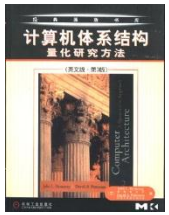
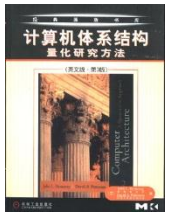


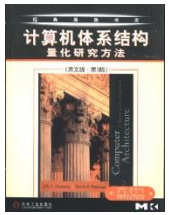
Lecture 2 for pipelining

- The pipelining hazard
- How to solve the structure hazard
- How to solve the data hazard



What we knew about pipeline

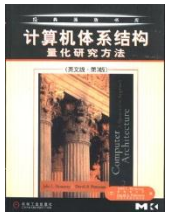
- Pipelining
  - implementation technique to execute instructions in a overlapped way to make fast CPUs(decrease CPUtime, improve throughput)
- Ideal speedup of pipeline equal to Number of pipe stages
- If the starting point is a multiple clock cycle per instruction machine then
  - pipelining decreases CPI.



Recall the MIPS 5 stage pipeline

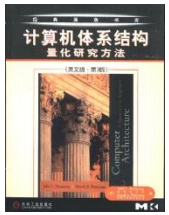
- IF (Instruction fetch cycle)
  - IR←Mem[PC];
  - NPC ←PC=PC+4;
- ID (Instruction decode/register fetch cycle)
  - A ←Regs[rs];
  - B ←Regs[rt];
  - Imm ←sign-extended immediate field of IR;

- Note: The first two stages of MIPS pipeline do the same functions for all kinds of instructions.



The third stage of MIPS pipeline

- EX (Execution/effective address cycle)
  - Memory reference:
    - √ ALUoutput ←A+Imm
  - Register-Register ALU instruction:
    - √ ALUoutput ←A func B;
  - Register-Immediate ALU instruction:
    - √ ALUoutput ←A op Imm;
  - Branch:
    - √ ALUoutput ←NPC+(Imm <<2);
    - √ Cond ←(A==0)



The last two stages of MIPS pipeline

- MEM(Memory acces/branch completion cycle)
  - Memory reference:
    - √ LMD ←Mem[ALUoutput] or
    - √ Mem[ALUoutput] ←B
  - Branch:
    - √ If (cond) PC ←ALUoutput
- WB (Write back cycle)
  - Register-Register ALU instruction
    - √ Regs[rd] ← ALUoutput;
  - Register-Immediate ALU instruction
    - √ Regs[rt] ←ALUoutput;
  - Load Instruction:
    - √ Regs[rt] ←LMD;

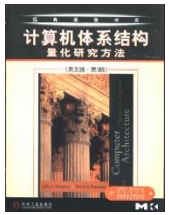
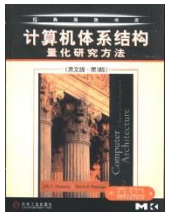
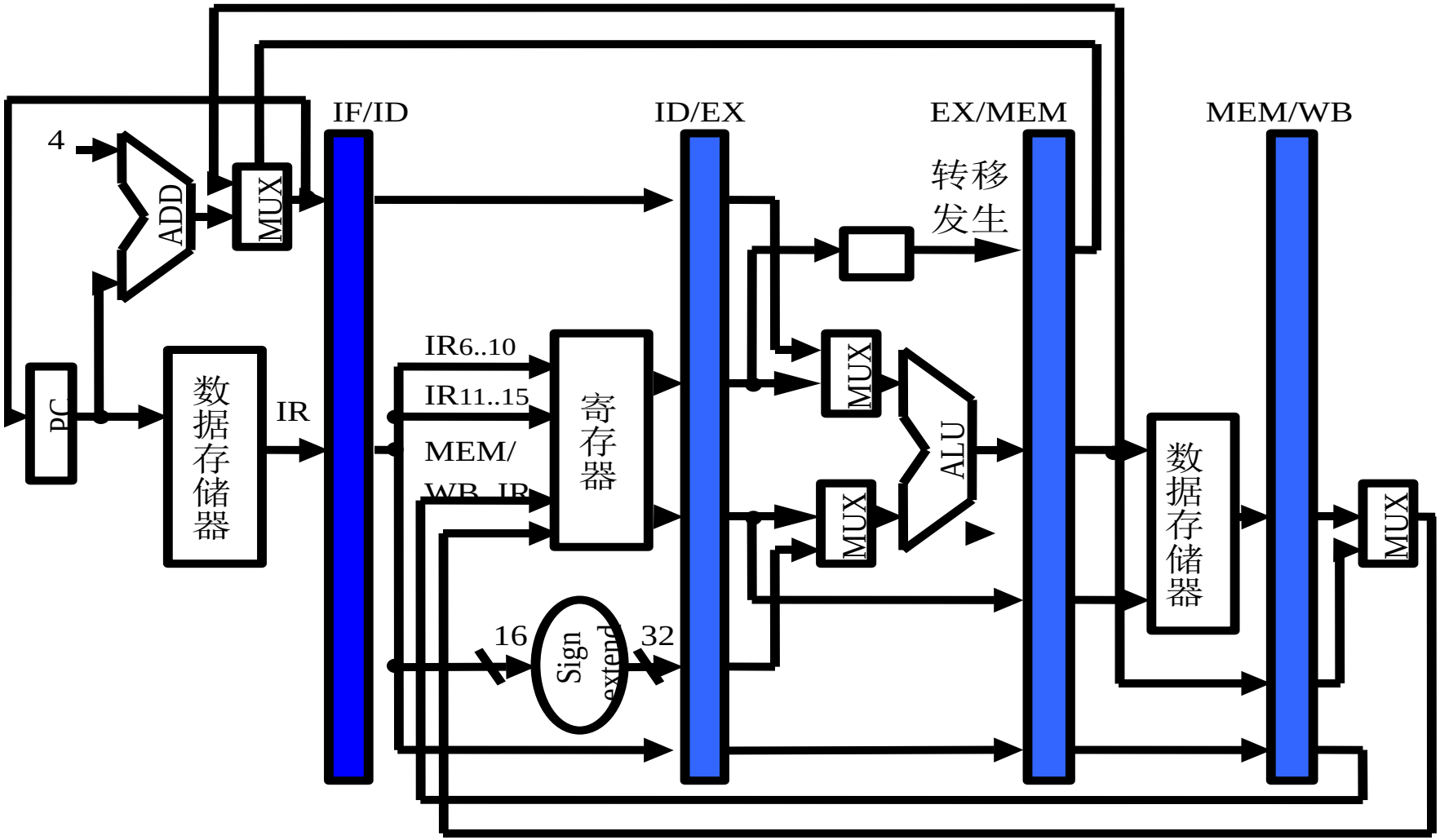


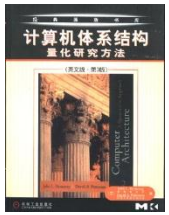
Table: Events on every stage

Stage	Any instruction		
IF	IF/ID.IR←Mem[PC]; IF/ID.NPC, PC ←(if ((EX/MEM.opcode==branch)&EX/MEM.cond) { EX/MEM.ALUoutput} else {PC+4});		
ID	ID/EX.A ←Regs[IF/ID.IR[rs]]; ID/EX.B ←Regs[IF/ID.IR[rt]]; ID/EX.NPC ←IF/ID.NPC; ID/EX.IR ←IF/ID.IR; ID/EX.Imm ←sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	Ld/st instruction	Branch instruction
EX	EX/MEM.IR ←ID/EX.IR; EX/MEM.ALUoutput ←ID/EX.A func ID/EX.B; or EX/MEM.ALUoutput ←ID/EX.A op ID/EX.Imm;	EX/MEM.IR ←ID/EX.IR; EX/MEM.ALUoutput ←ID/EX.A + ID/EX.Imm; EX/MEM.B ←ID/EX.B;	EX/MEM.ALUoutput ←ID/EX.NPC + (ID/EX.Imm<<2); EX/MEM.cond ←(ID/EX.A==0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUoutput ←EX/MEM.ALUoutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ←Mem[EX/MEM.ALUoutput]; Or Mem[EX/MEM.ALUoutput ←EX/MEM.B];	
WB	Regs[MEM/WB.IR[rd]] ←MEM/WB.ALUoutput; or Regs[MEM/WB.IR[rt]] ←MEM/WB.ALUoutput;	For Load only; Regs[MEM/WB.IR[rt]] ←MEM/WB.LMD	



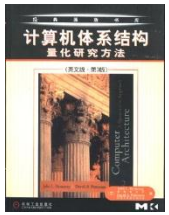
The MIPS pipelining





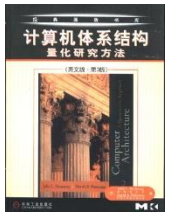
Pipeline hazard: the major hurdle

- **A hazard** is a condition that prevents an instruction in the pipe from executing its next scheduled pipe stage
- Taxonomy of hazard
  - **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



Hazards can always be resolved by Stall

- The simplest way to "fix" hazards is to **stall** the pipeline.
- Stall means suspending the pipeline for some instructions by one or more clock cycles.
- The stall delays all instructions issued after the instruction that was stalled, while other instructions in the pipeline go on proceeding.
- A pipeline stall is also called a pipeline bubble or simply bubble.
- No new instructions are fetched during a stall .



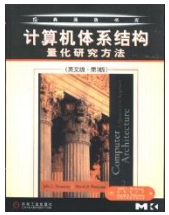
Performance of pipeline with stalls

- Pipeline stalls decrease performance from the ideal
- Recall the speedup formula:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$



Case of multi-cycle implementation

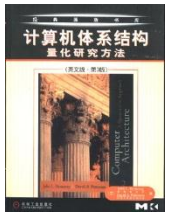
- The ideal CPI on a pipelined processor is almost always 1. (may less than or greater that)

So

$$\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clk cycles per instuction}$$

$$= 1 + \text{Pipeline stall clk cycles per instruction}$$

- Ignore the overhead of pipelining clock cycle.
- Pipe stages are ideal balanced.



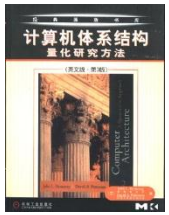
Case of multi-cycle implementation

So: Clock cycle unpipelined = Clock cycle pipelining

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

CPI unpipelined = pipeline depth

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

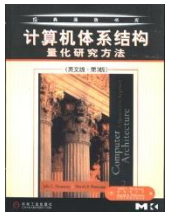


Case of single-cycle implementation

- CPI unpipelined = 1
- Clock cycle pipelined = $\frac{\text{Clock cycle unpipelined}}{\text{pipeline depth}}$

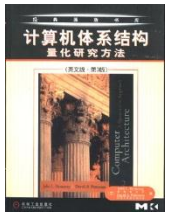
$$\text{Speedup} = \frac{1}{1 + \text{Pipeline Stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

