Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2018-2019

Controllo di flusso

Turno A
Nicola Basilico
Dipartimento di Informatica
Via Comelico 39/41 - 20135 Milano (MI)
Ufficio S242
nicola.basilico@unimi.it
+39 02.503.16294

Turno B
Marco Tarini
Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4010
marco.tarini@unimi.it
+39 02.503.16217

Controllo di flusso: nei linguaggi ad alto livello

Es: in C / C++ / Java / Go

Costrutti condizionali:

- `if (…) /*then*/ { … }`
- `if (…) /*then*/ { … } else { … }`

Costrutto switch:

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

Controllo di flusso: nei linguaggi ad alto livello

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 ]
}
```

O, ancora più ad alto livello…

```
voti := [] int { 28, 21, 30, 18, 18 }
somma := 0
for _ , voto := range voti {
    somma += voto
}
```

Controllo di flusso: nei linguaggi a basso livello

A basso livello, il controllo di flusso viene realizzato usando solo due costrutti:

- «Jump» (salto): cambia l'indirizzo della prossima istruzione.
- «Branch» (bivio, salto condizionato): come sopra, ma scatta solo se si verifica una data condizione

Entrambi hanno l'effetto di modificare il **PC**

- **PC** (program counter) = uno speciale registro che memorizza l'indirizzo della prossima istruzione da eseguire
- Con qualsiasi altra istruzione, il PC viene automaticamente incrementato per andare all'istruzione successiva:
  PC ← PC+4 (una istruzione MIPS = 4 bytes!)
- In caso di salto: il PC viene invece sostituito da un dato target address (specificato nell'istruzione di Jump / Branch)

Jump - Salto

- **E incondizionato** ("unconditional"): Il salto viene sempre eseguito
- `j` (jump), `jr` (jump register)

```
j   VALORE    # salta a un dato valore.
              # PC = Label (invece di PC+4)

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

Jump - Salto

Istruzione	*Esempio*	*Significato*
jump	`j 10000`	vai all' istruzione 10000 cioè PC ← 10000 (invece di PC+4)
jump register	`jr $12`	vai all'istruzione di indirizzo "valore di $12" cioè PC ← $12 (invece di PC+4)

Come trovare l'indirizzo di salto?

- Etichette!
- Nota: le etichette possono essere usate tanto nel segmento code quanto nel segmento data
- In ogni caso,
  registrano l'address del dato o istruzione immediatamente seguente

Esempio

```
.text:
...
...   # questa parte viene eseguita
...
j qui
...
... # codice che NON viene eseguito!
...
qui:
...
...   # questa parte viene eseguita
...
```

Esempio

```
.text:
li $t0 4
li $t1 5

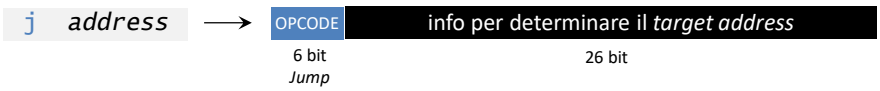
j qui
li $t0 0
li $t1 0

qui:
add $t0 $t1 $t0
```

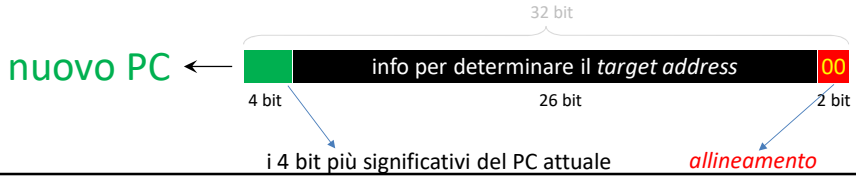
Quanto vale $t0 alla fine?

Salto in linguaggio macchina MIPS

- Il salto `j` si mappa su un'istruzione di **formato** J-type (J sta per Jump!):



- Il **target address** è di 32 bit (come ogni address MIPS)
- Problema: nell'istruzione ci sono solo 26 bit per specificarli!
- Espediente 1: consideriamo 2 bit sottointesi alla fine: 00 (allineamento)
- Espediente 2: i primi 4 bit dell'address rimangono quelli del PC corrente
- Detta **Modalità di indirizzamento pseudo-diretta**



Salti in linguaggio macchina MIPS

Conseguenze: il salto Jump

- non può modificare i primi 4 bits del PC
  - per es, una jump all'indirizzo `0xC-------`
    può saltare solo ad un'altra istruzione di indirizzo `0xC-------`
  - non può saltare fuori dal suo «blocco» di istruzioni
- salta sempre ad istruzioni allineate al word
  - ma è ok, tanto sarebbe un errore fare altrimenti.

blocco delle parole di RAM ad indirizzi che iniziano per `0xC`

Nota: Jump Register non invece ha alcuna limitazione!

