Calcolatori Elettronici A
a.a. 2008/2009

ISA e LINGUAGGIO ASSEMBLY MIPS

Massimiliano Giacomin

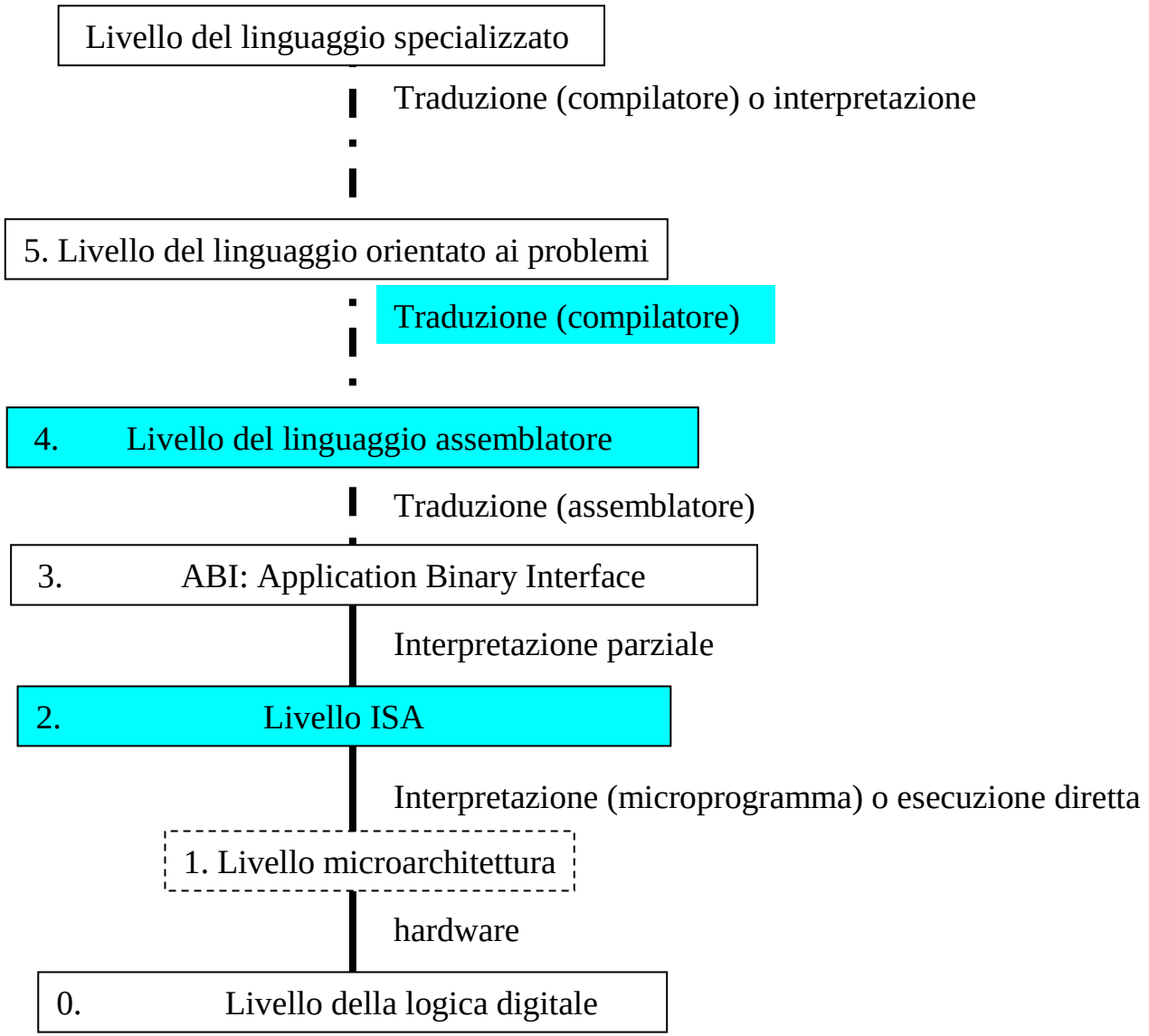
Architettura MIPS

- Architettura **RISC** **e** **load-store** sviluppata da John Hennessy nel 1981
- Usata da NEC, Nintendo, Silicon Graphics, Sony
- Ci permette di apprezzare i principi usati nella progettazione di tutte le ISA moderne
- Vedremo:
  - linguaggio assembly (principali istruzioni)
  - traduzione in linguaggio macchina e viceversa
  - introducendo le istruzioni,
    compilazione di parti di codice C in linguaggio assembly

AVVERTENZA

- Risulta praticamente impossibile parlare separatamente di livello ISA
  e linguaggio assembly:
  - è il motivo per cui è stato sviluppato il linguaggio assembly
  - linguaggio assembly diventa tipicamente uno standard legato al processore

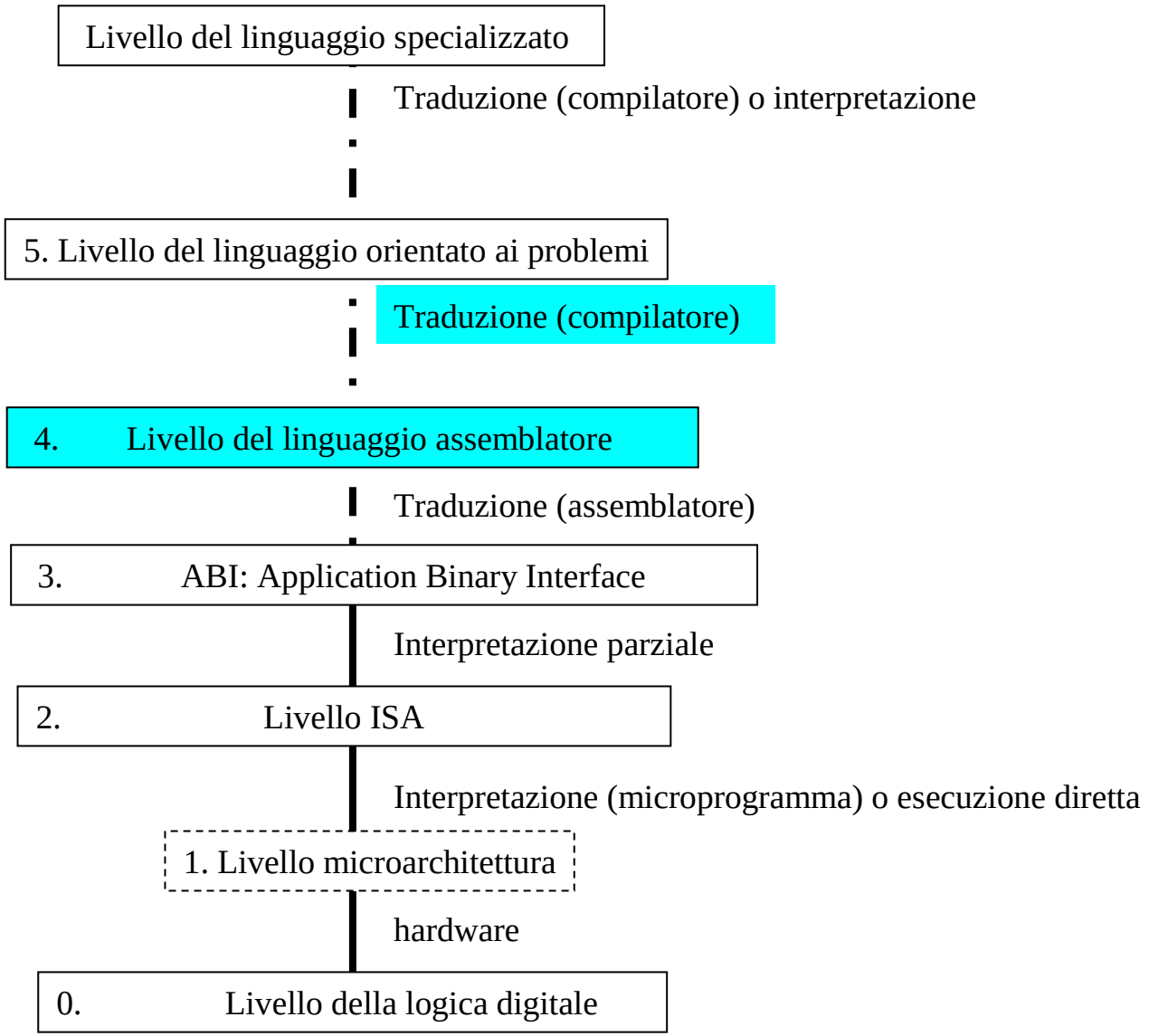
DOVE CI TROVIAMO…



I registri del MIPS

- Il processore MIPS è a 32 bit:
  - tutti i registri sono a 32 bit
  - gli indirizzi per accedere in memoria sono a 32 bit
- Il numero di registri è 32 (identificati da un numero 0-31: servono 5 bit)
- I registri sono general-purpose, ma:
  - vi sono convenzioni sul loro utilizzo, che vedremo di volta in volta (es. alcuni si usano per gestire lo stack, altri per passare o ricevere argomenti a/da una procedura, alcuni sono preservati da una chiamata ad una procedura altri no, uno è riservato al programma assembler per le pseudoistruzioni, ecc.)
  - il registro 0 ha sempre valore nullo
  - il registro 31 è utilizzato dall'istruzione jal per memorizzare l'indirizzo di ritorno da una procedura
- Queste convenzioni/usi specifici sono riflessi nei nomi che in assembly vengono dati ai registri (due caratteri preceduti da $). Per ora consideriamo:
  - $s0… $s7 (preservati, per "variabili non temporanee" – ovvero, registri che sono utilizzati per rappresentare variabili del linguaggio C)
  - $t0… $t9 (non preservati, per "variabili temporanee")

FOCALIZZIAMOCI SU:



Tipologie di istruzioni

- Istruzioni aritmetiche e logiche
- Istruzioni di trasferimento:
  - lw
  - sw
- Istruzioni di controllo:
  - salti condizionati
  - salti incondizionati

  (che includono la chiamata a una procedura e l'istruzione di ritorno)

NB: nel linguaggio assembly

- ogni linea contiene un'istruzione
- i commenti sono preceduti da #

Istruzioni aritmetiche: somma e sottrazione

- Istruzione **add**

```
add     a, b, c          # a = b + c
```

- Istruzione **sub**

```
sub     a, b, c          # a = b - c
```

- Ciascuna istruzione aritmetica MIPS esegue una sola operazione e contiene 3 operandi. Vi sono due possibilità:
  - tutti gli operandi sono registri
  - uno dei due operandi sorgente è una costante (immediate): in questo caso si usa l'istruzione **addi**

- Es. sommare $s1, $s2, $s3, $s4 e porre il risultato in $s0

```
add     $s0, $s1, $s2        # $s0 = $s1+$s2
add     $s0, $s0, $s3        # $s0 = $s1+$s2+$s3
add     $s0, $s0, $s4        # $s0 = $s1+$s2+$s3+$s4
```

Esempio

- Compilare l'istruzione C seguente

  f = (g + h) – (i + j)

  dove le variabili f, g, h, i, j vengono fatte corrispondere ai registri $s0, $s1, $s2, $s3, $s4

- Il programma compilato sarà:

```
add $t0, $s1, $s2     # $t0 contiene g+h
add $t1, $s3, $s4     # $t1 contiene i+j
sub $s0, $t0, $t1     # $s0 assume $t0-$t1 cioè
                        f =(g+h) – (i+j)
```

Somma con operando immediato: l'istruzione addi

```
addi    $s3, $s4, 4     # $s3 = $s4 + 4
```

- Come vedremo, l'operando immediato è rappresentato con 16 bit: in complemento a due, ciò consente di rappresentare valori da -2^{15} a $+(2^{15}-1)$
- Non occorre una istruzione subi, perché la sottrazione si ottiene sommando un numero negativo!

Istruzioni logiche

Shift left

```
sll  rd, rt, const      # rd = rt<<const
```

NB: utile per moltiplicare per 2^{const}

Shift right

```
srl  rd, rt, const      # rd = rt>>const
```

And

```
and  rd, rs, rt         # rd = rs & rt
andi  rt, rs, const     # rt = rs & const
```

NB: utile per applicare una "maschera"

Or

```
or  rd, rs, rt          # rd = rs | rt
ori  rt, rs, const      # rt = rs | const
```

NB: utile per porre a 1 determinati bit

Nor

```
nor  rd, rs, rt         # rd = ~(rs | rt)
```

(NB: nori non disponibile, utilizzo raro)

NB: NOT si ottiene con nor $rd, $rt, $zero

Istruzioni di trasferimento dati: load word (lw)

- Permette di trasferire una parola di memoria (32 bit, 4 bytes) in un registro
- Sintassi:

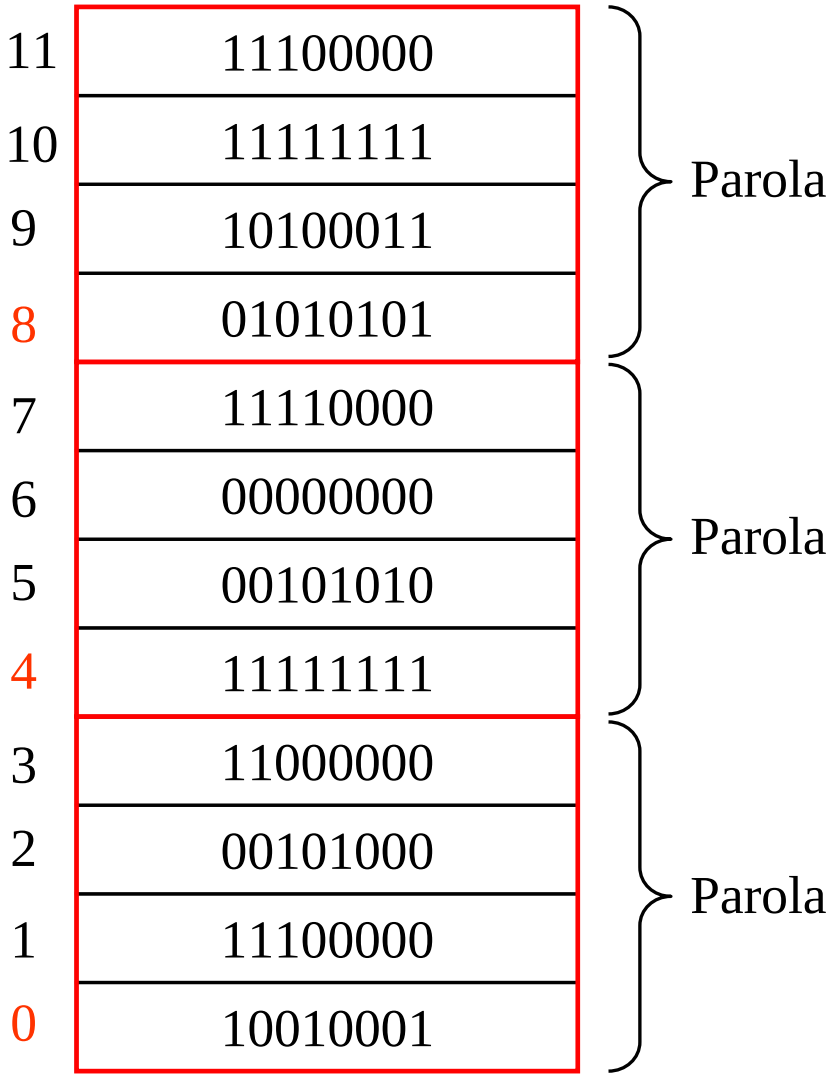
```
lw      rt, const(rs)          # rt = M[rs+const]
```

- rs e rt sono due registri
- const è una costante specificata dall'istruzione

NOTA SUGLI OPERANDI IN MEMORIA

- Il MIPS indirizza il singolo byte (8 bit)
- Gli indirizzi delle parole sono multipli di 4 (in una parola ci sono 4 byte)
- Gli indirizzi di due parole consecutive differiscono di 4 unità
- Nel MIPS le parole iniziano sempre ad indirizzi multipli di 4, cioè 0, 4, 8, 12, … (vincolo di allineamento)

Indirizzamento a byte e vincolo di allineamento



Esempio

- Le strutture dati complesse (p.es. array) dei linguaggi di alto livello sono allocate in memoria
- Esempio: tradurre g = h + A[8]

- Assumendo che il compilatore associ i registri $s1 e $2 a g e h e l'indirizzo di partenza del vettore A sia memorizzato in $s3 (***registro base***):
  - Occorre ***caricare dalla memoria in un registro*** ($t0) il valore in A[8]
  - L'indirizzo di questo elemento è dato dalla *somma dell'indirizzo di base* del vettore A (che è in $s3) *e del numero* (***offset***) che identifica l'elemento, moltiplicato per 4

- La compilazione dell'istruzione produrrà:

```
lw      $t0, 32($s3)      #ottava parola: 32-esimo byte
add     $s1, $s2, $t0
```

Istruzioni di trasferimento dati: store word (sw)

- Permette di trasferire il valore di un registro in una parola di memoria
- Sintassi:

```
sw      rt, const(rs)          # M[rs+const] = rt
```

- rs e rt sono due registri
- const è una costante specificata dall'istruzione

ESEMPIO

A[12] = h + A[8]

dove h sia associata a $s2 e il registro base del vettore A sia $s3

```
lw      $t0, 32($s3)  # carica in $t0 il valore A[8]
add     $t0, $s2, $t0 # somma h + A[8]
sw      $t0, 48($s3)  # memorizza la somma in A[12]
```

Esempio: indice variabile

- Si supponga che l'accesso a un vettore avvenga con indice variabile

- Es. : g = h + A[i]
- L'indirizzo base di A sia $s3 e alle variabili g, h, i siano associati i registri $s1, $s2, $s4

```
mul   $t1, $s4, 4    # i*4 in $t1
add   $t1, $t1, $s3  # indirizzo di A[i] in $t1
lw    $t0, 0($t1)    # carica A[i] in $t0
add   $s1, $s2, $t0  # g = h + A[i]
```

Moltiplicazione nel MIPS: un esempio di pseudoistruzione

- L'istruzione da usare sarebbe mult che fa il prodotto del contenuto di due registri di 32 bit e pone il risultato a 64 bit nei registri di 32 bit hi e lo
  (registri speciali utilizzati da una unità specifica per la moltiplicazione e divisione)
- Poi, per caricare il prodotto intero su 32 bit in un registro general purpose, il programmatore può usare mflo (*move from lo*)
- Esempio:

```
mult $s2, $s3     # hi, lo = $s2*$s3
mflo $s1          # s1 = lo
```

- In pratica useremo la *pseudoistruzione* mul

```
mul rdest, rsrc1, src2
```

  che mette il prodotto fra rsrc1 e src2 nel registro rdest

Caratteristiche dell'architettura fino a questo punto

LOAD-STORE

- Registri general-purpose

  (le istruzioni lavorano indifferentemente su tutti i 32 registri)

  - fa eccezione *$zero*, che si può usare ma rimane comunque al valore nullo

- Istruzioni aritmetiche: lavorano con operandi
  - registro
  - immediato (per evitare accessi in memoria)
  - consentono di rappresentare due sorgenti e una destinazione

    (usare i registri è più efficiente rispetto alla memoria; spilling dei compilatori)
- Istruzioni load-store: le uniche che accedono alla memoria

RISC: evitare la proliferazione delle istruzioni (sono possibili compromessi). P.es.

