

# Acknowledgement

---

- Slides adapted from Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley

# Review

---

- **CPU performance factors**

- **Instruction count**

- **Determined by ISA and compiler**

- **CPI and Cycle time**

- **Determined by CPU hardware**

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

- **Three groups of instructions**

- **Memory reference: lw, sw**

- **Arithmetic/logical: add, sub, and, or, slt**

- **Control transfer: jal, jalr, b\***

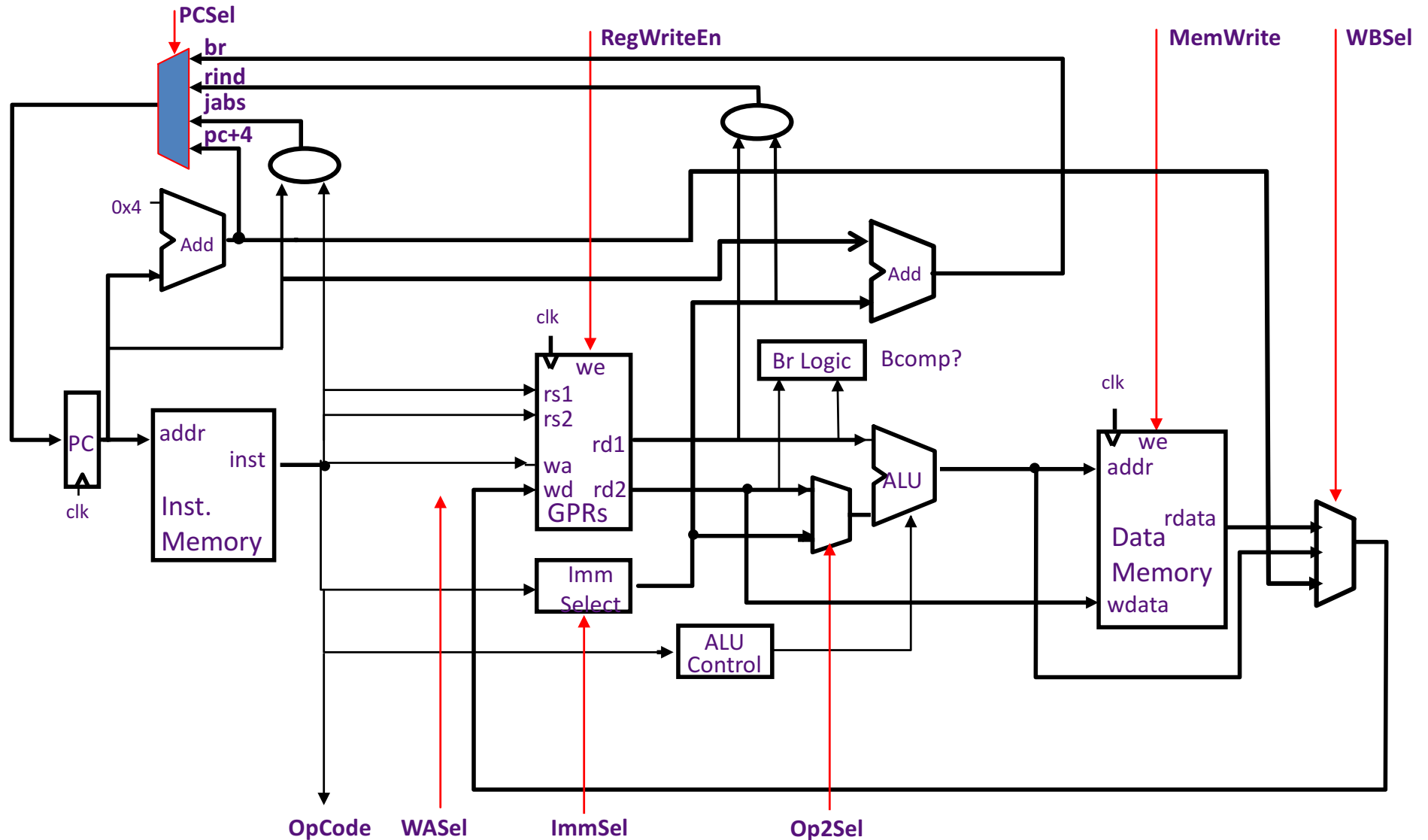
- **CPI**

- **Single-cycle, CPI = 1, and normally longer cycle**

- **5 stage unpipelined, CPI = 5**

- **5 stage pipelined, CPI = 1**

# Review: Unpipelined Datapath for RISC-V



# Review: Hardwired Control Table

Opcode	ImmSel	Op2Sel	FuncSel	MemWr	RFWen	WBSel	WASel	PCSel
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	IType <sub>12</sub>	Imm	Op	no	yes	ALU	rd	pc+4
LW	IType <sub>12</sub>	Imm	+	no	yes	Mem	rd	pc+4
SW	SType <sub>12</sub>	Imm	+	yes	no	*	*	pc+4
BEQ <sub>true</sub>	SType <sub>12</sub>	*	*	no	no	*	*	br
BEQ <sub>false</sub>	SType <sub>12</sub>	*	*	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	X1	jabs
JALR	*	*	*	no	yes	PC	rd	rind

Op2Sel= Reg / Imm

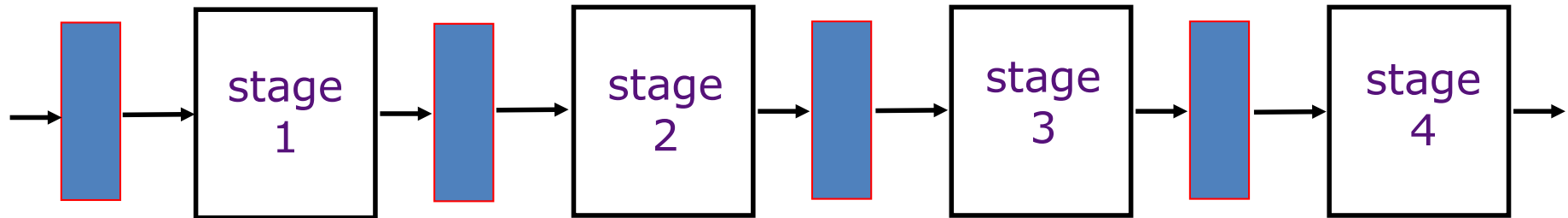
WASel = rd / X1

WBSel = ALU / Mem / PC

PCSel = pc+4 / br / rind / jabs

# 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***

**For laundry pipeline, two loads do not depend on each other.**

***But instructions depend on each other!***

# 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)

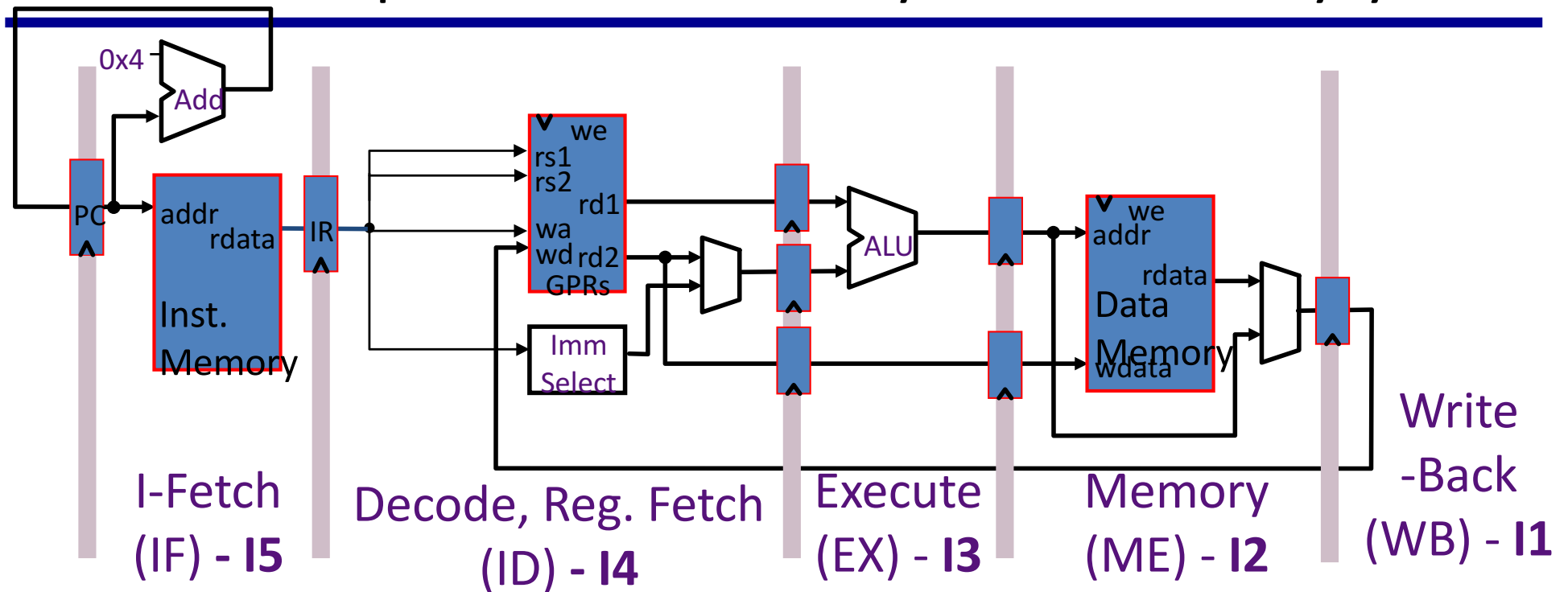
Thus, the following timing assumption is reasonable

$$t_{IF} \sim t_{ID/RF} \sim t_{EX} \sim t_{MEM} \sim t_{WB}$$

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: Resource Usage

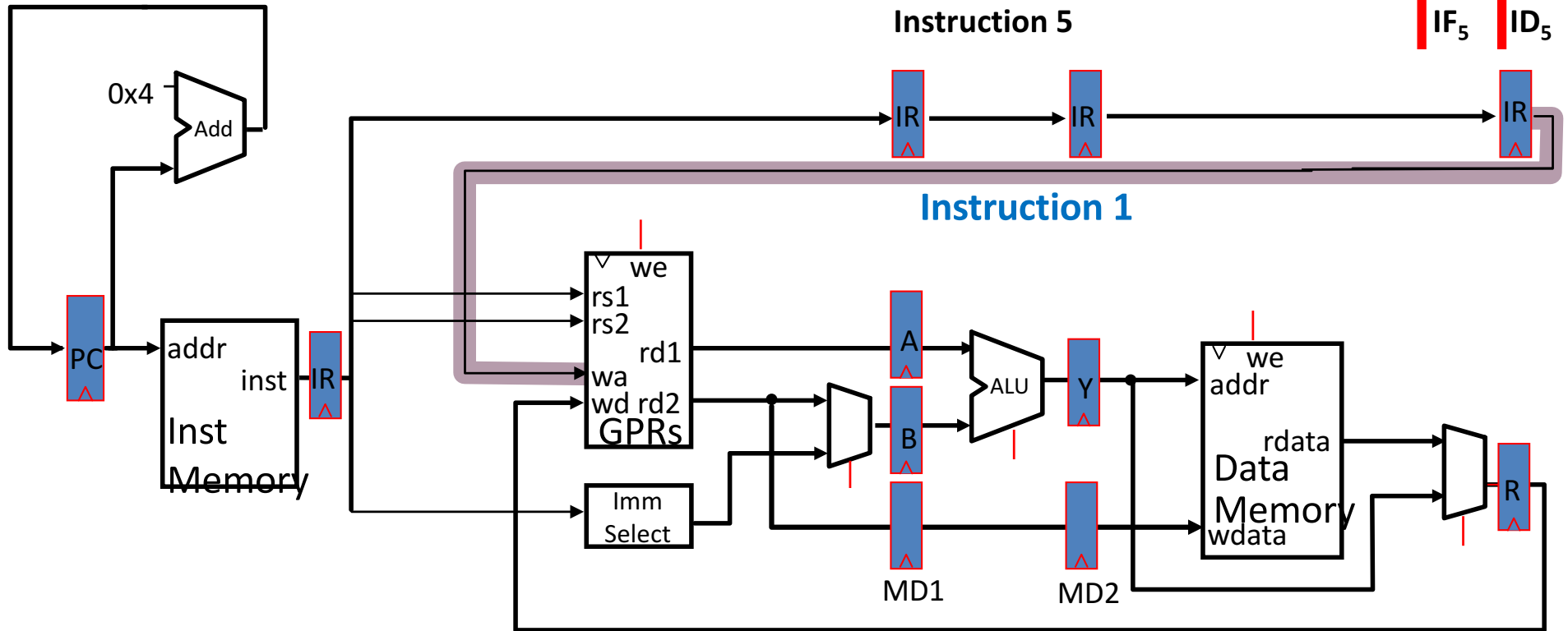
The Whole Pipeline Resources are Used by 5 Instructions in Every Cycle!



