

Architettura degli Elaboratori

L'architettura della CPU – Control Unit e nuove istruzioni



SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Checco

checco@di.uniroma1.it

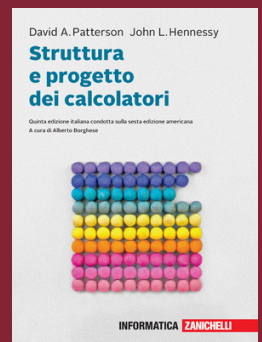
Special thanks and credits:

Andrea Sterbini, Iacopo Masi,

Claudio di Ciccio

[S&PdC]

4.1-4.4







Riepilogo

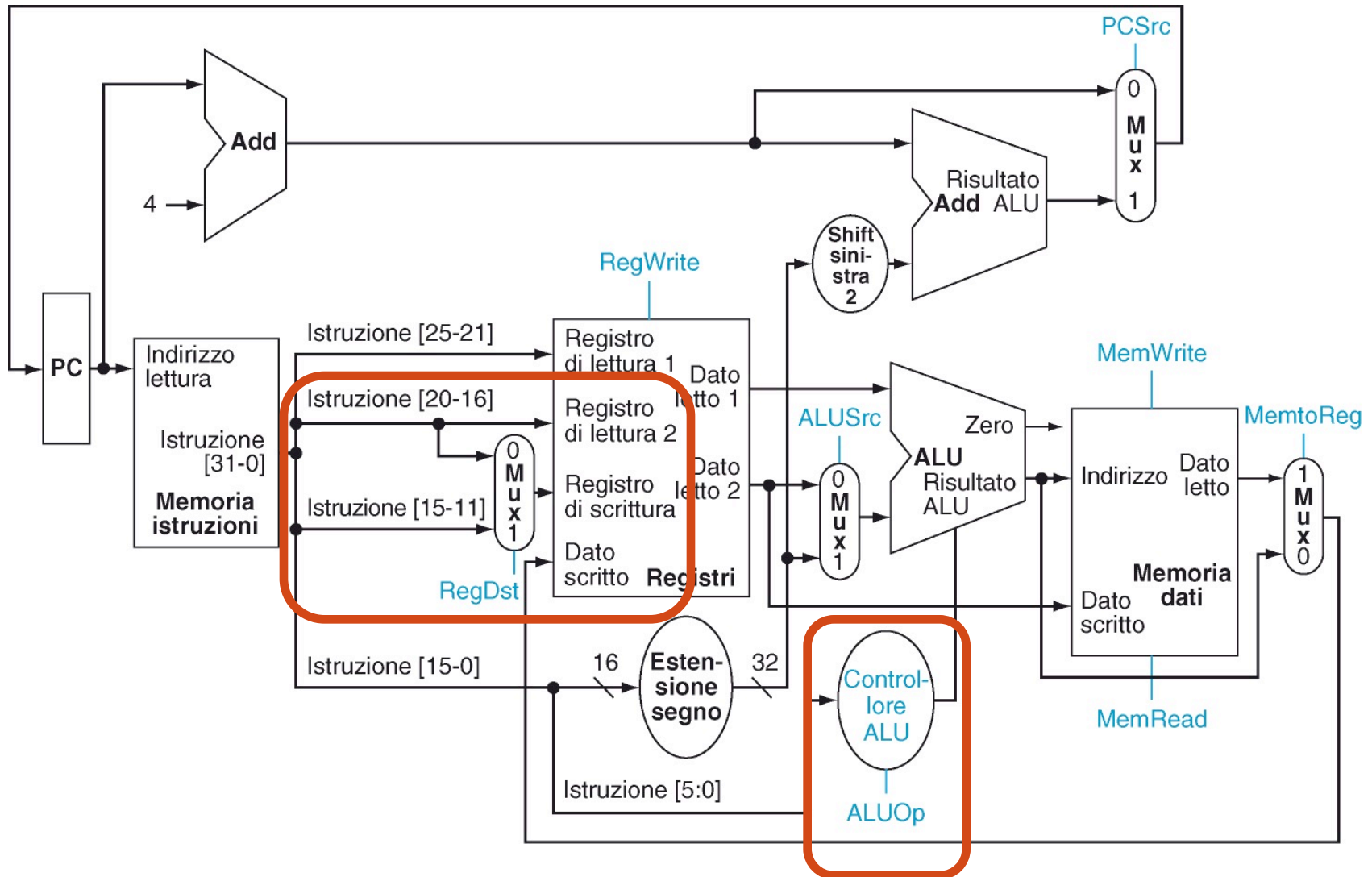


SAPIENZA
UNIVERSITÀ DI ROMA

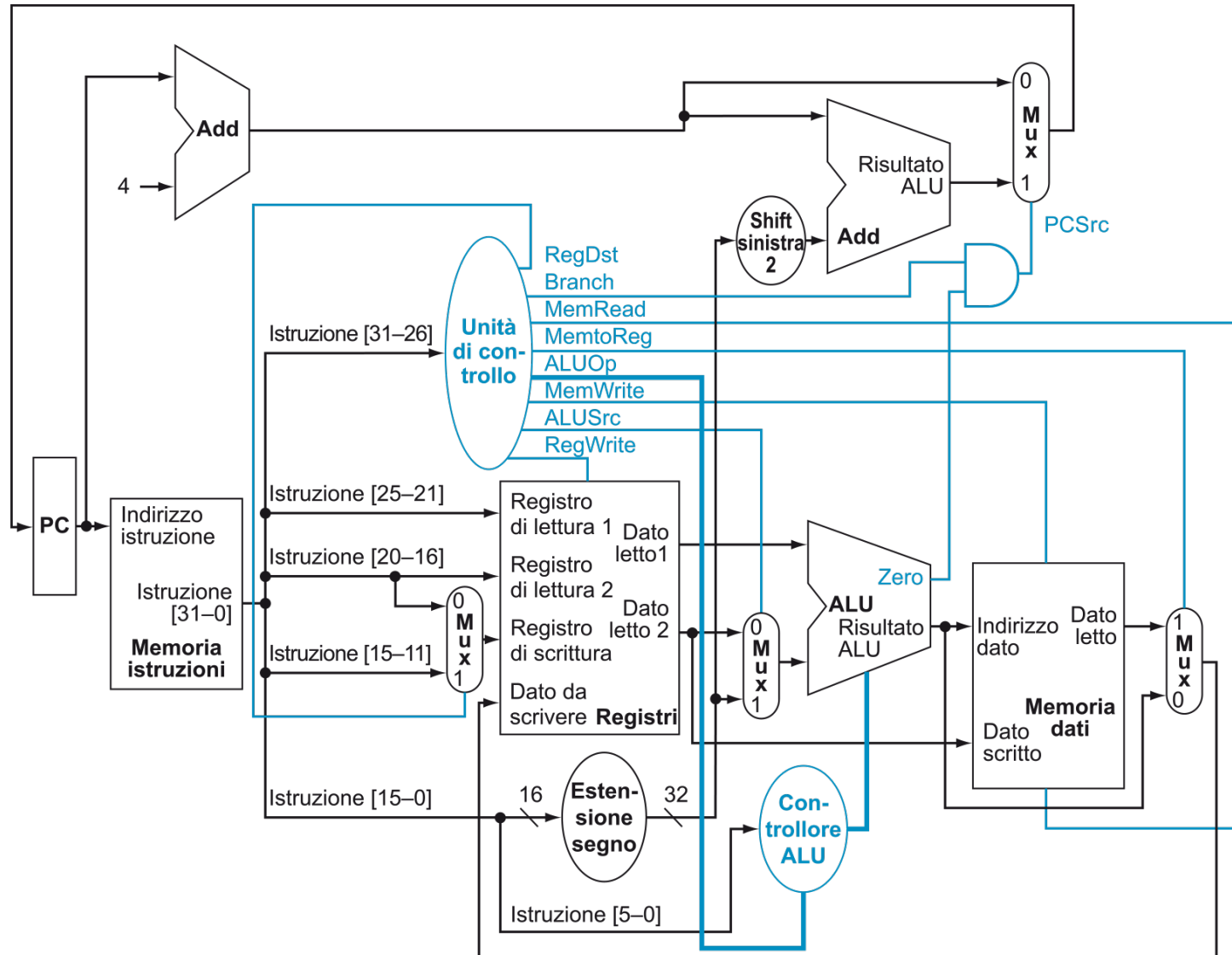
Input di controllo e tabella di verità

		ALUOp		Campo funz						Operazione
		ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0	
lw, sw	—	0	0	X	X	X	X	X	X	0010 
beq	—	X	1	X	X	X	X	X	X	0110 
add	—	1	X	X	X	0	0	0	0	0010 
sub	—	1	X	X	X	0	0	1	0	0110 
and	—	1	X	X	X	0	1	0	0	0000
or	—	1	X	X	X	0	1	0	1	0001
slt	—	1	X	X	X	1	0	1	0	0111

