ale-cci

Architettura dei calcolatori elettronici

April 5, 2020

Introduzione

Calcolatore Elettronico

Un calcolatore elettronico è un sistema gerarchico che possiede le funzioni di elaborazione, memorizzazione, trasmissione e di controllo. Queste funzioni corrispondono in prima approssimazione agli elementi: CPU, memoria, sistema I/O e Bus.

La CPU (unità di controllo) è ulteriormente divisa in 4 parti:

- ALU: esegue le operazioni aritmetiche e logiche.
- Control Unit: comanda le unità del processore.
- Registri: memorie interne al processore, utilizzate per tenere temporaneamente i dati che il processore deve elaborare.
- Bus: Interconnessione interna per il trasferimento dati nel processore.

Architettura di Von Neumann

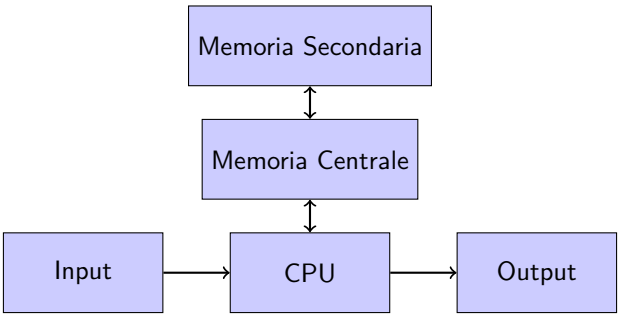
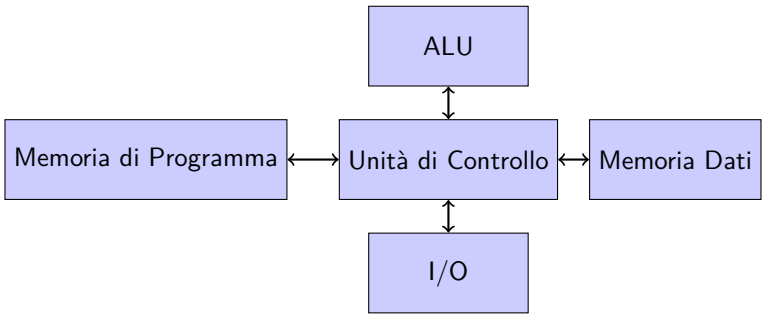


Figure 1: Computer secondo architettura di Von Neumann

La novità che porta l'architettura di Von Neumann è l'introduzione di una memoria. Prima di questa architettura, i programmi erano salvati esternamente al calcolatore in schede perforate. La memoria centrale (RAM) dell'architettura serviva per memorizzare temporaneamente i dati e permanentemente nella memoria secondaria (HD).

Come si può vedere dallo schema in figura 1, si interfaccia solamente con la memoria centrale.

Architettura di Harvard



La differenza fondamentale di questa architettura è la memoria del programma è separata dalla memoria dati, e le istruzioni devono obbligatoriamente passare dalla CPU. Questa architettura ha dato spunto a memorie separate come cache. L'idea di poter accedere memoria e dati velocizza le prestazioni del calcolatore.

Legge di Moore

1. Le prestazioni dei processori, e il numero di transistor ad esso relativo, raddoppiano ogni 18 mesi.
2. IL costo di una fabbrica di chip raddoppia da una generazione all'altra.

Legge di Amdahl

Nella storia dei computer, si è capito che il continuo aumento della frequenza di clock del numero di transistor del processore non era una cosa plausibile. Per questo si è iniziato a ragionare sul parallelismo.

Nel momento in cui voglio parallelizzare un algoritmo, ho due parti fondamentali: una componente sequenziale ed una componente parallelizzabile. Chiamata f la frazione di algoritmo parallelizzabile, ed ho a disposizione un numero di processori N, l'aumento di velocità di esecuzione S è calcolabile attraverso la legge di Amdahl:

$$S = \frac{1}{(1-f) + \frac{f}{N}}$$

È da tenere quindi presente che se utilizzo un grande numero di core per un algoritmo non parallelizzabile, il suo tempo di esecuzione sarà identico se eseguito con un solo core.

Il parallelismo è stato quindi stato adottato dai processori attraverso pipelining, coprocessori paralleli (processori dedicati a specifiche operazioni) ed architetture multicore.

RISC vs CISC

ISA

L'ISA (*Instruction set Architecture*) di un processore, non è altro che la lista di istruzioni disponibili al programmatore, interpretabili dal processore. Un' istruzione dell'ISA inviata al processore, viene prima trasformata in comandi di microarchitettura (linguaggio macchina) e poi eseguita dall'hardware. Un processore viene detto compatibile a livello di ISA con un altro processore, se tutte le istruzioni di quest'ultimo sono interpretabili dal processore.

La definizione di un ISA è la prima tra le diverse fasi della progettazione della CPU, ed in base a quanto verbosa (quanti comandi ne fanno parte), il processore si può definire di architettura CISC o RISC.

CISC

Un' ISA di tipo CISC (*Complex Instruction Set Computer*) facilita il compito del, fornendo a disposizione un vasto numero di istruzioni. Di contro ha che la sua realizzazione in hardware spesso risulta non efficiente. Inoltre le istruzioni utilizzate più di frequente dai programmatori sono approssimativamente il 20% di quelle disponibili.

RISC

All'esatto opposto ci sono le ISA di tipo RISC (*Reduced Instruction Set Computer*). È un ISA semplice più vicina all'hardware, hanno un'implementazione più veloce e più semplice. Fornisce un set di istruzioni ristretto ma efficiente. Di contro, essendo il numero di istruzioni a disposizione ridotto, scrivere un programma solitamente risulta più complesso e si impiega più tempo.

Confronto fra RISC e CISC

Altre differenze tra i due tipi di architettura, non ancora riportate, ma da tener presente sono:

RISC	CISC
▪ Istruzioni di lunghezza fissa	▪ Codice operativo e istruzioni di lunghezza variabile
▪ Decodifica più semplice	▪ Decodifica complessa, a più cicli di clock
▪ Unità di controllo cablata	▪ Unità di controllo microprogrammata
▪ pochi metodi di indirizzamento	▪ Molta flessibilità nei metodi di indirizzamento
▪ memoria allineata	▪ memoria non allineata
▪ molti registri di lunghezza fissa ed ortogonali	▪ pochi registri di varie lunghezze e non ortogonali
▪ processori load/store	

In una memoria allineata, i dati vengono disposti ad indirizzi multipli di n. Il vantaggio ovvio che si ha utilizzando questa struttura è un rapido accesso alla memoria. Ovviamente avere una memoria allineata porta con sé lo svantaggio di avere un maggiore consumo di memoria in caso dovessi salvare dei dati di grandezza minore di n.
I registri e la memoria sono collegati, l'approccio RISC cerca di lavorare con la memoria il meno possibile, attraverso poche modalità di indirizzamento e processori che accedono alla memoria attraverso

due sole istruzioni: load e store. Per evitare ripetuti accessi alla memoria, i dati vengono salvati temporaneamente in **registri** interni al processore. Questi registri possono essere ortogonali (ogni registro può effettuare ogni operazione) nel caso RISC o non ortogonali (esistono registri specifici per specifiche operazioni) nel caso CISC.

Tempo necessario al processore per eseguire un programma è dato dalla somma dei clock per instruction di ogni istruzione, moltiplicata al tempo di clock della CPU. Questo si traduce in:

$$T_{CPU} = T_{ck} \sum_i (NI_i \cdot CPI_i)$$

L'obbiettivo RISC è quello di eseguire la maggior parte delle istruzioni in un solo ciclo di clock, e rendere l'implementazione più semplice in modo da ridurre la durata del tempo di clock T_{CK}.

Microarchitettura CPU

La CPU

Rete logica combinatoria: rete priva di memoria, che dati ingressi da' delle uscite indipendentemente dal tempo. Rete sequenziale: rete che ha memoria, macchina a stati finiti, che dati degli ingressi fornisce delle uscite dipendenti dallo stato in cui si trova.

Dal punto di vista funzionale possiamo vedere la CPU, come composta da due parti: il data path e l'unità di controllo. Il data path di cui componente fondamentale è l'ALU, è una rete logica che si occupa di eseguire le istruzioni, mentre l'unita di controllo è una macchina a stati finiti che comanda il data path su quali istruzioni eseguire.
Possiamo schematizzare l'unità di controllo come una macchina a stati finiti a tre stati: fetch, decode ed execute.

Architettura di riferimento RISC

Analizziamo adesso un esempio di architettura RISC.

Gli indirizzi di memoria sono riferiti ai byte. Le istruzioni occupano sempre e solo una parola a 32 bit. Le istruzioni ed i dati si trovano sempre in indirizzi multipli di 4, per questo motivo il program counter è incrementato di 4 ad ogni istruzione. Avremo anche 32 registri di uso generale a 32bit. L'insieme dei registri è chiamato register file.

In generale l'esecuzione all'interno di un processore passa attraverso 5 fasi: una prima fase chiamata Instruction Fetch (IF) dove il processore carica dalla memoria l'istruzione, successivamente la decodifica nella fase (FT) e la esegue nella fase di execute (EX). Dopodiché in alcuni casi si ha un accesso alla memoria nella fase di Memory (ME), ed infine in alcuni casi ho una fase di write-back (WB), dove il risultato delle operazioni è riscritto nei registri.

Tutte le istruzioni di questa ISA, come già detto hanno una lunghezza fissa di 32bit, ed appartengono a 3 categorie:

- Un primo aritmetico logiche, Nei primi 6 bit opcode (codice operativo), seguiti da 5 che indicano il primo registro sorgente, altri 5 bit che indicano il secondo registro, 5 che indicano il registro destinazione ed i rimanenti 11 fanno riferimento all'operazione specifica dell'ALU (somma, sottrazione, ...).

  ```
  add  r4,r2,r5
  ```

- Un secondo formato utilizzato per accesso alla memoria e salti condizionati. I primi 6 bit di codice operativo, a seguire altri 5 ad indicare il primo registro ed altri 5 ad indicare il secondo registro (come nel primo caso) 16 che indicano un offset o un dato.

  ```
  je  r2,r3,0045h
  ```

- Un terzo ed ultimo caso utilizzato per i salti incondizionati. I primi 6 bit indicano sempre l'opcode ed i rimanenti indicano l'indirizzo di termine del salto.

  ```
  jmp 0045h
  ```

In questa prima architettura assumiamo che non ci sia lo stack (zona di memoria organizzata a pila LIFO), accessibile attraverso istruzioni apposite come `push` e `pop`.

Oltre a salto condizionato ed incondizionato esiste un terzo tipo di salto: la chiamata a funzione. È un tipo di salto incondizionato particolare, dato che quando eseguito è necessario salvare il valore del program counter per poter proseguire l'esecuzione una volta terminata la funzione.

Siccome non abbiamo uno stack, il program counter è salvato nell'indirizzo R_{31}

Per selezionare le operazioni da far eseguire all'ALU è specificato un ingresso chiamato opalu, dove vengono specificate le operazioni. Oltre all'uscita dell'operazione vengono tornati dei valori aggiuntivi come il flag di zero (messo ad uno quando il risultato dell'ALU è zero), ed il flag di segno (messo ad 1 quando il risultato è negativo).

Load e store permettono la lettura e scrittura in memoria.

Modelli di Memoria

Accesso alla memoria

Per scegliere gli operandi con i quali accedere alla memoria, si dividono le ISA in base a due numeri: il **numero di riferimenti diretti in memoria** indicati nelle istruzioni dell ALU ed il **numero di operandi indicati in modo esplicito nelle istruzioni**. Entrambi possono assumere solo valori compresi tra 0 e 3 inclusi.

Ad esempio nelle architetture chiamate *register-register*, il numero di riferimenti diretti in memoria è 0, ed il numero di operandi che indica in modi indicati in modo esplicito è 3. In altre parole le uniche operazioni che possono accedere in memoria sono LOAD e STORE.
Il numero di operandi indicati in modo esplicito indica il numero massimo di operandi specificati in modo esplicito come parametri di una funzione.

