第三章 Instruction-Level Parallelism and Its Dynamic Exploitation

陈文智
chenwz@zju.edu.cn
浙江大学计算机学院
2014年10月

3.3 The Major Hurdle of Pipelining—Pipeline Hazards

本科回顾------- Appendix A.2

3.3.1 Taxonomy of hazard

3.3.2 Performance of pipeline with Hazard

3.3.3 Structural hazard

3.3.4 Data Hazards

3.3.5 Control Hazards

3.3.1 Taxonomy of hazard

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

- 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 .

3.3.2 Performance of pipeline with stalls

- Recall the speedup formula:

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

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

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

Assumptions for calculation

- The ideal CPI on a pipelined processor is almost always 1.

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.

- Clock cycle unpipelined = Clock cycle pipelining
- CPI unpipelined = pipeline depth

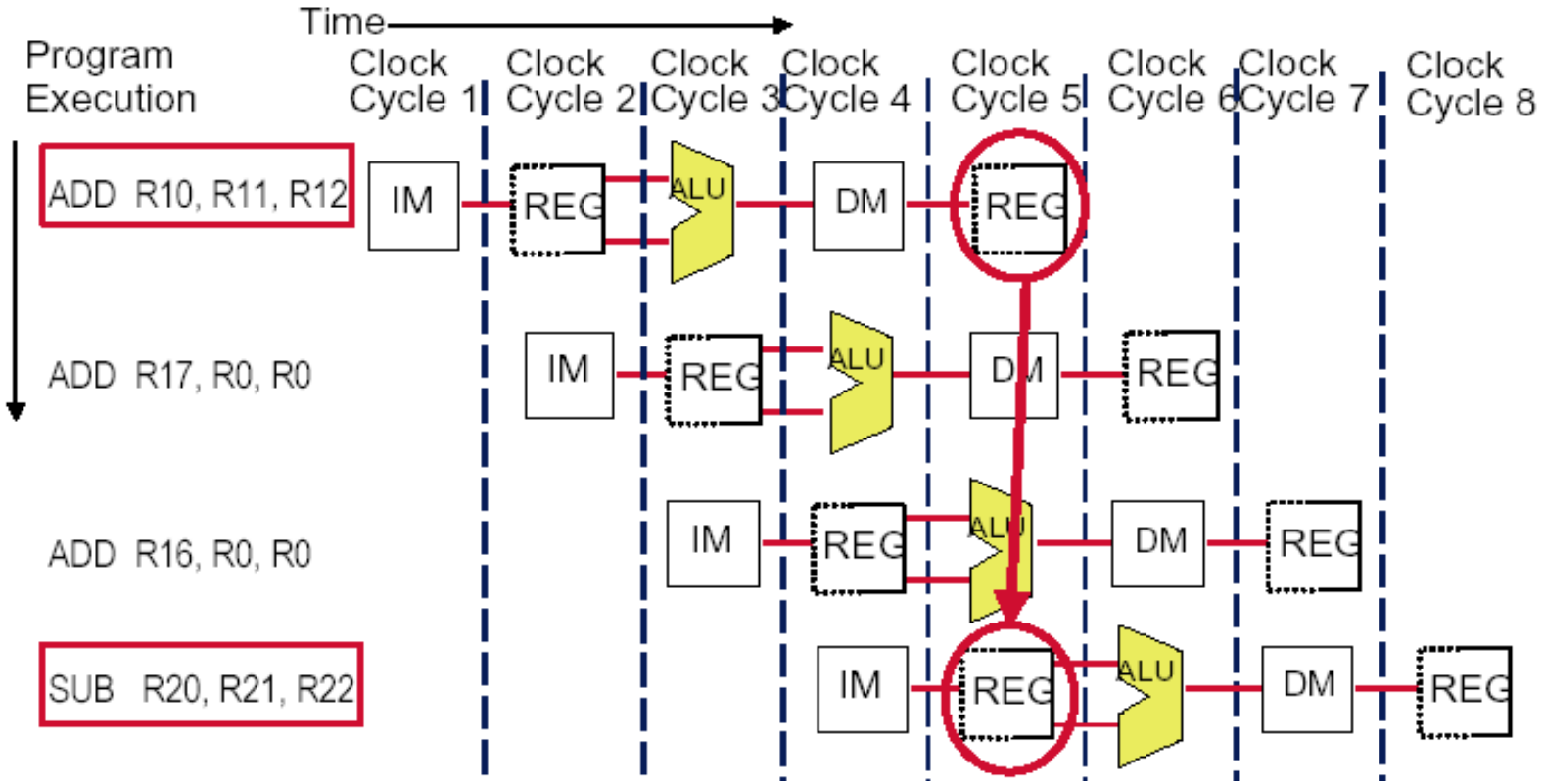
$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

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

3.3.3 Structural hazard

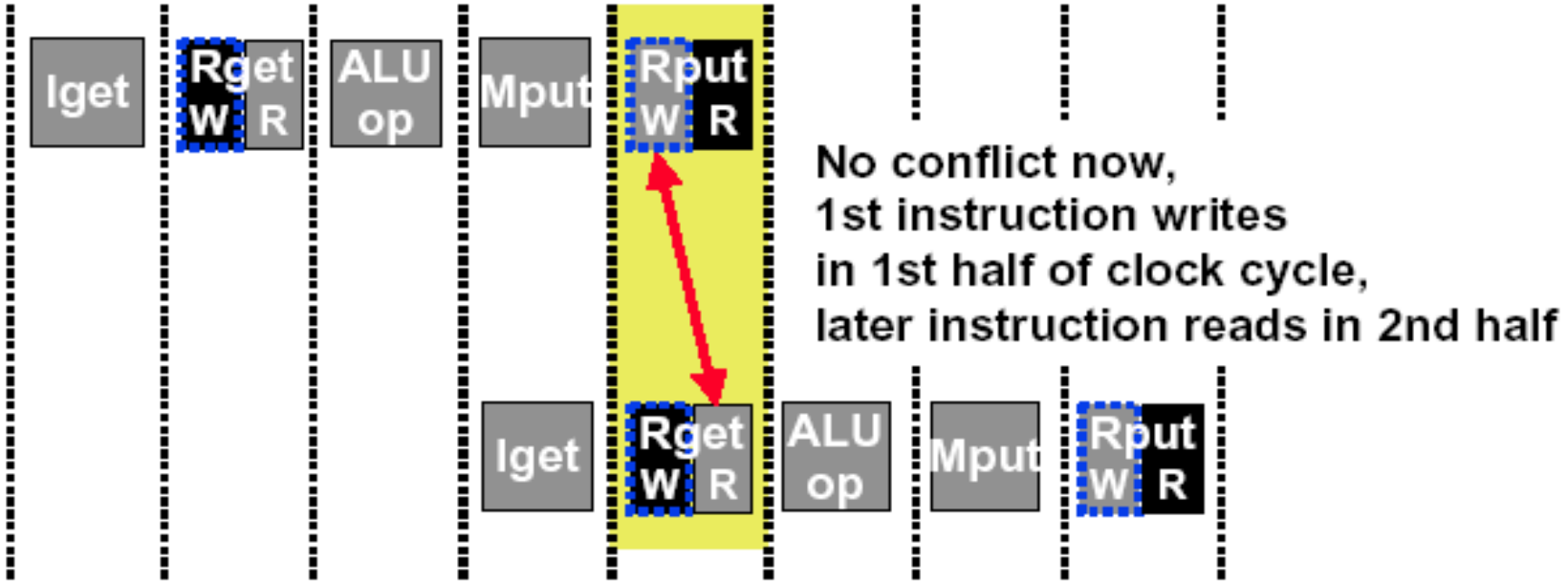
- **Structural hazards**
  - Occurs when two or more instructions want to use the same hardware resource in the same cycle
  - Causes bubble (stall) in pipelined machines
  - Overcome by replicating hardware resources
    - Multiple accesses to the register file
    - Multiple accesses to memory
   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

