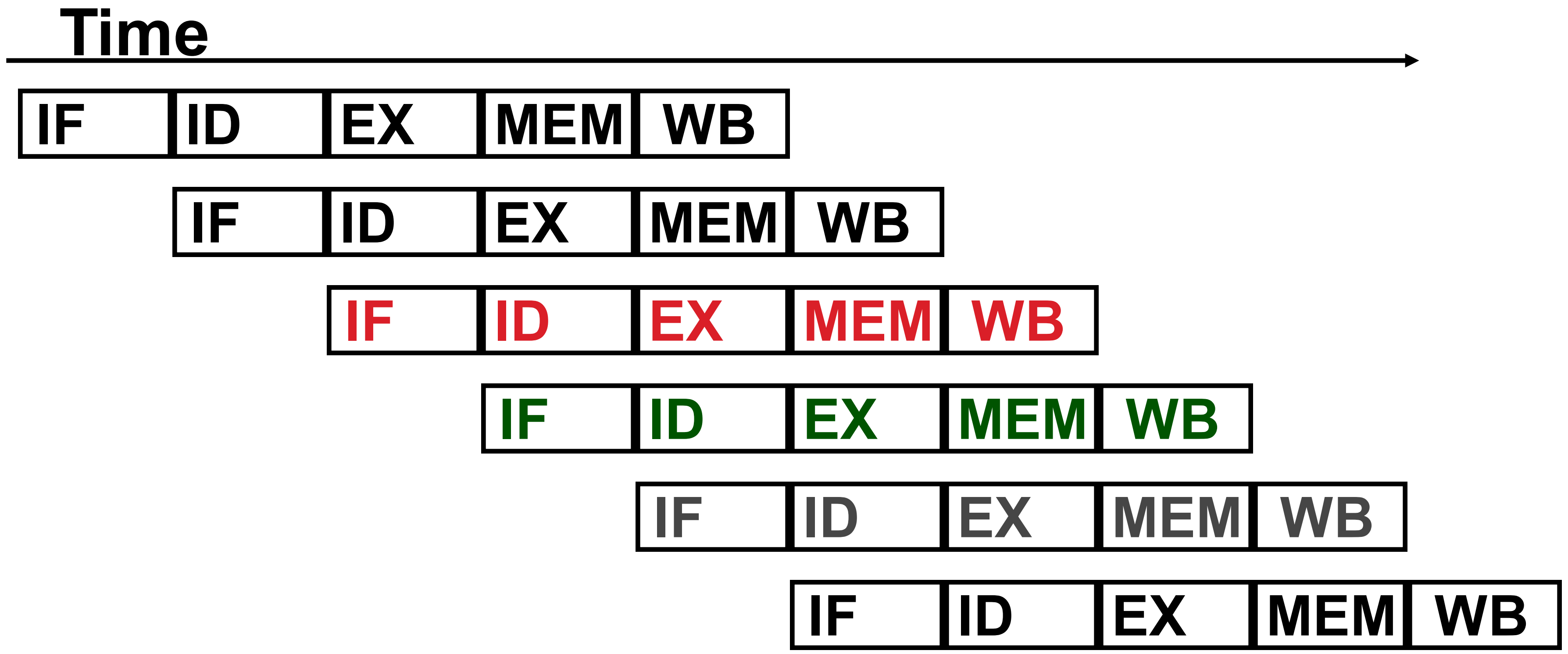
CS 61C: Great Ideas in Computer Architecture Pipelining and Hazards

Instructors:

Vladimir Stojanovic and Nicholas Weaver

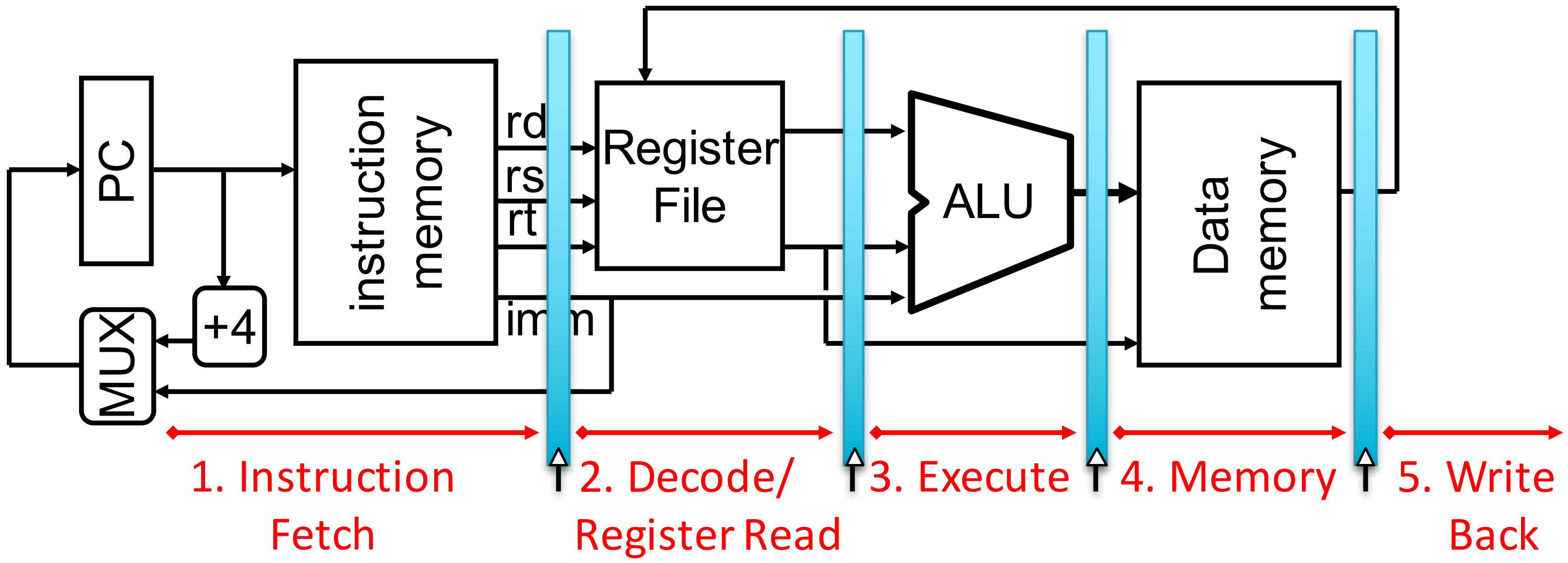
http://inst.eecs.Berkeley.edu/~cs61c/sp16

Pipelined Execution Representation

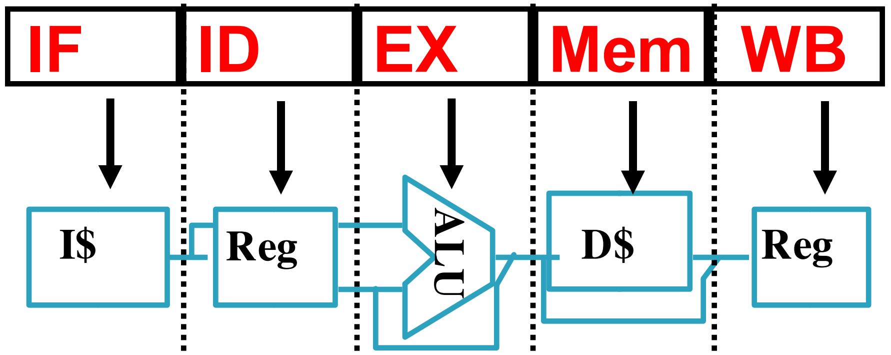


- Every instruction must take same number of steps, so some stages will idle
  – e.g. MEM stage for any arithmetic instruction

