



SAPIENZA
UNIVERSITÀ DI ROMA

UNIVERSITÀ "SAPIENZA" DI ROMA
FACOLTÀ DI INFORMATICA

Architettura degli Elaboratori

Appunti integrati con il libro "Computer Organization and Design" - D.A. Patterson, J.L. Hennessy.

Author
Simone Bianco

10 agosto 2022

Indice

0	Introduzione	1
1	Introduzione all'Architettura MIPS	2
1.1	Istruzioni, Assemblatore e Compilatore	2
1.2	Architettura di Von Neumann, CPU e Memorie	4
1.3	L'Architettura MIPS 2000	6
2	Il linguaggio Assembly MIPS	8
2.1	Formato delle istruzioni	8
2.2	Lista delle istruzioni	11
2.3	Organizzazione della Memoria	12
2.3.1	Parti della memoria	14
2.4	Direttive principali ed Esempi di codice	15
2.5	Salti condizionati e Salti assoluti	17
2.6	Vettori e Matrici	21
2.7	System Calls	25
2.7.1	Pseudoistruzioni	26
2.8	Funzioni e Procedure	27
2.8.1	Stack di memoria	29
2.8.2	Funzioni ricorsive	33
3	L'Architettura MIPS	35
3.1	Progettazione della CPU	35
3.1.1	Fase di Instruction Fetch	36
3.1.2	Fase di Instruction Decode	37
3.1.3	Fase di Instruction Execute	38
3.1.4	Aggiunta di Salti Condizionati e Incondizionati	40
3.1.5	Control Unit ed Esecuzione delle istruzioni di base	43

Capitolo 0

Introduzione

Il seguente corso è volto all'apprendimento dei principi fondamentali impiegati nel **progettare un calcolatore moderno** attraverso un focus sulla struttura interna di un **microprocessore MIPS** e il **linguaggio assembly** ad esso legato (MIPS asm):

- **Introduzione al calcolatore e alle istruzioni MIPS:** rappresentazione delle istruzioni nel calcolatore in assembly MIPS, utilizzo della memoria per salvare variabili e dati, utilizzo degli operatori logici, strutture di controllo, vettori e matrici.
- **Sviluppo di programmi avanzati:** chiamate di sistema e funzioni, gestione dello stack, chiamata di funzioni annidate e ricorsione singola/multipla.
- **Progettazione della CPU MIPS:** progettazione della CPU MIPS a singolo ciclo di clock e istruzioni assembly ad essa relative, introduzione alla pipeline e agli hazard, progettazione della CPU con pipeline e gestione dei data e control hazard.
- **Progettazione multilivello:** introduzione alla memoria cache, associatività e multilivello, cache multilivello e memoria virtuale, caches multiple e gestione delle eccezioni.

ATTENZIONE: all'interno di questo corso verranno dati per assunti i concetti principali espressi all'interno del corso "*Progettazione di Sistemi Digitali*", in particolare i concetti legati al **sistema numerico binario** (notazioni, utilizzi, algebra, ..) e alla **memorizzazione dei dati**.

Capitolo 1

Introduzione all'Architettura MIPS

1.1 Istruzioni, Assemblatore e Compilatore

Per comunicare con un sistema elettronico è necessario inviare dei segnali elettrici, corrispondenti a due semplici azioni: far passare corrente attraverso un componente (*on*) o non farla passare (*off*). Cercando di astrarre in modo matematico tale concetto, queste due azioni possono facilmente essere tradotte in quello che è il **sistema numerico binario**, dove un 1 rappresenta un segnale attivo ed uno 0 un segnale spento. Ogni cifra binaria (dunque 1 o 0) viene definita col termine **bit**.

A seconda di come vengono progettati, ogni componente di un calcolatore reagisce in base alle sequenze di 0 ed 1 che gli vengono impartite. Tali sequenze vengono dette **istruzioni** e possono essere interpretate da un calcolatore come un **comando** effettivo da svolgere o come un **numero**. Ad esempio, la sequenza di bit 1000110010100000 dice al calcolatore di effettuare la somma tra due numeri.

Linguaggio Assembly ed Assemblatori

I primi programmatori comunicavano con i calcolatori utilizzando direttamente i numeri binari, definendo sequenza per sequenza le istruzioni da svolgere. Ovviamente, tale processo risulta estremamente complesso, laborioso e soggetto a molti errori (anche un semplice bit errato può voler dire un output completamente diverso da quello desiderato).

Per risolvere tale problema, si penso ad una soluzione geniale: **utilizzare i calcolatori stessi per programmare altri calcolatori**. Nacquero così dei programmi in grado di tradurre delle **notazioni simboliche molto più semplici** da utilizzare in vere ed effettive istruzioni. Tali programmi vengono chiamati **assemblatori**.

Per esempio, l'istruzione

add A, B

viene tradotta dall'assemblatore in

1000110010100000

ossia l'istruzione in grado di comunicare al calcolatore di sommare il numero A e il numero B. Questo **linguaggio simbolico** viene detto **Linguaggio Assembler (o Assembly Language)**.

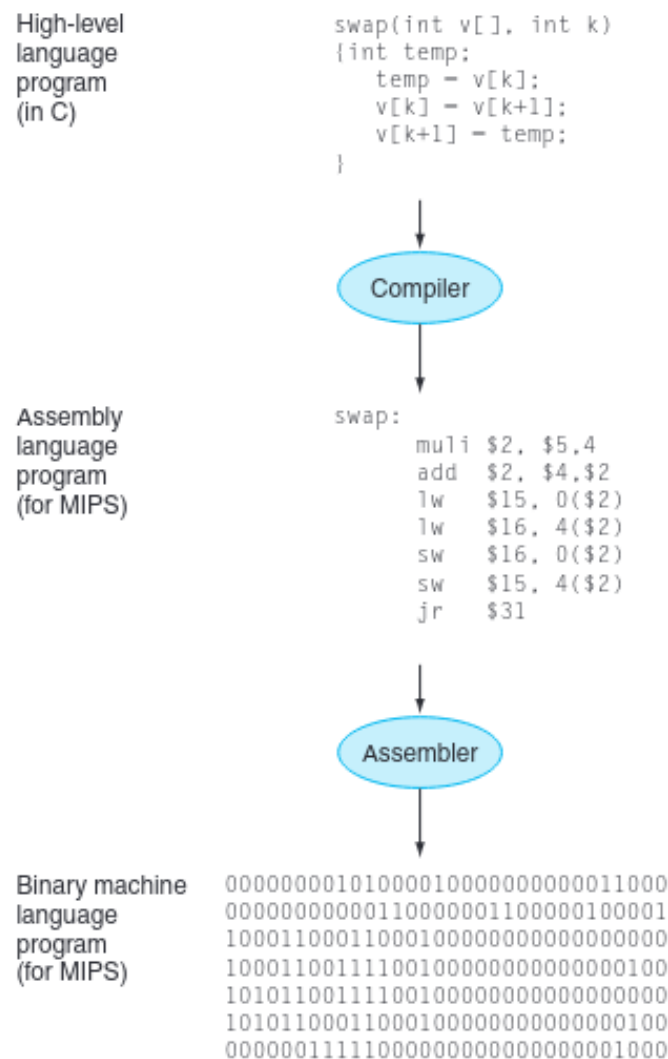
L'utilizzo del linguaggio assembly permise lo sviluppo agile e controllato di programmi avanzati, eliminando (parzialmente) fattori come l'**errore umano** (un calcolatore non può sbagliare a scrivere un bit al contrario di un umano) e la **lentezza** di progettazione.

Linguaggi ad Alto Livello e Compilatori

Nonostante esso risulti comunque estremamente più leggibile ed utilizzabile rispetto al **codice macchina** (ossia l'insieme di 0 ed 1 letto dal calcolatore), il **codice assembly** risulta comunque essere difficilmente interpretabile. Seguendo la stessa logica utilizzata in precedenza, gli esperti informatici decisero di sviluppare numerosi **linguaggi ancora più astratti** (Fortran, Cobol, C, ...) che permettessero di semplificare ulteriormente lo sviluppo del software. Tali linguaggi di programmazione vengono attualmente definiti col termine **linguaggi ad alto livello**.

I linguaggi di programmazione vengono interpretati da un software chiamato **compilatore**, il quale **traduce il codice ad alto livello in codice assembly**, il quale verrà poi a sua volta tradotto dall'**assemblatore** in codice macchina.

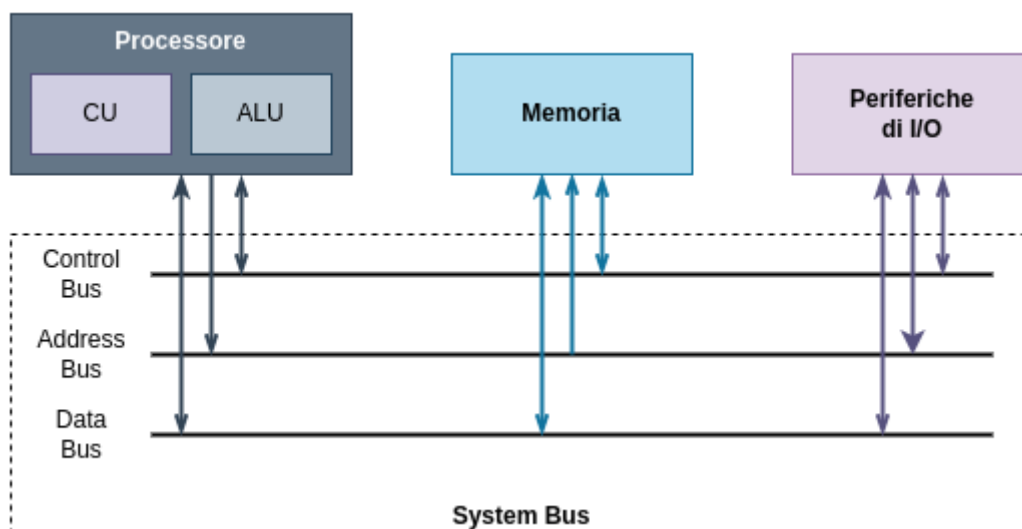
L'intera **catena di astrazione**, dunque, corrisponde a



1.2 Architettura di Von Neumann, CPU e Memorie

L'esempio classico di architettura generica di un computer è l'**Architettura di Von Neumann**, concepita da John Von Neumann, un noto matematico, fisico e informatico che visse nei tempi della seconda guerra mondiale. Neumann concepì un'architettura per i calcolatori **semplice e rivoluzionaria**, tanto che ancora oggi viene utilizzata come base per la realizzazione della maggior parte dei calcolatori comuni. Il modello prevedeva che il calcolatore dovesse essere costituito da **quattro elementi fondamentali**:

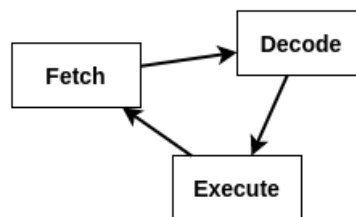
- **Central Processing Unit (CPU)**, ossia l'unità centrale di elaborazione (anche chiamato **processore**). Si occupa di eseguire una dopo l'altra tutte le istruzioni che compongono un **processo**, ossia un programma caricato in memoria. È a sua volta costituita da tre elementi:
 - **Control Unit (CU)**, che svolge e coordina tutte le operazioni da svolgere
 - **Arithmetic Logic Unit (ALU)**, che svolge le operazioni aritmetiche e logiche
 - **Registri**, ossia delle piccole memorie interne utilizzate per salvare dati temporanei
- **Memoria**: permette di memorizzare le **istruzioni** e i **dati** utili all'esecuzione dei **programmi** e al funzionamento generale del calcolatore
- **Periferiche di Input/Output**, che permettono al computer di comunicare con l'esterno
- **Bus di Sistema**, ossia un **canale unico di comunicazione** fra tutti i componenti, suddiviso in tre sotto-canali:
 - **Control Bus**, sul quale vengono comunicati i segnali di controllo che permettono ai componenti di coordinarsi
 - **Address Bus**, sul quale vengono comunicati gli indirizzi delle istruzioni da eseguire
 - **Data Bus**, sul quale vengono scambiati i dati all'interno del sistema



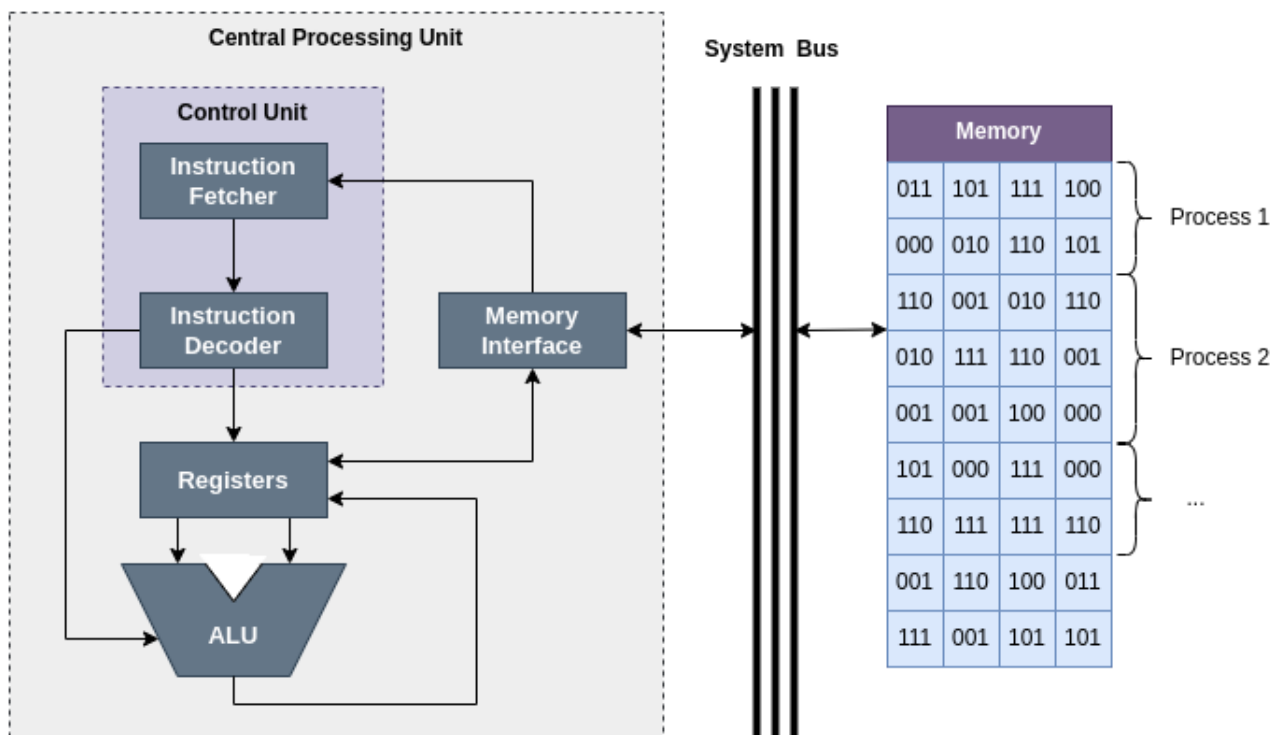
I primi modelli di computer (incluso quello di Neumann), erano progettati per eseguire un solo processo alla volta, mentre i moderni modelli sono provvisti di sistemi di **parallelismo**, permettendo la gestione di più processi in contemporanea che, attraverso un sistema di **scheduling**: una volta **eseguita** l'istruzione di un processo, esso viene momentaneamente **sospeso**, permettendo l'esecuzione dell'istruzione di un **secondo processo attivo**. Grazie all'estrema **rapidità** con cui la CPU esegue le istruzioni dei programmi, ripetere tale ciclo tra molti processi risulta nell'illusione di star eseguendo **più processi contemporaneamente**.

