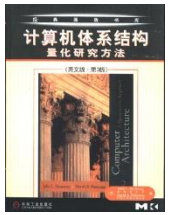
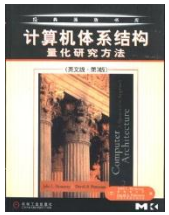


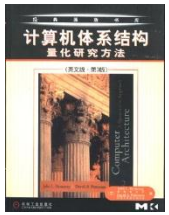
Lecture 3 for pipelining

- The control hazard
- How to solve the control hazard



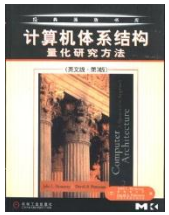
Pipelining Hazards

- **Taxonomy of Hazards**
  - Structural hazards
    - √ These are conflicts over hardware resources.
    - √ OK, maybe add extra hardware resources;
      or full pipelined the functional units(split duble bump);
      otherwise still have to stall
  - 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, 24	<- this branchs 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	...	

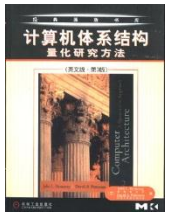
R0

We execute all these if R1 != R3

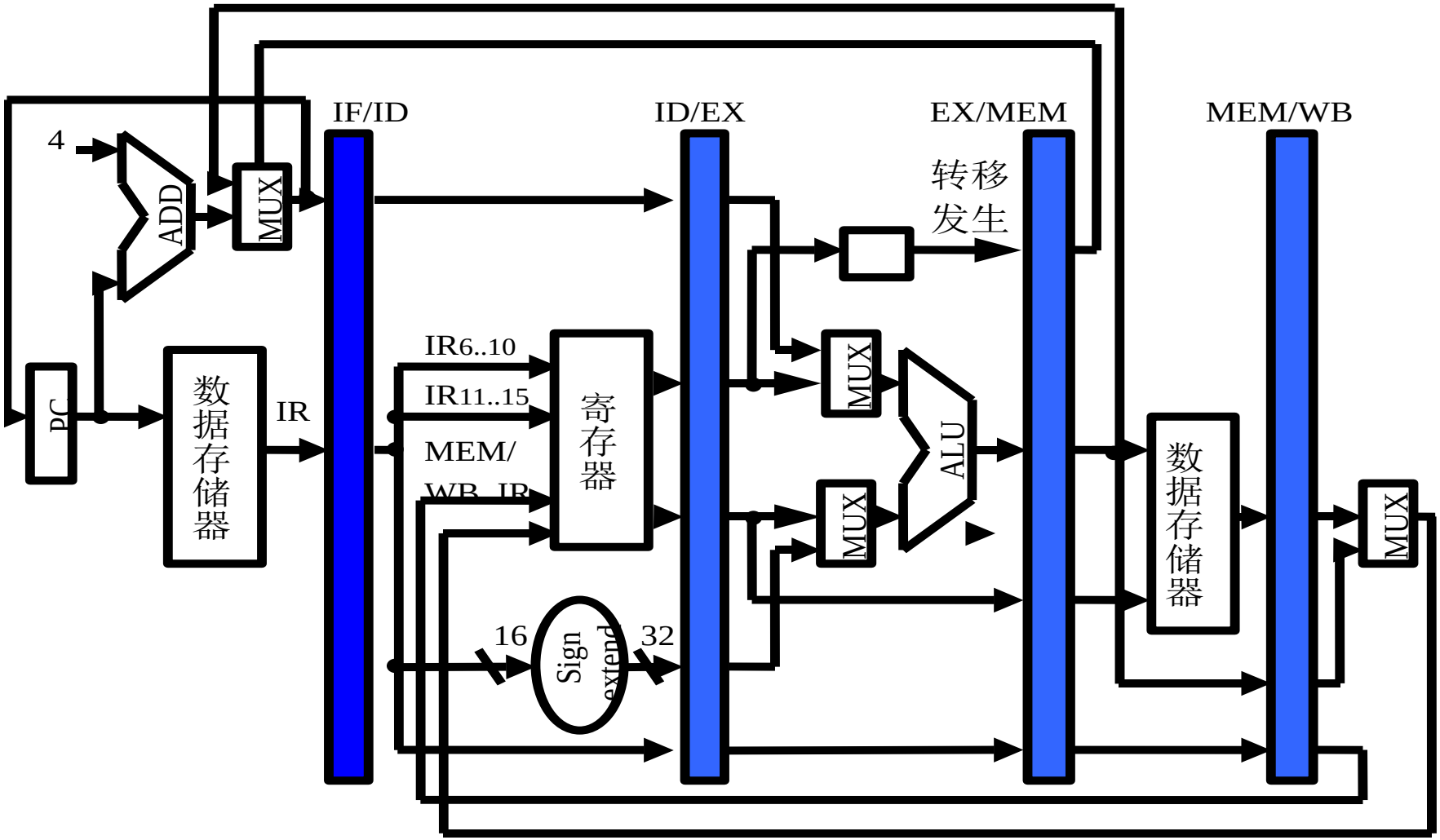
R0

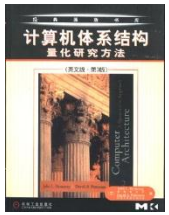
We execute just these if R1 == R3

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

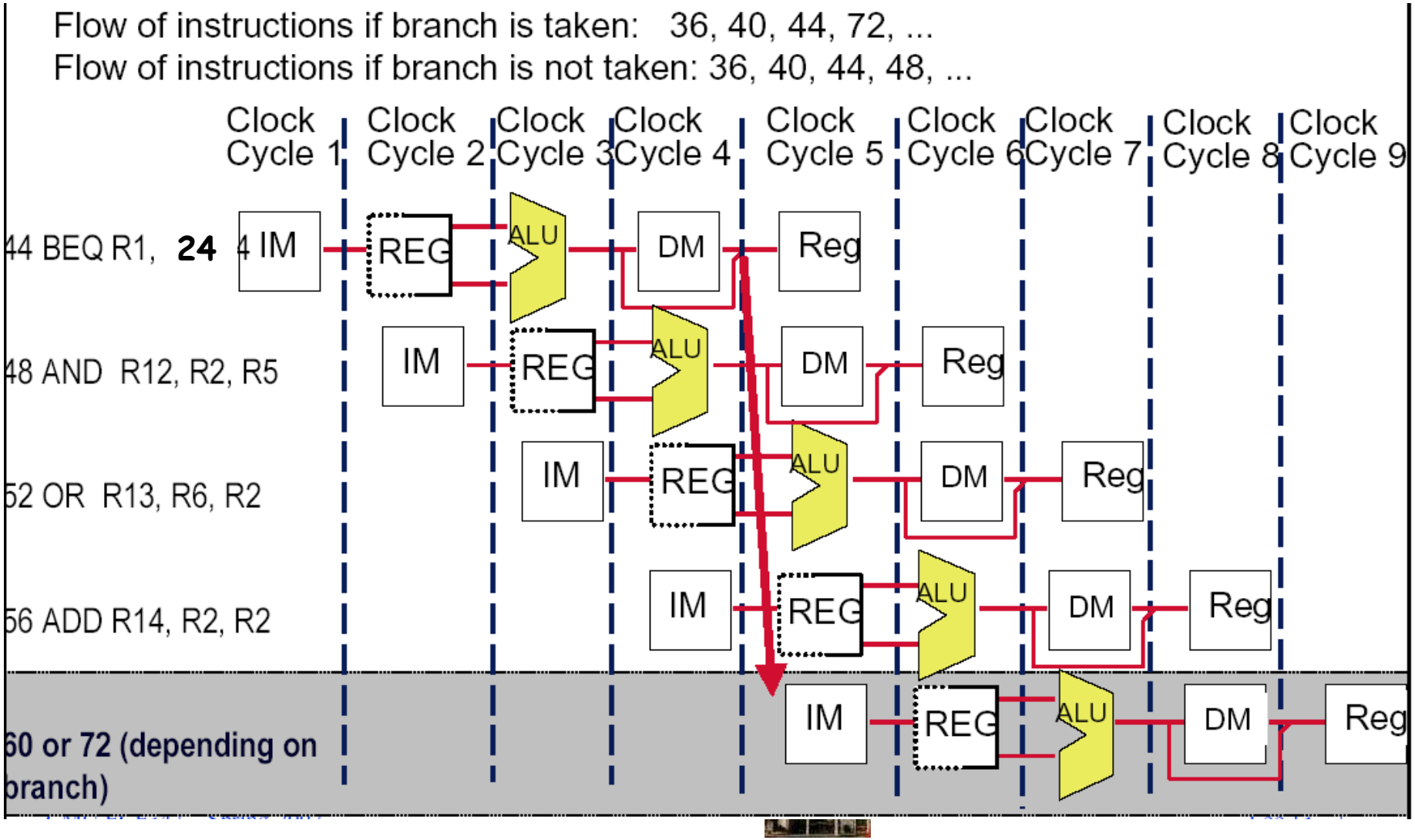


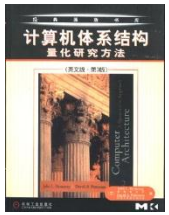
Recall: Basic Pipelined Datapath





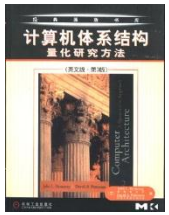
Control hazard





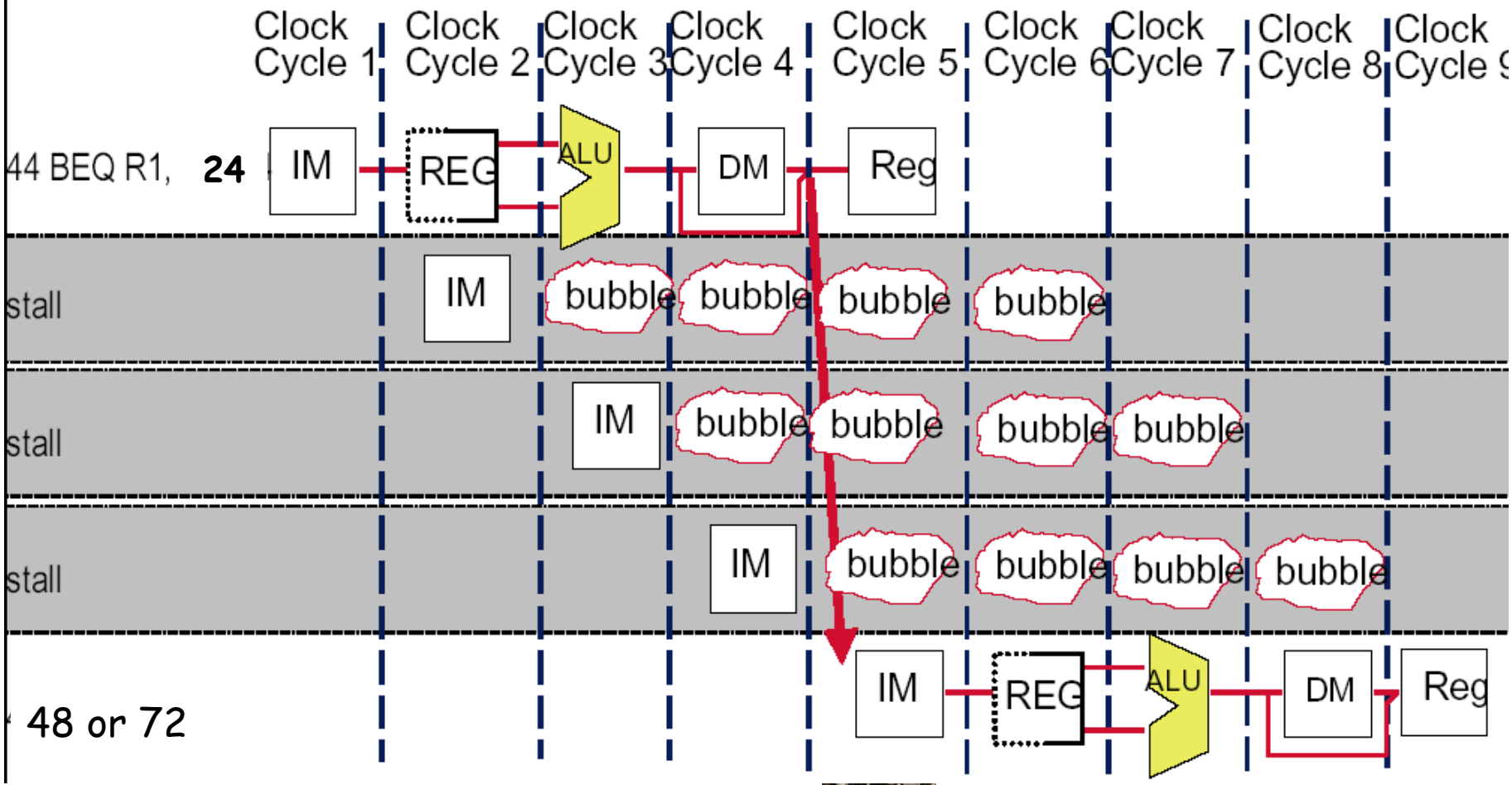