Graphical Pipeline Diagrams

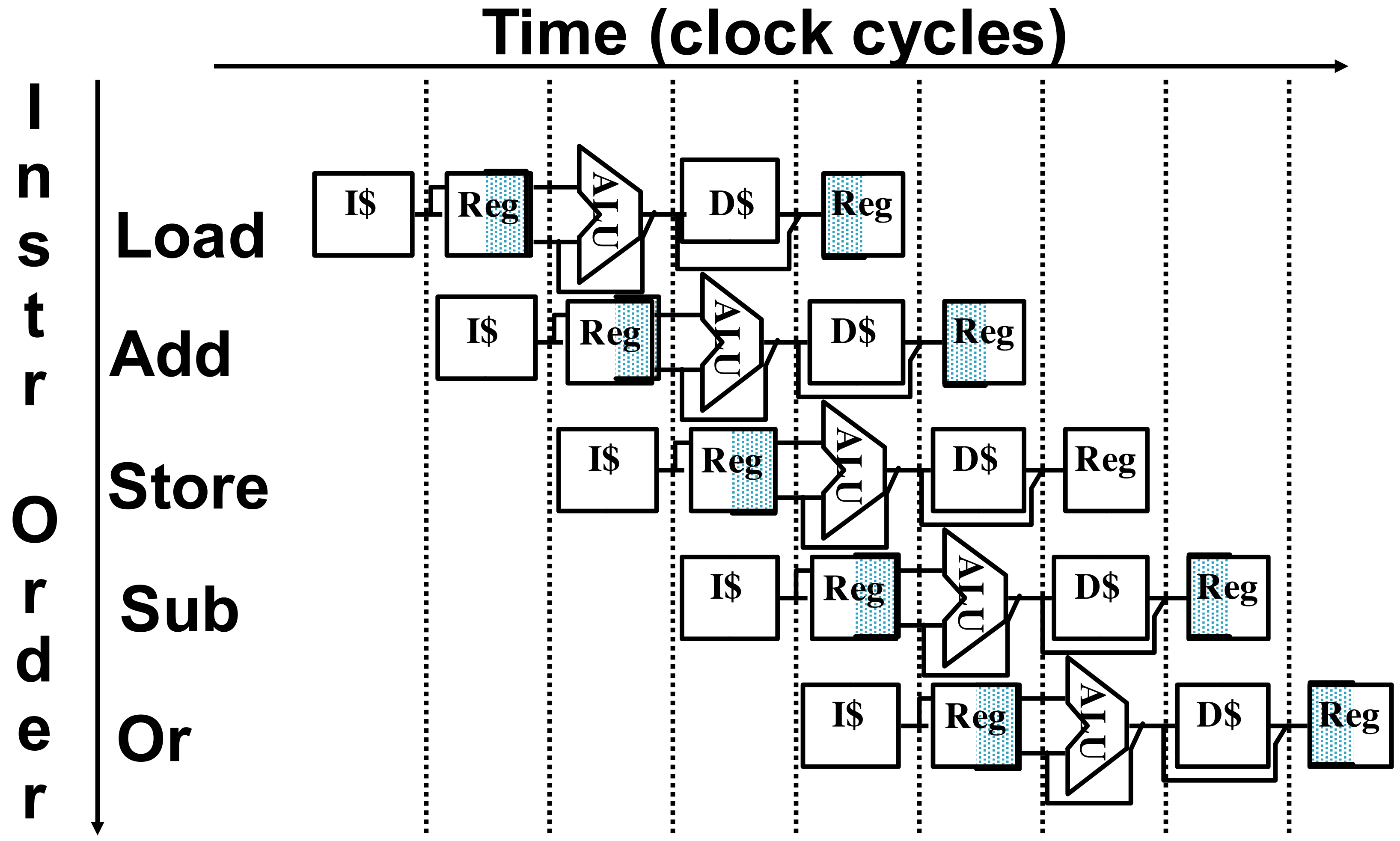


- Use datapath figure below to represent pipeline:



Graphical Pipeline Representation

- RegFile: left half is write, right half is read



Pipelining Performance (1/3)

- Use T_c ("time between completion of instructions") to measure speedup
  - $T_{c,\text{pipelined}} \geq \dfrac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$
  - Equality only achieved if stages are *balanced* (i.e. take the same amount of time)
- If not balanced, speedup is reduced
- Speedup due to increased *throughput*
  - *Latency* for each instruction does not decrease
  - In fact, *latency* must increase as the pipeline registers themselves add delay (why Nick's Ph.D. thesis has a "this was a stupid idea" chapter)

Pipelining Performance (2/3)

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

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

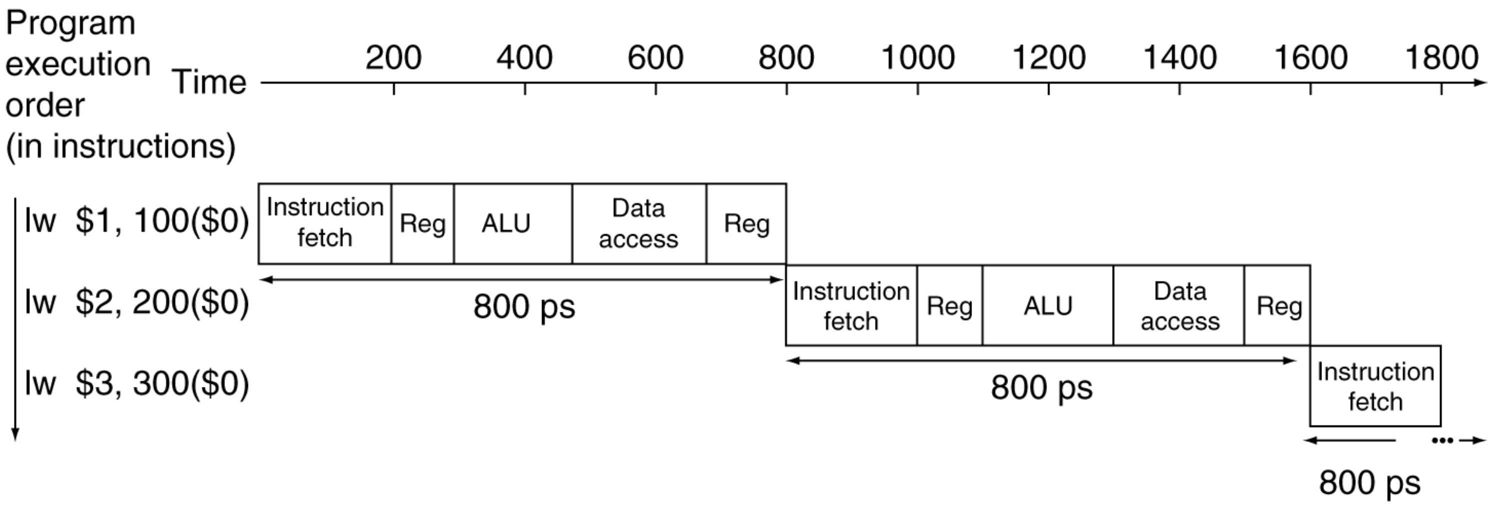
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

Pipelining Performance (3/3)

Single-cycle

T_c = 800 ps

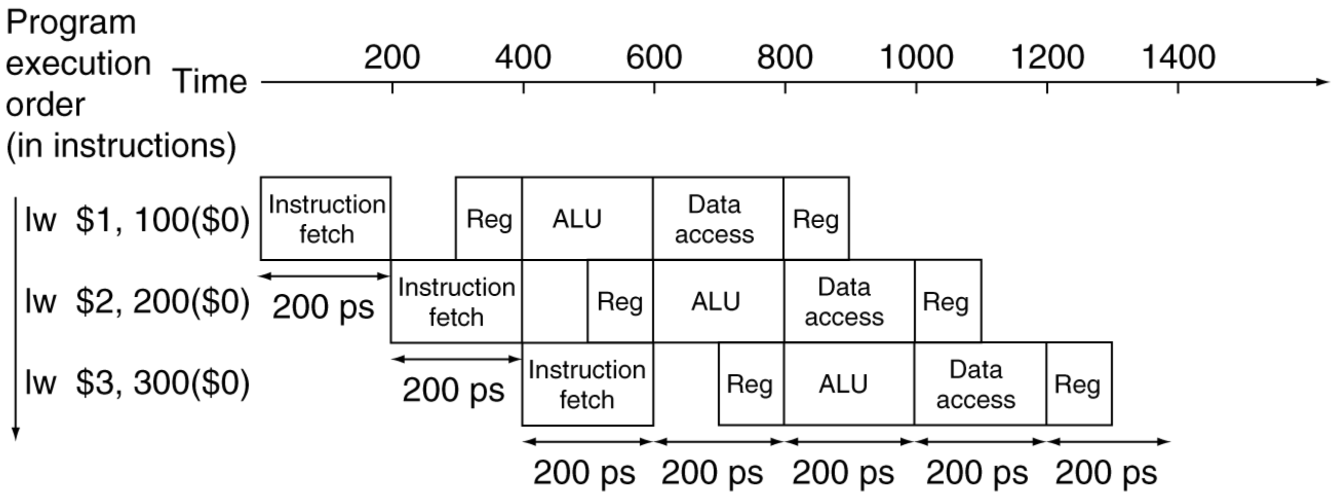
f = 1.25GHz



Pipelined

T_c = 200 ps

f = 5GHz



Clicker/Peer Instruction

Logic in some stages takes 200ps and in some 100ps. Clk-Q delay is 30ps and setup-time is 20ps. What is the maximum clock frequency at which a pipelined design can operate?

