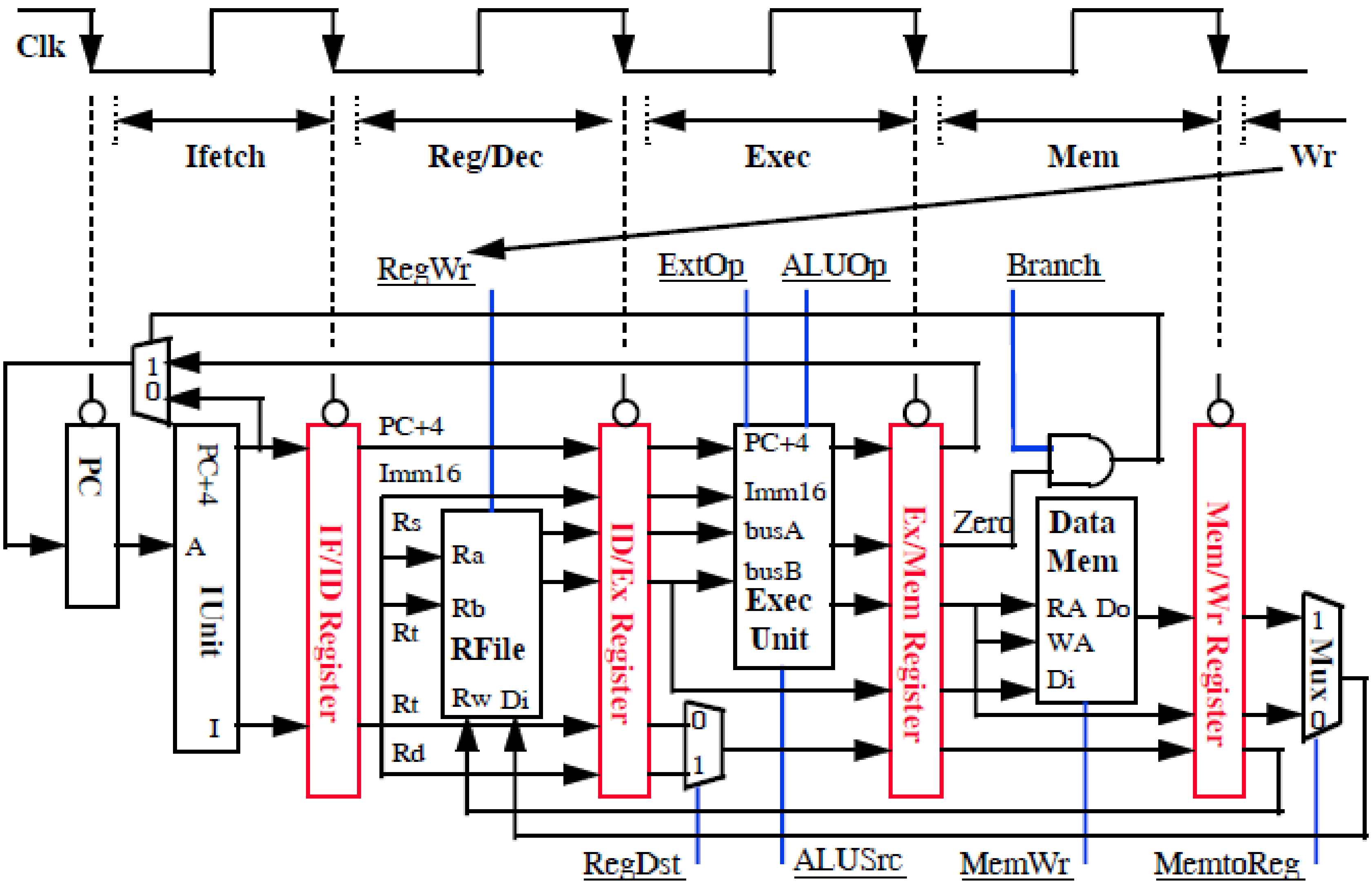
Acknowledgment: Almost all of these slides are based on Dave Patterson's CS152 Lecture Slides at UC, Berkeyley.

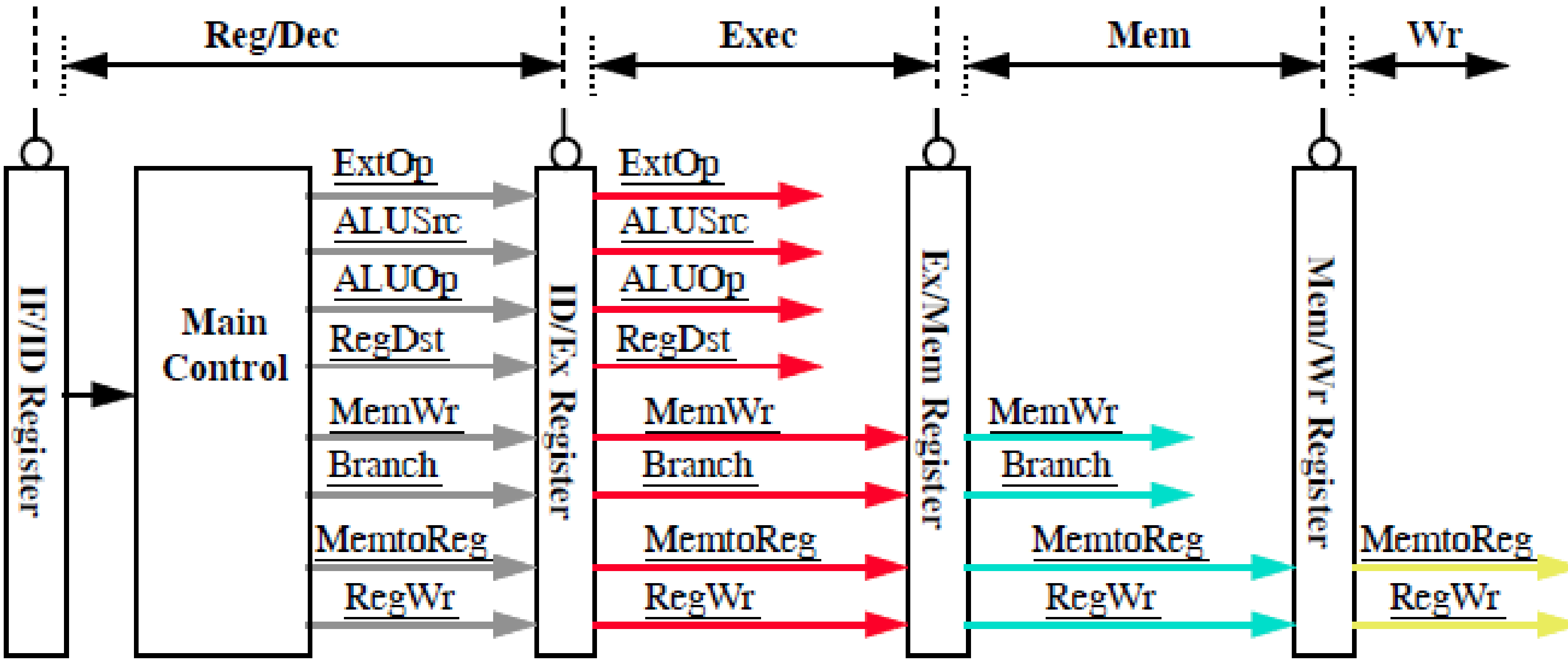
COMPUTER SYSTEMS ORGANIZATION

A Pipelined Datapath



Pipeline Control "Data Stationary Control"

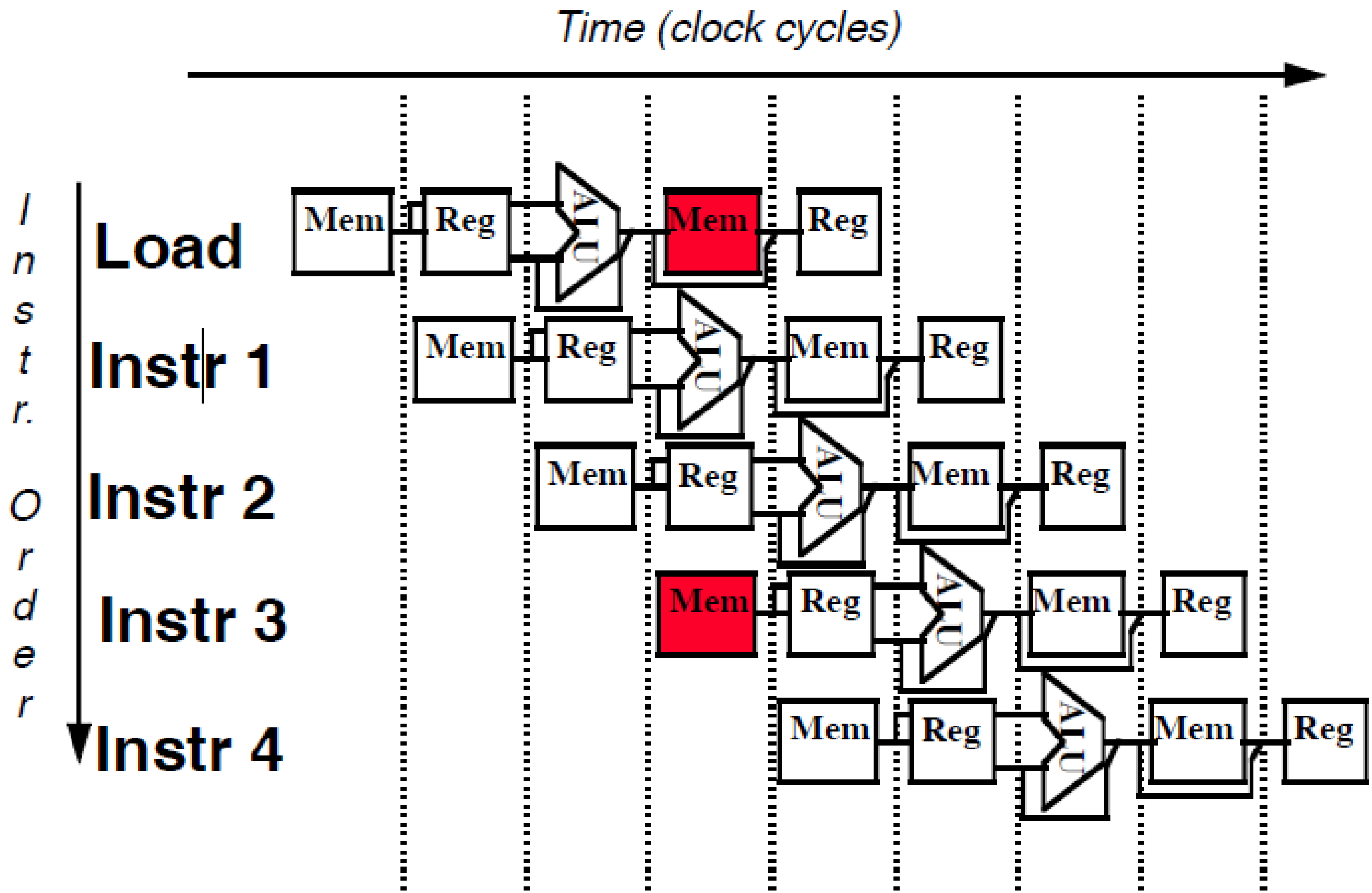
- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



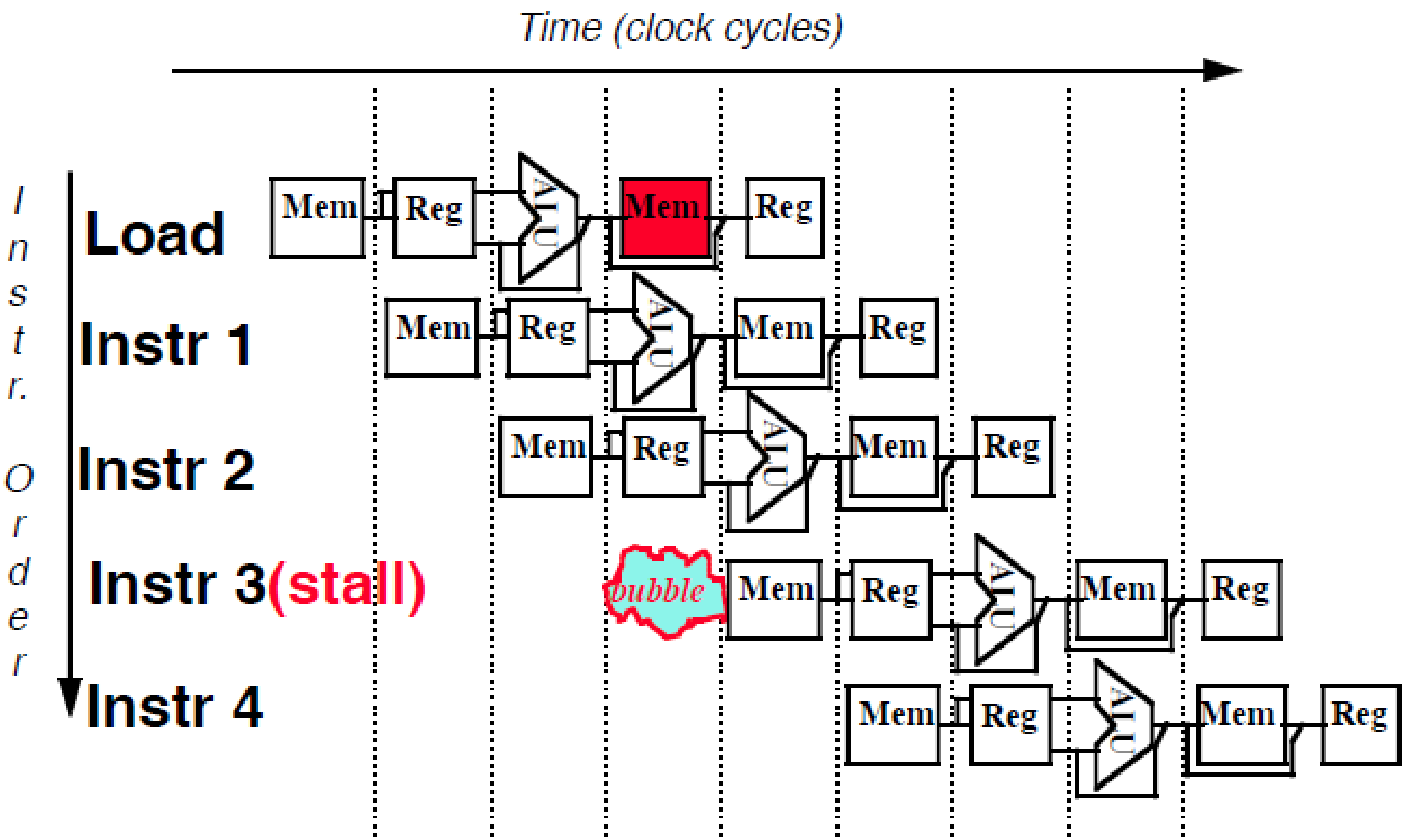
Its not that easy for computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - structural hazards: HW cannot support this combination of instructions
  - data hazards: instruction depends on result of prior instruction still in the pipeline
  - control hazards: pipelining of branches & other instructions that change the PC
- Common solution is to stall the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline

Single Memory is a Structural Hazard

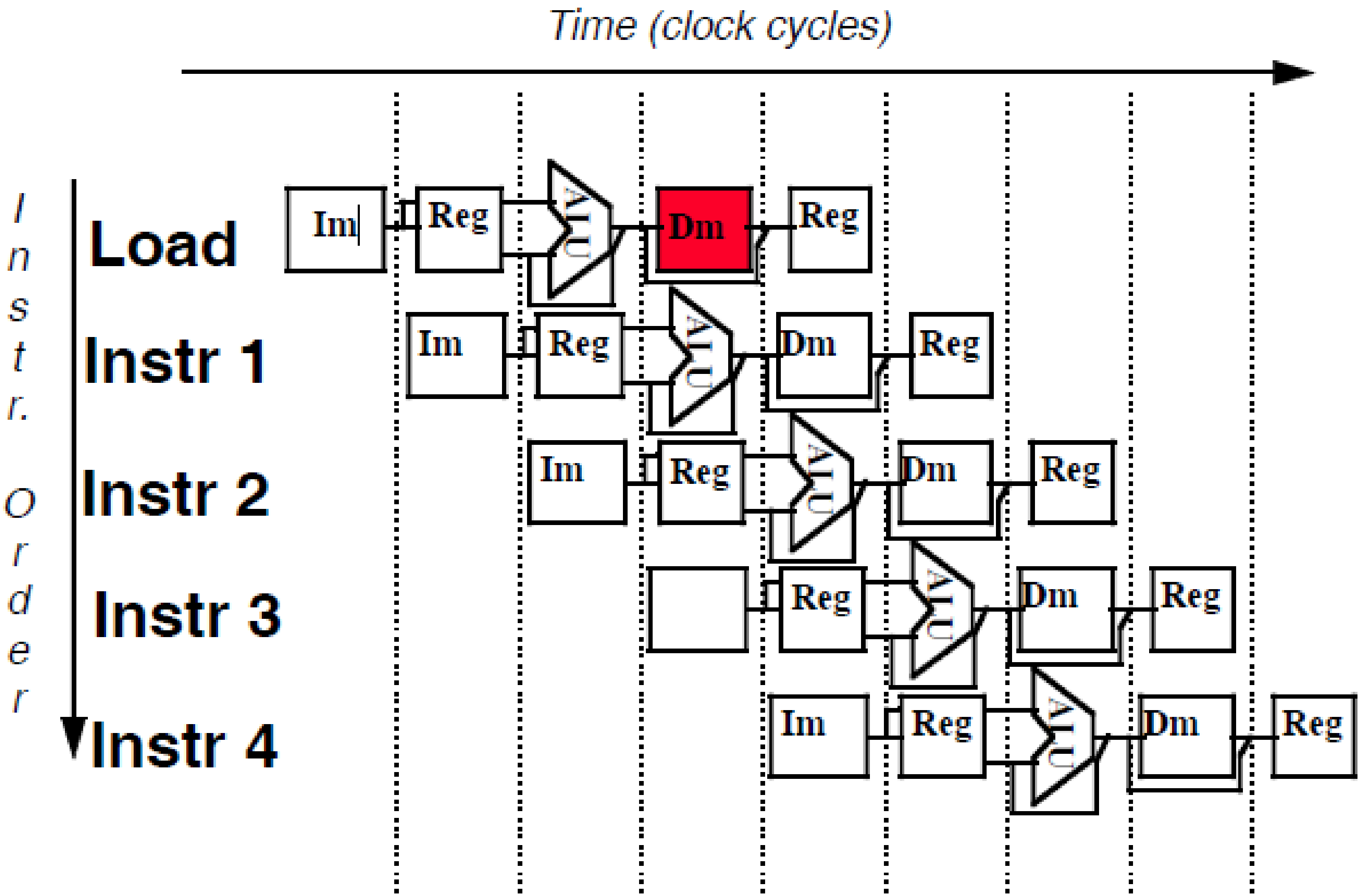


Option 1: Stall to resolve Memory Structural Hazard



Option 2: Duplicate to Resolve Structural Hazard

• Separate Instruction Cache (Im) & Data Cache (Dm)