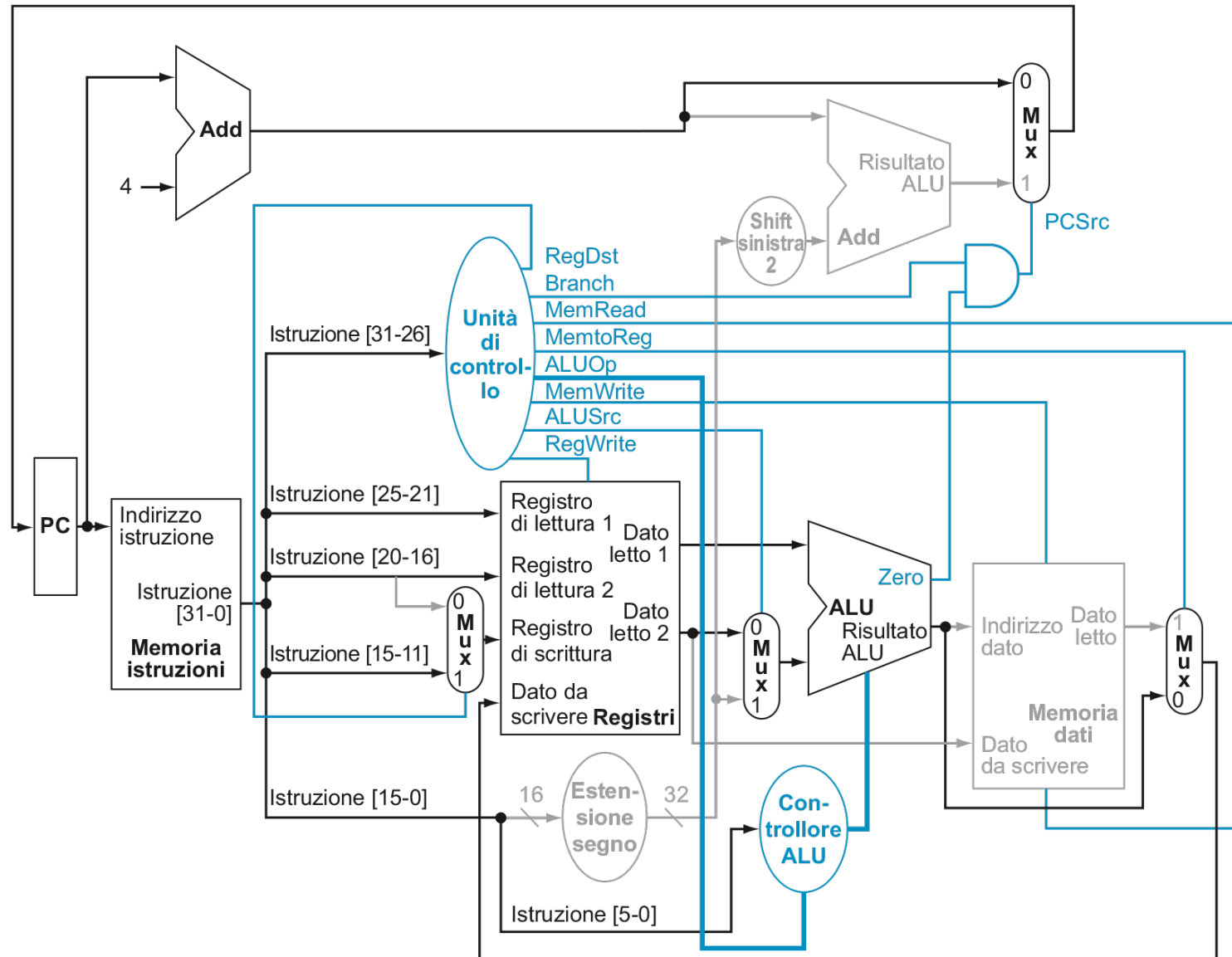
Datapath completo (senza CU)



