Computer Architecture

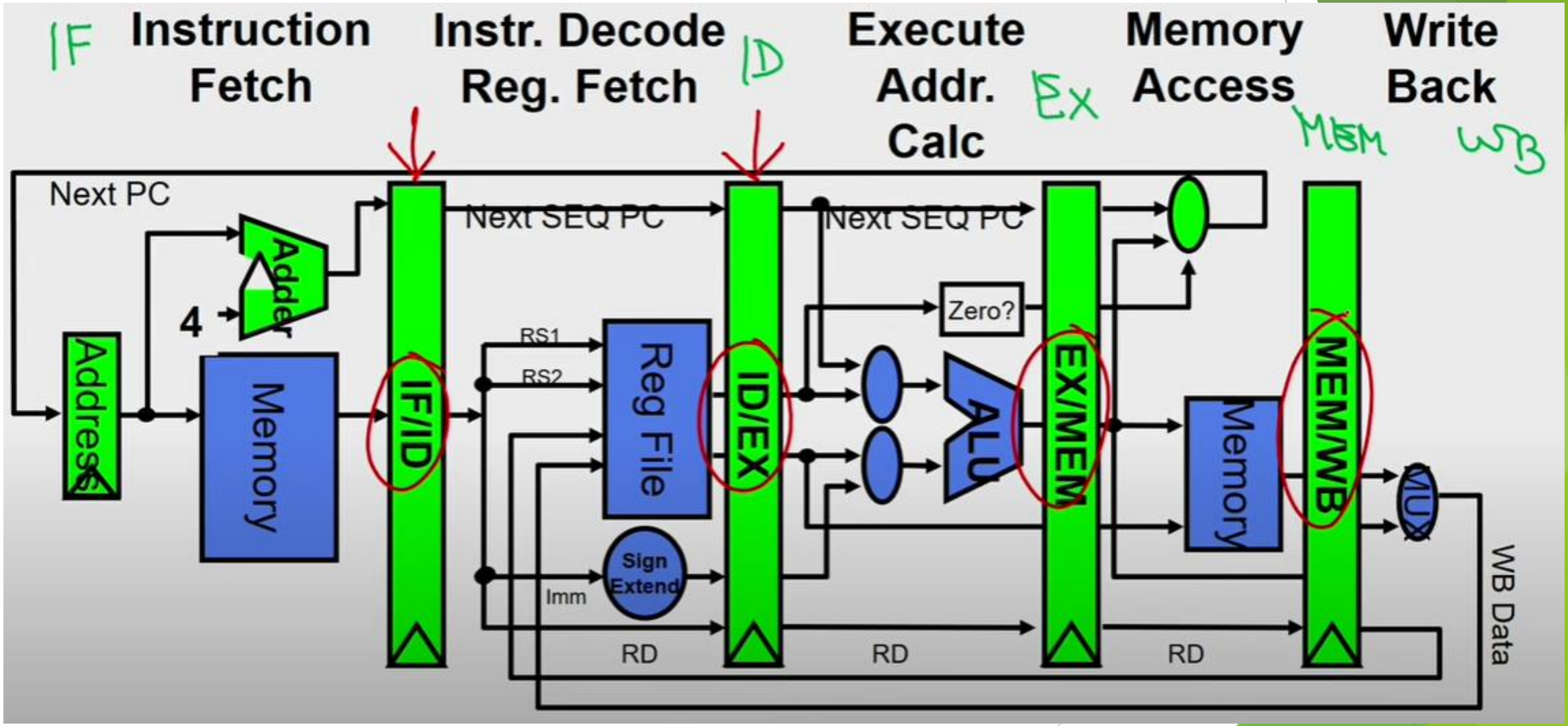
Control Hazards and Branch Prediction

Dr. Mohammad Reza Selim

Lecture Outline

- Data Hazard Review
- Problem Solving (Data Hazards)
- Control/Branch Hazards
- Four Branch Hazard Alternatives
- Dynamic Branch Prediction
- Problem Solving (Branch Prediction)

Pipelined RISC Datapath



Instruction Execution Cycle

- Each instruction can take at most 5 clock cycles
- **Instruction fetch cycle (IF)**
- **Instruction decode/register fetch cycle (ID)**
- **Execution/Effective address cycle (EX)**
- **Memory access cycle (MEM)**
- **Write-back cycle (WB)**

Visualizing the Pipeline

	Clock number							
Instruction number	1	2	3	4	5	6	7	8
i	IF	ID	EX	MEM	WB			
$i+1$		IF	ID	EX	MEM	WB		
$i+2$			IF	ID	EX	MEM	WB	
$i+3$				IF	ID	EX	MEM	WB
$i+4$					IF	ID	EX	MEM

Pipeline Hazards

- **Hazards**: circumstances that would cause incorrect execution if next instruction is fetched and executed
  - **Structural hazards**: Different instructions, at different stages, in the pipeline want to use the same hardware resource
  - **Data hazards**: An instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline
  - **Control hazards**: Succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline

Three Types of Data Hazards (1)

