

# Lecture 8

# Pipeline Hazards

## Control Hazard



# Summary of Data Hazard

## ■ Taxonomy of Hazards

- Structural hazards
  - These are conflicts over hardware resources.
- Data hazards
  - Instruction depends on result of prior computation which is not ready (computed or stored) yet
  - OK, we did these, Double Bump, Forwarding path, compiler scheduling, otherwise have to stall
- Control hazards
  - branch condition and the branch PC are not available in time to fetch an instruction on the next clock

# Pipelining Hazards

## ■ Taxonomy of Hazards

### - Structural hazards

- These are conflicts over hardware resources.

### - Data hazards

- Instruction depends on result of prior computation which is not ready (computed or stored) yet
- OK, we did these, Double Bump, Forwarding path, software scheduling, otherwise have to stall

### - Control hazards

- branch condition and the branch PC are not available in time to fetch an instruction on the next clock

# The Control hazard

## ■ Cause

- branch condition and the branch PC are not available in time to fetch an instruction on the next clock
- The next PC takes time to compute
- For conditional branches, the branch direction takes time to compute.

## ■ Control hazards can cause a greater greater performance loss for MIPS pipeline than do data hazards.

# Example: Branches

Address	Instruction
---------	-------------

36	NOP
----	-----

40	ADD R30, R30, R30
----	-------------------

44	BEQ R1, R3, 24
----	----------------

 <- this branches to address 72

48	AND R12, R2, R5
----	-----------------

52	OR R13, R6, R2
----	----------------

56	ADD R14, R2, R2
----	-----------------

60	...
----	-----

64	...
----	-----

68	
----	--

72	LW R4, 50(R7)
----	---------------

76	...
----	-----

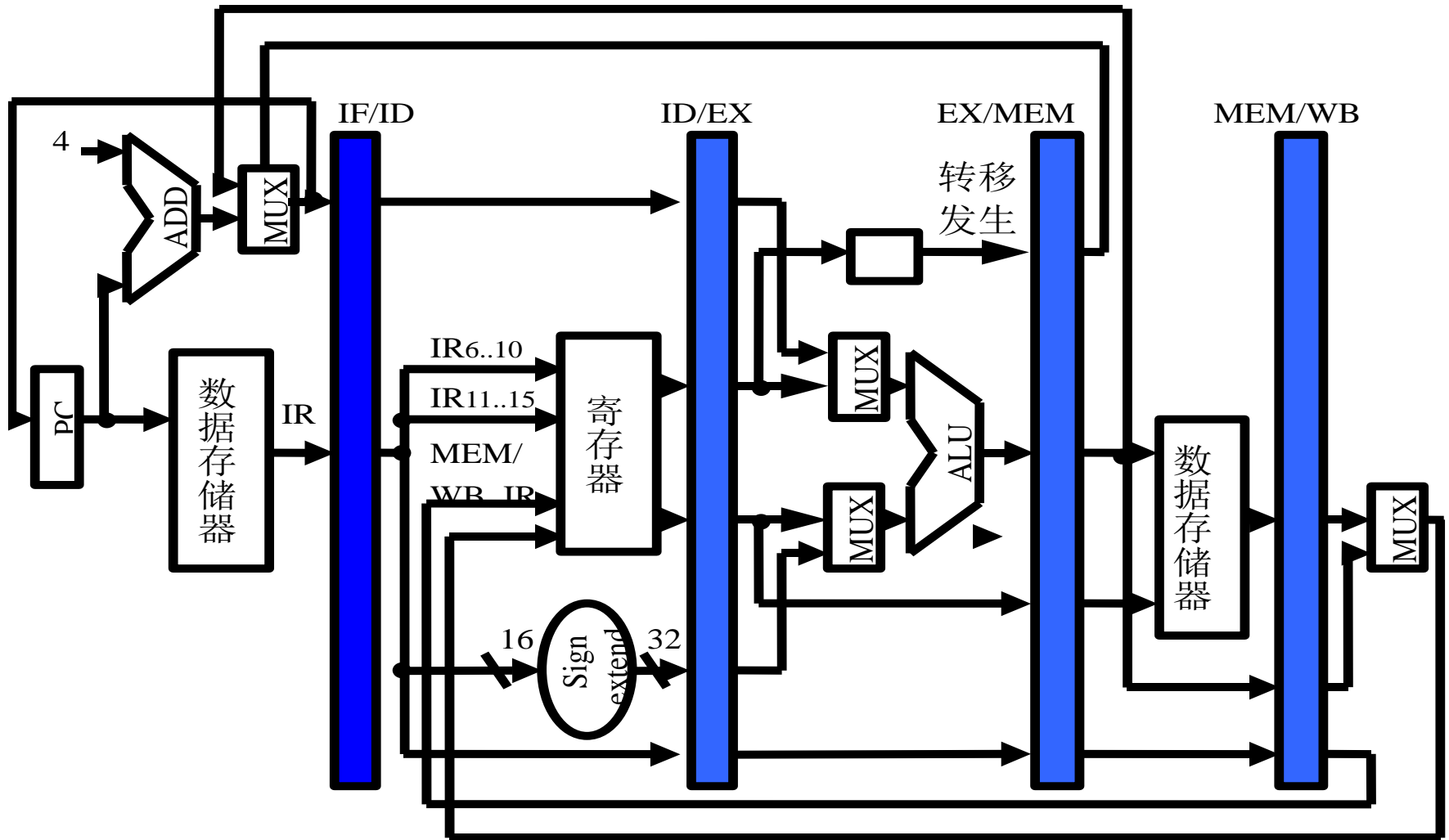
We execute all these if  $R1 \neq R3$

We execute just these if  $R1 == R3$

Flow of instructions if branch is taken: 36, 40, 44, 72, ...

Flow of instructions if branch is not taken: 36, 40, 44, 48, ...

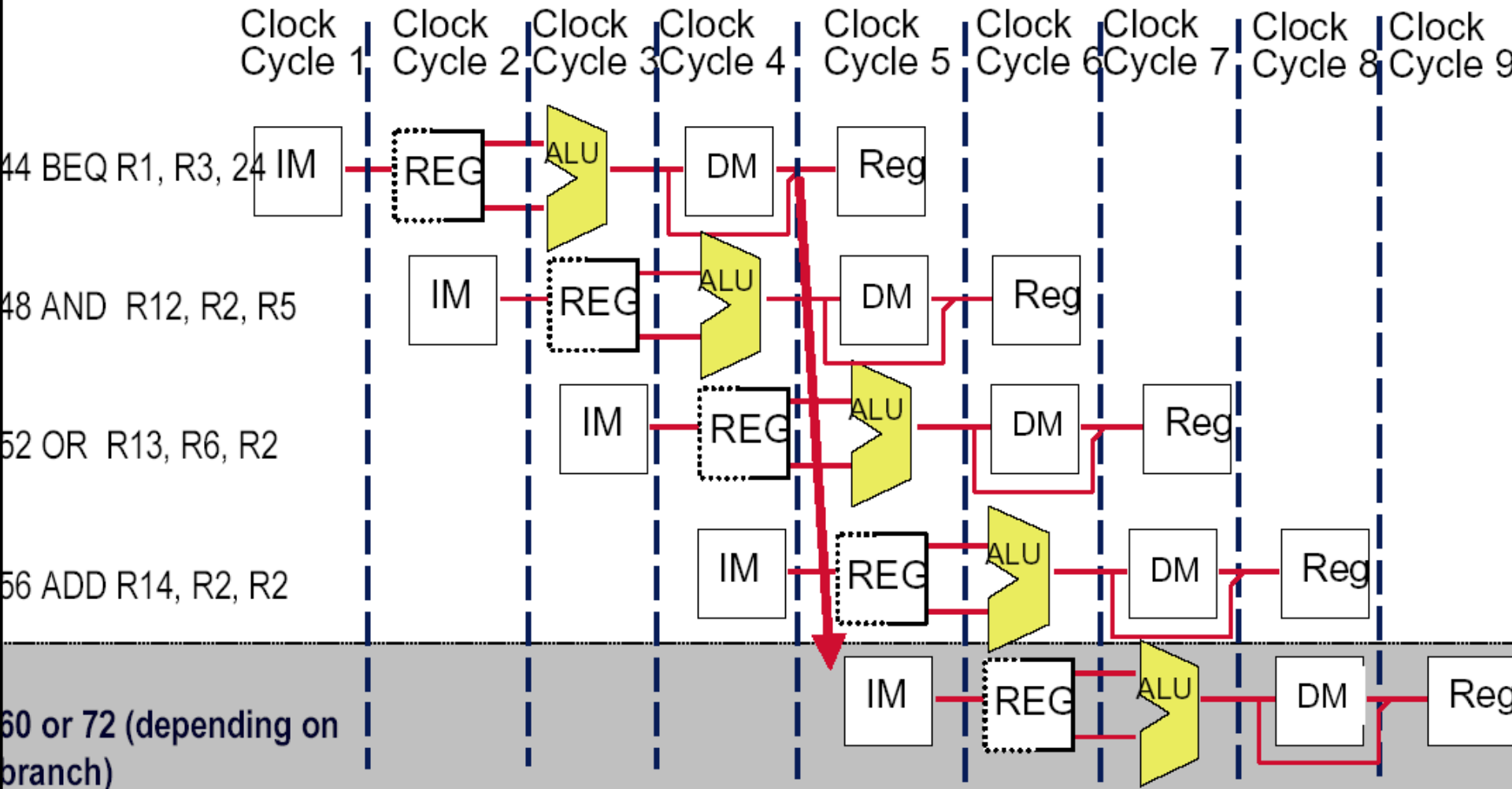
# Recall: Basic Pipelined Datapath



# Control hazard

Flow of instructions if branch is taken: 36, 40, 44, 72, ...

Flow of instructions if branch is not taken: 36, 40, 44, 48, ...



