Il linguaggio Assembly – parte I

Architettura
degli Elaboratori e delle Reti

Alberto Borghese
Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione
borghese@dsi.unimi.it

Materiale didattico

- **Capitolo 3, 8 + Appendice A** del testo:
  - "Struttura, organizzazione e progetto dei calcolatori: interdipendenza tra hardware e software", di D.A. Patterson e J.L. Hennessy, Jackson Libri, 1999 (2a edizione).
- Oppure in versione inglese:
  - "Computer Organization & Design: The Hardware/Software Interface", D.A. Patterson and J.L. Hennessy, Morgan Kaufmann Publishers, Second Edition.
- Per un'introduzione generale sulle architetture:
  - "Introduzione ai sistemi informatici" D. Sciuto, G. Buonanno, W. Fonaciari e L. Mari, McGraw-Hill, Seconda Edizione.

Incontro con gli studenti: giovedi' ore 10.30 – 11.30

Materiale didattico

- **SPIM: A MIPS R2000/R3000 Simulator : PCSPIM version 6.3**
- http://www.cs.wisc.edu/~larus/spim.html
- Piattaforme:
  - Unix or Linux system
  - Microsoft Windows (Windows 95, 98, NT, 2000)
  - Microsoft DOS
- Altro materiale didattico:
  - http://homes.dsi.unimi.it/~borghese/Borghese/Teaching/Architetture/Architetture.html.
  - http://www.elet.polimi.it/Users/DEI/Sections/Compeng/Cristina.Silvano/ita/architettura%20I.htm.
  - webcen.usr.dsi.unimi.it/arch/index.html.

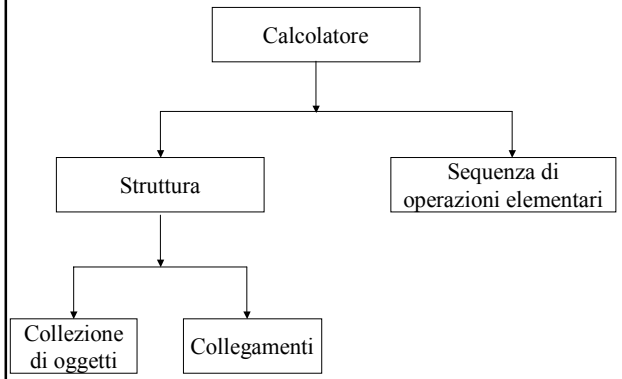
Architettura MIPS

- Obiettivo del corso:
  linguaggio assembly dell'architettura MIPS
- Architettura MIPS appartiene alla famiglia delle architetture **RISC (Reduced Instruction Set Computer)** sviluppate dal 1980 in poi
  - Esempi: Sun Sparc, HP PA-RISC, IBM Power PC, DEC Alpha, Silicon Graphics.
- Principali obiettivi delle architetture RISC:
  - Semplificare la progettazione dell'hardware e del compilatore
  - Massimizzare le prestazioni
  - Minimizzare i costi

Sommario

- Architettura di riferimento dei calcolatori
- Esecuzione delle istruzioni
- Il linguaggio assembly
- Il linguaggio macchina
- Insieme delle istruzioni
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

Descrizione di un calcolatore



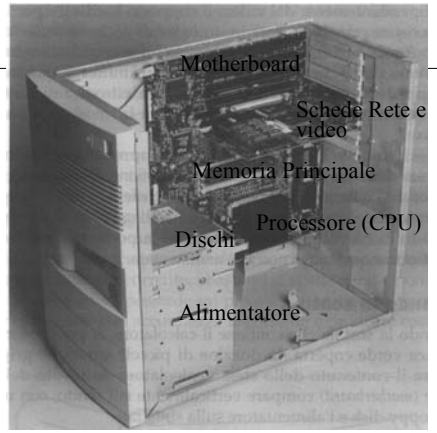
Architettura di riferimento dei calcolatori

Elabora in modo adeguato un input per produrre l'output.

Dispositivi di Input/Output

- Le unità di *ingresso* (tastiera del terminale video, mouse o altri dispositivi grafici di ingresso, ecc.) permettono al calcolatore di acquisire informazioni dall'ambiente esterno.
- Le unità di *uscita* (monitor grafico del terminale video, stampanti, ecc.) consentono al calcolatore di comunicare i risultati ottenuti dall'elaborazione all'ambiente esterno.

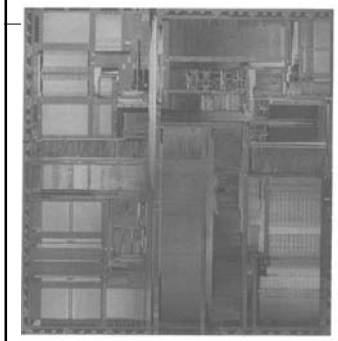
Componenti di un calcolatore

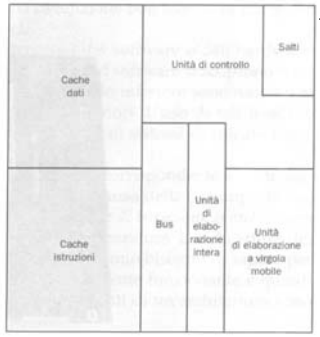


Architettura di riferimento dei calcolatori (Architettura di Von Neumann)

- Elementi principali di un calcolatore:
  - Unità centrale di elaborazione (*Central Processing Unit - CPU*)
  - Memoria di lavoro o memoria principale *(Main Memory - MM)*
- Sulla motherboard:menti principali di un calcolatore:
  - Bus di sistema (dati, indirizzi, controllo)
  - Interfacce per i dispositivi di *Input/Output - I/O*: il terminale, la memoria di massa (di solito dischi magnetici), le stampanti, …

Componenti di un processore Pentium





• 3,3 Milioni di transistor su 91mm^2

Unità centrale di elaborazione *(Central Processing Unit - CPU)*

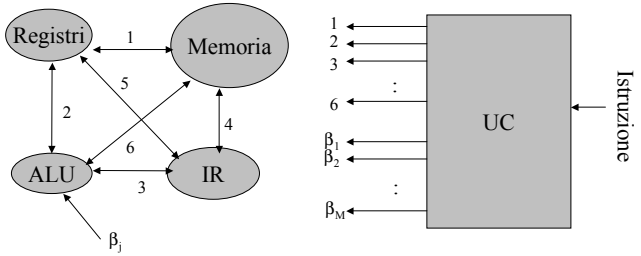
- La *CPU* provvede ad eseguire le istruzioni che costituiscono i diversi programmi elaborati dal calcolatore.
- Eseguire un'istruzione vuol dire operare delle scelte, eseguire dei calcoli a seconda dell'istruzione e dei dati a disposizione.

Elementi principali della CPU

- Banco di registri *(Register File)* ad accesso rapido, in cui memorizzare i dati di utilizzo più frequente. Il tempo di accesso ai registri è circa 10 volte più veloce del tempo di accesso alla memoria principale;
- Registro *Program counter (PC)*. Contiene l'indirizzo dell'istruzione corrente da aggiornare durante l'evoluzione del programma, in modo da prelevare dalla memoria la corretta sequenza di istruzione;
- Registro *Instruction Register (IR)*. Contiene l'istruzione in corso di esecuzione.
- Unità per l'esecuzione delle operazioni aritmetico-logiche (*Arithmetic Logic Unit - ALU*). I dati forniti all'*ALU* possono provenire da registri oppure direttamente dalla memoria, a seconda delle modalità di indirizzamento previste;
- Unità aggiuntive per elaborazioni particolari come unità aritmetiche per dati in virgola mobile *(Floating Point Unit – FPU)*, sommatori ausiliari, ecc.;

L'unità di controllo

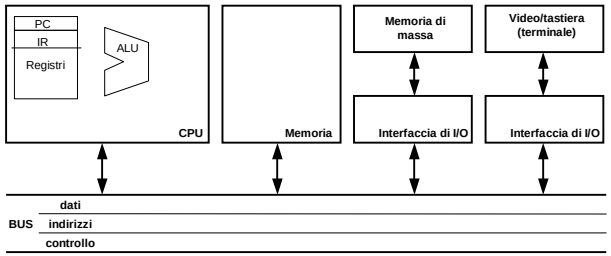
- Unità di controllo coordina i flussi di informazione:
- 1) abilitando le vie di comunicazione opportune a seconda dell'istruzione in corso di esecuzione.
- 2) selezionando l'operazione opportuna delle ALU.