Quindi per effettuare un operazione come `c = a + b` sono necessarie le operazioni:

```
load    r1, var1
load    r2, var2
add     r1, r2, r3
store   var3, r3
```

Utilizzando lo stesso esempio per una architettura *register-memory*, (1, 2) otteniamo:

```
mov     AX, var1
add     AX, var2            ; AX funziona sia da sorgente
                            ; che destinazione
mov     var3, AX
```

In questo caso posso avere al massimo solo un operando che fa riferimento alla memoria.

Un' ultimo esempio di architettura è la *memory-memory* (3, 3) dove sia sorgente che destinazione sono completamente esplicitati.

Ordinamento della memoria

La memoria è sempre organizzata come un lungo array di celle a 8bit. Quando un dato di lunghezza più grande di 8bit deve essere salvato in memoria, può essere utilizzata sia la codifica **little endian** (memorizza l' Least Significant Bit all'indirizzo più basso), sia la **big endian** (all'indirizzo più basso viene salvato il Most Significant Bit).

Allineamento della memoria

Se la memoria è forzata a salvare i dati in modo allineato, allora riesco a leggere i dati di grandezza maggiore di 1 byte in un **singolo ciclo**. L'unico svantaggio è che per salvare dei dati di grandezza inferiore 4byte, avrò delle celle inutilizzate.
Se la memoria non è allineata, risparmio più spazio in memoria, ma l'accesso può richiedere più di un ciclo di CPU.

Memoria lineare e segmentata

Si definisce con **effective address**, l' indirizzo reale in memoria.
La **riallocazione della memoria** (RAM) si intende il riordinamento dei blocchi di memoria, in modo da raggruppare un unico blocco di memoria, tutti i blocchi non utilizzati, isolati dalla memoria in uso.

Il problema che porta con se la riallocazione di memoria, è che una volta che un blocco di memoria viene spostato, tutti gli effective address utilizzati nel codice contenuto al suo interno sono invalidati, e devono essere aggiornati uno ad uno dal processore con il nuovo effective address.
Per questo motivo alcune ISA preferiscono utilizzare un modello di memoria segmentato, in cui il codice utilizza **indirizzi relativi** anziché lavorare direttamente con gli effective address. Per accedere alla memoria attraverso un indirizzo relativo, vengono salvati in due registri (CS: Code Segment e DS:

Data Segment) l'indirizzo di memoria da cui partono codice e dati del programma. Al momento di una riallocazione, per un modello di memoria segmentato, l'unica cosa invalidata sono i due registri segmento di ciascun programma.

Modello di memoria Intel 8086

Nel caso di Intel 8086, la memoria viene vista come un gruppo di paragrafi e segmenti: i **paragrafi** sono una zona di memoria a 16bit, i quali non si possono sovrapporre, mentre un **segmento** è un unita logica indipendente formata da locazioni continue di memoria, di dimensione massima 64k, ha inizio ad un indirizzo di memoria multiplo di 16, (in modo da essere allineato ad un paragrafo) ed a differenza dei paragrafi, sono sovrapponibili.

La dimensione massima di un segmento (64k) deriva direttamente dalla dimensione massima che può avere un indirizzo relativo. Dato che l'accesso ad un indirizzo avviene attraverso i registri, la dimensione massima è 2^n, e per Intel 8086: $n = 16 \Rightarrow 2^{16} = 64k$.

L'indirizzo di inizio di un segmento è salvato in un indirizzo di memoria a 20bit, ed è ottenuto da un registro a 16bit moltiplicato per 16.

La sovrapposizione dei segmenti era utilizzata in DOS nel tipo di eseguibile '.com', file che utilizzavano il modello di memoria 'tiny'. Prevedeva un unico segmento a cui corrispondevano DS SS (Stack Segment) e CS. I segmenti erano sovrapposti solo come indirizzo, non come dati. Siccome tutto era contenuto in un unico segmento, tra codice, dati e stack non era concesso di superare i 64k.

Modalità di Indirizzamento

Formato di Istruzione

Per definire un' istruzione all'interno del linguaggio, è necessario definire: il **codice operativo** (numero operandi espliciti), gli operandi ed il risultato e l'indirizzo della prossima istruzione. Per salvare tutte queste caratteristiche in memoria, diventa fondamentale definire un formato in cui codificare e decodificare l'istruzione.

Modalità di indirizzamento

La modalità di indirizzamento decide come indicare l'indirizzo in memoria in cui prendere i dati.