- A: 10GHz
- B: 5GHz
- C: 6.7GHz
- D: 4.35GHz
- E: 4GHz

Administrivia…

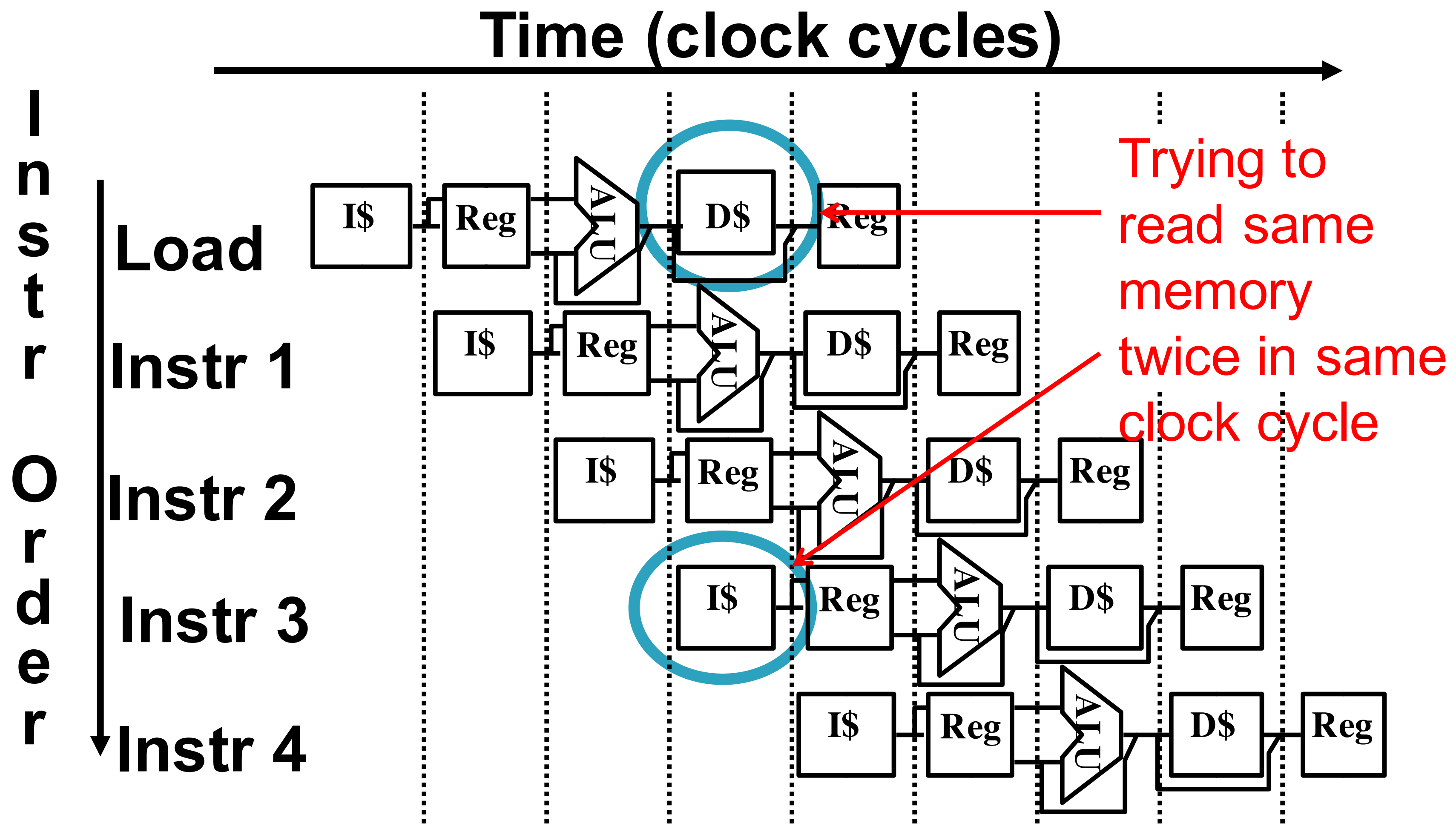
- Start on Project 3-1 now
  - Logisim can be a bit, well, tedious:
    The project isn't necessarily hard but it will take a fair amount of time
    - Alternative would be to have you learn ***yet another*** programming language in this class!
  - For reference, it took Nick about an hour of tediously drawing lines for his solution to part 1
    - 5 minutes to know what he wanted to do…
    - And 55 minutes to actually do it. ☹

Pipelining Hazards

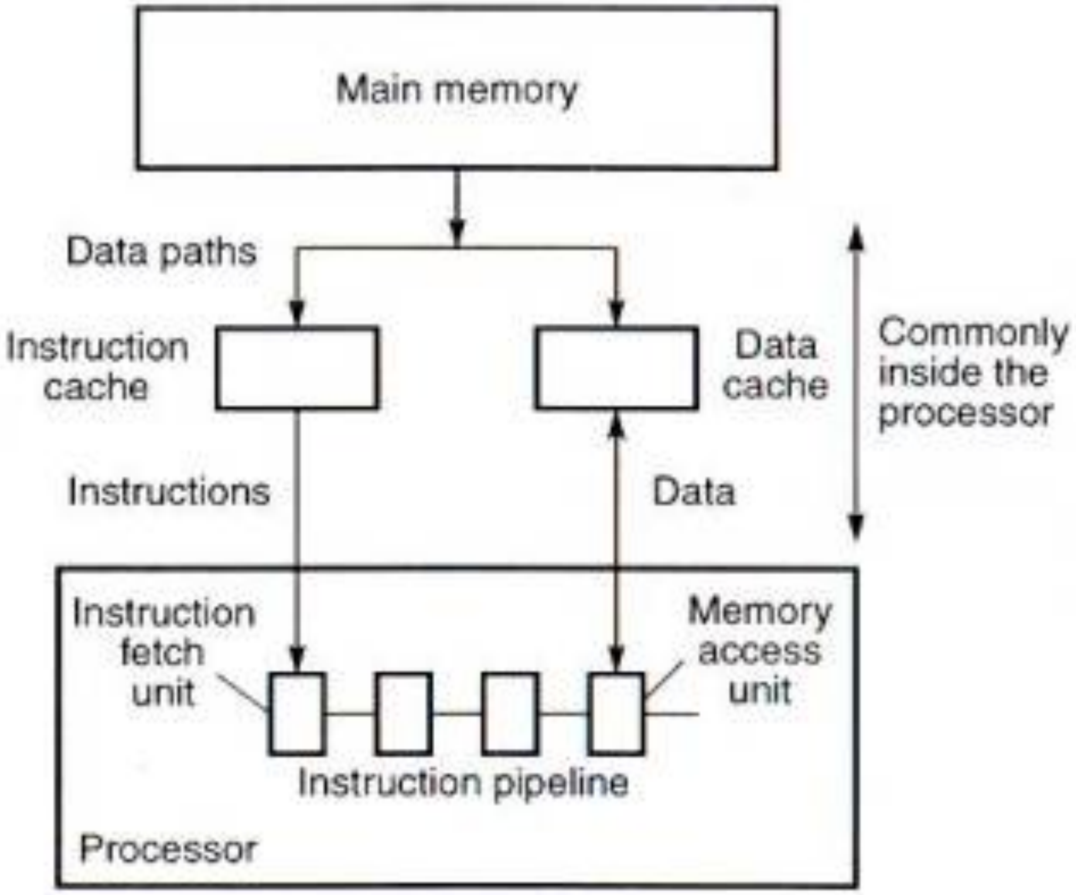
A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) *Structural hazard*
   - A required resource is busy (e.g. needed in multiple stages)
2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write
3) *Control hazard*
   - Flow of execution depends on previous instruction