Collegamenti bidirezionali tra i dispositivi: n (n-1) → non praticabile.

Collegamento tra i vari componenti (Architettura di Von Neumann)

Connessione *a nodo comune (bus).*

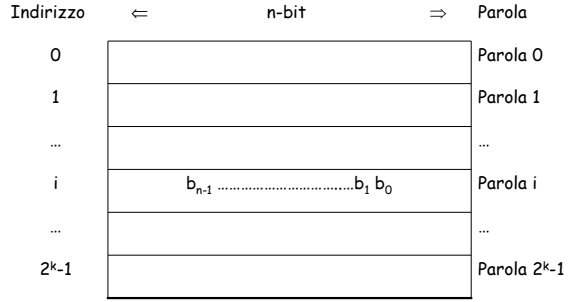


Numero di collegamenti: 2n.

Bus di sistema

- Permette la comunicazione tra le diverse unità del calcolatore ed è generalmente composto da tre parti:
  - *Bus dati*, comprende le linee per trasferire dati e istruzioni da/verso I dispositivi (la memoria). In generale, la dimensione del bus dati è tale da garantire il trasferimento contemporaneo di una o più parole di memoria;
  - *Bus indirizzi*, su cui la *CPU* provvede a trasmettere l'indirizzo da cui prelevare il dato nel caso di lettura dalla memoria, oppure in cui depositarlo nel caso di scrittura nella memoria (esempio la cella di memoria).
  - *Bus di controllo*, dove transitano le informazioni ausiliarie per la corretta definizione delle operazioni da compiere (per esempio l'indicazione che si vuole effettuare una *lettura* piuttosto che una *scrittura*) e per la sincronizzazione tra *CPU* e memoria.

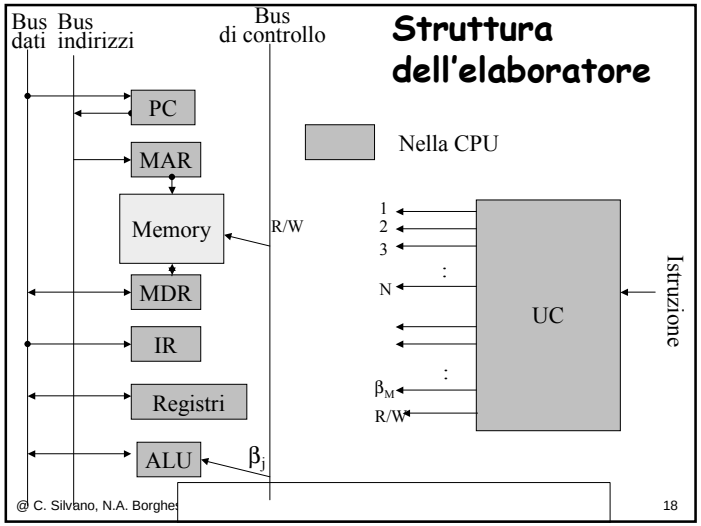
Indirizzi nella memoria principale



Indirizzi nella memoria principale

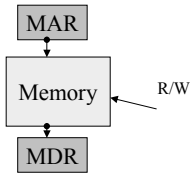
- La memoria è organizzata in *parole* o *word* composte da *n-bit* che possono essere caricate e memorizzate con una singola operazione di lettura/scrittura della memoria. (*n* è chiamata *lunghezza di parola,* tipicamente da *16* a *64* bit).
- Ogni parola di memoria è associata ad un indirizzo composto da *k-bit*.
- I 2^k indirizzi (corrispondenti a 2^k parole) costituiscono lo *spazio di indirizzamento* del calcolatore. Ad esempio un indirizzo composto da *32-bit* genera uno spazio di indirizzamento di 2^{32} o *4G* parole.

Struttura dell'elaboratore



Interfaccia processore-memoria

- **MAR – Memory Address Register:** registro degli indirizzi della memoria per memorizzare l'indirizzo della posizione della memoria principale in cui o da cui i dati o istruzioni devono essere trasferiti.
- **MDR – Memory Data Register:** registro dei dati della memoria per memorizzare i dati che devono essere scritti in memoria o letti dalla memoria e le istruzioni lette dalla memoria nella locazione indicata dall'indirizzo specificato.



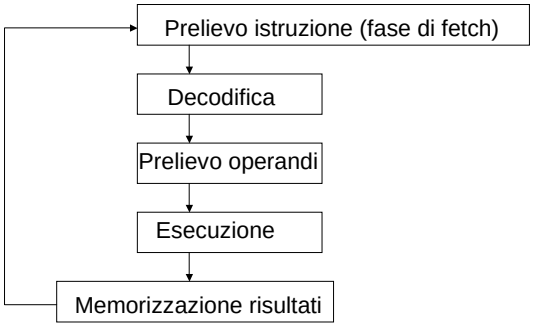
Sommario

- Architettura di riferimento dei calcolatori
- Esecuzione delle istruzioni
- Il linguaggio assembly
- Il linguaggio macchina
- Insieme delle istruzioni
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

Codifica dati e istruzioni

- Dati e istruzioni vengono manipolate dal calcolatore dopo essere state opportunamente *codificate*.
- L'istruzione è suddivisa in *campi* (*field*):
  - il campo *codice operativo* indica il tipo di operazione;
  - gli altri campi indicano gli indirizzi degli operandi oppure gli operandi stessi. Gli indirizzi possono riferirsi ad indirizzi di memoria o ai registri della CPU.
- Le *modalità di indirizzamento* indicano le diverse modalità attraverso le quali far riferimento agli operandi nelle istruzioni.

Ciclo di esecuzione di un'istruzione



Lettura dell'istruzione (fetch)

- Istruzioni e dati risiedono nella memoria principale, dove sono stati caricati attraverso un'unità di ingresso.
- L'esecuzione di un programma inizia quando il registro PC punta alla prima istruzione del programma.
- Il contenuto del PC viene trasferito nel MAR e un segnale di controllo per la lettura (READ) viene inviato alla memoria.
- Trascorso il tempo necessario all'accesso in memoria, la parola indirizzata (in questo caso la prima istruzione del programma) viene letta dalla memoria e trasferita nel registro MDR.
- Il contenuto del registro MDR (istruzione) viene trasferito nel registro IR.

Decodifica ed esecuzione dell'istruzione

- L'istruzione contenuta nel registro IR viene decodificata ed eseguita.
- Se l'istruzione deve essere svolta dall'unità aritmetico-logica è necessario recuperare gli operandi richiesti, che possono risiedere nei registri di uso generale oppure in memoria.
- Se un operando risiede in memoria, deve essere prelevato caricando l'indirizzo dell'operando nel registro MAR e attivando un ciclo di READ della memoria.
- L'operando letto dalla memoria viene posto nel registro MDR per essere trasferito alla ALU, che esegue l'operazione.