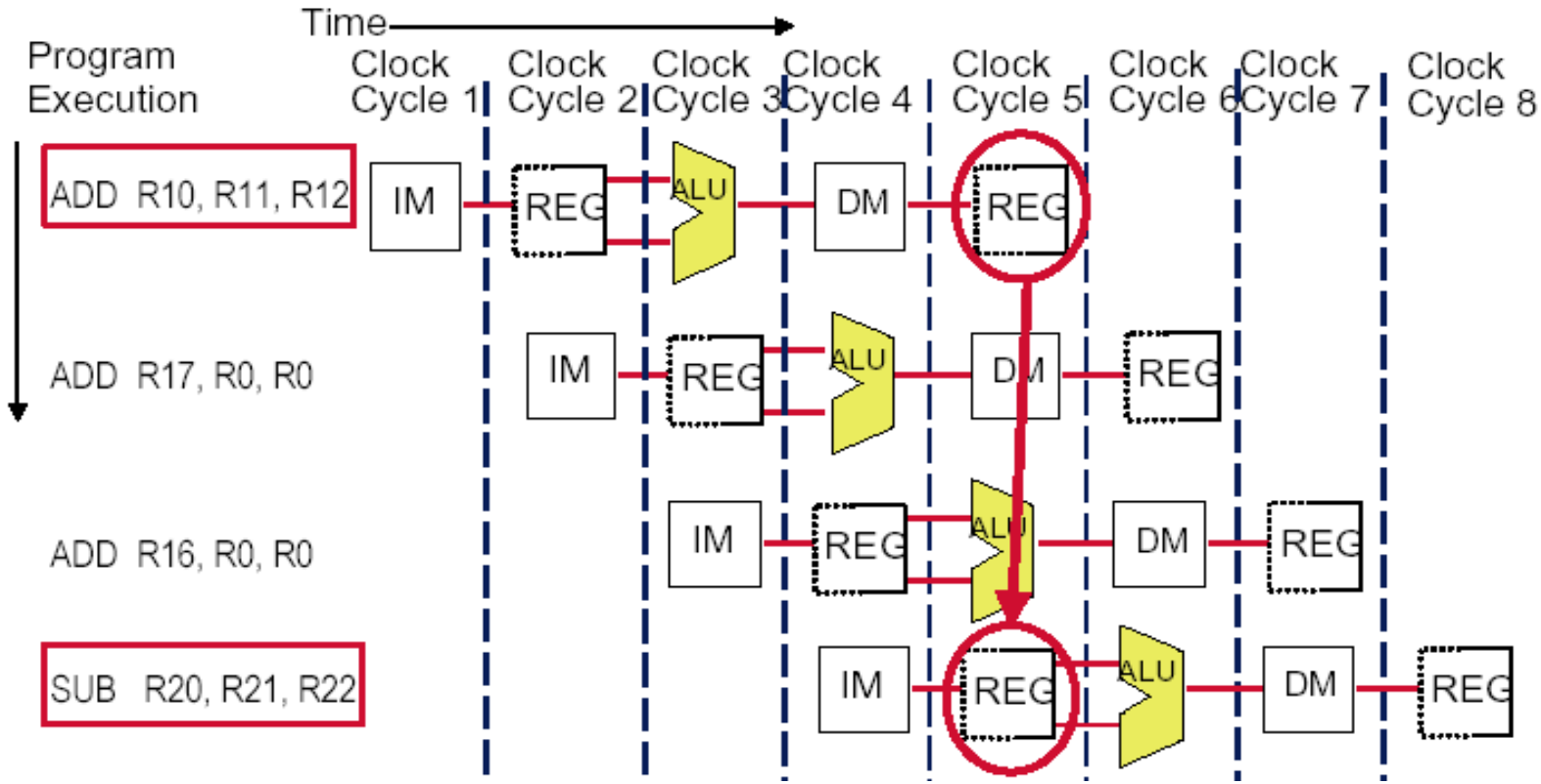


Structural hazard: Pipe Stage Contention

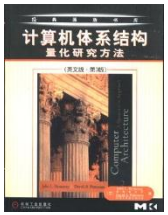
- **Structural hazards**
  - Occurs when two or more instructions want to use the same hardware resource in the same cycle
  - Causes bubble (stall) in pipelined machines
  - Overcome by replicating hardware resources
    - √ Multiple accesses to the register file
    - √ Multiple accesses to memory
    - √ some functional unit is not fully pipelined.
    - √ Not pipelined functional units



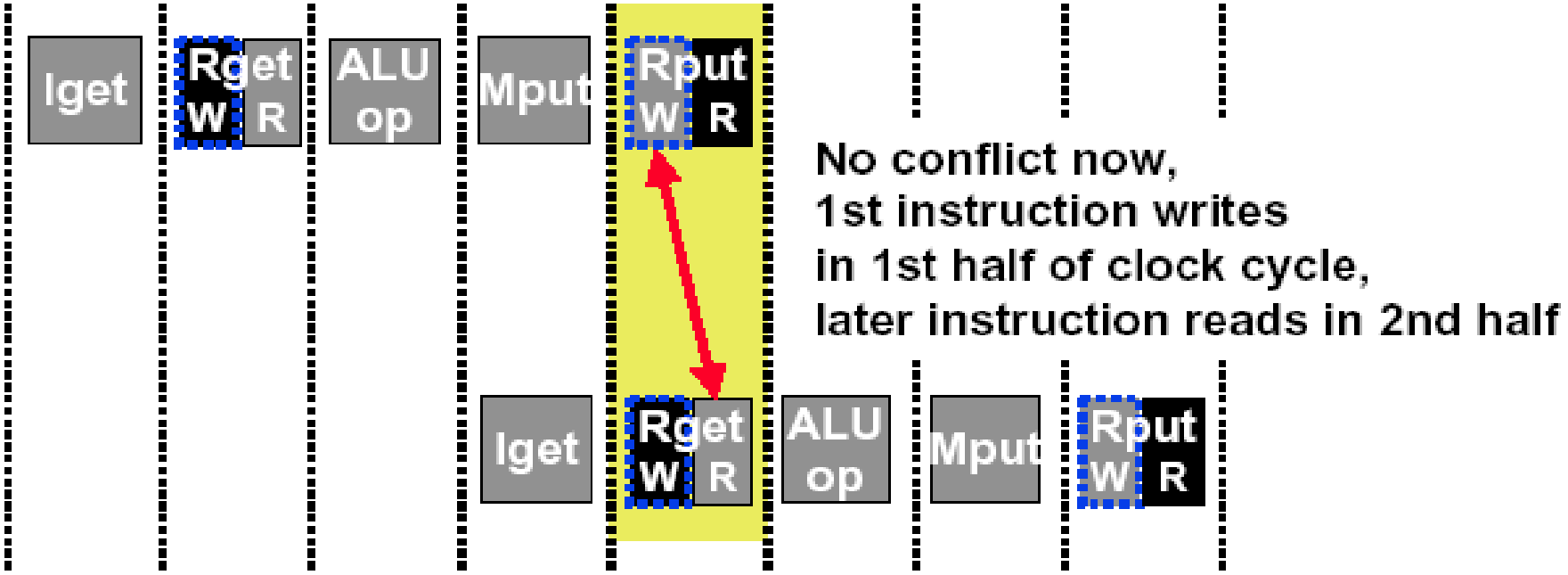
Multi access to the register file

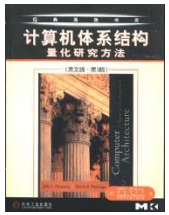


- Simply insert a stall , speedup will be decreased.
- We have resolved it with " double bump"

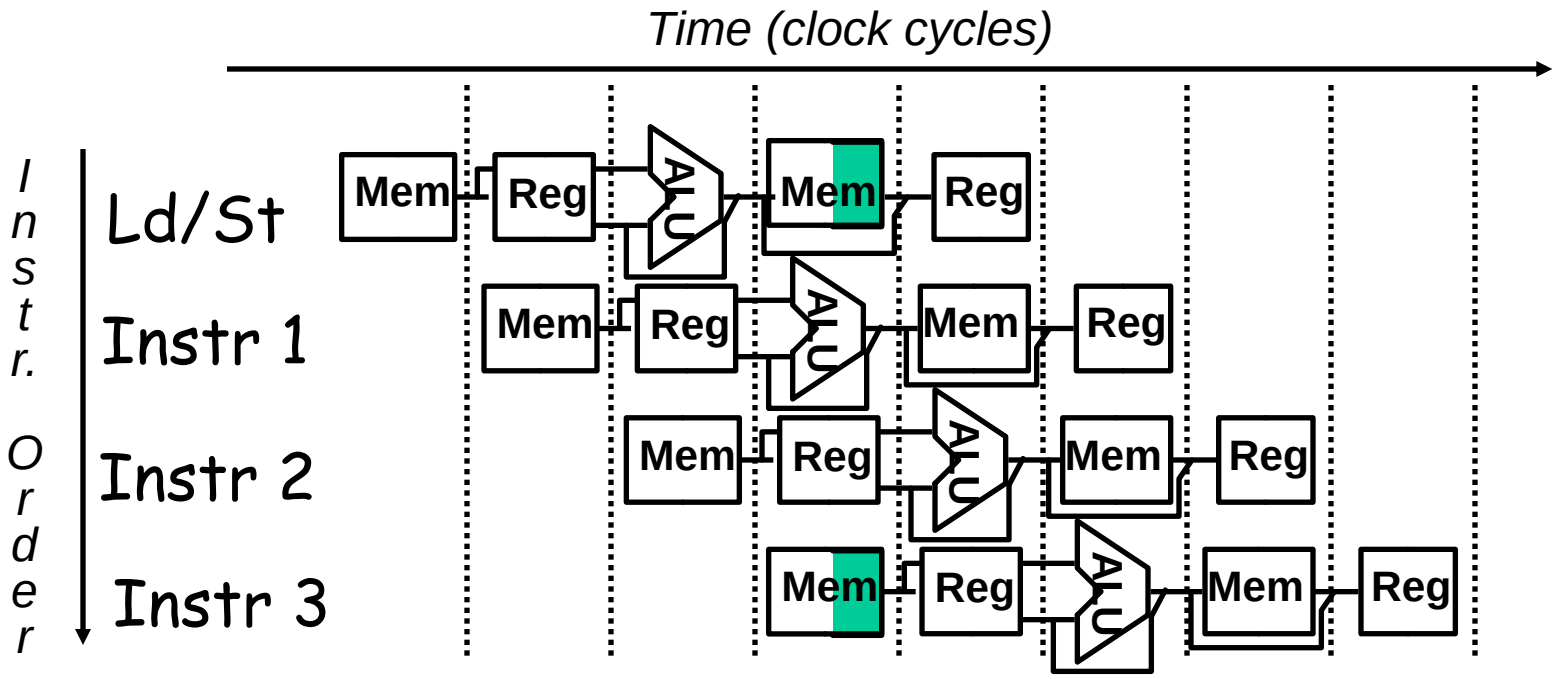


Double Bump Works !

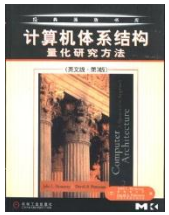




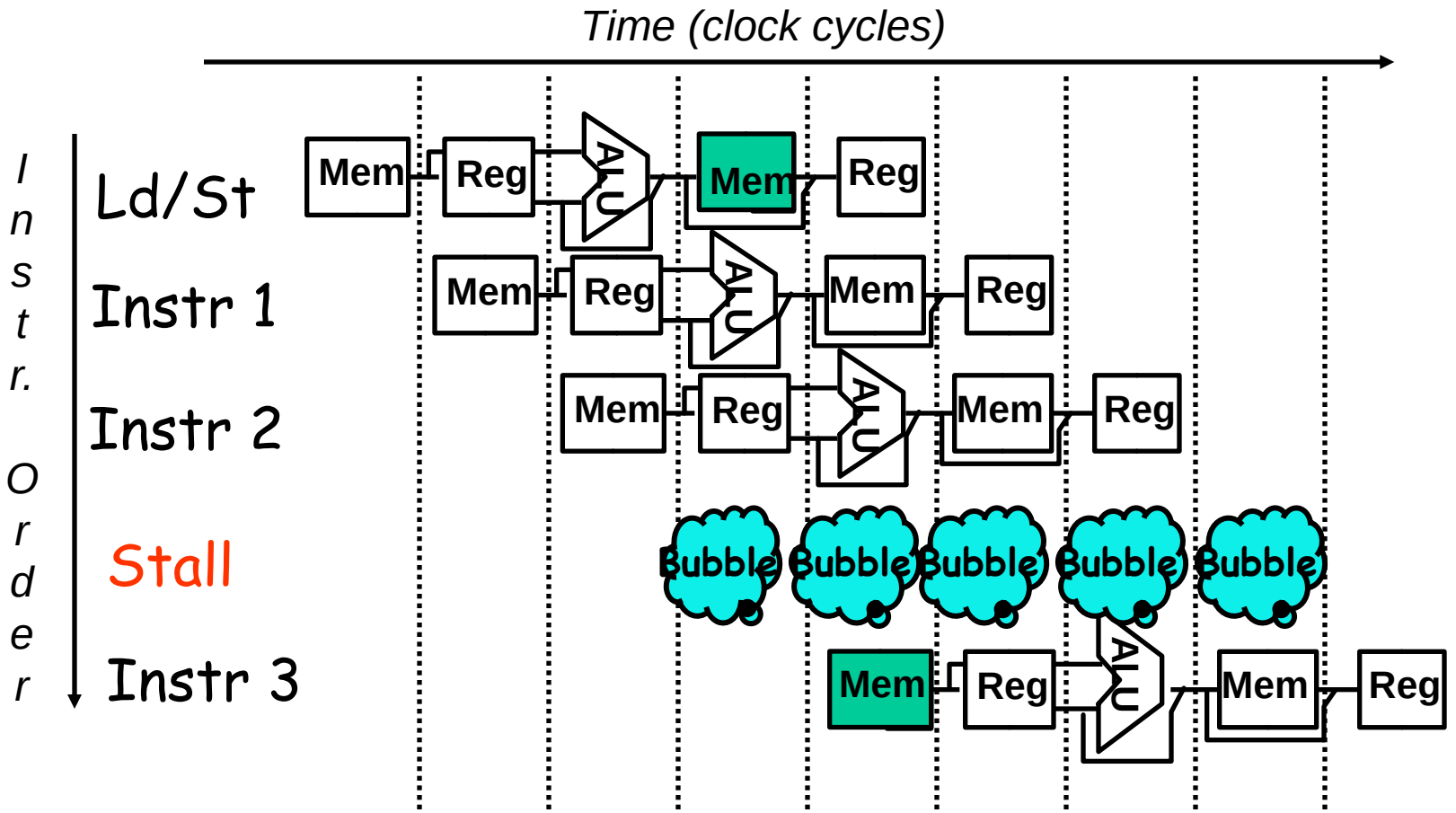
Multi access to Single Memory Port

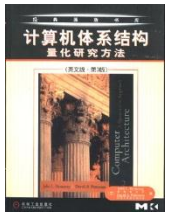


- Insert stall
- provide another memory port
- split instruction memory and data memory
- use instruction buffer

