Piccola Introduzione al Linguaggio Assembly e Simili Amenità

Luca Abeni

20 aprile 2016

Capitolo 1

Il Linguaggio Assembly

1.1 Introduzione

La CPU (Central Processing Unit) di un computer funziona eseguendo programmi scritti in *linguaggio macchina*, composti da istruzioni che la CPU ciclicamente legge ed esegue. Tali istruzioni macchina, codificate come sequenze di 0 e 1, sono le uniche che la CPU può capire ed eseguire correttamente; quindi, per essere eseguito ogni programma deve essere in qualche modo "tradotto" in linguaggio macchina. Questa traduzione viene fatta automaticamente da appositi programmi (*compilatori* o *interpreti*) ed il linguaggio macchina non viene mai usato direttamente per programmare; quando è necessario controllare direttamente le istruzioni eseguite dalla CPU si utilizza invece il linguaggio *Assembly*.

Il linguaggio Assembly è un linguaggio di basso livello, strettamente legato al linguaggio macchina, che codifica le istruzioni macchina tramite codici mnemonici (parole) che sono più semplici da ricordare ed utilizzare rispetto alle sequenze di 0 e 1.

Un programma Assembly è un file ASCII composto da una sequenza di istruzioni Assembly (codici mnemonici) che codificano le istruzioni macchina da eseguire. Per essere eseguito, un programma Assembly deve essere quindi tradotto in linguaggio macchina (assemblato) da un apposito programma compilatore, chiamato *Assembler*.

Salvo rare eccezioni che verranno chiarite più tardi (pseudo-istruzioni, riordinamento di istruzioni eseguite fuori ordine dalla CPU, label per i salti, ...) esiste una corrispondenza biunivoca fra istruzioni Assembly e istruzioni macchina ed il loro ordine. La struttura e la semantica di un programma Assembly sono quindi dipendenti da struttura e funzionamento della CPU. Come conseguenza, il linguaggio Assembly stesso è strettamente legato alla CPU su cui il programma dovrà eseguire e non è portabile; sebbene si parli genericamente di "linguaggio Assembly", non esiste quindi un singolo linguaggio ma un insieme di linguaggi (uno per ogni differente modello di CPU - talvolta anche più di un linguaggio per CPU) che vanno sotto il generico nome di "Assembly".

Ogni modello di CPU è caratterizzato dall'insieme delle istruzioni macchina che la CPU riconosce e sa eseguire, detto *Instruction Set Architecture* (ISA). Le istruzioni macchina che compongono l'ISA sono descritte dalla loro *sintassi* (la codifica binaria) e *semantica* (che ne descrive il comportamento). L'ISA è quindi definito non solo dai nomi delle varie istruzioni che lo compongono, ma anche da una descrizione del loro comportamento e del loro modo di accedere ai dati che manipolano. Mentre quando si usa un linguaggio di alto livello non è possibile specificare come e dove sono memorizzati i dati (per esempio, i programmi scritti usando un linguaggio imperativo come Java o il linguaggio C / C++ manipolano dati memorizzati in *variabili*), quando si programma in Assembly i dati possono essere memorizzati nella memoria principale del computer (memoria RAM) o in *registri* interni alla CPU. Ogni CPU contiene infatti un banco di registri, che è una memoria composta da un piccolo numero di registri (aventi tutti le stesse dimensioni in bit) a cui le istruzioni Assembly possono accedere molto velocemente. Tale memoria, che può essere acceduta specificando il numero (o nome) del registro su cui si vuole lavorare, è caratterizzata dalle sue piccole dimensioni. Riassumendo, le istruzioni Assembly manipolano quindi dati contenuti nei registri (accessibili in modo veloce) e/o in locazioni della memoria principale (accessibili attaverso il bus, quindi in modo più lento). Questa può essere vista come una prima forma di gerarchia delle memorie (anche se la memoria veloce - costituita dal banco dei registri - non è "trasparente" al programmatore, ma ogni programma Assembly deve esplicitamente gestire gli accessi a registri e/o memoria principale).

Le istruzioni Assembly vengono generalmente eseguite sequenzialmente, ma questa esecuzione sequenziale può essere modificata da apposite *istruzioni di salto* (come ogni linguaggio di basso livello, Assembly

non prevede costrutti strutturati come cicli e selezioni, ma solo istruzioni di salto). Le istruzioni Assembly devono permettere di:

1. Effettuare operazioni aritmetiche e logiche sui dati
2. Muovere dati fra memoria e registri della CPU
3. Modificare l'esecuzione sequenziale dei dati (istruzioni di salto)
4. Eseguire vari tipi di operazioni (almeno operazioni di salto) dipendentemente da determinate condizioni

Mentre le operazioni aritmetiche e logiche (punto 1) sono circa le stesse per ogni tipo di CPU, le loro modalità di esecuzione variano fortemente a seconda dell'ISA: in alcune architetture, tali istruzioni possono agire su dati contenuti in memoria (leggendo gli operandi dalla memoria o salvando i risultati in memoria), mentre in altre architetture le istruzioni aritmetiche e logiche possono solo lavorare su operandi contenuti in registri della CPU e salvare i risultati in registri.

La modalità di esecuzione delle istruzioni aritmetiche e logiche ha chiaramente impatto anche sulle istruzioni usate per muovere dati fra CPU e memoria (punto 2): nel primo caso, varie istruzioni aritmetiche e logiche possono essere usate per spostare dati fra registri e memoria, mentre nel secondo caso tale movimento di dati viene effettuato solo da istruzioni dedicate (istruzioni *load* e *store*).

Anche le modalità utilizzate per specificare l'indirizzo di memoria a cui accedere (modalità di indirizzamento) variano fortemente a seconda dell'ISA: nei casi più semplici l'indirizzo di memoria è contenuto in un registro (a cui si può eventualmente sommare un offset fisso), mentre in casi più complessi l'indirizzo può essere calcolato in modo più articolato, sommando i contenuti di vari registri, eventualmente shiftati.

Per finire, anche la modalità di esecuzione condizionale dipende fortemente dall'ISA: alcune CPU implementano solo salti condizionali, mentre altre CPU permettono di eseguire condizionatamente qualsiasi istruzione (e quindi il salto condizionale diventa solo un caso speciale di istruzione eseguita condizionalmente). Inoltre, in alcune CPU la condizione di esecuzione è basata sul valore di un registro generico, mentre altre CPU usano il valore (0 o 1) dei bit di un apposito *registro flag*.

Anche i registri utilizzabili dalle istruzioni Assembly dipendono dal tipo di CPU, sia come numero (alcune CPU forniscono un numero ridotto di registri - per esempio 8 - mentre altre arrivano a fornire anche più di 32 registri) che come dimensione (da 8 bit per le CPU più semplici a 64 o 128 bit per le CPU più moderne e potenti).

1.2 Application Binary Interface

Come già anticipato (e come sarà meglio chiarito nel seguito di questo documento), le istruzioni Assembly operano su dati che sono memorizzati in memoria o nei registri della CPU. Ogni differente modello di CPU fornisce un differente insieme di registri (che sono generalmente indicati con un numero che va da 0 a $n-1$, anche se alcune CPU utilizzano nomi simbolici diversi per i registri). Alcune CPU hanno registri "specializzati" (per esempio, alcuni registri possono contenere solo indirizzi, altri possono contenere solo dati, etc...), mentre altre CPU forniscono registri *general purpose*, che possono essere utilizzati in modo generico a discrezione del programmatore.

Per permettere la riusabilità del codice (e per permettere a subroutine scritte in contesti diversi - e magari derivanti dalla compilazione di programmi scritti in linguaggi diversi - di interagire fra loro) è importante stabilire alcune convenzioni riguardo all'utilizzo dei registri. In particolare, l'utilizzo dei registri va specificato:

- stabilendo se il passaggio di parametri quando si invoca una subroutine avviene tramite registri o pushando i parametri sullo stack
  - se i parametri sono passati nei registri, è anche necessario specificare quali registri utilizzare
- chiarendo come viene passato il valore di ritorno da subroutine
- specificando in modo chiaro quali registri vanno *preservati* nelle chiamate a subroutine (vale a dire: chi invoca la subroutine può aspettarsi che il valore di questi registri non venga modificato)
- specificando quali registri possono essere liberamente utilizzati all'interno di una subroutine senza doversi preoccupare se il loro valore all'uscita della subroutine è diverso da quello che avevano all'ingresso

- definendo se alcuni registri vengono utilizzati per compiti specifici (non dettati dall'ISA, ma da una convenzione di programmazione Assembly).

L'ultimo punto è particolarmente importante per gestire alcune strutture dati importanti che vengono utilizzate per l'esecuzione di programmi in linguaggio macchina (per esempio, stack, record di attivazione o simili). Mentre alcune architetture hardware "dedicano" specifici registri alla memorizzazione degli indirizzi di tali strutture dati (quindi, il "ruolo" di tali registri è imposto dall'ISA), altre architetture permettono di usare un qualsiasi registro general purpose per tale scopo. In questo secondo caso è importante definire convenzioni che descrivano in modo preciso tali strutture dati, i registri dedicati alla loro gestione ed il loro utilizzo.

L'insieme di tutte queste convenzioni è chiamato *Application Binary Interface*, o *ABI* e deve essere strettamente rispettato ogni qual volta si scriva un programma Assembly che necessita di interagire con altro software (per esempio, con un Sistema Operativo o una qualche libreria). In alcuni casi esiste una corrispondenza biunivoca fra ABI ed ISA (alcuni modelli di CPU hanno un unico ABI), ma esistono anche casi in cui diversi ABI incompatibili fra loro sono usati per la stessa CPU. Per esempio, le CPU MIPS possono usare un ABI chiamato "o32" o un ABI chiamato "n32" (escludendo gli ABI a 64 bit). Anche per le CPU Intel esistono vari ABI, ed addirittura esistono due diversi ABI (chiamati "x86_64" e "x32") per la loro modalità di funzionamento a 64 bit.

In alcuni casi l'ABI influenza addirittura la sintassi del linguaggio Assembly: per esempio, le CPU MIPS hanno 32 registri i cui nomi sarebbero `$0`, `$1`, ... `$31`, ma tali registri sono spesso riferiti usando nomi simbolici che descrivono il loro utilizzo nell'ABI utilizzato. Se si usa l'ABI o32, il registro `$1` viene riferito come `$at` (assembly temporary), i registri `$2` e `$3` sono riferiti come `$v0` e `$v1` (valori di ritorno) e così via. Tutti i nomi simbolici dei registri MIPS verranno specificati nel seguito di questo documento, quando si parlerà dettagliatamente dell'Assembly MIPS.

1.3 Istruzioni Assembly

Come precedentemente accennato, un programma Assembly è costituito da una serie di istruzioni (identificate da *codici mnemonici*) che operano su dati contenuti in memoria (le cui locazioni sono identificate tramite *indirizzi*) o in registri (identificati tramite *numeri* o *nomi simbolici*).

Come suggerisce il nome, un'istruzione aritmetico/logica effettua un'operazione aritmetica o logica su due operandi, salvando poi il risultato in memoria od in un registro. Tale operazione può essere quindi descritta specificando:

- un codice mnemonico che identifica l'operazione da eseguire (per esempio, `add` per la somma, etc...)
- il valore di ogni operando, o dove reperire l'operando (specificando il nome o il numero del registro o l'indirizzo di memoria dove l'operando è memorizzato)
- dove salvare il risultato (specificando il nome del registro o l'indirizzo di memoria in cui salvare il risultato)

In particolare, ogni operando può essere *immediato* (specificando direttamente il valore dell'operando), *in registro* (specificando il nome del registro o il suo numero) o *in memoria* (specificando in vario modo l'indirizzo della locazione di memoria in cui l'operando è memorizzato).

Dipendentemente dal tipo di CPU possono esistere dei vincoli riguardo agli operandi ed alla destinazione delle istruzioni Assembly. Per esempio, le CPU basate su ISA *RISC* (Reduced Instruction Set Computer) accettano solo operandi in registro e non sono quindi in grado di operare su dati contenuti in memoria: per compiere un'operazione su un dato, questo va quindi prima caricato in un registro. Al contrario, le CPU basate su ISA *CISC* (Complex Instruction Set Computer) permettono di operare direttamente su dati residenti in memoria, anche se spesso uno dei due argomenti deve necessariamente essere in un registro (per esempio, le CPU Intel della famiglia x86 lavorano in questo modo).

Sebbene una descrizione approfondita delle differenze fra ISA CISC e RISC non sia fra gli scopi di questo documento, una CPU RISC può essere brevemente descritta come segue:

- gli operandi di ogni operazione sono contenuti in un registro o specificati direttamente
- la destinazione di ogni operazione è sempre un registro
- ci sono 2 sole istruzioni che accedono alla memoria, una (chiamata tradizionalmente `load`) per muovere dati da memoria a registro ed una (chiamata tradizionalmente `store`) per muovere dati da registri a memoria

- tutte le istruzioni assembly sono codificate su un numero di bit costante (e pari alla dimensione di una word).

Al contrario, le CPU CISC si contraddistinguono per la loro capacità di operare su dati contenuti in memoria e salvare i risultati in memoria. Un'altra caratteristica peculiare delle CPU CISC è la notevole varietà di *modi di indirizzamento* (anche se gli ISA di alcune CPU RISC hanno recentemente cominciato a fornire modalità di indirizzamento più avanzate e complesse, che tradizionalmente erano ad appannaggio dei soli ISA CISC).

La destinazione dell'operazione può essere invece un registro (unica possibilità in caso di CPU RISC) o in memoria (possibilità concessa dalle CPU CISC). Mentre molte CPU permettono di specificare indipendentemente argomenti e destinazione (quindi, ogni operazione aritmetico/logica ha 3 parametri: i due argomenti e la destinazione), in alcuni modelli di CPU (per esempio, le CPU Intel della famiglia x86) la destinazione è implicitamente uguale alla posizione del secondo operando (quindi, l'operazione ha 2 parametri: primo argomento e locazione del secondo argomento/destinazione).

Le più importanti operazioni aritmetico/logiche, che fanno parte dell'ISA di praticamente tutte le CPU esistenti, sono: somma, sottrazione, and (logico), or (logico), not (logico), shift logico a destra o a sinistra, shift aritmetico a destra (lo shift aritmetico a sinistra equivale allo shift logico a sinistra). Ad esse si aggiungono le istruzioni aritmetiche di moltiplicazione e divisione, che sono però spesso "speciali", in quanto manipolano i registri in modo diverso rispetto alle altre istruzioni aritmetiche. Per capire come mai, si consideri che mentre la somma fra due operandi a 32 bit è generalmente esprimibile su 32 bit (con overflow) o 33 bit, la moltiplicazione fra due operandi a 32 bit può necessitare di 64 bit per il risultato! Per questo l'operazione di moltiplicazione tende a memorizzare il proprio risultato in 2 registri (o 2 parole di memoria) invece che in uno. Analoghi discorsi valgono per la divisione.

Mentre le operazioni aritmetico/logiche descritte qui sopra costituiscono il minimo necessario per ottenere un ISA utilizzabile, le CPU CISC tendono a definire ulteriori istruzioni Assembly che possono apparire "ridondanti" ma semplificano notevolmente la stesura di programmi Assembly. Come esempio, l'istruzione "ruota a destra di k bit" è implementabile tramite uno shift a sinistra di $w - k$ bit (dove w è la larghezza in bit di un registro) messo in or logico con uno shift a destra di k bit. Si noti che questa implementazione, oltre a necessitare di 3 istruzioni Assembly anziché una utilizza un registro in più (per salvare temporaneamente il valore shiftato a sinistra di $w - k$ bit). Questo è uno dei motivi per cui le CPU RISC necessitano di più registri rispetto alle CPU CISC.

Gli assemblatori di CPU sia RISC che CISC forniscono inoltre delle *macro* che implementano istruzioni utili nella scrittura dei programmi Assembly ma non presenti nell'ISA. Per esempio, molte CPU RISC non forniscono istruzioni per copiare il contenuto di un registro in un registro differente. È pero' definita una macro `mov` (o `move`) che compie questa operazione sommando 0 al contenuto del primo registro e mettendo il risultato nel secondo registro (o usando trucchi simili).

Come già ribadito più volte, un ISA RISC permette di effettuare operazioni aritmetico/logiche solo su registri; poiché il numero di registri è limitato (sebbene una CPU RISC abbia un numero di registri maggiore rispetto ad una CPU CISC, il numero di registri è comunque sempre piccolo) è importante poter salvare nella memoria principale i dati su cui il programma opera (e poter prelevare dalla memoria i dati su cui operare). Nelle CPU RISC, i dati possono essere mossi da memoria a registri o viceversa usando apposite istruzioni Assembly generalmente chiamate `load` e `store`. Si noti che spesso non esistono una singola istruzione di `load` ed una singola istruzione di `store`, ma due classi di istruzioni: esistono quindi molteplici istruzioni di load e molteplici istruzioni di store, che si distinguono per la dimensione dei dati su cui operano, per il loro allineamento, etc... Questi dettagli variano da CPU a CPU. Le istruzioni di load o di store accettano in genere come parametri il registro da cui / a cui spostare i dati ed un'espressione che permette di calcolare l'indirizzo di memoria a cui / da cui spostare i dati. Per tale indirizzo di memoria si utilizza in genere la modalità di indirizzamento indiretta o registo base + spiazzamento.

Nelle CPU basate su ISA CISC non esistono in genere delle istruzioni `load` e `store` specializzate, ma si usa una generica istruzione `mov` (o `move`) che può spostare dati da un registro ad un'altro, da memoria ad un registro o da registro a memoria. Per esprimere l'indirizzo di memoria si utilizzano in genere modalità di indirizzamento più complesse rispetto alle CPU RISC.

