Lecture 09: RISC-V Pipeline Implementation

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

Acknowledgement

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

Introduction

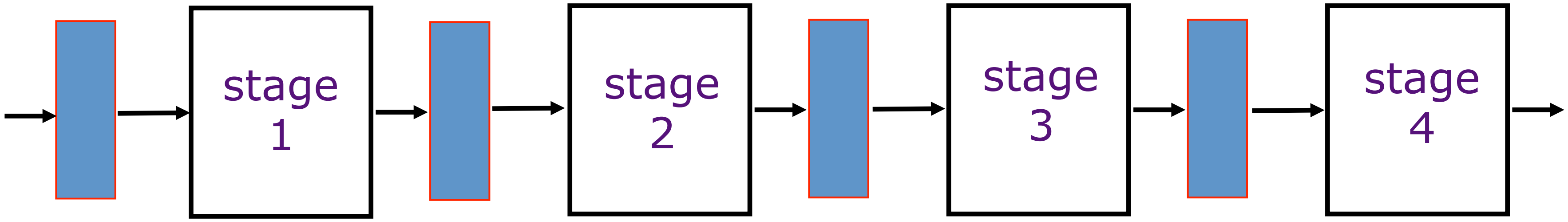
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

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

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

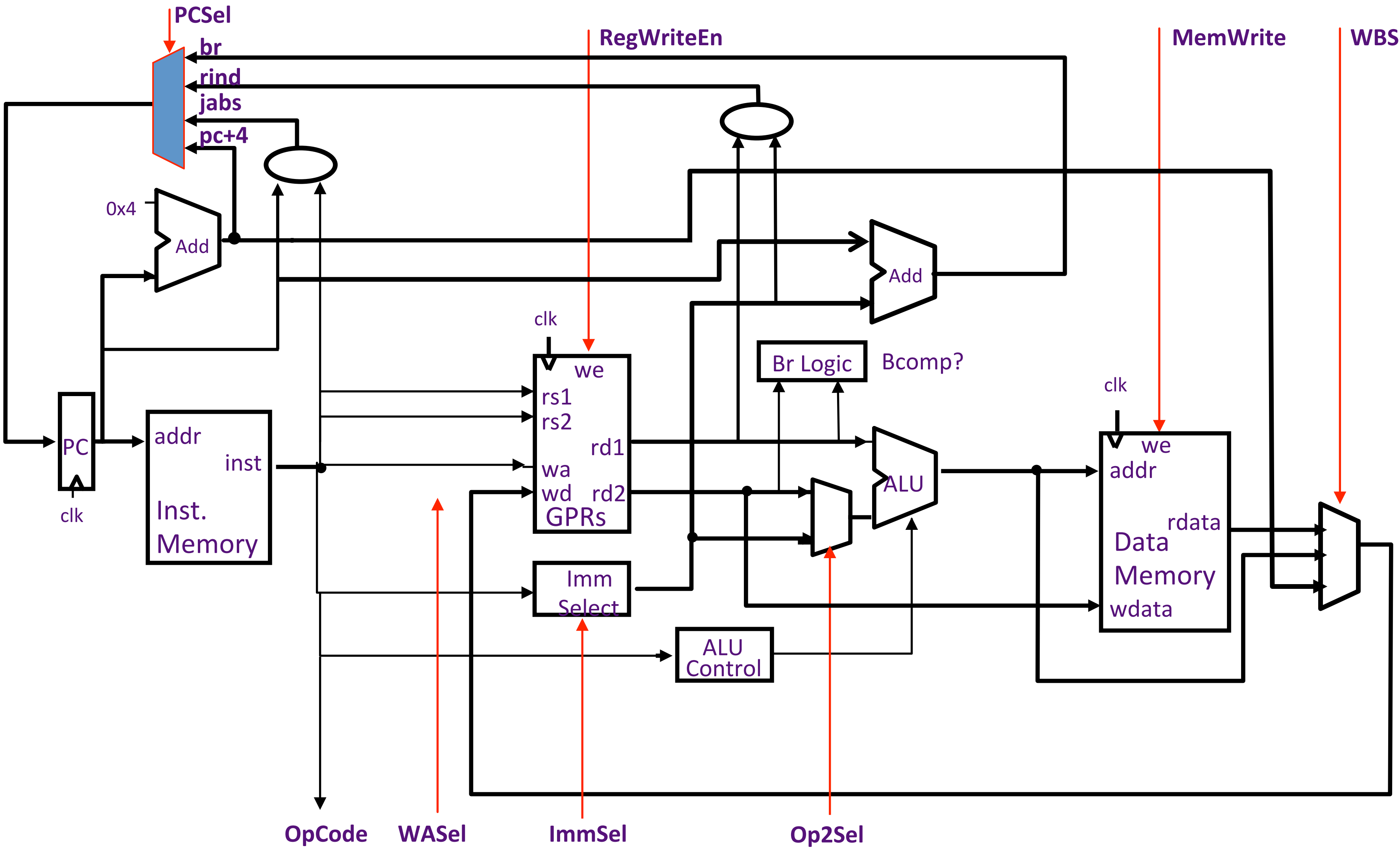
An Ideal Pipeline



- 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!

Review: Unpipelined Datapath for RISC-V



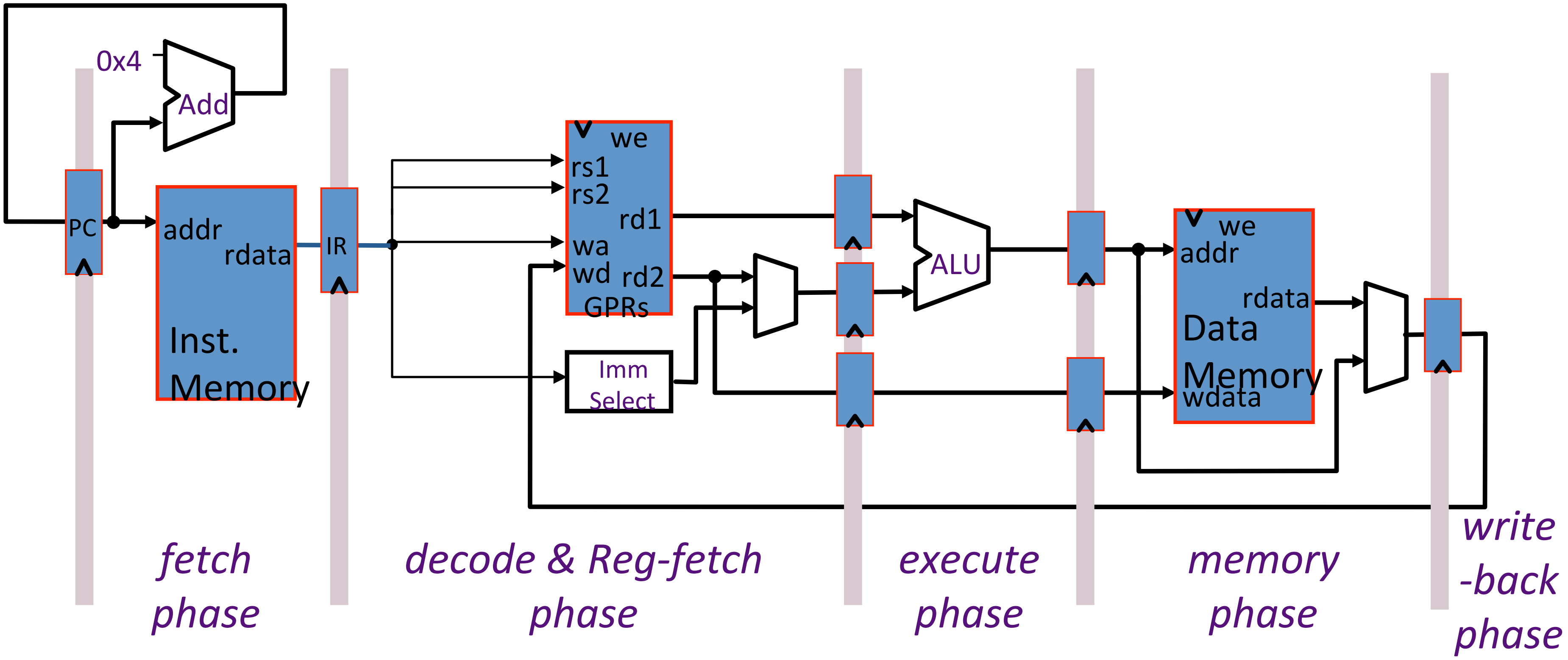
Review: Hardwired Control Table

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	BsType_{12}	Imm	+	yes	no	*	*	pc+4
BEQ_{true}	BrType_{12}	*	*	no	no	*	*	br
$\text{BEQ}_{\text{false}}$	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

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

Technology Assumptions

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

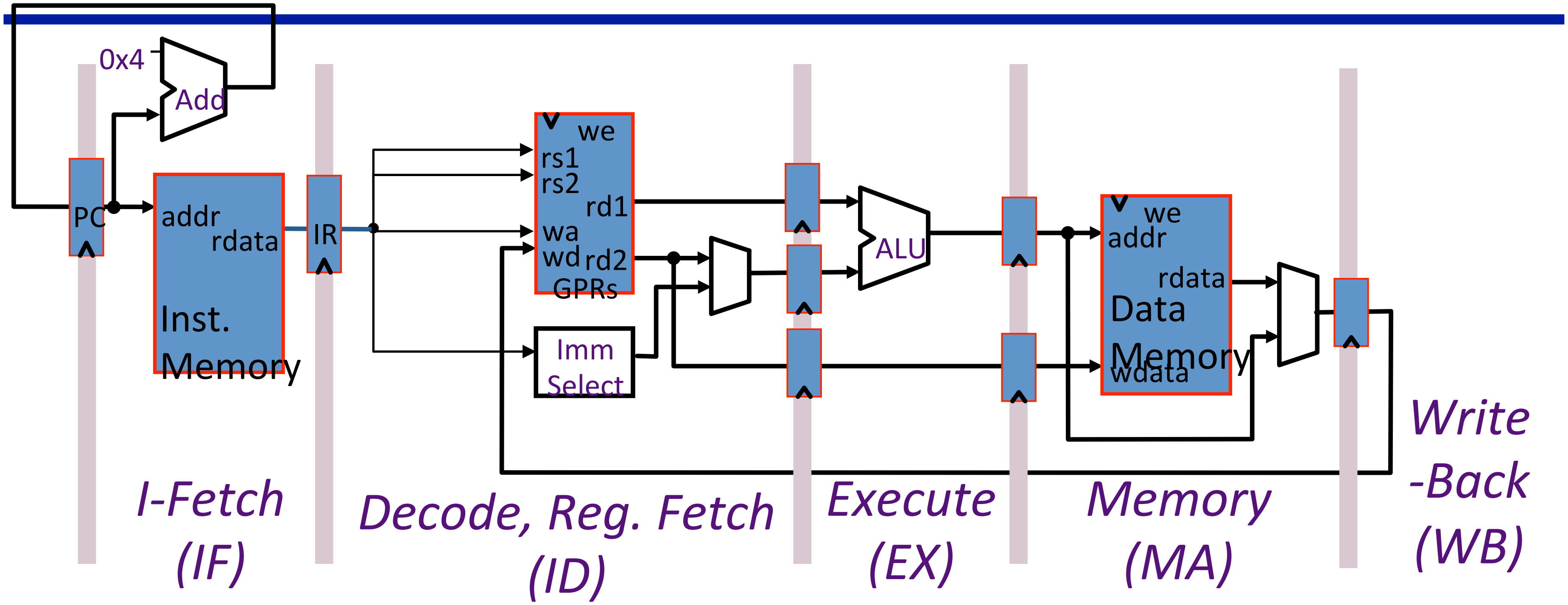
Thus, the following timing assumption is reasonable

