

# Architettura degli Elaboratori

Vettori, matrici e pseudoistruzioni



SAPIENZA  
UNIVERSITÀ DI ROMA

Alessandro Checco

[alessandro.checco@uniroma1.it](mailto:alessandro.checco@uniroma1.it)

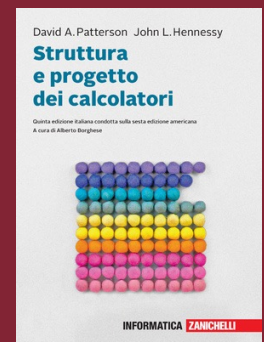
[alessandrochecco.github.io](https://alessandrochecco.github.io)

Special thanks and credits:

Andrea Sterbini, Iacopo Masi,

Claudio di Ciccio

[S&PdC] 2.1 – 2.8



# Modifica - Iterazioni (ciclo for)

## Esempio C

```
int N = 10;

for (i=0 ; i<N ; i++)
{
    // codice da ripetere
}

// codice seguente
```

## Esempio assembly

```
.data
N: .half 10

.text
# uso il registro $t0 per l'indice i
# uso il registro $t1 per il limite N
xor $t0, $t0, $t0    # azzero i
lhu $t1, N            # limite del ciclo
cicloFor:
    bge $t0, $t1, endFor    # test i >= N
    # codice da ripetere
    addi $t0, $t0, 1        # i += 1
    j cicloFor             # jump back
endFor:
    # codice seguente
```

# Vettori e matrici

---

## Vedremo:

- Vettori: manipolazione con indici e con puntatori
- Matrici a 2, 3 ed N dimensioni
- Esempi di programmi

## Vettore:

sequenza di **N elementi** di **dimensioni uguali**

consecutivi in memoria

indirizzabili per indice (da 0 a N-1)

dimensione totale =  $N \times \text{dimensione\_elemento}$

Si possono definire staticamente nella sezione **.data** del programma assembly  
usando un'**etichetta** per indicare l'**indirizzo del primo elemento** del vettore

Per indirizzare l'**elemento i-esimo** bisogna aggiungere l'offset  
 $i \times \text{dimensione\_elemento}$

# Vettore di word vs half word

---

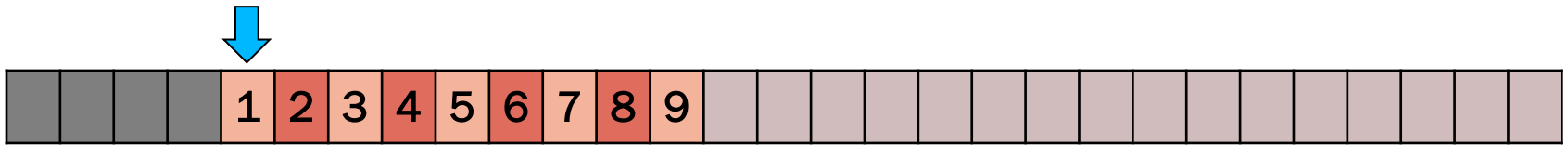
vettore di word a partire da 0x00001004

vettore di half word a partire da 0x0001000

# Vettori di byte in memoria

Vettore di **byte** (valori interi da 0 a 255)

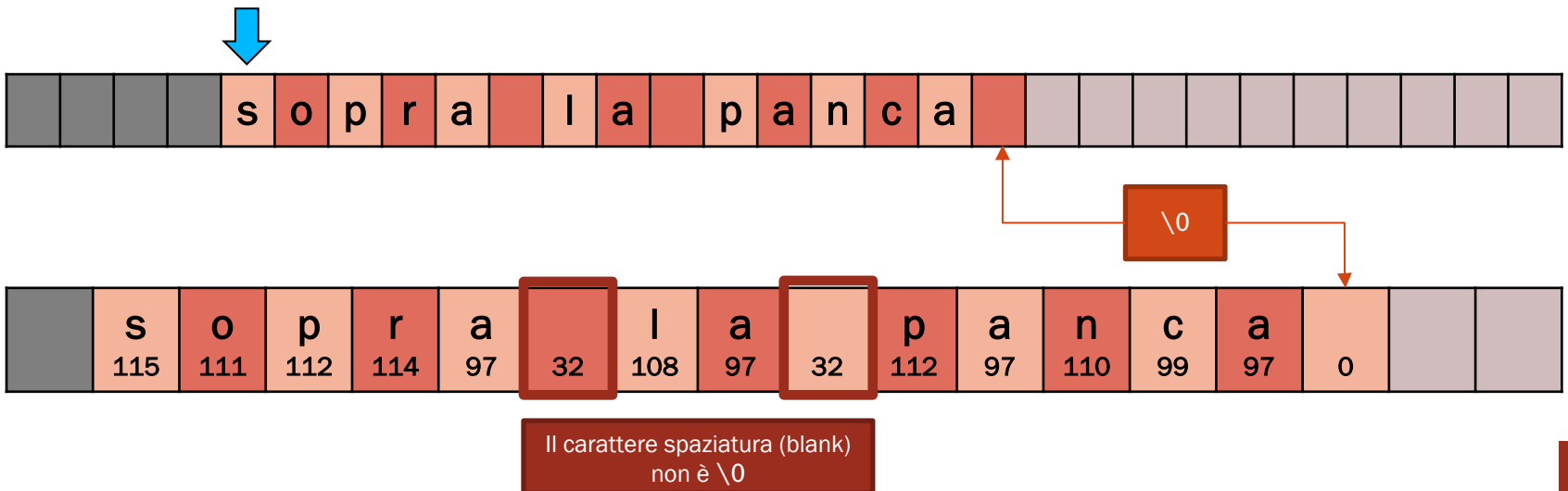
**label1: .byte**      1, 2, 3, 4, 5, 6, 7, 8, 9



