CS 152 Computer Architecture and Engineering

Lecture 4 - Pipelining

Dr. George Michelogiannakis
EECS, University of California at Berkeley
CRD, Lawrence Berkeley National Laboratory

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

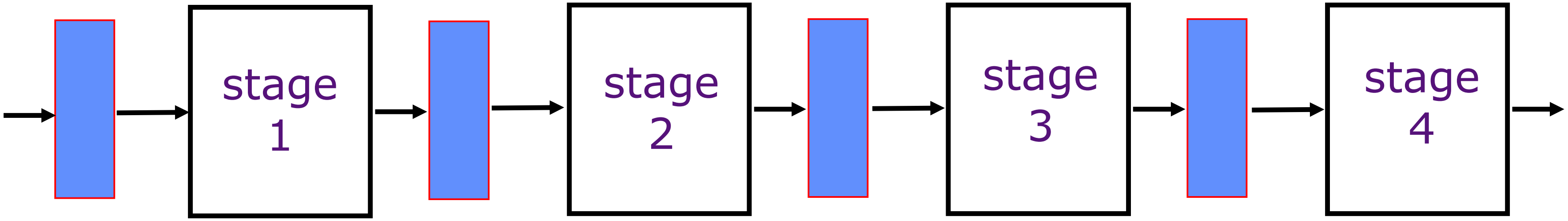
Last time in Lecture 3

- 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
- Load-Store RISC ISAs designed for efficient pipelined implementations
  - Very similar to vertical microcode
  - Inspired by earlier Cray machines (more on these later)
- Iron Law explains architecture design space
  - Trade instructions/program, cycles/instruction, and time/cycle

Question of the Day

- Why a five stage pipeline?

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!

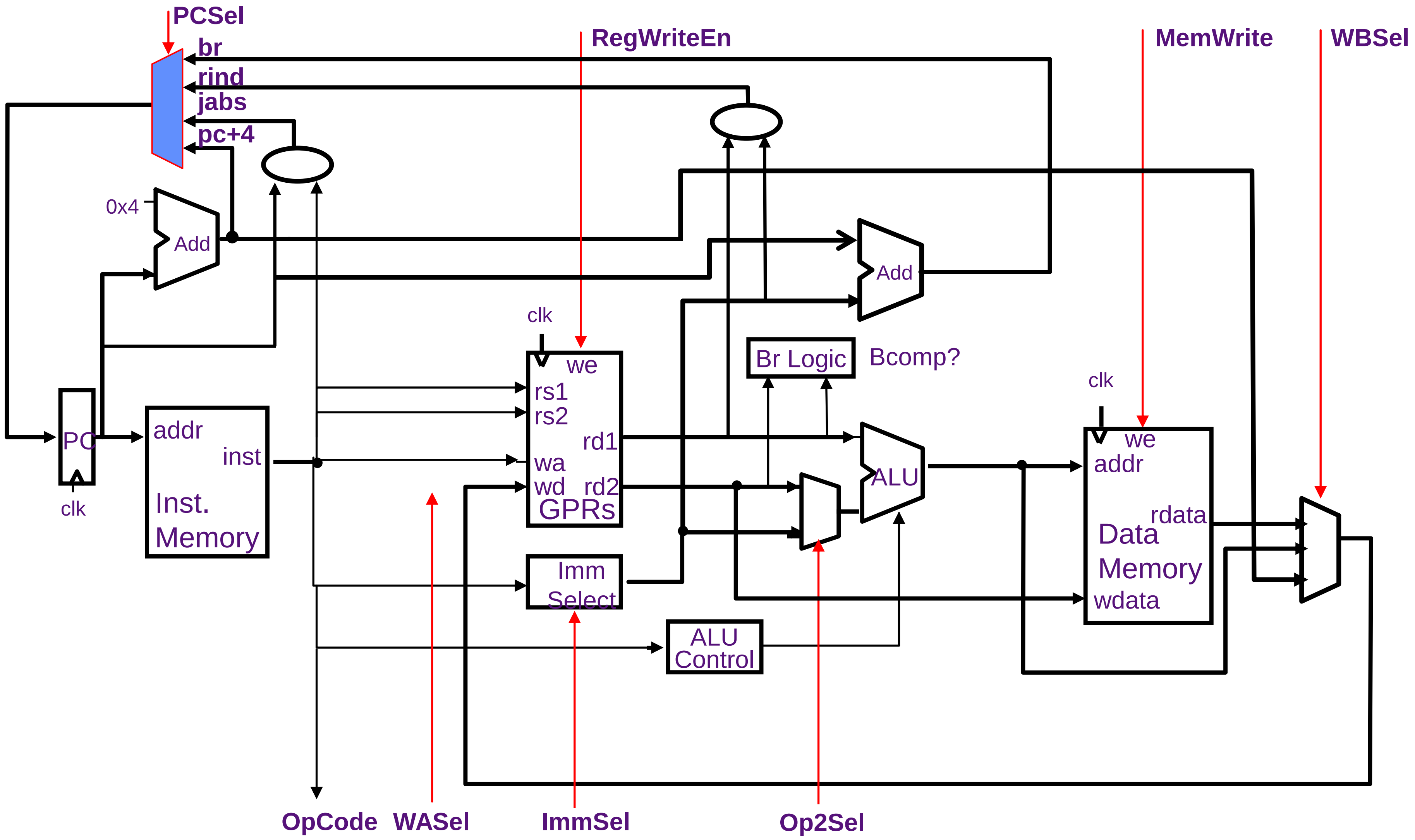
Pipelined RISC-V

- To pipeline RISC-V:

- First build RISC-V without pipelining with CPI=1

- Next, add pipeline registers to reduce cycle time while maintaining CPI=1

Lecture 3: Unpipelined Datapath for RISC-V



Lecture 3: Hardwired Control Table

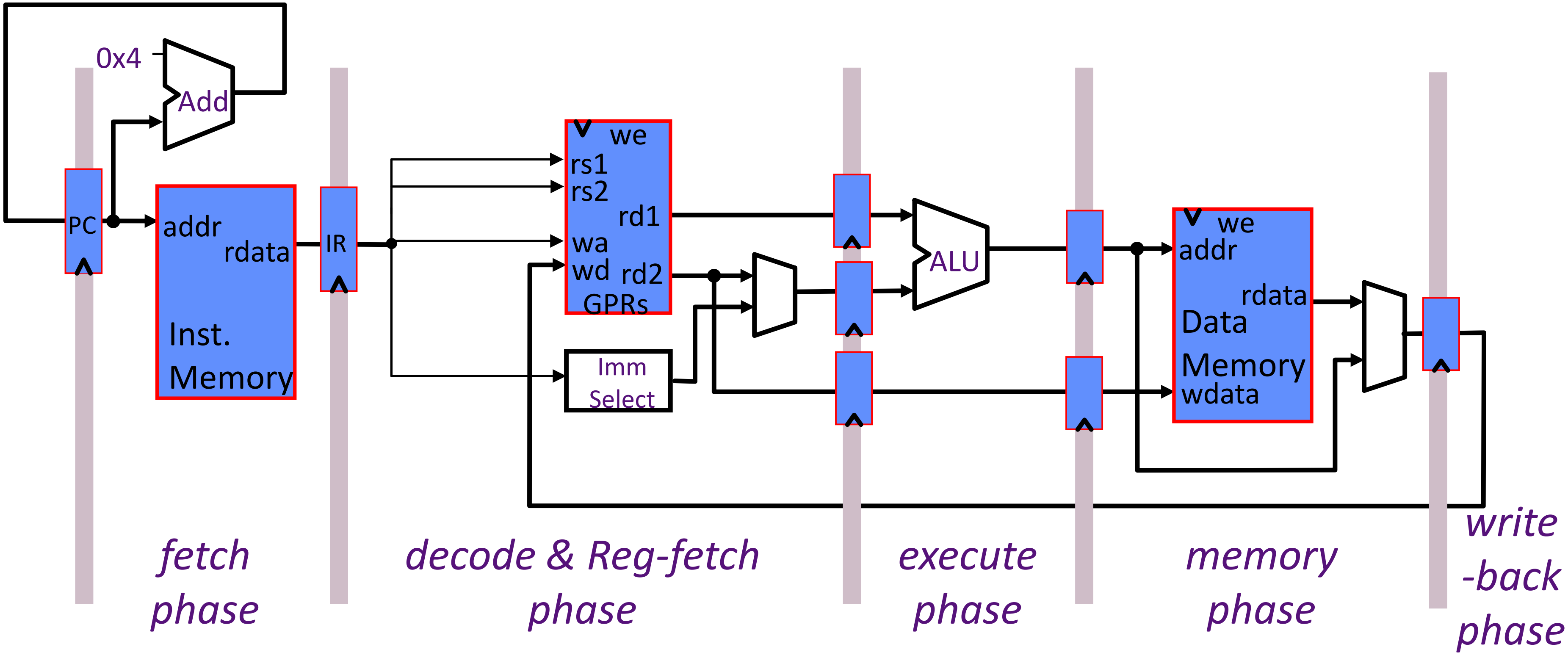
Opcode	ImmSel	Op2Sel	FuncSel	MemWr	RFWen	WBSel	WASel	PCSel
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	IType$_{12}$	Imm	Op	no	yes	ALU	rd	pc+4
LW	IType$_{12}$	Imm	+	no	yes	Mem	rd	pc+4
SW	BsType$_{12}$	Imm	+	yes	no	*	*	pc+4
BEQ$_{true}$	BrType$_{12}$	*	*	no	no	*	*	br
BEQ$_{false}$	BrType$_{12}$	*	*	no	no	*	*	pc+4
JAL	*	*	*		yes	PC	rd	jabs
JALR	*	*	*	no	yes	PC	rd	rind

Op2Sel= Reg / Imm

WBSel = ALU / Mem / PC
PCSel = pc+4 / br / rind / jabs

Correction since L3: “J” has been removed

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

“Iron Law” of Processor Performance

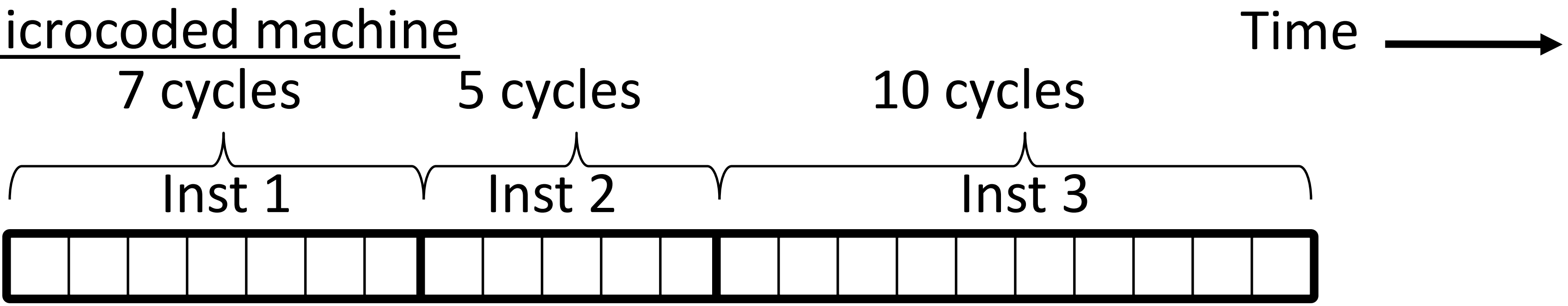
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

