Appunti di Architettura degli Elaboratori

Marco Natali

6 maggio 2018

Capitolo 1

Le istruzioni del Computer e il linguaggio Assembly

Tutti i calcolatori, dai più recenti e prestazionali ai più vetusti ed opsolescenti, eseguono ed effettuano le loro attività interpretando soltanto il linguaggio macchina, composto solo da sequenze di numeri binari.
Per permettere lo sviluppo dei programmi sempre più complessi ed agevolare il lavoro dei programmatori è stato introdotto il linguaggio Assembly, simbolica rappresentazione delle istruzioni in linguaggio macchina, il quale permette di utilizzare delle label per identificare particolari celle di memoria che tengono informazioni o dati.
Nonostante il miglioramento rispetto al linguaggio macchina, l'Assembly non è ancora sufficiente facile da essere utilizzato dai programmatori per cui nel corso degli anni sono stati introdotti dei linguaggi ad alto livello, come ad esempio il C, per migliorare la produttività, la chiarezza e la leggibilità, in cui le istruzioni somigliano all'inglese scritto rispetto a sequenze interminabili di bit; i programmi scritti in linguaggi ad alto livello vengono poi convertiti in Assembly dal compilatore, come verrà visto nel paragrafo sulla catena programmativa.
I vantaggi del linguaggio Assembly rispetto ai linguaggi ad alto livello risiedono nel poter ottenere una maggiore ottimizzazione e migliori prestazioni, a discapito però della portabilità, in quanto il linguaggio assembly è dipendente dalla macchina, e della produttività, dato che per ottenere un risultato in assembly bisogna scrivere un maggior numero di codice rispetto ai linguaggi ad alto livello.
Noi nel corso Architettura degli Elaboratori utilizziamo una macchina MIPS, già analizzata in un precedente capitolo, in particolare una macchina MIPS a 32 bit.
Per poter testare e verificare il funzionamento dei programmi Assembly utilizziamo due simulatori:Qtspim, tool che permette soltanto l'esecuzione di un programma Assembly, mentre Mars è un ide per assembly che permette di scrivere,eseguire ed analizzare l'effetto di ogni operazione.
I programmi Assembly, vengono caricati in memoria per venire eseguiti per cui ogni istruzione ha un proprio indirizzo in memoria.
Un programma scritto in linguaggio Assembly MIPS contiene un segmento *data*, in cui è presente la rappresentazione binaria dei dati utilizzati nel programma, e un segmento *text*, in cui sono presenti le istruzioni del programma in cui deve essere il label *main* per indicare l'inizio del programma.

Nel linguaggio MIPS sono presenti 3 tipologie di istruzioni, tutte da 32 bit:

R type sono le istruzioni di tipo aritmetico logico che prevede la seguente distribuzione:

op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)

I registri rs e rt rappresentano i due operandi mentre il registro rd indica il registro dove memorizzare il risultato;il campo $shamt$ indica lo shift effettuato, campo usato solo nelle istruzioni di shift.
Il campo op e $funct$ specificano l'operazione base dell'istruzione e la variante dell'operazione rispetto all'opcode, specificato da op.

I type sono utilizzate per rappresentare le operazioni di lettura e scrittura in memoria, delle operazioni immediate e delle istruzioni di branch. La distribuzione della istruzione nei 32 bit è la seguente:

op(6)	rs(5)	rt(5)	constant or address(16)

Il campo $constantoraddress$ permette di specificare un numero costante o un indirizzo di memoria.

J type è utilizzata per rappresentare le istruzioni di Jump e prevede la seguente codifica:

op(6)	address(26)

1.1 Istruzioni aritmetiche, logiche e di accesso alla memoria

Nel Mips32 le istruzioni aritmetiche prevedono l'utilizzo di solo 3 registri in cui due indicano gli operandi dell'operazione e un registro è utilizzato per salvare il risultato;nel linguaggio assembly del MIPS32 si possono soltanto utilizzare i valori presenti nei registri per effettuare le operazioni aritmetiche. Le istruzioni aritmetiche, come anche quelle logiche, sono operazioni di tipo R e le più utilizzate sono le seguenti:

add rd,rs,rt effettua la somma tra i registri rs e rt e la salva nel registro rd.

sub rd,rs,rt effettua la sottrazione tra i registri rs e rt e la salva nel registro rd.

