Lecture 08: RISC-V Pipeline Implementation

CSCE 513 Computer Architecture

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
https://passlab.github.io/CSCE513

Acknowledgement

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

Review

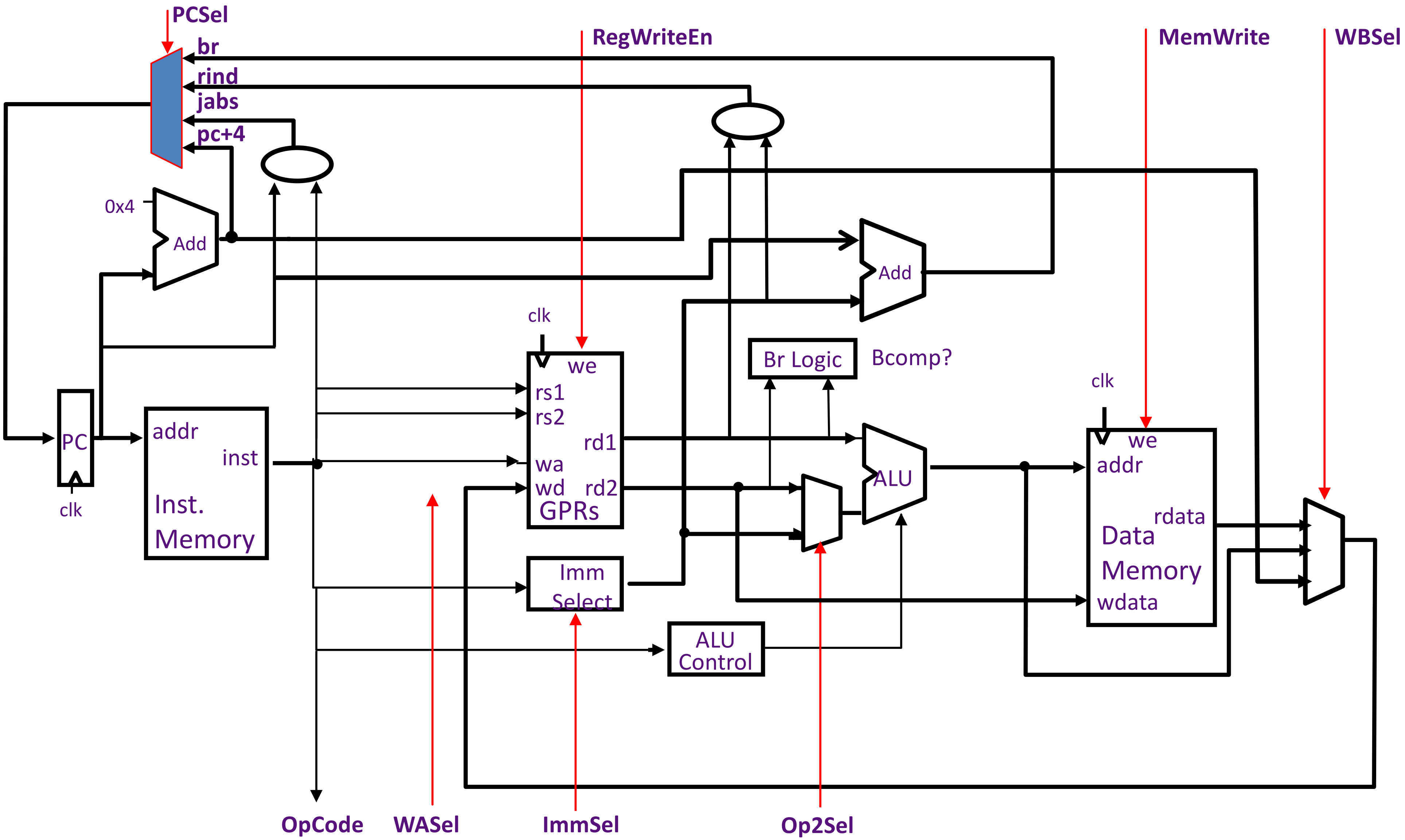
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

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

- 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



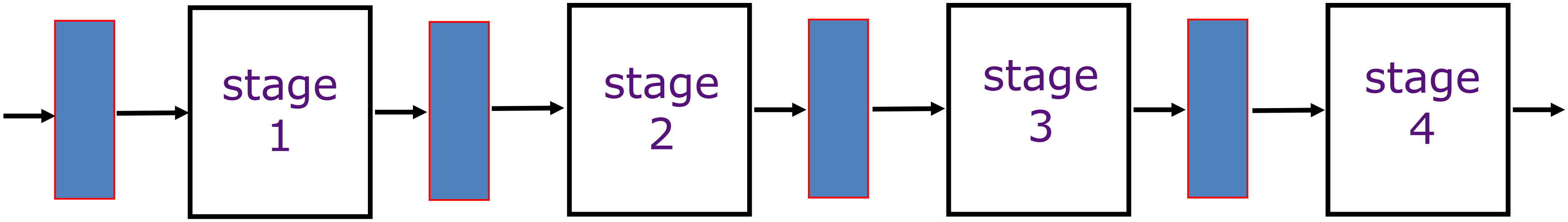
Review: Hardwired Control Table

Opcode	ImmSel	Op2Sel	FuncSel	MemWr	RFWen	WBSel	WASel	PCSel
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	IType_{12}	Imm	Op	no	yes	ALU	rd	pc+4
LW	IType_{12}	Imm	+	no	yes	Mem	rd	pc+4
SW	SType_{12}	Imm	+	yes	no	*	*	pc+4
BEQ_{true}	SBType_{12}	*	*	no	no	*	*	br
$\text{BEQ}_{\text{false}}$	SBType_{12}	*	*	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	X1	jabs
JALR	*	*	*	no	yes	PC	rd	rind

Op2Sel= Reg / Imm **WBSel = ALU / Mem / PC**

WASel = rd / X1 **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

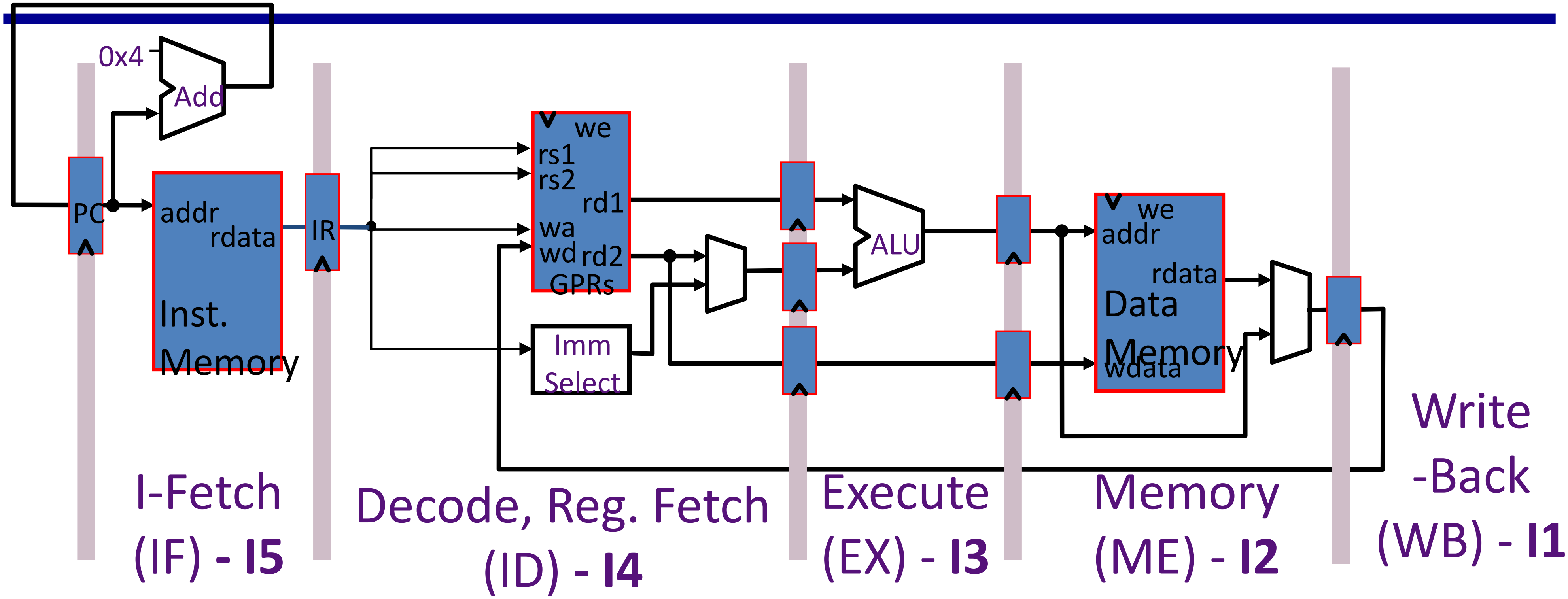
$$t_{IF} \sim= t_{ID/RF} \sim= t_{EX} \sim= t_{MEM} \sim= t_{WB}$$

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

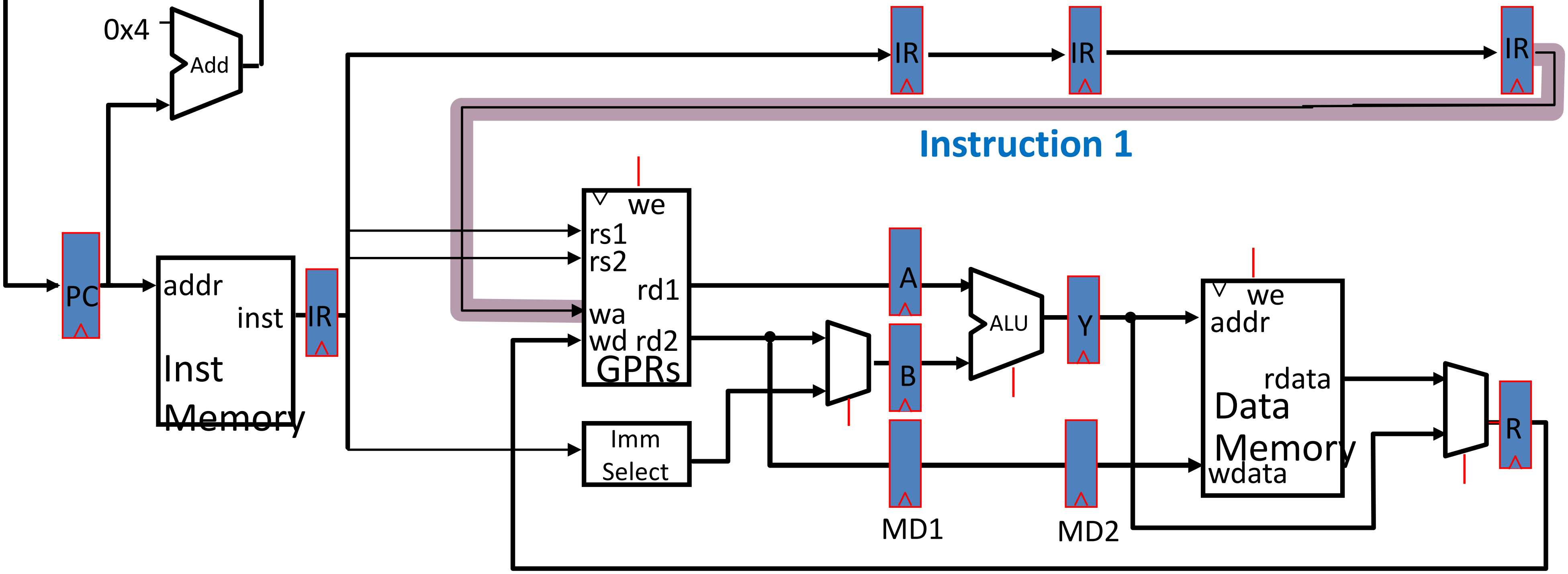
The Whole Pipeline Resources are Used by 5 Instructions in Every Cycle!



