Hazards in pipeling

https://youtu.be/h0lentwmZLU

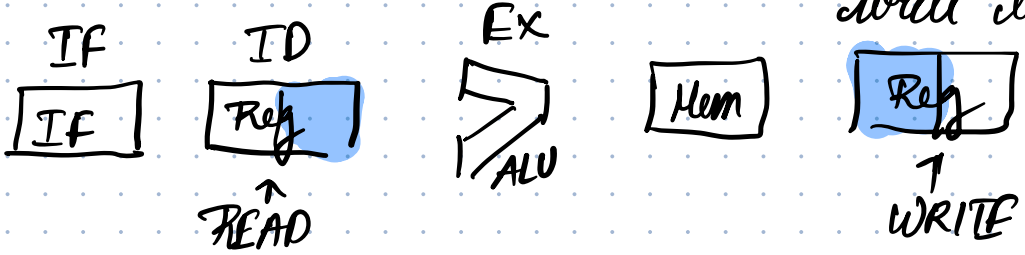
→ Data hazards
→ Structural hazards
→ Control hazard
→ The flow chart problem!

5 steps of RISC data path:

Each can take at most 5 clock cycles.

1) IF – Instruction Fetch
2) ID – Instruction Decode
3) EX – Execution
4) MEM – Memory access cycle
5) WB – Write-back

Mnemonics: Fetch it, then decode it, then execute it, then access the memory to do write back



Hazards:

Circumstances that would cause incorrect execution if next instruction is fetched and executed

→ Structural hazard:

Different instructions, at different stages simultaneously want the same hardware. I_1 & I_2 wants the same hardware.

→ Data hazard:

An instruction requires a data to be computed by a

previous instruction still in pipeline.

I_1 is still in pipeline.

I_2 requires I_1.

→ Control hazards:

An instruction has conditional dependance on a previous instruction. I_2 execution is based on conditional outcome of I_1.

Solution:

1) Structural hazard:

→ Wait / stall

→ Duplicate hardware

2) Data hazard:

Instruction Types and Stages

Let's break down what happens in each stage for typical RISC-V instructions:

- **sub, add, addi, subi:** These are ALU instructions.
  - **IF:** Instruction is fetched.
  - **ID:** Registers are read, immediate values are decoded (for `addi`, `subi`).
  - **EX:** The ALU performs the subtraction/addition.
  - **MEM:** (No memory operation is required.)
  - **WB:** The result is written back to a register.
- **ld (load):** Load from memory.
  - **IF:** Instruction is fetched.
  - **ID:** Registers are read, immediate values for address offset are decoded.
  - **EX:** Address is calculated.
  - **MEM:** Data is loaded from memory.
  - **WB:** Loaded data is written back to a register.
- **sd (store):** Store to memory.
  - **IF:** Instruction is fetched.
  - **ID:** Registers are read, immediate values for address offset are decoded.
  - **EX:** Address is calculated.
  - **MEM:** Data is written to memory.
  - **WB:** (No write-back to register, as the result is stored in memory.)

By this time I_1 will be done →

Important:

→ Every instruction would take almost 5 cycles.

Types of data hazards:

1) Read After Write (RAW) : Data dependence.

I : add (r1), r2, r3

J : sub r4, (r1), r3

J is reading before I writes on it

2) Write After Read (WAR) : Anti-dependence. Because of reuse of reg r1.

I : sub r4 r1, r3] WAR

J : add r1, r2, r3]

K : mul r6, r1, r7] RAW

→ Say due to some issue I is not executed, then J would modify r1 before I could read the old r1.

3) Write After Write (WAW) : Output dependence. Because of reuse of r1.

I : sub r1, r4, r3

J : add r1, r2, r3

K : mul r6, r1, r7

→ If I writes in r1 before J, then K would access the wrong r1.

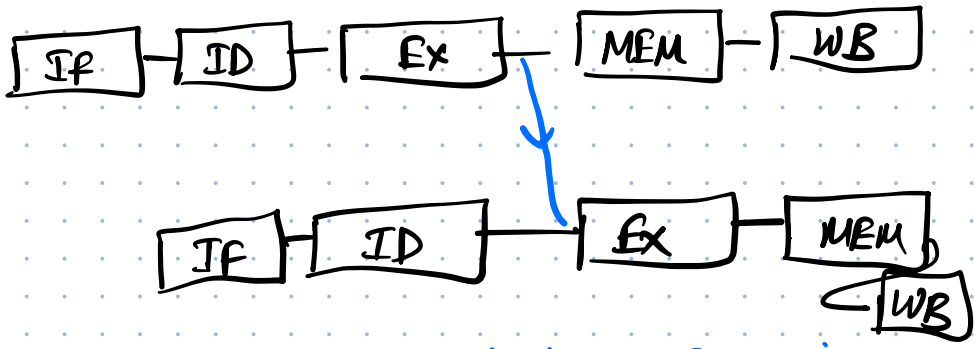
→ WAR & WAW cannot happen in, in-order pipelines, MIPS5.