一、Multi access to the register file

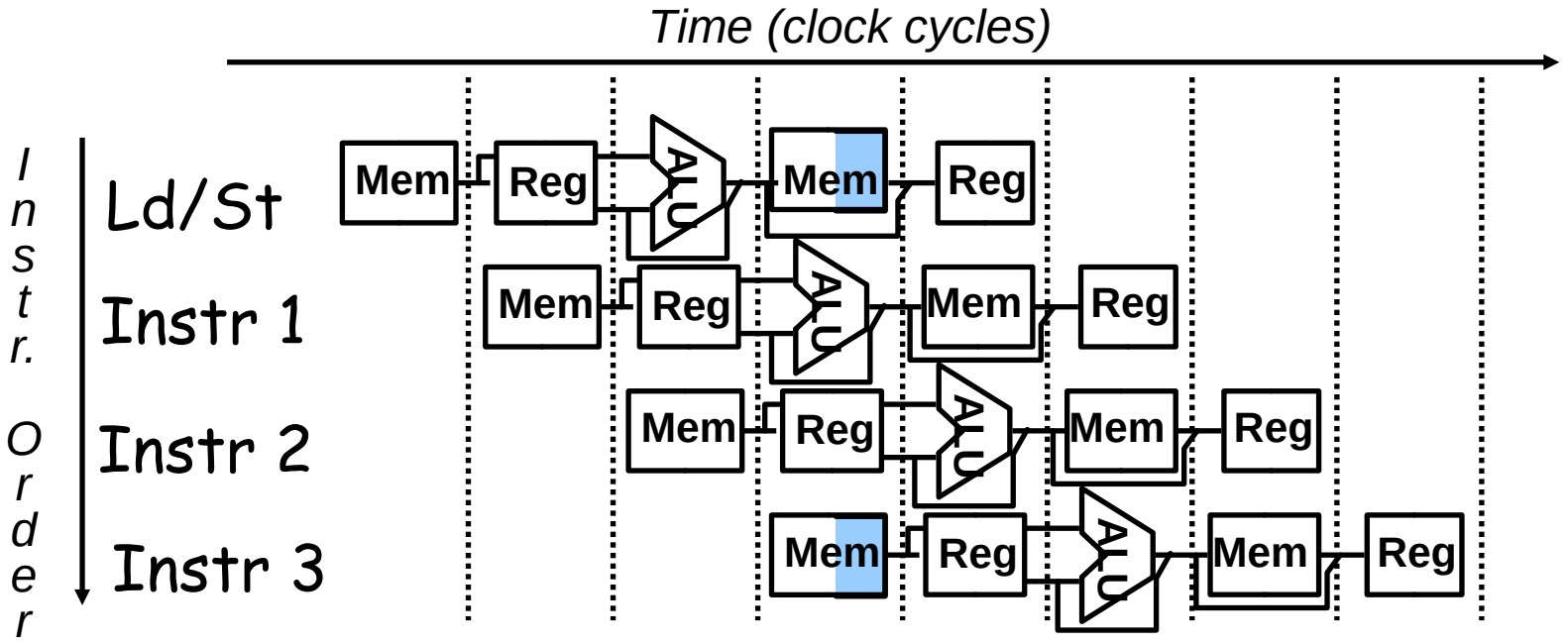


- Simply insert a stall , speedup will be decreased.
- We have resolved it with " double bump"

Double Bump Works !

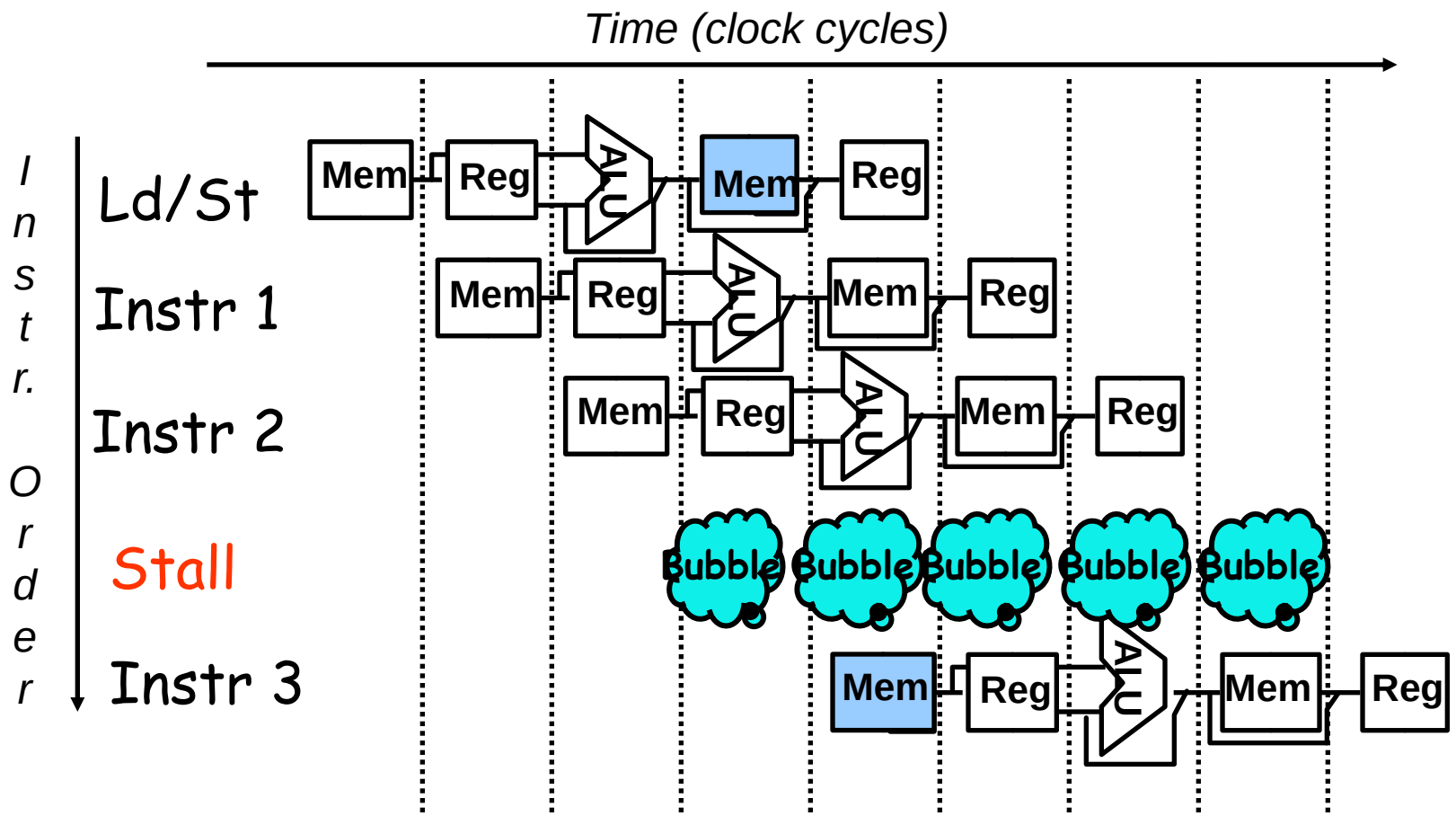


二、Multi access to Single Memory

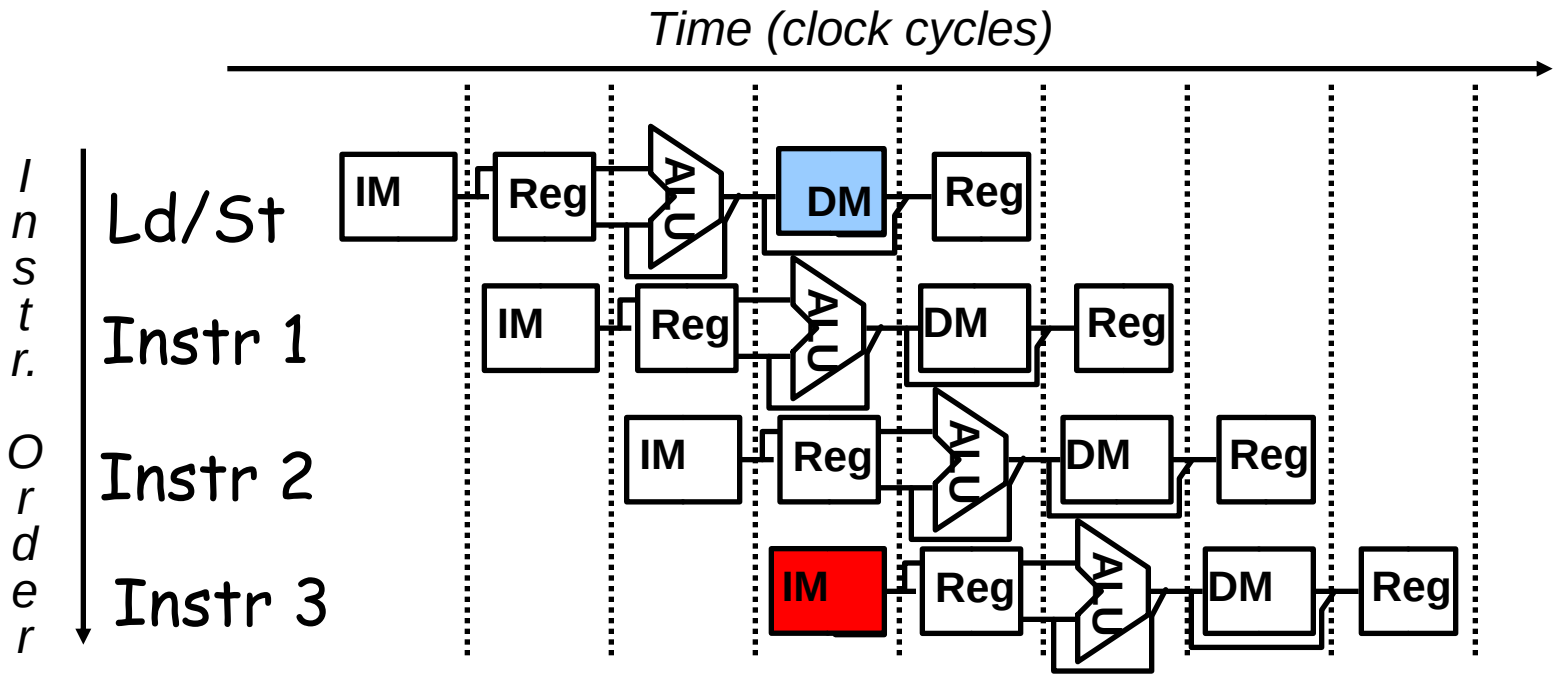


- Insert stall
- provide another memory port
- split instruction memory and data memory
- use instruction buffer

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.