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
Instruction 1	IF_1	ID_1	EX_1	ME_1	WB_1				
Instruction 2		IF_2	ID_2	EX_2	ME_2	WB_2			
Instruction 3			IF_3	ID_3	EX_3	ME_3	WB_3		
Instruction 4				IF_4	ID_4	EX_4	ME_4	WB_4	
Instruction 5					IF_5	ID_5	EX_5	ME_5	WB_5

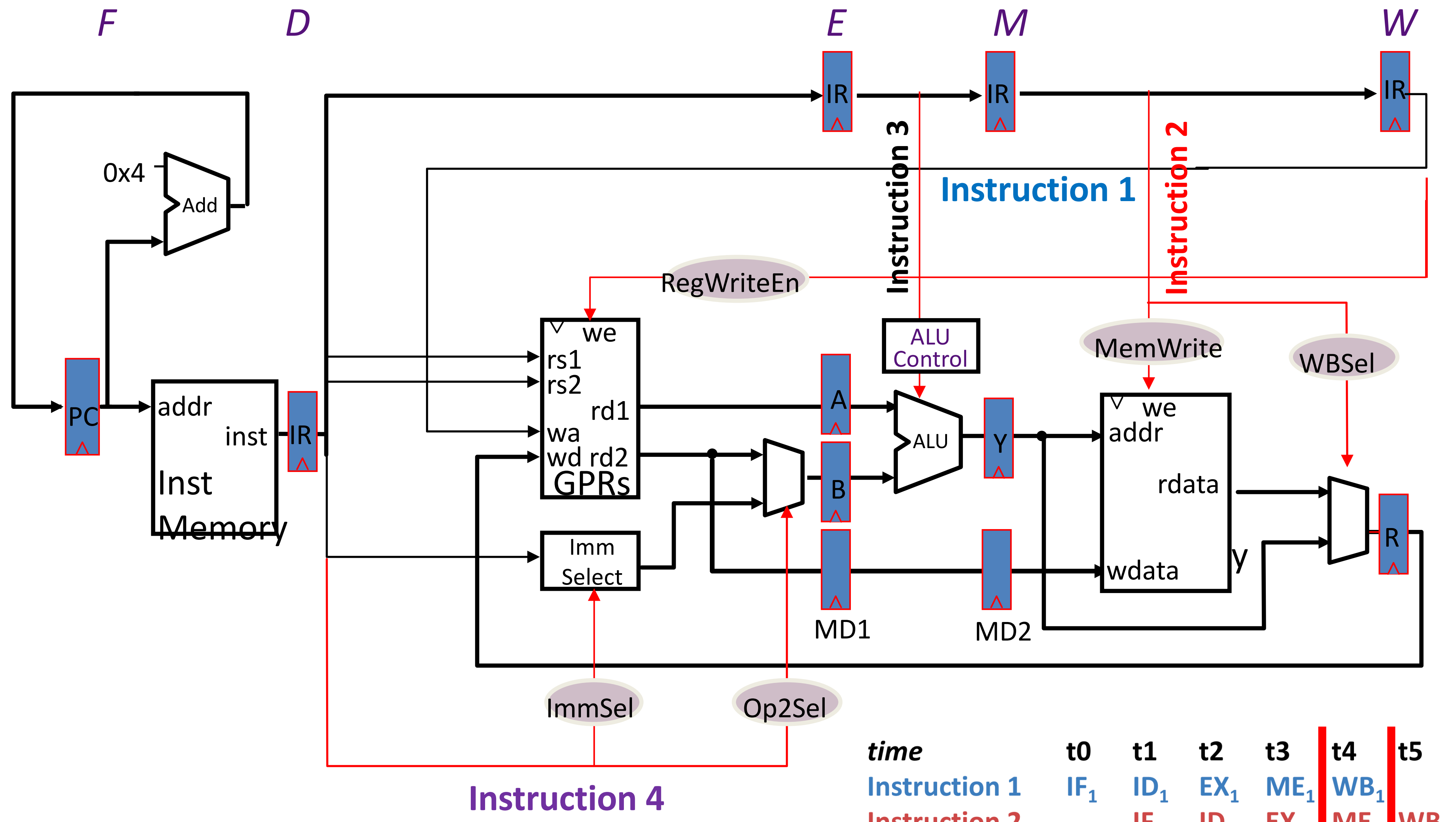
Instruction Register in Each Stage

time	t0	t1	t2	t3	t4	t5
Instruction 1	IF_1	ID_1	EX_1	ME_1	WB_1	
Instruction 2		IF_2	ID_2	EX_2	ME_2	WB_2
Instruction 3			IF_3	ID_3	EX_3	ME_3
Instruction 4				IF_4	ID_4	EX_4
Instruction 5					IF_5	ID_5



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

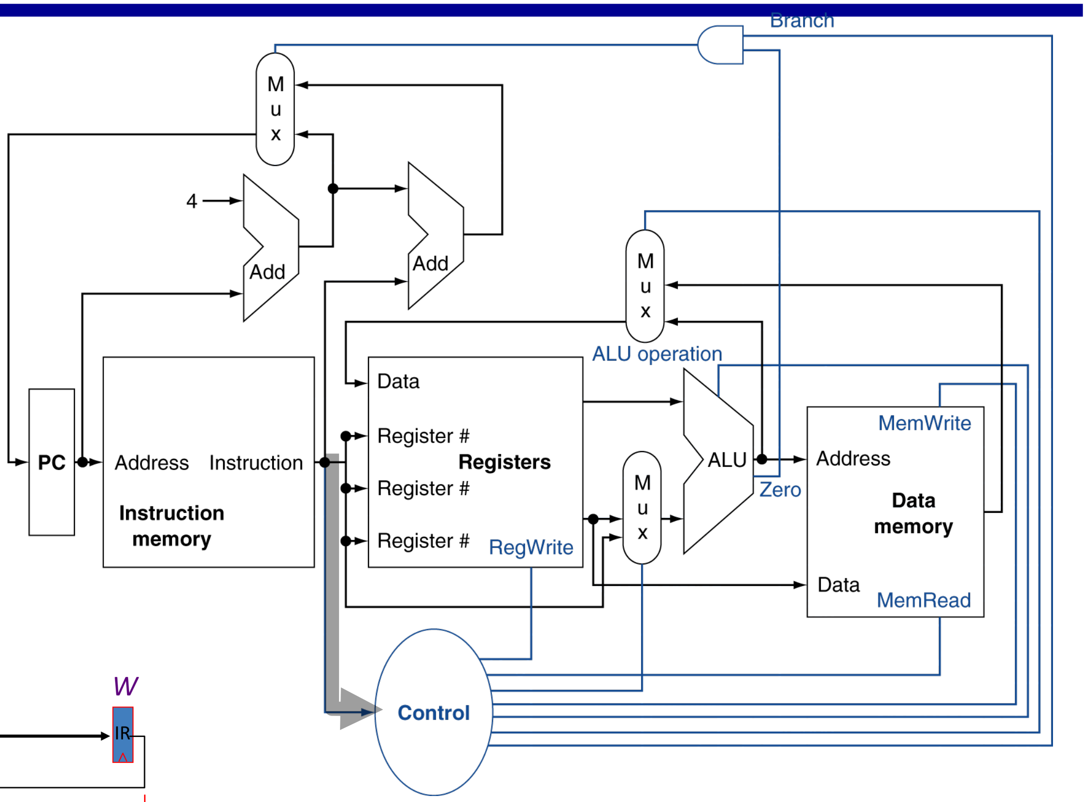
Connect Controls from Instruction Register

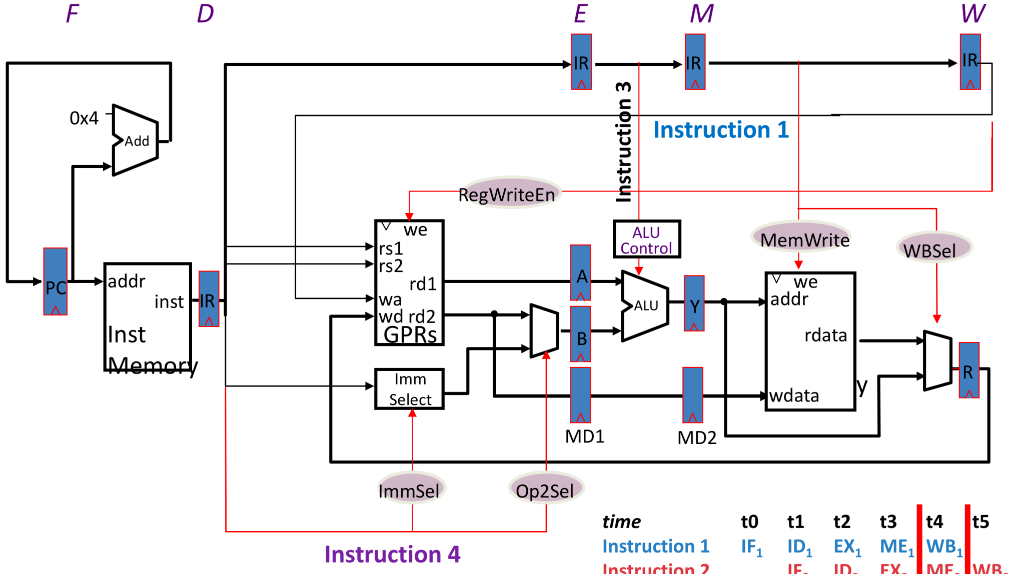


time	t0	t1	t2	t3	t4	t5
Instruction 1	IF$_1$	ID$_1$	EX$_1$	ME$_1$	WB$_1$	
Instruction 2		IF$_2$	ID$_2$	EX$_2$	ME$_2$	WB$_2$
Instruction 3			IF$_3$	ID$_3$	EX$_3$	ME$_3$
Instruction 4				IF$_4$	ID$_4$	EX$_4$
Instruction 5					IF$_5$	ID$_5$

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





time	t0	t1	t2	t3	t4	t5
Instruction 1	IF_1	ID_1	EX_1	ME_1	WB_1	
Instruction 2		IF_2	ID_2	EX_2	ME_2	WB_2
Instruction 3			IF_3	ID_3	EX_3	ME_3
Instruction 4				IF_4	ID_4	EX_4
Instruction 5					IF_5	ID_5

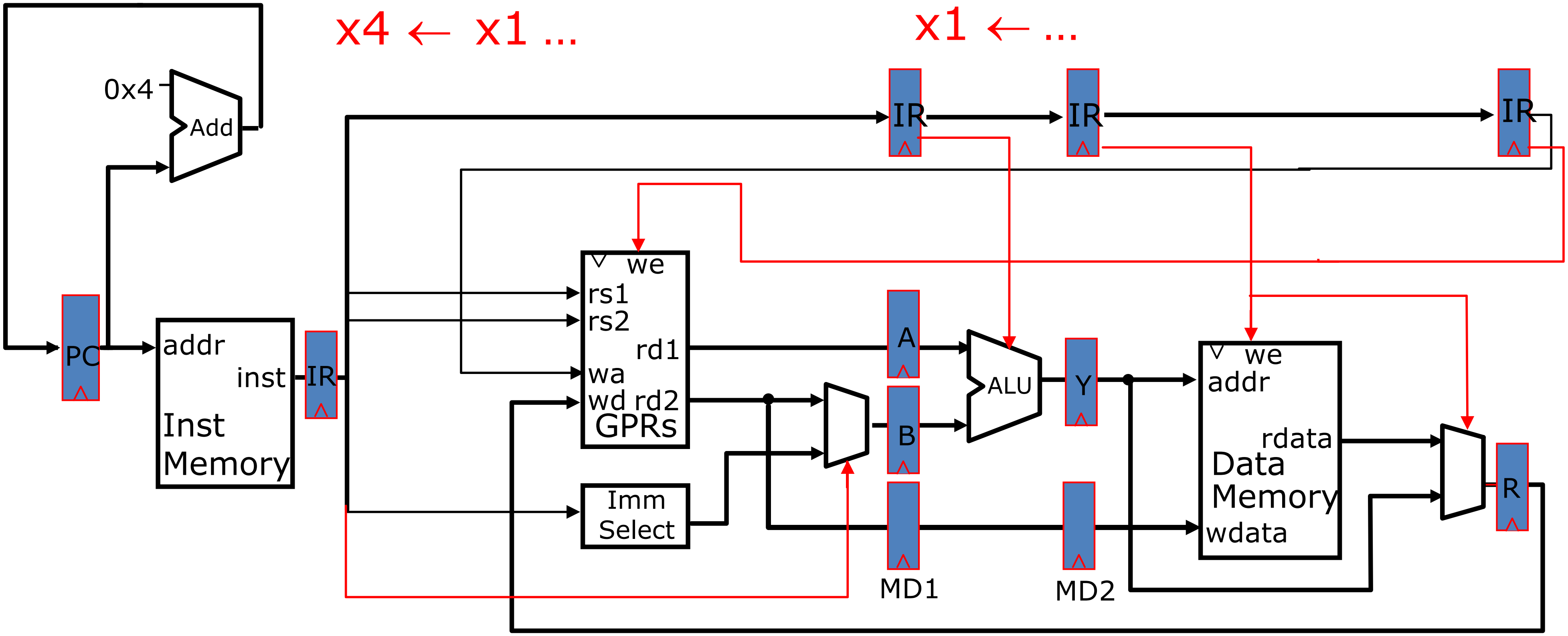
Instructions Interact With Each Other in Pipeline: Dealing with Hazards

- 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 ← x0 + 10
x4 ← 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

time

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
(I_1) x1 ← (x0) + 10	IF_1	ID_1	EX_1	ME_1	WB_1							
(I_2) x4 ← (x1) + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	ME_2	WB_2			
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	ME_3	WB_3		
(I_4)							IF_4	ID_4	EX_4	ME_4	WB_4	
(I_5)								IF_5	ID_5	EX_5	ME_5	WB_5

stalled stages

time

Resource Usage

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
IF	I_1	I_2	I_3	I_3	I_3	I_3	I_4	I_5				
ID		I_1	I_2	I_2	I_2	I_2	I_3	I_4	I_5			
EX			I_1	-	-	-	I_2	I_3	I_4	I_5		
ME				I_1	-	-	-	I_2	I_3	I_4	I_5	
WB					I_1	-	-	-	I_2	I_3	I_4	I_5

