ELE 475 / COS 475 Computer Architecture Slide Deck 2: Microcode and Pipelining Review

David Wentzlaff

Department of Electrical Engineering

Princeton University

PRINCETON UNIVERSITY

PRINCETON School of Engineering and Applied Science

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

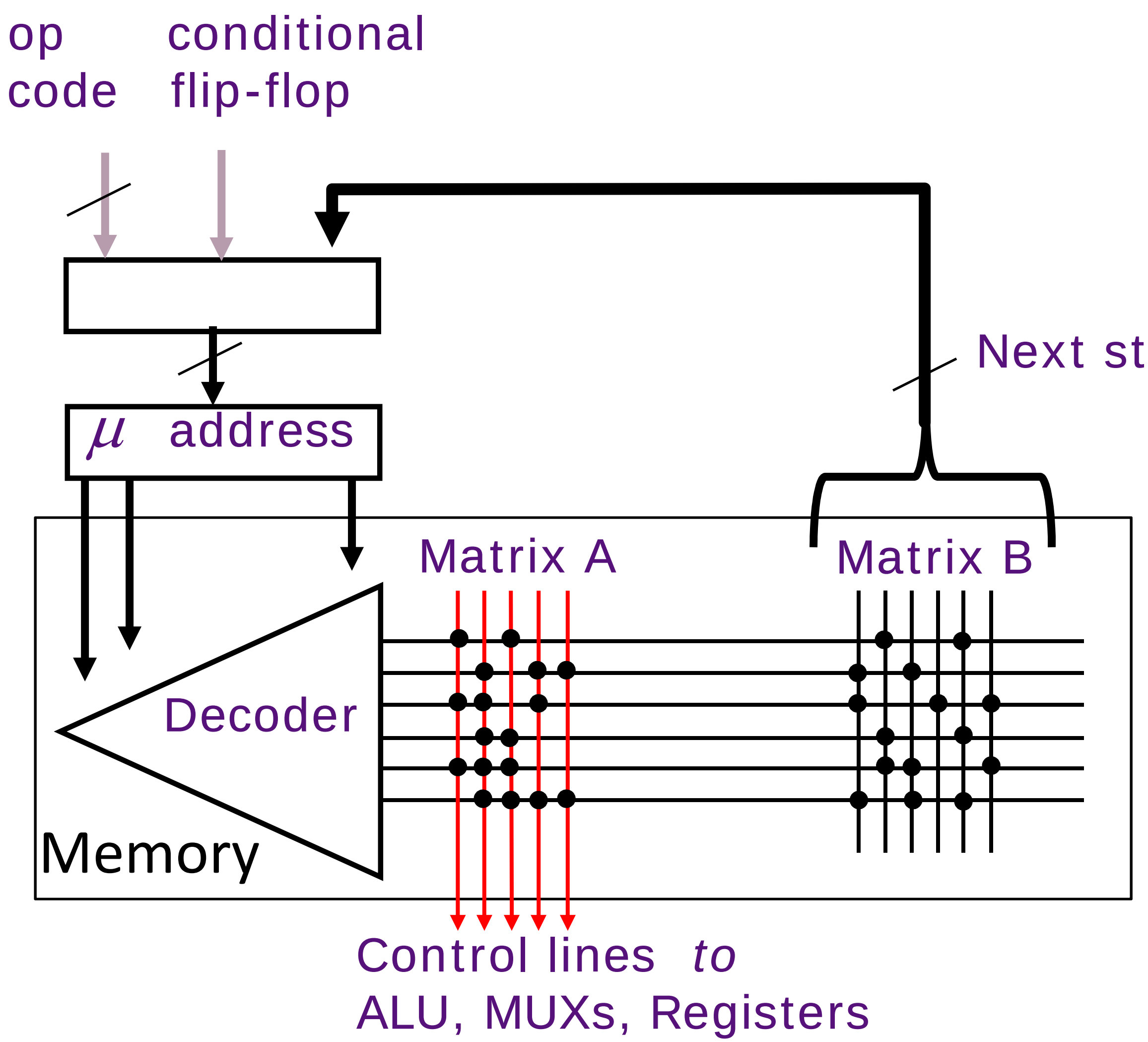
What Happens When the Processor is Too Large?

What Happens When the Processor is Too Large?

- Time Multiplex Resources!

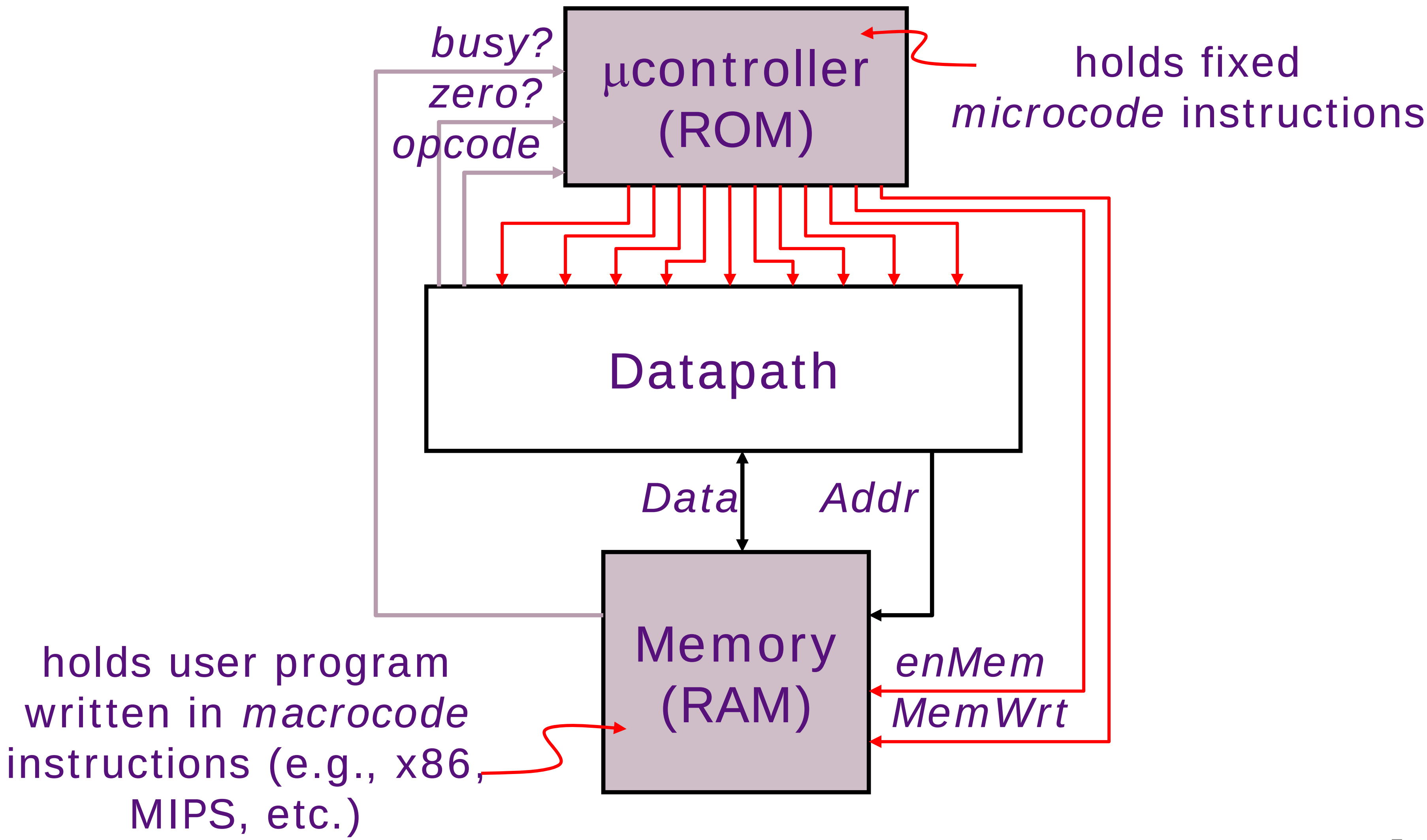
Microcontrol Unit *Maurice Wilkes, 1954*

First used in EDSAC-2, completed 1958

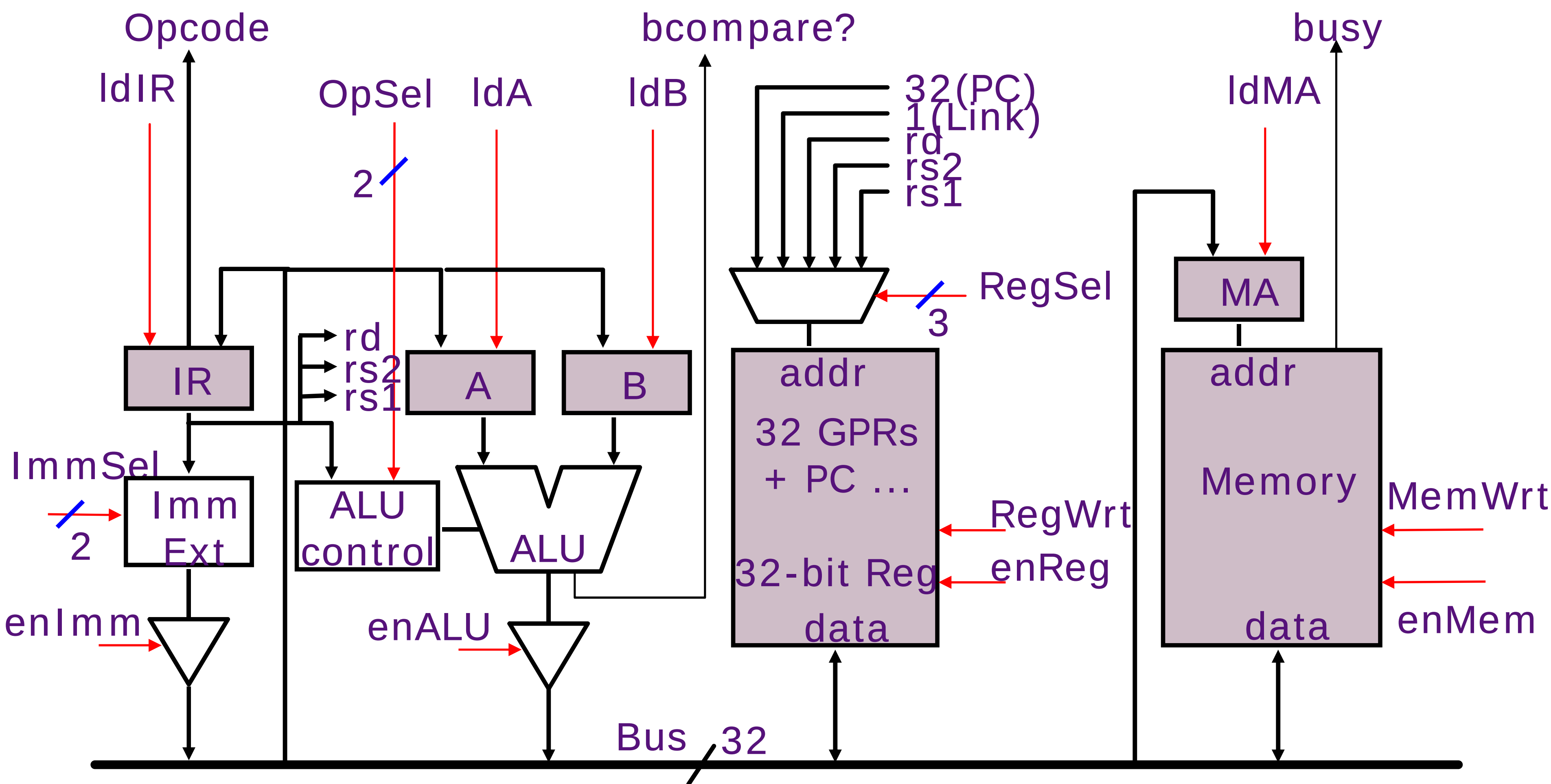


Embed the control logic state table in a memory array

Microcoded Microarchitecture



A Bus-based Datapath for RISC

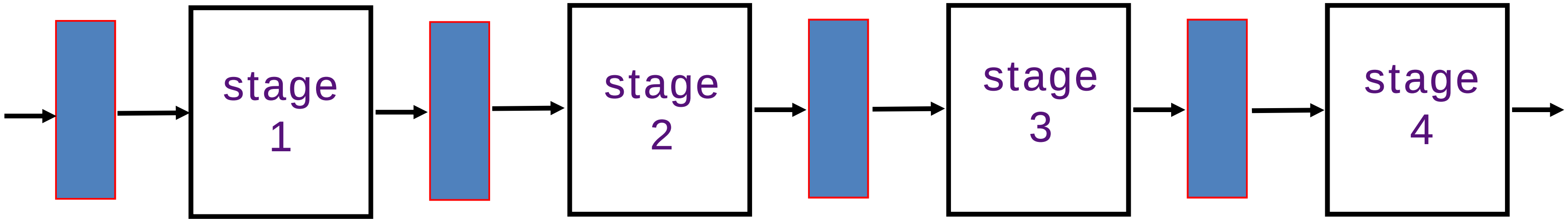


Microinstruction: register to register transfer (17 control signals)

Agenda

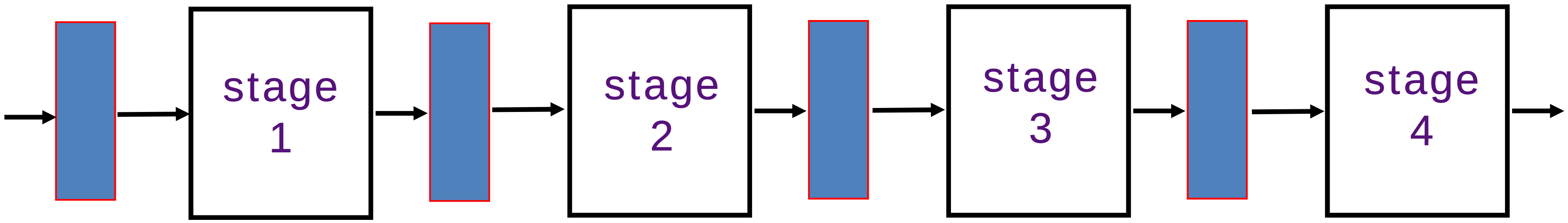
- Microcoded Microarchitectures
- Pipeline Review
  - Pipelining Basics
  - Structural Hazards
  - Data Hazards
  - Control Hazards