Per eseguire ogni istruzione, la CPU compie un **ciclo perenne** composto da tre fasi:

- **Fetch**, ossia la lettura della prossima istruzione
- **Decode**, ossia la decodifica dell'operazione da compiere
- **Execute**, ossia l'esecuzione dell'istruzione



Il **modello di Von Neumann** prevede che, prima di essere eseguiti, i programmi vengano **spostati nella memoria** per essere eseguiti. Quando un programma si trova nella memoria prende il nome di **processo**, ossia un programma in esecuzione. Per della loro natura stessa, ogni processo ha un effettivo **ciclo di vita**, poiché durante la loro esecuzione essi si evolvono raggiungendo vari **stati**.



1.3 L'Architettura MIPS 2000

In era moderna, possiamo individuare **due tipologie principali di architetture di calcolatori**:

- **Architettura CISC:**

- Acronimo di **Complex Instruction Set Computer**
- Le istruzioni sono di **dimensione variabile**, dunque per il fetch della successiva è necessaria prima la decodifica dell'istruzione stessa
- Gli operandi vengono effettuati in memoria, necessitando **molti accessi alla memoria** per ogni istruzione
- **Pochi registri interni**, dunque viene utilizzata la memoria anche per conservare i dati temporanei
- **Modi di indirizzamento più complessi** e con parziali conflitti tra le istruzioni più complesse, necessitando una pipeline più articolata

- **Architettura RISC:**

- Acronimo di **Reduced Instruction Set Computer**
- Le istruzioni sono di **dimensione fissa**, dunque non è necessario decodificarle prima del fetch della successiva
- Gli operandi vengono **effettuati dall'ALU e solo tra i registri**, dunque non è necessario accedere alla memoria
- **Molti registri interni**, dunque per risultati parziali non è necessario utilizzare la memoria
- **Modi di indirizzamento semplici** poiché ogni istruzione ha una dimensione fissa, dunque non si verificano conflitti

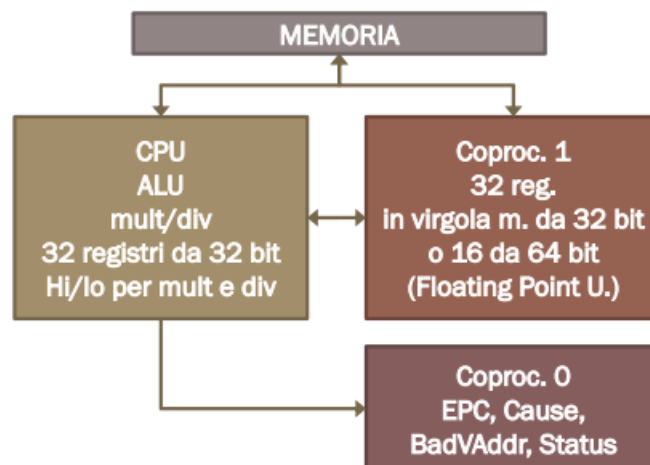
Riassumendo, possiamo dire che le **Architetture CISC** risultano più complesse ma ottimizzate per scopi singoli, mentre le **Architetture RISC**, in quanto più semplici, risultano adatte a scopi generici.

Per via delle sue caratteristiche, l'**Architettura MIPS**, acronimo di **Microprocessor without Interlocked Pipelined Stages**, risiede all'interno delle Architetture RISC.

In particolare, l'**Architettura MIPS 2000** è composta da:

- Tutte le **word hanno una dimensione fissa** di 32 bit
- Lo **spazio di indirizzamento** è di 2^{30} word di 32 bit ciascuna, per un totale di 4 GB
- Una **memoria indicizzata al byte**, dunque, dato un indirizzo di memoria t corrispondente all'inizio di una word, per leggere la word successiva è necessario utilizzare l'indirizzo $t + 4$, poiché 4 byte corrispondono a 32 bit (ricordiamo che ogni word è composta da 32 bit)
- Gli interi vengono salvati utilizzando la notazione del **Complemento a 2** su 32 bit

- Dotata di **3 microprocessori**:
 - La **CPU principale**, dotata di ALU, di 32 registri HI/LO ed addetta all'esecuzione delle istruzioni
 - Il **Coprocessore 0**, non è dotato di registri e non ha accesso alla memoria, ma è solo addetto alla gestione di "trap", eccezioni, Virtual Memory, Cause, EPC, Status, BadVAddr, ...
 - Il **Coprocessore 1**, addetto ai calcoli in virgola mobile e dotato di 32 registri da 32 bit, utilizzabili anche come 64 registri da 16 bit
- I **32 Registri della CPU principale**:
 - **Registro \$zero**, contenente un valore costante pari a 0 ed immutabile
 - **Registro \$at**, usato dalle pseudoistruzioni e dall'assemblatore
 - **Registri \$v0 e \$v1**, utilizzati per gli output delle procedure e funzioni utilizzate nel programma
 - **Registri dall'\$a0 all'\$a3**, utilizzati per gli input delle procedure e funzioni
 - **Registri dal \$t0 al \$t9**, utilizzati per memorizzare i valori temporanei, solitamente i risultati parziali di alcune operazioni
 - **Registri dal \$s0 al \$s7**, utilizzati per memorizzare i valori temporanei
 - **Registri \$k0 e \$k1**, utilizzati dal Kernel del Sistema Operativo, solitamente per le eccezioni e le interruzioni
 - **Registro \$gp**, ossia Global Pointer, utilizzato per la gestione della memoria dinamica
 - **Registro \$sp**, ossia Stack Pointer, utilizzato per la gestione dello Stack delle funzioni
 - **Registro \$fp**, ossia Frame Pointer, utilizzato dalle funzioni di sistema
 - **Registro \$ra**, ossia Return Address, utilizzato come puntatore di ritorno dalle funzioni



Capitolo 2

Il linguaggio Assembly MIPS

Come abbiamo visto, per impartire comandi ad un calcolatore, è necessario conoscere il suo linguaggio, in particolare le sue **istruzioni**. In queste sezioni, vedremo il "*vocabolario*" di un reale computer, sia in forma umanamente leggibile (**linguaggio assembly**), sia in forma meccanicamente leggibile (**linguaggio macchina**).

Nonostante tale concetto possa sembrare a prima vista complesso, è necessario ricordare che i computer sono macchine stupide in grado di eseguire **operazioni estremamente semplici** (sembrerà assurdo, ma in realtà quasi ogni istruzione corrisponde ad una somma tra valori) in modo estremamente veloce.

2.1 Formato delle istruzioni

Le istruzioni della CPU dell'architettura MIPS, seguono una struttura molto semplice:

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

Per comprendere meglio, vediamo direttamente un esempio pratico:

add \$s0, \$t0, \$t1

L'istruzione riportata corrisponde nient'altro che alle seguenti **tre operazioni**:

- **Leggi** il registro \$t0 e il registro \$t1 (sorgenti)
- **Somma** i loro valori
- **Scrivi** il risultato sul registro \$s0 (destinazione)

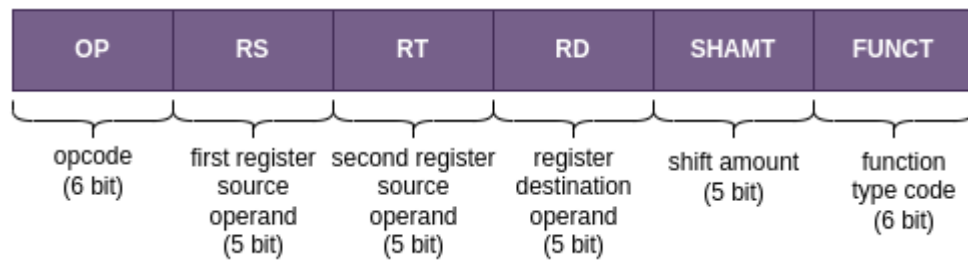
Tale struttura è solo una **generalizzazione** poiché, come vedremo in seguito, non è pienamente rispettata da ogni istruzione.

Ma non avevamo detto che **tutte le istruzioni** corrispondono ad una **word da 32 bit**? Come fanno ad avere una struttura variabile? Il motivo è semplice: ogni istruzione viene letta ed interpretata dall'assemblatore, il quale la tradurrà nel formato adeguato in **codice**

macchina. Dunque, ad avere formato fisso non sono le istruzioni in linguaggio assembly, bensì le istruzioni in codice macchina.

In particolare, tali istruzioni vengono tradotte dall'assemblatore in un **formato specifico** determinato dalla **tipologia stessa di istruzione**:

- **Istruzioni R-Type (tipo Registro):**
 - Senza accesso alla memoria
 - Istruzioni di tipo **aritmetico** e di tipo **logico**
 - Formato dei bit:



- Esempio:

Istruzione	OP	RS	RT	RD	SHAMT	FUNCT
add \$t0, \$s1, \$s2	000000	10001	10010	01000	00000	100000
sub \$t0, \$s1, \$s2	000000	10001	10010	01000	00000	100010

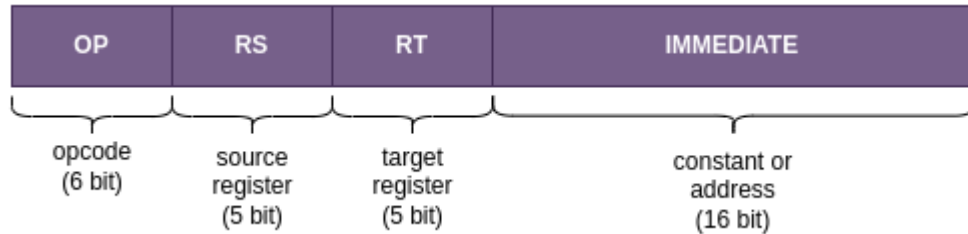
Analizziamo pezzo per pezzo il contenuto delle due istruzioni:

- * **Opcode (OP):** rappresenta la categoria di operazione da eseguire. In questo caso, 000000 indica un'operazione aritmetica.
- * **First register (RS):** rappresenta il primo registro sorgente da cui leggere il valore. In questo caso, 10001 corrisponde al registro \$s1
- * **Second register (RT):** rappresenta il secondo registro sorgente da cui leggere il valore. In questo caso, 10010 corrisponde al registro \$s2
- * **Destination register (RD):** rappresenta il registro su cui scrivere il risultato. In questo caso, 01000 corrisponde al registro \$t0
- * **Shift amount (SHAMT):** rappresenta la quantità di bit da shiftare. In questo caso vale 0 poiché non stiamo effettuando uno shift
- * **Function code (FUNCT):** rappresenta la specifica secondaria dell'operazione da eseguire, dunque corrisponde ad un'*estensione* dell'opcode. In questo caso, nella prima istruzione viene specificato che la tipologia di operazione aritmetica da eseguite è una somma, mentre nel secondo viene specificato di eseguite una sottrazione

Notiamo quindi come anche **solo 2 bit invertiti** possano corrispondere ad un'operazione totalmente diversa, motivo per cui utilizzare un assembler per programmare un calcolatore risulta **essenziale**.

- Istruzioni I-Type (tipo Immediato):

- Operazioni di **Load** e **Store**
- Utilizzate dai **salti condizionati** (ossia relativi al Program Counter)
- Formato dei bit:



- Esempio:

Istruzione	OP	RS	RT	IMMEDIATE
addi \$t2, \$s2, 17	00100	11010	01010	0000000000010001

- * **Opcode (OP)**: viene specificata l'operazione di addizione immediata, ossia non tra due registri ma tra un registro e un valore costante (immediato)
- * **Source register (RS)**: viene letto il valore del registro \$s2
- * **Destination register (RT)**: viene specificato di scrivere il risultato della somma nel registro \$t2
- * **Immediate**: viene specificato il valore costante con cui effettuare la somma immediata, in questo caso 17 (10001 in binario)

- Istruzioni J-Type (tipo Jump):

- Utilizzate dai **salti non condizionati** (ossia assoluti)
- Formato dei bit:



- Esempio:

Istruzione	OP	ADDRESS
j 2500	000010	00000000000010011100010000

- * **Opcode (OP)**: viene specificata l'operazione di jump incondizionato
- * **Address**: viene specificato l'indirizzo su cui effettuare il jump. Il valore indicato, in realtà, rappresenta $2500 \cdot 4$ (oppure $2500 \ll 2$), poiché ricordiamo che l'architettura MIPS è indicizzata al byte, dunque la 2500-esima parola corrisponde all'indirizzo 10000 della memoria

2.2 Lista delle 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)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una parola da memoria a registro
	Memorizzazione parola	sw \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di una parola da registro a memoria
	Lettura mezza parola	lh \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una mezza parola da memoria a registro
	Lettura mezza parola, senza segno	lhu \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una mezza parola da memoria a registro
	Memorizzazione mezza parola	sh \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di una mezza parola da registro a memoria
	Lettura byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di un byte da memoria a registro
	Lettura byte senza segno	lbu \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di un byte da memoria a registro
	Memorizzazione byte	sb \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di un byte da registro a memoria
	Lettura di una parola e blocco	ll \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Caricamento di una parola come prima fase di un'operazione atomica
	Memorizzazione condizionata di una parola	sc \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$; $\$s1=0$ 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	$\$s1 = \sim(\$s2 \$s3)$	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	$\$s1 = \$s2 \ll 10$	Spostamento a sinistra del numero di bit specificato dalla costante
	Scorrimento logico a destra	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	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 $(\$s1 \neq \$s2)$ 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
Salti incondizionati	Salto incondizionato	j 2500	Vai a 10000	Salto all'indirizzo della costante
	Salto indiretto	jr \$ra	Vai all'indirizzo contenuto in \$ra	Salto all'indirizzo contenuto nel registro, utilizzato per il ritorno da procedura e per i costrutti switch
	Salta e collega	jal 2500	$\$ra = \text{PC}+4$; vai a 10000	Chiamata a procedura

2.3 Organizzazione della Memoria

Come abbiamo già detto, nell'architettura MIPS la memoria è **indicizzata al byte**, dove ogni **word** è composta da **4 byte** (ossia 32 bit). A livello teorico, possiamo immaginare la memoria come una tabella composta da **4 colonne**, dove ogni colonna rappresenta un **byte**, e 2^{30} **righe**, dove ogni riga rappresenta una **word**. Dunque, possiamo dire che la memoria è composta da un totale di $4 \text{ Byte} \cdot 2^{30} \text{ Word} = 4 \text{ GigaByte}$.

Ad ogni byte della memoria è associato un **indirizzo**, rappresentato da un **8 cifre esadecimali**, poiché ricordiamo che ogni cifra esadecimale corrisponde esattamente a 4 bit, dunque due cifre esadecimali corrispondono ad un byte.

Memory

1st Word	0x00000000	0x00000001	0x00000002	0x00000003
2nd Word	0x00000004	0x00000005	0x00000006	0x00000007

2^{30}th Word	0xFFFFFFFFC	0xFFFFFFFDD	0xFFFFFFFEE	0xFFFFFFFFF

***Attenzione:** ogni cella del seguente disegno corrisponde ad un byte di memoria, mentre il valore esadecimale ad esso associato corrisponde al suo indirizzo*

Poiché ogni word corrisponde a 4 byte, ogni word risulta **indicizzata con uno sfalzo di 4 byte** (la prima word sarà all'indirizzo 0x00000000, la seconda all'indirizzo 0x00000004, ...).

Generalizzando il tutto, possiamo dire che il **k-esimo** byte si trova all'indirizzo di memoria $(k - 1)$, mentre la **j-esima** word si trova all'indirizzo $4 \cdot (j - 1)$.

