Architettura degli Elaboratori

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

slide a cura di Salvatore Orlando, Andrea Torsello, Marta Simeoni

Istruzioni e Linguaggio Macchina

I Linguaggi Macchina sono composti da istruzioni macchina, codificate in binario, con formato ben definito

- processori diversi hanno linguaggi macchina simili
- scopo: massimizzare le prestazioni
  - veloce interpretazione da parte del processore
  - efficace traduzione/compilazione di programmi ad alto livello

Molto più primitivi dei *Linguaggi ad Alto Livello*

- es., controllo del flusso poco sofisticato (non ci sono *for, while, if*)

Linguaggi molto restrittivi

- es., istruzioni aritmetiche con numero fisso di operandi

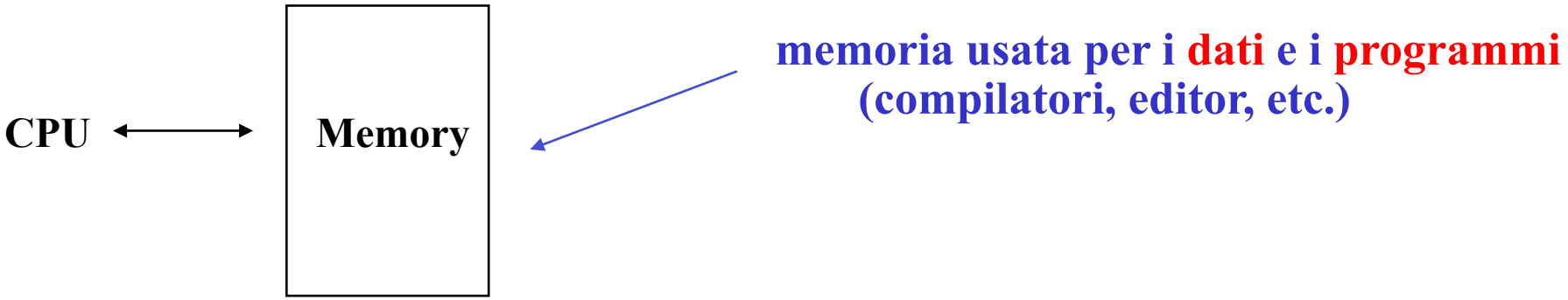
Concetto di “Stored Program”

Istruzioni sono stringhe di bit

Programmi: sequenze di istruzioni

Programmi (come i dati) memorizzati in *memoria*

- La CPU legge le istruzioni dalla memoria (come i dati)



Ciclo macchina (ciclo Fetch – Decode – Execute)

- CPU legge (fetch) l’istruzione corrente (indirizzata dal **PC=Program Counter**), e la pone in un registro speciale interno
- CPU usa i bit dell’istruzione per "*controllare*" le azioni da svolgere, e su questa base esegue l’istruzione
- CPU determina la “*prossima*” istruzione e ripete il ciclo

Livelli di astrazione

Livelli e Linguaggi

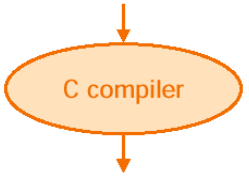
- Linguaggio ad Alto Livello
- Linguaggio Assembler
- Linguaggio Macchina

Scendendo di livello, diventiamo più concreti e scopriamo più informazione

Il livello più astratto omette dettagli, ma ci permette di trattare la *complessità*

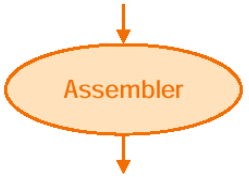
High-level language program (in C)

```
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
```

Istruzioni e Linguaggio Macchina

Il processore che studieremo sarà il MIPS, usato da Nintendo, Silicon Graphics, Sony...

- l'Instruction Set del MIPS è simile a quello di altre architetture RISC sviluppate dal 1980
- le istruzioni aritmetiche del MIPS permettono solo operazioni elementari (add, sub, mult, div) tra coppie di operandi a 32 bit
- le istruzioni MIPS operano su particolari operandi in memoria denominati registri, la cui lunghezza è di 32 bit = 4 Byte = 1 Word

Istruzioni Aritmetiche del MIPS

Tutte le istruzioni hanno 3 operandi

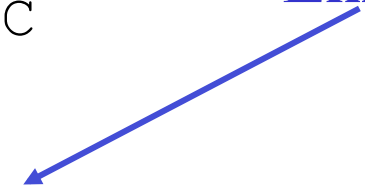
L'ordine degli operandi è fisso

- l'operando destinazione in prima posizione

Esempio:

C code (interi): `A = B + C`

Linguaggio Assembler



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

(variabili associate con i registri 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 = 32 Registri grandi 4B

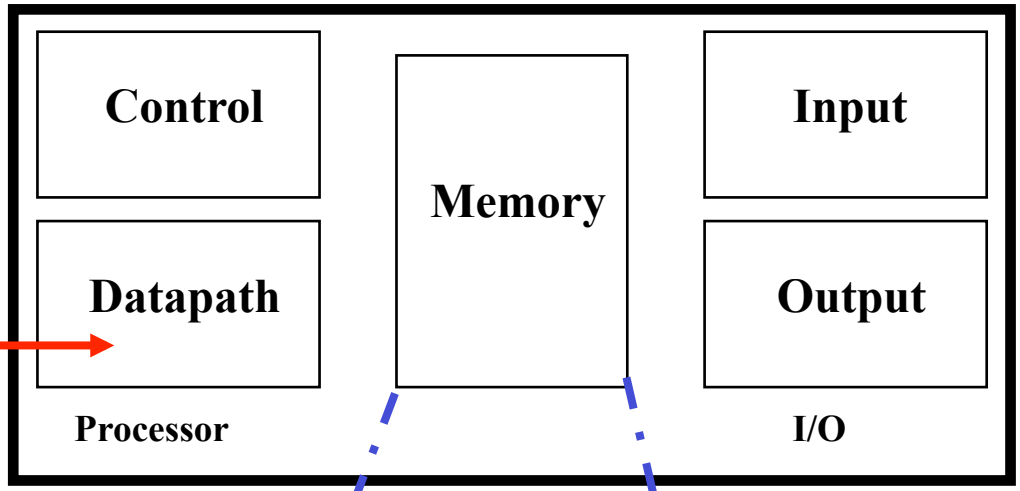
– $0, $1, $2, $3, ….

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

Registri e Memoria

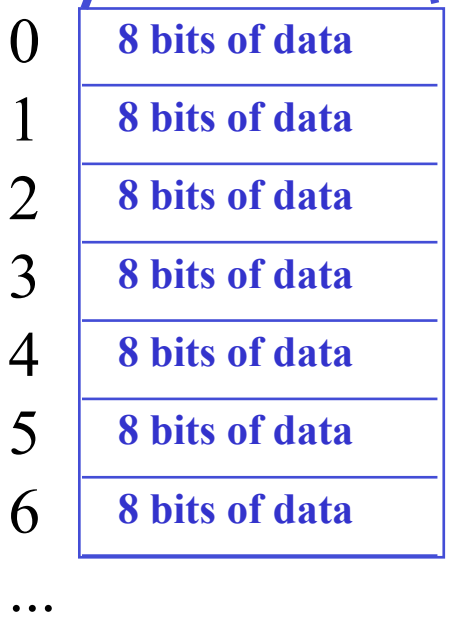
Le istruzioni aritmetiche operano su registri

- Compilatore associa variabili con registri
- Register File fa parte del Datapath del Processore



Cosa succede con programmi con tanti dati (tante variabili, o array)?

- Usiamo la memoria centrale
- 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];`

A array che contiene numeri interi

MIPS code:

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

Indirizzo della word in memoria &A[8]: $4 + 32 ← displacement

Nota che `sw` ha la destinazione come ultimo operando

Ricorda: gli operandi delle istruzioni aritmetiche sono registri, non celle di memoria !

Riassumendo

MIPS

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

Istruzioni	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 rappresentate in memoria con 1 word (4B)

- Esempio: **`add $8, $17, $18`**

Formato R-type (Register type) per istruzioni aritmetico-logiche:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