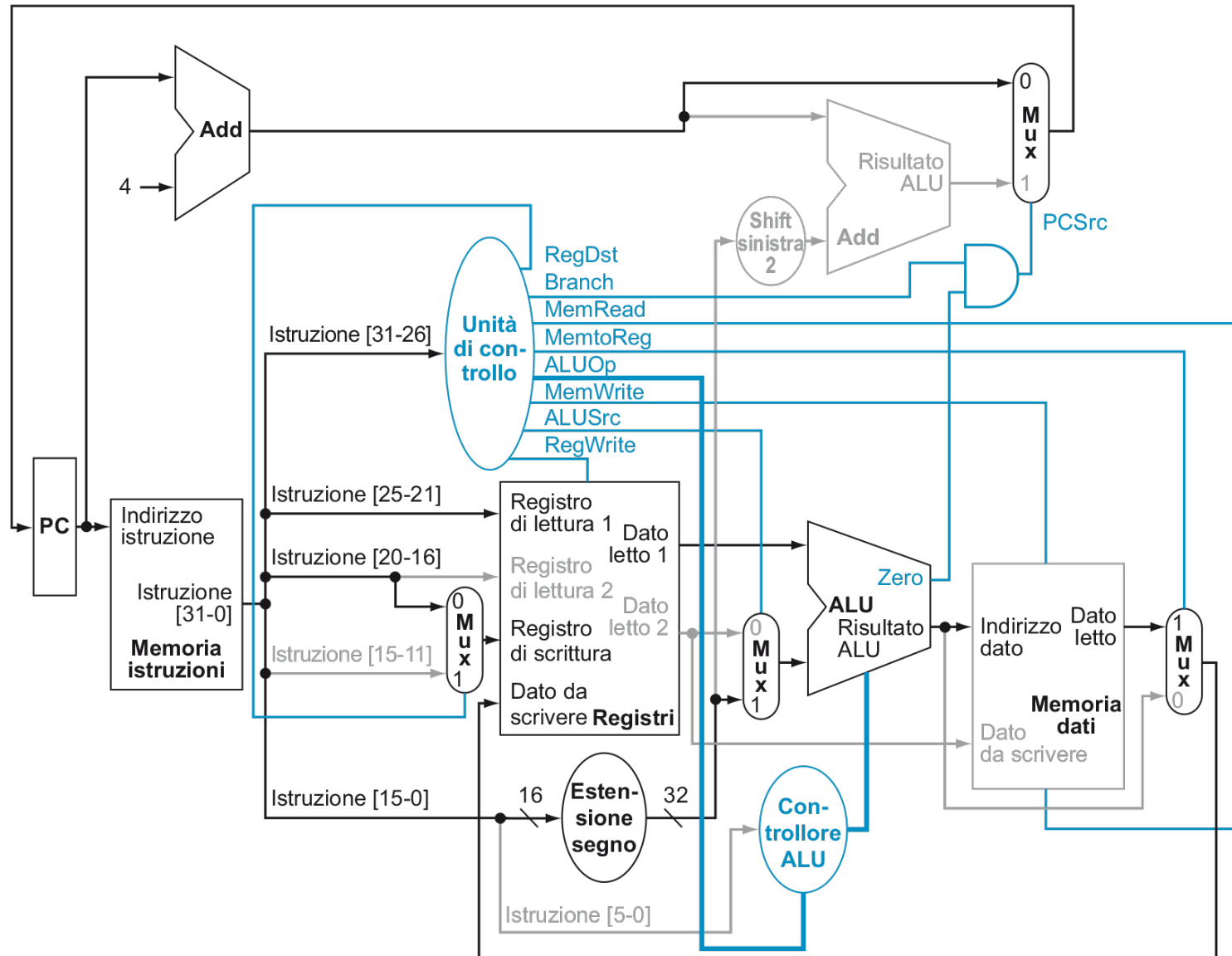
Datapath completo



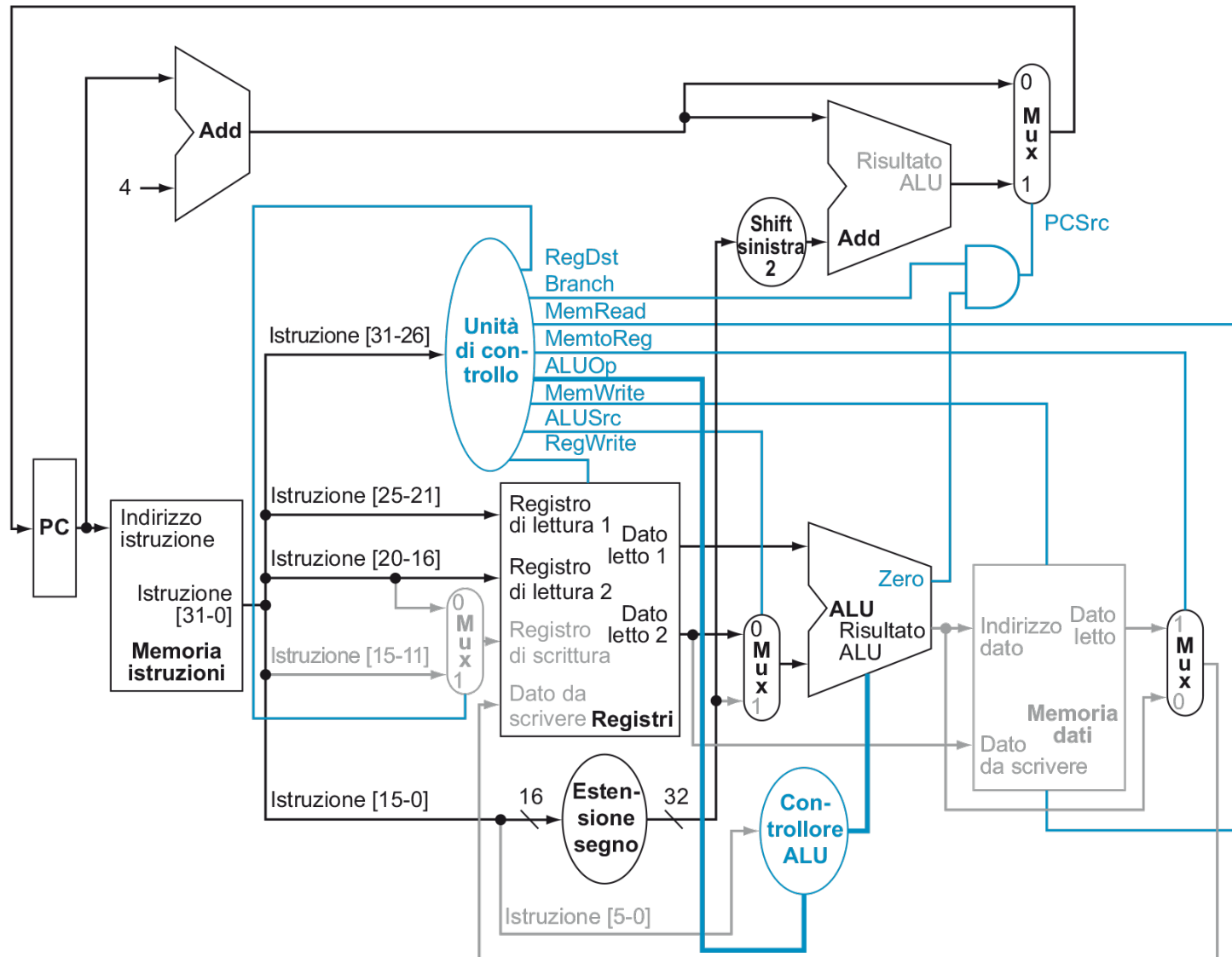
Esecuzione di un'istruzione di tipo R



Esecuzione di lw



Esecuzione di beq



Segnali di controllo

Nome del segnale	Effetto quando non asserito	Effetto quando asserito
RegDst	Il numero del registro di scrittura proviene dal campo rt (bit 20-16)	Il numero del registro di scrittura proviene dal campo rd (bit 15-11)
RegWrite	Nulla	Il dato viene scritto nel register file nel registro individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 16 bit meno significativi dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola il valore di $PC + 4$	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nulla	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea «dato letto»
MemWrite	Nulla	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea «dato scritto»
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati

I segnali da generare

L'ALU deve seguire 4 tipi di comportamento:

- Se l'istruzione è di **tipo R** eseguire l'operazione indicata dal campo **funct** dell'istruzione
- Se l'istruzione accede alla memoria (**lw, sw**) svolgere la **somma** che calcola l'indirizzo
- Se l'istruzione è un **beq** deve svolgere una **differenza**

Per codificare 3 comportamenti bastano 2 segnali dalla CU: **ALUOp1** ed **ALUOp0**

Quindi i segnali che la CU deve produrre per i diversi tipi di istruzione devono essere:

Istruzione	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
Tipo R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

X indica 'don't care' (termini indifferenti)

Esercizio per casa



Istruzione	Codice decimale	Codice binario
di tipo R	0	000000
lw	35	100011
sw	43	101011
beq	4	000100

Progettare la tabella di verità della **CU** (solo per le righe necessarie) avente

- in ingresso: i 6 bit del codice operativo dell'istruzione
- in uscita: i 9 bit dei segnali di controllo da produrre:

(RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp1, ALUOp0)

Soluzione

Se i codici dei 4 tipi di istruzioni sono:

istruzione	codice decimale	codice binario
di tipo R	0	000000
lw	35	100011
sw	43	101011
beq	4	000100

... e dobbiamo produrre i segnali dell'unità di controllo

Istruzione	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
000000	1	0	0	1	0 (X)	0	0	1	0
100011	0	1	1	1	1	0	0	0	0
101011	X	1	X	0	0 (X)	1	0	0	0
000100	X	0	X	0	0 (X)	0	1	0	1

Da cui possiamo produrre la PLA oppure le funzioni booleane necessarie

Per esempio **ALUSrc = Opcode0 Branch e ALUOp0 = Opcode2 MemWrite = Opcode3**

Aggiungere una nuova istruzione

Supponiamo di voler aggiungere la nuova istruzione, **J** (Jump), dobbiamo:

- Definire la sua **codifica**
- Definire **cosa faccia**
- Individuare le **unità funzionali necessarie** (e se sono già presenti)
- Individuare i **flussi delle informazioni** necessarie
- Individuare i **segnali di controllo** necessari
- Calcolare il **tempo necessario** per la nuova istruzione e se modifica il tempo totale



Supponiamo che abbia la **codifica** seguente (formato J)

Campo	000010	indirizzo
Posizione dei bit	31-26	25:0

... e che il campo da 26 bit in essa contenuto sia l'**istruzione di destinazione** del salto:

- è un **indirizzo assoluto** (invece che uno relativo al PC come per i branch)
- indica l'**istruzione di destinazione** (va moltiplicato per 4 perché le istr. sono «allineate»)
- i 4 bit «mancanti» verranno presi dal PC+4 (ovvero si rimane nello stesso blocco di 256M)
- (per i salti tra blocchi diversi sarà necessario introdurre l'istruzione **jr**)

Aggiungere il Jump

Cosa fa:	 $PC \leftarrow (\text{shift left di 2 bit di Istruzione}[25-0]) \text{ OR } (PC+4)[31-28]$
Unità funzionali:	$PC + 4$ (già presente) shift left di 2 bit con input a 26 bit (da aggiungere) OR dei 28 bit ottenuti con i 4 del PC+4 (si ottiene dalle connessioni) MUX per selezionare il nuovo PC (da aggiungere)
Flusso dei dati:	$Istruzione[25-0] \rightarrow SL2 \rightarrow (\text{OR con i 4 MSBs di } PC+4) \rightarrow MUX \rightarrow PC$
Segnali di controllo:	Jump asserito per selezionare la nuova destinazione sul MUX  $RegWrite=0$ e $MemWrite=0$ per evitare modifiche a registri e MEM
Tempo necessario:	Fetch e in parallelo il tempo dell'adder che calcola $PC+4$ (quindi il massimo tra i due tempi)



NOTA: l'hardware necessario al calcolo della destinazione del salto è sempre presente e calcola la destinazione anche se l'istruzione non è un Jump. Solo se la CU riconosce che è un Jump il valore calcolato viene immesso nel PC per passare (al colpo di clock successivo) alla destinazione del salto.