- **Read After Write (RAW)**

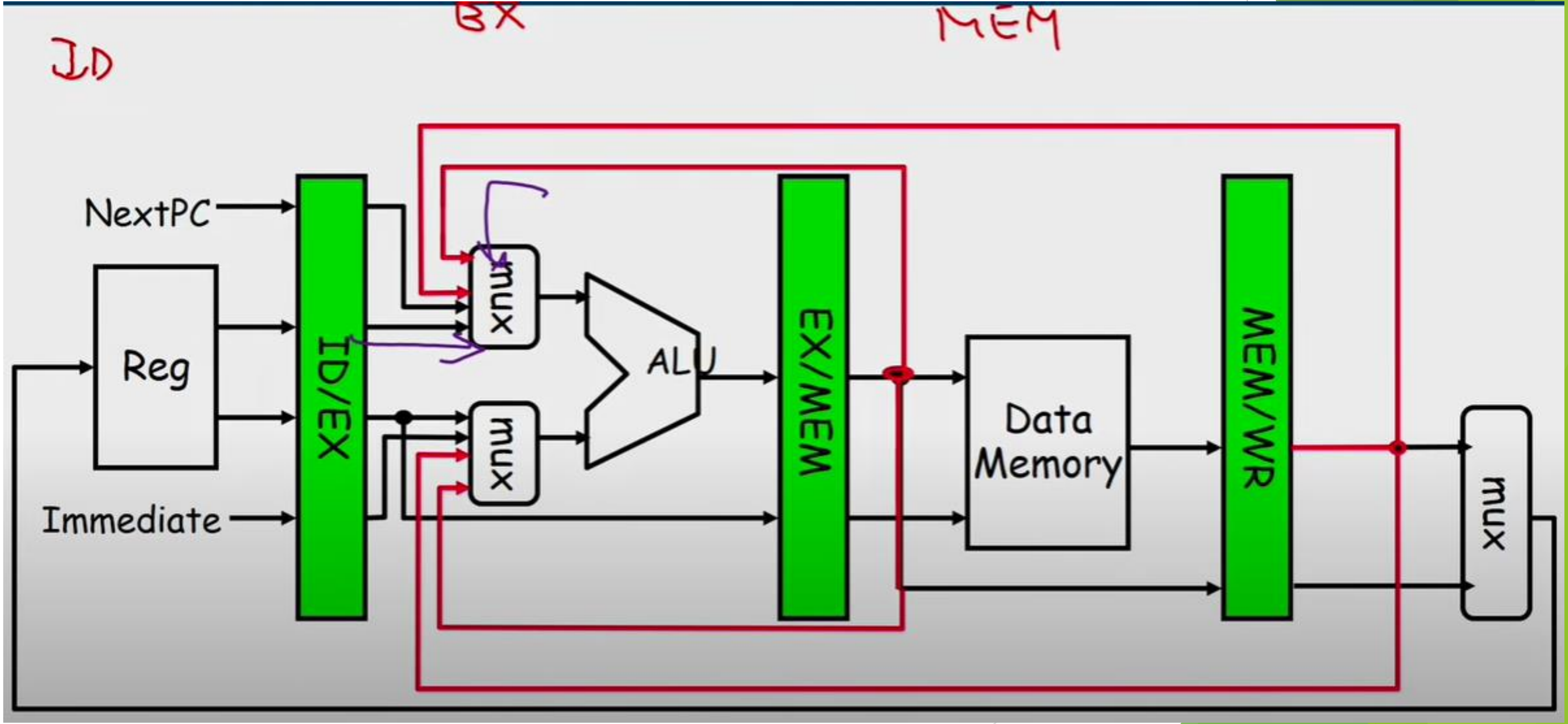
  Instr_J tries to read operand before Instr_I writes it

  I : add r1,r2,r3

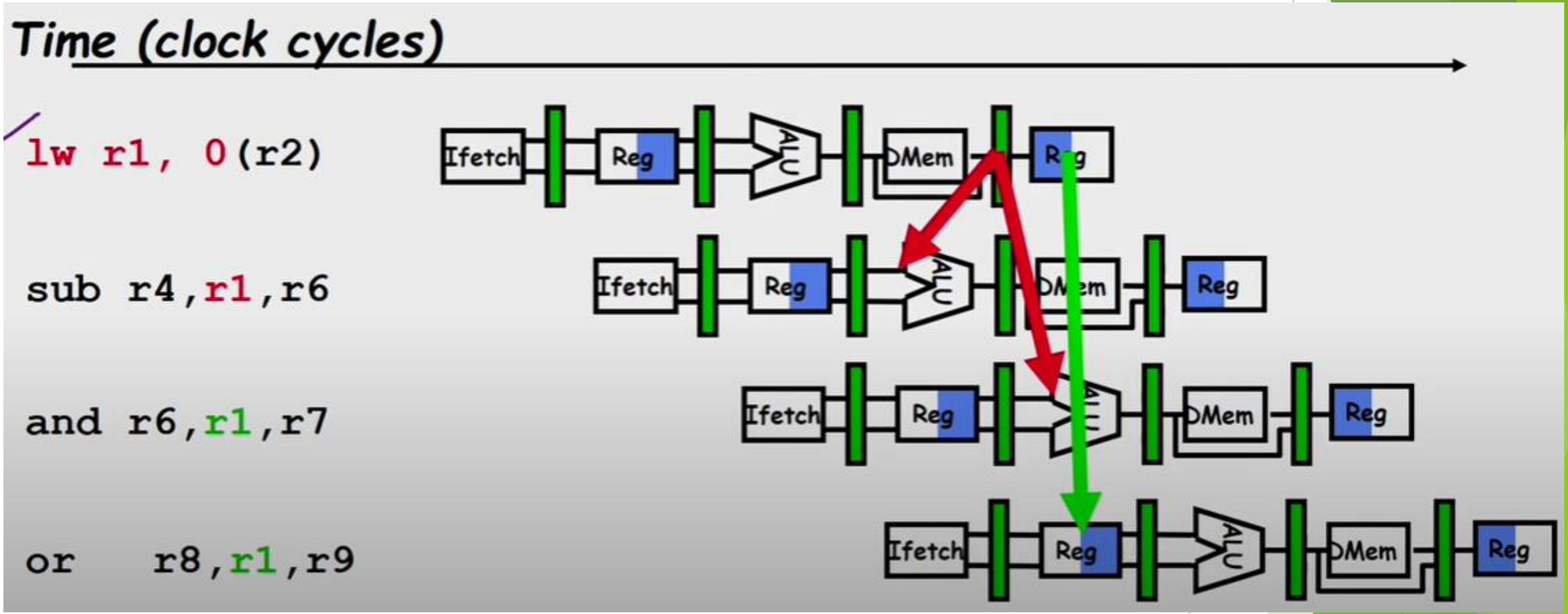
  J: sub r4,r1,r3

- Caused by a data dependence
- This hazard results from an actual need for communication.

