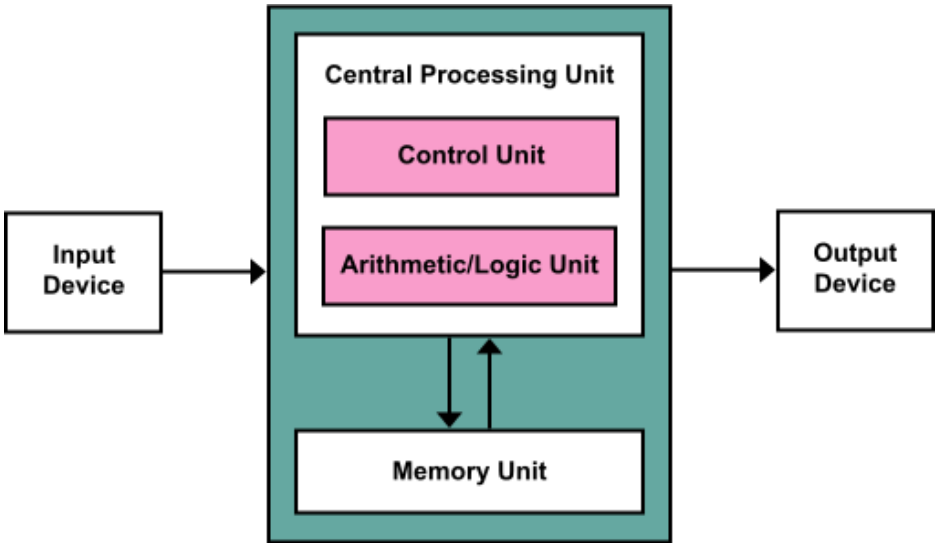
MIPS

Architettura di Von Neuman



La figura è *vintage*, ma da questo punto di vista non è cambiato molto dagli anni '40

- Concetto di *stored program*
  - le istruzioni di un programma sono mantenute in memoria, così come i dati
  - le istruzioni possibili (instruction set) definiscono una delle interfacce più significative di un calcolatore

Instruction Set Architecture

MIPS

- Instruction Set Architecture definita da MIPS Technologies a partire dagli anni '80
- Approccio RISC
  - *Reduced Instruction Set Computing*
  - il formato delle istruzioni è semplice e regolare
  - *Common case fast*
- Tutte le istruzioni MIPS sono lunghe una parola (word)
  - 4 byte, 32 bit
- Tutte le istruzioni MIPS appartengono a uno di tre tipi
  - R, I, J (vedi oltre)

Registri

- Le istruzioni tipicamente operano su dati disponibili in *registri* all'interno del processore
- Esistono istruzioni specifiche per:
  - caricare un registro con un dato presente in memoria
  - salvare il contenuto di un registro in memoria
- MIPS mette a disposizione 32 registri di 32 bit ciascuno

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

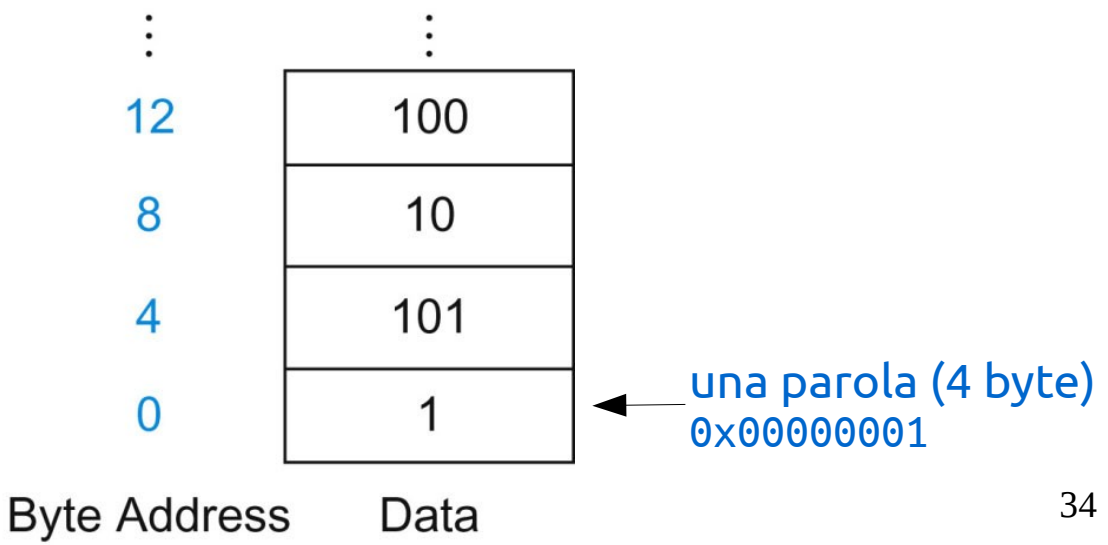
I registri 1 ($at), 26 ($k1) e 27 ($k2) sono riservati

