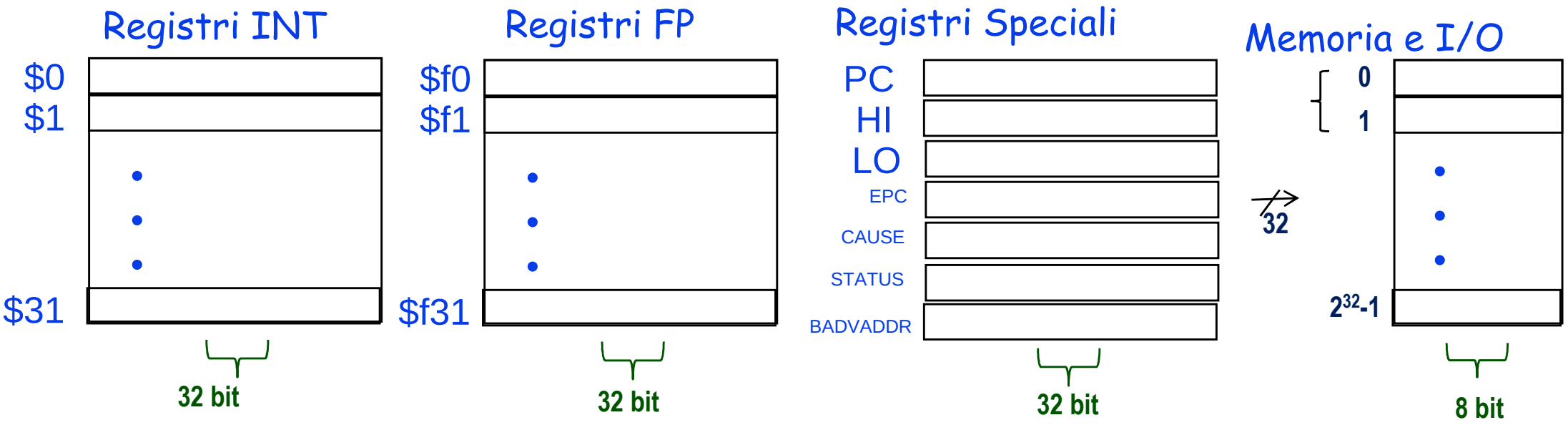
Esercitazione 2
Assembly MIPS (1)

http://www.dii.unisi.it/~giorgi/didattica/arcal1

All figures from Computer Organization and Design: The Hardware/Software Approach, Second Edition, by David Patterson and John Hennessy, are copyrighted material. (COPYRIGHT 1998 MORGAN KAUFMANN PUBLISHES, INC. ALL RIGHTS RESERVED.) Figures may be reproduced only for classroom or personal educational use in conjunction with the book and only when the above copyright line is included. They may not be otherwise reproduced, distributed, or incorporated into other works without the prior written consent of the publisher.

Other material is adapted from CS61C,CS152 Copyright (C) 2000 UCB

Instruction Set Architecture (es. MIPS 3000)



- **Categorie di Istruzioni**
  - Load/Store
  - Calcolo (per Interi e per Frazionari)
  - Jump/Branch
  - Istruzioni speciali

- **Formato delle Istruzioni:**
  - lunghezza FISSA a 32 bit → CPU a 32 bit

Il Set di Istruzioni MIPS

Istruzioni Aritmetiche MIPS (1)

- Tutte le istruzioni aritmetiche hanno 3 operandi
- L'ordine degli operandi e' fisso
  (il primo e' sempre l'operando destinazione)

Es:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

Istruzioni Aritmetiche MIPS (2)

- Principio di progetto:
  "la semplicita' favorisce la regolarita'".
- Naturalmente questo complica qualcosa…

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

→ $s0≡A, $s1≡B, $s2≡C, $s3=D, $s4≡E, $s5≡F

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

- Gli operandi delle istruzioni aritmetiche devono SEMPRE essere registri
- Si hanno solo 32 registri a disposizione

Registri o Memoria

- L'associazione fra variabili e registri viene fatta dal *compilatore*

- Cosa accade se il programma ha moltissime variabili o fa accesso a strutture dati complesse (es. vettori)?