- ⇒ ***pipeline bubble***

Insert Bubble for Interlocking

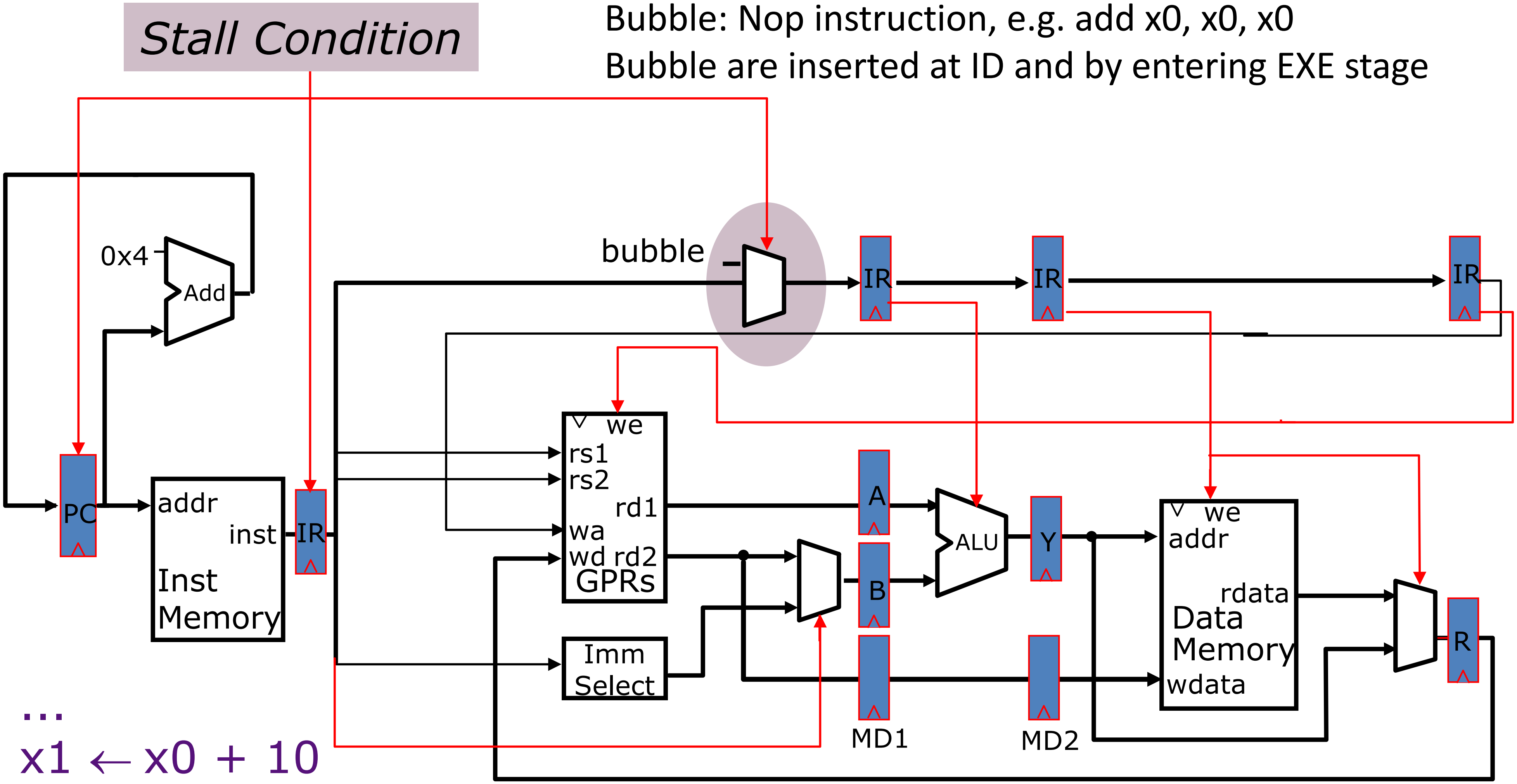
time

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) x1 ← (x0) + 10	IF_1	ID_1	EX_1	ME_1	WB_1				
(I_2) x4 ← (x1) + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	ME_2	WB_2
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	ME_3
(I_4)							IF_4	ID_4	EX_4
(I_5)								IF_5	ID_5

stalled stages

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

...

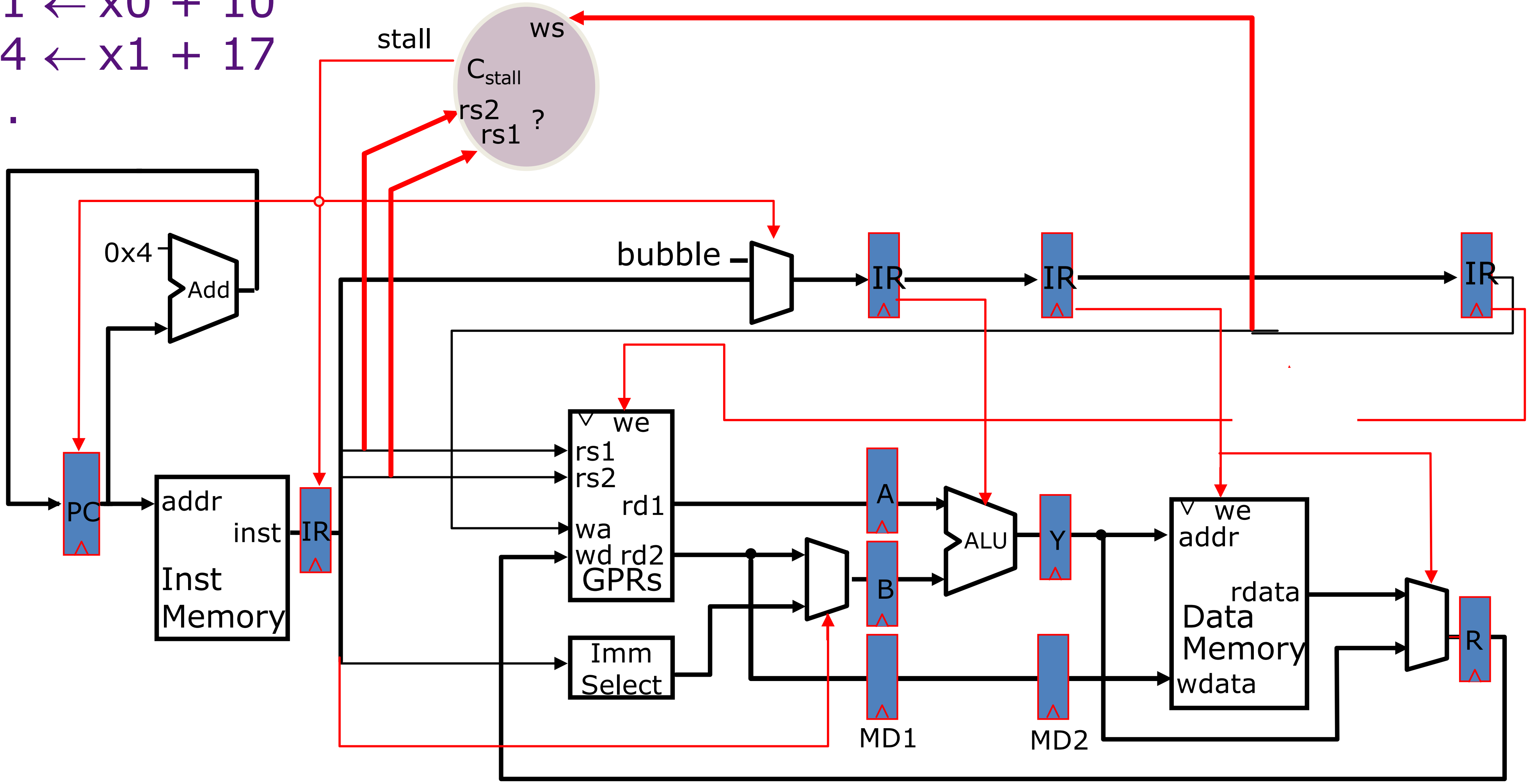
Interlock Control Logic

…

x1 ← x0 + 10

x4 ← 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

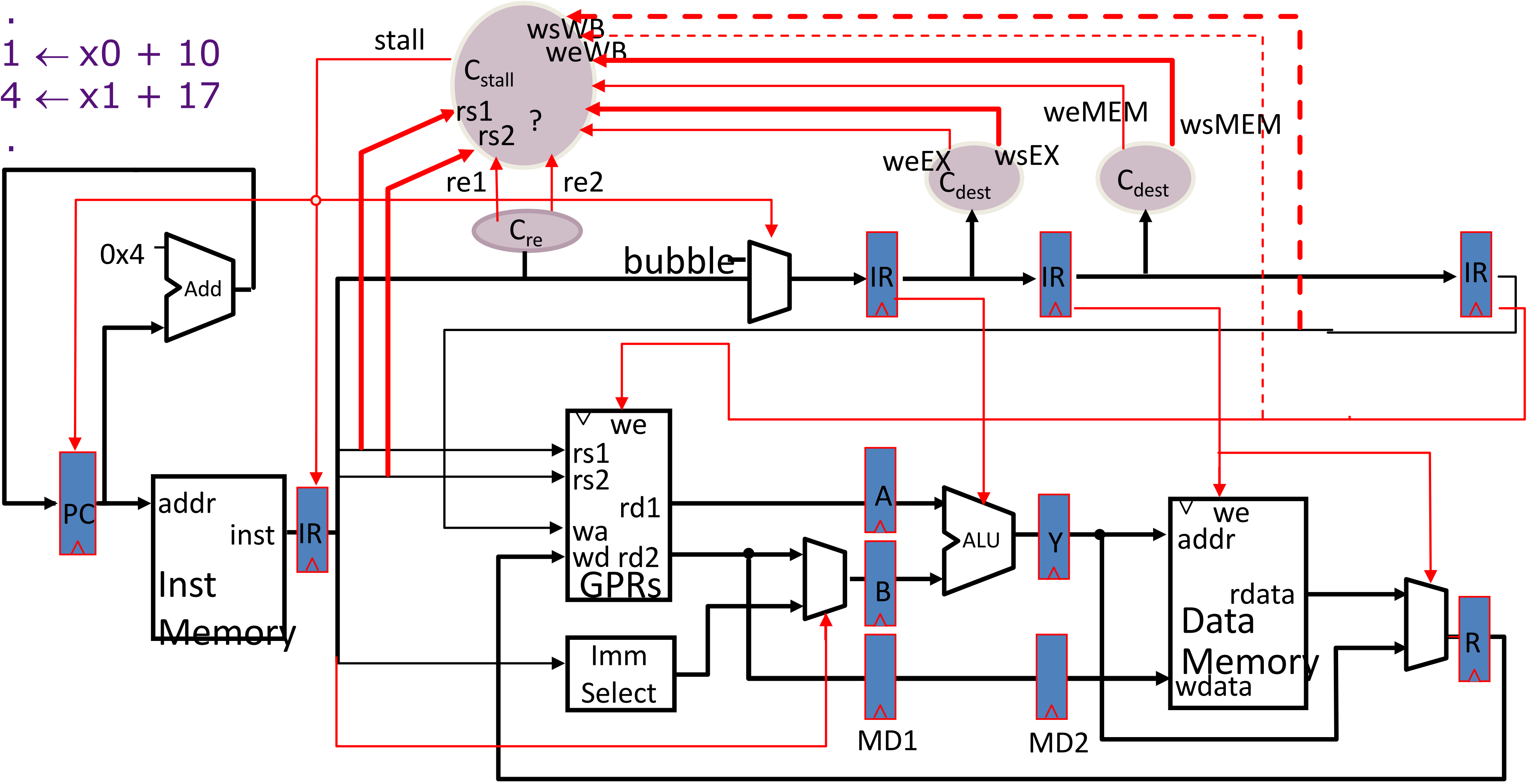
rs: read select, 5-bit register number

...

x1 ← x0 + 10

x4 ← x1 + 17

...



Should we always stall if an rs field matches some rd?

not every instruction writes a register => we

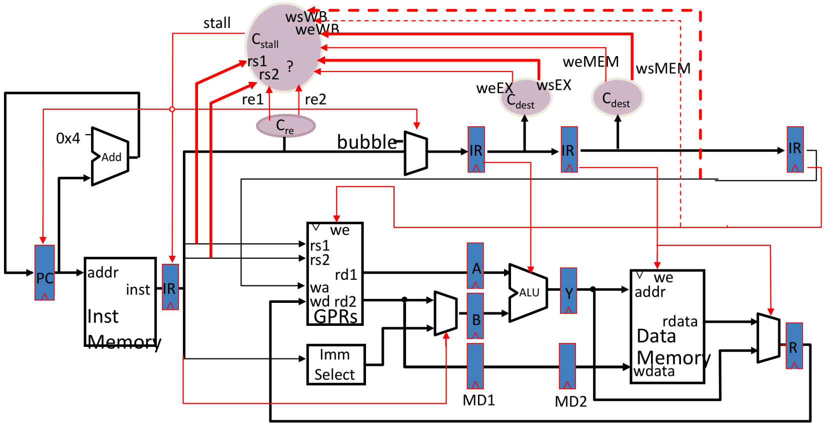
not every instruction reads a register => re

Source & Destination Registers

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

		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



		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_{stall}

$$
\begin{aligned}
stall = \; & ((rs1_D == ws_{EX}) \; \&\& \; we_{EX} \; + \\
& (rs1_D == ws_{MEM}) \; \&\& \; we_{MEM} \; + \\
& \cancel{(rs1_D == ws_{WB}) \; \&\& \; we_{WB}}) \; \&\& \; re1_D \; + \\
& ((rs2_D == ws_{EX}) \; \&\& \; we_{EX} \; + \\
& (rs2_D == ws_{MEM}) \; \&\& \; we_{MEM} \; + \\
& \cancel{(rs2_D == ws_{WB}) \; \&\& \; we_{WB}}) \; \&\& \; re2_D
\end{aligned}
$$

C_{dest}

ws = rd

we = *Case* opcode

ALU, ALUi, LW, JALR =>on

… =>off

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

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) $x1 \leftarrow x0 + 10$	IF_1	ID_1	EX_1	ME_1	WB_1				
(I_2) $x4 \leftarrow x1 + 17$		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	ME_2	WB_2
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	ME_3
(I_4)			*stalled stages*				IF_4	ID_4	EX_4
(I_5)								IF_5	ID_5

