Computer Architecture Great Ideas in Computer Architecture (Machine Structures) Pipeline Parallelism

Pipeline: Hazards

CSCE430/830

Lecturer: Prof. Hong Jiang

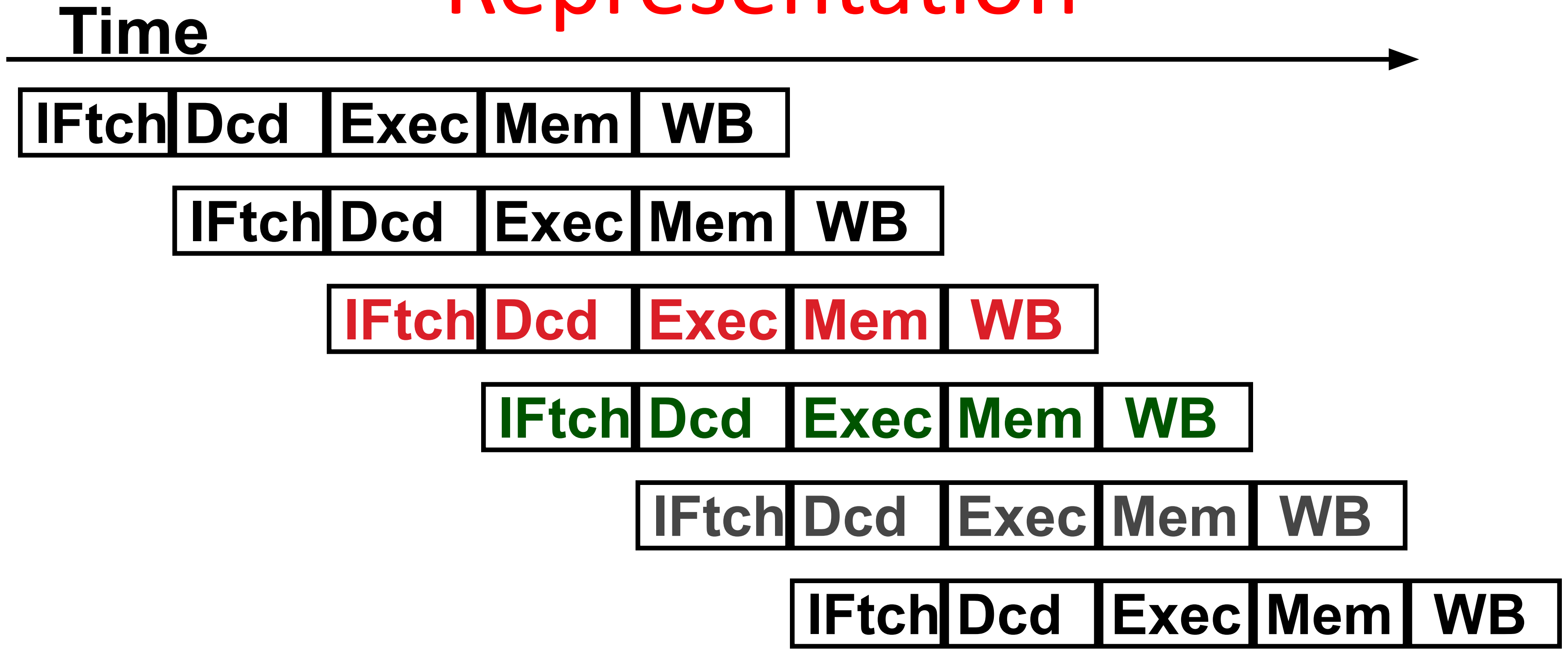
Courtesy of Prof. Yifeng Zhu, U of Maine

Fall, 2006

Mike Franklin
Dan Garcia
http://inst.eecs.Berkeley.edu/~cs61c/fa11

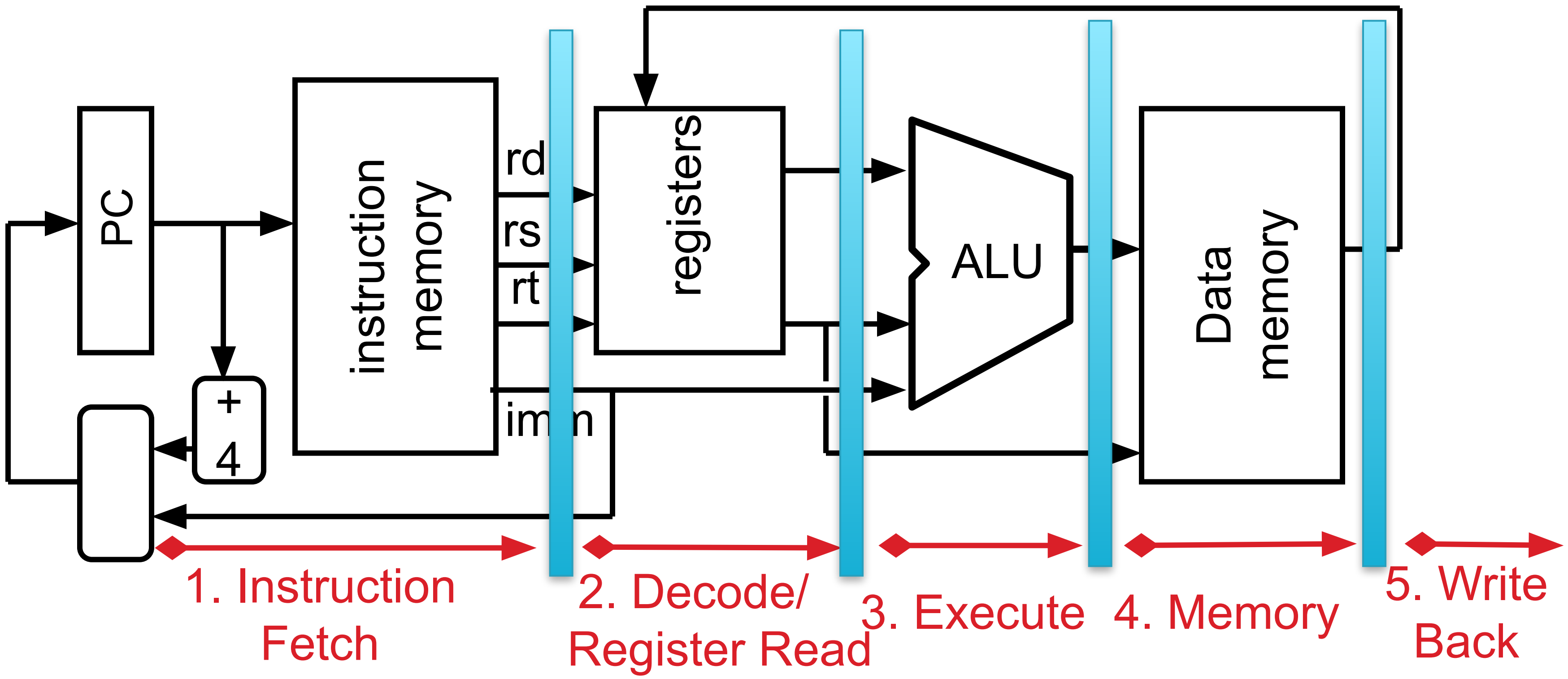
Fall, 2011

Pipelined Execution Representation

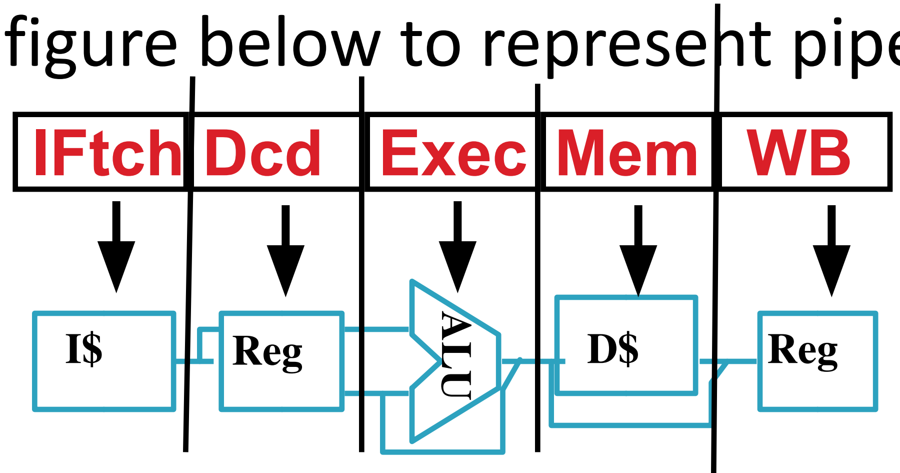


- Every instruction must take same number of steps, also called pipeline "stages", so some will go idle sometimes