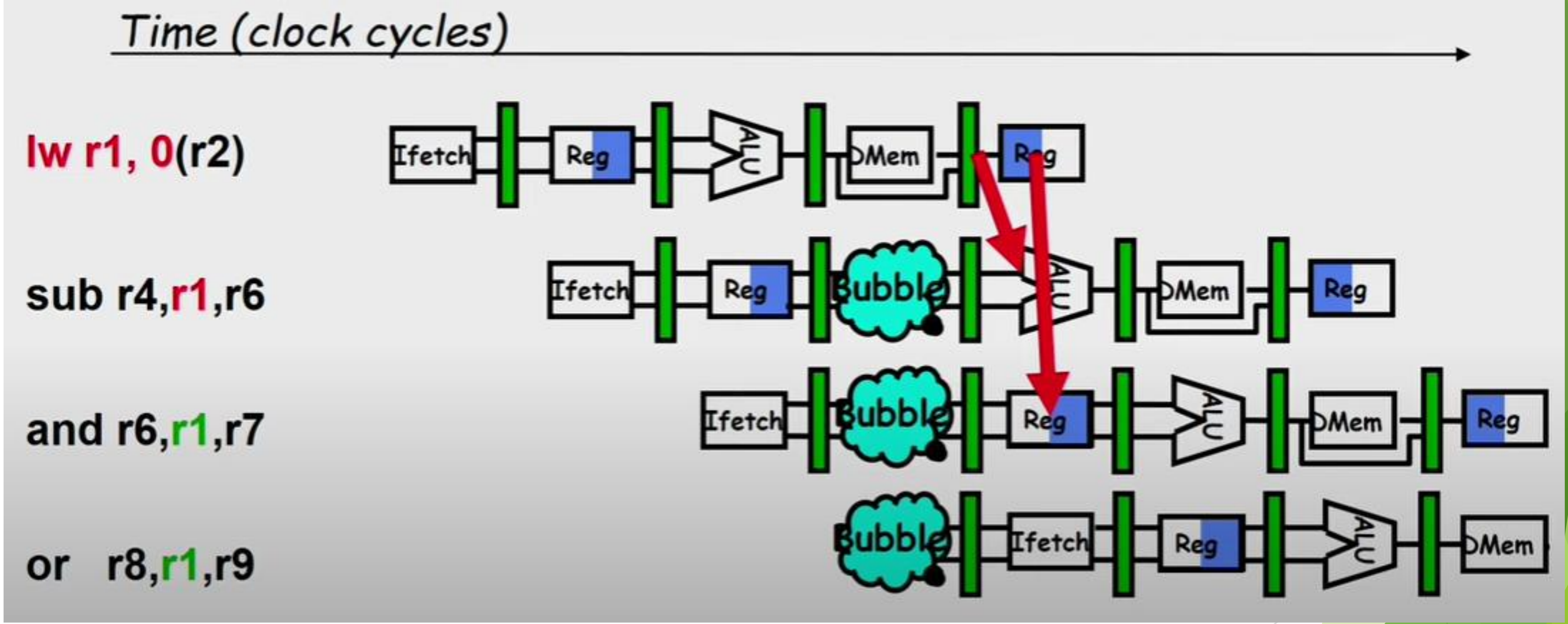
Hardware Change for Operand Forwarding



Data Hazards even with Operand Forwarding



Resolving the Load-ALU Hazard



Problem Solving: Pipeline Hazards(1)

Given a non-pipelined architecture running at 1.5 GHz, that takes 5 cycles to finish an instruction. You want to make it pipelined with 5 stages. Due to hardware overhead the pipelined design will operate only at 1 GHz. 5% of memory instructions cause a stall of 50 cycles, 30% of branch instruction cause a stall of 2 cycles and load-ALU combinations cause a stall of 1 cycle. Assume that in a given program, there exist 20% of branch instructions and 30% of memory instructions. 10% of instructions are load-ALU combinations. What is the speedup of pipelined design over the non-pipelined design?

(a) CPI_up = 5 1.5Ghz→0.67 ns : 1 Ghz→1 ns

Ex_T up= CPI x CCT = 5 x 0.66 ns = 3.33 ns / instruction

(b) Effective CPIp = Base CPI + stall CPI

{stalls= Memory stalls+ Branch stalls+ Load –ALU stalls}

= 1 + (0.3x0.05x50) + (0.2x0.3x2) + (0.1x1) = 1 + 0.75 + 0.12 +0.1 = 1.97

Ex_p = CPI x CCT = 1.97 x 1 ns = 1.97 ns / instruction

Speedup = Ex_up/ Ex_p = 3.33/1.97 = 1.69

Problem Solving: Pipeline Hazards(2)

A program has 2000 instructions in the sequence L.D, ADD.D, L.D, ADD.D,..... L.D, ADD.D. The ADD.D instruction depends on the L.D instruction right before it. The L.D instruction depends on the ADD.D instruction right before it. If the program is executed on the 5-stage pipeline what would be the actual CPI with and without operand forwarding technique?

Without operand forwarding.

ID of nth instruction can be only after WB of n-1th instruction.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
L.D	IF	ID	EX	ME	WB									
ADD		IF	*	*	*	ID	EX	ME	WB					
L.D						IF	*	*	*	ID	EX	ME	WB	
ADD										IF	*	*	*	ID

Instructions reach WB at clock cycles 5, 9, 13, 17, 21, 25, 29,.....

Last instruction (ADD) reaches WB in 5 + (1999x4) = 8001 cycles.

CPI= 8001/2000=4.0005

$\frac{8001}{2000} \approx 4$

Problem Solving: Pipeline Hazards(3)

With operand forwarding.

Every ADD after L.D has a stall,

but L.D after ADD do not have a stall.

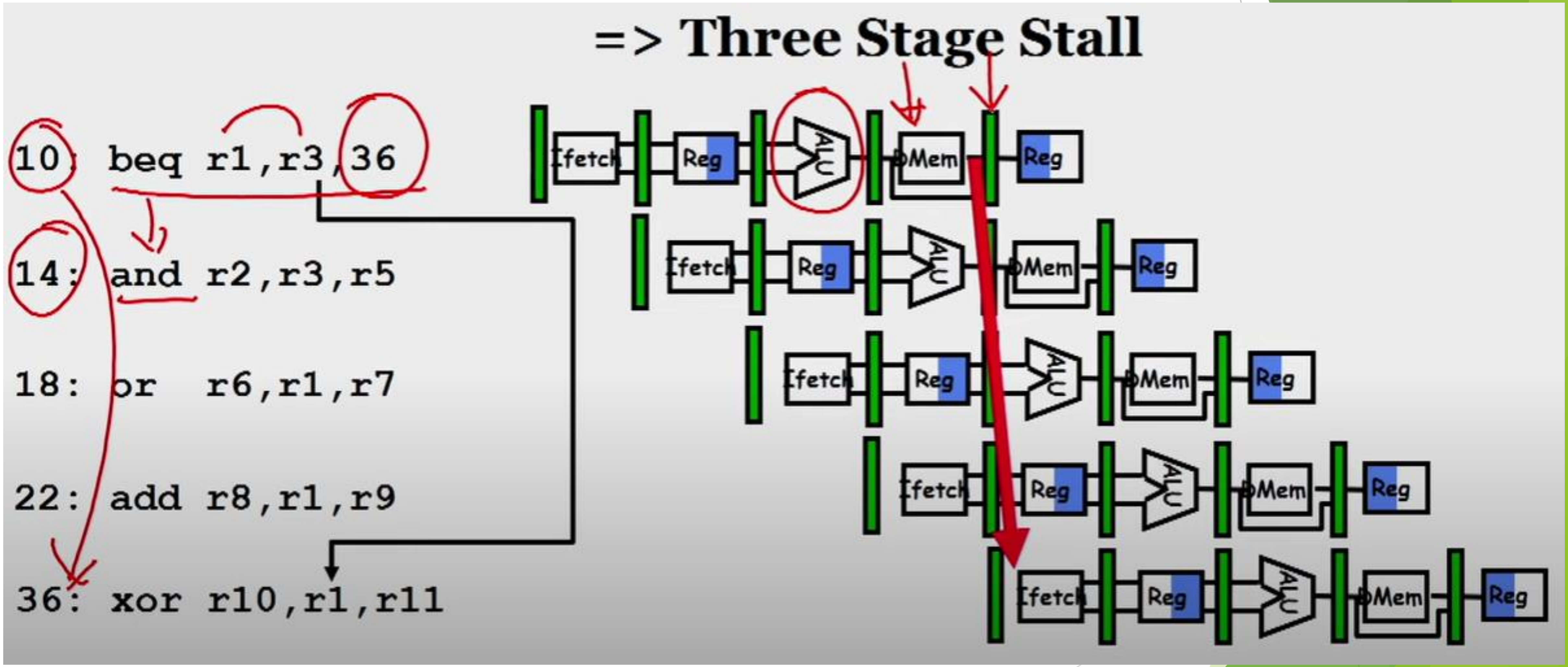
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
L.D	IF	ID	EX	ME	WB									
ADD		IF	*	ID	EX	ME	WB							
L.D				IF	ID	EX	ME	WB						
ADD					IF	*	ID	EX	ME	WB				

