ARCHITETTURA DEL SET DI ISTRUZIONI

Processore MIPS

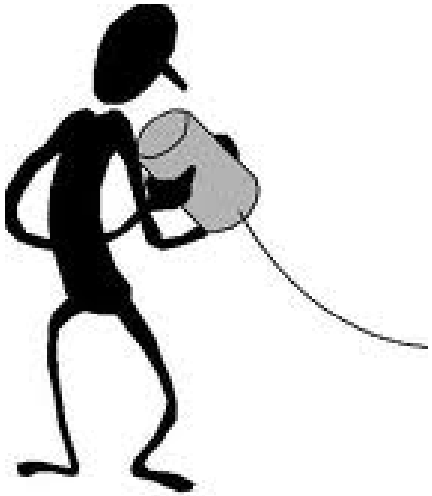
Istruzioni per il trasferimento dati da/verso la memoria

Michele Favalli

Rappresentazione delle istruzioni

- Cotinuiamo a fare riferimento all'architettura di Von Neumann
  - Concetto di istruzione e di dato
  - Ricordiamo che la CPU è un caso particolare (anche se molto utilizzato) di sistema digitale
  - In una CPU realizzata con le correnti tecnologie digitali sia le istruzioni che i dati sono rappresentati con configurazioni binarie

I primi programmatori comunicavano con i computer mediante linguaggio macchina

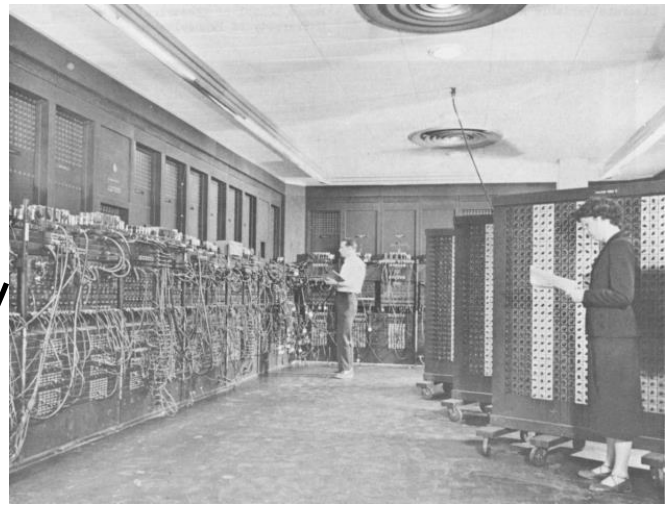


Linguaggio macchina

FUN REG VAL

0010001000000100
0010010000000110
1010000000100100

Istruzioni macchina



ENIAC, 1946

E' molto lontano dal modo con cui pensano gli uomini! Necessità di **notazioni simboliche. *L'assembler (o assembly)* ne fu il primo esempio.**

Assembler

```
li $t1,4
li $t2,6
add $t0,$t1,$t2
```

All'inizio la traduzione fu manuale, poi automatica (assembler)
(curiosità: *si usa la macchina per programmare la macchina!*)
Tuttavia:

- Il programmatore deve ancora specificare una linea simbolica per ogni istruzione-macchina!
- Il programmatore è costretto a pensare come la macchina!

Note sull'assembler

- Il linguaggio assembler ha quasi una corrispondenza 1 a 1 con le istruzioni al livello macchina e quindi con l'Instruction Set Architecture
- Ci sono delle piccole aggiunte che servono ad aiutare il programmatore)
  - macro
  - definizioni di dati
  - ….

L'Intuizione dei linguaggi a livello più alto

Si possono scrivere programmi che traducono linguaggi di programmazione ricchi di «astrazioni» in istruzioni macchina

Linguaggio C

```
/*esempio1.c*/

void main()
{
    int a, b, c;
    a=4;
    b=6;
    c=a+b;
}
```

Compilatore

Linguaggio assembler

```
/*esempio1.s*/

.text
li $t1,4
li $t2,6
add $t0,$t1,$t2
```

Assembler

Linguaggio macchina

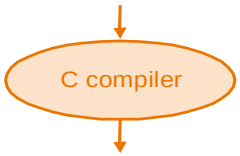
```
FUN REG VAL

00100010000000100
00100100000000110
10100000001001000
```