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

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

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

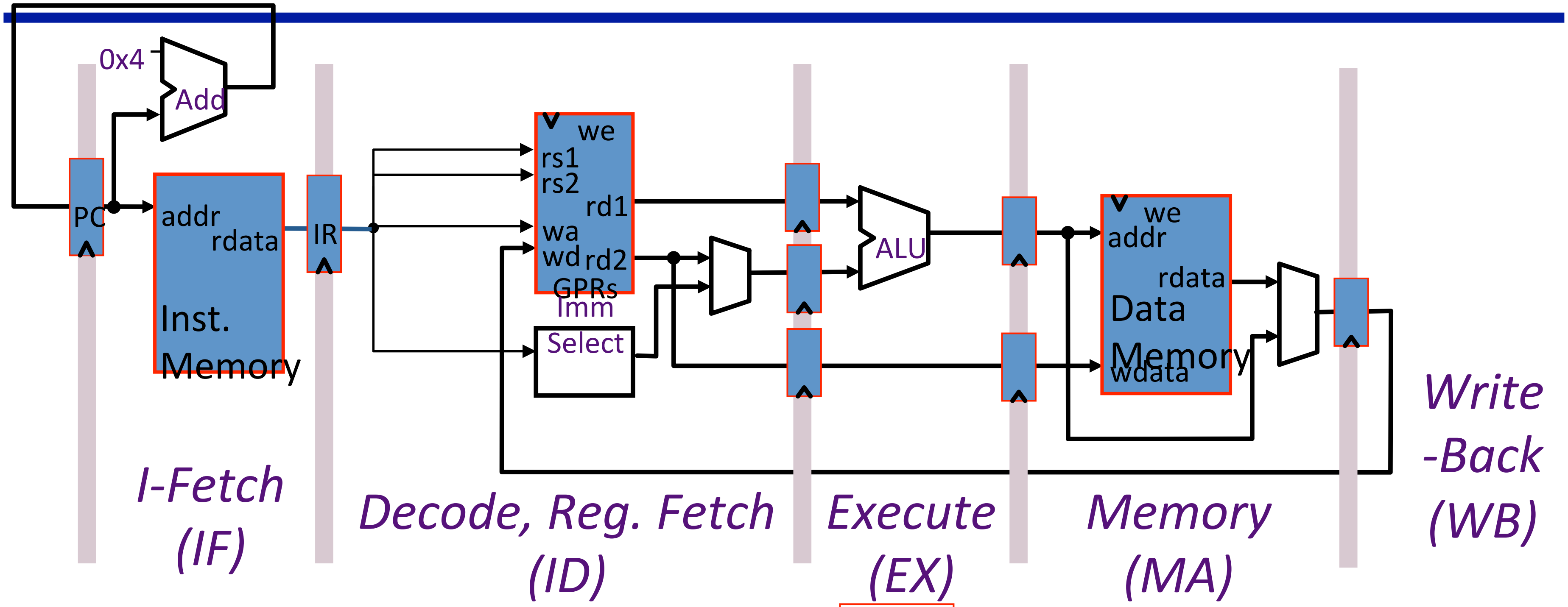
5-Stage Pipelined Execution



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

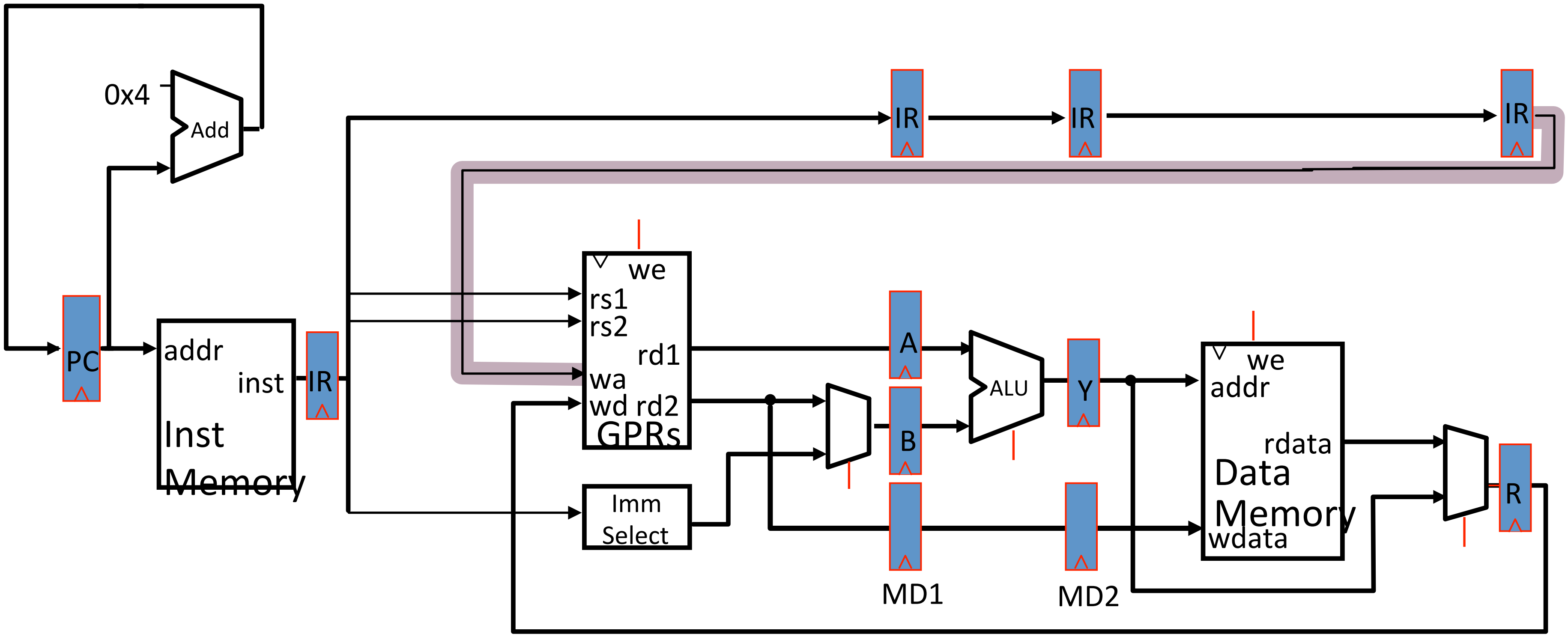
Resource Usage Diagram



	time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
Resources	*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

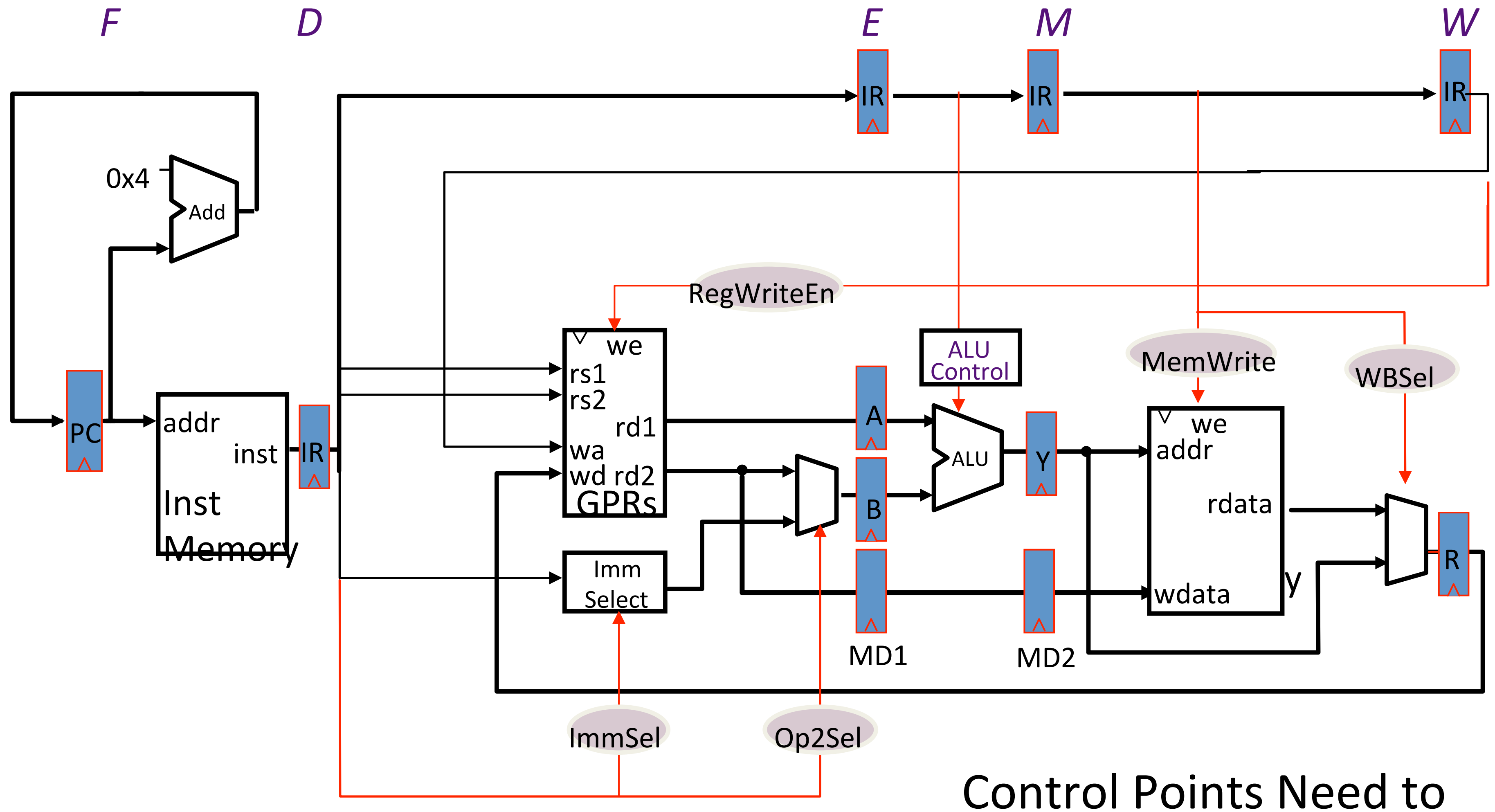


Not quite correct!

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

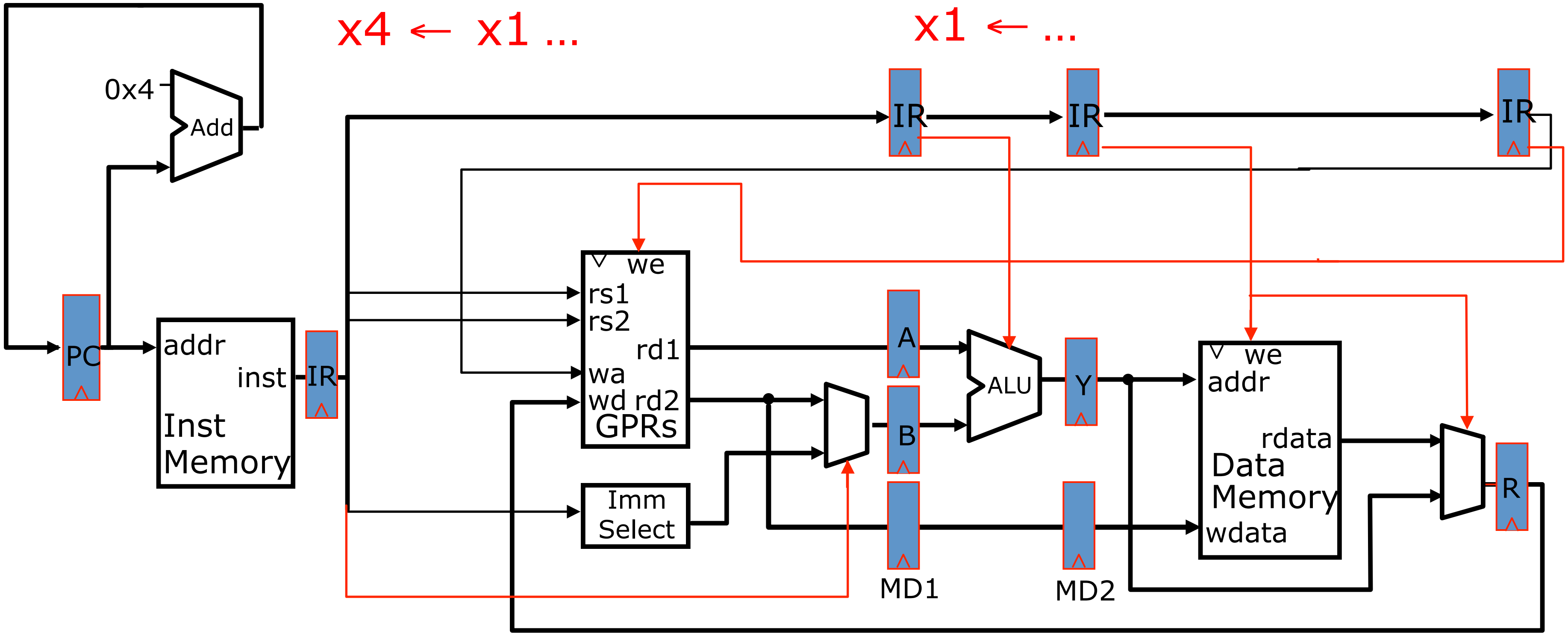
- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*

- 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?

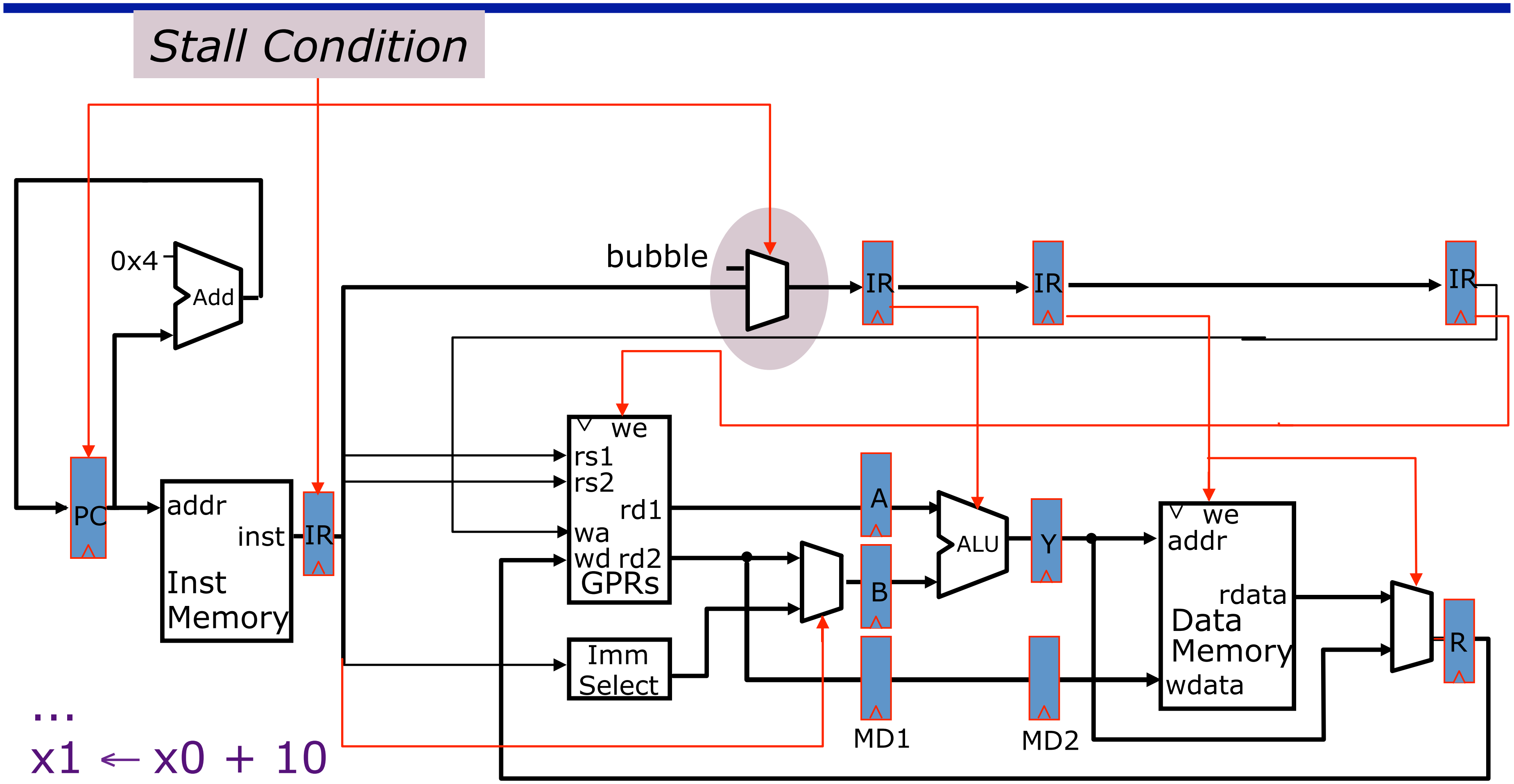
- Three options
  - Wait (stall)
  - Bypass: ask them for what you need before his/her final deliverable
  - ~~Speculate on values to read~~

Resolving Data Hazards (1)

Strategy 1:

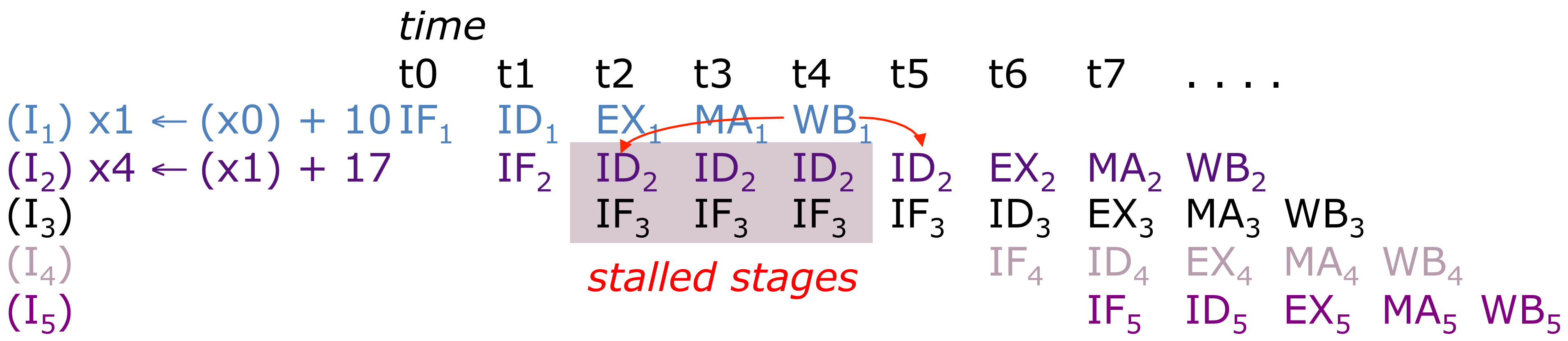
Wait for the result to be available by freezing earlier pipeline stages ➔ interlocks

