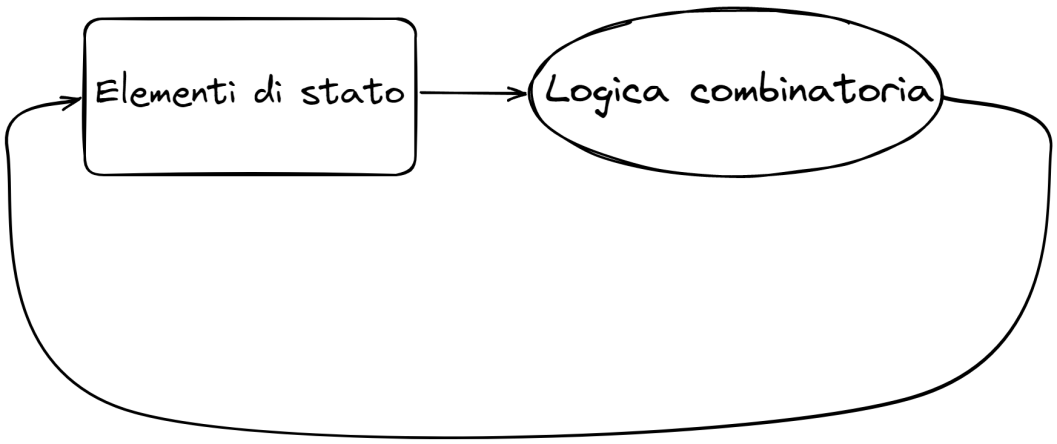
Architetture degli Elaboratori

Introduzione

La CPU (Central Processing Unit) è un componente elettronico, certe parti di essa rappresentano lo stato della macchina, altre la logica combinatoria che si occupa di modificare lo stato.

La CPU è composta da :

- **Control Unit (CU)**, si occupa di coordinare le operazioni da svolgere
- **Arithmetic Logic Unit (ALU)**, svolge le operazioni aritmetiche e logiche
- **Registri**, piccole memorie interne che immagazzinano temporaneamente i dati



Esistono due tipi di architetture di calcolatori :

CISC

Complex Instruction Set Computer, Le istruzioni sono di dimensione variabile, le operazioni vengono fatte in memoria, necessitando di molti accessi alla memoria per ogni istruzione, ha pochi registri interni e modi di indirizzamento più complessi con possibili conflitti tra istruzioni, necessitando quindi una pipeline più articolata.

RISC

Reduced Instrucion Set Computer, ha istruzioni di dimensione fissa, le operazioni vengono effettuate nell'ALU solamente tra i registri senza accedere alla memoria, ha molti registri interni e modi di indirizzamento più semplici.

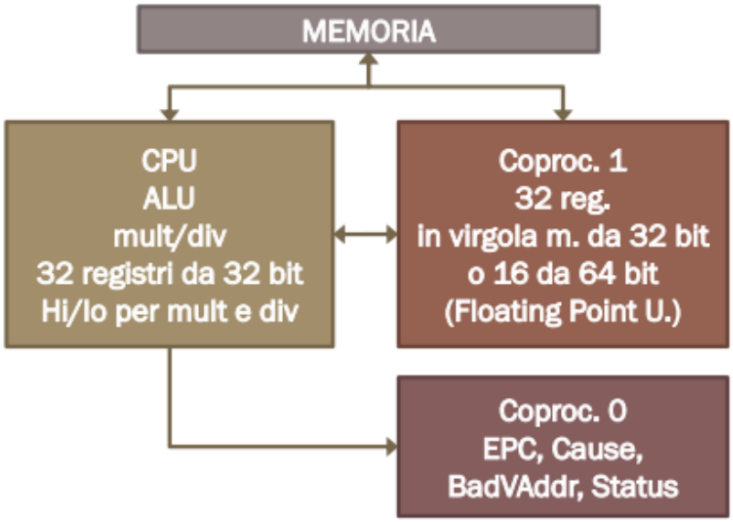
Caratteristiche dell'architettura MIPS

L'architettura **MIPS** è di tipo RISC, essa è composta da :

- Word da dimensione fissa di 32 bit
- Lo spazio di indirizzamento è di 2^{30} word da 32 bit ciascuna per un totale di 4 GB.

- La memoria è indicizzata al Byte, dato un indirizzo di memoria t, per leggere la word successiva bisogna spostarsi di 4 Byte, andando quindi all'indirizzo $t + 4$, questo perchè ogni parola da 32 bit corrisponde a 4 Byte.

L'architettura MIPS è dotata di 3 microprocessori (ALU, Coprocessore 0 e Coprocessore 1), e ben 32 registri della CPU principale, indicizzati da 0 a 31.



Il linguaggio Assembly

Le **istruzioni** del linguaggio Assembly seguono la seguente struttura:

```
<operazione><destinazione>,<sorgenti>,<argomenti>
```

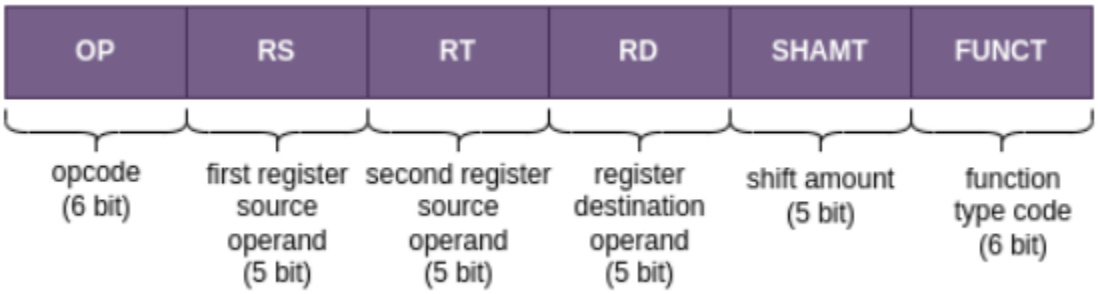
Un esempio è :

```
add $s0,$t0,$t1
```

Quest'istruzione non fa altro che leggere il registro $t0 e $t1, per sommarne i valori e scriverne il risultato dentro il registro $s0. Ogni istruzione rappresenta una word da 32 bit, essa viene letta ed intrepretata dall'assemblatore che tradurrà tale istruzione nel linguaggio macchina, ci sono 3 tipi di istruzioni :

Istruzioni R-Type

Sono istruzioni che lavorano sui **registri**, senza quindi accedere alla memoria, sono di tipo **aritmetico** e **logico** ed hanno il seguente formato dei bit :



