

Digital Logic Design + Computer Architecture

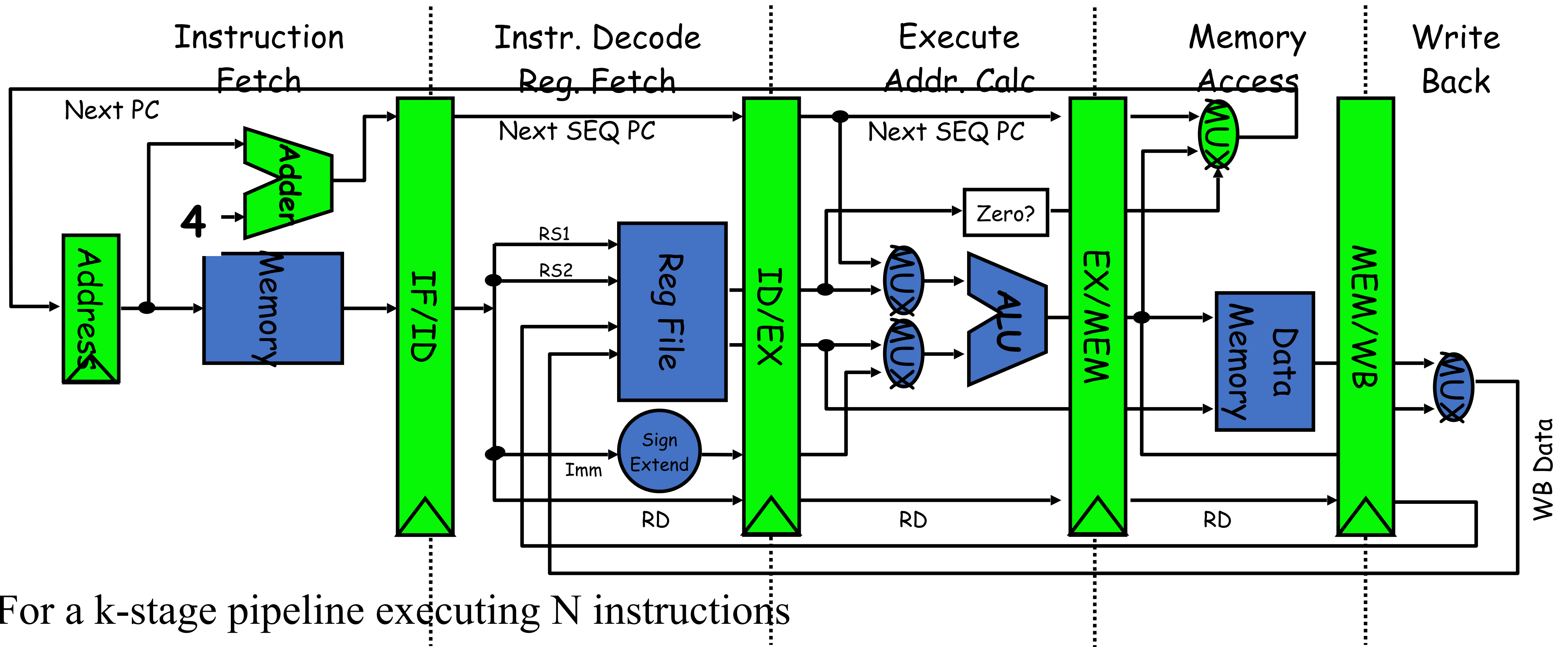
Sayandeep Saha

Assistant Professor
Department of Computer
Science and Engineering
Indian Institute of Technology
Bombay



Pipeline Hazards

Pipeline Recap...



For a k-stage pipeline executing N instructions

first instruction: k cycles

Next N-1 instructions: N-1 cycles, total = K + (N-1) cycles

Pipeline In Real World

Limited resources

Stages does not take uniform time

Inter-instruction dependency

Branches

Limited resources

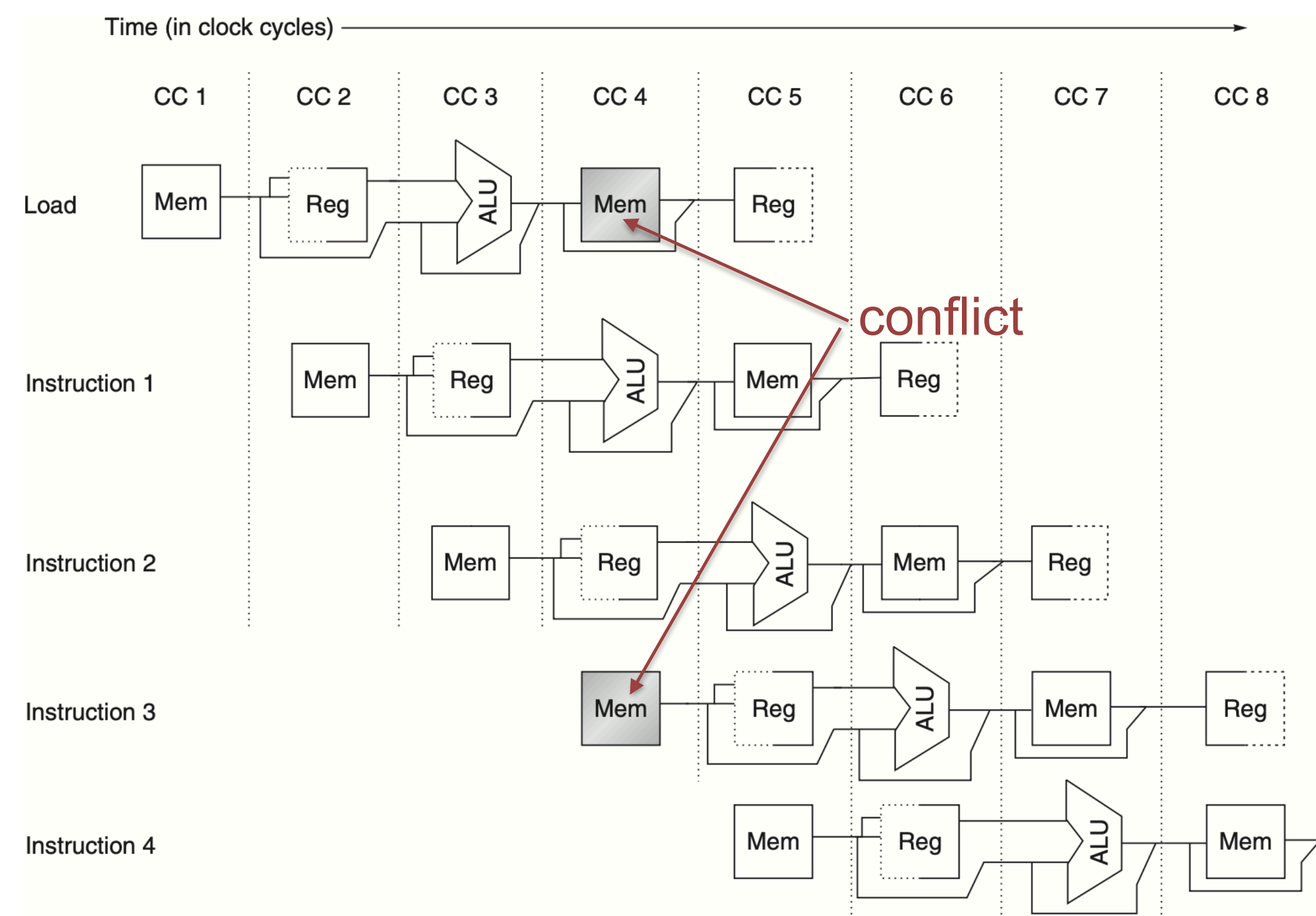
Pipeline Hazards

- Hazards are events in which prevents an instruction going down the pipeline. The pipeline is **stalled**
 - Structural hazards
 - Data hazards
 - Control hazards



Structural Hazards

- When two instructions want to access the same resource at the same clock cycle.

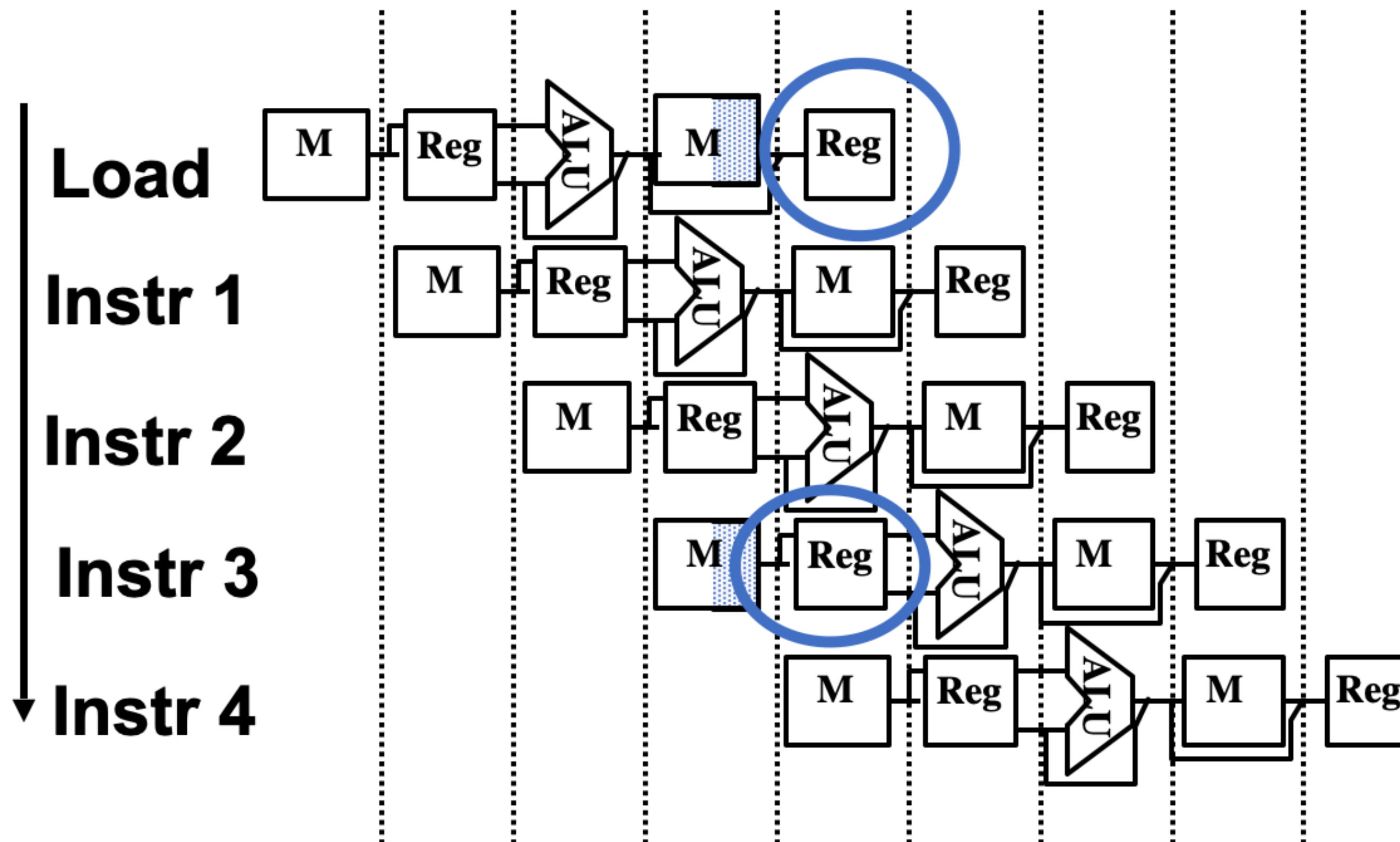


Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				Stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

- Issue: two instruction wants to access the memory simultaneously
 - One reading data
 - Other reading instruction.
- Solution: Separate instruction and data memory

Structural Hazards

- Can also happen for register files.



- Issue: conflict in ID and WB stage
 - Insufficient number of read/write ports
 - Read write in the same cycle, but no “write before read” convention.

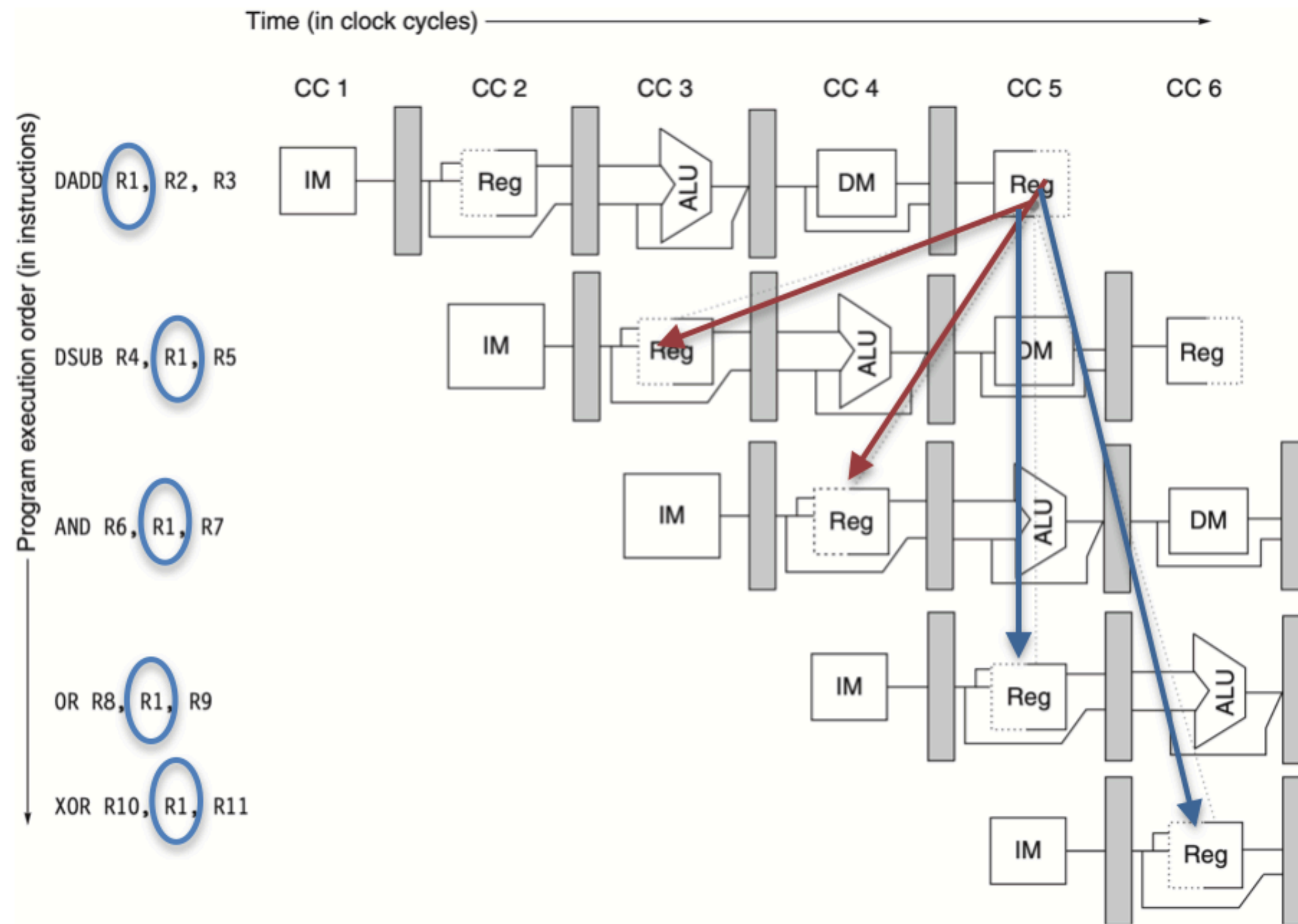
Solution:

- Separate and multiple read write ports
- Write in the first half of the clock cycle and read in the second half.

- Structural hazards are relatively rare in modern processors — compilers are smart.
- Only happens for less frequently used functional units

Data Hazards


- Hazards arising due to data dependency



- The first DADD writes R1 at WB stage.
- All succeeding instructions reads the R1 result
- So, all instruction except the last OR and XOR has to wait for the WB of the first instruction.
- So we need two stall cycles or something else!!

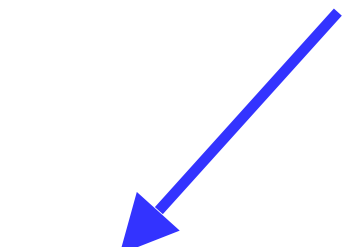
Data dependencies

add **R1**, R2, R3
sub R2, R4, **R1**
or R1, R6, R3




read-after-write
(RAW)
True dependence

add R1, **R2**, R3
sub **R2**, R4, R1
or R1, R6, R3



write-after-read
(WAR)
Anti dependence

add **R1**, R2, R3
sub R2, R4, R1
or **R1**, R6, R3



write-after-write
(WAW)
Output dependence

Data Hazards

Read-After-Write (RAW)

- Read must wait until earlier write finishes

Anti-Dependence (WAR)

- Write must wait until earlier read finishes. Not possible with vanilla 5-stage pipeline

• Output Dependence (WAW)

- Earlier write can't overwrite later write
Not possible with vanilla 5-stage pipeline)

Control Hazards

- Hazards arising due to branching...
- **Remember!!!** Branch target is not known during fetch.
- if a branch changes the PC to its target address, it is a *taken* branch.
- Else it is *untaken*.

Control Hazards

- Hazards arising due to branching...
- What happens to the instructions at 14, 18, 22?