Data Hazard on r1

add r1 ,r2,r3

sub r4, r1 ,r3

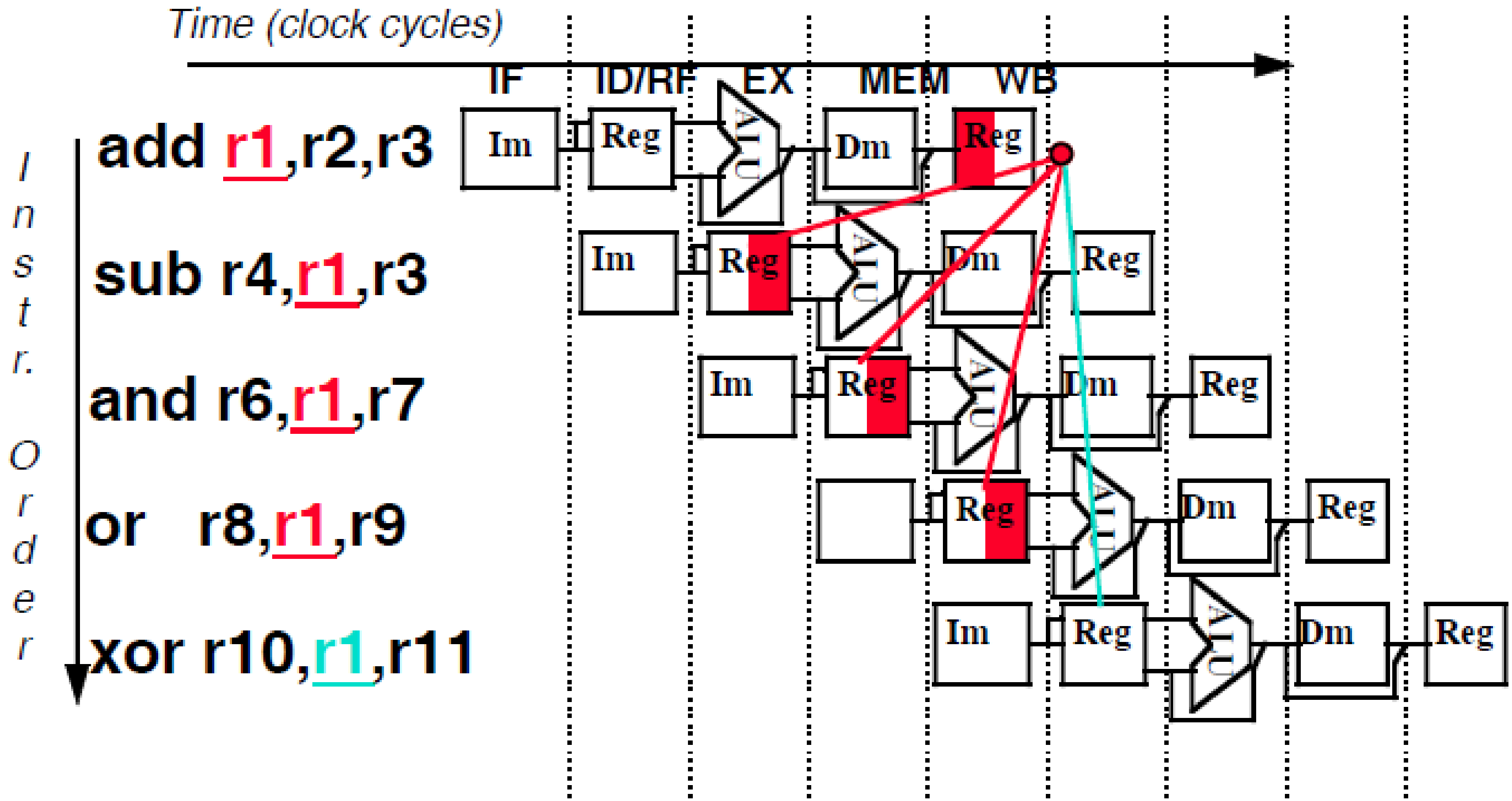
and r6, r1 ,r7

or r8, r1 ,r9

xor r10, r1 ,r11

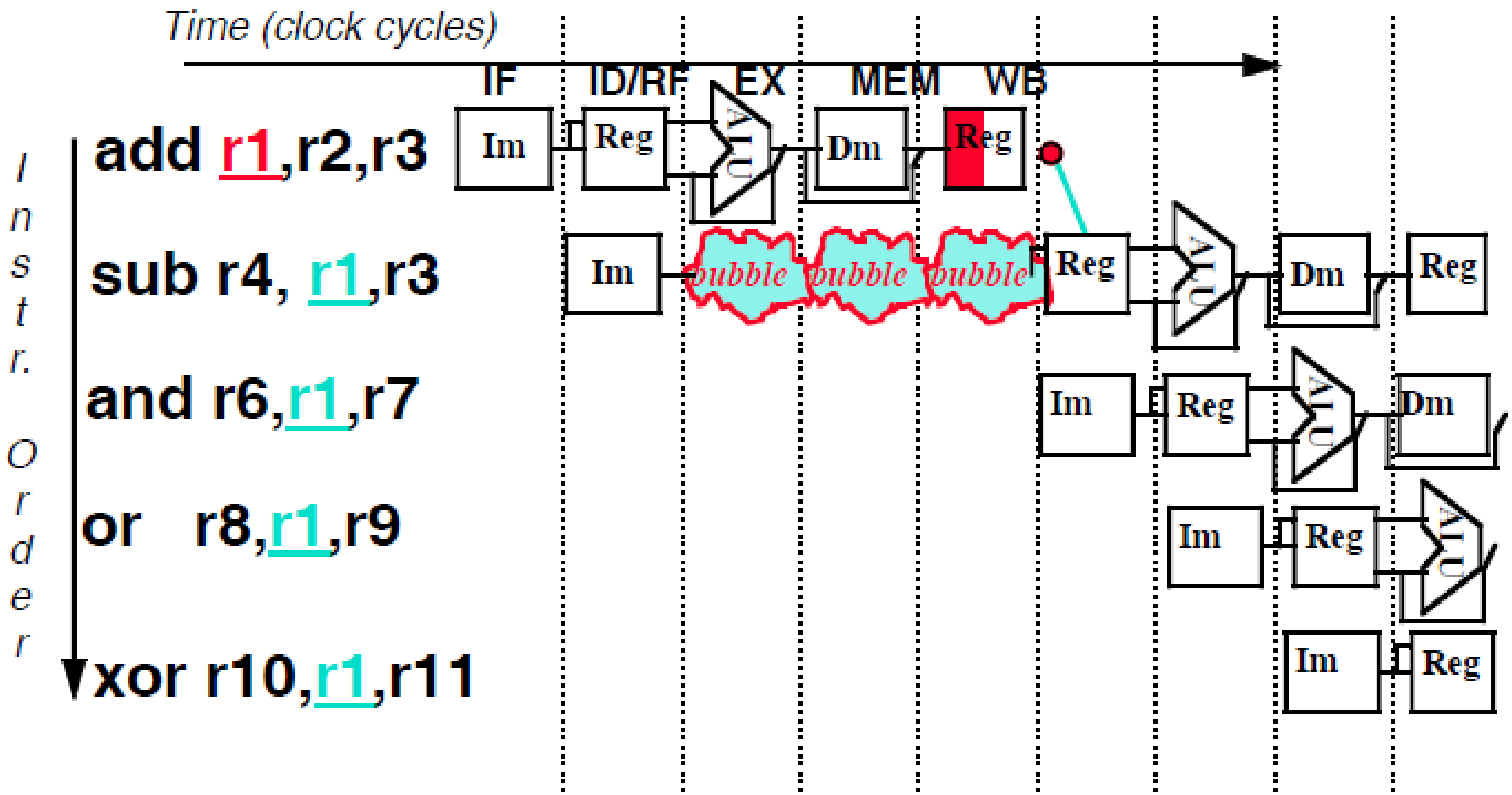
Data Hazard on r1:

- Dependencies backwards in time are hazards



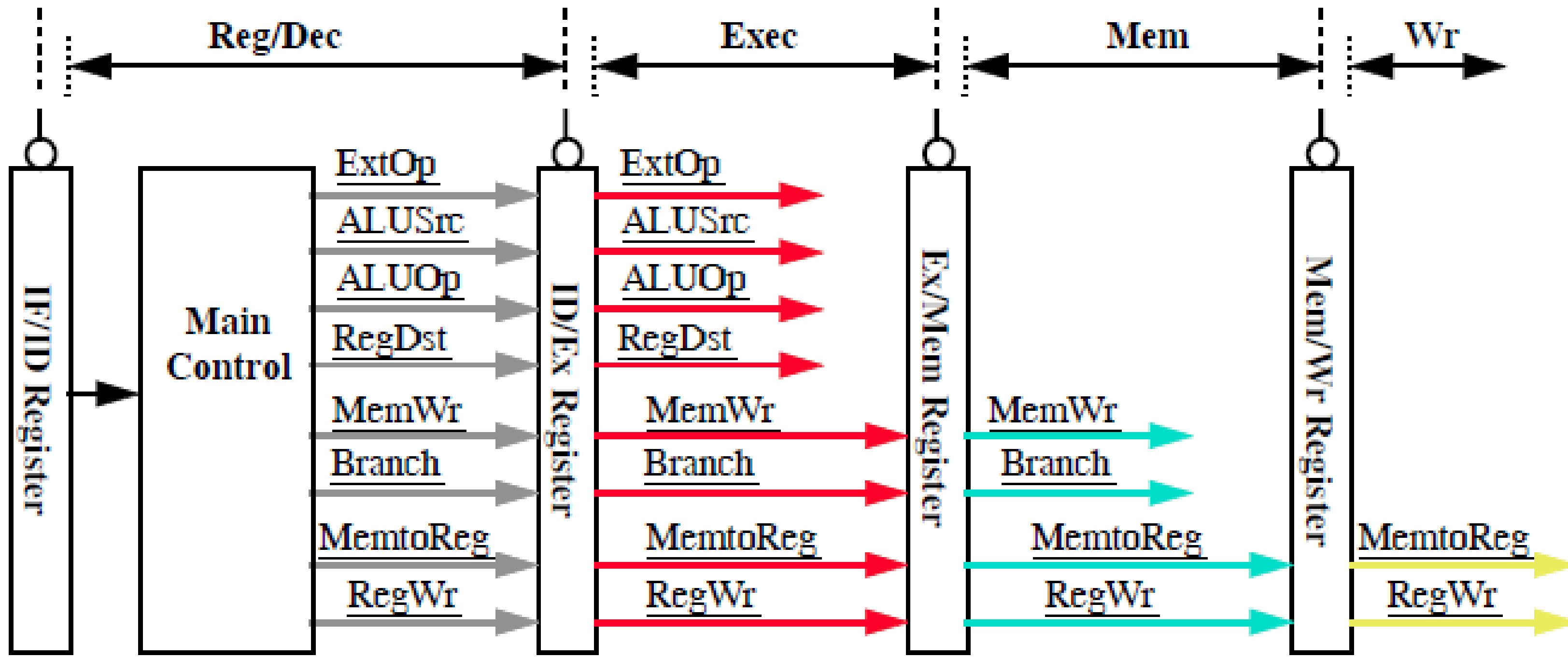
Option1: HW Stalls to Resolve Data Hazard