三、Not fully pipelined function unit

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

四、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%.

3.3.4 Pipelining Data 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

一、Data hazard

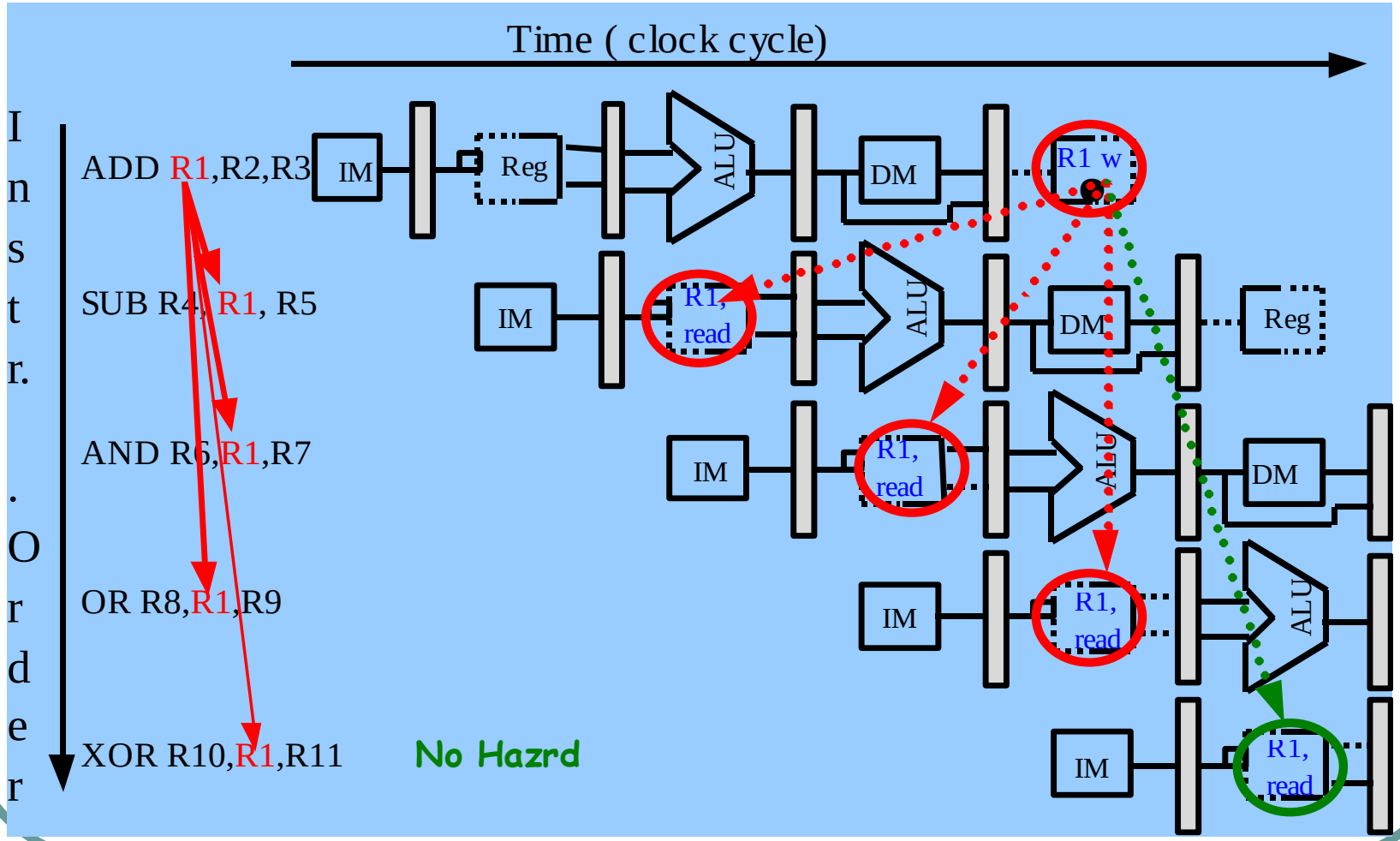
- Data hazards occur when the pipeline changes the order of read/write accesses to operands comparing with that in sequential executing .
- Let's see an Example