Dunque, se volessimo leggere il contenuto della 1000esima word, l'indirizzo di memoria corrispondente sarebbe

$$M = 4 \cdot (1000 - 1) = 3996_{10} = 0x00000F9C$$

Nel linguaggio assembly MIPS, per leggere il contenuto di una word in memoria viene utilizzata la seguente **notazione**:

offset(\$indirizzo)

Dove **\$indirizzo** corrisponde ad un registro all'interno del quale è stato caricato un **valore**, il quale verrà interpretato come l'indirizzo di memoria da cui prelevare la word, mentre l'**offset** corrisponde al **numero di byte successivi** all'indirizzo indicato.

Ad esempio, immaginiamo la seguente catena di istruzioni

```
li $t0, 3996      //carico in $t0 il valore 3996 all'interno di t0

lw $t1, 0($t0)    //carico in $t1 la word all'indirizzo $t0

lw $t1, 4($t0)    //carico in $t1 la word all'indirizzo ($t0 + 4 byte)
```

Nella prima istruzione, viene utilizzato il comando **Load Immediate**, che permette di **caricare un valore immediatamente** (dunque senza leggere il valore da un altro registro).

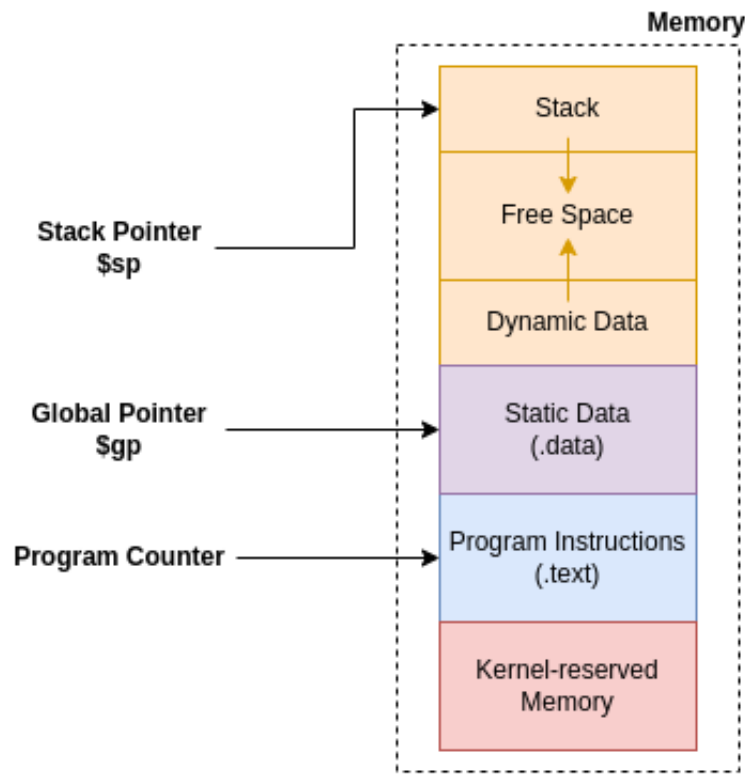
Successivamente, viene utilizzato due volte il comando **Load Word**, che permette **caricare un'intera word** all'interno del registro indicato (ricordiamo che sia la word e sia il registro sono formati da 32 bit).

- Nel primo utilizzo, viene prelevata dalla memoria la **word** di indirizzo corrispondente al **valore caricato nel registro \$t0**, poiché l'**offset definito è 0**, leggendo quindi la 1000-esima word in memoria, poiché l'indirizzo indicato è 3996 (0x00000F9C).
- Nel secondo utilizzo viene prelevata la word di **indirizzo \$t0 + 4 byte**, poiché l'**offset definito è 4**, leggendo quindi **òa parola direttamente successiva** a quella indicata da \$t0, ossia la 1001-esima parola, corrispondente all'indirizzo 4000 (0x00000FA0)

L'utilizzo dei registri come "puntatore" degli indirizzi di memoria permette un utilizzo estremamente **flessibile** della memoria stessa, evitando al programmatore di poter compiere operazioni sui valori stessi corrispondenti ad un indirizzo di memoria, senza dover scrivere ogni volta manualmente l'indirizzo che si vuole leggere. In seguito, esploreremo maggiormente tale concetto.

2.3.1 Parti della memoria

Una volta compreso il modo in cui viene indicizzata la memoria, possiamo vedere la sua **struttura nel completo**, definendone quelle che sono le **parti principali**.



- **Stack:** viene utilizzata per operazioni relative alle funzioni (o procedure), salvandone le chiamate ricorsive e le variabili locali. Non ha una dimensione fissa, dunque si può espandere nella **sezione di memoria libera condivisa**. Il registro \$sp, ossia **Stack Pointer**, viene utilizzato per operare all'interno di tale zona di memoria
- **Dynamic Data (o Heap):** contiene tutti i dati dinamici che vengono immagazzinati durante l'esecuzione del programma. Anch'essa si può espandere nella **sezione di memoria libera condivisa**
- **Static Data:** contiene tutti i dati statici che vengono definiti all'avvio del programma (etichettate sotto la direttiva `.data`). Il registro \$gp, ossia **Global Pointer**, viene utilizzato dall'assemblatore stesso per gestire gli indicizzamenti all'interno di questa zona
- **Program Instructions:** contiene tutte le istruzioni del programma (etichettate sotto la direttiva `.text`). All'interno di tale zona opera il **Program Counter**, ossia il registro che memorizza la posizione in memoria dell'istruzione successiva da eseguire
- **Kernel-reserved:** corrisponde ad uno spazio di memoria **inutilizzabile** dal programmatore, poiché riservato al Kernel del Sistema Operativo. Tentare di accedere a tale zona di memoria risulterà in un'eccezione (ossia un "blocco" o "divieto") generata dal sistema operativo stesso.

2.4 Direttive principali ed Esempi di codice

Prima di vedere alcuni esempi di codice, è necessario discutere di quelle che sono le **direttive principali** di un codice in linguaggio assembly. Tali direttive non corrispondono in modo diretto ad una particolare istruzione in linguaggio macchina, bensì vengono **interpretate esclusivamente dall'assemblatore**, il quale si occuperà poi di **tradurre** il tutto in istruzioni più complesse.

Le direttive principali del linguaggio assembly MIPS sono:

- **.data**: utilizzata per definire dei dati statici
- **.text**: utilizzata per definire le istruzioni del programma
- **.ascii**: utilizzata per definire una stringa di caratteri terminata da un byte null, ossia "
0", indicante la fine della stringa stessa
- **.byte**: utilizzata per definire una sequenza di byte
- **.double**: utilizzata per definire una sequenza di valori double, ossia a doppia precisione
- **.float**: utilizzata per definire una sequenza di valori float, ossia a singola precisione
- **.half**: utilizzata per definire una sequenza di half word, ossia metà word
- **.word**: utilizzata per definire una sequenza di word

Vediamo ora un **primitivo esempio di codice assembly**:

```
.text

main:
    li $t0, 5
    li $t1, 0x10
    add $s0, $t0, $t1
```

In questo breve codice, abbiamo utilizzato la direttiva **.text**, indicante l'inizio delle istruzioni da eseguire, l'istruzione **Load Immediate**, per **caricare** il valore decimale 5 in \$t0 e il valore esadecimale 0x10 (corrispondente a 16 in decimale) in \$t1, per poi salvare la **somma** dei due in \$s0.

Notiamo però anche la presenza della linea di codice contenente **main:**, corrispondente ad un altro concetto fondamentale da introdurre, ossia il concetto di **label (etichetta)**, definita come **nome_etichetta:** (inclusi i due punti).

Le etichette vengono **poste accanto ad istruzioni (anche vuote) e dati statici** e svolgono una funzione di "segnalibro" per l'assemblatore, il quale, in fase di compilazione, andrà a **tradurre tali etichette con l'indirizzo di memoria corrispondente** all'istruzione o dato statico a cui essa è stata associata.

Per comprendere meglio l'uso delle direttive e delle etichette, vediamo subito un **esempio più articolato**:

```
.data

vettore: 10, 2, 0x12
stringa: .asciiz "Sono una stringa"
vettore_float: .float 10.2, 3.33333

.text

main:
    la $s0, vettore      //carico in $s0 l'indirizzo del dato "vettore"

    lw $s1, 0($s0)
    lw $s2, 4($s0)
    lw $s3, 8($s0)

    add $t0, $s1, $s2    // $t0 = $s1 + $s2 ossia $t0 = 10 + 2
    sub $t0, $t0, $s3    // $t0 = $t0 + $s3 ossia $t0 = 12 - 18
```

Analizziamo pezzo per pezzo tale codice:

1. Vengono definiti dei **dati statici** sotto la direttiva **.data**. In particolare, viene definito un vettore di interi (indicabili sia in decimale sia in esadecimale), una stringa di caratteri ed un vettore di valori float.
2. Viene utilizzata la direttiva **.text**, indicando l'inizio delle istruzioni del programma
3. Viene usato il comando **Load Address**, che carica in \$s0 l'indirizzo di memoria associato all'etichetta "vettore"
4. Vengono caricati tutti i valori del vettore utilizzando il comando Load Word. Da tali istruzioni, possiamo notare come un **vettore di valori** (indipendentemente dal tipo) corrisponda esattamente ad un insieme di word messe una di fila all'altra, dunque distanti 4 byte ciascuna in memoria. Lo stesso discorso si applica anche per le **stringhe**, poiché esse non sono nient'altro che un vettore di caratteri.
5. Vengono svolte operazioni numeriche tra i registri in cui sono stati caricati i valori del vettore

Notiamo quindi l'estrema comodità dell'utilizzo delle **direttive** e delle **etichette**, permettendoci di utilizzare ed accedere in maniera facile ai dati statici presenti in memoria.

2.5 Salti condizionati e Salti assoluti

Introduciamo ora quello che è un concetto fondamentale e alla base dello sviluppo di ogni programma articolato, ossia i **salti condizionati e assoluti**.

In generale, con il termine "**salto**" si intende un'operazione che va a **modificare il Program Counter**, cambiando l'indirizzo di memoria contenuto al suo interno. Ricordiamo che il PC si occupa di tenere traccia dell'indirizzo di memoria della **prossima istruzione da eseguire**, dunque andando ad operare su tale indirizzo possiamo "spostarci" all'interno del programma, cambiandone il **flusso delle istruzioni**.

- **Salti assoluti:** non appena viene raggiunta l'istruzione di salto assoluto, il PC verrà modificato con l'indirizzo definito all'interno dell'istruzione.

```
.text

main:
    li $t0, 0           //carico 0 in $t0
loop:
    addi $t0, $t0, 1     //aggiungo uno a $t0
    j loop              //salto alla label "loop"
```

Nell'esempio superiore, una volta raggiunta l'istruzione "j loop" il PC verrà aggiornato con l'indirizzo di memoria corrispondente alla **label "loop"** (ricordiamo che le label sono sostanzialmente solo un "**segnalibro**" di un **indirizzo di memoria**).

Dunque, la prossima istruzione che verrà eseguita sarà "addi \$t0, \$t0, 1, per poi procedere con il normale flusso del programma, dunque eseguendo tutte le istruzioni successive (in questo caso verrà eseguita nuovamente l'istruzione "j loop", andando quindi a creare un **loop infinito**).

- **Salti condizionati:** seguono la stessa logica dei salti assoluti, ma con l'aggiunta di un **controllo logico di una condizione**. Ne esistono varie **tipologie**, ognuna associata ad un'istruzione diversa:

- **Branch on Equal:** il salto viene effettuato se e solo se il valore contenuto in \$s1 è **uguale** al valore contenuto in \$s2

```
beq $s1,$s2, label
```

- **Branch on Not Equal:** il salto viene effettuato se e solo se il valore contenuto in \$s1 **non è uguale** al valore contenuto in \$s2

```
bne $s1,$s2, label
```

- **Branch on Less Than or Equal Zero:** il salto viene effettuato se e solo se il valore contenuto in \$s1 **minore o uguale a zero**

```
blez $s1, label
```

- **Branch on Greater Than or Equal Zero:** il salto viene effettuato se e solo se il valore contenuto in \$s1 **maggiore o uguale a zero**

bgez \$s1, label

- **Branch on Less Than Zero:** il salto viene effettuato se e solo se il valore contenuto in \$s1 **minore di zero**

bltz \$s1, label

- **Branch on Greater Than Zero:** il salto viene effettuato se e solo se il valore contenuto in \$s1 **maggiore di zero**

bgtz \$s1, label

Vediamo ora un esempio di uso dei salti condizionati:

```
.text

main:
    li $t0, 0           //carico 0 in $t0
    li $t1, 100         //carico 100 in $t1
loop:
    addi $t0, $t0, 1     //aggiungo uno a $t0
    bne $t0, $t1, loop   //salto se e solo se $t0 == $t1
```

Al contrario del codice precedente, questa volta l'istruzione di salto è stata sostituita con un **Branch on Not Equal**, dove vengono comparati i registri \$t0 e \$t1. Il loop verrà quindi ripetuto **finché \$t0 e \$t1 non conterranno lo stesso valore**, ossia 100.

Notiamo tuttavia l'assenza di alcune istruzioni che potrebbero essere comode, ad esempio un ipotetico "Branch if Less Than", dove viene eseguito il controllo $\$t0 > \$t1$. L'assenza di tali istruzioni è dovuta al concetto base della progettazione dei circuiti digitali, ossia **ridurre al minimo possibile il numero di componenti**.

Difatti, l'assenza di tale condizione di salto è dovuta all'esistenza di un'istruzione simile, ossia l'istruzione **Set if Less Than**:

slt \$t0, \$s0, \$s1

In questo caso, tale istruzione modifica il valore contenuto nel registro \$t0 in base al risultato del controllo condizionale $\$s0 < \$s1$, modificandolo in **1 se il risultato è vero** o in **0 se il risultato è falso**. Dunque, se volessimo eseguire la nostra ipotetica istruzione "Branch if Less Than", dovremmo eseguire la seguente catena di istruzioni

```
    slt $t0, $s0, $s1     // $t0 = 1 se e solo se $s0 < $s1
    bne $t0, $zero, Check //salto solo se $t0 != 0
```

Esempio complesso di uso dei salti

Vediamo adesso un esempio molto più complesso di codice rispetto ai precedenti visti fino ad ora. Si tratta di un programma che cerca il valore massimo all'interno di un vettore di 4 valori

```
.data

values: 10, 13, 99, 9
maxValue: 0

.text

main:
    lw $s0, values           //carico values[0] in $s0
    lw $s1, values+4         //carico values[1] in $s1
    lw $s2, values+8         //carico values[2] in $s2
    lw $s3, values+12        //carico values[3] in $s3

CopyA: move $t0, $s0         //copio $s0 in $t0

CheckB: slt $t1, $t0, $s1    //metto 1 in $t1 se $t0 < $s1
        beq $t1, $zero, CheckC //salto a CheckC se $t1 == 0

        move $t0, $s1        //copio $s1 in $t0

CheckC: slt $t1, $t0, $s2    //metto 1 in $t1 se $t0 < $s2
        beq $t1, $zero, CheckD //salto a CheckD se $t1 == 0

        move $t0, $s2        //copio $s2 in $t0

CheckD: slt $t1, $t0, $s3    //metto 1 in $t1 se $t0 < $s3
        beq $t1, $zero, End  //salto a End se $t1 == 0

        move $t0, $s3        //copio $s3 in $t0

End:    sw $t0, maxValue     //salvo $t0 in memoria
```

Notiamo come in questo esempio siano stati usati i salti condizionati come metodo per **evitare di eseguire alcune istruzioni** e non come metodo per eseguire dei loop. Questa funzione evidenzia notevolmente l'**estrema flessibilità** delle istruzioni di salto.

