

# Architettura degli Elaboratori

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

# Indice

0.1	Introduzione . . . . .	2	4.5	Memoria Modulare . . . . .	28
0.2	Cosa riguarda il corso . . . . .	2	<b>5</b>	<b>Macchina Assembler</b>	<b>29</b>
<b>1</b>	<b>Fondamenti di strutturazione</b>	<b>3</b>	5.1	CPU . . . . .	29
1.1	Struttura a livelli . . . . .	3	5.2	Istruzioni ASM . . . . .	30
1.2	Macchine Virtuali . . . . .	3	5.3	Programmi e processi . . . . .	30
1.2.1	Le Macchine Virtuali . . . . .	4	5.4	Spazio di Indirizzamento Logico e Memoria Virtuale . . . . .	31
1.2.2	Struttura Interna . . . . .	5	5.4.1	Modalità di Indirizzamento . . . . .	31
1.2.3	Parallelismo . . . . .	5	5.5	RISC vs CISC . . . . .	31
1.2.4	Modelli di Cooperazione . . . . .	6	<b>6</b>	<b>D-RISC</b>	<b>32</b>
1.3	Compilazione vs Interpretazione . . . . .	7	6.1	Istruzioni . . . . .	32
<b>2</b>	<b>MV0 – Hardware</b>	<b>8</b>	6.1.1	Operative . . . . .	32
2.1	Reti Logiche . . . . .	8	6.1.2	Load/Store . . . . .	33
2.2	Reti Combinatorie . . . . .	8	6.1.3	Salto Condizionato . . . . .	33
2.2.1	Algebra Booleana . . . . .	8	6.1.4	Salto Incodizionato . . . . .	33
2.2.2	Tecnica della Somma di Prodotti, o codifica degli 1 . . . . .	9	6.2	Compilazione . . . . .	34
2.2.3	Porte Logiche . . . . .	10	6.3	Processore come UF . . . . .	36
2.2.4	Componenti Standard . . . . .	10	6.3.1	Interfaccia verso la memoria . . . . .	37
2.2.5	Ritardo di Stabilizzazione . . . . .	11	6.3.2	Interfaccia verso UNINT . . . . .	37
2.2.6	Registri e memorie . . . . .	11	6.4	Interprete Firmware . . . . .	38
2.3	Reti Sequenziali . . . . .	12	6.4.1	Valutazione delle prestazioni . . . . .	39
2.3.1	Modello di Mealy . . . . .	12	<b>7</b>	<b>Superamento dei Limiti del Processore Monolitico</b>	<b>40</b>
2.3.2	Modello di Moore . . . . .	12	7.1	Gerarchie di Memoria . . . . .	40
2.3.3	Reti Sequenziali di tipo Sincrono . . . . .	13	7.1.1	Paginazione . . . . .	41
2.3.4	Reti Sequenziali a Componenti Standard . . . . .	14	7.1.2	MMU . . . . .	43
<b>3</b>	<b>MV1 – Firmware</b>	<b>16</b>	7.1.3	Memoria Cache . . . . .	43
3.1	Unità Firmware . . . . .	16	7.2	I/O . . . . .	48
3.1.1	PC e PO . . . . .	16	7.2.1	Trasferimento dati . . . . .	48
3.1.2	Procedimento Formale . . . . .	18	7.3	Trattamento delle Interruzioni . . . . .	49
<b>4</b>	<b><math>\mu</math>-linguaggio</b>	<b>19</b>	7.3.1	DMA I/O . . . . .	49
4.1	Istruzioni . . . . .	19	<b>8</b>	<b>Livello dei processi</b>	<b>50</b>
4.2	Ottimizzazione del codice . . . . .	20	8.1	Supporto a tempo di esecuzione . . . . .	50
4.2.1	Condizioni di Bernstein . . . . .	20	8.2	Schedulazione a basso livello . . . . .	51
4.2.2	Variabili di Condizionamento . . . . .	20	8.3	Istruzione speciale Start-Process (D-RISC . . . . .	52
4.2.3	Tempo medio di elaborazione . . . . .	21	8.4	Commutazione di contesto . . . . .	52
4.2.4	Riflessioni finali sull'ottimizzazione . . . . .	21	8.5	Condivisione indirizzi tra processi . . . . .	53
4.3	Controllo Residuo . . . . .	21	<b>9</b>	<b>Elaborazione in parallelo</b>	<b>54</b>
4.4	Comunicazioni . . . . .	22	9.1	Forme di parallelismo . . . . .	54
4.4.1	Protocollo a Livelli . . . . .	23	9.1.1	Pipeline . . . . .	54
4.4.2	Protocollo a Transizione di Livello . . . . .	24	9.1.2	Farm . . . . .	55
4.4.3	Comunicazioni asincrone a n posizioni . . . . .	26	9.2	Processore . . . . .	57
4.4.4	Comunicazioni asimmetriche . . . . .	26	9.2.1	Processore Pipeline . . . . .	57
			9.2.2	Dipendenze Logiche . . . . .	60
			9.3	Ottimizzazione del codice D-RISC . . . . .	61
			9.3.1	Inlining . . . . .	61
			9.3.2	Out Of Order . . . . .	61
			9.3.3	Artimetiche lunghe con EU Master e Slave . . . . .	62
			9.3.4	Loop Unrolling . . . . .	62
			9.3.5	Delayed Branch . . . . .	63
			9.3.6	Dipendenze Logiche con Data-Flow . . . . .	63
			9.3.7	Utilizzo del registro modificato . . . . .	64
			9.3.8	Rimozione Invarianti . . . . .	65

## 0.1 Introduzione

Appunti del corso di **Architettura degli Elaboratori** a cura di **Federico Matteoni** e **Simone Pepi**.

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

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 modularità, modificabilità, portabilità, manutenibilità 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** Poiché 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

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

$$R = X * Y$$

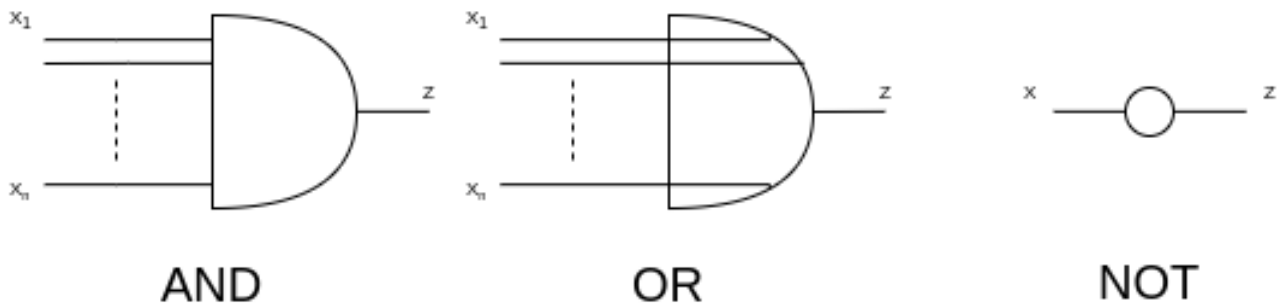
#### Esempio

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.

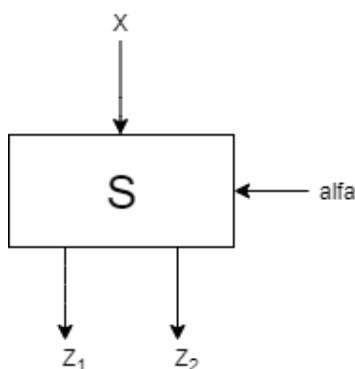
### 2.2.4 Componenti Standard

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.

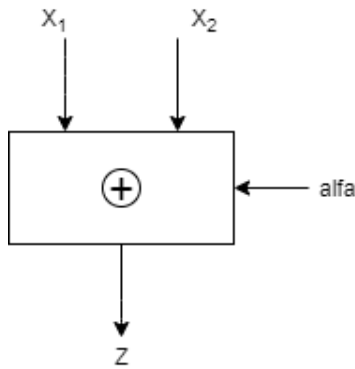
#### Commutatore



#### 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

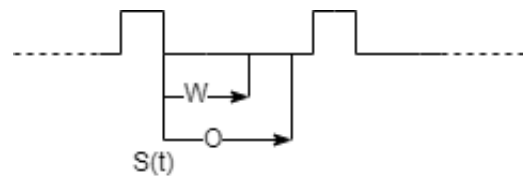


Considerando il comportamento al tempo t, lo **stato interno successivo  $S(t+1)$**  dipende sia dallo stato di ingresso al tempo t, cioè  $X(t)$ , sia dallo stato interno attuale  $S(t)$ .

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

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

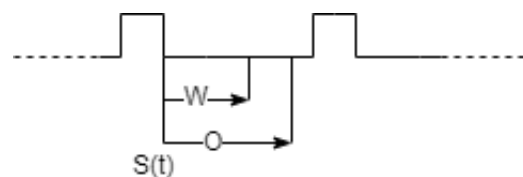


In maniera analoga al modello di Mealy, lo **stato interno successivo  $S(t+1)$**  dipende sia dallo stato di ingresso al tempo t, cioè  $X(t)$ , sia dallo stato interno attuale  $S(t)$ .

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

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**.

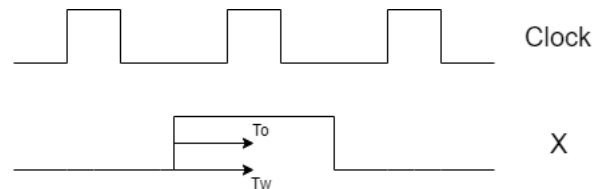


**Modo Sincrono** Questo modo di lavorare delle reti sequenziali con un registro impulsato che funge da cancello temporizzato grazie al ciclo di clock si chiama **Modo Sincrono**.

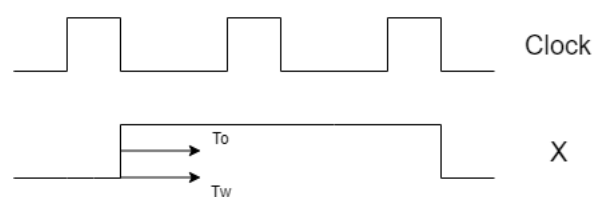
**Quando variare** Cerchiamo ora di capire quando devono variare gli ingressi e **per quanto tempo devono avere tale valore**.

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  $t_\omega = t_\sigma = 2t$ .

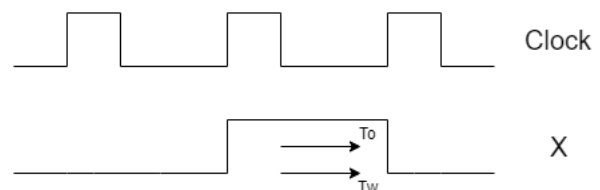
Se l'ingresso  $X$  variesse in un punto non precisato del ciclo di clock è probabile che  $\omega$  e  $\sigma$  non abbiano il **tempo necessario per produrre un risultato** e quindi avrei un **comportamento indefinito**.



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 (**variabili di condizionamento**)

$\omega_{PC}$ : 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

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

# 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.  $\mu op_1, \dots, \mu op_k, m$

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)  $\mu op_1, \dots, \mu op_k, m'$

(condizione = F)  $\mu op_1, \dots, \mu op_h, m''$

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
<pre>Q = 0 while (A &gt;= B) {     Q = Q + 1     A = A - B } R = A</pre>	<pre>0.  0 -&gt; Q, 1 1.  (segno(A - B) = 0) nop, 2     (= 1)                      nop, 4 2.  Q + 1 -&gt; Q, 3 3.  A - B -&gt; A, 1 4.  A -&gt; R, 0</pre>

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
<pre> 0.  0 -&gt; Q, 1 1.  (segno(A - B) = 0) nop, 2     (= 1)                nop, 4 2.  Q + 1 -&gt; Q, 3 3.  A - B -&gt; A, 1 4.  A -&gt; R, 0 </pre>	<pre> 0.  0 -&gt; Q, A - B -&gt; T, A - B -&gt; A, 1 1.  (T0 = 0) Q + 1 -&gt; Q, A - B -&gt; A, A - B -&gt; T     (= 1)      A -&gt; R, 0 </pre>

Considero un registro T dove memorizzo il risultato di A - B. Di quel registro T, considero  $T_0$  – 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.  $\mu op_A, i+1$

$i+1. \mu op_B, k$

è equivalente a

i.  $\mu op_A, \mu op_B, k$

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:

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

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

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

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

Le **condizioni da verificare** sono:

$$W(\mu op_A) \cap R(\mu op_B) = \emptyset$$

**Dipendenza:** non posso mettere insieme  $\mu$ -operazioni tali che la prima scrive in un registro letto dalla seconda.

$$W(\mu op_A) \cap W(\mu op_B) = \emptyset$$

**Dipendenza di output:** non posso scrivere nello stesso registro con due  $\mu$ -operazioni diverse nella stessa  $\mu$ -istruzione.

### 4.2.2 Variabili di Condizionamento

Le **variabili di condizionamento** possono essere così categorizzate:

**Semplici:** indicano le uscite di registri **senza trasformazioni**

$$\longrightarrow t_{\omega PO} = 0$$

**Complesse:** indicano **trasformazioni delle uscite di registri** fatte tramite reti combinatorie prive di ingressi di controllo.

Esempio: **segno(A - B), OR(A)**

$$\longrightarrow t_{\omega PO} = k t_p$$

### 4.2.3 Tempo medio di elaborazione

Il **tempo medio di elaborazione** di una UF viene valutato come:  $T = \sum_{i=0}^{n-1} (p_i * k_i)$

Dove:

$k_i$  è il numero medio di cicli di clock necessari per eseguire una generica operazione  $i$

$p_i$  è la probabilità di eseguire tale operazione

Quando non sono note le  $p_i$ , si assume che tutte le sottosequenze siano equiprobabili. Calcoliamo quindi  $T$  come media aritmetica dei  $k_i$ , oppure si cerca di stimare se possibile una distribuzione probabilistica attendibile.

### 4.2.4 Riflessioni finali sull'ottimizzazione

Bisogna prestare particolare attenzione quando si ottimizza il  $\mu$ -codice. Ridurre il numero di  $\mu$ -istruzioni ( $k_i$ ) non è sempre qualcosa di buono.

Talvolta, **unire due o più  $\mu$ -istruzioni obbliga ad aumentare il ciclo di clock  $T$**  per consentire al  $\mu$ -programma di eseguirle tutte. Questa modifica, che si applica a **tutte** le  $\mu$ -istruzioni, potrebbe aumentare il tempo medio di elaborazione  $T$ , rendendo il programma complessivamente più lento.

Concludendo, le ottimizzazioni che si possono fare sono:

Eliminare le **nop**, tranne quelle di attesa per operazioni esterne

Raggruppare le  $\mu$ -operazioni, attraverso le condizioni di Bernstein

Raggruppare le condizioni logiche

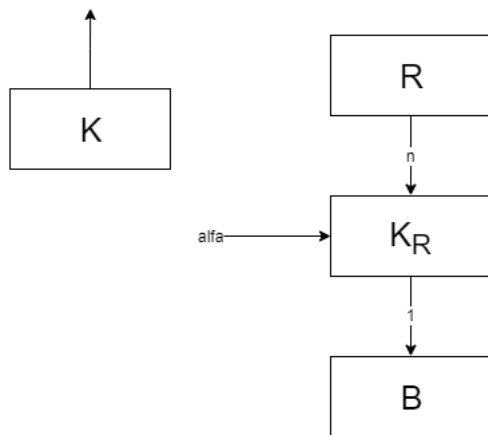
## 4.3 Controllo Residuo

Per diminuire ulteriormente la complessità della PC possiamo **delegare alla PO alcune delle decisioni che dovrebbe prendere la PC**.