Instructions reach WB at clock cycles 5,7, 8,10, 11,13, 14,16

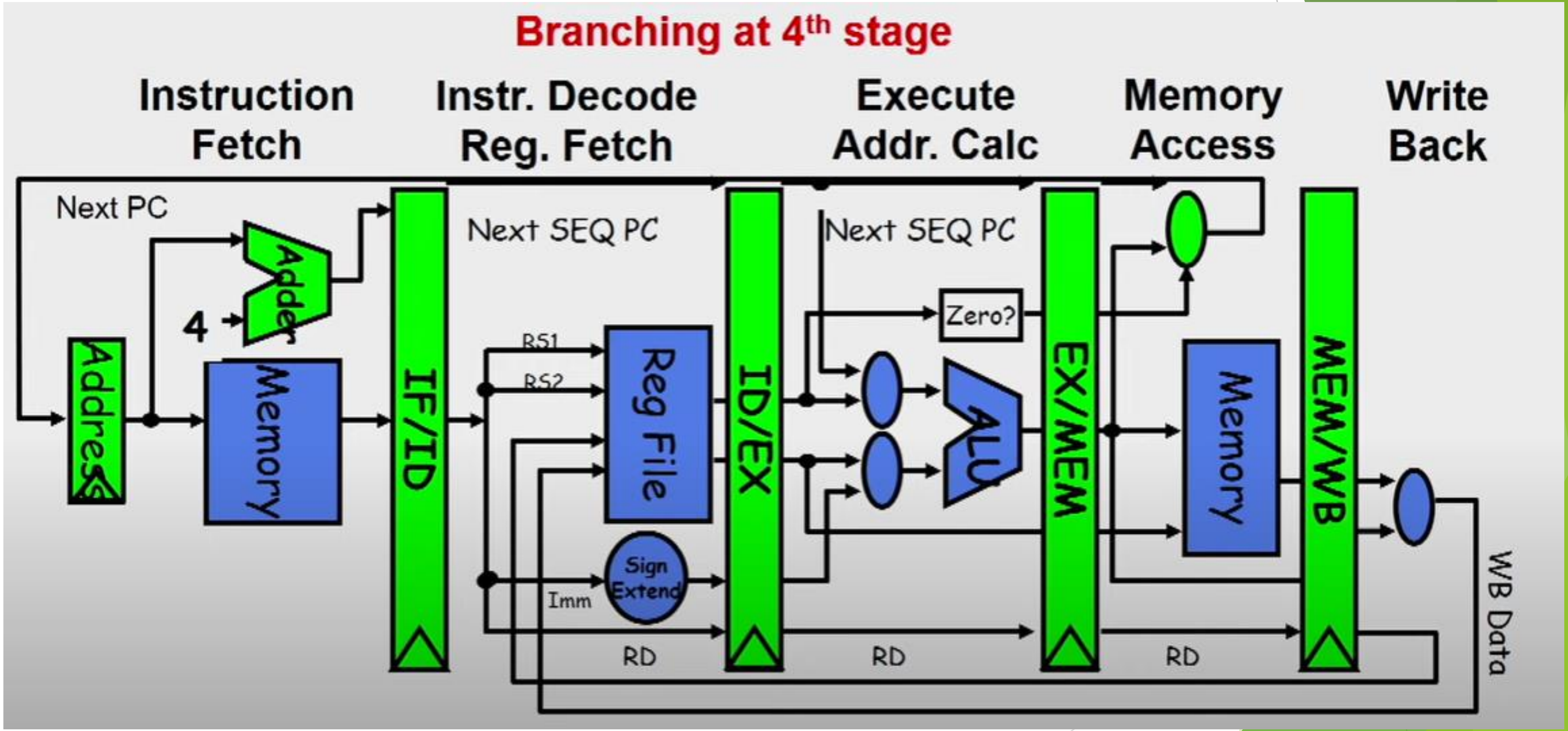
Last instruction (ADD) reaches WB in 7 + (999x3) = 3004 cycles.