Obiettivo: una CPU più veloce possibile a costo di usare più componenti hardware

Chiarimento su shift logico e numero di bit

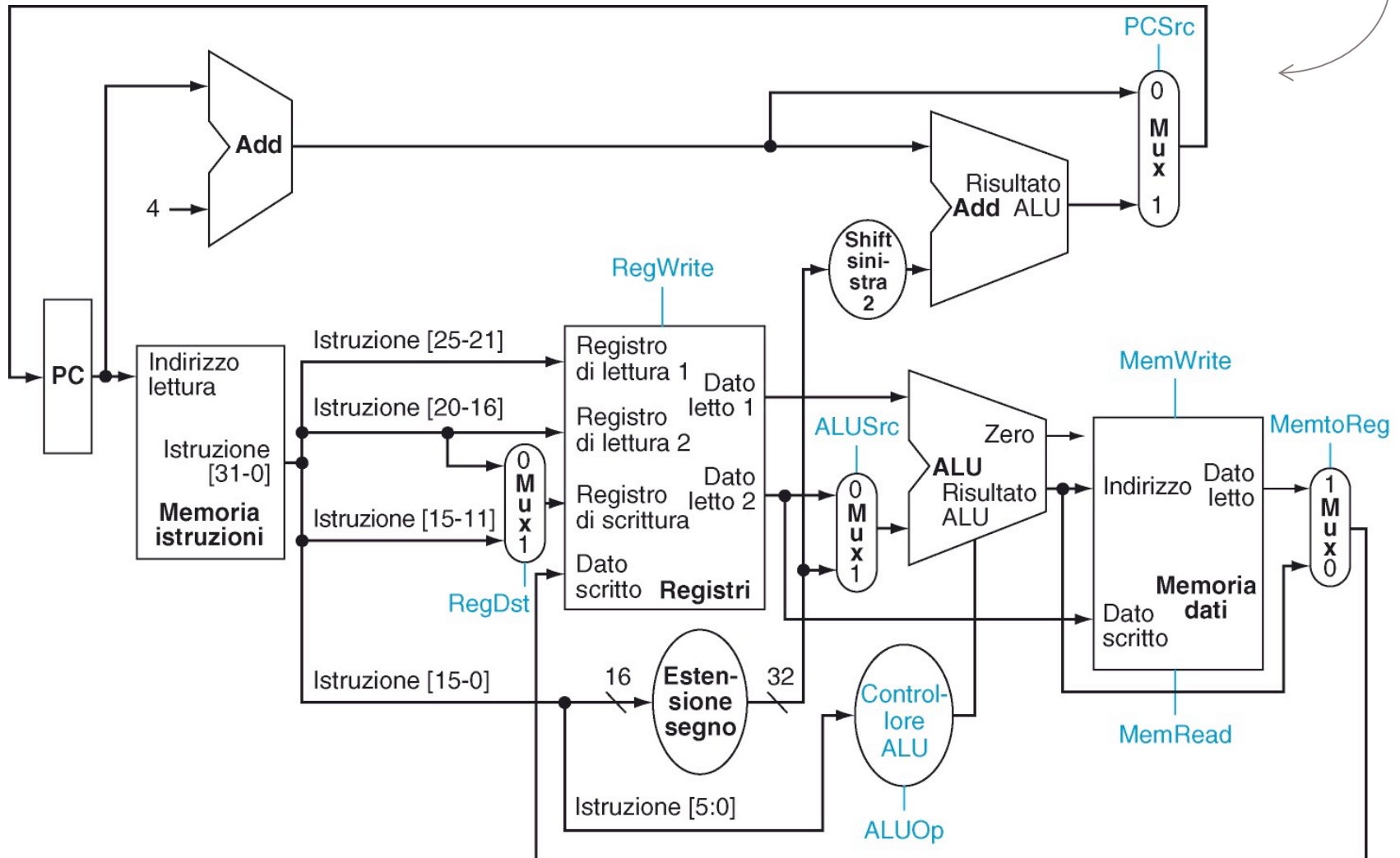
- lo shift logico (costante e.g. 2) si può fare usando solo connessioni e la terra (per gli underflow e overflow)
- Possiamo realizzare componenti anche con un sottoinsieme dei 32bit (anche se in alcuni casi si lavora con cavi di portata fissa, e alcuni possono essere messi a terra o isolati)

Aggiungere il Jump

Cosa fa:

$PC \leftarrow (\text{shift left di 2 bit di Istruzione}[25-0]) \text{ OR } (PC+4)[31-28]$