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

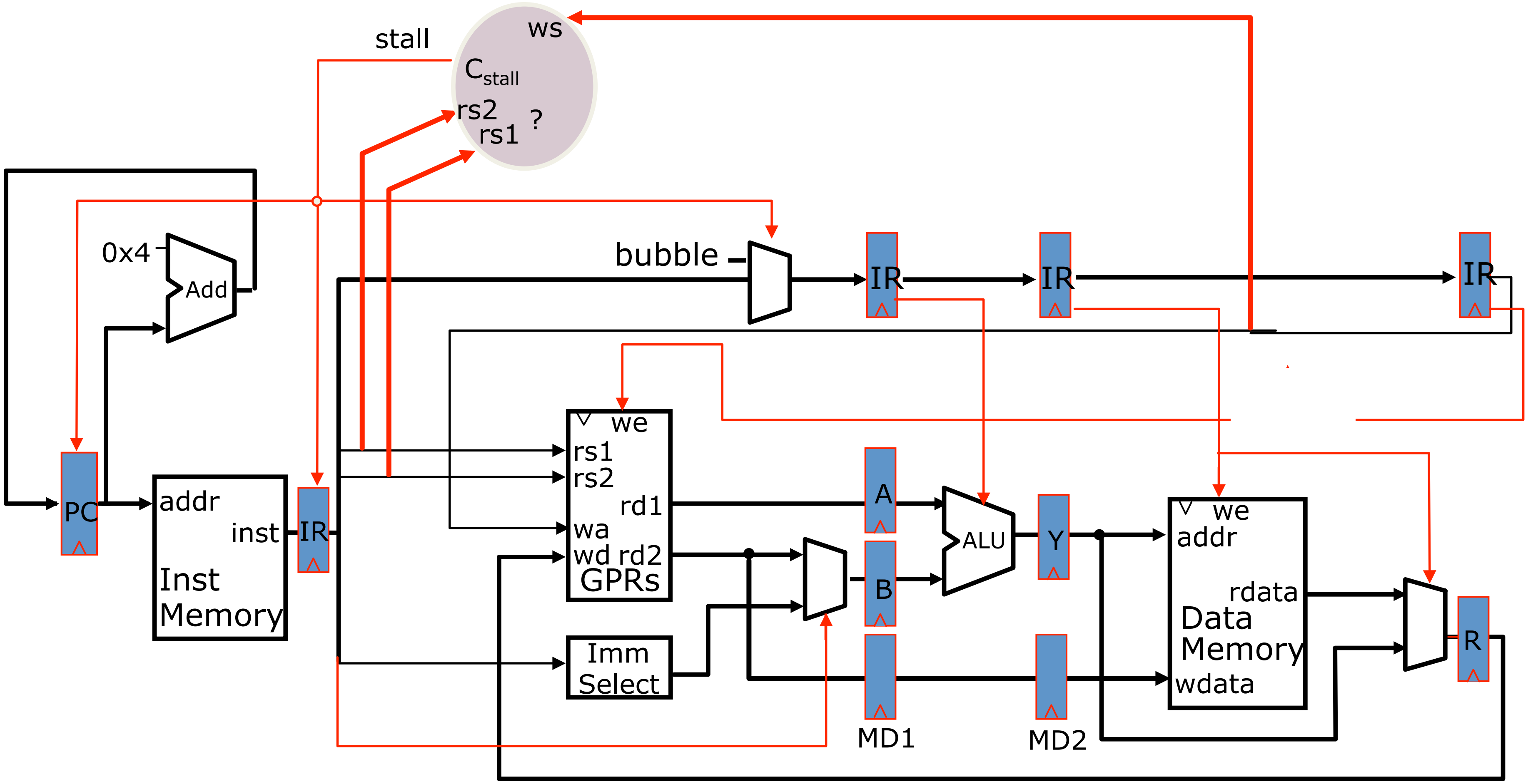
stalled stages

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.

we: write enable, 1-bit on/off

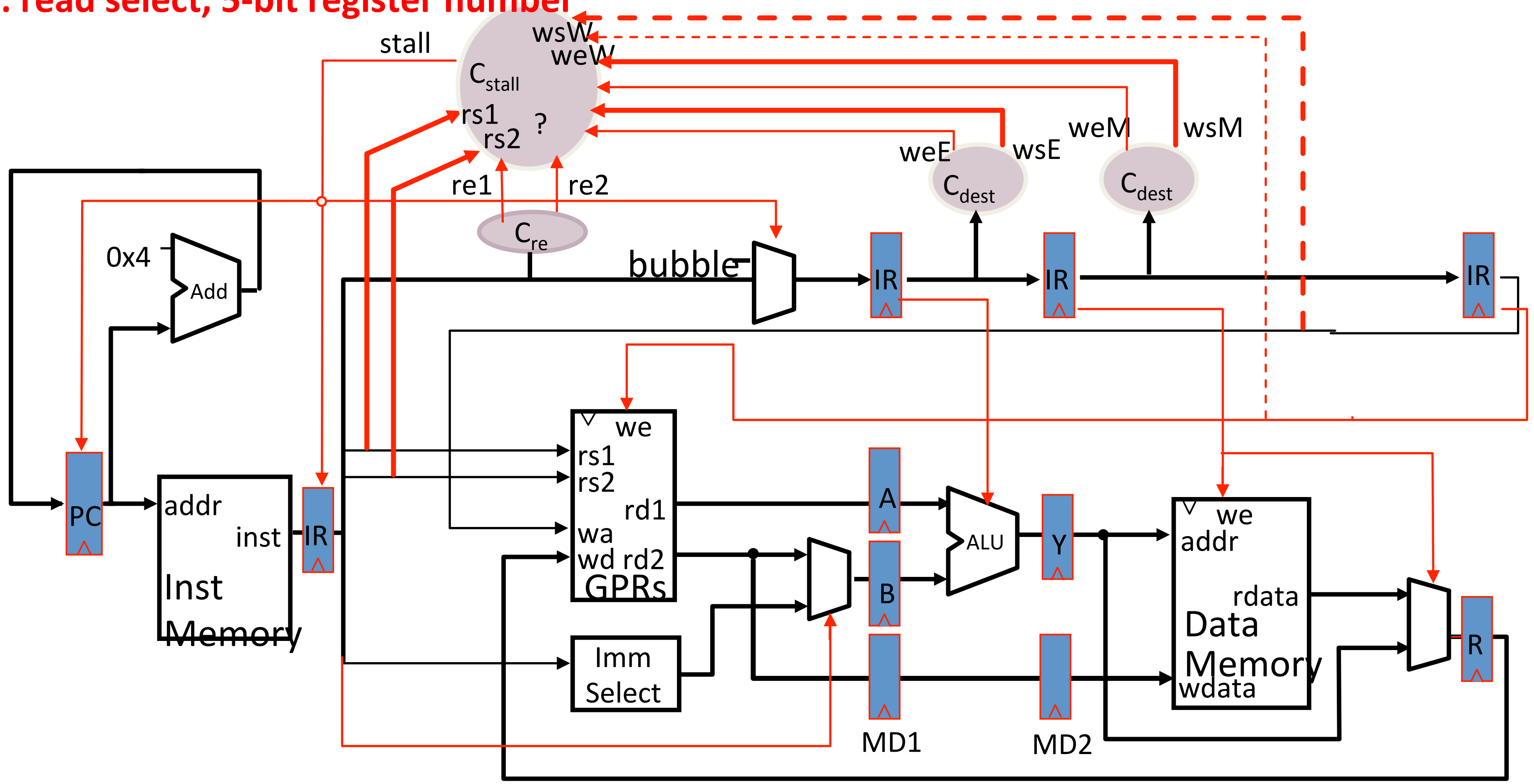
ws: write select, 5-bit register number

re: read enable, 1-bit on/off

rs: read select, 5-bit register number

Interlock Control Logic

ignoring jumps & branches

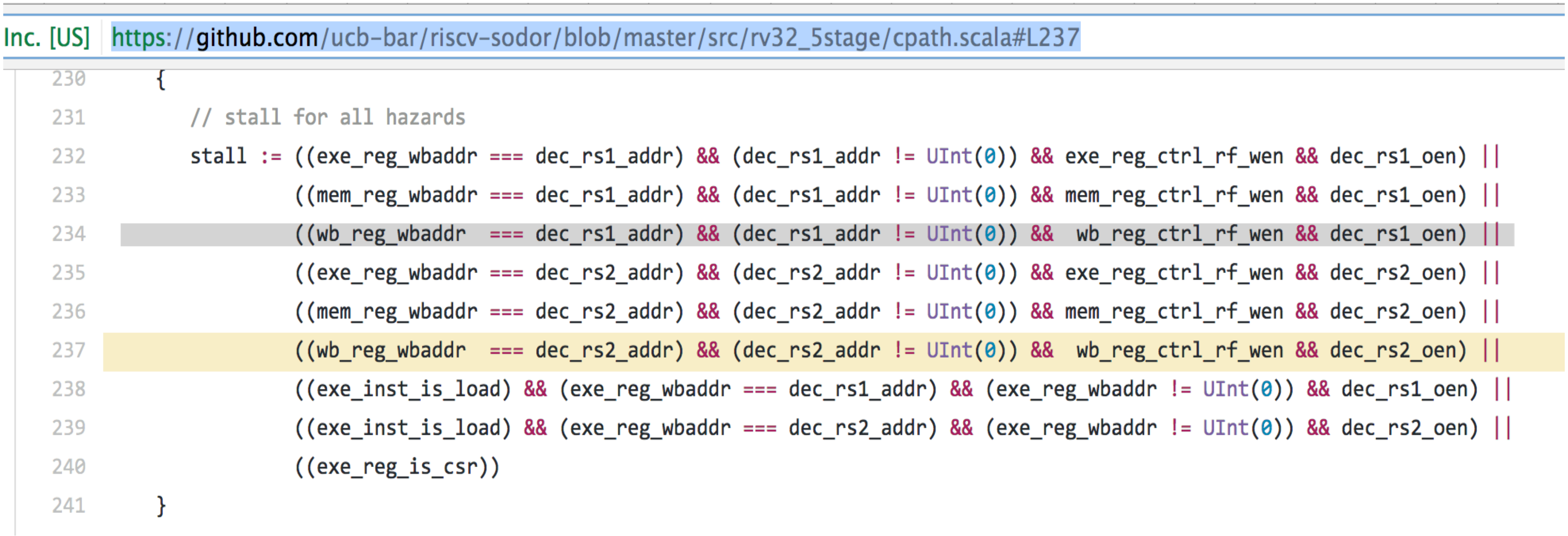


Should we always stall if an rs field matches some rd?

not every instruction writes a register => we

not every instruction reads a register => re

In RISC-V Sodor Implementation



Inc. [US] https://github.com/ucb-bar/riscv-sodor/blob/master/src/rv32_5stage/cpath.scala#L237

```
   {
      // stall for all hazards
      stall := ((exe_reg_wbaddr === dec_rs1_addr) && (dec_rs1_addr != UInt(0)) && exe_reg_ctrl_rf_wen && dec_rs1_oen) ||
               ((mem_reg_wbaddr === dec_rs1_addr) && (dec_rs1_addr != UInt(0)) && mem_reg_ctrl_rf_wen && dec_rs1_oen) ||
               ((wb_reg_wbaddr  === dec_rs1_addr) && (dec_rs1_addr != UInt(0)) &&  wb_reg_ctrl_rf_wen && dec_rs1_oen) ||
               ((exe_reg_wbaddr === dec_rs2_addr) && (dec_rs2_addr != UInt(0)) && exe_reg_ctrl_rf_wen && dec_rs2_oen) ||
               ((mem_reg_wbaddr === dec_rs2_addr) && (dec_rs2_addr != UInt(0)) && mem_reg_ctrl_rf_wen && dec_rs2_oen) ||
               ((wb_reg_wbaddr  === dec_rs2_addr) && (dec_rs2_addr != UInt(0)) &&  wb_reg_ctrl_rf_wen && dec_rs2_oen) ||
               ((exe_inst_is_load) && (exe_reg_wbaddr === dec_rs1_addr) && (exe_reg_wbaddr != UInt(0)) && dec_rs1_oen) ||
               ((exe_inst_is_load) && (exe_reg_wbaddr === dec_rs2_addr) && (exe_reg_wbaddr != UInt(0)) && dec_rs2_oen) ||
               ((exe_reg_is_csr))
   }
```

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	-
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

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

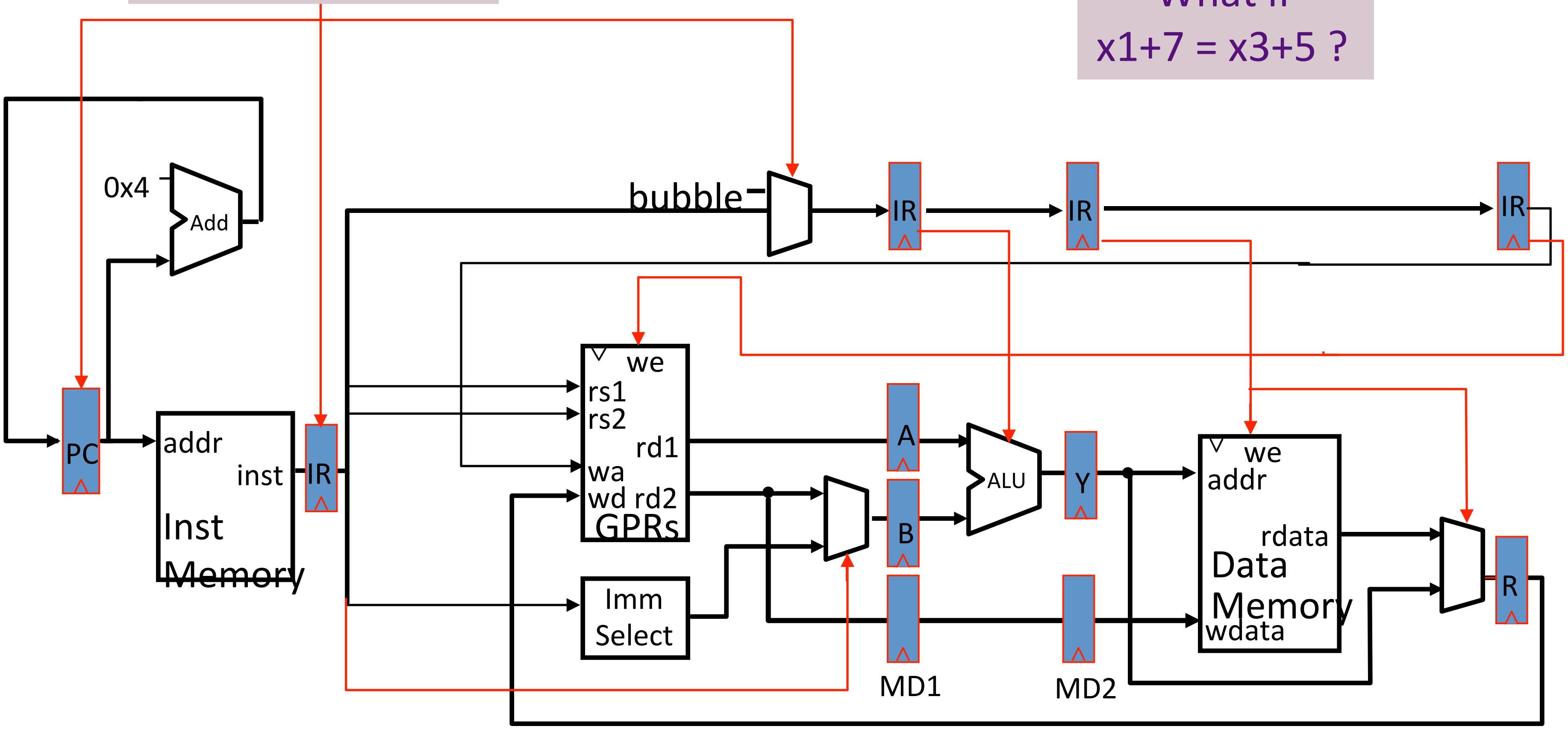
C_{stall}

stall = $((rs1_D == ws_E)\ \&\&\ we_E\ +$
$(rs1_D == ws_M)\ \&\&\ we_M\ +$
~~$(rs1_D == ws_W)\ \&\&\ we_W$~~$)\ \&\&\ re1_D\ +$
$((rs2_D == ws_E)\ \&\&\ we_E\ +$
$(rs2_D == ws_M)\ \&\&\ we_M\ +$
~~$(rs2_D == ws_W)\ \&\&\ we_W$~~$)\ \&\&\ re2_D$

Hazards due to Loads & Stores

Stall Condition

What if
x1+7 = x3+5 ?