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

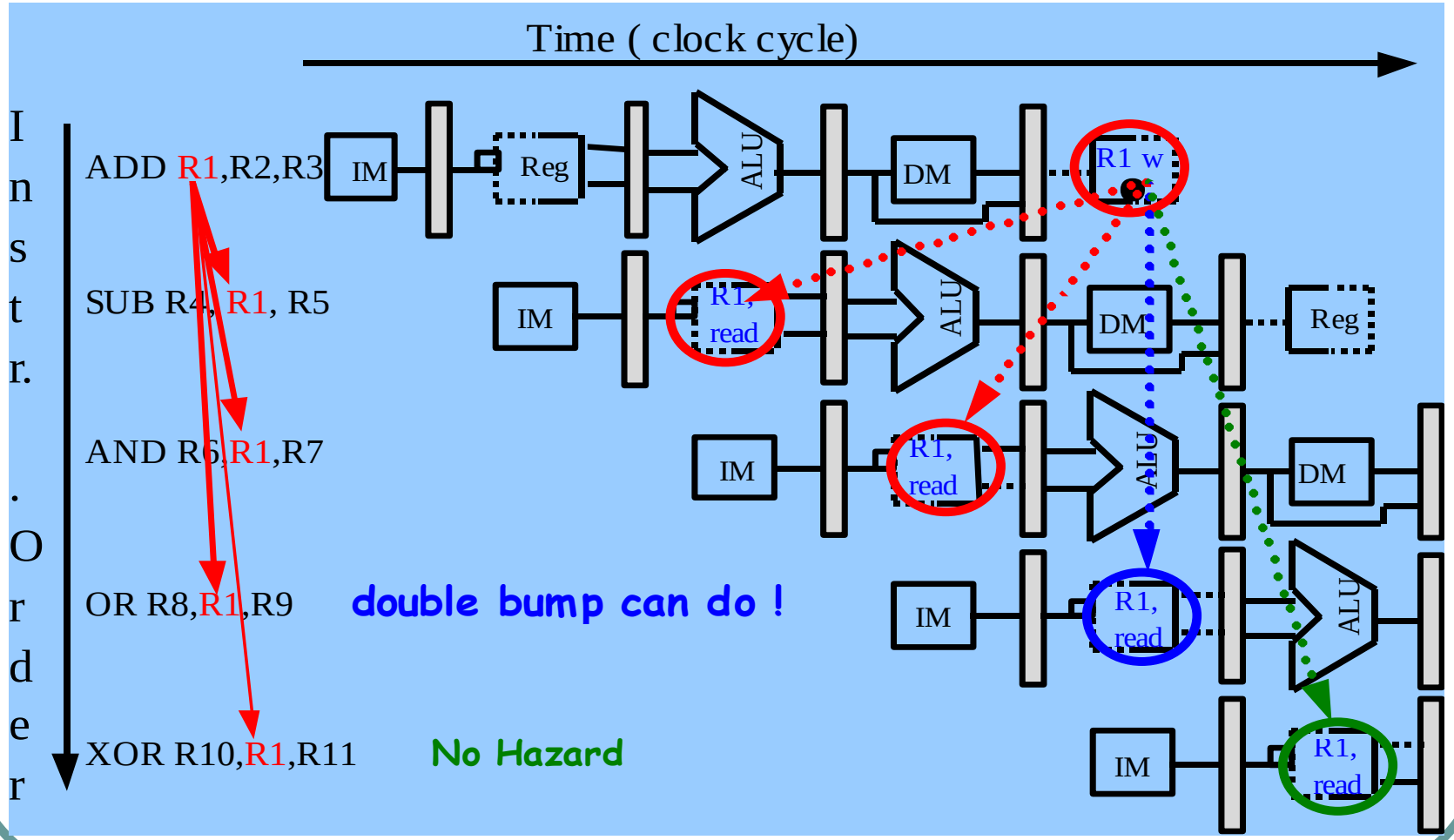
Data hazard

- 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



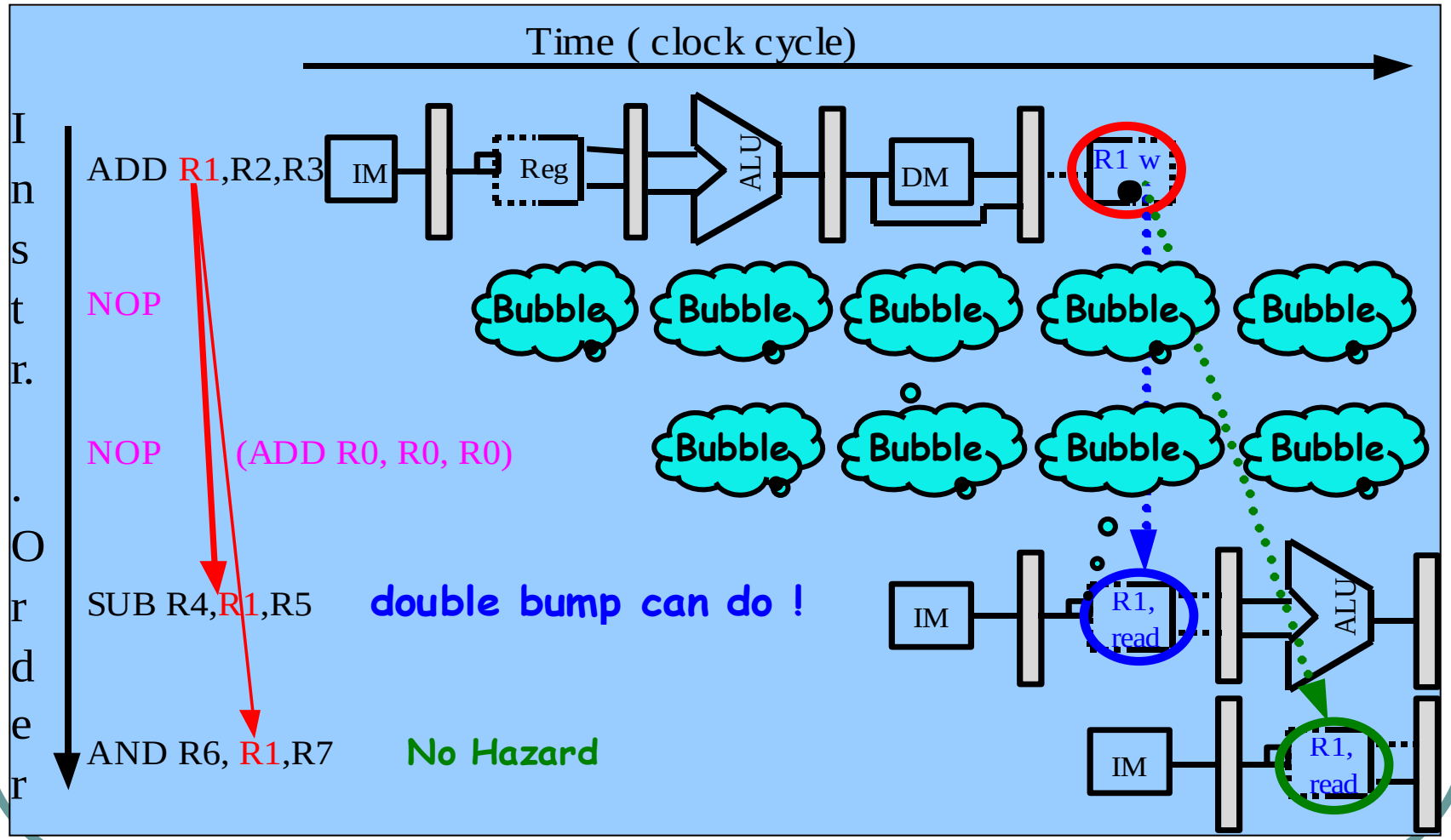
二、Somecases “Double Bump” can do !



三、Proposed solution— STALL

- **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**

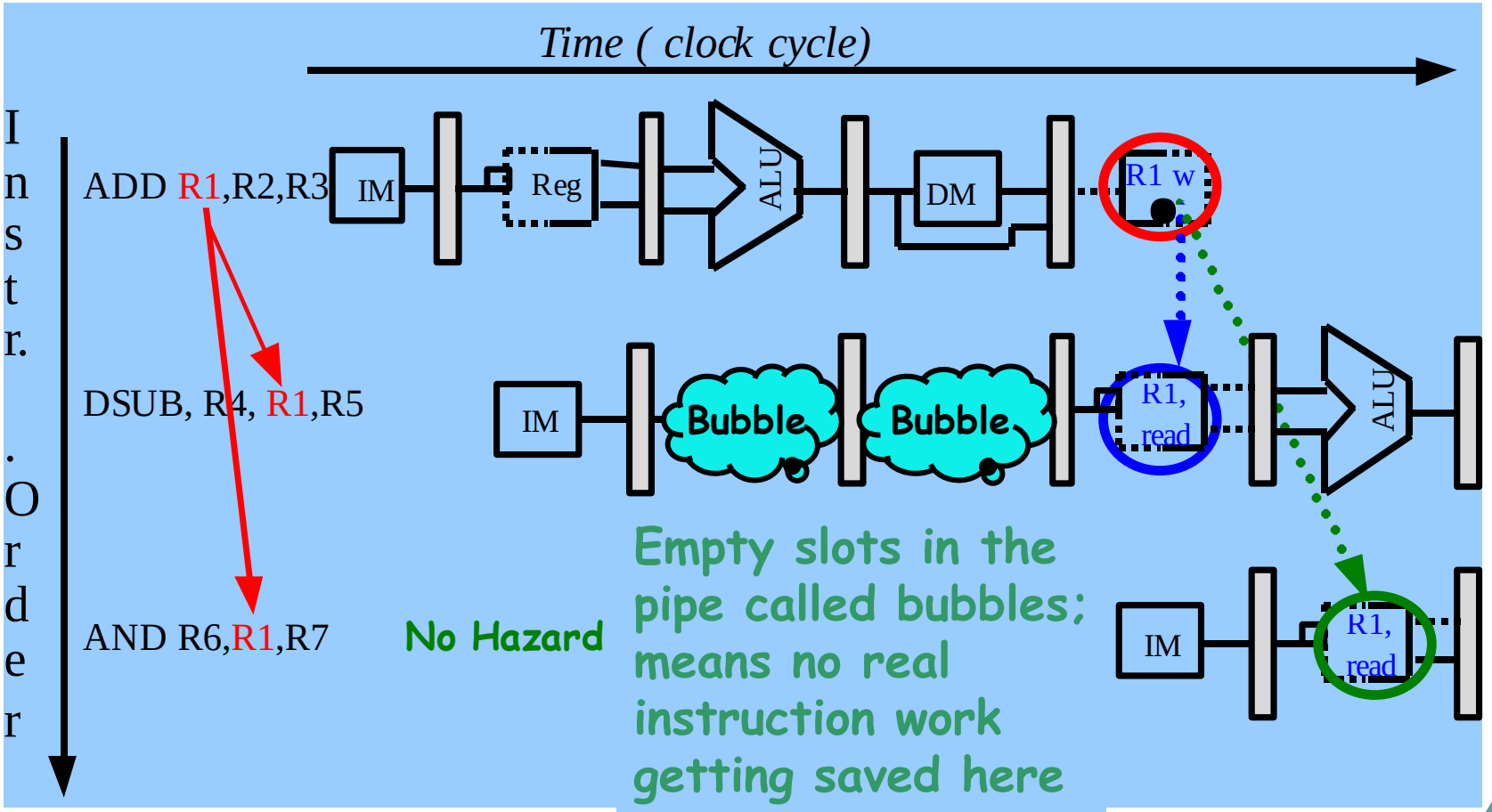
How do we stall ?
——Insert nop by compiler



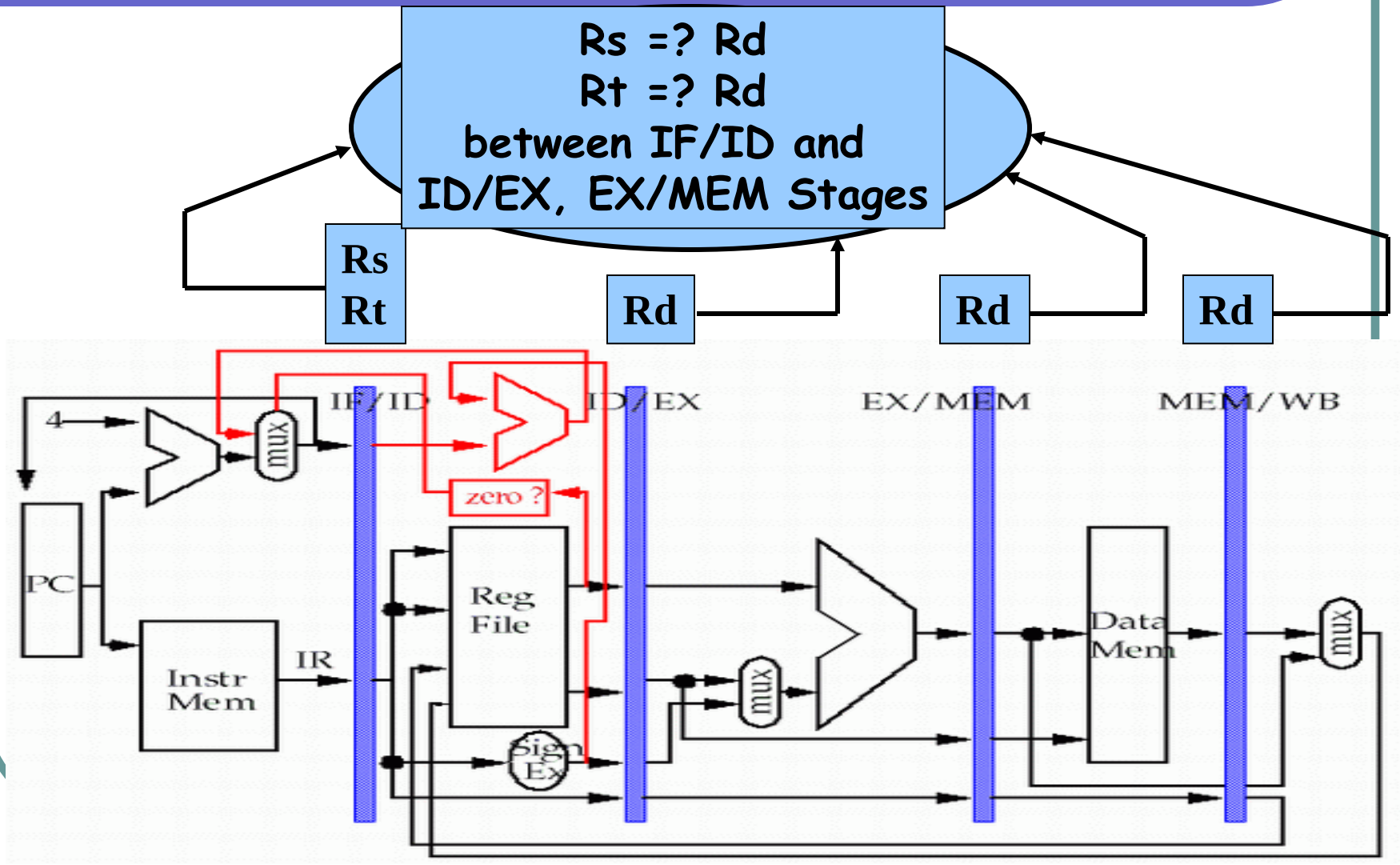
How do we stall?
——Add hardware Interlock !