Per finire, alcune CPU RISC forniscono istruzioni di accesso alla memoria specializzate per gestire lo stack. Tali istruzioni, spesso note come `push` e `pop`, permettono di salvare sullo stack il contenuto di un registro (inserendo il registro in testa allo stack e spostando lo stack pointer) e di recuperare il valore in testa allo stack caricandolo in un registro. Hanno quindi un parametro esplicito (il nome del registro il cui contenuto va salvato sullo stack o in cui mettere il valore recuperato dallo stack) ed un parametro implicito (l'indirizzo della testa dello stack, salvato in un registro specializzato della CPU). Altre CPU, invece, non forniscono istruzioni di `push` e `pop` specializzate, ma permettono di utilizzare

un registro general-purpose come puntatore alla testa dello stack. In questo caso, il salvataggio sullo stack del contenuto di un registro richiede un'istruzione di `store` (o `move`) verso la locazione di memoria puntata dal registro usato come stack pointer, seguita da un incremento o decremento di tale registro.

L'ultimo tipo di istruzioni Assembly (fra quelle considerate in questo documento) è costituito dalle *istruzioni di salto*: nel suo normale funzionamento, una CPU esegue le istruzioni macchina sequenzialmente, incrementando il program counter dopo il fetch di una istruzione[1] ed andando a fetchare l'istruzione immediatamente successiva. Le istruzioni di salto permettono di "rompere" questa sequenzialità dell'esecuzione settando esplicitamente il valore del program counter ad un valore arbitrario (specificato nell'istruzione stessa o calcolato in altro modo). Le principali istruzioni di salto sono il salto vero e proprio (spesso indicato con `jump`, `branch` o simile), la chiamata di subroutine ed il ritorno da subroutine. Mentre un salto semplicemente sostituisce il valore del program counter con un nuovo valore, l'istruzione di chiamata a subroutine salva da qualche parte (per esempio, sullo stack o in un registro) il valore del program counter prima di sovrascriverlo. Tale valore verrà recuperato e ri-inserito nel program counter dall'istruzione di ritorno da subroutine.

L'istruzione di salto e l'istruzione di chiamata a subroutine devono quindi specificare l'indirizzo da caricare nel program counter. Tale indirizzo può essere specificato direttamente nella codifica dell'istruzione Assembly (indirizzamento diretto), può essere contenuto in un registro (indirizzamento indiretto) o può essere calcolato in altro modo (per esempio, sommando un valore costante - specificato nella codifica dell'istruzione - al valore attuale del program counter). Si noti che se l'istruzione di chiamata a subroutine salva il valore del program counter in un registro, allora l'istruzione di ritorno da subroutine può essere semplicemente sostituita da un salto indiretto al valore di tale registro.

Per poter implementare costrutti strutturati come selezioni e cicli è necessario permettere l'esecuzione condizionale delle istruzioni di salto (per esempio: un ciclo che conta da 0 a 9 è implementabile caricando il valore 0 in un registro, incrementando il valore del registro e saltando indietro all'istruzione di incremento *se il valore del registro è minore di* 10). Ogni linguaggio Assembly fornisce quindi una o più istruzioni di *salto condizionale*; il funzionamento di tali istruzioni varia fortemente da CPU a CPU, ma in generale si possono identificare due diverse modalità di salto condizionale:

- salto eseguito in base al valore di uno o più bit in un registro speciale (*flags register* o *machine status register - msr*)
- salto eseguito in base a valori di registri general-purpose (per esempio, salto eseguito se due registri general purpose hanno valori uguali, o se hanno valori diversi).

Sebbene la seconda modalità sia tradizionalmente considerata come caratteristica di ISA RISC, esistono CPU RISC (come per esempio le CPU ARM) che utilizzano la prima modalità (basando le istruzioni di salto condizionale sui valori dei flag contenuti nel registro msr).

Se le istruzioni di salto condizionale si basano sul valore di uno o più flag, è necessario che le istruzioni aritmetico/logiche settino i valori di tali flag in base al risultato (per esempio: se il risultato di un'istruzione di somma è 0, viene settato uno *zero flag*); poiché può essere utile settare il valore dei flag senza dover salvare il risultato dell'operazione in alcun registro, molte delle CPU che usano questa tecnica forniscono un'istruzione di confronto (`cmp` o simile), che esegue la sottrazione fra due operandi e setta i valori dei flag in base al risultato di tale sottrazione, ma non salva il risultato in alcun registro o locazione di memoria. Il codice Assembly per contare da 0 a 9 sarà quindi:

1. Metti il valore immediato 0 nel registro `r1`
2. Incrementa il registro `r1` (può essere: somma il valore immediato 1 a `r1` e metti il risultato in `r1`)
3. Confronta il registro `r1` col valore immediato 10
4. Se i valori sono diversi, salta allo step 2 (se lo `zero flag` non è settato, salta a 2 - spesso indicato con `bnz <2>` o simile)
5. ...

Se invece le istruzioni di salto condizionato si basano sul valore di un registro general purpose, può essere utile che la CPU fornisca istruzioni macchina che permettano di settare il valore di un registro dipendentemente dal risultato di un confronto. In questo caso, il codice Assembly per contare da 0 a 9 potrebbe essere:

[1] questo incremento è di una quantità costante in caso di CPU RISC, mentre può variare da istruzione ad istruzione in caso di CPU CISC.

1. Metti il valore immediato 0 nel registro `r1`
2. Incrementa il registro `r1` (può essere: somma il valore immediato 1 a `r1` e metti il risultato in `r1`)
3. Confronta il registro `r1` col valore immediato 10 e setta il valore del registro `r2` se `r1` < 10
4. Se `r1` < 10, salta allo step 2 (se il valore di `r2` non è 0, salta a 2)
5. ...

1.4 Modalità di Indirizzamento

Un'istruzione Assembly può operare sul contenuto dei registri della CPU o sul contenuto di una locazione di memoria: in caso di ISA CISC, uno dei due operandi di un'operazione aritmetico/logica può essere contenuto in una locazione di memoria, o la destinazione dell'operazione può essere una locazione di memoria; in caso di ISA RISC la sorgente dell'operazione `load` o la destinazione dell'operazione `store` è una locazione di memoria. Tale locazione può essere, in generale, specificata combinando uno o più dei seguenti metodi:

- specificando direttamente l'indirizzo di memoria;
- specificando il nome (o il numero) di un registro che contiene l'indirizzo di memoria;
- specificando il nome di un registro (registro indice) che, opportunamente shiftato di un numero specificato di bit, permette di calcolare l'indirizzo di memoria.

Se l'indirizzo della locazione di memoria contenente il dato è calcolato usando un registro, alcune CPU permettono inoltre di aggiornare automaticamente il contenuto del registro (per esempio, incrementando in modo automatico un registro indice).

Combinando i tre meccanismi specificati qui sopra, si possono ottenere le seguenti modalità di indirizzamento, che sono supportate dalla maggior parte delle CPU CISC:

- Assoluto (o diretto): l'indirizzo della locazione di memoria da accedere è direttamente specificato nella codifica dell'istruzione Assembly. Questo può comportare limitazioni per le CPU RISC, dove ogni istruzione è codificata su un numero fisso di bit (quindi, il numero di bit utilizzati per codificare direttamente l'indirizzo assoluto è limitato)
- Indiretto (indirizzo in registro): l'indirizzo della locazione di memoria da accedere è contenuto in un registro (può essere un *registro indirizzi* in caso di CPU CISC). Questa è la modalità di indirizzamento più utilizzata nelle CPU RISC
- Registro (base) + Spiazzamento: l'indirizzo della locazione di memoria da accedere è dato dalla somma del contenuto di un registro (chiamato *base*) e di una costante (chiamata *spiazzamento*) che è specificata nella codifica dell'istruzione Assembly. Questa modalità di indirizzamento, che è simile all'indirizzamento indiretto con indirizzo in un registro, è supportata da praticamente tutte le CPU ed è utile per accedere ai vari campi di una struttura (dove il registro base contiene un puntatore alla struttura e lo spiazzamento indica l'offset del campo interessato rispetto all'inizio della struttura)
- Registro (Base) + Registro Scalato (indice): l'indirizzo della locazione di memoria da accedere è calcolato sommando al valore contenuto nel registro base il valore di un registro *indice* shiftato a sinistra di un numero di bit k specificato nell'istruzione Assembly (quindi, moltiplicato per 2^k). Questa modalità di indirizzamento, che tradizionalmente è supportata da CPU CISC, è utile per accedere ad array in cui il registro base è un puntatore al primo elemento dell'array, il registro indice contiene appunto l'indice dell'elemento che si vuole accedere e 2^k è la dimensione di un elemento dell'array
- Registro (Base) + Registro Scalato (indice) + Spiazzamento: come il caso precedente, ma si somma anche uno spiazzamento costante, specificato direttamente nella codifica dell'istruzione Assembly. Utile per gestire array di strutture.

A queste modalità di indirizzamento si aggiungono l'indirizzamento immediato (il valore di un argomento è direttamente specificato nell'istruzione Assembly) e l'indirizzamento a registro (il valore di un argomento è prelevato da un registro o il risultato dell'operazione è salvato in un registro).

Si noti che i linguaggi Assembly di differenti CPU possono utilizzare diverse sintassi per l'indirizzamento immediato: in particolare, gli ISA di alcune CPU hanno diverse istruzioni per l'indirizzamento immediato (per esempio, `add` fa la somma di valori contenuti in registri, mentre `addi` somma il contenuto di un registro con un valore immediato), mentre altri ISA utilizzano la stessa istruzione per diversi tipi di indirizzamento (individuando i valori immediati tramite una sintassi speciale, per esempio mettendogli un simbolo "`#`" davanti).

Capitolo 2

Assembly MIPS

2.1 Introduzione all'Architettura MIPS

L'architettura MIPS è basata strettamente su ISA RISC ed il suo design segue la filosofia RISC in modo molto rigoroso. Le CPU MIPS sono quindi dotate di un numero di registri (prevalentemente general-purpose) abbastanza alto (32 registri) ed hanno istruzioni macchina molto regolari codificate su un numero fisso di bit (1 word, cioè 32 bit). Inoltre, il numero di modalità di indirizzamento disponibili è limitato.

Per cercare di sopperire ad alcune delle limitazioni derivanti dalle caratteristiche riassunte qui sopra, un registro è dedicato a contenere la costante 0 (si tratta quindi di un registro a sola lettura, da usarsi per le operazioni aritmetico logiche che coinvolgono 0 come operando o per i confronti con 0).

Come risultato, la sintassi del linguaggio Assembly è molto regolare: per esempio, tutte le istruzioni aritmetico logiche hanno esattamente 3 argomenti (i due operandi ed il registro destinazione), che possono essere 3 registri o 2 registri ed una costante. Le istruzioni che operano su 3 registri sono chiamate *istruzioni di tipo R*, mentre le istruzioni che operano su 2 registri (avendo una costante come secondo operando) sono chiamate *istruzioni di tipo I*.

La sintassi delle istruzioni di tipo R è "`<opcode> <registro destinazione>, <registro operando>, <registro operando>`", mentre la sintassi delle istruzioni di tipo I è "`<opcode> <registro destinazione>, <registro operando>, <valore immediato>`" dove i codici mnemonici (opcode) delle operazioni di tipo I terminano con "`i`". I registri sono indicati tramite un numero che va da 0 a 31 (usando la sintassi "`$<numero>`") o tramite un nome simbolico (usando la sintassi "`$<nome>`") che indica il ruolo del registro nell'ABI utilizzato. Il primo registro ("`$0`" o "`$zero`") è un registro a sola lettura contenente il valore 0; gli altri registri sono quasi tutti general-purpose.

Poiché ogni istruzione deve essere codificabile su 32 bit, nelle istruzioni di tipo I il valore immediato non può essere di 32 bit, ma è espresso su 16 bit.

2.2 Istruzioni Aritmetico/Logiche

Come precedentemente detto, le istruzioni aritmetico/logiche dell'assembly MIPS possono essere di tipo R (lavorano su due operandi contenuti in due registri) o di tipo I (lavorano su un operando contenuto in un registro ed un operando immediato). Tutte queste istruzioni salvano il proprio risultato in un registro.

Se una stessa operazione (per esempio la somma) può avere sia tutti gli operandi a registro che un operando a registro ed uno immediato, esistono due diverse istruzioni Assembly: una di tipo R ed una di tipo I (che ha il nome dell'istruzione di tipo R equivalente, terminato con una `i`). Per esempio, `add` che lavora su 3 registri ed `addi` che ammette un operando immediato).

Esempi di istruzioni Assembly aritmetiche sono quindi

```
add  $t0, $s1, $zero
add  $8,  $17, $0
addi $s3, $s3, 4
```

o simili.

Per le operazioni aritmetiche che possono dare origine ad overflow (somma e sottrazione) o per cui il comportamento su numeri signed o unsigned cambia (divisione e moltiplicazione) esistono una versione normale ed una "unsigned". Si noti che per somma e sottrazione il termine "unsigned" può essere

Istruzione	Sintassi	Tipo	Semantica
`add`	`add $d, $s, $t`	R	somma, generando eccezione in caso di overflow
`sub`	`sub $d, $s, $t`	R	sottrazione, generando eccezione se overflow
`and`	`and $d, $s, $t`	R	and logico bit a bit
`or`	`or $d, $s, $t`	R	or logico bit a bit
`xor`	`xor $d, $s, $t`	R	or esclusivo bit a bit
`nor`	`nor $d, $s, $t`	R	or logico bit a bit negato
`addu`	`addu $d, $s, $t`	R	somma, ignorando overflow
`subu`	`subu $d, $s, $t`	R	sottrazione, ignorando overflow (no eccezioni)
`addi`	`addi $d, $s, I`	I	somma con valore immediato (eccezione se overflow)
`andi`	`andi $d, $s, I`	I	and logico bit a bit con valore immediato
`ori`	`ori $d, $s, I`	I	or logico bit a bit con valore immediato
`addiu`	`addiu $d, $s, I`	I	somma con valore immediato ignorando overflow (no eccezioni)
`sll`	`sll $d, $t, i`	R	shift logico a sinistra di i bit
`srl`	`srl $d, $t, i`	R	shift logico a destra di i bit
`sra`	`sra $d, $t, i`	R	shift aritmetico a destra di i bit
`sllv`	`sllv $d, $t, $s`	R	shift logico a sinistra di $s bit
`srlv`	`srlv $d, $t, $s`	R	shift logico a destra di $s bit
`srav`	`srav $d, $t, $s`	R	shift aritmetico a destra di $s bit
`mult`	`mult $s, $t`	R	moltiplicazione (risultato in registri speciali `HI` e `LO`)
`multu`	`multu $s, $t`	R	moltiplicazione fra unsigned (risultato in `HI` e `LO`)
`div`	`div $s, $t`	R	divisione; quoziente in `LO` e resto in `HI`
`divu`	`divu $s, $t`	R	divisione fra unsigned; quoziente in `LO` e resto in `HI`

Tabella 2.1: Istruzioni aritmetico/logiche dell'Assembly MIPS.

fuorviante, perché l'unica differenza fra la versione normale dell'istruzione e quella unsigned è che la prima genera un'eccezione in caso di overflow mentre la seconda non genera eccezioni.

Sempre riguardo alle istruzioni aritmetiche, è importante notare come le istruzioni di moltiplicazione e divisione costituiscano delle eccezioni rispetto alla regolarità delle istruzioni MIPS, in quanto hanno come destinazione implicita due registri speciali denominati `HI` e `LO` (la Sezione 2.3 mostrerà poi come recuperare i risultati da tali registri).

Le istruzioni MIPS aritmetico/logiche sono mostrate in Tabella 2.1.

E' interessante notare come per semplificare al massimo l'architettura alcune istruzioni di tipo R non hanno una corrispondente istruzione di tipo I: per esempio, esiste un'istruzione `sub` che calcola la differenza fra il contenuto di due registri, ma non la corrispondente istruzione `subi`, perché sottrarre un valore immediato equivale a sommare il complemento a 2 di tale valore costante (e tale complemento a 2 può essere calcolato in fase di compilazione).

Un'altra particolarità da notare osservando la Tabella 2.1 è che non esiste un'operazione logica di `not` (che avrebbe avuto un solo operando, rompendo così la regolarità delle istruzioni) ma esiste invece l'istruzione `nor` (`not $d, $r` può essere implementata come `nor $d, $r, $r`).

2.3 Istruzioni per il Movimento dei Dati

Essendo le CPU MIPS basate su ISA RISC, la maggior parte delle sue istruzioni lavorano solo su registri. Essendo però la dimensione del banco dei registri limitata (32 registri di 4 byte - 32 bit l'uno, con uno dei registri dedicato a contenere il valore 0), sono chiaramente necessarie istruzioni che permettano di muovere dati da memoria principale a registri e viceversa. Queste sono le istruzioni di `load` e `store` già citate in precedenza. Esistono però varie versioni di ognuna di queste due istruzioni, che permettono di muovere 4 byte (una word), 2 byte (half word) o un byte. Nel caso di caricamento di half word da memoria a registro, esistono 2 versioni dell'istruzione: una che considera i valori come signed (estendendo a 32 bit il valore in base al suo bit di segno) ed una che considera i valori come unsigned (riempiendo i 16 bit più significativi del registro con degli zeri).

L'unica modalità di indirizzamento supportata dalle varie istruzioni di caricamento/salvataggio dati da/in memoria è l'indirizzamento con registro base + spiazzamento (vedere Sezione 1.4) e le istruzioni sono di tipo I. La sintassi risultante è "<opcode> $r, S($b)", dove `$r` è il registro in/da cui muovere i

Istruzione	Sintassi	Tipo	Semantica
`lw`	`lw $d, S($b)`	I	load word: carica in `$d` la parola memorizzata a $b + S$
`lh`	`lh $d, S($b)`	I	load half word: `$b` = base, `S` = spiazzamento
`lhu`	`lhu $d, S($b)`	I	load half word unsigned: `$b` = base, `S` = spiazzamento
`lb`	`lb $d, S($b)`	I	load byte: `$b` = base, `S` = spiazzamento
`sw`	`sw $r, S($b)`	I	store word: salva in $b + S$ la parola contenuta in `$r`
`sh`	`sh $r, S($b)`	I	store half word: `$b` = base, `S` = spiazzamento
`sb`	`sb $r, S($b)`	I	store byte: `$b` = base, `S` = spiazzamento
`lui`	`lui $d, C`	I	load upper register immediato: carica C nei 16 bit più significativi di `$d`
`mfhi`	`mfhi $d`	R	muovi `HI` in `$d`
`mflo`	`mflo $d`	R	muovi `LO` in `$d`

Tabella 2.2: Istruzioni di movimento dati dell'Assembly MIPS.

Istruzione	Sintassi	Tipo	Semantica
`j`	`j C`	J	jump: salta a $C * 4$
`jal`	`jal C`	J	jump and link: salta a $C * 4$ e salva il valore di PC in `$ra`
`jr`	`jr $r`	R	jump register: salta all'indirizzo contenuto in `$r`
`beq`	`beq $s, $t, C`	I	branch equal: se s e t sono uguali, salta a `PC` $+4 + 4 * C$
`bne`	`bne $s, $t, C`	I	branch not equal: se s e t sono diversi, salta a `PC` $+4 + 4 * C$

Tabella 2.3: Istruzioni di salto dell'Assembly MIPS.

dati, `S` è una costante che indica uno spiazzamento, `$b` è il registro contenente la base ed `<opcode>` è il codice mnemonico dell'istruzione, mostrato in Tabella 2.2.

La tabella mostra anche altre 3 istruzioni di movimento dati (`lui`, `mfhi` e `mflo`) che non coinvolgono un registro ed una locazione di memoria. L'istruzione `lui` serve a caricare un valore immediato (a 16 bit) nei 16 bit più significativi di un registro. Le istruzioni `mfhi` e `mflo` servono invece per muovere dati da uno dei registri speciali `HI` e `LO` (vedere istruzioni `mult` e `div` in Sezione 2.2) in un registro general purpose.

Si noti che non esiste alcuna istruzione tipo "load register immediato" per caricare un valore immediato nei 16 bit meno significativi di un registro, perché tale operazione può essere eseguita con (per esempio) "**addi** $r, $zero, V".

Per finire, è da notare come non esista alcuna istruzione per muovere dati da un registro ad un altro, poiché tale operazione può essere eseguita con "**add** $d, $s, $zero".

2.4 Controllo del Flusso

L'ISA MIPS prevede varie istruzioni di salto, mostrate in Tabella 2.3. In particolare, ci sono 3 istruzioni di salto non condizionale (`j`, `jal` e `jr`) e due istruzioni di salto condizionale, che effettuano il salto se due registri contengono lo stesso valore (`beq`) o se contengono valori diversi (`bne`).

Delle 3 istruzioni di salto non condizionale, una (`j`) serve per i salti semplici mentre le rimanenti 2 servono per invocare subroutine e per ritornare da subroutine. Quando una subroutine viene invocata, tramite `jal` (jump and link), il valore del Program Counter incrementato di 4 (indirizzo della prossima istruzione da eseguire) viene salvato nel registro `$31` (anche noto come `$ra` - return address): questa è la parte "and link" dell'istruzione. Al termine della subroutine, basterà quindi saltare all'indirizzo contenuto in `$31` per riprendere l'esecuzione del chiamante, dall'istruzione successiva alla `jal`. Questo può essere fatto utilizzando l'istruzione `jr` (jump to register).

Istruzione	Sintassi	Tipo	Semantica
`slt`	`slt $d, $s, $t`	R	set if less than: setta `$d` a 1 se $s < t$
`sltu`	`sltu $d, $s, $t`	R	set if less than unsigned: setta `$d` a 1 se $s < t$ (confronto fra naturali)
`slti`	`slti $d, $s, C`	I	set if less than immediato: setta `$d` a 1 se $s < C$ (valore immediato)

Tabella 2.4: Istruzioni di supporto ai salti condizionali dell'Assembly MIPS.

Nome	Registro	Utilizzo Tipico	Preservato
`$zero`	`$0`	Costante 0 (read only)	
`$at`	`$1`	Riservato ad assembler	
`$v0 - $v1`	`$2 - $3`	Valori di ritorno	No
`$a0 - $a3`	`$4 - $7`	Parametri di subroutine	No
`$t0 - $t7`	`$8 - $15`	Valori Temporanei	No
`$s0 - $s7`	`$16 - $23`	Registri Salvati	Si
`$t8 - $t9`	`$24 - $25`	Valori Temporanei	No
`$k0 - $k1`	`$26 - $27`	Riservati al kernel	
`$gp`	`$28`	Gloabl Pointer	Si
`$sp`	`$29`	Stack Pointer	Si
`$fp`	`$30`	Frame Pointer	Si
`$ra`	`$31`	Return Address	

Tabella 2.5: Nomi e ruoli dei registri MIPS.

Le istruzioni di salto condizionale funzionano invece confrontando i contenuti di due registri general purpose (secondo caso di istruzioni di salto condizionale descritte i Sezione 1.3). E' quindi utile poter settare il contenuto di un registro in base a determinate condizioni, per implementare condizioni di salto più complesse. Questo può essere fatto utilizzando una delle istruzioni di confronto mostrate in Tabella 2.4.

2.5 MIPS Application Binary Interface

Come visto nella sezione 2.4, l'istruzione usata dall'Assembly MIPS per invocare una subroutine salva l'indirizzo di ritorno nel registro `$31` e non sullo stack. Rimane però da definire come passare parametri e valori di ritorno fra subroutine e chiamante, quali registri devono essere salvati dal chiamante, quali dal chiamato, etc... Tutti questi dettagli sono definiti dall'ABI.

Sebbene esistano varie ABI che possono essere utilizzate su CPU MIPS, in questo documento si fa riferimento alla più diffusa di esse, che si chiama `o32`. In base all'ABI o32, i primi 4 argomenti di una subroutine sono passati tramite registri, ed in particolare nei registri `$4`, `$5`, `$6` e `$7`. Per ricordare meglio il loro ruolo, questi registri sono spesso riferiti anche coi nomi `$a0`, `$a1`, `$a2` ed `$a3` (dove la "`a`" sta per "`a`rgument"). La subroutine ritorna invece i suoi risultati nei registri `$2` e `$3`, per questo chiamati anche `$v0` e `$v1` (dove "`v`" sta per "`v`alue", probabilmente da "return value"). Durante la sua esecuzione, la subroutine deve lasciare inalterato il contenuto dei registri da `$16` a `$23`; quindi, se la subroutine utilizza tali registri deve salvarne il contenuto iniziale per recuperarlo alla fine. Per questo, tali registri sono chiamati anche `$s0`...`$s7` dove "`s`" sta per "`s`aved" a ricordare che il loro contenuto deve essere salvato. Per finire, i registri da `$8` a `$15`, `$24` e `$25` possono essere utilizzati dalla subroutine per contenere valori temporanei (quindi, il chiamante non può assumere che il loro contenuto rimanga inalterato al ritorno dalla subroutine). Tali registri sono chiamati anche `$t0`...`$t9`, dove "`t`" sta per "`t`emporary" a ricordare che i valori salvati in tali registri sono temporanei e non è garantito che sopravvivano all'invocazione di subroutine.

Per quanto riguarda i rimanenti registri, l'utilizzo di alcuni di essi è imposto dall'ISA; per esempio, `$0` è un registro a sola lettura contenente il valore 0, mentre `$31` è il registro in cui `jal` salva l'indirizzo di ritorno, quindi l'ISA fornisce se non altro un "forte suggerimento" riguardo a come usare questo registro. L'ABI descrive invece l'utilizzo di tutti gli altri registri (per esempio, `$29` è utilizzato come stack pointer, etc...). In ogni caso, per ogni registro l'Assembler riconosce anche un nome simbolico che ne ricorda l'utilizzo; i nomi simbolici dei vari registri secondo l'ABI o32 sono riportati in Tabella 2.5.

2.6 Considerazioni Finali su MIPS

A causa della sua natura "rigidamente RISC", l'ISA MIPS non prevede alcune istruzioni che sono invece presenti nei linguaggi Assembly di altre CPU e che possono risultare particolarmente utili al programmatore. Un esempio tipico è l'istruzione per copiare il contenuto di un registro in un altro registro (generalmente nota come `mov` o `move`): tale istruzione non è presente nell'ISA MIPS perché violerebbe la regolarità delle istruzioni, non essendo ne' di tipo I (non ha operandi immediati) ne' di tipo R (ha solo un registro sorgente ed un registro destinazione; non due registri operando ed un registro destinazione).

Macro	Sintassi	Espansa a	Semantica
Move	`move $rt, $rs`	`add $rt, $rs, $zero`	rt = rs
Clear	`clear $rt`	`add $rt, $zero, $zero`	rt = 0
Not	`not $rt, $rs`	`nor $rt, $rs, $zero`	rt = rs
Load Address	`la $rd, Addr`	`lui $rd, Addr[31:16]` `ori $rd, $rd, Addr[15:0]`	rd = Address
Load Immediate	`li $rd, v`	`lui $rd, v[31:16]` `ori $rd, $rd, v[15:0]`	rd = 32 bit Immediate value `v`
Branch	`b Label`	`beq $zero, $zero, Label`	PC = Label
Branch if greater	`bgt $rs, $rt, Label`	`slt $at, $rt, $rs` `bne $at, $zero, Label`	if(rs > rt) PC=Label
Branch if less	`blt $rs, $rt, Label`	`slt $at, $rs, $rt` `bne $at, $zero, Label`	if(rs < rt) PC=Label
Branch if greater or equal	`bge $rs, $rt, Label`	`slt $at, $rs, $rt` `beq $at, $zero, Label`	if(rs >= rt) PC=Label
Branch if less or equal	`ble $rs, $rt, Label`	`slt $at, $rt, $rs` `beq $at, $zero, Label`	if(rs <= rt) PC=Label
32 bit Multiply	`mul $d, $s, $t`	`mult $s, $t; mflo $d`	d = s * t
Division	`div $d, $s, $t`	`div $s, $t; mflo $d`	d = s / t
Reminder	`rem $d, $s, $t`	`div $s, $t; mfhi $d`	d = s % t

Tabella 2.6: Macro spesso implementate da Assembler MIPS.

Tale istruzione è comunque "emulabile" in modo semplice usando l'istruzione `add` ed il registro `$0` come uno dei due operandi.

Per semplificare la vita al programmatore (e migliorare la leggibilità dei programmi Assembly MIPS), molti Assembler implementano quindi delle *macro*, che emulano il comportamento di istruzioni utili ma non esistenti usando istruzioni fornite dall'ISA MIPS. Per esempio, la macro `move` è espansa da molti Assembler come `move $rt, $rs` ≡ `add $rt, $rs, $zero`. Una lista delle macro più importanti è mostrata in Tabella 2.6.

Capitolo 3

Assembly x86

3.1 Introduzione all'Architettura Intel

Le CPU Intel della famiglia x86[1] sono sviluppate secondo un'architettura CISC, basata su modalità di funzionamento più complesse rispetto all'architettura RISC usata (per esempio) da MIPS o da ARM. Questo risulta in istruzioni macchina più complesse (e, soprattutto, in una codifica molto più complessa delle istruzioni macchina: mentre le istruzioni MIPS o ARM sono sempre tutte codificate su una parola, la codifica di un'istruzione x86 può andare da 8 a 64 bit) ed in modalità di indirizzamento più flessibili. Giusto per fare un esempio, l'Assembly x86 permette di esprimere valori immediati a 32 o 64 bit, mentre usando un'ISA RISC i valori immediati devono per forza avere una dimensione inferiore a quella della parola. Un'altra caratteristica fondamentale delle ISA CISC che differenzia notevolmente le CPU Intel da MIPS ed ARM è la possibilità di avere istruzioni aritmetico/logiche con operandi in memoria (e non solo nei registri): le istruzioni che accedono alla memoria non sono più quindi solo le istruzioni di load e store.

Le CPU Intel x86 sono però molto più complesse anche di "normali" CPU CISC, in quanto cercano di garantire compatibilità binaria con tutte le precedenti CPU della famiglia. In altre parole, una moderna CPU come un "core i7" (una CPU a 64 bit) è in grado di capire ed eseguire vecchi programmi in linguaggio macchina sviluppati per 8086 (una CPU ad 8 bit). Questo ha portato a notevoli complicazioni nella struttura dei registri della CPU; inoltre, una CPU x86 ha diversi "modi di funzionamento" (modo reale, modo protetto, modo a 64 bit, etc...) caratterizzati non solo da diverse ABI, ma addirittura da diverse ISA (il numero ed il tipo dei registri cambia da modo a modo, così come le istruzioni macchina riconosciute dalla CPU!).

Per semplicità, fra le varie ISA supportate dalle CPU Intel della famiglia x86, in questo documento verrà trattata la più moderna, chiamata `x86_64`. Questa ISA (nota anche come "`amd64`" o "modo a 64 bit") è caratterizzata da parole a 64 bit, rimuove alcune delle limitazioni che altre ISA si portavano dietro per "motivi di eredità" da generazioni precedenti (come, per esempio, l'architettura segmentata) ed ha caratteristiche che la mettono "al passo coi tempi" (come, per esempio, la possibilità di usare 16 registri "grosso modo general purpose"). Fra le varie ABI per l'ISA `x86_64`, verrà considerata quella utilizzata dal kernel Linux (i sistemi Microsoft ed Apple utilizzano ABI simili, ma con piccole variazioni nell'utilizzo dei registri; Linux supporta anche un'altra ABI chiamata "`x32`", che però non è ancora molto utilizzata).

Indipendentemente dall'ISA utilizzata, la sintassi delle istruzioni Assembly Intel è meno regolare di quella delle istruzioni MIPS (e, in minor misura, ARM). In generale, si può però dire che molte istruzioni hanno la forma `<opcode> <source> <destination>`, dove il secondo operando funge sia da destinazione che da operando implicito (non è quindi possibile specificare due diverse sorgenti ed una destinazione distinta dalle due sorgenti). Il primo operando (`<source>`) può essere un valore immediato (che nella sintassi Intel è indicato dal prefisso "`$`"), il nome di un registro (che nella sintassi Intel è indicato dal prefisso "`%`") o una espressione che indica una locazione di memoria (secondo una modalità di indirizzamento che verrà descritta nella Sezione 3.2). Il secondo operando (destinazione implicita) può chiaramente indicare solo un registro o una locazione di memoria. Una limitazione imposta dall'ISA Intel è che non è possibile avere sia sorgente che destinazione contemporaneamente in memoria (il numero di operandi in memoria può essere al più 1).

[1] Il termine "x86" che viene spesso utilizzato per identificare questa famiglia di CPU Intel deriva dal fatto che le prime CPU di questa famiglia erano Intel 8086; poi sono venute le CPU 80286, 80386 e così via. La "x" sta quindi ad identificare un numero crescente che rappresenta l'evoluzione della famiglia.

Come detto, nell'Assembly Intel i nomi dei vari registri sono identificati dal prefisso "%". Più in dettaglio, l'ISA x86_64 prevede 16 registri a 64 bit chiamati %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rsp, %rbp, %r8, %r9, %r10, %r11, %r12, %r13, %r14 ed %r15. Questi strani nomi (e l'inconsistenza fra di essi) sono dovuti ancora una volta a motivi storici:

- i primi 4 registri derivano dai registri inizialmente presenti sulle vecchie CPU 8080 (a - accumulatore - b, c e d);
- i registri %rsi ed %rdi derivano da 2 registri si e di utilizzati da vecchie CPU 8086 per la copia di array (si sta per source index e di sta per destination index);
- i registri %rsp ed %rbp sono utilizzati come stack pointer e base pointer (puntatore alla base del record di attivazione della subroutine corrente). Ancora una volta, i nomi derivano dai registri %sp e %bp presenti in precedenti CPU della famiglia x86;
- i registri %r8...%r15 sono stati introdotti ex novo nella ISA x86_64, quindi hanno nomi "più logici" non dovendo preservare compatibilità con CPU precedenti.

A questi registri si aggiungono i registri %rip (instruction pointer, nome Intel per il program counter) e %rflags (flag register, contenente i flag della CPU utilizzati per esempio dalle istruzioni di salto condizionale), che è più o meno simile alla program status word di ARM.

Il prefisso "r" che viene davanti al nome di tutti i registri ne indica l'ampiezza (64 bit); ogni registro a 64 bit contiene un registro a 32 bit (che coincide con i 32 bit meno significativi del "registro r") il cui nome ha un prefisso "e" invece che "r" (tali registri a 32 bit saranno quindi %eax, %ebx, etc). A loro volta, i "registri e" a 32 bit contengono dei sotto-registri a 16 bit, riferiti col nome senza prefisso (tali registri a 16 bit saranno quindi %ax, % bx, ...). Per finire, ognuno dei primi 4 registri a 16 bit (%ax, %bx, %cx e %dx) è composto da due sotto-registri ad 8 bit accessibili in modo indipendente, indicati sostituendo il suffisso "x" con "h" o "l" (%ax sarà quindi composto da ah ed al e così via).