- subi non presente (si può usare addi)
- NOT non presente (si può usare NOR + registro $zero)
- NORI non presente (pozo utilizzato, si può comunque usare or e "not con nor")

E LA STESSA COSA AVVIENE CON LE ISTRUZIONI DI SALTO…

Istruzioni di controllo: salti condizionati (beq, bne)

beq rs, rt, L1

Vai all'istruzione etichettata con L1 se il valore contenuto nel registro rs è uguale al valore contenuto nel registro rt (beq = branch if equal)

bne rs, rt, L1

Vai all'istruzione etichettata con L1 se il valore contenuto nel registro rs è diverso dal valore contenuto nel registro rt (bne = branch if not equal)

NB: sono istruzioni semplici da implementare (per confrontare due registri si può fare lo XOR bit a bit, che dà 1 se uguali, poi considerarne l'AND)

- ciò risulterà importante quando si parlerà di pipeline…

Istruzioni di controllo: salti incondizionati (j, jr e jal)

j L1

Vai incondizionatamente all'istruzione etichettata con L1
(j = jump)

jr rs

Vai incondizionatamente all'istruzione di indirizzo contenuto nel registro rs
(jr = jump register)

jal L1

Vai incondizionatamente all'istruzione etichettata con L1 e salva l'indirizzo di ritorno in $ra - cfr. procedure
(jal = jump and link)

NB: *beq*, *bne*, *j* consentono di tradurre gli if e i cicli

Esempio: if-else

- Frammento di codice C:

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

  Supponendo che f, g, h, i, j corrispondano ai registri $s0, $s1, $s2, $s3, $s4, la traduzione è la seguente

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

Versione alternativa equivalente

- Frammento di codice C:

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

  Supponendo che f, g, h, i, j corrispondano ai registri $s0, $s1, $s2, $s3, $s4, la traduzione è la seguente

```
        beq   $s3, $s4, If
        sub   $s0, $s1, $s2
        j     Esci
If:     add   $s0, $s1, $s2
Esci:
```

Esempio: ciclo do-while

- Frammento di codice C:

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

Supponiamo che le variabili g, h, i, j siano associate ai registri $s1, $s2, $s3, $s4 e l'indirizzo di partenza del vettore A sia in $s5.

La traduzione è la seguente:

```
Ciclo:  sll $t1, $s3, 2        # $t1 = i * 4
        add $t1, $t1, $s5      # $t1 = indirizzo di A[i]
        lw $t0, 0($t1)         # $t0 = A[i]
        add $s1, $s1, $t0      # g = g + A[i]
        add $s3, $s3, $s4      # i = i + j
        bne $s3, $s2, Ciclo    # salta a Ciclo se i ≠ j
```

Esempio: ciclo while

- Frammento di codice C:

  while (salva[i] = = k) i = i + j;

  Supponendo che i, j, k corrispondano ai registri $s3, $s4, $s5 e l'indirizzo base di salva sia in $s6.

  La traduzione è la seguente:

```
Ciclo:  sll   $t1, $s3, 2     # $t1 = i * 4
        add   $t1, $t1, $s6   # $t1 = indirizzo di salva[i]
        lw    $t0, 0($t1)     # $t0 = salva[i]
        bne   $t0, $s5, Esci  # vai a Esci se salva[i] ≠ k
        add   $s3, $s3, $s4   # i = i + j
        j     Ciclo           # vai a Ciclo
Esci:
```

  Si usano 1 salto condizionato e 1 salto incondizionato: ad ogni iterazione del ciclo effettivamente eseguita vengono eseguite 6 istruzioni

- Codice ottimizzato (con un solo salto condizionato): eseguo il corpo del ciclo in ogni caso, controllando la condizione di permanenza anziché di uscita ed eliminando l'effetto collaterale dopo l'uscita dal ciclo

```
Ciclo:  sll   $t1, $s3, 2      # $t1 = i * 4
        add   $t1, $t1, $s6    # $t1 = indirizzo di salva[i]
        lw    $t0, 0($t1)      # $t0 = salva[i]
        add   $s3, $s3, $s4    # i = i + j
        beq   $t0, $s5, Ciclo  # vai a Ciclo se salva[i]==k
        sub   $s3, $s3, $s4    # i = i – j
```

⇨ Un solo salto (condizionato): 5 istruzioni nel corpo del ciclo!

Istruzioni di controllo: l'istruzione set on less than (slt)

- Oltre a testare l'uguaglianza o diseguaglianza tra due variabili, talvolta è necessario confrontarle e *beq*, *bne* non bastano!
- E' prevista un'altra istruzione per confrontare due variabili: slt

  slt rd, rs, rt

  poni a 1 il valore del registro rd se il valore del registro rs è minore del valore del registro rt, altrimenti rd è posto a 0
  (slt = set on less than)
- Esiste anche la versione immediate: slti rt, rs, immediate
- Inoltre, il registro $zero che contiene sempre il valore 0, è un registro di sola lettura che può essere usato nelle condizioni relazionali

- Esempio:

```
slt     $t0, $s0, $s1          # $t0 diventa 1 se $s0 < $s1
bne     $t0, $zero, Minore     # vai a Minore se $t0 ≠ 0
```

Esempio: ciclo for

- Frammento di programma C:

```
for (i=0; i<k; i++)
    f = f + A[i];
```

siano i, k, f corrispondenti a $s1, $s2, $s3, supponendo che l'indirizzo base di A in $s4. La traduzione è:

```
        add   $s1, $zero, $zero   #i=0
Ciclo:  slt   $t2, $s1, $s2       # poni $t2 a 1 se i < k
        beq   $t2, $zero, Esci    # se $t2  = 0 vai a Esci
        sll   $t1, $s1, 2         # $t1 = i * 4
        add   $t1, $t1, $s4       # $t1 = indirizzo di A[i]
        lw    $t0, 0($t1)         # $t0 = A[i]
        add   $s3, $s3, $t0       # f = f + A[i]
        addi  $s1, $s1, 1         # i = i + 1
        j     Ciclo
Esci:
```

Esempio: case-switch

- Frammento di programma 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;
}
```

- Si potrebbe trasformare in una catena di *if-then-else* e quindi tradurlo con salti condizionati
- A volte si può usare in modo più efficace una tabella di indirizzi in cui si trovano le diverse sequenze di istruzioni (*tabella degli indirizzi di salto o jump address table*):
  - la tabella è un **vettore di parole** che contiene gli **indirizzi corrispondenti alle etichette** presenti nel codice
  - il programma usa un **indice** per saltare alla sequenza di codice opportuna

- Premessa per la traduzione del programma:

```
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;
}
```

  - Si assuma che f, g, h, i, j, k corrispondano ai registri da $s0 a $s5
  - La variabile k è usata per indicizzare la tabella degli indirizzi di salto
  - Sia in $t4 l'indirizzo di partenza della tabella dei salti
  - Si supponga che la tabella contenga in sequenza gli **indirizzi** corrispondenti alle **etichette** L0, L1, L2, L3

Traduzione

```
        slt     $t3, $s5, $zero     # controlla se k < 0
        bne     $t3, $zero, Esci    # se k < 0 (cioè $t3 = 1) salta a Esci
        slti    $t3, $s5, 4         # controlla se k < 4
        beq     $t3, $zero, Esci    # se k >= 4 (cioè $t3 = 0) salta a Esci
        sll     $t1, $s5, 2         # $t1 = k * 4 (k è l'indice di una tabella)
        add     $t1, $t1, $t4       # $t1 = indirizzo di TabelladeiSalti[k]
        lw      $t0, 0($t1)         # $t0 = TabelladeiSalti[k], cioè $t0
                                    # conterrà un indirizzo corrispondente
                                    # a un'etichetta
        jr      $t0                 # salto all'indirizzo contenuto in $t0
L0:     add     $s0, $s3, $s4       # k = 0, quindi f = i + j
        j       Esci                # fine del case, salta a Esci
L1:     add     $s0, $s1, $s2       # k = 1, quindi f = g + h
        j       Esci                # fine del case, salta a Esci
L2:     sub     $s0, $s1, $s2       # k = 2, quindi f = g -  h
        j       Esci                # fine del case, salta a Esci
L3:     sub     $s0, $s3, $s4       # k = 3, quindi f = i – j
Esci:
```

Caratteristiche dell'architettura fino a questo punto

- Abbiamo già visto la filosofia RISC + load-store nelle istruzioni aritmetico-logiche e di trasferimento
- Per le istruzioni di controllo, abbiamo visto che:
  - si sono mantenute al minimo le istruzioni di salto condizionato (*beq* e *bne*)
  - per effettuare salti basati su confronto, è necessario decomporre il salto in più istruzioni, utilizzando *slt*
    > Questo non "pesa" nelle architetture che utilizzano il parallelismo
    > Permette però una semplificazione che porta in generale a maggiore efficienza ad es. ad una diminuzione di T_{clock}

Realizzazione di procedure mediante assembler MIPS

- Distinguiamo due “attori”: chiamante vs. chiamato (la procedura)
- Per eseguire una procedura, devono essere effettuati i seguenti passi:

1. Chiamante:
   mettere i parametri della procedura in un luogo accessibile alla procedura chiamata
2. Chiamante:
   trasferire il controllo alla procedura

---

3. Chiamato:
   acquisire le risorse necessarie alla memorizzazione dei dati
4. Chiamato:
   Eseguire il compito richiesto
5. Chiamato:
   Mettere il risultato in un luogo accessibile al programma chiamante
6. Chiamato:
   Restituire il controllo al punto di origine

Consideriamo i punti 2 e 6…

Chiamata di una procedura

•Istruzione jal (*jump-and-link*): è l'istruzione apposita per le procedure, che salta a un indirizzo e contemporaneamente salva l'indirizzo dell'istruzione successiva nel registro $ra

```
jal        IndirizzoProcedura
```

•In pratica jal salva il valore di PC+4 nel registro $ra, che contiene l'indirizzo della prossima istruzione: questo crea un "collegamento" all'indirizzo di ritorno dalla procedura

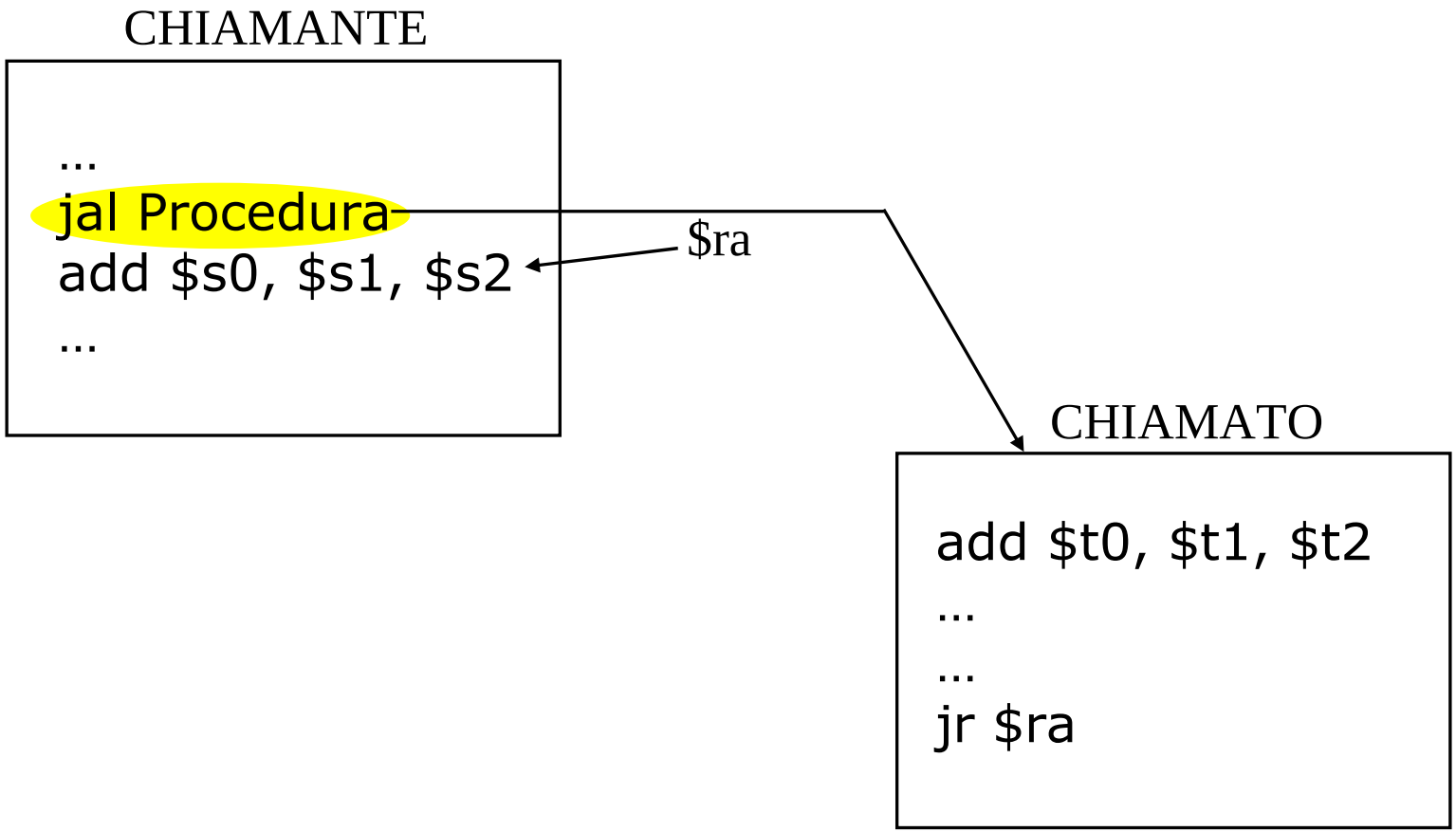
•L'indirizzo di ritorno è necessario perché la stessa procedura può essere richiamata in più parti del programma

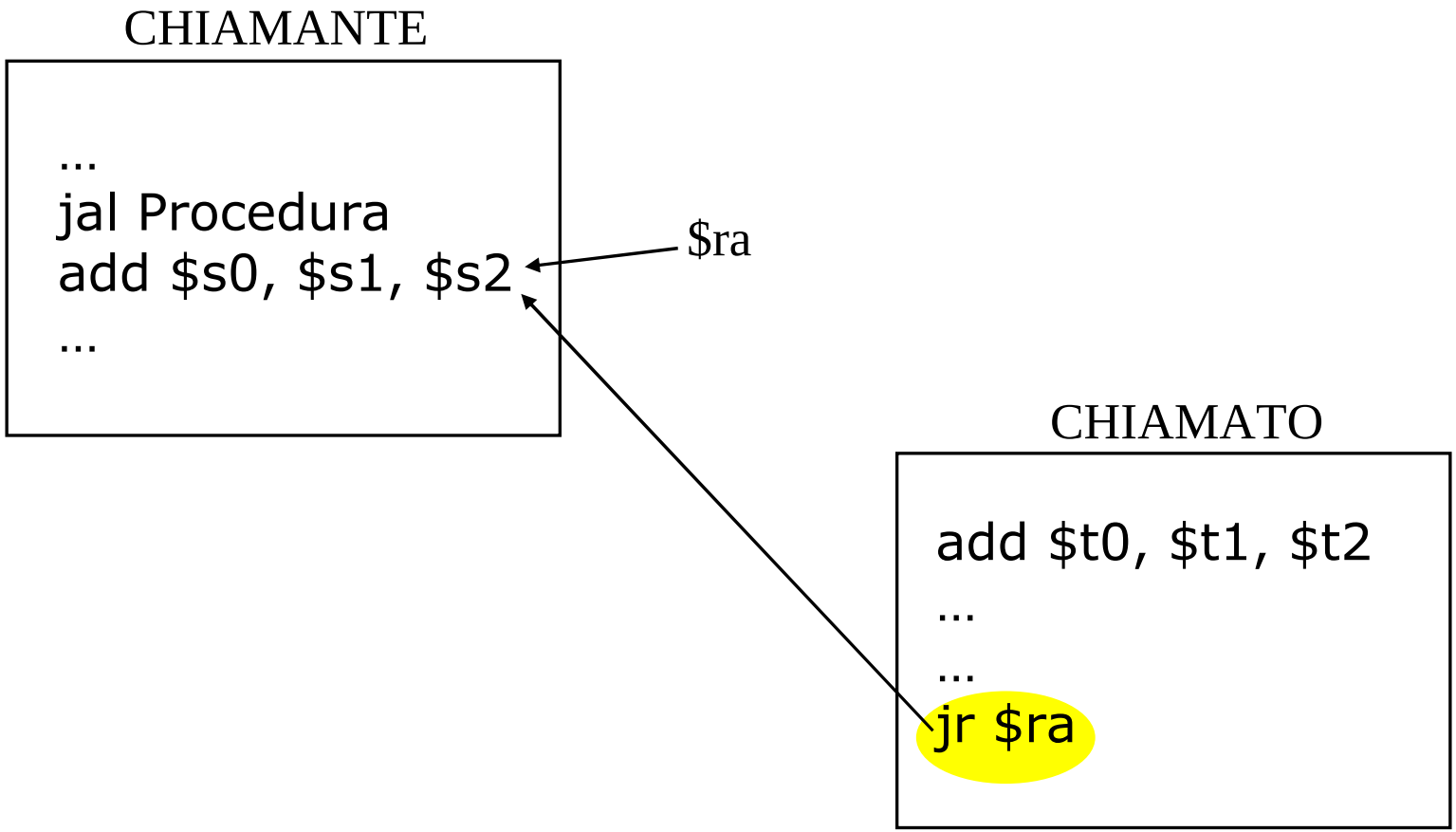
Ritorno da una procedura chiamata

- L'istruzione per effettuare il salto di ritorno è