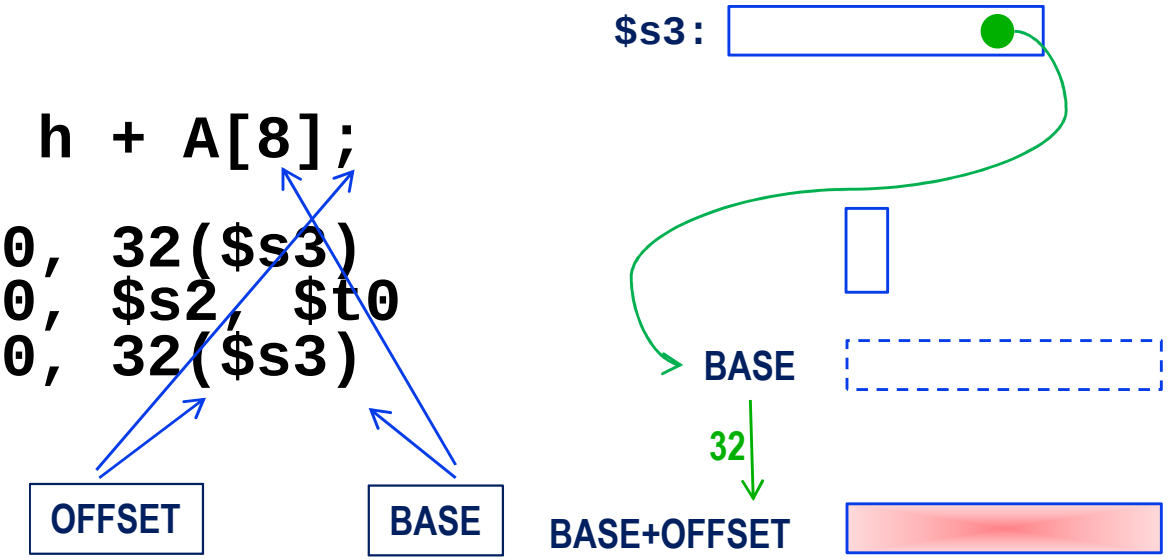
Istruzioni di accesso alla memoria

- Istruzioni Load e Store

  Es.: $s3≡A, $s2≡h

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

MIPS code:   lw  $t0, 32($s3)
             add $t0, $s2, $t0
             sw  $t0, 32($s3)
```



- NOTE:
  - Il dato caricato e' 32 bit (1 word=4 bytes)
  - Nella 'sw' (al contrario delle altre istruzioni con operandi), la destinazione (in memoria) e' l'ultimo operando
  - gli operandi di istruzioni Aritmetiche sono SEMPRE REGISTRI, mai celle di memoria! → IN MIPS, NON ESISTE add $t0, $t1, 32($s3)

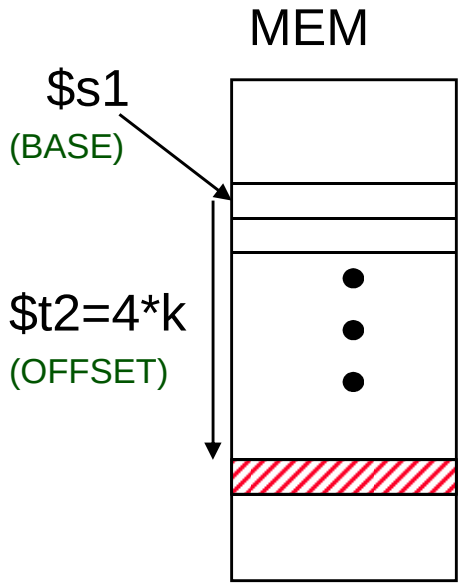
Primo Esempio

- Quali sono le istruzioni eseguite dalla macchina per il seguente frammento di codice C?

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

$s0 ≡ k
$s1 ≡ v