Graphical Pipeline Diagrams

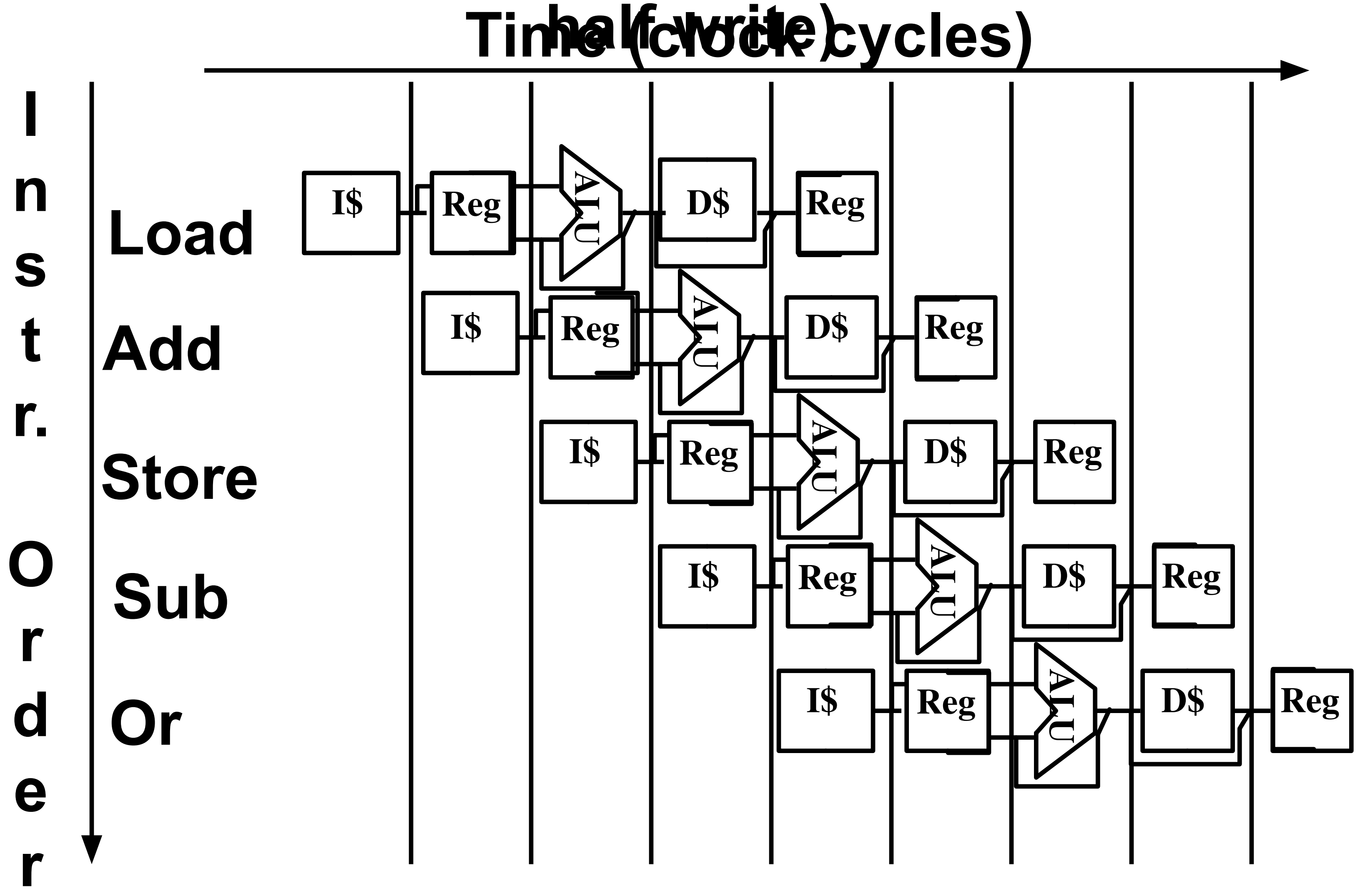


- Use datapath figure below to represent pipeline



Graphical Pipeline Representation

(In Reg, right half highlight read, left half write)



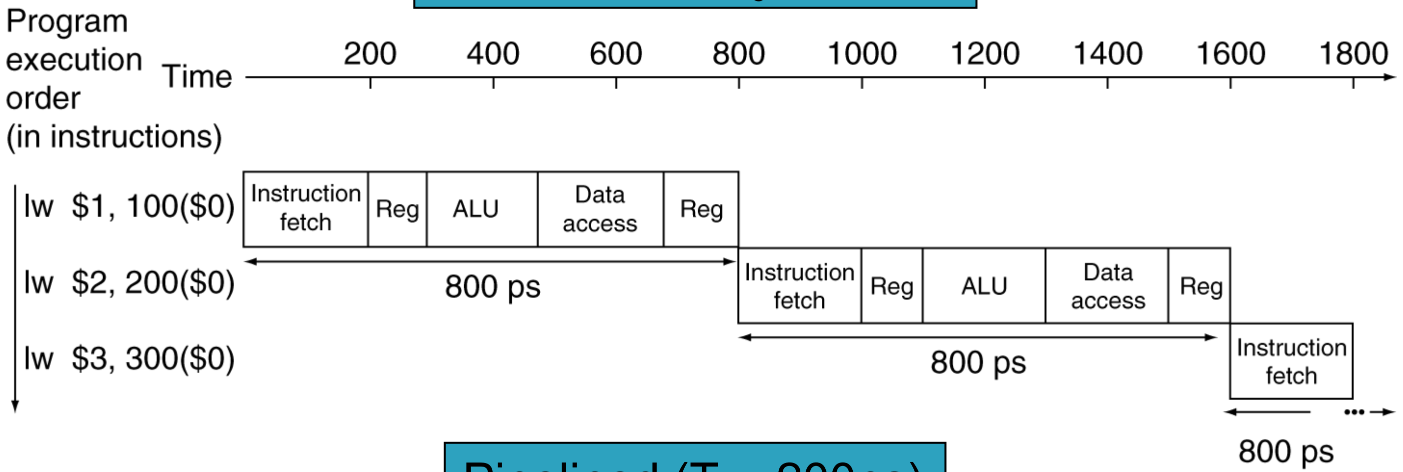
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

