A

Gli assemblatori, i linker e il simulatore SPIM

A cura di James R. Larus, Microsoft Research

APPENDICE

La paura di gravi disgrazie non può da sola giustificare la soppressione della libertà di parola e dell'assembler.
Louis Brandeis, Whitney v. California, 1927

A.1 Introduzione

L'impiego dei numeri binari nei calcolatori è un modo naturale ed efficiente per codificare le istruzioni. Gli esseri umani, invece, incontrano molta difficoltà nel comprendere e manipolare tali numeri. Le persone leggono e scrivono simboli (parole) molto più volentieri di lunghe sequenze di cifre. Nel capitolo 2 abbiamo mostrato che non dobbiamo scegliere se usare parole o numeri, perché le istruzioni del calcolatore possono essere rappresentate in molti modi. Gli esseri umani possono leggere e scrivere simboli, mentre i calcolatori possono gestire i numeri binari equivalenti a quei simboli. Questa appendice descrive il processo mediante il quale un programma leggibile dall'uomo viene tradotto in una forma che il calcolatore è in grado di eseguire, fornisce alcuni suggerimenti per la scrittura di programmi assembler e spiega come eseguire tali programmi su SPIM, un simulatore che esegue programmi MIPS. Le versioni per UNIX, Windows e Mac OS X del simulatore SPIM sono disponibili sul CD.

Il linguaggio assembler è la rappresentazione simbolica della codifica binaria delle istruzioni, il **linguaggio macchina**. Il linguaggio assembler è più leggibile del linguaggio macchina, perché utilizza simboli anziché bit. I simboli associano un nome alle sequenze di bit che ricorrono spesso, come i codici operativi o il numero dei registri, in modo tale che le persone li possano leggere e ricordare. Inoltre, il linguaggio assembler permette ai programmatori di utilizzare etichette per identificare e associare un nome a specifiche parole di memoria che contengono istruzioni o dati.

Linguaggio macchina: rappresentazione binaria utilizzata per la comunicazione all'interno di un calcolatore.

Assemblatore: un programma che traduce una versione simbolica delle istruzioni nella versione binaria.

Macro: una funzione che riconosce e sostituisce delle stringhe; fornisce un semplice meccanismo per dare un nome a sequenze di istruzioni utilizzate di frequente.

Riferimento non risolto: un riferimento che richiede più informazione, proveniente da una sorgente esterna, per essere completato.

Linker o **link editor:** un programma di sistema che combina programmi in linguaggio macchina assemblati indipendentemente e risolve tutte le etichette indefinite per produrre un file eseguibile.

Direttiva dell'assemblatore: un'istruzione che indica all'assemblatore come tradurre un programma, ma che non produce istruzioni in linguaggio macchina. Inizia sempre con un punto.

Un dispositivo, chiamato **assemblatore**, traduce il linguaggio assembler in istruzioni binarie. Gli assemblatori permettono quindi di utilizzare una rappresentazione più comprensibile di quella binaria del calcolatore, rendendo così più semplice la scrittura e la lettura dei programmi. Una caratteristica di questa rappresentazione è che vengono assegnati nomi simbolici alle operazioni e alle posizioni dei dati. Esistono, inoltre, strumenti di programmazione che migliorano la leggibilità di un programma. Per esempio, le **macro**, descritte nel Paragrafo A.2, permettono al programmatore di estendere il linguaggio assembler definendo operazioni nuove.

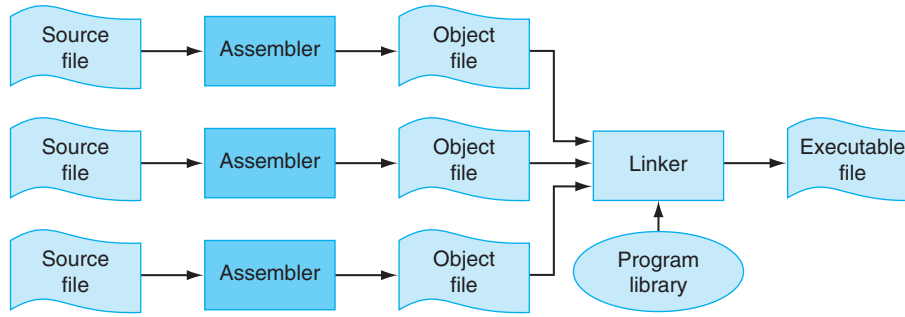


Figura A.1.1. Il processo che produce un file eseguibile. Un assemblatore traduce il file contenente codice assembler in un file oggetto, il quale viene collegato ad altri file e a funzioni di libreria per produrre un file eseguibile.

Un assemblatore legge un singolo **file sorgente** in linguaggio assembler e produce un **file oggetto** contenente istruzioni in linguaggio macchina e alcune informazioni ausiliarie che permettono di combinare diversi file oggetto in un unico programma. La Figura A.1.1 illustra il modo in cui si costruisce un programma. La maggior parte dei programmi consiste di parecchi file, detti anche **moduli**, che vengono scritti, compilati e assemblati separatamente. Un programma può anche utilizzare procedure scritte in precedenza, fornite in una **libreria di programmi**. Un modulo contiene tipicamente dei **riferimenti** a sottoprogrammi e dati definiti negli altri moduli e nelle librerie. Il codice contenuto in un modulo non può essere eseguito quando contiene **riferimenti non risolti** a etichette presenti in altri file oggetto o nelle librerie. Un altro strumento, chiamato **linker**, combina diversi file oggetto e di libreria e crea un **file eseguibile**, il quale può essere eseguito dal calcolatore.

Per comprendere i vantaggi del linguaggio assembler si osservino le prossime figure, che contengono tutte un breve sottoprogramma che calcola e stampa la somma dei quadrati da 1 a 100. La Figura A.1.2 mostra il linguaggio macchina che viene eseguito da un calcolatore MIPS. Con uno sforzo enorme si possono utilizzare le tabelle dei codici operativi e dei formati delle istruzioni che abbiamo visto nel secondo capitolo per tradurre le istruzioni in linguaggio macchina in un programma simbolico simile a quello mostrato nella pagina seguente.

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101111000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

Figura A.1.2. Il codice in linguaggio macchina MIPS di una procedura che calcola e stampa la somma dei quadrati degli interi da 0 a 100.

La rappresentazione del programma mostrata in Figura A.1.3 è molto più semplice da leggere, perché le operazioni e gli operandi sono rappresentati da simboli anziché da sequenze di bit. Questo linguaggio assembler, tuttavia, è ancora poco comprensibile, perché le locazioni di memoria sono identificate dal loro indirizzo e non da un'etichetta simbolica.

La Figura A.1.4 mostra il linguaggio assembler in cui agli indirizzi di memoria viene assegnata un'etichetta. La maggior parte dei programmatori preferisce leggere e scrivere il codice in questa forma. I nomi che iniziano con un punto, per esempio `.data` e `.globl`, sono **direttive dell'assemblatore** che indicano all'assemblatore come tradurre il programma, ma non producono istruzioni macchina. I nomi seguiti dai due punti, come `str:` o `main:`, sono etichette che vengono associate alla locazione di memoria successiva. Questo programma possiede la leggibilità della maggior parte dei programmi assembler (eccezion fatta per la mancanza di commenti), ma risulta ancora difficile da «utilizzare», perché sono necessarie molte operazioni elementari per eseguire anche le operazioni più semplici e perché fornisce pochi indizi sul funzionamento del codice (a causa della mancanza dei costrutti di controllo del flusso).

```
addiu       $29, $29, -32
sw          $31, 20($29)
sw          $4,  32($29)
sw          $5,  36($29)
sw          $0,  24($29)
sw          $0,  28($29)
lw          $14, 28($29)
lw          $24, 24($29)
multu       $14, $14
addiu       $8,  $14, 1
slti        $1,  $8, 101
sw          $8,  28($29)
mflo        $15
addu        $25, $24, $15
bne         $1,  $0, -9
sw          $25, 24($29)
lui         $4,  4096
lw          $5,  24($29)
jal         1048812
addiu       $4,  $4, 1072
lw          $31, 20($29)
addiu       $29, $29, 32
jr          $31
move        $2,  $0
```

Figura A.1.3. La stessa procedura scritta in linguaggio assembler. Il codice della procedura non contiene delle etichette in corrispondenza delle locazioni di memoria e non presenta commenti.

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

Figura A.1.4. La stessa procedura scritta in linguaggio assembler con le etichette, ma senza commenti. I comandi che iniziano con un punto sono direttive dell'assemblatore. `.text` indica che le linee seguenti contengono istruzioni mentre `.data` indica che contengono dati. `.align n` indica che gli elementi contenuti nelle linee seguenti devono essere allineati a multipli di 2^n byte; di conseguenza, `.align` indica che l'elemento successivo deve essere allineato al confine di una parola. `.globl main` dichiara che main è un simbolo globale che deve essere visibile anche al codice memorizzato in altri file. Infine, `.asciiz` scrive in memoria una stringa che viene terminata con il carattere null.

La procedura C riportata in Figura A.1.5, invece, risulta allo stesso tempo più breve e più chiara, grazie al fatto che alle variabili vengono assegnati dei nomi facilmente memorizzabili e che il ciclo è reso esplicito (e non realizzato mediante diramazioni del codice). Di fatto, la procedura C è l'unica che abbiamo scritto; le altre versioni del programma sono state prodotte da un compilatore C e da un assemblatore.

```
   #include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

Figura A.1.5. La funzione scritta nel linguaggio ad alto livello C.

Linguaggio sorgente: il linguaggio ad alto livello in cui viene originariamente scritto un programma.

In generale, il linguaggio assembler svolge due ruoli (si veda la Figura A.1.6). Anzitutto costituisce il linguaggio di output dei compilatori. Un **compilatore** traduce un programma scritto in un **linguaggio ad alto livello** (come C o Pascal) in un programma equivalente in assembler o in linguaggio macchina. Il linguaggio ad alto livello viene chiamato **linguaggio sorgente**, mentre l'output del compilatore è prodotto nel suo **linguaggio target**.

L'assembler, però, è anche un linguaggio con cui scrivere programmi. In passato questo era il suo impiego principale. Oggi, invece, con l'avvento di memorie principali di grandi dimensioni e di compilatori sempre più efficienti, la maggior parte dei programmatori scrive il codice in linguaggio ad alto livello e raramente «vede» le istruzioni eseguite dal calcolatore. Ciononostante, il linguaggio assembler continua ad essere utilizzato per scrivere quei programmi in cui la velocità o la dimensione della memoria risulta critica, o per sfruttare funzionalità hardware che non hanno una corrispondenza nei linguaggi ad alto livello.

Sebbene questa appendice si concentri sul linguaggio assembler del MIPS, la programmazione in assembler sulla maggior parte degli altri tipi di processore è molto simile. Le istruzioni aggiuntive e le modalità di indirizzamento delle macchine CISC, come il VAX, possono rendere i programmi assembler più brevi, ma non cambiano il processo richiesto per assemblare un programma né forniscono al linguaggio assembler i vantaggi dei linguaggi ad alto livello, come il controllo dei tipi o il flusso strutturato.

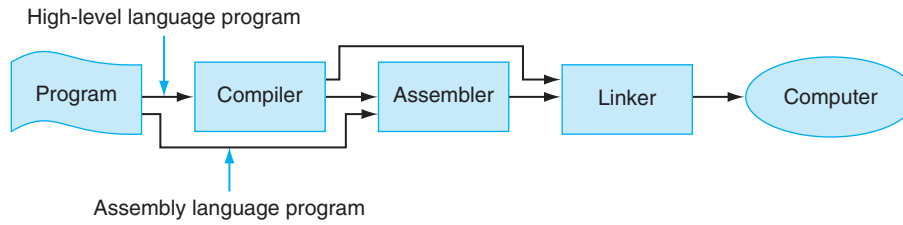


Figura A.1.6. Il codice in linguaggio assembler viene scritto dai programmatori o viene prodotto dai compilatori.

Quando conviene utilizzare il linguaggio assembler

Come abbiamo già accennato, la scelta di scrivere un programma in linguaggio assembler dipende dal fatto che in certe situazioni la velocità e l'occupazione della memoria da parte del programma sono due aspetti critici. Per esempio, consideriamo un calcolatore che controlla un sistema meccanico, come i freni di un'automobile (un calcolatore incorporato in un altro dispositivo, in questo caso un'automobile, viene chiamato **calcolatore embedded**): questo tipo di calcolatore deve rispondere rapidamente e in modo prevedibile agli eventi esterni. Poiché un compilatore introduce incertezza nel tempo di esecuzione delle operazioni, i programmatori potrebbero avere difficoltà ad assicurare che un programma scritto in un linguaggio ad alto livello abbia sempre un determinato tempo di risposta; nel caso dell'automobile, per esempio, la necessità è che sia sempre attivata una risposta entro 1 millisecondo da quando un sensore rileva che uno pneumatico sta pattinando. I programmatori assembler, invece, hanno un controllo completo sulla sequenza e sui tempi di esecuzione delle istruzioni; inoltre, nelle applicazioni embedded una riduzione delle dimensioni del programma che consenta di salvare il programma in memoria utilizzando un numero minore di chip ne diminuisce il costo.

Un approccio ibrido, in cui la maggior parte di un programma è scritta in linguaggio ad alto livello e alcune funzioni (quelle per cui il tempo di esecuzione è un fattore critico) sono scritte in assembler, sfrutta i punti di forza di entrambi i linguaggi. I programmi tipicamente trascorrono la maggior parte del loro tempo di esecuzione eseguendo una piccola frazione del codice sorgente

del programma stesso. Questa osservazione è, di fatto, il principio di località su cui sono basate le memorie cache (si veda il Paragrafo 5.1 del Capitolo 5).

La profilazione di un programma (*program profiling*) indica dove si concentra l'esecuzione di un programma e può quindi identificare le parti del codice il cui tempo di esecuzione è critico: in molti casi, tali porzioni possono essere velocizzate impiegando strutture dati o algoritmi migliori. In alcuni casi, tuttavia, miglioramenti significativi delle prestazioni si ottengono soltanto mediante la riscrittura in linguaggio assembler della parte critica del programma.

Tali miglioramenti non sono necessariamente un'indicazione del fallimento dei compilatori. Questi, in generale, producono un codice macchina più efficiente di quanto farebbero i programmatori. Un programmatore, però, è in grado di comprendere meglio gli algoritmi e il comportamento di un programma, e questa comprensione gli permette di escogitare soluzioni che possano migliorare notevolmente le prestazioni di piccole parti del codice. Per esempio, i programmatori spesso considerano parecchie procedure contemporaneamente durante la scrittura del codice, mentre i compilatori gestiscono tipicamente una procedura alla volta. I compilatori, inoltre, devono seguire convenzioni rigorose nella gestione dei registri nelle chiamate a procedura; i programmatori, invece, possono aumentare le prestazioni di un programma mantenendo nei registri le variabili utilizzate più di frequente, anche a cavallo delle chiamate a procedura.

Un altro vantaggio importante dell'uso del linguaggio assembler è la possibilità di sfruttare le istruzioni speciali, per esempio quelle che copiano stringhe o che riconoscono strutture di codice. I compilatori, in molti casi, non sono in grado di determinare se il ciclo di un programma possa essere sostituito da una singola istruzione. Il programmatore che ha scritto il ciclo, invece, lo può facilmente sostituire con una singola istruzione.

L'utilizzo del linguaggio assembler per scrivere programmi diventa una scelta obbligata quando non è disponibile alcun linguaggio ad alto livello. È il caso, per esempio, di molti calcolatori datati o speciali, che non dispongono di compilatori.

Il vantaggio dei programmatori sui compilatori negli anni si è comunque ridotto, grazie al continuo miglioramento delle tecniche di compilazione e alla crescente complessità delle pipeline (si veda il Capitolo 4).

Svantaggi del linguaggio assembler

I molteplici svantaggi del linguaggio assembler hanno portato al suo abbandono da parte di molti sviluppatori. Forse lo svantaggio principale è che i programmi scritti in assembler sono specifici per una particolare macchina e devono essere completamente riscritti per poter essere eseguiti su un'altra architettura. La rapida evoluzione dei calcolatori discussa nel primo capitolo comporta che le architetture diventino obsolete; un programma in linguaggio assembler rimane strettamente legato all'architettura per cui è stato scritto e non può essere eseguito sulle architetture nuove, più performanti e meno costose.

Un altro svantaggio è che i programmi assembler sono più lunghi dei programmi equivalenti scritti in un linguaggio ad alto livello. Per esempio, il programma C di Figura A.1.5 è lungo 11 linee, mentre il programma assembler di Figura A.1.4 è lungo 31 linee. Per programmi più complessi, il rapporto tra la lunghezza dei due codici (detto **rapporto di espansione**) può risultare molto maggiore del fattore tre riscontrato in questo esempio. Sfortunatamente, alcuni studi sperimentali hanno mostrato che un programmatore scrive circa lo stesso numero di righe di codice al giorno sia in assembler sia in linguaggio ad alto livello; questo significa che i programmatori sono circa x volte più pro-

duttivi quando lavorano con un linguaggio ad alto livello, dove x è il rapporto di espansione del linguaggio assembler.

È importante sottolineare, inoltre, che i programmi più lunghi sono più difficili da leggere e da capire, oltre al fatto che contengono più errori. Il linguaggio assembler aggrava questo problema, a causa della sua mancanza di strutturazione. Le comuni espressioni di programmazione, come i cicli e il costrutto *if then*, devono essere gestite per mezzo di salti condizionati e incondizionati. I programmi che ne risultano sono difficili da leggere: per farlo è necessario risalire a ogni costrutto di alto livello a partire dalle istruzioni, e ogni istanza del costrutto può essere leggermente diversa. Per esempio, osservando la figura A.1.4 si provi a rispondere alle seguenti domande: che tipo di ciclo viene utilizzato? Quali sono gli estremi inferiore e superiore del ciclo?

Approfondimento. I compilatori possono produrre direttamente il linguaggio macchina anziché appoggiarsi a un assemblatore. Questi compilatori tipicamente lavorano molto più velocemente di quelli che utilizzano un assemblatore durante la compilazione. Tuttavia, un compilatore che genera direttamente il linguaggio macchina deve assolvere a molti dei compiti che normalmente sono propri dell'assemblatore, come risolvere gli indirizzi e codificare le istruzioni come numeri binari. I compilatori sono quindi il risultato di un compromesso tra la velocità di compilazione e la semplicità del compilatore stesso.

Approfondimento. A dispetto delle considerazioni fin qui fatte, alcune applicazioni embedded vengono scritte in linguaggi ad alto livello. Si tratta spesso di programmi grandi e complessi che devono essere estremamente affidabili. Poiché i programmi assembler sono più lunghi e difficili da scrivere (e da leggere), il costo della scrittura del codice è più elevato e la verifica della correttezza dei programmi scritti in questo linguaggio è più complessa. Queste considerazioni portarono il Dipartimento della Difesa degli Stati Uniti, che investe molte risorse nello sviluppo dei sistemi embedded, a sviluppare Ada, un nuovo linguaggio ad alto livello pensato specificatamente per programmare dispositivi embedded.

A.2 Gli assemblatori

Etichetta esterna o **etichetta globale:** un'etichetta che si riferisce a un oggetto a cui si può fare riferimento da file diversi da quello in cui è definita.

Etichetta locale: un'etichetta che si riferisce a un oggetto che può essere utilizzato soltanto all'interno del file nel quale è definito.

Riferimento in avanti: un'etichetta che viene utilizzata prima di essere definita.

Un assemblatore traduce un file di istruzioni in linguaggio assembler in un file contenente istruzioni in linguaggio macchina e dati binari. Il processo di traduzione è costituito da due passi principali. Il primo consiste nell'individuare le locazioni di memoria identificate da etichette, in modo tale che la corrispondenza tra i nomi simbolici e gli indirizzi in memoria sia nota nel momento in cui le istruzioni vengono tradotte. Il secondo passo consiste nel considerare ogni istruzione assembler e tradurre in binario il codice operativo, il numero dei registri e le etichette, e inserendo la traduzione in un'unica stringa binaria (che rappresenta l'istruzione in linguaggio macchina). Come mostrato in Figura A.1.1, l'assemblatore produce un file in uscita chiamato **file oggetto**, che contiene le istruzioni in linguaggio macchina, i dati e le informazioni ausiliarie.

Un file oggetto di norma non può essere eseguito, poiché fa riferimento a procedure o dati contenuti in altri file. Un'**etichetta** si dice **esterna** (o anche **globale**) se all'oggetto etichettato si può fare riferimento da file diversi da quello nel quale è definito. Un'etichetta si dice **locale** se l'oggetto può essere utilizzato soltanto all'interno del file in cui è definito. Nella maggior parte degli assemblatori, le etichette sono considerate locali di default, a meno che non siano esplicitamente dichiarate etichette globali. Le procedure e le variabili globali richiedono etichette esterne, perché molti file di un programma possono fare riferimento a esse. Le **etichette locali** nascondono nomi che de-

vono rimanere invisibili agli altri moduli; per esempio, le funzioni statiche in C possono essere chiamate soltanto da altre funzioni presenti nello stesso file. Inoltre, i nomi generati dal compilatore, come il nome di un'istruzione all'inizio di un ciclo, sono considerati locali, in modo che il compilatore non debba generare nomi unici per ogni file.

Etichette locali e globali

ESEMPIO

Si consideri il programma in linguaggio assembler riportato in Figura A.1.4. La procedura ha l'etichetta esterna (globale) `main`. Esso contiene anche due etichette locali, `ciclo` e `str`, che sono visibili soltanto all'interno di questo file. Infine, la procedura contiene anche un riferimento non risolto all'etichetta esterna `printf`, che è la procedura (contenuta in libreria) per stampare valori. A quali etichette di questo frammento di codice si potrebbe far riferimento da un altro file?

SOLUZIONE

Solo le etichette globali sono visibili dall'esterno, per cui l'unica etichetta a cui si può far riferimento da un altro file è `main`. ■

Dato che l'assemblatore elabora ogni file di un programma individualmente e in modo indipendente, conosce solamente gli indirizzi delle etichette locali. L'assemblatore ha bisogno di un altro strumento, il linker, per unire i diversi file oggetto e le librerie in un unico file eseguibile, risolvendo le etichette esterne. L'assemblatore assiste il linker fornendogli l'elenco delle etichette e dei riferimenti non risolti.

Tuttavia, anche le etichette locali rappresentano una sfida interessante per un assemblatore: a differenza dei nomi utilizzati nella maggior parte dei linguaggi ad alto livello, le etichette in assembler possono essere utilizzate prima che vengano definite. Nell'esempio di Figura A.1.4 l'etichetta `str` viene utilizzata dall'istruzione `la` ancor prima di essere definita. Il fatto di poter effettuare **riferimenti in avanti**, come questo, implica che l'operazione di traduzione da parte dell'assemblatore avvenga in due passi successivi: prima vengono individuate tutte le etichette e dopo vengono prodotte le istruzioni in linguaggio macchina. Nell'esempio precedente, quando l'assemblatore incontra l'istruzione `la` non conosce la posizione della parola etichettata con `str` e nemmeno se `str` denoti un'istruzione o un dato.

Durante il primo stadio di traduzione, l'assemblatore legge le linee di un file assembler e spezza ciascuna linea nei suoi componenti. Questi, chiamati **lessemi**, sono parole singole, numeri e caratteri di interpunzione. Per esempio, la linea:

