

# Architettura degli Elaboratori

Appunti: Simone Pepi  
Stesura in L<sup>A</sup>T<sub>E</sub>X: Federico Matteoni

# Indice

0.1	Introduzione . . . . .	2
0.2	Cosa riguarda il corso . . . . .	2
<b>1</b>	<b>Fondamenti di strutturazione</b>	<b>3</b>
1.1	Struttura a livelli . . . . .	3
1.2	Macchine Virtuali . . . . .	3
1.2.1	Le Macchine Virtuali . . . . .	4
1.2.2	Struttura Interna . . . . .	5
1.2.3	Parallelismo . . . . .	5
1.2.4	Modelli di Cooperazione . . . . .	6
1.3	Compilazione vs Interpretazione . . . . .	7
<b>2</b>	<b>MV0 – Hardware</b>	<b>8</b>
2.1	Reti Logiche . . . . .	8
2.2	Reti Combinatorie . . . . .	8
2.2.1	Algebra Booleana . . . . .	8
2.2.2	Tecnica della Somma di Prodotti, o codifica degli 1 . . . . .	9
2.2.3	Porte Logiche . . . . .	10
2.2.4	Componenti Standard . . . . .	10
2.2.5	Ritardo di Stabilizzazione . . . . .	11
2.2.6	Registri e memorie . . . . .	11
2.3	Reti Sequenziali . . . . .	12
2.3.1	Modello di Mealy . . . . .	12
2.3.2	Modello di Moore . . . . .	12
2.3.3	Reti Sequenziali di tipo Sincrono . . . . .	13
2.3.4	Reti Sequenziali a Componenti Standard . . . . .	14
<b>3</b>	<b>MV1 – Firmware</b>	<b>16</b>
3.1	Unità Firmware . . . . .	16
3.1.1	PC e PO . . . . .	16
3.1.2	Procedimento Formale . . . . .	18
<b>4</b>	<b><math>\mu</math>-linguaggio</b>	<b>19</b>
4.1	Istruzioni . . . . .	19
4.2	Ottimizzazione del codice . . . . .	20
4.2.1	Condizioni di Bernstein . . . . .	20
4.3	Assembler D-RISC . . . . .	21
4.4	moduli operativi/unità funzionali . . . . .	21
4.5	La Memoria . . . . .	21

## 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](http://didawiki.cli.di.unipi.it/doku.php/informatica/ae/start)

Ricevimento: Martedì 10-12, stanza 294 DE

Esame: **scritto** (*closed book*) e **orale**. I compiti 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 internamente 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 modularità, modificabilità, portabilità, manutibilità 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 Struttura Interna

**Da Verticale a Orizzontale** Fin'ora abbiamo parlato delle macchine virtuali in senso **verticale**, adesso vogliamo trovare un modo concettualmente uniforme – **orizzontale** – per poter studiare i livelli **al loro interno**.

**Sistema di Elaborazione** Una volta scelta la struttura verticale di un sistema, dobbiamo capire come funziona l'interno di ciascun livello per poter far funzionare tutto il sistema, cioè dobbiamo capire **come funziona il sistema di elaborazione di un livello** composto da due componenti:

**Moduli di Elaborazione**, ad ognuno dei quali è affidata l'elaborazione di un sottoinsieme di operazioni del livello.

**Struttura di Interconnessione**, con la quale i moduli di elaborazione del livello **cooperano e comunicano tra loro**.

Essa può essere di due tipi:



**Moduli di Elaborazione** Un modulo di elaborazione è definito come **un'entità autonoma e sequenziale**.

**Autonomia** L'autonomia è **data dal fatto che ogni modulo** di elaborazione **esegue un proprio controllo in maniera indipendente da altri moduli**.

Esso dunque **definisce le proprie strutture dati, operazioni elementari e interfacce** verso altri moduli.

**Sequenzialità** La sequenzialità è **data dal fatto che ogni modulo** di elaborazione **ha un singolo luogo di controllo: la sua attività è descritta da un algoritmo di controllo costituito da una lista sequenziale di comandi**.