Ris. Registro $t2:
0000000000001010

Linguaggi di Programmazione di Alto Livello

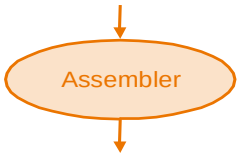
High-level language program (in C)

```
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
```

Permettono ai programmatori di pensare ed esprimersi in un modo più «naturale»

- ✓ Parole in inglese + operazioni aritmetiche o logiche
- ✓ Programma in forma testuale

I linguaggi possono essere creati sulla base delle esigenze applicative

- ✓ ***Fortran*** per il calcolo scientifico;
- ✓ ***Cobol*** per business data processing;
- ✓ ***Lisp*** per l'utilizzo di funzioni, ..

Maggior produttività dei programmatori

- ✓ grazie alla notazione concisa (poche righe di codice)
- ✓ possono «astrarsi» dallo specifico processore che eseguirà il programma (portabilità del codice)

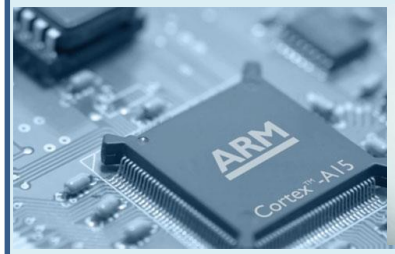
Motivazioni e note

- Si dà per scontata la conoscenza di un linguaggio ad alto livello (C)
- Perché studiare il linguaggio assembler visto che esistono i compilatori?
- Consente di scrivere moduli di codice a prestazioni molto elevate
- Consente di comprendere come funzionano CPU e compilatori (chi non conosce il linguaggio assembly non potrà mai scrivere un intero compilatore)

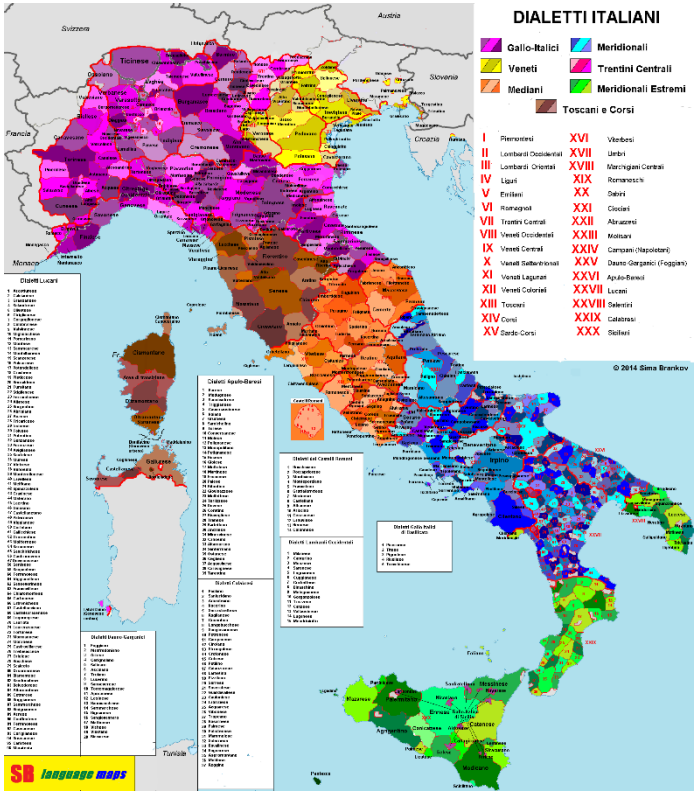
Set di Istruzioni

Per dare comandi ad un microprocessore, occorre parlare la sua lingua!

Lettere dell'alfabeto: **«0» e «1»**
Parole: **istruzioni**
Vocabolario: **set di istruzioni**



Il set di istruzioni non è lo stesso per tutti i microprocessori, ma l'analogia non è quella tra lingue diverse (es., italiano e cinese), ma tra dialetti diversi della stessa lingua.



Per comodità, verrà utilizzata la notazione simbolica in linguaggio assembler.
Gli esempi saranno tratti dal set di istruzioni «MIPS»

Obiettivo

Trovare un set di istruzioni che renda semplice costruire
- sia l'hardware che lo processa
- sia il compilatore lo supporta
massimizzando la performance e minimizzando il costo

ARITMETICA

<<There must certainly be instructions for performing the fundamental arithmetic operations>>
(Burks, Goldstine, von Neumann, 1947)

- **add** *a, b, c*
  somma il contenuto delle «variabili» *b* e *c*, e mettilo nella variabile *a*!