```
ble $t0, 100, ciclo
```

contiene sei lessemi: il codice operativo `ble`, il nome del registro `$t0`, una virgola, il numero `100`, una virgola e, infine, il simbolo `ciclo`.

Se una linea inizia con un'etichetta, l'assemblatore registra nella sua **tabella dei simboli** il nome dell'etichetta e l'indirizzo della parola di memoria occupata dall'istruzione. L'assemblatore, quindi, calcola quante parole di memoria occuperà l'istruzione presente alla linea corrente. Tenendo traccia dello spazio occupato dalle istruzioni, l'assemblatore è in grado di determinare dove verrà scritta l'istruzione seguente. Per calcolare la dimensione di un'istruzione a lunghezza variabile, come quelle del VAX, l'assemblatore deve esaminarla in dettaglio. Per istruzioni a lunghezza fissa, invece, come quelle del MIPS, basta un esame «superficiale». L'assemblatore esegue un calcolo analogo sulle istruzioni che riguardano i dati per valutare lo spazio necessario a contenere i dati. Quando l'assemblatore raggiunge la fine di un file assembler, la tabella dei simboli contiene la posizione di tutte le etichette definite nel file.

Tabella dei simboli: una tabella che associa il nome di ogni etichetta all'indirizzo della parola di memoria occupata dall'istruzione corrispondente.

L'assemblatore utilizza le informazioni contenute nella tabella dei simboli durante il secondo passaggio di traduzione del file, quando viene effettivamente generato il codice macchina. Poi esamina nuovamente tutte le linee di codice: se una linea contiene un'istruzione, l'assembler unisce la rappresentazione binaria del codice operativo e degli operandi (nomi dei registri o indirizzi di memoria) in un'unica istruzione binaria. Il processo è simile a quello utilizzato nel Paragrafo 2.5 del secondo capitolo. Le istruzioni e le parole dei dati che si riferiscono a simboli esterni, definiti in un altro file, non possono essere assemblati completamente (rimangono non risolti), perché l'indirizzo di questi simboli non è contenuto nella tabella dei simboli. Un assemblatore accetta la presenza di riferimenti non risolti, poiché l'etichetta corrispondente è probabilmente definita in un altro file.

Quadro d'insieme

L'assembler è un linguaggio di programmazione. La principale differenza rispetto ai linguaggi ad alto livello, come BASIC, Java e C, è che il linguaggio assembler fornisce soltanto pochi e semplici tipi di dato e di strutture di controllo del flusso. I programmi in assembler non specificano il tipo di dato contenuto in una variabile, ma è il programmatore che deve applicare le operazioni adatte ai dati considerati: per esempio, la somma intera invece che quella in virgola mobile. Inoltre, nel linguaggio assembler i programmi devono realizzare tutti i controlli del flusso mediante istruzioni di *go to*. Entrambi questi fattori rendono la programmazione in assembler su tutti i calcolatori, MIPS o x86, più difficile e più esposta a errori, rispetto alla scrittura del codice in un linguaggio ad alto livello.

Correzione a posteriori: un metodo di traduzione dal linguaggio assembler alle istruzioni in linguaggio macchina nel quale l'assemblatore genera una rappresentazione binaria di ogni istruzione, che in certi casi può essere incompleta, in un unico passaggio su tutto il file; l'assemblatore deve quindi tornare a sostituire il contenuto delle etichette precedentemente non definite.

Approfondimento. Quando la velocità dell'assemblatore diventa un aspetto rilevante, il processo di traduzione in linguaggio macchina del file assembler può essere effettuato in un unico passaggio, grazie a una tecnica chiamata **correzione a posteriori** (*backpatching*). Mentre elabora il file assembler, l'assemblatore genera una rappresentazione binaria, che in certi casi può essere incompleta, di ogni istruzione. Se l'istruzione fa riferimento a un'etichetta che non è ancora stata definita, l'assemblatore registra l'etichetta e l'istruzione in una tabella. Quando viene definita un'etichetta, l'assemblatore consulta questa tabella e trova tutte le istruzioni che contengono un riferimento in avanti a tale etichetta. L'assembler quindi torna indietro e corregge la rappresentazione binaria di tali istruzioni inserendo l'indirizzo corrispondente all'etichetta. La correzione a posteriori accelera il processo di traduzione in linguaggio macchina perché l'assemblatore deve leggere il file in ingresso soltanto una volta; tuttavia, essa richiede che l'assemblatore mantenga in memoria la rappresentazione binaria di tutto il programma per poter effettuare la correzione a posteriori delle istruzioni. Questo vincolo può limitare la dimensione dei programmi che possono essere assemblati; inoltre, la compilazione in un unico passaggio è ulteriormente complicata nelle macchine dotate di molti tipi di salto condizionato che caratterizzano diversi tipi di istruzioni. Quando un assemblatore trova un'etichetta non risolta in un'istruzione di salto condizionato, deve utilizzare il salto condizionato più lontano, altrimenti rischierebbe di dover tornare indietro e riposizionare molte istruzioni per fare spazio a un salto condizionato più lungo.

Formato dei file oggetto

Gli assemblatori producono file oggetto. Un file oggetto UNIX contiene le seguenti sei sezioni distinte (si veda la Figura A.2.1):

– L'**intestazione del file oggetto** (*object file header*), che descrive la posizione e le dimensioni delle altre sezioni del file.

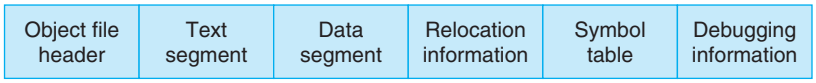


Figura A.2.1. File oggetto. Un assemblatore UNIX produce un file oggetto composto da sei sezioni distinte.

- Il **segmento testo**, che contiene il codice in linguaggio macchina delle procedure del file sorgente. Queste procedure potrebbero essere non eseguibili a causa di riferimenti non risolti.
- Il **segmento dati**, che contiene una rappresentazione binaria dei dati definiti nel file sorgente. Anche i dati potrebbero essere incompleti a causa di riferimenti non risolti a etichette definite in altri file.
- Le **informazioni di rilocazione**, che identificano le istruzioni e le parole di dati che dipendono da **indirizzi assoluti**. La posizione di queste istruzioni e dati deve essere modificata se parti del programma vengono spostate in memoria.
- La **tabella dei simboli**, che associa un indirizzo alle etichette esterne contenute nel file sorgente e contiene l'elenco dei riferimenti non risolti.
- Le **informazioni di debug**, che contengono una descrizione sintetica di come il programma è stato compilato, in modo che un debugger possa trovare gli indirizzi delle istruzioni che corrispondono alle linee del file sorgente e possa stampare le strutture dati in forma leggibile.

Segmento testo: il segmento di un file oggetto UNIX che contiene il codice in linguaggio macchina delle procedure definite nel file sorgente.

Segmento dati: il segmento di un file oggetto o eseguibile UNIX che contiene una rappresentazione binaria dei dati inizializzati, utilizzati dal programma.

Informazioni di rilocazione: il segmento di un file oggetto UNIX che identifica istruzioni e parole di dati che dipendono da indirizzi assoluti.

Indirizzo assoluto: l'indirizzo reale in memoria di una variabile o di una procedura.

L'assemblatore produce un file oggetto che contiene una rappresentazione binaria del programma e dei dati, oltre a informazioni ausiliarie destinate al linker per il collegamento delle diverse parti del programma.

Tali informazioni di rilocazione sono necessarie, perché l'assemblatore non sa quali locazioni di memoria saranno occupate da una procedura o da una parte dei dati dopo che queste saranno unite al resto del programma. Le procedure e i dati provenienti da un file vengono memorizzati in un blocco contiguo di memoria, ma l'assemblatore non sa dove questo blocco sarà posizionato. L'assemblatore, inoltre, comunica al linker alcune informazioni contenute nella tabella dei simboli. In particolare, l'assemblatore deve memorizzare quali simboli esterni sono definiti in un file e quali riferimenti non risolti si presentano.

Approfondimento. Per comodità, gli assemblatori presuppongono che ogni file parta dallo stesso indirizzo (per esempio dall'indirizzo 0) e assumono che il linker effettui la **rilocazione** del codice e dei dati nel momento in cui a questi vengono assegnate le locazioni di memoria. L'assemblatore produce le **informazioni di rilocazione** che descrivono ogni istruzione o parola di dati del file che facciano rifemento a un indirizzo assoluto. Sulle architetture MIPS, soltanto le istruzioni di chiamata a procedura e di lettura e scrittura in memoria fanno riferimento a indirizzi assoluti. Le istruzioni che utilizzano indirizzi relativi al Program Counter, come i salti condizionati, non hanno bisogno di essere rilocate.

Altre funzionalità degli assemblatori

Gli assemblatori mettono a disposizione diverse funzionalità che aiutano a rendere i programmi assembler più compatti e semplici da scrivere, senza alterare il linguaggio assembler in modo significativo. Per esempio, le **direttive di definizione dei dati** permettono a un programmatore di descrivere i dati in modo più conciso e naturale rispetto alla rappresentazione binaria.

In Figura A.1.4, la direttiva

```
.asciiz "La somma da 0 a 100 è %d\n"
```

salva in memoria i caratteri della stringa. Confrontate questa linea di codice con l'alternativa, rappresentata dallo scrivere il codice ASCII associato a ciascun carattere (la Figura 2.15 del Capitolo 2 descrive la codifica ASCII dei caratteri):

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

La direttiva `.asciiz` è più semplice da leggere perché i caratteri sono rappresentati come lettere anziché numeri binari. Un assemblatore può tradurre i caratteri nella loro rappresentazione binaria molto più velocemente e accuratamente di quanto possa fare una persona. Le direttive di definizione dei dati specificano i dati in una forma leggibile, che viene poi tradotta in binario dall'assemblatore. Altre rappresentazioni dei dati sono descritte nel Paragrafo A.10.

Direttiva di stringa

ESEMPIO

Si determini la sequenza di byte prodotta dalla seguente direttiva:

```
.asciiz "The quick brown fox jumps over the lazy dog"
```

SOLUZIONE

```
.byte 84, 104, 101, 32, 113, 117, 105, 99
.byte 102, 32, 98, 114, 111, 119, 110, 32
.byte 102, 111, 120, 32, 106, 117, 109, 112
.byte 115, 32, 111, 118, 101, 114, 32, 116
.byte 104, 101, 32, 108, 97, 122, 121, 32
.byte 100, 111, 103, 0
```

La **macro** è una funzione di ricerca e sostituzione di stringhe di caratteri che fornisce un semplice meccanismo per gestire una sequenza di istruzioni utilizzata di frequente. Invece di scrivere ripetutamente le stesse istruzioni ogni volta che vengono utilizzate, il programmatore invoca la macro e l'assemblatore sostituisce la chiamata con la corrispondente sequenza di istruzioni. Le macro, come le procedure, permettono al programmatore di creare e dare un nome a una nuova astrazione associata a un'operazione frequente. A differenza delle procedure, però, le macro non implicano una chiamata e un ritorno da una procedura durante l'esecuzione di un programma, perché una chiamata a macro viene sostituita dal corpo della macro stessa quando il programma viene assemblato. Dopo tale sostituzione, il codice assembler risultante è indistinguibile dal programma equivalente scritto senza macro.

Le macro

ESEMPIO

Si supponga che un programmatore abbia bisogno di stampare molti numeri. La procedura di libreria `printf` accetta come argomenti una stringa di formato e uno o più numeri da stampare. Un programmatore, per esempio, potrebbe stampare il numero intero contenuto nel registro `$7` con le istruzioni seguenti:

```
          .data
int_str:  .asciiz "%d"
          .text
          la $a0, int_str   # Carica l'indirizzo della stringa
                            # come primo argomento
          mov    $a1, $7    # Carica il numero come
                            # secondo argomento
          jal    printf     # Chiama la procedura printf
```

La direttiva `.data` indica all'assemblatore di memorizzare la stringa nel segmento dati del programma e la direttiva `.text` indica di memorizzare le istruzioni nel suo segmento testo.

Tuttavia, stampare molti numeri in questa maniera produce un programma prolisso che risulta di difficile comprensione. Un'alternativa è quella di introdurre una macro, `print_int`, che stampi un numero intero:

```
          .data
int_str:  .asciiz "%d"
          .text
          .macro print_int($arg)
          la      $a0, int_str    # Carica l'indirizzo
                                  # della stringa
                                  # come primo argomento
          mov     $a1, $arg       # Carica il parametro
                                  # della macro
                                  # ($arg) come secondo
                                  # argomento
          jal     printf          # Chiama la procedura printf
          .end_macro
print_int($7)
```

La macro ha un **parametro formale**, `$arg`, ossia il suo argomento. Quando la macro viene espansa, l'argomento presente nella chiamata viene sostituito al parametro formale in tutto il corpo della macro. L'assemblatore sostituisce quindi la chiamata alla macro con il corpo della macro stessa. Nella prima chiamata a `print_int`, l'argomento è `$7`, per cui la macro viene sostituita dal seguente codice:

Parametro formale: una variabile che rappresenta l'argomento passato a una procedura o a una macro; viene sostituita dall'argomento nel momento in cui la macro viene espansa.

```
  la      $a0, int_str
  mov     $a1, $7
  jal     printf
```

Nella seconda chiamata, `print_int($t0)`, l'argomento è `$t0`, per cui la macro diventa:

```
  la      $a0, int_str
  mov     $a1, $t0
  jal     printf
```

Con quale frammento di codice potrà essere sostituita la chiamata alla funzione `print_int($a0)`?

SOLUZIONE

```
  la      $a0, int_str
  mov     $a1, $a0
  jal     printf
```

■

Questo esempio mette in evidenza una complicazione dovuta alle macro. Il programmatore che utilizza macro deve essere consapevole che `print_int` utilizza il registro `$a0` e quindi non può stampare correttamente il contenuto di questo registro.

Interfaccia hardware/software

Alcuni assemblatori implementano anche **pseudoistruzioni**; queste sono istruzioni messe a disposizione dall'assemblatore ma non implementate in hardware. Il Capitolo 2 riporta molti esempi di come l'assemblatore MIPS «traduca» le

pseudoistruzioni e le modalità di indirizzamento disponibili a partire dal semplice insieme di istruzioni hardware del MIPS. Per esempio, il Paragrafo 2.7 del secondo capitolo descrive il modo in cui l'assemblatore implementa l'istruzione `blt` a partire da due altre istruzioni: `slt` e `bne`. Estendendo l'insieme delle istruzioni, l'assemblatore MIPS rende la programmazione in linguaggio assembler più semplice senza aumentare la complessità dell'hardware. Molte pseudoistruzioni potrebbero venire anche simulate mediante macro, ma l'assemblatore MIPS riesce a generare un codice migliore per queste istruzioni perché utilizza un registro dedicato (`$at`) ed è in grado di ottimizzare il codice generato.

Approfondimento. Gli assemblatori assemblano parti del codice in **modo condizionato**, il che permette al programmatore di includere o escludere gruppi di istruzioni quando un programma viene assemblato. Questa funzionalità è particolarmente utile nel caso di programmi di cui esistono parecchie versioni che differiscono di poco tra loro. Invece di tenere assemblati questi programmi in file separati, cosa che complica notevolmente l'eliminazione dei bachi nelle parti di codice in comune, i programmatori generalmente fondono le diverse versioni in un unico file. Il codice di una particolare versione viene assemblato in modo condizionato, cosicché possa essere escluso quando vengono assemblate altre versioni del programma.

Se le macro e l'assemblaggio condizionato sono utili, perché gli assemblatori dei sistemi UNIX li supportano solo raramente? Un motivo è che la maggior parte dei programmatori di tali sistemi scrive programmi in linguaggi ad alto livello, come il C. La gran parte del codice assembler è prodotto dai compilatori per i quali risulta più comodo ripetere il codice piuttosto che definire delle macro. Un'altra ragione è che UNIX mette a disposizione altri strumenti, come il cpp, il preprocessore C, o l'm4, un generico processore di macro, che sono in grado di fornire macro e assemblaggio condizionato ai programmi scritti in assembler.

A.3 I linker

Compilazione separata: la suddivisione di un programma in molti file, ciascuno dei quali può essere compilato senza sapere cosa è contenuto negli altri file.

La **compilazione separata** consente di suddividere un programma in più parti memorizzate in file differenti. Ogni file contiene un insieme di procedure e di strutture dati logicamente correlate che costituiscono il **modulo** di un programma più grande. Un file può essere compilato e assemblato indipendentemente dagli altri file; in questo modo, le modifiche a un modulo non richiedono la ricompilazione dell'intero programma. Come già discusso in precedenza, la compilazione separata necessita di una fase aggiuntiva di collegamento (*linking*) per unire i file oggetto provenienti da moduli separati e sistemare i riferimenti non risolti.

Lo strumento utilizzato per unire questi file è il **linker** (si veda la Figura A.3.1). Esso assolve i seguenti tre compiti:

- cercare tra le librerie del programma per trovare le relative procedure di libreria utilizzate;
- determinare le locazioni di memoria che verranno occupate dal codice di ogni modulo e rilocare le istruzioni correggendo i riferimenti assoluti;
- risolvere i riferimenti tra file diversi.

Uno dei primi compiti del linker è assicurare che un programma non contenga etichette non definite. Il linker mette in corrispondenza i simboli esterni con i riferimenti non risolti dei file del programma. Un simbolo esterno di un file risolve un riferimento in un altro file se entrambi fanno riferimento a un'etichetta con lo stesso nome; se si trovano riferimenti che non hanno una corrispondenza, significa che ci sono simboli che vengono utilizzati senza mai essere definiti in alcun punto del programma.

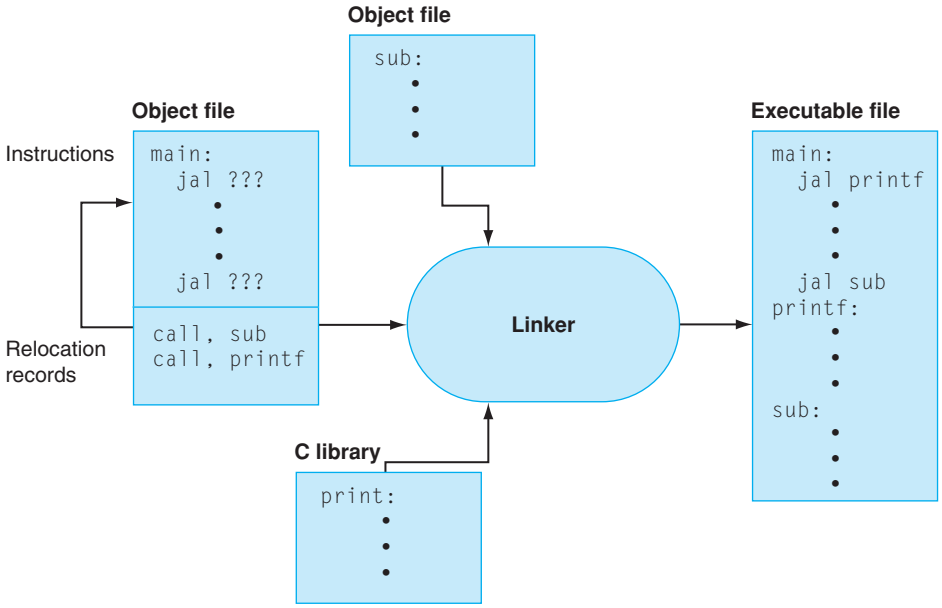


Figura A.3.1. Il linker cerca all'interno di un insieme di file oggetto e di librerie di programmi le procedure non locali utilizzate da un programma, unisce i file oggetto in un singolo file eseguibile e risolve i riferimenti tra procedure contenute in file differenti.

La presenza di riferimenti non risolti in questo stadio del processo di linking non significa necessariamente che il programmatore abbia commesso un errore. Il programma potrebbe fare riferimento a una procedura di libreria il cui codice non era presente nei file oggetto passati al linker. Dopo aver trovato le corrispondenze dei simboli del programma, il linker cerca nelle librerie di sistema le procedure e le strutture dati a cui il programma fa riferimento. Le librerie di base contengono procedure che scrivono e leggono i dati, allocano e disallocano la memoria e svolgono operazioni numeriche. Altre librerie contengono procedure per accedere a basi di dati o gestiscono le finestre del terminale video. Un programma che faccia riferimento a un simbolo non risolto che non si trova in alcuna libreria contiene errori e il linking non può essere portato a termine. Quando il programma utilizza una procedura di libreria, il linker estrae il codice della procedura dalla libreria e lo incorpora nel segmento testo del programma. Quest'ultima procedura, a sua volta, potrebbe dipendere da altre procedure di libreria, quindi il linker continua a prelevare altre procedure di libreria fino a quando tutti i riferimenti esterni risultino risolti, oppure fino a quando non venga trovata una procedura adeguata al simbolo.

Se tutti i riferimenti esterni sono stati risolti, il linker passa a determinare le locazioni di memoria che ogni modulo occuperà. Dato che i file sono stati assemblati separatamente, l'assemblatore non poteva sapere dove sarebbero state posizionate le istruzioni e i dati di un modulo rispetto agli altri moduli. Quando il linker posiziona un modulo in memoria tutti i riferimenti assoluti devono essere **rilocati** nella loro posizione effettiva. Poiché il linker dispone delle informazioni di rilocazione che identificano tutti i riferimenti da rilocare, è in grado di identificare ed effettuare la correzione a posteriori dei riferimenti in modo efficiente.

Il linker produce un file eseguibile, cioè un file che può essere eseguito su un calcolatore. Tipicamente, questo file ha lo stesso formato di un file oggetto, con la differenza che non contiene riferimenti non risolti né informazioni di rilocazione.

A.4 Caricamento

Un programma il cui linking si conclude senza errori è pronto per essere eseguito. Esso viene scritto su un file salvato sul dispositivo di memoria di mas-

sa, per esempio su disco. Nei sistemi UNIX, è il kernel del sistema operativo a caricare un programma nella memoria principale e a lanciarne l'esecuzione. Per far partire un programma, il sistema operativo effettua le seguenti operazioni:

1. Leggere l'intestazione del file eseguibile per determinare le dimensioni dei segmenti testo e dati.
2. Creare un nuovo spazio di indirizzamento per il programma. Questo spazio è abbastanza grande da contenere i segmenti di testo e dei dati, nonché un segmento per lo stack (si veda il Paragrafo A.5).
3. Copiare le istruzioni e i dati del file eseguibile in memoria all'interno del nuovo spazio di indirizzamento.
4. Copiare nello stack gli argomenti passati al programma.
5. Inizializzare i registri dell'architettura. In generale, il contenuto della maggior parte dei registri viene cancellato, tranne quello del registro stack pointer al quale viene assegnato l'indirizzo della prima locazione libera dello stack (si veda il Paragrafo A.5).
6. Saltare a una procedura di avvio che copia gli argomenti del programma dallo stack ai registri, per poi chiamare la procedura `main` del programma. Quando la procedura `main` termina, la procedura di avvio conclude il programma attraverso la chiamata di sistema `exit`.

A.5 Utilizzo della memoria

Dati statici: la porzione di memoria che contiene dati la cui dimensione è nota al compilatore e il cui tempo di vita è il tempo totale di esecuzione del programma.

I prossimi paragrafi descrivono più in dettaglio l'architettura MIPS presentata nel testo. I capitoli precedenti erano focalizzati soprattutto sull'hardware e sulla sua relazione con il software a basso livello; questi paragrafi, invece, si focalizzano principalmente su come i programmatori in linguaggio assembler utilizzano l'hardware MIPS. In particolare, viene descritto un insieme di convenzioni seguite in molti sistemi MIPS; nella maggior parte dei casi l'hardware non impone tali convenzioni, ma esse sono il frutto della scelta dei programmatori di seguire le stesse regole, affinché parti di software scritte da persone diverse possano convivere utilizzando l'hardware MIPS in modo efficace.

I sistemi basati su processori MIPS tipicamente dividono la memoria in tre parti (si veda la Figura A.5.1). La prima, vicina al limite inferiore dello spazio di indirizzamento, a partire dall'indirizzo $40\,0000_{esa}$, è il **segmento testo**, che contiene le istruzioni del programma.

La seconda parte, collocata immediatamente sopra il segmento testo, è il **segmento dati**, che viene ulteriormente suddiviso in due parti. La sezione dei **dati statici**, a partire dall'indirizzo $1000\,0000_{esa}$, contiene oggetti di dimensione nota al compilatore e il cui tempo di vita (l'intervallo durante il quale un programma può accedervi) è il periodo di esecuzione dell'intero programma.

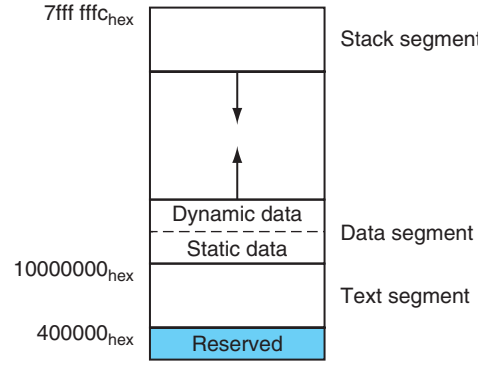


Figura A.5.1. Organizzazione della memoria.

Per esempio, in C le variabili globali sono allocate staticamente ed è possibile fare riferimento a esse in qualunque momento durante l'esecuzione di un programma. Il linker si occupa sia di assegnare agli oggetti statici delle locazioni di memoria nel segmento dati sia di risolvere i riferimenti a questi oggetti.

Immediatamente al di sopra dei dati statici si trovano i **dati dinamici**. Questi, come suggerisce il nome, vengono allocati dal programma durante la sua esecuzione. La procedura di libreria `malloc`, utilizzata nei programmi C, trova e restituisce un nuovo blocco di memoria. Dato che un compilatore non può prevedere quanta memoria allocherà un programma, il sistema operativo espande l'area dei dati dinamici in modo da soddisfare le esigenze. La procedura `malloc` espande l'area dinamica come indicato dalle frecce della Figura A.5.1, utilizzando la chiamata di sistema `sbrk`, che induce il sistema operativo ad aggiungere altre pagine allo spazio di indirizzamento virtuale del programma (si veda il Paragrafo 5.4), immediatamente al di sopra del segmento dei dati dinamici attuale.

La terza parte, il **segmento di stack** del programma, si trova nella parte superiore dello spazio di indirizzamento virtuale, a partire dall'indirizzo $7FFF\,FFFF_{esa}$. Come per i dati dinamici, la dimensione massima dello stack di un programma non è nota a priori. Ogni qualvolta il programma introduce valori nello stack, il sistema operativo espande il segmento di stack verso il basso, cioè verso il segmento dati.

Questa suddivisione della memoria non è l'unica possibile; tuttavia, essa presenta due caratteristiche importanti: i due segmenti espandibili dinamicamente sono il più possibile distanti tra loro e quindi possono crescere fino a utilizzare l'intero spazio di indirizzamento del programma.

Segmento di stack: la porzione di memoria utilizzata da un programma per contenere le strutture delle chiamate a procedura.

Interfaccia hardware/software

Poiché il segmento dati inizia molto al di sopra del programma, all'indirizzo $1000\,0000_{esa}$, le istruzioni di accesso alla memoria (load e store) non possono fare riferimento direttamente agli oggetti in esso contenuti, avendo campi di 16 bit per lo spiazzamento (si veda il Paragrafo 2.5). Per esempio, per caricare la parola situata nel segmento dati all'indirizzo $1001\,0020_{esa}$ nel registro `$v0` sono necessarie due istruzioni:

```
lui $s0, 0x1001          # 0x1001 significa 1001 in base 16
lw  $v0, 0x0020($s0)     # 0x10010000 + 0x0020 = 0x10010020
```

Ricordiamo che il prefisso `0x` davanti a un numero denota un numero esadecimale. Per esempio, `0x8000` corrisponde a 8000_{esa}, cioè $32\,768_{dec}$.

Per evitare di ripetere l'istruzione `lui` ogni volta che si deve utilizzare una load o una store, i sistemi MIPS di solito utilizzano un registro (`$gp`) come **puntatore globale** (*global pointer*) al segmento dei dati statici. Questo registro conterrà quindi l'indirizzo $1000\,8000_{esa}$, così che le istruzioni di load e store possano utilizzare il loro campo spiazzamento di 16 bit dotato di segno per accedere ai primi 64 kB del segmento dei dati statici. Grazie al puntatore globale possiamo riscrivere l'esempio precedente utilizzando una singola istruzione:

```
lw  $v0, 0x8020($gp)
```

Ovviamente, il registro global pointer rende più veloce l'indirizzamento delle locazioni di memoria comprese tra $1000\,0000_{esa}$ e $1001\,0000_{esa}$ rispetto a quello delle altre locazioni dell'area dati. Il compilatore MIPS di solito memorizza in quest'area le **variabili globali**, perché sono memorizzate in locazioni fisse e sono più adatte a questa modalità di indirizzamento di altri dati globali (come i vettori).

A.6 Convenzioni di chiamata a procedura

Convenzione di utilizzo dei registri: detta anche **convenzione di chiamata a procedura**, è un protocollo software che regola l'utilizzo dei registri da parte delle procedure.

Registro salvato dal chiamante: un registro salvato dalla procedura chiamante.

Registro salvato dal chiamato: un registro salvato dalla procedura che viene chiamata.

Le convenzioni che regolano l'utilizzo dei registri sono necessarie quando le procedure di un programma vengono compilate separatamente. Per compilare una particolare procedura, il compilatore deve sapere quali registri può utilizzare e quali sono riservati ad altre procedure. Le regole di utilizzo dei registri sono dette **convenzioni di utilizzo dei registri**, oppure convenzioni **di chiamata a procedura**. Queste sono in maggioranza convenzioni software e non regole imposte dall'hardware. Tuttavia, la maggior parte dei compilatori e dei programmatori cerca sempre di rispettare queste convenzioni perché la loro violazione può causare errori difficili da individuare.

La convenzione di chiamata a procedura descritta in questa sezione è quella utilizzata dal compilatore `gcc`. Il compilatore nativo MIPS impiega una convenzione più complessa che risulta leggermente più veloce.

La CPU MIPS contiene 32 registri di utilizzo generale numerati da 0 a 31. Il registro `$0` contiene sempre il valore 0.

- I registri `$at` (1), `$k0` (26) e `$k1` (27) sono riservati all'assemblatore e al sistema operativo, e non dovrebbero essere utilizzati dai programmi utente o dai compilatori.
- I registri `$a0`-`$a3` (4-7) vengono utilizzati per passare i primi quattro argomenti alle procedure (gli argomenti rimanenti vengono passati mediante lo stack), mentre i registri `$v0` e `$v1` (2 e 3) vengono utilizzati per restituire i risultati calcolati dalle funzioni.
- I registri `$t0`-`$t9` (8-15, 24, 25) sono **registri salvati dal chiamante** e vengono utilizzati per contenere valori temporanei che non hanno bisogno di essere salvati dalla procedura (si veda il Paragrafo 2.8).
- I registri `$s0`-`$s7` (16-23) sono **registri salvati dal chiamato** e contengono valori più duraturi, che devono essere preservati durante la chiamata e l'esecuzione delle procedure.
- Il registro `$gp` (28) è il puntatore globale che punta nel mezzo di un blocco di 64 K locazioni di memoria del segmento dei dati statici.
- Il registro `$sp` (29) è lo stack pointer (il puntatore allo stack), il quale punta all'ultima locazione dello stack. Il registro `$fp` (30) è il **frame pointer** (puntatore al frame della procedura) L'istruzione `jal` scrive nel registro `$ra` (31) l'indirizzo di ritorno da una procedura. L'utilizzo di questi ultimi due registri verrà spiegato nel prossimo paragrafo.

I nomi di questi registri e la loro abbreviazione, per esempio `$sp` per il puntatore a stack, denotano come devono essere utilizzati i registri secondo le convenzioni di chiamata a procedura. Nel descrivere queste convenzioni utilizzeremo i nomi anziché i numeri dei registri. In Figura A.6.1 vengono elencati i registri, accompagnati da una descrizione del loro uso.

Le chiamate a procedura

Questo paragrafo descrive le operazioni che vengono effettuate quando una procedura, detta **chiamante**, chiama un'altra procedura, detta **chiamata**. I programmatori che sviluppano codice in un linguaggio ad alto livello, come il C o il Pascal, non sanno in che modo una procedura chiami un'altra procedura, perché i dettagli di basso livello della chiamata vengono definiti dal compilatore stesso. I programmatori in linguaggio assembler, invece, devono implementare esplicitamente le operazioni di chiamata a procedura e di ritorno alla procedura chiamante.

Nome del registro	Numero	Utilizzo
$zero	0	costante 0
$at	1	riservato all'assemblatore
$v0	2	calcolo di espressioni e risultati di una funzione
$v1	3	calcolo di espressioni e risultati di una funzione
$a0	4	argomento 1
$a1	5	argomento 2
$a2	6	argomento 3
$a3	7	argomento 4
$t0	8	temporaneo (non preservato dalle procedure)
$t1	9	temporaneo (non preservato dalle procedure)
$t2	10	temporaneo (non preservato dalle procedure)
$t3	11	temporaneo (non preservato dalle procedure)
$t4	12	temporaneo (non preservato dalle procedure)
$t5	13	temporaneo (non preservato dalle procedure)
$t6	14	temporaneo (non preservato dalle procedure)
$t7	15	temporaneo (non preservato dalle procedure)
$s0	16	temporaneo salvato (preservato dalle procedure)
$s1	17	temporaneo salvato (preservato dalle procedure)
$s2	18	temporaneo salvato (preservato dalle procedure)
$s3	19	temporaneo salvato (preservato dalle procedure)
$s4	20	temporaneo salvato (preservato dalle procedure)
$s5	21	temporaneo salvato (preservato dalle procedure)
$s6	22	temporaneo salvato (preservato dalle procedure)
$s7	23	temporaneo salvato (preservato dalle procedure)
$t8	24	temporaneo (non preservato dalle procedure)
$t9	25	temporaneo (non preservato dalle procedure)
$k0	26	riservato per il kernel del sistema operativo
$k1	27	riservato per il kernel del sistema operativo
$gp	28	puntatore all'area globale
$sp	29	stack pointer (puntatore allo stack)
$fp	30	frame pointer (puntatore al frame della procedura)
$ra	31	indirizzo di ritorno (utilizzato dalla chiamata a procedura)

Figura A.6.1. I registri MIPS e le loro convenzioni di utilizzo.

La maggior parte delle operazioni associate a una chiamata viene effettuata su un blocco di memoria chiamato **frame di chiamata a procedura**. Questo blocco di memoria viene utilizzato per diversi scopi:

Frame di chiamata a procedura: un blocco di memoria che viene utilizzato per contenere i valori passati a una procedura sotto forma di argomenti, per salvare i registri che possono essere modificati dalla procedura ma che la procedura chiamante non vuole trovare modificati e per fornire spazio alle variabili locali della procedura.

- contenere i valori passati a una procedura sotto forma di argomenti;
- salvare i registri che possono essere modificati dalla procedura, ma che la procedura chiamante non vuole che siano modificati;
- fornire spazio alle variabili locali della procedura.

Nella maggior parte dei linguaggi di programmazione, la chiamata e il ritorno da procedura seguono rigorosamente un ordinamento del tipo «ultima a essere chiamata, prima a essere terminata» (LIFO, *last-in, first-out*), per cui il blocco di memoria può essere allocato e disallocato nello stack; per questo motivo tali blocchi di memoria vengono talvolta chiamati frame di stack (*stack frames*).

La Figura A.6.2 mostra un tipico frame di stack; esso è costituito dal blocco di memoria compreso tra la cella di memoria puntata dal frame pointer (`$fp`), che punta alla prima parola del frame, e quella puntata dallo stack pointer (`$sp`), che punta all'ultima parola del frame. Lo stack cresce verso il basso partendo dagli indirizzi più alti della memoria, quindi il frame pointer punta

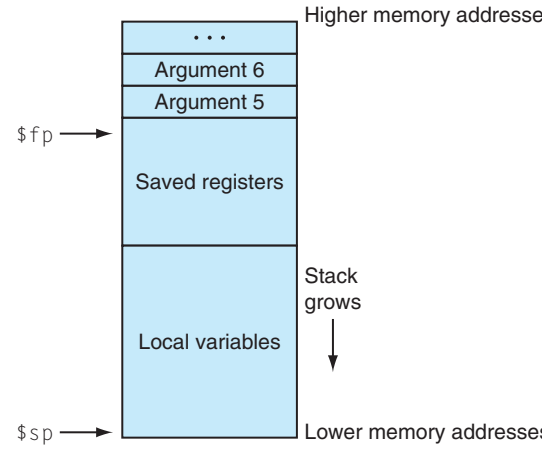


Figura A.6.2. Organizzazione di un frame di stack. Il frame pointer (`$fp`) punta alla prima parola del frame di stack della procedura correntemente in esecuzione. Lo stack pointer (`$sp`) punta all'ultima parola del frame. I primi quattro argomenti vengono passati attraverso i registri, per cui il quinto argomento è il primo che viene memorizzato nello stack.

a una cella di memoria al di sopra dello stack pointer. La procedura in esecuzione utilizza il frame pointer per accedere velocemente ai valori contenuti nel proprio frame di stack. Per esempio, un argomento presente nel frame di stack può essere caricato nel registro `$v0` con l'istruzione:

```
lw   $v0, 0($fp)
```

Un frame di stack può essere costruito in molti modi diversi: il chiamante e il chiamato devono però essere d'accordo sulla sequenza delle operazioni da eseguire. Le operazioni riportate di seguito descrivono le convenzioni di chiamata utilizzate nella maggior parte delle macchine MIPS. Queste convenzioni entrano in gioco tre volte durante una chiamata a procedura: immediatamente prima che il chiamante invochi la procedura chiamata, quando la procedura chiamata inizia la sua esecuzione e subito prima che la procedura chiamata ritorni alla procedura chiamante. Nel primo passo, la procedura chiamante pone gli argomenti da passare alla procedura chiamata nelle posizioni convenute e poi invoca la procedura. I passi sono i seguenti:

1. Passaggio degli argomenti. Per convenzione, i primi quattro argomenti vengono passati nei registri `$a0–$a3`. Eventuali altri argomenti vengono inseriti nello stack e appaiono all'inizio del frame di stack della procedura chiamata.
2. Salvataggio dei registri temporanei da parte del chiamante. La procedura chiamata può utilizzare i registri (`$a0-$a3` e `$t0-$t9`) senza dovere salvare preventivamente il loro valore. Se il chiamante deve utilizzare il contenuto di uno di questi registri dopo la chiamata, deve salvarne il contenuto prima della chiamata stessa.
3. Esecuzione dell'istruzione `jal` (si veda il Paragrafo 2.8), la quale salta alla prima istruzione della procedura chiamata e salva l'indirizzo di ritorno nel registro `$ra`.

Prima che inizi l'esecuzione della procedura chiamata, questa deve svolgere le seguenti operazioni per organizzare il proprio frame di stack:

1. Allocare la memoria per il frame di stack. Ciò viene ottenuto sottraendo la dimensione del frame dallo stack pointer.
2. Salvare i registri da preservare nel frame di stack. Una procedura chiamata deve salvare il contenuto dei registri `$s0–$s7`, `$fp` e `$ra` prima di modificarli, dato che la procedura chiamante si aspetta di ritrovare questi registri inalterati dopo il ritorno dalla procedura chiamata. Il registro `$fp` viene salvato da ogni procedura che allochi un nuovo frame di stack; il registro `$ra`, invece, deve essere salvato soltanto se la procedura chiamata effettua

a sua volta un'altra chiamata. I registri `$s0`–`$s7`, se vengono utilizzati dalla procedura chiamata, devono essere anch'essi salvati.
3. Determinare il contenuto del frame pointer. Ciò si ottiene sommando il valore della dimensione del frame di stack diminuito di 4 allo stack pointer, `$sp`, e memorizzando il risultato in `$fp`.

Infine, la procedura chiamata ritorna a quella chiamante effettuando le seguenti operazioni:

1. Scrivere il risultato del registro `$v0` se il chiamato è una procedura che restituisce un risultato.
2. Ripristinare tutti i registri da preservare che erano stati salvati all'inizio della procedura.
3. Rimuovere il frame di stack aggiungendo la dimensione del frame a `$sp`.
4. Far tornare l'esecuzione al chiamante. Ciò si ottiene saltando all'indirizzo contenuto nel registro `$ra`.

Interfaccia hardware/software

La convenzione sull'utilizzo dei registri del MIPS prevede che alcuni di essi debbano essere salvati dal chiamato e altri dal chiamante, perché questa modalità si rivela utile in differenti situazioni. I registri salvati dal chiamato sono i registri da preservare; essi sono più adatti a contenere valori che devono essere disponibili a lungo, come le variabili di un programma utente: ciascuno di questi registri viene salvato durante una chiamata a procedura soltanto se ci si aspetta che la procedura chiamata utilizzi tale registro. Viceversa, i registri salvati dal chiamante sono più adatti a contenere variabili temporanee caratterizzate da vita breve, come lo spiazzamento utilizzato in un calcolo di indirizzo. All'interno della procedura, anche il chiamato può utilizzare questi registri per salvare variabili temporanee.

Approfondimento. Un linguaggio di programmazione che non permette **procedure ricorsive** (procedure che chiamano se stesse, direttamente o indirettamente, attraverso una catena di chiamate) non ha bisogno di allocare i frame di procedura nello stack. In un linguaggio non ricorsivo, ogni frame di procedura può essere allocato staticamente, poiché può essere attiva soltanto una procedura alla volta. Le prime versioni del Fortran non ammettevano la ricorsione perché i frame allocati staticamente producevano un codice più veloce su alcune macchine ormai obsolete. Invece, su architetture di tipo load-store come il MIPS, i frame di stack possono essere altrettanto veloci, perché il registro frame pointer punta direttamente al frame di stack attivo, il che permette di utilizzare una singola istruzione di load o store per accedere alle variabili contenute nel frame. Inoltre, la ricorsione è una tecnica di programmazione molto utile.

Procedure ricorsive: procedure che chiamano se stesse, direttamente o indirettamente, attraverso una catena di chiamate.

Esempio di chiamata a procedura

Come esempio si consideri la seguente procedura C:

```
main ()
{
  printf("Il fattoriale di 10 è %d\n", fatt(10));
}

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

che calcola e stampa 10!, cioè il fattoriale di 10, 10! = 10 × 9 × … × 1. `fatt` è una procedura ricorsiva che calcola *n*! moltiplicando *n* per (*n*–1)!. Il codice assembler di questa procedura illustra come i programmi gestiscono i frame di stack.