...
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.

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) $x1 \leftarrow x0 + 10$	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) $x4 \leftarrow x1 + 17$		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

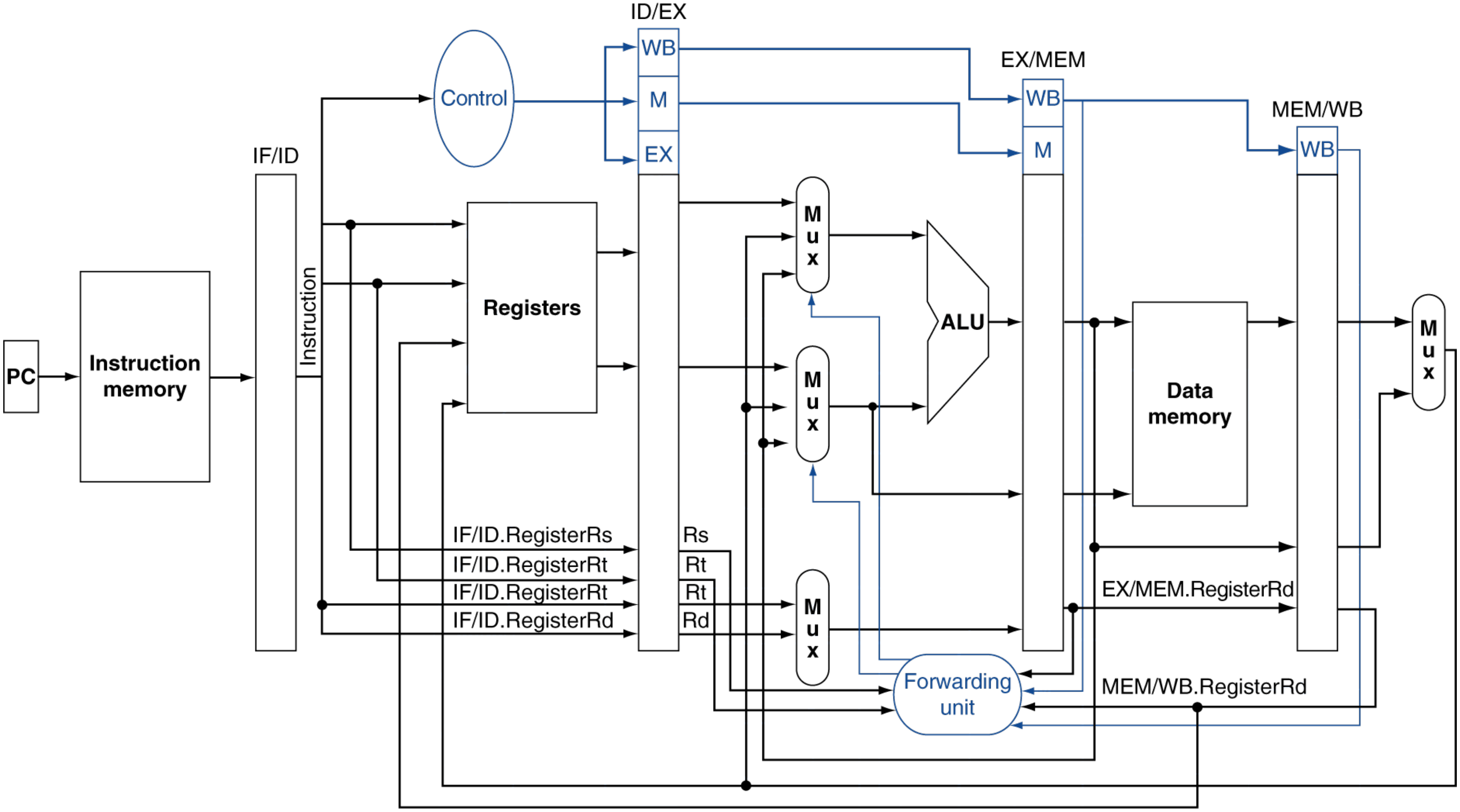
Each *stall or kill* introduces a bubble in the pipeline

=> *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) $x1 \leftarrow x0 + 10$		IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) $x4 \leftarrow x1 + 17$			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

Hardware Support for Forwarding



Detecting RAW Hazards

- Pass register numbers along pipeline
  - ID/EX.RegisterRs = register number for Rs in ID/EX
  - ID/EX.RegisterRt = register number for Rt in ID/EX
  - ID/EX.RegisterRd = register number for Rd in ID/EX
- Current instruction being executed in ID/EX register
- Previous instruction is in the EX/MEM register
- Second previous is in the MEM/WB register
- RAW Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from MEM/WB pipeline reg

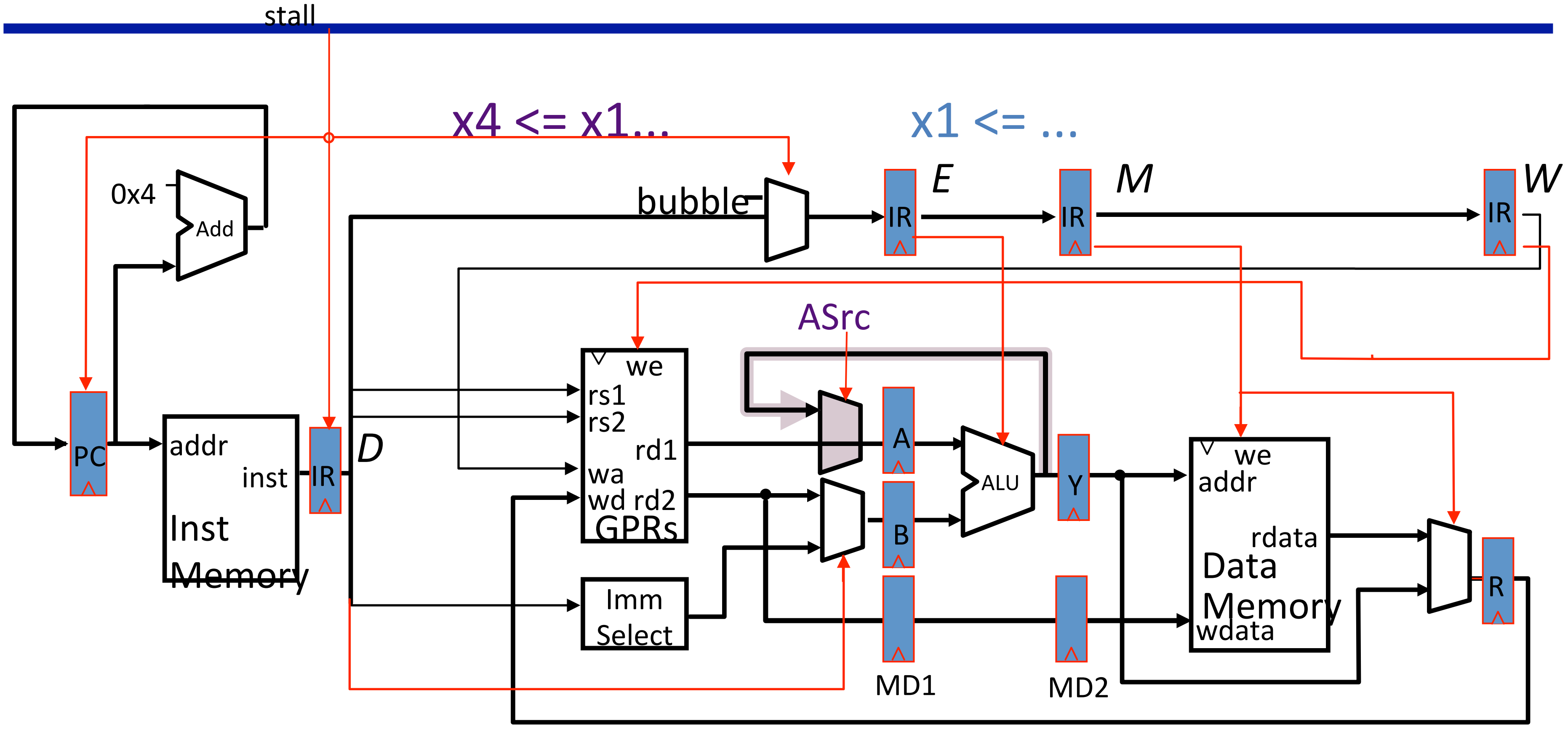
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not R0
  - EX/MEM.RegisterRd ≠ 0
  - MEM/WB.RegisterRd ≠ 0

Forwarding Conditions

- 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



When does this bypass help?

...

(I_1) x1 <= x0 + 10

(I_2) x4 <= x1 + 17

yes

x1 <= M[x0 + 10]

x4 <= x1 + 17

no

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

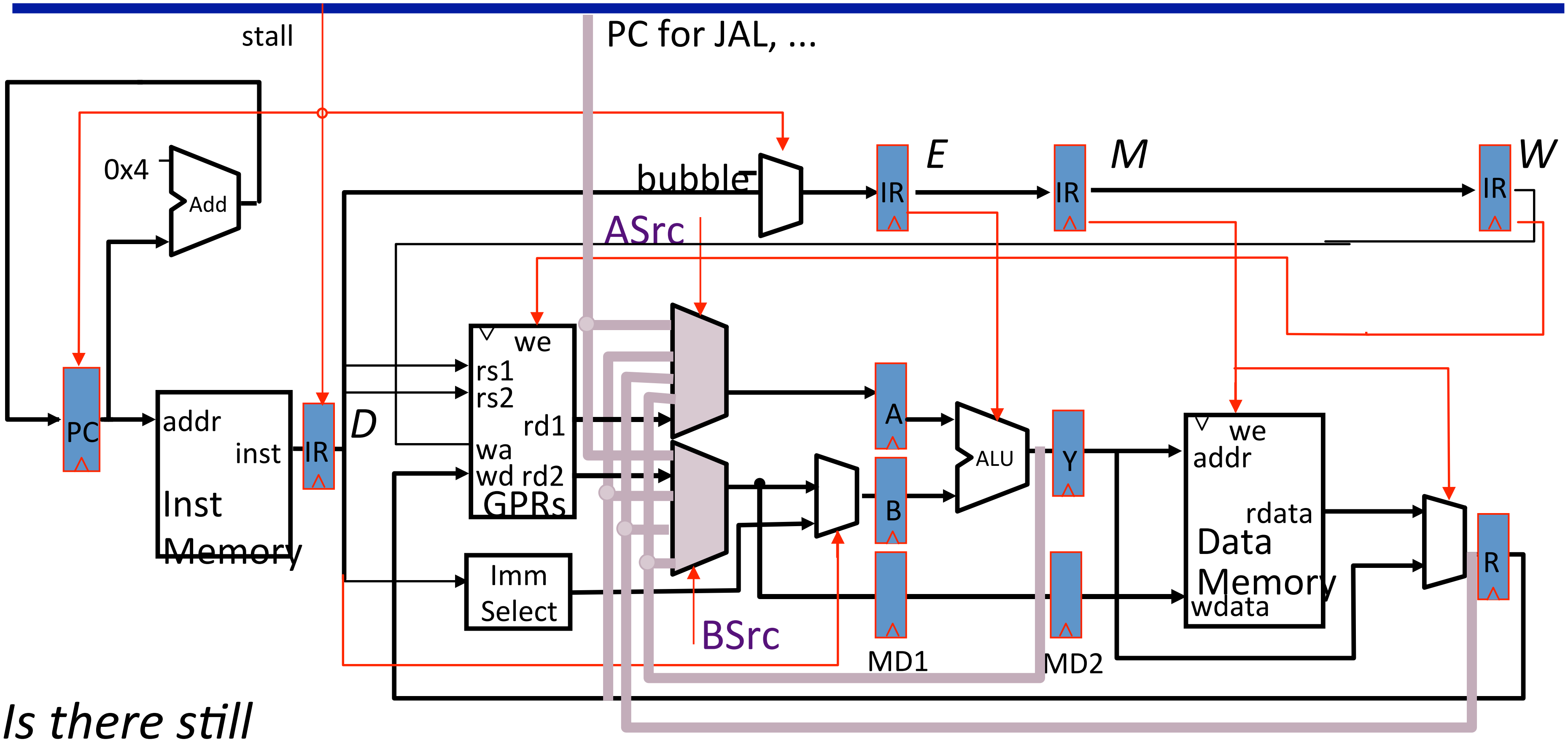
we-bypass$_E$ = *Case* opcode$_E$
ALU, ALUi => on
... => off

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 ?

$$
\begin{aligned}
\text{stall} = \ & (rs1_D == ws_E)\ \&\&\ (opcode_E == LW_E) \&\& (ws_E != 0\ ) \&\& re1_D \\
& + (rs2_D == ws_E)\ \&\&\ (opcode_E == LW_E) \&\& (ws_E != 0\ ) \&\& re2_D
\end{aligned}
$$

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

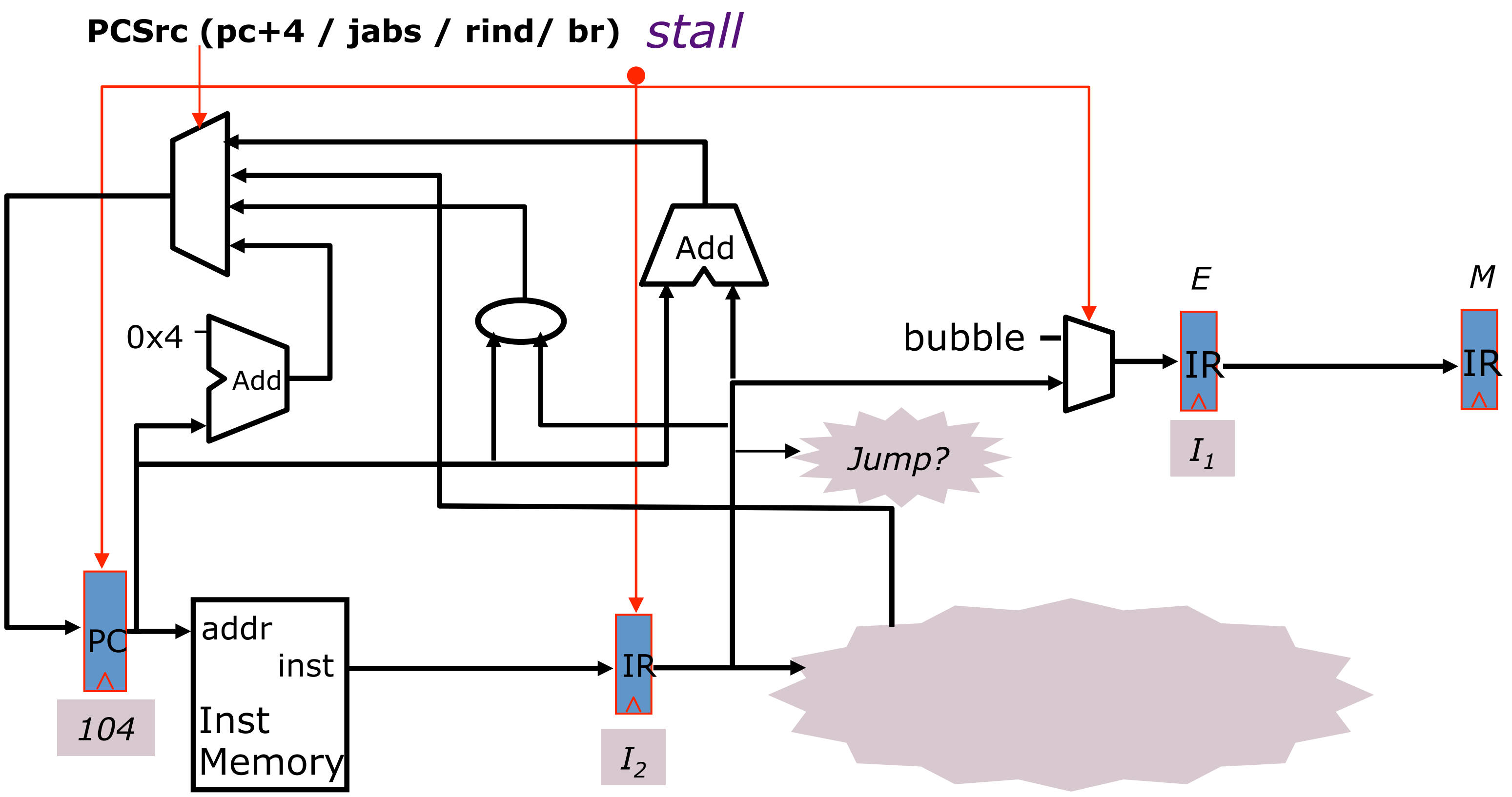
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
(I_1) $x1 \leftarrow x0 + 10$	IF$_1$	ID$_1$	EX$_1$	MA$_1$	WB$_1$						
(I_2) $x3 \leftarrow x2 + 17$		IF$_2$	IF$_2$	ID$_2$	EX$_2$	MA$_2$	WB$_2$				
(I_3)				IF$_3$	IF$_3$	ID$_3$	EX$_3$	MA$_3$	WB$_3$		
(I_4)						IF$_4$	IF$_4$	ID$_4$	EX$_4$	MA$_4$	WB$_4$

