Politecnico di Torino

Architetture e Sistemi di elaborazione

Anno accademico 2022-2023

Marco Lampis

17 dicembre 2022

Indice

0 Informazioni

I seguenti appunti sono stati presi nell'anno accademico 2022-2023 durante il corso di Architetture dei sistemi di elaborazione.

Il materiale non è ufficiale e non è revisionato da alcun docente, motivo per cui non mi assumo responsabilità per eventuali errori o imprecisioni.

Per qualsiasi suggerimento o correzione non esitate a contattarmi.

E' possibile riutilizzare il materiale con le seguenti limitazioni:

- Utilizzo non commerciale
- Citazione dell'autore
- Riferimento all'opera originale

E' per tanto possibile:

- Modificare parzialmente o interamente il contenuto

Questi appunti sono disponbili su GitHub al seguente link:

```
https://github.com/Guray00/polito_lectures/tree/main/Tecnologie%20e%20Servizi%20di%20Rete
```

Repository GitHub

1 Introduzione

Introduzione al corso

1.1 Dependability evaluation

L'affidabilità è spesso misurata utilizzando:

- MTTF, Mean Time To Failor, oppure in FIT, failure in one bilions hours.
- Mean Time Between Failurs, ovvero il tempo che intercorre tra i guasti
- Mean Time To Repair, ovvero il tempo che intercorre tra il guasto e la riparazione

Le tre misure sono legate dalla seguente formula:

$$MTTF = MTBF + MTTR$$

Per riuscire a garantire un rateo di "zero guasti" si studia la "bathtub curve", ovvero la curva che descrive il numero di guasti in funzione del tempo. La curva è caratterizzata da tre fasi:

- Infant mortality: fase iniziale in cui si verifica un numero elevato di guasti
- Normal life: fase in cui il numero di guasti è costante
- End of Life (EOL): fase in cui il numero di guasti aumenta

1.2 Computer Performance

La performance di un dispositivo può essere analizzata da due punti di vista:

1. **Utente**: la risposta nel tempo.
2. **System Manager**: Throughput, la quantità di lavoro che può essere svolta in una unità di tempo.

Il tempo che deve essere considerato per la performance sono:

- elapsed time: tempo che intercorre tra l'inizio e la fine dell'esecuzione di un programma

- cpu time: user cpu time e system cpu time

La valutazione della performance viene spesso effettuata eseguendo le applicazioni e valutando il loro comportamento. La scelta dell'applicativo inficia particolarmente sull'analisi, ma nel caso ideale si dovrebbe utilizzare un carico di lavoro paragonabile all'utilizzo utente. Per questo motivo si utilizzano i benchmark, ovvero del software su misura che simulano il comportamento di un utente.

I benchmark spesso vengono utilizzati eseguendo algoritmi (es quicksort molto grosso), programmi reali (compilatore C) o applicazioni apposite.

In particolare noi utilizzeremo ***MIBench*** che consente di eseguire test inerenti a vari tipi di applicazioni.

E' importante garantire la riproducibilità dei test, per questo motivo è importante utilizzare uno stesso hardware e software per tutti i test (oltre al programma di input).

Può essere interessante avere una media pesata dei risultati, in modo da poter valutare la performance in base al tipo di applicazione.

1.3 Linee guida e principi per il computer design

Le linee guida per la misurazione della performance si basano su due principi:

- legge di Amdahl
- CPU performance equation

1.3.1 Legge di Amdahl

La legge di Amdahl è una formula che descrive il miglioramento della performance in funzione del numero di processori. La formula è la seguente:

$$\text{speedup} = \frac{\text{performance with enhancement}}{\text{performance with without enhancement}}$$

Lo speedup risultante da un miglioramento dipende da due fattori:

- fraction enhanced: la frazione del tempo di computazione che può essere migliorata
- speedup enhanced: la dimensione del miglioramento che le parti ricevono.

$$\text{execution time new} = \text{execution time old} * ((1 - \text{fraction enhanced}) + \frac{\text{fraction enhanced}}{\text{speedup enhanced}})$$

$$\text{speedup overall} = \frac{\text{execution time old}}{\text{execution time new}} = \frac{1}{(1 - \text{fraction enhanced}) + \frac{\text{fraction enhanced}}{\text{speedup enhanced}}}$$

1.3.1.1 Esempio 1

Supponiamo di avere una macchina che è 10 volte più veloce nel 40% dei programmi che girano. Quale è lo speedpup totale?

$$\text{fraction enhanced} = 0.4$$

$$\text{speedup enhanced} = 10$$

$$\text{speedup overall} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56$$

1.3.1.2 Esempio 2

Sono disponibili due soluzioni per migliorare la performane di una macchina floating point:

- *soluzione 1*: aumentando di 10 le performance delle radici quadrate (circa il 20% del tempo di esecuzione) aggiungendo un hardware dedicato.
- *soluzione 2*: aumentare di 2 la performance di tutte le operazioni floating point (circa il 50% del tempo di esecuzione).