CPI= 3004/2000=1.502

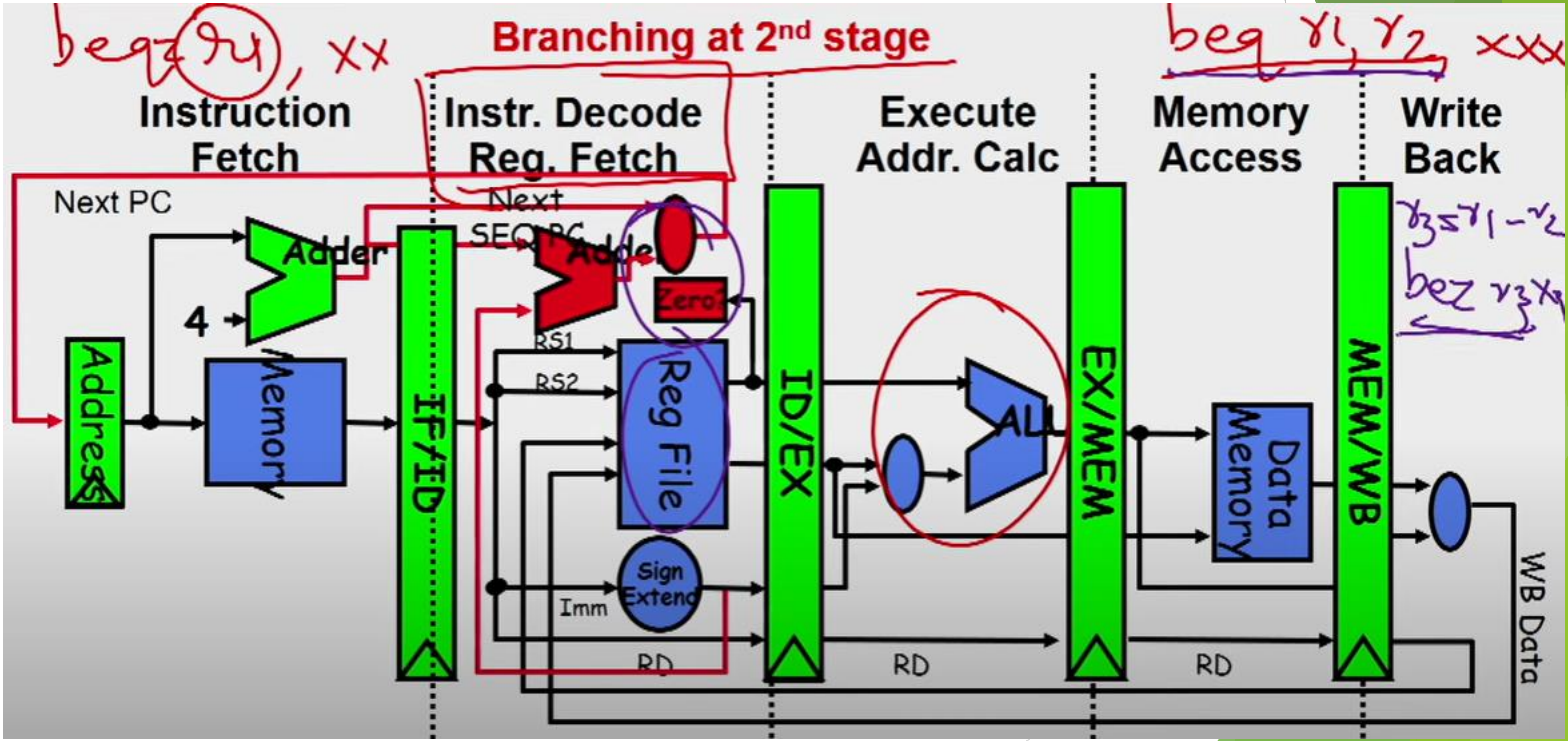
Control Hazards on Branches



Conventional MIPS Pipeline



Branch Optimized MIPS Pipeline



Four Branch Hazards Alternatives (1)

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

#3: Predict Branch Taken

#4: Delayed Branch

Four Branch Hazards Alternatives (2)

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Four Branch Hazards Alternatives (3)

#3: Predict Branch Taken

- But branch target address in is not known by IF stage
- Target is known at same time as branch outcome (IDstage)
- MIPS still incurs 1 cycle branch penalty

Four Branch Hazards Alternatives (4)

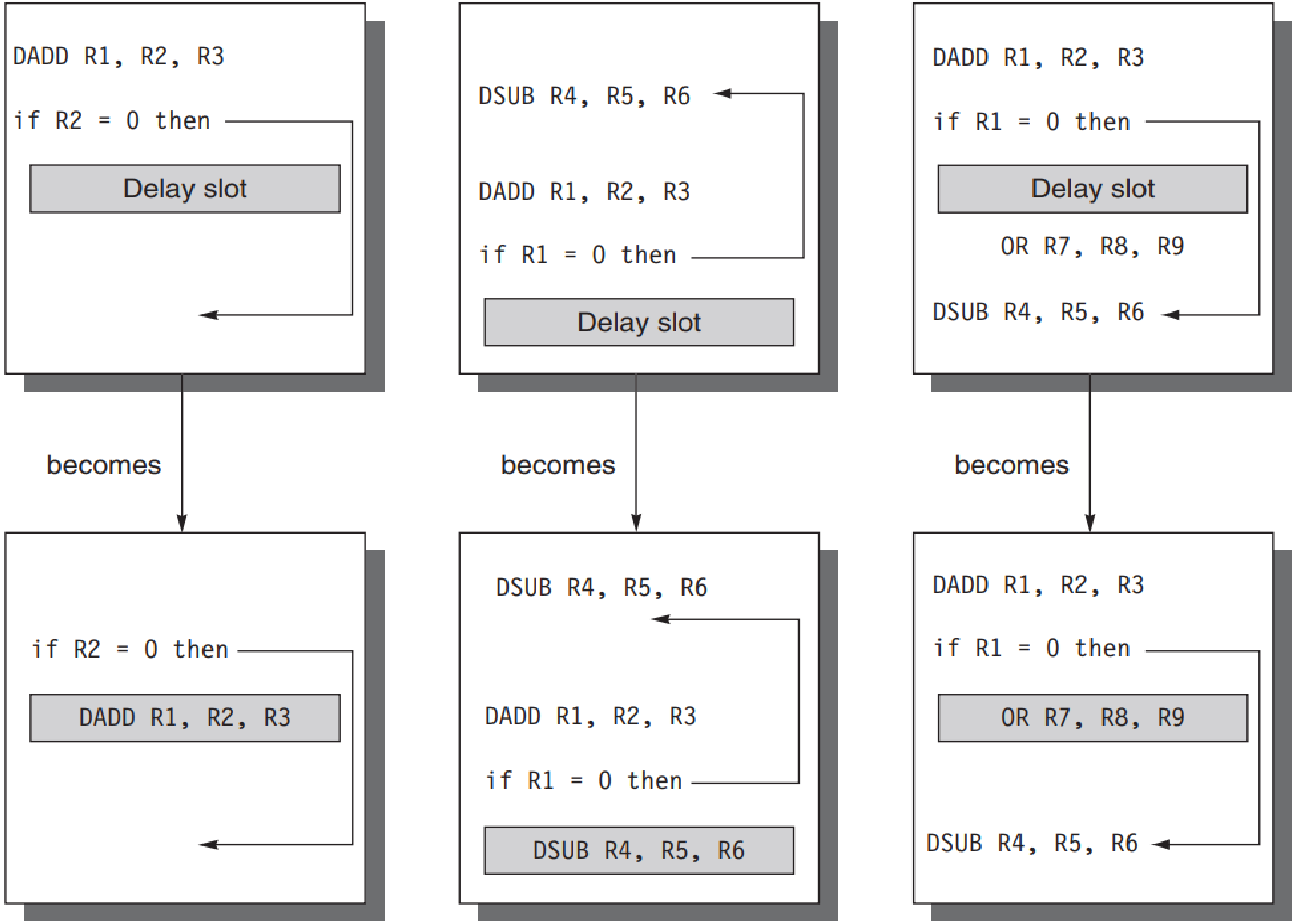
#4: Delayed Branch

- Define branch to take place AFTER one instruction following the branch instruction.
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline (MIPS uses this approach)
- **Where to get instructions to fill branch delay slot?**

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Filling Branch Delay Slots



(a) From before (b) From target (c) From fall-through

Execution of Conditional Branches Instructions

- When do you know you have a branch?
  - During ID cycle (Could you know before that?)
- When do you know if the branch is Taken or Not-Taken
  - During EXE cycle/ ID stage depending on the design
- We need for sophisticated solutions for following cases
  - Modern pipelines are deep (10 + stages)
  - Several instructions issued/cycle
  - Several predicted branches in-flight at the same time

Dynamic Branch Prediction (1)

- Execution of a branch requires knowledge of:
  - Branch instruction - encode whether instruction is a branch or not. Decide on taken or not taken (i.e., prediction can be done at IF stage)