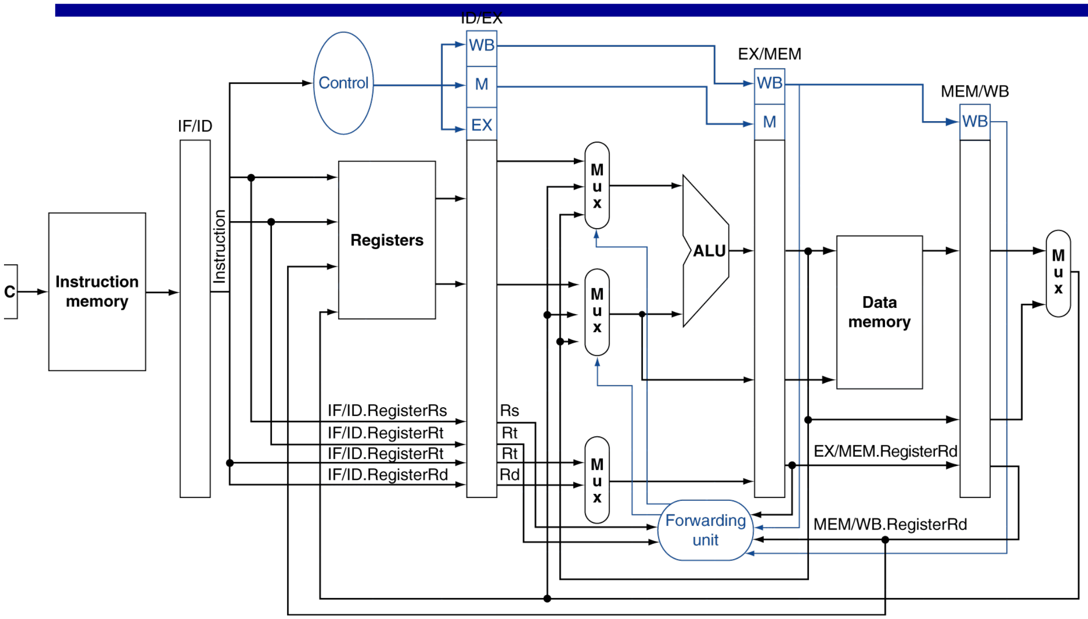
Each *stall or kill* introduces a bubble in the pipeline

=> *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) $x1 \leftarrow x0 + 10$		IF_1	ID_1	EX_1	ME_1	WB_1				
(I_2) $x4 \leftarrow x1 + 17$			IF_2	ID_2	EX_2	ME_2	WB_2			
(I_3)				IF_3	ID_3	EX_3	ME_3	WB_3		
(I_4)					IF_4	ID_4	EX_4	ME_4	WB_4	
(I_5)						IF_5	ID_5	EX_5	ME_5	WB_5

Review: Hardware Support for Forwarding, and Detecting RAW Hazards with Previous and 2nd Previous Instructions



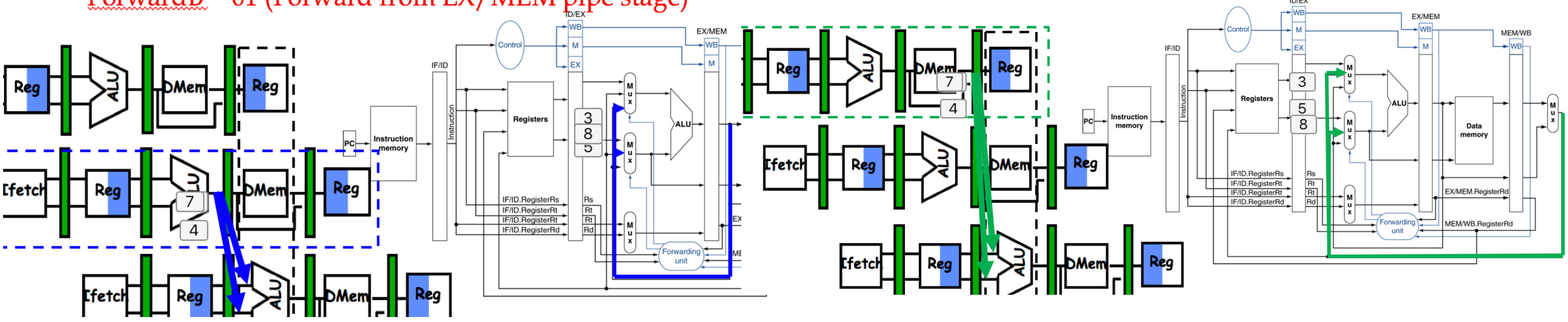
- Slide 48 of lecture05-06

▣ Detecting RAW hazard with Previous Instruction

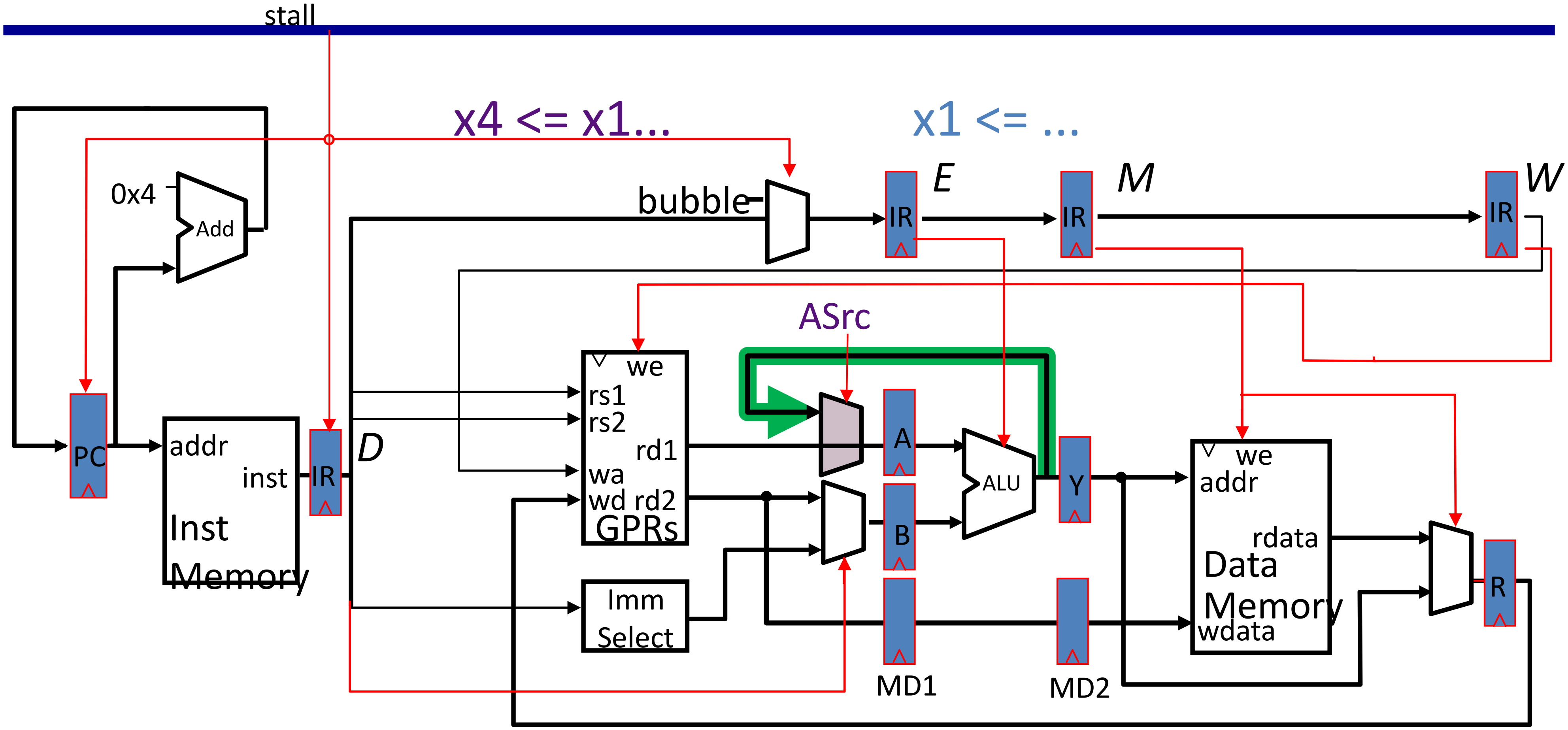
- ◆ if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01 (Forward from EX/MEM pipe stage)
- ◆ if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01 (Forward from EX/MEM pipe stage)

▣ 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$_1$) x1 <= x0 + 10

(I$_2$) x4 <= x1 + 17

Yes

x1 <= M[x0 + 10]

x4 <= x1 + 17

No, Load→EXE-Use

JAL 500

x4 <= x1 + 17

No

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

stall = (((~~$rs1_D$ == 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~~) && $re2_D$)

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

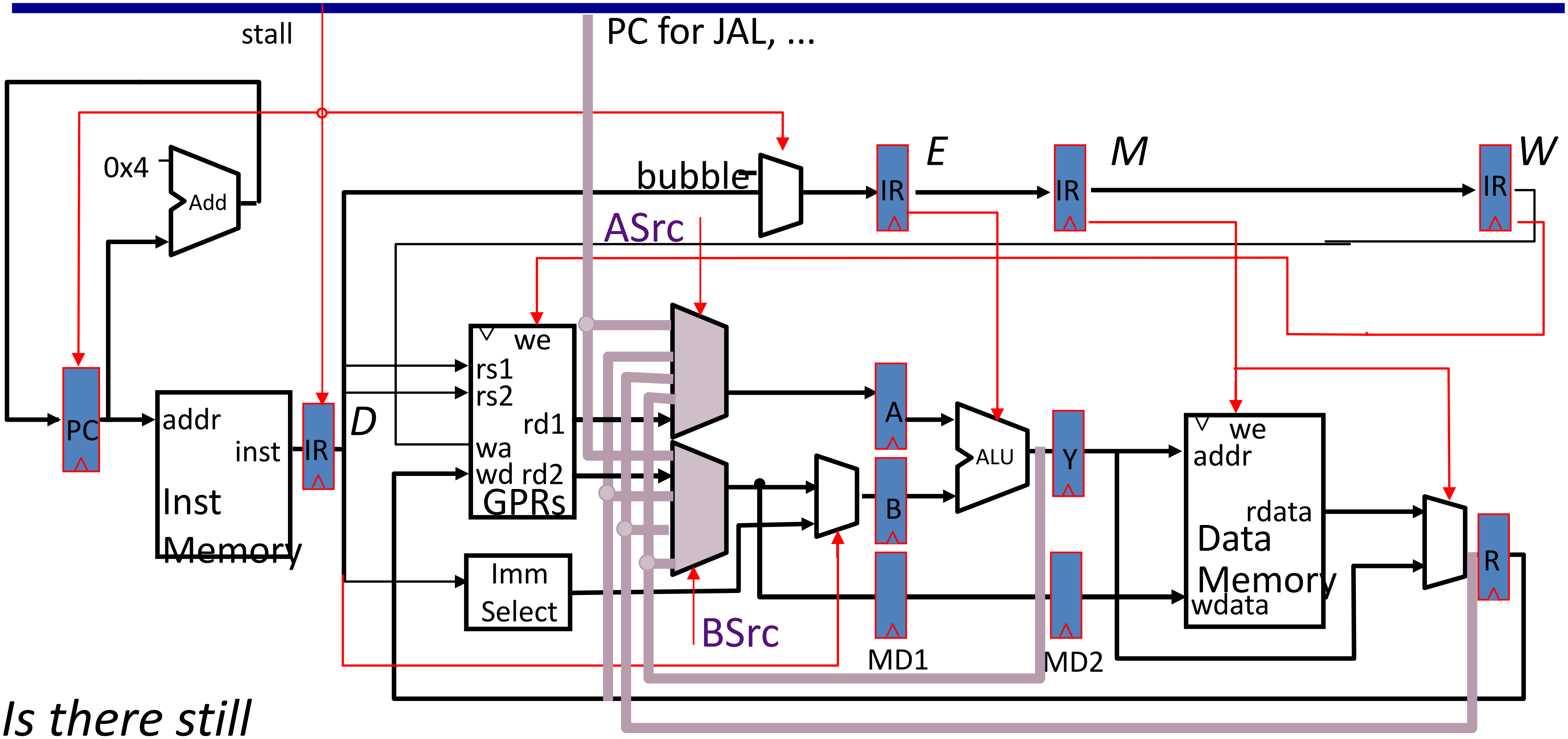
we-bypass$_E$ = *Case* opcode$_E$
ALU, ALUi => on
... => off

we-stall$_E$ = *Case* opcode$_E$
LW, JAL, JALR=> on
JAL => on
... => off

ASrc = (rs1$_D$ == ws$_E$) && we-bypass$_E$ && re1$_D$

stall = ((rs1$_D$ == ws$_E$) && 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



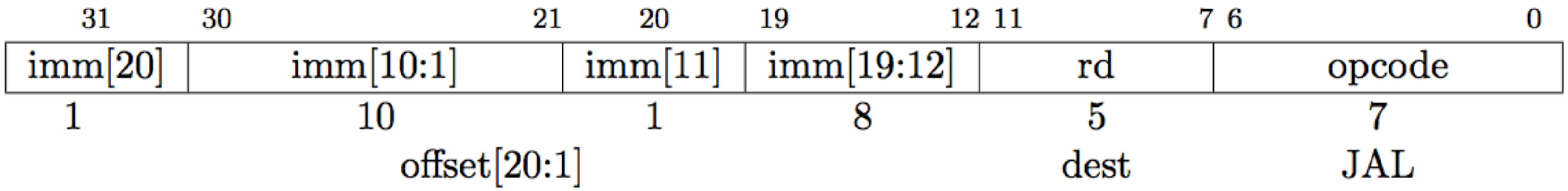
Is there still a need for the stall signal ?

