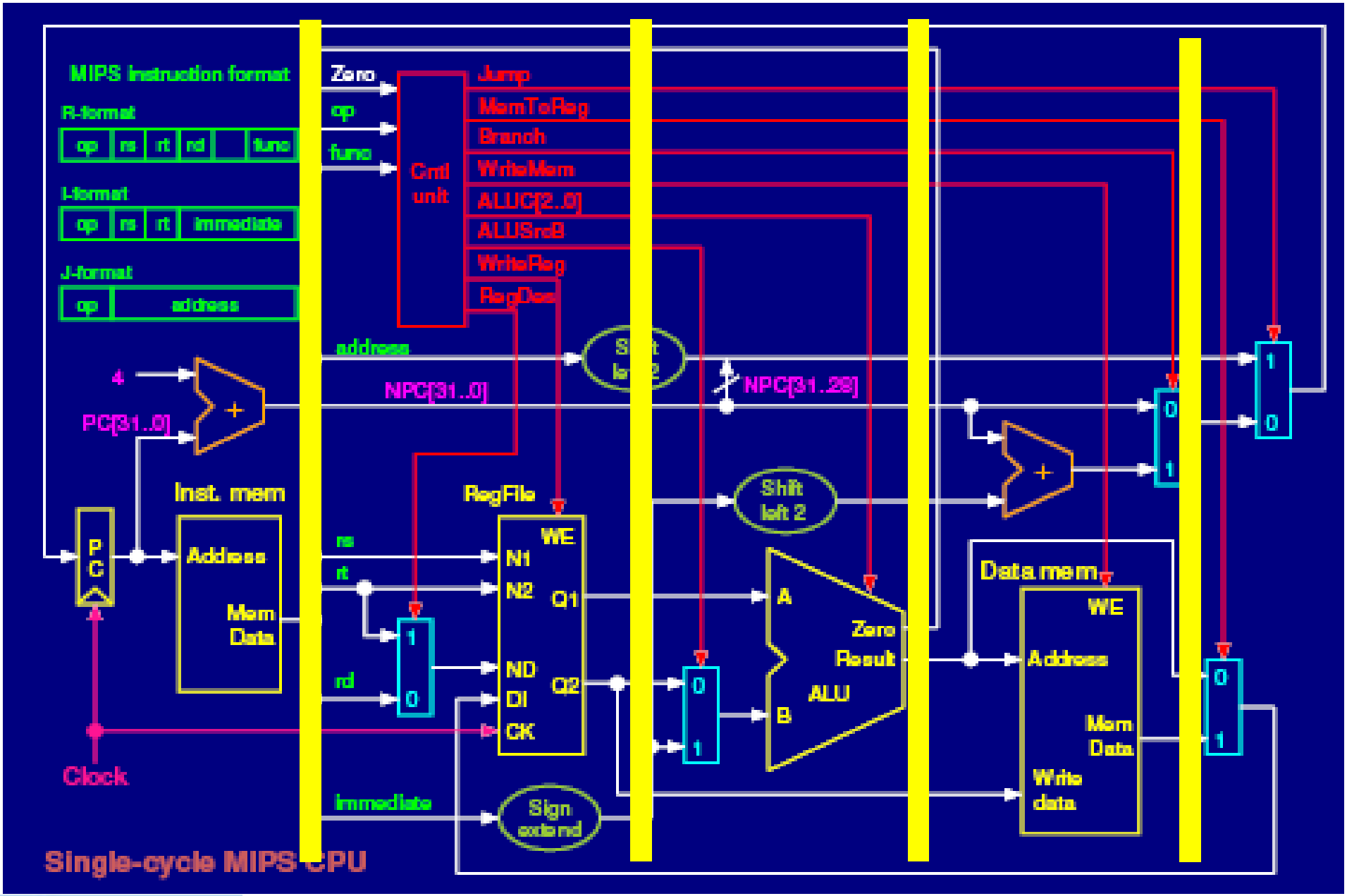
Pipeline Hazards

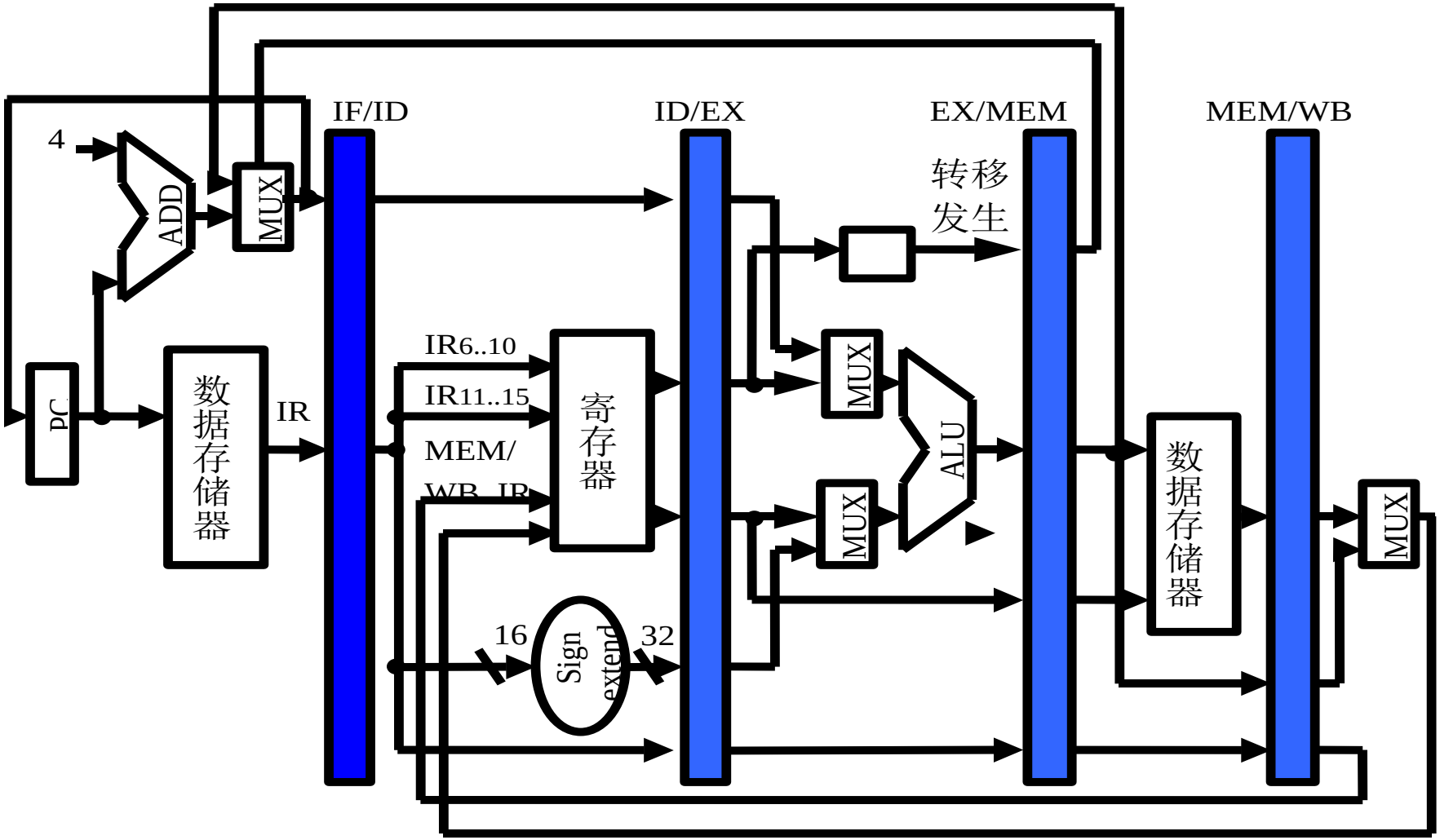
Structure hazard & Data hazard


MIPS instruction format
R-format
op
rs
rt
rd
func
I-format
op
rs
rt
immediate
J-format
op
address
Zero
op
func
Cntl unit
Jump
MemToReg
Branch
WriteMem
ALUC[2..0]
ALUSrcB
WriteReg
RegDest
address
4
PC[31..0]
NPC[31..0]
NPC[31..28]
Inst. mem
Address
Mem Data
PC
Clock
RegFile
WE
N1
N2
ND
DI
CK
Q1
Q2
rs
rt
rd
Shift left 2
A
B
ALU
Zero
Result
Data mem
WE
Address
Mem Data
Write data
immediate
Sign extend
Single-cycle MIPS CPU

What should be latched ?

- Is register PC a latch ?
- What are stored in latch IF/ID ?
  - IR, anythingelse ?
- What are stored in latch ID/EX ?
  - A, B, imm, anythingelse ?
- What are stored in latch EX/MEM ?
  - ALUoutput, anythingelse ?
- What are stored in latch MEM/WB ?
  - LMDR, ALUoutput, Wreg, anythingelse ?

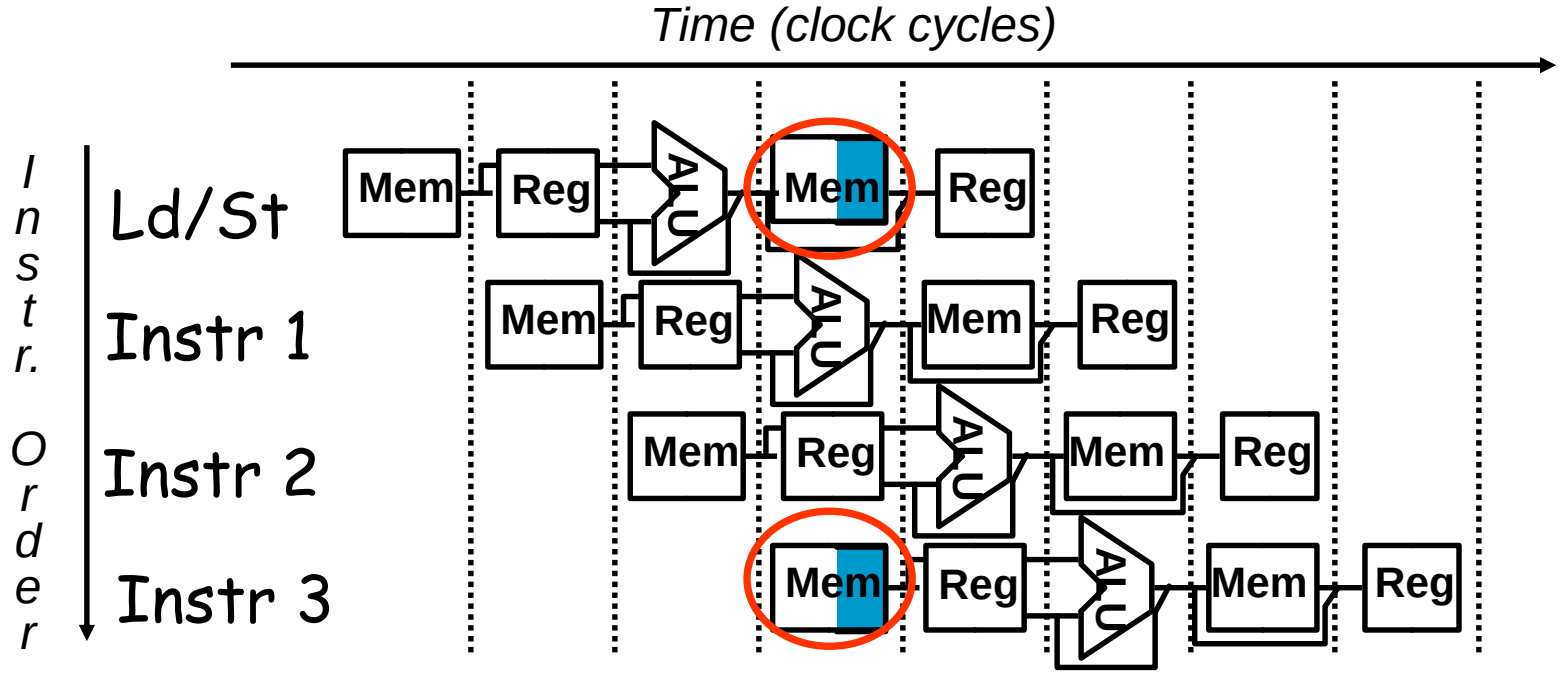
5-stage Version of MIPS Datapath



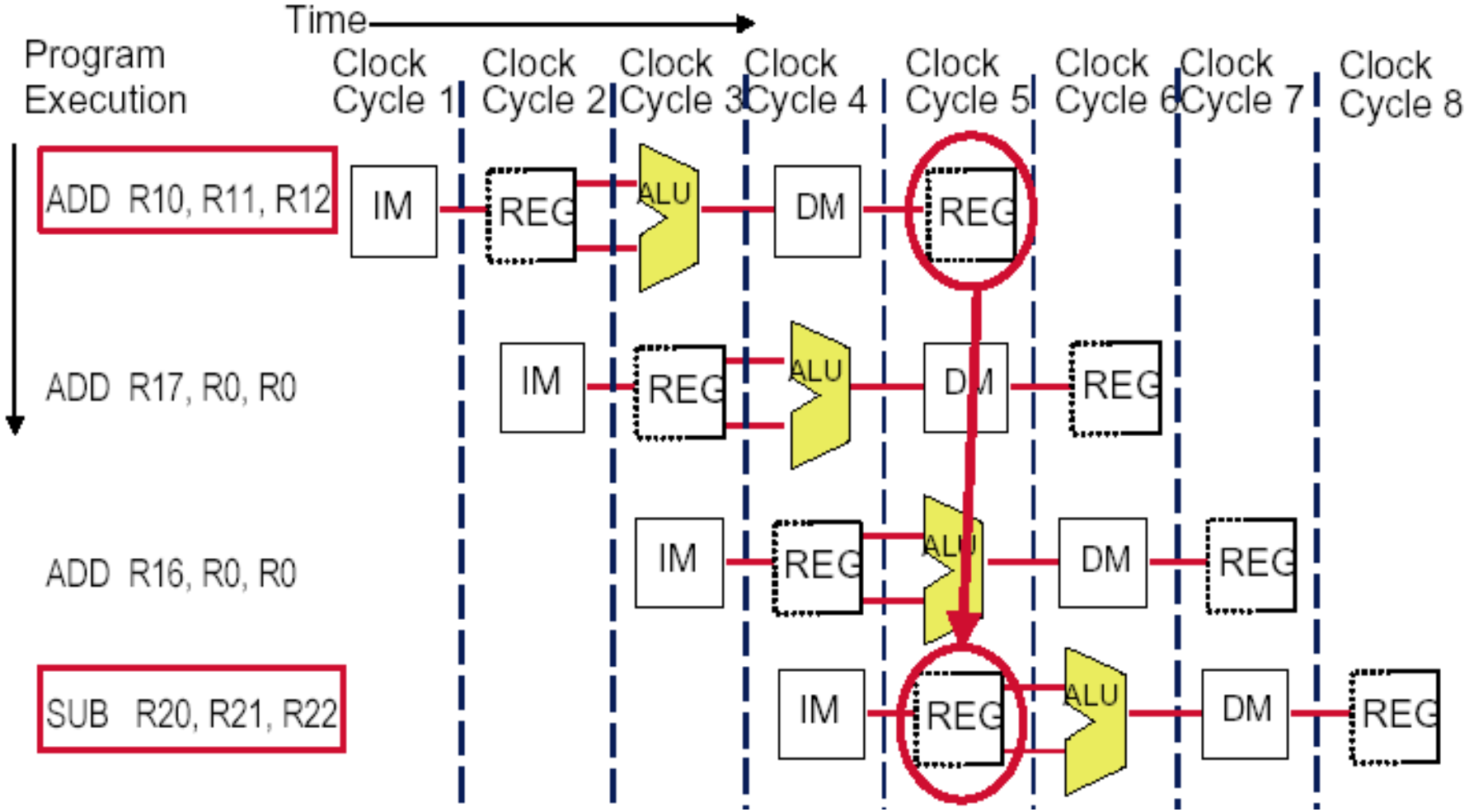
Problems that pipelining introduces

Focus: no different operations with the same data path resource on the same clock cycle. (structure hazard)