- La definizione delle istruzioni è molto **rigida a scapito della flessibilità**:
  - Non è possibile sommare 4 variabili (b,c,d,e) con un'unica istruzione! Soluzione:

- **add** a, b, c
- **add** a, a, d
- **add** a, a, e

Paghi 3
Prendi 1

- **Si potrebbero creare istruzioni più flessibili? Si, ma a scapito dell'incremento di complessità nella progettazione hardware.**

Principio di progettazione:
la semplicità favorisce la performance

Il Compilatore al Lavoro

Le istruzioni in un linguaggio di programmazione di alto livello come il C vengono **trasformate in istruzioni assembler dal programma «compilatore»**

VINCOLO:

Ogni istruzione assembler può effettuare una sola operazione

ESITO:

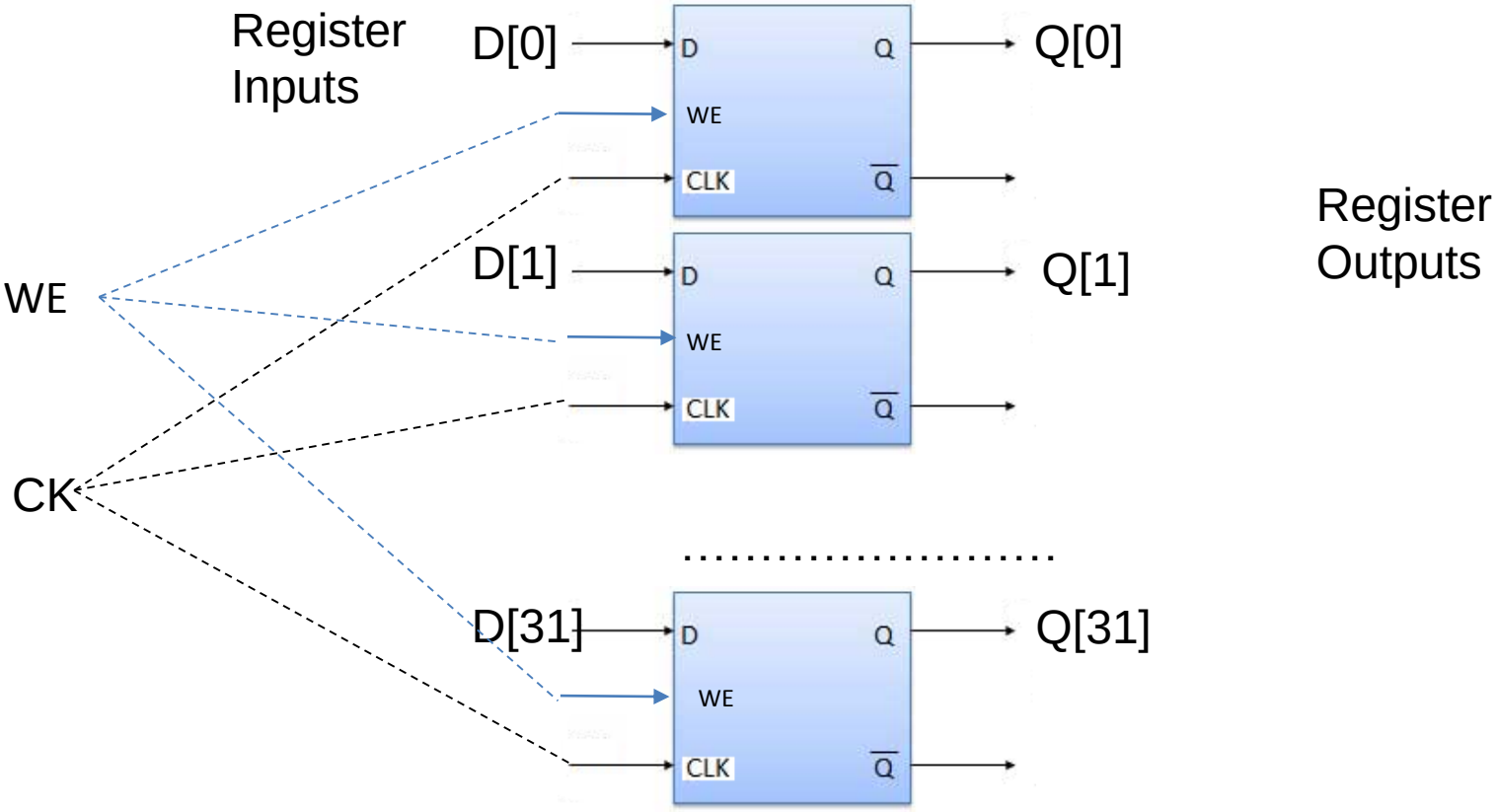
moltiplicazione a valanga delle istruzioni ASM in corrispondenza di «statement» di alto livello complessi