Memoria

- Tutti i dati usati da un programma stanno in memoria, anche sotto forma di strutture dati complesse
- La memoria è una sequenza mono-dimensionale di *locazioni*, ognuna associata a un *indirizzo*
- MIPS permette di indirizzare locazioni di un byte (8 bit), anche se più frequentemente si caricano nei registri parole di memoria, non singoli byte
- Le parole sono allineate in memoria su 4 byte
- Indirizzi a 32 bit

  → 2^{32} byte indirizzabili

  → 2^{30} parole indirizzabili



Esempio

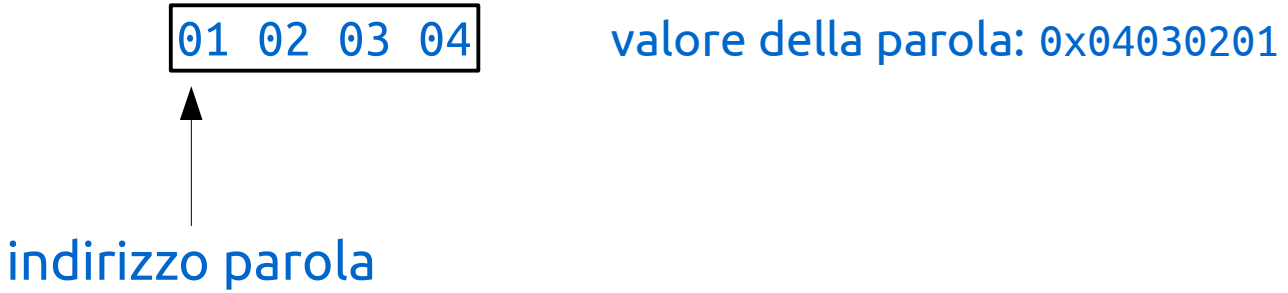
```
r = 0;
s = a - b;
```

```
add $t1, $zero, $zero
lw $t2, 0($s1)          # carica a
lw $t3, 0($s2)          # carica b
sub $t4, $t2, $t3
```

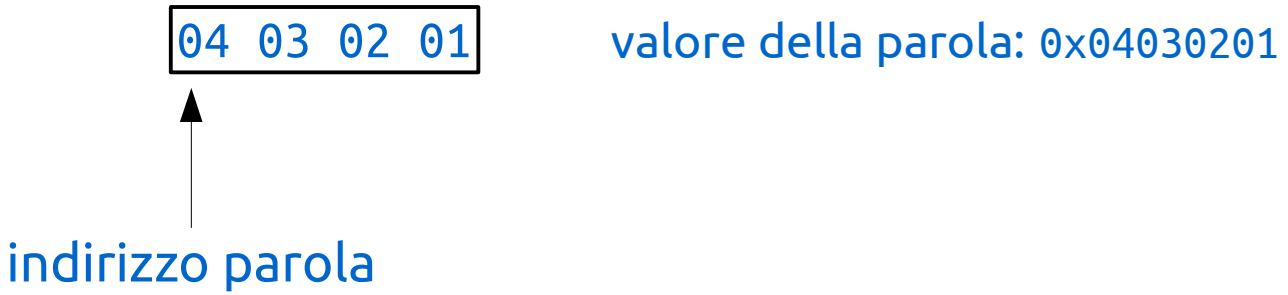
- Le variabili r e s sono mantenute nei registri t1 e t4
- Le variabili a e b stanno in memoria
  - di tipo int → 32 bit (una parola)
  - sono raggiungibili attraverso i registri s1 e s2 (vedi slide successiva)
  - sono caricate nei registri t2 e t3

