# Linguaggio assembler e linguaggio macchina (caso di studio: processore MIPS)

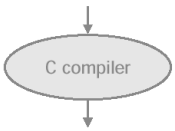
*Salvatore Orlando*



# Livelli di astrazione

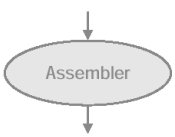
High-level language program (in C)

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




Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```




Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

- **Scendendo di livello, diventiamo più concreti e scopriamo più informazione**
- **Il livello astratto omette dettagli, ma ci permette di trattare la *complessità***

**Quali sono i dettagli che via via scopriamo scendendo di livello?**

# Istruzioni Macchina

- Linguaggio della Macchina
- Più primitivo dei Linguaggi ad Alto Livello
  - es., controllo del flusso poco sofisticato (non ci sono *for, while, if*)
- Linguaggio molto restrittivo
  - es., istruzioni aritmetiche del MIPS sono solo a 3 operandi

- Studieremo l'ISA del MIPS
  - simile ad altre architetture (RISC) sviluppate a partire dagli anni '80
  - usato da NEC, Nintendo, Silicon Graphics, Sony

***Scopi di progetto dell'ISA: massimizzare le prestazioni - minimizzare i costi, anche riducendo i tempi di progetto***



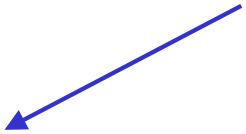
# Istruzioni Aritmetiche del MIPS

- Tutte le istruzioni hanno 3 operandi
- L'ordine degli operandi è fisso
  - l'operando destinazione in prima posizione

Example:

C code: `A = B + C`

Linguaggio Assembler



MIPS code: `add $8, $9, $10`

(operandi associati con variabili dal compilatore)

# Istruzioni Aritmetiche MIPS

- **Principio di Progetto:** ***semplicità favorisce la regolarità***

- **Ma la regolarità può complicare le cose….**

**C code:**

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

**MIPS code:**

```
add $8, $4, $5
add $8, $8, $6
sub $9, $7, $8
```

- **Operandi devono essere registri: solo 32 registri da 4B (1W)**
  - **$0, $1, $2, $3, ….**

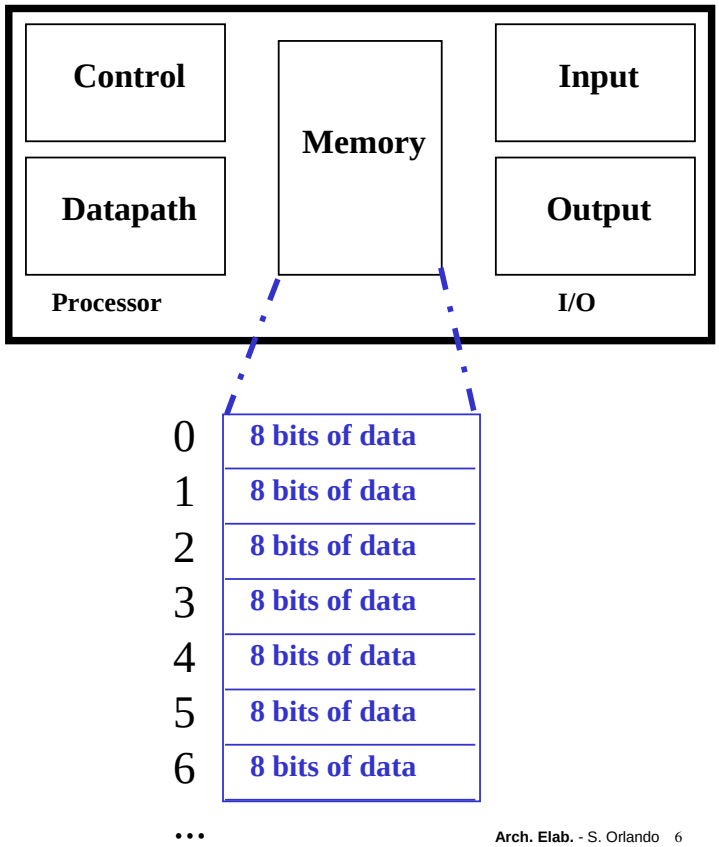
- **Principio di progetto:** ***più piccolo è anche più veloce***



# Registri e Memoria

- **Le istruzioni aritmetiche operano su registri**
  - **solo 32 registri**
  - **ogni registro 1 word (4B)**
- **Compilatore associa variabili con registri**

- **Cosa succede con programmi con tanti dati (tante variabili, o array)?**
  - **Usiamo la memoria, che contiene anche i programmi**
  - **Memoria MIPS *indirizzata al Byte***



# Instruzioni di load / store

- `sw` (Store Word): reg → word in memoria
- `lw` (Load Word): word in memoria → reg

- Esempio:

  C code: `A[8] = h + A[8];`

  MIPS code:

```
lw $15, 32($4)
add $15, $5, $15
sw $15, 32($4)
```

- Indirizzo della word in memoria: $4 + 32
- Nota che `sw` ha la destinazione come ultimo operando
- Ricorda: gli operandi aritmetici sono registri, non celle di memoria !



# Riassumendo

- MIPS
  - — load/store word, con indirizzamento al byte
  - — aritmetica solo su registri

- **Instruzioni** — **Significato**

| | |
|---|---|
| `add $4, $5, $6` | `$4 = $5 + $6` |
| `sub $4, $5, $6` | `$4 = $5 - $6` |
| `lw $4, 100($5)` | `$4 = Memory[$5+100]` |
| `sw $4, 100($5)` | `Memory[$5+100] = $4` |

# Linguaggio Macchina

- Anche le istruzioni sono rappresentati in memoria con 1 word (4B)
  - Esempio: `add $8, $17, $18`
- Formato istruzione (R-type):

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |
|---|---|---|---|---|---|

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|



# Linguaggio Macchina

- Formato istruzioni lw e sw
  - necessario introdurre un nuovo tipo di formato
  - I-type (Immediate)
  - diverso dall'R-type (Register) usato per le istruzioni aritmetico-logiche
- Esempio: `lw $18, 32($9)`

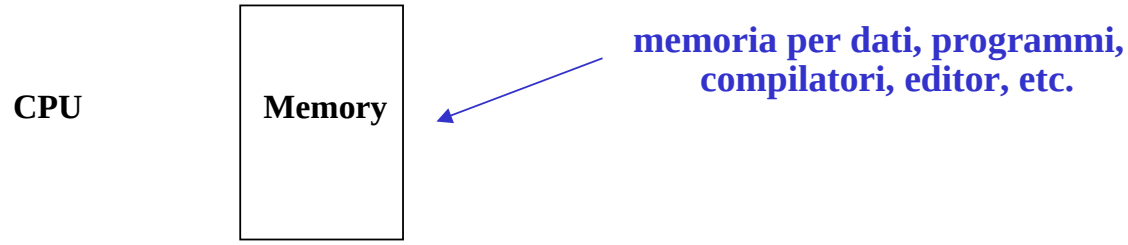
| 35 | 18 | 9 | 32 |
|---|---|---|---|

| op | rs | rt | 16 bit number |
|---|---|---|---|

- Compromesso di progetto
  - anche `lw`/`sw` sono lunghe 4B, con displacement costante (operando immediato di 16 b) inserito direttamente nell'istruzione

# Concetto di “Stored Program”

- Istruzioni sono stringhe di bit
- Programmi, costituiti da sequenze di istruzioni, sono memorizzati in memoria
  - La CPU legge le istruzioni dalla memoria come i dati




- Ciclo Fetch & Execute
  - CPU legge (fetch) istruzione corrente, e la pone in un registro speciale interno
  - CPU usa i bit dell’istruzione per "*controllare*" le azioni susseguenti, e su questa base esegue l’istruzione
  - CPU determina “*prossima*” istruzione e ripete ciclo



# Istruzioni di controllo

- Istruzioni per prendere decisioni sul “futuro”
  - alterano il controllo di flusso (sequenziale)
  - cambiano  quindi la ”prossima" istruzione da eseguire

- Istruzioni MIPS di salto condizionato:

```
beq $4, $5, Label     # branch if equal
bne $6, $5, Label     # branch if not equal
```

- Esempio:

```
if (i==j) h = i + j;
```

```
            bne $4, $5, Label
            add $19, $4, $5
Label:      ....
```

# Istruzioni di controllo

- **Salto non condizionato**

```
j  label
```

- **Esempio:**

```
if (i!=j)                    beq $4, $5, Lab1
    h=i+j;                   add $3, $4, $5
else                         j Lab2
    h=i-j;             Lab1: sub $3, $4, $5
                       Lab2: ...
```



# Riassumendo

- **Istruzione** — **Significato**

| Istruzione | Significato |
|---|---|
| `add $4,$5,$6` | `$4 = $5 + $6` |
| `sub $4,$5,$6` | `$4 = $5 - $6` |
| `lw $4,100($5)` | `$4 = Memory[$5+100]` |
| `sw $4,100($5)` | `Memory[$5+100] = $4` |
| `bne $4,$5,Label` | Prossima istr. letta all'indirizzo Label, ma solo se $s4 ≠ $s5 |
| `beq $4,$5,Label` | Prossima istr. letta all'indirizzo Label, ma solo se $s4 = $s5 |
| `j Label` | Prossima istr. letta all'indirizzo Label |

- **Formati:**

| | | | | | | |
|---|---|---|---|---|---|---|
| R | op | rs | rt | rd | shamt | funct |
| I | op | rs | rt | 16 bit address | | |
| J | op | 26 bit address | | | | |

# Istruzioni di controllo

- **Abbiamo visto: `beq`, `bne`**
  - **ma come facciamo per esprimere Branch-if-less-than?**

- **Nel MIPS c'è un'istruzione aritmetica**
  - **`slt`: Set-if-Less-Than**
  - **`slt` può essere usata in congiunzione con beq e bne**

| Istruzione | Significato |
|---|---|
| `slt $10, $4, $5` | `if $4 < $5 then`<br>`  $10 = 1`<br>`else`<br>`  $10 = 0` |



# Costanti

- **Costanti "piccole" sono molto frequenti (50% degli operandi)**
  **es.: A = A + 5;**
  **B = B + 1;**
  **C = C - 18;**
  - **costanti piccole trovano posto all'interno delle istruzioni come operandi immediati**

- **Istruzioni MIPS aritmetico/logiche con operandi immediati:**

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

- **Formato I**
  - **ancora istruzioni regolari rappresentabili su 32 bit**