<i>time</i>	t0	t1	t2	t3	t4	t5	t6	t7	....
Instruction 1	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	ME <sub>1</sub>	WB <sub>1</sub>				
Instruction 2		IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	ME <sub>2</sub>	WB <sub>2</sub>			
Instruction 3			IF <sub>3</sub>	ID <sub>3</sub>	EX <sub>3</sub>	ME <sub>3</sub>	WB <sub>3</sub>		
Instruction 4				IF <sub>4</sub>	ID <sub>4</sub>	EX <sub>4</sub>	ME <sub>4</sub>	WB <sub>4</sub>	
Instruction 5					IF <sub>5</sub>	ID <sub>5</sub>	EX <sub>5</sub>	ME <sub>5</sub>	WB <sub>5</sub>

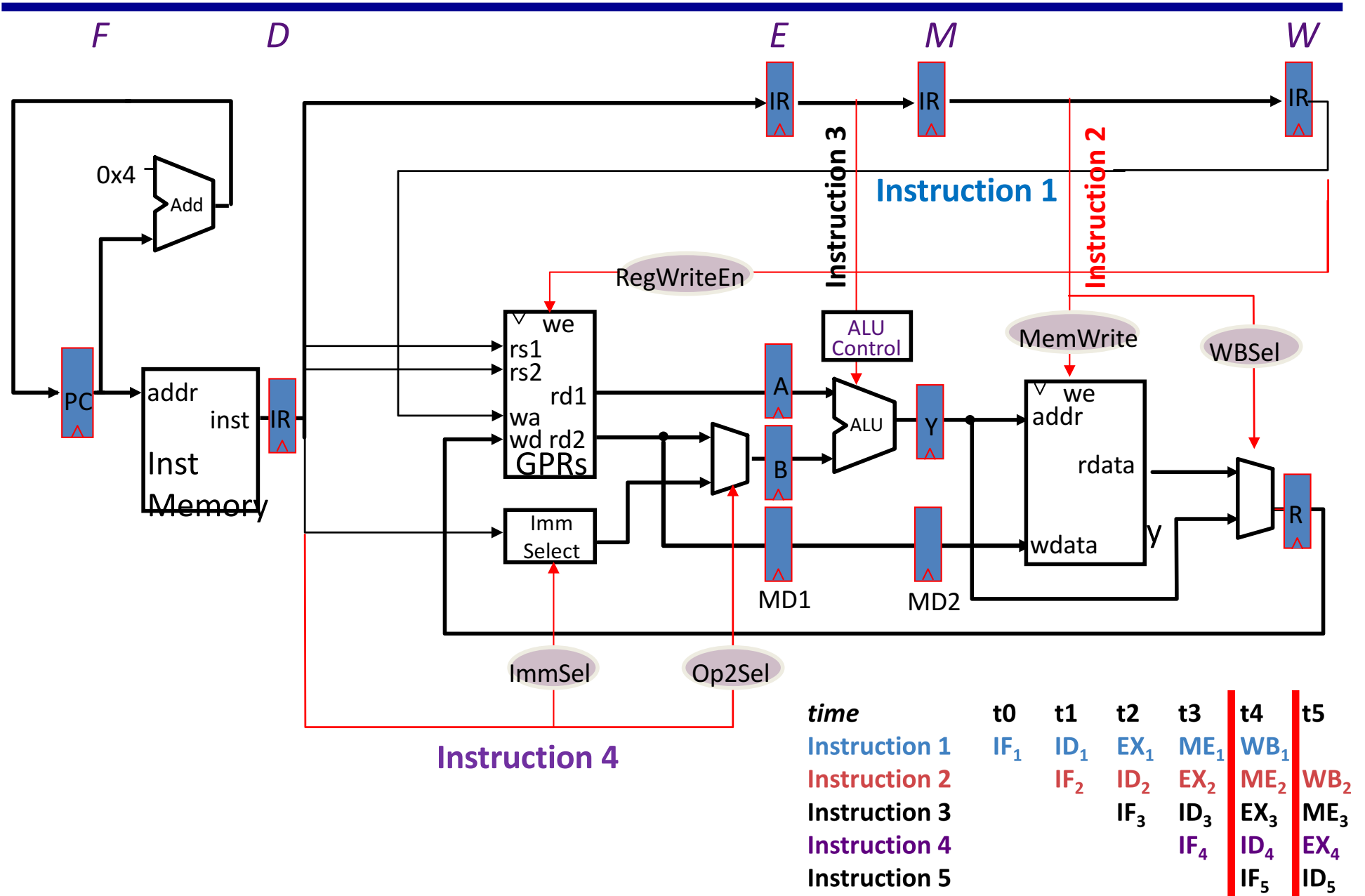
# Instruction Register in Each Stage

time	t0	t1	t2	t3	t4	t5
Instruction 1	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	ME <sub>1</sub>	WB <sub>1</sub>	
Instruction 2		IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	ME <sub>2</sub>	WB <sub>2</sub>
Instruction 3			IF <sub>3</sub>	ID <sub>3</sub>	EX <sub>3</sub>	ME <sub>3</sub>
Instruction 4				IF <sub>4</sub>	ID <sub>4</sub>	EX <sub>4</sub>
Instruction 5					IF <sub>5</sub>	ID <sub>5</sub>



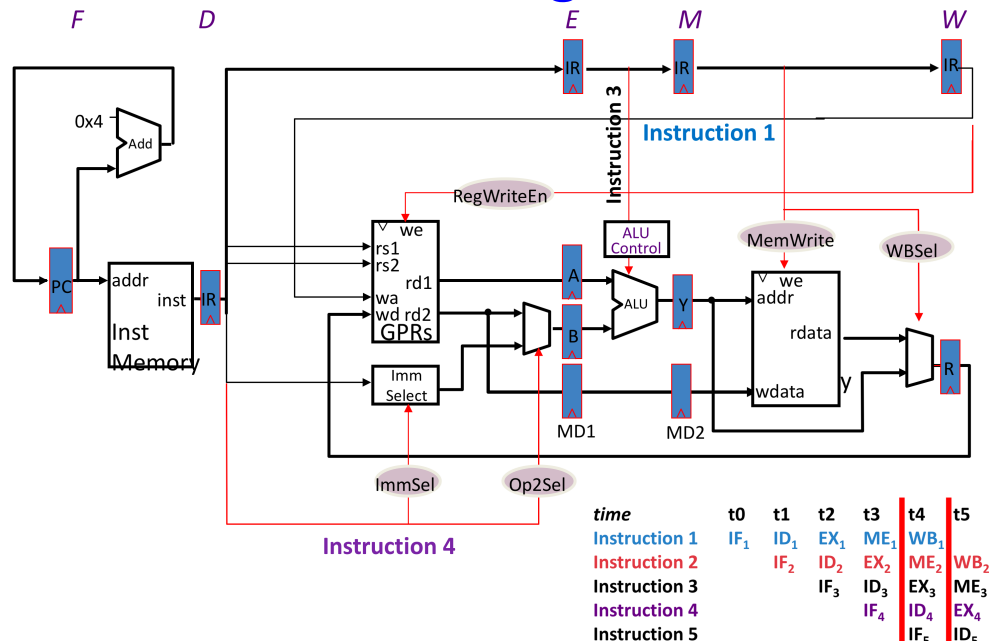
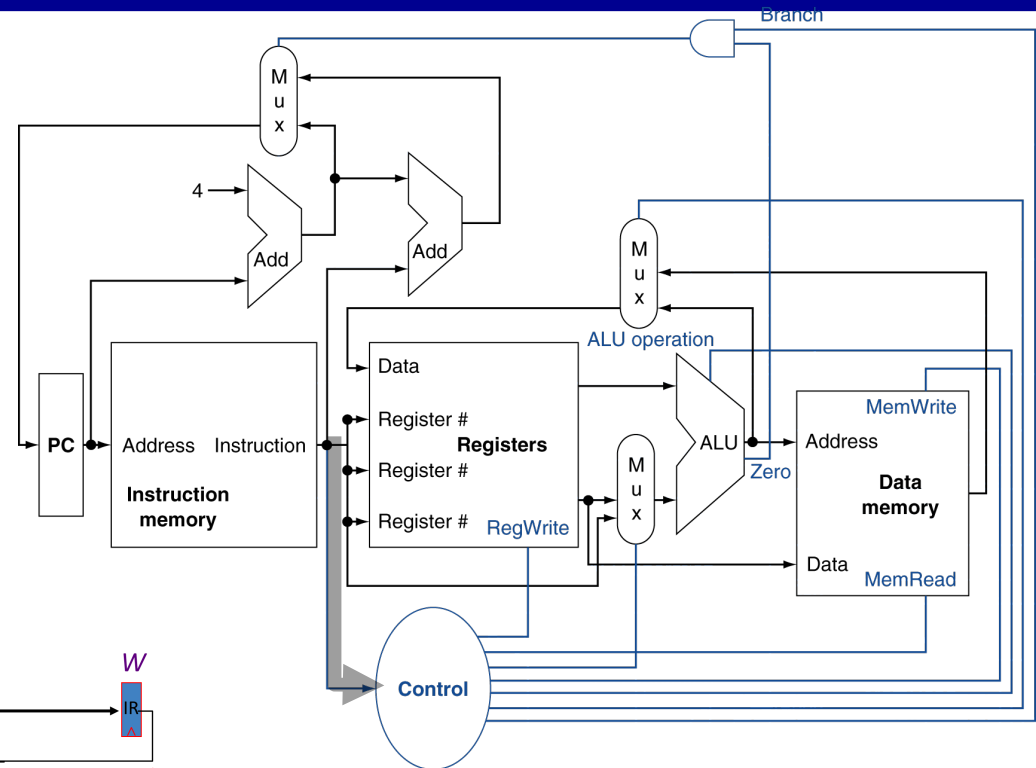
- An instruction Reg (IR) in each stage to contain the instruction in that stage**

# Connect Controls from Instruction Register



# Compared With Control Logic in Unpipelined

- Unpipelined:
  - Single control logic uses the instruction from IF
- Pipelined:
  - Distributed logics that uses instructions from IRs in each stage

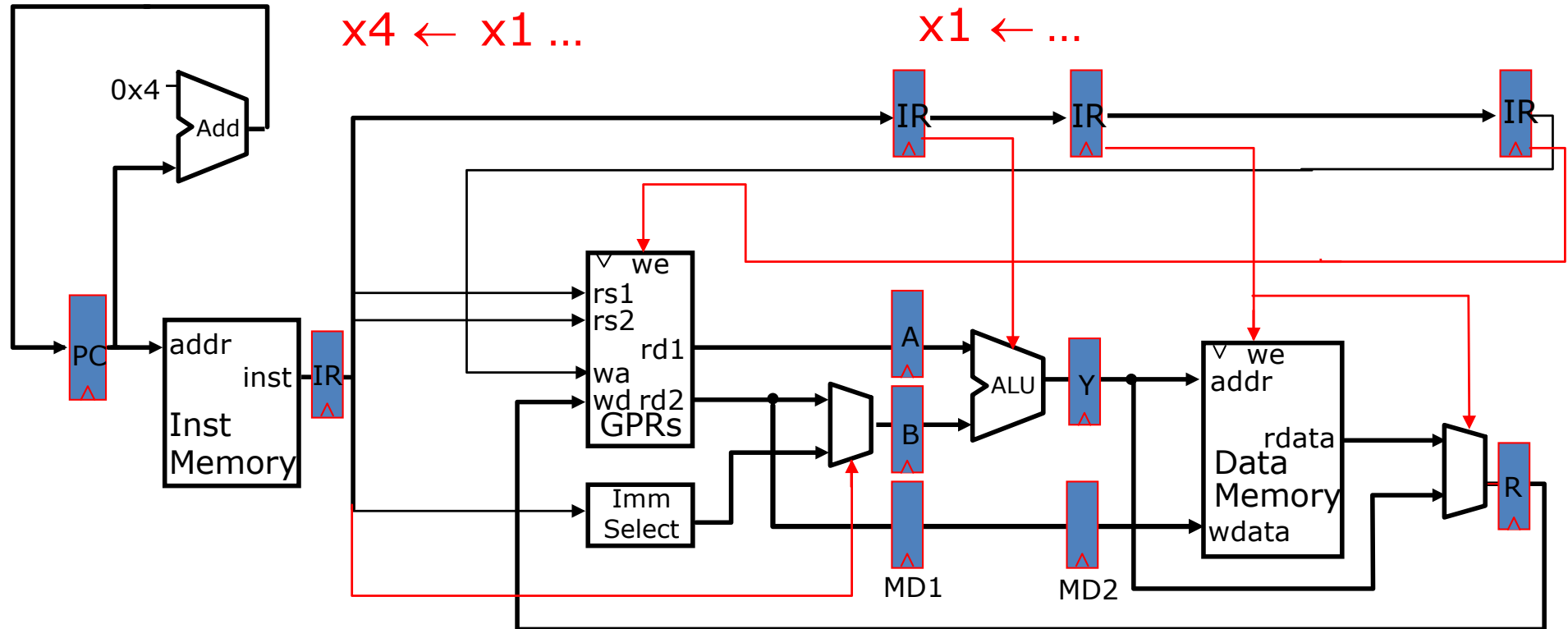


# Instructions Interact With Each Other in Pipeline: Dealing with Hazards

---

- An instruction may need a resource being used by another instruction → *structural hazard*
  - Solution #1: Stalling newer instruction till older instruction finishes
  - Solution #2: Adding more hardware to design
    - E.g., separate memory into I-memory and D-memory
  - Our 5-stage pipeline has no structural hazards by design
- An instruction depends on something produced by an earlier one
  - Dependence may be for a data value or for using same register (not the value) → *data hazard*
    - Solutions for RAW hazards: #1, interlocking (bubble delay), and #2, forwarding
    - WAR and WRW hazards: not possible for 5-stage pipeline
  - Dependence may be for the next instruction's address → *control hazard (branches, exceptions)*
    - Solutions: # delay, prediction, etc