- 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
  - 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



Detect: Data Hazard Logic



四、Forwarding

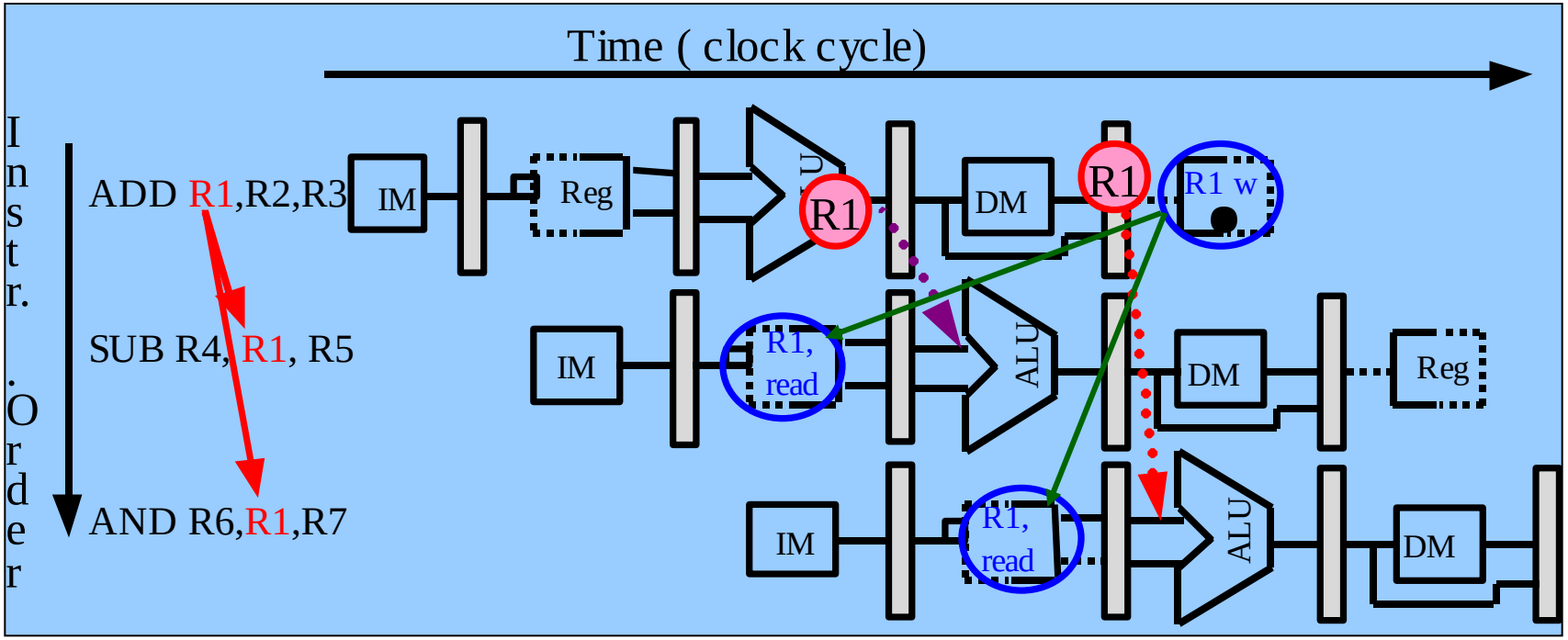
- If the result you need does not exist AT ALL yet,
  - you are out of luck, sorry.
- But, what if the result exists, but is not stored back yet?
  - Instead of stalling until the result is stored back in its "natural" home…
  - **grab the result "on the fly" from "inside" the pipe, and send it to the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

Forwarding

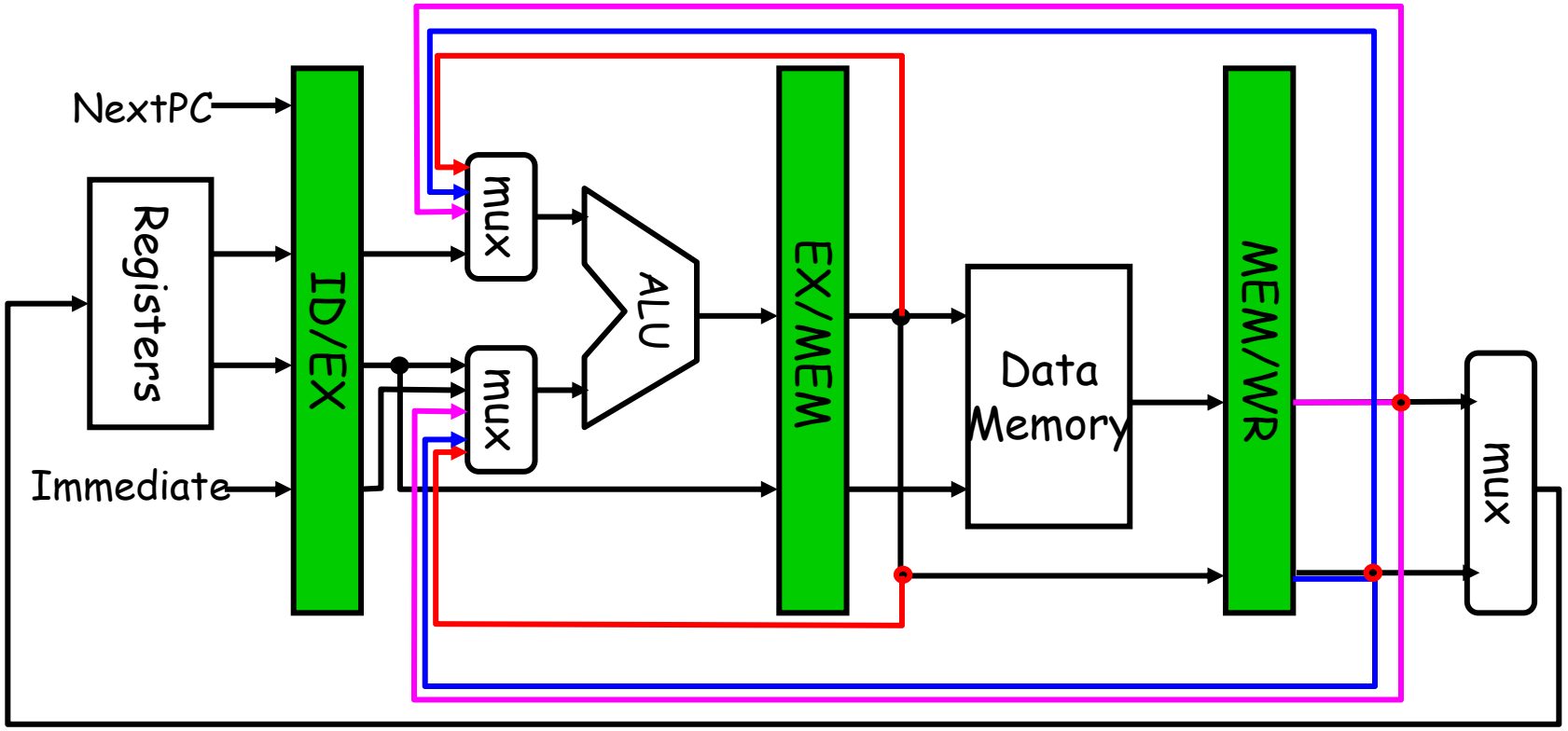
- Generic name: forwarding (bypass, short-circuiting)
  - Instead of waiting to store the result, we forward it immediately (more or less) to the instruction that wants it
  - Mechanically, we add **buses** to the datapath to move these values
  - around, and these buses always "point backwards" in the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

