

Pipeline e Hazard

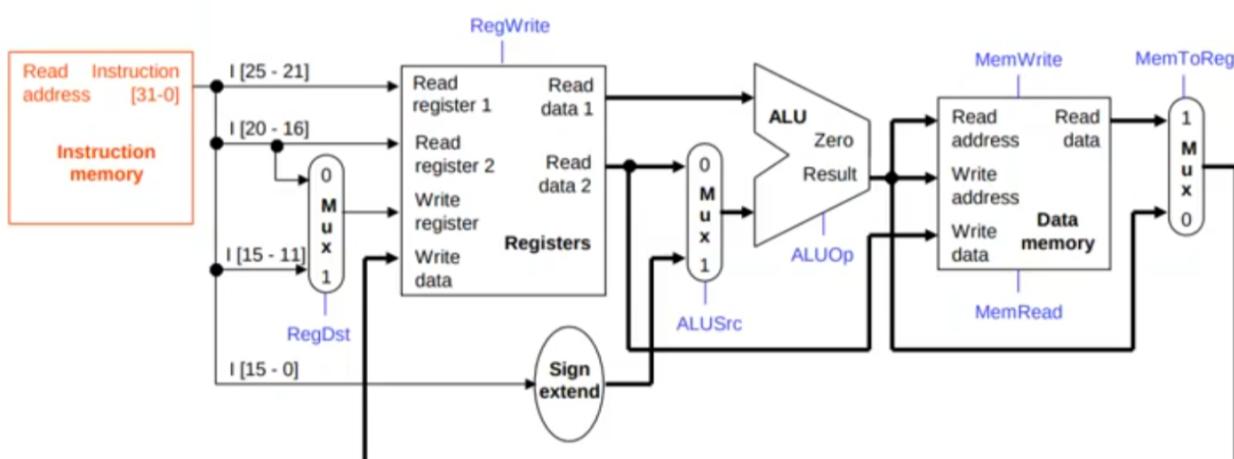
Introduzione

Pipeline: metodo per accelerare le computazioni del datapath
a singolo ciclo

Hazard: Problemi che nascono a causa della pipeline

Datapath Singolo Ciclo

Utilizza sempre un ciclo di clock (piuttosto lungo) per tutte le istruzioni. Nello specifico, usa il ciclo della lw, l'istruzione + lenta.