# Dealing with the control hazard

## ■ Four simple solutions

- Freeze or flush the pipeline
- Predict-not-taken (Predict-untaken)
  - Treat every branch as not taken
- Predict-taken
  - Treat every branch as taken
- Delayed branch

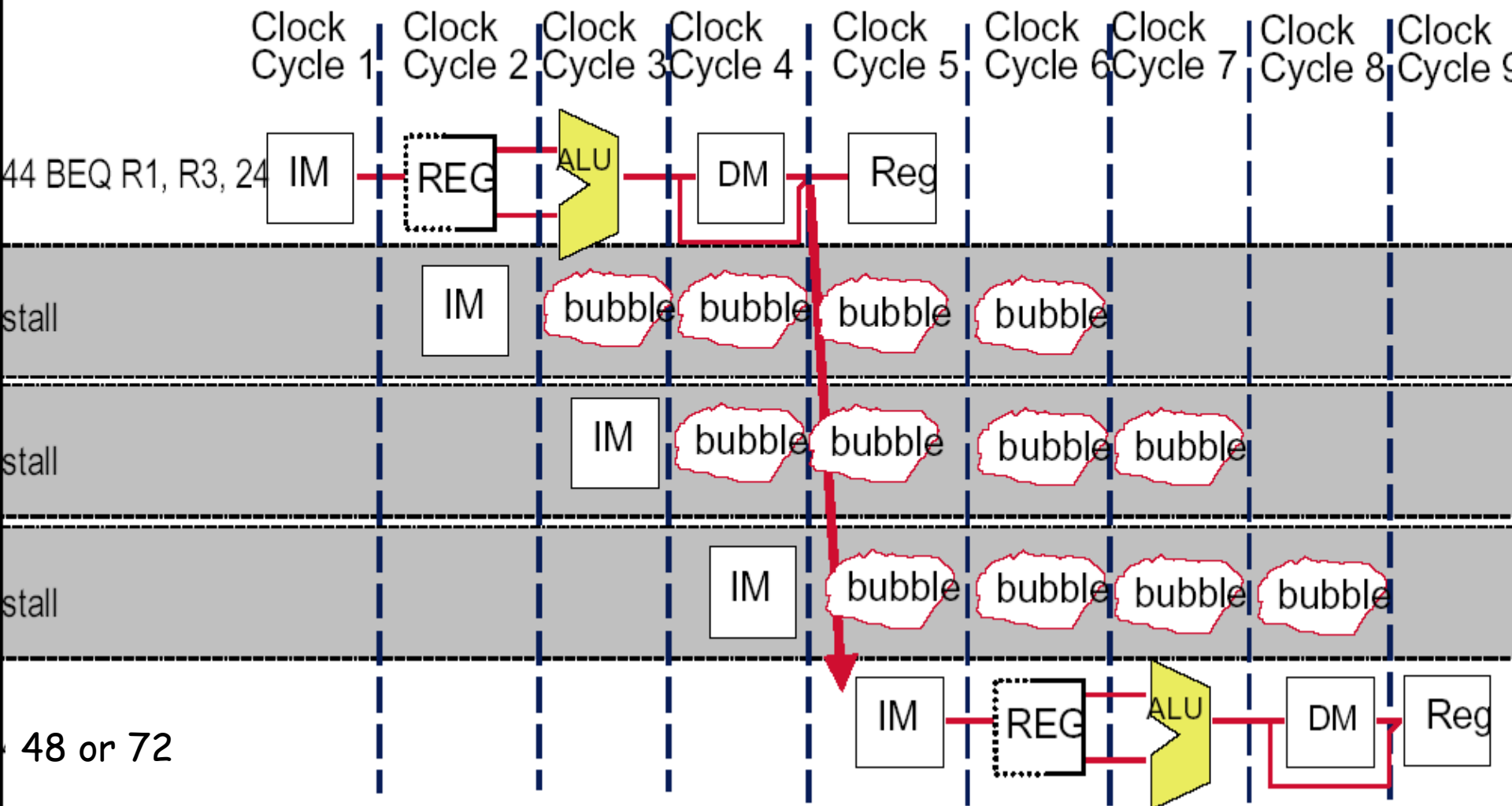
## ■ Note:

- Fixed hardware
- Compile time scheme using knowledge of hardware scheme and of branch behavior



# Recall: solve the hazard by inserting stalls

Flow of instructions if branch is not taken: 36, 40, 44, 48, ...



# The pipeline status

Branch instruction	IF	ID	EX	MEM	WB		
Branch Successor		IF	stall	stall	idle	idle	
Branch successor+1					IF	ID	EX
Branch successor+2						IF	ID
Branch successor+3							IF

# Flushing the pipeline

- Simplest hardware:
  - Holding or deleting any instruction after branch until the branch destination is know.
  - Penalty is fixed.
  - Can not be reduced by software.

# Stalls greatly hurt the performance

## ■ Problem:

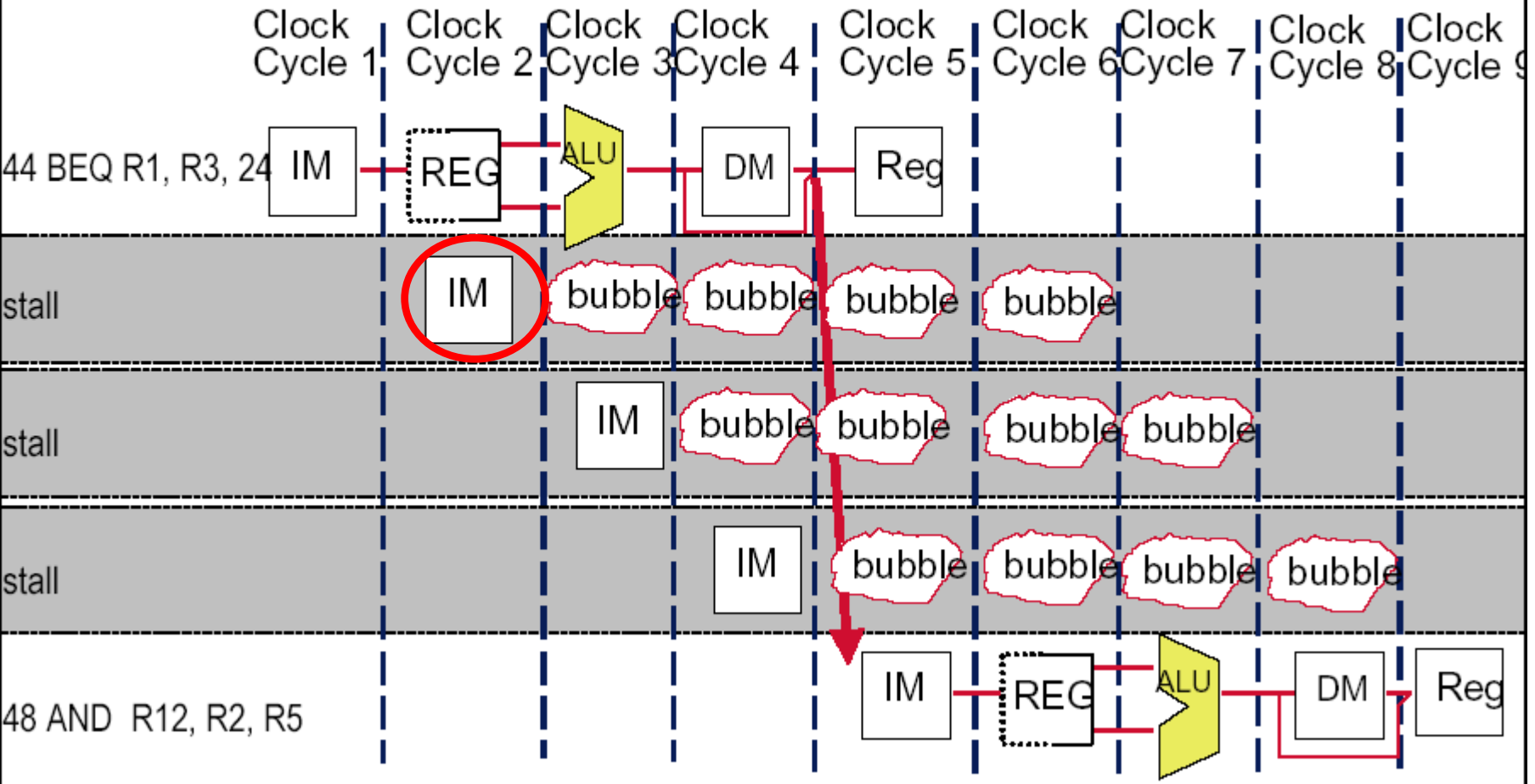
- With a 30% branch frequency and an ideal CPI of 1, how much the performance is by inserting stalls ?

## ■ Answer:

- $CPI = 1 + 30\% \times 3 = 1.9$
- this simple solution achieves only about **half** of the ideal performance.

# Always Stall Hurts the Not- taken case

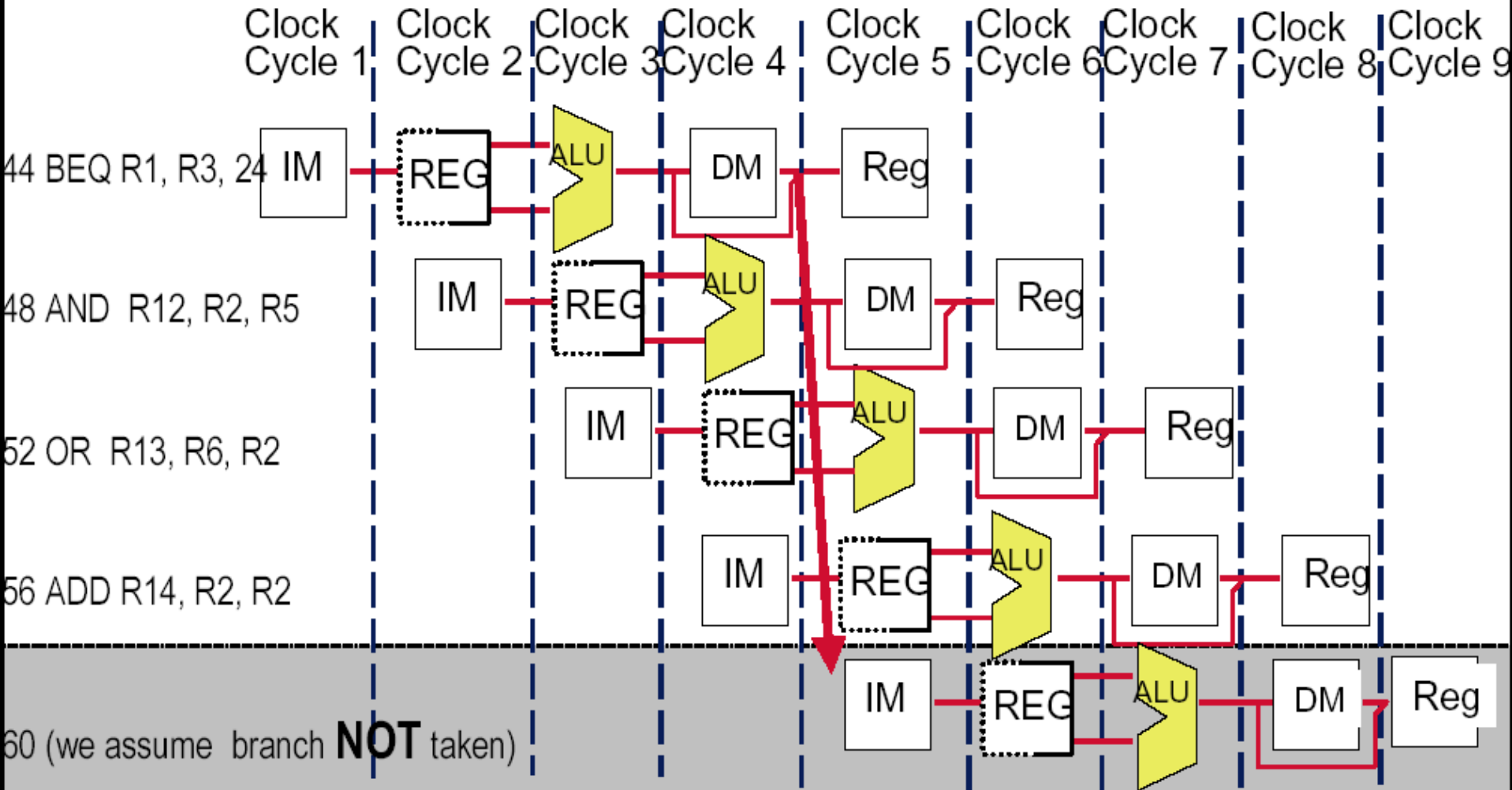
Flow of instructions if branch is not taken: 36, 40, 44, 48, ...