Dealing with the control hazard

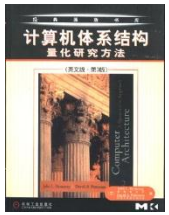
- Four simple solutions
  - Freeze or flush the pipeline
  - Predict-not-taken (Predict-untaken)
    - √ Treat every branch as not taken
  - Predict-taken
    - √ Treat every branch as taken
  - Delayed branch
- Note:
  - Fixed hardware
  - Compile time scheme using knowledge of hardware scheme and of branch behavior



Recall: solve the hazard by inserting stalls

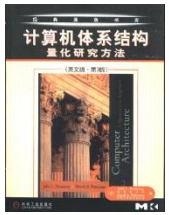
Flow of instructions if branch is not taken: 36, 40, 44, 48, ...





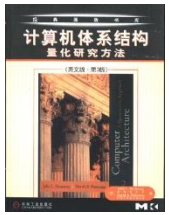
The pipeline status

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

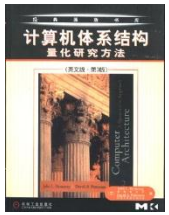
- Simplest hardware:
  - Holding or deleting any instruction after branch until the branch destination is know.
  - Penalty is fixed.
  - Can not be reduced by software.



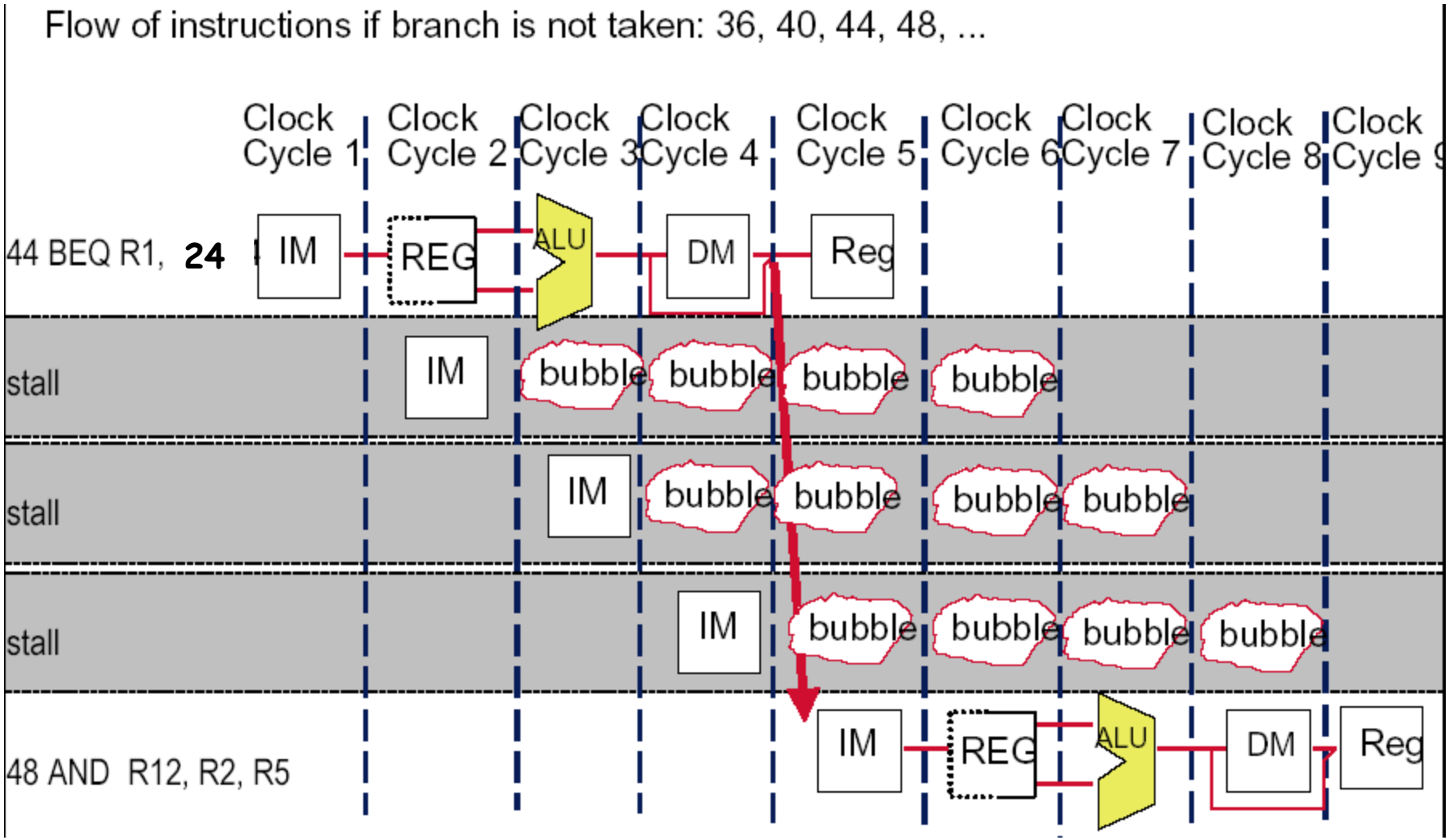
Stalls greatly hurt the performance

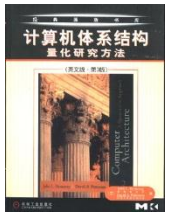
- *Problem:*
  - *With a 30% branch frequency and an ideal CPI of 1, how much the performace is by inserting stalls ?*

