

## Speed up with pipeline

- **For a pipeline processor:**

- $k$ -stage pipeline processes with a clock cycle time  $t_p$  is used to execute  $n$  tasks.
- The time required for the first task  $T_1$  to complete the operation  $= k * t_p$  (if  $k$  segments in the pipe)
- The time required to complete  $(n-1)$  tasks  $= (n-1) * t_p$
- Therefore to complete  $n$  tasks using a  $k$ -segment pipeline requires  $= k + (n-1)$  clock cycles.

- **For the non-pipelined processor :**

- Time to complete each task  $= t_n$
- Total time required to complete  $n$  tasks  $= n * t_n$
- Speed up  $=$  non pipelining processing / pipelining processing

$$S = \frac{T_1}{T_K} = \frac{nt_n}{(k + (n - 1))t_p}$$

---

## Pipeline Hazards

Any anomalies that prevent a pipeline datapath from functioning without pipeline stalls or flushes

## Three Major Pipeline Hazards

1. Structural Hazards

2. Data Hazards

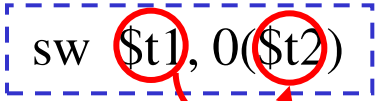
3. Control Hazards

## 1. Structural Hazards:

= pipeline hazards due to hardware resource conflicts

**Example** Instruction fetch and data access

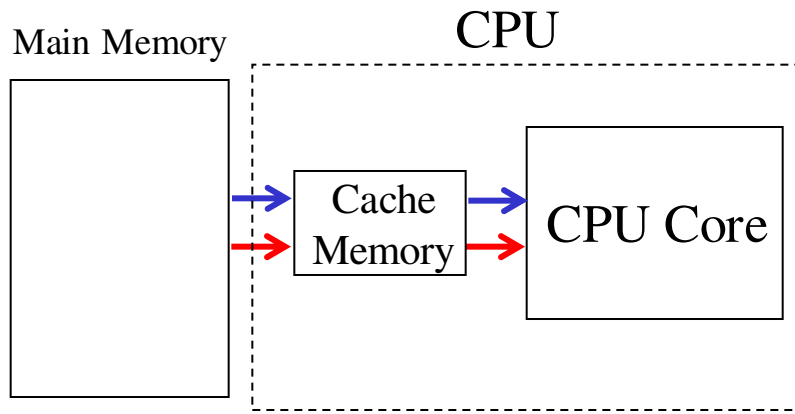
A processor instruction = [operator] + [parameters]

processor instruction:  ① instruction fetch  
processor instruction: sw \$t1, 0(\$t2) // \$t1 → Mem[\$t2+0]  
② save register to memory

Execution of this instruction requires two memory accesses

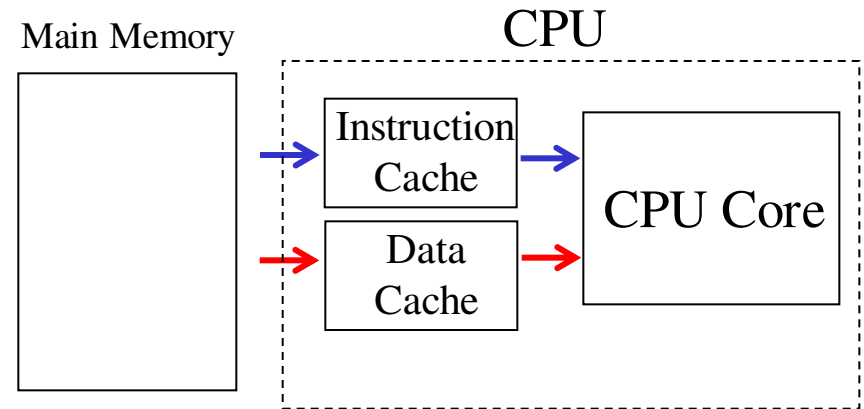
## 1. Structural Hazards (continued)

### Unified Cache Architecture



Instruction code and data  
need to take the same path  
(i486)