Qui:
scarabocchi

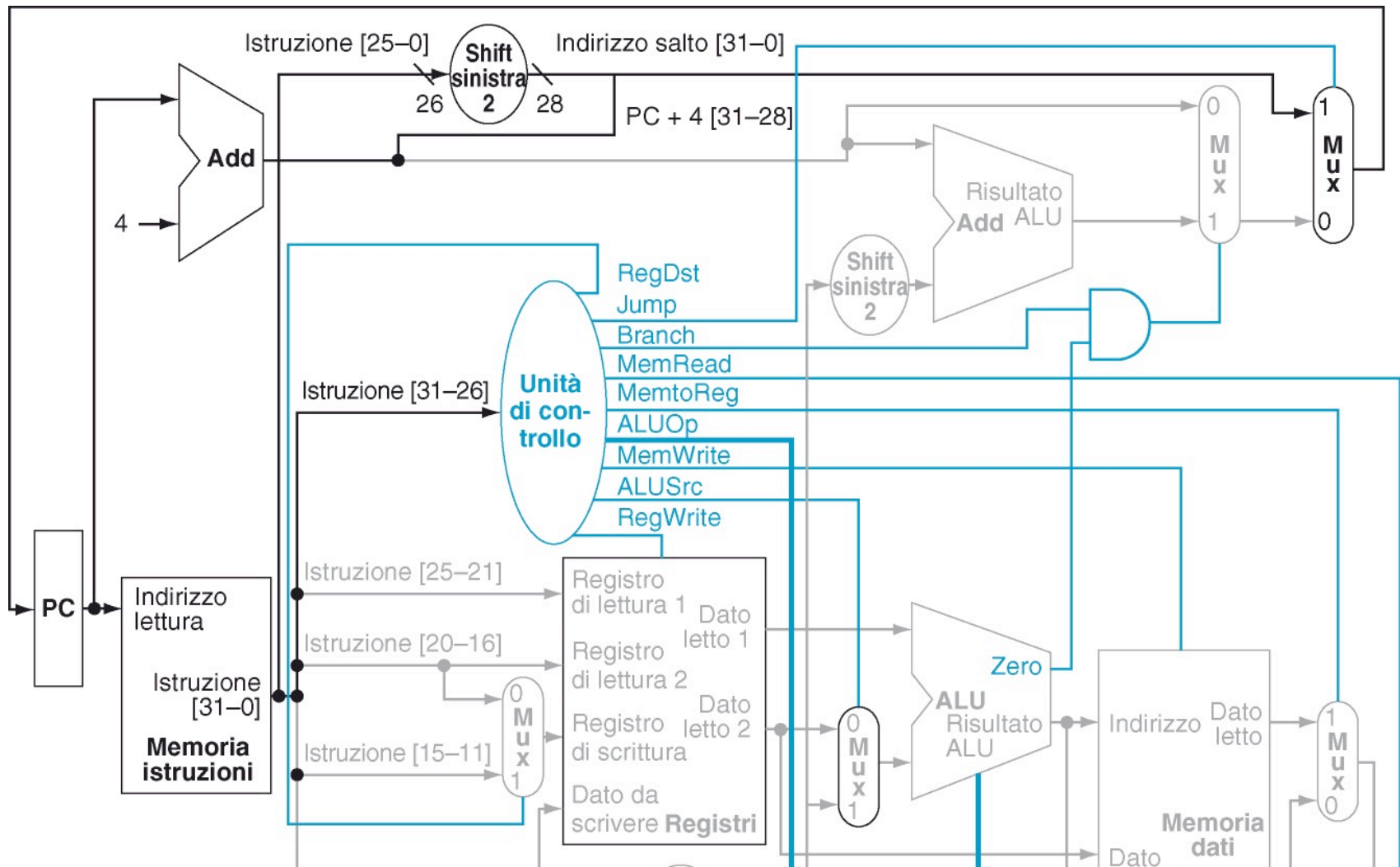


Modifiche per il Jump

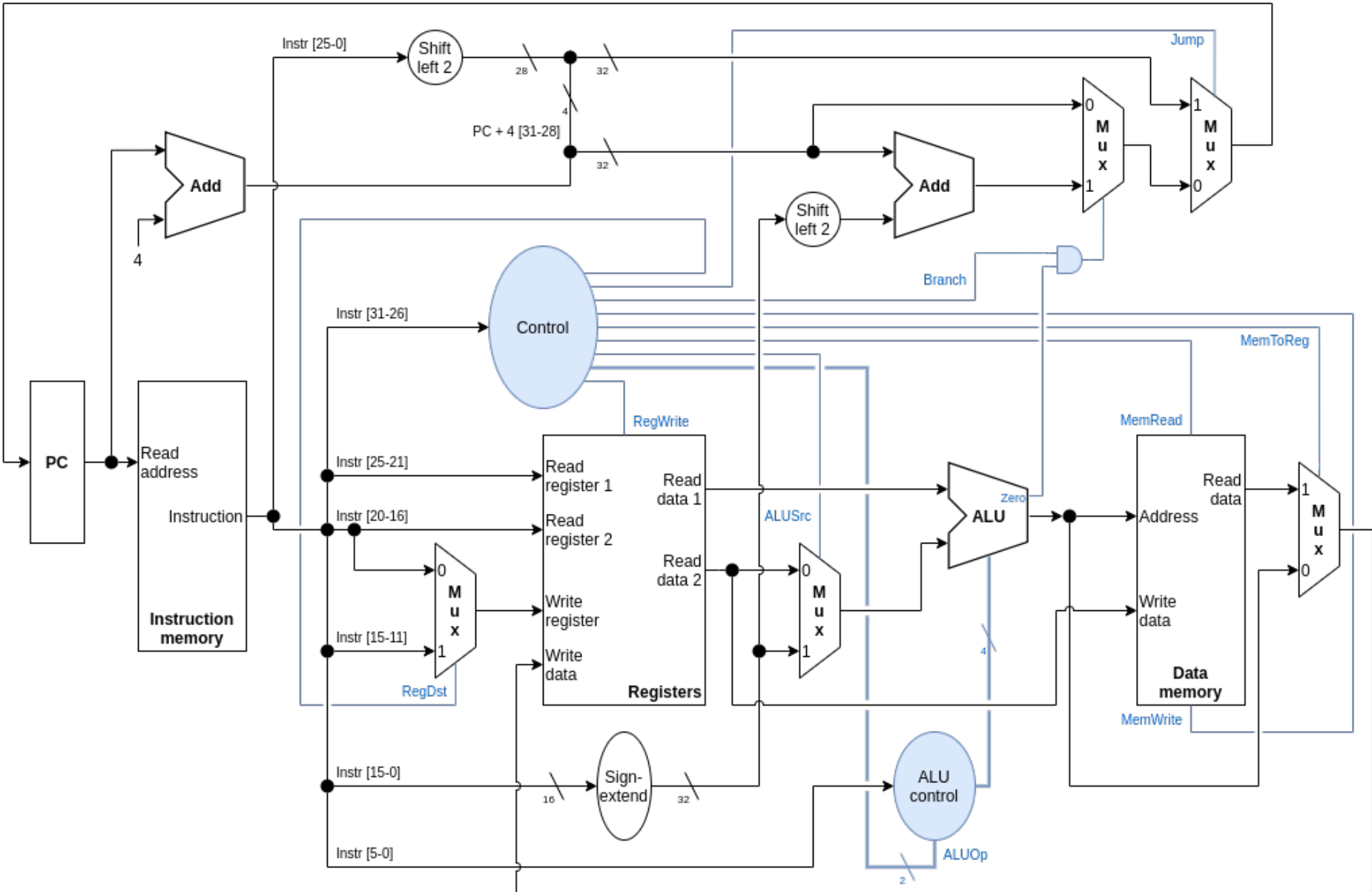
Cosa fa:

$PC \leftarrow (\text{shift left di 2 bit di Istruzione}[25-0]) \text{ OR } (PC+4)[31-28]$

Oh,
meglio



Architettura MIPS per add, beq, sw, lw, j



JAL (Jump and Link)

Come è codificata: **tipo J** come Jump

Cosa fa: $PC \leftarrow (\text{shift left di 2 bit di Istruzione}[25-0]) \text{ OR } (PC+4)[31-28]$ come Jump

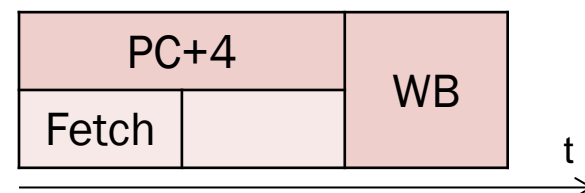
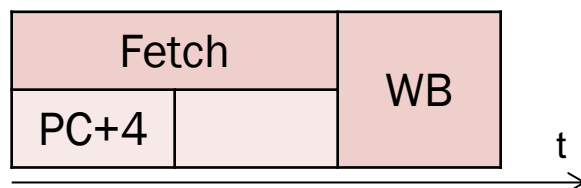
$\$ra \leftarrow PC+4$

Unità funzionali: le stesse del Jump
più **MUX** per selezionare il valore di PC+4 come valore di destinazione
più **MUX** per selezionare il numero del registro **\$ra** come destinazione