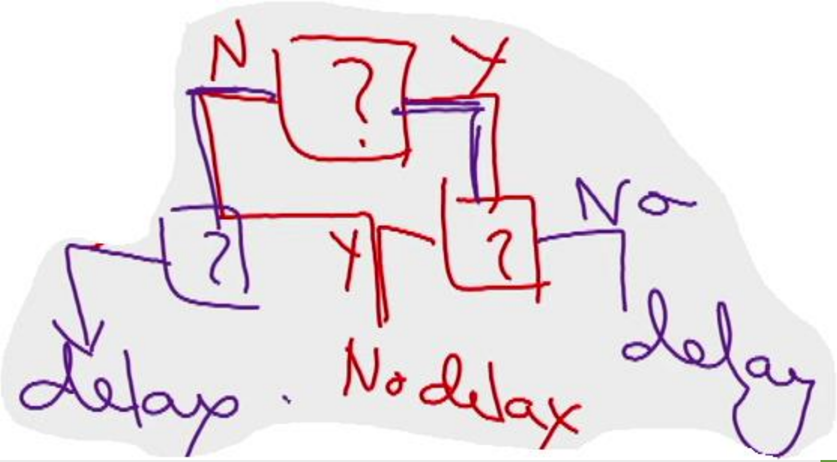
  - Whether the branch is Taken/Not-Taken
  - If the branch is taken what is the target address (can be computed but can also be "precomputed", i.e., stored in some table)
  - If the branch is taken what is the instruction at the branch target address (saves the fetch cycle for that instruction)

Dynamic Branch Prediction (2)

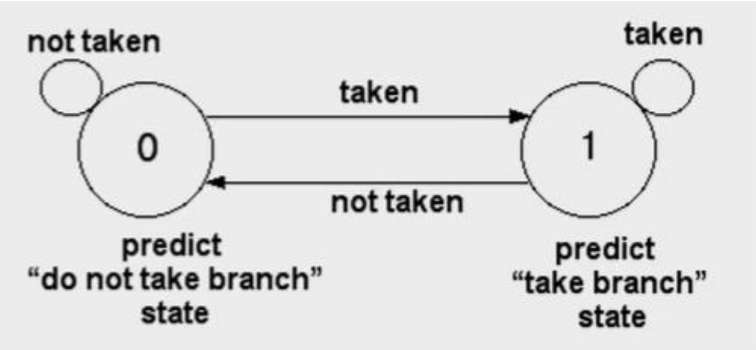
- Use a Branch Prediction Buffer (BPB)
  - Also called Branch Prediction Table (BPT), Branch History Table (BHT)
  - Records previous outcomes of the branch instruction
  - How to index into the table is an issue.
- A prediction using BPB is attempted when the branch instruction is fetched IF stage or equivalent)
- It is acted upon during ID stage (when we know we have a branch)

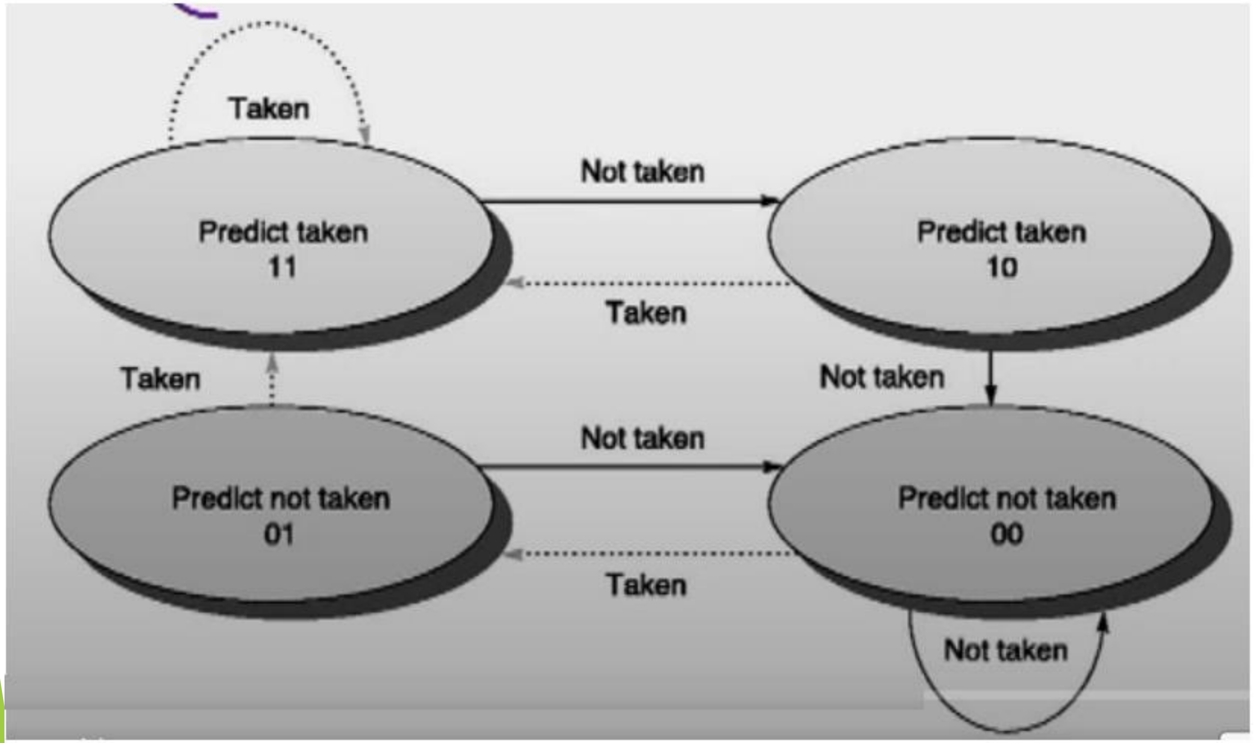
Dynamic Branch Prediction (3)

- Has a prediction been made (Y/N)
  - If not use default "Not Taken"
- Is it correct or incorrect ?
- Two cases:
  - Case 1: Yes and the prediction was correct (known at ID stage) or No but the default was correct: No delay
  - Case 2: Yes and the prediction was incorrect or No and the default was incorrect: Delay



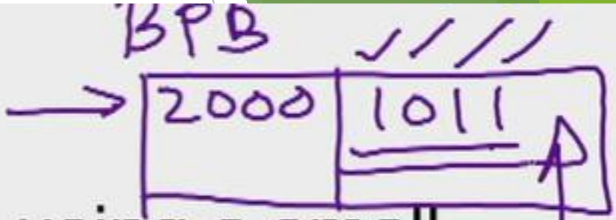
Prediction using 1/2 bit FSM





- The use of a 2-bit predictor will allow branches that favor taken (or not taken) to be mispredicted less often than the one-bit case. (reinforcement learning)

Branch Prediction in Hardware



- Branch prediction is extremely useful in loops.
- A simple branch prediction can be implemented using a small amount of memory indexed by lower order bits of the address of the branch instruction. **(branch prediction buffer)**

