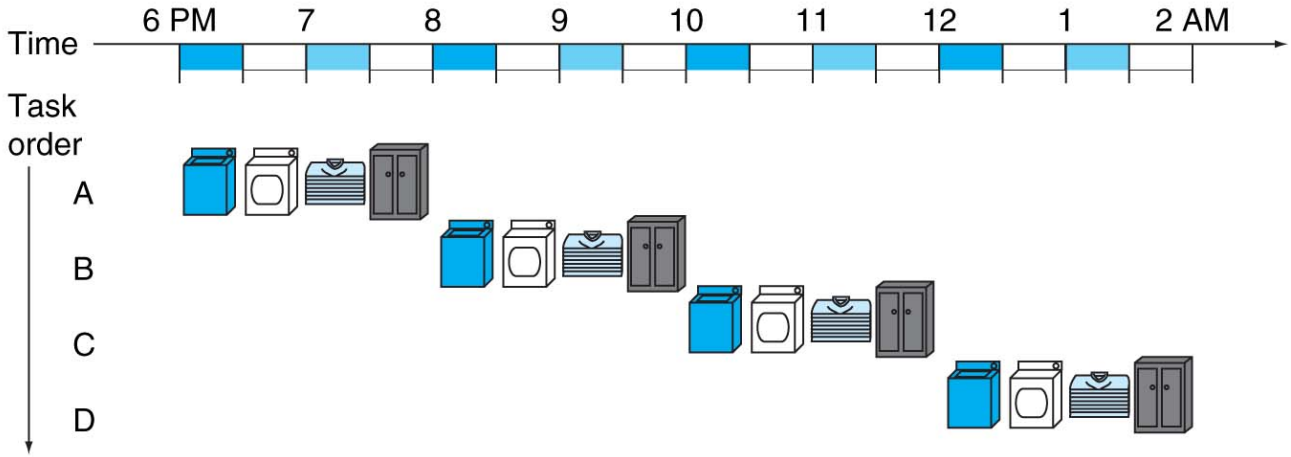
Intro to Architecture of Computers – III Pipelining

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

Today

- Analogies in real world
- Pipelining in modern processors — MIPS
- Pipelining in Y86
- Hazards

Clothes-washing analogy



Time per load:– 2 hours
Time for 4 loads:– 8 hours!
0.5 Loads/hour

Time per load:– 2 hours (still)
Time for 4 loads:– 3.5 hours!
2 loads per hour sustained rate!

Real-world pipelines: car washes

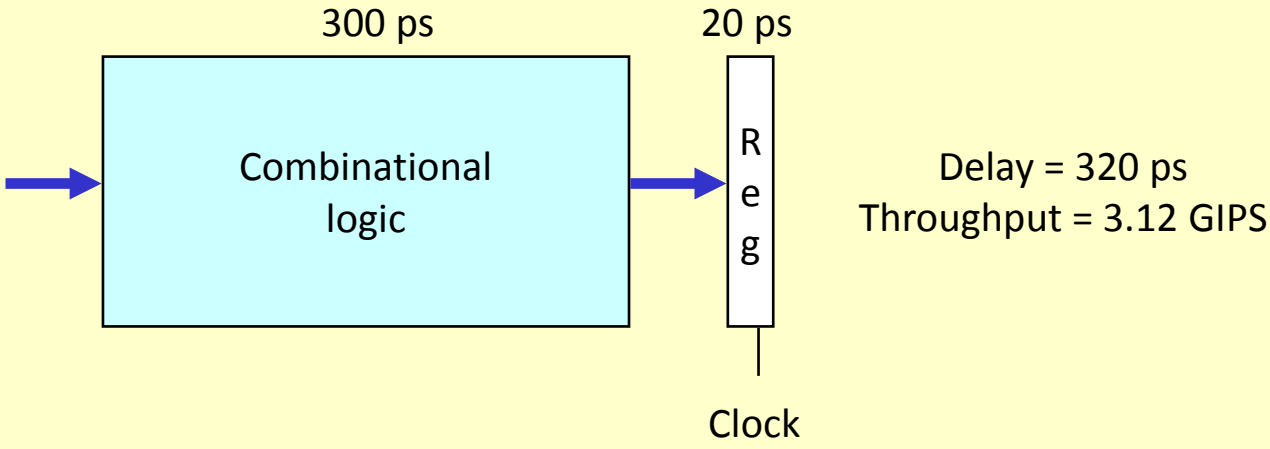
Sequential

Parallel

Pipelined

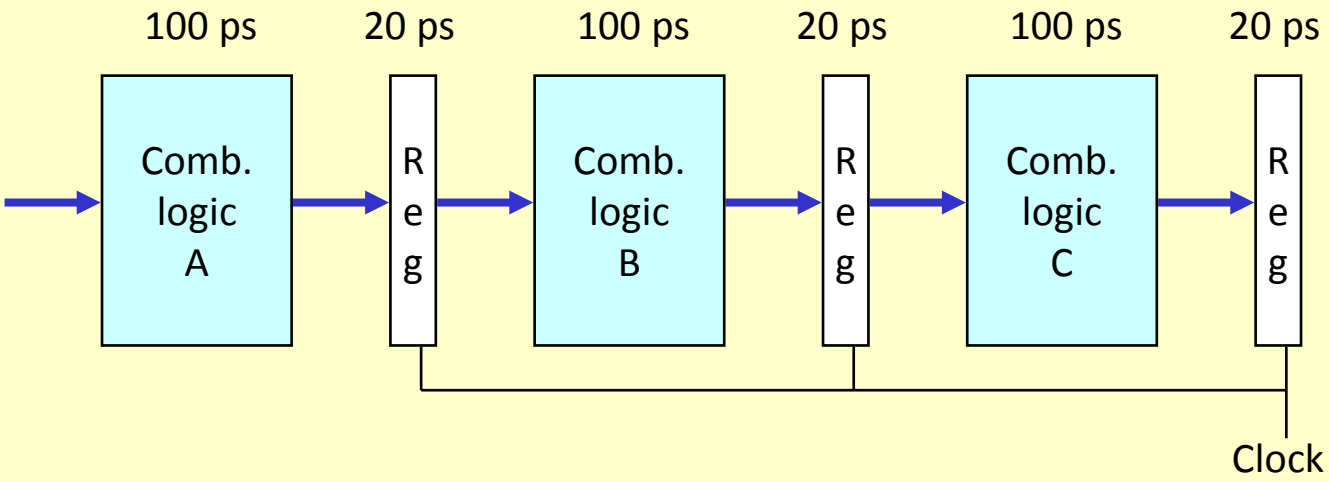
- **Idea**
  - Divide process into independent stages
  - Move objects through stages in sequence
  - At any given times, multiple objects being processed

Computational example



- **System**
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps

3-way pipelined version

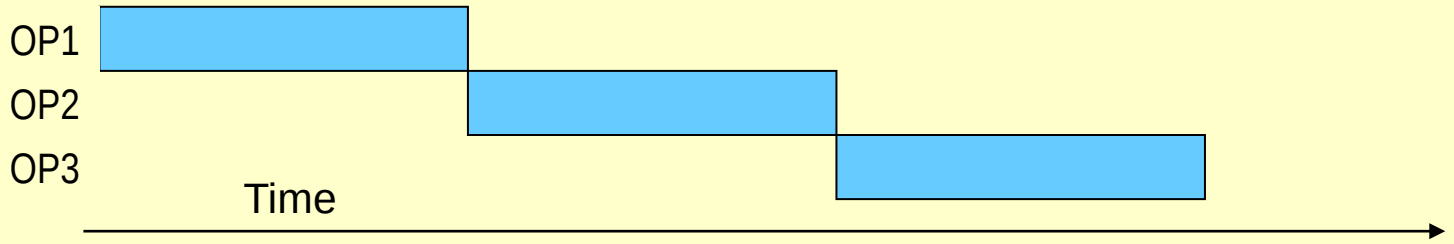


Delay = 360 ps
Throughput = 8.33 GIPS

- **System**
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

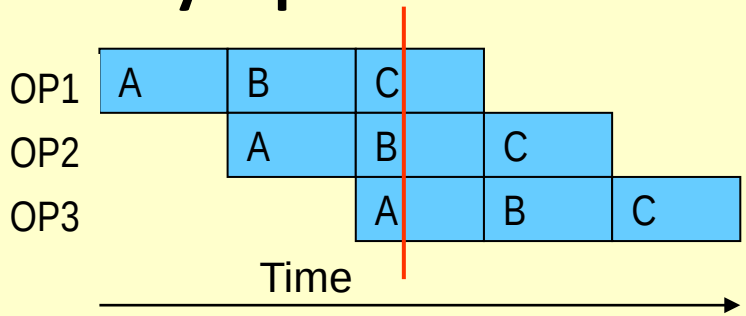
Pipeline diagrams

- **Unpipelined**



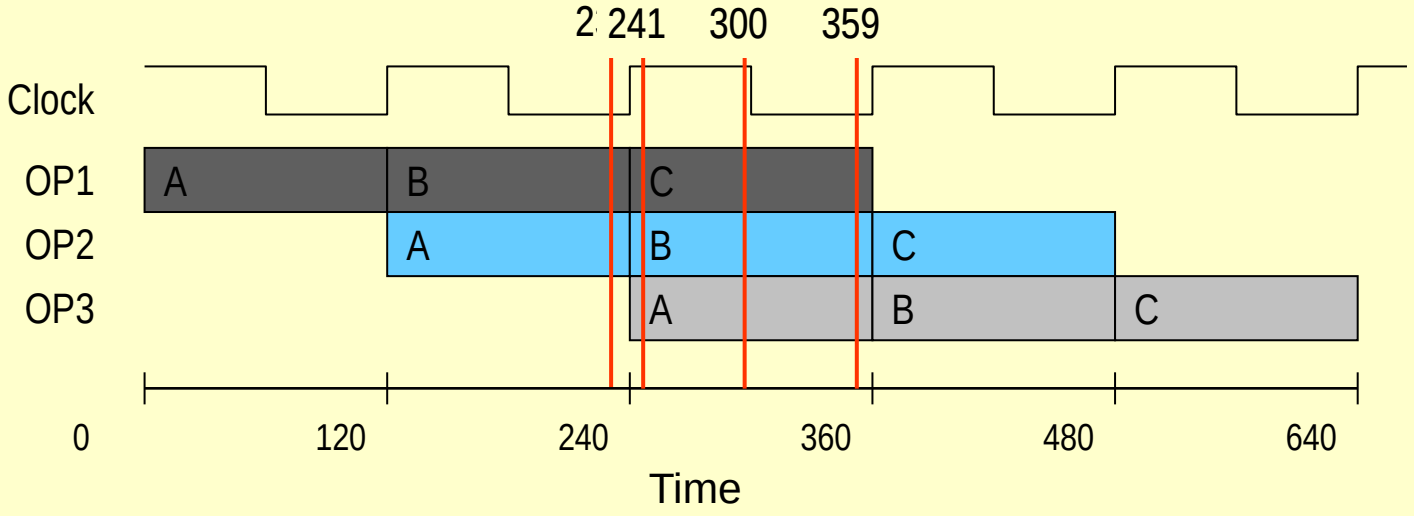
  - Cannot start new operation until previous one completes