Little endian e big endian

- Little-endian: l'indirizzo di una parola si riferisce al *least significant byte*



- Big-endian: l'indirizzo di una parola si riferisce al *most significant byte*



- MIPS è big-endian

Istruzioni aritmetiche e logiche
(per numeri interi, con e senza segno)

- Una istruzione aritmetica ha la forma generale

  `op rd, rs, rt`

  con il significato

  `rd = rs op rt`

- `op` corrisponde all'operatore aritmetico/logico
  - add, addu, sub, subu, mult, multu, div, divu, …
  - or, and, nor, …
- rd è il registro destinazione
  - dove va il risultato
- rs e rt sono i registri sorgente
  - termini dell'operazione

Istruzioni aritmetiche e logiche

- Alcune operazioni hanno anche la versione con *immediate operands*

  `op rd, rs, costante`

  con il significato

  `rd = rs op costante`

- `op` corrisponde all'operatore aritmetico/logico
  - addi, addiu
  - ori, andi, sll*, srl*
- rd è il registro destinazione
- rs e `costante` sono i termini dell'operazione
- l'operando `costante` è limitato a 16 bit

* sll e srl hanno un termine *immediate* ma il loro formato è di tipo R, non I (vedi oltre)

La costante zero

- Nella programmazione è frequente l'uso della costante `0`
- MIPS offre un registro in sola lettura ($zero) che contiene quel valore
- Utile per l'inizializzazione di registri (a zero o con il valore di un altro registro)
  - azzera il registro t1

    ```
    add $t1, $zero, $zero
    ```
  - copia il registro s1 nel registro t2

    ```
    add $t2, $s1, $zero
    ```

Addizione

- Signed – add

```
int a = 1, b = -2 , c = 3, d;

d = a + b + c;
```

```
# d ↔ $s1, a … c ↔ $s2 … $s4
add $s1, $s2, $s3 # d = a + b
add $s1, $s1, $s4 # d = d + c
```

- Unsigned – addu

```
unsigned a = 1, b = 2, c = 3, d;

d = a + b + c;
```

```
# d ↔ $s1, a … c ↔ $s2 … $s4
addu $s1, $s2, $s3 # d = a + b
addu $s1, $s1, $s4 # d = d + c
```

Addizione

- Signed immediate – `addi`

```
int a = 1, c = 3, d;

d = a + (-2) + c;
```

```
# d ↔ $s1, a ↔ $s2, c ↔ $s4
addi $s1, $s2,  -2 # d = a + (-2)
add  $s1, $s1, $s4 # d = d + c
```

- Unsigned immediate – `addiu`

```
unsigned a = 1, c = 3, d;

d = a + 2 + c;
```

```
# d ↔ $s1, a ↔ $s2, c ↔ $s4
addiu $s1, $s2,   2 # d = a + 2
addu  $s1, $s1, $s4 # d = d + c
```

Sottrazione

- Signed – sub

```
int a = 1, b = -2 , c = 3, d = 4, e;

e = (a + b) – (c - d);
```

```
# e ↔ $s1, a … d ↔ $s2 … $s5
# t ↔ $t1 temporary
add $s1, $s2, $s3 # e = a + b
sub $t1, $s4, $s5 # t = c - d
sub $s1, $s1, $t1 # e = e - t
```

- Unsigned – subu

```
unsigned a = 2, b = 1, c;

c = a - b;
```

```
# c ↔ $s1, a ↔ $s2, b ↔ $s3
subu $s1, $s2, $s3 # c = a - b
```

Sottrazione

- La sottrazione *immediate* non c'è, si implementa con l'opportuna addizione
  - Signed

```
int a = 1, b;

b = a - 1; /* diventa a + (-1)
```

```
# b ↔ $s1, a ↔ $s2
addi $s1, $s2, -1 # b = a + (-1)
```

  - Unsigned

```
unsigned a = 1, b;

b = a - 1; /* diventa a + (-1)
```

```
# b ↔ $s1, a ↔ $s2
addiu $s1, $s1, -1 # b = a + (-1)
```