3.2 Istruzioni Aritmetico/Logiche

Le istruzioni aritmetico/logiche dell'assembly Intel hanno una sintassi meno regolare rispetto a quelle di ARM e MIPS (hanno due soli argomenti, di cui uno funge anche da destinazione implicita). D'altra parte, sono molto più flessibili, in quanto possono operare sia su due registri che su un registro ed una locazione di memoria.

A differenza di quanto accade nelle architetture basate su ISA RISC, le operazioni aritmetiche e logiche possono quindi direttamente accedere alla memoria (anche se un'istruzione assembly Intel non può avere entrambi gli operandi in memoria).

Le istruzioni aritmetiche e logiche possno essere nella forma "<opcode>[<size>] <src>, <dst>" o "<opcode>[<size>] <dst>", dipendentemente dal numero di argomenti dell'operazione aritmetica o logica considerata (per esempio, l'operazione di negazione logica - not - ha un solo argomento e la corrispondente istruzione assembly avrà la seconda forma, mentre l'operazione di somma ha due argomenti e la corrispondente istruzione assembly avrà la prima forma).

Notare che <src> e <dst> rappresentano gli operandi dell'operazione e <dst> rappresenta la *destinazione* in cui viene salvato il risultato; <src> e <dst> possono essere sia registri (identificati dal prefisso "%") che indirizzi di memoria, ma come già precedentemente anticipato i due operandi non possono essere simultaneamente in memoria. L'argomento "<src>", se presente, può essere anche un valore immediato, identificato dal prefisso "$" (chiaramente, un valore immediato come destinazione non ha senso). A questo proposito è da notare come, a differenza dell'assembly MIPS, non esistono istruzioni speciali aventi operandi immediati, ma la stessa istruzione (per esempio, add) può avere come operando "<src>" sia un valore immediato, che il contenuto di un registro, che il contenuto di una locazione di memoria.

L'ISA Intel supporta modalità di indirizzamento più avanzate rispetto a MIPS (che supporta solo l'indirizzamento con registro base + spiazzamento). In particolare, la sintassi usata per identificare una locazione di memoria può essere:

- <displacement>
- [<displacement>](<base register>)
- [<displacement>](<index register>, [<scale>])

Istruzione	Sintassi	Semantica
`add`	`add <src>, <dst>`	somma
`adc`	`adc <src>, <dst>`	somma con carry
`sub`	`sub <src>, <dst>`	sottrazione
`sbb`	`sbb <src>, <dst>`	sottrazione con prestito
`div`	`div <dst>`	divisione
`idiv`	`idiv <dst>`	divisione fra interi con segno
`idiv`	`idiv <src>, <dst>`	divisione fra interi con segno
`mul`	`mul <src>`	moltiplicazione
`imul`	`imul <src>`	moltiplicazione fra interi con segno
`imul`	`imul <src>, <dst>`	moltiplicazione fra interi con segno
`inc`	`inc <dst>`	incrementa la destinazione
`dec`	`dec <dst>`	decrementa la destinazione
`sal`	`sal <cnt>, <dst>`	shift aritmetico a sinistra di `<cnt>` bit
`shl`	`shl <cnt>, <dst>`	shift logico a sinistra di `<cnt>` bit
`sar`	`sar <cnt>, <dst>`	shift aritmetico a destra di `<cnt>` bit
`shr`	`shr <cnt>, <dst>`	shift logico a destra di `<cnt>` bit
`rol`	`rol <cnt>, <dst>`	rotazione a sinistra di `<cnt>` bit
`ror`	`ror <cnt>, <dst>`	rotazione a destra di `<cnt>` bit
`rcl`	`rcl <cnt>, <dst>`	rotazione a sinistra di `<cnt>` bit con carry
`rcr`	`rcr <cnt>, <dst>`	rotazione a destra di `<cnt>` bit con carry
`neg`	`neg <dst>`	complemento a 2 (negazione)
`not`	`not <dst>`	complemento a 1 (not logico bit a bit)
`and`	`and <src>, <dst>`	and logico
`or`	`or <src>, <dst>`	or logico
`xor`	`xor <src>, <dst>`	or esclusivo
`lea`	`lea <addr>, <dst>`	load effective address: carica in `<dst>` l'indirizzo specificato da `<addr>`

Tabella 3.1: Istruzioni aritmetico/logiche dell'Assembly Intel.

- `[<displacement>](<base register>, <index register>, [<scale>])`

dove `<displacement>` e `<scale>` sono valori immediati mentre `<base register>` e `<index register>` stanno ad indicare due generici registri (nelle prime famiglie di CPU Intel esistevano dei vincoli su quali registri fossero utilizzabili come base e come indice, ma tali vincoli sono ad oggi stati rimossi).

L'indirizzo della locazione di memoria corrispondente è calcolato come $< displacement > + < base > + < index > *2^{<scale>}$; la forma in cui si omette `<displacement>` sta ad indicare che il valore del displacement fisso è 0, mentre la forma in cui si omette `<scale>` indica un valore implicito di 0 per tale parametro (indicando che il contenuto del registro `<index>` viene moltiplicato per 1). Come si può facilmente vedere, questa sintassi permette di supportare tutte le modalità di indirizzamento introdotte nella Sezione 1.4.

Per finire, il suffisso "`<size>`", può essere usato per indicare l'ampiezza in bit degli operandi: `b` significa ampiezza di 8 bit (byte), `w` significa ampiezza di 16 bit (word), `l` significa ampiezza di 32 bit (long word) e `q` significa ampiezza di 64 bit (quad word). Mentre questo suffisso è opzionale (e può quindi essere omesso) quando uno degli operandi è un registro (perché in questo caso l'ampiezza degli operandi può essere estrapolata dall'ampiezza del registro), diventa necessario quando un'istruzione assembly non ha registri come operandi (in questo caso, l'assembler non avrebbe modo di conoscere l'ampiezza degli operandi).

Esempi di istruzioni Assembly aritmetiche sono quindi

```
add $1,          %rax
add %rbx,        %rax
add 100(%rbx), %rax
not %rax
notl 200(%rbp)
```

o simili; si noti come l'istruzione `notl 200(%rbp)` non operi su un registro, ma sulla locazione di memoria indicata dal contenuto di `%rbp` +200 ed abbia quindi bisogno di un suffisso per indicare la dimensione dell'operando (32 bit in questo caso).

Istruzione	Sintassi	Semantica
`mov`	`mov <src>, <dst>`	move: copia dati da `<src>` a `<dst`
`movsx`	`movsx <src>, <dst>`	move with sign extension
`movzx`	`movzx <src>, <dst>`	move with zero extension
`push`	`push <src>`	salva un valore sullo stack
`pop`	`pop <dst>`	recupera un valore dallo stack
`cmov`	`cmov<cond> <src>, <dst>`	move condizionale

Tabella 3.2: Alcune istruzioni di movimento dati dell'assembly Intel.

Le principali istruzioni logiche ed aritmetiche dell'Assembly Intel sono mostrate in Tabella 3.1. Ancora, guardando la tabella si può notare come le istruzioni assembly dell'ISA Intel siano molto meno regolari rispetto alle ISA MIPS ed ARM: per esempio, come già anticipato alcune istruzioni hanno un unico argomento. Le istruzioni di shift e rotazione (molto più numerose e complesse delle corrispettive istruzioni MIPS) introducono inoltre ulteriori anomalie: il numero di bit `<cnt>` di cui shiftare o ruotare la destinazione `<dst>` può essere infatti indicato solo come numero intero (operando immediato) o tramite il registro `%cl` (non è possibile utilizzare altri registri per il valore `<cnt>`).

Come sempre le istruzioni di moltiplicazione hanno un comportamento diverso rispetto alle altre, in quanto hanno una destinazione implicita (`ax`, `dx:ax`, `edx:eax` o `rdx:rax`, dipendentemente dalla dimensione dell'argomento `<src>`). Per operandi a 16, 32 o 64 bit il risultato viene memorizzato in una coppia di registri, in quanto ha dimensione doppia rispetto agli operandi (per esempio: il risultato di una moltiplicazione fra interi a 32 bit è espresso su 64 bit). L'istruzione `imul` (moltiplicazione fra numeri interi con segno), a differenza di `mul` (moltiplicazione fra numeri naturali) ha anche una forma con due argomenti, che permette di specificare il registro destinazione `<dst>` (volendo essere precisi, esiste anche un'ulteriore forma a tre argomenti, che permette di specificare i due argomenti e la destinazione). In questo caso, il risultato viene espresso su un numero di bit uguale a quello degli operandi. Considerazioni analoghe si applicano anche alle istruzioni di divisione (`div` ed `idiv`), in cui a poter essere indicato in modo implicito (e a poter avere "dimensione doppia") non è però la destinazione ma uno degli argomenti (il dividendo).

Un'ultima considerazione va fatta riguardo all'istruzione `lea` (load effective address), che calcola l'indirizzo `<addr>` secondo base, spiazzamento ed indice shiftato, salvandolo in `<dst>`. Sebbene tale istruzione sia stata inizialmente progettata per effettuare il calcolo di indirizzi di memoria, viene in realtà spesso utilizzata come una semplice istruzione aritmetica che effettua contemporaneamente due somme (un valore immediato e due registri) shiftando uno degli addendi (vedere l'esempio in Sezione 5.1).

3.3 Istruzioni per il Movimento dei Dati

A differenza di quanto accade nelle ISA di CPU RISC, l'ISA delle CPU Intel non prevede due sole istruzioni (`lw` ed `sw` per le CPU MIPS) che possono accedere alla memoria, ma tutte le istruzioni aritmetico/logiche sono in grado di farlo.

Esiste comunque una classe di istruzioni che permettono di spostare dati fra registri, fra registri e memoria, o di caricare valori immediati in memoria o nei registri della CPU. Tali istruzioni si distinguono dalle istruzioni aritmetico/logiche principalmente perché non modificano il valore dei flag della CPU (usati per le istruzioni di salto condizionale, vedere Sezione 3.4). Le più importanti di tali istruzioni sono riportate in Tabella 3.2.

La principale istruzione per il movimento esplicito dei dati (`mov`) può essere usata per caricare dati da memoria a registro o per salvare il contenuto di un registro in memoria (come nelle CPU con ISA RISC), ma anche per per spostare valori da un registro all'altro o per caricare valori immediati in un registo o direttamente in memoria. In altre parole, `<src>` può essere un valore immediato, il nome di un registro o l'indirizzo di una locazione di memoria (espresso come spiegato in Sezione 3.2), mentre `<dst>` può essere il nome di un registro o l'indirizzo di una locazione di memoria. Come già spiegato per le istruzioni aritmetiche e logiche, `<src>` e `<dst>` non possono essere contemporaneamente indirizzi di memoria. Considerando l'istruzione `mov` si può notare un'ulteriore differenza rispetto alle ISA RISC: mentre un'istruzione RISC generalmente ha un solo comportamento, ben definito e chiaramente specificato (per esempio, MIPS usa istruzioni diverse a seconda che `<src>` sia un valore immediato o un registro), alcune istruzioni Intel hanno differenti comportamenti dipendentemente dalla sintassi usata (la stessa istruzione `mov` può essere usata per caricare un valore immediato in un registro o per salvare il contenuto di un

Istruzione	Sintassi	Semantica
`jmp`	`jmp <address>`	jump immediato: salta all'indirizzo `<address>` (valore immediato)
`jmp`	`jmp *<reg>`	jump to register: salta all'indirizzo contenuto in `<reg>`
`j<cond>`	`j<cond> <address>`	salto condizionale: salta ad `<address>` se la condizione `<cond>` si verifica
`j<cond>`	`j<cond> *<reg>`	salto condizionale: salta al contenuto di `<reg>` se la condizione `<cond>` si verifica
`call`	`call <address>`	invocazione di subroutine ad indirizzo `<address>`
`call`	`call *<reg>`	invocazione di subroutine all'indirizzo contenuto in `<reg>`
`ret`	`ret`	ritorno da subroutine

Tabella 3.3: Istruzioni di salto dell'Assembly Intel.

registro in memoria). Esistono poi due istruzioni simili a `mov` (`movsx` e `movsz`) che permettono di muovere un valore verso un registro (quindi, `<dst>` è sempre il nome di un registro) estendendo la dimensione del dato copiato (`movsx` estende in base al segno, mentre `movsz` estende con degli zeri).

Altre due istruzioni permettono di muovere dati fra registri della CPU e stack, salvando sullo stack valori contenuti in un registro (`push`) o recuperando dallo stack i valori salvati (`pop`)[2]. Mentre in altre CPU (soprattutto basate su ISA RISC) la manipolazione dello stack avviene utilizzando le normali istruzioni di load e store e modificando esplicitamente il valore del registro usato come stack pointer (che è un registro general purpose come tutti gli altri), le CPU Intel forniscono quindi due istruzioni che provvedono a spostare i dati e contemporaneamente modificare il registro `%esp`, che non è quindi un registro general purpose.

Tipicamente, le istruzioni `push` vengono utilizzate nel prologo delle subroutine per salvare il contenuto dei registri preservati (vedere Sezione 3.5) e le istruzioni `pop` vengono utilizzate nell'epilogo per recuperare il contenuto dei registri salvati. Si noti che a differenza delle CPU MIPS (e, come si vedrà in seguito, ARM) non è necessario salvare sullo stack il contenuto del link register perché l'istruzione `call` già salva il contenuto del registro `%rip` sullo stack.

3.4 Controllo del Flusso

Le istruzioni di salto fornite dall'ISA Intel sono mostrate in Tabella 3.3. Come si può vedere, esiste un'unica istruzione di salto non condizionale (`jmp`) che può essere utilizzata sia per saltare ad una locazione indicata da un valore immediato (in genere, nei programmi assembly non si esprime direttamente tale valore immediato ma si usano delle `label`) che per saltare alla locazione di memoria contenuta in un registro (non esiste quindi un'istruzione "jump to register" separata).

Esistono poi varie istruzioni di salto condizionale, indicate con `j<cond>`, che effettuano il salto se è verificata la condizione `<cond>`, espressa in base al valore di alcuni flag del *flags register*. Questa è un'altra importante differenza rispetto all'ISA MIPS, in cui le condizioni di salto sono espresse in base al valore di registri general purpose. Nelle CPU Intel esiste invece un registro special purpose i cui bit rapprresentano dei flag che vengono settati durante l'esecuzione di istruzioni aritmetiche e logiche, per segnalare condizioni di overflow (overflow flag), il segno del risultato (sign flag), il fatto che il risultato è zero (zero flag), etc... Mentre nelle CPU MIPS esistono istruzioni che permettono di settare il valore di un registro general purpose in base al confronto dei contenuti di due registri (`slt` e simili), nelle CPU Intel le istruzioni aritmetiche e logiche settano i flag del flag register in base al risultato dell'operazione. Esiste poi un'istruzione chiamata `cmp` (che possiamo considerare come l'istruzione corrispondente a `slt` e simili) che calcola la differenza fra due valori, senza salvare il risultato da nessuna parte ma settando opportunamente i flag.

Le possibilità per la condizione `<cond>` sono:

- `e` (equal): salta se lo *zero flag* (`zf`) è settato. Questo avviene se una precedente operazione di confronto è stata effettuata su valori uguali;
- `ne` (not equal): salta se lo zero flag (`zf`) non è settato (una precedente operazione di confronto è stata effettuata fra due valori diversi)
- `g` (greater): salta se lo zero flag (`zf`) non è settato ed il sign flag (`sf`) ha lo stesso valore dell'overflow flag (`of`);

[2]In realtà esistono altre istruzioni della famiglia di `push` e `pop`, che permettono di muovere il contenuto di più registri con una sola istruzione, ma queste istruzioni vengono tralasciate per semplicità.

- `ge` (greater or equal): salta se il sign flag `sf` ha lo stesso valore dell'overflow flag `of`
- `a` (above): salta se il *carry flag* (`cf`) non è settato e lo zero flag `zf` non è settato. Questa condizione è simile alla condizione `g` (vera dopo un confronto fra due numeri di cui il primo è maggiore del secondo), ma considera confronti fra numeri naturali (valori senza segno)
- `ae` (above or equal): salta se il carry flag `cf` non è settato
- `l` (less): salta se il sign flag `sf` ha un valore diverso da quello dell'overflow flag `of`
- `le` (less or equal): salta se il sign flag `sf` ha un valore diverso da quello dell'overflow flag `of` o se lo zero flag `zf` è settato
- `b` (below): salta se il carry flag `cf` è settato (condizione simile a `l`, ma considera numeri naturali)
- `be` (below or equal): salta se il carry flag `cf` è settato o se lo zero flag `zf` è settato (come `b` ma salta anche in caso di confronto fra due numeri uguali)
- `o` (overflow): salta se l'*overflow flag* (`of`) è settato. Questa condizione identifica le operazioni che hanno generato overflow
- `no` (not overflow): salta se l'overflow flag `of` non è settato. In pratica, è l'opposto della condizione `o`
- `z` (zero): salta se lo zero flag `zf` è settato. Questa condizione è un sinonimo per la condizone `e`
- `nz` (not zero): salta se lo zero flag `zf` non è settato; equivalente alla condizione `ne`
- `s` (sign): salta se il sign flag `sf` è settato
- `ns` (not sign): salta se il sign flag `sf` non è settato

Ancora, da quanto appena descritto si vede come l'ISA Intel sia più complessa (e potente) dell'ISA MIPS, consistentemente con la filosofia CISC.

