Introduction to Pipelined Execution

Review (1/3)

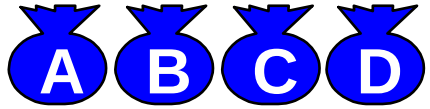
- Datapath is the hardware that performs operations necessary to execute programs.
- Control instructs datapath on what to do next.
- Datapath needs:
  - access to storage (general purpose registers and memory)
  - computational ability (ALU)
  - helper hardware (local registers and PC)

Outline

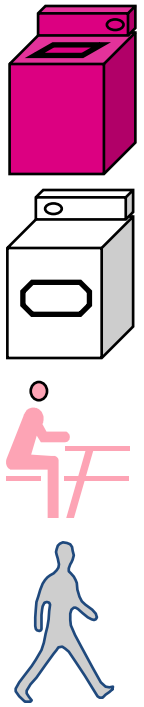
- Pipelining Analogy
- Pipelining Instruction Execution
- Hazards

With or without your laundry

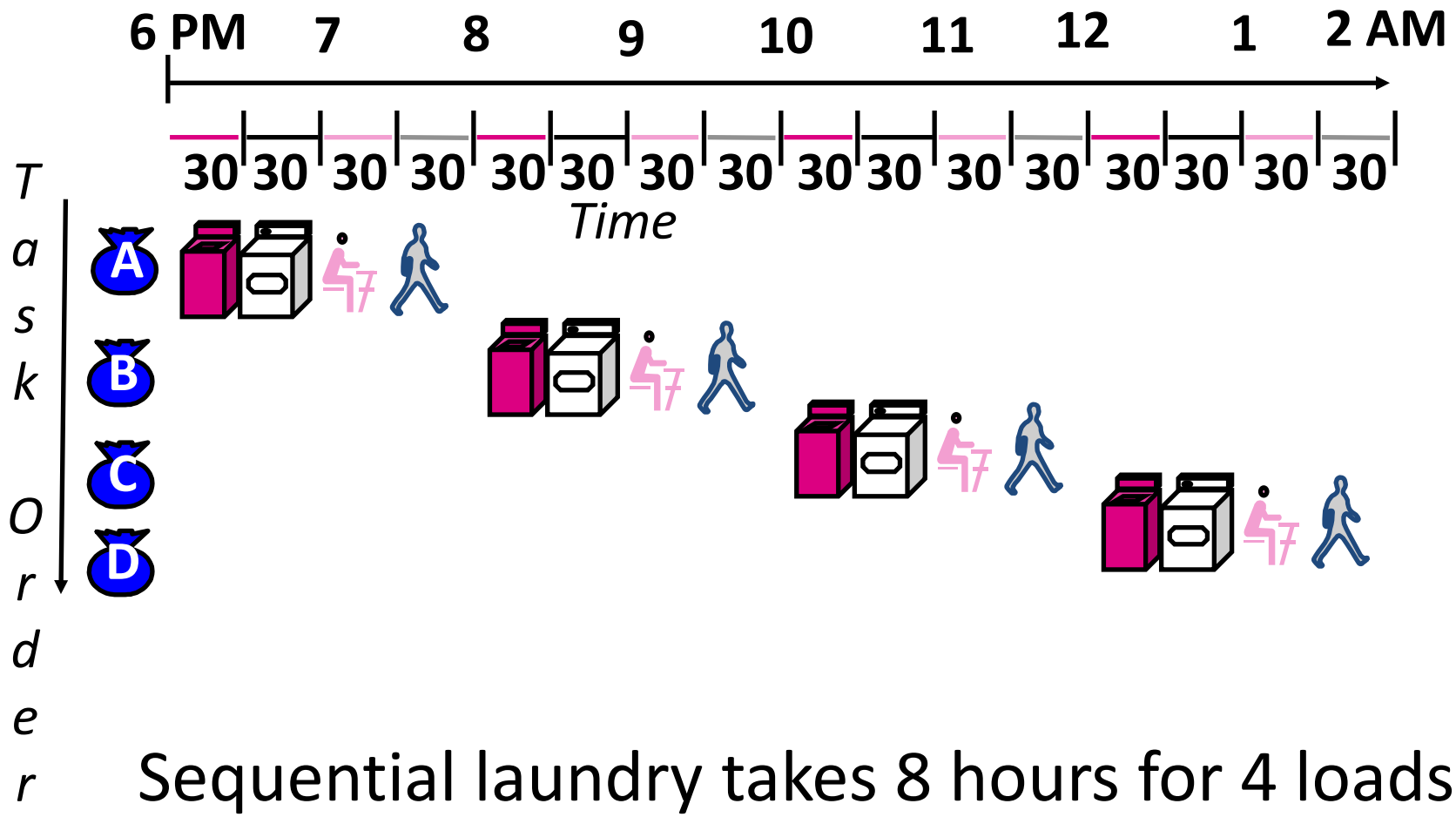
- Bono, The Edge, Adam Clayton, and Larry Mullen Jr., each have one load of clothes to wash, dry, fold, and put away



- **Washer takes 30 minutes**
- **Dryer takes 30 minutes**
- **“Folder” takes 30 minutes**
- **“Stasher” takes 30 minutes to put clothes into drawers**