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

- $\Rightarrow$ *pipeline bubble*

Speculate next address is PC+4



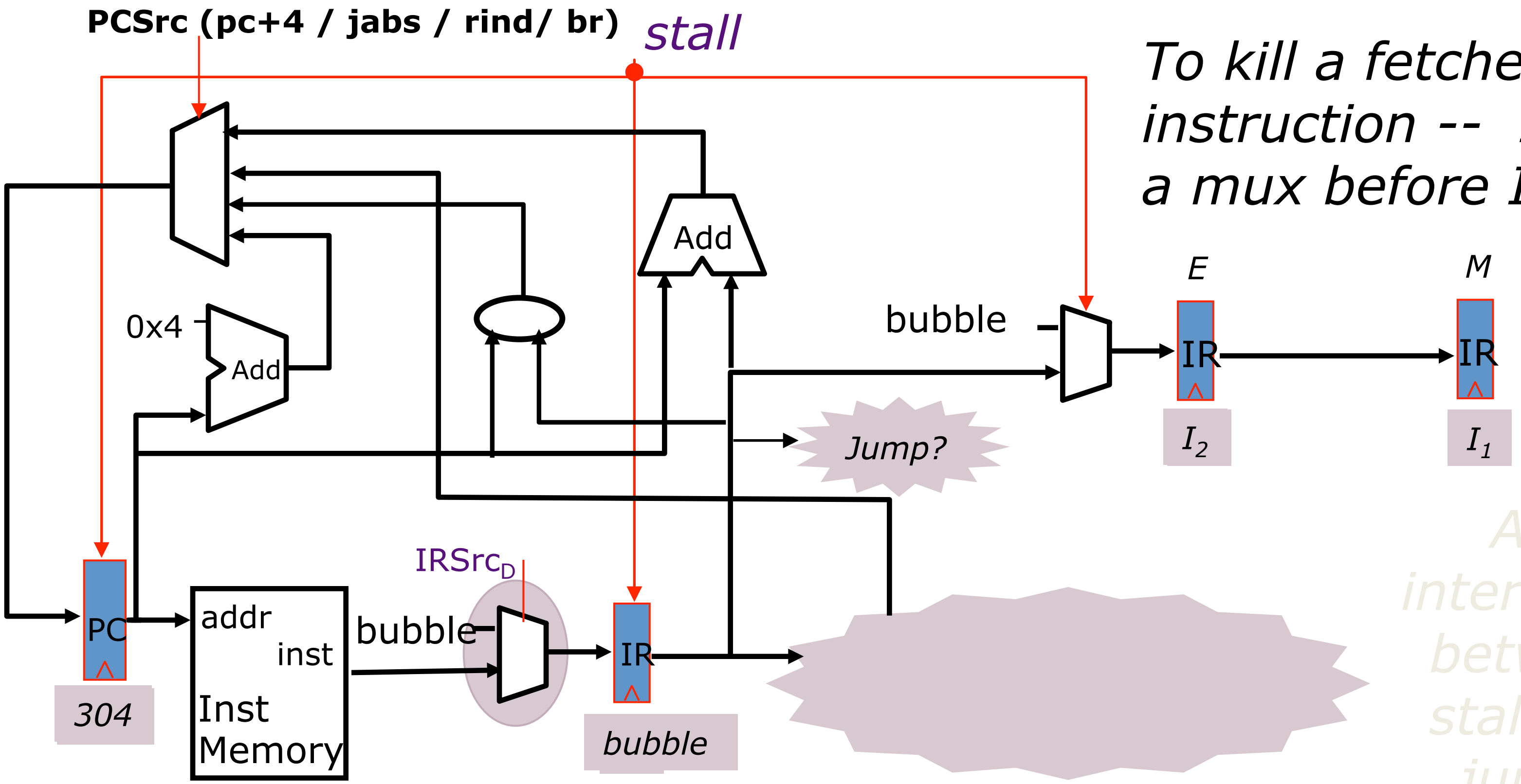
I_1	096	ADD
I_2	100	J 304
I_3	~~104~~	~~ADD~~
I_4	304	ADD

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?

$IRSrc_D$ = *Case* $opcode_D$
JAL ⇒ bubble
... ⇒ IM

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

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	

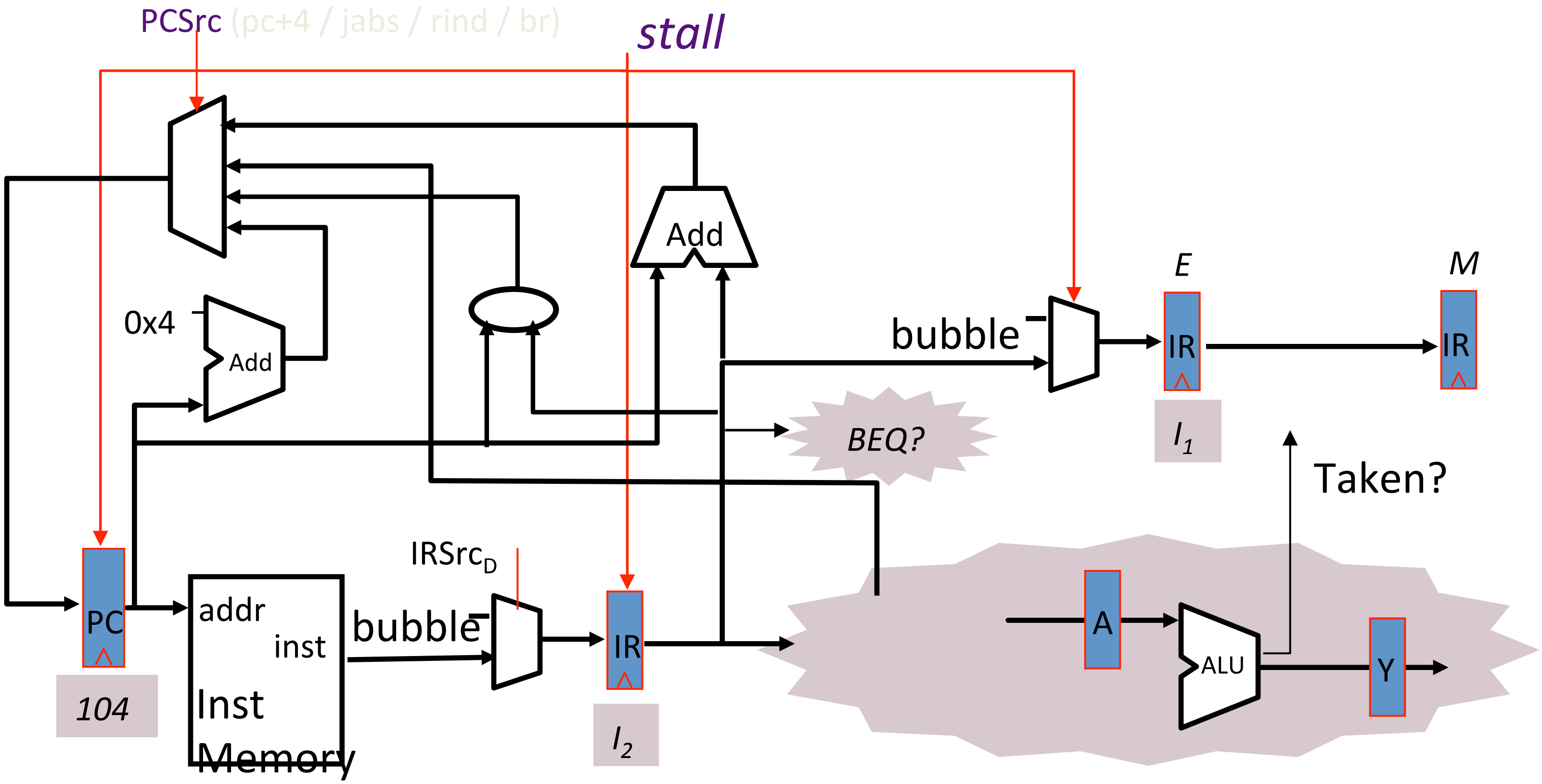
Resource Usage

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

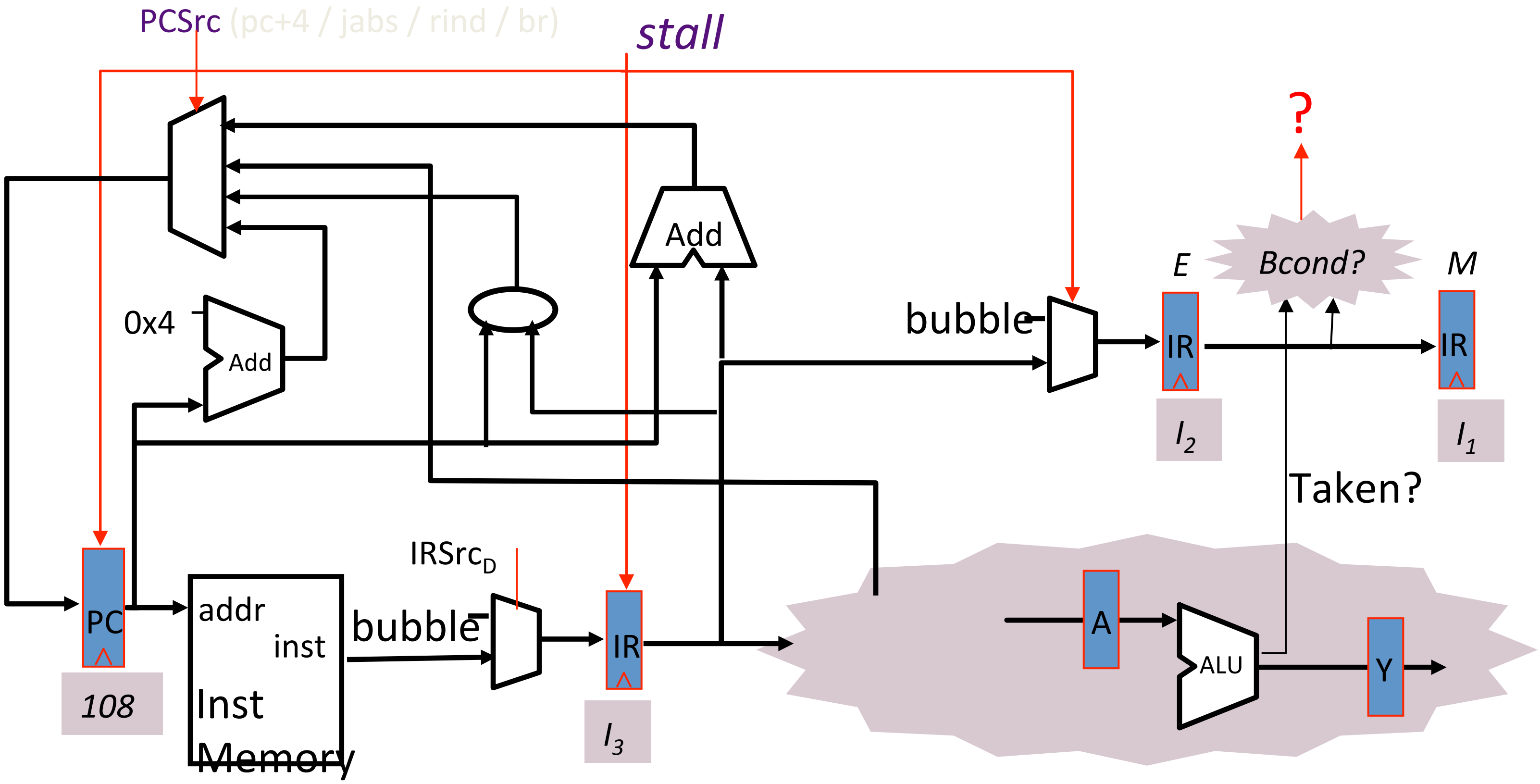


I_1	096 ADD
I_2	100 BEQ x1,x2 +200
I_3	104 ADD
I_4	304 ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage ?