Linguaggio C: **f= (g + h) – (i +j)**

```
add t0, g, h   # somma g ed h nella variabile temporanea t0
add t1, i, j   # somma i e j nella variabile temporanea t1
sub  f, t0, t1 # sottrazione e produzione del risultato in f
```

In realtà nell'assembler le variabili corrispondono ai registri della CPU

Registri



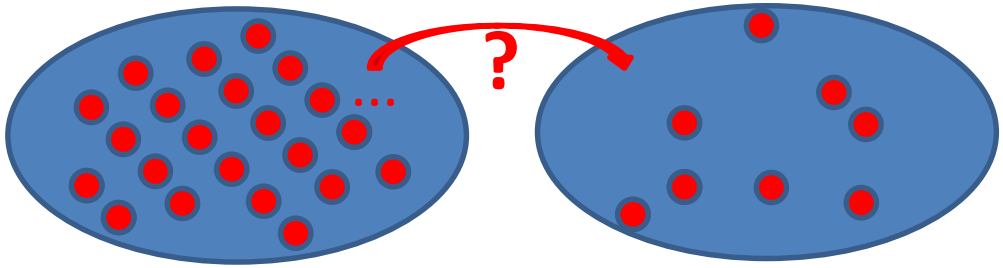
Array di *n* di flip flop di tipo D con WE

I Registri

- Rispetto a una memoria possono essere letti e scritti in maniera estremamente rapida
- Hanno due parametri significativi
  1. Il numero di bit che compongono il registro (*n=32, n=64)*
  2. Il numero finito e molto ridotto di registri (32 nell'architettura MIPS)

Differenza con le «variabili» usate dal programmatore:

Linguaggio di alto livello **Linguaggio assembler**



Potenzialmente infinite (limitate dalla memoria fisica disponibile)

In numero limitato e fisso

Principio di progettazione:

Pochi registri permettono di:

- **raggiungere frequenze di clock più elevate**
- **minimizzare il numero dei bit che codificano le istruzioni**

Mismatch fra registri e memoria

Principio di progettazione:

I registri sono un piccolo pezzo di memoria, per di più vicino al microprocessore => l'accesso ai registri avviene più rapidamente rispetto all'accesso alla memoria di massa per due motivi:

1- memorie più grandi sono anche più lente!

2- la latenza di accesso è inferiore grazie alla vicinanza

Al livello circuitale tecnologia dei registri è diversa da quella delle memorie

Registri vs. variabili

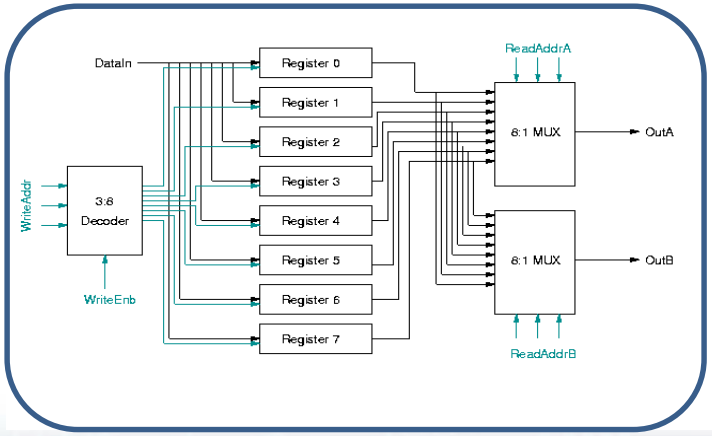
Variabili di un programma

Variabili usate di frequente, oppure variabili di prossimo utilizzo

Memoria RAM

«Register File» del micropr.