**Testo:** vettore di caratteri (**byte**) seguiti da `\0` (carattere codificato con zero, 0x0)

**label2: .ascii**      "sopra la panca"

Viene memorizzata come sequenza dei codici ASCII dei caratteri inseriti nella direttiva **.ascii**

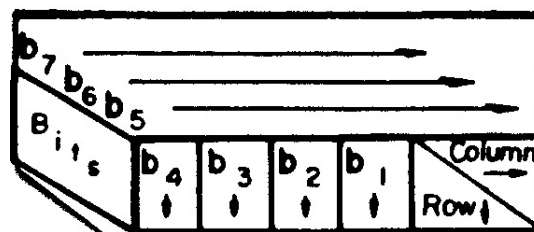


# American Standard Code for Information Interchange

<https://theasciicode.com.ar>

USASCII code chart

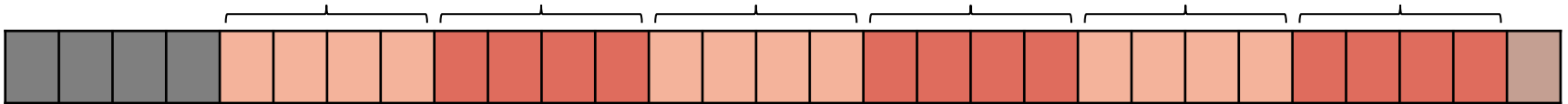
A  $\mapsto$  0x41 = 0100 0001  
a  $\mapsto$  0x61 = 0110 0001

					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
					0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

# I vettori di word

Vettore di **word**: numeri a 32 bit in Complemento a 2 (da  $-2^{31}$  a  $2^{31} - 1$ ) codificati in 4 byte

**Label3:** **.word** 1, 2, 3, 4, 5, 6 (6 elementi di 4 byte)

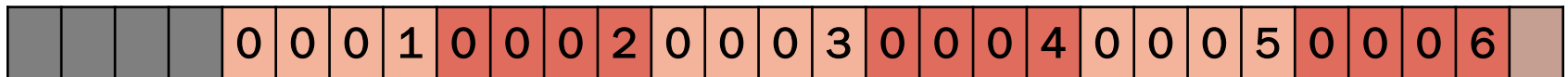


**Label4:** **.word** 0:100 (100 elementi di valore 0)

Il processore MIPS permette l'ordinamento dei byte di una word in due modi:

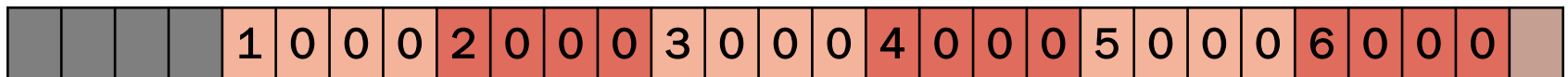
- **Big-endian** (o network-order, usato da Java e dalle CPU SPARC Sun/Oracle)

i byte della word sono memorizzati dal **most** significant *byte* al **least** significant *byte*



- **Little-endian** (usato dalle CPU Intel, ad es. Windows, e da **MARS**)

i byte della word sono memorizzati dal **least** significant *byte* al **most** significant *byte*