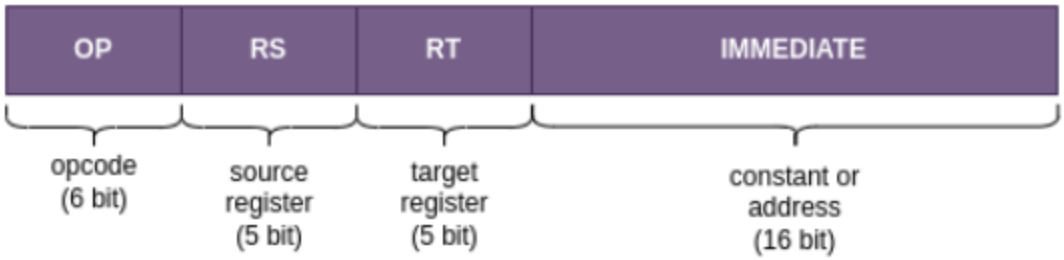
Ogni istruzione è quindi suddivisa in 6 parti :

1. **Opcode (OP)** : Rappresenta la categoria di istruzione da eseguire (Es: 000000 = aritmetica)
2. **First Register (RS)** : Rappresenta il primo registro “sorgente” dalla quale leggere il valore.
3. **Second Degister (RT)** : Rappresenta il secondo registro “sorgente” dalla quale leggere il valore.
4. **Destination Register (RD)** : Il registro di destinazione nella quale scriveremo il risultato
5. **Shift amount (SHAMT)** : Rappresenta la quantità di bit da shiftare
6. **Function code (FUNCT)** : Rappresenta un *estensione* dell’Opcode, e specifica il tipo di operazione da eseguire per categoria (somma o sottrazione per le operazioni aritmetiche).

```
add $t0, $s1, $s2
```

Istruzioni I-Type

Sono operazioni **immediate** di tipo **load** e **store**, sono utilizzate dai **salti condizionati**(i blocchi if/else del linguaggio assembly), hanno il seguente formato di bit:



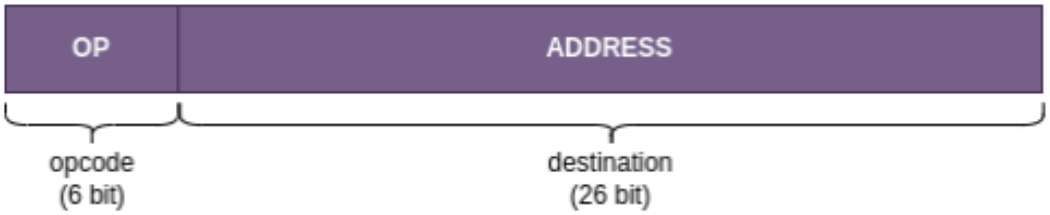
Vediamo nello specifico le 4 parti :

1. **Opcode (OP)** : Rappresenta la categoria di istruzione immediata da eseguire, quindi tra un registro ed un valore costante.
2. **Source Register (RS)** : Viene letto il valore del registro operando.
3. **Destination Register (RD)** : Il registro di destinazione nella quale scriveremo il risultato.
4. **Immediate** : Viene specificato il valore costante con cui effettuare l'operazione.

```
addi $t1, $s2, 17
```

Istruzioni J-Type

Sono istruzioni utilizzate dai **salti non condizionati**, ed hanno il seguente formato di bit :



Vediamo le 2 parti :

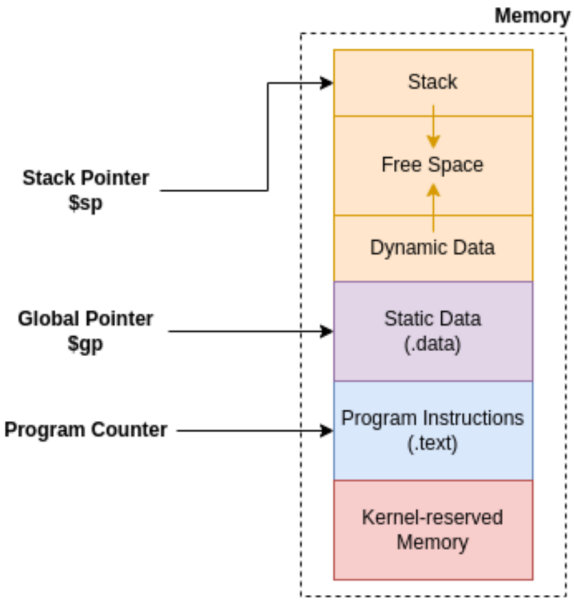
1. **Opcode (OP)** : Viene specificata l'operazione di jump incondizionato.
2. **Address** : Viene specificato l'indirizzo su cui effettuare il jump.

Alcune istruzioni

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Aritmetiche	Somma	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Operandi in tre registri
	Sottrazione	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Operandi in tre registri
	Somma immediata	`addi $s1,$s2,20`	`$s1 = $s2 + 20`	Utilizzata per sommare delle costanti
Trasferimento dati	Lettura parola	`lw $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di una parola da memoria a registro
	Memorizzazione parola	`sw $s1,20($s2)`	`Memoria[$s2+20]=$s1`	Trasferimento di una parola da registro a memoria
	Lettura mezza parola	`lh $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di una mezza parola da memoria a registro
	Lettura mezza parola, senza segno	`lhu $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di una mezza parola da memoria a registro
	Memorizzazione mezza parola	`sh $s1,20($s2)`	`Memoria[$s2+20]=$s1`	Trasferimento di una mezza parola da registro a memoria
	Lettura byte	`lb $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di un byte da memoria a registro
	Lettura byte senza segno	`lbu $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Trasferimento di un byte da memoria a registro
	Memorizzazione byte	`sb $s1,20($s2)`	`Memoria[$s2+20]=$s1`	Trasferimento di un byte da registro a memoria
	Lettura di una parola e blocco	`ll $s1,20($s2)`	`$s1=Memoria[$s2+20]`	Caricamento di una parola come prima fase di un'operazione atomica
	Memorizzazione condizionata di una parola	`sc $s1,20($s2)`	`Memoria[$s2+20]=$s1; $s1=0` oppure 1	Memorizzazione di una parola come seconda fase di un'operazione atomica
	Caricamento costante nella mezza parola superiore	`lui $s1,20`	`$s1 = 20 * 2^16`	Caricamento di una costante nei 16 bit più significativi
Logiche	And	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Operandi in tre registri; AND bit a bit
	Or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Operandi in tre registri; OR bit a bit
	Nor	`nor $s1,$s2,$s3`	`$s1 = ~($s2 | $s3)`	Operandi in tre registri; NOR bit a bit
	And immediato	`andi $s1,$s2,20`	`$s1 = $s2 & 20`	And bit a bit tra un operando in registro e una costante
	Or immediato	`ori $s1,$s2,20`	`$s1 = $s2 | 20`	OR bit a bit tra un operando in registro e una costante
	Scorrimento logico a sinistra	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Spostamento a sinistra del numero di bit specificato dalla costante
	Scorrimento logico a destra	`srl $s1,$s2,10`	`$s1 = $s2 >> 10`	Spostamento a destra del numero di bit specificato dalla costante
Salti condizionati	Salta se uguale	`beq $s1,$s2,25`	Se `($s1==$s2)` vai a PC+4+100	Test di uguaglianza; salto relativo al PC
	Salta se non è uguale	`bne $s1,$s2,25`	Se `($s1!=$s2)` vai a PC+4+100	Test di disuguaglianza; salto relativo al PC
	Poni uguale a 1 se minore	`slt $s1,$s2,$s3`	Se `($s2 < $s3) $s1 = 1;` altrimenti `$s1 = 0`	Comparazione di minoranza; utilizzata con bne e beq
	Poni uguale a uno se minore, numeri senza segno	`sltu $s1,$s2,$s3`	Se `($s2 < $s3) $s1 = 1;` altrimenti `$s1 = 0`	Comparazione di minoranza su numeri senza segno
	Poni uguale a uno se minore, immediato	`slti $s1,$s2,20`	Se `($s2 < 20) $s1 = 1;` altrimenti `$s1 = 0`	Comparazione di minoranza con una costante
	Poni uguale a uno se minore, immediato e senza segno	`sltiu $s1,$s2,20`	Se `($s2 < 20) $s1 = 1;` altrimenti `$s1 = 0`	Comparazione di minoranza con una costante, con numeri senza segno