```
jr         $ra
```

In questo modo il chiamato restituisce il controllo al chiamante


CHIAMANTE
...
jal Procedura
add $s0, $s1, $s2
...
$ra
CHIAMATO
add $t0, $t1, $t2
...
...
jr $ra


CHIAMANTE
...
jal Procedura
add $s0, $s1, $s2
...
$ra
CHIAMATO
add $t0, $t1, $t2
...
...
jr $ra

Le convenzioni

- Le procedure dei linguaggi di alto livello possono essere compilate separatamente
- Programmatori assembler possono implementare/chiamare procedure, realizzate da compilatori o da altri programmatori

⇨ Necessarie delle convenzioni
(*register use* o *procedure call conventions*)

Alcune convenzioni (non imposte dall'hardware ad eccezione di $ra!):

- $a0 - $a3: 4 registri argomento per il passaggio dei parametri
- $v0-$v1: 2 registri valore per la restituzione dei valori
- $ra: registro di ritorno per tornare al punto di origine

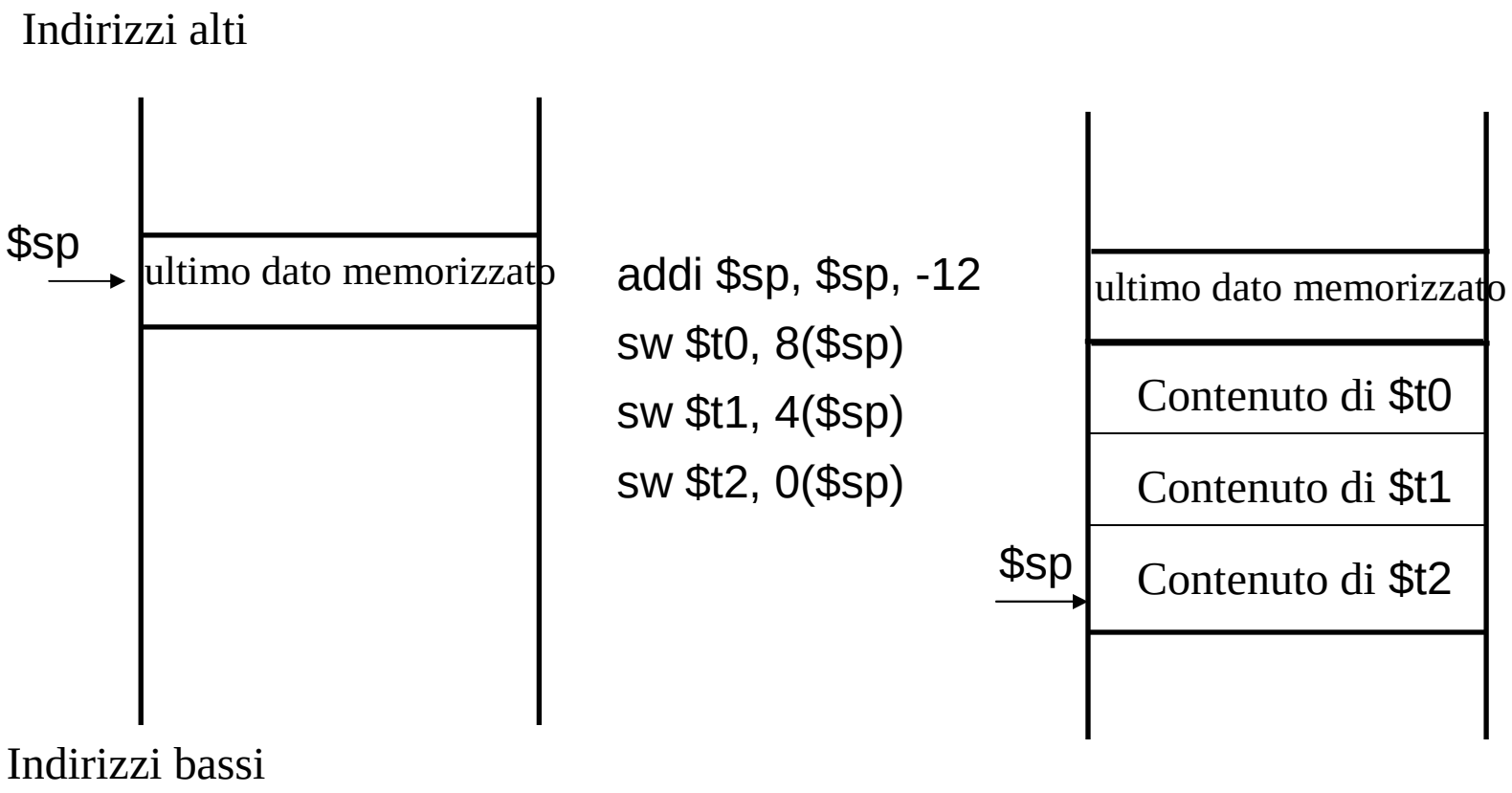
⇨
- Il ***programma chiamante*** mette i valori dei **parametri** da passare alla procedura nei **registri $a0-$a3** e utilizza l'istruzione jal X per saltare alla procedura X (programma chiamato)
- Il ***programma chiamato*** esegue le operazioni richieste, memorizza i **risultati nei registri $v0-$v1** e restituisce il controllo al chiamante con l'istruzione jr $ra

L’uso dello stack

- Spesso, una procedura necessita di salvare i valori di registri alcuni registri (ad esempio, perché chiama un’altra procedura che li cambia o perché non deve alterarli per il programma chiamante)
- Lo **stack** (pila) è una struttura del tipo *last-in first-out* dove è possibile memorizzare il valore di un registro (PUSH) e recuperare l’ultimo valore inserito (POP)
- Si usa un puntatore allo stack (**stack pointer**) che contiene l’indirizzo del dato introdotto più recentemente nello stack.
- Convenzioni sullo stack:
  - lo stack *cresce* a partire da indirizzi di memoria alti verso indirizzi di memoria bassi: quando vengono inseriti dei dati nello stack il valore dello stack pointer diminuisce, viceversa, quando sono estratti dati dallo stack, il valore dello stack pointer aumenta
  - il **registro** allocato dal MIPS come stack pointer è **$sp**

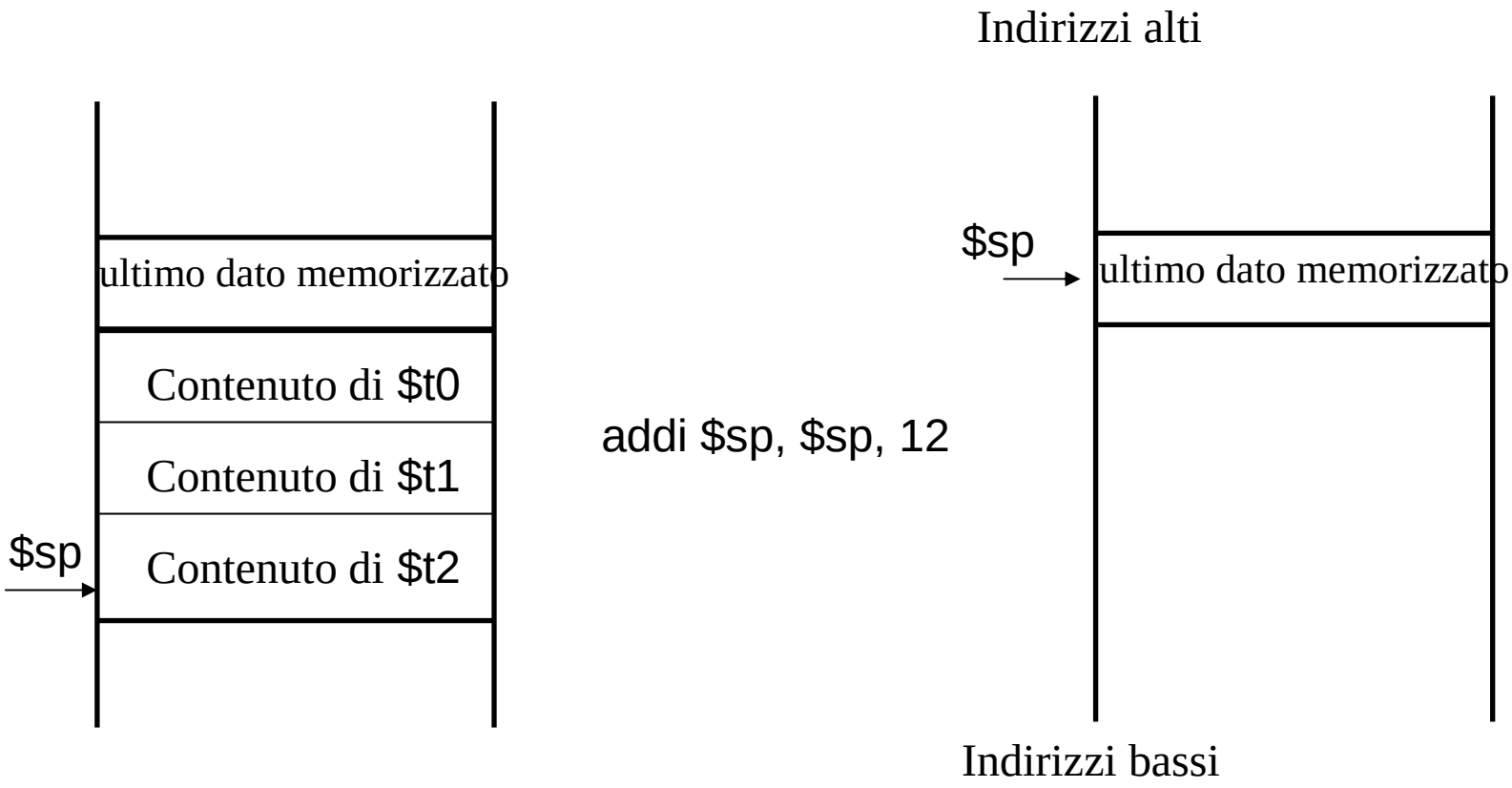
Operazione di push

Vogliamo memorizzare nello stack i valori di $t0, $t1, $t2



I valori possono poi essere caricati nei registri mediante la lw

Svuotamento dello stack



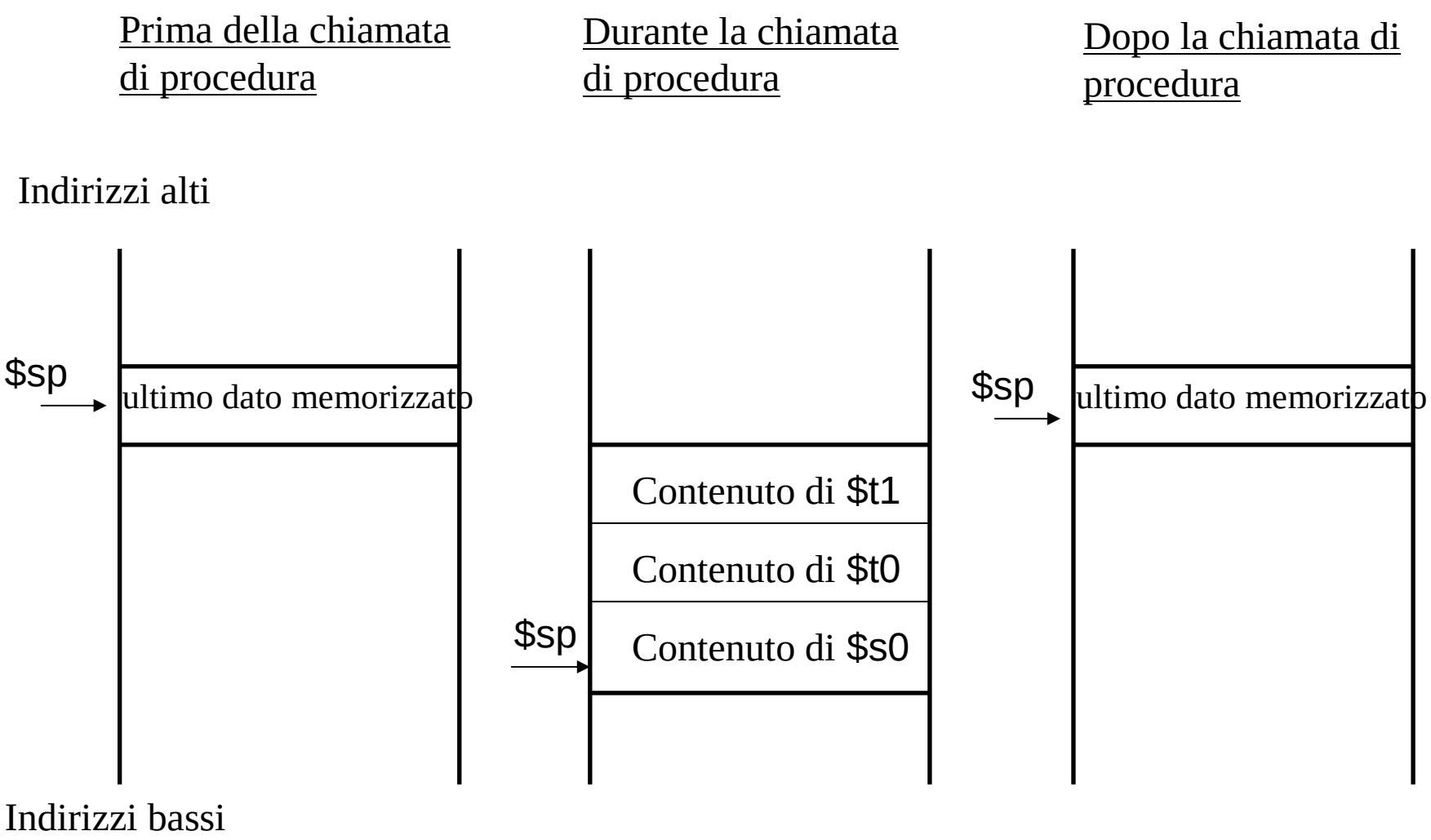
Esempio: procedura con uso dello stack

Procedura C

```
int proc(int g, int h, int i, int j)
{
        int f;
        f = (g + h) – (i + j)
        return f;
}
```

dove i parametri g, h, i, j corrispondono a $a0, $a1, $a2, $a3, e f corrisponde a $s0

```
proc:                           # etichetta della procedura
        addi    $sp, $sp, -12   # aggiornamento dello stack per fare
                                # spazio a 3 elementi
        sw      $t1, 8($sp)     # salvataggio del registro $t1
        sw      $t0, 4($sp)     # salvataggio del registro $t0
        sw      $s0, 0($sp)     # salvataggio del registro $s0
        add     $t0, $a0, $a1   # $t0 = g + h
        add     $t1, $a2, $a3   # $t1 = i + j
        sub     $s0, $t0, $t1   # f = $t0 - $t1
        add     $v0, $s0, $zero # restituzione di f
        lw      $s0, 0($sp)     # ripristino del registro $s0
        lw      $t0, 4($sp)     # ripristino del registro $t0
        lw      $t1, 8($sp)     # ripristino del registro $t1
        addi    $sp, $sp, 12    # aggiornamento dello stack per
                                # eliminare 3 elementi
        jr      $ra             # ritorno al programma chiamante
```


Prima della chiamata di procedura
Durante la chiamata di procedura
Dopo la chiamata di procedura
Indirizzi alti
$sp
ultimo dato memorizzato
Contenuto di $t1
Contenuto di $t0
$sp
Contenuto di $s0
$sp
ultimo dato memorizzato
Indirizzi bassi

Il salvataggio dei registri

- Nell'esempio precedente la procedura chiamata ha salvato nello stack tutti i registri da essa usati, in modo da non alterarne nessuno.
  In generale però si hanno due strategie:

  Salvataggio da parte del chiamato

  - La procedura ha la responsabilità di non alterare nessun registro
  - Il chiamante quindi si aspetta che nessun registro sia modificato
  - Inconveniente:
    i registri salvati dalla procedura potrebbero non essere usati dal chiamante (vengono quindi salvati e ripristinati inutilmente)

  Salvataggio da parte del chiamante

  - La procedura può alterare qualunque registro (tutti quelli che usa)
  - E' compito del chiamante salvare i registri che desidera non siano modificati nello stack
  - Inconveniente:
    se la procedura non usa i registri che il chiamante utilizza (ad esempio, usa un solo registro) il chiamante salva inutilmente tutti i registri che usa!

  ⇒ Si usa un approccio ibrido mediante specifiche convenzioni

Convenzione su registri preservati e non preservati

Per convenzione, si usano 2 classi di registri:

$t0-$t9: registri temporanei che non sono preservati
in caso di chiamata di procedura

$s0-$s7: registri che devono essere preservati
(se utilizzati devono essere salvati e ripristinati
dal programma chiamato)

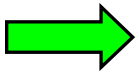
$\Rightarrow$ In questo modo si riduce la necessità di salvare registri in memoria

$\Rightarrow$ Nell'esempio precedente si possono eliminare
2 istruzioni di store e 2 di load (quelle che si riferiscono a $t0 e $t1)

NB: per convenzione $a0-$a3 non sono preservati

Procedure annidate

- Una procedura A può chiamare un'altra procedura B: possono esistere conflitti
  - es. jal B altera $ra, che serve ad A per ritornare al programma chiamante
- Soluzione: usare le convenzioni, aggiungendo la convenzione che
  $ra è un registro preservato!



- Il programma *chiamante* memorizza nello stack
  i registri argomento ($a0 - $a3) o i registri temporanei ($t0 - $t9)
  di cui ha ancora bisogno dopo la chiamata
- Il programma *chiamato* salva nello stack il registro di ritorno $ra
  e gli altri registri ($s0 - $s7) che utilizza

NB: notare che le convenzioni garantiscono la correttezza, perché sono applicate "ricorsivamente". Infatti, sono a prova di ricorsione…

Esempio: procedura ricorsiva (per il calcolo del fattoriale)

```
int  fatt(int n) {
        if (n < 1) return (1);
           else return (n *  fatt (n-1));
        }
```

il parametro n corrisponde al registro argomento $a0

Procedura in assembler per il calcolo del fattoriale

```
fatt:
        addi    $sp, $sp, -8        # aggiornamento dello stack per fare
                                    # spazio a 2 elementi
        sw      $ra, 4($sp)         # salvataggio del registro di ritorno (come
                                    # programma chiamato)
        sw      $a0, 0($sp)         # salvataggio del parametro n (come
                                    # programma chiamante)
        slti    $t0, $a0, 1         # test per n < 1
        beq     $t0, $zero, L1      # se n >= 1 salta a L1
        addi    $v0, $zero, 1       # restituisce 1 (per n<1)
        addi    $sp, $sp, 8         # aggiornamento dello stack
        jr      $ra                 # ritorno all'istruzione successiva a jal
L1:     addi    $a0, $a0, -1        # n > 1: l'argomento diventa n-1
        jal     fatt                # chiamata a fatt con (n-1)
        lw      $a0, 0($sp)         # ritorno da jal: ripristino di n
        lw      $ra, 4($sp)         # ripristino del registro di ritorno
        add     $sp, $sp, 8         # aggiornamento dello stack
        mul     $v0, $a0, $v0       # restituzione di n * fatt (n-1)
        jr      $ra                 # ritorno al programma chiamante
```

$a0 e $ra non sono cambiati →

chiamata ricorsiva →

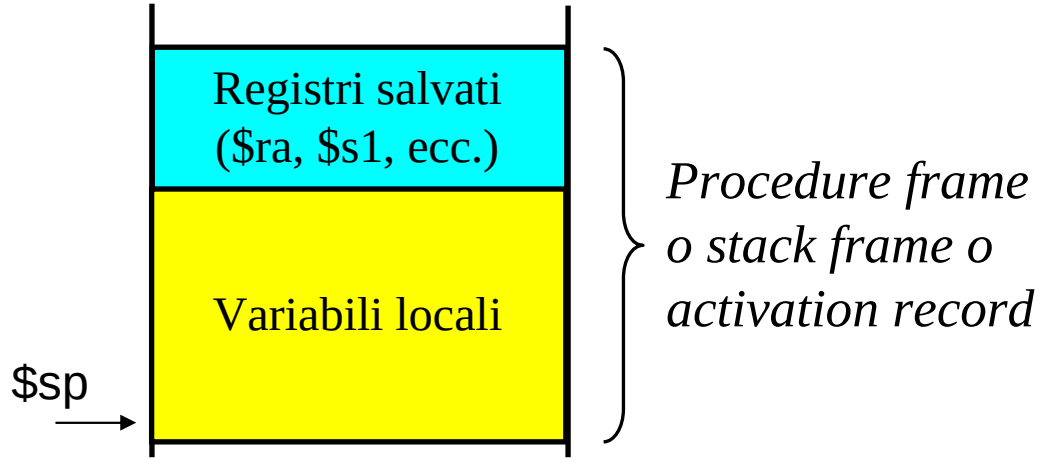
Prodotto di fatt(n-1) ritornato in $v0 per il vecchio parametro $a0 →

Modifica di $a0

Creazione di variabili locali nelle procedure

• Le procedure possono definire variabili locali (visibili solo internamente): create all'inizio dell'esecuzione della procedura, distrutte al ritorno

➡ Si usa ancora lo stack



PROBLEMA

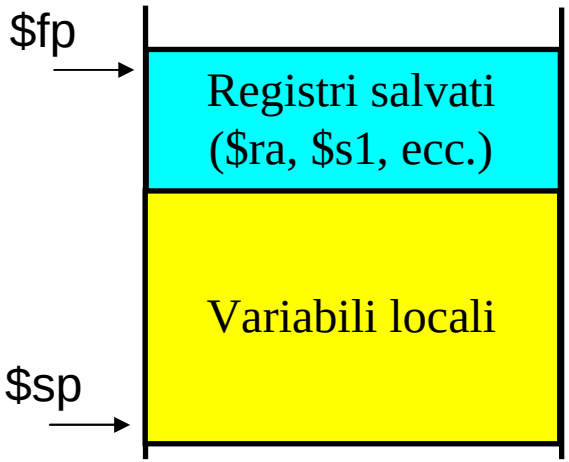
La dimensione dello stack può cambiare durante l'esecuzione della procedura

$\Rightarrow$ difficile riferire una variabile rispetto a $sp

(offset diversi nel corso del programma: scarsa leggibilità)

➡ Utilizzo un registro, per convenzione $fp, per riferire l'inizio dello stack

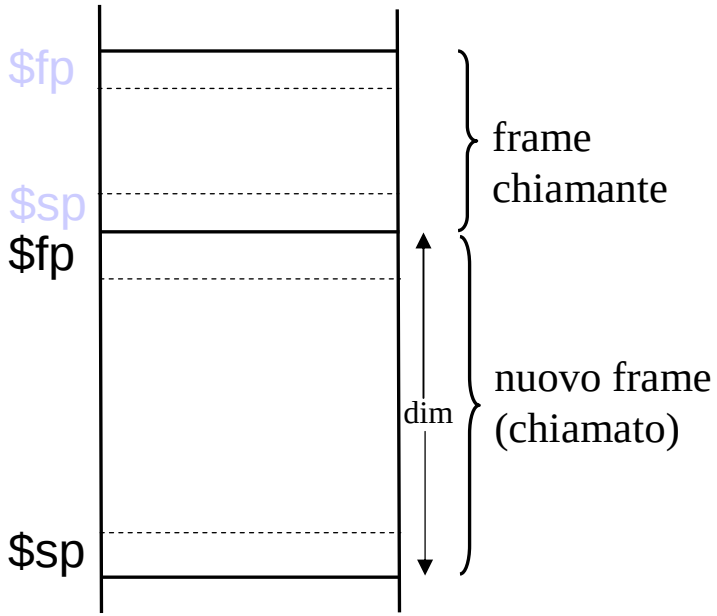
Il registro “frame pointer” $fp



Il registro $fp punta stabilmente all’inizio dello stack frame (punta al primo elemento)

⇒ $s1 sarà sempre in const($fp)

- All’inizio dell’esecuzione, la procedura:
  - crea spazio nello stack per valori da salvare:
    addi $sp, $sp, -dim
  - salva nello stack i valori da salvare (compreso $fp)
  - aggiorna $fp
    addi $fp, $sp, dim-4



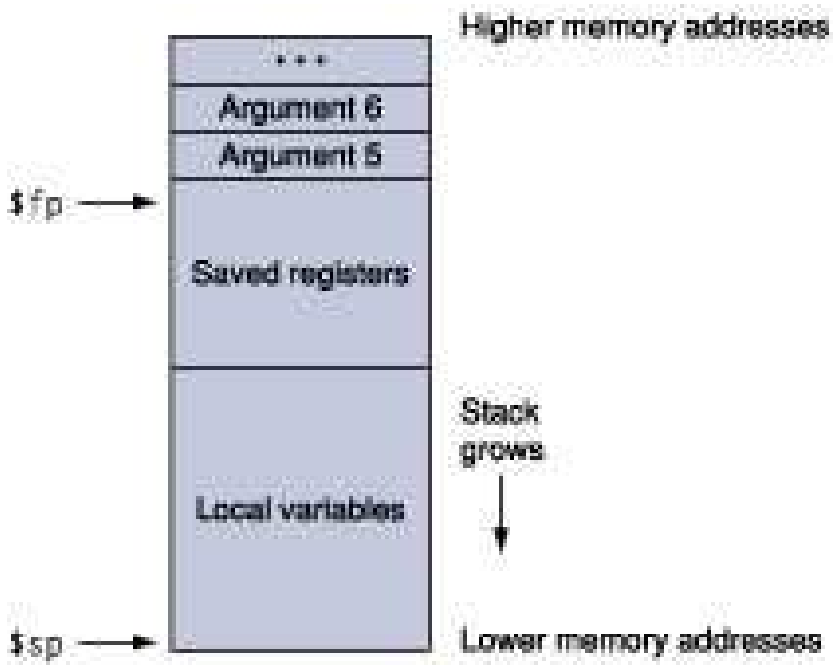
Procedura in assembler per il calcolo del fattoriale utilizzando $fp

Es. di uso di $fp

```
fatt:
        addi    $sp, $sp, -12     # aggiornamento dello stack per fare
                                  # spazio a 3 elementi ($fp compreso)
        sw      $ra, 4($sp)       # salvataggio del registro di ritorno (come
                                  # programma chiamato)
        sw      $fp, 0($sp)       # salvataggio del frame pointer (come
                                    programma chiamato)
        addi    $fp, $fp, 8       # $fp punta all'inizio dello stack
        sw      $a0, 0($fp)       # salvataggio del parametro n (come
                                  # programma chiamante)
        slti    $t0, $a0, 1       # test per n < 1
        beq     $t0, $zero, L1    # se n >= 1 salta a L1
        addi    $v0, $zero, 1     # restituisce 1 (per n<1)
        j       EXIT              # ritorno
L1:     addi    $a0, $a0, -1      # n > 1: l'argomento diventa n-1
        jal     fatt              # chiamata a fatt con (n-1)
        lw      $a0, 0($fp)       # ritorno da jal: ripristino di n
        mul     $v0, $a0, $v0     # restituzione di n * fatt (n-1)
EXIT:   lw      $ra, 4($sp)       # ripristino del registro di ritorno
        lw      $fp, 0($sp)       # ripristino del frame pointer
        addi    $sp, $sp, 12      # aggiornamento dello stack
        jr      $ra               # ritorno al programma chiamante
```

Passaggio di più di 4 parametri

- Il chiamante:
  - mette i parametri da passare alla procedura in cima allo stack
  - chiama la procedura con jal
- La procedura chiamata trova i parametri passati “sopra” $fp
  (in cima al frame del chiamante)



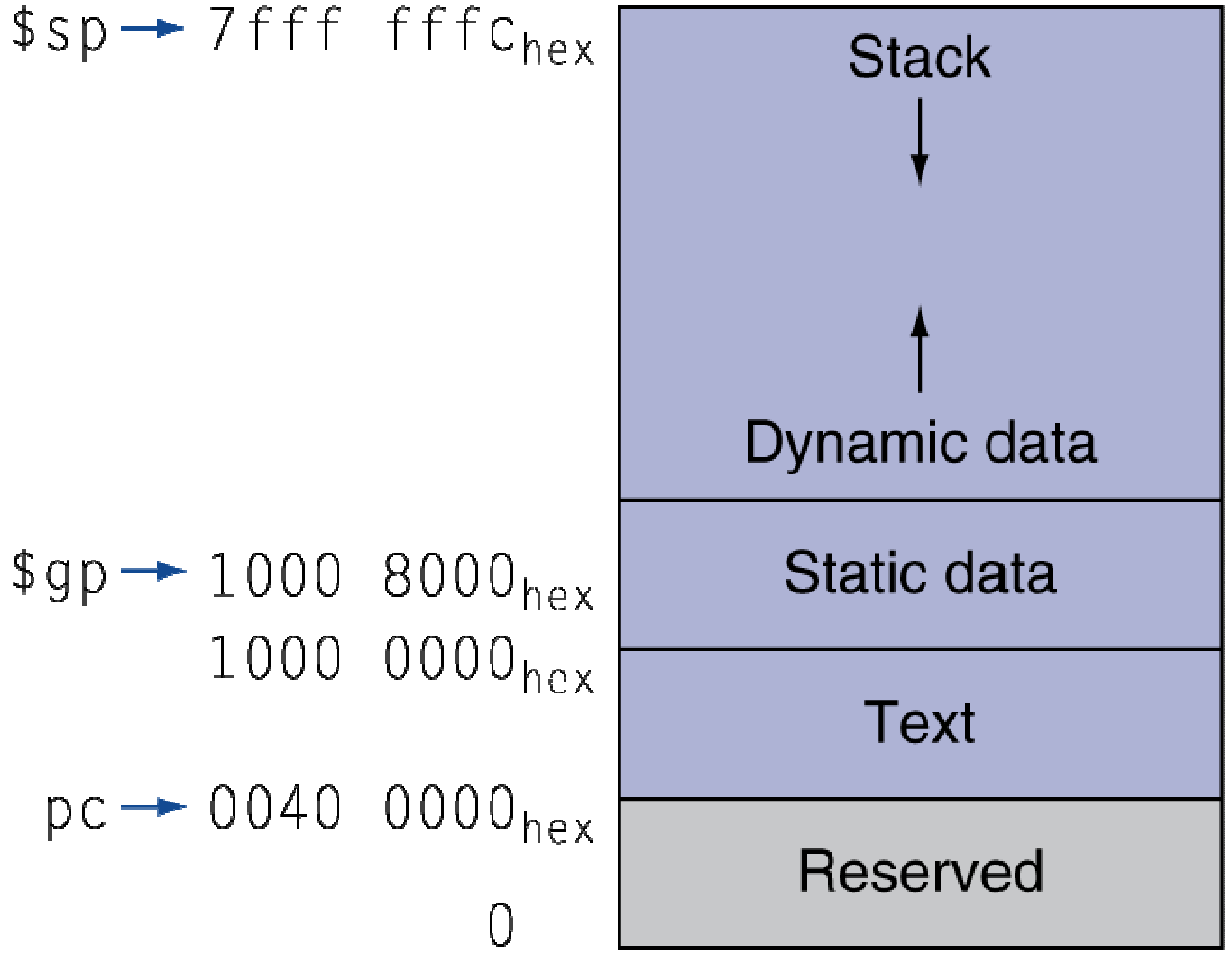
Riassunto: operazioni del chiamante e del chiamato

1. Il chiamante:
    1. Salva gli eventuali registri che non vuole siamo modificati dal chiamato ($a0 - $a3, $t0 - $t9)
    2. Pone gli argomenti in più rispetto ai registri $a0-$a4 in cima allo stack
    3. Chiama la procedura tramite l'istruzione jal
2. La procedura chiamata prima di eseguire il compito (appena chiamata):
    1. Alloca spazio per il nuovo frame (*addi $sp, $sp, -dim*)
    2. Salva i registri che modifica su cui ha responsabilità come chiamato ($s0-$s7, $fp, $ra)
    3. Modifica $fp in modo che punti alla base dello stack frame (*addi $sp, $sp, dim-4*)
3. La procedura chiamata dopo aver eseguito il compito:
    1. Pone il risultato in $v0
    2. Ripristina i registri salvati (come chiamato)
    3. Libera lo spazio sullo stack (*addi $sp, $sp, dim*)
    4. Ritorna al chimante con *jr $ra*

Note

- Così come non è necessario salvare sullo stack tutti i registri *$si*
  ma solo quelli usati, la procedura chiamata:
  - salverà *$ra* solo se ne chiama un'altra
  - salverà *$fp* solo se questo è usato
    (utile solo se la dimensione dello stack può variare durante l'esecuzione!)
    NB: alcuni compilatori non fanno uso di $fp, disponendo così di $s8
- Esistono precise convenzioni sull'uso dello stack, che ne complicano l'utilizzo
  rispetto a quanto visto; tuttavia sono convenzioni dipendenti strettamente
  dal processore e dal compilatore usati e perciò non le trattiamo

Convenzioni nell'uso della memoria



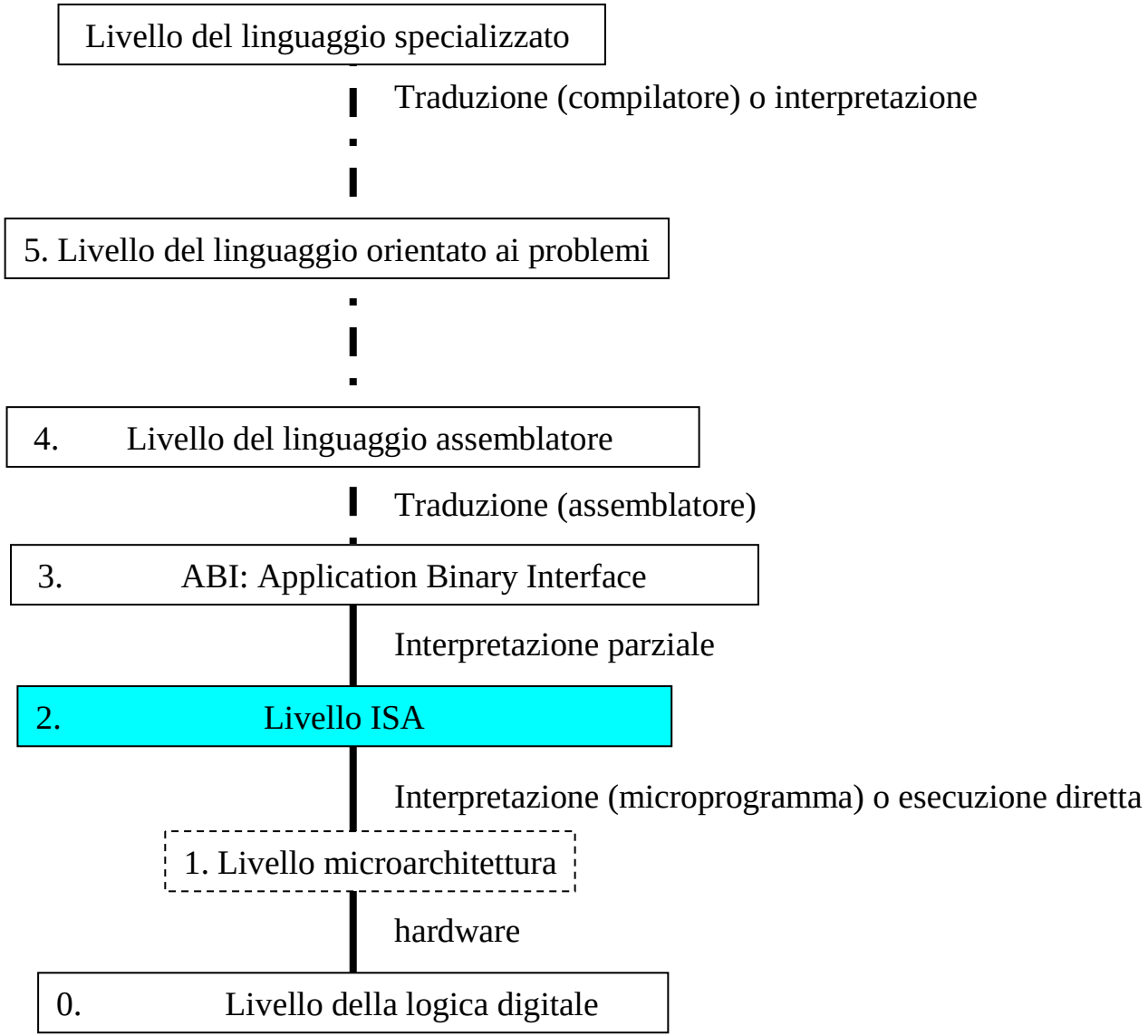
Spiegazione

- Il codice macchina MIPS è posto nel segmento ***text***, a partire dall'indirizzo $0040\ 0000_{16}$
- Il segmento static data è utilizzato dal compilatore per oggetti:
  - la cui lunghezza non varia durante l'esecuzione (è conosciuta dal compilatore)
  - che durano per tutta l'esecuzione del programma

  Il registro *$gp* (global pointer) è posto a $1000\ 8000_{16}$ e può essere utilizzato dall'istruzione lw (con un offset a 16 bit) per accedere al primo blocco di 64 K che conterrà le variabili statiche di uso più frequente
- Il segmento dynamic data (chiamato heap) contiene dati allocati dal programma durante l'esecuzione (es. in C tramite malloc) e quindi può espandersi (verso indirizzi più alti) o ritirarsi
- Lo stack parte dall'indirizzo $7FFF\ FFFC_{16}$ e si espande verso indirizzi più bassi

NB: in realtà ci si riferisce ad uno spazio virtuale

Linguaggio macchina: i formati delle istruzioni del MIPS



- Il MIPS adotta istruzioni a lunghezza costante: 32 bit
  (una parola di memoria, un registro)
- Vediamo come questi 32 bit sono stati suddivisi in diversi campi per definire il ***formato delle istruzioni***, rispettando il principio della regolarità
- Nel far questo, emergeranno i modi di indirizzamento del MIPS

Da quanto visto, le istruzioni esaminate si possono suddividere in 3 categorie:

- istruzioni che devono indicare 3 registri (add, sub, and, slt, ecc.)
- istruzioni che devono indicare due registri e una costante, in particolare:
  - lw e sw
  - istruzioni che riferiscono un operando immediato (addi, andi, ori, ecc.)
  - salti condizionati (due registri per il confronto + costante per il salto)
- istruzioni di salto incondizionato, che non riferiscono alcun registro ma (presumibilmente) indicano una costante "con molti bit"…

⇒ Le tre categorie danno luogo a tre (semplici) formati, con cui si riescono a rappresentare tutte le istruzioni!

Formato-R (register) [per istruzioni aritmetiche e logiche]

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

op: codice operativo

rs: primo operando sorgente (registro)

rt: secondo operando sorgente (registro)

rd: registro destinazione

shamt: shift amount (per operazioni di scalamento)

funct: seleziona una variante specifica dell'operazione base definita nel campo *op*

Per esempio: campo funct per le istruzioni add, sub, and, or, slt

op = 0
- ADD: 100 000
- SUB: 100 010
- AND: 100 100
- OR: 100 101
- SLT: 101 010

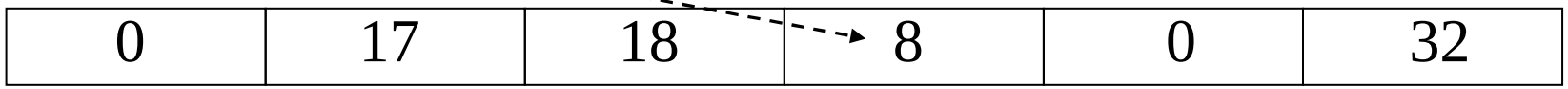
Indirizzamento tramite registro

L’operando è un registro: servono 5 bit (32 registri)

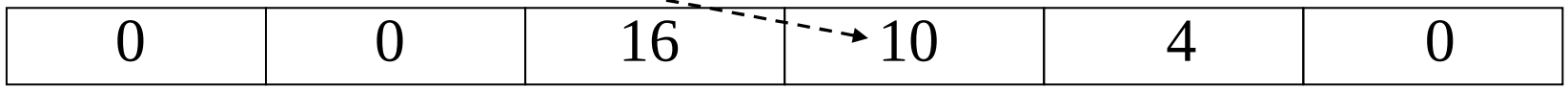
Nome	Codifica
$zero	0
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$gp	28
$sp	29
$fp	30
$ra	31

Esempi

add $t0, $s1, $s2



sll $t2, $s0, 4



NON USATO

Perché si utilizza un formato che “divide il codice operativo” in due campi?

- Le istruzioni che hanno lo stesso codice operativo sono molto simili:

  p.es. per add, sub, and, or, slt l’implementazione può prevedere

  - prelievo dei due operandi *rs* e *rt*
  - operazione ALU determinata dal campo funct
  - memorizzazione risultato in *rd*

  $\Rightarrow$ l’unità di controllo “fa la stessa cosa” se funct è in ingresso alla ALU

- In questo modo si riesce a rendere il Formato-R quanto più simile ai due successivi (vedi prossimi lucidi)

Consideriamo ora le istruzioni del secondo tipo.

P.es. le istruzioni lw e sw riferiscono:

- due registri (*rs* e *rt*)
- una costante da sommare a *rt* per calcolare l'indirizzo

$\Rightarrow$ Per avere costanti con un intervallo di valori maggiore di 32 (-15…+16), ovviamente insufficiente, è necessario introdurre un altro formato

Si farà in modo comunque di mantenerlo "il più simile possibile" al precedente

Formato-I (immediate) [per istruzioni di trasferimento, immediate, branch]

Consideriamo solo:

le istruzioni “immediate”, p.es. addi rt, rs, const

le istruzioni di trasferimento p.es. sw rt, const(rs)

6 bit 5 bit 5 bit 16 bit

rs: nel caso di istruzioni immediate: registro sorgente

nel caso di lw e sw: registro base al cui contenuto va sommato address

rt: nel caso di istruzioni immediate: registro destinazione

nel caso di lw e sw: primo registro che compare nell’istruzione

(registro destinazione per lw e registro sorgente per sw)

const/address: costante da sommare a 16 bit (-2^{15} … $+2^{15}-1$)

Indirizzamento immediato

- L’operando è una costante
- Usato dalle versioni “immediate” delle istruzioni aritmetiche (ad es. addi) che usano due operandi registro (sorgente e destinazione) e un operando immediato
- La costante è contenuta nel “immediate” nel formato-I
- Essendo un intero a 16 bit, ha ancora valori da -2^{15} a $+(2^{15}-1)$
- Nelle versioni “unsigned” delle istruzioni esso viene interpretato come “unsigned” (da 0 a $2^{16}-1$)

Indirizzamento tramite base o spiazzamento

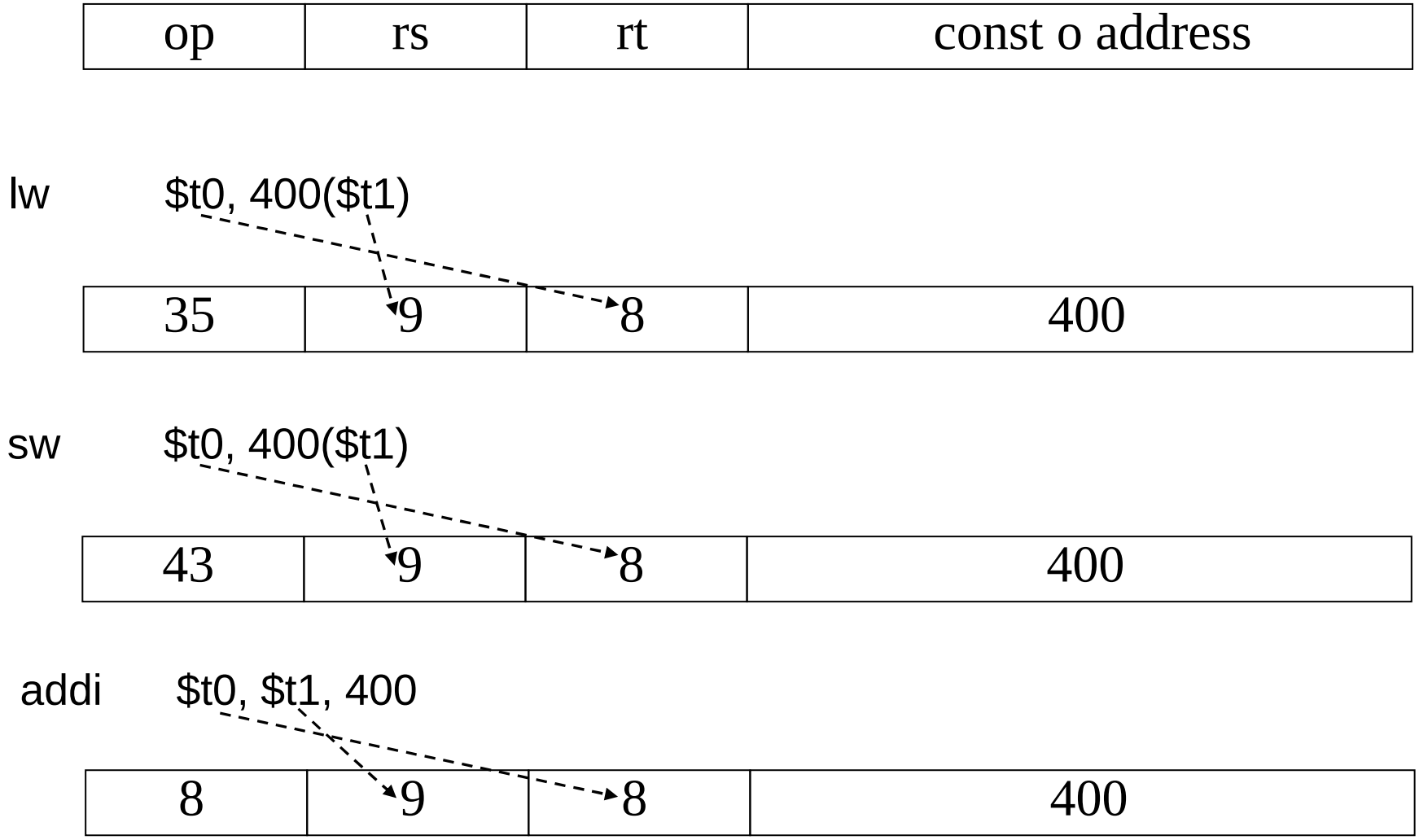
L'operando è in memoria. P.es.

```
lw          $t1, 32($t0)
```

- $t0 contiene un indirizzo base
- 32 è uno spiazzamento da sommare al contenuto di $t0

NB: lo spiazzamento è espresso come intero con segno a 16 bit,
quindi può valere da -2^{15} a $+(2^{15}-1)$

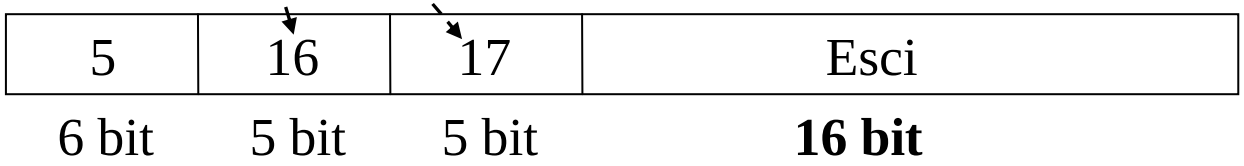
Esempi


op
rs
rt
const o address
lw $t0, 400($t1)
35
9
8
400
sw $t0, 400($t1)
43
9
8
400
addi $t0, $t1, 400
8
9
8
400

Codifica delle istruzioni di salto condizionato

- Anche per le istruzioni beq e bne si può usare il Formato-I

```
bne     $s0, $s1, Esci    # vai Esci se $s0 è diverso da $s1
```



(dove 16 e 17 sono i numeri dei registri $s0 e $s1)

PROBLEMA: come viene codificata l'etichetta?

Abbiamo a disposizione 16 bit…

Indirizzamento relativo al Program-Counter (PC-relative)

- Se si codificasse direttamente l'indirizzo con 16 bit nessun programma potrebbe avere dimensioni maggiori di 2^{16}

  $\Rightarrow$ idea: specificare un registro da sommare all'indirizzo di salto, cioè:

  Program counter = *Registro* + Indirizzo di salto

- Ma quale *Registro* di partenza usare?

  - Considerando che normalmente i salti condizionati vengono usati nei cicli e nei costrutti *if*, i salti sono in genere ad istruzioni vicine

    $\Rightarrow$ usare PC, che ha l'indirizzo dell'istruzione [successiva a quella] corrente

- Per sfruttare i 16 bit a disposizione, si può considerare l'indirizzo in istruzioni

  $\Rightarrow$ a partire da PC si può saltare fino a una *distanza di* $\pm 2^{15}-1$ *istruzioni*

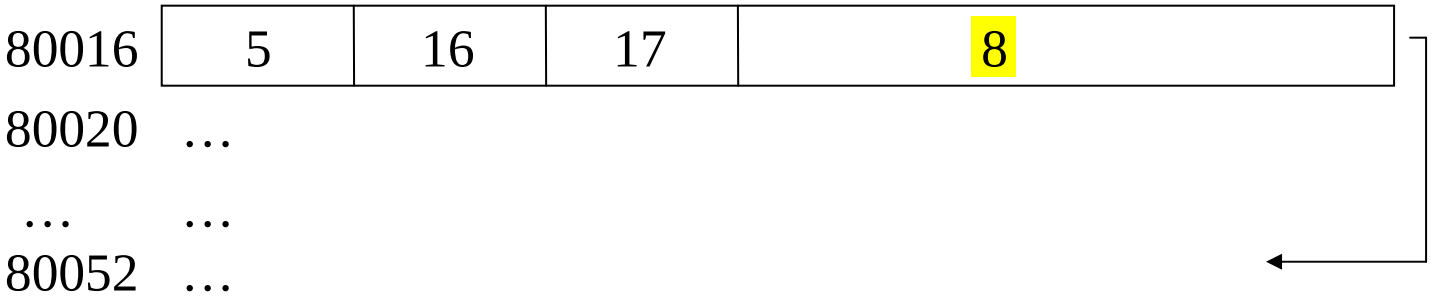
  (basta moltiplicare per 4 la costante, ovvero concatenarla con '00') rispetto all'istruzione [successiva a quella] corrente e questo è sufficiente

- L'indirizzamento PC-relative è usato nelle istruzioni di *salto condizionato*, mentre le istruzioni *jump* e *jump-and-link* (chiamata di procedure) utilizzano un altro modo di indirizzamento

Esempio di indirizzamento PC-relative

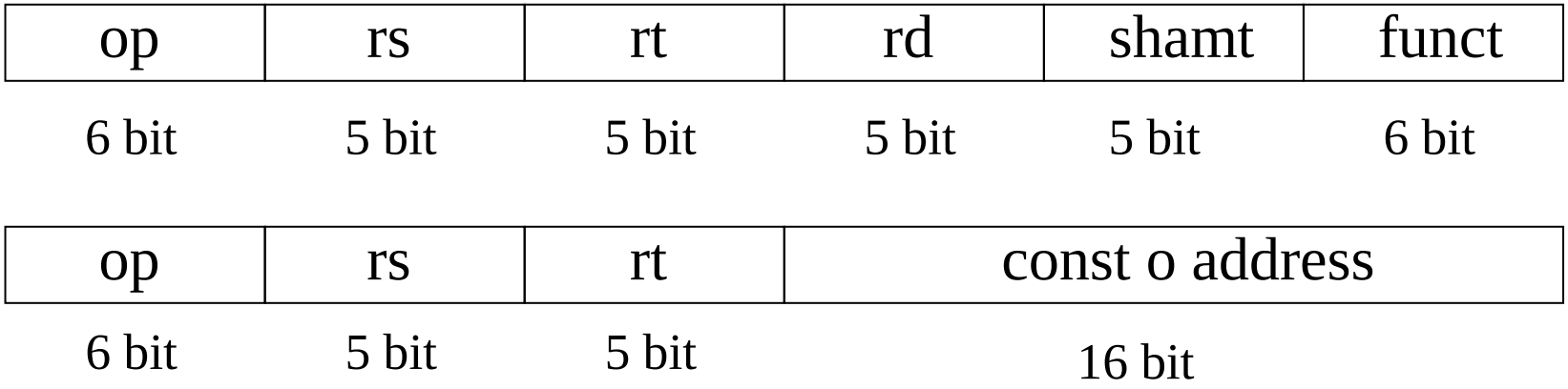
Supponiamo che alla locazione 80016 vi sia l'istruzione

bne $s0, $s1, Esci



- L'indirizzo PC-relative si riferisce al numero di parole che lo separano dalla destinazione, considerando che il PC contiene già l'indirizzo dell'istruzione successiva (è stato infatti incrementato nella fase preliminare dell'esecuzione dell'istruzione corrente)
- Quindi l'offset è (52-20) / 4 = 8: l'istruzione bne somma 8x4=32 byte all'indirizzo dell'istruzione successiva (80020)

Similarità tra i formati

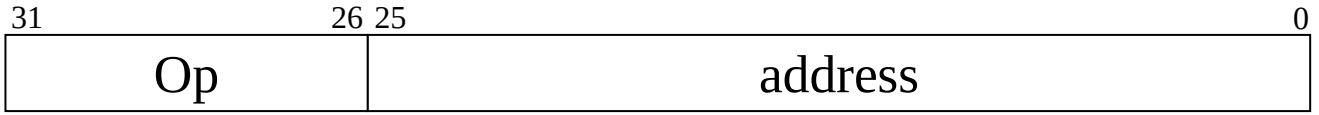


- Il codice op è nella stessa posizione
- Il campo per i registri rs e rt è lo stesso:

  p.es. nel caso un'istruzione debba usare il valore di due registri, questi registri sono sempre indicati nei campi rs e rt!

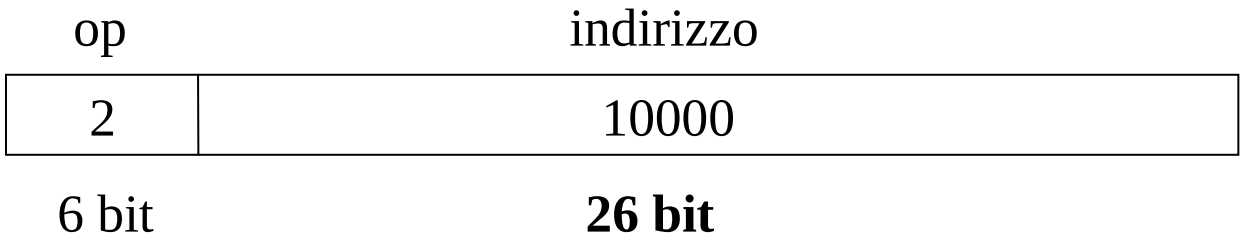
Formato-J (jump) [per istruzioni di salto incondizionato j e jal]

- Queste istruzioni non specificano alcun registro
- I salti della jump e jal non sono in genere "locali": abbiamo bisogno di un formato che permette di specificare una costante con più di 16 bit!



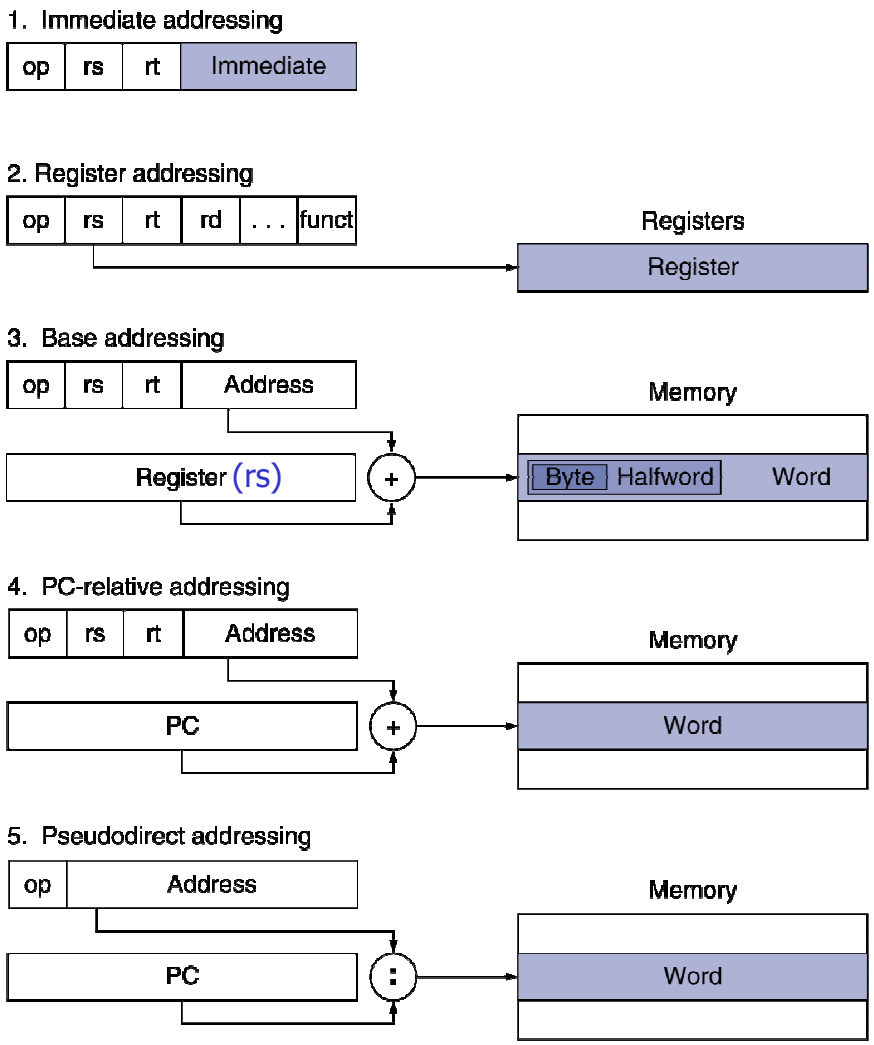
Indirizzamento pseudo-diretto (per i salti incondizionati):

```
j  10000      # vai alla locazione 10000 (espresso come indirizzo di parola)
```



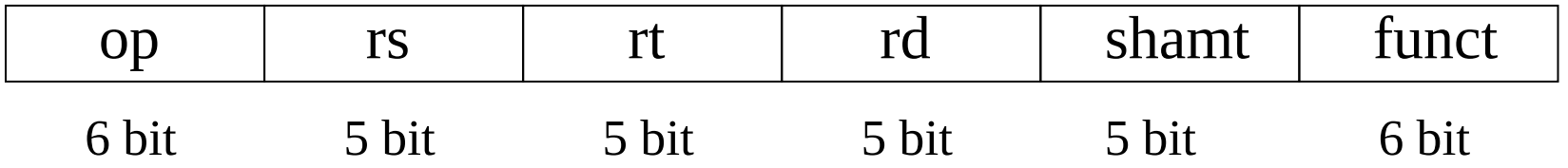
- I 26 bit all'interno dell'istruzione jump contengono un indirizzo in parole, questi 26 bit vengono concatenati con i 4 bit più significativi del PC
- L'indirizzo in byte si ottiene poi moltiplicando per 4 (ovvero, concatenando altri due bit 00) per ottenere un indirizzo a 32 bit

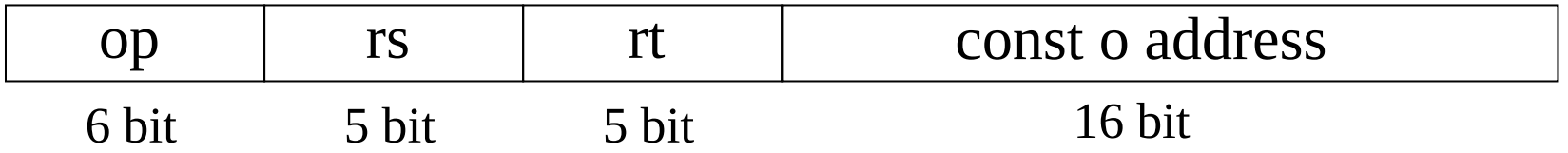
Schema riassuntivo: i 5 modi di indirizzamento del MIPS

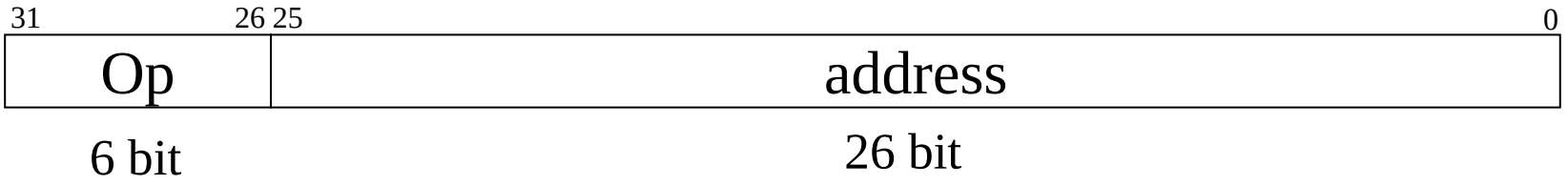


1. ***Indirizzamento immediato***: l'operando è una costante specificata in un campo (16 bit) dell'istruzione
2. ***Indirizzamento tramite registro***: l'operando è un registro specificato in un campo (5 bit) all'interno dell'istruzione
3. ***Indirizzamento tramite base o tramite spiazzamento***: l'operando è in una locazione di memoria individuata dalla somma del contenuto di un registro e di una costante specificata nell'istruzione (16 bit)
4. ***Indirizzamento relativo al program counter*** (PC-relative addressing): l'indirizzo è la somma del contenuto del PC e della costante contenuta nell'istruzione (16 bit) moltiplicata per 4 (esprime uno scostamento in parole)
5. ***Indirizzamento pseudo-diretto***: l'indirizzo è ottenuto concatenando i 26 bit dell'istruzione con i 4 bit più significativi del PC a sinistra e con i bit 00 a destra

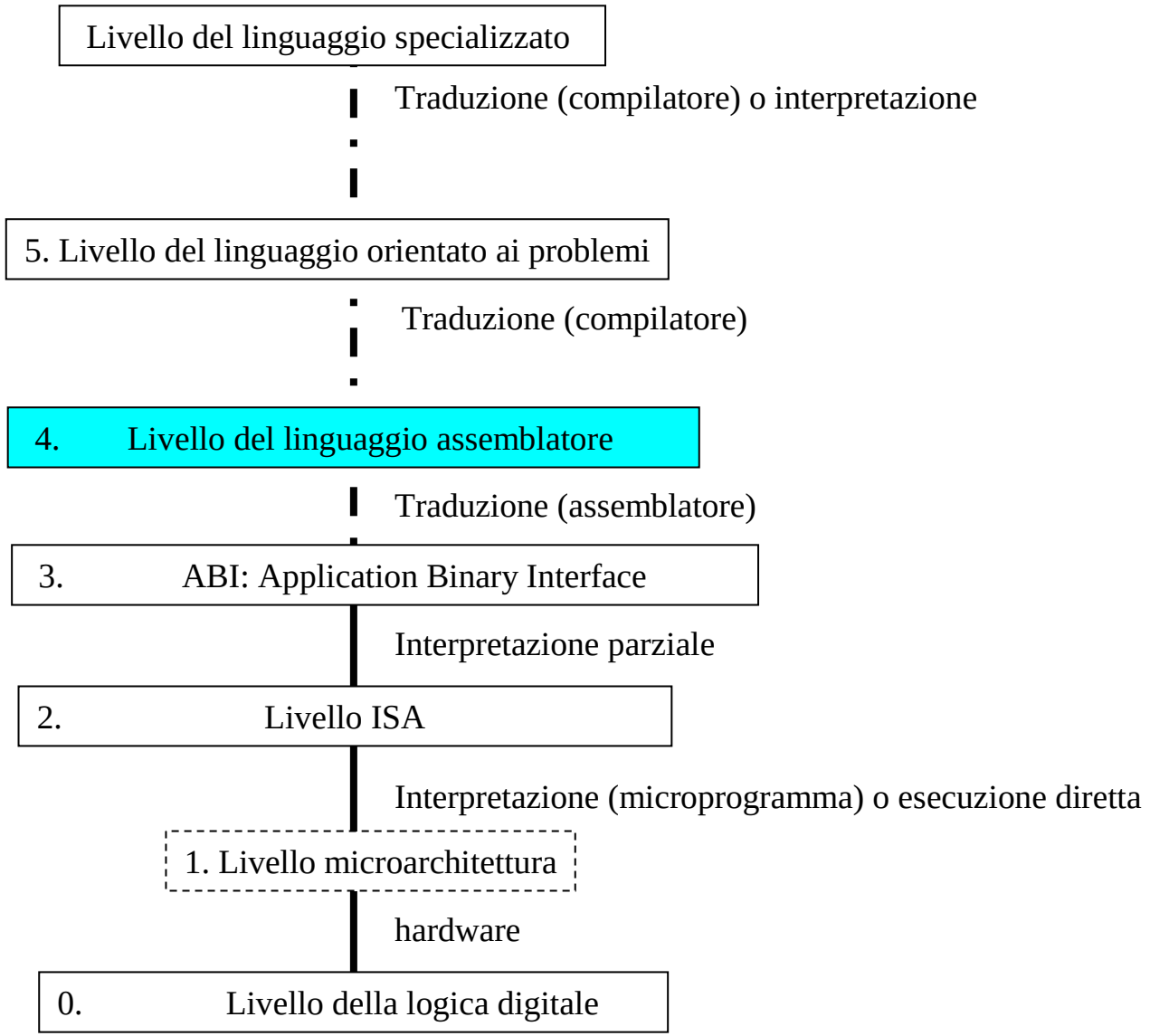
Schema riassuntivo: i 3 formati delle istruzioni MIPS







IL LINGUAGGIO ASSEMBLY DEL MIPS



Oltre ad una pura rappresentazione simbolica delle istruzioni,
il linguaggio assembly “estende” ISA fornendo:

- gestione delle ***etichette***:

  rappresentazione simbolica degli indirizzi di istruzioni e dati

- ***pseudoistruzioni***:

  istruzioni non supportate direttamente dall’hardware, che l’assembler implementa con più istruzioni elementari

- ***macro***:

  nuove istruzioni definite dall’utente, composte da gruppi di istruzioni che vengono sostituite nel codice ad ogni occorrenza della macro

- supporto alla suddivisione del programma in file distinti
  (che possono essere assemblati separatamente)

Uno dei costrutti utilizzati è rappresentato dalle ***direttive***:

comandi all’assemblatore che non generano istruzioni macchina

Etichette

- Nell'assembler MIPS sono *nomi* seguiti da due punti
- Associano il nome alla locazione di memoria successiva
- Possono riferire:
  - istruzioni (ed essere usate nei salti)
  - dati (ed essere usate nelle istruzioni lw e sw)

Direttive .data e .text

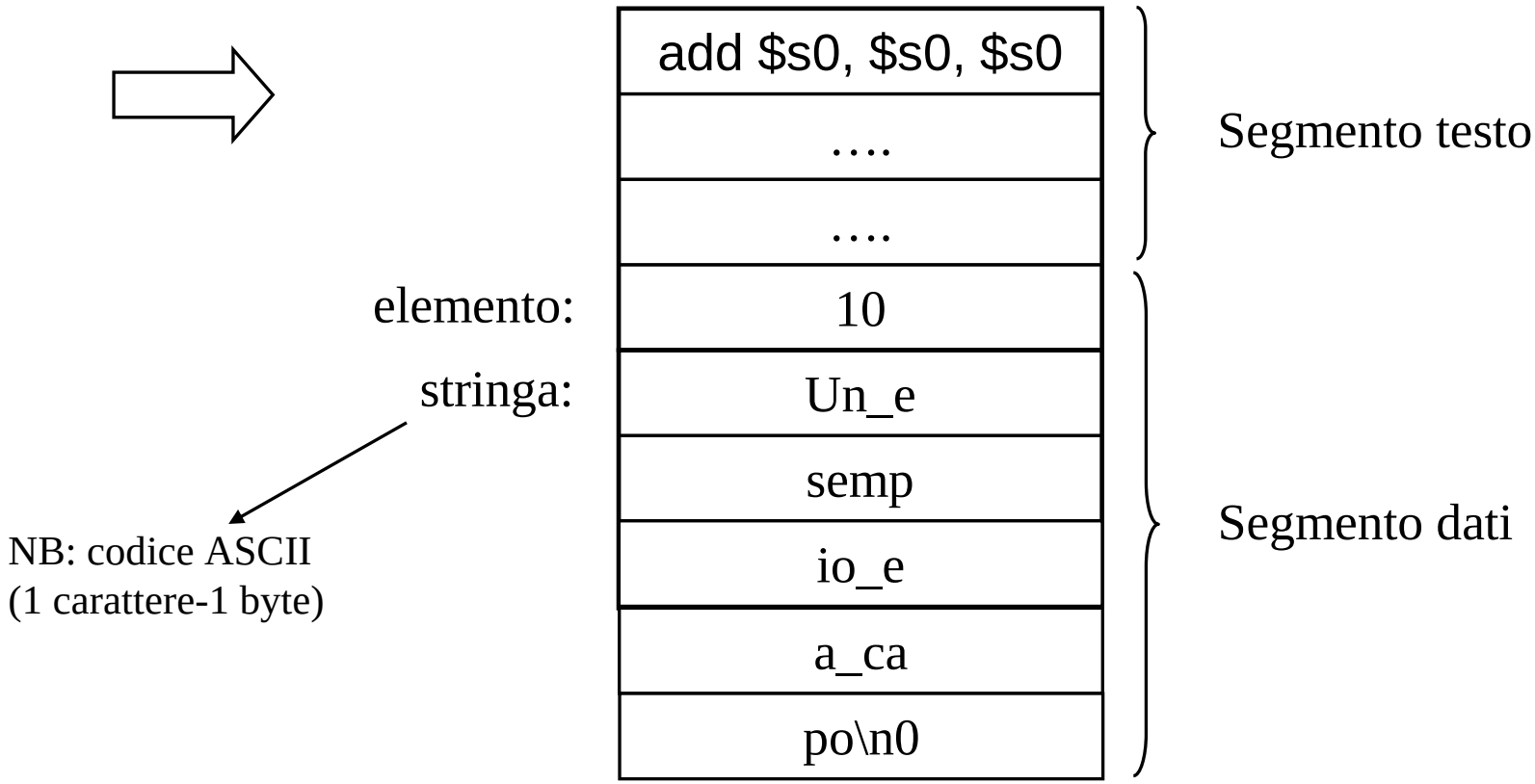
.data	indica che le linee seguenti contengono dati
.text	indica che le linee seguenti contengono istruzioni

Direttive sui dati

.byte b1, …, bn	copia gli *n* valori indicati in *n* bytes successivi in memoria
.word w1, …, wn	copia gli *n* valori a 32 bit in *n* parole successive in memoria
.asciiz str	copia stringa in memoria con un carattere nullo di terminaz.
.ascii str	copia stringa in memoria senza carattere nullo di terminaz.

Esempio

