Lezione 11

Assembly e Linguaggio Macchina

A. Borghese, F. Pedersini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

Sommario

- Introduzione
- Insieme delle istruzioni (ISA)
- Registri e memoria
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

C:

```
main()
{
   int i;
   int sum = 0;

   for (i = 0; i <= 100; i = i + 1)
              sum = sum + i*i;
}
```

Linguaggio macchina (MIPS)

```
00:     00100111101111011111111111100000
04:     10101111101111110000000000010100
08:     10101111101001000000000000100000
0C:     10101111101001010000000000100100
10:     10101111101001010000000000100100
14:     10101111101001010000000000011000
18:     ........................
```

Assembly (MIPS):

```
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
```


Linguaggio Assembly

- ASSEMBLY: rappresentazione simbolica del linguaggio macchina
  - Vero e proprio linguaggio di programmazione
  - Più comprensibile del linguaggio macchina in quanto utilizza simboli invece che sequenze di bit
- Rispetto ai linguaggi ad alto livello…
  - Assembly è il linguaggio target di compilazione di codice in linguaggi ad alto livello
  - Assembly fornisce limitate forme di controllo del flusso
  - Assembly non prevede articolate strutture dati
- **VANTAGGI:** visibilità diretta dell'hardware
  - → Massimo sfruttamento delle potenzialità HW della macchina
  - → Ottimizzazione delle prestazioni
- **SVANTAGGI:**
  - Mancanza di portabilità dei programmi
  - Maggiore lunghezza e difficoltà di comprensione

Assembly come linguaggio di programmazione

LIMITI:

- Le strutture di controllo hanno forme limitate
- Pochi tipi di dati: **interi , virgola mobile , caratteri**
- Gestione delle strutture dati e delle chiamate a procedura deve essere fatta in modo esplicito dal programmatore

- In alcune applicazioni conviene un *approccio ibrido:*
  - le parti più critiche del programma sono scritte in Assembly (per massimizzare le prestazioni)
  - le altre sono scritte in un linguaggio ad alto livello (prestazioni dipendono dalle capacità di ottimizzazione del compilatore)
  - Esempio: sistemi *embedded*, applicazioni in tempo reale

Fase di compilazione: C ➔ Assembly

- COMPILATORE: sistema "automatico" di traduzione di un codice da linguaggi ad **alto livello** a linguaggio **Assembly**.

Programma in linguaggio ad alto livello (C)

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

Compilatore

Programma in linguaggio assembly (MIPS)

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

Da assembly a linguaggio macchina

- Compilazione effettuata dall' **ASSEMBLER**

Programma in linguaggio Assembly (MIPS)

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

Assembler

Programma in linguaggio macchina

```
01110001010101 0
00011010100011 1
00001000001000 0
00100010001000 0
```


Linguaggio macchina e architettura: ISA

Linguaggio macchina:
il linguaggio di programmazione
direttamente comprensibile dalla macchina

- Ogni architettura di processore ha il proprio linguaggio macchina
  - Architettura definita dall' insieme delle istruzioni – ***Instruction Set***
  - Due processori con la stessa architettura hanno lo stesso linguaggio macchina anche se le implementazioni hardware possono essere diverse

ISA (Instruction Set Architecture)
L'insieme delle istruzioni-macchina eseguibili
da una architettura di processore.

Sommario

- Introduzione
- Insieme delle istruzioni (ISA)
- Registri e memoria
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

L'insieme delle istruzioni (ISA)

- Le istruzioni del linguaggio macchina di ogni calcolatore possono essere classificate in base:
  - alla loro **funzione (Categoria)**
  - al loro **formato (Tipo)**

- **MIPS**: categorie di istruzioni
  1. Istruzioni **aritmetico-logiche**;
  2. Istruzioni di **trasferimento da/verso la memoria**;
  3. Istruzioni di **salto** condizionato e non condizionato per il controllo del flusso di programma;
  4. Istruzioni di **trasferimento in ingresso/uscita** (I/O).
  5. Istruzioni di **controllo**.