mul rd,rs,rt effettua la moltiplicazione tra i registri rs e rt e la salva in rd.

div rd,rs,rt effettua la divisione tra i due registri rs e rt e lo salva nel registro rd

sll rd,rt,shamt sposta a sinistra di n bit(shamt) il registro rt e lo salva in rd;rappresenta la moltiplicazione 2^n.

srl rd,rt,shamt sposta a destra di n bit(shamt) il registro rt e lo salva in rd e rappresenta la divisione per 2^n.

and rd,rs,rt effettua l'and logico bit a bit tra i registri rs e rt e lo salva nel registro rd.

or rd,rs,rt effettua l'or logico bit a bit tra i registri rs e rt e lo salva nel registro rd.

nor rd,rs,rt effettua il nor bit a bit tra i registri rs e rt;per implementare il $not(A)$ si effettua $A\ NOR 0$.

```
.data #inserimento dei dati conosciuti
numeri: .word 6,3,5
somma: .word 0
.text #parte con il codice sorgente del programmma
.globl main #Dichiarazione dei label globali
main:
        la $t1,numeri#carica l'indirizzo di numeri in un registro temporaneo
        la $t0,somma #carica l'indirizzo dove memorizzare il risultato della so
        lw $s1,0($t1) #carica il primo numero da numeri nel registro s1
        lw $s2,4($t1)
        lw $s3,8($t1)
        add $s4,$s1,$s2 #effettua la somma tra $s1 e $s2
        add $s4,$s4,$s3 #effettua la somma tra $s4 e $s3
        sw $s4,0($t0)#salva in memoria il risultato della somma tra i 3 numeri
        li $v0,10 #uscita dal programma
        syscall
```

La syscall utilizzata alla fine viene utilizzata per effettuare operazioni di input-output nel simulatore e queste istruzioni verranno introdotte nel paragrafo sulle procedure.
In una macchina MIPS a 32 bit, come già accennato, sono presenti soltanto 32 registri i quali non permettono l'utilizzo di dati più complessi di quelli primitivi, come i numeri e i caratteri, per cui per ovviare si utilizza la memoria per salvare e memorizzare i dati complessi, come array e strutture. Per permettere di utilizzare e scrivere i dati in memoria per eseguire le operazioni aritmetiche il MIPS32 fornisce le istruzioni di lettura e scrittura chiamate *load* e *store*.
Per caricare dalla memoria una word, ossia un dato a 32 bit, si utilizza l'istruzione lw il quale ha la seguente sintassi: $lw\ rt, offset(rs)$ L'istruzione lw è allineata a 4 bytes per cui cercare di accedere ad un indirizzo non allineato a 4bytes genera un eccezione mentre l'indirizzo da cui leggere i dati è determinato da $address = offset + rs$.
L'istruzione di salvataggio in memoria di una word è l'istruzione sw che ha la stessa struttura e sintassi dell'istruzione lw. In caso si abbia la necessità di caricare e/o salvare dati in memoria più piccoli o grandi di una word ci sono le relative istruzioni per bytes,halfword e double word da vedere nella documentazione MIPS disponibile nell'appendice(INSERIRE RIFERIMENTO AD APPENDICE).
Esempio:

```
.data
massimo: .word 0

.text
.globl main

main: la $t0,massimo
        li $v0,5 #syscall eseguira read_int
```

```
        syscall
        move $s0,$v0 #sposta l'input dato nel registro $s0

        li $v0,5
        syscall
        move $s1,$v0 # sposta il secondo numero in input in $s1

        bgt $s0,$s1,Else #Va ad Else se e solo se $s0 > $s1
        sw $s1,0($t0) #il massimo e' il secondo numero e lo salva in memoria
        j Exit

Else: sw $s0,0($t0)#il massimo e' $s0 e lo salva in memoria

Exit: li $v0,10
        syscall
```

I registri a differenza dei valori in memoria permettono di usare le istruzioni Aritmetiche e logiche, per cui andrebbero utilizzati i registri per le variabili utilizzate spesso in un programma per rendere l'esecuzione del programma il più veloce possibile e con minor consumo di energia.
Un istruzione similare a quella di caricamento e salvataggio in memoria, ossia hanno la stessa tipologia di codifica la I-type, è l'addizione immediata, fornita dall'istruzione *addi* $rt, rs, constant$.
Vi sono le versioni immediate di tutte le operazioni aritmetiche e logiche, tranne il nor.