La sequenzialità **non implica che un modulo non possa fare uso di forme di elaborazione concorrenti o parallele**. Alcuni o tutti i comandi di una lista sequenziale possono essere costituiti da una o più operazioni elementari eseguite **simultaneamente**.

### 1.2.3 Parallelismo

**Sovrapporre** Poichè i moduli sono autonomi fra loro, sono in grado di **operare indipendentemente l'uno dall'altro**. **Le loro attività possono quindi essere sovrapposte nel tempo** eccetto quando, per ragioni legate alla sincronizzazione, alcuni di loro devono attendere il verificarsi di certi eventi dipendenti dall'elaborazione di altri.

In alcuni **casi limite** seppur realistici, il **funzionamento di tutti i moduli è rigidamente sequenziale**.

Tutto questo **vale per qualsiasi livello** o Macchina Virtuale, quindi sia per firmware che hardware.

### 1.2.4 Modelli di Cooperazione

Dato un sistema di elaborazione a un certo livello, i vari moduli presenti possono cooperare secondo due modalità:

**Ambiente Globale:** esiste un insieme di oggetti comuni accessibili da tutti i moduli che devono cooperare tra loro, e tutti i moduli possono operare su tale insieme

**Ambiente Locale:** i moduli non condividono nulla, quindi non esiste alcun oggetto condiviso tra i moduli. La cooperazione avviene tramite scambio di messaggi.



## 1.3 Compilazione vs Interpretazione

**Programmi** L'obiettivo di un calcolatore è **rendere possibile l'esecuzione di programmi** con una certa qualità di servizio. I programmi vengono **progettati mediante linguaggi di alto livello**, quindi **occorre operare una traduzione da linguaggio di alto livello a linguaggio assembler**.

Tale traduzione può essere effettuata tramite due ben note tecniche e loro combinazioni:

**Compilatore:** è statico.

**Sostituisce l'intera sequenza del programma** sorgente con una sequenza di istruzioni assembler. Questa traduzione viene effettuata staticamente, vale a dire in fase di preparazione e **prima che il programma passi in esecuzione**.

Uno compilatore ha **completa visione del codice** e quindi **può ottimizzarlo**. La sua attività è analoga all'opera di un traduttore, che può leggersi il testo più volte per tradurlo alla perfezione.

**Interprete:** è dinamico

Scandisce la sequenza **sostituendo ogni singolo comando** con una sequenza di istruzioni assembler. La traduzione è effettuata dinamicamente, cioè **a tempo di esecuzione**, quindi non può ottimizzare. Il firmware riceve un'istruzione alla volta, quindi la interpreta.

Il suo svantaggio è che il **tempo di interpretazione viene pagato ogni volta che lancio il programma** e che **non può ottimizzare non avendo una visione globale del programma**.

Entrambe servono per tradurre il **codice sorgente** nel **programma oggetto** o **eseguibile**. L'esecuzione è quindi **più veloce in un programma compilato** rispetto ad un programma interpretato.

$\text{ADD R1, R2, R3} \rightarrow \text{compilatore} \rightarrow \text{OBJ} \rightarrow \text{Interprete Firmware (interfaccia tra MV ASM e MV FW)}$

Intuitivamente, dall'istruzione ad alto livello viene **compilato un programma oggetto OBJ** il quale è un insieme di bit che **viene interpretato dall'interprete firmware**.