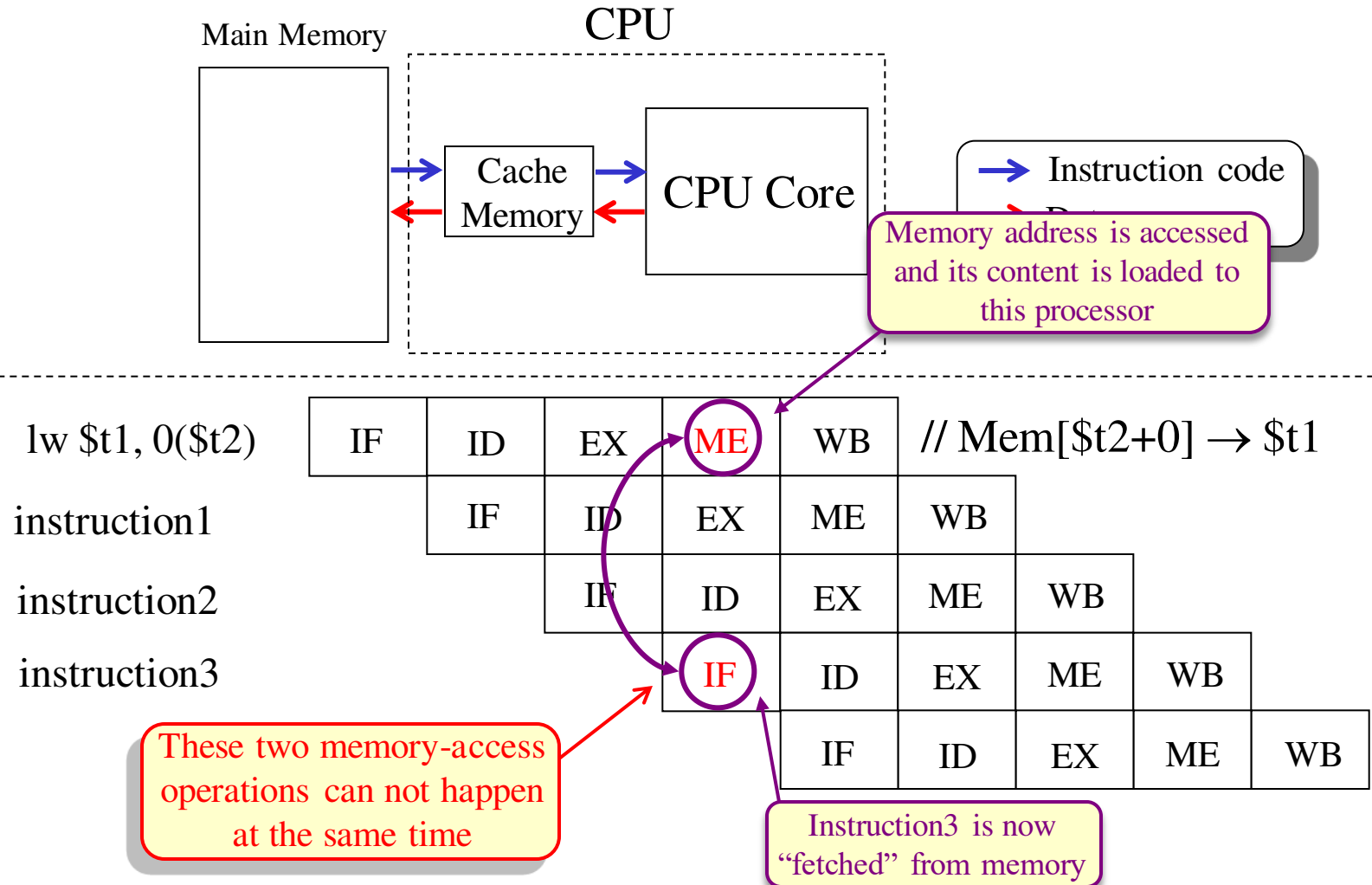
### Split Cache Architecture



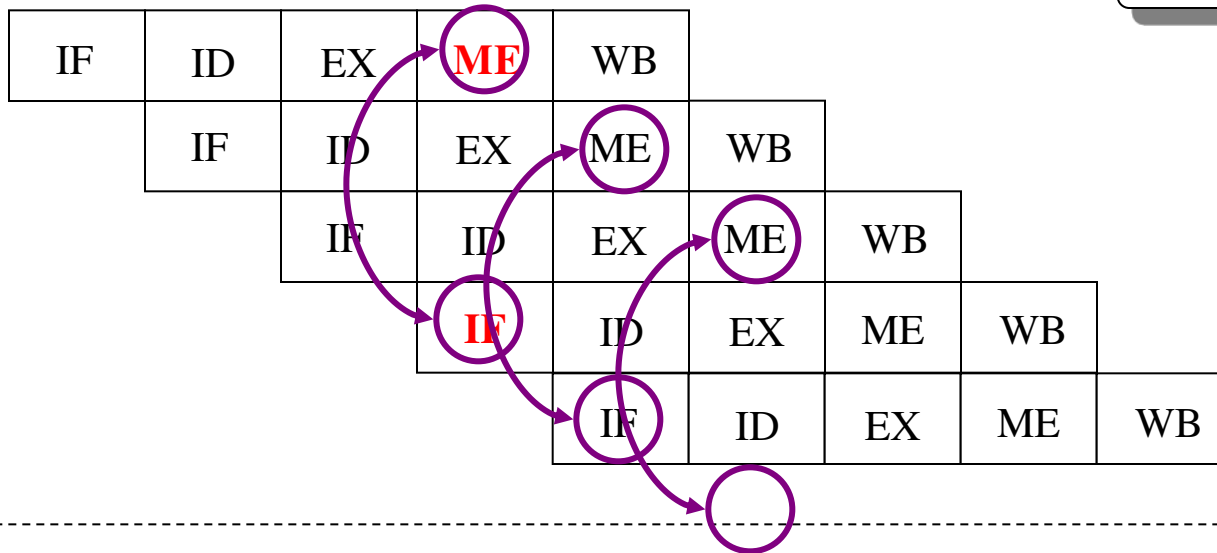
Instruction code and data  
take its own path  
(Pentium and later)

→ Instruction code  
→ Data

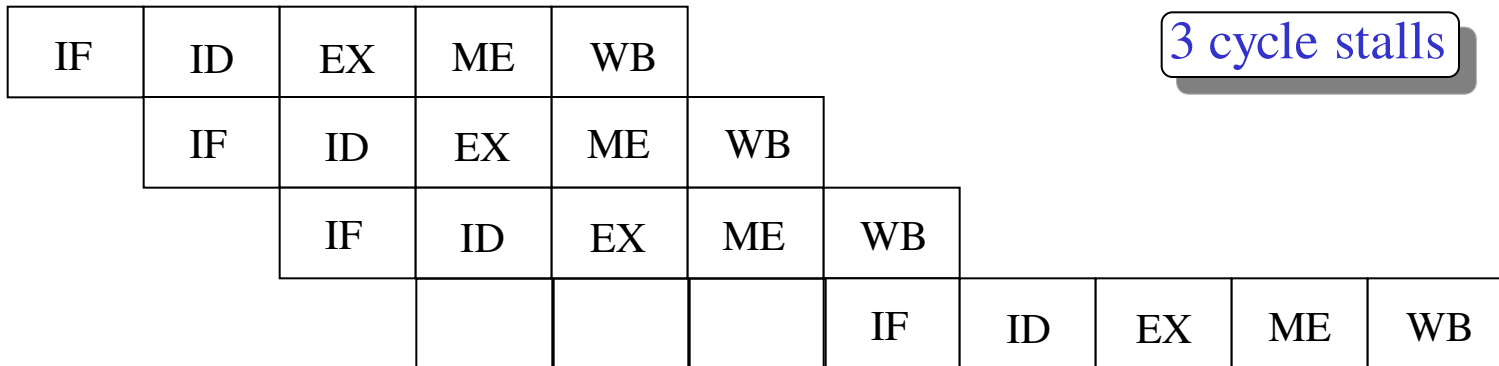
## Unified Cache Architecture



Resource Conflict

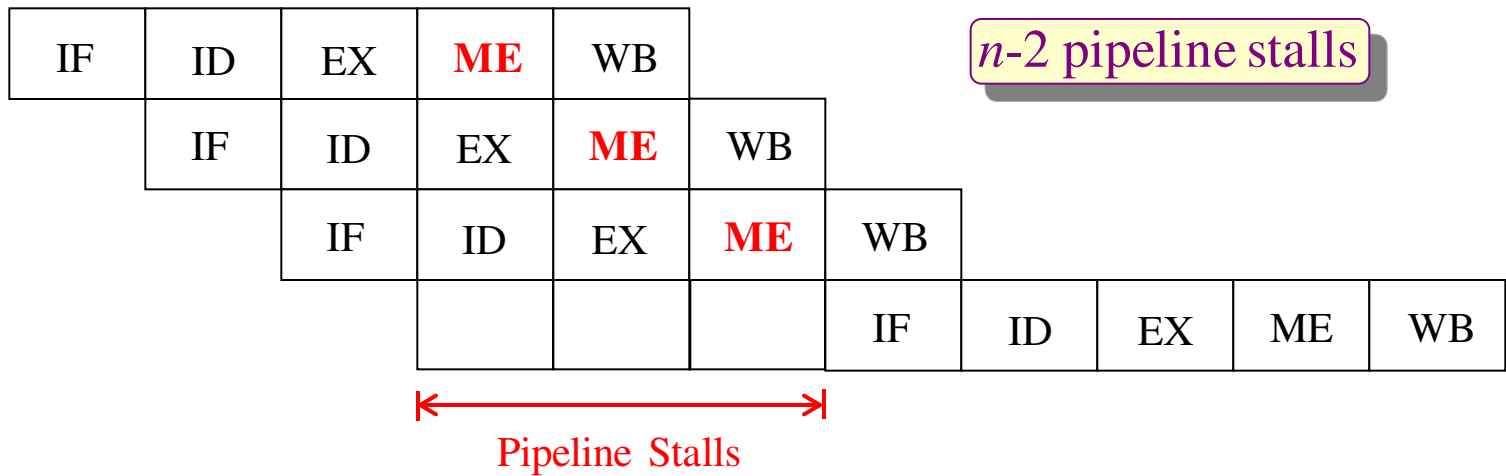


3 cycle stalls



Benefits from pipeline execution is lost

## Worst case structural pipeline hazards





## 2. Data Hazards:

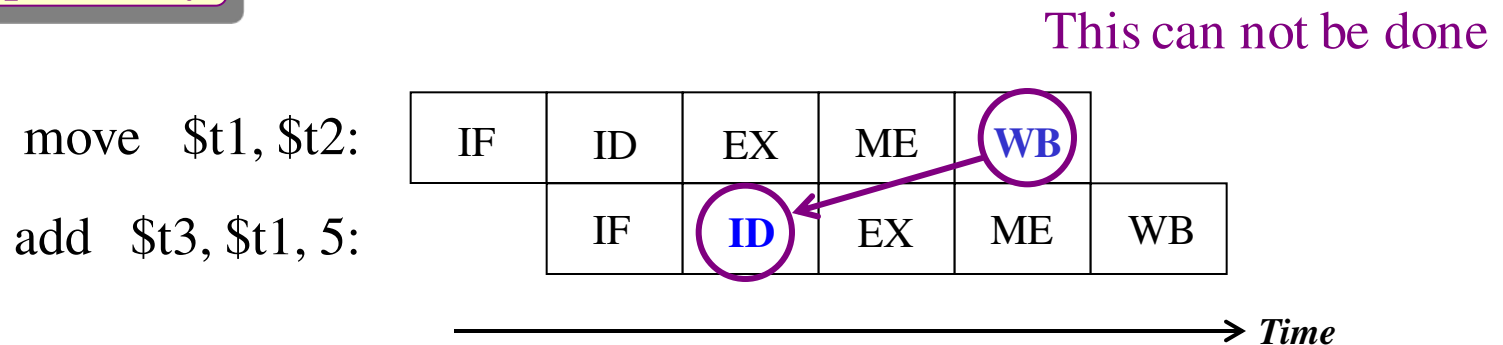
= pipeline hazards due data-dependency between instructions