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

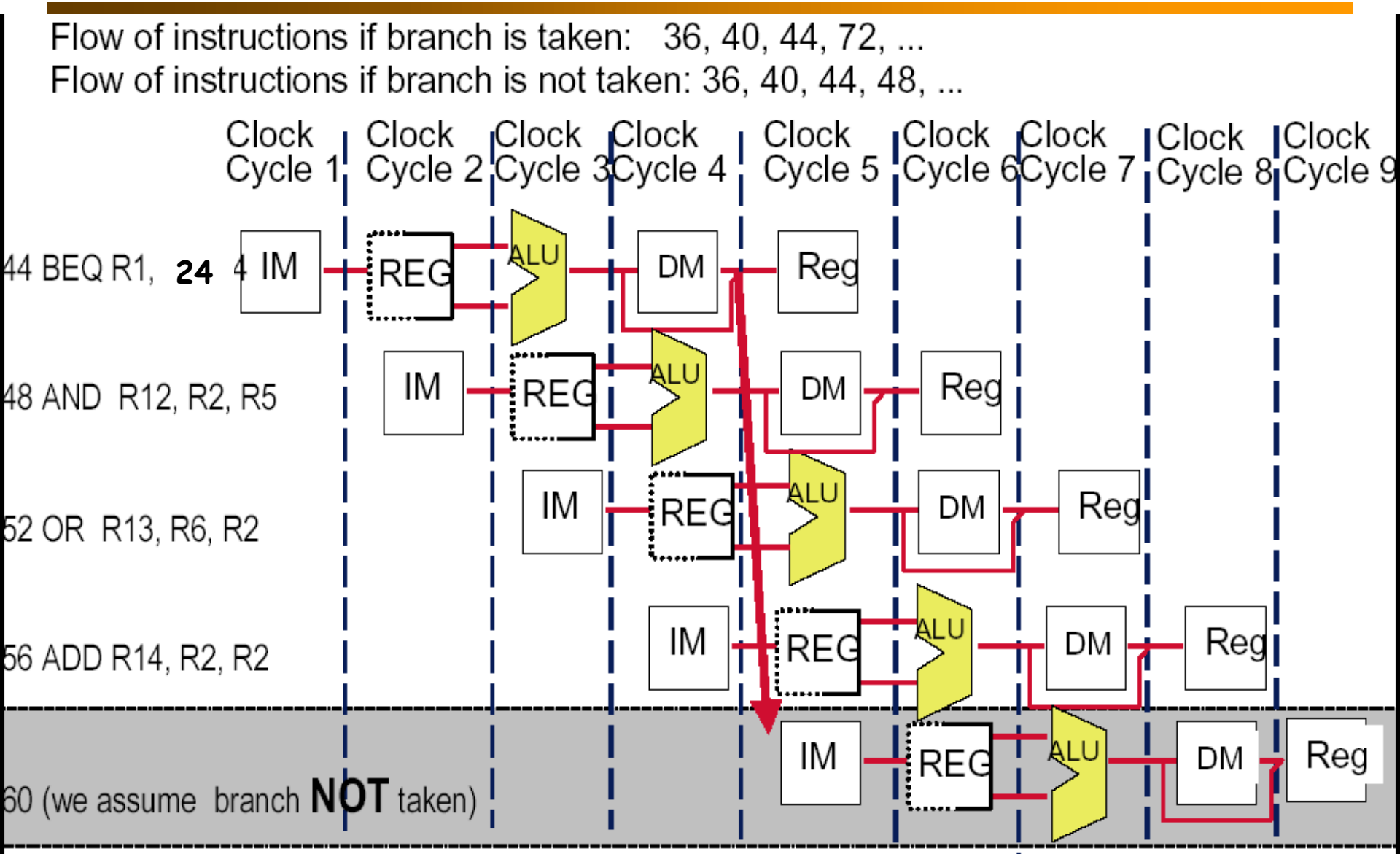


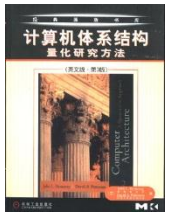
Always Stalling Hurts the Not-taken case





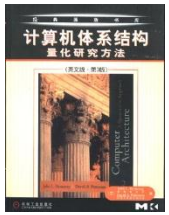
How about assume Branch *Not* Taken





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



What If Branch *Was* Taken…?

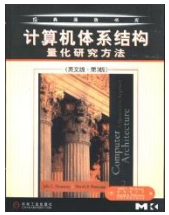
…i.e., what if we guessed wrong on the branch?

Address	Instruction
36	NOP
40	ADD R30,R30,R30
44	BEQ R1, 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	…

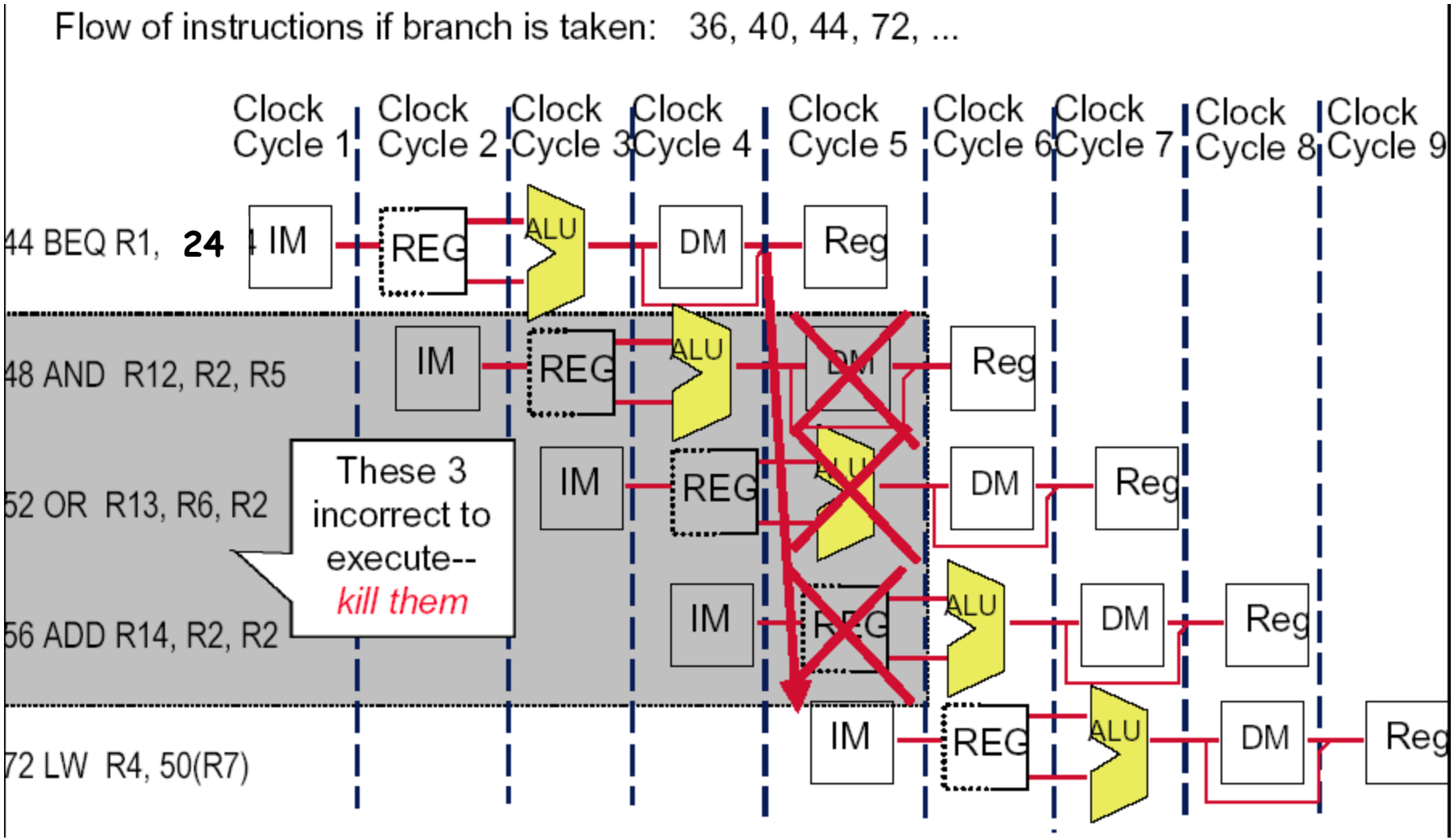
We already started some of these since we assumed NO branch taken