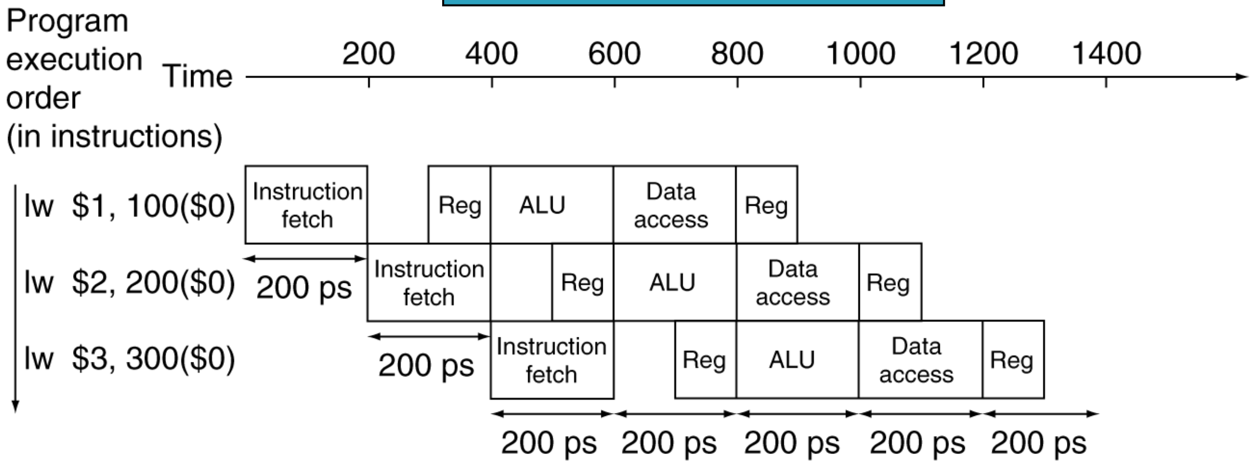
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Comments about Pipeline

- The good news
  - Multiple instructions are being processed at same time
  - This works because stages are isolated by registers
  - Best case speedup of N
- The bad news
  - Instructions interfere with each other - hazards
    - Example: different instructions may need the same piece of hardware (e.g., memory) in same clock cycle
    - Example: instruction may require a result produced by an earlier instruction that is not yet complete

Pipeline Hazards

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: two different instructions use same h/w in same cycle
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Pipelining of branches & other instructions that change the PC

Summary - Pipelining Overview

- Pipelining increase throughput (but not latency)
- Hazards limit performance
  - Structural hazards
  - Control hazards
  - Data hazards

Pipelining Outline

- Introduction
  - Defining Pipelining
  - Pipelining Instructions
- Hazards
  - Structural hazards ▯
  - Data Hazards
  - Control Hazards
- Performance
- Controller implementation

Hazards

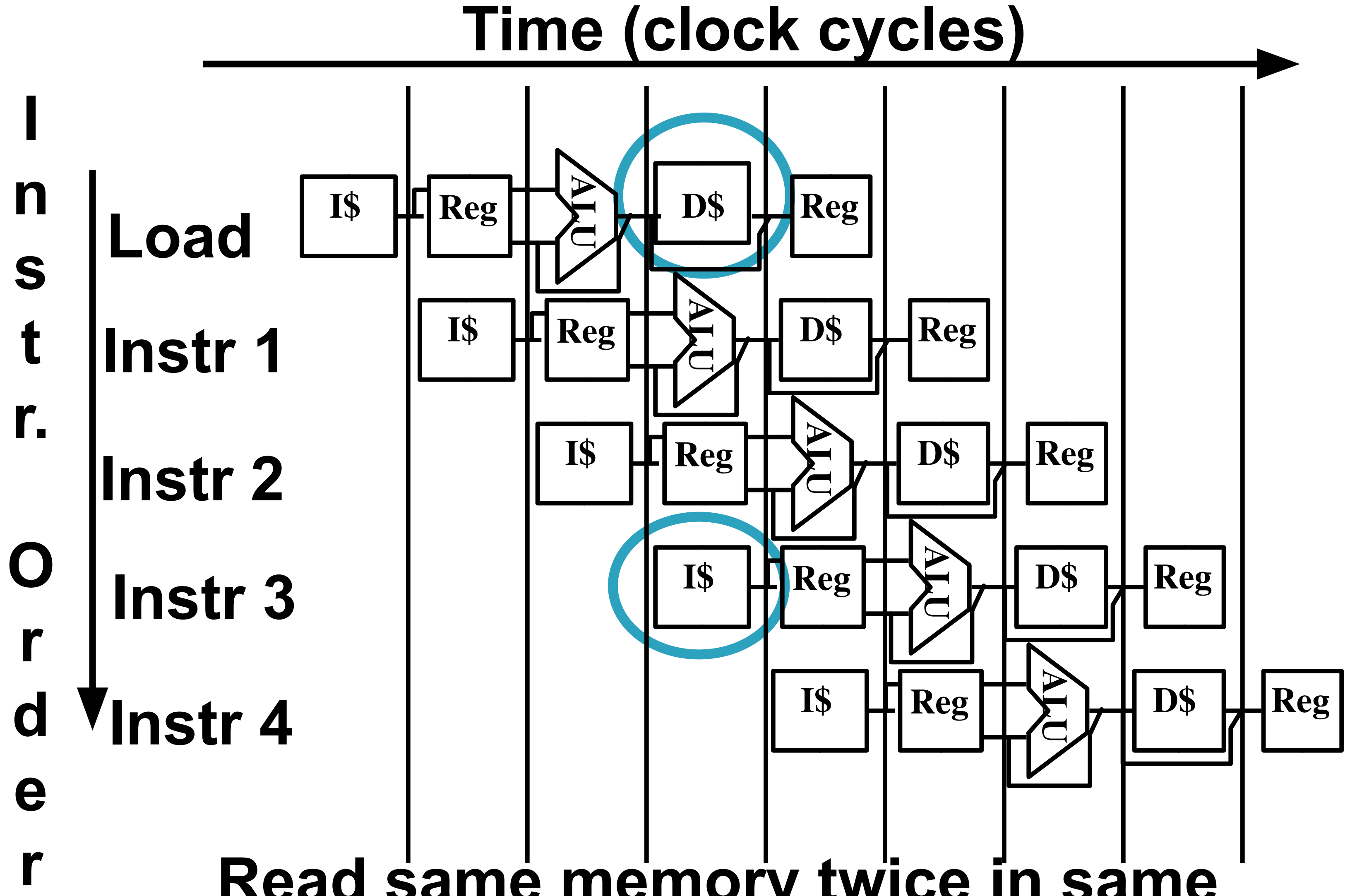
Situations that prevent starting the next logical instruction in the next clock cycle

1. Structural hazards
   – Required resource is busy (e.g., roommate studying)
2. Data hazard
   – Need to wait for previous instruction to complete its data read/write
3. Control hazard
   – Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)