stall = $(rs1_D == ws_E)$ && $(opcode_E == LW_E)$&&$(ws_E != 0)$&&$re1_D$

+ $(rs2_D == ws_E)$ && $(opcode_E == LW_E)$&&$(ws_E != 0)$&&$re2_D$

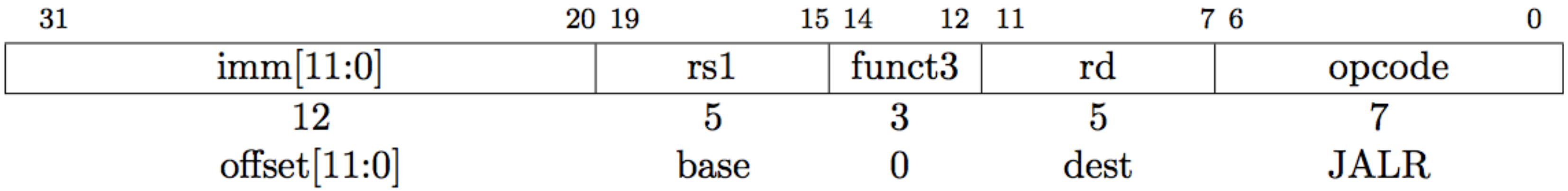
Control Hazards: Branches and Jumps

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



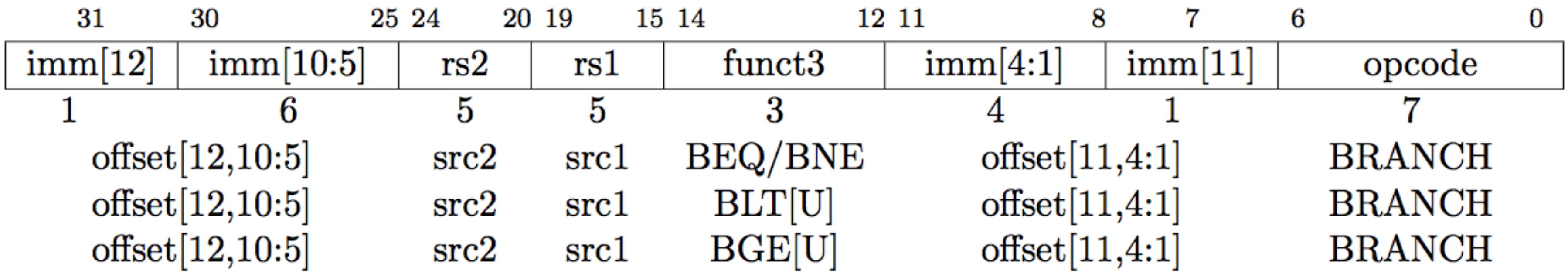
31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
offset[20:1]				dest	JAL

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



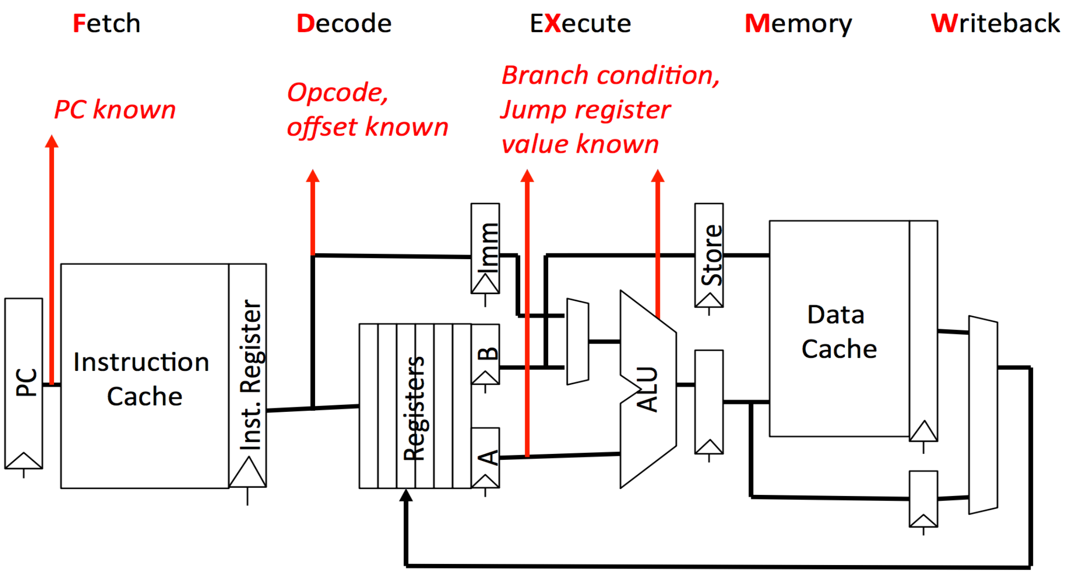
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

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



31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

Info for Control Transfer



PC known

Opcode, offset known

Branch condition, Jump register value known

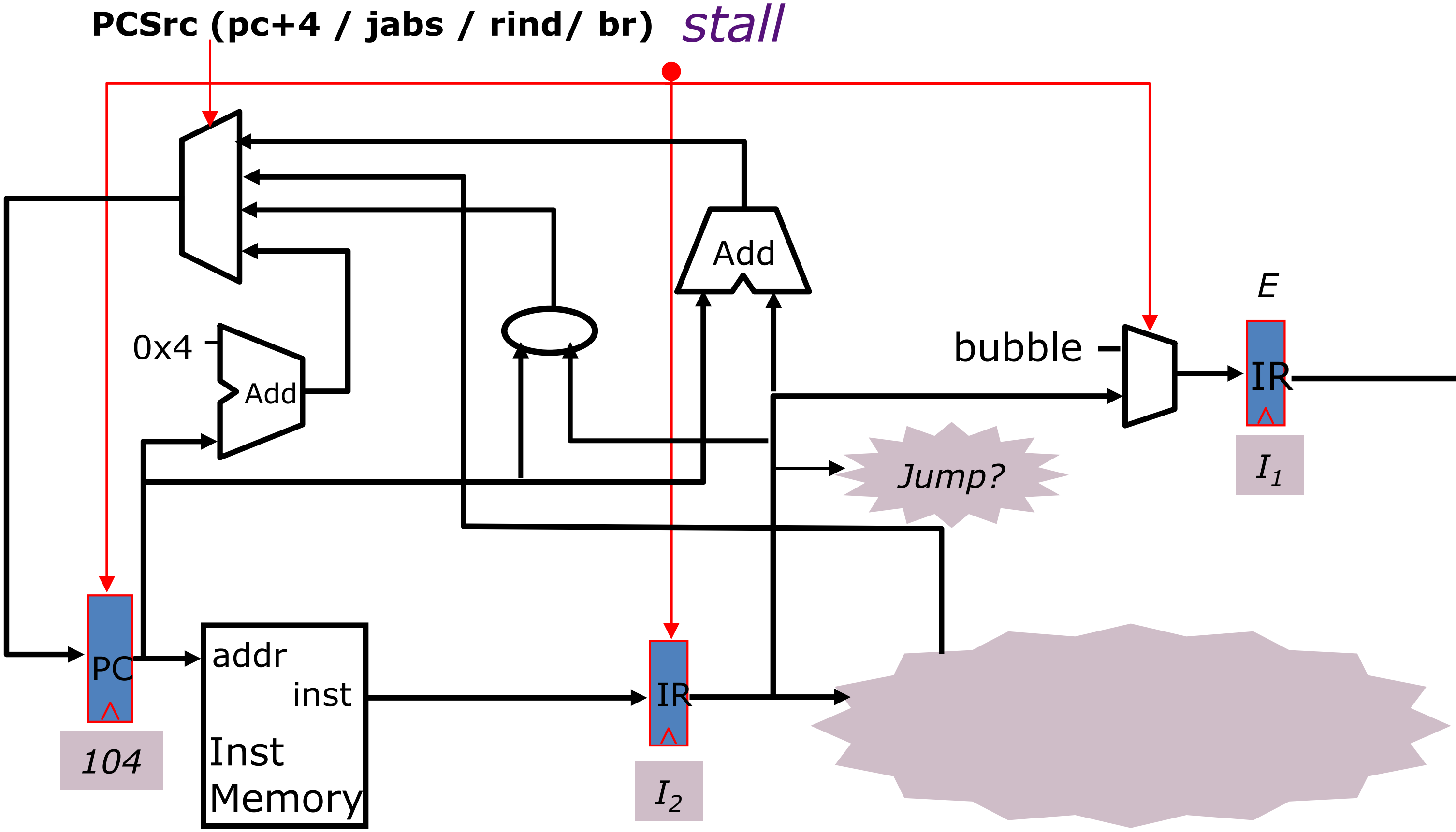
Two pieces of info:

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

Instruction	*Taken known?*	*Target known?*
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

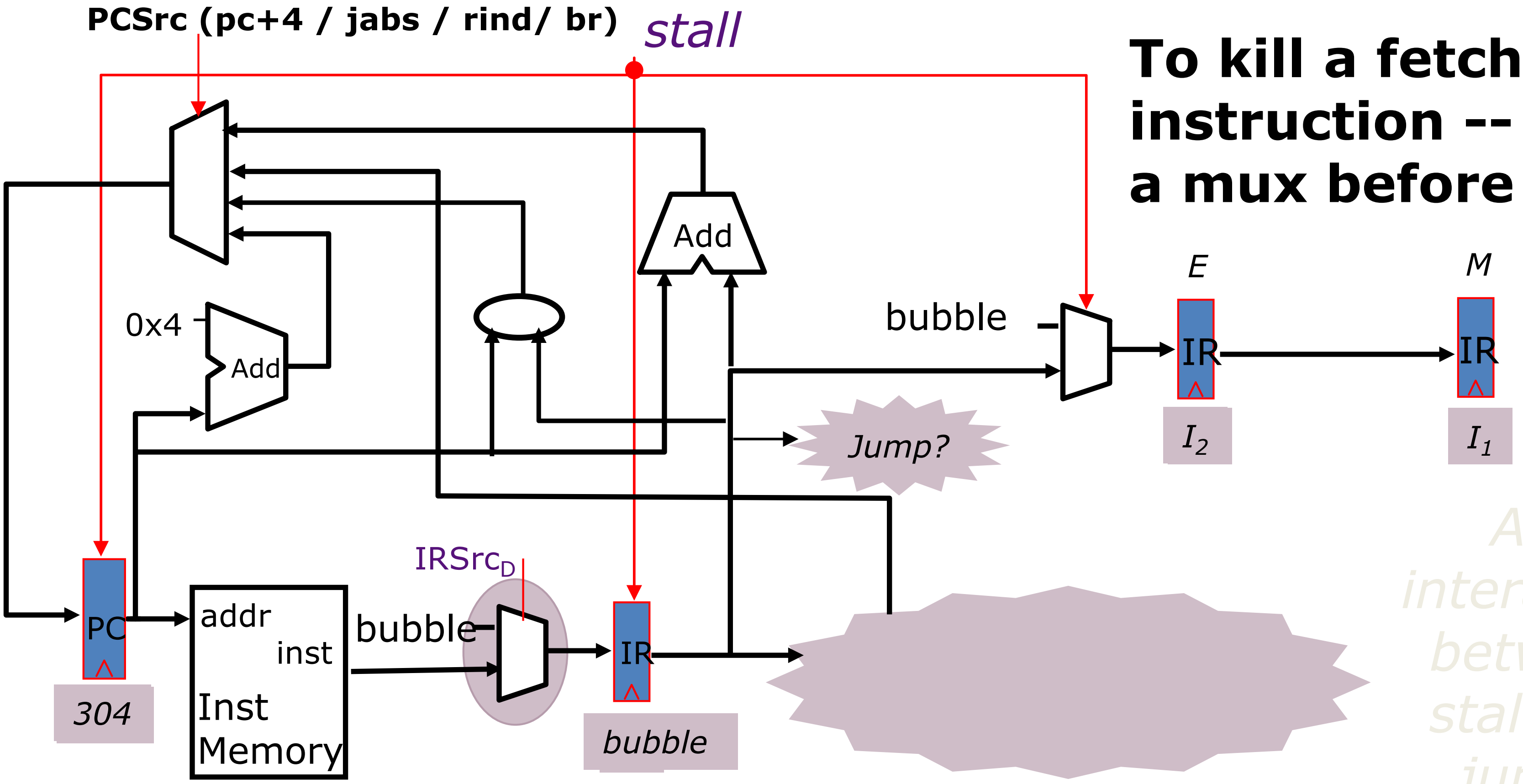


I_1	096	ADD	
I_2	100	J 304	
I_3	~~104~~	~~ADD~~	kill
I_4	304	ADD	

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~~	*kill*
I_4	304	ADD	

$IRSrc_D$ = *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	ME_1	WB_1				
(I_2) 100: J 304		IF_2	ID_2	EX_2	ME_2	WB_2			
(I_3) 104: ADD				IF_3	-	-	-	-	
(I_4) 304: ADD					IF_4	ID_4	EX_4	ME_4	WB_4