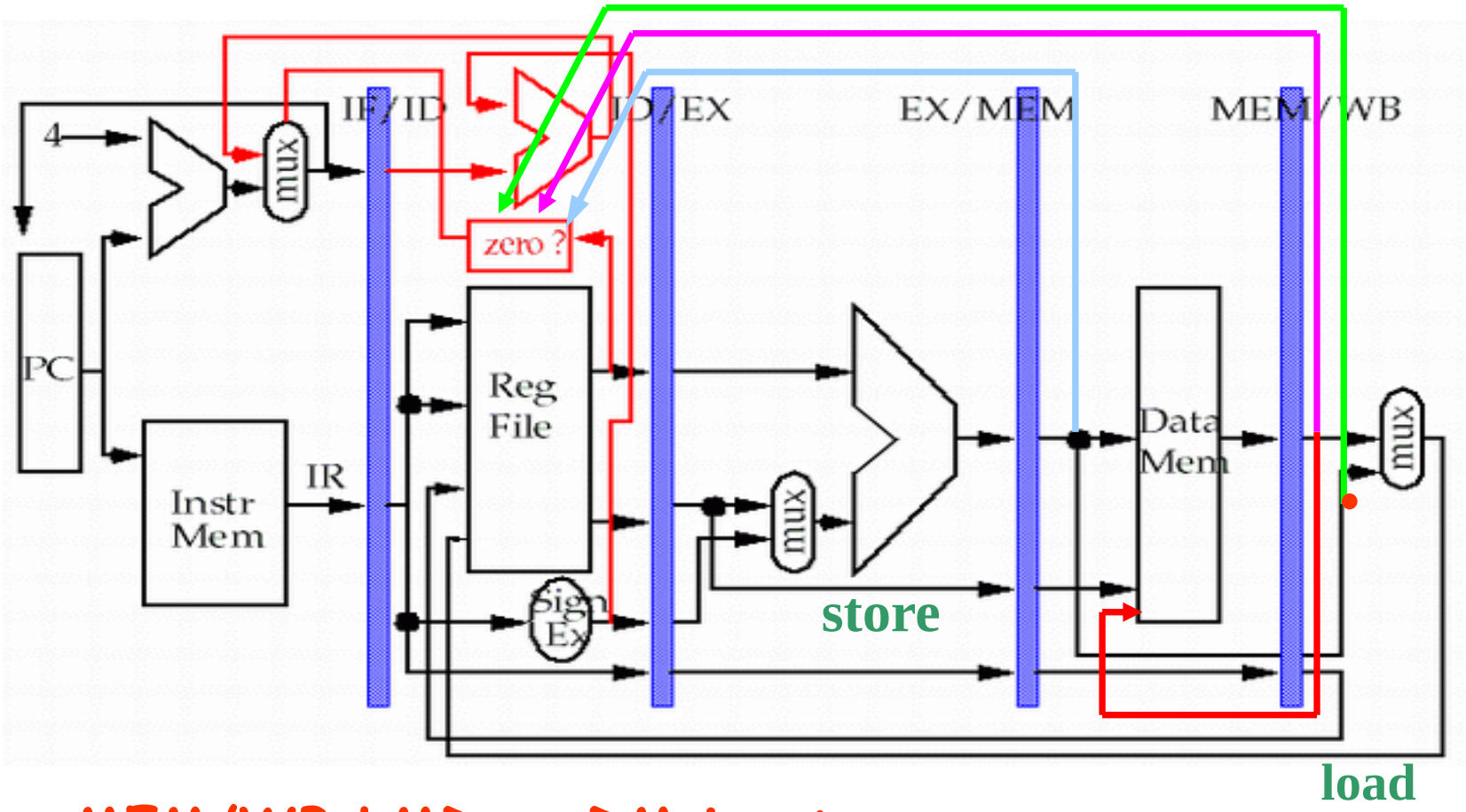


EX/Mem.ALUoutput → ALU input

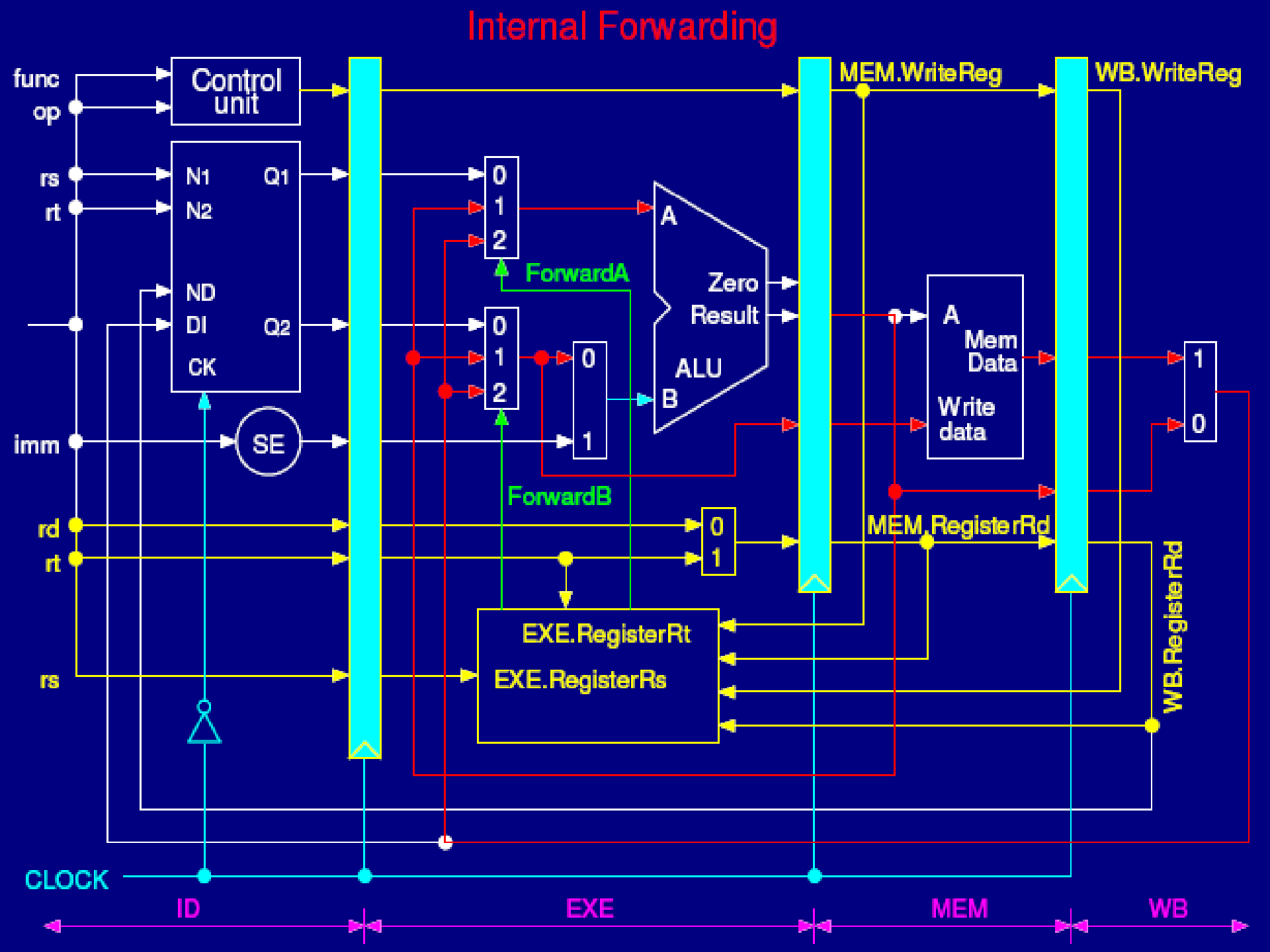
MEM/WB.ALUoutput → ALU input

MEM/WB.LMD → ALU input

Forwarding path to other input entry



MEM/WB.LMD → DM input


Internal Forwarding
func
op
rs
rt
imm
rd
rt
rs
Control unit
N1
N2
ND
DI
CK
Q1
Q2
SE
0
1
2
ForwardA
ForwardB
A
Zero
Result
ALU
B
MEM.WriteReg
WB.WriteReg
A
Mem Data
Write data
MEM.RegisterRd
WB.RegisterRd
EXE.RegisterRt
EXE.RegisterRs
CLOCK
ID
EXE
MEM
WB