**Esempio** Suppongo programmi:

**A**

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

**B**

```
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  $R_{base}$ ,  $R_I$ ,  $R_1$

ADD  $R_1$ ,  $R_2$ ,  $R_1$

STORE  $R_{base}$ ,  $R_I$ ,  $R_1$

INC  $R_I$

IF<  $R_I$ ,  $R_N$ , LOOP

$M[R[base] + R[I]] \rightarrow R[1]$

$R[1] + R[2] \rightarrow R[1]$

$R[1] \rightarrow M[R[base] + R[I]]$

$R[I] + 1 \rightarrow R[I]$

**Microlinguaggio corrispondente**



## Capitolo 2

# MV0 – Hardware

### 2.1 Reti Logiche

L'implementazione a livello hardware di funzioni "pure" dà luogo alle **Reti Combinatorie**.

L'implementazione a livello hardware di funzioni "con stato" dà luogo alle **Reti Sequenziali**.

**Famiglia** Entrambe definiscono la famiglia delle **Reti Logiche** che permettono di realizzare il livello hardware di un sistema di elaborazione.

### 2.2 Reti Combinatorie

Una **rete combinatoria** è una rete logica con  $n$  ingressi binari  $X_1 \dots X_n$  e  $m$  uscite binarie  $Z_1 \dots Z_m$ . Ad ogni combinazione di valori in entrata corrisponde una ed una sola combinazione di valori in uscita. La corrispondenza è definita secondo la funzione implementata dalla rete combinatoria.

Indichiamo  $X_1 \dots X_n$  e  $Z_1 \dots Z_m$  come **variabili logiche** di ingresso ed uscita. **Tutte le combinazioni possibili** delle variabili logiche **sono dette stati** di ingresso – con  $2^n$  possibilità – e di uscita –  $2^m$  possibilità.

Per descrivere le proprietà e la struttura interna delle reti combinatorie si usa un'algebra isomorfa a quella logica, chiama **Algebra Booleana**.

#### 2.2.1 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**.

### 2.2.2 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**, ovvero 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

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

$$R = X * Y$$

#### Esempio

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

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

### 2.2.3 Porte Logiche

Una volta ricavata l'espressione logica dalla tabella di verità, è **immediato realizzare lo schema logico utilizzando le componenti hardware elementari**, dette anche **porte logiche**:



Ogni porta logica AND e OR **comporta un ritardo nel calcolo di  $1 T_p$** . Inoltre, ogni AND e OR può avere **massimo 8 ingressi**, quindi se ho più di 8 segnali in ingresso devo avere *almeno* due livelli: un livello con tante porte logiche quando  $n/8$  con  $n$  numero di segnali in ingresso, e *almeno* un livello in cui "unire" i segnali in uscita in una porta logica analoga.

### 2.2.4 Componenti Standard

Di seguito sono le specifiche di alcune **reti combinatorie** che verranno supposte come **standard**, ovvero come componenti utilizzabili come blocchi elementari nella progettazione di strutture più complesse.

#### Commutatore



#### Selezionatore



### Confrontatore



### ALU



## 2.2.5 Ritardo di Stabilizzazione

**Prestazioni** Per valutare le prestazioni di un sistema, occorre saper **valutare le prestazioni delle reti combinatorie**. Ogni rete reale è **caratterizzata da un ritardo  $T_r$** , necessario affinché **a seguito di una variazione dello stato d'ingresso si produca la corrispondente variazione dello stato in uscita**.

Solo dopo questo tempo si dice che **la rete è stabilizzata**.