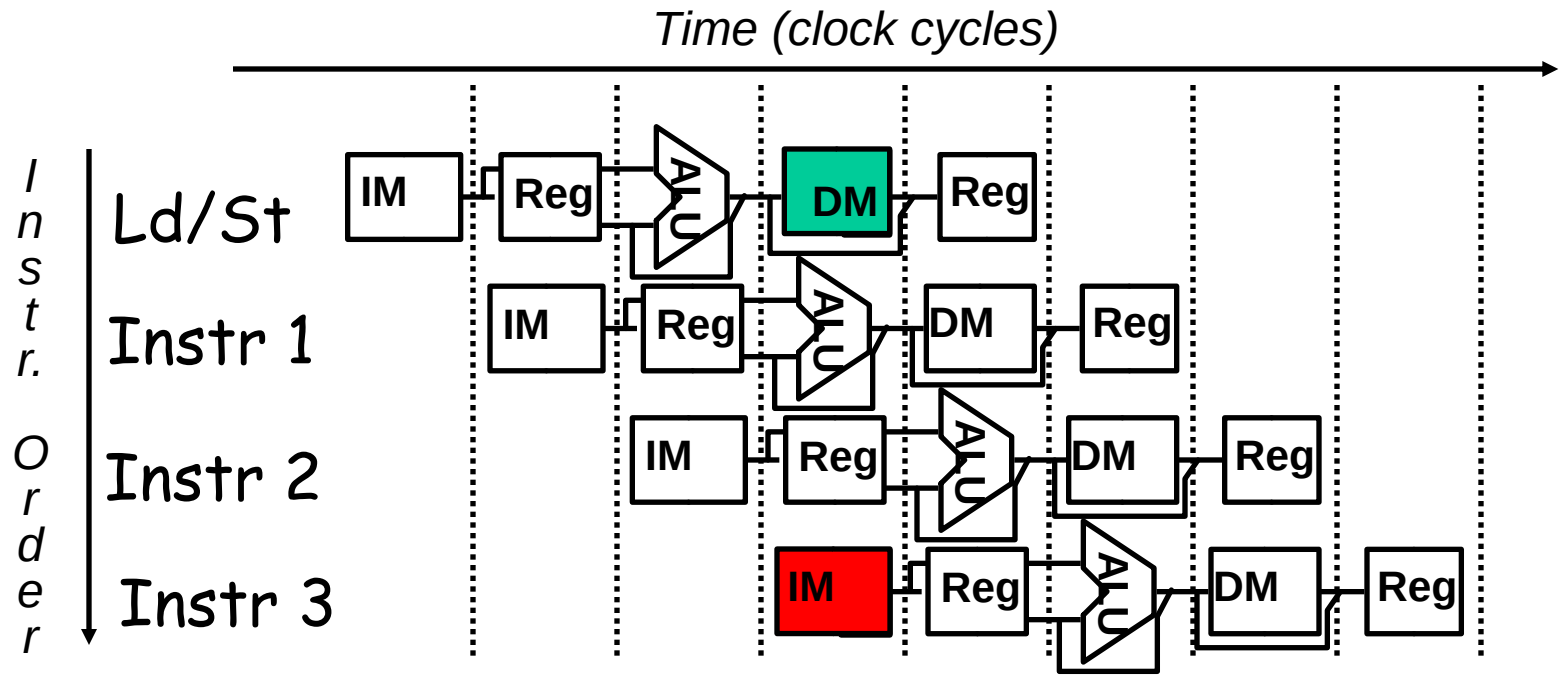


Insert Stall



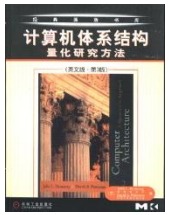


Split instruction and data memory



- Split instruction and data memory / multiple memory port / instruction buffer means:

  fetch the instruction and data inference using different hardware resources.



Not fully pipelined function unit : may cause structural hazard

Unpipelined Float Adder

ADDD	IF	ID	ADDD						WB	
ADDD		IF	ID	stall	stall	stall	stall	stall	ADDD	

Not fully pipelined Adder

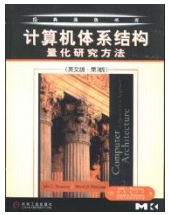
ADDD	IF	ID	A1		A2	A3	WB	
ADDD		IF	ID	stall	A1	A2	A3	

Fully pipelined Adder

ADDD	IF	ID	A1	A2	A3	A4	A5	A6	WB	
ADDD		IF	ID	A1	A2	A3	A4	A5	A6	WB

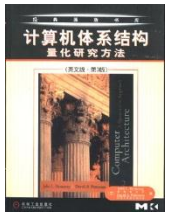
Or multiple unpipelined Float Adder

ADDD	IF	ID	ADDD1		WB	
ADDD		IF	ID	ADDD2		WB



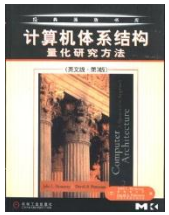
Machine *without* structural hazards will always have a lower CPI

- *Example (pA-14)*
  - *Data reference constitute 40% of the mix*
  - *Ideal CPI ignoring the structural hazard is 1*
  - *The processor with the structural hazard has a clock rate that is 1.05 times higher than that of a processor without structural hazard.*
- Answer
  - Average instruction time = CPI×Clock cycle time
    $=(1+0.4\times1)\times CC_{ideal}/1.05$
    $= 1.3 \times Cc_{ideal}$
  - Clearly, the processor without the structural hazard is faster.



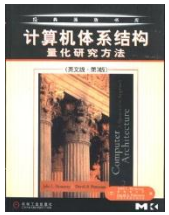
Why allow machine with structural hazard ?

- To reduce cost .
  - i.e. adding split caches, requires twice the memory bandwidth.
  - also fully pipelined floating point units costs lots of gates.
  - It is not worth the cost if the hazard does not occur very often.
- To reduce latency of the unit.
  - Making functional units pipelined adds delay
    (pipeline overhead -> registers.)
  - An unpipelined version may require fewer clocks per operation.
  - Reducing latency has other performance benefits, as we will see.



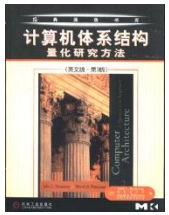
Example: impact of structural hazard to performance

- *Example*
  - *Many machines have unpipelined float-point multiplier.*
  - *The function unit time of FP multiplier is 6 clock cycles*
  - *FP multiply has a frequency of 14% in a SPECfp benchmark*
  - *Will the structural hzard have a large performance impact on the SPECfp benchmark?*



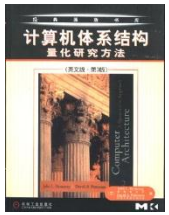
Answer to the example

- In the best case: FP multiplies are distributed uniformly.
  - There is one multiply in every 7 clock. 1/14%
  - Then there will be no structural hazard,then there is no performance penalty at all.
- In the worst case: the multiplies are all clustered with no intervening instructions.
  - Then every multiply instruction have to stall 5 clock cycles to wait for the multiplier be released.
  - The CPI will increase 70% to 1.7, if the ideal CPI is 1.
- Experiment result:
  - This structural hazard increase execution time by less than 3%.



Summary of Structural hazard

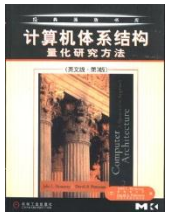
- **Taxonomy of Hazards**
  - **Structural hazards**
    - √ These are conflicts over hardware resources.
    - √ OK, maybe add extra hardware resources;
      or full pipelined the functional units;
      otherwise still have to stall
  - **Data hazards**
    - √ Instruction depends on result of prior 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



Data hazard

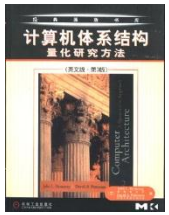
- Data hazards occur when the pipeline changes the order of read/write accesses to operands comparing with that in sequential executing .
- Let's see an Example

```
DADD R1,  R1, R3
DSUB R4,  R1, R5
AND  R6,  R1, R7
OR   R8,  R1, R9
XOR  R10, R1, R11
```

