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

28 marzo 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

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)**.

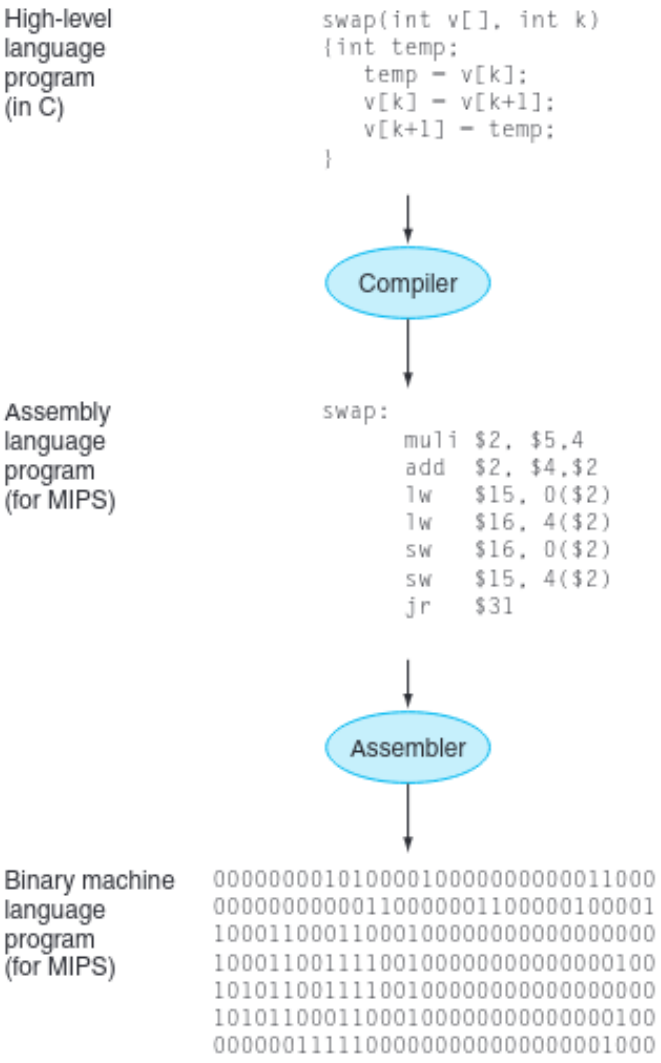
L'utilizzo del linguaggio assembly permise lo sviluppo agile e controllato di programmi avanzati, eliminando (parzialmente) fattori come l'**errore umano** (un calcolatore non può sbagliare a scrivere un bit al contrario di un umano) e la **lentezza** di progettazione.

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.

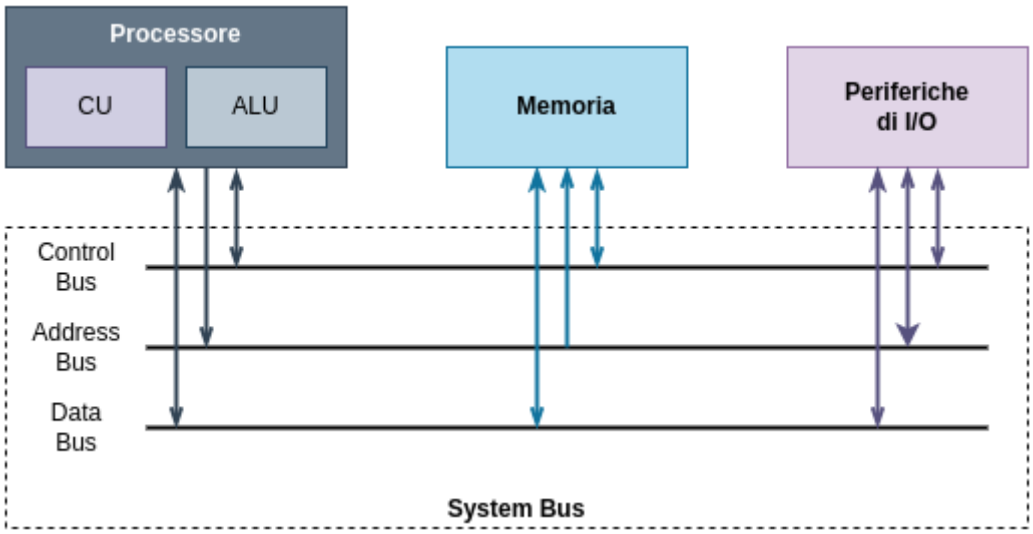
L'intera **catena di astrazione**, dunque, corrisponde a



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**:

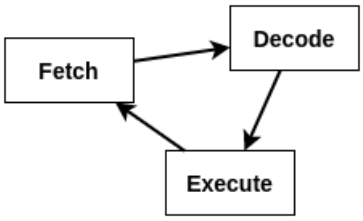
- **Central Processing Unit (CPU)**, ossia l'unità centrale di elaborazione (anche chiamato **processore**). Si occupa di eseguire una dopo l'altra tutte le istruzioni che compongono un **processo**, ossia un programma caricato in memoria. È a sua volta costituita da tre elementi:
  - **Control Unit (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



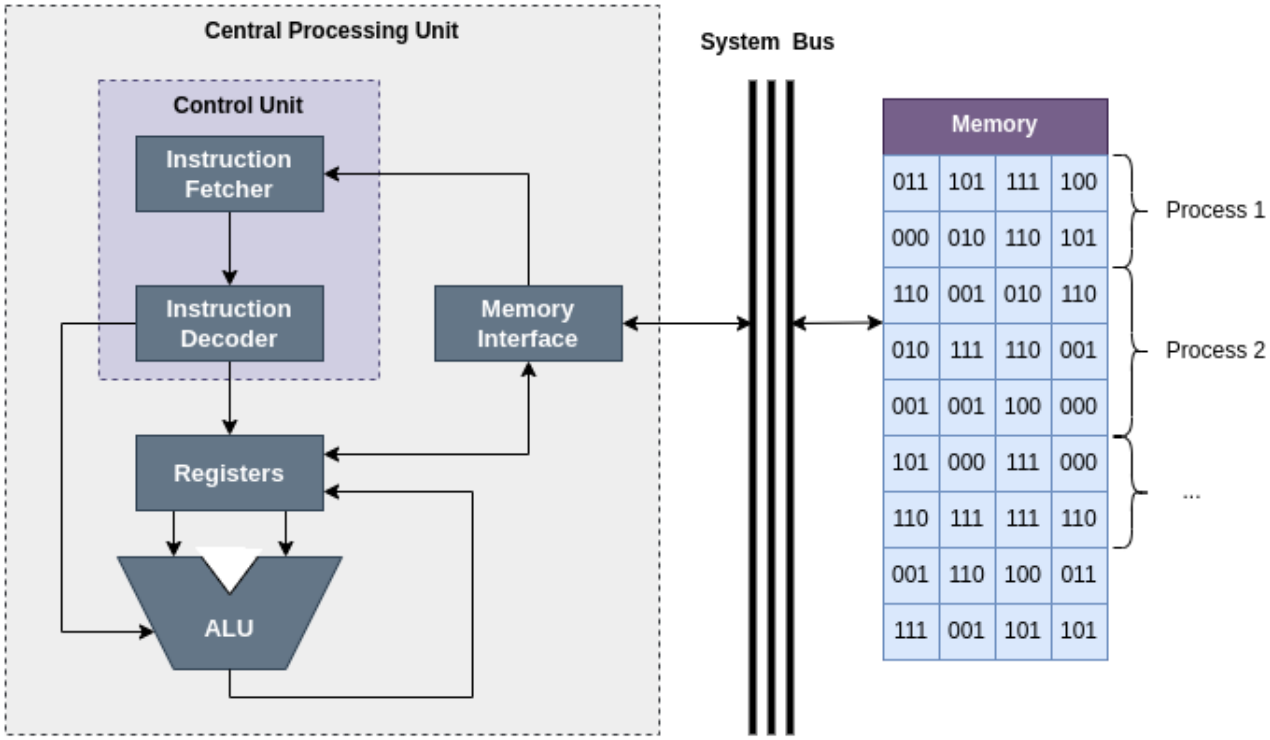
I primi modelli di computer (incluso quello di Neumann), erano progettati per eseguire un solo processo alla volta, mentre i moderni modelli sono provvisti di sistemi di **parallelismo**, permettendo la gestione di più processi in contemporanea che, attraverso un sistema di **scheduling**: una volta **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 Istruction 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 Istruction 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

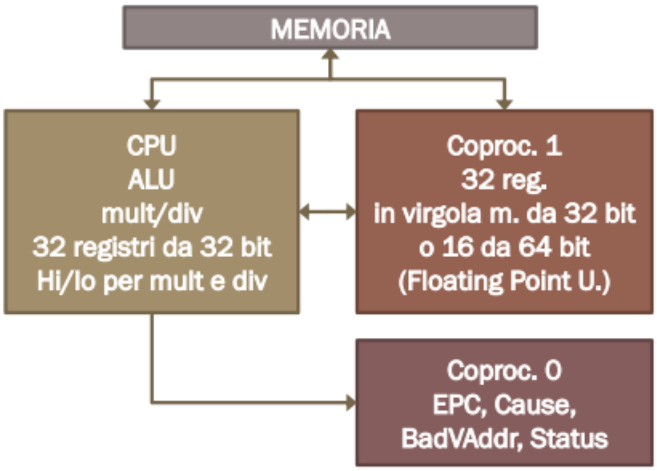
Riassumendo, possiamo dire che le **Architetture CISC** risultano più complesse ma ottimizzate per scopi singoli, mentre le **Architetture RISC**, in quanto più semplici, risultano adatte a scopi generici.