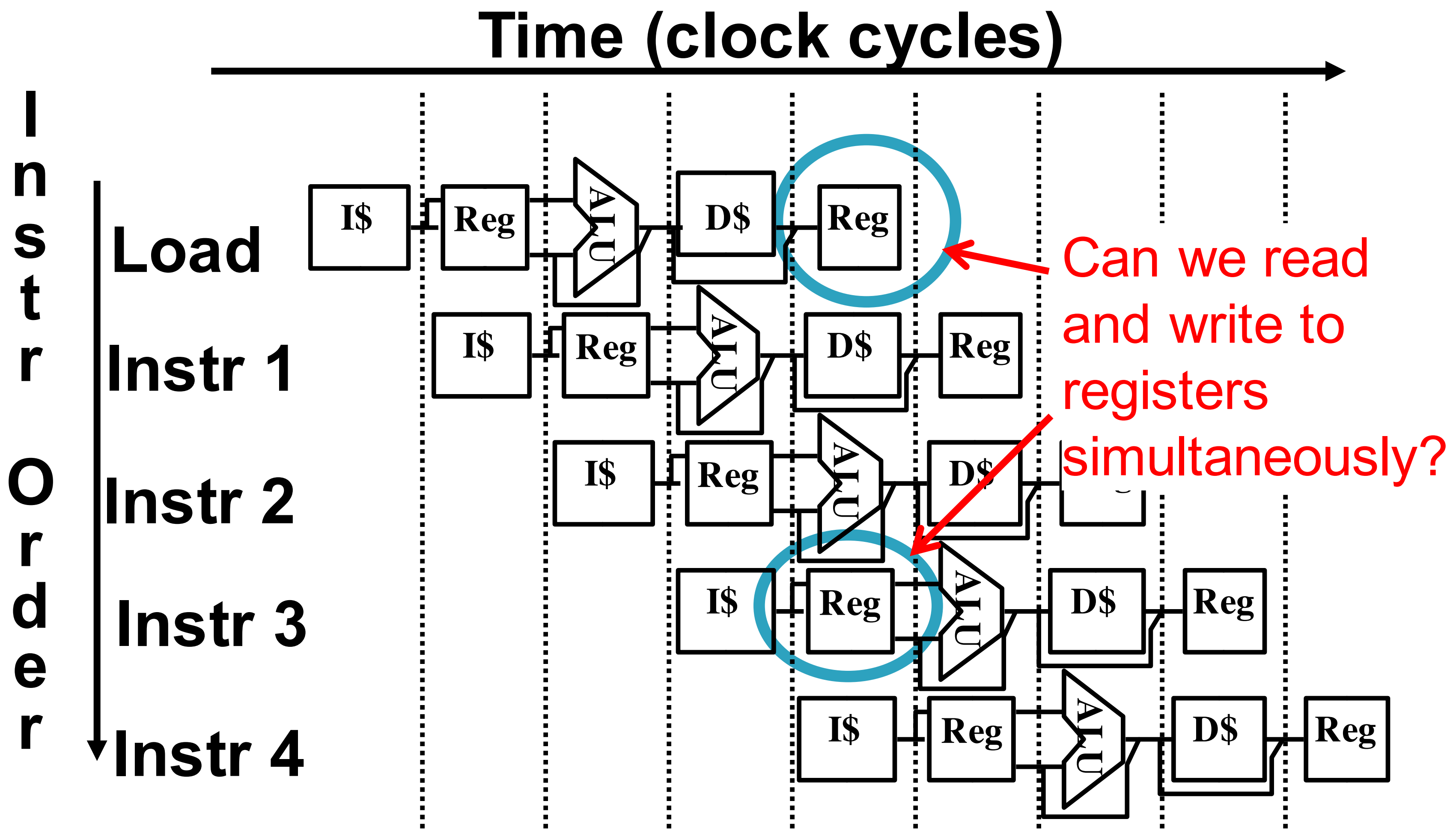
Structural Hazard #1: Single Memory



Solving Structural Hazard #1 with Caches



Structural Hazard #2: Registers (1/2)



Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) Split RegFile access in two: Write during 1st half and Read during 2nd half of each clock cycle
     - Possible because RegFile access is *VERY* fast (takes less than half the time of ALU stage)
  2) Build RegFile with independent read and write ports (E.g. for your project)
- **Conclusion:** Read and Write to registers during same clock cycle is okay

Structural hazards can (almost) always be removed by adding hardware resources

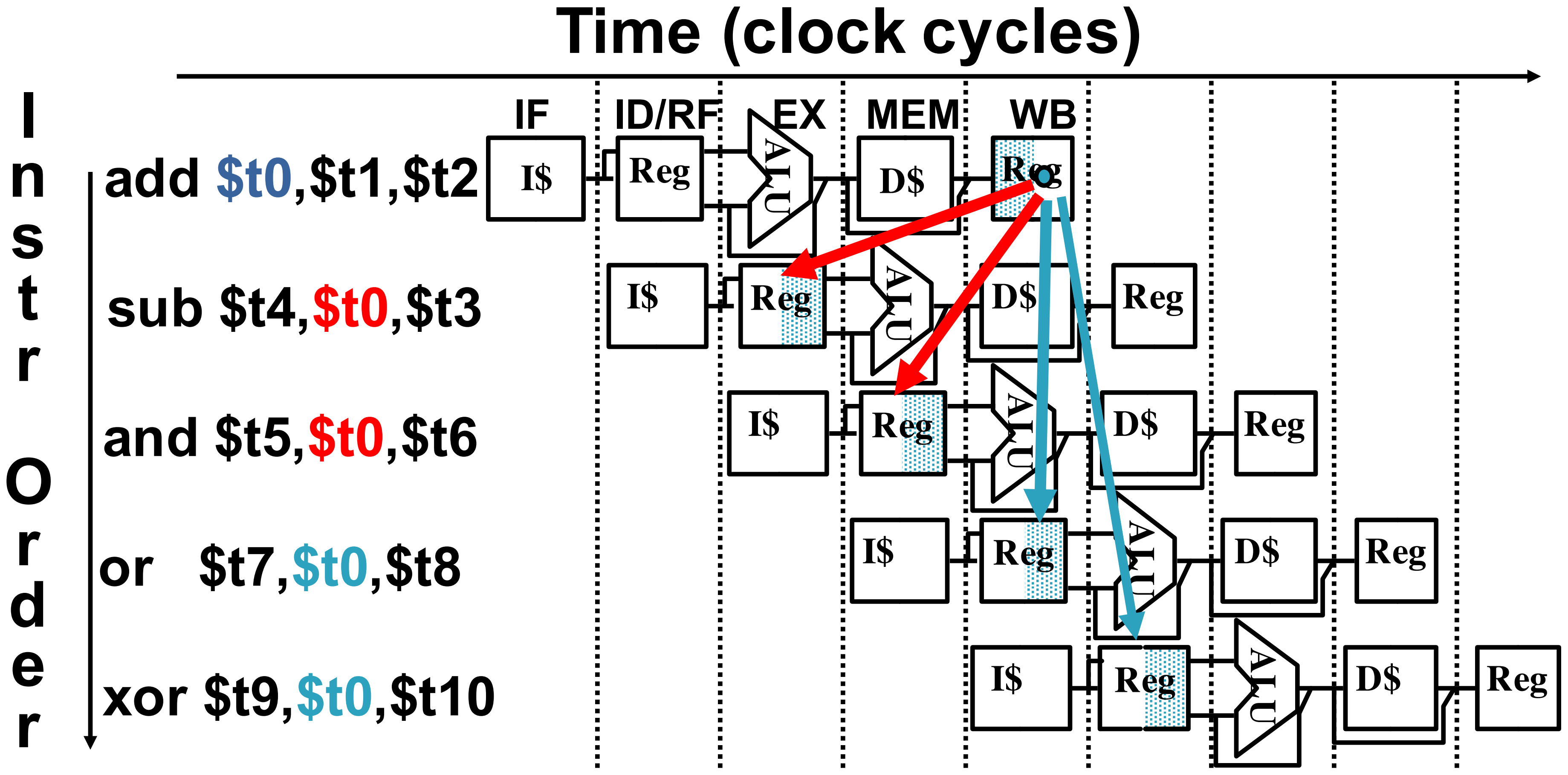
Data Hazards (1/2)