### Example

•  
•  
move \$t1, \$t2 // t1 ← t2  
add \$t3, \$t1, 5 // t3 ← t1 + 5  
•  
•

The same registers are used in more than one instruction

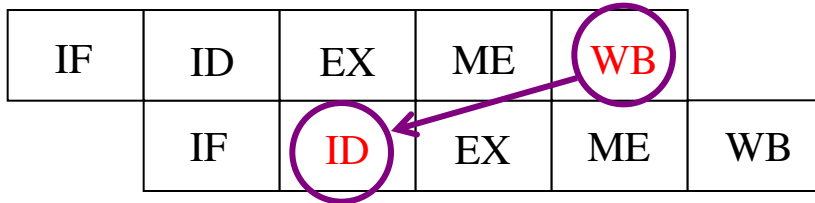
### Data Dependency



## Data Dependency

move \$t1, \$t2:

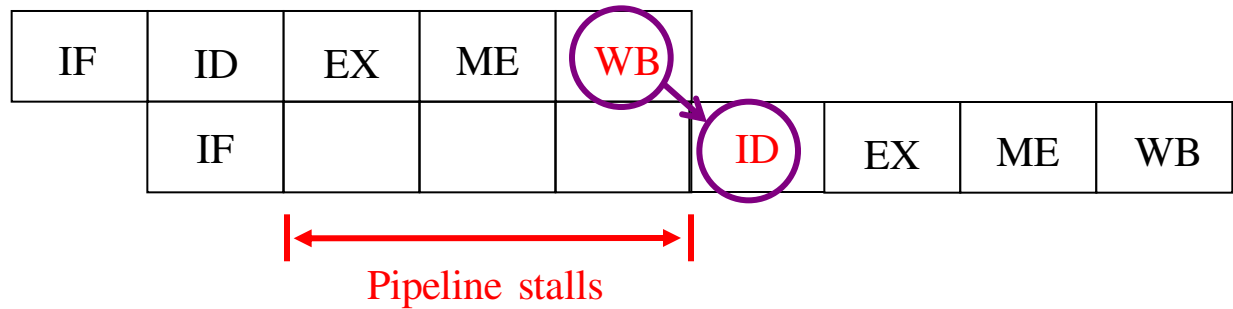
add \$t3, \$t1, 5:



## Resolve data dependency by stalls

move \$t1, \$t2:

add \$t3, \$t1, 5:



# Data Hazards

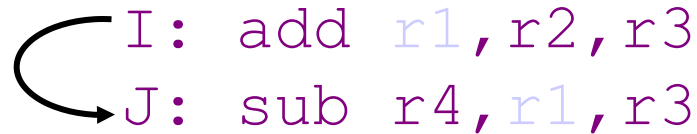
Execution Order is:

Instr<sub>I</sub>

Instr<sub>J</sub>

## Read After Write (RAW)

Instr<sub>J</sub> tries to read operand before Instr<sub>I</sub> writes it



```
I: add r1, r2, r3
J: sub r4, r1, r3
```

- Known as true dependency.

# Data Hazards

## Write After Read (WAR)

- i1.  $R4 \leftarrow R1 + \mathbf{R5}$   
i2.  $\mathbf{R5} \leftarrow R1 + R2$
- In any situation with a chance that i2 may finish before i1 (i.e., with concurrent execution), it must be ensured that the result of register R5 is not stored before i1 has had a chance to fetch the operands.
  - Called an “anti-dependence”.

# Data Hazards

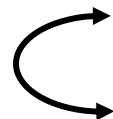
Execution Order is:

Instr<sub>I</sub>

Instr<sub>J</sub>

## Write After Write (WAW)

Instr<sub>J</sub> tries to write operand *before* Instr<sub>I</sub> writes it  
– Leaves wrong result ( Instr<sub>I</sub> not Instr<sub>J</sub> )



