Architettura degli Elaboratori

Appunti: Simone Pepi
Stesura: Federico Matteoni

Indice

0.1 Introduzione

Appunti del corso di **Architettura degli Elaboratori** presi a lezione da **Federico Matteoni**.

Prof.: **Maurizio Bonuccelli**, maurizio.angelo.bonuccelli@unipi.it
Riferimenti web:

- *http://pages.di.unipi.it/bonuccelli/aeb.html*
- *didawiki.cli.di.unipi.it/doku.php/informatica/ae/start*

Ricevimento: Martedì 10-12, stanza 294 DE
Esame: **scritto** (*closed book*) e **orale**. I compitini sono validi solo per la sessione invernale (gen-feb)
Libri

- M. Vanneschi *Architettura degli Elaboratori*, Pisa UniversitY Press
- D. A. Patterson *Computer Organization & Design - The Hardware/Software Interface*

0.2 Cosa riguarda il corso

Consiste in come sono fatti pc internamento da un punto di vista di sottosistemi senza scendere nei dettagli elettrici.
Il corso è diviso in quattro parti:

- Fondamenti e strutturazione firmware (I Compitino)
- Macchina assembler (D-RISC) e processi
- Architetture General-Purpose
- Architetture parallele (II Compitino)

Capitolo 1

Fondamenti di strutturazione

1.1 Struttura a livelli

Dividere Per dedicarci allo studio di un sistema complesso spesso è utile **dividerlo in pezzi**. Nel caso di un sistema di elaborazione, in alcuni casi è **interessante avere una visione vicina alla struttura fisica** in termini di componenti hardware. In altri casi è **interessante avere una visione astratta del sistema** per poterne osservare le funzionalità e le strutture più adatte alla specifica applicazione.

Astrarre Da questa necessità deriva la possibilità di strutturare un sistema a vari **livelli di astrazione** che non descrivono una reale struttura fisica, ma è **utile per ragioni specifiche** quali:

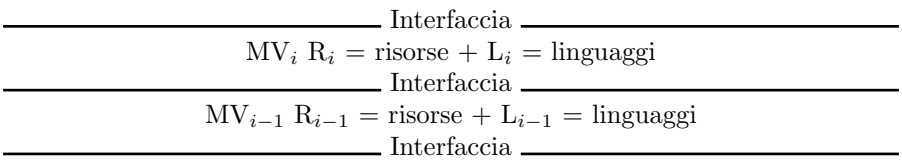
> Saper riconoscere **quale metodo di progettazione strutturata** viene seguito o conviene seguire (**top-down**, **bottom-up**, **middle-out**)
>
> Saper riconoscere **se i vari livelli rispettano una relazione gerarchica** oppure se non esiste alcun tipo di ordinamento
>
> Essere in grado di **valutare a quali livelli conviene descrivere e implementare** determinate funzioni del sistema

1.2 Macchine Virtuali

Sistema di elaborazione **Le funzionalità di un sistema di elaborazione** nel suo complesso possono essere **ripartite su un certo numero di livelli** che vengono definite **macchine virtuali**. La suddivisione può seguire **due approcci** fondamentali:

- **Linguistico**: stabilisce i livelli in base ai linguaggi usati
- **Funzionale**: stabilisce i livelli in base a cosa fanno

I vari livelli sono schematizzati come in figura:



MV$_i$ realizza politica P$_i$ con linguaggio L$_i$ e risorse R$_i$.
MV$_i$ utilizza le funzionalità che il livello MV$_{i-1}$ (cioè le sue primitive) fornisce **attraverso l'interfaccia**.
L'interfaccia definita è fondamentale per poter rendere possibile la collaborazione tra le macchine virtuali, e permettere così ai linguaggi di MV$_i$ di sfruttare funzionalità e meccanismi di MV$_{i-1}$.
Le macchine virtuali godono delle **seguenti proprietà**:

> L'**insieme** degli oggetti o risorse **R$_i$** di MV$_i$ **è accessibile soltanto da parte dei meccanismi di L$_i$**
>
> Al livello MV$_i$ **non sono note le politiche adottate dai livelli inferiori**

Supporto a tempo di esecuzione Anche detto **Runtime Support**, è l'**insieme dei livelli sottostanti**. Nell'esempio, MV_i ha come runtime support i livelli $MV_{i-1} \dots MV_0$.