Pipelining Conditional Branches

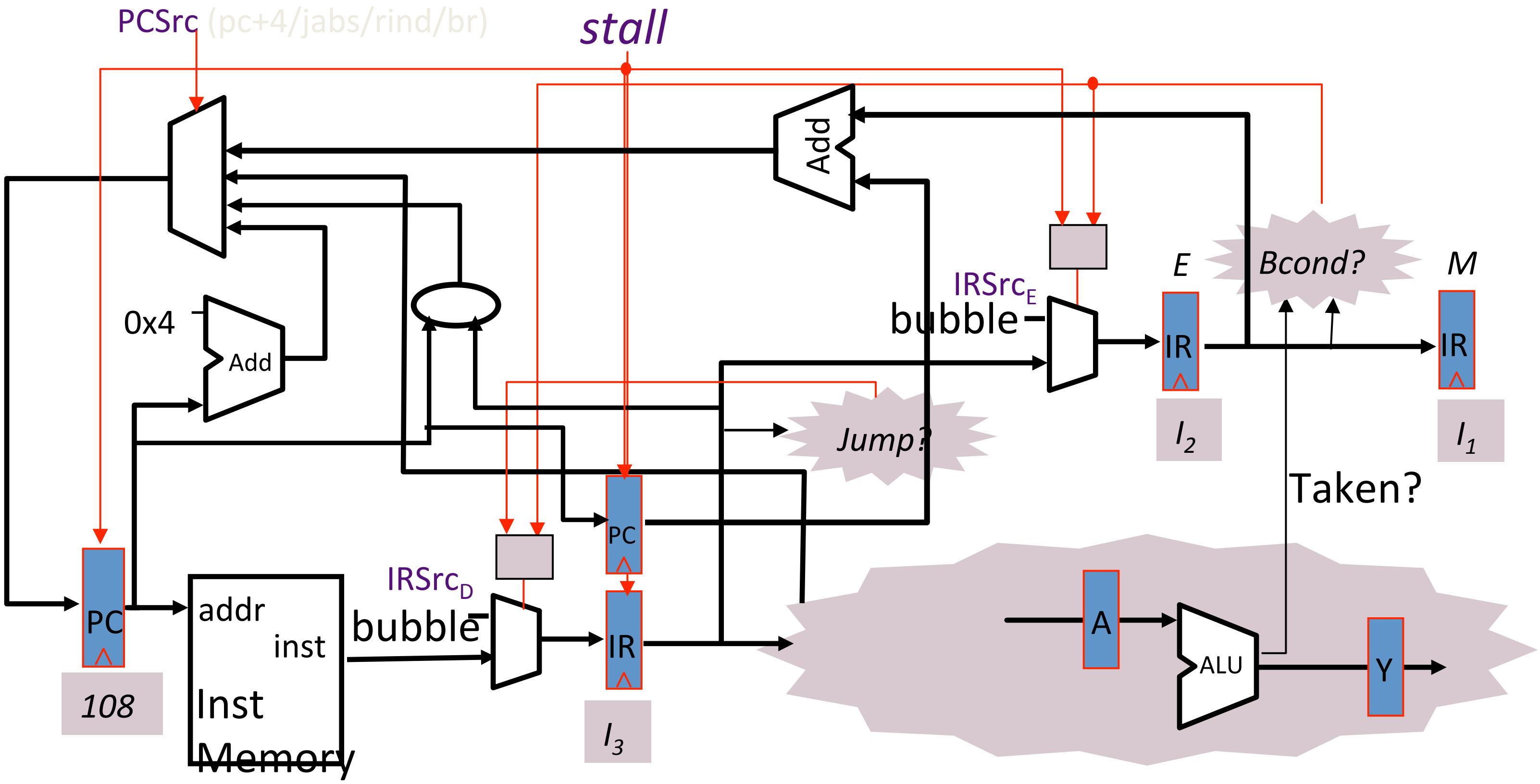


I_1	096 ADD
I_2	100 BEQ x1,x2 +200
I_3	104 ADD
I_4	304 ADD

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

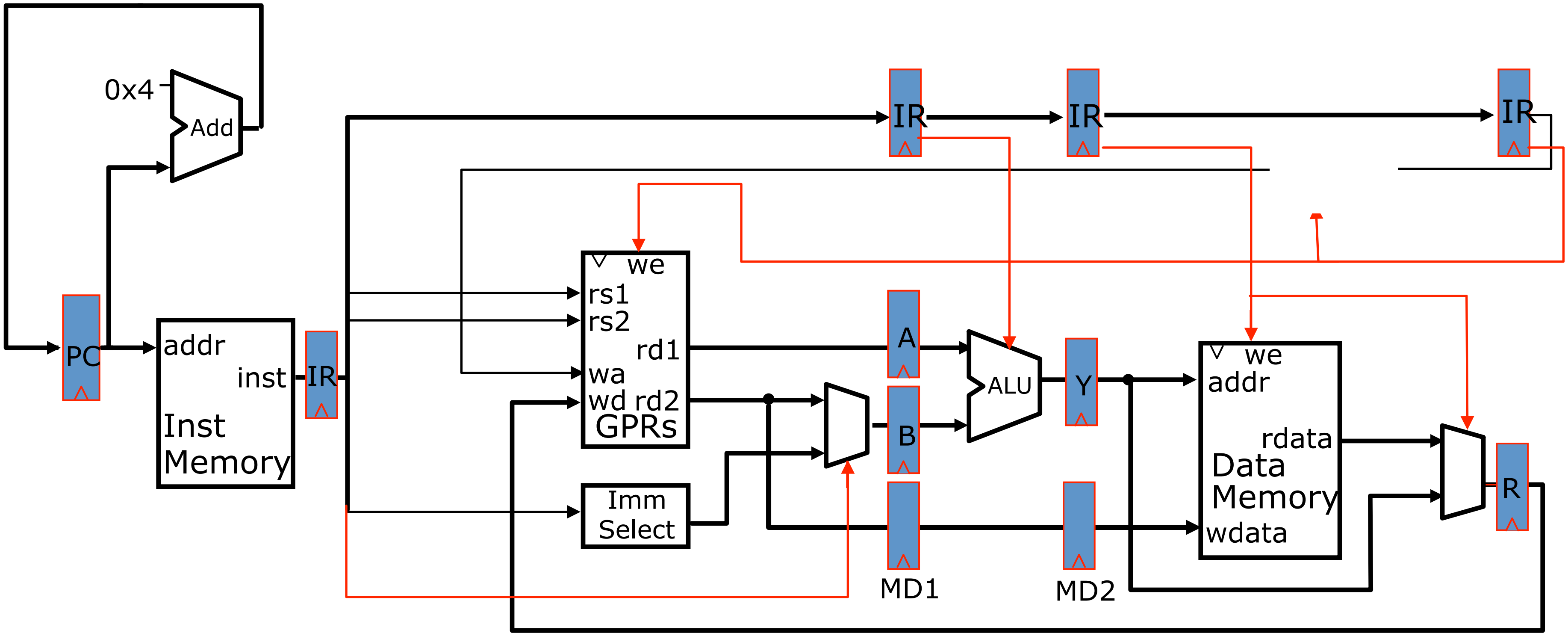
Resource Usage

	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*

What If…

- We used a simple branch that compares only one register (rs1) against zero
- Can we do any better?



Use simpler branches (e.g., only compare one reg against zero) with compare in decode 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	-	-	-	-		
(I_4) 300: 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*

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	300	ADD

Delay slot instruction executed regardless of branch outcome

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) 300: ADD				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	

Post-1990 RISC ISAs don't have delay slots

- Encodes microarchitectural detail into ISA
  - C.f. IBM 650 drum layout

- What are the problems with delay slots?

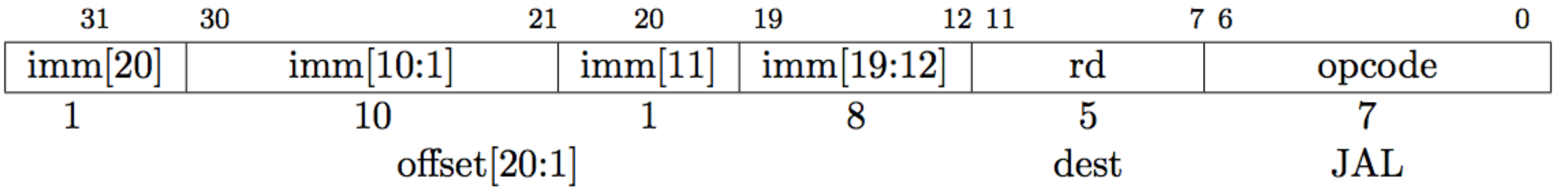
- Performance issues
  - E.g., I-cache miss or page fault on delay slot instruction causes machine to wait, even if delay slot is a NOP
- Complicates more advanced microarchitectures
  - 30-stage pipeline with four-instruction-per-cycle issue
- Complicates the compiler's job
- Better branch prediction reduced need for delay slots

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

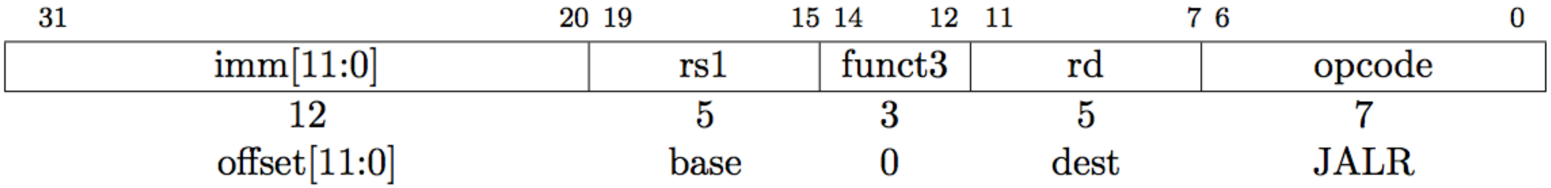
RISC-V Branches and Jumps

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



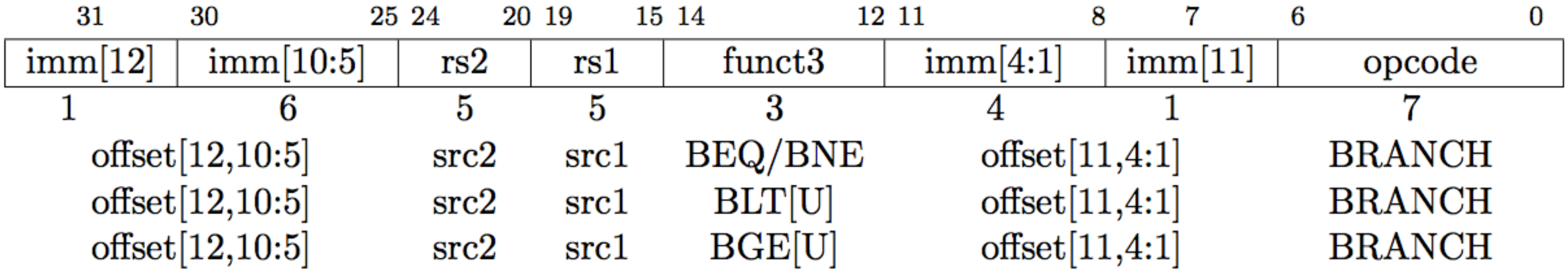
31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

- **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

RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

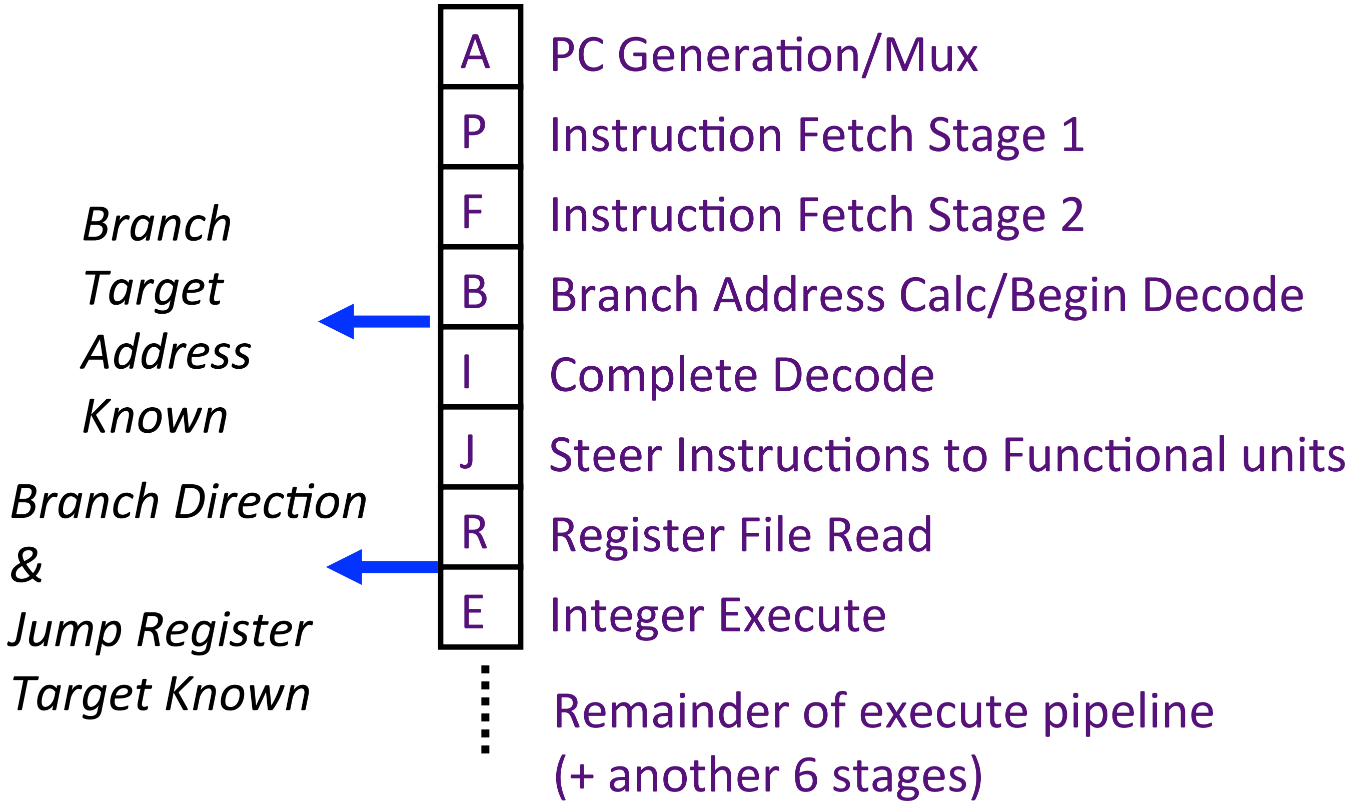
1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

- **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**

Branch Penalties in Modern Pipelines

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)



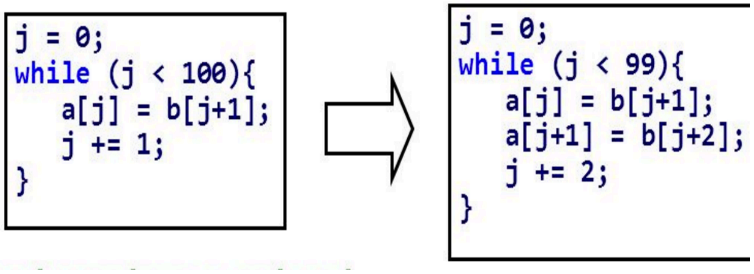
A	PC Generation/Mux
P	Instruction Fetch Stage 1
F	Instruction Fetch Stage 2
B	Branch Address Calc/Begin Decode
I	Complete Decode
J	Steer Instructions to Functional units
R	Register File Read
E	Integer Execute

Remainder of execute pipeline
(+ another 6 stages)

Branch Target Address Known

Branch Direction & Jump Register Target Known

Reducing Control Flow Penalty



```
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;
}
```

- 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)

- 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.