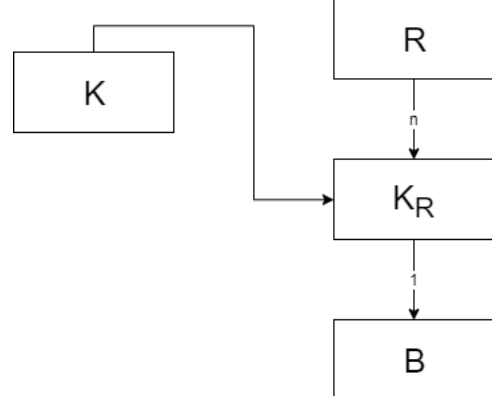
Vediamo alcuni esempi:

Leggere il  $k$ -esimo bit di un registro  $R$  di  $n$  bit

**Soluzione classica**  
Variabili di condizionamento alla PC

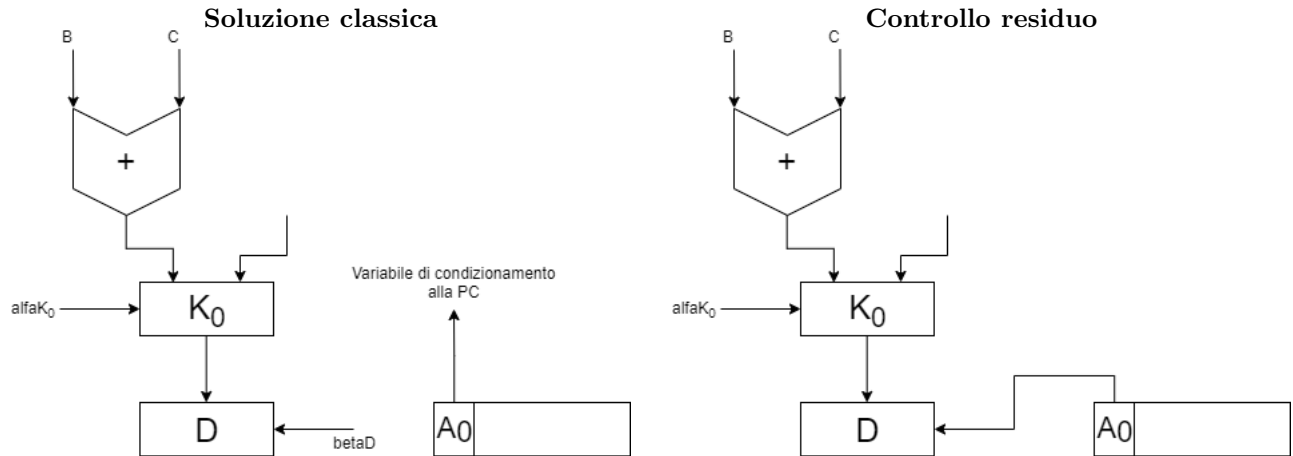


**Controllo residuo**

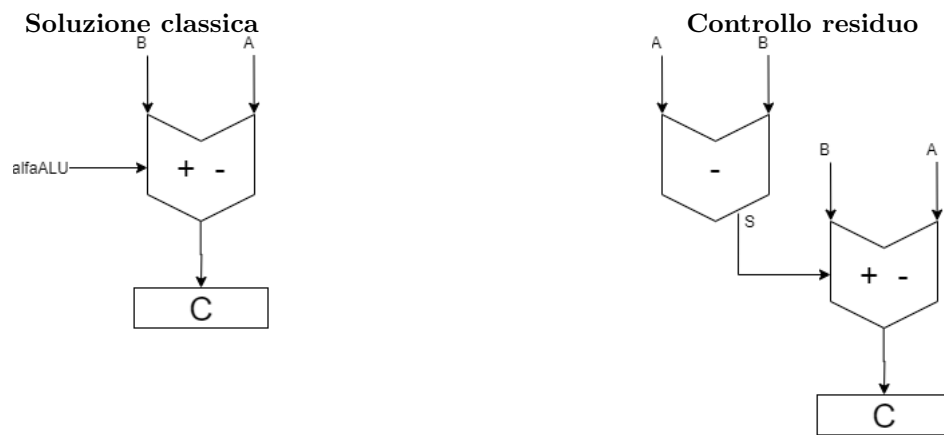


Risparmio complessità della PC e riduco il traffico di dati da PO a PC

Supponiamo una  $\mu$ -istruzione da eseguire a seconda di una certa condizione, ad esempio:  $(A_0 = 0) B + C \rightarrow D$



$(\text{segno}(A - B) = 0) B - A \rightarrow C$   
 $(= 1) B + A \rightarrow C$



## 4.4 Comunicazioni

Con **comunicazioni** si intendono le **comunicazioni fra unità firmware e mondo esterno** e viceversa. Nell'esempio preso in esame, della divisione fra A e B interi con Q ed R risultati, **A e B provengono dal mondo esterno** e **Q ed R sono comunicati verso di esso**.

$$A, B \rightarrow UF \rightarrow Q, R$$

**Categorie** Le comunicazioni sono classificate in due categorie:

### Simmetriche/Asimmetriche

Simmetriche: un solo mittente, un solo destinatario (**uno-a-uno**)

Asimmetriche: asimmetria in ingresso (**più mittenti**) o in uscita (**più destinatari**)

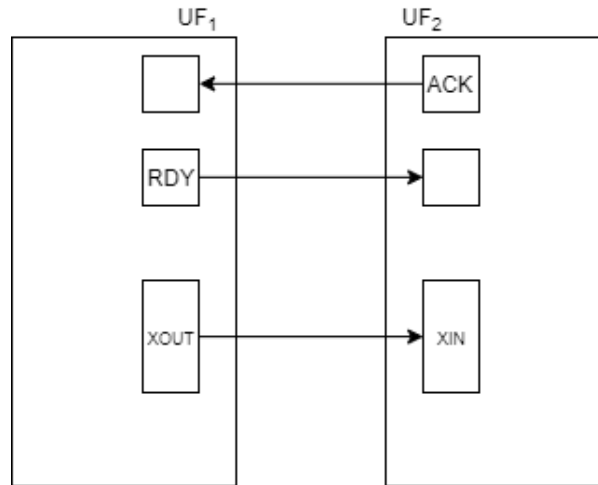
### Sincrone/Asincrone

Sincrone: la comunicazione avviene **"istantaneamente"**

Asincrone: il destinatario legge il messaggio **dopo del tempo** (es. e-mail)

#### 4.4.1 Protocollo a Livelli

**Simmetrico e asincrono**, il protocollo a livelli **funziona aggiungendo ai registri XOUT di UF<sub>1</sub> e XIN di UF<sub>2</sub> altri due registri da 1 bit ciascuno, che indicano quando avviene la comunicazione: ACK e RDY**



Di seguito i passi del funzionamento del protocollo:

1. UF<sub>1</sub> scrive XOUT e il primo registro da 1 bit

Situazione iniziale  
 0      0  
 1   →   0

Situazione finale  
 0      0  
 1   →   1

1 in RDY di UF<sub>2</sub> significa che ci sono dati significativi in XIN

2. UF<sub>2</sub> utilizza XIN e comunica che ha finito scrivendo nel proprio registro di OUT da 1 bit

Situazione iniziale  
 0   ←   1  
 1   →   0

Situazione finale  
 1   ←   1  
 1   →   1

- 3/4. Ritorno alla situazione iniziale con tutti i registri da un bit a 0

1   ←   1  
 0   →   1

1   ←   1  
 0   →   0

1   ←   0  
 0   →   0

0   ←   0  
 0   →   0  
 Si può riniziare

Vediamo i cicli di clock necessari per usare questo protocollo:

1. UF<sub>1</sub> scrive 1
2. UF<sub>2</sub> vede 1 → ... UF<sub>2</sub> agisce... → UF<sub>2</sub> scrive 1
3. UF<sub>1</sub> vede 1 di ritorno → UF<sub>1</sub> scrive 0
4. UF<sub>2</sub> vede 0 → UF<sub>2</sub> scrive 0  
 ⇒ Condizioni iniziali: **4 cicli di clock**

Se le due UF hanno clock sfasati uso lo stesso ragionamento, probabilmente finendo per dover usare più cicli di clock.

**Nel programma** Come rendere questo meccanismo nel  $\mu$ -codice? Proviamo a scriverlo per UF<sub>2</sub>:

```

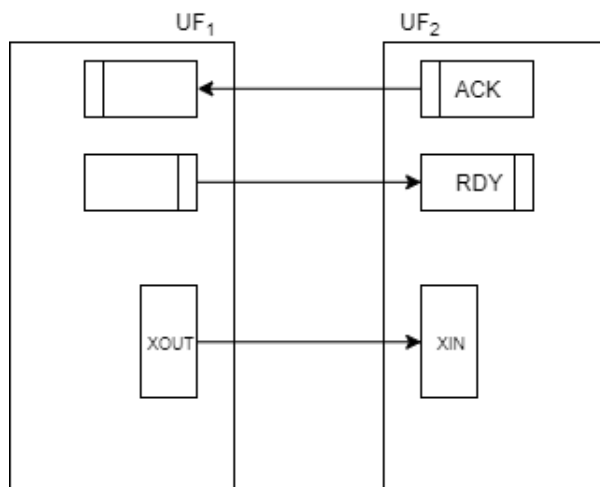
0.  (RDY = 0) nop, 0
    (= 1)      A -> TEMPB, B -> TEMPB, 1 -> ACK, 1
1.  (RDY = 1) nop, 0
    (= 0)      0 -> ACK, <proseguo con altro>
  
```



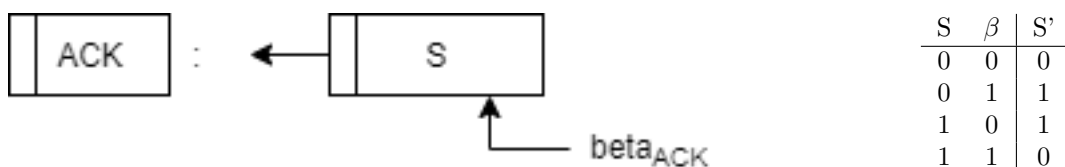
Questo protocollo è particolarmente semplice e necessita **pochissimo hardware**, ma **richiede troppi cicli di clock per comunicare**. Vediamo un'alternativa migliore.

#### 4.4.2 Protocollo a Transizione di Livello

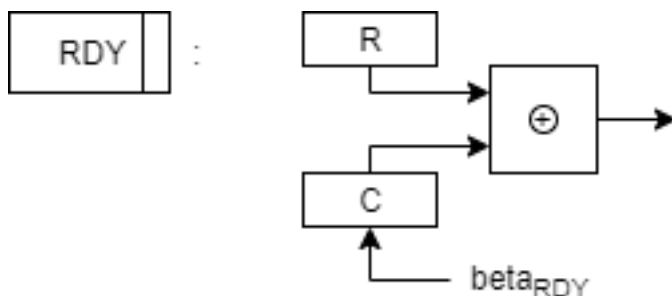
**Simmetrico e asincrono**, simile al protocollo a livelli ma **usa degli indicatori di transizione di livello**.



**ACK** Contatore in modulo 2, quindi cambia stato ( $0 \leftrightarrow 1$ ) ogni volta che ci scrivo.



**RDY** Risultato di un confrontatore fra un contatore modulo 2 e un registro in ingresso.



**Funzionamento** Dal punto di vista di UF<sub>2</sub>.

Quando arriva un segnale da UF<sub>1</sub>, la rete RDY diventa 1, quindi si può lavorare con i dati ricevuti.

Per comunicare a UF<sub>1</sub> che l'operazione è conclusa, mando un  $\beta_{ACK} = 1$  al mio ACK, che diventerà 1 anche in uscita.

Questo procedimento è il medesimo del protocollo a livelli visto in precedenza. Quello che cambia è come mi riporto nelle condizioni iniziali.

Dopo aver lavorato con XIN, chiamo **reset RDY** che **riporta la rete RDY a 0**, quindi pronta ad accogliere un nuovo messaggio.

In poche parole, torno alle impostazioni iniziali nello stesso momento in cui ricevo il messaggio.

## Esempio

Per capire meglio vediamo come esempio quello in esame, la divisione fra A e B interi con risultati Q, R.



```

0.  (RDYIN = 0) nop, 0
    (= 1)      A -> TA, B -> TB, set ACKIN,
              reset RDYIN, 1
1.  0 -> TQ, 2
2.  (segno(TA - TB), ACKOUT = 0-) TQ + 1 -> TQ,
                                  TA - TB -> TA
    (= 11) TA -> R, TQ -> Q, set RDYOUT, reset

```

Possiamo notare che  $UF_1$  manda i segnali A e B e **non ha altre operazioni mentre attende la risposta Q, R da  $UF_2$** . Questa situazione si chiama **protocollo domanda/risposta** e solo in questo caso basta una coppia di indicatori di transizione.

```

UF1
0.  ... -> A, set RDY, 1
1.  (ACK = 0) nop, 1
    (= 1)      B -> ..., reset ACK

```

```

UF2
0.  (RDY = 0) nop, 0
    (= 1) A -> ..., 1
... elaborazione di UF2 ...
n.  ... -> B, set ACK, reset RDY, 0

```

#### 4.4.3 Comunicazioni asincrone a n posizioni

In questo caso **non esiste una soluzione basata su semplici interfacce nelle due unità comunicati**, ma è **necessario introdurre una terza unità** (chiamata **unità buffer**) che **implementi una coda FIFO a n posizioni**. Il mittente può spedire al più n messaggi senza che il destinatario effettui ricezioni.



- 1 Se ci sono messaggi in memoria M, manda un messaggio a  $UF_2$
- 2 Se c'è posto in memoria M, riceve un messaggio da  $UF_1$  e lo memorizza in M

Se valgono entrambe ed M è vuota, allora il messaggio in ingresso da  $UF_1$  viene passato direttamente ad  $UF_2$

$U_{buffer}$  Il codice di  $U_{buffer}$  avrà come **variabili di condizionamento RDY da  $UF_1$ , ACK da  $UF_2$ , condizione di memoria piena e condizione di memoria vuota**.

Le due condizioni di memoria piena/vuota possono essere gestite tramite un semplice contatore. SE per esempio abbiamo una memoria M con  $2^k$  posizioni, useremo un contatore da  $k + 1$  bit:

Memoria vuota  $OR(CONT) = 0$

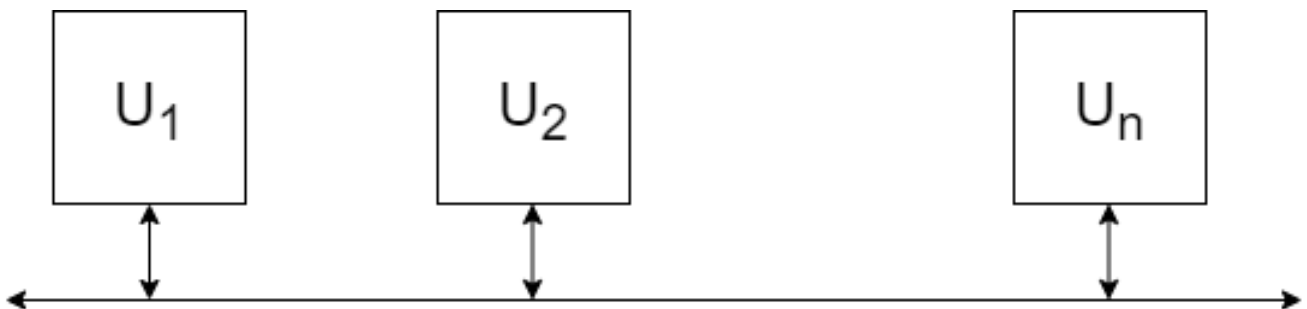
Memoria piena  $CONT_0 = 1$

$\Rightarrow 0. (RDY, ACK, OR(CONT), CONT_0, \dots)$

Il buffer implementa una politica **FIFO**: il **primo messaggio** inviato da  $UF_1$  deve essere il **primo messaggio letto** da  $UF_2$

#### 4.4.4 Comunicazioni asimmetriche

**BUS** Per poter parlare delle comunicazioni asimmetriche è necessario introdurre il concetto di BUS. Un **BUS** è un **insieme di linee per trasportare dati** (es. da 32 bit), **l'indirizzo** (es.  $\log_2 n$  bit) e **una linea da 1 bit che indica l'operazione da svolgere**.