- 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
```

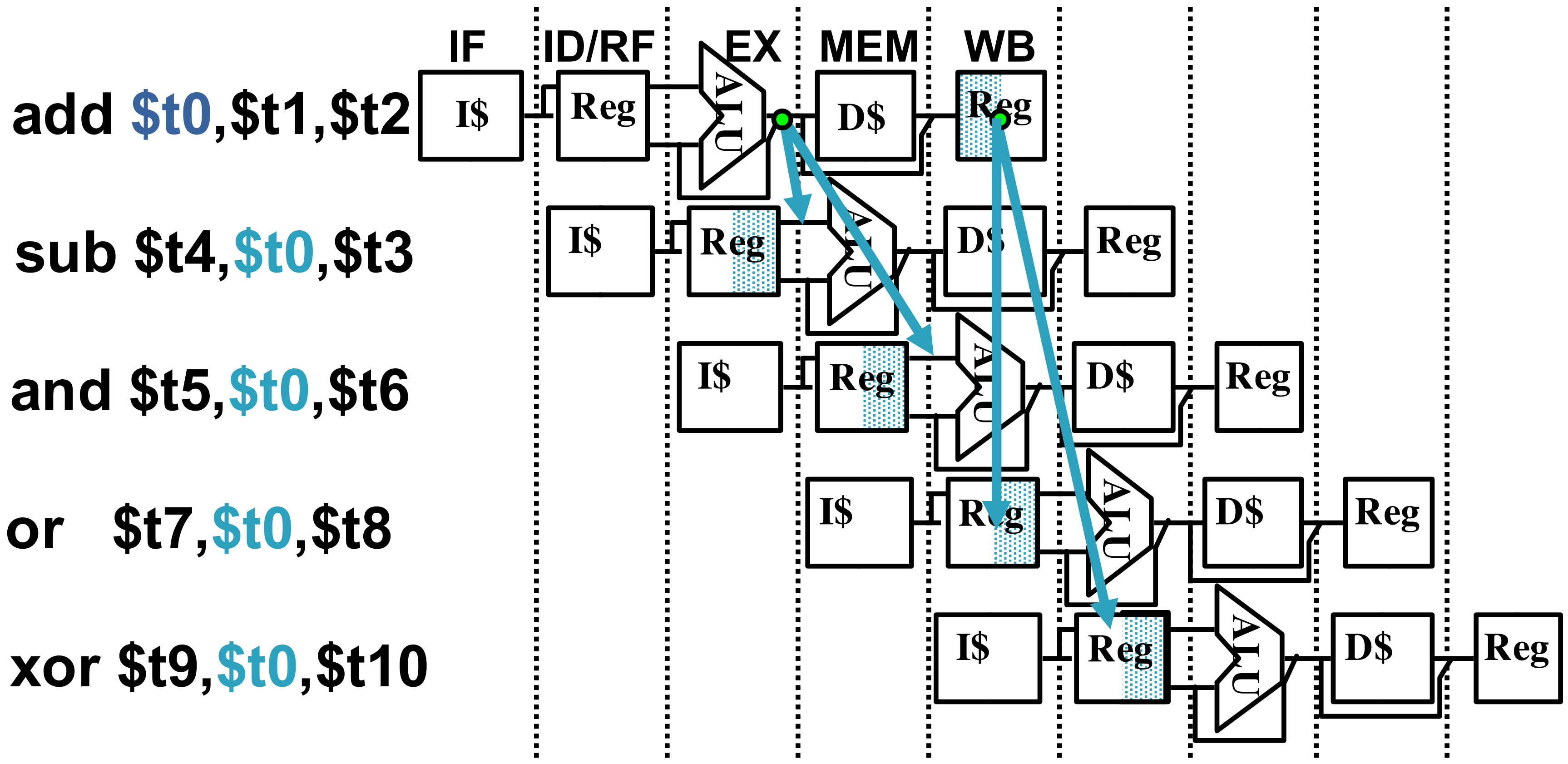
2. Data Hazards (2/2)

- Data-flow *backwards* in time are hazards



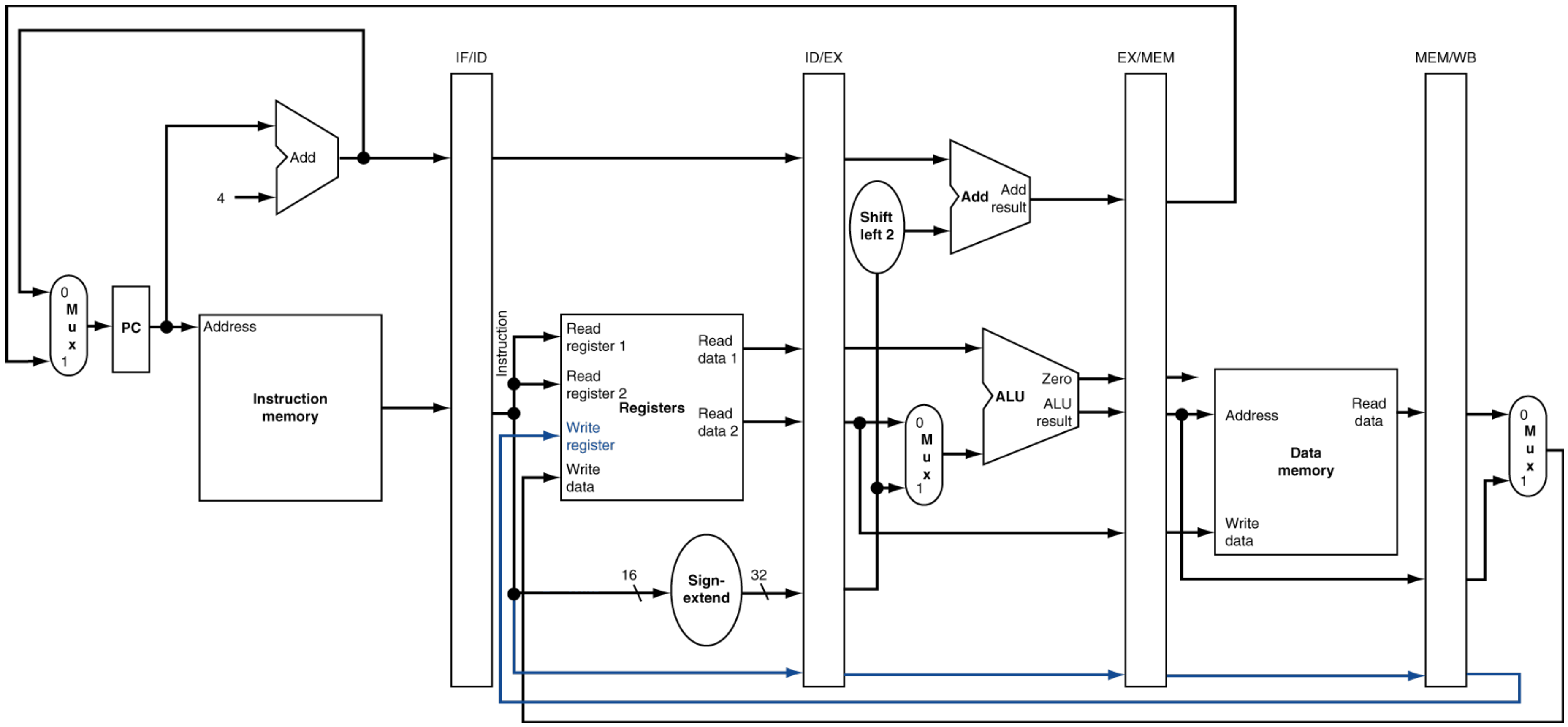
Data Hazard Solution: Forwarding

- Forward result as soon as it is available
  - OK that it's not stored in RegFile yet



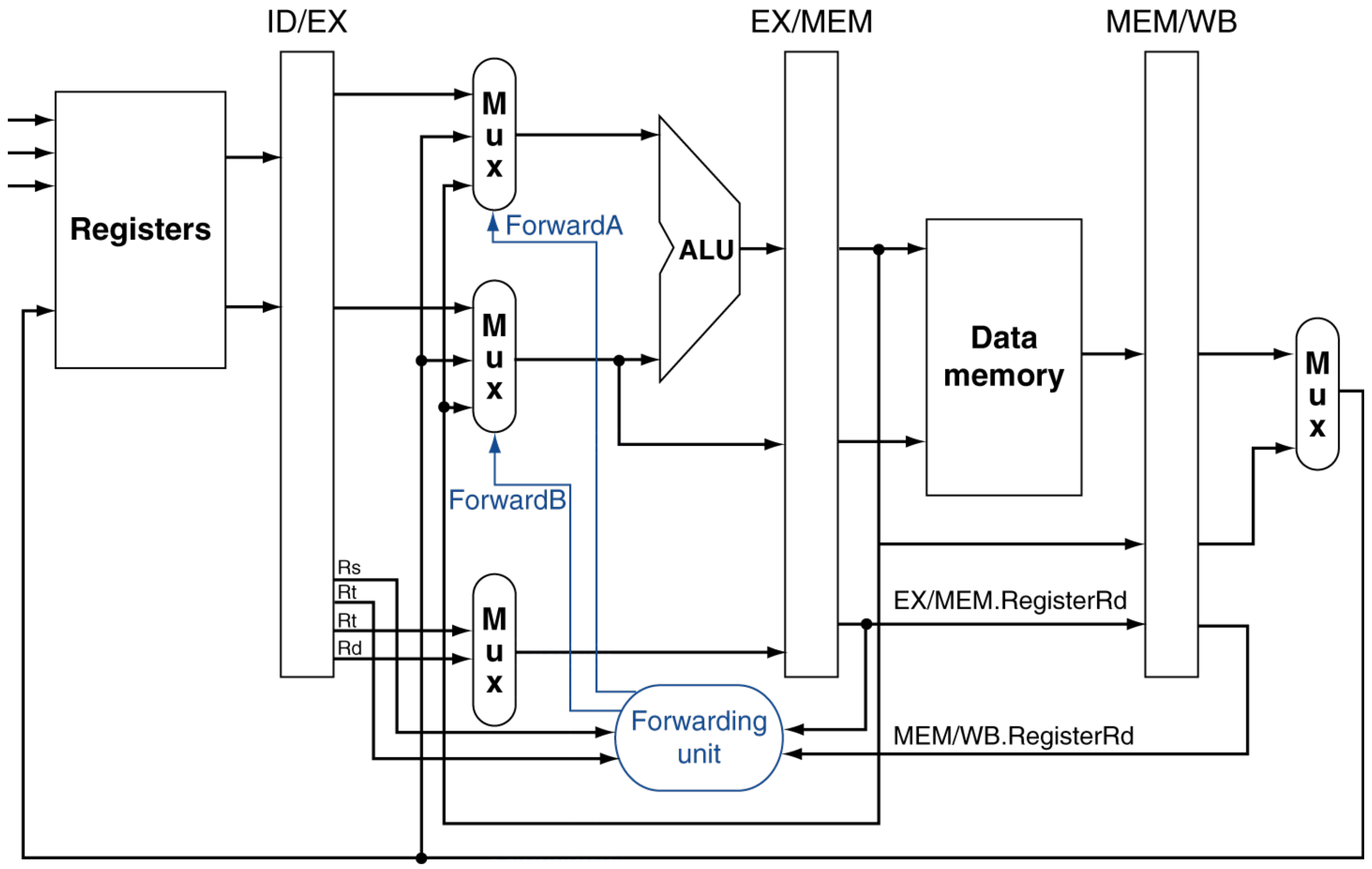
Datapath for Forwarding (1/2)

- What changes need to be made here?

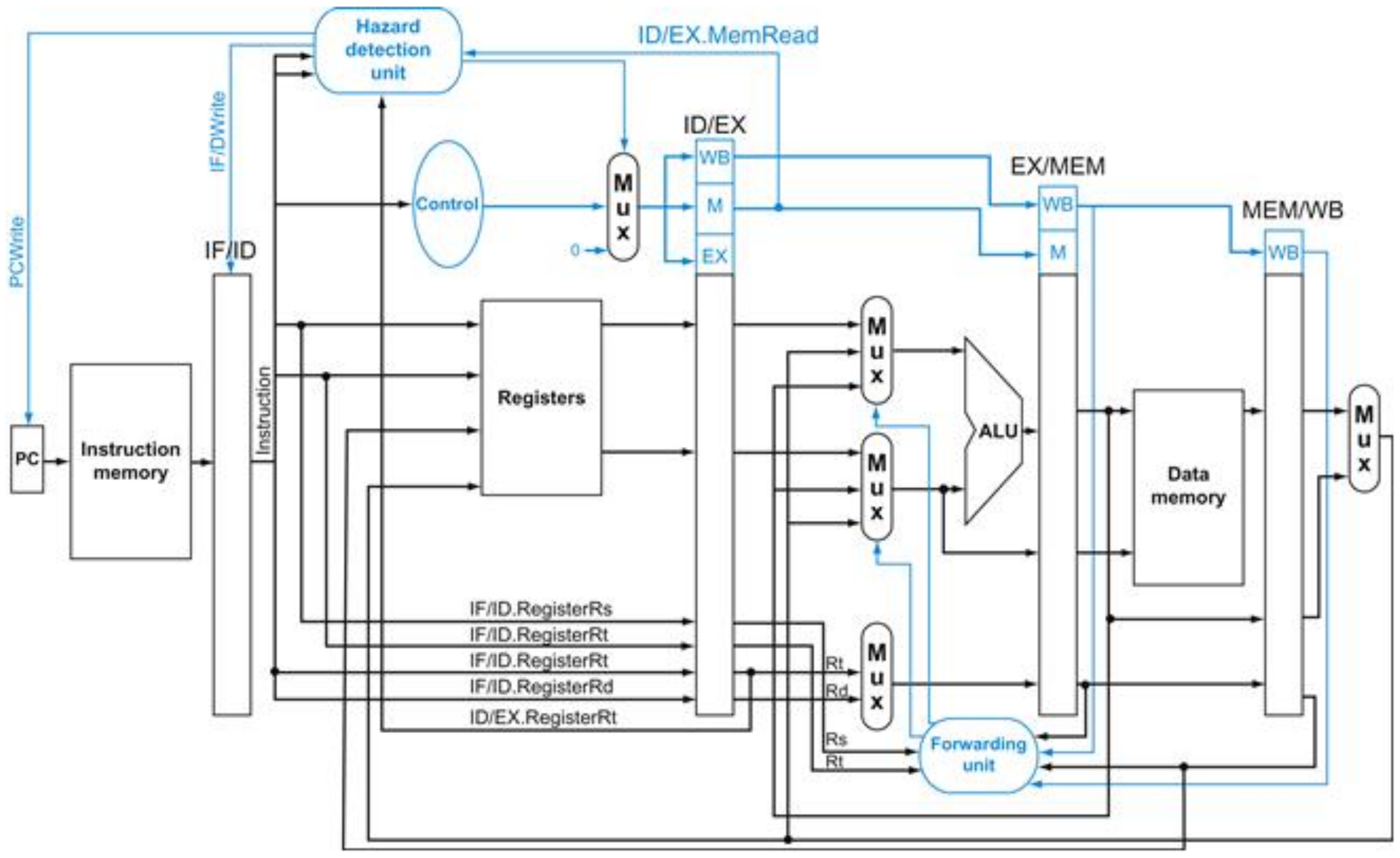


Datapath for Forwarding (2/2)

- Handled by *forwarding unit*



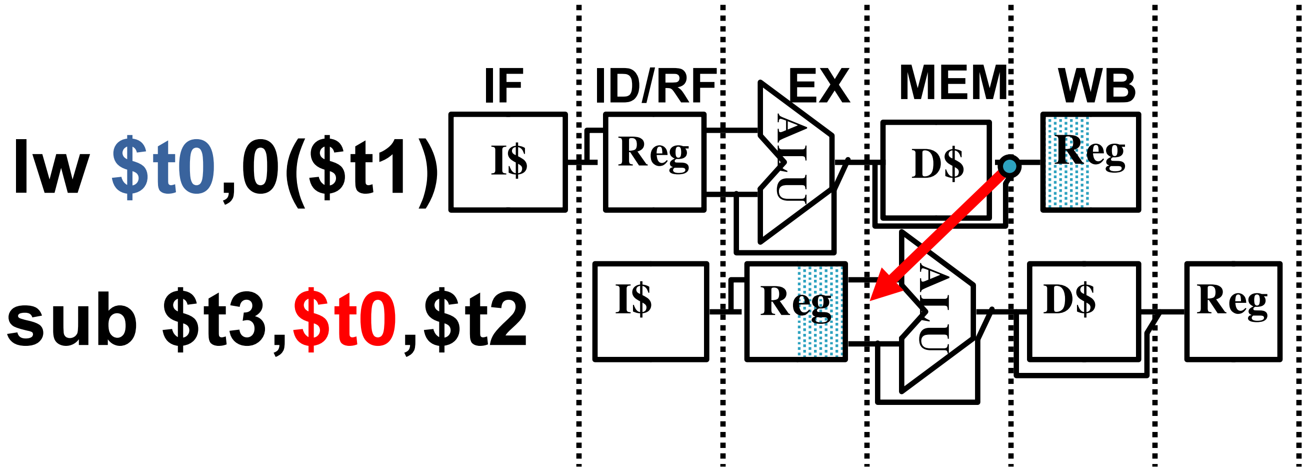
Datapath and Control



- The control signals are pipelined, too

Data Hazard: Loads (1/3)