Organizzazione della Memoria

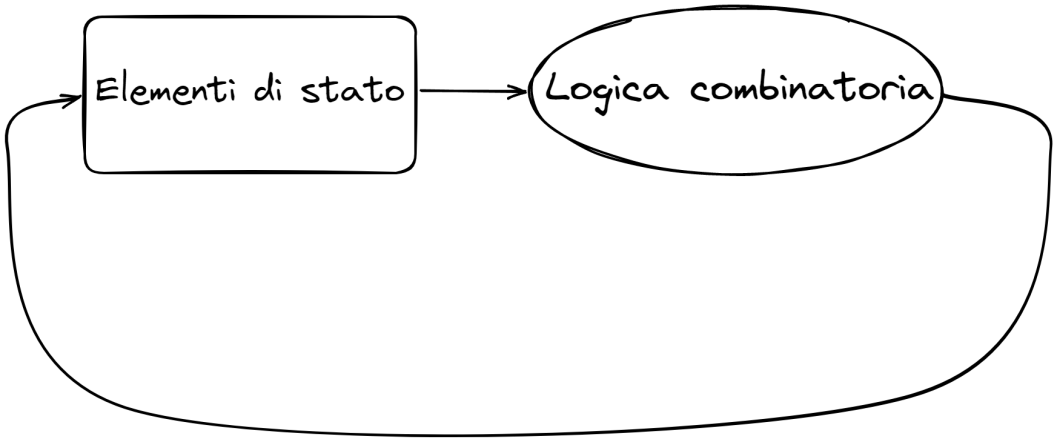
Vediamo le parti principali della struttura della memoria :



- **Stack** : Utilizzata per operazioni relative alle funzioni, si può espandere con la memoria libera condivisa, al suo interno viene utilizzato il registro $sp.
- **Dynamic Data** : Contiene i dati dinamici utilizzati durante l'esecuzione del programma.
- **Static Data** : Contiene i dati statici definiti all'avvio del programma (etichette sotto la direttiva .data)
- **Program Instructions** : Contiene le istruzioni del programma sotto la direttiva .text,in esso opera il program counter.
- **Kernel-reserved** : È uno spazio inutilizzabile, è riservato al Kernel del sistema operativo.

L’architettura MIPS

Vediamo come le istruzioni viste nel linguaggio assembly, interagiscono con le componenti della CPU, ogni istruzione è **l’attivazione in sequenza di più componenti** interconnessi tra loro in **un circuito sequenziale** sensibile al **fronte di salita**.



Progettazione della CPU

Vogliamo progettare una CPU che svolga istruzioni basilari, ricordando l’architettura di **Von Neumann**, sono necessari i seguenti componenti fondamentali :

- **Program Counter** - un registro contenente l’indirizzo in memoria dell’istruzione prossima da eseguire.
- **Register File** - un banco di registri contenete i dati ed i valori necessari alle istruzioni.
- **Memoria** - suddivisa tra memoria dati e memoria istruzioni.
- **ALU** - L’unità in grado di svolgere le operazioni logico-aritmetiche.
- **Control Unit** - un unità in grado di gestire i segnali necessari all’esecuzione di un istruzione.
- **Datapath** - L’insieme di interconnessioni tra i vari componenti

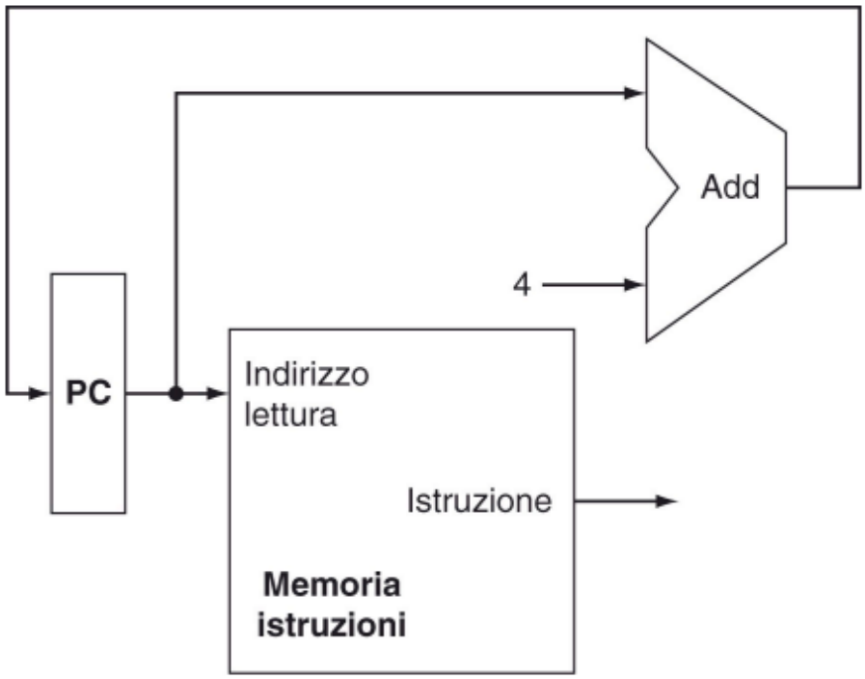
Le fasi di esecuzione di un istruzione

Adesso bisogna sfruttare questi ingredienti, e definire delle **fasi** nella quale essi interagiscono tra loro in modo tale da poter computare le istruzioni richieste, possiamo suddividere il ciclo di vita di un istruzione in :

1. Fetch
2. Decodifica
3. Esecuzione
4. Accesso alla memoria
5. WriteBack

Fetch

La fase di **Fetch** (*In italiano, "andare a prendere"*) prevede la lettura dell'istruzione successiva da eseguire, in tale fase sono necessari il **Program Counter**, contenente l'indirizzo (32 bit) in memoria dell'istruzione da eseguire e la **memoria destinata alle istruzioni**, che prende in input l'indirizzo (32 bit) dell'istruzione da prelevare e in output l'istruzione corrispondente a tale indirizzo (32 bit), l'input della memoria sarà quindi il registro Program Counter, essendo esso l'istruzione successiva. Bisognerà poi **incrementare l'indirizzo del PC di 4 byte**, in modo tale che al ciclo di clock successivo verrà eseguita la prossima istruzione. È dunque necessario aggiungere all'architettura un **Sommatore** avente in input il PC ed un valore costante di 4, che eseguirà tale somma ad ogni ciclo di clock.