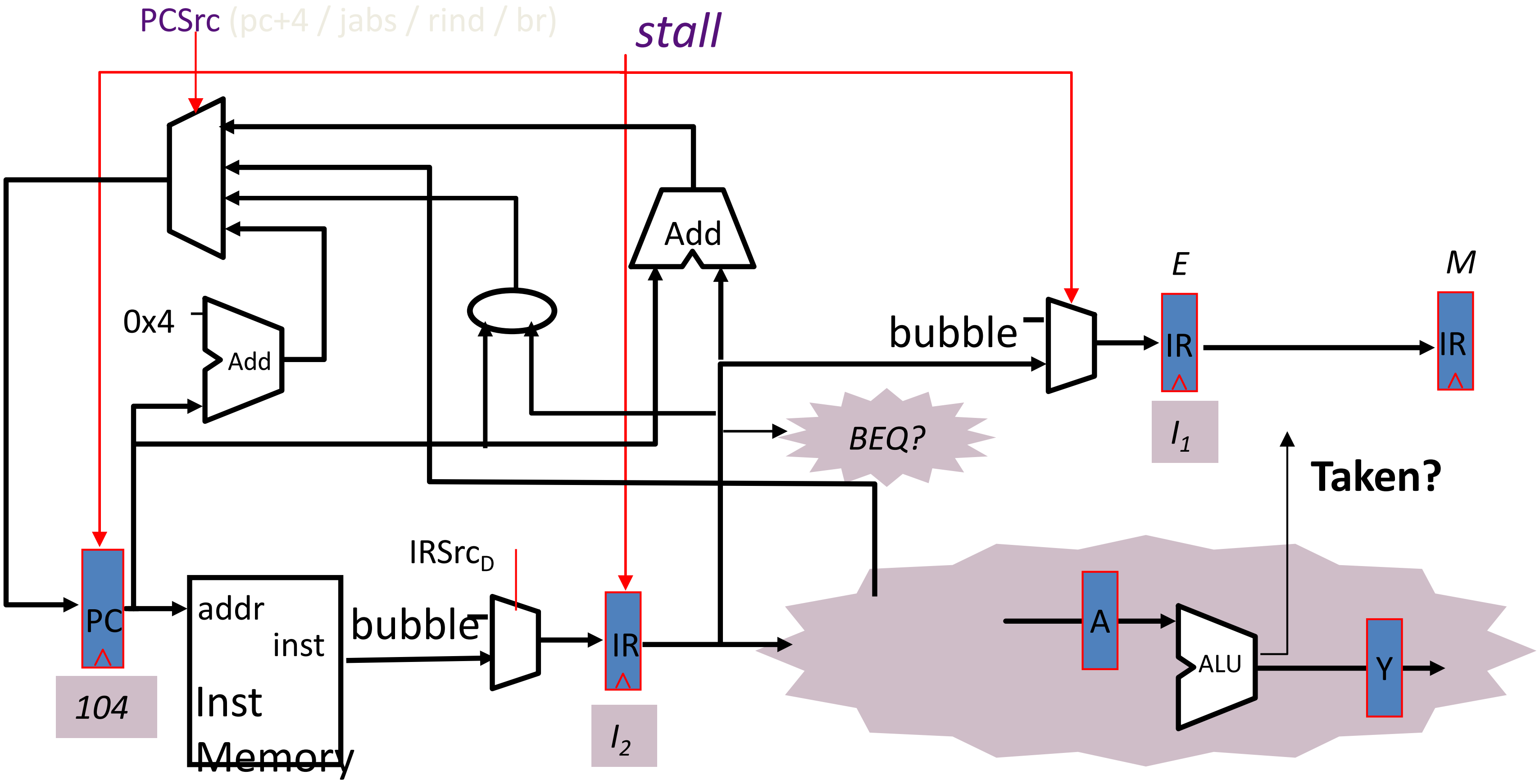
Resource Usage

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		
ME				I_1	I_2	-	I_4	I_5	
WB					I_1	I_2	-	I_4	I_5

- $\Rightarrow$ *pipeline bubble*

Pipelining Conditional Branches



I_1 096 ADD

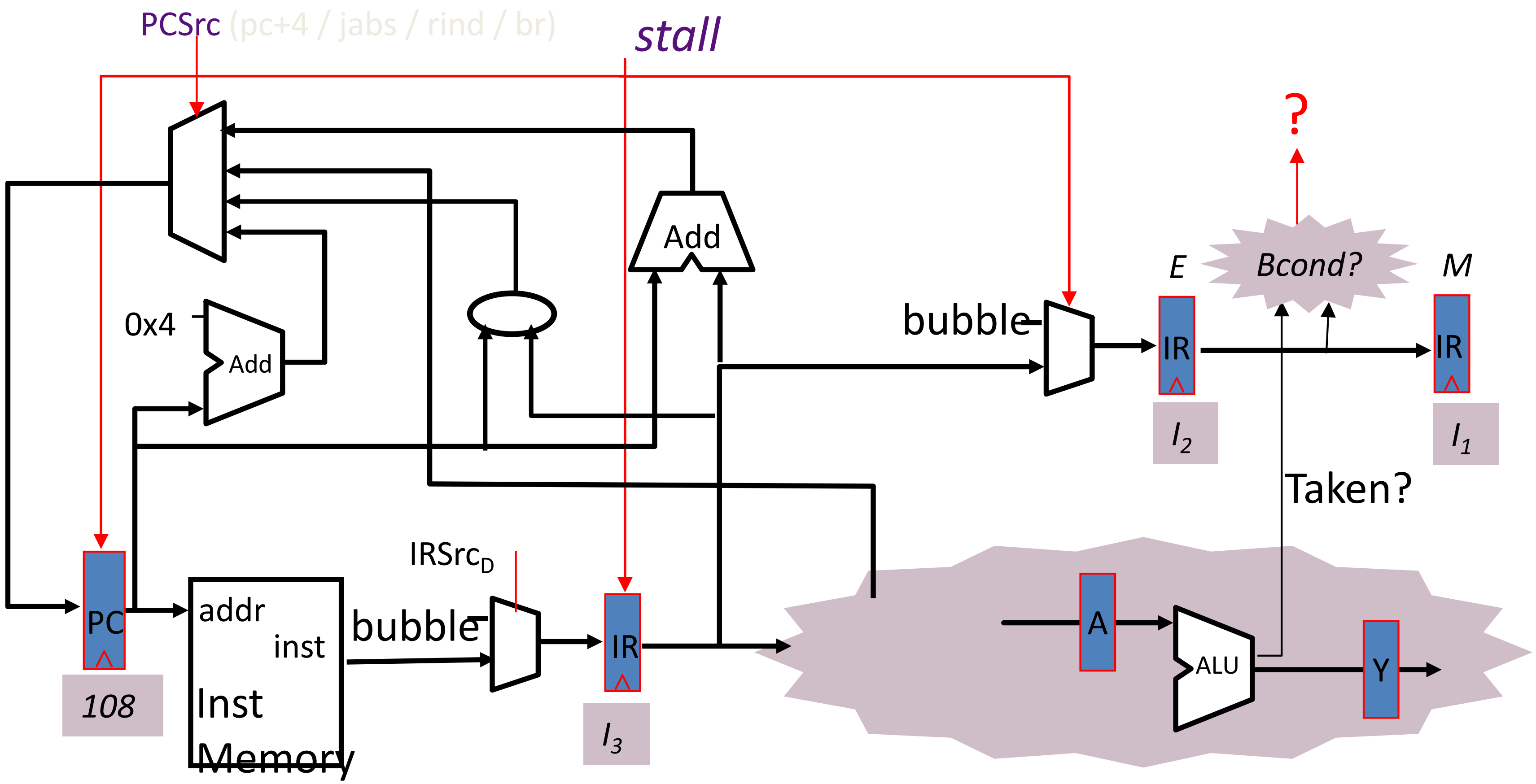
I_2 100 BEQ x1,x2 +200

I_3 104 ADD

I_4 304 ADD

Branch condition is not known until the execute stage

Pipelining Conditional Branches



I_1 096 ADD

I_2 100 BEQ x1,x2 +200

I_3 104 ADD

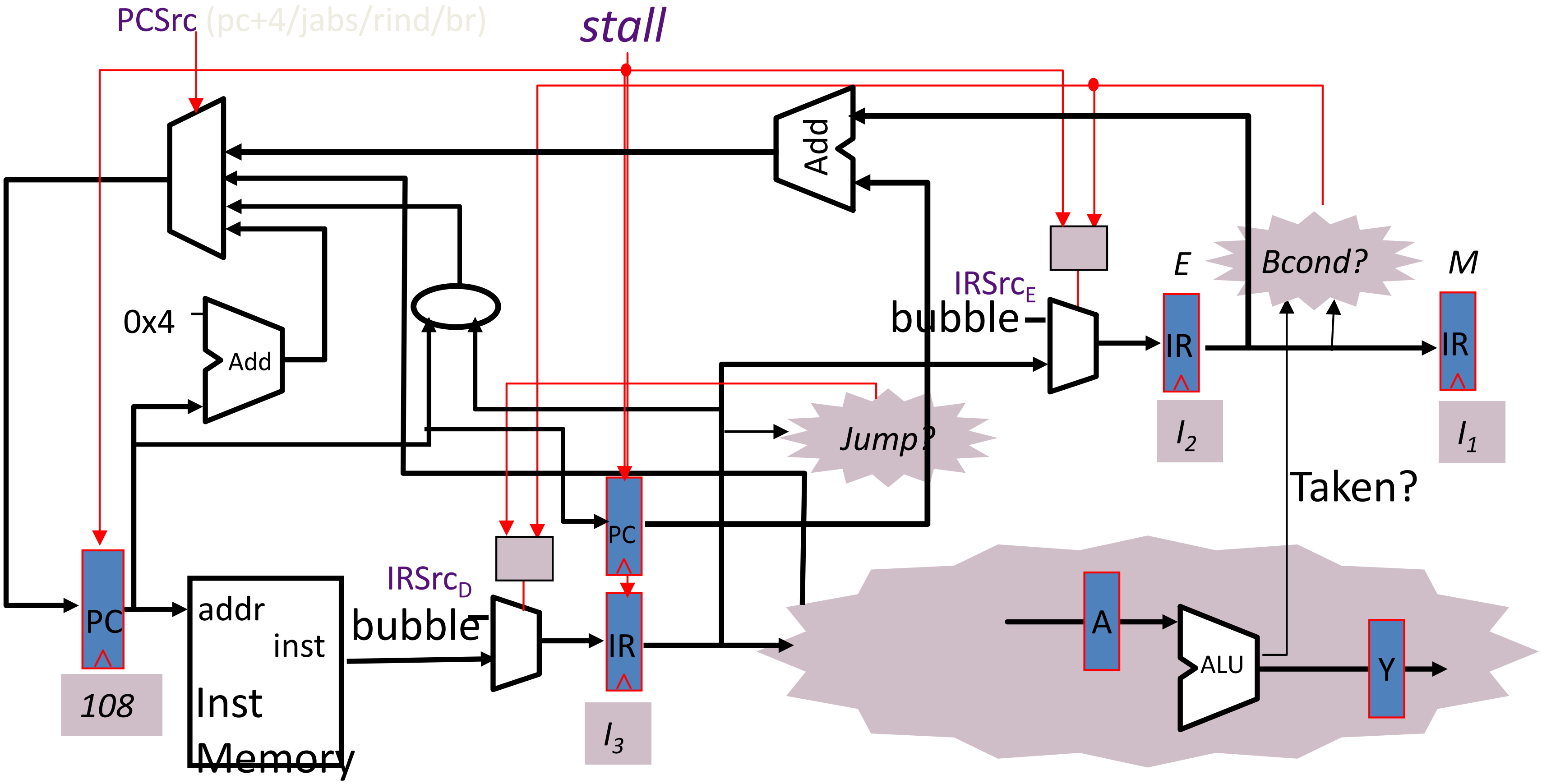
I_4 304 ADD

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	ME_1	WB_1					
(I_2) 100: BEQ +200		IF_2	ID_2	EX_2	ME_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	ME_5	WB_5

time

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

- ⇒ ***pipeline bubble***

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

time

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	ME_1	WB_1				
(I_2) 100: BEQZ +200		IF_2	ID_2	EX_2	ME_2	WB_2			
(I_3) 104: ADD				IF_3	-	-	-	-	
(I_4) 300: ADD					IF_4	ID_4	EX_4	ME_4	WB_4

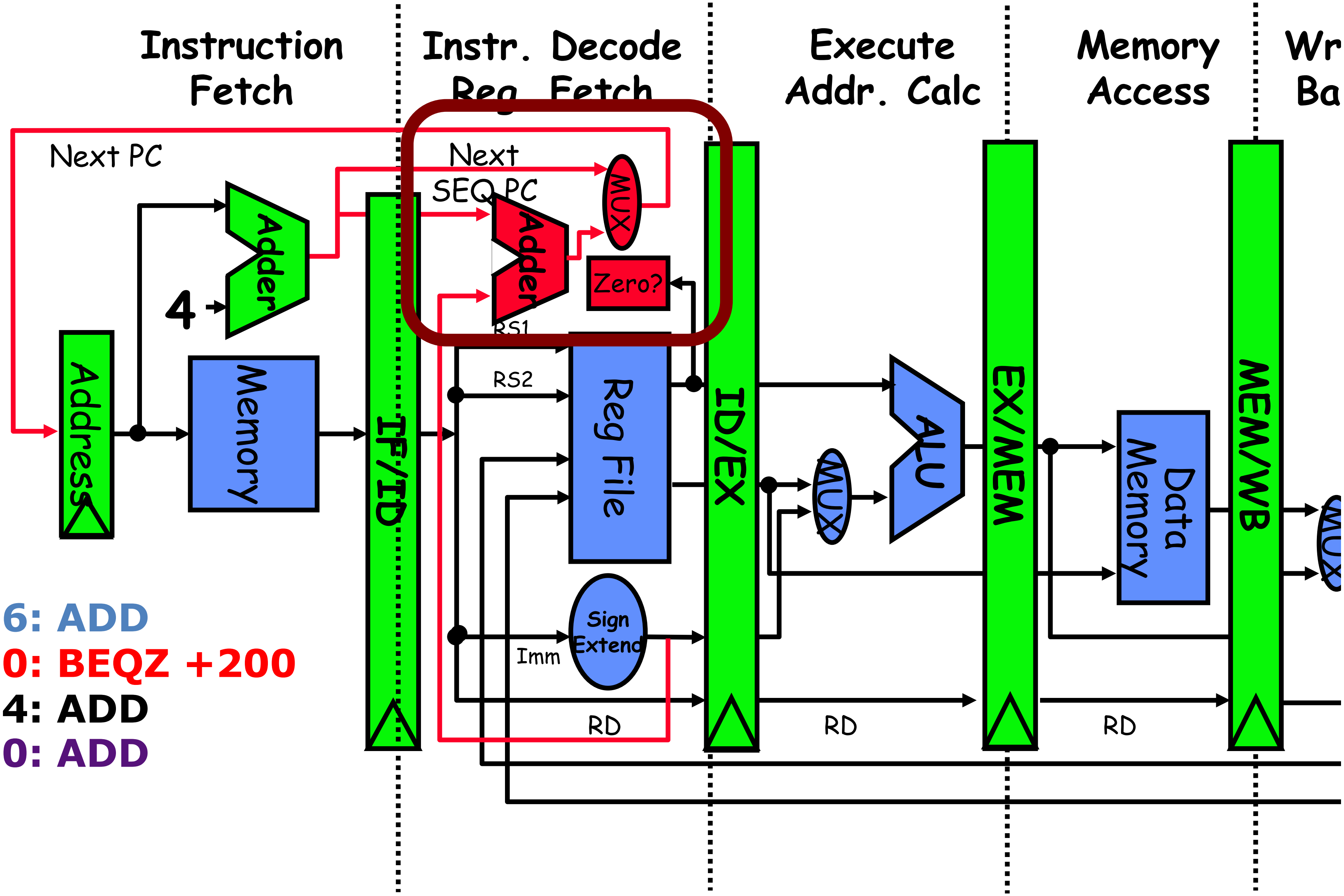
Resource Usage

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		
ME				I_1	I_2	-	I_4	I_5	
WB					I_1	I_2	-	I_4	I_5