# How about assume Branch Not Taken

Flow of instructions if branch is taken: 36, 40, 44, 72, ...

Flow of instructions if branch is not taken: 36, 40, 44, 48, ...



# What If Branch Was Taken...?

...i.e., what if we guessed **wrong** on the branch?

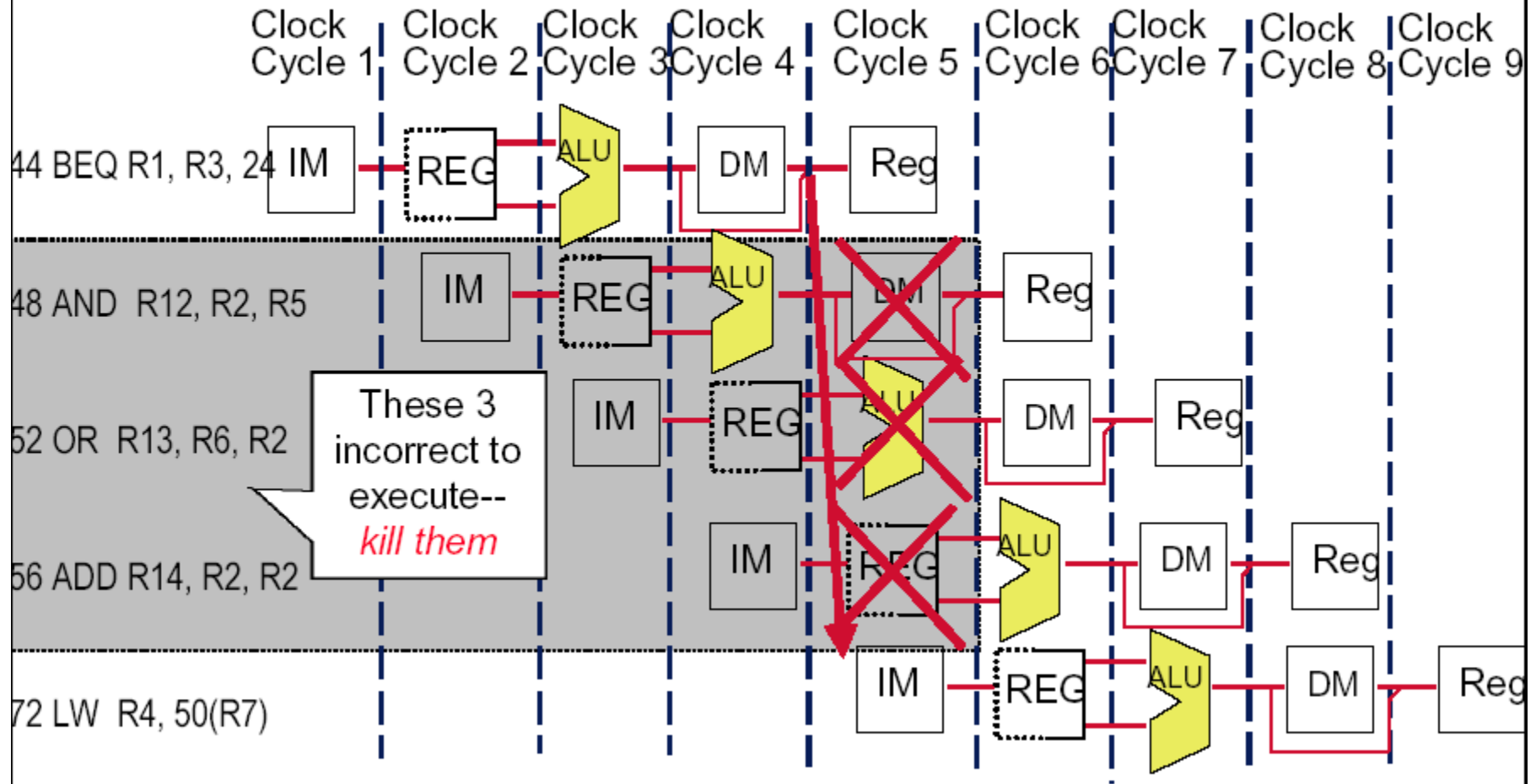
Address	Instruction	
36	NOP	
40	ADD R30, R30, R30	
44	BEQ R1, R3, 24	<- this branches to address 72
48	AND R12, R2, R5	} We already <b>started</b> some of these since we assumed NO branch taken
52	OR R13, R6, R2	
56	ADD R14, R2, R2	
60	...	
64	...	
68	...	
72	LW R4, 50(R7)	} But a few clock cycles later, We figure out these are right Instructions to go next
76	...	

Flow of instructions if branch is taken: 36, 40, 44, 72, ...

Flow of instructions if branch is not taken: 36, 40, 44, 48, ...

# How to do with the branch taken ?

Flow of instructions if branch is taken: 36, 40, 44, 72, ...





# Predict -not-taken

## ■ Hardware:

- Treat every branch as not taken (or as the formal instruction)
  - When branch is not taken, the fetched instruction just continues to flow on. **No stall** at all.
  - If the branch is taken, then restart the fetch at the branch target, which cause **3** stall.(**should turn the fetched instruction into a no-op**)
  - $\text{Perf} = 1 + \text{br}\% * \text{take}\% * 3$

## ■ Compiler:

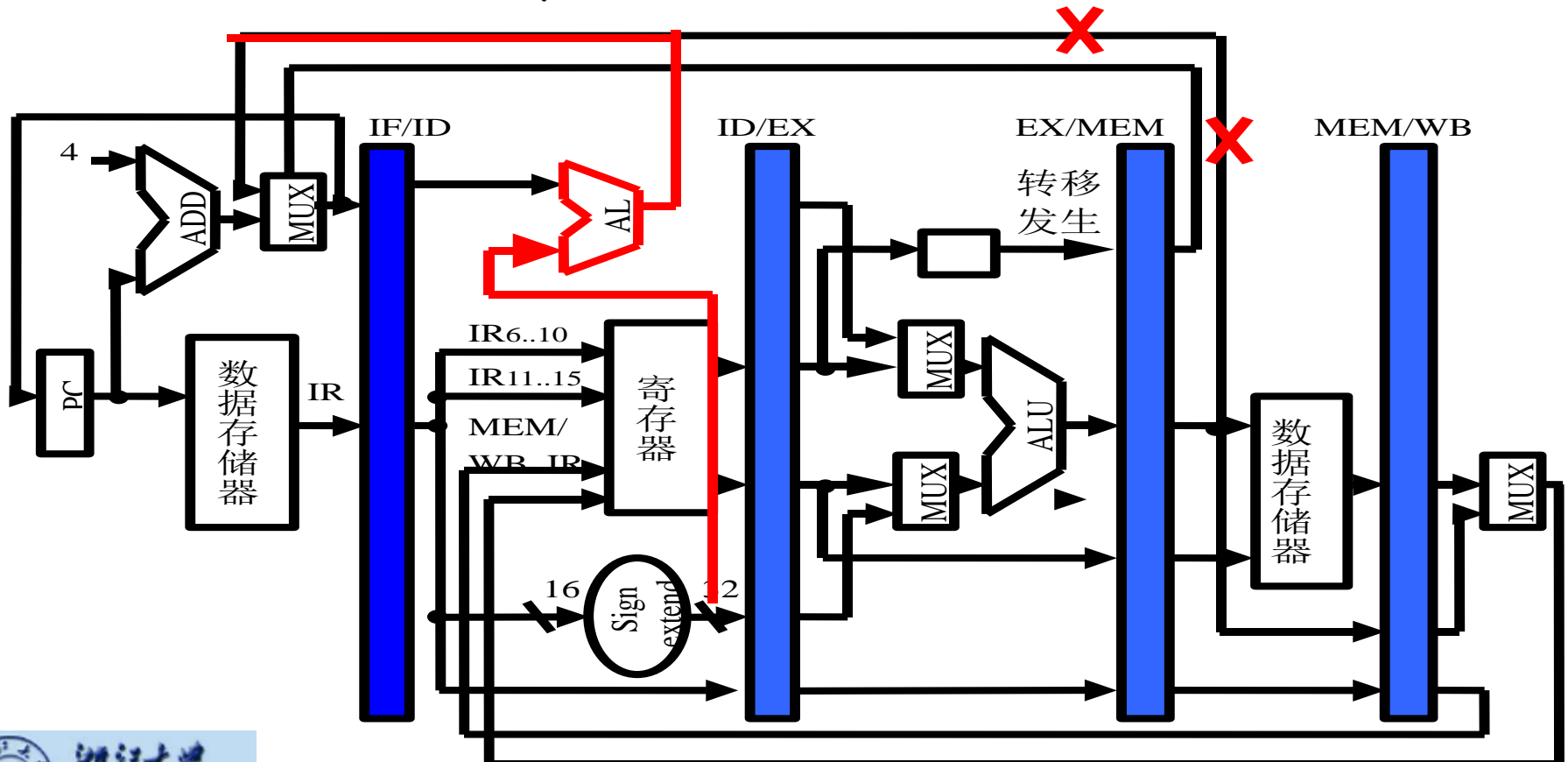
- Can improve the performance by coding the most frequent case in the untaken path.

# Which code segment is better ?

- If     `cond=(malloc(..))`
  - Then   `statements_taken`
  - Else    `statements_not_taken`
- 
- If     `not cond`
  - Then    `statements_not_taken`
  - Else    `statements_taken`

# Alternative is assuming the branch always taken

- Most branches(60%) are taken, so we should make the taken branch more faster. Why not try assuming the branch always taken?



# Predict -taken

## ■ Hardware

- Treat every branch as taken (evidence: more than 60% branches are taken)
- As soon as the branch target address is computed, assume the branch to be taken and begin fetching and executing at the target.
- Only useful when the target is known before the branch outcome.
- No advantage at all for MIPS 5-stage pipeline.

## ■ Compiler

- Can improve the performance by coding the most frequent case in the taken path.

