Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Controllo di flusso

Controllo di flusso nei linguaggi ad alto livello

Costrutti IF:

```
If(condizione){
 /*then*/
}
```

```
if (condizione) {
/*then*/
}
else {
}
```

Costrutto SWITCH:

```
switch (expr) {
        case 1: {…}
        case 2: {…}
        case 3: {…}
        default: {…}
}
```

Cicli (loops):

```
while (condizione) {
        fai qualcosa
}
```

```
do {
 fai qualcosa
}
while ( condizione )
```

```
for ( init ; condiz ; passo ) {
        fai qualcosa
}
```

Controllo di flusso nei linguaggi ad alto livello

Per esempio (in Go)

```
voti := [] int { 28, 21, 30, 18, 18 }
somma := 0
for i := 0; i < 5; i++ {
    somma += voti[ i ]
}
```

Controllo di flusso nei linguaggi a basso livello

Il controllo di flusso a basso livello si ottiene cambiando, a runtime, l'indirizzo della prossima istruzione da eseguire

- **PC** (program counter) = indirizzo della prossima istruzione da eseguire
- Di default PC viene automaticamente incrementato per andare all'istruzione successiva: **PC ← PC+4** (ricorda: la differenza tra indirizzi contigui è 4 byte)
- Modifica del flusso: in PC viene scritto il target address di un'istruzione diversa dalla successiva

Come possiamo farlo? Con due tipologie di istruzioni:

- Salti incondizionati, detti **«jump»**: cambiano sempre l'indirizzo della prossima istruzione
- Salti condizionati, detti **«branch»**: cambiano l'indirizzo della prossima istruzione se si verifica una data **condizione**

Salti Incondizionati (Jump)

- **Incondizionato** significa che il salto viene **sempre** eseguito
- Istruzioni: `j` (jump), `jr` (jump register)

```
j   INDIRIZZO   # salta a un dato indirizzo
```

Esempio:

```
J 0x00400084
```

```
jr  $rx # salta all'indirizzo contenuto in $rx
```

Esempio:

```
la $s1 0x00400084
jr $s1
```

Salti Incondizionati (Jump)

- **Incondizionato** significa che il salto viene **sempre** eseguito
- Istruzioni: `j` (jump), `jr` (jump register)

```
j   INDIRIZZO   # salta a un dato indirizzo
```

Esempio:

```
J 0x00400084
```

?

```
jr  $rx # salta all'indirizzo contenuto in $rx
```

Esempio:

```
la $s1 0x00400084
jr $s1
```

?

Ma come facciamo a conoscere l'indirizzo delle istruzioni a cui vogliamo saltare mentre scriviamo il nostro programma? Con le **label**

Label

- Se scriviamo "`Label1:` *element*" Assembler assocerà l'identificatore `Label1` **all'indirizzo** di *element*
- *element* può essere un **dato** o un'**istruzione**, quindi le label possono essere usate sia nel segmento dati che nel segmento testo

Le posso dichiarare così

Nel codice **array1** indicherà l'indirizzo di un array con 4 interi che sta nel segmento dati

Nel codice **blocco1** indicherà l'indirizzo della **add**

```
        .data
# dati ...
array1: .word 45 67 -3 7
# dati ...

        .text
        # istruzioni ...
blocco1:
        add $t0 $t0 $t1
        li $t2 4
        mul $t0 $t0
        # istruzioni ...

        la $s0 array1
        j blocco1
```

Le posso usare così

Carico un indirizzo nel registro

Salto alla **add**

Label

- Se scriviamo "`Label1:` *element*" Assembler assocerà l'identificatore `Label1` **all'indirizzo** di *element*
- *element* può essere un **dato** o un'**istruzione**, quindi le label possono essere usate sia nel segmento dati che nel segmento testo

Le posso dichiarare così

Nel codice **array1** indicherà l'indirizzo di un array con 4 interi che sta nel segmento dati

Nel codice **blocco1** indicherà l'indirizzo della **add**

```
        .data
# dati ...
array1: .word 45 67 -3 7
# dati ...

        .text
        # istruzioni ...
blocco1:
        add $t0 $t0 $t1
        li $t2 4
        mul $t0 $t0
        # istruzioni ...

        la $s0 array1
        j blocco1
```

Non devo preoccuparmi di conoscere i valori numerici degli indirizzi a cui dati e istruzioni verranno memorizzate!