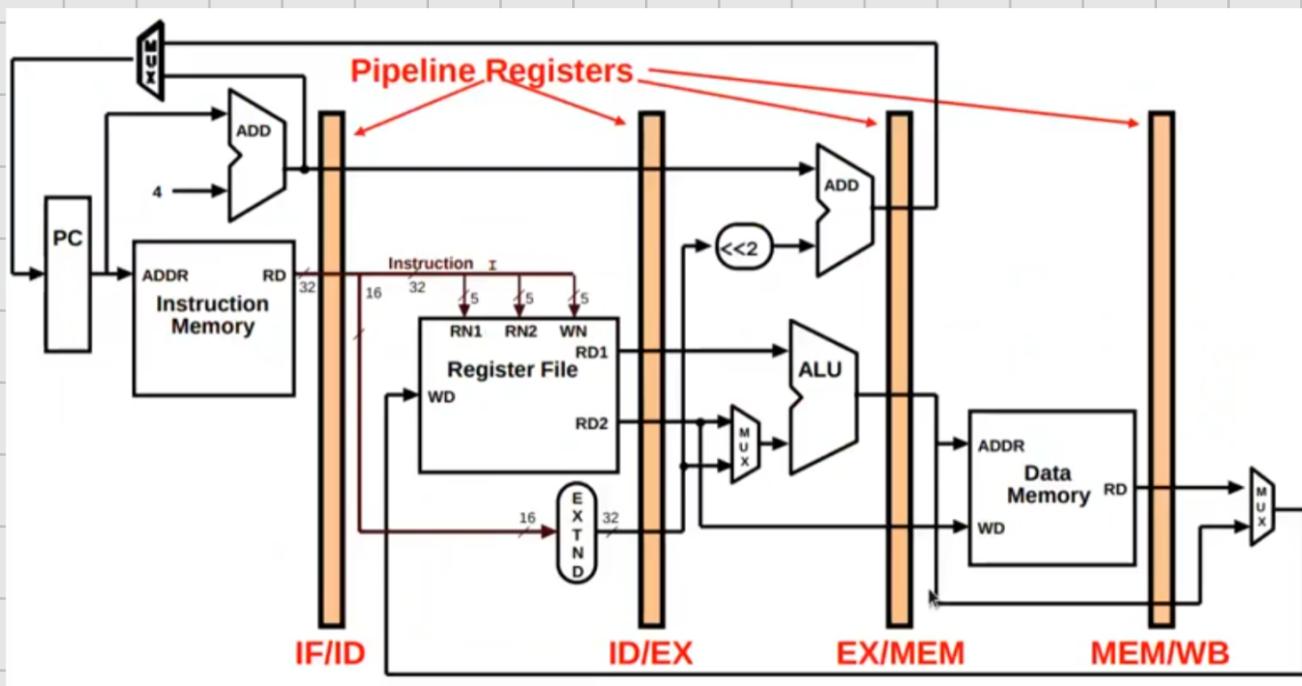
5 step istruzioni Mips

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

Analisi istruzioni

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

Datapath con registri pipeline



Registri pipeline permettono di "separare" il nostro ciclo di clock di tenere separate le varie sezioni del datapath.

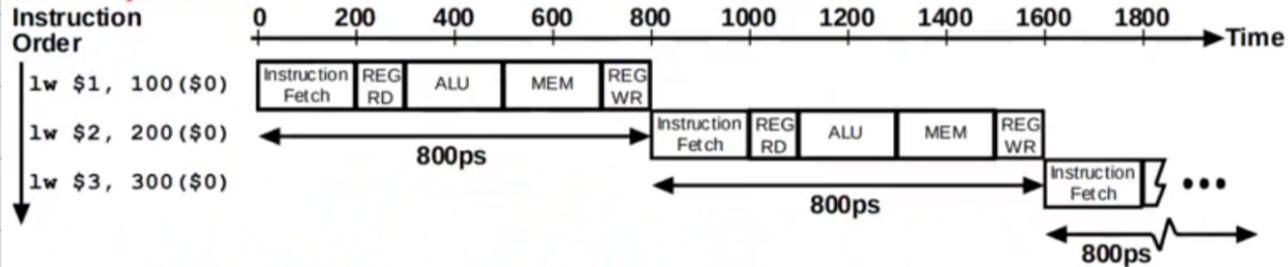
Nell'originale il segnale era per lo più combinatorio.

Si può passare oltre alla "barriera" solo quando il valore viene aggiornato.

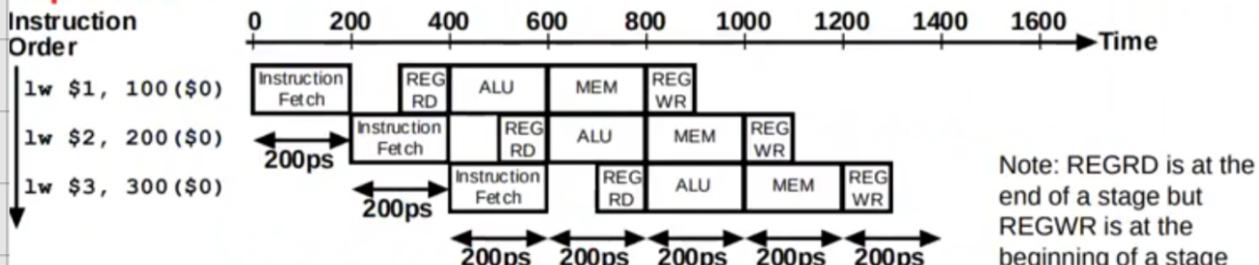
Così facendo, possiamo utilizzare più istruzioni nello stesso momento, in quanto lavoreranno in zone del datapath diverse.