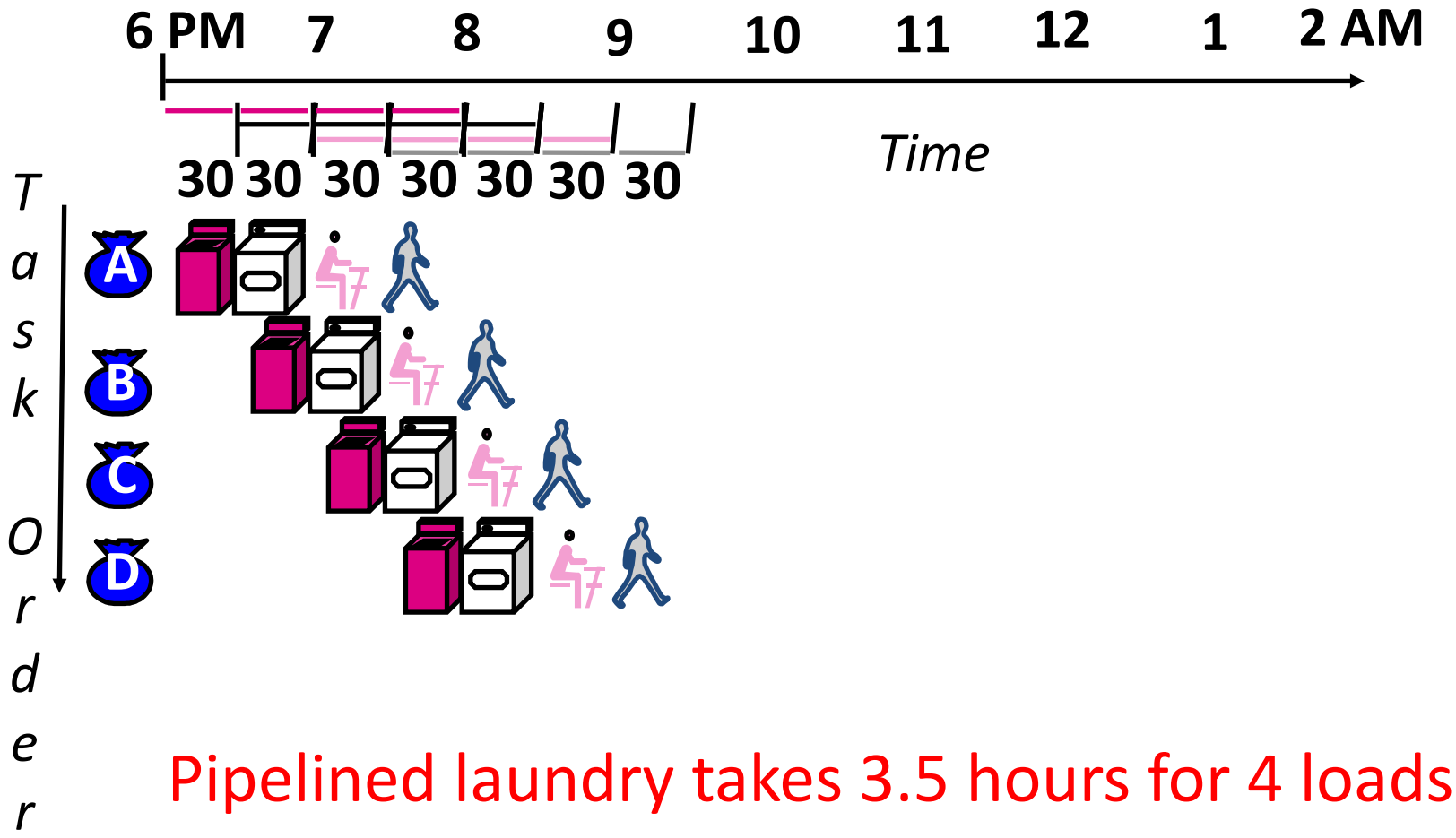


Sequential Laundry



Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

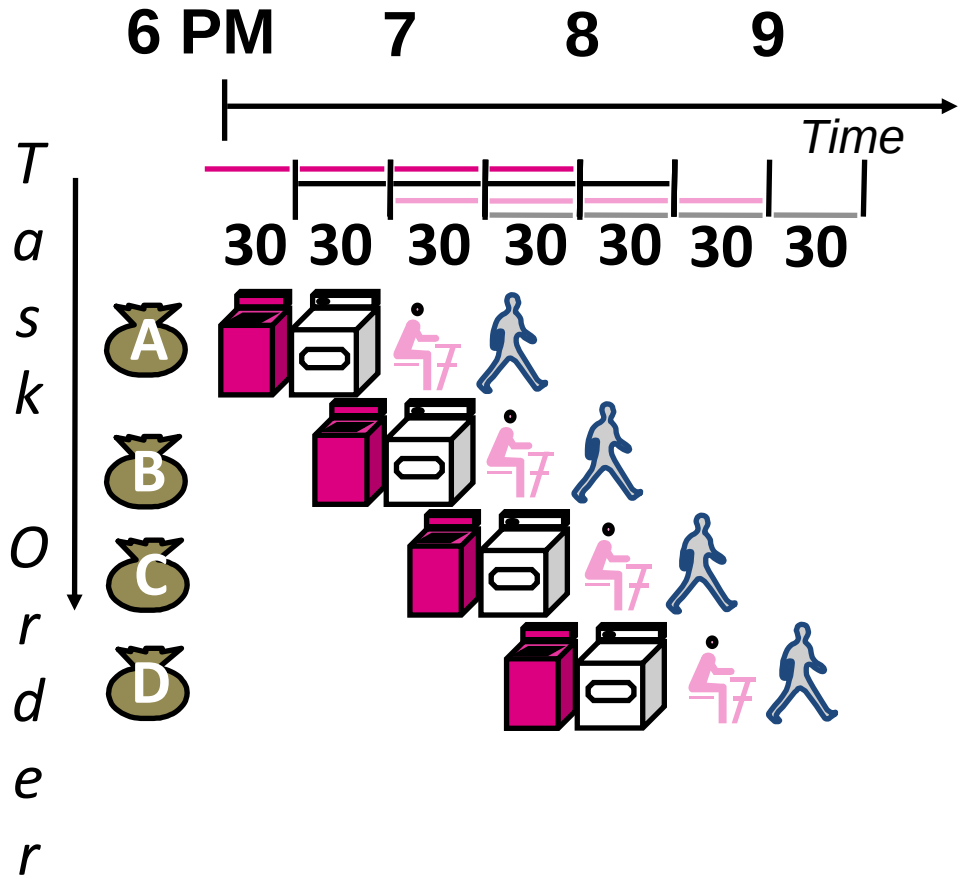


Pipelined laundry takes 3.5 hours for 4 loads!

General Definitions

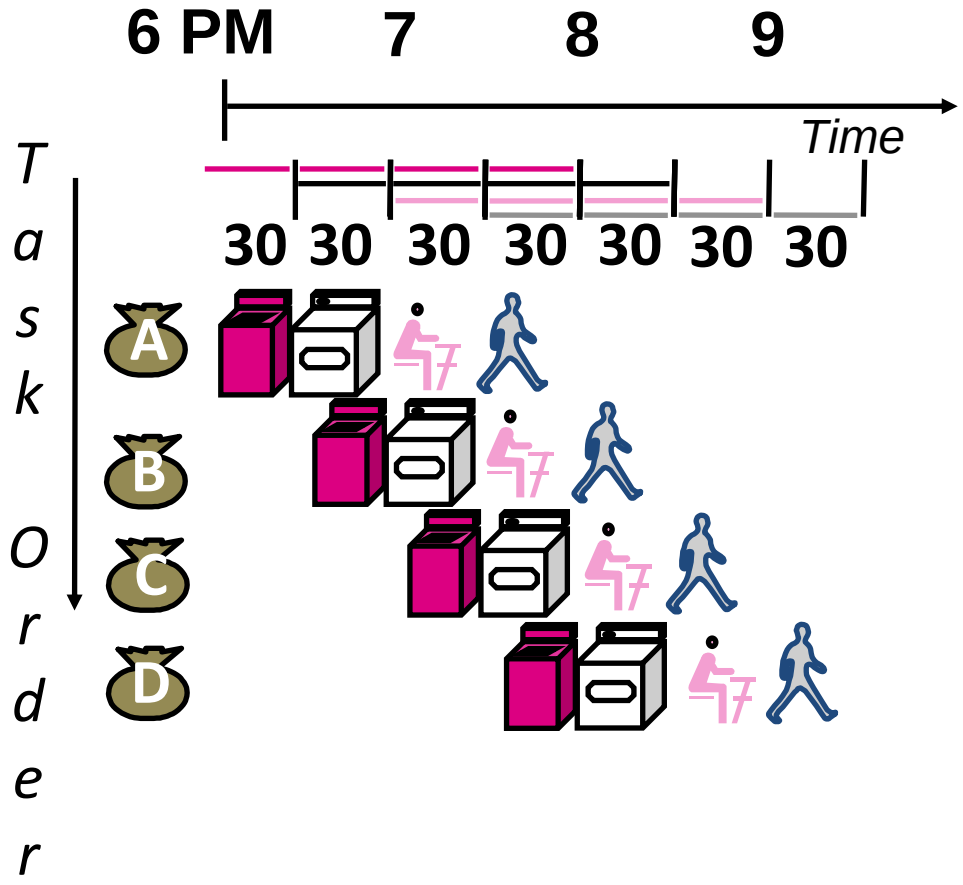
- **Latency**
  - Time to completely execute a certain task
    - For example, time to read a sector from disk is disk access time or disk latency
- **Throughput**
  - Amount of work that can be done over a period of time

Pipelining Lessons (1/2)



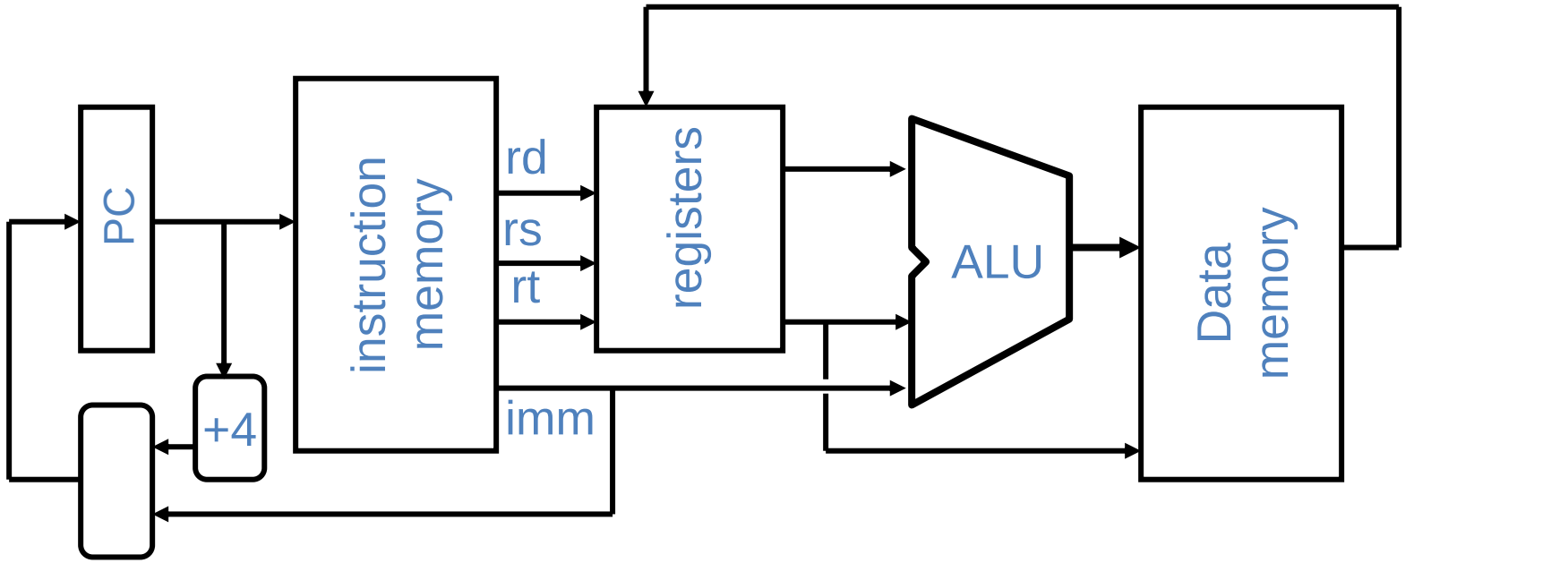
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup:
  2.3X v. 4X in this example

Pipelining Lessons (2/2)



- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by ***slowest*** pipeline stage
- Unbalanced lengths of pipe stages also reduces speedup