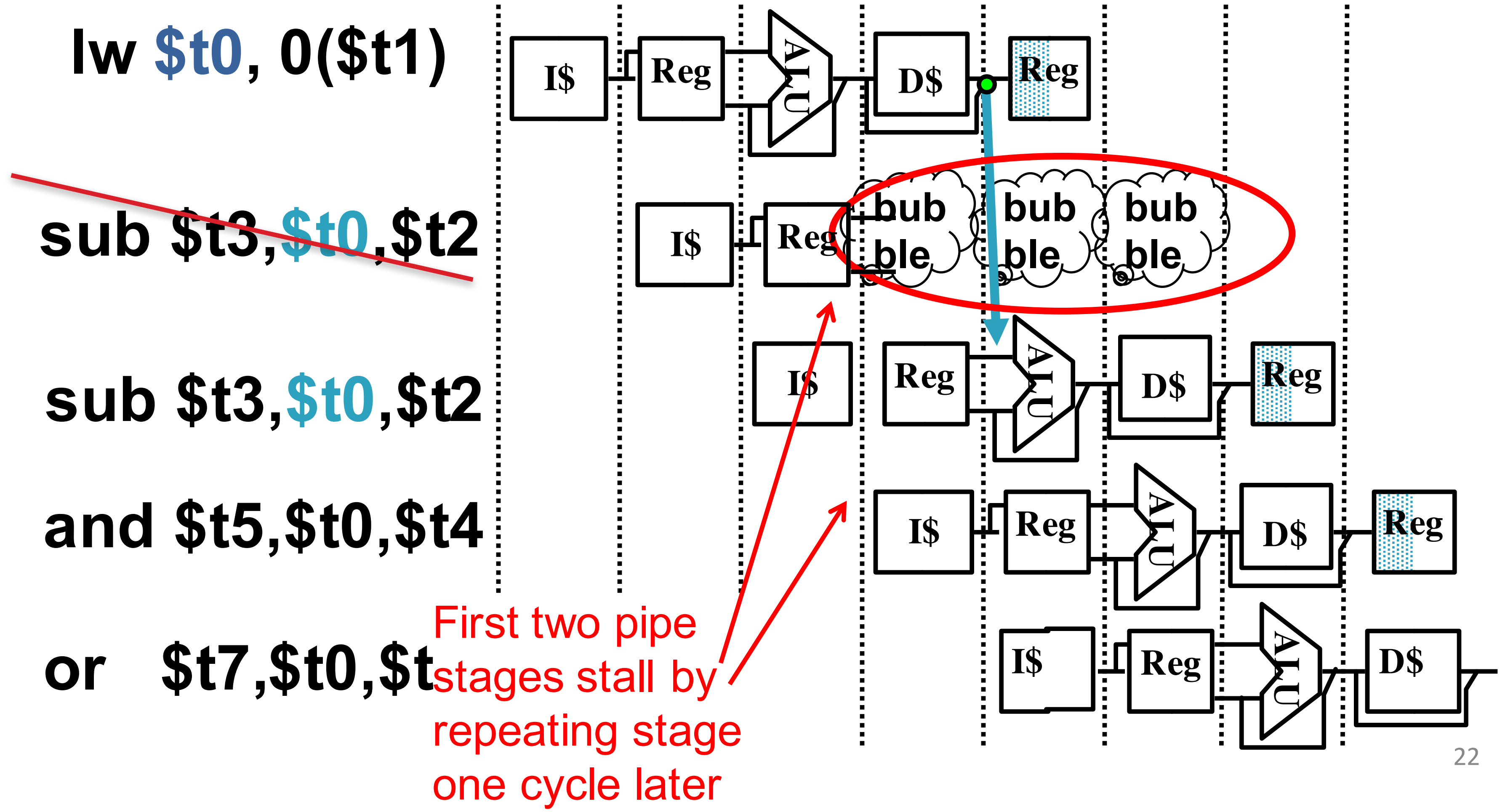
- **Recall:** Dataflow backwards in time are hazards



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

Data Hazard: Loads (2/3)

- Stalled instruction converted to “bubble”, acts like nop



Data Hazard: Loads (4/4)

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  - Letting the hardware stall the instruction in the delay slot is equivalent to putting an explicit `nop` in the slot (except the latter uses more code space)
- **Idea:** Let the compiler put an unrelated instruction in that slot → no stall!

Clicker Question

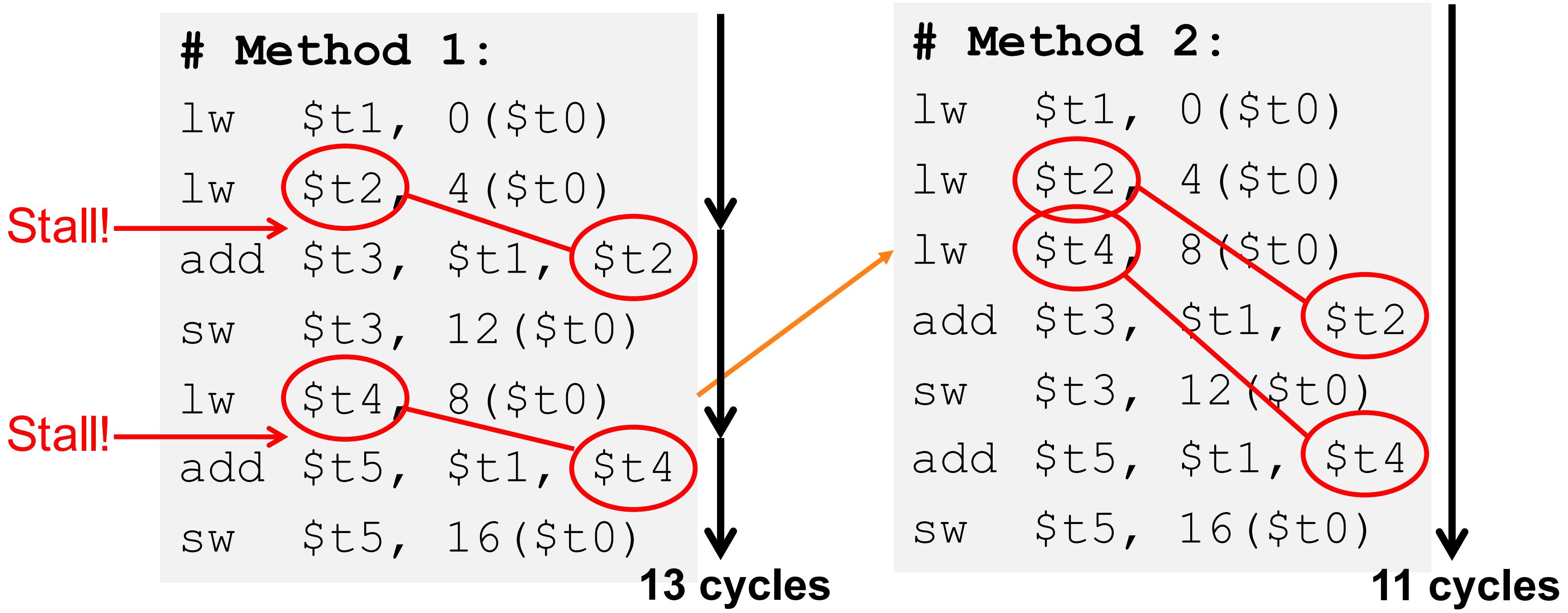
How many cycles (pipeline fill+process+drain) does it take to execute the following code?