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

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

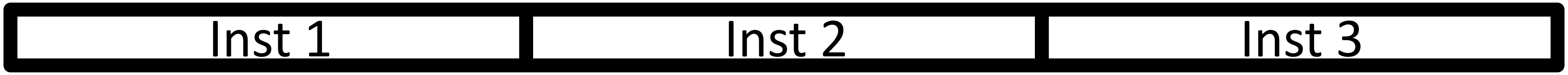
CPI Examples

Microcoded machine



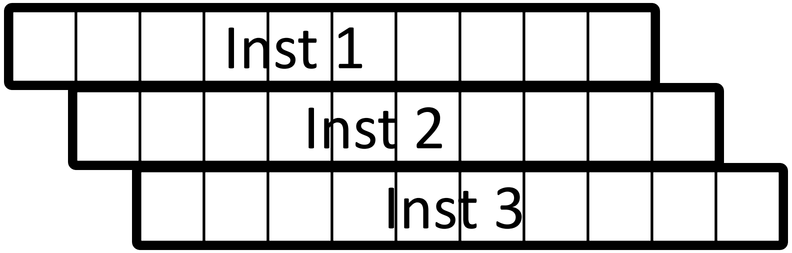
3 instructions, 22 cycles, CPI=7.33

Unpipelined machine



3 instructions, 3 cycles, CPI=1

Pipelined machine



3 instructions, 3 cycles, CPI=1

5-stage pipeline CPI≠5!!!

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!)

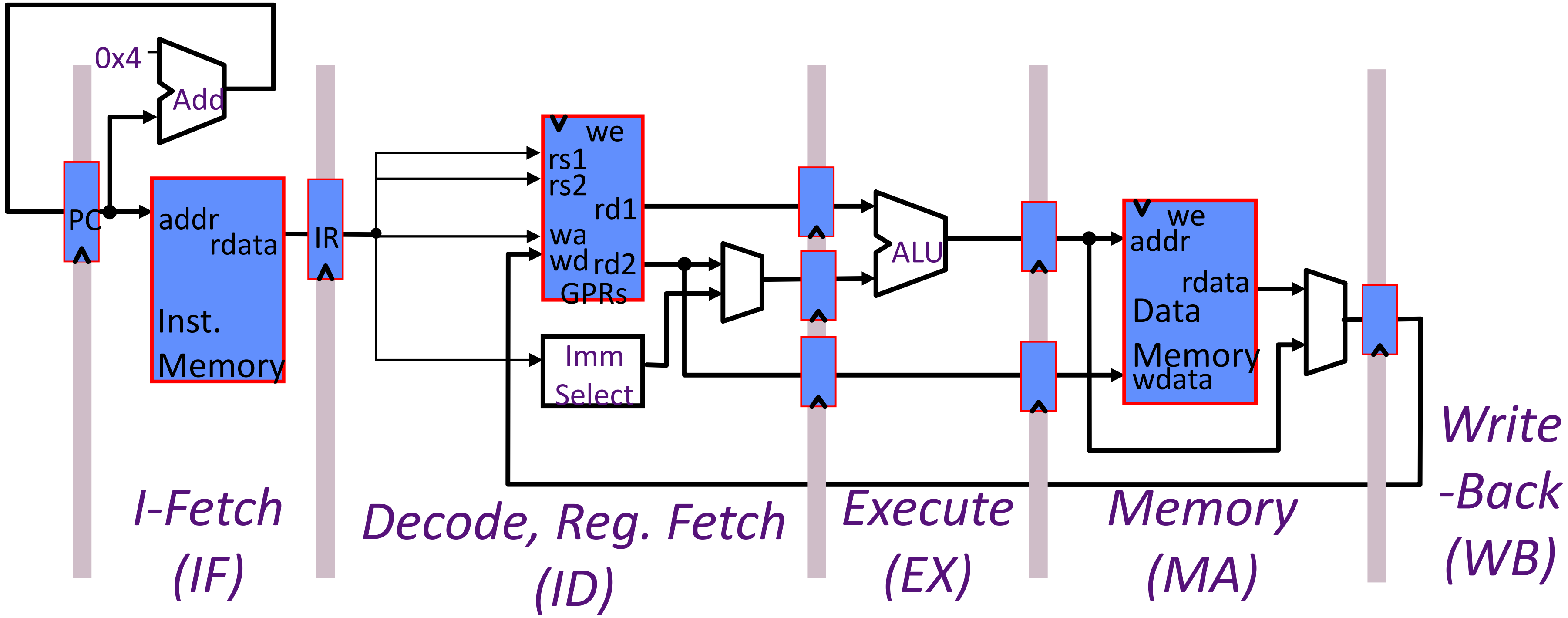
Thus, the following timing assumption is reasonable