Prima di iniziare, la procedura `main` crea il proprio frame di stack e salva i due registri da preservare che verranno utilizzati: `$fp` e `$ra`. Il frame è più grande di quanto necessario per contenere questi due registri, poiché la convenzione di chiamata richiede che la dimensione minima del frame di stack sia di 24 byte: il frame minimo può contenere i quattro registri argomento (`$a0–$a3`) e l'indirizzo di ritorno `$ra`, e la sua dimensione viene arrotondata per eccesso al primo multiplo della parola doppia (8 byte), in questo caso a 24 byte. All'interno dei 24 byte, `main` salva anche il registro `$fp`.

```
        .text
        .globl main
main:
        subu    $sp, $sp, 32    # Il frame di stack è di 32 byte
        sw      $ra, 20($sp)    # Salva l'indirizzo di ritorno
        sw      $fp, 16($sp)    # Salva il vecchio frame pointer
        addiu   $fp, $sp, 28    # Imposta il frame pointer
```

La procedura `main` chiama quindi la procedura che calcola il fattoriale e le passa il singolo argomento 10. Dopo il ritorno da `fatt`, `main` chiama la procedura di libreria `printf` e le passa sia una stringa di formato che il risultato restituito da `fatt`:

```
    li      $a0, 10         # Mette l'argomento (10) in $a0
    jal     fatt            # Chiama la funzione fattoriale

    la      $a0, $LC        # Metti la stringa di formato in $a0
    move    $a1, $v0        # Copia il risultato di fatt in $a1
    jal     printf          # Chiama la funzione di stampa
```

Infine, dopo aver stampato il fattoriale, `main` restituisce il controllo al programma che lo ha chiamato. Prima, però, deve ripristinare i registri che aveva salvato e rimuovere il frame di stack che aveva creato:

```
    lw      $ra, 20($sp)    # Ripristina l'indirizzo di ritorno
    lw      $fp, 16($sp)    # Ripristina il frame pointer
    addiu   $sp, $sp, 32    # Rimuovi il frame di stack
    jr      $ra             # Ritorna al chiamante

    .rdata
$LC:
    .ascii  "Il fattoriale di 10 è %d\n\000"
```

La procedura fattoriale ha una struttura simile alla procedura `main`. Per prima cosa, crea un frame di stack e salva i registri tempornei che userà in seguito. Oltre a salvare `$ra` e `$fp`, `fatt` salva anche il suo argomento (`$a0`) che verrà utilizzato per la chiamata ricorsiva:

```
    .text
fatt:
    subu    $sp, $sp, 32    # Il frame di stack è lungo 24 byte
    sw      $ra, 20($sp)    # Salva l'indirizzo di ritorno
    sw      $fp, 16($sp)    # Salva il frame pointer
    addiu   $fp, $sp, 28    # Imposta il frame pointer
    sw      $a0, 0($fp)     # Salva l'argomento (n)
```

Il cuore della procedura `fatt` effettua il calcolo riportato nel programma C. Controlla se l'argomento è maggiore di 0, se non lo è la procedura restituisce

il valore 1. Se l'argomento è maggiore di 0 la procedura chiama se stessa ricorsivamente per calcolare `fatt(n-1)` e moltiplica tale valore per *n*:

```
   lw      $v0, 0($fp)    # Carica n
   bgtz    $v0, $L2       # Salta se n > 0
   li      $v0, 1         # Restituisci 1
   jr      $L1            # Salta all'etichetta $L1 per ritornare
                          # al chiamante.

$L2:
   lw      $v1, 0($fp)    # Carica n
   subu    $v0, $v1, 1    # Calcola n - 1
   move    $a0, $v0       # Copia questo valore in $a0
   jal     fatt           # Chiama la funzione fattoriale

   lw      $v1, 0($fp)    # Carica n
   mul     $v0, $v0, $v1  # Calcola fatt(n-1) * n
```

Infine, la procedura fattoriale ripristina i registri da preservare, salvati all'atto della chiamata, e restituisce il risultato nel registro `$v0`:

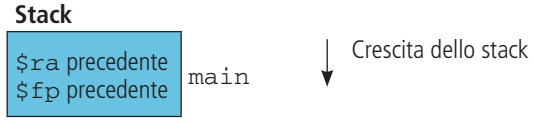
```
$L1:                      # Il risultato è in $v0
   lw      $ra, 20($sp)   # Ripristina $ra
   lw      $fp, 16($sp)   # Ripristina $fp
   addiu   $sp, $sp, 32   # Rimuovi il frame dallo stack
   jr      $ra            # Ritorna alla procedura chiamante
```

Lo stack nelle procedure ricorsive

ESEMPIO

La figura A.6.3 mostra la situazione dello stack al momento della chiamata di `fatt(7)`. La procedura `main` viene eseguita per prima, per cui il suo frame è il più profondo nello stack; essa chiama `fatt(10)`, il cui frame nello stack è il successivo. Ciascuna chiamata invoca `fatt` ricorsivamente per calcolare il fattoriale del numero intero inferiore successivo. I frame di stack seguono l'ordine LIFO di queste chiamate. Che aspetto avrà lo stack quando il controllo tornerà da `fatt(10)` alla procedura chiamante?

SOLUZIONE



■

Approfondimento. La differenza tra il compilatore MIPS e il compilatore gcc è che il MIPS di solito non utilizza il frame pointer, per cui questo registro è a disposizione come ulteriore registro che deve essere salvato dal chiamato, denominato `$s8`. Questa modifica permette di risparmiare un paio di istruzioni nella chiamata a procedura e nel ritorno alla procedura chiamante. Tuttavia, questo rende più complessa la produzione del codice perché una procedura deve accedere al proprio frame di stack mediante il registro `$sp`, il cui contenuto può variare durante l'esecuzione della procedura stessa, se vengono inseriti nuovi valori in stack.

Un altro esempio di chiamata a procedura

Come ulteriore esempio si consideri la seguente procedura che lancia la funzione `tak`, un benchmark molto diffuso creato da Ikuo Takeuchi. Questa fun-

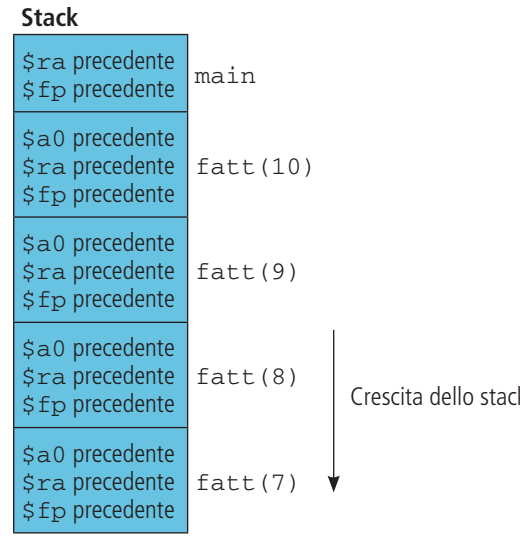


Figura A.6.3. **I frame di stack durante la chiamata di** `fatt(7)`.

zione non calcola niente di utile, ma è basata sul un algoritmo pesantemente ricorsivo che illustra bene le convenzioni di chiamata del MIPS.

```
int tak (int x, int y, int z)
{
  if (y < x)
    return 1 + tak( tak(x-1, y, z) ,
        tak(y-1, z, x) ,
        tak(z-1, x, y));
  else
    return z;
}

int main ()
{
  tak(18, 12, 6);
}
```

Il codice assembler di questo programma è mostrato di seguito. La funzione `tak` dapprima salva il suo indirizzo di ritorno nel suo frame di stack e i suoi argomenti in registri da preservare, poiché la procedura potrebbe effettuare chiamate che hanno bisogno di utilizzare i registri `$a0–$a2` e `$ra`. La funzione impiega i registri da preservare poiché essi contengono dei valori che devono persistere per tutto il tempo di vita della funzione, durante il quale avvengono parecchie chiamate che potrebbero potenzialmente modificare i registri.

```
    .text
    .globl tak

  tak:
    subu    $sp, $sp, 40
    sw      $ra, 32($sp)

    sw      $s0, 16($sp)     # x
    move    $s0, $a0
    sw      $s1, 20($sp)     # y
    move    $s1, $a1
    sw      $s2, 24($sp)     # z
    move    $s2, $a2
    sw      $s3, 28($sp)     # temporaneo
```

La procedura inizia quindi l'esecuzione controllando se $y < x$. Se la condizione non è soddisfatta, salta all'etichetta `L1`, riportata più avanti.

```
bge         $s1, $s0, L1      # Se (y < x)
```

Se $y < x$, si esegue il nucleo della procedura che contiene quattro chiamate ricorsive. La prima chiamata utilizza quasi gli stessi argomenti del suo genitore:

```
addiu       $a0, $s0, -1
move        $a1, $s1
move        $a2, $s2
jal         tak               # tak (x-1, y, z)
move        $s3, $v0
```

Si noti che il risultato della prima chiamata ricorsiva viene salvato nel registro `$s3`, in modo da poter essere utilizzato più avanti.

La funzione ora prepara gli argomenti per la seconda chiamata ricorsiva.

```
addiu       $a0, $s1, -1
move        $a1, $s2
move        $a2, $s0
jal         tak               # tak (y-1, z, x)
```

Nelle istruzioni riportate qui sotto, il risultato di questa chiamata ricorsiva viene salvato nel registro `$s0`, ma prima occorre leggere da questo registro, per l'ultima volta, il valore del primo argomento che vi era stato salvato.

```
addiu       $a0, $s2, -1
move        $a1, $s0
move        $a2, $s1
move        $s0, $v0
jal         tak               # tak (z-1, x, y)
```

Dopo le tre chiamate ricorsive interne, siamo pronti per la chiamata ricorsiva finale, a valle della quale il risultato della funzione si trova in `$v0` e l'esecuzione salta alla parte conclusiva della funzione.

```
move        $a0, $s3
move        $a1, $s0
move        $a2, $v0
jal         tak         # tak (tak(...), tak(...), tak(...))
addiu       $v0, $v0, 1
j           L2
```

Il codice all'etichetta L1 è il seguito del costrutto *if then else*. Esso semplicemente copia il valore dell'argomento z nel registro di ritorno e poi inizia la parte finale della funzione.

```
L1:
   move     $v0, $s2
```

Il codice sottostante è l'epilogo della funzione, nel quale vengono ripristinati i registri salvati e restituito il risultato della funzione al chiamante.

```
L2:
   lw       $ra, 32($sp)
   lw       $s0, 16($sp)
   lw       $s1, 20($sp)
   lw       $s2, 24($sp)
   lw       $s3, 28($sp)
   addiu    $sp, $sp, 40
   jr       $ra
```

La procedura `main` chiama la funzione `tak` con i suoi argomenti iniziali, quindi prende il risultato calcolato (7) e lo stampa utilizzando la chiamata di sistema di SPIM per stampare gli interi.

```
    .globl main
main:
    subu    $sp, $sp, 24
    sw      $ra, 16($sp)

    li      $a0, 18
    li      $a1, 12
    li      $a2, 6
    jal     tak             # tak(18, 12, 6)

    move    $a0, $v0
    li      $v0, 1          # Chiamata di sistema: print_int
    syscall

    lw      $ra, 16($sp)
    addiu   $sp, $sp, 24
    jr      $ra
```

A.7 Eccezioni e interrupt

Gestore degli interrupt: un frammento di codice che viene eseguito in risposta a un'eccezione o a un interrupt.

Il Paragrafo 4.9 descrive il modo in cui il MIPS gestisce le eccezioni. Queste comprendono sia eccezioni causate da errori che si verificano durante l'esecuzione di un'istruzione sia interrupt esterni generati da dispositivi I/O.[1] Questo paragrafo descrive più in dettaglio la gestione delle eccezioni e degli interrupt parte del **gestore degli interrupt**. Nei processori MIPS, una parte della CPU chiamata **coprocessore 0** memorizza le informazioni di cui il software ha bisogno per gestire le eccezioni e gli interrupt. Lo SPIM, il simulatore MIPS, non implementa tutti i registri del coprocessore 0, perché molti di essi sono inutili in un simulatore o fanno parte del sistema di memoria che non è implementato in SPIM. SPIM gestisce i seguenti registri del coprocessore 0:

Nome registro	Numero registro	Utilizzo
BadVAddr	8	indirizzo sui cui c'è stato un riferimento illecito alla memoria
Count	9	timer
Compare	11	valore che viene confrontato con il timer e provoca un interrupt quando i due valori sono uguali
Status	12	maschera di interrupt e bit di abilitazione
Causa	13	tipo di eccezione e bit di interrupt pendenti
EPC	14	indirizzo dell'istruzione che ha causato l'eccezione
Config	16	configurazione della macchina

Questi sette registri fanno parte dell'insieme dei registri del coprocessore 0; a essi si può accedere mediante le istruzioni `mfc0` e `mtc0`. Quando si verifica un'eccezione, viene scritto nel registro EPC l'indirizzo della relativa istruzione

[1] Questo paragrafo descrive le eccezioni che si possono verificare nell'architettura MIPS 32 che è la versione implementata nella versione 7.0 e nelle versioni successive di SPIM. Le versioni precedenti sono basate sul MIPS 1 che gestisce le eccezioni in maniera leggermente diversa. Convertire i programmi scritti per l'architettura MIPS 1 in modo che possano essere eseguiti sui MIPS 32 non dovrebbe risultare complesso, poiché le modifiche sono limitate ai campi dei registri Causa e Stato e alla sostituzione dell'istruzione `rfe` con l'istruzione `eret`.

in esecuzione. Se l'eccezione è causata da un interrupt esterno, invece, la gestione dell'eccezione inizia quando l'esecuzione dell'istruzione corrente non è ancora iniziata: in fase di fetch viene letta la prima istruzione di risposta all'interrupt invece dell'istruzione successiva nel codice. Se l'eccezione viene generata internamente al processore, essa è causata dall'esecuzione dell'istruzione puntata da EPC, tranne quando l'istruzione che genera l'eccezione si trova nel delay slot di un salto condizionato o incondizionato (si veda il Capitolo 4). In questo caso, EPC punta all'istruzione di salto, il bit BD del registro Causa viene impostato a 1 e il gestore delle eccezioni sa che l'istruzione che ha generato l'eccezione si trova all'indirizzo in EPC + 4. In entrambi i casi, comunque, il gestore delle eccezioni deve riprendere l'esecuzione del programma ritornando all'istruzione puntata da EPC.

Se l'istruzione che ha causato l'eccezione stava effettuando un accesso a memoria, il registro BadVAddr conterrà l'indirizzo della locazione di memoria a cui l'istruzione faceva riferimento.

Il registro Count (conteggio) è un contatore che viene incrementato a una velocità prefissata (il valore predefinito è di 10 millisecondi) mentre SPIM è in esecuzione. Quando il contenuto di Count eguaglia il valore contenuto nel registro Compare, viene generato un interrupt hardware con livello di priorità 5.

La figura A.7.1 mostra il sottoinsieme dei campi del registro Stato implementati dal simulatore SPIM. Il campo `maschera di interrupt` contiene un bit per ognuno dei sei livelli di interrupt hardware e dei due livelli software. Quando un bit della maschera è impostato a 1, un interrupt di quel livello può interrompere il processore, mentre se il bit della maschera è impostato a 0 gli interrupt di quel livello vengono disabilitati. Quando arriva un interrupt, imposta a 1 il suo bit di interrupt pendente (cioè «in attesa») nel registro Causa anche se il corrispondente bit della maschera è disabilitato. Quando un interrupt è pendente, interromperà il processore non appena il suo bit della maschera viene abilitato.

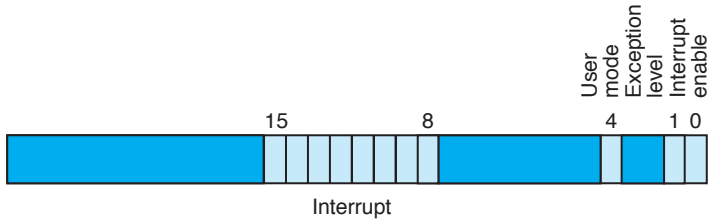


Figura A.7.1. Il registro Stato.

Il bit che definisce la modalità utente è impostato a 0 se il processore sta lavorando in modalità kernel e a 1 se sta lavorando in modalità utente. Su SPIM, questo bit è fissato a 1, dato che il processore simulato da SPIM non implementa la modalità kernel. Il bit di livello di eccezione di default è impostato a 0 e viene posto a 1 in caso di eccezione. Quando questo bit è impostato a 1, gli interrupt sono disabilitati e il registro EPC non viene aggiornato se si verifica un'altra eccezione; questo bit evita che il gestore delle eccezioni venga «disturbato» da un interrupt o da un'eccezione, ma deve essere impostato ancora a 0 non appena il gestore termina. Se il bit `abilita interrupt` è impostato a 1, gli interrupt sono permessi, altrimenti sono disabilitati.

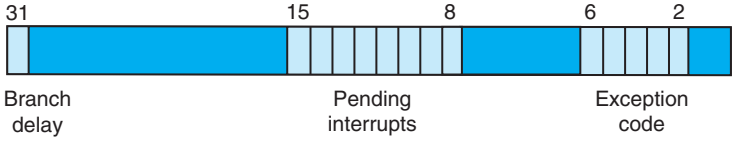


Figura A.7.2. Il registro Causa.

La figura A.7.2 mostra il sottoinsieme dei campi del registro Causa che sono implementati in SPIM. Il bit di ritardo nel salto è impostato a 1 se l'ultima eccezione è avvenuta durante l'esecuzione di un'istruzione contenuta in un delay slot di un salto condizionato. I bit degli interrupt pendenti vengono impostati a 1 quando viene sollevato un interrupt al corrispondente livello hardware o software. La parte del registro Causa riservata al codice dell'eccezione descrive la causa dell'eccezione attraverso i codici seguenti:

Numero	Nome	Causa dell'eccezione
0	Int	interrupt (hardware)
4	AdEL	eccezione d'errore di indirizzo (load o prelievo istruzione)
5	AdES	eccezione d'errore di indirizzo (store)
6	IBE	errore sul bus durante la lettura di un'istruzione
7	DBE	errore sul bus durante una load o store di dati
8	Sys	eccezione associata a una chiamata di sistema
9	Bp	eccezione da breakpoint
10	RI	eccezione di istruzione riservata
11	CpU	non implementata nel coprocessore
12	Ov	eccezione di overflow aritmetico
13	Tr	eccezione di trap
15	FPE	eccezione in un'operazione in virgola mobile

Eccezioni e interrupt nel processore MIPS fanno sì che l'esecuzione salti a una parte di codice contenuta a partire dall'indirizzo $8000\,0180_{esa}$ (compresa nello spazio kernel e non nello spazio di indirizzamento dell'utente) chiamata *gestore delle eccezioni* (*exception handler*). Tale codice esamina la causa dell'eccezione e salta quindi a un indirizzo opportuno nello spazio di indirizzamento del sistema operativo. Il sistema operativo risponde all'eccezione facendo terminare il processo che ha causato l'eccezione oppure effettuando determinate azioni. Un processo che causa un errore, come la richiesta di eseguire un'istruzione non implementata, viene «ucciso» (*killed*) dal sistema operativo. Viceversa, altre eccezioni come gli errori di page fault sono tradotte in richieste al sistema operativo da parte del processo in esecuzione, affinchè esso fornisca un determinato servizio (come quello di caricare in memoria una pagina dal disco); il sistema operativo elabora queste richieste e poi riprende l'esecuzione del processo. L'ultimo tipo di eccezione è costituito dagli interrupt generati da dispositivi esterni; questi generalmente richiedono al sistema operativo di spostare dati da e verso un dispositivo di I/O, per poi riprendere il processo interrotto.

Il codice nell'esempio seguente è un semplice gestore di eccezioni che invoca una procedura che stampa un messaggio ogni volta che si verifica un'eccezione (ma non un interrupt). Questo codice è simile al gestore di eccezioni (`exception.s`) utilizzato dal simulatore SPIM.

Un gestore delle eccezioni

ESEMPIO

Il gestore delle eccezioni salva dapprima il registro `$at`, che viene utilizzato dalle pseudoistruzioni contenute nel codice del gestore, e poi `$a0` e `$a1`, impiegati in seguito per il passaggio degli argomenti. Il gestore delle eccezioni non può salvare il contenuto di questi registri nello stack, come farebbe una procedura ordinaria, perché la causa dell'eccezione potrebbe essere un indirizzo della memoria costruito a partire da un valore sbagliato dello stack pointer, per esempio 0. Il gestore di eccezioni memorizza invece questi registri in uno dei registri dedicati alla gestione delle eccezioni, `$k1`, dato che non può accedere alla memoria senza utilizzare `$at`, e in due locazioni di memoria,

save0 e save1. Se la procedura di gestione delle eccezioni potesse essere interrotta, due locazioni di memoria non sarebbero sufficienti perché la seconda eccezione sovrascriverebbe i valori salvati durante la gestione della prima. Ma questa semplice procedura di gestione delle eccezioni termina la sua esecuzione prima di abilitare gli interrupt, per cui il problema non si pone.

```
.ktext 0x80000180
mov   $k1, $at     # Salva il registro $at
sw    $a0, save0   # Il gestore è non-rientrante e non può
                   # utilizzare lo stack per salvare $a0 e $a1.
sw    $a1, save1   # Non occorre salvare $k0/$k1
```

SOLUZIONE

Il gestore delle eccezioni copia quindi i registri Causa ed EPC nei registri della CPU. I registri Causa ed EPC non fanno parte dell'insieme dei registri della CPU, ma sono registri situati nel coprocessore 0 che è la parte della CPU che gestisce le eccezioni. L'istruzione mfc0 $k0, $13 copia il registro $13 del coprocessore 0 (il registro Causa) nel registro $k0 della CPU. Si noti che il gestore delle eccezioni non ha bisogno di salvare i registri $k0 e $k1 perché si suppone che i programmi utente non utilizzino questi registri. Il gestore delle eccezioni analizza il contenuto del registro Causa per capire se l'eccezione è stata causata da un interrupt (si veda la tabella precedente). Se è così, l'eccezione viene ignorata; se invece l'eccezione non è un interrupt, il gestore chiama la funzione print_eccez per stampare il messaggio.

```
mfc0   $k0, $13        # Copia il registro Causa in $k0

srl    $a0, $k0, 2     # Estrai il campo EccezCod
andi   $a0, $a0, 0xf

bgtz   $a0, done       # Salta se EccezCod è "Int" (0)

mov    $a0, $k0        # Copia il registro Causa in $a0
mfco   $a1, $14        # Copia il registro EPC in $a1
jal    print_eccez     # Stampa il messaggio d'errore
                       # se è un'eccezione
```

Prima di terminare, il gestore di eccezioni cancella il registro Causa, pone a 0 il bit opportuno del registro Stato per abilitare gli interrupt e pone a 0 il bit EXL che permette alle successive eccezioni di modificare il registro EPC. Infine, ripristina i registri $a0, $a1 e $at prima di eseguire l'istruzione eret (ritorno da eccezione) che ritorna all'istruzione puntata da EPC. Questo gestore delle eccezioni restituisce il controllo all'istruzione seguente a quella che ha causato l'eccezione, in modo da non eseguire una seconda volta l'istruzione sbagliata e causare di nuovo la stessa eccezione.

```
done:  mfco     $k0, $14      # Prendi l'EPC
       addiu    $k0, $k0, 4   # Non rieseguire
                              # l'istruzione errata

       mtc0     $k0, $14      # Scrivi l'EPC

       mtc0     $0, $13       # Cancella il registro Causa

       mfc0     $k0, $12      # Prendi il registro Stato
       andi     $k0, 0xfffd   # Cancella (poni a 0) il bit EXL
       ori      $k0, 0x1      # Abilita gli interrupt
       mtc0     $k0, $12
```

```
        lw      $a0, save0      # Ripristina i registri
        lw      $a1, save1
        mov     $at, $k1

        eret                    # Ritorna a EPC

        .kdata
save0: .word 0
save1: .word 0
```

Approfondimento. Nei processori MIPS reali, il ritorno dalla procedura di gestione delle eccezioni è più complesso. Il gestore delle eccezioni non può sempre saltare all'istruzione successiva a quella puntata da EPC. Per esempio, se l'istruzione che ha causato l'eccezione si trovava in un delay slot di un'istruzione di salto condizionato, quella da eseguire potrebbe non essere l'istruzione successiva in memoria.