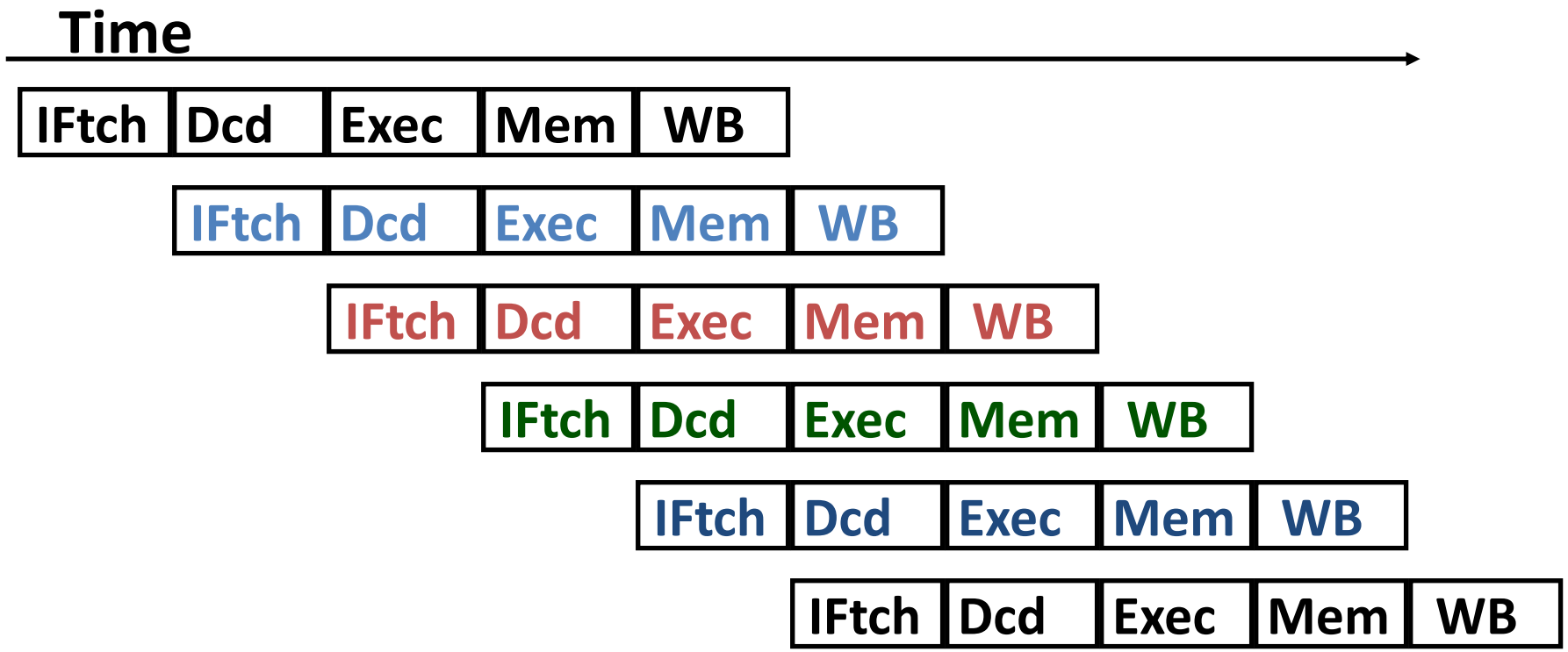
Review Datapath



Steps in Executing MIPS

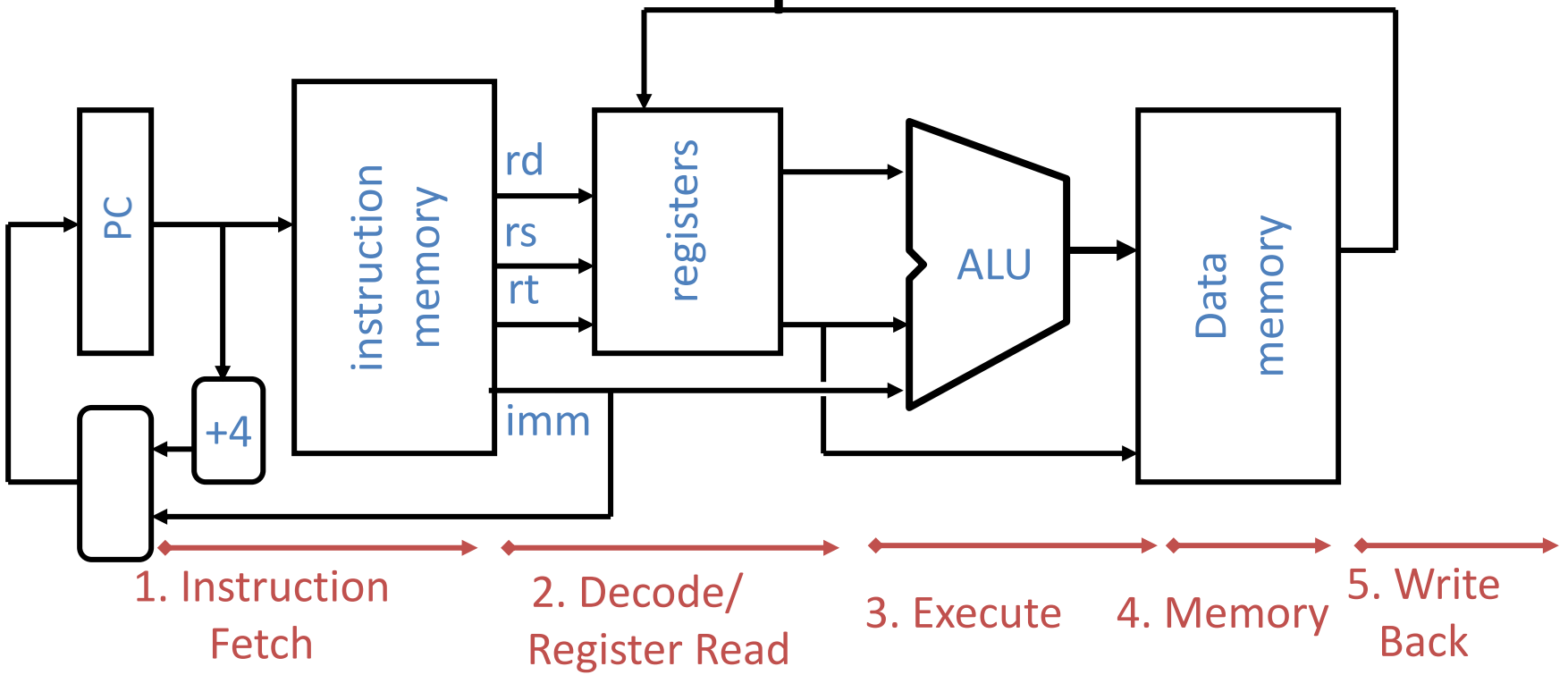
1) IFetch: Fetch Instruction, Increment PC
2) Decode: Instruction, Read Registers
3) Execute:
   - Mem-ref: Calculate Address
   - Arith-log: Perform Operation
4) Memory:
   - Load: Read Data from Memory
   - Store: Write Data to Memory
5) Write Back: Write Data to Register

Pipelined Execution Representation

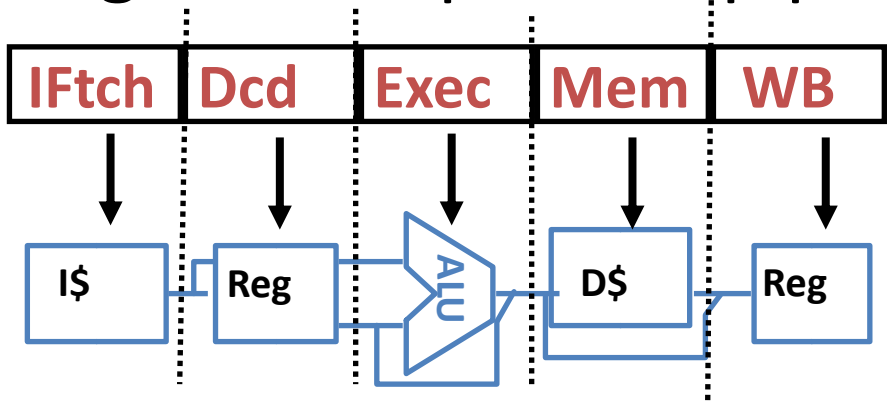


- Every instruction must take same number of steps, also called pipeline "***stages***", so some will go idle sometimes