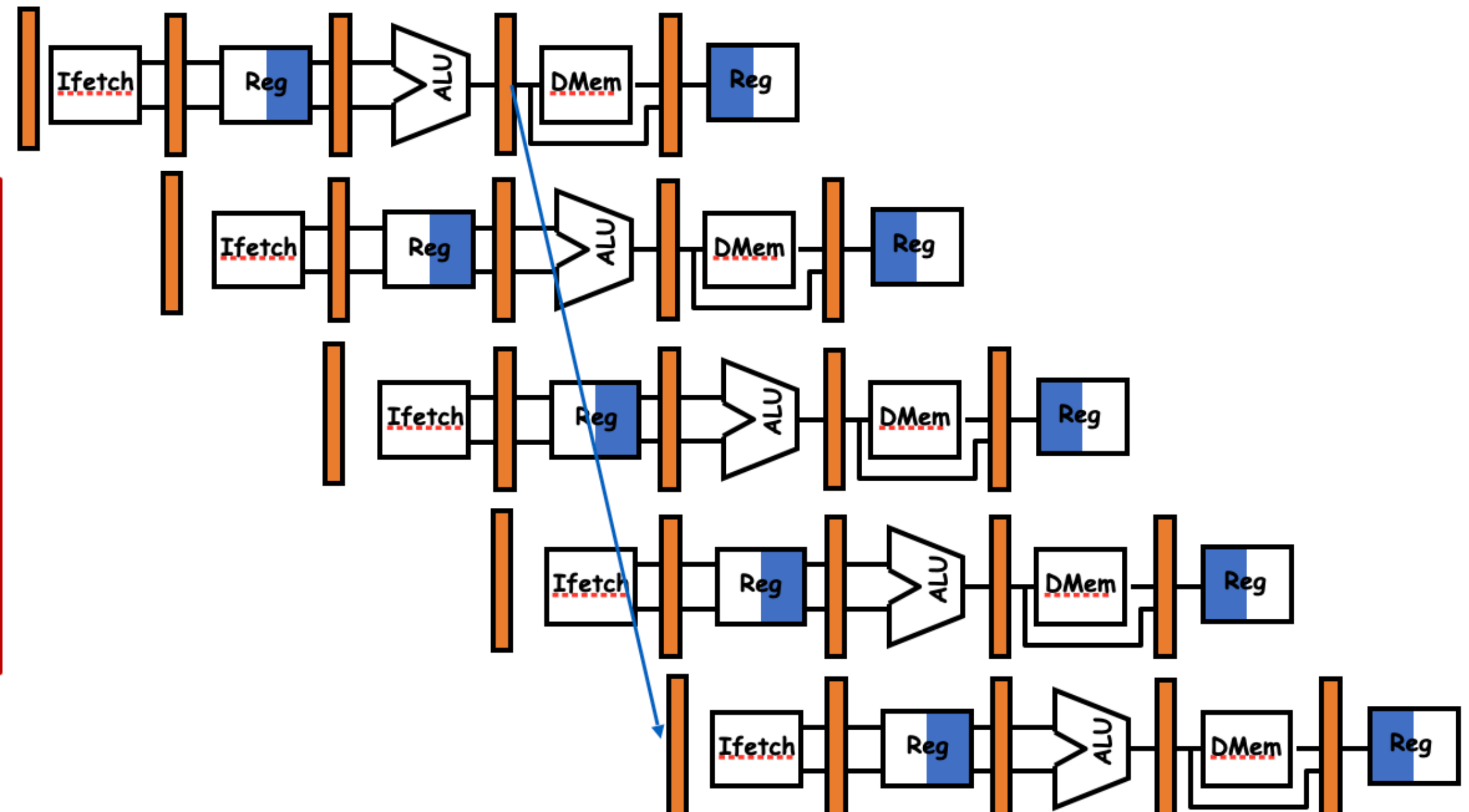
10: beq r1,r3,36

14: and r2,r3,r5 ☹

18: or r6,r1,r7 ☹

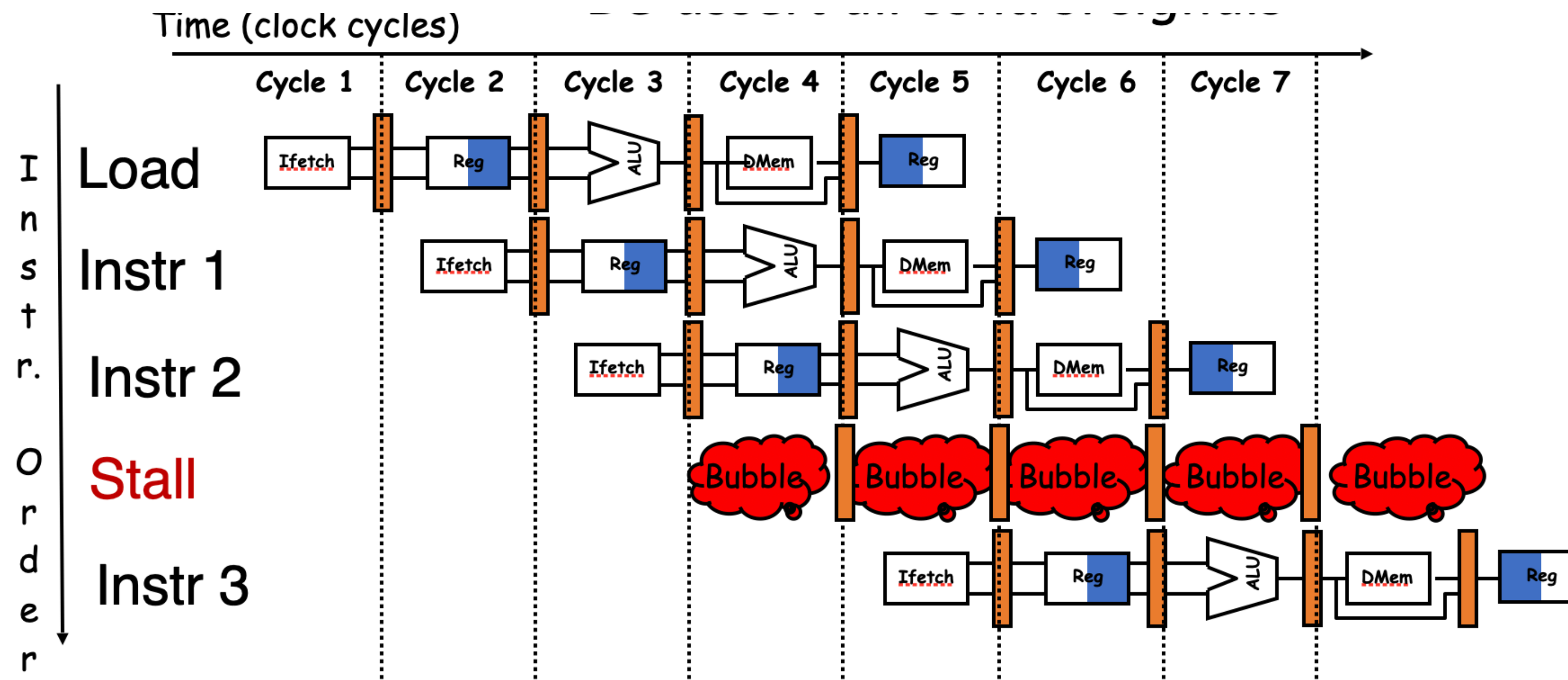
22: add r8,r1,r9 ☹

50: xor r10,r1,r11



What is a Stall

- Putting bubbles in pipeline.
- Actually wasting clock cycles — in a stall cycle no instruction can enter the pipeline
- De-assert all control signals
- Compiler way — put a `nop` instruction. — eg, `sll $0 $0` (in MIPS)



Control Hazard and Stalls

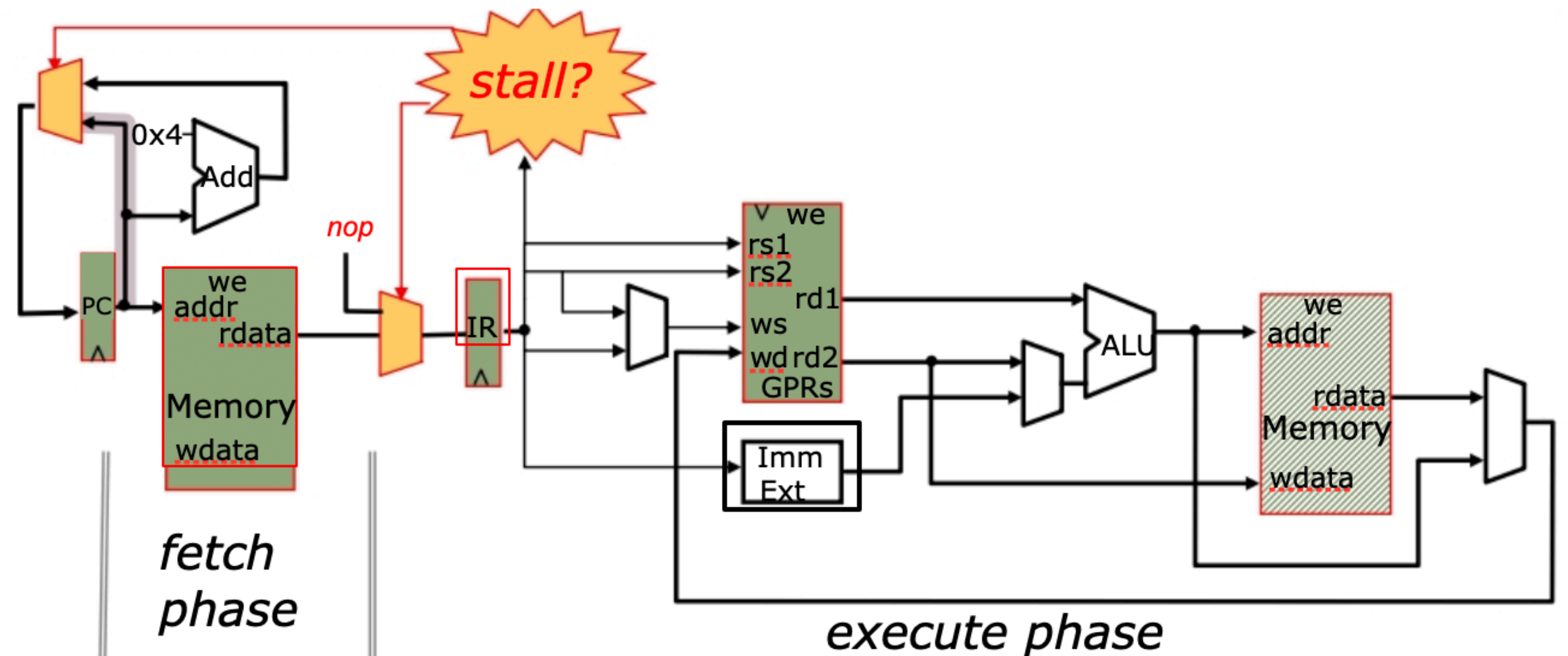
- The earliest we can get to know the branch outcome is at the end of ID stage (needs some simple hardware modification)

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

This IF has to be undone — results in a stall cycle

How Stall is implemented in Pipelines

- Can be detected from the content of the pipeline registers.
- Upon detecting stall, just do not update the PC and dessert all control signals



Data Hazard Detector and stalls

EX to DEC:

EX/MEM.RegisterRd = ID/EX.RegisterRs

EX/MEM.RegisterRd = ID/EX.RegisterRt

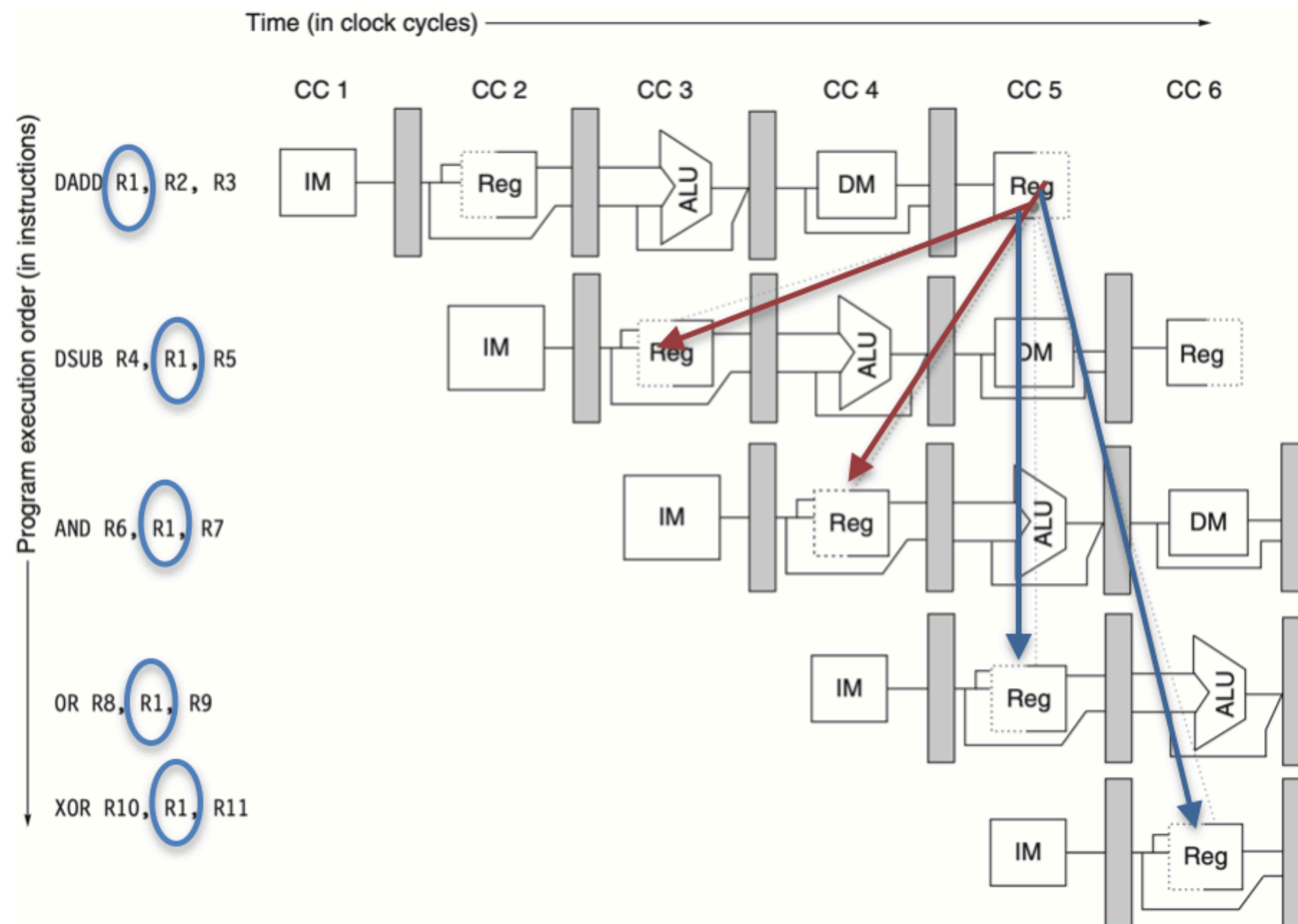
MEM to DEC:

MEM/WB.RegisterRd = ID/EX.RegisterRs

MEM/WB.RegisterRd = ID/EX.RegisterRt

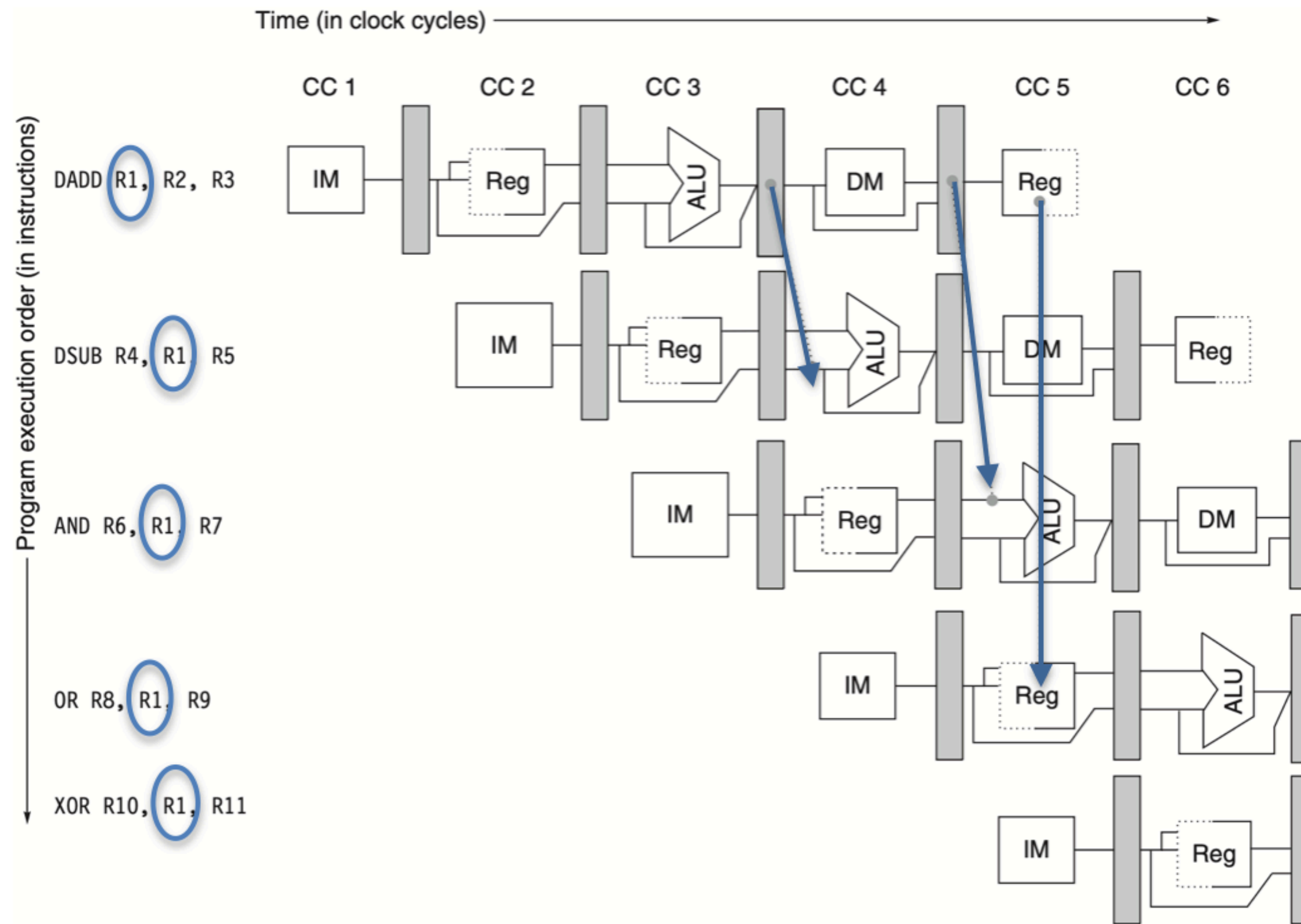
Data Hazards

- Hazards arising due to data dependency



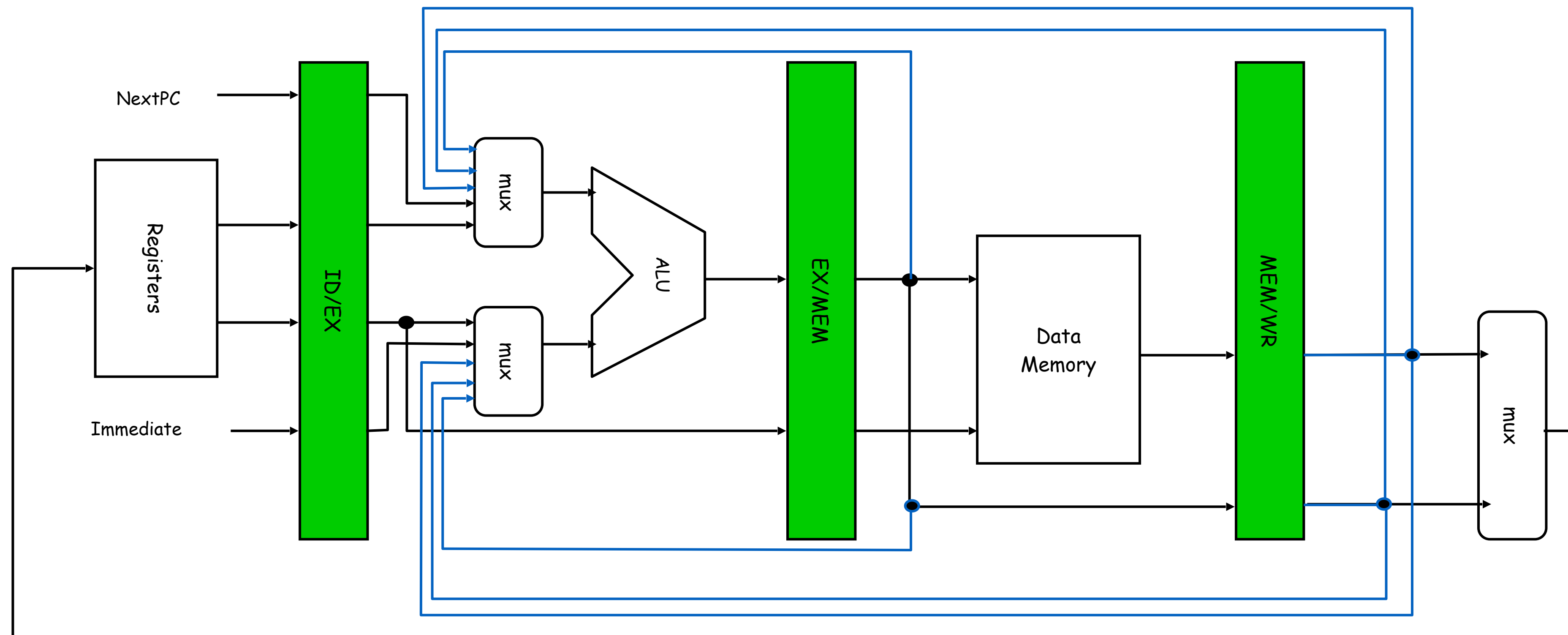
- The first DADD writes R1 at WB stage.
- All succeeding instructions reads the R1 result
- So, all instruction except the last OR and XOR has to wait for the WB of the first instruction.
- So we need two stall cycles or something else!!

