

Introduction to Hazards in (computer Architecture)



Hazard in computer Architecture

In the domain of central processing unit(cpu) design, hazards are problems with the instruction pipeline in cpu microarchitecture when the

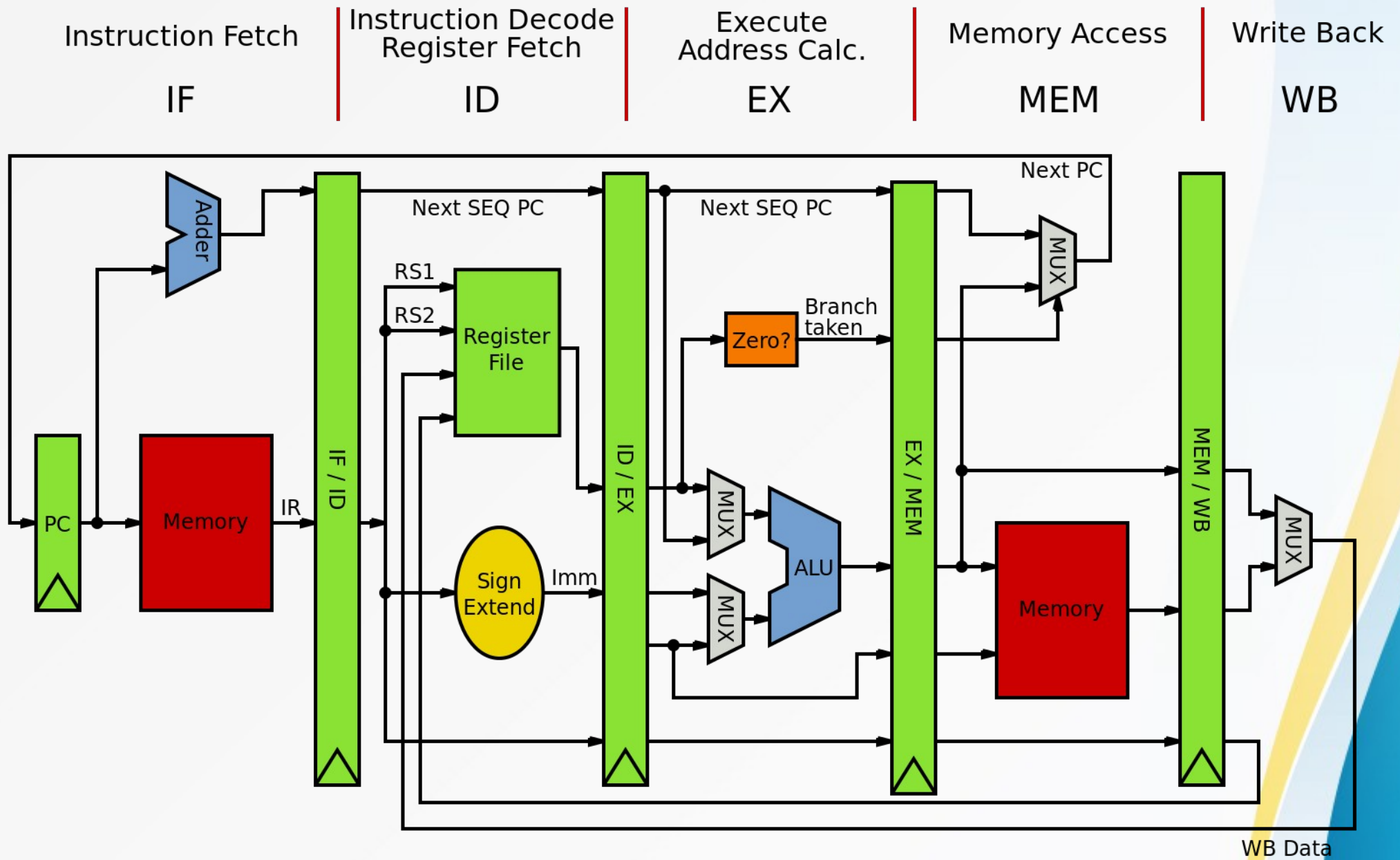
- **next instruction cannot execute in the following clock cycle and**
- **can potentially leads to incorrect computation results.**

Pipeline Hazards

- What is a pipeline hazard?
 - A situation that prevents an instruction from executing during its designated clock cycles.
- There are 3 classes of hazards:
 - Data Hazards
 - Structural Hazards
 - Control Hazards

Pipeline Hazards

- Hazards can result in incorrect operations:
 - 📖 Data hazards: Instruction depends on result of a prior instruction that is still in pipeline
 - ❖ Data dependency
 - 📖 Structural hazards: Two instructions requiring the same hardware unit at same time.
 - 📖 Control hazards: Caused by delay in decisions about changes in control flow (branches and jumps).
 - ❖ Control dependency



5 stage pipeline of a computer architecture

Data Hazards

- Occur when an instruction under execution depends on data from an instruction ahead in pipeline.

- Example:

$\mathcal{A} = \mathcal{B} + \mathcal{C}; \mathcal{D} = \mathcal{A} + \mathcal{E};$

$\mathcal{A} = \mathcal{B} + \mathcal{C};$

<i>IF</i>	<i>ID</i>	<i>EXE</i>	<i>MEM</i>	<i>WB</i>
-----------	-----------	------------	------------	-----------

$\mathcal{D} = \mathcal{A} + \mathcal{E};$

<i>IF</i>	<i>ID</i>	<i>Exe</i>	<i>Mem</i>	<i>WB</i>
-----------	-----------	------------	------------	-----------

Dependent instruction uses old data: Results in wrong computations



- **Stall:** A stall is commonly called a **pipeline bubble** or just bubble, since it floats through the pipeline taking space but carrying no useful work.


DEPENDENCE

- A hazard is a potential problem in a pipeline that may arise from a dependence.
- A dependence is a property of instructions in a program

Inter-Instruction Dependences



^ Data dependence


r_3  r_1 op r_2 *Read-after-Write*
 r_5  r_3 op r_4 (RAW)



True Dependency




^ Anti-dependence


r_3  r_1 op r_2 *Write-after-Read*
 r_1  r_4 op r_5 (WAR)



*Name
dependences
or*

^ Output dependence

r_3  r_1 op r_2 *Write-after-Write*
 r_5  r_3 op r_4 (WAW)
 r_3  r_6 op r_7



False Dependency

◆ Control dependence

Data Dependency

- Data dependency is a situation in which a program statement (Instruction) refers to the data of a preceding statement.
- An **instruction j** is a data dependent on **instruction i** if either of the following holds:
 - **Instruction i** produces a result that may be used by **instruction j**.
 - **Instruction j** is data dependent on **instruction k**, and **instruction k** is data dependent on **instruction i**.
- True dependency arises when one instruction use the value produced by an earlier instruction
- True dependency can not be eliminated however false dependency can be eliminated by register renaming.

Flow Dependency or data dependency

A flow Dependency also known as data Dependency or true Dependency or read after write(RAW) occurs when an instruction depends on the result of a previous instruction.

1.A=3

2.B=A

3. C=B

- Instruction 3 is truly dependent on instruction 2, as the final value of c depends on the instruction updating B.
- Instruction 2 is truly dependent on instruction 1, as the final value of B depends on the instruction updating A
- Hence instruction level parallelism can not be applied.

Anti Dependency

Anti Dependency also known as write after read(WAR) occurs when an instruction requires a value that is later updated

1.B=3

2.A=B+1

3. B=7

Instruction 2 is anti dependent on instruction 3, Renaming of variable can remove the anti dependency

An **antidependence** between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads.

1.B=3

N.B2=B

2.A=B2+1

3. B=7

Output Dependency

An output Dependency also known as write after write(WAW) occurs when the ordering of instructions will affect the final output value of a variable.

1.B=3

2.A=B+1

3. B=7

in the above example there is an output dependency between 3 and 1. changing the ordering of instruction will change the final value of A. thus the instruction cannot be executed in parallel.

An **output dependence** occurs when instruction i and instruction j write the same register or memory location.

Control Dependency

An instruction B has a control dependency on a preceding instruction A if the outcome of A determines wheather B should be executed or not.

Example:

A: bre \$s1,\$s2,Label

B: add \$s3,\$s4,\$s5

C: Label sub \$t0,\$t1,\$t2

Q. Find out all types dependencies for the following MIPS instructions

1.DADD R_1, R_2, R_3

2.LD $R_4, 0(R_1) : R4 \leftarrow M[R1+0]$

3.DADDUI $R_5, R_1, \#10$

4.SD $R_4, 0(R_5) : M[R5] \leftarrow R4$

5.OR R_4, R_3, R_5

6.AND R_1, R_6, R_7

Solution

given.