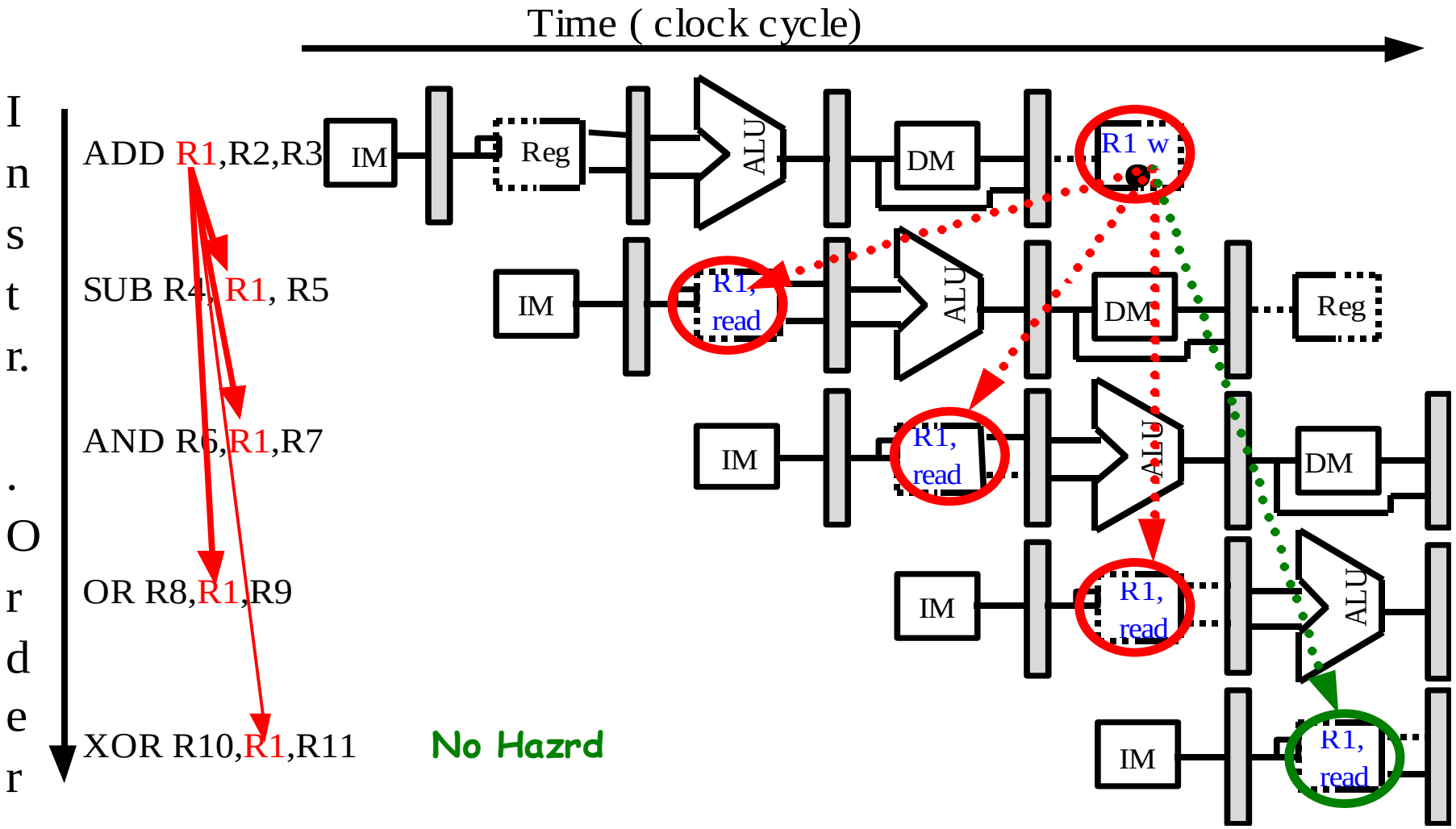


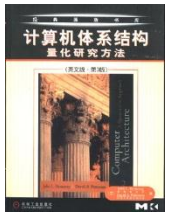
Data hazard

- Basic structure
  - An instruction in flight wants to use a data value that's not "done" yet
  - "Done" means "it's been computed" and "it's located where I would normally expect to go look in the pipe hardware to find it"
- Basic cause
  - You are used to assuming a purely sequential model of instruction execution
  - Instruction N finishes before instruction N+k, for k >= 1
  - There are dependencies now between "nearby" instructions ("near" in sequential order of fetch from memory)
- Consequence+
  - Data hazards -- instructions want data values that are not done yet, or in the right place yet

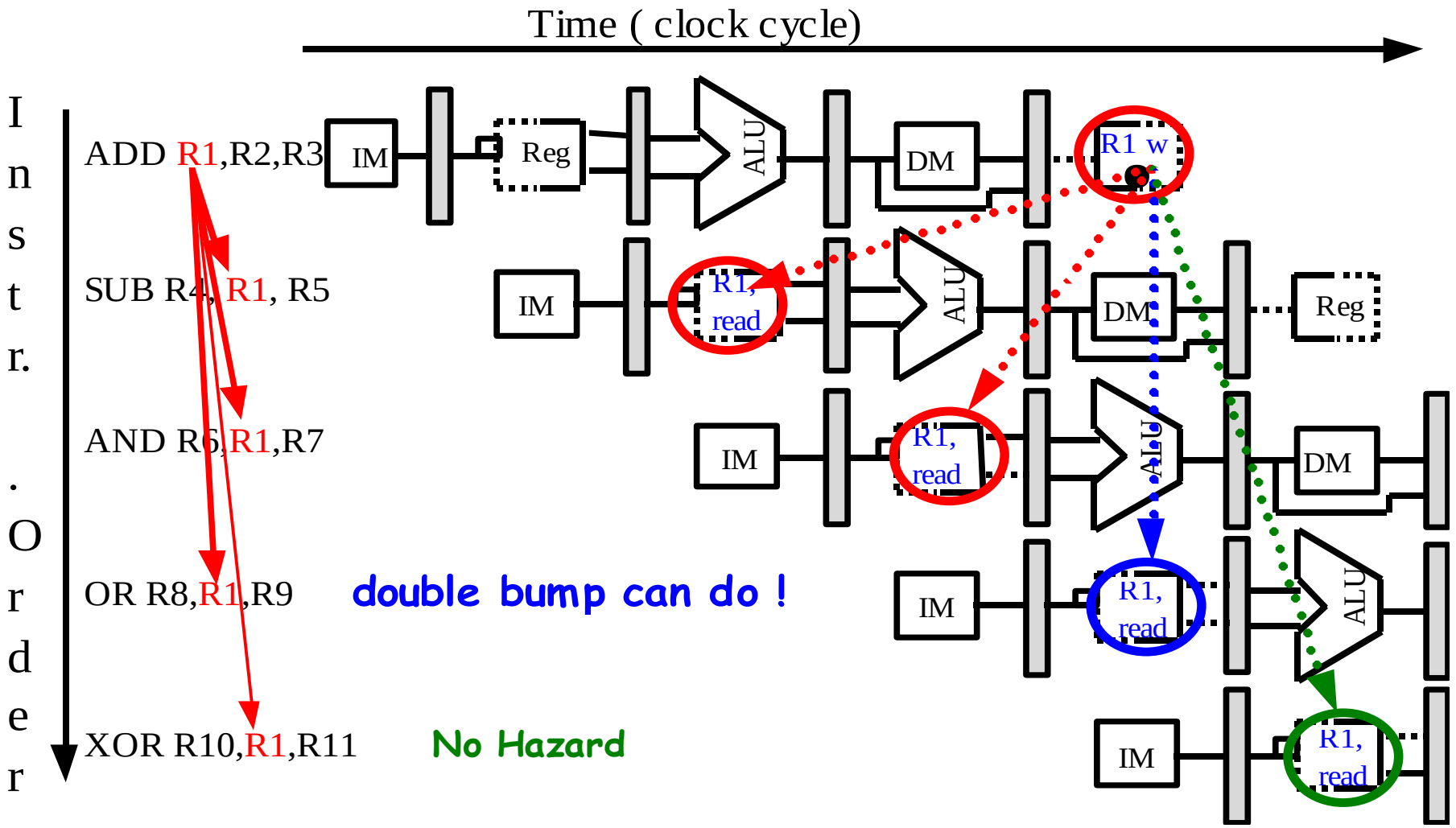


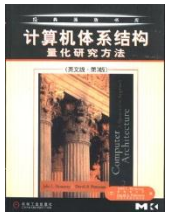
Coping with data hazards:example





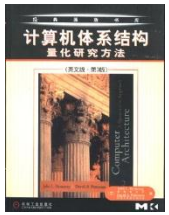
Somecases "Double Bump" can do !



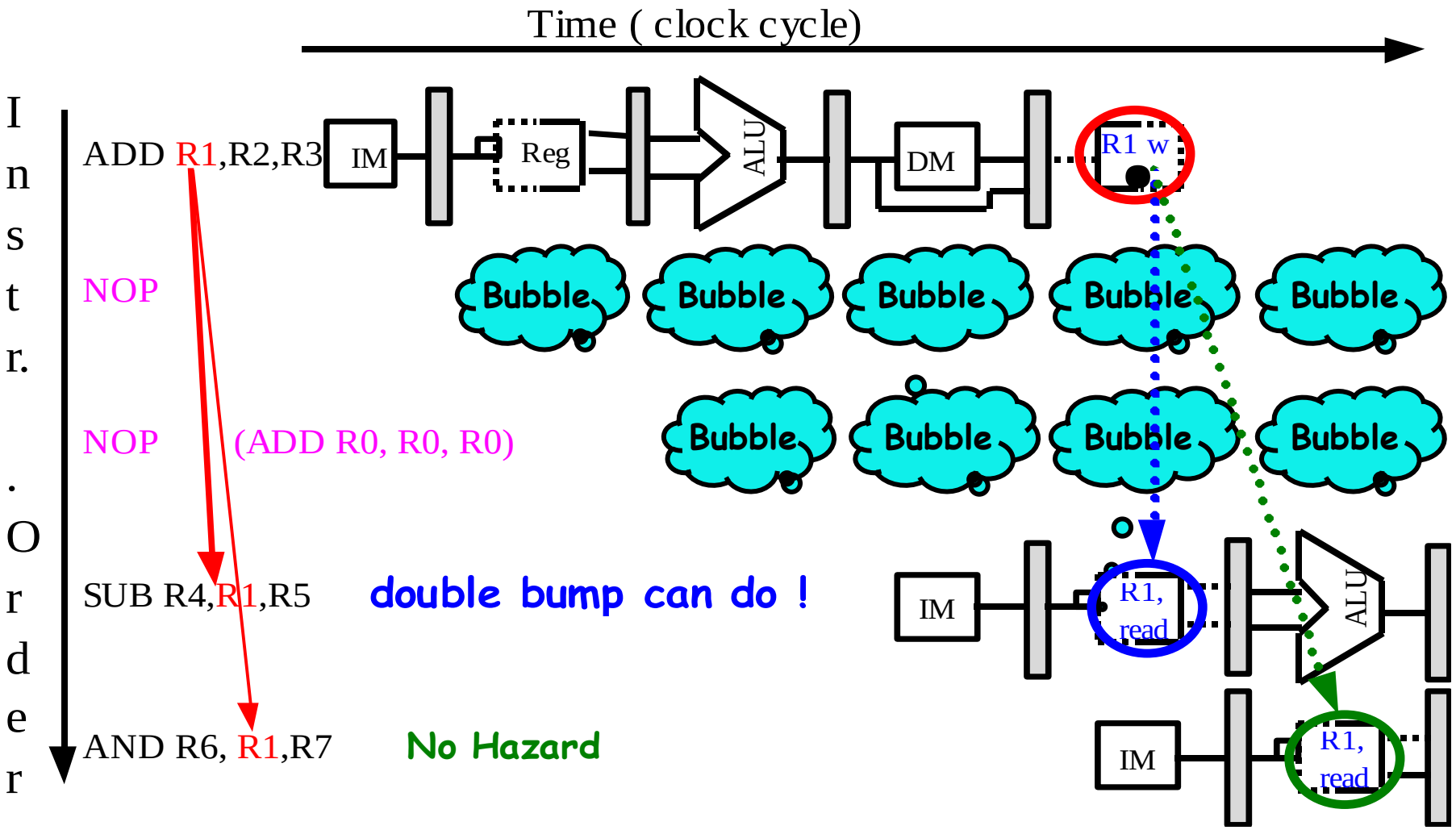


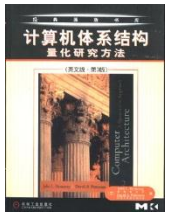
Proposed solution

- **Proposed solution**
  - **Don't** let them **overlap** like this…?
- **Mechanics**
  - Don't let the instruction flow through the pipe
  - In particular, don't let it **WRITE** any bits anywhere in the pipe hardware that represents **REAL** CPU state (e.g., register file, memory)
  - Let the instruction wait until the hazard resolved.
  - Name for this operation: **PIPELINE STALL**



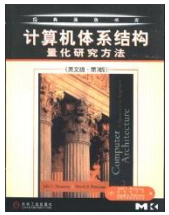
How do we stall ?
Insert nop by compiler