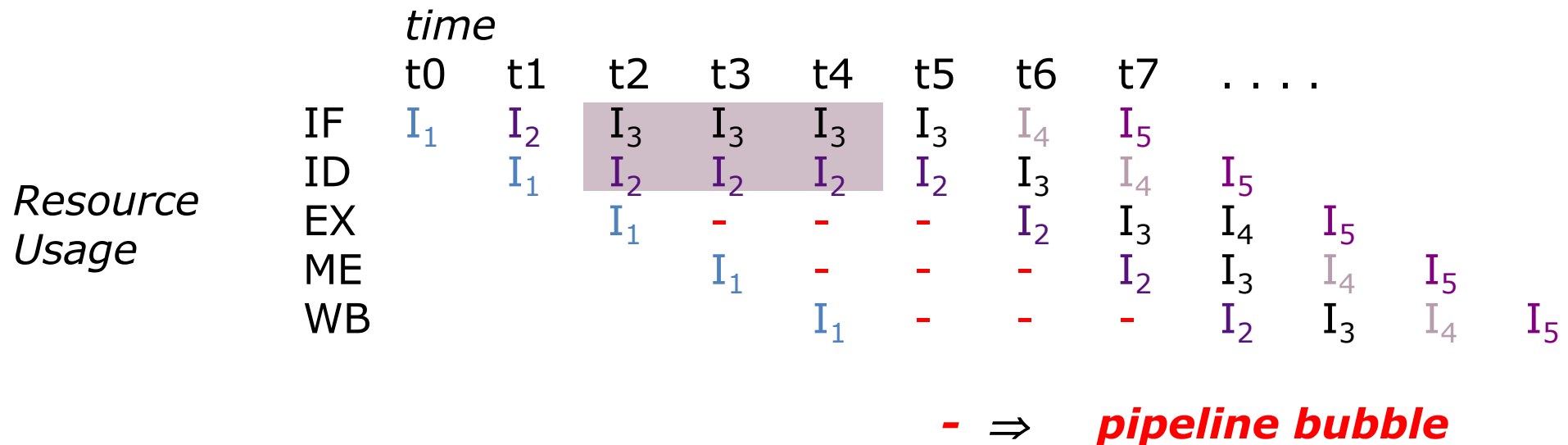
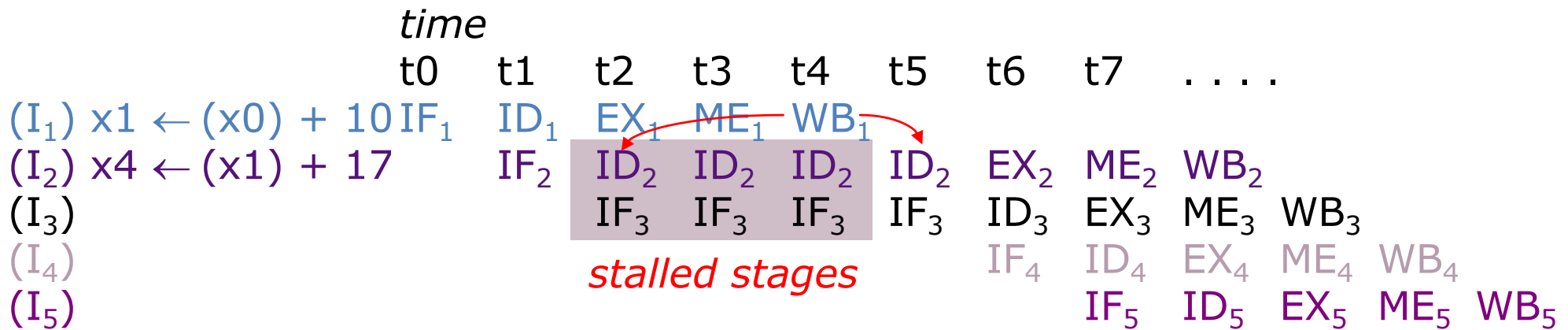
# Read-After-Write (RAW) Data Hazards



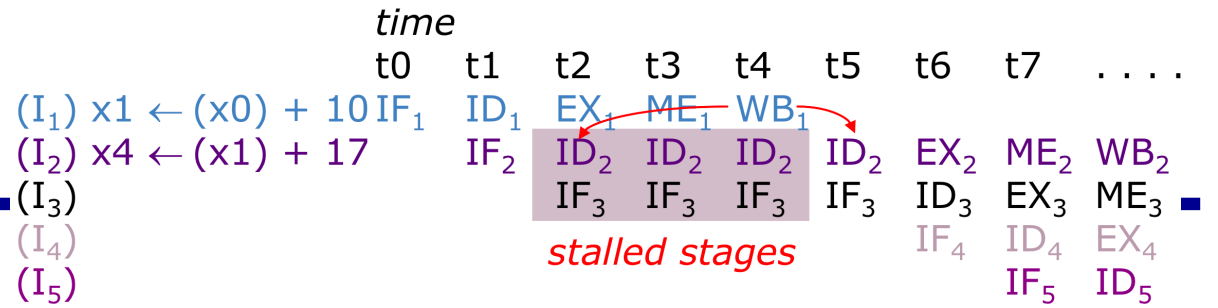
...  
 $x1 \leftarrow x0 + 10$   
 $x4 \leftarrow x1 + 17$   
 ...

**$x1$  in GPRs contains stale value since the passing of value between two instructions has to go through GPRs (register file).**

# To Resolve Data Hazards: #1, Interlocking, i.e. Stall Pipeline by Inserting Bubbles

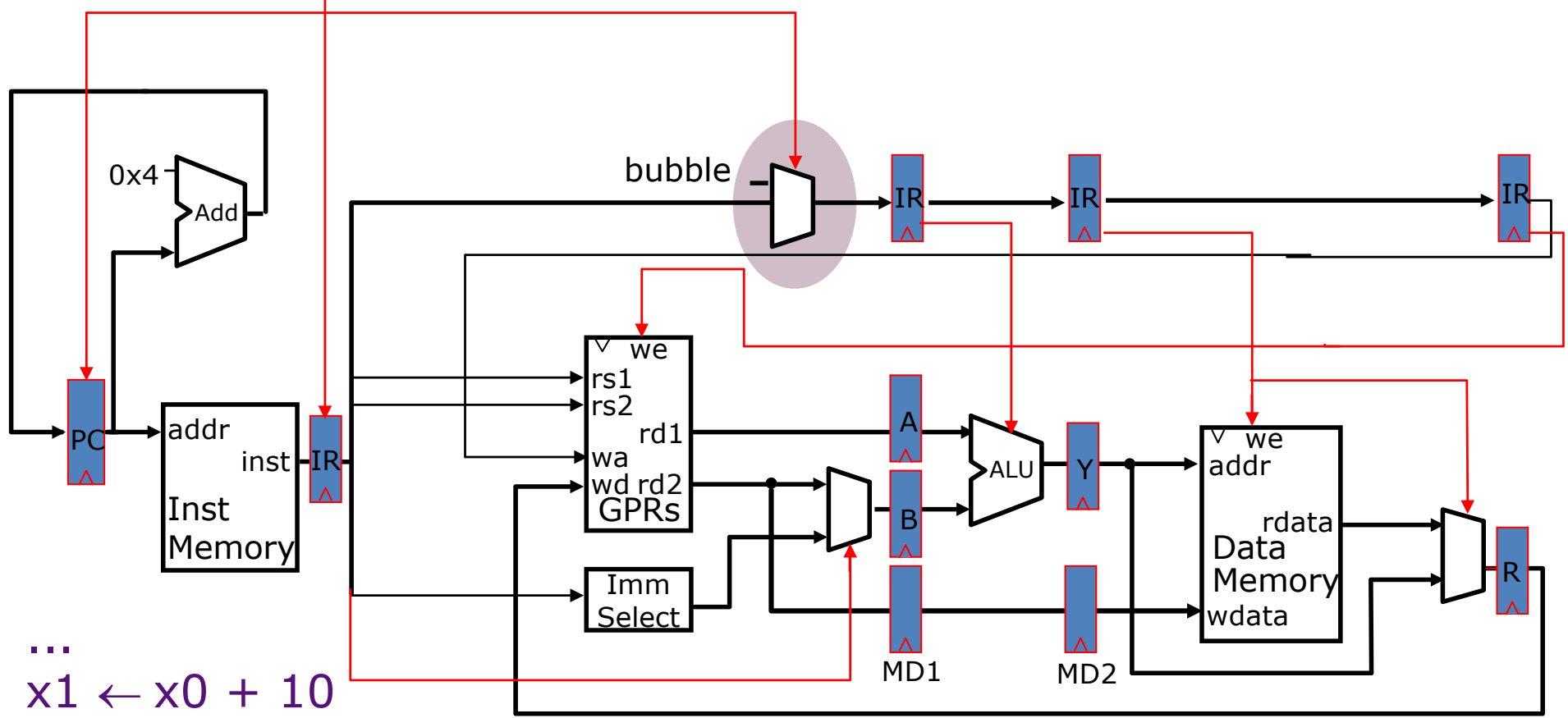


# Insert Bubble for Interlocking



*Stall Condition*

Bubble: Nop instruction, e.g. add x0, x0, x0  
 Bubble are inserted at ID and by entering EXE stage

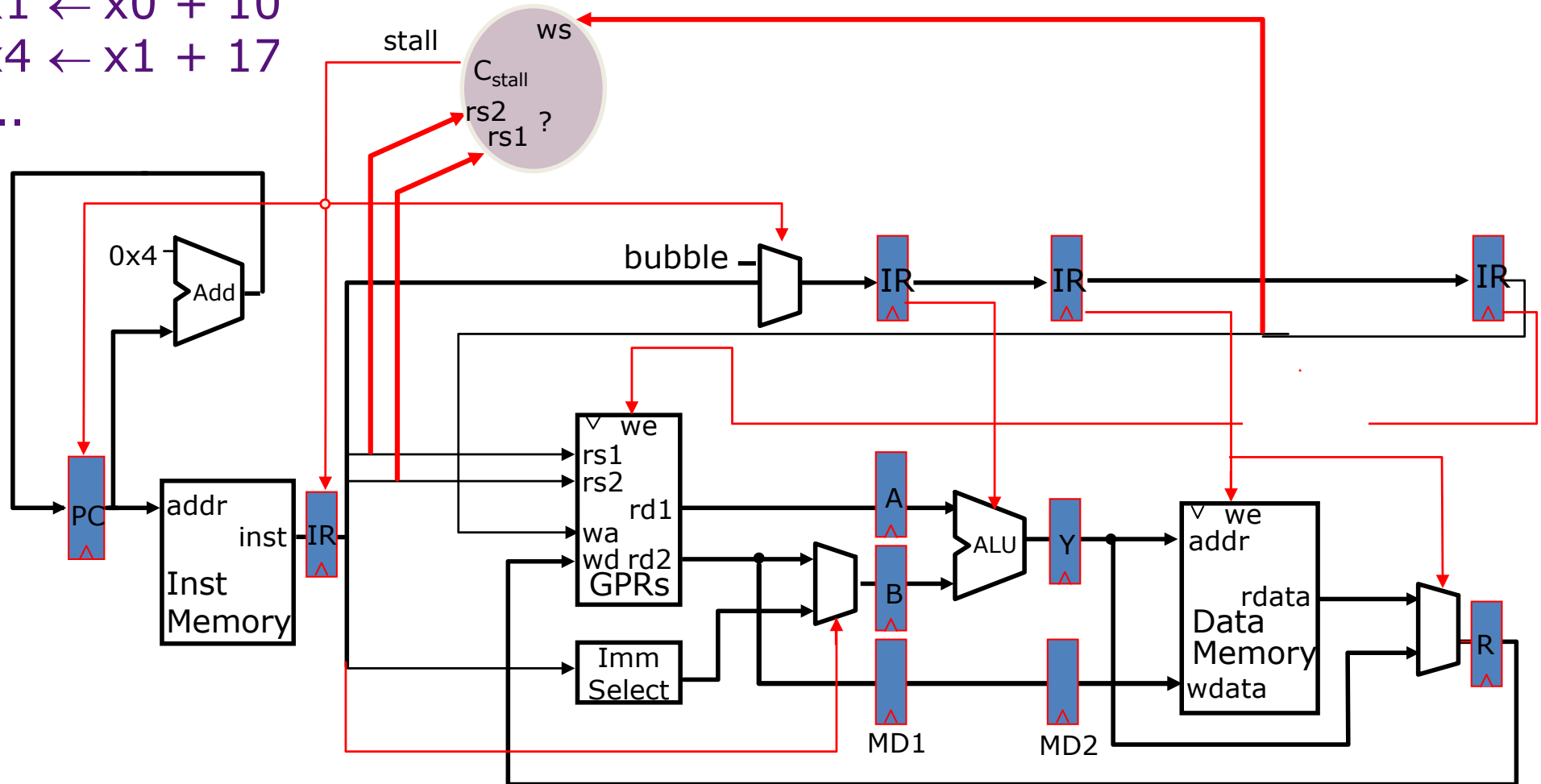


```

...
x1 ← x0 + 10
x4 ← x1 + 17
...
    
```

# Interlock Control Logic

...  
 $x1 \leftarrow x0 + 10$   
 $x4 \leftarrow x1 + 17$   
 ...