# Pipeline status for predict-taken

Branch is **taken**:  $1 \text{ stall perf} = 1 + \text{Br}\% * (\text{taken}\% * 1 + \text{untaken}\% * 3)$

44 BEQ R1, 24	IF	ID	EX	MEM	WB		
48 AND R12, R2, R5		IF	idle	idle	idle	idle	
72 LW R4, 50(R7)			IF	ID	EX	MEM	WB
76				IF	ID	EX	MEM
80					IF	ID	EX

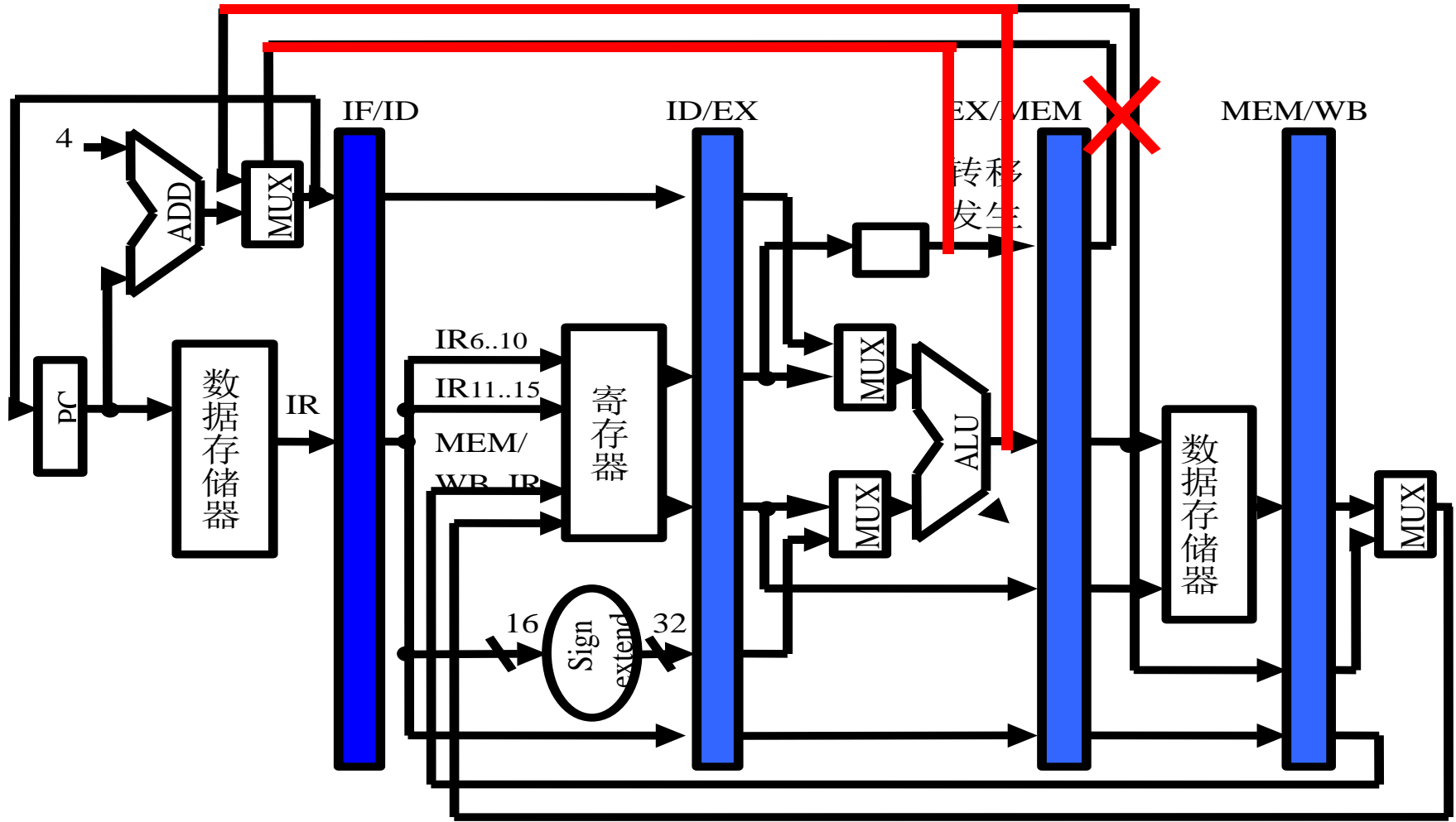
Branch is **not taken**: 3 stall

44 BEQ R1, 24	IF	ID	EX	MEM	WB		
48 AND R12, R2, R5		IF	idle	idle	idle	idle	
72 LW R4, 50(R7)			IF	ID	idle	idle	idle
76				IF	idle	idle	idle
48 AND R12, R2, R5					IF	ID	EX

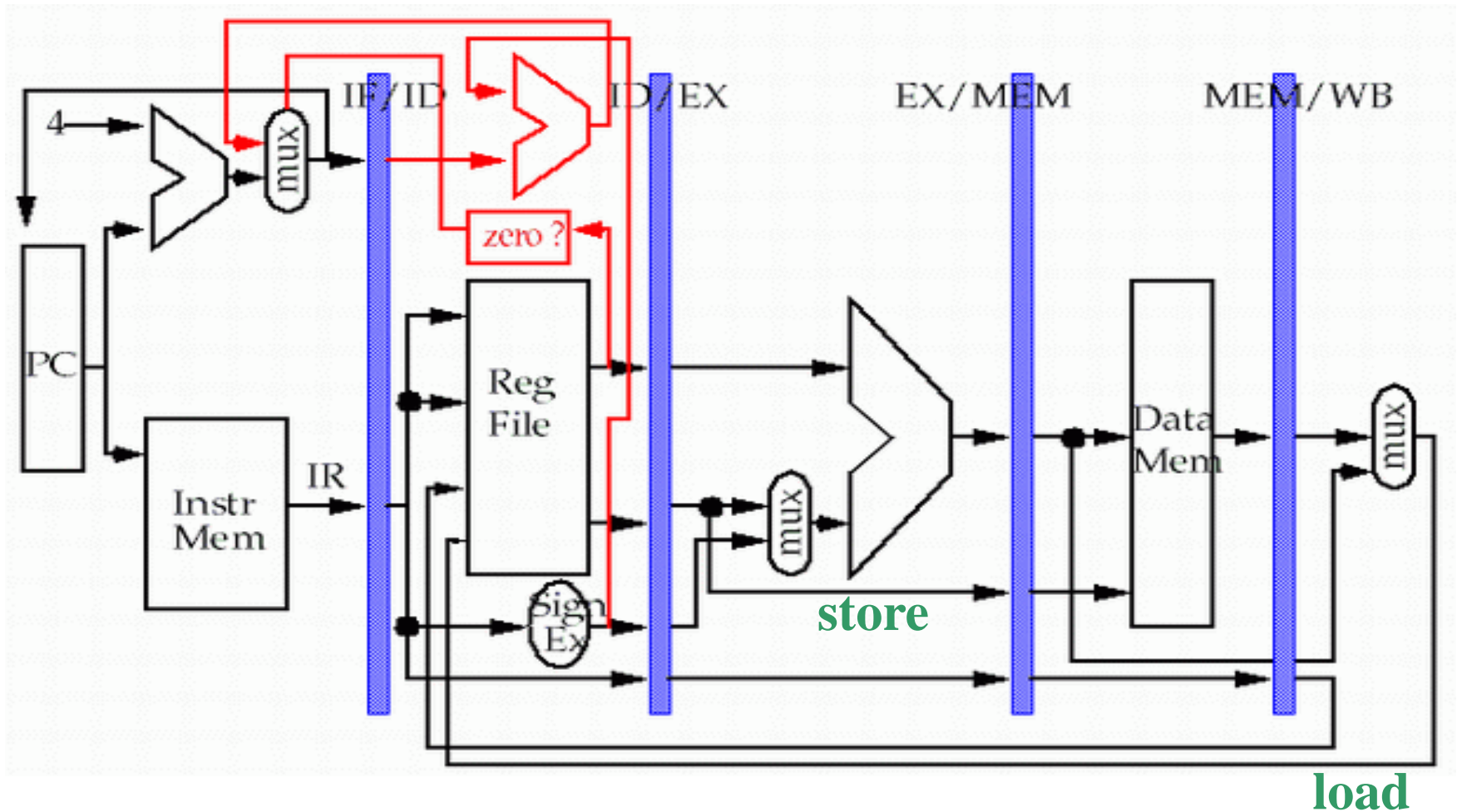
# Common Side-Effect in Pipelines

- Sometimes, you just have to guess what will execute
  - Often, we can do it right, and this saves cycles
  - But, occasionally, we are wrong
- Consequences
  - We mistakenly start executing the wrong instructions
  - To repair this, must make sure that they DO NOT really execute
  - In particular, must ensure they do not incorrectly corrupt machine state