1. Structural Hazards

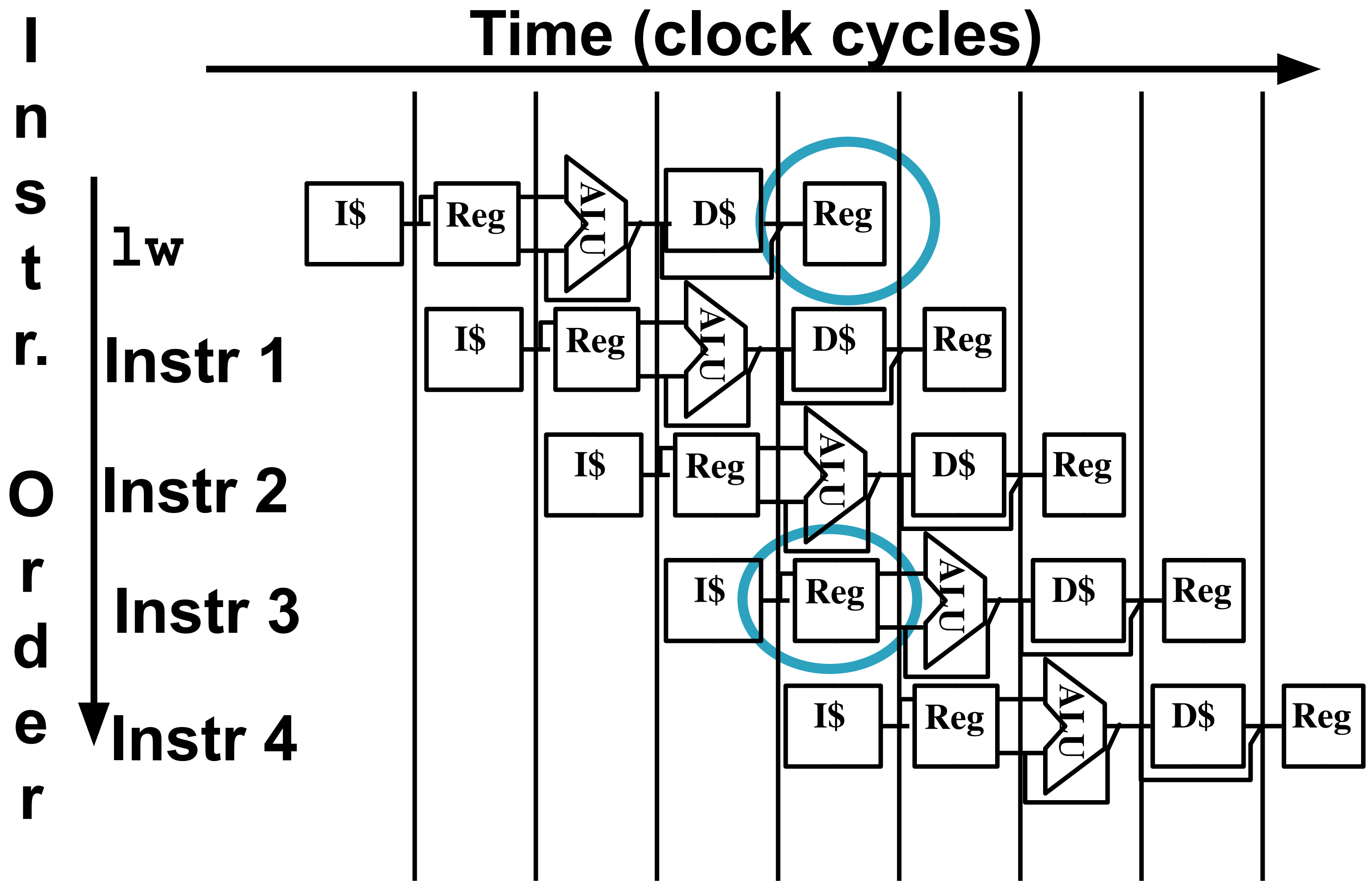
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/Store requires memory access for data
  - Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline "*bubble*"
- Hence, pipelined datapaths require separate instruction/data memories
  - In reality, provide separate L1 I$ and L1 D$

1. Structural Hazard #1: Single Memory



Read same memory twice in same clock cycle

1. Structural Hazard #2: Registers (1/2)



Can we read and write to registers simultaneously?

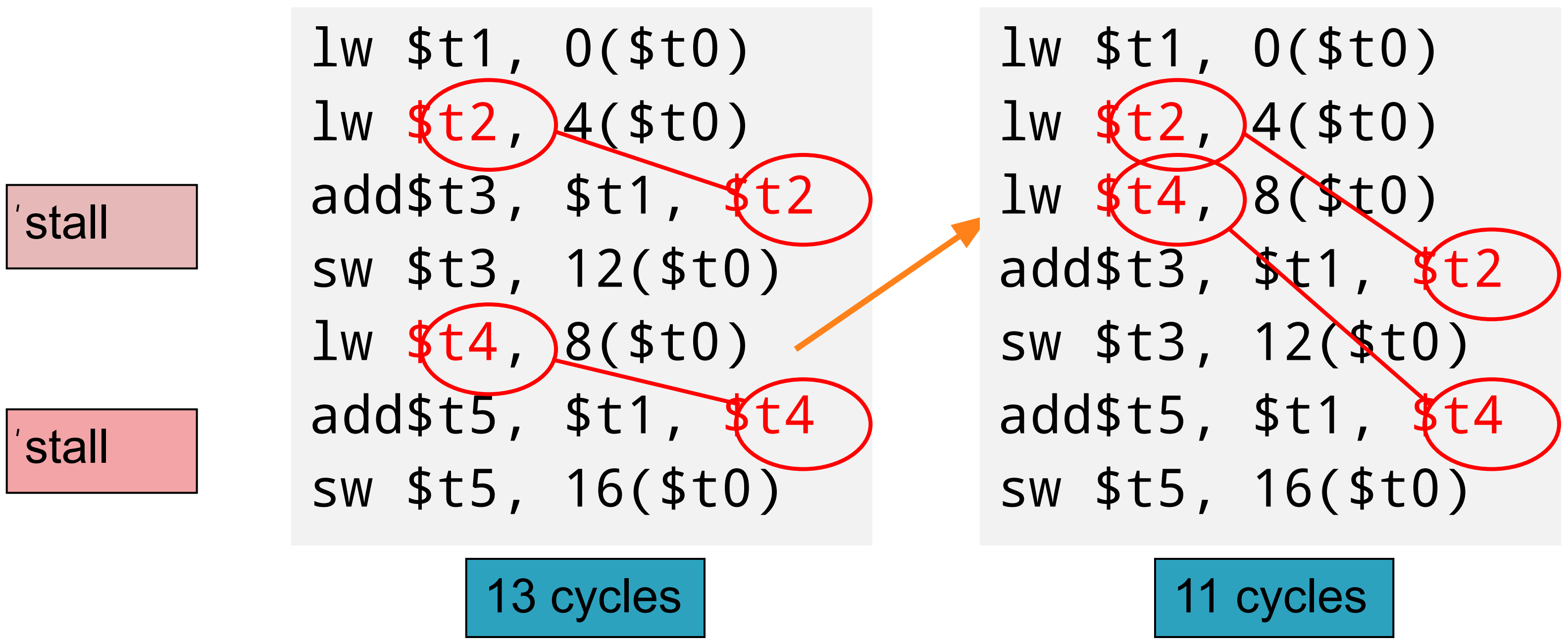
1. Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is *VERY* fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle

  2) Build RegFile with independent read and write ports
- Result: can perform Read and Write during same clock cycle

