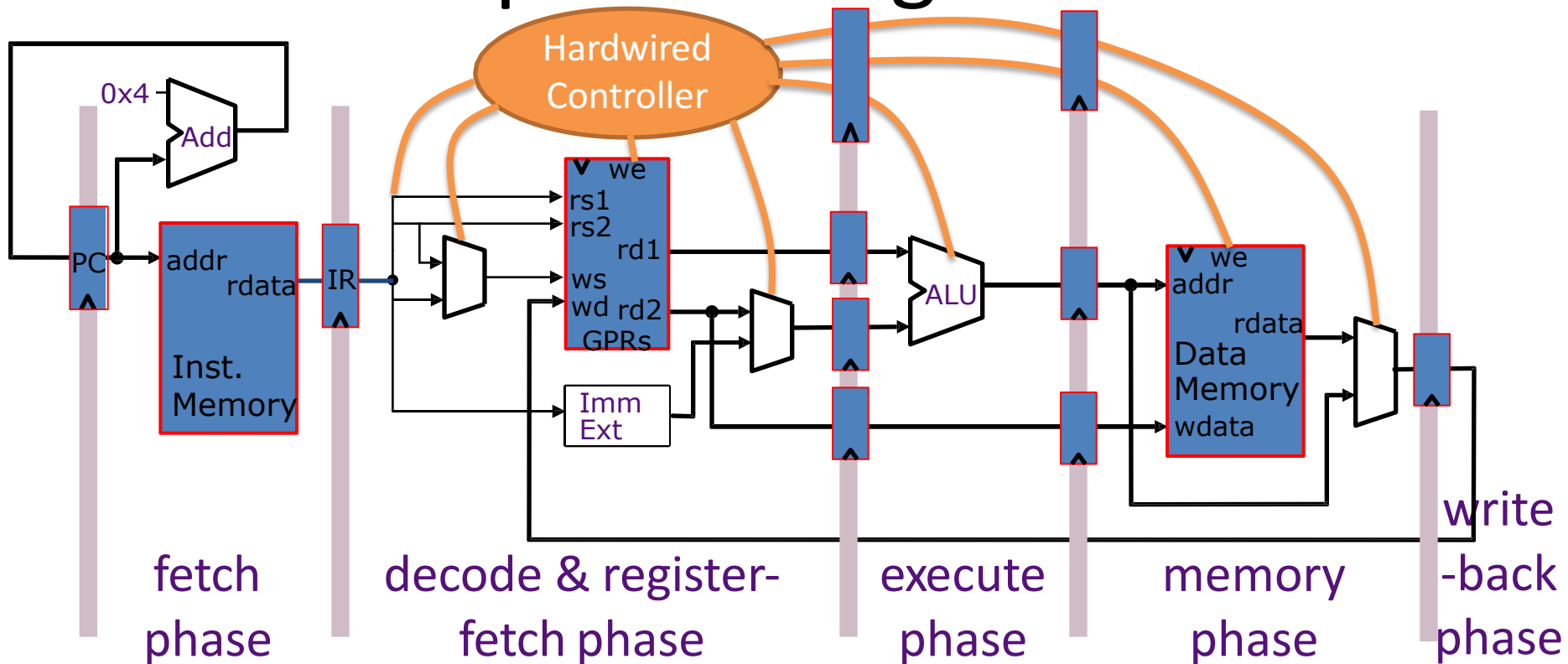
Thus, the following timing assumption is reasonable

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

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

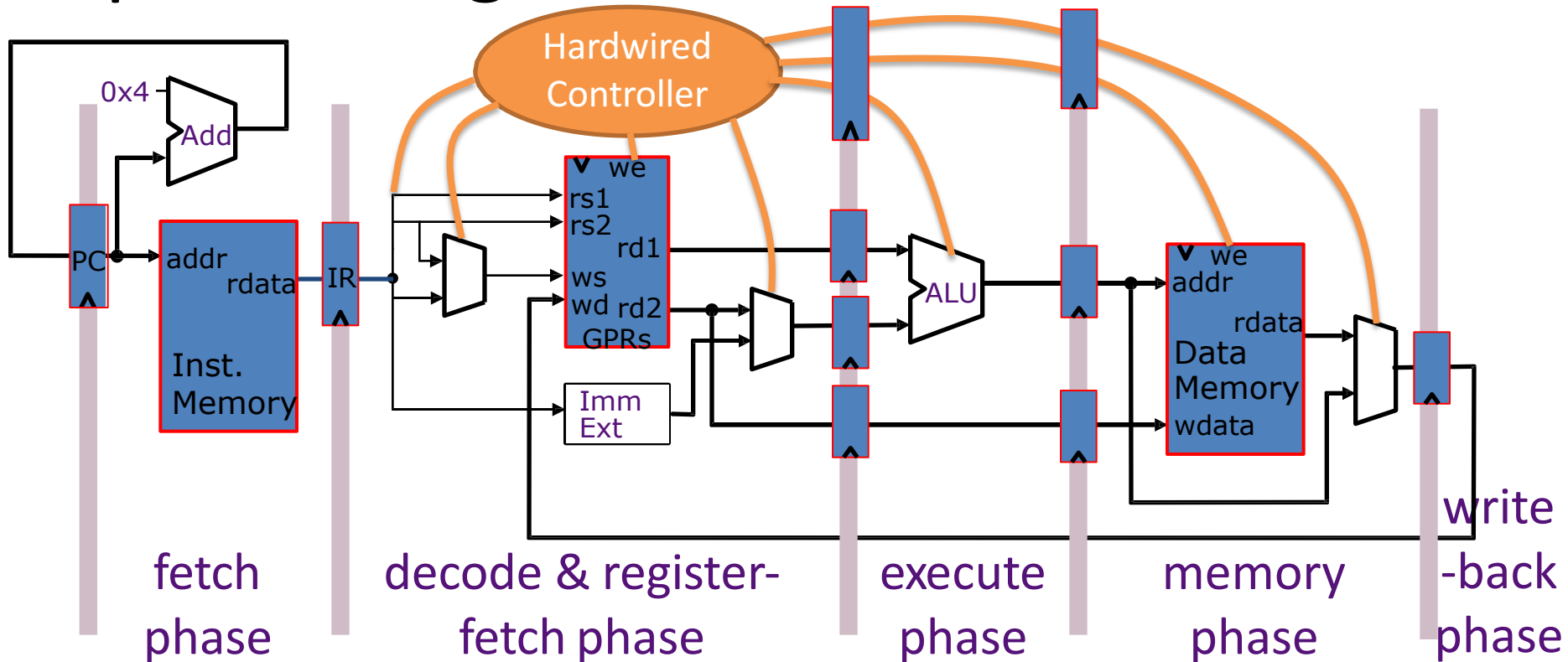
- *some commercial designs have over 30 pipeline stages to do an integer add!*

Pipeline Diagrams



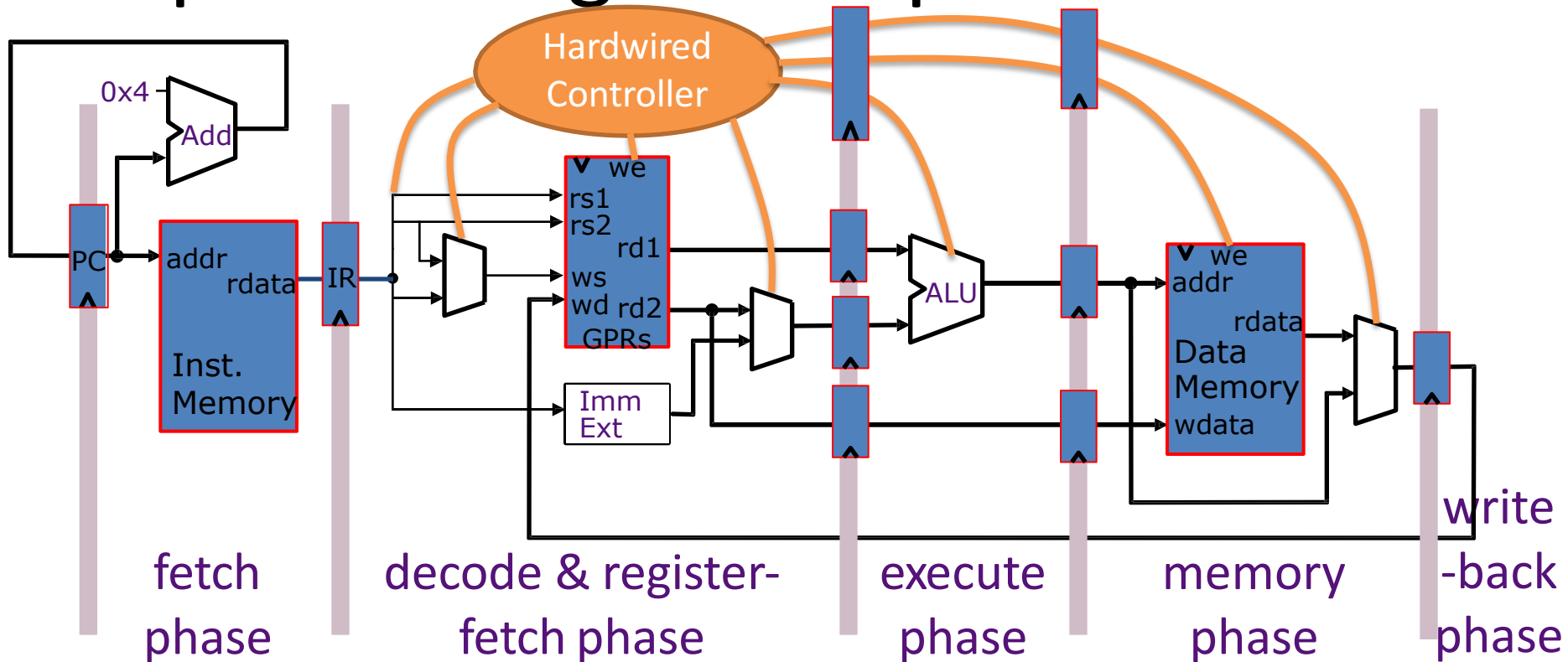
We need some way to show multiple simultaneous transactions in both space and time

Pipeline Diagrams: Transactions vs. Time



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

Pipeline Diagrams: Space vs. Time



Resource	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

Instructions Interact With Each Other in Pipeline

- **Structural Hazard:** An instruction in the pipeline needs a resource being used by another instruction in the pipeline
- **Data Hazard:** An instruction depends on a data value produced by an earlier instruction
- **Control Hazard:** Whether or not an instruction should be executed depends on a control decision made by an earlier instruction

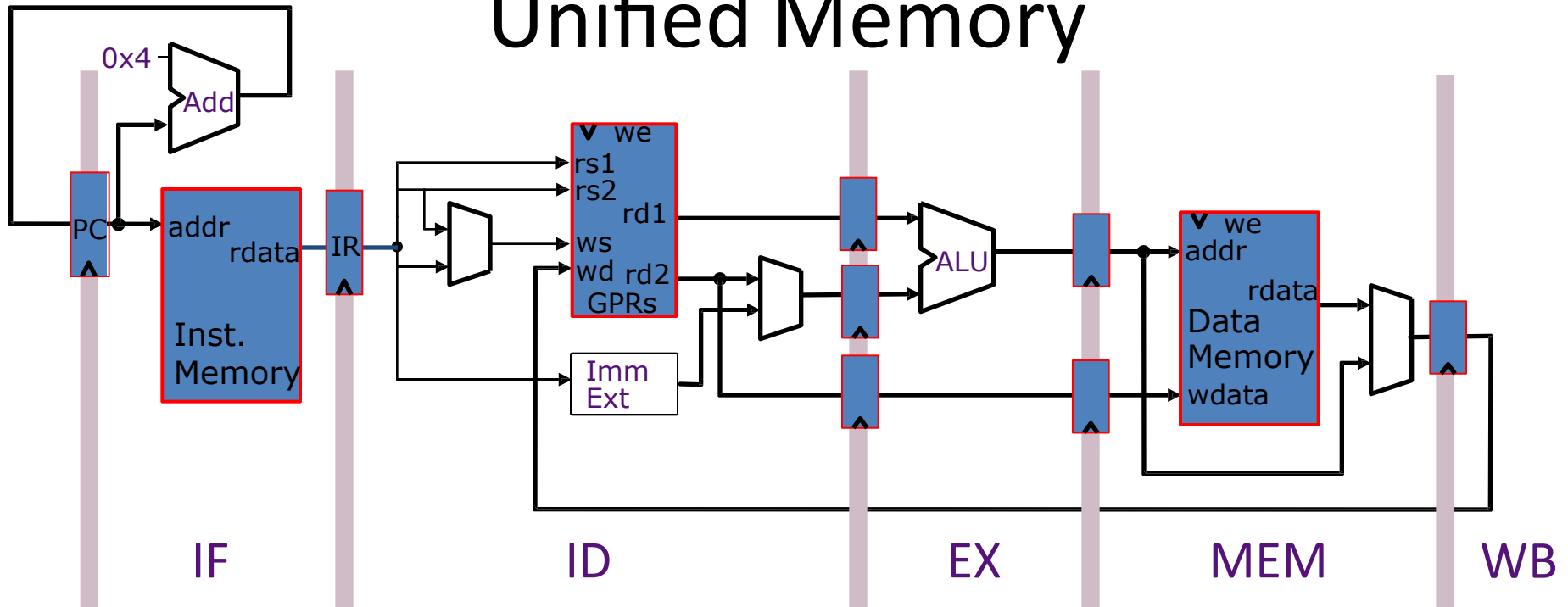
Agenda

- Microcoded Microarchitectures
- Pipeline Review
 - Pipelining Basics
 - Structural Hazards
 - Data Hazards
 - Control Hazards