Decodifica

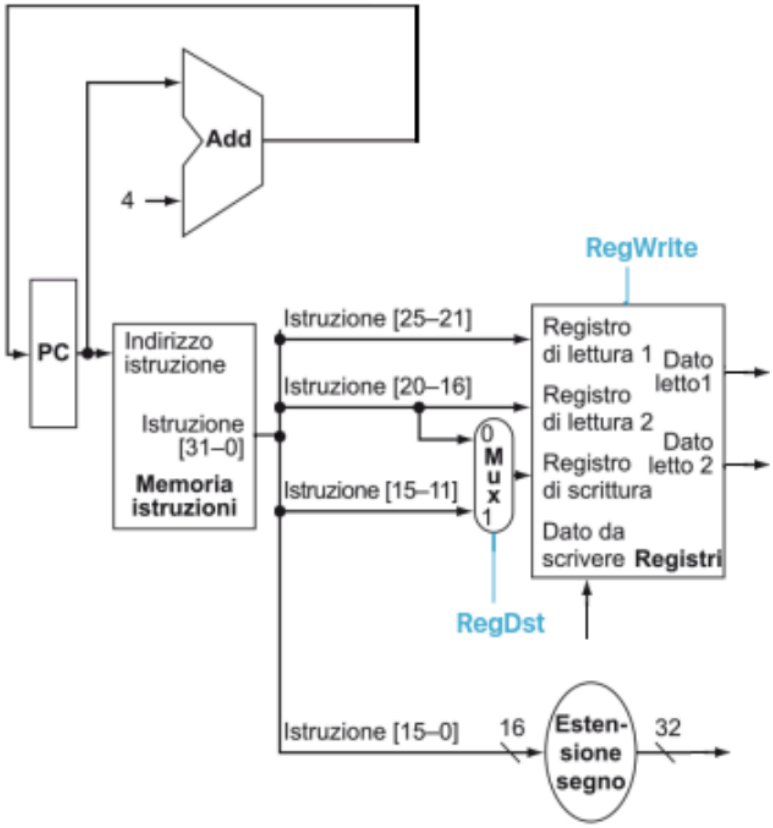
Avviene poi la fase di **decodifica dell'istruzione**, quindi, l'istruzione in output dalla fase di fetch viene **scomposta in diversi campi** prelevandone i contenuti, vediamo come sono codificate le varie istruzioni (Tipo R, Tipo I, Tipo J).

Nome	Campi						Commenti
Dimensione del campo	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Tutte le istruzioni MIPS sono a 32 bit
Formato R	op	rs	rt	rd	shamt	funct	Formato delle istruzioni aritmetiche
Formato I	op	rs	rt	indirizzo / costante			Formato delle istruzioni di trasferimento dati di salto condizionato e immediate
Formato J	op	indirizzo di destinazione					Formato delle istruzioni di salto incondizionato

Ricordare che :

- **op** - Codice che seleziona l'operazione da eseguire.
- **rs** - Primo registro sorgente dalla quale leggere il valore.
- **rt** - Secondo registro sorgente dalla quale leggere il valore.
- **rd** - Registro di destinazione nella quale scrivere il risultato.
- **shamt** - Quantità di bit da *shiftare*.
- **funct** - estensione dell'*op*, specifica il tipo di operazione da eseguire.

Essendo l'architettura capace di decodificare tutti e 3 i formati delle istruzioni, sono necessari dei **mux (multiplexer)** e dei segnali di controllo che selezionino e modifichino il comportamento a seconda dell'istruzione.



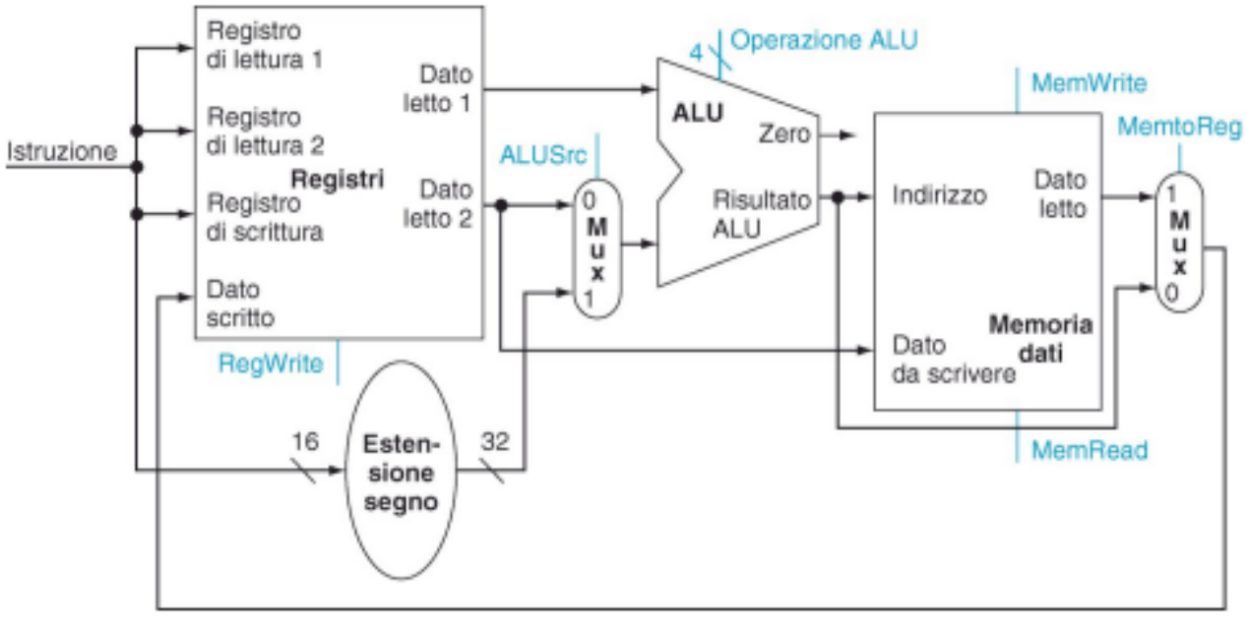
L’istruzione viene **splittata ,** vediamo cosa sono e a cosa servono i componenti ed i segnali aggiunti (tenendo conto della tabella di decodifica delle istruzioni precedentemente mostrata) :