Moltiplicazione

- `mul`

```
int a = …, b = …, c;

c = a * b;
```



```
# c ↔ $s1, a ↔ $s2, b ↔ $s3
mul $s1, $s2, $s3  # c = a * b
```

  Salva i 32 bit low-order del risultato

- Ma la moltiplicazione tra due numeri di 32 bit può facilmente eccedere i 32 bit
- L'istruzione MIPS più generica per la moltiplicazione salva il risultato in due registri speciali, ciascuno a 32 bit, chiami *hi* e *lo*
- Esistono poi due istruzioni speciali per recuperare quei valori

Moltiplicazione

- Le istruzioni per la moltiplicazione sono `mult` (signed) e `multu` (unsigned)
- Le istruzioni per recuperare la parte "alta" e la parte "bassa" del risultato sono `mfhi` (*move from hi*) e `mflo` (*move from lo*)
- `mul` (slide precedente) è quindi equivalente* a

```
int a = …, b = …, c;

c = a * b;
```

```
# c ↔ $s1, a ↔ $s2, b ↔ $s3
mult $s2, $s3  # {hi, lo} = a * b
mflo $s1       # c = lo
```

* se non si considera l'overflow

Divisione

- `div` e `divu`
- La divisione intera è in realtà un'operazione che ha due risultati
  - quoziente
  - resto

  `131 / 16 = 8 resto 3`
- Servono due registri per i due risultati
  - `lo` per il quoziente
  - `hi` per il resto

```
int a = …, b = …, q, r;

q = a / b;
r = a % b;
```

```
# q ↔ $s1, r ↔ $s2, a ↔ $s3, b ↔ $s4
div $s2, $s3 # hi = a % b; lo = a / b
mflo $s1     # q = lo
mfhi $s2     # r = hi
```

Operazione OR

- `or` e `ori`
- Utile per includere dei bit in una parola
  - Settarne alcuni a 1 lasciando inalterati gli altri

0000 0000 0000 0000 0000 1101 1100 0000	00 00 0D C0
0000 0000 0000 0000 0011 1100 0000 0000	00 00 3C 00
0000 0000 0000 0000 0011 1101 1100 0000	00 00 3D C0

```
unsigned a = 0xDC0, b = 0x3C00, c1, c2;

c1 = a | b;
c2 = a | 0x3C00;
```



```
# c1 ↔ $s1, c2 ↔ $s2, a ↔ $s3, b ↔ $s4
or  $s1, $s3, $s4      # c1 = a | b
ori $s2, $s3, 0x3C00   # c2 = a | 0x3C00
```

Operazione AND

- `and` e `andi`
- Utile per selezionare solo alcuni bit da una parola, settando gli altri a `0`

0000 0000 0000 0000 0000 1101 1100 0000	00 00 0D C0
0000 0000 0000 0000 0011 1100 0000 0000	00 00 3C 00
0000 0000 0000 0000 0000 1100 0000 0000	00 00 0C 00

```
unsigned a = 0xDC0, b = 0x3C00, c1, c2;

c1 = a & b;
c2 = a & 0x3C00;
```



```
# c1 ↔ $s1, c2 ↔ $s2, a ↔ $s3, b ↔ $s4
and  $s1, $s3, $s4     # c1 = a & b
andi $s2, $s3, 0x3C00  # c2 = a & 0x3C00
```

Operazione NOT

- Non esiste direttamente in MIPS, bisogna usare NOR

  `a NOR b = NOT (a OR b)`

  `NOT a = NOT (a OR 0) = a NOR 0`

- Utile per invertire dei bit in una parola
  - `0 → 1, 1 → 0`

`0000 0000 0000 0000 0000 1101 1100 0000`	`00 00 0D C0`
`1111 1111 1111 1111 1111 0010 0011 1111`	`FF FF F2 3F`

```
unsigned a = 0xDC0, b;

b = ~a;
```

```
# b ↔ $s1, a ↔ $s2
nor $s1, $s2, $zero # b = NOT (a | 0)
```

Operazione di Shift

- Shift Left utile per moltiplicare per 2^n (solo unsigned)
  - `0001 1010 << 3 = 1101 0000`