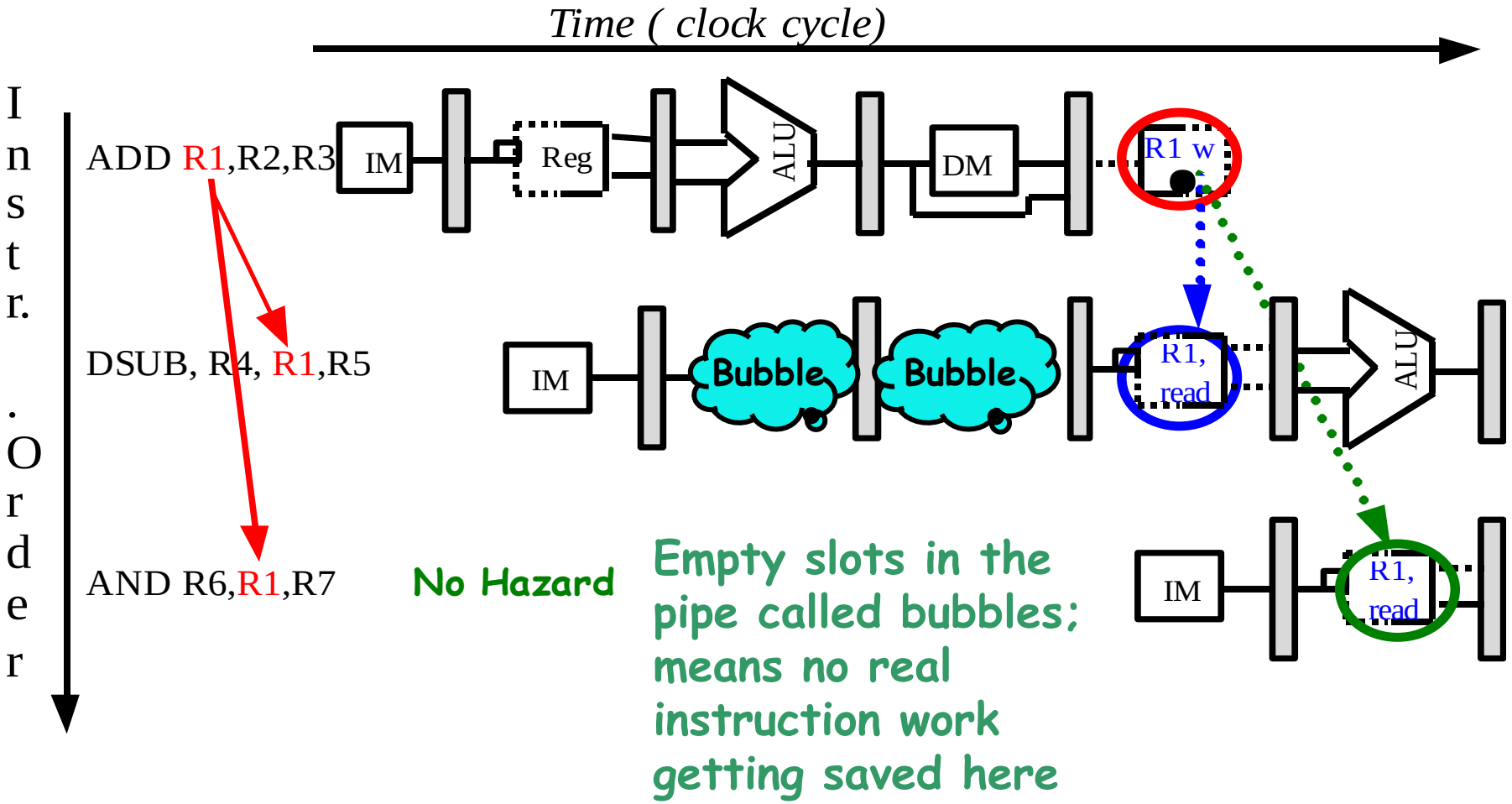


How do we stall? Add hardware Interlock !

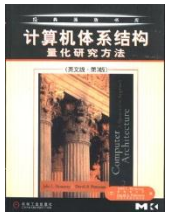
- Add extra hardware to detect stall situations
  - Watches the instruction field bits
  - Looks for "read versus write" conflicts in particular pipe stages
  - Basically, a bunch of careful "case logic"
- Add extra hardware to push bubbles thru pipe
  - Actually, relatively easy
  - Can just let the instruction you want to stall GO FORWARD through the pipe…
  - …but, TURN OFF the bits that allow any results to get written into the machine state
  - So, the instruction "executes" (it does the work), but doesn't "save"



Interlock: insert stalls

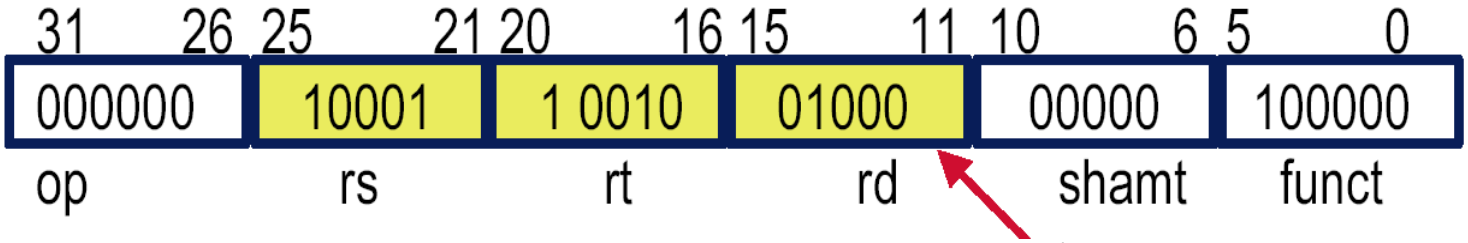


How the interlock is implementated ?

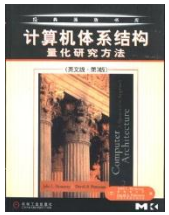


Recall MIPS Instruction format

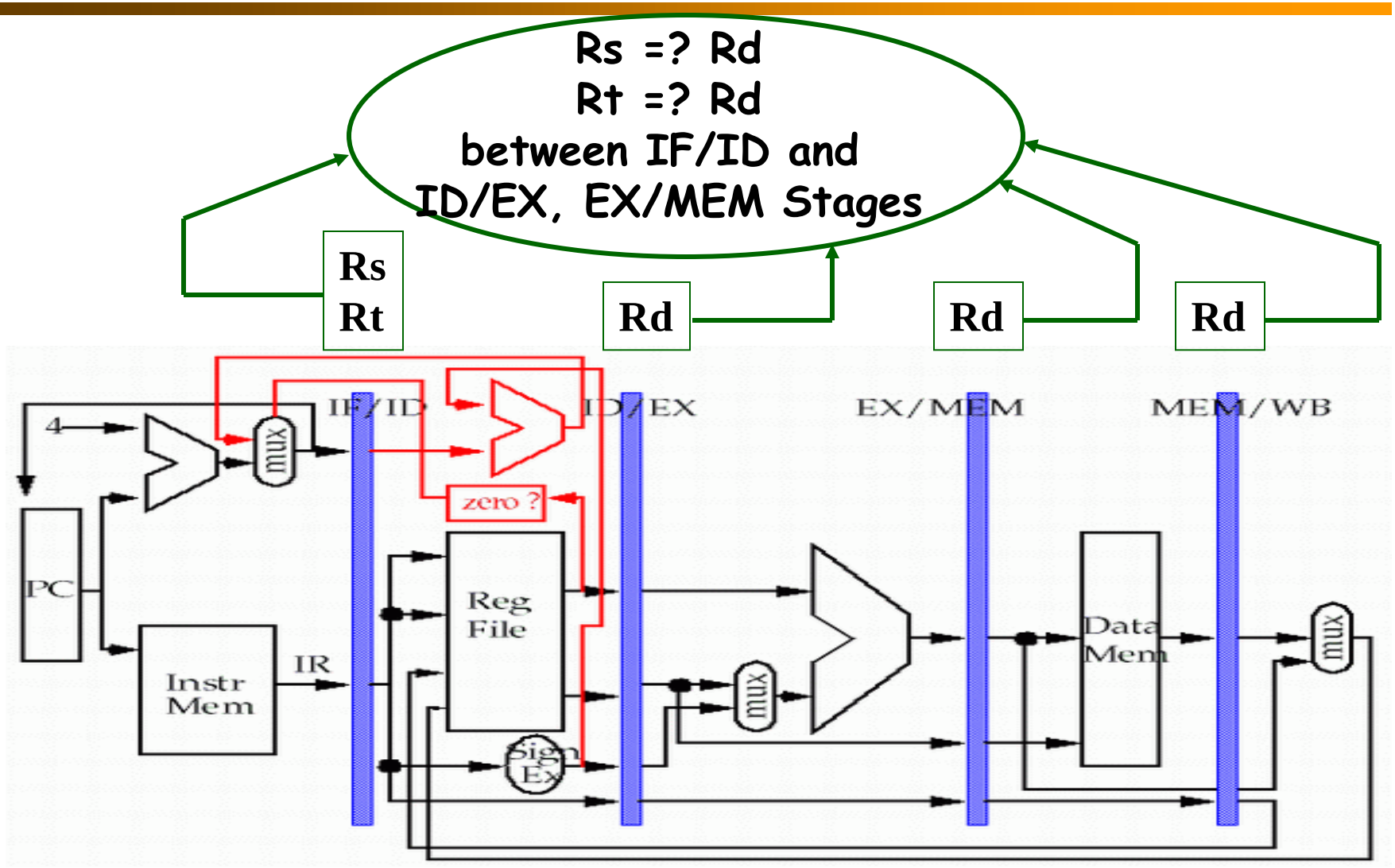
- add R8, R17, R18
  - is stored in binary format as
  - 00000010 00110010 01000000 00100000

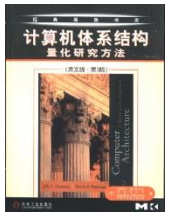


- MIPS lays out instructions into "fields"
  - op operation of the instruction
  - rs first register source operand
  - rt second register source operand
  - rd register destination operand
  - shamt shift amount
  - funct function (select type of operation)



Detect: Data Hazard Logic

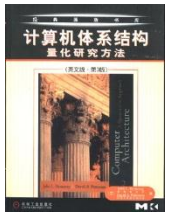




Example

DSUB	R2, R1, R3	Rd = R2	Rs = R1	Rt = R3
AND	R12, R2, R5	Rd = R12	Rs = R2	Rt = R5
OR	R13, R6, R2	Rd = R13	Rs = R6	Rt = R2
DADD	R14, R2, R2	Rd = R14	Rs = R2	Rt = R2
SW	R15, 100(R2)	Rd = R15	Rs = R2	Rt = XX

- SUB-AND Hazard
  - ID/EX.RegRd(sub) == IF/ID. RegRs(and) == R2
- SUB-OR Hazard
  - EX/MEM.RegRd(sub) == IF/ID. RegRt(or) == R2
- AND-OR: No Hazard
  - ID/EX.RegRd(and)==R12 ≠ IF/ID.RegRt Or IF/ID.RegRs

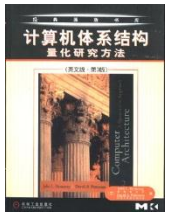


How to delay the instruction ?

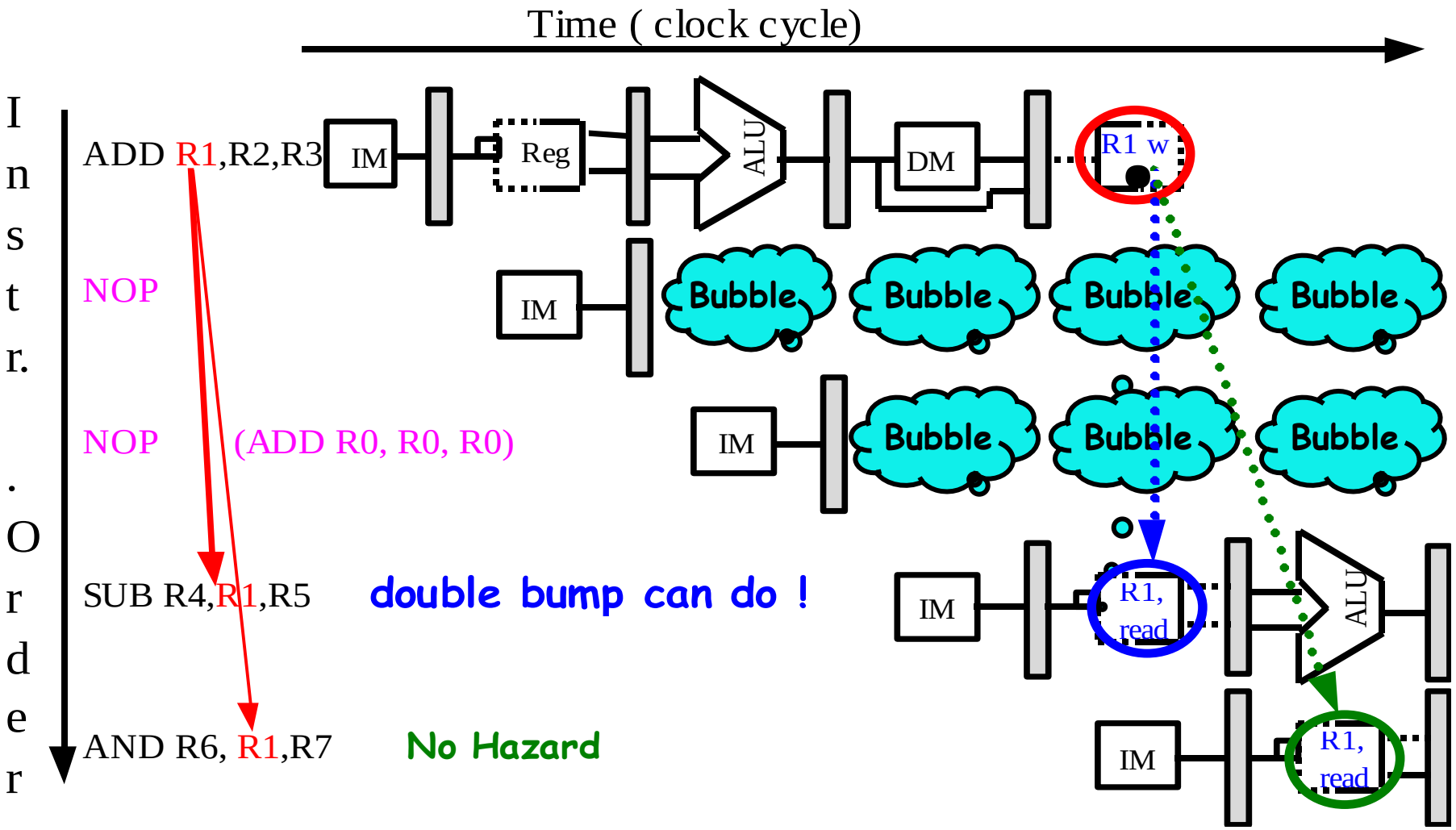
- The Interlock can simulate the NOP:

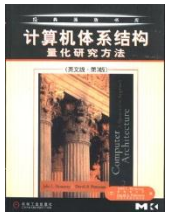
  Once it is detected need to add a stall, then

  - Clear the ID/EX.IR to be the instruction of NOP.
  - Reserve the IF/ID.IR unchanged for one more clock cycle.