$T_p$  Per una porta logica indichiamo con  $T_p$  **il ritardo di stabilizzazione** – ad oggi è di circa  $10^{-2}$  millisecondi. Supponiamo che le **porte NOT** abbiano un **ritardo nullo**, pari a 0  $T_p$ , mentre per le **porte AND/OR** il valore  $T_p$  dipende dal numero di ingressi  $n$  della porta. Per  $n \leq 8$  supponiamo che le porte AND/OR abbiano un **ritardo di stabilizzazione di 1  $T_p$** .

Il costo in  $T_p$  sarà quindi pari ai livelli di AND/OR presenti. Ad esempio, se ho una tabella di verità con  $n$  termini ed  $m$  variabili, avrò  $\log_8 n$  livelli di OR e  $\log_8 m$  livelli di AND. Il costo in  $T_p$  sarà quindi  $= (\log_8 n + \log_8 m) T_p$

## 2.2.6 Registri e memorie

## 2.3 Reti Sequenziali

Una **rete sequenziale** è un oggetto con **un ingresso ed una uscita**, capace di **mantenere uno stato interno** – ecco perché si parla di funzioni con stato. A livello hardware, possiamo identificare una rete sequenziale con un **automa a stati finiti**.

**ASF** Un **automa a stati finiti** è caratterizzato da:

n variabili di ingresso  $\Rightarrow h = 2^n$  **stati di ingresso**  $X_1 \dots X_h$

m variabili di uscita  $\Rightarrow k = 2^m$  **stati di uscita**  $Z_1 \dots Z_k$

r variabili logiche dello stato interno  $\Rightarrow p = 2^r$  **stati interni**  $S_1 \dots S_p$

una **funzione di transizione** dello stato interno  $\sigma: X \times S \rightarrow S$  che **definisce il passaggio tra gli stati**

una **funzione delle uscite**  $\omega: X \times S \rightarrow Z$  che **calcola le uscite**

Una **rete sequenziale** è quindi **composta da due reti combinatorie  $\sigma$  e  $\omega$** , che rispettivamente calcolano la variazione dello stato e l'uscita, **e da un registro  $R$**  che contiene lo stato interno.

### 2.3.1 Modello di Mealy



Considerando il comportamento al tempo  $t$ , lo **stato interno successivo  $S(t+1)$**  dipende sia dallo stato di ingresso al tempo  $t$ , cioè  $X(t)$ , sia dallo stato interno attuale  $S(t)$ .

$$S(t+1) = \sigma(X(t), S(t))$$

Lo **stato di uscita al tempo  $t$ ,  $Z(t)$** , dipende sia dallo stato di ingresso  $X(t)$  sia dallo stato interno attuale  $S(t)$ .

$$Z(t) = \omega(X(t), S(t))$$



### 2.3.2 Modello di Moore



In maniera analoga al modello di Mealy, lo **stato interno successivo  $S(t+1)$**  dipende sia dallo stato di ingresso al tempo  $t$ , cioè  $X(t)$ , sia dallo stato interno attuale  $S(t)$ .

$$S(t+1) = \sigma(X(t), S(t))$$

Lo **stato di uscita al tempo  $t$ ,  $Z(t)$** , dipende solo dallo stato interno attuale  $S(t)$ .

$$Z(t) = \omega(S(t))$$



### 2.3.3 Reti Sequenziali di tipo Sincrono

Vediamo adesso come si comportano nel tempo le reti sequenziali e spieghiamo perché adotteremo quelle di tipo sincrono. Come riferimento usiamo una rete di Mealy.

**Spezzare** Abbiamo detto che lo stato al tempo successivo  $S(t + 1)$  dipende sia dall'ingresso  $X$  sia dallo stato interno attuale  $S(t)$ , cioè  $S(t+1) = \sigma(X(t), S(t))$ .

Il registro  $R$  funge come un "cancello temporizzato" che **spezza la sequenza temporale degli eventi**.

Se il registro  $R$  non fosse presente, si verificherebbe la situazione in figura. In questo esempio, la porta logica o il componente  $\sigma$  **potrebbero non stabilizzarsi mai**.

