Istruzioni di controllo

- **Istruzioni per prendere le decisioni**
  - alterano il controllo del flusso (sequenziale)
  - cambiano quindi la prossima istruzione da eseguire
- **Istruzioni MIPS di salto condizionato *(I-type)***

```
bne $t0, $t1, Label   # branch if not equal
beq $t0, $t1, Label   # branch if equal
```

- **Esempio:** `if (i==j) h = i + j;`

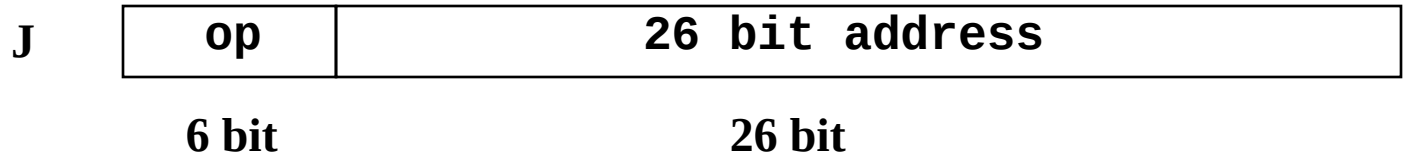
```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:      ....
```

Istruzioni di controllo

- **Salto non condizionato**

```
j   label
```

- **Formato *J-Type***



- **Esempio (costrutto *if...then...else*):**

```
if (i != j) h=i+j;
 else h=i-j;
```

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

Esempio: costrutto `while`

`while (vett[i] == k) i=i+j;`

- i, j e k sono contenuti nei registri `$s3`, `$s4` e `$s5` e l'indirizzo base di **`vett`** in `$s6`

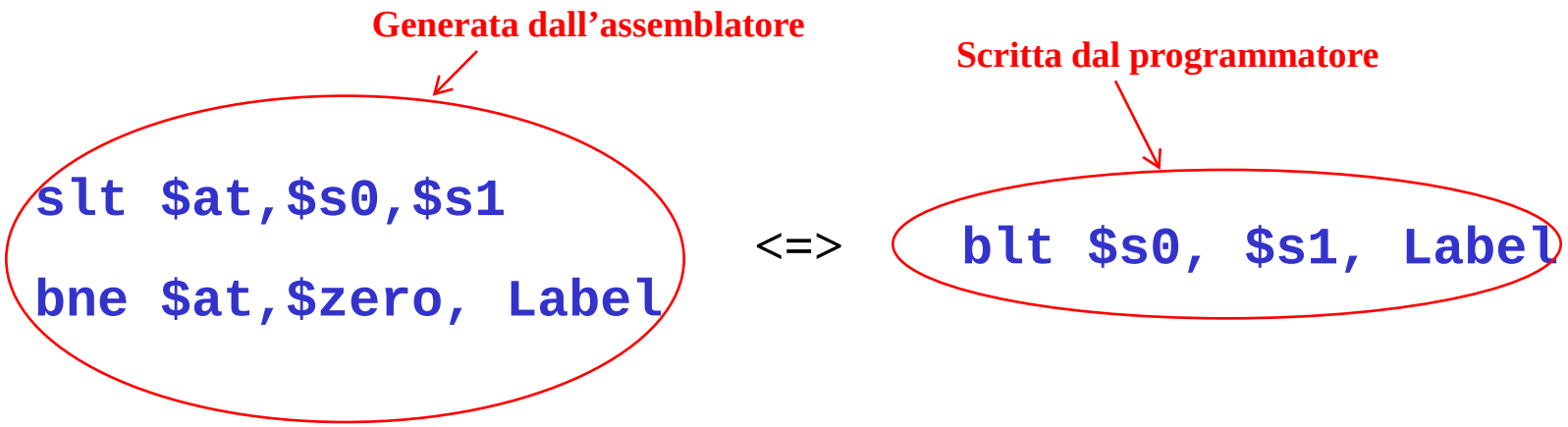
```
ciclo:add $t1, $s3, $s3       # t1 =2*i
      add $t1, $t1, $t1       # t1 =4*i
      add $t1, $t1, $s6       # t1 = indirizzo di vett[i]
      lw  $t0, 0($t1)         # t0 = vett[i]
      bne $t0, $s5, Esci      # vai a esci se
                              # vett[i] != k
      add $s3, $s3, $s4       # i=i+j
      j ciclo                 # vai a ciclo
Esci:       ...
```