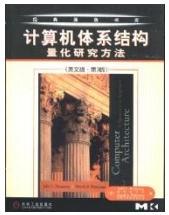
But a few clock cycles later, We figure out these are right Instructions to go next

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



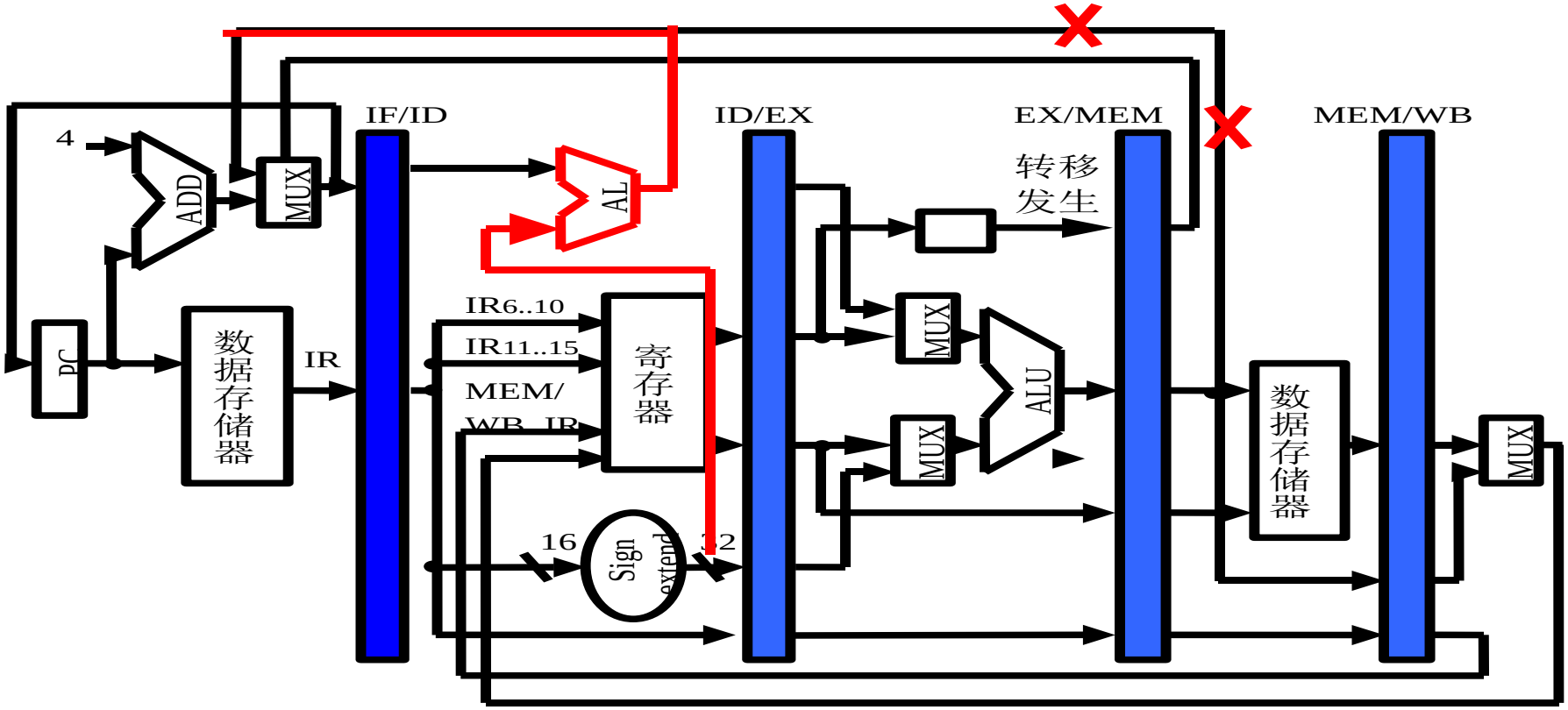
How to do with the branch taken ?

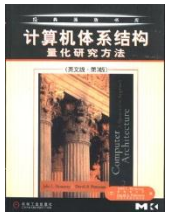




Alternative is assuming the branch always *taken*

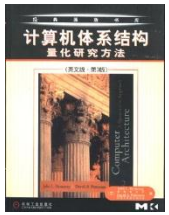
- Most branches(60%) are taken, so we should make the taken branch more faster. Why not try assuming the branch always taken?





Predict –taken

- Hardware
  - Treat every branch as taken (evidence: more than 60% braches 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.



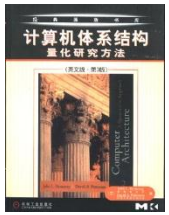
Pipeline status for predict-taken

Branch is taken: 1 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	EX	MEM	WB
76				IF	ID	EX	MEM
80					IF	ID	EX