- registro = address = 32 bit

Nota: l'assembler fa per noi il lavoro di ricostruire
i 26 bit a partire dal *target address*

Posso saltare “fuori dal blocco”?

```
0xA0000000
0xA0000004  …
0xA0000008  j foo
0xA000000C  …
0xA0000010
0xA0000014
   ...
0xB0000000
0xB0000004  …
0xB0000008  foo: …
0xB000000C  …
0xB0000010
```

ERRORE

- Si: con la Jump Register:

```
0xA0000000
0xA0000004  …
0xA0000008  la $t0 foo
0xA000000C  jr $t0
0xA0000010  …
0xA0000014
   ...
0xB0000000
0xB0000004  …
0xB0000008  foo: …
0xB000000C  …
0xB0000010
```

OK

Inizializzazione diretta degli indirizzi

- Come fare a caricare un indirizzo in un registro?
  Esempio: caricare in `$s1` l’indirizzo `0x1A0BC204` ricorda: registro = 32 bit
- Tentativo di risposta: `li $s1 0x1A0BC204` (0x1A0BC204: 32 bit)
- Questa pesudoistruzione verrebbe tradotta come: `addi $s1 $0 0x1A0BC204` (0x1A0BC204: 32 bit)
- ERRORE!
  Il valore «immediato» di `addi` deve essere un intero a max 16 bit!
  (l’intera istruzione occupa solo 32 bit)

Inizializzazione diretta degli indirizzi

- Come fa `la` («**load address**») a caricare un indirizzo in un registro?
  Esempio: caricare in `$s1` l'indirizzo 0x1A0BC204 ricorda: registro = 32 bit
  (16bit 16bit)
- Risposta corretta: servono DUE istruzioni separate, ciascuna per caricare 16 bit.

```
lui   $t0 0x1A0B
```
"Load Upper Immediate": $t0 diventa 0x1A0B0000 (upper 16 bits)

```
ori   $s1 $t0 0xC204
```
"OR Immediate": $s1 diventa:
```
  0x1A0B0000  or
  0x0000C024  =
  ----------
  0x1A0BC024
```

(variante: usare ADDI invece di ORI)

Inizializzazione diretta degli indirizzi

- `la` («**load address**») è una pseudo-istruzione:
- L'assembler la traduce nel modo più opportuno, in una oppure due istruzioni (a seconda della lunghezza indirizzo da copiare nel registro)
  Ad esempio:

```
la   $s1   0x204        ---->   addi $s1 $0  0x0204

la   $s1   0xA010C204   ---->   lui  $at     0xA010
                                ori  $s1 $at 0xC204
```

- Nota: il registro $at (cioè $1) viene potenzialmente sovrascritto!
  Infatti questo registro è per convenzione «riservato dall'assembler».
  at = "assembler temp"
  I programmatori non devono usarlo!
  L'assembler lo usa per tradurre alcune pseudo-istruzioni (come questa).

Branch – Bivio , Biforcazione

- **Cioè salto condizionato:**
  il salto viene eseguito solo se una certa **condizione** risulta verificata,
  altrimenti si passa alla prossima istruzione (come normale)

- Esempio: **branch on equal**

```
beq $ra $rb address
```

Se i registri `$ra` e `$rb` hanno lo stesso valore, allora salta ad *address*

Instruzioni di Branch (bivio)

- Con confronto fra due registri

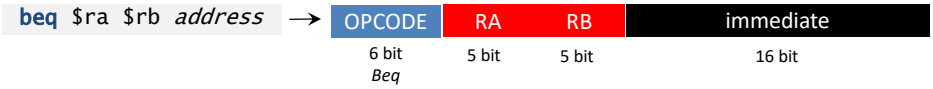
`beq $ra $rb addr`	branch on *equal*	$ra = $rb
`bne $ra $rb addr`	branch on *not equal*	$ra ≠ $rb
`blt $ra $rb addr`	branch on *less then*	$ra < $rb

- Con confronto fra registro e zero

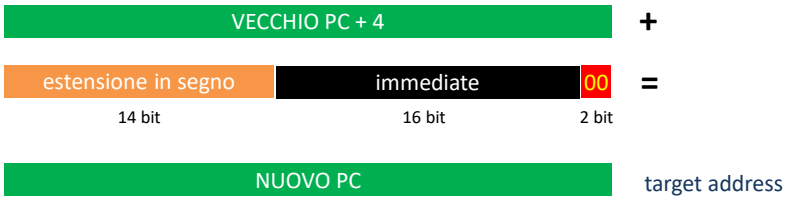
`bgez $ra addr`	branch on *less-than zero*	$ra < 0
`bgtz $ra addr`	branch on *greater-then zero* >0	$ra > 0
`blez $ra addr`	branch on *less-or-equal to zero*	$ra ≤ 0
`bltz $ra addr`	branch on *greater-or-equal zero* >=0	$ra ≥ 0

Branch in linguaggio macchina MIPS

- I Branch in MIPS hanno **formato** I-type



- Problema: il **target address** (come ogni address in MIPS) è di 32 bit.
- Nell'istruzione I-type abbiamo il campo immediate, di 16 bit, per specificarli
- Soluzione: (detta **modalità di indirizzamento Relativa):**



Branch in linguaggio macchina MIPS

Conseguenze: i salti condizionati (Branch)

- possono saltare al max a 2^15 istruzioni sopra o sotto
- possono saltare solo a indirizzi allineati alla parola (questo è ok)
- (possono però uscire dal «blocco» di istruzioni -
  per es: dalla fine di 0xA… posso saltare all'inizio di 0xB...)

