CS 61C: Great Ideas in Computer Architecture

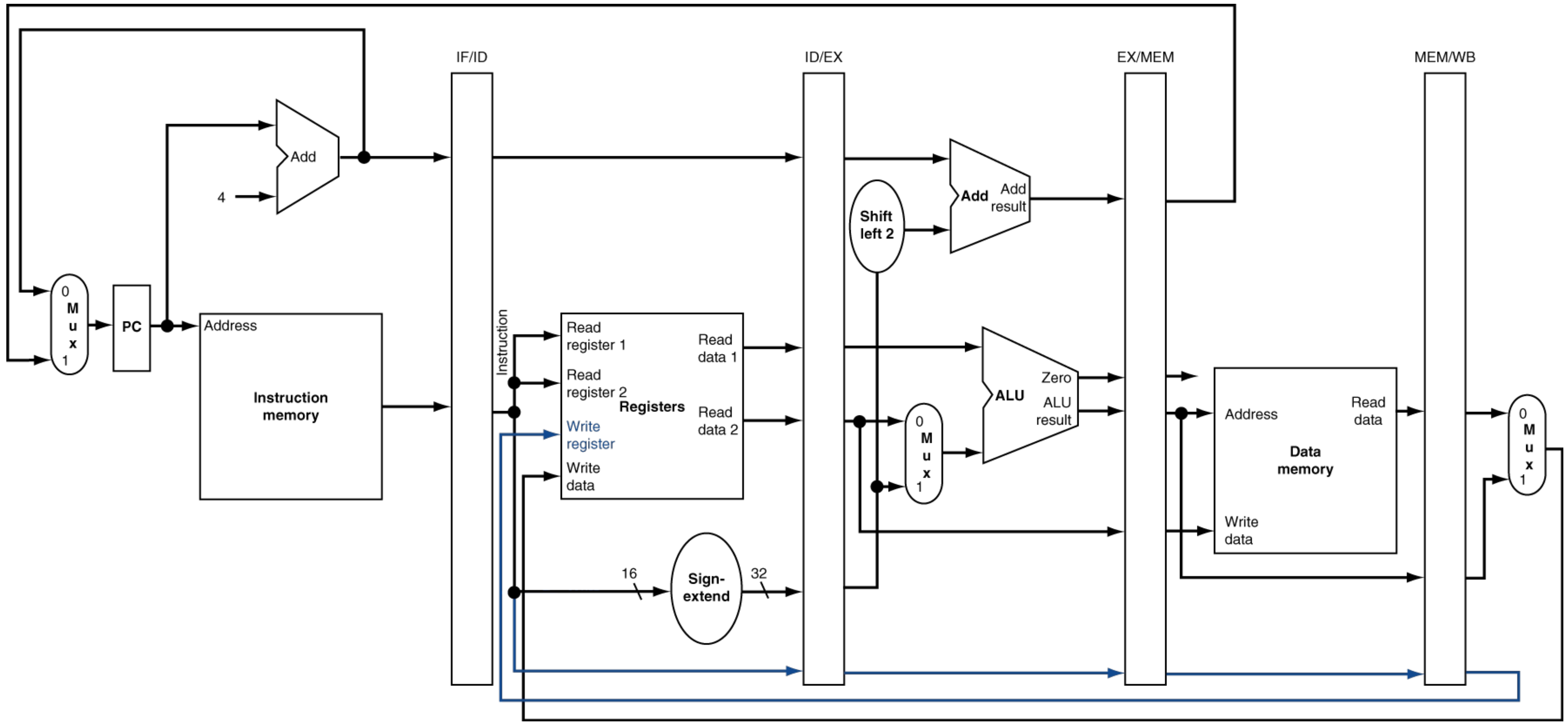
Pipelining Hazards

Instructor: Alan Christopher

Review of Last Lecture

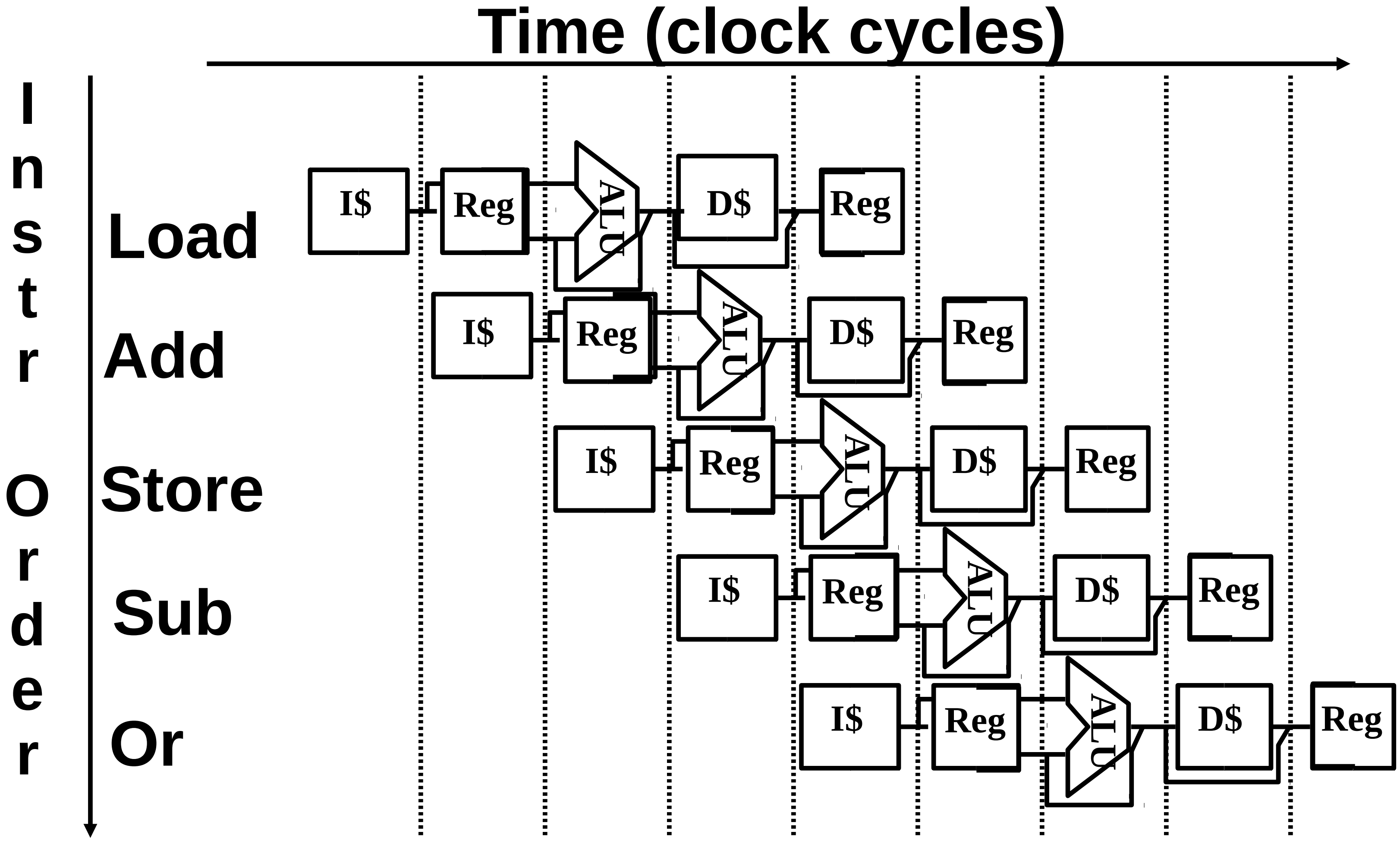
- Implementing controller for your datapath
  - Take decoded signals from instruction and generate control signals
  - Use “AND” and “OR” Logic scheme
- Pipelining improves performance by exploiting Instruction Level Parallelism
  - 5-stage pipeline for MIPS: IF, ID, EX, MEM, WB
  - Executes multiple instructions in parallel
  - Each instruction has the same latency

Review: Pipelined Datapath



Graphical Pipeline Representation

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



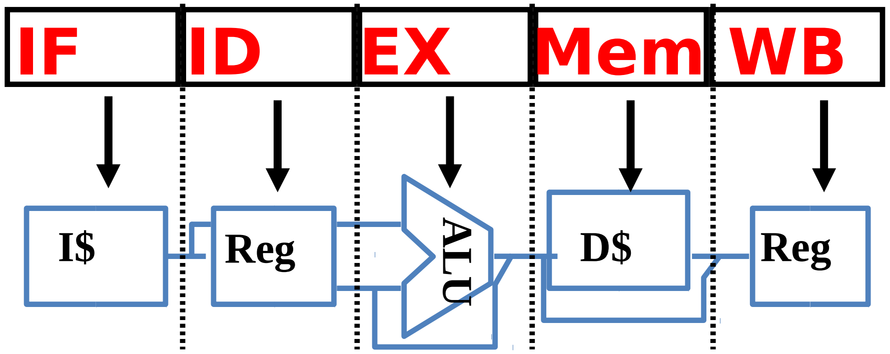
Question: Which of the following signals (buses or control signals) for MIPS-lite does NOT need to be passed into the EX pipeline stage?

(B) ALUsrc

(G) MemWr

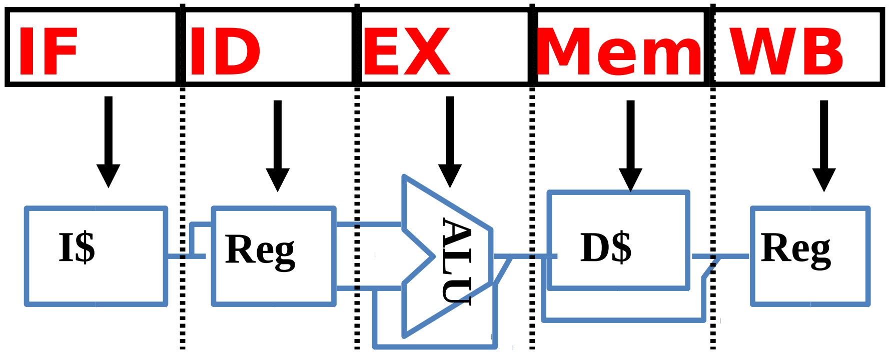
(P) RegWr

(Y) imm16



Question: Which of the following signals (buses or control signals) for MIPS-lite does NOT need to be passed into the EX pipeline stage?

- **(B) ALUsrc**
- **(G) MemWr**
- **(P) RegWr**
- **(Y) imm16**



Pipelining Hazards

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

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)

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 write

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 write

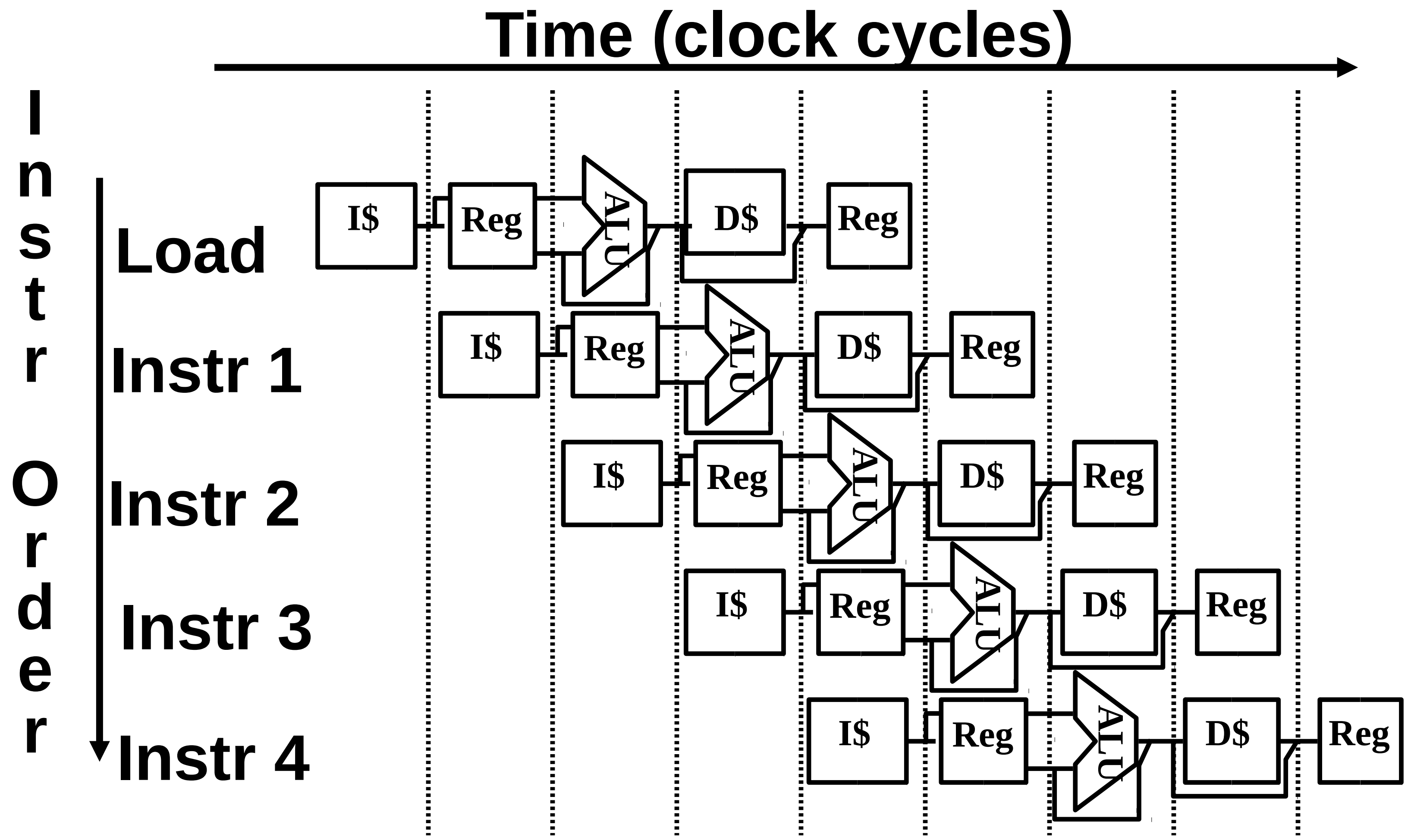
3) *Control hazard*
   - Flow of execution depends on previous instruction

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Administrivia
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