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)

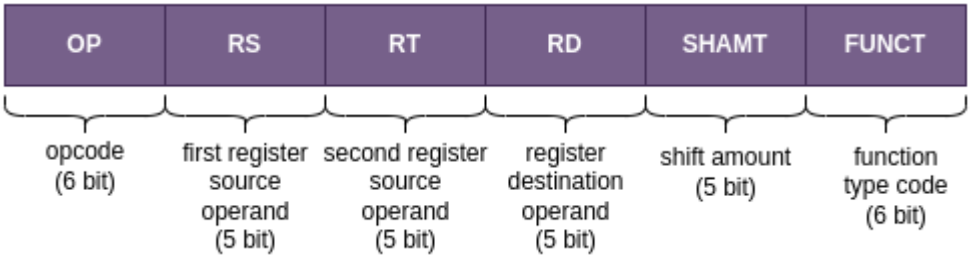
Tale struttura è solo una **generalizzazione** poiché, come vedremo in seguito, non è pienamente rispettata da ogni istruzione.

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:

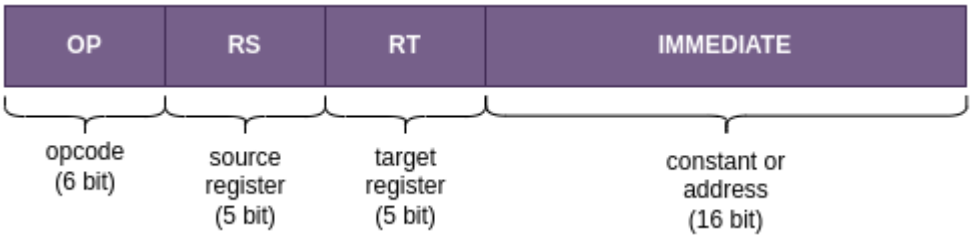
Istruzione	OP	RS	RT	RD	SHAMT	FUNCT
`add $t0, $s1, $s2`	000000	10001	10010	01000	00000	100000
`sub $t0, $s1, $s2`	000000	10001	10010	01000	00000	100010

    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:

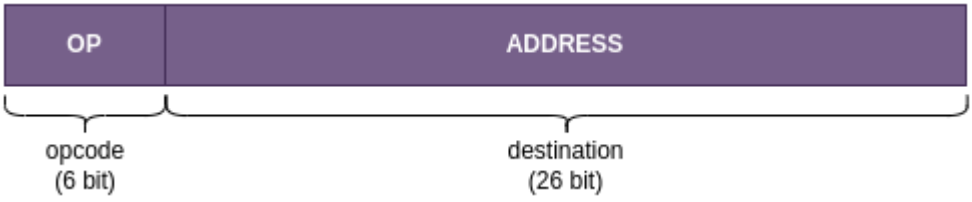


  - Esempio:

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 <u>non</u> 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 $2500 << 2$), poiché ricordiamo che l'architettura MIPS è indicizzata al byte, dunque la 2500-esima parola corrisponde all'indirizzo 10000 della memoria