Review: Datapath for MIPS

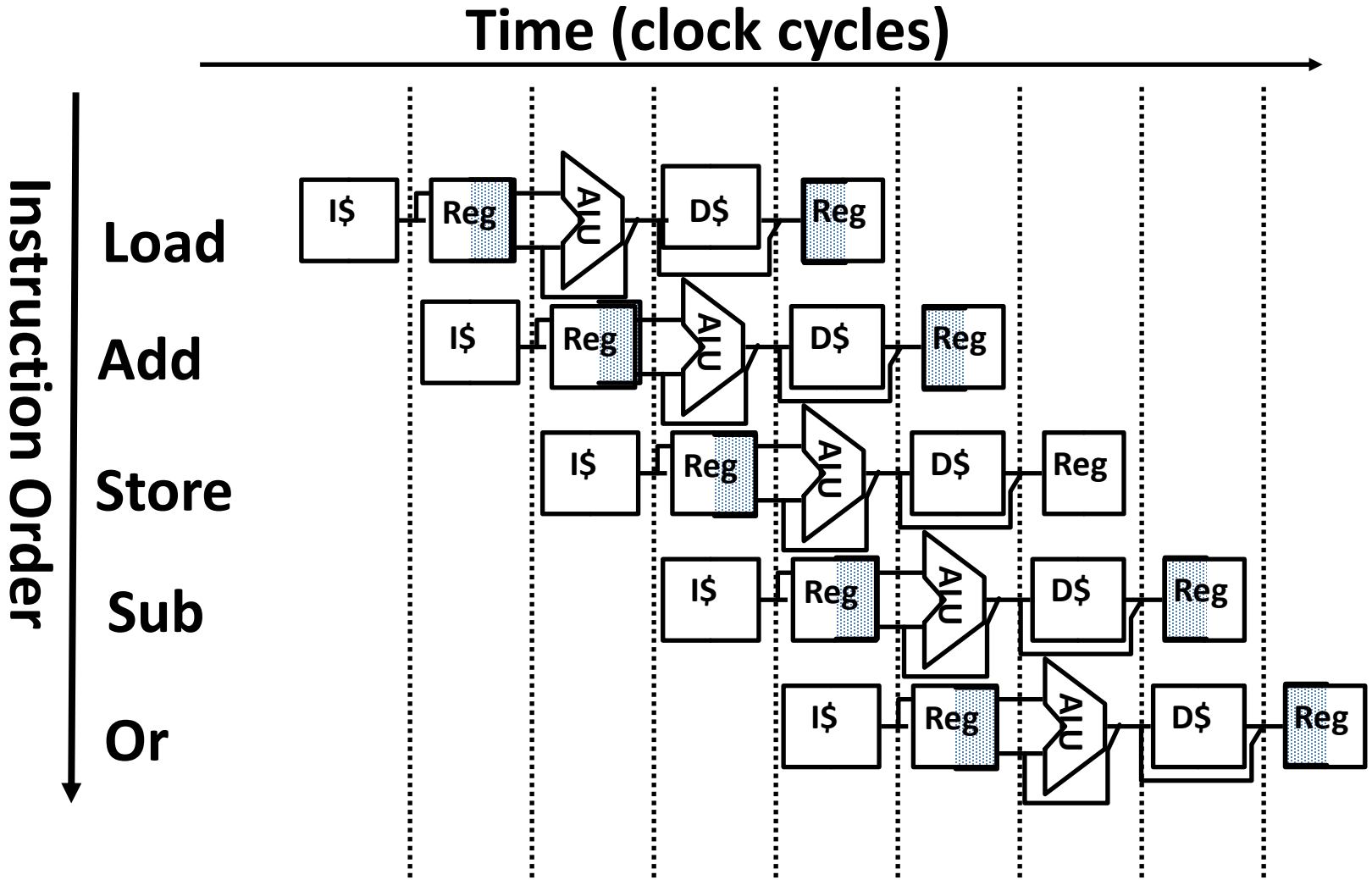


- Use datapath figure to represent pipeline



Graphical Pipeline Representation

(In Reg, right half highlight read, left half write (Hint: See Slide#21))



Example

- Suppose 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write; what is the ***instruction execution rate***?
- Non-pipelined Execution, consider LW and ADD:

  LW: IF + Read Reg + ALU + Memory + Write Reg

  = 2 + 1 + 2 + 2 + 1 = 8 ns

  ADD: IF + Read Reg + ALU + Write Reg

  = 2 + 1 + 2 + 1 = 6 ns

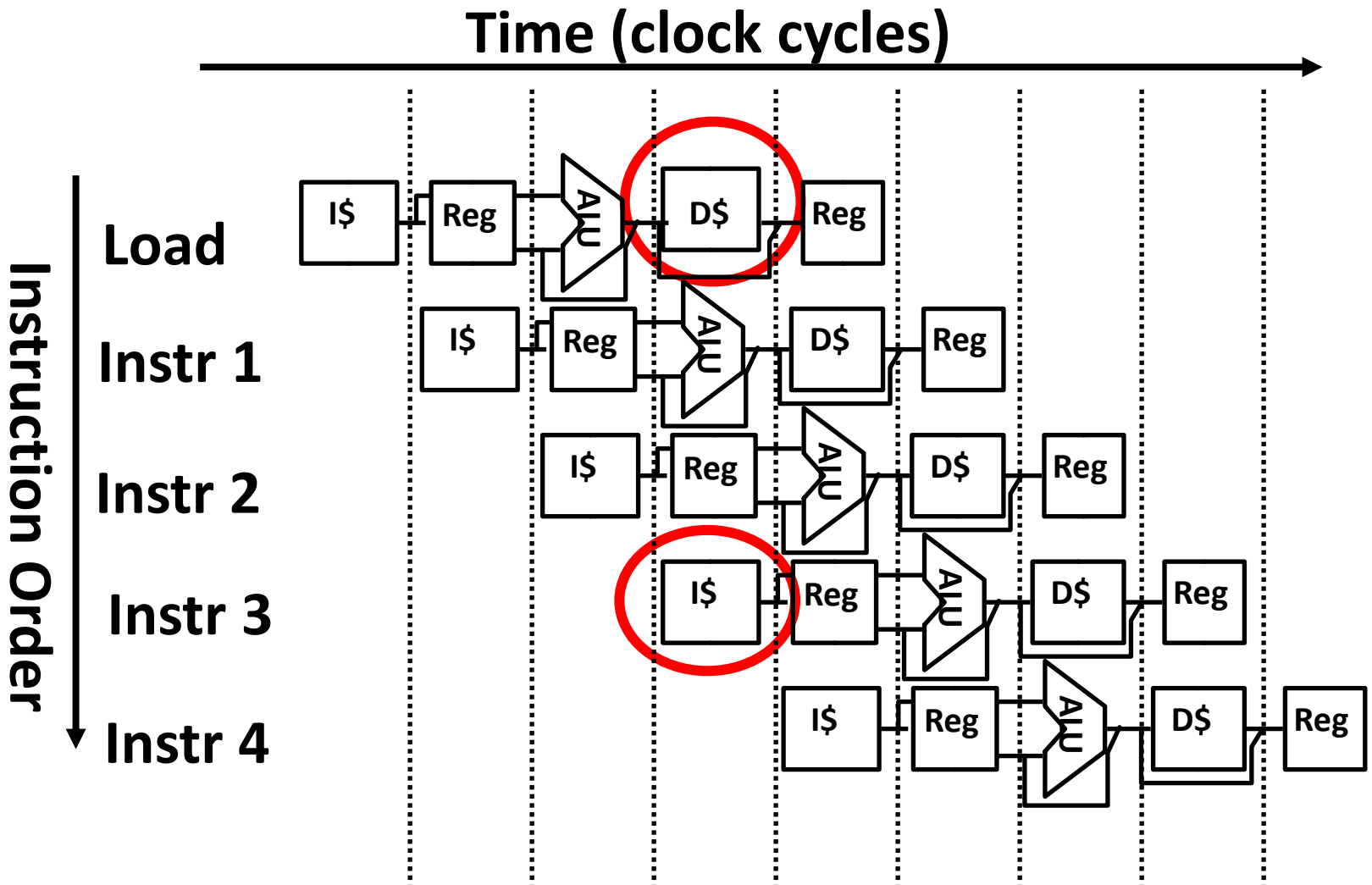
- Pipelined Execution:

  Max(IF,Read Reg,ALU,Memory,Write Reg) = 2 ns

Problems for Pipelined Processors

- There exist ***Hazards*** that prevent the next instruction from executing during its designated clock cycle
  - ***Structural hazards***: HW cannot support this combination of instructions
  - ***Control hazards***: Pipelining of branches & other instructions can ***stall*** the pipeline until the hazard
  - ***Data hazards***: Instruction depends on result of prior instruction still in the pipeline

Structural Hazard #1: Single Memory (1/2)