Indipendentemente dal tipo di memoria utilizzata (lineare, segmentata...) e dal tipo di operazione da effettuare, per indicare all'istruzione richiesta dove trovare l' operando, posso:

- passarlo attraverso un registro
- passarne il valore all'istruzione (modalità immediata)
- leggerlo dalla memoria

Quello che cambia tra le varie ISA sono quante e quanto complesse sono le operazioni per l' accesso alla memoria. Più modalità di indirizzamento ho più diventa facile l'accesso, ma allo stesso tempo aumenta la complessità della rete logica.

Esistono diverse opzioni per quanto riguarda alla lettura dell' operando dalla memoria. In caso di accesso **diretto** viene indicato l'indirizzo a cui prendere il dato in memoria. Diversamente se viene utilizzata una modalità di indirizzamento **indiretta** viene indicato l'indirizzo di memoria dell' operando in un registro.
Entrambi gli approcci diretto ed indiretto possono combinarsi nella modalità di indirizzamento **base** dove il registro base utilizzato come offset è unito ad un indirizzo diretto per trovare la posizione in memoria dell' operando.

Esiste un ulteriore versione non implementata in tutte le ISA chiamato **indiciato**: indico due registri e l'indirizzo in memoria dell' operando è la somma dei due registri. Viene chiamato in questo modo perché il primo registro funziona da registro base, mentre il secondo si comporta da indice (es. Accesso ai dati di un vettore).

Altre tipologie di indirizzamento, (meno frequentemente adottate dalle ISA) sono: L'accesso **indiretto** dove viene indicata la cella di memoria contenente l'indirizzo dell' operando, la modalità di indirizzamento **scalato** che si comporta esattamente come l' indiciato, ma viene specificato un ulteriore valore di offset, e per finire le modalità di **autoincremento** e **autodecremento**, le quali funzionano esattamente come la modalità d'accesso tramite registro, solo che dopo la letture il valore contenuto nel registro viene automaticamente incrementato o decrementato.

Tramite registro	`mov BL AL`
Immediato	`mov BL, 12`
Diretto	`mov AX, [12]`
Indiretto tramite registro	`mov AX, [BX]`
Indiretto tramite indice	`mov AX, [SI]`

Figure 2: Rispettive istruzioni Assembly

Le modalità di accesso possono essere combinate: `mov AX [12 + BX + SI]` .

Se nella modalità di indirizzamento è presente il registro **BP**, il segmento di riferimento sarà **SS** (l'indirizzo è relativo allo stack), altrimenti il riferimento sarà sempre **DS** (segmento dati).

Se si vuole comunque accedere ad un altro segmento di memoria, è possibile effettuare un segment override, specificando il segmento di memoria dove si vuole accedere: `mov AX, [CX:BX + 5]`.

Modi di indirizzamento nel trasferimento di controllo

Con questo nome si intende come indicare il valore del program counter (o instruction pointer) al momento di un salto.
Normalmente viene indicato in modo diretto dalla istruzione di jump, e può essere espresso sia in modo assoluto, che in modo relativo. Nell' ISA Intel è disponibile anche gli indirizzamenti intrasegment ed intersegment, sia in modo diretto che indiretto.

Vengono chiamati intrasegment i salti che si trovano e terminano nello stesso code segment, mentre intersegment i salti che riguardano terminano in un code segment differente da quello di partenza. A seconda dei casi si può avere un indirizzamento diretto o indiretto, ovvero viene indicato direttamente il termine del salto o l'indirizzo di termine è contenuto in un registro.

Modi di indirizzamento I/O

Dal punto di vista dell' ISA esistono due metodi di indirizzamento: il **memory mapped I/O**, dove gli indirizzi per l'interazione con i dispositivi I/O sono contenuti in memoria, e **separated I/O**, dove gli spazi di indirizzamento I/O sono separati dalla memoria, per accedere ai dispositivi ho istruzioni diverse (`in` e `out`).

Per comunicare con i dispositivi I/O è possibile utilizzare un metodo ad indirizzamento diretto: `in AL, 100`, ma l'indirizzo massimo è limitato a 256. Se si vuole utilizzare indirizzi con valori maggiori a 256, è necessario utilizzare un metodo ad indirizzamento tramite registro. Il registro (a 16b) dedicato a queste operazioni è DX.

