

Architettura degli Elaboratori e Laboratorio

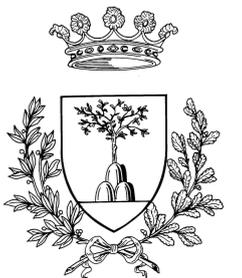
Matteo Manzali

Università degli Studi di Ferrara

Anno Accademico 2016 - 2017

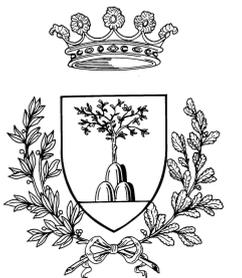
Rappresentazione dei caratteri

- Anche i caratteri, come i numeri, sono rappresentati nei calcolatori sotto forma di codice binario.
- Le regole che associano un carattere ad una sequenza di bit sono chiamati **codici**:
 - due calcolatori devono usare lo stesso codice per poter comunicare
 - ci sono diversi codici che dipendono dalla lingua, dal sistema operativo, ...
 - tendenzialmente tutti seguono alcune regole di base (cifre e lettere consecutive hanno rappresentazioni binarie consecutive)



Codice ASCII

- American Standard Code for Information Interchange (ASCII).
- Prima codifica a larga diffusione (anni 60), pensata per le telescriventi.
- Ogni carattere viene rappresentato da un byte:
 - vengono utilizzati solo 7 bit (il MSB veniva usato come bit di parità)
 - i primi 32 caratteri sono di controllo e non vengono stampati
 - i restanti 95 sono caratteri stampabili



Caratteri di controllo

Hex	Name	Meaning	Hex	Name	Meaning
0	NUL	Null	10	DLE	Data Link Escape
1	SOH	Start Of Heading	11	DC1	Device Control 1
2	STX	Start Of TeXt	12	DC2	Device Control 2
3	ETX	End Of TeXt	13	DC3	Device Control 3
4	EOT	End Of Transmission	14	DC4	Device Control 4
5	ENQ	Enquiry	15	NAK	Negative Acknowledgement
6	ACK	ACKnowledgement	16	SYN	SYNchronous idle
7	BEL	BELI	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	HT	Horizontal Tab	19	EM	End of Medium
A	LF	Line Feed	1A	SUB	SUBstitute
B	VT	Vertical Tab	1B	ESC	ESCape
C	FF	Form Feed	1C	FS	File Separator
D	CR	Carriage Return	1D	GS	Group Separator
E	SO	Shift Out	1E	RS	Record Separator
F	SI	Shift In	1F	US	Unit Separator



Caratteri stampabili

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL



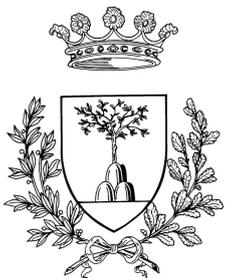
Codice ASCII esteso

- Oramai il codice ASCII non è più utilizzato come protocollo di trasmissione per le telescriventi.
- L'ottavo bit viene quindi utilizzato per codificare caratteri non standard:
 - si conservano le prime 128 codifiche
 - i successivi 128 caratteri dipendono dall'estensione utilizzata (ve ne sono diverse)
 - lo Standard 8859 definisce le estensioni ASCII (pagine di codice) e le associa ad una o più lingue
 - grazie alle diverse pagine di codice è possibile rappresentare caratteri accentati e simboli non presenti nello standard ASCII



Assembly

Matteo Manzali - Università degli Studi di Ferrara

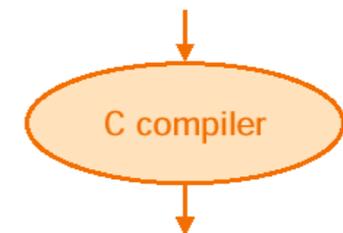


Livelli di astrazione

- Diversi livelli di astrazione:
 - linguaggio ad alto livello
 - linguaggio assembly
 - linguaggio macchina
- Il livello più astratto omette dettagli:
 - tendiamo a perdere il controllo sulle operazioni da eseguire
 - ci permette di descrivere algoritmi complessi in maniera intuitiva

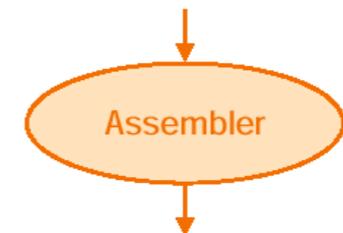
High-level
language
program
(in C)

```
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
0000001111100000000000000001000
```