# Endianess: chiarimenti

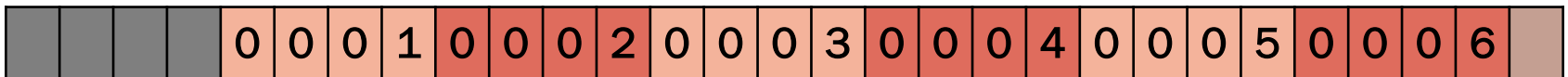
Il concetto di **Endianess** (se un dato sistema è little endian o big endian) ha a che fare con:

- indirizzamento in memoria (che avviene **per byte**) – ogni indirizzo specifica il **singolo byte**
- come i **byte** di una **word** sono ordinati in memoria

Il processore MIPS permette l'ordinamento dei byte di una word in due modi:

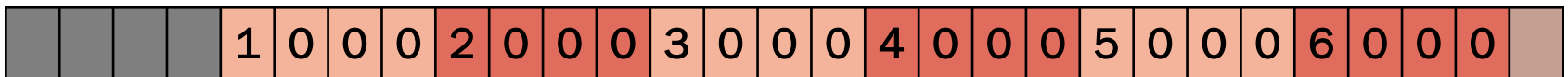
- **Big-endian** (o network-order, usato da Java e dalle CPU SPARC Sun/Oracle)

i byte della word sono memorizzati dal **most** significant *byte* al **least** significant *byte*



- **Little-endian** (usato dalle CPU Intel, ad es. Windows, e da **MARS**)

i byte della word sono memorizzati dal **least** significant *byte* al **most** significant *byte*





## Esempio – vettore di word con 0x01020304, 0x4321

---

.text

Big Endian

Little Endian

# E i byte?

---

Provate

vector: .byte 1,2,3,4 (quale word equivalente?)

# MARS e i vettori

MARS è little-endian

- nella finestra dei dati le **stringhe** sono visualizzate come **gruppi di 4 caratteri rovesciati**

Labels window:

Label	Address
label1	0x10010000
label2	0x10010002
label3	0x10010018

Data Segment window:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	r p o s	a l a	n a p	. \0 a c	. . . .	\t \b . .	\0 \0 \0 .	\0 \0 \0 .
0x10010020	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100e0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

label1: .asciiz "sopra la panca"

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values **ASCII**

## Accesso agli elementi per indice

Indirizzo dell'elemento  $i$  = indirizzo base del vettore +  $i \times \text{dimensione\_elemento}$

**Esempio:** (frammento che calcola l'indirizzo di un elemento in un vettore di word)

# \$t0 contiene l'indice dell'elemento (e.g., 2)

# \$t1 contiene l'indirizzo del vettore (e.g., 0x10010040)

# in \$t2 si ottiene l'indirizzo dell'elemento (da caricare da, o scrivere in, memoria)

```
sll $t2, $t0, 2      # word → 4 byte; shift di due bit → moltiplicazione per 4
```

```
add $t2, $t2, $t1    #
```

Per elementi di dimensioni diverse (half word o byte o altro) va cambiata la prima istruzione.

Se il vettore è allocato staticamente (i.e., viene dichiarato in sezione `.data`), la seconda istruzione può essere direttamente sostituita dall'istruzione di accesso in memoria

```
sll $t2, $t0, 2      # offset in byte dell'elem. rispetto all'inizio del vettore
```

```
lw  $s0, Array($t2)  # lettura della word
```

# Esempio

---

**Esempio:** (frammento che calcola l'indirizzo di un elemento in un vettore di word)

# \$t0 contiene l'indice dell'elemento (e.g., 2)

# \$t1 contiene l'indirizzo del vettore (e.g., 0x10010040)

# in \$t2 si ottiene l'indirizzo dell'elemento (da caricare da, o scrivere in, memoria)

sll \$t2, \$t0, 2      # word → 4 byte; shift di due bit → moltiplicazione per 4

add \$t2, \$t2, \$t1    # i\*\$t0+\$t1

# Cicli

---

Nei cicli si possono usare due stili di scansione di un vettore

## Scansione per indice

- Pro:
  - comoda se si deve usare l'indice dell'elemento per controlli o altro
  - l'incremento dell'indice non dipende dalla dimensione degli elementi
  - comoda se il vettore è allocato staticamente (nella sezione `.data`)
- Contro:
  - bisogna convertire ogni volta l'indice nel corrispondente offset in byte