Internal Forwarding

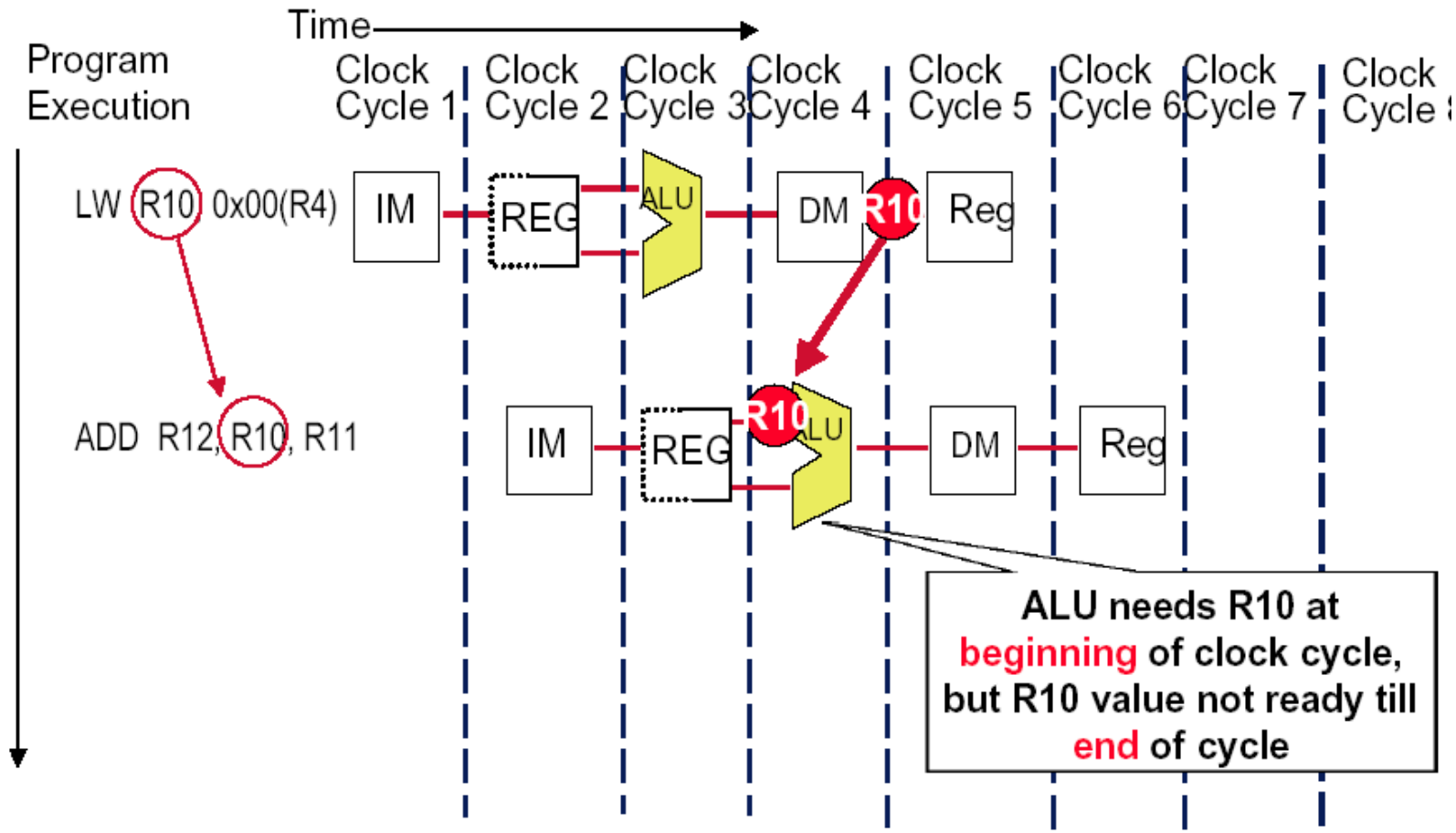
if (MEM.WriteReg
and (MEM.RegisterRd≠0)
and (MEM.RegisterRd==EXE.RegisterRs)) ForwardA=01

if (MEM.WriteReg
and (MEM.RegisterRd≠0)
and (MEM.RegisterRd==EXE.RegisterRt)) ForwardB=01

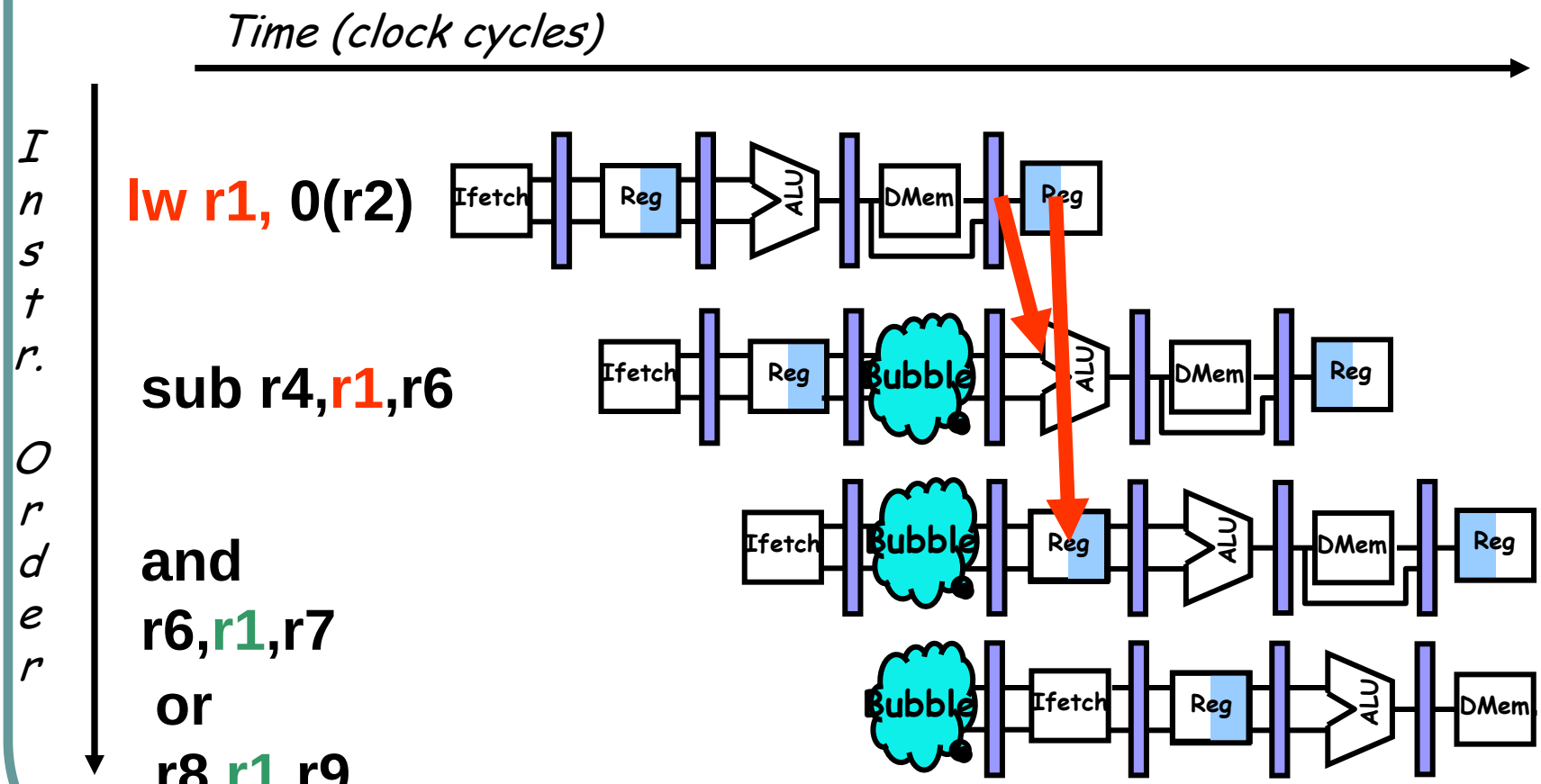
if (WB.WriteReg
and (WB.RegisterRd≠0)
and (MEM.RegisterRd≠EXE.RegisterRs))
and (WB.RegisterRd==EXE.RegisterRs)) ForwardA=10

if (WB.WriteReg
and (WB.RegisterRd≠0)
and (MEM.RegisterRd≠EXE.RegisterRt))
and (WB.RegisterRd==EXE.RegisterRt)) ForwardB=10