Handling Data Hazards: Forwarding

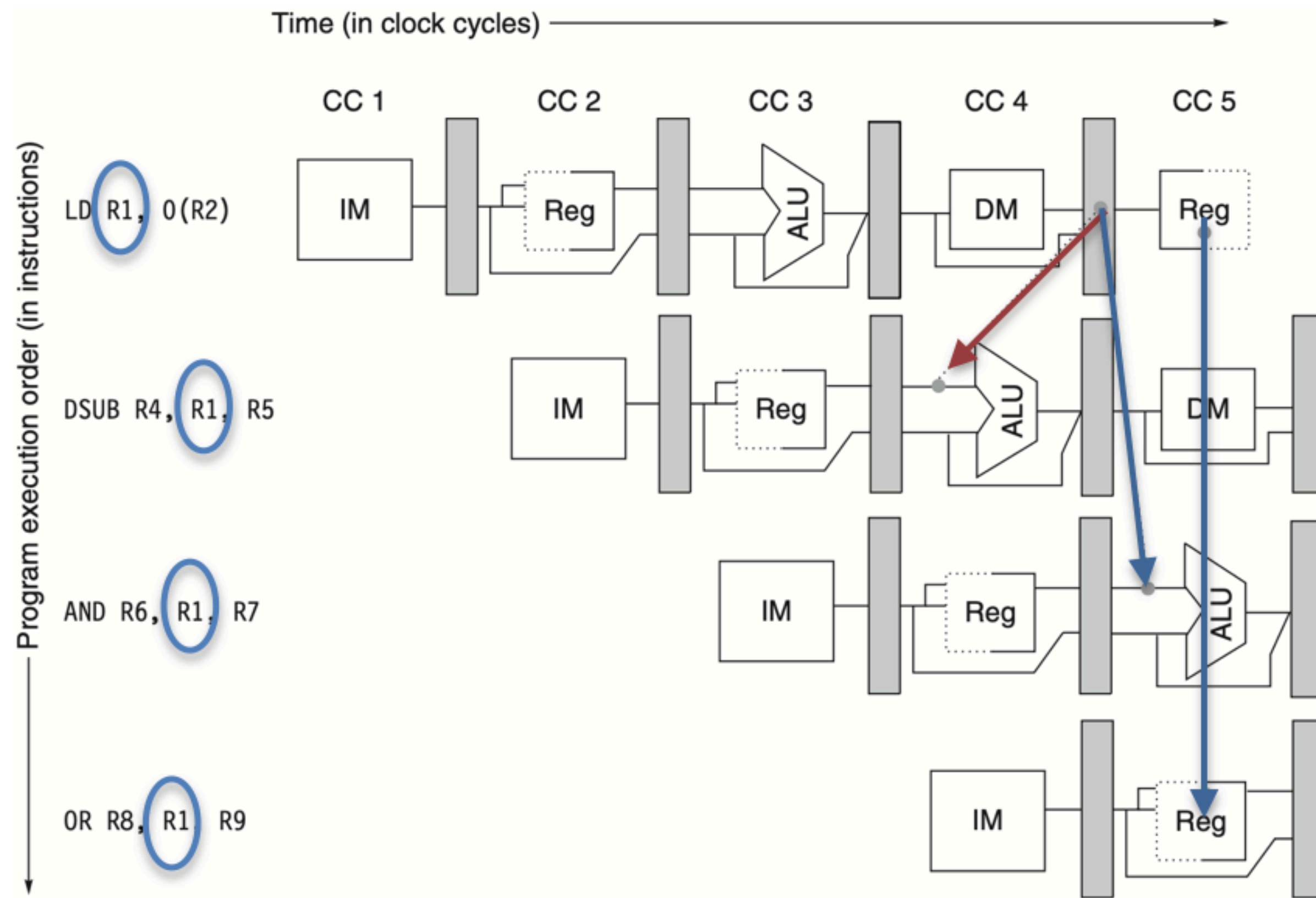


- No stalls!!
- Can be generalised — any functional unit generating data can forward to any other input whenever needed.

Handling Data Hazards: Forwarding



Can Forwarding Solve All the Problems?



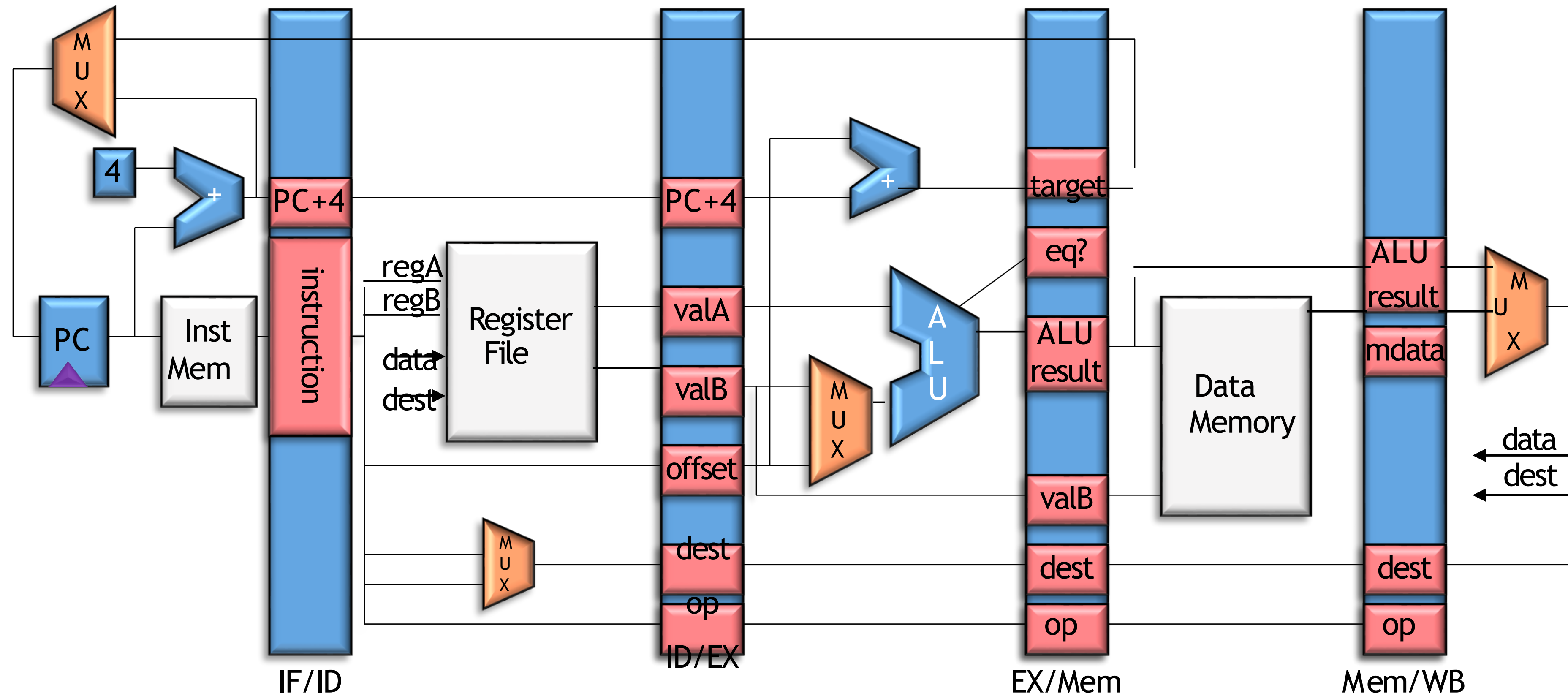
- No problems for the AND and OR — forwarding works fine
- But we cannot forward for the DSUB as it is backward in time.
- So one cycle stall is needed.

What Happens to Speedup with Stalls

$$Speedup = \frac{CPI\ Unpipelined}{CPI\ Ideal + stall\ cycles}$$

- CPI Ideal = 1
- Also assume stages are perfectly balanced so that if the unpipelined cycle time is T, the pipelined cycle time becomes T/k for a k stage pipeline. — so easy to cancel out T

The Complete Picture



Control Hazards

What do we need to calculate—next PC?

- For Jumps
 - Opcode, offset, and PC
- For Jump Register
 - Opcode and register value
- For Conditional Branches
 - Opcode, offset, PC, and register (for condition)
- For all others

Control Hazards

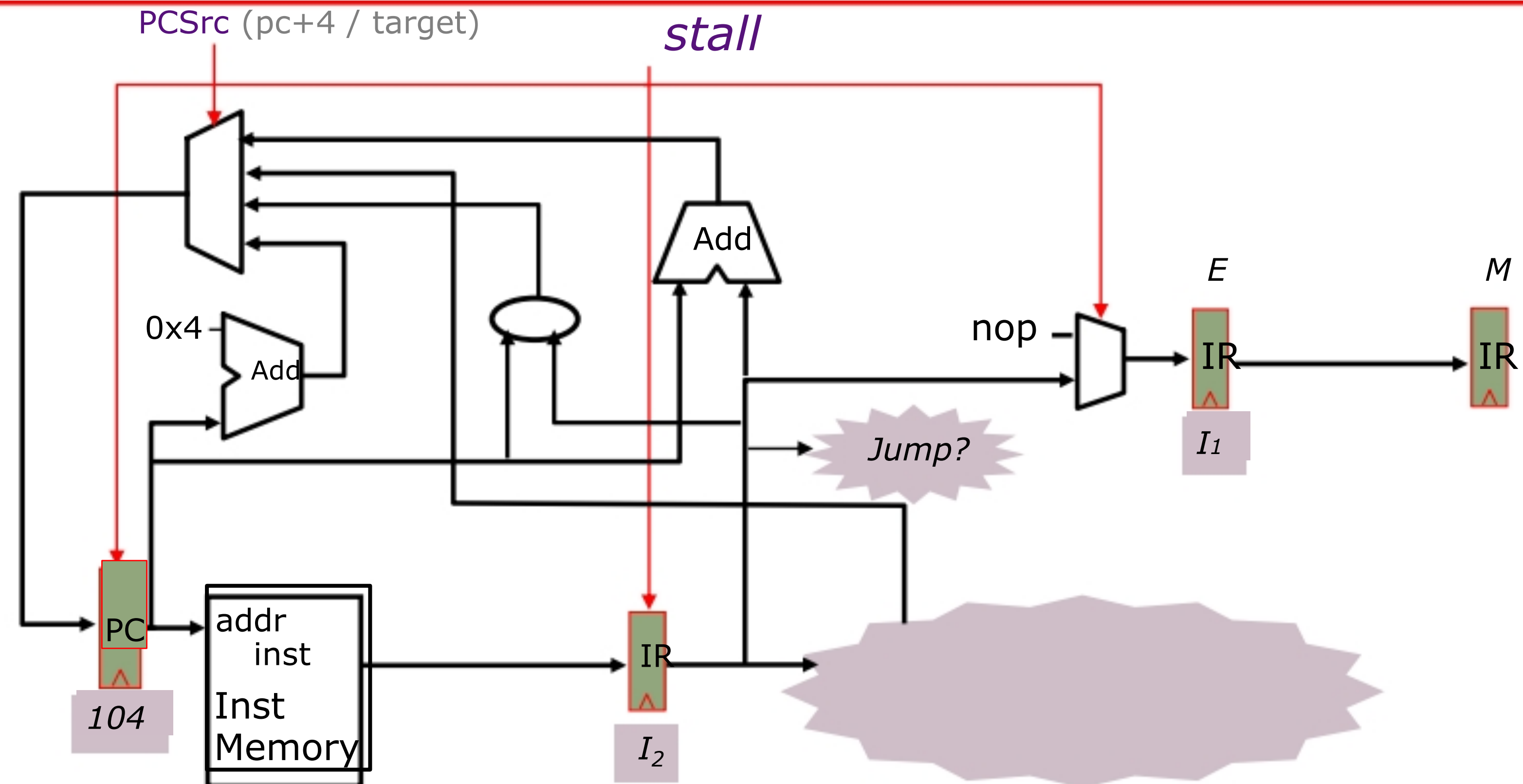
What do we need to calculate next PC?

- For Jumps
 - Opcode, offset, and PC
- For Jump Register
 - Opcode and register value
- For Conditional Branches
 - Opcode, offset, PC, and register (for condition)

In what stage do we know these?

- PC - Fetch
- Opcode, offset - Decode (or Fetch?)
- Register value - Decode
- Branch condition $((rs) == 0)$ - Execute (or Decode?)

Speculate, PC=PC+4



I ₁	096	ADD
I ₂	100	J 304
I ₃	104	ADD
I ₄	304	ADD

What happens on mis-speculation, i.e., when next instruction is not PC+4?

kill

How? Insert NOPs

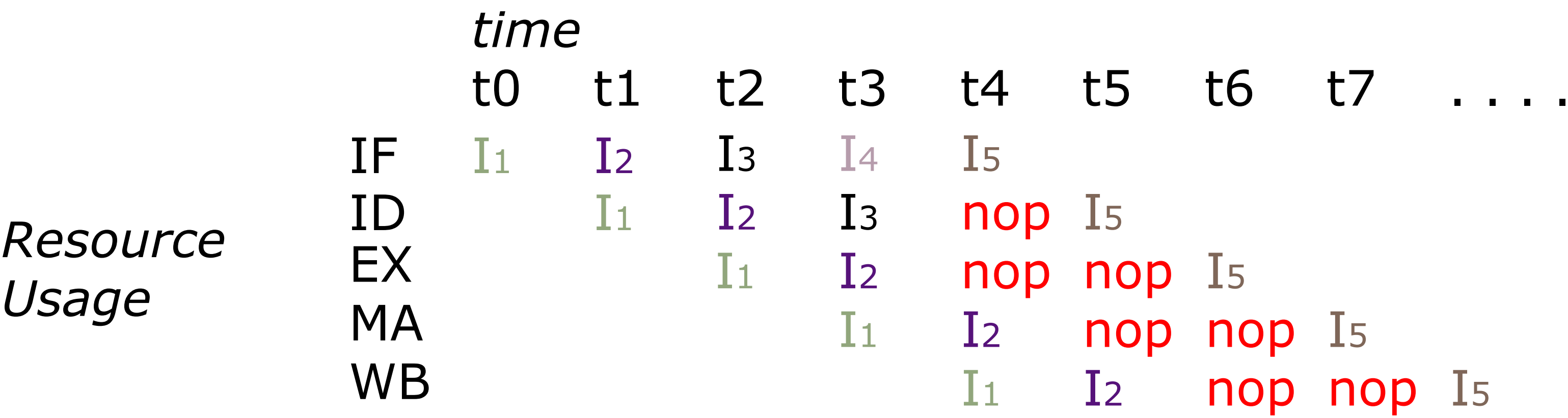
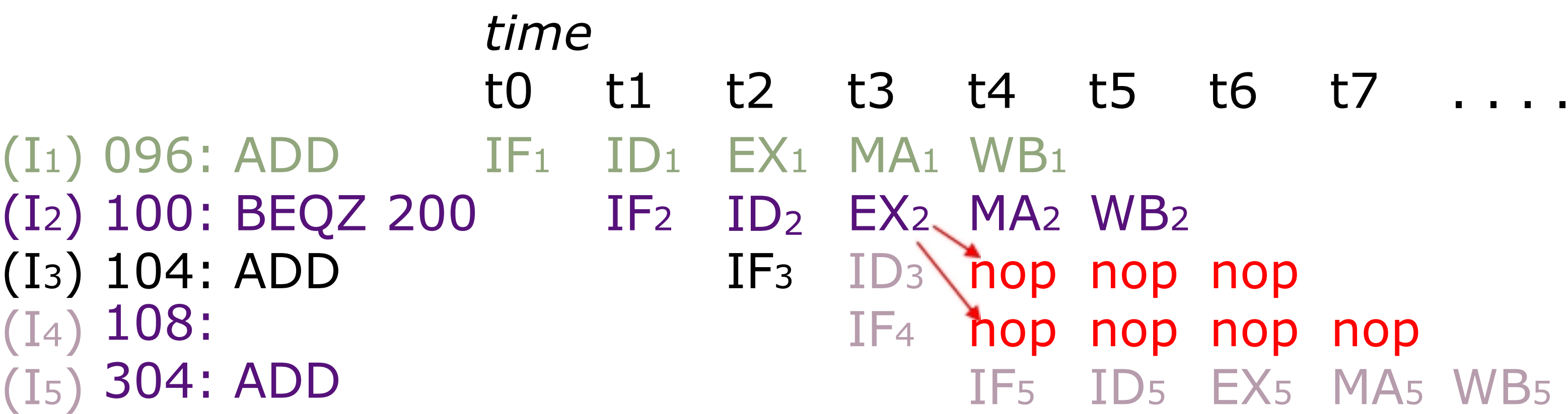
Conditional branches

I ₁	096	ADD
I ₂	100	BEQZ r1 200
I ₃	104	ADD
I ₄	304	ADD

Branch condition is not known
until the execute stage

Instructions between a branch instruction and the target are
in the **wrong-path** if the branch is not taken

Again (stalls/NOPs)



What else can be done? Compiler?