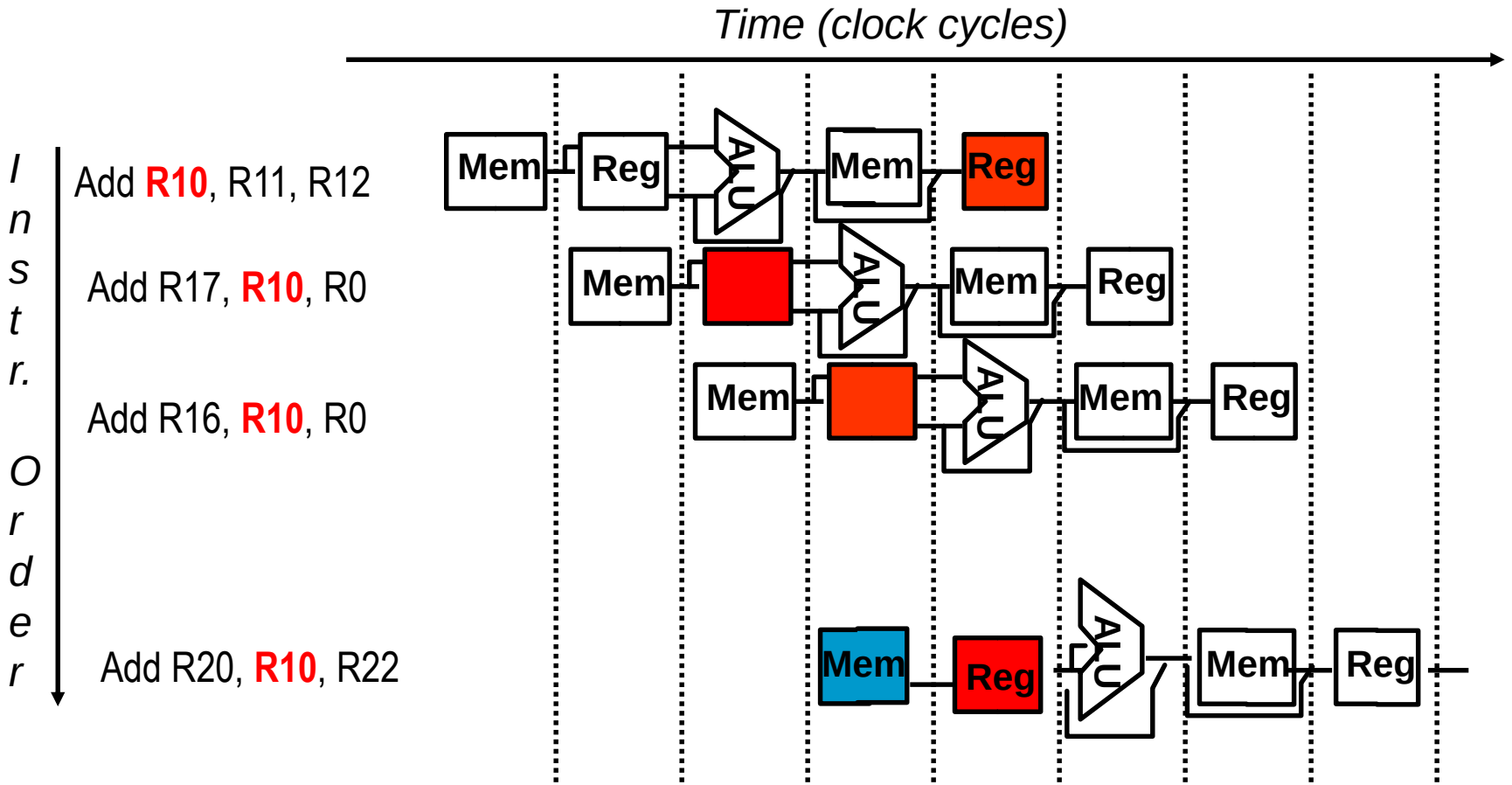
- There is conflict about the memory !



The conflict about registers !



Another kind of register conflict ! -time order problem



Conflict occurs when PC update

- Must increment and store the PC every clock.
- What happens when meet a branch ?
  - Branches change the value of the PC -- but the condition is not evaluated until ID !
  - If the branch is taken, the instructions fetched behind the branch are invalid !
- This is clearly a serious problem (Control hazard) that needs to be addressed. We will deal it later.

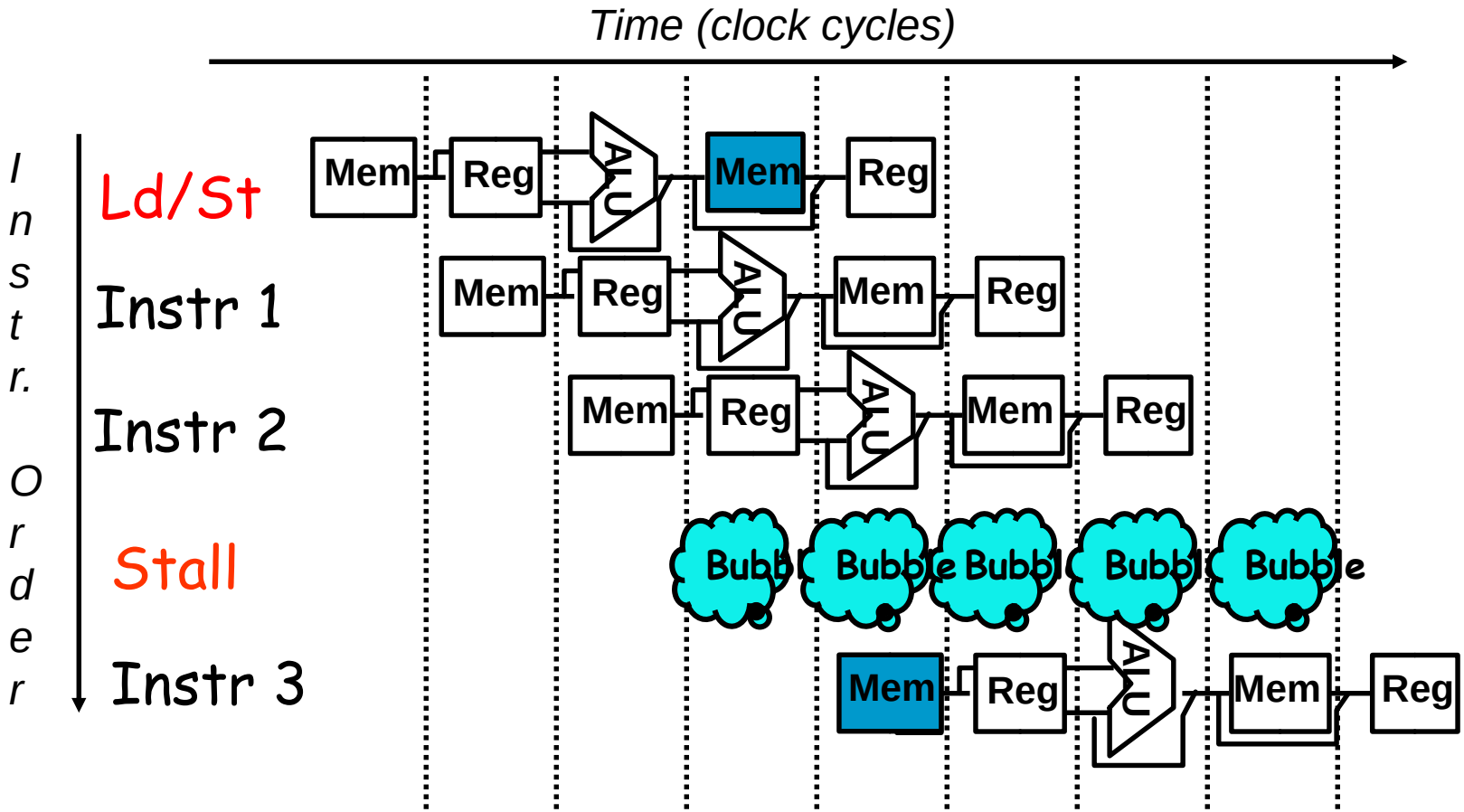
Pipeline hazard: the major hurdle

- *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 .

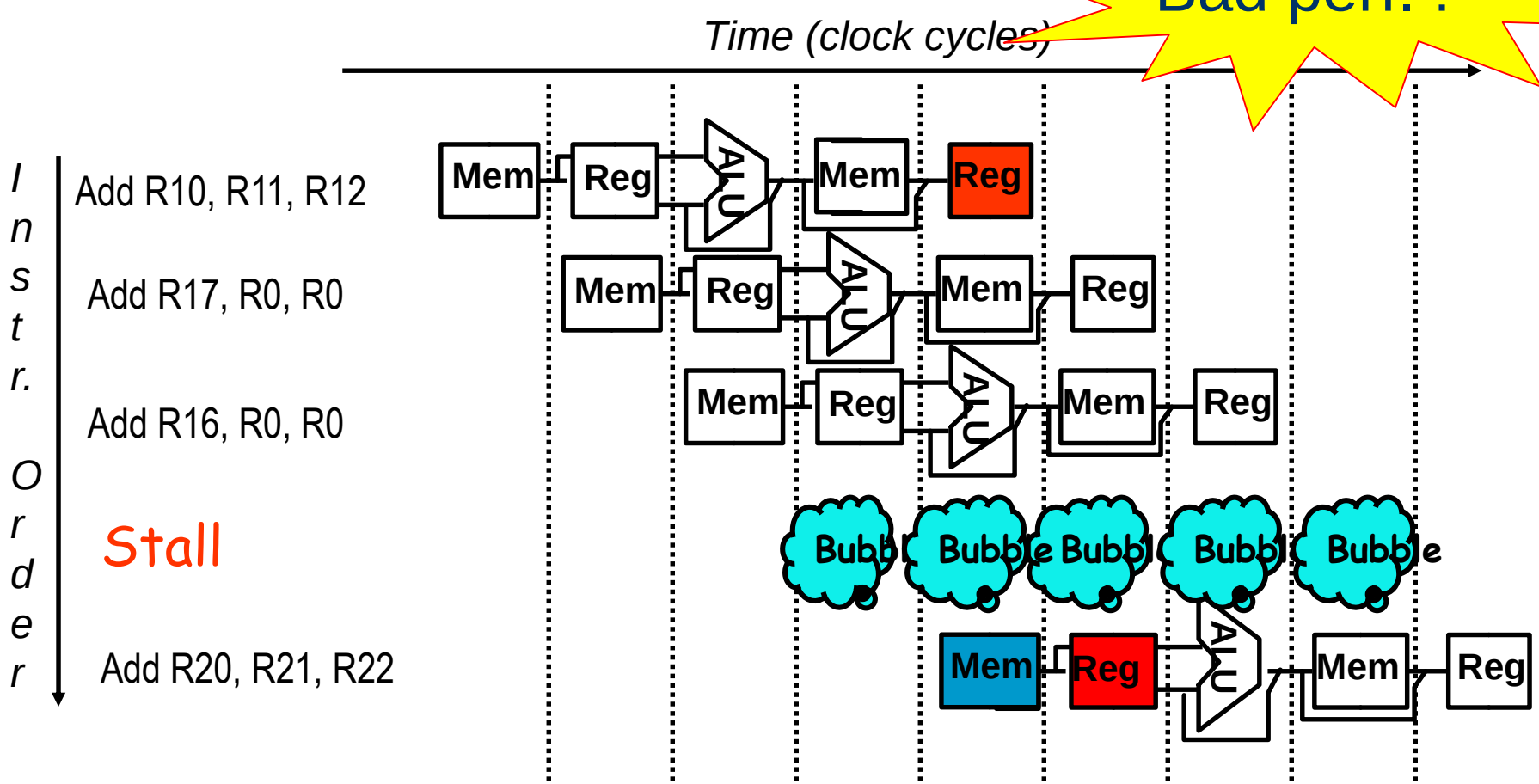
Insert Stall—simplest way



In general it can be drawn in another way.

Insert Stall—simplest way

Bad perf. !



How bad is the performance ?