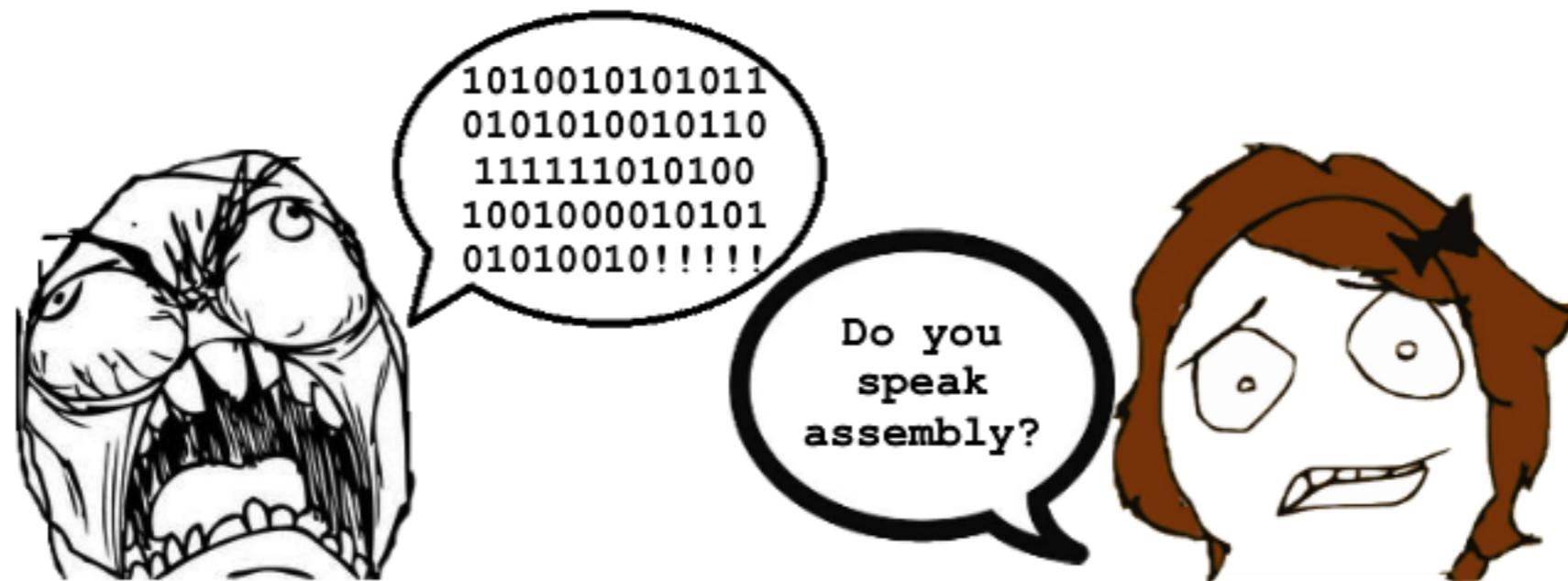
Assembly

- Il linguaggio assembly descrive in maniera univoca le istruzioni che l'elaboratore deve eseguire:
 - il programma assembler traduce l'assembly in codice binario
- Controllo del flusso poco sofisticato (non ci sono for, while, if).
- Istruzioni aritmetiche con un numero fisso di operandi.
- E' necessario avere coscienza delle risorse di memoria a disposizione



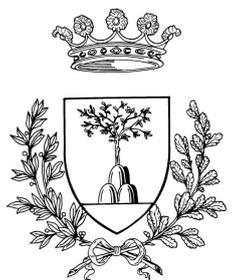
Assembly - motivazione

- Programmare in assembly aiuta a capire funzionamento e meccanismi base di un calcolatore.
- Fa meglio comprendere cosa accade durante l'esecuzione di un programma scritto ad alto livello.
- Permette di parlare il linguaggio macchina in modo "umano".



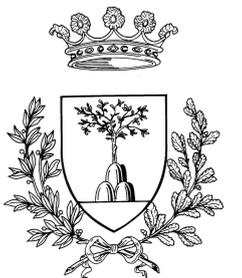
Assembly - vantaggi

- Si definiscono esattamente le istruzioni da eseguire e le locazioni di memoria da modificare:
 - controllo sul codice
 - controllo sull'hardware
- Permette di ottimizzare il codice ed avere programmi più efficienti.



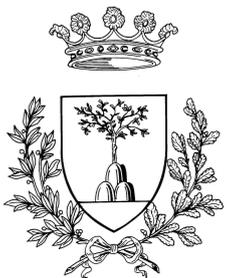
Assembly - svantaggi

- Scarsa portabilità (ogni famiglia di processori ha il suo linguaggio).
- I programmi tendono a essere molto lunghi a causa della “semplicità” delle istruzioni.
- Molto facile cadere in errore a causa della scarsa leggibilità del codice.
- I moderni compilatori ottimizzano producono codice assembly estremamente efficiente: è quasi impossibile fare di meglio programmando direttamente in assembly.



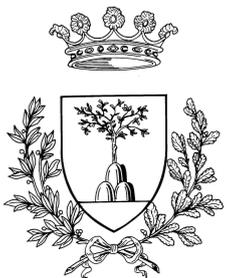
MIPS

- Il mercato offre tantissime famiglie di processori, ciascuno col suo linguaggio.
- Tutti i linguaggi sono simili a grandi linee, nonostante ciascuno abbia particolari caratteristiche che lo differenziano dagli altri.
- In questo corso studieremo l'assembly del processore **MIPS**:
 - uno dei primi processori RISC (Reduced Instruction Set Computer)
 - nato nel 1985
 - nonostante studieremo la prima versione a 32 bit, ne sono state prodotte anche a 64 bit e con l'utilizzo di istruzioni vettoriali



MIPS

- **Vantaggi:**
 - semplice e facile da imparare
 - molto usato in didattica
- **Svantaggi:**
 - istruzioni semplici → codici molto lunghi
 - fare programmi complessi diventa snervante
- Il MIPS è più diffuso di quanto si pensi:
 - stampanti laser, routers e molti altri sistemi embedded
 - console di gioco (Play Station 2, Nintendo 64)



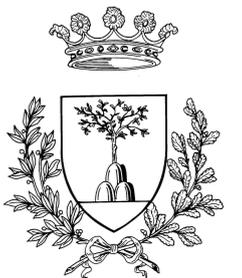
Simulatore

- Fino all'anno scorso veniva usato QtSpim.
- Da quest'anno abbiamo deciso di adottare **MARS**:
 - contiene un IDE per programmare
 - è molto più evoluto di QtSpim
 - offre la possibilità di eseguire istruzioni non permesse da QtSpim (statistiche sulle istruzioni, generazione numeri random, etc...)
 - è scritto in java, basta scaricare il file .jar ed eseguirlo



I registri

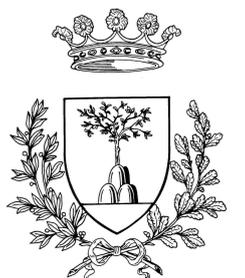
- Tipicamente le istruzioni operano su dati disponibili in **registri** all'interno del processore.
- Vi sono 32 registri da 4 byte ciascuno.
- I nomi dei registri iniziano con il simbolo \$ seguito dal loro numero (\$0, \$1, \$2, ..., \$31).
- Ciascun registro ha un suo compito specifico:
 - valore di ritorno di una funzione
 - argomento di funzione
 - etc...
- Per facilitare la programmazione e la comprensione del codice sono stati introdotti dei nomi che descrivono lo scopo del registro.