Scrittura in memoria

- Il risultato dell'operazione può essere memorizzato nei registri ad uso generale oppure in memoria.
- Se il risultato dell'operazione deve essere posto in memoria, esso viene caricato nel registro MDR. L'indirizzo della posizione di memoria in cui scrivere il risultato viene caricato nel registro MAR e si attiva un ciclo di scrittura (WRITE) della memoria.
- Mentre viene eseguita un'istruzione, il contenuto del PC viene incrementato in modo da puntare alla prossima istruzione da eseguire.
- Non appena è terminata l'esecuzione dell'istruzione corrente, si preleva l'istruzione successiva dalla memoria.

Architetture LOAD/STORE

- Il numero dei registri ad uso generale (ad esempio 32 registri da 32 bit ciascuno) non è sufficientemente grande da consentire di memorizzare tutte le variabili di un programma ⇒ ad ogni variabile viene assegnata una locazione di memoria nella quale trasferire il contenuto del registro quando questo deve essere utilizzato per contenere un'altra variabile.
- *Architetture LOAD/STORE:* gli operandi dell'ALU possono provenire soltanto dai registri ad uso generale contenuti nella CPU e **non** possono provenire dalla memoria. Sono necessarie apposite istruzioni di:
  - *caricamento (LOAD)* dei dati da memoria ai registri;
  - *memorizzazione (STORE)* dei dati dai registri alla memoria.

CPU di tipo RISC (*Reduced Instruction Set Computer*)

- Ispirate al principio di eseguire soltanto istruzioni semplici: le operazioni complesse vengono scomposte in una serie di istruzioni più semplici da eseguire in un ciclo base ridotto, con l'obiettivo di migliorare le prestazioni ottenibili dalle *CPU CISC*.
- Caratterizzate da istruzioni molto semplificate.
- Gli operandi dell'*ALU* possono provenire dai registri ma *non* dalla memoria. Per il trasferimento dei dati da memoria ai registri e viceversa si utilizzano delle apposite operazioni di caricamento (*load*) e di memorizzazione (*store)* ⇒ *architetture load/store*.

CPU di tipo RISC (*Reduced Instruction Set Computer*)

- *CPU* relativamente semplice ⇒ si riducono i tempi di esecuzione delle singole istruzioni, che sono però meno potenti delle istruzioni *CISC*.
- Dimensione *fissa* delle istruzioni ⇒ più semplice la gestione della fase di prelievo *(fetch)* e della codifica delle istruzioni da eseguire.

CPU di tipo CISC (*Complex Instruction Set Computer*)

- Caratterizzate da elevata complessità delle istruzioni eseguibili ed elevato numero di istruzioni che costituiscono l'insieme delle istruzioni.
- Numerose modalità di indirizzamento per gli **operandi** dell'*ALU* che possono provenire da registri oppure da memoria, nel qual caso l'indirizzamento può essere diretto, indiretto, con registro base, ecc.

CPU di tipo CISC (*Complex Instruction Set Computer*)

- Dimensione *variabile* delle istruzioni a seconda della modalità di indirizzamento di ogni operando ⇒ complessità di gestione della fase di prelievo o *fetch* in quanto a priori non è nota la lunghezza dell'istruzione da caricare.
- Elevata complessità della *CPU* stessa (cioè dell'hardware relativo) in termini degli elementi che la compongono con la conseguenza di rallentare i tempi di esecuzione delle operazioni. Elevata profondità dell'albero delle porte logiche, utilizzato per la decodifica.

Interruzioni

- Oltre a trasferire dati/istruzioni tra memoria e processore, il calcolatore acquisisce dati dai dispositivi di ingresso e invia dati ai dispositivi di uscita attraverso apposite istruzioni che gestiscono i trasferimenti di I/O.
- Il normale flusso di esecuzione di un programma può essere interrotto da un segnale di interruzione *(INTERRUPT)* per una richiesta di intervento che un dispositivo di I/O manda al processore.

Interruzioni

- Il processore fornisce il servizio richiesto mediante l'esecuzione di una procedura di servizio delle interruzioni *(Interrupt Service Routine)* che deve salvare in memoria lo stato del processore prima di servire l'interruzione: il contenuto del PC, dei registri ad uso generale e alcune informazioni di controllo vengono salvati in memoria.
- Quando la procedura di servizio dell'interruzione viene completata, lo stato del processore viene ripristinato in modo che il programma interrotto possa proseguire.

Memoria Principale

- Compito principale consiste nel contenere i moduli attivi del sistema operativo ed i processi in esecuzione (completi di istruzioni e dati).
- Caratteristica fondamentale è la dimensione complessiva.
- L'unità di misura della capacità di memoria è il ***bit***, anche se in genere si adotta il ***byte*** (che corrisponde a ***8 bit)*** ed i suoi multipli: *Kbyte (2^{10} = 1024 byte), Mbyte (2^{20} = 1 048 576 byte), Gbyte (2^{30} = 1 073 741 824 byte).*
- Caratterizzata anche dalla dimensione di ogni singolo elemento (*parola*) che può essere trasferito. Nei calcolatori più recenti la dimensione della parola va dai *32* bit ai *128* bit (dai *4* ai *16* byte).

Memoria Principale

- In genere, la dimensione della parola di memoria coincide con la dimensione dei registri contenuti nella *CPU*, in modo da poter caricare una parola di memoria in un registro della *CPU*. Se anche il bus dati è largo come la parola di memoria
  ⇒ l'operazione di *load/store* avviene in un singolo ciclo.
- Le memorie in cui ogni locazione può essere raggiunta in un breve e prefissato intervallo di tempo misurato a partire dall'istante in cui si specifica l'indirizzo desiderato, vengono chiamate memorie ad accesso casuale *(Random Access Memory – RAM)*
- Nelle RAM il *tempo di accesso alla memoria* (tempo necessario per accedere ad una parola di memoria) è *fisso* e *indipendente* dalla posizione della parola alla quale si vuole accedere.

Bus di sistema

- Permette la comunicazione tra le diverse unità del calcolatore ed è generalmente composto da tre parti:
  - *Bus dati*, comprende le linee per trasferire dati e istruzioni da/verso la memoria. In generale, la dimensione del bus dati è tale da garantire il trasferimento contemporaneo di una o più parole di memoria;
  - *Bus indirizzi*, Trasmette l'indirizzo del dispositivo, ad esempio della cella di memoria.
  - *Bus di controllo*, dove transitano le informazioni ausiliarie per la corretta definizione delle operazioni da compiere (per esempio l'indicazione che si vuole effettuare una *lettura* piuttosto che una *scrittura*) e per la sincronizzazione tra *CPU* e memoria.

Esempio di utilizzo del bus

- Esempio: operazione di lettura dalla memoria. La *CPU* fornisce l'indirizzo della parola desiderata sul bus indirizzi, quindi viene richiesta l'operazione di lettura attivando il bus di controllo. Quando la memoria ha completato la lettura della parola richiesta, il dato viene trasferito sul bus dati e la *CPU* può prelevarlo ed utilizzarlo nelle sue elaborazioni.
- La struttura del bus può essere realizzata secondo diverse topologie di interconnessione
- Nella struttura a *bus singolo* tutte le unità del calcolatore sono connesse al bus (architettura a nodo comune).
- Il bus può esser utilizzato per un solo trasferimento alla volta ⇒ in ogni istante soltanto due unità *(Master e Slave)* possono usare il bus.
- Nelle architetture di riferimento l'unità Master è solitamente la CPU. Esistono alcune schede di I/O particolarmente performanti che possono diventare anch'esse bus-master.
- Le linee di controllo del bus vengono utilizzate per inviare più richieste contemporanee di utilizzo del bus che vengono gestite dalla logica di *arbitraggio* del bus.