Virtualizzazione ed Emulazione Con **virtualizzazione** o astrazione intendiamo il **processo secondo cui un livello MV_i usa funzionalità dei livelli superiori**.
Con **emulazione** o concretizzazione intendiamo il **processo secondo cui un livello MV_i usa funzionalità dei livelli inferiori**.

Modularità Tutte queste funzionalità sono **alla base della strutturazione di sistemi con elevata modularià**, **modificabilità**, **portabilità**, **manutebilità** e **testabilità**.

1.2.1 Le Macchine Virtuali

MV_4 Applicazioni
L_4: Java, C, ML...
R_4: oggetti astratti, costrutti, tipi di dato definibili dall'utente

Interfaccia: chiamate di sistema

MV_3 Sistema Operativo
L_3: C, linguaggi di programmazione concorrente, linguaggi sequenziali con librerie che implementano meccanismi di concorrenza
R_3: variabili condivise, risorse condivise, oggetti astratti usati per la cooperazione tra processi e thread

Interfaccia: istruzioni assembler

MV_2 Macchina assembler
L_2: assembler (D-RISC)
R_2: registri, memoria, canali di comunicazione

Interfaccia: istruzioni firmware per l'assembler

MV_1 Firmware
L_1: microlinguaggio
R_1: sommatore, commutatore, registri, strutture di interconnessione intra-unità e inter-unità

Interfaccia: hardware

MV_0 Hardware
L_0: *funzionamento dei circuiti elettronici*
R_0: circuiti elettronici elementari (AND, OR, NOT), collegamenti fisici, reti logiche

Il corso riguarderà principalmente i livelli $MV_2 \rightarrow MV_0$ inclusi, comprese le istruzioni assembler.
Il livello firmware sarà fatto da **memoria**, **processore** e **dispositivi I/O**. I dispositivi di I/O comunicano bilateralmente con la memoria e il processore comunica bilateralmente con memoria. Opzionalmente, i dispositivi di I/O comunicano bilateralmente direttamente con il processore. Questa è l'architettura standard, presentata in maniera estremamente semplicistica.
Vedremo nel dettaglio il processore e la memoria, non i dispositivi di I/O perché troppo complessi.

1.2.2 Compilazione vs Interpretazione

Compilatore: è **statico**, vedendo tutto il codice può ottimizzarlo. Sostanzialmente è l'opera di un traduttore, che può leggersi il testo più volte per tradurlo alla perfezione.

Interprete: è **dinamico**, quindi non può ottimizzare. Il firmware riceve un'istruzione alla volta quindi la interpreta.

Entrambe servono per tradurre il **codice sorgente** nel **programma oggetto** o **eseguibile**.

Suppongo programmi:

```
   for i=0; i++; i<n
A[i] = A[i] + B[i];
```

```
   for i=0; i++; i<n
B[i] = B[i] + C;
```

Ricevendo i due blocchi di istruzioni, il compilatore riconosce che sono diverse e le compila in modo diverso. Però in entrambi i casi sono del tipo *oggetto = somma due oggetti*, quindi produce una sequenza di istruzioni analoga (a meno di registri e dati, ovviamente).

Parte del secondo pezzo di codice, ad esempio, verrà tradotto in questa maniera:

LOAD R_{base}, R_I, R_1	M[R[base] + R[I]] → R[1]
ADD R_1, R_2, R_1	R[1] + R[2] → R[1]
STORE R_{base}, R_I, R_1	R[1] → M[R[base] + R[I]]
INC R_I	R[I] + 1 → R[I]
IF< R_I, R_N, LOOP	Microlinguaggio corrispondente

1.3 Assembler D-RISC

Istruzioni lunghe 32bit, primi 8bit per identificativo istruzione. Poi tre blocchi di 6Bit (R_i, R_j, R_h, in ogni blocco vi è mem semplicemente l'indice i, j o h). Poi 6 bit tipicamente inutilizzati (per estensioni future, istruzioni particolare e per riempire le locaz. di mem che sono tutte a 32 bit).
$2^6 = 64$ registri generali nel processore
Ad esempio ADD R_i, R_j, R_h significa M[R[i] + R[j]] → R[h], e ADD è memorizzato con un determinato codice identificativo.
Per l'inizializzazione, ho il registro R_0 che contiene sempre 0.

Esempio di RTS MV3 C = A + B
Su MV_2 diventa ADD R_A, R_B, R_C
Su MV_1 ho registro A, registro B verso addizionatore/sottrattore (con alfa che indica operazione) e porta in C (con beta che indica scrittura attiva o meno)
Su MV_0 i vari componenti sono costruiti da una serie di gate (AND, OR, NOT).