I registri

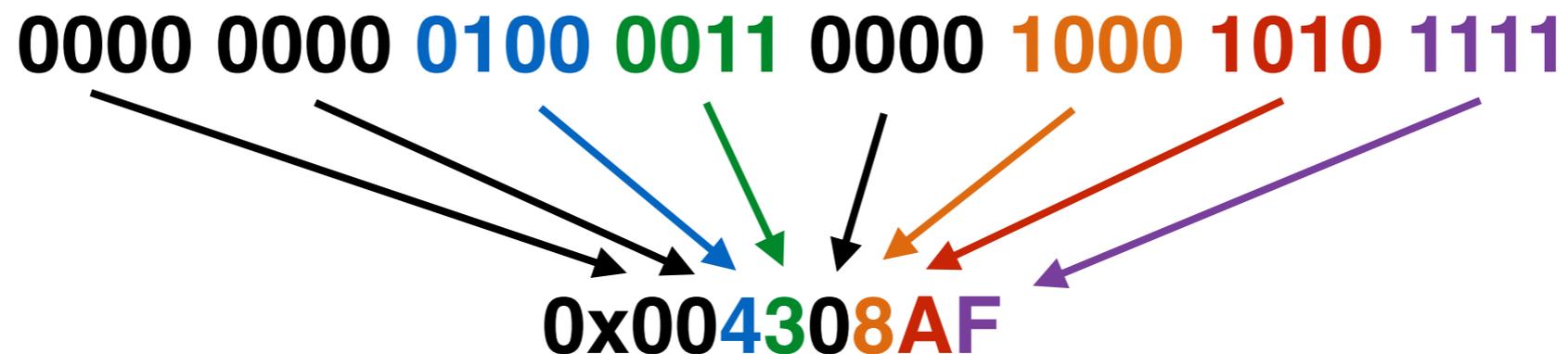
- Ciascun registro ha un suo compito specifico, è **necessario seguire delle convenzioni!**

Name	Register Number	Usage	Should preserve on call?
\$zero	0	the constant 0	no
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	yes
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	yes
\$t8 - \$t9	24-25	temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes



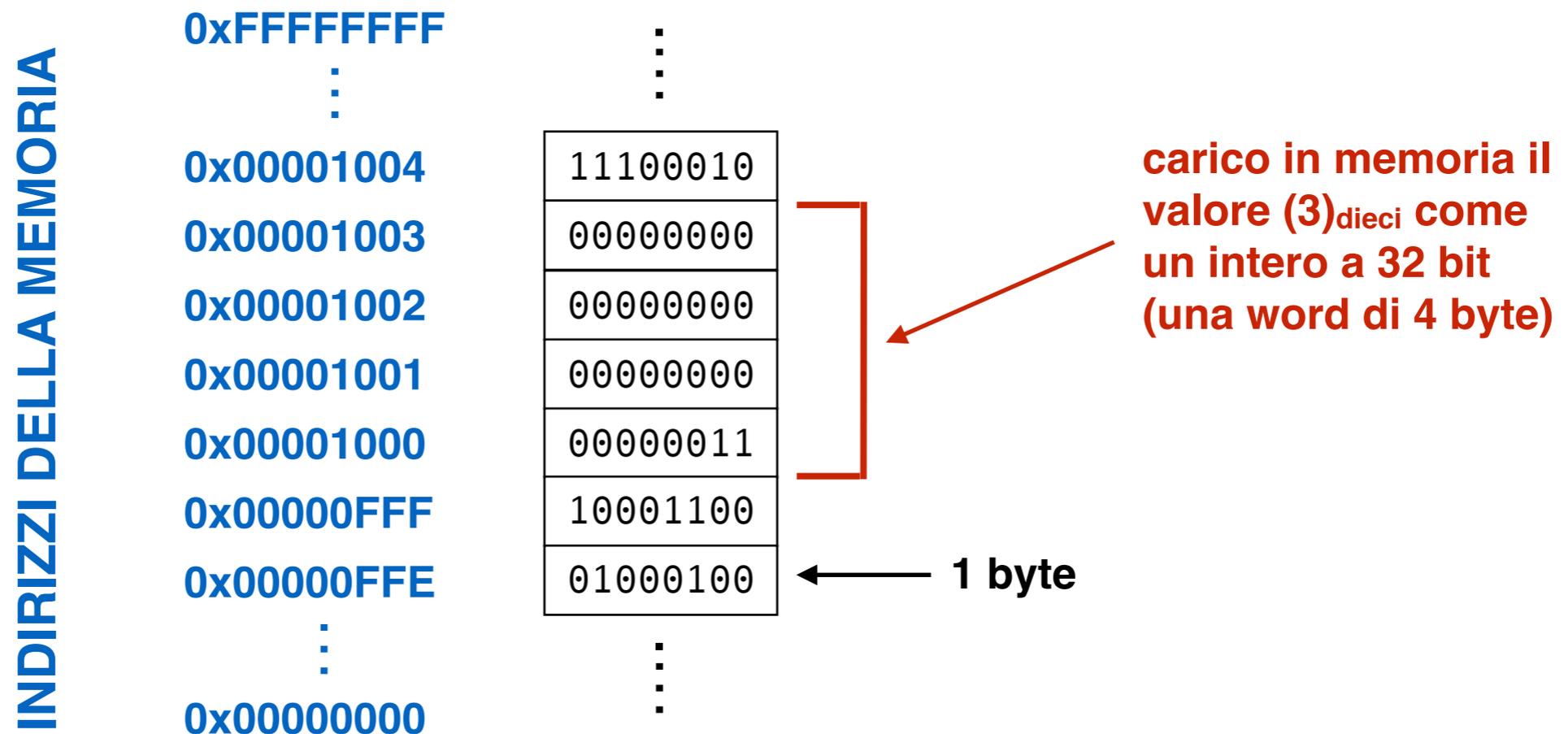
Memoria principale

- Nel caso in cui i registri non bastino (o per altri casi specifici) c'è a disposizione una **memoria principale**.
- Composta da 2^{32} locazioni, ciascuna di dimensione 32 bit (1 byte).
- Ogni locazione è individuata da un indirizzo di 32 bit.
- Gli indirizzi sono tipicamente espressi in esadecimale per comodità (8 cifre invece delle 32 necessarie in binario):



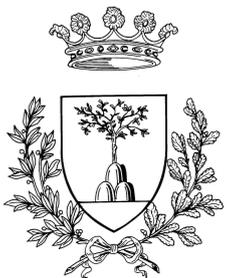
Memoria principale

- Quando si lavora con gli interi o con i floating points (32 bit) ci si sposta sempre di 4 in 4 tra gli indirizzi della memoria.
- 2^{32} locazioni da 1 byte ciascuna $\rightarrow 2^{30}$ parole indirizzabili.



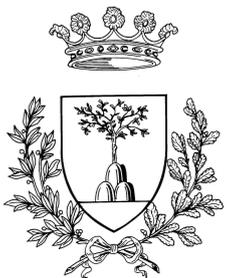
Istruzioni in MIPS

- Nelle prossime slides verranno introdotte tutte le principali istruzioni previste dall'architettura MIPS.
- Verranno inoltre introdotte le tecniche per implementare i costrutti dei linguaggi ad alto livello come for, while, etc.
- Non prendete queste slides come un manuale:
 - il mio consiglio è quello di tenere sempre una pagina del browser aperta sulla descrizione dell' instruction set di MIPS
- <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html> (en)
- https://it.wikiversity.org/wiki/ISA_e_Linguaggio_Assembly_MIPS (it)
- etc...



Pseudo-istruzioni

- Per facilitare (un poco) la vita ai programmatori, sono state introdotte delle **pseudo-istruzioni**:
 - sono istruzioni MIPS che **non** hanno un corrispettivo diretto nel linguaggio macchina
 - l'assembler solitamente espande una pseudo istruzione in due o più istruzioni presenti nel linguaggio macchina
- Nelle prossime slides vedremo anche alcune pseudo istruzioni (mul, move, bgt, etc...).
- E' importante che capiate la differenza tra una pseudo-istruzione ed una istruzione:
 - fondamentale per capire performances e per debugging



Istruzioni aritmetico-logiche

- Una istruzione aritmetica ha la forma generale:

$$op\ rd, rs, rt \rightarrow rd = rs\ op\ rt$$

- ***op*** corrisponde all'operatore aritmetico o logico:
 - add, addu, sub, subu, mult, multu, div, divu, ...
 - or, and, nor, ...
- ***rd*** è il registro di destinazione (dove va il risultato).
- ***rs*** e ***rt*** sono i registri sorgente (i termini dell'operazione) e possono anche coincidere con ***rd***.



Istruzioni aritmetico-logiche

- Alcune istruzioni hanno anche la versione con *immediate operands*:

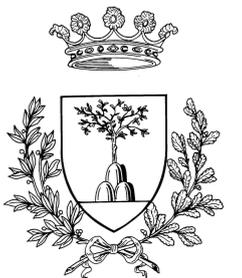
op rd, rs, costante → $rd = rs \text{ op } \textit{costante}$

- ***op*** corrisponde all'operatore aritmetico o logico:
 - addi, addiu
 - ori, andi, nor, ...
- ***rd*** è il registro di destinazione (dove va il risultato).
- ***rs*** e ***costante*** sono i termini dell'operazione.
- L'operando ***costante*** è limitato a 16 bit.



La costante zero

- Nella programmazione è frequente l'uso della costante “0”.
- MIPS offre un registro in sola lettura (\$zero o \$0) 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, b, c  
c = a + b;
```

```
# a → $s1, b → $s2, c → $s3  
add $s3, $s1, $s2
```

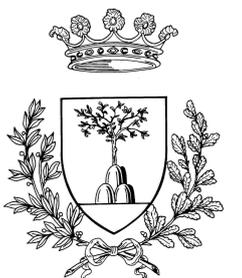
```
// int a, b, c, d  
d = a + b + c;
```

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

- Unsigned add:

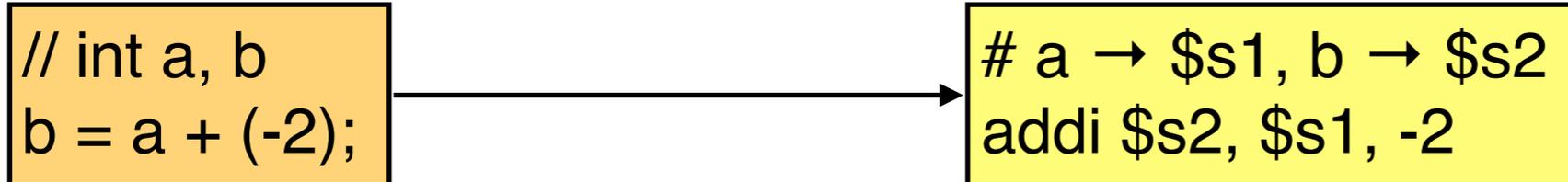
```
// unsigned int a, b, c  
c = a + b;
```