- ⇒ ***pipeline bubble***

Pipelined MIPS Datapath



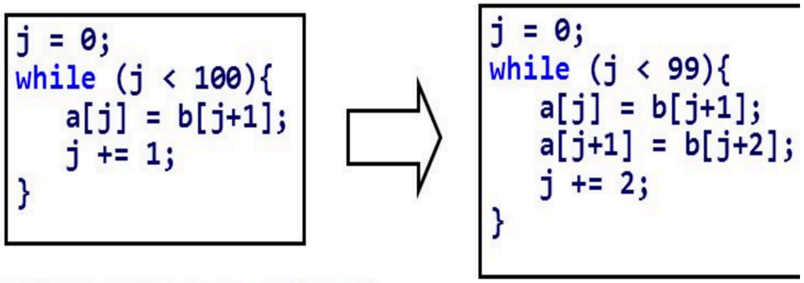
(I$_1$) 096: ADD
(I$_2$) 100: BEQZ +200
(I$_3$) 104: ADD
(I$_4$) 300: ADD

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**

Additional Materials – Branch Prediction

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

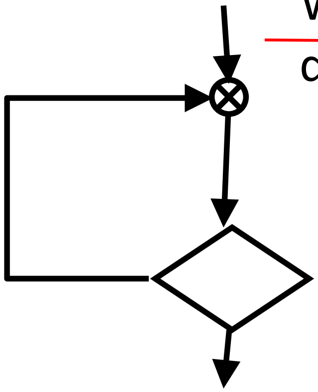
Static Branch Prediction

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

What C++ statement does this look like

backward

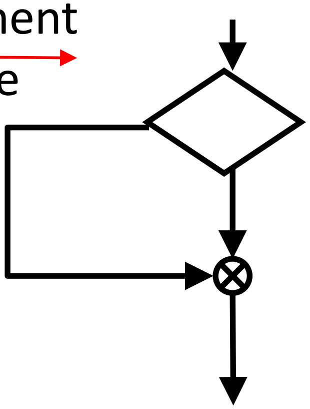
90%



What C++ statement does this look like

forward

50%



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

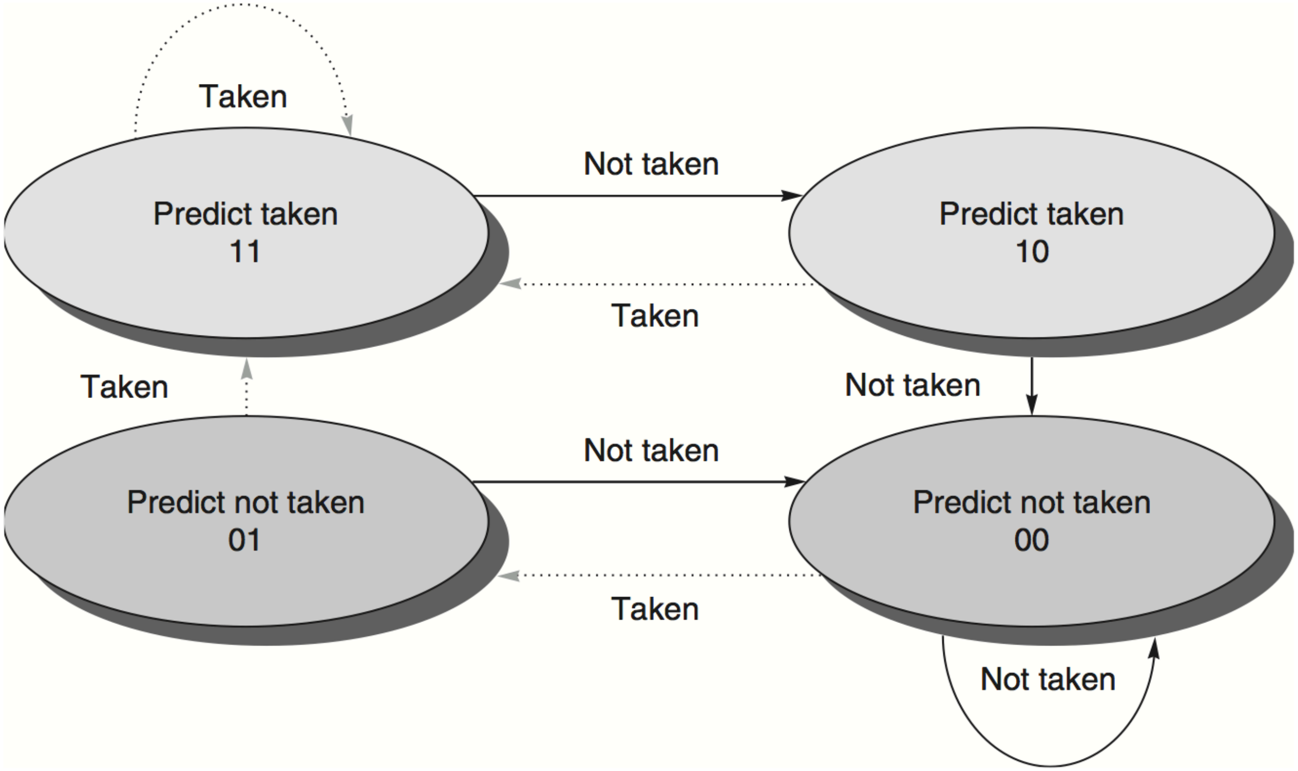
- Spatial correlation (space)
  - Several branches may resolve in a highly correlated manner
  - For instance, a preferred path of execution

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

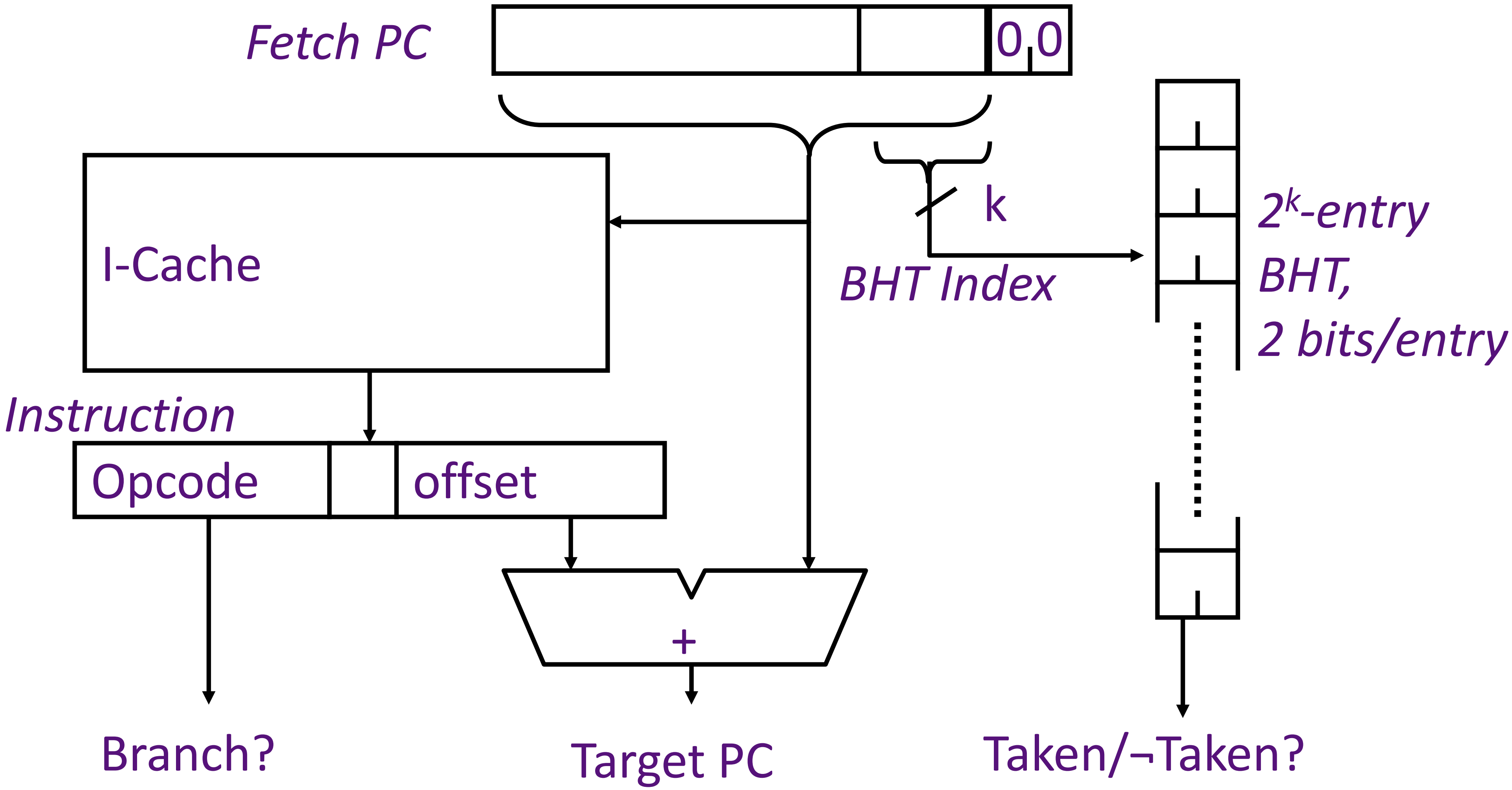
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



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

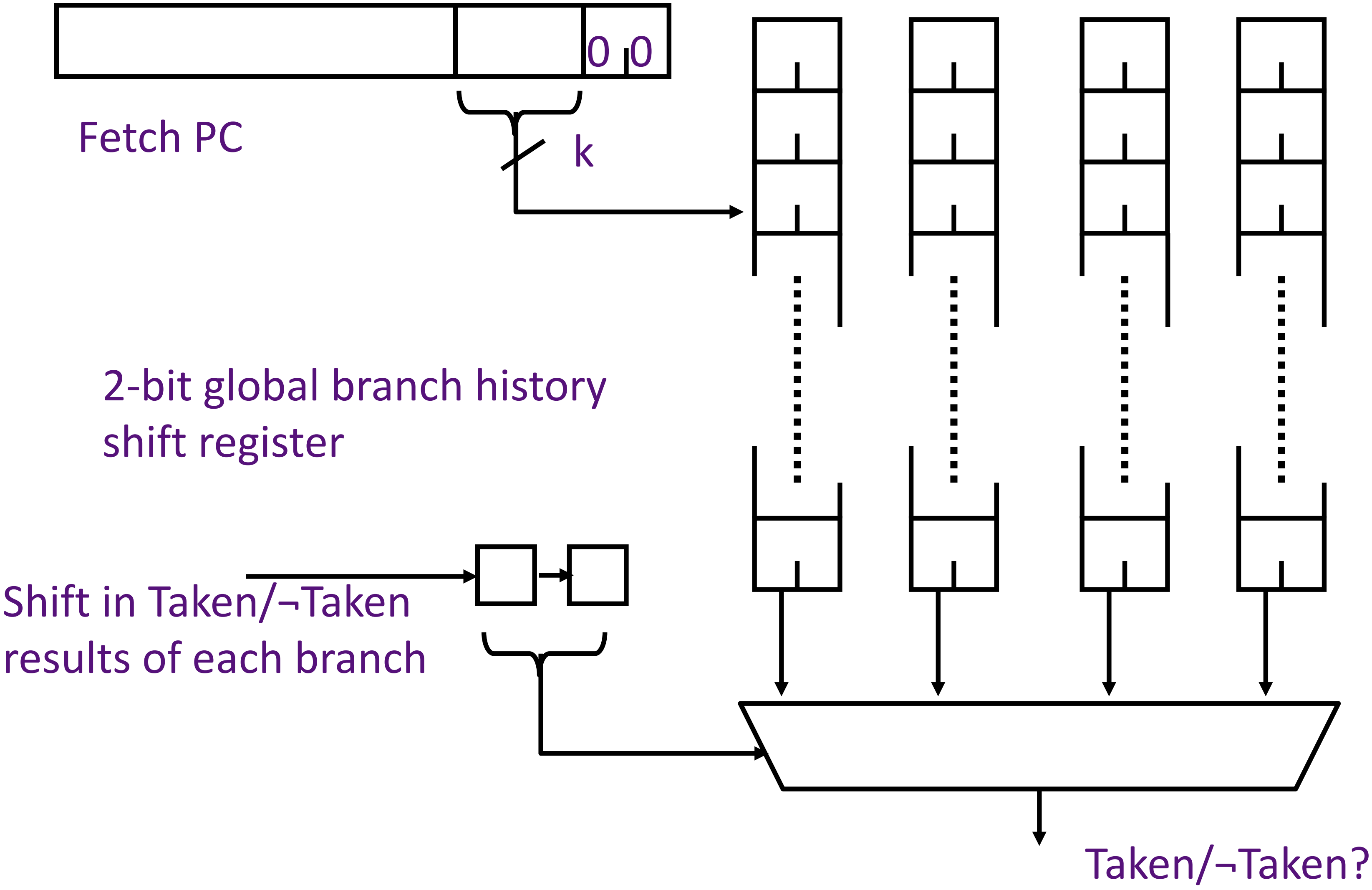
```
if (x[i] < 7) then
   y += 1;
if (x[i] < 5) then
   c -= 4;
```