Data Hazards: Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`



```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

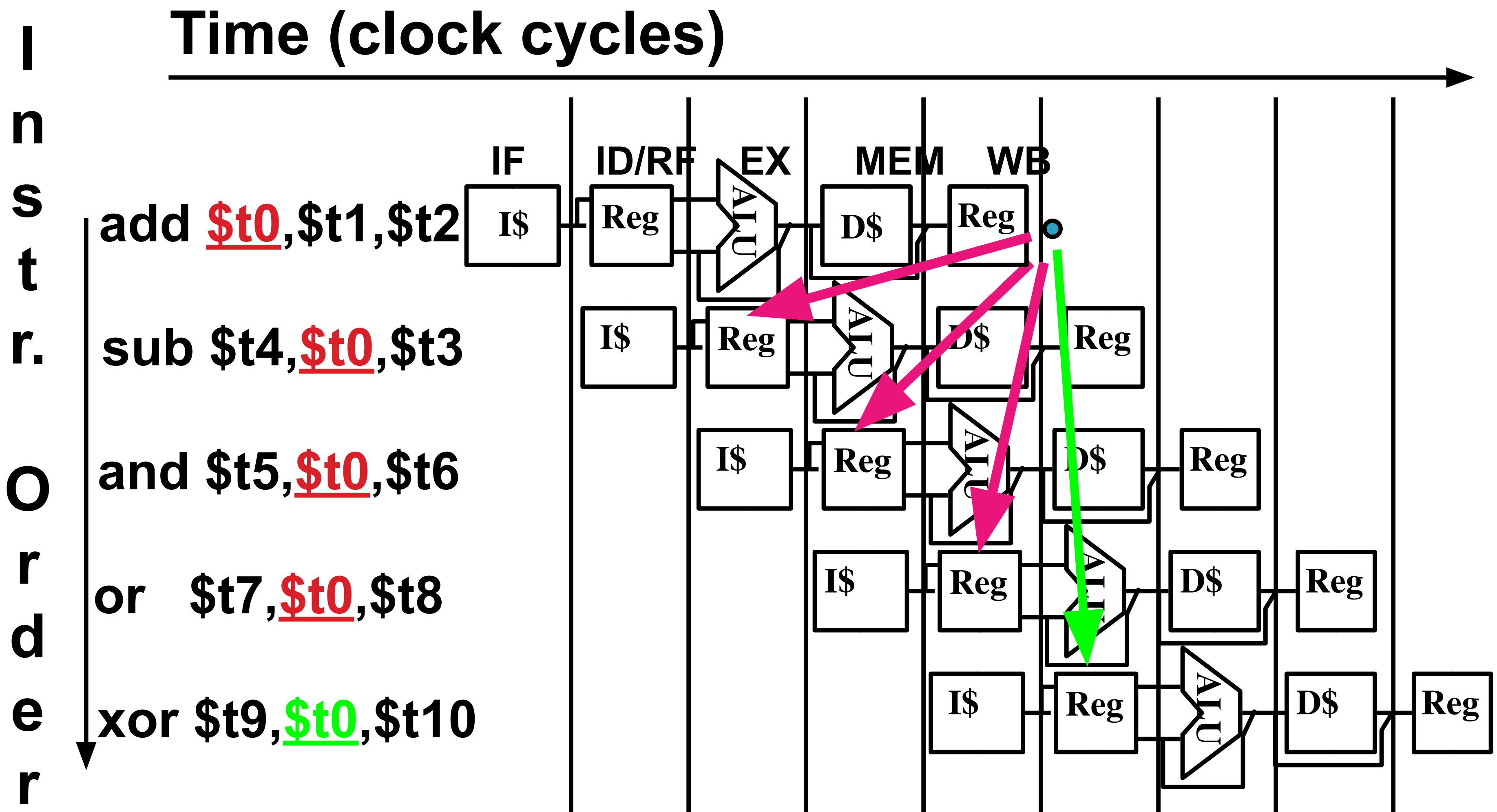
Data Hazards (1/2)

Consider the following sequence of instructions