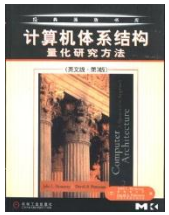
Branch is *not* taken: 3 stall

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

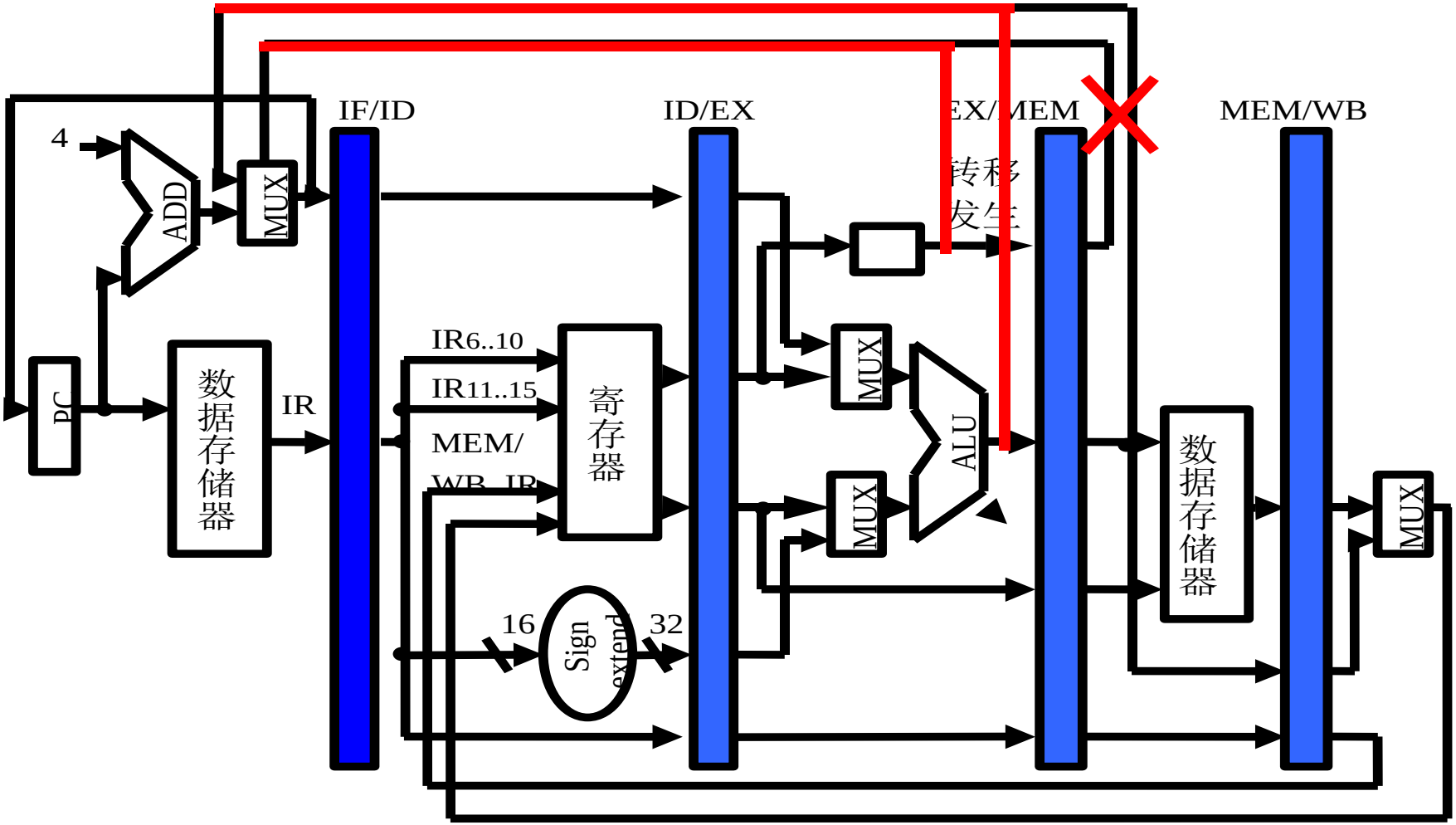


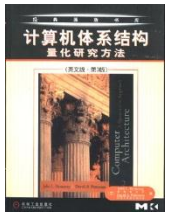
Common Side-Effect in Pipelines

- 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

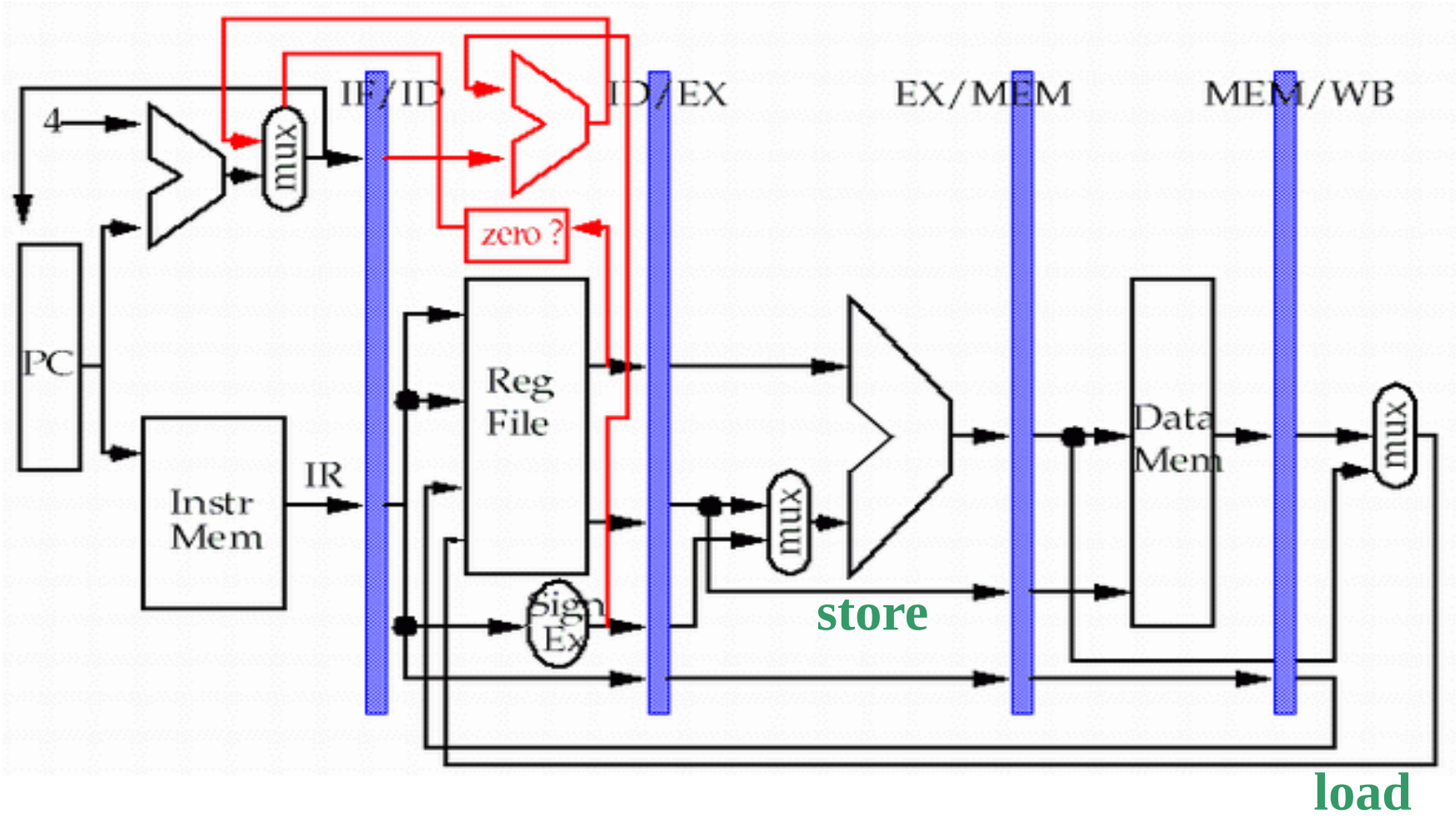


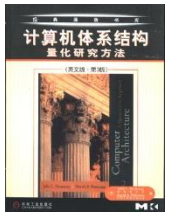
Move the Branch Computation Forward





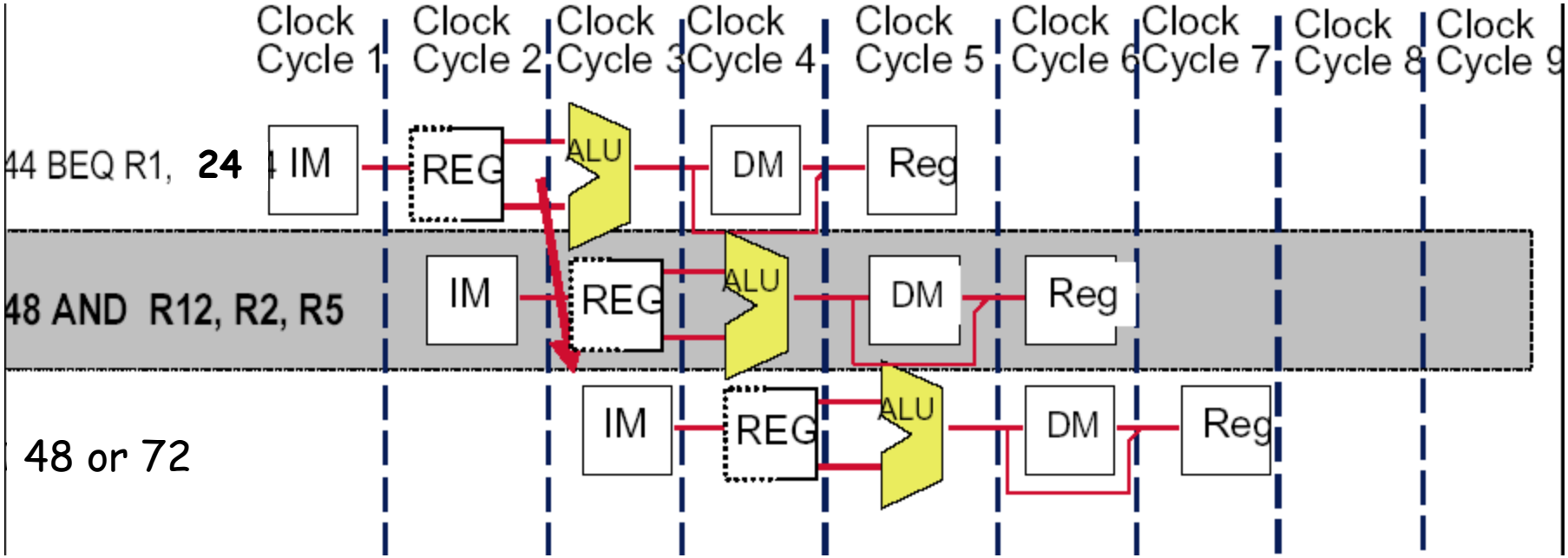
Move the Branch Computation more Forward