Per finire, si noti che le istruzioni di salto condizionale si utilizzano generalmente in combinazione con l'istruzione `cmp`, ma possono essere utilizzate anche a seguire altre istruzioni logiche o matematiche (per esempio, per implementare un loop con contatore l'istruzione di salto condizionato può seguire una istruzione `dec`).

3.5 Intel 64bit Application Binary Interface

Come spiegato nella Sezione 3.4, a differenza delle CPU MIPS ed ARM le CPU della famiglia Intel implementano le istruzioni di invocazione di subroutine salvando l'indirizzo di ritorno sullo stack e non in un registro. Tutti i rimanenti dettagli riguardo all'invocazione di subroutine sono specificati dall'ABI.

Secondo l'ABI utilizzata da Linux per l'ISA Intel a 64 bit (x86_64), i primi 6 argomenti sono passati nei registri `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` e `%r9` mentre eventuali argomenti oltre al sesto sono passati sullo stack (si noti che secondo l'ABI utilizzata per l'ISA a 32 bit tutti i parametri sono passati sullo stack). I valori di ritorno sono invece passati nei registri `%rax` e `%rdx`. Durante la sua esecuzione, la subroutine deve lasciare inalterato il contenuto dei registri `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14` e `%r15`, mentre può modificare senza necessità di salvarli i registri `%rax`, `%r10` e `%r11` (oltre ai registri utilizzati per i primi 6 argomenti).

Capitolo 4

Assembly ARM

4.1 Introduzione all'Architettura ARM

L'architettura ARM è basata su ISA RISC come l'architettura MIPS, ma il suo design segue un approccio più pragmatico per sopperire a quelle che sono risultate essere le limitazioni delle architetture RISC "tradizionali" (eccessiva dimensione degli eseguibili, dovuta al gran numero di istruzioni necessario per codificare i programmi, perdita di efficienza dei meccanismi di pipeline, dovuta ad operazioni di salto, etc...).

Come risultato, le CPU ARM sono dotate di un numero di registri (prevalentemente general-purpose) inferiore rispetto alle CPU MIPS (16 registri invece di 32), in modo da "salvare" alcuni bit nella codifica dell'istruzione per permettere di supportare istruzioni più complesse e potenti. La sintassi delle istruzioni macchina è anche meno regolare rispetto a MIPS (ci sono per esempio istruzioni aritmetico/logiche che operano su 3 registri, altre che operano su 2 registri ed altre ancora che operano su solo 1 registro). Questa maggior complessità (e minor regolarità) delle istruzioni macchina ha però il vantaggio che non è necessario avere un registro read-only per contenere il valore 0 (come si vedrà, le CPU ARM permettono di esprimere il valore immediato 0 su 32 bit). Per finire, le modalità di indirizzamento disponibili sono più articolate di quelle utilizzabili su CPU MIPS.

Come per le CPU MIPS, le istruzioni macchina sono codificate su un numero fisso di bit (1 word, cioè 32 bit).

Una delle caratteristiche peculiari dell'Assembly ARM è la possibilità di eseguire condizionalmente *tutte* le istruzioni (e non solo le istruzioni di salto, come in tutte le altre architetture). Dopo l'opcode di ogni istruzione Assembly può quindi essere aggiunta una *condizione* composta da due caratteri che indica quando l'istruzione deve essere eseguita. Tale condizione è espressa in base al valore di alcuni flag in uno speciale registro di stato (PSR - Program Status Register). In particolare, si considerano i 4 flag contenuti in un registro di stato chiamato `apsr` (Application Program Status Register), che è un sottoinsieme di un registro di stato più ampio chiamato `cpsr` (Current Program Status Register). I 4 flag rilevanti, contenuti nel registro di stato, sono: il flag `N` (negative - viene settato quando il risultato di un'operazione matematica è negativo), il flag `Z` (zero - viene settato quando il risultato di un'operazione è 0), il flag `C` (carry flag - settato quando una somma ha riporto uguale ad 1 nel bit più significativo, quando una sottrazione ha prestito 1 nel bit più significativo o quando uno shift a sinistra sposta un 1 fuori dalla parola) ed il flag `V` (oVerflow - settato quando un'operazione aritmetica risulta in un overflow).

Le possibili condizioni utilizzabili per l'esecuzione condizionale sono:

- `eq` (equal): esegui se la precedente operazione era fra numeri uguali (esegui se il flag `Z` vale 1)
- `ne` (not equal): esegui se la precedente operazione era fra numeri diversi (esegui se il flag `Z` vale 0)
- `hs` (higher or same) o `cs` (carry set): esegui se il flag `C` vale 1
- `lo` (lower) o `cc` (carry clear): esegui se il flag `C` vale 0
- `mi` (minus): esegui quando il risultato dell'ultima operazione è negativo (esegui se il flag `N` vale 1)
- `pl` (plus): esegui quando il risultato dell'ultima operazione è positivo (esegui se il flag `N` vale 0)
- `vs` (overflow): esegui quando l'ultima operazione è risultata in un overflow (esegui se il flag `V` vale 0)

- `vc` (overflow clear): opposto di `vs` (esegui se il flag `V` vale 0)
- `hi` (higher): esegui se in un'operazione di confronto il primo operando è maggiore del secondo, assumendo unsigned (esegui se `C` vale 1 e `Z` vale 0)
- `ls` (lower or same): esegui se in un confronto il primo operando è minore o uguale al secondo, assumendo unsigned (esegui se `C` vale 0 o `Z` vale 1)
- `ge` (greater or equal): esegui se `N` vale 1 e `V` vale 1, o se `N` vale 0 e `V` vale 0
- `lt` (less than): esegui se `N` vale 1 e `V` vale 0, o se `N` vale 0 e `V` vale 1
- `gt` (greater than): come `ge`, ma con `Z` che vale 0
- `le` (less or equal): come `lt`, ma esegue anche se `Z` vale 1

Tecnicamente, esistono anche la condizione `al` (always), che comporta l'esecuzione incondizionata dell'istruzione, e la condizione `nv` (never), che comporta la non esecuzione dell'istruzione. La condizione `al` è considerata come implicita nel caso in cui non venga specificata alcuna condizione, quindi un'istruzione Assembly senza suffissi particolari viene sempre eseguita (come in tutti gli altri linguaggi Assembly).

4.2 Istruzioni Aritmetico/Logiche

Le istruzioni aritmetico/logiche dell'Assembly ARM hanno una sintassi più complessa rispetto a quelle dell'Assembly MIPS, sia a causa della possibilità di esecuzione condizionale che a causa di feature quali la possibilità di shiftare il secondo operando. Come in tutte le CPU che adottano ISA RISC le istruzioni aritmetico/logiche operano solo su registri e non possono accedere direttamente alla memoria.

La sintassi delle istruzioni aritmetico/logiche è:

"`<opcode>`*[<cond>][s]* `rd, rl, <r>`"

dove `rd` è il *registro destinazione* in cui viene memorizzato il risultato, `rl` (left register) è il registro che contiene il primo operando e `<r>` (right) indica il secondo operando. Il suffisso `s` può essere opzionalmente specificato per far si che l'istruzione aritmetico/logica aggiorni i flag del registro di stato in base al suo risultato (se non si specifica il suffisso `s`, i flag non vengono aggiornati e l'esecuzione condizionale delle prossime istruzioni non sarà influenzata dal risultato di questa istruzione).

Si noti che il secondo operando è specificato come `<r>` e non come `rr` perché può essere un valore immediato e non solo un registro. Il valore del secondo operando può essere rotato o shiftato prima di usarlo. Per questo, si può indicare il valore o nome di registro seguito da una operazione di shift, specificando i bit di cui shiftare come valore immediato o come registro. Le "operazioni di shift" (`<shift op>`) sono:

- lsl (asl)
- lsr
- asr
- ror (rotate right immediate)
- rrx (rotate right one bit with extend)

Il parametro `<r>` può essere un valore immediato (indicato tramite il suffisso "`#`"), il nome di un registro (`rr` - right register), il nome di un registro il cui contenuto va shiftato di un numero di bit fisso `#i` (`rr, <shift op> #i`) o il nome di un registro il cui contenuto va shiftato di un numero di bit variabile, contenuto in un altro registro `rs` (`rr, <shift op> rs`).

Esempi di istruzioni aritmetico logiche sono quindi:

```
add  r0, r1, #1
adds r0, r1, r2
add  r0, r1, r2, lsl #2
add  r0, r1, r2, lsl r3
```

Istruzione	Sintassi	Semantica
`add`	`add rd, rl, <r>`	somma
`adc`	`adc rd, rl, <r>`	somma con riporto
`sub`	`sub rd, rl, <r>`	sottrazione
`sbc`	`sbc rd, rl, <r>`	sottrazione con prestito
`rsb`	`rsb rd, rl, <r>`	sottrazione inversa (sottrai rl da ¡r¿)
`rsc`	`rsc rd, rl, <r>`	sottrazione inversa con prestito
`and`	`and rd, rl, <r>`	and logico
`orr`	`orr rd, rl, <r>`	or logico
`eor`	`eor rd, rl, <r>`	or esclusivo (xor)
`bic`	`bic rd, rl, <r>`	and logico fra rl e not ¡r¿ (azzera bit selezionati)
`mul`	`mul rd, rl, rr`	moltiplicazione
`mla`	`mla rd, rl, rr, ra`	moltiplicazione con somma

Tabella 4.1: Istruzioni aritmetico/logiche dell'Assembly ARM.

Istruzione	Sintassi	Semantica
`mov`	`mov rd, <r>`	muovi `r` in `rd`
`mvn`	`mvn rd, <r>`	mov negato: muovi il complemento a 1 di `r` in `rd`
`ldr`	`ldr rd, <addr>`	carica dati da memoria a registro
`str`	`str rl, <addr>`	salva dati da registro a memoria

Tabella 4.2: Istruzioni di movimento dati dell'Assembly ARM.

Le istruzioni aritmetico/logiche dell'ISA ARM sono mostrate in Tabella 4.1. Si noti come non esista un'istruzione di "not logico" (il perché diventerà chiaro in Sezione 4.3, parlando dell'istruzione di "mov negato"). La soluzione implementata per fornire la negazione logica è però diversa da quella usata dall'ISA MIPS (vedere Sezione 2.2).

Un'altra cosa interessante da notare è la presenza delle istruzioni `rsb` ed `rsc`, che sono equivalenti a `sub` e `sbc`, ma invertono il ruolo dei due operandi. Queste operazioni permettono di sfruttare meglio la potenza espressiva fornita dall'operando `r`, che può essere un valore immediato, un registro o un registro shiftato/rotato (al contrario, l'operando `rl` è sempre un registro). Si pensi per esempio a come implementare in Assembly l'espressione `1 - r`, dove `r` è un valore contenuto in un registro. Senza `rsb`, sono necessarie almeno 2 istruzioni Assembly, mentre usando `rsb` l'espressione può essere codificata usando un'unica istruzione.

Per finire, ancora una volta le istruzioni di moltiplicazione (`mul` e `mla`) sono speciali: in particolare, il secondo operando (`rr`) può essere solo un registro (esistono poi altre limitazioni: per esempio, `rd` ed `rr` devono essere registri diversi e `rd` deve essere diverso da `r15`).

4.3 Istruzioni per il Movimento dei Dati

Come le CPU MIPS, anche le CPU ARM sono basate su ISA RISC, quindi la maggior parte delle istruzioni lavorano solo su registri (o su valori immediati). Sono quindi necessarie istruzioni per muovere dati dalla memoria principale a registri (e viceversa) o per muovere valori immediati in un registro. Questa seconda possibilità (caricamento di valori immediati in un registro) costituisce un'altra importante differenza rispetto ad ISA RISC "pure" (come MIPS), in cui il movimento di valori immediati in un registro non è implementato tramite una vera istruzione Assembly, ma tramite una macro che utilizza l'istruzione di somma immediata.

Le istruzioni di movimento dati dell'ISA ARM sono mostrate in Tabella 4.2. Si noti che le istruzioni `mov` e `mvn` permettono di copiare un valore immediato, un registro, o un registro shiftato/rotato (l'operando `<r>` ha lo stesso significato e lo stesso potere espressivo già visto per le operazioni aritmetico/logiche) in un registro destinazione `rd` (sostanzialmente, la sintassi di `mov` e `mvn` è simile a quella delle operazioni aritmetico/logiche, ma manca il primo operando `rl`). L'istruzione `mvn` copia in `rd` il *complemento a* 1 di `<r>`, permettendo di implementare l'operazione logica not.

Le istruzioni di trasferimento dati fra registri e memoria (`ldr` - load register - e `str` - store register) permettono di specificare l'indirizzo di memoria su cui operare utilizzando delle modalità di indirizzamento più complesse rispetto a quelle fornite dall'architettura MIPS. In particolare, l'ISA ARM supporta

Istruzione	Sintassi	Semantica
`ldm<mode>`	`ldm<mode> r`*[!]*`, <register list>`	carica una lista di registri da memoria
`stm<mode>`	`stm<mode> r`*[!]*`, <register list>`	salva una lista di registri in memoria

Tabella 4.3: Caricamento e salvataggio di più registri nell'Assembly ARM.

le modalità di indirizzamento chiamate "indirizzamento indiretto" (l'indirizzo di memoria da accedere è contenuto in un registro), "indirizzamento con base e spiazzamento" ed "indirizzamento con base ed indice" (registro eventualmente scalato) in Sezione 1.4. In più, le istruzioni di accesso alla memoria possono opzionalmente modificare il contenuto del registro base; tale modifica può avvenire prima dell'accesso alla memoria (pre-indexed addressing) o dopo l'accesso (post-indexed addressing). Queste istruzioni possono inoltre avere un suffisso "`<size>`" che indica la dimensione dei dati da spostare: aggiungendo "`b`" come suffisso a `ldr` o `str` si specifica che l'operazione deve spostare un singolo byte (il byte meno significativo del registro specificato), mentre aggiungendo "`h`" si specifica che l'istruzione deve spostare mezza parola (2 byte).

La sintassi per l'indirizzamento pre-indexed è

`<op>`*[<cond>][<size>]* `r, [rb, <offset>]`*[!]*

dove `<offset>` è un valore immediato ("`#i`"), un registro ("`+|- ro`") o un registro shiftato/rotato ("`+|-ro,<shift>`"). Il "`!`" opzionalmente aggiunto dopo la specifica dell'indirizzo indica che il valore del registro specificato come base viene aggiornato a base + offset. Quindi, per esempio, "**ldr** r0, [r1,#1]!" carica in `r0` il contenuto della parola di memoria all'indirizzo `r1` +1 ed incrementa di 1 il valore di `r1`.

La sintassi per l'indirizzamento post-indexed è invece

`<op>`*[<cond>][<size>]* `r, [rb], <offset>`

Si noti che in questo caso il valore del registro base viene sempre aggiornato (ma che la locazione di memoria acceduta è quella indicata dal valore del registro base prima della modifica) e che l'offset non è opzionale ma obbligatorio (post-indexing senza offset non ha senso, perché lascerebbe invariato il valore del registro base). Per finire, non è necessario specificare "`!`" per indicare che il valore del registro base deve essere aggiornato (l'aggiornamento del registro base è implicito nella semantica dell'indirizzamento post-indexed).

Riassumendo, la sintassi dell'indirizzamento pre-indexed è

`<op>`*[<cond>][<size>]* `r,[rb, #i]`*[!]*

per indirizzamento con registro base o base e spiazzamento immediato o

`<op>`*[<cond>][<size>]* `r,[rb,{+|−}ro`*[,<shift>]*`]`*[!]*

per indirizzamento con registro base e registro indice (eventualmente shiftato).

Come detto, l'unico indirizzamento supportato da post-indexed è l'indirizzamento con registro base (la locazione di memoria da accedere è specificata sempre e solo dal contenuto di un registro); esistono però varie sintassi per specificare come aggiornare tale registro dopo l'accesso. La sintassi

`<op>`*[<cond>][<size>]* `r,[rb],#i`

specifica che il registro base va aggiornato sommandoci uno spiazzamento immediato, mentre la sintassi

`<op>`*[<cond>][<size>]* `r,[rb],{+|-}ro`*[,<shift>]*

specifica che il registro base va aggiornato sommandoci il valore di un registro indice eventualmente shiftato o rotato.

Oltre alle istruzioni per il caricamento da memoria e salvataggio in memoria di singoli registri, l'ISA ARM fornisce istruzioni per salvare e caricare contemporaneamente più registri, mostrate in Tabella 4.3. Nella tabella, `r` rappresenta un registro in cui è contenuto l'indirizzo a partire dal quale leggere o scrivere i dati, mentre il suffisso `<mode>` può essere `ia` (increment after), `ib` (increment before), `da` (decrement after) o `db` (decrement before).

I registri indicati in `<register list>` vengono salvati in memoria a partire dall'indirizzo contenuto in `r`, secondo una strategia che dipende dal suffisso `<mode>`:

- in caso di "increment after" (`ia`), i registri sono salvati nelle locazioni indicate da `r`, `r`+4, `r`+8, etc...
- in caso di "increment before" (`ib`), i registri sono salvati nelle locazioni indicate da `r`+4, `r`+8, `r`+12, etc...
- in caso di "decrement after" (`da`), i registri sono salvati nelle locazioni indicate da `r`, `r`−4, `r`−8, etc...