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)



Speculating Both Directions

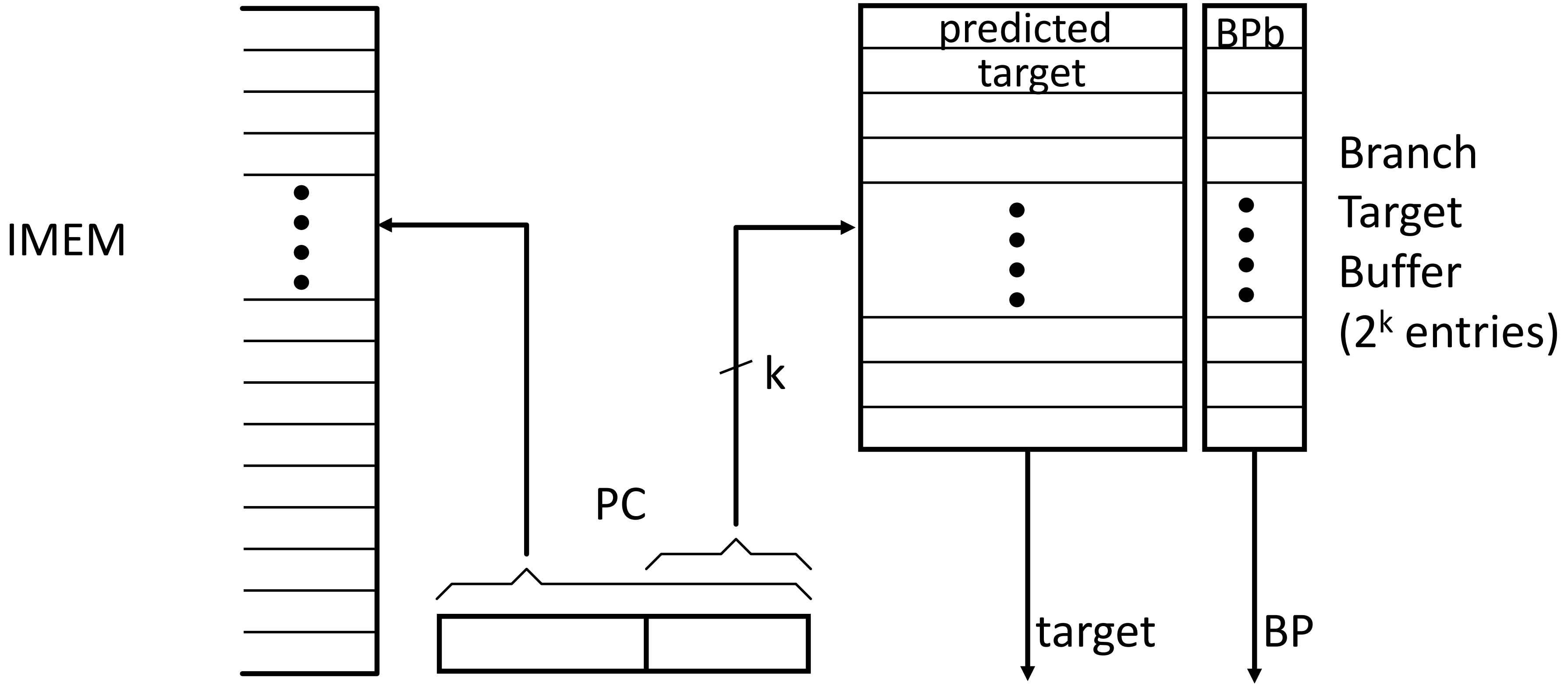
- An alternative to branch prediction is to execute both directions of a branch speculatively
  - resource requirement is proportional to the number of concurrent speculative executions
  - only half the resources engage in useful work when both directions of a branch are executed speculatively
  - branch prediction takes less resources than speculative execution of both paths
- With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction!
  - What would you choose with 80% accuracy?

Are We Missing Something?

- Knowing whether a branch is taken or not is great, but what else do we need to know about it?

Branch target address

Branch Target Buffer

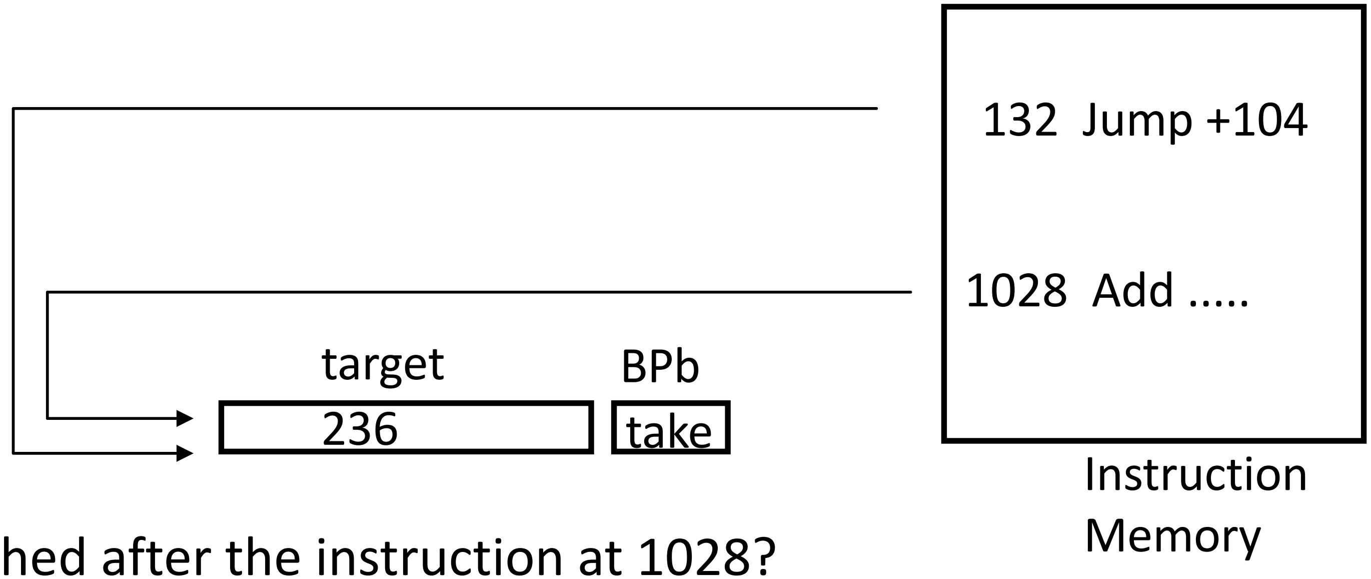


BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
Later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

Address Collisions (MisPrediction)

Assume a 128-entry BTB



132 Jump +104

1028 Add .....

target

BPb

236

take

Instruction Memory

What will be fetched after the instruction at 1028?

BTB prediction = 236

Correct target = 1032

=> *kill* PC=236 and *fetch* PC=1032

Is this a common occurrence?

BTB is only for Control Instructions

- Is even branch prediction fast enough to avoid bubbles?
- When do we index the BTB?
  - i.e., what state is the branch in, in order to avoid bubbles?

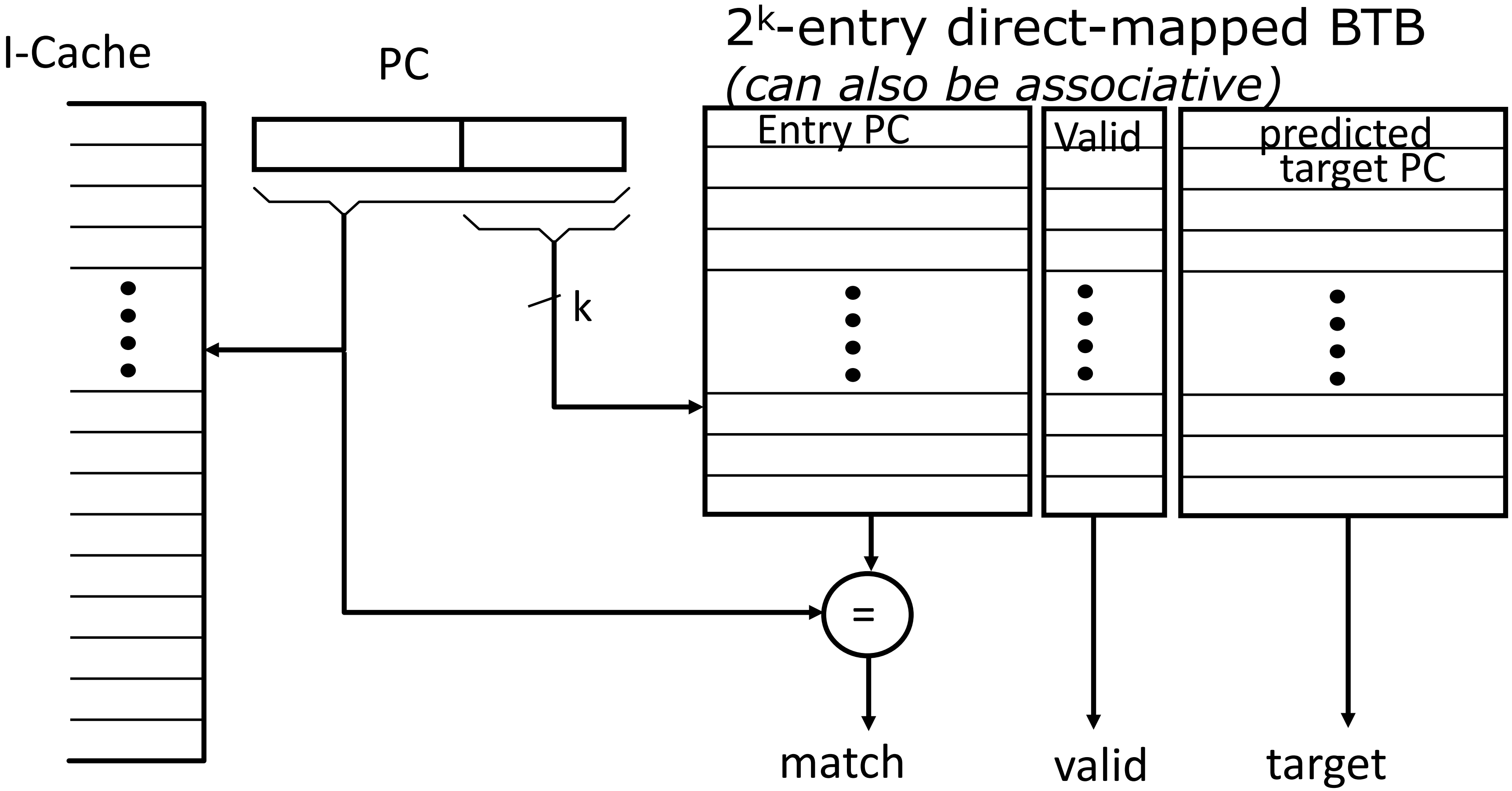
- **BTB contains useful information for branch and jump instructions only**

  => Do not update it for other instructions

- For all other instructions the next PC is PC+4 !

- *How to achieve this effect without decoding the instruction?*

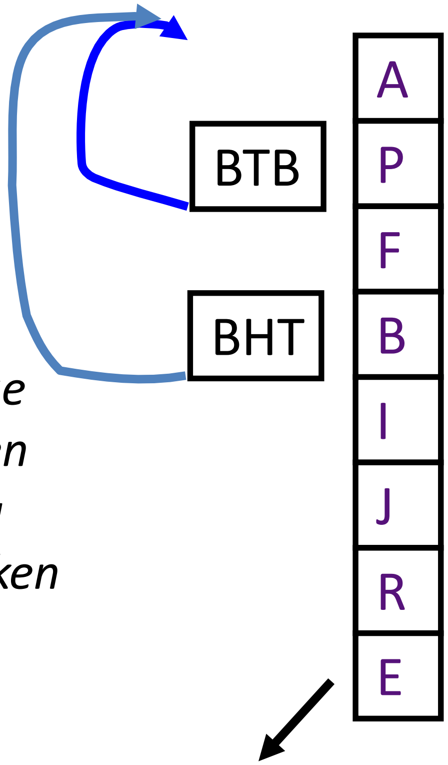
Branch Target Buffer (BTB)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate



A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

BHT in later pipeline stage corrects when BTB misses a predicted taken branch

BTB/BHT only updated after branch resolves in E stage

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)

  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if usually return to the same place

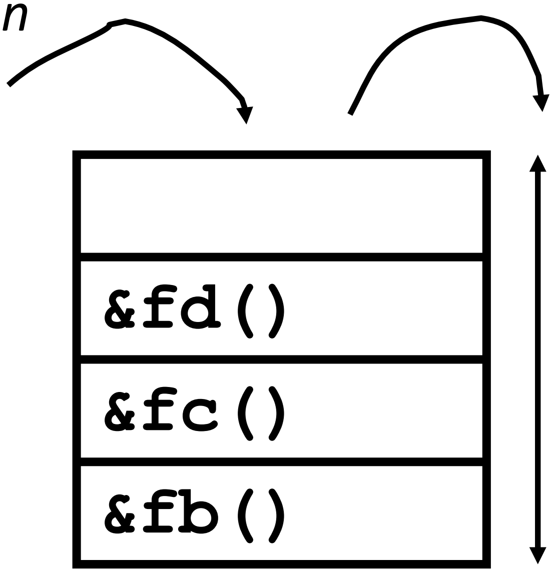
  ⇒ *Often one function called from many distinct call sites!*

How well does BTB work for each of these cases?

Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```



Push call address when function call executed

Pop return address when subroutine return decoded

&fd()

&fc()

&fb()

k entries (typically k=8-16)