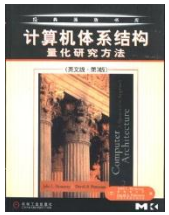




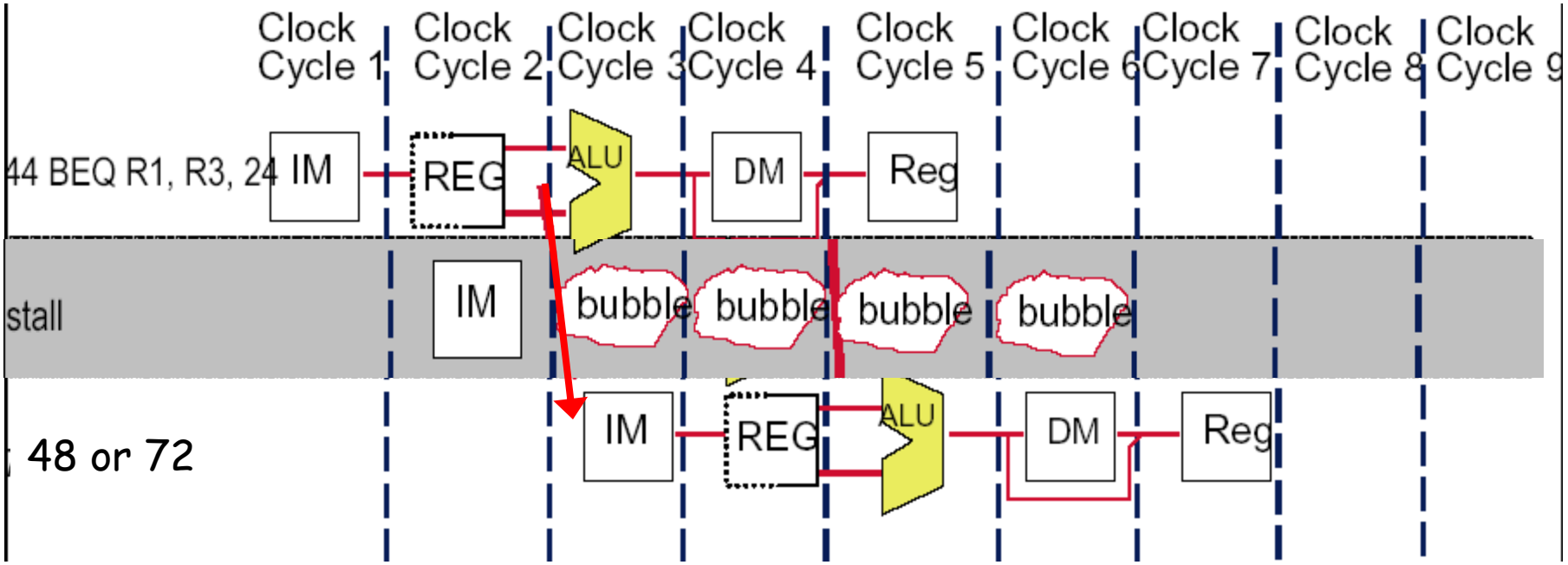
Result: New & Improved MIPS Datapath

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

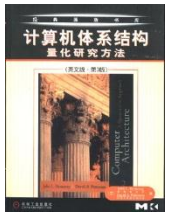




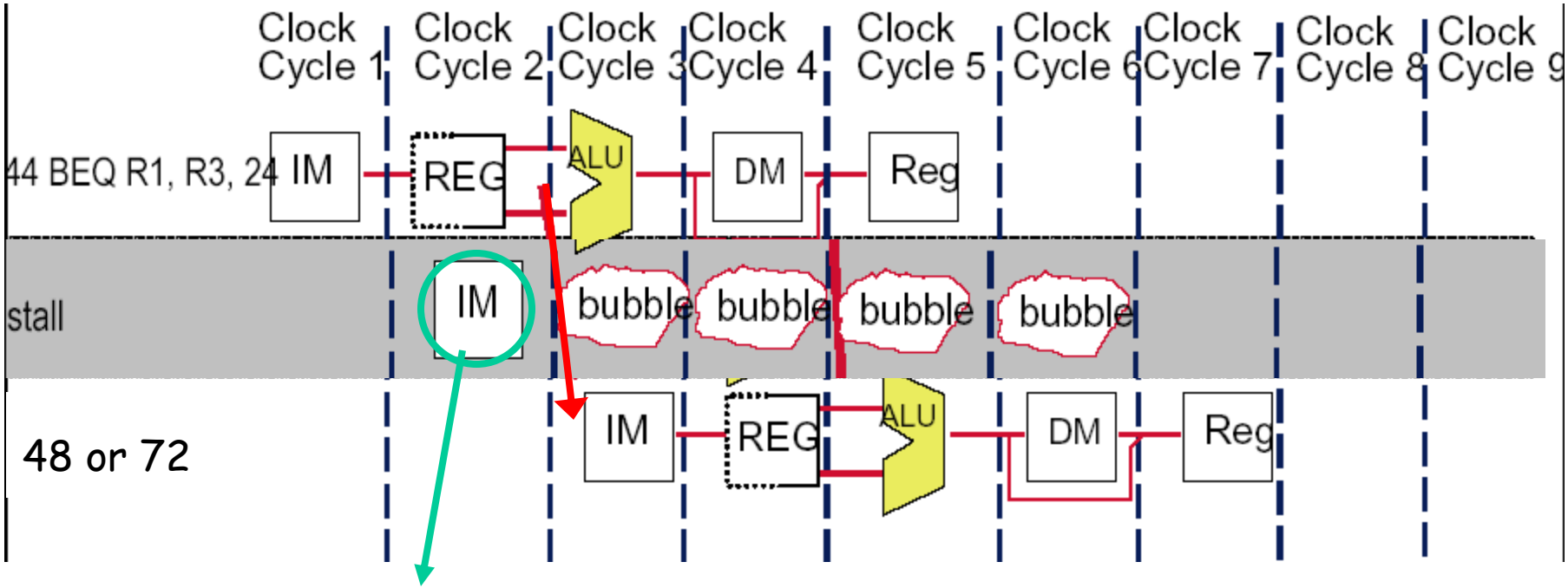
Flushing : need only to insert one stall to resolve control hazard



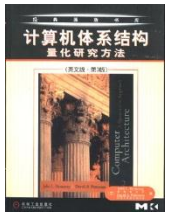
40 ADD R30,R30,R30	IF	ID	EX	MEM	WB		
44 BEQ R1, 24		IF	ID	EX	MEM	WB	
48 AND R12, R2, R5			IF	idle	idle	idle	idle
48 or 72				IF	ID	EX	MEM



Why "waste" the fetched instruction ?



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



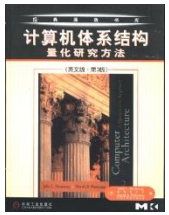
The pipeline status for predict-not-taken

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

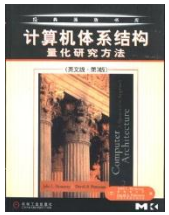
Branch is taken: 1 stall

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)



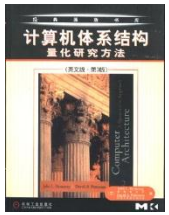
Branch delay slot

- Hence the name: branch delay slot

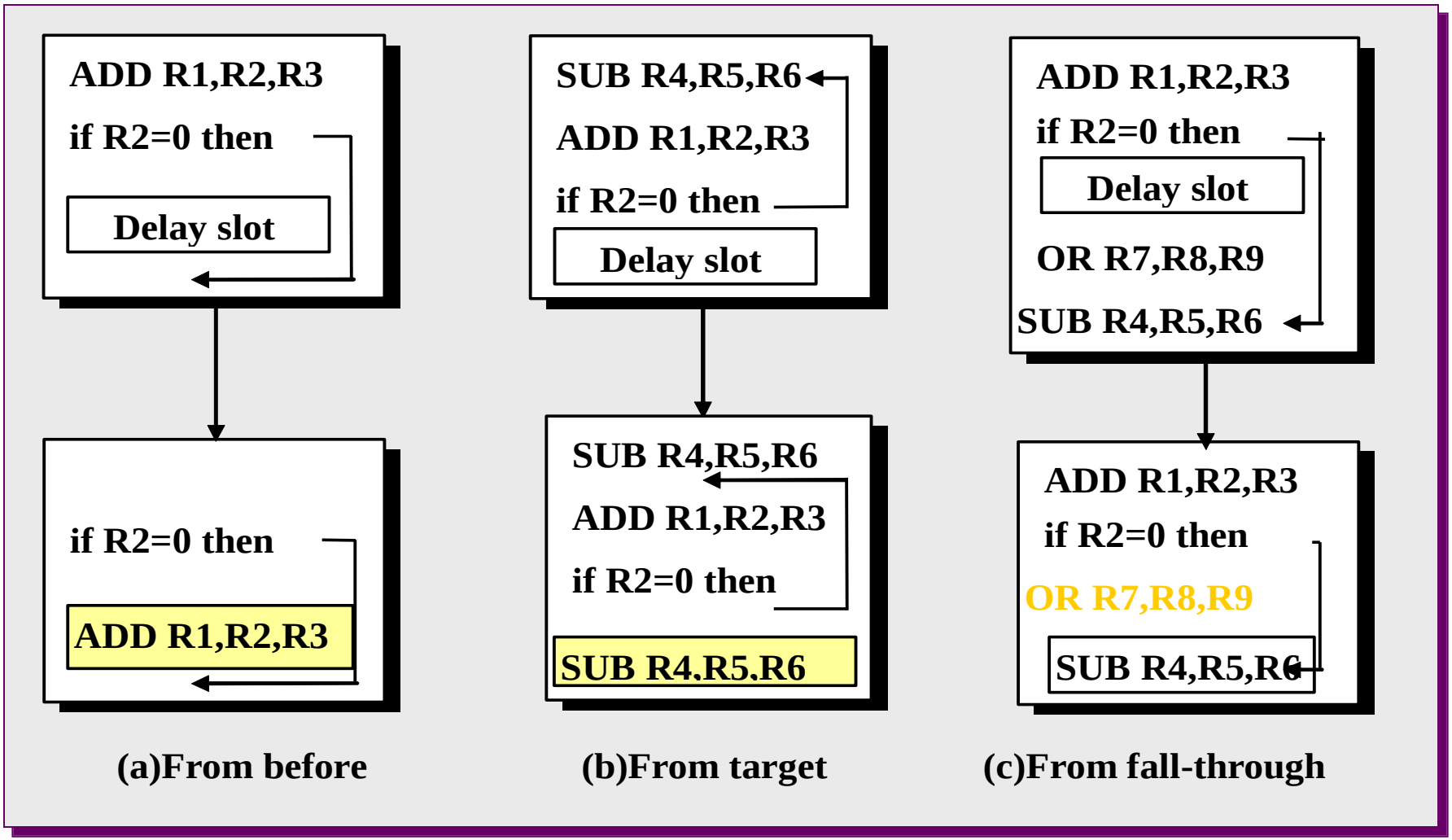
```
branch instruction
sequential successor1
sequential successor2
...
sequential successorn
branch target if taken
```