Bus di sistema & buffer

- Principali vantaggi della struttura a bus singolo: elevata flessibilità e bassi costi.
- I dispositivi collegati al bus variano in termini di velocità dell'esecuzione delle operazioni ⇒ necessario un meccanismo di sincronizzazione per garantire il trasferimento efficiente delle informazioni sul bus.
- Tipicamente all'interno delle unità che utilizzano il bus sono presenti dei registri di buffer per mantenere l'informazione durante i trasferimenti e non vincolarsi alla velocità del dispositivo più lento connesso al bus.

Sommario

- Architettura di riferimento dei calcolatori
- Esecuzione delle istruzioni
- Il linguaggio assembly
- Il linguaggio macchina
- Insieme delle istruzioni
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

Linguaggio assembly

- Rappresentazione simbolica del linguaggio macchina
  - Più comprensibile del linguaggio macchina in quanto utilizza simboli invece che sequenze di bit
- Rispetto ai linguaggi ad alto livello, l'assembly fornisce limitate forme di controllo del flusso e non prevede articolate strutture dati

Linguaggio C: esempio

```
main()
{
       int i;
       int sum = 0;
       for (i = 0; i <= 100; i = i + 1)
              sum = sum + i*i;
       printf("La somma da 0 a 100 è %d\n",sum);
}
```

Linguaggio assembly: esempio

```
.text
.align 2
.globl main
main:
        subu $sp, $sp, 32
        sw $ra, 20($sp)
        sw $a0, 32 ($sp)
        sw $0, 24($sp)
        sw $0, 28($sp)
loop:   lw  $t6, 28($sp)
        lw $t8, 24($sp)
        mult $t4, $t6, $t6
```

```
addu $t9, $t8, $t4
addu $t9, $t8, $t7
sw  $t9, 24($sp)
addu $t7, $t6, 1
sw $t7, 28($sp)
bne $t5, 100, loop

……..
```


Codifica binaria

- 00100111101111011111111111100000
- 10101111101111110000000000010100
- 10101111101001000000000000100000
- 10101111101001010000000000100100
- 10101111101001010000000000100100
- 10101111101001010000000000011000
- ……

Linguaggio assembly

- Linguaggio usato come linguaggio target nella fase di compilazione di un programma scritto in un linguaggio ad alto livello (es: C, Pascal,ecc.)
- Vero e proprio linguaggio di programmazione che fornisce la visibilità diretta sull'hardware.

Assembly come linguaggio di programmazione

- Principali *svantaggi* della programmazione in linguaggio assembly:
  - Mancanza di portabilità dei programmi su macchine diverse
  - Maggiore lunghezza, difficoltà di comprensione, facilità d'errore rispetto ai programmi scritti in un linguaggio ad alto livello

Assembly come linguaggio di programmazione

- Principali *vantaggi* della programmazione in linguaggio assembly:
  - Ottimizzazione delle prestazioni.
  - Massimo sfruttamento delle potenzialità dell'hardware sottostante.

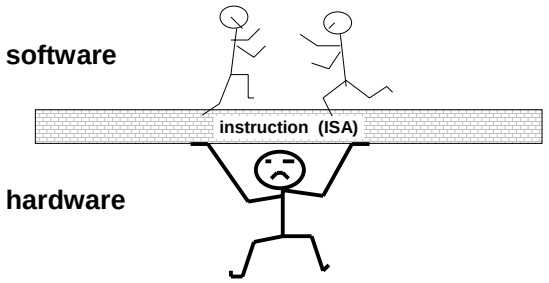
Assembly come linguaggio di programmazione

- Le strutture di controllo hanno forme limitate
- Non esistono tipi di dati all'infuori di interi, virgola mobile e caratteri.
- La gestione delle strutture dati e delle chiamate a procedura deve essere fatta in modo esplicito dal programmatore

Assembly come linguaggio di programmazione

- Alcune applicazioni richiedono un approccio *ibrido* nel quale le parti più critiche del programma sono scritte in assembly (per massimizzare le prestazioni) e le altre parti sono scritte in un linguaggio ad alto livello (le prestazioni dipendono dalle capacità di ottimizzazione del compilatore).
- Esempio: Sistemi embedded o dedicati

Insieme delle istruzioni



Quale è più facile modificare?

Fase di compilazione da C ad assembly

Programma in linguaggio ad alto livello (C)

```
n_maschi = n_maschi + nuovoMaschio
n_femmine = n_femmine + nuovaFemmina
n_personePerEta = n_persone[eta]
```

Programma in linguaggio assembly (MIPS)

```
add $2, $2, $4
add $3, $3, $2
lw $15, 4($2)
```


Esempio di compilazione da C ad assembly (MIPS)

- Si consideri il seguente segmento di programma C che utilizza 5 variabili (f, g, h, i e j):

```
f = (g + h) – (i + j)
```

- Il compilatore associa ad un'istruzione C complessa delle istruzioni assembly a tre operandi e introduce due variabili temporanee (t0 e t1)

```
add t0, g, h       # var. temp t0 ⇐ g + h
add t1, i, j       # var. temp. t1 ⇐ i + j
sub f, t0, t1      # t2 ⇐ t0 – t1
```

- Le istruzioni assembly sono una rappresentazione simbolica del linguaggio macchina comprensibile dal processore MIPS.

Sommario

- Architettura di riferimento dei calcolatori
- Esecuzione delle istruzioni
- Il linguaggio assembly
- Il linguaggio macchina
- Insieme delle istruzioni
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

Linguaggio macchina

- Linguaggio di programmazione direttamente comprensibile dalla macchina
  - Alfabeto binario (alfabeto posizionale)
  - Parole sono le *istruzioni*
  - Vocabolario è *l'insieme delle istruzioni (instruction set)*

Linguaggio macchina

- Ogni architettura di processore ha il suo linguaggio macchina
  - Architettura definita dall'insieme delle istruzioni
    - **ISA (Instruction Set Architecture)**
  - Due processori con lo stesso linguaggio macchina hanno la stessa architettura anche se le implementazioni hardware possono essere diverse
  - Consente di accedere direttamente all'hardware di un calcolatore

Fase di compilazione da assembly a linguaggio macchina

Programma in linguaggio assembly (MIPS)

```
add $2, $4, $2
add $3, $3, $2
lw $15, 4($2)
```

Assembler

Programma in linguaggio macchina

```
011100010101010
000110101000111
000010000010000
001000100010000
```


Programma in linguaggio ad alto livello (C)

```
a = a + c
b = b + a
var = m [a]
```

Compilatore

Programma in linguaggio assembly (MIPS)

```
add $2, $4, $2
add $3, $3,$2
lw $15, 4($2)
```

Assemblatore

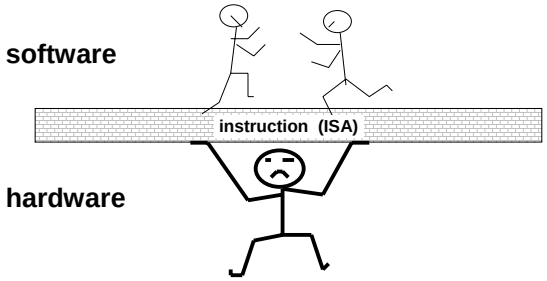
Programma in linguaggio macchina

```
011100010101010
000110101000111
000010000010000
001000100010000
```


Sommario

- Architettura di riferimento dei calcolatori
- Esecuzione delle istruzioni
- Il linguaggio assembly
- Il linguaggio macchina
- Insieme delle istruzioni (Innstruction Set Architecture, ISA)
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

Insieme delle istruzioni



Quale è più facile modificare?