- One bit stores whether the branch was taken or not.
- The next time the branch instruction is fetched refer this bit

Advanced Dynamic Branch Prediction (1)

- **Basic 2-bit predictor:**
  - For each branch:- Predict T or NT
  - If the prediction is wrong for two consecutive times, change prediction
- **Correlating predictor:**
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches

```
if ( x = = 2)      /* br-1*/
    x = 0;
if ( y = = 2)      /* br-2*/
    y = 0;
if ( x != y)       /* br-3/
    do this
else do that
```

Advanced Dynamic Branch Prediction (2)

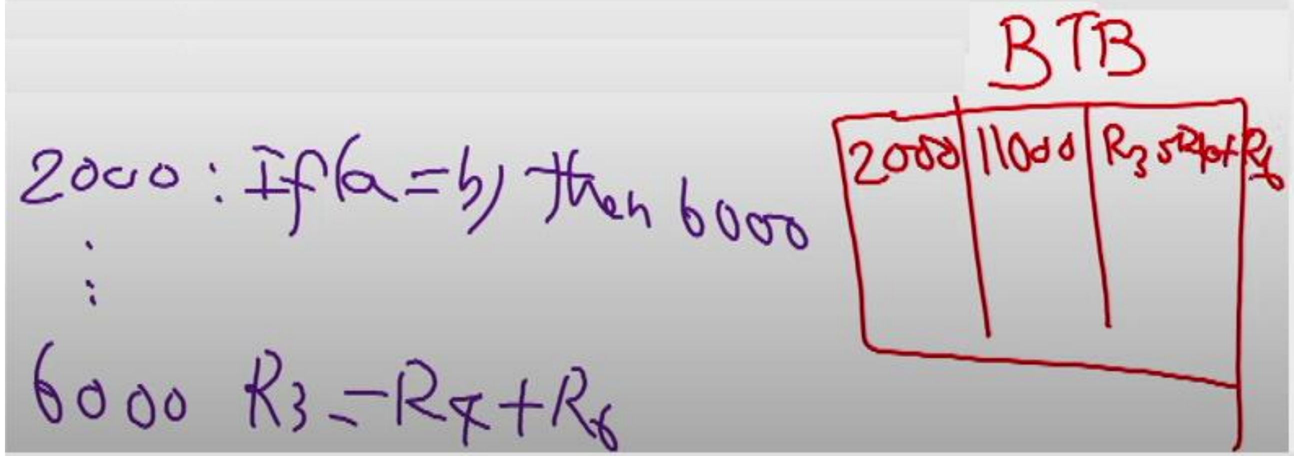
If(a=b) ✓

- **Local predictor:**
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- **Tournament predictor:**
  - Combine correlating predictor with local predictor

Branch Target Buffer (1)

- To reduce the branch penalty, know whether the as-yet-un-decoded instruction is a branch. If so, what the next program counter (PC) should be.

- If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero.

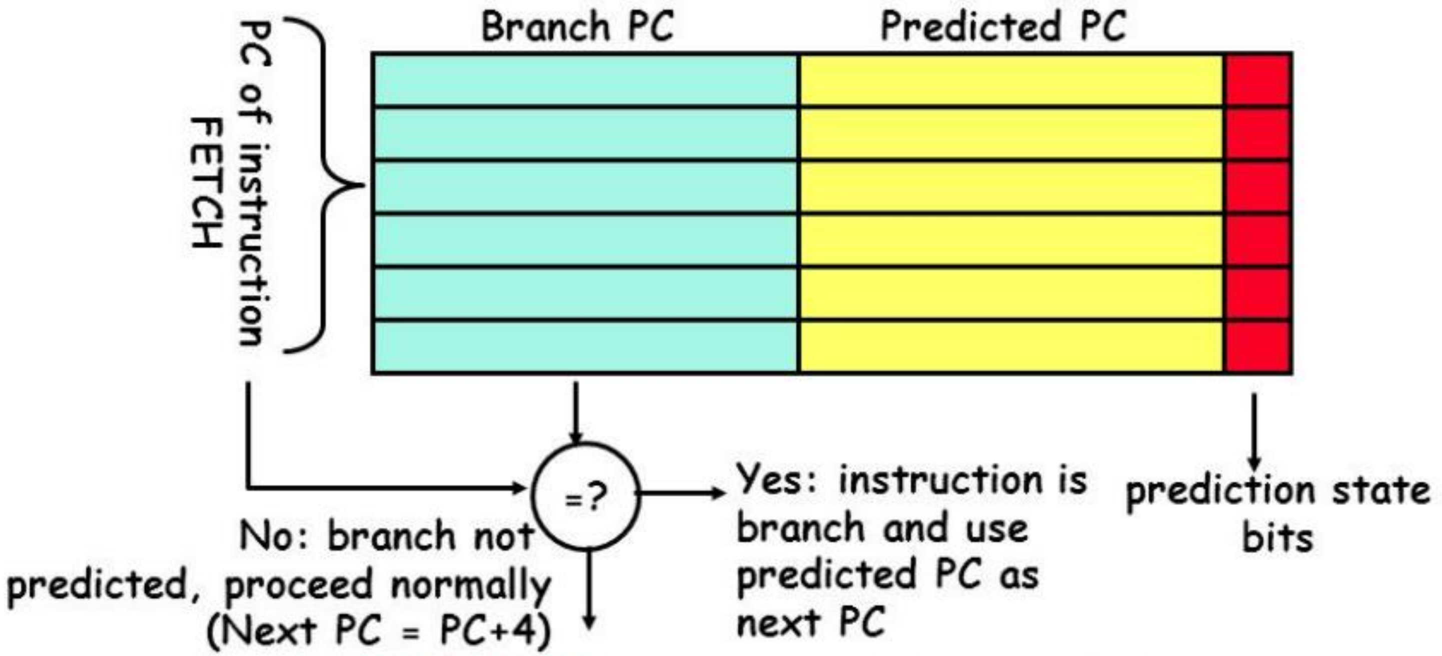


- A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a **branch-target buffer (BTB)** or **branch-target cache.**

Branch Target Buffer (2)

BTB: Branch Address at Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)