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

# Interlock Control Logic

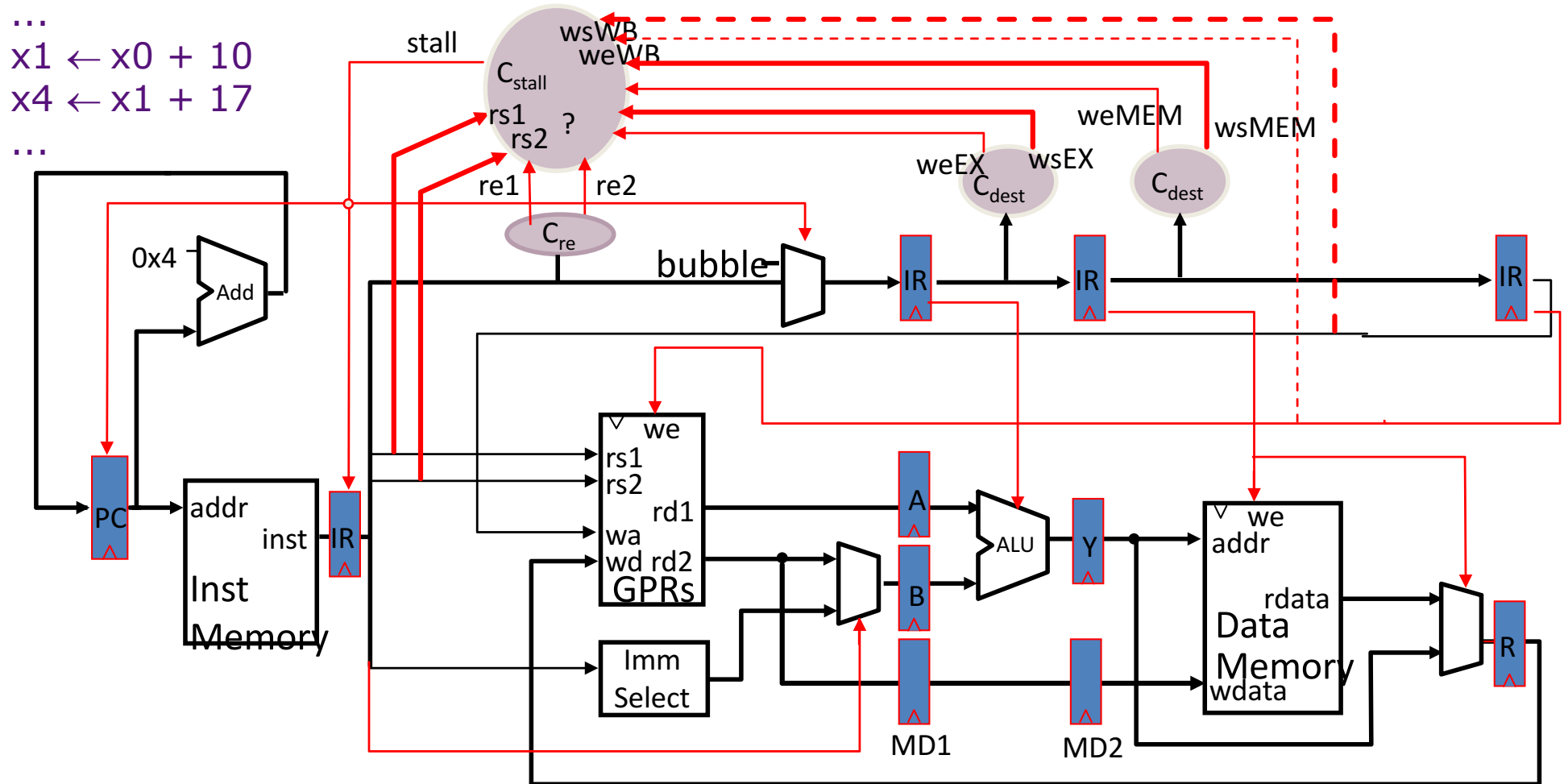
*ignoring jumps & branches*

**we:** write enable, 1-bit on/off

**ws:** write select, 5-bit register number

**re:** read enable, 1-bit on/off

**rs:** read select, 5-bit register number



```

...
x1 ← x0 + 10
x4 ← x1 + 17
...

```

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



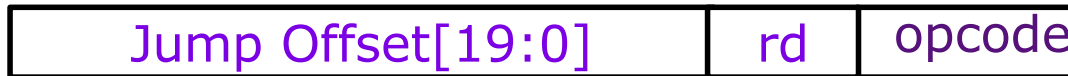
ALU



ALUI/LW/JALR

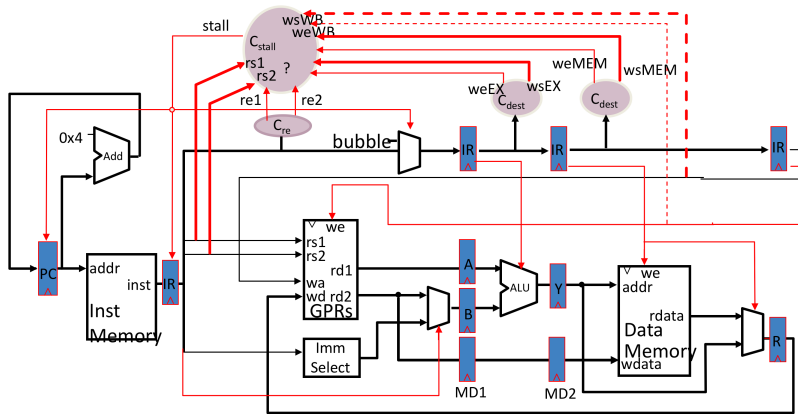


SW/Bcond



		<i>source(s)</i>	<i>destination</i>
<b>ALU</b>	$rd \leftarrow rs1 \text{ func10 } rs2$	rs1, rs2	rd
<b>ALUI</b>	$rd \leftarrow rs1 \text{ op } imm$	rs1	rd
<b>LW</b>	$rd \leftarrow M [rs1 + imm]$	rs1	rd
<b>SW</b>	$M [rs1 + imm] \leftarrow rs2$	rs1, rs2	-
<b>Bcond</b>	rs1,rs2 <i>true:</i> $PC \leftarrow PC + imm$ <i>false:</i> $PC \leftarrow PC + 4$	rs1, rs2	-
<b>JAL</b>	$x1 \leftarrow PC, PC \leftarrow PC + imm$	-	rd
<b>JALR</b>	$rd \leftarrow PC, PC \leftarrow rs1 + imm$	rs1	rd

# Deriving the Stall Signal



		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	-
Bcond	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

$C_{re}$

re1 = Case opcode

ALU, ALUi, LW, SW, Bcond, JALR => on  
JAL => off

re2 = Case opcode

ALU, SW, Bcond => on

... => off

No need the WB for interlock control since we only need to deal with hazard between MEM-EXE and EXE-EXE. For two instructions which are in WB and EXE, and have RAW hazard, the dependency are handled through the register file.

$C_{dest}$

ws = rd

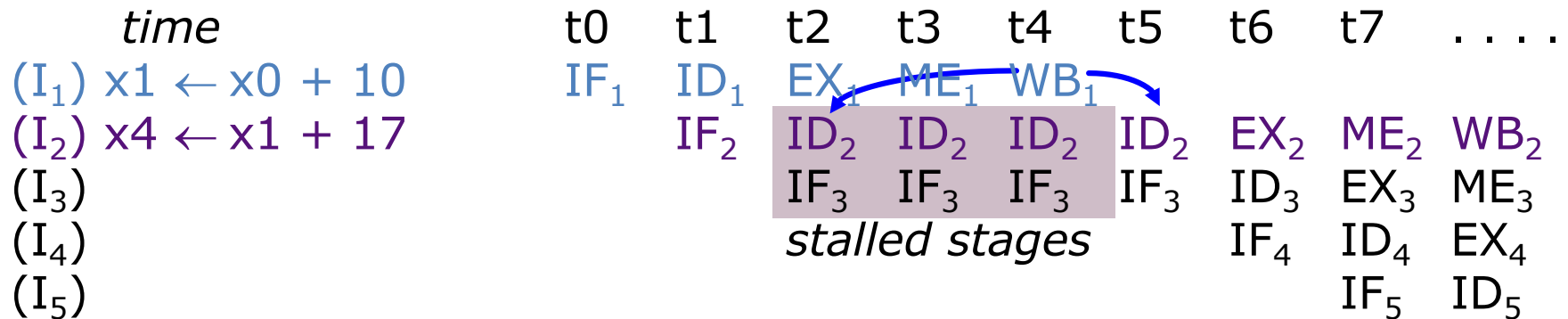
we = Case opcode

ALU, ALUi, LW, JALR => on

... => off

$$C_{stall} \text{ stall} = ((rs1_D == ws_{EX}) \&\& we_{EX} + (rs1_D == ws_{MEM}) \&\& we_{MEM} + \cancel{(rs1_D == ws_{WB}) \&\& we_{WB}}) \&\& re1_D + ((rs2_D == ws_{EX}) \&\& we_{EX} + (rs2_D == ws_{MEM}) \&\& we_{MEM} + \cancel{(rs2_D == ws_{WB}) \&\& we_{WB}}) \&\& re2_D$$

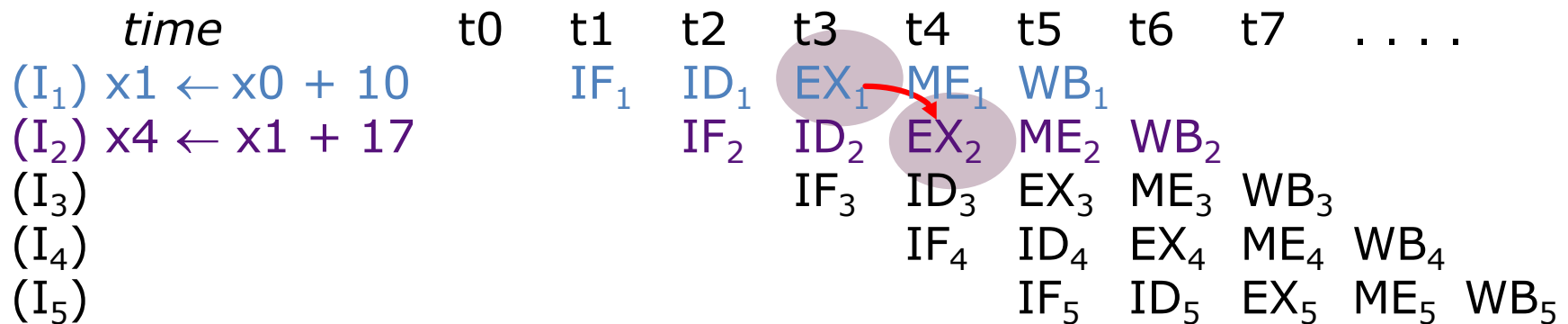
## To Resolve Data Hazards: #2, Forwarding (Bypassing)



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

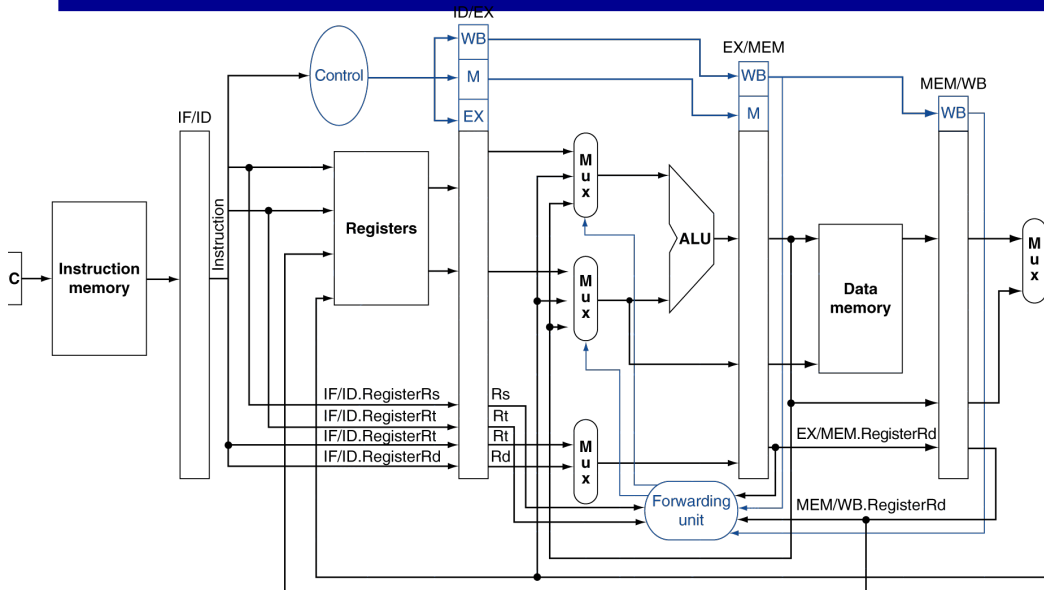
$$\Rightarrow CPI > 1$$

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



# Review: Hardware Support for Forwarding, and Detecting RAW Hazards with Previous and 2<sup>nd</sup> Previous Instructions

- Slide 48 of lecture05-06

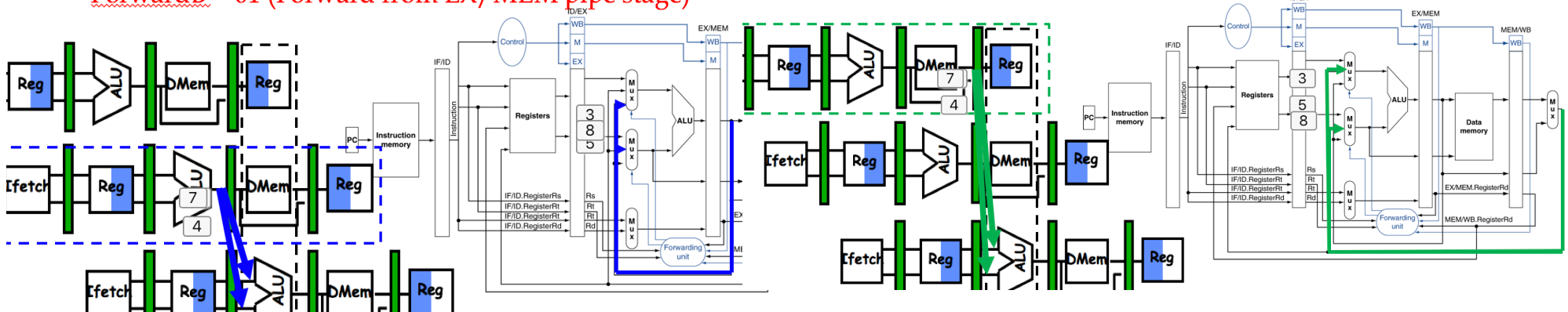


## ▣ Detecting RAW hazard with Previous Instruction

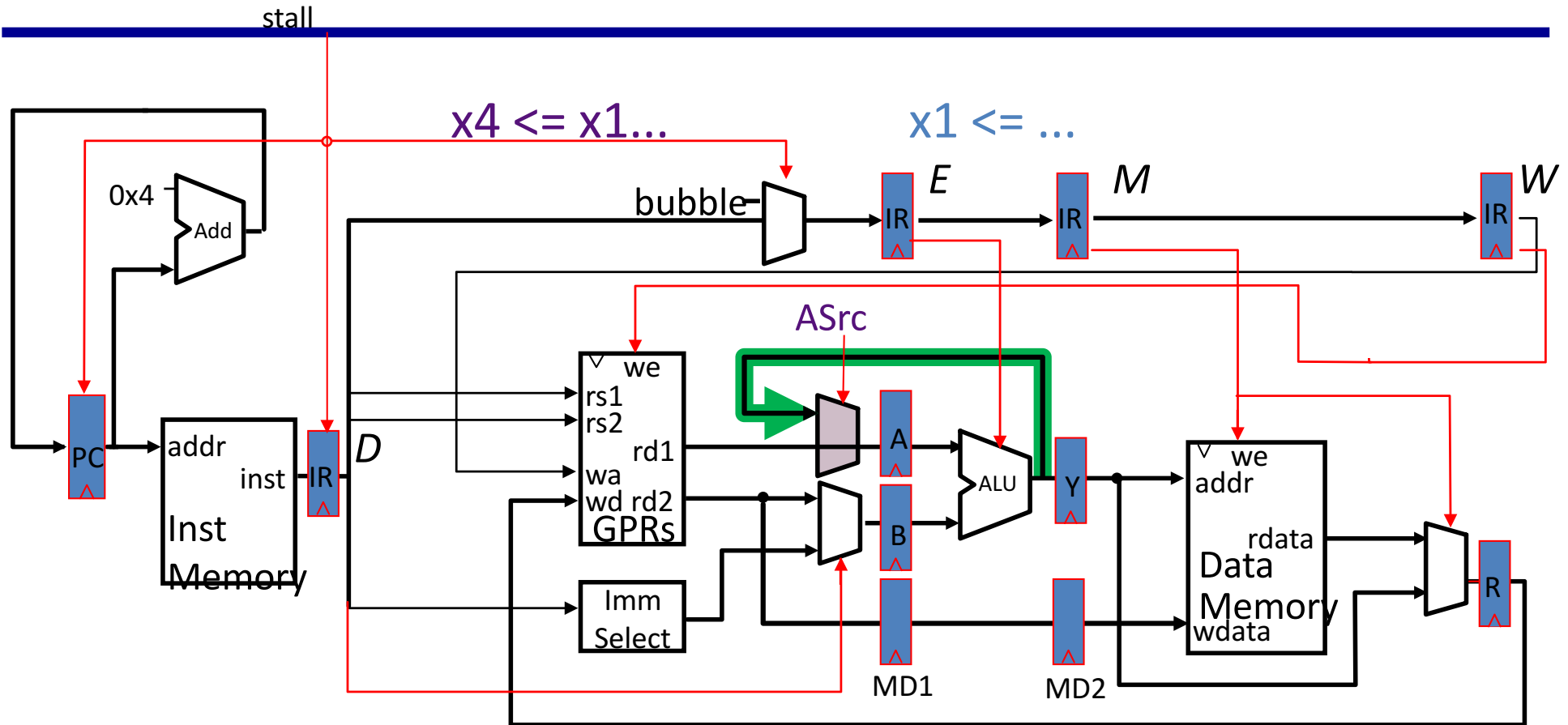
- ◆ if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
ForwardA = 01 (Forward from EX/MEM pipe stage)
- ◆ if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
ForwardB = 01 (Forward from EX/MEM pipe stage)