Esecuzione Singolo Ciclo vs Pipeline

Non-Pipelined



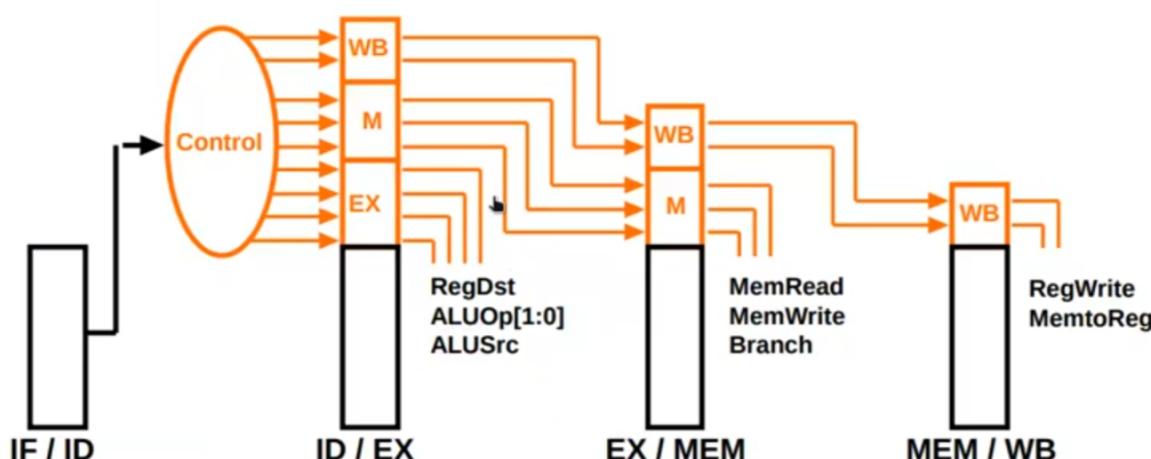
Pipelined



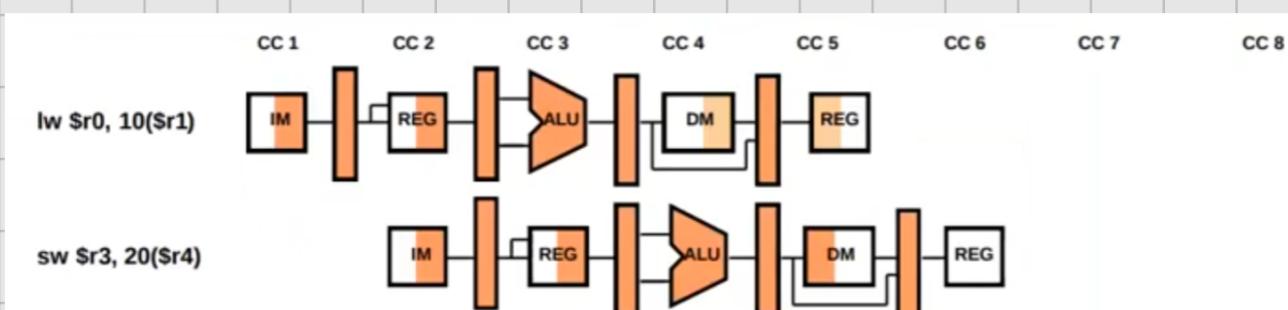
Come possiamo notare, abbiamo un notevole aumento di efficienza!

- 2 istruzioni singolo ciclo: ~600
- 2 istruzioni pipeline: ~300

Passaggio Controlli Pipeline



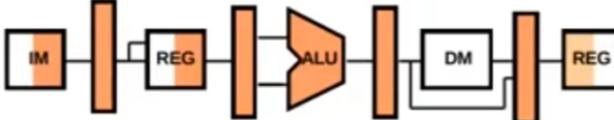
Analisi di datapath con pipeline che esegue 4 istruzioni:



add \$r5, \$r6, \$r7



sub \$r8, \$r9, \$r10



Legenda:

- colorato nella 1^a metà: utilizzo in scrittura "store"
- colorato nella 2^a metà: utilizzo in lettura "load"
- non colorato: non utilizzato dall'istruzione

Osservando questo schema, notiamo che ci possono essere degli errori, ovvero gli hazards.

Hazards

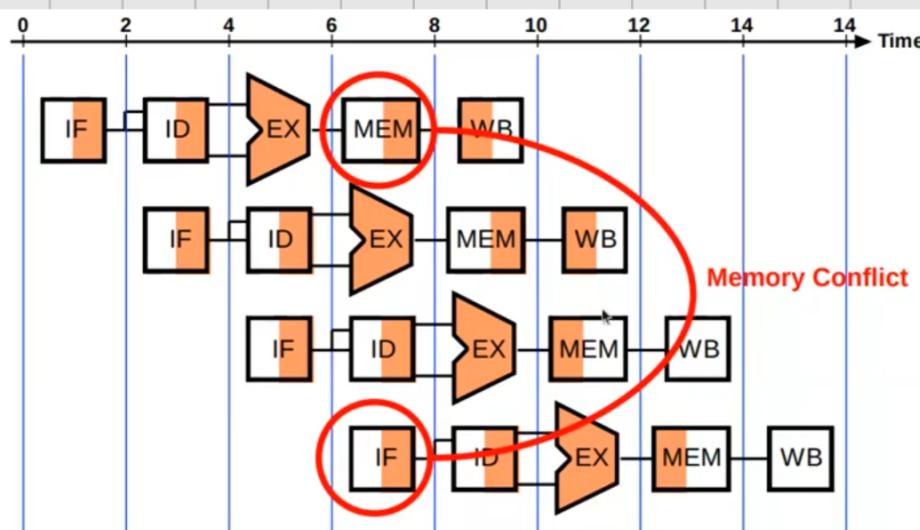
hazard = rischio di malfunzionamento, imprevisto. Ci sono 3 tipi:

- Structural Hazards: un componente è usato due volte nello stesso ciclo
- Control Hazards: quando abbiamo branch, avere le istruzioni una dopo l'altra porta ai problemi.

→ più frequenti
• Data Hazards: quando accediamo ad un dato prima che sia pronto / aggiornato.

Possiamo **sempre** risolvere hazards aspettando: "stall". Tuttavia si perde efficienza.

Structural Hazard:



Questo tipo di hazard
è già risolto nel datapath a
singolo ciclo!

Questo perché abbiamo due memorie
separate.

Essendo una questione già risolta dalla datapath, non ci dovrebbero essere domande.

• Control Hazards

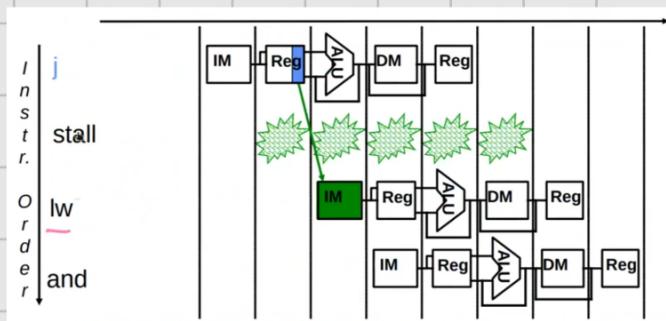
Avvengono quando il flusso di istruzioni può alterarsi (branch, j)

Possiamo prendere in considerazione diversi metodi risolutivi:

- stalli
- prendere decisione prima nella pipeline
- predire
- compilatore che in assembly sposta le istruzioni

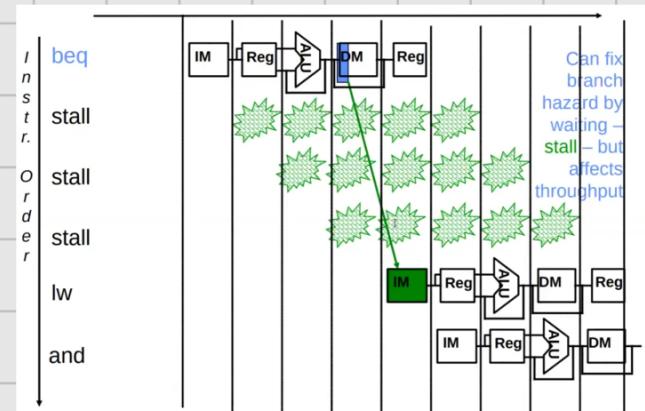
• Stalli in Control Hazard:

Jump



Jump ha l'indirizzo nell'istruzione stessa
quindi l'aggiornamento è + veloce.

branch



Dobbiamo fare il confronto nella execute
e aggiornare PC se la cond. è soddisfatta.

• Branch Prediction

Nelle branch possiamo predire il risultato della condizione, considerando come se il controllo fallisca sempre, e quindi non dobbiamo aggiornare il PC e quindi continuare il flusso di str.

Se il sottoprogramma fatto però, il PC andava incrementata e ciò che avevamo eseguito prima, devono essere "sciaccuate" e riportare i valori intatti prima della branch.