Insieme delle istruzioni (Instruction Set)

- Le istruzioni comprese nel linguaggio macchina di ogni calcolatore possono essere classificate nelle seguenti quattro categorie:
  - Istruzioni aritmetico-logiche;
  - Istruzioni di trasferimento da/verso la memoria *(load/store);*
  - Istruzioni di salto condizionato e non condizionato per il controllo del flusso di programma;
  - Istruzioni di trasferimento in ingresso/uscita (I/O).

Istruzioni aritmetiche

- Ogni istruzione aritmetica ha **un numero prefissato di** operandi (generalmente tre)
- L'ordine degli operandi **è fisso:**
  - Prima il registro contenente il risultato dell' operazione (registro destinazione) e poi i due operandi (registri sorgente)
  - In alcuni casi (es. moltiplicazione e divisione non floating point) il registro destinazione è implicito.

Istruzioni di trasferimento da/verso la memoria *(load/store)*

- Per eseguire un'istruzione, essa deve essere trasferita dalla memoria alla *CPU*.
- Operandi e risultati delle istruzioni devono essere trasferiti tra memoria e *CPU*.
- Necessarie due modalità di trasferimento di dati/istruzioni tra memoria e registri della *CPU*:
  - *load* (caricamento) o *fetch* (prelievo) o *read* (lettura)
  - *store* (memorizzazione) o *write* (scrittura)

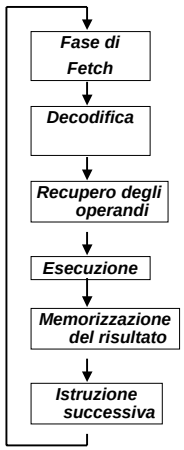
Istruzioni di salto condizionato e incondizionato

- Istruzioni di salto: viene caricato un nuovo indirizzo nel registro contatore di programma (PC) invece dell'indirizzo seguente l'indirizzo di salto secondo l'ordine sequenziale delle istruzioni.
- Istruzioni di salto *condizionato (branch):* il salto viene eseguito solo se una certa condizione risulta soddisfatta.
- Istruzioni di salto *incondizionato (jump):* il salto viene sempre eseguito.

Sommario

- Architettura di riferimento dei calcolatori
- Esecuzione delle istruzioni
- Il linguaggio assembly
- Il linguaggio macchina
- Insieme delle istruzioni (Instruction Set Architecture, ISA)
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

Caratteristiche di un'ISA



° Formato e codifica di un'istruzione
– tipo e dimensione.
° Posizione degli operandi e risultato.
– quanti?
– memoria e/o registri
° Tipo e dimensione dei dati
° Operazioni consentite

- Fetch-decodifica-esecuzione è implicita!

Architetture RISC

- Caratteristiche principali:
  - Istruzioni dell'ISA eseguite direttamente dall'hardware.
  - Massimizzare la frequenza di completamento dell'esecuzione delle istruzioni
  - Istruzioni facili da decodificare
    - "poche", poco "varie", tutte di uguale lunghezza
  - Accesso alla memoria solo attraverso istruzioni dedicate di caricamento/memorizzazione **(load/store)**
    - *Gli operandi di un'istruzione devono sempre risiedere nei registri del processore.*

I registri

- I registri sono associati alle variabili di un programma dal compilatore
- Un processore possiede un numero limitato di registri: ad esempio il processore MIPS possiede **32 registri composti da 32-bit (word)**
- Per convenzione si usano nomi simbolici preceduti da $ per denotare i registri, ad esempio:
  `$s0,$s1,...,$s7($s8)` Per indicare variabili in C
  `$t0, $t1, ... $t9` Per indicare variabili temporanee
- I registri possono essere anche direttamente indicati mediante il loro numero (0, …, 31) preceduto da $: ad es.
  `$0, $1, …, $31`

I registri

- Esistono dei registri *special purpose (32 registri)*
  - per l'esecuzione di alcune operazioni
- Esistono **32** registri per le operazioni floating point (virgola mobile) indicati come
  `$f0, …, $f31`
  - Per le operazioni in doppia precisione si usano i registri contigui
    `$f0, $f2, $f4, …`

Esempio di compilazione da C ad assembly usando registri

- Si consideri il seguente segmento di programma C che utilizza 5 variabili (f, g, h, i e j):
  `f = (g + h) – (i + j)`
- Il compilatore associa alle variabili presenti nel programma i registri presenti nella CPU. Ad es. le variabili (f, g, h, i e j): sono associate ai registri `$s0, $s1, $s2, $s3` e `$s4`
- Il compilatore introduce due variabili temporanee (t0 e t1) che associa a due registri temporanei `$t0` e `$t1`

```
add $t0, $s1, $s2       # var. temp t0 ⇐ g + h
add $t1, $s3, $s4       # var. temp. t1 ⇐ i + j
sub $s0, $t0, $t1       # f ⇐ t0 – t1
```


Formato delle istruzioni MIPS

- Tutte le istruzioni MIPS hanno la **stessa** dimensione **(32 bit)**
- I 32 bit hanno un significato diverso a seconda del formato (o tipo) di istruzione
  - il tipo di istruzione è riconosciuto in base al valore di alcuni bit **(6 bit)** più significativi **(codice operativo - *OPCODE*)**
- Le istruzioni MIPS sono di **3** tipi (formati):
- **Tipo R (register)**
  - Istruzioni aritmetico-logiche
- **Tipo I (immediate)**
  - Istruzioni di accesso alla memoria o contenenti delle costanti
- **Tipo J (jump)**
  - Istruzioni di salto

I tipi di istruzione

- Istruzioni aritmetico-logiche
- Istruzioni di trasferimento dati
- Istruzioni di salto

Istruzioni aritmetico-logiche

- In MIPS, un'istruzione aritmetico-logica possiede *tre* operandi: i due registri contenenti i valori da elaborare *(registri sorgente)* e il registro contenente il risultato *(registro destinazione).*
- L'ordine degli operandi è **fisso:** prima il registro contenente il risultato dell'operazione e poi i due operandi.
- L'istruzione assembly contiene il codice operativo e tre campi relativi ai tre operandi:

`OPCODE DEST, SORG1, SORG2`

Esempio: istruzione add

add serve per sommare il contenuto di due registri sorgente rs e rt:

```
add rd, rs, rt
```

e mettere la somma del contenuto di rs e rt in rd

```
add rd, rs, rt          # rd ← rs + rt
```


Esempio: istruzione add

Codice *C*: **`R = A + B`**

Codice assembler MIPS: **`add $s0, $s1, $s2`**

Nella traduzione da linguaggio ad alto livello a linguaggio assembly, le variabili sono associate ai registri dal compilatore

Esempio: istruzione sub

sub serve per sottrarre il contenuto di due registri sorgente rs e rt:

```
sub rd rs rt
```

e mettere la differenza del contenuto di rs e rt in rd

```
sub rd, rs, rt          # rd ← rs - rt
```


Istruzioni aritmetico-logiche

Il fatto che ogni istruzione aritmetica ha tre operandi sempre nella stessa posizione consente di semplificare l'hw, ma complica alcune cose…

Codice C:

```
A = B + C + D
E = F - A
```

Codice MIPS:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```


Istruzioni aritmetico-logiche

- Operazioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici.
- Ad esempio, per eseguire la somma delle variabili b, c , d ed e nella variabile a servono tre istruzioni :

Codice C:

```
A = B + C + D + E
```

Codice MIPS:

```
add $t0, $s1, $s2
add $t0, $t0, $s3
add $s0, $t0, $s4
```


add: varianti

- **`addi $s1, $s2, 100`** **`#add immediate`**
  - Somma una costante: il valore del secondo operando è presente nell'istruzione come costante e sommata estesa in segno
- **`addu $s0, $s1, $s2`** **`#add unsigned`**
  - Evita overflow: la somma viene eseguita tra numeri senza segno
- **`addiu $s0, $s1, 100`** **`#add immediate unsigned`**
  - Somma una costante ed evita overflow.