Tipi e struttura degli operandi

Sono i tipi di dato supportati dall' ISA. Possono essere di tipo intero (signed/unsigned), floating point (single, double o extended precision), caratteri (ascii/unicode), bool o multimediali.

Ed ovviamente è necessario scegliere quali operazioni sono previste dall' ISA per lavorare con i tipi di dato supportati.

Linguaggio Assembly 8086

In due parole

Non è case sensitive.
Ogni statement è terminato da `\n`, lo statement può proseguire alla riga successiva solo se questa comincia con il carattere `&`.
Gli identificatori hanno una lunghezza massima di 31 caratteri ed il nome non può iniziare con un numero.

Tipi di costante

```
mov ax, 13              ; decimale, anche 13D
mov ax, 13h             ; esadecimale (devono iniziare come un numero)
mov ax, 00100B          ; binario
mov ax, 13O             ; ottale

mov ax, 2.34            ; numeri reali
mov ax, 112E-3          ; rappresentabili anche in esponenziale

mov ax, 'T'             ; Costanti carattere
mov ax, 'test'          ; o anche stringa
```

Istruzioni per il trasferimento dati

```
mov dest, sorg          ; sposta il contenuto del secondo operando
                        ; nel primo
mov [bx], al            ; salva nell'indirizzo indicato da BX il
                        ; valore di al
xchg dest, sorg         ; scambia il contenuto dei due operandi
push w0rd               ; inserisce una word nello stack
pop  w0rd               ; estrae una word dallo stack
in   accum, porta       ; legge un dato dalla porta specificata
out  porta, accum       ; scrive un dato sulla porta specificata
```

Si ricorda che istruzioni come `mov [bx], [si]` non sono permesse perché siccome non stiamo utilizzando una macchina *memory-memory*, si può avere al più 1 riferimento alla memoria nella stessa istruzione.

Esistono altri trasferimenti non ammessi dalla `mov`:

- `mov ds, 100`, modificare direttamente il valore di un registro. Occorre utilizzare un registro general purpose:

  ```
  mov ax, 100
  mov ds, ax
  ```

- `mov dx, es`, trasferimento da segment register a segment register.
- `mov cs, 100`, qualsiasi trasferimento che abbia `cs` come destinazione. Ovvero cambiare il codice in esecuzione.

Lo stack pointer `sp` parte con valore iniziale 0xffff, ad ogni istruzione `push`, `sp` diminuisce di 2, mentre ad ogni `pop` aumenta di 2.

> Quando tolgo i dati dallo stack con pop, la cella di memoria non viene azzerata.

Istruzioni di aritmetica binaria

Operazioni di aritmetica binaria

```
; Operazioni ad 1 parametro
inc var1        ; Incrementa di 1 var1
dec var1        ; Decrementa di 1 var1

mul sorg    ; Moltiplicazione sorg * al oppure sorg * ax
div sorg    ; Divisione:      sorg / al oppure sorg / ax

imul sorg   ; mul con segno
idiv sorg   ; mul con segno

neg var1    ; nega il registro var1 (negato aritmetico, non binario)


; Operazioni a 2 parametri
add dest, sorg  ; Salvano entrambe il risultato dell'
sub dest, sorg  ; operazione in `dest`

cmp dest, sorg  ; uguale a sub ma non salva il
                ; risultato in dest

adc dest, sorg  ; add with carry: dest = dest + sorg + carry_flag
sbb dest, sorg  ; sub with carry: dest = dest - sorg - carry_flag
```

NOTE

Nelle operazioni a due parametri, entrambi i registri devono avere stessa dimensione. Ad esempio `add AX, BL` non è permesso.

Nel caso in cui `div` sia troppo grande per essere contenuto nel registro destinazione, o il divisore sia 0, viene generato un `int 0h` (Divisione per zero).

Sono supportati i formati **signed**, **unsigned**, numeri decimali **packed** [1] e **unpacked** [2].
Nel caso di numeri decimali unpacked, i 4 bit superiori devono essere a 0 se il numero è usato in un'operazione di moltiplicazione o divisione.