PO Parte Operativa
PC Parte Controllo
roba eventuale

1.4 Reti Combinatorie

In una rete combinatoria si ha una serie di segnali in input ($X_1 \dots X_n$) che vengono trasformati in una serie di segnali in output ($Y_1 \dots Y_m$). A seconda delle varie componenti presenti sulla rete combinatoria, un insieme di segnali 0/1 viene trasformato in un altro insieme di segnali 0/1 seguendo le regole dell'**algebra booleana**.
Elettricamente, quando un segnale vale 1 significa che la tensione è circa 5V.

1.4.1 Algebra Booleana

L'algebra booleana è computata su **due valori** e **tre operatori**:

false	AND
true	OR
	NOT

Esistono anche altri operatori, derivati dai tre precedenti: XOR, NAND, NOR ecc..

Proprietà Vale la proprietà distributiva anche per la somma rispetto alla moltiplicazione, oltre il viceversa, quindi: A(B+C) = AB + AC, ma anche A + BC = (A + B)(A + C).
Inoltre si hanno le cosiddette **proprietà di DeMorgan**:

- $\overline{A+B}$ = $\overline{A}$ * $\overline{B}$
- $\overline{AB}$ = $\overline{A}$ + $\overline{B}$

AND

Anche detta **moltiplicazione logica**.

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR

Anche detta **somma logica**.

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

NOT

Anche detta **negazione logica**.

Y	Z
0	1
1	0

Per costruire una **rete combinatoria** esistono varie tecniche. Quella che useremo si chiama **somma di prodotti**.

1.4.2 Tecnica della Somma di Prodotti, o codifica degli 1

La tecnica nel dettaglio Partendo dalla **tabella di verità**, identifico le uscite che valgono 1. Di quelle uscite, **moltiplico (`AND`)** tra loro le entrate **sulla stessa riga**, **nego le entrate che valgono 0** e **sommo (`OR`) tra loro le diverse righe**.

Un esempio con la somma algebrica Partendo dalla seguente tabella di verità.

X	Y	Z	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Sfruttando la tecnica descritta sopra ottengo le seguenti espressioni per le due uscite:

`Z =` $\overline{X}$ `* Y + X *` $\overline{Y}$

`R = X * Y`

Alternativamente, posso anche realizzare la **funzione complementare**, ovver fare il solito procedimento ma per le uscite che valgono 0 per poi negarle.

X	Y	$\overline{Z}$	R
0	0	1	0
0	1	0	0
1	0	0	0
1	1	1	1

`Z =` $\overline{\overline{X} * \overline{Y} + X * Y}$

`R = X * Y`

1.5 26-09-2019

S1	S2	X	Y	S1*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

S1* = $\overline{S1}$*$\overline{S2}$*X*Y + $\overline{S1}$*S2*X*Y + S1*$\overline{S2}$*$\overline{X}$*$\overline{Y}$ + S1*$\overline{S2}$*X*$\overline{Y}$ + S1*$\overline{S2}$*X*Y + S1*S2*$\overline{X}$*Y + S1*S2*X*$\overline{Y}$ + S1*S2*X*Y

mappa di carnaut

00 01 11 10, così che tra due colonne cambi un solo bit

prendere multipli di due "uni", cioè 2/4/8... uni
Gli estremi sono logicamente collegati (colonna 00 e colonna 10 sono adiacenti quindi posso formare rettangoli anche tra loro)

1.6 moduli operativi/unità funzionali

Parte operativa: produce l'output
Parte controllo: dice alla PO *come controllare* i suoi componenti (es. produe gli alfa (dicono cosa fare ai componenti) e i beta (quali registri in scrittura e quali no)). La PO porta alla PC le **variabili di condizionamento**, che istruiscono la PC su *come* produrre alfa e beta.
PO e PC sono reti sequenziali.
In generale le mealY sono migliori: mediamente costano non di più e sono non più lente.
in mealY x va anche in omega.

1. PO –var condiz–¿ PC
2. PC –alfa, beta–¿PO
3. PO –output Z–¿ *fuori*

PO la faccio M-
PC la faccio M-
PO moore (ma non sarà automa), PC mealY

Funzionalmente MealY e Moore sono **equivalenti**. Moore rispodne dopo un clock, mealY risponde subito.
var condiz: info che PO passa alla PC affincché P generi alfa e beta.

microlinguaggio uao

1.7 La Memoria