$$\boxed{t_{IM} \sim= t_{RF} \sim= t_{ALU} \sim= t_{DM} \sim= t_{RW}}$$

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

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

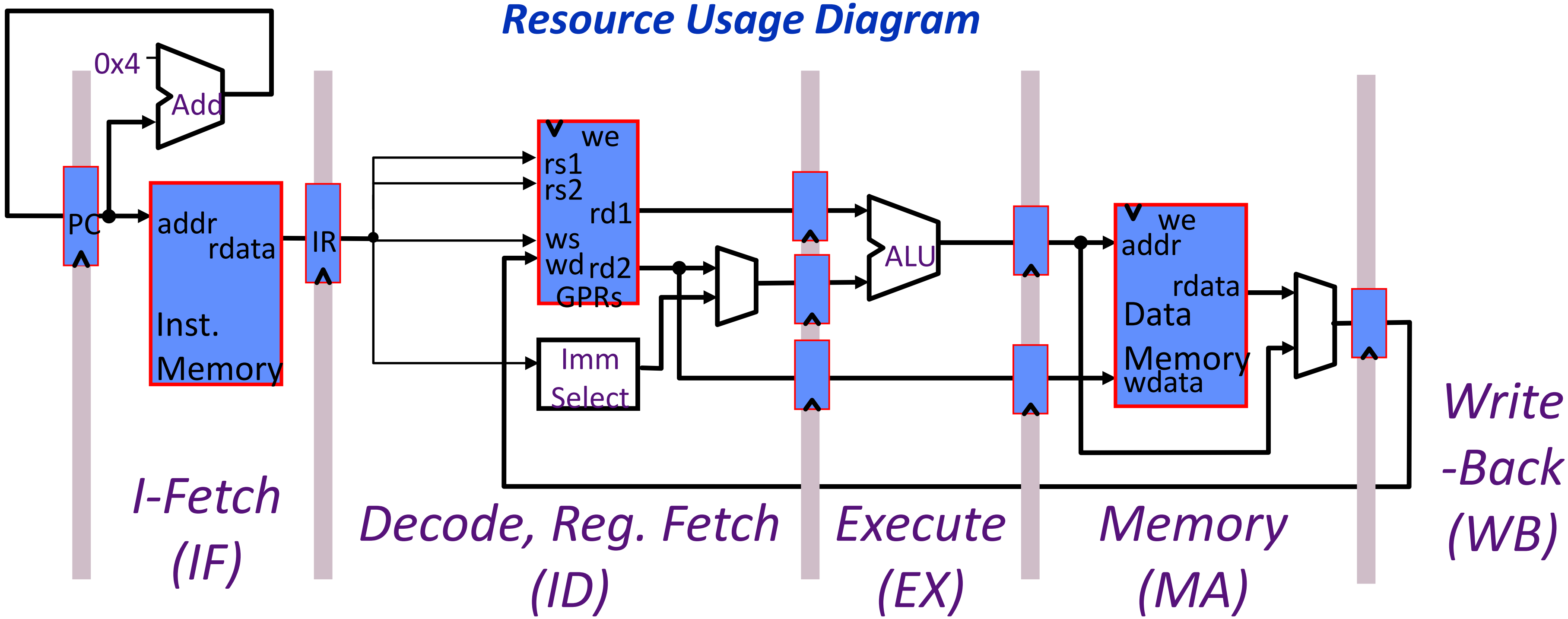
5-Stage Pipelined Execution



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

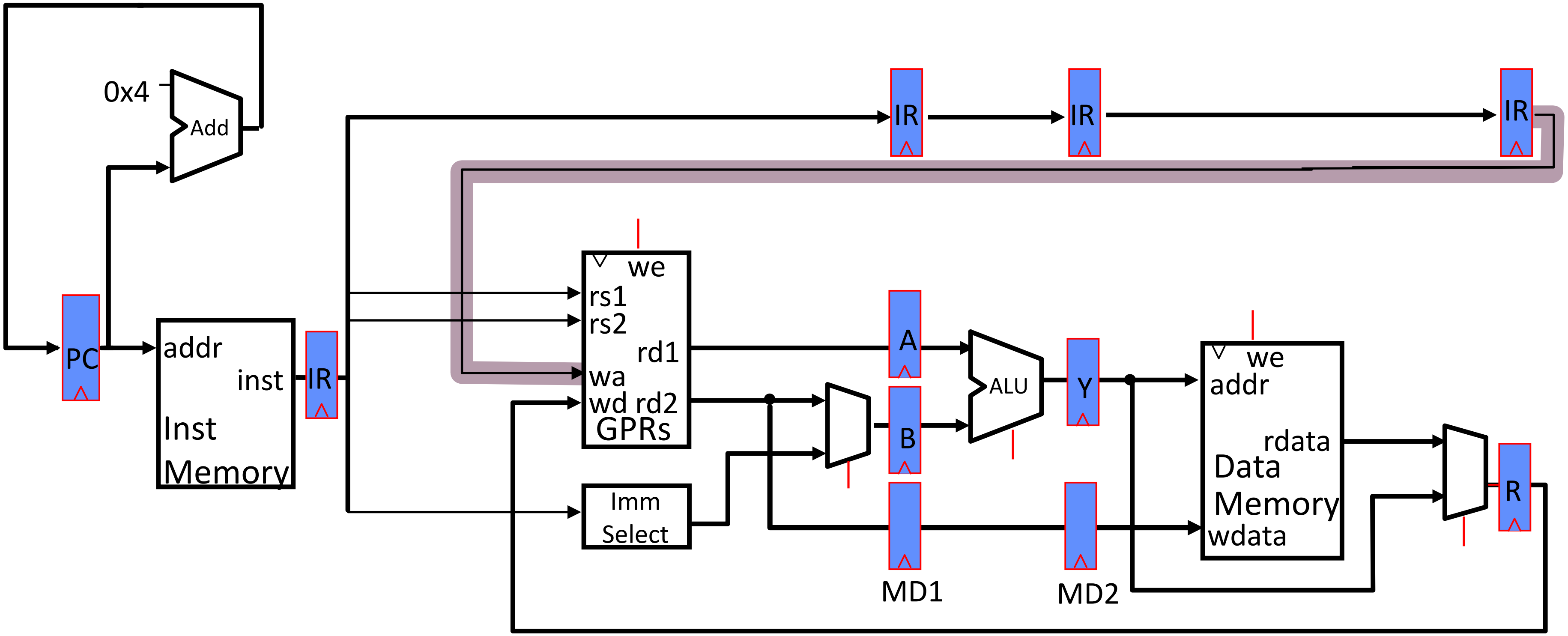


Resources

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

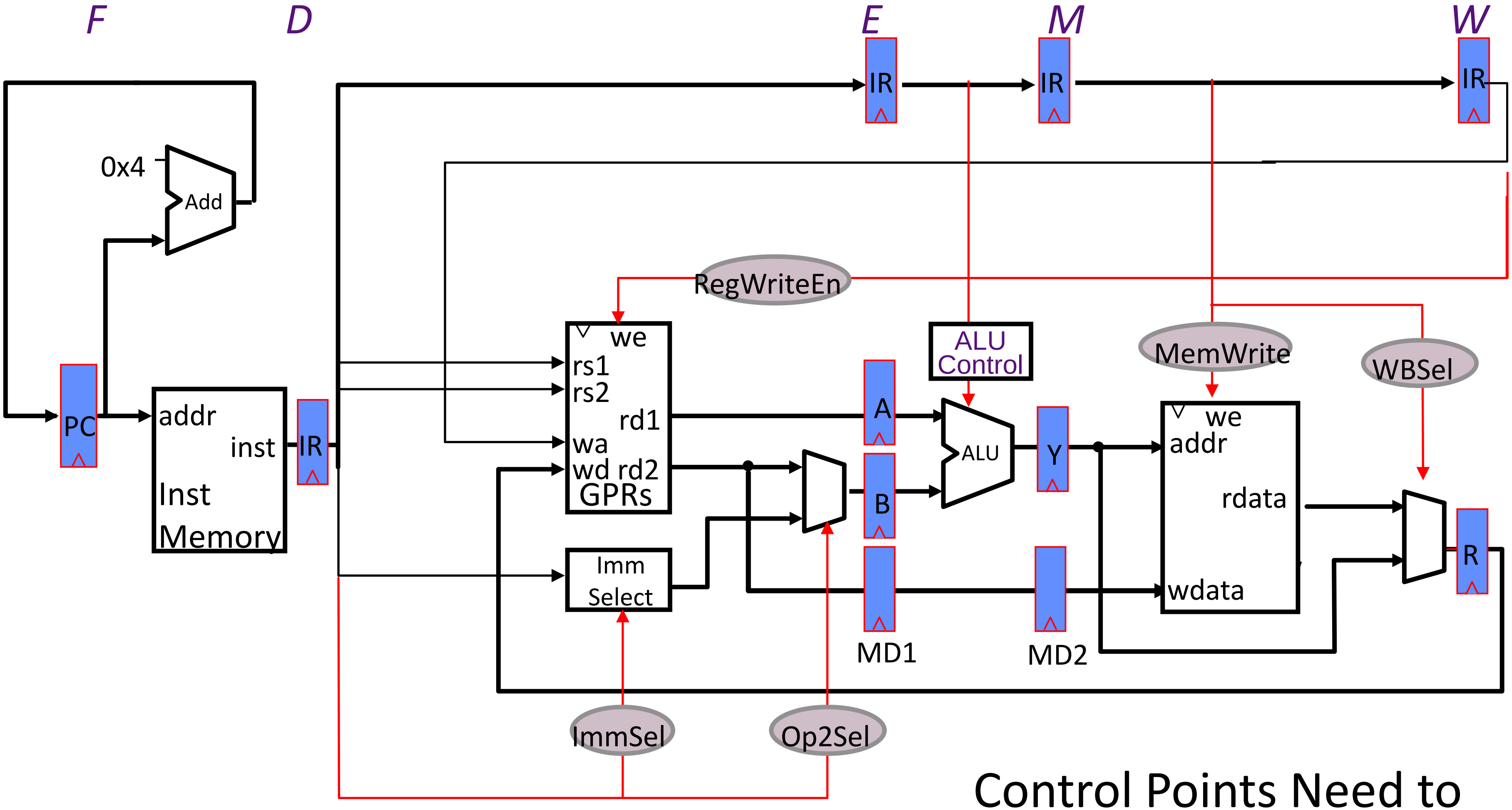


Not quite correct!

We need an Instruction Reg (IR) for each stage

Pipelined RISC-V Datapath

without jumps



Control Points Need to Be Connected

Instructions interact with each other in pipeline

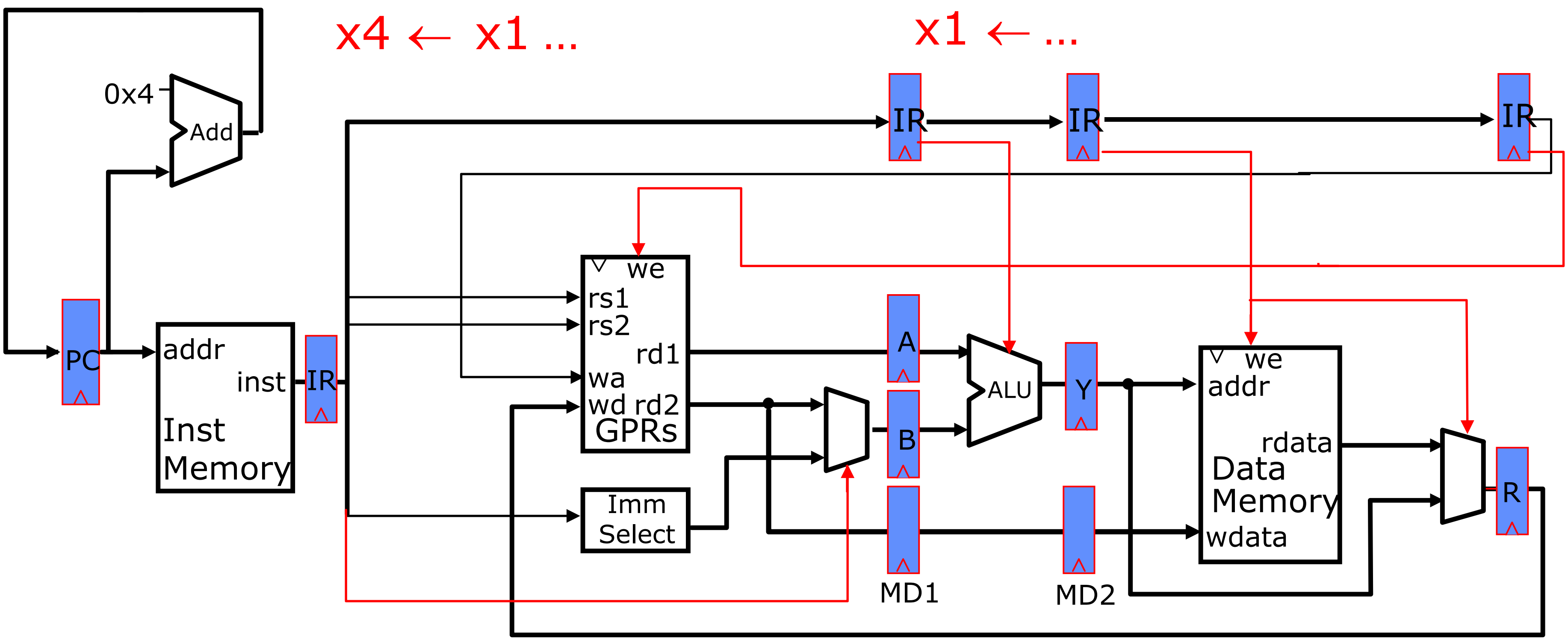
- An instruction in the pipeline may need a resource being used by another instruction in the pipeline
  → *structural hazard*

- 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 hazard occurs when two instructions 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 pipeline has no structural hazards by design
  - Thanks to RISC-V ISA, which was designed for pipelining

Data Hazards



…
x1 ← x0 + 10
x4 ← x1 + 17
…

x1 is stale. Oops!

How Would You Resolve This?

- What can you do if your project buddy has not completed their part, which you need to start?

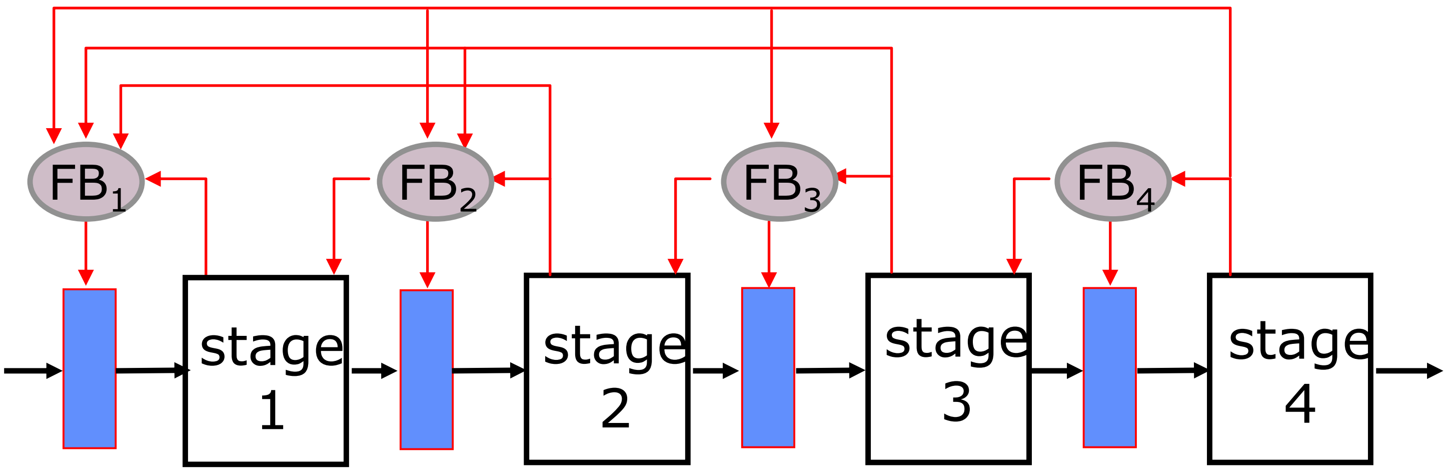
- Three options
  - Wait (stall)
  - Speculate on the value of the deliverable
  - Bypass: ask them for what you need before his/her final deliverable
  - (ask him/her to work harder?)