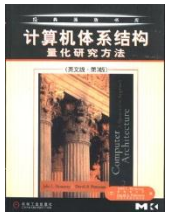
Branch delay slots

  - 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)

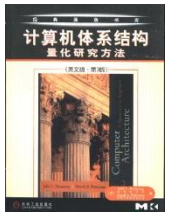
- Without Branch Delay Slot

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	...

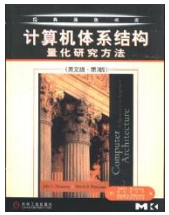
- Flow of instructions if branch is taken: 36, 40, 44, 72, ...
- Flow of instructions if branch is not taken: 36, 40, 44, 48, ...



Example: rewrite the code (b-1)

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

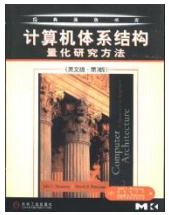
```
      LW   R2, 0(R1)
Loop: ADD  R3, R2, R4
      SW   R3, 0(R1)
      ……
      SUB  R1,R1, #4
      BNEZ R1, Loop
      LW   R2, 0(R1)
```



Example: rewrite the code (b-2)

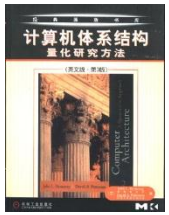
```
Loop: LW   R2, 0(R1)
      ADD  R3, R2, R4
      SW   R3, 0(R1)
      DIV  .....
      ......
      SUB  R1, R1, #4
      BNEZ R1, Loop
```

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



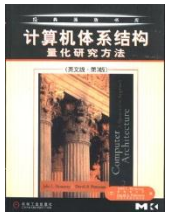
Scheduling strategy vs. performance improvement

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.



Constrains of the delayed branch

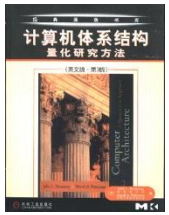
- There are restrictions on the instructions that are scheduled into the delay slots
- The compiler's ability to predict accurately whether or not a branch is taken determines how much useful work is actually done.
- For scheduling scheme b and c,
  - It must be O.K. to execute the SUB instruction if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



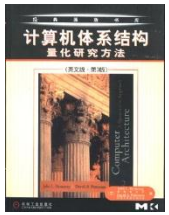
Delayed branch with cancelling (of case b)

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

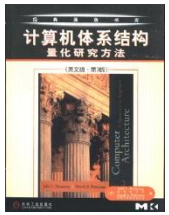
预测正确,性能没有损失

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



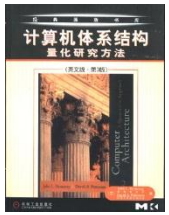
Efficiency of delayed branch

	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%



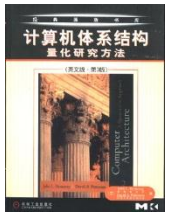
About delayed branch

- 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 four 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%



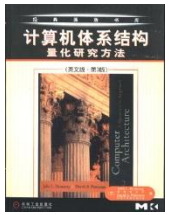
Pipeline status for various schemes

- **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)			

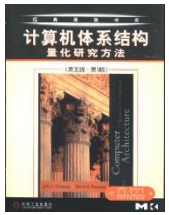
- **Predict untaken:**

	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)			



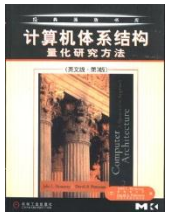
Resolution

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



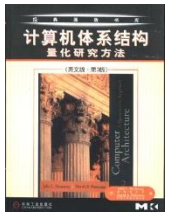
Pipeline hazards

- **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)**



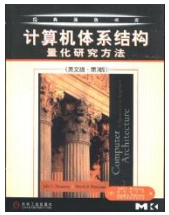
Summary for control hazard

- 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.