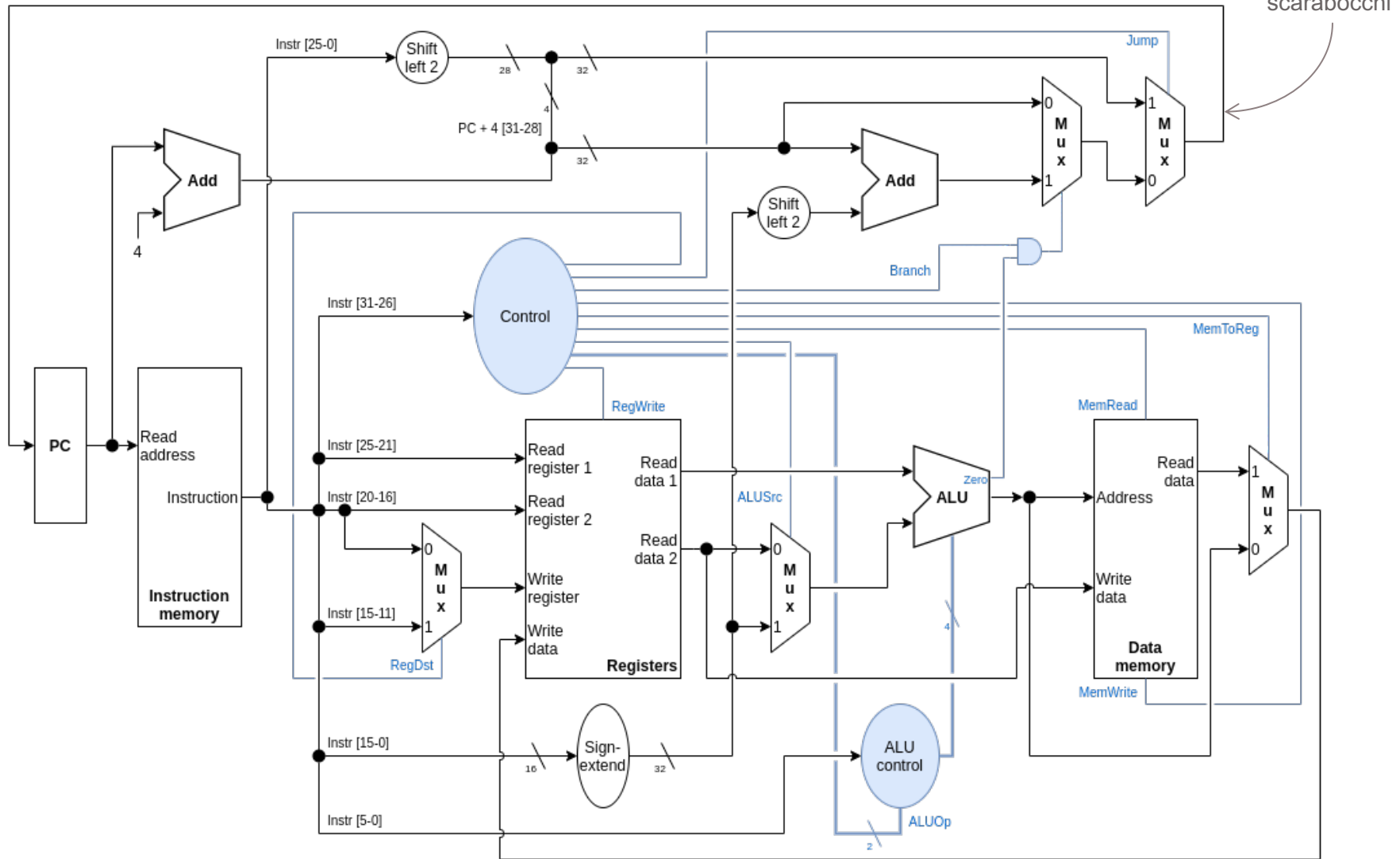
Flusso dei dati: lo stesso del Jump, inoltre
 $PC+4 \rightarrow \text{MUX} \rightarrow \text{Registri (dato da memorizzare)}$
 $31 \rightarrow \text{MUX} \rightarrow \text{Registri (registro destinazione)}$

Segnali di controllo: Il segnale **Jump** deve essere asserito
la **CU** deve produrre un segnale **Link** per attivare i due nuovi MUX

Tempo necessario: il **WB** deve avvenire dopo che sono finiti sia il Fetch (per leggere l'istruzione) sia il calcolo di PC+4 (che va memorizzato in \$ra) per cui possono presentarsi due casi

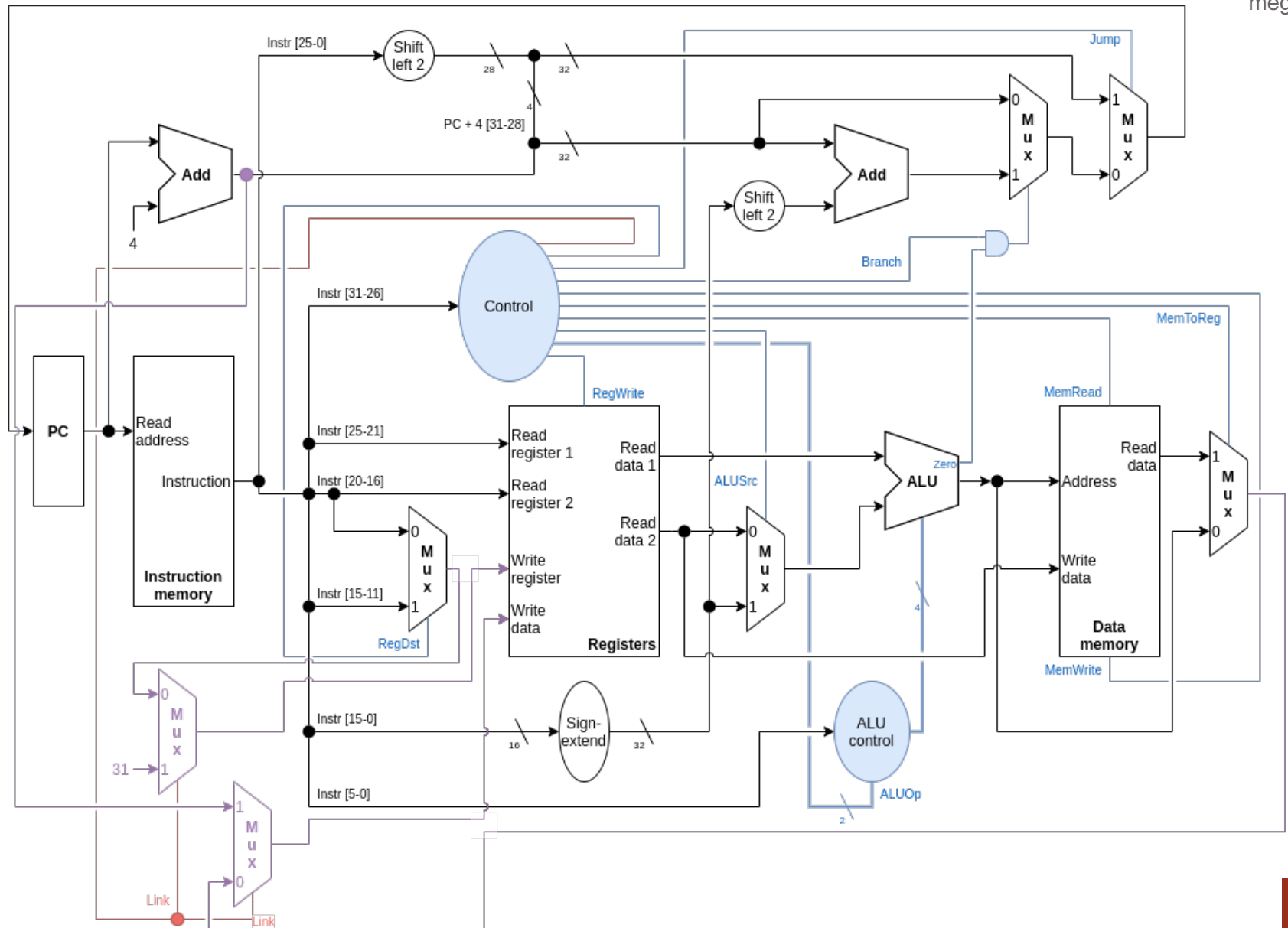


Architettura MIPS per add, beq, sw, lw, j e jal



Architettura MIPS per add, beq, sw, lw, j e jal

Oh,
meglio



Istruzione addi/la



Assembly: **addi rt, rs, costante** (add immediate) di tipo I

Cosa fa: Somma la parte immediata al registro **rs** e pone il risultato in **rt**

Unità funzionali: **ALU** per la somma (presente)

MUX che seleziona la parte immediata come secondo arg (presente)

Estensione del segno della parte immediata (presente)