- If given:
  - ALU op 50%
  - Load /store 25%
  - Branch 25%
- Solution:
  - CPI = 1 + (50%+25%)*1 = 1.75
  - Without considering data hazard, control hazard

Performance of pipeline with stalls

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

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

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

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

Assumptions for calculation

- The ideal CPI on a pipelined processor is almost always 1. (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.
- Two different implementation
  - single-cycle implementation
  - multi-cycle implementation

Case of single-cycle implementation

- CPI unpipelined = 1

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

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

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

Case of multi-cycle implementation

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

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

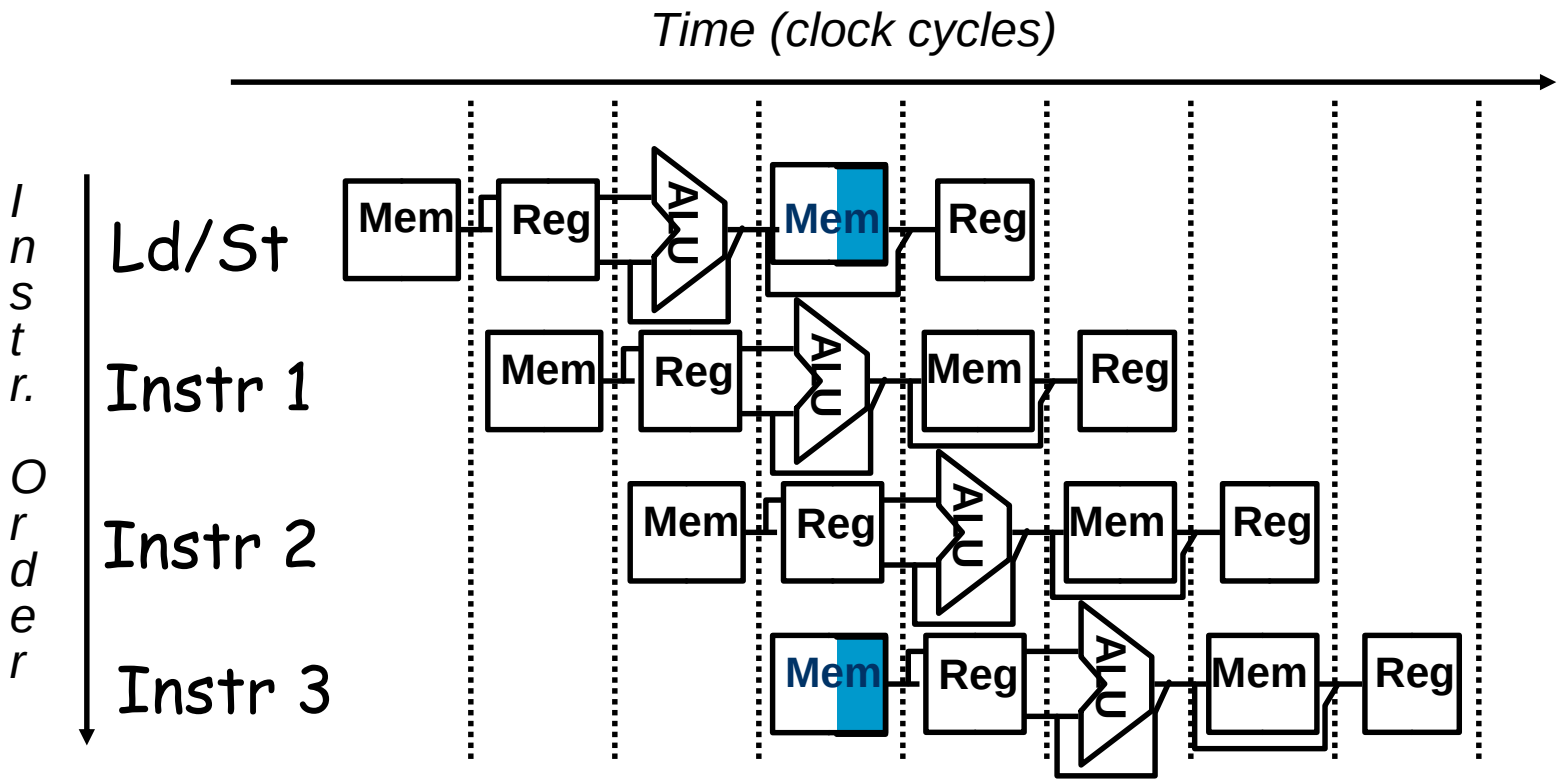
$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

So comes the approaches to decrease the stalls !

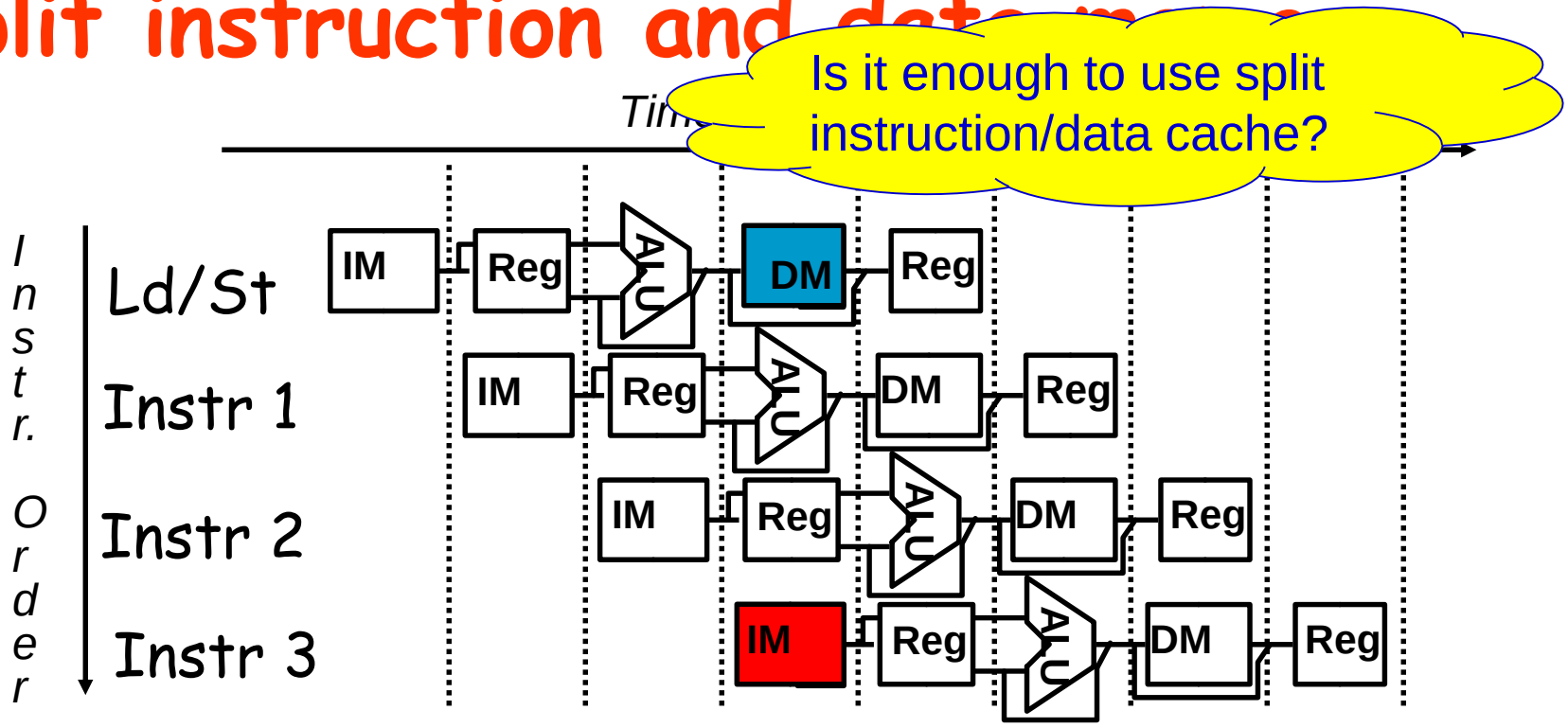
Structural hazard: Pipe Stage Contention

- **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 memory
    - Multiple accesses to the register file
    - Not fully pipelined functional units

Multi access to Single Memory Port

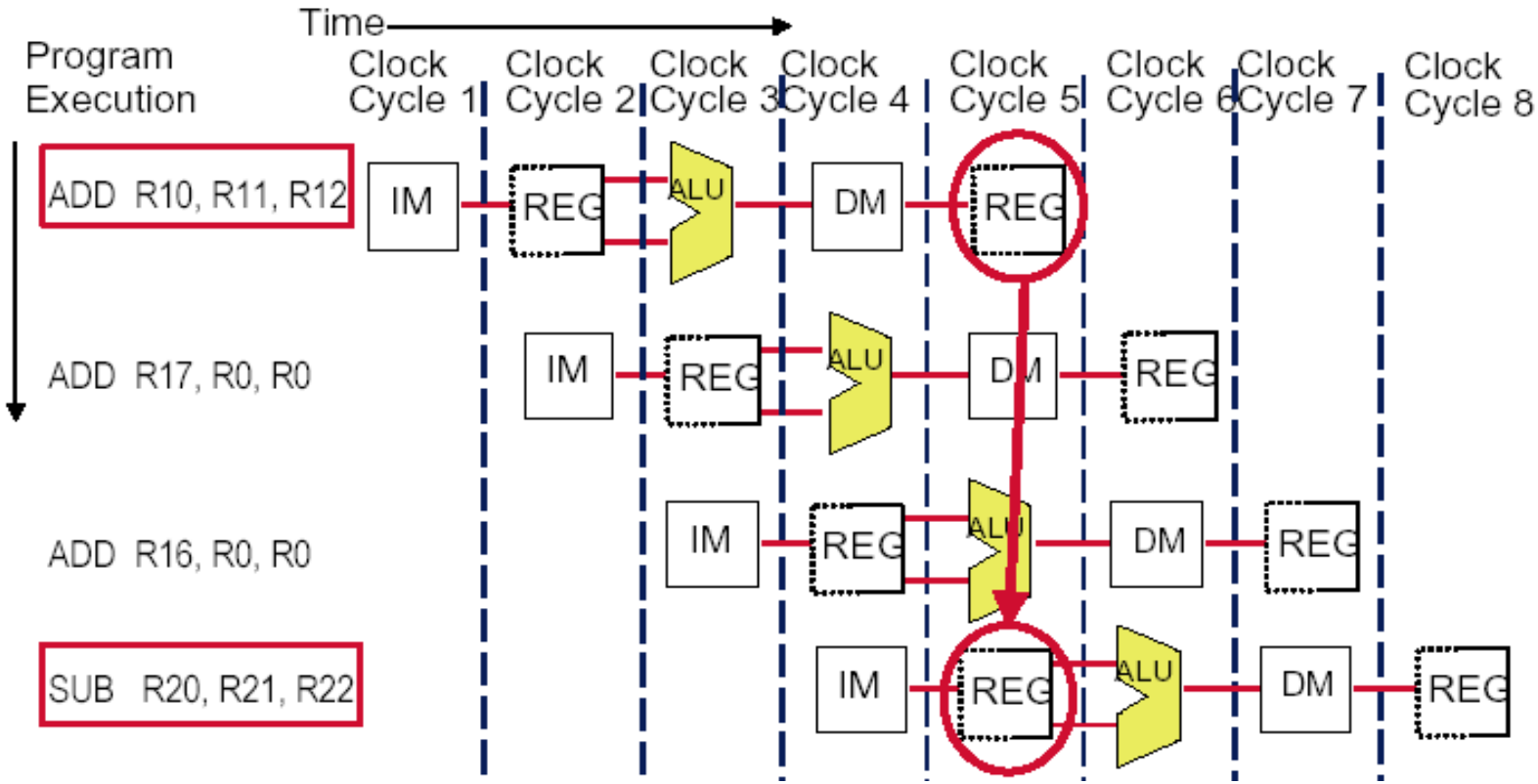


Split instruction and data m



- Split instruction and data memory / multiple memory port / instruction buffer means:
- fetch the instruction and data inference using different hardware resources.

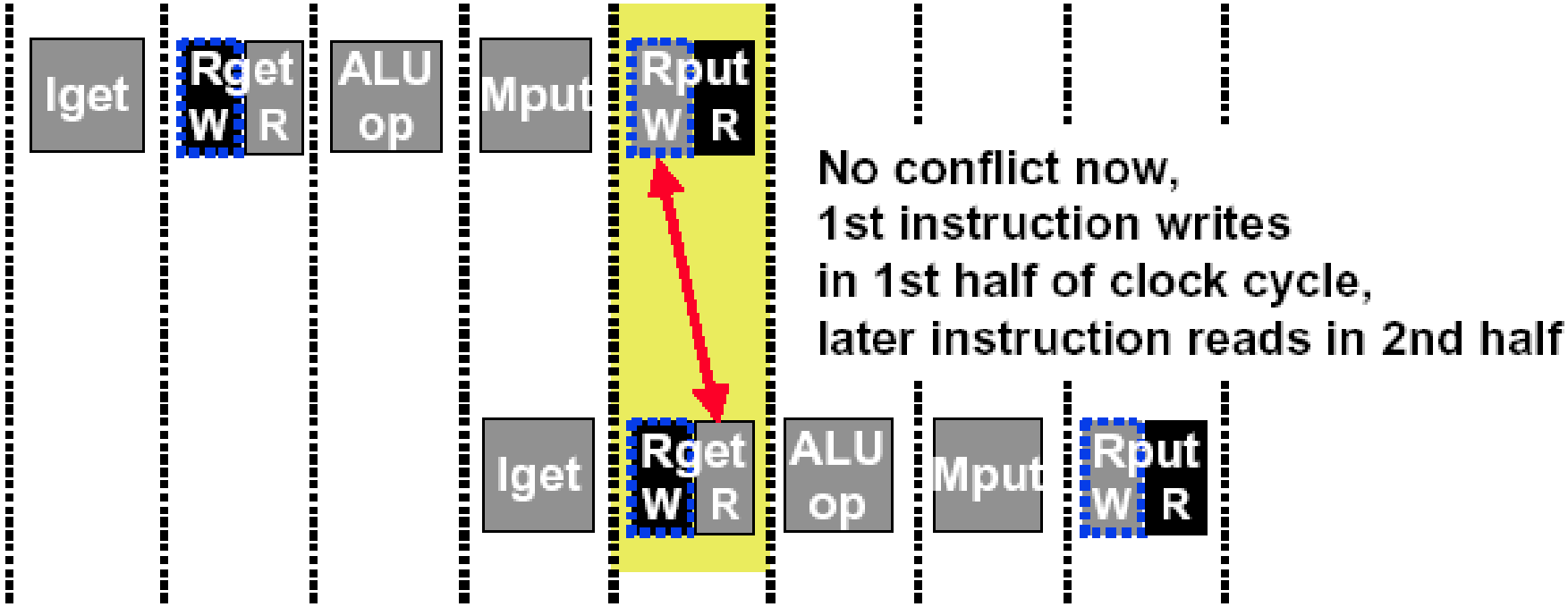
Multi access to the register file



- Simply insert a stall , speedup will be decreased.

double bump(双重触发) works

- allow WRITE-then-READ in one clock cycle (double bump)



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*
  - *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: solutions for structural hazard

- Multiple accesses to memory
  - Split instruction and data memory / multiple memory port / instruction buffer
  - Memory bandwidth need to be imporved by 5 folds.
- Multiple accesses to the register file
  - Double bump
- Not fully pipelined functional units
  - Fully pipeline the functional unit
  - Using multiple functional unites
- Real machine often with structural hazards.