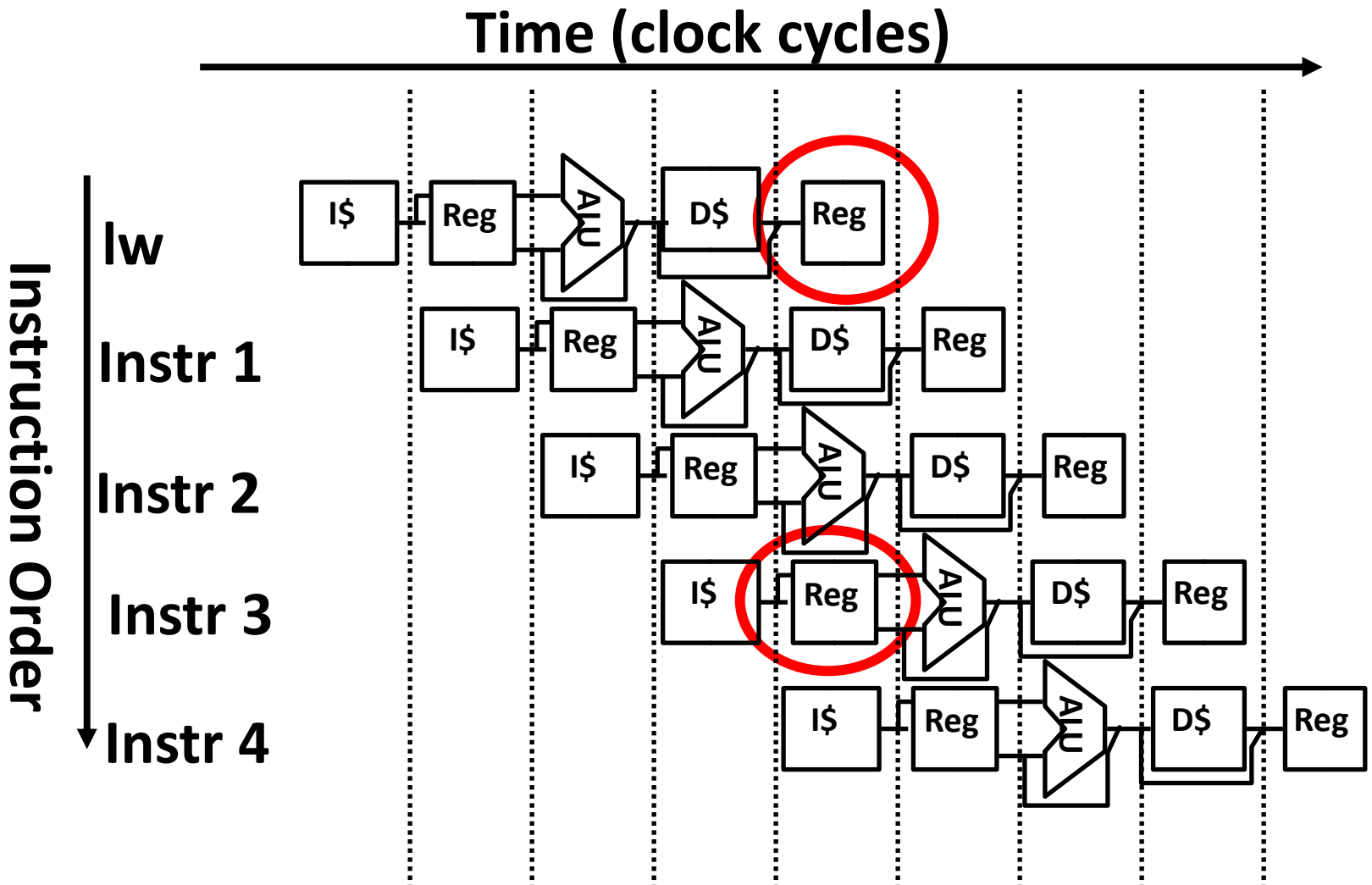


Read same memory twice in same clock cycle

Structural Hazard #1: Single Memory (2/2)

- Solution:
  - Infeasible and inefficient to create an independent second memory, but can simulate this by having two Level 1 Caches
  - Use 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)

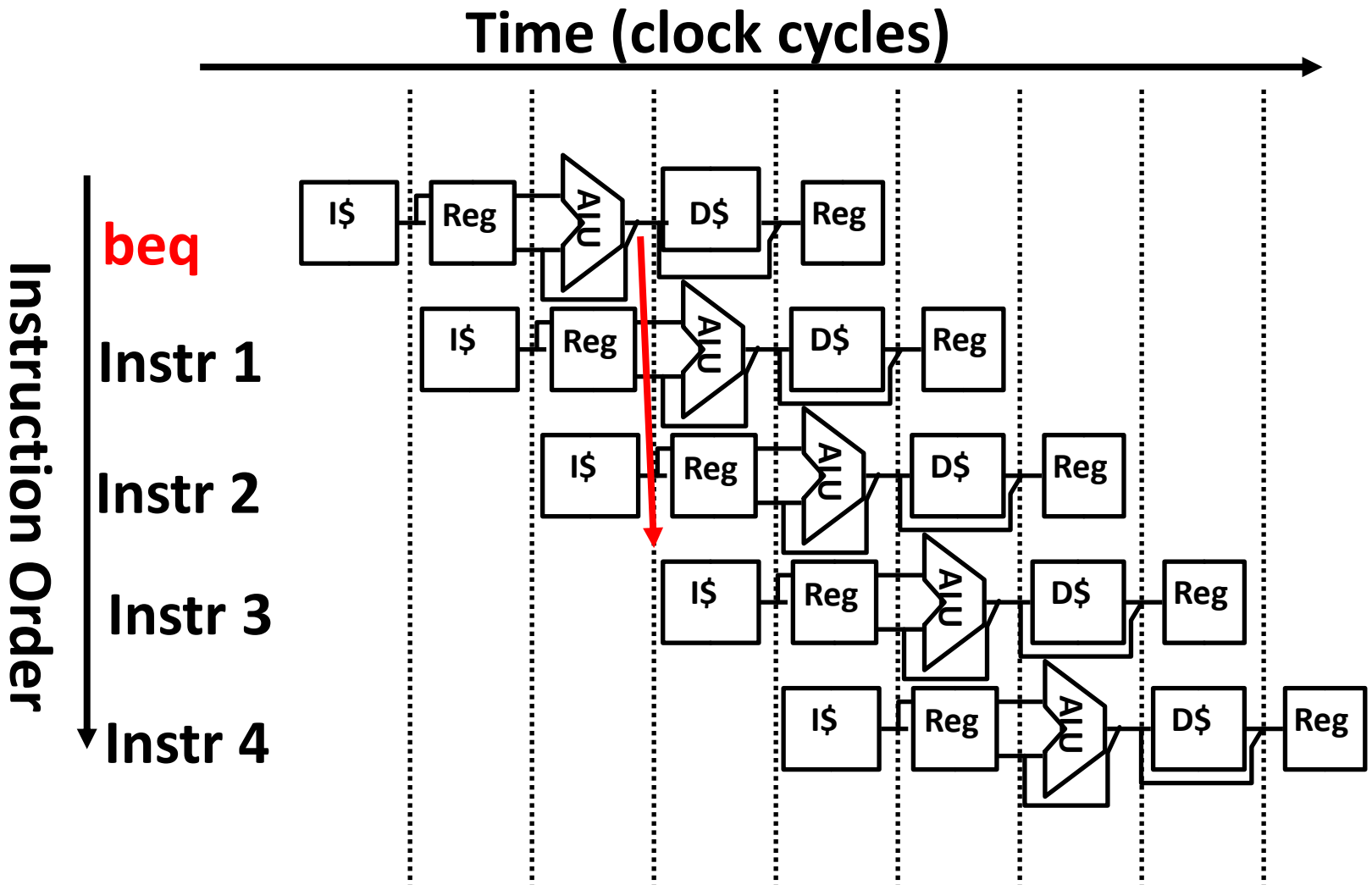


Can’t read and write to registers simultaneously

Structural Hazard #2: Registers (2/2)

- Fact: Register access is ***VERY*** fast: takes less than half the time of ALU stage
- Solution: modify clocking 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 (1/7)



Where do we do the compare for the branch?

Control Hazard: Branching (2/7)

- We naively put branch decision-making hardware in ALU stage
  - therefore ***two*** more instructions after the branch will *always* be fetched, whether or not the branch is taken
- Consider: 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 (3/7)

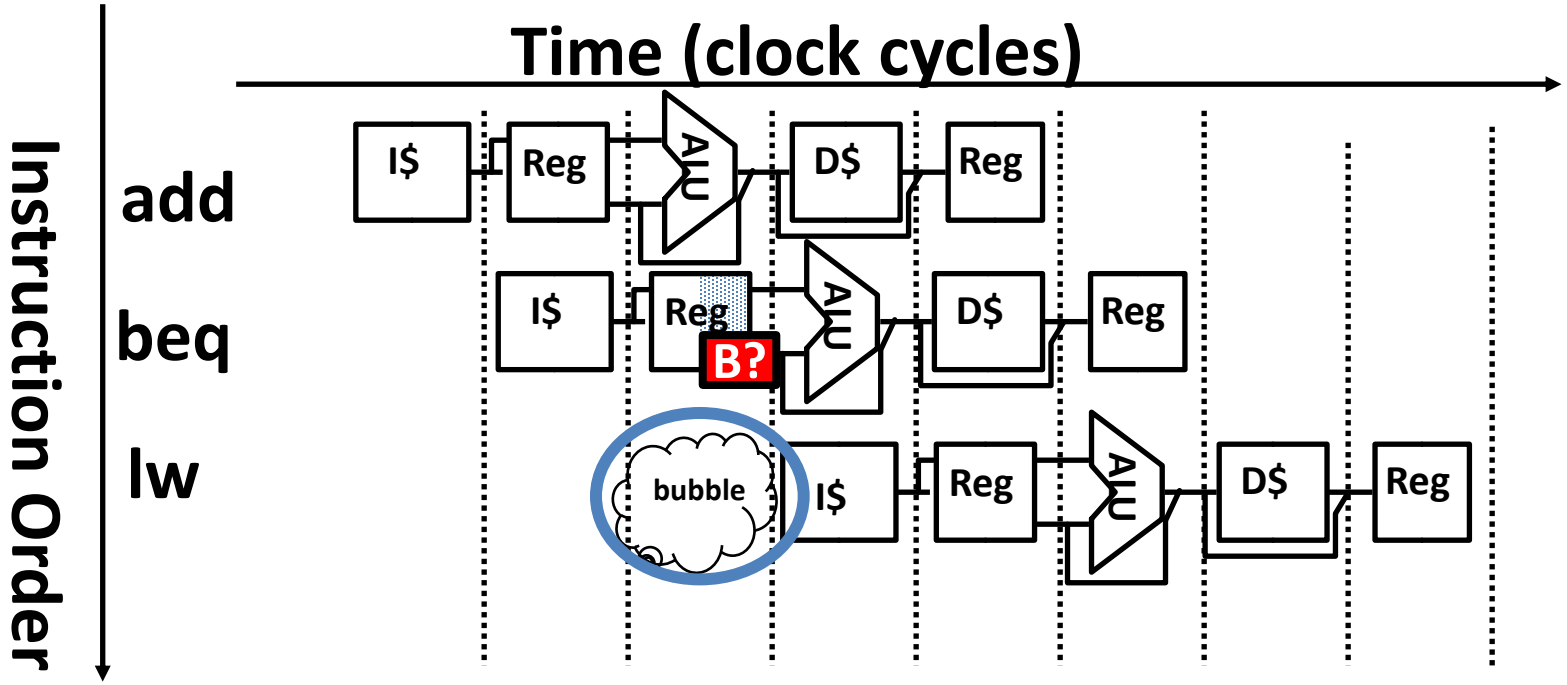
- Initial Solution: ***Stall*** until decision is made
  - insert “no-op” instructions that accomplish nothing, just take time
  - Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)