- **Registro di lettura 1** - prende in input i bit corrispondenti al **campo $rs** (Primo registro sorgente), nel range [25-21].
- **Registro di lettura 2** - prende in input i bit corrispondenti al **campo $rt** (Secondo registro sorgente), nel range [20-16].
- **RegDst** - è un segnale di controllo, usato come **selettore di un mux** avente in input i campi $rt, nel range [20-16] e $rd (Registro di destinazione) nel range [15-11].
- **Registro di scrittura** - prende in input i bit selezionati dal mux precedentemente citato (avente RegDst come selettore), scrivendo nel registro corrispondente, il contenuto del dato da scrivere passato in input.
- **RegWrite** - non è altro che un segnale di scrittura, che se positivo, abilità la scrittura sul **Register File**.

- **Estensione del segno** - Quando si esegue un **istruzione di tipo I (Immediate)**, i bit nel range [15-0] rappresentano la costante dichiarata, questo componente non fa altro che **estendere il segno** di quel valore, da 16 bit a 32 bit (Si ricordi che i numeri utilizzati sono in complemento a 2).

Esecuzione

Avendo adesso i dati necessari prelevati dall'istruzione, prima "estratta", e poi decodificata possiamo procedere con **l'esecuzione** utilizzando **l'ALU**, ed eventualmente accedendo alla **memoria dati**.



Come si vede in figura, l'ALU presenta **4 bit di controllo**, essi stabiliscono/selezionano l'operazione da svolgere.

Segnali ALU	Operazione
0000	AND
0001	OR
0010	ADD (addizione)
0110	SUB (sottrazione)
0111	SLT (poni a 1 se minore)
1100	NOR

- **ALUSrc** - È presente un mux selezionato dal segnale ALUSrc, esso seleziona **il secondo dato che sarà di input nell'ALU**, che può essere il **registro $rs** (per le istruzioni di tipo R), o il **valore immediato** che è stato esteso (per le istruzioni di tipo I).
- L'ALU ha in output 2 segnali, il primo è il **risultato dell'operazione svolta** (32 bit), il secondo è un flag chiamato **Zero**, è un solo bit, e varrà 1 se il risultato dell'operazione svolta è uguale a 0, è un segnale fondamentale per le istruzioni di **branch**.

Vediamo adesso l'implementazione dell'accesso alla **memoria dati**:

- L'ALU calcola gli indirizzi di memoria con cui interagire, ed il risultato dell'operazione verrà usato come input d'indirizzo alla memoria dati.
- L'istruzione `lw`, è l'unica in grado di leggere dalla memoria, quindi nel caso venga eseguita, è necessario che il **dato prelevato dalla memoria** venga restituito come dato da scrivere all'interno del registro di scrittura selezionato, in qualsiasi altro caso, sarà necessario restituire il risultato dell'ALU, dato che bisogna prevedere questi due casi, si utilizza un mux con un segnale di controllo denominato **MemToReg**, determina quale dei due dati andranno scritti nei registri.
- Essendo che solamente `lw` è in grado di leggere dalla memoria, e solamente `sw` è in grado di scrivere su di essa, la memoria presenterà due segnali di controllo, **MemRead** (che si occupa di abilitare la lettura) e **MemWrite** (che si occupa di abilitare la scrittura).

Unità di controllo dell'ALU

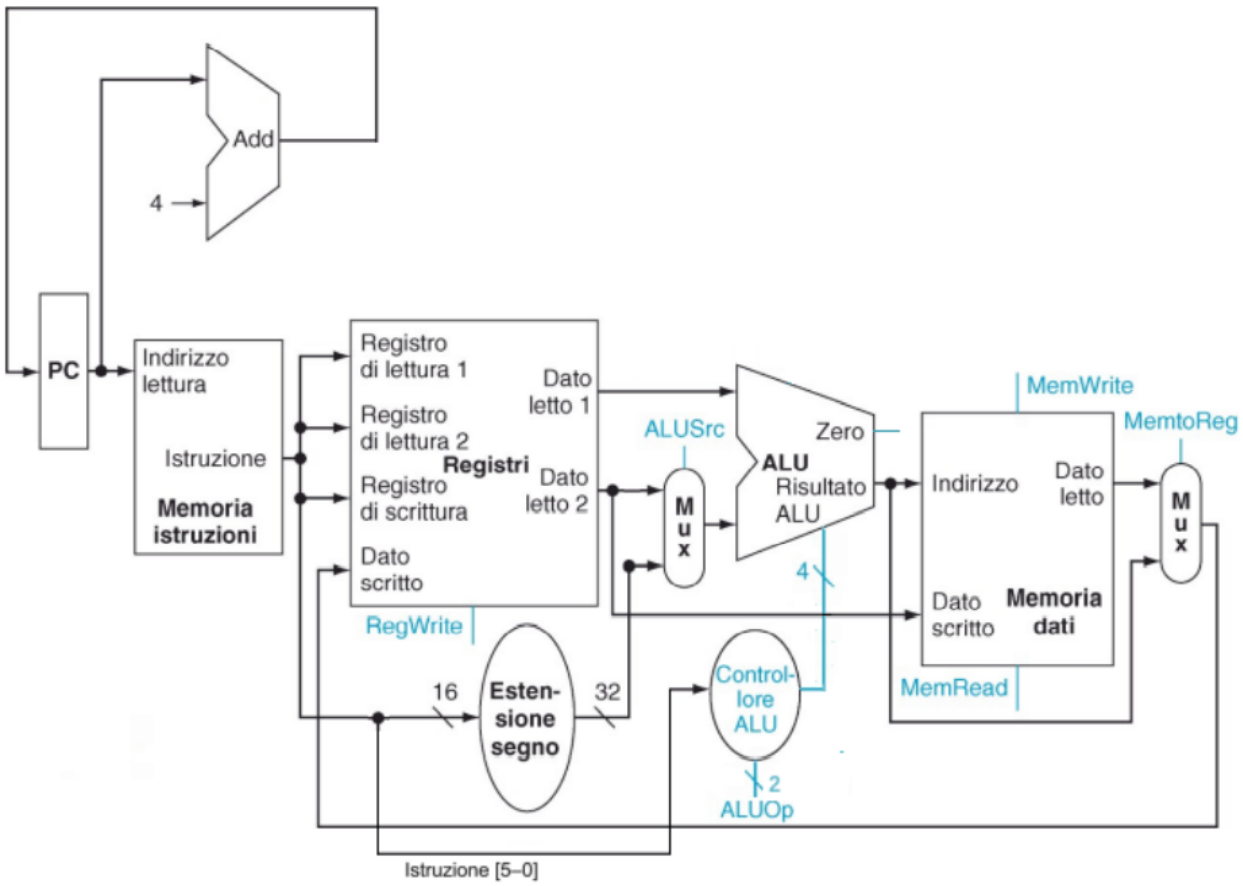
Nel circuito completo, è implementata un **unità di controllo** per l'ALU, con lo scopo di semplificare l'uso dei segnali di controllo che determinano l'operazione da eseguire. Tale unità ha come input il campo **funct** dell'istruzione, avente i bit contenuti nel range [5-0], e due segnali di controllo **ALUOp**, che determinano se eseguire tale operazione.