```
# a → $s1, b → $s2, c → $s3  
addu $s3, $s1, $s2
```

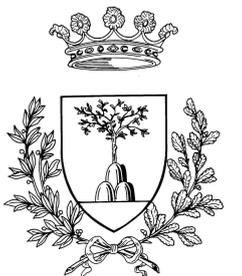
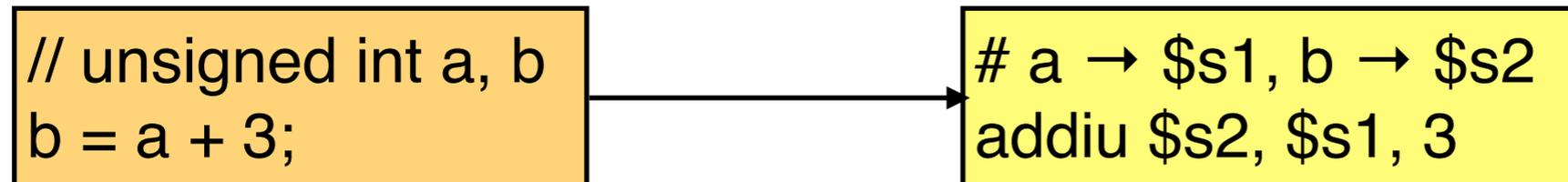


Addizione

- Signed immediate add:



- Unsigned immediate add:



Sottrazione

- Signed sub:

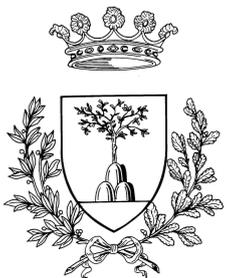
```
// int a, b, c  
c = a - b;
```

```
# a → $s1, b → $s2, c → $s3  
sub $s3, $s1, $s2
```

- Unsigned sub:

```
// unsigned int a, b, c  
c = a - b;
```

```
# a → $s1, b → $s2, c → $s3  
subu $s3, $s1, $s2
```



Sottrazione

- La sottrazione immediate non esiste, si implementa tramite addizione:

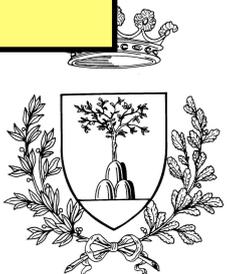
```
// int a, b  
b = a - 2;
```

```
# a → $s1, b → $s2  
addi $s2, $s1, -2 # b = a + (-2)
```

- Vediamo qualcosa di più complesso che coinvolge somme e sottrazioni:

```
// int a, b, c, d, e  
e = (a + b) - (c - d);
```

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



Moltiplicazione

- mul (pseudo-istruzione):

```
// int a, b, c  
c = a * b;
```

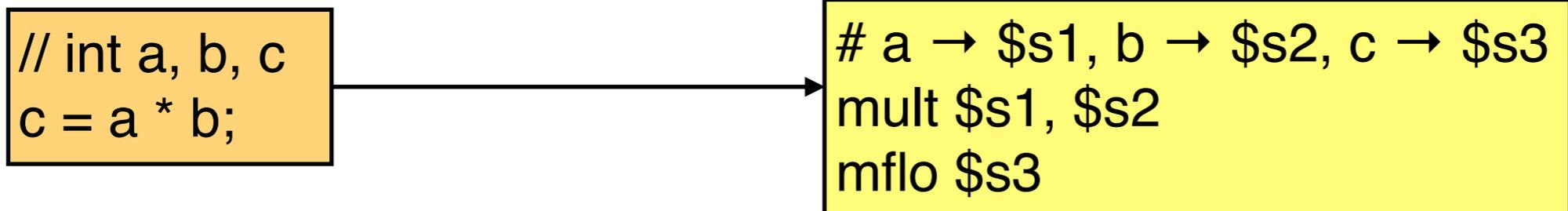
```
# a → $s1, b → $s2, c → $s3  
mul $s3, $s1, $s2
```

- Salva i 32 bit più “bassi” del risultato in \$s3.
- La moltiplicazione può facilmente generare un overflow.
- L'istruzione MIPS più generica per la moltiplicazione salva il risultato in due registri speciali, ciascuno a 32 bit, chiamati hi e lo.
- Esistono poi due istruzioni speciali per recuperare quei valori.

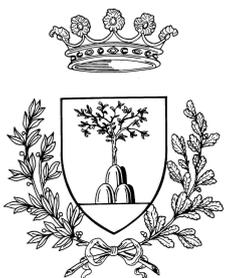


Moltiplicazione

- 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:



- Notare la sintassi di `mult` (contiene in maniera esplicita solo i due operandi).
- Si può poi utilizzare `mfhi` per recuperare i 32 bit più alti.



Divisione

- La divisione intera è un'operazione che ha due risultati:
 - quoziente
 - resto
 - Es.: $131 / 16 = 8$ 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;
```

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



Operazione OR

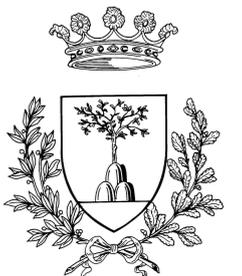
- or e ori (or with immediate).
- Utile per settare alcuni bit di una parola a 1 lasciando inalterati gli altri:

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;
```

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

Matteo Manzali - Università degli Studi di Ferrara



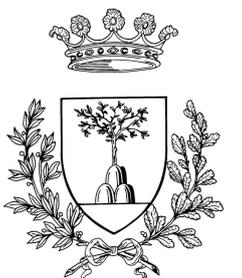
Operazione AND

- and e andi (and with immediate).
- 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;
```