```
add $t0, $t1, $t2
sub $t4, $t0 ,$t3
and $t5, $t0 ,$t6
or  $t7, $t0 ,$t8
xor $t9, $t0 ,$t10
```

Data Hazards (2/2)

- Data-flow backward in time are hazards



Data Hazard Solution: Forwarding

- Forward result from one stage to another

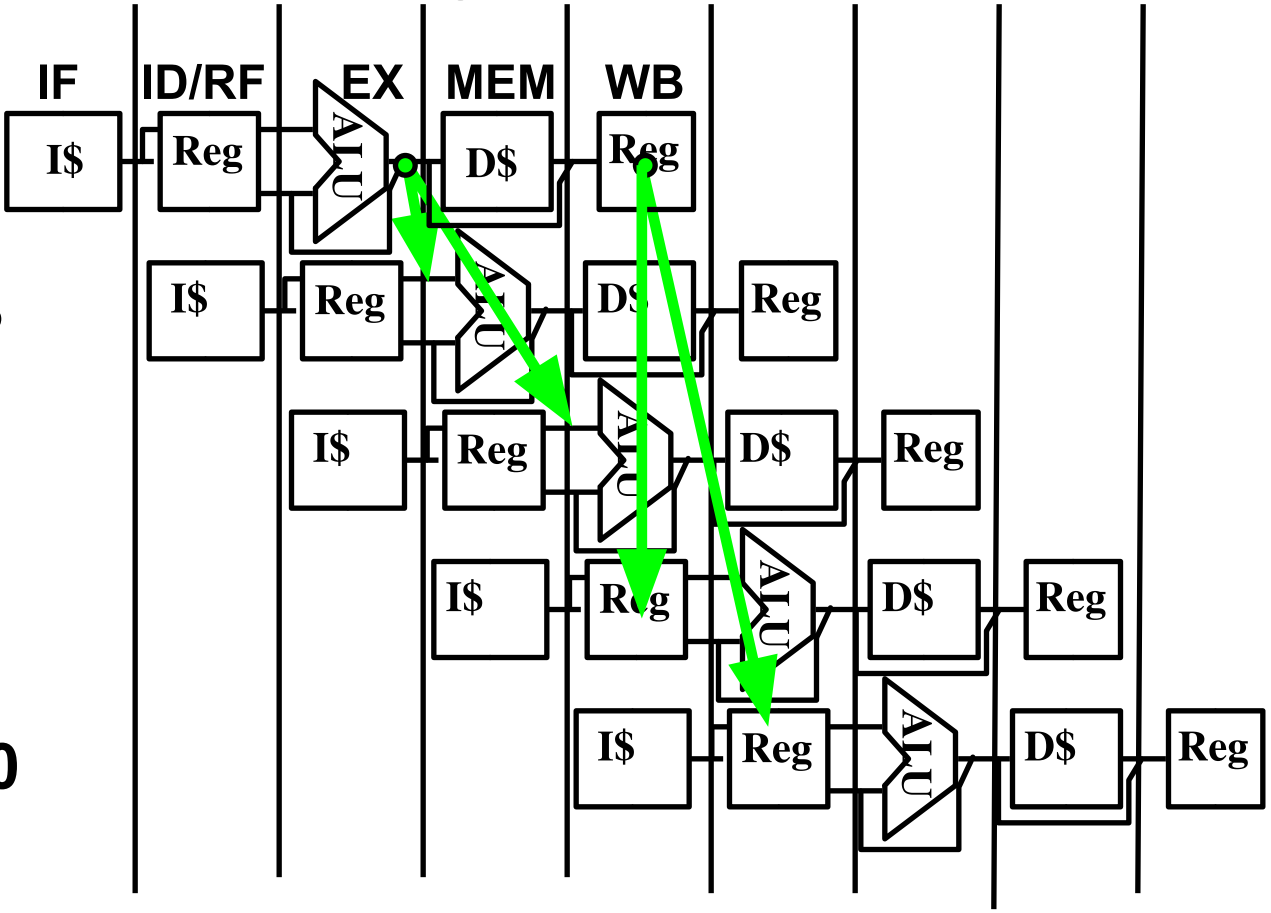
add $t0,$t1,$t2

sub $t4,$t0,$t3

and $t5,$t0,$t6

or $t7,$t0,$t8

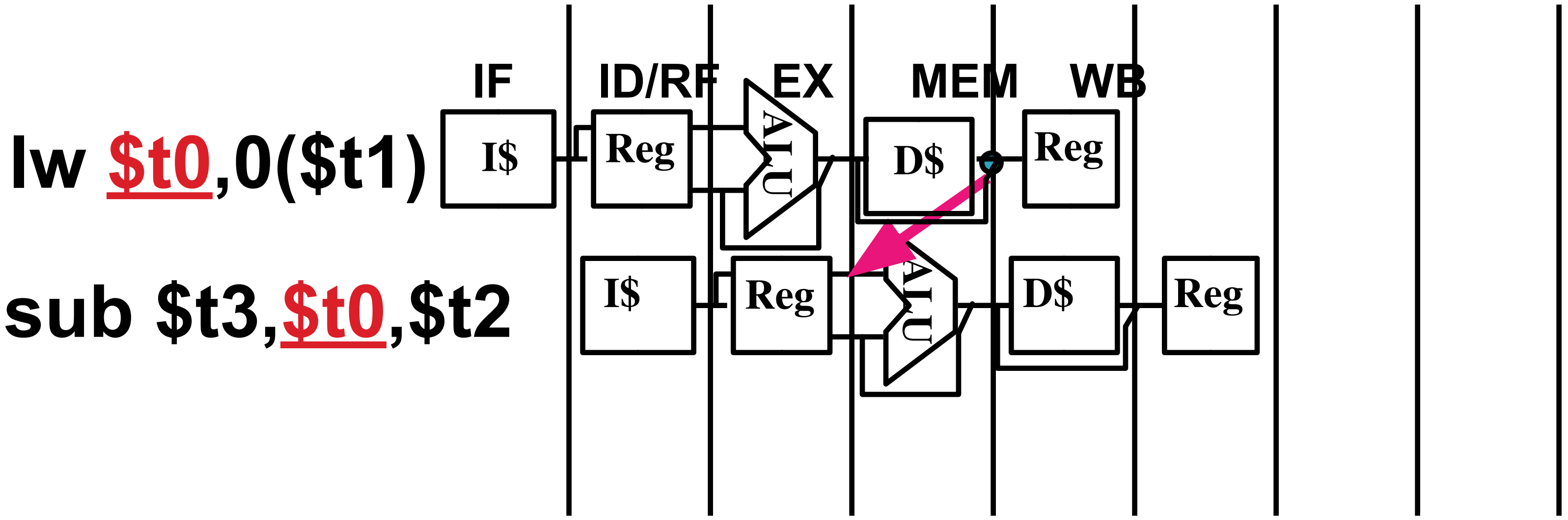
xor $t9,$t0,$t10



"`or`" hazard solved by register hardware

Data Hazard: Load/Use (1/4)

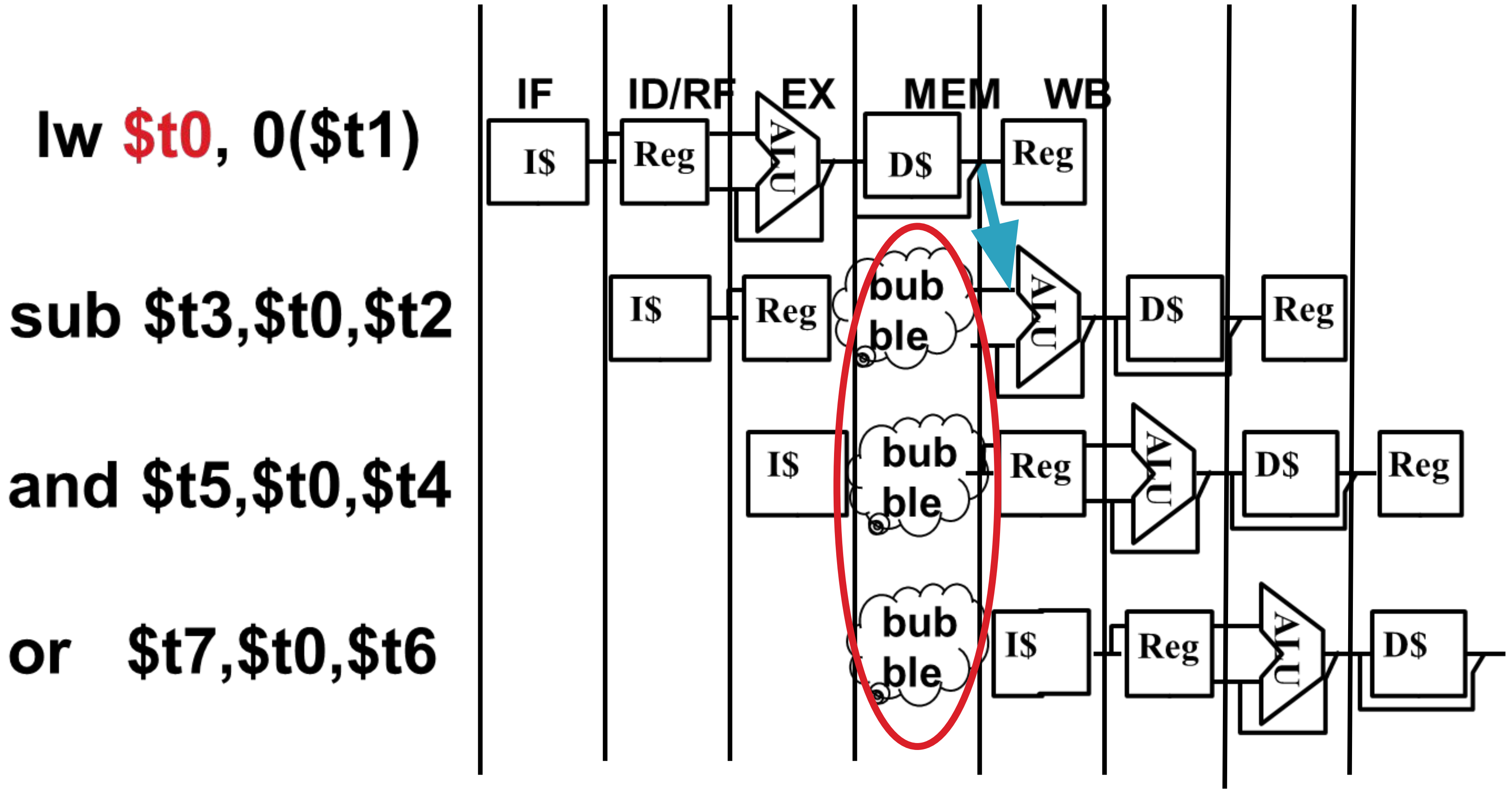
- Dataflow backwards in time are hazards



- Can't solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

Data Hazard: Load/Use (2/4)

Hardware stalls pipeline (Called "interlock")



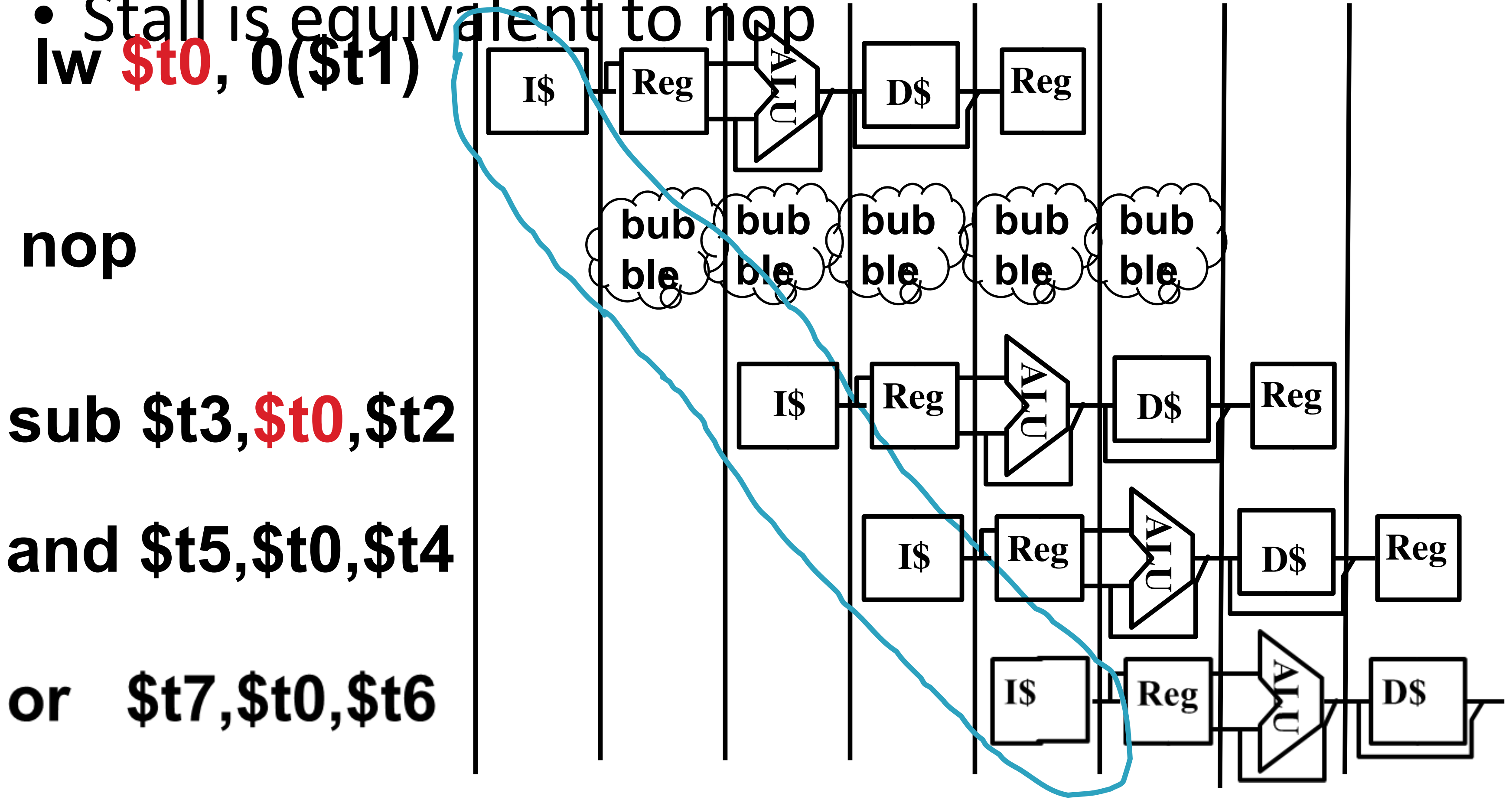
Not in MIPS: (MIPS = Microprocessor without Interlocked Pipeline Stages)

Data Hazard: Load/Use (3/4)

- Instruction slot after a load is called "load delay slot"