# Move the Branch Computation Forward



# Move the Branch Computation more Forward

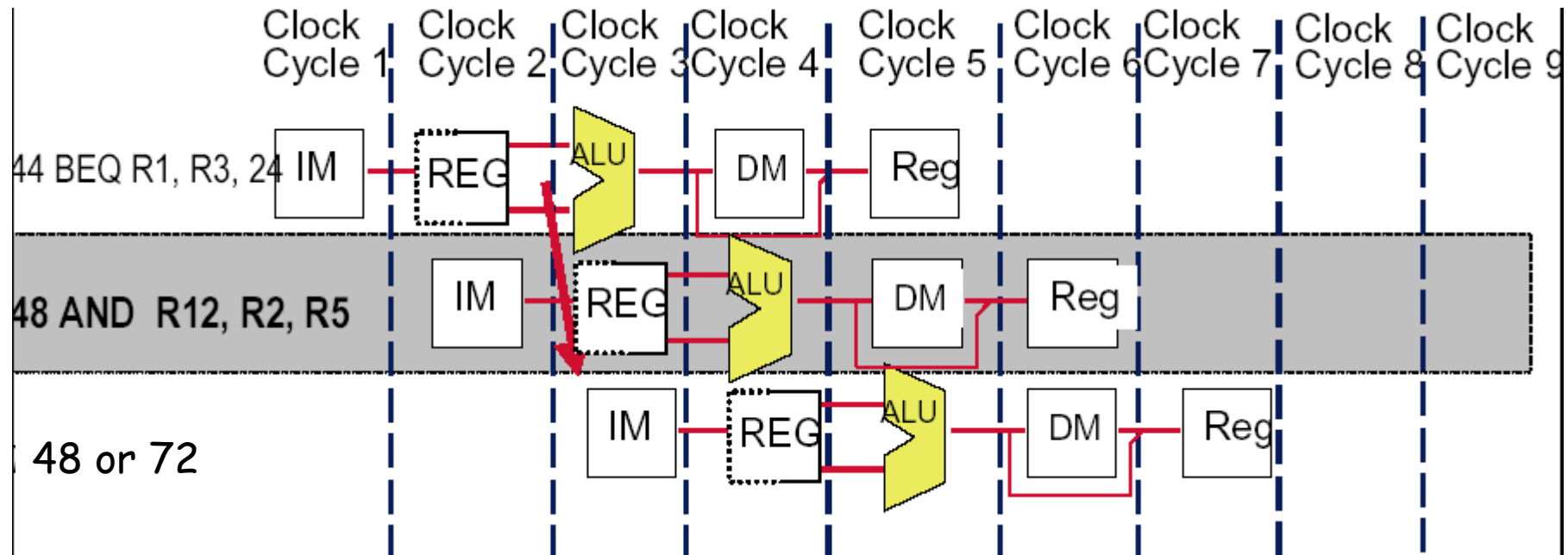




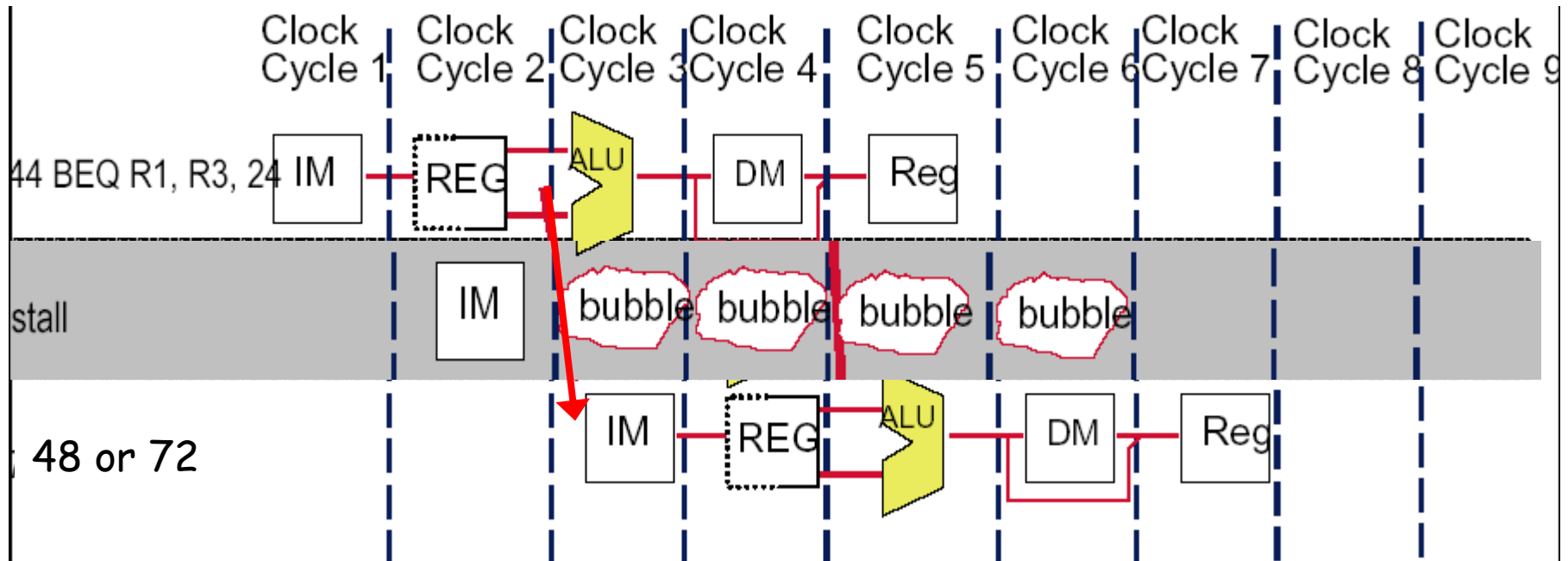
# Result: New & Improved MIPS

## Datapath

- Need just **1** extra cycle after the BEQ branch to know right address
- On MIPS, its called - **the branch delay slot**

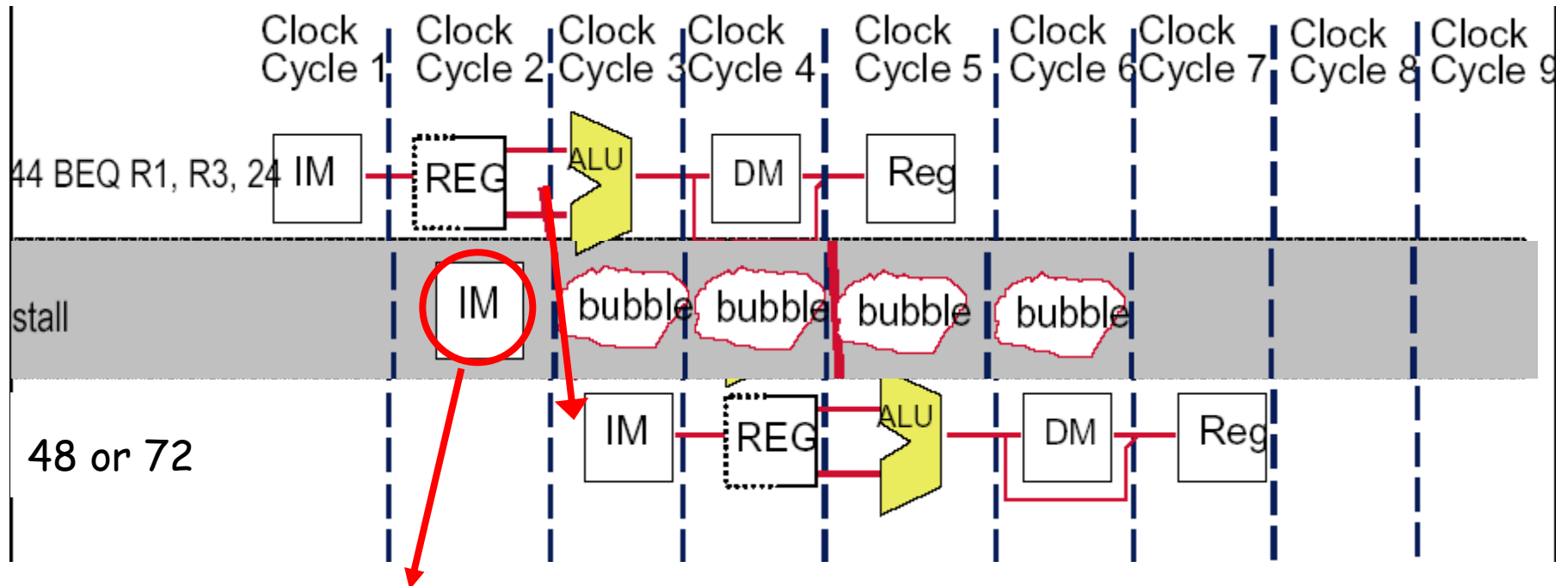


# Flushing : need only to insert 1 stall to resolve control hazard



40 ADD R30,R30,R30	IF	ID	EX	MEM	WB		
44 BEQ R1, 24		IF	ID	EX	MEM	WB	
48 AND R12, R2, R5			IF	idle	idle	idle	idle
48 or 72				IF	ID	EX	MEM

# Why “waste” the fetched instruction ?



- We have fetched the instruction 48, why we fetch the second time if the branch not taken at last ?

# The pipeline status for predict-not-taken

Branch is not taken: No stall

40 ADD R30,R30,R30	IF	ID	EX	MEM	WB		
44 BEQ R1, 24		IF	ID	EX	MEM	WB	
48 AND R12, R2, R5			IF	ID	EX	MEM	WB
52 OR R13, R6, R2				IF	ID	EX	MEM

Branch is taken: 1 stall

40 ADD R30,R30,R30	IF	ID	EX	MEM	WB		
44 BEQ R1, 24		IF	ID	EX	MEM	WB	
48 AND R12, R2, R5			IF	idle	idle	idle	idle
72 LW R4, 50(R7)				IF	ID	EX	MEM
76					IF	ID	EX

# Delayed branch

## ■ Good news

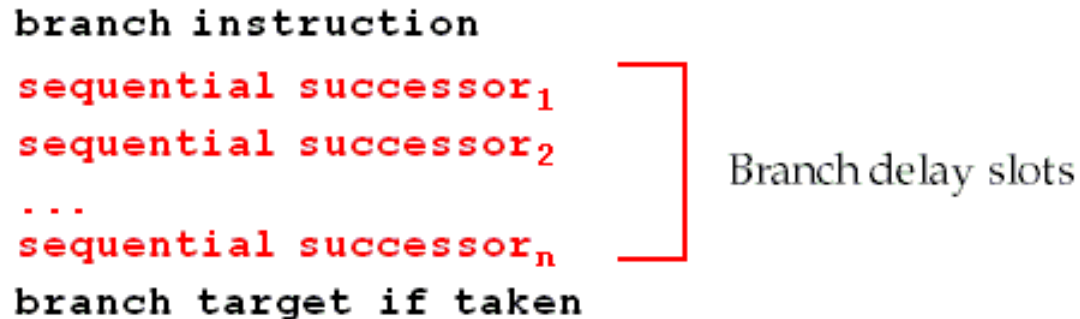
- Just 1 cycle to figure out what the right branch address is
- So, not 2 or 3 cycles of potential NOP or stall

## ■ Strange news

- OK, it's **always** 1 cycle, and we **always** have to wait
- And on MIPS, **this instruction always executes, no matter whether the branch taken or not taken. (hardware scheme)**

# Branch delay slot

- Hence the name: **branch delay slot**

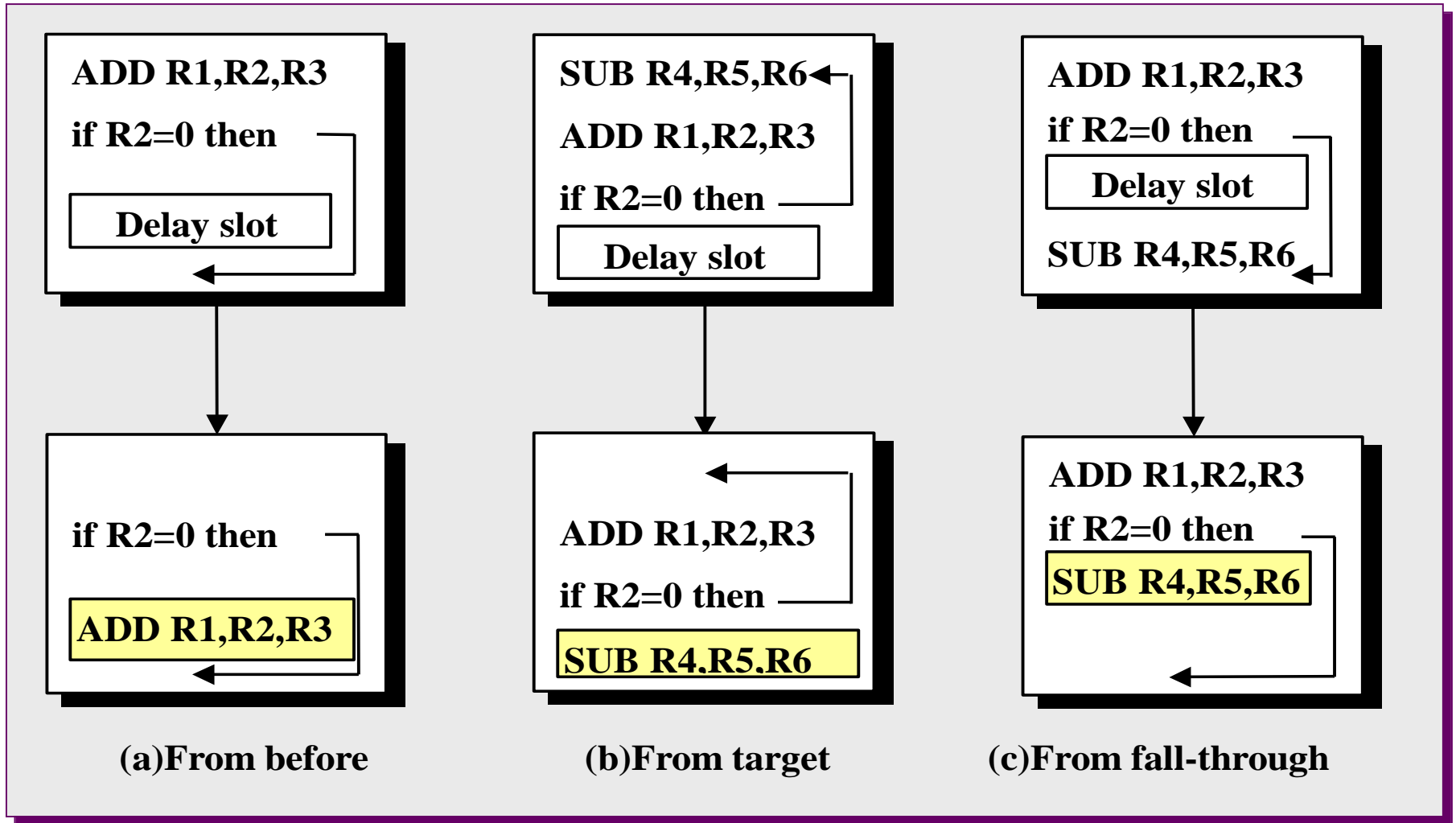


- The instruction cycle after the branch is used for address calculation , 1 cycle delay necessary
- SO...we regard this as a **free instruction cycle**, and we just DO IT

- **Consequence**

- You (or your compiler) will need to **adjust your code** to **put some useful work in that "slot"**, since just putting in a **NOP** is wasteful (**compiler scheme**)

# How to adjust the codes?



# Example: rewrite the code (a)

## Without Branch Delay Slot

Address	Instruction
36	NOP
40	ADD R30, R30, R30
44	BEQ R1, 24
48	AND R12, R2, R5
52	OR R13, R6, R2
56	ADD R14, R2, R2
60	...
64	...
68	...
72	LW R4, 50(R7)
76	...