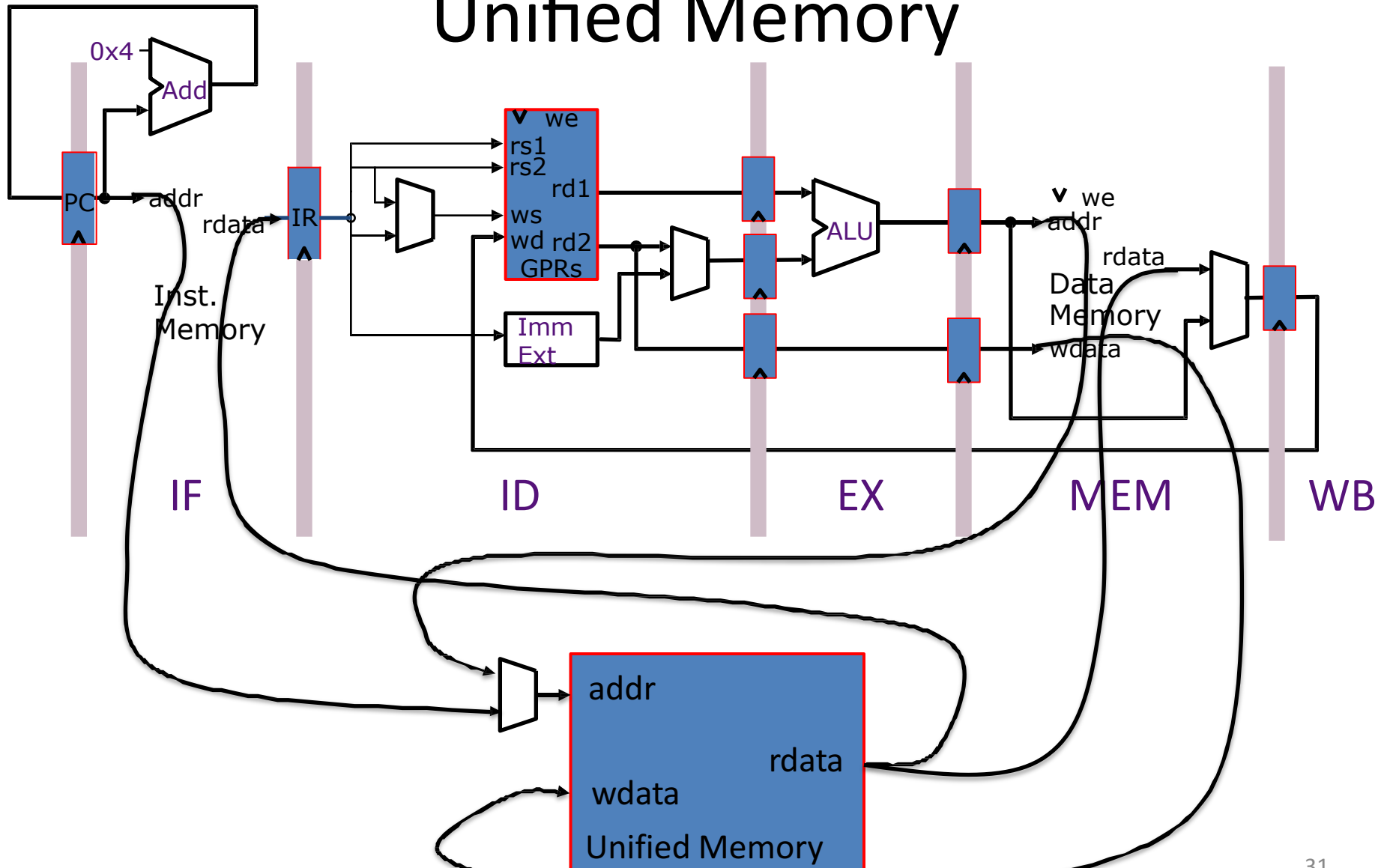
Overview of Structural Hazards

- Structural hazards occur when two instructions need the same hardware resource at the same time
- Approaches to resolving structural hazards
 - **Schedule:** Programmer explicitly avoids scheduling instructions that would create structural hazards
 - **Stall:** Hardware includes control logic that stalls until earlier instruction is no longer using contended resource
 - **Duplicate:** Add more hardware to design so that each instruction can access independent resources at the same time
- Simple 5-stage MIPS pipeline has no structural hazards specifically because ISA was designed that way

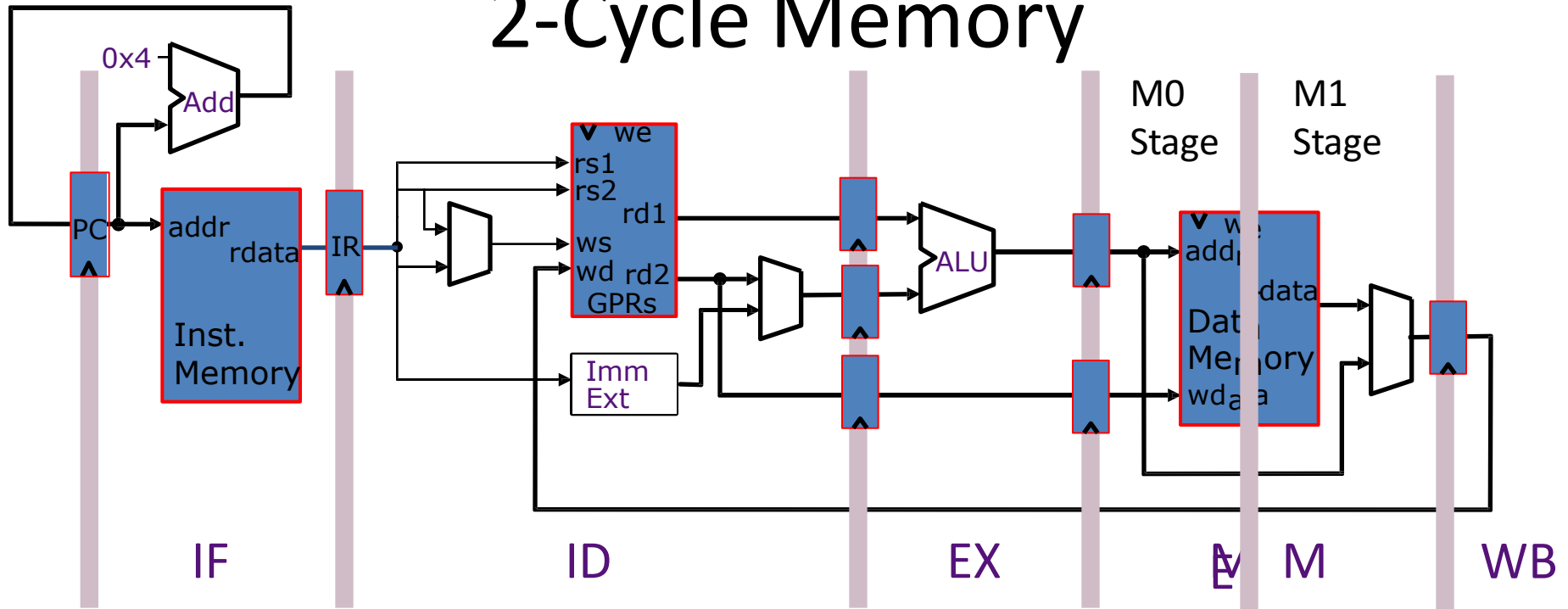
Example Structural Hazard: Unified Memory



Example Structural Hazard: Unified Memory



Example Structural Hazard: 2-Cycle Memory



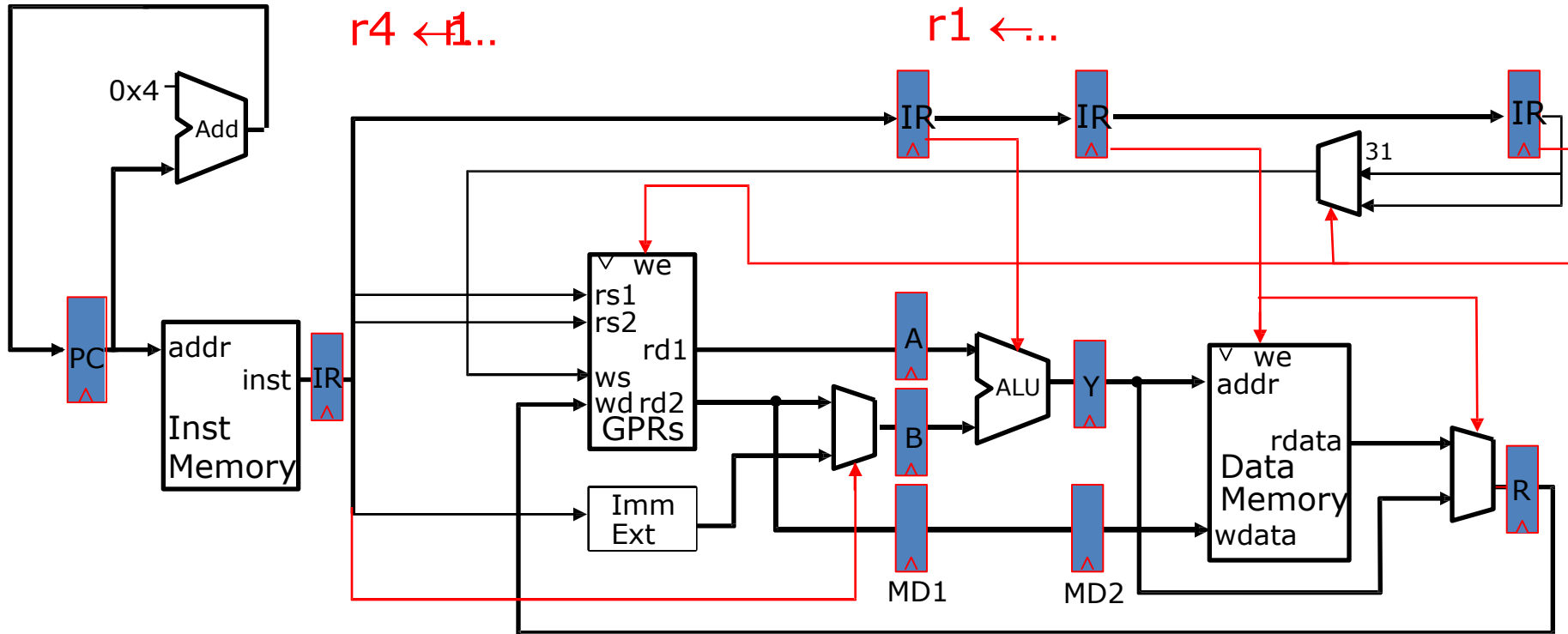
Agenda

- Microcoded Microarchitectures
- Pipeline Review
 - Pipelining Basics
 - Structural Hazards
 - Data Hazards
 - Control Hazards

Overview of Data Hazards

- Data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline
- Approaches to resolving data hazards
 - **Schedule:** Programmer explicitly avoids scheduling instructions that would create data hazards
 - **Stall:** Hardware includes control logic that freezes earlier stages until preceding instruction has finished producing data value
 - **Bypass:** Hardware datapath allows values to be sent to an earlier stage before preceding instruction has left the pipeline
 - **Speculate:** Guess that there is not a problem, if incorrect kill speculative instruction and restart

Example Data Hazard



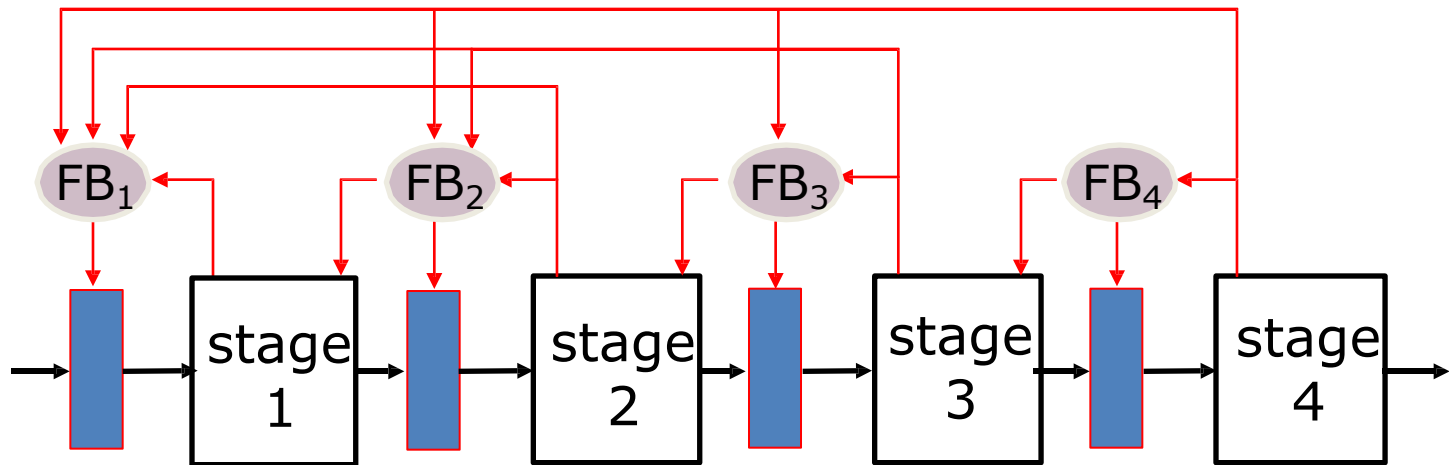
$r4 \leftarrow r1 + 17$

$r1 \leftarrow r0 + 10$

...
 $r1 \leftarrow r0 + 10$ (ADDI R1, R0, #10)
 $r4 \leftarrow r1 + 17$ (ADDI R4, R1, #17)
 ...