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- Alternative: If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)

Data Hazard: Load/Use (4/4)

- Stall is equivalent to nop



Pipelining and ISA Design

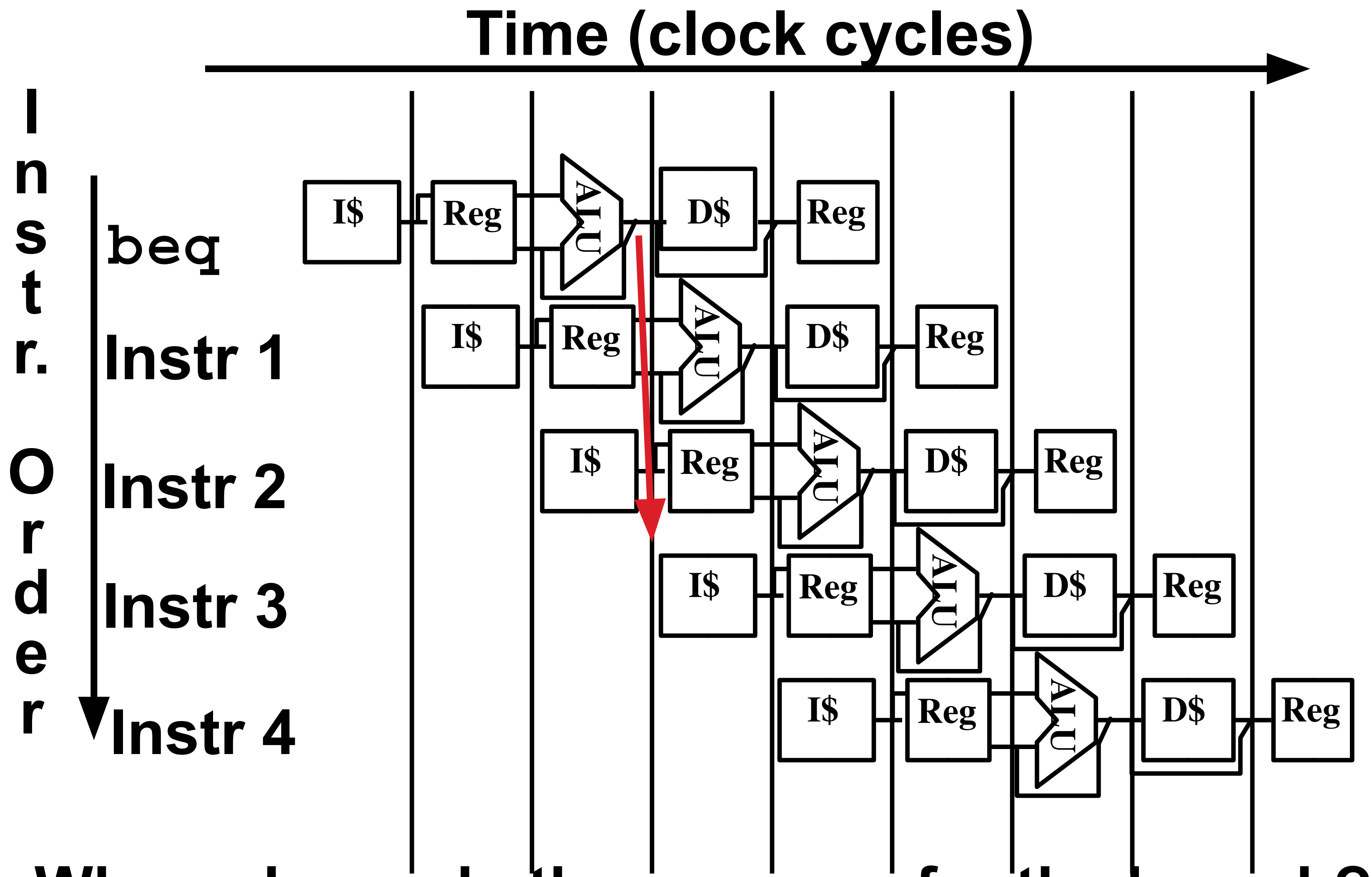
- MIPS Instruction Set designed for pipelining
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - x86: 1- to 17-byte instructions

  (x86 HW actually translates to internal RISC instructions!)
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

3. Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- BEQ, BNE in MIPS pipeline
- Simple solution Option 1: *Stall* on every branch until have new PC value
  - Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)

Stall => 2 Bubbles/Clocks



Where do we do the compare for the branch?

Until next time …

The BIG Picture

- Pipelining improves performance by increasing instruction throughput: exploits ILP
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