## Scansione per puntatore (ovvero manipolando direttamente indirizzi in memoria)

- Pro:
  - si lavora direttamente su indirizzi in memoria
  - ci sono meno calcoli nel ciclo
- Contro:
  - non si ha a disposizione l'indice dell'elemento
  - l'incremento del puntatore dipende dalla dimensione degli elementi
  - bisogna calcolare l'indirizzo successivo all'ultimo elemento

## Esempio (con indice)

Esempio: somma degli elementi di un vettore di word a posizione divisibile per tre

*.data*

```
Vettore: .word 1, 2, 3, 4, 5, 6, 7, 8, 9 # vettore da sommare
N:       .word 9                        # numero di elementi
Somma:   .word 0                        # risultato
```

*.text*

```
main:  li      $t0, 0                    # i = 0
        lw      $t1, N                  # lettura di N
        li      $t2, 0                  # somma = 0
loop:  bge     $t0, $t1, fine            # è finito il ciclo?
        sll     $t3, $t0, 2              # offset: i*4
        lw      $t3, Vettore($t3)       # lettura di Vettore[i] (riuso t3)
        add     $t2, $t2, $t3            # somma += Vettore[i]
        addi    $t0, $t0, 3              # i += 3
        j       loop                    # riparte il ciclo
fine:  sw      $t2, Somma                # memorizzo il risultato
```

## Esempio (con puntatori)

.data

```
Vettore: .word 1, 2, 3, 4, 5, 6, 7, 8, 9 # vettore da sommare
N:       .word 9                        # numero di elementi
Somma:   .word 0                        # risultato
```

.text

```
main:    lw      $t1, N                # lettura di N
         la      $t0, Vettore          # indirizzo di Vettore
         sll     $t1, $t1, 2           # dimensione = N * 4
         add     $t1, $t1, $t0         # fine = ind.Vettore + dim
         li      $t2, 0                # somma = 0
loop:    bge     $t0, $t1, fine         # è finito il ciclo?
         lw      $t3, ($t0)            # lettura di Vettore[i]
         add     $t2, $t2, $t3         # somma += Vettore[i]
         addi    $t0, $t0, 12          # i += 3 * dim_elemento
         j       loop
fine:    sw      $t2, Somma            # memorizzo il risultato
```



# Matrici: vettori di vettori

Una matrice  $M \times N$  è altro una successione di  $M$  vettori, ciascuno di  $N$  elementi

- il numero di elementi totali è:  $M \times N$
- la dimensione totale in byte è:  $M \times N \times \text{dimensione\_elemento}$
- la si definisce staticamente come un vettore contenente  $M \times N$  elementi uguali

**Matrice:** **.word** **0:91** # spazio per una matrice  $7 \times 13$  word

L'elemento **e**,  
di coordinate  
**x=9, y=2** si trova  
ad una distanza di:

- 2 righe
- più 9 elementi  
dall'inizio, ovvero ad  
un offset di  
 $2 \times 13 + 9 = 35$  word  
cioè  
 $35 \times 4 = 140$  byte

13 colonne

7 righe	0	1	2	3	4	5	6	7	8	9	10	11	12	0
	13	14	15	16	17	18	19	20	21	22	23	24	25	1
	26	27	28	29	30	31	32	33	34	<b>e</b>				2
														3
														4
														5
														6
	0	1	2	3	4	5	6	7	8	9	10	11	12	

x

y

# Matrice 3d

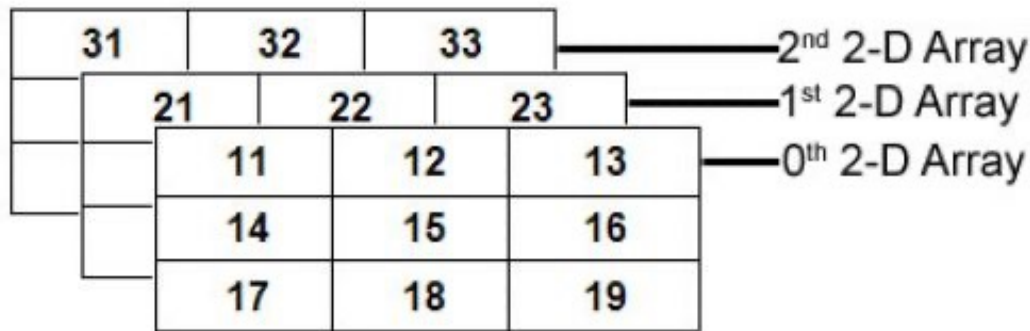
---

2 righe, 3 colonne, 2 strati

# Matrici 2D e 3D

Questo è esattamente il calcolo svolto dal compilatore C, per esempio

scrivere	<code>matrice[x][y]</code>
per una matrice di interi definita	<code>int matrice[NUM_RIGHE][NUM_COL]</code>
equivale a usare l'indirizzo	<code>matrice + (y*NUM_COL+x)*sizeof(int)</code>