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



Operazione NOT

- Utile per invertire dei bit in una parola:
 - $0 \rightarrow 1, 1 \rightarrow 0$
- Non esiste direttamente in MIPS, bisogna usare NOR:
 - $a \text{ NOR } b = \text{NOT } (a \text{ OR } b)$
 - $\text{NOT } a = \text{NOT } (a \text{ OR } 0) = a \text{ NOR } 0$

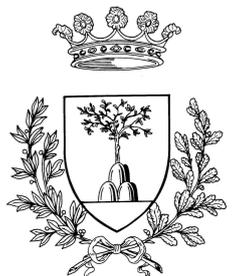
0000	0000	0000	0000	0000	1101	1100	0000
1111	1111	1111	1111	1111	0010	0011	1111

00	00	0D	C0
FF	FF	F2	3F

```
// unsigned int a = 0xDC0, b  
b = ~a;
```



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



Operazione di Shift

- Shift Left utile per moltiplicare per 2^n (solo unsigned):

$0001\ 1010 \ll 3 = 1101\ 0000$

$$26 \cdot 2^3 = 208$$

- Shift Right utile per dividere per 2^n (solo unsigned):

$0011\ 1010 \gg 3 = 0000\ 0111$

$$58 / 2^3 = 7$$

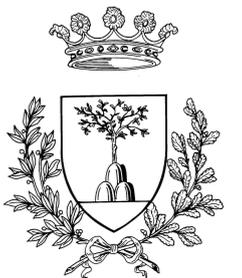
```
// unsigned int a, b, c, d  
b = a << 3;  
d = c >> 3;
```

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



MIPS e memoria

- In MIPS le istruzioni sono separate dai dati in memoria.
- **“Instruction memory”**:
 - parte della memoria che contiene le istruzioni (in linguaggio macchina)
 - read only
- **“Data memory”**:
 - parte della memoria che contiene i dati manipolati dal programma
 - read / write
- **Anche le istruzioni, come i dati in memoria, sono identificate da un indirizzo!**



Etichette

- Le etichette (labels) vengono usate in MIPS per associare un nome ad un indirizzo.
- Per distinguerle dal codice vengono solitamente scritte tutte in maiuscolo (o solo con la prima lettera maiuscola).
- Vengono utilizzate per indicare l'indirizzo di istruzioni o di dati in memoria.

```
        add $t2, $t1, $zero
LABEL:  sub $t1, $t0, $t2
        or  $s3, $s1, $s2

# LABEL contiene l'indirizzo dell'istruzione sub
```

- L'etichetta è poi convertita dal assembler nell'indirizzo dell'istruzione (o del dato) corrispondente.



Branches

- Il comando di branch permette di saltare ad un'istruzione:
 - se una certa condizione è verificata (branch condizionato)
 - incondizionatamente (branch incondizionato)
- Branch condizionato:
 - **beq rt, rs, L** → “branch equal”: se $rt == rs$ salta a L, altrimenti prosegui sequenzialmente
 - **bne rt, rs, L** → “branch not equal”: se $rt != rs$ salta a L, altrimenti prosegui sequenzialmente
- Branch incondizionato:
 - **j L** → “jump”: prosegui l'esecuzione all'istruzione con etichetta L



if - then

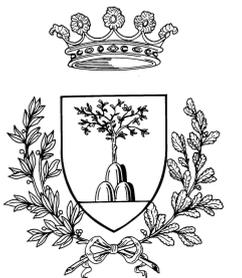
```
if (i == j) {  
    f = i + j;  
}  
// L:
```

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

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

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

- **Notare il test invertito in MIPS rispetto al C!**



if - then - else

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

```
# i → $s1, j → $s2, f → $s3  
bne $s1, $s2, ELSE    # se i != j salta a L  
add $s3, $s1, $s2     # f = i + j  
j END                 # salta a END  
ELSE:  
sub $s3, $s1, $s2     # f = i - j  
END:
```

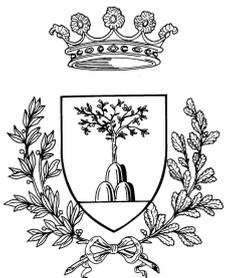
```
if (i != j) {  
    f = i + j;  
} else {  
    // ELSE:  
    f = i - j;  
}  
// END:
```

```
# i → $s1, j → $s2, f → $s3  
beq $s1, $s2, ELSE    # se i == j salta a L  
add $s3, $s1, $s2     # f = i + j  
j END                 # salta a END  
ELSE:  
sub $s3, $s1, $s2     # f = i - j  
END:
```



Altre condizioni

- In MIPS le condizioni dei branch implementate in hardware sono solamente beq e bne (poichè sono le più comuni).
- In C è però possibile utilizzare altre condizioni oltre a “==” e “!=“:
 - “<” , “>” , “<=” , “>=”
- I branch con altre condizioni si implementano 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 rd, rs, costante → se $rs < \text{costante}$, rd = 1, altrimenti rd = 0
 - **set less than unsigned:** sltu rd, rs, rt
 - **set less than unsigned immediate:** sltiu rd, rs, costante



Altre condizioni

- Branch less than:

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

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

- Branch less equal:

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

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

N.B.: $i \leq j \rightarrow !(j < i)$



Altre condizioni

- Esistono delle pseudo-istruzioni che possono essere utilizzate per evitare di costruire i branch condizionati con le condizioni di maggiore o minore:
 - **blt** (branch less then) \rightarrow if ($a < b$) ...
 - **ble** (branch less equal) \rightarrow if ($a \leq b$) ...
 - **bgt** (branch greater then) \rightarrow if ($a > b$) ...
 - **bge** (branch greater equal) \rightarrow if ($a \geq b$) ...