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
    - Allow machine with Structural hazard, since it happens not so often
  - **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

Pipeline Hazards

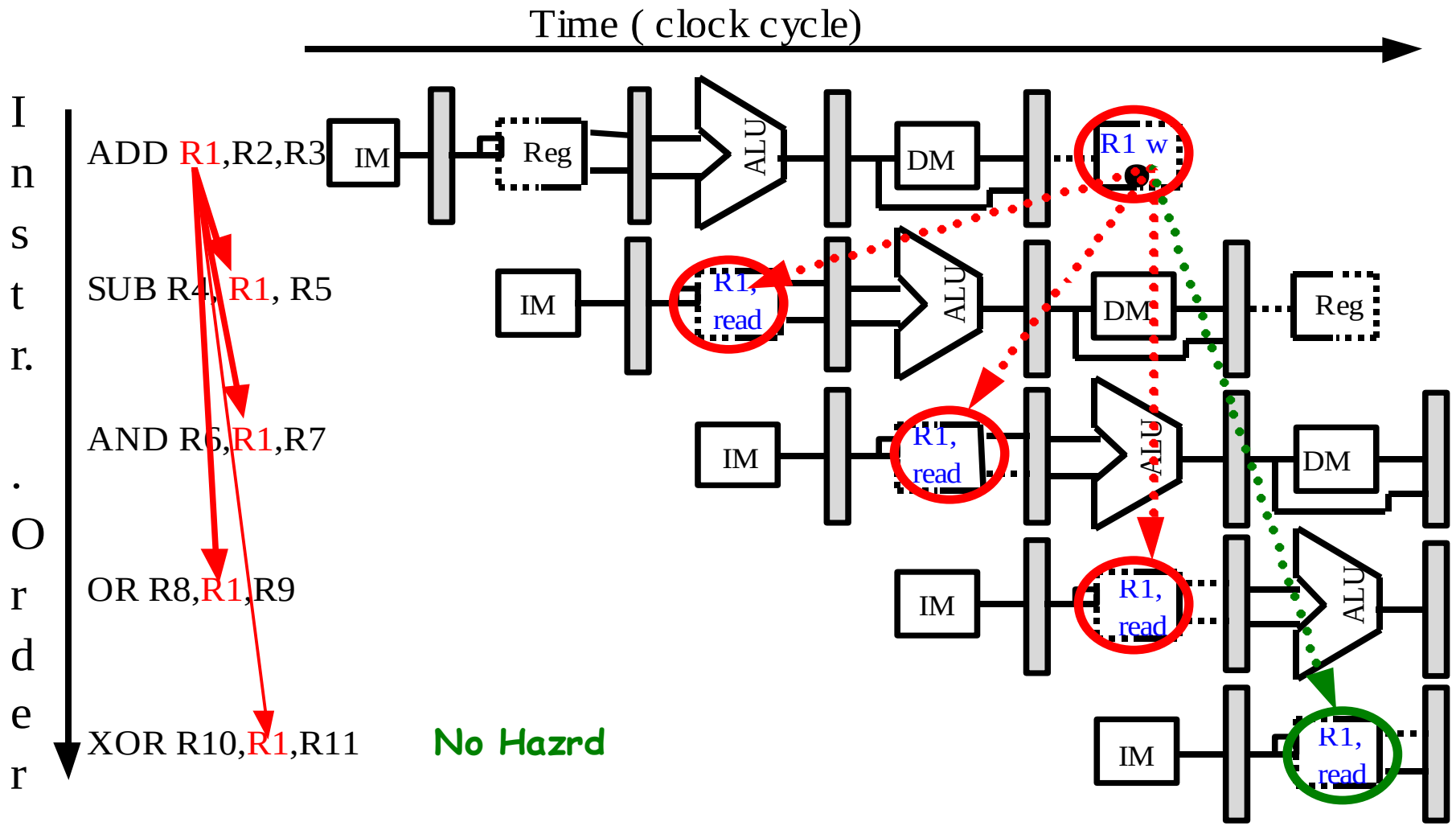
- **Taxonomy of Hazards**
  - Structural hazards
    - These are conflicts over hardware resources.
  - Data hazards
    - Instruction depends on result of prior computation which is not ready (computed or stored) yet
  - Control hazards
    - branch condition and the branch PC are not available in time to fetch an instruction on the next clock

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
```

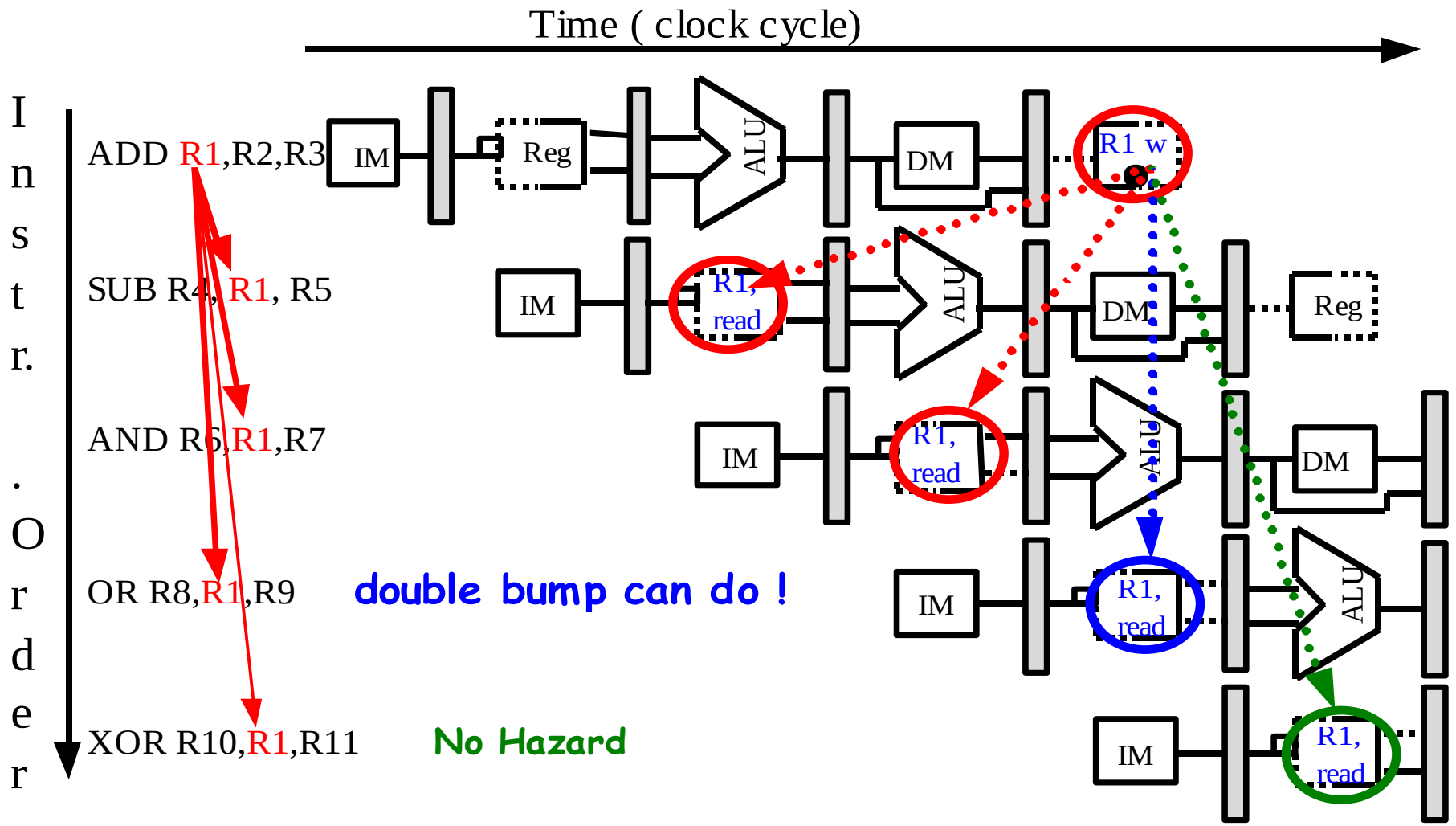
Coping with data hazards:example



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 not in the right place yet

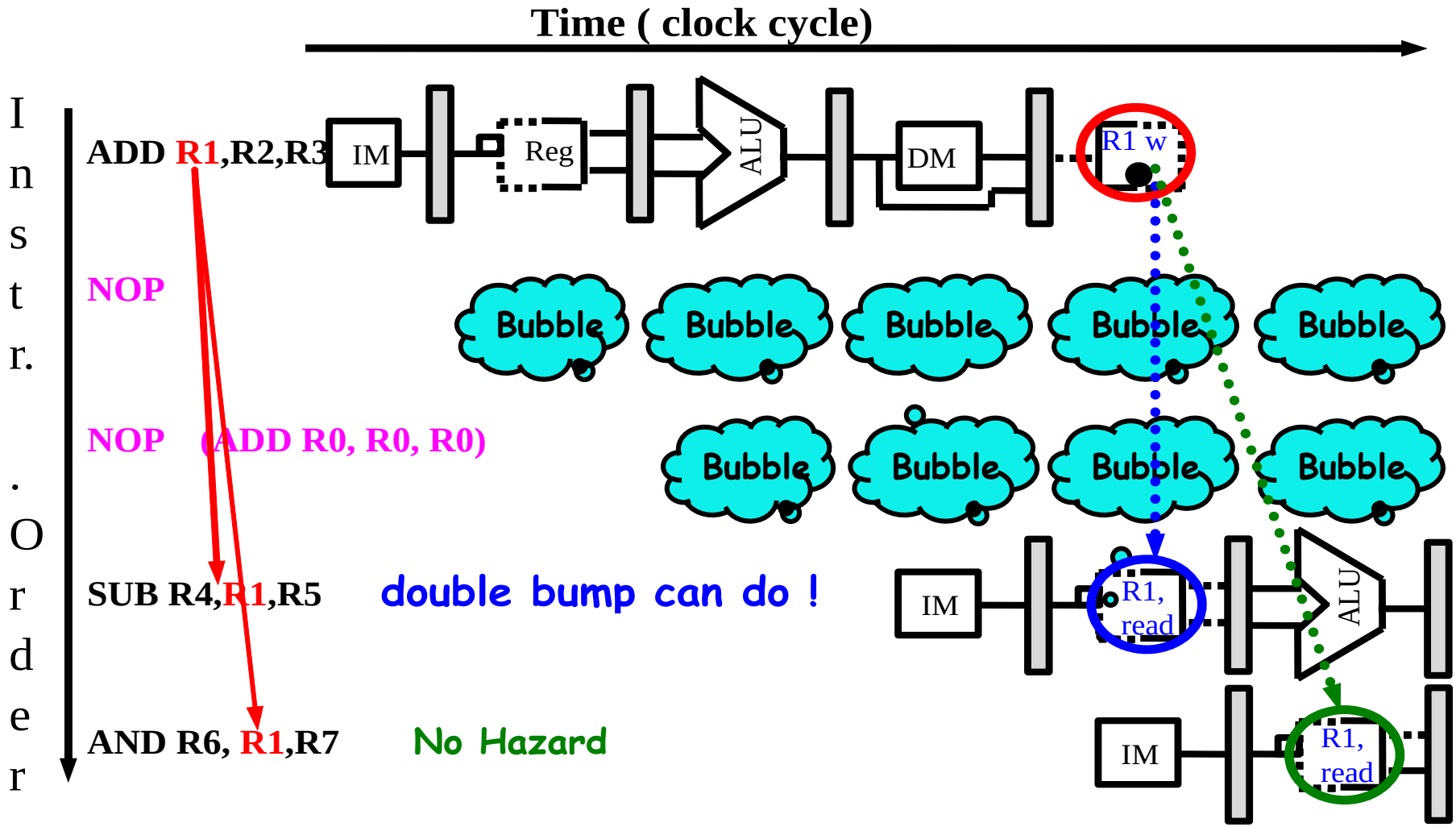
Somecases "Double Bump" can do !



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**

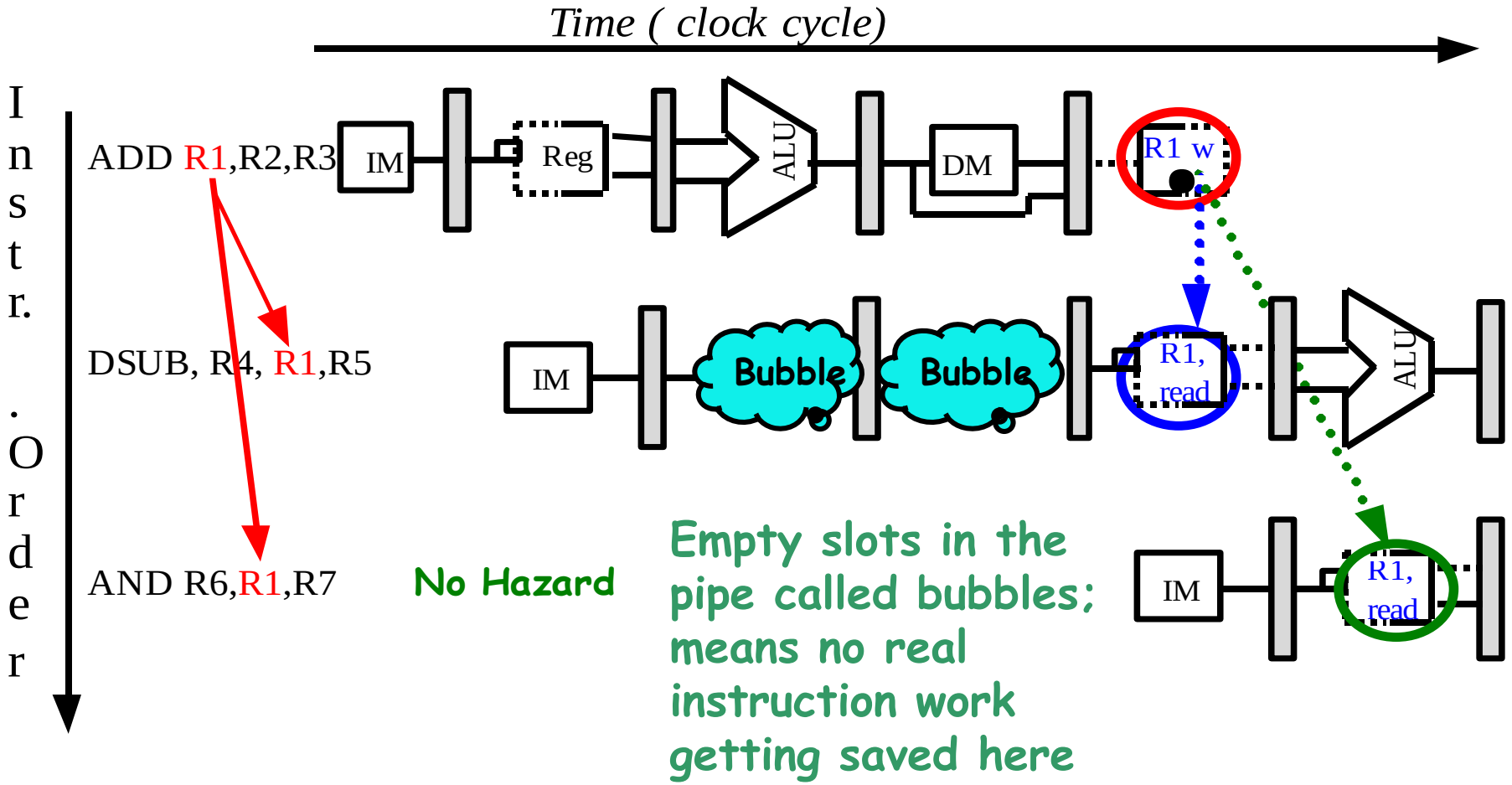
How do we stall ?
Insert nop by compiler



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"

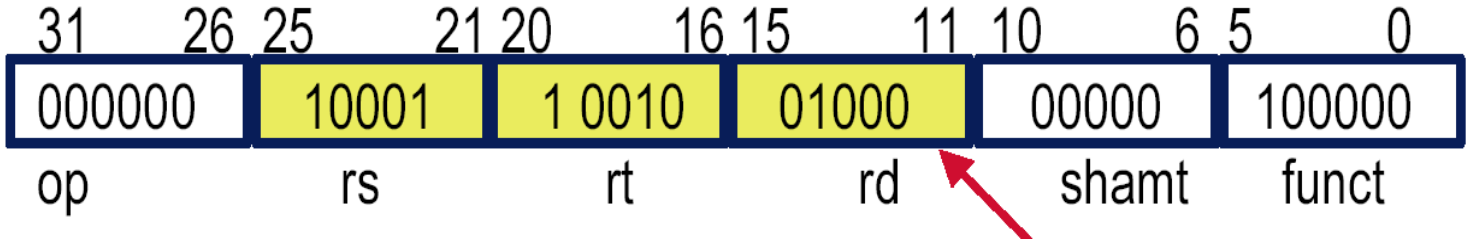
Interlock: insert stalls



How the interlock is implementated ?

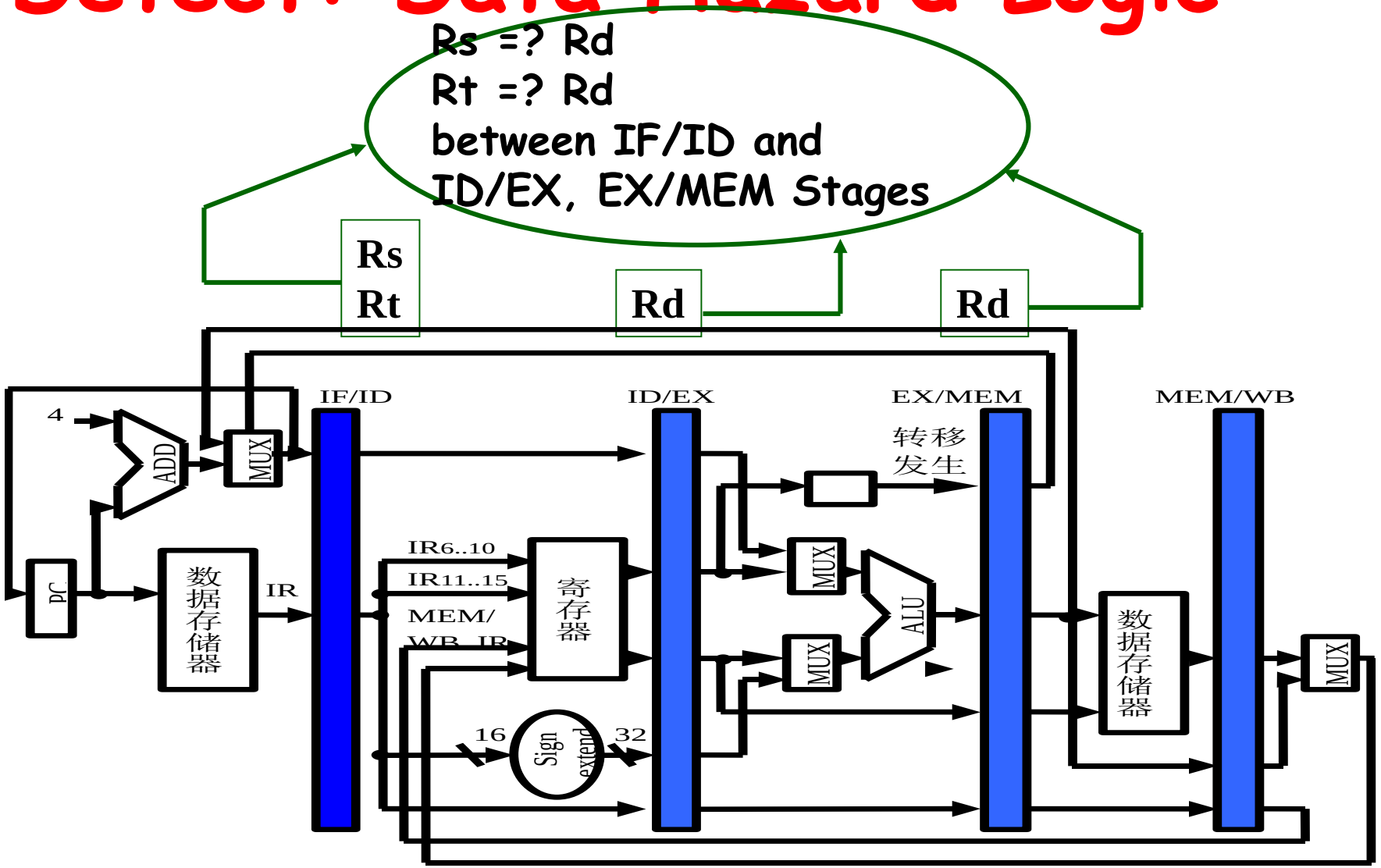
Recall MIPS Instruction format

- 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

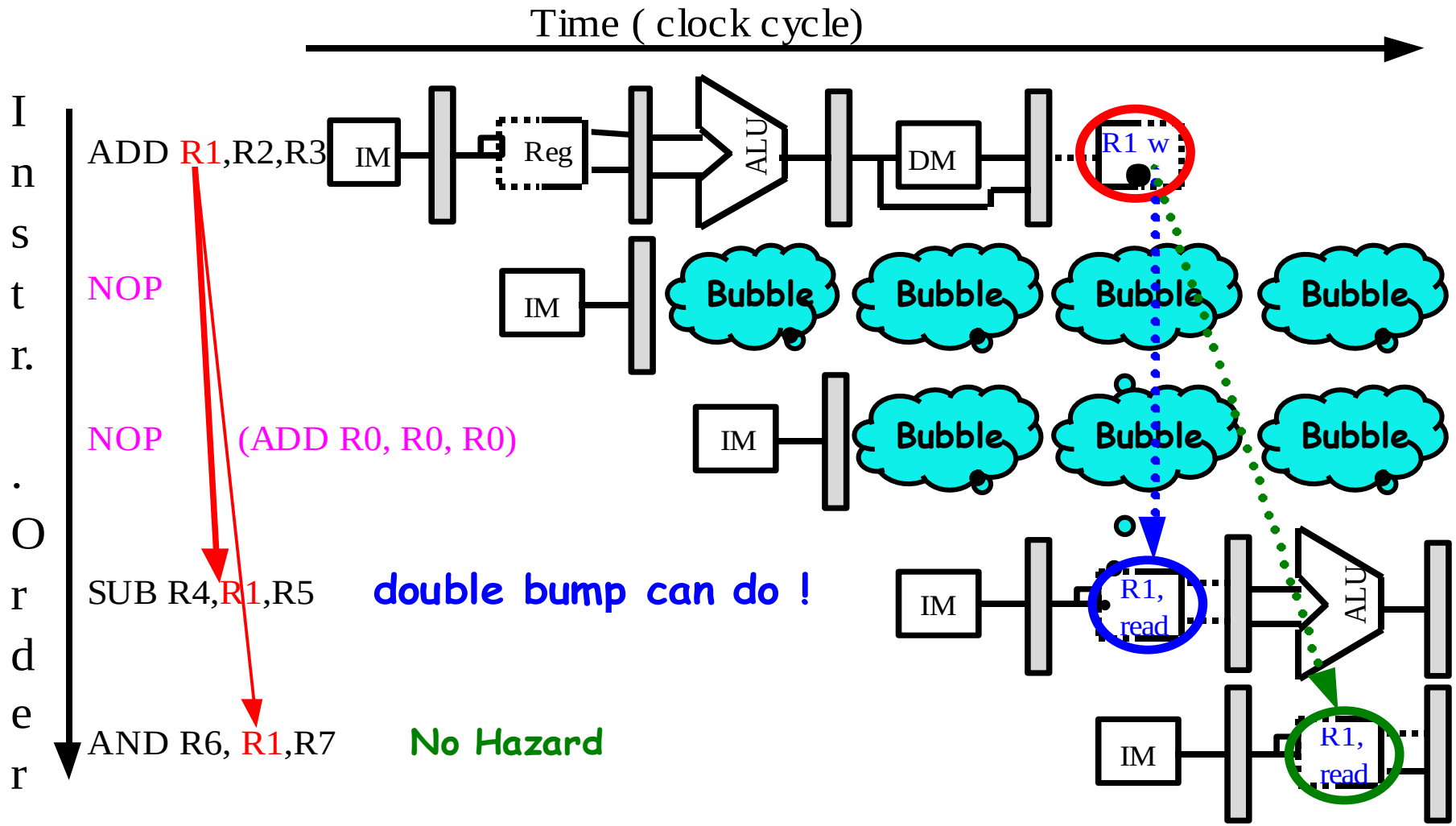
How to delay the instruction ?

- 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.
    - disable the write signal: "wreg, wmem".
  - Reserve the IF/ID. IR unchanged for one more clock cycle.
    - Disable the write signal: "WritePC, WriteIR".

Hardware simulates NOP



Forwarding: reduce data hazard stalls

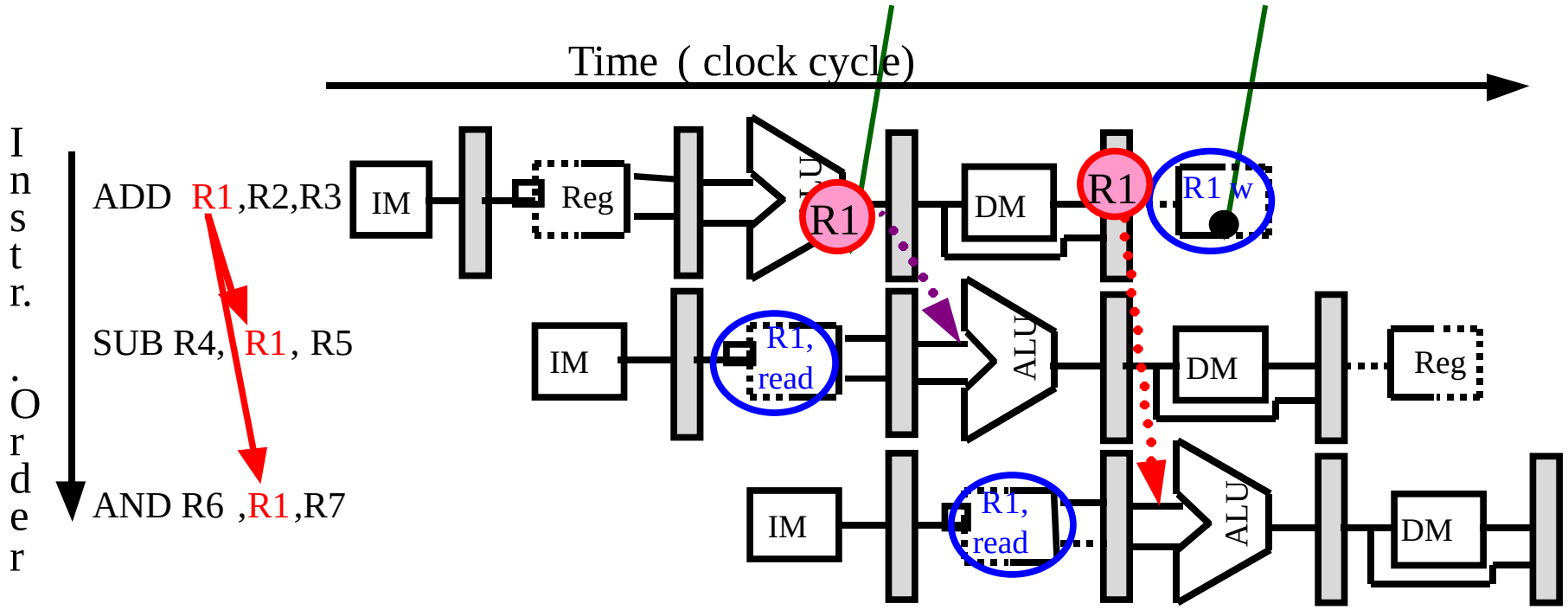
- If the result you need does not exist AT ALL yet,
  - you are out of luck, sorry.
- But, what if the result exists, but is not stored back yet?
  - Instead of stalling until the result is stored back in its "natural" home…
  - **grab the result "on the fly" from "inside" the pipe, and send it to the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

Forwarding

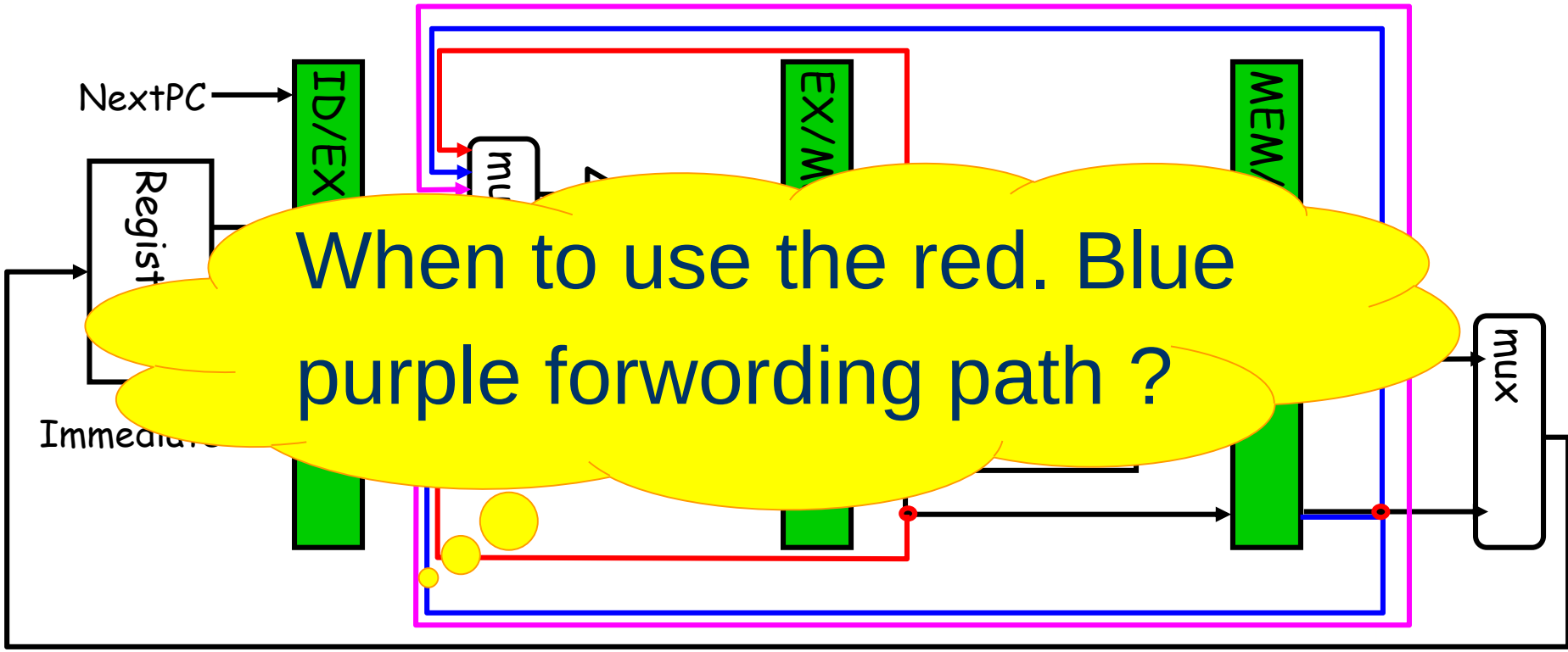
- Generic name: forwarding (bypass, short-circuiting)
  - Instead of waiting to store the result, we forward it immediately (more or less) to the instruction that wants it
  - Mechanically, we add buses to the datapath to move these values
  - around, and these buses always "point backwards" in the datapath,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



Source → sink

- EX/Mem.ALUoutput → ALU input
- MEM/WB.ALUoutput → ALU input
- MEM/WB.LMD → ALU input

When to use the forwarding path ?

- **EX/Mem.ALUoutput → ALU** input
  - The previous instruction in EX/MEM is ALU
  - The instruction in ID/EX has Rs or Rt source register
  - EX/MEM.Rd == ID/EX.Rs or EX/MEM.Rd ==ID/EX.Rt
- **MEM/WB.ALUoutput → ALU input**
  - The previous instruction in MEM/WB is ALU
  - The instruction in ID/EX has Rs or Rt source register
  - MEM/WB.Rd == ID/EX.Rs or MEM/WB.Rd == IF/ID.Rt
- **MEM/WB.LMD → ALU input**
  - The previous instruction in MEM/WB is Load
  - The instruction in ID/EX has Rs or Rt source register
  - MEM/WB.Rd == ID/EX.Rs or MEM/WB.Rd == ID/EX.Rt

How to know if it's ALU or Load ?

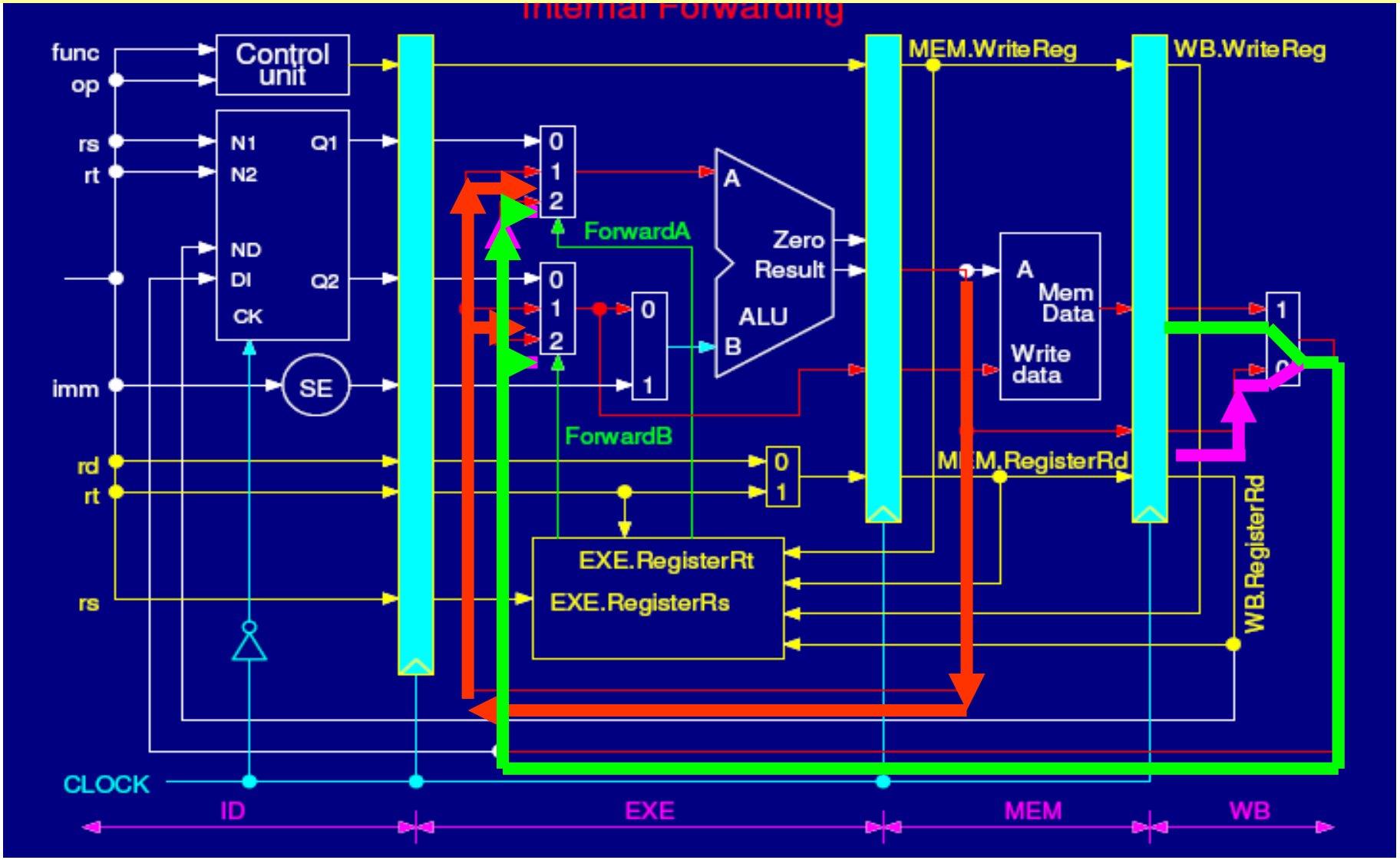
- ALU:
  - Wreg == 1 && W2Reg == 0
- Load:
  - Wreg ==1 && W2Reg == 1
- Rd
  - If ALU reg-reg, then Rd = Rd
  - If Load or ALU reg-imm, then Rd = Rt
- Rs or Rt
  - If it's Not JMP then it has Rs
  - If it's ALU reg-reg or Store or Branch then it has Rt

3 Forwarding paths:

EX/MEM.ALUoutput → ALU input port

MEM/WB.ALUoutput → ALU input port (0)

MEM/WB.LDMR → ALU input port(1)



Internal forwarding Logic

- 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

How to select the forwarding path: the forwarding logic

- PA-36 in Edition 3th and Edition 4th

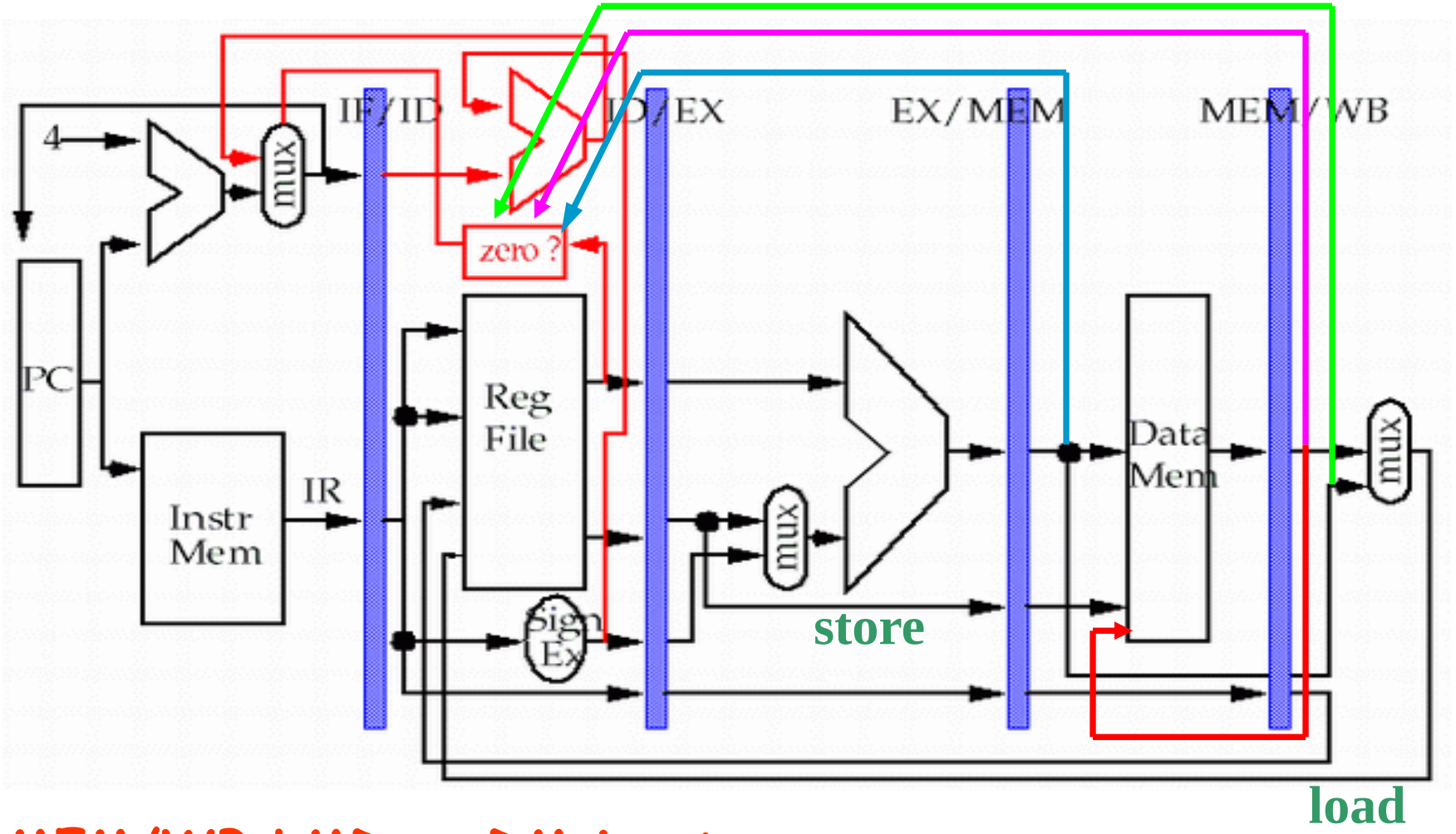
含源指令的锁存器	源指令的操作	含目标指令的锁存器	目标指令的操作(后继指令)	旁路电路的输入端	比较检测(相等则直接送结果
EX/MEM	R-R ALU	ID/EX	ALU,LD,ST,Branch	ALU 上端	EX/MEM.$IR_{16..20}$=ID/EX.$IR_{6..10}$
EX/MEM	R-R ALU	ID/EX	R-R ALU	ALU 下端	EX/MEM.$IR_{16..20}$=ID/EX.$IR_{11..15}$
MEM/WB	R-R ALU	ID/EX	ALU,LD,ST,Branch	ALU 上端	MEM/WB.$IR_{16..20}$=ID/EX.$IR_{6..10}$
MEM/WB	R-R ALU	ID/EX	R-R ALU	ALU 下端	MEM/WB.$IR_{16..20}$=ID/EX.$IR_{11..15}$