$r1$ is stale. Oops!

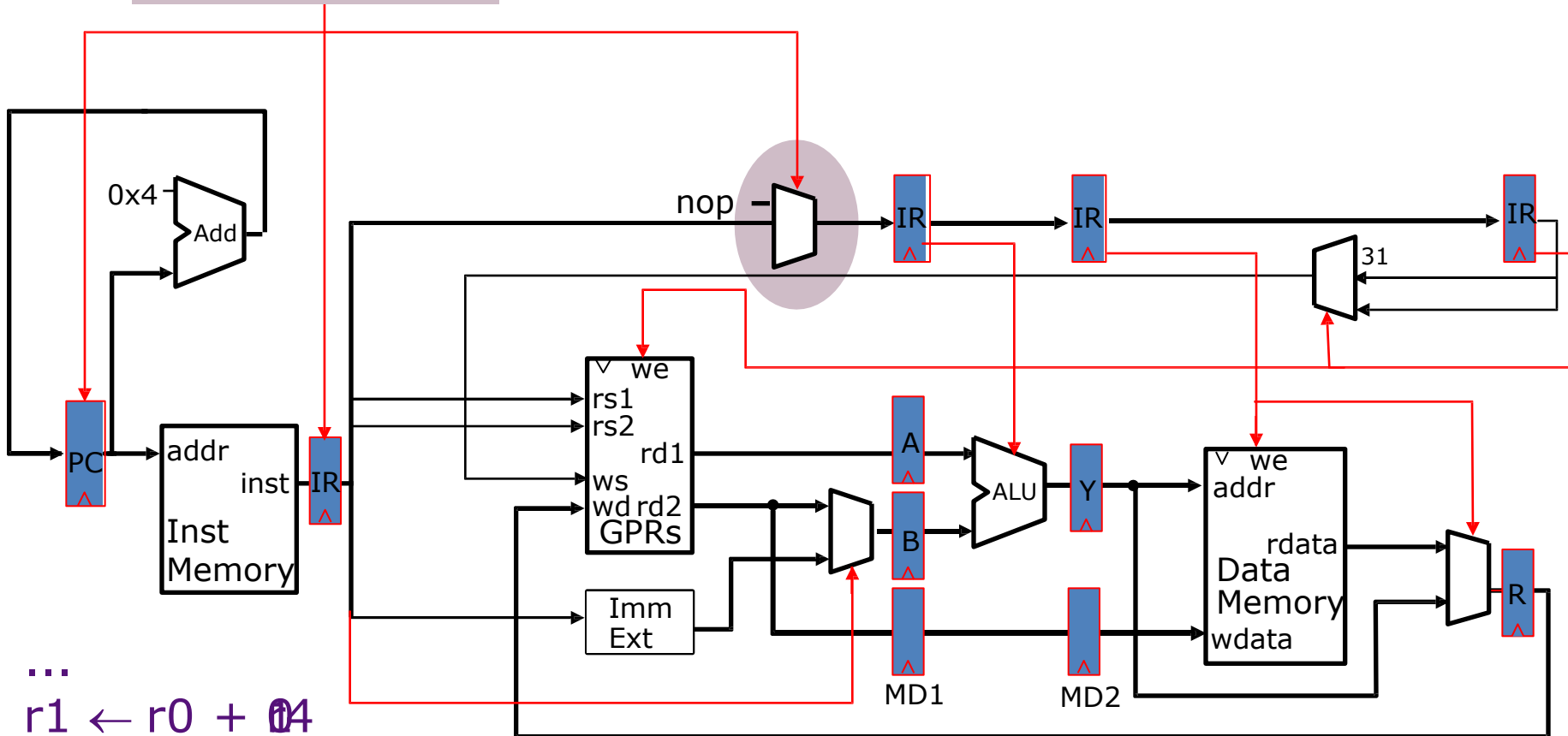
Feedback to Resolve Hazards



- 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 interaction from instructions in stages 1 to i (otherwise deadlock)

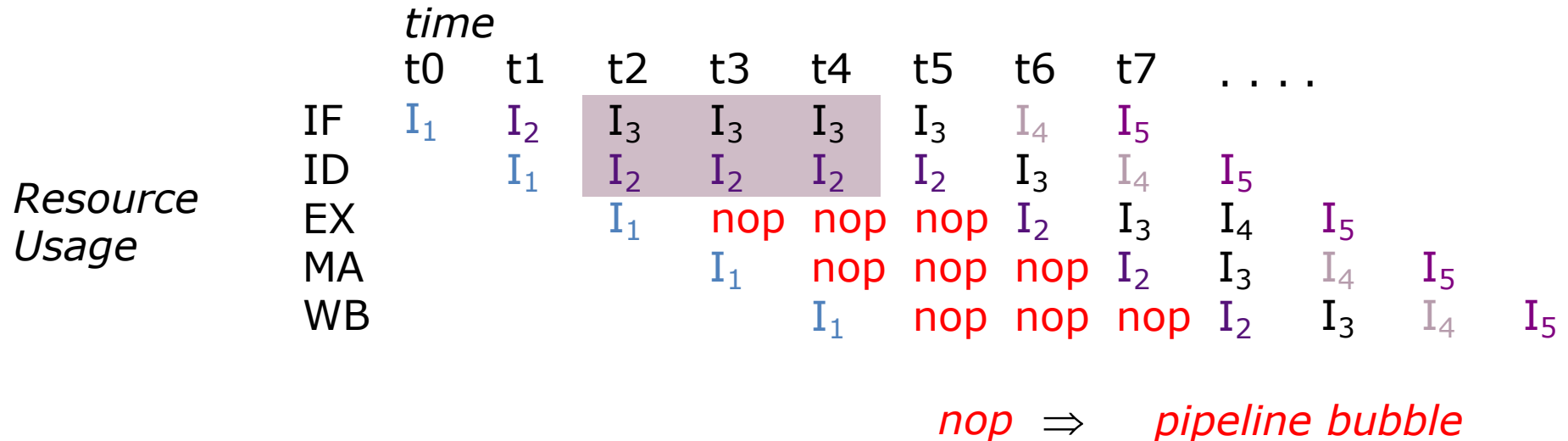
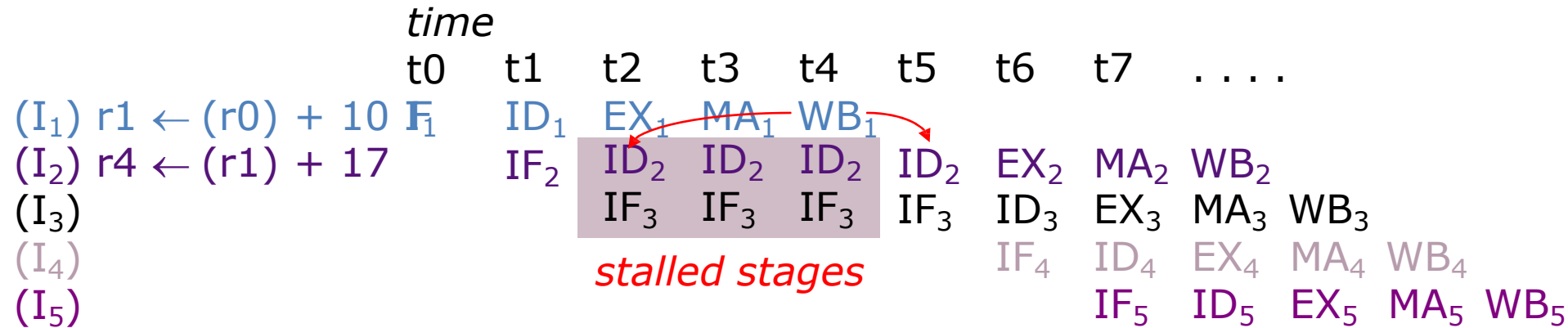
Resolving Data Hazards with Stalls (Interlocks)

Stall Condition

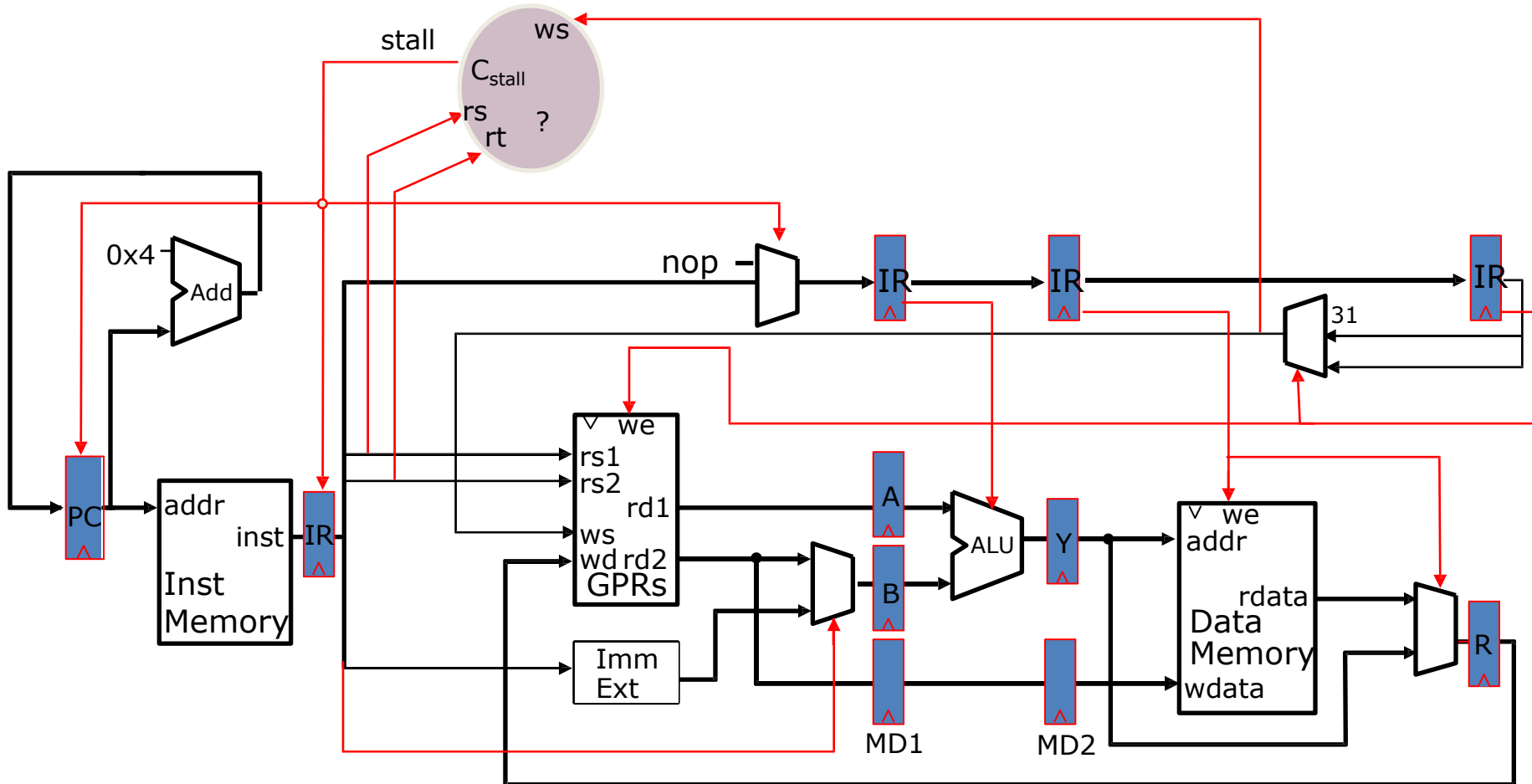


...
 $r1 \leftarrow r0 + 04$
 $\leftarrow r1 + 1$
 ...