```
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)
```

A. 7
B. 9
C. 11
D. 13
E. 14

Code Scheduling to Avoid Stalls

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



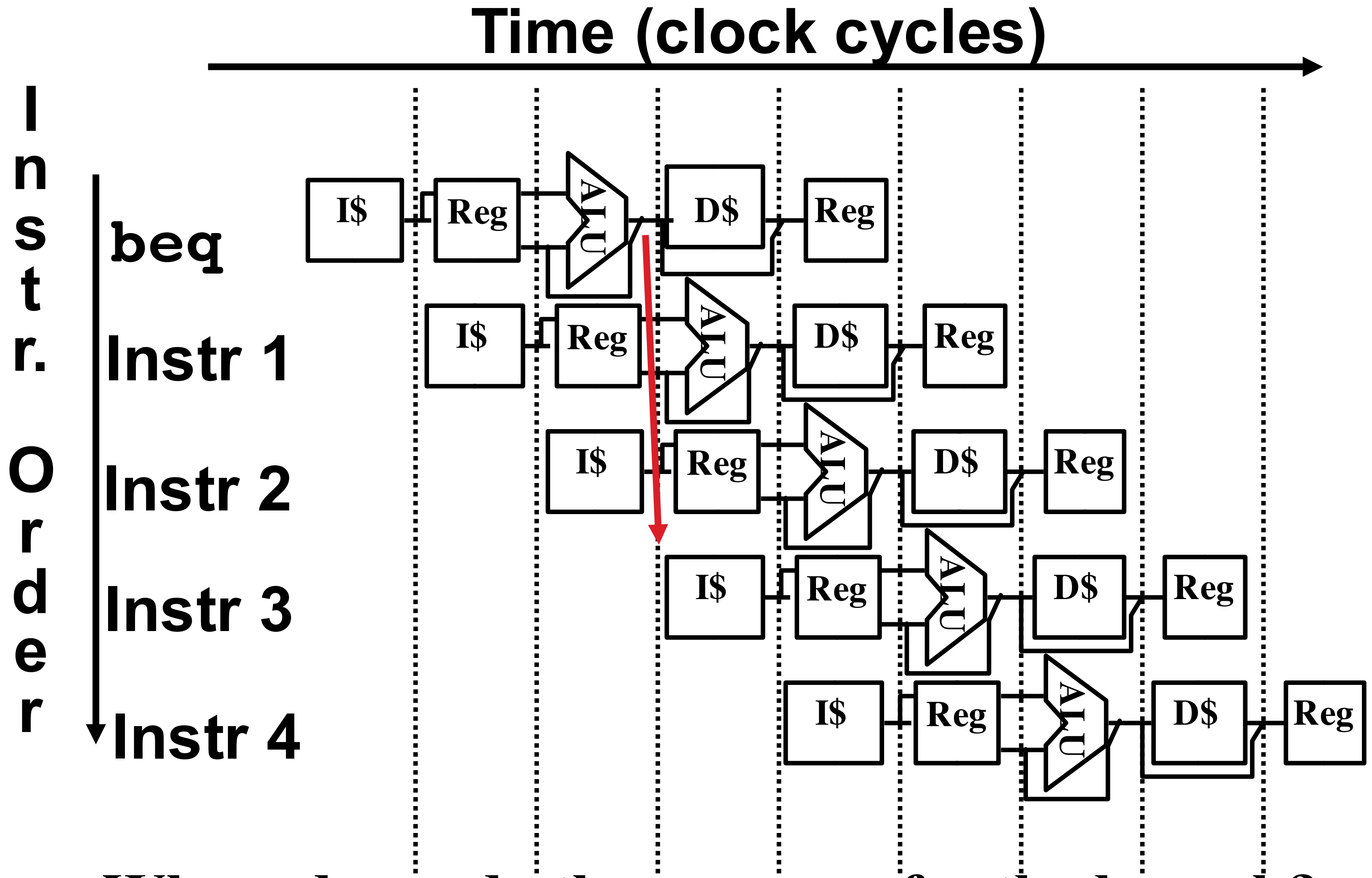
```
# Method 1:
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

```
# Method 2:
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)
```