A.8 Input e Output

SPIM simula un solo dispositivo di I/O: una console mappata in memoria mediante la quale un programma può leggere e scrivere dei caratteri. Quando un programma è in esecuzione, SPIM connette il proprio terminale (o una finestra di console separata nella versione per X-window, `xspim`, o nella versione per Windows, `PCSpim`) al processore. Un programma MIPS in esecuzione su SPIM può leggere i caratteri digitati dall'utente; inoltre, se il programma MIPS stampa dei caratteri, questi appaiono sul terminale o sulla finestra di consolle di SPIM. Un'eccezione a questa regola è la combinazione di tasti «control-C» che non viene passata al programma SPIM per la stampa ma causa l'arresto di SPIM e il ritorno alla modalità di comando. Quando l'esecuzione di un programma si arresta, per esempio perché è stato premuto control-C o perché il programma ha incontrato un breakpoint, il terminale viene riconnesso a SPIM in modo che l'utente possa scrivere dei comandi SPIM.

Per poter utilizzare l'I/O mappato in memoria (si veda più avanti), `spim` o `xspim` devono essere lanciati con l'opzione `-mapped_io`. `PCSpim` può abilitare l'I/O mappato in memoria attraverso un'opzione a linea di comando oppure dalla finestra di dialogo «*Settings*».

Il dispositivo terminale è composto da due unità indipendenti: un *ricevitore* e un *trasmettitore*. Il ricevitore legge i caratteri digitati sulla tastiera, mentre il trasmettitore visualizza i caratteri sulla console; le due unità sono completamente indipendenti. Questo significa, per esempio, che i caratteri digitati sulla tastiera non vengono automaticamente stampati a video, ma un programma deve leggere i caratteri dal ricevitore e scriverli sul trasmettitore.

Un programma controlla il terminale attraverso quattro registri di dispositivo mappati in memoria, come illustrato in figura A.8.1. «Mappato in memoria» significa che ogni registro corrisponde a una locazione di memoria speciale. Il **registro di Controllo del ricevitore** si trova nella locazione $\text{FFFF 0000}_{\text{esa}}$. Soltanto due dei suoi bit vengono effettivamente utilizzati: il bit 0 è chiamato *ready* (pronto) e quando è impostato a 1 significa che un carattere è arrivato dalla tastiera ma non è ancora stato letto dal registro Dati del ricevitore. Il bit ready è a sola lettura: eventuali scritture di questo bit vengono ignorate. Il bit ready cambia da 0 a 1 quando un carattere viene digitato sulla tastiera e da 1 a 0 quando il carattere viene letto dal registro Dati del ricevitore.

Il bit 1 del registro di controllo del ricevitore è il bit di «abilitazione degli interrupt» della tastiera; questo bit può essere sia letto che scritto dai programmi ed è impostato inizialmente a 0. Quando il bit viene impostato a 1 da

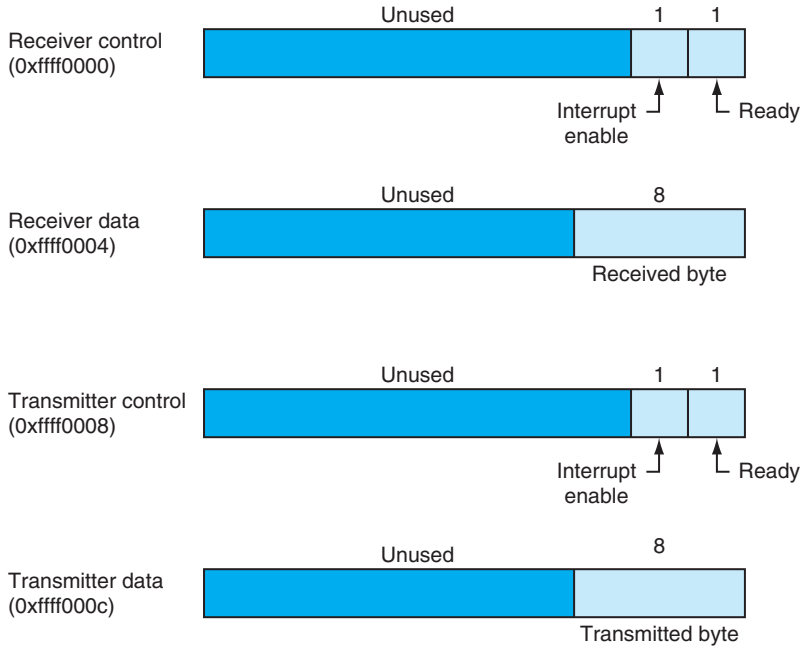


Figura A.8.1. Il terminale è controllato da quattro registri di dispositivo, ciascuno dei quali corrisponde a una locazione di memoria a un determinato indirizzo. Soltanto pochi bit di questi registri vengono effettivamente utilizzati. Gli altri bit vengono sempre letti come 0 e vengono ignorati in scrittura.

un programma, il terminale invia un interrupt a livello hardware ogni volta che viene digitato un carattere, e il bit ready diventa 1. Per far sì che l'interrupt abbia effetto sul processore, gli interrupt devono essere abilitati anche nel registro Stato (si veda il Paragrafo A.7). Tutti gli altri bit del registro di controllo del ricevitore non vengono utilizzati.

Il secondo registro del dispositivo terminale è il *registro dati del ricevitore* (contenuto nell'indirizzo FFFF 0004_{esa}). Gli otto bit meno significativi di questo registro contengono l'ultimo carattere digitato sulla tastiera, tutti gli altri bit contengono 0. Questo registro è di sola lettura e cambia soltanto quando viene premuto un nuovo tasto sulla testiera. La lettura del registro Dati del ricevitore porta a 0 il bit ready del registro di controllo del ricevitore; il contenuto del registro Dati del ricevitore è indefinito se il registro di controllo del ricevitore è 0.

Il terzo registro del terminale è il *registro di controllo del trasmettitore* (contenuto nell'indirizzo FFFF 0008_{esa}). Soltanto i due bit meno significativi di questo registro vengono utilizzati; essi si comportano in modo molto simile ai corrispondenti bit del registro di controllo del ricevitore. Il bit 0 è detto ready ed è di sola lettura. Se questo bit è impostato a 1, il trasmettitore è pronto a ricevere un nuovo carattere in uscita; se è impostato a 0, il trasmettitore è ancora occupato a scrivere il carattere precedente. Il bit 1 è detto *bit abilitazione degli interrupt* e può essere sia letto che scritto. Se tale bit è valorizzato a 1, il terminale richiede un interrupt a livello hardware ogni volta che il trasmettitore è pronto a ricevere un nuovo carattere, e il bit ready diventa 1.

L'ultimo registro del dispositivo è il registro Dati del trasmettitore (contenuto nell'indirizzo FFFF $000C_{esa}$). Quando un numero viene scritto in questa locazione, i suoi otto bit meno significativi (cioè un carattere ASCII, come mostrato nella Figura 2.15 del Capitolo 2) sono inviati alla console. Quando il registro Dati del trasmettitore viene scritto, il bit ready del registro di controllo del trasmettitore viene posto a 0. Tale bit rimane a 0 finché è passato un tempo sufficiente a trasmettere il carattere al terminale, dopodichè ritorna a 1. Il registro Dati del trasmettitore dovrebbe essere scritto soltanto quando il bit Ready del registro di controllo del trasmettitore è impostato a 1. Se il trasmettitore non è pronto, le scritture nel registro Dati del trasmettitore vengono ignorate: la scrittura sembra abbia avuto successo, ma il carattere non viene prodotto in uscita.

I calcolatori reali hanno bisogno di tempo per inviare i caratteri a una console o al terminale e questi ritardi vengono simulati da SPIM. Per esempio, quando il trasmettitore inizia a scrivere un carattere il bit ready del trasmettitore è impostato a 0 per un certo periodo di tempo. SPIM misura il tempo in numero di istruzioni eseguite, non in tempo reale; ciò significa che il trasmettitore non è di nuovo pronto finché il processore non ha eseguito un numero prefissato di istruzioni. Se fermate il calcolatore e osservate il bit ready, trovate che esso non ha cambiato valore. Altrimenti, lasciando lavorare il calcolatore, il bit ready prima o poi viene nuovamente impostato a 1.

A.9 SPIM

SPIM è un simulatore software che esegue programmi in linguaggio assembler scritti per i processori che implementano l'architettura MIPS-32, più precisamente la versione 1 di questa architettura, con una mappatura della memoria prefissata, nessuna memoria cache e soltanto i coprocessori 0 e 1.[2] Il nome SPIM è semplicemente la parola MIPS letta al contrario. SPIM può leggere ed eseguire direttamente file in linguaggio assembler; si tratta di un sistema indipendente che esegue programmi MIPS: contiene un programma di debug e fornisce alcuni servizi di sistema. SPIM è molto più lento di un calcolatore reale (di oltre 100 volte); ciononostante, il suo bassissimo costo e la sua disponibilità non si possono ottenere con soluzioni hardware reali.

Molti di voi forse si staranno chiedendo: «perché utilizzare un simulatore quando la maggior parte delle persone possiede PC che contengono processori notevolmente più veloci di SPIM?» Una ragione è che il processore nei PC è un Intel 80x86, la cui architettura risulta di gran lunga meno modulare e più complessa da comprendere e da programmare dei processori MIPS; l'architettura MIPS, invece, può essere considerata la sintesi di un'architettura RISC pulita e semplice.

Inoltre, i simulatori possono fornire un ambiente migliore per la programmazione in assembler rispetto a un calcolatore reale, perché possono rivelare più errori e fornire un'interfaccia più recente.

Infine, i simulatori sono strumenti utili per studiare i calcolatori e i programmi che vi vengono eseguiti. Essendo implementati a livello di software (e non di silicio), i simulatori possono essere rivisti e modificati facilmente per aggiungere nuove istruzioni, costruire nuovi sistemi (come i multiprocessori) o, semplicemente, raccogliere dati.

Simulazione di una macchina virtuale

L'architettura MIPS di base è difficile da programmare direttamente a causa dei salti condizionati ritardati, del caricamento ritardato dei dati dalla memoria (load) e delle limitate modalità di indirizzamento. Tali difficoltà sono tollerabili se si considera che questi calcolatori sono stati progettati per essere programmati in linguaggi ad alto livello e presentano un'interfaccia pensata più per i compilatori che per i programmatori assembler. Una buona parte

[2] Le vecchie versioni di SPIM (quelle precedenti la 7.0) implementavano l'architettura MIPS-1 utilizzata dai processori MIPS R2000 originali. Questa architettura è quasi un sottoinsieme proprio dell'architettura MIPS-32, dove la differenza sta nel modo di gestire le eccezioni. Il MIPS-32 ha anche introdotto circa 60 nuove istruzioni che sono supportate da SPIM. I programmi che venivano eseguiti sulle precedenti versioni e non utilizzavano le eccezioni dovrebbero essere eseguiti sulle versioni più recenti di SPIM, senza bisogno di modifiche, mentre i programmi che usavano le eccezioni richiedono piccole modifiche.

della complessità della programmazione è dovuta alle istruzioni con azioni ritardate. Un **salto condizionato ritardato** richiede due cicli di clock per essere eseguito (si veda il Capitolo 4), ma nel secondo ciclo di clock inizia l'esecuzione dell'istruzione successiva al salto condizionato. Questa istruzione può effettuare un'operazione utile che normalmente dovrebbe essere effettuata prima del salto, ma può essere costituita anche da una `nop` (*not operation*) che non esegue alcuna operazione. Analogamente il caricamento ritardato dalla memoria richiede due cicli per trasferire un dato dalla memoria alla CPU, per cui l'istruzione immediatamente successiva a una load non può utilizzare tale dato (si veda il Paragrafo 4.2).

Il MIPS ha scelto saggiamente di nascondere questa complessità facendo in modo che l'assemblatore implementi una **macchina virtuale** in cui i salti condizionati e il caricamento dei dati avvengono senza ritardi; inoltre, contiene un insieme di istruzioni più ricco di quello dell'hardware effettivo e riorganizza le istruzioni in modo da riempire gli slot dei ritardi. Il calcolatore virtuale fornisce anche delle pseudoistruzioni che appaiono come istruzioni reali nei programmi in linguaggio assembler. D'altro canto, l'hardware non conosce le pseudoistruzioni, per cui l'assemblatore si incarica di tradurle in sequenze di istruzioni in linguaggio macchina effettive. Per esempio, l'hardware MIPS fornisce soltanto istruzioni di salto condizionato basate sul confronto tra il contenuto di un registro e il numero 0. Altri salti condizionati, come quelli che valutano se il contenuto di un registro sia maggiore di un altro, vengono implementati utilizzando un'istruzione che confronta il contenuto di due registri e un'istruzione che salta se il risultato del confronto è vero (cioè diverso da zero).

Macchina virtuale: un calcolatore virtuale che implementa senza ritardo i salti condizionati e il caricamento dei dati e che comprende un insieme di istruzioni più ricco di quello dell'hardware reale.

Se non diversamente specificato, SPIM simula la macchina più «ricca», essendo questa la macchina che la maggior parte dei programmatori trova più utile. SPIM però può anche simulare i salti condizionati e il caricamento ritardato presenti nell'hardware reale. Nel seguito descriveremo la macchina virtuale e accenneremo solo occasionalmente al fatto che una certa caratteristica non appartiene all'hardware effettivo. Così facendo, seguiamo la convenzione dei programmatori (e dei compilatori) in linguaggio assembler MIPS, i quali utilizzano per consuetudine la macchina completa come se fosse implementata su silicio.

Primi passi con SPIM

Il seguito di questa appendice introduce SPIM e il linguaggio assembler del MIPS R2000. È probabile che non dobbiate mai occuparvi di gestire molti dei dettagli qui presentati: la notevole quantità di informazioni può talvolta oscurare il fatto che SPIM è un programma semplice e intuitivo da utilizzare. Questo paragrafo inizia con una breve guida all'utilizzo di SPIM, che dovrebbe permettervi di caricare ed eseguire semplici programmi MIPS, e farne il debug.

SPIM è disponibile in diverse versioni per i diversi tipi di calcolatori. La versione di base è quella più semplice: è chiamata `spim` ed è costituita da un programma gestito da linea di comando, eseguito in una finestra di console. Esso funziona come la maggior parte dei programmi di questo tipo: si scrive una linea di testo, si digita il tasto `return` e `spim` esegue il comando corrispondente. Nonostante la mancanza di un'interfaccia intuitiva, `spim` è in grado di fare tutto ciò che sanno fare i suoi cugini più sofisticati.

Sono due i cugini più avanzati di `spim`. La versione che lavora nell'ambiente X-window di un sistema UNIX o Linux si chiama `xspim`. `xspim` è un programma più semplice da imparare rispetto a `spim`, perché i suoi comandi sono sempre visibili sullo schermo e perché mostra a video il contenuto dei registri della macchina e della memoria. L'altra versione sofisticata si chiama

`PCspim` e lavora sotto Microsoft Windows. Le versioni UNIX e Windows di SPIM sono contenute nel CD (sezione «*Tutorials*»). I tutorial su `xpim`, `pcSpim`, `spim` e le opzioni dei comandi a linea di comando di SPIM sono contenute nel CD (sezione «*Software*»).

Prima di lanciare in esecuzione questi programmi, vi consigliamo di leggere i relativi manuali

Caratteristiche principali

Benché simuli fedelmente il calcolatore MIPS, SPIM è pur sempre un simulatore e certe sue caratteristiche non sono identiche a quelle di un calcolatore reale. È ovvio che la temporizzazione delle istruzioni e il sistema di memoria non sono identici. SPIM non simula le memorie cache o la latenza della memoria, e nemmeno rispecchia accuratamente i ritardi introdotti dalle operazioni in virgola mobile e dalle istruzioni di moltiplicazione e divisione. Inoltre, le istruzioni in virgola mobile non rilevano molte condizioni di errore che genererebbero un'eccezione su una macchina reale.

Un'altra caratteristica, tipica anche di una macchina reale, è che una pseudoistruzione viene espansa in parecchie istruzioni in linguaggio macchina. Quando si esamina la memoria, le istruzioni che compaiono sono diverse da quelle contenute nel programma sorgente. La corrispondenza tra i due insiemi di istruzioni è abbastanza semplice, poiché SPIM non riorganizza le istruzioni per riempire gli slot dei ritardi.

Ordinamento dei byte

I processori possono enumerare i byte di una parola in modo che il byte con numero d'ordine inferiore sia il più a sinistra o il più a destra. La convenzione adottata da una macchina è detta ordinamento dei byte. I processori MIPS possono operare sia con l'ordinamento *big endian* dei byte sia con l'ordinamento *little endian*. Per esempio, in una macchina big endian la direttiva `.byte 0, 1, 2, 3` darebbe come risultato una parola di memoria contenente:

N. byte			
0	1	2	3

mentre in una macchina little endian, la parola conterrebbe

N. byte			
3	2	1	0

SPIM può funzionare con entrambi gli ordinamenti dei byte. L'ordinamento dei byte di SPIM è lo stesso di quello della macchina su cui il simulatore è in esecuzione. Per esempio, su un Intel 80x86 SPIM è little endian, mentre su una macchina Macintosh o Sun SPARC SPIM è big endian.

Chiamate di sistema

SPIM fornisce un piccolo insieme di funzioni di servizio simili a quelle del sistema operativo mediante l'istruzione di chiamata di sistema (`syscall`). Per richiedere una funzione di servizio, un programma carica il codice di chiamata di sistema relativo (si veda la Figura A.9.1) nel registro `$v0` e gli argomenti nei registri `$a0`–`$a3`, o `$f12` se il codice è in virgola mobile. Le chiamate di sistema che restituiscono dei dati inseriscono il risultato nel registro `$v0`, o

Funzione di Servizio	Codice di chiamata di sistema	Argomenti	Risultato
print_int	1	$a0 = intero	
print_float	2	$f12 = virgola mobile, singola pr.	
print_double	3	$f12 = virgola mobile, doppia pr.	
print_string	4	$a0 = stringa	
read_int	5		Intero (in $v0)
read_float	6		Virgola mobile, singola pr. (in $v0)
read_double	7		Virgola mobile, doppia pr. (in $v0)
read_string	8	$a0 = buffer, $a1 = lunghezza	
sbrk	9	$a0 = quantità	Indirizzo (in $v0)
exit	10		
print_char	11	$a0 = carattere	
read_char	12		Carattere (in $v0)
open	13	$a0 = nome file (stringa), $a1 = flags, $a2 = modalità	Descrittore del file (in $a0)
read	14	$a0 = descrittore del file, $a1 = buffer, $a2 = lunghezza	Num. caratteri letti (in $a0)
write	15	$a0 = descrittore file, $a1 = buffer, $a2 = lunghezza	Num. caratteri scritti (in $a0)
close	16	$a0 = descrittore del file	
exit2	17	$a0 = risultato	

Figura A.9.1. Funzioni di servizio di sistema.

$f0 per i risultati in virgola mobile. Per esempio, il codice seguente stampa la stringa «La risposta è 5»:

```
   .data
str:
   .asciiz "La risposta è "
   .text
   li          $v0, 4      # Codice della chiamata di sistema
                           # per print_str
   la          $a0, str    # Indirizzo della stringa da stampare
   syscall                 # Stampa la stringa

   li          $v0, 1      # Codice della chiamata di sistema
                           # per print_int
   la          $a0, 5      # Numero intero da stampare
   syscall                 # Stampa l'intero
```

Alla chiamata della funzione di sistema print_int viene passato un intero e questa lo stampa sulla console. print_float stampa un numero in virgola mobile in singola precisione e print_double stampa un numero in virgola mobile in doppia precisione; a print_string viene passato un puntatore a una stringa che termina con il carattere null (un byte contenente 0), la quale viene stampata sul terminale della console.

Le chiamate di sistema read_int, read_float e read_double leggono un'intera linea di input fino al carattere newline compreso; i caratteri successivi al numero vengono ignorati, read_string ha lo stesso funzionamento della procedura di libreria UNIX fgets: essa legge fino a un massimo di $n - 1$ caratteri e li scrive in un buffer; inoltre termina la stringa con il byte null. Se la linea contiene meno di $n - 1$ caratteri, read_string legge fino al carattere

`newline` compreso; anche in questo caso termina la stringa con il byte null.

Attenzione: i programmi che utilizzano queste chiamate di sistema per leggere dal terminale non possono utilizzare l'I/O mappato in memoria (si veda il Paragrafo A.8).

`sbrk` restituisce un puntatore a un blocco di memoria contenente *n* byte. `exit` arresta il programma che SPIM sta eseguendo, mentre `exit2` termina il programma SPIM e l'argomento di `exit2` diventa il valore restituito quando il simulatore SPIM stesso termina.

`print_char` e `read_char` scrivono e leggono un singolo carattere. `open`, `read`, `write` e `close` corrispondono alle chiamate di sistema standard di UNIX.

A.10 Il linguaggio assembler del MIPS R2000

Un processore MIPS consiste di un'unità di elaborazione intera, la CPU, e di una collezione di coprocessori che eseguono compiti ausiliari o che lavorano su altri tipi di dati, come i numeri in virgola mobile (si veda la Figura A.10.1). SPIM simula due coprocessori: il coprocessore 0 gestisce le eccezioni e gli interrupt, mentre il coprocessore 1 è l'unità di calcolo in virgola mobile. SPIM simula la maggior parte delle caratteristiche di queste unità.