1.2 Istruzioni di controllo e di ripetizione

A differenza dei normali calcolatori, i computer sono in grado di prendere delle decisioni, ossia la capacità di eseguire differenti istruzioni in base a determinati valori dati in input.
Questa capacità di prendere decisioni nei linguaggi ad alto livello viene rappresentata dalle istruzioni $if - else$ mentre MIPS prevede due istruzioni per fare tutto ciò:

beq register1,register2,L1 questa istruzione, chiamata *branch if equals* punta all'istruzione nel L1 in caso i due registri siano uguali.

bne register1,register2,L1 l'istruzione, chiamata *branch if not equal* punta al label L1 in caso i due registri abbiano valori diversi.

Questa due istruzioni branch sono chiamate *salti condizionali*, istruzioni che permettono di alterare il flusso di esecuzione in base al soddisfamento di una data condizione.
Esempio: Scrivere un programma MIPS Assembly che dice se due stringhe sono uguali
Le istruzioni di ripetizioni permettono la ripetizione di una serie di istruzioni fino al soddisfamento di una serie di condizioni;nei linguaggi ad alto livello viene effettuata tramite le istruzioni while,do-while e for mentre nel MIPS viene implementato nel seguente modo, con ad esempio un programma che determina la dimensione di una stringa.

```
.data
stringa: .asciiz "Hello World"
dim: .word 0
stringaDimensione: .ascii "La dimensione della stringa e' "
fine: .asciiz ""

.text
.globl main

main: la $t0,stringa #caricamento indirizzo stringa
      lb $s0,0($t0) #lettura primo carattere stringa
      la $t1,fine
      lb $s1,0($t1)
      move $s2,$zero


loop: beq $s0,$s1,exit #test termine stringa
      addi $s2,$s2,1 #aggiorna dimensione della stringa
      addi $t0,$t0,1 #aggiorna indice della stringa
      lb $s0,0($t0)#legge un carattere dalla memoria
      j loop #ritorna all'inizio del loop


exit: li $v0,4
      la $a0,stringaDimensione #stampa la frase
      syscall
      li $v0,1
      addu $a0,$zero,$s2 #stampa dimensione della stringa
      syscall
      li $v0,10
      syscall
```

Le istruzioni dentro il label *loop* sono chiamate *blocco di istruzioni* ossia istruzioni in cui non compare alcun branch, eccetto eventualmente alla fine, e senza alcun label di branch, eccetto eventualmente all'inizio. Il test di uguaglianza e disuguaglianza sono i test più popolari ma a volte si ha bisogno di sapere se una variabile è minore di un altra e ciò nel MIPS viene verificato tramite *slt*(set less than) in cui è memorizzato 1 se il primo registro è minore dell'altro; dell'istruzione slt esiste la variante immediate ed unsigned per permettere il confronto immediato con una costante e il confronto tra due numeri positivi.

1.3 Procedure in linguaggio Assembly

Le procedure sono una sequenza di istruzioni con un nome che risolvono un determinato compito e possono essere riutilizzate e servono per implementare l'astrazione e la mantenibilità. Le procedure implementano il concetto matematico di funzione e possono avere dei parametri formali e possono ritornare o meno dei valori alla parte di programma che lo chiama.
Le procedure, qualsiasi operazione svolgano, hanno le seguente esecuzione:

1. Posizionano i parametri in maniera tale che si possa accedervi.
2. Sposta il controllo del programma alla procedura.
3. Si ricavano le risorse di memoria necessarie per l'esecuzione della procedura.
4. si esegue il compito desiderato della procedura.
5. si salva il valore di ritorno in una posizione accessibile dal chiamante.
6. sposta il controllo del programma al punto in cui è stata chiamata la procedura.