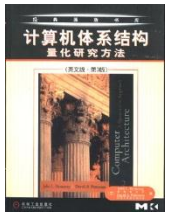
Hardware simulates NOP





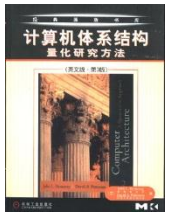
Forwarding: reduce data hazard stalls

- If the result you need does not exist AT ALL yet,
  - you are out of luck, sorry.
- But, what if the result exists, but is not stored back yet?
  - Instead of stalling until the result is stored back in its "natural" home…
  - **grab the result "on the fly" from "inside" the pipe, and send it to the other instruction (another pipe stage) that wants to use it**



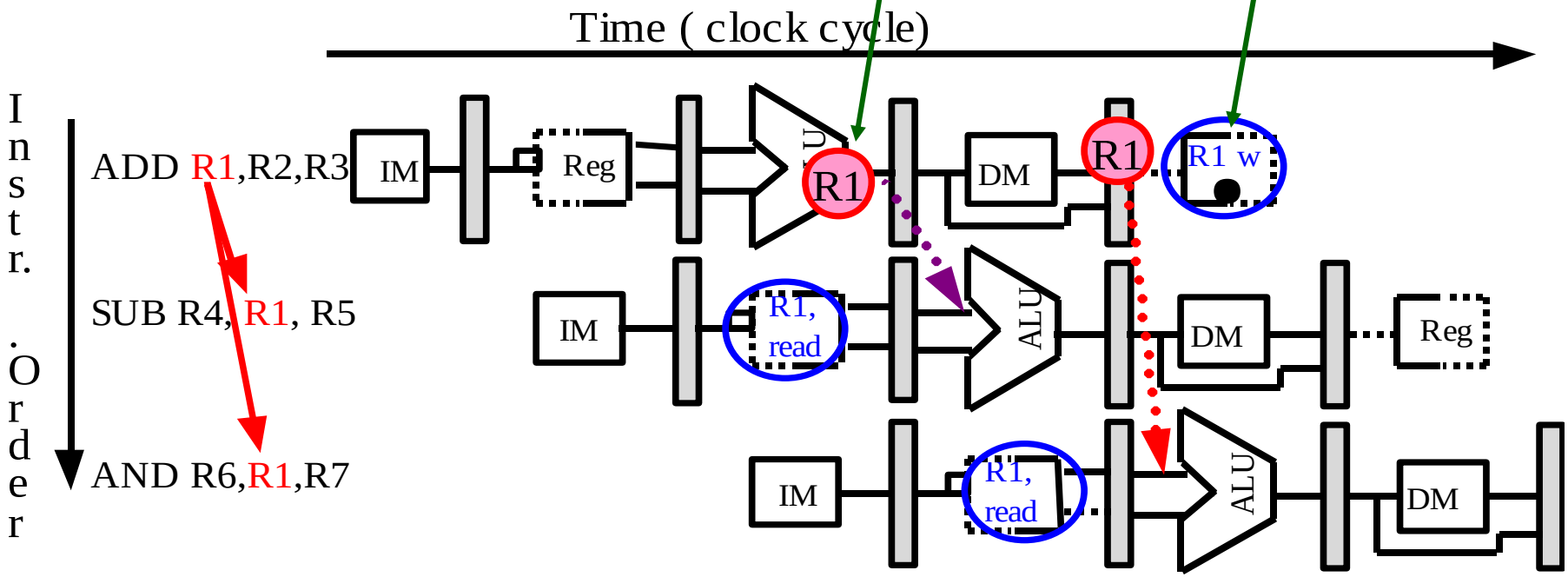
Forwarding

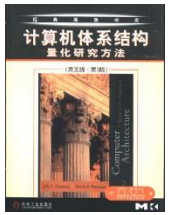
- Generic name: forwarding (bypass, short-circuiting)
  - Instead of waiting to store the result, we forward it immediately (more or less) to the instruction that wants it
  - Mechanically, we add buses to the datapath to move these values
  - around, and these buses always "point backwards" in the datapath, from later stages to earlier stages



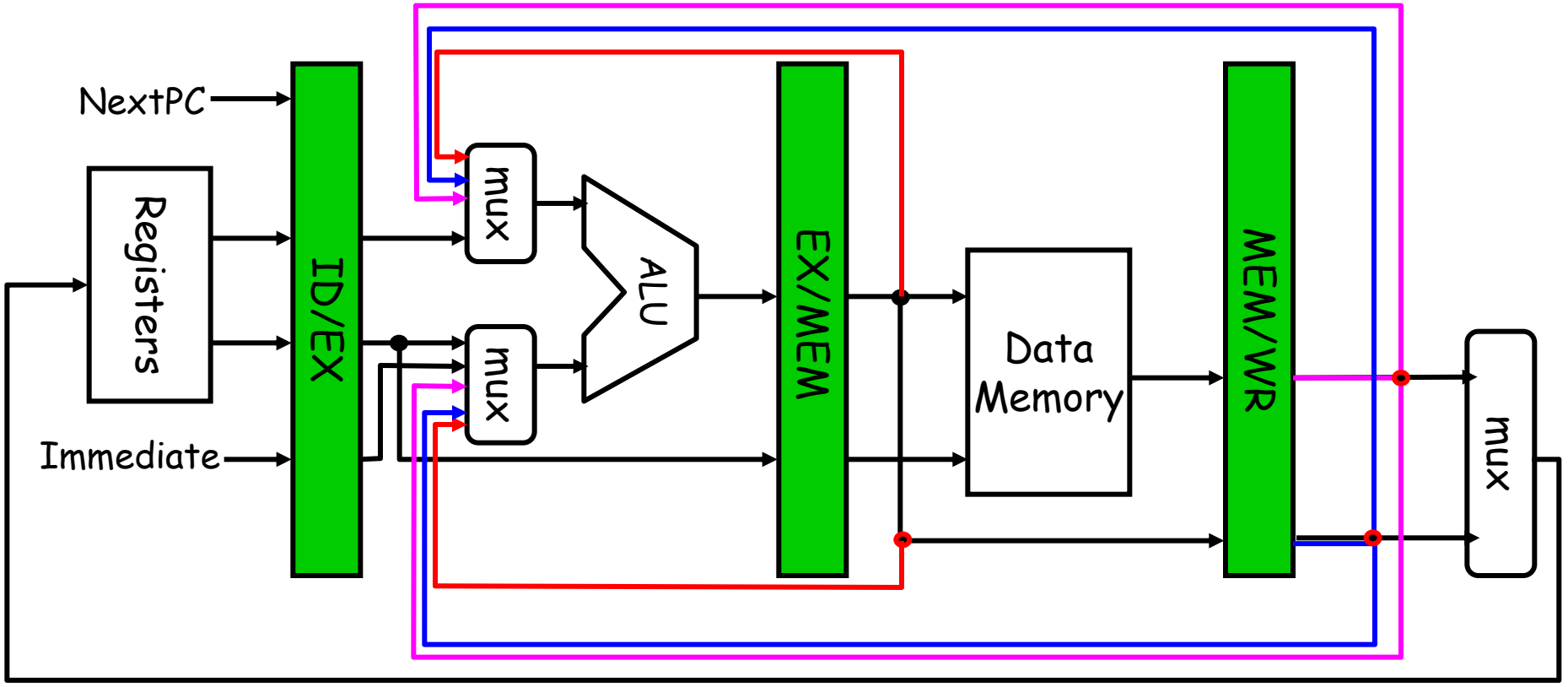
Forwarding: reduce data hazard stalls

- Data may be already computed - just not in the Register File





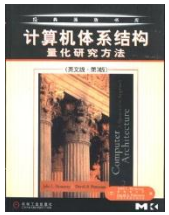
Hardware Change for Forwarding



EX/Mem.ALUoutput → ALU input

MEM/WB.ALUoutput → ALU input

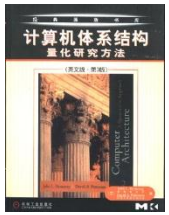
MEM/WB.LMD → ALU input



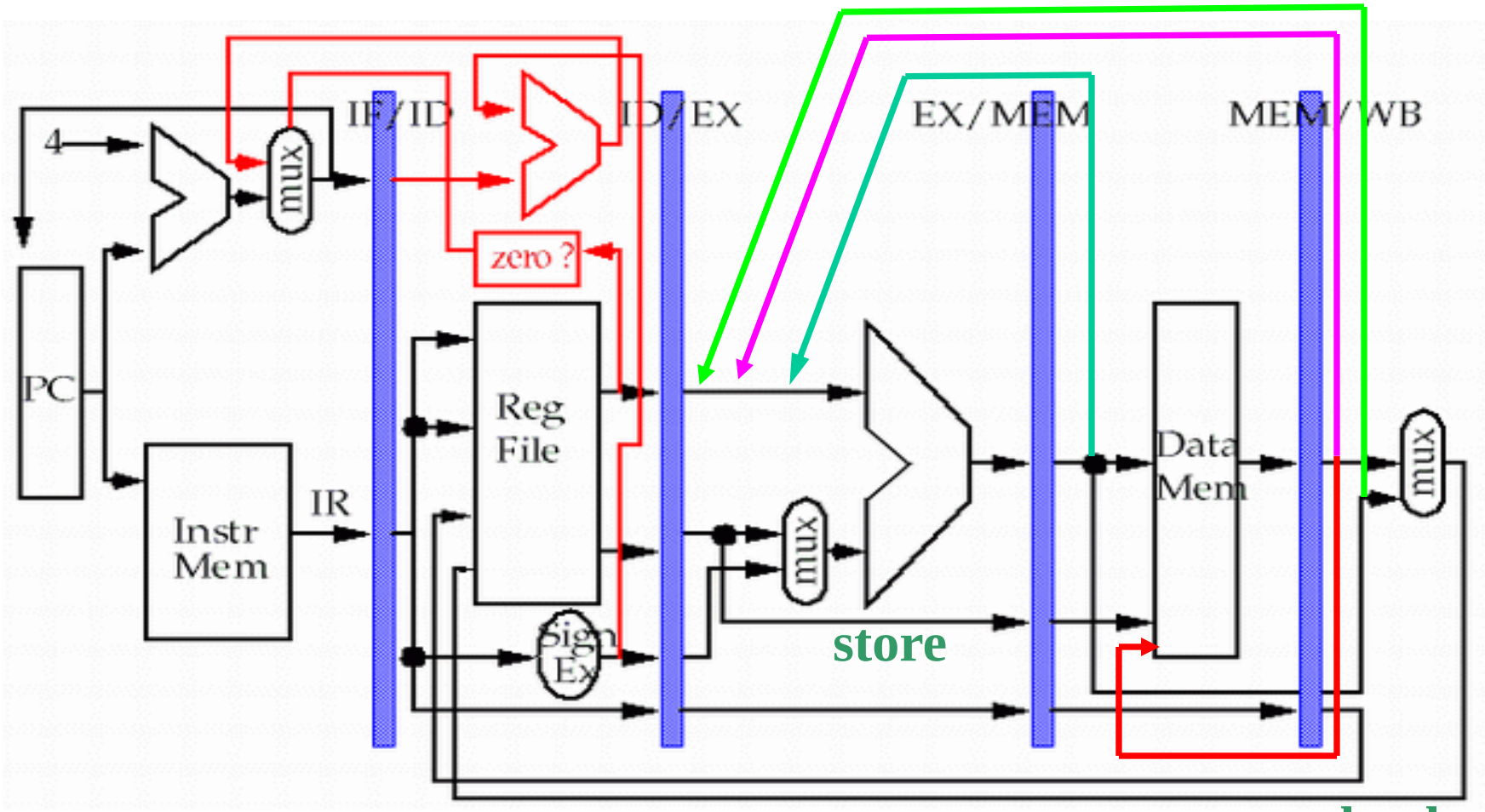
How to select the forwarding path: the forwarding logic