- **Dependencies backwards in time are hazards**



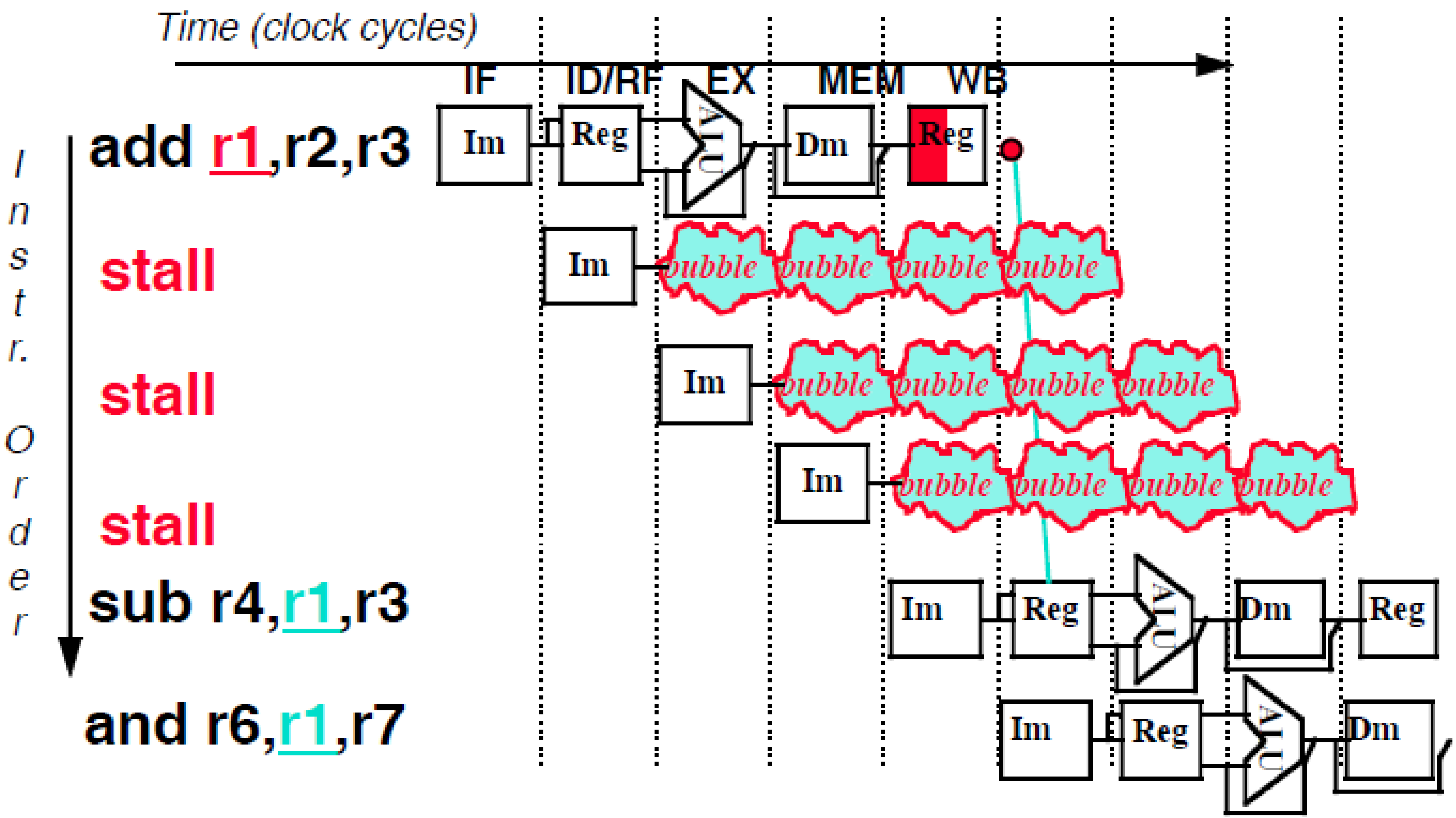
But recall use of "Data Stationary Control"

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



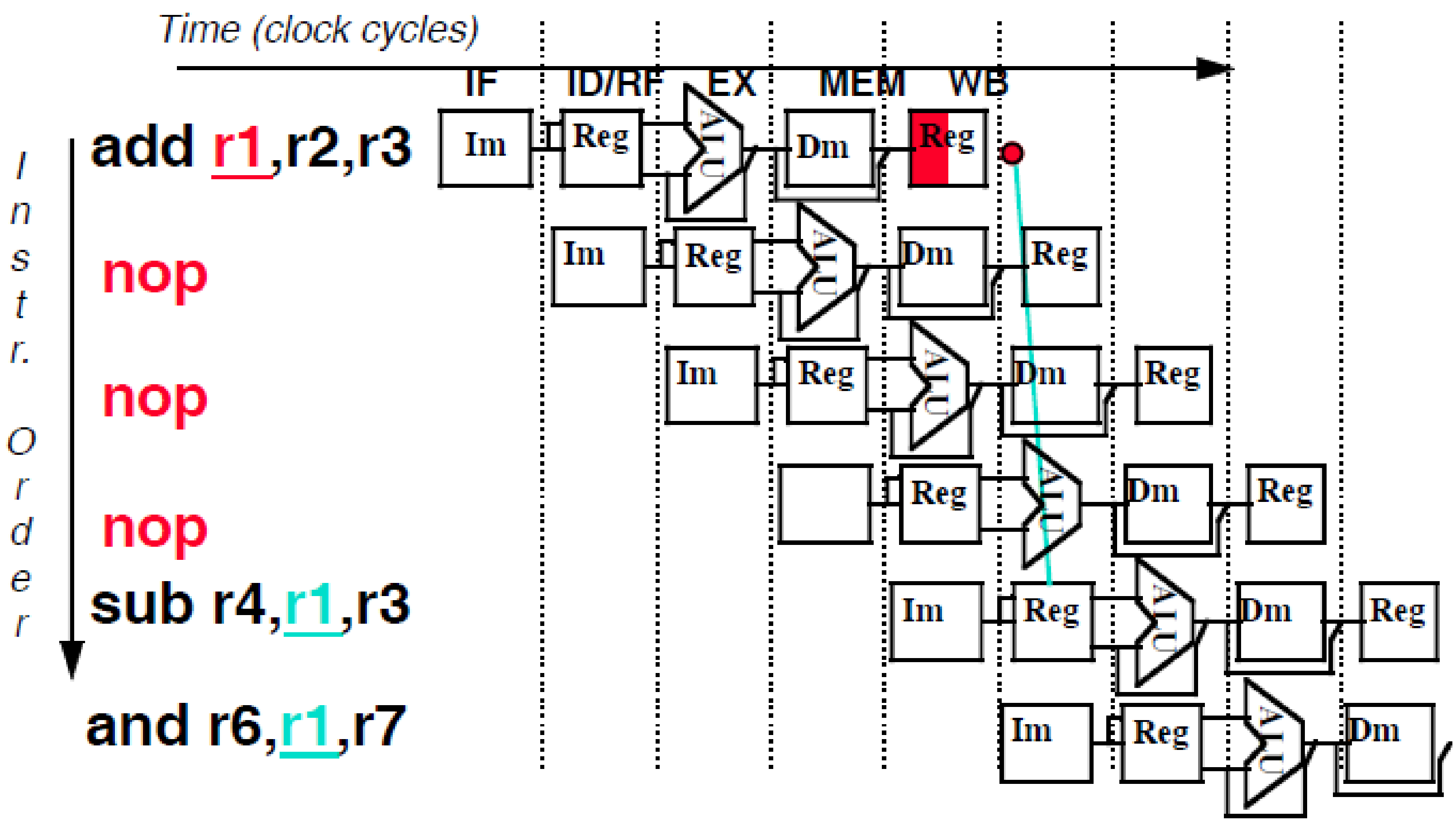
Option 1: How HW really stalls pipeline

- HW doesn't change PC => keeps fetching same instruction & sets control signals to to benign values (0)



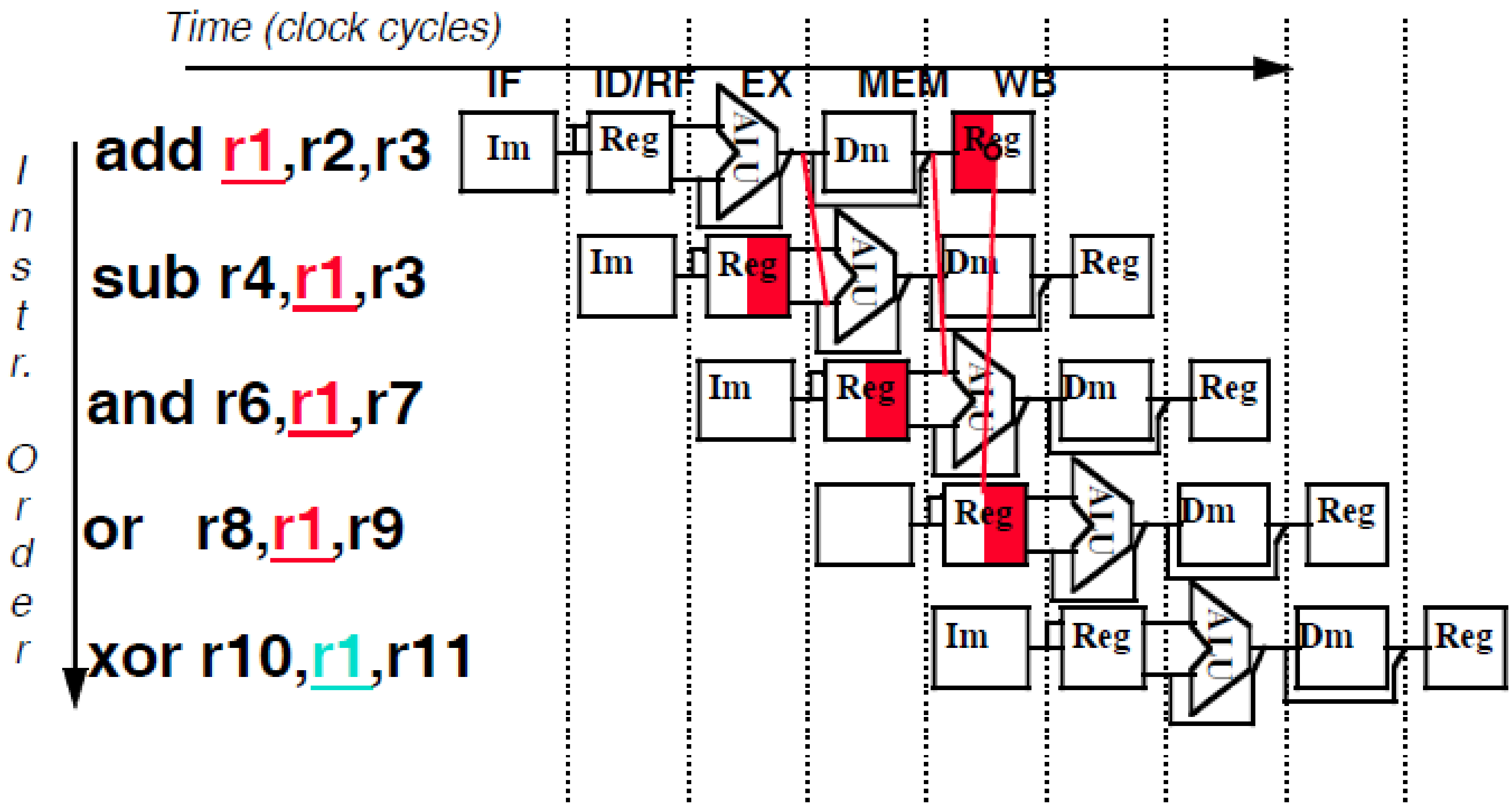
Option 2: SW inserts indepdendent instructions