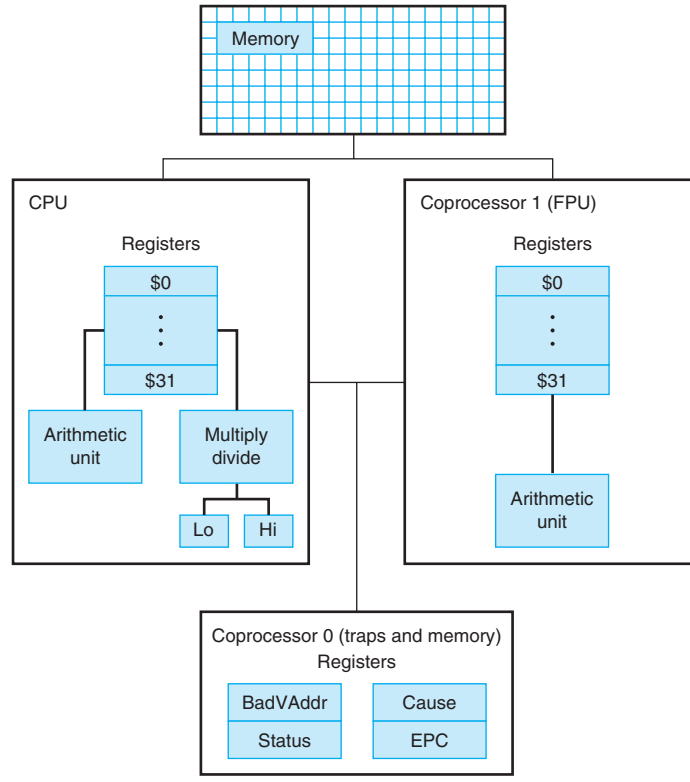


Figura A.10.1. CPU e FPU del MIPS R2000.

Modalità di indirizzamento

Il MIPS è un'architettura di tipo load/store; ciò significa che soltanto le istruzioni di load e di store accedono alla memoria. Le istruzioni di calcolo operano soltanto su numeri contenuti nei registri. La macchina semplice supporta soltanto una modalità di indirizzamento, `c(rx)`, nella quale l'indirizzo viene formato sommando l'operando immediato `c` al contenuto del registro `rx`. La macchina virtuale fornisce anche le seguenti modalità di indirizzamento per le istruzioni di load e di store:

Formato	Calcolo dell'indirizzo
(registro)	contenuto del registro
imm	immediato
imm (registro)	immediato + contenuto del registro
etichetta	indirizzo dell'etichetta
etichetta ± imm	indirizzo dell'etichetta + o – immediato
etichetta ± imm (registro)	indirizzo dell'etichetta + o – (immediato + contenuto del registro)

La maggior parte delle istruzioni di load e di store operano solo su dati allineati. Una quantità si dice **allineata** se il suo indirizzo di memoria è un multiplo della sua dimensione in byte. Se un oggetto occupa una mezza parola (*halfword*) sarà memorizzato, quindi, a partire da indirizzi pari, mentre se un oggetto occupa una parola intera sarà memorizzato a partire da indirizzi multipli di quattro. Il MIPS, comunque, mette a disposizione alcune istruzioni per leggere e scrivere dati non allineati: `lwl`, `lwr`, `swl` e `swr`.

Approfondimento. L'assembler del MIPS (e lo SPIM) gestisce le modalità di indirizzamento più complesse generando una o più istruzioni per calcolare l'indirizzo complesso prima dell'operazione di load o di store. Per esempio, si supponga che l'etichetta tabella si riferisca alla locazione di memoria 0x10000004 e un programma contenga l'istruzione

```
ld $a0, tabella + 4($a1)
```

L'assemblatore tradurrebbe questa istruzione nelle istruzioni

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

La prima istruzione carica i 16 bit superiori dell'indirizzo associato all'etichetta nel registro `$at`, che è il registro che l'assemblatore riserva alle proprie funzioni. La seconda istruzione aggiunge il contenuto del registro `$a1` all'indirizzo parziale dell'etichetta. Infine, l'istruzione di load sfrutta la modalità effettiva di indirizzamento hardware per aggiungere al contenuto del registro `$at` la somma dei 16 bit inferiori dell'indirizzo associato all'etichetta e dei 16 bit dello spiazzamento nell'istruzione originale.

Sintassi dell'assemblatore

I commenti in un file assembler iniziano con il carattere cancelletto (#): tutto ciò che è compreso tra questo carattere e la fine della linea viene ignorato dall'assemblatore.

Gli identificatori sono costituiti da una sequenza di caratteri alfanumerici, caratteri di sottolineatura (_) e punti (.) che non cominciano con un numero. I codici operativi delle istruzioni sono parole riservate che non possono essere utilizzate come identificatori. Le etichette vengono dichiarate ponendole all'inizio di una linea, seguite dai due punti (:); per esempio:

```
        .data
elem:   .word 1
        .text
        .globl main     # Deve essere globale!
main:   lw $t0, elem
```

I numeri sono considerati decimali, se non diversamente specificato. Se sono preceduti da *0x*, vengono interpretati come numeri esadecimali: le stringhe 256 e 0x100 denotano quindi lo stesso valore.

Le stringhe sono delimitate dalle doppie virgolette (" "). I caratteri speciali nelle stringhe seguono la convenzione del linguaggio C:

- a capo (*newline*) `\n`
- tabulazione `\t`
- virgolette `\"`

SPIM supporta un sottoinsieme delle direttive MIPS per l'assemblatore:

`.align n`	Allinea il prossimo dato a un confine di 2^n byte. Per esempio, `.align 2` allinea il valore successivo al confine di una parola. `.align 0` disattiva l'allineamento automatico delle direttive `.half`, `.word`, `.float` e `.double`, fino alla direttiva `.data` o `.kdata` successiva.
`.ascii str`	Salva la stringa `str` in memoria ma non la termina con il carattere null.
`.asciiz str`	Memorizza la stringa `str` in memoria e la termina con il carattere null.
`.byte b1, ..., bn`	Memorizza gli *n* valori in byte successivi della memoria.
`.data <addr>`	Gli elementi seguenti vengono memorizzati nel segmento dati. Se è presente l'argomento opzionale `addr`, gli elementi che seguono vengono memorizzati a partire dall'indirizzo `addr`.
`.double d1, ..., dn`	Memorizza gli *n* numeri in virgola mobile in doppia precisione in locazioni di memoria consecutive.
`.extern sym size`	Dichiara che il dato memorizzato in corrispondenza di `sym` occupa `size` byte ed è un'etichetta globale. Questa direttiva permette all'assemblatore di memorizzare il dato in una porzione del segmento dati cui si può accedere in modo efficiente per mezzo del registro `$gp`.
`.float f1, ..., fn`	Memorizza gli *n* numeri in virgola mobile in singola precisione in locazioni di memoria consecutive.
`.globl sym`	Dichiara che l'etichetta `sym` è globale e che si può fare riferimento a essa da altri file.
`.half h1, ..., hn`	Memorizza gli *n* numeri a 16 bit in locazioni di memoria consecutive.
`.kdata <addr>`	I dati che seguono vengono memorizzati nel segmento dati del kernel. Se è presente l'argomento opzionale `addr`, gli elementi che seguono vengono memorizzati a partire dall'indirizzo `addr`.
`.ktext <addr>`	Gli elementi che seguono vengono memorizzati nel segmento testo del kernel. In SPIM, tali elementi possono essere soltanto istruzioni o parole (si veda la direttiva `.word`). Se è presente l'argomento opzionale `addr`, gli elementi che seguono vengono memorizzati a partire dall'indirizzo `addr`.
`.set noat` e `.set at`	La prima direttiva evita che SPIM segnali che le istruzioni successive possano utilizzare il registro `$at`.

	La seconda direttiva riabilita questa segnalazione. Dato che le pseudoistruzioni vengono espanse in codice che utilizza il registro `$at`, i programmatori devono essere molto cauti nel lasciare dei valori in tale registro.
`.space n`	Alloca *n* byte di spazio nel segmento corrente (che, in SPIM, deve essere necessariamente il segmento dati).
`.text <addr>`	Gli elementi che seguono vengono posti nel segmento testo. In SPIM, tali elementi possono essere soltanto istruzioni o parole (si veda la direttiva `.word`). Se è presente l'argomento opzionale `addr`, gli elementi che seguono vengono memorizzati a partire dall'indirizzo `addr`.
`.word w1, ..., wn`	Memorizza le *n* variabili a 32 bit in parole di memoria successive.

SPIM non fa distinzione fra le diverse parti del segmento dati (`.data`, `.rdata` e `.sdata`).

La codifica delle istruzioni MIPS

La figura A.10.2 mostra come un'istruzione MIPS sia codificata come numero binario. Ogni colonna contiene la codifica di un campo (un gruppo di bit contigui) di un'istruzione. I numeri sul margine sinistro rappresentano il contenuto di un campo. Per esempio, il codice operativo dell'istruzione `j` presenta il valore 2 nel campo del codice operativo. Il testo al di sopra di ciascuna colonna definisce il nome del campo e specifica quali bit occupa all'interno dell'istruzione. Per esempio, il campo `op` è contenuto nei bit 26-31 di un'istruzione; questo campo è contenuto nella maggior parte delle istruzioni. Alcuni gruppi di istruzioni, tuttavia, utilizzano campi addizionali per differenziare istruzioni simili. Per esempio, le diverse istruzioni in virgola mobile sono definite dai bit 0-5. Le frecce che partono dalla prima colonna mostrano i codici operativi che utilizzano questi campi addizionali.

Il formato delle istruzioni

Nel seguito di questa appendice sono descritte sia le istruzioni implementate dall'hardware effettivo del MIPS sia le pseudoistruzioni messe a disposizione dall'assemblatore MIPS. I due tipi di istruzioni si distinguono facilmente: per le istruzioni effettive, nella rappresentazione binaria si possono identificare i diversi campi dell'istruzione. Per esempio:

Addizione (con overflow)

`add rd, rs, rt`

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

L'istruzione `add` contiene sei campi. La dimensione in bit di ogni campo è il piccolo numero riportato sotto il campo stesso. Questa istruzione inizia con sei bit a 0. I numeri che specificano i registri iniziano sempre con la lettera *r*, per cui il campo successivo specifica un registro su 5 bit, chiamato `rs`. Questo è il registro specificato dal secondo argomento nella codifica assembler simbolica alla sinistra dell'istruzione. Un altro campo molto comune è $\texttt{imm}_{16}$, che rappresenta un numero su 16 bit.

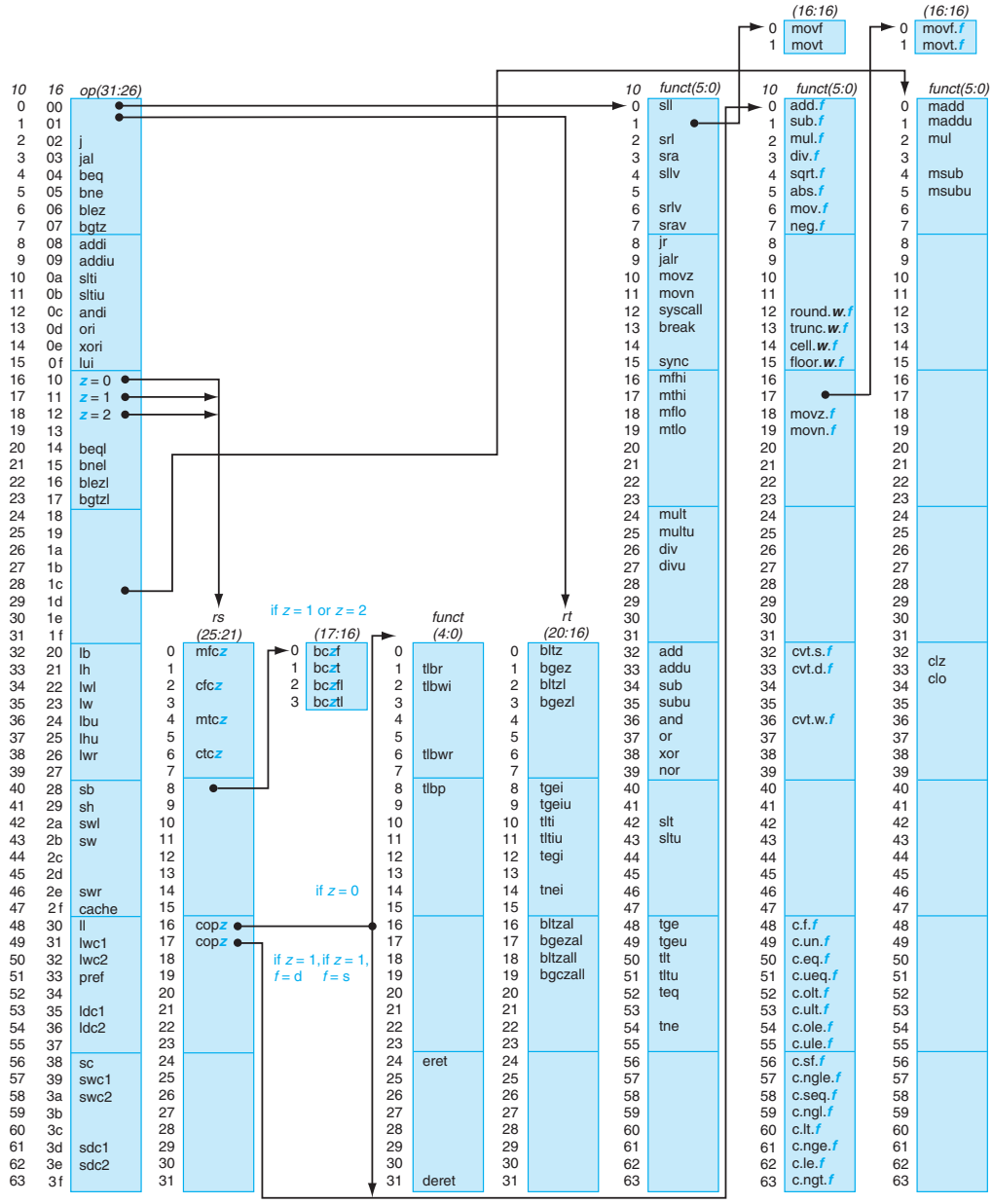


Figura A.10.2. Mappa dei codici operativi del MIPS. Il contenuto di ciascun campo è riportato alla sua sinistra. Il codice operativo (bit da 31 a 26) riportato nella terza colonna viene mostrato in base 10 nella prima colonna e in base 16 nella seconda. Il codice operativo, `op`, specifica completamente l'operazione che il MIPS deve eseguire, tranne che per sei particolari valori di `op`: 0, 1, 16, 17, 18, 19. Per questi codici operativi, l'operazione da eseguire viene determinata da altri campi, identificati dalle frecce. Nell'ultimo campo (`funct`), «*f*» sta per «s» se `rs` = 16 e `op` = 17 o «d» se `rs` = 17 e `op` = 17. Nel secondo campo (`rs`), «*z*» sta a significare «0», «1», «2» o «3» per `op` = 16, 17, 18 o 19, rispettivamente. Se `rs` = 16, l'operazione viene specificata altrove: se z = 0, l'operazione viene specificata nel quarto campo (bit da 4 a 0); se z = 1, l'operazione viene specificata nell'ultimo campo con *f* = s. Se `rs` = 17 e z = 1, allora le operazioni sono specificate nell'ultimo campo con *f* = d.

Le pseudoistruzioni seguono grossomodo le stesse convenzioni, ma omettono le informazioni sulla codifica dell'istruzione. Per esempio:

Moltiplicazione (senza overflow)

```
mul rdest, rsrc1, rsrc2        pseudoistruzione
```

Nelle pseudoistruzioni, `rdest` e `rsrc1` sono registri, mentre `rsrc2` può essere un registro oppure un valore immediato. L'assemblatore e SPIM traduco-

no un'istruzione da una forma più generale, come `add $v1, $a0, 0x55`, in una forma specializzata, per esempio `addi $v1, $a0, 0x55`.

Le istruzioni aritmetiche e logiche

Valore assoluto

`abs rdest, rsrc` pseudoistruzione

Inserisce il valore assoluto del registro `rsrc` nel registro `rdest`.

Addizione (con overflow)

`add rd, rs, rt`

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

Addizione (senza overflow)

`addu rd, rs, rt`

0	rs	Rt	rd	0	0x21
6	5	5	5	5	6

Inserisce la somma dei registri `rs` ed `rt` nel registro `rd`.

Addizione con immediato (con overflow)

`addi rt, rs, imm`

8	rs	rt	imm
6	5	5	16

Addizione con immediato (senza overflow)

`addiu rt, rs, imm`

9	rs	rt	imm
6	5	5	16

Inserisce nel registro `rt` la somma del registro `rs` e dell'immediato, esteso con segno.

AND

`and rd, rs, rt`

0	rs	rt	rd	0	0x24
6	5	5	5	5	6

Inserisce l'AND logico dei registri `rs` ed `rt` nel registro `rd`.

AND con immediato

`andi rt, rs, imm`

0xc	rs	rt	imm
6	5	5	16

Inserisce nel registro `rt` l'AND logico del registro `rs` e dell'immediato, esteso con segno.

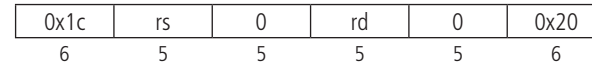
Conta gli 1 nelle posizioni più significative

`clo rd, rs`

0x1c	rs	0	rd	0	0x21
6	5	5	5	5	6

Conta gli 0 nelle posizioni più significative

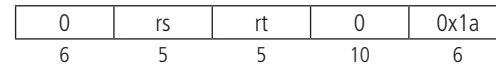
`clz rd, rs`



Conta il numero degli 1 (0) nelle posizioni più significative della parola contenuta nel registro `rs` e mette il risultato nel registro `rd`. Se una parola è composta da tutti 1 (0), il risultato è 32.

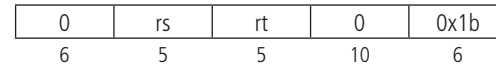
Dividi (con overflow)

`div rs, rt`



Dividi (senza overflow)

`divu rs, rt`



Divide il registro `rs` per il registro `rt`. Pone il quoziente nel registro `lo` e il resto nel registro `hi`. Si noti che se un operando è negativo, il resto non viene specificato dall'architettura MIPS e dipende dalla convenzione della macchina sulla quale SPIM è in esecuzione.

Dividi (con overflow)

`div rdest, rsrc1, src2` pseudoistruzione

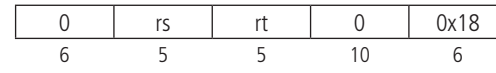
Dividi (senza overflow)

`divu rdest, rsrc1, src2` pseudoistruzione

Inserisce nel registro `rdest` il quoziente della divisione del registro `rsrc1` per `src2`.

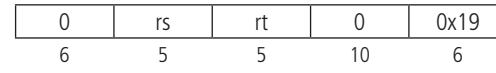
Moltiplica

`mult rs, rt`



Moltiplica senza segno

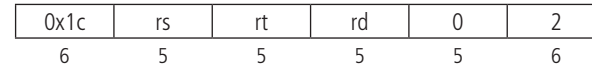
`multu rs, rt`



Moltiplica i registri `rs` e `rt`. Inserisce la parola di ordine più basso (meno significativa) nel registro `lo` e la parola di ordine maggiore (più significativa) nel registro `hi`.

Moltiplica (senza oveflow)

`mul rd, rs, rt`



Inserisce i 32 bit meno significativi del prodotto fra i registri `rs` e `rt` nel registro `rd`.

Moltiplica (con oveflow)

`mulo rdest, rsrc1, src2` pseudoistruzione

Moltiplica senza segno (con oveflow)

`mulou rdest, rsrc1, src2` pseudoistruzione

Inserisce i 32 bit meno significativi del prodotto fra i registri `rsrc1` e `src2` nel registro `rdest`.

Moltiplica e somma

`madd rs, rt`

0x1c	rs	rt	0	0
6	5	5	10	6

Moltiplica e somma senza segno

`maddu rs, rt`

0x1c	rs	rt	0	1
6	5	5	10	6

Moltiplica i registri `rs` e `rt` e somma il prodotto su 64 bit risultante al numero su 64 bit contenuto nei registri `lo` e `hi` concatenati.

Moltiplica e sottrai

`msub rs, rt`

0x1c	rs	rt	0	4
6	5	5	10	6

Moltiplica e sottrai senza segno

`msubu rs, rt`

0x1c	rs	rt	0	5
6	5	5	10	6

Moltiplica i registri `rs` ed `rt` e sottrae il prodotto su 64 bit risultante dal numero su 64 bit contenuto nei registri `lo` e `hi` concatenati.

Valore negativo (con oveflow)

`neg rdest, rsrc` pseudoistruzione

Valore negativo (senza oveflow)

`negu rdest, rsrc` pseudoistruzione

Inserisce il negativo del registro `rsrc` nel registro `rdest`.

NOR

`nor rd, rs, rt`

0	rs	rt	rd	0	0x27
6	5	5	5	5	6

Inserisce il NOR logico dei registri `rs` ed `rt` nel registro `rd`.

NOT

`not rdest, rsrc` pseudoistruzione

Inserisce la negazione logica bit a bit del registro `rsrc` nel registro `rdest`.

OR

`or rd, rs, rt`

0	rs	rt	rd	0	0x25
6	5	5	5	5	6

Inserisce l'OR logico dei registri `rs` ed `rt` nel registro `rd`.

OR con immediato

`ori rt, rs, imm`

0xd	rs	rt	imm
6	5	5	16

Inserisce nel registro `rt` l'OR logico del registro `rs` e dell'immediato, esteso con degli 0 a 32 bit.

Resto

`rem rdest, rsrc1, rsrc2` pseudoistruzione

Resto senza segno

`remu rdest, rsrc1, rsrc2` pseudoistruzione

Inserisce nel registro `rdest` il resto della devisione del registro `rsrc1` per il registro `rsrc2`. Si noti che se un operando è negativo, il resto nell'architettura MIPS non è specificato e dipende dalla convenzione della macchina sulla quale SPIM è in esecuzione.

Scorrimento logico a sinistra

`sll rd, rt, shamt`

0	rs	rt	rd	shamt	0
6	5	5	5	5	6

Scorrimento logico variabile a sinistra

`sllv rd, rt, rs`

0	rs	rt	rd	0	4
6	5	5	5	5	6

Scorrimento aritmetico a destra

`sra rd, rt, shamt`

0	rs	rt	rd	shamt	3
6	5	5	5	5	6

Scorrimento aritmetico variabile a destra

`srav rd, rt, rs`

0	rs	rt	rd	0	7
6	5	5	5	5	6

Scorrimento logico a destra

`srl rd, rt, shamt`

0	rs	rt	rd	shamt	2
6	5	5	5	5	6

Scorrimento logico variabile a destra

`srlv rd, rt, rs`

0	rs	rt	rd	0	6
6	5	5	5	5	6

Fa scorrere il registro `rt` a sinistra (destra) del numero di posizioni indicato dall'immediato `shamt` o dal registro `rs` e pone il risultato nel registro `rd`. Si noti che l'argomento `rs` viene ignorato dalle istruzioni `sll`, `sra` e `srl`.

Ruota a sinistra

`rol rdest, rsrc1, rsrc2` pseudoistruzione

Ruota a destra

`ror rdest, rsrc1, rsrc2` pseudoistruzione

Ruota il registro `rsrc1` a sinistra (destra) del numero di posizioni indicato nel registro `rsrc2` e pone il risultato nel registro `rdest`.