Only *predicted taken* branches and jumps held in BTB

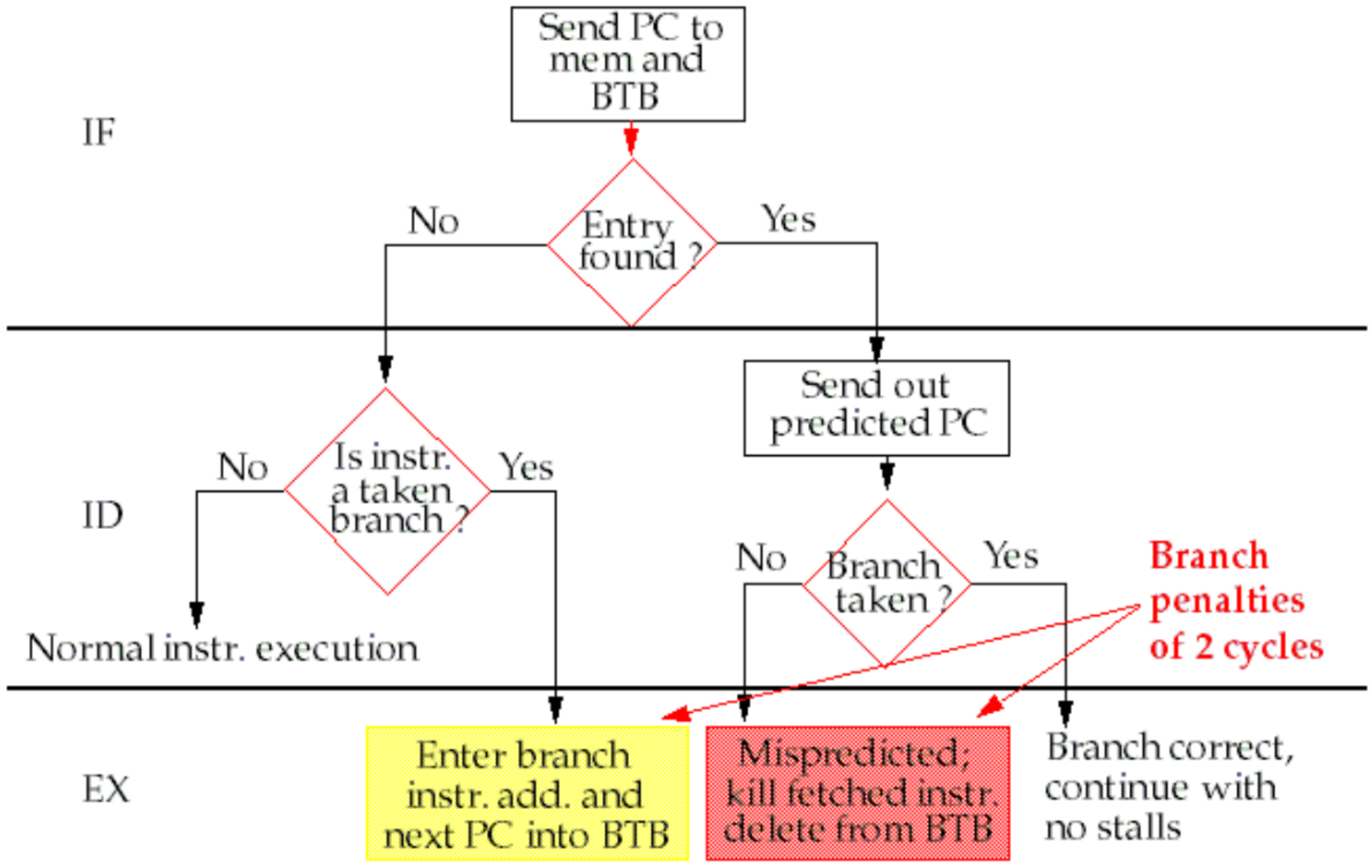
Next PC determined *before* branch fetched and decoded

later: *check prediction, if wrong kill instruction, update BPb*

Branch Target Buffer (3)

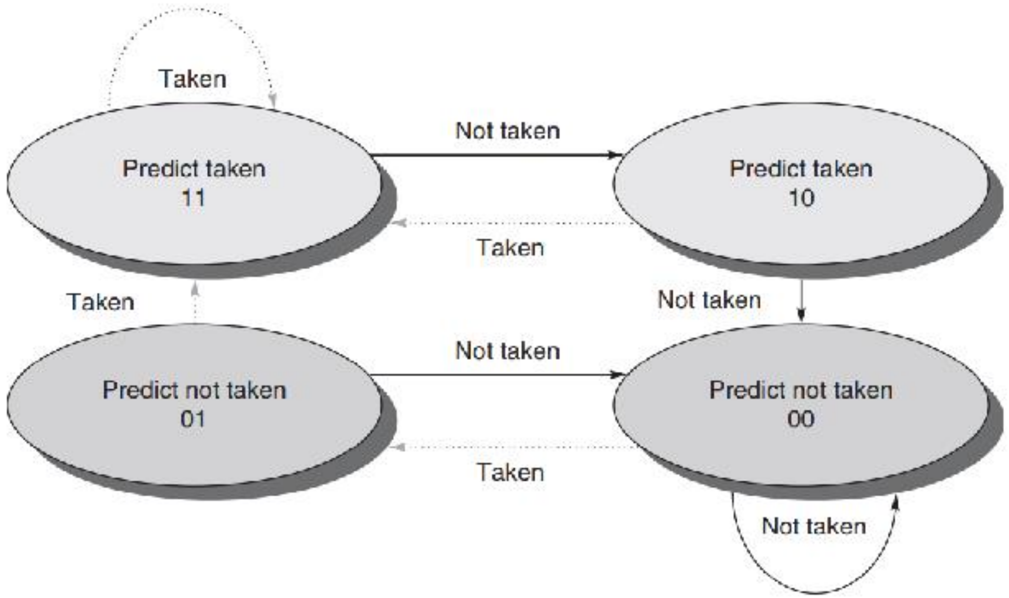
Branch-Target Buffers

- *Steps in handling an instruction with a Branch-Target Buffer.*



Problem Solving: Branch Prediction (1)

Consider the last 16 actual outcomes of a single static branch. T means branch is taken and N means not taken.
{oldest→ T T N N T N T T T N T N T T N T ← latest}
A two level branch predictor of (1,2) type is used. Since there is only one branch in the program indexing to BHT with PC is irrelevant. Hence only last branch outcome only is used to index to the BHT. How many mis-predictions are there and which of the branches in this sequence would be mis-predicted? Fill up the table for 16 branch outcomes.



Problem Solving: Branch Prediction (2)

Sl.No	Last Outcome	BHT N/T	Prediction	Outcome	Mis-Pre Y/N ?
1	N (initial)	00 / 11	N	T	YES
2	T	01 / 11	T	T	NO
3	T	01 / 11	T	N	YES
4	N	01 / 10	N	N	NO
5	N	00 / 10	N	T	YES
6	T	01 / 10	T	N	YES
7	N	01 / 00	N	T	YES
8	T	11 / 00	N	T	YES
9	T	11 / 01	N	T	YES
10	T	11 / 11	T	N	YES
11	N	11 / 10	T	T	NO
12	T	11 / 10	T	N	YES
13	N	11 / 00	T	T	NO
14	T	11 / 00	N	T	YES
15	T	11 / 01	N	N	NO
16	N	11 / 00	T	T	NO

T
T
N
N
T
N
T
T
T
N
T
N
T
T
N
T