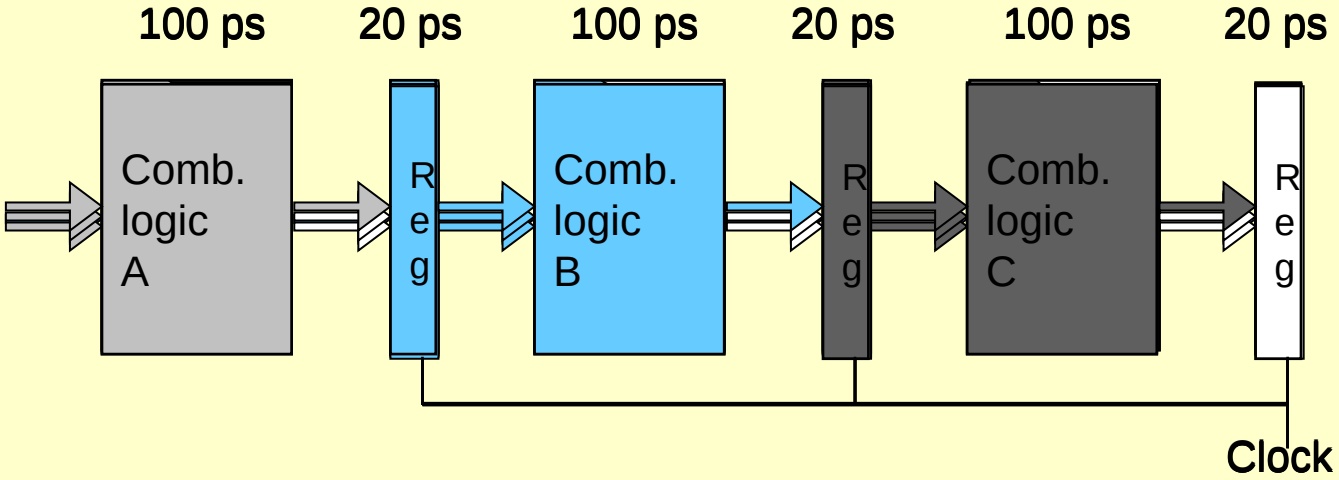
- **3-Way Pipelined**



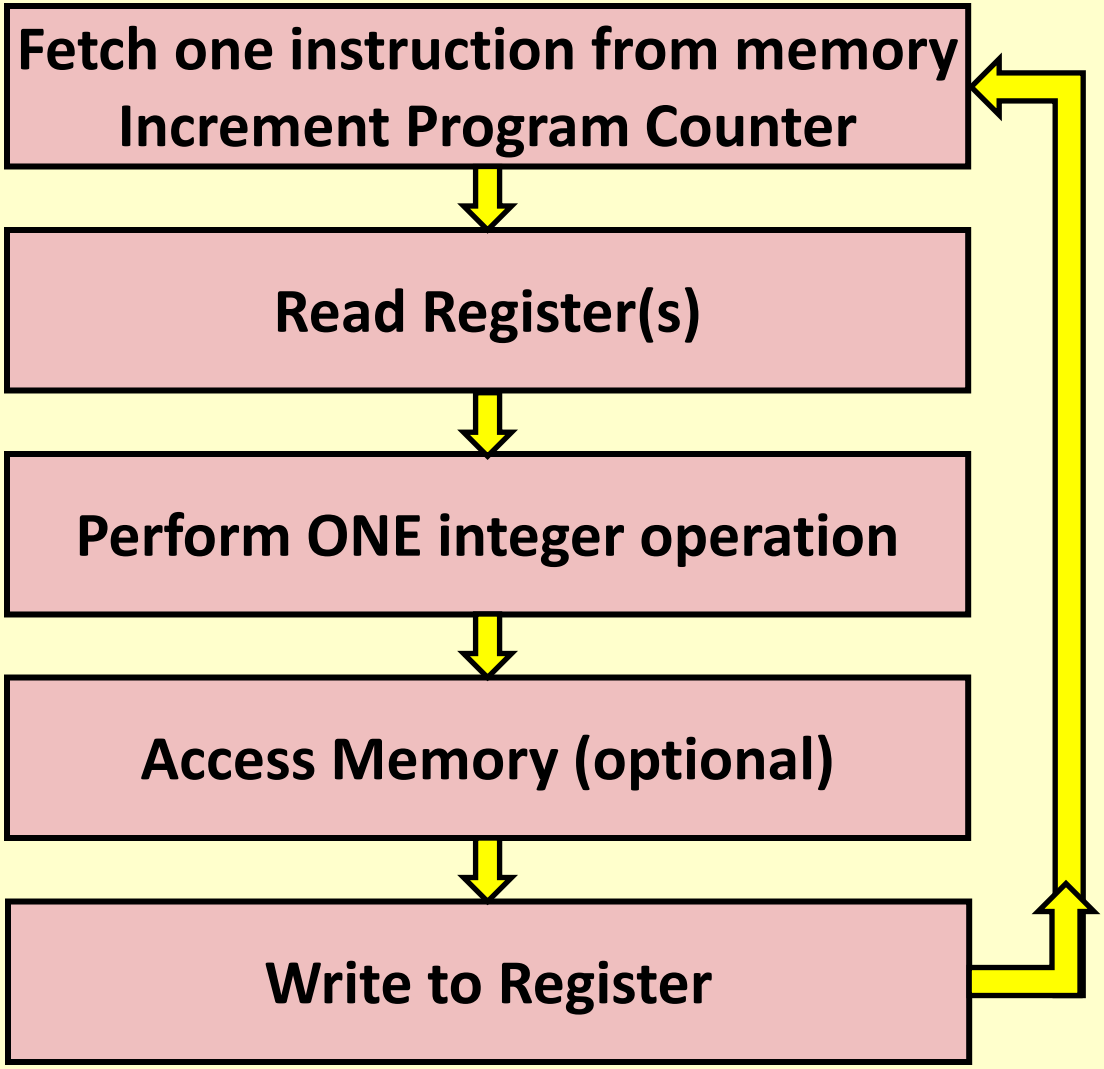
  - Up to 3 operations in process simultaneously