- in caso di "decrement before" (`db`), i registri sono salvati nelle locazioni indicate da `r`-4, `r`-8, `r`-12, etc...

Se il simbolo `!` è specificato dopo il registro `r`, il valore di `r` è aggiornato (analogamente a quanto fatto in caso di indirizzamento pre-indexed con `!` o di indirizzamento post-indexed) come `r` = `r` $+ 4 * n$ (dove n è il numero di registri contenuti in `<register list>`) se `<mode>` è `ia` o `ib` o `r` = `r` - $4 * n$ se `<mode>` è `da` o `db`.

Sebbene le istruzioni di load e store multipli `ldm` e `stm` possano essere utilizzate per molti scopi, sono spesso usate per salvare registri sullo stack (`stm`) nel prologo di una subroutine o recuperarli dallo stack (`ldm`) nell'epilogo. In questo caso, il registro `r` rappresenta lo stack pointer. Le diverse varianti delle istruzioni `ldm` ed `stm` (`ia`, `ib`, `da` e `db`) permettono di usare diverse convenzioni riguardo alla gestione dello stack (`i` e `d` stabiliscono se lo stack cresce verso l'alto o verso il basso, mentre `a` e `b` stabiliscono se il registro `r` punta alla prima locazione libera sullo stack o all'ultima locazione occupata). A causa di questo utilizzo delle istruzioni `ldm` ed `stm` per la gestione dello stack, le loro varie forme sono riconosciute anche tramite i seguenti sinonimi:

- `stmia` è noto anche come `stmea`, dove `ea` sta per **e**mpty **a**scending, per indicare che lo stack pointer (il registro `r`) indica la prima locazione libera sullo stack (`r` punta ad una locazione di memoria libera - vuota == empty) e lo stack cresce incrementando lo stack pointer `r`. Quindi, il valore di `r` deve essere modificato *dopo* aver salvato dati sullo stack e deve essere modificato incrementandolo;
- `stmib` è noto anche come `stmfa`, dove `fa` sta per **f**ull **a**scending: lo stack pointer `r` indica l'ultima posizione *occupata* (piena - full) dello stack, che cresce incrementando lo stack pointer
- `stmda` è noto anche come `stmed`, dove `ed` sta per **e**mpty **d**escending
- `stmdb` è noto anche come `stmfd`, dove `fd` sta per **f**ull **d**escending
- `ldmia` è noto anche come `ldmfd`
- `ldmib` è noto anche come `ldmed`
- `ldmda` è noto anche come `ldmfa`
- `ldmdb` è noto anche come `ldmea`

Il maggior vantaggio che si ha nell'usare questi sinonimi è che prologo ed epilogo di un subroutine possono usare versioni consistenti di `stm` ed `ldm`. Per esempio, si consideri una subroutine che deve salvare i registri `lr` ed `r4` su uno stack che cresce verso il basso (decrementando lo stack pointer) in cui `sp` indica l'ultima posizione piena sullo stack. Il prologo della subroutine può salvare i due registri tramite l'istruzione "**stmdb** sp!, {r4, lr}" che decrementa `sp` di 4 prima di salvare il primo registro (poiché `sp` punta ad una locazione occupata dello stack). L'epilogo potà poi recuperare i valori originali dei registri tramite "**ldmia** sp!, {r4, lr}" che recupera il valore del primo regisrto dalla locazione puntata da `sp` e solo dopo incrementa tale registro. In questo caso, prologo ed epilogo devono usare due tipi diversi per `stm` ed `ldm` (`db` nell'epilogo ed `ia` nel prologo). Grazie ai sinonimi appena descritti, il prologo può invece usare "**stmfd** sp!, {r4, lr}" mentre l'epilogo della subroutine può usare "**ldmfd** sp!, {r4, pc". In questo modo, prologo ed epilogo usano entrambi la "versione fd" delle istruzioni di load e store multiplo, portando a codice più comprensibile.

4.4 Controllo del Flusso

Come noto, indipendentemente dal tipo di architettura o di ISA utilizzato ogni CPU utilizza un registro detto Program Counter (PC) o Instruction Pointer (IP) per memorizzare l'indirizzo della prossima istruzione macchina da prelevare dalla memoria (fase di fetch). Consistentemente con la filosofia RISC, l'ISA ARM considera il PC come un *registro general purpose*; questo significa che è possibile operare su tale registro con le "normali" istruzioni aritmetico/logiche o di movimento dati. Come conseguenza, l'ISA ARM permette di implementare operazioni di salto manipolando il PC tramite (per esempio) istruzioni come `add`, `mov` o simili. Le cose sono però complicate dal fatto che nell'architettura ARM il PC è quindi un registro a 24 bit ed occupa i bit 2..26 del registro `r15` (condividendo `r15` con il registro flag). Inoltre, esistono dei vincoli sugli operandi immediati (che nell'architettura ARM sono codificati su 12 bit) che complicano l'implementazione di salti generici tramite istruzioni aritmetico/logiche o di movimento dati. Per sopperire a tali limitazioni, l'ISA ARM fornisce alcune istruzioni di salto "specializzate".

Istruzione	Sintassi	Semantica
`b`	`b`[<*cond*>] `C`	branch: salta all'indirizzo indicato da C
`bl`	`bl`[<*cond*>] `C`	branch with link: salta all'indirizzo indicato da C e salva il PC in `r14`

Tabella 4.4: Istruzioni di salto dell'Assembly ARM.

Grazie all'enorme flessibilità fornita dall'esecuzione condizionale ed alla potenza delle istruzioni Assembly `mov`, le istruzioni di salto dell'Assembly ARM (mostrate in Tabella 4.4) sono solamente 2 e sono estremamente semplici. In particolare, le istruzioni di salto condizionale diventano un caso speciale delle istruzioni di salto `b` e `bl` con un'oppourtuna condizione come suffisso.

Come per l'ISA MIPS, le subroutine vengono invocate salvando l'indirizzo della prossima istruzione (PC + 4) in un *link register* (`r14` nel caso di ARM) tramite l'istruzione di branch and link `bl`. L'istruzione *jump to register* fornita dall'ISA MIPS per il ritorno da subroutine (`jr $31` in caso di MIPS) non è invece necessaria per ARM, in quanto l'istruzione `mov` permette di spostare il contenuto di `r14` nel `PC` senza bisogno di ulteriori istruzioni. In particolare, il ritorno da subroutine è realizzabile con `mov r15, r14` (o `movs r15, r14` nel caso si vogliano ripristinare i flag) in quanto il PC è memorizzto nei bit 2..25 del registro `r15`. Si noti che comunque alcuni modelli di CPU ARM forniscono un'istruzione `bx r` che è più o meno equivalente ad `mov r15, r` (salta all'indirizzo contenuto nel registro `r`).

Le due istruzioni di salto dell'Assembly ARM codificano un offset rispetto al valore attuale del PC, espresso su 24 bit. Sarà compito dell'Assembler convertire un valore assoluto "`C`" usato in istruzioni come "`b C`" o simili in un offset appropriato; generalmente, nei programmi Assembly il valore "`C`" è semplicemente un'etichetta definita nel programma.

4.5 ARM Application Binary Interface

Analogamente a quanto fatto da CPU MIPS, anche le CPU ARM forniscono un'istruzione di invocazione di subroutine (`bl`) che salva l'indirizzo di ritorno in un registro della CPU, il registo `r14` (vedere la Sezione 4.4).

Come al solito, l'ABI definisce tutti i rimanenti aspetti delle invocazioni di subroutine: i meccanismi di passaggio dei parametri e valori di ritorno fra subroutine e chiamante, i registri che devono essere salvati dal chiamante, quelli il cui salvataggio è responsabilità del chiamato, etc...

Secondo l'ABI più utilizzata su ISA ARM (chiamata "eabi"), i primi 4 argomenti sono passati nei registri `r0`, `r1`, `r2` e `r3` ed eventuali ulteriori argomenti sono passati sullo stack. I valori di ritorno sono invece passati nei registri `r0` ed `r1`. Durante la sua esecuzione, la subroutine deve lasciare inalterato il contenuto dei registri da `r4` ad `r11` (alcune varianti dell'ABI considerano però il registro `r9` come eccezione a questa regola), mentre il chiamante non può fare assunzioni riguardo al registro `r12` (oltre ai registri usati per passare i primi 4 argomenti). Come precedentemente notato, il ruolo del registro `r14` (che funge da link register per l'istruzione `bl`) è fissato dall'ISA.

Capitolo 5

Esempi

Questo capitolo presenta alcuni esempi di programmazione Assembly per le tre ISA considerate (MIPS, ARM ed x86_64), per permettere di capire meglio come scrivere programmi Assembly e la differenza fra le tre ISA.

La maggior parte degli esempi di codice assembly è stata ottenuta usando il compilatore `gcc` (configurato per il target opportuno) con le opzioni `-O1 -fno-dwarf2-cfi-asm -S`: l'opzione `-O1` è necessaria per evitare che `gcc` utilizzi lo stack per salvare valori intermedi, `-fno-dwarf2-cfi-asm` è necessaria per evitare che `gcc` inserisca nel codice Assembly informazioni di debugging che ne compromettono la leggibilità, mentre `-S` è l'opzione utilizzata per indicare a `gcc` di compilare il codice C generando codice Assembly. Il codice Assembly generato da `gcc` è stato in alcuni casi modificato per renderlo più leggibile: per esempio, per il codice MIPS i numeri dei registri (`$0`, `$1`, ... `$31`) generati da `gcc` sono stati sostituiti dai nomi simbolici (`$zero`, `$at`, ... `$ra`); inoltre, alcune istruzioni `nop` sono state eliminate e le istruzioni sono state riordinate secondo il loro ordine logico.

5.1 Semplici Espressioni Aritmetico/Logiche

Il primo esempio mostra come scrivere in Assembly semplici espressioni aritmetiche o logiche ed evidenzia come in questo caso le ISA RISC (nel nostro caso MIPS ed ARM) permettono di scrivere in modo semplice codice che può essere facilmente capito. Al contrario, la traduzione in Assembly Intel è un po' più macchinosa. L'esempio considerato si basa sulla traduzione in Assembly dell'espressione C

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

secondo le ISA MIPS, ARM ed x86_64.

Per l'ISA MIPS, si assuma che i valori delle variabili `g`, `h`, `i` e `j` siano contenuti nei registri `$a0`, `$a1`, `$a2` ed `$a3`, salvando il risultato nel registro `$v0`. Una volta mappate le variabili sui registri, la conversione in Assembly dell'espressione è abbastanza naturale.

```
add     $a0, $a0, $a1
add     $v0, $a2, $a3
sub     $v0, $a0, $v0
```

La prima istruzione (**add** $a0, $a0, $a1) memorizza in `$a0` la somma `g + h`, la seconda istruzione (**add** $v0, $a2, $a3) memorizza in `$v0` la somma `i + j`, mentre la terza istruzione calcola la differenza fra questi due valori e la salva in `$v0`, ottenendo il risultato finale desiderato.

Anche la traduzione in Assembly ARM è immediata, una volta mappate le variabili `g`, `h`, `i` e `j` sui registri `r0`, `r1`, `r2` ed `r3`. Salvando il risultato nel registro `r0`, il codice Assembly risultante è:

```
adds    r1, r0, r1
adds    r3, r2, r3
subs    r0, r1, r3
```

Si noti che sono state utilizzate le istruzioni `adds` e `subs`, vale a dire le varianti delle due istruzioni che aggiornano i flag, ma anche le "semplici" `add` e `sub` (che non aggiornano i flag) avrebbero generato il risultato desiderato.

La conversione in Assembly Intel (assumendo che `g`, `h`, `i` e `j` siano in `%rdi`, `%rsi`, `%rcx` e `%rdx` e salvando il risultato in `%rax`) è leggermente più complessa, a causa del fatto che le normali istruzioni di

somma non possono specificare l'indirizzo destinazione (come per le ISA MIPS ed ARM) ma hanno una destinazione implicita:

```
leaq    (%rdi,%rsi), %rax
addq    %rcx, %rdx
subq    %rdx, %rax
```

Il problema è risolto utilizzando l'istruzione `leaq` per sommare `%rdi` e `%rsi` salvando il risultato in un registro diverso (`%rax`). Si ricordi infatti che `lea <addr>, <dst>` (load effective address) calcola l'indirizzo `<addr>` secondo base, spiazzamento ed indice shiftato, salvandolo in `<dst>`. Quindi, usando spiazzamento 0 e shift 0 per l'indice `lea` permette di sommare il contenuto del registro base (`%rdi` in questo caso) al contenuto del regitro indice (`%rsi` in questo caso) salvando il risultato nel registro destinazione (`%rax` in questo caso). Sebbene l'istruzione `lea` sia stata inizialmente progettata per effettuare il calcolo di indirizzi di memoria, risulta spesso utile come semplice istruzione aritmetica che permette di specificare un registro destinazione diverso dal secondo operando.

5.2 Accessi alla Memoria / Array

Il secondo esempio mostra come scrivere in Assembly codice che accede alla memoria (ed in particolare a vari elementi di un array). Questo esempio evidenzia come le ISA RISC (in questo caso MIPS ed ARM) richiedano di utilizzare due istruzioni separate per accedere alla memoria, mentre l'ISA Intel permette di utilizzare meno istruzioni, risultando in codice più compatto. L'esempio considerato si basa sulla traduzione in Assembly MIPS, ARM ed Intel dell'assegnamento

```
a[12] = h + a[8];
```

assumendo che `a` sia un array di `int` ed una variabile di tipo `int` occupi 4 byte (quindi `a` è un array di elementi di 4 byte l'uno).

Per l'ISA MIPS, assumendo che il valore della variabile `h` sia contenuto nel registro `$a0` e che l'indirizzo dell'array `a` sia contenuto nel registro `$a1` si ottiene

```
lw      $v0,32($a1)
add     $a0,$v0,$a0
sw      $a0,48($a1)
```

Come si può notare è necessaria una prima istruzione di load (`lw $v0,32($a1)`) per caricare il valore di `a[8]` in un registro (in questo caso viene usato il registro `$v0`, il cui valore non deve essere preservato). E' anche importante notare che l'indirizzo di `a[8]` è `32($a1)` (e non `8($a1)` come si potrebbe inizialmente sospettare), in quanto ogni elemento dell'array è grande 4 byte e $4 \cdot 8 = 32$. Una volta caricato `a[8]` in `$v0`, tale valore può essere sommato con il valore di `h` (contenuto in `$a0`) e finalmente salvato in memoria tramite `sw`. Ancora, si noti come l'indirizzo di `a[12]` sia `48($a1)` in quanto $12 \cdot 4 = 48$.

La versione ARM (assumendo che il valore di `h` sia contenuto in `r0` e l'indirizzo di `a` sia contenuto in `r1`) è simile

```
ldr     r3, [r1, #32]
add     r0, r3, r0
str     r0, [r1, #48]
```

In questo caso, `a[8]` è caricato in `r3` e sommato ad `r0` per poi salvare il risultato in `a[12]` tramite `str r0, [r1, #48]`.

Come accennato, la versione Intel (assumendo che il valore di `h` sia contenuto in `%edi` e l'indirizzo di `a` sia contenuto in `%rsi`) è più semplice (utilizzando solo 2 istruzioni Assembly) in quanto l'istruzione `addl` può avere un argomento in memoria. Il risultato viene poi salvato in `a[12]` tramite `movl`.

```
addl    32(%rsi), %edi
movl    %edi, 48(%rsi)
```

5.3 Condizioni e Cicli

Dopo aver visto come utilizzare le istruzioni aritmetiche e logiche per implementare il calcolo di semplici espressioni e come accedere alla memoria, il terzo esempio mostra come utilizzare l'esecuzione condizionale

di istruzioni Assembly per implementare il costrutto di selezione `if`. Tradizionalmente, questo costrutto è implementato in Assembly usando le istruzioni di salto condizionale descritte nelle Sezioni 2.4, 4.4 e 3.4; alcuni ISA permettono però l'esecuzione condizionale di altre istruzioni (per esempio, ARM permette l'esecuzione condizionale di tutte le istruzioni, mentre Intel fornisce l'istruzione di move condizionale `cmov` e questa feature può essere utile per implementare alcuni costrutti `if`). L'esempio preso in considerazione si basa sulla traduzione in Assembly del seguente codice C:

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

Assumendo che i valori di `g`, `h`, `i` e `j` siano contenuti in `$a0`, `$a1`, `$a2` e `$a3` e che il valore di `f` vada salvato in `$v0`, la traduzione in Assembly MIPS è:

```
        bne     $a2, $a3, L2
        add     $v0, $a0, $a1
        j       L3
L2:
        sub     $v0, $a0, $a1
L3:
```

In questo caso, l'istruzione `bne` viene utilizzata per saltare alla label `L2` se i valori di `i` e `j` sono diversi (si noti come il predicato usato per decidere l'esecuzione condizionale sia basato sul confronto fra i valori contenuti in due registri general purpose). Nel caso in cui il salto non sia effettuato (`i == j`), la somma di `g` ed `h` viene salvata in `$v0` e si salta alla label `L3`; altrimenti, in `$v0` viene salvata la differenza fra `g` ed `h` (istruzione `sub`). Questa è la tecnica generalmente utilizzata per implementare costrutti `if ... else` usando l'Assembly MIPS.