Flusso dei dati: Registri[rs] → ALU

 Costante → Estensione segno → ALU

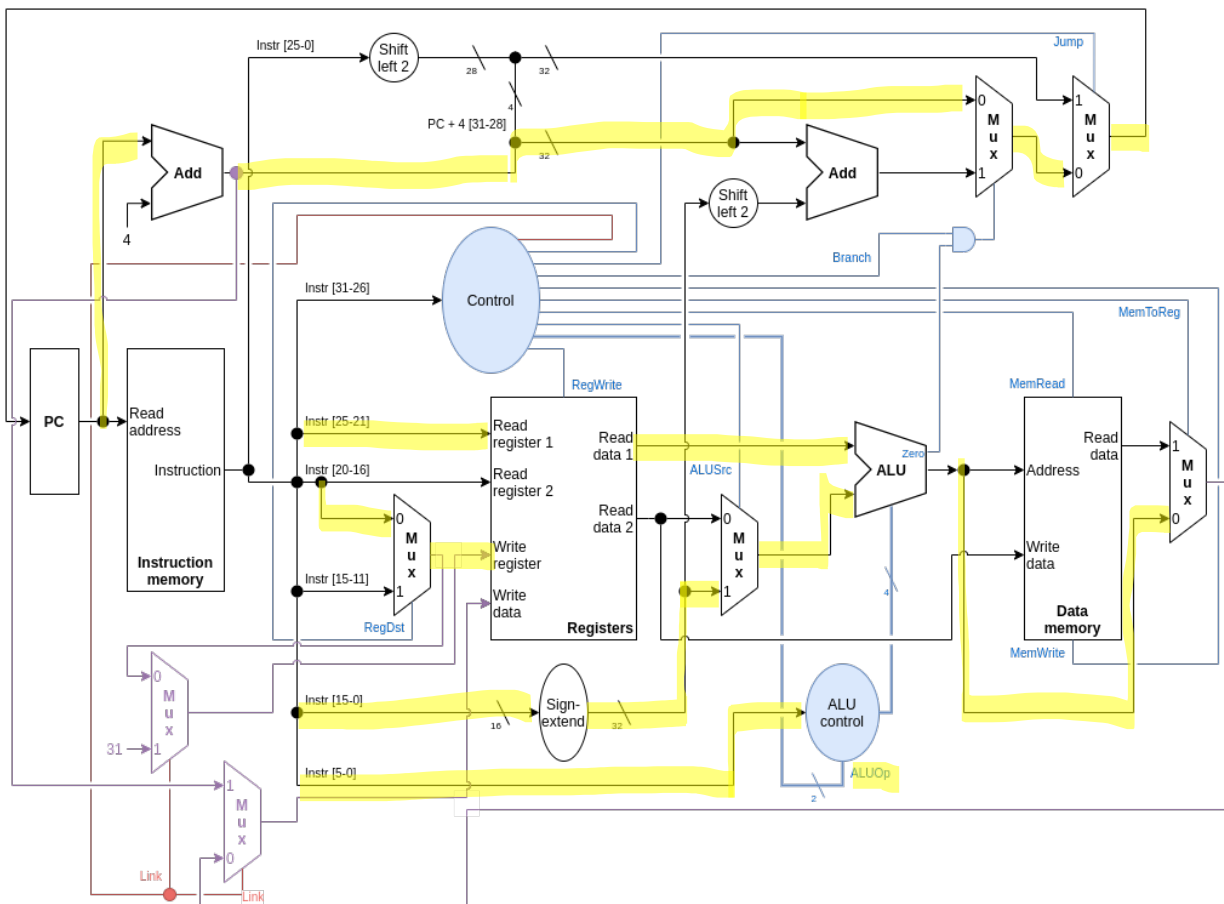
 ALU → Registri[rt]

Si comporta quasi come una **lw rt, costante(rs)** (ma lw memorizza l'indirizzo invece che il dato) ovvero come l'istruzione **la rt, costante(rs)/label** (load address). La CU produce i segnali:

Istruzione	Reg Dst	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
addi \$rd, \$rs, costante	0	1	0	1	X	0	0	0	0	0

Tempo necessario: **come una istruzione di tipo R**

Istruzione addi/la: niente circuiteria aggiuntiva



Istruzione	Reg Dst	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
addi \$rd, \$rs, costante	0	1	0	1	X	0	0	0	0	0

Tempo necessario: come una istruzione di tipo R

Istruzione jr (Jump to Register)

L'istruzione **jr rs** è di tipo R

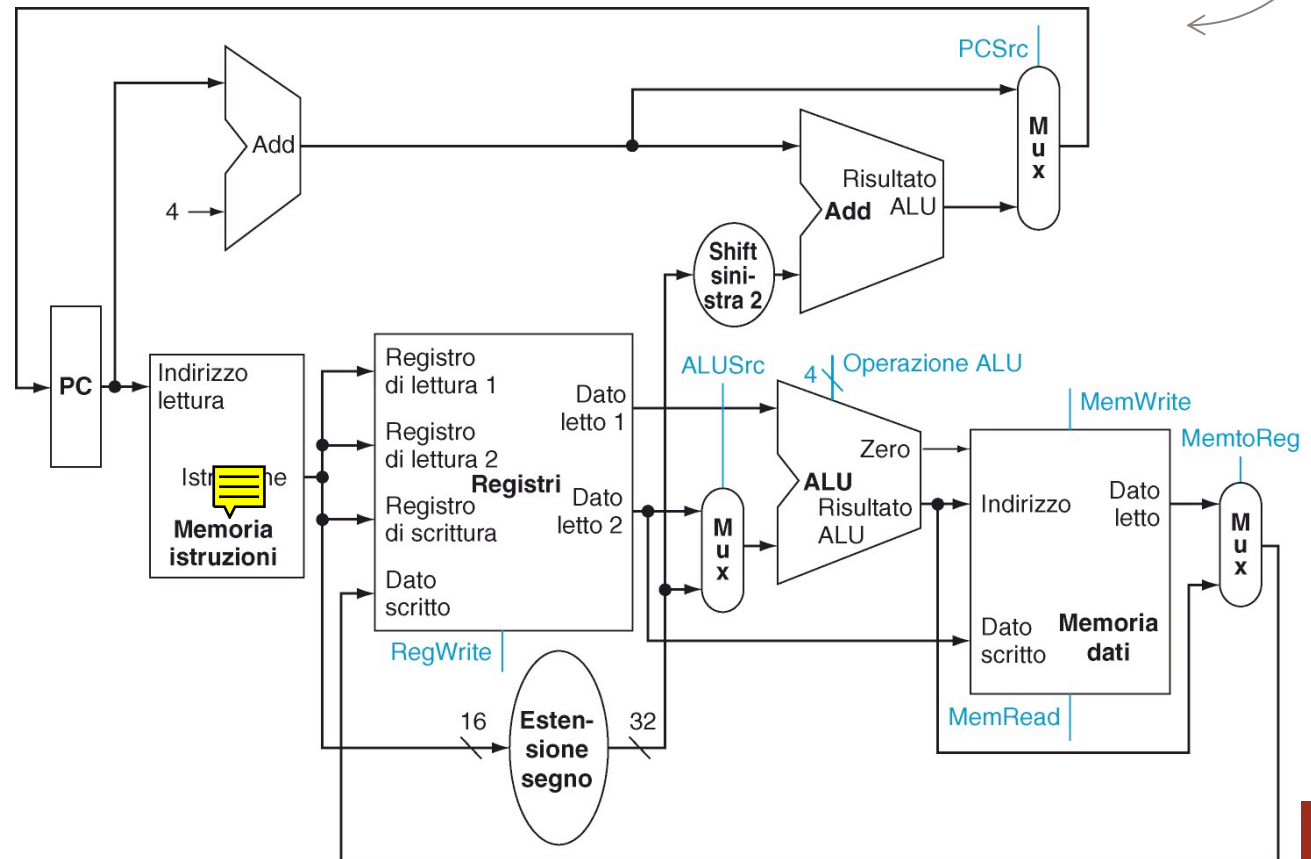
Cosa fa: trasferisce nel **PC** il contenuto del registro **rs**

Unità funzionali: **MUX** per selezionare il PC dall'uscita del blocco registri

Flusso dei dati: **Registri[rs] → PC**

Segnali di controllo: **JumpToReg** che abilita il MUX per inserire in PC il valore del registro