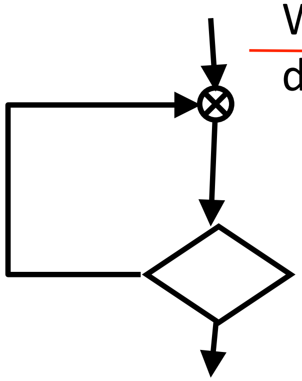
  - *Mispredict recovery mechanisms:*
    - *Keep result computation separate from commit*
    - Kill instructions following branch in pipeline
    - Restore state to that following branch

Static Branch Prediction

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

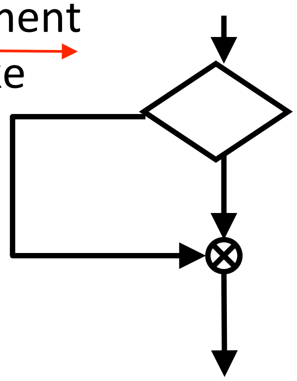
What C++ statement does this look like

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

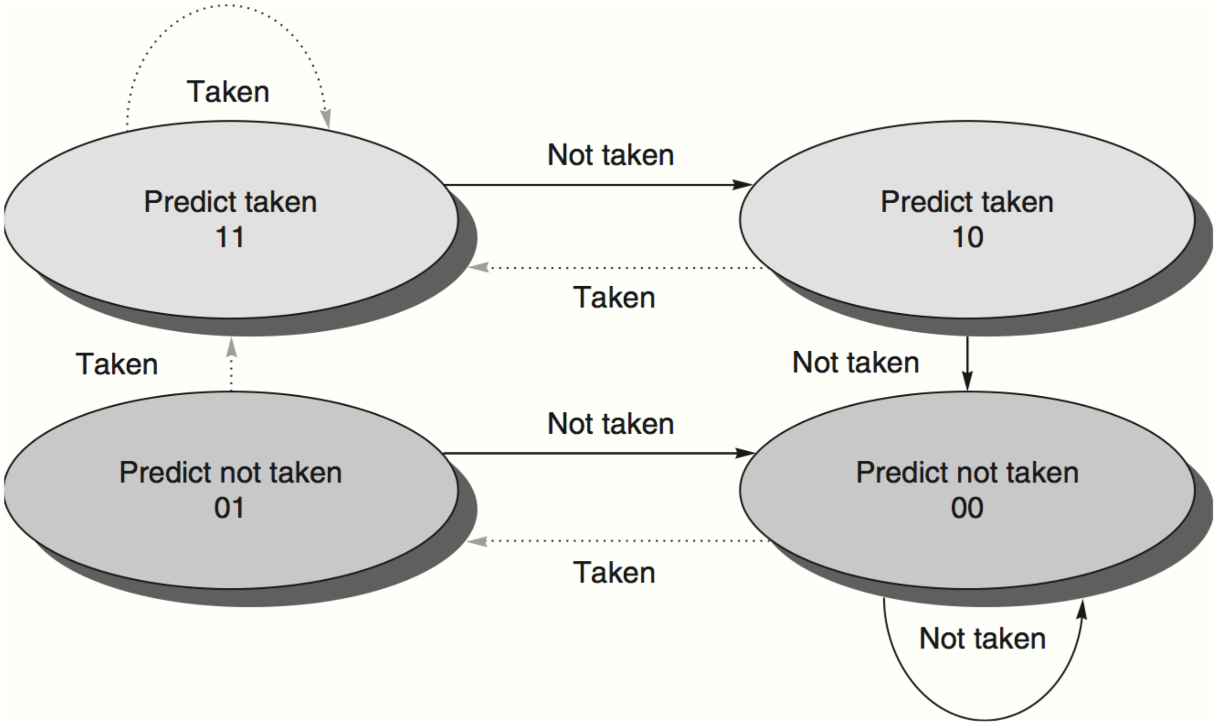
- Spatial correlation (space)
  - Several branches may resolve in a highly correlated manner
  - For instance, a preferred path of execution

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

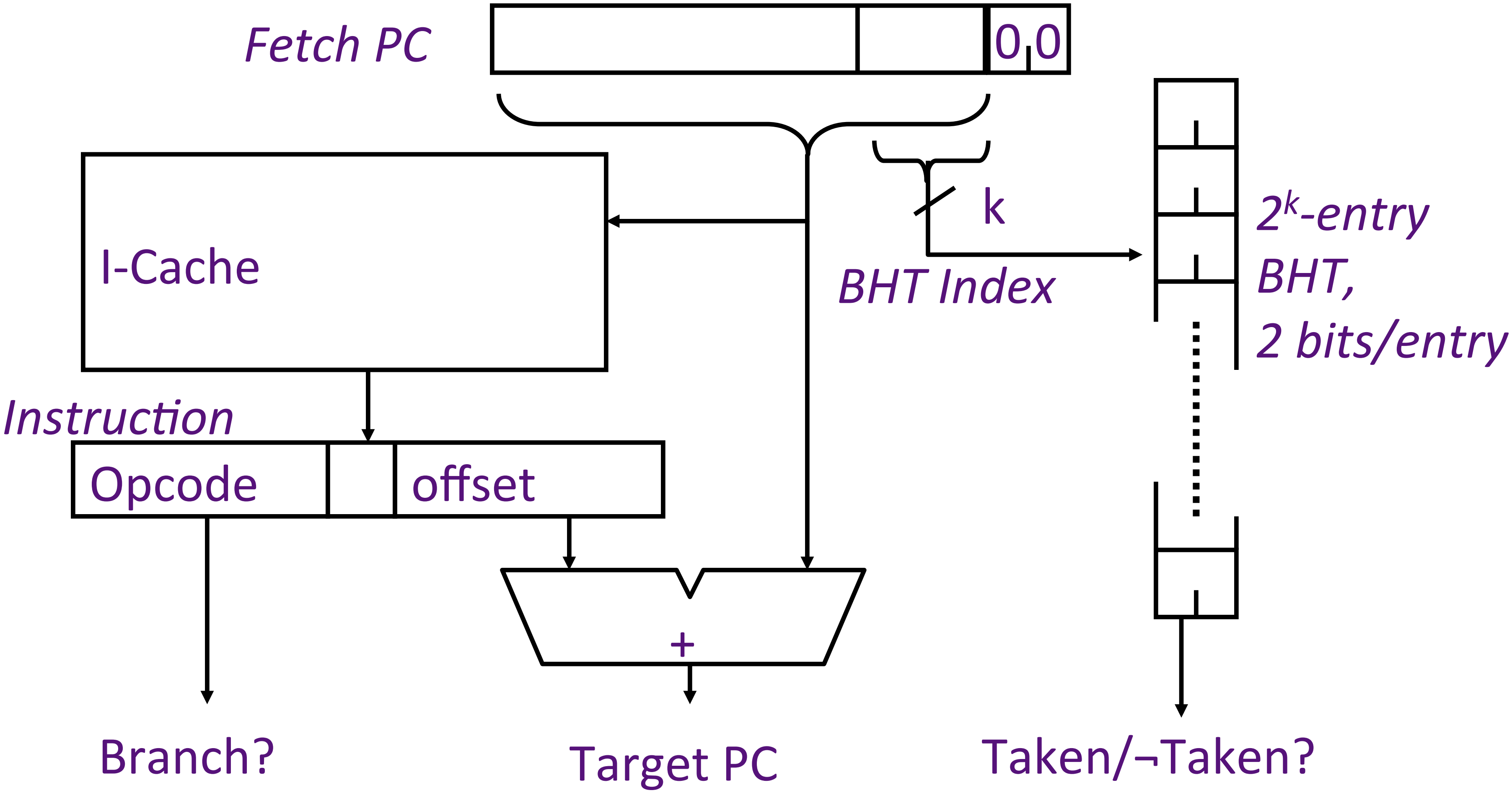
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



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

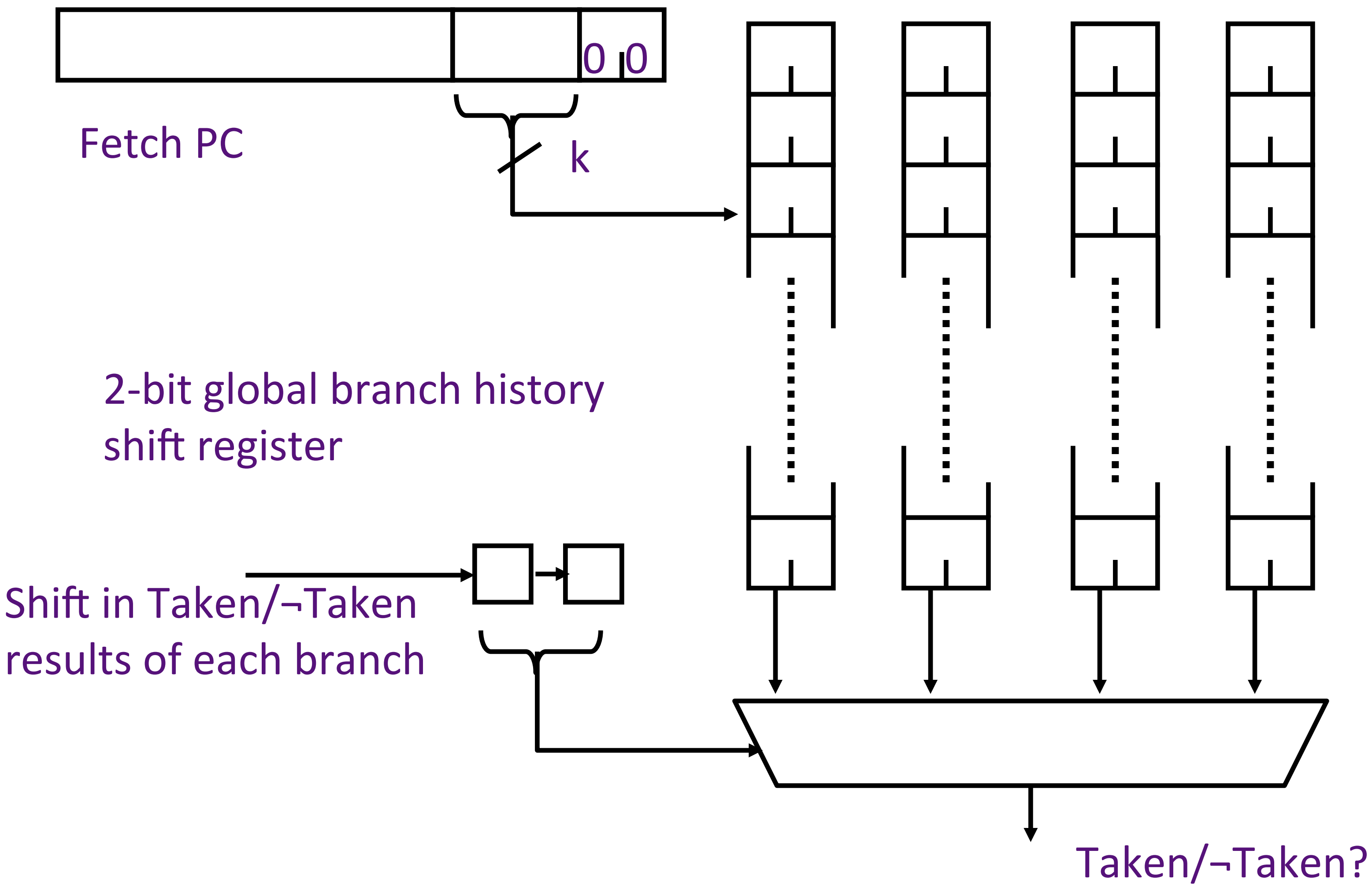
```
if (x[i] < 7) then
   y += 1;
if (x[i] < 5) then
   c -= 4;
```

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)


0 0
Fetch PC
k
2-bit global branch history shift register
Shift in Taken/¬Taken results of each branch
Taken/¬Taken?

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?

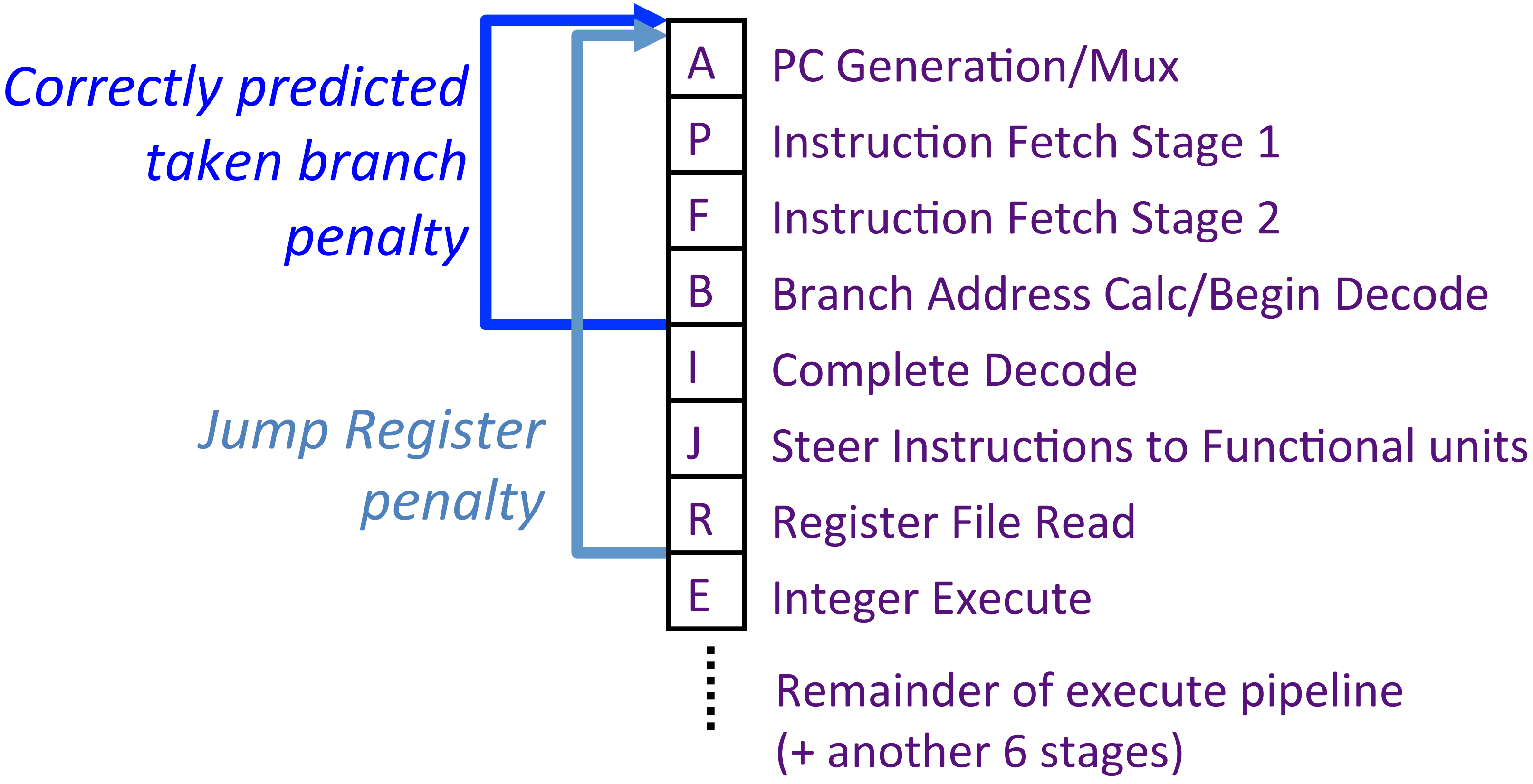
Are We Missing Something?

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

Branch target address

Limitations of BHTs

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.