La versione ARM è più interessante, in quanto permette di implementare il costrutto `if ... else` senza usare istruzioni di salto, ma sfruttando la possibilità di esecuzione condizionale di istruzioni aritmetiche offerta dall'ISA ARM:

```
cmp     r2, r3
addeq   r0, r0, r1
subne   r0, r0, r1
```

La prima istruzione `cmp` setta i flag (in particolare, per questo esempio è di interesse lo zero flag Z) confrontando i valori di `r2` ed `r3` (in cui si assume siano contenuti i valori di `i` e `j`). Se i due valori sono uguali (condizione `eq`), viene eseguita la somma fra `r0` ed `r1` (che si assumono essere `g` ed `h`), altrimenti (condizione `ne`) si esegue la sottrazione (operazione `sub`). Si noti che nel codice originariamente generato da `gcc` la terza istruzione era `rsbne r0, r1, r0`, che è equivalente a `subne r0, r0, r1` (sottrazione inversa invece che sottrazione, invertendo i due operandi).

Una possibile implementazione usando l'Assembly Intel (assumendo che i valori di `g`, `h`, `i` e `j` siano memorizzati in `%rcx`, `%rdx`, `%rdi` e `%rsi` e che il risultato `f` debba essere memorizzato in `%eax`) è la seguente:

```
        cmpq    %rcx, %rdx
        jne     L2
        leaq    (%rdi,%rsi), %rax
        jmp     L3
L2:
        movq    %rdi, %rax
        subq    %rsi, %rax
L3:
```

Si noti come rispetto alle versioni MIPS ed ARM questa versione sia complicata dal fatto di usare `lea` invece di `add` (al proposito, vedere il primo esempio in Sezione 5.1) e dal fatto di dover usare una `mov` oltre a `sub` per salvare la differenza fra `g` e `h` in `%rax` (questo perché le istruzioni Intel hanno una destinazione implicita uguale al secondo operando). Si noti inoltre come sia necessario utilizzare un'istruzione `cmpq` (analogamente a quanto fatto nel caso ARM) per settare i flag prima di poter usare l'istruzione di salto condizionale `jne` (salta se `i` e `j` hanno valori diversi).

Come anticipato, l'implementazione può essere semplificata usando l'istruzione `cmov` (`mov` condizionale):

```
        leaq    (%rdi,%rsi), %rax
        subq    %rsi, %rdi
        cmpq    %rcx, %rdx
        cmovne  %rdi, %rax
```

Se invece di usare un test di uguaglianza (`i == j`) nel predicato il costrutto di selezione if avesse usato un confronto più complesso, la traduzione in Assembly MIPS si sarebbe leggermente complicata, mentre le versioni ARM ed Intel sarebbero rimaste più o meno uguali. Per esempio, si consideri il seguente codice C:

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

Assumendo gli assegnamenti registro/variabile visti nel precedente esempio, la traduzione in Assembly MIPS diventa:

```
        slt     $a2, $a2, $a3
        beq     $a2, $zero, L2
        add     $v0, $a0, $a1
        j       L3
L2:
        sub     $v0, $a0, $a1
L3:
```

Si noti come sia stato necessario introdurre l'istruzione di confronto `slt` prima del salto condizionale.

Il codice ARM corrispondente è invece:

```
        cmp     r2, r3
        addlt   r0, r0, r1
        subge   r0, r0, r1
```

Si noti come rispetto alla versione precedente siano semplicemente cambiate le condizioni di esecuzione di `add` e `sub`: invece di `eq` ed `ne` si usano `lt` e `ge`.

Anche le versioni Intel sono simili alle precedenti; senza usare `cmov` si ottiene:

```
        cmpq    %rcx, %rdx
        jge     L2
        leaq    (%rdi,%rsi), %rax
        jmp     L3
L2:
        movq    %rdi, %rax
        subq    %rsi, %rax
L3:
```

mentre usando `cmov` la traduzione in Assembly è:

```
        leaq    (%rdi,%rsi), %rax
        subq    %rsi, %rdi
        cmpq    %rcx, %rdx
        cmovge  %rdi, %rax
```

Ancora una volta, cambiano solo le condizioni usate per i salti o per `cmov`: si usa `ge` invece di `le`.

L'esempio successivo mostra come implementare un ciclo (con condizione di terminazione basata sui valori contenuti in un array) in Assembly. In questo caso, la traduzione in Assembly più immediata utilizza sempre le istruzioni di salto condizionale, anche con ISA (come ARM) che permettono l'esecuzione condizionale di altre istruzioni. L'esempio è basato sul seguente codice C (ancora una volta, `a` è un array di `int`, aventi dimensione 4 byte):

```
i = 0;
while (a[i] == k)
  i += 1;
```

Utilizzando l'ISA MIPS (ed assumendo che il valore di `k` sia inizialmente contenuto in `$a0`, l'indirizzo di `a` sia inizialmente contenuto in `$a1` ed il valore di `i` debba essere salvato in `$v0`) il ciclo può essere tradotto come (si noti l'utilizzo della macro `move` per inizializzare il valore di `$v0`):

```
        move    $v0, $zero
L1:
        sll     $t1, $v0, 2
        add     $t1, $t1, $a1
        lw      $t0, 0($t1)
        bne     $t0, $a0, L2
        addi    $v0, $v0, 1
        j       L1

L2:
```

Si noti come per poter confrontare `a[i]` con `k` sia sempre necessario caricarne il valore all'interno di un registro di appoggio (`$t0` in questo caso). Inoltre, le modalità di indirizzamento non troppo potenti offerte dall'ISA MIPS costringono a moltiplicare `$v0` per 4 (`sll $t1, $v0, 2`) ad ogni ciclo.

Il compilatore `gcc` traduce invece il codice come:

```
        lw      $a2, 0($a1)
        bne     $a2, $a0, L2
        addi    $a1, $a1, 4

        move    $v0, $zero
L1:
        addi    $a1, $a1, 4
        lw      $v1, -4($a1)
        addi    $v0, $v0, 1
        beq     $v1, $a0, L1
        j       L3

L2:
        move    $v0, $0
L3:
```

In questo caso, il compilatore genera codice leggermente più complesso, probabilmente perché cerca di ottimizzare il caso in cui `salva[0] != k` (ciclo mai eseguito). Ancora una volta, per confrontare `a[i]` con `k` è necessario caricarne il valore all'interno di un registro di appoggio (`$a2` o `$v1`, entrambi registri il cui valore non deve essere preservato). Il problema causato dalla semplicità delle modalità di indirizzamento offerte dall'ISA MIPS in questo caso è risolto usando 2 registri come contatori: `$v0`, in cui è contenuto il valore di `i` e `$a1`, in cui è contenuto l'indirizzo del prossimo elemento dell'array da accedere (si noti come `$a1` sia aggiornato incrementandone il valore di 4, in quanto ogni elemento dell'array è grande 4 byte).

Utilizzando l'ISA ARM (assumendo che il valre di `k` sia inizialmente contenuto in `r1`, che l'indirizzo dell'array `a` sia inizialmente contenuto in `r0` e che il valore di `i` vada salvato in `r0`) la traduzione in Assembly generata da `gcc` è leggermente più compatta:

```
        ldr     r3, [r1]
        cmp     r0, r3
        bne     L2
        mov     r3, #0
L1:
        add     r3, r3, #1
        ldr     r2, [r1, #4]!
        cmp     r2, r0
        beq     L1
        b       L3
L2:
        mov     r3, #0
```

```
L3:
        mov     r0, r3
```

Si noti l'utilizzo della modalità di indirizzamento pre-indexed `ldr r2, [r1, #4]!` che incrementa `r1` di 4 prima di accedere alla locazione di memoria puntata da tale registro. Questo permette di evitare di aggiornare a mano il valore di `r1`. Ancora, si noti come due diversi registri (`r3` ed `r1`) siano utilizzati per contenere il valore dell'indice `i` e della locazione di memoria a cui accedere.

Per finire, vediamo la traduzione in Assembly Intel. Tale traduzione è fatta assumendo che il valore di `i` vada alla fine lasciato in `%rax`, mentre inizialmente l'indirizzo di `a` sia contenuto in `%rsi` ed il valore di `k` e sia contenuto in `%edi`; a tale proposito si noti come per `k` si sia utilizzato un registro a 32 bit (`%edi` e non `%rdi`) perché il valore di `k` è confrontato coi valori degli elementi dell'array, che abbiamo detto essere `int` rappresentati su 4 byte. In ogni caso, la traduzione in Assembly Intel risulta stavolta piu' compatta, mostrando la maggiore potenza di un ISA CISC:

```
        cmpl    (%rsi), %edi
        jne     L2
        movq    $0, %rax
L1:
        addq    $1, %rax
        cmpl    %edi, (%rsi,%rax,4)
        je      L1
        jmp     L3
L2:
        movq    $0, %rax
L3:
```

Si noti l'utilizzo dell'istruzione `cmpl %edi, (%rsi, %rax, 4)` che confronta direttamente il contenuto di una locazione di memoria col contenuto del registro `%edi`, senza bisogno di caricare in un registro il contenuto della memoria. Tale istruzione utilizza inoltre una modalità di indirizzamento più avanzata (registro indice `%rax` moltiplicato per 4) che permette di usare un unico registro come indice, senza bisogno di usare un ulteriore registro che viene aggiornato incrementandolo di 4.

5.4 Invocazione di Subroutine

I prossimi esempi hanno l'obiettivo di esemplificare le ABI e le convenzioni di chiamata usate da MIPS, ARM ed Intel, iniziando dal caso più semplice di *funzione foglia*. Una funzione foglia è una funzione che non invoca altre funzioni o subroutine, come la seguente:

```
int esempio_foglia(int g, int h, int i, int j)
{
 int f;

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

 return f;
}
```

La conversione in Assembly di una funzione foglia è semplificata dal fatto che non è necessario salvare il contenuto del link register (o return address register), dei registri contenenti gli argomenti, etc...

Secondo l'ABI MIPS o32, i 4 argomenti `g`, `h`, `i` e `j` sono passati nei registri `$a0`, `$a1`, `$a2` e `$a3`, mentre il valore di ritorno della funzione deve essere salvato nel registro `$v0`. Il codice Assembly risultante è quindi il seguente (si veda l'esempio della Sezione 5.1 per maggiori dettagli):

```
        addu    $a0, $a0, $a1
        addu    $v0, $a2, $a3
        subu    $v0, $a0, $v0
        jr      $ra
```

Si noti come il ritorno da subroutine è implementato dall'istruzione `jr $ra` (salta all'indirizzo contenuto nel return address register `$ra`).

Anche la traduzione in Assembly ARM (tenendo conto delle convenzioni di chiamata specificate dall'ABI ARM) è molto semplice (l'unica cosa da notare è l'utilizzo di `rsb` invece di `sub`, come discusso in Sezione 5.1):

```
esempio_foglia:
        add     r0, r0, r1
        add     r3, r2, r3
        rsb     r0, r3, r0
        bx      lr
```

Anche in questo caso il ritorno da subroutine è implementato saltando all'indirizzo di ritorno (che stavolta è contenuto nel registro `lr` - link register). Questo potrebbe essere fatto tramite `mov pc, lr` (o `mov r15, r14`), ma `gcc` preferisce usare un'istruzione di salto a registro (`bx`. In particolare, `bx lr` salta all'indirizzo contenuto nel link register `lr`.

Per quanto riguarda la traduzione in Assembly Intel, l'unico particolare a cui fare attenzione è che secondo l'ABI le variabili di tipo `int` sono rappresentate su 32 bit, mentre i "registri `r`" sono registri a 64 bit (questa non è una specificità dell'ISA Intel, ma è comune a molti ISA a 64 bit). I parametri verranno quindi passati in "registri `e`" (`%edi`, `%esi`, `%ecx` ed `%ecx`) e non in "registri `r`" (`%rdi`, `%rsi`, `%rcx` ed `%rcx`). A parte questo, anche la traduzione in Assembly Intel appare immediata (ancora, si noti l'utilizzo di `lea`):

```
esempio_foglia:
        leal    (%rdi, %rsi), %eax
        addl    %ecx, %edx
        subl    %edx, %eax
        ret
```

In questo caso, il ritorno da subroutine è implementato con l'istruzione `ret`, che è dedicata a questo compito.

La conversione di funzioni che invocano altre funzioni (che non sono quindi funzioni foglia) è più complessa. Si consideri per esempio il seguente codice:

```
int inc(int n)
{
  return n + 1;
}

int f(int x)
{
  return inc(x) - 4;
}
```

Mentre la conversione in Assembly MIPS della funzione `inc` (che è una funzione foglia) è banale, la funzione `f` è più complessa in quanto richiede di salvare il contenuto del registro `$ra` prima di invocare `inc`. Tale registro può essere salvato sullo stack decrementando di 4 byte il valore dello stack pointer `$sp` prima di salvare il return address `$ra` nella locazione puntata da `$sp` (utilizzando `sw`), come fatto dal seguente codice:

```
inc:
        addiu   $v0,$a0,1
        jr      $ra

f:
        addiu   $sp,$sp,-4
        sw      $ra,0($sp)
        jal     inc
        addiu   $v0,$v0,-4
        lw      $ra,0($sp)
        addiu   $sp,$sp,4
        jr      $ra
```

Si noti che il valore di `$ra` va recuperato dallo stack (ed il valore di `$sp` incrementato di 4 byte) prima di ritornare dalla funzione `f`.

In realtà il codice generato da `gcc` è leggermente diverso dal precedente in quanto l'ABI MIPS o32 è più complessa di quanto descritto precedentemente e prevede che lo stack frame (o record di attivazione) di una funzione abbia una dimensione multipla di 8. che `gcc` crei uno stack frame di 8 byte aggiungendo 4 byte di padding per arrivare ad un multiplo di 8, come nel codice seguente (che è il codice effettivamente generato da `gcc`):

```
inc:
        addiu   $v0, $a0, 1
        jr      $ra

f:
        addiu   $sp, $sp, -8
        sw      $ra, 4($sp)
        jal     inc
        addiu   $v0, $v0, -4
        lw      $ra, 4($sp)
        addiu   $sp, $sp,8
        jr      $ra
```

In ogni caso (sia considerando lo stack frame di 8 byte generato da `gcc` che quello generato "a mano"), questo esempio mostra come l'ISA MIPS complichi il codice Assembly generato per funzioni non foglia. Si noti inoltre che le dimensioni dello stack frame dipendono dall'ABI: lo stack frame di 8 byte mostrato sopra è quello generato usando l'ABI denominata "eabi" ed altri ABI avrebbero generato stack frame diversi. Per esempio, usando "o32" lo stack frame sarebbe stato di 32 byte: 8 byte di spazio per salvare il registro `$gp`, $4 * 4 = 16$ byte di spazio per i primi 4 parametri (anche se sono passati tramite registri), `$4` byte per salvare `$ra` e 4 byte di padding per arrivare ad un multiplo di 8 byte.

Il codice Assembly ARM è più semplice:

```
inc:
        add     r0, r0, #1
        bx      lr
f:
        str     lr, [sp, #-4]!
        bl      inc
        sub     r0, r0, #4
        ldr     pc, [sp], #4
```

Anche in questo caso la funzione `f` salva il link register sullo stack, ma il codice è semplificato dalla possibilità di usare indirizzamento pre-indexed (`str lr, [sp, #-4]!`) che in questo caso coincide con l'istruzione `push` dell'ISA Intel (decrementa `sp` di 4 e memorizza `lr` nella locazione di memoria puntata da `sp` dopo il decremento). Inoltre, il fatto che il program counter `pc` sia un registro "visibile" (e più o meno general purpose) permette di effettuare contestualmente il `pop` del valore del link register precedentemente salvato sullo stack ed il caricamento del program counter con tale valore. Infatti, `ldr pc, [sp], #4` carica direttamente dallo stack nel program counter il valore del link register, analogamente a quanto fatto dall'istruzione `ret` dell'ISA Intel. Si noti come per prelevare un valore dallo stack (`pop`) si utilizza l'indirizzamento post-indexed (il contrario del pre-indexed usato per pushare il valore sullo stack).

Si noti che il codice generato da `gcc` è leggermente diverso, in quanto l'ABI usata da `gcc` su CPU ARM prevede che lo stack sia allineato a multipli di 8 byte. Per questo motivo, uno stack frame deve avere dimensione multipla di 8 byte ed anche se si deve salvare un solo registro sullo stack `gcc` decrementa il registro `sp` di 8 byte. Invece che usare 4 byte di padding (come fatto per MIPS), `gcc` preferisce salvare sullo stack un ulteriore registro (anche se non viene utilizzato), scelto arbitrariamente. In questo caso, il codice generato dal compilatore `gcc` salva sullo stack anche il registro `r4` usando una store multipla (ed una load multipla per recuperarlo).

```
inc:
        add     r0, r0, #1
        bx      lr
f:
        stmfd   sp!, {r4, lr}
        bl      inc
        sub     r0, r0, #4
```

```
        ldmfd   sp!, {r4, pc}
```

Per finire il codice Intel in questo caso è nettamente più semplice, grazie al fatto che `call` e `ret` già gestiscono il salvataggio (e recupero) dell'instruction pointer sullo stack:

```
inc:
        leal    1(%rdi), %eax
        ret

f:
        call    inc
        subl    $4, %eax
        ret
```

Ancora una volta, si noti come l'istruzione `lea` (`leal 1(%rdi), %eax` in questo caso) sia usata per fare una somma specificando un registro destinazione.