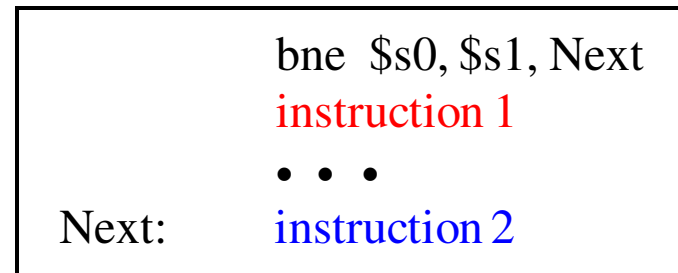
```
I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- Called an “**output dependence**”.

### 3. Control Hazards:

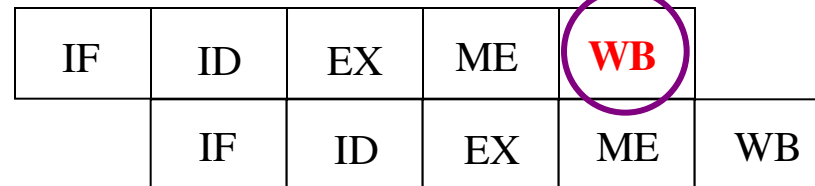
= pipeline hazards due to branch and jump instructions

Example: jump instructions



Pipeline flashes for control hazards

bne \$s0, \$s1, NEXT  
instruction 1



PC is set at the end of  
WB for “bne” instruction

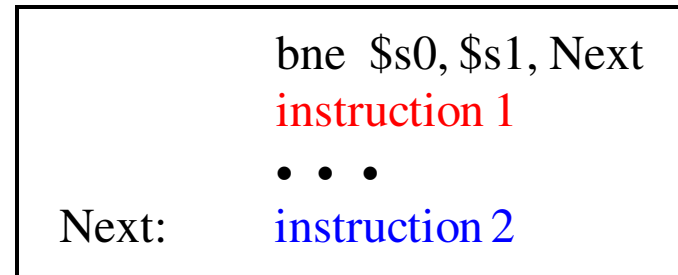
We can not start this  
instruction here

*Flushing the pipeline* occurs when a branch instruction jumps to a new memory location, invalidating all prior stages in the pipeline. These prior stages are cleared, allowing the pipeline to continue at the new instruction indicated by the branch

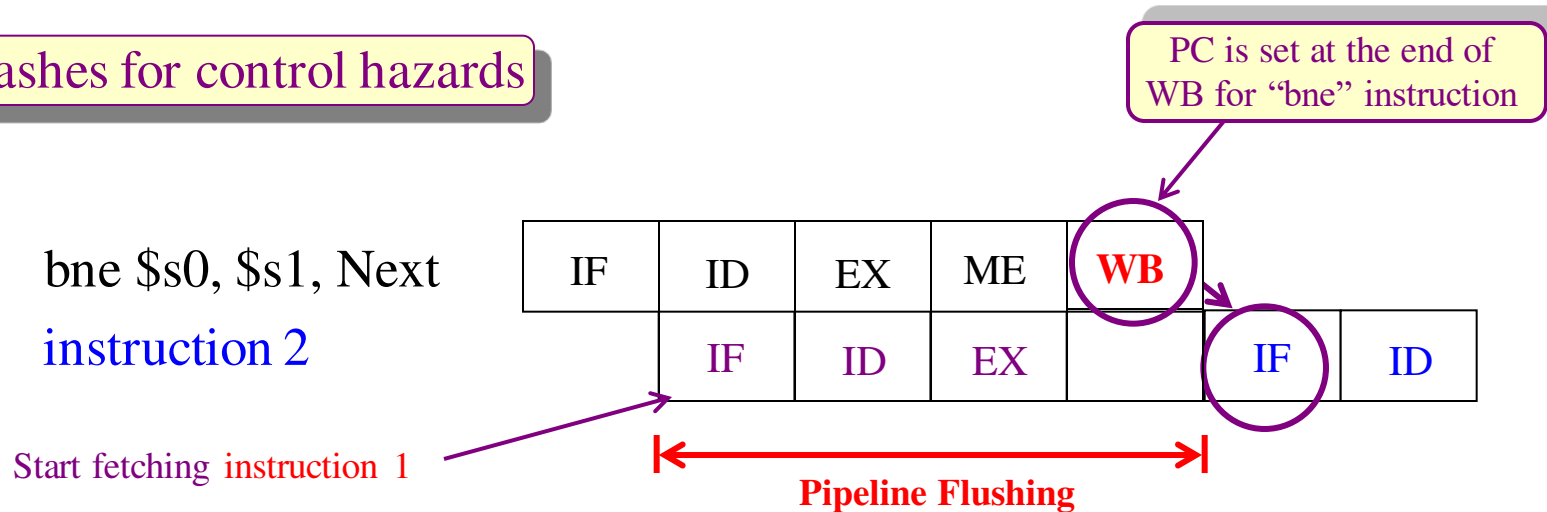
### 3. Control Hazards:

= pipeline hazards due to branch and jump instructions

Example: jump instructions



Pipeline flashes for control hazards





# Reducing pipeline hazards - three techniques

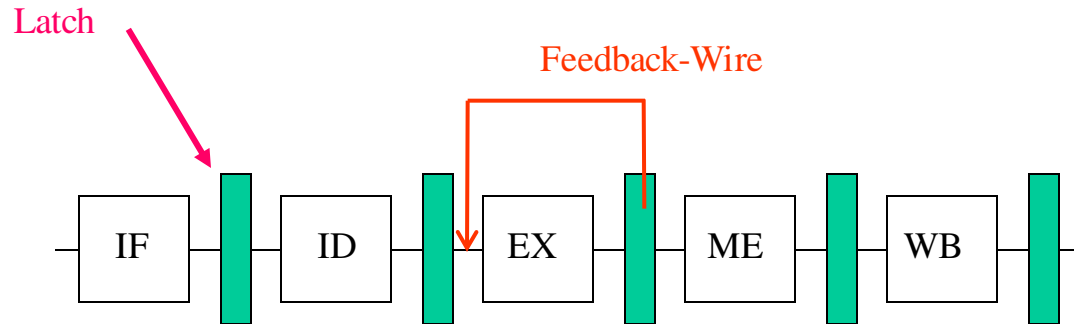
## Three techniques for different types of pipeline hazards

1. **Forwarding** – for reducing **RAW** data dependencies
2. **Instruction Scheduling** – for reducing **data hazards**
3. **Delayed Branch** – for reducing **control hazards**

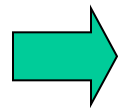
# Reducing pipeline hazards - three techniques

## Technique 1: Forwarding

= Internal pipeline circuit to feedback outputs of a stage



Outputs from a pipeline stage can be fed to the same or different stages of another instruction



Need hardware support

# Reducing pipeline hazards - three techniques

## Example

ADD R1, R2, R3