## ▣ Detecting RAW hazard with Second Previous

- ◆ if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
ForwardA = 10 (Forward from MEM/WB pipe stage)
- ◆ if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
ForwardB = 10 (Forward from MEM/WB pipe stage)



# Adding a Bypass (To Bypass Register Files)



**When does this bypass help?**

...  
 (I<sub>1</sub>)  $x1 \leq x0 + 10$   
 (I<sub>2</sub>)  $x4 \leq x1 + 17$   
**Yes**

$x1 \leq M[x0 + 10]$   
 $x4 \leq x1 + 17$   
**No, Load  $\rightarrow$  EXE-Use**

JAL 500  
 $x4 \leq x1 + 17$   
**No**

# The Bypass Signal: *Deriving it from the Stall Signal*

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

$$ws = rd$$

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

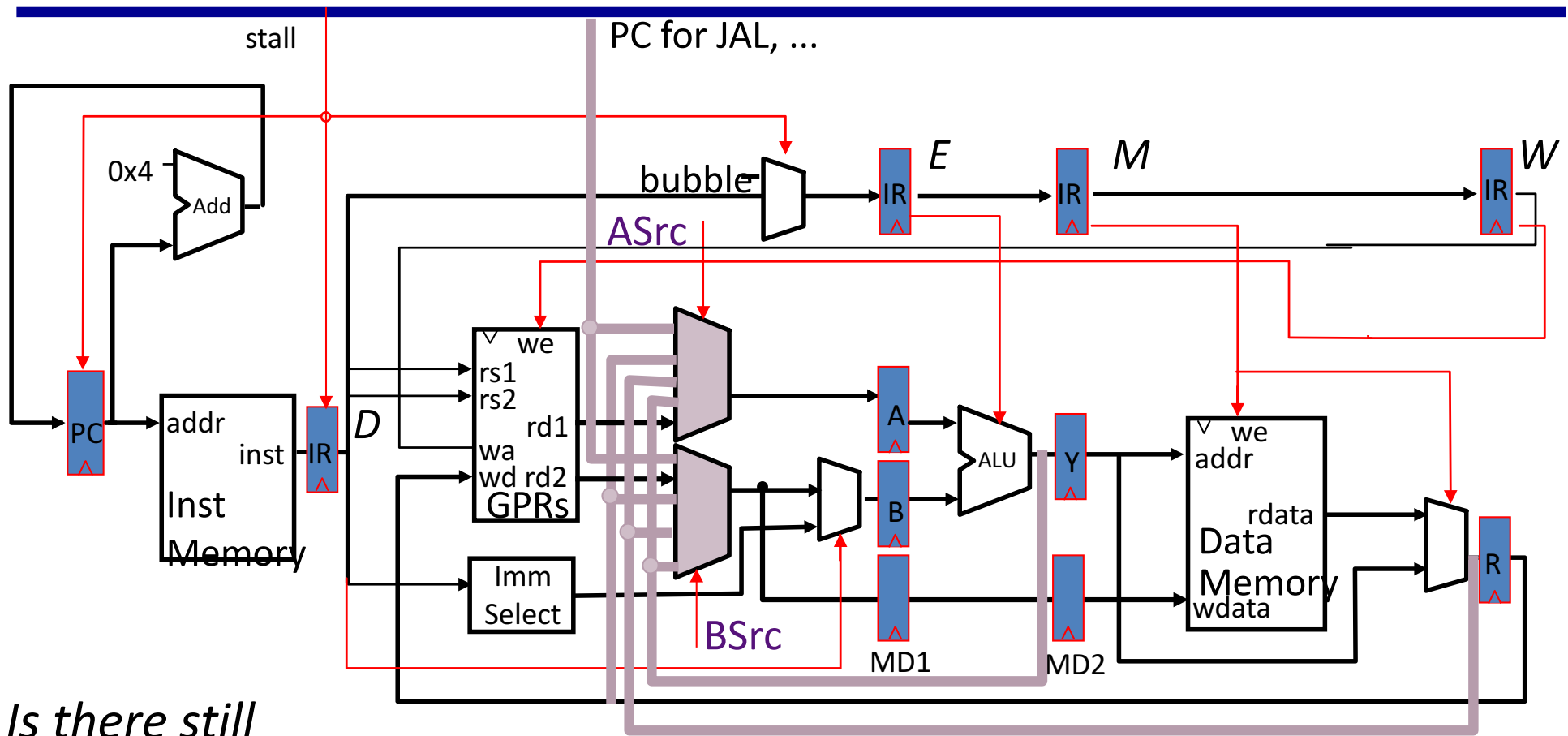
$we\_bypass_E = \text{Case opcode}_E$   
ALU, ALUi => on  
... => off

$we\_stall_E = \text{Case opcode}_E$   
LW, JAL, JALR=> on  
JAL => on  
... => off

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

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

# Fully Bypassed Datapath

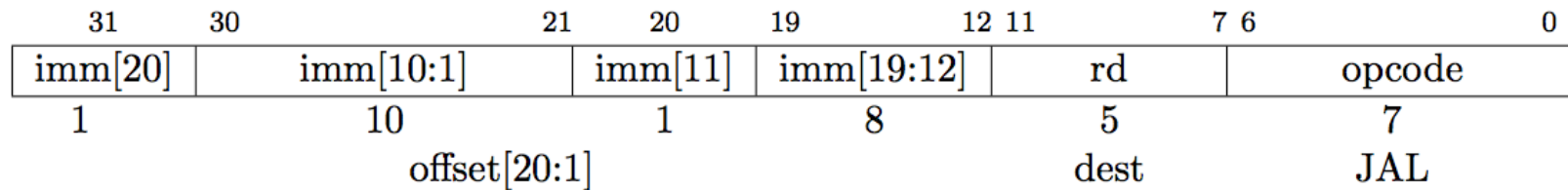


Is there still  
a need for the  
stall signal ?

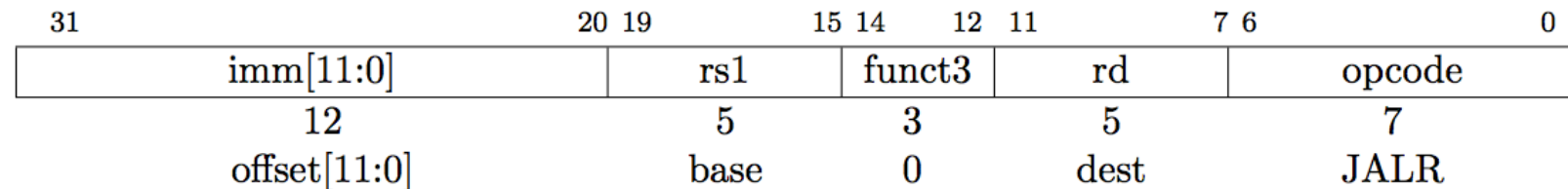
$$\text{stall} = (\text{rs1}_D == \text{ws}_E) \&\& (\text{opcode}_E == \text{LW}_E) \&\& (\text{ws}_E \neq 0) \&\& \text{re1}_D \\ + (\text{rs2}_D == \text{ws}_E) \&\& (\text{opcode}_E == \text{LW}_E) \&\& (\text{ws}_E \neq 0) \&\& \text{re2}_D$$

# Control Hazards: Branches and Jumps

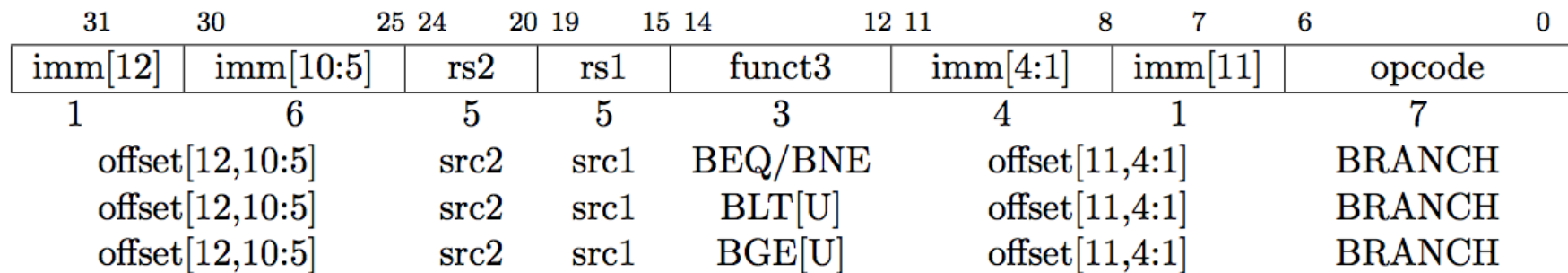
- **JAL: unconditional jump to PC+immediate**



- **JALR: indirect jump to rs1+immediate**



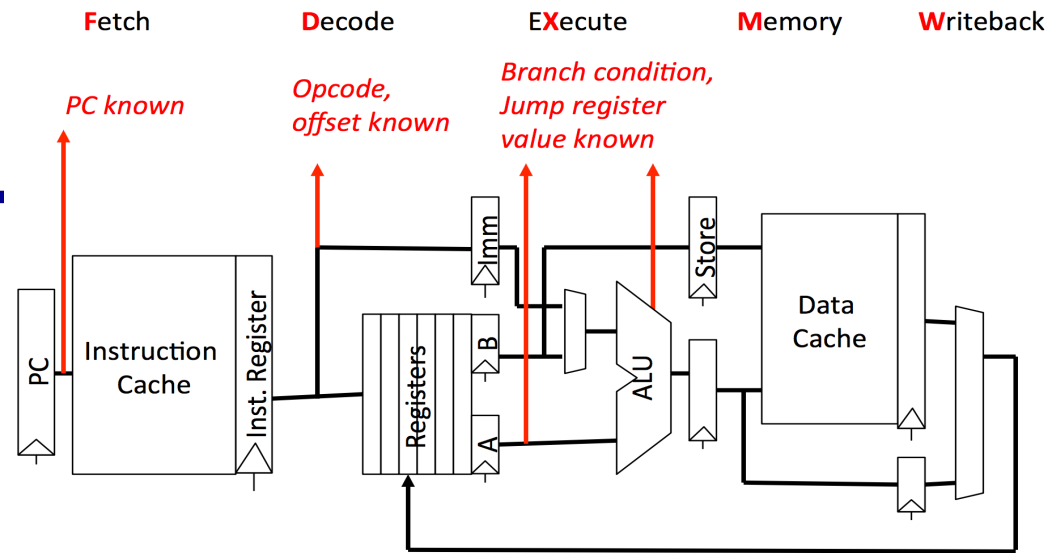
- **Branch: if (rs1 conds rs2), branch to PC+immediate**



# Info for Control Transfer

Two pieces of info:

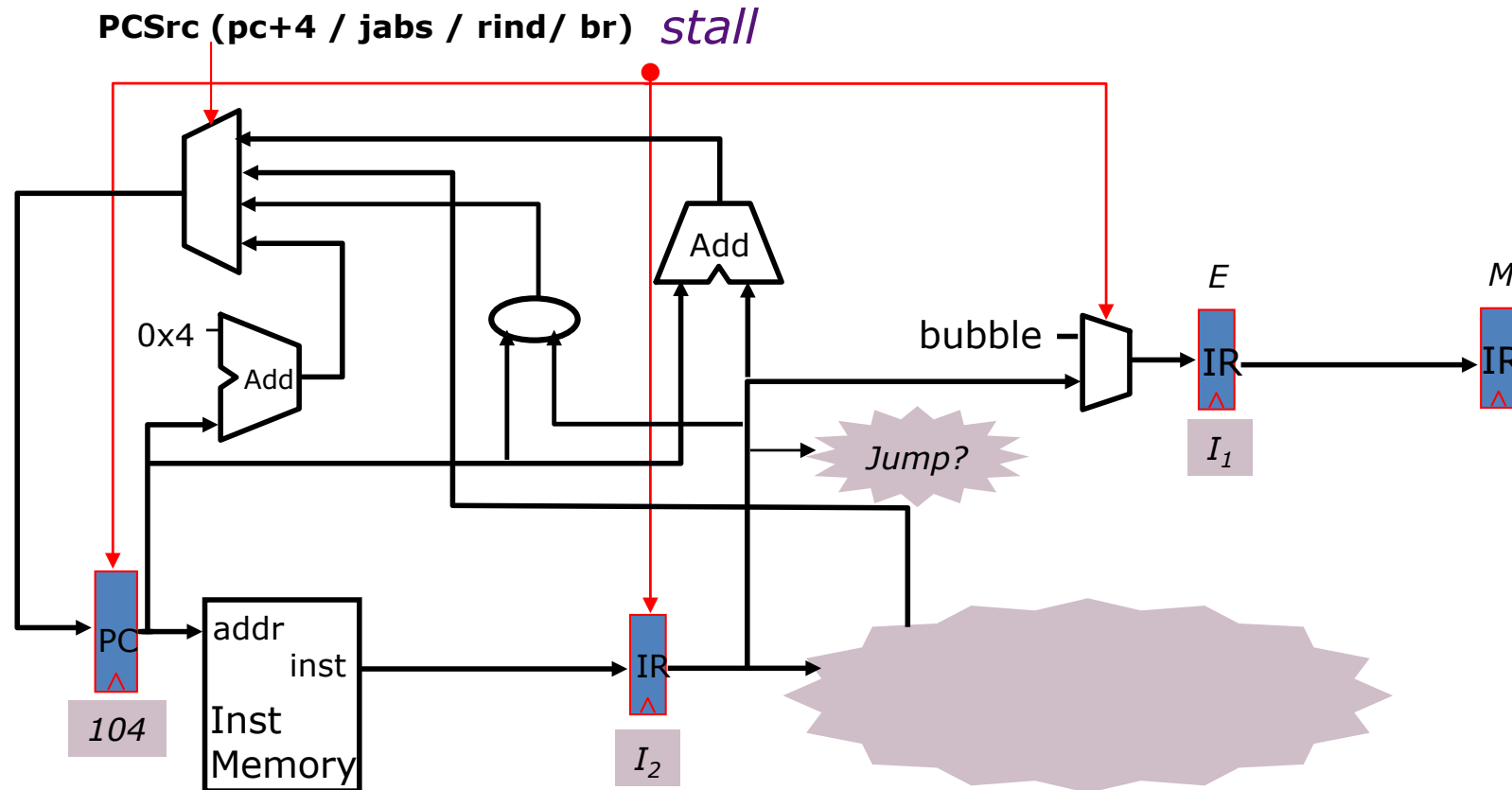
1. Taken or Not Taken
2. Target address?



- **JAL:** unconditional jump to **PC+immediate**
- **JALR:** indirect jump to **rs1+immediate**
- Branch: if (**rs1 conds rs2**), branch to **PC+immediate**

<i>Instruction</i>	<i>Taken known?</i>	<i>Target known?</i>
JAL	After Inst. Decode	After Inst. Decode
JALR	After Inst. Decode	After Reg. Fetch
B<cond.>	After Execute	After Inst. Decode

# Speculate Next Address is PC+4



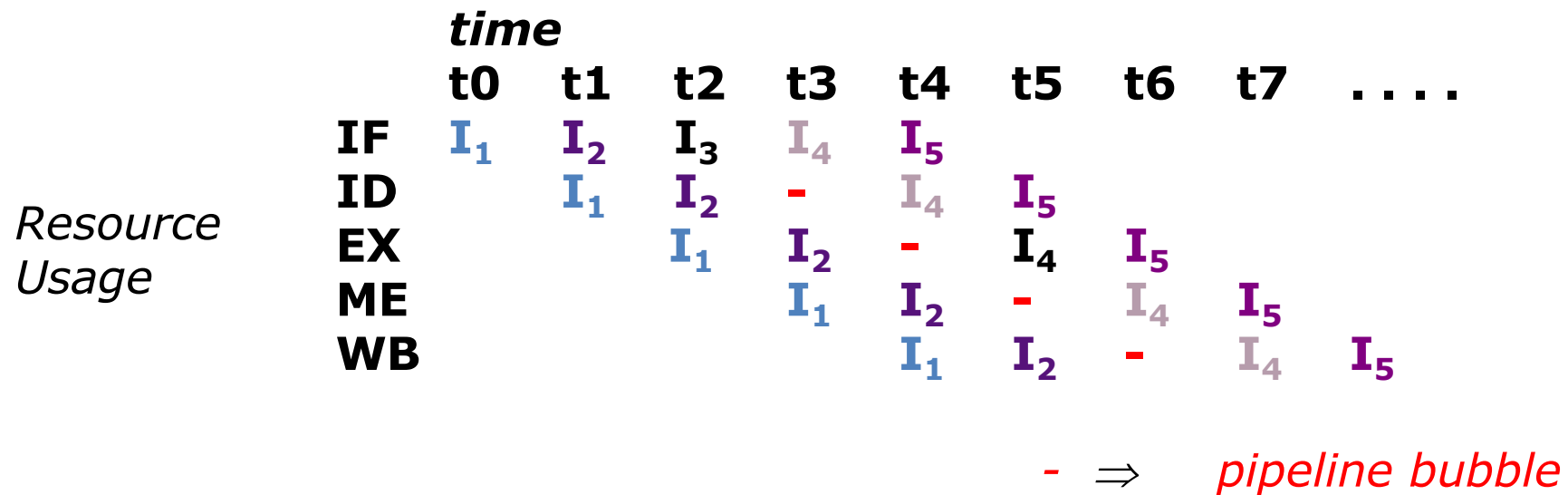
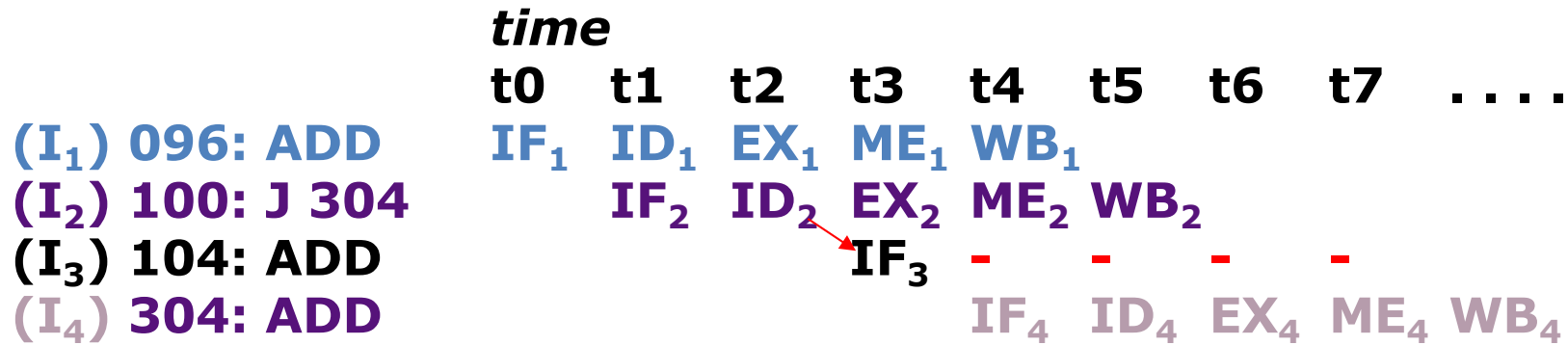
$I_1$	096	ADD	
$I_2$	100	J 304	
$I_3$	<del>104</del>	<del>ADD</del>	<i>kill</i>
$I_4$	304	ADD	

**A jump instruction kills (not stalls) the following instruction**

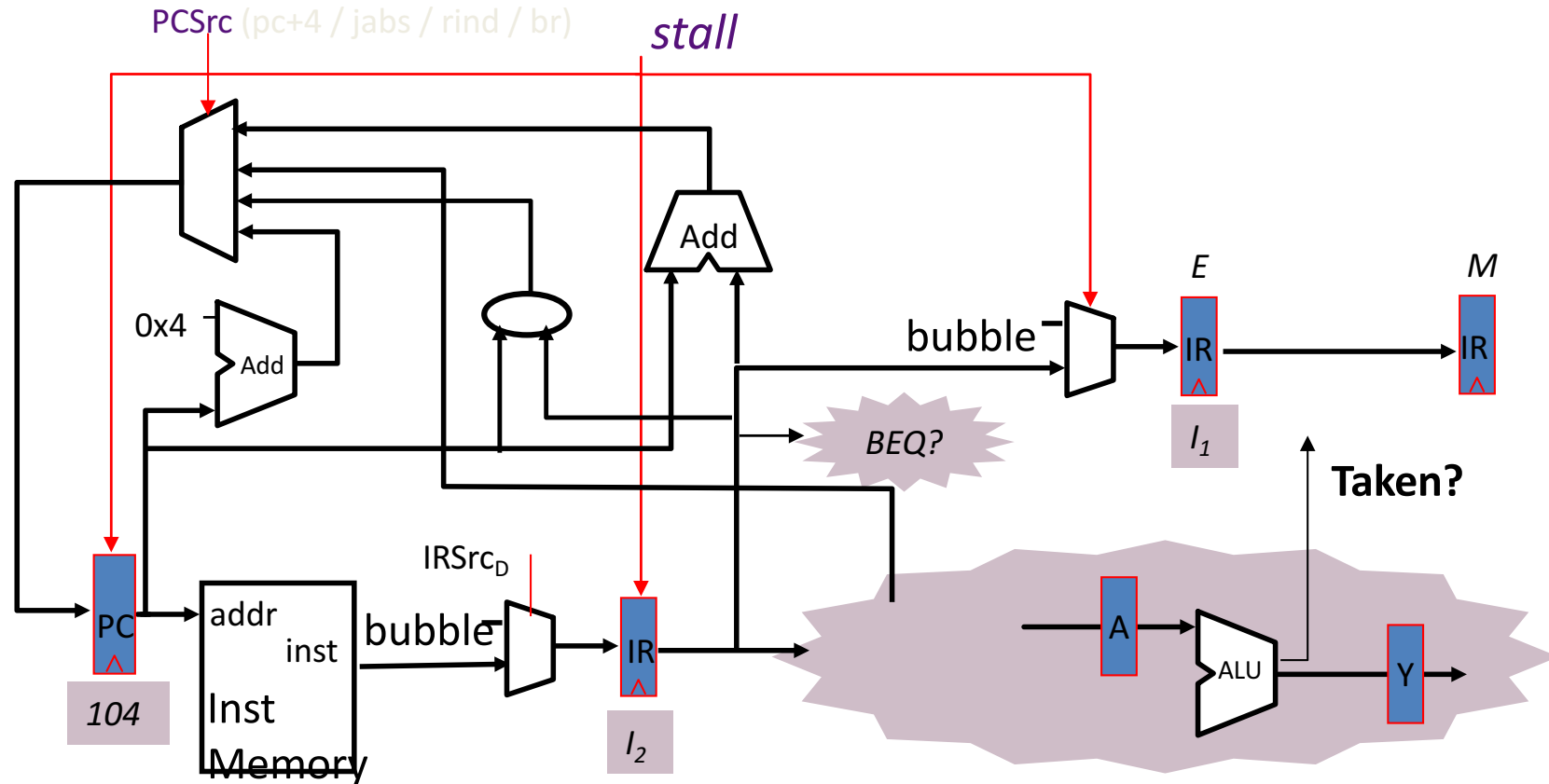
*How?*



# Jump Pipeline Diagrams



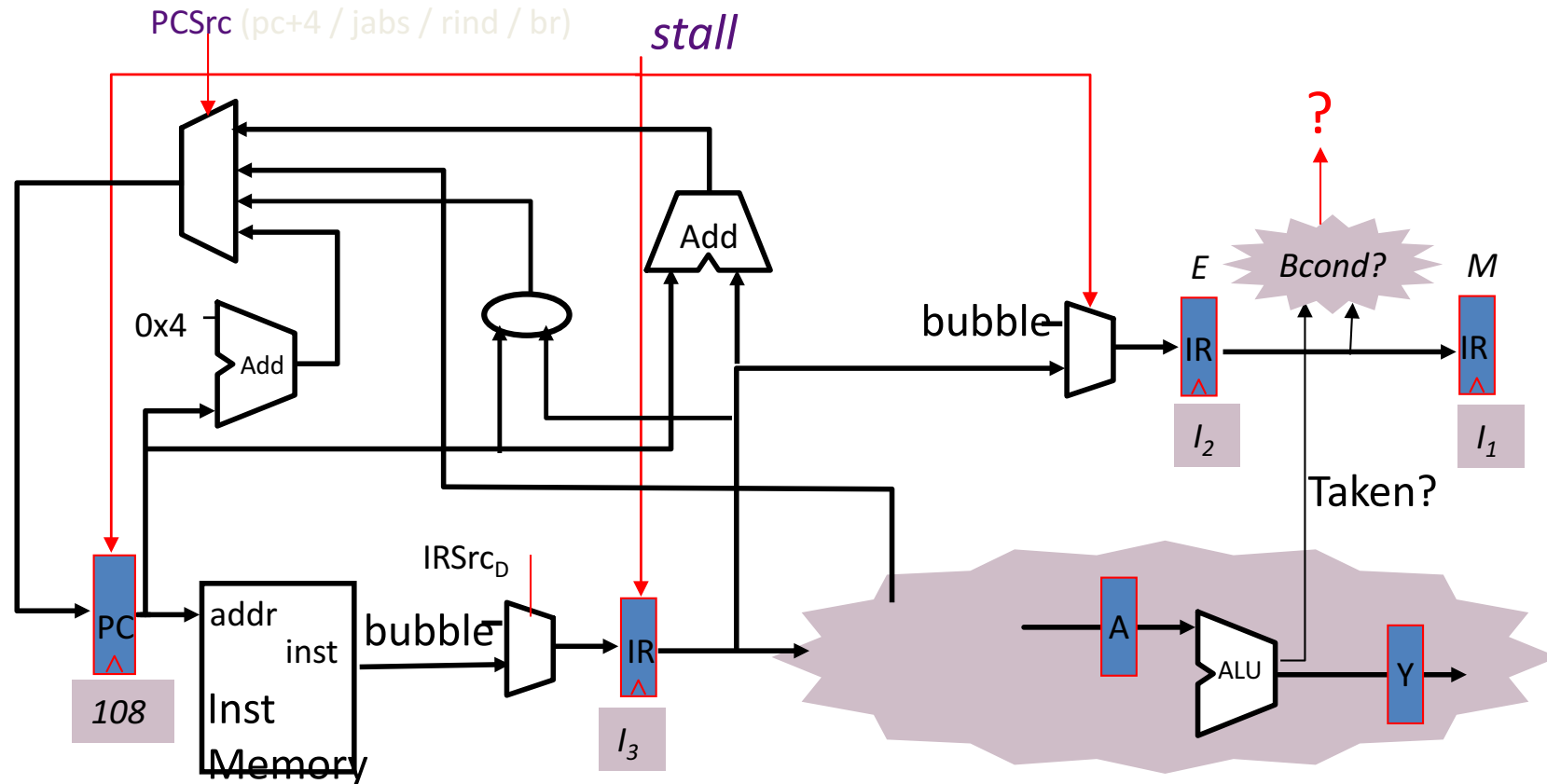
# Pipelining Conditional Branches



- $I_1$  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**

# Pipelining Conditional Branches

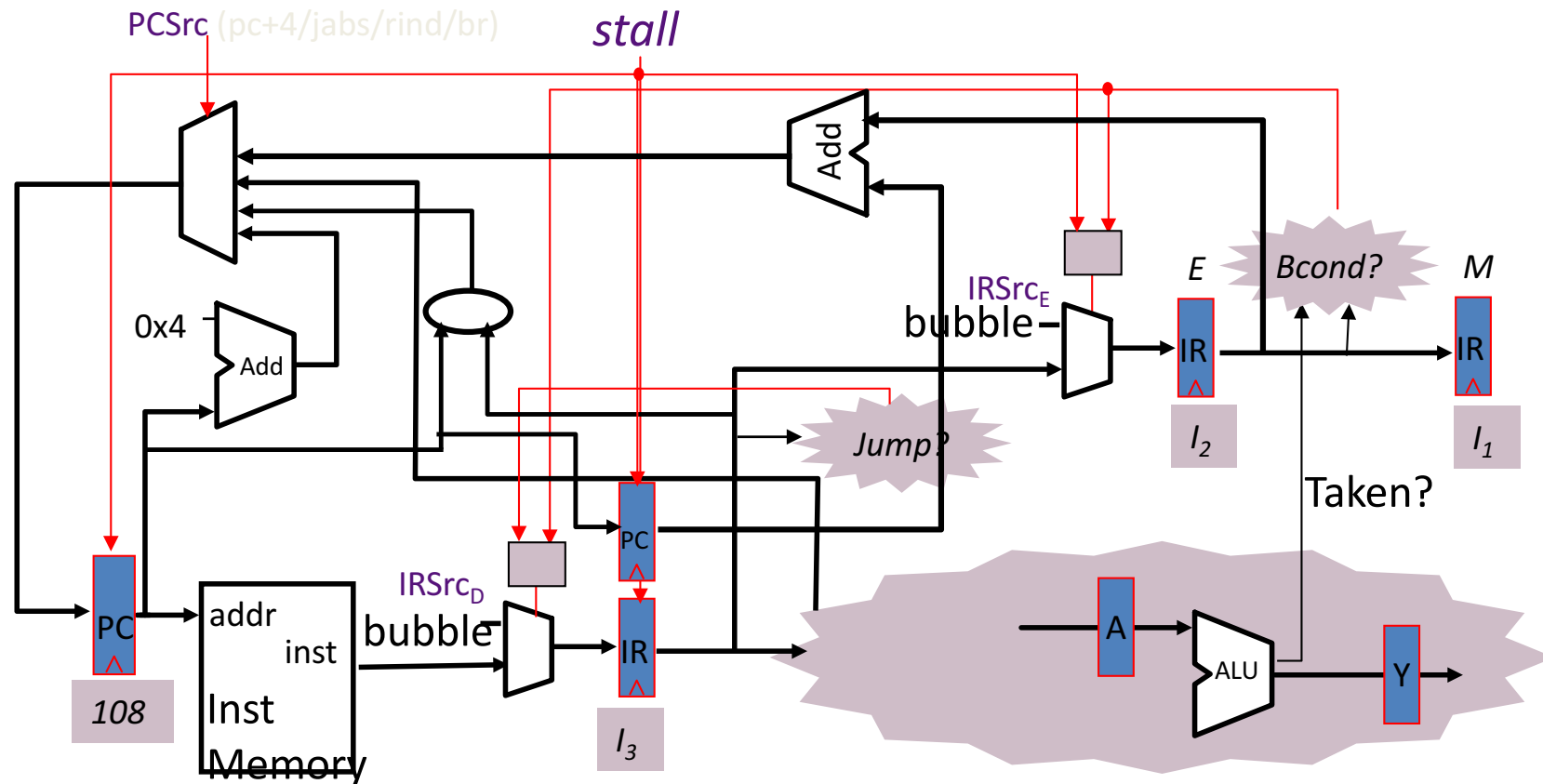


- I<sub>1</sub> 096 ADD
- I<sub>2</sub> 100 BEQ x1,x2 +200
- I<sub>3</sub> 104 ADD
- I<sub>4</sub> 304 ADD

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



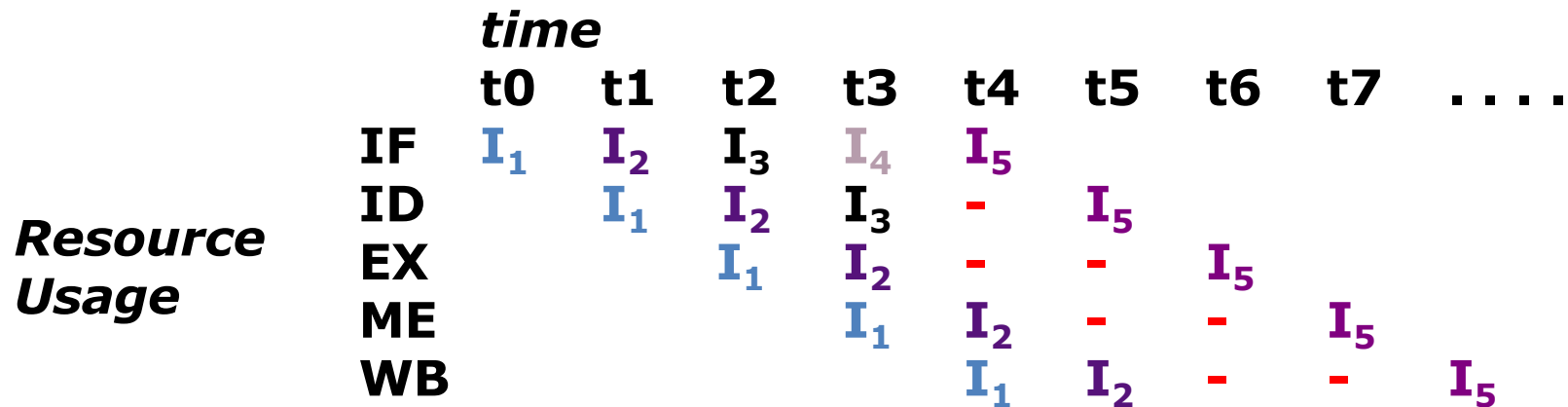
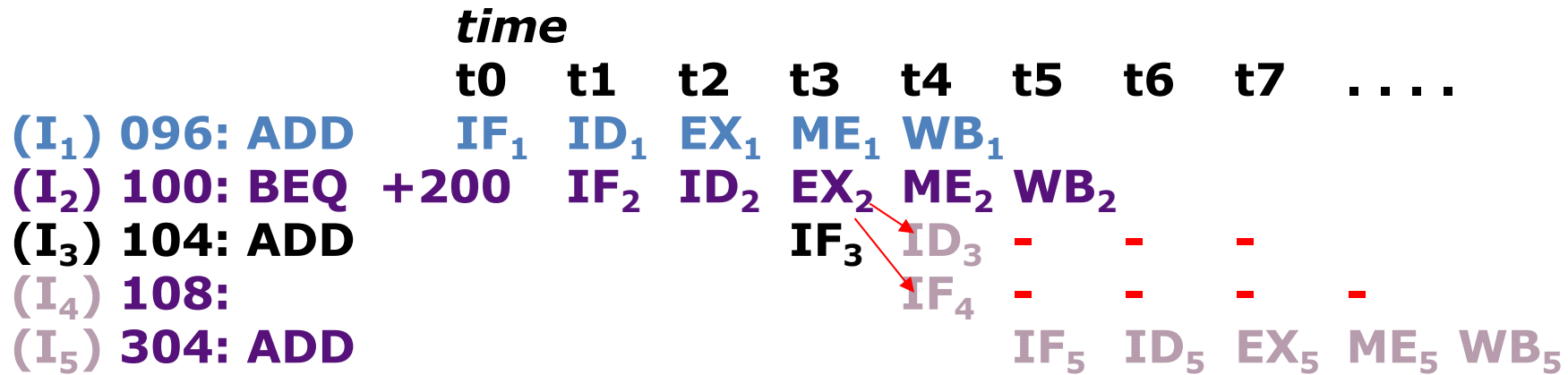
- I<sub>1</sub>: 096 ADD
- I<sub>2</sub>: 100 BEQ x1,x2 +200
- I<sub>3</sub>: 104 ADD
- I<sub>4</sub>: 304 ADD

If the branch is taken

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

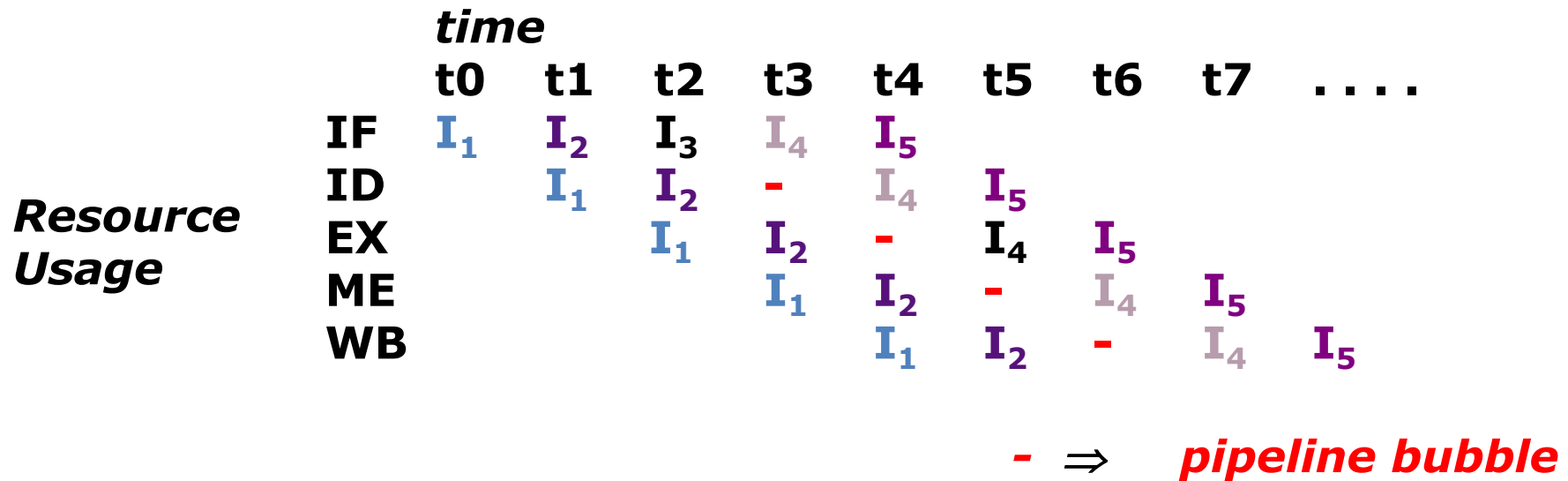
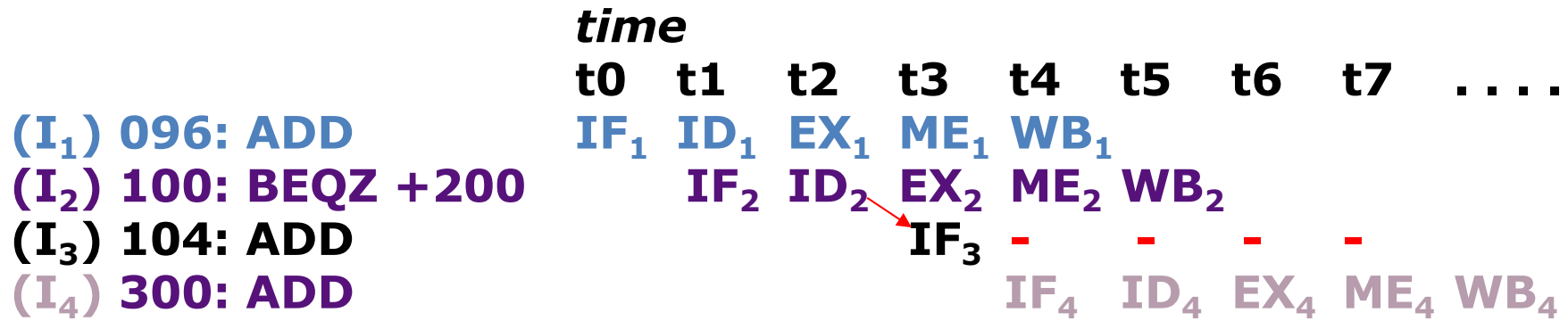
# Branch Pipeline Diagrams

(resolved in execute stage)