Operazioni su 32 bit

Considerando come unico numero a 32bit i registri `bx` e `ax` (con 16bit più significativi salvati in `ax` e 16 meno significativi salvati in `bx`), ed un altro numero a 32bit salvato in analogo modo in `dx` `cx`, somma e sottrazione possono essere eseguite nel seguente modo

```
add ax, cx  ; Somma parti meno significative
adc bx, ds  ; Somma parti più significative con carry

sub ax, cx  ; Analogo per sottrazione
sbb bx, ds
```

[1] Ogni byte contiene due numeri decimali, la cifra più significativa è allocata nei 4 bit superiori. Es: 35=0011.0101
[2] Ogni byte contiene un solo numero decimale BCD nei 4 bit inferiori. Es: 35=0000.0011 0000.0101

Moltiplicazione e Divisione

Esistono due tipi, quelle che operano con segno (`imul` e `idiv`), e quelle che operano in modo unsigned (`mul` e `div`).
Prendono un solo operando, che può essere un registro generale o una variabile. Il secondo operando viene scelto dinamicamente in base alla dimensione del primo. Nel caso di moltiplicazione:

- se è di tipo **byte**: 8bit, il secondo è `al` , ed il risultato è salvato in `ax`
- se è di tipo **word**: 16bit, è `ax` ed il risultato è messo in `dx` : `ax` [3]
- se è di tipo **dword**: 32bit

Se viene preso dalla memoria è necessario specificare manualmente la dimensione attraverso le keyword elencate sopra (es: `mul word [0100]`)

In caso di divisione le operazioni di tipo byte utilizzano `ax` come secondo operando, e salvano risultato e resto in `al` ed `ah` rispettivamente. Per operazioni di tipo word, `dx` : `ax` è il secondo operando, resto e risultato sono salvati in `dx` e `ax` .

Esempio di divisione a 16 bit

```
    mov dx, 0234h
    mov ax, 5678h
    mov cx, 1000h
    div cx

    ; dx = 678
    ; ax = 2345
```

Se `dx` fosse maggiore di $1000h$, il risultato della divisione risulterebbe a 20byte e non sarebbe possibile salvarlo in `ax` . Quindi genera un interrupt.

Operazioni su numeri decimali

Esistono istruzioni che lavorano con i numeri salvati in formato packed ed unpacked, ma non prendono parametri, dato che lavorano solamente attraverso i registri AL

- AAA converte il risultato di una somma in decimale unpacked
- AAS converte il risultato di una sottrazione in decimale unpacked
- AAM converte il risultato di una moltiplicazione in decimale unpacked
- AAD converte il dividendo di una divisione da decimale unpacked a binario
- DAA converte il risultato di un addizione in decimale packed
- DAS converte il risultato di una sottrazione in decimale packed.

[3] Indico con `ax` : `bx` , un numero i cui bit più significativi sono salvati in `ax` , ed i bit meno significativi sono salvati in `bx` .

Trasferimento di controllo

Salti

Tutti i salti prendono come unico argomento l'indirizzo di destinazione. L'istruzione per il salto incondizionato (equivalente a goto in C) è `jmp`. Esistono anche i salti condizionati, i quali solitamente sono preceduti da un'istruzione `cmp`.

Instruction	Jump if
`JE`	`zf` $= 1$
`JNE`	`zf` $= 0$
`JA` o `JNBE`	`cf` $= 0$ e `zf` $= 0$
`JAE` o `JNB`	`cf` $= 0$
`JB` o `JNAE`	`cf` $= 1$
`JBE` o `JNA`	`cf` $= 1$ o `zf` $= 1$
`JG` o `JNLE`	`zf` $= 0$ e `sf` $=$ `of`
`JGE` o `JNL`	`sf` $=$ `of`
`JL` o `JNGE`	`sf` $\neq$ `of`
`JLE` o `JNG`	`zf` $= 1$ o `sf` $\neq$ `of`

Flag	
`JC` - `JNC`	Jump if Carry (Carry flag a 1)
`JO` - `JNO`	Jump overflow
`JS` - `JNS`	Jump Sign / Jump Not Sign
`JZ` - `JNZ`	Jump Zero (alias di `JE` e `JNE`)
`JP` o `JPE`	Jump Parity (Even). (bit di parità)
`JNP` o `JPO`	Jump Not Parity, o Jump Parity Odd
`JCXZ`	Jump if `cx` (registro contatore) Zero.