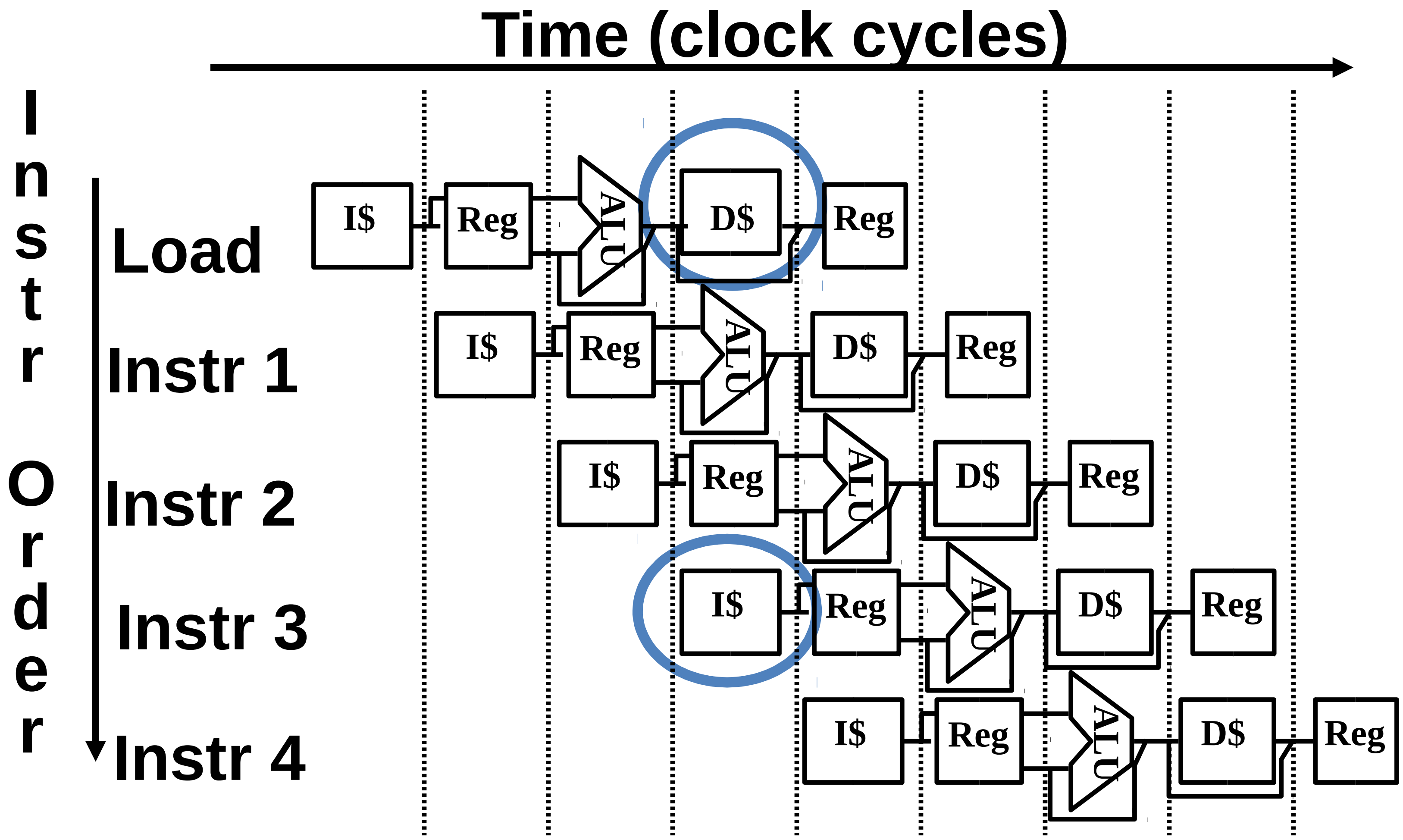
1. Structural Hazards

- Conflict for use of a resource



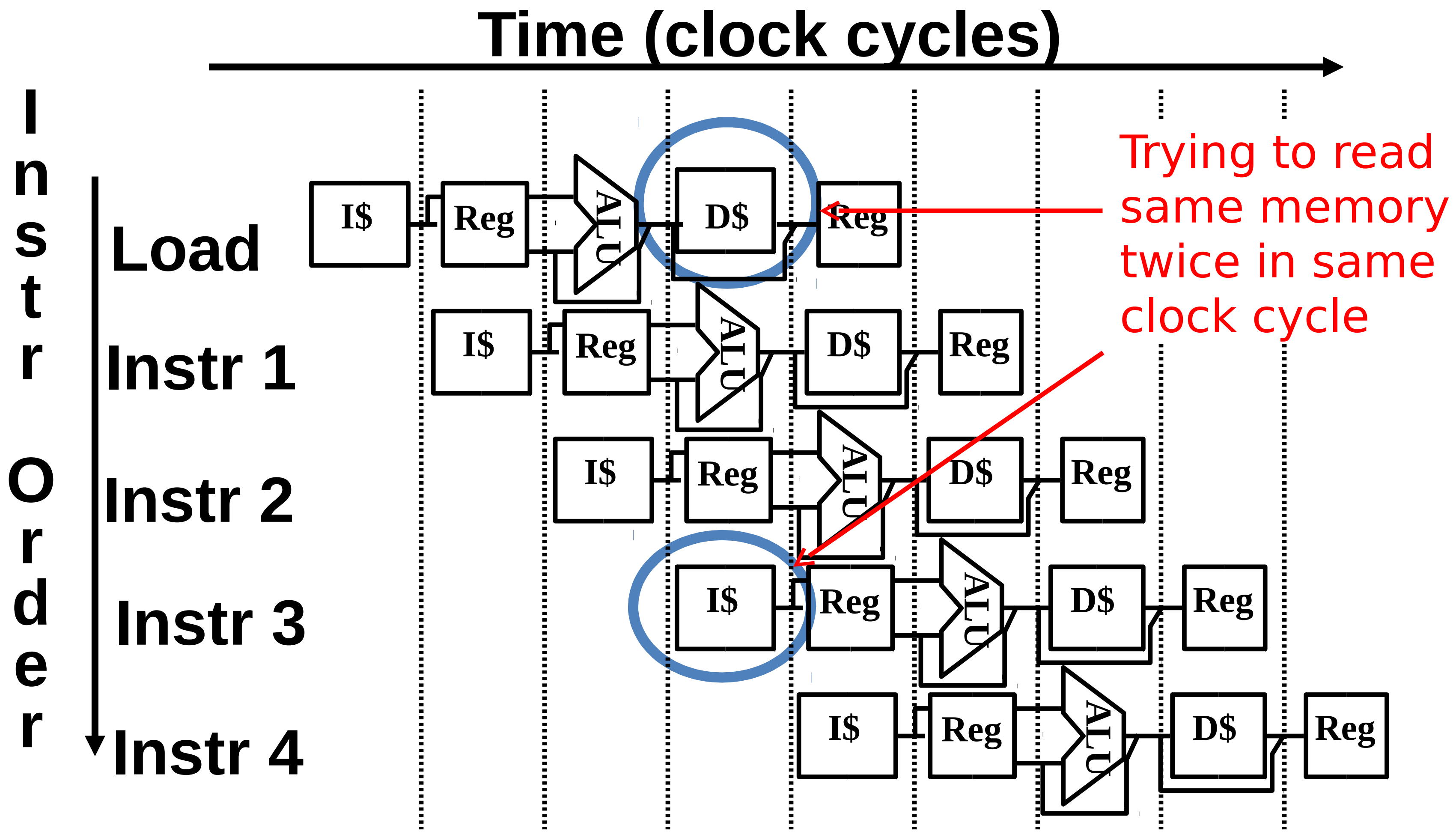
1. Structural Hazards

- Conflict for use of a resource



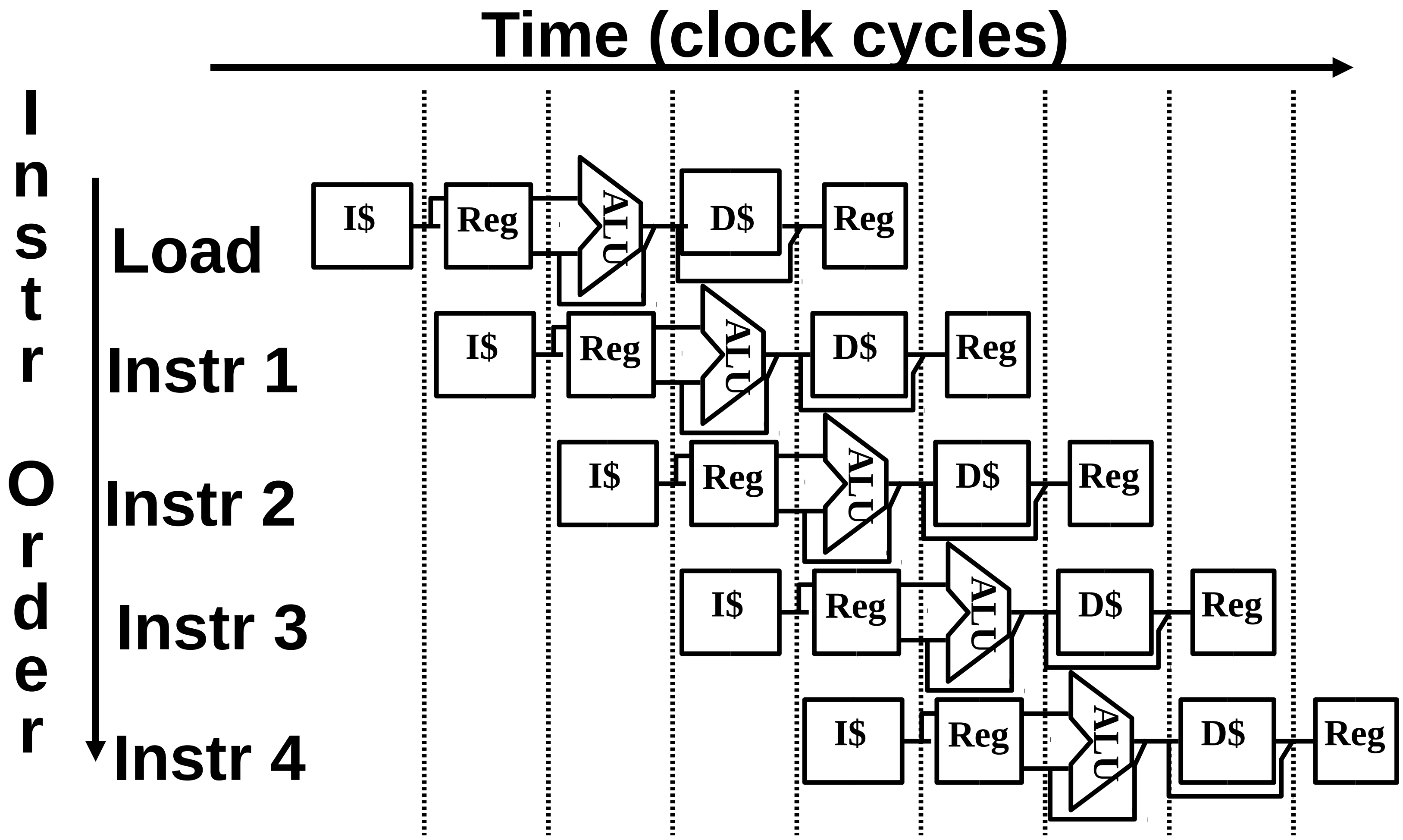
1. Structural Hazards

- Conflict for use of a resource



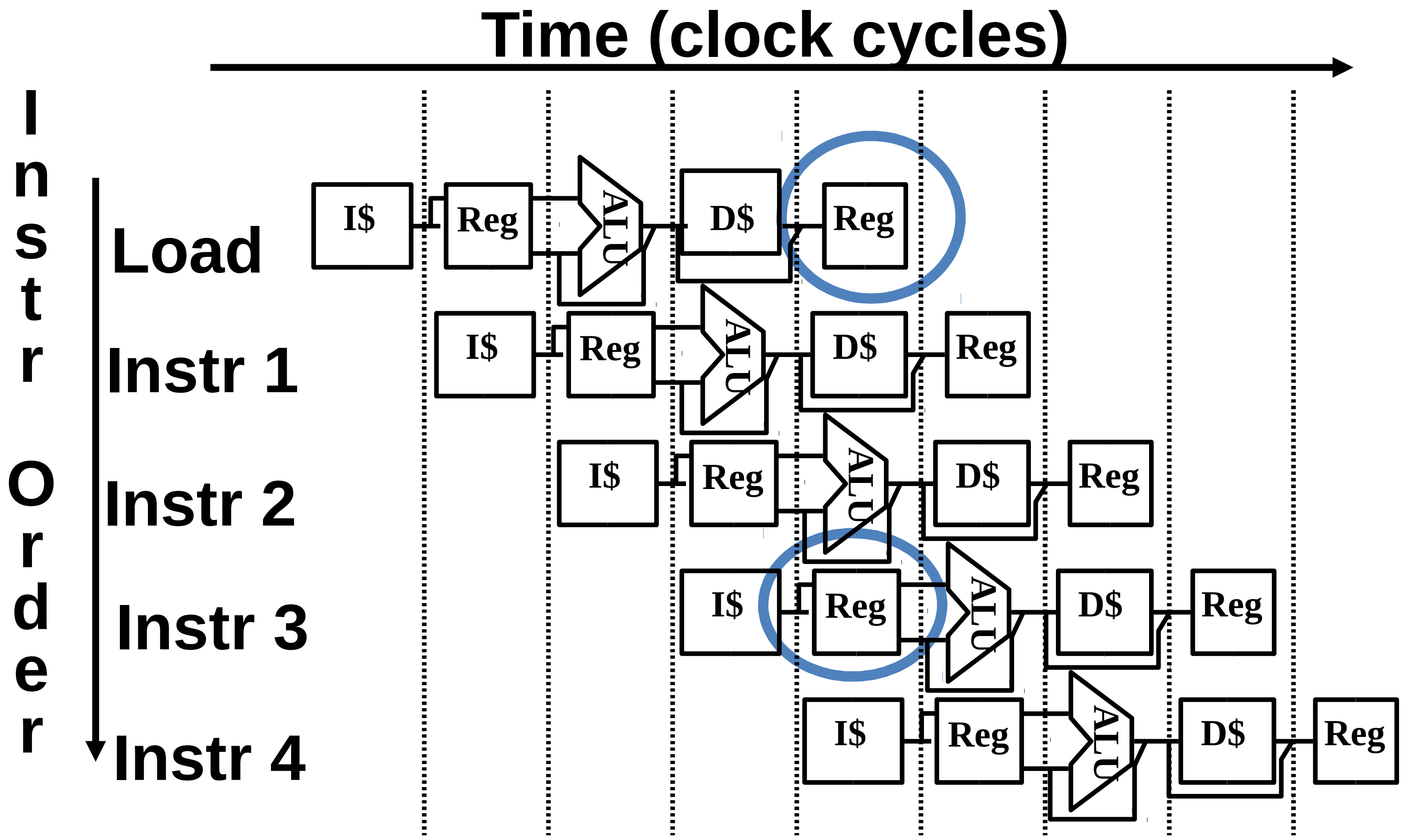
1. Structural Hazards

- Conflict for use of a resource



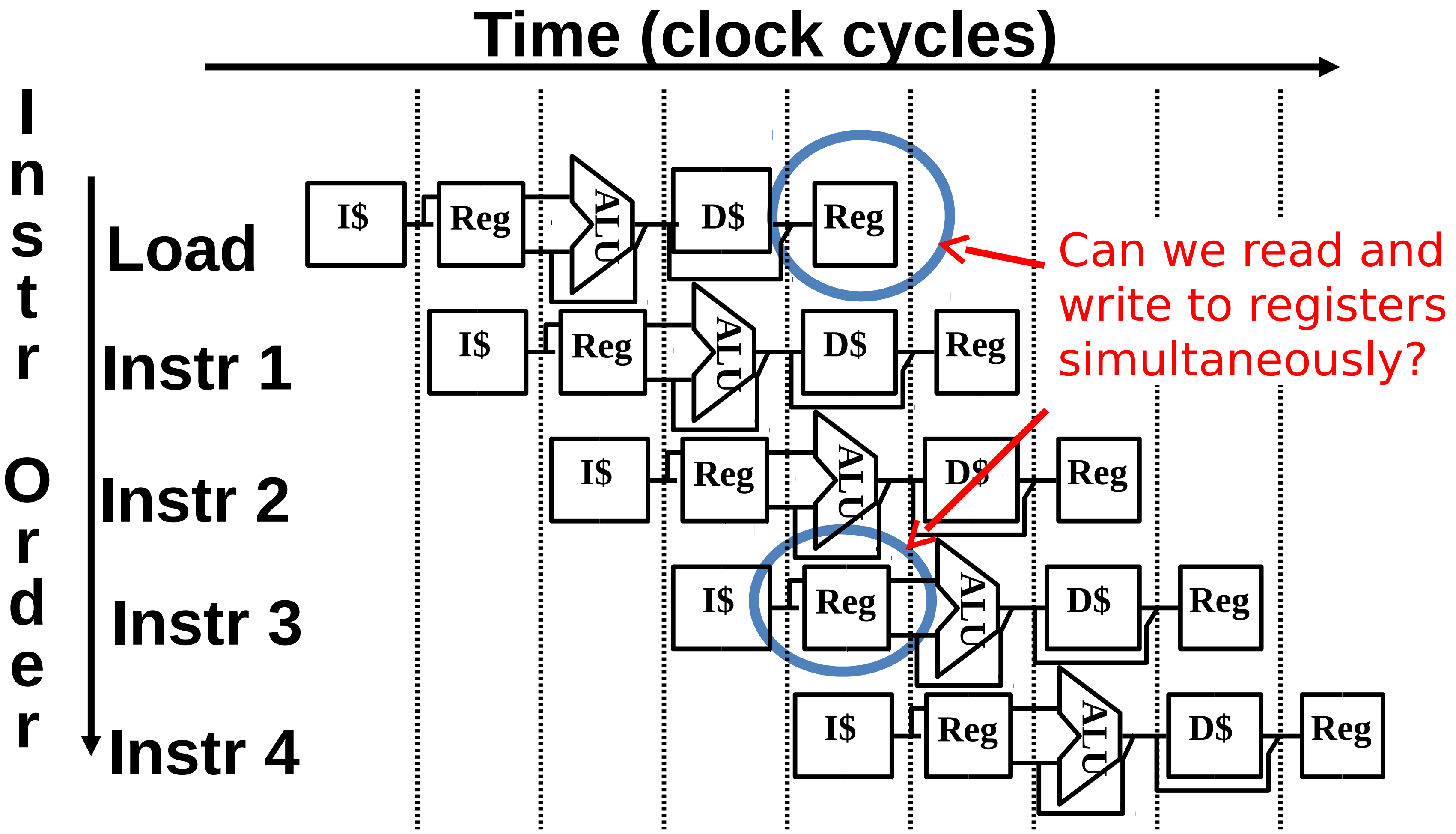
1. Structural Hazards

- Conflict for use of a resource



1. Structural Hazards

- Conflict for use of a resource



Structural Hazard #1: Single Memory

- 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*”

Structural Hazard #1: Single Memory

- 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
  - Separate L1 I$ and L1 D$ take care of this

Structural Hazard #2: Registers

- We use two solutions simultaneously:
  - 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)
  - Build RegFile with independent read and write port
- **Conclusion:** Read and Write to registers during same clock cycle is okay

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Administrivia
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

2. 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 (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
```

Stored during WB

2. Data Hazards (1/2)

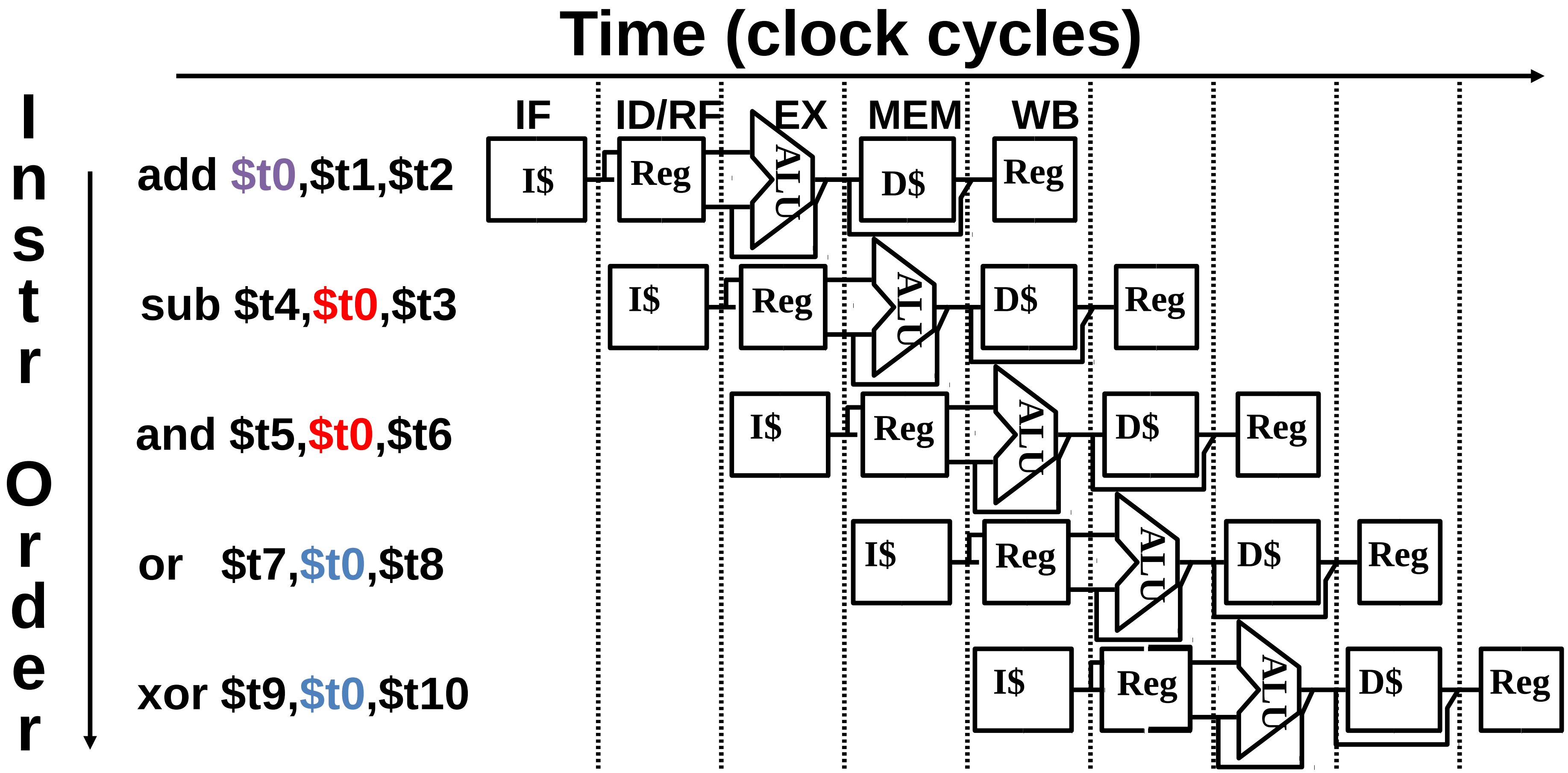
- Consider the following sequence of instructions:

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

$\underbrace{\texttt{\$t9}}_{\text{Stored during WB}}$, $\underbrace{\texttt{\$t0, \$t10}}_{\text{Read during ID}}$

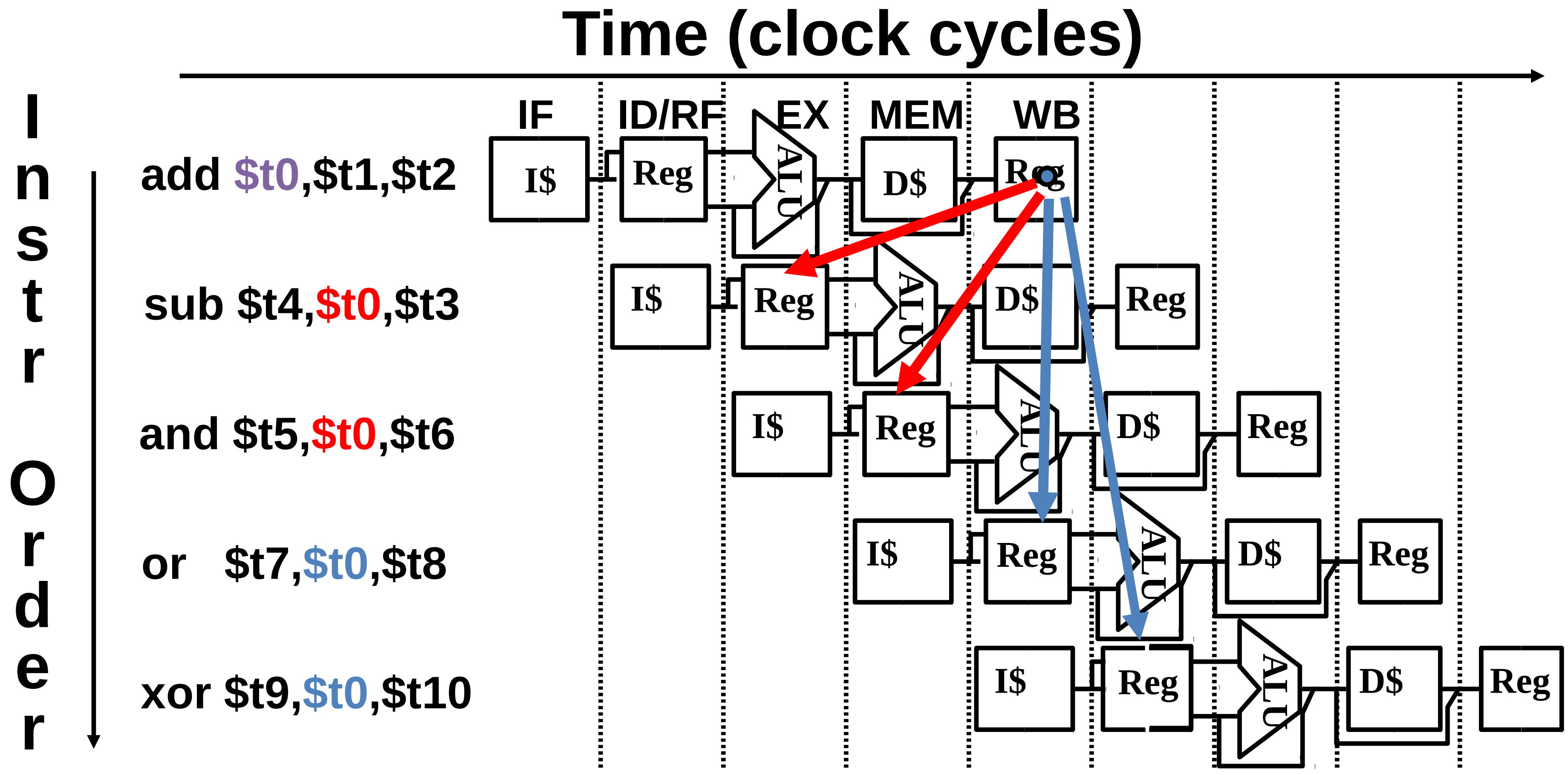
2. Data Hazards (2/2)

- Data-flow *backward* in time are hazards



2. Data Hazards (2/2)

- Data-flow *backward* 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

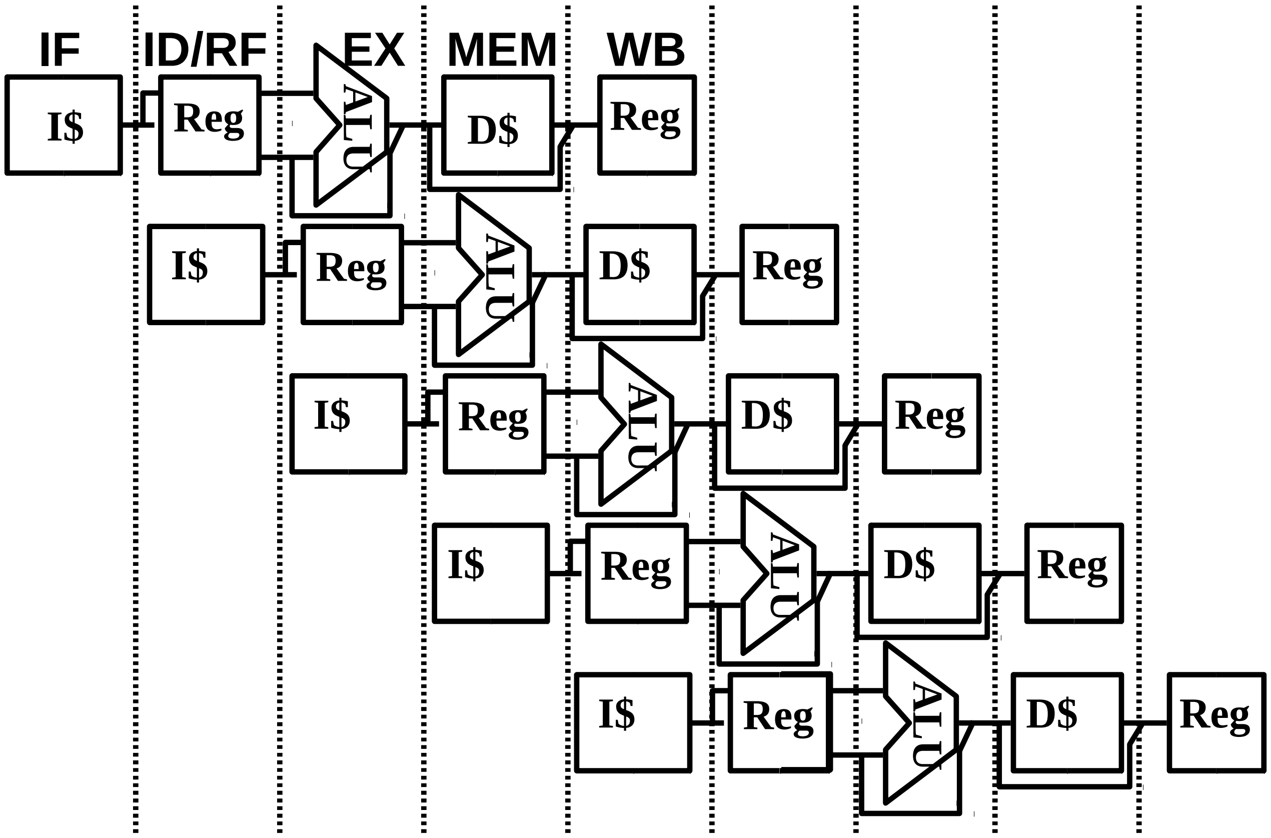
add **$t0**,$t1,$t2

sub $t4,**$t0**,$t3

and $t5,**$t0**,$t6

or $t7,**$t0**,$t8

xor $t9,**$t0**,$t10



Data Hazard Solution: Forwarding

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

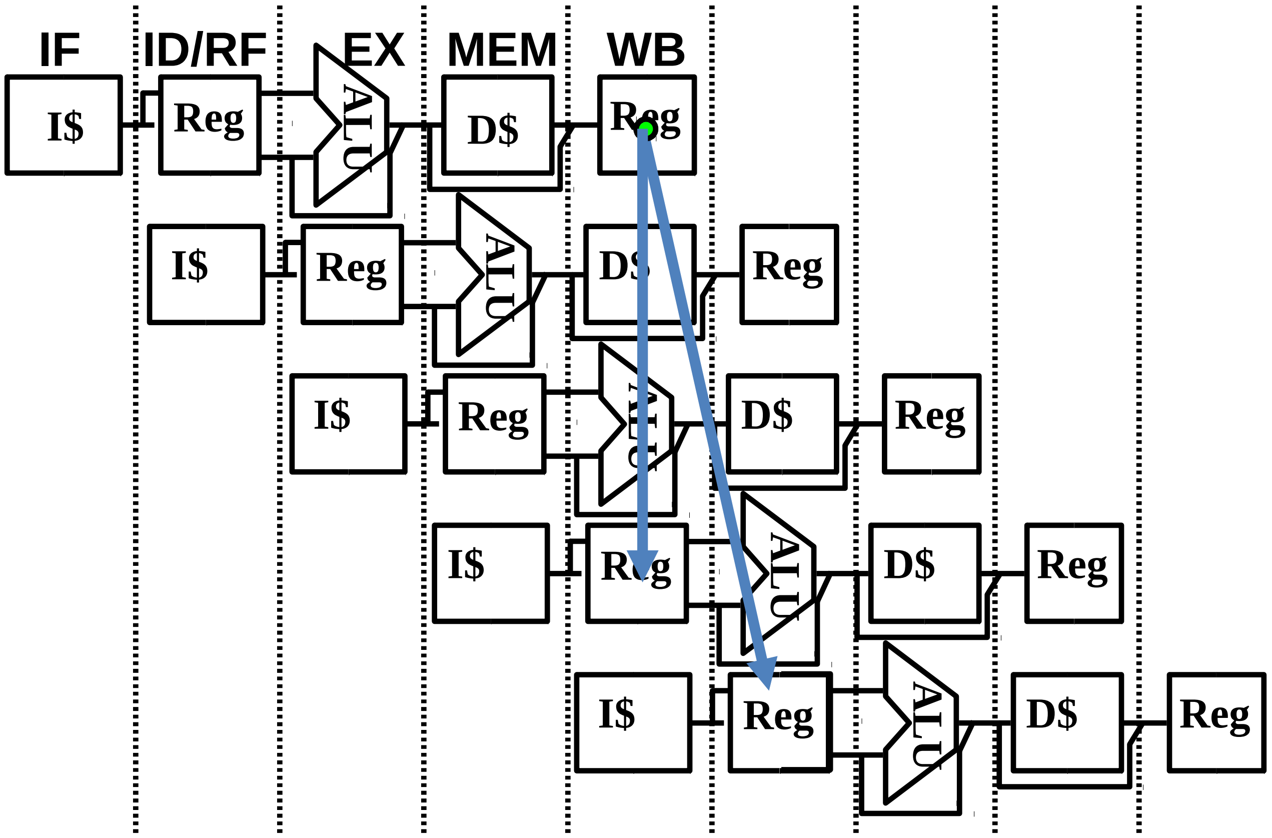
add $t0,$t1,$t2

sub $t4,$t0,$t3

and $t5,$t0,$t6

or $t7,$t0,$t8

xor $t9,$t0,$t10



Data Hazard Solution: Forwarding

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

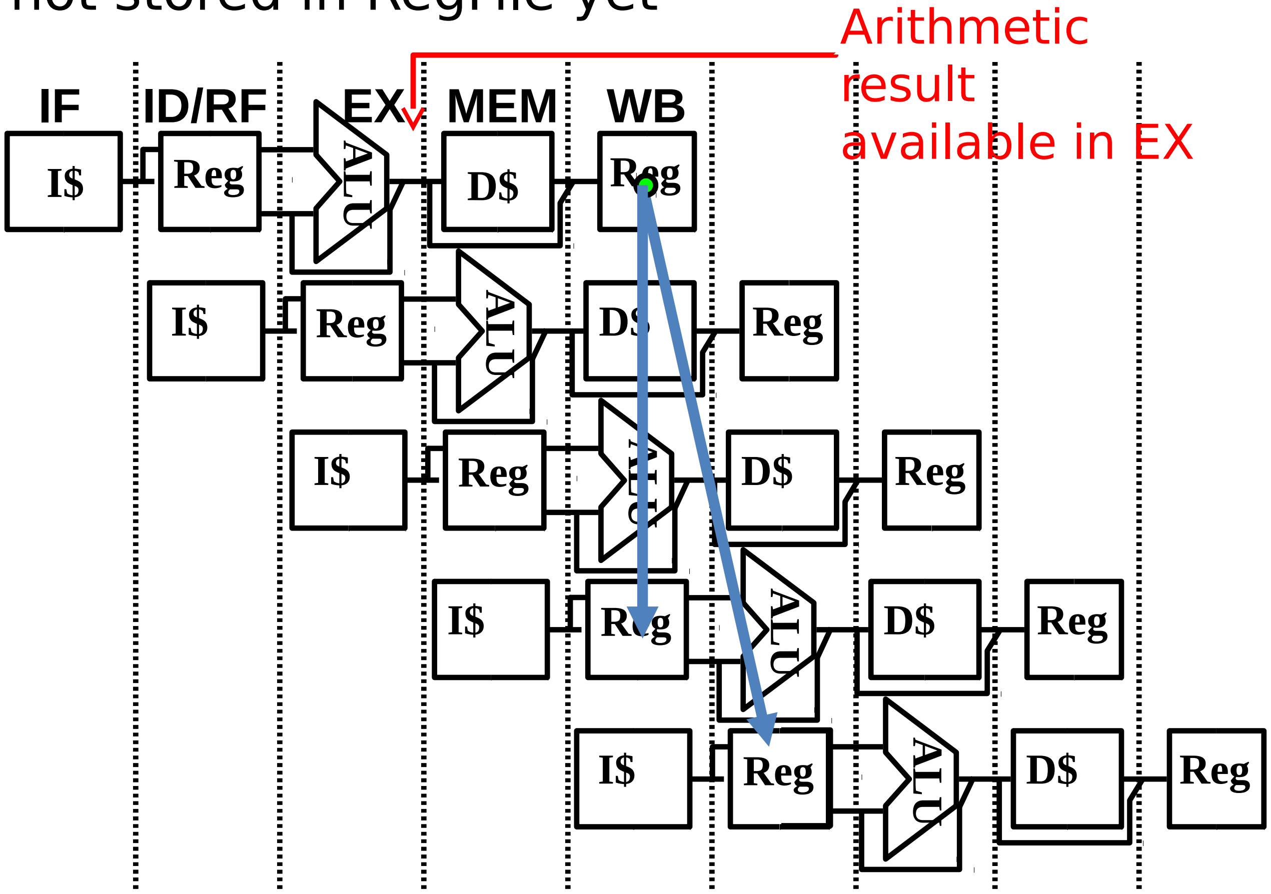
add $t0,$t1,$t2

sub $t4,$t0,$t3

and $t5,$t0,$t6

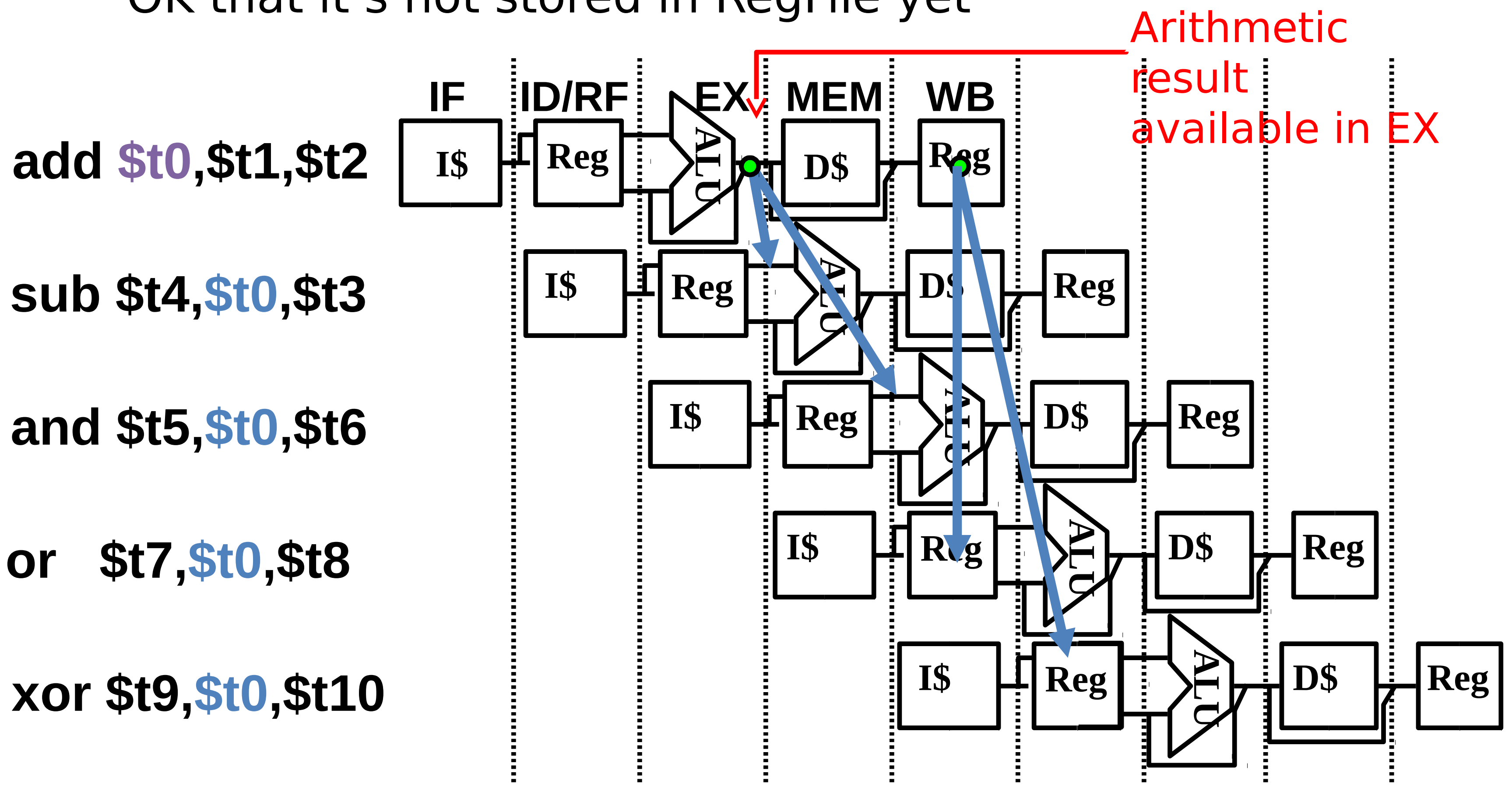
or $t7,$t0,$t8

xor $t9,$t0,$t10



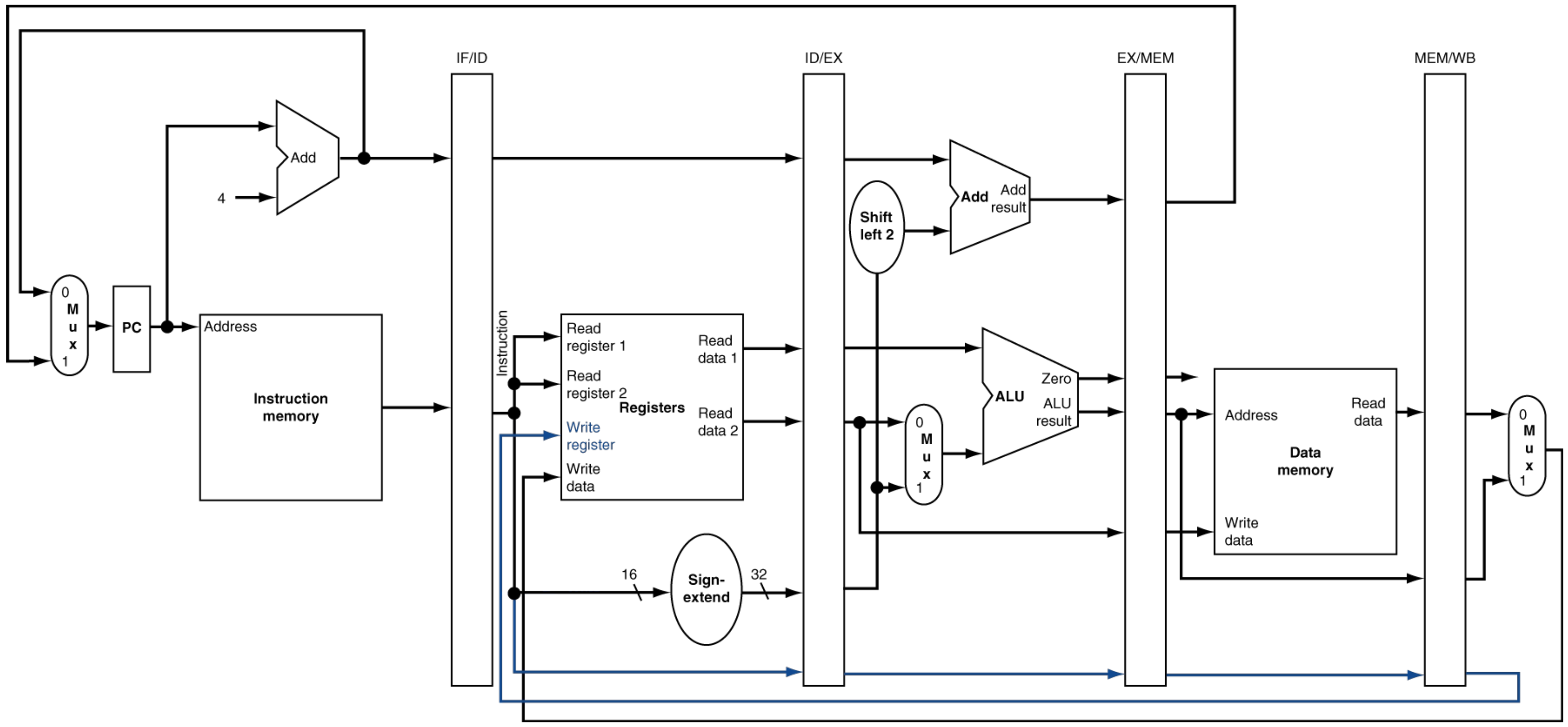
Data Hazard Solution: Forwarding

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



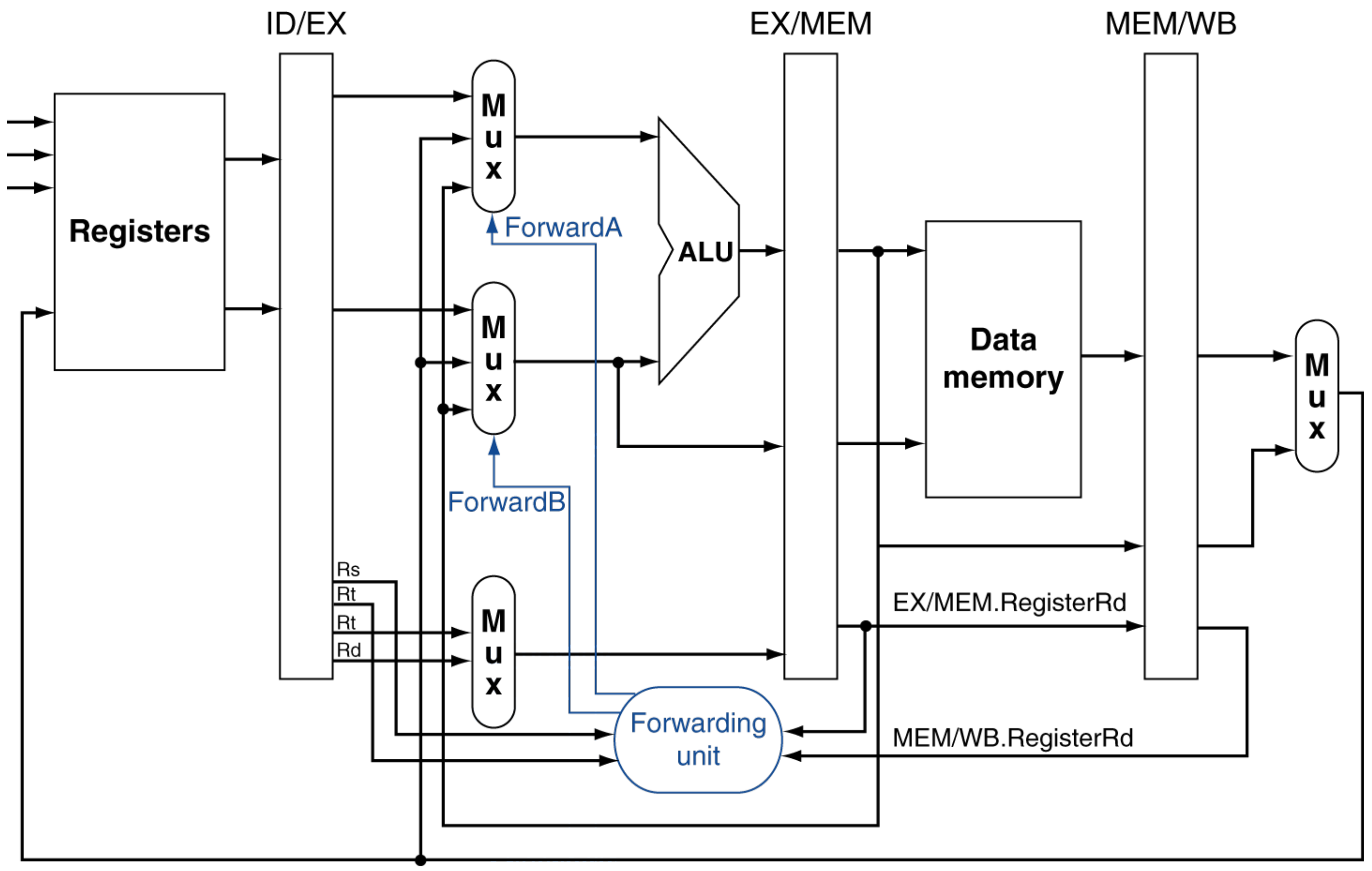
Datapath for Forwarding (1/2)

- What changes need to be made here?



Datapath for Forwarding (2/2)

- Handled by *forwarding unit*



Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Administrivia
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

Administrivia

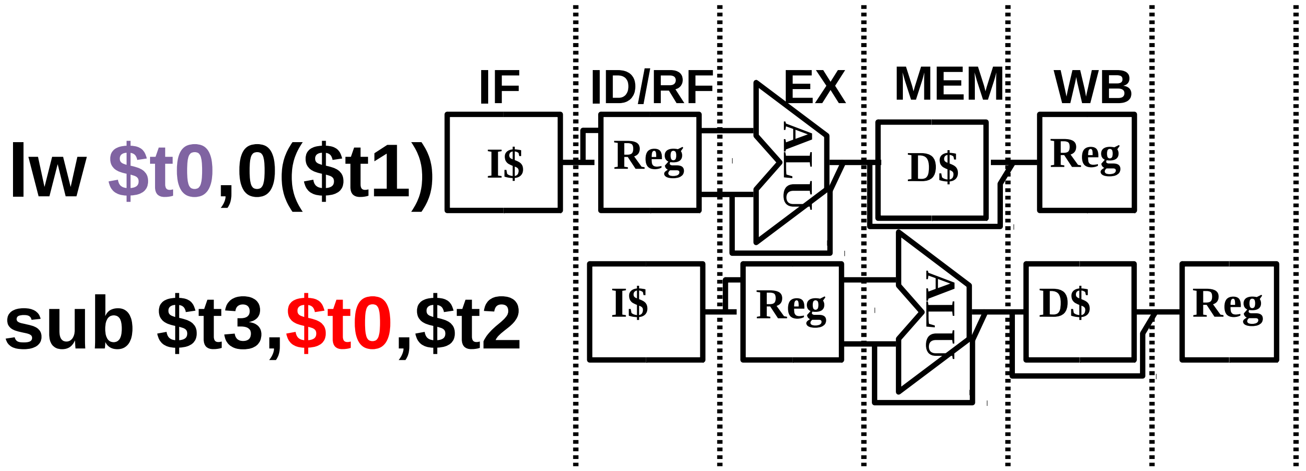
- HW 5 (still) due tomorrow
- Project 2 (still) due Sunday
  - Reduced lenience for botched submissions
  - Use the provided script to check your submission after submitting!
  - Make sure that only one partner has a submission on file, with his partner listed!

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Administrivia