Resolving Data Hazards (1)

Strategy 1:

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

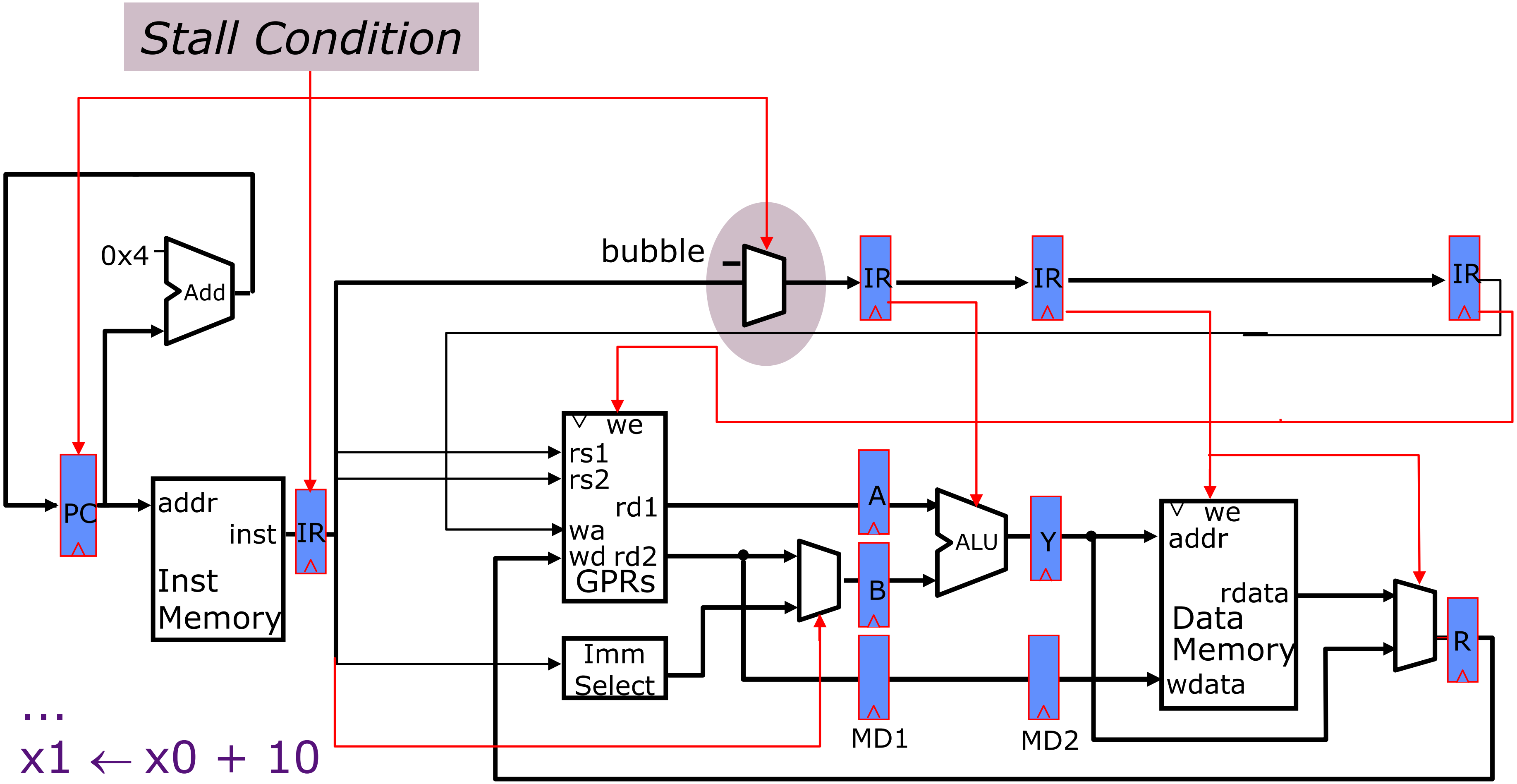
Feedback to Resolve Hazards



- 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



...
x1 ← x0 + 10
x4 ← x1 + 17
...

Stalled Stages and Pipeline Bubbles

	time											
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
(I_1) x1 ← (x0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1							
(I_2) x4 ← (x1) + 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

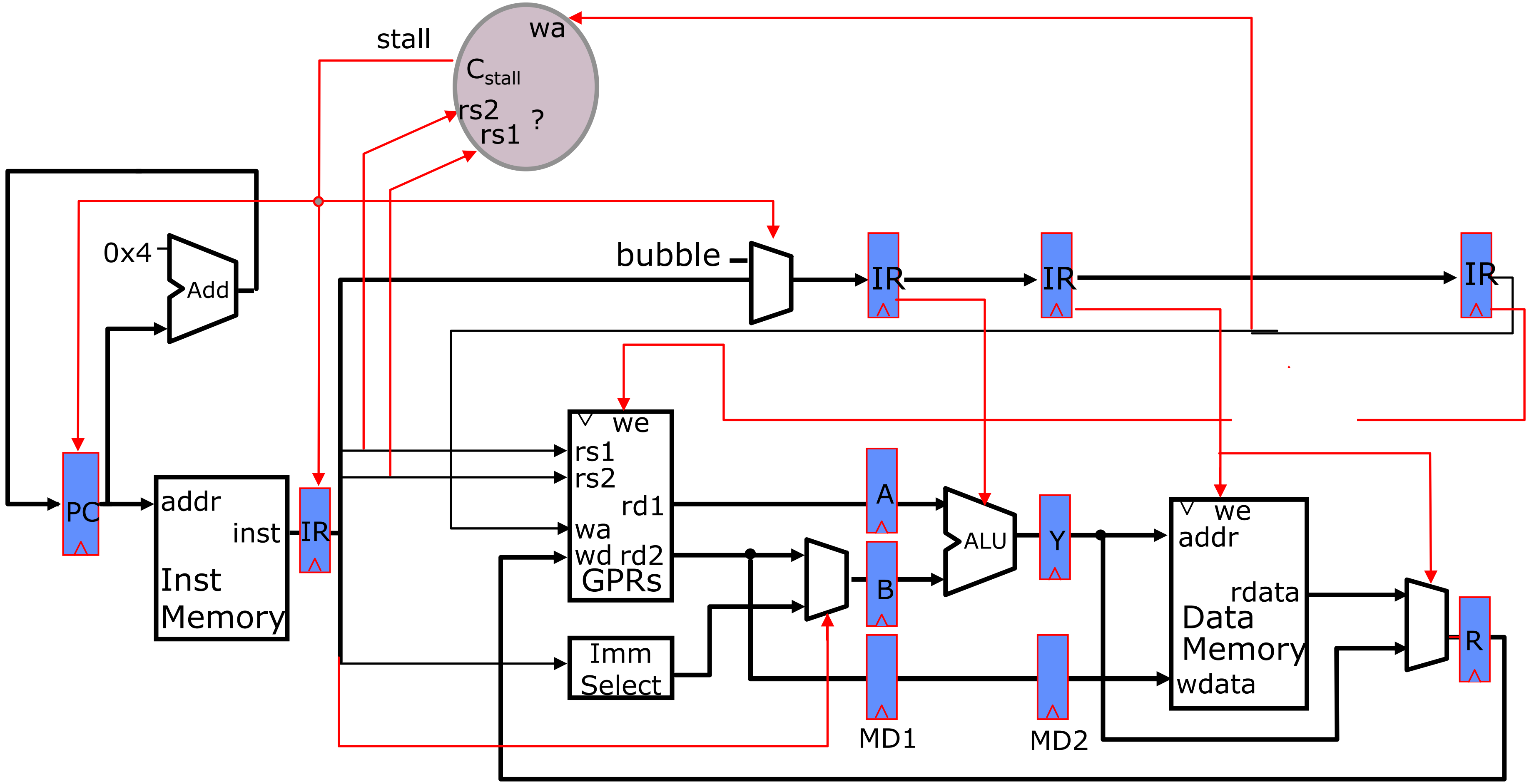
stalled stages

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	-	-	-	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

- ⇒ *pipeline bubble*

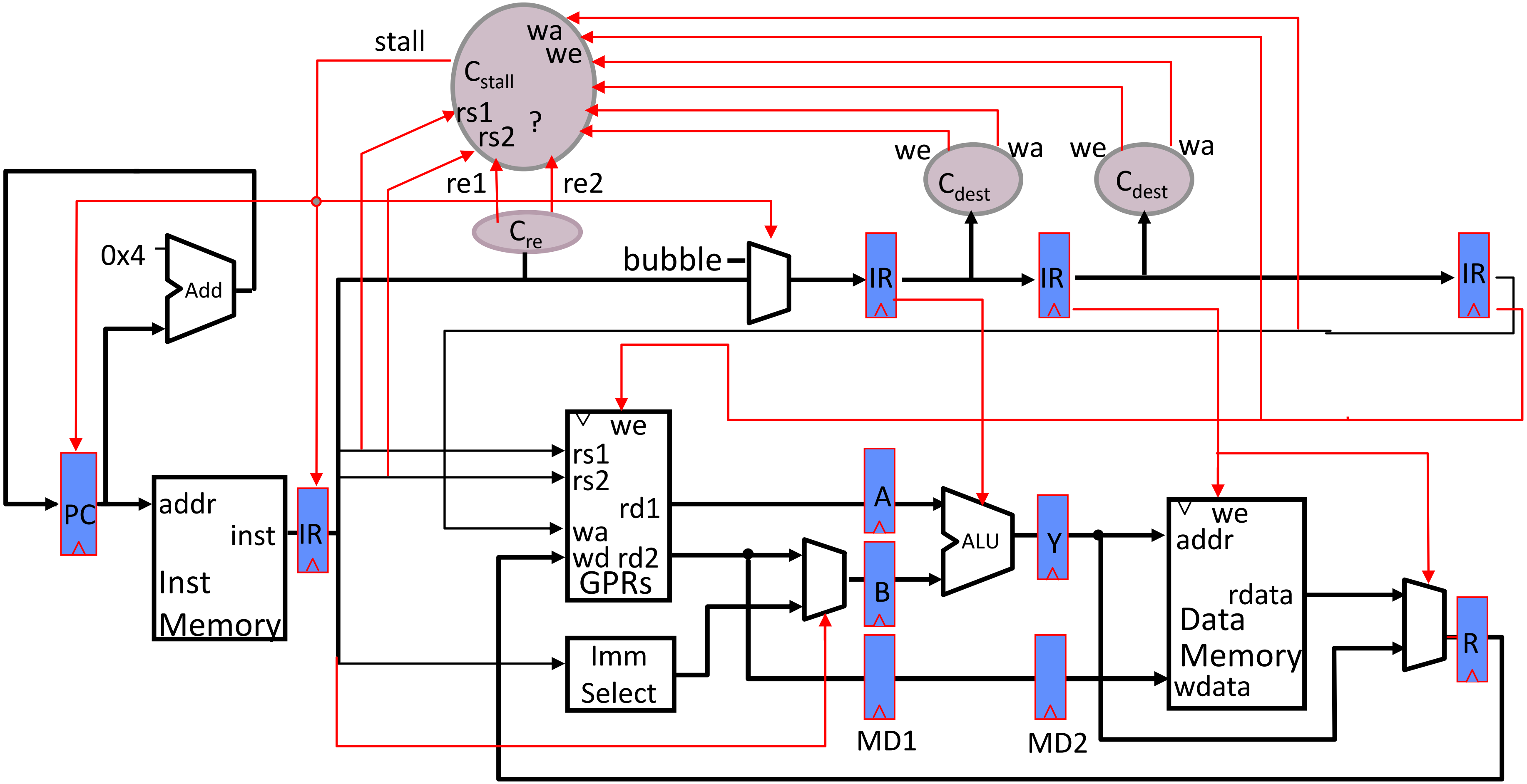
Interlock Control Logic



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 an rs field matches some rd?

not every instruction writes a register => we

not every instruction reads a register => re

Source & Destination Registers

func7	rs2	rs1	func3	rd	opcode	ALU
immediate12		rs1	func3	rd	opcode	ALU/LW/JALR
imm	rs2	rs1	func3	imm	opcode	SW/Bcond
Jump Offset[19:0]				rd	opcode	

0

		source(s)	*destination*
ALU	rd <= rs1 func10 rs2	rs1, rs2	rd
ALUI	rd <= rs1 op imm	rs1	rd
LW	rd <= M [rs1 + imm]	rs1	rd
SW	M [rs1 + imm] <= rs2	rs1, rs2	-
B*cond*	rs1,rs2	rs1, rs2	-
	true: PC <= PC + imm		
	false: PC <= PC + 4		
JAL	x1 <= PC, PC <= PC + imm	-	rd
JALR	rd <= PC, PC <= rs1 + imm	rs1	rd

Deriving the Stall Signal

C_{dest}

ws = rd

we = *Case* opcode
    ALU, ALUi, LW, JALR =>on
    ...   =>off

C_{re}

re1 = *Case* opcode
    ALU, ALUi,
    LW, SW, Bcond,
    JALR   =>on
    JAL   =>off

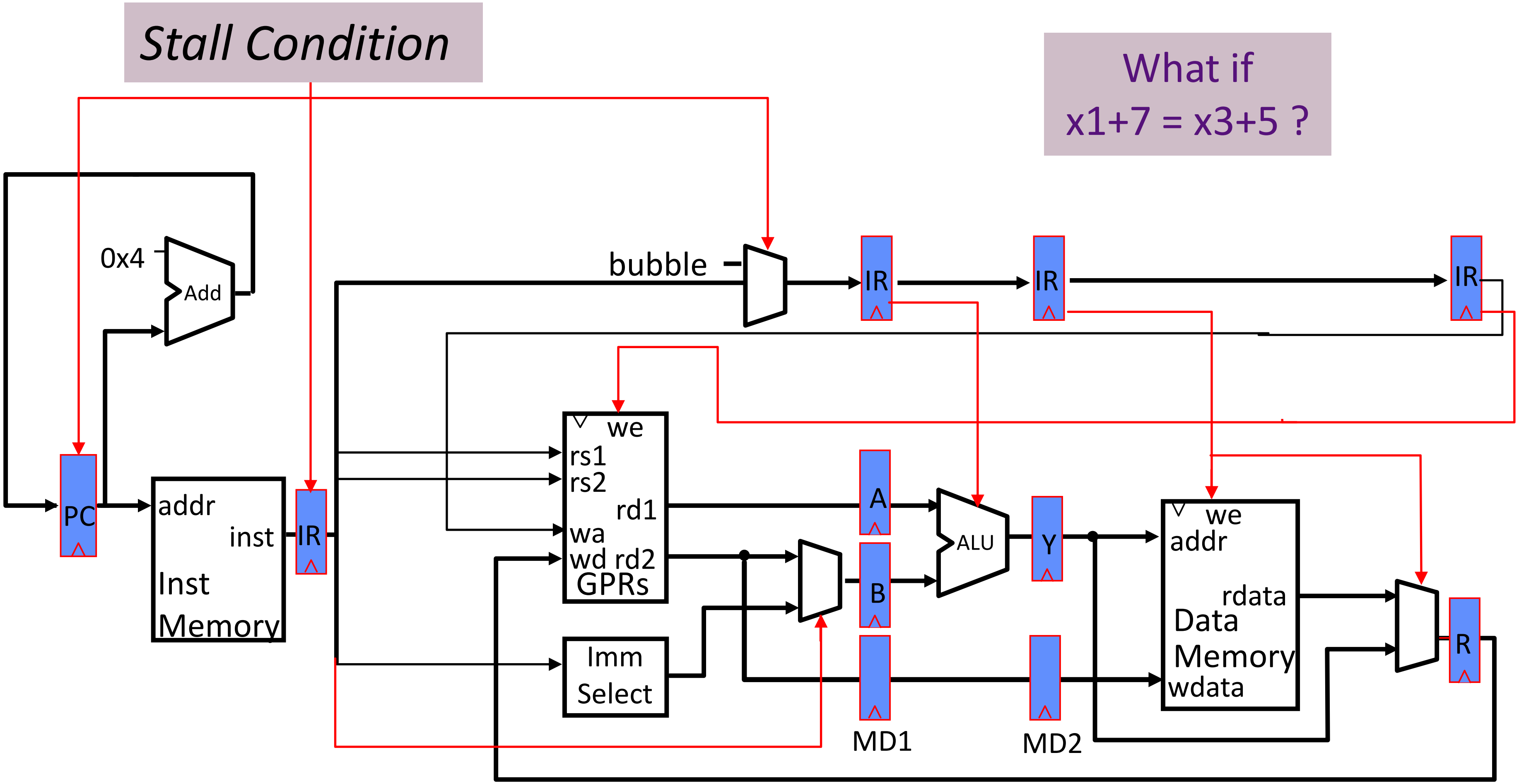
re2 = *Case* opcode
    ALU, SW, Bcond =>on
    ...   ->off

C_{stall}

$$
\begin{aligned}
\text{stall} = \; & ((rs1_D = ws_E).we_E + \\
& (rs1_D = ws_M).we_M + \\
& (rs1_D = ws_W).we_W) \; . \; re1_D \quad + \\
& ((rs2_D = ws_E).we_E + \\
& (rs2_D = ws_M).we_M + \\
& (rs2_D = ws_W).we_W) \; . \; re2_D
\end{aligned}
$$

This is not the full story !

Hazards due to Loads & Stores



...

M[x1+7] <= x2

x4 <= M[x3+5]

...

Is there any possible data hazard in this instruction sequence?

Load & Store Hazards

```
…
M[x1+7] <= x2
x4 <= M[x3+5]
…
```

x1+7 = x3+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.

CS152 Administrivia

- Quiz 1 on Feb 17 will cover PS1, Lab1, lectures 1-5, and associated readings.

Resolving Data Hazards (2)

Strategy 2:

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

Bypassing

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) x1 ← x0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) x4 ← x1 + 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
(I_4)			*stalled stages*				IF_4	ID_4	EX_4
(I_5)								IF_5	ID_5

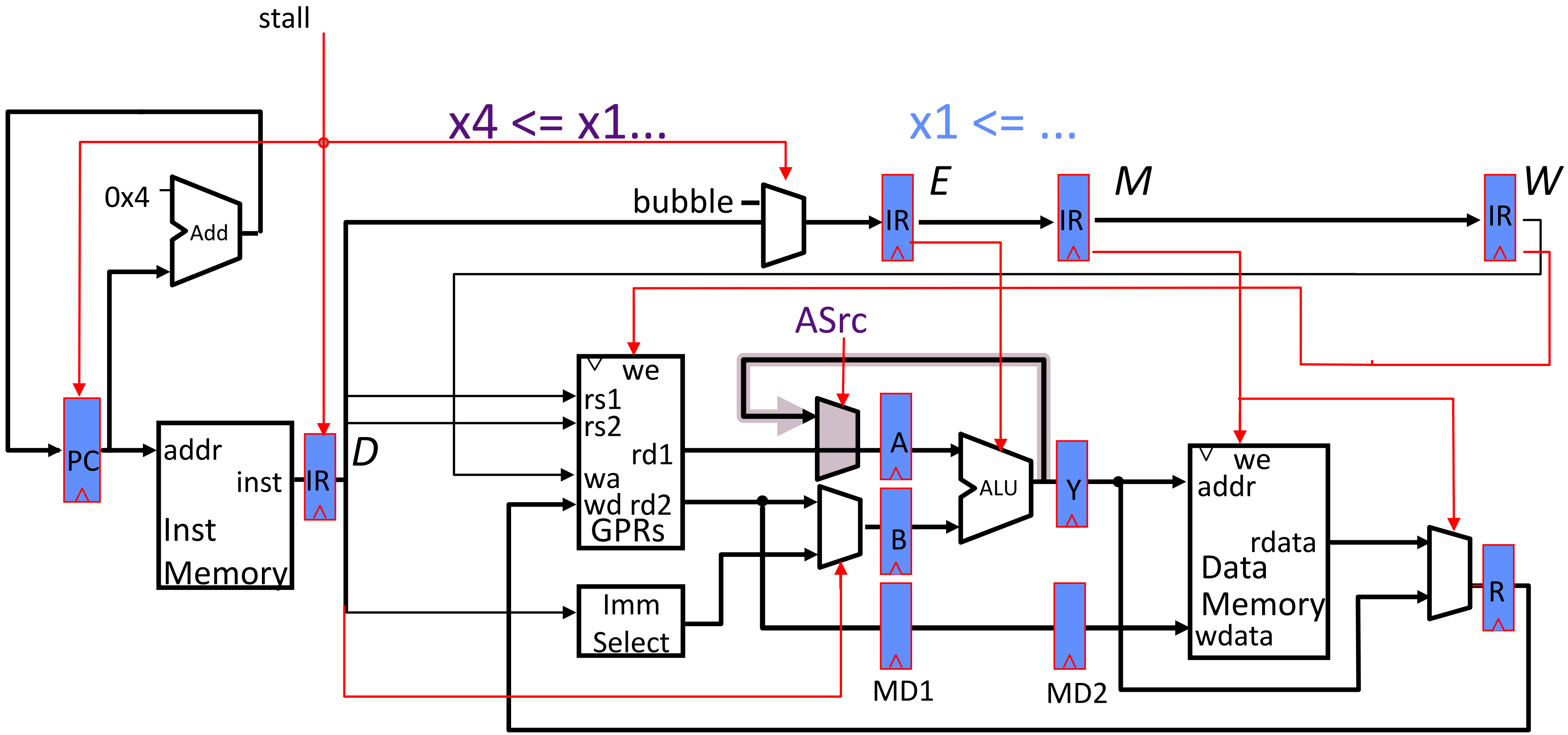
Each *stall or kill* introduces a bubble in the pipeline

=> *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) x1 ← x0 + 10		IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) x4 ← x1 + 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

Adding a Bypass



When does this bypass help?

...

(I_1)	x1 <= x0 + 10	x1 <= M[x0 + 10]	JAL 500
(I_2)	x4 <= x1 + 17	x4 <= x1 + 17	x4 <= x1 + 17
	yes	*no*	*no*

The Bypass Signal

Deriving it from the Stall Signal

$$\begin{aligned} stall = ((&\cancel{(rs1_D = ws_E).we_E} + (rs1_D = ws_M).we_M + (rs1_D = ws_W).we_W).re1_D \\ &+((rs2_D = ws_E).we_E + (rs2_D = ws_M).we_M + (rs2_D = ws_W).we_W).re2_D) \end{aligned}$$

ws = rd

we = *Case* opcode
    ALU, ALUi, LW,, JAL JALR => on
    ...   => off

$ASrc = (rs1_D = ws_E).we_E.re1_D$

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 = *Case* opcode_E

- ALU, ALUi => on
- ... => off

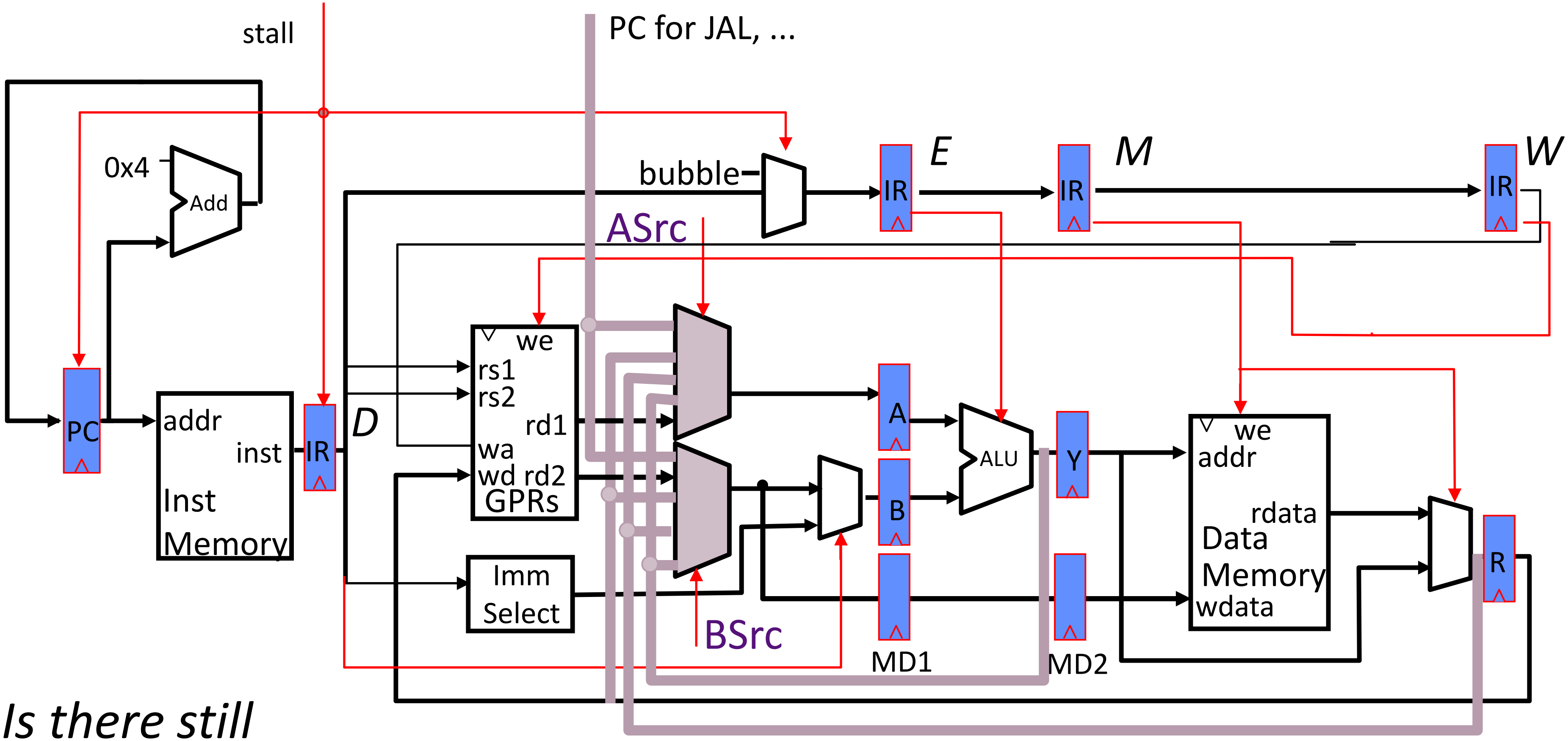
we-stall_E = *Case* opcode_E

- LW, JAL, JALR=> on
- JAL => on
- ... => off

$$\text{ASrc} = (rs1_D = ws_E).\text{we-bypass}_E \,.\, re1_D$$

$$\text{stall} = ((rs1_D = ws_E).\text{we-stall}_E + (rs1_D = ws_M).we_M + (rs1_D = ws_W).we_W).\, re1_D + ((rs2_D = ws_E).we_E + (rs2_D = ws_M).we_M + (rs2_D = ws_W).we_W).\, re2_D$$

Fully Bypassed Datapath

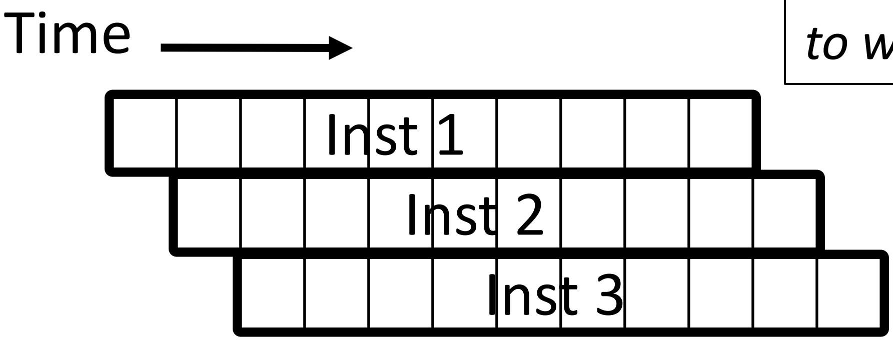


Is there still a need for the stall signal ?

$$\text{stall} = (rs1_D = ws_E).(opcode_E = LW_E).(ws_E != 0).re1_D + (rs2_D = ws_E).(opcode_E = LW_E).(ws_E != 0).re2_D$$

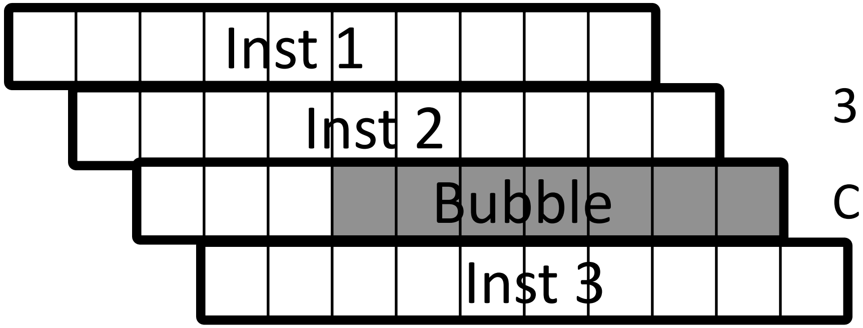
Pipeline CPI Examples

Measure from when first instruction finishes to when last instruction in sequence finishes.



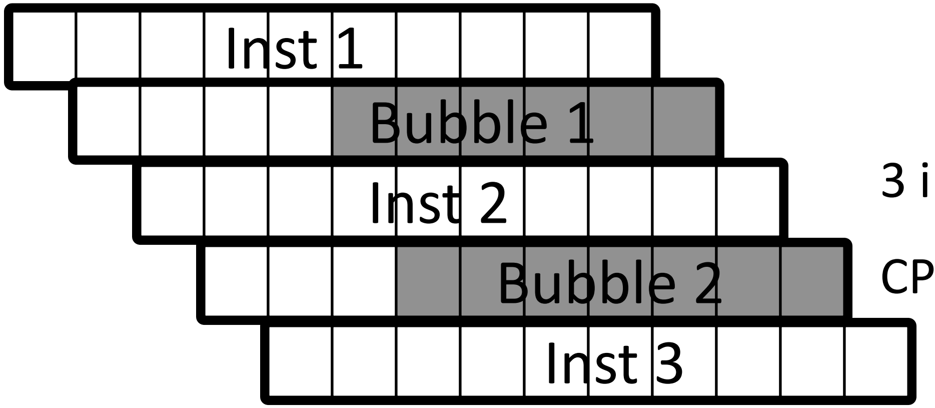
3 instructions finish in 3 cycles

CPI = 3/3 =1



3 instructions finish in 4 cycles

CPI = 4/3 = 1.33



3 instructions finish in 5cycles

CPI = 5/3 = 1.67

Resolving Data Hazards (3)

Strategy 3: Speculate on the dependence!

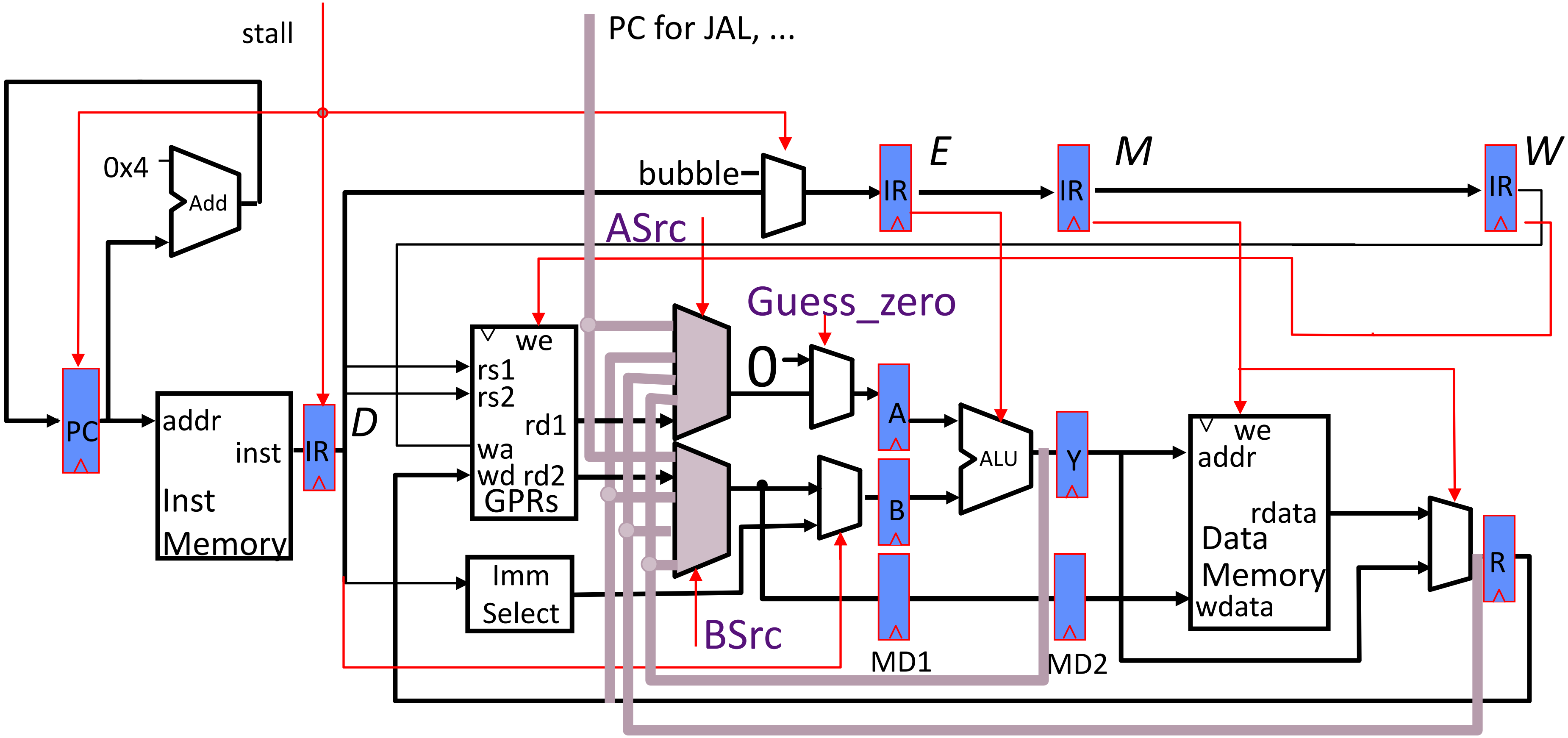
Two cases:

Guessed correctly → do nothing

Guessed incorrectly → kill and restart

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

Speculation that load value=zero



Guess_zero= $(rs1_D=ws_E).(opcode_E=LW_E).(ws_E!=0).re1_D$

Also need to add circuitry to remember that this was a guess and flush pipeline if load not zero!