```
swap:
     add  $t2, $s0, $s0
     add  $t2, $t2, $t2
     add  $t3, $s1, $t2
     lw   $t4, 0($t3)
     lw   $t5, 4($t3)
     sw   $t5, 0($t3)
     sw   $t4, 4($t3)
     jr   $ra
```



- `jr $ra` conclude sempre una subroutine (vedremo poi perche'...)

Nota: v e' un vettore, quindi quando (in C) scrivo "v" intendo implicitamente "l'indirizzo base del vettore v"

Regole di allocazione dei registri

Num. registro	Scopo	Nome	invariato su "call"?
0	la costante 0	$zero	n.a.
1	riservato per l'assemblatore	$at	n.a.
2-3	risultati di funzioni ed espressioni	$v0-$v1	no
4-7	argomenti della funzione	$a0-$a3	no
8-15	temporanei (il chiamato li puo' sovrascrivere)	$t0-$t7	no
16-23	temporanei salvati (il chiamato DEVE salvarli**)	$s0-$s7	SI
24-25	ulteriori temporanei (come sopra)	$t8-$t9	no
26-27	riservati per il kernel	$k0-$k1	n.a.
28	global pointer (il chiamato DEVE salvarlo**)	$gp	SI
29	stack pointer (il chiamato DEVE salvarlo**)	$sp	SI
30	frame pointer (il chiamato DEVE salvarlo**)	$fp	SI
31	return address (il chiamato DEVE salvarlo**)	$ra	SI

** nel caso in cui tale registro sia utilizzato

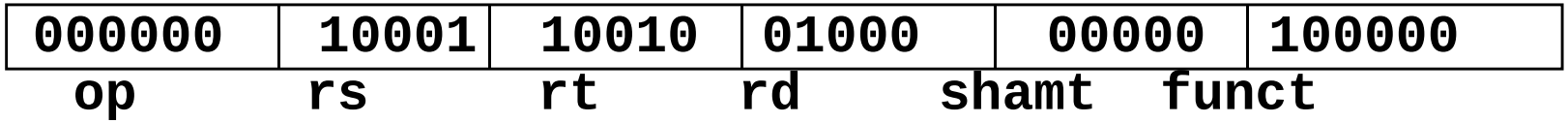
Linguaggio Macchina

- Le istruzioni, come i registri e le word sono a 32 bit

  Es.: `add $t0, $s1, $s2`

  i registri simbolici corrispondono ai registri numerici:
  `$t0≡$8, $s1≡$17, $s2≡$18`

- Formato delle istruzioni:



- *Che cosa significano i nomi dei campi?*
  - *rs = source register*
  - *rt = second source register ("t"="s"+1)*
  - *rd = destination register*
  - *op = (opcode) codice operativo*
  - *shamt = shift amount*
  - *funct = codice funzione (estende op)*

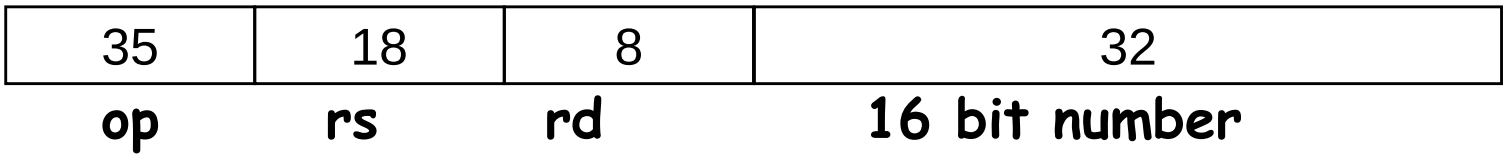
QUESTO FORMATO E' CHIAMATO "R"

Linguaggio Macchina

- Consideriamo le istruzioni load-word e store-word
  - Cosa dovremmo fare in base al "principio di regolarita'"?
  - Nuovo principio: "un progetto ottimo richiede compromessi"

- Introduciamo un nuovo tipo di formato per le istruzioni
  - i registri simbolici corrispondono ai registri: `$t0≡$8, $s2≡$18`

Es.: `lw $t0, 32($s2)`

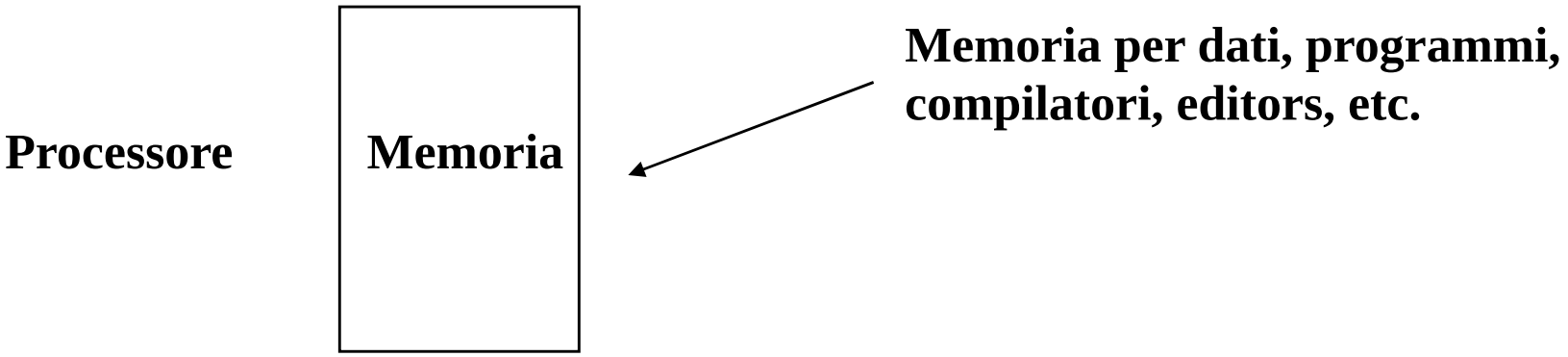


QUESTO FORMATO E' CHIAMATO "I"

Che tipo di compromesso abbiamo fatto?

Caricamento del programma e sua esecuzione

- Anche le istruzioni sono sequenze di bit
- I programmi sono caricati in memoria
  - devono essere letti cosi' come vanno letti i dati



- Fetch & Execute Cycle
  - Le istruzioni sono prelevate (fetched) e messe in un registro interno (non visibile al programmatore)
  - I bit in tale registro "controllano" le azioni successive del processore
  - Si preleva l'istruzione successiva e si continua

Controllo del flusso del programma

- Istruzioni che regolano le decisioni
  - alterano il flusso principale del programma
  - ovvero: cambiano la "prossima istruzione" da eseguire

- Istruzioni di salto condizionale del MIPS

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

Es.: if (b==c) a = b + c;

$s3≡a, $s4≡b, $s5≡c