## Matrici a 3 dimensioni

Una matrice 3D di **dimensioni**  $M \times N \times P$  è una successione di  $P$  matrici 2D grandi  $M \times N$

L'elemento a coordinate  $x, y, z$  è preceduto da:

$z$ «strati»	(matrici $M \times N$ formate da $M \cdot N$ elementi)
$y$ «righe»	di $M$ elementi sullo stesso strato
$x$ «elementi»	sulla stessa riga e strato

Quindi l'elemento si trova a  $z * (M * N) + y * N + x$  elementi dall'inizio della matrice 3D  
e la sua posizione in memoria è  $\text{indirizzo\_matrice} + (z * (M * N) + y * N + x) * \text{dim\_el.}$

---

Riprendiamo dall'esercizio



SAPIENZA  
UNIVERSITÀ DI ROMA

# Compito per casa (o per qualsiasi altro edificio 😊)



1. Installare **MARS** e prendere familiarità con interfaccia
2. Scrivere i programmi visti a lezione e provare ad eseguirli e debuggarli passo passo
  - Controllare come cambiano i registri sulla destra in base ai passi svolti.
  - Ispezionare come cambiano le pseudo-istruzioni immesse nelle istruzioni che poi svolge veramente il calcolatore
3. Partendo dal programma che trova il max in un vettore scrivere un programma in linguaggio assembly MIPS con MARS che dato un vettore ingresso **vector** e la sua dimensione **N** calcoli due somme dei numeri del vettore.



1. La prima somma deve sommare i valori del vettore di indice **dispari**. (Indice parte da 1)
2. La seconda somma deve sommare i valori di un vettore con **indice pari**. (Indice parte da 0)

Vector .word 4, -1, 5, 500, 0, 10000, -256  
N .word 5

Somme: .word 0, 0

Name	Num...	Value
\$ZERO	0	0
\$at	1	0
\$v0	2	0
\$v1	3	0
\$a0	4	0
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	0
\$t1	9	0
\$t2	10	0
\$t3	11	0
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	0
\$s1	17	0
\$s2	18	0
\$s3	19	0
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194384
hi		0
lo		0

Una volta scritto con questi dati cambiate N e controllate se continua a funzionare

# Una possibile soluzione

```
1 .data
2 #####
3 # Declaration of constants and variables in memory
4 #####
5 # Array: .word 4, -1, 5, 500, 0, 10000, -256
6 # N: .word 7
7 Array: .word 1, 2, 2, 4
8 N: .word 4
9 Sums: .word 0, 0
10
11 .text
12 #####
13 # Initialisation and loading
14 #####
15 and $t1, $t1, $zero # $t1: cursor index for the Array
16 add $t2, $zero, $0 # $t2: temporary value loaded from Array[$t1]
17 addi $t3, $zero, 0 # $t3: a flag for 3ven elements
18 li $t5, 0 # $t5: offset
19 andi $s3, $s3, 0 # $s3: sum of 3ven elements in Array
20 xor $s0, $s0, $s0 # $s0: sum of 0dd elements in Array
21
22 lw $s7, N # $s7: the length of Array ($s7 = N)
23
24 #####
25 # Core business
26 #####
27 while:
28     bge $t1, $s7, whileEnd # Exit the cycle if the cursor goes beyond the length of Array
29 # {
30     nor $t3, $t3, $0 # Switch the flag. When $t1 == 0, $t3 = 11...1. When $t0 == 1, $t3 = 00...0
31     sll $t5, $t1, 2 # offset = index * 4
32     lw $t2, Array($t5) # Base + offset (indexed absolute)
33     if:
34         bnez $t3, else # Is $t3 != 0 ?
35         then: # If not, $t2 contains a value at an odd index
36             add $s0, $s0, $t2 # Add the content of Array[$t1] to $s0 (0dd)
37             j endif # Exit the if-then-else block
38         else: # If $t2 contains a value at an even index
39             add $s3, $s3, $t2 # Add the content of Array[$t1] to $s3 (3ven)
40         endif:
41         addi $t1, $t1, 1 # Increase the index
42     j while
43 # }
44 whileEnd:
45 # Store the results in Sums
46 sw $s3, Sums # Store the sum of 3even elements in Sums[0]
47 sw $s0, Sums+4 # Store the sum of 0dd elements in Sums[1] (again, remember we are considering 4-byte words)
```