Delayed branch: Define branch to take place **AFTER** a following instruction (used to be in early RISC processors)

branch instruction

sequential successor₁

sequential successor₂

.....

sequential successor_n

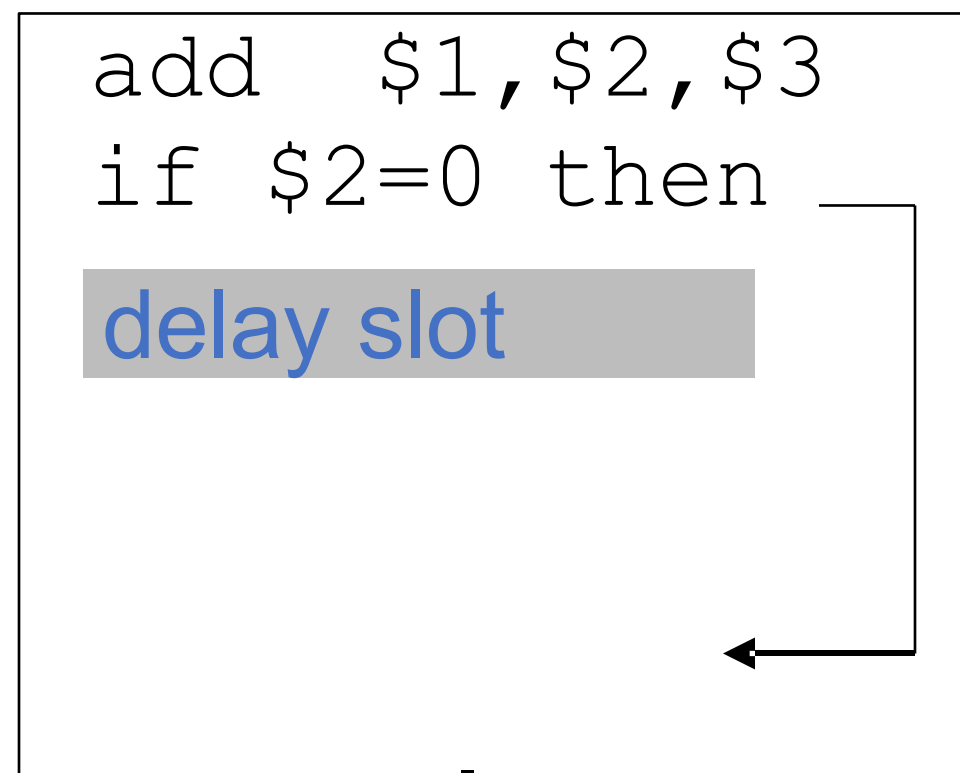
branch target if taken



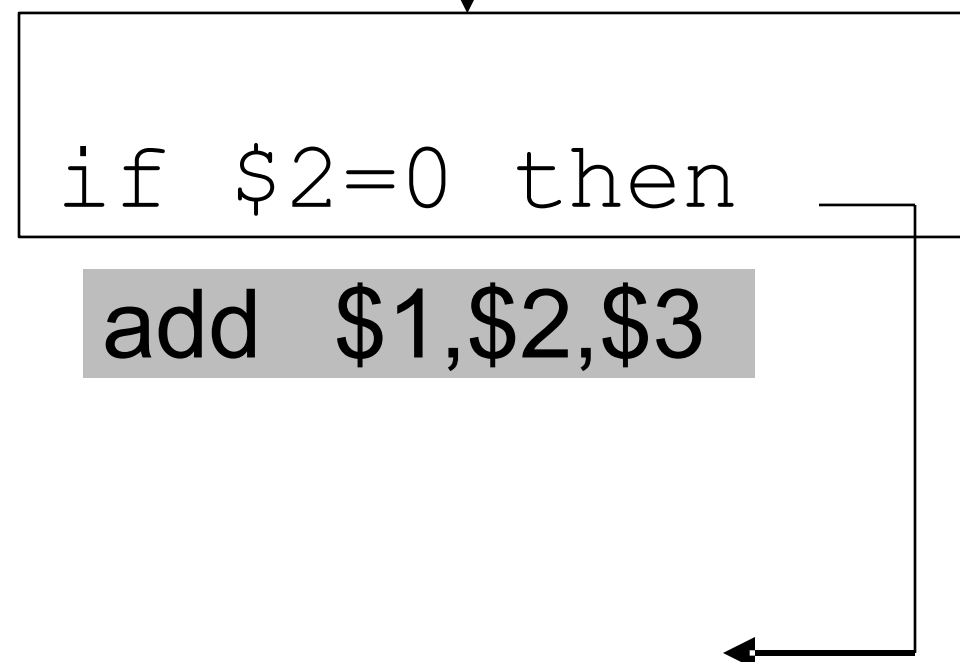
Branch delay of length n

Scheduling Branch Delay Slots

A. From before branch



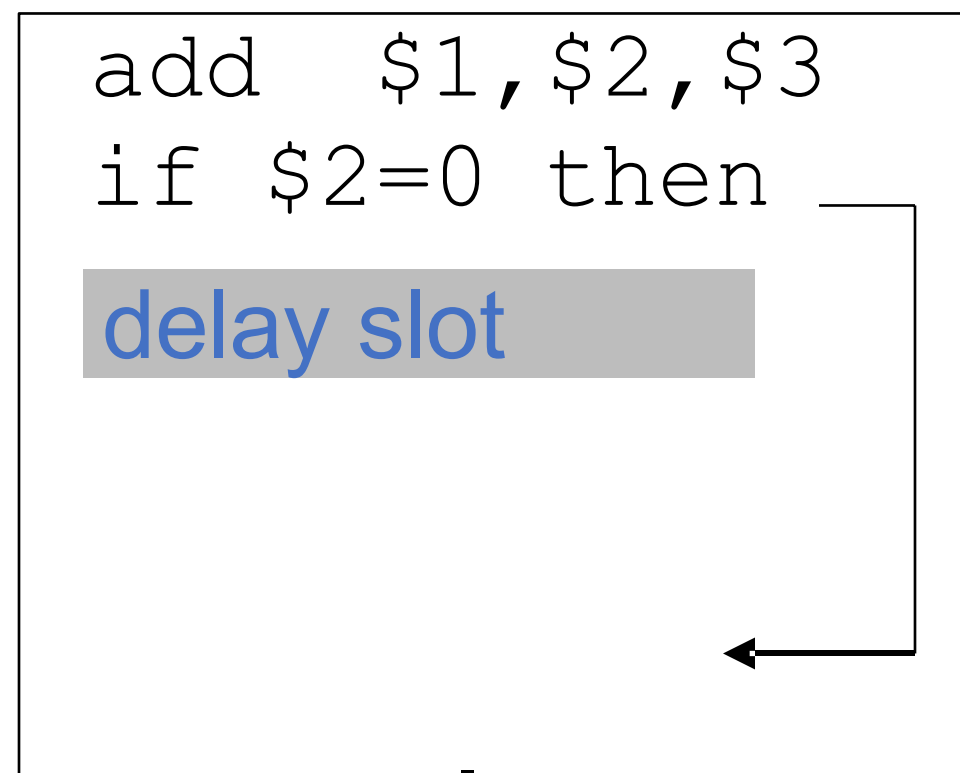
becomes



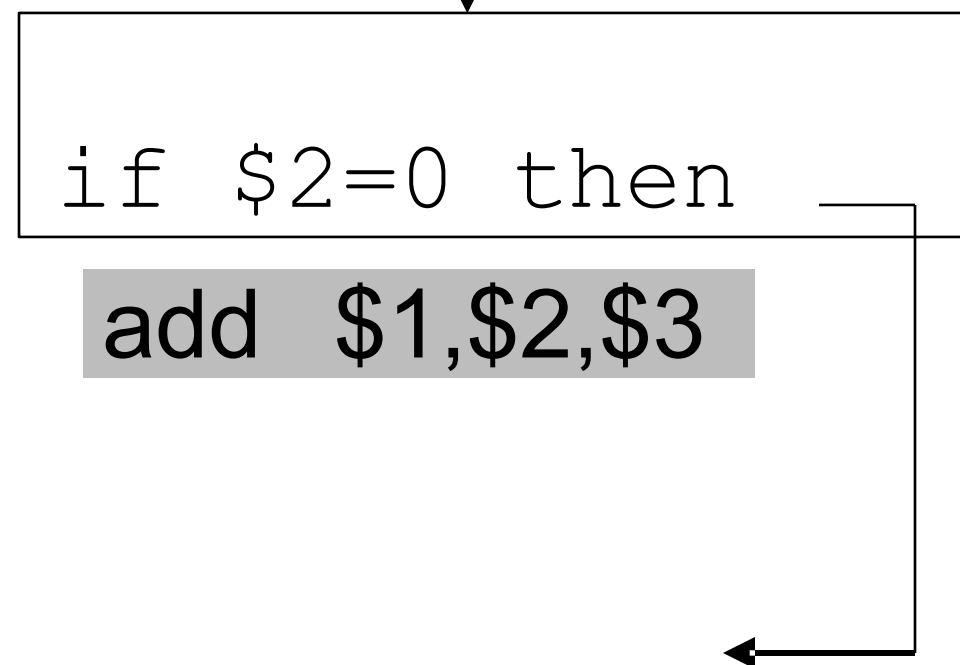
A is the best choice, fills delay slot & reduces instruction count (IC)

Scheduling Branch Delay Slots

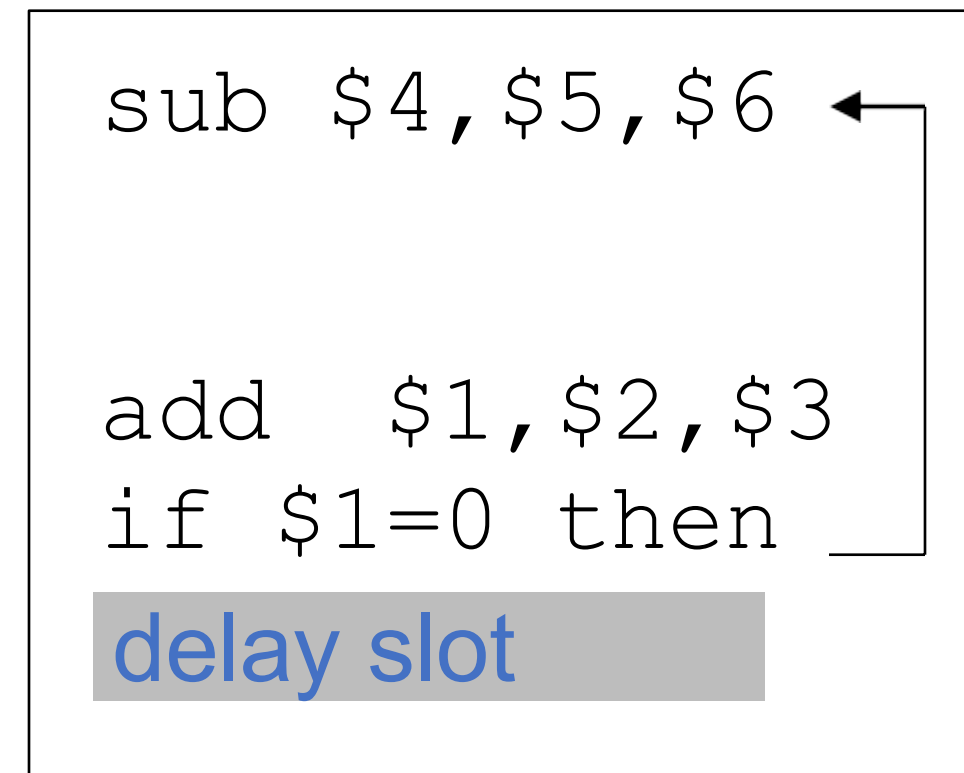
A. From before branch



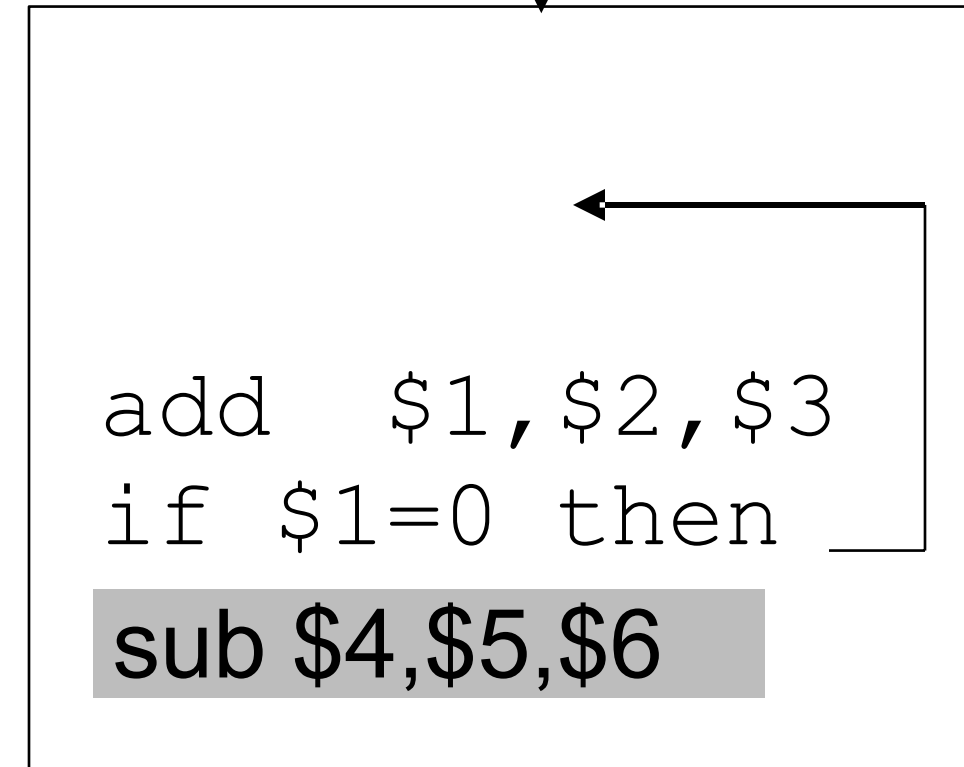
becomes



B. From branch target



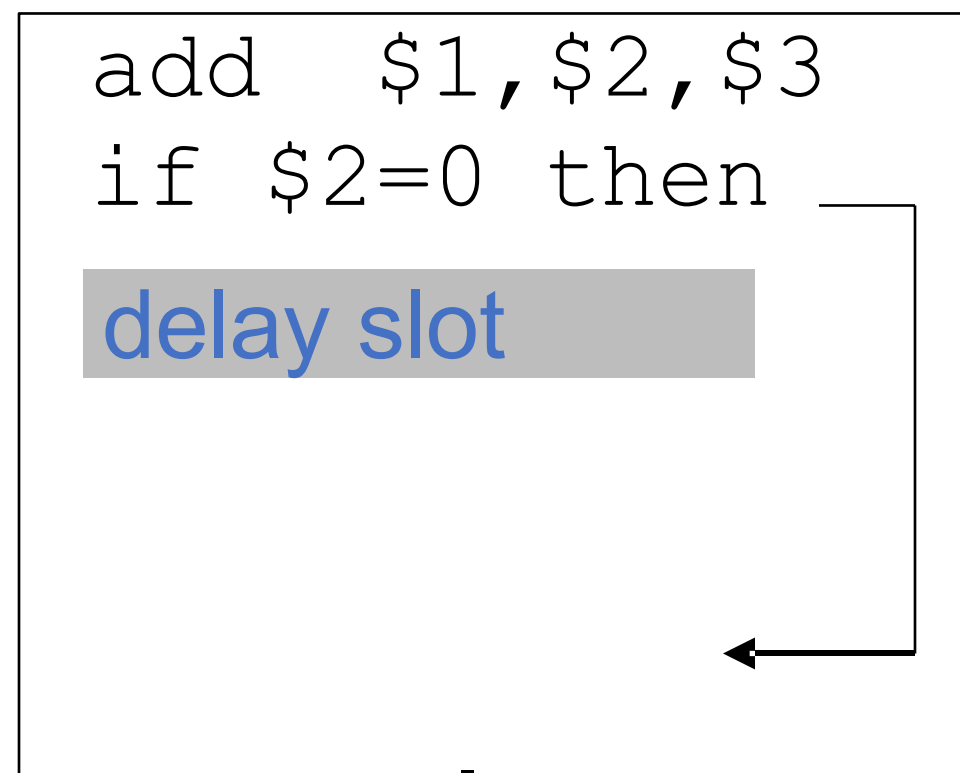
becomes



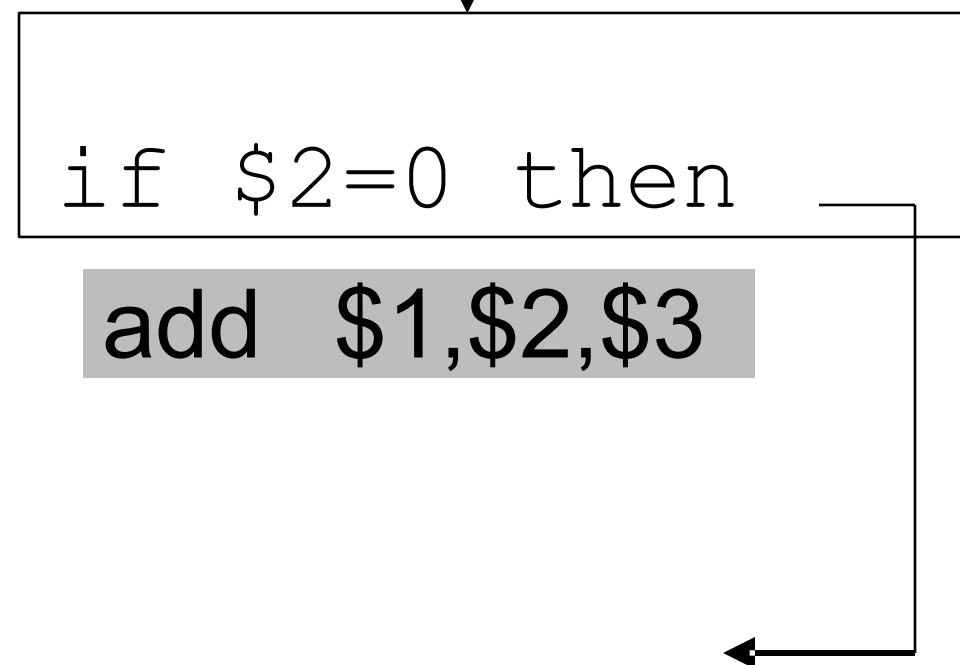
A is the best choice, fills delay slot & reduces instruction count (IC)

Scheduling Branch Delay Slots

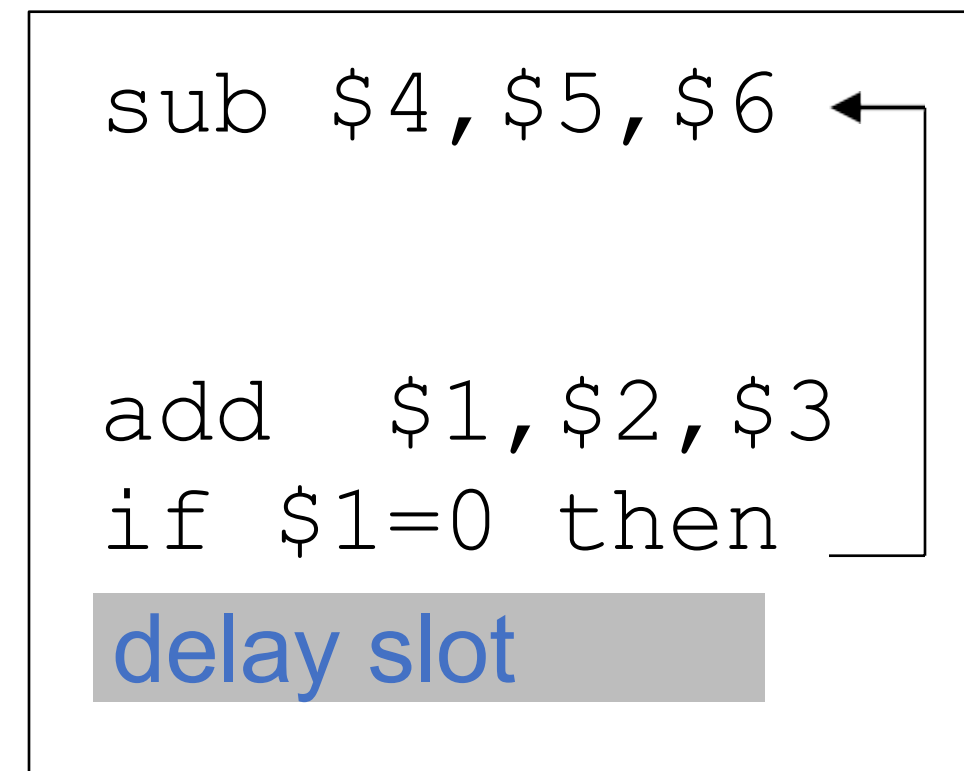
A. From before branch



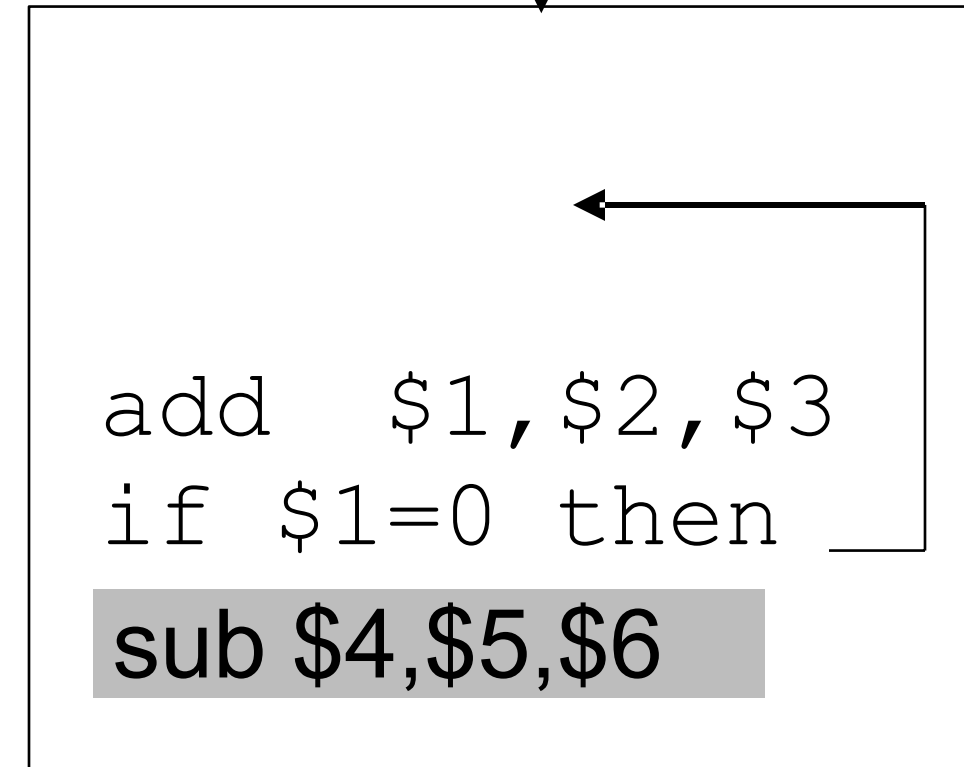
becomes



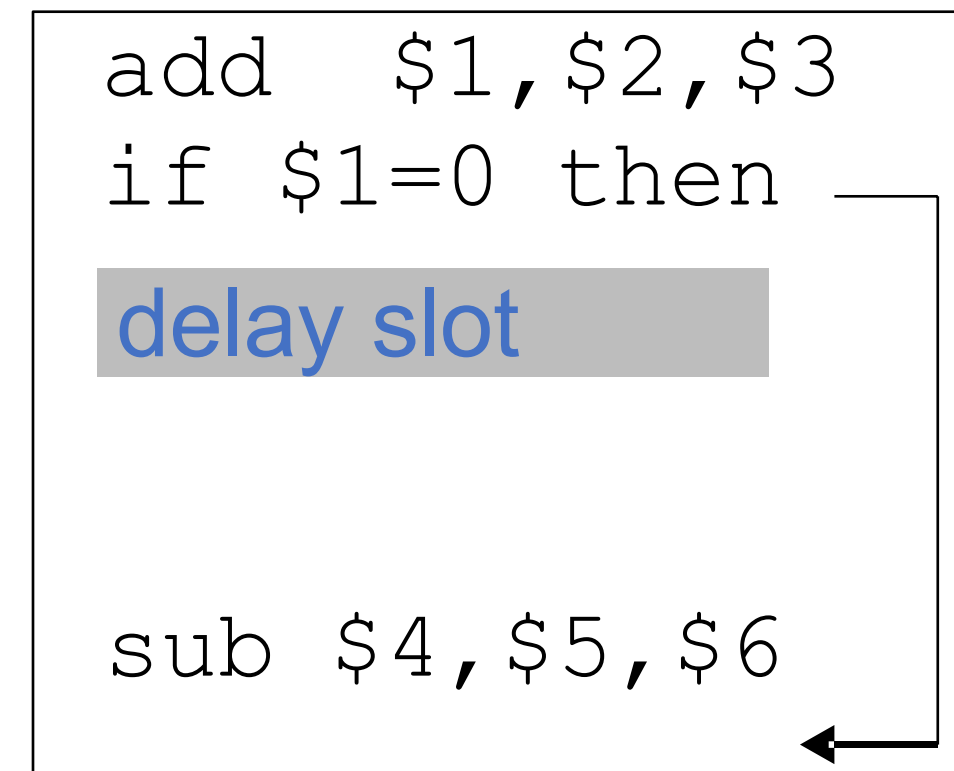
B. From branch target



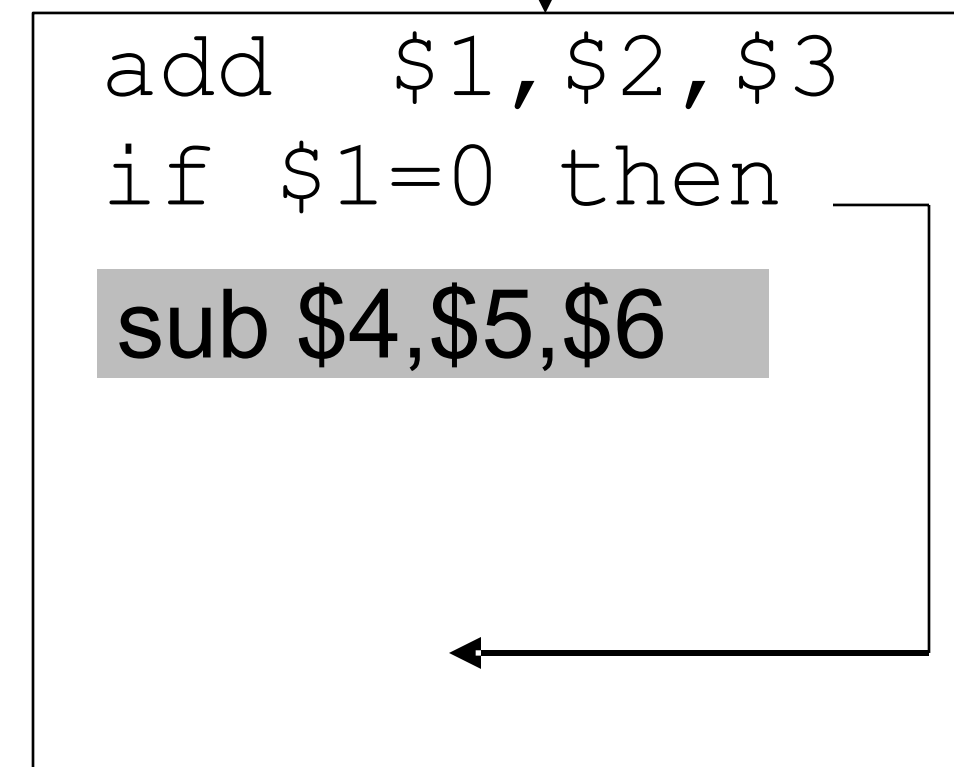
becomes



C. From fall through



becomes



A is the best choice

Do not put a branch in the delay slot :P

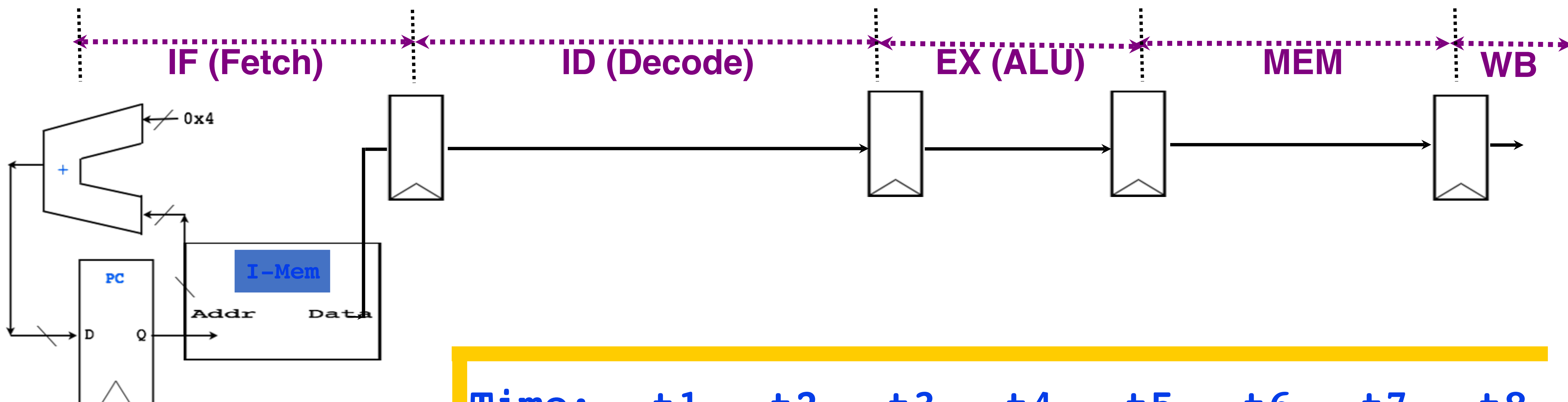
New Pipeline Speedup

Pipeline Speedup = Pipeline Depth

1+pipeline stalls because of branches

Pipeline stalls (branches) = Branch frequency X penalty

Branch instructions

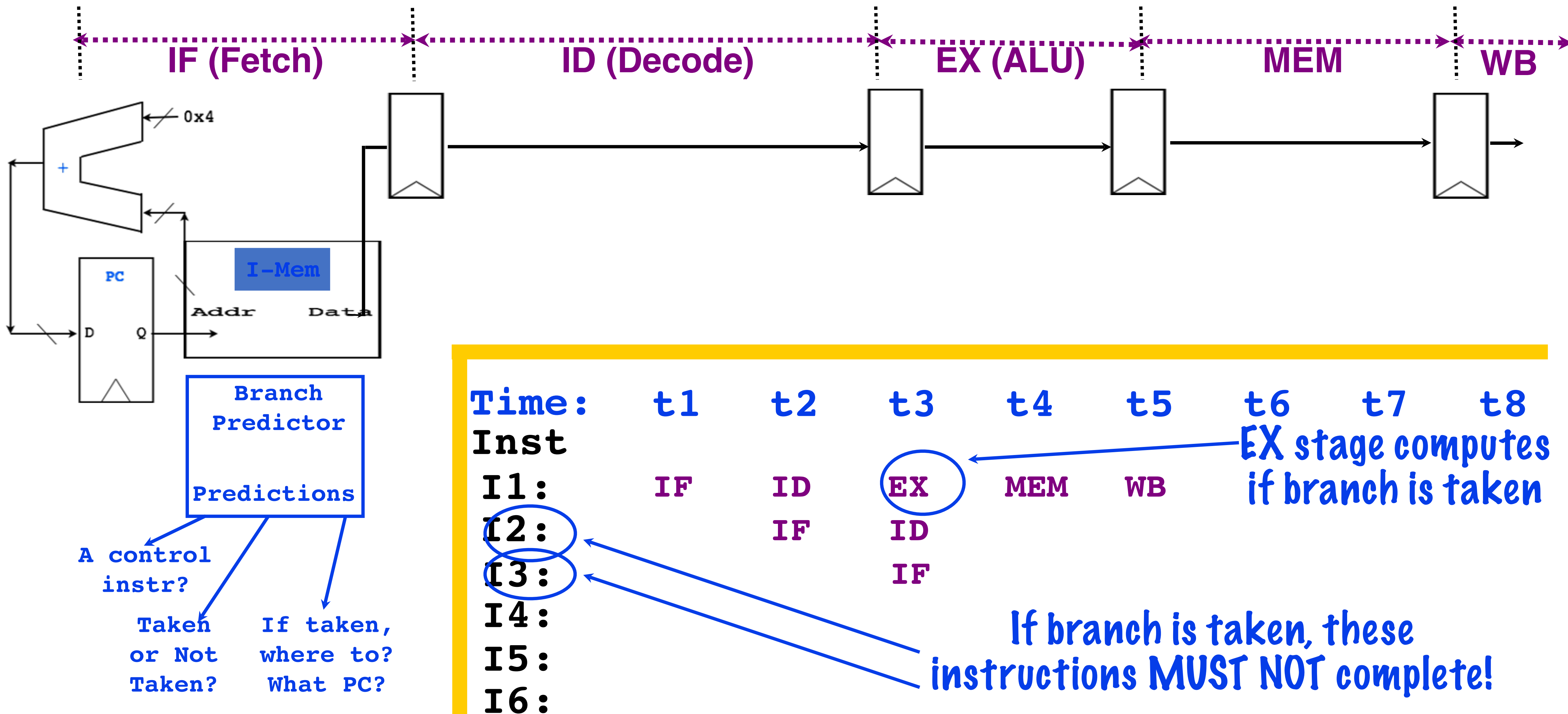


Time:	t1	t2	t3	t4	t5	t6	t7	t8
Inst								
I1:	IF	ID	EX	MEM	WB			
I2:		IF	ID					
I3:			IF					
I4:								
I5:								
I6:								

If branch is taken, these instructions **MUST NOT** complete!

EX stage computes if branch is taken

Branch Predictors



Branch Predictors

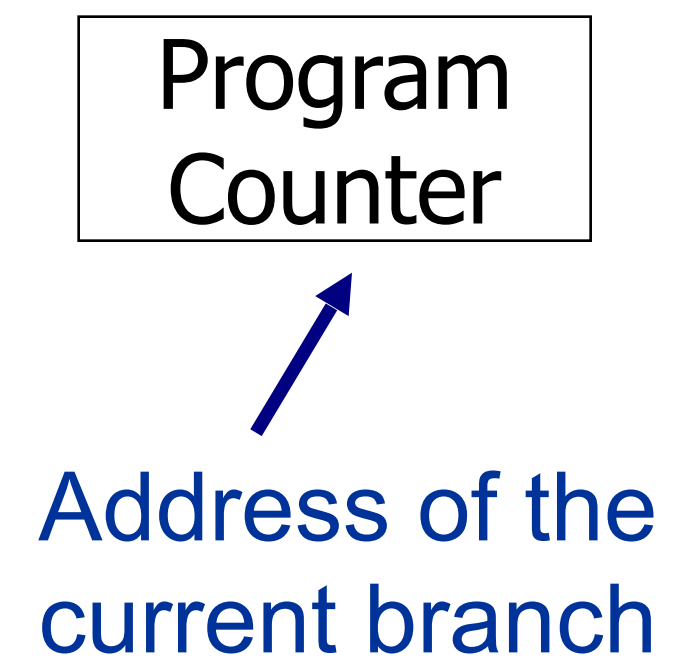
- Predict whether the next PC is a branch PC in the fetch stage
- But:
 - If it is branch, will it be taken?
 - What is the target address?
 - Not known at fetch.....

Branch Predictor: A bit deeper

Three tasks

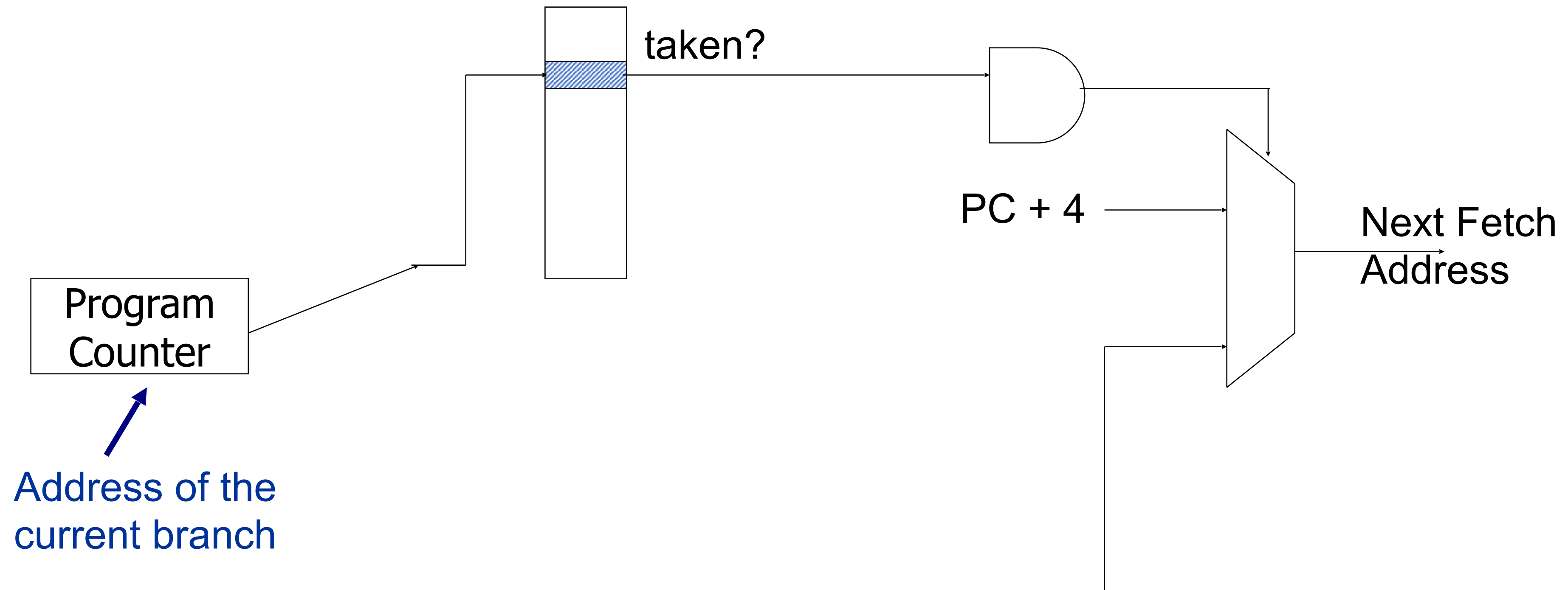
1. Is the PC a branch/jump? YES/NO
2. If Yes, can we predict the direction? Taken or not-taken
3. If taken, can we predict the target address?