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 branch condition resolved
  - Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)

Stall => 2 Bubbles/Clocks

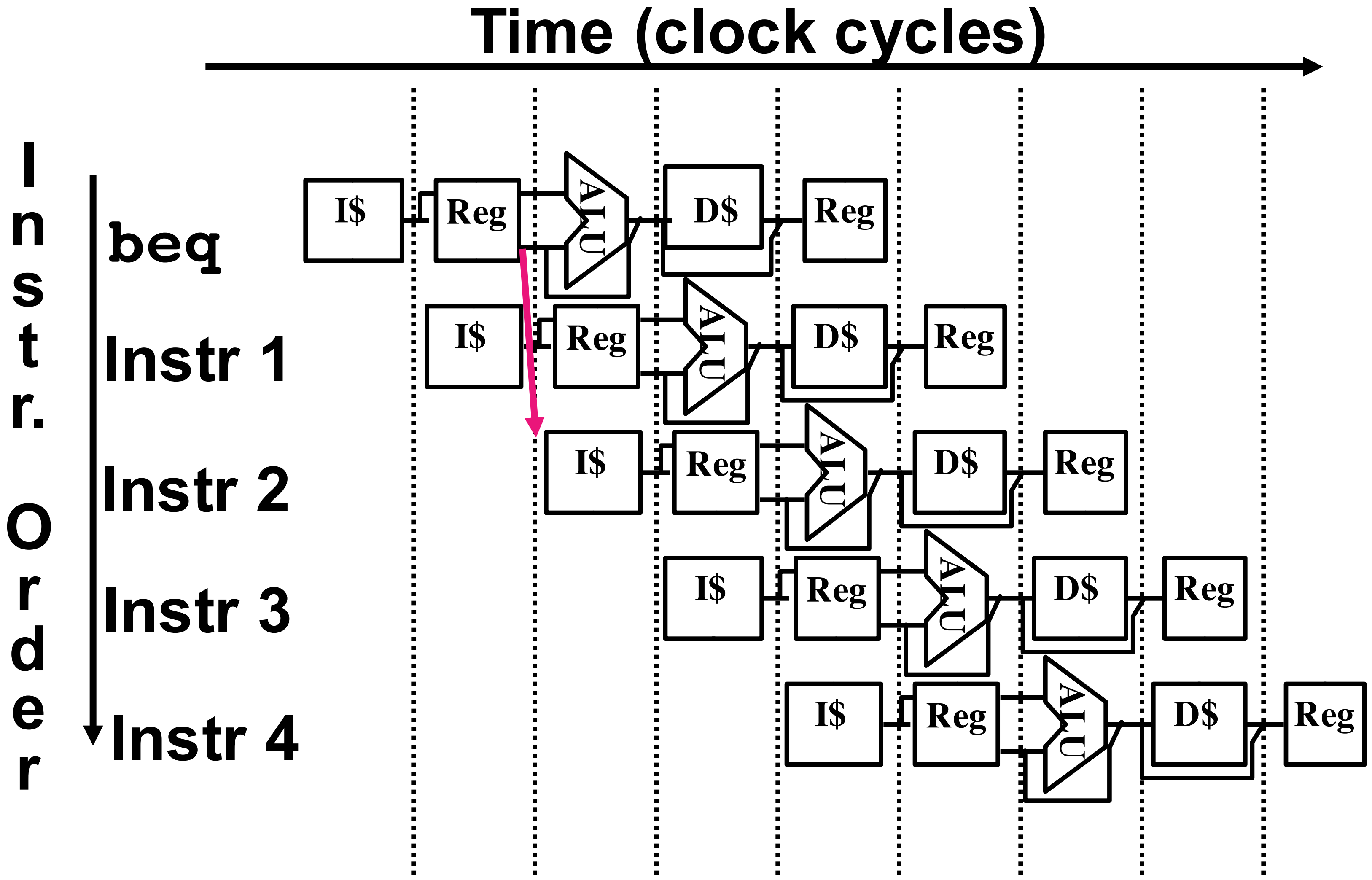


Where do we do the compare for the branch?

Control Hazard: Branching

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

One Clock Cycle Stall



Branch comparator moved to Decode stage.

Control Hazards: Branching

- Option 2: *Predict* outcome of a branch, fix up if guess wrong
  - Must cancel all instructions in pipeline that depended on guess that was wrong
  - This is called "flushing" the pipeline
- Simplest hardware if we predict that all branches are NOT taken
  - Why?

Control Hazards: Branching

- Option #3: 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 (the *branch-delay slot*)
- *Delayed Branch* means *we always execute inst after branch*
- This optimization is used with MIPS

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:
```

Control Hazards: Branching

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: put a nop in the branch-delay slot
  - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn't affect the logic of program
    - Re-ordering instructions is common way to speed up programs
    - Compiler usually finds such an instruction 50% of time
    - Jumps also have a delay slot …

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
- “Out-of-Order” execution
  - Reorder instructions dynamically in hardware to reduce impact of hazards
- "Multithreading"
  - Share functional units between independent threads of execution
- *Take CS152 next to learn about these techniques!*

In Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions in different pipestages
- Pipestages should be balanced for highest clock rate
- Three types of pipeline hazard limit performance
  - Structural (always fixable with more hardware)
  - Data (use interlocks or bypassing to resolve)
  - Control (reduce impact with branch prediction or branch delay slots)