Not worth doing in practice – why?

Control Hazards

What do we need to calculate next PC?

- For Jumps
  - Opcode, PC and offset
- For Jump Register
  - Opcode, Register value, and PC
- For Conditional Branches
  - Opcode, Register (for condition), PC and offset
- For all other instructions
  - Opcode and PC (and have to know it’s not one of above)

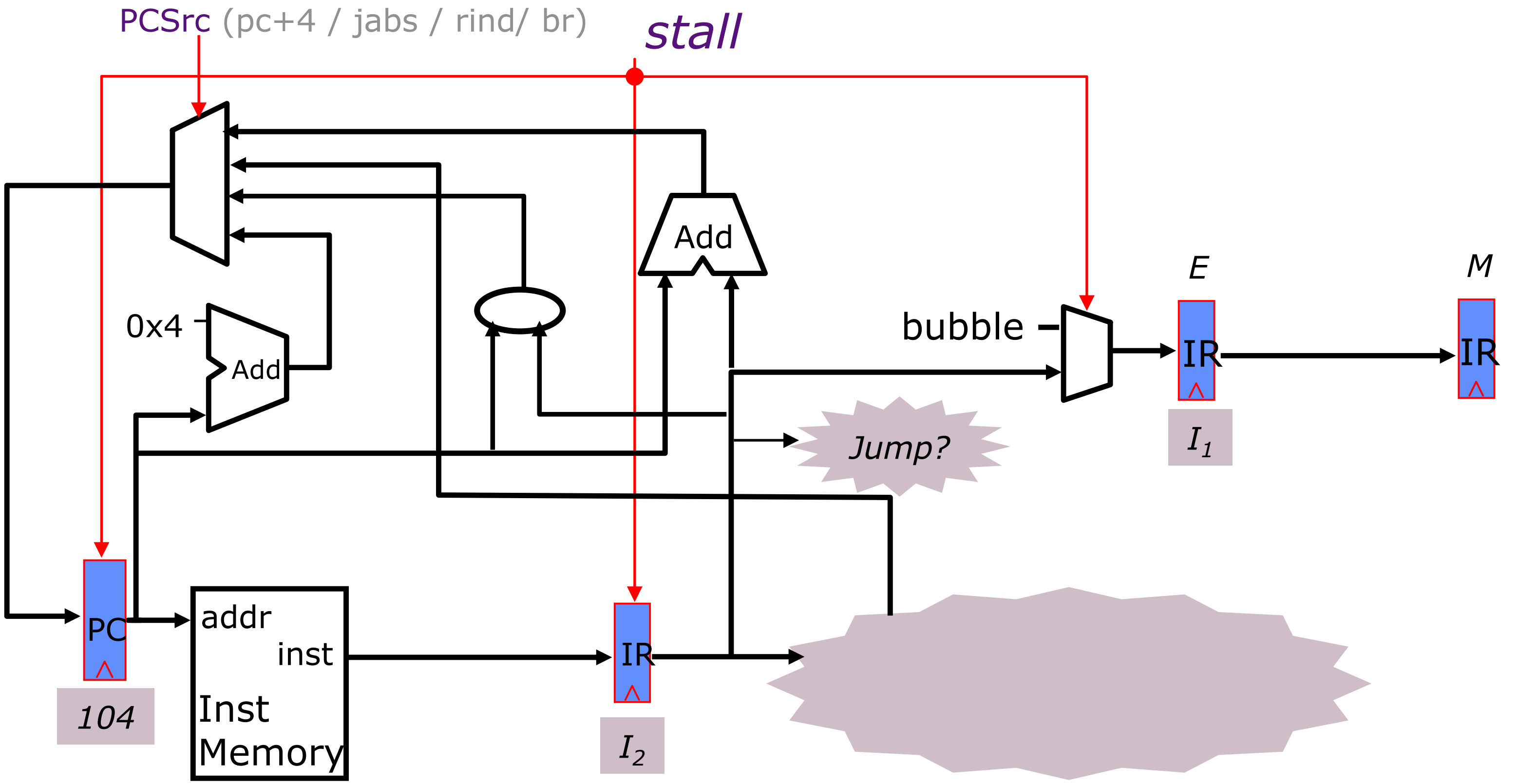
PC Calculation Bubbles

	time										
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
(I_1) x1 ← x0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1						
(I_2) x3 ← x2 + 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

		time										
		t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
Resource Usage	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

- ⇒ *pipeline bubble*

Speculate next address is PC+4



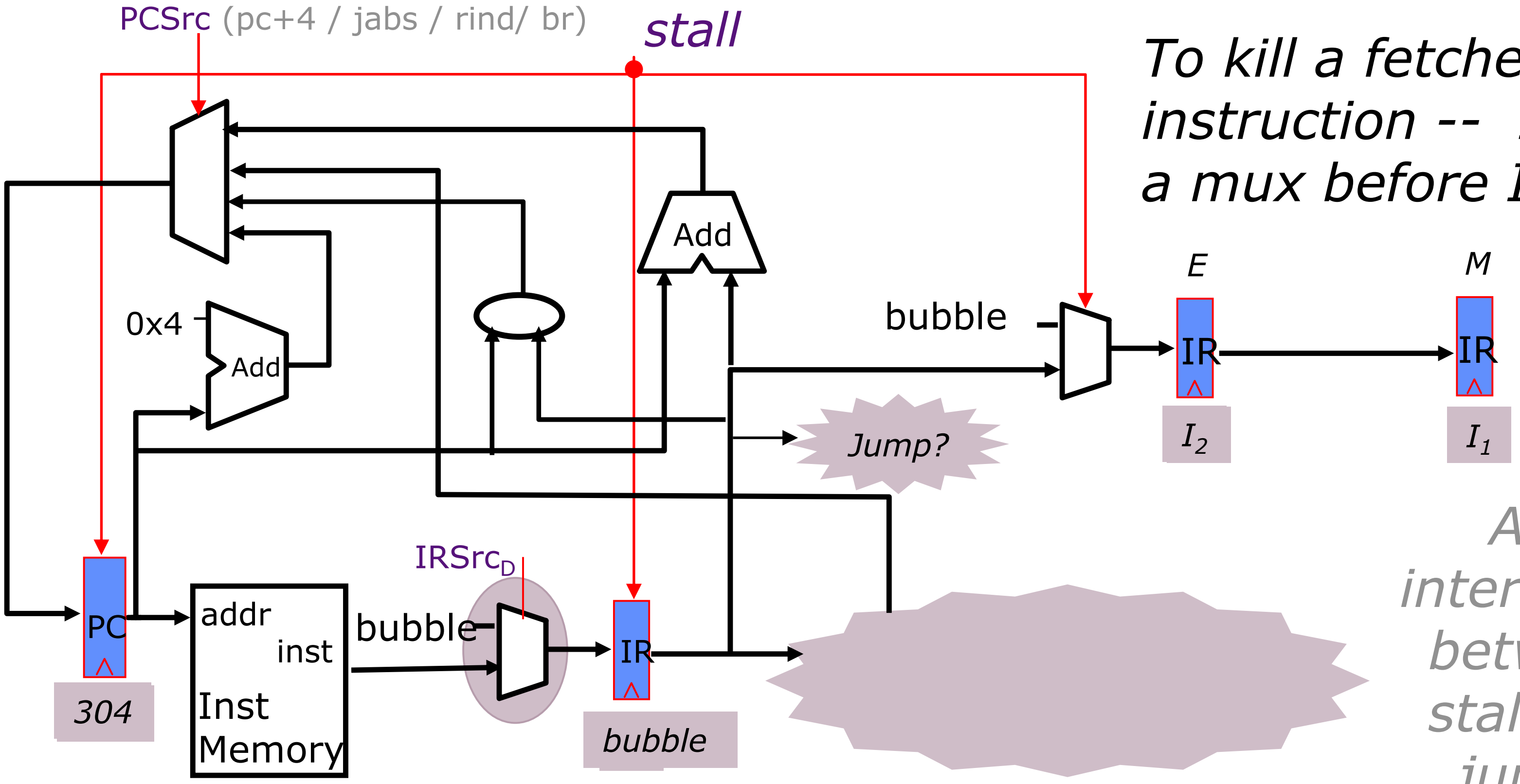
I_1	096	ADD
I_2	100	J 304
I_3	~~104~~	~~ADD~~
I_4	304	ADD

kill

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* $opcode_D$
JAL ⇒ bubble
... ⇒ 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	-	-	-	-		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