Handling data hazard: (Just RAW):

1) Data forward

add R1, R2, R3

sub R4, R1, R5



forward from Ex unit directly.

* But the needs should be always after the ALU aka Ex.

2) Use stalls when nothing can be done

3) Re-arranging the instructions without affecting the correctness.

(Eg.) a = b + c
d = e - f

a, b, c, d, e, f are in memory.

			End cycle	
	lw	Rb, b	5	
	lw	Rc, c	6	
RAW	add	Ra, Rb, Rc	8 (stall)	IF → ID → EX → MEM → WB
	sw	a, Ra	9	IF → ID → EX → MEM → WB

```
lw  Re, e          10
lw  Rf, f          11
sub Rd, Re, Rf     13 (stall)
sw  d, Rd          14
```

Now lets use re-arranging,

```
lw  Rb, b          5
lw  Rc, c          6
lw  Re, e          7
add Ra, Rb, Rc     8
lw  Rf, f          9
sw  a, Ra          10
sub Rd, Re, Rf     11
sw  d, Ra          12
```

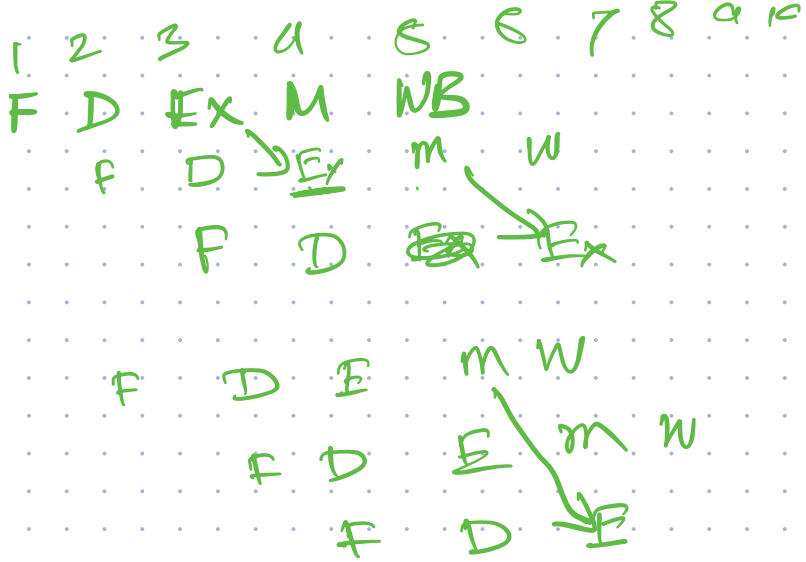
```
add   x2, x3, x4
ld    x3, 6(x2)      RAW  — 1 stall
sub   x3, x3, x4     WAR  — Not possible
sd    x3, 6(x4)      WAW  — Not possible
ld    x7, 7(x3       RAW  — 1 stall
add   x6, x7, x8     RAW  — 1 stall
addi  x8, x7, 45     No hazard
add   x12, x13, x14  No hazard
```

3 stalls.

Rearranged:

```
add   x2, x3, x4
add   x12, x13, x14     No Hazard
ld    x3, 6(x2)         No Hazard
sub   x3, x3, x4        WAR
sd    x3, 6(x4)         WAW
ld    x7, 7(x3          RAW
add   x6, x7, x8        RAW
addi  x8, x7, 45
```

```
add   x2, x3, x4
ld    x3, 6(x2)
sub   x3, x3, x4
sd    x3, 6(x4)
ld    x7, 7(x3
add   x6, x7, x8
addi  x8, x7, 45
add   x12, x13, x14
```



add
addi
sub
subi
→ ALU → Data available at Ex

ld → Data available at MEM

ld → Data required at EX

Instruction Types and Stages

Let's break down what happens in each stage for typical RISC-V instructions:

- **sub, add, addi, subi**: These are ALU instructions.
  - **IF**: Instruction is fetched.
  - **ID**: Registers are read, immediate values are decoded (for `addi`, `subi`).
  - **EX**: The ALU performs the subtraction/addition.
  - **MEM**: (No memory operation is required.)
  - **WB**: The result is written back to a register.
- **ld (load)**: Load from memory.
  - **IF**: Instruction is fetched.
  - **ID**: Registers are read, immediate values for address offset are decoded.
  - **EX**: Address is calculated.
  - **MEM**: Data is loaded from memory.
  - **WB**: Loaded data is written back to a register.
- **sd (store)**: Store to memory.
  - **IF**: Instruction is fetched.
  - **ID**: Registers are read, immediate values for address offset are decoded.
  - **EX**: Address is calculated.
  - **MEM**: Data is written to memory.
  - **WB**: (No write-back to register, as the result is stored in memory.)