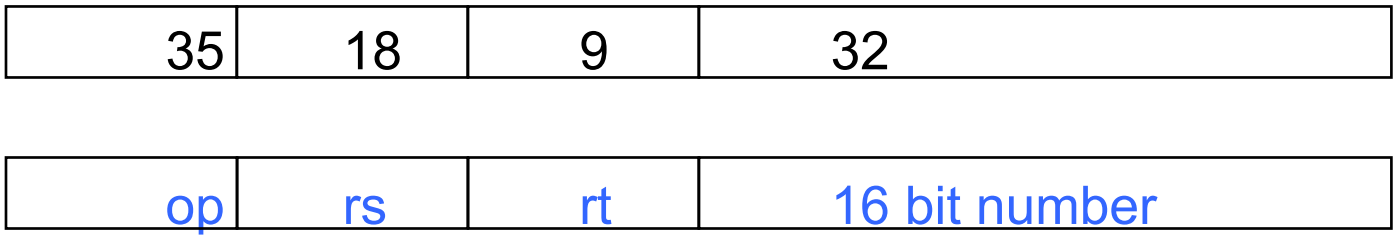
op: operazione base dell'istruzione
rs: registro del primo operando
rt: registro del secondo operando
rd: registro destinazione, che contiene il risultato dell'operazione
shamt: utilizzato per istruzioni di shift; posto a zero per le altre istruzioni
funct: seleziona la specifica variante dell'operazione base definita nel campo op.

Linguaggio Macchina

Formato istruzioni lw e sw

- necessario introdurre un nuovo tipo di formato
- I-type (Immediate type)
- diverso dal formato R-type usato per le istruzioni aritmetico-logiche

Esempio: `lw $9, 32($18)`



Compromesso di progetto

- anche `lw`/`sw` sono lunghe 4B
- displacement nell'istruzione (operando immediato = 2B)
- rt in questo caso è il registro destinazione!

Istruzioni di controllo

Istruzioni per prendere decisioni sul “futuro”

- alterano il controllo di flusso (sequenziale)
- cambiano quindi la ”prossima" istruzione da eseguire (PC)

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:  ....
```

Formato I-type

Istruzioni di controllo

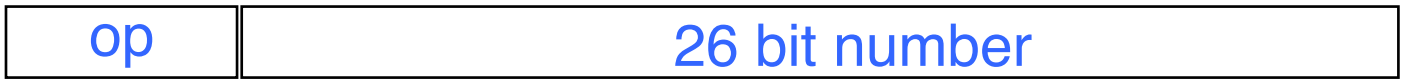
Salto non condizionato

```
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: ...
```

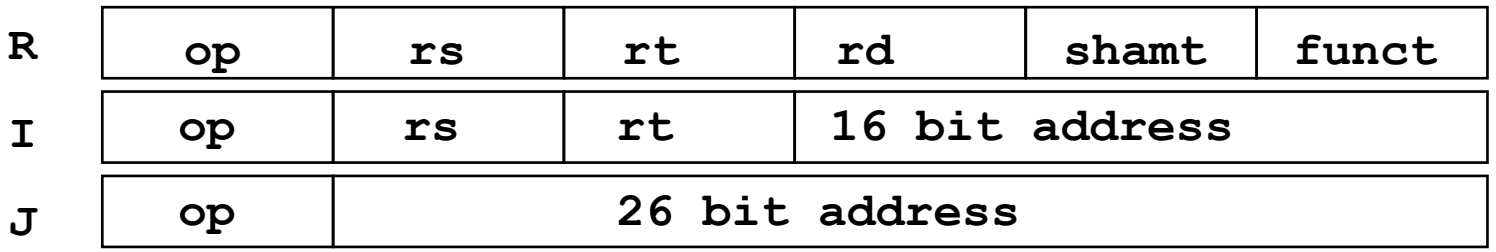
Formato j-type (jump type)



Riassumendo

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`	Se $s4 ≠ $s5, prossima istr. caricata dall'indirizzo Label
`beq $4,$5,Label`	Se $s4 = $s5, prossima istr. caricata dall'indirizzo Label
`j Label`	Prossima istr. caricata dall'indirizzo Label

Formati:



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`
	`  $10 = 1`
	`else`
	`  $10 = 0`

Formato R-type

Costanti

Costanti “piccole” sono molto frequenti (50% degli operandi), e trovano posto all’interno delle istruzioni come operandi immediati

es.: A = A + 5;
B = B + 1;
C = C - 18;

Istruzioni MIPS aritmetico/logiche con operandi immediati:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
slti $3, $1, 5
```

Formato I-type

Riassunto istruzioni MIPS

Istruzioni aritmetico/logiche:

```
add $8, $8, $6
sub $9, $7, $8
slt $10, $4, $5
and $29, $28, $6 (and bit a bit)
or $27, $8, $16  (or bit a bit)
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

Istruzioni di salto:

```
bne $4, $5, Label
beq $4, $5, Label
j Label
```

Istruzioni di lettura/scrittura in memoria:

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

vedere l'elenco completo sul libro di testo!