- Worst case inserts NOP instructions



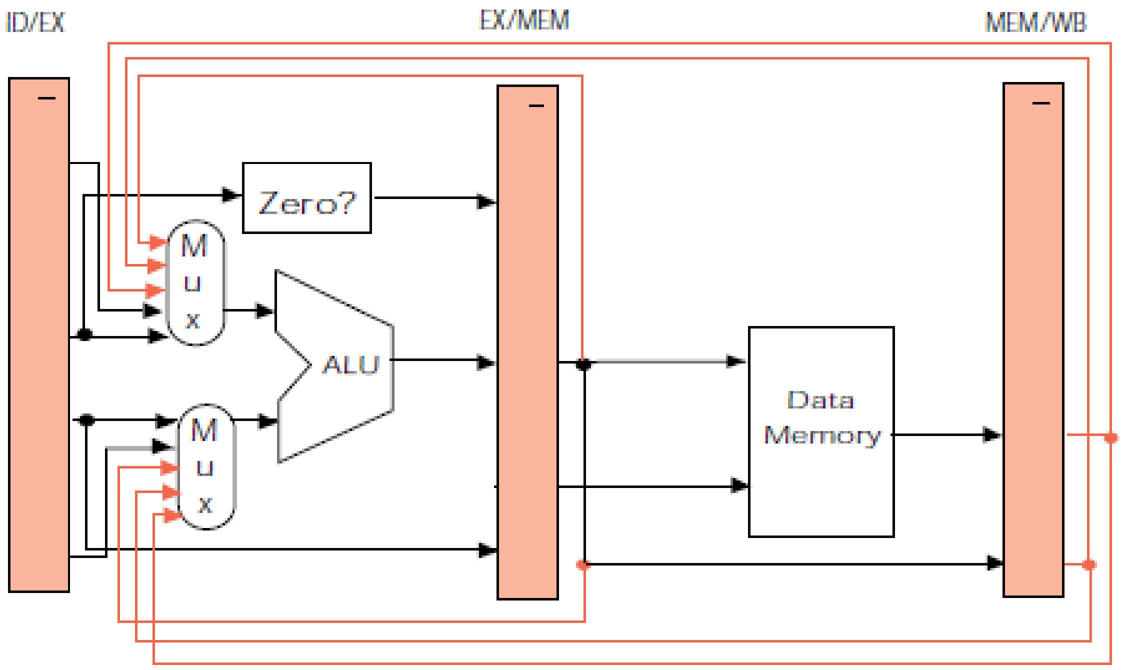
Option 3 Insight: Data is available!

- Pipeline registers already contain needed data

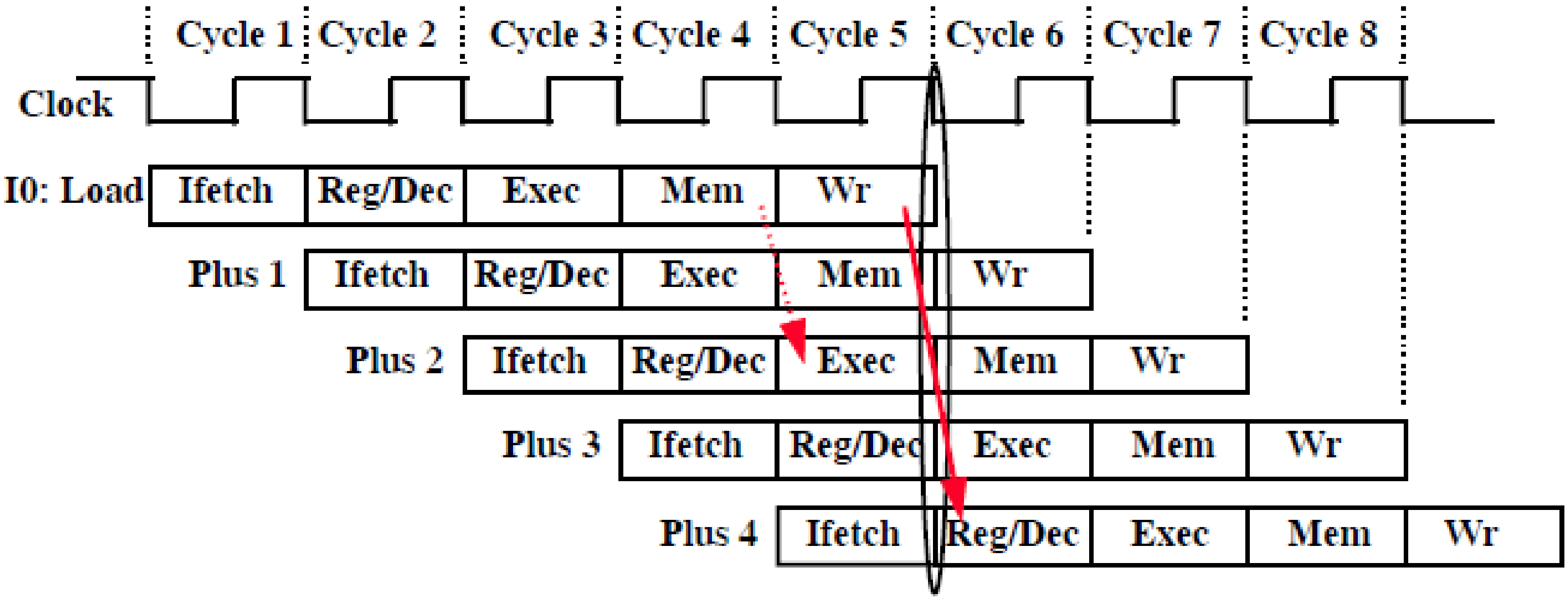


HW Change for “Forwarding” (Bypassing):

- Increase multiplexors to add paths from pipeline registers
- Assumes register read during write gets new value (otherwise more results to be forwarded)