Istruzioni logico–aritmetiche

- Ogni istruzione aritmetica ha un numero prefissato di **operandi** (generalmente **tre**)
- **L'ordine** degli operandi **è fisso:**
  - Prima il risultato (registro *destinazione*)
  - Poi i due operandi (registri *sorgente*)
  - In alcuni casi il registro destinazione è implicito (es. moltiplicazione e divisione intera)
- Operandi e risultato sono contenuti nei registri
  - MIPS è un'architettura Load/Store

Esempio:

```
add $1, $4, $5   # somma il contenuto dei registri 4 e 5
                 # e scrivi il risultato nel registro 1
```


Trasferimento da/verso memoria

- Istruzioni di trasferimento a memoria:
- **load** e **store**
  1. Trasferire l'istruzione dalla memoria alla *CPU*
  2. Operandi e risultati delle istruzioni devono essere trasferiti tra memoria e *CPU*
- Necessarie **diverse modalità** di trasferimento di dati/istruzioni tra memoria e registri della *CPU*:
  - **load** (caricamento) o **fetch** (prelievo) o **read** (lettura I/O)

```
lw $1, 0($4)    # MEM[$4] -> $1
```

  - **store** (memorizzazione) o **write** (scrittura I/O)

```
lw $1, 0($4)    # $1 -> MEM[$4]
```

Istruzioni di salto (condizionato e non)

- **Salto:** nel registro **PC** – *Program Counter* viene caricato l'indirizzo di salto, invece dell'indirizzo seguente.
- 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
- Istruzioni di salto indirizzato da registro ***(jump register)***:
  - salto (incondizionato) all'indirizzo contenuto in un registro

MIPS: architettura RISC

- I.S.A. di MIPS: caratteristiche principali
- È un ISA di tipo RISC:
  - Istruzioni dell'ISA eseguite direttamente dall'hardware
    - In alcune CPU, l'istruzione è eseguita attraverso un microprogramma
  - Istruzioni facili da decodificare
    - Poche istruzioni, tutte di uguale lunghezza
    - Accesso alla memoria solo con istruzioni dedicate di caricamento/memorizzazione **`(load/store)`**
    - Gli operandi di un'istruzione si trovano necessariamente nei registri del processore.
  - Massimizzazione della velocità di completamento delle istruzioni

Sommario

- Insieme delle istruzioni – I.S.A.
- Registri e memoria
- Formato delle istruzioni (MIPS)
- Codifica delle istruzioni (MIPS)
- Modalità di indirizzamento

I registri

REGISTRI:

- Ogni processore possiede un certo numero di registri

- **MIPS**: Register File – 32 registri da 32-bit "general purpose"

- Utilizzo dei registri:
  - In Assembly, li utilizzo per memorizzare informazioni (variabili)
  - Un compilatore associa automaticamente le variabili di programma ai registri

- Convenzione di rappresentazione dei registri MIPS
  - I registri possono essere direttamente indicati con il loro numero (0 … 31) preceduto da $:

    `$0, $1, …, $31`
  - Si usano anche nomi simbolici convenzionali, preceduti da $:

    `$s0,$s1,...,$s7` Per memorizzare **variabili "da salvare"**

    `$t0, $t1, ... $t9` Per variabili **temporanee**

I registri MIPS

- Convenzione d'uso dei 32 registri "general purpose"
  - è solo una convenzione!!!

Nome	Numero	Utilizzo
$zero	0	costante zero
$at	1	riservato per l'assemblatore
$v0-$v1	2-3	valori di ritorno di una procedura
$a0-$a3	4-7	argomenti di una procedura
$t0-$t7	8-15	registri temporanei (non salvati)
$s0-$s7	16-23	registri salvati
$t8-$t9	24-25	registri temporanei (non salvati)
$k0-$k1	26-27	gestione delle eccezioni
$gp	28	puntatore alla global area (dati)
$sp	29	stack pointer
$s8	30	registro salvato (fp)
$ra	31	indirizzo di ritorno

Registri MIPS "special purpose"

Altri registri MIPS utilizzati in programmazione:

- Registri *"special purpose",* per operazioni particolari:

- Registro di destinazione per le moltiplicazioni/divisioni

  [hi , lo]

  - Registro doppio: 64 bit
  - Moltiplicazione: **hi: MSW, lo: LSW**
  - Divisione: **hi: resto lo: quoziente**

- 32 registri per le operazioni floating point (virgola mobile)

  $f0, …, $f31

  - Per le operazioni in doppia precisione si usano registri contigui:

    $f0, $f2, $f4,…

Compilazione C ➔ Assembly usando i registri

- Si consideri il seguente segmento di programma C che utilizza 5 variabili:

```
f = (g + h) – (i + j)
```

- Il compilatore associa alle variabili presenti nel programma i registri presenti nella CPU.
  - Ad es: variabili **f, g, h, i** e **j** associate ai registri

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

- Il compilatore introduce due variabili temporanee (t0 e t1) che associa a due registri temporanei `$t0` e `$t1`

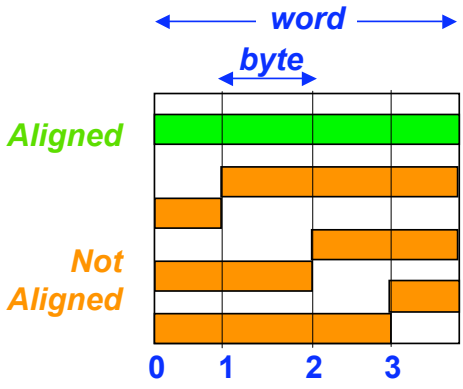
```
add $t0, $s1, $s2    # var. temp. t0 ⇐ g + h
add $t1, $s3, $s4    # var. temp. t1 ⇐ i + j
sub $s0, $t0, $t1    # f ⇐ t0 – t1
```


La memoria nel MIPS

- La memoria è vista come un unico grande
  array uni-dimensionale di bytes/words
  - La memoria contiene sia istruzioni che dati (Von Neumann)
  - **Indirizzo di memoria** → indice all'interno dell'array

- Indirizzamento al byte
  - l'indice punta ad un byte di memoria
- Indirizzamento alla parola
  - La memoria è vista come un array di PAROLE
  - 32 bit: gli indirizzi di parole consecutive differiscono di un fattore 4

- Alignment:
  - L'indirizzamento al byte mi permette di scrivere a cavallo tra 2 parole
  - Le parole devono avere indirizzi MULTIPLI della loro dimensione



"Endianess" e "alignment"

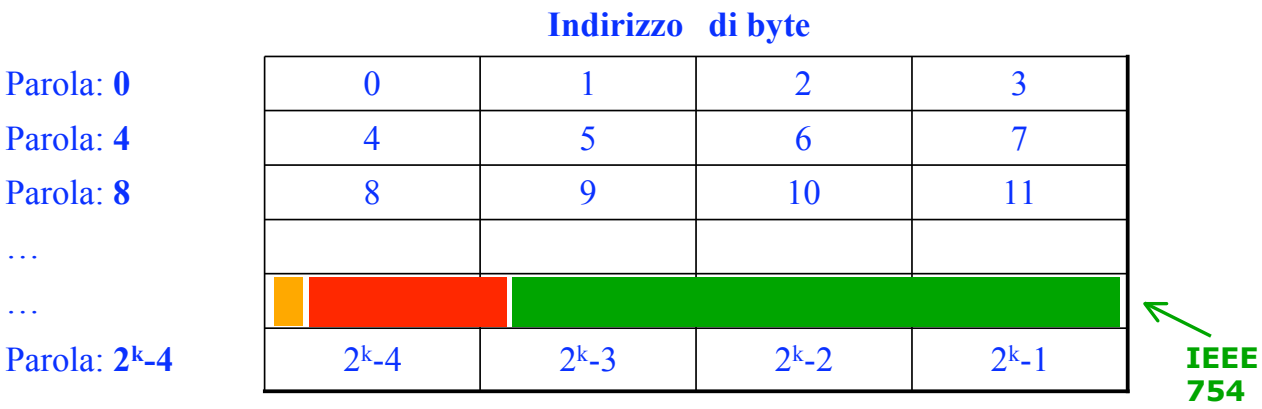
- **Endianess**:
  come si rappresenta un dato su
  più bytes

  Big Endian:
  Word address = address of MSB
  (most significant byte)
  - Motorola 68k, PowerPC, MIPS, Sparc, HP-PA

  Little Endian:
  Word address = address of LSB
  (least significant byte)
  - Intel x86, DEC Vax, DEC Alpha

Disposizione Big-Endian

Big-endian:
- i byte sono numerati partendo dalla posizione **più** significativa;
- indirizzo di parola = indirizzo del suo byte più significativo.



Disposizione Little-Endian

Little-endian:

- i byte sono numerati partendo dalla posizione **meno** significativa;
- indirizzo di parola = indirizzo del suo byte meno significativo.

Indirizzo di byte

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

IEEE 754

Indirizzamento della memoria: MIPS

MIPS memory:
32-bit words – big-endian

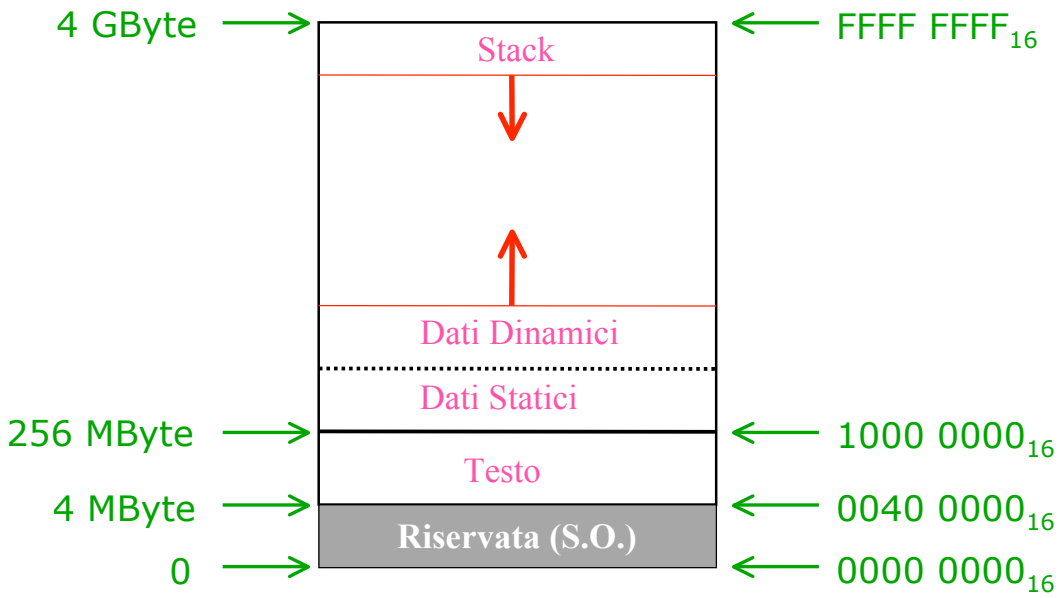
Indirizzo word	Indirizzo byte				
0	0	0	1	2	3
1	4	4	5	6	7
2	8	8	9	10	11
$2^k/4$	2^k	2^K	2^K+1	2^K+2	2^K+3

Organizzazione logica della memoria

- Organizzazione memoria MIPS Intel:
- La memoria associata un programma è divisa in **tre parti**:
  - **Segmento testo**: contiene le **istruzioni del programma**
  - **Segmento dati**: contiene i **dati di elaborazione**
    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

MEMORIA MIPS

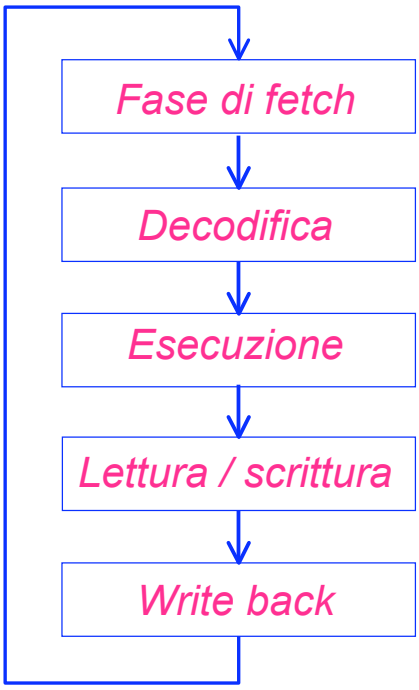


Sommario

- Introduzione
- Insieme delle istruzioni (ISA)
- Registri e memoria
- Formato delle istruzioni
- Codifica delle istruzioni
- Modalità di indirizzamento

Formato di un'istruzione

Ciclo di esecuzione di un'istruzione MIPS



- **Tipo di Istruzione**
  - Categoria funzionale (salto, accesso a memoria, logico-aritmetica)
- **Formato** e **codifica** di un'istruzione
  - Tipo e dimensione istruzione
  - Posizione operandi e risultato
  - Tipo e dimensione dei dati
  - Operazioni consentite

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 dei bit più significativi

(6 bit: codice operativo - *OPCODE*)

- Le istruzioni MIPS sono di 3 tipi – 3 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 MIPS

- Tipi di istruzioni MIPS
  - Istruzioni aritmetico-logiche
  - Istruzioni di trasferimento dati
  - Istruzioni di salto

Istruzioni aritmetico-logiche

- MIPS: un'istruzione aritmetico-logica possiede tre operandi:
  - due registri contenenti i valori da elaborare (**2 registri sorgente**) oppure **1 registro** (1° operando) ed un **numero** (2° operando)
  - un registro che conterrà il risultato (**registro destinazione**)
- L'ordine degli operandi è fisso
  - Prima il registro destinazione, poi i due registri sorgente, in ordine
- Struttura istruzione, in Assembly:
  - codice operativo e tre campi relativi ai tre operandi:

```
OPCODE  DEST, SORG1, SORG2
```

Istruzioni: add, sub, tipo R

- **add**: Addizione

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

  - somma il contenuto di due registri sorgente **rs** e **rt**
  - e mette la somma nel registro destinazione: **rd**

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

- **sub**: Sottrazione

```
sub rd rs rt
```

  - sottrae il contenuto di due registri sorgente **rs** e **rt**
  - e mette la differenza nel registro destinazione **rd**

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

Varianti: unsigned, tipo I

```
addi $s1, $s2, 100      #add immediate
subi $s1, $s2, 100      #sub immediate
```

- Somma/sottrazione di una costante: il valore del secondo operando è presente nell'istruzione come costante (ultimi 16 bit dell'istruzione)

```
addu $s0, $s1, $s2      #add unsigned
subu $s0, $s1, $s2      #sub unsigned
```

- L'operazione viene eseguita tra numeri senza segno

```
addiu $s0, $s1, 100     #add immediate unsigned
subiu $s0, $s1, 100     #sub immediate unsigned
```

- Ilsecondo operando è una costante, senza segno

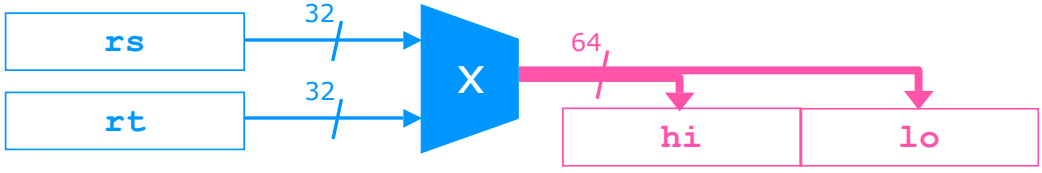
Moltiplicazione

- Assembly MIPS:

```
mult rs rt          # multiply
multu rs rt         # unsigned multiply
```

- La moltiplicazione di due numeri di **32 bit** dà come risultato un numero rappresentabile in **64 bit**
- Il registro destinazione è ***implicito:*** il risultato viene posto sempre in due registri dedicati: **[hi , lo]**

HIGH-order 32-bit word → hi
LOW-order 32-bit word → lo



Moltiplicazione / Divisione

- Il risultato di moltiplicazione / divisione si preleva dal registro **hi** e dal registro **lo** utilizzando le due istruzioni:

```
mfhi rd      # move from hi: rd ← hi
mflo rd      # move from lo: rd ← lo
```

- Il risultato viene trasferito nel registro destinazione specificato

Divisione

- Divisione in MIPS:

```
div rs rt      # division: rs/rt
divu rs rt     # unsigned division
```

- Come nella moltiplicazione, anche nella divisione il registro destinazione è **implicito**, di **dimensione doppia**: [hi , lo]
  - Necessari 64 bit: quoziente (32 bit) + resto (32 bit)
  - Quoziente (intero) della divisione nel registro **lo**
  - Resto della divisione nel registro **hi**

QUOZIENTE (32-bit) → **lo**

RESTO (32-bit) → **hi**

Pseudoistruzioni

- Per semplificare la programmazione, in ogni Assembly vengono definite alcune pseudoistruzioni
  - Significato intuitivo
  - Non hanno un corrispondente 1 a 1 con le istruzioni dell'ISA
- Vantaggi:
  - Parziale standardizzazione del linguaggio Assembly
    - La stessa pseudoistruzione viene tradotta in modi differenti, per architetture (I.S.A.) differenti
  - Rappresentazione più compatta ed intuitiva di istruzioni Assembly comunemente utilizzate
    - La traduzione della pseudoistruzione nelle istruzioni equivalenti viene attuata automaticamente dall' Assembler

Esempi:

Pseudoistruzione:		Codice MIPS:
`move $t0, $t1`	➜	`add $t0, $zero, $t1`
`# t0 ← t1`		
`mul $s0, $t1, $t2`	➜	`mult $t1, $t2`
`# s0 ← t1*t2`		`mflo $s0`
`div $s0, $t1, $t2`	➜	`div $t1, $t2`
`# s0 ← t1/t2`		`mflo $s0`