Operating a pipeline



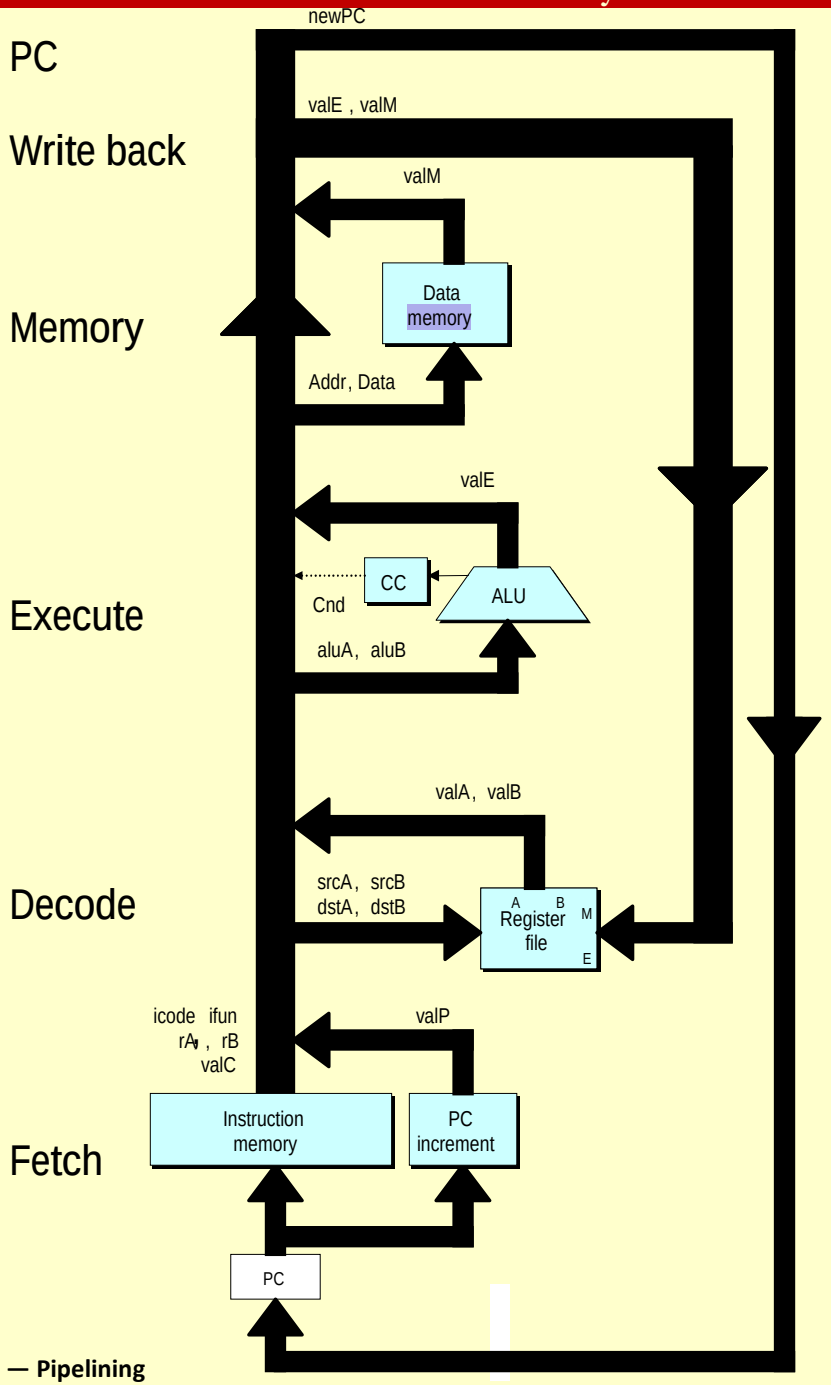


Execution model for modern computers

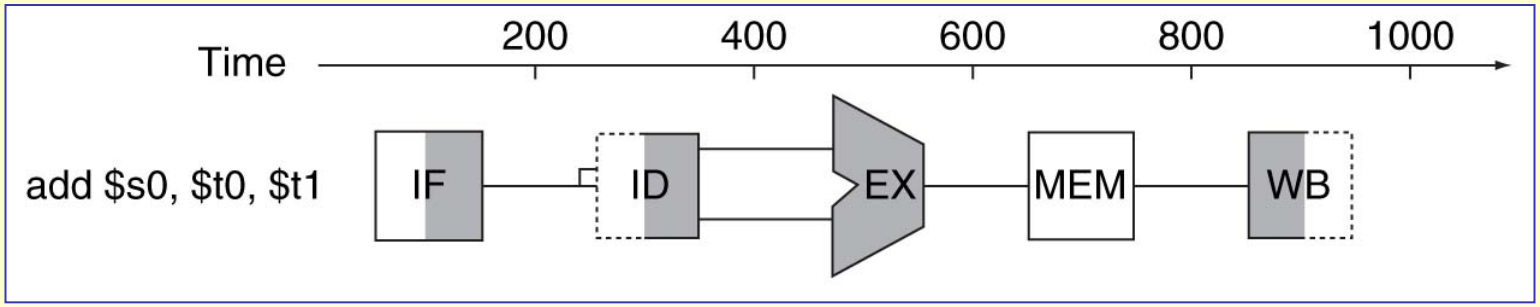


Sequential stages

- **Fetch**
  - Read instruction from instruction memory
  - Increment program counter (**`%eip`**)
- **Decode**
  - Read registers named in instruction
- **Execute**
  - Compute value or address
- **Memory**
  - Read or write data
- **Write Back**
  - Write program registers
- **PC**
  - Update program counter (incremented or jump/call/ret target)



Stylized processor execution stages



Instruction fetch

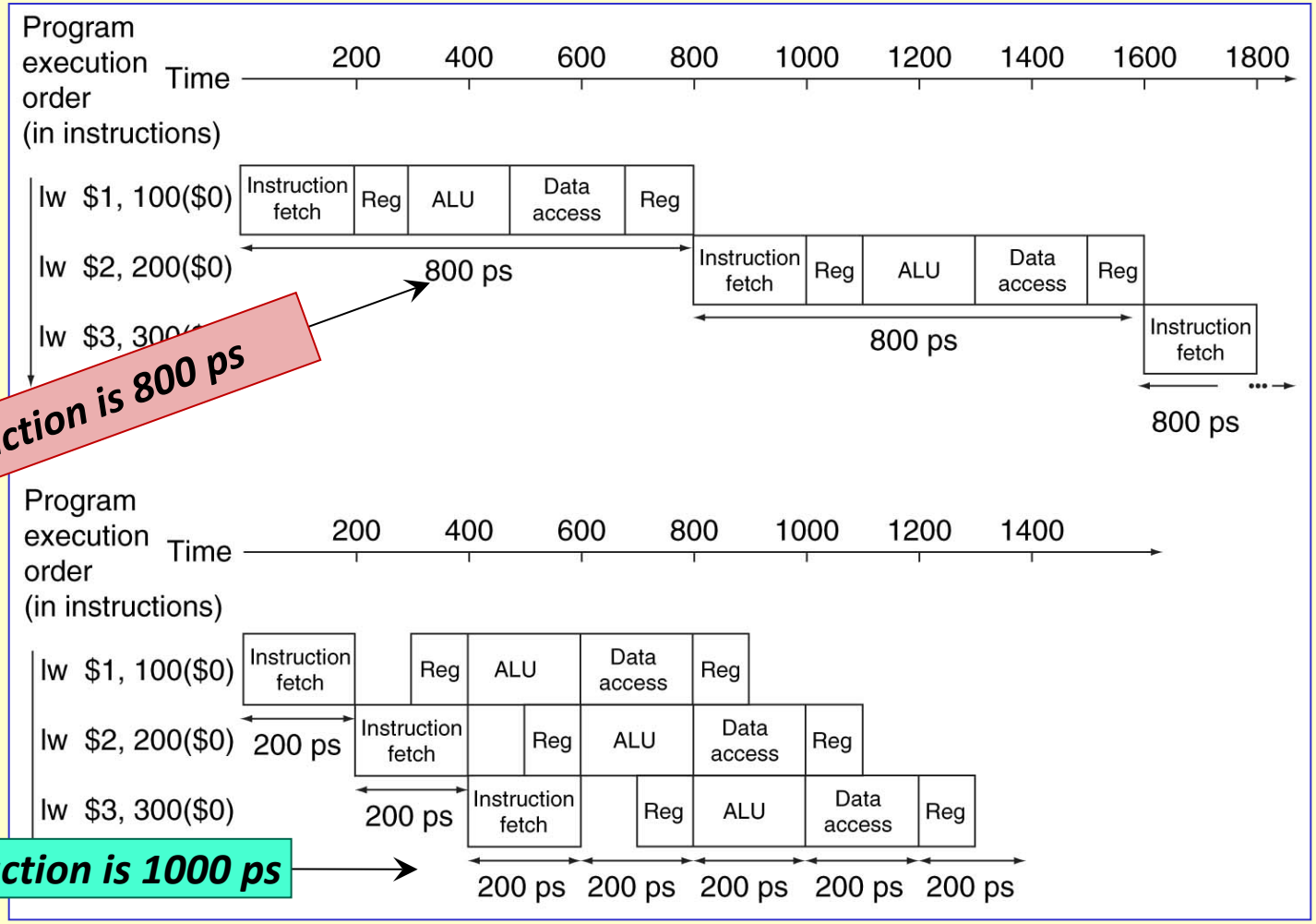
Instruction decode
Register read

Execute arithmetic op.