Nota:
l'assembler fa per noi il lavoro di ricostruire l'offset di 16 bit

- sottraendo al target address richiesto
  l'indirizzo dell'istruzione (successiva al) branch
- se l'indirizzo target è troppo distante (>2^15),
  genera un errore

Posso saltare “lontano” condizionalmente?

- Si: combiando 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

If – Then: esempio

Codice Go:

```
età := 16
limiteEtà := 18
prezzo := 100
/* sconto per i minorenni */
if età < limiteEtà  { prezzo = 80; }
```

```
.data
mieiDati: .word 16 18 100;
.text
la $t0 mieiDati      # ora $t0 punta a «eta»
lw $s1 ($t0)         # carico «eta» in s1
lw $s2 4($t0)        # carico «limite» in s2
lw $s3 8($t0)        # carico «prezzo» in s3

bge $s1 $s2 end      # se s1 è maggiore o uguale a s2,
                     # vado alla fine e non faccio
li $s3 80            # s3 = 0
end:
```

If – Then else: esempio

Codice
C / C++ / Java:

```
int età = 16 ;
int limiteEtà = 18 ;
int prezzo;

if (età < limiteEtà)  {

  prezzo = 80 ;

} else {

  prezzo = 100 ;

}
```

oppure, meglio:

```
int età = 16 ;
int limiteEtà = 18 ;

int prezzo = (età < limiteEtà ) ? 80 : 100 ;
```

If – Then else: esempio

Codice Go:

```
età := 16
limiteEtà := 18
var prezzo int

if età < limieteEtà  { prezzo = 80 }
else { prezzo = 100 }
```

```
      .data
dati: .word 16 18
      .text
      la $t0 dati    # $t0 punta a «eta»
      lw $s1  ($t0) # carico «eta» in s1
      lw $s2 4($t0) # carico «limite» in s2

      ble $s1 $s2 then    # se s1 è < s2 vado a else
      li $s3 80           # ramo else
      j end
then: li $s3 100
end:  # resto del programma…
```

Do – While: esempio

PseudoCodice di un programma interattivo (legge numeri finchè non ne viene immesso uno neg)

```
do{
   scrivi «inserisci un numero (o negativo per uscire)»;
   leggi numero;
} while (numero >= 0) ;
```

```
.data
msg:  .ascii "inserisci un numero (o negativo per uscire)"
.text
loop: li $v0 4
      la $a0 msg
      syscall

      li $v0 5
      syscall

      bgtz $v0 loop

      li $v0 10
      syscall
```

While do: esempio

PseudoCodice di un programma interattivo

```
do{
   scrivi «inserisci un numero (o negativo per uscire)»;
   leggi numero;
}
while (numero >= 0) ;
```

```
.data
msg:  .ascii "inserisci un numero (o negativo per uscire)"
.text
loop: li $v0 4
      la $a0 msg
      syscall

      li $v0 5
      syscall

      bgtz $v0 loop

      li $v0 10
      syscall
```

While do: esempio

data una sequenza di voti in RAM
trovare la loro somma

Guarda, o terminatore.
Segnala che la lista di voto è terminata
NB: non è un voto!

```
.data
voti: .word 30 28 27 18 -1
.text
loop: la $s1 voti
      move $s0 $zero   # s0 = la somma (parziale) dei voti

      lw $t0 ($s1)     # t0 = il prossimo voto

      bltz $t0 fine

      addi $s1 $s1 4   # ora s4 punta al prossimo voto
      addi $s0 $s0 $t0

      j loop
fine: # resto del codice… Ora, $s0 è la somma dei voti
```

While do: esempio

- Esercizi:
  - verificare cosa succede se la lista di voti è vuota (cioè se in RAM abbiamo solo la guardia, -1). Funziona?
  - trovare il NUMERO di voti, in $s2
  - trovare la MEDIA dei voti (arrotondata per difetto), in $s3, dividendo la somma per il numero di voti
  - evitare la divisione per zero, quando non è presente neanche un voto (in questo caso, la media deve essere 0)
  - stampare la media trovata a schermo

Controllo di flusso: nei linguaggi ad alto livello

Anche alcuni linguaggi ad “alto” livello (es C) prevedono un costrutto del tipo:

goto <locazione>

Si tratta di un salto.

Evitare! E’ sempre possibile usare, al suo posto, uno degli altri costrutti disponibili (if then else, loop, switch…)

- Programmazione “ben strutturata”
- Usare goto porta invece a programmi “spaghetti”

A basso livello (e.g. MIPS) siamo invece obbligati ad usare salti (condizionati e non). Non abbiamo altro, per controllare il flusso.

https://xkcd.com/292/