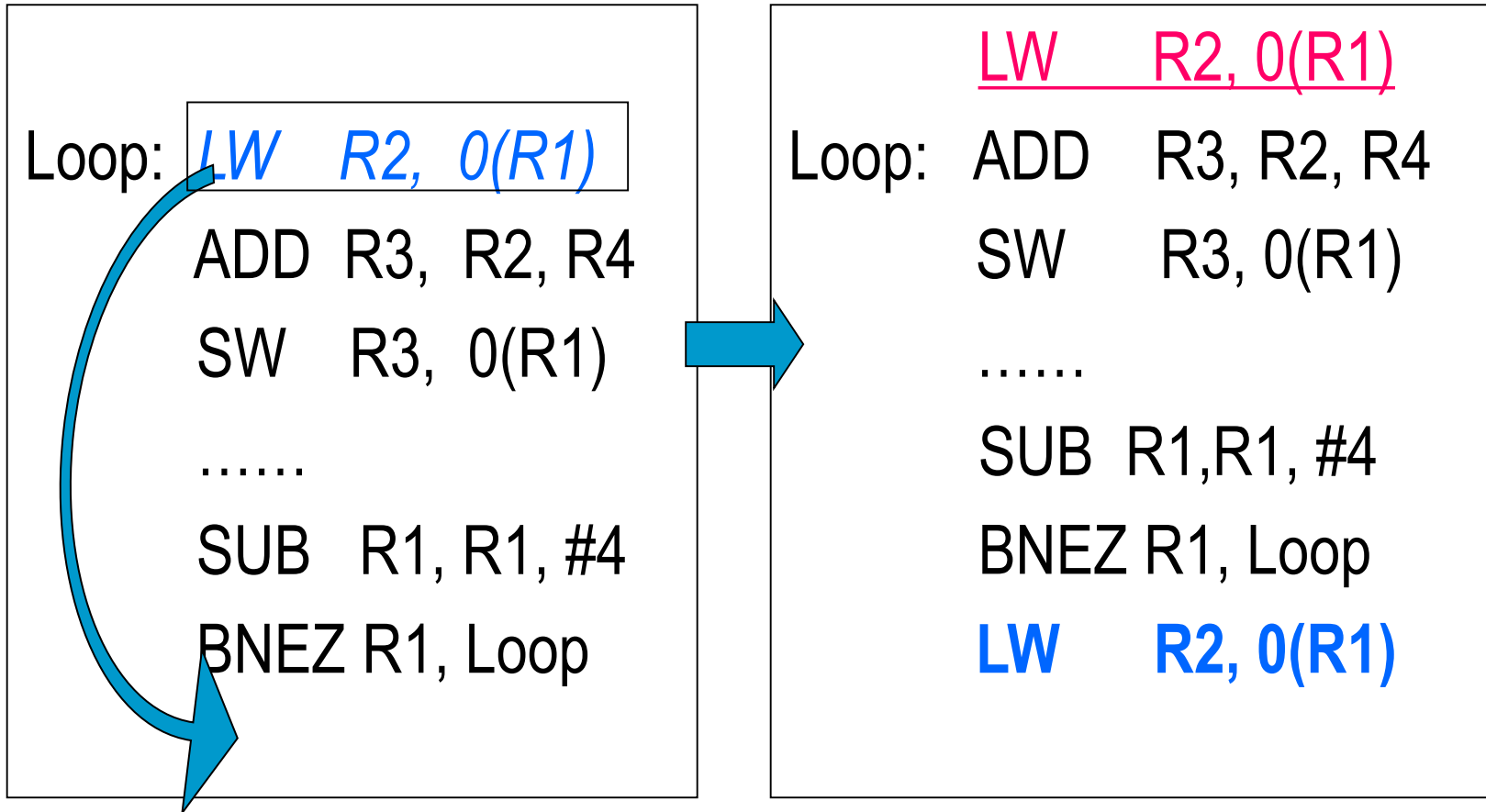
## With Branch Delay Slot

Address	Instruction
36	NOP
40	BEQ R1, R3, 28
44	ADD R30, R30, R30
48	AND R12, R2, R5
52	OR R13, R6, R2
56	ADD R14, R2, R2
60	...
64	...
68	...
72	LW R4, 50(R7)
76	...

- Flow of instructions if branch is taken: 36, 40, 44, 72, ...
- Flow of instructions if branch is not taken: 36, 40, 44, 48, ...

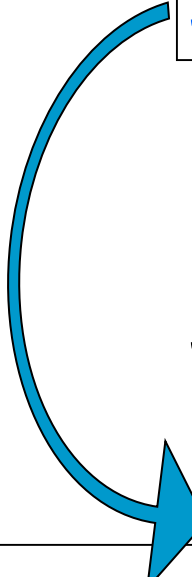


# Example: rewrite the code (b-1)

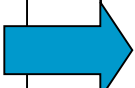


# Example: rewrite the code (b-2)

```
Loop: LW  R2, 0(R1)
      ADD R3, R2, R4
      SW  R3, 0(R1)
      DIV .....
      .....
      SUB R1, R1, #4
      BNEZ R1, Loop
```



```
Loop: LW  R2, 0(R1)
      ADD R3, R2, R4
      DIV .....
      .....
      SUB R1, R1, #4
      BNEZ R1, Loop
      SW  R3, +4(R1)
```



# Scheduling strategy vs. performance improvement

Scheduling strategy	Requirements	Improves performance when?
a. From before branch	Branch must not depend on the rescheduled instruction	Always
b. From target	Must be OK to execute rescheduled instruction if branch if not taken. May need to duplicate instructions.	When branch is taken. May enlarge program if instructions are duplicated.
c. From fall through	Must be OK to execute instruction if branch is taken.	When branch is not taken.
d. place a no-op		No improvement.

# Pipeline status for delayed branch

misprediction, ( predict Taken, but not taken)

Untaken br. ins.	IF	ID	EX	MEM	WB				
Br. delay ins.(i+1)		IF	ID	EX	MEM	WB	多执行一条指令		
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Accurate Prediction

taken br. ins.	IF	ID	EX	MEM	WB				
Br. delay ins.(i+1)		IF	ID	EX	MEM	WB	应该执行,无 stall		
Branch target			IF	ID	EX	MEM	WB		
Branch target +1				IF	ID	EX	MEM	WB	
Branch target +2					IF	ID	EX	MEM	WB

# Constraints of the delayed branch

- There are restrictions on the instructions that are scheduled into the delay slots
- The compiler's ability to predict accurately whether or not a branch is taken determines how much useful work is actually done.
- For scheduling scheme b and c,
  - It must be O.K. to execute the SUB instruction if the prediction is wrong.
  - Or the hardware must provide a way of **cancelling** the instruction.

# Cancelling function

- Includes the direction that the branch is predicted to go.
- If branch is predicted incorrectly , CPU turns the instruction in the branch delay slot into a no-op.
- Can reduce the complexity for compiler to select useful instructions into delay slot.

# Delayed branch with cancelling(of case b)

预测出错时,由硬件取消延时槽指令(转换成一条空操作指令)

Untaken br. ins.	IF	ID	EX	MEM	WB				
Br. delay ins.(i+1)		IF	idle	idle	idle	idle			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

预测正确,性能没有损失

taken br. ins.	IF	ID	EX	MEM	WB				
Br. delay ins.(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

# Efficiency of delayed branch

	A	B	C	D	E = C × D	F = B + E
Banchmar	% cond. Br.	%cond. Br. with empty slots	%cond. br. That are canceling	%cancelling br. That are cancelled	%br. with cancelled delay slots	Total% Br. with empty or cancelled delay slots
Compress	14%	18%	31%	43%	13%	31%
Eqntott	24%	24%	50%	24%	12%	36%
Espresso	15%	29%	19%	21%	4%	33%
Gcc	15%	16%	33%	34%	11%	27%
Li	15%	20%	55%	48%	26%	46%
Integer Ave.	17%	21%	38%	34%	13%	35%
Doduc	8%	33%	12%	62%	7%	40%
Ear	10%	37%	36%	14%	5%	42%
Hydro21	12%	0%	69%	24%	7%	17%
Mdljdp2	9%	0%	86%	10%	8%	8%
Su2cor	3%	7%	17%	57%	10%	17%
FP average	8%	16%	44%	34%	9%	25%
Overall Ave.	12%	18%	41%	34%	12%	30%





# About delayed branch

- Delayed branch are adopted in most RISC processors.
- In general, the length of branch delay is more than 1 . However, always just one slot is used due to the compiler complexity.