Istruzione	ALUOP	Campo funct	Segnali ALU	Operazione
`lw` e `sw`	0 0	- - - - -	0010	ADD
`beq`	- 1	- - - - -	0110	SUB
`add`	1 -	- 0000	0010	ADD
`sub`	1 -	- 0010	0110	SUB

`and`	1 -	- 0100	0000	AND
`or`	1 -	- 0101	0001	OR
`slt`	1 -	- 1010	0111	SLT

Il segno “-” equivale ad un segnale Don’t Care

Vediamo come appare il circuito con l’implementazione dell’unità di controllo ALU.



Salti condizionati e incondizionati

Salti condizionati (Branch)

Per eseguire i **salti condizionati** come Branch if Equal (`beq`) e Branch if not Equal (`bne`), ricordando che essi stessi possono saltare solo un limitato numero di istruzioni, si utilizza l’ALU

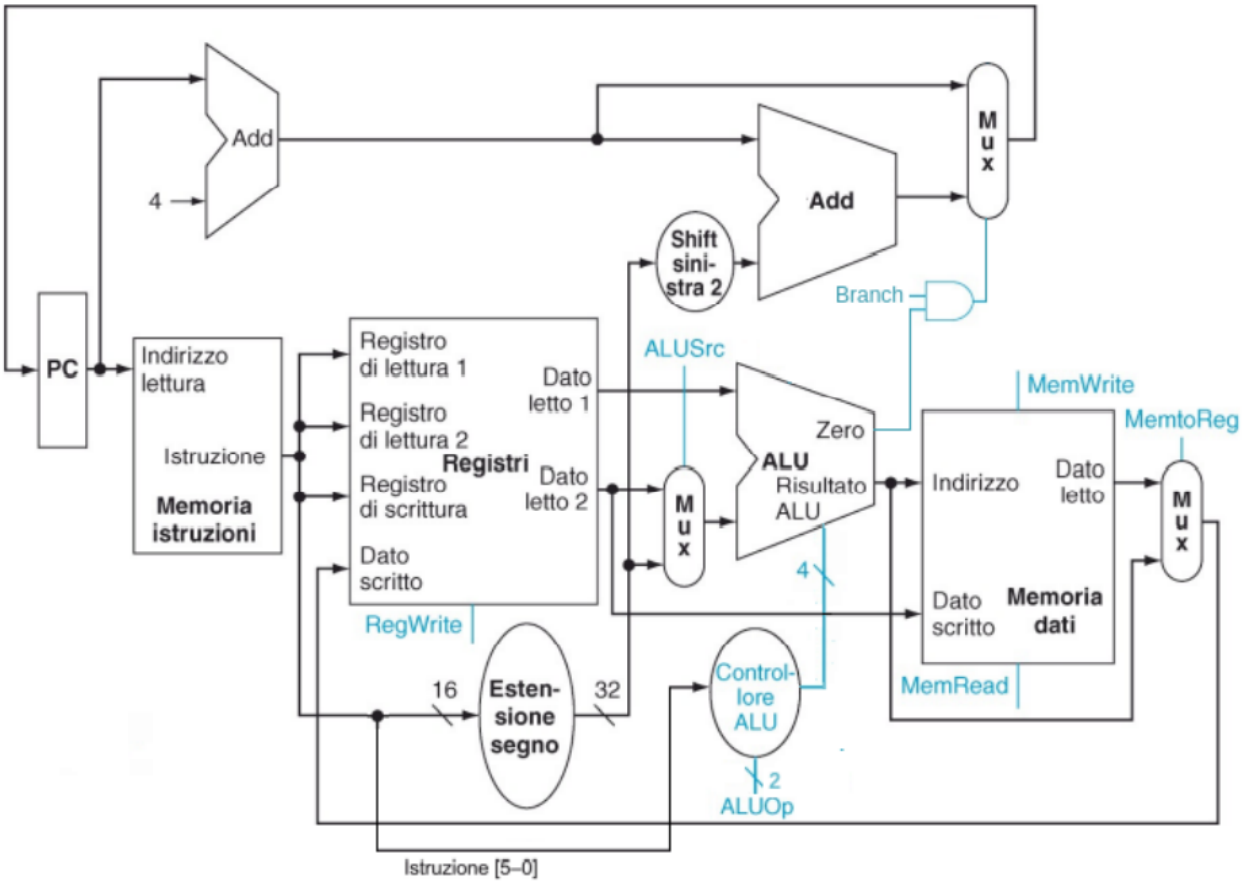
come **comparatore**, eseguendo una semplice **sottrazione** tra i due elementi in input, utilizzando il nostro **flag Zero**, che sarà uguale ad 1, se il risultato della sottrazione vale 0, quindi se **i due numeri sono uguali**.

`beq $rs, $rt, 5` → Salta di 5 istruzioni solo se `$rs` − `$rt` $== 0$

Nell'istruzioni di salto condizionato, la **parte immediata** dell'istruzione, contenuta nel range di bit [15-0], rappresenta il numero di istruzioni da saltare, bisognerà quindi per ottenere il numero di byte corrispondenti da saltare, moltiplicare tale numero per 4, e ciò equivale a **shiftare a sinistra di due posizioni** il numero, basterà poi sommare tale valore all'indirizzo pre-calcolato dell'istruzione successiva, equivalente a ProgramCounter+4.

`beq $rs, $rt, offset` → PC $=$PC$+4+($`offset` $<< 2)$ solo se `$rs` − `$rt` $== 0$

Sarà quindi necessario introdurre un MUX, avente come selettore l'AND tra il **flag Zero** e il segnale di controllo **Branch**, che varrà 1 solo se l'istruzione eseguita è un branch. Introdurremo pure un ulteriore **sommatore** che si occuperà di eseguire la somma tra il PC+4 e il numero di istruzioni da saltare.



Salti incondizionati (Jump)

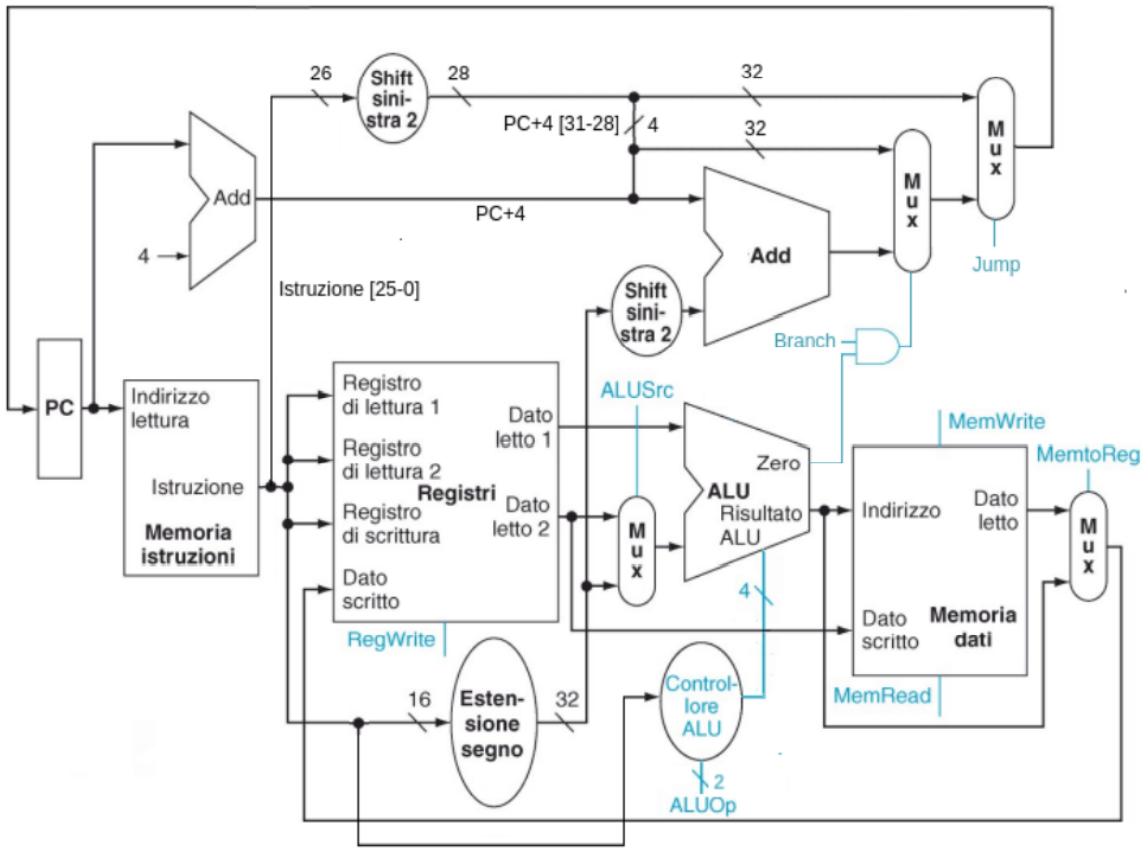
Essendo il salto incondizionato un istruzione di salto **assoluta**, ossia non condizionata dall'istruzione corrente, sarà necessario fare alcune considerazioni aggiuntive.

- L'indirizzo del salto calcolato, deve essere confinato all'interno della **zona di memoria dedicata alle istruzioni**, e per mantenere tale condizione valida i 4 bit del range [31-28] del PC rimangono invariati.
- La parte immediata nell'istruzione di salto incondizionato è rappresentata nel range [25-0] dell'istruzione, essa contiene il numero dell'istruzione a cui saltare, è necessario quindi **shiftare a sinistra** tale valore di due posizioni, ottenendo l'indirizzo in byte, avendo così i **28 bit** indicanti l'effettiva a istruzione a cui saltare.

Facendo così, l'indirizzo calcolato sarà formato dall'unione dei 4 bit invariati del PC più i 28 bit calcolati tramite lo shift.

`j label` $\rightarrow$ PC = PC $+4[31-28]\ OR\ ($ `label` $<< 2\ )$

oppure $\rightarrow$ PC = PC $+4[31-28]\ OR\ (istruzione[25-0] << 2\ )$



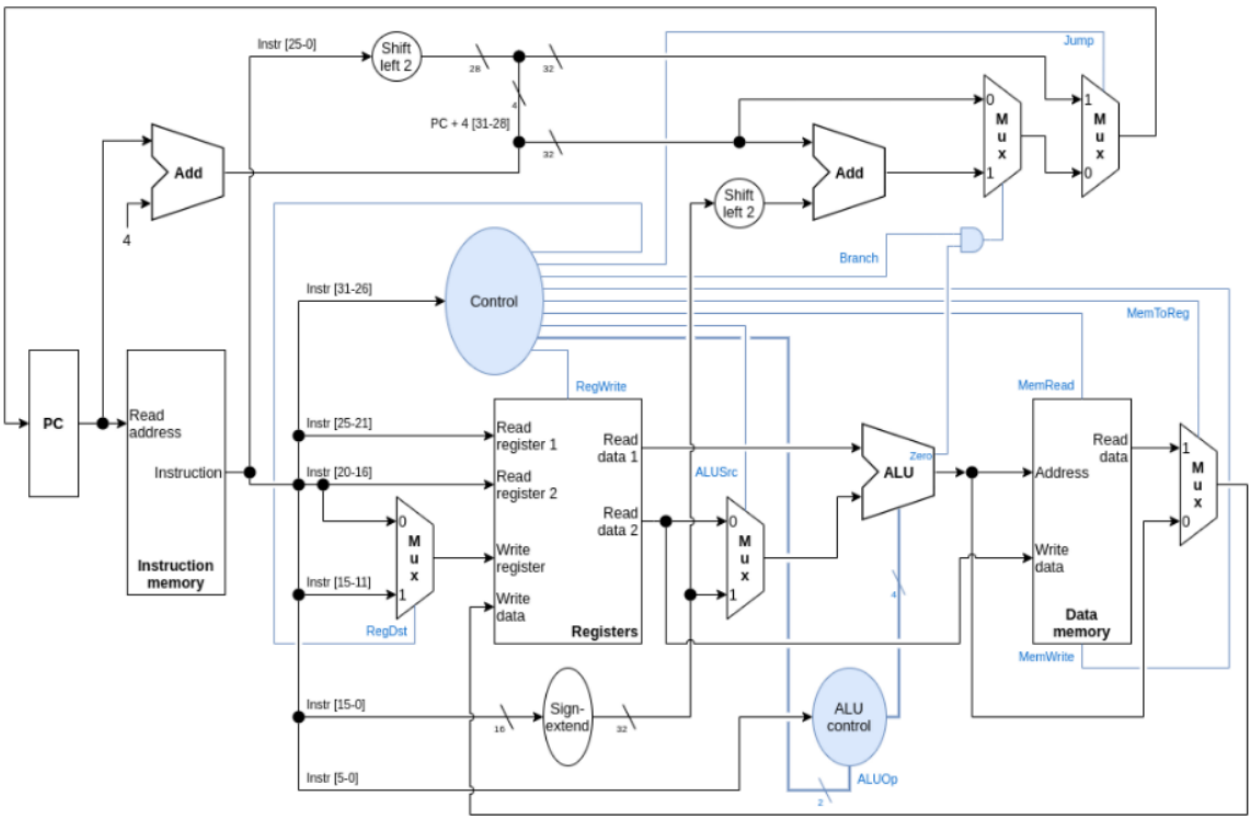
Control Unit

Abbiamo visto come nel circuito precedentemente mostrato, molti dei mux implementati utilizzano come selettori dei **segnali di controllo** specifici che condizionano il comportamento dell'architettura in base al tipo di istruzione richiesta, tuttavia, non abbiamo specificato dove vengono assegnati i valori di tali segnali (come ad esempio *Jump*, *Branch* o *MemToReg*).