Se per esempio mettiamo una porta AND con due variabili in ingresso che nega il proprio risultato, tale rete tenderà a non stabilizzarsi mai ma a produrre una sequenza infinita di 0 e 1 in uscita.

Quindi devo avere necessariamente **un meccanismo che mi possa aiutare a determinare il valore dell'uscita** al tempo  $t$ ,  $t + 1 \dots$

Questo strumento è il registro impulsato, dove la **scrittura è scandita dal ciclo di clock**.

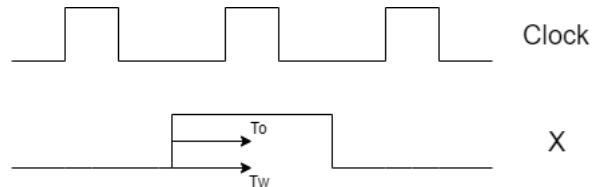


**Modo Sincrono** Questo modo di lavorare delle reti sequenziali con un registro impulsato che funge da cancello temporizzato grazie al ciclo di clock si chiama **Modo Sincrono**.

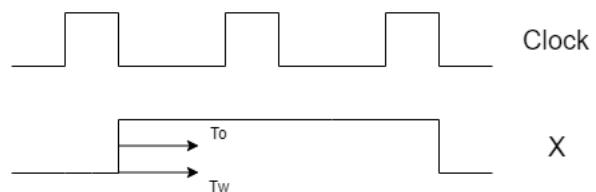
**Quando variare** Cerchiamo ora di capire quando devono variare gli ingressi e **per quanto tempo devono avere tale valore**.

Supponiamo di avere gli ingressi  $X_0 = 0$  al tempo  $t$ ,  $X_1 = 1$  al tempo  $t + 1$  e  $X_2 = 0$  al tempo  $t + 2$ , e supponiamo che  $t_\omega = t_\sigma = 2t$ .

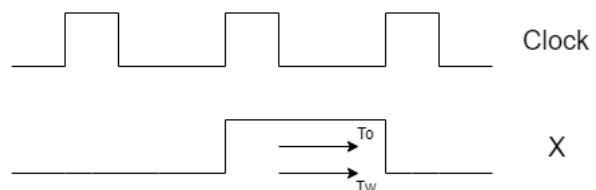
Se l'ingresso  $X$  variesse in un punto non precisato del ciclo di clock è probabile che  $\omega$  e  $\sigma$  non abbiano il **tempo necessario per produrre un risultato** e quindi avrei un **comportamento indefinito**.



In questo caso, cambiando il valore  $X$  all'inizio del ciclo di clock do il tempo necessario a  $\sigma$  e  $\omega$  di produrre un risultato stabile, ma al prossimo impulso del ciclo di clock ( $t + 2$ ) leggerò di nuovo  $X = 1$ , che non è l'input corretto al tempo  $t + 2$ .



Questa è la soluzione giusta per il nostro esempio, che rispetta tutte le condizioni da noi elencate.



Per far funzionare le nostre reti, il ciclo di clock deve essere tale che  $T = \text{MAX}(t_\sigma, t_\omega) + \delta$ . Le reti funzionano anche con  $T > \text{MAX}(t_\sigma, t_\omega) + \delta$ , ma avrei del **tempo perso** poiché la rete non opera, **aspetta solo che il clock sia alto per andare a scrivere nel registro**.

### 2.3.4 Reti Sequenziali a Componenti Standard

Per poter sintetizzare una rete devo prima **decidere se implementare un modello di Mealy o di Moore, derivare le tabelle di verità** di  $\omega$  e  $\sigma$  e dire **quanti bit** ha il registro R.

Fatto questo, **ricavare le reti combinatorie e capire quanto valga il ciclo di clock T** (con  $T = \text{MAX}(t_\sigma, t_\omega) + \delta$ ) che fa funzionare l'intera rete sequenziale.

**Sintesi Classica** ASF  $\longrightarrow$  Mealy o Moore  $\rightarrow$  Tabelle verità, bit di R  $\Rightarrow$  Reti combinatorie  $\Rightarrow$  Ciclo di clock

**Componenti Standard** In realtà per sintetizzare le reti sequenziali non usiamo questo procedimento di sintesi, ma bensì **usiamo i componenti standard**. Per esempio, prendiamo una rete che vuole calcolare il numero di persone presenti dentro una stanza con capienza massima 100 persone.

**Con la sintesi classica** R ha bisogno di 7 bit per contare da 0 a 100.

Se andiamo, per esempio, a fare la tabella di verità per  $\omega$ , abbiamo ben 8 colonne negli ingressi, quindi  $2^8$  possibili combinazioni (righe).

Potrei avere  $2^8/2 = 2^7$  "uni" per colonna, di conseguenza un **numero considerevole di porte logiche**.

Diventa quindi praticamente impossibile sintetizzare questo esempio con il metodo classico. Procediamo con l'alternativa: l'utilizzo delle componenti standard.