Stalled Stages and Pipeline Bubbles

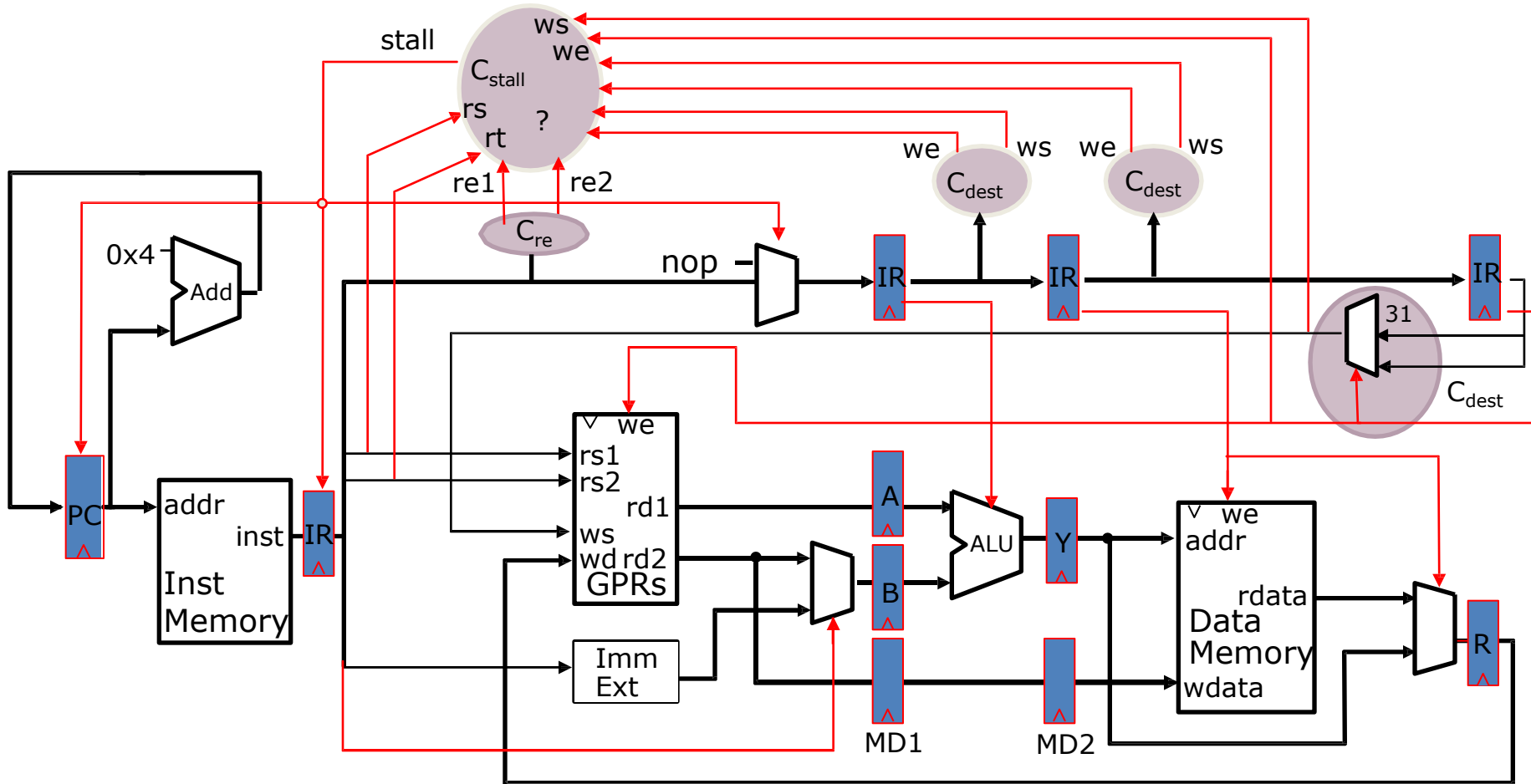


Stall Control Logic



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 \Rightarrow we
 not every instruction reads a register
 \Rightarrow re

Source & Destination Registers



		<i>source(s)</i>	<i>destination</i>
ALU	$rd \leftarrow (rs) \text{ func } (rt)$	rs, rt	rd
ALUI	$rt \leftarrow (rs) \text{ op } \text{immediate}$	rs	rt
LW	$rt \leftarrow M [(rs) + \text{immediate}]$	rs	rt
SW	$M [(rs) + \text{immediate}] \leftarrow (rt)$	rs, rt	
BZ	<i>cond</i> (rs)		
	<i>true:</i> $PC \leftarrow (PC) + \text{immediate}$	rs	
	<i>false:</i> $PC \leftarrow (PC) + 4$	rs	
J	$PC \leftarrow (PC) + \text{immediate}$		
JAL	$r31 \leftarrow (PC), PC \leftarrow (PC) + \text{immediate}$		31
JR	$PC \leftarrow (rs)$	rs	
JALR	$r31 \leftarrow (PC), PC \leftarrow (rs)$	rs	31

Deriving the Stall Signal

C_{dest}

$ws = \text{Case opcode}$

ALU $\Rightarrow rd$
 ALUi, LW $\Rightarrow rt$
 JAL, JALR $\Rightarrow R31$

$we = \text{Case opcode}$

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

C_{re}

$re1 = \text{Case opcode}$

ALU, ALUi,
 LW, SW, BZ,
 JR, JALR $\Rightarrow on$
 J, JAL $\Rightarrow off$

$re2 = \text{Case opcode}$

ALU, SW $\Rightarrow on$
 ... $\Rightarrow of$

f

C_{stall}

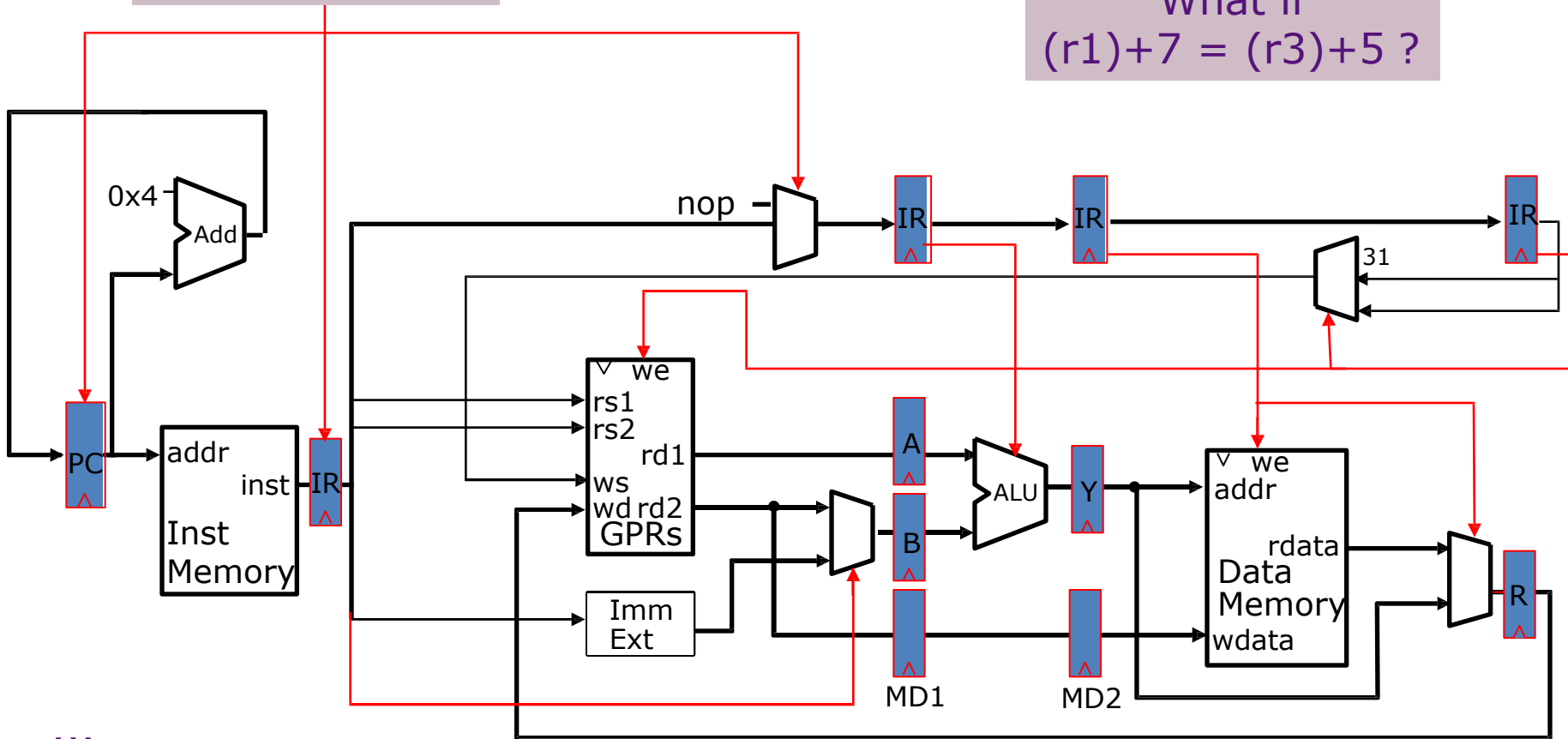
$$\begin{aligned} \text{stall} = & ((rs_D = ws_E).we_E + \\ & (rs_D = ws_M).we_M + \\ & (rs_D = ws_W).we_W) \cdot re1_D + \\ & ((rt_D = ws_E).we_E + \\ & (rt_D = ws_M).we_M + \\ & (rt_D = ws_W).we_W) \cdot re2_D \end{aligned}$$

*This is not
the full story!*

Hazards due to Loads & Stores

Stall Condition

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



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

Is there any possible data hazard in this instruction sequence?

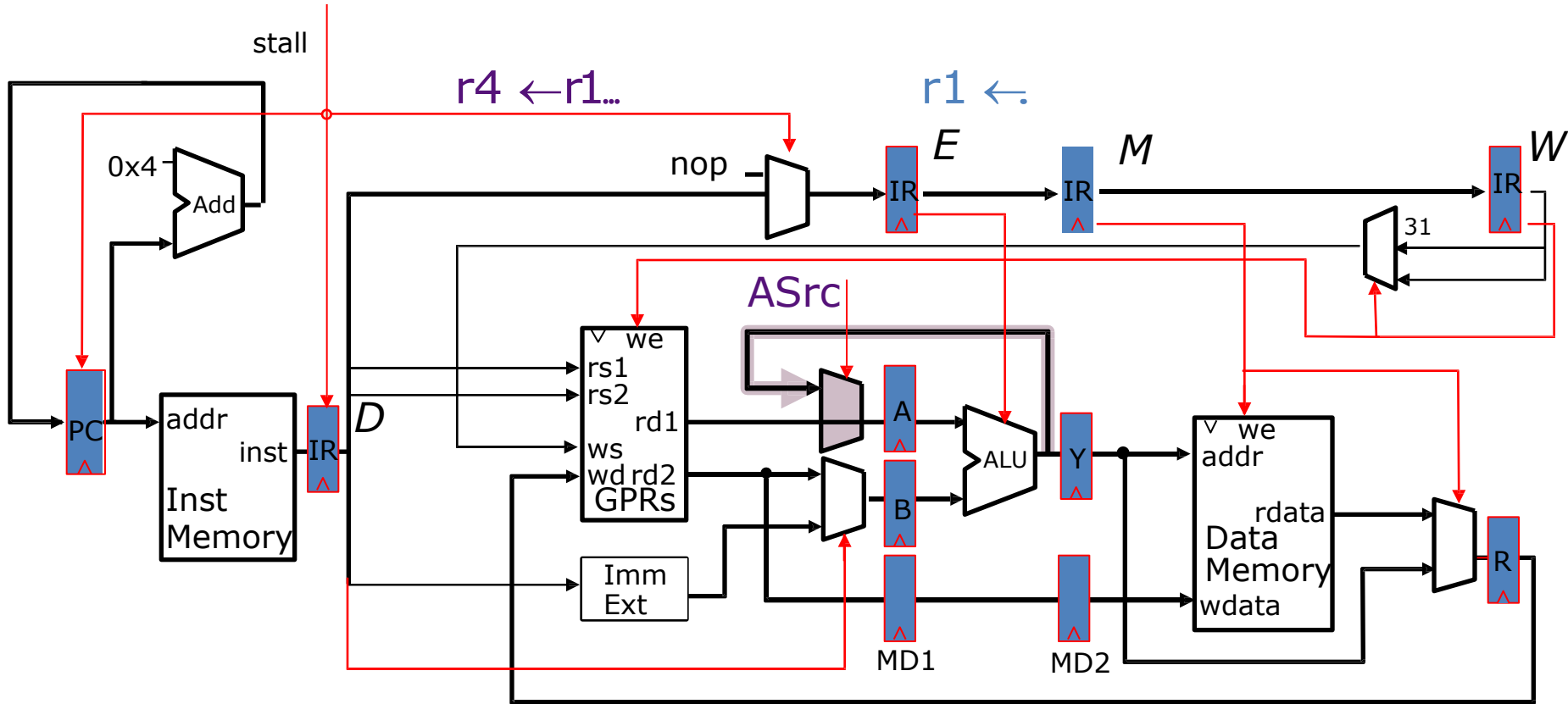
Data Hazards Due to Loads and Store

- Example instruction sequence
 - $\text{Mem}[\text{Regs}[r1] + 7] \leftarrow \text{Regs}[r2]$
 - $\text{Regs}[r4] \leftarrow \text{Mem}[\text{Regs}[r3] + 5]$
- What if $\text{Regs}[r1] + 7 == \text{Regs}[r3] + 5$?
 - Writing and reading to/from the same address
 - Hazard is avoided because our memory system completes writes in a single cycle
 - More realistic memory system will require more careful handling of data hazards due to loads and stores

Overview of Data Hazards

- Data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline
- Approaches to resolving data hazards
 - **Schedule:** Programmer explicitly avoids scheduling instructions that would create data hazards
 - **Stall:** Hardware includes control logic that freezes earlier stages until preceding instruction has finished producing data value
 - **Bypass:** Hardware datapath allows values to be sent to an earlier stage before preceding instruction has left the pipeline
 - **Speculate:** Guess that there is not a problem, if incorrect kill speculative instruction and restart

Adding Bypassing to the Datapath



When does this bypass help?