Moltiplicazione

- Due istruzioni:
  - **`mult rs rt`**
  - **`multu rs rt`** **`# unsigned`**
- Il registro destinazione è ***implicito.***
- Il risultato della moltiplicazione viene posto sempre in due registri dedicati (special purpose) denominati ***hi (High order word)*** e ***lo (Low order word)***
- La moltiplicazione di due numeri rappresentabili con 32 bit può dare come risultato un numero non rappresentabile in 32 bit

Moltiplicazione

- Il risultato della moltiplicazione si preleva dal registro **hi** e dal registro **lo** utilizzando le due istruzioni:
  - **`mfhi rd`** **`# move from hi`**
    - Sposta il contenuto del registro **hi** nel registro **rd**
  - **`mflo rd`** **`# move from lo`**
    - Sposta il contenuto del registro **lo** nel registro **rd**

L'overflow verrà analizzato in seguito.

Divisione

- Due istruzioni:
  - **`div rs rt`** **`#divide rs per rt`**
  - **`divu rs rt`** **`#unsigned`**
- Come nella moltiplicazione, anche nella divisione il registro destinazione è ***implicito.***
- Il quoziente della divisione è posto nel registro ***lo,*** mentre il resto è posto nel registro ***Hi***

Pseudoistruzioni

- Per semplificare la programmazione, MIPS fornisce un insieme di *pseudoistruzioni*
- Le pseudoistruzioni sono un modo compatto ed intuitivo di specificare un insieme di istruzioni
  - Non hanno un corrispondente 1 a 1 con le istruzioni dell'ISA.
- La traduzione della pseudoistruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore

Esempio

- `move $t0, $t1`
  - `add $t0, $zero, $t1`
- `mul $s0, $t1, $t2`
  - `mult $t1, $t2`
  - `mflo $s0`
- `div  $s0, $t1, $t2`
  - `div $t1, $t2`
  - `mflo $s0`

I tipi di istruzione

- Istruzioni aritmetico-logiche
- Istruzioni di trasferimento dati
- Istruzioni di salto

Istruzioni di trasferimento dati

- Gli operandi di una istruzione aritmetica devono risiedere nei registri
- I registri MIPS sono 32
- Cosa succede ai programmi i cui dati richiedono più di 32 registri (32 variabili)? ⇒ Alcuni dati risiedono in memoria ***(Register Spilling)***

⇓

Servono istruzioni apposite per trasferire dati da memoria a registri e viceversa

Istruzioni di trasferimento dati

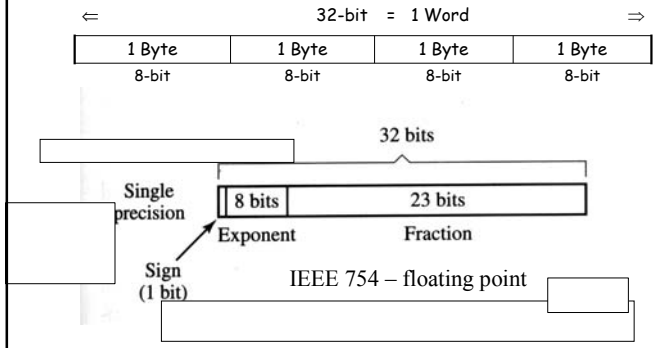
- MIPS fornisce due operazioni base per il trasferimento dei dati:
  - **lw** (load word) per trasferire una parola di memoria in un registro
  - **sw** (store word) per trasferire il contenuto di un registro in una parola di memoria

lw e sw richiedono come argomento l'indirizzo della locazione di memoria sulla quale devono operare

La memoria nel MIPS

- I contenuti delle locazioni di memoria possono rappresentare sia istruzioni che dati.
- La memoria è vista come un unico grande array uni-dimensionale
- Un **indirizzo di memoria** costituisce un indice all'interno dell'array
- MIPS utilizza un **indirizzamento al byte**, cioè l'indice punta ad un byte di memoria
  - byte consecutivi hanno indirizzi consecutivi
  - indirizzi di parole consecutive (adiacenti) differiscono di un fattore 4 (8-bit x 4 = 32-bit).

Indirizzamento dei byte all'interno della parola



Indirizzamento dei byte all'interno della parola

- In genere, la più piccola unità di memoria indirizzabile è il *byte*.
- La lunghezza di parola è *32*-bit ⇒ in una singola parola possono essere disposti *4* byte. In genere, l'indirizzo delle parole viene specificato indicando l'indirizzo del loro *primo byte*.
- Il primo byte all'interno della parola può essere indirizzato secondo due modalità:
  - Disposizione *big-endian* (MIPS, PowerPC, 68000)
  - Disposizione *little-endian* (Intel)

Disposizione big-endian

	Indirizzo di byte			
Parola 0	0	1	2	3
Parola 4	4	5	6	7
Parola 8	8	9	10	11
…				
Parola 2^k-4	2^k-4	2^k-3	2^k-2	2^k-1

- Nella disposizione *big-endian* i byte sono numerati partendo dalla posizione **più** significativa; alla parola viene assegnato lo stesso indirizzo del suo byte più significativo.

Disposizione little-endian

	Indirizzo di byte			Sign bit
Parola 0	3	2	1	0
Parola 4	7	6	5	4
Parola 8	11	10	9	8
…				
Parola 2^k-4	2^k-1	2^k-2	2^k-3	2^k-4

- Nella disposizione *little-endian* i byte sono numerati partendo dalla posizione **meno** significativa; alla parola viene assegnato lo stesso indirizzo del suo byte meno significativo.

Indirizzamento della memoria

- Gli indirizzi di parole adiacenti in memoria differiscono per un fattore quattro (8-bitx4=32-bit)
- In MIPS ogni parola (word) deve iniziare ad un indirizzo multiplo di 4
  - Half word (16-bit) allineate ai multipli di 2
- Per convenzione l'indirizzo di una parola coincide con l'indirizzo del suo byte *più a sinistra (disposizione big-endian)*

Indirizzamento della memoria

0	32 bit
4	32 bit
8	32 bit
12	32 bit

Indirizzo di byte

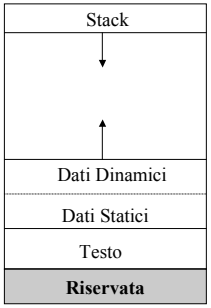
0	1	2	3
4	5	6	7
8	9	10	11
2^k-4	2^k-3	2^k-2	2^k-1

Organizzazione logica della memoria

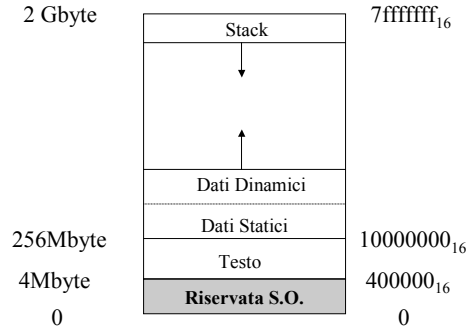
Nei sistemi basati su processore MIPS (e Intel) la memoria è solitamente divisa in **tre** parti:

- **Segmento testo**: contiene le **istruzioni** del programma
- **Segmento dati**: ulteriormente suddiviso in:
  - ***dati statici***: contiene dati la cui dimensione è conosciuta al momento della compilazione e il cui intervallo di vita coincide con l'esecuzione del programma
  - ***dati dinamici***: contiene dati ai quali lo spazio è allocato dinamicamente al momento dell'esecuzione del programma su richiesta del programma stesso.
- **Segmento stack**: contiene lo stack allocato automaticamente da un programma durante l'esecuzione.