```
            .data
elemento:   .word 10
stringa:    .asciiz “Un esempio e a capo\n”
            .text
            add $s0, $s0, $s0
```



Pseudoistruzioni

- Istruzioni non supportate dall'hardware ma "implementate" dall'assemblatore e utilizzabili come se fossero istruzioni vere e proprie: consentono all'assembler MIPS di avere un insieme di istruzioni più ricco
- Permettono di semplificare le fasi di traduzione e di programmazione
- L'assemblatore le traduce in una o più istruzioni assembly e può utilizzare il registro $at, che per convenzione è riservato all'assemblatore
- Vediamo alcuni esempi…

La pseudoistruzione move

```
move $t0, $t1     # il registro $t0 assume il valore del registro $t1
```

Questa istruzione viene accettata dall'assemblatore e convertita in:

```
add $t0, $zero, $t1
```

La pseudoistruzione la (load address)

la rdest, elemento

carica nel registro *rdest* l'indirizzo (non il valore!) dell'elemento

Esempio

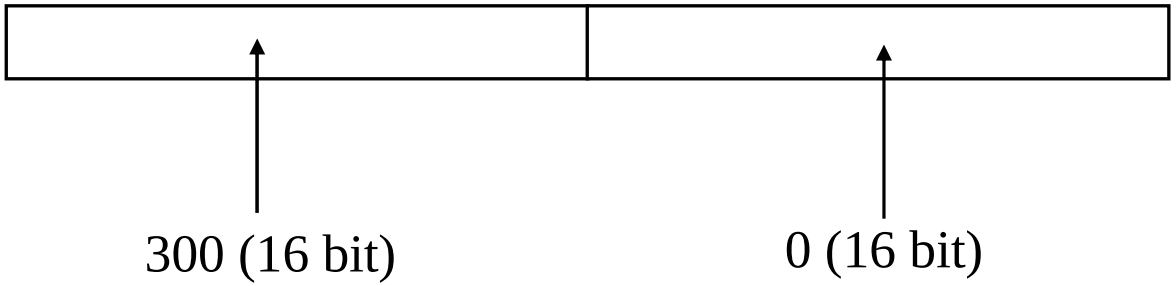
```
                .data
elemento:       .word 10
stringa:        .asciiz "Un esempio e a capo\n"
                .text
    proc:       la $t0, elemento      # carica indirizzo di elemento
                lw $s0, 0($t0)        # carica in $s0 il valore 10
```

Nota sulla gestione delle costanti

- L'ISA del MIPS rende disponibile una istruzione per caricare una costante in un registro, ma la costante è a 16 bit (cfr. formato-I)

load upper immediate

```
lui $s1, 300      # ha come effetto
```



- Ma il programmatore può usare costanti anche più lunghe di 16 bit: l'assemblatore provvede a gestire queste costanti:
  - eventualmente decomponendo le istruzioni del programmatore in più istruzioni
  - utilizzando un registro per convenzione "riservato all'assemblatore" ($at)

Esempio

```
                .data
elemento1:      .word 10            # supponiamo indirizzo = 1001 0000_16
elemento2:      .word 20            # questo è quindi        1001 0004_16
                .text
    proc:       la $t0, elemento1   # carica indirizzo di elemento1
                la $t1, elemento2   # carica indirizzo di elemento2
```

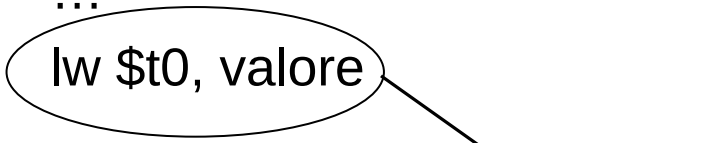
⇨ L'assemblatore può tradurre il codice in questo modo:

```
lui $t0, 4097           # la $t0, elemento1  (4097=1001_16)
lui $at, 4097           # la $t1, elemento2
ori $t1, $at, 4
```

La gestione delle etichette con la lw

- Nel linguaggio assembly si possono usare *lw* e *sw* riferendo direttamente un dato indirizzato da un'etichetta: utilizzando *$at* l'assemblatore implementa questa "pseudoistruzione"

```
            .data
valore:     .word    50          #supponendo sia all'indirizzo 1000 0000(16)
            .text
            …
            lw $t0, valore
```



```
lui $at, 1000(16)
lw $t0, 0($at)
```

Pseudoistruzioni di salto condizionato

blt (branch on less than):

convertita nelle due istruzioni slt e bne

Esempio:

```
blt $s1, $s2, L1
```

diventa

```
slt $at, $s1, $s2
bne $at, $zero, L1
```

NB: notare che in questo caso (a differenza dei precedenti), l'uso di *$at* è necessario

In modo simile, l'assembler supporta:

- bgt (branch on greater than)
- bge (branch on greater/equal than)
- ble (branch on less/equal than)

Saltare ad indirizzi lontani

- Se l'indirizzo specificato in un salto condizionato è troppo lontano, l'assembler risolve il problema inserendo un salto incondizionato al posto di quello condizionato invertendo la condizione originale

```
        beq     $s0, $s1, L1
```

diventa

```
        bne     $s0, $s1, L2
        j       L1
    L2:
```

Macro

- Permettono di dare un nome ad una sequenza di istruzioni usata di frequente nel programma e di utilizzare questo nome al posto della sequenza
- L'assembler sostituisce ogni occorrenza del nome con la sequenza di istruzioni
- E' possibile definire parametri formali della macro
- Nota: ben diverse dalle procedure!

Esempio

```
.macro scambia($arg1, $arg2)        #direttiva!
move $t0, $arg1
move $arg1, $arg2
move $arg2, $t0
.end_macro

scambia($t4, $t5)
```

Etichette locali e globali

- Etichetta locale: visibile solo all'interno del file in cui è definita
- Etichetta globale (o esterna): può essere riferita anche in un altro file
- Per default le etichette sono locali: per dichiararle globali deve essere usata la direttiva .globl

Esempio