Le posso usare così

Carico un indirizzo nel registro

Salto alla **add**

Esempio

```
        .text
        .globl main
main:
        li $t0 4
        li $t1 5
        j qui
        li $t0 0
qua:
        li $t1 0
qui:
        add $t0 $t1 $t0
        j qua
```

Quanto vale t0 alla fine?

Esempio

```
        .text
        .globl main
main:
        li $t0 4
        li $t1 5
        j qui
        li $t0 0
qua:
        li $t1 0
qui:
        add $t0 $t1 $t0
        j qua
```

Quanto vale t0 alla fine?

Non termina mai!! 😐

Proviamo a correggere…

Esempio

```
        .text
        .globl main
main:
        li $t0 4
        li $t1 5
        j qui
        li $t0 0
qua:
        li $t1 0
        j end
qui:
        add $t0 $t1 $t0
        j qua
end:
```

Quanto vale t0 alla fine?

Esempio

```
        .text
        .globl main
main:
        li $t0 4
        li $t1 5
        j qui
        li $t0 0
qua:
        li $t1 0
        j end
qui:
        add $t0 $t1 $t0
        j qua
end:
```

Quanto vale t0 alla fine?

Risposta: 9

Jump in linguaggio macchina

- Il salto `j` (e anche `jal`, che vedremo poi) è un'istruzione J-type (J sta per Jump):

```
j  label
```

- Il **target address** è di 32 bit (come ogni indirizzo in MIPS32)

Problema: nell'istruzione ci sono solo 26 bit per specificare il target address
Soluzione: indirizzamento **pseudo-diretto**:

1. I bit in posizione 0 e 1 (i due meno significativi) sono **impliciti** ed uguali a 0 (allineamento)
2. I bit dalla posizione 2 alla 25 sono uguali ai 26 bit specificati nell'istruzione
3. I bit dalla posizione 26 alla 31 (i quattro più significativi) sono **impliciti** ed uguali ai quattro bit più significativi del PC

Effetto della jump:

Salti in linguaggio macchina MIPS

Il salto Jump:

- non può modificare i primi 4 bit del PC
  - per esempio, una jump all'indirizzo `0xC-------`
    può saltare solo ad un'altra istruzione di indirizzo `0xC-------`
  - Si dice che non può saltare «fuori dal blocco»

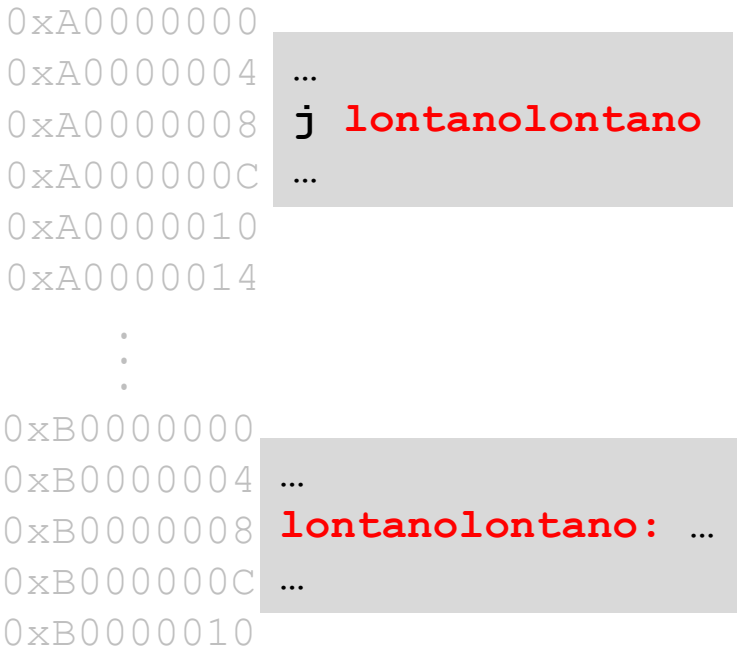
L'istruzione Jump Register non ha questa limitazione:

```
jr  $rx
```