//  $R1 = R2 + R3$

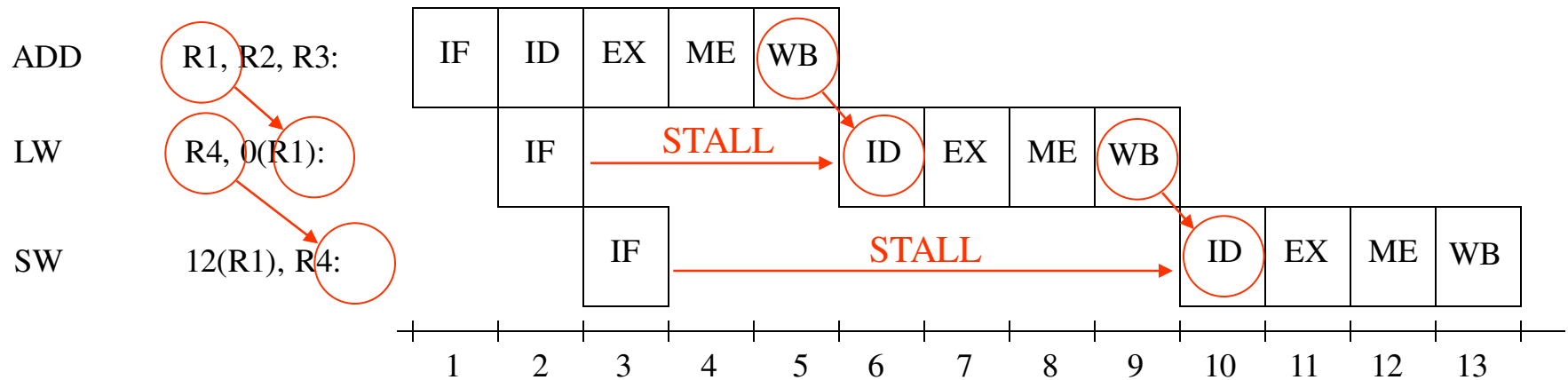
LW R4, 0(R1)

//  $R4 \leftarrow \text{MEM}[R1 + 0]$

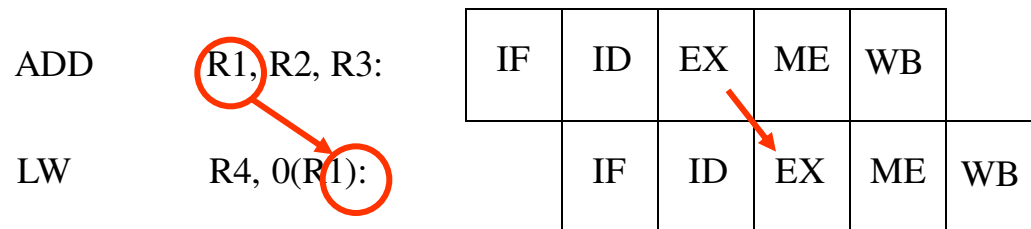
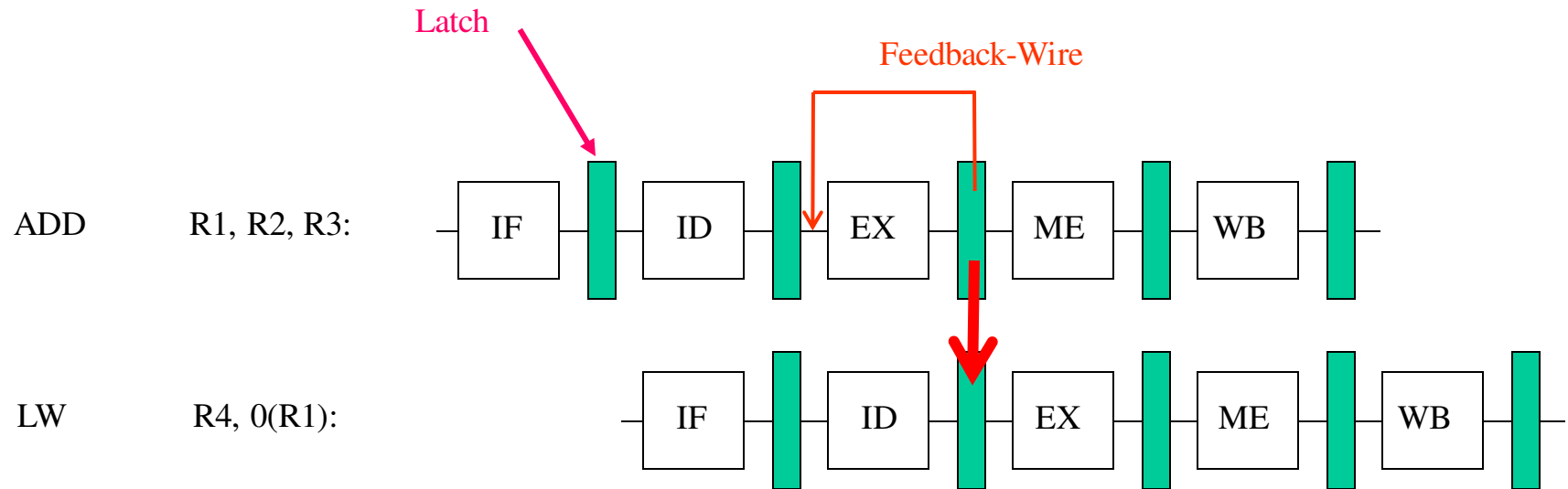
SW 12(R1), R4

//  $\text{MEM}[R1 + 12] \leftarrow R3$

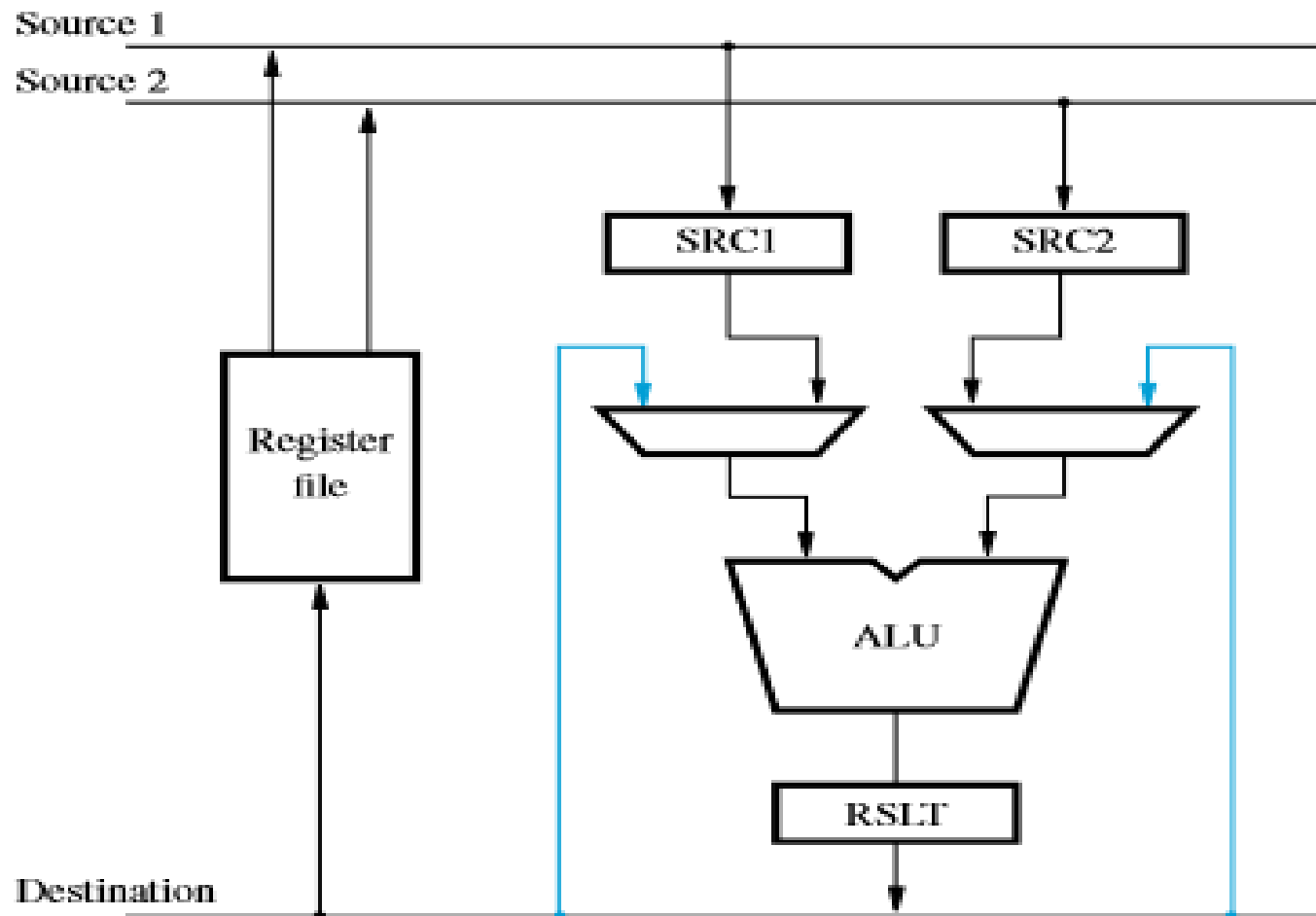
## Pipeline time chart for an ordinary pipeline processor



# Reducing pipeline hazards - three techniques

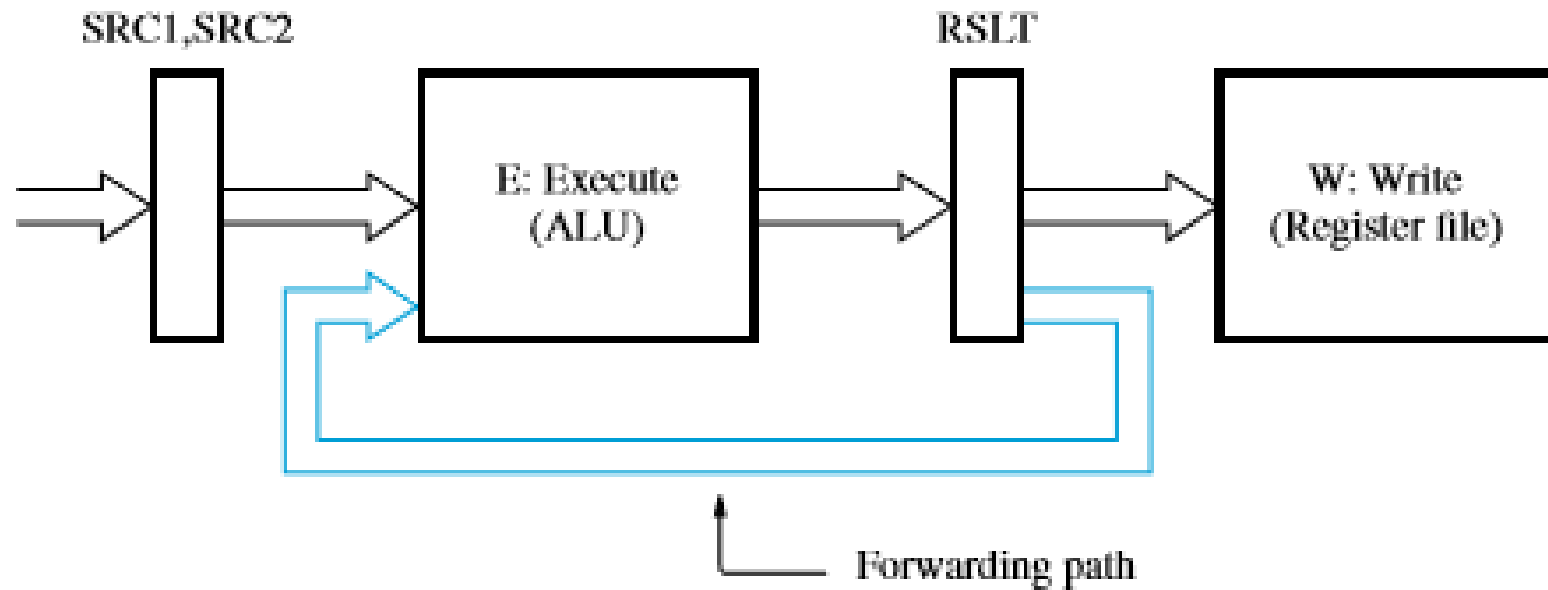


# Data Path

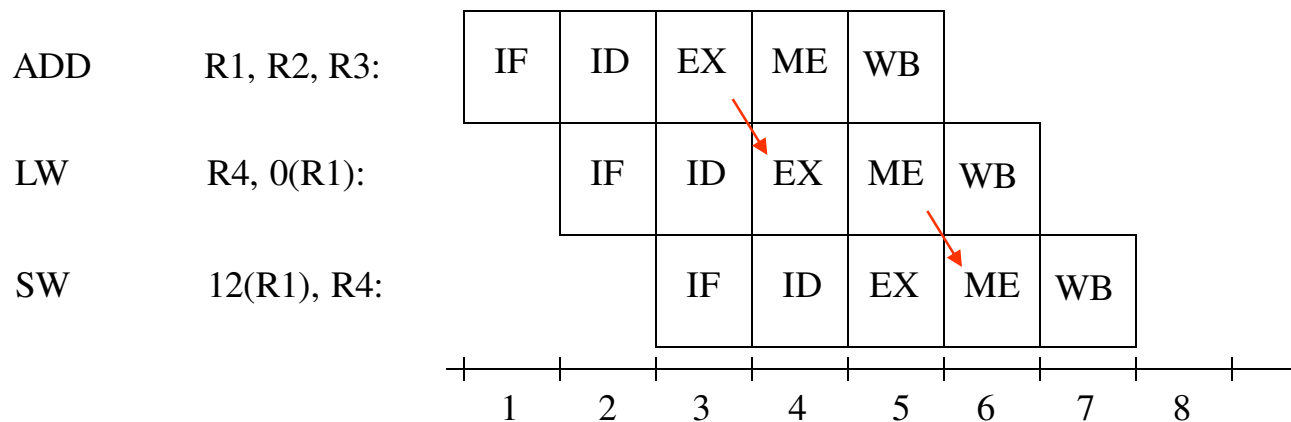
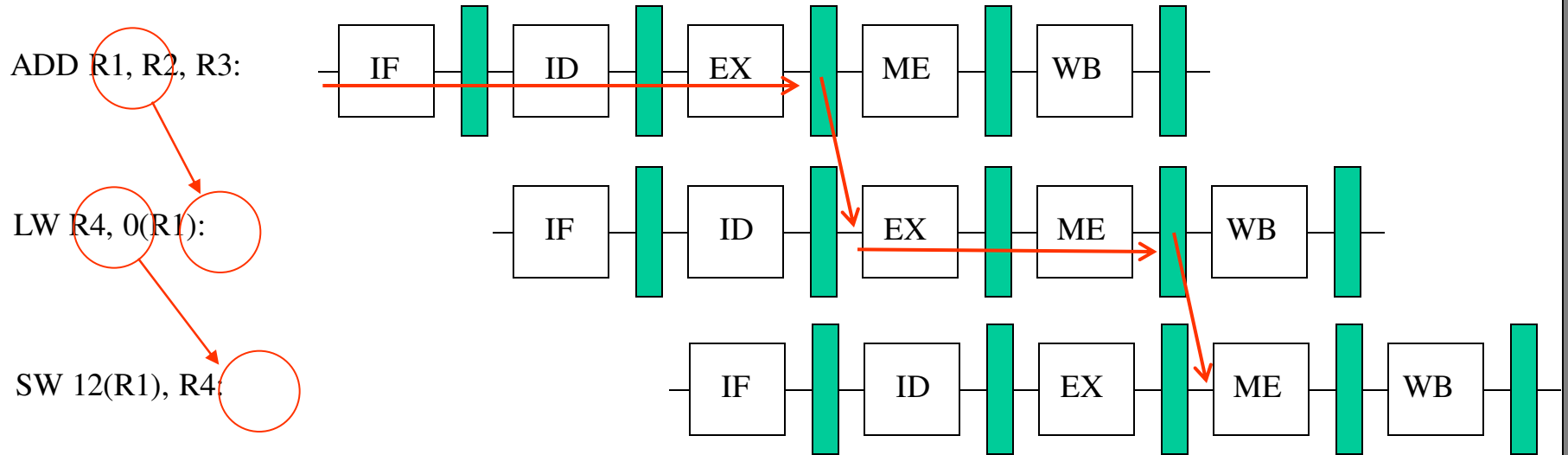


(a) Datapath

## Data forwarding



# Reducing pipeline hazards - three techniques



# Reducing pipeline hazards - three techniques

**Technique 2:** Instruction scheduling by a compiler

$a = b + c$

(in high-level language, such as C++)

$a = b + c$

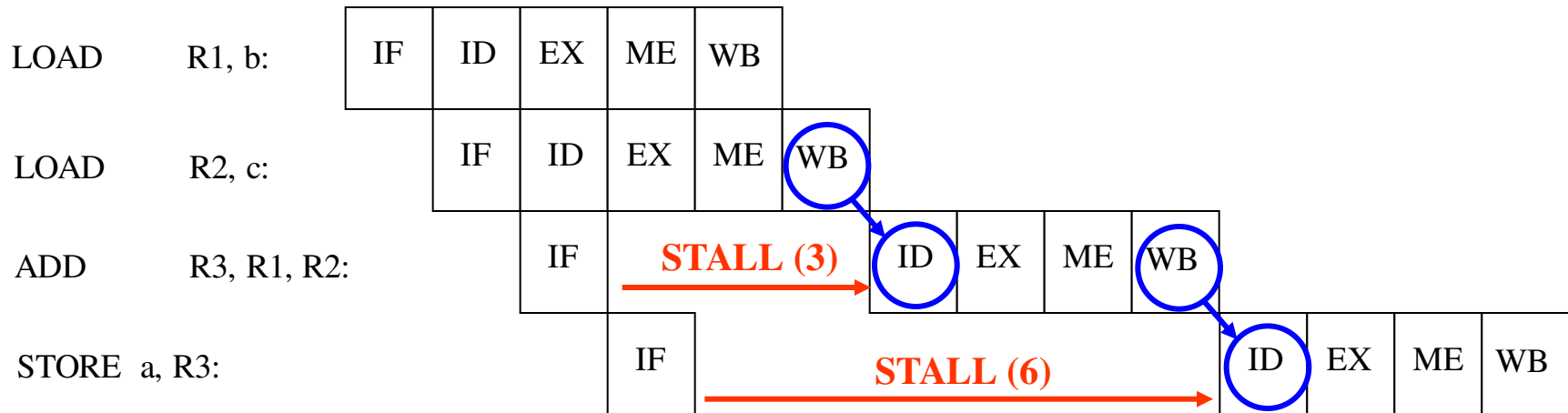