Control Hazard: Branching (4/7)

- Optimization #1:
  - Move asynchronous 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 (5/7)

Insert a single no-op (bubble)



Impact: 2 clock cycles per branch instruction, slow

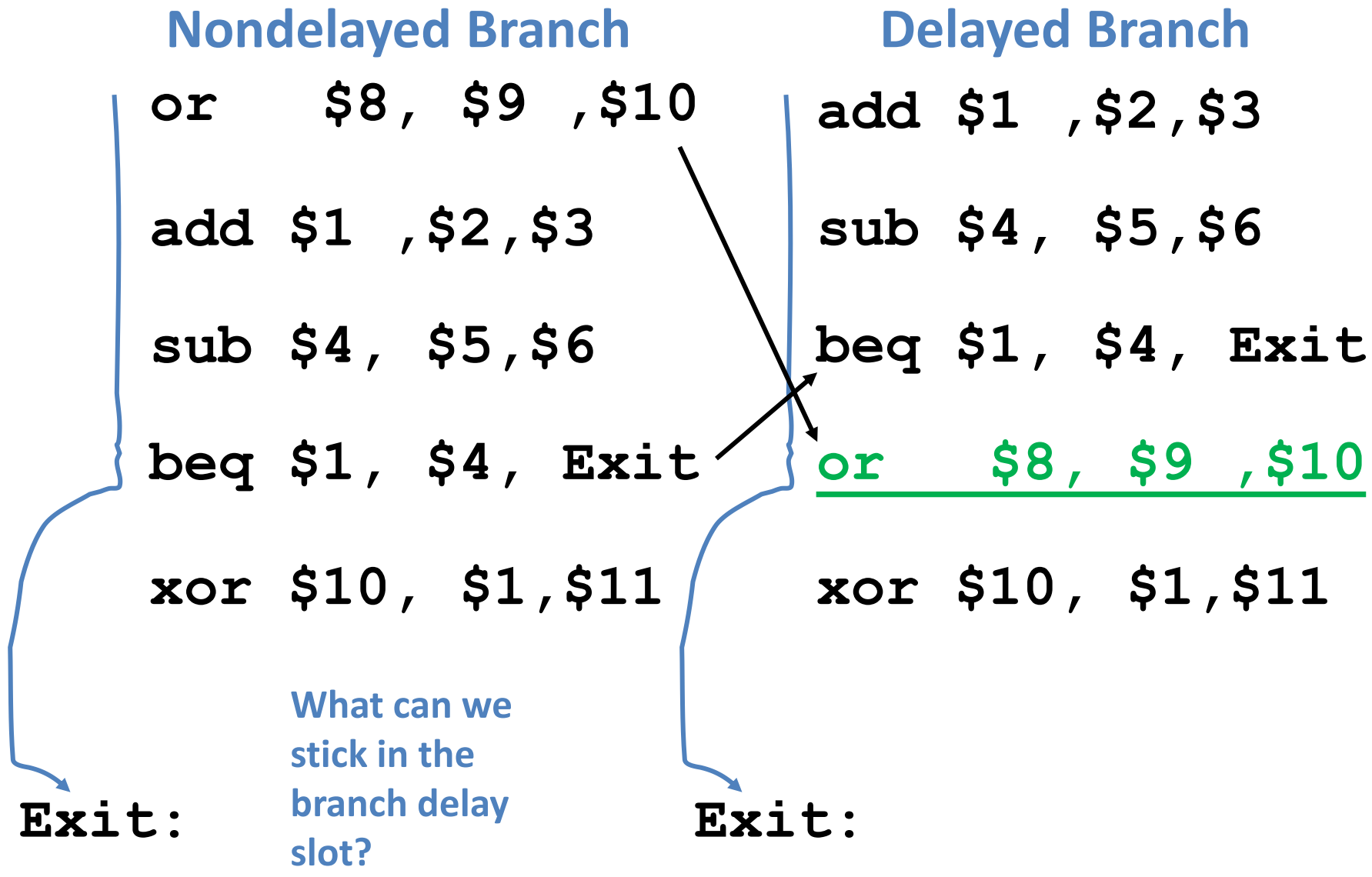
Control Hazard: Branching (6/7)

- 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
  - This is called the ***branch-delay slot***

Control Hazard: Branching (7/7)

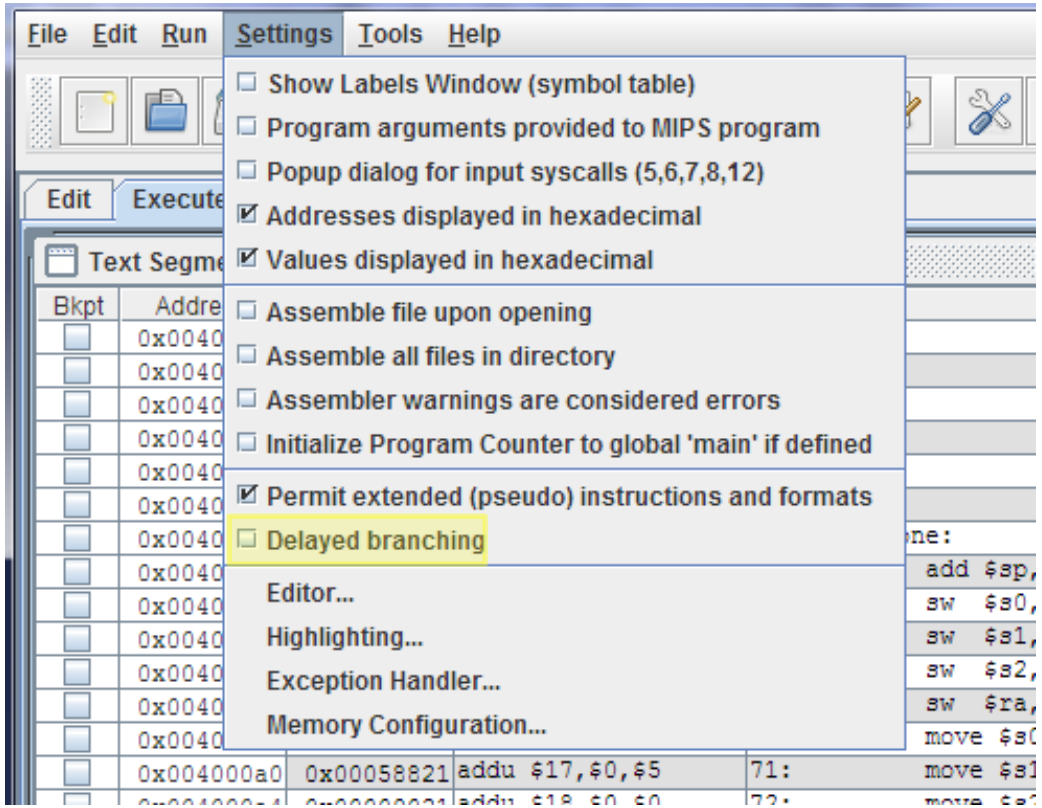
- Notes on ***Branch-Delay Slot***
  - Worst-Case Scenario: can always put a no-op in the branch-delay slot
  - Better Case: can find an instruction preceding the branch which can be placed in the branch-delay slot without affecting flow of the program
    - ***Re-ordering instructions*** is a common 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**
    - Jumps also have a delay slot!