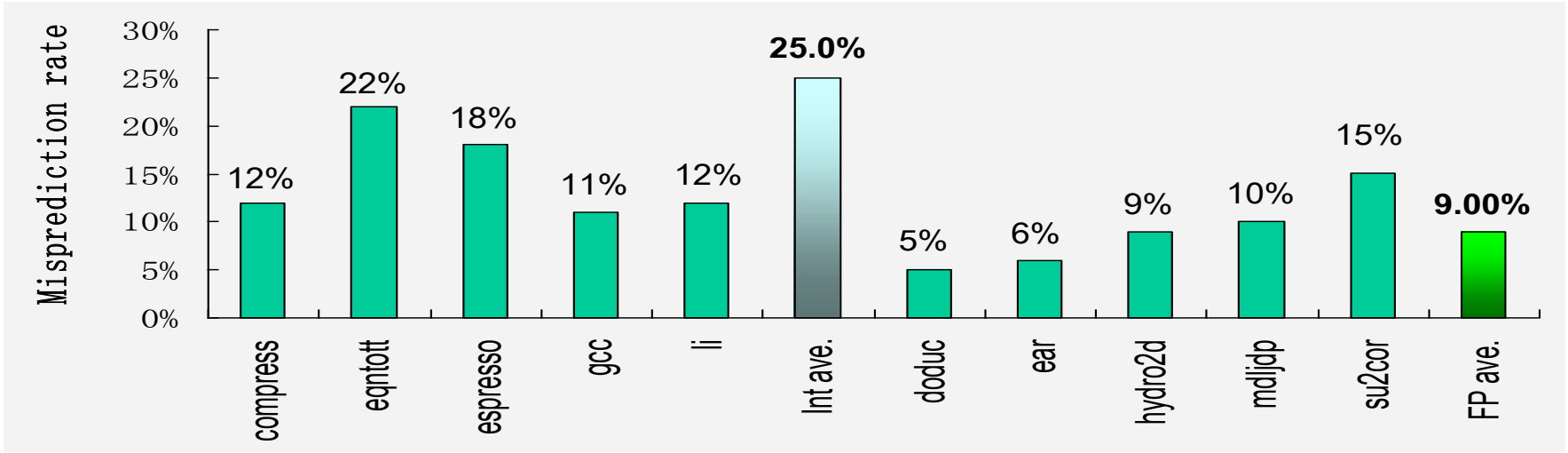


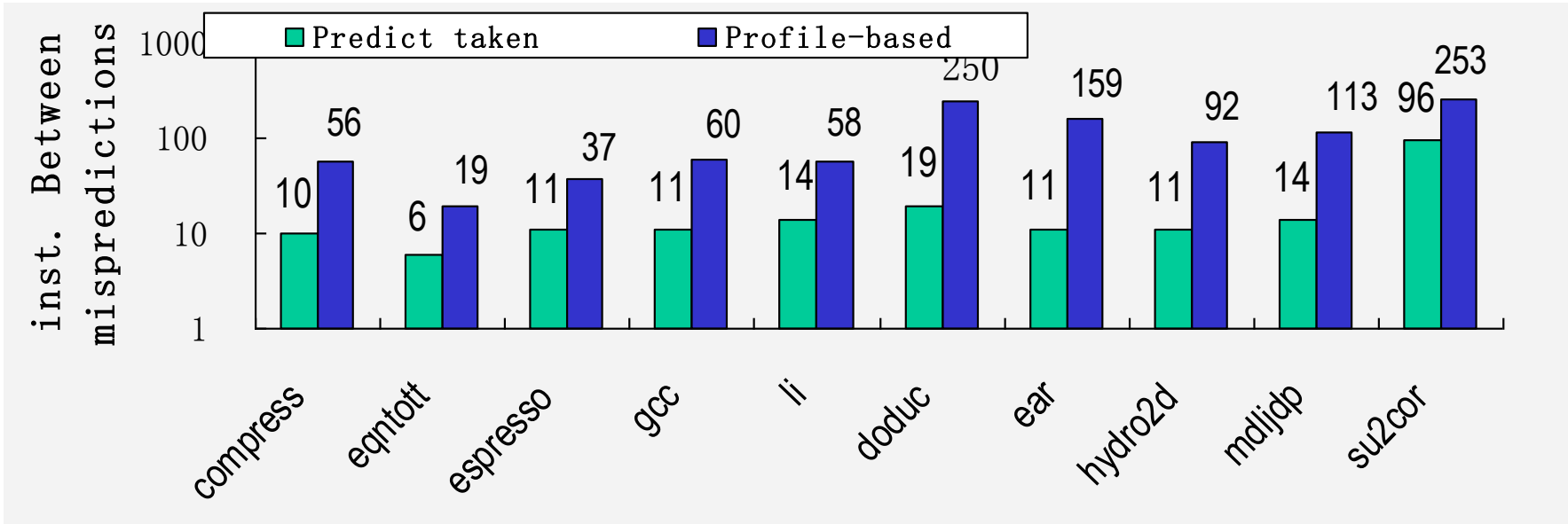
Branch prediction

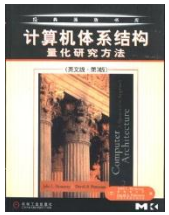
- There are many different schemes
- static branch prediction
  - Assume taken
    - √ This is surprisingly effective since 85% of backward branches and 60% of forward branches are taken.
  - Assume not taken
  - Predict by using profile information from previous run
- dynamic branch prediction by hardware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

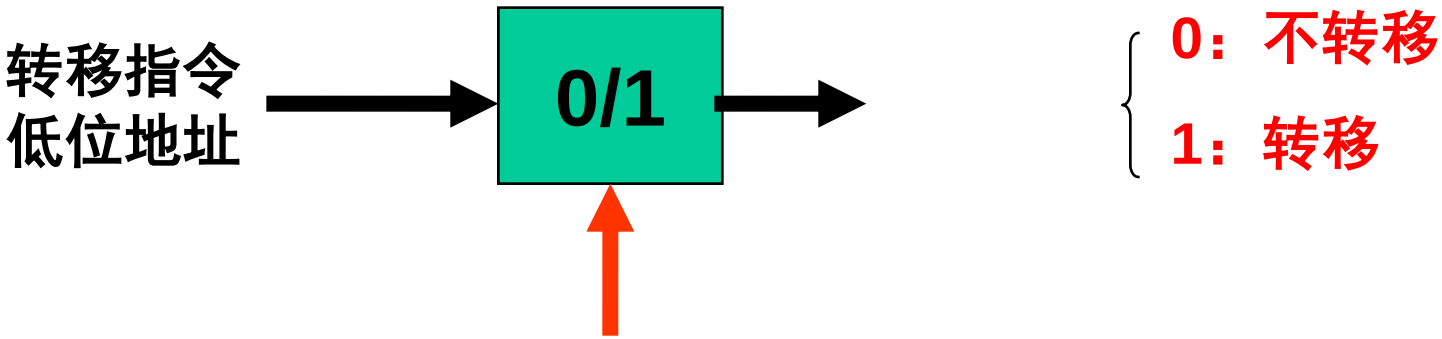


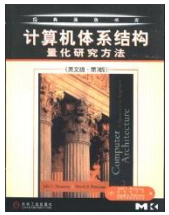
Accuracy of prediction based on profile



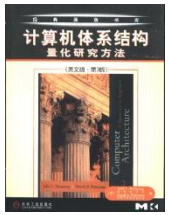



计算机体系结构
量化研究方法


转移指令
低位地址
0/1
0：不转移
1：转移
预测失败取反



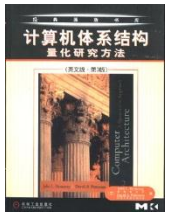

```
IF ( aa == 2)
        aa = 0;

IF ( bb == 2)
        bb = 0;

IF (aa !== bb ) {
   ……
   }
```



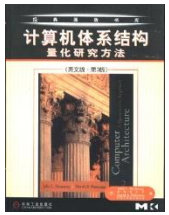

```
If (d==0)
        d=1;
if (d==1){
    ……

}
```

- **设Reg[R1] = d**

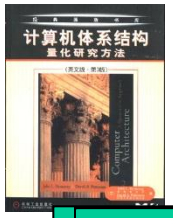
```
      BNEZ      R1, L1      ;  br b1, (d!=0)
      DADDIU  R1, R0, #1  ; d==0, so  d=1
L1: DADDIU     R3, R1, # -1  ;
      BNEZ      R3, L2       ;  br b2, (d!=1)
          ……
L2:
```



d 的初值	d==0?	B1	在 b2 以前的 d 值	d==1?	b2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	no	Taken



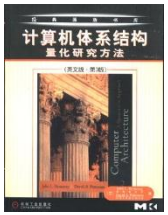
d=?	b1 预测	b1 动作	新的 b1 预测	b2 预测	b2 动作	新的 b2 预测
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

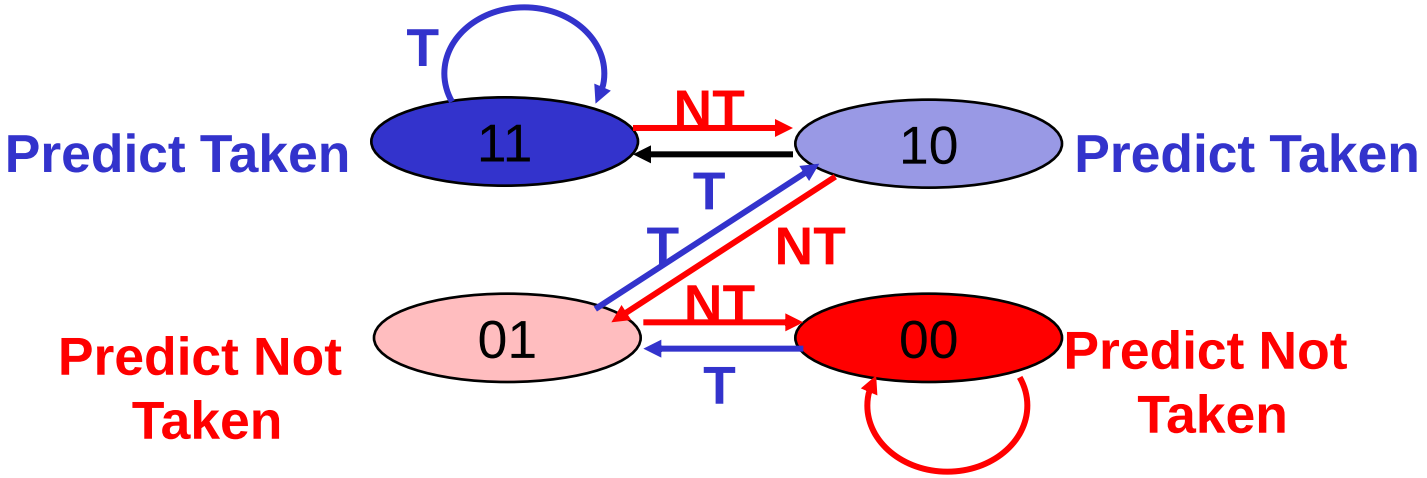


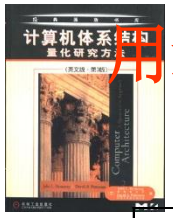
四种组合的含义：

预测组合	上一次 Br 为 NT,预测本次位(看第一位)	上一次 Br 为 T,预测本次为(看第二位)
NT/NT	NT	NT
NT/ T	NT	T
T /NT	T	NT
T / T	T	T

- 注意：
  - 这里体现了相关性
  - 虽然上一次Br指令，并非一定是本次br指令，但在简单的loop中是可能的，如简单loop中不含其它br指令。


计算机体系结构
量化研究方法


T
Predict Taken
11
NT
10
Predict Taken
T
T
NT
NT
Predict Not Taken
01
00
Predict Not Taken
T



用这种相关预测器来预测上述例子。初值为NT/NT.（第一次迭代预测错，其余均正确）

d=?	b1 预测	b1 动作	新的 b1 预测	b2 预测	b2 动作	新的 b2 预测
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

粗体字表示本次预测

预测出错

预测正确

新预测值

b1动作与b2预测值选用关系

b2动作与b1预测值选用关系