**Con le componenti standard** Procediamo col nostro esempio:



In questo caso abbiamo usato il modello di Moore. Il risultato è disponibile al prossimo impulso del ciclo di clock.



Qua invece è stato usato il modello di Mealy. In questo caso si vede bene come **la rete di Mealy sia più veloce**, poiché **il risultato è subito disponibile** prima del prossimo impulso del ciclo di clock: infatti Z non viene scritto in R prima di essere pubblicato.

Di seguito un esempio di rete sequenziale a componenti standard più complesso.



Con sintesi classica Avrei:

$R = \{A, B\}$ , due registri da 32 bit  $\Rightarrow$  64 bit

Ingressi:  $X + Y + \alpha_{K1} + \alpha_{K2} + \alpha_{ALU} + \beta_A + \beta_B = 32 + 32 + 3 + 2 = 69$

Uscite:  $Z \Rightarrow 32$  bit

La tabella di verità di  $\omega$ , per esempio, avrebbe 69 colonne di ingressi, quindi  $2^{69}$  righe, **senza considerare gli ingressi di A e B.**

Il risultato è che è molto scomodo lavorare con una tabella di circa  $5.9 \cdot 10^{20}$  righe.



## Capitolo 3

# MV1 – Firmware

### 3.1 Unità Firmware

Un sistema di elaborazione, a livello Firmware, è costituito da un certo numero di **Unità Firmware** che interagiscono fra loro mediante un sistema di interconnessione. Le UF sono capaci di svolgere un certo numero di operazioni esterne.

**Unità Firmware** Una **unità firmware** è un **modulo di elaborazione autonomo** – cioè capace di controllare la propria operazione in modo del tutto indipendente – e **sequenziale** – cioè dal funzionamento descritto da un programma sequenziale – **capace di eseguire delle operazioni esterne** – istruzioni **assembler**.

**Struttura di interconnessione** Tipicamente la struttura di interconnessione tra unità firmware è **punto-a-punto** quindi a **collegamenti dedicati**.

#### 3.1.1 PC e PO

Per capire bene cosa sono e a cosa servono le parti controllo (PC) e operativa (PO), vediamo un semplice esempio di come arriviamo a strutturare un'unità firmware.



Il nostro obiettivo è quello di realizzare una unità capace di produrre un risultato Z a partire dalle variabili in input, fornendogli solo le istruzioni per l'operazione da implementare e capace di gestire tutte le variabili di controllo ( $\alpha$ ,  $\beta$ ) in maniera autonoma.

L'unità firmware è quindi l'unione di due oggetti:

**Parte Controllo** che "comanda" l'operazione da eseguire

**Parte Operativa** che "esegue" l'operazione

## Ciclo di Clock



PO e PC sono reti sequenziali impulsate dallo stesso segnale di clock, quindi aventi lo stesso ciclo di clock. Il ciclo di clock dell'unità firmware viene determinato in modo da permettere la stabilizzazione di entrambe le reti per l'esecuzione di una qualsiasi microistruzione.

Questo modello di programmazione è **sincrono**.

**Schematizzazione** del diagramma del ciclo di clock sopra:

$\omega_{PO}$ : prepara i valori che servono alla PC per decidere cosa fare

$\omega_{PC}$ : prepara  $\{\alpha, \beta\}$  che implementano l'operazione richiesta

$\sigma_{PC}$ : decido il prossimo stato interno della PC  $\rightarrow$  scrivo il registro R della PC

$\sigma_{PO}$ : eseguo l'operazione pianificata  $\rightarrow$  scrivo i nuovi valori nei registri che compongono lo stato interno della PO