1. **Numero limitato di registri => spesso le variabili memorizzate nei registri devono essere salvate in memoria (spilling) per fare spazio a nuove variabili**
2. **I valori delle variabili devono essere reperiti in memoria e scritti in registri prima che un istruzione le usi come operandi**



Mapping dei Registri

Gli operandi delle istruzioni aritmetiche del microprocessore MIPS DEVONO essere scelti tra i 32 registri a 32 bit della architettura:
$s0, $s1,… per memorizzare variabili dei programmi di alto livello
$t0, $t1, … da usarsi come registri temporanei

Linguaggio C
f= (g + h) – (i +j)

Associare le variabili (g,h,i,j,f) ai registri è **compito del compilatore**

```
add  $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

```
# registro $t0 contiene g+h
# registro $t1 contiene i+j
# registro $s0 contiene f
```

Questo è il vero *assembler*, perché indirizzo esplicitamente i registri del processore, non le variabili del linguaggio di programmazione di alto livello!

Instruzioni di Load e Store

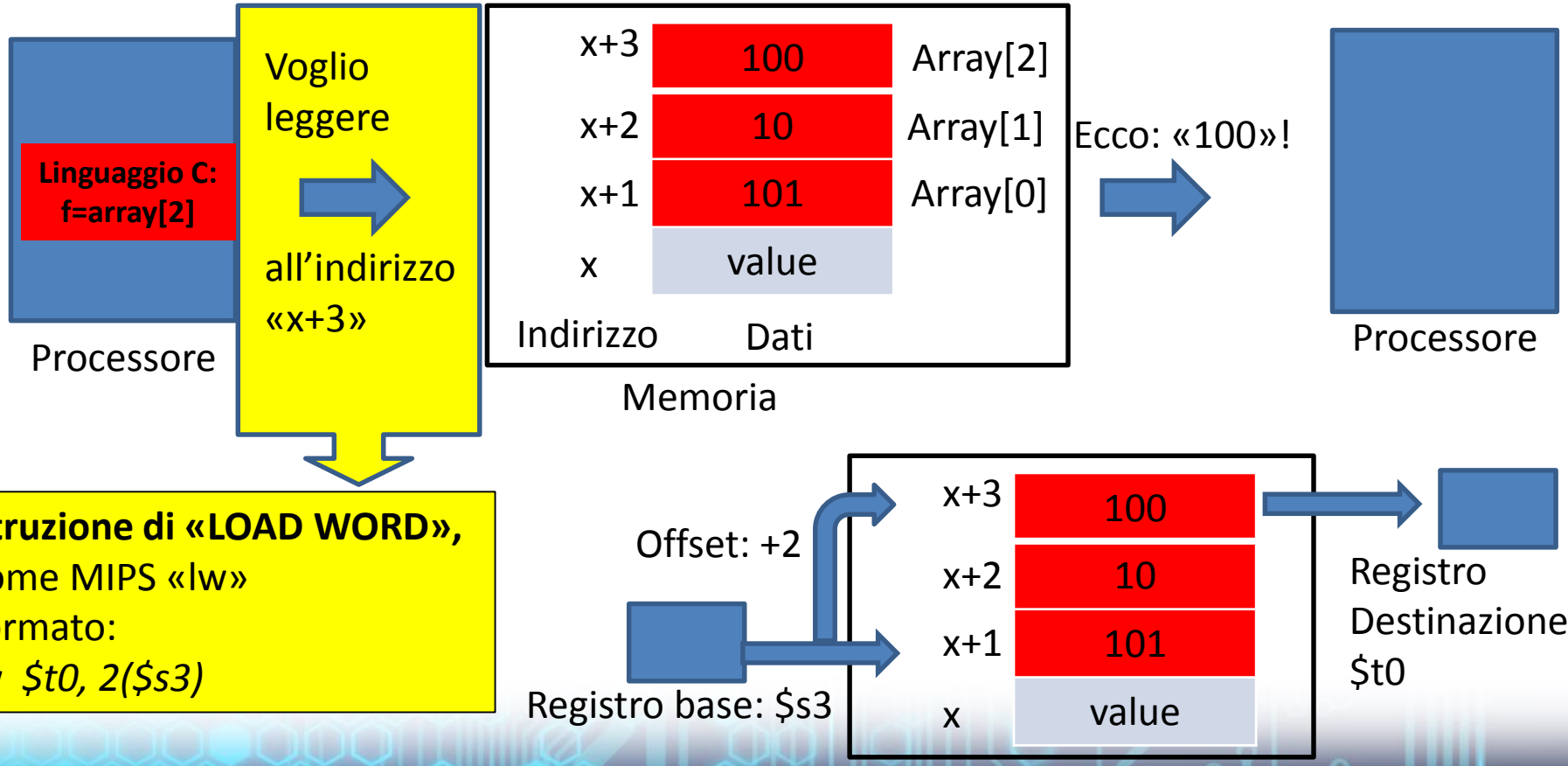
*Solo un numero limitato di variabili è memorizzato nei registri del processore in un certo istante. Il resto delle variabili si trova in **memoria** **(tipicamente, in memoria RAM).***

Istruzioni per il trasferimento dei dati «da» e «verso» la memoria

- **Letture in memoria: LOAD**
- **Scritture in memoria: STORE**

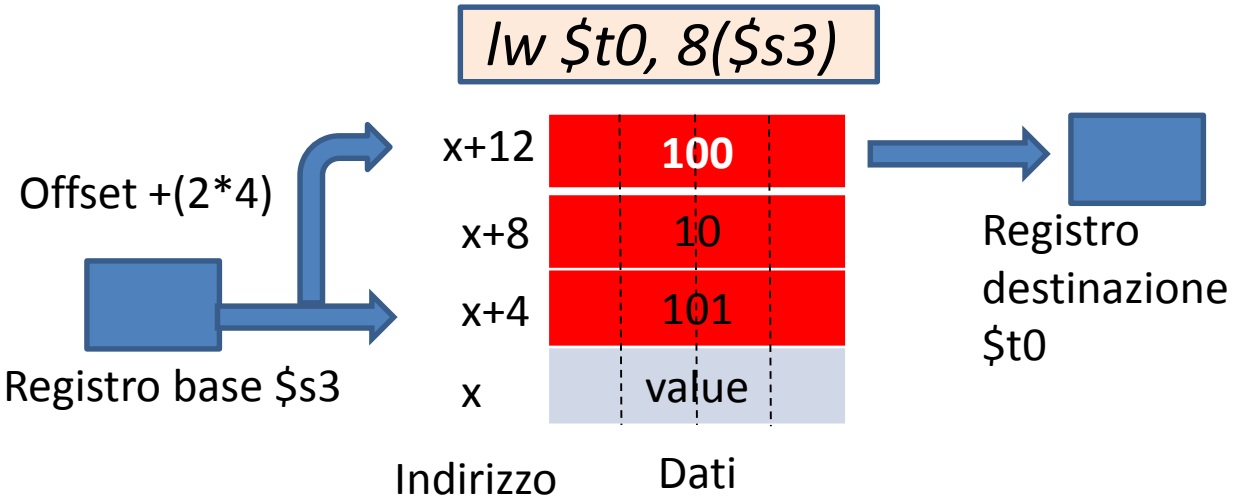