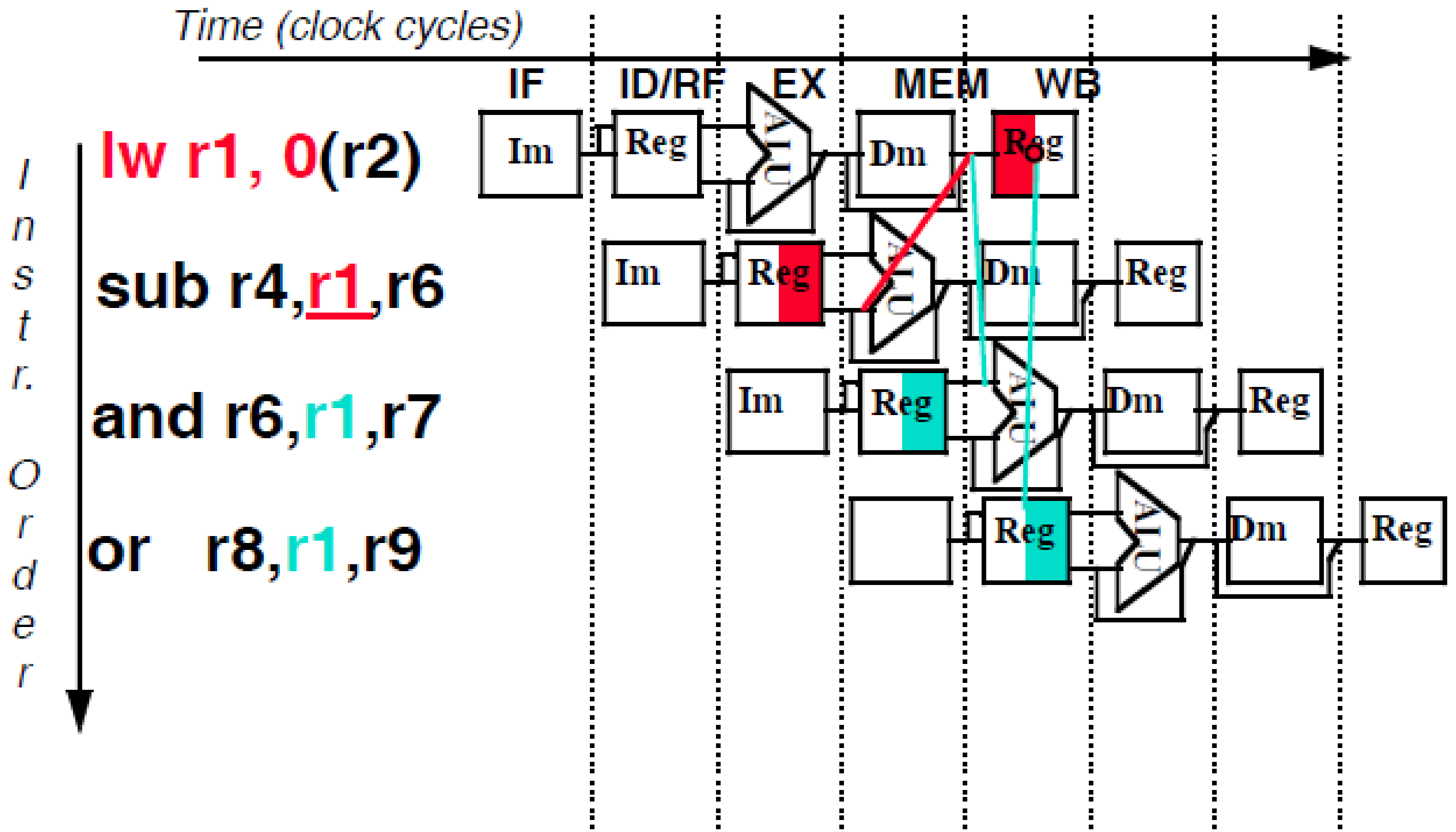


From Last Lecture: The Delay Load Phenomenon



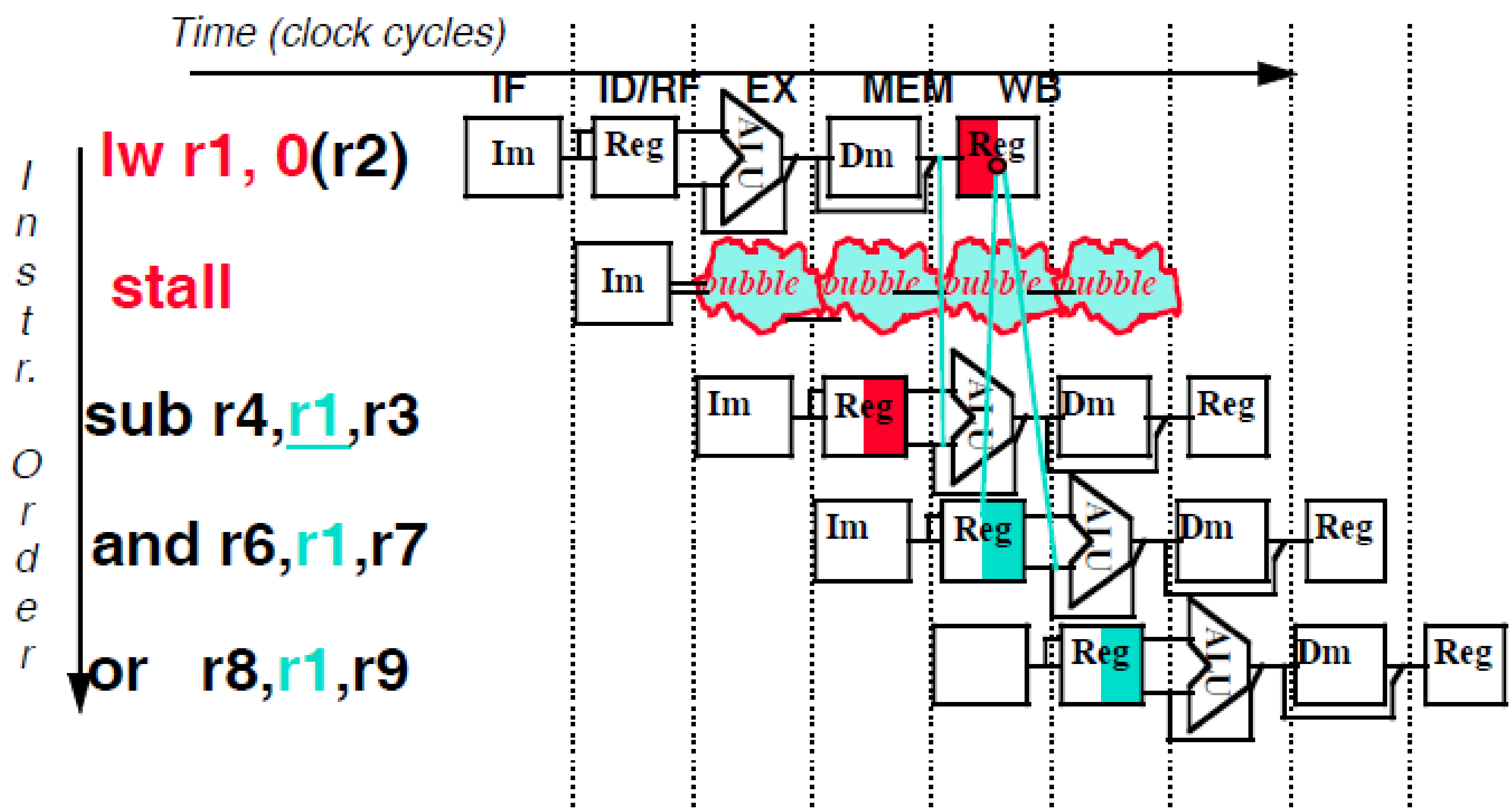
- Although Load is fetched during Cycle 1:
  - The data is NOT written into the Reg File until the end of Cycle 5
  - We cannot read this value from the Reg File until Cycle 6
  - 3-instruction delay before the load take effect

Forwarding reduces Data Hazard to 1 cycle:



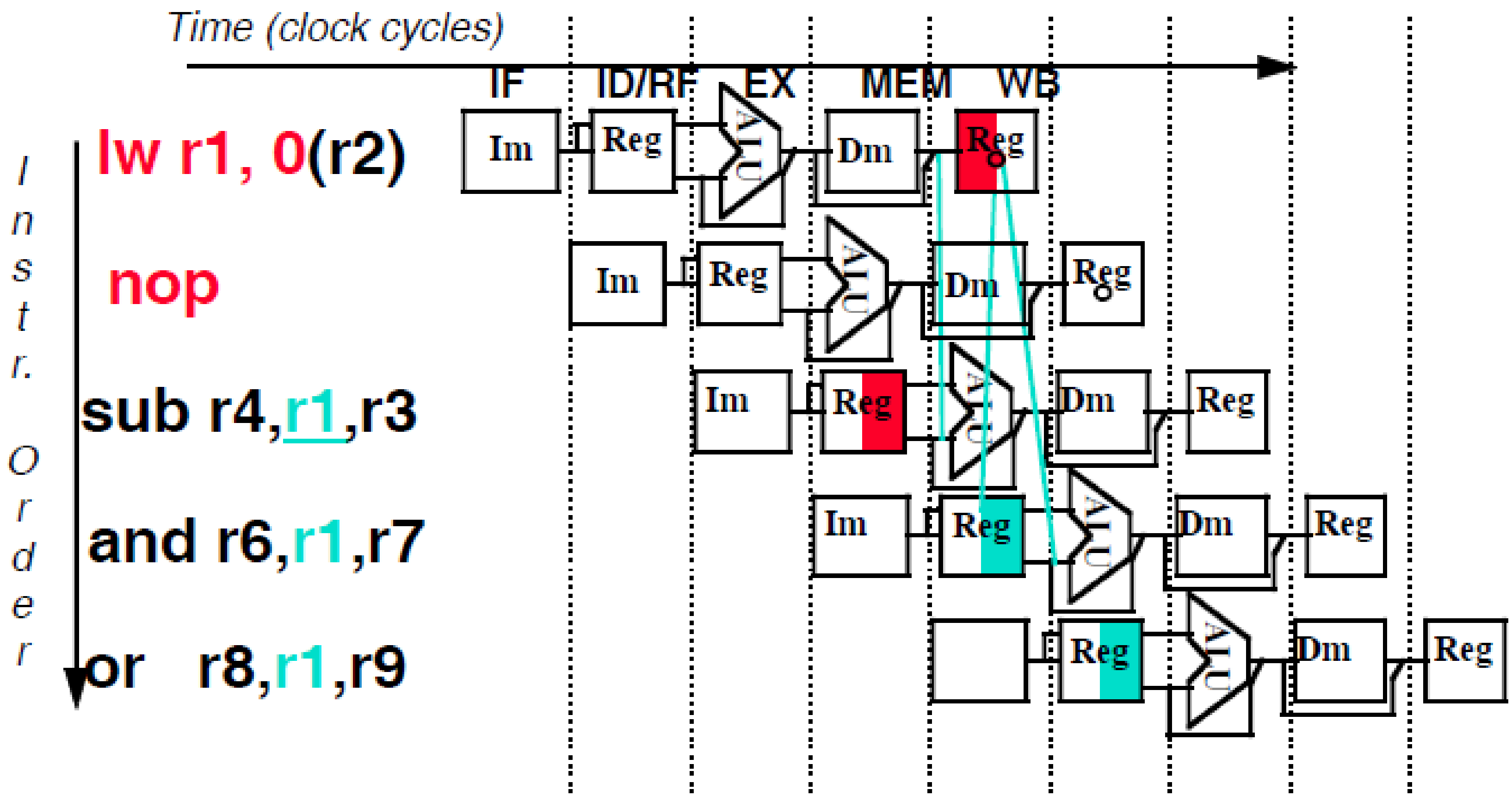
Option1: HW Stalls to Resolve Data Hazard