Branch Predictor: A bit deeper



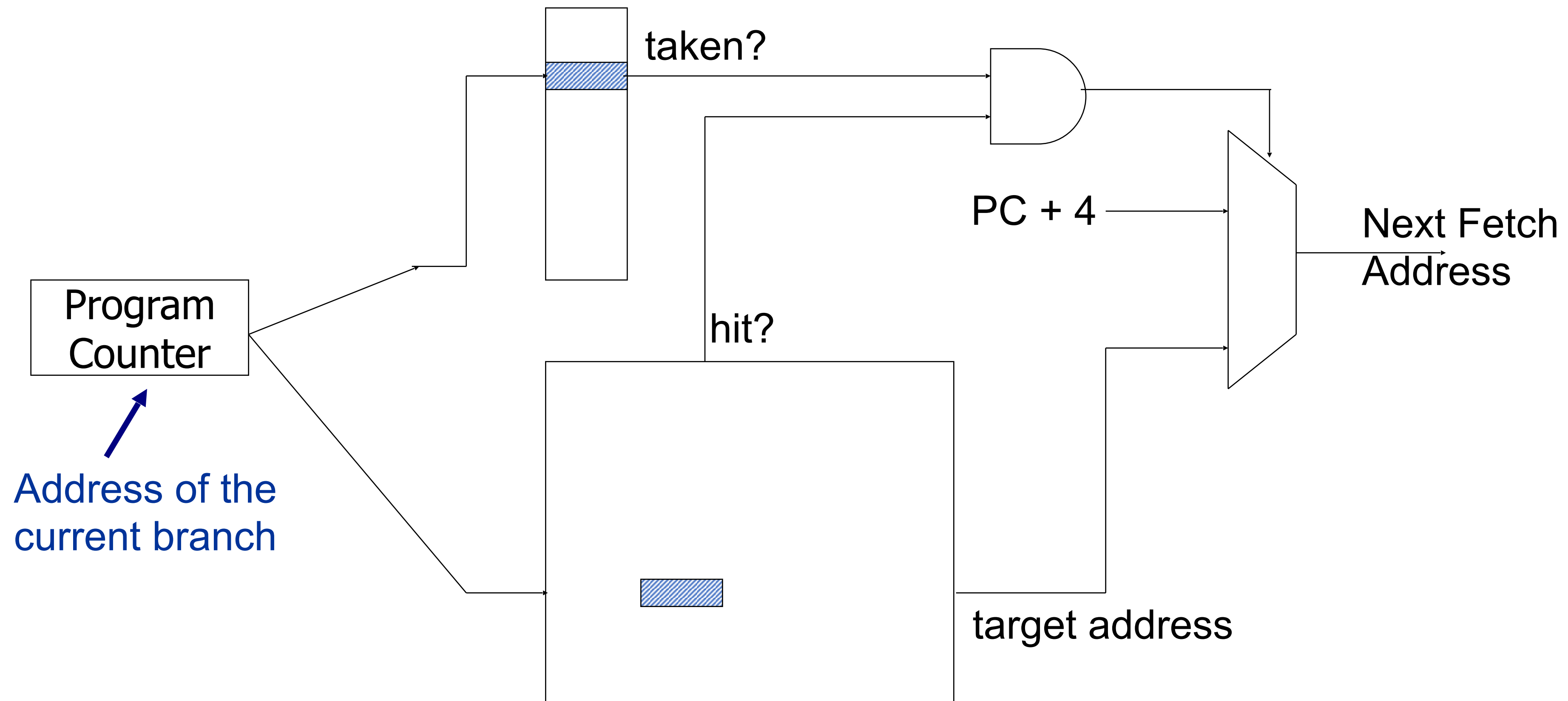
Branch Predictor: A bit deeper

Direction predictor



Branch Predictor: A bit deeper

Direction predictor



Repository of Target Addresses (BTB: Branch Target Buffer)

Static (compiler) Direction Prediction

Always not-taken: Simple to implement: no need for BTB,
no direction prediction

Low accuracy: ~30-40%

Always taken: No direction prediction, we need BTB though

Better accuracy: ~60-70%

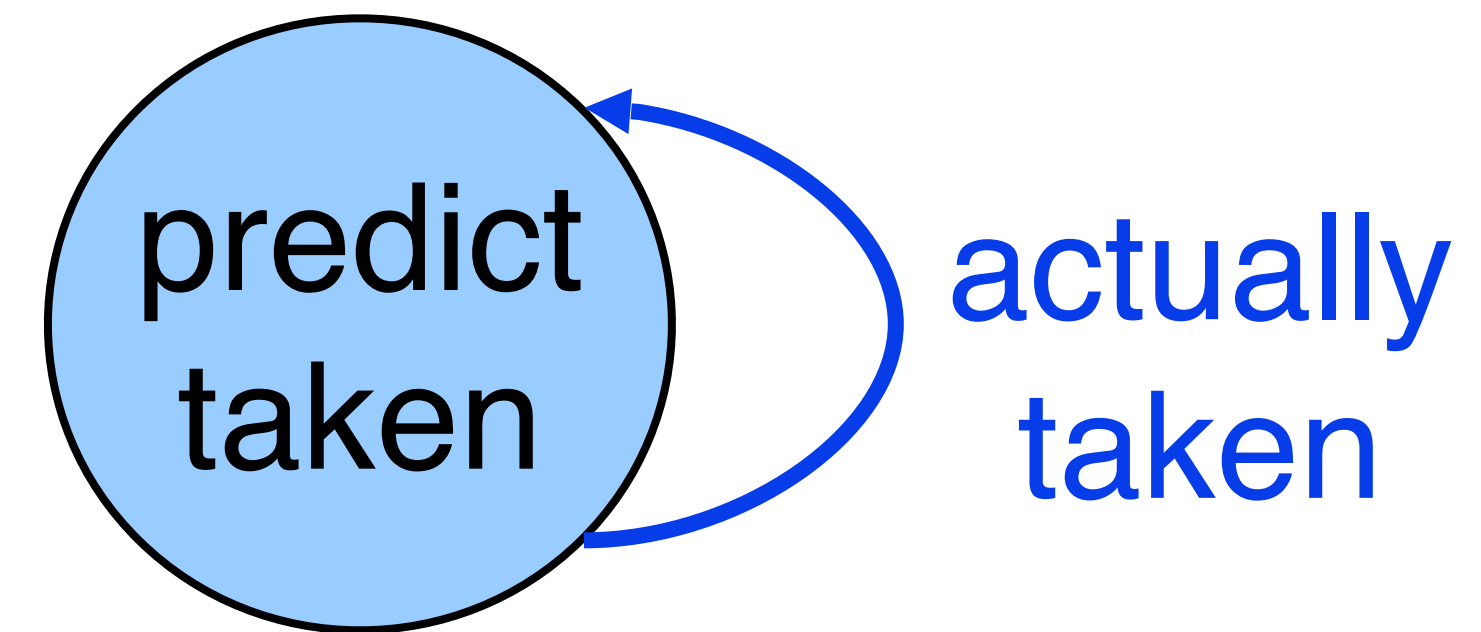
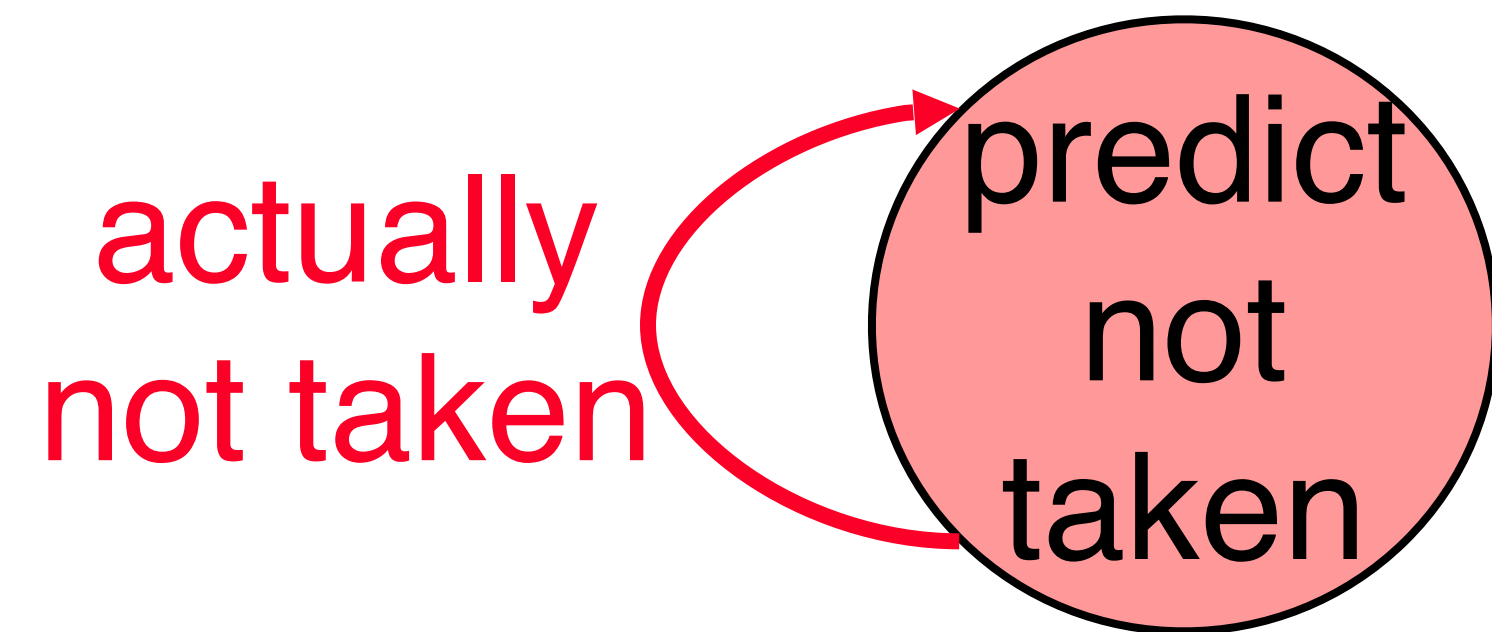
Backward branches (i.e., loop branches) are usually taken

Dynamic Predictors

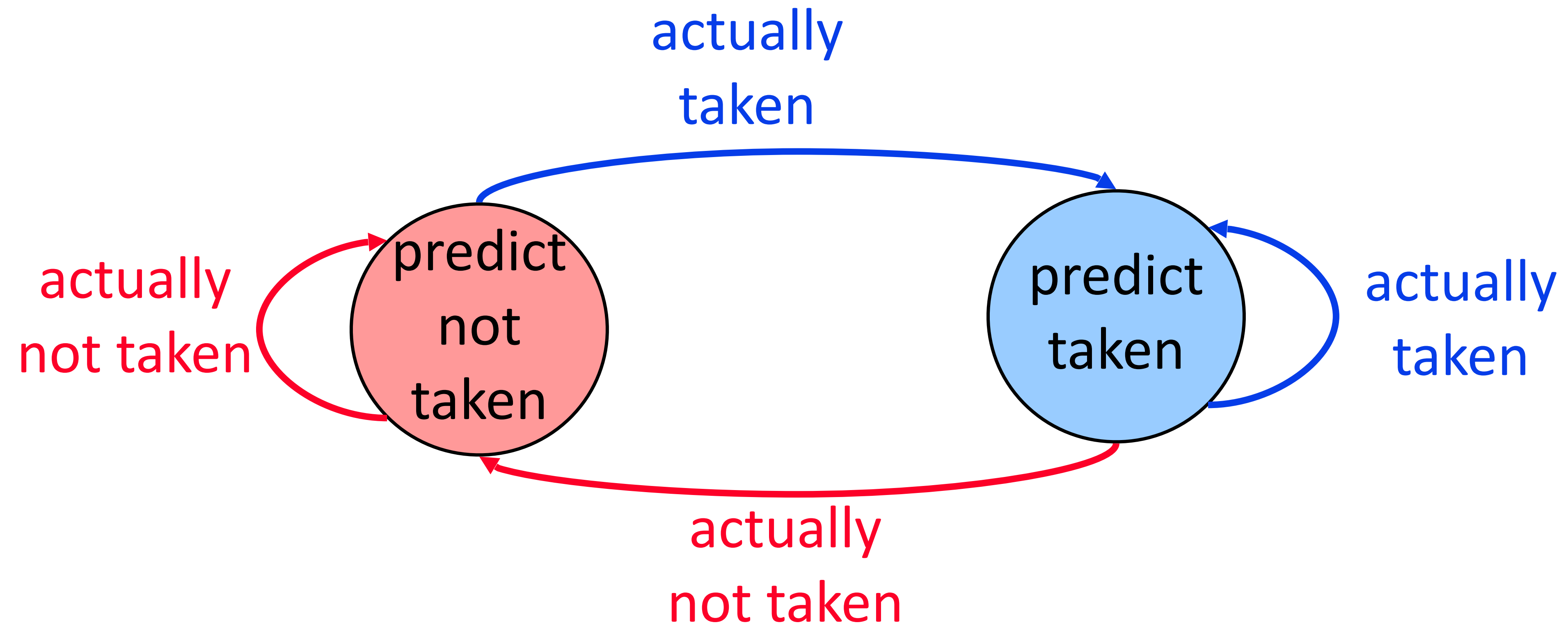
Microarchitectural way of predicting it.

Simple one: Last time predictor

Last-time predictor



Last-time predictor



Implementation

K bits of branch
instruction address



Implementation

K bits of branch
instruction address

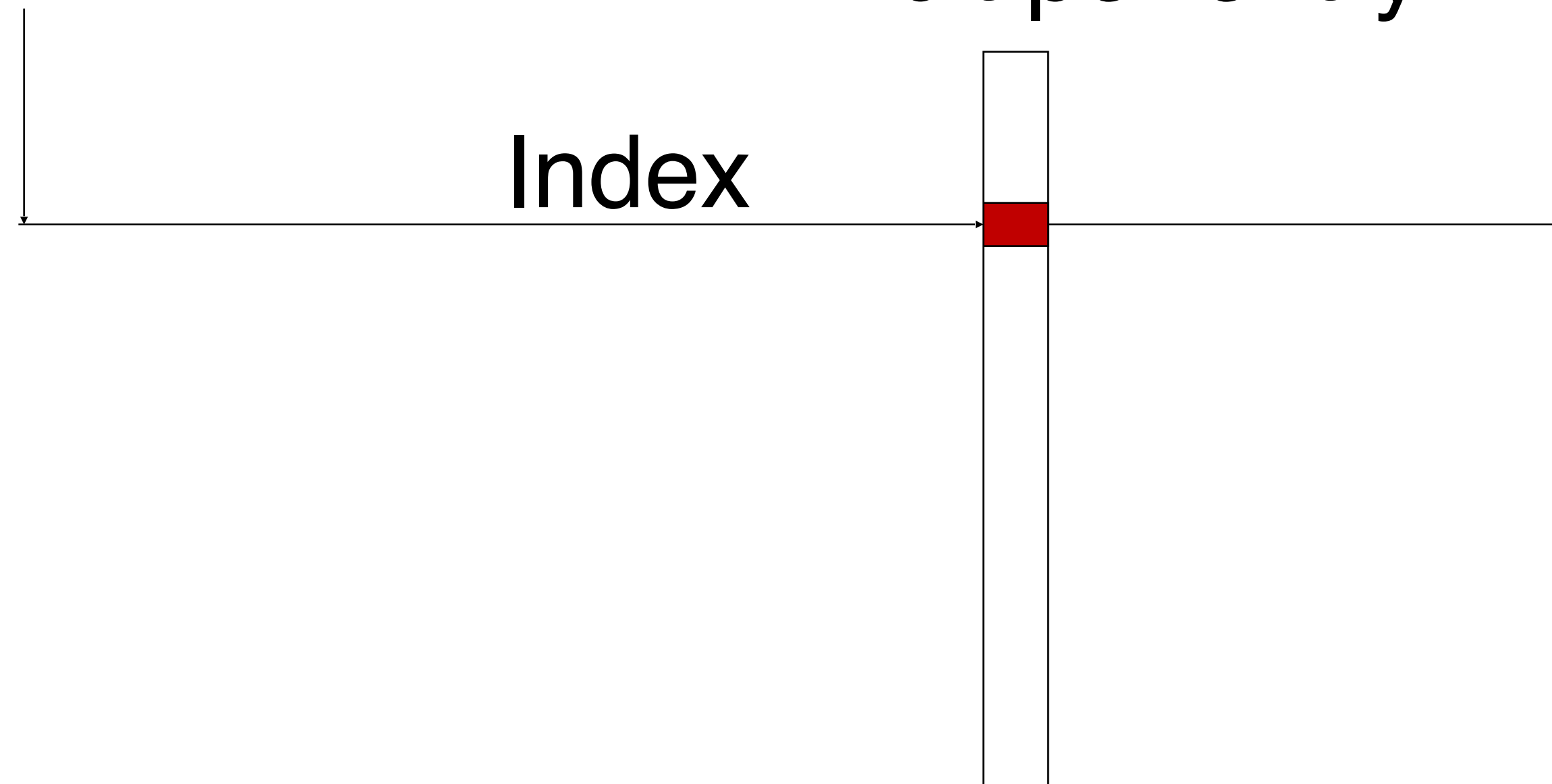
Branch history
table of 2^K entries,
1 bit per entry



Implementation

K bits of branch
instruction address

Branch history
table of 2^K entries,
1 bit per entry



Use this entry to
predict this branch:

0: predict not taken
1: predict taken

Performance of Last-time predictor

TTTTTTTTTTNNNNNNNNNN - 90% accuracy

Always mispredicts the last iteration and the first iteration of a loop branch

Accuracy for a loop with N iterations = $(N-2)/N$

+ Loop branches for loops with large number of iterations

-- Loop branches for loops with small number of iterations

TNTNTNTNTNTNTNTNTN — 0% accuracy

Performance: Calculating the CPI

20% of all instructions are branches, 85% accuracy

Last-time predictor CPI =

$$[1 + (0.20 * \underline{0.15}) * 2] =$$

1.06 (minimum two stalls to resolve a branch)

Types of Branches

	Conditional	Unconditional
Direct	if - then- else for loops (bez, bnez, etc)	procedure calls (jal) goto (j)
Indirect		return (jr)

Why do we need branch prediction?

- Allows useful work to be completed while waiting for a branch to resolve.

- Processors with deep pipelines
 - Intel Core 2 Duo: 14 stages
 - AMD Athlon 64: 12 stages
 - Intel Pentium 4: 31 stages
- Many cycles before branch is resolved
 - Wasting time if wait...
 - Would be good if can do some useful work...

Branch Prediction

- **Key Idea:** Predict branch outcome heuristically.
 - If successful, then we've gained a performance improvement.
 - Otherwise, discard instructions that have been executed speculatively.
 - Program execution state is still correct, all we've done is “waste” a little power.

Branch Prediction Strategies

- **Static:**
 - Decided before runtime
 - Examples:
 - Always-Not Taken
 - Always-Taken
 - Backwards Taken, Forward Not Taken (BTFNT)
- **Dynamic** (aka profile-driven prediction):
 - Prediction decisions may change during the execution of the program

What happens when a branch is mispredicted?