## Parte Operativa, Moore

**Rete Sequenziale** progettata con **componenti standard** che provvede all'esecuzione di **istruzioni** tramite commutatori, selettori, ALU e registri.

## Parte Controllo, Mealy

**Rete Sequenziale** progettata tramite **sintesi classica** che provvede a determinare le **variabili di controllo**  $\alpha$  e  $\beta$  per la parte operativa.

## Mealy o Moore?

Essendo entrambe due reti sequenziali bisogna decidere quale modello usare. Analizziamo le varie combinazioni di modelli.

**Mealy–Mealy** Se uso un modello Mealy–Mealy le uscite di  $\omega_{PO}$  vanno direttamente nella  $\omega_{PC}$  e le uscite di  $\omega_{PC}$  ritornano in  $\omega_{PO}$ . Non ho un registro che ferma il ciclo continuo tra  $\omega_{PC}$  e  $\omega_{PO}$ , quindi non riuscirò mai a stabilizzare i segnali che si scambiano PO e PC.

Viene naturale pensare di farle entrambe Mealy–Mealy poiché, come visto in precedenza, il modello di Mealy è più veloce di quello di Moore.

**Almeno una Moore** Concludiamo che almeno una tra PC e PO deve essere di Moore per poter stabilizzare l'intera UF, ma quale?

La risposta corretta è usare il **modello di Mealy per la Parte Controllo** e il **modello di Moore per la Parte Operativa** in modo da avere dei **comandi veloci** ed una **esecuzione più lenta** rispetto alla PC.

**Il contrario?** Non scegliamo un modello Moore per PC e Mealy per PO perché è **illogico avere un esecutore veloce che deve aspettare un controllore lento**: inutile avere una macchina molto veloce se inserisco i comandi molto lentamente.

Anche un modello Moore–Moore non è comodo da usare, seppure funzionante correttamente, perché avrei entrambe le parti lente.



**Condizione di Correttezza**  $\Rightarrow$  PO di Moore

Le uscite della Parte Operativa, cioè le variabili di condizionamento, dipendono esclusivamente dallo stato interno di PO, cioè **tutte le variabili di condizionamento devono essere prodotte senza usare alcun  $\alpha$**

### 3.1.2 Procedimento Formale

Schematizzazione dei passaggi del procedimento formale per la costruzione e l'analisi di una rete sequenziale.

1. Descrizione a parole delle operazioni esterne
2. Programma scritto in  $\mu$ -linguaggio
3. Componenti

$R_{PO}$ : capire quali sono i registri di stato della PO

$\omega_{PO}, \sigma_{PO}$ : capire le funzioni che mi servono nella PO

$R_{PC}$ : capire cosa è lo stato della PC

$\omega_{PC}, \sigma_{PC}$ : capire cosa calcolare nella PC

$\rightarrow T = t(\omega_{PO}) + \text{MAX}\{t(\omega_{PC}) + t(\sigma_{PO}), t(\omega_{PC})\} + \delta$

# Capitolo 4

## $\mu$ -linguaggio

Si formalizza un linguaggio chiamato  $\mu$ -linguaggio che permetta di **derivare formalmente com'è fatta la PC e la PO di una certa UF**.

### 4.1 Istruzioni

Nel  $\mu$ -linguaggio sono presenti solamente **due tipi di istruzione**:

n.  $\mu\text{op}_1, \dots, \mu\text{op}_k, m$

Le op sono **operazioni di trasferimento tra registri**. Le varie op separate da una virgola sono eseguite **contemporaneamente** – cioè nello stesso ciclo di clock.

m finale indica **a quale istruzione andare dopo aver eseguito questa istruzione**, la n.

n. (condizione = T)  $\mu\text{op}_1, \dots, \mu\text{op}_k, m'$

(condizione = F)  $\mu\text{op}_1, \dots, \mu\text{op}_h, m''$

Le **condizioni** sono **date in termini di variabili di condizionamento**. Possono essere messe in sequenza, venendo **valutate in sequenza**.