- "Interlock": checks for hazard & stalls



Option 2: SW inserts independent instructions

- Worst case inserts NOP instructions
- MIPS I solution: No HW checking



Software Scheduling to Avoid Load Hazards

Try producing fast code for

$$a = b + c;$$

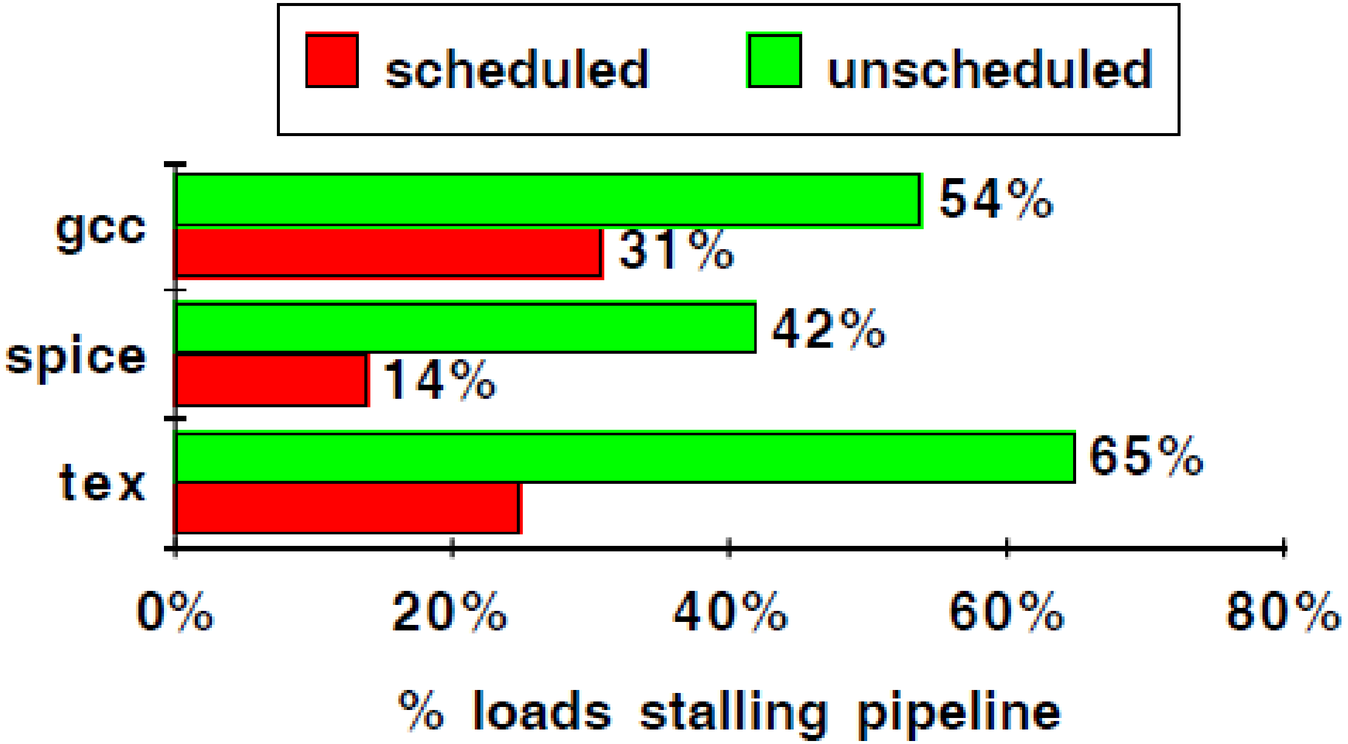
$$d = e - f;$$

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

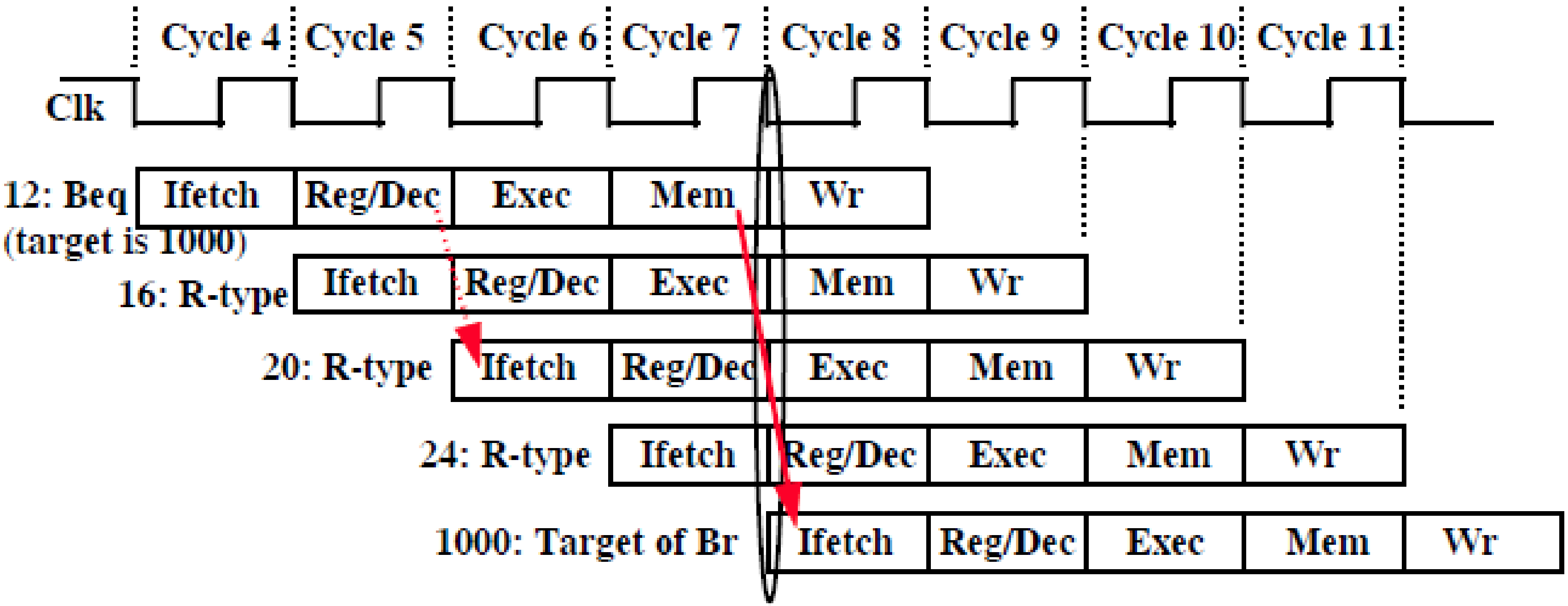
Slow code:

```
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
```

Compiler Avoiding Load Stalls:

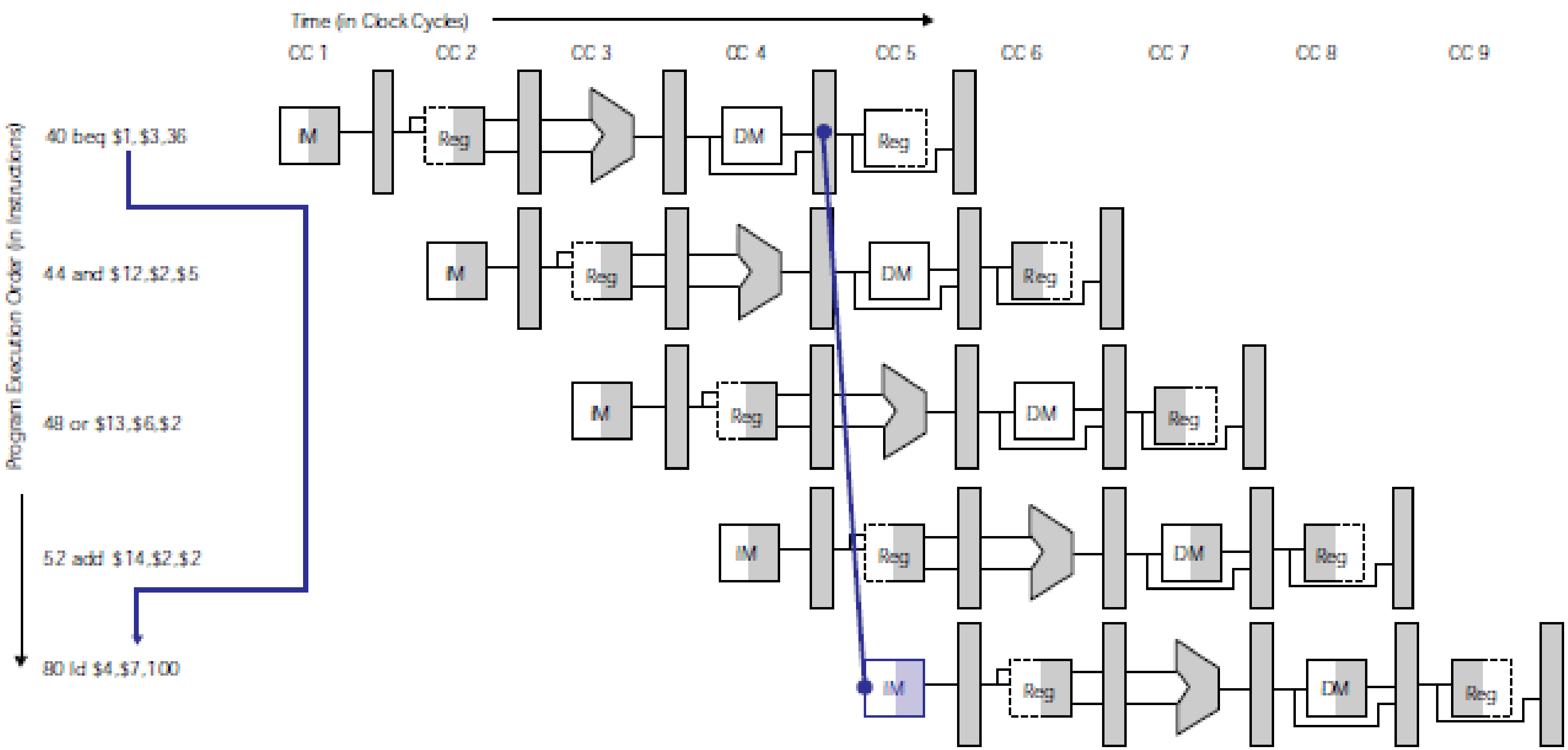


From Last Lecture: The Delay Branch Phenomenon



- Although Beq is fetched during Cycle 4:
  - Target address is NOT written into the PC until the end of Cycle 7
  - Branch's target is NOT fetched until Cycle 8
  - 3-instruction delay before the branch take effect

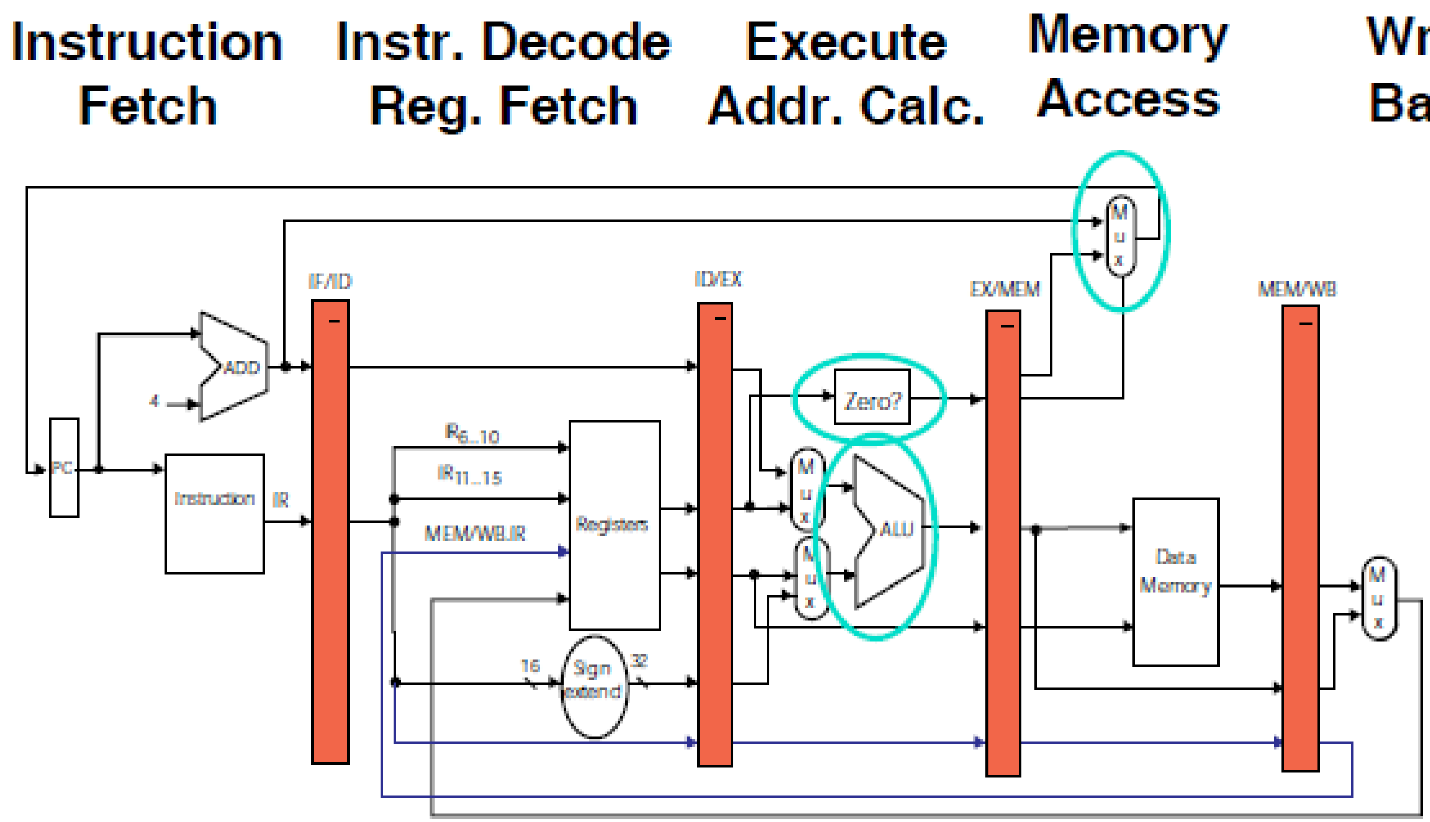
Control Hazard on Branches: 3 stage stall



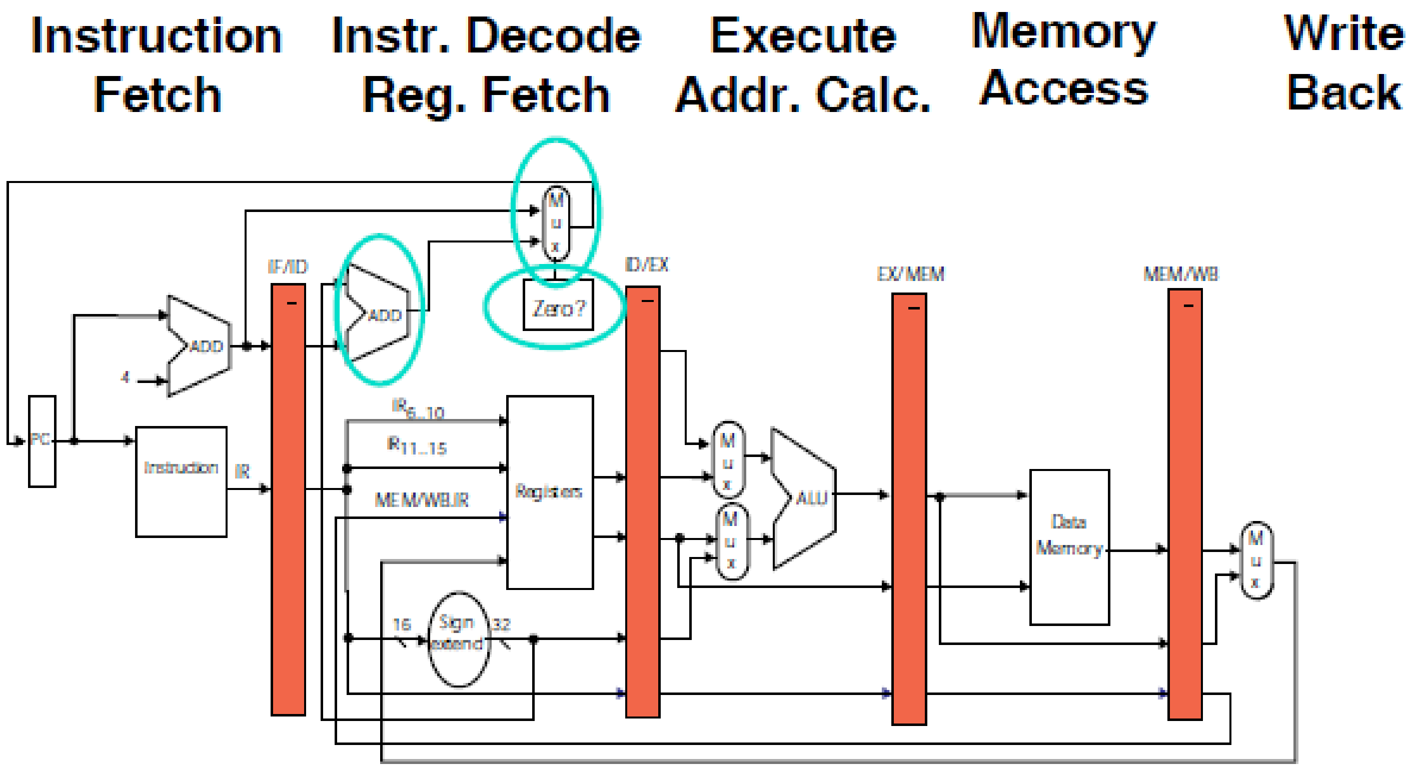
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- 2 part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests = 0 or ≠ 0
- Solution Option 1:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch vs. 3

Option 1: move HW forward to reduce branch delay



Branch Delay now 1 clock cycle



Option 2: Define Branch as Delayed

- Worst case, SW inserts NOP into branch delay
- Where get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch
  - From fall through: only valuable when don't branch
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - about 50% (60% x 80%) of slots usefully filled

When is pipelining hard?

- Interrupts: 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restrart?
  - Who caused the interrupt?

Stage	*Problem interrupts occurring*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation

- Load with data page fault, Add with instruction page fault?
- Solution 1: interrupt vector/instruction□, check last stage
- Solution 2: interrupt ASAP, restart everything incomplete

When is pipelining hard?

- Complex Addressing Modes and Instructions
- Address modes: Autoincrement causes register change during instruction execution
  - Interrupts?
  - Now worry about write hazards since write no longer last stage
    - Write After Read (WAR): Write occurs before independent read
    - Write After Write (WAW): Writes occur in wrong order, leaving wrong result in registers
    - (Previous data hazard called RAW, for Read After Write)
- Memory-memory Move instructions
  - Multiple page faults
  - make progress?

When is pipelining hard?

- Floating Point: long execution time
- Also, may pipeline FP execution unit so that can initiate new instructions without waiting full latency

FP Instruction	*Latency*	*Initiation Rate*	*(MIPS R4000)*
Add, Subtract	4	3	
Multiply	8	4	
Divide	36	35	
Square root	112	111	
Negate	2	1	
Absolute value	2	1	
FP compare	3	2	

- Divide, Square Root take ≈10X to ≈30X longer than Add
  - Exceptions?
  - Adds WAR and WAW hazards since pipelines are no longer same length

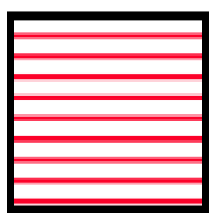
Hazard Detection

Suppose instruction *i* is about to be issued and a predecessor instruction *j* is in the instruction pipeline.

Rregs (*i*) = Registers read by instruction *i*

Wregs (*i*) = Registers written by instruction *i*

- A RAW hazard exists on register ρ if $\exists\, \rho,\ \rho \in$ Rregs(*i*) $\cap$ Wregs(*j*)
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When instruction issues, reserve its result register.
  - When on operation completes, remove its write reservation.



- A WAW hazard exists on register ρ if $\exists\, \rho,\ \rho \in$ Wregs(*i*) $\cap$ Wregs(*j*)

- A WAR hazard exists on register ρ if $\exists\, \rho,\ \rho \in$ Wregs(*i*) $\cap$ Rregs(*j*)

First Generation RISC Pipelines

- All instructions follow same pipeline order ("static schedule").
- Register write in last stage
  - Avoid WAW hazards
- All register reads performed in first stage after issue.
  - Avoid WAR hazards
- Memory access in stage 4
  - Avoid all memory hazards
- Control hazards resolved by delayed branch (with fast path)
- RAW hazards resolved by bypass, except on load results

which are resolved by fiat (delayed load).

Substantial pipelining with very little cost or complexity.

Machine organization is (slightly) exposed!

Relies very heavily on "hit assumption"of memory accesses in cache

Review: Summary of Pipelining Basics

- Speed Up $\leq$ Pipeline Depth; if ideal CPI is 1, then:

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

- Hazards limit performance on computers:
  - structural: need more HW resources
  - data: need forwarding, compiler scheduling
  - control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards since pipelining helps instruction bandwidth, not latency
- Compilers key to reducing cost of data and control hazards
  - load delay slots
  - branch delay slots
- Exceptions, Instruction Set, FP makes pipelining harder
- Longer pipelines => Branch prediction, more instruction parallelism?