# Performance comparison for 4 schemes

- MIPS R4000, deeper pipeline
  - Takes at least **three pipeline stages** before the branch target address is known
  - **An additional cycle** before the branch condition is evaluated.
  - Assuming branch frequencies as followed
    - Unconditional branch 4%
    - Conditional branch, untaken 6%
    - Conditional branch, taken 10%

# Pipeline status for various schemes

■ **uncond.**    L1 L2 L3 **L4** L5 L6    2 stall

**L1** **s** **L1**(branch target)

■ **Stall pipeline:**    L1 L2 L3 L4 **L5** L6    3 stall

**L1** **s** **s** **L1**(branch target/i+1)

■ **Predict taken:**

                 L1 L2 L3 L4 **L5** L6

taken:            **L1** **s** L1 L2 L3    2 stall

untaken:        **L1** **s** **L1** idle    3 stall

                 L1(i+1)

■ **Predict untaken:**

                 L1 L2 L3 L4 **L5** L6

untaken:        L1 L2 L3 L4 L5    0 stall

taken:            **L1** **L2** **L3** idle    3 stall

**L1** **L2** idle

**L1** idle

                 L1(branch target)

# Resolution

Branch scheme	Additiona to the CPI						All branches (20%)
	Unconditional Branches(4%)		Taken cond. Branches(10%)		Untaken cond. Branches(6%)		
Stall pipeline	2	0.08	3	0.30	3	0.18	0.56
Predict taken	2	0.08	2	0.20	3	0.18	0.46
Predict untaken	2	0.08	3	0.30	0	0.00	0.38

# Pipeline hazards

## ■ Taxonomy of Hazards

- Structural hazards
  - These are conflicts over hardware resources.
- Data hazards
  - Instruction depends on result of prior computation which is not ready (computed or stored) yet
- Control hazards
  - branch condition and the branch PC are not available in time to fetch an instruction on the next clock
  - OK, we did these, calculate the destination address and condition asap, Flushing the pipeline, predict-not-taken, predict-taken, delayed branch ( with/without cancelling)

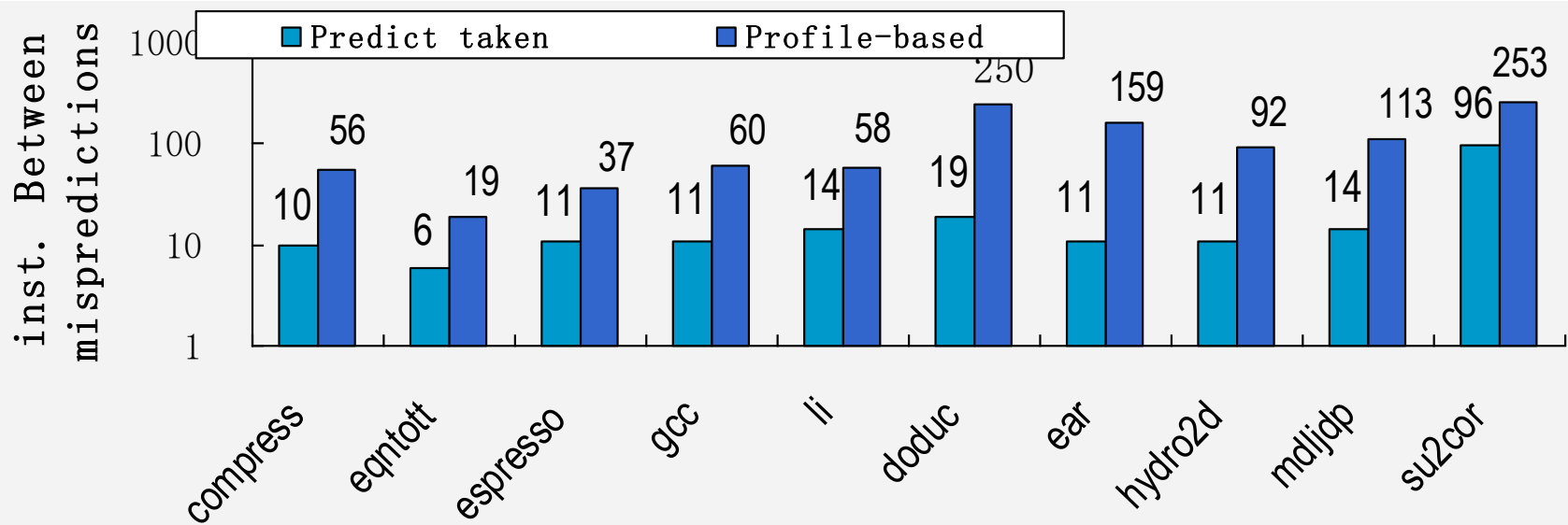
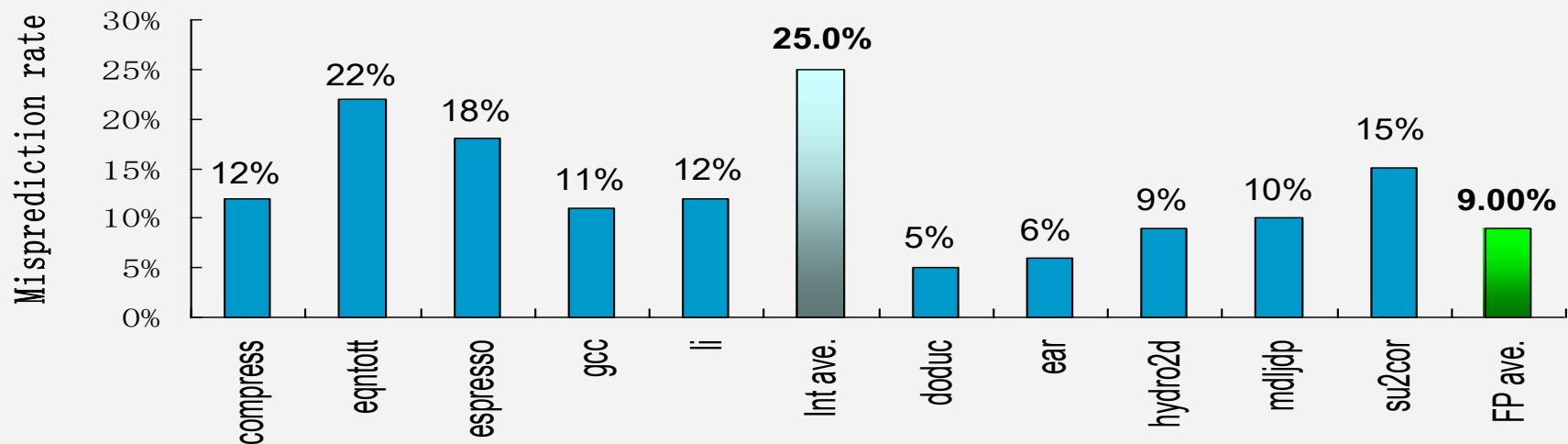
# Summary for control hazard

- Control hazards can cause a greater performance loss than do data hazards.
- In general, **the deeper the pipeline, the worse the branch penalty** in clock cycles.
- A higher CPI processor can afford to have more expensive branches.
- The efficiency of the three schemes greatly depends on the **branch prediction**.

# Branch prediction

- There are many different schemes
- static branch prediction
  - Assume taken
    - This is surprisingly effective since 85% of backward branches and 60% of forward branches are taken.
  - Assume not taken
  - Predict by using profile information from previous run
- dynamic branch prediction by hardware
  - 1-bit Branch Prediction
  - 2-bit Branch Prediction
  - N-bit Branch Prediction
  - Correlating Branch Prediction

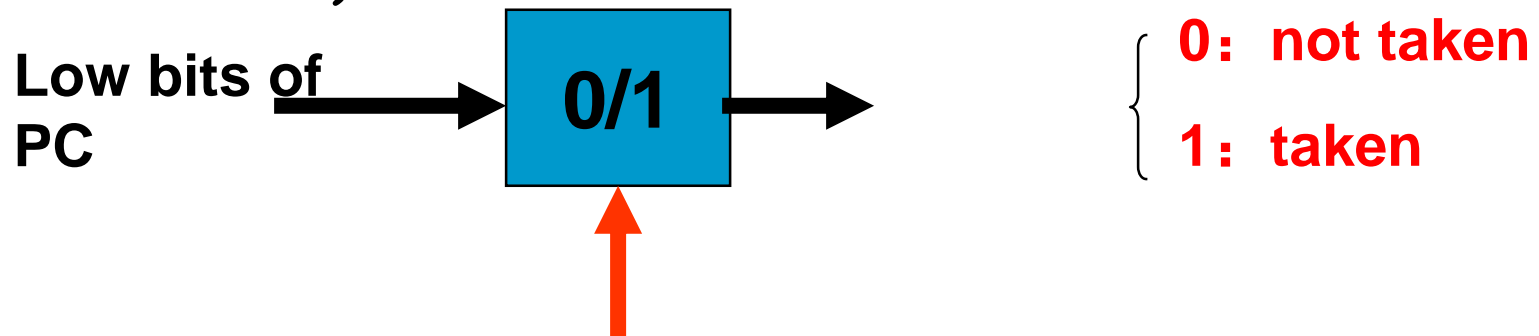
# Accuracy of prediction based on profile





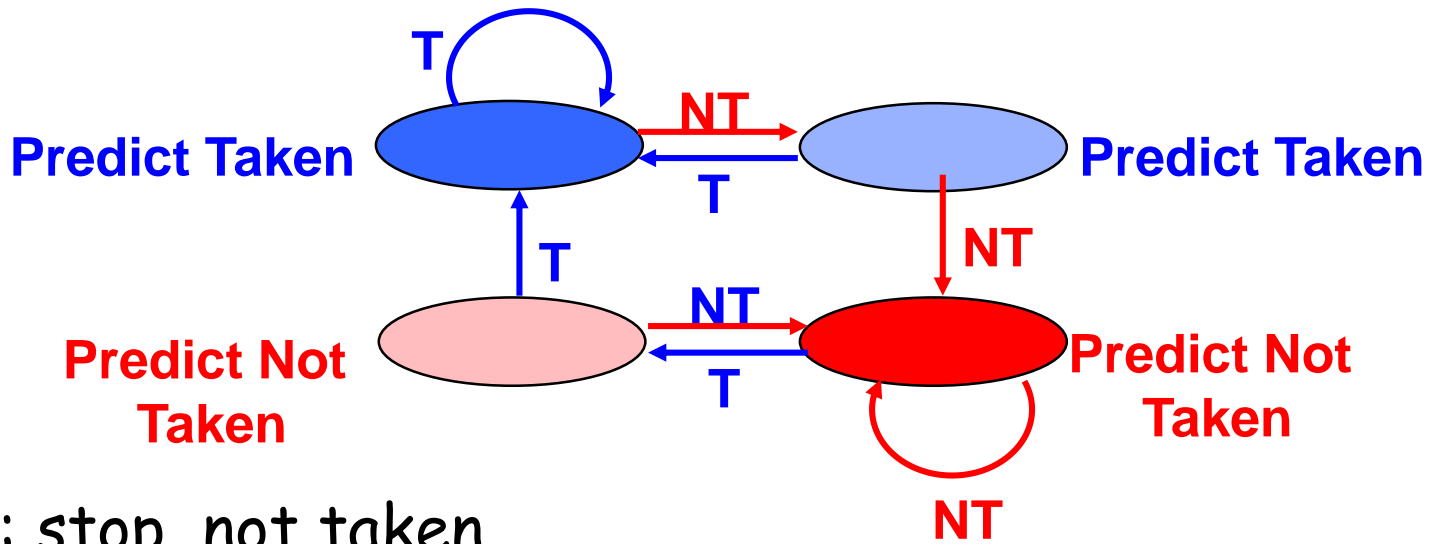
# 1. One bit branch predictor

- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check (saves HW, but may not be right branch)



## 2、2-bit Branch-Prediction Buffer

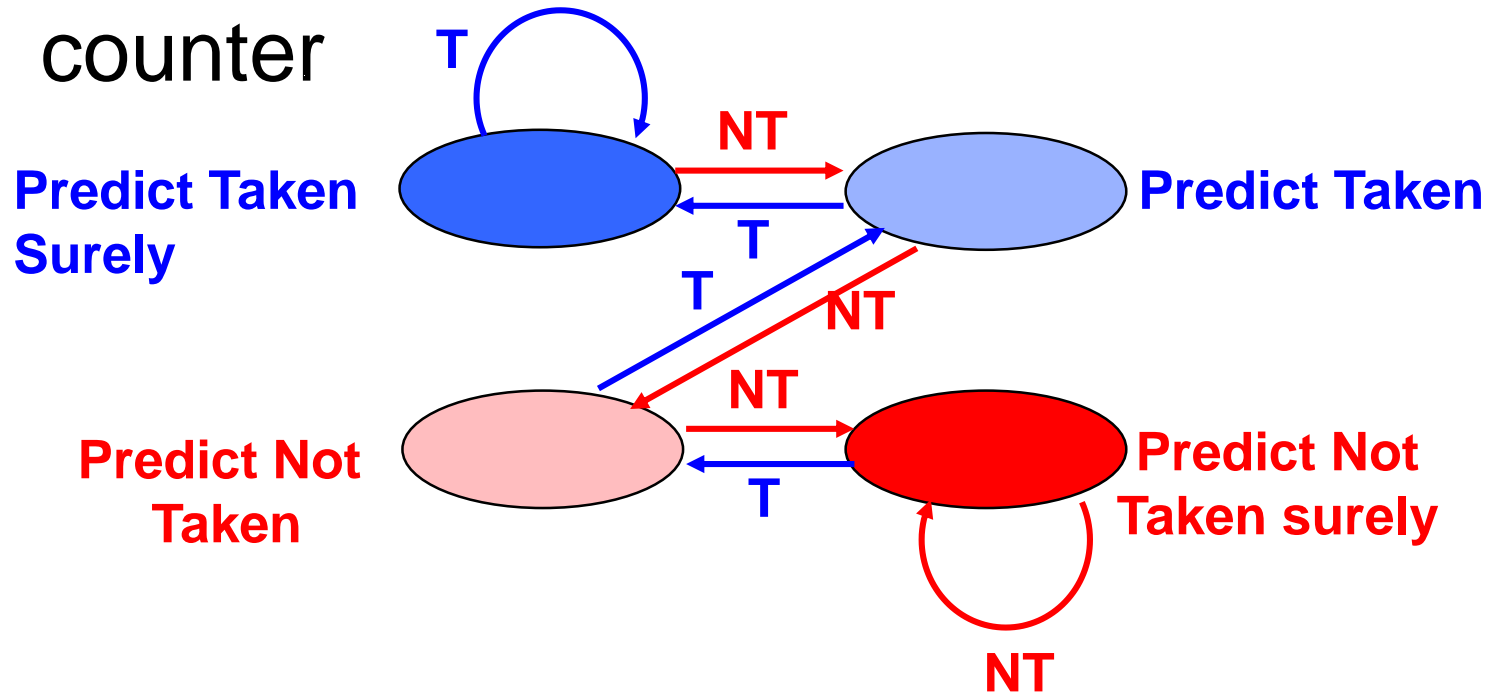
- Solution: 2-bit scheme where change prediction only if get misprediction *twice*:
  - (Jim Smith, 1981, [A study of branch prediction strategies](#))



- Red: stop, not taken
- Blue: go, taken
- Adds *hysteresis* to decision making process

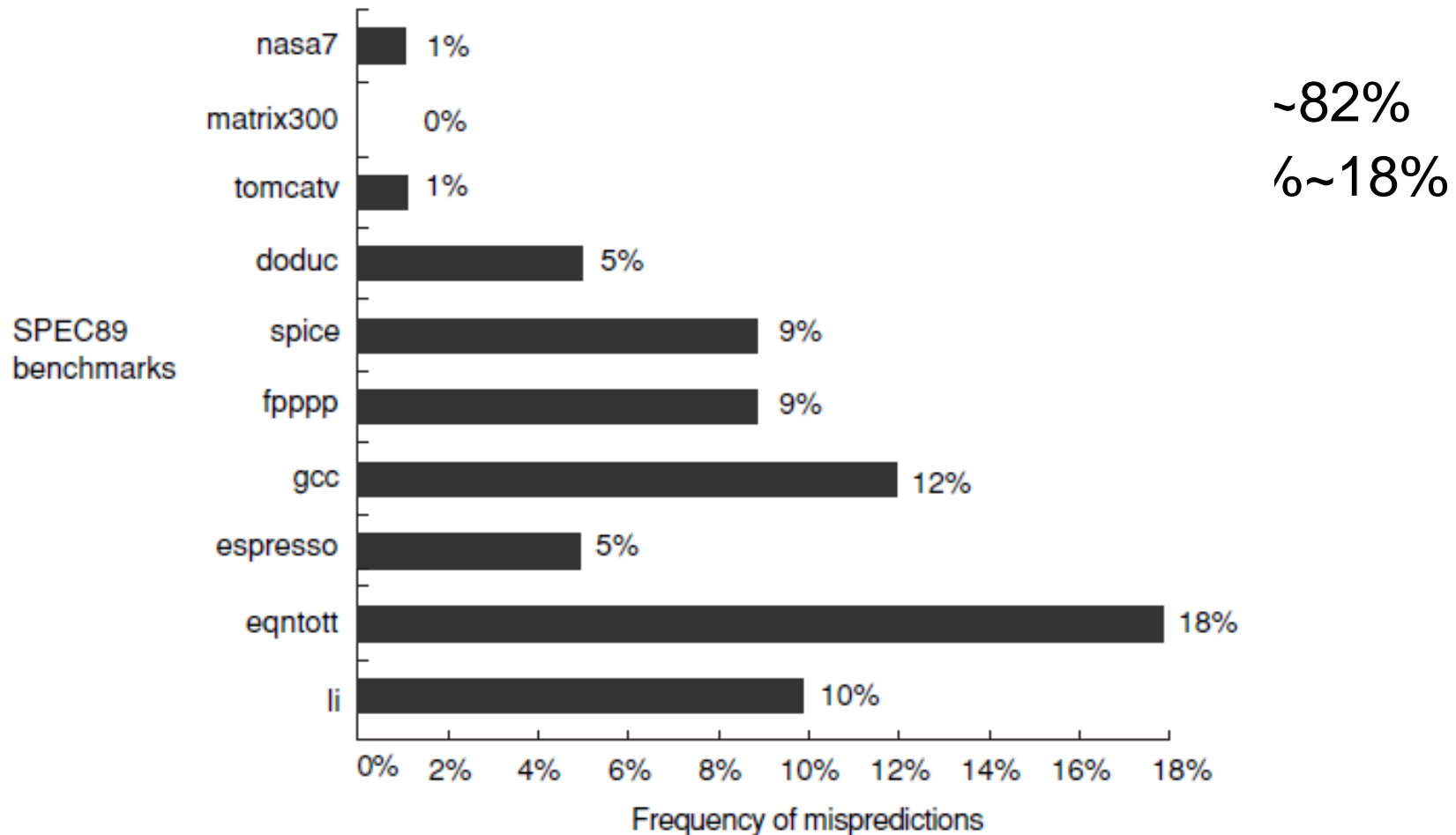
# An alternative 2-bit predictor

- It can be implemented using a 2-bit counter



# Accuracy of the 2-bit predictor

■ 4096 entries, 2-bit predictor

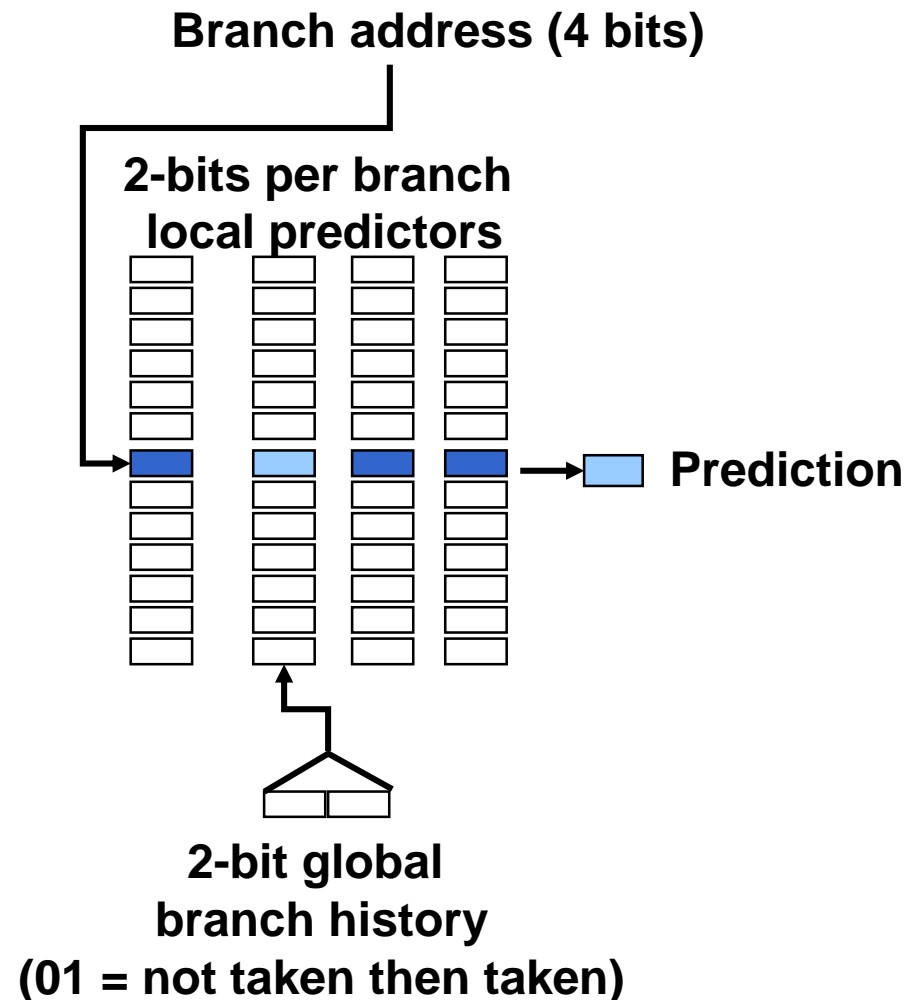


### 3. Generalize the 2-bit predictor to n-bit predictor

- N-bit counter ( $0 < x < 2^n - 1$ )
- Predict taken  $X \geq 2^{n-1}$
- Predict untaken  $X < 2^{n-1}$
- Counter + 1 when a taken branch
- Counter - 1 when a not taken branch
- Most system rely on 2-bit predictor rather than n-bit predictor.

# 4. Correlating Branches prediction buffer

- Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)
  - Then behavior of recent branches selects between, say, 4 predictions of next branch, updating just that prediction
- (2,2) predictor: 2-bit global, 2-bit local



# Readings

- Tse-Yu Yeh and Yale Patt, "[Alternative Implementations of Two-level Adaptive Branch Prediction](#)", *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1992. [See ISCA Retrospective](#)
- Cliff Young, Nicolas Gloy, and Michael D. Smith, "[A Comparative Analysis of Schemes for Correlated Branch Prediction](#)", *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995.
- Chih-Chieh Lee, I-Cheng Chen, and Trevor Mudge, [The Bi-Mode Branch Predictor](#), *Proceedings of International Symposium on Microarchitecture (MICRO-30)*, 1997
- Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt, "[Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work](#)," *Proceedings of the International Symposium on Computer Architecture (ISCA-25)*, June 1998