1. $DADD\ R_1, R_2, R_3$

2. $LD\ R_4, 0(R_1)$

3. $DAADUI\ R_5, R_4, \#10$

4. $SD\ R_4, 0(R_5)$

5. $OR\ R_4, R_3, R_5$

6. $AND\ R_1, R_6, R_7$

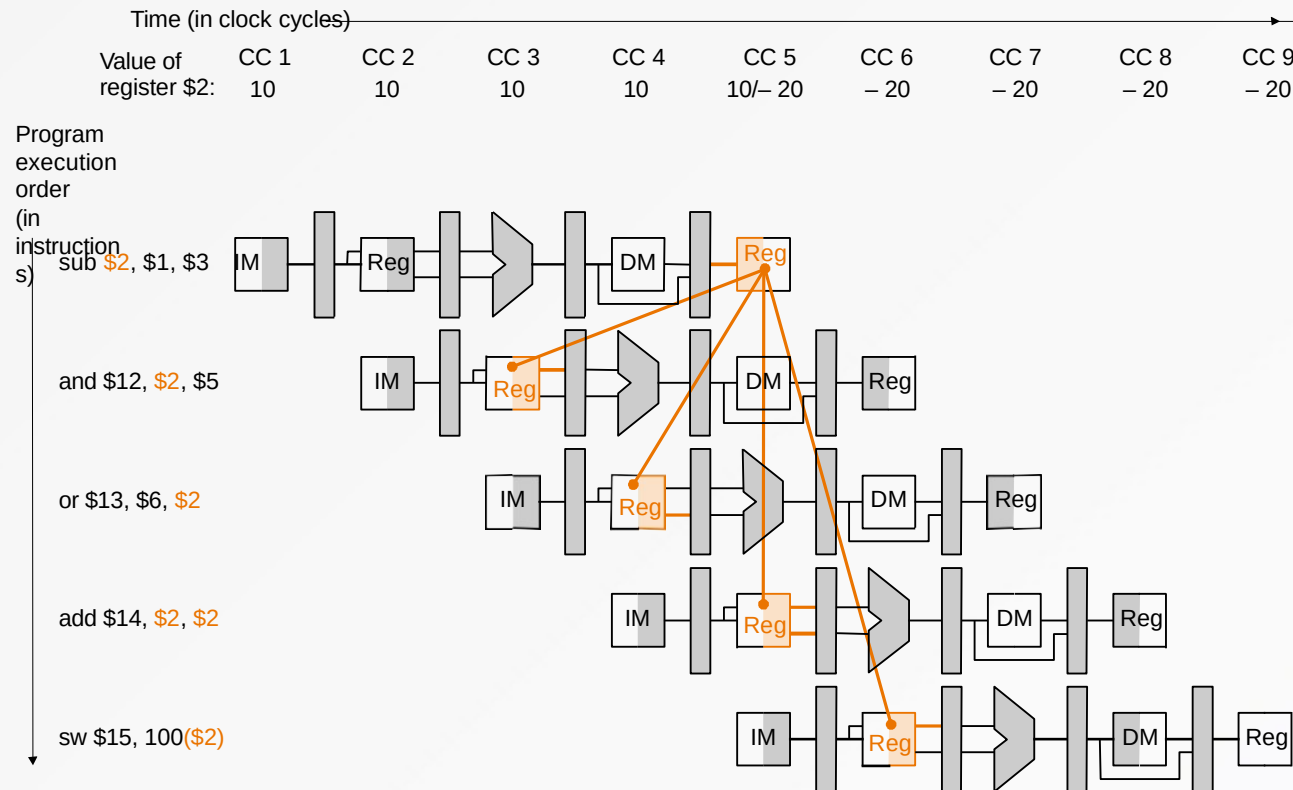
(RAW) True dependency : 1-2, 1-3, 2-4, 3-4, 3-5, 1-5.

(WAR) Anti dependency: 2-6, 3-6, 4-5.

(WAW) Write after write dependency: 1-2, 3-5.

Data Hazards

- Data hazards occur when data is used before it is ready*



The use of the result of the SUB instruction in the next three instructions causes a data hazard, since the register \$2 is not written until after those instructions read it.

Pipeline Hazards

Execution Order is:
Instr_i
Instr_j

Read After Write (RAW)

Instr_j tries to read operand before Instr_i writes it

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


- Caused by a “**Dependence**” (in compiler nomenclature).

Execution Order is:
Instr_i
Instr_j

Write After Read (WAR)

Instr_j tries to write operand before Instr_i reads it

- Gets wrong operand

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

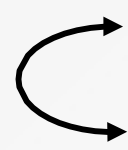
- Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

Execution Order is:
Instr_i
Instr_j

Write After Write (WAW)

Instr_j tries to write operand before Instr_i writes it

- Leaves wrong result (Instr_i not Instr_j)

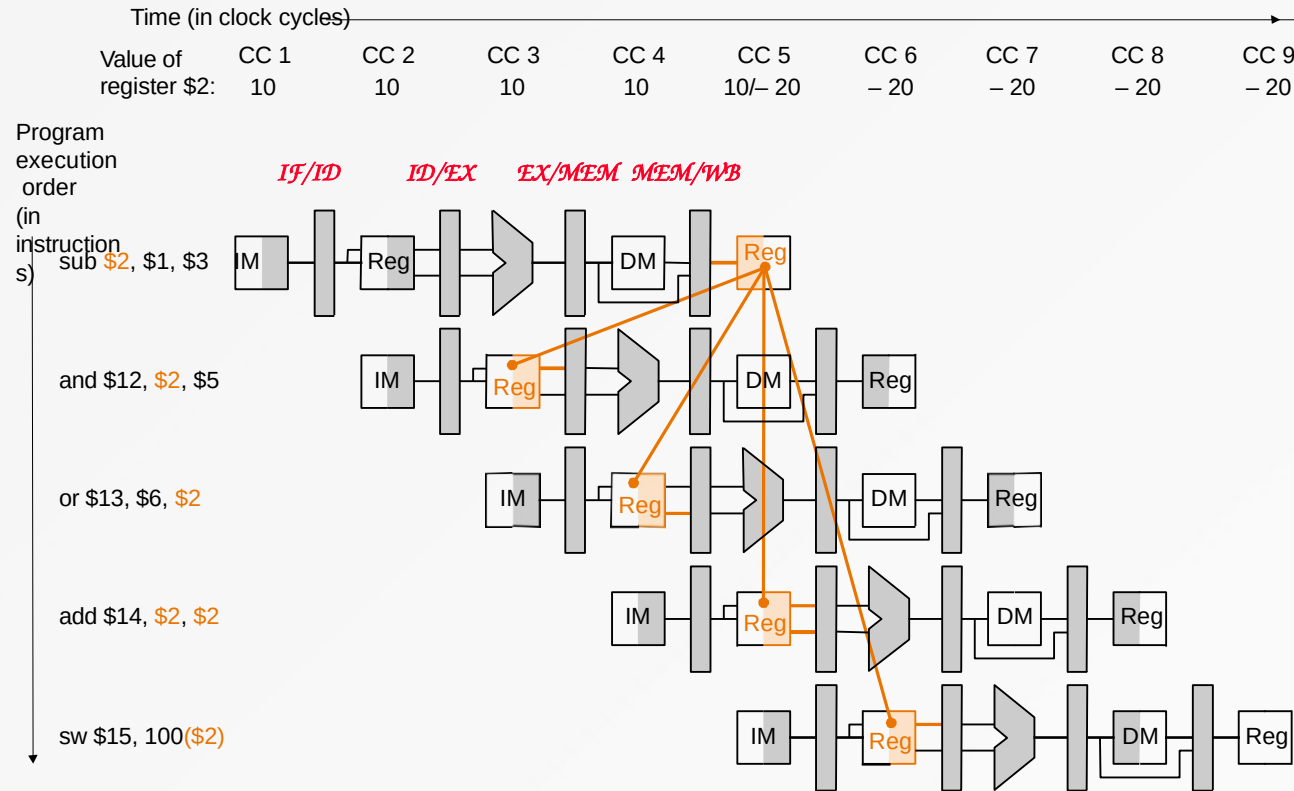


I: sub **r1**, r4, r3
J: add **r1**, r2, r3
K: mul r6, r1, r7

- Called an “**output dependence**” by compiler writers
This also results from the reuse of name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5
-