```
LOAD R1, b      // R1 ← MEM [Address of b]
LOAD R2, c      // R2 ← MEM [Address of b]
ADD R3, R1, R2   // R3 ← R1 + R2
STORE a, R3      // MEM [Address of a] ← R3
```

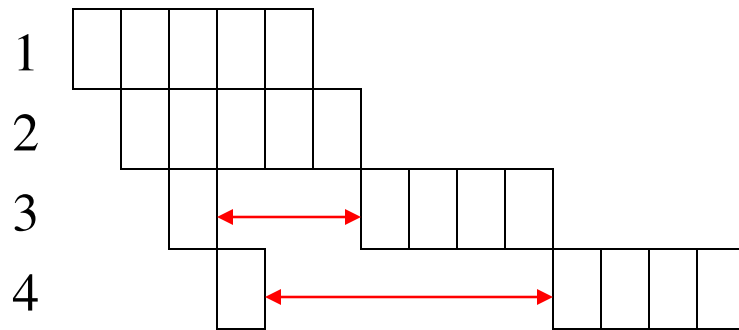


# Reducing pipeline hazards - three techniques

LOAD R1, b      // R1  $\leftarrow$  MEM [Address of b]  
LOAD R2, c      // R2  $\leftarrow$  MEM [Address of c]  
ADD R3, R1, R2   // R3  $\leftarrow$  R1 + R2  
STORE a, R3      // MEM [Address of a]  $\leftarrow$  R3



# Reducing pipeline hazards - three techniques



1	LOAD R1, b
2	LOAD R2, c
3	X
4	X
5	X
6	ADD R3, R1, R2
7	X
8	X
9	X
10	STORE a, R3

## Reducing pipeline hazards - three techniques

Now, we are going to execute two instructions

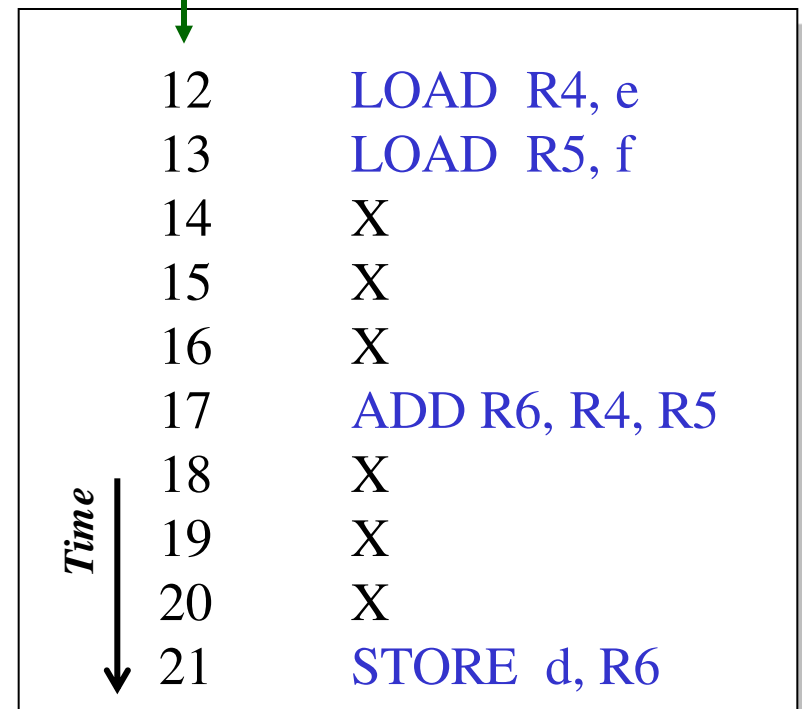
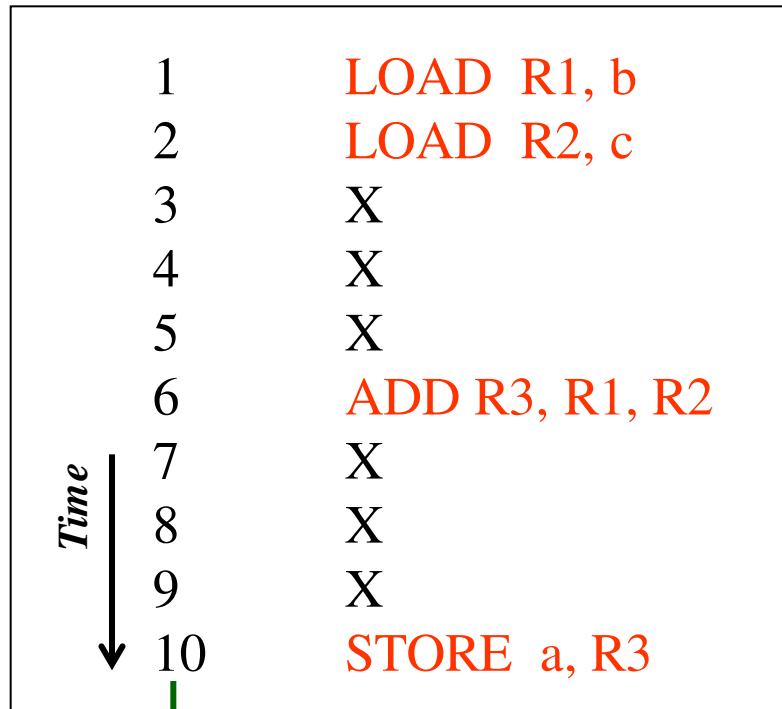
$a = b + c$

$d = e + f$

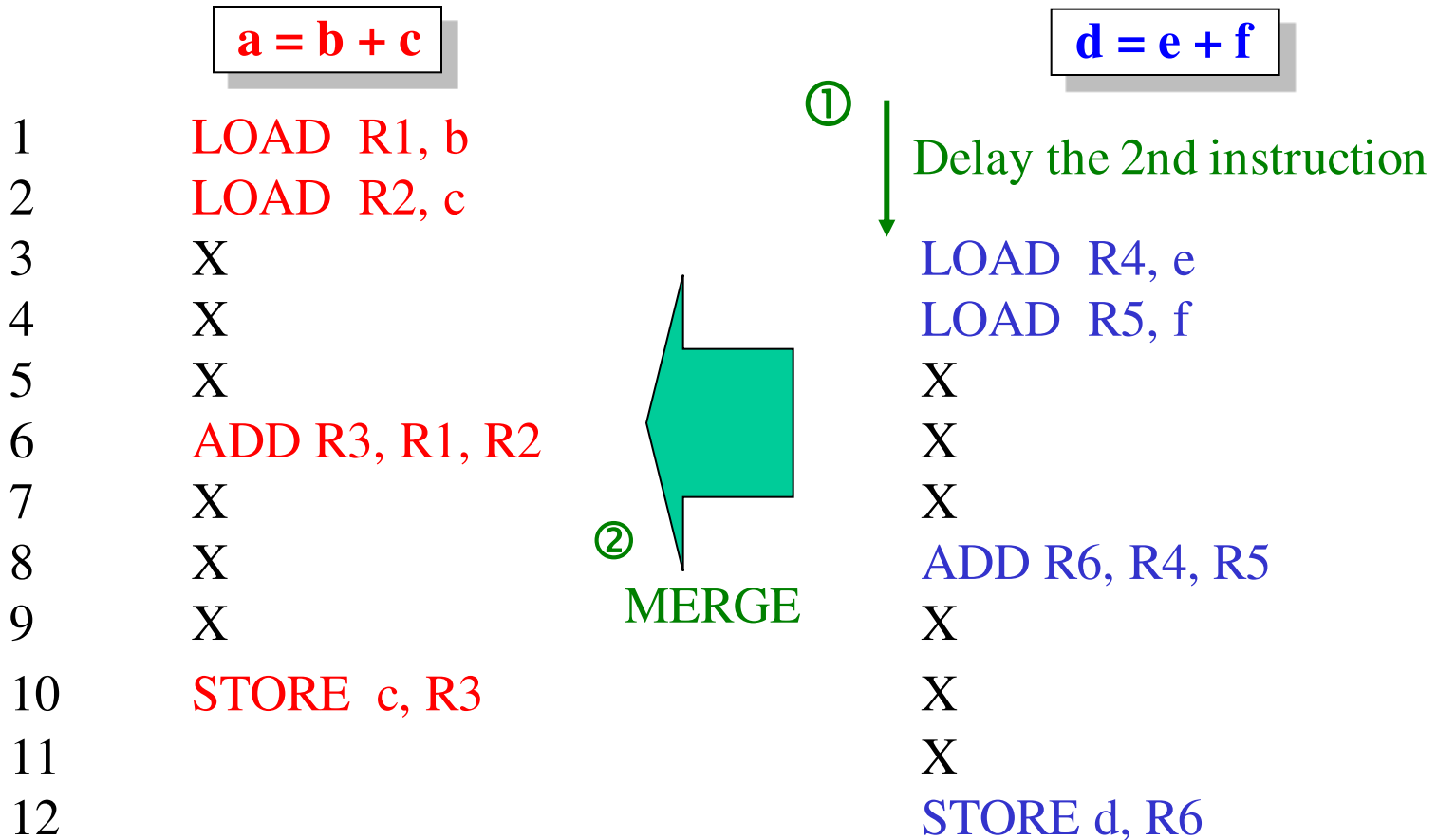
# Reducing pipeline hazards - three techniques

**a = b + c**

**d = e + f**



# Reducing pipeline hazards - three techniques



# Reducing pipeline hazards - three techniques

**a = b + c**

**d = e + f**

1	LOAD R1, b
2	LOAD R2, c
3	LOAD R4, e
4	LOAD R5, f
