ale-cci

Architettura Calcolatori Elettronici

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

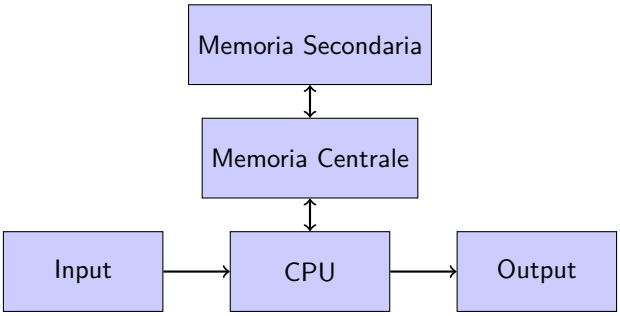
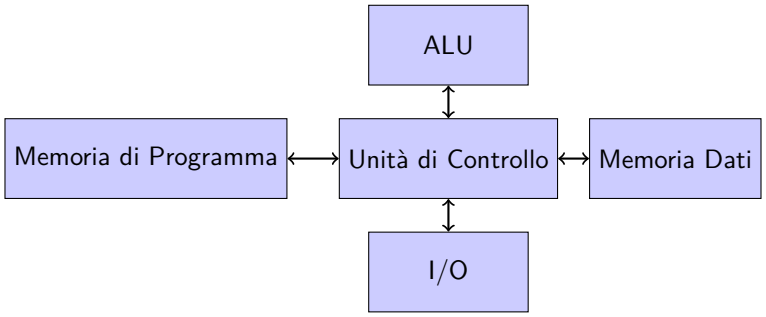


Figure 1: Computer secondo architettura di Von Neumann

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

- Istruzioni di lunghezza fissa
- Decodifica più semplice
- Unità di controllo cablata
- pochi metodi di indirizzamento
- memoria allineata
- molti registri di lunghezza fissa ed ortogonali
- processori load/store

CISC

- Codice operativo e istruzioni di lunghezza variabile
- Decodifica complessa, a più cicli di clock
- Unità di controllo microprogrammata
- Molta flessibilità nei metodi di indirizzamento
- memoria non allineata
- pochi registri di varie lunghezze e non ortogonali

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

test

Lezione 3

CPU: Unità di controllo a stati finiti (fetch, decode, execute)
PC: Program Counter
MR: Memory register
DR: Data register
La memoria si interfaccia con:

- Bus di indirizzi (solo in input)
- Bus di dati,
  utilizzato sia per leggere che per scrivere i dati sulla memoria

Con $R_{2..n}$ vengono indicati i vari registri
Register File: (slide 43)

CPU Monociclo

Modello Harvard: Due memorie separate.
Richiede due memorie separate, siccome accede due volte alla memoria nello stesso ciclo di clock.

1. fetch: accede alla memoria per leggere istruzioni della CPU
2. decode
3. execute
4. memoria

Writeback (WB): Il contenuto letto dalla memoria viene scritto su un registro.
Siccome tutte le operazioni vengono eseguite in un unico ciclo di clock, il tempo impiegato da ciascuna istruzione diventa uguale al tempo impiegato dalla 'funzione' più lenta: $T_{mono} = 83ns$.
Il tempo di esecuzione del programma diventa $N \cdot 83ns$

Lezione 3.6 - Architettura Multiciclo

A differenza della architettura monociclo, in ogni singolo stadio opera in un ciclo di clock. Motivato dal fatto che non tutte le istruzioni impiegano lo stesso tempo di esecuzione.
Per calcolare il tempo impiegato da ogni istruzione $\sum T_{multi}$. A volte $\sum T_{mutli} \geq T_{mono}$.
Non richiede più di separare la memoria, dato che l'esecuzione dell'istruzione è separata in più cicli di clock.

- Fetch
  - Recupero istruzione dalla memoria
  - Aumento PC
    Posso usare l'ALU per il fetch ($Pc = Pc + 5$)
- Decode: Decodifica operandi, Anticipo il calcolo dei registri
  Viene calcolato a prescindere anche il valore di eventuali salti condizionali, sfruttando l'ALU non ancora utilizzata in questo passaggio
- Execute:
  Varia da istruzione ad istruzione,

Lezione 4 - Architetture Avanzate

Benchmark: Strumento per la valutazione delle prestazioni di esecuzione di un calcolatore

Tempo di esecuzione

Valuta il tempo di esecuzione della CPU. In generale il tempo di CPU, può essere misurato tramite tempo di esecuzione, il modello più semplice per farlo è attraverso il *CPI*: Clock per Istruzione (in media),

- $CPI = N_{cc}/N_i$
- N_{cc}: Numero di cicli di clock
- N_i: Numero delle istruzioni

$$\begin{cases} T_{cpu} & = N_{cc} \cdot T_{ck} \\ N_{cc} & = N_i \cdot C_{pi} \end{cases}$$

Obbiettivo RISC:
Ridurre il clock per instruction, rimuovendo istruzioni non necessarie, ottimizzando il numero ridotto di istruzioni dell'architettura.

MIPS: Milioni di istruzioni al secondo $= N_i/CPU_{time} * 11^6$

Pipeline ed Alee

- Alee Strutturali: Due blocchi richiedono la stessa risorsa
- Alee di Dato: Un blocco è in attesa di una risorsa prodotta da un altro blocco
- Alee di Controllo: Quando il processore non ha ancora determinato l'indirizzo di un salto condizionato

Alee di Dato

- **RAW** *Read After Write*: B ha bisogno di un dato prodotto da A, ma B deve leggere il dato prima che A abbia finito di scriverlo.
  Caso più frequente
- **WAR** *Write After Read*: B deve modificare un dato prima che sia letto da A
- **WAW** *Write After Write*: B deve scrivere un dato prima che A lo abbia scritto.

Prima di poter risolvere un *Alea di Dato*, occorre riconoscerla.
Per questo viene prima confrontata la fase di *decode* con quella di *memory*, *execute* e *writeback*.

Oppure occorrono campi specifici sulle pipeline (*latch*), che vanno ad indicare quali registri vengono utilizzati da una determinata istruzione.

5.3.1 Metodi Risolutivi

- **Stallo**:
  Attendo il termine dell'istruzione (molto penalizzante e da evitare)
- **Anticipazione**: Produco il dato immediatamente, introducendo.
  Costosa e non banale da realizzare

- **Sovrapposizione**: Permetto la produzione di 3 risultati in un unico ciclo di clock.
  Sovrappongo le due istruzioni che richiedono la stessa risorsa, e eseguo un istruzione nel fronte di clock di salita, e l'altra nel fronte del clock in discesa.
  Di fatto è come raddoppiare il clock.

- **Riordinamento**:
  Viene cambiato l'ordine di esecuzione delle istruzioni.

Alee di controllo

Avvengono sia in caso di salti condizionati, che in caso di salti
incondizionati.
È risolvibile inserendo uno *stallo* di 1 ciclo di clock, altrimenti anche attraverso il riordinamento, ma non sempre é possibile.

Architettura Supercalcolatori

Parallelismo pipelines
ROB: Buffer di riordinamento

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.