Data Hazard Detection in MIPS

Read after Write



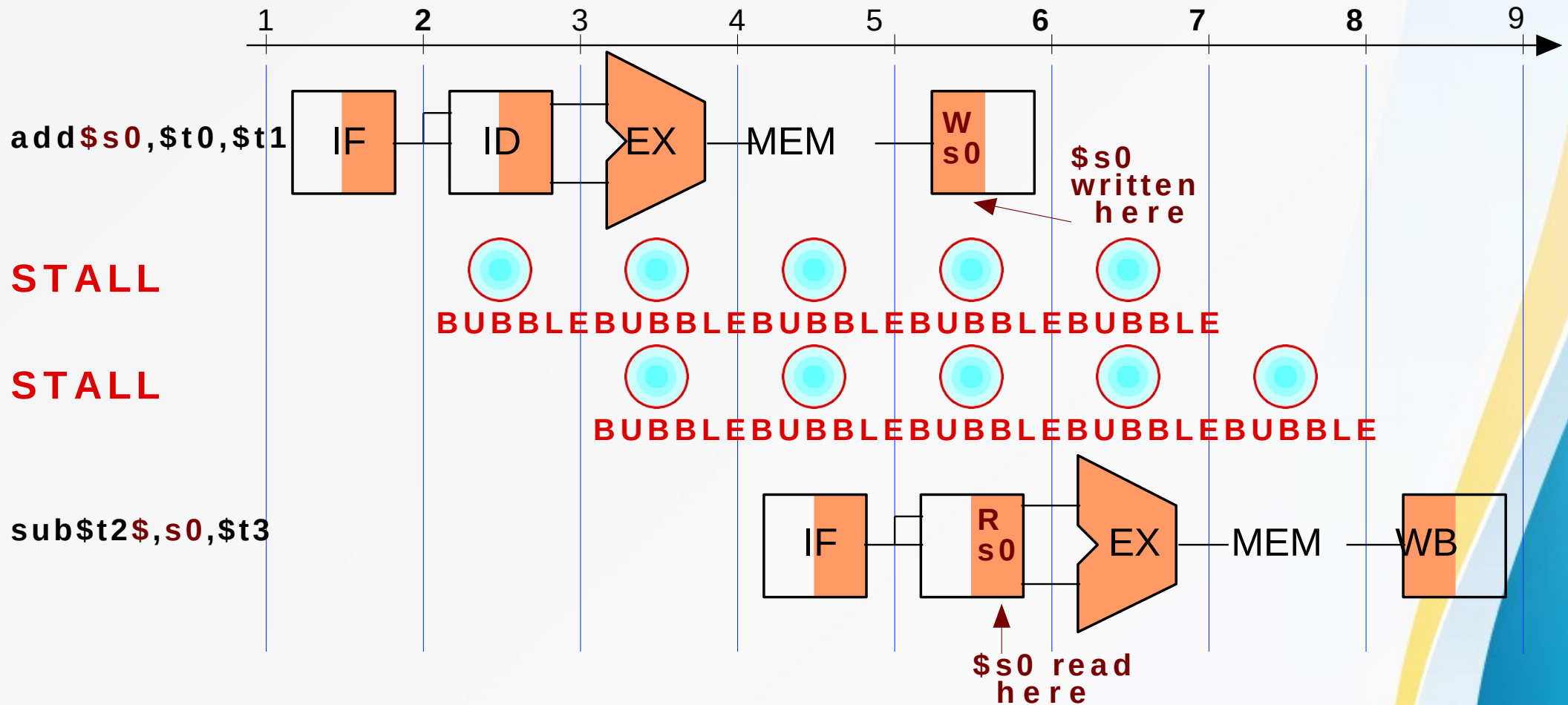
Data Hazards

- **Solutions for Data Hazards**
 - **Stalling**
 - **Forwarding:**
 - » connect new value directly to next stage
 - **Reordering**

Handling data hazard by S/W(stalling)

- *Compiler introduce **NOP** in between two instructions*
- *NOP = a piece of code which keeps a gap between two instruction*
- *Detection of the dependency is left entirely on the S/W*
- *Advantage :- We find the easy technique called as instruction reordering.*

Data Hazard - Stalling



Forwarding

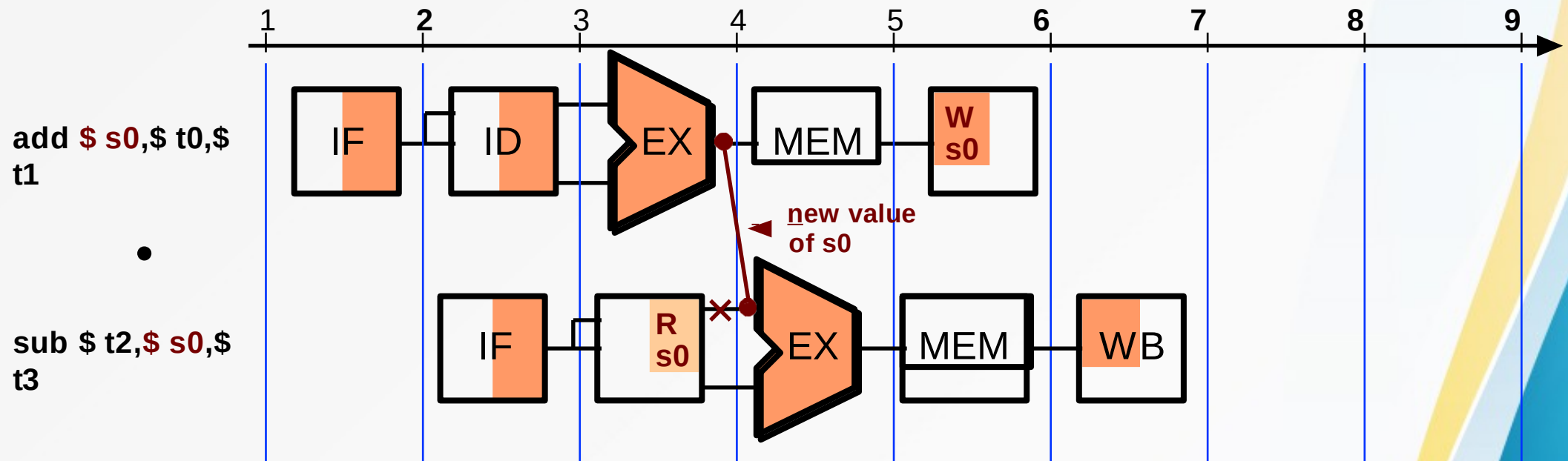
- Generally speaking:
 - Forwarding occurs when a result is passed directly to the functional unit that requires it.
 - Result goes from output of one pipeline stage to input of another.

Forwarding

- Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?
 - **Yes!**
 - The ALU result from both the EX/MEM and MEM/WB pipeline registers is always feedback to the ALU inputs

Data Hazards - Forwarding

- Key idea: connect new value directly to next stage
- Still read s0, but ignore in favor of new result

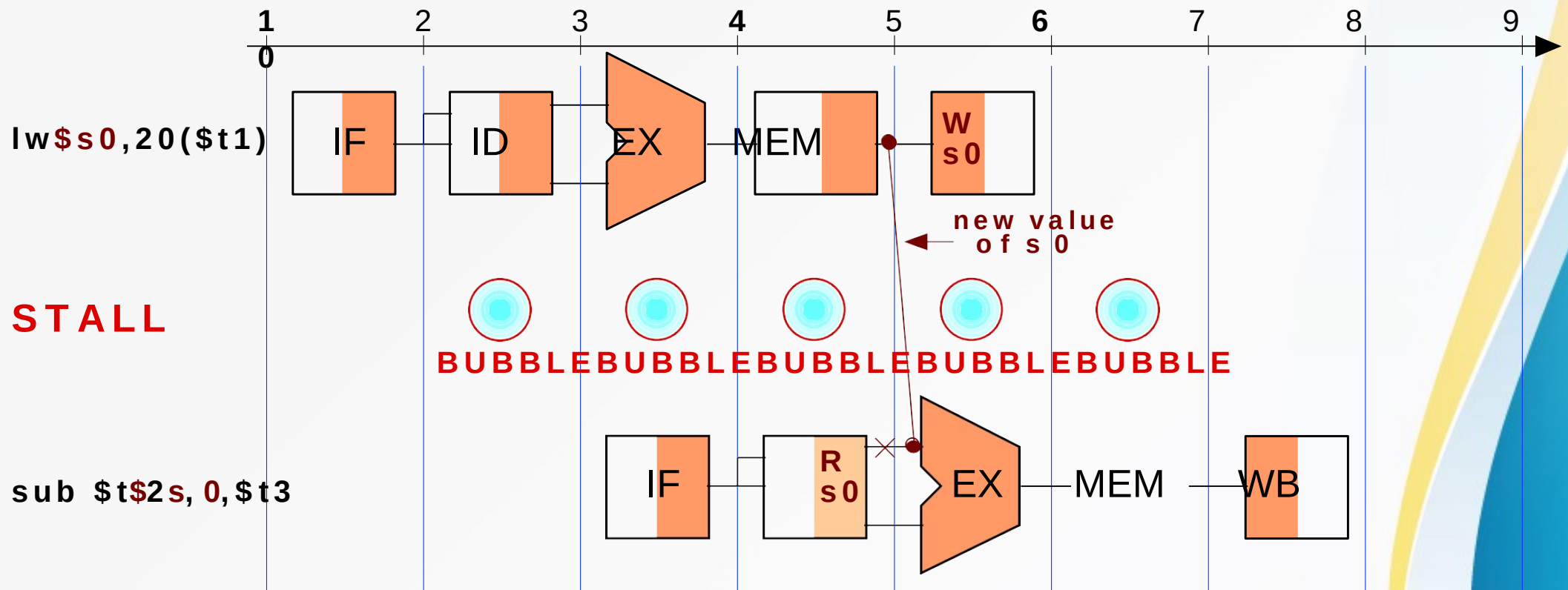


- **Problem: what about load instructions?**

Pipeline Hazards

Data Hazards - Forwarding

- STALL still required for load - data avail. after MEM
- MIPS architecture calls this delayed load,