Sottrazione (con overflow)

`sub rd, rs, rt`

0	rs	rt	rd	0	0x22
6	5	5	5	5	6

Sottrazione (senza overflow)

`subu rd, rs, rt`

0	rs	rt	rd	0	0x23
6	5	5	5	5	6

Inserisce la differenza tra i registri `rs` e `rt` nel registro `rd`.

OR esclusivo

`xor rd, rs, rt`

0	rs	rt	rd	0	0x26
6	5	5	5	5	6

Inserisce nel registro `rd` l'XOR logico dei registri `rs` e `rt`.

OR esclusivo con immediato

`xori rt, rs, imm`

0xe	rs	rt	imm
6	5	5	16

Inserisce nel registro `rt` l'XOR logico del registro `rs` e dell'immediato, esteso a 32 bit con degli 0.

Istruzioni di manipolazione delle costanti

Carica immediato nella parte superiore

`lui rt, imm`

0xf	0	rt	imm
6	5	5	16

Carica la mezza parola corrispondente all'immediato `imm` nella mezza parola superiore del registro `rt`. I bit inferiori del registro vengono posti a 0.

Carica immediato

`li rdest, imm` pseudoistruzione

Copia l'immediato `imm` nel registro `rdest`.

Istruzioni di confronto

Imposta a 1 se inferiore a

`slt rd, rs, rt`

0	rs	rt	rd	0	0x2a
6	5	5	5	5	6

Imposta a 1 se inferiore a, senza segno

`sltu rd, rs, rt`

0	rs	rt	rd	0	0x2b
6	5	5	5	5	6

Imposta il registro `rd` a 1 se il contenuto del registro `rs` è inferiore a quello del registro `rt`, altrimenti lo pone uguale a 0.

Imposta a 1 se inferiore all'immediato

`slti rt, rs, imm`

Imposta a 1 se inferiore all'immediato, senza segno

`sltiu rt, rs, imm`

Imposta il registro `rd` a 1 se il contenuto del registro `rs` è inferiore a quello dell'immediato esteso di segno, altrimenti lo pone uguale a 0.

Imposta a 1 se uguale

`seq rdest, rscr1, rsrc2` pseudoistruzione

Imposta il registro `rdest` a 1 se il contenuto del registro `rsrc1` è uguale a quello del registro `rsrc2`, altrimenti lo pone uguale a 0.

Imposta a 1 se maggiore o uguale

`sge rdest, rscr1, rsrc2` pseudoistruzione

Imposta a 1 se maggiore o uguale, senza segno

`sgeu rdest, rscr1, rsrc2` pseudoistruzione

Imposta il registro `rdest` a 1 se il contenuto del registro `rsrc1` è maggiore o uguale a quello del registro `rsrc2`, altrimenti lo pone uguale a 0.

Imposta a 1 se maggiore

`sgt rdest, rscr1, rsrc2` pseudoistruzione

Imposta a 1 se maggiore, senza segno

`sgtu rdest, rscr1, rsrc2` pseudoistruzione

Imposta il registro `rdest` a 1 se il contenuto del registro `rsrc1` è maggiore di quello del registro `rsrc2`, altrimenti lo pone uguale a 0.

Imposta a 1 se minore o uguale

`sle rdest, rscr1, rsrc2` pseudoistruzione

Imposta a 1 se minore o uguale, senza segno

`sleu rdest, rscr1, rsrc2` pseudoistruzione

Imposta il registro `rdest` a 1 se il contenuto del registro `rsrc1` è minore o uguale a quello del registro `rsrc2`, altrimenti lo pone uguale a 0.

Imposta a 1 se diverso

`sne rdest, rscr1, rsrc2` pseudoistruzione

Imposta il registro `rdest` a 1 se il contenuto del registro `rsrc1` è diverso da quello del registro `rsrc2`, altrimenti lo pone uguale a 0.

Le istruzioni di salto condizionato

Le istruzioni di salto condizionato (branch) utilizzano un campo spiazzamento di 16 bit dotato di segno; ne consegue che esse possono saltare fino a 2^{15}–1 istruzioni (non byte) in avanti o 2^{15} istruzioni all'indietro. L'istruzione di salto incondizionato (jump) contiene un campo indirizzo di 26 bit. Nei processori MIPS reali, le istruzioni di salto condizionato sono implementate come salti ritardati: questi eseguono eventualmente il salto solo dopo che l'istruzione successiva al salto, contenuta nel suo delay slot, è stata eseguita (si veda il Capitolo 4).

I salti condizionati ritardati influenzano il calcolo dello spiazzamento, perché lo spiazzamento deve essere calcolato relativamente all'indirizzo dell'istruzione contenuta nel delay slot (PC + 4), cioè quando eventualmente si verifica il salto. SPIM non simula questo delay slot, a meno che non vengano specificati i flag `-bare` o `-delayed_branch`.

Nel codice assembler, lo spiazzamento di solito non viene specificato con un numero, ma l'istruzione salta a un'etichetta ed è l'assemblatore che calcola la distanza tra l'istruzione di branch e l'istruzione di arrivo.

Nell'architettura MIPS-32, tutte le istruzioni reali (non le pseudoistruzioni) di salto condizionato hanno una variante «probabile», per esempio la variante probabile di `beq` è `beql` che *non esegue* l'istruzione presente nello slot di ritardo se il salto non viene effettuato. Si sconsiglia di non utilizzare queste istruzioni: potrebbero essere rimosse nelle versioni successive dell'architettura. SPIM implementa queste istruzioni, ma non le descriveremo.

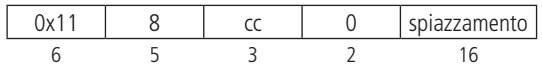
L'istruzione di salto incondizionato

`b label` pseudoistruzione

Salto incondizionato all'istruzione corrispondente all'etichetta.

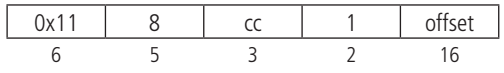
Salta se coprocessore è falso

`bclf cc label`



Salta se coprocessore è vero

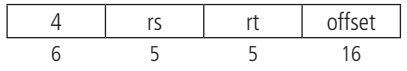
`bclt cc label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il flag numero *cc* del coprocessore delle operazioni in virgola mobile è falso (vero). Se *cc* è omesso nell'istruzione, viene considerato il flag associato al codice di condizione 0.

Salta se uguale

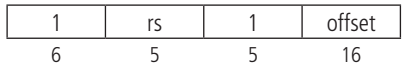
`beq rs, rt, label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il contenuto del registro `rs` è uguale a quello del registro `rt`.

Salta se maggiore o uguale a zero

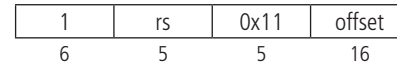
`bgez rs, label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il contenuto del registro `rs` è maggiore o uguale a 0.

Salta se maggiore o uguale a zero e collega

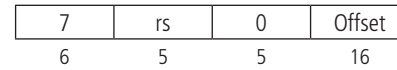
`bgezal rs, label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il contenuto del registro `rs` è maggiore o uguale a 0. Salva nel registro 31 l'indizzo dell'istruzione successiva.

Salta se maggiore a zero

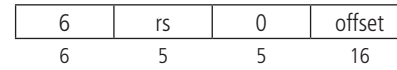
`bgtz rs, label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il contenuto del registro `rs` è maggiore di 0.

Salta se minore o uguale a zero

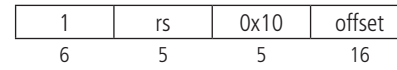
`blez rs, label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il contenuto del registro `rs` è minore o uguale a 0.

Salta se minore di zero e collega

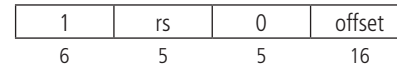
`bltzal rs, label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il contenuto del registro `rs` è minore di 0. Salva nel registro 31 l'indizzo dell'istruzione successiva.

Salta se minore di zero

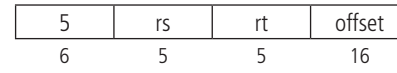
`bltz rs, label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il contenuto del registro `rs` è minore di 0.

Salta se diverso

`bne rs, rt, label`



Salta il numero di istruzioni specificato dallo spiazzamento `offset` se il contenuto del registro `rs` è diverso da quello del registro `rt`.

Salta se uguale a zero

`beqz rscr, label` pseudoistruzione

Salta all'istruzione corrispondente all'etichetta `label` se il contenuto del registro `rsrc` è uguale a 0.

Salta se maggiore o uguale

`bge rscr1, rsrc2, label` pseudoistruzione

Salta se maggiore o uguale, senza segno

`bgeu rscr1, rsrc2, label` pseudoistruzione

Salta all'istruzione corrispondente all'etichetta `label` se il contenuto del registro `rsrc1` è maggiore o uguale a quello di `rsrc2`.

Salta se maggiore

`bgt rscr1, rsrc2, label` pseudoistruzione

Salta se maggiore, senza segno

`bgtu rscr1, rsrc2, label` pseudoistruzione

Salta all'istruzione corrispondente all'etichetta `label` se il conteuto del registro `rsrc1` è maggiore di quello di `rsrc2`.

Salta se minore o uguale

`ble rscr1, rsrc2, label` pseudoistruzione

Salta se minore o uguale, senza segno

`bleu rscr1, rsrc2, label` pseudoistruzione

Salta all'istruzione corrispondente all'etichetta `label` se il contenuto del registro `rsrc1` è minore o uguale a `rsrc2`.

Salta se minore

`blt rscr1, rsrc2, label` pseudoistruzione

Salta se minore, senza segno

`bltu rscr1, rsrc2, label` pseudoistruzione

Salta all'istruzione corrispondente all'etichetta `label` se il contenuto del registro `rsrc1` è minore di quello di `rsrc2`.

Salta se diverso da zero

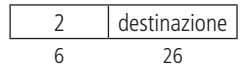
`bnez rscr, label` pseudoistruzione

Salta all'istruzione corrispondente all'etichetta `label` se il contenuto del registro `rsrc` è diverso da 0.

Istruzioni di salto incondizionato

Salto incondizionato (jump)

`j destinazione`



Salto incondizionato all'istruzione che si trova all'indirizzo destinazione.

Salta e collega

`jal destinazione`

3	destinazione
6	26

Salto incondizionato all'istruzione che si trova all'indirizzo destinazione. Salva l'indirizzo dell'istruzione successiva nel registro `$ra`.

Salto a registro e collega

`jalr rs, rd`

0	rs	0	rd	0	9
6	5	5	5	5	6

Salto incondizionato all'istruzione il cui indirizzo si trova nel registro `rs`. Salva l'indirizzo dell'istruzione successiva nel registro `$rd` (se non specificato, si considera il registro 31).

Salto a registro

`jr rs`

0	rs	0	8
6	5	15	6

Salto incondizionato all'istruzione il cui indirizzo si trova nel registro `rs`.

Istruzioni di trap

Trap se uguale

`teq rs, rt`

0	rs	rt	0	0x34
6	5	5	10	6

Se il contenuto del registro `rs` è uguale a quello del registro `rt`, solleva un'eccezione di trap.

Trap se uguale a immediato

`teqi rs, imm`

1	rs	0xc	imm
6	5	5	16

Se il contenuto del registro `rs` è uguale al valore immediato `imm` esteso con il segno, solleva un'eccezione di trap.

Trap se diverso

`tne rs, rt`

0	rs	rt	0	0x36
6	5	5	10	6

Se il contenuto del registro `rs` è diverso da quello del registro `rt`, solleva un'eccezione di trap.

Trap se diverso da immediato

`tnei rs, imm`

1	rs	0xe	imm
6	5	5	16

Se il contenuto del registro `rs` è diverso dal valore immediato `imm` esteso con il segno, solleva un'eccezione di trap.

Trap se maggiore o uguale

`tge rs, rt`

0	rs	rt	0	0x30
6	5	5	10	6

Trap se maggiore o uguale, senza segno

`tgeu rs, rt`

0	rs	rt	0	0x31
6	5	5	10	6

Se il contenuto del registro `rs` è maggiore o uguale a quello del registro `rt`, solleva un'eccezione di trap.

Trap se maggiore o uguale a immediato

`tgei rs, imm`

1	rs	8	imm
6	5	5	16

Trap se maggiore o uguale a immediato, senza segno

`tgeiu rs, imm`

1	rs	9	imm
6	5	5	16

Se il contenuto del registro `rs` è maggiore o uguale al valore immediato `imm` esteso con il segno, solleva un'eccezione di trap.

Trap se minore

`tlt rs, rt`

0	rs	rt	0	0x32
6	5	5	10	6

Trap se minore, senza segno

`tltu rs, rt`

0	rs	rt	0	0x33
6	5	5	10	6

Se il contenuto del registro `rs` è minore di quello del registro `rt`, solleva un'eccezione di trap.

Trap se minore di immediato

`tlti rs, imm`

1	rs	0xa	imm
6	5	5	16

Trap se minore di immediato, senza segno

`tltiu rs, imm`

1	rs	0xb	imm
6	5	5	16

Se il contenuto del registro `rs` è minore del valore immediato `imm` esteso con il segno, solleva un'eccezione di trap.

Istruzioni di lettura dalla memoria (load)

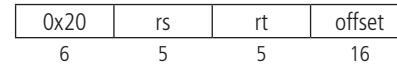
Carica un indirizzo

`la rdest, address` pseudoistruzione

Carica il valore dell'indirizzo calcolato (non il contenuto della locazione a tale indirizzo) nel registro `rdest`.

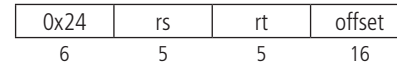
Leggi un byte

`lb rt, address`



Leggi un byte senza segno

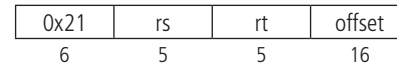
`lbu rt, address`



Carica il byte che si trova all'indirizzo `address` nel registro `rt`. Il byte viene esteso con il segno dall'istruzione `lb` ma non da `lbu`.

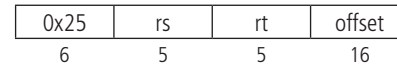
Leggi una mezza parola

`lh rt, address`



Leggi una mezza parola senza segno

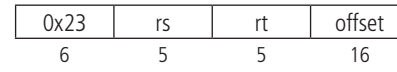
`lhu rt, address`



Carica un numero a 16 bit (mezza parola) all'indirizzo `address` nel registro `rt`. La mezza parola viene estesa con il segno dall'istruzione `lh` ma non da `lhu`.

Leggi una parola

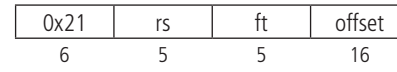
`lw rt, address`



Carica un numero a 32 bit (parola) che si trova all'indirizzo `address` nel registro `rt`.

Leggi una parola nel coprocessore 1

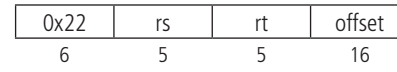
`lwc1 ft, address`



Carica la parola che si trova all'indirizzo `address` nel registro `ft` dell'unità a virgola mobile.

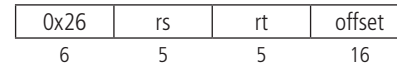
Leggi una parola a sinistra

`lwl rt, address`



Leggi una parola a destra

`lwr rt, address`



Carica nel registro `rt` i byte a sinistra (destra) della parola che si trova all'indirizzo `address`, che può essere disallineato.

Leggi una parola doppia

`ld rdest, address` pseudoistruzione

Carica un numero a 64 bit all'indirizzo `address` nei registri `rdest` e `rdest+1`.

Lettura di una mezza parola disallineata

`ulh rdest, address` pseudoistruzione

Lettura di una mezza parola senza segno e disallineata

`ulhu rdest, address` pseudoistruzione

Carica nel registro `rdest` un numero a 16 bit (mezza parola) che si trova all'indirizzo `address`, che può essere disallineato. La mezza parola viene estesa con il segno dall'istruzione `ulh` ma non da `ulhu`.

Lettura di una parola disallineata

`ulw rdest, address` pseudoistruzione

Carica nel registro `rdest` un numero a 32 bit (parola) che si trova all'indirizzo `address`, che può essere disallineato.

Lettura di una parola e blocco

`ll rt, address`

0x30	rs	Rt	offset
6	5	5	16

Carica nel registro `rt` la parola che si trova all'indirizzo `address` e avvia un'operazione atomica di lettura-modifica-scrittura. Questa operazione è completata da un'istruzione di scrittura condizionata (`sc`), che fallisce se un altro processore scrive nel blocco contenente la parola precedentemente caricata. Dato che SPIM non simula processori multipli, l'operazione di scrittura condizionata ha sempre successo.

Istruzioni di scrittura in memoria (store)

Memorizza un byte

`sb rt, address`

0x28	rs	rt	offset
6	5	5	16

Scrive il byte inferiore del registro `rt` all'indirizzo `address`.

Memorizza una mezza parola

`sh rt, address`

0x29	rs	rt	offset
6	5	5	16

Scrive la mezza parola inferiore del registro `rt` all'indirizzo `address`.

Memorizza una parola

`sw rt, address`

0x2b	rs	rt	offset
6	5	5	16

Scrive la parola nel registro `rt` all'indirizzo `address`.

Memorizza una parola nel coprocessore 1

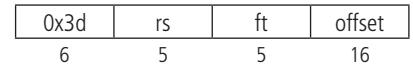
`swc1 ft, address`

0x31	rs	ft	offset
6	5	5	16

Scrive il numero in virgola mobile contenuto nel registro `ft` dell'unità a virgola mobile all'indirizzo `address`.

Memorizza una parola doppia nel coprocessore 1

`sdc1 ft, address`



Scrive la parola doppia rappresentante un numero in virgola mobile nei registri `ft` e `ft+1` dell'unità a virgola mobile all'indirizzo `address`. Il registro `ft` deve essere identificato da un numero pari.

Memorizza una parola a sinistra

`swl rt, address`

Memorizza una parola a destra

`swr rt, address`

Scrive i byte a sinistra (destra) del registro `rt` all'indirizzo `address`, che può essere disallineato.

Memorizza una parola doppia

`sd rsrc, address` pseudoistruzione

Scrive il numero a 64 bit presente nei registri `rsrc` e `rsrc+1` all'indirizzo `address`.

Memorizzazione di una mezza parola, disallineata

`ush rsrc, address` pseudoistruzione

Scrive la mezza parola inferiore contenuta nel registro `rsrc` all'indirizzo `address`, che può essere disallineato.

Memorizzazione di una parola disallineata

`usw rsrc, address` pseudoistruzione

Scrive la parola contenuta nel registro `rsrc` all'indirizzo `address`, che può essere disallineato.

Memorizzazione condizionata

`sc rt, address`

Scrive il numero a 32 bit (parola) presente nel registro `rt` all'indirizzo `address` e completa un'operazione atomica di lettura-modifica-scrittura. Se questa operazione atomica termina con successo, la parola di memoria contenuta in `address` viene modificata e il registro `rt` viene impostato a 1. Se l'operazione atomica fallisce perché un altro processore ha scritto in una locazione nel blocco contenente la parola indirizzata, questa istruzione non modifica la memoria e scrive 0 nel registro `rt`. Dato che SPIM non simula processori multipli, l'istruzione ha sempre successo.

Le istruzioni di spostamento dati

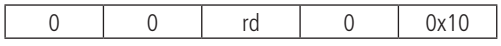
Sposta

`move rdest, rsrc` pseudoistruzione

Copia il contenuto del registro `rsrc` nel registro `rdest`.

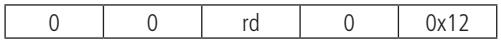
Sposta da hi

`mfhi rd`



Sposta da lo

`mflo rd`

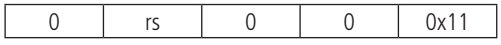


L'unità di moltiplicazione e divisione produce i propri risultati in due registri dedicati: `hi` e `lo`. Queste istruzioni spostano numeri da e verso tali registri. Le pseudoistruzioni di moltiplicazione, divisione e resto fanno sembrare che questa unità operi sui registri generici, in realtà il risultato viene spostato subito dopo il completamento del calcolo.

Copia il contenuto del registro `hi`(`lo`) nel registro `rd`.

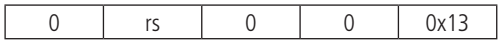
Sposta in hi

`mthi rs`



Sposta in lo

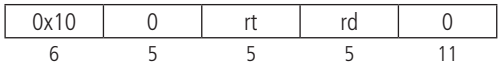
`mtlo rs`



Copia il contenuto del registro `rs` nel registro `hi`(`lo`).

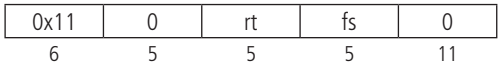
Sposta dal coprocessore 0

`mfc0 rt, rd`



Sposta dal coprocessore 1

`mfc1 rt, fs`



I coprocessori possiedono ciascuno un proprio set di registri. Queste istruzioni spostano i numeri tra tali registri e i registri della CPU.
Copia il contenuto del registro `rd` del coprocessore (il registro `fs` nella FPU) nel registro `rt` della CPU. L'unità in virgola mobile (FPU) è il coprocessore 1.

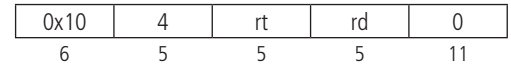
Sposta una parola doppia dal coprocessore 1

`mfc1.d rdest, frsrc1` pseudoistruzione

Copia il contenuto dei registri a virgola mobile `fsrc1` e `fsrc1+1` nei registri di CPU `rdest` e `rdest+1`.

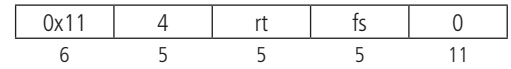
Sposta al coprocessore 0

`mtc0 rd, rt`



Sposta al coprocessore 1

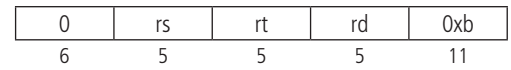
`mtc1 rd, fs`



Copia il contenuto del registro `rt` della CPU nel registro `rd` del coprocessore (registro `fs` nella FPU).

Sposta se la condizione non vale 0

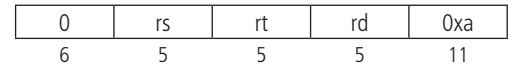
`movn rd, rs, rt`



Copia il contenuto del registro `rs` nel registro `rd` se il contenuto del registro `rt` è diverso da 0.