Quale soluzione rende più rapida la macchina? Per rispondere è sufficiente riapplicare la legge di Amdahl.

$$\text{speedup1} = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = 1.22$$

$$\text{speedup2} = \frac{1}{(1 - 0.5) + \frac{0.5}{2}} = 1.33$$

La soluzione 2 è più vantaggiosa.

1.3.2 CPU performance equation

Per misurare il tempo richiesto per eseguire un programma sono utilizzabili 3 approcci:

1. osservando il **sistema reale**

2. effettuando delle **simulazioni** (molto costoso)
3. utilizzando la **CPU performance equation**

La terza opzioni consiste nel calcolo della seguente formula:

$$\text{CPU time} = (\sum_{i=1}^{n} CPIi * ICi) * \text{clock cycle time}$$

- **CPI**: clock cycles per instruction
- **IC**: instruction count, ovvero il numero di istruzioni
- **clock cycle time**: è l'inverso della frequenza del clock

2 Instruction Set

L'instruction set Architecture (ISA) è come il computer è visto da un programmatore e dal compilatore. Ci sono molte alternative per un designer ISA, che possono essere valutati in base a vari criteri:

- performance del processore
- complessità del processore
- complessità del compilatore
- dimensione del codice
- consume energetico
- ecc

Le CPU sono spesso classificate in accordo al loro tipo di memoria interna:

- **stack**: unicamente dalla memoria
- **accumulatori**: dalla memoria e da un accumulatore, il secondo risulta sempre la destinazione
- **registri**
  - register-memory (utilizzo di registro e memoria)
  - register-register (unicamente mediante registri)
  - memory-memory

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R1,B`	`Load R2,B`
`Add`	`Store C`	`Store C,R1`	`Add R3,R1,R2`
`Pop C`			`Store C,R3`

Figura 2.1: Esempio di codice

2.1 GPR Machines

Attualmente tutti i processori utilizzano General Purpose Register senza accedere direttamente alla memoria. Non hanno dunque dei ruoli specifici, anche se arm in alcuni casi fa eccezione. Questo è un

favore perché risultano più veloci rispetto alla lettura in memoria ed è più semplice per il compilatore per gestire le variabili.

2.2 Memory Addresses

Esistono due tipi di memorizzazione in accordo all'andianess:

- **big endian**: il byte con l'indirizzo più piccolo viene salvato nella posizione più significativa. L'indirizzo del dato è quello del most significant byte.
- **little endian**: il byte con l'indirizzo più piccolo viene salvato nella posizione meno significativa. L'indirizzo del dato è quello del least significant byte.

Dunque se leggiamo un dato nel modo sbagliato avremo i dati invertiti.

I dati possono essere salvati in modo:

- **allineato**: le letture allineate rappresentano una limitazione per l'accesso in memoria (nel nostro caso sarà sempre allineato)
- **non allineato**: è il caso di intel x86, dove le istruzioni possono avere lunghezza differente, causando overhead sia per le performance sia per l'hardware

La memoria può essere acceduta in tre differenti modi:

- **registro**: `ADD R4, R3`
- **immediato**: `ADD R4, #3`
- **displacement**: `ADD R4, 100(R1)`
- **indiretto**: `ADD R4, (R1)`
- **indexed**: `ADD R3, (R1+R2)`
- **diretto (o assoluto)**: `ADD R1, (1001)`
- **memory inderect**: `ADD R1, @(R3)`
- **autoincrement**: `ADD R1, (R2)+`
- **autodecrement**: `ADD R1, -(R2)`
- **scaled**: `ADD R1, 100(R2)[R3]`

Scegliere una metodologià piuttosto che un'altra può portare a ridurre il numero di istruzioni o aumentare la complessità dell'architettura CPU o aumentare l'average CPI. Quello più diffuso è sicuramente con displacement. La dimensione dell'indirizzo in modalità displacement dovrebbe essere tra 12 e 16 bit mentre la dimensione per la immediate field dovrebbe essere tra 8 e 16 bit.