Tuttavia, notiamo come, nonostante si tratti di un semplice programma che cerca il massimo tra 4 valori, il codice risulti **lungo e ridondante**. Possiamo provare a **migliorare** tale codice sfruttando un uso migliore dei registri e dei salti condizionati.

Possiamo realizzare la versione migliorata del programma salvando nel registro \$s0 il valore dell'**indirizzo in memoria** corrispondente al vettore "values" tramite l'istruzione Load Address, in modo da poter operare **direttamente su tale registro** per poter accedere agli altri valori del vettore, senza dover usare più volte l'istruzione Load Immediate.

Successivamente, andremo a caricare in \$s1 il valore 3, corrispondente a *LunghezzaVettore* - 1, per poi andare a generare un **ciclo condizionato** che verrà terminato solo quando il valore in \$s1 avrà raggiunto zero, venendo decrementato ad ogni iterazione del ciclo.

```
.data

values: 10, 13, 99, 1000
maxValue: 0

.text

main:
    la $s0, values           //carico l'indirizzo in $s0
    li $s1, 3                //carico 3 in $s1

    lw $s2, 0($s0)           //carico MEM[$s0+0] in $s2

CheckNext:
    subi $s1, $s1, 1         //decremento il contatore
    addi $s0, $s0, 4         //incremento di 4 l'indirizzo

    lw $t0, 0($s0)           //carico il nuovo MEM[$s0+0] in $s2

    slt $t1, $s2, $t0
    beq $t1, $zero, CheckEnd //salto solo se $s2 > $t0

    move $s2, $t0            //altrimenti copio $s2 in $s2

CheckEnd:
    bne $s1, $zero, CheckNext //salto se il contatore non è zero

    sw $s2, maxValue         //salvo il massimo trovato
```

Sebbene la prima versione del codice risulti leggermente **più leggibile ed intuitiva**, la nuova versione risulta **più compatta e generalizzata**: se volessimo trovare il massimo tra 5, 7 o 100 numeri, ci basterebbe modificare il valore iniziale di \$s1 ed aggiungere dei numeri in coda al vettore "values".

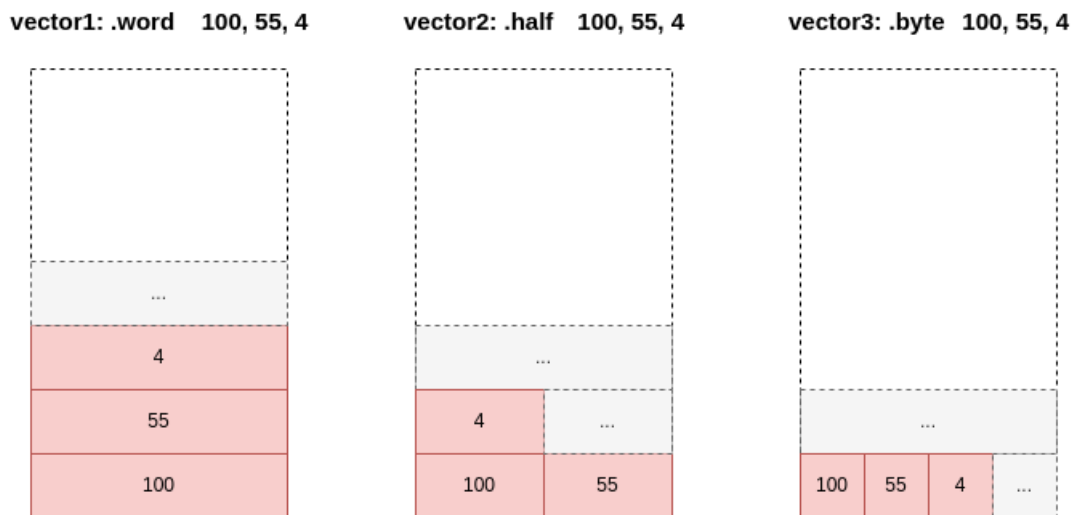
2.6 Vettori e Matrici

Nei codici precedenti, abbiamo già introdotto il concetto di **vettore**, ossia una **collezione di elementi della stessa dimensione posti in memoria uno dopo l'altro**. Tale concetto, seppur banale, è **fondamentale** nella gestione della memoria in contesti come cicli o ottimizzazioni di codice.

Consideriamo i seguenti due vettori:

```
vector1: .word 100, 55, 4      (o anche solo vector1: 100, 55, 4)
vector2: .half 100, 55, 4
vector3: .byte 100, 55, 4
```

L'unica **differenza** tra i tre vettori, in questo caso, è la **dimensione dei loro elementi**. All'interno della memoria, quindi, i tre vettori saranno conservati nel seguente modo:



Tale differenza implica anche una **gestione diversa** all'interno del codice assembly:

```
.data

vector1: .word 100, 55, 4
vector2: .half 100, 55, 4
vector3: .byte 100, 55, 4

.text

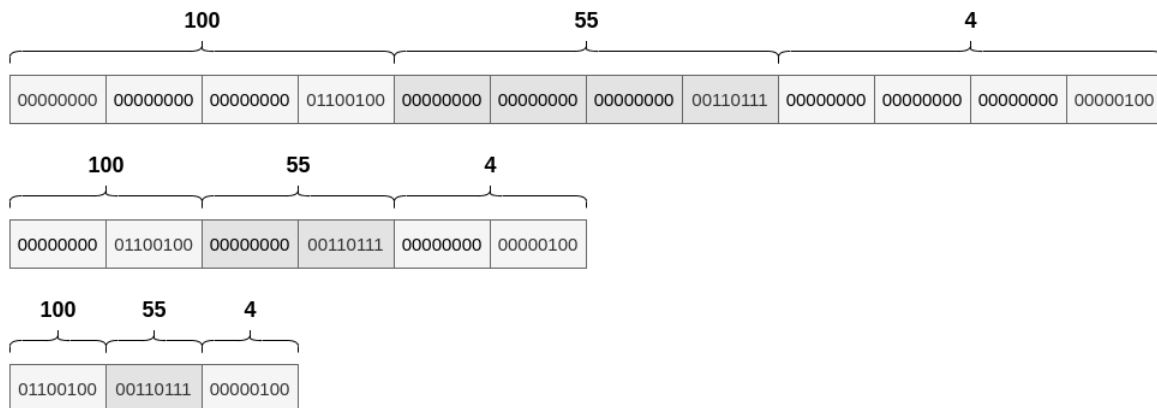
main:
    li $s0, vector1          //$s0 = vector1[0]
    li $s1, vector1 + 4      //$s1 = vector1[1]

    li $s0, vector2          //$s0 = vector2[0]
    li $s1, vector2 + 2      //$s1 = vector2[1]

    li $s0, vector3          //$s0 = vector2[0]
    li $s1, vector3 + 1      //$s1 = vector2[1]
```

Notiamo come, per via dell'**indicizzazione al byte** dell'architettura MIPS, per accedere al secondo elemento di un **vettore di half word** è necessario incrementare l'indirizzo in memoria di 2 byte, mentre nel caso di un **vettore di byte** è necessario incrementarlo di 1 byte.

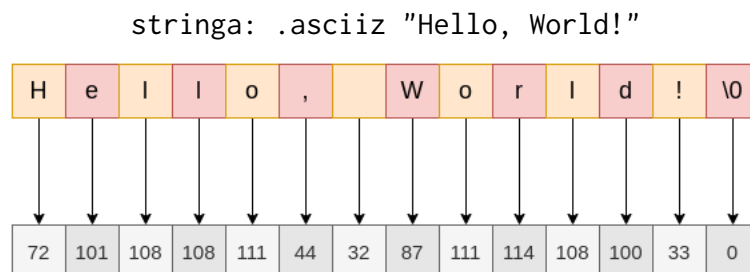
Convertendo il tutto in termini di bit, notiamo come i tre vettori risultano **molto differenti**, nonostante essi stiano svolgendo la stessa identica funzione, ossia conservare i tre valori 100, 55 e 4, occupando però **quantità** molto differenti di memoria, ad esempio nel caso del primo vettore stiamo utilizzando il **quadruplo dello spazio necessario**, sprecando molta memoria.



Estendiamo ora il concetto di vettore al mondo delle **stringhe di caratteri**. Come sappiamo, **ogni singolo dato**, indipendentemente dalla sua forma, complessità ed utilizzo, all'interno della memoria e dei registri deve essere rappresentato sottoforma di **insieme di bit**.

Lo stesso vale anche per i **caratteri alfanumerici**, difatti ogni carattere viene codificato in un **valore binario di 8 bit** (utilizzando la famosa **codifica ASCII**). Dunque, ogni carattere corrisponde esattamente ad un byte.

Una volta precisato ciò, è facile intuire come una **stringa di testo**, ossia una frase, parola o un qualsiasi insieme di più caratteri, non sia nient'altro che un **vettore di caratteri**, dove ogni carattere corrisponde in realtà al suo **valore intero** rispettivo della codifica ASCII.



Notiamo la presenza di un **carattere aggiuntivo** alla fine della stringa, ossia **0**, chiamato **Null Byte**. Tale carattere **viene aggiunto alla fine di ogni stringa** di caratteri per indicare la fine della stringa stessa, evitando che i valori in memoria che la seguono vengano interpretati anche essi come caratteri della stringa.

Accedere ai vettori

Principalmente, per accedere agli elementi dei vettori vi sono due modi:

- **Accesso tramite puntatore:** L'indirizzo in memoria viene caricato in un registro che farà da **puntatore** all'indirizzo di memoria degli elementi del vettore, permettendo di raggiungere gli elementi del vettore **modificando il valore del puntatore**.

Per raggiungere il **k-esimo elemento del vettore**, l'indirizzo del puntatore dovrà valere

$$\text{indirizzo_k_esimo_elem} = \text{indirizzo_base_vett} + k \cdot \text{dim_elementi}$$

Esempio:

```
.data
vector: 10, 123, 33

.text
main:
    la $s0, vector           //carico l'indirizzo del vettore

    li $t0, 2                //carico il valore di k
    sll $t1, $t0, 2          //$t1 = $t0 << 2 (molt. per 4)

    add $s0, $s0, $t1        //$s0 = $s0 + $t1
```

Nota: in questo esempio è stato usato uno shift sinistro di 2 bit per effettuare la moltiplicazione $k \cdot 4$

- **Accesso tramite indice:** Nel caso in cui il vettore sia stato creato come **dato statico** (quindi sotto `.data`) e non durante l'esecuzione del programma, è possibile accedervi usando direttamente un valore come **indice del vettore**.

Esempio

```
.data
vector: 10, 123, 33

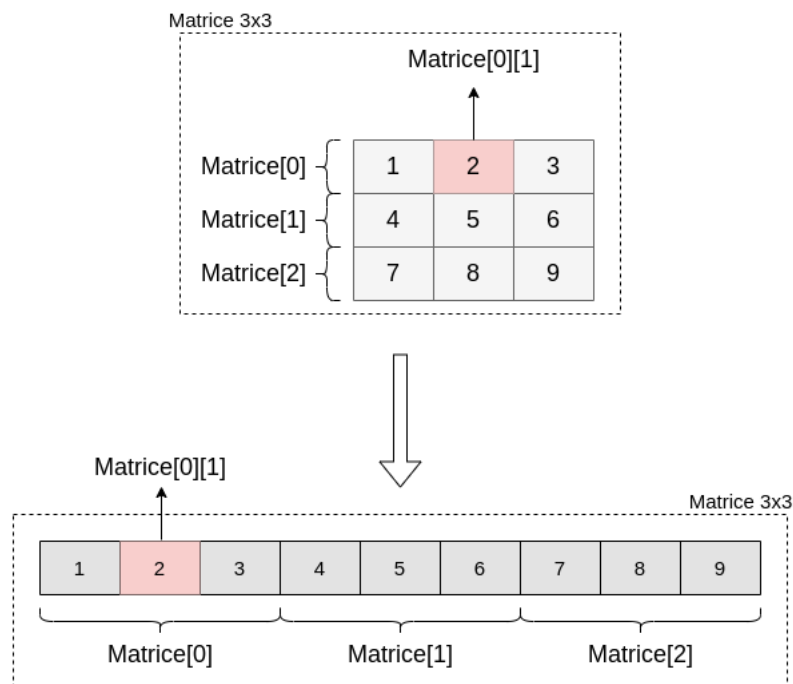
.text
main:
    li $t0, 2                //carico il valore di k
    sll $t1, $t0, 2          //$t1 = $t0 << 2 (molt. per 4)

    lw $s0, vector($t1)      //leggo dall'indirizzo vector+$t1
```

Matrici: vettori di vettori

Un modo molto semplice per comprendere il funzionamento delle matrici, è immaginare un **vettore di M elementi** dove ogni elemento è un **vettore di N elementi**. Possiamo quindi fare alcune assunzioni:

- Il numero totale di elementi è $M \cdot N$
- La dimensione totale in byte è $M \cdot N \cdot \text{dim_elem}$
- Poiché si tratta di un vettore di vettori, all'interno della memoria verrà immagazzinato come una serie composta da M serie composte a loro volta da N elementi, risultando quindi in un'**unica serie di $M \cdot N$ elementi adiacenti**



Difatti, nel linguaggio assembly (a differenza degli altri linguaggi più ad alto livello) non ci sono modi per specificare il numero di colonne e di righe di una matrice, bensì essa viene definita come un unico vettore di grandi dimensioni:

```
.data
matrix: .word 0:36      //matrice di 36 elementi
```

Nell'esempio qui sopra abbiamo creato una **matrice di 36 elementi**, dove ogni elemento viene inizializzato con il valore di default, ossia zero. Essendo una matrice di 36 elementi, **starà a noi decretare che tipo di matrice essa sia**, gestendo di conseguenza il programma, poiché:

- $4 \cdot 9 = 36$
- $12 \cdot 3 = 36$
- $6 \cdot 6 = 36$
- ...

2.7 System Calls

Con il termine **System Call** (ossia Chiamata al Sistema, abbreviato come **syscall**), si intende un **set di servizi complessi** messi a disposizione del programmatore da parte del **Kernel del Sistema Operativo stesso**. L'esempio tipico di una syscall è la **stampa su terminale** (print) di un valore numerico o una stringa.

Ogni sistema operativo gestisce le proprie syscall in modo diverso. Generalmente, ogni architettura, MIPS inclusa, segue una struttura del seguente formato:

- **Input:**
 - **Registro \$v0**, al cui interno viene inserito il **codice della syscall** che si vuole richiedere
 - **Registri \$a0, \$a1, \$a2, \$f0**, dove vengono inseriti eventuali parametri aggiuntivi che verranno letti dalla syscall
- **Output:**
 - **Registri \$v0 e \$f0**, al cui interno vengono restituiti eventuali valori dalla syscall stessa

Hello, World!

La syscall che utilizzeremo di più sarà sicuramente il **print di una stringa o valore**. Vediamo come viene implementato in ASM MIPS il classico programma che stampa la stringa "Hello, World!":

```
.data
stringa: .asciiz "Hello, World!"

.text
main:
    la $a0, stringa    //carico l'indirizzo della stringa in $a0

    li $v0, 4          //carico il valore 4 in $v0
    syscall            //eseguo la syscall
```

Analizziamo il programma: abbiamo definito in memoria statica la stringa "Hello, World!", per poi andare a caricare l'indirizzo di tale stringa all'interno del **registro \$a0**, che ricordiamo essere uno dei registri in cui vengono **passati parametri aggiuntivi** che verranno letti dalle syscall.