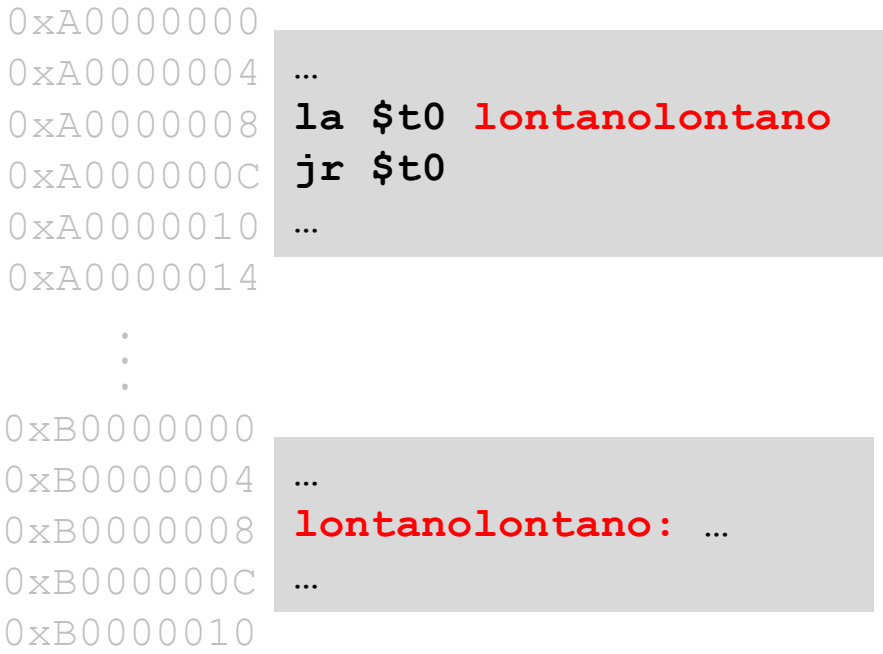
- Il target address sta dentro il registro $rx, non è un operando specificato dentro all'istruzione (nell'istruzione si specifica il numero di registro per cui bastano 5 bit)
- Non sarà Assembler a costruire il target address, dobbiamo farlo noi caricandone il valore nel registro $rx

Salti in linguaggio macchina MIPS

- Con jump register posso saltare «fuori dal blocco»



ERRORE: il target address è troppo distante



OK!

Branch – Bivio, Biforcazione

- Salto **condizionato**: viene eseguito solo se una certa **condizione** risulta verificata, altrimenti si continua normalmente con la prossima istruzione

- Esempio: **branch on equal**

```
beq $ra $rb label
```

- Se i registri `$ra` e `$rb` contengono lo stesso valore,
  allora salta all'istruzione memorizzata all'indirizzo rappresentato da *`label`*

Instruzioni di Branch

- Con confronto fra due registri

beq `$ra $rb` *addr*	branch on *equal*	$ra = $rb
bne `$ra $rb` *addr*	branch on *not equal*	$ra ≠ $rb
blt `$ra $rb` *addr*	branch on *less then*	$ra < $rb

- Con confronto fra registro e zero

bgez `$ra` *addr*	branch on *greater-or-equal zero*	$ra ≥ 0
bgtz `$ra` *addr*	branch on *greater-than zero*	$ra > 0
blez `$ra` *addr*	branch on *less-or-equal to zero*	$ra ≤ 0
bltz `$ra` *addr*	branch on *less-than zero*	$ra < 0