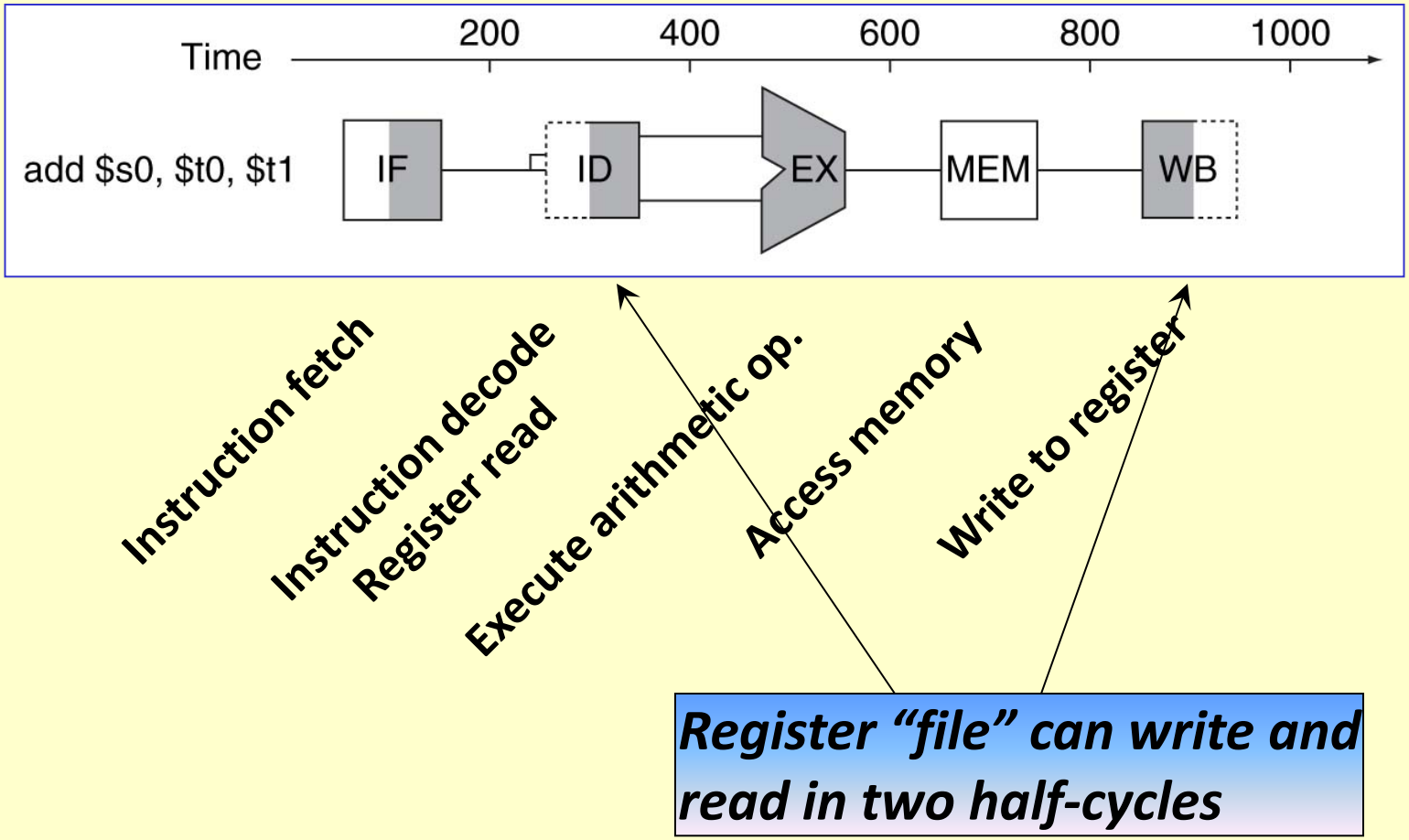
Access memory

Write to register

Stylized processor — not pipelined *vs.* pipelined



Stylized processor pipeline

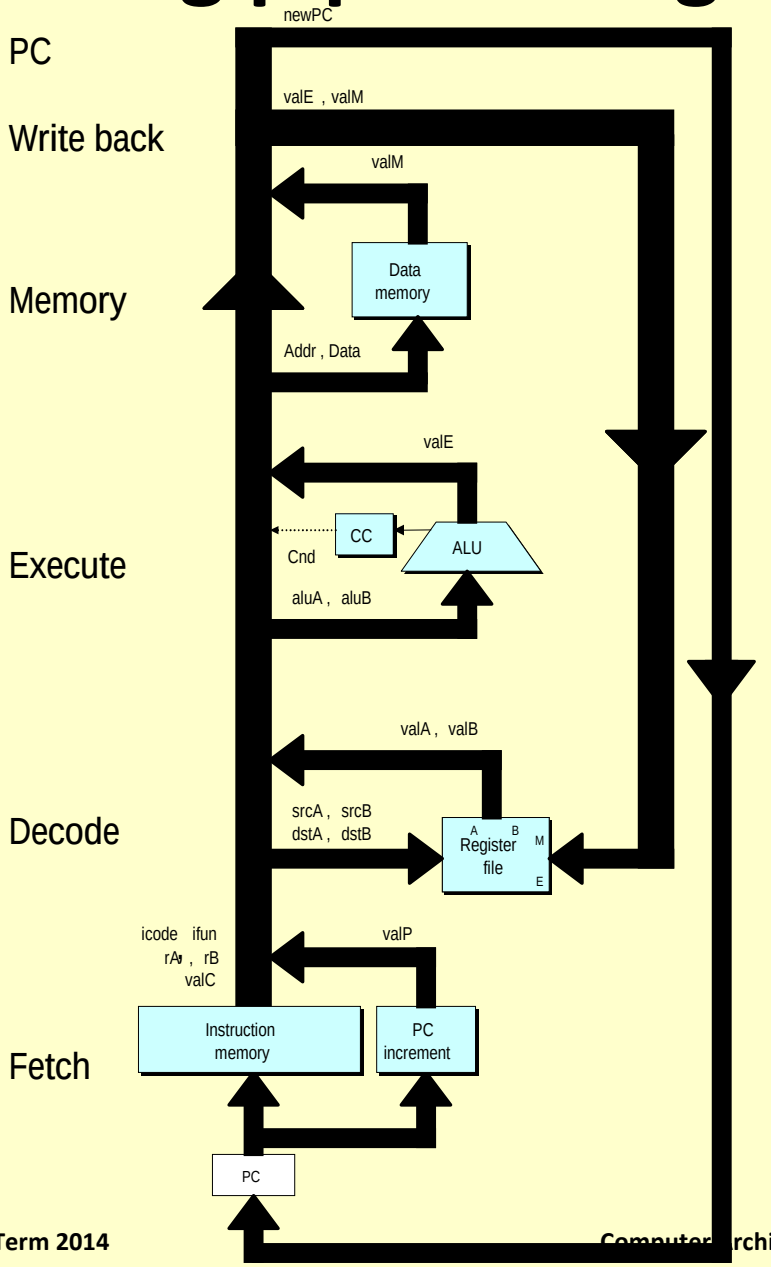


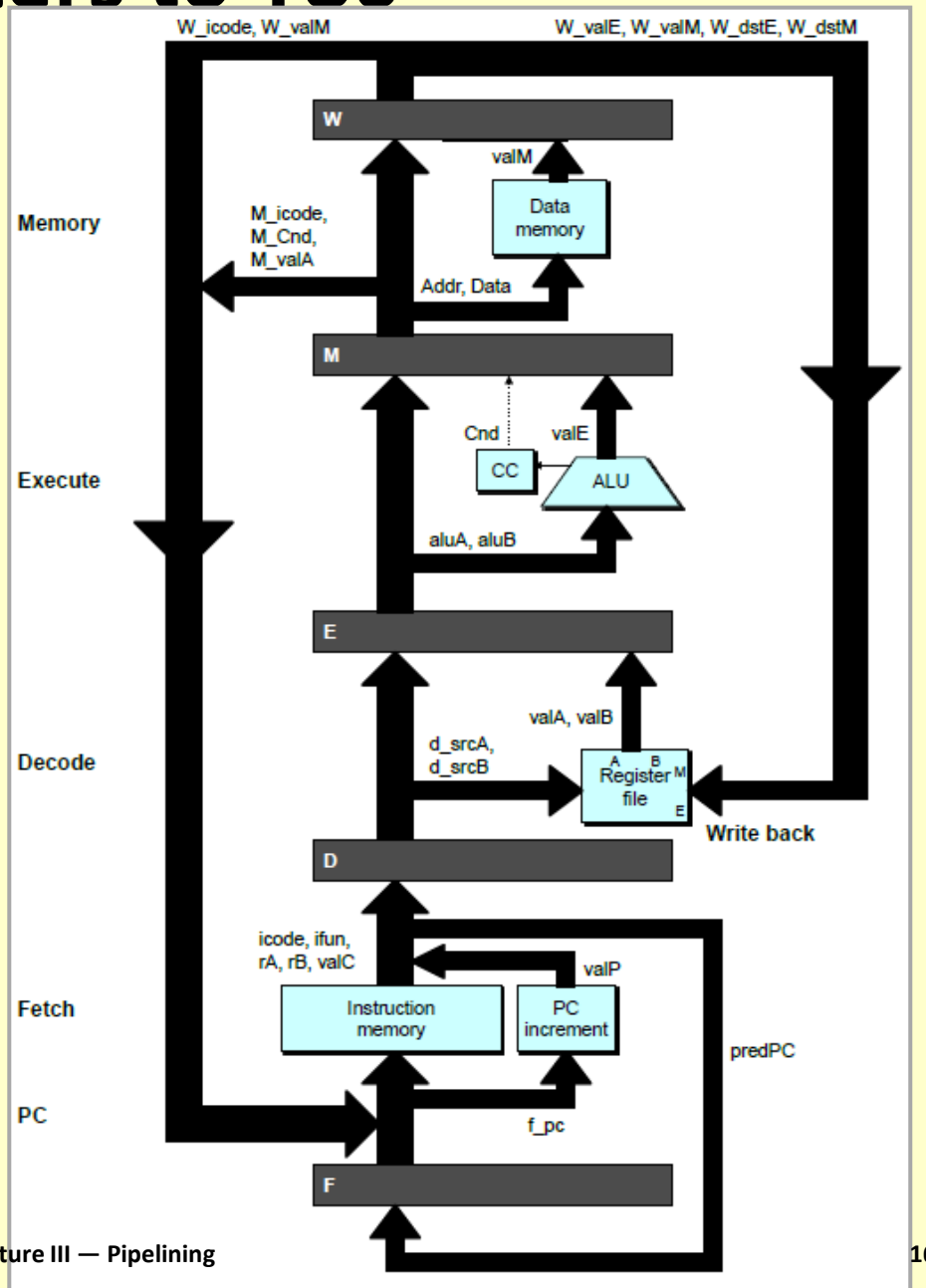
Questions?

Today

- Analogies in real world
- Pipelining in modern processors — MIPS
- **Pipelining in Y86**
- **Hazards**

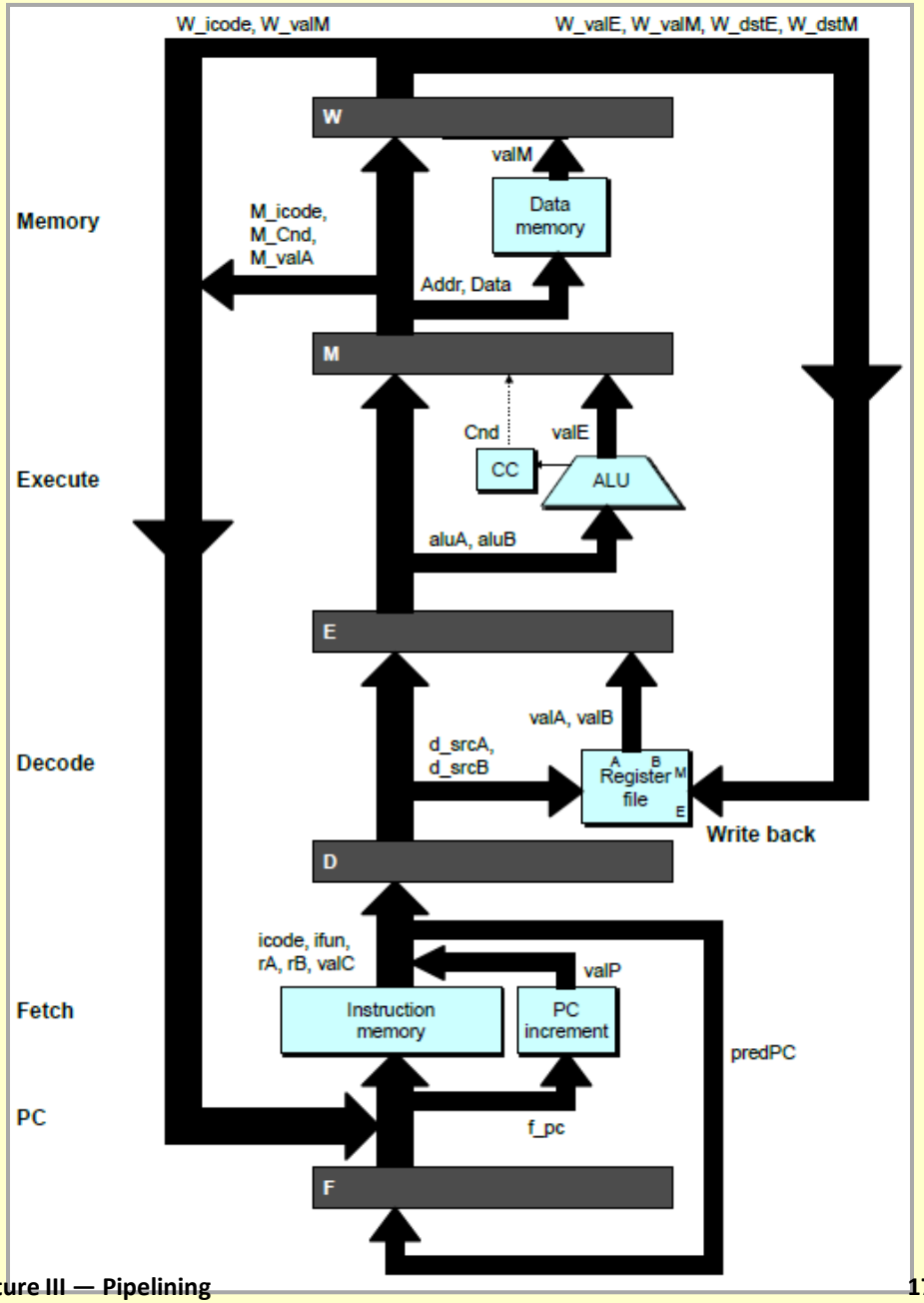
Adding pipeline registers to Y86



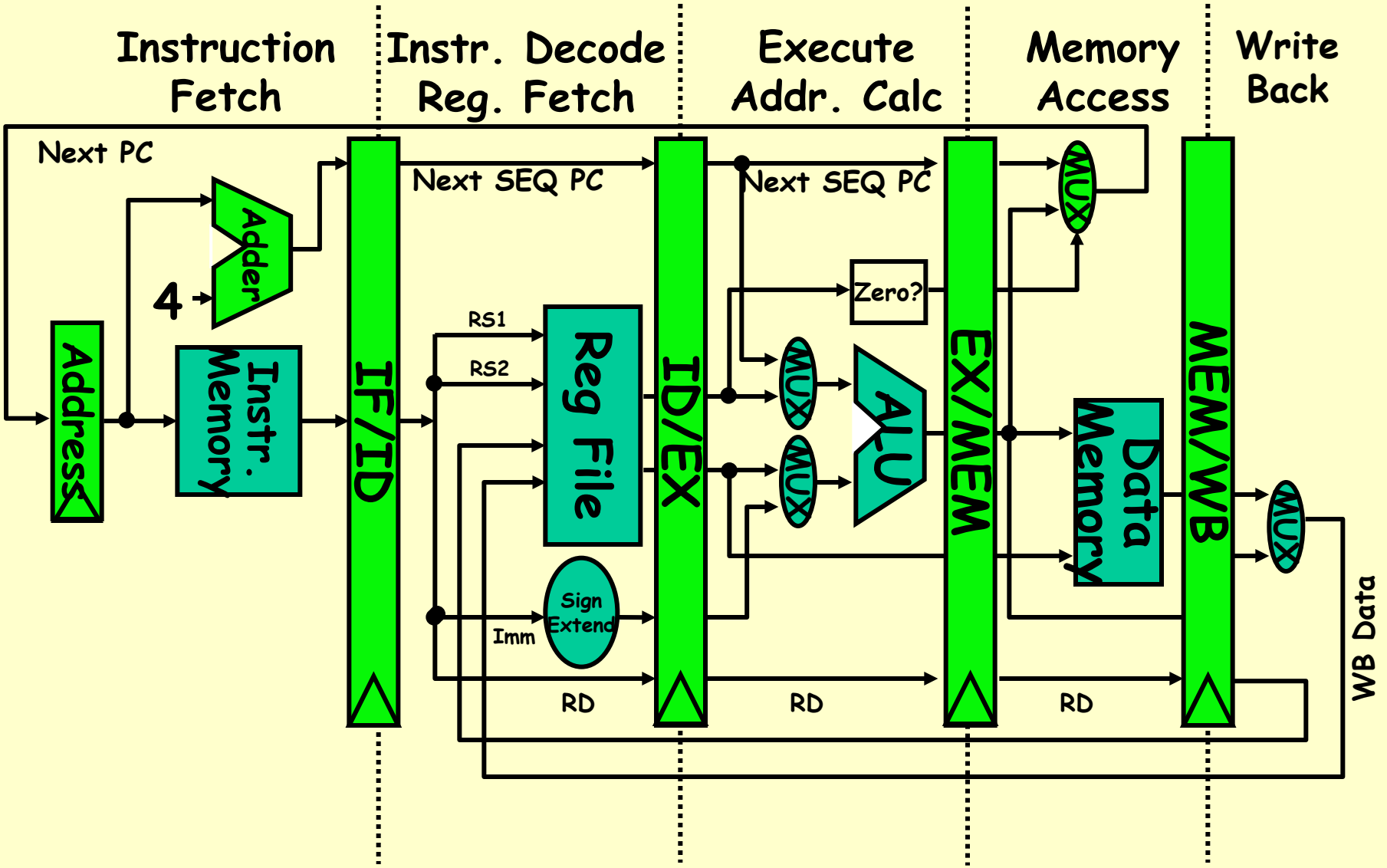


Pipeline stages

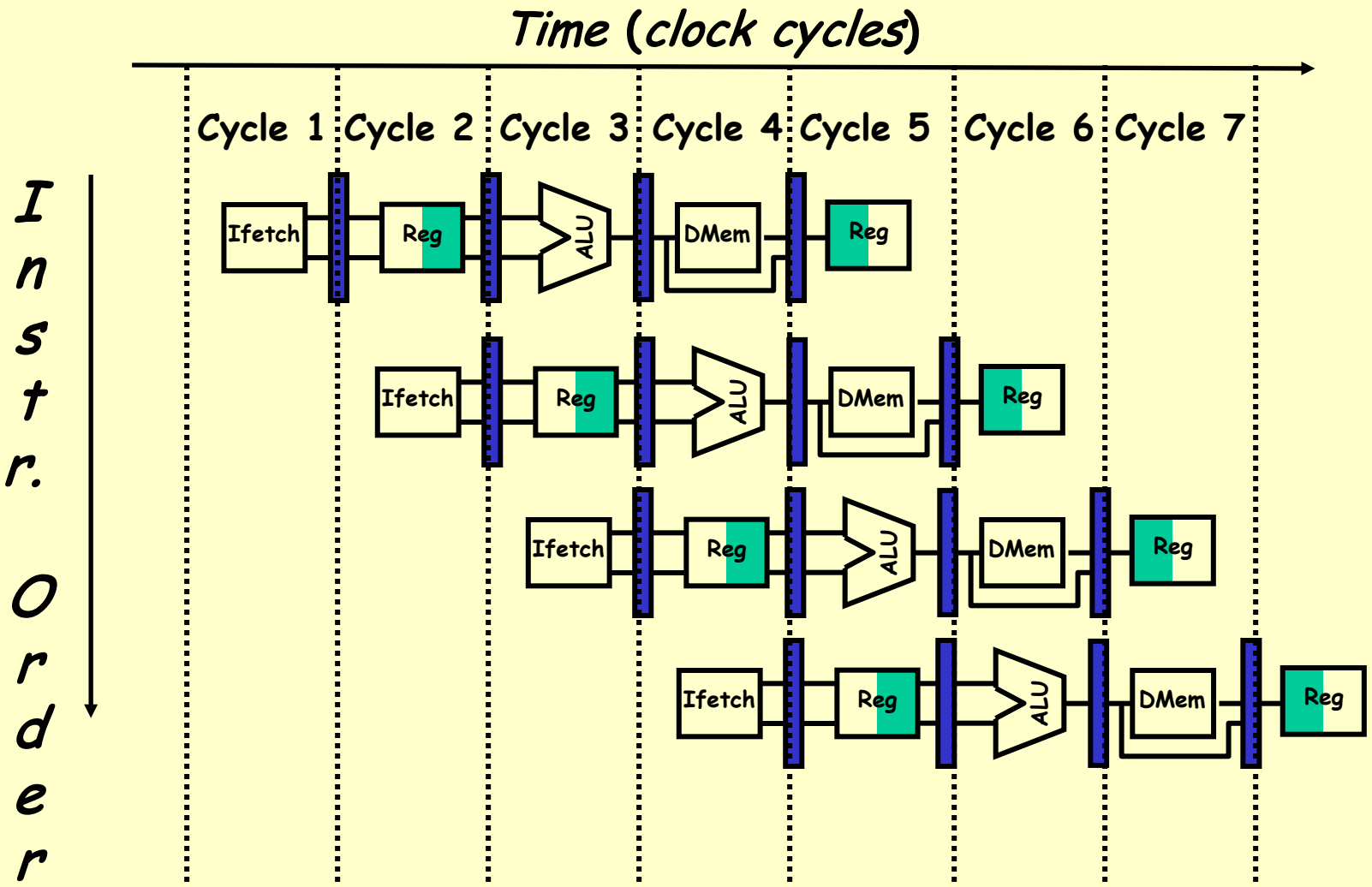
- **Fetch**
  - Select current PC
  - Read instruction
  - Compute incremented PC
- **Decode**
  - Read program registers
- **Execute**
  - Operate ALU
- **Memory**
  - Read or write data memory
- **Write Back**
  - Update register file



Alternate view (Hennessey & Patterson)



Visualizing pipeline behavior



Stylized pipeline performance

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Stylized pipeline performance

Assumes L1 D-cache

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Assumes L1 I-cache

Quantifying the speedup

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

- **Assume**
  - 40% ALU operations
  - 20% branches
  - 30% Load operations
  - 10% Store operations
  - No pipeline penalty
- **Average (non-pipelined) instruction duration = 650 ps**

$$0.4 \times 600 + 0.2 \times 500 + 0.3 \times 800 + 0.1 \times 700 = 650$$

Quantifying the speedup (continued)

$$Speedup = \frac{AveUnPipelinedTime}{AvePipelinedTime} = \frac{650}{200} = 3.25$$

- **Unpipelined architecture would allow variable duration instructions**
  - Branches much faster than Loads
- **Pipelined architecture requires every cycle to take exactly the same time**
  - Some wasted time in Branch, ALU, and Store operations
- **Speedup is less if there is a penalty for pipelining**

Quantifying the speedup (continued)

$$Speedup = \frac{AveUnPipelinedTime}{AvePipelinedTime} = \frac{650}{200} = 3.25$$

- **Unpipelined architecture would allow variable duration instructions**
  - Branches much faster than Loa
- **Pipelined architecture requires every cycle to take exactly the same time**
  - Some wasted time in Branch, ALU, and Store operations
- **Speedup is less if there is a penalty for pipelining**

Note: If there is a penalty for including pipelining (vs. non-pipelined design), speed-up decreases

Life is never that simple!

- Introducing *hazards*

Definition!

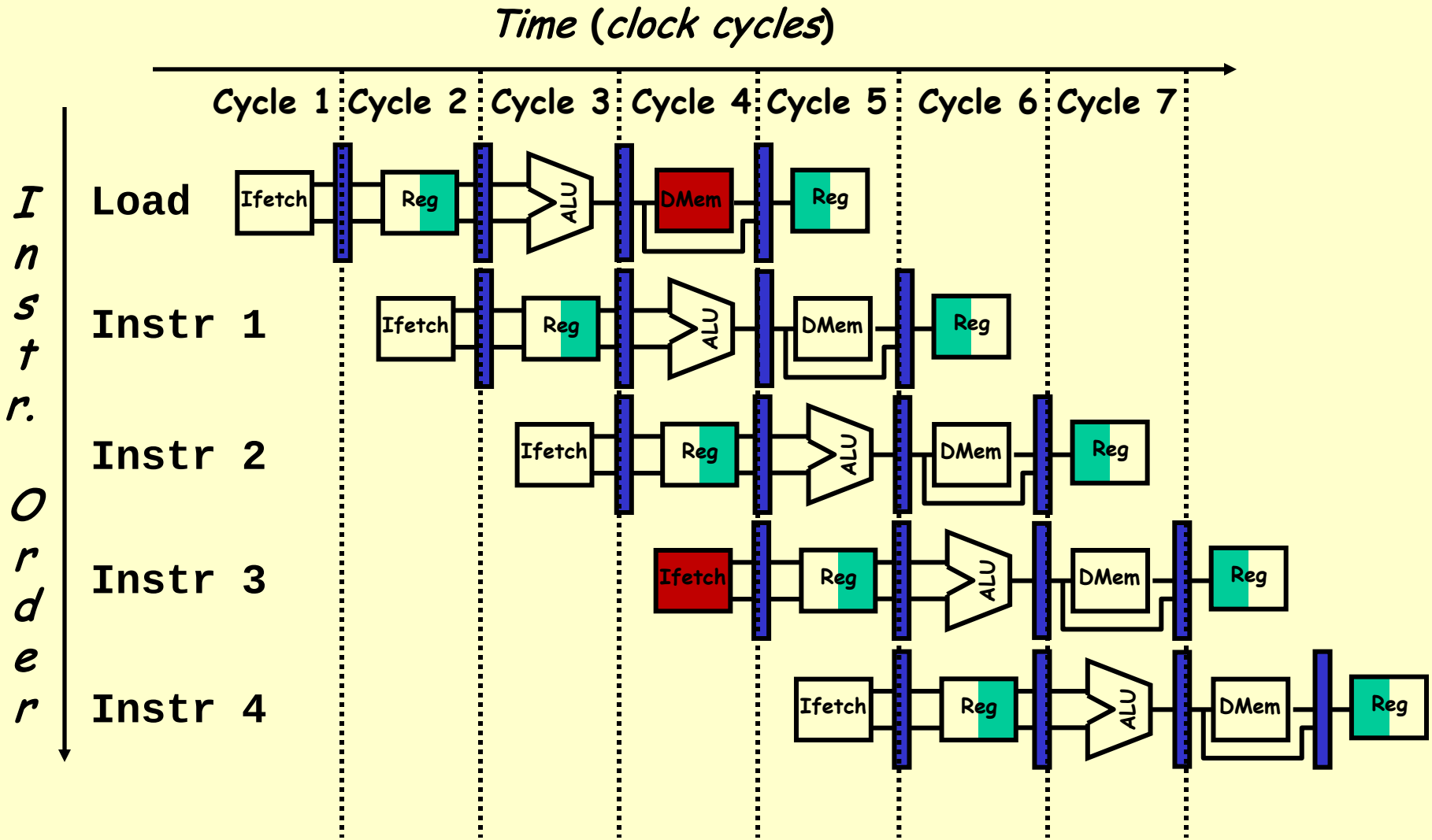
- I.e., factors that interfere with full and efficient pipelining

- Prevent instruction from starting or continuing on next cycle

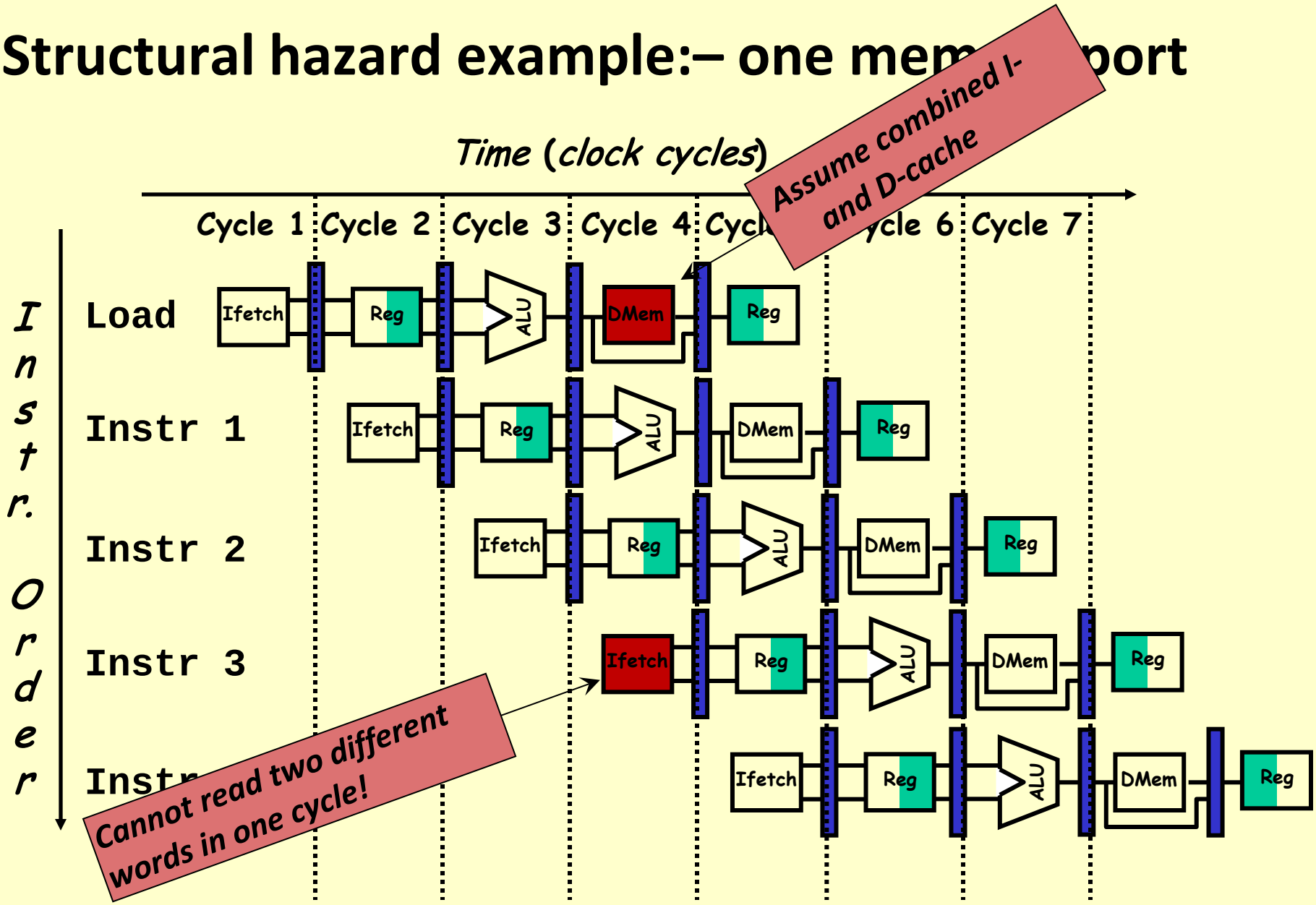
Hazards

- ***Structural*** **– resource conflicts among successive instructions**
  - Hardware cannot support all possible combinations of instructions in rapid succession
- ***Data*** **– dependencies of instructions on results of other instructions**
  - *x = a * b + c* Cannot add until multiply is done
- ***Control*** **– branches that change instruction fetch order**
  - Will affect instructions that have already been fetched!