Example: Nondelayed vs. Delayed Branch



Simulating Hazards

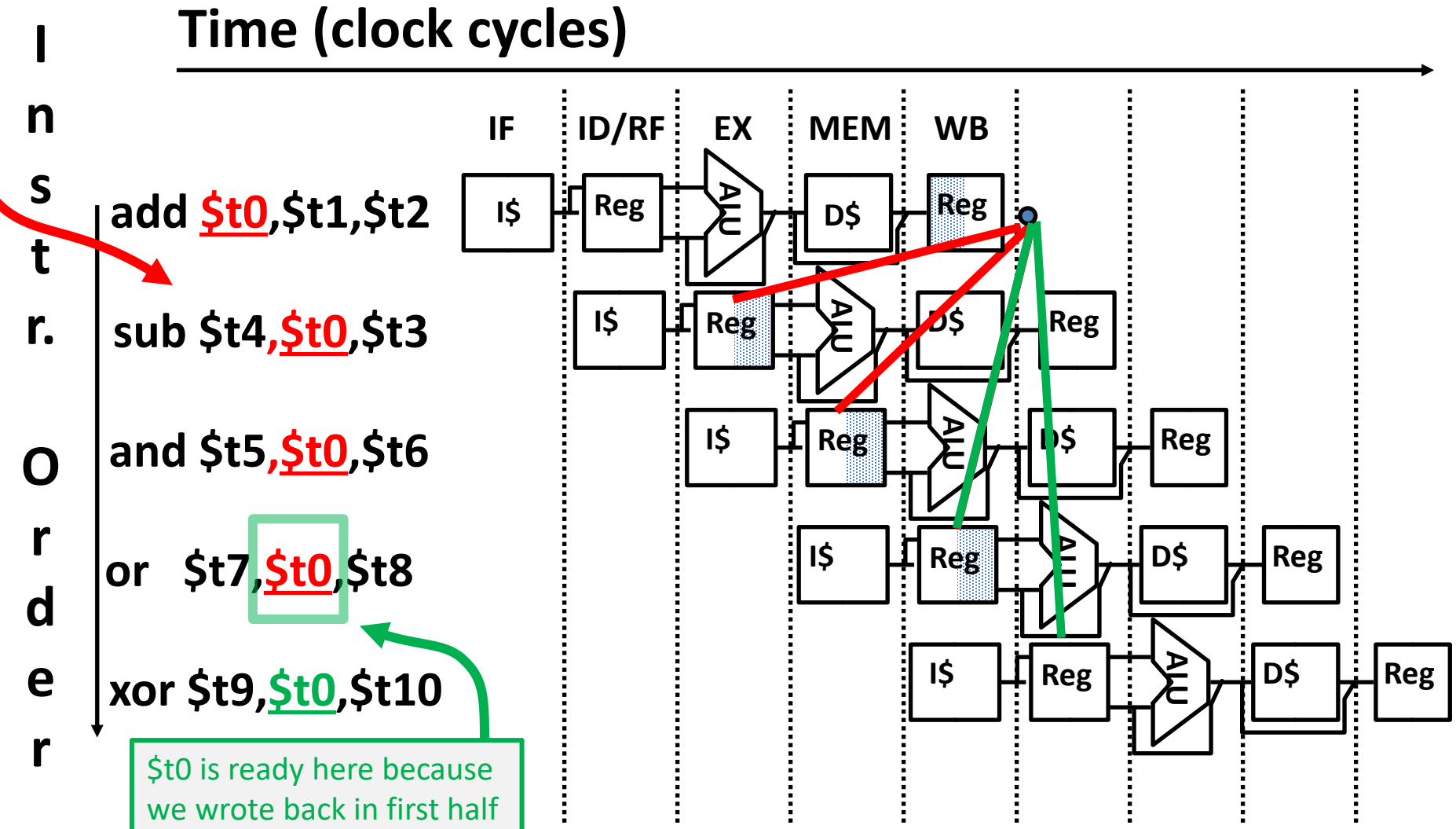
- MARS can simulate delayed branches



Data Hazards

Sub will try to read $t0 before the result is written!

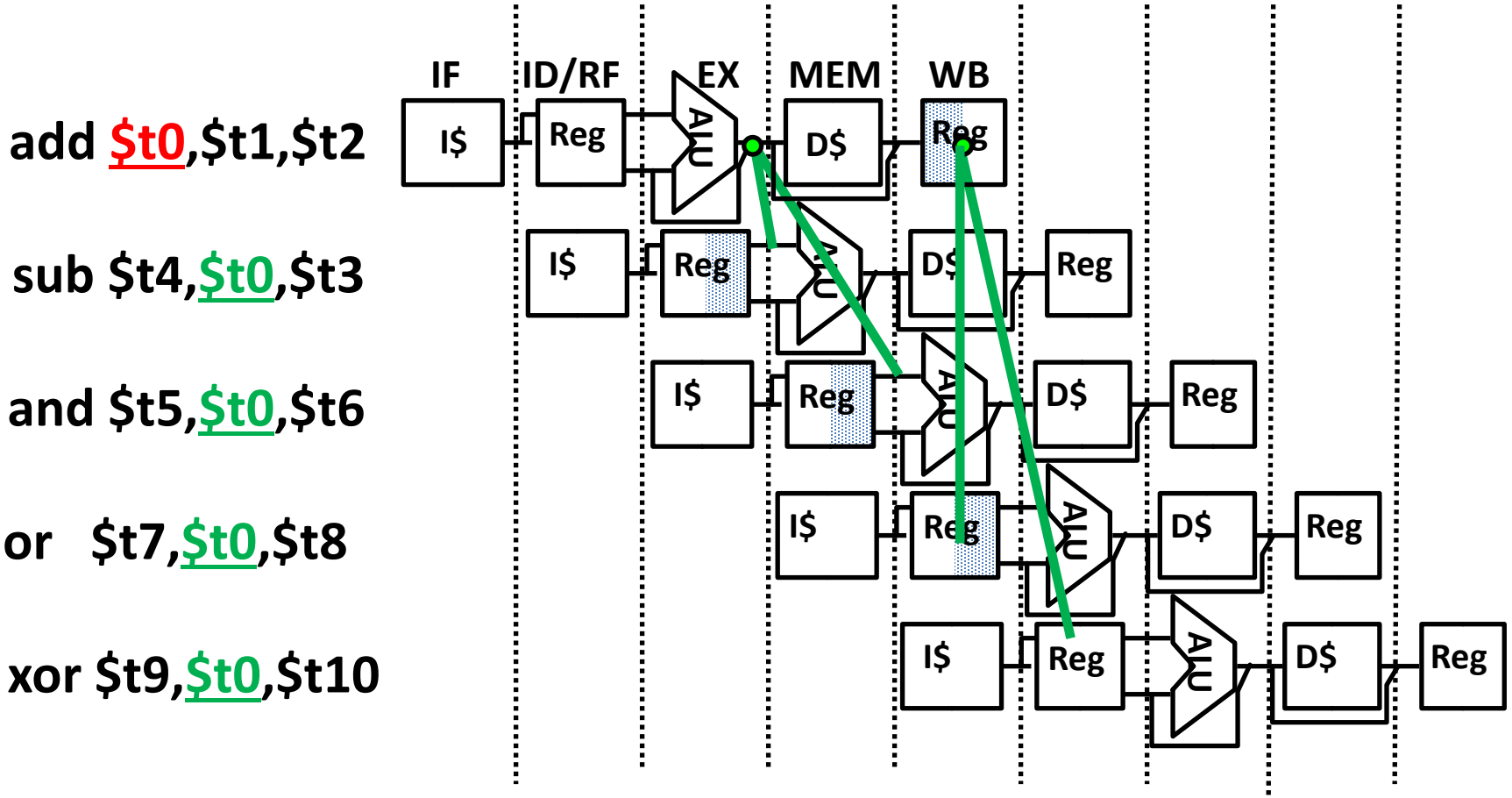
Dependencies *backwards in time* are hazards



$t0 is ready here because we wrote back in first half of clock cycle

Data Hazard Solution: Forwarding

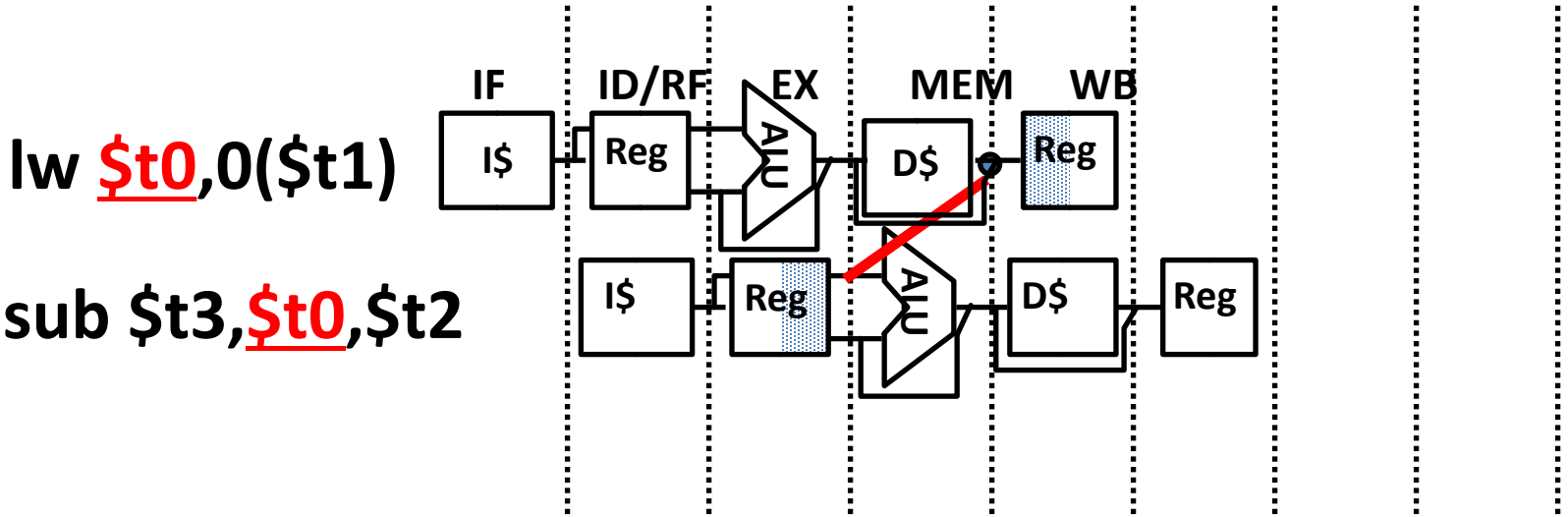
Forward result from one stage to another