An Ideal Pipeline



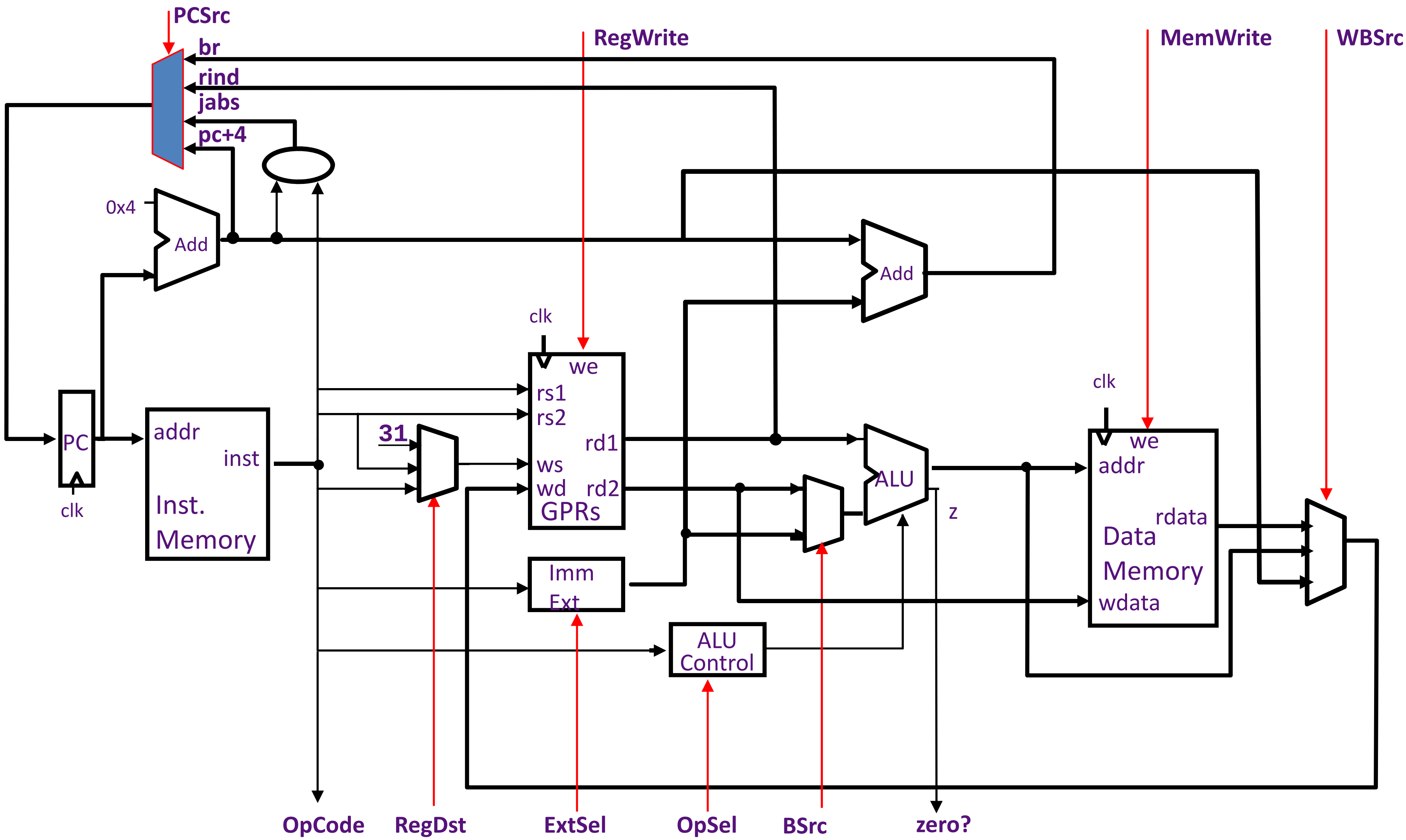
- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- Scheduling of a transaction entering the pipeline is not affected by the transactions in other stages

An Ideal Pipeline

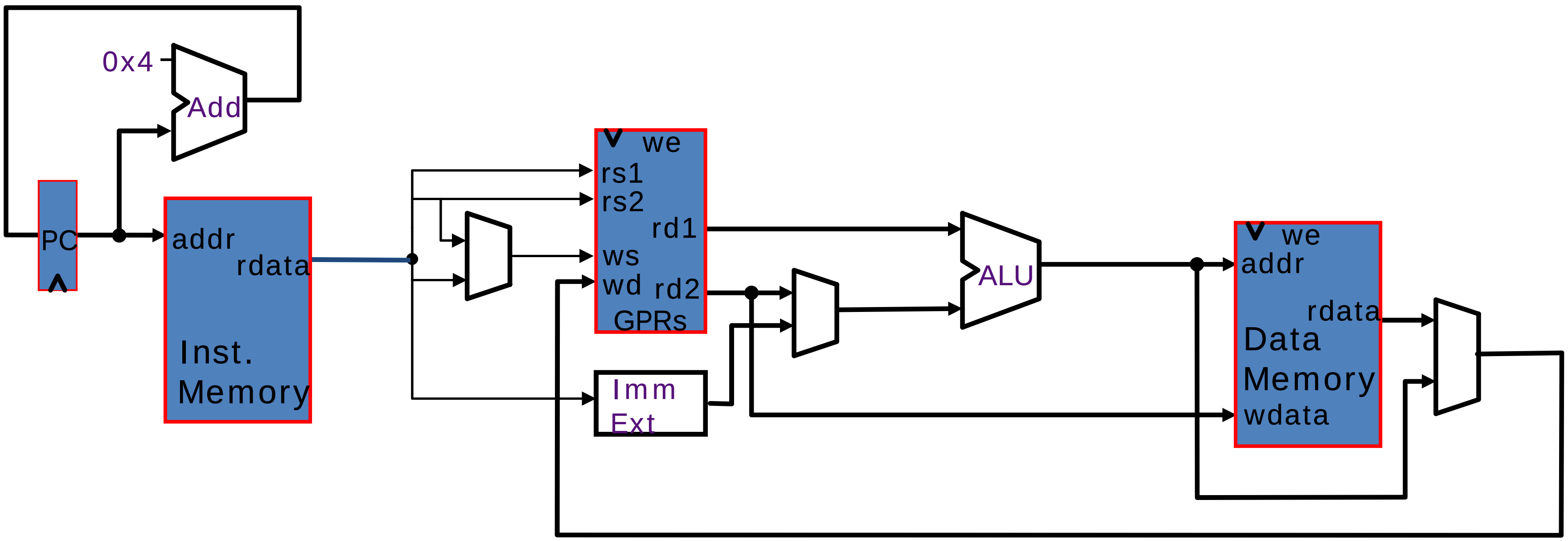


- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- Scheduling of a transaction entering the pipeline is not affected by the transactions in other stages
- These conditions generally hold for industry assembly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

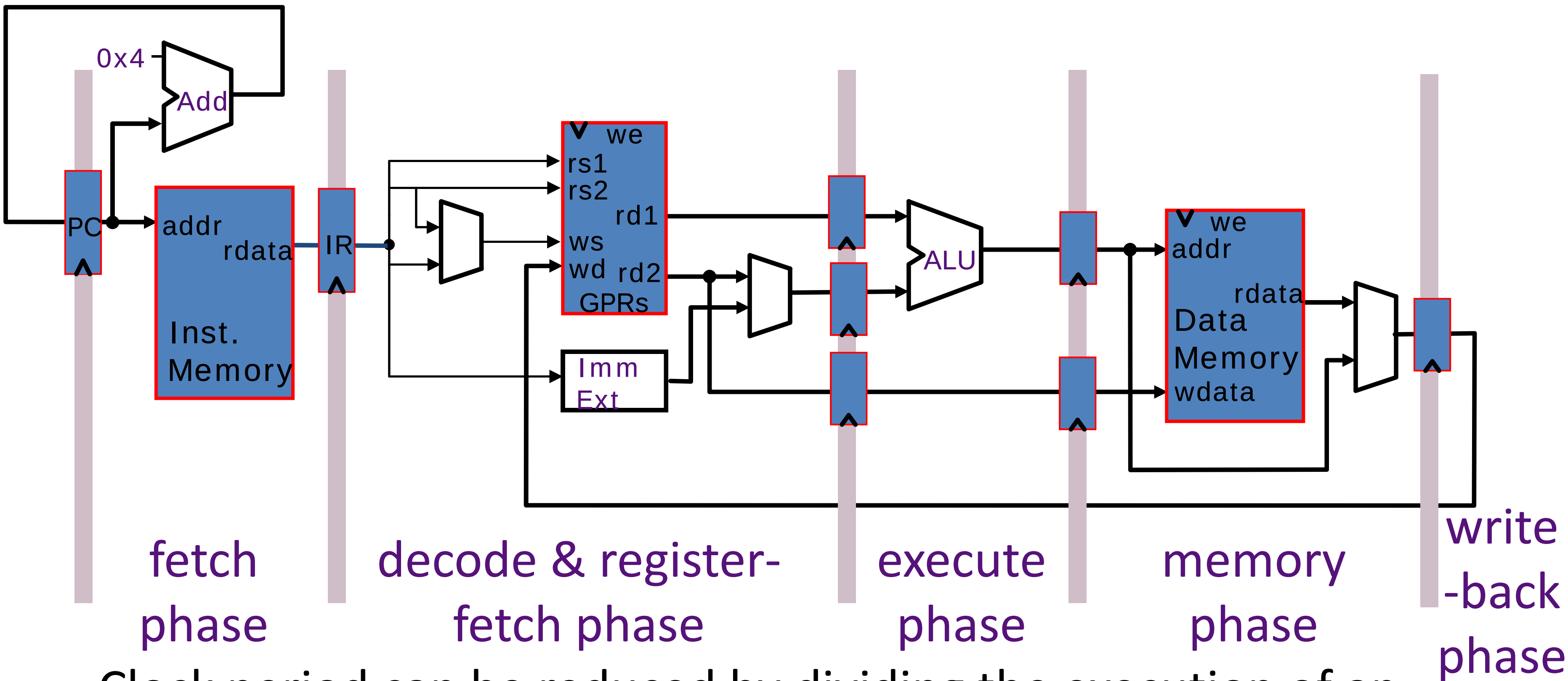
Unpipelined Datapath for MIPS



Simplified Unpipelined Datapath



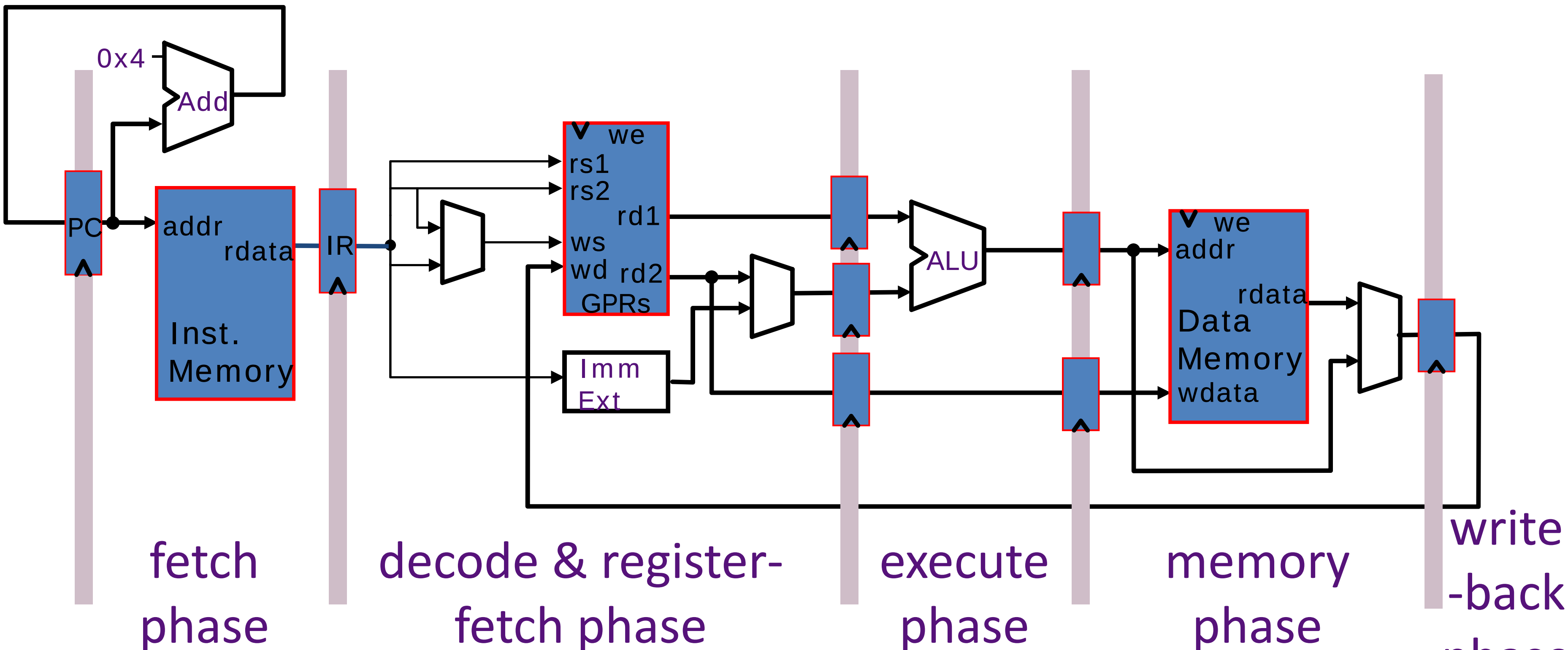
Pipelined Datapath



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

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

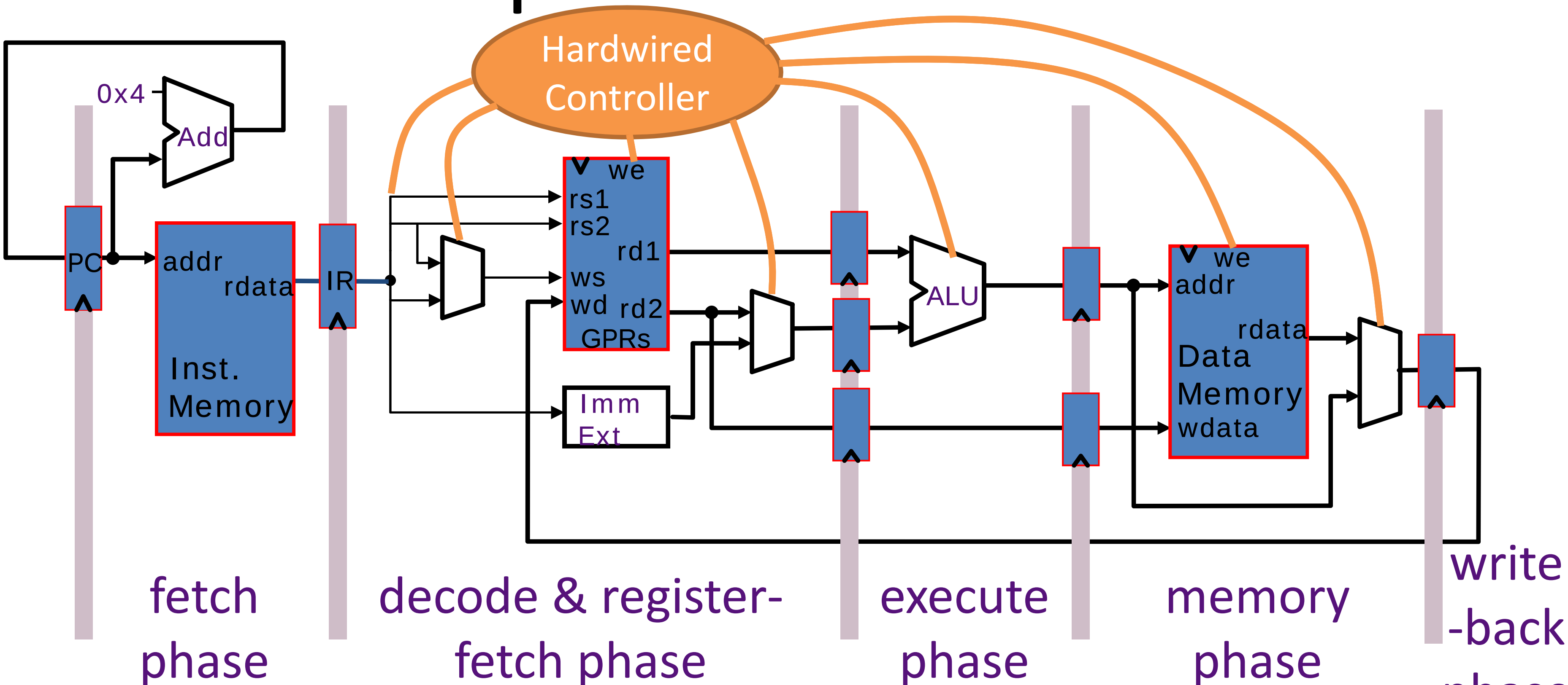
Pipelined Control



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

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

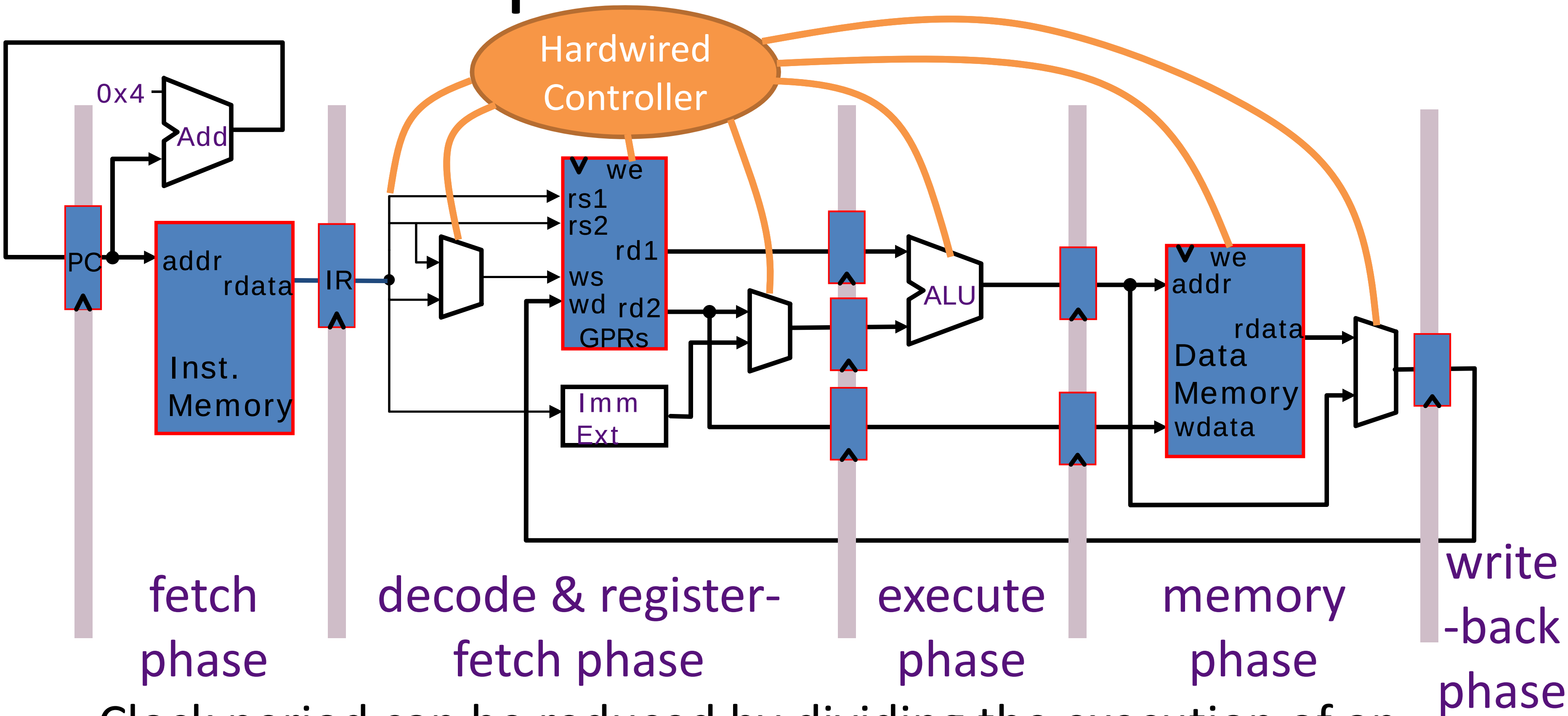
Pipelined Control



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

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

Pipelined Control

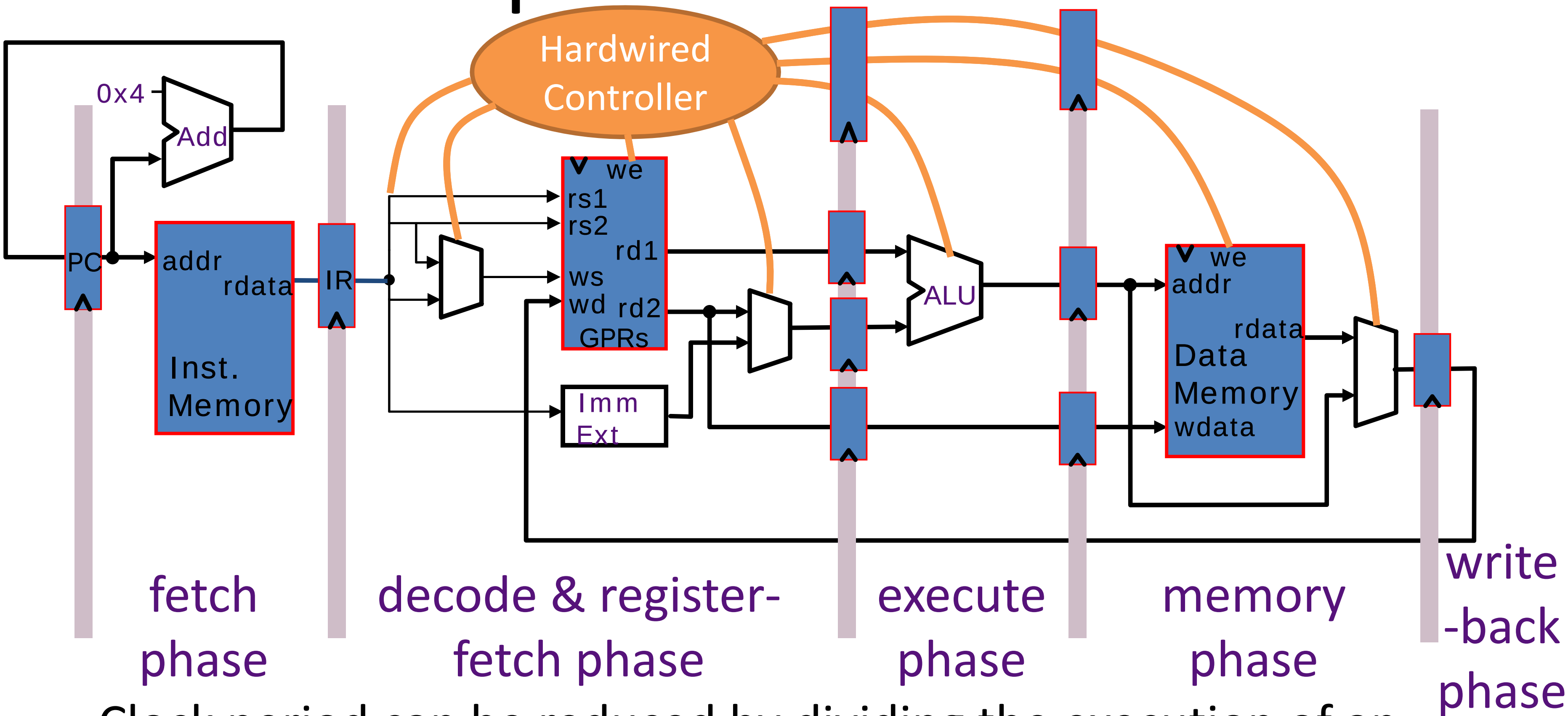


Clock period can be reduced by dividing the execution of an instruction into multiple cycles

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

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

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

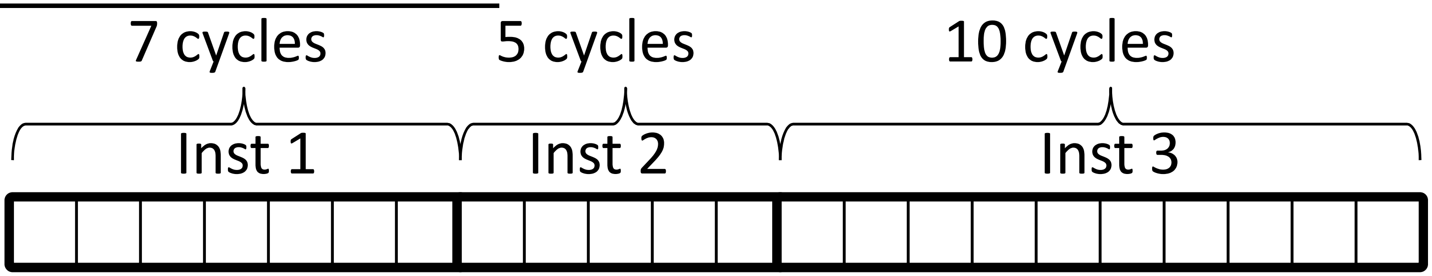
– Instructions per program depends on source code, compiler technology, and ISA
– Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
– Time per cycle depends upon the microarchitecture and the base technology

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

Time

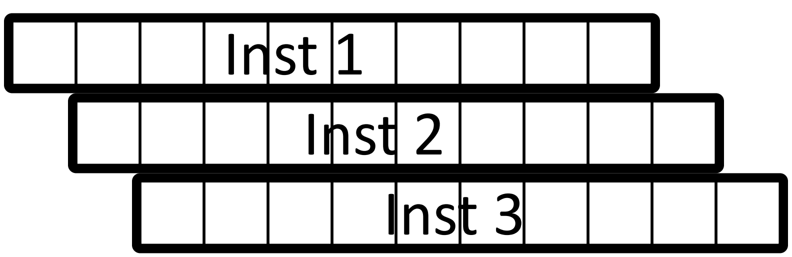


3 instructions, 22 cycles, CPI=7.33

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!)

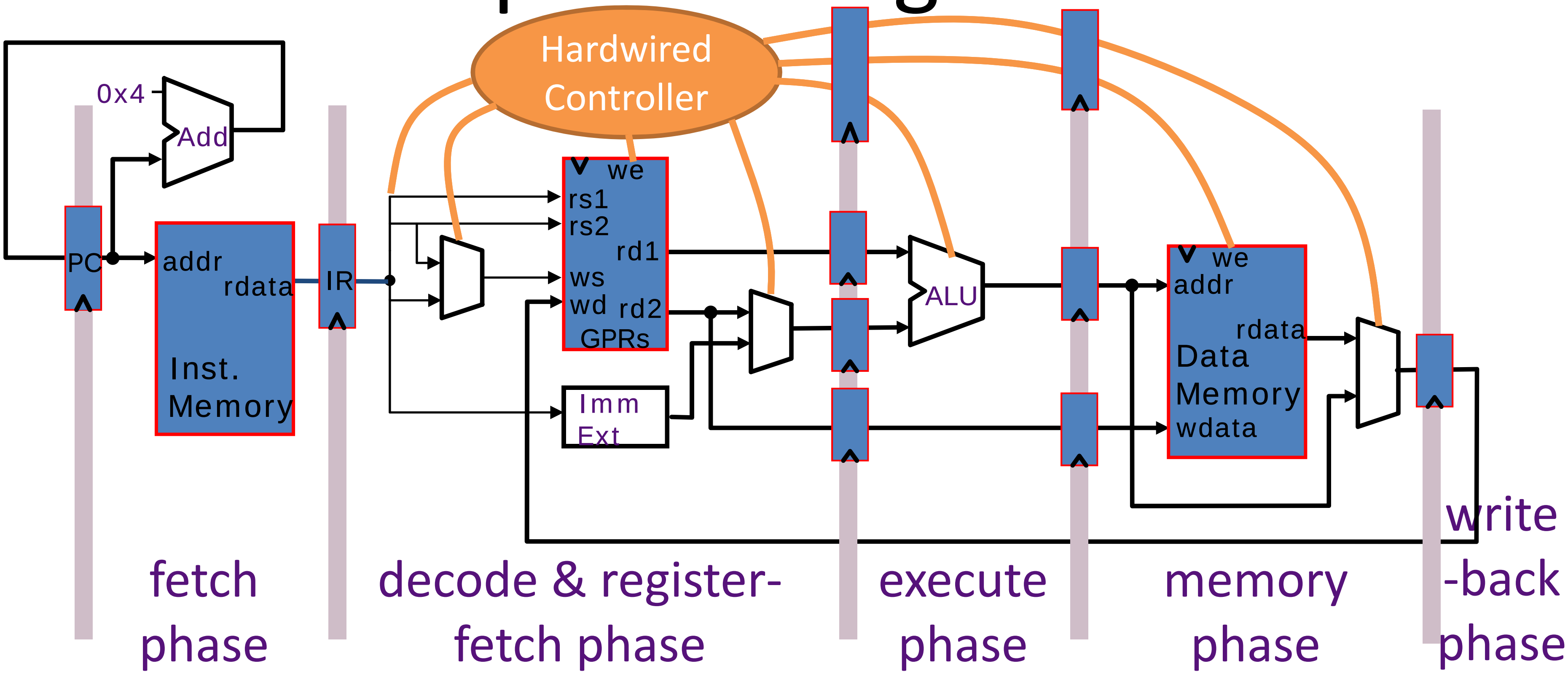
Thus, the following timing assumption is reasonable

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

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

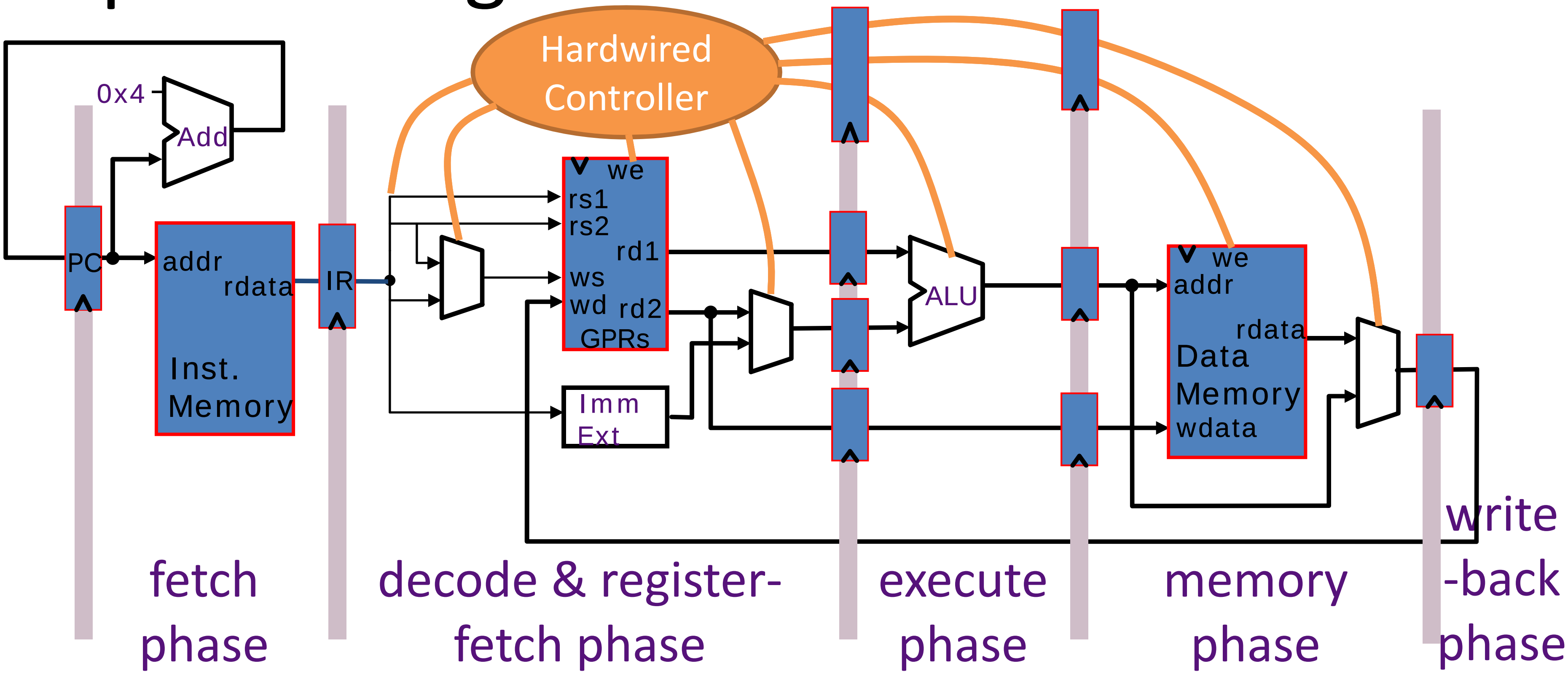
- some commercial designs have over 30 pipeline stages to do an integer add!

Pipeline Diagrams



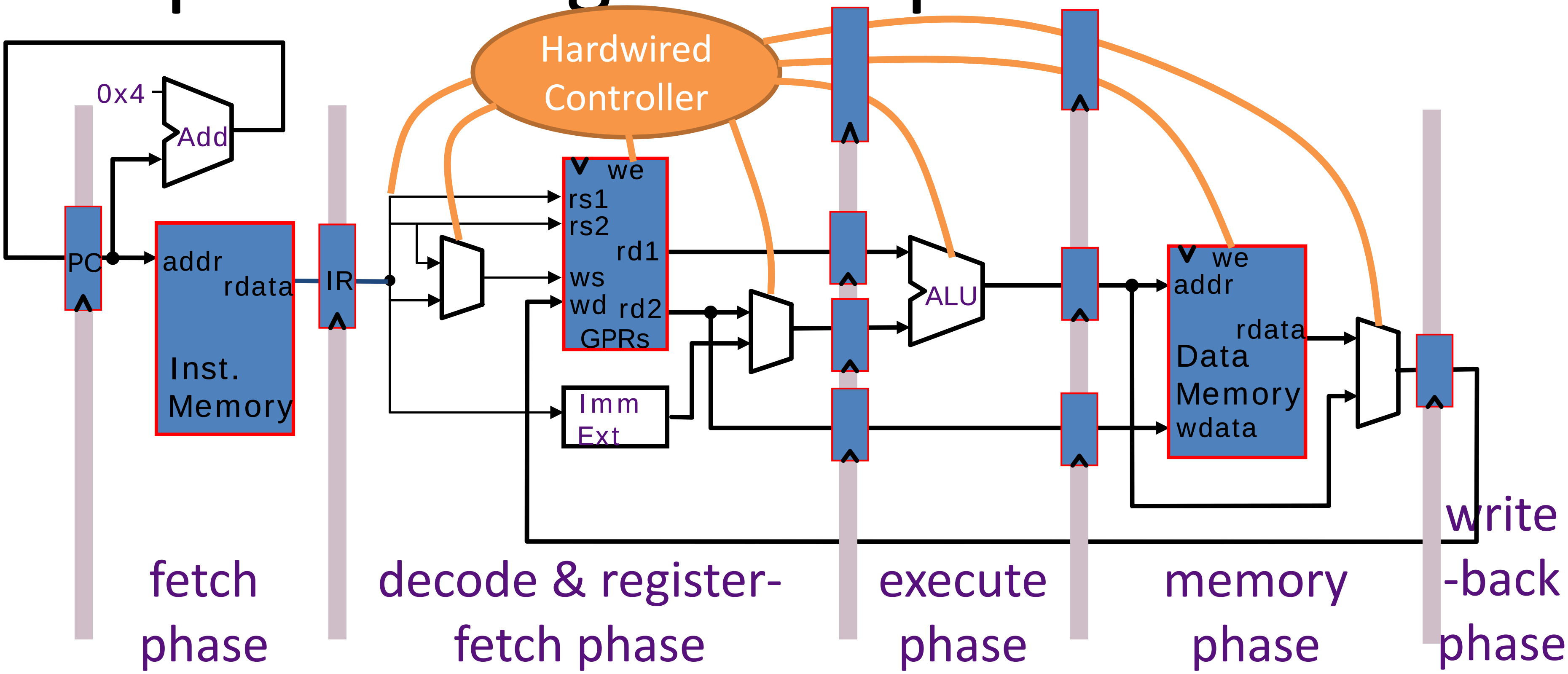
We need some way to show multiple simultaneous transactions in both space and time