Successivamente, abbiamo caricato il valore 4 nel **registro \$v0**, che ricordiamo essere il registro il cui valore definisce il **tipo di syscall** che verrà eseguita (in questo caso, 4 corrisponde al codice del servizio "**read_string**"). Infine, effettuiamo la richiesta al sistema operativo tramite l'istruzione syscall.

Elenco delle Syscall in MIPS

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$v0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

2.7.1 Pseudoistruzioni

Le **pseudoistruzioni** sono istruzioni "fittizie" utilizzabili nel linguaggio assembly MIPS ma che tuttavia non sono implementate a livello hardware. Tali pseudoistruzioni vengono **tradotte dall'assembler in una sequenza di istruzioni realmente implementate nella CPU**. Esse, quindi, risultano essere per lo più una **comodità per il programmatore**, permettendogli di scrivere del codice più compatto e leggibile.

Nella sezione 2.5, abbiamo già discusso di come **non esista** un'istruzione "Branch if Less Than", costringendoci a dover utilizzare due istruzioni (`slt` e `bne`) per svolgere un controllo del tipo $\$s0 < \$s1$.

Tuttavia, ciò è in parte falso, poiché in realtà **tale istruzione esiste sottoforma di pseudoistruzione**, ossia `blt`, la quale viene tradotta dall'assembler nelle due istruzioni `slt` e `bne`.

Address	Code	Basic	
4194304	0x24110064	<code>addiu \$17,\$0,100</code>	5: <code>li \$s1, 100</code>
4194308	0x241200c8	<code>addiu \$18,\$0,200</code>	6: <code>li \$s2, 200</code>
4194312	0x00129821	<code>addu \$19,\$0,\$18</code>	8: <code>move \$s3, \$s2</code>
4194316	0x0233082a	<code>slt \$1,\$17,\$19</code>	10: <code>blt \$s1, \$s3, subroutine</code>
4194320	0x14200000	<code>bne \$1,\$0,0</code>	

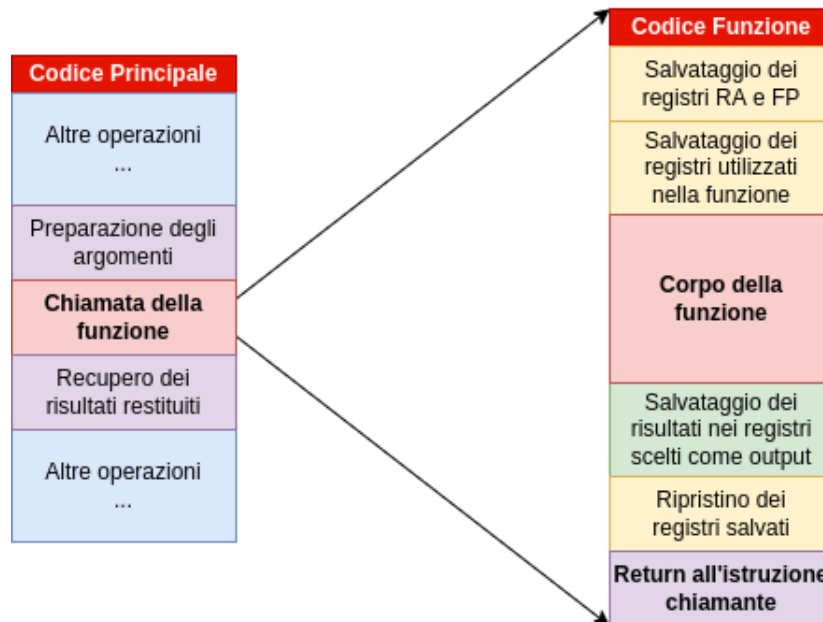
Notiamo, inoltre, come anche le istruzioni `li` e `move` siano in realtà **pseudoistruzioni**, nonché l'uso del **registro di appoggio \$1**, corrispondente al **registro \$at**.

2.8 Funzioni e Procedure

Come negli altri linguaggi di programmazione, una **funzione** (o procedura) è un **frammento di codice che riceve degli argomenti e calcola un risultato**. Esse sono utili per rendere il codice riusabile e modulare.

Una funzione è strutturata :

- Possiede un **indirizzo di partenza**
- Legge uno o più registri scelti come **argomenti in input**
- Svolge un calcolo ed altre operazioni
- Carica il risultato delle operazioni in uno o più registri scelti come **output**
- Una volta terminata, **ritorna all'istruzione da cui è stata chiamata**, riprendendo l'esecuzione del codice principale



Per **richiamare una funzione** viene utilizzata l'istruzione `jal <label>`, ossia **Jump And Link**, la quale, prima di effettuare il salto incondizionato, salva nel registro `$ra` l'indirizzo di memoria dell'istruzione successiva (dunque `$ra <- PC+4`).

Una volta terminata la funzione, è possibile **tornare all'istruzione chiamante** (ossia `jal`) tramite l'istruzione `jr <registro>`, ossia **Jump to Register**. Poiché l'indirizzo dell'istruzione successiva alla chiamante è contenuto nel registro `$ra`, eseguendo `jr $ra` torneremo ad eseguire il codice principale.

Nell'**uso dei registri all'interno delle funzioni**, vi sono alcune convenzioni, tra cui:

- **Registri di input** (`$a0`, `$a1`, `$a2`, `$a3`): usati come argomenti della funzione
- **Registri di output** (`$v0`, `$v1`): usati per restituire i risultati della funzione
- **Registri temporanei** (`$t0`, `$t1`, ...): possono cambiare tra una chiamata e l'altra
- **Registri salvati** (`$s0`, `$s1`, ...): non cambiano tra una chiamata e l'altra

Vediamo ora come realizzare una funzione `somma_con_quadrato` in grado di addizionare due numeri ed una funzione `stampa_intero` in grado di stampare un intero:

```
.text

main:
    li $a0, 5           //carico il primo argomento della funzione
    li $a1, 7           //carico il secondo argomento della funzione

    jal somma_con_quadrato    //eseguo il salto alla funzione

    move $a0, $v0         //sposto in $a0 il risultato della somma

    jal stampa_intero    //stampo il risultato

    // eseguo la syscall che termina il programma
    li $v0, 10
    syscall

somma_con_quadrato:
    // Input/Output della funzione:
    //      somma(int $a0, int $a1) => int $v0
    // Funzionamento:
    //      Somma $a0 con il quadrato di $a1

    mult $t0, $a1, $a1    //$t0 = $a1 * $a1
    add $v0, $a0, $t0     //uso $v0 come registro di output

    jr $ra$              //ritorno all'indirizzo chiamante

stampa_intero:
    // Input/Output della funzione:
    //      stampa_intero(int $a0) => null
    // Funzionamento:
    //      Stampa il contenuto di $a0

    //stampo il valore contenuto in $a0
    li $v0, 1
    syscall

    jr $ra               //ritorno all'indirizzo chiamante
```

Tuttavia, tale struttura di implementazione delle funzioni presenta alcune **falle**: ogni funzione modifica il contenuto di alcuni registri, comportamento che potrebbe generare grandi problemi all'interno del codice, richiedendo di dover salvare in un ulteriore registro lo **stato precedente alla chiamata della funzione**.

Ad esempio, la funzione `somma_con_quadrato` modifica il contenuto del registro `$t0`, il quale potrebbe contenere dati utili, necessitando quindi di doverne salvare il valore in un

ulteriore registro temporaneo, per poi ripristinarlo una volta terminata la funzione.

Tale anomalia viene risolta tramite l'uso dello **stack di memoria**.

2.8.1 Stack di memoria

Come abbiamo visto nella sezione precedente, prima di poter effettuare una chiamata ad una funzione è necessario dover **preservare** il contenuto precedente dei **registri utilizzati** all'interno della funzione stessa, per poi **ripristinarlo** una volta terminata.

Effettuare tali operazioni fornisce anche la possibilità di poter **chiamare altre funzioni all'interno di una funzione stessa** tramite la conservazione dell'indirizzo contenuto nel **registro \$ra**, il quale altrimenti verrebbe sovrascritto una volta chiamata la seconda funzione.

- Viene salvato lo stato dei registri precedente alla prima funzione
 - Viene salvato lo stato dei registri precedente alla seconda funzione
 - * ...
 - Viene ripristinato lo stato dei registri precedente alla seconda funzione
- Viene ripristinato lo stato dei registri precedente alla prima funzione

Tale comportamento coincide con quello di una **pila** (o **Stack**), in cui viene aggiunto un elemento in cima ad essa (**push**) e viene rimosso l'elemento in cima ad essa (**pop**).

Per realizzare ciò, quindi, viene utilizzata una zona della memoria adibita a tale funzionamento, chiamata **stack di memoria** (sezione 2.3.1), la quale cresce verso il basso, tenendo traccia dell'ultimo elemento preservato nella funzione tramite il **registro \$sp**, ossia lo **stack pointer**.

Apertura e Chiusura dello Stack

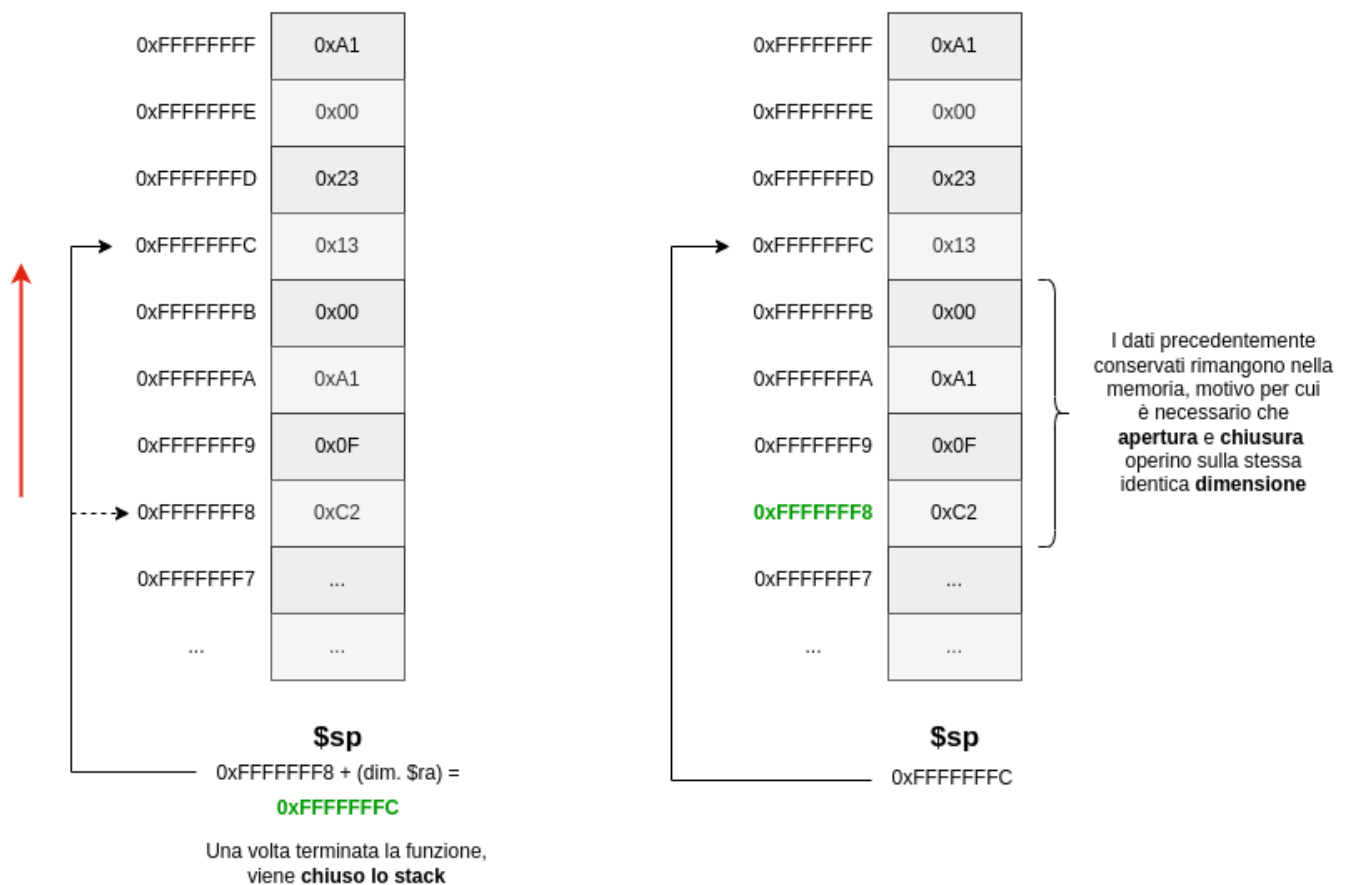
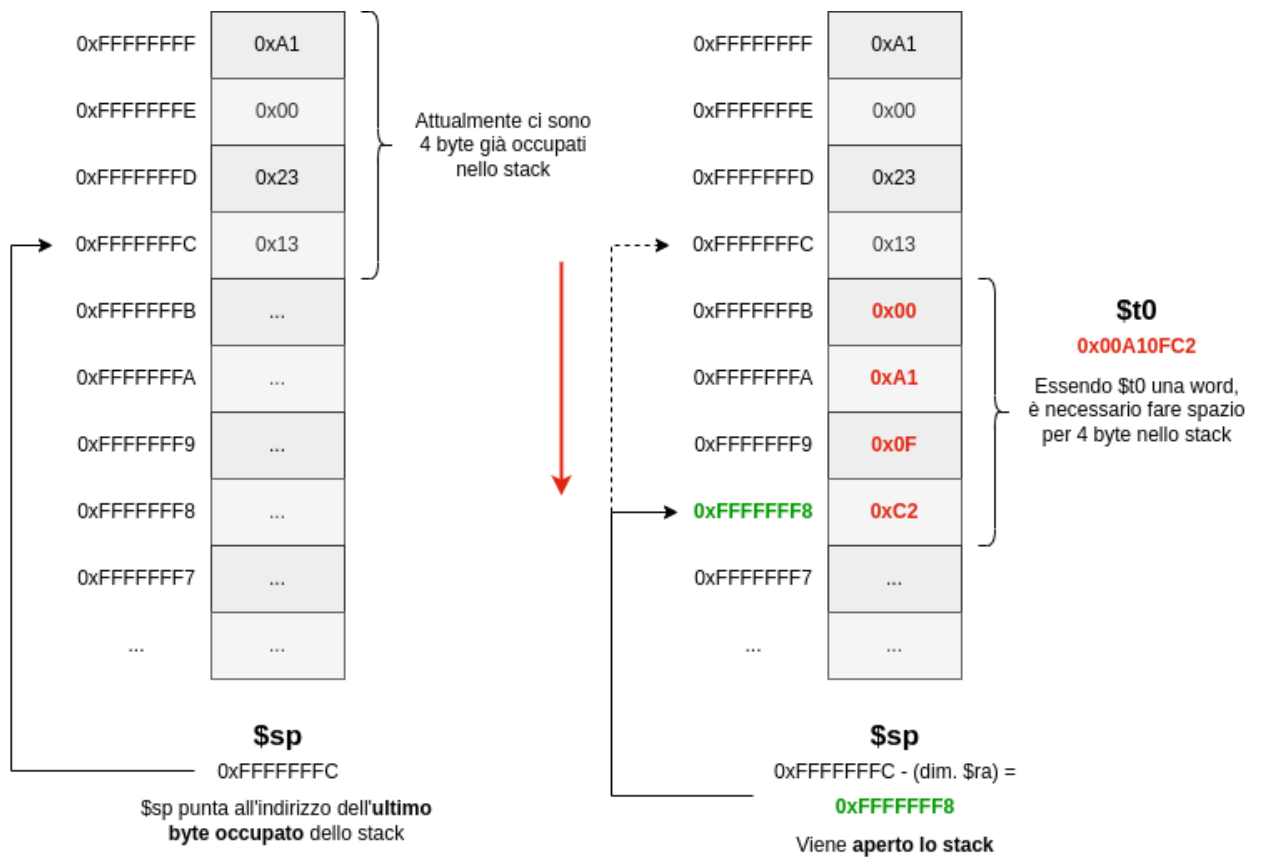
Immaginiamo di voler salvare nello stack il contenuto del registro \$t0, in modo da poterne modificare il contenuto all'interno di una funzione per poi ripristinarlo.