- On a mispredict:
 - No speculative state may commit
 - Squash instructions in the pipeline
 - Cannot allow stores to registers for instructions which would not get to commit
 - Need to handle exceptions appropriately

Direction Based Prediction

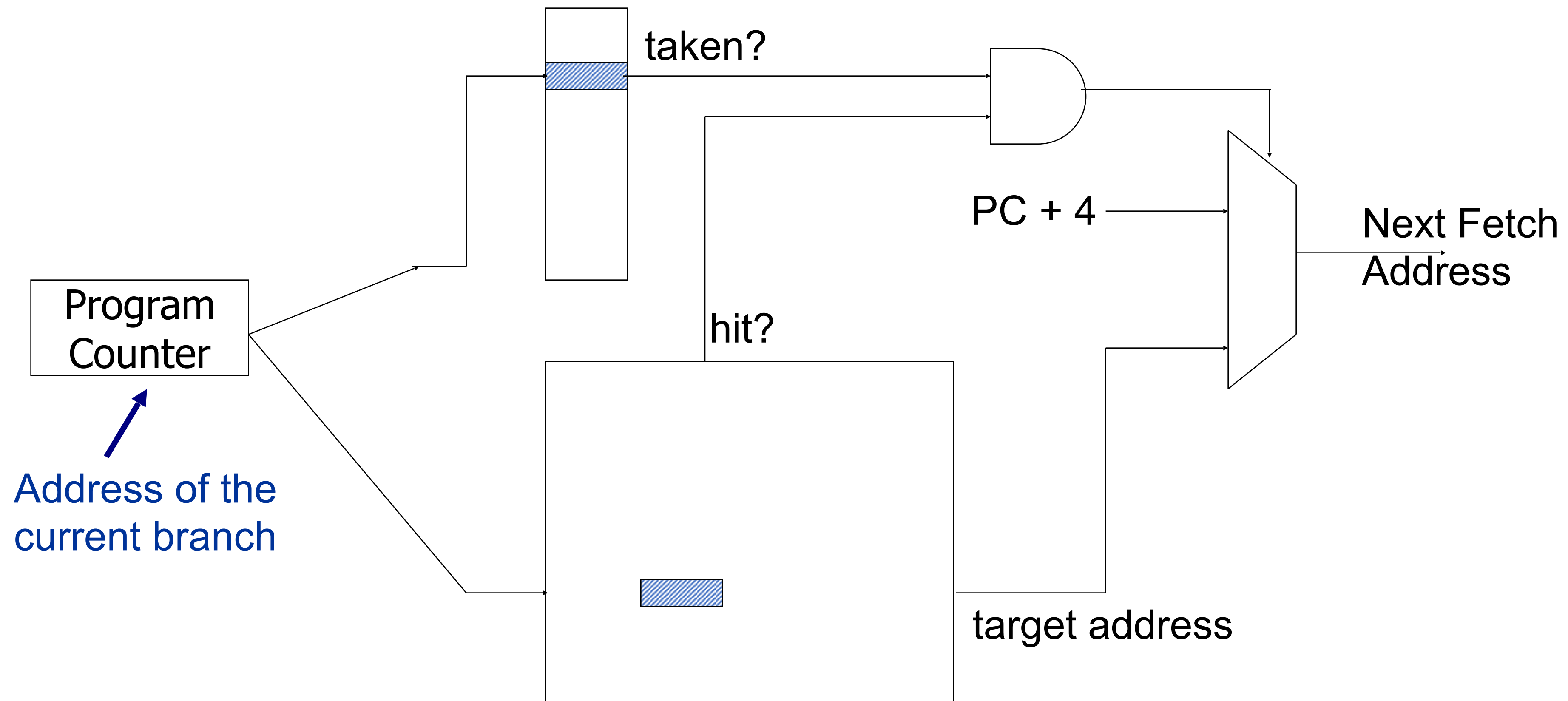
- **Pro:** Simple to implement
 - But, branch behaviour is often variable (dynamic) and depends on how the program is behaving recently.
 - Can't capture such behaviour at compile time with simple direction based prediction!
- **Need history (aka profile)-based prediction.**

Direction-Based Branch Prediction

- Which things exactly to predict?
 - **Direction:**
 - Taken / Not Taken
 - Can only be Direction
 - **Target Address**
 - PC+offset (Taken)/ PC+4 (Not Taken)
 - How implemented?
 - Using Branch Target Address Cache (BTAC) or Branch Target Buffer (BTB)

Branch Predictor: A bit deeper

Direction predictor



Repository of Target Addresses (BTB: Branch Target Buffer)

Example: Branch Penalty Calculation

- Assume a MIPS pipeline using predict taken:
 - 16% of all instructions are branches:
 - 4% unconditional branches: 3 cycle penalty
 - 12% conditional:
 - 50% taken: 3 cycle penalty
 - 50% not taken: 0 cycle penalty

Solution

- For a sequence of N instructions:
 - $3 * 0.04 * N$ delays due to unconditional branches
 - $0.5 * 3 * 0.12 * N$ delays due to conditional taken
- Overall CPI=
 - $1.3 * N$
 - (or 1.3 cycles/instruction)
 - 30% Performance Hit!!!

History-based Branch Prediction

- An important example is **State-based branch prediction**:
- Consists of 2 parts:
 - “**Predictor**” to guess where/if instruction will branch (and to where)
 - “**Recovery Mechanism**”: A way to fix mistakes

History-based Branch Prediction

- **One bit predictor:**
 - Use the outcome from the last time the branch instruction was executed.
- **Problem:**
 - Even if branch is almost always taken, we will be wrong at least twice
 - if branch alternates between taken, not taken
 - **We get 0% accuracy**

Example

- Let initial value = T
- Suppose actual outcome of branches
NT, NT, NT, T, T, T
 - Predictions are: T, NT, NT, NT, T, T
 - 2 wrong (in red), 4 correct = 66% accuracy
- 2-bit predictors can do better
- In general, can have k-bit predictors.

1-bit Predictor: Exercise

- Program assumptions:
 - 23% loads and in $\frac{1}{2}$ of cases, next instruction uses load value
 - 13% stores
 - 19% conditional branches
 - 2% unconditional branches
 - 43% other

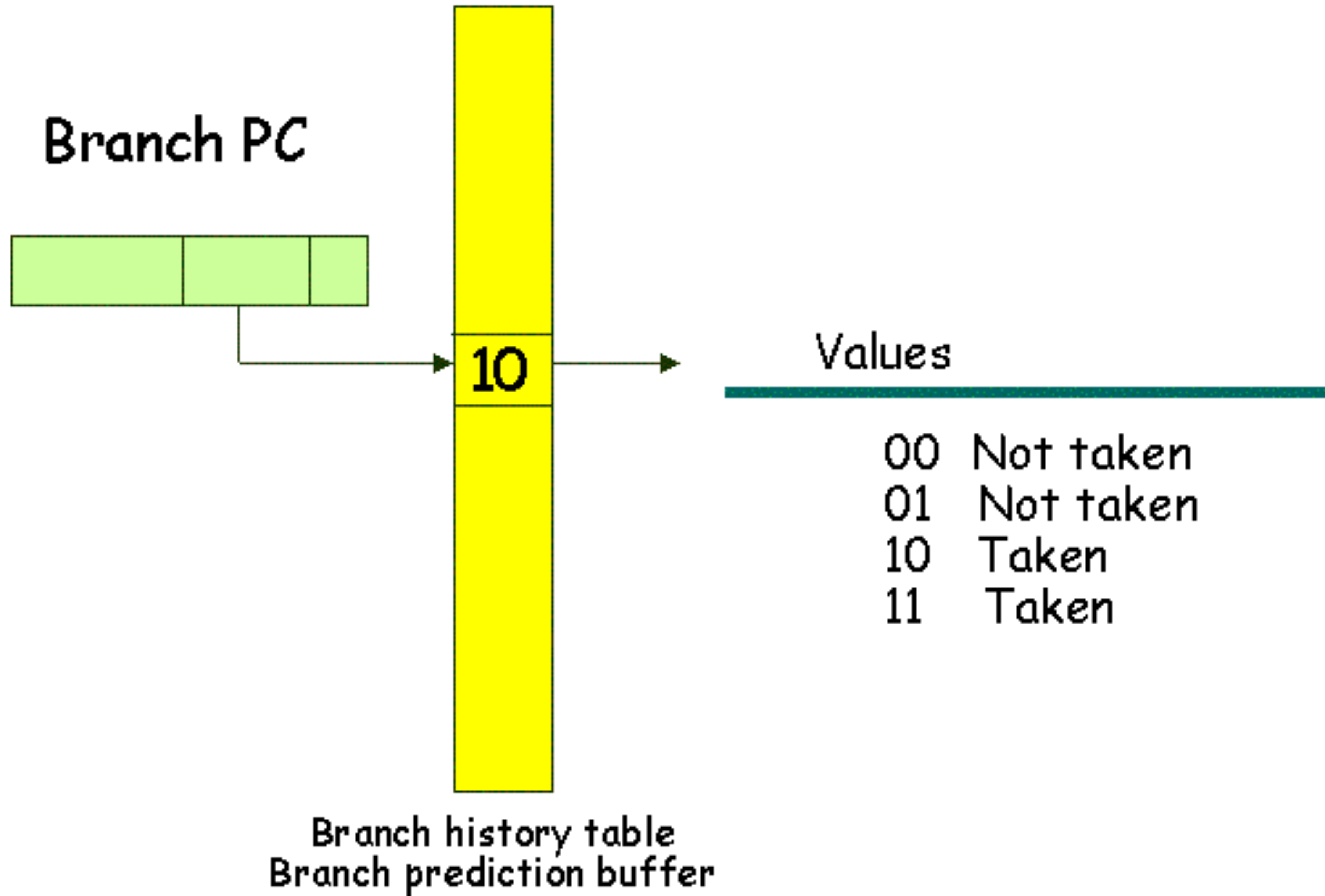
Exercise

- Machine Assumptions:
 - 5 stage pipe
 - Penalty of 1 cycle on use of load value immediately after a load.
 - Jumps are resolved in ID stage and incur a 1 cycle branch penalty.
 - 75% branch prediction accuracy and 1 cycle delay (penalty) on misprediction.

Solution

- CPI penalty calculation:
 - Loads:
 - 50% of the 23% of loads have 1 cycle penalty: $0.5 * .23 = 0.115$
 - Jumps:
 - All 2% of jumps have 1 cycle penalty: $0.02 * 1 = 0.02$
 - Conditional Branches:
 - 25% of the 19% are mispredicted, have a 1 cycle penalty:
 $0.25 * 0.19 * 1 = 0.0475$
- Total Penalty: $0.115 + 0.02 + 0.0475 = 0.1825$
- Average CPI: $1 + 0.1825 = 1.1825$

2-bit branch prediction

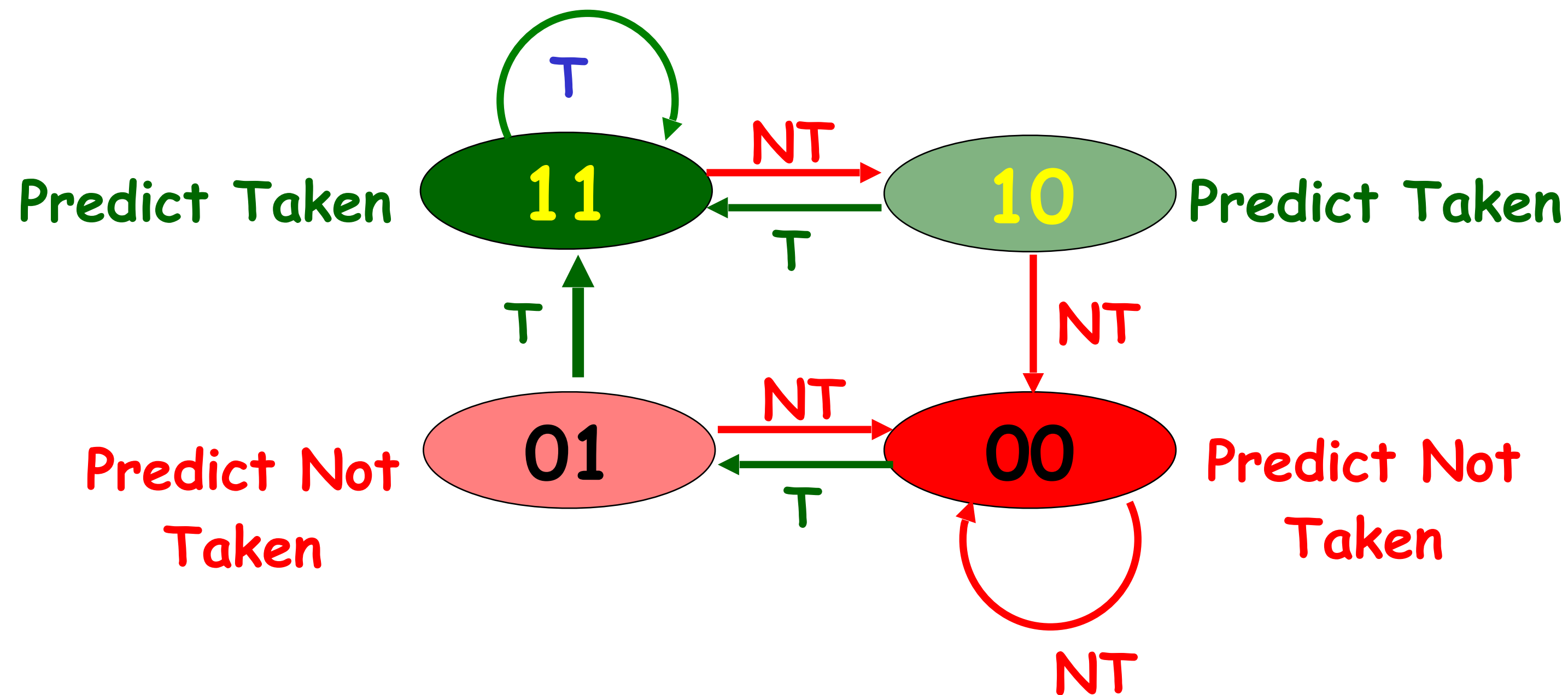


2-Bit Branch Prediction

- Approach: Prediction is changed only if mispredicted **twice**
- Adds **hysteresis** to decision making process

Red: stop, not taken

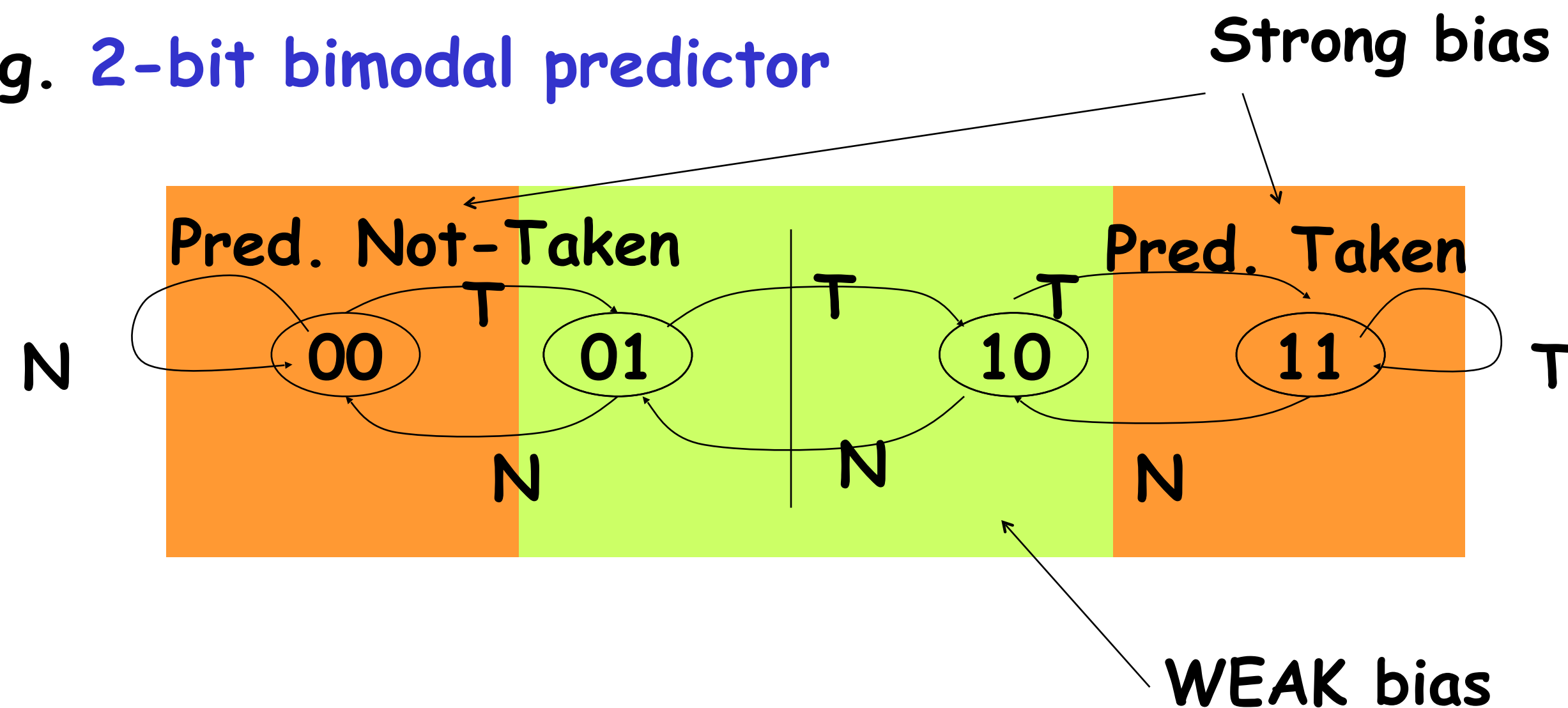
Green: go, taken



AKA Saturation Counter Predictor

- Observation: branches highly **bimodal**
- n-bit saturation counter
 - Hysteresis
 - n-bit entries in branch prediction table

e.g. 2-bit bimodal predictor



n-bit Saturating Counter

- Values: $0 \sim 2^n - 1$
- When the counter is **greater than** or equal to one-half of its maximum value,
 - the branch is predicted as **taken**. Otherwise, not taken.
- Studies have shown that the 2-bit predictors do almost as well

2-bit Predictor

- What is the prediction accuracy using a 4096 entry 2-bit branch predictor for a typical application?
 - 99% to 80% depending upon the application.
- Can an n-bit ($n > 2$) predictor do better?
 - Not really! 2-bit predictors do almost as well as any n-bit predictors.
- How can then the accuracy of branch prediction be improved?
 - **Correlating branch predictor.**

Predictors in Simple Pipelines

- Initial pipelined processors, e.g. MIPS, SOLARIS, etc.:
 - Did only trivial branch predictions.
- Possible reasons could be:
 - The penalty of mispredictions not as severe as in deeper pipelined processors.
 - Sophisticated branch predictors did not exist.
- Advanced branch prediction techniques have now become very important:
 - With the use of deeper pipelines.
 - Introduction of superscalar processors.

Improving Accuracy of Branch Predictors

- It may be possible to improve the accuracy of branch prediction:
 - By observing the recent behavior of other branches.
- **Example:**

```
if (a==2){  
    b=2;}  
  
if(b==2){  
    b=0;}
```

The Main Idea

- **Record m most recently executed branches as taken or not taken,**
 - **Use that pattern to select the proper n-bit branch history table (BHT).**

Main Idea

- An (m,k) predictor:
 - Makes use of the outcomes observed for the last m branches:
 - Uses m number of k -bit predictors.
 - Behavior of a branch can be predicted by choosing from 2^m branch predictors.

Local and global history

- Local Behavior

What is the predicted direction of Branch A given the outcomes of previous instances of Branch A ?

- Global Behavior

What is the predicted direction of Branch Z given the outcomes of *all** previous branches A, B, ..., X and Y?