    $26 \cdot 2^3 = 208$
- Shift Right utile per dividere per 2^n (solo unsigned)
  - `0011 1010 >> 3 = 0000 0111`

    $58 / 2^3 = 7$

```
unsigned a = 26, b;
unsigned c = 58, d;
b = a << 3;
d = c >> 3;
```

```
# b ↔ $s1, a ↔ $s2, d ↔ $s3, c ↔ $s4
sll $s1, $s2, 3 # b = a << 3
srl $s3, $s4, 3 # d = c >> 3
```

Cosa fa?

```
sll  $t0, $s1, 2
sll  $t1, $s1, 3
addu $s2, $t0, $t1
```

Branches

- `if … then … else`, `goto`, `...`
- Salta (branch) a una certa istruzione
  - se una certa condizione è vera, oppure
  - incondizionatamente
- Ogni istruzione, come i dati, può essere identificata tramite l'indirizzo di memoria in cui è tenuta
- In assembly le istruzioni che sono oggetto di un branch possono essere associate a un'etichetta
- L'etichetta è poi convertita nell'indirizzo dell'istruzione corrispondente dall'assembler/linker/loader

Branches

- Branch condizionali
  - branch equal

    `beq rt, rs, L`

    se rt == rs salta a L, altrimenti prosegui sequenzialmente
  - branch not equal

    `bne rt, rs, L`

    se rt != rs salta a L, altrimenti prosegui sequenzialmente
- Branch incondizionato
  - `j L`

    prosegui l'esecuzione all'istruzione con etichetta L

if … then

```
if (i == j) {
  f = g + h;
}
/*L:*/
```

```
   bne $s1, $s2, L   # se i != j salta a L
   add $s3, $s4, $s5 # f = g + h
L: ...
```

```
if (i != j) {
  f = g + h;
}
/*L:*/
```

```
   beq $s1, $s2, L   # se i == j salta a L
   add $s3, $s4, $s5 # f = g + h
L: ...
```

Notare il test invertito

if … then … else

```
if (i == j) {
  f = g + h;
} else {
/*ELSE:*/ f = g - h;
}
/*END:*/
```

```
      bne $s1, $s2, ELSE  # se i != j salta a ELSE
      add $s3, $s4, $s5   # f = g + h
      j END               # salta a END
ELSE: sub $s3, $s4, $s5   # f = g - h
END:  ...
```

```
if (i != j) {
  f = g + h;
} else {
/*ELSE:*/ f = g - h;
}
/*END:*/
```

```
      beq $s1, $s2, ELSE  # se i == j salta a ELSE
      add $s3, $s4, $s5   # f = g + h
      j END               # salta a END
ELSE: sub $s3, $s4, $s5   # f = g - h
END:  ...
```

Altre condizioni

- E <, <=, >, >= (`blt`, `ble`, `bgt`, `bge`)?
  - `beq` e `bne` sono le condizioni più usate; vale la pena ottimizzare l'hardware per i casi più frequenti
- Le altre condizioni sono implementate in termini di beq/bne e di altre istruzioni per fare confronti
  - set less than

    `slt rd, rs, rt`

    se rs < rt, rd = 1, altrimenti rd = 0
  - set less than immediate

    `slti rt, rs, costante`

    se rs < costante, rt = 1, altrimenti rt = 0
  - set less than unsigned – sltu
  - set less than immediate unsigned – sltiu

Altre condizioni

- branch less than

```
if (i < j) {
  f = g + h;
}
/*L:*/
```

```
   slt $t1, $s1, $s2 # t1 = i < j
   beq $t1, $zero, L # se t1 == 0 salta a L
   add $s3, $s4, $s5 # f = g + h
L: ...
```

- branch less equal

```
if (i <= j) {
/*THEN:*/  f = g + h;
}
/*L:*/
```

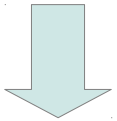
```
      slt $t1, $s1, $s2    # t1 = i < j
      bne $t1, $zero, THEN # se t1 != 0 (i < j) salta a THEN
      bne $s1, $s2, L      # se i != j salta a L
THEN: add $s3, $s4, $s5    # f = g + h
L: ...
```