Il grosso svantaggio di questo tipo di comunicazione è che **le unità possono comunicare solamente una alla volta**. Ho bisogno quindi di un **meccanismo di arbitraggio** che regola l'ordine di comunicazione delle varie unità. Vedremo due tipi di arbitraggio: **centralizzato** e **distribuito**.

Inoltre avremo a disposizione anche un protocollo d'interazione Unità — BUS:

richiesta da  $U_i$  di uso del BUS → comunicazione (uso del BUS) → rilascio della risorsa.

Rimane da sottolineare il **problema della sincronizzazione**. Abbiamo bisogno di indicatori a livello per poter comunicare in maniera asimmetrica con un BUS.

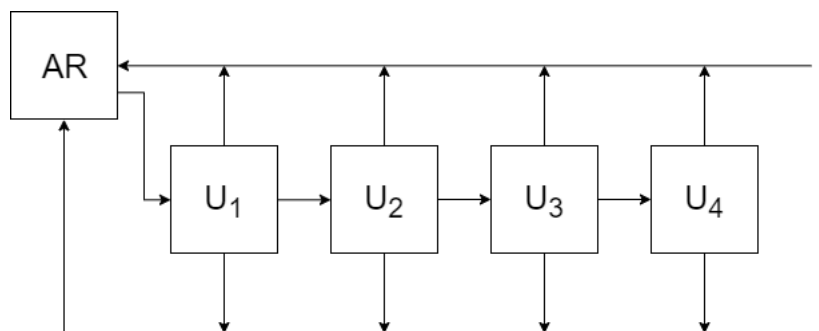
## Arbitri Centralizzati

**Arbitro Centralizzato a Richieste Indipendenti** La unità hanno un **collegamento diretto con l'arbitro**



L'arbitro AR **considera le richieste**, fra di esse **individua l'unità  $U_i$  vincitrice secondo una certa politica P** e **assegna la richiesta** all'unità individuata. Dopodiché, attende il rilascio della risorsa assegnata.

**Arbitro Centralizzato Daisy Chaining** Abbiamo un numero minore di ingressi ma la stessa politica del caso precedente.



Il vantaggio di questo arbitro è la sua semplicità: due ingressi ed una uscita.

L'arbitro, se c'è richiesta, manda il segnale di disponibilità alla prima unità. Il segnale viene passato da unità ad unità, finché non viene trovata l'unità che ne ha fatto richiesta. A tal punto, tale unità manda all'arbitro il bit di occupato, ed inizia ad utilizzare la risorsa. Infine, l'arbitro attende che il bit di occupato torni a 0.

**Arbitro Centralizzato Polling** L'arbitro interroga le varie unità per sapere se hanno bisogno della risorsa BUS, ed assegna le risorse secondo una certa politica.

**Arbitro Centralizzato a Divisione di Tempo** Questa soluzione consiste nell'assegnare l'accesso alla risorsa comune ad ogni unità  $U_i$  per **istanti di tempo ben determinati**.

## Arbitri Decentralizzati

### Arbitro Decentralizzato a Disciplina Circolare (Token Ring)



Se  $U_i$  riceve 1 (token) in ingresso e deve utilizzare la risorsa allora la usa. In ogni caso, sia che debba utilizzarla sia che non debba farlo, passa il token in uscita alla prossima unità.

**Non Deterministici** Usati soprattutto nelle reti wireless.



In questa tipologia di arbitro può succedere che due o più unità rilevino la disponibilità e inizino a trasmettere **contemporaneamente**.

Tale problema viene risolto con **un sistema per rilevare le collisioni**: ogni unità che trasmette ascolta ciò che l'arbitro trasmette. Se ciò che ascolta è il medesimo segnale che ha inviato allora l'invio del messaggio è andato a buon fine, altrimenti se ascolta un messaggio differente allora aspetta un lasso di tempo casuale per poi ritentare la trasmissione.

## 4.5 Memoria Modulare

## Capitolo 5

# Macchina Assembler

In questa parte andremo a vedere alcuni moduli che sono **unità firmware a tutti gli effetti**, con una **certa struttura di interconnessione** e che **riescono ad eseguire istruzioni assembler ASM**. In sostanza, i **processori**.



### 5.1 CPU

**Processore** Una unità firmware in grado di eseguire operazioni esterne che sono istruzioni del linguaggio ASM.

**Cache** Una memoria piccola ma molto veloce, che contiene il sottoinsieme della memoria principale M che permette di eseguire un certo programma.

**MMU** La Memory Management Unit è il componente che permette di tradurre gli indirizzi logici generati dal processore in indirizzi fisici

**Logicamente** Dal punto di vista logico, un processore P fa un ciclo infinito in cui legge l'istruzione all'indirizzo IC (Instruction Counter) o PC (Program Counter), la decodifica e la esegue, poi aggiorna IC e gestisce le interruzioni (eventi generati dal sottosistema I/O)

```
while(true) {
    FETCH    ;legge istruzione a indirizzo IC/PC
    DECODE   ;decodifica istruzione
    EXECUTE  ;esegue istruzione
            ;aggiorna IC/PC
    INT      ;gestisci interruzioni/eccezioni
}
```

## 5.2 Istruzioni ASM

**Dati** Le istruzioni assembler (ASM) operano sostanzialmente su **due tipi di dato**:

**Registri Generali RG:** pochi, velocissimi, lunghi una parola, a doppia porta (permettono di leggere due locazioni e scrivere una locazione nello stesso ciclo di clock)

**Locazioni di memoria:** molte, lente, lunghe una parola, **esterne al processore** che interagisce con M tramite un meccanismo domanda/risposta.

**Istruzioni** I tipi di istruzioni presenti sono:

**Operative:** somma, shift...

**Accesso alla memoria:** lettura ( $M \rightarrow RG$ ) o scrittura ( $RG \rightarrow M$ )

**Salto Condizionale**, con condizione sui RG

**Salto Incondizionale**

## 5.3 Programmi e processi

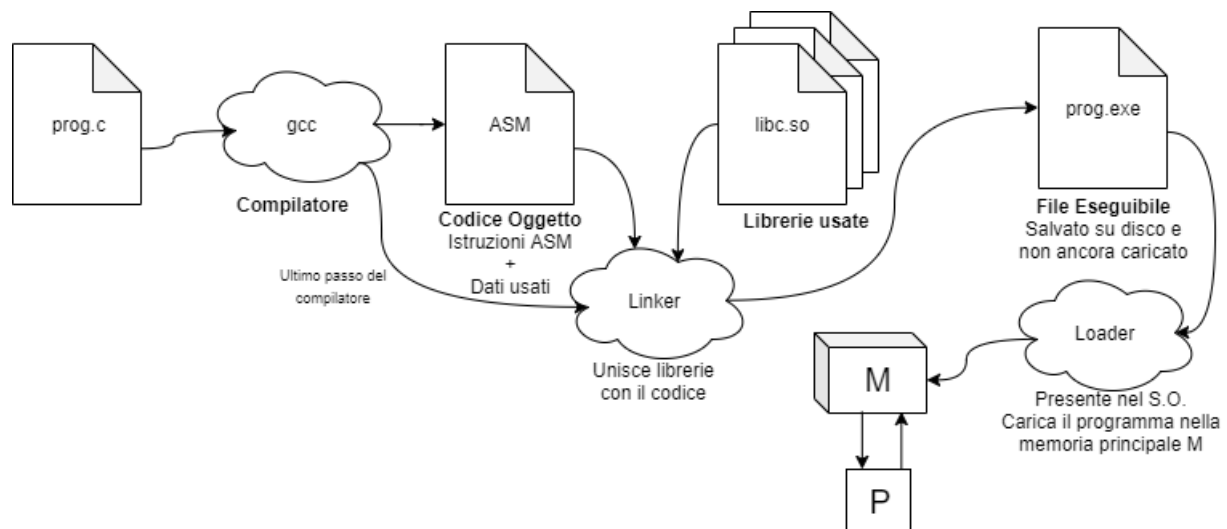
Assumeremo che **ogni programma ASM veda uno Spazio di Indirizzamento detto Memoria Virtuale (MV)**, schematizzato di seguito.

**MV** La **memoria virtuale** la vediamo come un **vettore** le cui **posizioni vanno da un indirizzo 0 ad un indirizzo MAX** (indirizzi logici), ognuno dei quali corrisponde ad **una parola** (32/64 bit).

Lo spazio viene organizzato secondo lo schema a fianco.



**Da programma a processo** Quando un **programma entra in esecuzione diventa un processo**. Di seguito uno schema che racconta le fasi di un programma C che diventa un processo.



## 5.4 Spazio di Indirizzamento Logico e Memoria Virtuale

**Indirizzi** Gli indirizzi generati dal processore, durante l'esecuzione di un processo, **non sono indirizzi della memoria principale** – cioè **indirizzi fisici** – ma **indirizzi logici** cioè riferiti ad un'astrazione della memoria del processo detta **Memoria Virtuale (MV)**.

L'insieme degli indirizzi logici di un processo è detto **Spazio Logico di Indirizzamento**.

Il codice eseguibile del processo/programma, generato dal compilatore, è quindi riferito alla MV, ed il processore genera **indirizzi logici** sia per il codice che per i dati.

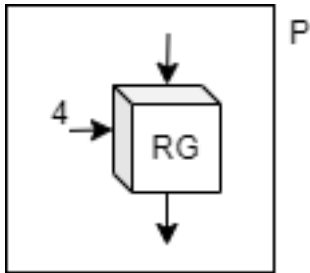
**Rilocazione** Quando un processo viene **creato e caricato**, ad esso viene **allocata una porzione della memoria principale** la cui **ampiezza** e i cui **indirizzi non coincidono con quelli della MV** del processo, ma viene **stabilita una corrispondenza tra indirizzi logici della MV e indirizzi fisici della M**. Questa funzione, detta **funzione di rilocazione o traduzione dell'indirizzo**, è **implementata come una tabella associata al processo della Tabella di Rilocazione**.

Il processo di traduzione deve essere **molto veloce** ed è eseguito dalla **MMU (Memory Management Unit)**.

### 5.4.1 Modalità di Indirizzamento

Vediamo come esprimere la locazione di un certo operando di un'istruzione ASM. Supponiamo di avere l'istruzione **INC X**, che incrementa un valore intero. Vediamo la **X** prima come un **RG** e poi come una **M**.

**Registro**  $RG[4], R4$



**Locazione di memoria** Ci sono diversi modi per esprimerlo:

**Assoluto**

$$R_i \Rightarrow M[RG[i]]$$

**Base + indice**

$$R_i(R_s) \Rightarrow M[RG[i] + RG[s]]$$

**Relativo ad IC/PC**

$$R_i \Rightarrow M[IC + RG[i]]$$

Le modalità di indirizzamento **registro** e **locazione assoluta** sono **molto simili**: la differenza sta in come si usa il registro, nel primo caso direttamente come dato e nel secondo come indirizzo di memoria.

Istruzioni ASM	Operandi
Operative	Registro
Memoria	Base + Indice
Stato	Assoluto/Relativo

## 5.5 RISC vs CISC

**RISC** **R**educed **I**nstruction **S**et **C**omputer

**CISC** **C**omplex **I**nstruction **S**et **C**omputer

**Dilemma** Il progettista dell'architettura di un calcolatore si è sempre trovato di fronte al dilemma "*set di istruzioni semplici*" (RISC) oppure "*set di istruzioni complesse*" (CISC). L'esigenza di avere un set di istruzioni complesso nasce dal desiderio di avere una **corrispondenza il più possibile uno-a-uno tra le istruzioni della macchina ASM e i comandi e le strutture dati dei linguaggi ad alto livello**.

In linea di principio, l'approccio CISC dovrebbe **comportare un aumento di prestazioni rispetto al RISC**, in quanto, a parità di computazione, **ciò che in RISC viene espresso con una sequenza molto lunga di istruzioni ASM, in CISC è espresso con poche istruzioni**.

**Vantaggi e Svantaggi** Molti dei vantaggi potenziali ottenibili con l'approccio CISC si sono rilevati meno ovvi di quanto sembri a prima vista, talvolta addirittura tramutandosi in svantaggi. Infatti **la complessità delle istruzioni e dei modi di indirizzamento può aumentare la lunghezza delle parole dell'istruzione e il numero di accessi in memoria**.

Stabilire il migliore è impossibile, ma a livello didattico conviene studiare l'approccio RISC, in particolare il D-RISC.



# Capitolo 6

## D-RISC

Il **Didactic-Reduced Instruction Set Computer** è un **assembler didattico di tipo RISC**. Le sue caratteristiche sono le seguenti.

**Registri Generali** Sono presenti **64 registri general purpose** ( $RG[0 \dots 63]$ ), cioè in cui posso scrivere qualsiasi cosa, **da 32 bit**.

Il registro  $R_0$  è **particolare** e **contiene sempre 0 al proprio interno**.

**Parole** Le parole all'interno dei registri **sono da 32 bit**.

Questa caratteristica non è vincolante, anche avendo parole da 64 bit la struttura della macchina non cambia.

**Istruzioni D-RISC** In generale il D-RISC contiene **questo set di istruzioni**:

**Operative**: operazioni aritmetico-logiche fra registri

**Load/Store**: caricano dati da/in memoria

**Salto** condizionato/incondizionato

Delle istruzioni si vedranno sintassi e formato in memoria, cioè come vengono rappresentate mediante parole da 32 bit.

**Memoria** La **memoria principale è indirizzabile alla parola**.

### 6.1 Istruzioni

#### 6.1.1 Operative

Comprendono le istruzioni aritmetiche su interi e logiche più comuni. Sono **tutte registro-registro**. Durano 17

**Con Operandi e risultato in RG** ADD op1, op2, op3

Es. ADD  $R_5, R_{27}, R_3$ , semantica:  $R[5] + R[27] \rightarrow R[3]$ .

In memoria:	8 bit	6 bit	6 bit	6 bit	6 bit
	ADD	$R_5$	$R_{27}$	$R_3$	
	CODOP	op1	op2	op3	

**Con uno dei due operandi immediato** ADDI op1, op2, op3

Es. ADDI  $R_5, \#8, R_{27}$ , semantica:  $R[5] + 8 \rightarrow R[27]$

In memoria:	8 bit	6 bit	6 bit	12 bit
	ADD	$R_5$	$R_{27}$	<i>costante in comp. a 2</i>

Questi tipi di operazioni posso farle con ADD, SUB, MUL, DIV, SHR, SHL.

**Istruzioni logiche** AND, OR e NOT

Esempio: AND  $R_1, R_2, R_3$ , semantica  $R[1] \text{ AND } R[2] \rightarrow R[3]$

In memoria:	8 bit	6 bit	6 bit	6 bit	6 bit
	AND	$R_1$	$R_2$	$R_3$	

### 6.1.2 Load/Store

Non sono singole  $\mu$ -istruzioni perchè c'è una comunicazione tra UF (processore-memoria). Sono le **uniche istruzioni memoria-registro del D-RISC**. Come visto in precedenza, l'**indirizzo logico in memoria è calcolato come somma del contenuto di due registri generali**, uno che funge da **base** e uno da **indice**.

Una LOAD dura  $2\tau$ , mentre una STORE dura  $3\tau$ .

**Sintassi**  
LOAD  $R_B, R_i, R_X$   
STORE  $R_B, R_i, R_X$

**Semantica**  
 $M[R[B] + R[i]] \rightarrow R[X]$   
 $R[X] \rightarrow M[R[B] + R[i]]$

**In memoria** (analogo per entrambe):

8 bit	6 bit	6 bit	6 bit	6 bit
LOAD/STORE	$R_B$	$R_i$	$R_X$	

Inoltre, possiamo avere un'**ulteriore istruzione che realizza lo scambio del contenuto tra una locazione di memoria e un RG**.

**Sintassi**  
EXCHANGE  $R_B, R_i, R_X$

**Semantica**  
 $M[R[B] + R[i]] \leftrightarrow R[X]$

**In memoria:**

8 bit	6 bit	6 bit	6 bit	6 bit
EXCHANGE	$R_B$	$R_i$	$R_X$	

### 6.1.3 Salto Condizionato

Permettono di **saltare un numero preciso di istruzioni** a seconda della condizione imposta. Solitamente prende  $2\tau$ .

**Confronto fra due RG** Usando gli operatori logici  $>, <, =, \neq, \leq, \geq$

Sintassi, esempio con  $>$ : IF $_{>}, R_i, R_s, \text{offset}$

Semantica: se  $R[i] > R[s]$  allora  $IC + \text{offset} \rightarrow IC$ , altrimenti  $IC + 1 \rightarrow IC$ .

**In memoria:**

8 bit	6 bit	6 bit	12 bit
IF $_{>}$	$R_i$	$R_s$	offset

**Confronto fra un RG e una costante** Sempre usando gli operatori logici.

Sintassi, esempio con  $>$ : IF $_{>0}, R_i, \text{offset}$

Semantica: se  $R[i] > 0$  allora  $IC + \text{offset} \rightarrow IC$ , altrimenti  $IC + 1 \rightarrow IC$ .

**In memoria:**

8 bit	6 bit	18 bit
IF $_{>0}$	$R_i$	offset

### 6.1.4 Salto Incodizionato

Permettono di **saltare un numero preciso di istruzioni** senza valutare nessuna condizione. Solitamente prende  $1\tau$ .

**Salto con offset** GOTO offset

Semantica:  $IC + \text{offset} \rightarrow IC$

**In memoria:**

8 bit	24 bit
GOTO	offset

**Salto con RG** GOTO  $R_i$

Semantica:  $IC + R[i] \rightarrow IC$

**Non c'è lo stack**, quindi è impossibile la ricorsione.

**Chiamata a procedura** CALL  $R_F, R_{RET}$

$R_F$ : indirizzo della funzione/procedura

$R_{RET}$ : indirizzo di ritorno all'uscita

Semantica:  $R[F] \rightarrow IC, IC + 1 \rightarrow R[RET]$ , operazioni fatte in contemporanea.

Solo per procedure non ricorsive!

## 6.2 Compilazione

Vediamo una serie di esempi su come si compilano alcuni comandi presi dal linguaggio C.

### Assegnamento

```
int x;  
x = 0;  
x = 123;  
x = z;  
  
int y[16];  
y[8] = 0;  
x = y[8];
```

```
Rx  
ADD R0, R0, Rx  
ADDI R0, #123, Rx  
;Possibile, 123 < 2^12  
ADD Rz, R0, Rx  
  
Rby  
;Indirizzo di partenza del vettore  
STORE Rby, #8, R0  
LOAD Rby, #8, Rx
```

### Diramazione Condizionale

```
if (e) {  
    //body  
}  
  
if (x == 0) {  
    y = y + 1;  
}
```

```
IF not(e), cont  
;compilazione body  
cont: ;resto del programma
```

```
IF!=0 Rx, cont  
ADDI Rx, #1, Ry  
cont: ;resto del programma
```

### Diramazione Condizionale con else

```
if (e) {  
    //body then  
} else {  
    //body else  
}  
  
if (x == 0) {  
    y = y + 1;  
} else {  
    y = x - 1;  
}
```

```
IF e, then  
;compilazione body else  
GOTO cont  
then: ;compilazione body then  
cont: ;resto del programma
```

```
IF=0 Rx, then  
SUB Rx, #1, Ry  
GOTO cont  
then: ADD Rx, #1, Ry  
cont: ;resto del programma
```

### Ciclo Indeterminato

```
while (e) {  
    //body  
}  
  
while (x < n) {  
    y = y + x;  
    x = x * 2;  
}
```

```
while:  IF not(e), cont  
        ;compilazione body  
        GOTO while  
cont:   ;resto del programma  
  
while:  IF>= Rx, Rn, cont  
        ADD Ry, Rx, Ry  
        MUL Rx, #2, Rx  
        GOTO while  
cont:   ;resto del programma
```

### Ciclo Determinato

```
for (iniz; cond; incr) {  
    //body  
}  
  
for (i = 0; i < n; i++) {  
    x[i] = 0;  
}
```

```
;assumo di fare almeno un'iterazione  
;compilazione iniz  
for:   ;compilazione body  
        ;compilazione incr  
        IF cond, for  
cont:   ;resto del programma  
  
for:    STORE Rbx, Ri, R0  
        ADD Ri, #1, Ri  
        IF< Ri, Rn, for  
cont:   ;resto del programma
```

Si può ottimizzare con la **tecnica dell'unrolling**: se n è pari, posso fare cicli con i += 2, se è multiplo di 10 con i += 10...

### Prodotto fra vettori

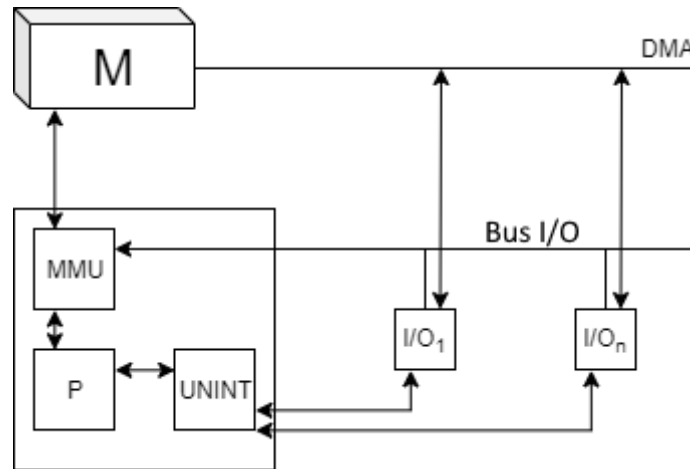
```
sum = 0;  
for (i = 0; i < n; i++) {  
    sum += x[i] * y[i]  
}
```

```
ADD R0, R0, Rsum  
ADD R0, R0, Ri  
for:  LOAD Rbx, Ri, Rxi  
        LOAD Rby, Ri, Ryi  
        MUL Rxi, Ryi, Rmul  
        ADD Rsum, Rmul, Rsum  
        ADD Ri, #1, Ri  
        IF< Ri, Rn, for
```

## 6.3 Processore come UF

Vediamo come **schematizzare un processore (P) come una generica Unità Firmware**.

**DMA** **D**irect **M**emory **A**ccess, BUS che consente alle unità di I/O di **comunicare direttamente con la memoria M**.

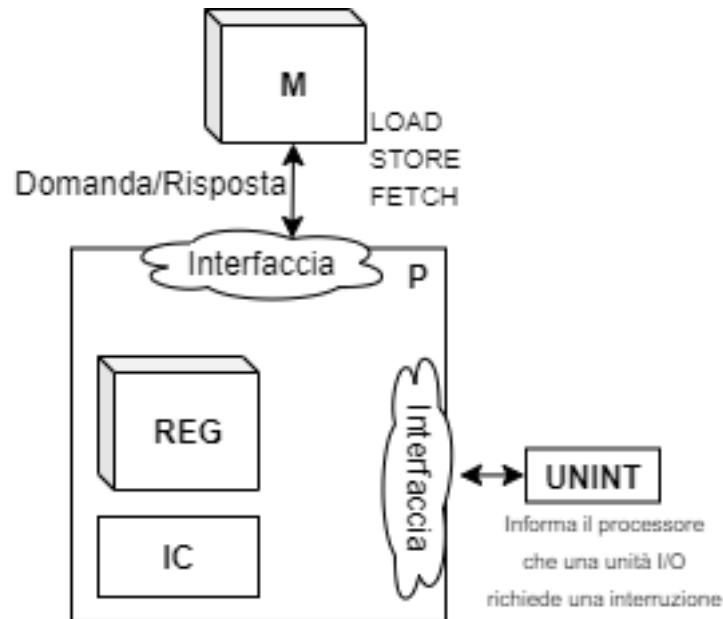


**Processore** Possiamo definire un processore come un'unità firmware che esegue un ciclo infinito in cui si eseguono le operazioni esterne. Le operazioni del processore possono essere descritte, in un linguaggio più "informatico", come segue:

```
while(true) {
    fetch ISTR (IC/PC) //LOAD di una istr, ricordiamo che PC=IC stessa cosa
    decode ISTR
    exec ISTR
    trattamento interruzioni
}
```

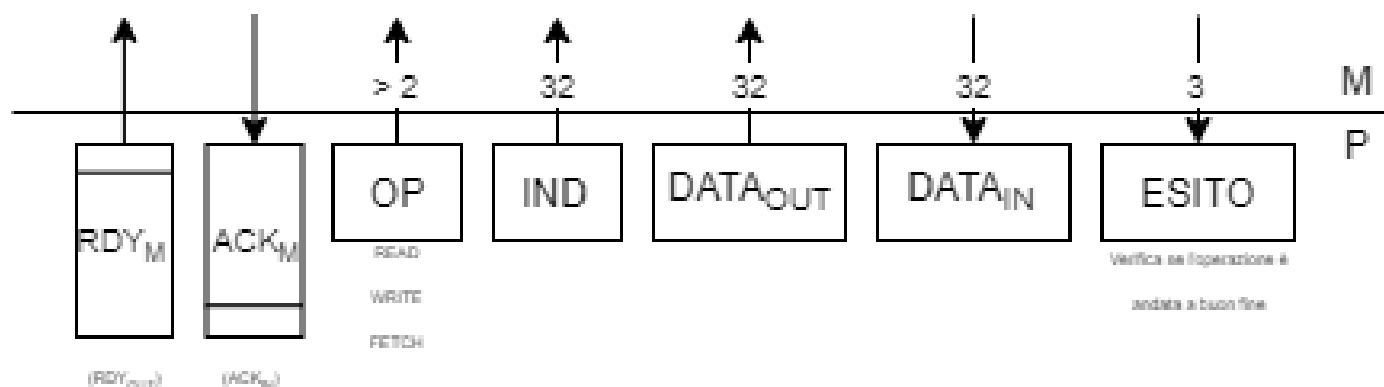
Che diventa l'interprete ASM in  $\mu$ -codice a livello firmware.

Un processore può essere semplicisticamente schematizzato come segue:



### 6.3.1 Interfaccia verso la memoria

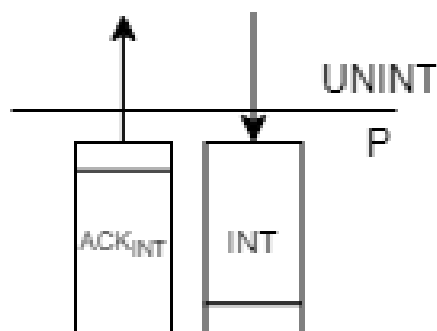
Di seguito lo schema dell'interfaccia a transizione di livello del processore verso la memoria su collegamenti dedicati.



Su altre fonti possiamo trovare due coppie di sincronizzatori (ACK e RDY), ma si assume che l'interfaccia sia a domanda/risposta, quindi possiamo usare una coppia sola.

### 6.3.2 Interfaccia verso UNINT

Di seguito lo schema dell'interfaccia del processore verso l'unità UNINT.



Se durante l'esecuzione del ciclo infinito si riceve  $INT = 1$ , allora il processore deve trattare l'interruzione: manda  $ACK_{INT}$  e fa due **LOAD** per ottenere il numero di unità I/O che ha richiesto l'interruzione e il motivo. Successivamente invoca il driver corretto per il dispositivo.

## 6.4 Interprete Firmware

Scriviamo una prima versione dell'interprete firmware con il  $\mu$ -linguaggio, un estratto:

```
0.      fetch -> OP, IC -> IND, set RDYM, 1

1.      (ACKM, OR(ESITO) = 0-) nop, 1
      (= 10) DATAIN -> IR, reset ACKM, 2
      ;
      ;          ^^ IR = Instruction Register
      (= 11) ..., trattecc
      ;
      ;          ~~~~~ Trattamento eccezioni

2.      (IR.CODOP, INT = "ADD", 0) R[IR.RA] + R[IR.RB] -> R[IR.RC], IC + 1 -> IC, 0
      (= "ADD", 1) R[IR.RA] + R[IR.RB] -> R[IR.RC], IC + 1 -> IC, trattint
      ;
      ;          Trattamento interruzioni ~~~~~
      (= "LOAD", 0) read -> OP, R[IR.RA] + R[IR.RB] -> IND, set RDYM, reset ACKM, 3
      (= "IF<", 0) segno(R[IR.RA] - R[IR.RB]) -> S, 4
      ...

3.      (ACKM, OR(ESITO), INT = 0--) nop, 3
      (= 100) DATAIN -> R[IR.RC], reset ACKM, IC + 1 -> IC, 0
      (= 101) DATAIN -> R[IR.RC], reset ACKM, IC + 1 -> IC, trattint
      (= 110) ..., trattecc

4.      (S, INT = 10) IC + IR.offset -> IC, 0
      (= 11) IC + IR.offset -> IC, trattint
      (= 00) IC + 1 -> IC, 0
      (= 01) IC + 1 -> IC, trattint
      ...
```

**IR** L'Instruction Register è da intendersi come un **registro interno**, cioè non d'interfaccia. In questo registro andiamo a **memorizzare tutti i bit che ci indicano l'istruzione da eseguire**.

Con IR.CODOP o IR.RB si intendono **i soli bit che indicano il codice operativo (operazione), i registri o l'etichetta per i salti**.

**Problema** Il problema di questa prima versione dell'interprete firmware **riguarda la decodifica del Codice Operativo** che ogni volta viene mandato alla PC del processore per gestire i vari  $\alpha$ ,  $\beta$  della PO in modo da eseguire le operazioni corrette.

Essendo il codice operativo un **frammento di 8 bit**, possiamo avere una  $\mu$ -istruzione con ben **256 frasi alternative** che comporta un **esponenziale aumento di complessità di progettazione**.

**Soluzione** La soluzione al problema è quella di **avere una  $\mu$ -istruzione che usa il codice operativo per dire al processore di andare ad eseguire quella istruzione di  $\mu$ -codice che ha come indirizzo il valore del codice operativo**.

Quindi il codice operativo viene mandato alla PC, che **decide quale sarà il suo stato interno** (e quindi la **prossima  $\mu$ -istruzione**) durante il prossimo ciclo di clock. Questa tecnica si chiama **Salto Forzato**, simile per principio al controllo residuo.

A pagina seguente il codice che sfrutta questa tecnica. Notare l'evidente ottimizzazione.

```

fch0.    fetch -> OP, IC -> IND, set RDYM, fch1

fch1.    (ACKM, OR(ESITO) = 0-) nop, fch1
        (= 10) DATAIN -> IR, DATAIN.COP -> RC ;decido la prossima istruzione

add0.    (INT = 0) R[IR.RA] + R[IR.RB] -> R[IR.RC], IC + 1 -> IC, fch0
        (= 1) R[IR.RA] + R[IR.RB] -> R[IR.RC], IC + 1 -> IC, trattint

load0.   read -> OP, R[IR.RA] + R[IR.RB] -> IND, set RDYM, load1

load1.   (ACKM, OR(ESITO), INT = 0--) nop, load1
        (= 100) DATAIN -> R[IR.RC], IC + 1 -> IC, fch0
        (= 101) DATAIN -> R[IR.RC], IC + 1 -> IC, trattint
        (= 110) ..., trattecc

if<0.    segno(R[IR.RA] - R[IR.RB]) -> S, if<1

if<1.    (S, INT = 10) IC + IR.offset -> IC, fch0
        (= 11) IC + IR.offset -> IC, trattint
        (= 00) IC + 1 -> IC, fch0
        (= 01) IC + 1 -> IC, trattint

```

In questa versione abbiamo almeno **una istruzione in  $\mu$ -codice per ogni istruzione ASM**.

Si può notare che la decisione di quale  $\mu$ -istruzione eseguire è **determinata da DATAIN.COP -> RC**, dove andiamo a scrivere gli 8 bit che determinano il codice operativo dell'istruzione ASM direttamente nel registro di stato della PC (RC), in modo da determinare quale  $\mu$ -istruzione eseguire immediatamente dopo.

### 6.4.1 Valutazione delle prestazioni

Il tempo effettivo di accesso in memoria  $t_a$  è maggiore del ciclo di clock del processore: per questo è uno dei principali problemi da risolvere per il raggiungimento di prestazioni elevate.



Con il  $\mu$ -programma visto prima, siamo in grado di dare una valutazione del tempo di elaborazione per le istruzioni che caratterizzano il processore:

$$t_{fch0} + t_{fch1} = 2\tau + t_a \text{ chiamata e decodifica}$$

$$t_{add} = t_{sub} = 1\tau$$

$$t_{saltocond} = 2\tau$$

$$t_{saltoincond} = 1\tau$$

$$t_{load} = 2\tau + t_a$$

$$t_{store} = 3\tau + t_a$$

$$t_{mul} = t_{div} = 50\tau \text{ per convenzione}$$

$t_a = 2(\tau + t_{tr}) + \tau_m = 72\tau$  accesso alla memoria molto lento.

Tempo medio di elaborazione  $T$ , con formula vista in precedenza  $T = \tau * \sum_{i=0}^{n-1} (p_i * k_i)$   
Le probabilità come seguono:

Aritm-Logiche corte 40%

Aritm-Logiche lunghe 10%

Load/Store 30%

Salto 20%

Inoltre le prestazioni variano anche in relazione al tipo di RAM: l'accesso a memorie dinamiche costa 60-100 $\tau$ , mentre a quelle statiche costa 20-40 $\tau$



## Capitolo 7

# Superamento dei Limiti del Processore Monolitico

Nel processore visto fin'ora come un'unica struttura firmware troviamo **due problemi essenziali**: il lungo tempo di accesso alla memoria e il lungo tempo di esecuzione delle istruzioni.

**Tempo di Accesso alla Memoria** Una soluzione al problema consiste nell'introdurre un ulteriore livello di gerarchia di memoria tra la memoria principale ed i registri generali.

Tale memoria, detta **memoria cache**, è **molto piccola**, **molto veloce** ma anche **molto costosa** da realizzare. La cache **conterrà un sottoinsieme delle locazioni della memoria principale**.

**Tempo di Esecuzione delle Istruzioni** Nel processore monolitico le varie operazioni da compiere (fetch, decode, operandi, execute, int) vengono **eseguite sequenzialmente una alla volta**. Possiamo **ottimizzare** questo comportamento in due modi:

**Processori Pipeline o a catena di montaggio.** Si caratterizzano per il **parallelismo nell'interprete firmware**, cioè avere più UF che lavorano in parallelo, ognuna con un particolare compito.

Spenderò tempo inizialmente per **riempire la catena di montaggio** ( $2\tau$  per comunicare tra le UF), ma dopo un certo tempo **produrrà risultati in un tempo sicuramente inferiore** rispetto al processore monolitico.



**Processori Superscalari.** Il concetto alla base è di **avere più processori che lavorano in simultanea** avendo una **memoria di registri in comune**.

### 7.1 Gerarchie di Memoria

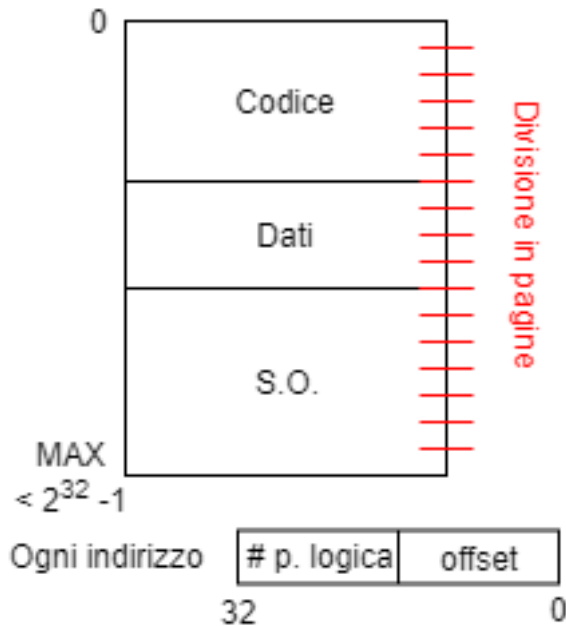
**Idea** Il concetto di gerarchie di memoria è **centrale alla strutturazione dei sistemi** ad un qualunque livello di astrazione.

All'interno di uno stesso elaboratore esistono vari tipi di memoria con caratteristiche molto diverse tra loro, quindi esiste una **gerarchia di memoria** nella quale al **livello più alto** stanno i **dispositivi di memoria più capaci**, più **lenti** e **meno costosi**, mentre man mano che **si scende di livello** i supporti hanno **capacità sempre più piccola**, **tempo di accesso inferiore** e **costo per bit maggiore**.

In termini di ottimizzazione del rapporto prestazioni/costo, l'obiettivo è di raggiungere **prestazioni paragonabili** (appena inferiori) a quelle che avremo se **tutta la massa di informazioni del sistema fosse concentrata nel solo livello inferiore**, e ad un **costo paragonabile** (appena superiore) a quello che avremo se **tutta la massa di informazioni del sistema fosse concentrata nel solo livello più alto**.

### 7.1.1 Paginazione

**Memoria Virtuale – Memoria Principale** Ritorniamo al concetto di memoria virtuale di un processo mandato in esecuzione ed ampliamo i concetti usati.



Ogni **indirizzo logico** è **diviso in due parti**:

pochi bit indicano l'**offset**

i restanti indicano il numero di pagina logica

In questo modo **suddividiamo la MV in pagine tutte della medesima dimensione**, indirizzate dall'offset.

Tale meccanismo è **molto utile per la traduzione** di indirizzi logici in fisici. In particolar modo quando in memoria principale vengono caricati più processi con dimensioni differenti e con tempi di esecuzione differenti. Spesso accade che un processo viene **caricato in memoria in maniera frammentaria** (cioè non è detto che gli indirizzi  $i$  e  $i+1$  corrispondano agli indirizzi fisici  $s$  e  $s+1$ ). Quindi avere un ulteriore metodo per raggruppare gli indirizzi è molto comodo.

**Tabella di rilocalizzazione** Ogni processo ha quindi una **tabella di rilocalizzazione** che **associa ad ogni numero di pagina logica il numero di pagina fisica corrispondente**.

Ogni riga della tabella è una parola di 32 bit, usati per indicare la pagina fisica, ed 1 bit per indicare se è stata caricata o no.

**Tipologie** La paginazione può essere:

**Statica:** carico tutte le pagine del processo in memoria e **ogni posizione rimane tale finché il processo è in esecuzione**

**Dinamica:** carico tutte le pagine del processo in memoria, ma **può capitare che alcune pagine vengano scaricate** – cioè messe in memoria secondaria – **e ricaricate successivamente**. Questo metodo permette di caricare **solo le pagine utili all'esecuzione** del processo, quindi di **ottimizzare l'uso della memoria principale**.

La paginazione dinamica ha due proprietà su cui basa la propria efficacia:

**Località:** se al tempo  $t$  accedo a  $indx$  è molto probabile che al tempo  $t'$  non lontano da  $t$  acceda anche a  $indx+k$  con  $k$  piccolo.

**Riuso:** se al tempo  $t$  accedo a  $indx$  è probabile che in un istante  $t'$  non lontano da  $t$  acceda nuovamente a  $indx$ .

### Working Set

Con **working set** si intende l'**insieme delle pagine che permettono l'esecuzione del programma in un certo istante alla massima velocità minimizzando il numero di fault**, ovvero senza dover portare in memoria principale altre pagine che si trovano in memoria secondaria.

## Dimensione della pagina

Adesso non resta che da capire quanto devono essere grandi le pagine. Partiamo da alcune semplici osservazioni:

**Pagine Piccole**  $\Rightarrow$  Tante pagine

## Dimensione pagina

**cresce**  $\Rightarrow$  diminuisce la probabilità di informazioni che probabilmente accederò fra un po' in memoria

**decresce**  $\Rightarrow$  aumenta la probabilità che fra un po' mi serva un'altra pagina

**Fault di pagina** Il pagefault è il tentativo di accedere ad un indirizzo non presente in memoria principale.



Dopo un certo limite le probabilità di fault tornano a crescere poiché con **pagine troppo grandi può accadere che per caricare nuove pagine vada a scartare pagine in memoria più utili di quella caricata.**

Il grafico è valido per capacità di memoria principale limitate.

Nella **paginazione dinamica**, quando ho un pagefault e quindi ho bisogno di caricare una nuova pagina in memoria principale, se la **memoria è satura** e quindi devo liberare spazio **viene scelta la pagina da scartare secondo regole euristiche LRU** (Least Recently Used).

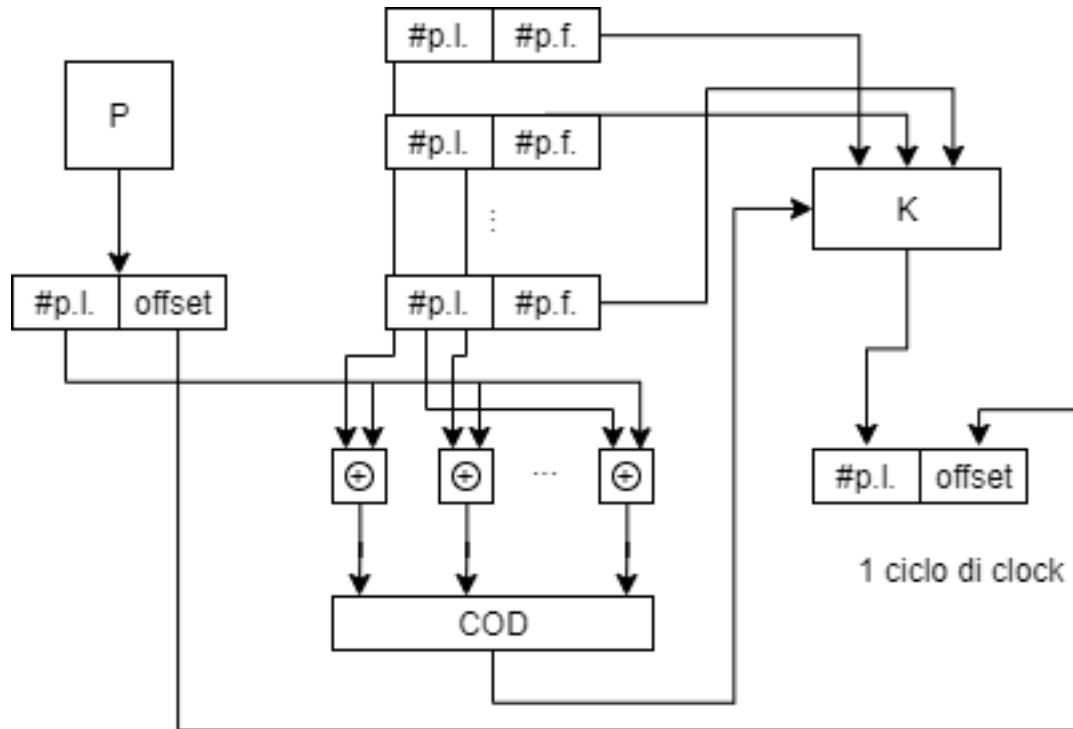
Se si tiene un **contatore sui secondi dopo l'ultimo accesso di una pagina per stabilire la pagina vittima** si ha un **costo molto grande sia di tempo che di spazio** (i confronti).

Un approccio più pratico è quello di **marchiare con un bit le pagine ogni volta che le carichiamo in memoria e azzerare il contatore dopo un certo tempo prestabilito** (pochi istanti). Quando dobbiamo trovare la pagina da scartare non si fa altro che scorrere la memoria e cercare la prima pagina con il contatore da un bit a 0.

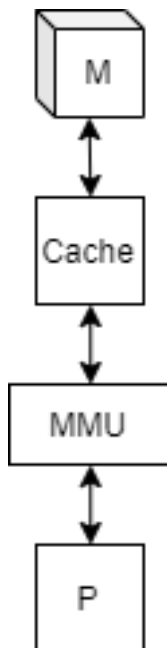
### 7.1.2 MMU

La **traduzione dell'indirizzo** da logico a fisico e il **controllo della protezione** – e l'eventuale generazione dei pagefault – sono implementate ad hardware–firmware nella **Memory Management Unit**, facente parte della CPU. **Al processore P non è visibile come avviene la traduzione** degli indirizzi, ma deve ovviamente aver noto l'esito di ogni accesso in memoria. Come già visto, tale **esito viene esplicitamente inviato a P dalla MMU per ogni richiesta** di accesso.

Non è realistico pensare che, viste le dimensioni, l'intera tabella di rilocazione del processo venga copiata all'interno della MMU. Occorre realizzare in hardware una **tabella accessibile per contenuto**: questo componente è chiamato **memoria associativa**



### 7.1.3 Memoria Cache



Per rendere i calcolatori più veloci, quindi rendere l'accesso in memoria principale più rapido, usiamo delle **memorie cache** che sono **molto più piccole della memoria principale ma anche molto più veloci**.

L'idea è quella di **memorizzare nella cache C solo il working set** di un programma per rendere la traduzione degli indirizzi molto veloce.

$$\dim(M) \gg \dim(C)$$

$$t_{aM} \gg t_{aC}$$

$$M = O(\text{Gb}), t_a = O(100\tau)$$

$$C = O(\text{Kb})/O(\text{Mb})$$

**Livelli di cache**

$$C_2 = 2\text{--}16 \text{ Mb}, t_a = O(10\tau)$$

$$C_1 = 16\text{--}32 \text{ Kb}, t_a = O(2\text{--}3\tau)$$

Nella cache memorizziamo parole da 32 bit della memoria principale facenti parte del working set del processo in uso.



Quando il processo mi chiede di usare una parola non ancora caricata in cache, non solo viene memorizzata in cache tale parola ma anche le parole contigue che **probabilmente verranno usate negli istanti successivi** (principio di località).

Tale tecnica funziona poiché la memoria principale è una memoria interlacciata e mi consente in un solo ciclo di accedere a più indirizzi.

### Indirizzamento diretto

In questo caso **ogni blocco di memoria principale può essere trasferito solo in un determinato blocco della cache**: esiste **uno ed un solo blocco** della cache in cui una certa informazioni **può risiedere**.



### Procedura logica per leggere nella cache la parola di IND

Vado nella linea di indirizzo # linee

Controllo che  $IND.TAG = C[\# linee]$

Se è vero → uso  $IND.offset$  per prendere la parola cercata

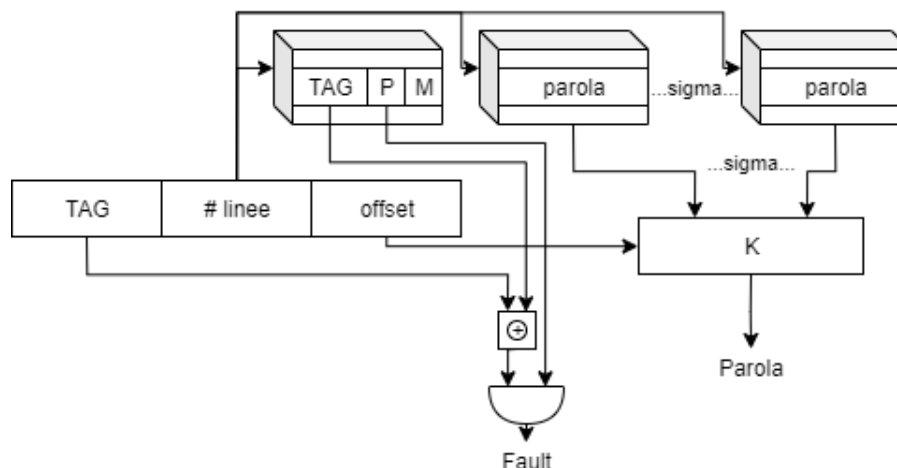
Se è falso → **fault di cache**



**Bit di modifica** In ogni **linea di cache** è presente anche un **bit di modifica** che indica se una delle  $\sigma$  parole è **stata modificata**, così da sapere che devo salvare in memoria la linea aggiornata prima di cancellarla. Vediamo com'è implementata a livello firmware.

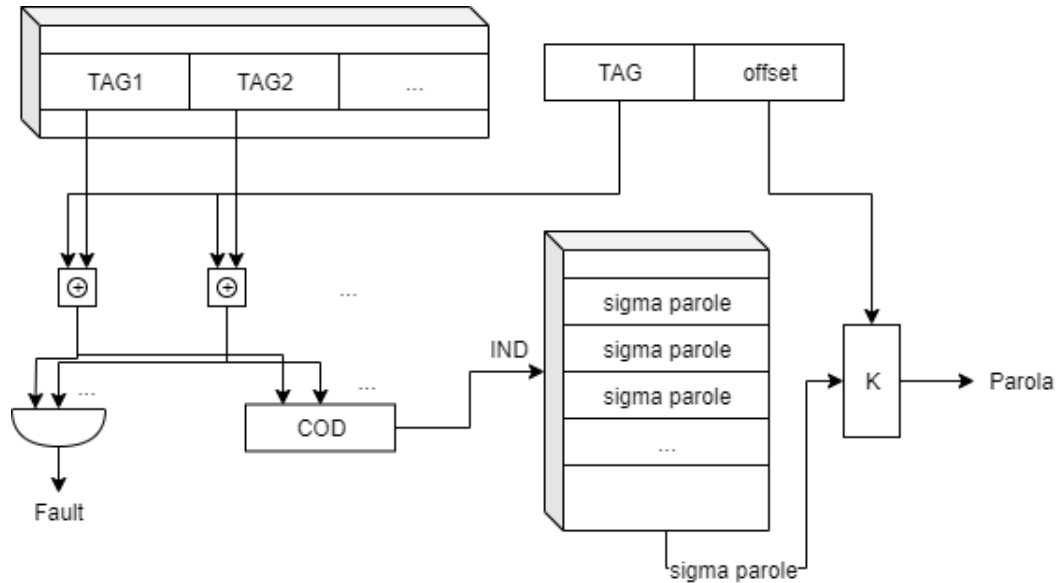
**Vantaggi e svantaggi** i vantaggi di questo metodo sono **la velocità** e **la semplicità**.

Presenta uno svantaggio che può essere molto **gravoso** e deriva dalla **rigidità della legge di corrispondenza**: quando devo accedere più volte a coppie di informazioni che stanno in blocchi della memoria principale corrispondenti allo stesso blocco della cache (quindi con **# linee** uguali), il numero di fault è molto elevato (**trashing**)



### Indirizzamento completamente associativo

Questo metodo **offre la massima flessibilità circa la corrispondenza tra le linee di M e quelle di C**: ogni blocco o linea di M può risiedere in qualsiasi blocco di C. Per vedere se ho l'indirizzo memorizzato enlla cache, considero l'**AND** tra il risultato dei confronti, se è 0 allora esiste il tag giusto, sennò **fault di cache**.



Offre la massima flessibilità al prezzo di un maggiore tempo di accesso e di un aumento di costo dovuto alla memoria associativa

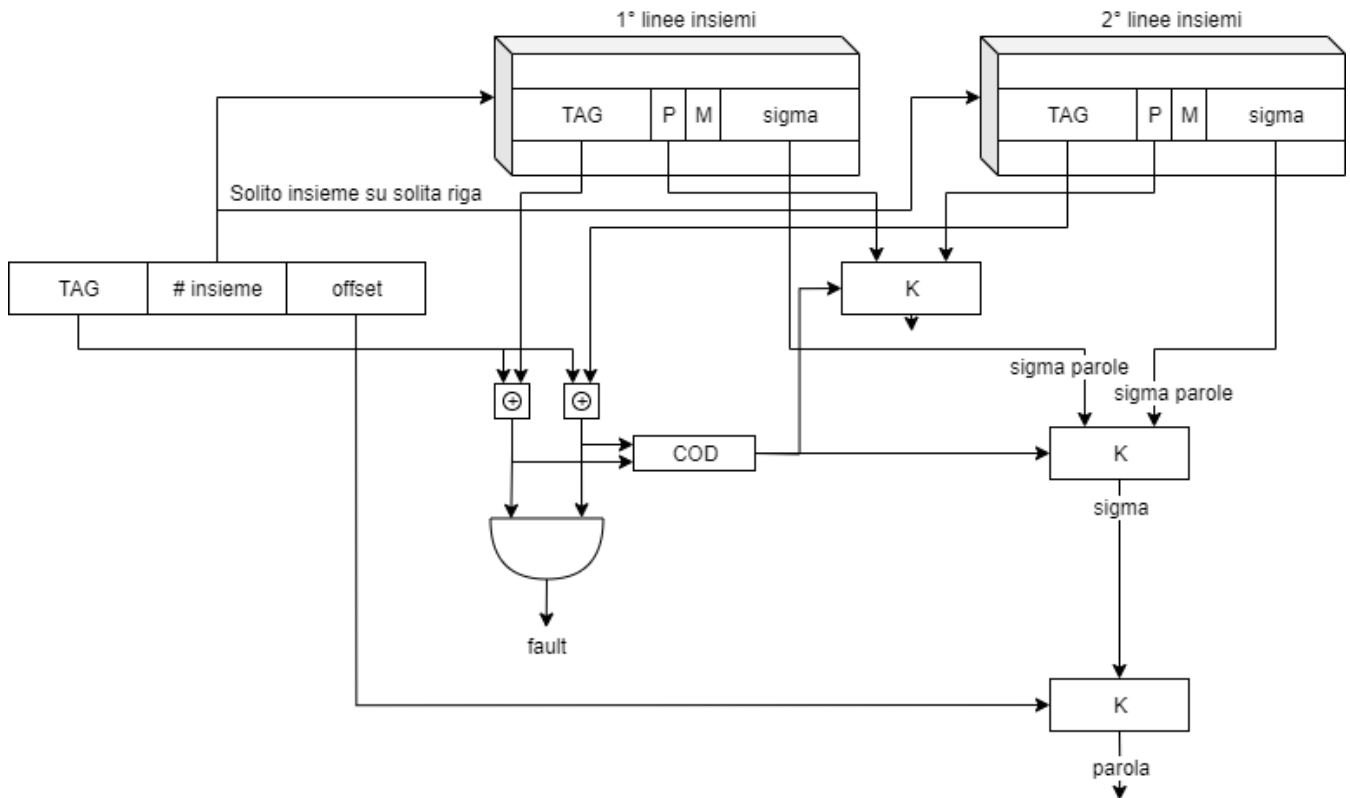
### Indirizzamento associativo su insiemi

Questo metodo **approssima** in maniera soddisfacente sia **la flessibilità** del metodo completamente associativo, sia **la semplicità** del metodo diretto. Ogni linea di memoria viene fatta **corrispondere ad un insieme determinato di linee cache, potendo essere allocato in una qualsiasi linea di tale insieme**.

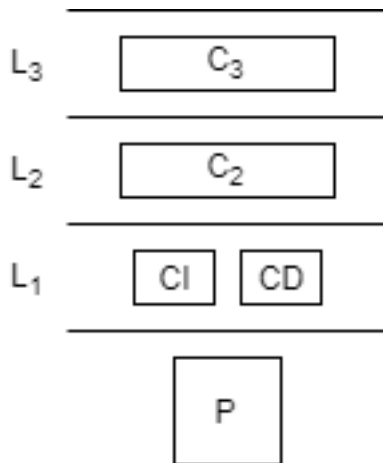
$\text{insieme} = \{\text{gruppo di linee/blocchi di cache di dimensione } k\}$  Dall'indirizzo fisico, prendo i bit che mi indicano il  $\#ins$  e decido quale insieme andare a guardare. Dopo prendo i TAG dalle linee dell'insieme in esame e li confronto con  $IND.TAG$ , decidendo quale linea prendere o se c'è un **fault di cache**.

Successivamente con  $IND.offset$  decido quale delle  $\sigma$  parole prendere.

Di seguito l'implementazione a livello firmware con  $k = 2$ , per questioni di semplicità dello schema.



## Livelli di Cache



**3° Livello di cache** – 10 Mb – Dati e istruzioni in una sola cache

**2° Livello di cache** – 1 Mb – Dati e istruzioni in una sola cache

**1° Livello di cache** – 10 Kb – Una cache per istruzioni, una cache per dati

Tutte le cache che lavorano su dati (CD, C<sub>2</sub>, C<sub>3</sub>) usano un **metodo associativo su insiemi**, mentre la cache istruzioni può essere ad indirizzamento diretto o associativo su insiemi.

Nel 1° livello ci sono **solo i dati del processo in esecuzione** (cache molto piccola), al cambio di contesto si cancella il contenuto della cache (logicamente, la rimozione fisica richiede tantissimo tempo, cioè **i bit di presenza vengono settati a 0**).

A questo livello ho una **politica on-demand**: quando ho un fault rimpiazzo una linea di cache con la linea richiesta.

Nel 2° livello si potrebbe avere una politica di **pre-fetch**: se accedo ad un indirizzo tento di portare in cache anche la linea che contiene tale indirizzo +  $\sigma$  se non è già presente (**precaricamento**).

Quindi abbiamo dati e codice di processi diversi con politica ondemand e prefetching.

Il 3° livello è analogo al secondo, ma più capiente.

## Modifiche in Cache

Abbiamo **due strategie** per effettuare le modifiche in cache:

### Write Back

Ogni STORE **modifica l'indirizzo solo in cache, successivamente** – quando dobbiamo liberare la linea modificata – **la copiamo in memoria principale** o nel livello superiore della gerarchia di memoria.

### Write Through

Ogni STORE **modifica il dato in cache e immediatamente in memoria principale** (o nel livello superiore della gerarchia).

La **scrittura in cache** è **sincrona**, mentre nel livello superiore è **asincrona**.

**Un esempio** Vediamo come esempio l'inizializzazione di un array, per valutare le prestazioni dei due metodi.

		<b>Costo</b>
<code>for (i = 0; i &lt; n; i++) x[i] = 1;</code>		
1. <code>loop:</code>	<code>STORE Rbx, Ri, R1</code>	1. $5\tau + 2t_a$
2.	<code>INC Ri</code>	2. $3\tau + t_a$
3.	<code>IF&lt; Ri, Rn, loop</code>	3. $4\tau + t_a$
4.	<code>END...</code>	$= 12\tau + t_a = 20\tau$

Facendo l'ipotesi che lavoriamo su cache di primo livello  $C_1$ :

CD (Associativo su insiemi):  $2\tau$

CI (Diretto):  $2\tau$

**Write Through** Per il metodo write through, perché funzioni devo **verificare che la banda della STORE sia minore o uguale della banda del livello successivo della gerarchia di memoria**.

**Write Back** Nel caso del write back **non ho bisogno di fare questa verifica** poiché **vado a scrivere nel livello successivo della gerarchia di memoria solo quando ho un fault** e quindi finché non è stato copiato il tutto il processore non riceve le istruzioni, quindi non genera ulteriori modifiche.

## Modello di costo

**Dal file ASM** # istruzioni \* durata

⇒ Tempo di completamento ideale (accessi in cache) =  $T_{cid}$

⇒ Tempo di completamento reale  $T_c = T_{cid} + \#Fault * T_{transf}$ .

**Dove**  $T_{transf}$ : tempo di trattamento del singolo fault, dipende unicamente dall'hardware.

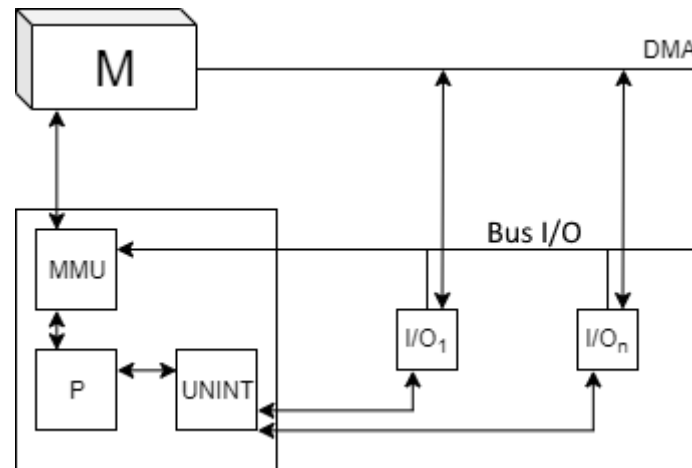
$\#Fault$ : dipende dal codice (working set) e in parte dall'hardware (capacità della cache).

$T_{transf} = 2(T_{tr} + \tau) + \frac{\sigma}{m}\tau_m$  ( oppure, se  $T_{tr} \ll \tau_m$ ,  $= 2(T_{tr} + \tau) + \frac{\sigma}{m}T_{tr}$  )

**Sempre proporzionale a**  $\frac{\sigma}{m}$ , con  $T_{tr}$  tempo di trasferimento da cache a memoria e  $m$  le parole lette alla volta.



## 7.2 I/O



Ogni unità I/O svolge il compito di interfacciare, nei confronti dell'Unità Centrale, un certo tipo di dispositivo periferico come: hard disk, stampanti, interfacce di rete....

**Categorie** Possiamo dividere le unità I/O in due categorie:

**Carattere:** unità che trasferiscono **pochi byte** per ogni operazione effettuata, ad esempio il mouse

**Blocchi:** unità che lavorano con una **grossa quantità di dati**, ad esempio hard disk o interfacce di rete.

Per queste unità è previsto un **bus DMA** che gli **permette di lavorare con i dati della memoria principale indipendentemente dal lavoro che sta svolgendo il processore** in quel momento.

L'**interazione** tra processore e il **sottosistema I/O** avviene **mediante i segnali di interruzione**. Questo è dovuto al fatto che i tempi di reazione delle unità periferiche sono molto maggiori rispetto al tempo di reazione del processore P, quindi non vale la pena fare attendere P o fare polling di ogni dispositivo.

### 7.2.1 Trasferimento dati

Esistono due modelli principali che descrivono il trasferimento dei dati tra processore/memoria e le periferiche I/O: **Memory Mapped I/O** e **uso di istruzioni speciali**.

Nello spazio di memoria virtuale, una certa porzione viene dichiarata come "riservata" e la **mappa sull'I/O**.

Se questa mappatura avviene in maniera per cui io **posso direttamente usare l'indirizzo della MMU**, che interpreta l'indirizzo del sottosistema I/O allora **parliamo di Memory Mapped I/O**.

Se invece **uso delle istruzioni speciali** allora parliamo di istruzioni speciali, meccanismi di comunicazione tra processore le periferiche, che usano quindi **interfacce dedicate**.

**Memory Mapped I/O** Per il MMI/O il meccanismo di comunicazione/esecuzione con I/O è il seguente.

Si inizia con delle **STORE** con il **comando da eseguire sulla periferica** con i suoi parametri, nella memoria virtuale che rappresenta la memoria dell'unità I/O.

Dopo il processore deve **ordinare il comando alla periferica**: questo avviene come una **STORE #1** sul registro RDY della periferica.

Successivamente il processore attende una interruzione, nel mentre può eseguire altri processi. Quando avviene l'interruzione, si eseguono delle **LOAD** sulla memoria mappata dell'I/O per leggere i risultati.

## 7.3 Trattamento delle Interruzioni

Quando una periferica I/O richiede una interruzione, manda una **richiesta di interruzione all'unità UNINT**. Se l'**unità di arbitraggio UNINT accetta la richiesta**, manda il segnale INT (interruzione) al processore che provvede a mandare un ACK.

UNINT manda anche un segnale all'unità I/O che ha vinto l'arbitraggio, che è libera di mandare alla MMU due parole: **numero dell'unità** – per capire quale unità ha richiesto l'interruzione – e **la ragione dell'interruzione**.

La MMU, a questo punto, manda le due parole ricevute al processore come se fossero due LOAD.

```
trattint.      reset INT, set ACKINT, trattint1
trattint1.    (ACKM, OR(ESITO) = 00) nop, trattint1
              (= 10) DATAIN -> R[61], reset ACKM, SET RDYM, trattint2      *
trattint2.    (ACKM, OR(ESITO) = 00) nop, trattint2
              (= 10) DATAIN -> R[62], IC -> R[63], set RDYM, reset ACKM,
              R[60] -> IC, fch0 ** ; da qui fase ASM
```

\* Ricevuto il codice dell'evento, questo viene passato alla Routine d'Interfacciamento Interruzioni via R[61].

\*\* Ricevuta la seconda parola del messaggio di I/O, questa viene passata alla Routine d'Interfacciamento Interruzioni via R[62]. Viene poi salvato l'indirizzo di ritorno dalla routine ed effettuato il salto alla routine stessa.

Quando ho una interruzione, **smetto di fare ciò che sto facendo** e vado ad eseguire il codice che tratta questa interruzione **eseguendo solo quel codice** che deve essere molto veloce.

### 7.3.1 DMA I/O

**Direct Memory Access** La memoria principale ha due interfacce: una che comunica con il processore e una che comunica con il bus DMA. Quindi M è un'unità attiva che **contiene un arbitro** per stabilire se comunicare con P o il Bus.

Un esempio con una periferica I/O come un hard disk. Come primo passo vengono memorizzati operazioni e parametri via MM I/O sulla periferica, dopo di memorizza "GO" in RDY della periferica.

Adesso la periferica I/O completa l'operazione, accede al bus DMA ed esegue il ciclo di trasferimento da M a M I/O. A questo punto richiede una interruzione ed attende che sia accolta.

# Capitolo 8

## Livello dei processi

Un processo può essere immaginato **come l'esecuzione di un programma**. Una tipica situazione è quella in cui si **distinguono due sottoinsiemi** di processi:

### Processi di Sistema

Sono processi che **esistono permanentemente nel sistema** e sono **delegati alla gestione di risorse e servizi** nei confronti di richieste dalle applicazioni. Possono anche cooperare tra loro oltre che con le applicazioni.

### Processi Applicativi

Derivano dalla **compilazione** e dalla **richiesta di esecuzione di programmi applicativi**. In generale **nascono e muoiono**.

L'interazione fra processi può avvenire in due modi distinti e molto differenti tra loro: a **scambio di messaggi** – cooperano spedendo informazioni – o a **memoria condivisa** – più processi con accesso alle medesime locazioni di memoria, con opportune tecniche di traduzione degli indirizzi.

## 8.1 Supporto a tempo di esecuzione

Tale supporto definisce la macchina virtuale che, rispetto alla macchina sottostante (assembler, firmware, hardware) possiede in più il concetto di processo e di meccanismi di concorrenza e di cooperazione tra processi.

**Descrittore di processo** Il **Process Control Block (PCB)** è l'insieme delle informazioni che permette al sistema di gestire il processo. All'interno troviamo i seguenti campi:

**Stato del processo:** indica se il processo è in esecuzione, attesa di una periferica o potrebbe essere in esecuzione ma in quel momento non può usare il processore.

**Area Salvataggio dei registri:** area di memoria dove ci sono le struttura dati in cui, dal momento in cui decidiamo che un processo non ha più il controllo della CPU, copiamo il contenuto dei registri e dell'IC. Questo in modo da poter rimandare in esecuzione il processo dal punto dove è stato sospeso.

**Puntatore alla tabella di rilocalizzazione:** così che quando il processo è in esecuzione la MMU possa tradurre correttamente gli indirizzi logici del processo.

**Riferimento alla lista dei processi "pronti"**

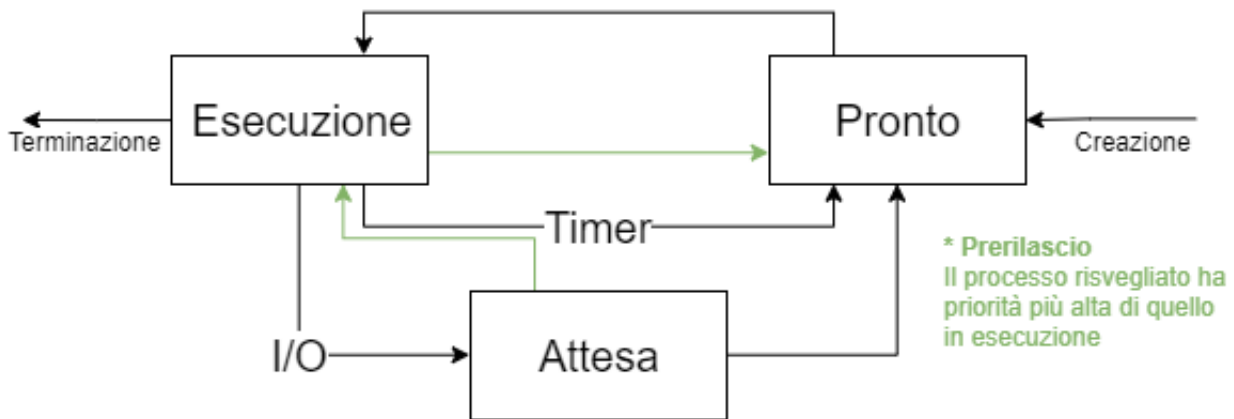
**Riferimento al processo in esecuzione**

**Riferimenti alle strutture condivise:** per interazioni fra processi a memoria condivisa.

## 8.2 Schedulazione a basso livello

Con tali informazioni si organizza la **schedulazione a basso livello**, ovvero il **complesso di funzionalità per la gestione degli stati di avanzamento dei processi** e quindi la gestione della risorse processore.

Un tipico schema di avanzamento è il seguente:



### Creazione

Crea un PCB

Carica almeno la prima pagina di codice in memoria principale, modificando la tabella di rilocalizzazione del processo

Mette il PCB nella lista dei processi pronti

### Ready → Exec

Copia di registri e IC da PCB nei registri firmware

Istruzione particolare per caricare IC

Istruzione particolare che passa l'indirizzo della tabella di rilocalizzazione alla MMU

### Exec → Ready

Terminazione attivata da un timer

Salvataggio dei registri e IC nel PCB

PCB viene messo in coda alla lista dei processi pronti

### Exec → Wait

Terminazione attivata da una richiesta I/O

Salvataggio dei registri e IC nel PCB

Registro il PCB come in attesa

### Wait → Ready

Transizione su completamento dell'operazione I/O

Inserimento in fondo alla lista dei processi pronti

### Terminazione

Cancellazione del PCB

### Wait → Exec (Priorità)

Transizione attivata dal completamento dell'operazione I/O per un processo a priorità maggiore di quello in esecuzione.

Copia dei registri e IC nei registri firmware e passaggio dell'indirizzo della tabella di rilocalizzazione ad MMU

### Exec → Ready (Prerilascio)

Mette il PCB **in testa** alla coda dei processi pronti.

Salvataggio dei registri e IC nel PCB

## 8.3 Istruzione speciale Start-Process (D-RISC)

Questa istruzione **conclude la fase di commutazione di contesto**: prende due parametri:

$R_{IC}$ , che contiene l'indirizzo della prima istruzione da eseguire quando il processo andrà in esecuzione

$R_{tabRil}$  che contiene l'indirizzo della tabella di rilocazione del processo da inviare alla MMU

```
startp0.      "startprocess" -> OP, R[IR.Rtabril] -> IND, set RDYM, startp1
startp1.      (ACKM, OR(ESITO) = 0-) nop, startp1.
              (= 10) R[IR.Ric] -> IC, fch0
              (= 11) R[IR.Ric] -> IC, trattint
```

Invio alla MMU le informazioni necessarie a riferire la tabella di rilocazione del processo che entra in esecuzione. Contemporaneamente ripristino il contenuto di IC all'indirizzo logico della prima istruzione da cui il processo deve riprendere o iniziare l'esecuzione.

Ciò permette di effettuare in maniera atomica tutte le azioni necessarie a iniziare l'esecuzione di un processo una volta che i registri generali sono stati ripristinati.

## 8.4 Commutazione di contesto

Trattiamo la **commutazione di contesto** che avviene alla **fine del quanto di tempo**.

**Situazione** Abbiamo un processo ProcA in esecuzione. Quando il tempo di esecuzione finisce, il registro **INT** diventerà **1** quindi si salterà al **trattamento dell'interruzione** (trattint): **l'IC corrente verrà salvato in R[63]** (fase firmware) ecc. . .

Successivamente si passa alla **fase assembler** ASM, dove vengono eseguiti i seguenti passaggi:

**Salvare lo stato** di ProcA in esecuzione (una serie di STORE per da PCB+posizione. . .)

**Posizionare  $PCB_A$**  in coda alla lista dei processi pronti

**Ripristino lo stato del primo processo nella lista pronti** nei registri del processore (una serie di LOAD)

**Rimuovere il puntatore al processo nella lista pronti e aggiornare lo stato della EXEC**

**Start-Process**, quindi passo IC e l'indirizzo della tabella di rilocazione del nuovo processo in esecuzione

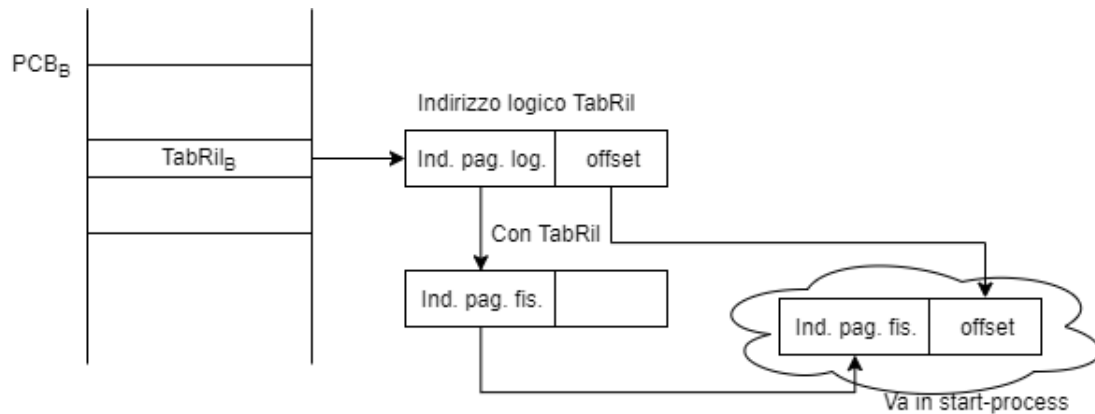
**Commuta Contesto** Tutti i passaggi prendono il nome di **commuta contesto**. Tale codice viene eseguito quando è sempre attiva la tabella di rilocazione di A, quindi il codice di commuta contesto deve esser **presente nella memoria virtuale di A** – e in tutti gli altri processi. Inoltre, **tutti i PCB devono essere accessibili a tutti i processi** per completare la commutazione di contesto.

Tutti i passaggi fin'ora descritti vengono **effettuati con la MMU che lavora mediante la tabella di rilocazione di A**. Sarà la **start-process** ad occuparsi di **cambiare la tabella di rilocazione con quella del nuovo processo**, da cui sorge un problema: **che tipo di indirizzo** (logico o fisico) **devo mandare come parametro alla start-process per la nuova tabella di rilocazione?**

Quando inizia a lavorare, la start-process deve **liberare la cache della MMU** (setta tutti i bit P a 0) per poter caricare i valori della nuova tabella di rilocazione. Quindi **l'indirizzo passato come parametro non può essere logico** perché in tale situazione **non c'è modo di tradurlo in indirizzo fisico dato che la cache della MMU è "vuota"**.

Quindi il parametro della start-process **deve essere un indirizzo fisico**.

Nella PCB è memorizzato l'indirizzo logico della tabella di rilocazione del processo, quindi **serve un meccanismo che traduca molto velocemente tale indirizzo in fisico**: prendo la pagina logica e con essa vado a chiedere dove si trova in memoria quella pagina, ci metto l'indirizzo di pagina fisica e rimando il tutto come indirizzo fisico alla start-process.



**Problema** Il problema di questo meccanismo è che **per capire dove sia l'indirizzo fisico della tabella di rilocazione del nuovo processo, devo convertire l'indirizzo logico con la tabella di rilocazione del nuovo processo e non con quella del vecchio processo.**  
Quindi **da ogni processo deve essere accessibile l'indirizzo logico delle tabelle di rilocazione di ogni altro processo.**

## 8.5 Condivisione indirizzi tra processi

Vediamo come due o più processi possono **usare indirizzi in comune.**

I vari indirizzi logici dei vari processi devono essere tradotti nell'unico indirizzo fisico dove risiede realmente il dato in comune. Abbiamo **tre soluzioni** per ottenere quei dati:

**Gli indirizzi logici in A e B sono gli stessi** e corrispondono alla stessa locazione di memoria fisica. Come se  $TabRil(x)_A = TabRil(x)_B$ .

**Gli indirizzi logici di x in A e B sono diversi ma puntano alla stessa locazione di memoria.**

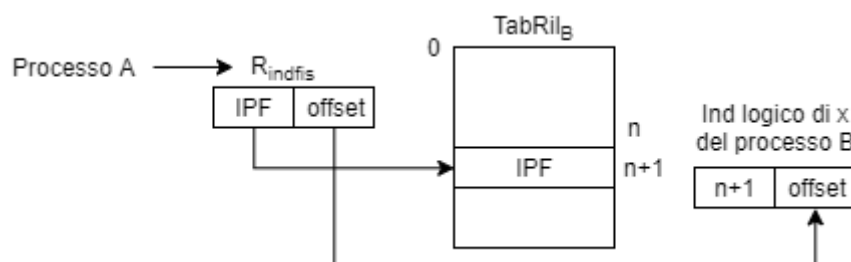
**Capability:** il processo A comunica l'indirizzo fisico della variabile condivisa al processo B in modo che possa accedere a tale variabile.

Il processo A deve calcolare quindi l'indirizzo fisico di X e conosce  $R_{baseX}$  (cioè pagina indirizzo di pagina logica ed offset),  $R_{TabRil}$ ,  $R_{PCB}$

<pre> SHR      Rbasex, # bit offset, R1 LOAD     Rtabril, R1, R2 AND      Rbasex, #00..011..1, R3 AND      R2, #11..100..0, R4 ADD      R3m R4, Rindfis ;IPF = indirizzo pagina fisica ;IPL = indirizzo pagina logica </pre>	<pre> ;Rbasex = [IPL   offset] ;[0000   IPL] ;[IPF   ///] ;[0000   offset] Tanti 1 quanti i bit offset ;[IPF   0000] ;[IPF   offset] </pre>
--	---

Con  $R_{indfis}$  non posso fare nè LOAD nè STORE perché la MMU interpreterebbe tale indirizzo come logico, quindi lo andrebbe a tradurre, ma posso mandare tale indirizzo al processo B.