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

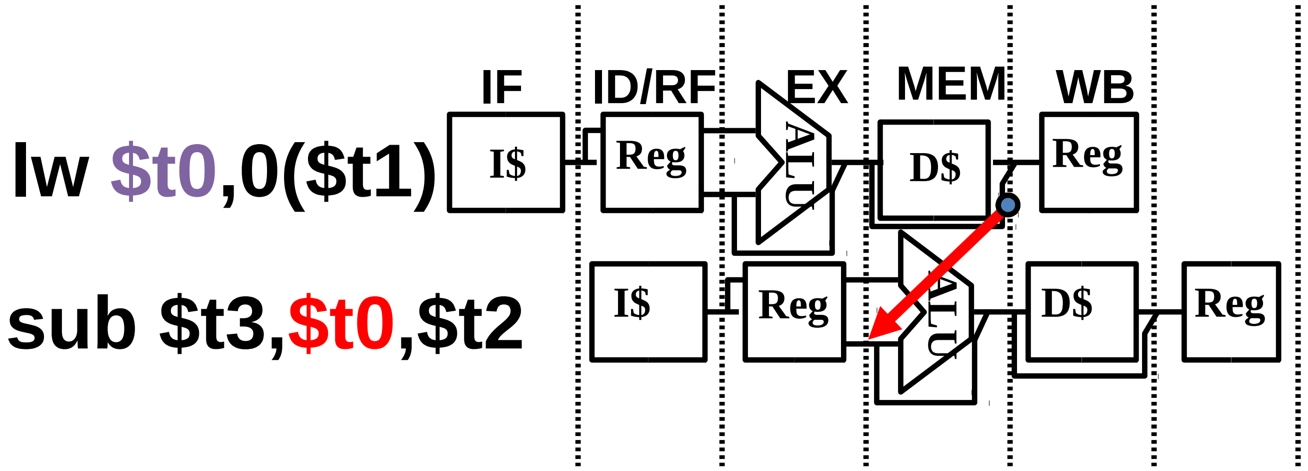
Data Hazard: Loads (1/4)

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



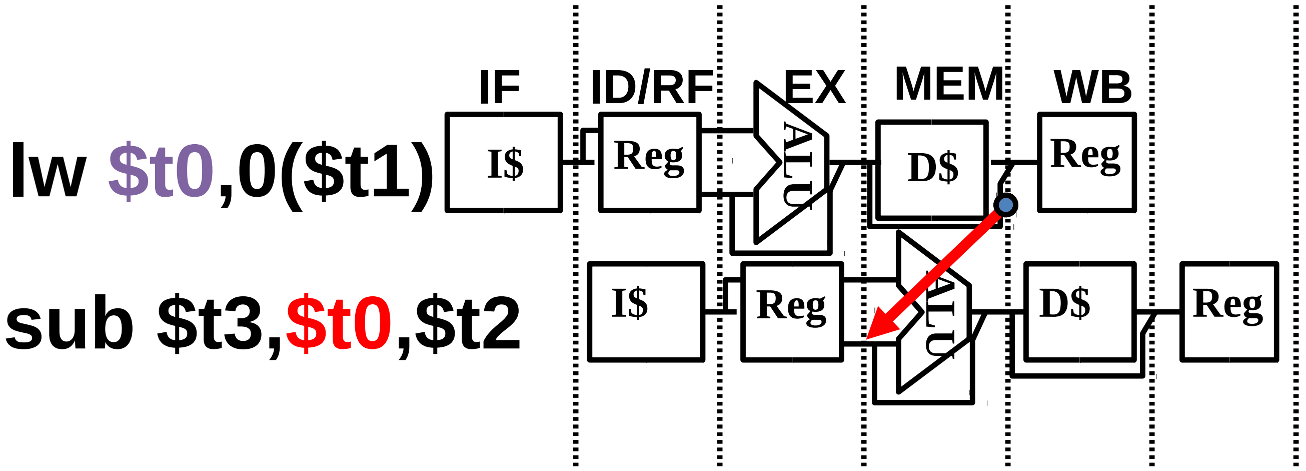
Data Hazard: Loads (1/4)

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



Data Hazard: Loads (1/4)

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



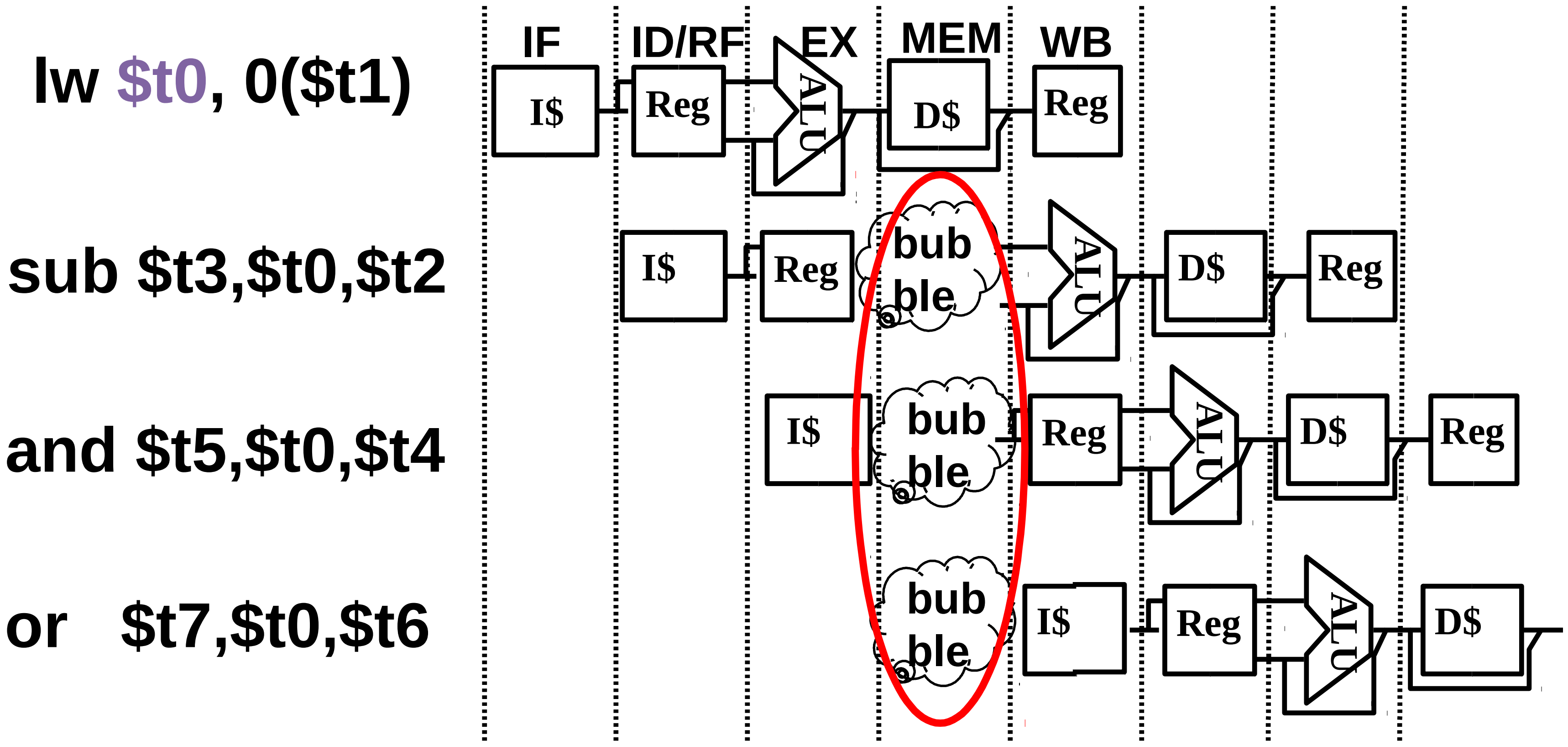
- Can't solve all cases with forwarding
  - Must *stall* instruction dependent on load, then forward (more hardware)

Data Hazard: Loads (2/4)

- *Hardware* stalls pipeline
  - Called “hardware interlock”

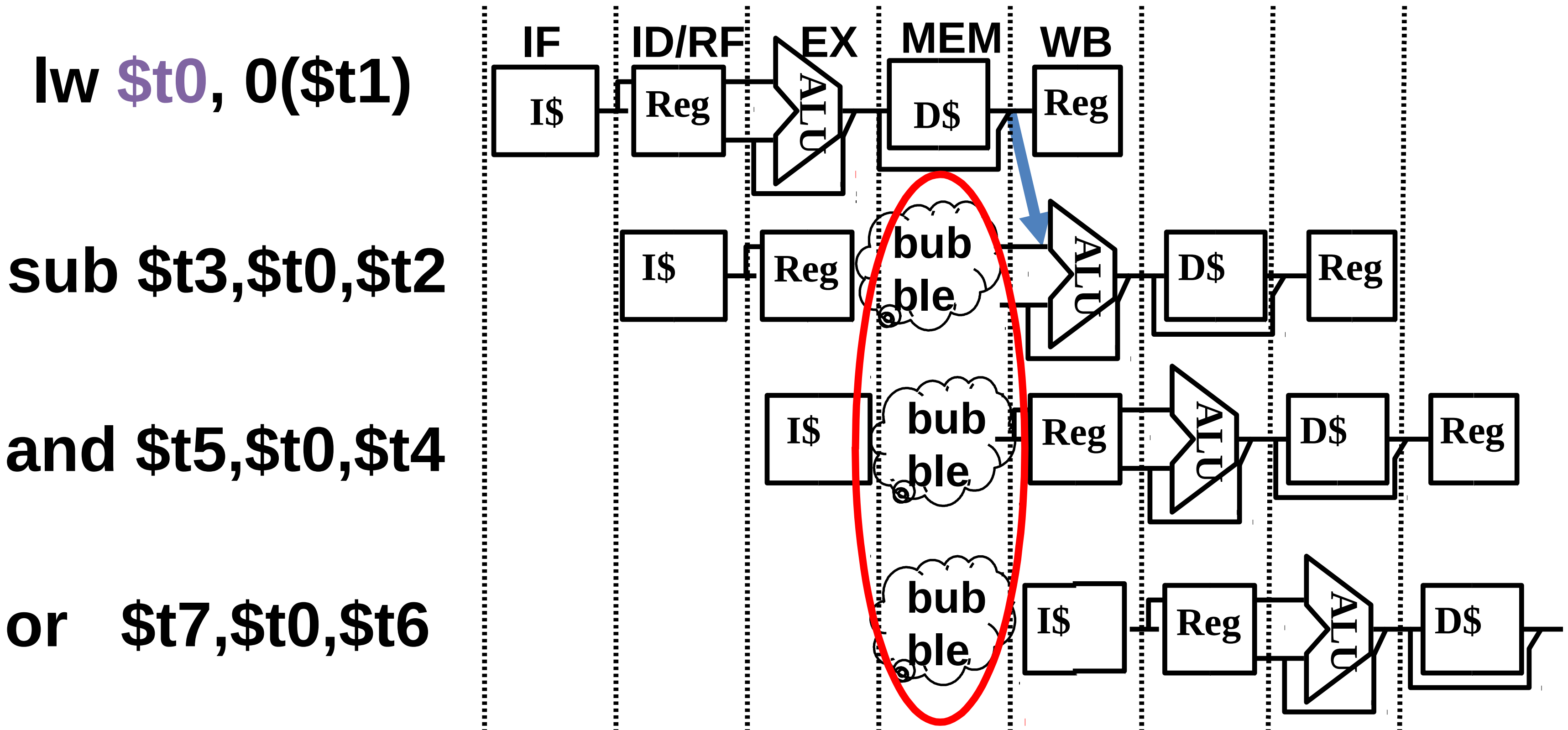
Data Hazard: Loads (2/4)

- *Hardware* stalls pipeline
  - Called “hardware interlock”



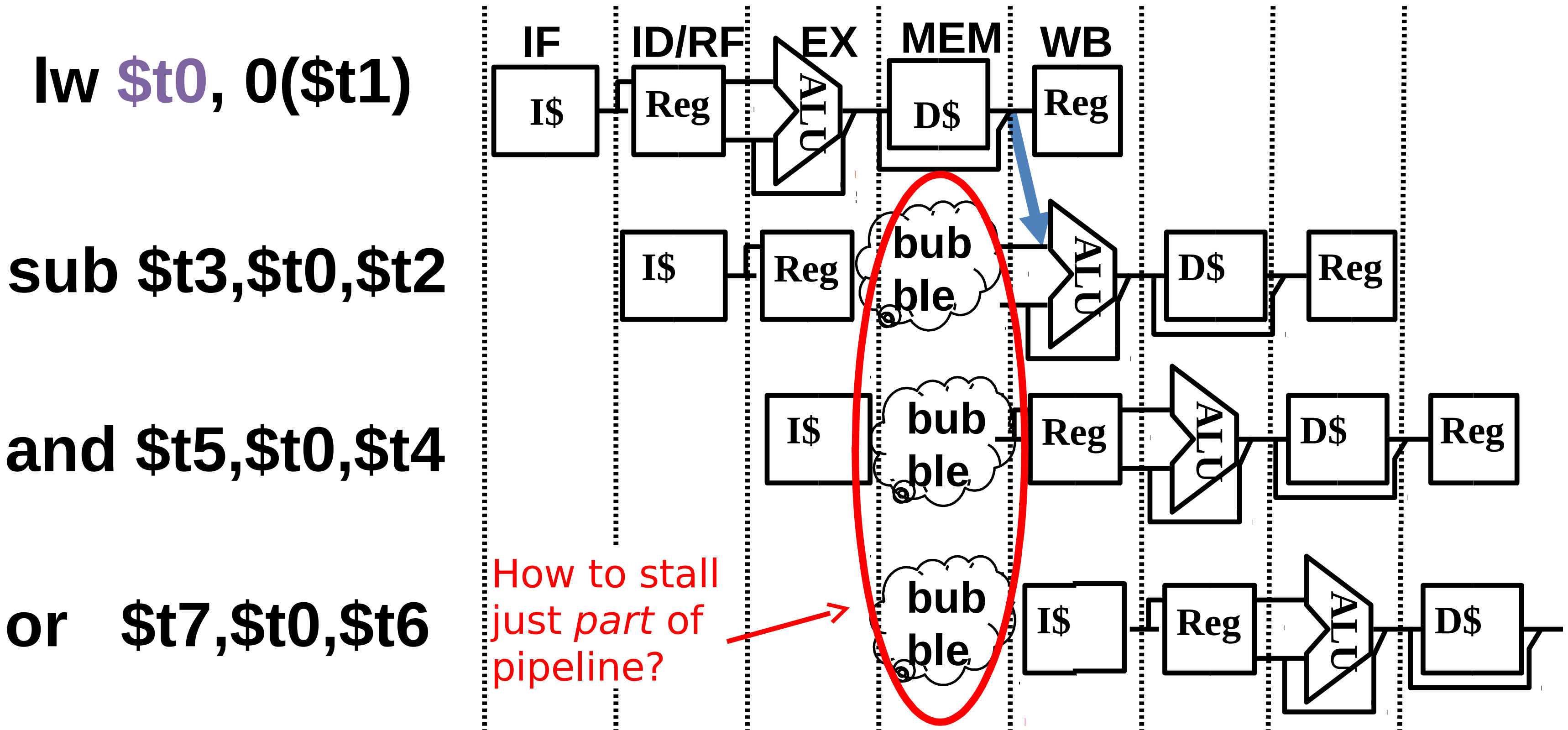
Data Hazard: Loads (2/4)

- *Hardware* stalls pipeline
  - Called “hardware interlock”



Data Hazard: Loads (2/4)

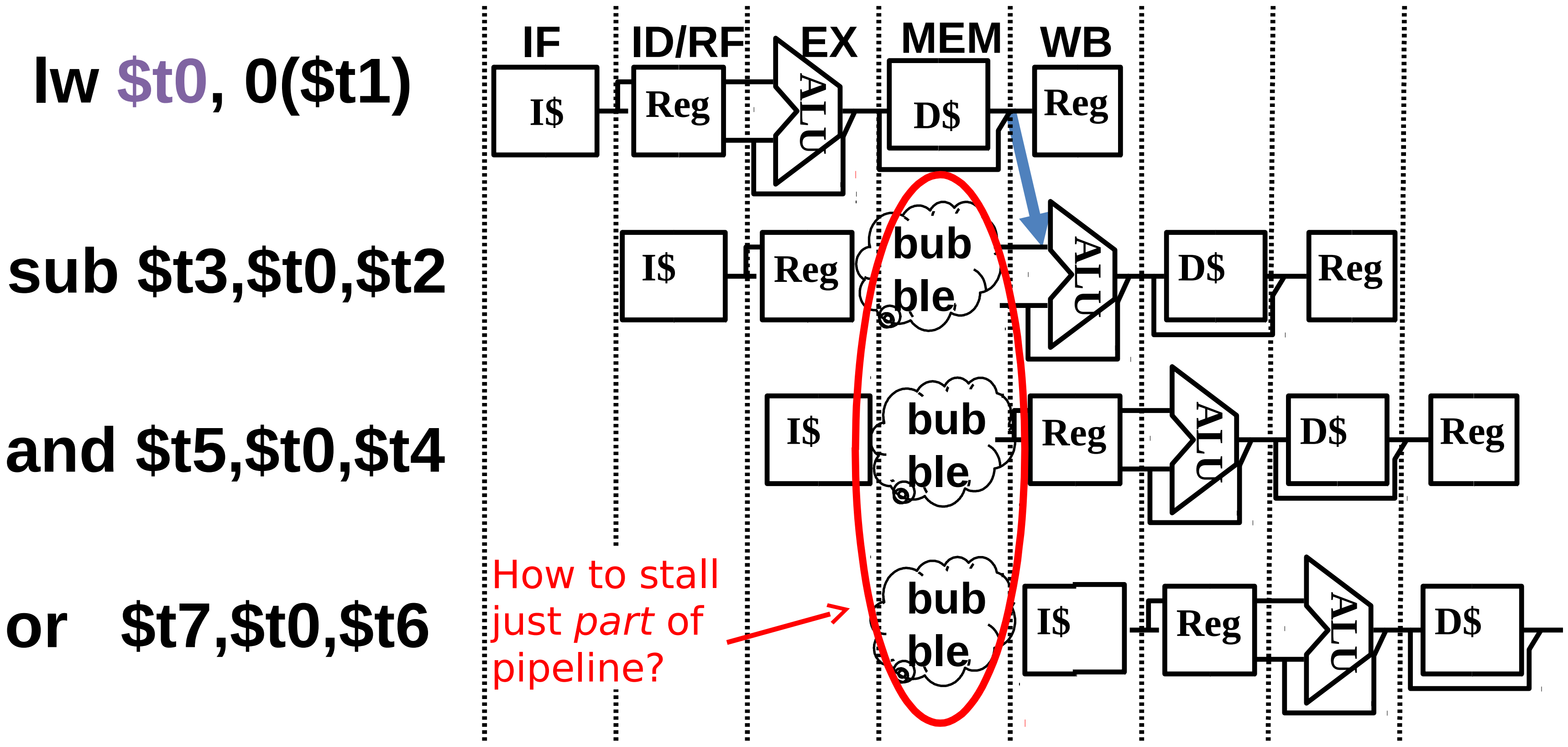
- *Hardware* stalls pipeline
  - Called "hardware interlock"



Data Hazard: Loads (2/4)

- *Hardware* stalls pipeline
  - Called "hardware interlock"

Schematically, this is what we want, but in reality stalls done "horizontally"



Data Hazard: Loads (3/4)

- Stall is equivalent to `nop`

Data Hazard: Loads (3/4)

- Stall is equivalent to nop

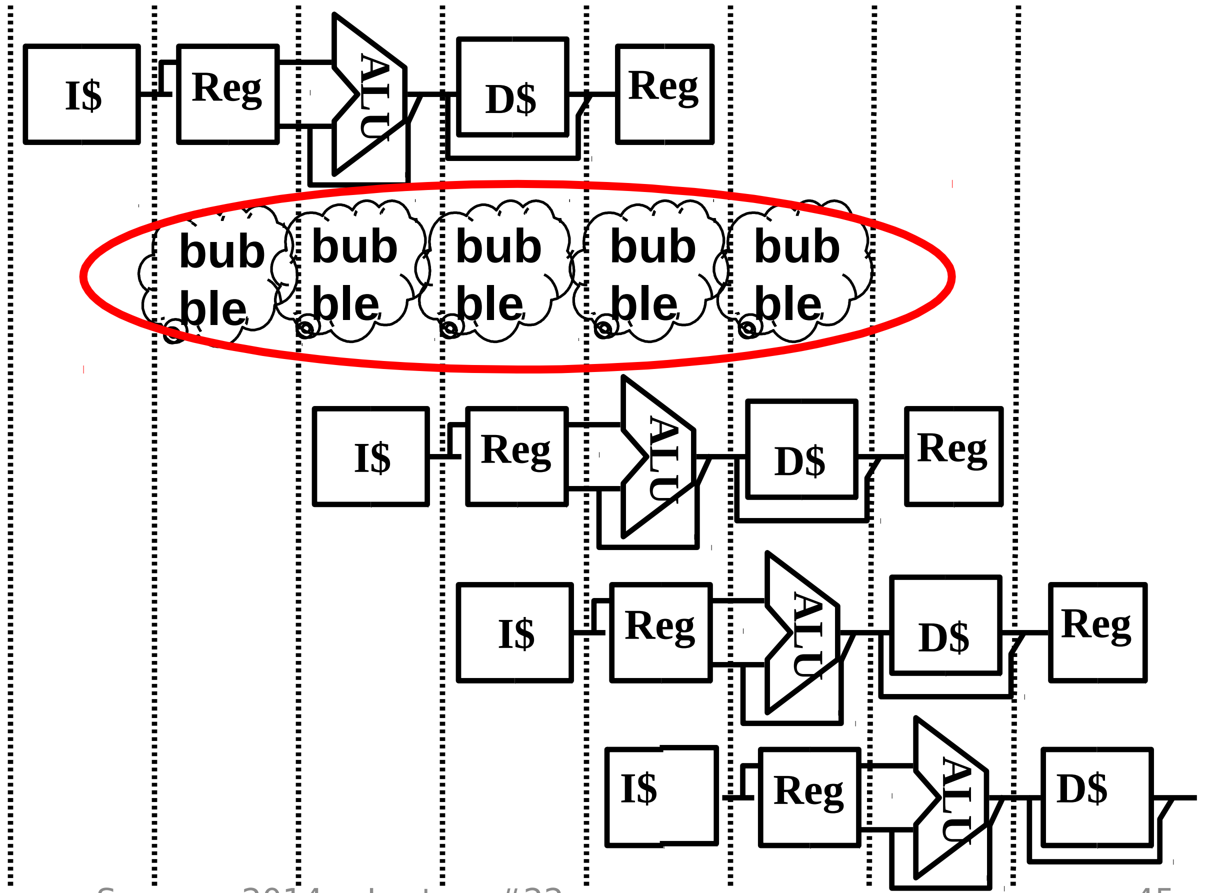
lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



Data Hazard: Loads (3/4)

- Stall is equivalent to nop

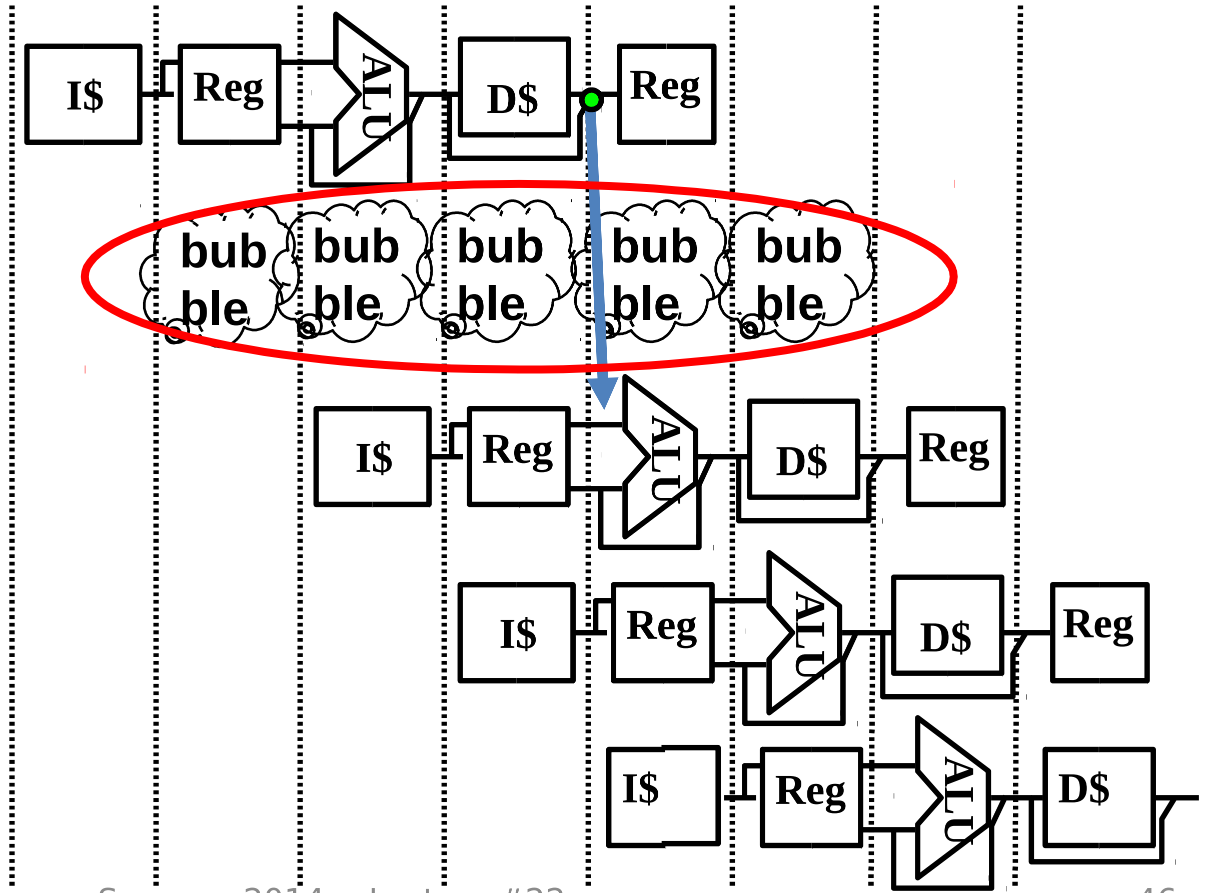
lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



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

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 a nop in the slot (except the latter uses more code space)

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 a nop in the slot (except the latter uses more code space)

Idea: Let the compiler put an unrelated instruction in that slot → no stall!

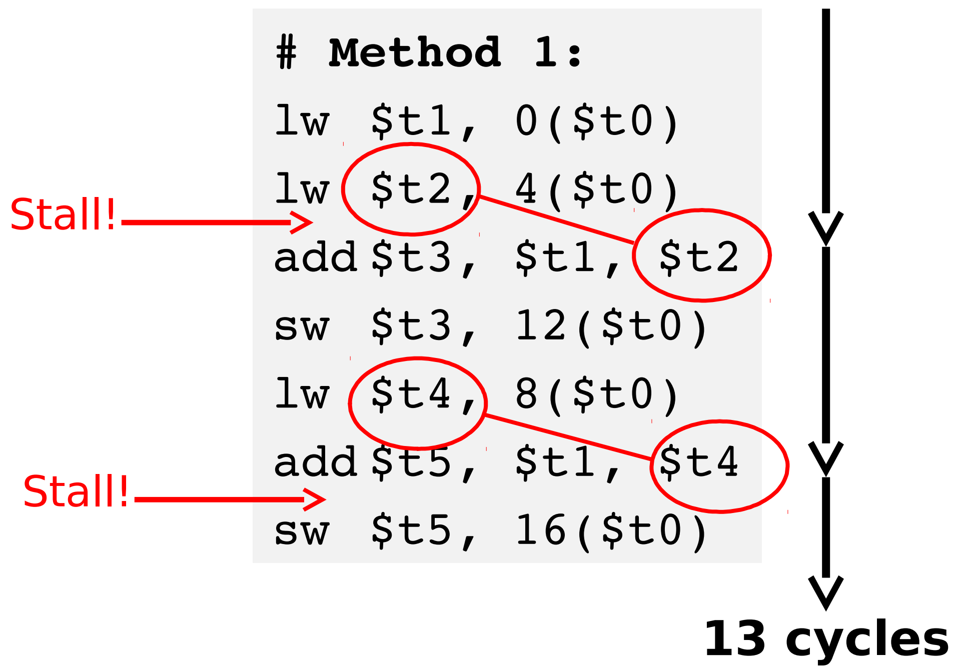
Code Scheduling to Avoid Stalls

- 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)
```