Correctly predicted taken branch penalty

Jump Register penalty

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

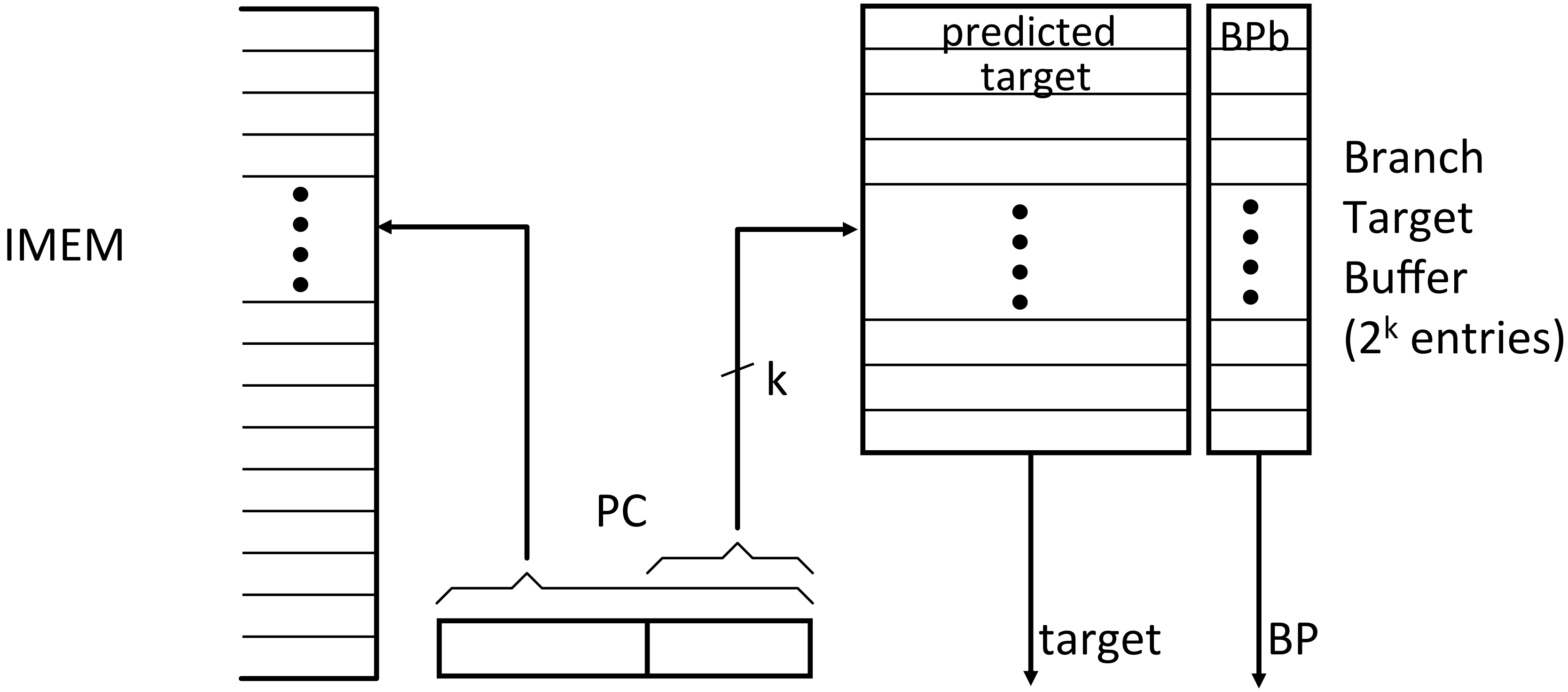
R Register File Read

E Integer Execute

Remainder of execute pipeline
(+ another 6 stages)

UltraSPARC-III fetch pipeline

Branch Target Buffer

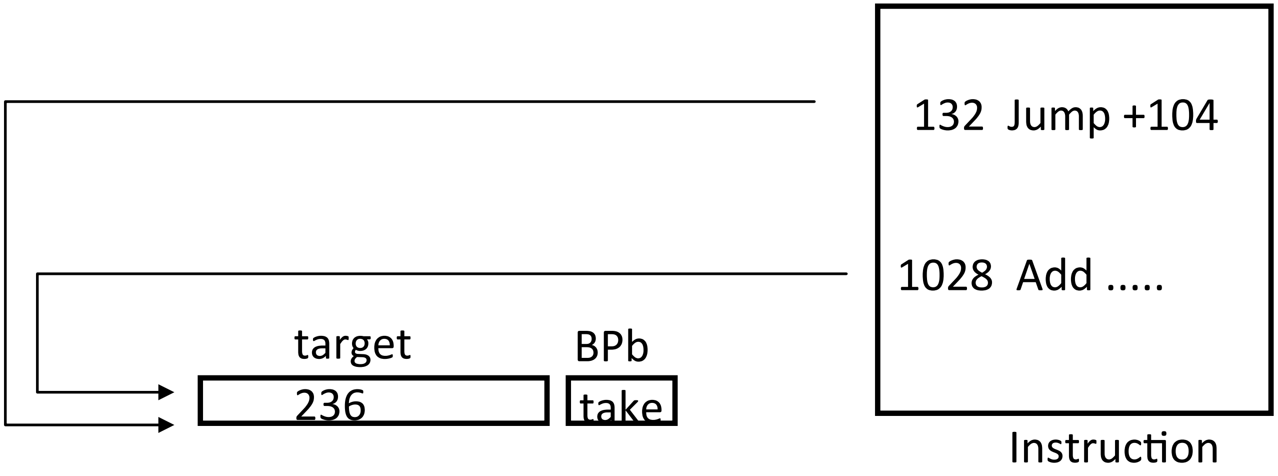


BP bits are stored with the predicted target address.

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 (Mis-Prediction)

Assume a 128-entry BTB



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

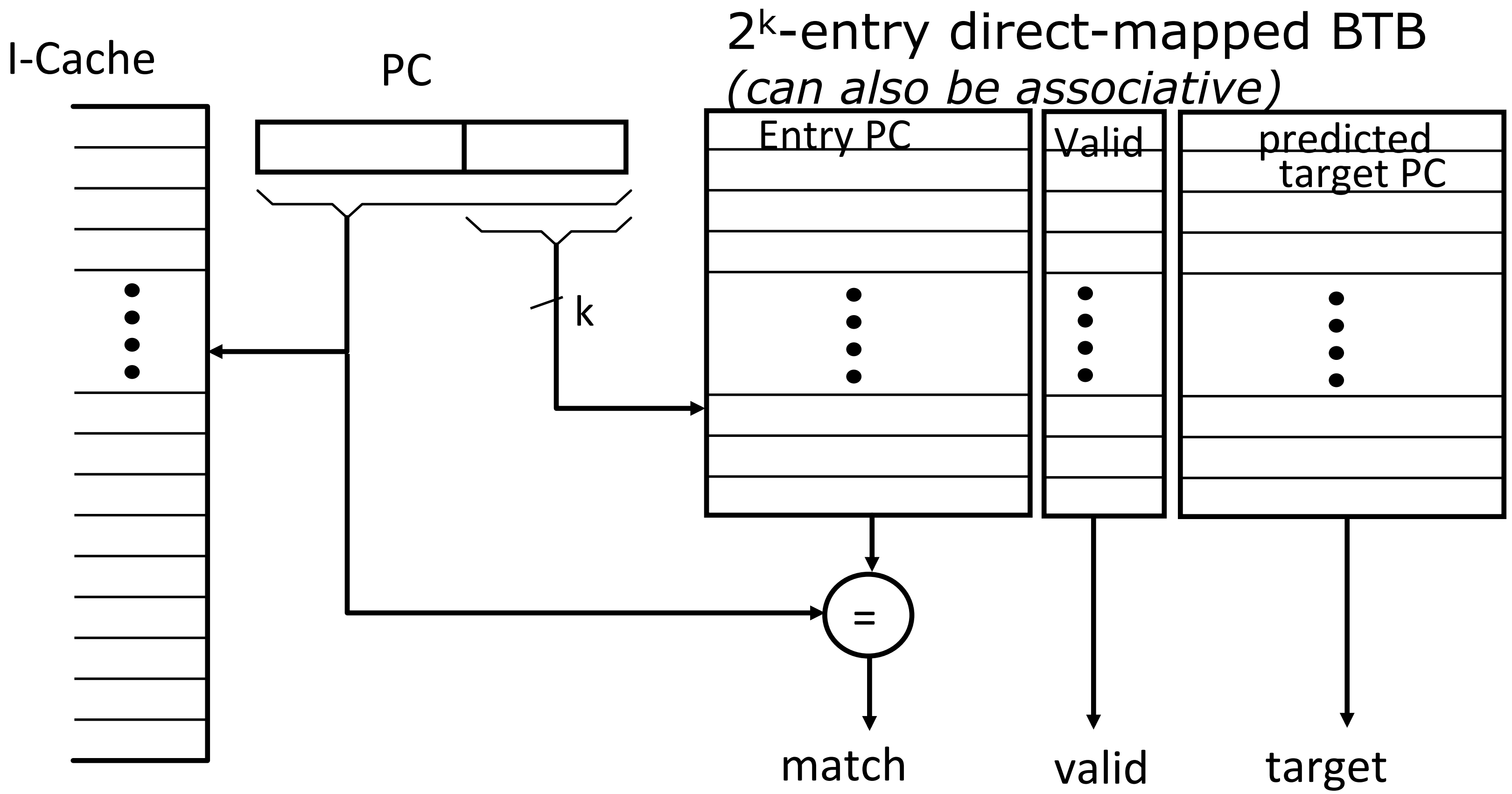
- Is even branch prediction fast enough to avoid bubbles?
- When do we index the BTB?
  - i.e., what state is the branch in, in order to avoid bubbles?

- **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?*

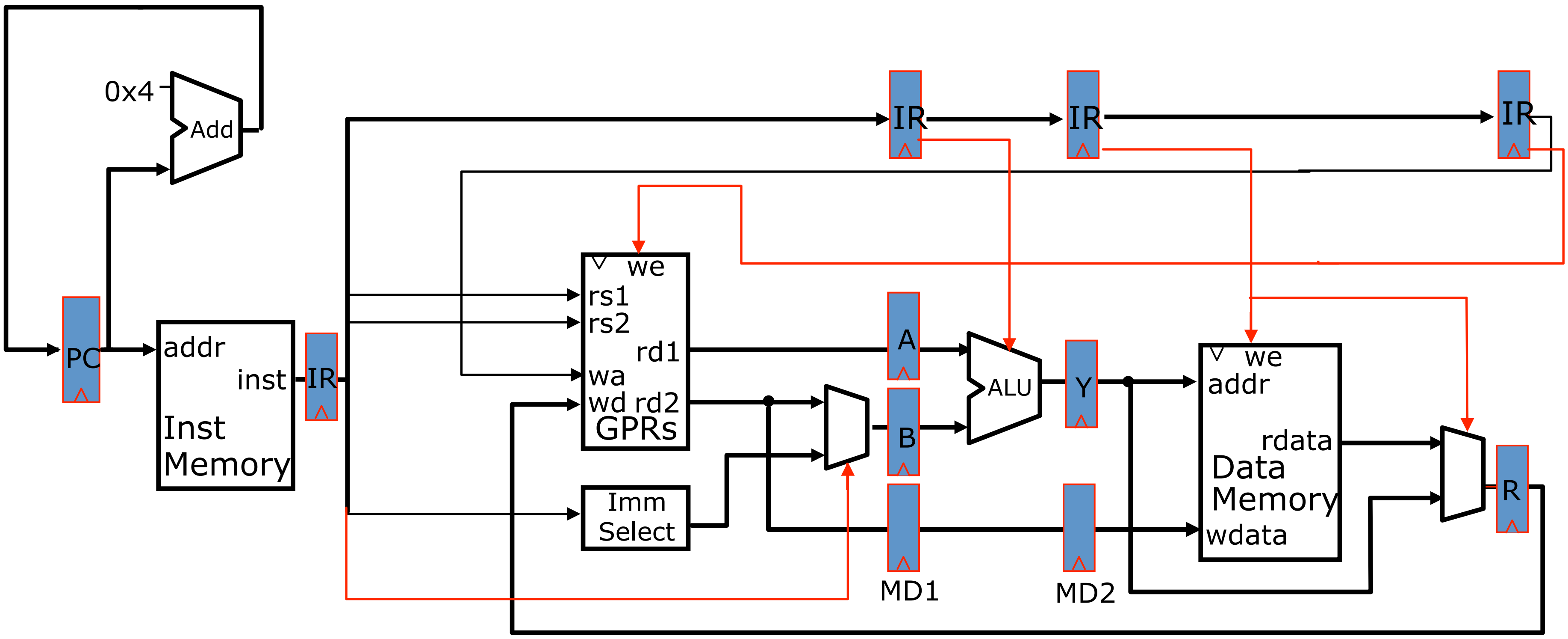
Branch Target Buffer (BTB)



- 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

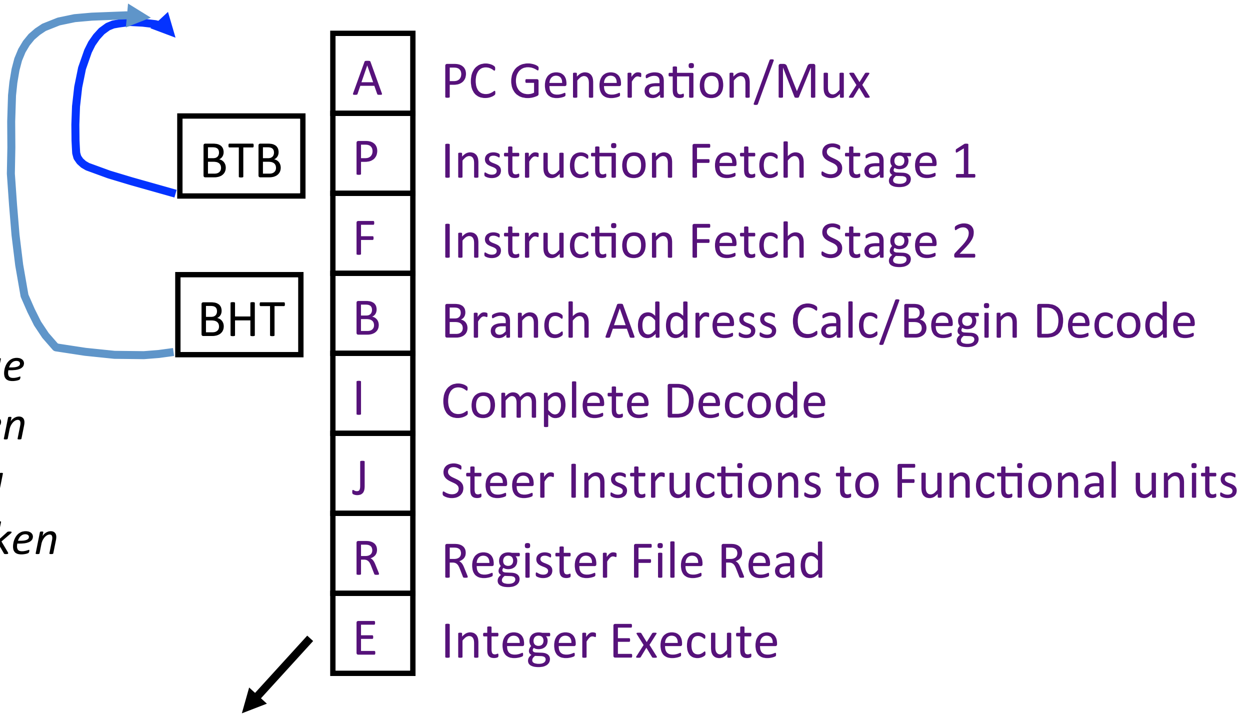
Are We Missing Something? (2)

- When do we update the BTB or BHT?



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

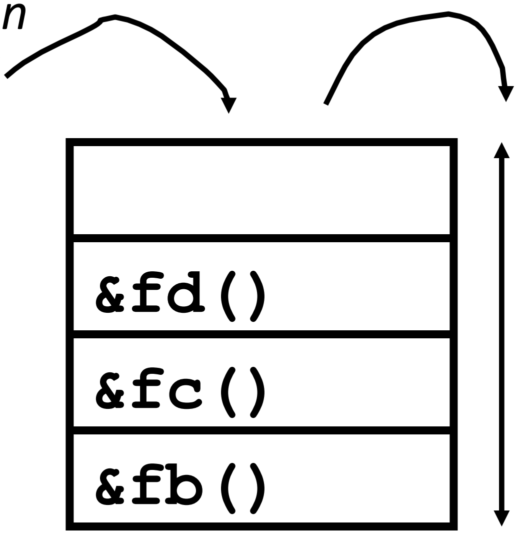
  ⇒ *Often one function called from many distinct call sites!*

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)