```
            .text
            .globl      main
main:       add $t0, $s1, $s2
            …
loop:       lw $t0, 0($s0)
            …
            j loop
            …
            .data
Str:        .asciiz “Ciao mondo”
```

Etichette locali e globali

- Etichetta locale: visibile solo all'interno del file in cui è definita
- Etichetta globale (o esterna): può essere riferita anche in un altro file
- Per default le etichette sono locali: per dichiararle globali deve essere usata la direttiva .globl

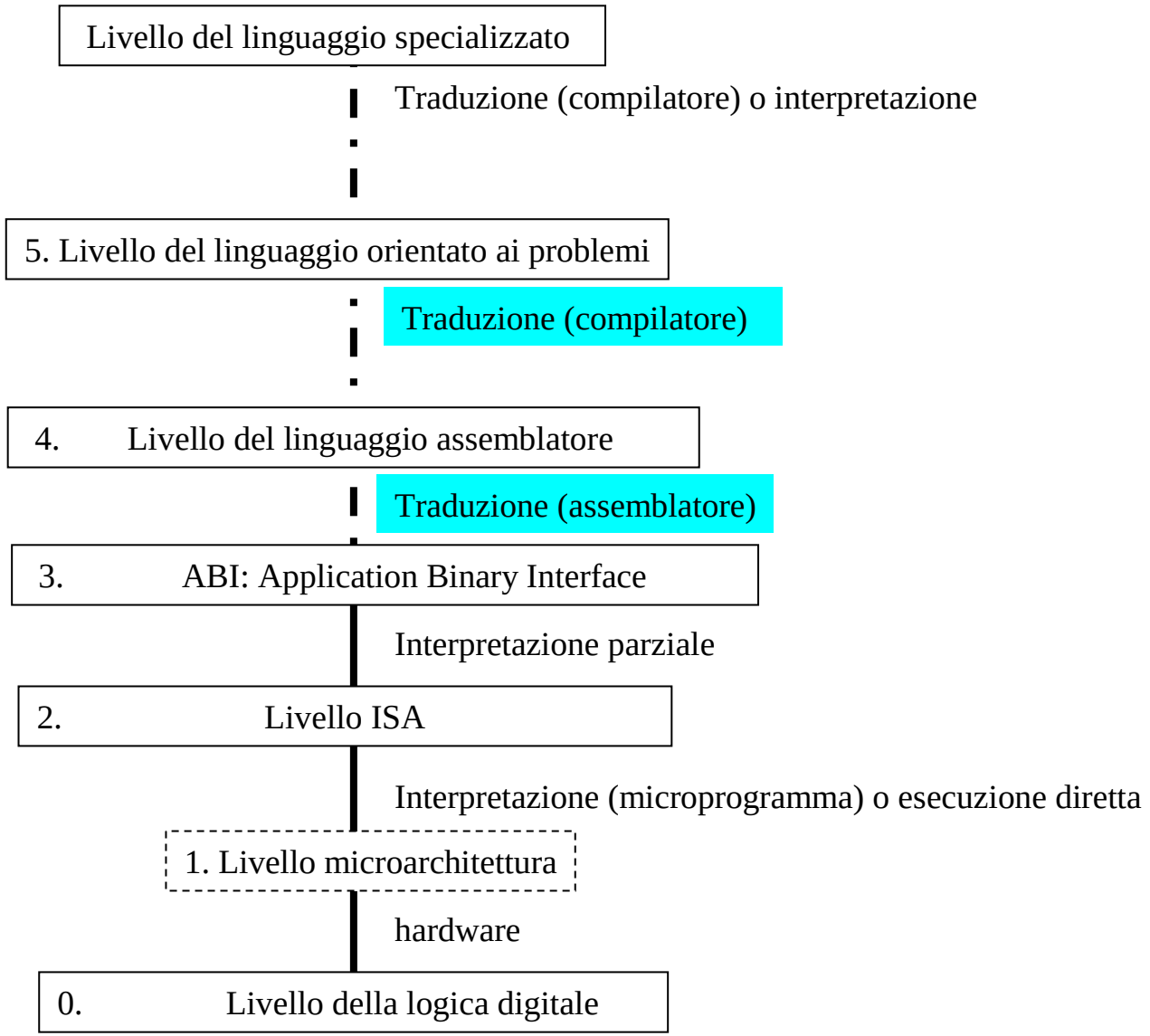
Esempio

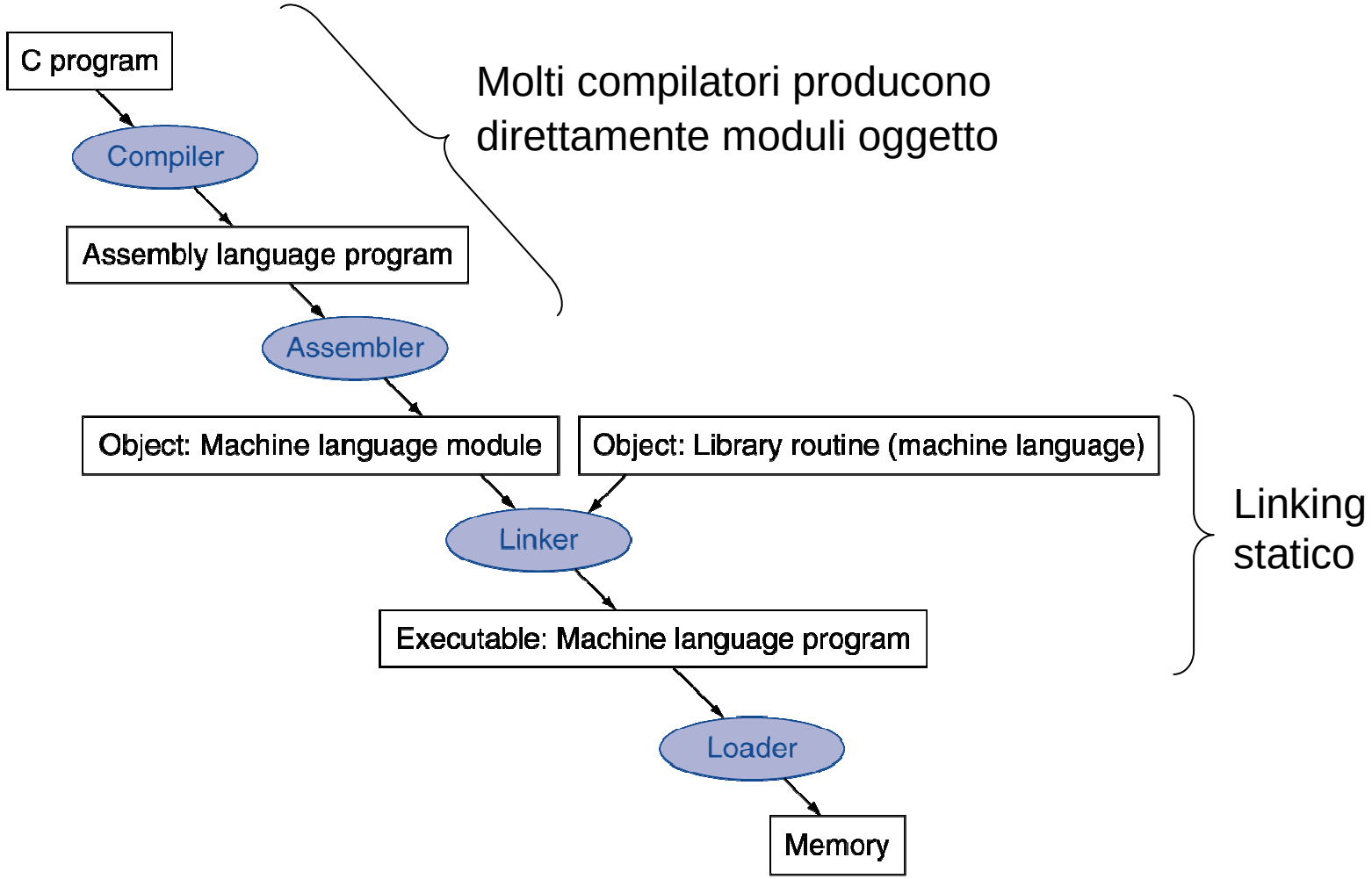
```
            .text
            .globl      main
main:       add $t0, $s1, $s2
            …
loop:       lw $t0, 0($s0)
            …
            j loop
            …
            .data
Str:        .asciiz “Ciao mondo”
```

main è globale (accessibile da altri file)

loop e Str sono locali

COMPILATORE, ASSEMBLER E LINKER




C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Molti compilatori producono direttamente moduli oggetto
Linking statico

Assemblatore

• L'assemblatore lavora su diversi sorgenti indipendentemente l'uno dall'altro: conosce solo le etichette definite nello stesso file!

PRIMA PASSATA

Partendo dall'indirizzo 0 per la parte *text* e dall'indirizzo 0 per i dati (statici):

• calcola l'indirizzo di tutte le etichette per istruzioni e dati

• memorizza le associazioni etichetta-indirizzo in una ***tabella dei simboli***

SECONDA PASSATA

Ripercorre istruzione per istruzione e codifica in linguaggio macchina:

• alcune istruzioni riferiscono etichette definite nello stesso file:
si usa la tabella dei simboli

• alcune istruzioni riferiscono etichette definite in altri file: unresolved reference

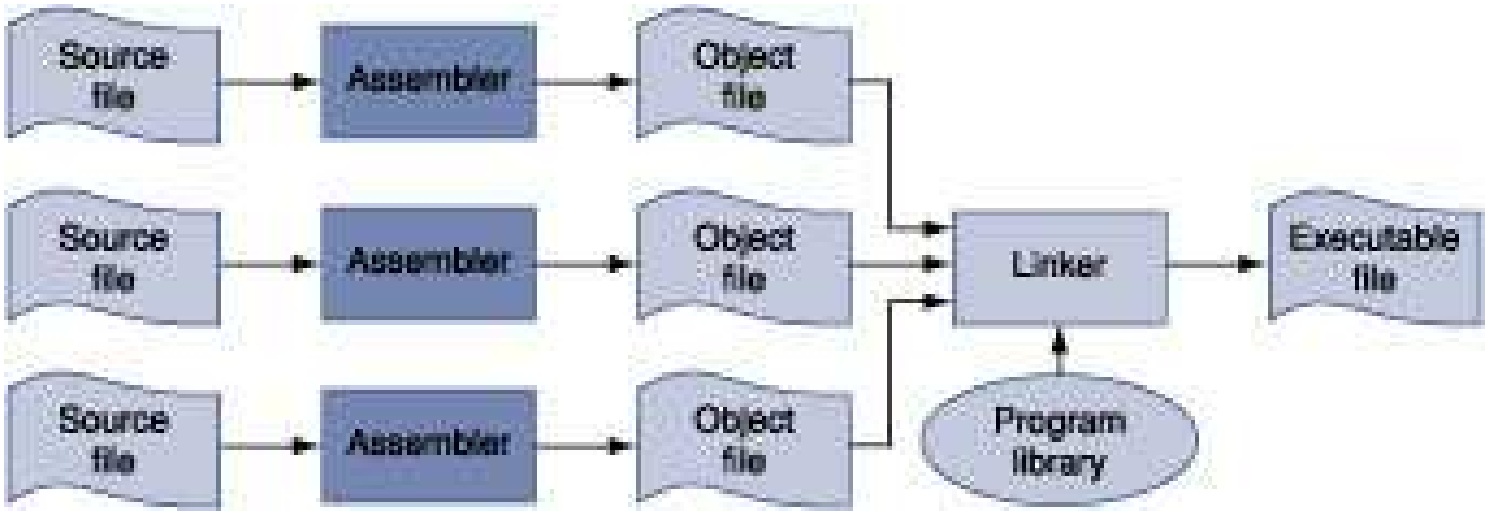
⇨ PRODUCE UN CODICE OGGETTO
(deve contenere tutte le informazioni per consentire linking)

Il codice oggetto

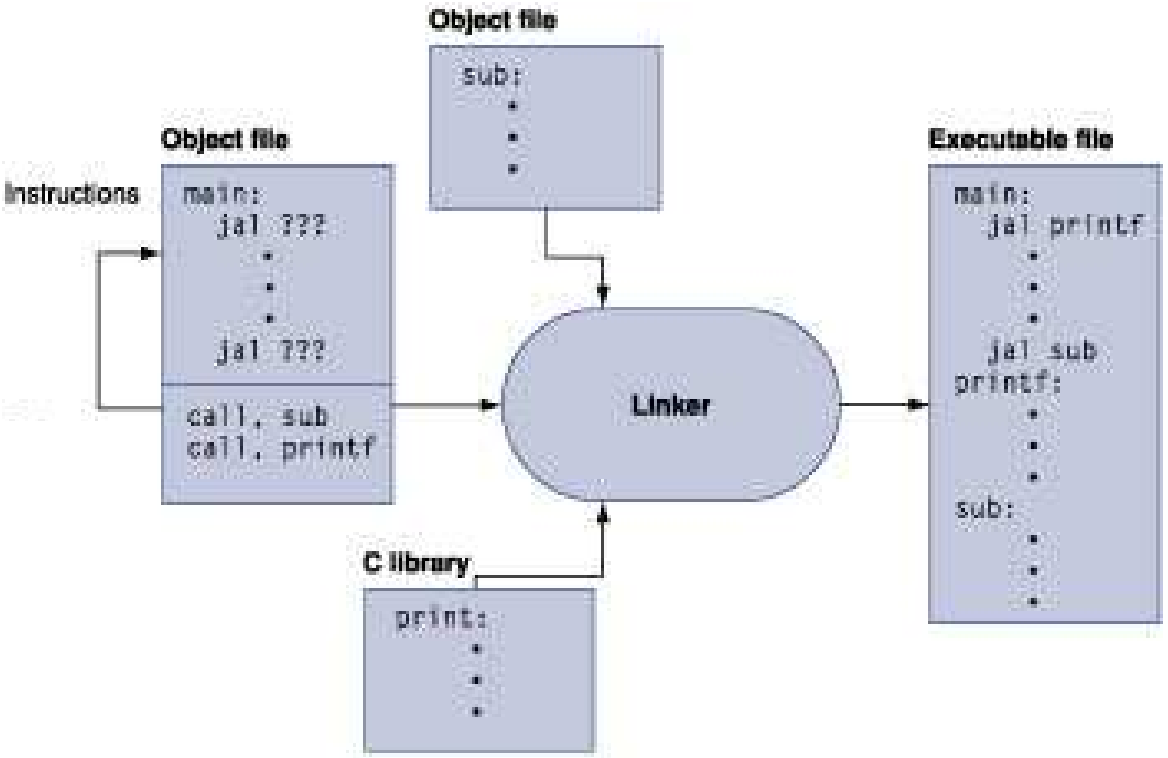
- *Header*: descrive i blocchi successivi del file (dimensioni e posizione)
- *Text segment*: codice macchina (contiene riferimenti non risolti)
- *Data segment*: dati "statici" (durata: intera vita del programma)
- *Relocation information*: permette di identificare le istruzioni che devono cambiare a seconda dell'indirizzo a partire da cui il linker collocherà il modulo (nel MIPS: istruzioni *j*, *jal*, *lw*, *sw*, mentre salti condizionati no!)
- *Tabella dei simboli*: contiene
  - gli indirizzi delle etichette globali (visibili dall'esterno) definite nel modulo
  - la lista dei riferimenti non risolti
- *Informazioni di debugging*

NB: il modulo oggetto non è eseguibile (riferimenti non risolti)

Il linker



- Assicura che non ci siano riferimenti non risolti: ad ogni riferimento non risolto all'interno di un modulo oggetto deve corrispondere
  - un'etichetta globale definita in un altro oggetto (symbol table)
  - procedure o strutture dati definite in una libreria di programmi
- Determina le posizioni di memoria delle parti *text* e *data* di ciascun modulo:
  - dispone tutti i codici in sequenza nella parte *text* della memoria
  - dispone tutti i dati in sequenza nella parte *data* della memoria
  - usa le informazioni degli header per calcolare gli indirizzi per ciascun modulo



• Riloca le istruzioni (elencate nella relocation information) e codifica le istruzioni con riferimenti non risolti

⇨ Produce un file eseguibile

(header, text, data, debugging information)

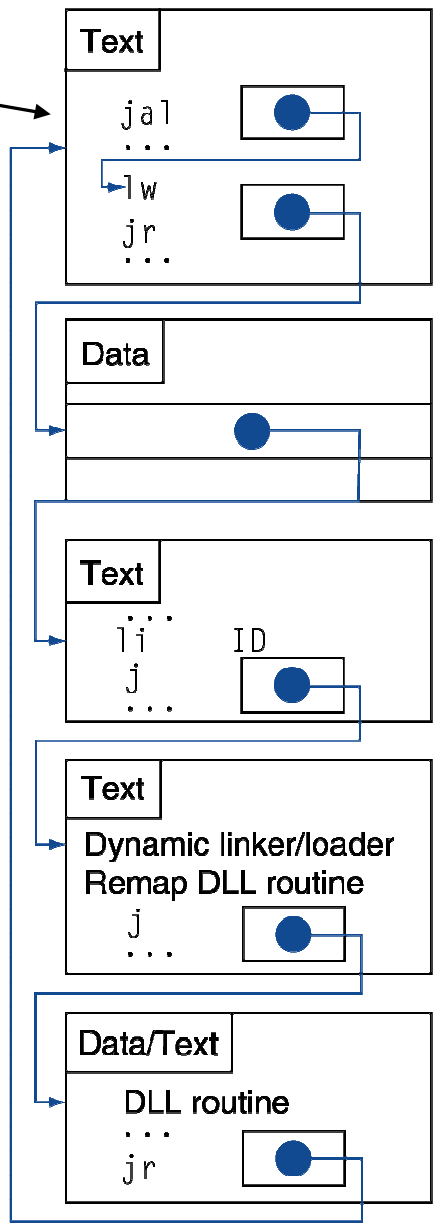
caricato in memoria e posto in esecuzione dal ***loader*** (∈ S.O.)

Linking dinamico

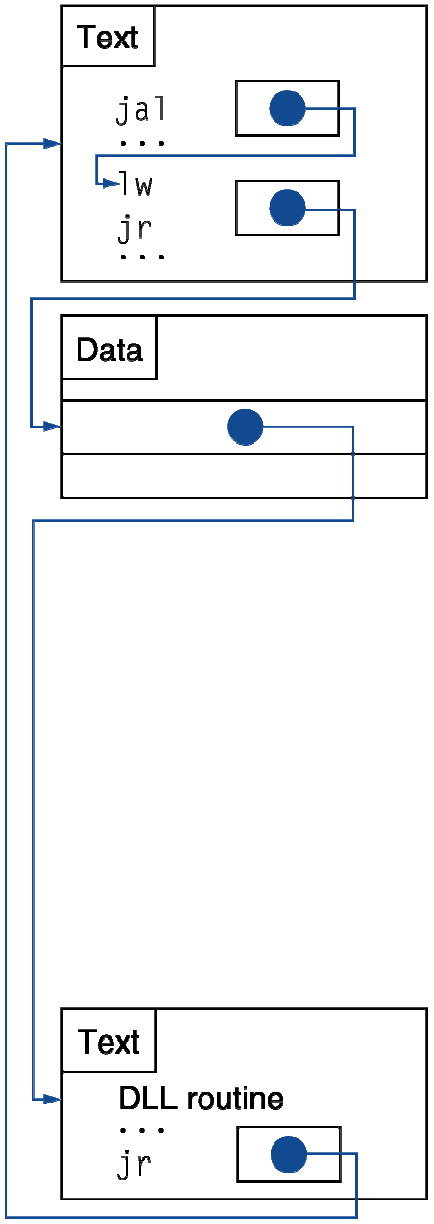
- Il linking statico (visto in precedenza) ha due svantaggi:
  - nuove versioni delle librerie (p.es. correzione bugs) non vengono utilizzate
  - linking di un'intera libreria anche se ne viene usata solo una parte
- IDEA: dynamically linked libraries (DLL - librerie collegate dinamicamente)
  - le procedure sono collegate al programma (e caricate in memoria)
    in fase di esecuzione e solo se e quando servono

NB: non esiste la jal che salta ad un indirizzo puntato da un registro…

Dummy routine: carica in un registro l'ID della procedura richiesta (ora serve, poi non più) e salta a linker/loader dinamico

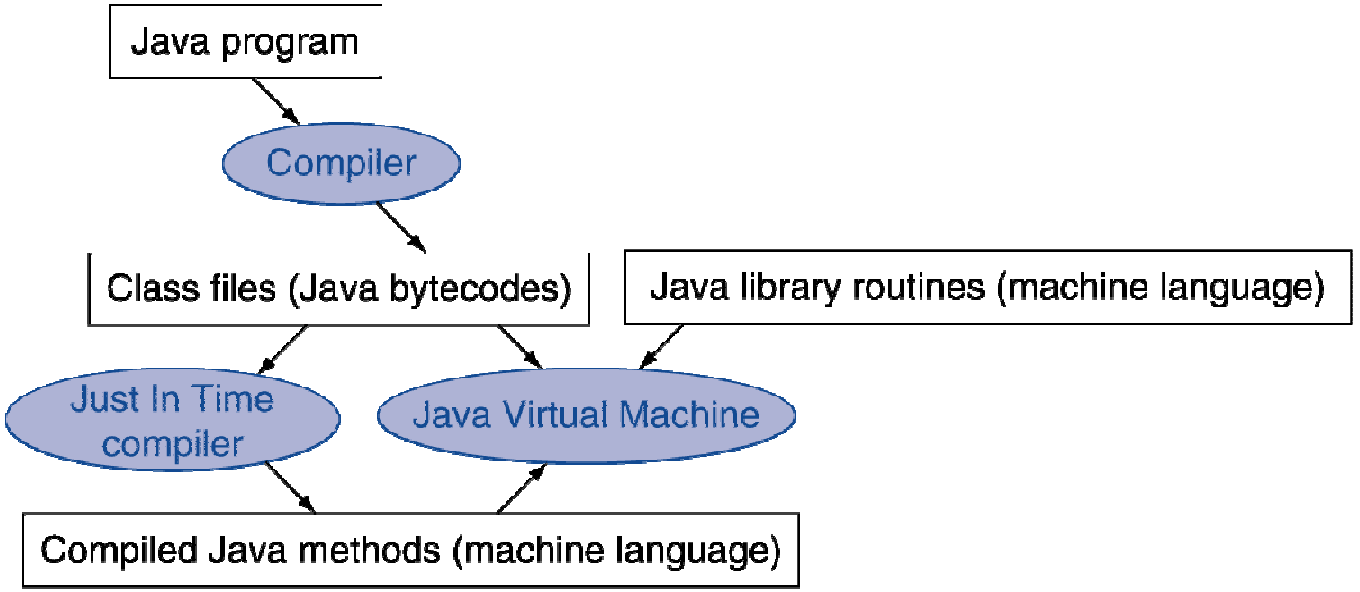


a. First call to DLL routine



b. Subsequent calls to DLL routine

Approccio ibrido e compilazione just-in-time



- Per il java, la compilazione produce un codice intermedio (***java bytecode***) indipendente dalla macchina, che può essere eseguito dalla ***Java Virtual Machine***

  $\Rightarrow$ portabilità, ma bassa performance (interpretazione)

- Compilazione just-in-time: i metodi più frequentemente utilizzati vengono compilati in codice macchina a runtime

  $\Rightarrow$ la performance migliora nel tempo ed è migliore per programmi “frequenti”