5.5 Un Esempio più Complesso: Ordinamento di Array

Come esempio di conversione in Assembly di codice più complesso, si consideri l'algoritmo di *ordinamento per inserimento diretto* (*insert sort*) implementato dal seguente codice C:

```
void sposta(int v[], int i)
{
  int j;
  int appoggio;

  appoggio = v[i];
  j = i - 1;

  while ((j >= 0) && (v[j] > appoggio)) {
    v[j + 1] = v[j];
    j = j - 1;
  }
  v[j + 1] = appoggio;
}

void ordina(int v[], int n)
{
  int i;

  i = 1;
  while (i < n) {
    sposta(v, i);
    i = i + 1;
  }
}
```

Si noti che l'algoritmo è stato volutamente diviso in due subroutine (`ordina()` e `sposta`) per mostrare come convertire sia funzioni foglia che non foglia.

TBD: Commentare il codice Assembly!!!

Una prima conversione in Assembly MIPS fatta "a mano" può apparire come segue:

```
sposta:         # a0 = v, a1 = i
                # t0 = j * 4, t1 = appoggio
        sll $t0, $a1, 2
        add $t2, $a0, $t0       # t2 = &v[i]
        lw  $t1, 0($t2)         # appoggio = v[i]
        addi $t0, $t0, -4

ciclo:
```

```
        slt $t3, $t0, $zero     # t3 = 1 se t0 < 0 (se j < 0)
        bne $t3, $zero, out     # esci se j < 0
        add $t2, $a0, $t0       # t2 = &v[j]
        lw  $t4, 0($t2)         # t4 = v[j]
        slt $t3, $t1, $t4       # t3 = 1 se t1 < t4: t3 = 1 se appoggio < v[j]
        beq $t3, $zero, out     # esci se appoggio >= v[j]
        addi $t5, $t0, 4        # t5 = (j + 1) * 4
        add $t2, $a0, $t5       # t2 = &v[j + 1]
        sw $t4, 0($t2)
        addi $t0, $t0, -4
        j ciclo

out:
        addi $t5, $t0, 4
        add $t2, $a0, $t5       # t2 = &v[j + 1]
        sw $t1, 0($t2)
        jr $ra

ordina:                 # a0 = v, a1 = n
                        # s0 = i
        addi $sp, $sp, -12
        sw   $s0, 0($sp)
        sw   $ra, 4($sp)
        sw   $a1, 8($sp)

        addi  $s0, $zero, 1             # i = 1
loop_ordina:
        slt   $t0, $s0, $a1             # t0 = 1 se i < n
        beq   $t0, $zero, out_ordina
        add   $a1, $s0, $zero
        jal   sposta
        lw    $a1, 8($sp)
        addi  $s0, $s0, 1
        j     loop_ordina
out_ordina:
        lw    $s0, 0($sp)
        lw    $ra, 4($sp)
        addi  $sp, $sp, 12
        jr    $ra
```

Il codice effettivamente generato da `gcc` sarebbe però più complesso:

```
sposta:
        sll     $v0, $a1, 2
        addu    $v0, $a0, $v0
        lw      $t0, 0($v0)
        addiu   $v0, $a1, -1
        bltz    $v0, L2

        move    $a2, $v0
        sll     $v1, $v0, 2
        addu    $v1, $a0, $v1
        lw      $v1, 0($v1)
        slt     $a3, $t0, $v1
        beq     $a3, $zero, L2

        sll     $a1, $a1, 2
        addu    $a1, $a0, $a1
        li      $t1, -1                 # 0xffffffffffffffff
L3:
```

```
        addiu   $a2, $a2, 1
        sll     $a2, $a2, 2
        addu    $a2, $a0, $a2
        sw      $v1, 0($a2)
        addiu   $v0, $v0, -1
        beq     $v0, $t1, L2

        move    $a2, $v0
        addiu   $a1, $a1, -4
        lw      $v1, -4($a1)
        slt     $a3, $t0, $v1
        bne     $a3, $zero, L3

L2:
        addiu   $v0, $v0, 1
        sll     $v0, $v0, 2
        addu    $a0, $a0, $v0
        sw      $t0, 0($a0)
        jr      $ra

ordina:
        addiu   $sp, $sp, -16
        sw      $ra, 12($sp)
        sw      $s2, 8($sp)
        sw      $s1, 4($sp)
        sw      $s0, 0($sp)
        move    $s1, $a1
        slt     $v0, $a1, 2
        bne     $v0, $zero, L5

        move    $s2, $a0
        li      $s0, 1                  # 0x1
L7:
        move    $a0, $s2
        move    $a1, $s0
        jal     sposta

        addiu   $s0, $s0, 1
        bne     $s0, $s1, L7

L5:
        lw      $ra, 12($sp)
        lw      $s2, 8($sp)
        lw      $s1, 4($sp)
        lw      $s0, 0($sp)
        addiu   $sp, $sp, 16
        jr      $ra
```

Codice ARM:

```
sposta:
        mov     r2, r1, asl #2
        add     r3, r0, r2
        ldr     ip, [r0, r1, asl #2]
        subs    r1, r1, #1
        bmi     L2
        ldr     r2, [r3, #-4]
        cmp     ip, r2
        bge     L2
L3:
```

```
        str     r2, [r3], #-4
        sub     r1, r1, #1
        cmn     r1, #1
        beq     L2
        ldr     r2, [r3, #-4]
        cmp     ip, r2
        blt     L3
L2:
        add     r1, r1, #1
        str     ip, [r0, r1, asl #2]
        bx      lr
ordina:
        cmp     r1, #1
        bxle    lr
        stmfd   sp!, {r4, r5, r6, lr}
        mov     r5, r1
        mov     r6, r0
        mov     r4, #1
L8:
        mov     r1, r4
        mov     r0, r6
        bl      sposta
        add     r4, r4, #1
        cmp     r5, r4
        bne     L8
        ldmfd   sp!, {r4, r5, r6, pc}
```

Ecco invece il codice Intel (che appare più semplice):

```
sposta:         # rdi = v, rsi = i
                # rax = j, r10d = appoggio
        movq  %rsi, %rax
        movl  (%rdi, %rax, 4), %r10d     # appoggio = v[i]
        dec   %rax

ciclo:
        cmpq  $0, %rax           # confronta 0 e rax
        jl    out                # esci se j < 0
        movl  (%rdi, %rax, 4), %r11d     # metti v[j] in %r11
        cmpl  %r10d, %r11d       # confronta v[j] e appoggio
        jle   out                # se v[j] < appoggio, esci
        movl  %r11d, 4(%rdi, %rax, 4)
        dec   %rax
        jmp   ciclo

out:
        movl  %r10d, 4(%rdi, %rax, 4)
        ret

ordina:                 # rdi = v, rsi = n
                        # rbx = i
        pushq  %rbx

        movq  $1, %rbx
loop_ordina:
        cmp   %rbx, %rsi
        jle   out_ordina
        pushq %rsi
        movq  %rbx, %rsi
        call  sposta
```

```
        popq   %rsi
        inc    %rbx
        jmp    loop_ordina
out_ordina:
        popq   %rbx
        ret
```

Codice generato da gcc:

```
sposta:
        movslq  %esi, %rax
        leaq    0(,%rax,4), %r8
        movl    (%rdi,%rax,4), %ecx
        subl    $1, %esi
        js      L2
        movslq  %esi, %rdx
        movl    -4(%rdi,%r8), %eax
        cmpl    %eax, %ecx
        jge     L2
L4:
        movl    %eax, 4(%rdi,%rdx,4)
        subl    $1, %esi
        cmpl    $-1, %esi
        je      L2
        movslq  %esi, %rdx
        movl    (%rdi,%rdx,4), %eax
        cmpl    %eax, %ecx
        jl      L4
L2:
        movslq  %esi, %rsi
        movl    %ecx, 4(%rdi,%rsi,4)
        ret

ordina:
        cmpl    $1, %esi
        jle     L12
        pushq   %r12
        pushq   %rbp
        pushq   %rbx
        movl    %esi, %ebp
        movq    %rdi, %r12
        movl    $1, %ebx
L8:
        movl    %ebx, %esi
        movq    %r12, %rdi
        call    sposta
        addl    $1, %ebx
        cmpl    %ebx, %ebp
        jne     L8
        popq    %rbx
        popq    %rbp
        popq    %r12
L12:
        ret
```

5.6 Ulteriori Esempi

In questa sezione vengono presentati alcuni ulteriori esempi di conversione da C ad Assembly, da potersi usare come esercizi. Questi esempi non sono però commentati; per ogni esempio è semplicemente

presentato il codice C, seguito dal codice Assembly per MIPS, ARM ed Intel generato da `gcc`.

Il primo esempio presenta una funzione che calcola iterativamente 2^n:

```
int potenza_due(int n)
{
  if (n == 0) {
    return 1;
  } else {
    int ret = 1;
    int i;

    for (i = 0; i < n; i++) {
      ret = ret * 2;
    }

    return ret;
  }
}
```

La traduzione in Assembly MIPS generata da `gcc` è:

```
potenza_due:
        beq     $a0, $zero, L4

        blez    $a0, L5

        move    $v1, $zero
        li      $v0, 1                  # 0x1
L3:
        sll     $v0, $v0, 1
        addiu   $v1, $v1, 1
        bne     $v1, $a0, L3

        jr      $ra

L4:
        li      $v0, 1                  # 0x1
        jr      $ra

L5:
        li      $v0, 1                  # 0x1
        jr      $ra
```

La traduzione in Assembly ARM generata da `gcc` è:

```
potenza_due:
        subs    r2, r0, #0
        ble     L4
        mov     r3, #0
        mov     r0, #1
L3:
        mov     r0, r0, asl #1
        add     r3, r3, #1
        cmp     r2, r3
        bne     L3
        bx      lr
L4:
        mov     r0, #1
        bx      lr
```

La traduzione in Assembly Intel generata da `gcc` è:

```
potenza_due:
        testl   %edi, %edi
        jle     L4
        movl    $0, %edx
        movl    $1, %eax
L3:
        addl    %eax, %eax
        addl    $1, %edx
        cmpl    %edx, %edi
        jne     L3
        ret
L4:
        movl    $1, %eax
        ret
```

Il secondo esempio effettua lo stesso calcolo usando la ricorsione invece dell'iterazione (quindi, è un esempio di funzione non foglia):

```
int potenza_due(int n)
{
  if (n < 1) {
    return 1;
  } else {
    return 2 * potenza_due(n - 1);
  }
}
```

La traduzione in Assembly MIPS generata da `gcc` è:

```
potenza_due:
        blez    $a0, L3

        addiu   $sp, $sp, -8
        sw      $ra, 4($sp)
        addiu   $a0, $a0, -1
        jal     potenza_due

        sll     $v0, $v0, 1
        j       L2

L3:
        li      $v0, 1                  # 0x1
        jr      $ra

L2:
        lw      $ra, 4($sp)
        addiu   $sp, $sp, 8
        jr      $ra
```

La traduzione in Assembly ARM generata da `gcc` è:

```
potenza_due:
        cmp     r0, #0
        ble     L3
        stmfd   sp!, {r4, lr}
        sub     r0, r0, #1
        bl      potenza_due
        mov     r0, r0, asl #1
        ldmfd   sp!, {r4, pc}
L3:
        mov     r0, #1
        bx      lr
```

La traduzione in Assembly Intel generata da `gcc` è:

```
potenza_due:
        movl    $1, %eax
        testl   %edi, %edi
        jle     L6
        subq    $8, %rsp
        subl    $1, %edi
        call    potenza_due
        addl    %eax, %eax
        addq    $8, %rsp
L6:
        ret
```

Nel prossimo esempio, si considera una funzione di copia fra stringhe:

```
void copia_stringa(char *d, const char *s)
{
  int i = 0;

  while ((d[i] = s[i]) != 0) {
    i += 1;
  }
}
```

La traduzione in Assembly MIPS generata da `gcc` è:

```
copia_stringa:
        lb      $v0, 0($a1)
        sb      $v0, 0($a0)
        beq     $v0, $zero, L1

        move    $v0, $zero
L3:
        addiu   $v0, $v0, 1
        addu    $v1, $a1, $v0
        lb      $v1, 0($v1)
        addu    $a2, $a0, $v0
        sb      $v1, 0($a2)
        bne     $v1, $zero, $L3

L1:
        jr      $ra
```

La traduzione in Assembly ARM generata da `gcc` è:

```
copia_stringa:
        ldrb    r3, [r1]
        strb    r3, [r0]
        cmp     r3, #0
        bxeq    lr
L3:
        ldrb    r3, [r1, #1]!
        strb    r3, [r0, #1]!
        cmp     r3, #0
        bne     L3
        bx      lr
```

La traduzione in Assembly Intel generata da `gcc` è:

```
copia_stringa:
        movzbl  (%rsi), %eax
        movb    %al, (%rdi)
```

```
        testb   %al, %al
        je      L1
        movl    $0, %eax
L3:
        addl    $1, %eax
        movslq  %eax, %rcx
        movzbl  (%rsi,%rcx), %edx
        movb    %dl, (%rdi,%rcx)
        testb   %dl, %dl
        jne     L3
L1:
        ret
```

Funzione che calcola l'n-esimo numero di Fibonacci:

```
int fibonacci(int n)
{
  if(n < 2) {
    return 1;
  } else {
    return 3 * fibonacci(n - 1) - fibonacci(n - 2);
  }
}
```

La traduzione in Assembly MIPS generata da `gcc` è:

```
fibonacci:
        addiu   $sp, $sp, -16
        sw      $ra, 12($sp)
        sw      $s1, 8($sp)
        sw      $s0, 4($sp)
        move    $s0, $a0
        slt     $v0, $a0, 2
        bne     $v0, $zero, L3

        addiu   $a0, $a0, -1
        jal     fibonacci

        move    $s1, $v0
        addiu   $a0, $s0, -2
        jal     fibonacci

        sll     $v1, $s1, 1
        addu    $s1, $v1, $s1
        subu    $v0, $s1, $v0
        j       L2

L3:
        li      $v0,1                   # 0x1
L2:
        lw      $ra, 12($sp)
        lw      $s1, 8($sp)
        lw      $s0, 4($sp)
        addiu   $sp, $sp, 16
        jr      $ra
```

La traduzione in Assembly ARM generata da `gcc` è:

```
fibonacci:
        cmp     r0, #1
        ble     L3
        stmfd   sp!, {r4, r5, r6, lr}
```

```
        mov     r5, r0
        sub     r0, r0, #1
        bl      fibonacci
        mov     r4, r0
        sub     r0, r5, #2
        bl      fibonacci
        add     r4, r4, r4, lsl #1
        rsb     r0, r0, r4
        ldmfd   sp!, {r4, r5, r6, pc}
L3:
        mov     r0, #1
        bx      lr
```

La traduzione in Assembly Intel generata da `gcc` è:

```
fibonacci:
        movl    $1, %eax
        cmpl    $1, %edi
        jle     L6
        pushq   %rbp
        pushq   %rbx
        subq    $8, %rsp
        movl    %edi, %ebx
        leal    -1(%rdi), %edi
        call    fibonacci
        movl    %eax, %ebp
        leal    -2(%rbx), %edi
        call    fibonacci
        leal    0(%rbp,%rbp,2), %edx
        subl    %eax, %edx
        movl    %edx, %eax
        addq    $8, %rsp
        popq    %rbx
        popq    %rbp
L6:
        ret
```

Si consideri poi la funzione che calcola la somma dei primi n numeri naturali, in versione tail recursive:

```
int sum(int n, int acc)
{
  if (n > 0) {
    return sum(n-1, acc + n);
  } else {
    return acc;
  }
}
```

Per prima cosa, la funzione è stata compilata senza attivare l'ottimizzazione relativa alle chiamate in coda (quindi, `gcc` non ha rimosso le chiamate ricorsive). La traduzione in Assembly MIPS generata da `gcc` è:

```
sum:
        move    $v1, $a0
        move    $v0, $a1
        blez    $a0, L5

        addiu   $sp, $sp, -8
        sw      $ra, 4($sp)
        addiu   $a0, $a0, -1
        addu    $a1, $a1, $v1
        jal     sum
```

```
        lw      $ra, 4($sp)
        addiu   $sp, $sp,8
L5:
        jr      $ra
```

La traduzione in Assembly ARM generata da `gcc` è:

```
sum:
        cmp     r0, #0
        ble     L3
        stmfd   sp!, {r4, lr}
        add     r1, r0, r1
        sub     r0, r0, #1
        bl      sum
        ldmfd   sp!, {r4, pc}
L3:
        mov     r0, r1
        bx      lr
```

La traduzione in Assembly Intel generata da `gcc` è:

```
sum:
        movl    %esi, %eax
        testl   %edi, %edi
        jle     L6
        subq    $8, %rsp
        addl    %edi, %esi
        subl    $1, %edi
        call    sum
        addq    $8, %rsp
L6:
        ret
```

E' stata poi attivata l'opzione di `gcc` che ottimizza le chiamate in coda (`-foptimize-sibling-calls`), per vedere le differenze nel codice generato. La traduzione in Assembly MIPS generata da `gcc` è:

```
sum:
        move    $v0, $a1
        blez    $a0, L2

L4:
        addu    $v0, $v0, $a0
        addiu   $a0, $a0, -1
        bne     $a0, $zero, L4

L2:
        jr      $ra
```

La traduzione in Assembly ARM generata da `gcc` è:

```
sum:
        cmp     r0, #0
        ble     L2
L4:
        sub     r3, r0, #1
        add     r1, r1, r0
        mov     r0, r3
        cmp     r3, #0
        bne     L4
L2:
        mov     r0, r1
        bx      lr
```

La traduzione in Assembly Intel generata da `gcc` è:

```
sum:
        movl    %esi, %eax
        testl   %edi, %edi
        jle     L2
L4:
        addl    %edi, %eax
        subl    $1, %edi
        jne     L4
L2:
        ret
```