EX/MEM	R-I ALU	ID/EX	ALU,LD,ST,Branch	ALU 上端	EX/MEM.$IR_{11..15}$=ID/EX.$IR_{6..10}$
EX/MEM					EX.$IR_{11..15}$
					EX.$IR_{6..10}$
					.$IR_{11..15}$
					.$IR_{11..15}$=ID/EX.$IR_{6..10}$
MEM/WB				ALU 下端	MEM/WB.$IR_{11..15}$=ID/EX.$IR_{11..15}$

EX/MEM.Rd == ID/EX.rs or rt

What's the difference with

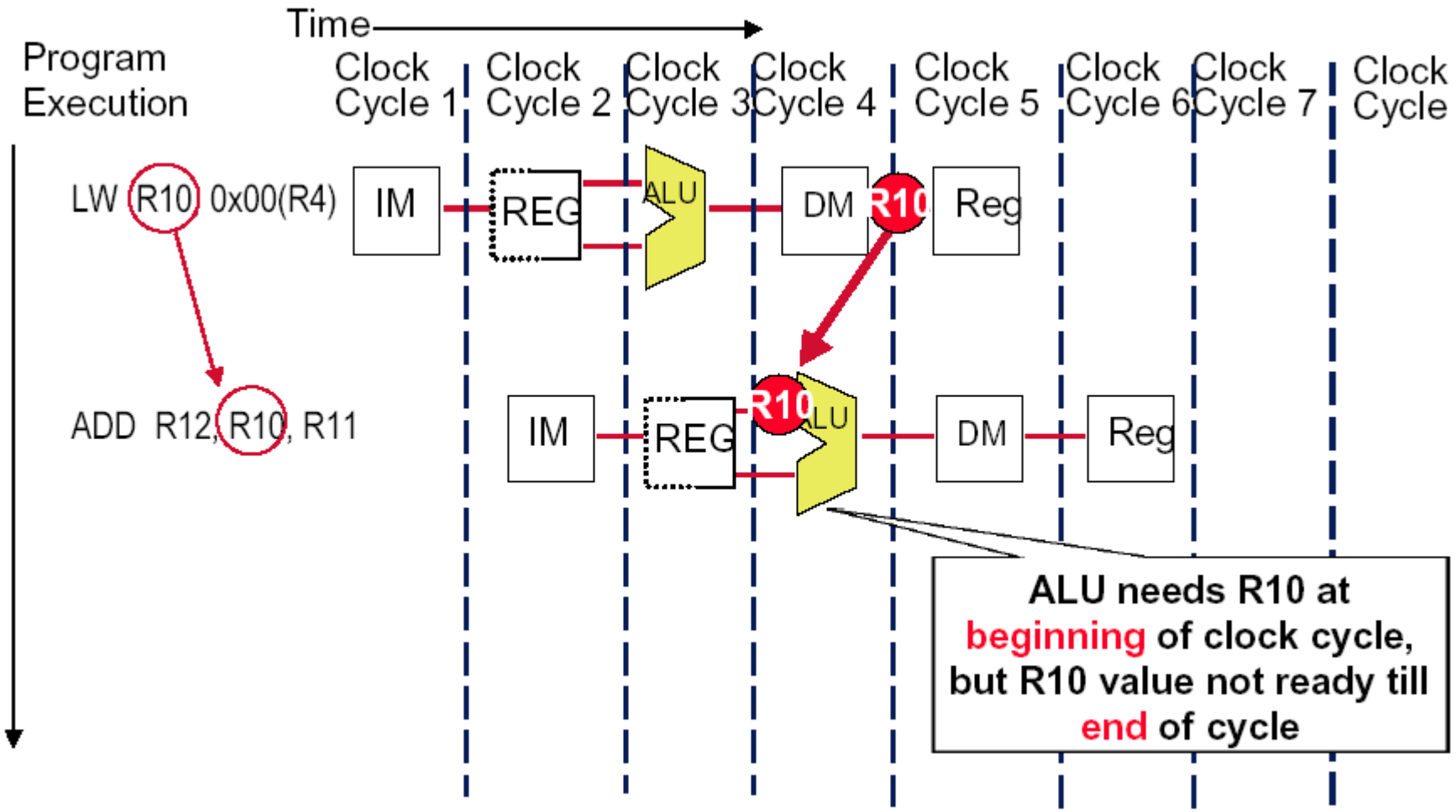
ID/EX.Rd == IF/ID.rs or rt ?

Forwarding path to other input entry



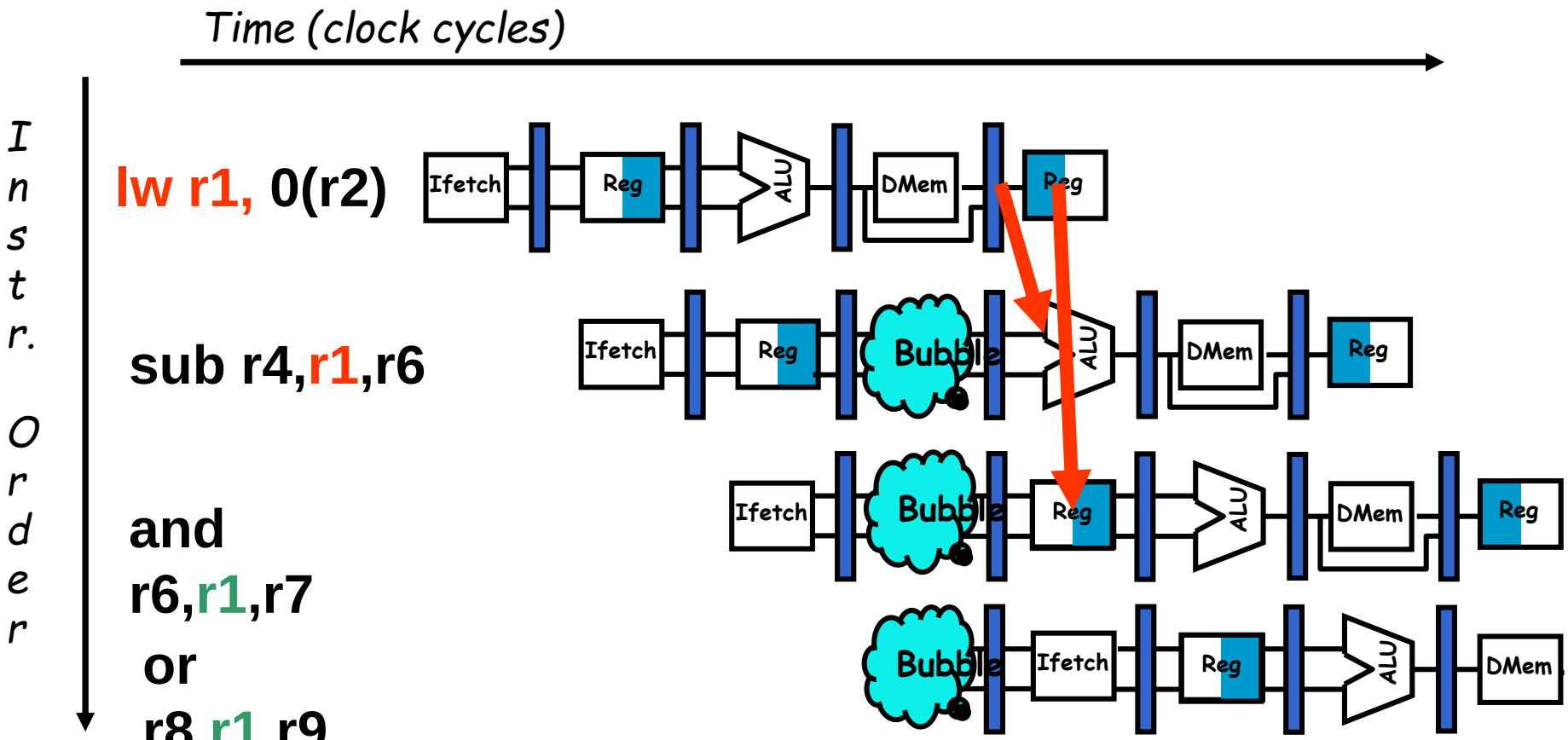
MEM/WB.LMD → DM input

Forwarding Doesn't Always Work



So we have to insert stall:

Load stall



How to implement Load Interlock

- **Detect** when should use Load Interlock PA34

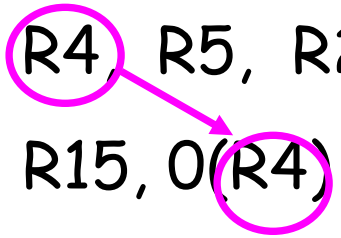
situation	Example code sequence	Action
No dependence	LD R1, 45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because of no dependence
Dependence requiring stall	LD R1, 45(R2) DADD R5,R1,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and 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

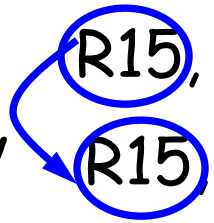
Opcode field of ID/EX	Opcode Field of IF/ID	Matching operand fields
Load	Reg-Reg ALU	ID/EX.IR[rt]==IF/ID.IR[rs]
Load	Reg-Reg ALU	ID/EX.IR[rt]==IF/ID.IR[rt]
Load	Load,store, ALU immediate, branch	ID/EX.IR[rt]==IF/ID.IR[rs]

Example of Forwarding and Load Delay

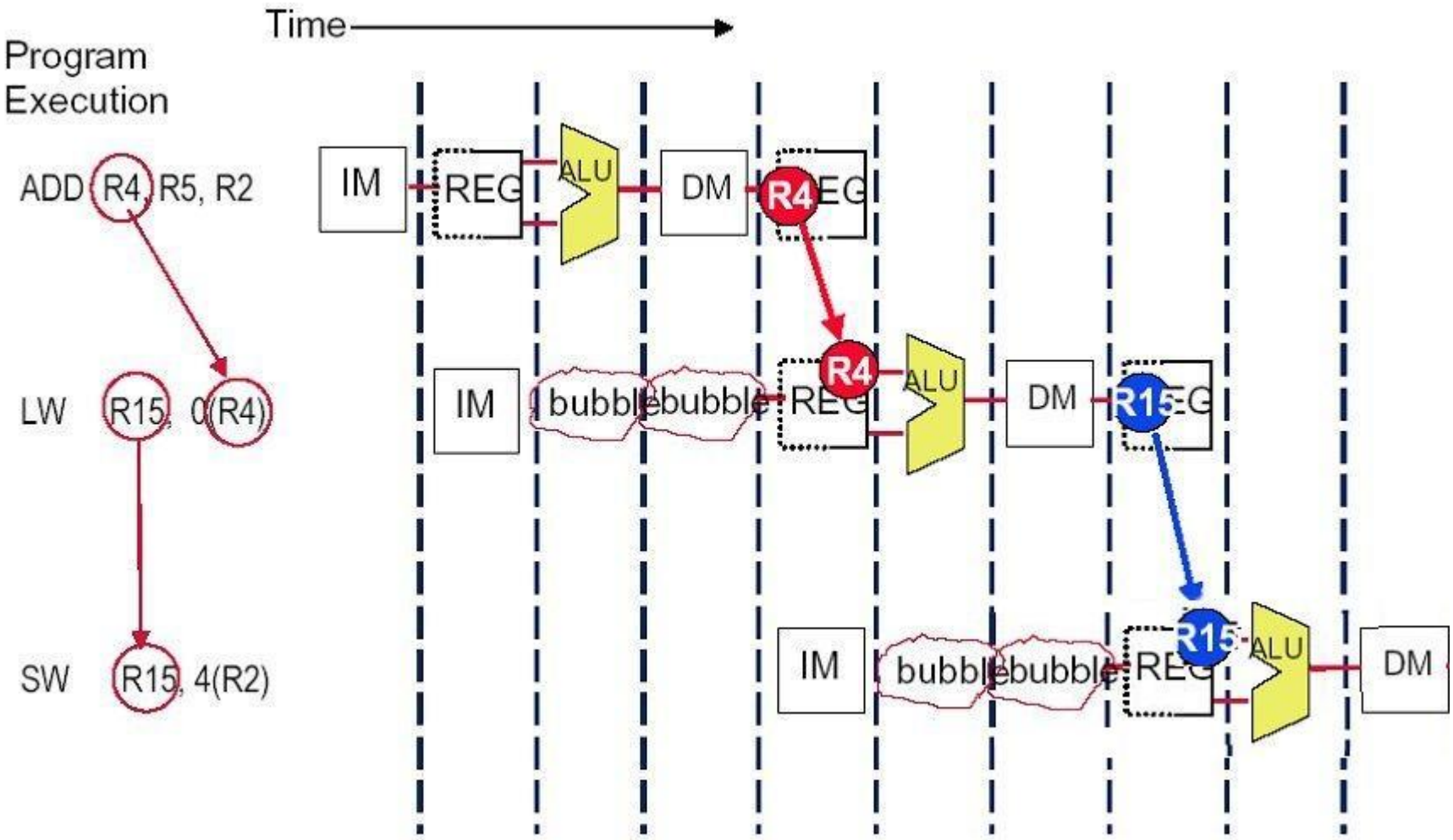
- Why forwarding?
  - ADD R4, R5, R2
  - LW R15, 0(R4)
  - SW R15, 4(R2)



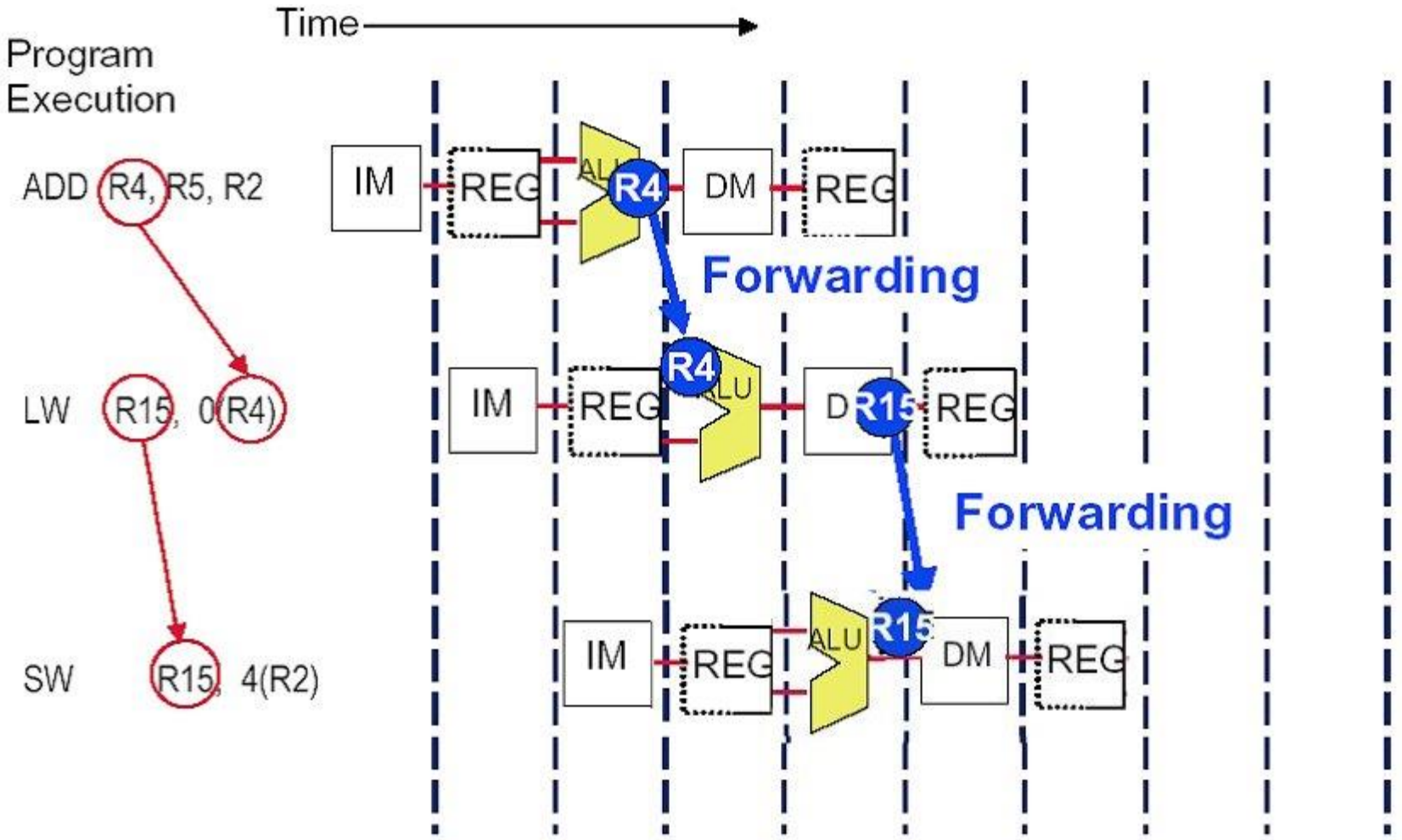
- Why load delay?
  - ADD R4, R5, R2
  - LW R15, 0(R4)
  - SW R15, 4(R2)



Solution (without forwarding)



Solution (with forwarding)

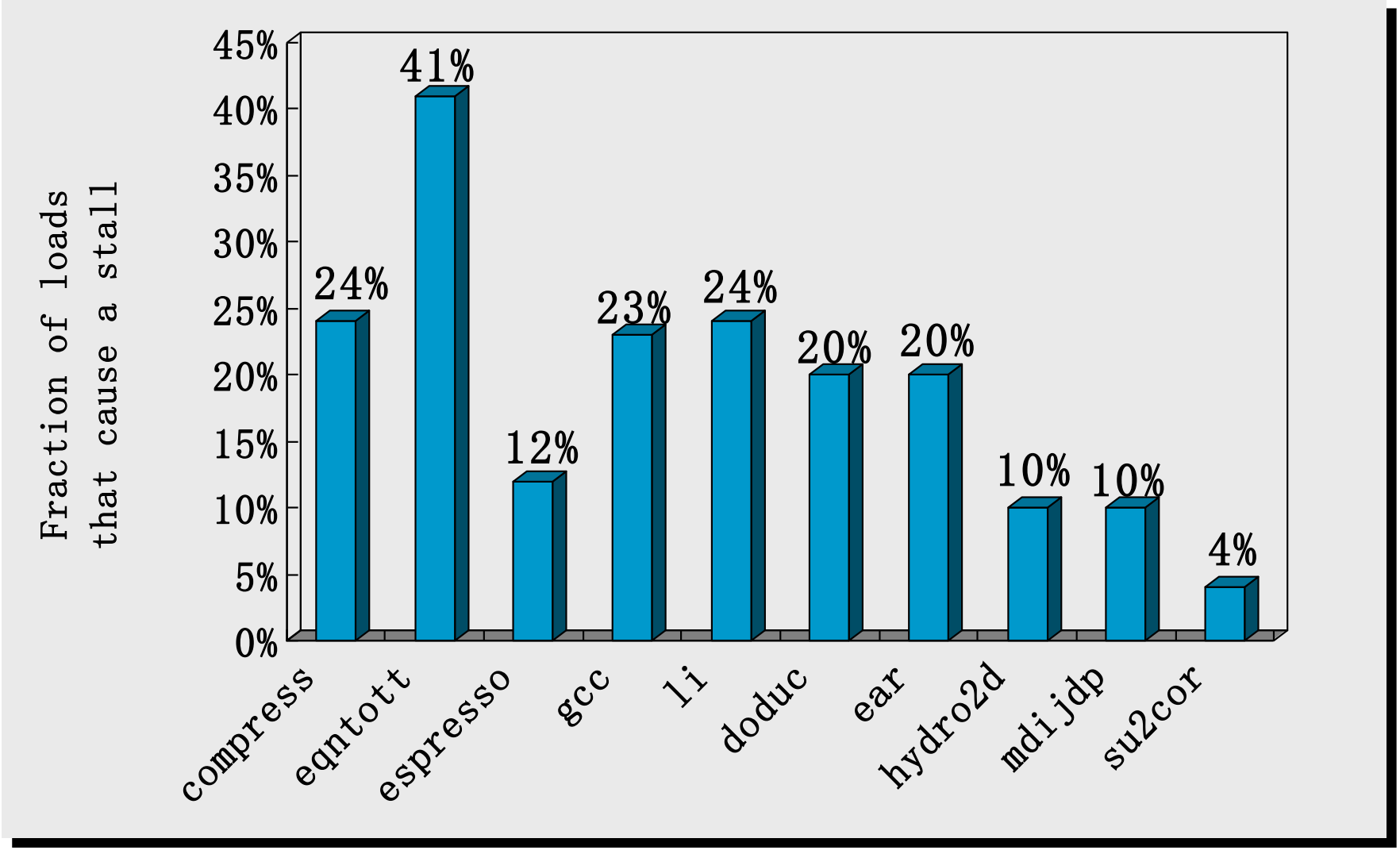


EX/MEM.ALUoutput → ALU input port
MEM/WB.ALUoutput → DM data write port

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.

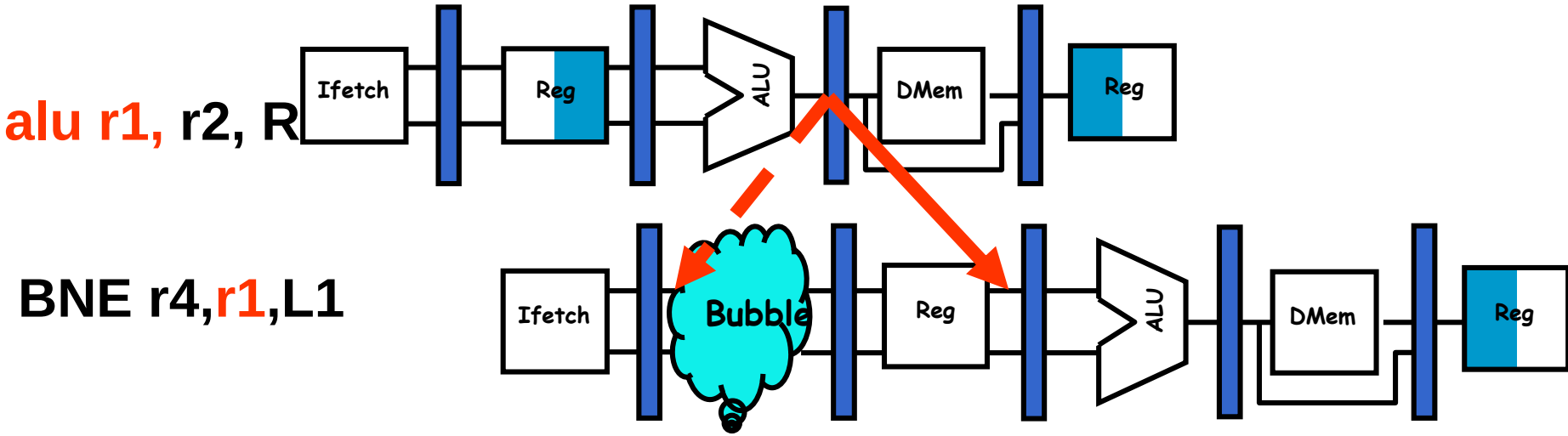
- **Slow code:**