Code Scheduling to Avoid Stalls

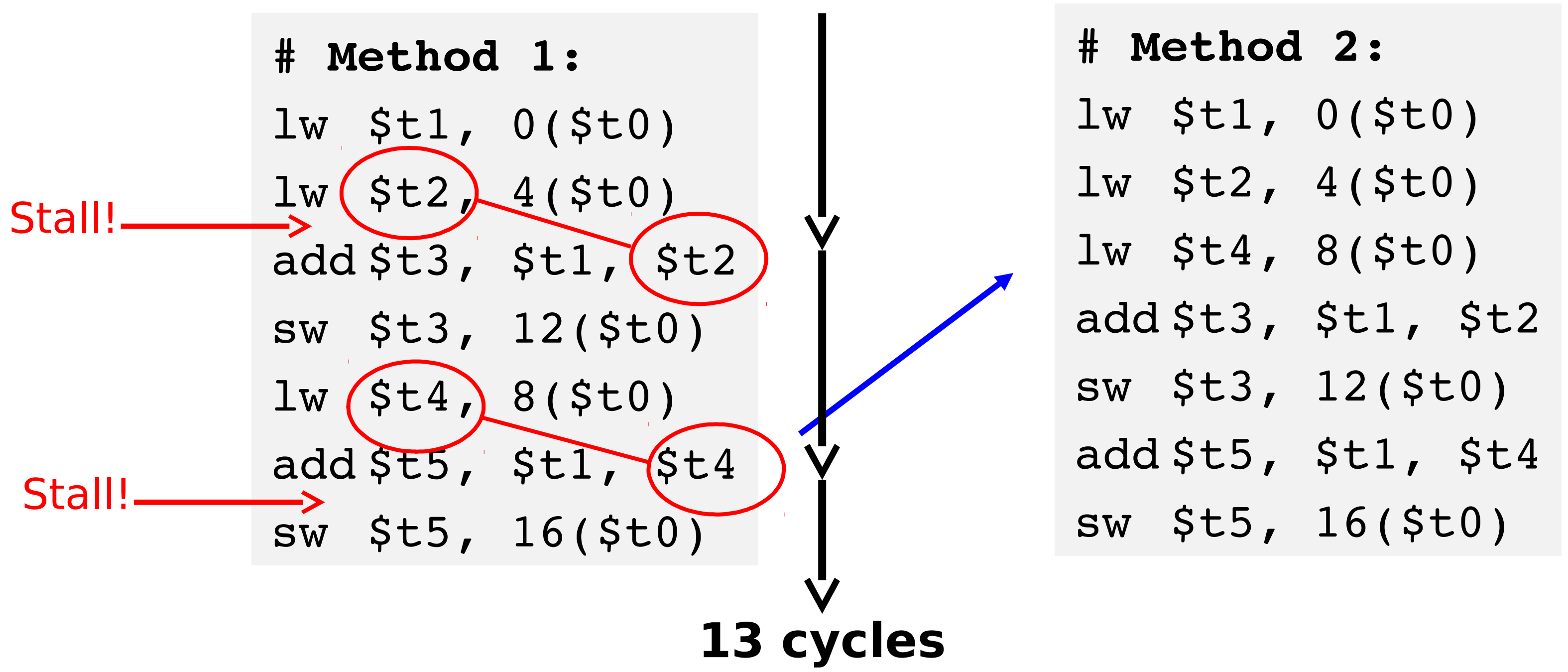
- Reorder code to avoid use of load result in the next instruction!
- MIPS code for `D=A+B; E=A+C;`



```
# 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)
```

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;`

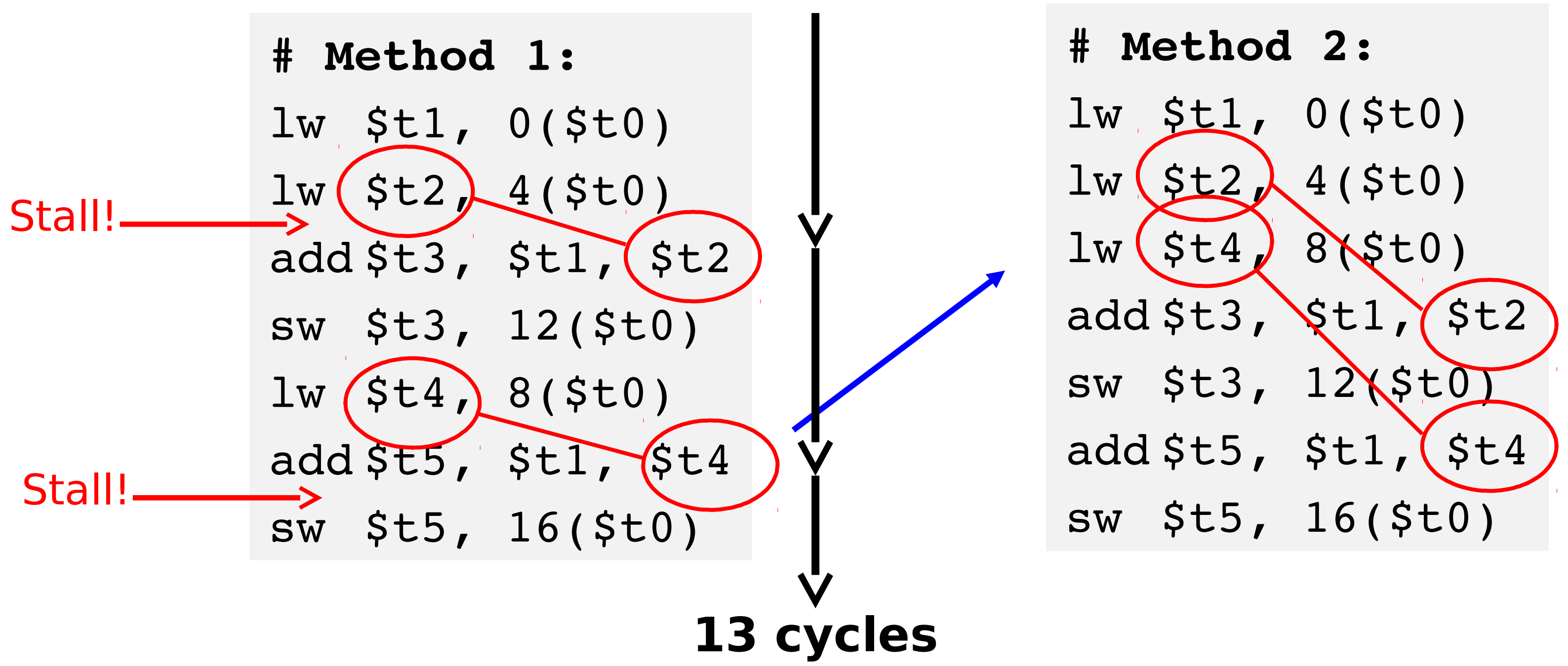


```
# Method 1:
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

```
# 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)
```

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;`

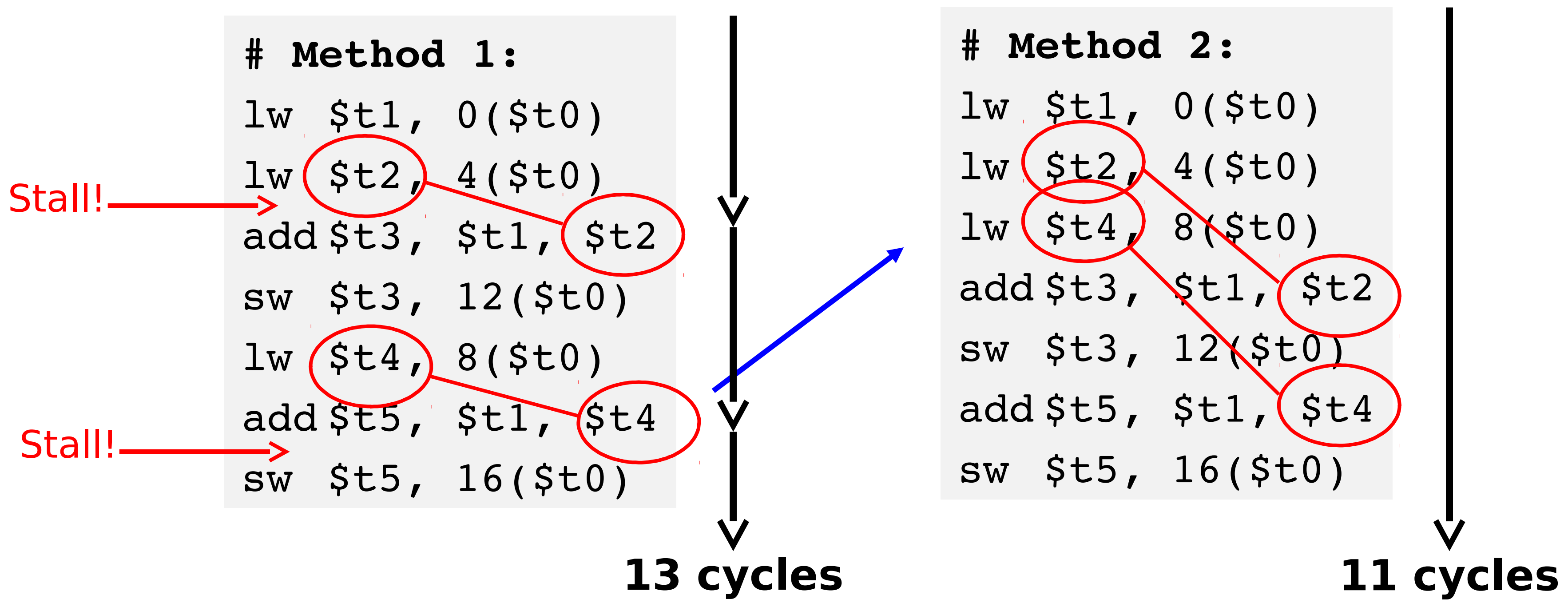


```
# Method 1:
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

```
# 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)
```