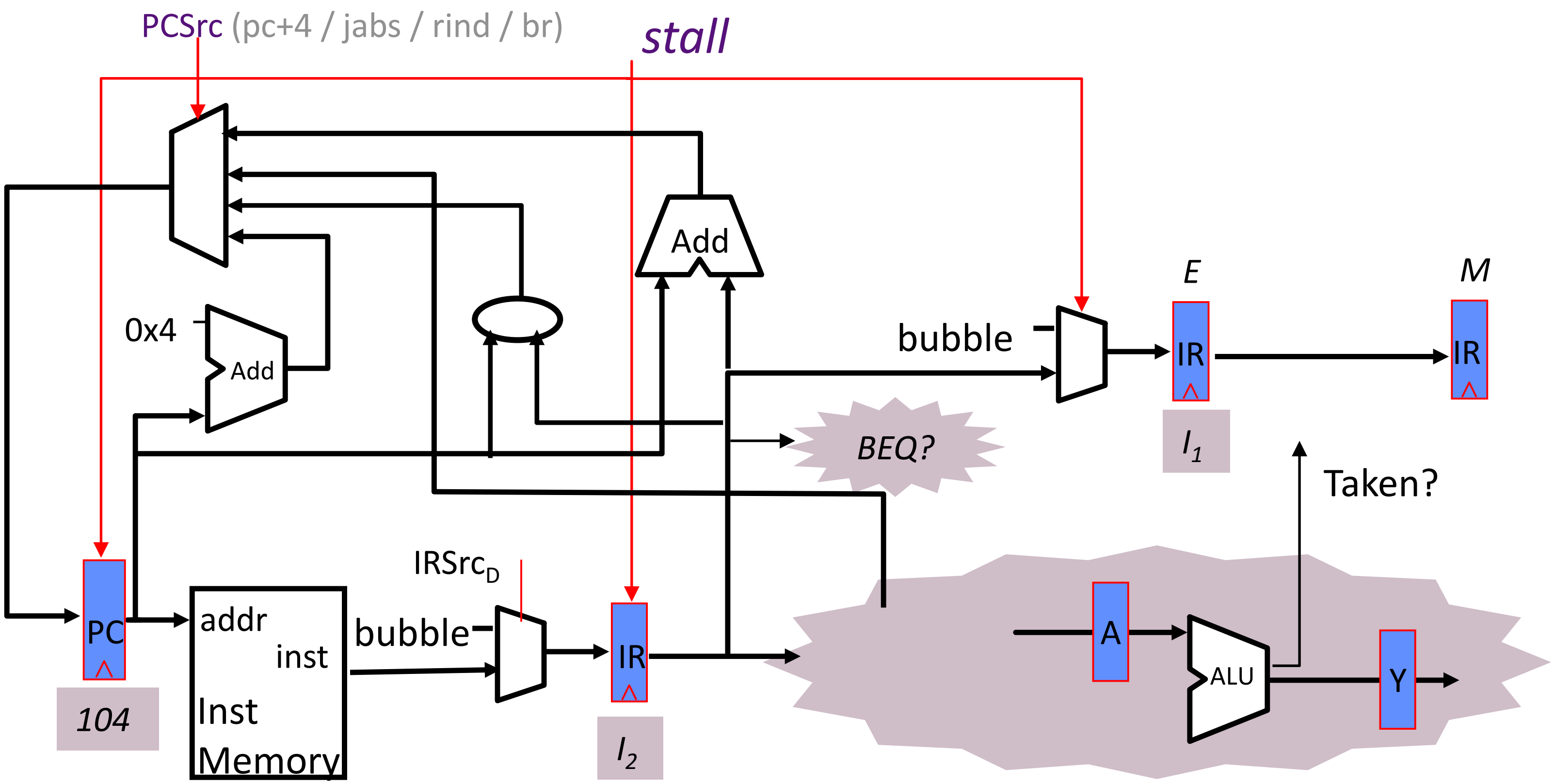
Resource Usage

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	-	I_4	I_5			
EX			I_1	I_2	-	I_4	I_5		
MA				I_1	I_2	-	I_4	I_5	
WB					I_1	I_2	-	I_4	I_5

- ⇒ *pipeline bubble*

Pipelining Conditional Branches

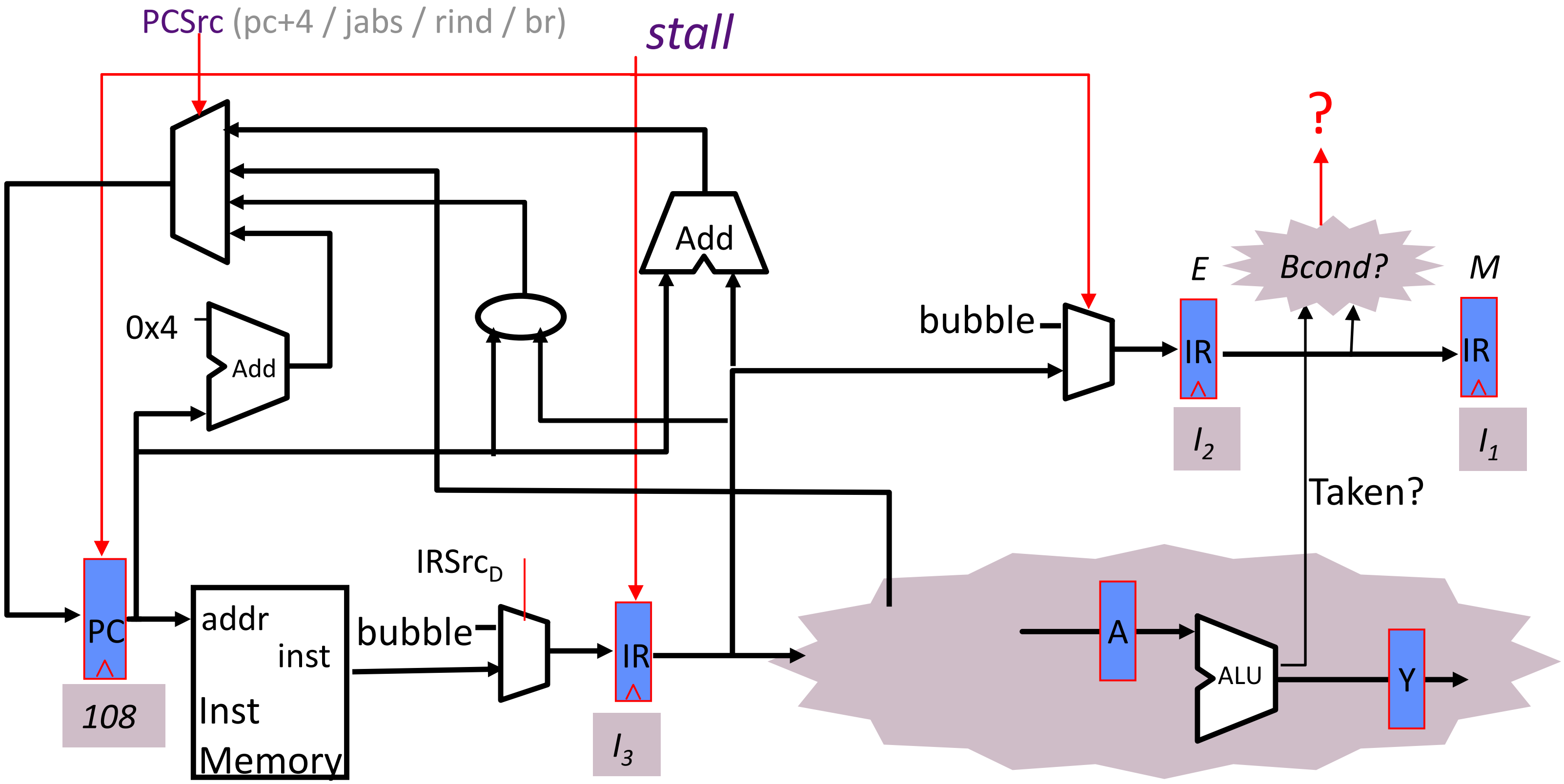


I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	304	ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Pipelining Conditional Branches

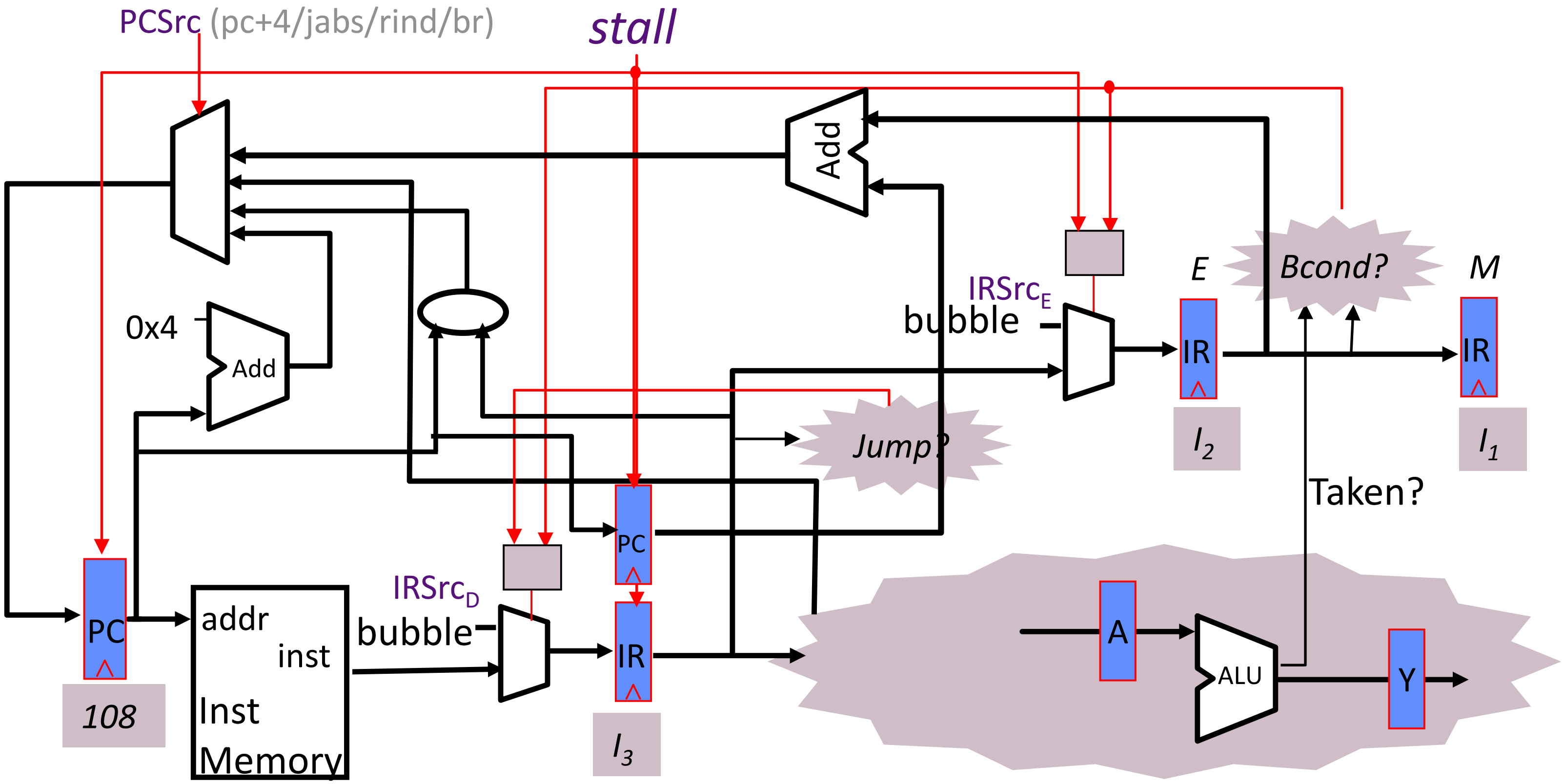


I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	304	ADD

If the branch is taken

- kill the two following instructions
- the instruction at the decode stage is not valid ⇒ *stall signal is not valid*

Pipelining Conditional Branches



$I_{1:}$	096	ADD
$I_{2:}$	100	BEQ x1,x2 +200
$I_{3:}$	104	ADD
$I_{4:}$	304	ADD

If the branch is taken

- kill the two following instructions
- the instruction at the decode stage is not valid ⇒ *stall signal is not valid*

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

time

Resource Usage

	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_5			
EX			I_1	I_2	-	-	I_5		
MA				I_1	I_2	-	-	I_5	
WB					I_1	I_2	-	-	I_5

- ⇒ *pipeline bubble*

Question of the Day

- Why a five stage pipeline?

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