Number of previous branches tracked limited by the history length

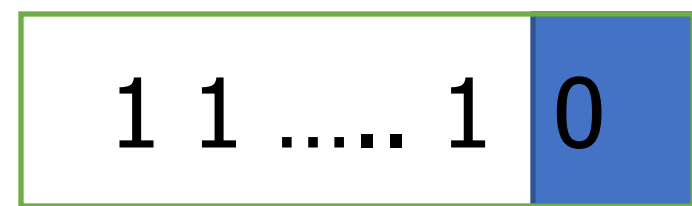
Two Level Branch Predictors

First level: **Global branch history register** (N bits)

The direction of last N branches

Second level: **Table of saturating counters** for each history entry

The direction the branch took the last time the same history was seen



GHR
(global history register)

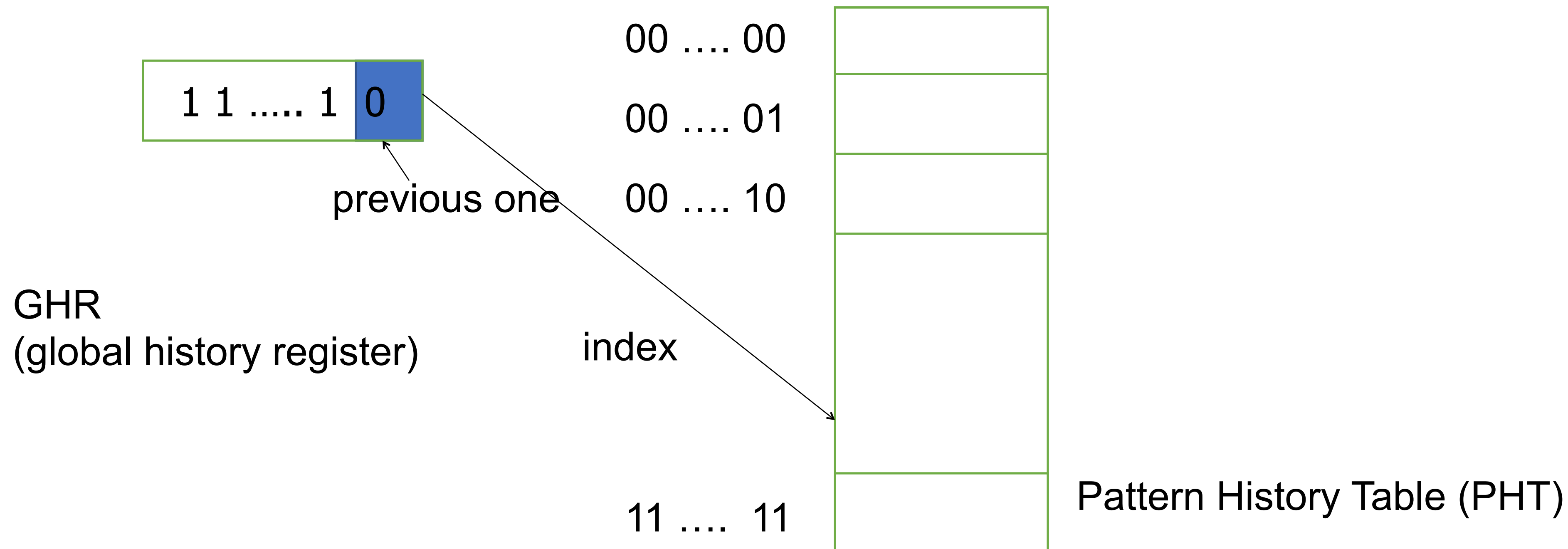
Two Level Branch Predictors

First level: **Global branch history register** (N bits)

The direction of last N branches

Second level: **Table of saturating counters for each history entry**

The direction the branch took the last time the same history was seen



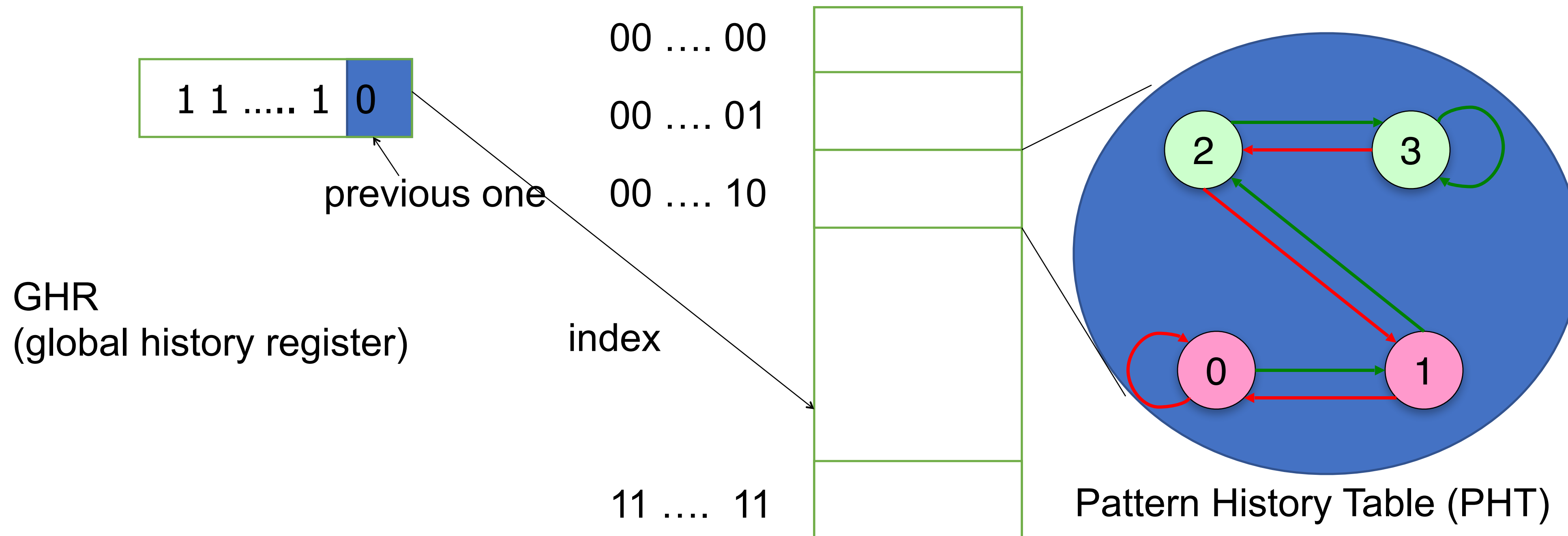
Two Level Branch Predictors

First level: **Global branch history register** (N bits)

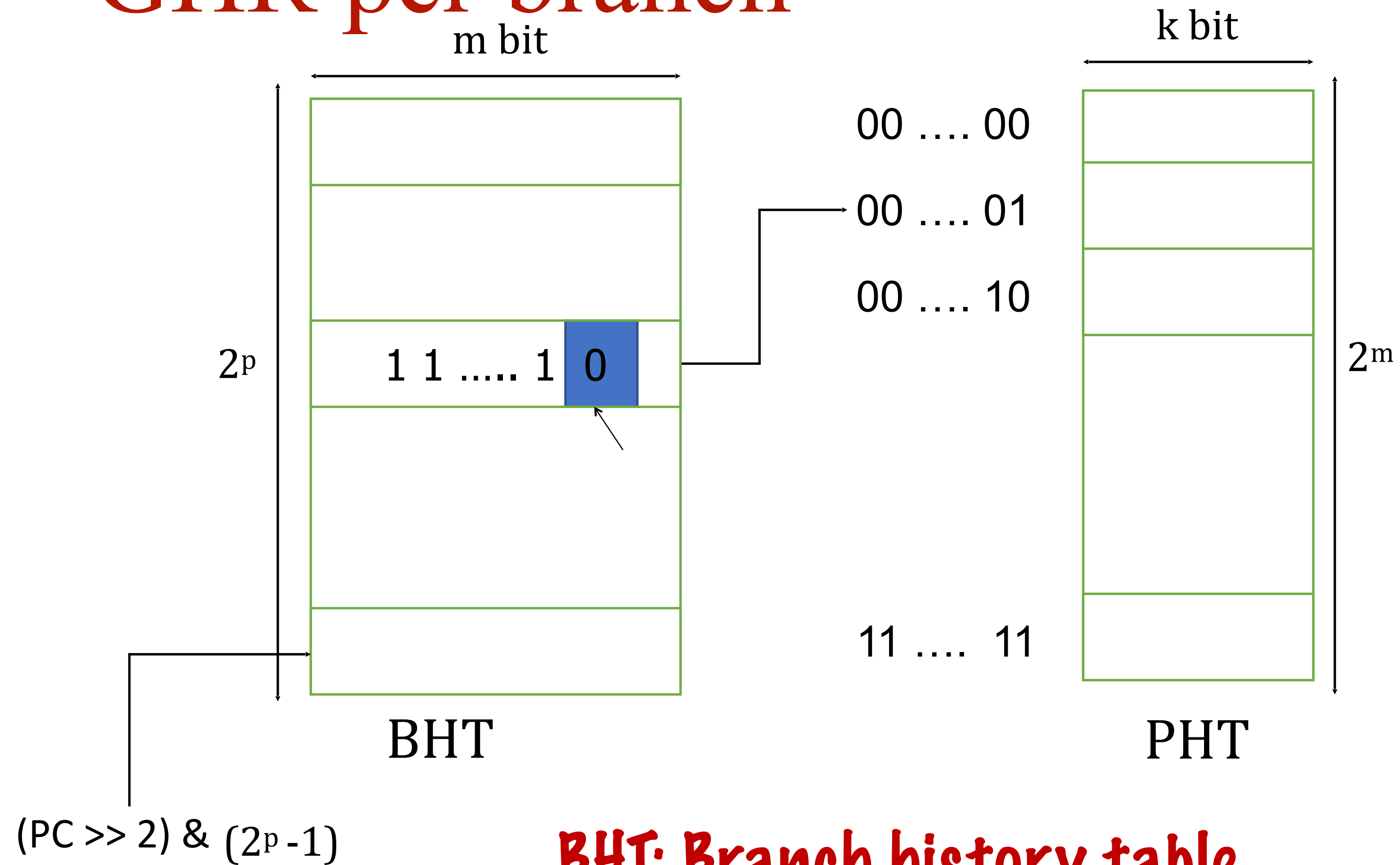
The direction of last N branches

Second level: **Table of saturating counters for each history entry**

The direction the branch took the last time the same history was seen



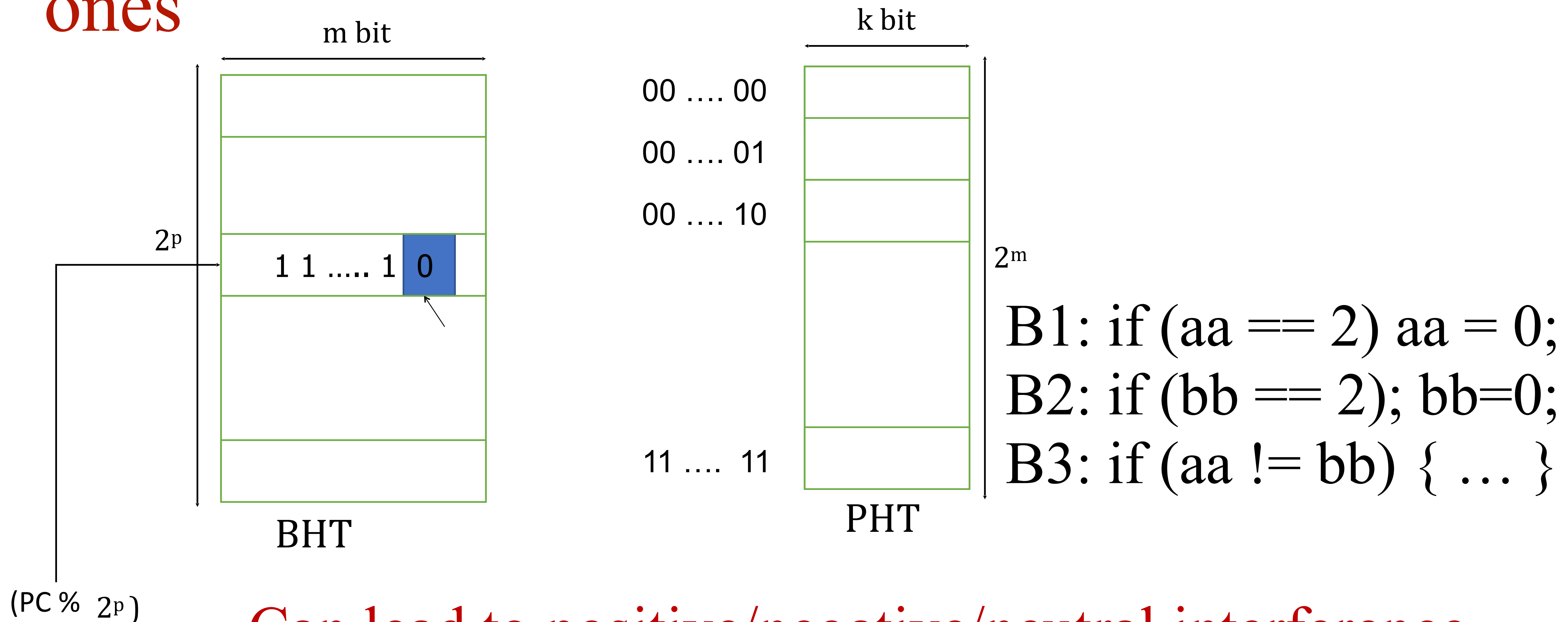
GHR per branch



BHT: Branch history table

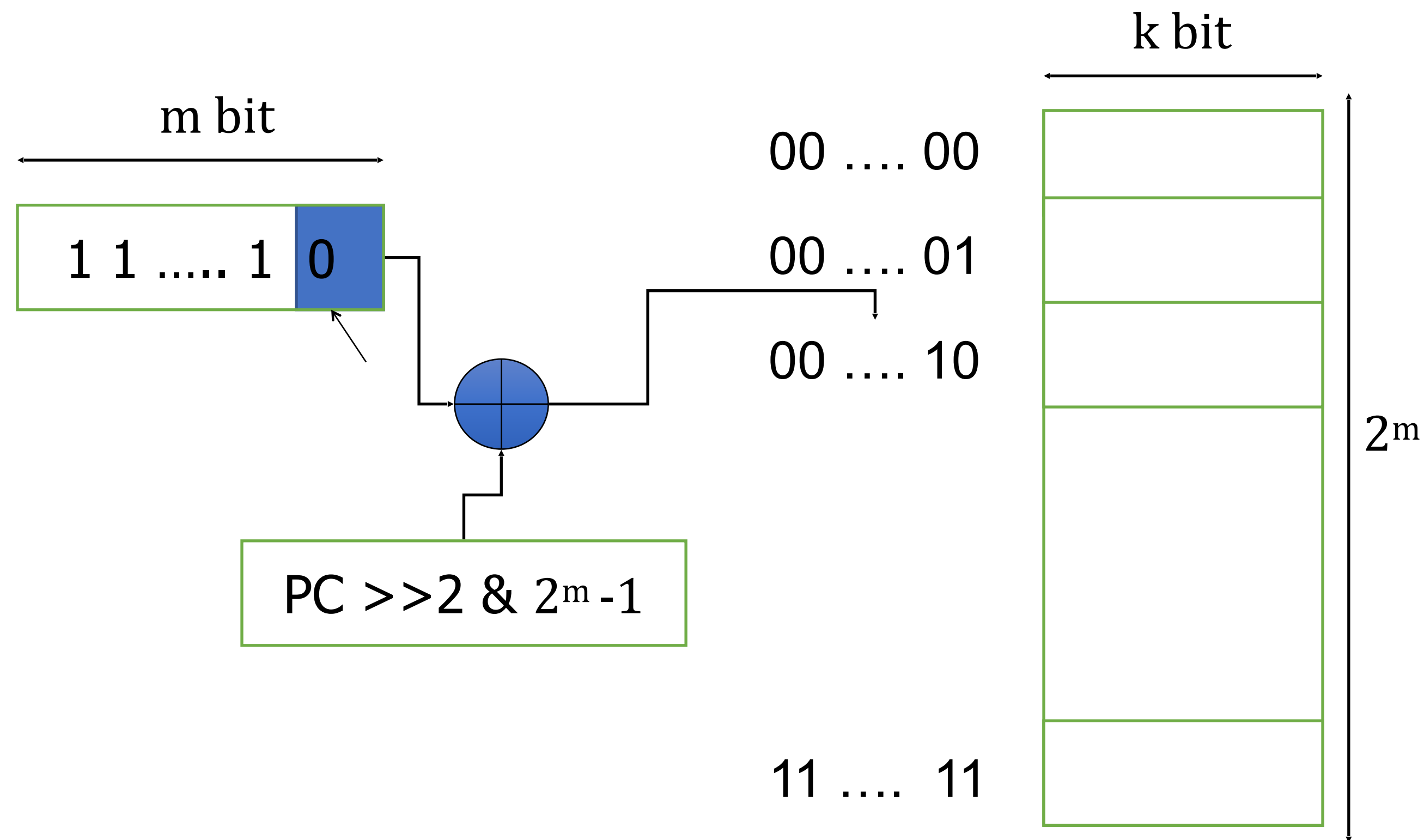
Mostly $K=2$, $m=12$ for example

Set of branches: One register for correlated ones



Can lead to positive/negative/neutral interference

Gshare is the answer



For a given history and for a given branch (PC) counters are trained

Few Important Points

Branch prediction happens at the IF stage.

We know the target outcome at the end of EX stage.

So BHT and PHT will be updated after EX stage for the corresponding PC. Any issues here?

Issue

I1 F D E

I2 F D E

I3 F D E

Lets assume I1 and I3 are branch instructions. I1 will update BHT and PHT in E stage, and I3 will probe BHT and PHT in F stage. To make sure PHT is updated correctly with the correct BHT entry, BHT entry is communicated till the E stage.

State-of-the-art

State of the art: Neural vs. TAGE

1970: Flynn

1972: Riseman/Foster

1979: Smith Predictor

1991: Two-level prediction

1993: gshare, tournament

1996: Confidence estimation

1996: Vary history length

1998: Cache exceptions

2001: Neural predictor

2004: PPM

2006: TAGE

2016: Still TAGE vs Neural

- Neural: AMD, Samsung

- TAGE: Intel?, ARM?

Similarity

- Many sources or “features”

- Key difference: how to combine them

- TAGE: Override via partial match

- Neural: integrate + threshold

- Every CBP is a cage match

- Andre Seznec vs. Daniel Jimenez



BTB (Target Address Predictor)

Address of branch instruction

0b0110[...]01001000

Branch instruction

BNEZ R1 Loop

Branch Target Buffer (BTB)

30-bit address tag

target address

0b0110[...]0010	PC + 4 + Loop

Branch
History Table
(BHT)

2 state bits

BTB is probed in the fetch stage along with the direction predictor. A hit in the BTB means the PC is a branch PC.

Branch Target Buffer

❑ For BTB to make a correct prediction, we need:

- **BTB hit:** the branch instruction should be in the BTB
- **Prediction hit:** the prediction should be correct
- **Target match:** the target address must not be changed from the last time

❑ **Example:** BTB hit ratio of 96%, 97% prediction hit, 1.2% of target change,
The overall prediction accuracy = $0.96 * 0.97 * 0.988 = 92\%$

Branch Instruction Address	Branch Prediction Statistics	Branch Target Address
.	.	.
.	.	.
.	.	.

Book

Textbook reading: P & H, Chapter 4