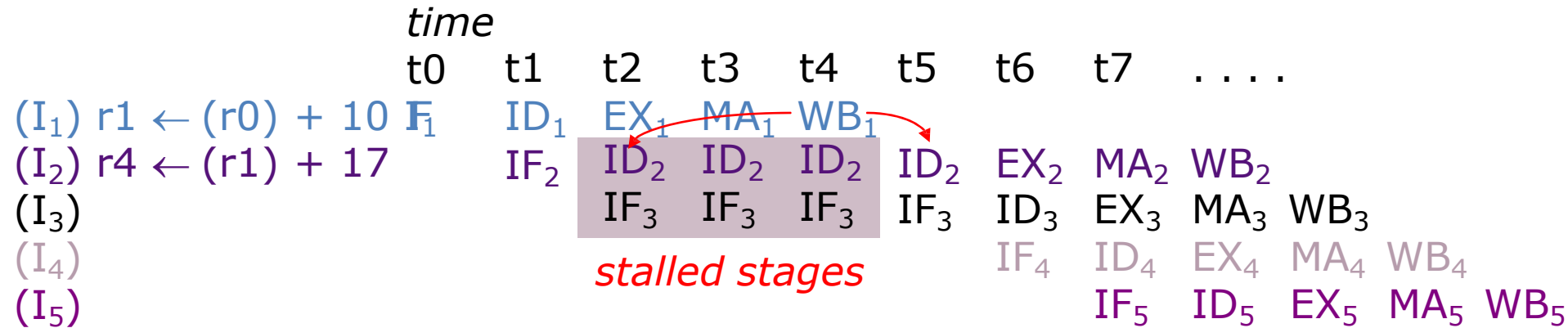
...

(I₁) $r1 \leftarrow r0 + 10$
 (I₂) $r4 \leftarrow r1 + 17$

$r1 \leftarrow \text{Mem}[r0 + 10]$
 $r4 \leftarrow r1 + 17$

JAL 500
 $r4 \leftarrow r31 + 17$

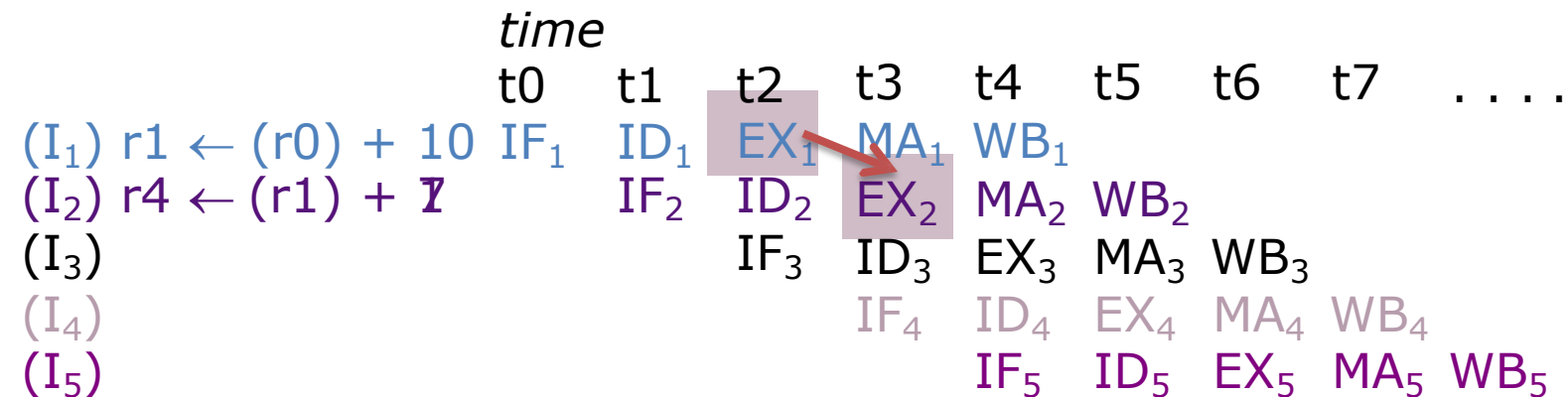
Deriving the Bypass Signal



Each stall or kill introduces a bubble in the pipeline

$$\Rightarrow \text{CPI} > 1$$

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



The Bypass Signal

Deriving it from the Stall Signal

$$\text{stall} = (\cancel{((rs_D = ws_E).we_E)} + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re1_D \\ + ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re2_D)$$

ws = Case opcode

ALU \Rightarrow rd

ALUi, LW \Rightarrow rt

JAL, JALR \Rightarrow R31

we = Case opcode

ALU, ALUi, LW \Rightarrow (ws \neq 0)

JAL, JALR \Rightarrow on

... \Rightarrow off

$$\text{ASrc} = (rs_D = ws_E).we_E.re1_D$$

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

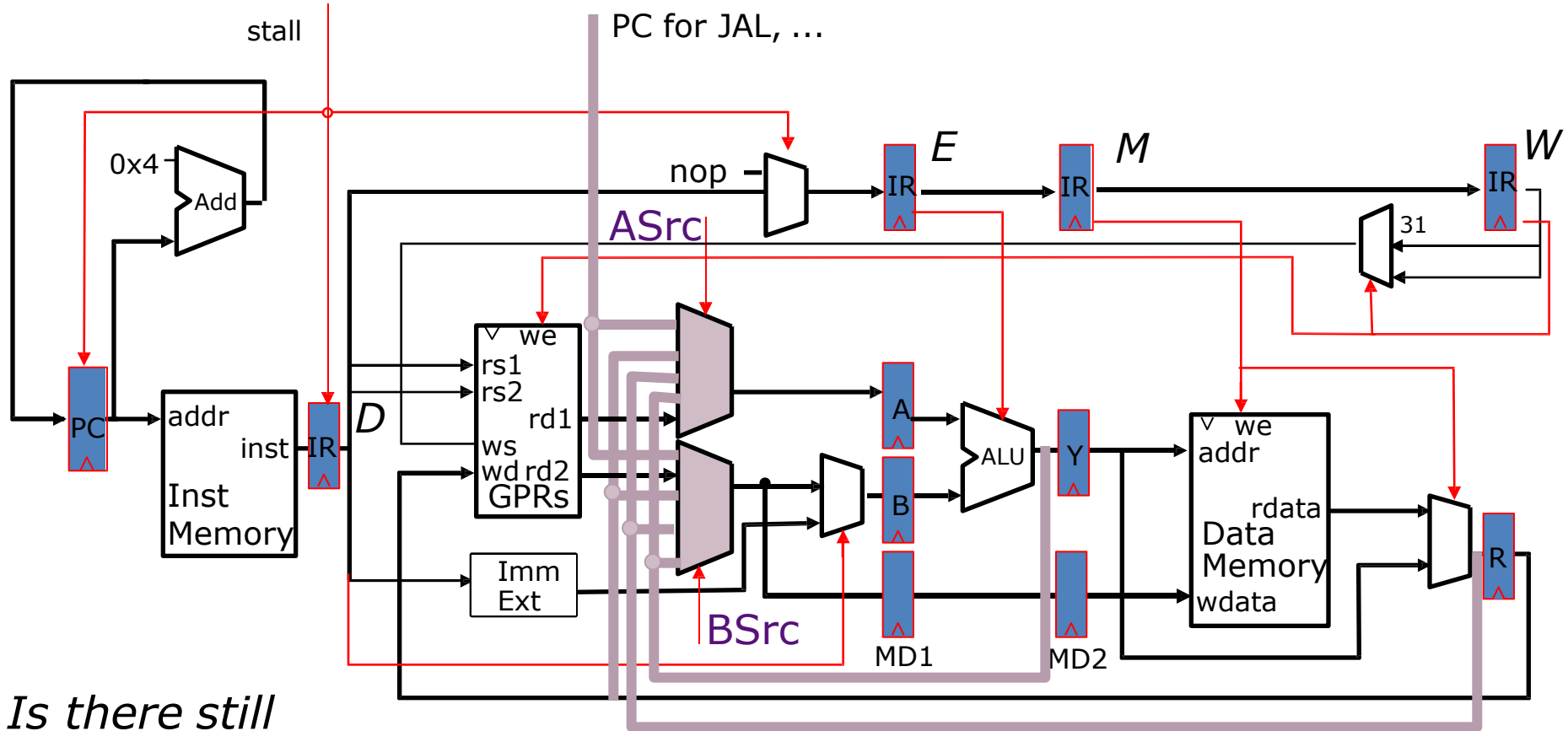
$we_bypass_E = \text{Case opcode}_E$
ALU, ALUi $\Rightarrow (ws \neq 0)$
... $\Rightarrow \text{off}$

$we_stall_E = \text{Case opcode}_E$
LW $\Rightarrow (ws \neq 0)$
JAL, JALR $\Rightarrow \text{on}$
... $\Rightarrow \text{off}$

$ASrc = (rs_D = ws_E).we_bypass_E . re1_D$

$stall = ((rs_D = ws_E).we_stall_E +$
 $(rs_D = ws_M).we_M + (rs_D = ws_W).we_W). re1_D$
 $+ ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W). re2_D$

Fully Bypassed Datapath



Is there still a need for the stall signal ?

$$\text{stall} = (\text{rs}_D = \text{ws}_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (\text{ws}_E \neq 0) \cdot \text{re1}_D + (\text{rt}_D = \text{ws}_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (\text{ws}_E \neq 0) \cdot \text{re2}_D$$

Overview of Data Hazards

- Data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline
- Approaches to resolving data hazards
 - **Schedule:** Programmer explicitly avoids scheduling instructions that would create data hazards
 - **Stall:** Hardware includes control logic that freezes earlier stages until preceding instruction has finished producing data value
 - **Bypass:** Hardware datapath allows values to be sent to an earlier stage before preceding instruction has left the pipeline
 - **Speculate:** Guess that there is not a problem, if incorrect kill speculative instruction and restart

Later in course

Agenda

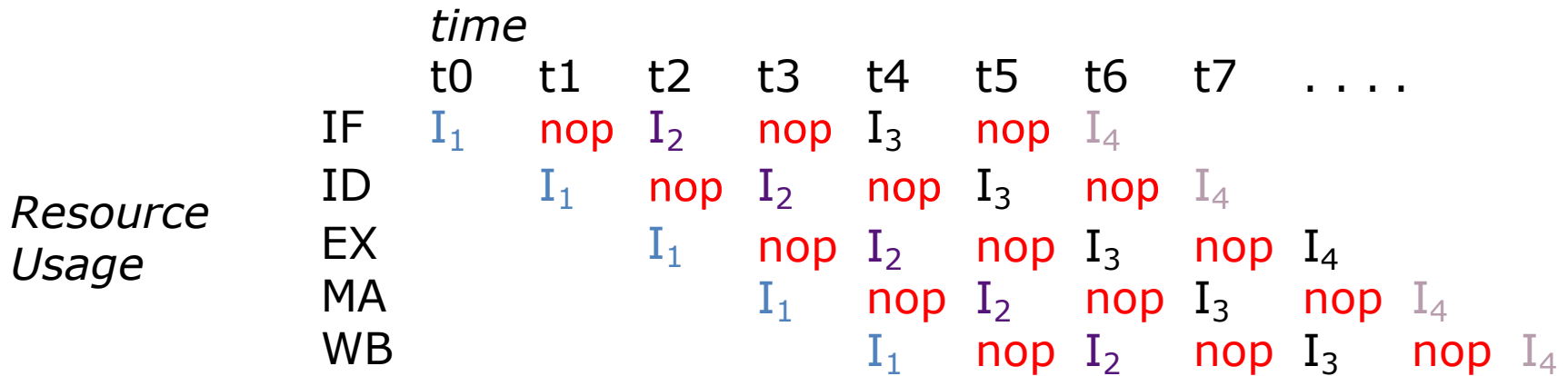
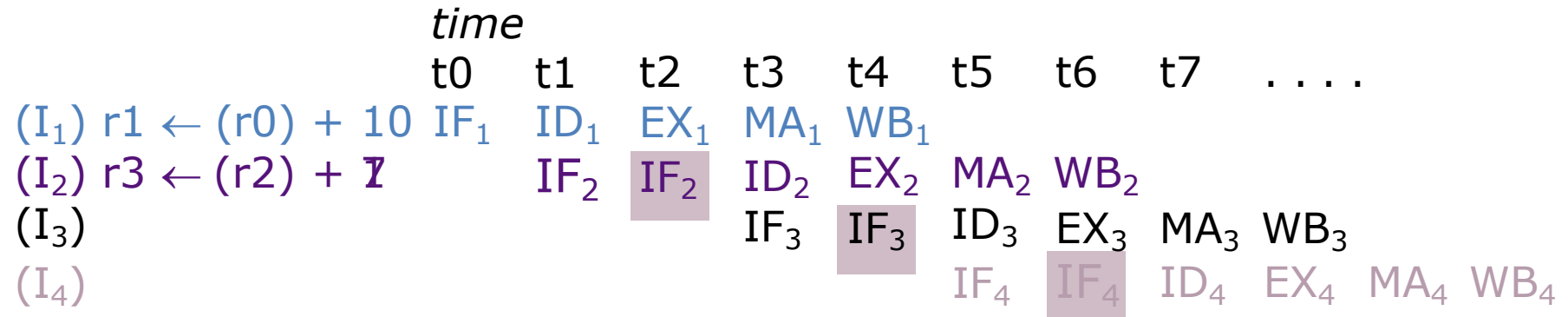
- Microcoded Microarchitectures
- Pipeline Review
 - Pipelining Basics
 - Structural Hazards
 - Data Hazards
 - Control Hazards

Control Hazards

- 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, PC, Register (for condition), and offset
 - For all other instructions
 - Opcode and PC
 - have to know it's not one of above!