Sposta se la condizione vale 0

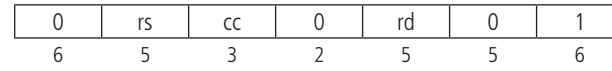
`movz rd, rs, rt`



Copia il contenuto del registro `rs` nel registro `rd` se il contenuto del registro `rt` è uguale a 0.

Sposta se la condizione su numeri in virgola mobile risulta falsa

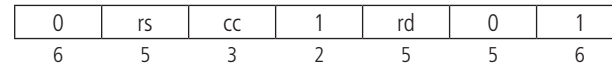
`movf rd, rs, cc`



Copia il contenuto del registro `rs` nel registro `rd` se il flag del codice di condizione numero `cc` è uguale a 0. Se `cc` viene omesso, si considera il flag di codice di condizione 0.

Sposta se la condizione su numeri in virgola mobile risulta vera

`movt rd, rs, cc`



Copia il contenuto del registro `rs` nel registro `rd` se il flag del codice di condizione numero `cc` è uguale a 1. Se `cc` viene omesso, si considera il flag di codice di condizione 0.

Istruzioni su numeri in virgola mobile

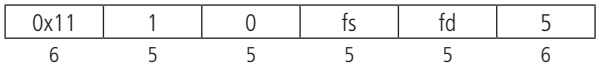
Il MIPS possiede un coprocessore per operazioni in virgola mobile (identificato dal numero 1) che opera su numeri in virgola mobile in singola (32 bit) e in doppia precisione (64 bit). Questo coprocessore possiede dei propri registri enumerati da `$f0` a `$f31`. Essendo questi registri larghi solamente 32 bit, ne servono due per contenere un numero in doppia precisione, per cui soltanto i registri pari possono memorizzare i numeri in doppia precisione. Il coprocessore in virgola mobile contiene anche otto flag associati a condizioni particolari (codici di condizione, cc), numerati da 0 a 7, i quali vengono impostati dalle istruzioni di confronto e interrogati dalle istruzioni di salto condizionato (`bclf` o `bclt`) e di spostamento condizionato.

I numeri vengono copiati da e in questi registri, una parola (32 bit) alla volta, dalle istruzioni `lwc1`, `swc1`, `mtc1` e `mfc1`, oppure una parola doppia (64 bit) alla volta, dalle istruzioni `ldc1` e `sdc1` descritte in precedenza, o dalle pseudoistruzioni `l.s`, `l.d`, `s.s`, `s.d` descritte di seguito.

Nelle istruzioni reali seguenti, i bit 21-26 contengono 0 per le operazioni in singola precisione e 1 per quelle in doppia precisione. Nelle pseudoistruzioni, `fdest` è un registro per la virgola mobile (per esempio `$f2`).

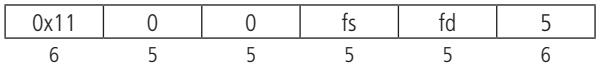
Valore assoluto in virgola mobile in doppia precisione

`abs.d fd, fs`



Valore assoluto in virgola mobile in singola precisione

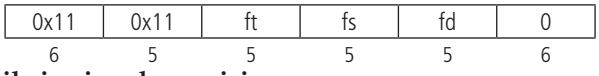
`abs.s fd, fs`



Calcola il valore assoluto del numero in doppia (singola) precisione contenuto nel registro `fs` e lo inserisce nel registro `fd`.

Addizione in virgola mobile in doppia precisione

`add.d fd, fs`



Addizione in virgola mobile in singola precisione

`add.s fd, fs`

Calcola la somma dei numeri in doppia (singola) precisione contenuti nei registri `fs` e `ft` e la inserisce nel registro `fd`.

Arrotondamento in virgola mobile all'intero superiore di una parola

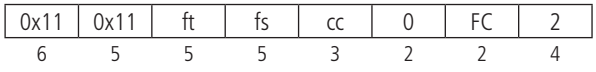
`ceil.w.d fd, fs`

`ceil.w.s fd, fs`

Calcola l'intero superiore al numero in doppia (singola) precisione contenuto nel registro `fs` e lo converte in un numero a 32 bit che viene inserito nel registro `fd`.

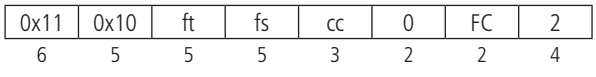
Confronta se uguale in doppia precisione

`c.eq.d cc, fs, fd`



Confronta se uguale in singola precisione

`c.eq.s cc, fs, fd`



Confronta il numero in doppia (singola) precisione contenuto nel registro `fs` con quello contenuto in `ft` e imposta a 1 il flag dei codici di condizione numero `cc`, se sono uguali. Se `cc` viene omesso, si considera il flag dei codici di condizione numero 0.

Confronta se minore o uguale in doppia precisione

`c.le.d cc, fs, fd`

0x11	0x11	ft	fs	cc	0	FC	0xe
6	5	5	5	3	2	2	4

Confronta se minore o uguale in singola precisione

`c.le.s cc, fs, fd`

0x11	0x10	ft	fs	cc	0	FC	0xe
6	5	5	5	3	2	2	4

Confronta il numero in doppia (singola) precisione contenuto nel registro `fs` con quello contenuto in `ft` e imposta a 1 il flag dei codici di condizione numero `cc`, se il primo numero è minore o uguale al secondo. Se `cc` viene omesso, si considera il flag dei codici di condizione numero 0.

Confronta se minore in doppia precisione

`c.lt.d cc, fs, fd`

0x11	0x11	ft	fs	cc	0	FC	0xc
6	5	5	5	3	2	2	4

Confronta se minore in singola precisione

`c.lt.s cc, fs, fd`

0x11	0x10	ft	fs	cc	0	FC	0xc
6	5	5	5	3	2	2	4

Confronta il numero in doppia (singola) precisione contenuto nel registro `fs` con quello contenuto in `ft` e imposta a 1 il flag dei codici di condizione numero `cc`, se il primo numero è minore del secondo. Se `cc` viene omesso, si considera il flag dei codici di condizione numero 0.

Converte da singola a doppia precisione

`cvt.d.s fd, fs`

0x11	0x10	0	fs	fd	0x21
6	5	5	5	5	6

Converte da intero a doppia precisione

`cvt.d.w fd, fs`

0x11	0x14	0	fs	fd	0x21
6	5	5	5	5	6

Converte il numero in virgola mobile in singola precisione o intero, contenuto nel registro `rs`, in un numero in virgola mobile in doppia precisione e lo pone nel registro `fd`.

Converte da doppia a singola precisione

`cvt.s.d fd, fs`

0x11	0x11	0	fs	fd	0x20
6	5	5	5	5	6

Converte da intero a singola precisione

`cvt.s.w fd, fs`

0x11	0x14	0	fs	fd	0x20
6	5	5	5	5	6

Converte il numero in virgola mobile in doppia precisione o intero, contenuto nel registro `rs`, in un numero in virgola mobile in singola precisione e lo pone nel registro `fd`.

Converte da doppia precisione a intero

`cvt.w.d fd, fs`

0x11	0x11	0	fs	fd	0x24
6	5	5	5	5	6

Converte da singola precisione a intero

`cvt.w.s fd, fs`

0x11	0x10	0	fs	fd	0x24
6	5	5	5	5	6

Converte il numero in virgola mobile in singola o doppia precisione contenuto nel registro `rs` in un intero e lo pone nel registro `fd`.

Divisione in virgola mobile in doppia precisione

`div.d fd, fs, ft`

0x11	0x11	ft	fs	fd	3
6	5	5	5	5	6

Divisione in virgola mobile in singola precisione

`div.s fd, fs, ft`

0x11	0x10	ft	fs	fd	3
6	5	5	5	5	6

Calcola il quoziente dei numeri in virgola mobile in doppia (singola) precisione contenuti nei registri `fs` e `ft` e lo pone nel registro `fd`.

Arrotondamento in virgola mobile all'intero inferiore di una parola

`floor.w.d fd, fs`

0x11	0x11	0	fs	fd	0xf
6	5	5	5	5	6

`floor.w.s fd, fs`

0x11	0x10	0	fs	fd	0xf
6	5	5	5	5	6

Calcola l'intero inferiore del numero in doppia (singola) precisione contenuto nel registro `fs`, lo converte in un numero intero a 32 bit e pone la parola risultante nel registro `fd`.

Carica un numero in virgola mobile in doppia precisione

`l.d fdest, address` pseudoistruzione

Carica un numero in virgola mobile in singola precisione

`l.s fdest, address` pseudoistruzione

Carica il numero in virgola mobile in doppia (singola) precisione che si trova all'indirizzo `address` nel registro `fdest`.

Spostamento in virgola mobile in doppia precisione

`mov.d fd, fs`

0x11	0x11	0	fs	fd	6
6	5	5	5	5	6

Spostamento in virgola mobile in singola precisione

`mov.s fd, fs`

0x11	0x10	0	fs	fd	6
6	5	5	5	5	6

Copia il numero in virgola mobile in doppia (singola) precisione dal registro `fs` al registro `fd`.

Spostamento su condizione falsa in virgola mobile in doppia precisione

`movf.d fd, fs, cc`

0x11	0x11	cc	0	fs	fd	0x11
6	5	3	2	5	5	6

Spostamento su condizione falsa in virgola mobile in singola precisione

`movf.s fd, fs, cc`

0x11	0x10	cc	0	fs	fd	0x11
6	5	3	2	5	5	6

Copia il numero in virgola mobile in doppia (singola) precisione dal registro `fs` al registro `fd` se il flag dei codici di condizione numero `cc` è 0. Se `cc` viene omesso, si considera il flag dei codici di condizione numero 0.

Spostamento su condizione vera in virgola mobile in doppia precisione

`movt.d fd, fs, cc`

0x11	0x11	cc	1	fs	fd	0x11
6	5	3	2	5	5	6

Spostamento su condizione vera in virgola mobile in singola precisione

`movt.s fd, fs, cc`

0x11	0x10	cc	1	fs	fd	0x11
6	5	3	2	5	5	6

Copia il numero in virgola mobile in doppia (singola) precisione dal registro `fs` al registro `fd` se il flag dei codici di condizione numero `cc` è 1. Se `cc` viene omesso, viene considerato il flag dei codici di condizione numero 0.

Spostamento su condizione «diverso da zero» in virgola mobile in doppia precisione

`movn.d fd, fs, rt`

0x11	0x11	rt	fs	fd	0x13
6	5	5	5	5	6

Spostamento su condizione «diverso da zero» in virgola mobile in singola precisione

`movn.s fd, fs, rt`

0x11	0x10	rt	fs	fd	0x13
6	5	5	5	5	6

Copia il numero in virgola mobile in doppia (singola) precisione dal registro `fs` al registro `fd` se il contenuto del registro della CPU `rt` è diverso da 0.

Spostamento su condizione «uguale a zero» in virgola mobile in doppia precisione

`movz.d fd, fs, rt`

0x11	0x11	rt	Fs	fd	0x12
6	5	5	5	5	6

Spostamento su condizione «uguale a zero» in virgola mobile in singola precisione

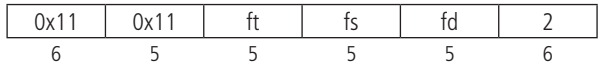
`movz.s fd, fs, rt`

0x11	0x10	rt	fs	fd	0x12
6	5	5	5	5	6

Copia il numero in virgola mobile in doppia (singola) precisione dal registro `fs` al registro `fd` se il contenuto del registro della CPU `rt` è uguale a 0.

Moltiplicazione in virgola mobile in doppia precisione

`mul.d fd, fs, ft`



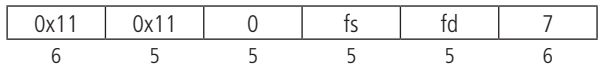
Moltiplicazione in virgola mobile in singola precisione

`mul.s fd, fs, ft`

Calcola il prodotto dei numeri in virgola mobile in doppia (singola) precisione contenuti nei registri `fs` e `ft` e lo pone nel registro `fd`.

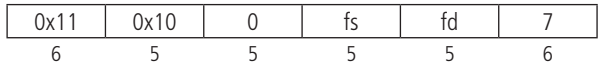
Negativo in virgola mobile in doppia precisione

`neg.d fd, fs`



Negativo in virgola mobile in singola precisione

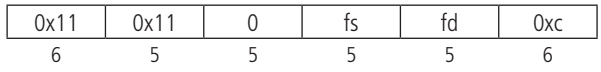
`neg.s fd, fs`



Inserisce nel registro `fd` il negativo del numero in virgola mobile in doppia (singola) precisione contenuto nel registro `fs`.

Arrotondamento in virgola mobile all'intero più vicino di una parola

`round.w.d fd, fs`

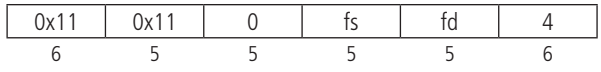


`round.w.s fd, fs`

Calcola l'intero più vicino del numero in doppia (singola) precisione contenuto nel registro `fs` e lo converte in un numero intero a 32 bit che viene posto nel registro `fd`.

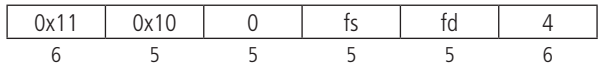
Radice quadrata in virgola mobile in doppia precisione

`sqrt.d fd, fs`



Radice quadrata in virgola mobile in singola precisione

`sqrt.s fd, fs`



Calcola la radice quadrata del numero in doppia (singola) precisione contenuto nel registro `fs` e la pone nel registro `fd`.

Copia un numero in virgola mobile in doppia precisione

`s.d fdest, address` pseudoistruzione

Copia un numero in virgola mobile in singola precisione

`s.s fdest, address` pseudoistruzione

Scrive nel registro `fdest` il numero in virgola mobile in doppia (singola) precisione che si trova all'indirizzo `address`.

Sottrazione in virgola mobile in doppia precisione

`sub.d fd, fs, ft`

0x11	0x11	ft	fs	fd	1
6	5	5	5	5	6

Sottrazione in virgola mobile in singola precisione

`sub.s fd, fs, ft`

0x11	0x10	ft	fs	fd	1
6	5	5	5	5	6

Calcola la differenza in virgola mobile dei numeri in doppia (singola) precisione contenuti nei registri `fs` e `ft` e la pone nel registro `fd`.

Tronca un numero in virgola mobile a una parola

`trunc.w.d fd, fs`

0x11	0x11	0	fs	fd	0xd
6	5	5	5	5	6

`trunc.w.s fd, fs`

0x11	0x10	0	fs	fd	0xd
6	5	5	5	5	6

Tronca il numero in doppia (singola) precisione presente nel registro `fs` convertendolo in un numero intero su 32 bit che viene inserito nel registro `fd`.

Istruzioni di gestione di eccezioni e interrupt

Ritorno da eccezione

`eret`

0x10	1	0	0x18
6	1	19	6

Imposta a 1 il bit EXL nel registro Stato del coprocessore 0 e ritorna all'istruzione a cui punta il registro EPC del coprocessore 0.

Chiamata di sistema

`syscall`

0	0	0xc
6	20	6

Il registro `$v0` contiene il numero della chiamata di sistema (si veda la Figura A.9.1) tra quelle messe a a disposizione da SPIM.

Break

`break code`

0	code	0xd
6	20	6

Causa l'eccezione il cui numero è `code`. L'eccezione 1 è riservata al programma di debug.

Nessuna operazione

`nop`

0	0	0	0	0	0
6	5	5	5	5	6

Non effettua alcuna operazione.

A.11 Considerazioni conclusive

Programmare in linguaggio assembler impone al programmatore di rinunciare ad alcune caratteristiche utili dei linguaggi ad alto livello, come le strutture dati, la verifica dei tipi e i costrutti di controllo di flusso, in cambio del completo controllo sulle istruzioni che il calcolatore esegue. I vincoli esterni di certe applicazioni, come il tempo di risposta o la dimensione del programma, richiedono al programmatore di dedicare molta attenzione a ogni singola istruzione. Tuttavia, il costo di tale livello di attenzione è rappresentato dalla lunghezza e dall'elevata complessità dei programmi assembler.

Inoltre, i recenti sviluppi relativi a tre aspetti delle architetture stanno progressivamente riducendo la necessità di scrivere programmi in assembler. Il primo è quello dei compilatori: i compilatori moderni producono codice che generalmente è all'altezza del miglior codice scritto a mano e in certi casi è addirittura migliore. Il secondo aspetto è quello dell'elaborazione parallela: i nuovi processori, recentemente introdotti, non sono soltanto più veloci ma sono anche in grado di eseguire più istruzioni contemporaneamente, e quindi sono anche più difficili da programmare a basso livello. Per di più, la rapida evoluzione dei processori moderni favorisce i programmi scritti nei linguaggi ad alto livello, i quali non sono legati a una singola architettura. Infine, la tendeza attuale è di scrivere applicazioni sempre più articolate, caratterizzate da interfacce grafiche complesse e da un maggior numero di funzionalità rispetto alle applicazioni del passato. Queste applicazioni di grandi dimensioni vengono scritte da vere e proprie squadre di programmatori; devono quindi essere modulari e necessitano delle funzioni di controllo semantico offerte dai linguaggi ad alto livello.

Letture ulteriori

A. Aho, R. Sethi and J. Ullman (1985). *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.
Il testo di rifermento per i compilatori, anche se leggermente datato.

D. Sweetman (1999). See MIPS Run, San Francisco, CA: Morgan Kaufman Publishers.
Un'introduzione completa, dettagliata e affascinante all'insieme di istruzioni del MIPS e alla programmazione in linguaggio assembler di queste macchine.

Documentazione più dettagliata sull'architettura MIPS-32 è disponibile sul sito Web della MIPS (*http://www.mips.com/products/product-materials/processor/mips-architecture*).

MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS32™ Architecture.

MIPS32® Architecture for Programmers Volume II: The MIPS32® Instruction Set.

MIPS32® Architecture for Programmers Volume III: The MIPS32® Privileged Resource Architecture.

A.12 Esercizi

A.1 [5] <A.5> Il Paragrafo A.5 descrive come viene partizionata la memoria nella maggior parte dei sistemi MIPS. Proporre un'altra maniera di dividere la memoria per raggiungere gli stessi obiettivi.

A.2 [20] <A.6> Riscrivere il codice della funzione `fact` utilizzando meno istruzioni.

A.3 [5] <A.7> È sempre sicuro utilizzare i registri `$k0` o `$k1` in un programma utente?

A.4 [25] <A.7> Il Paragrafo A.7 presenta il codice di un gestore delle eccezioni molto semplice. Un problema serio di questo gestore è che disabilita gli interrupt per molto tempo: ciò significa che gli interrupt provenienti da un dispositivo di I/O veloce potrebbero andare persi. Scrivere un gestore delle eccezioni migliore, che non si possa interrompere e che abiliti gli interrupt prima possibile.

A.5 [15] <A.7> Il gestore delle eccezioni semplificato torna sempre all'istruzione che segue quella che ha generato l'eccezione. Ciò funziona bene fintanto che l'istruzione che causa l'eccezione non si trova nel delay slot di un salto condizionato. In questo caso, l'istruzione successiva è l'istruzione di destinazione del salto. Scrivere un gestore migliore che utilizzi il registro EPC per determinare quale istruzione debba essere eseguita dopo l'eccezione.

A.6 [5] <A.9> Utilizzando SPIM, scrivere e provare un programma che implementa una macchina sommatrice, la quale legge ripetutamente un intero e lo aggiunge alla somma corrente. Il programma si deve fermare quando riceve in ingresso il valore 0; in questo caso deve stampare la somma calcolata fino a quel momento. Utilizzare le chiamate di sistema di SPIM descritte nel Paragrafo A.9.

A.7 [5] <.9> Utilizzando SPIM, scrivere e provare un programma che legge tre interi e stampa la somma dei due interi maggiori. Utilizzare le chiamate di sistema di SPIM descritte nel Paragrafo A.9. Si possono gestire in modo arbitrario i casi di uguaglianza tra i numeri.

A.8 [5] <A.9> Utilizzando SPIM, scrivere e provare un programma che legge un intero positivo utilizzando le chiamate di sistema di SPIM. Se l'intero non è positivo, il programma deve terminare stampando il messaggio «Input non valido»; altrimenti il programma deve stampare i nomi delle cifre dell'intero letto, delimitati esattamente da uno spazio. Per esempio, se l'utente inserisce «728», l'uscita dovrà essere: «Sette Due Otto».

A.9 [25] <A.9> Scrivere e provare un programma nel linguaggio assembler MIPS per calcolare i primi 100 numeri primi. Un numero n è detto primo se nessun numero, eccetto 1 e n, è suo divisore intero. Si implementino due procedure:

- `test_primo(n)` Restituisce 1 se n è primo e 0 se non lo è.
- `main(n)` Itera sugli interi, controllando se l'intero corrente sia un numero primo. Stampa i primi 100 numeri che risultano primi.

Provare i programmi eseguendoli su SPIM.

A.10 [10] <A.6, A.9> Utilizzando SPIM, scrivere e provare un programma ricorsivo per risolvere il problema matematico classico detto «La torre di Hanoi». Per supportare la ricorsione sarà necessario utilizzare i frame di stack. Il gioco consiste di tre pioli (1, 2 e 3) e n dischi, dove il numero n può variare; valori tipici di n sono compresi tra 1 e 8. Il disco 1 è più piccolo del disco 2 che, a sua volta, è più piccolo del disco 3, e così via. Inizialmente tutti i dischi si trovano sul piolo 1, con il disco n sul fondo, il disco $n-1$ sopra di esso e così via fino al disco 1, che si trova sopra tutti gli altri. Scopo del gioco è portare tutti i dischi sul piolo 2. È possibile muovere solo un disco alla volta, il che significa spostare un disco dalla cima di uno dei pioli e metterlo in cima ai dischi già posti sugli altri due pioli. Inoltre c'è un vincolo: non si può posare un disco su un altro disco più piccolo.

Il seguente programma in C può essere utilizzato come guida per scrivere il programma in linguaggio assembler.

```
/* sposta gli n dischi più piccoli da inizio a fine utilizzando extra */

void hanoi(int n, int inizio, int fine, int extra) {
   if(n != 0) {
       hanoi(n-1, inizio, extra, fine);
       print_string("Sposta il disco");
       print_int(n);
       print_string("dal piolo");
       print_int(inizio);
       print_string("al piolo");
       print_int(fine);
       print_string(".\n");
       hanoi(n-1, extra, fine, inizio);
   }
   }

main() {
   int n;
   print_string("Inserisci il numero di
   dischi > ");
   n = read_int();
   hanoi(n, 1, 2, 3);
   return 0;
   }
```