Pipeline Hazards

Instruction Reordering

- ADD R1 , R2 , R3
 - SUB R4 , R1 , R5
 - XOR R8 , R6 , R7
 - ~~AND R9 , R10 , R11~~
- Before**

- ADD R1 , R2 , R3
 - XOR R8 , R6 , R7
 - AND R9 , R10 , R11
 - SUB R4 , R1 , R5
- After**

Structural Hazards

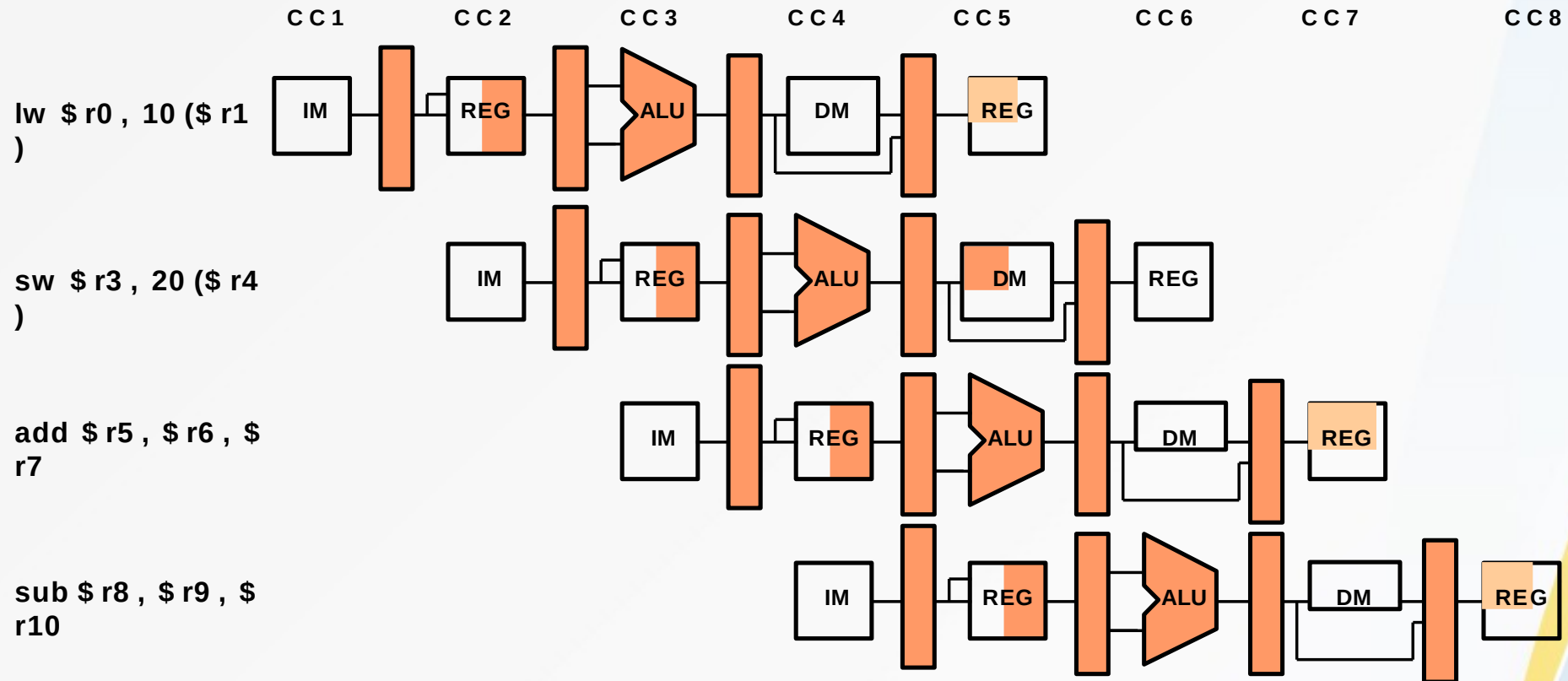
- **Attempt to use the same resource by two or more instructions at the same time**
- **Example: Single Memory for instructions and data**
 - Accessed by IF stage
 - Accessed at same time by MEM stage
- **Solutions**
 - Delay the second access by one clock cycle, OR
 - Provide separate memories for instructions & data
 - » This is called a “**Harvard Architecture**”
 - » Real pipelined processors have separate **caches**

Pipelined Example - Executing Multiple Instructions

- **Consider the following instruction sequence:**

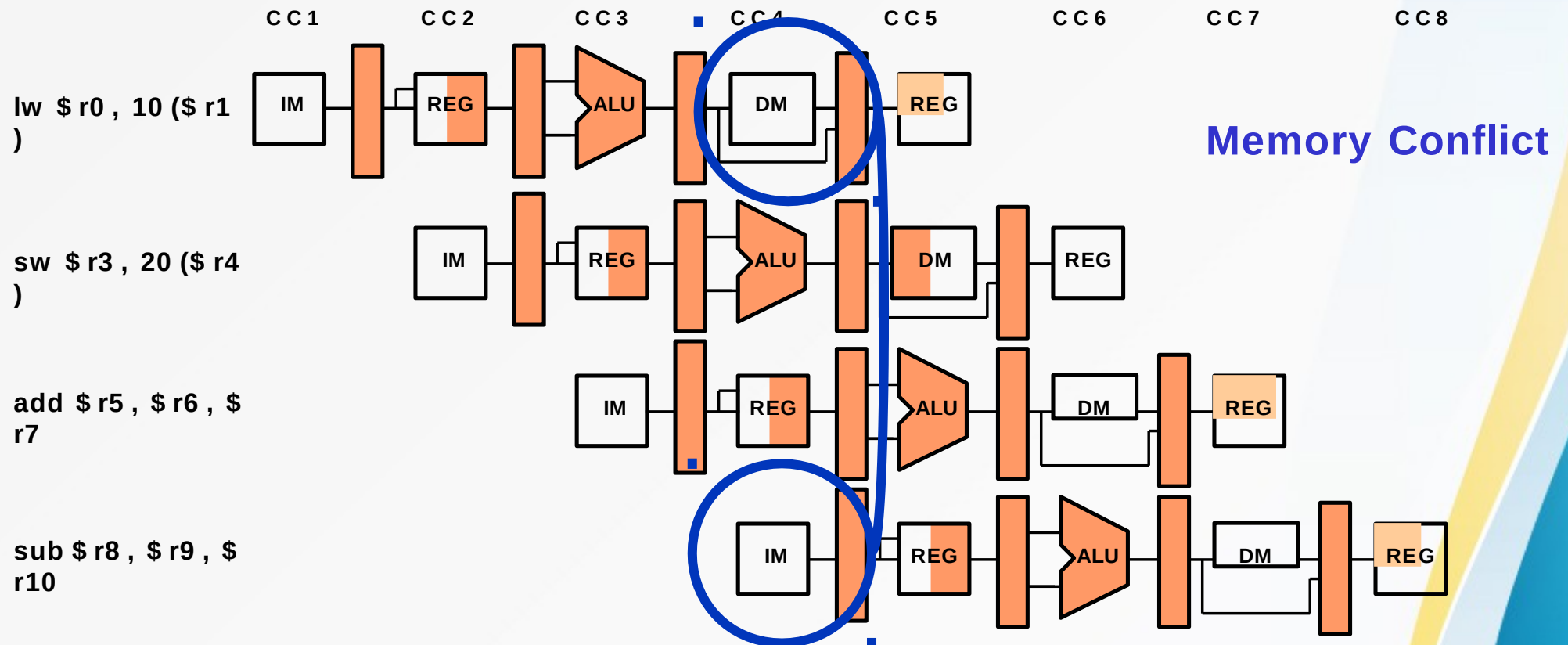
```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```

Alternative View - Multicycle Diagram



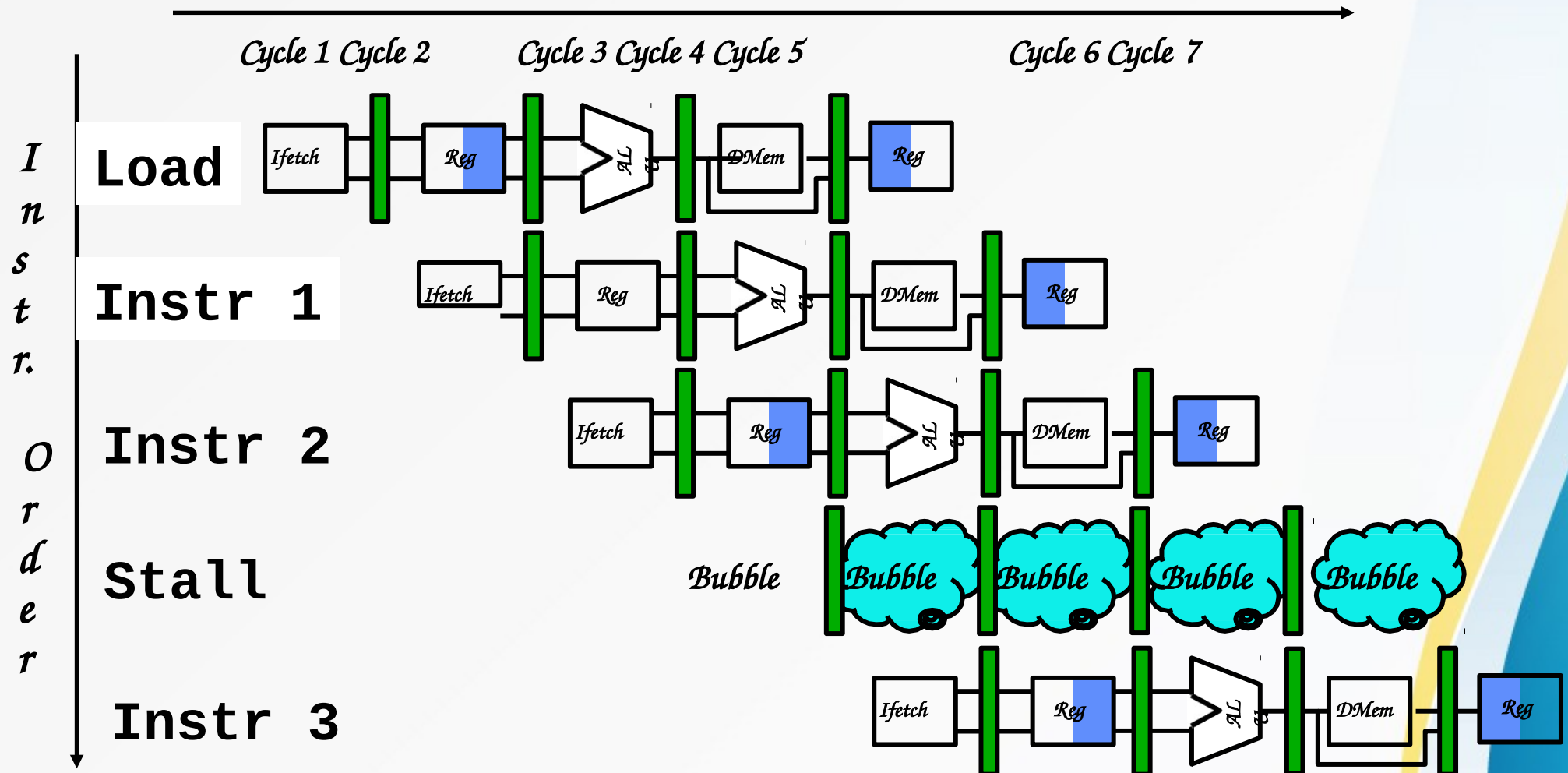
Pipeline Hazards

Alternative View - Multicycle Diagram



One Memory Port Structural Hazards

Time (clock cycles)



Pipeline Hazards

solution to Structural Hazards

Dealing with Structural Hazards

Stall

- low cost, simple
- Increases CPI
- use for rare case

Pipeline hardware resource

- since stalling has performance effect
- useful for multi-cycle resources
- good performance
-

Replicate resource

- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

NUMERICALS

1. Calculate the no. of stall without operand forwarding and with forwarding

DADD R1, R2, R3

LD R4, 0(R1)

SD R4, 12(R1)

2. Calculate the no. of stall without operand forwarding and with forwarding

DADD R1, R2, R3

DSUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11

3. Calculate the no. of stall without operand forwarding and with forwarding

LD R1, 0(R2)

DSUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

4.

A five stage pipeline processor has IF, ID, EXE, MEM, WB. The IF, ID, MEM, WB stages takes 1 clock cycles each for any instruction. The EXE stage takes 1 clock cycle for ADD & SUB instructions, 2 clock cycles for MUL instructions and 3 clock cycles for DIV instructions respectively.

Consider the following instructions:-

DIV	R1, R2, R4
ADD	R5, R1, R6
SUB	R7, R1, R8
MUL	R9, R1, R10
ADD	R8, R1, R7

For the above sequence:-

- (i) Find out all type of data dependency?
- (ii) Find out total number of clock cycles required to complete the execution, if operand forwarding is used?

Control Hazards

A *control hazard* is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

A branch is either

Taken: $PC \leq PC + 4 + \text{Immediate}$

Not Taken: $PC \leq PC + 4$

- Control hazards can cause a greater performance loss for pipeline than data hazards. When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4.
- If a branch changes the PC to its target address, it is a **taken branch**; if it falls through, it is **not taken**.
- If instruction i is a **taken branch**, then the PC is normally not changed until the end of ID stage, after the completion of the address calculation and comparison (RISC processor)

Four Simple Control/Branch Hazard Solutions

These following solutions assume that we are dealing with **Static Branches (Compile time)**.

1. Flush Pipeline/ Stall
2. Predict Branch Not Taken(Untaken):
3. Predict Branch Taken
4. Delayed branch.

Solution-1. Stall pipeline:

- The simplest method of dealing with branches is to **stall the pipeline** as soon as the branch is detected until we reach the ID stage, which determines the new PC.
- The simplest scheme to handle branches is to freeze or flush the pipeline, holding or deleting any instructions after the branch until the branch destination is known.
- Advantage: simple both to software and hardware
- Let us see an example, we will stall the pipeline until the branch is resolved (in that case we repeated the IF stage until the branch is resolved and modifies the PC)

- **Performing IF Twice:** We take a big performance hit by performing the instruction fetch whenever a branch occurs. Note, this happens even if the branch is taken or not. This guarantees that the PC will get the correct value.

IF ID EX MEM WB

branch

IF ID EX MEM WB

IF IF ID EX MEM WB

Branch Prediction Schemes

- **Predict taken**
- **Predict not taken**
- **Delayed branch**

Solution-2. Predict Not Taken

- A higher performance, and only slightly more complex, scheme is to **predict the branch as not taken**, simply allowing the hardware to continue as if the branch were not executed. Care must be taken not to change the machine state until the branch outcome is definitely known.
- What if we treat every branch as “not taken” remember that not only do we read the registers during ID, but we also perform an equality test in case we need to branch or not.
- We can improve performance by assuming that the branch will not be taken.
 - Execute successor instructions in sequence as if there is no branch
 - undo instructions in pipeline if branch actually taken
- The “branch-not taken” scheme is the same as performing the IF stage a second time in our 5 stage pipeline if the branch is taken.
- If not there is no performance degradation.
- 47% branches not taken on average

The pipeline with this scheme implemented behaves as shown below:

<i>Untaken</i> Branch Instr	IF	ID	EX	MEM	WB			
Instr i+1		IF	ID	EX	MEM	WB		
Instr i+2			IF	ID	EX	MEM	WB	

<i>Taken</i> Branch Instr	IF	ID	EX	MEM	WB			
Instr i+1		IF	<i>idle</i>	<i>idle</i>	<i>idle</i>	<i>idle</i>		
Branch target			IF	ID	EX	MEM	WB	
Branch target+1				IF	ID	EX	MEM	WB

When branch is not taken, determined during ID, we have fetched the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall one clock cycle.

Solution-3: Predict Taken

- An alternative scheme is **to predict the branch as taken**. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target address.
- Because in pipeline **the target address is not known any earlier than the branch outcome**, there is **no advantage in this approach**. In some machines where the target address is known before the branch outcome a predict-taken scheme might make sense.
- incurs 1 cycle branch penalty even with predict taken
- 53% branches taken on average.

Solution-4: Delayed branch

- The fourth method for dealing with a control hazard is to implement a “delayed” branch scheme.
- In this scheme an instruction is inserted into the pipeline that is useful and **not dependent** on whether the branch is taken or not. It is the job of the compiler to determine the delayed branch instruction.
- If the branch is actually taken, we need to clear the pipeline of any code loaded in from the “not-taken” path.

In a delayed branch, the execution cycle with a branch delay of length n is

Branch instr

sequential successor 1

sequential successor 2

.....

sequential successor n

Branch target (if taken)

- Sequential successors are in the branch-delay slots. These instructions are executed whether or not the branch is taken.
- Insert unrelated successor in the branch delay slot

- Delayed branches – code rearranged by compiler to place independent instruction after every branch (in delay slot).

```
add $R4,$R5,$R6  
beq $R1,$R2,20  
lw $R3,400($R0)
```



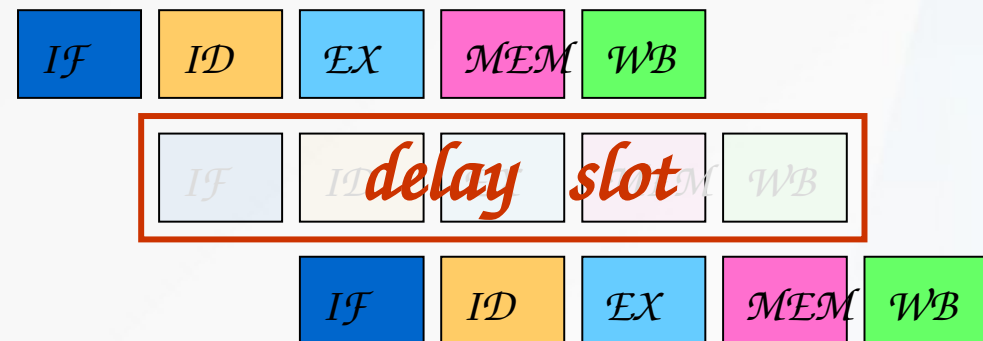
```
beq $R1,$R2,20  
add $R4,$R5,$R6  
lw $R3,400($R0)
```


- **Simple idea:** Put an instruction that would be executed anyway right after a branch.

Branch

Delayed slot instruction

Branch target OR successor



- **Question:** What instruction do we put in the delay slot?
- **Answer:** **one that can safely be executed no matter what the branch does.**
 - The compiler decides this.

Three branch-scheduling schemes in branch delay slot

- **From before branch**
- **From target**
- **From fall through**

From before branch

Branch must not depend
on the rescheduled instructions

From target

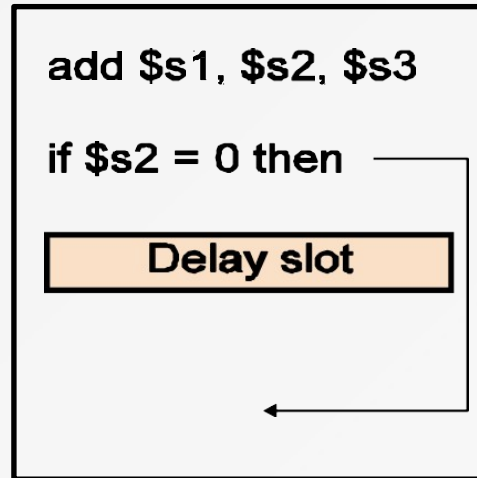
Must be OK to execute rescheduled
instructions if branch is not taken

From fall though

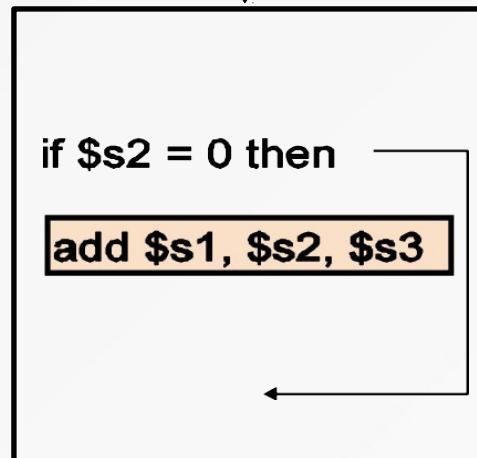
Must be OK to execute instructions if
branch is taken

Scheduling the Delay Slot

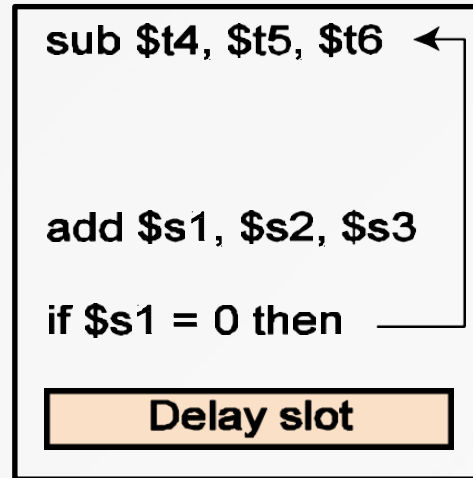
a. From before



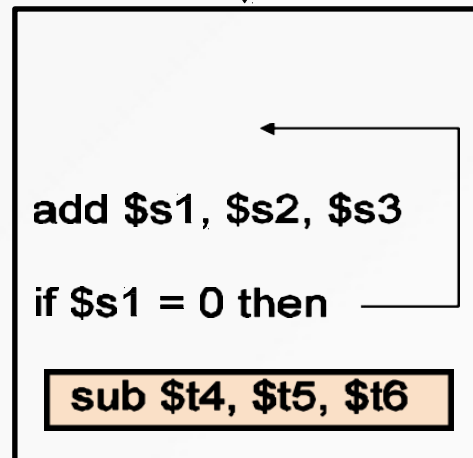
Become s



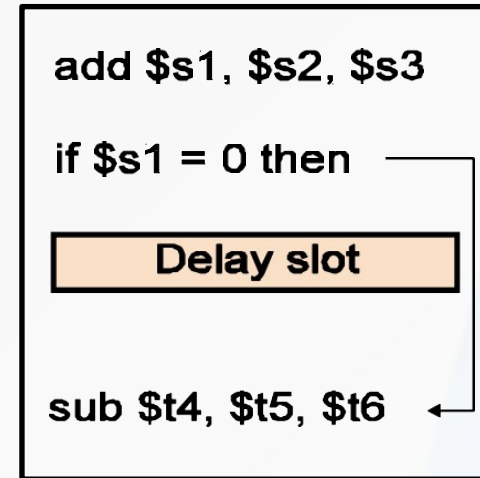
b. From target



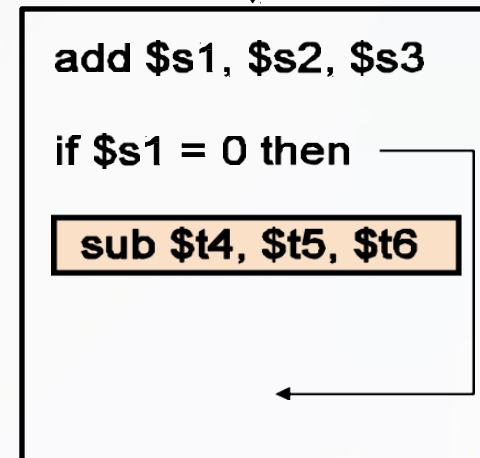
Becomes



c. From fall through

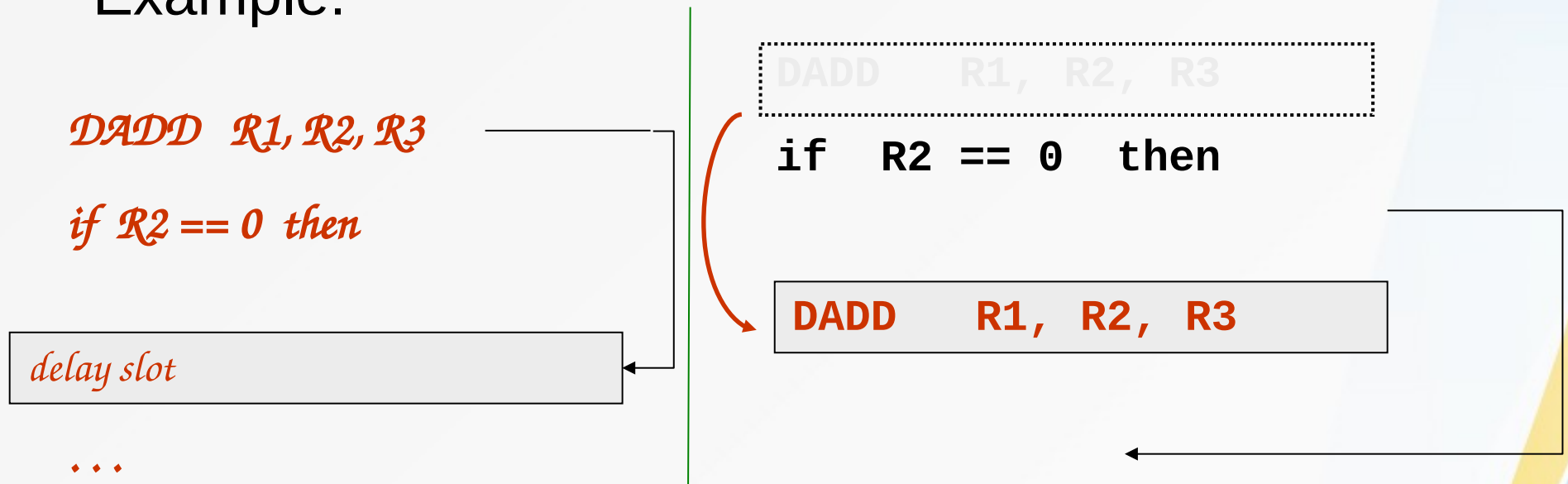


Becomes



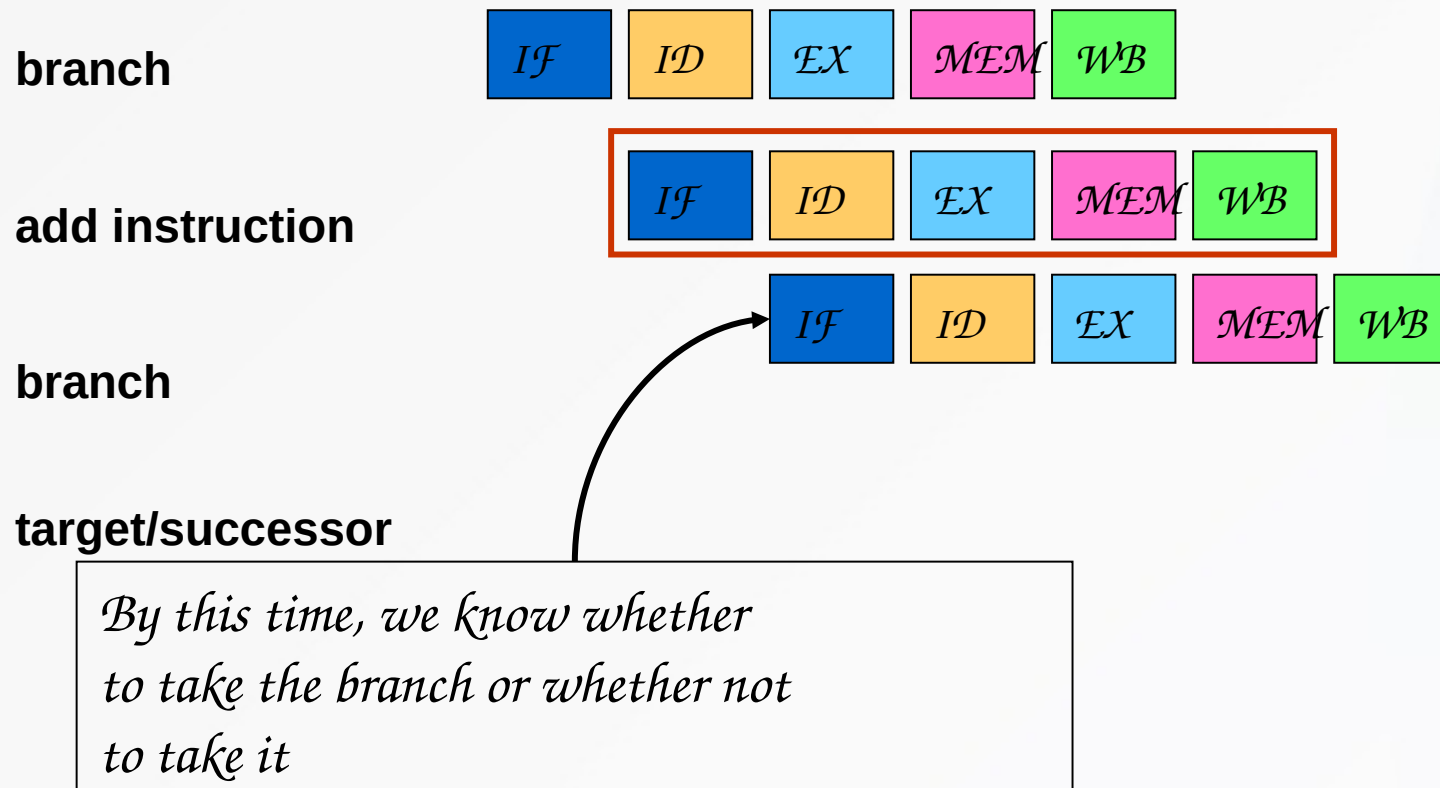
Delayed Branch from Before Branch

- One possibility: An instruction **from before**
- Example:



- The DADD instruction is executed no matter what happens in the branch:
 - Because it is executed before the branch!
 - Therefore, it can be moved

- We get to execute the “DADD” execution “for free”



Delayed Branch from target

- Another possibility: An instruction much **before** from target

- Example:

DSUB R4, R5, R6

...

DADD R1, R2, R3

if R1 == 0 then

delay slot



- The DSUB instruction can be replicated into the delay slot, and the branch target can be changed

- Example: ...
 DADD R1, R2, R3
 if R1 == 0 then

DSUB R4, R5, R6

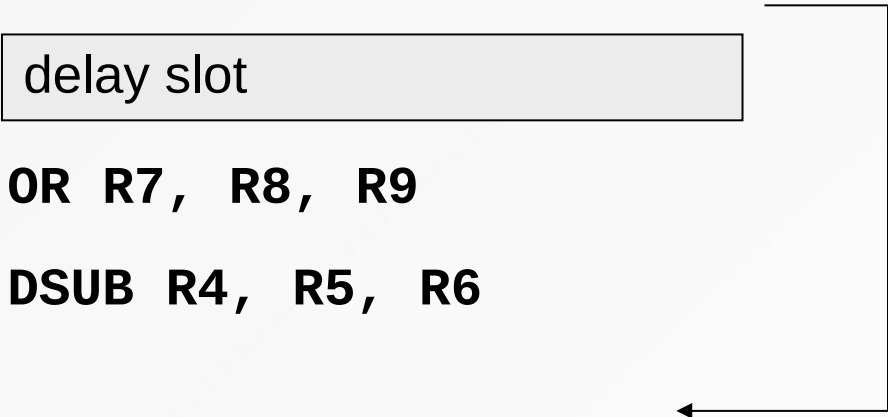
- The DSUB instruction can be replicated into the delay slot, and the branch target can be changed

Delayed Branch from fall through

- Yet another possibility: An instruction from inside the taken path: **fall through**

• Example:

```
DADD R1, R2, R3  
if R1 == 0 then  
    delay slot  
OR R7, R8, R9  
DSUB R4, R5, R6
```



The diagram illustrates a branch instruction. A box labeled 'delay slot' is positioned between the 'if' statement and the 'OR' instruction. A line connects the 'if' statement to the 'OR' instruction, bypassing the 'delay slot' box, indicating that the branch is taken and the instruction in the delay slot is not executed.

- The OR instruction can be moved into the delay slot **ONLY IF** its execution doesn't disrupt the program execution (e.g., R7 is overwritten later)

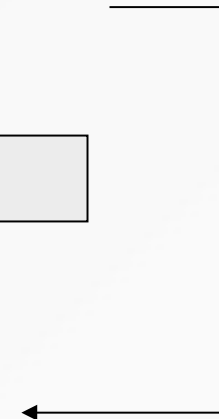
- Third possibility: An instruction from inside the taken path

DADD R1, R2, R3

- Example: **if R1 == 0 then**

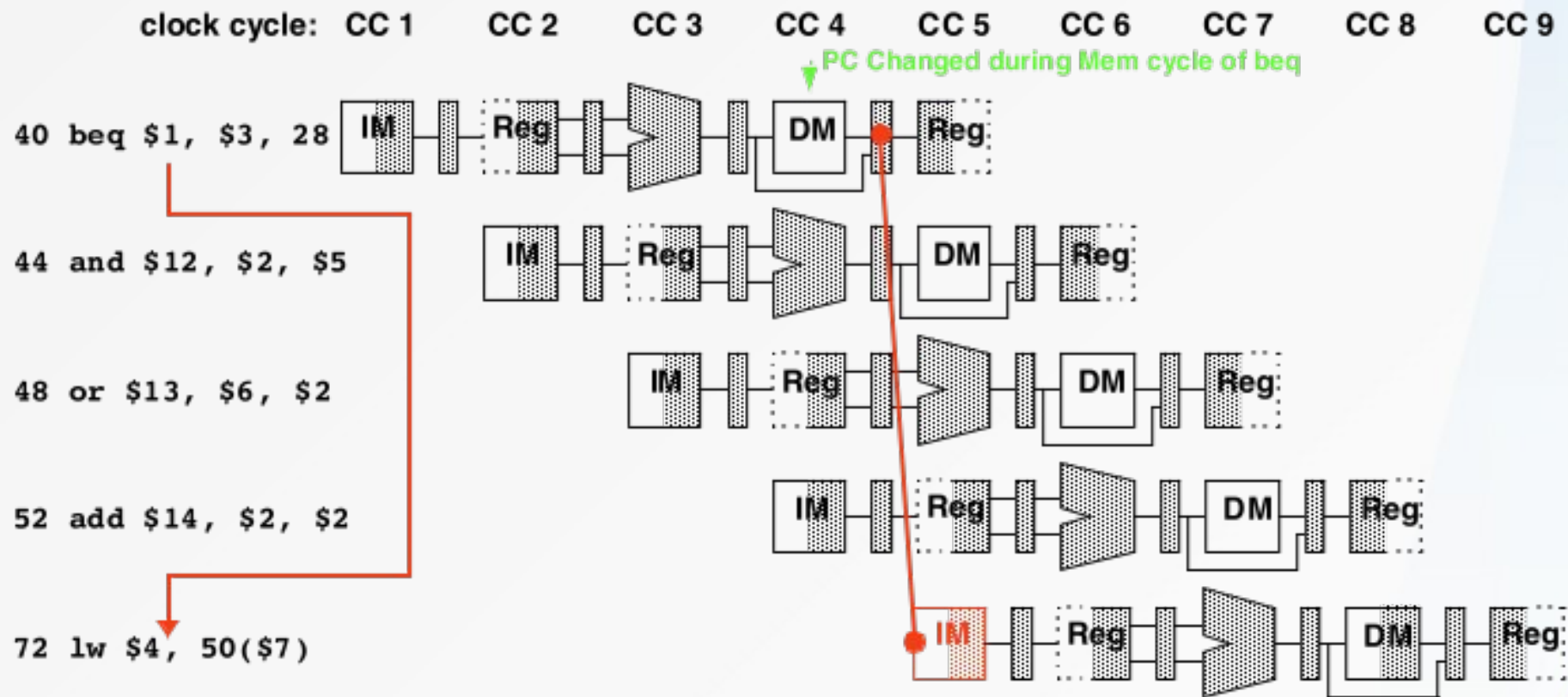
OR R7, R8, R9

DSUB R4, R5, R6



- The OR instruction can be moved into the delay slot **ONLY IF** its execution doesn't disrupt the program execution (e.g., R7 is overwritten later)

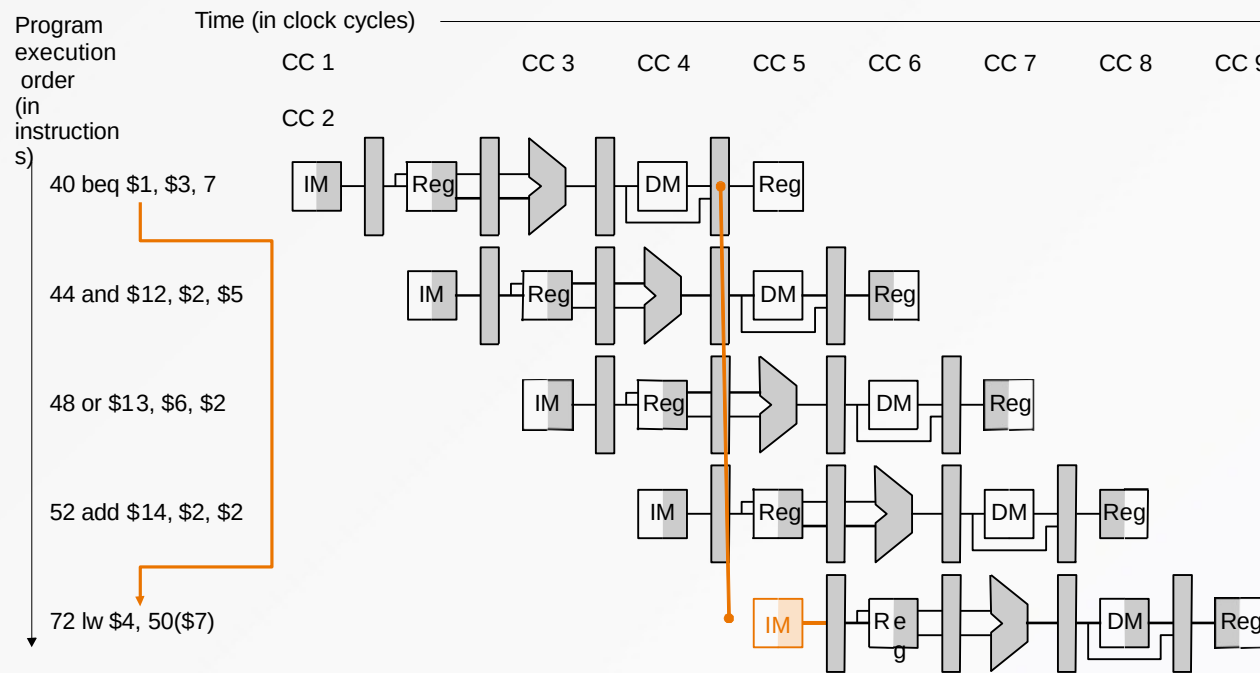
How branches impact pipelined instructions in MIPS Processor



- If branch condition true, must skip 44, 48, 52
 - But, these have already started down the pipeline
 - They will complete unless we do something about it

Branch Hazards

- Just stalling for each branch is not practical
- Common assumption: branch not taken
- When assumption fails: flush three instructions



Penalty for MIPS processor

Calculate CPI for each branch scheme.

Branch Scheme	Penalty for unconditional branches	Penalty for conditional branches (if Untaken)	Penalty for conditional branches (if taken)
Flush Pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Types of branch	Frequency
Unconditional branch	4%
Conditional Branch, untaken	6%
Conditional Branch, taken	10%

Performance of branch with Stalls

- Stalls degrade performance of a pipeline:
 - Result in deviation from 1 instruction executing/clock cycle.
 - Let's examine by how much stalls can impact CPI...
- $\text{CPI}_{\text{pipelined}} =$
 - $= \text{Ideal CPI} + \text{Pipeline stall cycles per instruction}$
 - $= 1 + \text{Pipeline stall cycles per instruction}$
- $\text{CPI}_{\text{unpipelined}} = \text{Pipeline depth} = \text{no. of stages}$

Performance of branch schemes

- Pipeline speed up=

$$\frac{\text{Pipeline depth}}{1 + \text{pipeline stall cycle from branch}}$$

- Pipeline stall cycle from branches = Branch frequency * branch penalty
- Pipeline speed up =

$$\frac{\text{Pipeline depth}}{1 + \text{Branch frequency} * \text{Branch Penalty}}$$

- Additions to the CPI from branch costs

Branch Scheme	unconditional branches (4%)	Untaken Conditional branches (6%)	Taken Conditional branches (10%)	All branches (20%)
Flush Pipeline	$2 \times 0.04 = 0.08$	$3 \times 0.06 = 0.18$	$3 \times 0.1 = 0.3$	$0.08 + 0.18 + 0.3 = 0.56$
Predicted taken	$2 \times 0.04 = 0.08$	$3 \times 0.06 = 0.18$	$2 \times 0.1 = 0.2$	$0.08 + 0.18 + 0.2 = 0.46$
Predicted untaken	$2 \times 0.04 = 0.08$	0	$3 \times 0.1 = 0.3$	$0.08 + 0 + 0.3 = 0.38$

- If flush pipeline is used then $\text{CPI} = 1 + 0.56 = 1.56$
- If predicted taken is used then $\text{CPI} = 1 + 0.46 = 1.46$
- If predicted taken is used then $\text{CPI} = 1 + 0.38 = 1.38$

Question: An Example of Computing Performance

- Program assumptions:
 - 23% loads and in $\frac{1}{2}$ of cases, next instruction uses load value
 - 13% stores
 - 19% conditional branches
 - 2% unconditional branches
 - 43% other
 - 5 stage pipe
 - Penalty of 1 cycle on use of load value immediately after a load.
 - Jumps are resolved in ID stage for a 1 cycle branch penalty.
 - 75% branch prediction accuracy.
 - 1 cycle delay on misprediction.

- CPI penalty calculation:
 - Loads:
 - 50% of the 23% of loads have 1 cycle penalty: $.5 \times .23 = 0.115$
 - Jumps:
 - All of the 2% of jumps have 1 cycle penalty: $0.02 \times 1 = 0.02$
 - Conditional Branches:
 - 25% of the 19% are mispredicted, have a 1 cycle penalty: $0.25 \times 0.19 \times 1 = 0.0475$
- Total Penalty: $0.115 + 0.02 + 0.0475 = 0.1825$
- Average CPI: $1 + 0.1825 = 1.1825$

An Example of Impact of Branch Penalty

- Assume for a MIPS pipeline:
 - 16% of all instructions are branches:
 - 4% unconditional branches: 3 cycle penalty
 - 12% conditional: 50% taken: 3 cycle penalty

Impact of Branch Penalty

Answer:

$3 * 0.04$ due to unconditional branches

$0.12 * 0.5 * 3$ due to conditional taken

- Overall CPI = $1 + (3 * 0.04 + 0.12 * 0.5 * 3) = 1.3$