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)
```

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Administrivia
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

3. Control Hazards

- Branch (beq, bne) determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction

3. Control Hazards

- Branch (beq, bne) 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

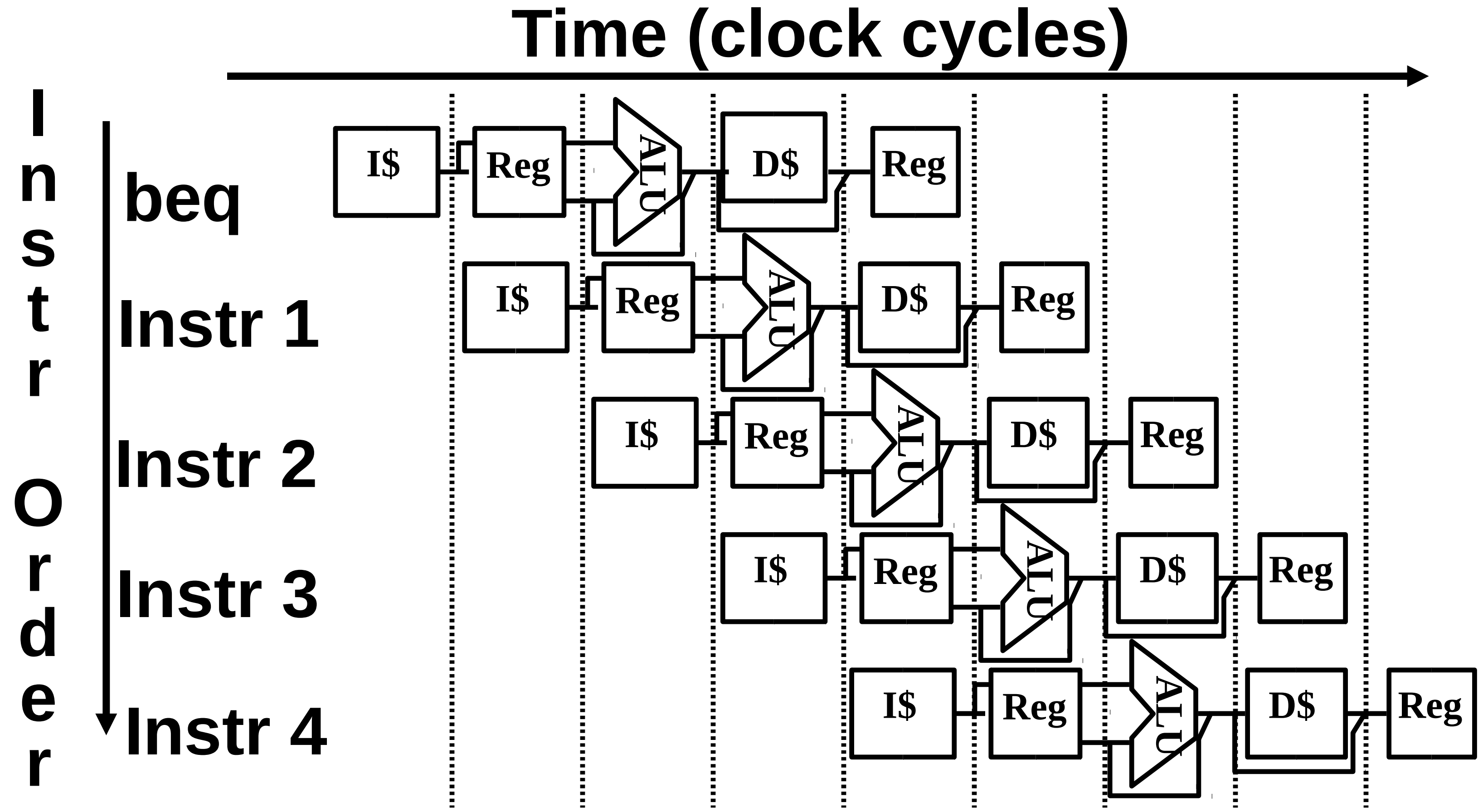
3. Control Hazards

- Branch (beq, bne) 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

- **Simple Solution:** Stall on *every* branch until we have the new PC value
  - How long must we stall?

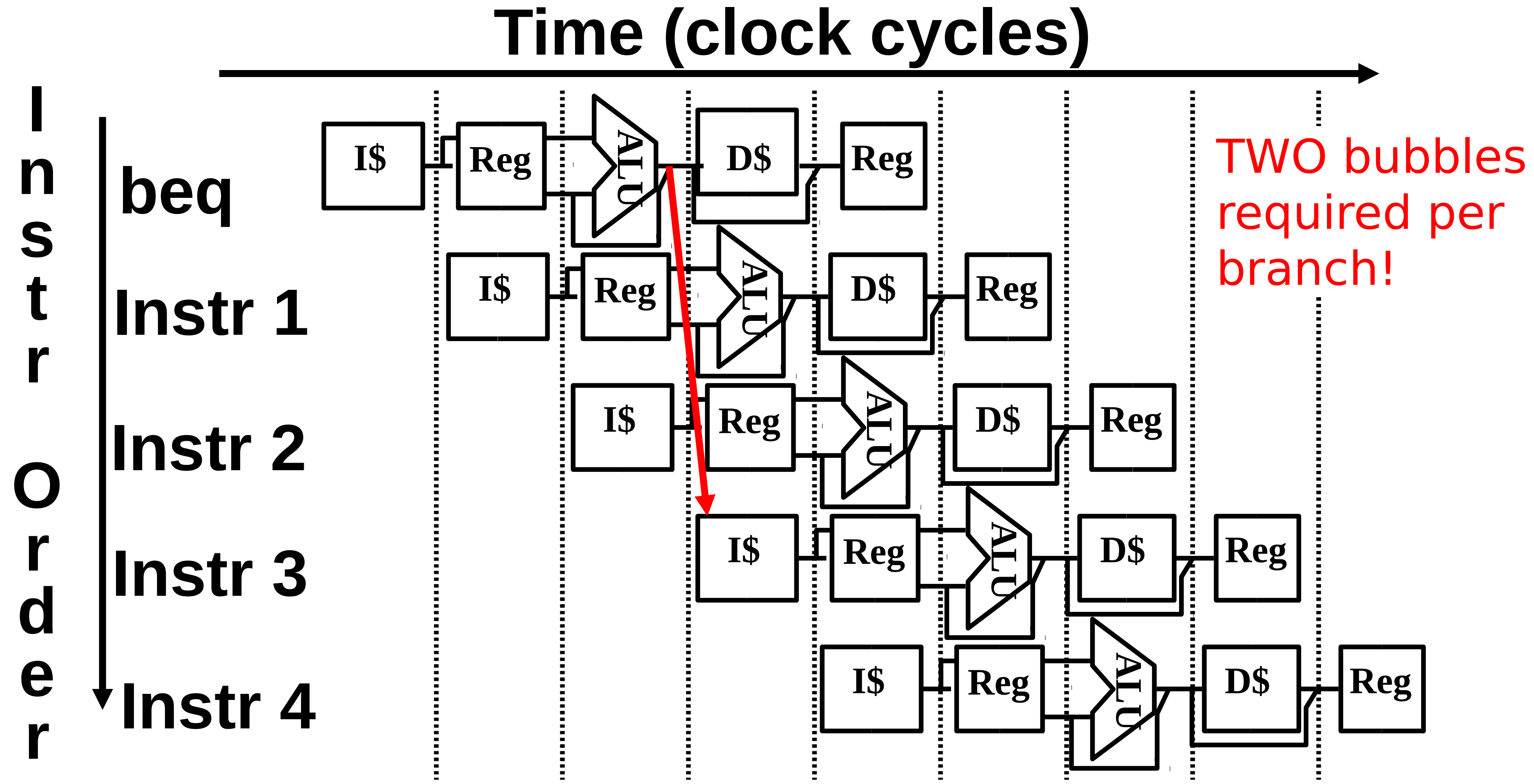
Branch Stall

- When is comparison result available?



Branch Stall

- When is comparison result available?



3. Control Hazard: Branching

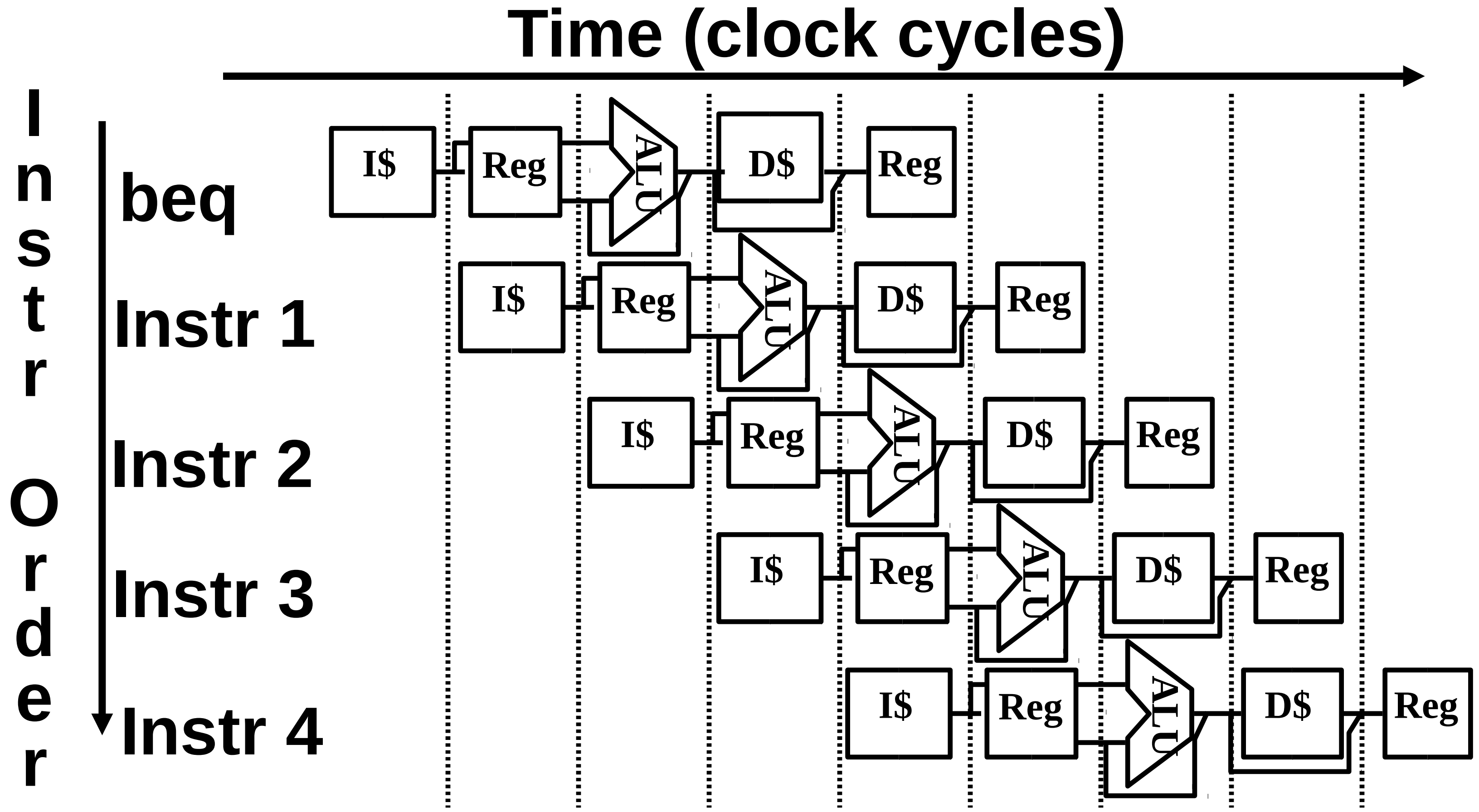
- **Option #1:** Insert special branch comparator in ID stage
  - As soon as instruction is decoded, immediately make a decision and set the new value of the PC

3. Control Hazard: Branching

- **Option #1:** Insert special branch comparator in ID stage
  - As soon as instruction is decoded, immediately make a decision and set the new value of the PC
  - **Benefit:** Branch decision made in 2nd stage, so only one nop is needed instead of two
  - **Side Note:** This means that branches are idle in EX, MEM, and WB

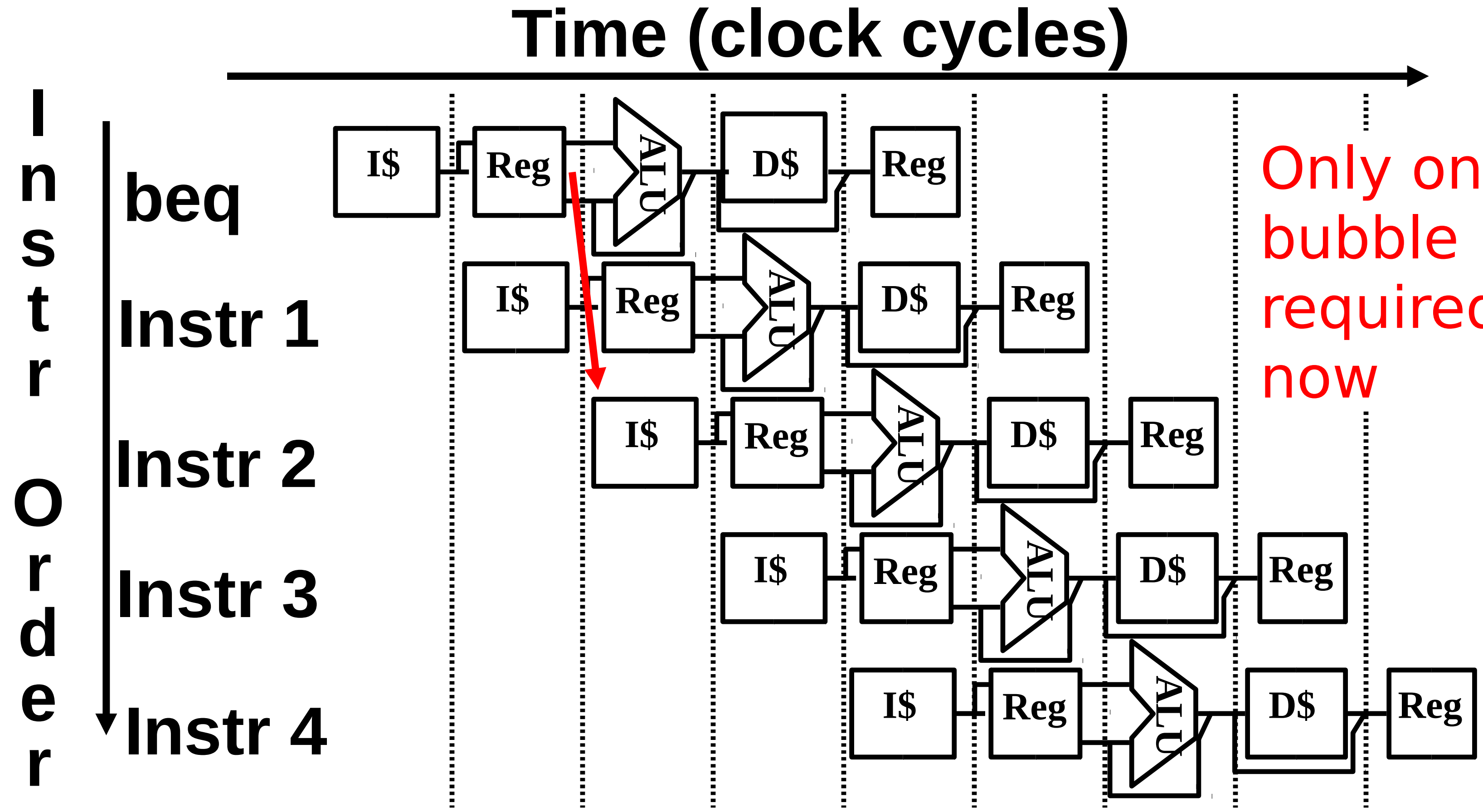
Improved Branch Stall

- When is comparison result available?



Improved Branch Stall

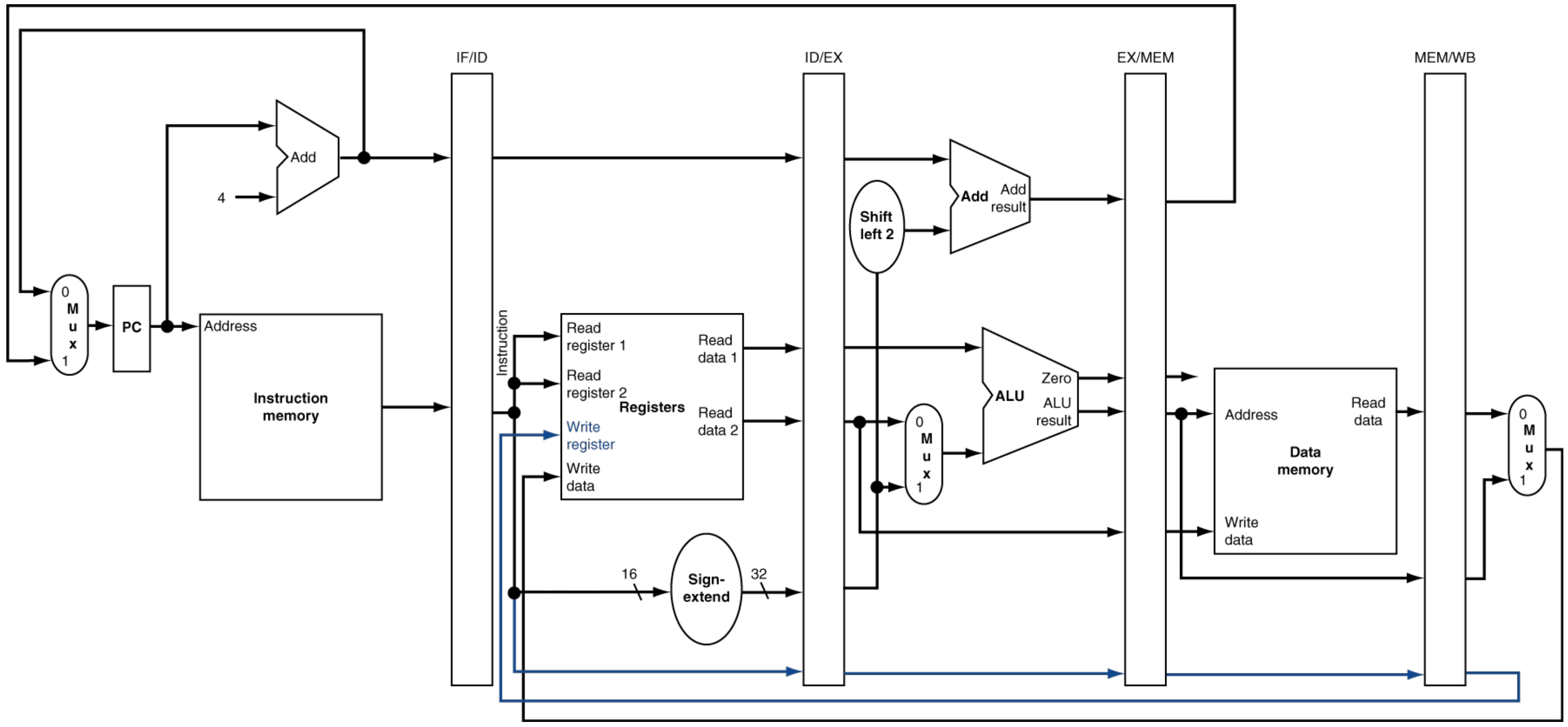
- When is comparison result available?



Only one bubble required now

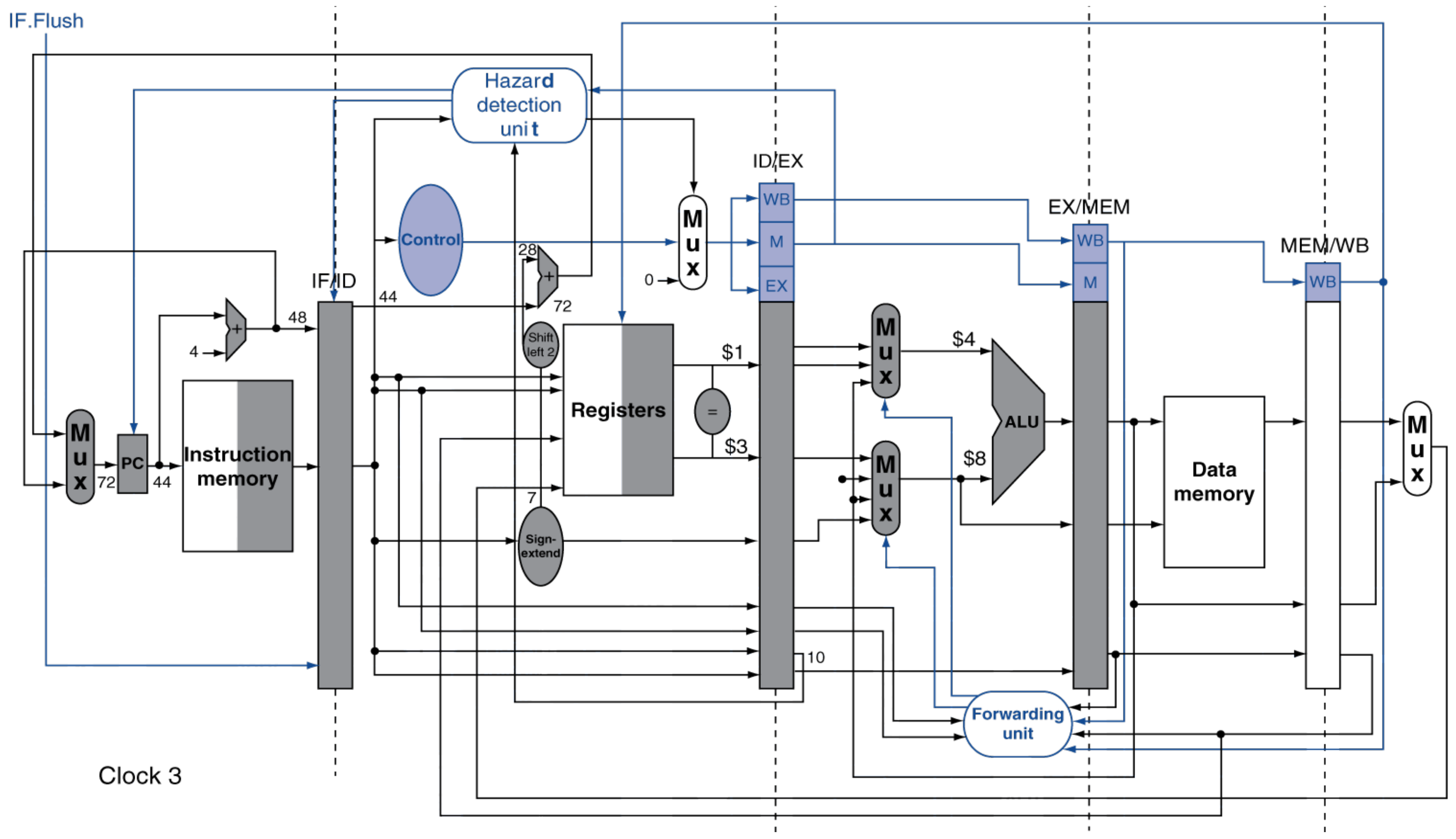
Datapath for ID Branch Comparator

- What changes need to be made here?



Datapath for ID Branch Comparator

- Handled by *hazard detection unit*



3. Control Hazard: Branching

- **Option #2:** *Branch Prediction* – guess outcome of a branch, fix afterwards if necessary

3. Control Hazard: Branching

- **Option #2:** *Branch Prediction* – guess outcome of a branch, fix afterwards if necessary
  - Must cancel (*flush*) all instructions in pipeline that depended on guess that was wrong
  - How many instructions do we end up flushing?

3. Control Hazard: Branching

- **Option #2:** *Branch Prediction* – guess outcome of a branch, fix afterwards if necessary
  - Must cancel (*flush*) all instructions in pipeline that depended on guess that was wrong
  - How many instructions do we end up flushing?

- Achieve simplest hardware if we predict that all branches are NOT taken

3. Control Hazard: Branching

- **Option #3:** *Branch delay slot*
  - Whether or not we take the branch, *always* execute the instruction immediately following the branch

3. Control Hazard: Branching

- **Option #3:** *Branch delay slot*
  - Whether or not we take the branch, *always* execute the instruction immediately following the branch
  - Worst-Case: Put a `nop` in the branch-delay slot

3. Control Hazard: Branching

- **Option #3:** *Branch delay slot*
  - Whether or not we take the branch, *always* execute the instruction immediately following the branch
  - Worst-Case: Put a nop in the branch-delay slot
  - Better Case: Move an instruction from before the branch into the branch-delay slot
    - Must not affect the logic of program

3. Control Hazard: Branching

- MIPS uses this *delayed branch* concept
  - Re-ordering instructions is a common way to speed up programs
  - Compiler finds an instruction to put in the branch delay slot ≈ 50% of the time

3. Control Hazard: Branching

- MIPS uses this *delayed branch* concept
  - Re-ordering instructions is a common way to speed up programs
  - Compiler finds an instruction to put in the branch delay slot ≈ 50% of the time
- Jumps also have a delay slot
  - Why is one needed?

Delayed Branch Example

Nondelayed Branch

```
or  $8, $9, $10

add $1, $2, $3

sub $4, $5, $6

beq $1, $4, Exit

xor $10, $1, $11
```

Delayed Branch Example

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 Example

Nondelayed Branch

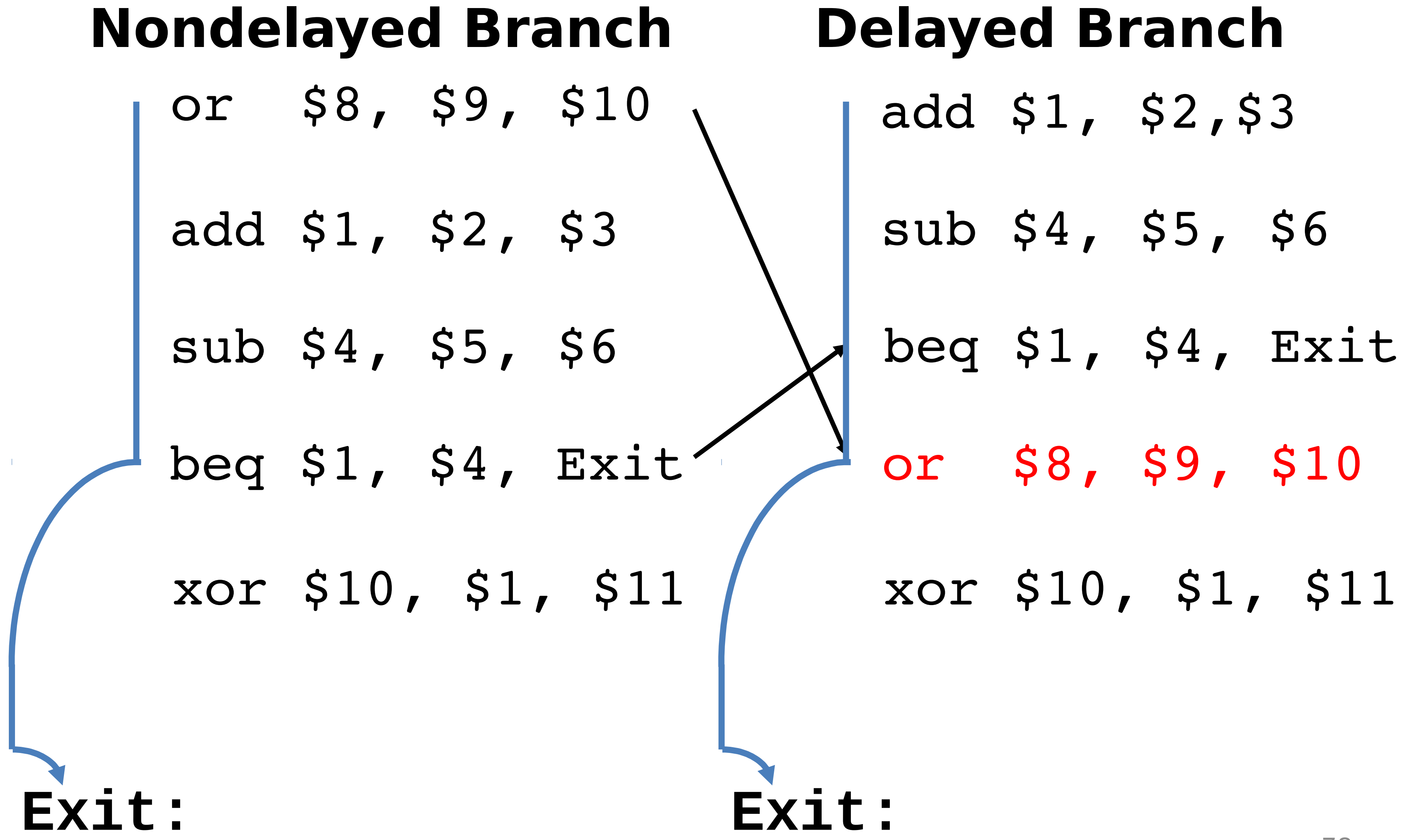
```
or   $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11
```

Exit:

Delayed Branch

```
add $1, $2,$3
sub $4, $5, $6
beq $1, $4, Exit
or   $8, $9, $10
xor $10, $1, $11
```

Delayed Branch Example



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:

Delayed Branch Example

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:
```

Why not any of the other instructions?