Posso considerare solo **variabili booleane** o **espressioni di cui mi interessa solo il risultato** senza memorizzarlo.

**Esempio** Vediamo un esempio di come scrivere un programma in  $\mu$ -linguaggio. Prendiamo come esempio la divisione fra interi.

Linguaggio pseudo-C	$\mu$ -linguaggio
Q = 0	0. 0 -> Q, 1
while (A >= B) {	1. (segno(A - B) = 0) nop, 2
Q = Q + 1	(= 1) nop, 4
A = A - B	2. Q + 1 -> Q, 3
}	3. A - B -> A, 1
R = A	4. A -> R, 0

Ogni  $\mu$ -istruzione è **eseguita esattamente in un ciclo di clock**. Nell'esempio, per eseguire il programma avrò bisogno di almeno 5 cicli di clock, a meno di iterazioni interne.

## 4.2 Ottimizzazione del codice

Dopo aver scritto il  $\mu$ -codice possiamo provare ad **ottimizzarlo**, cioè **ridurre il numero di cicli di clock necessari ad eseguirlo**.

Un'ottimizzazione possibile dell'esempio precedente è la seguente.

Prima	Dopo
0. 0 $\rightarrow$ Q, 1	0. 0 $\rightarrow$ Q, A - B $\rightarrow$ T, A - B $\rightarrow$ A, 1
1. (segno(A - B) = 0) nop, 2 (= 1) nop, 4	1. (T0 = 0) Q + 1 $\rightarrow$ Q, A - B $\rightarrow$ A, A - B $\rightarrow$ T, 1 (= 1) A $\rightarrow$ R, 0
2. Q + 1 $\rightarrow$ Q, 3	
3. A - B $\rightarrow$ A, 1	
4. A $\rightarrow$ R, 0	

Considero un registro T dove memorizzo il risultato di A - B. Di quel registro T, considero T<sub>0</sub> – cioè il bit più significativo – per il segno.

Inoltre elimino le **nop**, che sono *tempo sprecato*.

### 4.2.1 Condizioni di Bernstein

Per eseguire le ottimizzazioni sul  $\mu$ -codice, dobbiamo **seguire le Condizioni di Bernstein**. Tali condizioni forniscono delle regole per verificare se due o più  $\mu$ -operazioni possono essere eseguite nella medesima  $\mu$ -istruzione.

**Le Condizioni** Per capire se

i.  $\mu\text{op}_A, i+1$

i+1.  $\mu\text{op}_B, k$

è equivalente a

i.  $\mu\text{op}_A, \mu\text{op}_B, k$

Bisogna **valutare il dominio R(op)** – registri **letti** da op – e **il codominio W(op)** – registri **scritti** da op – delle  $\mu$ -operazioni.

Nell'esempio precedente:

$$R(A - B \rightarrow T) = \{A, B\}$$

$$W(A - B \rightarrow T) = \{T\}$$

$$R(Q + 1 \rightarrow Q) = \{Q\}$$

$$W(Q + 1 \rightarrow Q) = \{Q\}$$

## 4.3 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  $\text{ADD } R_i, R_j, R_h$  significa  $M[R[i] + R[j]] \rightarrow R[h]$ , e  $\text{ADD}$  è memorizzato con un determinato codice identificativo.

Per l'inizializzazione, ho il registro  $R_0$  che contiene sempre 0.

**Esempio di RTS**  $\text{MV3 } C = A + B$

Su  $MV_2$  diventa  $\text{ADD } R_A, R_B, R_C$

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 ( $\text{AND}, \text{OR}, \text{NOT}$ ).

PO Parte Operativa

PC Parte Controllo

roba eventuale

## 4.4 moduli operativi/unità funzionali

Parte operativa: produce l'output

Parte controllo: dice alla PO *come controllare* i suoi componenti (es. produce 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  $\rightarrow$  var condiz  $\rightarrow$  PC

2. PC  $\rightarrow$  alfa, beta  $\rightarrow$  PO

3. PO  $\rightarrow$  output  $Z \rightarrow$  fuori

PO la faccio M-

PC la faccio M-

PO moore (ma non sarà automa), PC mealY

Funzionalmente MealY e Moore sono **equivalenti**. Moore risponde dopo un clock, mealY risponde subito.

var condiz: info che PO passa alla PC affinché P generi alfa e beta.

microlinguaggio uao

## 4.5 La Memoria