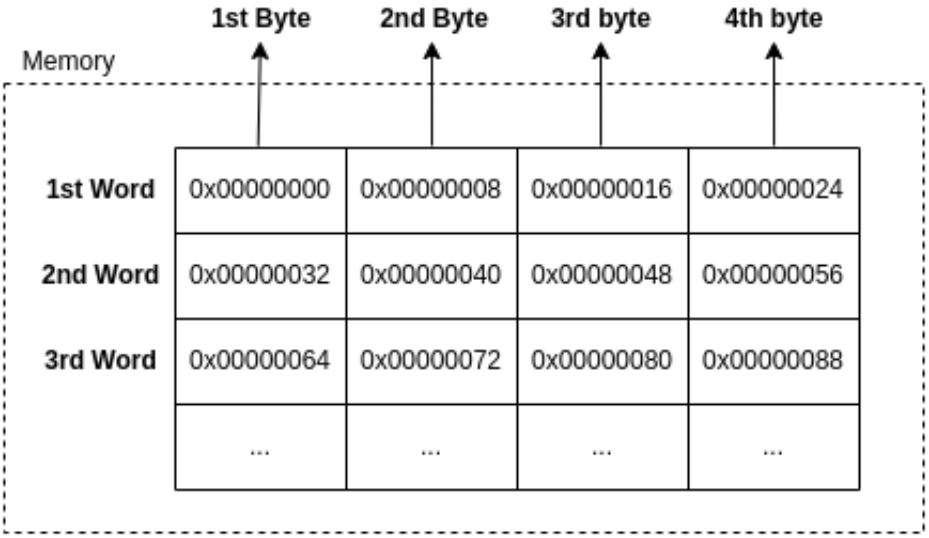
2.2 Lista delle istruzioni

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)`	`$s1=Memoria[$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
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` = 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 Byte $\cdot\, 2^{30}$ Word = 4 GigaByte.

Ad ogni byte della memoria è associato un **indirizzo**, rappresentato da un **numero esadecimale**.



***Nota**: gli indirizzi di memoria vengono rappresentati da 8 cifre esadecimali, poiché ricordiamo che ad ogni cifra esadecimale corrispondono esattamente 4 bit, dunque* $8 \cdot 4 = 32bit$ *per indirizzo di memoria*

Poiché ogni byte corrisponde ad 8 bit, l'indirizzo di memoria di o**gni byte adiacente** risulta **sfalzato di 8**. Nel caso delle **word**, invece, essere risultano **sfalzate di 32**, poiché ogni word è composta da 4 byte.

Generalizzando tutto ciò, possiamo dire che il **k-esimo** byte si trova all'indirizzo di memoria $8 \cdot (k-1)$, mentre la **j-esima** word si trova all'indirizzo $4 \cdot 8 \cdot (j-1)$.

Dunque, se volessimo leggere il contenuto della 100esima word, l'indirizzo di memoria corrispondente sarebbe

$$M = 4 \cdot 8 \cdot (100-1) = 3168_{10} = 0x00000c60$$

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, 3168     //carico in $t0 il valore 3168 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+32
```

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).

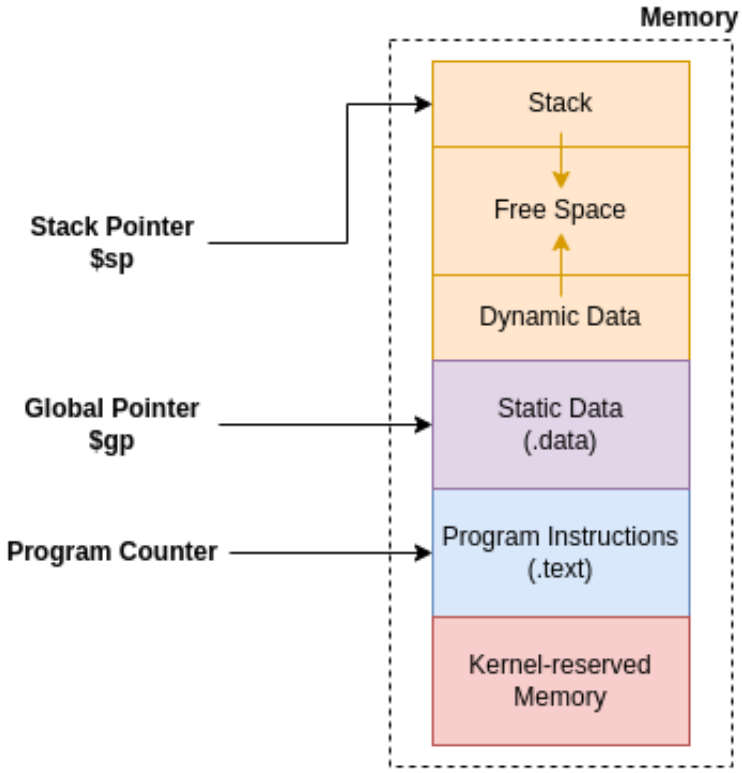
Successivamente, viene utilizzato due volte il comando **Load Word**, che permette **caricare un'intera word** all'interno del registro indicato (ricordiamo che sia la word e sia il registro sono formati da 32 bit).

- 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 100-esima word in memoria, poiché l'indirizzo indicato è 3168 (0x00000c60).
- Nel secondo utilizzo viene prelevata la word di **indirizzo $t0+32**, poiché l'**offset definito è 4**, dunque $4 \cdot 8 = 32$, leggendo quindi l**a parola direttamente successiva** a quella indicata da $t0, ossia la 101-esima parola, corrispondente all'indirizzo 3200 (0x00000c80)

L'utilizzo dei registri come "contenitore" 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**: 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 P**rogram 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
- **.asciiz**: 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 **primissimo 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 contente **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.