2.3 Control Flow Instruction

Le istruzioni di controllo possono essere divise in quattro categorie:

- conditional branch
- jumps
- procedure calls
- procedure returns

Gli indirizzi di destinazione sono normalmente specificati come displacement rispetto al valore corrente del program counter. In questo modo:

- riduciamo il numero di bit, in quanto l'istruzione target è spesso molto vicina da quella sorgente
- il codice diviene indipendente dalla posizione

Le chiamate a procedure e i salti indiretti mediante registri consentono di scrivere codice che include salti che non sono conosciuti a tempo di compilazione e di implementare case o switch statements. Supporta le librerie condivise dinamicamente e le funzioni virtuali (chiamare differenti funzioni in base al tipo di dato)

Nel caso di utilizzo di procedure, alcune informazioni devono essere salvate:

- il return address
- i registri utilizzati
  - caller saving
  - callee saving

Riassumendo: Poche istruzioni sono realmente indispensabili come load, store, add subtract, move register-register, and, shift compare equal, compare not equal, branch, jump, call e return. Branch displacements relativo al program counter dovrebbe essere di almeno 8 bit, mentre register-indirect e PC-relative addressing possono essere utilizzati nelle chiamate alle procedure e ritorno.

3 MIPS64

Il MIPS prende il nome da ***M**icroprocessor without **I**nterlocked **P**ipeline **S**tages*, e fa parte della famiglia di processori RISC. Il primo processore è stato inventato nel 1985 a cui si sono susseguiti ulteriori versioni. Quello che si è scoperto è che dimunendo la complessità di ogni passaggio si rendeva più veloce il funzionamento, dunque rimuovendo il sistema di interlock.

Questo tipo di processori quando eseguono un operazione in memoria si limitano a fare "solamente questo". Hanno un simple load-store instruction set. Sono pensati per l'efficienza delle pipeline, in particolare con una lunghezza di istruzioni prefissata e pensate per applicazioni a basso consumo energetico (a differenza dei processori SISC). Il misc per tanto potrebbe risultare più compatto in quanto ogni istruzione fa più operazioni, ma a costo di una maggiore complessità.

I registri sono a 64 bit e il registro 0 è sempre `0` (non `R0`). Questo consente di utilizzare metodi di indirizzamento alternativi rispetto a quelli già visti.

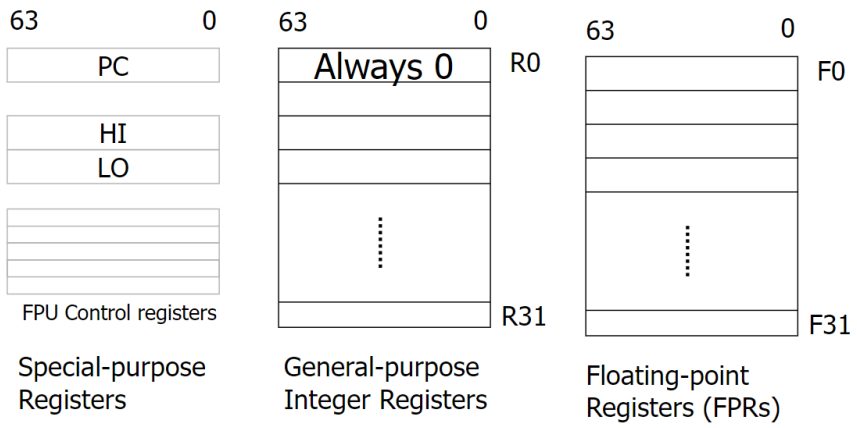

Figura 3.1: Architettura

I tipi di dato utilizzabili sono i classici:

- byte
- half word
- words

- double words
- 32-bit signle precisin floating point
- 64 bit double precision floating point

3.1 Metodi di indirizzamento

3.1.1 Indirizzamento immediato

Viene utilizzato 16 bit di immediate field. Il primo registro è quello destinazione, mentre il secondo e terzo campo sono gli operandi.

3.1.2 Indirizzamento con displacement

LD R1, 30(R2) // carica il valore di R2 + 30 in R1

R2 = XX

R1 <- MEM[30 + R2]

- reistro indiretto
- indirizzamento assoluto

3.2 Formato delle istruzioni

Una istruzione CPU è un single 32 bit aligned word. Include un opcode di 6 bit iniziali. Le istruzioni sono in 3 formati:

- immediato
- registro
- salto (jump)

3.2.1 Immediato

Il primo tipo è quello immediato, caratterizzato da:

- **opcode**: 6 bit di opcode
- **Rs**: 5 bit di indirizzo sorgente *(source register)*
- **Rt**: 5 bit di indirizzo destinazione *(target register)*