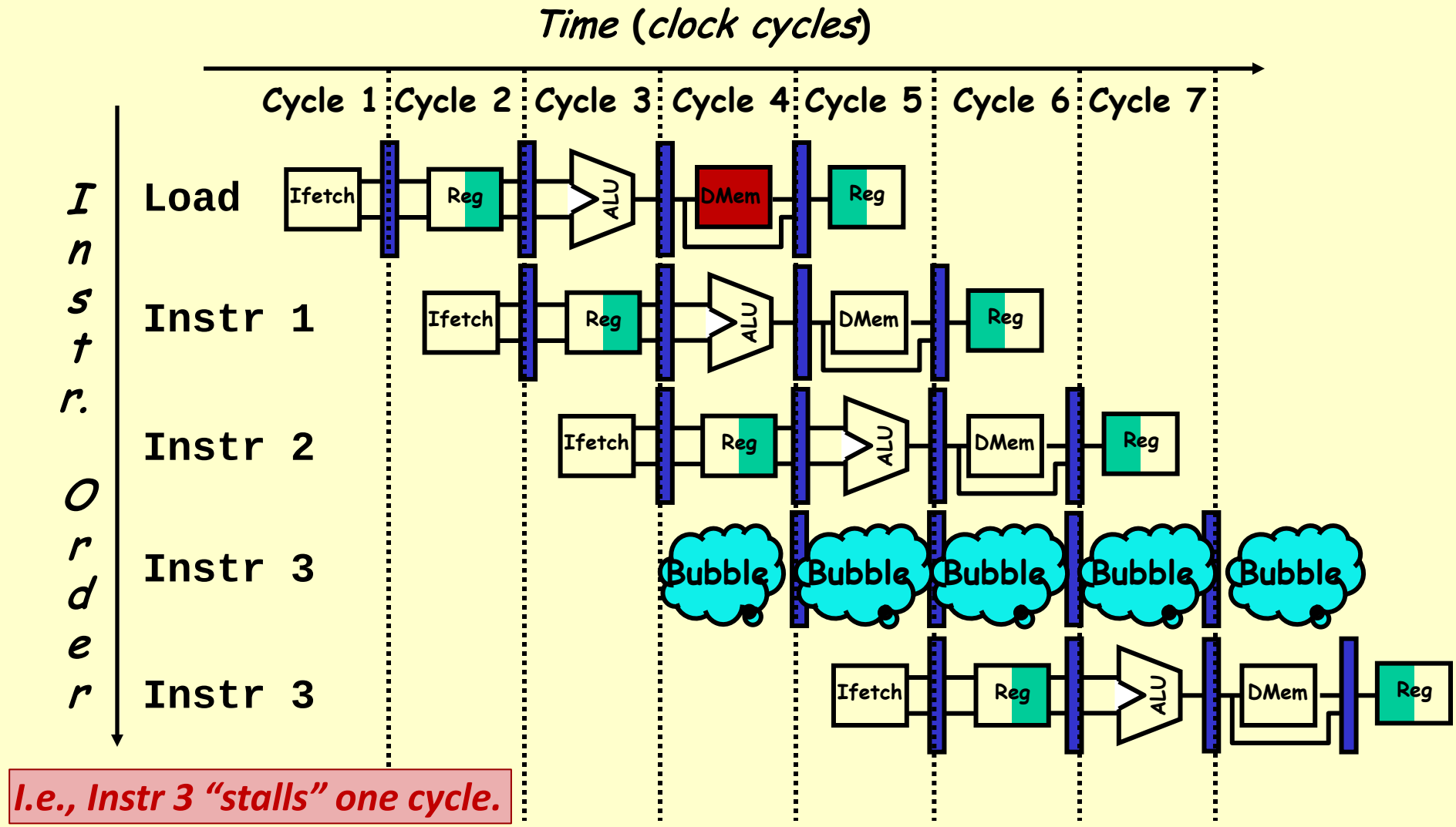
Structural hazard example:– one memory port



Structural hazard example:– one mem port



Structural hazard example:– one memory port



"Harvard Architecture"

- (Originally) physically separate storage and signal pathways for instructions and data
- (Today) separate I-caches and D-caches

Other structural hazards (examples)

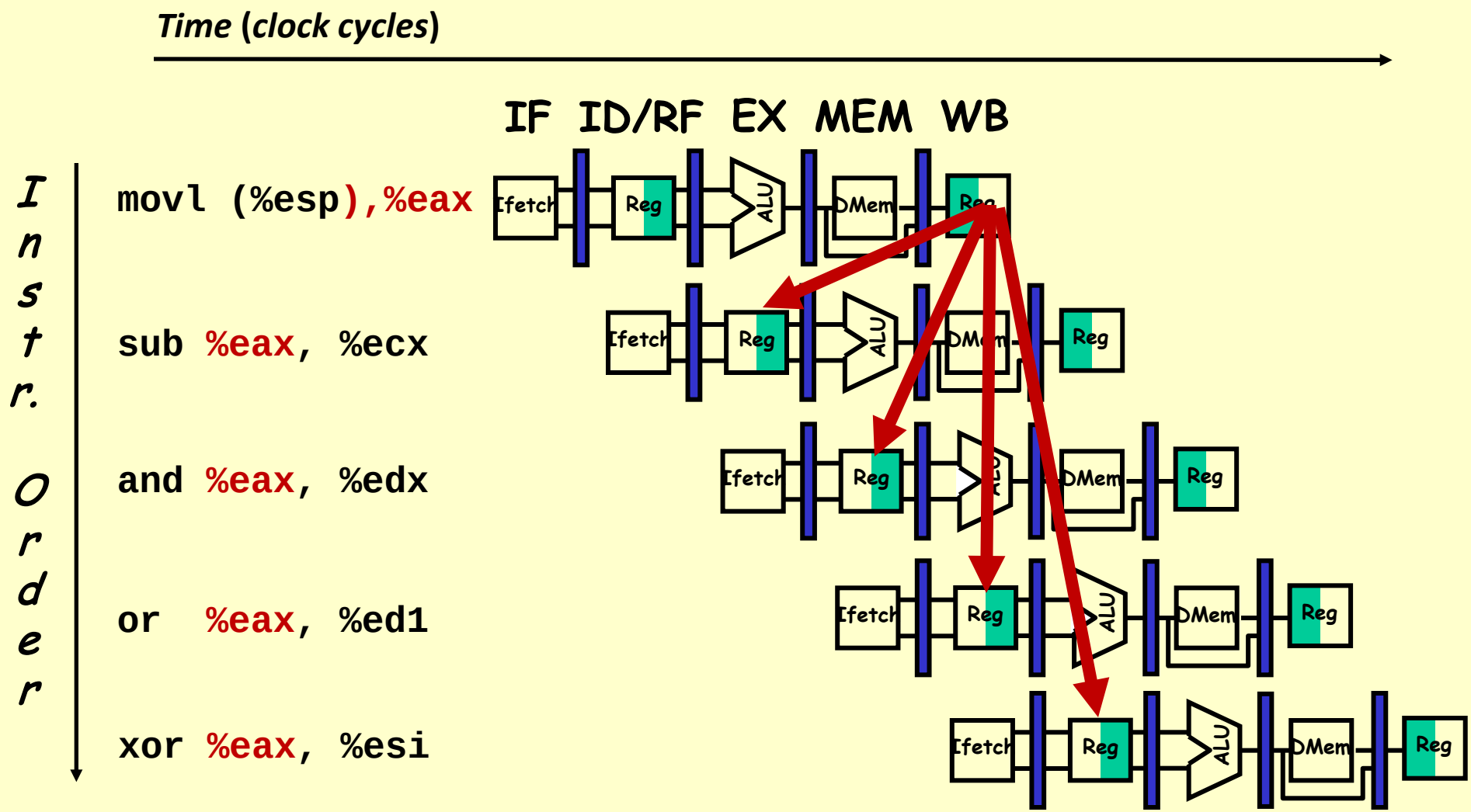
- **Multiply**
  - Expensive in gates to make fully pipelined
- **Floating-point divide**
  - Very expensive to make fully pipelined
- **Floating-point square-root**
  - Prohibitively expensive to pipeline at all

Structural hazards (conclusion)

- **Processor hardware *must* check**
  - Introduce “stalls”
- **Compiler *should be* aware**
  - Re-arrange compiled code

Questions?