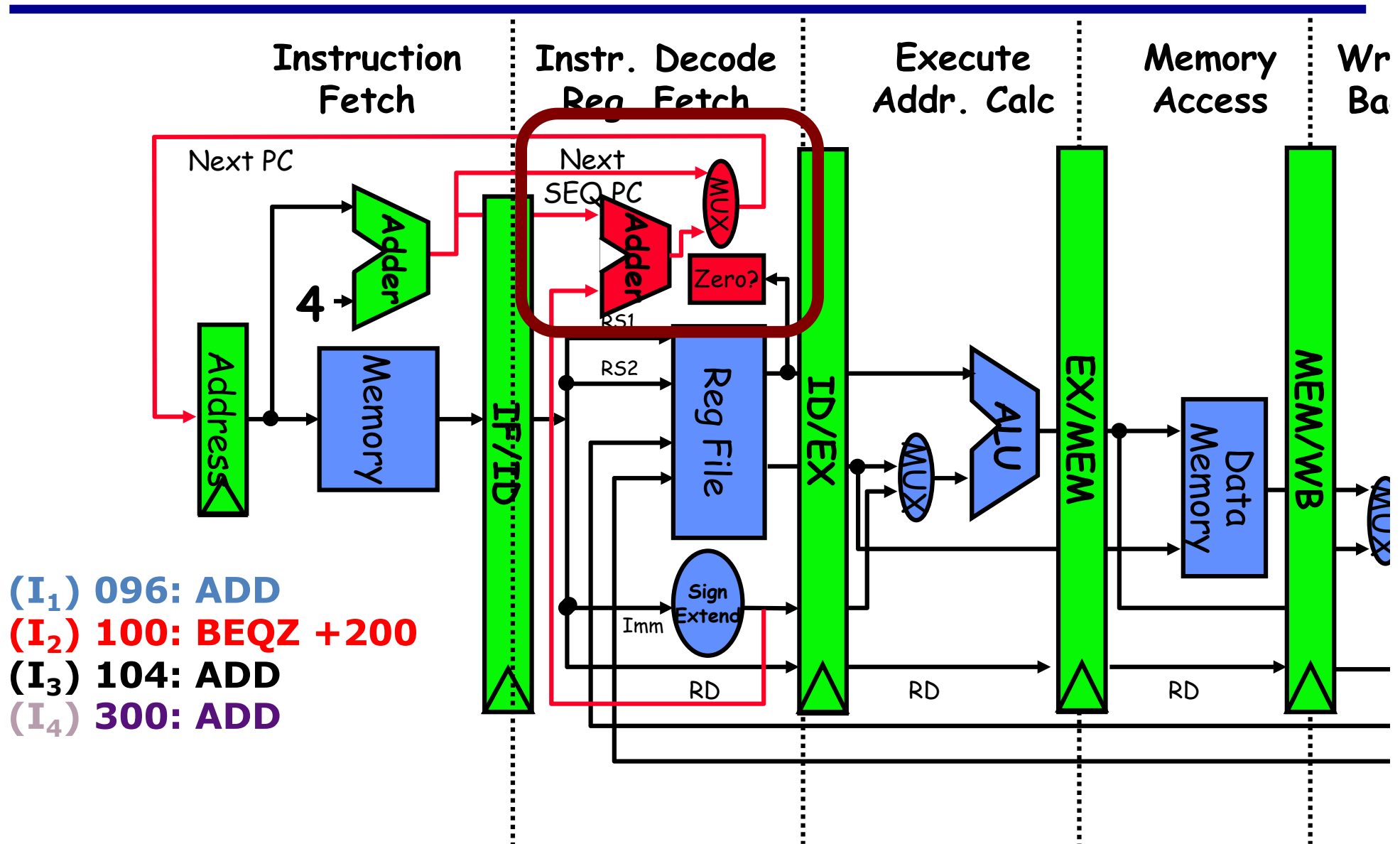


- ⇒ **pipeline bubble**

# Use Simpler Branches: E.g. Only Compare One Register Against Zero in ID Stage



# Pipelined MIPS Datapath



- (I<sub>1</sub>) 096: ADD
- (I<sub>2</sub>) 100: BEQZ +200
- (I<sub>3</sub>) 104: ADD
- (I<sub>4</sub>) 300: ADD

# Control Hazard Delay Summary

---

- **JAL**: unconditional jump to **PC+immediate**
  - 1 cycle delay of pipeline
- **JALR**: indirect jump to **rs1+immediate**
  - 1 cycle delay
- **Branch**: if (**rs1 conds rs2**), branch to **PC+immediate**
  - 2 cycles delay
  - 1 cycle delay for simpler branch (BEQZ) with pipeline improvement

# Reducing Control Flow Penalty

- Software solutions

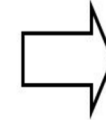
- Eliminate branches - loop unrolling

- Increases the run length

- Reduce resolution time - instruction scheduling

- Compute the branch condition as early as possible (of limited value because branches often in critical path through code)

```
j = 0;
while (j < 100){
    a[j] = b[j+1];
    j += 1;
}
```



```
j = 0;
while (j < 99){
    a[j] = b[j+1];
    a[j+1] = b[j+2];
    j += 2;
}
```

- Hardware solutions

- Find something else to do - delay slots

- Replaces pipeline bubbles with useful work (requires software cooperation)

- Speculate - branch prediction

- Speculative execution of instructions beyond the branch

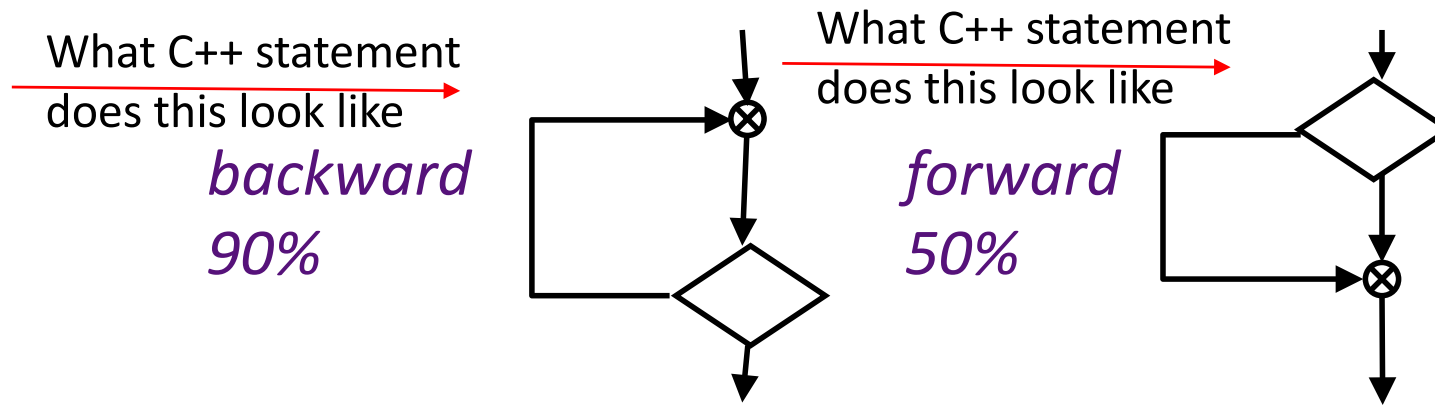
# Branch Prediction

---

- Motivation
  - Branch penalties limit performance of deeply pipelined processors
  - Modern branch predictors have high accuracy
  - (>95%) and can reduce branch penalties significantly
- Required hardware support:
  - Prediction structures:
    - Branch history tables, branch target buffers, etc.
- Mispredict recovery mechanisms:
  - Keep result computation separate from commit
  - Kill instructions following branch in pipeline
  - Restore state to that following branch

# Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:



ISA can attach preferred direction semantics to branches, e.g.,  
Motorola MC88110

*bne0 (preferred taken)*    *beq0 (not taken)*

# Dynamic Branch Prediction

## learning based on past behavior

---

- Temporal correlation (time)
  - If I tell you that a certain branch was taken last time, does this help?
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation (space)
  - Several branches may resolve in a highly correlated manner
  - For instance, a preferred path of execution

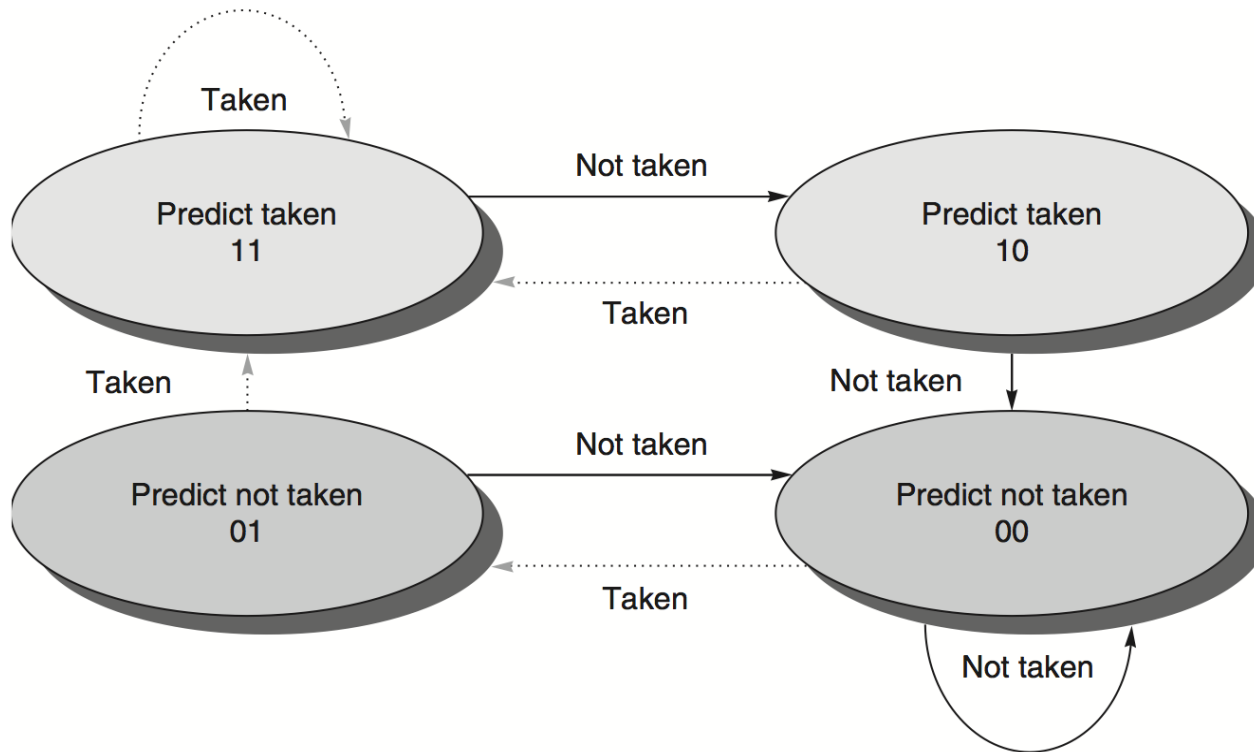
# Dynamic Branch Prediction

---

- 1-bit prediction scheme
  - Low-portion address as address for a one-bit flag for Taken or NotTaken historically
  - Simple
- 2-bit prediction
  - Miss twice to change

# Branch Prediction Bits

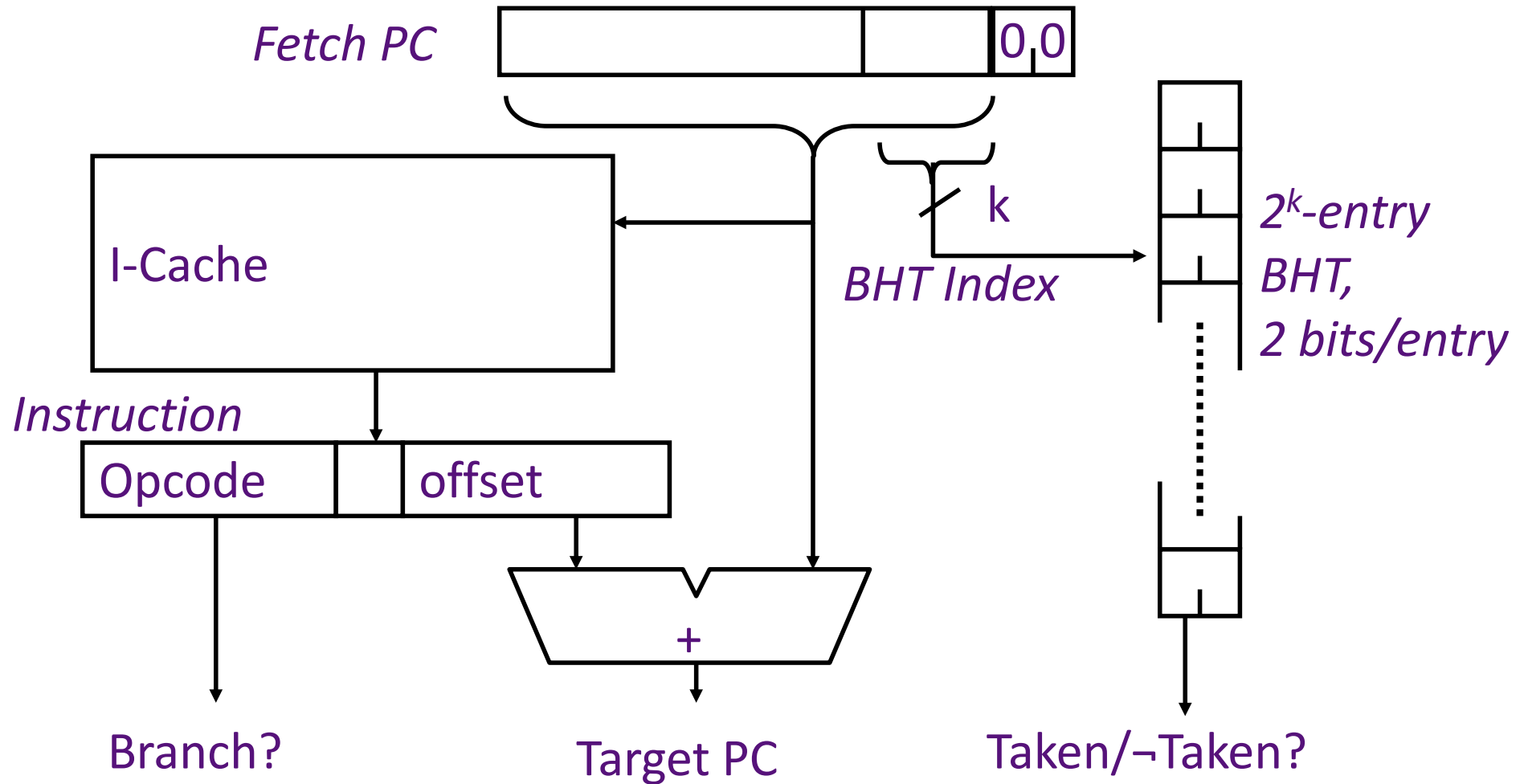
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



*BP state:*

*(predict take/¬take) x (last prediction right/wrong)*

# Branch History Table



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

# Exploiting Spatial Correlation

*Yeh and Patt, 1992*

---

```
if (x[i] < 7) then  
  y += 1;  
if (x[i] < 5) then  
  c -= 4;
```

If first condition false, second condition also false

*History register, H*, records the direction of the last N branches executed by the processor

# Two-Level Branch Predictor

*Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)*