Poiché lo stack di memoria **cresce verso il basso** e poiché il registro \$sp deve puntare all'**ultimo elemento salvato** nello stack, è necessario **sottrarre a \$sp** la dimensione in byte dell'elemento che si vuole salvare, per poi andare a salvare in memoria l'elemento stesso a tale indirizzo puntato da \$sp (*apertura dello stack*). Una volta terminate le operazioni nella nostra funzione, possiamo ripristinare lo stato precedente dei registri eseguendo le **operazioni inverse** (*chiusura dello stack*).

```
subi $sp, $sp, 4    //faccio spazio per una word
sw $t0, 0($sp)      //salvo $t0 all'ind. in memoria puntato da $sp

// corpo della funzione

lw $t0, 0($sp)      //carico il valore precedente di $t0
addi $sp, $sp, 4     //rimuovo lo spazio per una word
```



L'insieme totale degli elementi da salvare nello stack viene detto **Stack Frame** (o Activation Record) ed è composto da:

- **Argomenti** passati alla funzione (contenuti in \$a0, ..., \$a3)
- **Indirizzo di ritorno** (contenuto in \$ra)
- **Frame pointer**, ossia l'indirizzo in memoria da cui parte lo stack frame (contenuto in \$fp). Spesso non viene salvato poiché ridondante o non necessario.
- **Registri utilizzati** all'interno della funzione (ad esempio \$t0, \$s0, ...)
- **Variabili locali** create nella funzione, in modo da essere "eliminate" una volta che quest'ultima si è chiusa



Una volta definito l'**uso corretto dello stack** come metodo di conservazione e ripristino dello stato precedente, riscriviamo nel modo corretto le due funzioni dell'esempio della sezione 2.8, salvando il valore dei registri \$t0 e \$v0, che altrimenti verrebbero alterati:

```
.text

main:
    li $a0, 5           //carico il primo argomento della funzione
    li $a1, 7           //carico il secondo argomento della funzione

    jal somma_con_quadrato //eseguo il salto alla funzione

    move $a0, $v0        //sposto in $a0 il risultato della somma

    jal stampa_intero     //stampo il risultato

    // eseguo la syscall che termina il programma
    li $v0, 10
    syscall
```

```
somma_con_quadrato:
    //somma(int $a0, int $a1) => int $v0

    //apertura dello stack
    subi $sp, $sp, 8
    sw $ra, 0($sp)
    sw $t0, 4($sp)

    mult $t0, $a1, $a1    //$t0 = $a1 * $a1
    add $v0, $a0, $t0    //uso $v0 come registro di output

    //chiusura dello stack
    lw $t0, 4($sp)
    lw $ra, 0($sp)
    addi $sp, $sp, 8

    jr $ra$

stampa_intero:
    //stampa_intero(int $a0) => null

    //apertura dello stack
    subi $sp, $sp, 8
    sw $ra, 0($sp)
    sw $v0, 4($sp)

    li $v0, 1
    syscall

    //chiusura dello stack
    lw $v0, 4($sp)
    lw $ra, 0($sp)
    addi $sp, $sp, 8

    jr $ra
```

2.8.2 Funzioni ricorsive

Come in ogni altro linguaggio di programmazione, in alcuni casi risulta più efficace sviluppare una versione **ricorsiva** di un programma per via della natura stessa del problema (esempio tipico: il calcolo di un numero di Fibonacci). Una volta in grado di realizzare funzioni nel modo corretto anche in Assembly MIPS, l'implementazione di una funzione ricorsiva risulta essere **analogo** all'implementazione di una soluzione ricorsiva in **qualsiasi altro linguaggio di programmazione**.

Esempio - Fibonacci Ricorsivo

```
.text

main:
    li $a0, 3           //dichiarazione argomenti della funzione
    jal Fibonacci       //chiamata alla funzione

    move $a0,$v0        //printa risultato
    li $v0,1
    syscall

    li $v0, 10          //chiudi il programma
    syscall

Fibonacci:
    // Fib(int $a0) => int $v0
    // -- Se $a0 == 0, allora $v0 = 0
    // -- Se $a0 == 1, allora $v0 = 1
    // -- Se $a0 > 1, allora $v0 = Fib($a0 - 1) + Fib($a0 - 2)

    beq $a0, 0, BaseCase_0    // Se $a0 == 0

    blt $a0, 1, BaseCase_1    // Se $a0 == 1

    j RecursiveStep           // Else

BaseCase_0:
    li $v0, 0
    jr $ra                   // return

BaseCase_1:
    li $v0, 1
    jr $ra                   // return

RecursiveStep:
    //apertura dello stack
    subi $sp,$sp,12
```

```
sw $ra,0($sp)
sw $a0,4($sp)
sw $v1,8($sp)

subi $a0,$a0,1
jal Fibonacci
move $v1, $v0           // $v1 = fib($a0 - 1)

subi $a0,$a0,1
jal Fibonacci           // $v0 = fib($a0 - 2)

add $v0,$v1,$v0         // $v0 = $v0 + $v1

//chiusura dello stack
lw $v1,8($sp)
lw $a0,4($sp)
lw $ra,0($sp)
addi $sp,$sp,12

jr $ra                  // return
```

Capitolo 3

L'Architettura MIPS

Una volta appreso quali **istruzioni** siamo in grado di svolgere tramite una CPU basata sull'architettura MIPS, siamo finalmente in grado di poterne studiare il funzionamento "sotto il cofano", ossia come vengono **realmente eseguite** tali **istruzioni**.

A primo impatto, l'esecuzione di un'istruzione da parte di una CPU può sembrare come una sorta di *magia arcana*. Tuttavia, come vedremo a breve, l'esecuzione di un'istruzione non è altro che l'**attivazione in modo sequenziale di una serie di componenti interconnessi tra loro**, riducendo il tutto a semplici operazioni come l'attivazione di un determinato bit per poter svolgere una somma o una sottrazione.

3.1 Progettazione della CPU

Vogliamo progettare una CPU semplice in grado di poter svolgere le seguenti **istruzioni di base**:

- **Accesso alla memoria**: lw, sw
- **Salti condizionati** (branch): beq, bne
- **Salti incondizionati** (jump): j, jal
- **Operazioni aritmetico-logiche tra registri**: add, sub, and, or, slt, ...
- **Operazioni aritmetico-logiche con costanti**: addi, subi, andi, ori, ...

Ricordando l'**architettura di base** di un calcolatore definita da Von Neumann (sezione [1.2](#)), siamo in grado di stabilire che, affinché si possano implementare tali istruzioni, siano necessari alcuni "ingredienti fondamentali":

- Un **registro Program Counter (PC)**, che possa tenere traccia dell'indirizzo in memoria dell'istruzione successiva da eseguire
- Un **Banco di Registri (Register File)**, contenenti i dati e gli argomenti necessari alle istruzioni

- Una **memoria**, che d'ora in poi vedremo come divisa (solo dal punto di vista teorico) tra **memoria dati** e **memoria istruzioni**, in modo da poterne semplificare l'interpretazione logica.
- Un **Arithmetic-Logic Unit (ALU)** in grado di svolgere tutte le operazioni aritmetico-logiche necessarie
- Una **Control Unit (CU)** in grado di gestire tutti i segnali necessari all'esecuzione di un'istruzione
- Un **Datapath**, ossia l'insieme delle interconnessioni tra i vari componenti

3.1.1 Fase di Instruction Fetch

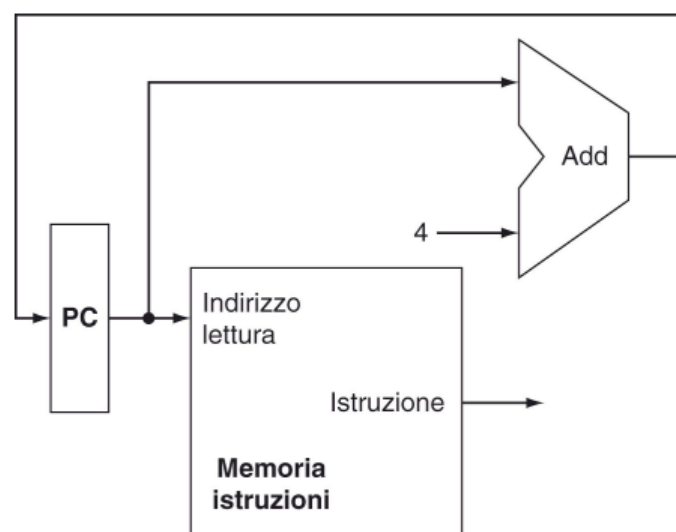
Per prima cosa, è necessario progettare la fase di **Instruction Fetch**, ossia il prelevamento dell'istruzione successiva da eseguire. I componenti necessari sono:

- Il **Program Counter**, ossia il registro (32 bit) contenente l'indirizzo in memoria dell'istruzione successiva
- La **memoria istruzioni**, avente in input l'indirizzo (32 bit) in memoria dell'istruzione da prelevare e in output l'istruzione (32 bit) corrispondente a tale indirizzo

Sapendo che l'indirizzo di memoria dell'istruzione successiva è contenuto all'interno del **Program Counter**, ci basta semplicemente connettere quest'ultimo all'input della **memoria istruzioni**, in modo da poter prelevare l'istruzione corrispondente a tale **indirizzo**.

Successivamente, poiché ogni istruzione corrisponde ad una **word**, sarà necessario **incrementare l'indirizzo contenuto del PC di 4 byte**, in modo da poter prelevare al prossimo ciclo di clock l'istruzione direttamente **successiva** a quella precedentemente eseguita.

Dunque, sarà necessario aggiungere all'architettura anche un **Sommatore (Adder)** avente in input il **PC** e un **valore costante pari a 4**, in modo da poter eseguire tale somma ad ogni ciclo di clock:

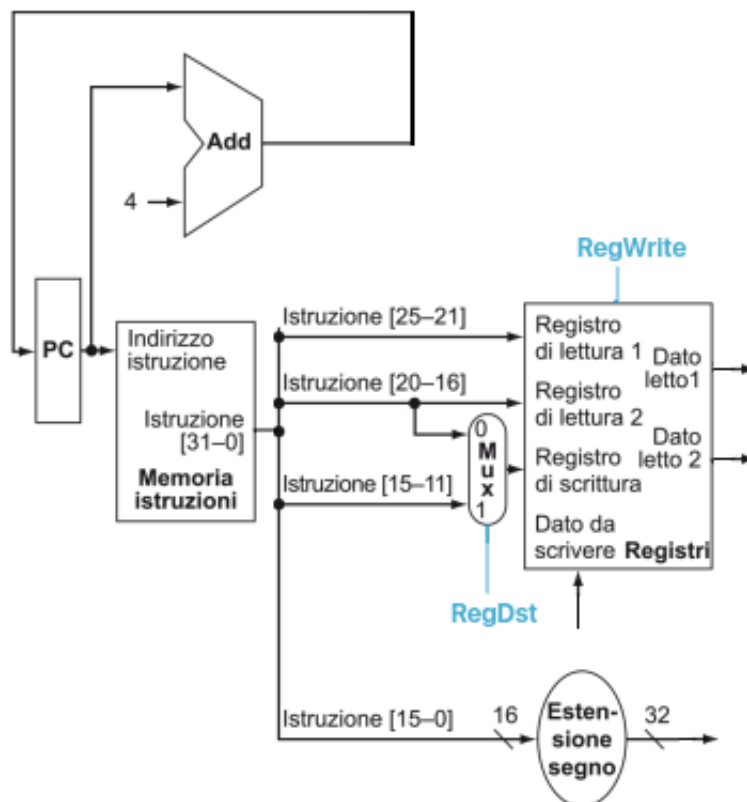


3.1.2 Fase di Instruction Decode

Successivamente, viene eseguita la fase di **Instruction Decode**, dove l'istruzione in output dalla fase di **fetch** viene **scomposta** in ognuno dei suoi **campi**, in modo da poterne prelevare i contenuti.

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

Poiché l'architettura deve essere in grado di poter lavorare con tutti e tre i formati di istruzioni, saranno necessari alcuni **multiplexer (mux)** e **segnali di controllo**, in modo che ne modifichino in comportamento a seconda del tipo di istruzione:

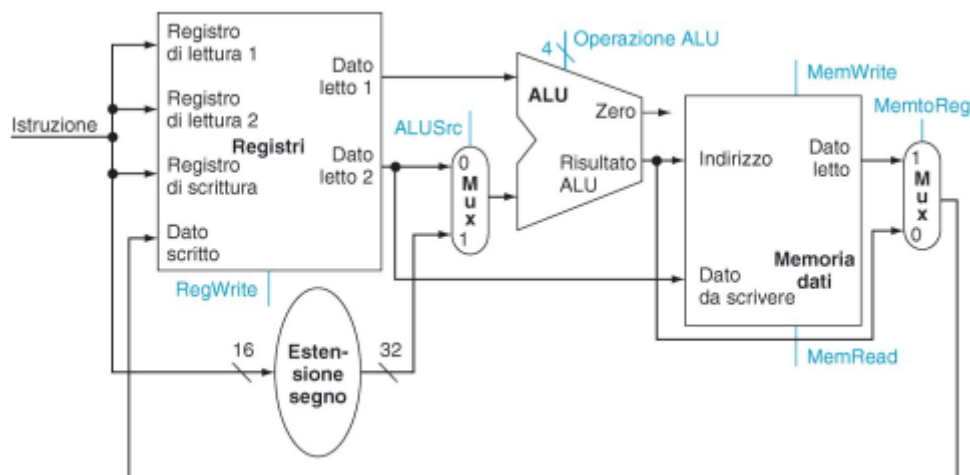


- **Registro di Lettura 1:** prenderà in input i bit corrispondenti al **campo \$rs** dell'istruzione, ossia i bit nel range [25-21]
- **Registro di Lettura 2:** prenderà in input i bit corrispondenti al **campo \$rt** dell'istruzione, ossia i bit nel range [20-16]
- **RegDst:** un segnale di controllo utilizzato come selettore di un Mux avente in input il **campo \$rt** e il **campo \$rd** dell'istruzione, ossia i bit nel range [20-16] e nel range [15-11]

- **Registro di Scrittura:** prenderà in input i bit corrispondenti al **campo selezionato** dal mux avente come selettore **RegDst**, scrivendo all'interno del registro corrispondente il contenuto del dato da scrivere passato in input
- **RegWrite:** un segnale di controllo che, se asserito (ossia vale 1), **abilita la scrittura** sul Register File, ignorandola altrimenti.
- **Estensione del Segno:** prenderà in input i bit corrispondenti alla **parte immediata** dell'istruzione, ossia i bit nel range [15-0], **estendendone il segno fino a 32 bit** copiando il valore del bit più a sinistra (ricordiamo che i numeri utilizzati sono in Complemento a 2)

3.1.3 Fase di Instruction Execute

Una volta decodificata l'istruzione prelevando i dati ad essa necessari, l'istruzione procederà con la fase di esecuzione, andando ad utilizzare l'**ALU** ed eventualmente ad accedere alla **memoria dati**.