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



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

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

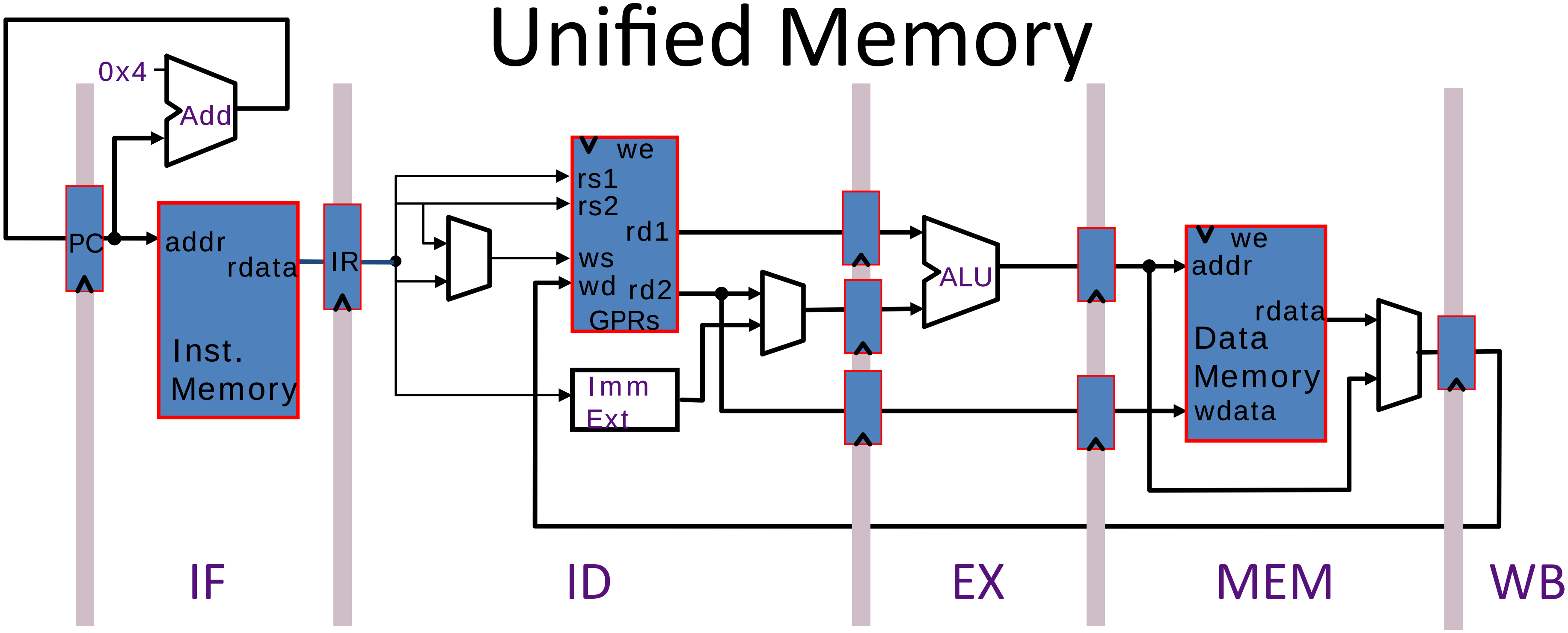
Agenda

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

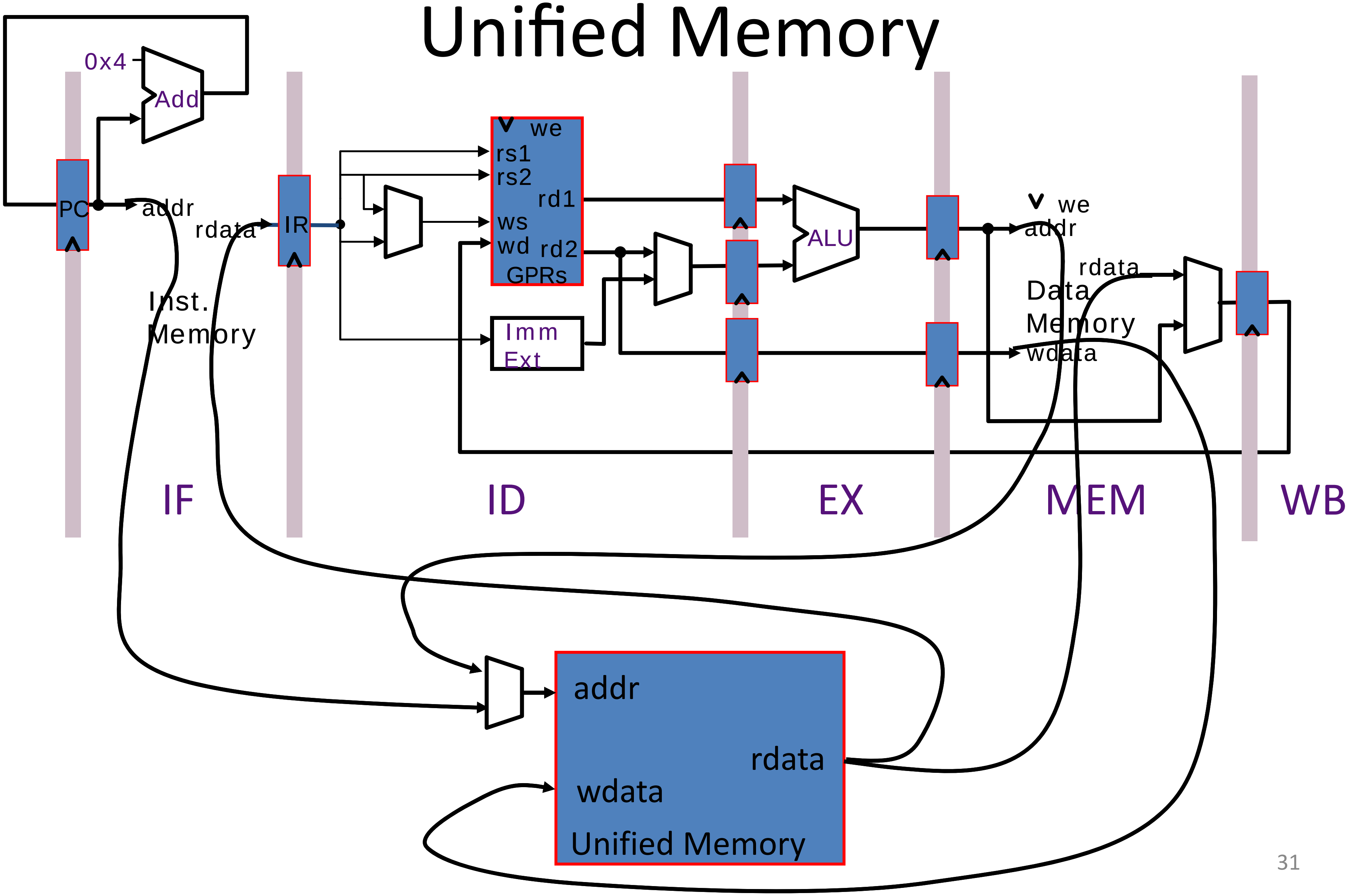
Overview of Structural Hazards

- 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

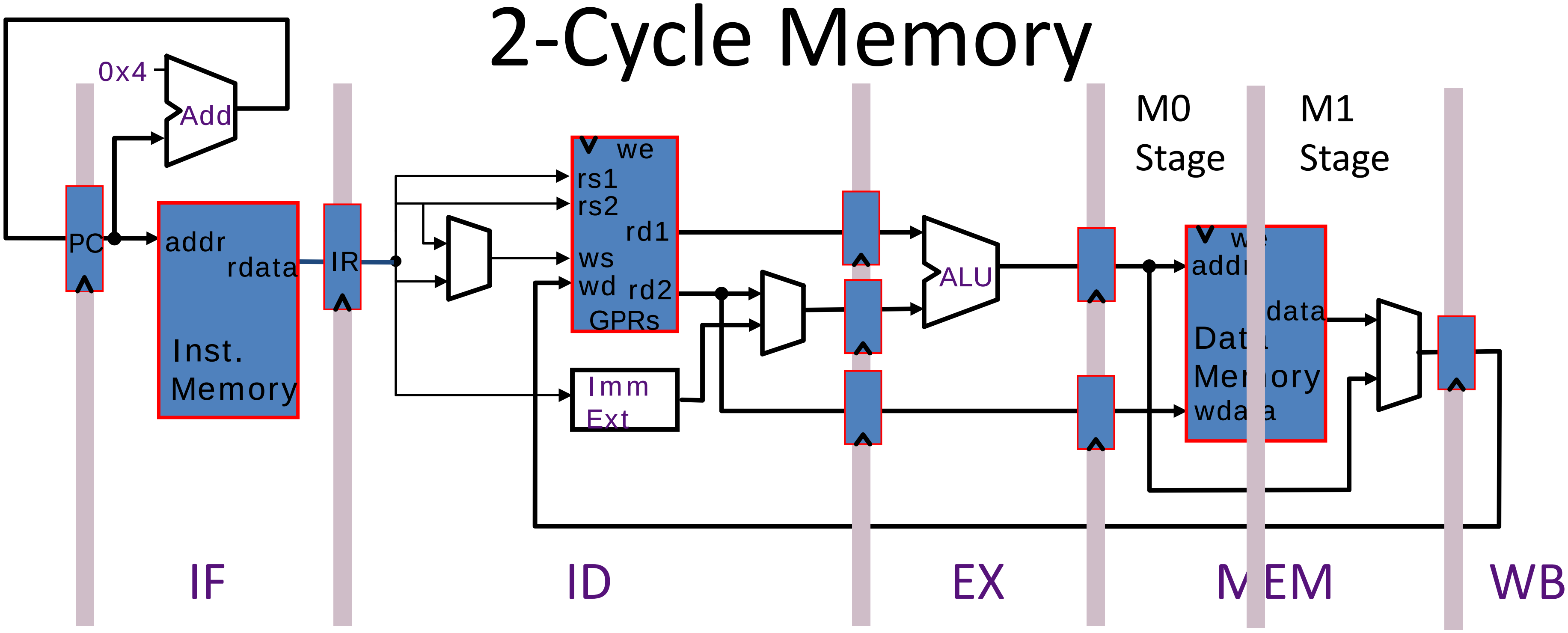
Example Structural Hazard: Unified Memory



Example Structural Hazard: Unified Memory



Example Structural Hazard: 2-Cycle Memory



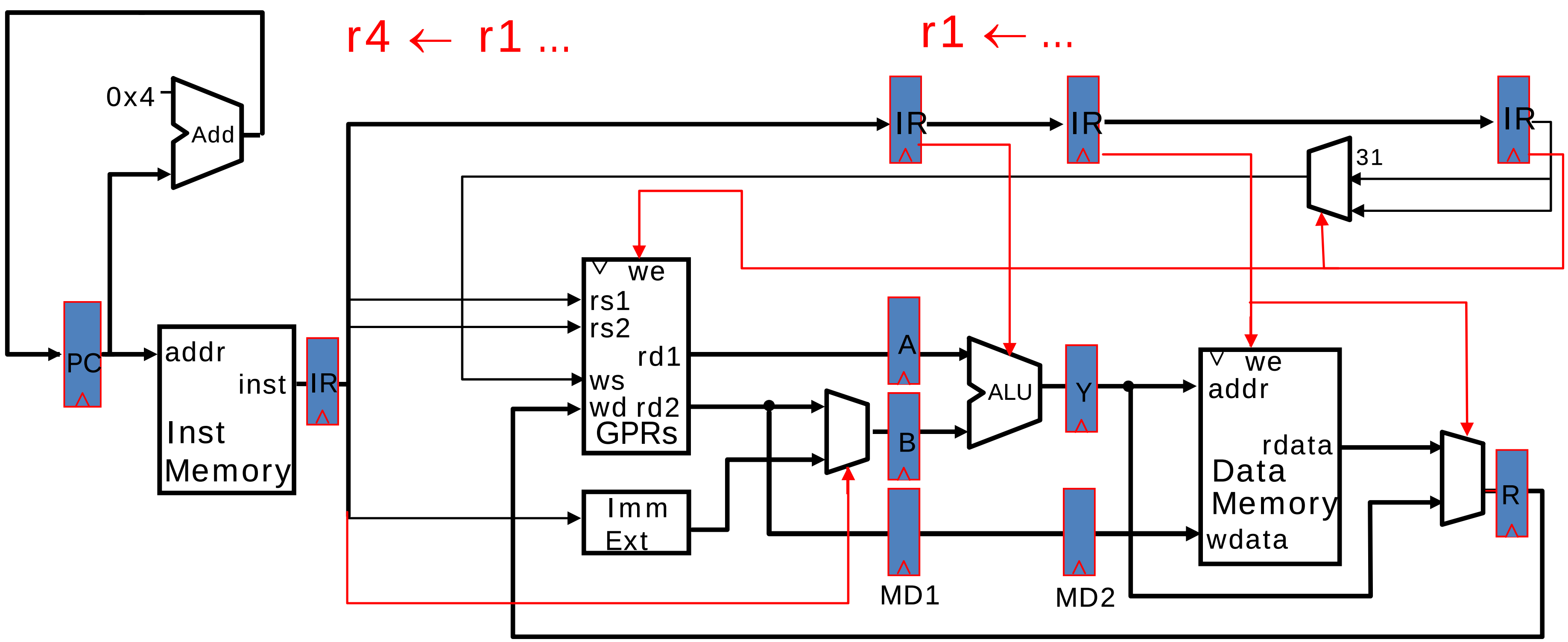
Agenda

- 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



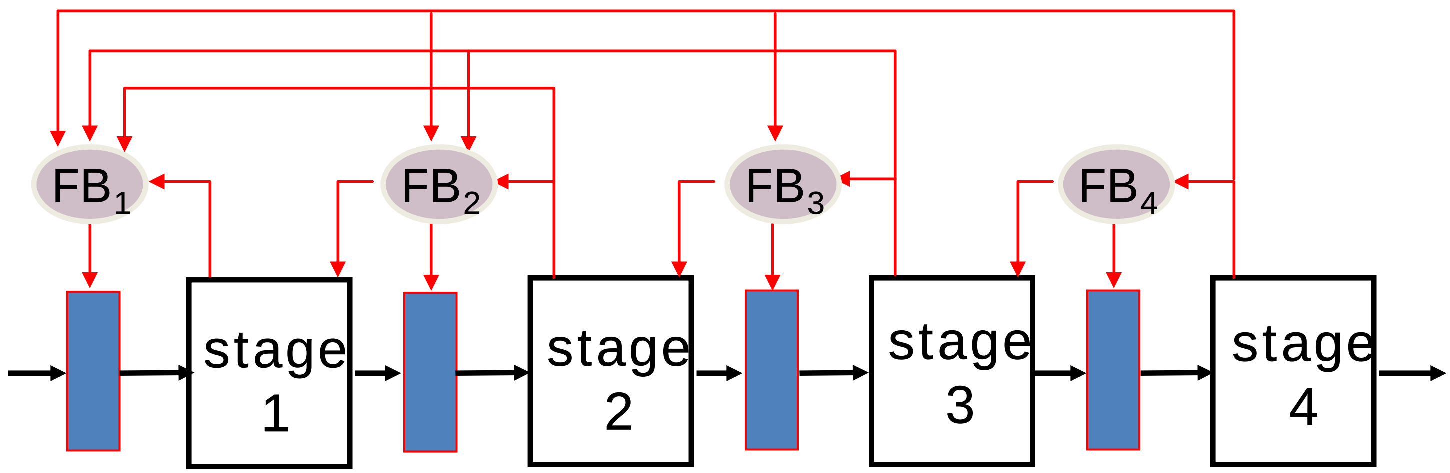
...

r1 ← r0 + 10 (ADDI R1, R0, #10)

r4 ← r1 + 17 (ADDI R4, R1, #17) *r1 is stale. Oops!*

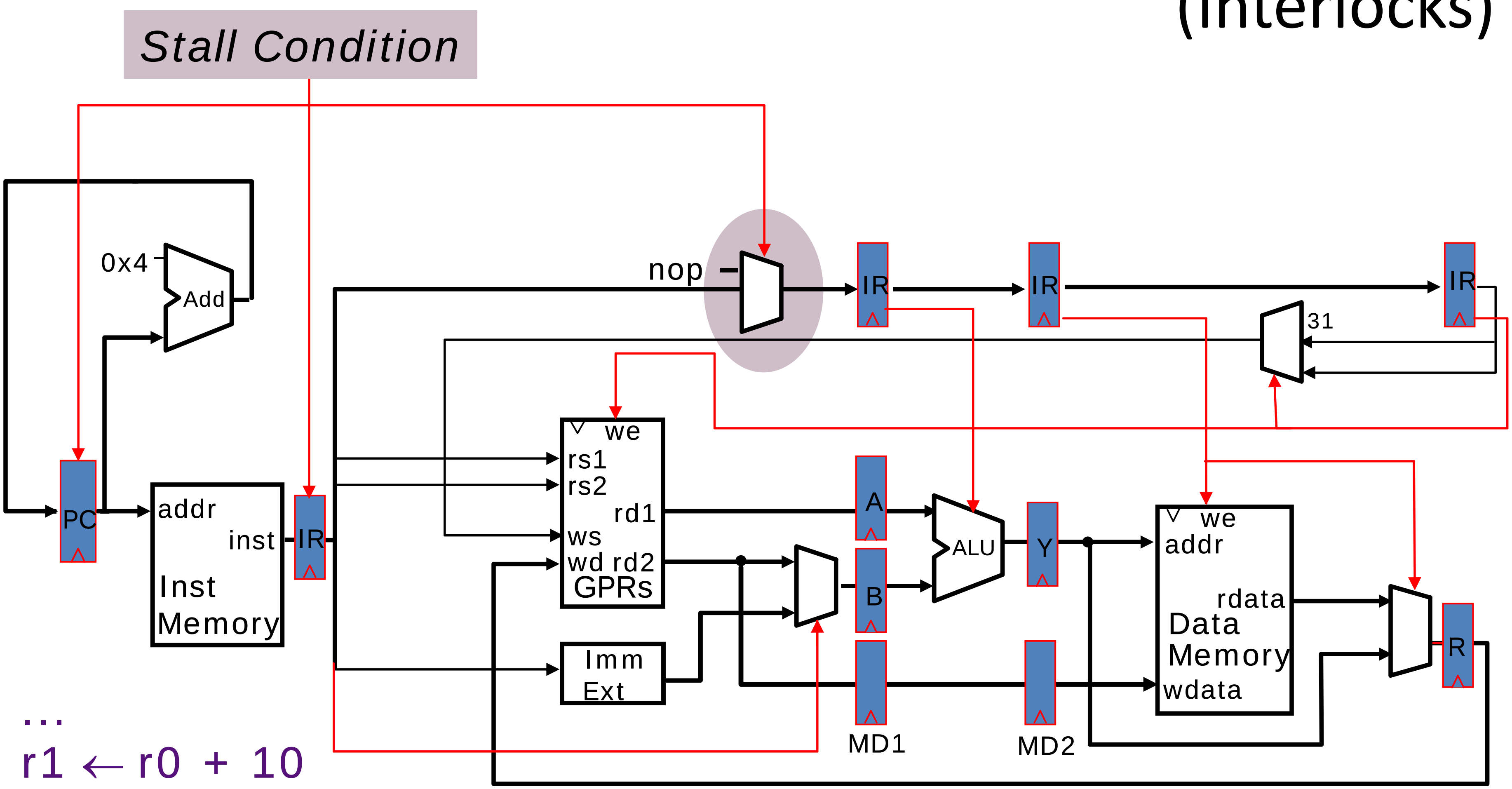
...

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)



...

