

(a) Data Hazards

- Data hazards occur due to data dependencies between instructions that are in various stages of execution in the pipeline.
- Example:

Instruction	1	2	3	4	5	6
ADD R2,R5,R8	IF	ID	EX	MEM	WB	
SUB R9,R2,R6		IF	ID	EX	MEM	WB



R2 written here

R2 read here

Unless proper precaution is taken, the SUB instruction can fetch the wrong value of R2.

- Naïve solution by inserting stall cycles:
 - After the SUB instruction is decoded and the control unit determines that there is a data dependency, it can insert stall cycles and re-execute the ID stage again later.
 - 3 clock cycles are wasted.

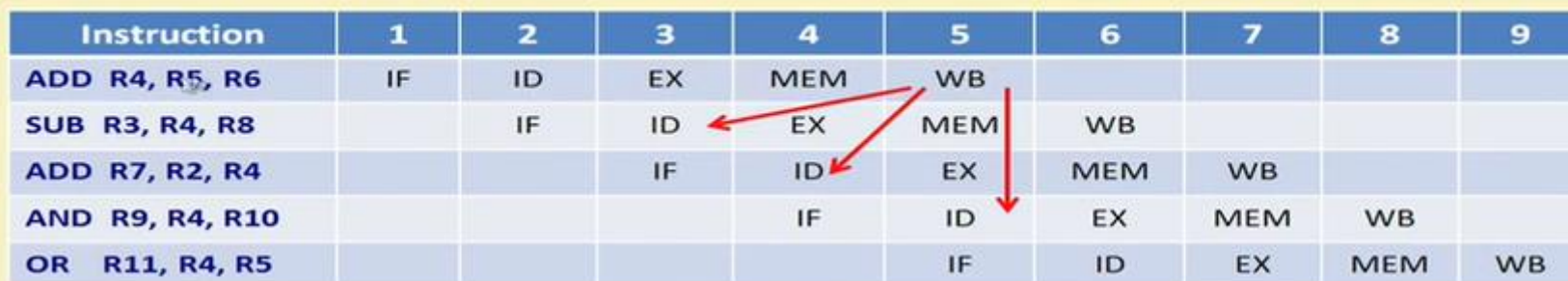
Instruction	1	2	3	4	5	6	7	8
ADD R2,R5,R8	IF	ID	EX	MEM	WB			
SUB R9,R2,R6		IF	ID	STALL	STALL	ID	EX	MEM
Instr. (i+2)			IF	STALL	STALL	IF	ID	EX
Instr. (i+3)				STALL	STALL		IF	ID
Instr. (i+4)				STALL	STALL			IF

- 
- 
- How to reduce the number of stall cycles?
 - We shall explore two methods that can be used together.
 - a) **Data forwarding / bypassing**: By using additional hardware consisting of multiplexers, the data required can be forwarded as soon as they are computed, instead of waiting for the result to be written into the register.
 - b) **Concurrent register access**: By splitting a clock cycle into two halves, register read and write can be carried out in the two halves of a clock cycle (register write in first half, and register read in second half).



Reducing Data Hazards using Bypassing

- Basic idea:
 - The result computed by the previous instruction(s) is stored in some register within the data path (e.g. ALUOut).
 - Take the value directly from the register and forward it to the instruction requiring the result.
- What is required?
 - Additional data transfer paths are to be added in the data path.
 - Requires multiplexers to select these additional paths.
 - The control unit identifies data dependencies and selects the multiplexers in a suitable way.



Instruction	1	2	3	4	5	6	7	8	9
ADD R4, R5, R6	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	EX	MEM	WB			
ADD R7, R2, R4			IF	ID	EX	MEM	WB		
AND R9, R4, R10				IF	ID	EX	MEM	WB	
OR R11, R4, R5					IF	ID	EX	MEM	WB

- The first instruction computes R4, which is required by all the subsequent four instructions.
- The dependencies are depicted by red arrows (result written in WB, operands read in ID).
- The last instruction (OR) is not affected by the data dependency.



Instruction	1	2	3	4	5	6	7	8	9
ADD R4, R5, R6	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	EX	MEM	WB			
ADD R7, R2, R4			IF	ID	EX	MEM	WB		
AND R9, R4, R10				IF	ID	EX	MEM	WB	
OR R11, R4, R5					IF	ID	EX	MEM	WB



Data forwarding requirements:

- The first instruction (ADD) finishes computing the result at the end of EX (shown in **RED**), but is supposed to write into R4 only in WB.
- We need to forward the result directly from the output of the ALU (in EX stage) to the appropriate ALU input registers of the following instructions.
 - To be used in the respective EX stages, shown by the arrow.

- What is the solution?

- As we have seen the hazard cannot be eliminated by forwarding alone.
- Common solution is to use a hardware addition called *pipeline interlock*.
 - The hardware detects the hazard and stalls the pipeline until the hazard is cleared.
- The pipeline with stall is shown.

Instruction	1	2	3	4	5	6	7	8	9
LW R4, 100(R5)	IF	ID	EX	MEM	WB				
SUB R3, R4, R8		IF	ID	STALL	EX	MEM	WB		
ADD R7, R2, R4			IF	STALL	ID	EX	MEM	WB	
AND R9, R4, R10				STALL	IF	ID	EX	MEM	WB

- A common example:

$$A = B + C$$

- Pipelined execution of the corresponding MIPS32 code is shown.

Instruction	1	2	3	4	5	6	7	8	9
LW R1, B	IF	ID	EX	MEM	WB				
LW R2, C		IF	ID	STALL	EX	MEM	WB		
ADD R5, R1, R2			IF	STALL	ID	EX	MEM	WB	
SW R5, A				STALL	IF	ID	EX	MEM	WB

Instruction Scheduling or Pipeline Scheduling:

- Compiler tries to avoid generating code with a LOAD followed by an immediate use.

Example 1

A C code segment:


$x = a - b;$

$y = c + d;$

MIPS32 code:

```
LW    R1, a
LW    R2, b
SUB   R8, R1, R2
SW    R8, x
LW    R1, c
LW    R2, d
ADD   R9, R1, R2
SW    R9, y
```

Two load
interlocks



Scheduled MIPS32 code:

```
LW    R1, a
LW    R2, b
LW    R3, c
SUB   R8, R1, R2
LW    R4, d
SW    R8, x
ADD   R9, R3, R4
SW    R9, y
```

Both load
interlocks
are
eliminated



Original code with data hazards

1. LW R1, 0(R0); Load word...
2. LW R2, 4(R0); Load word...
3. ADD R3,R1,R2;
4. SW R3,R2(R0); Store word...
5. LW R4, 8(R0); Load word...
6. ADD R5, R1, R4
7. SW R5, 16(R0); Store word...

1. LW R1, 0(R0);
2. LW R2, 4(R0);
3. ADD R3, R1, R2;
4. SW R3, R2(R0);
5. LW R4, 8(R0);
6. ADD R5, R1, R4;
7. SW R5, 16(R0);





```
1.  LW R1, 0(R0);
2.  LW R2, 4(R0);
5.  LW R4, 8(R0);
3.  ADD R3, R1, R2;
4.  SW R3, R2(R0);
6.  ADD R5, R1, R4;
7.  SW R5, 16(R0);
```