5	X
6	ADD R3, R1, R2
7	X
8	ADD R6, R4, R5
9	X
10	STORE c, R3
11	X
12	STORE d, R6

# Reducing pipeline hazards - three techniques

## Technique 3: Delayed Branch:

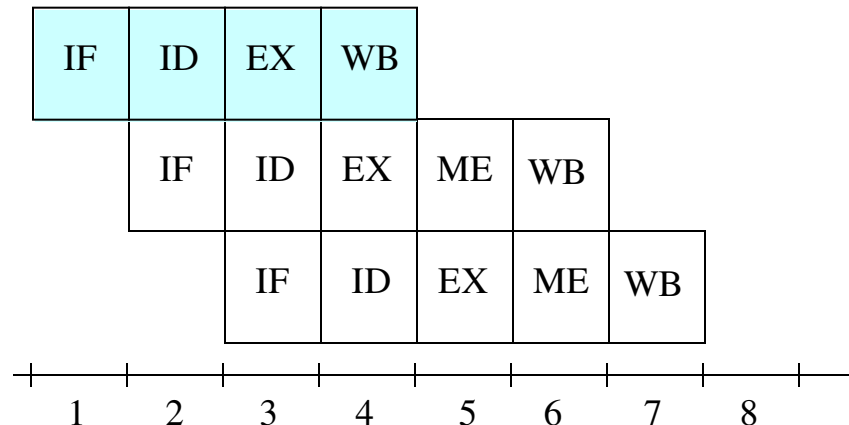
= Fill up clock cycles that will be flushed by a branch instruction

If branch NOT taken

Branch Instruction<sub>(i)</sub>:

Instruction<sub>(i+1)</sub>:

Instruction<sub>(i+2)</sub>:



# Reducing pipeline hazards - three techniques

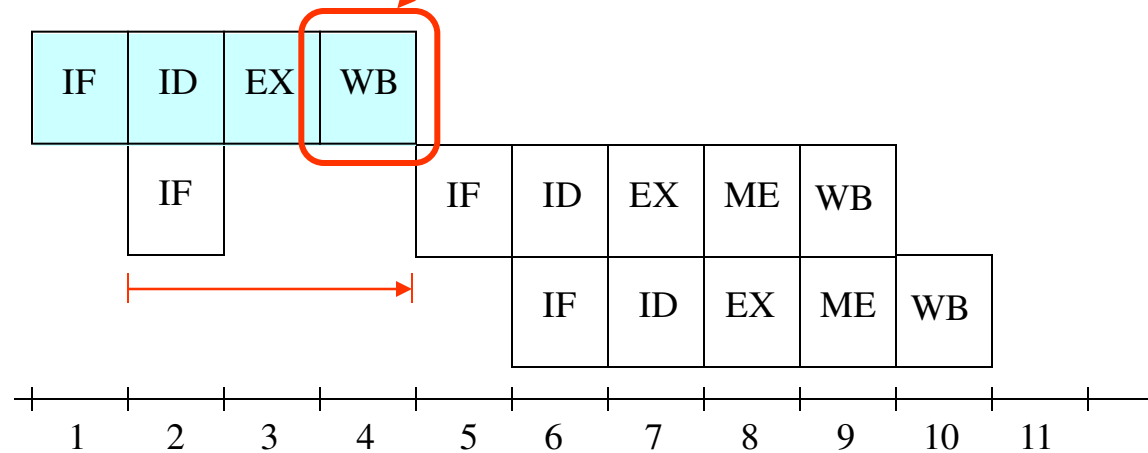
If branch taken

New destination address  
is set in PC

Branch Instruction<sub>(i)</sub>:

Instruction<sub>(i+1)</sub>:

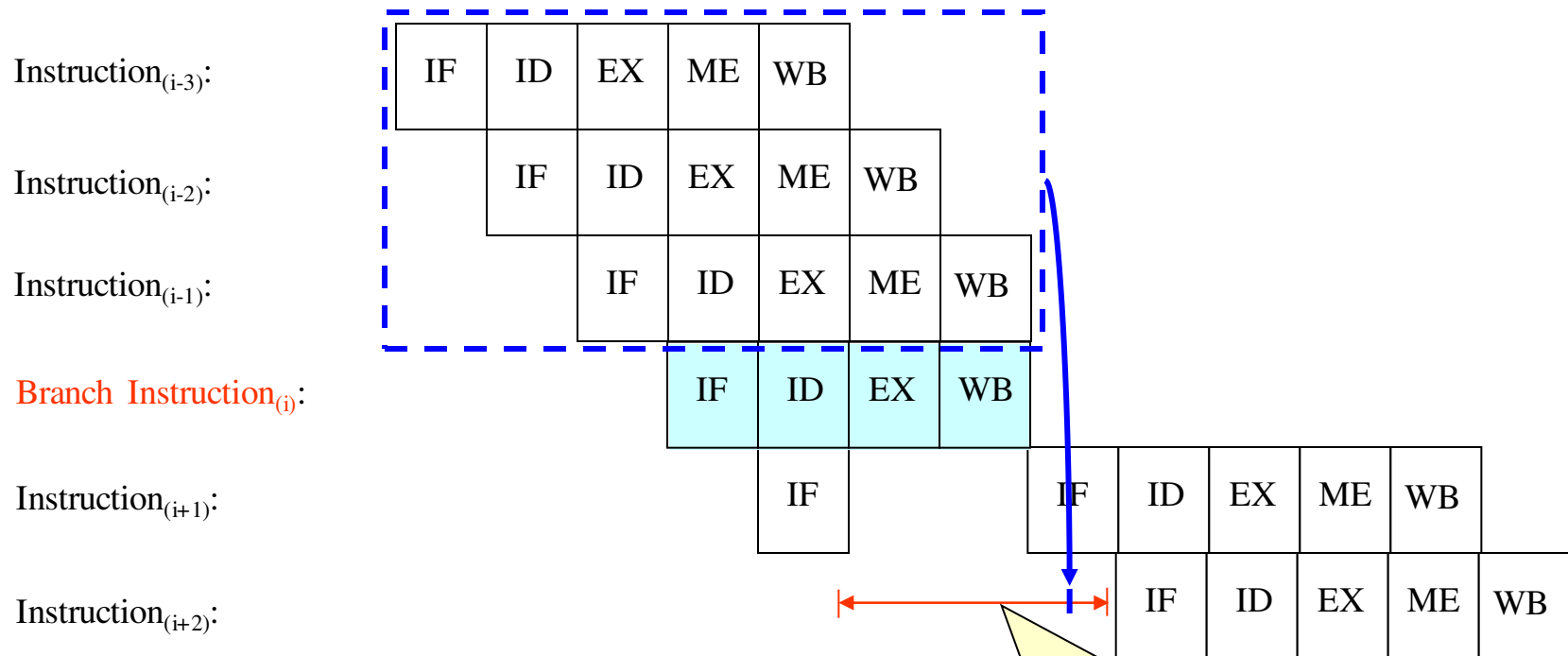
Instruction<sub>(i+2)</sub>:





# Reducing pipeline hazards - three techniques

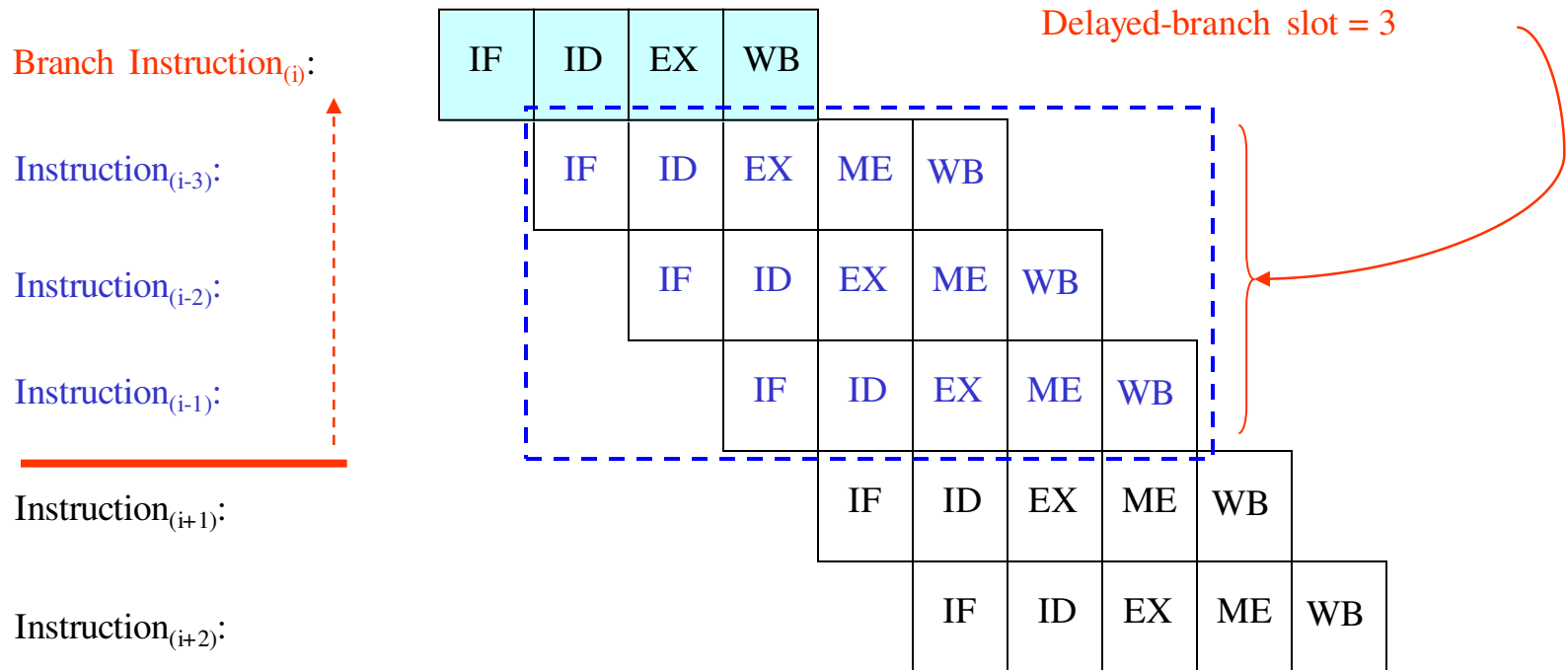
## Before Delayed Branch Applied



We are going to lose 3 cycles

# Reducing pipeline hazards - three techniques

## After Delayed Branch Applied



# Reducing pipeline hazards - three techniques

**Problem in delayed-branch:** data dependency to the branch instruction

Example:

SUB R1, R2, R3  
JPEZ R1  
LW R8, 0(R4)

Conditional branch  
(Jump if R1 = 0)