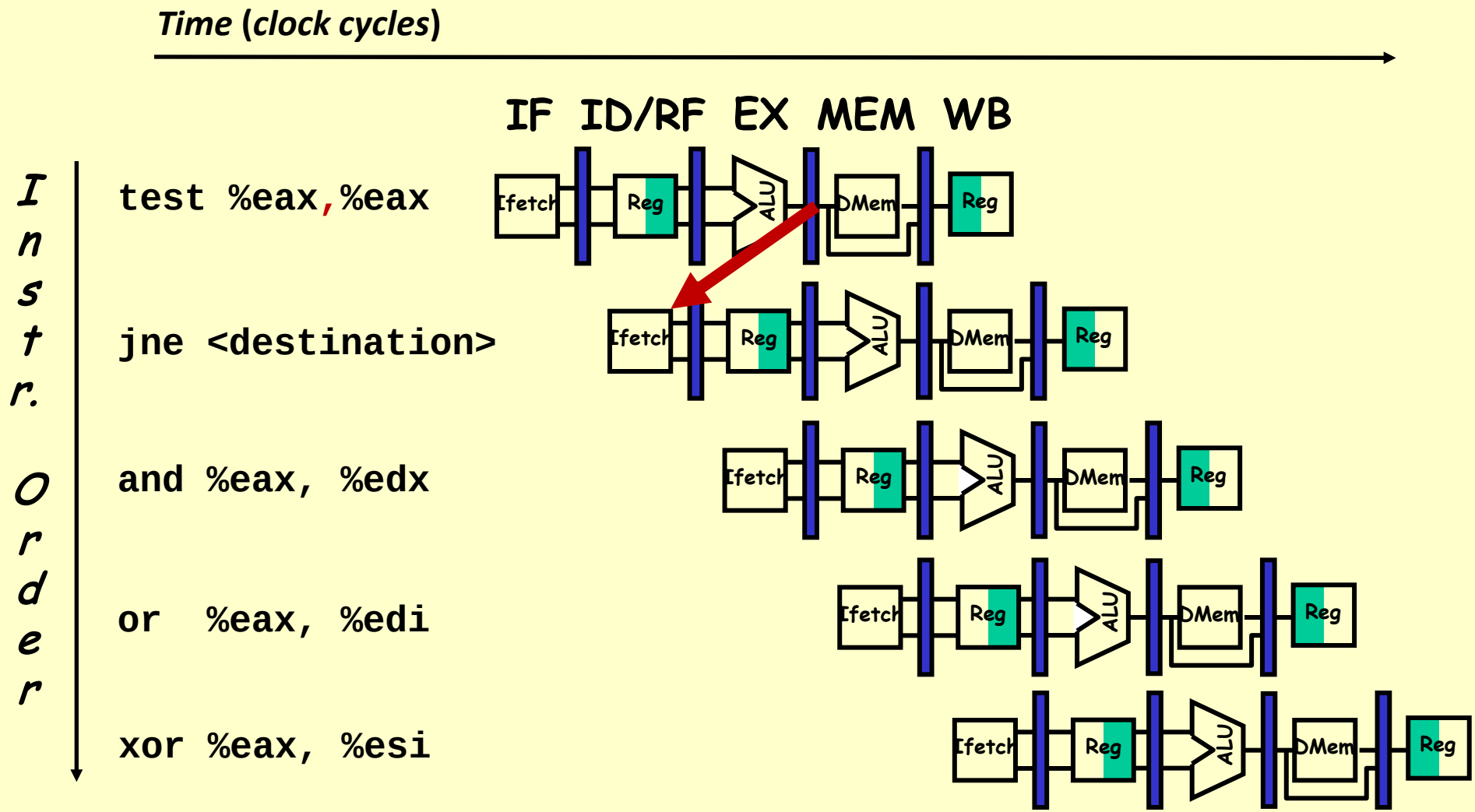
Data hazard on %eax



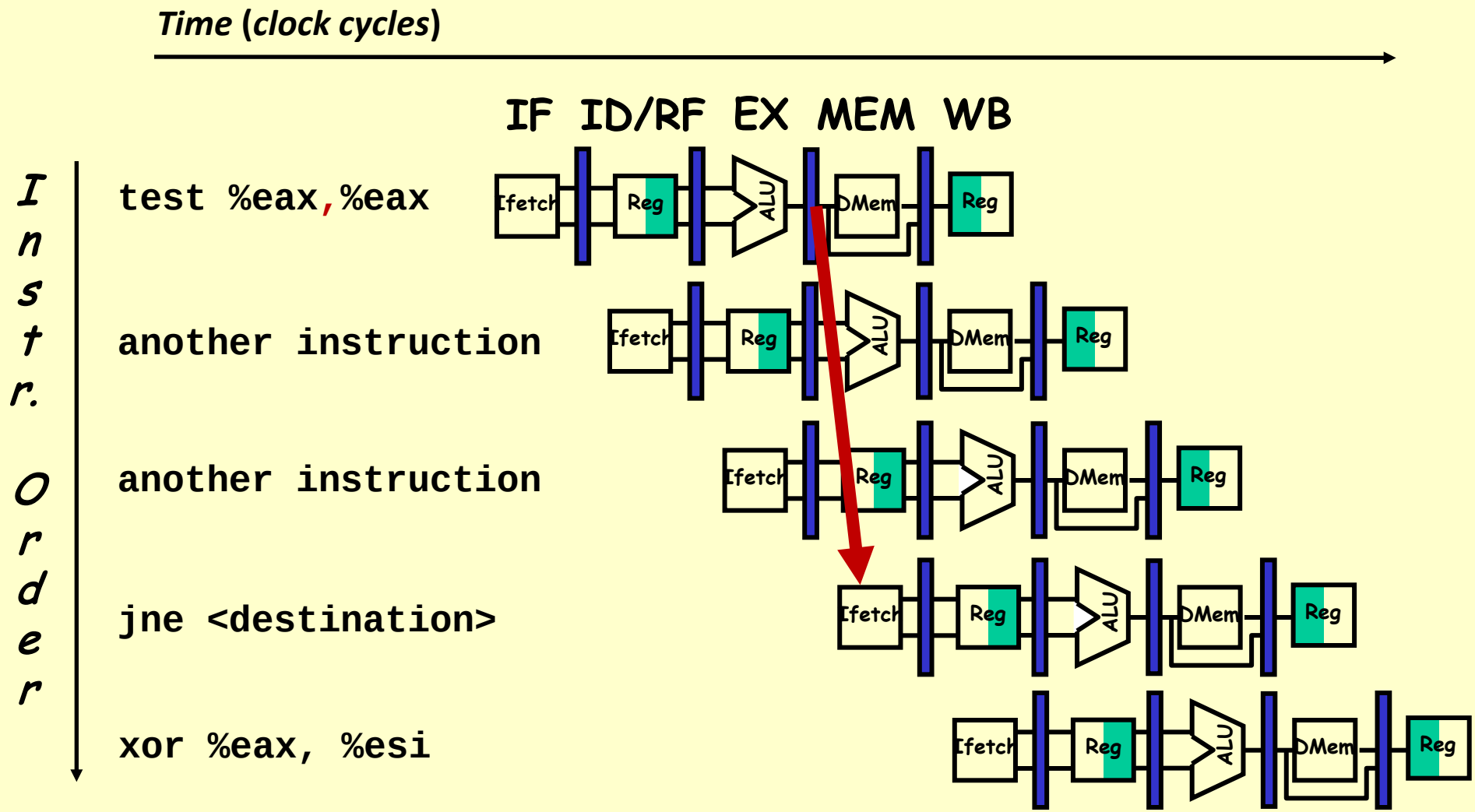
Data hazards

- **Extremely common**
- **Some hardware solutions**
  - Special forwarding data paths
  - Other extraordinary solutions
- **Compilers need to beware!**
  - Move code to minimize
  - Be aware during register allocation

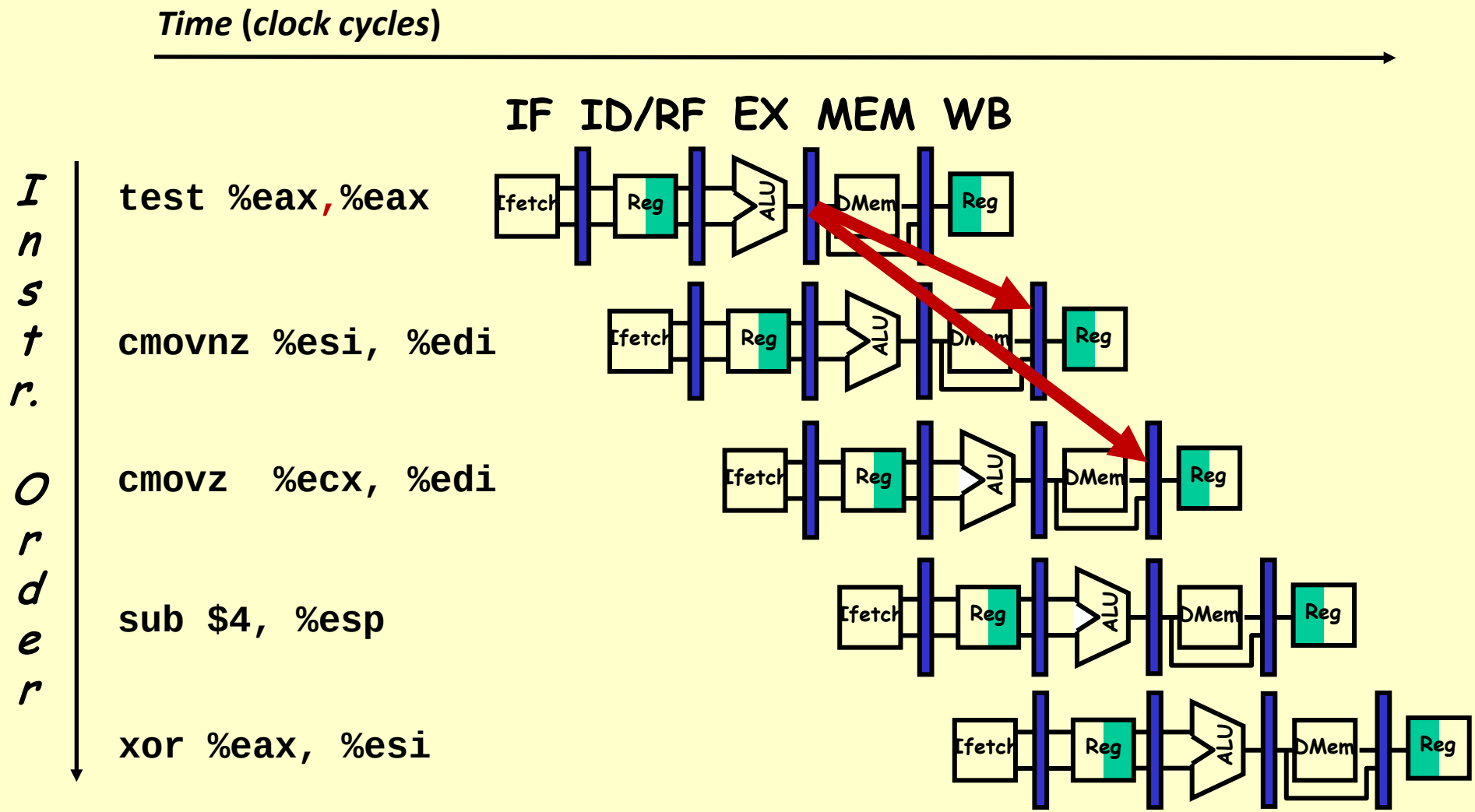
Control hazard on jumps



Control hazard on jumps — better



Control hazard on jumps — even better



Other control hazards

- **Call — PC changes on very next instruction**
- **Ret — ditto**
  - Plus data hazards on
- **Architectural solutions**
  - Branch delay
- **Compiler solutions**
  - Careful code motion
- **Hardware solutions**
  - Speculative execution

Summary

- **Pipelining makes computers faster**
  - Ubiquitous since ~1985
- **Rule of "no free lunch"**
  - Hazards limit speedup
- **Compilers must be keenly aware**
- **Programmers should not forget**

Questions?