Legenda	
A	Above
B	Below
G	Greater
L	Less
E	Equal
N	Not

Esempio di utilizzo di salti condizionati

```
init:   mov ax, 10
        mov bx, 5

check:  cmp ax, bx
        ja halt          ; jump to halt only if ax > bx

        inc ax
        jmp check

halt:   mov  ax, 4c00h
        int 21h
```

CALL e RET

Una procedura è una label, la cui chiamata corrisponde ad un salto incondizionato, i parametri sono passati via stack. La differenza da un normale salto incondizionato è che al momento di una call, è salvato l'instruction pointer nello stack.

Una procedura, nel caso sia all'interno di uno stesso segmento di codice (inter-segment) è detta di tipo **NEAR**, mentre se può esser chiamata all'interno di un segmento di codice qualsiasi (intra-segment) è detta di tipo **FAR**.

Nel momento in cui effettuo una `call` di tipo NEAR, l'unica cosa che cambia è l'instruction pointer, dato che non cambia il segment. Diversamente se effettuo una `call` FAR, siccome cambia anche il code segment, viene anch'esso pushato all'interno dello stack.

```
start:      call function

halt:       mov ax, 4c00h
            int 21h

function:   mov ax, 10h
            ret
```

> jmp e call hanno la stessa sintassi. Per questo se confuse il compilatore non dà errore. Se una funzione è invocata con jmp l'istruzione ret fa comunque il pop di un valore dallo stack e cambiato l'instruction pointer.

LOOP

L'istruzione `loop etichetta` o `loope etichetta` è equivalente ad effettuare le operazioni:

```
dec  cx
cmp  cx, 0
je  etichetta
```

Esistono anche le varianti: `loopz` e `loopne` che controllano inoltre lo zero flag.
Esempio di utilizzo di `loop`:

```
start:  mov ax, 0h
        mov cx, 10h
cycle:  add ax, 10h     ; Eseguita 10h = 16 volte
        loop cycle
```

INT ed IRET

Gli Interrupt interrompono l'esecuzione normale del programma. Possono essere di tipo hardware o invocati via software (es, tramite istruzione `int`) Il programma, una volta fermato, passa il controllo ad una procedura di tipo FAR, chiamata RRI (*Inserire acronimo*). Al termine dell'esecuzione di questa procedura è eseguita l'istruzione `iret`.

Dato che il programma è interrotto e deve riprendere la sua normale esecuzione, al momento di un'interrupt vengono eseguite in ordine le operazioni di:

- Salvare nello stack il register flag (`pushf`)
- Trap Flag = 0 (disabilita esecuzione step by step per ragioni di sicurezza), e IF = 0 (Interrupt Flag = 0, per evitare l'interruzione di altri interrupt mascherabili).
- Salvare nello stack CS e carica CS della RRI
- Salvare nello stack IP e carica IP della RRI

L'istruzione duale `iret`, recupera le istruzioni di IP, CS e register flag precedentemente salvate nello stack.

Esistono due possibili categorie di interrupt:

- Interrupt BIOS, che dal nome agiscono direttamente a livello di BIOS. Esempi sono la 10h per l'output su video e la 16h per l'input da tastiera.
- Interrupt DOS, che agiscono a livello di sistema operativo. Esempio è 21h, utilizzata sempre per I/O da tastiera e terminazione processo.

Ogni interrupt ha un elenco di funzioni, ed il registro `ah` specifica quale utilizzare.

Code	Function	Description	Info
10h	0Eh	Write chatacter on TTY	AL = Character ASCII code BH = page number (0 current page) BL = foreground color (only gui mode)
16h	00h	Keyboard Read	AL = Read ASCII code AH = scan code (specifies input source)
21h	02h	Character Output	DL = ASCII Code
21h	03h	Keyboard Read and echo	AL = Read ASCII code
21h	4Ch	Terminate Process and EXIT	AL = Exit Code

Contents