Opcode Decoding Bubble

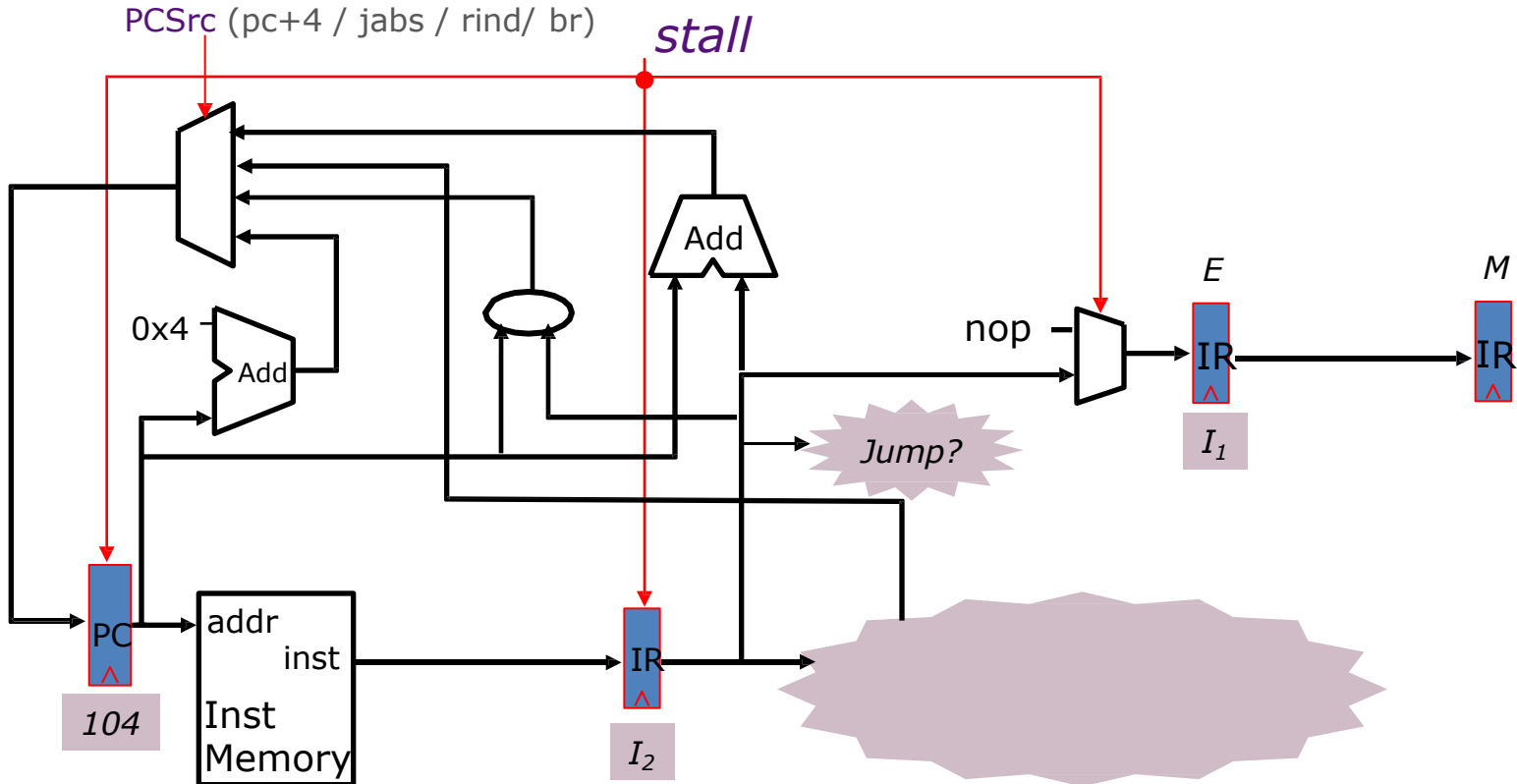
(assuming no branch delay slots for now)



CPI = 2!

nop ⇒ pipeline bubble

Speculate next address is PC+4

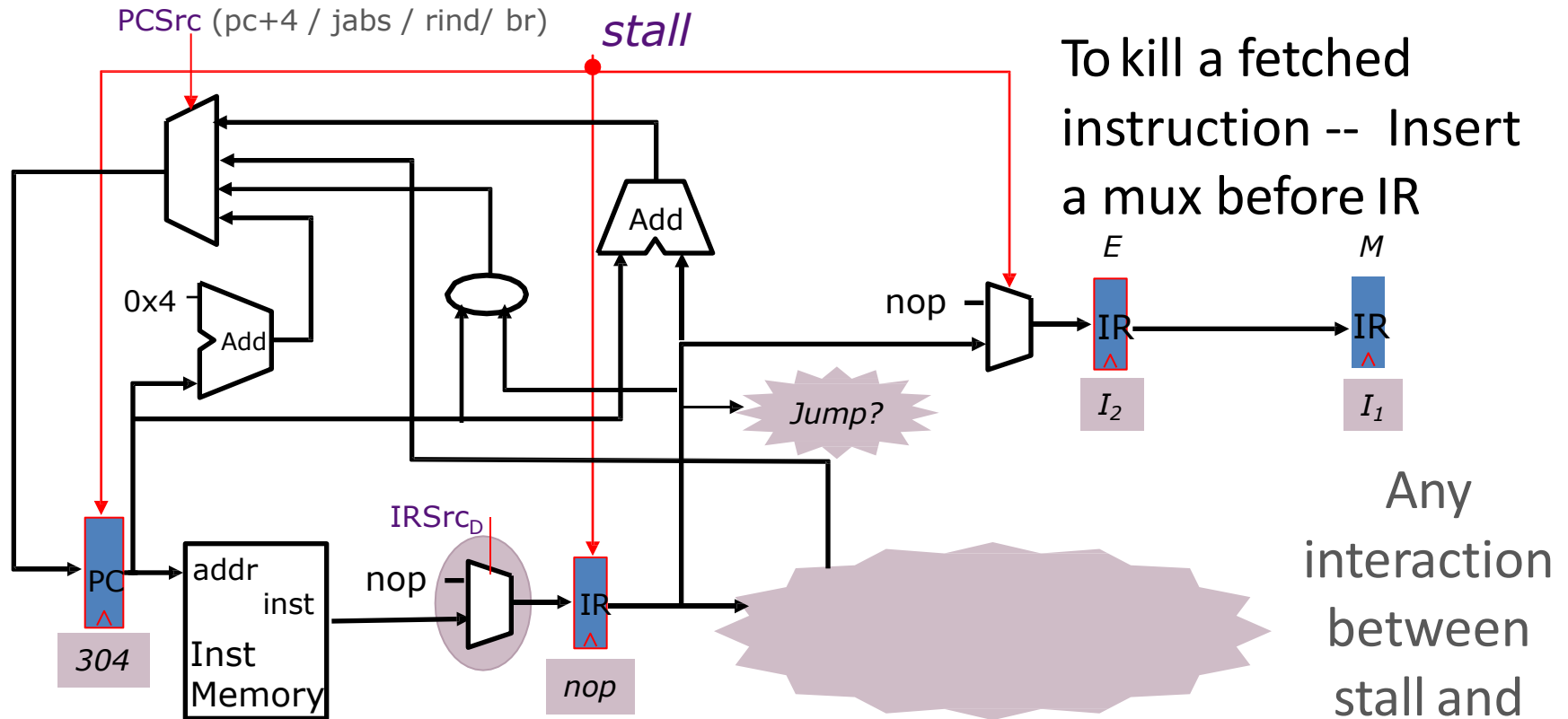


I_1	096	ADD	
I_2	100	J 304	
I_3	104	ADD	<i>kill</i>
I_4	304	ADD	

A jump instruction kills (not stalls) the following instruction

How?

Pipelining Jumps



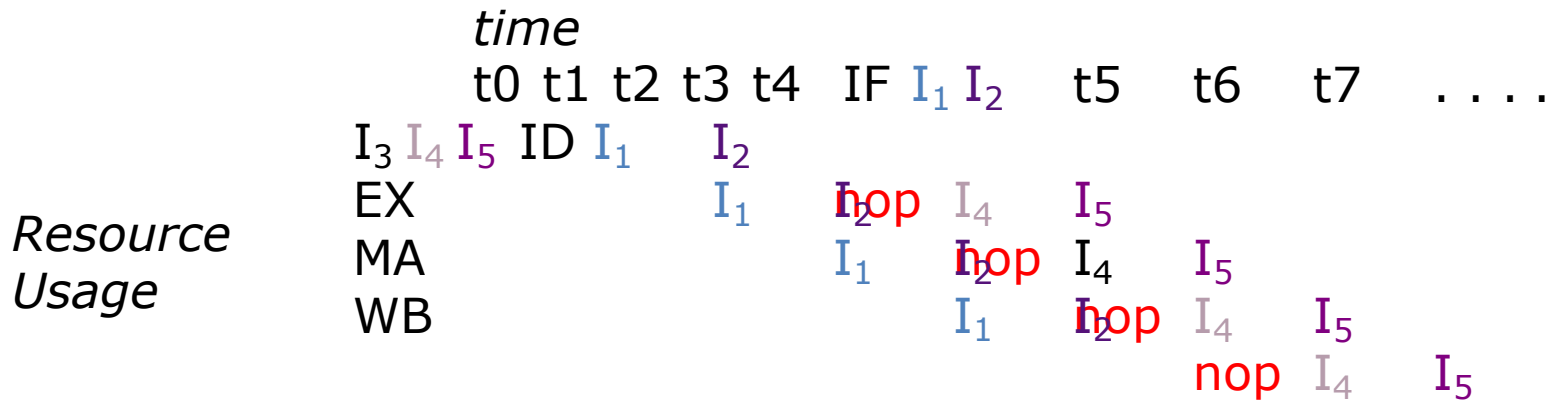
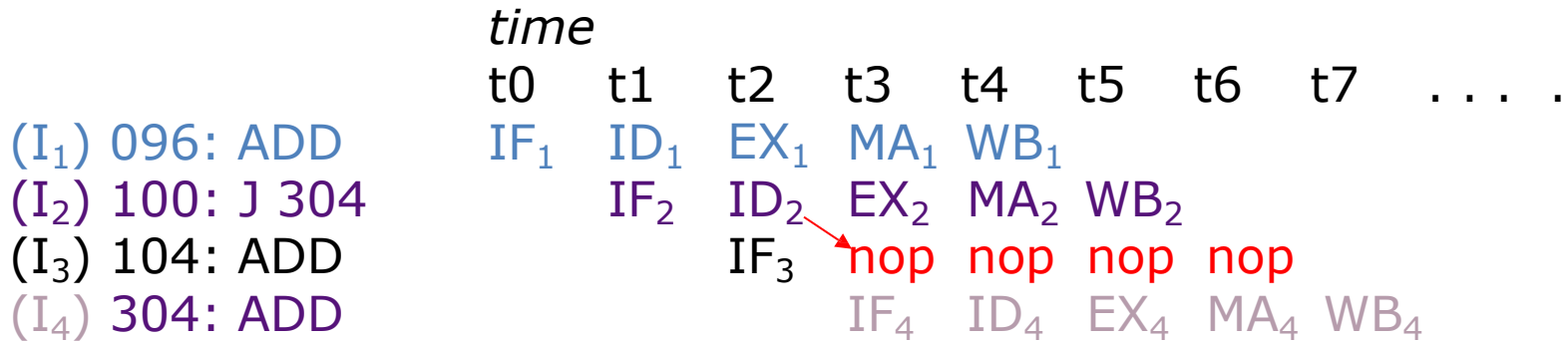
I_1	096	ADD	
I_2	100	J 304	
I_3	104	ADD	<i>kill</i>
I_4	304	ADD	