. In quanto i registri sono il posto migliore per salvare i dati, il MIPS ha previsto dei registri appositi per le procedure all'interno dei 32 disponibili:

1. $a0-$a3: 4 registri per salvare 4 argomenti passati come parametri alla procedura.
2. $v0-$v1:2 registri per tenere 2 valori di ritorno della procedura
3. $ra: registro per tenere l'indirizzo del punto in cui viene chiamata la procedura

Il linguaggio assembly del MIPS prevede un istruzione chiamata $jalProcedureAddress$, per puntare ad un indirizzo e salvare simultaneamente l'indirizzo dell'istruzione successiva nel registro $ra e ovviamente codesta istruzione è utile ed usata per chiamare una procedura.
Per effettuare il ritorno al punto di origine e terminare l'esecuzione della procedura il linguaggio assembly del MIPS prevede l'istruzione $jrregister$ per effettuare il salto al contenuto del registro; nel caso di ritorno da una procedura si usa $jr\$ra$.
In caso si abbiano bisogno di più di 4 registri per gli argomenti e/o più di due registri per i valori di ritorno si utilizza lo stack per salvare i registri aggiuntivi necessari e lo stack prevede un puntatore all'ultimo elemento inserito, a cui il MIPS ha previsto un registro, chiamato $sp, per memorizzare l'indirizzo del puntatore dello stack. Lo stack è una struttura dati LIFO(Last in First Out) in cui l'inserimento, chiamato *push*, nel MIPS consiste nel diminuire il valore dello stack pointer in quanto lo stack cresce dall'alto verso il basso, mentre la rimozione, chiamata *pop*, avviene aumentando il valore dello stack pointer.
Per evitare di salvare e ripristinare i registri i cui i valori possono non venire più usati, come ad esempio i registri temporanei, il linguaggio MIPS separa i registri in due gruppi:

1. i registri non preservati dalla procedura come ad esempio i registri temporanei $t0-$t9
2. i registri preservati dalle procedure(in caso la procedura li usa devono essere ripristinati) come ad esempio i registri $s0-$s7

1.4 Procedure annidate

Le procedure che fino ad ora abbiamo visto sono delle procedure foglie, ossia procedure che non chiamano altre, però alcune volte per poter svolgere la propria

attività una procedura ne chiama un altra e ciò viene definita procedura annidata; un particolare tipo di procedura annidata è la ricorsione, in cui una procedura chiama se stessa.
Con le procedure annidate bisogna stare attenti ai registri infatti gli argomenti $a0-$a4 e il registro $ra creano conflitto tra le diverse procedure per cui bisogna salvarli nello stack per poter mantenere i risultati dopo la chiamata alla procedura ausiliaria, come si può notare nell'esempio:
Nello stack sono anche presenti le variabili locali non rappresentate da dei registri, come gli array locali o le strutture e il segmento dello stack contenente i registri salvati dalla procedura e le variabili locali si chiama *record di attivazione*. Alcuni software MIPS usano il registro $fp(frame pointer), il quale punta alla prima parola del record della procedura ed è utile rispetto allo stack pointer in quanto non cambia durante la procedura per cui il riferimento ad una variabile ha lo stesso offset, rendendo più leggibile e mantenibile il codice assembly della procedura.

Capitolo 2

Rappresentazione delle Informazioni

Nei calcolatori viene utilizzata la codifica binaria, in quanto il passaggio o meno della corrente è a due valori e ciò coincide con il numero di simboli nella codifica binaria.
Nella realtà i numeri hanno una cardinalità infinita mentre purtroppo i calcolatori, avendo dei limiti di memoria e di componestistica, possono rappresentare soltanto un numero finito di numeri, pari a $base^n$, stabilito al momento della progettazione della macchina. Le informazioni, di qualsiasi tipo siano, vengono rappresentati sui calcolatori come una sequenza di numeri per questo iniziamo prima l'analisi di come vengono rappresentati i numeri.

2.1 Rappresentazione Numeri interi e negativi

Per poter rappresentare un numero sia nella realtà che nei calcolatori si usa per convenzione la rappresentazione posizionale in cui la cifra più a destra rappresenta la potenza più alta ossia $274 = 2 * 100 + 7 * 10 + 4 * 1$.
La forma generale di un numero è $\sum_{i=-k}^{n} d_i * base^i$ e le base più comunemente utilizzate sono:

- base 10(decimale):base utilizzata comunemente dagli umani per rappresentare i numeri mentre non viene usata nei calcolatori; le cifre usate sono da 0 a 9
- base 2(binaria): base usata comunemente nei calcolatori elettronci e prevede soltanto le cifre 0 e 1;un unica cifra in base binaria viene chiamata *bit*.
- base 8(ottale): si utilizzano soltanto le prime 8 cifre da 0 a 7
- base 16(esadecimale):utilizzata solitamente per rappresentare le celle di memoria e prevede come cifre: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