Signed vs unsigned

- Bisogna prestare molta attenzione nel confrontare numeri interpretandoli come signed o unsigned.
- Esempio:

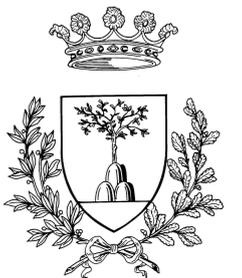
```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111  
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
```

Interpretazione con segno:

```
slt $t0, $s0, $s1 → -1 < +1 → $t0 = 1
```

Interpretazione senza segno:

```
sltu $t0, $s0, $s1 → +4294967295 > +1 → $t0 = 0
```



Cicli

- I cicli non sono previsti in MIPS.
- Si utilizzano i branch e le istruzioni di controllo per imitarne il funzionamento.

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

```
int s = 0;
int i = 0;
int t = 10;
L: if (i == t) goto E;
   s += i;
   ++i;
   goto L;
E:
```

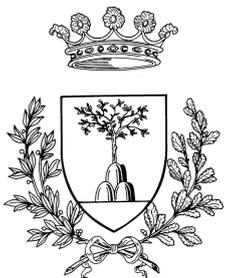
- N.B. L'istruzione goto è un'istruzione C di salto incondizionato.



Cicli

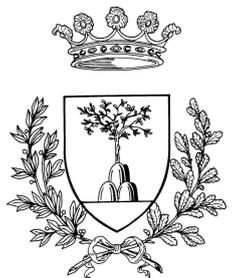
```
int s = 0;
int i = 0;
int t = 10;
L: if (i == t) goto E;
   s += i;
   ++i;
   goto L;
E:
```

```
add $s0, $zero, $zero # s = 0
add $t0, $zero, $zero # i = 0
addi $t1, $zero, 10   # t = 10
L: beq $t0, $t1, E     # if i == t vai a E
   add $s0, $s0, $t0  # s += i
   addi $t0, $t0, 1   # ++i
   j L                # vai a L
E:
```



Accesso a memoria

- Finora abbiamo operato con registri che contenevano già il valore delle variabili (dati) del nostro programma.
- Come già anticipato è possibile scrivere e leggere dati nella memoria principale dedicata ai dati (data memory).
- Operazioni per accedere alla memoria:
 - **lw rt, offset(rs)** → “load word”: legge la parola all’indirizzo specificato e la copia nel registro rt
 - **sw rt, offset(rs)** → “store word”: legge la parola contenuta nel registro rt e la scrive all’indirizzo specificato
- L’indirizzo utilizzato da lw e sw è calcolato come:
 - valore contenuto in rs + offset



Accesso a memoria

- Esempio: cerco il massimo tra due numeri.
- Supponiamo:

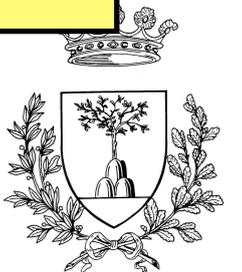
p_i → \$s0 → puntatore al primo numero

p_j → \$s1 → puntatore al secondo numero

p_m → \$s2 → puntatore al risultato

```
int i = *p_i;
int j = *p_j;
if (i < j) {
    *p_m = j;
}
else {
    *p_m = i;
}
```

```
lw $t0, 0($s0)      # carica i
lw $t1, 0($s1)      # carica j
slt $t2, $t0, $t1   # test i < j
beq $t2, $zero, I   # se no, salta a I
J: sw $t1, 0($s2)    # salva t1 (j) in p_m
j E                 # salta a E
I: sw $t0, 0($s2)    # salva t2 (i) in p_m
E:
```



Accesso a memoria

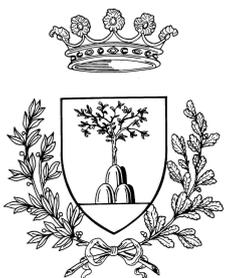
- Supponiamo di avere in memoria un array (chiamato arr) di 5 interi.
- Notare che gli indirizzi vanno di 4 in 4, perchè sono visualizzati solo gli indirizzi delle word (gruppi di 4 byte).
- Indirizzi degli elementi di arr:

&arr[0] → 0x00001000
&arr[1] → 0x00001004
&arr[2] → 0x00001008
&arr[3] → 0x0000100C
&arr[4] → 0x00001010

INDIRIZZI DELLA MEMORIA

0xFFFFFFFF	⋮
⋮	⋮
0x00001010	725
0x0000100C	-980
0x00001008	-2
0x00001004	324
0x00001000	3210
0x00000FFC	0
0x00000FF8	0
⋮	⋮
0x00000000	⋮

indirizzo i-esimo elemento = indirizzo elemento 0 + (i • 4)



Accesso a memoria

```
// arr → array inizializzato  
  
int i = 0;  
int e = 5;  
int t;  
for (; i < e; ++i) {  
    t = arr[i];  
    // ...  
}
```

```
# arr → $s0 , $t2 usata come temporanea  
  
    add $t0, $zero, $zero    # i = 0  
    addi $t1, $zero, 5      # e = 5  
  
L:  slt $t2, $t0, $t1        # if i < e , 1 else 0  
    beq $t2, $zero, E        # if i >= e vai a E  
  
    sll $t2, $t0, 2          # t2 = i * 4  
    add $t2, $t2, $s0        # t2 += s0  
    lw $t3, 0($t2)          # t = arr[i]  
    # ...  
  
    addi $t0, $t0, 1         # ++i  
    j L                      # vai a L  
  
E:
```

- \$s0 non viene modificato.
- Per moltiplicare faccio shift a sinistra di 2.