Branch in linguaggio macchina

- I Branch sono istruzioni I-type (I sta per Immediate)

```
beq $ra $rb label
```

Problema: nell'istruzione ci sono solo 16 bit per specificare il target address
Soluzione: indirizzamento **relativo al PC (PC-relative)**:

1. I bit in posizione 0 e 1 (i due meno significativi) sono **impliciti** ed uguali a 0 (allineamento)
2. I bit dalla posizione 2 alla 17 sono uguali ai 16 bit specificati nell'istruzione
3. I bit dalla posizione 17 alla 31 sono l'estensione del segno

Effetto della branch se il salto viene fatto:

Branch in linguaggio macchina

- L'offset sommato al PC è un numero in complemento a 2 ed è relativo all'**istruzione successiva** alla branch
- Massimo salto in avanti: $+4(2^{15}-1)$ bytes dall'istruzione successiva alla branch, quindi 2^{15} istruzioni **dopo** quella corrente
- Massimo salto all'indietro: $-4(2^{15})$ bytes dall'istruzione successiva alla branch quindi $2^{15}-1$ istruzioni **prima** di quella corrente
- Sono salti «corti», ma si può uscire dal blocco. Ad esempio posso saltare da 0x**A**FFFFFFE a di 0x**B**0000000

Nota: quando scrivo `beq $ra $rb label`
Assembler fa per noi il lavoro di ricostruire l'offset di 16 bit a partire dalla label che ho specificato:

- Sottrae all'indirizzo specificato dalla label l'indirizzo dell'istruzione successiva al branch
- se l'indirizzo target è troppo distante ($>2^{15}$) genera un errore

Posso saltare lontano condizionalmente?

- Sì, combinandolo con jump:

```
0xA0000000
0xA0000004   …
0xA0000008   bgez $t0 far
0xA000000C   …
0xA0000010
0xA0000014
   . . .
0xA5130000
0xA5130004   …
0xA5130008   far: …
0xA513000C   …
0xA5130010
```

ERRORE! too far!

```
0xA0000000
0xA0000004   …
0xA0000008   bltz $t0 near
0xA000000C   j far
0xA0000010   near: …
0xA0000014
   . . .
0xA5130000
0xA5130004   …
0xA5130008   far: …
0xA513000C   …
0xA5130010
```

OK

Condizioni di disuguaglianza

- Spesso è utile condizionare l'esecuzione di un'istruzione al fatto che una variabile sia minore di un'altra, istruzione **Set Less Than**.

```
slt $s1, $s2, $s3
```

- Assegna il valore 1 (set) a `$s1` se `$s2 < $s3` altrimenti assegna il valore 0.
- Con `slt, beq` e `bne` si possono implementare tutti i test sui valori di due variabili `(==, !=, <, <=, >,>=)`.

Condizioni di disuguaglianza

- Si completi la seguente tabella con il corrispettivo codice assembly

Pseudo codice	Assembly
if($s1==$s2) addi $s3, $s3, 1	
if($s1!=$s2) addi $s3, $s3, 1	
if($s1>$s2) addi $s3, $s3, 1	
if($s1>=$s2) addi $s3, $s3, 1	
if($s1<$s2) addi $s3, $s3, 1	
if($s1<=$s2) addi $s3, $s3, 1	

Condizioni di disuguaglianza

- Si completi la seguente tabella con il corrispettivo codice assembly

Pseudo codice	Assembly
if($s1==$s2) addi $s3, $s3, 1	bne $s1, $s2, L addi $s3, $s3, 1 L:
if($s1!=$s2) addi $s3, $s3, 1	beq $s1, $s2, L addi $s3, $s3, 1 L:
if($s1>$s2) addi $s3, $s3, 1	slt $t0, $s2, $s1 bne $t0, 1, L addi $s3, $s3, 1 L:
if($s1>=$s2) addi $s3, $s3, 1	bne $s1, $s2, T j A T: slt $t0, $s2, $s1 bne $t0, 1, L A: addi $s3, $s3, 1 L:
if($s1<$s2) addi $s3, $s3, 1	slt $t0, $s1, $s2 bne $t0, 1, L addi $s3, $s3, 1 L:
if($s1<=$s2) addi $s3, $s3, 1	bne $s1, $s2, T j A T: slt $t0, $s1, $s2 bne $t0, 1, L A: addi $s3, $s3, 1 L:

Alcune strutture di controllo di alto livello in Assembly

If - Then

Codice C:

```
if (i==j)
    f=g+h;
…
```

Si supponga che le variabili `f, g, h, i` e `j` siano associate rispettivamente ai registri `$s0, $s1, $s2, $s3` e `$s4`

- Riscriviamo il codice C in una forma equivalente, ma più «vicina» alla sua traduzione Assembly

↓

```
if (i!=j)
    goto L;
f=g+h;
L:
…
```

→

Codice Assembly:

```
bne $s3, $s4, L                    # if i ≠ j
go to L
add $s0, $s1, $s2
L:
…
```

If - Then - Else

Codice C:

```
if (i==j)
    f=g+h;
else
    f=g-h
…
```

Si supponga che le variabili `f, g, h, i` e `j` siano associate rispettivamente ai registri `$s0, $s1, $s2, $s3` e `$s4`

Codice Assembly:

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j End
Else:
sub $s0, $s1, $s2
End:
…
```

Do - While

Codice C:

```
i=0;
do{
   g = g + A[i];
   i = i + j;
}
while (i!=h);
```

Si supponga che:

`g` e `h` siano in `$s1, $s2`

`i` e `j` siano in `$s3, $s4`

`A` sia in `$s5`

- Riscriviamo il codice C:

```
i = 0;
Loop:
g = g + A[i];
i = i + j;
if (i != h)
  goto Loop
```

Codice Assembly:

```
li $s3, 0
Loop:
mul $t1, $s3, 4
add $t1, $t1, $s5
lw $t0, 0($t1)
add $s1, $s1, $t0
add $s3, $s3, $s4
bne $s3, $s2, Loop
```

While

Codice C:

```
while (A[i]==k){
  i=i+j;
}
```

Si supponga che:

`i` e `j` siano in `$s3, $s4`

`k` sia in `$s5`

`A` sia in `$s6`

- Riscriviamo il codice C:

```
Loop:
If (A[i]!=k)
   go to End;
i=i+j;
go to Loop;
```

Codice Assembly:

```
Loop:
mul $t1, $s3, 4
add  $t1, $t1, $s6
lw  $t0, 0($t1)
bne $t0, $s5, End
add $s3, $s3, $s4
j Loop
End:
```

Il costrutto switch

- Può essere implementato con una serie di `if-then-else`
- *Alternativa: uso di una jump address table*

Codice C:

```
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;
default:
   break;
}
```



```
if (k < 0)
   t = 1;
else
   t = 0;
if (t == 1)                          // k < 0
   goto Exit;
t2 = k;
if (t2 == 0)                         // k = 0
   goto L0;
t2--; if (t2 == 0)                   // k = 1
   goto L1;
t2--; if (t2 == 0)                   // k = 2
   goto L2;
t2--; if (t2 == 0)                   // k = 3
   goto L3;
goto Exit;                           // k > 3

L0:  f = i + j; goto Exit;
L1:  f = g + h; goto Exit;
L2:  f = g – h; goto Exit;
L3:  f = i - j; goto Exit;

Exit:
```

Il costrutto switch

- Si supponga che `$s0, …, $s5` contengano `f,g,h,i,j,k`,

Codice Assembly:

```
slt $t3, $s5, $zero
bne $t3, $zero, Exit

beq $s5, $zero, L0

addi $s5, $s5, -1
beq $s5, $zero, L1

addi $s5, $s5, -1
beq $s5, $zero, L2

addi $s5, $s5, -1
beq $s5, $zero, L3
```

```
j Exit;

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:
```

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio