

# Computer Organization and Architecture

Under Graduate Course  
(B. Tech-Information Technology, 2<sup>nd</sup> Semester)  
Jan 2020-July 2020

By

**Dr. Satish Kumar Singh**



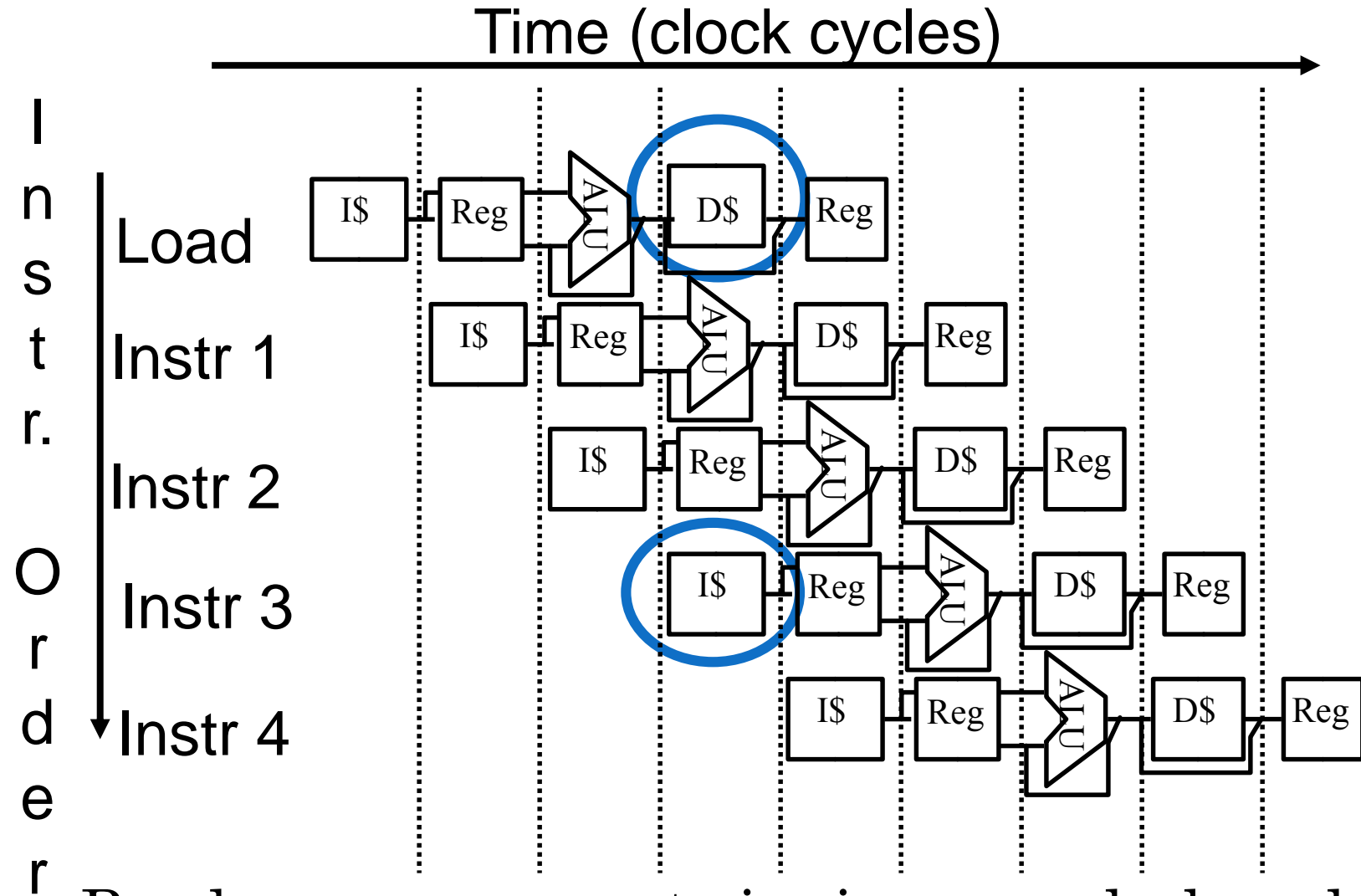
Associate Professor  
Indian Institute of Information Technology, Allahabad  
Email: [sk.singh@iiita.ac.in](mailto:sk.singh@iiita.ac.in)

# PROBLEMS 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 (single person to fold and put clothes away)
  - Control hazards: Pipelining of branches & other instructions stall the pipeline until the hazard “bubbles” in the pipeline
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)



# STRUCTURAL HAZARD #1: SINGLE MEMORY (1/2)



Read same memory twice in same clock cycle



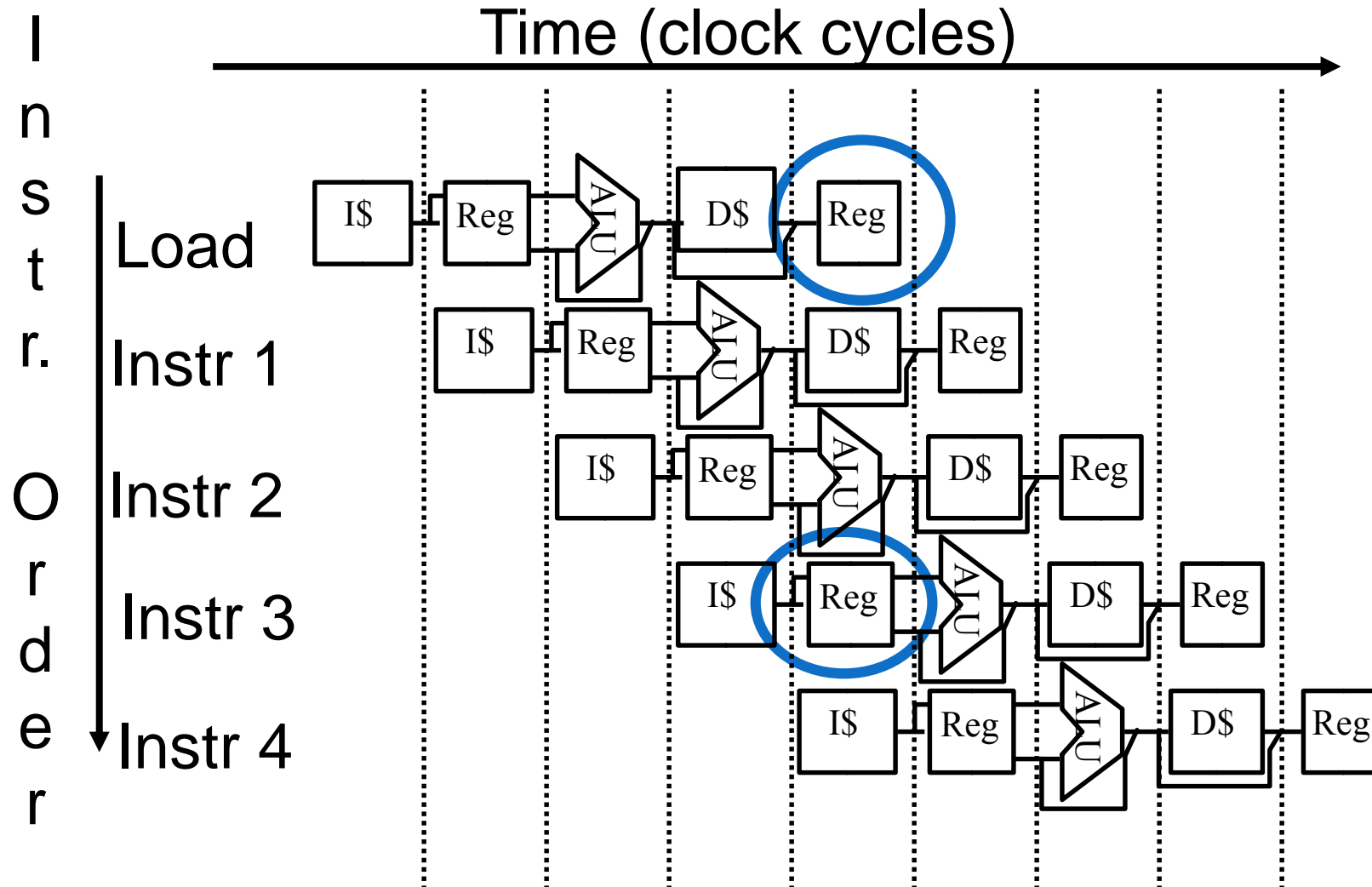
## STRUCTURAL HAZARD #1: SINGLE MEMORY (2/2)

- Solution:

- infeasible and inefficient to create second memory
- so simulate this by having two Level 1 Caches
- have both an L1 Instruction Cache and an L1 Data Cache
- need more complex hardware to control when both caches miss



## STRUCTURAL HAZARD #2: REGISTERS (1/2)



Can't read and write to registers simultaneously



## STRUCTURAL HAZARD #2: REGISTERS (2/2)

- Fact: Register access is *VERY* fast: takes less than half the time of ALU stage
- Solution: introduce convention
  - always Write to Registers during first half of each clock cycle
  - always Read from Registers during second half of each clock cycle
  - Result: can perform Read and Write during same clock cycle



## CONTROL HAZARD: BRANCHING (1/6)

- Suppose we put branch decision-making hardware in ALU stage
  - then two more instructions after the branch will *always* be fetched, whether or not the branch is taken
- Desired functionality of a branch
  - if we do not take the branch, don't waste any time and continue executing normally
  - if we take the branch, don't execute any instructions after the branch, just go to the desired label



## CONTROL HAZARD: BRANCHING (2/6)

- Initial Solution: Stall until decision is made
  - insert “no-op” instructions: those that accomplish nothing, just take time
  - Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)