IRSrc_D = Case opcode_D

J, JAL ⇒ nop

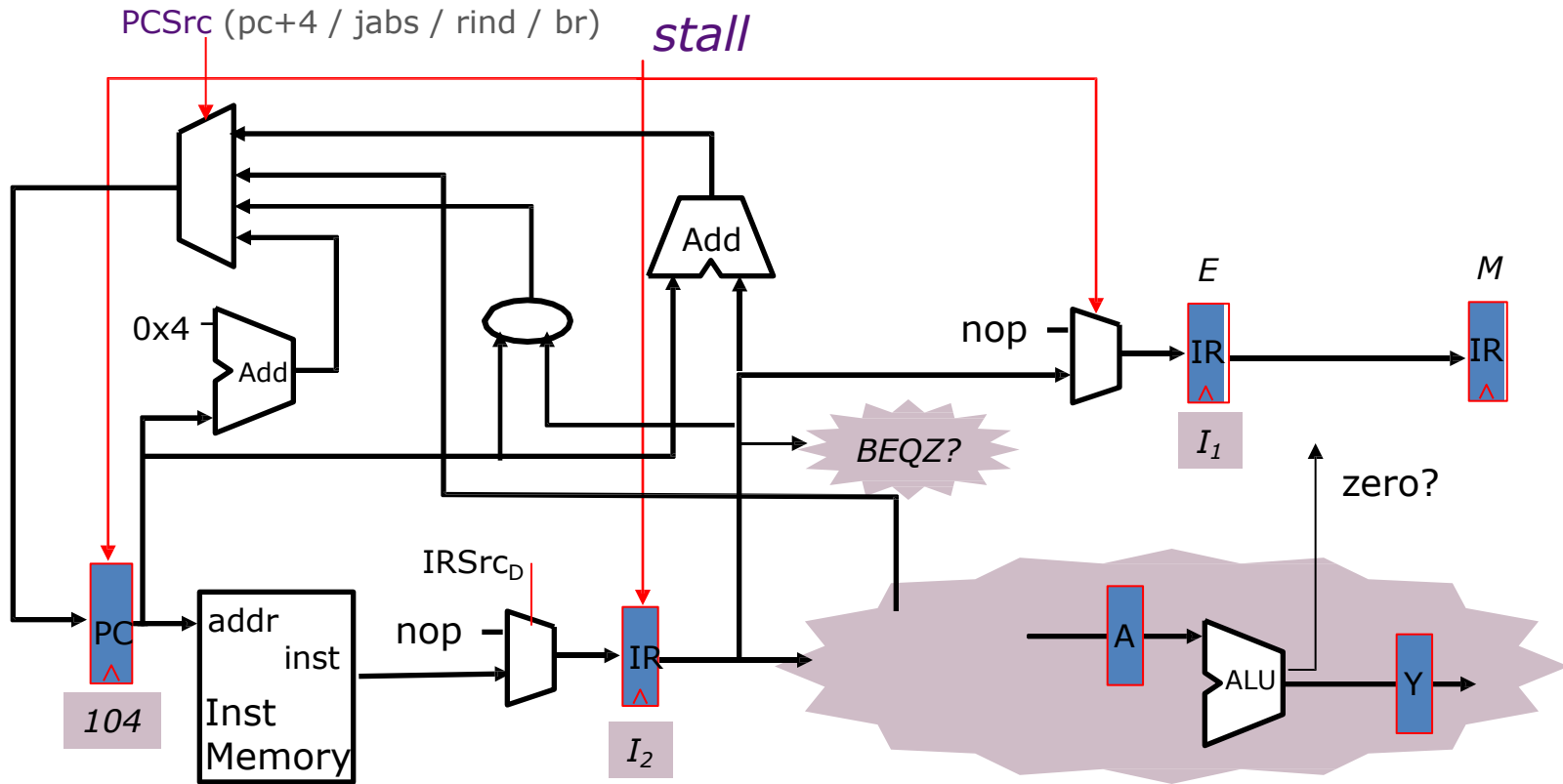
... ⇒ IM

Jump Pipeline Diagrams



nop ⇒ *pipeline bubble*

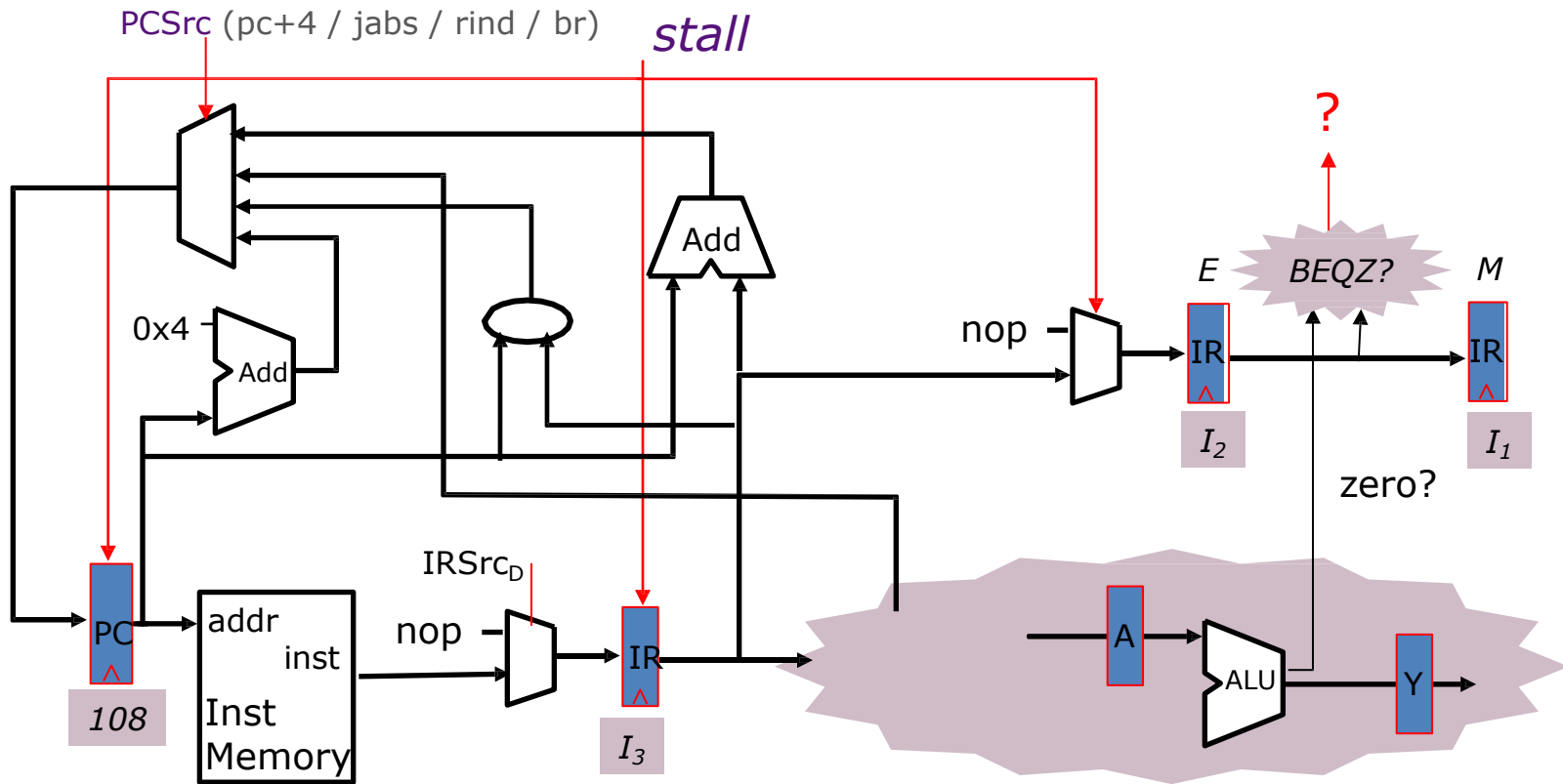
Pipelining Conditional Branches



I ₁	096	ADD
I ₂	100	BEQZ r1 +200
I ₃	104	ADD
	108	...
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



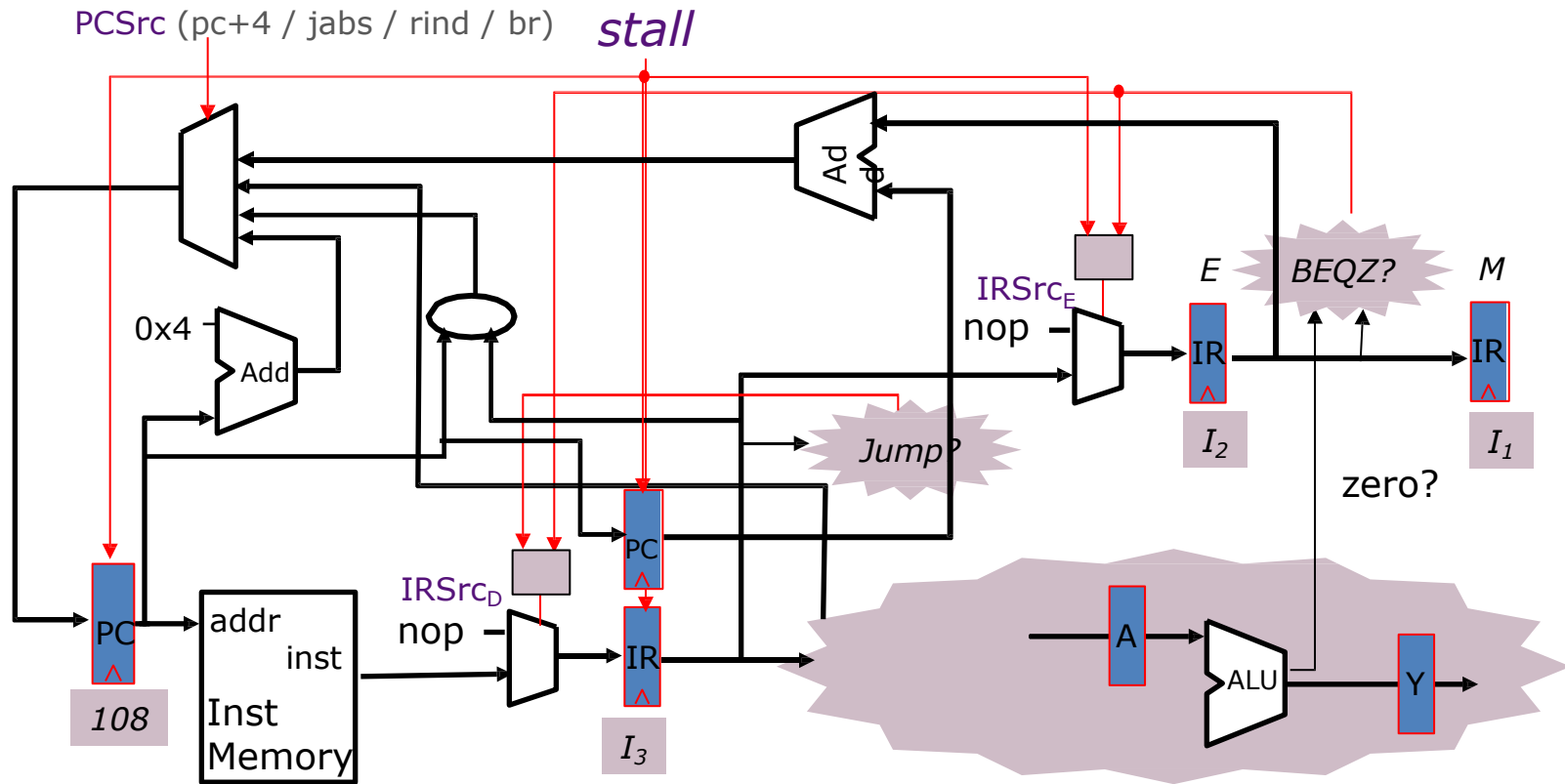
If the branch is taken

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

⇒ *stall signal is not valid*

I ₁	096	ADD
I ₂	100	BEQZ r1 +200
I ₃	104	ADD
	108	...
I ₄	304	ADD

Pipelining Conditional Branches



If the branch is taken

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

⇒ *stall signal is not valid*

I ₁	096	ADD
I ₂	100	BEQZ r1 +200
I ₃	104	ADD
	108	...
I ₄	304	ADD

New Stall Signal

$$\text{stall} = (((rs_D = ws_E).we_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re1_D \\ + ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re2_D) \\ . !((opcode_E = BEQZ).z + (opcode_E = BNEZ).!z)$$

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

Instruction at the decode stage is invalid

Control Equations for PC and IR Muxes

```
PCSrc = Case opcodeE
  BEQZ.z, BNEZ.!z    ⇒ br
  ...                ⇒
                    Case opcodeD
                      J, JAL    ⇒ jabs
                      JR, JALR  ⇒ rind
                      ...       ⇒ pc+4
```