Organizzazione logica della memoria



Organizzazione logica della memoria



Istruzioni di trasferimento dati

- MIPS fornisce due operazioni base per il trasferimento dei dati:
  - **lw (load word)** per trasferire una parola di memoria in un registro della CPU
  - **sw (store word)** per trasferire il contenuto di un registro della CPU in una parola di memoria

lw e sw richiedono come argomento l'indirizzo della locazione di memoria sulla quale devono operare

Istruzione *load*

- L'istruzione di *load* trasferisce una copia dei dati/istruzioni contenuti in una specifica locazione di memoria ai registri della *CPU*, lasciando inalterata la parola di memoria:

```
load LOC, r1          # r1 ← [LOC]
```

- La *CPU* invia l'indirizzo della locazione desiderata alla memoria e richiede un'operazione di lettura del suo contenuto.
- La memoria effettua la lettura dei dati memorizzati all'indirizzo specificato e li invia alla *CPU*.

Istruzione di *store*

- L'istruzione di *store* trasferisce una parola di informazione dai registri della *CPU* in una specifica locazione di memoria, sovrascrivendo il contenuto precedente di quella locazione:

```
store r2, LOC          # [LOC] ← r2
```

- La *CPU* invia l'indirizzo della locazione desiderata alla memoria, assieme con i dati che vi devono essere scritti e richiede un'operazione di scrittura.
- La memoria effettua la scrittura dei dati all'indirizzo specificato.

Istruzione lw

- Nel MIPS, l'istruzione **lw** ha tre argomenti:
  - il *registro destinazione* in cui caricare la parola letta dalla memoria
  - una costante o *spiazzamento* (*offset)*
  - un registro base (*base register*) che contiene il valore dell'indirizzo base *(base address)* da sommare alla costante.
- L'indirizzo della parola di memoria da caricare nel registro destinazione è ottenuto dalla somma della costante e del contenuto del registro base.

Memorizzazione di un vettore

Indirizzo di byte

A[0]	0	1	2	3
	4	5	6	7
offset	8	9	10	11
	2^k-4	2^k-3	2^k-2	2^k-1

Istruzione lw: trasferimento da memoria a registro

```
lw $s1, 100($s2)     # $s1 ← M[$s2 + 100]
```

Al registro destinazione $s1 è assegnato il valore contenuto all'indirizzo di memoria ($s2 + 100)

Istruzione sw: trasferimento da registro a memoria

- Possiede argomenti analoghi alla lw

Esempio:

```
sw $s1, 100($s2)        # M[$s2 + 100] ← $s1
```

Alla locazione di memoria di indirizzo ($s2 + 100) è assegnato il valore contenuto nel registro $s1

lw & sw: esempio di compilazione

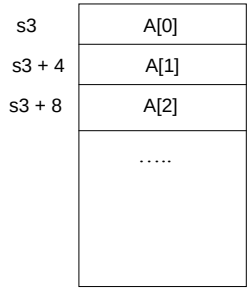
Codice C: `A[12] = h + A[8];`

- Si suppone che:
  - la variabile **h** sia associata al registro **$s2**
  - l'indirizzo del primo elemento dell'array *(base address)* sia contenuto nel registro **$s3 (A[0])**

Codice MIPS:

```
lw $t0, 32($s3)         # $t0 ← M[$s3 + 32]
add $t0, $s2, $t0       # $t0 ← $s2 + $t0
sw $t0, 48($s3)         # M[$s3 + 48] ← $t0
```


Array



Array

- L'elemento numero **i-esimo** di un array si troverà nella locazione
  `br + 4 * i` dove:
  - **`br`** è il registro base;
  - **`i`** è l'indice ad alto livello;
  - il fattore **4** dipende dall'indirizzamento al byte della memoria nel MIPS

Array: esempio

- Sia A un array di N word
- Istruzione C: g = h + A[i]
- Si suppone che:
  - le variabili **g, h, i** siano associate rispettivamente ai registri **$s1, $s2,** ed **$s4**
  - l'indirizzo del primo elemento dell'array *(base address)* sia contenuto nel registro **$s3**

Array: esempio indirizzamento

- L'elemento **i-esimo** dell'array si trova nella locazione di memoria di indirizzo **($s3 + 4 * i).**
- Caricamento dell'indirizzo di A[i] nel registro temporaneo **$t1:**

```
muli $t1, $s4, 4        # $t1 ← 4 * i
add $t1, $t1, $s3       # $t1 ← add. of A[i]
                        # that is ($s3 + 4 * i)
```

- Per trasferire A[i] nel registro temporaneo **$t0:**

```
lw $t0, 0($t1)          # $t0 ← A[i]
```

- Per sommare h e A[i] e mettere il risultato in g:

```
add $s1, $s2, $t0       # g = h + A[i]
```


Register Spilling

- I programmi in genere possiedono più variabili dei registri della CPU.
- Il compilatore cerca di mantenere le variabili usate più frequentemente nei registri e le altre variabili in memoria, usando istruzioni di *load/store* per trasferire le variabili tra registri e memoria.
- La tecnica di mettere le variabili meno usate (o usate successivamente) in memoria viene chiamata ***Register Spilling***.

Istruzioni aritmetiche vs. load/store

- Le istruzioni aritmetiche leggono il contenuto di due registri (operandi) , eseguono una computazione e scrivono il risultato in un terzo registro (destinazione o risultato)
- Le operazioni di trasferimento dati leggono e scrivono un solo operando senza effettuare nessuna computazione

I tipi di istruzione

- Istruzioni aritmetico-logiche
- Istruzioni di trasferimento dati
- Istruzioni di salto

Le strutture di controllo

- Queste istruzioni:
  - Alterano l'ordine di esecuzione delle istruzioni:
    - La prossima istruzione da eseguire non è l'istruzione successiva all'istruzione corrente
  - Permettono di eseguire cicli e condizioni
- In assembly le strutture di controllo sono molto semplici e primitive

Istruzioni di salto condizionato e incondizionato

- Istruzioni di salto: viene caricato un nuovo indirizzo nel contatore di programma (PC) invece dell'indirizzo seguente l'indirizzo di salto secondo l'ordine sequenziale delle istruzioni.
- Istruzioni di ***salto condizionato (conditional branch)***: il salto viene eseguito solo se una certa condizione risulta soddisfatta.
- Esempi: **beq** (*branch on equal*) e **bne** (*branch on not equal*)

```
beq r1, r2, L1      # go to L1 if (r1 == r2)
bne r1, r2, L1      # go to L1 if (r1 != r2)
```

- Istruzioni di ***salto incondizionato (unconditional jump):*** il salto viene sempre eseguito.
  Esempi: **j** (jump) e **jr** (jump register) e **jal** (jump and link)

```
j   L1              # go to L1
jr  r31             # go to add. contained in r31
jal L1              # go to L1. Save add. of next
                    # instruction in reg. ra (ad
                    # esempio return address).
```


Esempio if … then

Codice C: **if (i==j) f=g+h;**

- Si suppone che le variabili **f, g, h, i** e **j** siano associate rispettivamente ai registri **$s0, $s1, $s2, $s3 e $s4**

La condizione viene trasformata in codice C in:

if (i != j) goto Etichetta;
f=g+h;
Etichetta:

Codice MIPS:

```
      bne $s3, $s4, Etichetta      # go to Lab1 if i≠j
      add $s0, $s1, $s2            # f=g+h (skipped if i ≠ j)
Etichetta:
```

Esempio if… then … else

Codice C: **if (i==j) f=g+h;**
else f=g-h;

- Si suppone che le variabili **f, g, h, i** e **j** siano associate rispettivamente ai registri **$s0, $s1, $s2, $s3 e $s4**

Codice MIPS:

```
     bne $s3, $s4, Else # go to Else if i≠j
     add $s0, $s1, $s2  # f=g+h (skipped if i ≠ j
     j   End            # go to End
Else: sub $s0, $s1, $s2  # f=g-h (skipped if i = j)
End:
```


Esempio: do … while (repeate)

Codice C:

```
do
  g = g + A[i];
  i = i + j;
while (i != h)
```

- Si suppone che **g** e **h** siano associate a **$s1** e **$s2,** i e **j** associate a **$s3** e **$s4** e che **$s5** contenga il *base address* di **A**.
- Si noti che il corpo del ciclo modifica la variabile **i** ⇒ devo moltiplicare **i** per **4** ad ogni iterazione del ciclo per indirizzare il vettore A.

Esempio: do … while

Codice C modificato:

```
      i = 0;
Ciclo: g = g + A[i];
       i = i + j;
       if (i != h) goto Ciclo;
```

g e **h** -> **$s1** e **$s2**
i e **j** -> **$s3** e **$s4**
A[0] -> **$s5**

Codice MIPS:

```
      add $s3, $zero, $zero
Loop: muli $t1, $s3, 4     # $t1 ← 4 * i
      add $t1, $t1, $s5    # $t1 ← add. of A[i]
      lw $t0, 0($t1)       # $t0 ← A[i]
      add $s1, $s1, $t0    # g ← g + A[i]
      add $s3, $s3, $s4    # i ← i + j
      bne $s3, $s2, Loop   # go to Loop if i ≠ h
```


Esempio: while

Codice C:

```
while (A[i] == k)
   i = i + j;
```

```
Ciclo:  if (A[i] != k) goto Fine;
        i = i + j; goto Ciclo;
Fine;
```

- Si suppone che **i, j** e **k** siano associate a **$s3, $s4, e $s5** e che **$s6** contenga il *base address* di **A**

Esempio: while

Codice C:

Ciclo: if (A[i] != k) goto Fine;
i = i + j; goto Ciclo;
Fine;

i, j e k -> $s3, $s4, e $s5

A[0] -> **$s6**

Codice MIPS:

```
Loop: muli $t1, $s3, 4        # $t1 ← 4 * i
      add  $t1, $t1, $s6      # $t1 ← add. of A[i]
      lw  $t0, 0($t1)         # $t0 ← A[i]
      bne $t0, $s5, Exit      # go to Exit if A[i]≠k
      add $s3, $s3, $s4       # i ← i + j
      j Loop                  # go to Loop
Exit:
```


Registro $zero

- Spesso la verifica di uguaglianza richiede il confronto con il valore 0 ⇒ per rendere più veloce il confronto, in MIPS il registro **$zero** contiene il valore **0** e non può mai essere utilizzato per contenere altri dati.

Strutture di controllo

- Spesso è utile condizionare l'esecuzione di una istruzione al fatto che una variabile sia minore di una altra:
  - **slt $s1, $s2, $s3** **# set on less than**
    - Assegna il valore **1** a **$s1** se **$s2 < $s3**; altrimenti assegna il valore **0**
- Con **slt, beq** e **bne** si possono implementare tutti i test sui valori di due variabili **(=, !=, <, <=, >,>=)**

Esempio

```
if (i < j) then
   k = i + j;
else
   k = i - j;
```

```
   if (i < j)
        t = 1;
   If (t == 0) goto Else;
   k = i + j;
   goto Exit;
Else:   k = i – j;
Exit:
```

```
#$s0 ed $s1 contengono i e j
#$s2 contiene k

      slt $t0, $s0, $s1
      beq $t0, $zero, Else
      add $s2, $s0, $s1
      j Exit
Else: sub $s2, $s0, $s1
Exit:
```

Struttura switch/case

- Può essere implementata mediante una serie di *if-then-else*
- Alternativa: uso di una *jump address table* cioè di una tabella che contiene una serie di indirizzi di istruzioni alternative

Struttura switch/case

```
switch(k) {
   case 0:      f = i + j; break;
   case 1:      f = g + h; break
   case 2:      f = g - h; break;
   case 3:      f = i - j; break;
  default:      break;
}
```


Struttura switch/case

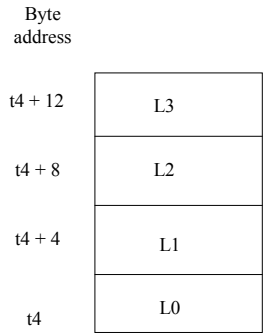
```
if (k < 0)
      t = 1;
else
      t = 0;
if (t == 1)          // k < 0
      goto Exit;
if (k == 0)          // k >= 0
      goto L0;
k--; if (k == 0)     // k = 1;
      goto L1;
k--; if (k == 0)     // k = 2;
      goto L2;
k--; if (k == 0)     // k = 3;
      goto L3;
goto Exit;           // k > 3;
```

```
L0: f = i + j; goto Exit;
L1: f = g + h; goto Exit;
L2: f = g – h; goto Exit;
L3: f = i - j; goto Exit;

Exit:
```


Struttura switch/case

```
#$s0, .., $s5 contengono f,..,k
#$t2 contiene la costante 4

   slt $t3, $s5, $zero
   bne $t3, $zero, Exit    # if k<0
#case vero e proprio
   beq $s5, $zero, L0              L0: add $s0, $s3, $s4
   subi $s5, $s5, 1                    j Exit
   beq $s5, $zero, L1              L1: add $s0, $s1, $s2
   subi $s5, $s5, 1                    j Exit
   beq $s5, $zero, L2              L2: sub $s0, $s1, $s2
   subi $s5, $s5, 1                    j Exit
   beq $s5, $zero, L3              L3: sub $s0, $s3, $s4
   j Exit;                 # if k>3 Exit:
```

Jump address table



Struttura switch/case

```
#$s0, .., $s5 contengono f,..,k
#$t4 contiene lo start address
#della jump address table (che si
#Suppone parta da k = 0).

#verifica prima i limiti (default)
   slt  $t3, $s5, $zero
   bne  $t3, $zero, Exit
   slti $t3, $s5, 4
   beq  $t3, $zero, Exit
#case vero e proprio
   muli   $t1, $s5, 4
   add    $t1, $t4, $t1
   lw     $t0, 0($t1)
   jr     $t0          # j A[k]
```

```
L0: add $s0, $s3, $s4
    j Exit
L1: add $s0, $s1, $s2
    j Exit
L2: sub $s0, $s1, $s2
    j Exit
L3: sub $s0, $s3, $s4
Exit:
```


Le procedure

Registri coinvolti:

$a0, $a1, $a2, $a3 – per il passaggio dei parametri per argomento.

$v0, $v1 – registri per il ritorno di valori calcolati.

$ra – registro di ritorno. Punta a PC + 4.

Esempio di procedura

```
Swap (int v[ ], int k);
{
 int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

```
swap:
muli $a2, $a0, 4
add $t0, $a1, $a2
lw $t1, 0($t0)
lw $t2, 4($t0)
sw $t2, 0($t0)
sw $t1, 4($t0)
jr $ra
```

```
$a0  ->  k
$a1  ->  v[0]
```