Indici VS puntatori

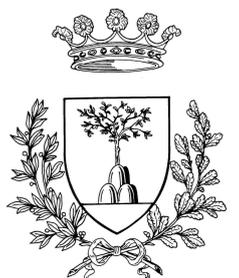
- Esempio: inizializzare un array attraverso l'uso di indici.

```
// arr → array
// N → size of array

int i = 0;
for (; i != N; ++i) {
    arr[i] = 0;
}
```

```
# arr → $s0 , N → $s1

    add $t0, $zero, $zero    # i = 0
L:   beq $t0, $s1, E        # if i == N vai a E
    sll $t1, $t0, 2         # t1 = i * 4
    add $t1, $t1, $s0       # t1 += s0
    sw $zero, 0($t1)        # arr[i] = 0
    addi $t0, $t0, 1        # ++i
    j L                      # vai a L
E:
```



Indici VS puntatori

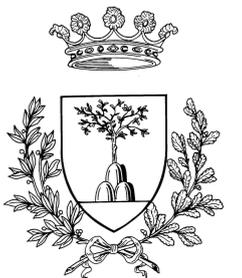
- Esempio: inizializzare un array attraverso l'uso di puntatori.

```
// arr → array
// N → size of array

int* p = arr;
int* p_end = arr + N;
while (p != p_end) {
    *p = 0;
    ++p;
}
```

```
# arr → $s0 , N → $s1
```

```
    sll $s1, $s1, 2           # N *= 4
    add $t0, $s0, $zero      # $t0 → p
    add $t1, $s0, $s1       # $t1 → p_end
L:  beq $t0, $t1, E         # if p == p_end vai a E
    sw $zero, 0($t0)        # *p = 0
    addi $t0, $t0, 4        # p += 4
    j L                     # vai a L
E:
```



Indici VS puntatori

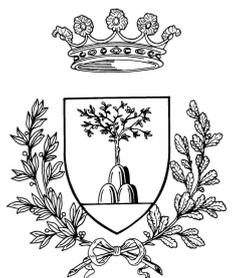
con indici

```
    add $t0, $zero, $zero
L:   beq $t0, $s1, E
    sll $t1, $t0, 2
    add $t1, $t1, $s0
    sw $zero, 0($t1)
    addi $t0, $t0, 1
    j L
E:
```

con puntatori

```
    sll $s1, $s1, 2
    add $t0, $s0, $zero
    add $t1, $s0, $s1
L:   beq $t0, $t1, E
    sw $zero, 0($t0)
    addi $t0, $t0, 4
    j L
E:
```

- L'uso diretto dei puntatori riduce i calcoli necessari per determinare l'indirizzo di memoria dell'elemento i-esimo (specialmente dentro il ciclo).



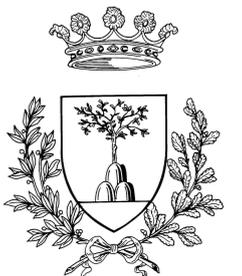
Direttive principali

- Le direttive sono parole precedute da un punto.
- Servono per istruire l'assembler su come interpretare il codice.
- In un programma MIPS si possono distinguere due parti:
 - **.data** quello che segue sono dati da inserire in memoria.
 - **.text** quello che segue sono le istruzioni del programma.

```
.data  
    # ...  
  
.text  
    # ...
```

ATTENZIONE

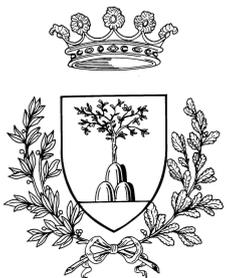
Se utilizzate QtSpim dovete aggiungere l'etichetta "main:" subito dopo il .text



Direttive principali

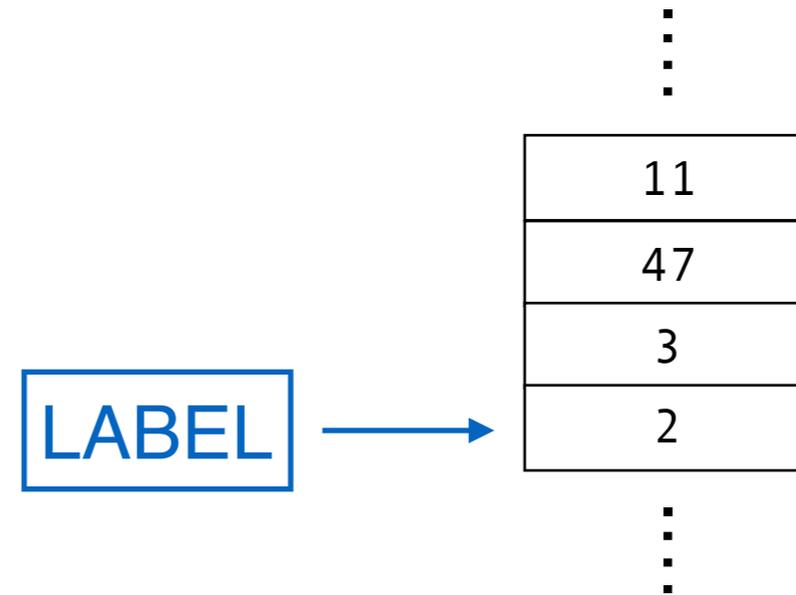
- I dati in `.data` si possono descrivere attraverso speciali direttive:
 - **.word** inizializza un array di elementi in cui ogni valore è una parola di 4 byte
 - **.byte** inizializza un array di elementi in cui ogni valore è 1 byte
 - **.space** riserva N bytes
 - **.asciiz** specifica una stringa

```
.data  
A: .word 1, 2, 3, 4, 5  
B: .byte 1, 2, 3, 4, 5  
C: .space 20  
D: .asciiz "hello world"
```



Esempio di programma

```
.data
    LABEL: .word 2, 3, 47, 11
.text
    la $t0, LABEL
    lw $t1, 0($t0)
    # ...
```



- LABEL è una etichetta che indica l'indirizzo di partenza dell'array formato da 4 interi di 4 byte l'uno.
- La pseudo-istruzione **la \$rd, LABEL** permette di caricare in un registro l'indirizzo associato ad una etichetta.