L'implementazione delle **operazioni sull'ALU** risulta di facile intuizione:

- Sono presenti **4 bit di controllo** che stabiliscono l'operazione svolta dall'ALU

Segnali ALU	Operazione
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	SLT
1100	NOR

- Sono presenti **2 segnali di input a 32 bit** corrispondenti agli argomenti dell'operazione da svolgere: se l'istruzione da eseguire è di Tipo R, allora verranno usati i **due dati letti dai registri** come input, altrimenti verranno utilizzati il dato letto dal **registro \$rs** e la **parte immediata estesa**.

Per svolgere ciò, dunque, è necessario aggiungere un mux controllato da un segnale **ALUSrc** in grado di selezionare la fonte del secondo input dell'ALU.

- Sono presenti **2 segnali di output**, uno corrispondente al **risultato dell'operazione svolta** (32 bit) ed uno corrispondente ad una **segnale di flag** chiamato **Zero** (1 bit), il quale risulterà asserito (ossia valente 1) solo nel caso in cui il **risultato** dell'ALU sia esattamente **pari a 0**. Tale segnale risulterà fondamentale per le istruzioni di branch.

L'implementazione dell'accesso alla **memoria dati**, invece, risulta leggermente più complesso, poiché ogni istruzione necessita di svolgere **operazioni diverse** su di essa:

- L'ALU viene utilizzata per calcolare gli indirizzi di memoria con cui interagire, dunque il risultato dell'operazione svolta verrà utilizzato come input d'indirizzo
- Solo l'istruzione **lw** è in grado di leggere dalla memoria, dunque nel caso in cui essa venga eseguita sarà necessario restituire il **dato prelevato dalla memoria** stessa come dato da scrivere all'interno del registro di scrittura selezionato, mentre in qualsiasi altro caso sarà necessario restituire il **risultato stesso dell'ALU**.

Ciò determina quindi la necessità di dover inserire un mux avente come selettore un segnale di controllo **MemToReg** in grado di determinare quale dei due dati andare a scrivere nei registri.

- Solo l'istruzione **lw** deve essere in grado di poter **leggere** dalla memoria, mentre solo l'istruzione **sw** deve essere in grado di poter **scrivere** su di essa.

Ciò determina quindi la necessità di dover inserire un segnale di controllo **MemRead** abilitante la **lettura** ed un segnale di controllo abilitante la **scrittura** sulla memoria dati.

Unità di Controllo dell'ALU

Per semplificare l'uso dei segnali di controllo che determinano l'operazione da eseguire nell'ALU, possiamo implementare un'**unità di controllo** avente come input il **campo funct** dell'istruzione (ossia i bit nel range [5-0]), contenente la vera operazione da eseguire, e due segnali di controllo **ALUOp**, i quali 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	-	-	0	0	0	0	0010	ADD
sub	1	-	-	0	0	1	0	0110	SUB
and	1	-	-	0	1	0	0	0000	AND
or	1	-	-	0	1	0	1	0001	OR
slt	1	-	-	1	0	1	0	0111	SLT

Nota: con il segno "-" viene indicato un segnale Don't Care

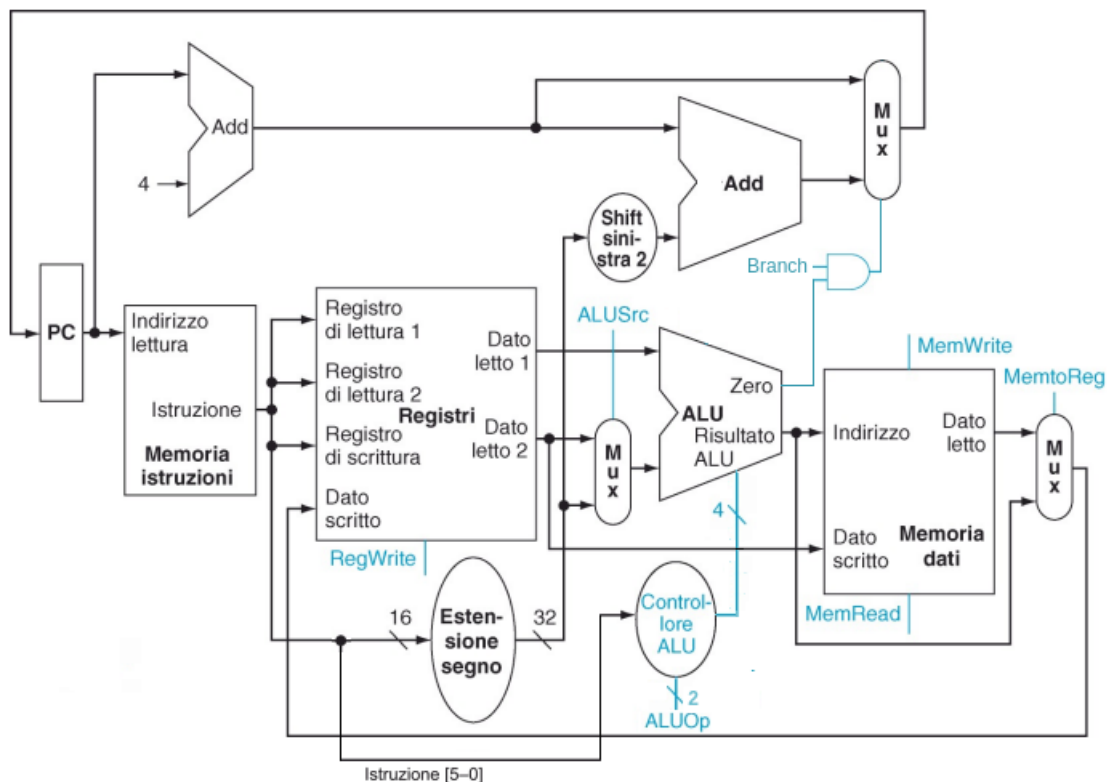
Difatti, è in realtà possibile utilizzare le istruzioni di branch senza dover necessariamente indicare una label (Es: `beq $rs, $rt, 14` corrisponde ad un salto in avanti di 14 istruzioni).

Una volta definita la logica necessaria alla **verifica della condizione** da soddisfare per effettuare il branch, sarà necessario modificare l'architettura per renderla in grado di calcolare l'**indirizzo relativo a cui effettuare il salto**.

Poiché la **parte immediata** dell'istruzione contiene il numero di istruzioni da saltare, sarà necessario **shiftare a sinistra di due posti** tale numero (dunque moltiplicandolo per 4), in modo da poter ottenere il **numero di byte corrispondenti** da saltare. Successivamente, ci basterà **sommare** tale quantità di byte da saltare all'indirizzo pre-calcolato dell'istruzione successiva, ossia $PC+4$.

$$\text{beq } \$rs, \$rt, \text{amount} \rightarrow PC = PC + 4 + (\text{amount} \ll 2) \text{ solo se } \$rs - \$rt == 0$$

Infine, sarà necessario introdurre un mux avente come selettore l'**AND** tra la **flag Zero** e il segnale di controllo **Branch**, il quale sarà asserito solo se l'istruzione eseguita è un branch.



Implementazione dei Salti Incondizionati (Jump)

L'implementazione di un salto incondizionato risulta estremamente simile a quella della sua variante condizionata, richiedendo tuttavia una manipolazione più accurata dei bit trattandosi di un **salto assoluto**, ossia non relativo all'istruzione corrente.

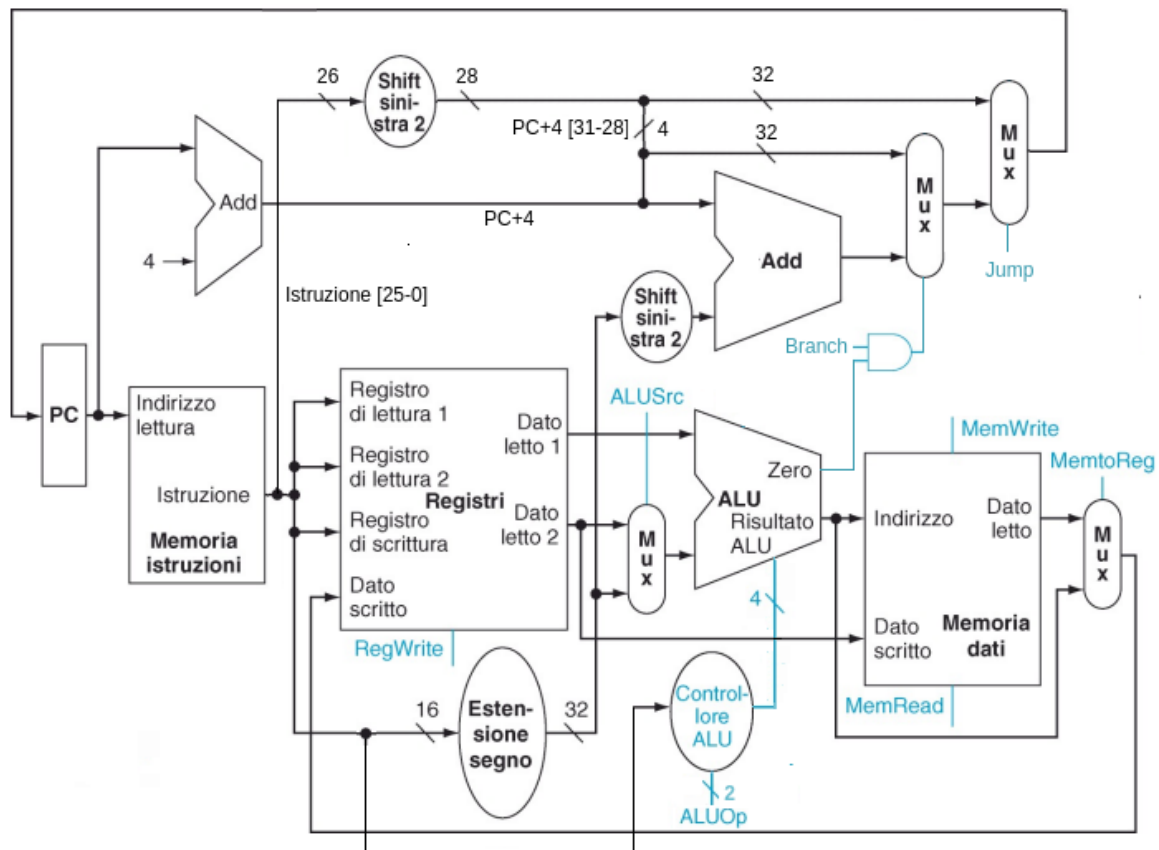
Per poter calcolare l'**indirizzo assoluto** a cui effettuare il salto è necessario fare alcune considerazioni:

- Essendo un salto assoluto, è necessario che l'indirizzo calcolato sia un indirizzo di memoria **confinato** all'interno della **zona di memoria dedica alla memoria istruzioni** (sezione 2.3.1). Per mantenere tale condizione sempre valida, i **4 bit** nel range [31-28] del PC rimangono **invariati** durante il calcolo
- La parte immediata dell'istruzione **jump**, ossia i bit nel range [25-0] dell'istruzione, contiene il numero dell'istruzione a cui saltare, dunque è necessario **shiftare a sinistra di due posti** tale valore, in modo da ottenere l'indirizzo di memoria in byte, ottenendo così i **28 bit** indicanti l'effettiva istruzione a cui saltare

L'indirizzo calcolato, quindi, sarà formato dall'**unione dei 4 bit invariati** (ossia $PC+4[31-28]$) **ai 28 bit calcolati tramite lo shift** a sinistra della parte immediata (ossia $Istruzione[25-0] \ll 2$).