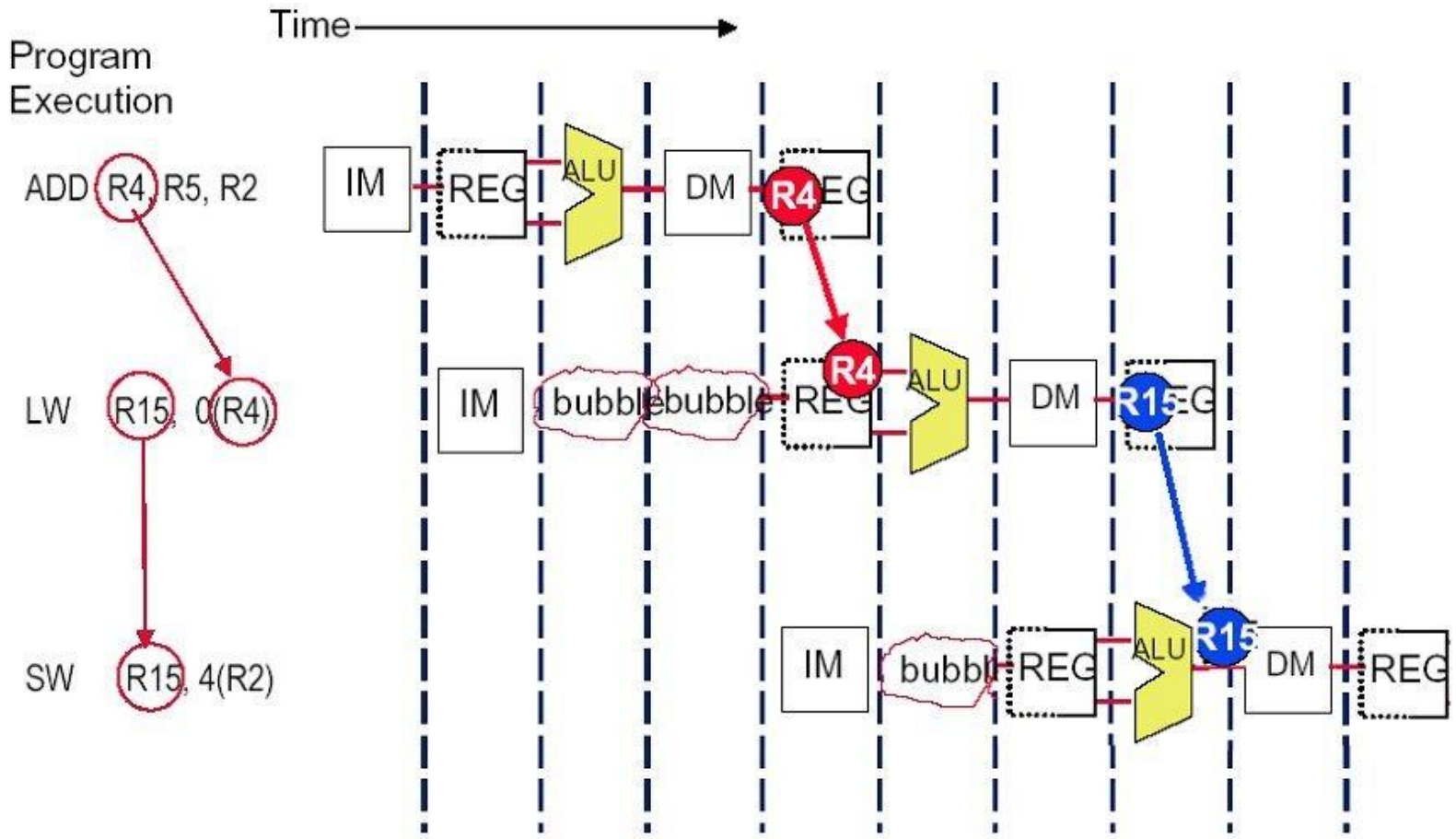
Forwarding isn't Always availability



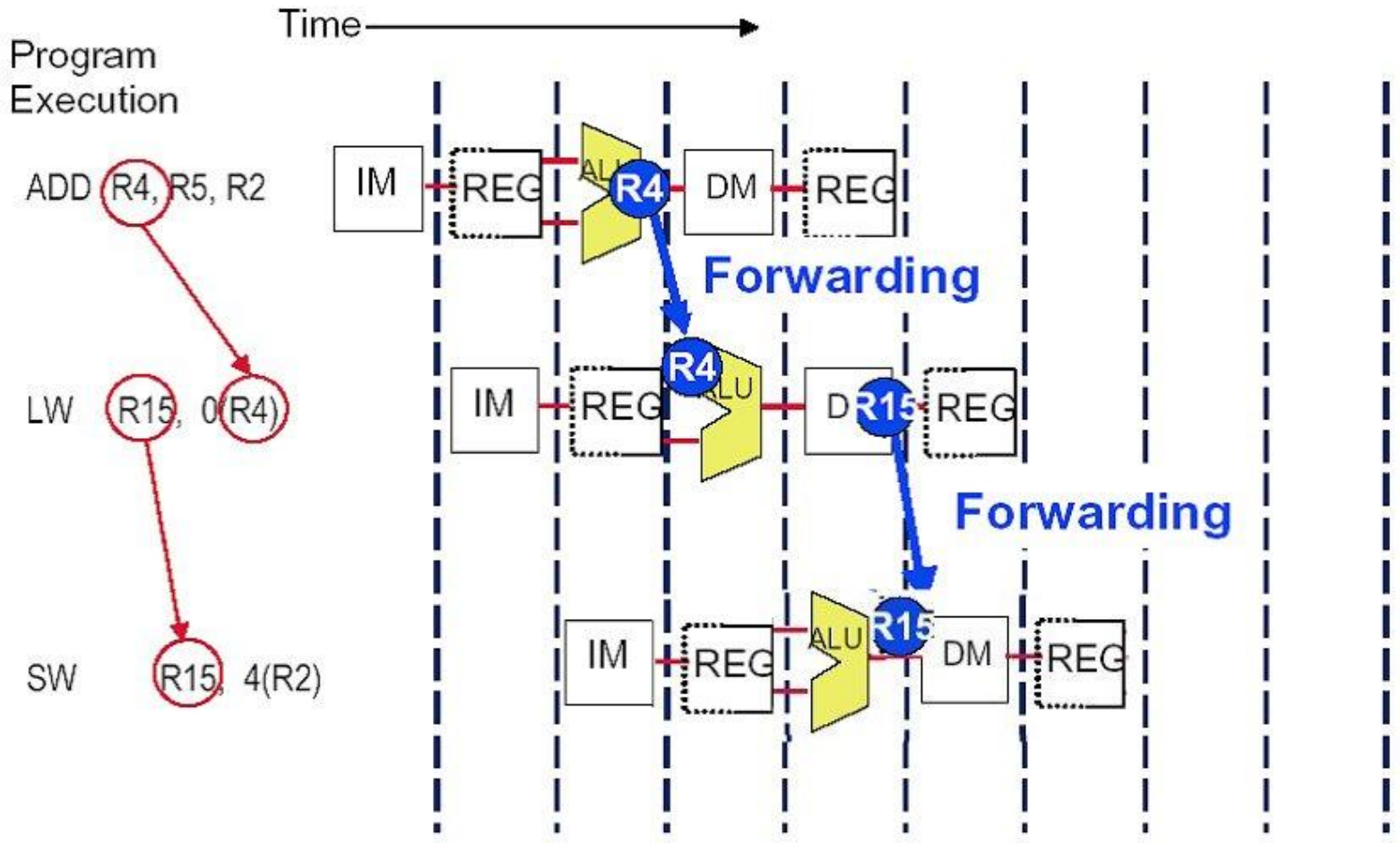
So we have to insert stall: Load stall



Solution (without forwarding)



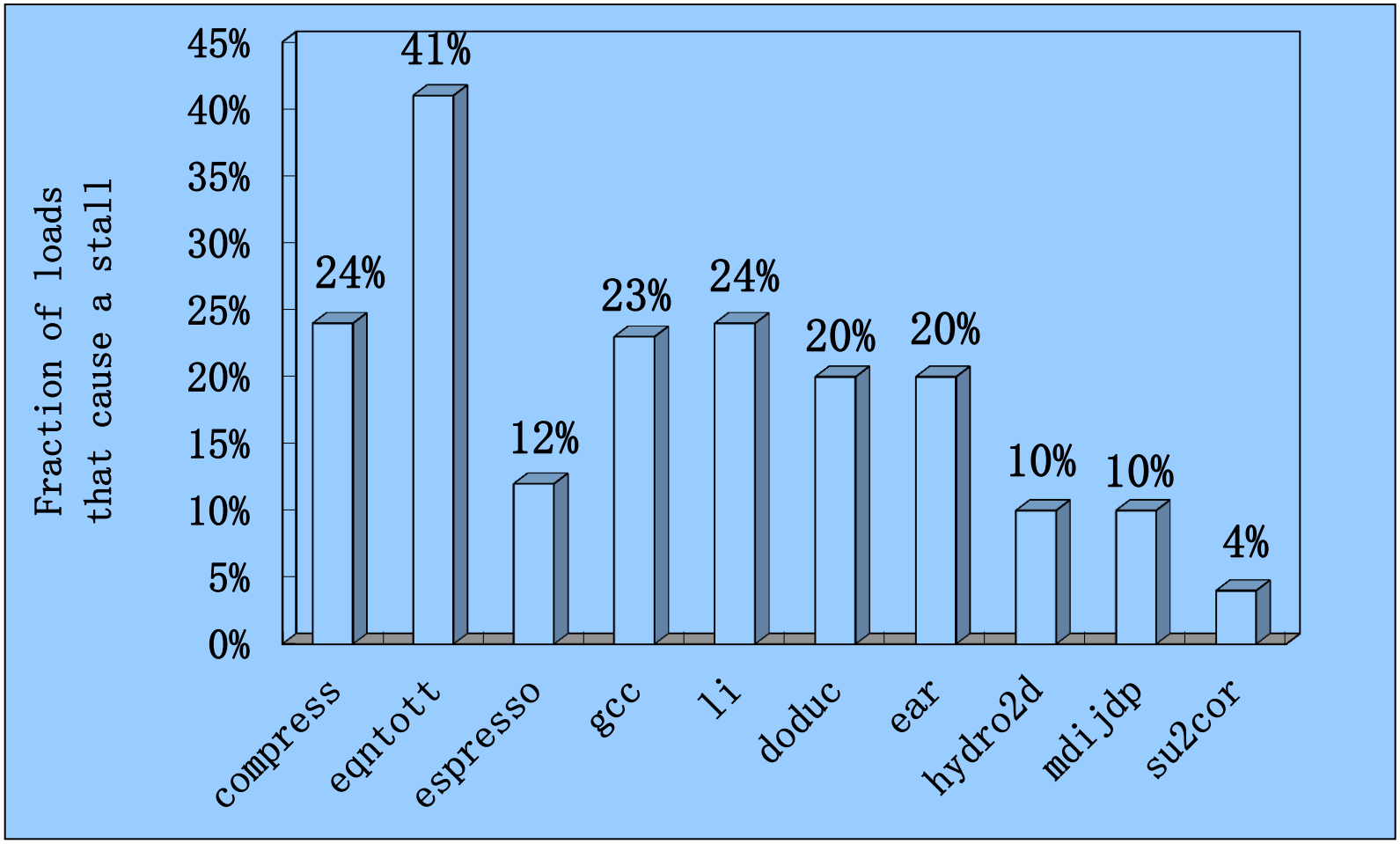
Solution (with forwarding)



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



五、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
```

3.3.5 Pipelining Control 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**