- P161 in Edition 2; PA-36 in Edition 3

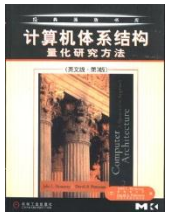
Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR [rd] == ID/EX.IR [rs]
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR [rd] == ID/EX.IR [rt]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR [rd] == ID/EX.IR [rs]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR [rd] == ID/EX.IR [rt]
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR [rt] == ID/EX.IR [rs]
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR [rt] == ID/EX.IR [rt]
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR [rt] == ID/EX.IR [rs]
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR [rt] == ID/EX.IR [rt]
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR [rt] == ID/EX.IR [rs]
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR [rt] == ID/EX.IR [rt]



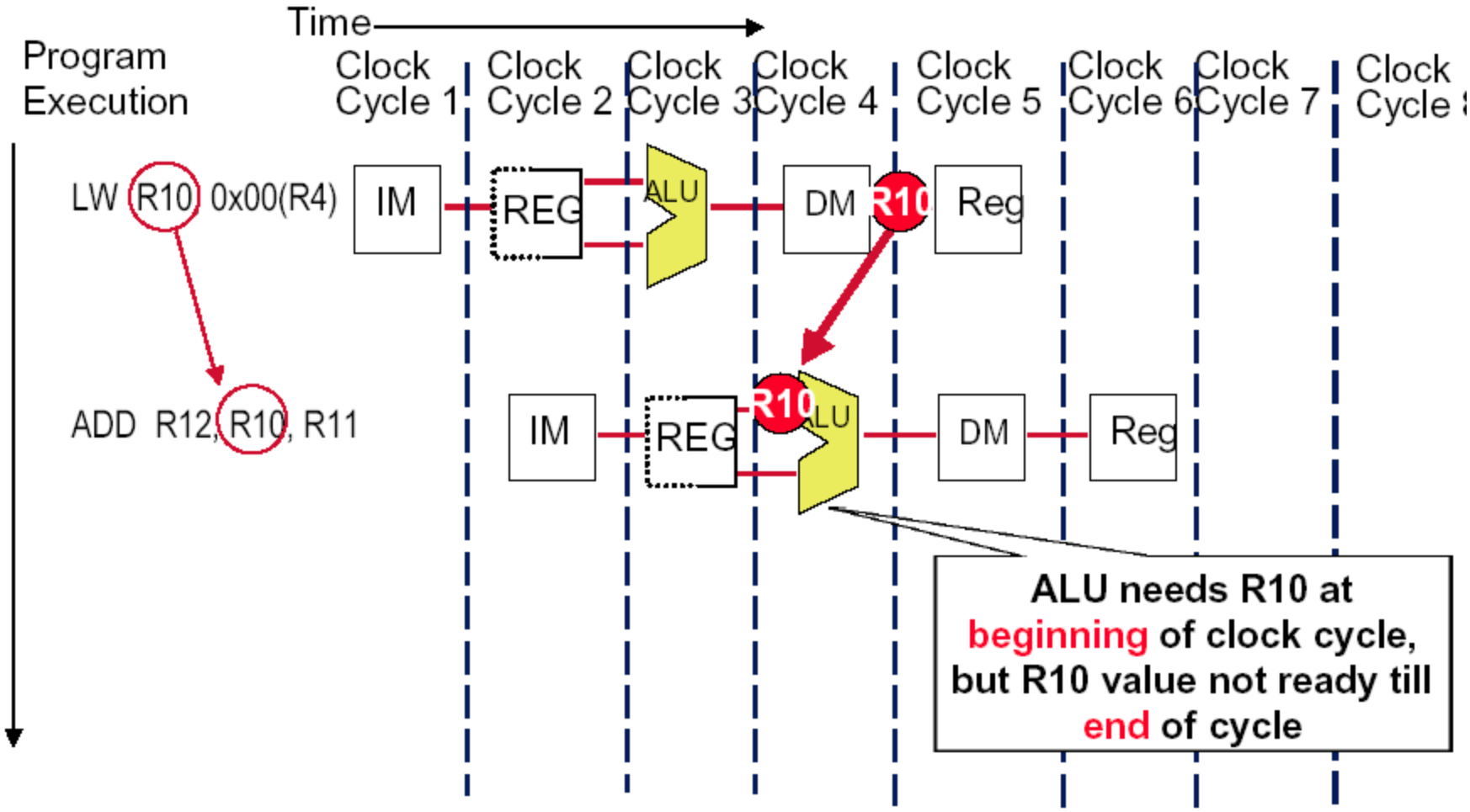
Forwarding path to other input entry

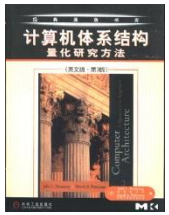


MEM/WB.LMD → DM input

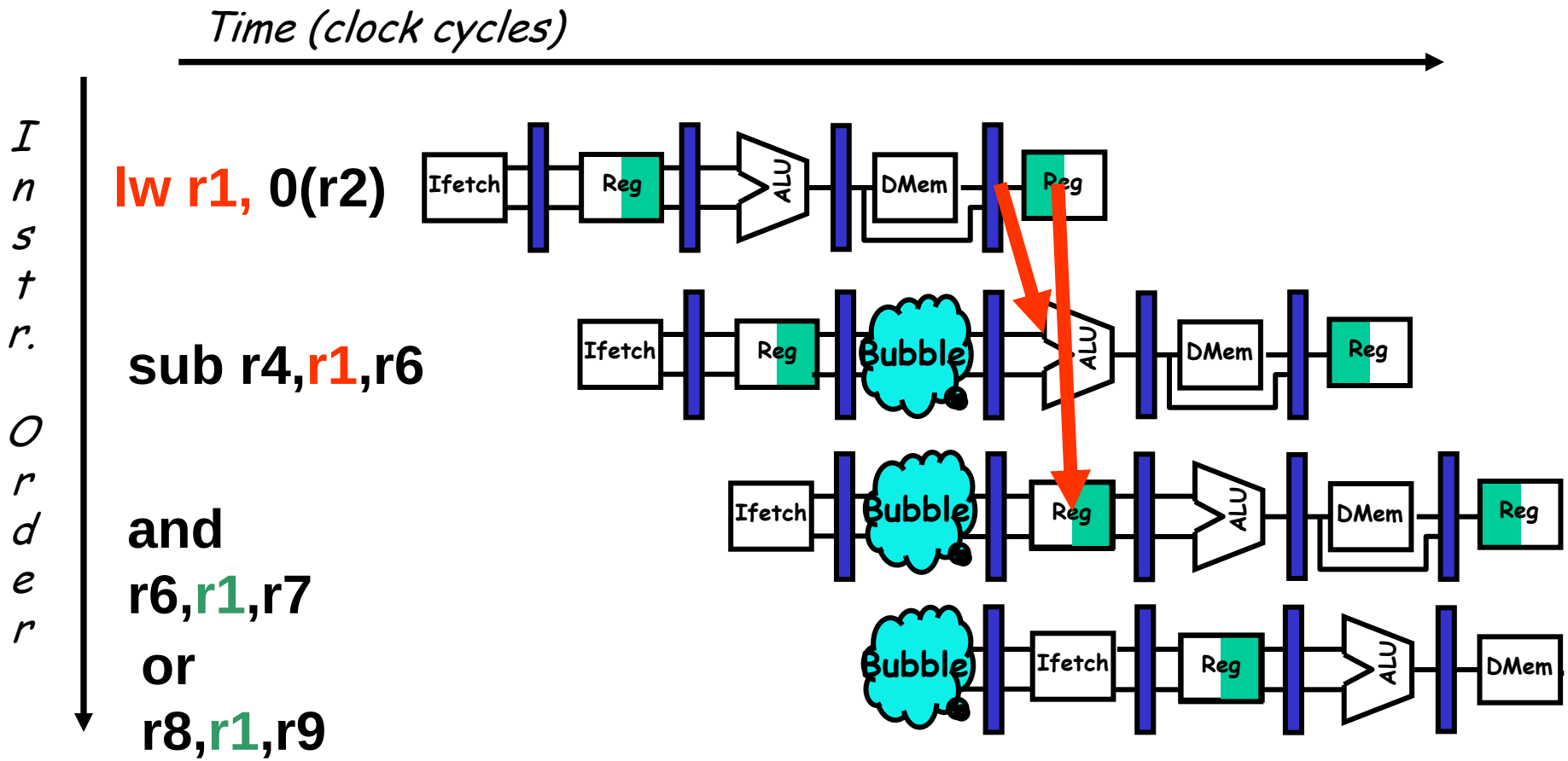


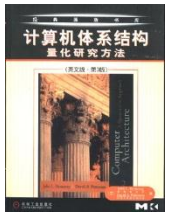
Forwarding Doesn't Always Work





So we have to insert stall: Load stall

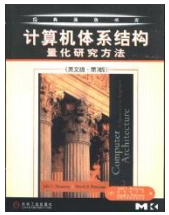




How to implement Load Interlock

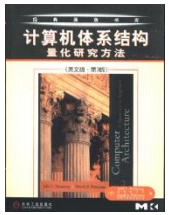
- **Detect** when should use Load Interlock

situation	Example code sequence	Action
No dependence	LD R1, 45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because of no dependence
Dependence requiring stall	LD R1, 45(R2) DADD R5,R1,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX
Dependence overcome by forwarding	LD R1, 45(R2) DADD R5,R6,R7 DSUB R8,R1,R7 OR R9,R6,R7	Comparators detect the use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX
Dependence with accesses in order	LD R1, 45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R1,R7	No action required because read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.



The logic to detect for Load interlock

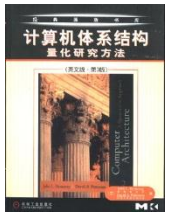
Opcode field of ID/EX	Opcode Field of IF/ID	Matching operand fields
Load	Reg-Reg ALU	ID/EX.IR[rt]==IF/ID.IR[rs]
Load	Reg-Reg ALU	ID/EX.IR[rt]==IF/ID.IR[rt]
Load	Load,store, ALU immediate, branch	ID/EX.IR[rt]==IF/ID.IR[rs]



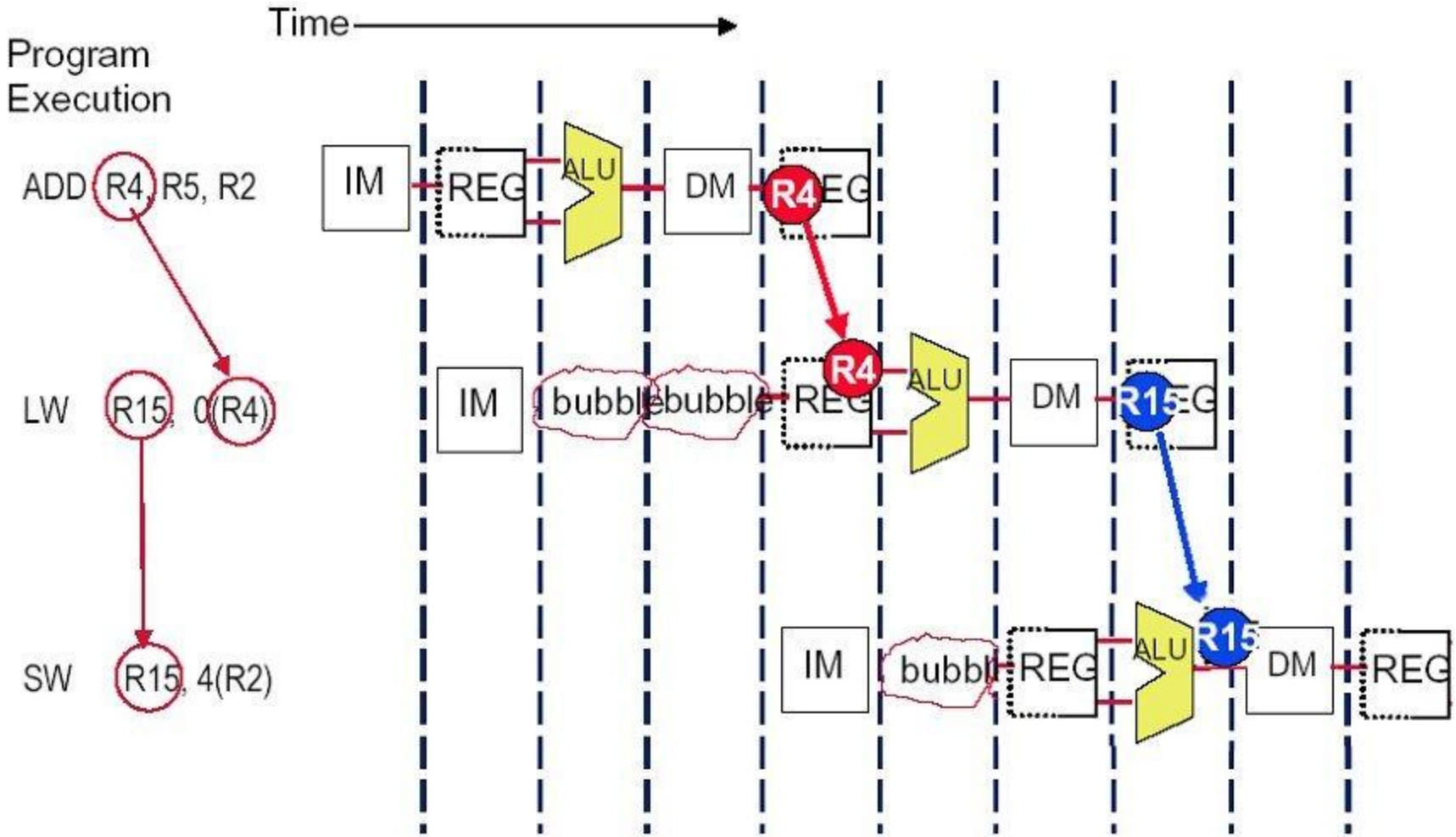
Example of Forwarding and Load Delay

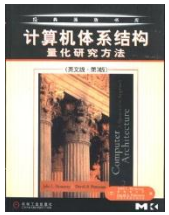
- Why forwarding?
  - ADD R4, R5, R2
  - LW R15, 0(R4)
  - SW R15, 4(R2)

- Why load delay?
  - ADD R4, R5, R2
  - LW R15, 0(R4)
  - SW R15, 4(R2)

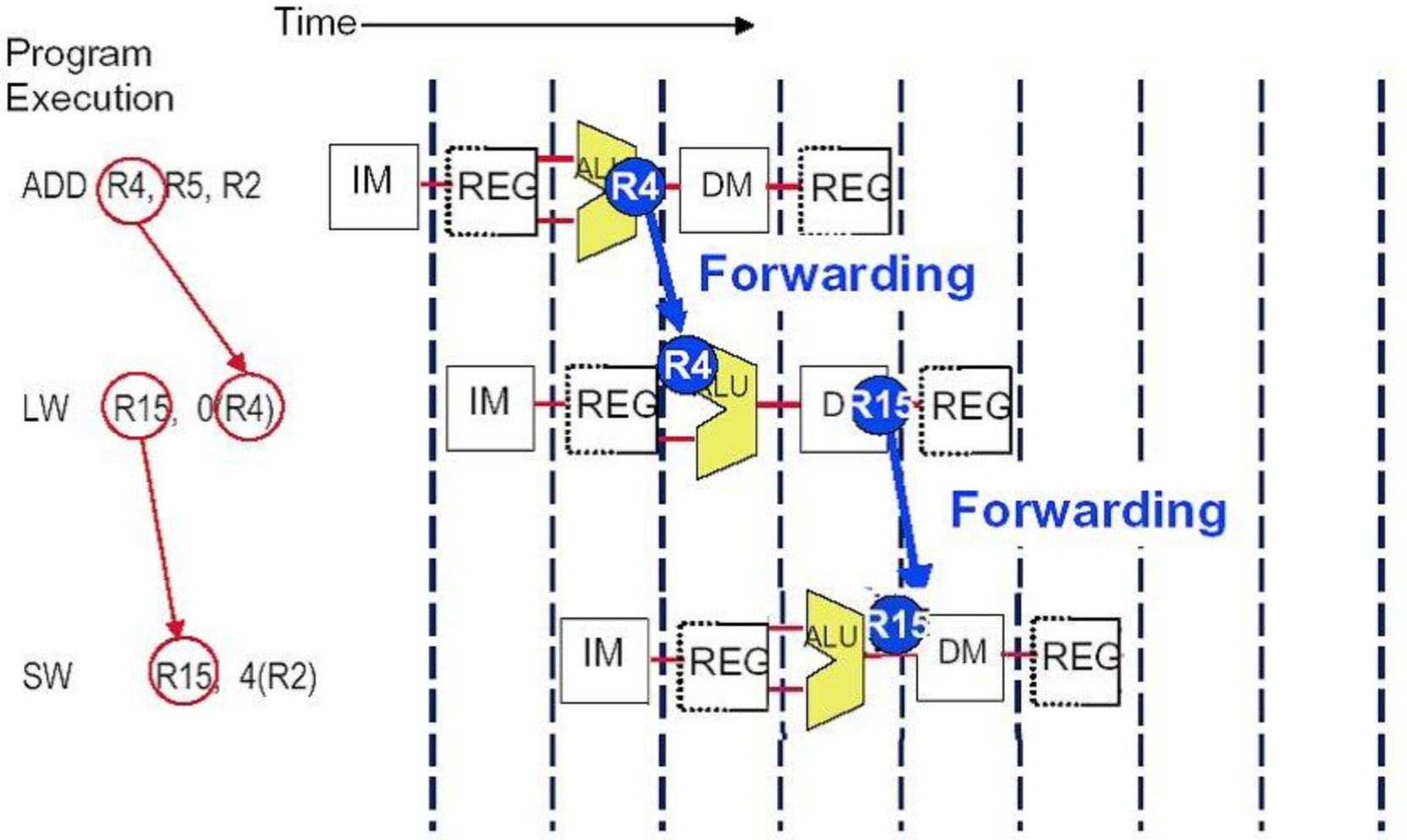


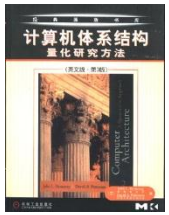
Solution (without forwarding)





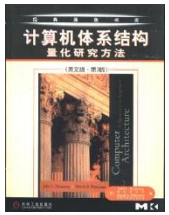
Solution (with forwarding)



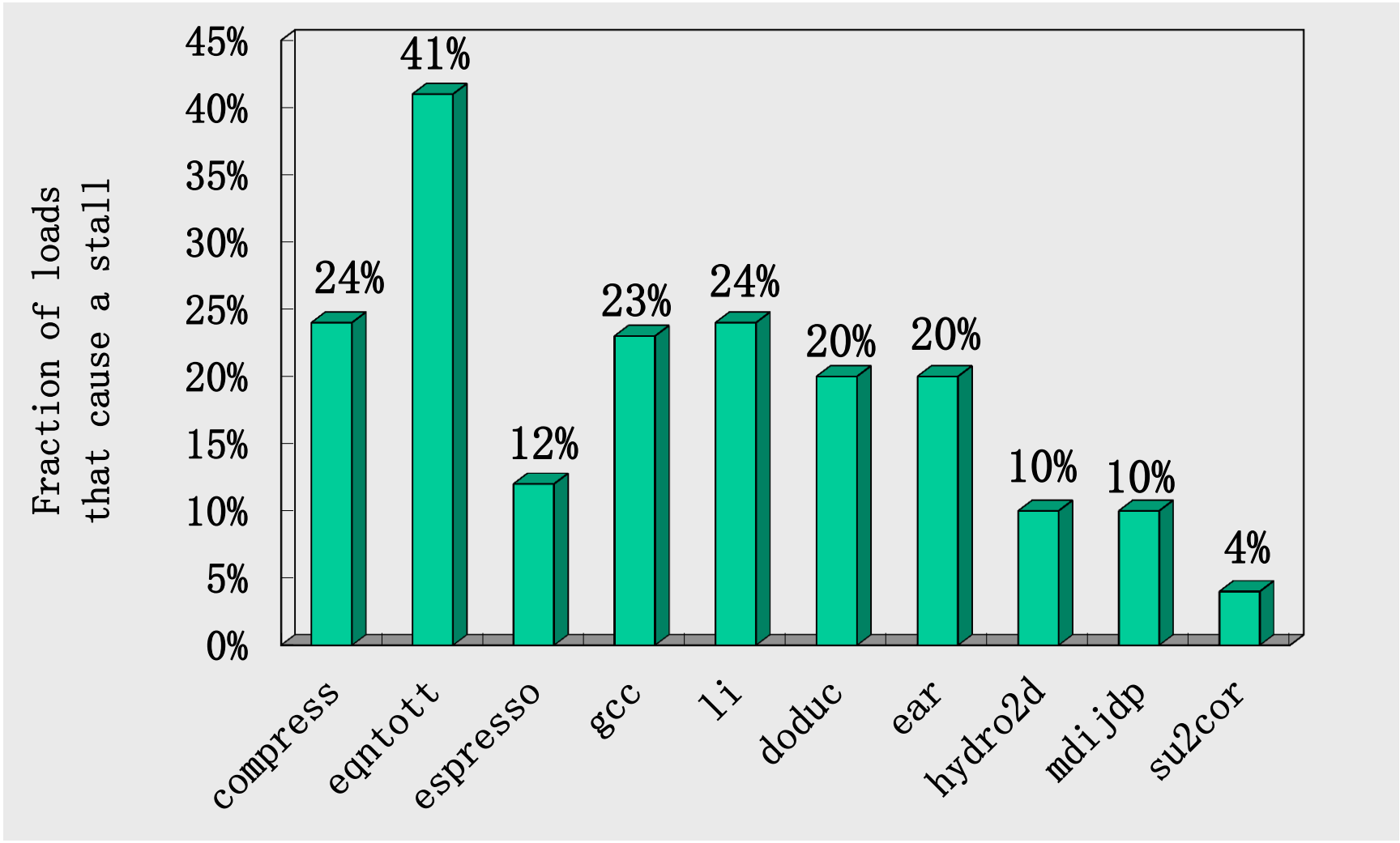


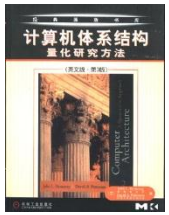
The performance influence of load stall

- *Example*
  - *Assume 30% of the instructions are loads.*
  - *Half the time, instruction following a load instruction depends on the result of the load.*
  - *If hazard causes a single cycle delay, how much faster is the ideal pipeline ?*
- Answer
  - CPI = 1+30%×50% ×1=1.15
  - The performance decrease about 15% due to load stall.



Fraction of load that cause a stall





Instruction reordering by compiler to avoid load stall

- Try producing fast code for

  a = b + c;

  d = e – f;

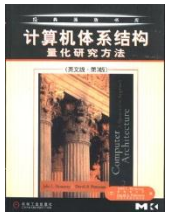
  assuming a, b, c, d ,e, and f in memory.

- **Slow code:**

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Fast code:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```



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, 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