La conversione tra le basi potenze di due sono immediate infatti basta espandere/restringere le cifre per rappresentare le cifre dell'altra base infatti ad esempio:

La conversione tra un numero binario ed un numero decimale prevede qualche calcolo in più: il metodo più utilizzato e valido per tutti i numeri è quello di sottrarre il numero con la più grande potenza di 2 minore del numero e ripetendo lo stesso procedimento con la differenza fino ad arrivare a 0.
Si inserisce la cifra 1 nelle posizioni che corrispondono alle potenze di 2 usate.
Per rappresentare i numeri negativi nei calcolatori vi sono 4 modalità:

modulo e segno si utilizza il bit più a sinistra per rappresentare il segno (0 per + e 1 per -) e i restanti $n-1$ bit sono il valore assoluto del numero. Non viene utilizzata nei calcolatori attuali in quanto si ha la problematica di rappresentare il doppio zero.

complemento a 1 prevede anch'esso di utilizzare il bit del segno(0 per i positivi e 1 per i negativi) e la negazione di un numero si ottiene invertendo tutti bit, compreso quello di segno. Ha anch'esso la problematica del doppio zero per cui non viene utilizzato nei calcolatori attuali.

complemento a 2 prevede il bit di segno e l'opposto di un numero si ottiene invertendo tutti i bit e poi alla fine aggiungere 1 al numero invertito. Con il complemento a due si risolve il problema del doppio zero ma si inserisce uno sfasamento di 1 tra i numeri negativi e i numeri positivi rappresentabili.

notazione in eccesso di 2^{m-1} si rappresent il numero memorizzando come somma del numero e 2^{m-1} ed è poco utilizzato in quanto confusionario e porta facilmente ad errori ed incomprensioni nella lettura del numero.

2.2 Rappresentazione numeri in virgola fissa e mobile

Per rappresentare i numeri con la virgola nei calcolatori vi sono due modalità:

virgola fissa Si riservano un numero di bit per la rappresentazione della parte intera e un altro quantitativo di bit per rappresentare la parte frazionaria. Questa modalità non permette di rappresentare molti numeri e l'errore di approssimazione è uguale in tutte le posizioni dell'asse ed ha senso se si sa il range preciso di valori da rappresentare.
Per convertire la parte frazionaria in base B in base 2 si procede per moltiplicazioni successive come si può vedere nell'esempio:

virgola mobile sono comunemente utilizzati nei linguaggi di programmazione per rappresentare i numeri con la virgola e si utilizza la notazione scientifica $n = f * base^{exp}$ con f indicante la parte frazionaria, per la rappresentazione. Essendoci infiniti modi per rappresentare un numero in notazione scientifica, ma in un calcolatore, avendo una capacità finita, si stabilisce una convenzione per permetterne di rappresentare la maggiore combinazione di numeri.

Al contrario della realtà i numeri in virgola mobile non sono densi per cui alcuni numeri non possono essere rappresentati in maniera esatta e si commette per ciò un certo errore di approssimazione.
A differenza della rappresentazione in virgola fissa, l'errore complessivo varia al variare del numero mentre l'errore relativo, ossia la percentuale di errore commessa, rimane costante in tutto l'asse rappresentato.

2.3 Rappresentazione dei caratteri e stringhe

I caratteri e le stringhe, sequenza di caratteri, vengono rappresentati nei calcolatori come numeri e attraverso una serie di convenzioni sono trasformati nel carattere desiderato.
Il primo standard per rappresentare i caratteri è l'ASCII(cerca nome completo), presente all'inizio a 7bit, in cui si riesce soltanto a convertire le lettere inglesi, i numeri e i caratteri speciali; successivamente è stato introdotto l'ASCII a 8bit, che causò dei problemi di comunicazione in quanto i caratteri aggiuntivi all'ASCII normale dipendevano e cambiavano in base alla nazione del computer. Per permettere di rappresentare un numero maggiore di caratteri e lingue è stato introdotta la codifica UNICODE, utilizzata ad esempio in Java, che permette la gestione dei caratteri delle principali lingue e vi sono diverse versioni di UNICODE come ad esempio vi è un unicode per rappresentare le emoji.

2.4 Rappresentazione dei valori derivati:Suoni,Immagini e Video