$$j \text{ label} \longrightarrow PC = PC+4[31-28] \text{ OR } (label \ll 2)$$

$$\longrightarrow PC = PC+4[31-28] \text{ OR } (Istr[25-0] \ll 2)$$

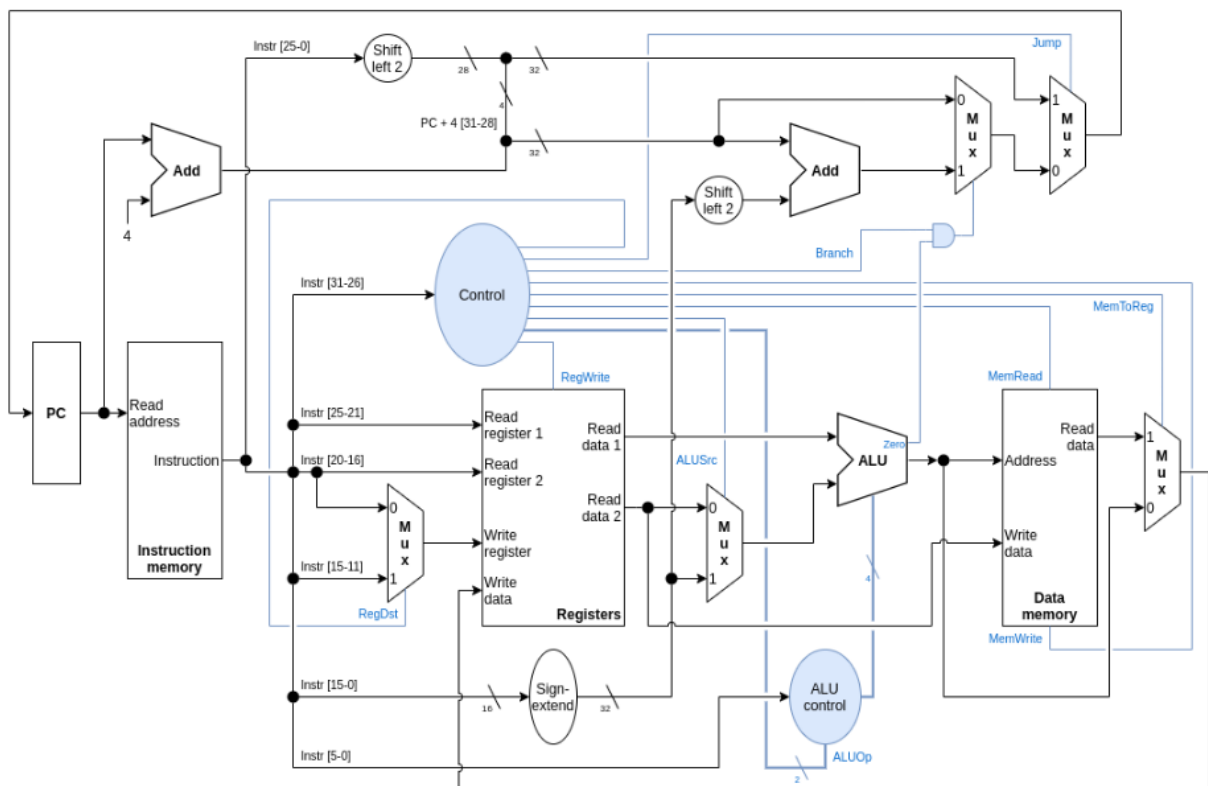


3.1.5 Control Unit ed Esecuzione delle istruzioni di base

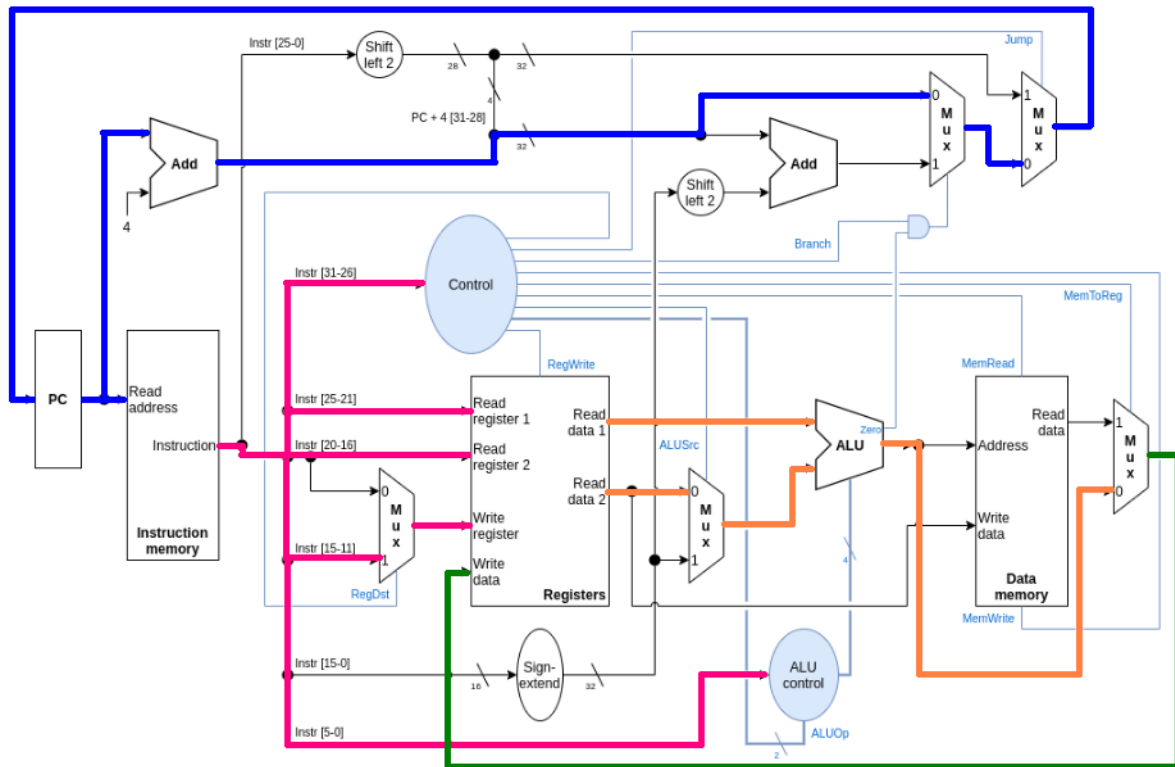
Abbiamo visto come all'interno dell'architettura sia necessario implementare dei **segnali di controllo** in grado di dettarne il **comportamento** a seconda dell'**istruzione** da eseguire.

Tali segnali di controllo vengono gestiti dalla **Control Unit (CU)**, la quale prenderà in input l'**opcode** dell'istruzione (ossia i bit nel range[31-25]) per poi attivare i segnali necessari all'esecuzione dell'istruzione stessa.

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



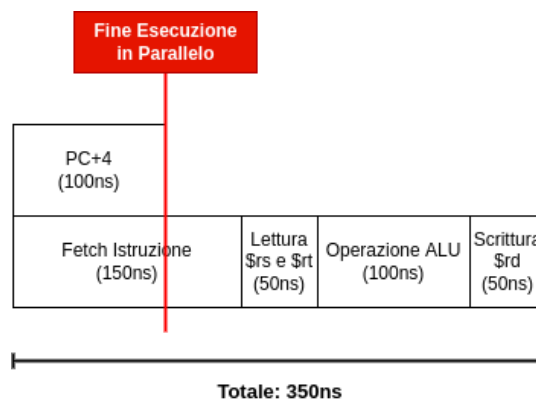
Segnali, Datapath e Tempo di esecuzione di una Type R

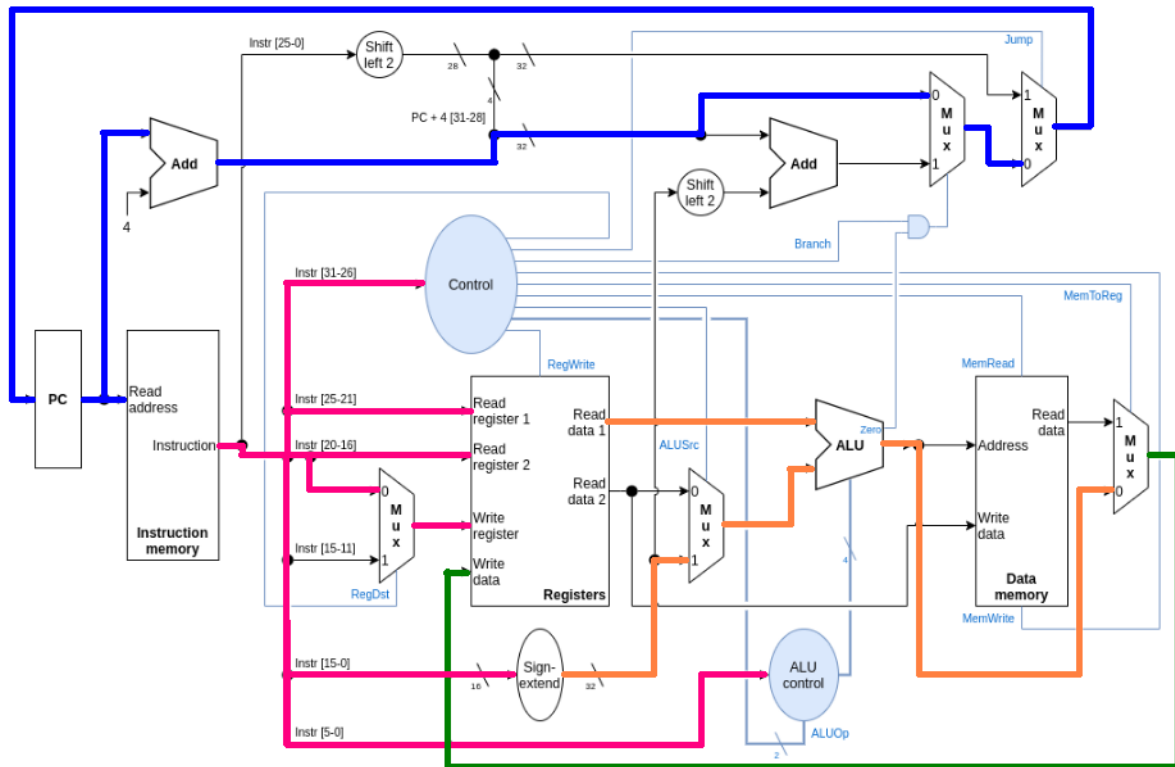


Istruzione	RegDst	RegWrite	ALUSrc	ALUOp [1]	ALUOp [0]	MemRead	MemWrite	MemToReg	Branch	Jump
Tipo R	1	1	0	1	-	-	0	0	0	0

Supponendo che:

- L'accesso alla **Memoria** impieghi **150ns**
- L'accesso al **Register File** impieghi **50ns**
- L'utilizzo dell'**ALU** e dei Sommatore impieghi **100ns**

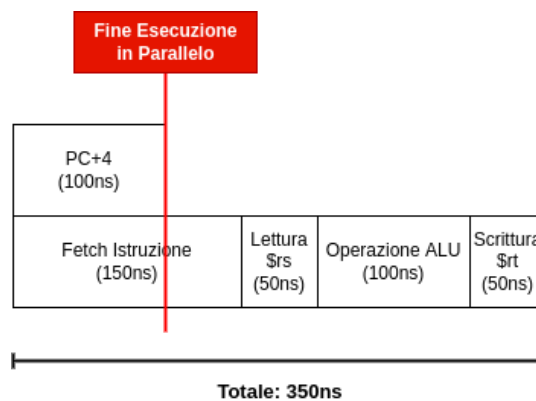


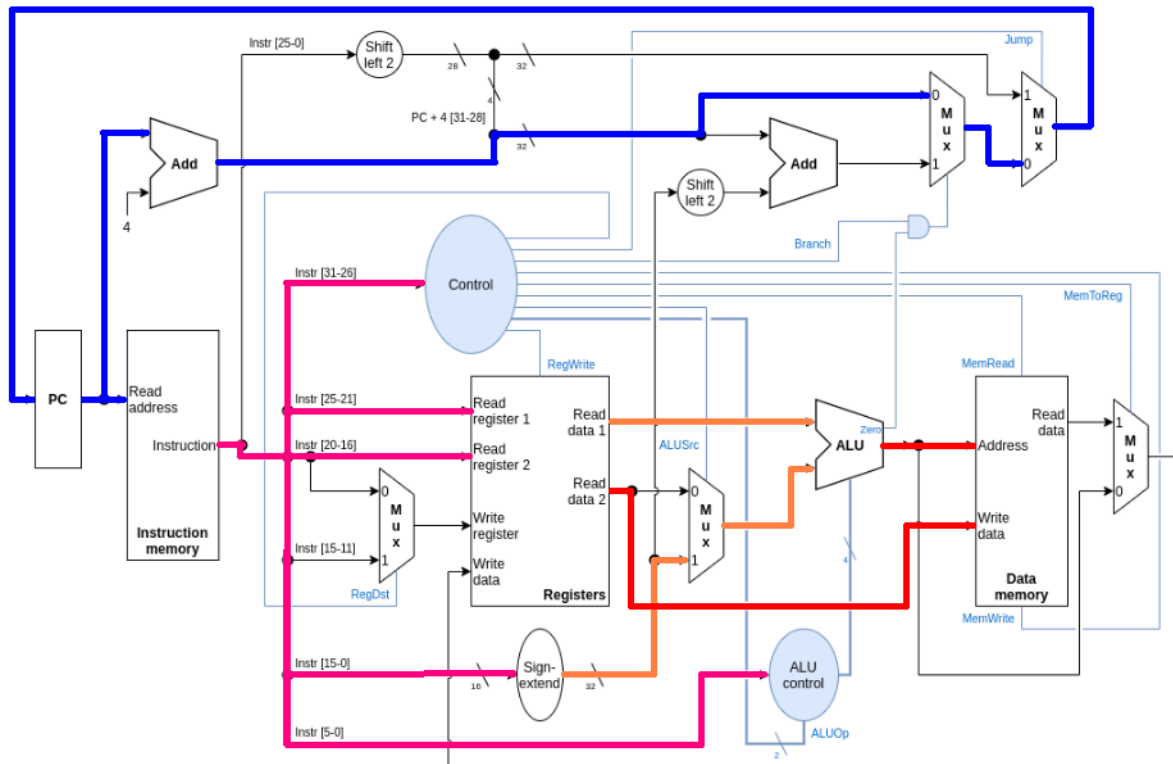
Segnali, Datapath e Tempo di esecuzione di una **Type I**

Istruzione	RegDst	RegWrite	ALUSrc	ALUOp [1]	ALUOp [0]	MemRead	MemWrite	MemToReg	Branch	Jump
Tipo I	0	1	1	1	-	-	0	0	0	0

Supponendo che:

- L'accesso alla **Memoria** impieghi **150ns**
- L'accesso al **Register File** impieghi **50ns**
- L'utilizzo dell'**ALU** e dei Sommatore impieghi **100ns**

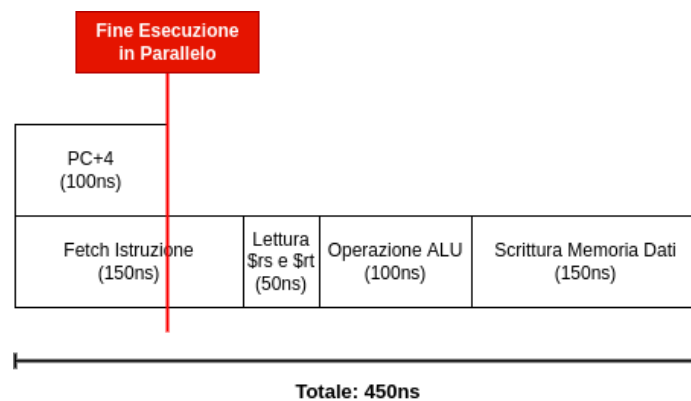


Segnali, Datapath e Tempo di esecuzione di una *sw*

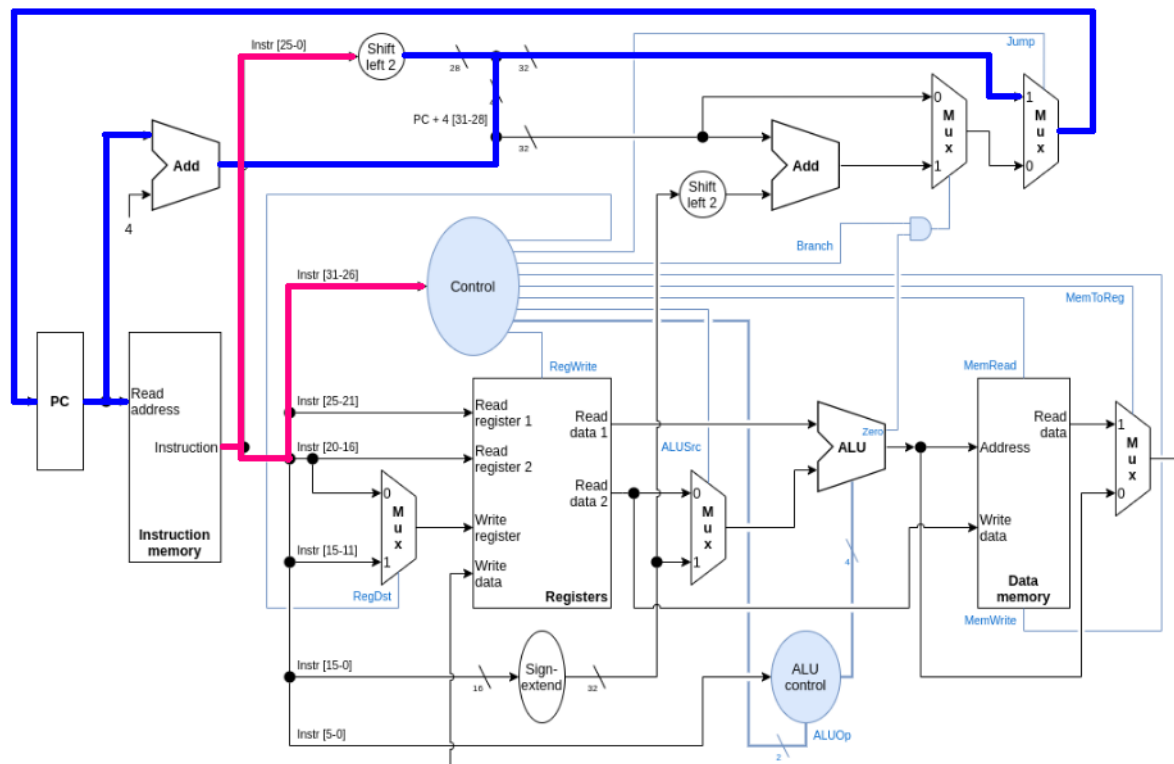
Istruzione	RegDst	RegWrite	ALUSrc	ALUOp [1]	ALUOp [0]	MemRead	MemWrite	MemToReg	Branch	Jump
sw	-	0	1	0	0	0	1	-	0	0

Supponendo che:

- L'accesso alla **Memoria** impieghi **150ns**
- L'accesso al **Register File** impieghi **50ns**
- L'utilizzo dell'**ALU** e dei Sommatore impieghi **100ns**



Segnali, Datapath e Tempo di esecuzione di una j



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

Supponendo che:

- L'accesso alla **Memoria** impieghi **150ns**
- L'accesso al **Register File** impieghi **50ns**
- L'utilizzo dell'**ALU** e dei Sommatore impieghi **100ns**