Esempio di lettura in memoria

Assumiamo un array immagazzinato in memoria dal compilatore all'indirizzo base x+1 (è un indirizzo di «parola»/«word» a 32 bit). Dove x è memorizzato nel registro $s3.



In realtà....

- *La maggior parte delle architetture **indirizza byte per byte***
- *Inoltre, le parole di dato sono da 4 byte in un processore a 32 bit.*
- *Dunque, la vera istruzione è:*



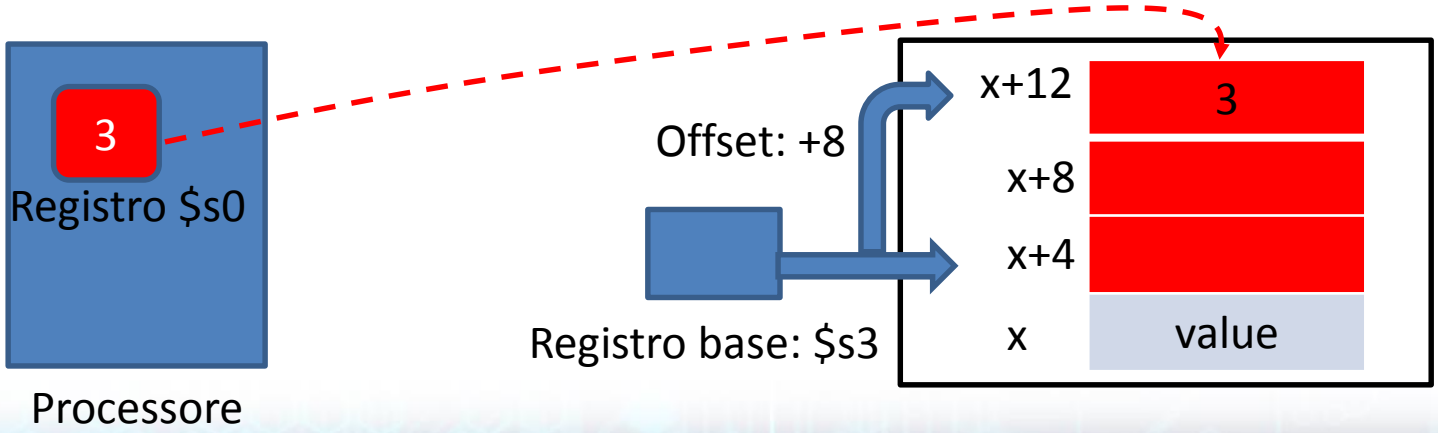
Questo meccanismo fa parte *dell'Addressing Mode* del MIPS, ovvero del modo con cui le istruzioni generano indirizzi di memoria

Analogamente…

E’ possibile copiare dati da un registro ($s0) ad una specifica locazione di memoria mediante l’istruzione di «STORE WORD»,
nome assembly MIPS: «sw»

Linguaggio C
r=3;
array[2]=r;

Il compilatore traduce l’assegnazione all’array in: **sw** *$s0, 8($s3)*



Quiz

- Sia A un array di 100 parole (numeri binari da 32 bit).
- Il compilatore ha effettuato le seguenti scelte:

Linguaggio C **A[12] = h + A[8]** **ASSEMBLER=??**

Occorre dapprima trasferire A[8] dalla memoria in un registro:

???

A[8] trasferito nel registro temporaneo $t0.
effettua h + A[8]
aggiorna A[12]

Soluzione

- Sia A un array di 100 parole (numeri binari da 32 bit).
- Il compilatore associa la variabile h al registro $s2.

Linguaggio C **A[12] = h + A[8]** **ASSEMBLER=??**

Occorre dapprima trasferire A[8] dalla memoria in un registro:

```
lw      $t0, 32($s3)
```

A[8] trasferito nel registro temporaneo $t0. *Offset=4byte* x *8parole*
effettua h + A[8]
aggiorna A[12]

Soluzione

- Sia A un array di 100 parole (numeri binari da 32 bit).
- Il compilatore associa la variabile h al registro $s2.

Linguaggio C **A[12] = h + A[8]** **ASSEMBLER=??**

Occorre dapprima trasferire A[8] dalla memoria in un registro:

```
lw    $t0, 32($s3)   # A[8] trasferito nel registro temporaneo $t0. Offset=4byte x 8parole
add   $t0, $s2, $t0  # effettua h + A[8]
                     # aggiorna A[12]
```

Soluzione

- Sia A un array di 100 parole (numeri binari da 32 bit).
- Il compilatore associa la variabile h al registro $s2.

Linguaggio C **A[12] = h + A[8]** **ASSEMBLER=??**

Occorre dapprima trasferire A[8] dalla memoria in un registro:

```
lw    $t0, 32($s3)   # A[8] trasferito nel registro temporaneo $t0. Offset=4byte x 8parole
add   $t0, $s2, $t0  # effettua h + A[8]
sw    $t0, 48($s3)   # aggiorna A[12]
```

Costanti

- Nei programmi reali, le istruzioni fanno uso massiccio di costanti
  - Nei benchmark SPEC2000, metà delle istruzioni MIPS ne fanno uso.

Le costanti andrebbero di volta in volta caricate dalla memoria mediante operazioni di LOAD => estrema lentezza!

Offrire versioni delle istruzioni aritmetiche in cui un operando è una costante (ADD IMMEDIATE, **addi**)

Esempio: **addi** *$s3,$s3,4* # somma 4 al registro $s3

MIPS supporta costanti negative, quindi non ha senso l'istruzione **subi**