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

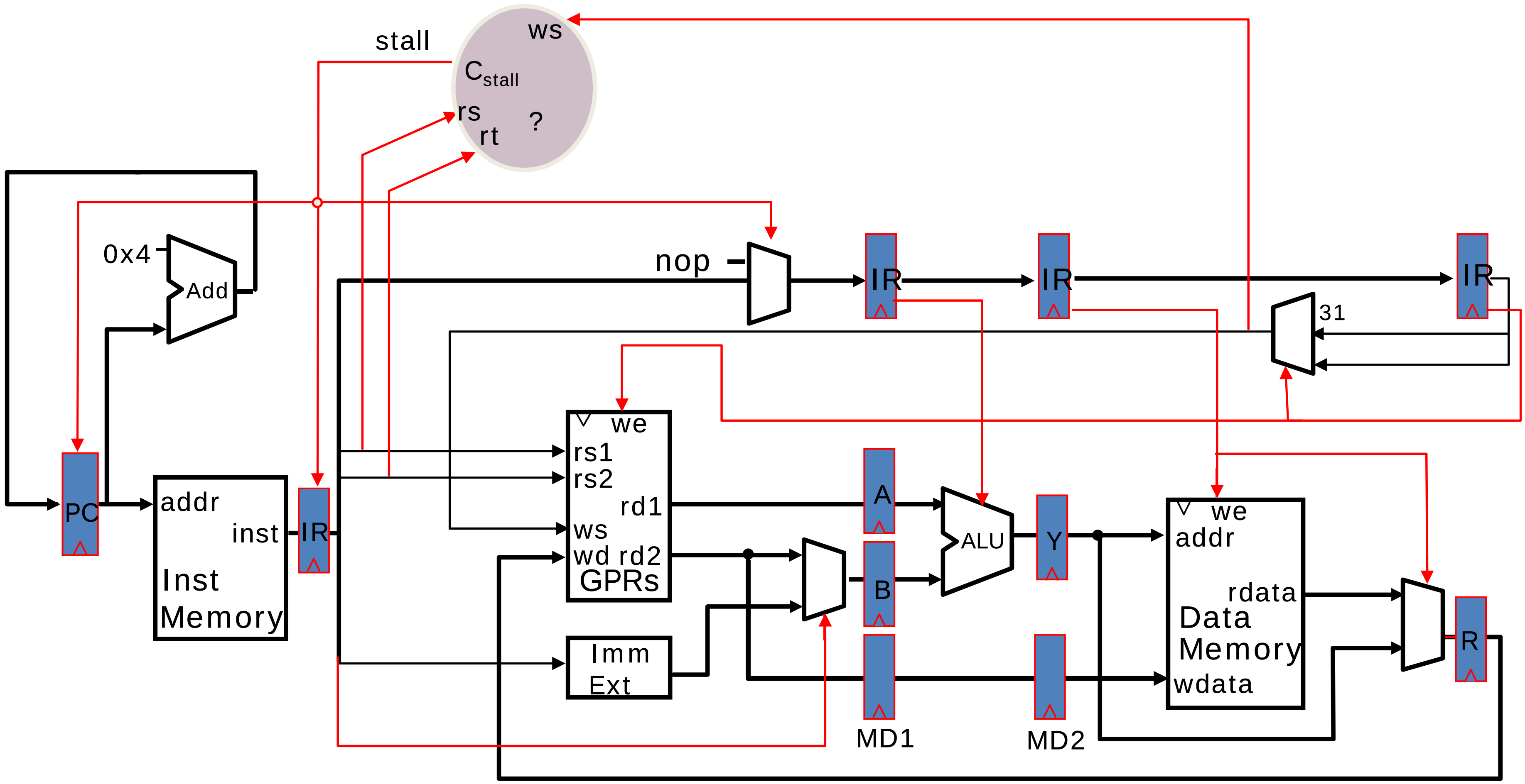
stalled stages

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

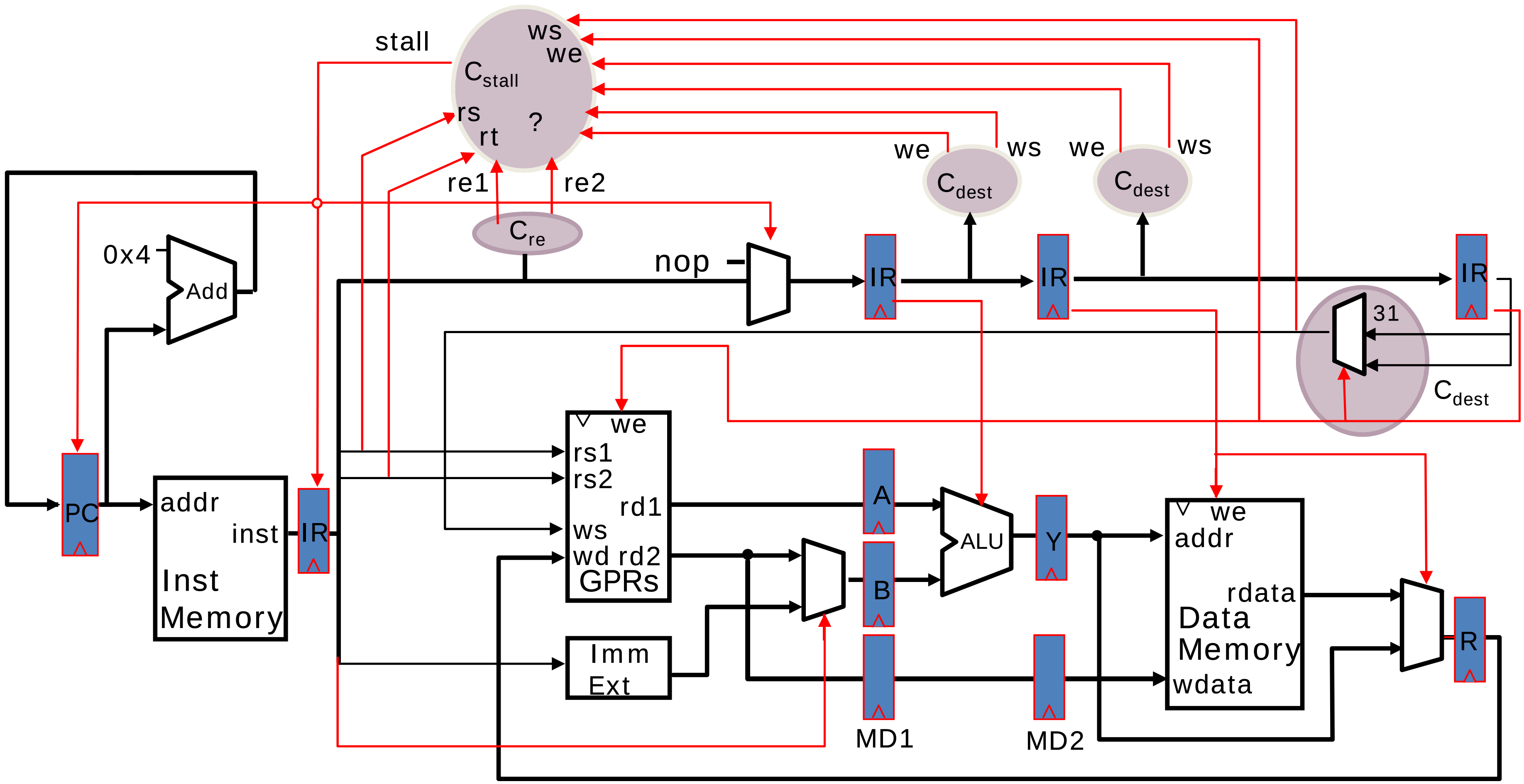
nop ⇒ *pipeline bubble*

Stall Control Logic



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

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 immediate	rs	rt
LW	rt ← M [(rs) + immediate]	rs	rt
SW	M [(rs) + immediate] ← (rt)	rs, rt	
BZ	*cond* (rs)		
	true: PC ← (PC) + immediate	rs	
	false: PC ← (PC) + 4	rs	
J	PC ← (PC) + immediate		
JAL	r31 ← (PC), PC ← (PC) + immediate		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}

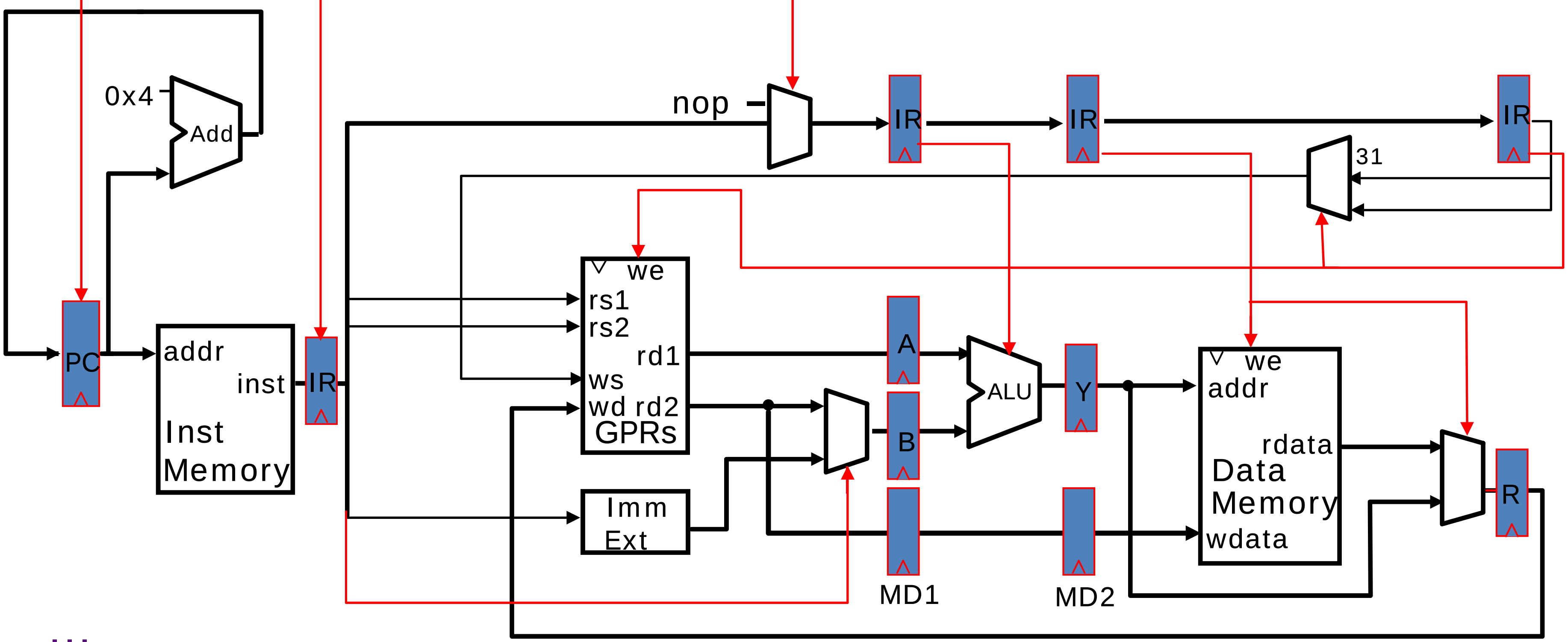
$$
\begin{aligned}
stall = \; & ((rs_D = ws_E).we_E + \\
& (rs_D = ws_M).we_M + \\
& (rs_D = ws_W).we_W) \;.\; re1_D \quad + \\
& ((rt_D = ws_E).we_E + \\
& (rt_D = ws_M).we_M + \\
& (rt_D = ws_W).we_W) \;.\; re2_D
\end{aligned}
$$

This is not the full story !

Hazards due to Loads & Stores

Stall Condition

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



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

Is there any possible data hazard in this instruction sequence?

Data Hazards Due to Loads and Store

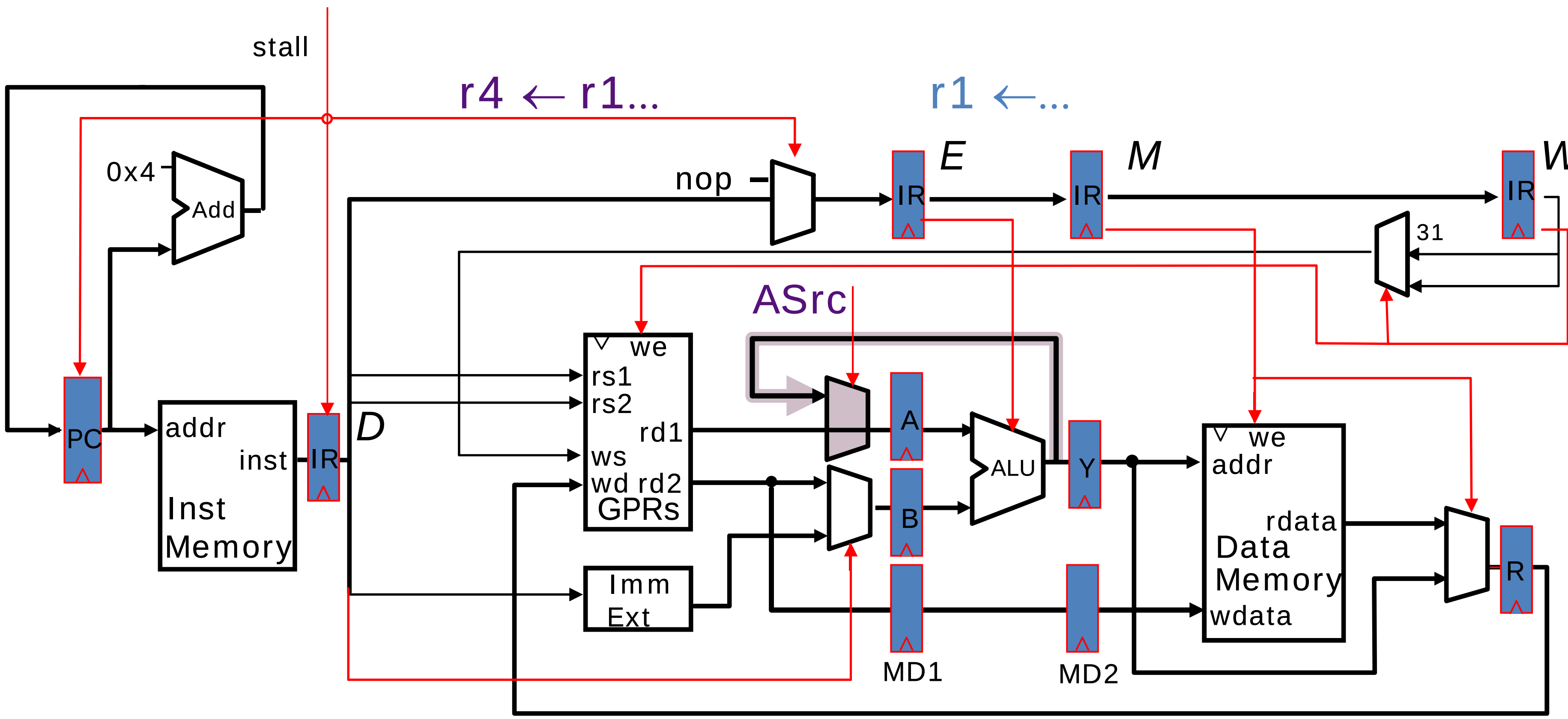
- Example instruction sequence
  - Mem[Regs[r1] + 7] <- Regs[r2]
  - Regs[r4] <- Mem[Regs[r3] + 5]

- What if Regs[r1]+7 == 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?

...

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

Deriving the Bypass Signal

time

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
(I_1) $r1 \leftarrow (r0) + 10$	IF_1	ID_1	EX_1	MA_1	WB_1							
(I_2) $r4 \leftarrow (r1) + 17$		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

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) $r1 \leftarrow (r0) + 10$	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) $r4 \leftarrow (r1) + 17$		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

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 ⇒ rd
- ALUi, LW ⇒ rt
- JAL, JALR ⇒ R31

we = *Case* opcode
- ALU, ALUi, LW ⇒ (ws ≠ 0)
- JAL, JALR ⇒ on
- ... ⇒ 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

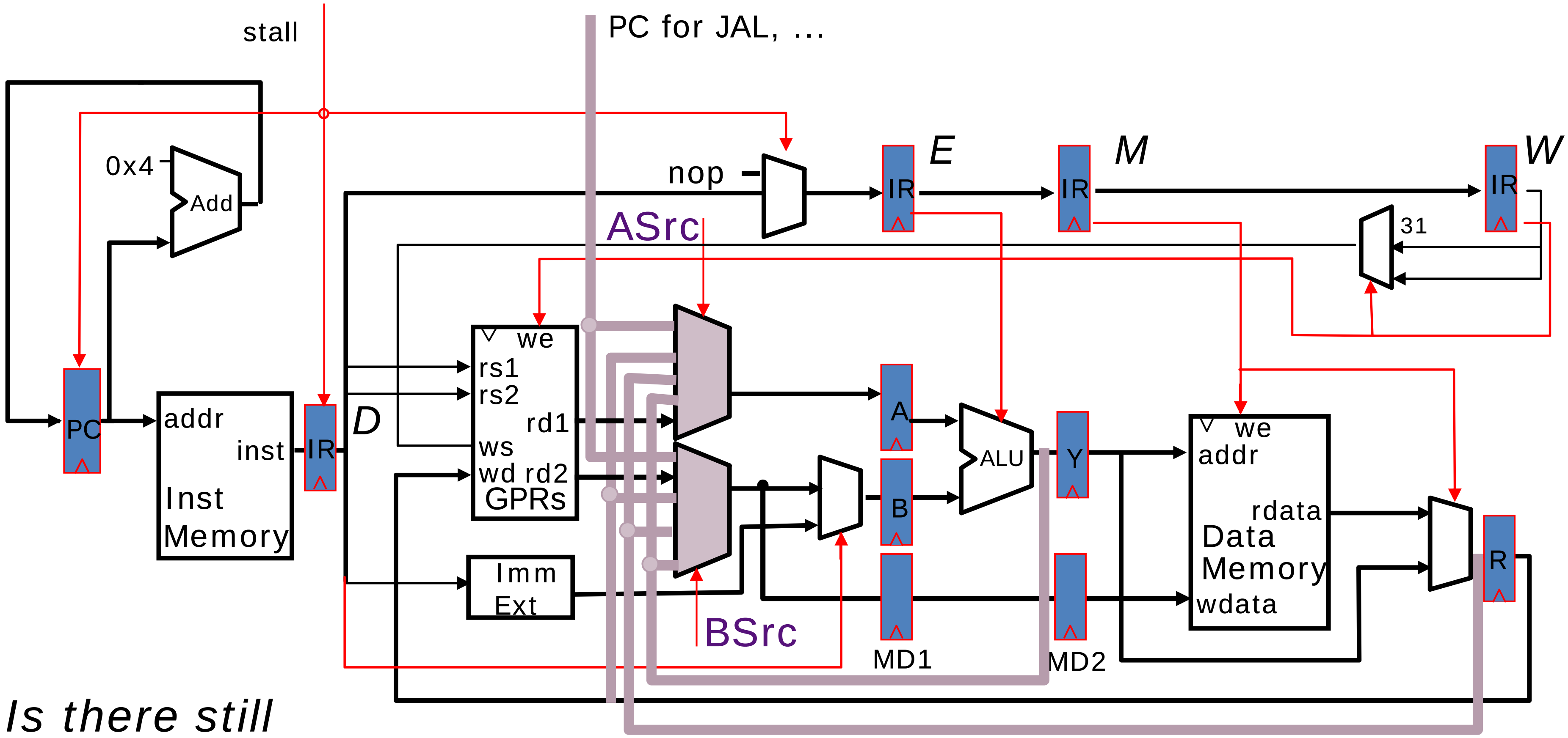
$$
\begin{aligned}
&\text{we-bypass}_E = \textit{Case}\ \text{opcode}_E \\
&\quad \text{ALU, ALUi} \quad \Rightarrow (ws \neq 0) \\
&\quad \ldots \quad \Rightarrow \text{off}
\end{aligned}
$$

$$
\begin{aligned}
&\text{we-stall}_E = \textit{Case}\ \text{opcode}_E \\
&\quad \text{LW} \quad \Rightarrow (ws \neq 0) \\
&\quad \text{JAL, JALR} \quad \Rightarrow \text{on} \\
&\quad \ldots \quad \Rightarrow \text{off}
\end{aligned}
$$

$$\text{ASrc} = (rs_D = ws_E).\text{we-bypass}_E \,.\, re1_D$$

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

Fully Bypassed Datapath



Is there still a need for the stall signal ?

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

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)

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) $r1 \leftarrow (r0) + 10$	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) $r3 \leftarrow (r2) + 17$		IF_2	IF_2	ID_2	EX_2	MA_2	WB_2		
(I_3)				IF_3	IF_3	ID_3	EX_3	MA_3	WB_3
(I_4)						IF_4	IF_4	ID_4	EX_4 MA_4 WB_4

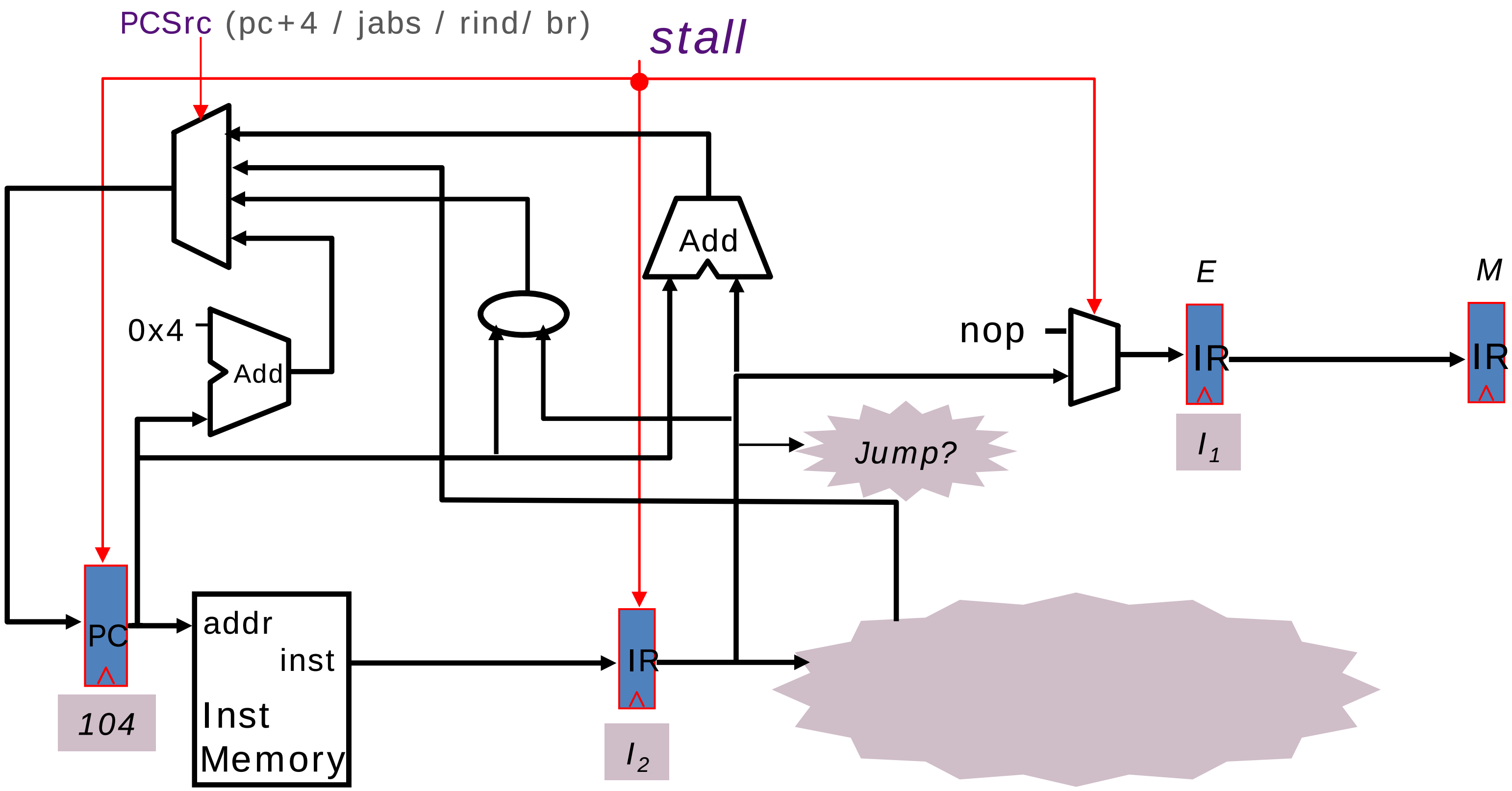
Resource Usage

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

CPI = 2!

nop $\Rightarrow$ *pipeline bubble*

Speculate next address is PC+4

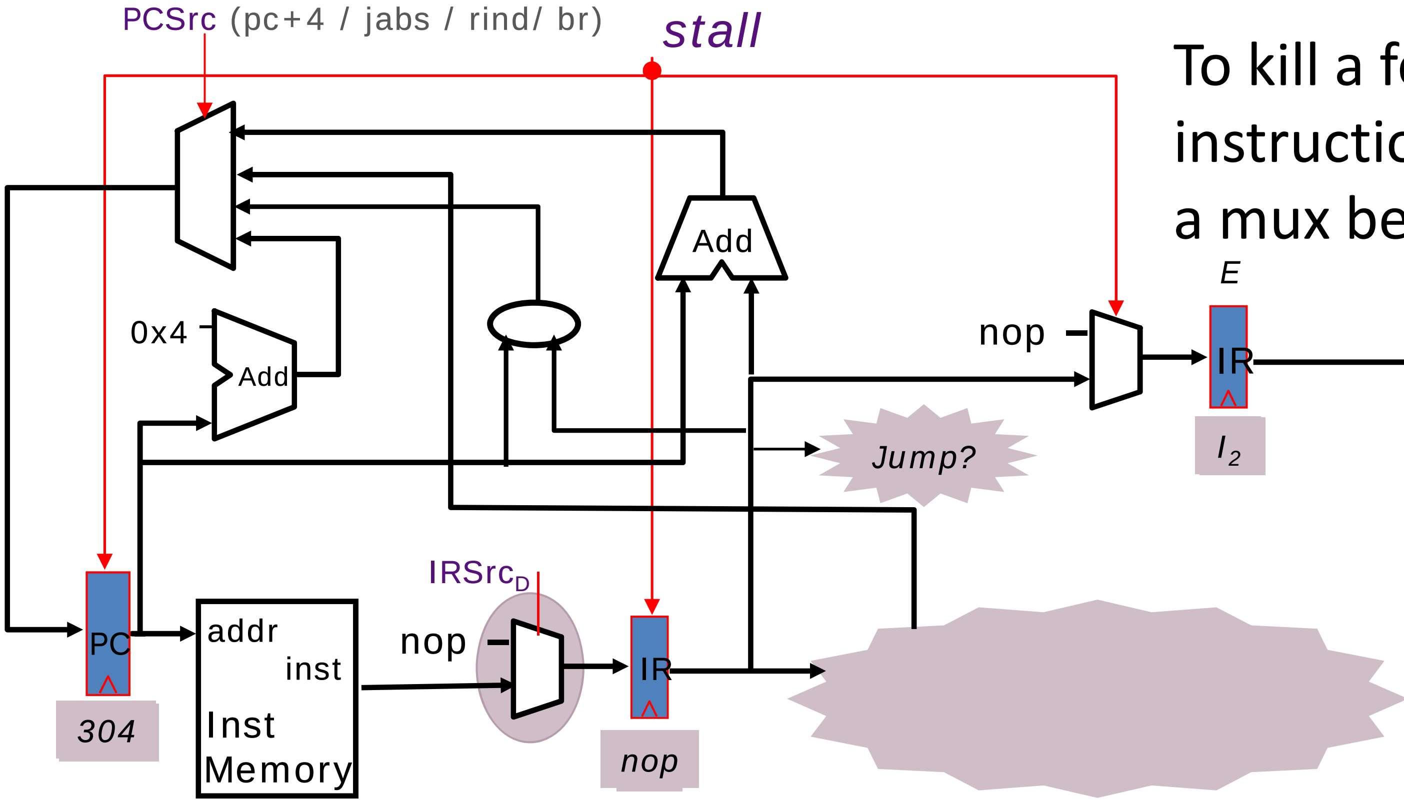


I_1	096	ADD	
I_2	100	J 304	
I_3	~~104~~	~~ADD~~	kill
I_4	304	ADD	

A jump instruction kills (not stalls) the following instruction

How?

Pipelining Jumps



To kill a fetched instruction -- Insert a mux before IR

Any interaction between stall and jump?

I_1	096	ADD
I_2	100	J 304
I_3	~~104~~	~~ADD~~
I_4	304	ADD

kill

IRSrc$_D$ = *Case* opcode$_D$
J, JAL $\Rightarrow$ nop
... $\Rightarrow$ IM

Jump Pipeline Diagrams

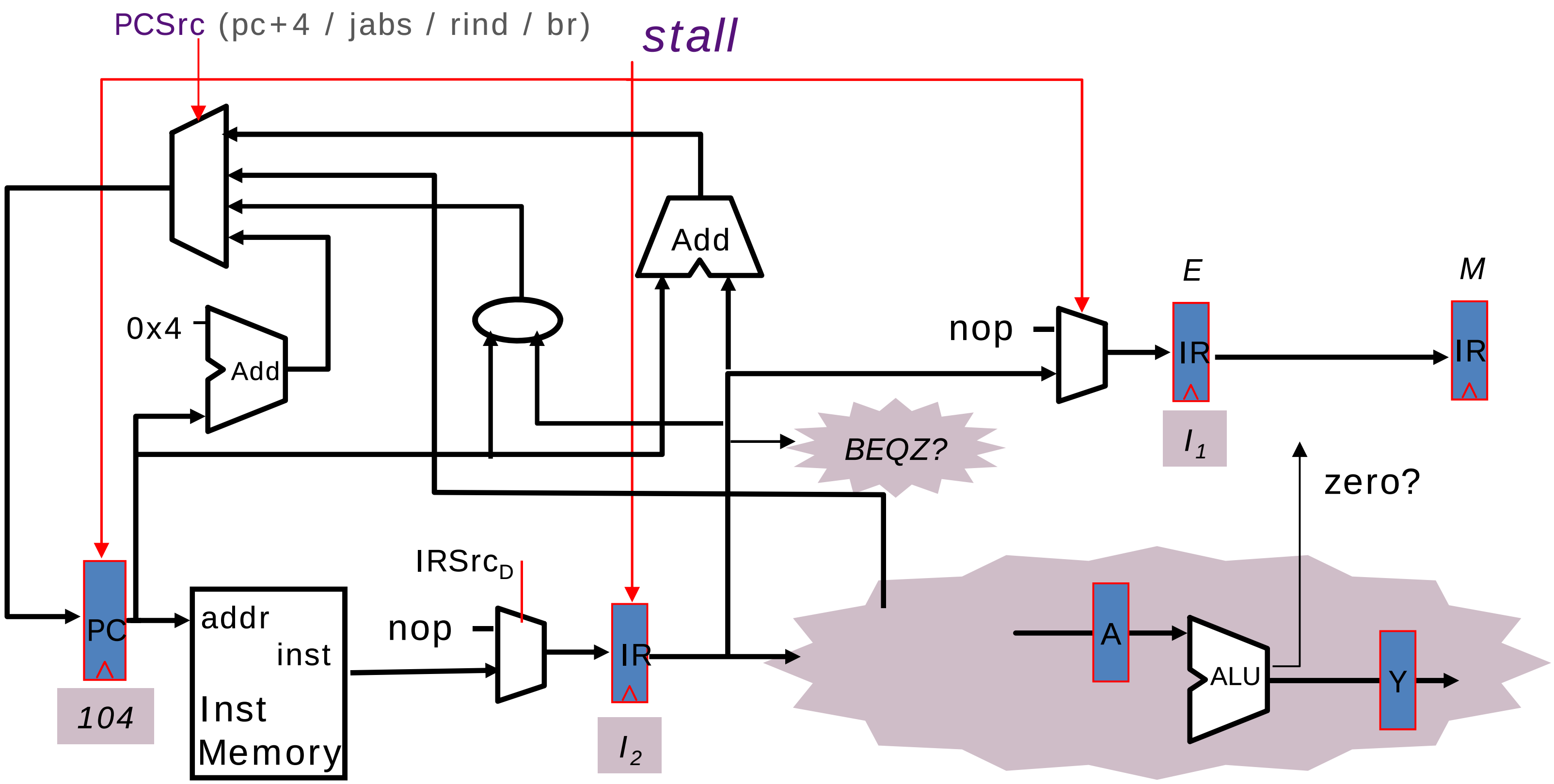
	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: J 304		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	nop	nop	nop	nop		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

Resource Usage

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

nop $\Rightarrow$ *pipeline bubble*

Pipelining Conditional Branches

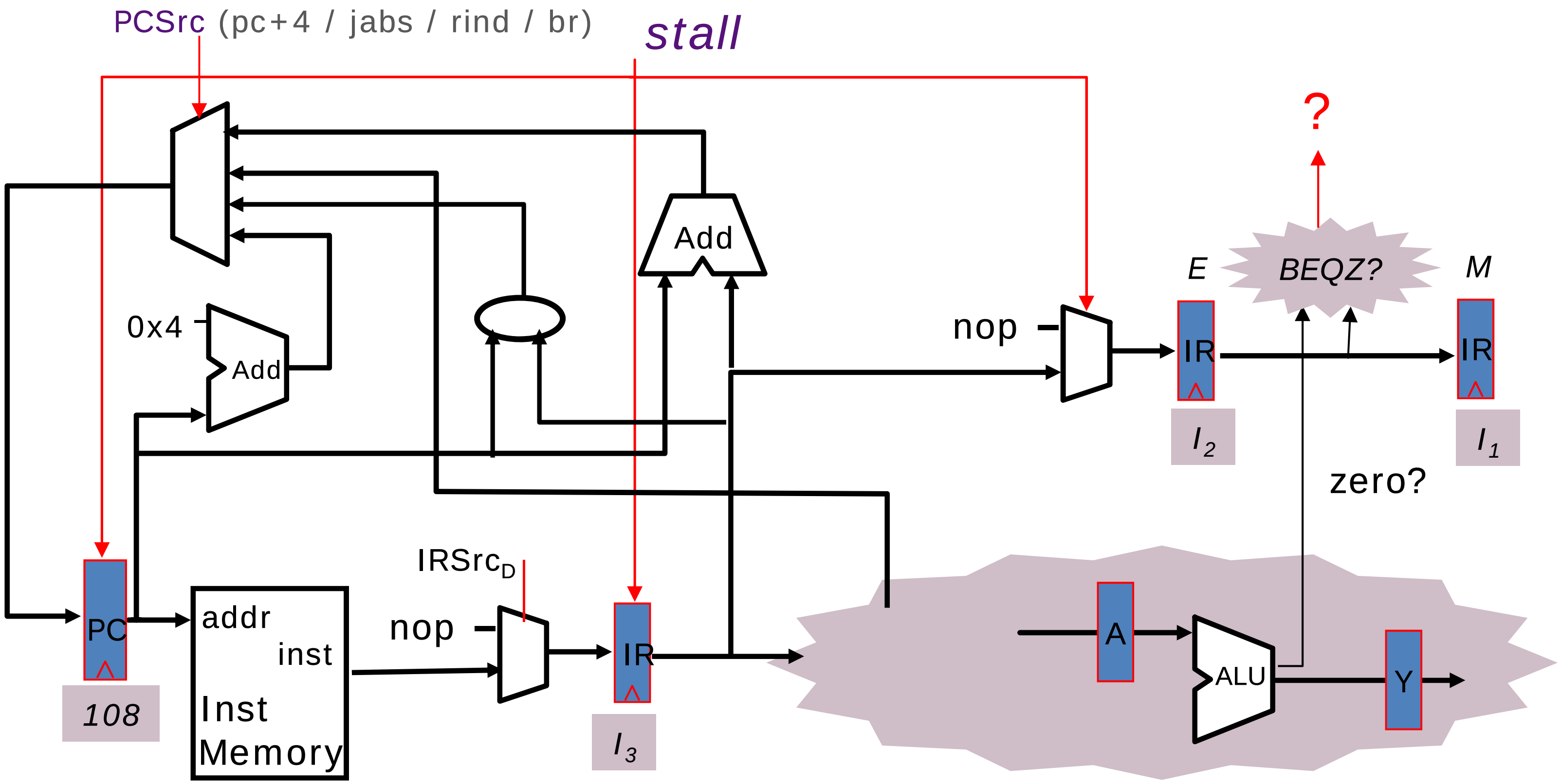


I_1	096	ADD
I_2	100	BEQZ r1 +200
I_3	104	ADD
	108	...
I_4	304	ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Pipelining Conditional Branches



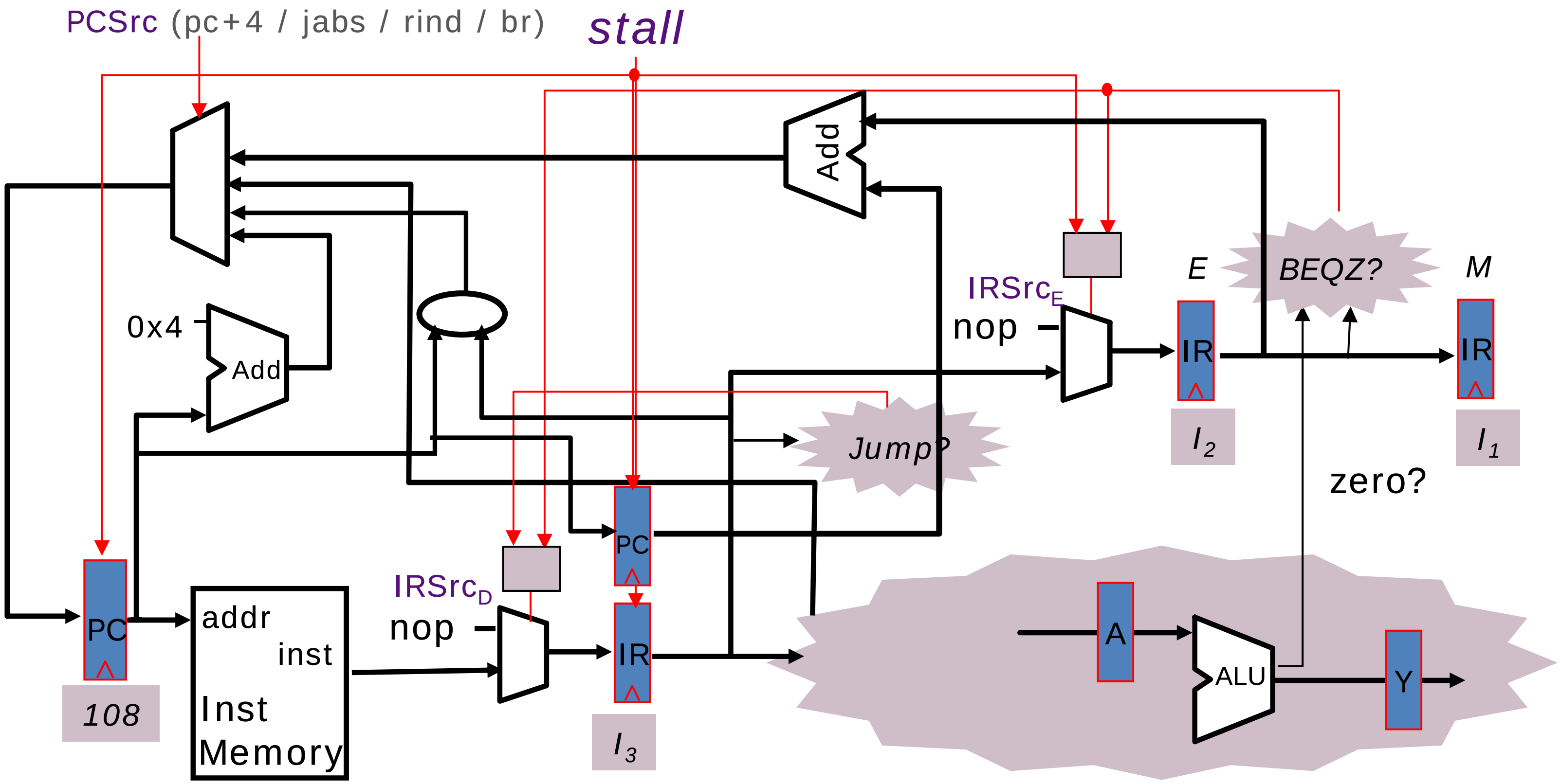
I_1	096	ADD
I_2	100	BEQZ r1 +200
I_3	104	ADD
	108	...
I_4	304	ADD

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*

Pipelining Conditional Branches



I_1	096	ADD
I_2	100	BEQZ r1 +200
I_3	104	ADD
	108	...
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

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

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

Instruction at the decode stage is invalid

Control Equations for PC and IR Muxes

$PCSrc = \textit{Case}\ opcode_E$
BEQZ.z, BNEZ.!z ⇒ br
... ⇒
$\textit{Case}\ opcode_D$
J, JAL ⇒ jabs
JR, JALR ⇒ rind
... ⇒ pc+4

$IRSrc_D = \textit{Case}\ opcode_E$
BEQZ.z, BNEZ.!z ⇒ nop
... ⇒
$\textit{Case}\ opcode_D$
J, JAL, JR, JALR ⇒ nop
... ⇒ IM

$IRSrc_E = \textit{Case}\ opcode_E$
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

Branch Pipeline Diagrams

(resolved in execute stage)

	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: BEQZ +200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	ID_3	nop	nop	nop		
(I_4) 108:				IF_4	nop	nop	nop	nop	
(I_5) 304: ADD					IF_5	ID_5	EX_5	MA_5	WB_5

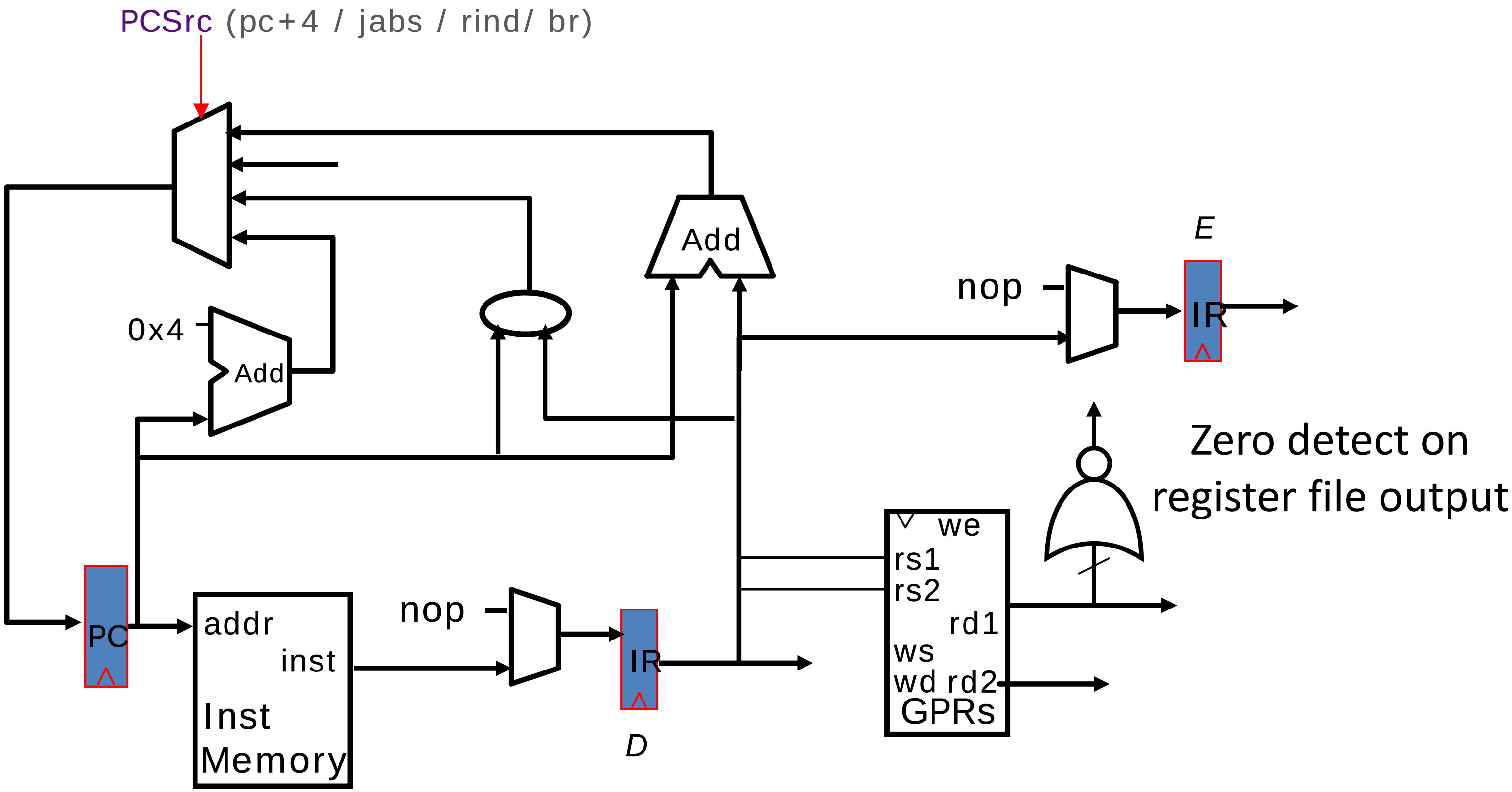
		time								
		t0	t1	t2	t3	t4	t5	t6	t7	. . . .
Resource Usage	IF	I_1	I_2	I_3	I_4	I_5				
	ID		I_1	I_2	I_3	nop	I_5			
	EX			I_1	I_2	nop	nop	I_5		
	MA				I_1	I_2	nop	nop	I_5	
	WB					I_1	I_2	nop	nop	I_5

nop $\Rightarrow$ *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

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.

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

Delay slot instruction executed regardless of branch outcome

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

Branch Pipeline Diagrams

(branch delay slot)

	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: BEQZ +200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4) 304: ADD				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	I_3	I_4				
EX			I_1	I_2	I_3	I_4			
MA				I_1	I_2	I_3	I_4		
WB					I_1	I_2	I_3	I_4	

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

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)
  - Christopher Batten (Cornell)

- MIT material derived from course 6.823
- UCB material derived from course CS252 & CS152
- Cornell material derived from course ECE 4750