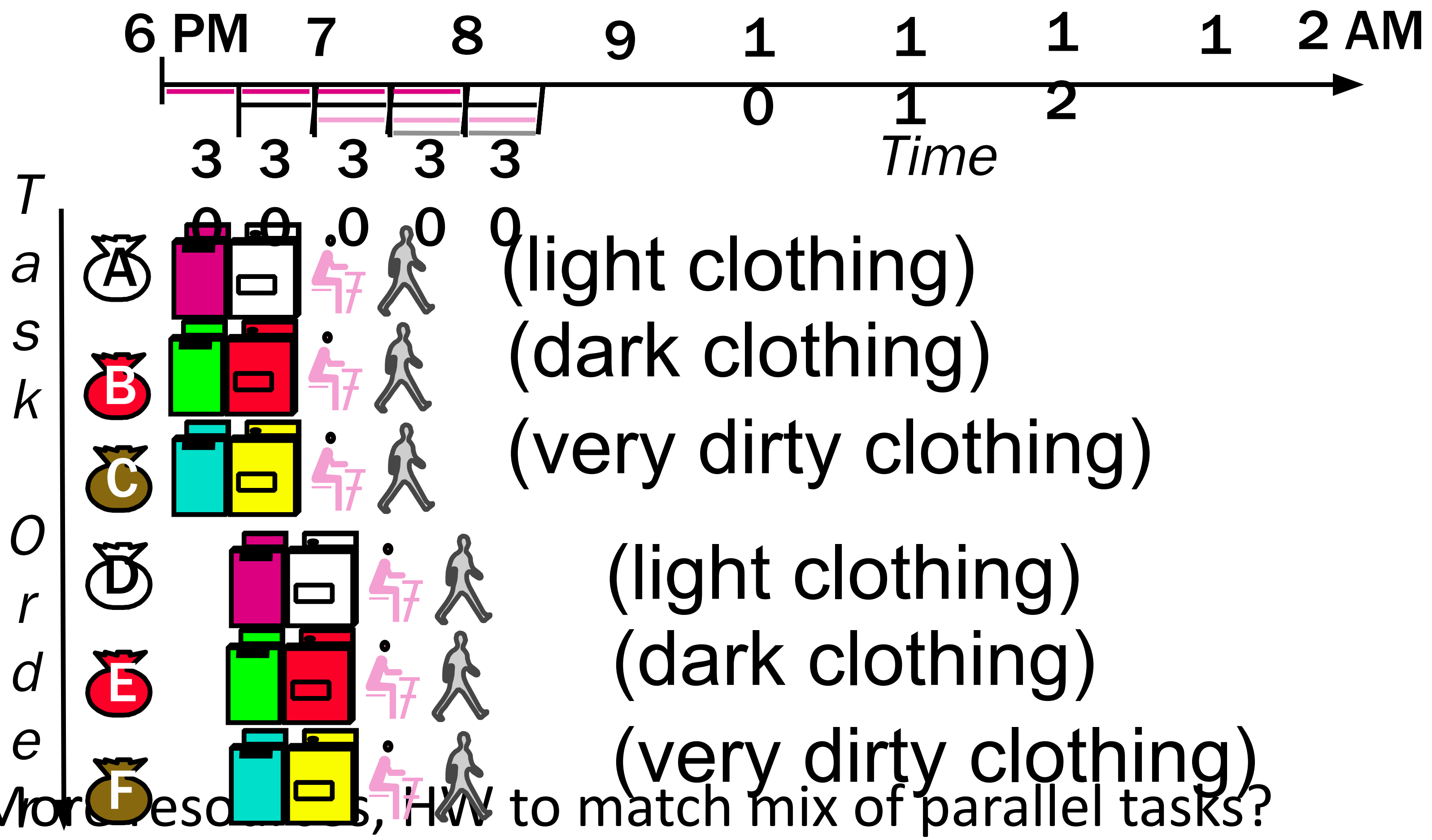
Greater Instruction-Level Parallelism (ILP)

- Deeper pipeline (5 => 10 => 15 stages)
  - Less work per stage ⇒ shorter clock cycle
- Multiple issue "superscalar"
  - Replicate pipeline stages ⇒ multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Superscalar Laundry: Parallel per stage



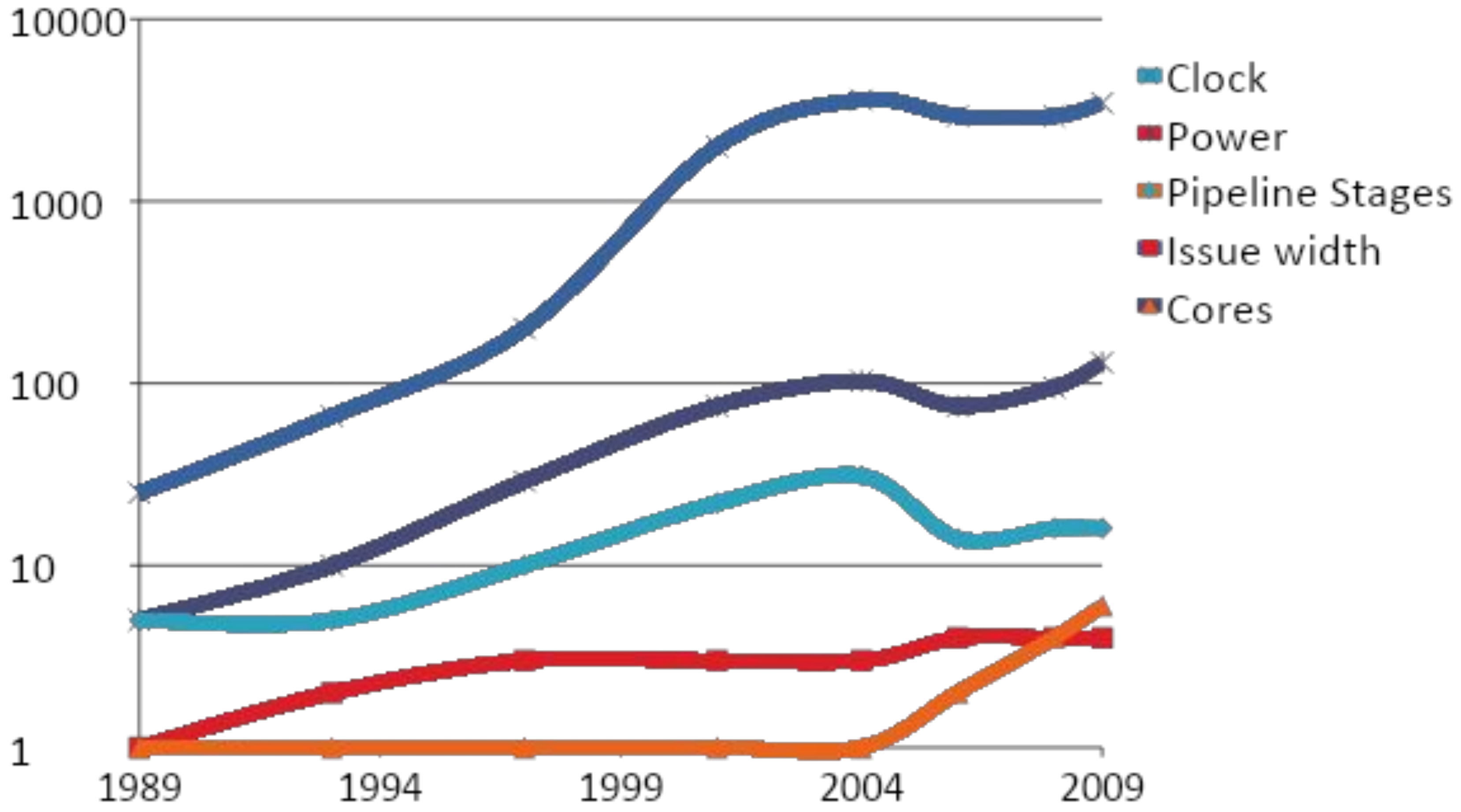
- More resources, HW to match mix of parallel tasks?

Pipeline Depth and Issue Width

- Intel Processors over Time

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Cores	Power
i486	1989	25 MHz	5	1	1	5W
Pentium	1993	66 MHz	5	2	1	10W
Pentium Pro	1997	200 MHz	10	3	1	29W
P4 Willamette	2001	2000 MHz	22	3	1	75W
P4 Prescott	2004	3600 MHz	31	3	1	103W
Core 2 Conroe	2006	2930 MHz	14	4	2	75W
Core 2 Yorkfield	2008	2930 MHz	16	4	4	95W
Core i7 Gulftown	2010	3460 MHz	16	4	6	130W

Pipeline Depth and Issue Width


Clock
Power
Pipeline Stages
Issue width
Cores