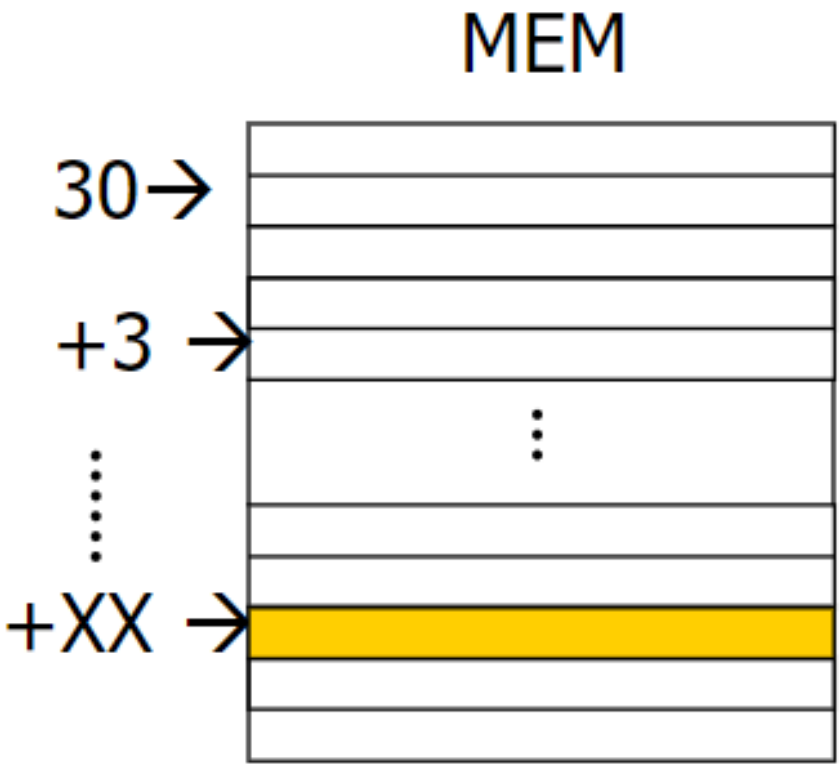

Figura 3.2: Indirizzamento

- **Immediate**: 16 bit signed immediate utilizzati per gli operandi logici, aritmetici, signed operands, load/store address byte offsets, istruzioni di displacemnte rispetto a PC.

3.2.2 Registro

Il secondo tipo è quello registro, caratterizzato da:

- **opcode**: 6 bit di opcode
- **Rd**: 5 bit di indirizzo destinazione
- **Rs**: 5 bit di indirizzo sorgente
- **Rt**: 5 bit di indirizzo di indirizzo target
- **Sa**: 5 bit di shift amount, utile per fare shift a sinistra e a destra
- **Function**: 6 bit di funzione utilizzato per indicare le funzioni

3.2.3 Salto

Il terzo tipo è quello salto, caratterizzato da:

- **opcode**: 6 bit di opcode
- **offset**: Indice a 26 bit spostato a sinistra di due bit per fornire i 28 bit di ordine inferiore dell'indirizzo di destinazione del salto

3.3 Instruction Set

le istruzioni sono raggruppato per il loro funzionamento:

- Load and store
- operazioni ALU
- branches e salti
- floating point
- miscellanee

Nota: le istruzioni sono lunghe 32 bit.

3.3.1 Load and store

I processori MIPS utilizzano un architettura di caricamento e salvataggio, attraverso le quali avviene l'accesso alla memoria principale.

- **LB**: Carica un byte da memoria in un registro. `LB R1, 28(R8)`
- **LD**: Carica una doppia parola da memoria in un registro `LD R1, 28(R8)`
- **LBU**: Carica un byte senza segno da memoria in un registro `LBU R1, 28(R8)`
- **L.S**: Carica un floating point single precision in un registro `L.S F4, 46(R5).`
- **L.D**: Carica un floating point double precision in un registro `L.D F4, 46(R5)`
- **SD**: memorizza un double `SD R1, 28(R8)`
- **SW**: salvo una word `SW R1, 28(R8)`
- **SH**: salvo la half word più significativa `SH R1, 28(R8)`
- **SB**: salvo gli 8 bit meno significativi `SB R1, 28(R8)`
- stessa cosa con i reali

Ovviamente avviene l'estensione dei valori ripetendo il bit più significativo. Nel floating point il primo bit è il segno. Attenzione: per L.S abbiamo il risultato nella parte più significativa.

3.3.2 Operazioni ALU

Tutte le operazioni vengono eseguiti con operandi memorizzati nei registri. Le istruzioni possono essere di tipo immediato, con due operandi, shift, moltiplicazione, divisione, ecc. oltre ad aritmetica in complemento a due come somma, sottrazione, moltiplicazione, divisione.

3.3.2.1 ADD

- **DADDU**: dobule add unsigned `DADDU R1, R2, R3`
- **DADDUI**: double add unsigned immediate `DADDUI R1, R2, 74`
- **LUI**: load upper immediate `LUI R1, 0X47`

3.3.3 Branch e jump

- J Unconditional Jump `J name`
- JAL Jump and Link `JAL name`
- JALR Jump and Link Register `JALR R4`
- JR Jump Register
- BEQZ Branch Equal Zero
- BNE Branch Not Equal
- MOVZ Conditional Move if Zero
- NOP No Operation (nella realtà è uno shift a sinistra di 0 bit di R0 in R0)