Istruzioni di controllo

- **Istruzione *set on less than* (*R-type*):**

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

- **Questa istruzione è anche utilizzata nella "`blt`", che è una *pseudoistruzione* (è messa a disposizione dall'assemblatore ma non è implementata in hardware). La "`blt`" utilizza il registro `$at`**



Esempio: costrutto `for`

- **C code:**

```
for (i=0; i <= n; i = i+k)
   a = a+b;
```

- **MIPS:**

```
      add $t0, $zero, $zero # i=0
Loop:
      blt $s3, $t0, Exit    # if (n<i) goto Exit
      add $s1, $s1, $s2     # a = a+b;
      add $t0, $t0, $t1     # i = i+k;
      j   Loop              # goto Loop
Exit:
```

Esempio: comando `case/switch`

```
switch (k){
    case 0: f = i+j; break;
    case 1: f = g+h; break;
    case 2: f = g-h; break;
    case 3: f = i-j; break;
  }
```

- può essere tradotto in una catena di if-then-else
- caso peggiore: tempo di esecuzione proporzionale al numero di casi
- può essere reso **più veloce** utilizzando una **tabella degli indirizzi di salto** (***jump address table***) dove si trovano gli indirizzi delle sequenze di istruzioni = ***vettore con indirizzi delle etichette***

- **Nuova istruzione (*R-type*):**

  salto tramite registro `jr $t0` (*jump register*)

Comando `case/switch`

Assembler code:

1. Test se 0 <= k <= 3
2. $t0 = JAT[k] (= address Lk)
3. Salta a $t0
4. Esegue il codice

Jump Address Table (JAT):

$t4	address L0
$t4+4	address L1
$t4+8	address L2
$t4+12	address L3

$t4 <-> baseJAT $t2 = 4
$s0 <-> f $s1 <-> g
$s2 <-> h $s3 <-> i
$s4 <-> j $s5 <-> k

```
        slt  $t3, $s5, $zero     } k>=0
        bne  $t3, $zero, Exit    }
        slt  $t3, $s5, $t2       } k<=3
        beq  $t3, $zero, Exit    }
        add  $t1, $s5, $s5       } $t1=4k
        add  $t1, $t1, $t1       }
        add  $t1, $t1, $t4
        lw   $t0, 0($t1)
        jr   $t0
L0:     add  $s0, $s3, $s4
        j    Exit
L1:     add  $s0, $s1, $s2
        j    Exit
L2:     sub  $s0, $s1, $s2
        j    Exit
L3:     sub  $s0, $s3, $s4
Exit:
```

Altre istruzioni MIPS: operandi *Immediate*

- **Le *costanti* sono utilizzate frequentemente (il 50% delle operazioni aritmetiche contiene una costante).**

  Esempio: A = B + 5;

- **Possibile soluzione:**
  - mettere le costanti in memoria e caricarle nei registri
- **Istruzioni MIPS per manipolare costanti:**
  - particolari versioni delle istruzioni aritmetiche (sempre *I-type*)
  - **operando immediato** di 16 bits

```
addi $sp, $sp, 4
ori  $t0, $s1, 19
slti $t0, $s2, 10
```

- **<u>Principio di progetto</u>: rendere veloce l'evento più frequente (costanti come parte dell'istruzione, rende la loro esecuzione più veloce)**

Come caricare costanti lunghe 32 bits

- **Si procede in due passi**
- **Esempio: caricare `0x aabbccdd` in `$t0`.**

1. Si caricano i 16 bit più significativi (**`0xaabb`**):

```
lui $t0, 0xaabb
```

   - (`0xaabb` = `1010 1010 1011 1011`)
   - **Bit meno significativi posti a zero!**

2. Si caricano i 16 bit meno significativi (**`0xccdd`**):

```
ori  $t0, $t0, 0xccdd
```

   - `(0xccdd = 1100 1100 1101 1101)`
   - **Bit più significativi inalterati**

0xaabb	0x0000
0x0000	0xccdd

0xaabb	0xccdd

Esempi di caricamento di costanti

- **Load 0x12345678 in $t0** **(li $t0, 0x12345678)**

pseudoistruzione

```
lui  $t0, 0x1234
ori  $t0, 0x5678
```

Zero Fill

1234	0000
1234	5678

- **Load 0x1234 in $t0** **(li $t0, 0x1234)**

```
addi $t0, $zero, 0x1234
```

Sign Extended

0000	1234

- **Load -1 in $t0** **(li $t0, -1)**

```
addi $t0, $zero, -1
```

Sign Extended

FFFF	FFFF

Altre istruzioni MIPS: Stringhe

- **I caratteri sono raggruppati in stringhe**
- **Necessità di istruzioni per trasferire i byte: `lb,sb`**
  - `lb $t0, 0($sp)` `# legge un byte dalla sorgente`
  - `sb $t0, 0($gp)` `# scrive un byte nella destinazione`
- **NB: per il trasferimento si considerano gli 8 bit *meno significativi* dei registri. Nel caso di `lb`, anche quelli più significativi vengono in generale modificati (v. oltre).**

Tabella ASCII standard

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`	128	80	Ç	160	A0	á	192	C0	└	224	E0	α
1	01	Start of heading	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
2	02	Start of text	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
3	03	End of text	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ú	195	C3	├	227	E3	π
4	04	End of transmit	36	24	$	68	44	D	100	64	d	132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
5	05	Enquiry	37	25	%	69	45	E	101	65	e	133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
6	06	Acknowledge	38	26	&	70	46	F	102	66	f	134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
7	07	Audible bell	39	27	'	71	47	G	103	67	g	135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
8	08	Backspace	40	28	(	72	48	H	104	68	h	136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i	137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j	138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m	141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n	142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o	143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
16	10	Data link escape	48	30	0	80	50	P	112	70	p	144	90	É	176	B0	░	208	D0	╨	240	F0	≡
17	11	Device control 1	49	31	1	81	51	Q	113	71	q	145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
18	12	Device control 2	50	32	2	82	52	R	114	72	r	146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
19	13	Device control 3	51	33	3	83	53	S	115	73	s	147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
20	14	Device control 4	52	34	4	84	54	T	116	74	t	148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u	149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v	150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
23	17	End trans. block	55	37	7	87	57	W	119	77	w	151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
24	18	Cancel	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
25	19	End of medium	57	39	9	89	59	Y	121	79	y	153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z	154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{	155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|	156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}	157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~	158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	⌂	159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

Indirizzamento alla *half-word*

- **Per trasferire coppie di byte (o caratteri) contigui**

- **`lh $t0, Imm($s0)`**

  # la half-word meno significativa di `$t0` prende la half-word meno significativa di `M[Imm+$s0]`; l'altra half-word di `$t0` viene modificata secondo il segno del dato caricato.

- **Nota: esistono anche versioni "unsigned" di `lb` e `lh`, cioè `lbu` e `lhu` (v. oltre…)**

Estensione di segno: operandi di 16 e 8 bit

- **Gli operandi *Immediate* a 16 bit delle istruzioni aritmetiche (es. `addi`) e logiche (es. `ori`, v. oltre) sono estesi a 32 bit nel seguente modo:**
  - gli operandi ***logici*** immediati sono "zero extended"
  - gli operandi ***aritmetici*** immediati sono "sign extended" (incluso `addiu`)

- **I dati caricati da `lb` (8 bit) e `lh` (16 bit) sono estesi a 32 bit nel seguente modo:**
  - `lbu, lhu` "zero extended"
  - `lb, lh` "sign extended"

Moltiplicazione e divisione: `mult` e `div`

- **(`Hi,Lo`): coppia di registri 32 bit che contengono**
  - il **prodotto** di una moltiplicazione.
  - il **resto** ed il **quoziente**, rispettivamente, di una divisione.
- **Istruzioni `mflo,mfhi` per accedere ai registri `Hi` e `Lo`**
- **Esempio:**
  - `mult $s2,$s3`
  - `mflo $s1  #$s1 = Lo (copia di Lo in $s1)`

Istruzione	Esempio	Significato	Commenti
move from Hi	mfhi $s1	$s1 = Hi	*Copia Hi in $s1*
move from Lo	mflo $s1	$s1 = Lo	*Copia Lo in $s1*
move to Hi	mthi $s1	Hi = $s1	*Copia $s1 in Hi*
move to Lo	mtlo $s1	Lo = $s1	*Copia $s1 in Lo*

- **`div` e `mult` operano su numeri *signed* (-2^{31}… $2^{31}-1$).**

  Versioni *unsigned*: `multu, divu` operano su numeri *unsigned* (0… $2^{32}-1$).

Operazioni logiche

AND e OR bit a bit

- **Esempio 1:** `and $t0, $t1, $t2`

  0000 0000 0000 0000 0110 1010 1110
  0000 0000 0000 0000 1100 0001 1100
  0000 0000 0000 0000 0100 0000 1100

- **Esempio 2:** `or $t0, $t1, $t2`

  0000 0000 0000 0000 0110 1010 1110
  0000 0000 0000 0000 1100 0001 1100
  0000 0000 0000 0000 1110 1011 1110

- **Esistono versioni con operando *immediate* a 16 bit, es:**

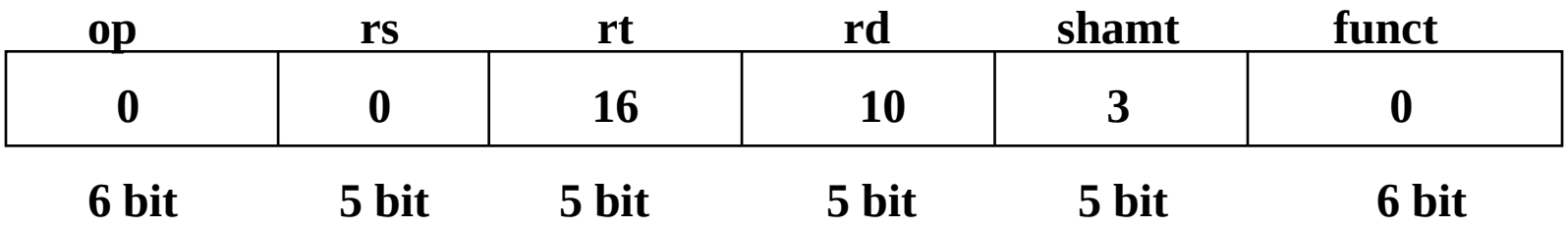
```
ori $t0, $t1, 0xFFFF
```

Ricorda: gli operandi logici immediati sono "zero extended", quindi 0xFFFF esteso a 32 bit = 0000 0000 0000 0000 1111 1111 1111 1111

Operazioni logiche

- **A volte è utile lavorare sui singoli bit all'interno di una parola**
- **Shift (scalamento): sposta tutti i bit di una parola verso sinistra o verso destra riempiendo con degli 0 i bit rimasti vuoti**
- **shift left logical (`sll`) e shift right logical (`srl`)**
- **Esempio: `sll $t2, $s0, 3`**

(`$s0`) 0000 0000 0000 0000 0000 0000 0000 1010
(`$t2`) 0000 0000 0000 0000 0000 0000 0101 0000



Logical shifts for performing multiplications or divisions

- Logical shifts can be useful as efficient ways of performing multiplication or division of integers by powers of two.
  - Shifting left by n bits on a **signed** or **unsigned** binary number has the effect of multiplying it by 2^n.
    - NOTA: **sll non fa alcun controllo sull'overflow**, che quindi occorre quando il valore dell'intero dopo lo shift (quindi dopo essere stato moltiplicato per 2^n) non è rappresentabile. Es: sll $t2, $s0, 1
      - ($s0) 1000 0000 0000 0000 0000 0000 0000 0000
      - ($t2) 0000 0000 0000 0000 0000 0000 0000 0000

        Se $s0 è un numero con segno, allora lo shift ha l'effetto di moltiplicare per 2 il valore -2^{31}, ma -2^{32} non appartiene all'intervallo dei valori rappresentabili $[-2^{31},+2^{31}-1]$, e come risultato si ha che $t0 = 0.

        Se $s0 è un numero senza segno, allora lo shift ha l'effetto di moltiplicare per 2 il valore $+2^{31}$, ma $+2^{32}$ non appartiene all'intervallo di valori rappresentabili $[0,+2^{32}-1]$, e come risultato si ha che $t0 = 0.
  - Shifting right by n bits on an **unsigned** binary number has the effect of dividing it by 2^n (rounding towards 0).

Overflow

- **Situazione in cui il risultato dell'operazione non è rappresentabile dall'hw disponibile**
- **I numeri rappresentabili sono di due tipi**
  - da 0 a (2^{32} -1) (unsigned)
  - da -2^{31} a (2^{31}-1) (signed)
- **Nel processore MIPS, le istruzioni `add`, `addi`, `sub` causano eccezione nel caso di overflow**

  Condizioni di overflow con `add`, `addi`, `sub` (numeri signed):
  - A>0; B>0 e si ottiene A+B<0
  - A<0; B<0 e si ottiene A+B≥0
  - A ≥0; B<0 e si ottiene A-B<0
  - A<0;B ≥0 e si ottiene A-B ≥0
- **Le operazioni `addu`,`subu`,`mult`,`multu`,`div` e `divu` non fanno alcun controllo sull'overflow**