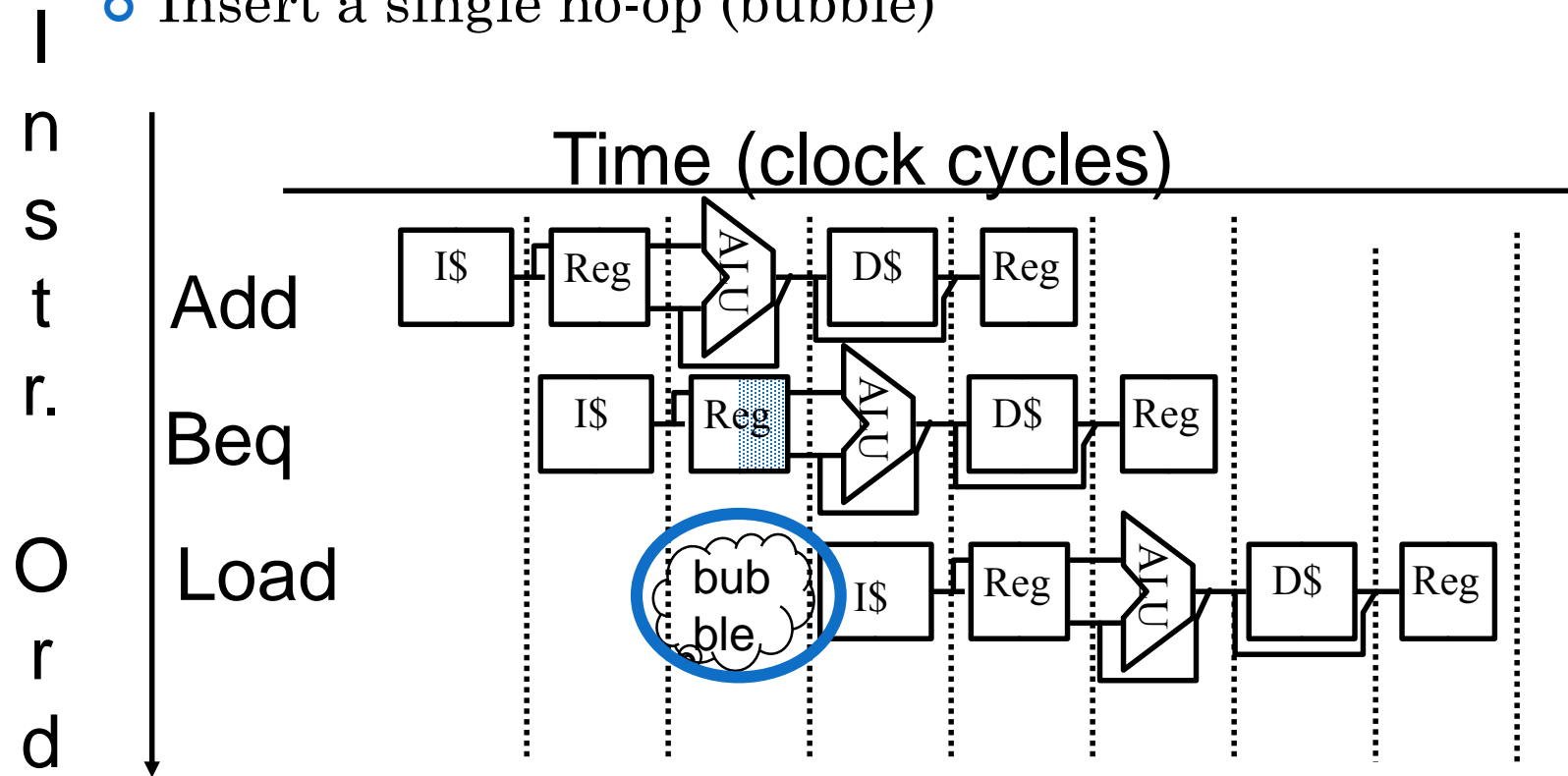
## CONTROL HAZARD: BRANCHING (3/6)

- Optimization #1:
  - move comparator up to Stage 2
  - as soon as instruction is decoded (Opcode identifies is as a branch), immediately make a decision and set the value of the PC (if necessary)
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Side Note: This means that branches are idle in Stages 3, 4 and 5.



# CONTROL HAZARD: BRANCHING (4/6)

- Insert a single no-op (bubble)



- Impact: 2 clock cycles per branch instruction  $\Rightarrow$  slow



## CONTROL HAZARD: BRANCHING (5/6)

- Optimization #2: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the **branch-delay slot**)



# CONTROL HAZARD: BRANCHING (6/6)

- Notes on Branch-Delay Slot

- Worst-Case Scenario: can always put a no-op in the branch-delay slot
- Better Case: can find an instruction preceding the branch which can be placed in the branch-delay slot without affecting flow of the program
  - re-ordering instructions is a common method of speeding up programs
  - compiler must be very smart in order to find instructions to do this
  - usually can find such an instruction at least 50% of the time



## EXAMPLE: NONDELAYED VS. DELAYED BRANCH

### Nondelayed Branch

or \$8, \$9, \$10

add \$1, \$2, \$3

sub \$4, \$5, \$6

beq \$1, \$4, Exit

xor \$10, \$1, \$11

Exit:

### Delayed Branch

add \$1, \$2, \$3

sub \$4, \$5, \$6

beq \$1, \$4, Exit

or \$8, \$9, \$10

xor \$10, \$1, \$11

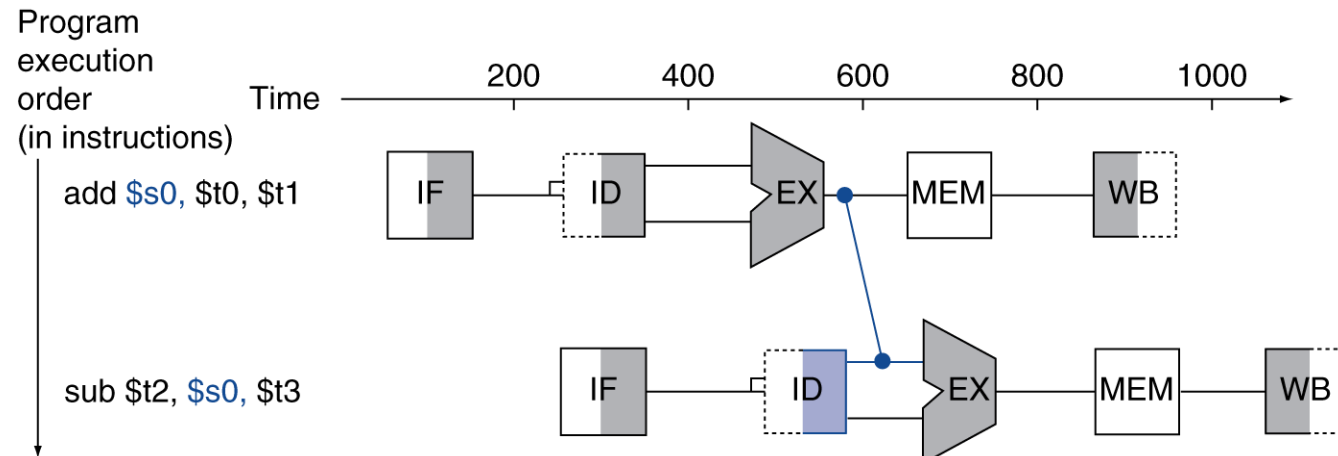
Exit:





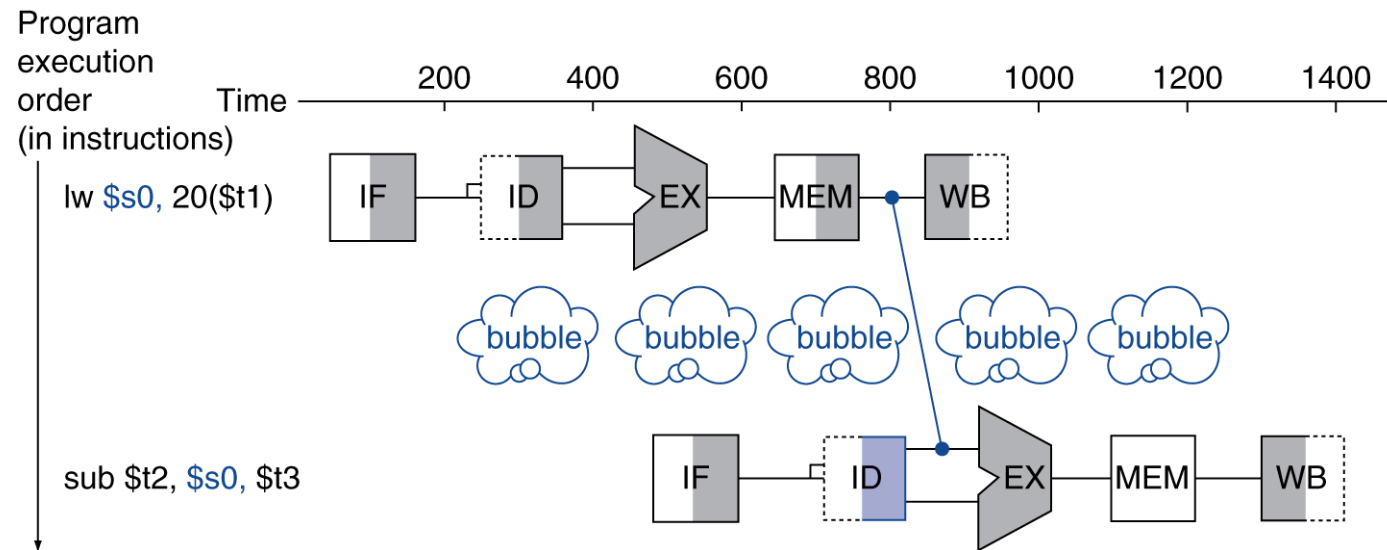
# FORWARDING (AKA BYPASSING)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



# LOAD-USE DATA HAZARD

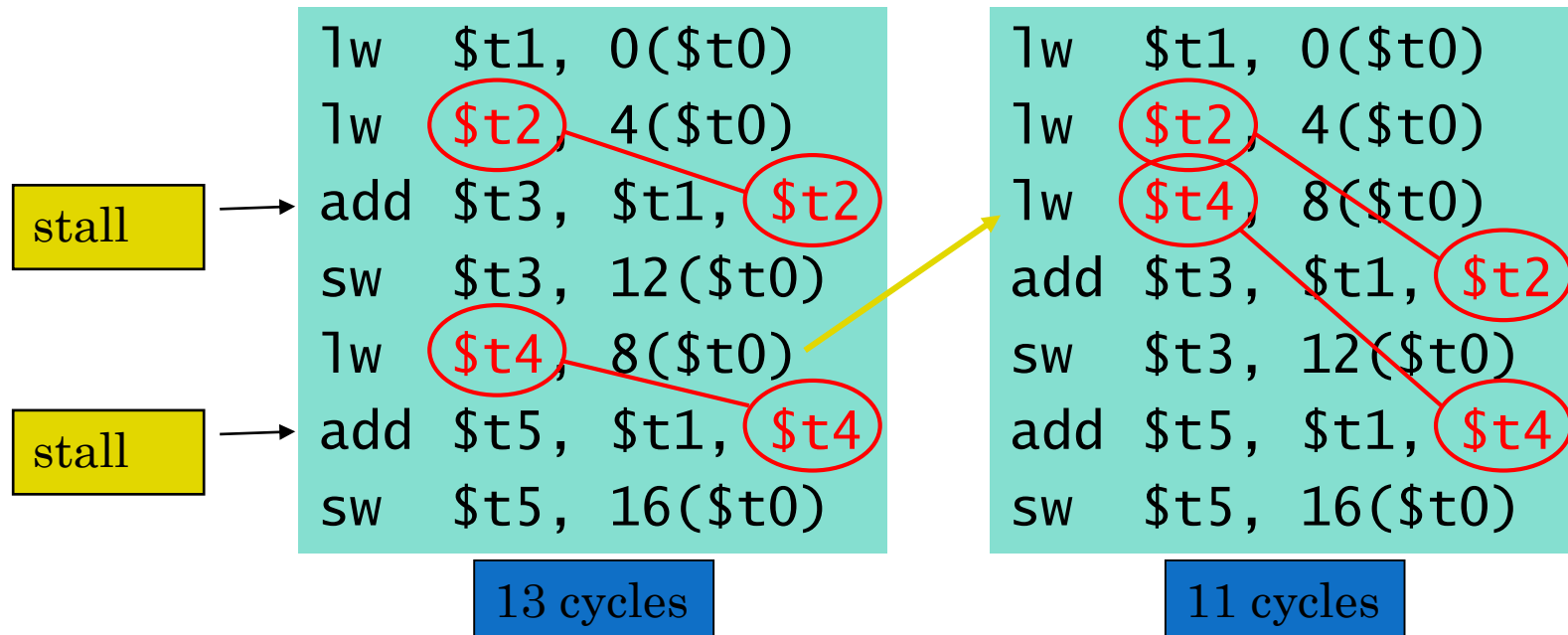
- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!





# CODE SCHEDULING TO AVOID STALLS

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



## THINGS TO REMEMBER (1/2)

- Optimal Pipeline
  - Each stage is executing part of an instruction each clock cycle.
  - One instruction finishes during each clock cycle.
  - On average, execute far more quickly.
- What makes this work?
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.

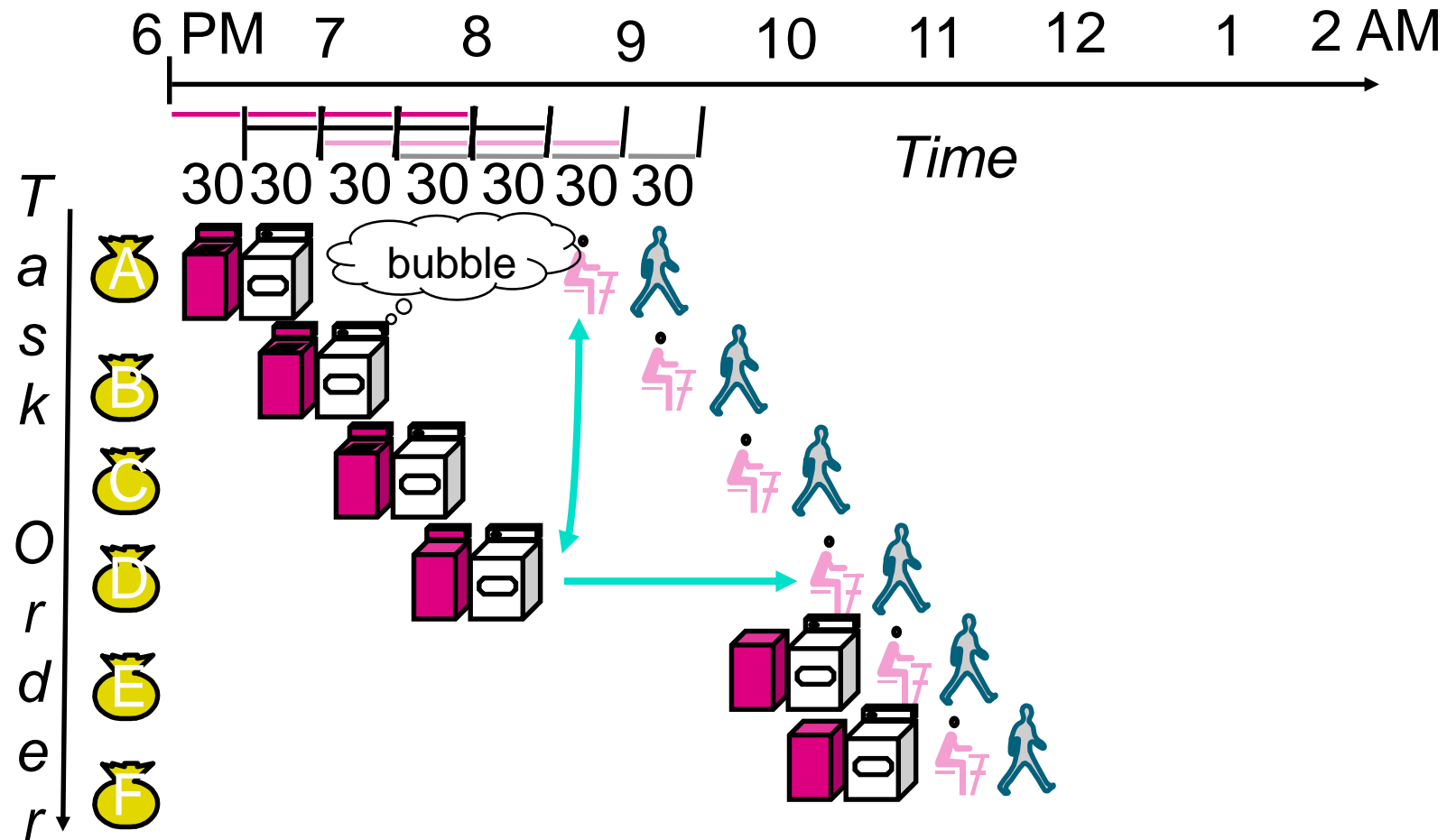


## ADVANCED PIPELINING CONCEPTS (IF TIME)

- “Out-of-order” Execution
- “Superscalar” execution
- State-of-the-Art Microprocessor



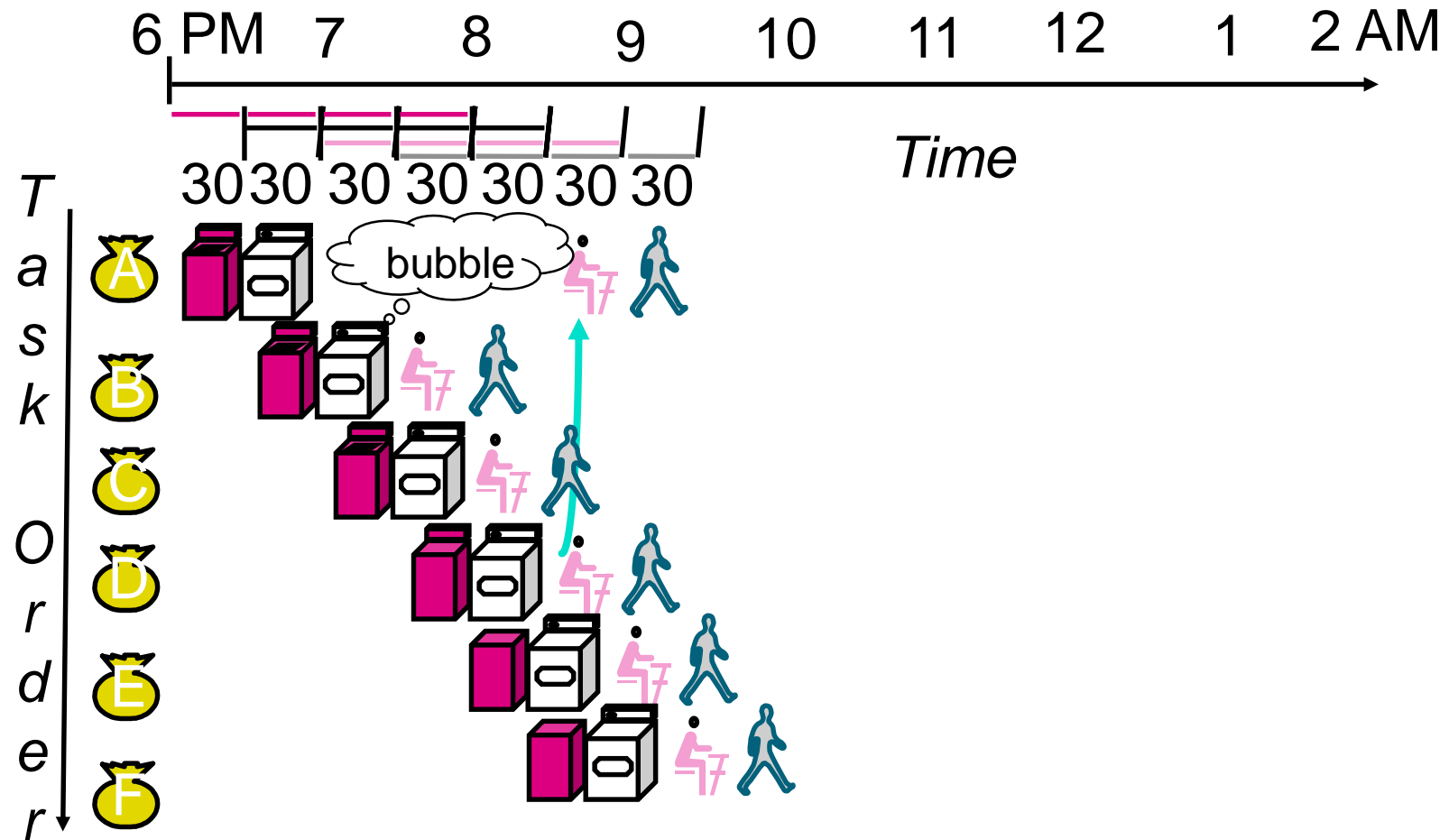
# REVIEW PIPELINE HAZARD: STALL IS DEPENDENCY



A depends on D; stall since folder tied up



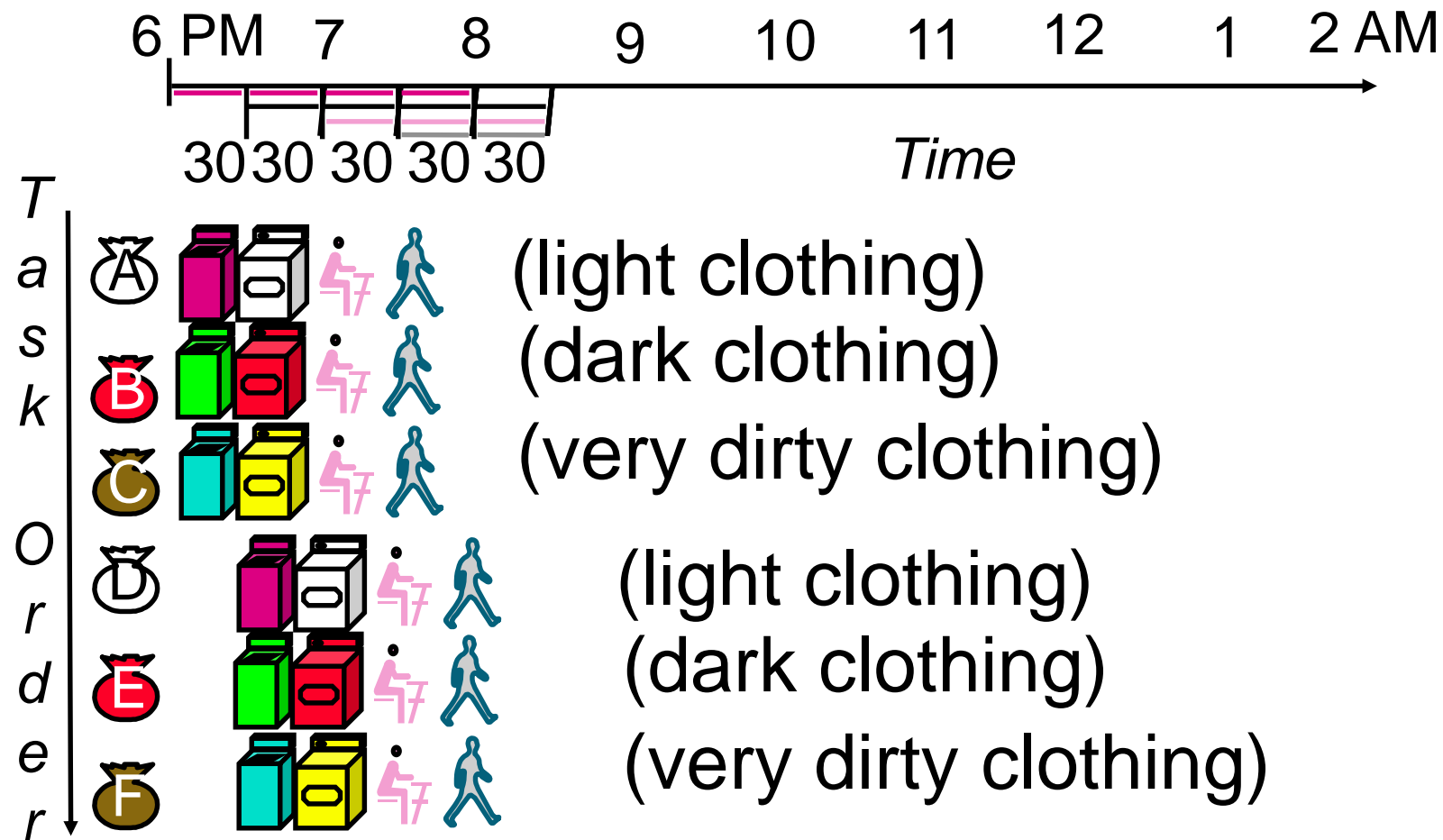
# OUT-OF-ORDER LAUNDRY: DON'T WAIT



A depends on D; rest continue; need more resources to allow out-of-order



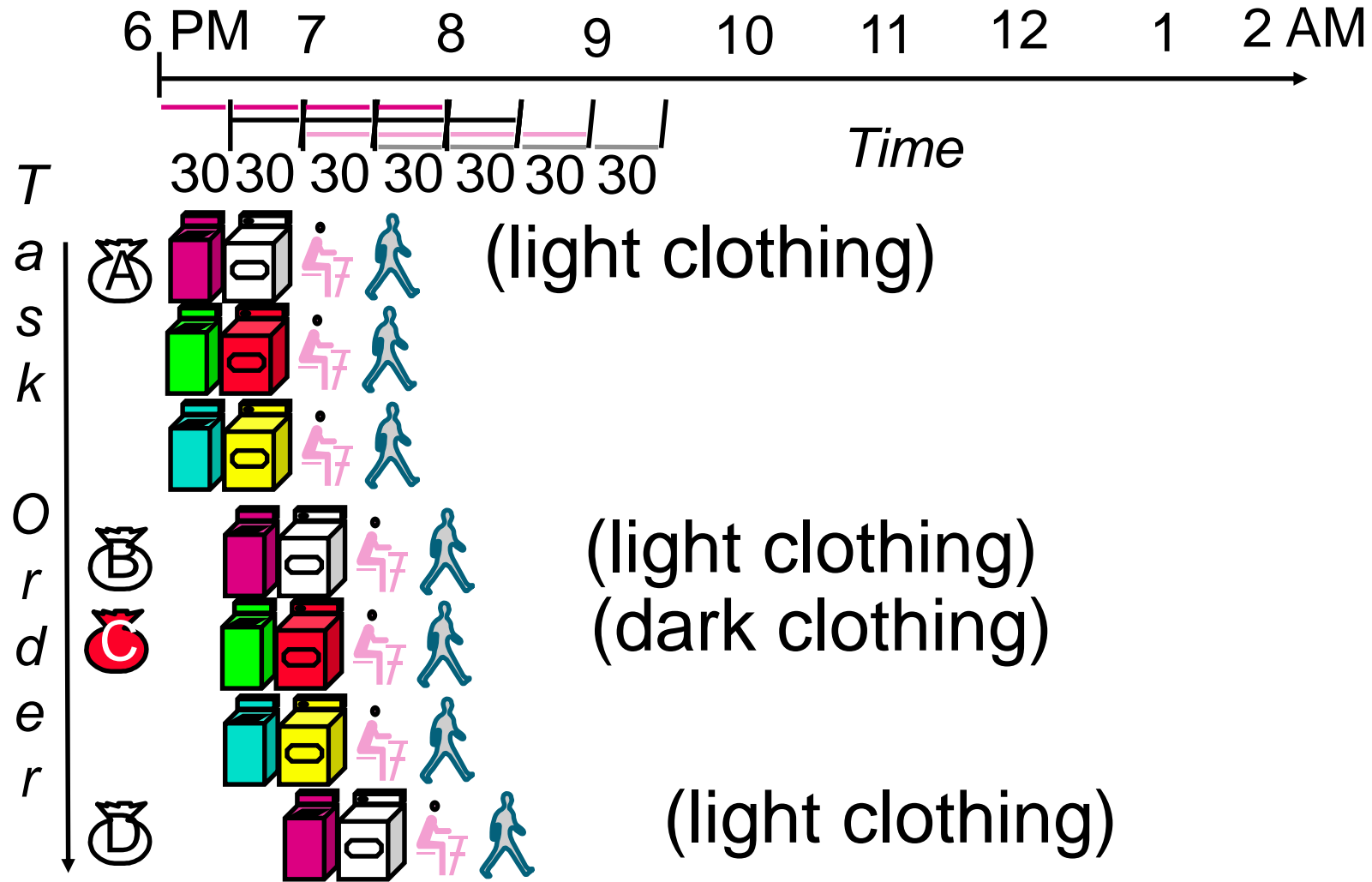
# SUPERSCALAR LAUNDRY: PARALLEL PER STAGE



More resources, HW to match mix of parallel tasks?



# SUPERSCALAR LAUNDRY: MISMATCH MIX



Task mix underutilizes extra resources



# STATE OF THE ART: COMPAQ ALPHA 21264

- Very similar instruction set to MIPS
- 1 64KB Instruction cache, 1 64 KB Data cache on chip; 16MB L2 cache off chip
- Clock cycle = 1.5 nanoseconds, or 667 MHz clock rate
- Superscalar: fetch up to 6 instructions /clock cycle, retires up to 4 instruction/clock cycle
- Execution out-of-order
- 15 million transistors, 90 watts!





## THINGS TO REMEMBER (2/2)

- Pipelining a Big Idea: widely used concept
- What makes it less than perfect?
  - Structural hazards: suppose we had only one cache?
    - ⇒ Need more HW resources
  - Control hazards: need to worry about branch instructions?
    - ⇒ **D**elayed branch
  - Data hazards: an instruction depends on a previous instruction?



Thank you