Delayed Jump in MIPS

- MIPS Green Sheet for `jal`:
  `R[31]=PC+8; PC=JumpAddr`

Delayed Jump in MIPS

- MIPS Green Sheet for `jal`:
  `R[31]=PC+8; PC=JumpAddr`
  - `PC+8` because of *jump delay slot*!
  - Instruction at `PC+4` always gets executed before `jal` jumps to label, so return to `PC+8`

Technology Break

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Administrivia
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

Dynamic Branch Prediction

- Branch penalty is more significant in deeper pipelines
  - Also superscalar pipelines (discussed tomorrow)

Dynamic Branch Prediction

- Branch penalty is more significant in deeper pipelines
  - Also superscalar pipelines (discussed tomorrow)
- Use *dynamic branch prediction*
  - Have branch prediction buffer (a.k.a. branch history table) that stores outcomes (taken/not taken) indexed by recent branch instruction addresses

Dynamic Branch Prediction

- Branch penalty is more significant in deeper pipelines
  - Also superscalar pipelines (discussed tomorrow)
- Use *dynamic branch prediction*
  - Have branch prediction buffer (a.k.a. branch history table) that stores outcomes (taken/not taken) indexed by recent branch instruction addresses
  - To execute a branch
    - Check table and predict the same outcome for next fetch
    - If wrong, flush pipeline and flip prediction

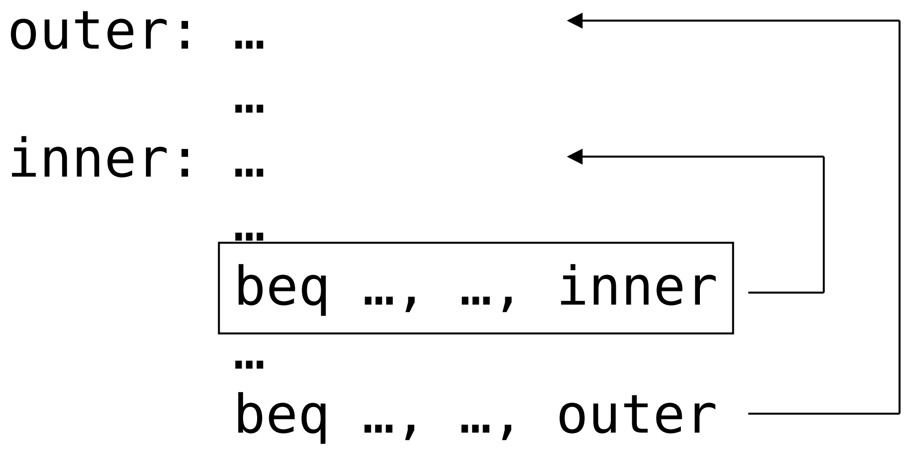
1-Bit Predictor: Shortcoming

- Examine the code below, assuming both loops will be executed multiple times:

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

1-Bit Predictor: Shortcoming

- Examine the code below, assuming both loops will be executed multiple times:

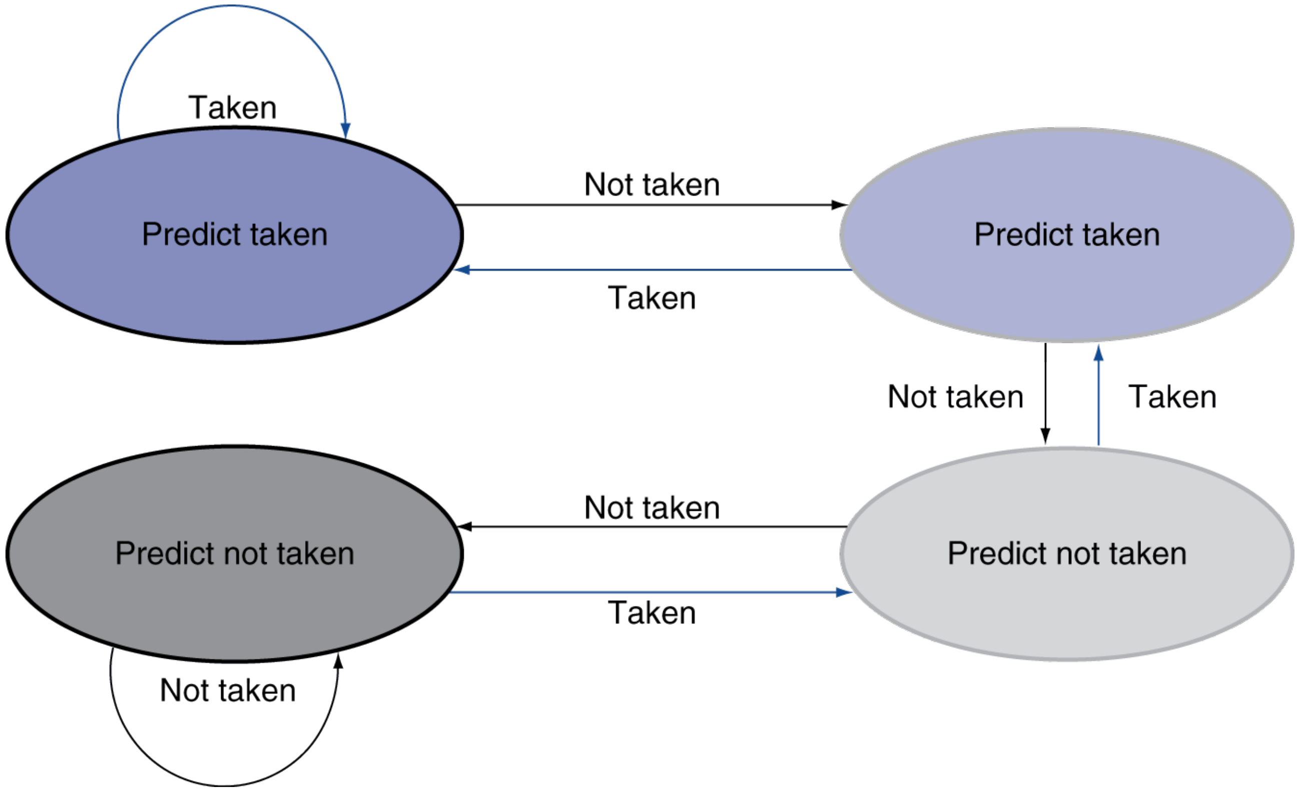


```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

- Inner loop branches are predicted wrong twice!
  - Predict as <u>taken</u> on last iteration of inner loop
  - Then predict as <u>not taken</u> on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction after two successive incorrect predictions



Question: For each code sequences below, choose one of the statements below:

1:
```
lw $t0,0($t0)
add $t1,$t0,$t0
```

2:
```
add $t1,$t0,$t0
addi $t2,$t0,5
addi $t4,$t1,5
```

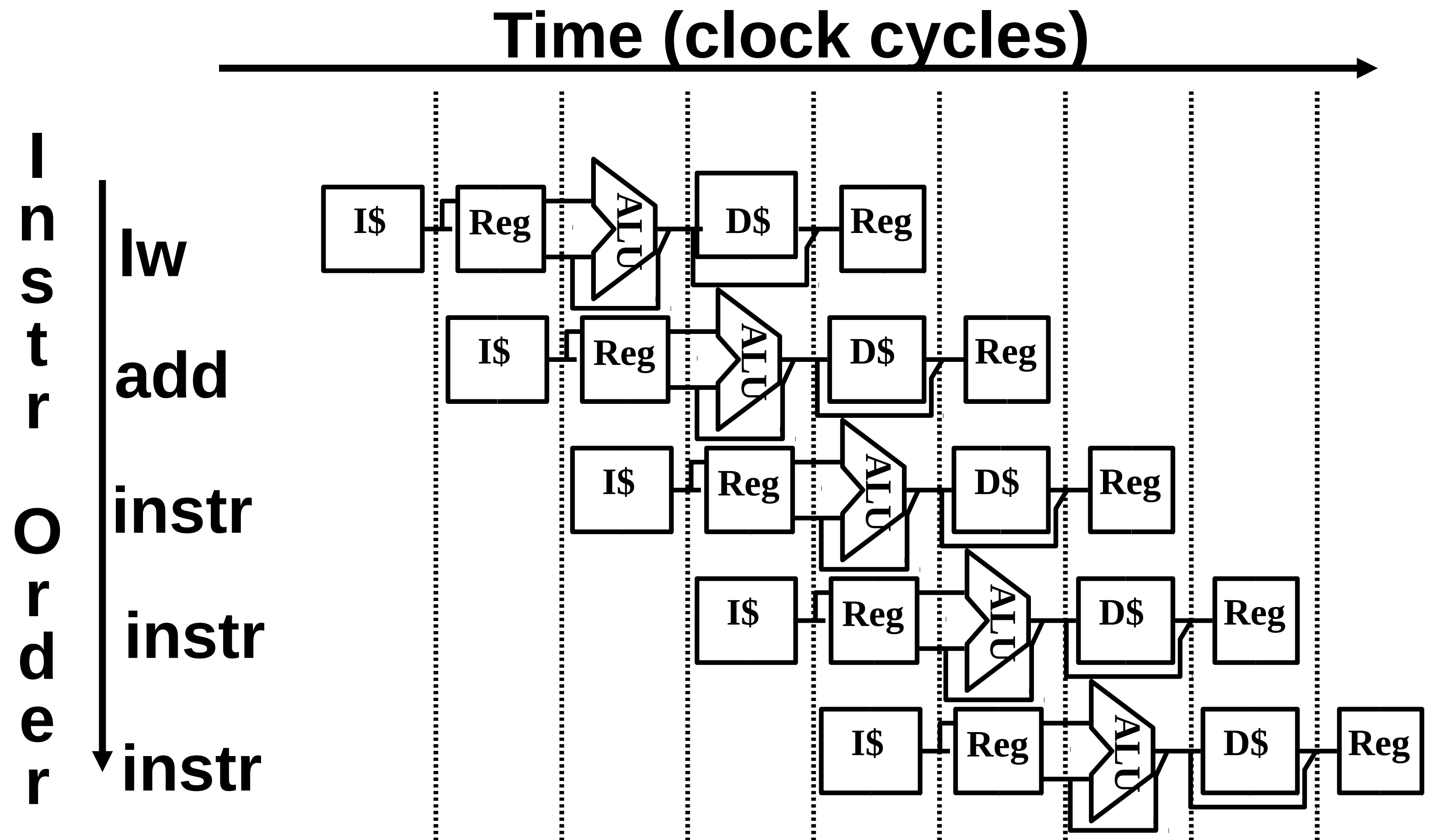
3:
```
addi $t1,$t0,1
addi $t2,$t0,2
addi $t3,$t0,2
addi $t3,$t0,4
addi $t5,$t1,5
```

(B) No stalls as is

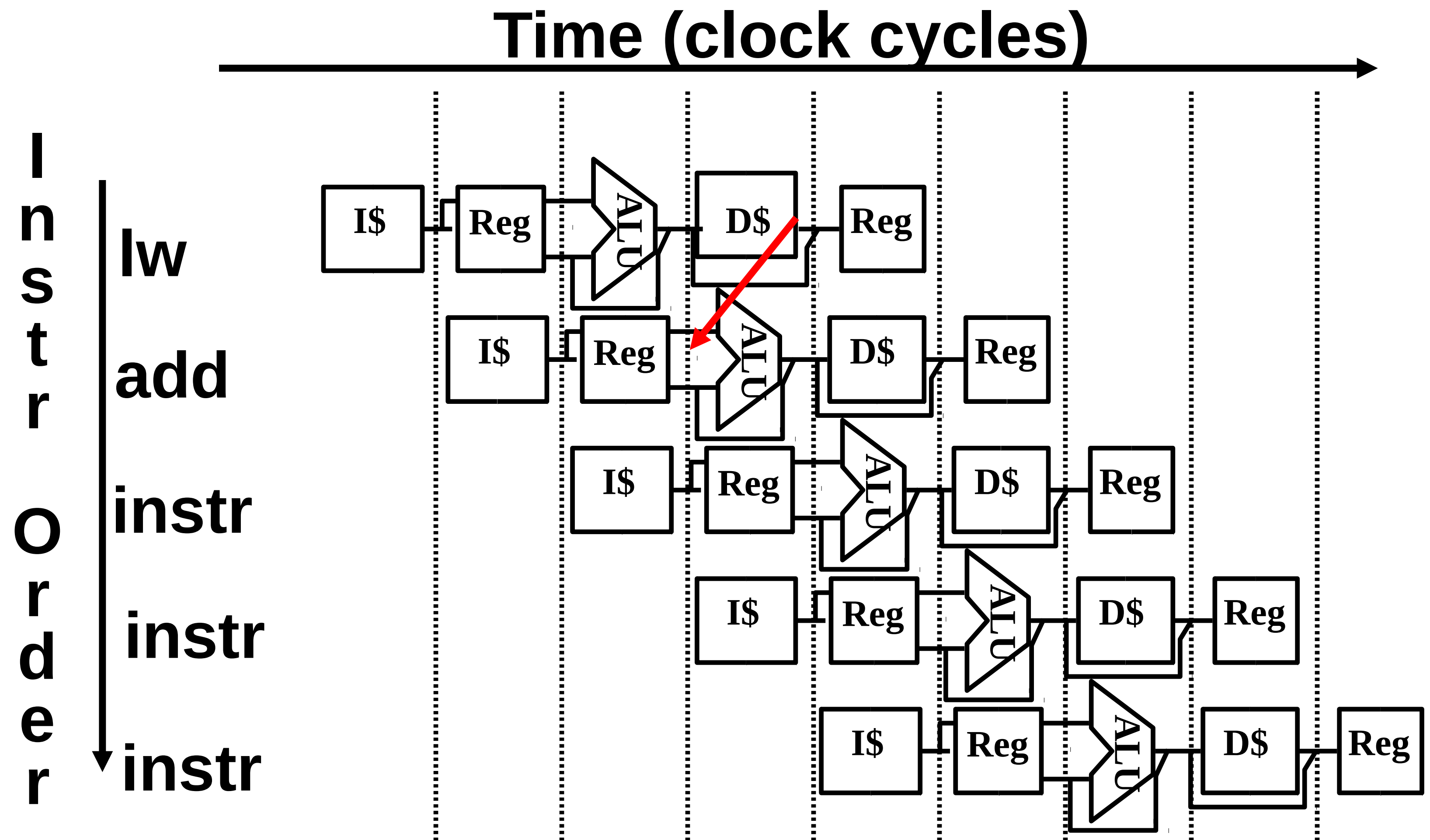
(G) No stalls with forwarding

(P) Must stall

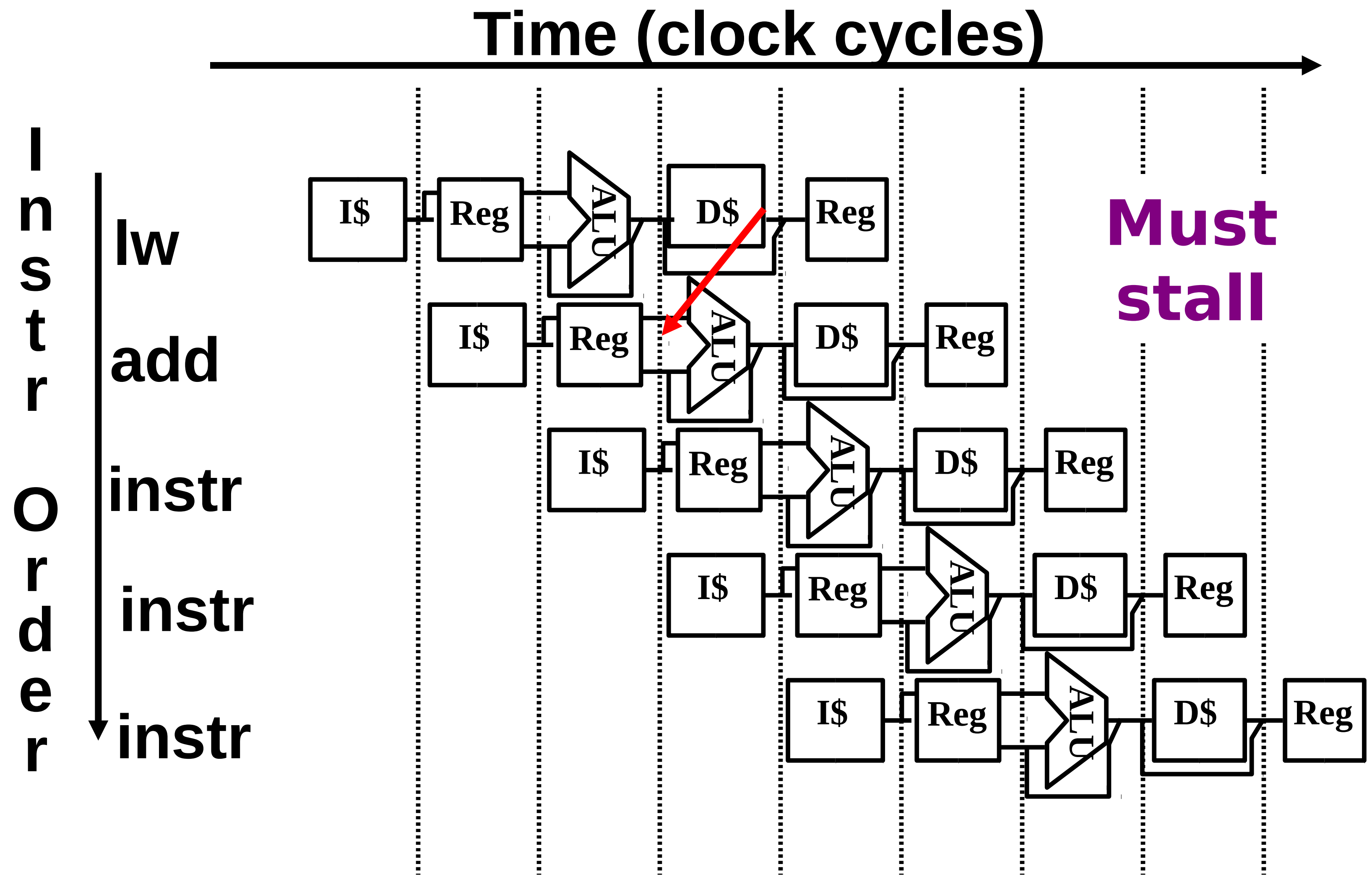
Code Sequence 1



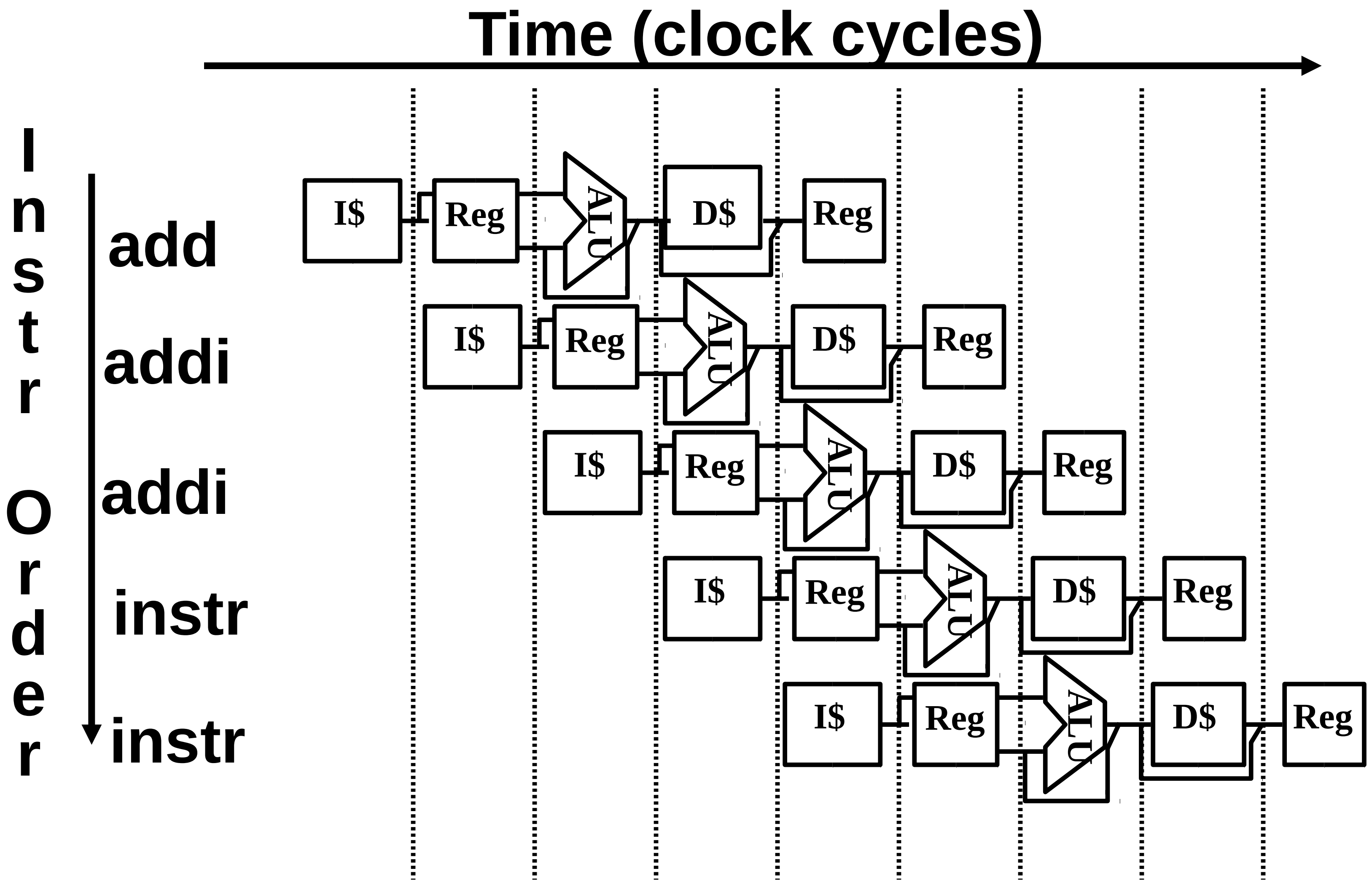
Code Sequence 1



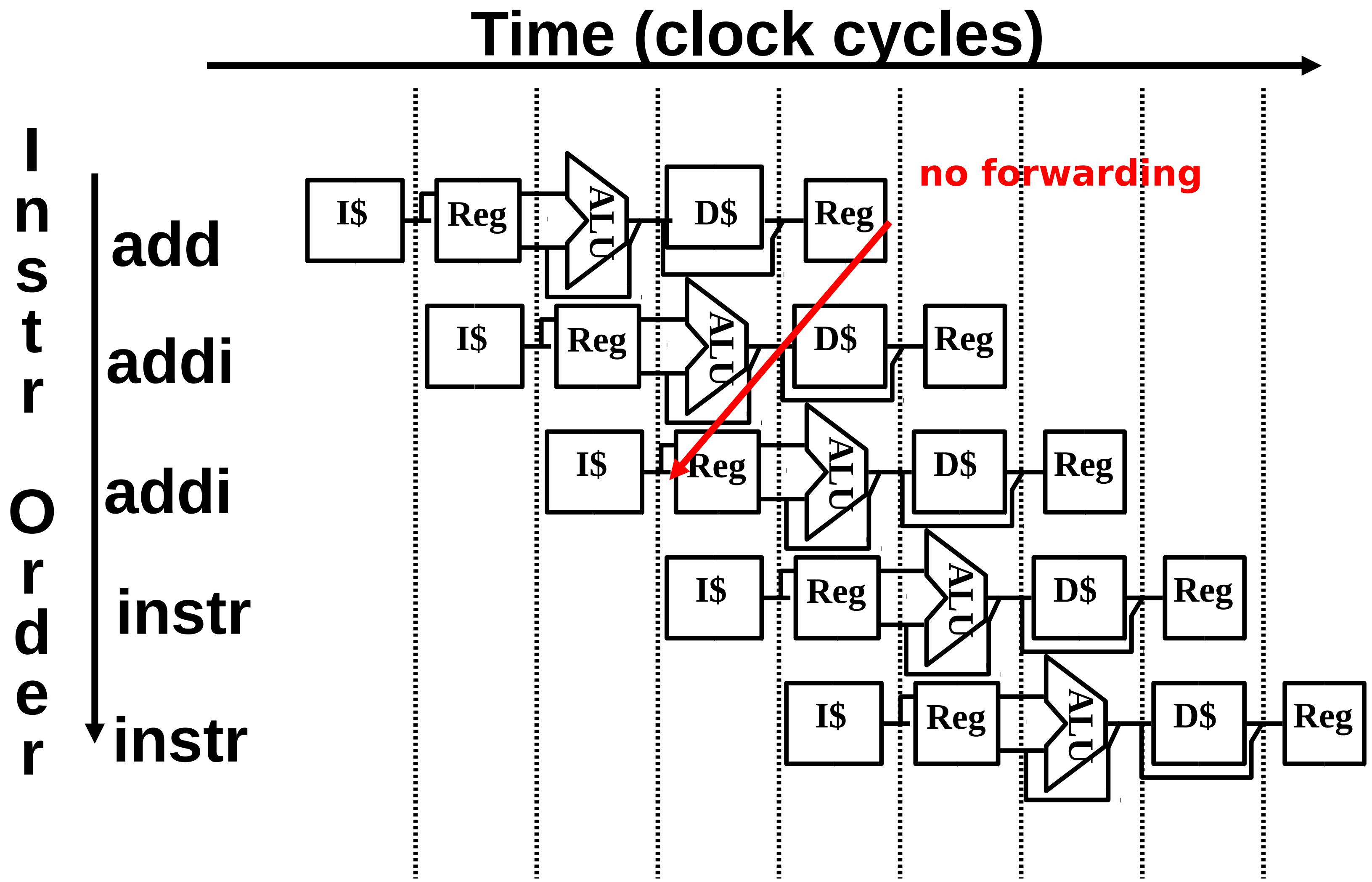
Code Sequence 1



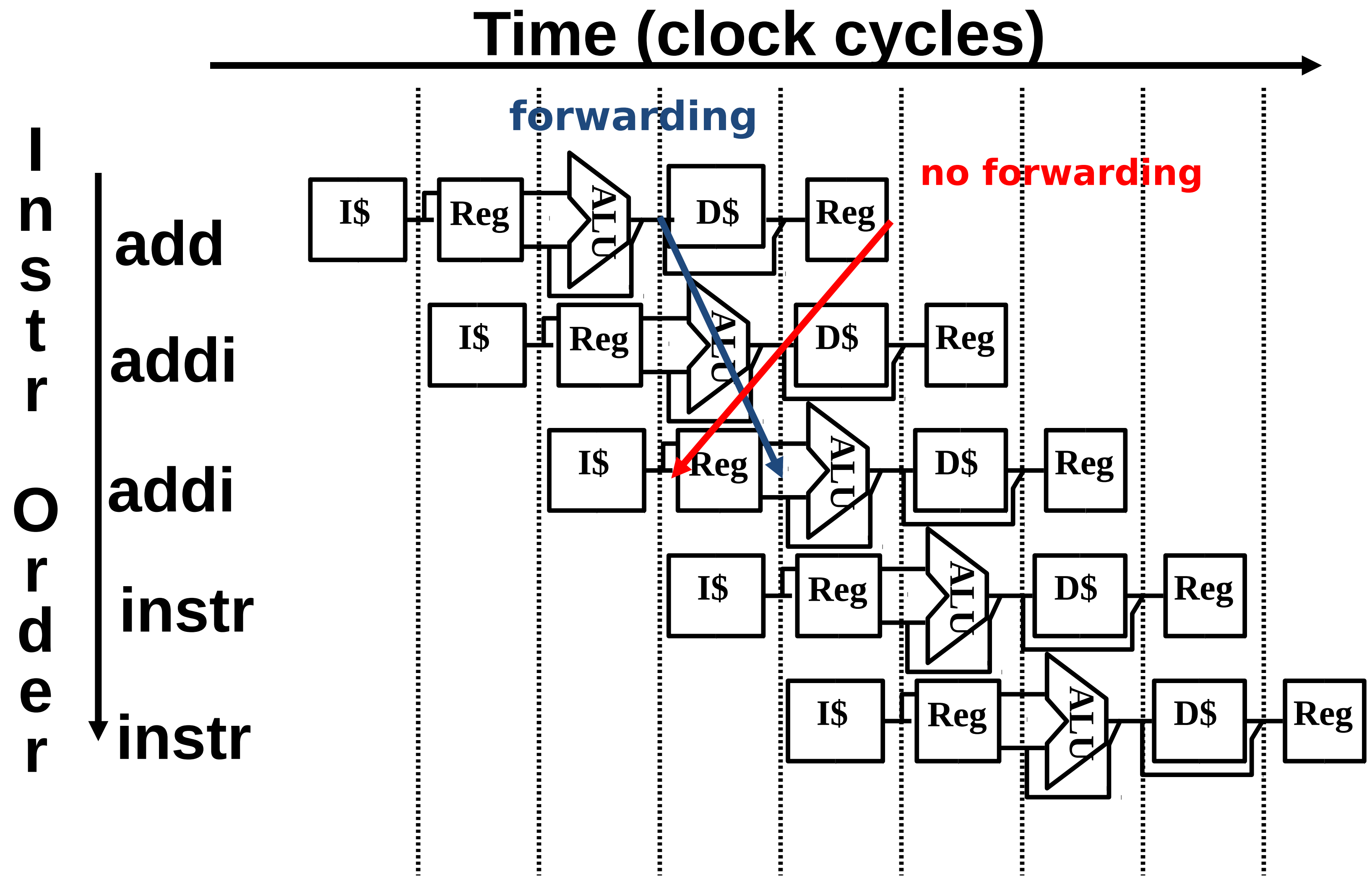
Code Sequence 2



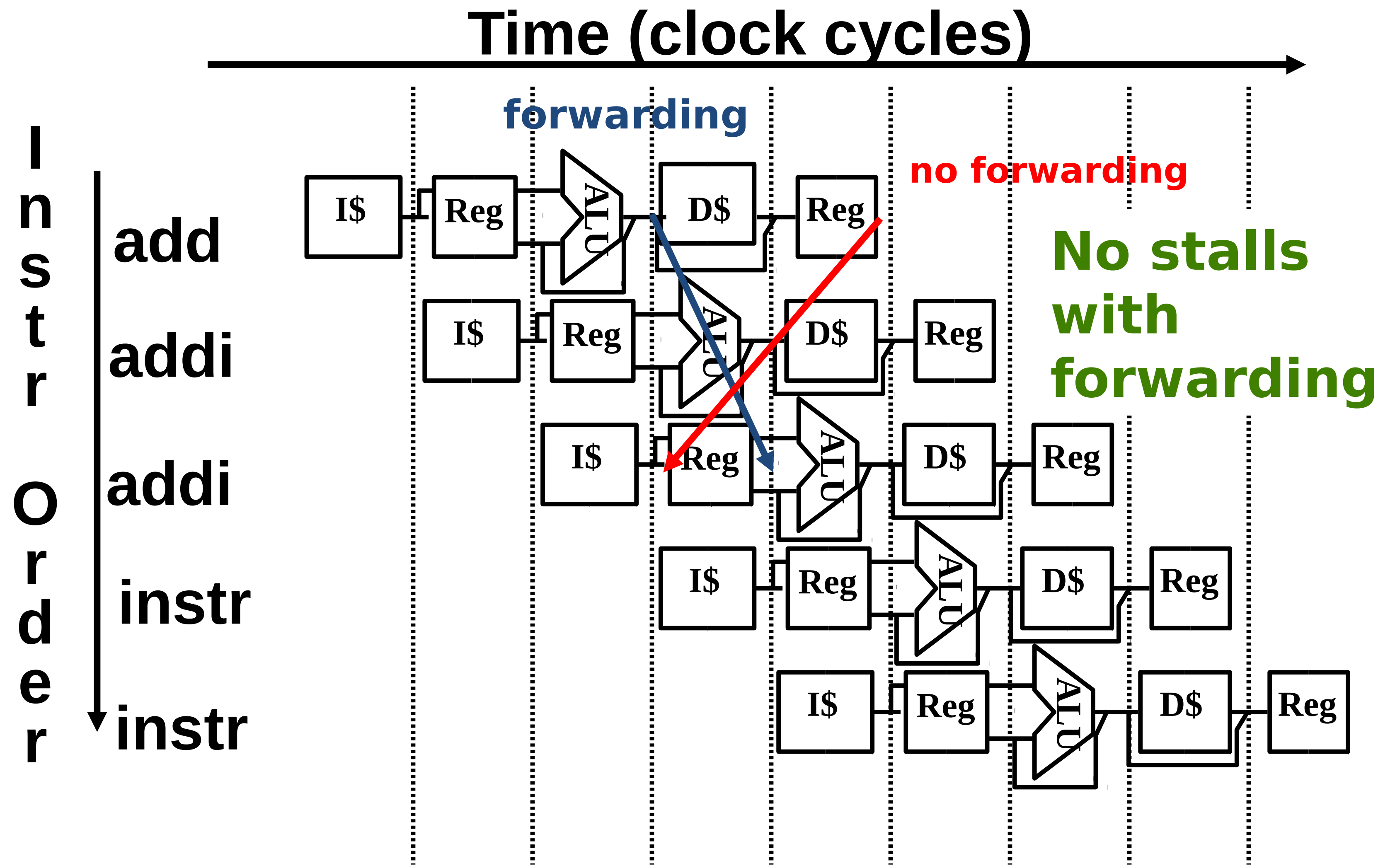
Code Sequence 2



Code Sequence 2

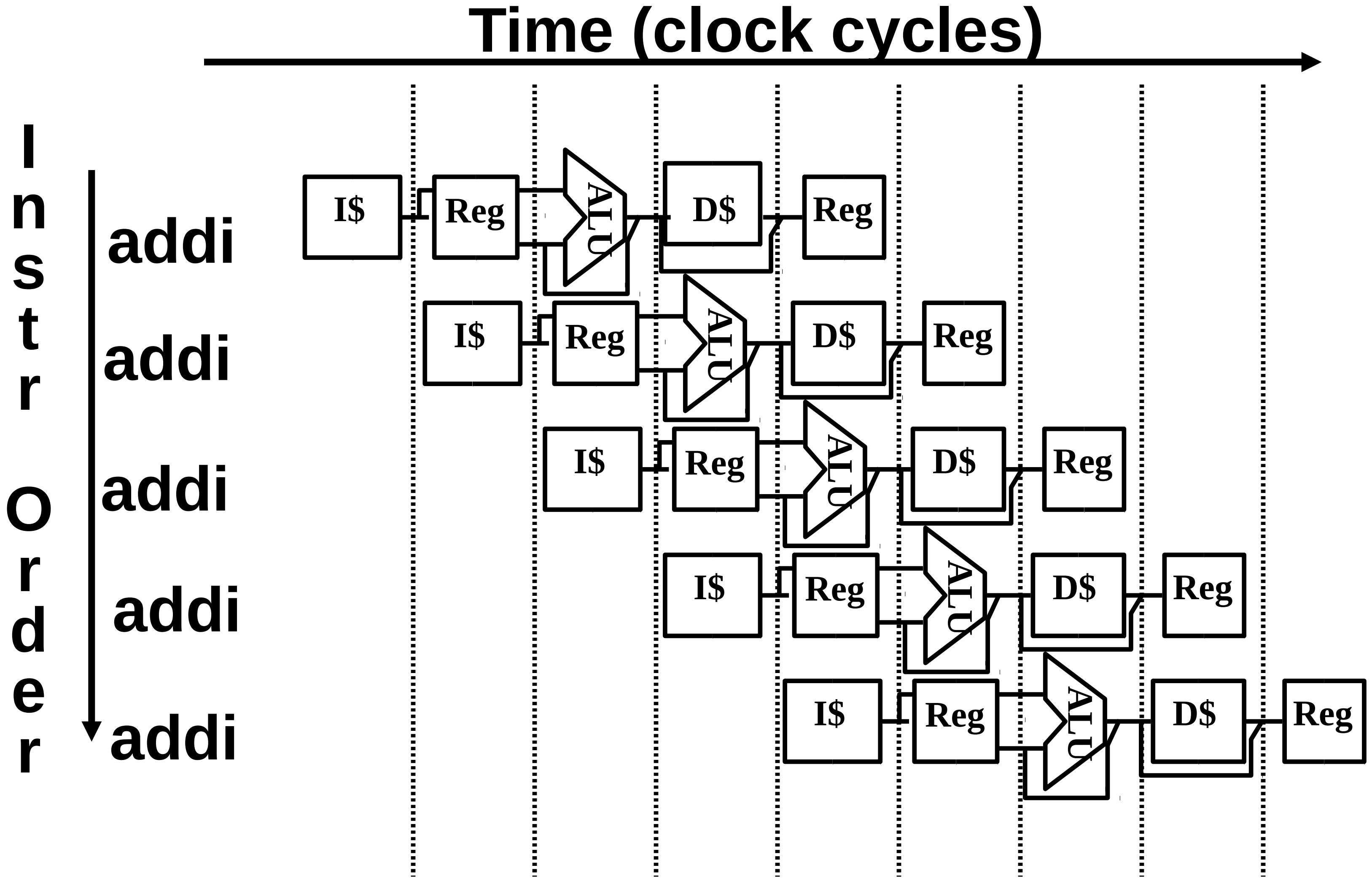


Code Sequence 2

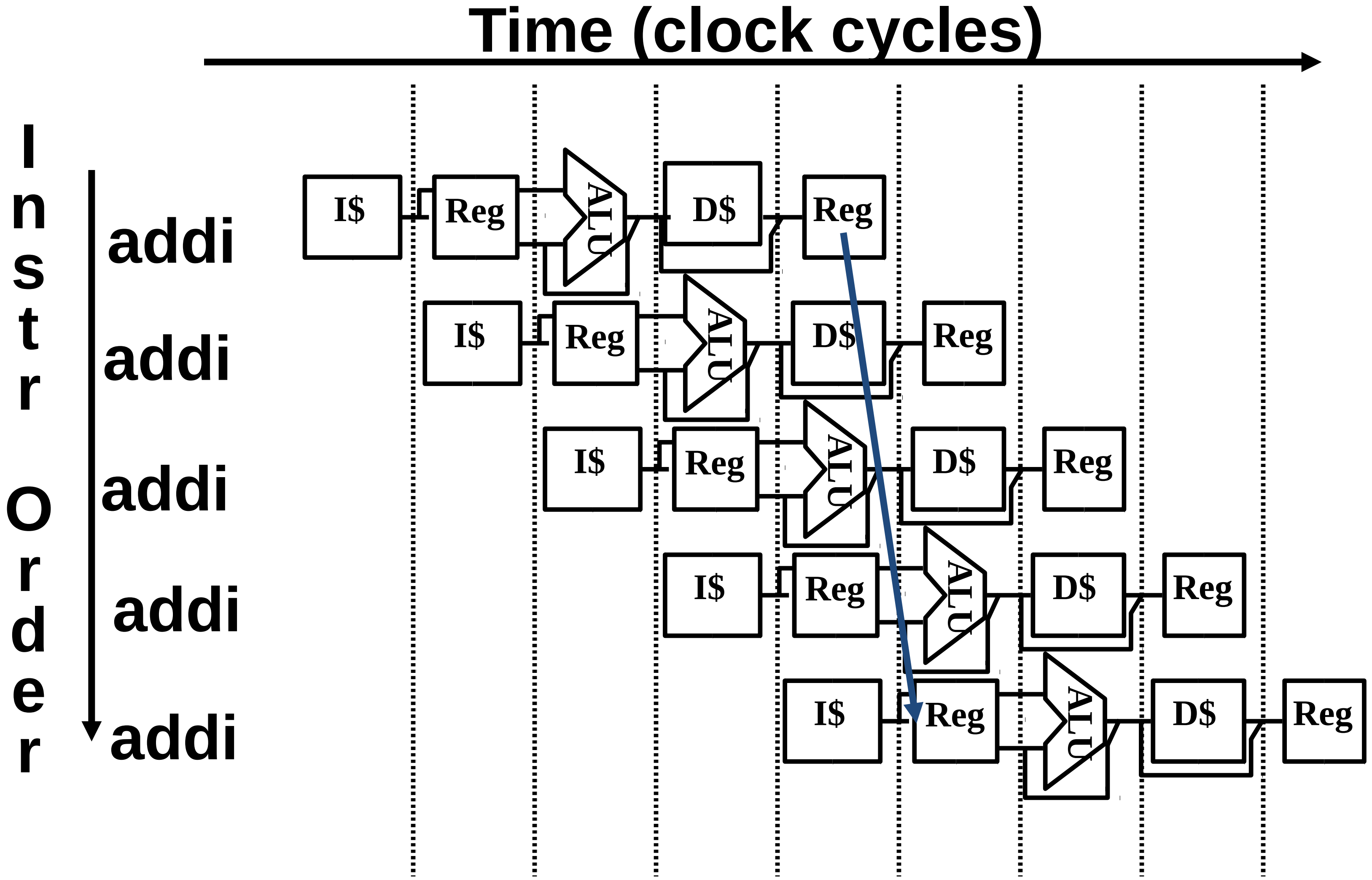


No stalls with forwarding

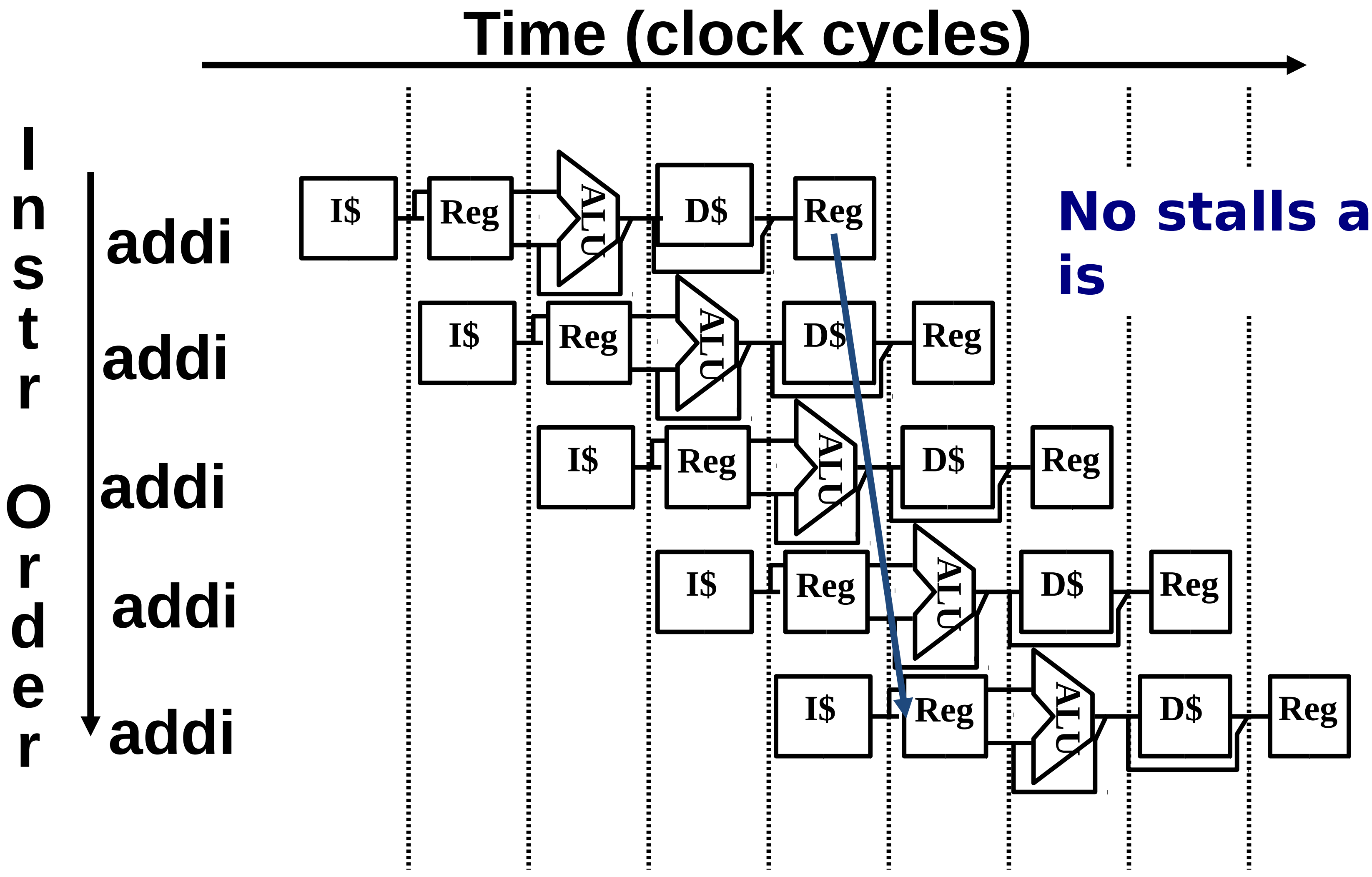
Code Sequence 3



Code Sequence 3



Code Sequence 3



No stalls as is

Summary

- Hazards reduce effectiveness of pipelining
  - Cause stalls/bubbles
- Structural Hazards
  - Conflict in use of datapath component
- Data Hazards
  - Need to wait for result of a previous instruction
- Control Hazards
  - Address of next instruction uncertain/unknown
  - Branch and jump delay slots