```
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
```

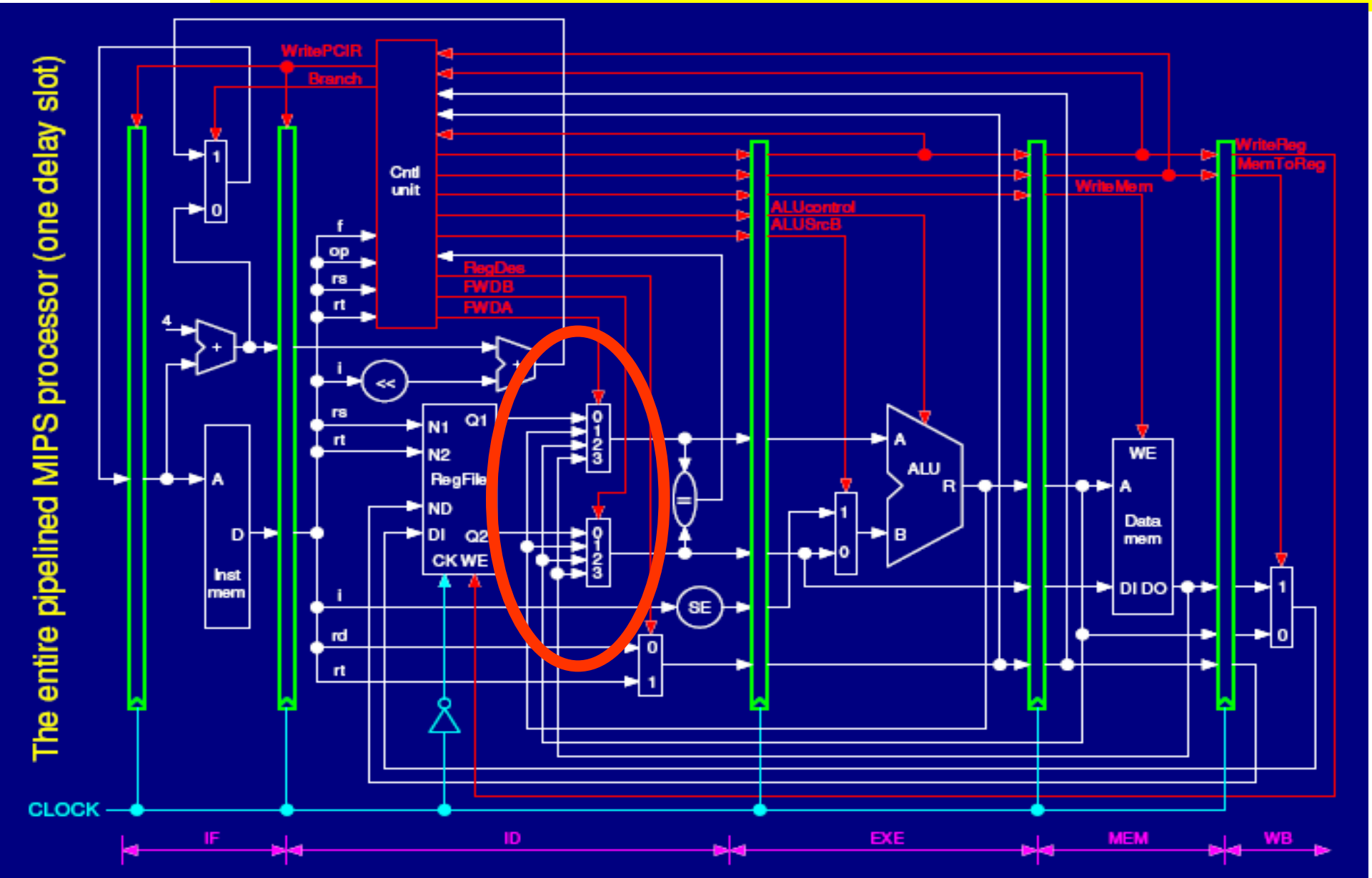
Another case: has stall even with forwarding path



Assume branch is resolved in ID stage.

Wh

- Move the Forwarding path to ID stage
- Move the forwarding control logic to ID stage



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

Related materials

- an overview of pipelining computer processors
  - http://granite.sru.edu/~venkatra/pipelining1.html
- pipelining: a summery
  - http://www-ee.eng.hawaii.edu/~tep/EE461/Notes/Pipeline/summary.html