Esercizio Descrittivo Control Hazard

```

.data 0x10008000
array: .word 10, 21, 32, 23, 40, 12, 86, 72, 23, 18, 35, 68, 27, 46, 55

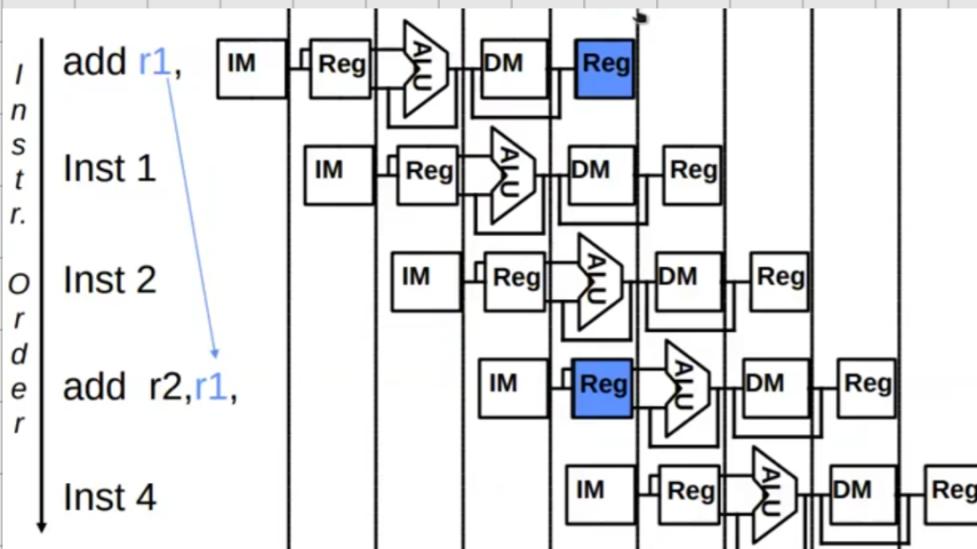
.text
main:
li $v0,0          # init risultato
lui $t0,0x1000    # base vettore, ms half-word
ori $t0,$t0,0x8000 # base vettore, ls half-word
li $t1,2          # init primo indice
li $t2,12         # init ultimo indice
loop:
beq $t1,$t2,endloop 3° # test di uscita, esci se primo indice = ultimo indice
sll $t3,$t1,2      # indice => offset (una word contiene 4 byte)
add $t4,$t0,$t3    # base + offset
! lw $t5,0($t4)     # accesso al valore dell'array in memoria
add $v0,$v0,$t5 DH! # calcolo della somma
addi $t1,$t1,1      1° # incremento indice
j loop             2°
endloop:
jr $ra → jal $ra! # fine esecuzione

```

La sequenza che conta è quella temporale, non sequenziale.

Bisogna controllare i registri prima e dopo i salti, verificare se i dati sono aggiornati. Verificare se nello stesso ciclo ci possono essere Data Hazard. (in rosso evidenzio un data hazard)

Data Hazards:



Come possiamo osservare, nella 4^a istr. abbiamo bisogno di R3, il quale perciò deve essere aggiornato, in quanto R3 è il registro di arrivo della 3^a istruzione.