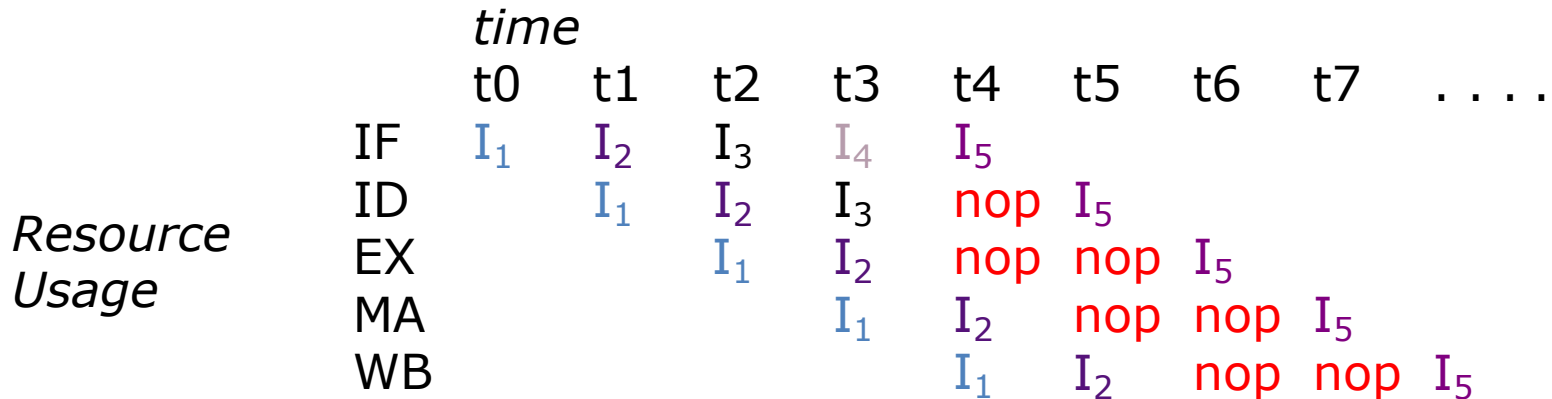
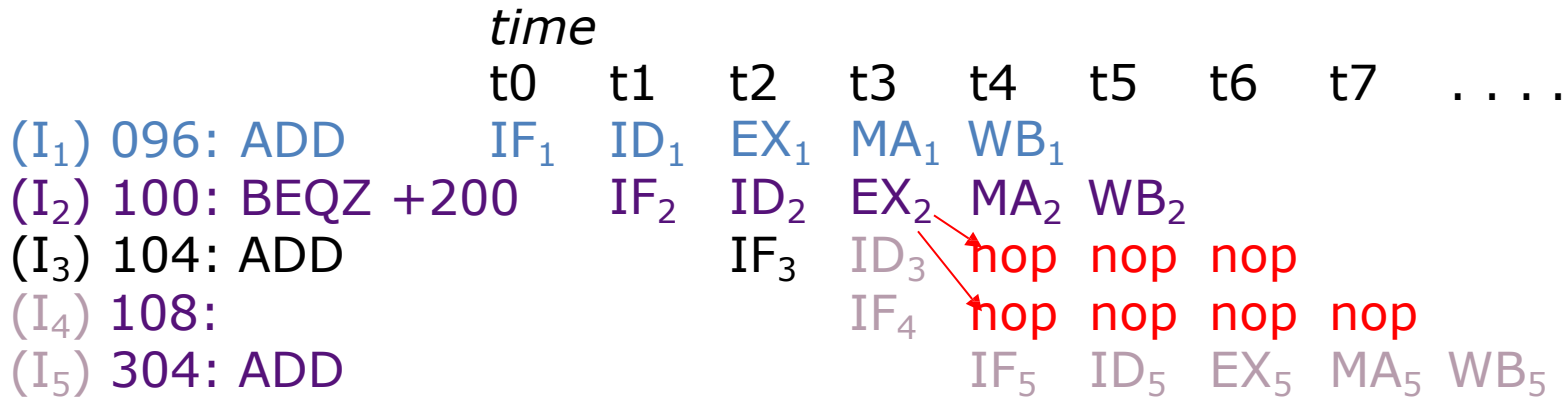
```
IRSrcD = Case opcodeE
  BEQZ.z, BNEZ.!z    ⇒ nop
  ...                ⇒
                    Case opcodeD
                      J, JAL, JR, JALR ⇒ nop
                      ...              ⇒ IM
```

```
IRSrcE = Case opcodeE
  BEQZ.z, BNEZ.!z    ⇒ nop
  ...                ⇒ stall.nop + !stall.IRD
```

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

Branch Pipeline Diagrams

(resolved in execute stage)

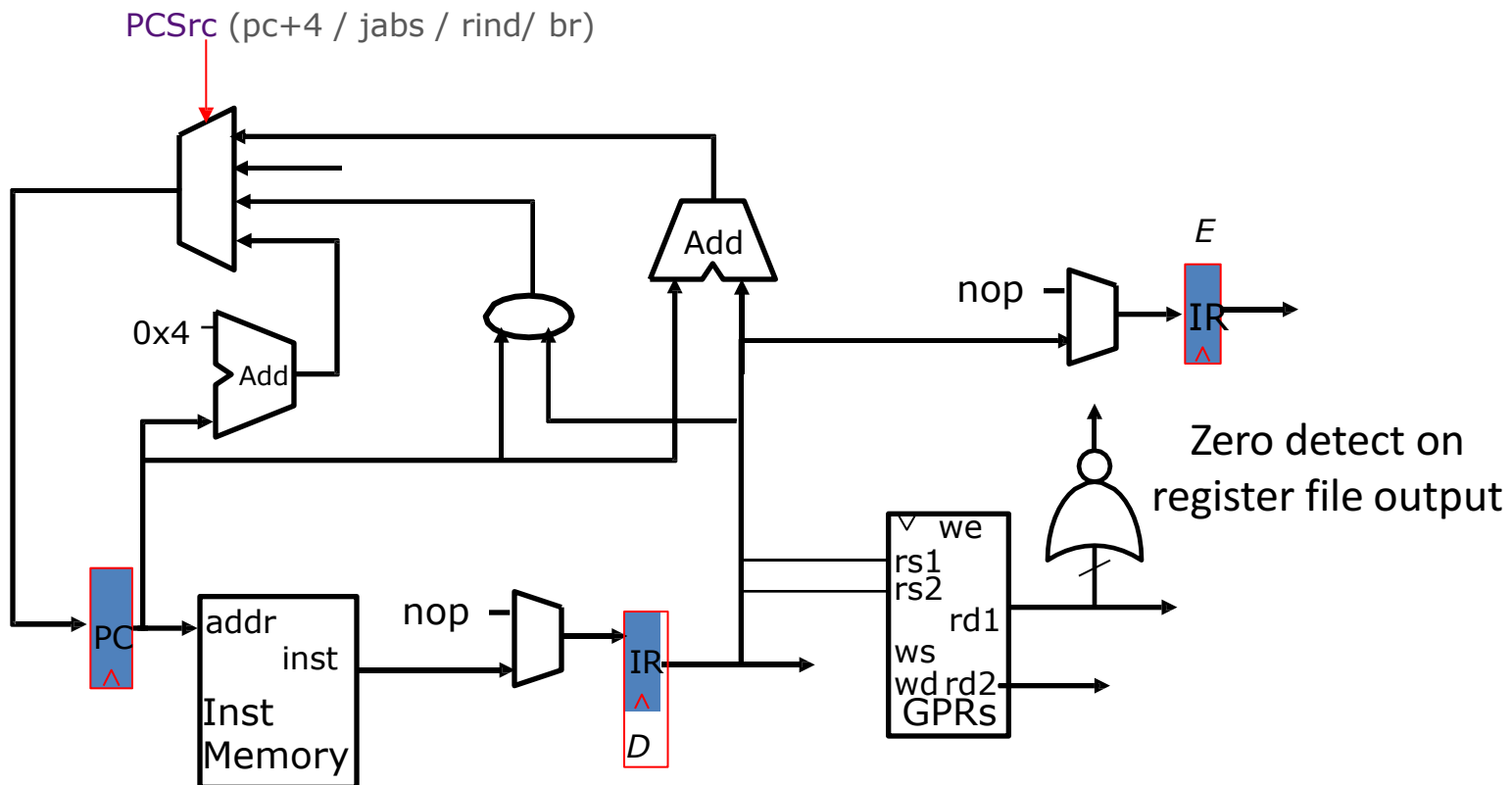


nop ⇒ *pipeline bubble*

Reducing Branch Penalty

(resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage
 - But might elongate cycle time

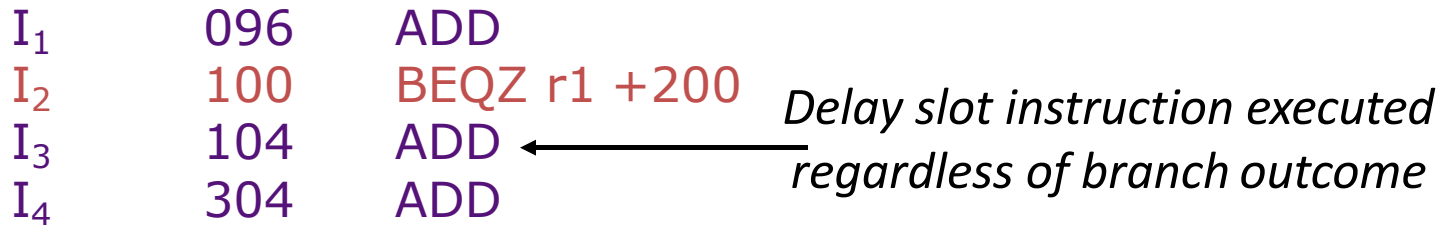


*Pipeline diagram now same as for jumps*₆₄

Branch Delay Slots

(expose control hazard to software)

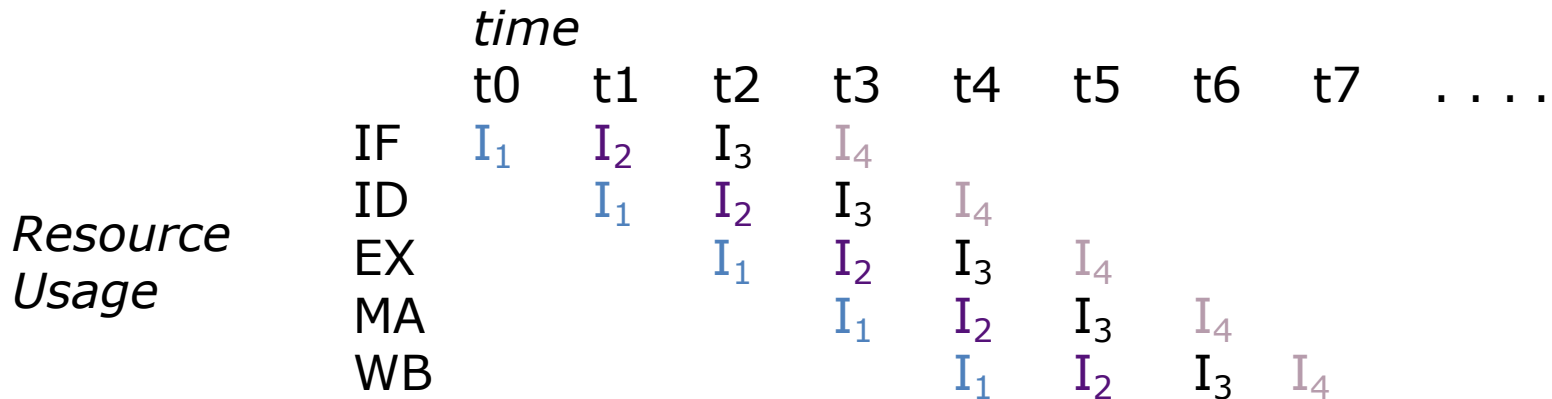
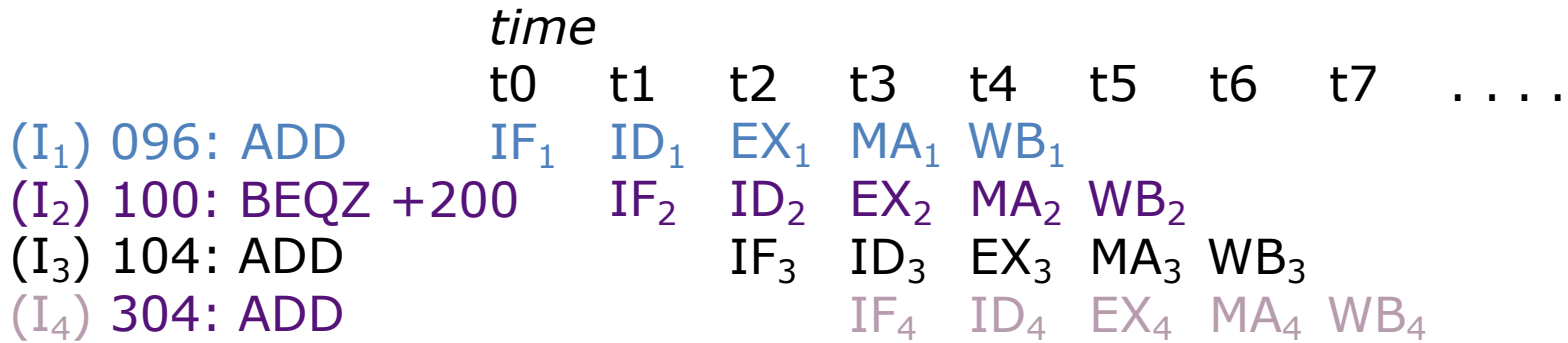
- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
 - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.



- Other techniques include more advanced branch prediction, which can dramatically reduce the branch penalty... *to come later*

Branch Pipeline Diagrams

(branch delay slot)



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 (pipeline interlocks added in hardware)
 - MIPS: “**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages”
- 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. NOPs not counted in useful CPI (alternatively, increase instructions/program)

Other Control Hazards

- Exceptions
- Interrupts

More on this later in the course

Agenda

- Microcoded Microarchitectures
- Pipeline Review
 - Pipelining Basics
 - Structural Hazards
 - Data Hazards
 - Control Hazards