```
    bne $s4, $s5, Label
    add $s3, $s4, $s5
Label:    ....
```

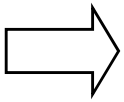
Controllo del flusso del programma

- Istruzioni di salto incondizionato del MIPS

```
j   label
```

Utile per implementare ad es.: $s3≡a, $s4≡b, $s5≡c

```
if (b!=c)
  a=b+c;
else
  a=b-c;
```



```
        beq $s4, $s5, Lab1
        add $s3, $s4, $s5
        j   Lab2
Lab1:   sub $s3, $s4, $s5
Lab2:   ...
```

PER SUPPORTARE UN INDIRIZZO SUFFICIENTEMENTE AMPIO PER "LABEL" VIENE INTRODOTTO UN ULTERIORE FORMATO DI ISTRUZIONE DETTO "J"

op	26 bit address

- *Come si puo' costruire un ciclo "for", un ciclo "while" e uno "do..while"*

```
for(k=0;k<100;++k) {…}
    add $s0,$zero,$zero
ini_for:
  slt $t0,$s0,$s1
  beq $t0,$zero,fin_for
  …
  add $s0,$s0,$s2
  j ini_for
fin_for:
```

```
while(k<100){…}
ini_wh:
 slt $t0,$s0,$s1
 beq $t0,$zero,fin_wh
 …
 j ini_wh
fin_wh:
```

```
do {…} while(k<100);
ini_do:
  …
 slt $t0,$s0,$s1
 bne $t0,$zero,ini_do
```

Controllo del flusso di programma

- Abbiamo: beq, bne ⇨ come si fa un salto su "minore"?

- Nuova istruzione (Set on Less Than):

```
slt $t0, $s1, $s2   ⇔   if  $s1 < $s2 then
                           $t0 = 1
                        else
                           $t0 = 0
```

- Possiamo usare questa istruzione insieme a

```
beq $t0, $zero, Label
```

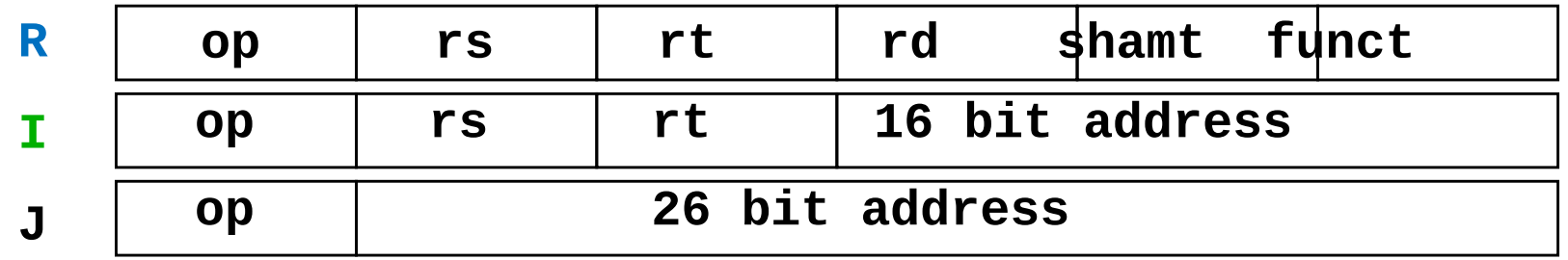
  - possiamo cosi' costruire strutture di controllo con condizioni di salto generiche

Ricapitolazione:

- **Istruzione** **Significato**

```
add $s1,$s2,$s3   $s1 = $s2 + $s3
sub $s1,$s2,$s3   $s1 = $s2 – $s3
slt $t0,$s1,$s2   $t0 = ($s1 < $s2) ? 1 : 0
lw $s1,100($s2)   $s1 = Memory[$s2+100]
sw $s1,100($s2)   Memory[$s2+100] = $s1
bne $s4,$s5,L     Next inst. is at Label if $s4!=$s5
beq $s4,$s5,L     Next inst. is at Label if $s4==$s5
j Label           Next inst. is at Label
```

- Formati:



Linguaggio Assembly e Linguaggio Macchina

- Il linguaggio Assembly fornisce una conveniente rappresentazione simbolica
  - e' piu' comprensibile (a noi) che sequenze di numeri (binari)

- Il linguaggio macchina e' la "realta' sottostante"
  - es. La destinazione non e' il primo operando...

- Il linguaggio Assembly fornisce "pseudoistruzioni"
  - es., "move $t0, $t1" esiste solo in Assembly
  - l'assemblatore traduce usando "add $t0,$t1,$zero"

- Quando le prestazioni diventano un fattore critico e' necessario ricordare quali sono le istruzioni reali

SIMULATORE SPIM