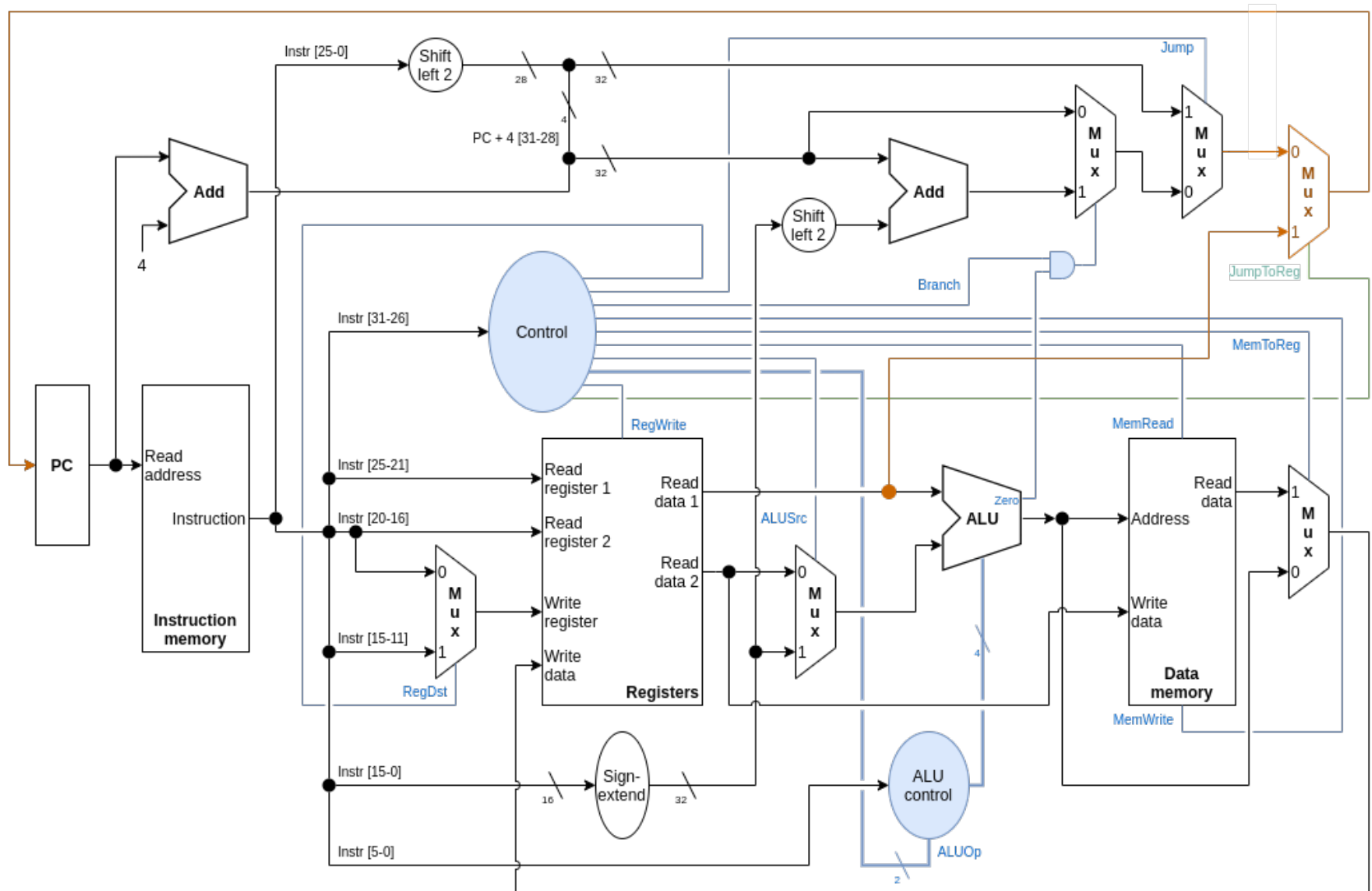
Qui:
scarabocchi



Tempo necessario:
Fetch+Reg

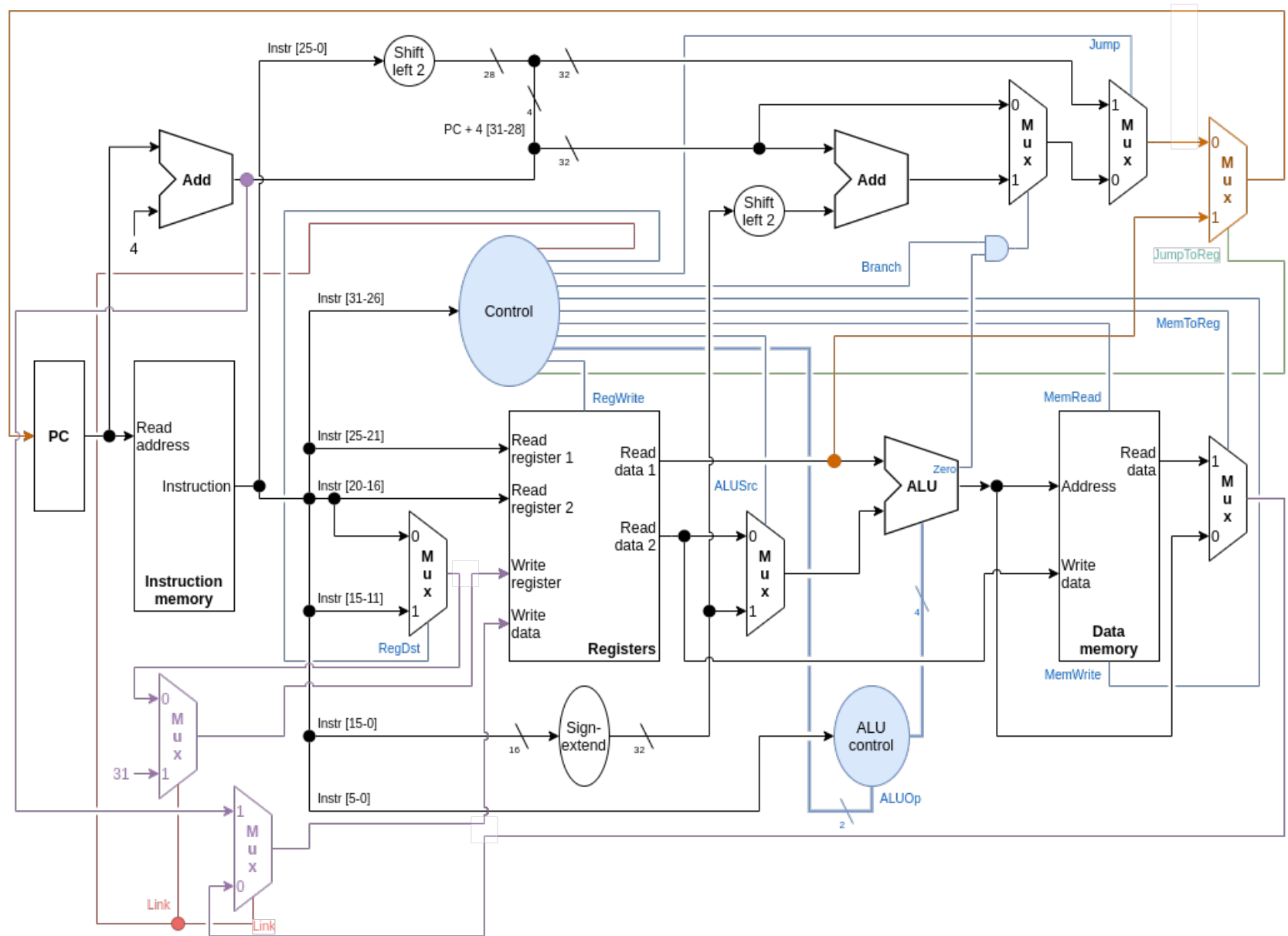
Architettura MIPS per aritm(+i), beq, sw, lw, la, j e jr

Oh,
meglio



Architettura MIPS per aritm(+i), beq, sw, lw, la, j, jal e jr

Oh,
meglio



Esercizio per casa



Aggiungere alla CPU l'istruzione **jrr rs** (Jump Relative to Register)

di tipo R, che salta all'indirizzo (relativo al PC) contenuto nel registro **rs**

Ovvero che esegue come prossima istruzione quella che si trova all'indirizzo

PC+4+Registri[rs]



- a) Modificare lo schema per realizzare l'istruzione (riciclando il più possibile)
- b) Indicare tutti i segnali di controllo che la CU deve generare
- c) Calcolare il tempo di esecuzione della istruzione assumendo che:

Accesso a memorie = 66ns, accesso ai registri = 33ns, ALU e sommatori = 100ns

Multiplexer a più ingressi (quando c'è una cascata di mux)