Tali **segnali** sono gestititi dalla **Contro Unit (CU)**, che a seconda dei bit in input dell'**opcode**, di range [31-25], attiverà i segnali necessari all'istruzione stessa. Vediamo quindi quali di questi 6 bit di opcode vengono selezionati come segnali.

Istruzione	RegDst	RegWrite	ALUSrc	ALUOp [1]	ALUOp [0]	MemRead	MemWrite	MemToReg	Branch	Jump
Tipo R	1	1	0	1	-	-	0	0	0	0
Tipo I	0	1	1	1	-	-	0	0	0	0
lw	0	1	1	0	0	1	0	1	0	0
sw	-	0	1	0	0	0	1	-	0	0
beq	-	0	0	-	1	-	0	-	1	0
j	-	0	-	-	-	-	0	-	-	1

Risulta adesso più chiaro, come i primi 6 bit di un istruzione condizionino il funzionamento dell'architettura, vediamo adesso lo **schema completo** con la CU implementata :



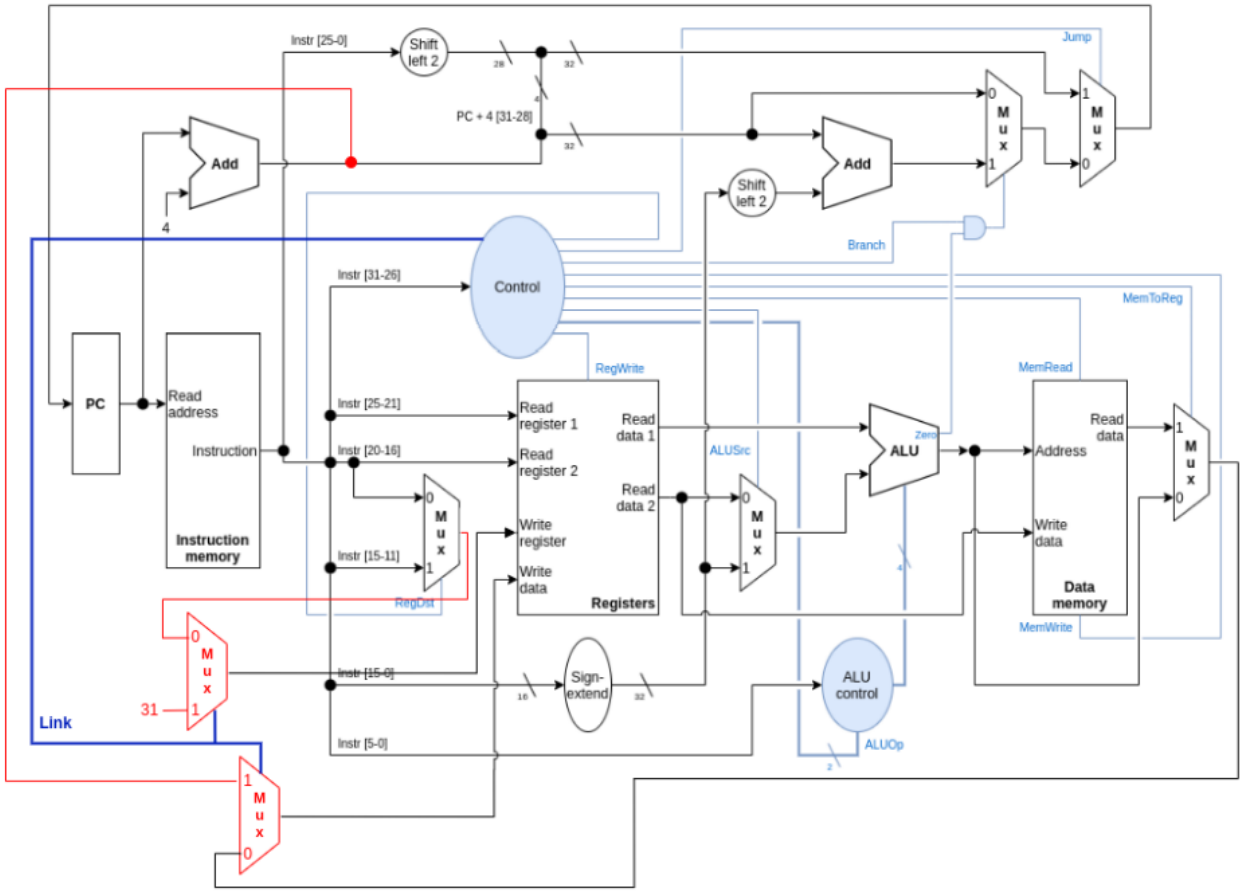
Aggiungere nuove istruzioni

Si dia il caso che vogliamo aggiungere un'**istruzione aggiuntiva** a quelle già esistenti, dovendo quindi modificare l'architettura.

L'istruzione `jal` (Jump and Link)

Vogliamo aggiungere un istruzione di tipo J, `jal` (Jump and Link), in grado di effettuare un salto indicato dalla parte immediata, salvando però nel registro `$ra` il valore PC$+4$. La parte del salto è già implementata nell'architettura, dobbiamo però aggiungere due MUX aventi come selettore un nuovo segnale di controllo, **Link**.

- **il primo mux** seleziona tra l'output del mux controllato dal segnale RegDst, ed il valore costante 31, in modo da poter utilizzare $ra come registro di scrittura.
- **il secondo mux** seleziona tra l'output del mux controllato dal segnale MemToReg, ed il valore PC+4 in modo da utilizzarlo come dato da scrivere dentro $ra.



Vediamo adesso come l'**opcode** dovrà interagire con i vari segnali :

Istruzione	RegDst	RegWrite	ALUSrc	ALUOp [1]	ALUOp [0]	MemRead	MemWrite	MemToReg	Branch	Jump	Link
jal	-	1	-	-	-	-	0	-	-	1	1