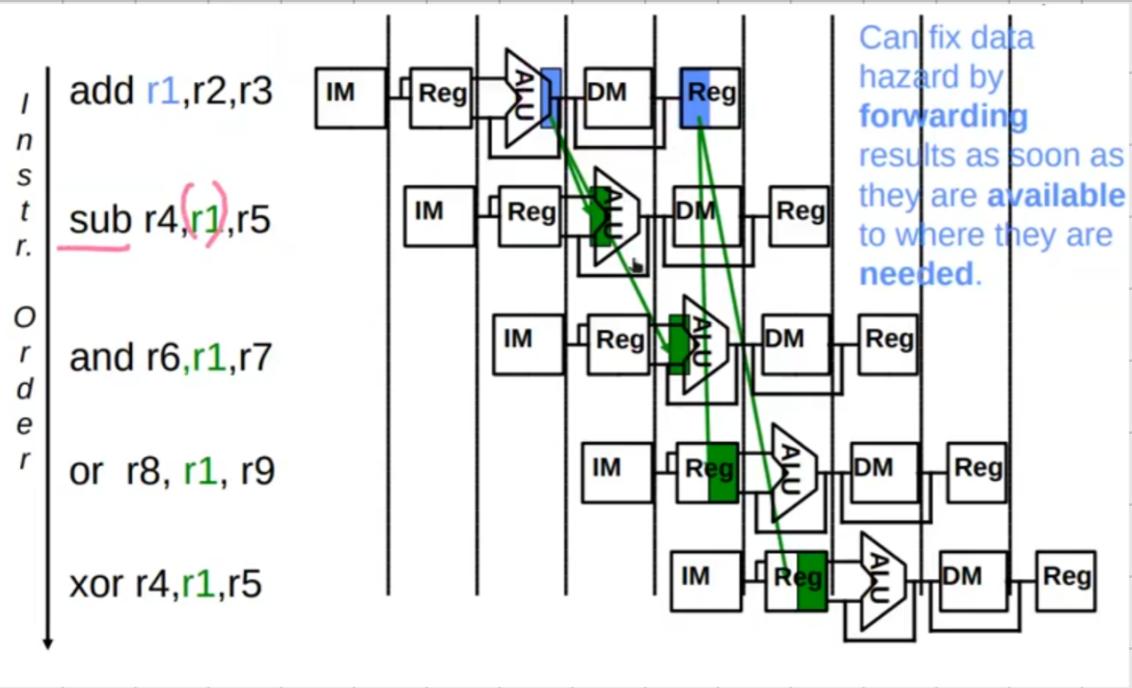
Risoluzione Data Hazard

<p>n° istruzioni tra le due = 2 → divido il clock in due parti, nel primo aggiorno e nel secondo prelevo.</p> <p>n° istruzioni tra le due < 2 → creo fasidi stall + utilizzo metodo</p> <p>n° istruzioni tra le due > 2 → non ho data hazard</p>	<p>→</p>
--	----------

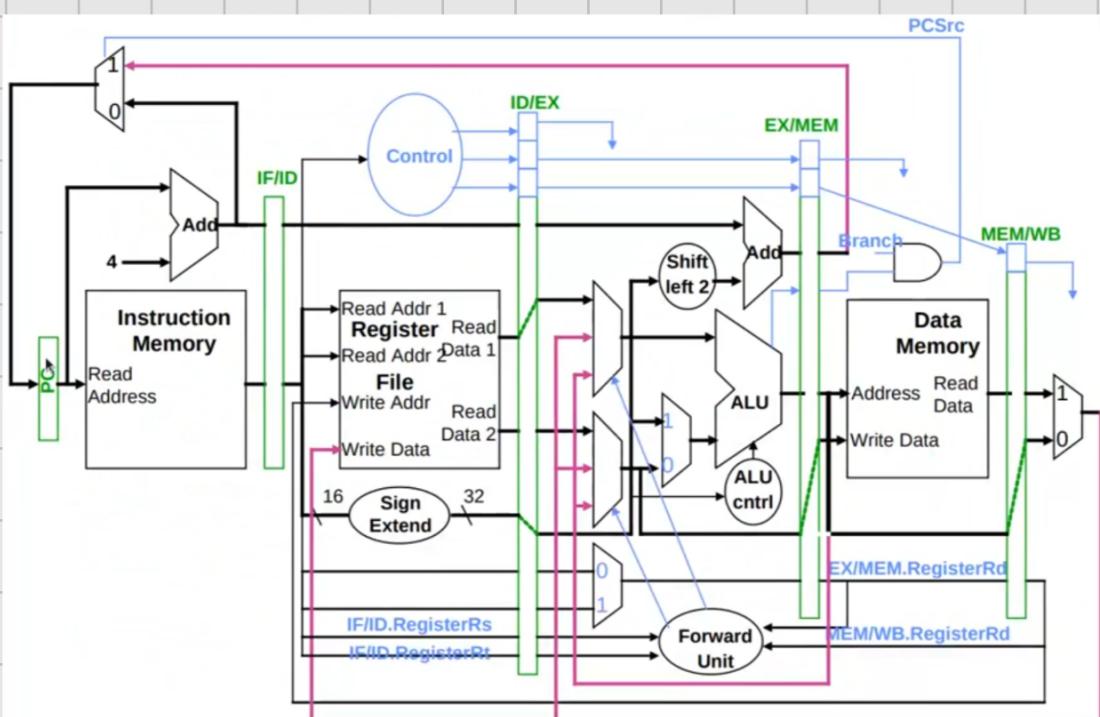
Un altro miglioramento per evitare data hazard, è quello di **forwarding**, cioè creare un collegamento

all'uscita dell'ALU che contiene il valore, il quale poi rientrerà nella stessa ALU, in modo tale che le istruzioni successive abbiano già il valore pronto, anche se effettivamente non è ancora stato salvato il valore in memoria (scc, write-back)

Ricordiamo però che, al livello hardware, bisogna avere qualche (mux + unità di segnali) che controlla se è necessario il valore aggiornato o meno.



Aggiornamento Datapath (teoricamente a solo scopo illustrativo)



Potremmo comunque necessitare di un ciclo di stall anche con forward:

IW r1,100(r2)

(flush)

sub r4,r1,r5

and r6,r1,r7

or r8, r1, r9

xor r4,r1,r5