"or" hazard solved by register hardware

Data Hazard: Loads (1/3)

- Dependencies backwards in time are hazards



- Can't solve with forwarding
- Must ***stall*** (i.e., inserting nop) instruction dependent on load, then forward (more hardware)

Data Hazard: Loads (2/3)

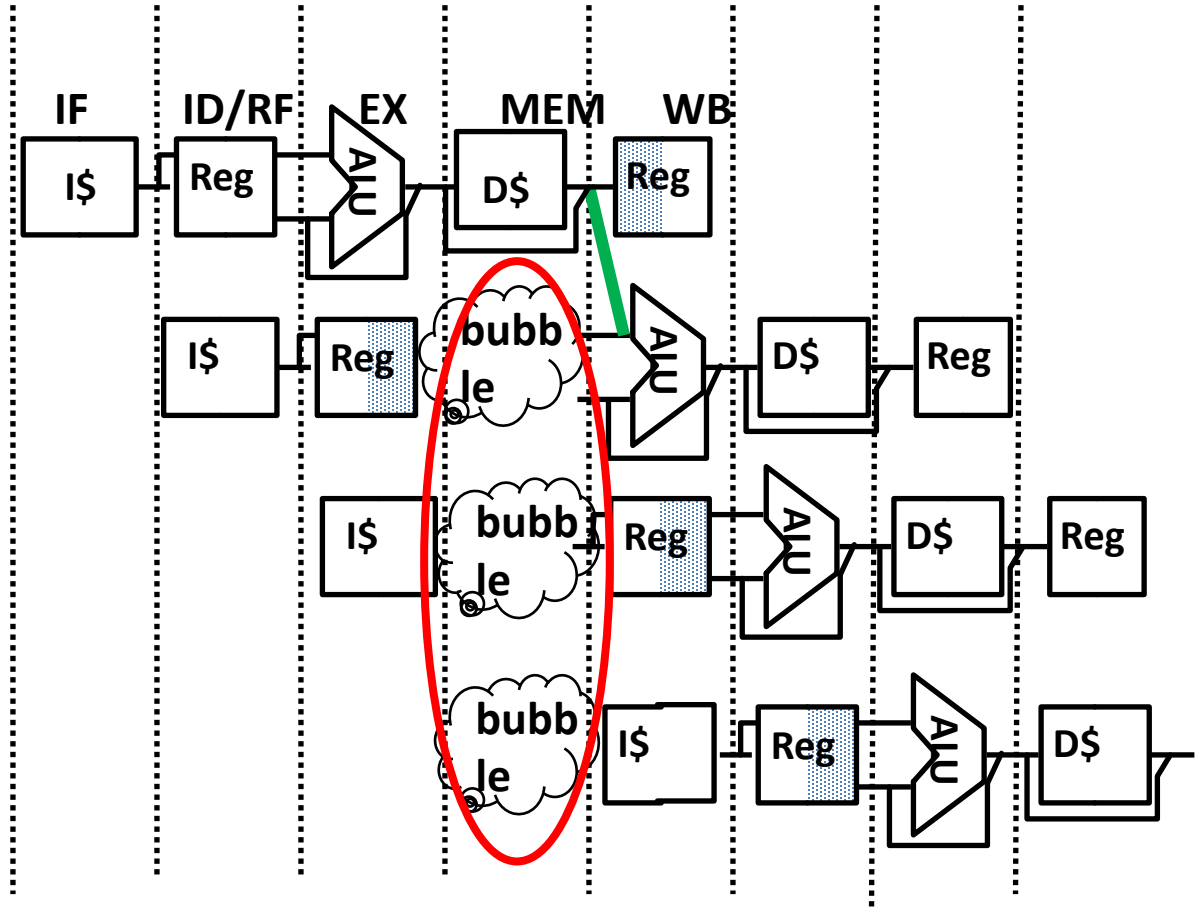
- Hardware must ***stall*** pipeline
- Also called "***interlock***"

lw $t0, 0($t1)

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



Data Hazard: Loads (3/3)

- 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.
- If 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
- *Alternatively: could redefine instruction semantics to introduce a delay slot, i.e., after lw (or lb or lbu) instruction, the data isn't actually available in the destination register until an extra instruction later*

Optimization (1/3)

- Now that we know what is fast and what is slow, how do we write fast programs?
  - As long as we avoid hazards and corresponding pipeline stalls, we maximize instruction throughput.
- First, simplest technique: maintain locality
  - Stalling a cycle or two for a control/data hazard is nothing compared to stalling hundreds of cycles for a cache miss!

Optimization (2/3)

- Instruction reordering:
  - Be aware of delay slots, reorder instructions to put a useful yet unrelated instruction in a delay slot.
  - This is a pretty tedious task: compilers and even assemblers do a good job of doing the dirty work for you.

Question

- **Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after 10^3 loops, so pipeline full)**

```
Loop:   lw    $t0, 0($s1)
        addu  $t0, $t0, $s2
        sw    $t0, 0($s1)
        addiu $s1, $s1, -4
        bne   $s1, $s3, Loop
        nop
```

- **How many pipeline stages (clock cycles) per loop iteration to execute this code?**

Answer

- **Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after 10^3 loops, so pipeline full)**

```
Loop:    1  lw     $t0, 0($s1)
         3  addu   $t0, $t0, $s2
         4  sw     $t0, 0($s1)
         5  addiu  $s1, $s1, -4
         6  bne    $s1, $s3, Loop
         7  nop
```

2 (data hazard causes stall)

(delayed branch means we execute this instruction)

- **How many pipeline stages (clock cycles) per loop iteration to execute this code?**

Optimization Question

- Instruction Reordering Example
  - The loop takes 7 clock cycles per iteration. Can you improve it?

```
Loop: lw    $t0, 0($s1)
      nop
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      addiu $s1, $s1, -4
      bne   $s1, $s3, Loop
      nop
```

Intruction Reordering Solution

- Reordering can reduce the number of cycles to 5 per iteration:

```
Loop: lw $t0, 0($s1)
      addiu $s1, $s1, -4
      addu $t0, $t0, $s2
      bne $s1, $s3, loop
      sw $t0, 4($s1)
```

Optimization (3/3): Loop Unrolling

- Branches are difficult, just avoid them if possible.
- For a loop with a fixed number of iterations, write the assembly code by just writing the body of the loop out repeatedly rather than using conditional branches.
- Tradeoff: ***more total code*** but ***fewer instructions*** executed overall and fewer branch instructions → means faster execution time.
- Can also partially unroll large loops!

Loop Unrolling Example

```
        addi $s0, $0, $0     # $s0 = 0
loop:   lw   $t0, 0($s1)
        add  $s2, $s2, $t0
        addi $s1, $s1, 4
        addi $s0, $s0, 1
        slt  $t1, $s0, 4     # if $s0 < 4
        bne  $t1, $0, loop   # then loop
```

If this loop were longer, it would be useful to eliminate the counter s0 and instead keep track of the target end address s1+4n and eliminate two instructions in the loop

What does this code do

Loop Unrolling Example

```
lw $t0, 0($s1)
add $s2, $s2, $t0
lw $t0, 4($s1)
add $s2, $s2, $t0
lw $t0, 8($s1)
add $s2, $s2, $t0
lw $t0, 12($s1)
add $s2, $s2, $t0
```

Loop Unrolling Example

```
lw $t0, 0($s1)
lw $t1, 4($s1)
lw $t2, 8($s1)
lw $t3, 12($s1)
add $s2, $s2, $t0
add $s2, $s2, $t1
add $s2, $s2, $t2
add $s2, $s2, $t3
```

Can still rearrange to use fewer registers

Loop Unrolling Example

```
lw $t0, 0($s1)
lw $t1, 4($s1)
add $s2, $s2, $t0
add $s2, $s2, $t1
lw $t0, 8($s1)
lw $t1, 12($s1)
add $s2, $s2, $t0
add $s2, $s2, $t1
```

Can still rearrange to use fewer registers

The Big Picture

- Although the compiler generally relies on the hardware to resolve hazards and thereby ensure correct execution, the compiler must understand the pipeline to achieve the best performance. Otherwise, unexpected stalls will reduce the performance of the compiled code.

Questions

A. Thanks to pipelining, I have reduced the time it took me to wash my shirt.

B. Longer pipelines are always a win (since less work per stage & a faster clock).

C. We can rely on compilers to help us avoid all data hazards by reordering instructions.

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.

Things to Remember (2/2)

- Pipelining is 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?
    ⇒ Delayed branch
  - Data hazards: an instruction depends on a previous instruction

Review and More Information

- Textbook Section 4.5 and 4.6
- Hazards in Sections 4.7 and 4.8

Extra Question

- Implement the memory copy function

```
memcopy( int[] A, int[] B, int n ) {
    for ( int i = 0; i < n; i++ ) A[i] = B[i];
}
```

  - Assume a MIPS machine with 1 instruction per clock cycle, delayed branching, a 5 stage pipeline, forwarding, and interlock on unresolved load hazards
  - Respect register conventions
  - Use only true assembly language
  - Use careful instruction ordering to make a loop that takes the shortest possible number of cycles to complete