Signed vs unsigned

- Confronti con segno: slt, slti
- Confronti senza segno: sltu, sltui
- Esempio
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt  $t0, $s0, $s1  # signed`
    - `-1 < +1 → $t0 = 1`
  - `sltu $t0, $s0, $s1  # unsigned`
    - `+4294967295 > +1 → $t0 = 0`

Cicli

```
unsigned s = 0;
unsigned i;
for (i = 0; i != 10; ++i) {
  s += i;
}
```



```
unsigned s = 0;
unsigned i = 0;
unsigned t = 10;
LOOP:
if (i == t) goto END;
s+=i;
++i;
goto LOOP;
END:
```

```
      add $s1, $zero, $zero # s = 0
      add $t0, $zero, $zero # i = 0
      addi $t1, $zero, 10   # t = 10
LOOP: beq $t0, $t1, END     # se i == t vai a END
      add $s1, $s1, $t0     # s += i
      addi $t0, $t0, 1      # ++i
      j LOOP                # vai a LOOP
END:
```

Si trasforma il controllo del ciclo in istruzioni di confronto e branch condizionati e incondizionati

Accesso alla memoria

- Finora abbiamo operato con registri che contenevano già il valore delle variabili (dati) del nostro programma in C
- Ogni dato sta in memoria ad un indirizzo univoco
- Supponendo di conoscere quell'indirizzo...
- Nell'esempio la variabile a è di tipo int e il suo indirizzo in memoria è in $s1

```
++a;
```

```
lw   $t0, 0($s1)  # carica il valore di a nel registro $t0
addi $t0, $t0, 1  # ++a
sw   $t0, 0($s1)  # salva il nuovo valore in memoria
```

Accesso alla memoria

- lw: load word
  - legge una parola (4 byte)
- sw: store word
  - scrive una parola (4 byte)
- Nella notazione `0($s1)`, `$s1` è il base register e rappresenta il base address e `0` rappresenta l'offset
  - l'indirizzo effettivo è quindi: base address + offset
  - utile ad esempio per accedere agli elementi di un array (vedi oltre)

Accesso alla memoria

- Non tutti gli accessi in memoria richiedono di leggere o scrivere parole di 4 byte
  - ad es. nella manipolazione di stringhe
- Anche se l'accesso a singoli byte o half-words sarebbero possibili usando operazioni sui bit, MIPS offre delle istruzioni specifiche
- `lb rt, offset(rs)` `lh rt, offset(rs)`
  - Carica ed estende il segno nel registro rt
- `lbu rt, offset(rs)` `lhu rt, offset(rs)`
  - Carica ed estende a zero nel registro rt
- `sb rt, offset(rs)` `sh rt, offset(rs)`
  - Salva solo il byte/halfword più a destra (meno significativo)

Cicli/Accesso in memoria

- Accesso a un array di char (byte)
  - supponiamo che ci sia almeno un carattere diverso da c nell'array

```
char a[] = "...";

while (a[i] == c) {
  i += 1;
}
```

```
# i ↔ $s3, a ↔ $s6, c ↔ $s5
LOOP: add  $t1, $s3, $s6  # t1 = a + i
      lb   $t0, 0($t1)    # t0 ← MEM[t1]
      bne  $t0, $s5, END  # se t0 != c vai a END
      addi $s3, $s3, 1    # ++i
      j    LOOP           # torna a LOOP
END: …
```

- La prima parte del loop (istruzioni add e lb) servono per “preparare” il test

Cicli/Accesso in memoria

- Accesso a un array di int (word)
  - supponiamo che ci sia almeno un intero diverso da k nell'array

```
int a[] = { … };

while (a[i] == k) {
  i += 1;
}
```

```
# i ↔ $s3, a ↔ $s6, k ↔ $s5
LOOP: sll  $t1, $s3, 2    # t1 = i * 4
      add  $t1, $t1, $s6  # t1 += a
      lw   $t0, 0($t1)    # t0 ← MEM[t1..t1+3]
      bne  $t0, $s5, END  # se t0 != k vai a END
      addi $s3, $s3, 1    # ++i
      j    LOOP           # torna a LOOP
END: …
```

- Ricordarsi di incrementare l'indirizzo di 4 in 4!
- La prima parte del loop (istruzioni sll, add e lw) servono per “preparare” il test