Il processo B **deve trattare  $R_{indfis}$  come dato e non come indirizzo**, quindi dobbiamo trovare un modo per **convertire tale indirizzo fisico in logico** seguendo la tabella di rilocazione del processo B. La soluzione è quella di **scrivere i bit IPF di  $R_{indfis}$  nella prima posizione libera della tabella di rilocazione di B**, in modo tale che accedendo a quella posizione la MMU possa correttamente tradurre l'indirizzo logico in fisico e accedere alla variabile condivisa.



## Capitolo 9

# Elaborazione in parallelo

Fin'ora abbiamo studiato il comportamento, costo e prestazioni di un **processore monolitico** che **esegue le istruzioni una dopo l'altra**.

Abbiamo anche detto che le prestazioni di un processore sono calcolate secondo la funzione  $P = f(K, \tau, t_a)$  dove  $K$  è il numero di cicli di clock necessari per eseguire una istruzione,  $\tau$  è la lunghezza del ciclo di clock e  $t_a$  è il tempo di accesso alla memoria principale.

Il tempo di accesso alla memoria è già stato "ottimizzato" introducendo la cache e la gerarchia di memoria. Cerchiamo adesso di **diminuire il  $K$  e in misura minore il  $\tau$** , introducendo nuovi tipi di processore che riescono ad **operare in parallelo**.

### 9.1 Forme di parallelismo

#### 9.1.1 Pipeline

La forma di parallelismo **pipeline** è definita come **una catena di Stadi Interconnessi fra loro, che lavorano ognuno producendo qualcosa che verrà preso in ingresso dallo stadio successivo**.

**Ogni stadio** del processore **ha un compito ben preciso** per arrivare ad un risultato.

Definiamo alcune **metriche** per capire quali sono le prestazioni di questa tecnica:

**Latenza** –  $L$

**Tempo per completare un calcolo** da quando inizia a quando finisce. Ogni stadio  $i$  ha una latenza  $L_i$ .

**Tempo di servizio** –  $T_S$

**Tempo che intercorre fra l'invio di due risultati successivi**. Nel processore monolitico  $L = T_S$

**Throughput** –  $B = \frac{1}{T_S}$

$L_{pipe} = \sum_{i=1}^{n_{stadi}} (L_i + t_{comm})$ , dove:

$L_i$  è la latenza del singolo stadio

$t_{comm}$  è la latenza di comunicazione

$T_{Spipe} = \max\{T_{Si}\}$

**Tempo di completamento** –  $T_C$

**Tempo fra l'arrivo del primo input e l'uscita dell'ultimo output**.

$T_{Cpipe} \simeq m * T_S$ , con  $m$  numero di task

La migliore condizione per una pipeline è **quando tutti gli stadi hanno lo stesso tempo di servizio**.

**Efficienza** Su tutte queste metriche andiamo a misurare l'**efficienza**, ovvero il **rapporto tra il tempo ideale  $T_{id}$  e il tempo misurato  $T(n)$** .

$\varepsilon(n) = \frac{T_{id}(n)}{T(n)}$  — Quando  $m \gg n$  allora  $\varepsilon \rightarrow 1$  (  $\varepsilon = 1$  è ideale) —  $T_{id} = \frac{T_{seq}}{n}$  Se supponiamo di trasformare un

modulo (sistema) sequenziale in una struttura parallela equivalente con grado di parallelismo  $n$ , ha senso parlare di **scalabilità** o **speed-up**: lo speed-up usa al numeratore il miglior tempo sequenziale, mentre la scalabilità usa il tempo di un gradi di parallelismo pari a 1.

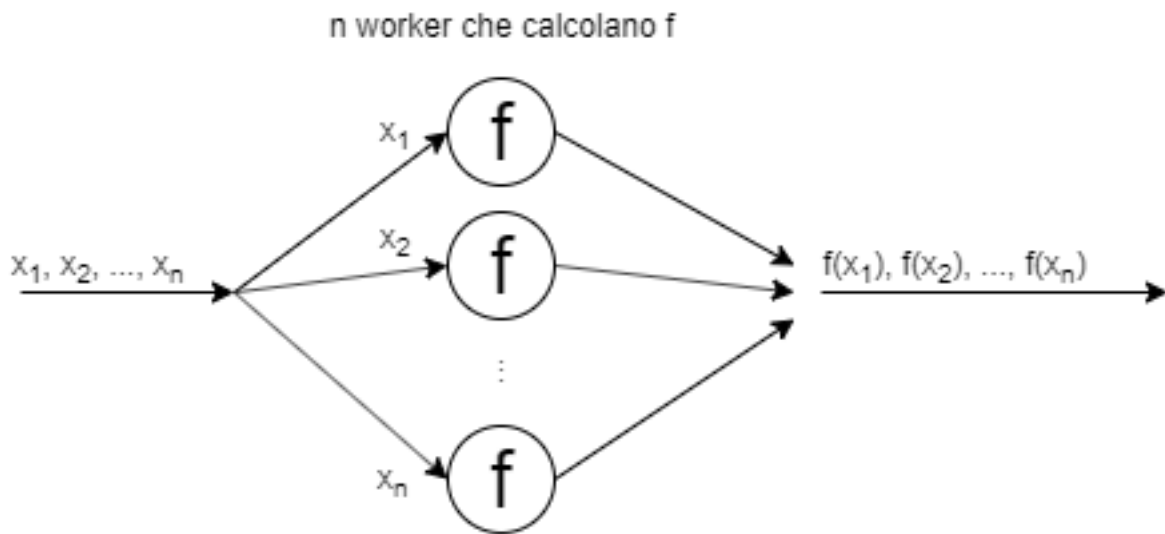
$$sp(n) = \frac{T_{seq}}{T(n)} \quad \text{---} \quad sc(n) = \frac{T(1)}{T(n)}$$



$$T_{id} = \frac{T_{seq}}{n} \\ \varepsilon(n) = \frac{T_{id}}{T(n)} = \frac{T_{seq}}{n * T(n)} = \frac{sp(n)}{n} \Rightarrow sp(n) = \varepsilon(n) * n$$

### 9.1.2 Farm

**Replicazione Funzionale** Questa forma di parallelismo lavora **dividendo i dati in ingresso su vari stadi in modo che essi lavorino parallelamente**.



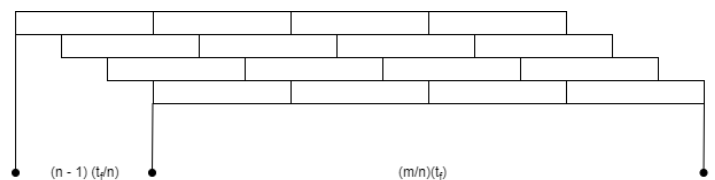
$t_f$  = latenza calcolo di  $f$

$$L_{farm} = t_{sched} + t_f + t_{collez}$$

$$T_{Sfarm} = \max\{t_{sched}, \frac{t_f}{n}, t_{collez}\}$$

$$T_{Cfarm} = (n - 1) \frac{t_f}{n} + \frac{m}{n} * t_f$$

$$\Rightarrow T_{Cfarm} = (n - 1)T_S + \frac{m}{n} * T_S$$

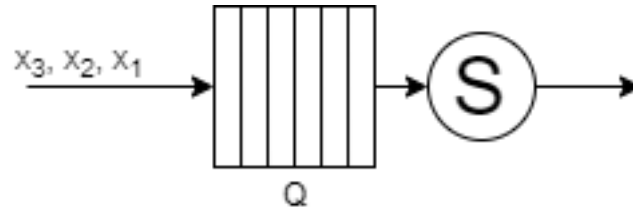


**Rimozione di colli di bottiglia** Se nella pipeline ho uno stadio che impiega molto più tempo dei restati, posso dividere quello stadio in  $K$  worker usando una replicazione funzionale (farm) per abbassare il suo  $T_S$  in modo da renderlo simile agli altri.



## Teoria delle Code

Un sistema a coda modella il comportamento di un **servente S** (centro di servizio) al quale **uno o più clienti**  $C_1, \dots, C_n$  **si rivolgono** attendendo in una **fila di attesa Q** di ricevere il servizio erogato.



$t_a$  tempo di interarrivo

$t_s$  tempo di servizio

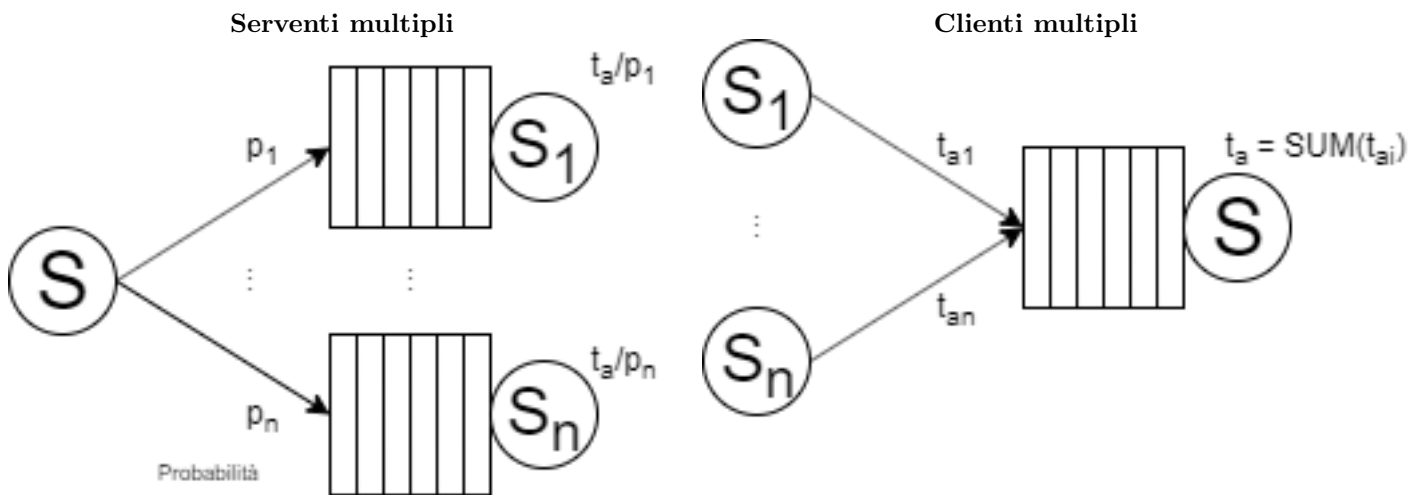
1 task ogni  $t_a$  dalla coda

1 task ogni  $t_s$  dal sistema

**Fattore di utilizzo**  $p = \frac{t_s}{t_a}$

L'obiettivo è avere questo valore  $< 1$ , altrimenti arriverebbero in coda più task di quanti se ne riescano ad elaborare (**collo di bottiglia**).

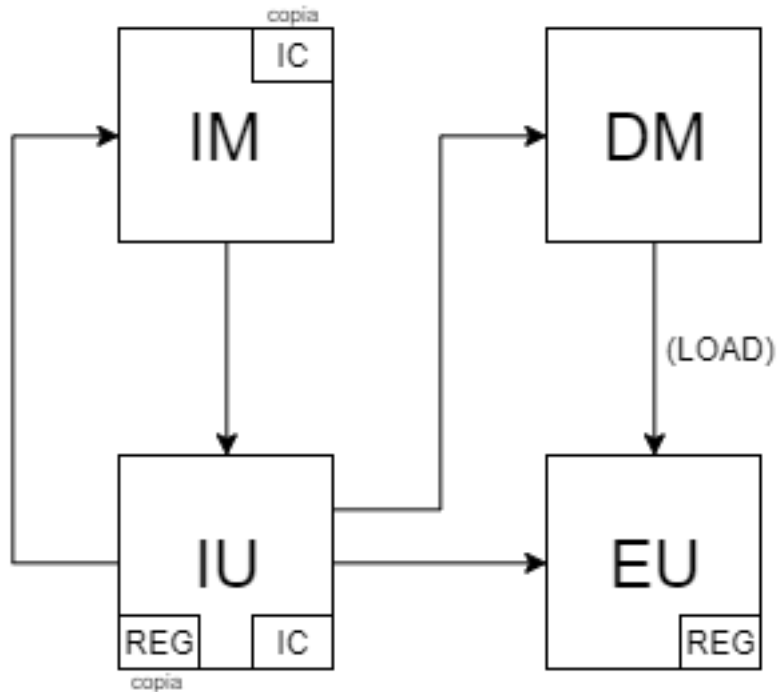
Chiaramente si intende un valore **molto vicino a 1**, altrimenti avrei **tempi dove non arrivano task** ed il **servente non processerebbe nulla** quindi uno spreco di risorse.



## 9.2 Processore

### 9.2.1 Processore Pipeline

Il concetto alla base di questa architettura è la **parallelizzazione della CPU mediante la parallelizzazione dell'interprete firmware**, eseguito dal processore con la collaborazione delle altre unità della CPU stessa. Per scopo didattico, useremo una visione semplificata dell'architettura composta da soli **quattro stadi**.



#### Instruction Memory – IM

Unità memoria dedicata a **memorizzare solo istruzioni** e quindi a **compiere operazioni di fetch**. Dotata di MMU propria.

#### Instruction Unit – IU

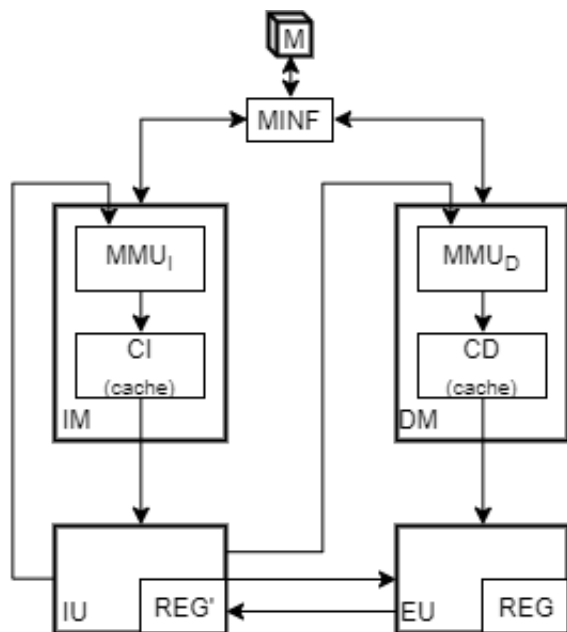
**Cuore vero e proprio** del processore, dove le **istruzioni che provengono da IM** vengono **decodificate ed eseguite**. Si collega alla IM per le istruzioni di salto.

#### Data Memory – DM

Unità memoria dedicata a **memorizzare solo dati**. Il collegamento dalla IU è dato dalla decodifica di istruzioni di **LOAD** e **STORE**.

#### Execution Unit – EU

Unità che si occupa di **eseguire istruzioni aritmetico-logiche**. Il collegamento dalla DM è dato per concludere le istruzioni di **LOAD**.



La IM ha al suo interno la propria MMU<sub>I</sub>, per la traduzione degli indirizzi delle istruzioni, e la cache istruzioni CI. Tale unità è collegata tramite la MMU<sub>I</sub> alla IU, per prelevare IC nel caso di istruzioni di salto, e alla **memory interface MINF** per dialogare con il sottosistema di memoria.

Analogamente la DM avrà una propria MMU<sub>D</sub> e una propria cache dati CD. Anch'essa è collegata tramite MMU<sub>D</sub> alla IU e a MINF.

La EU ha una copia dei registri per effettuare le operazioni aritmetico-logiche.

Anche IU ha una copia dei registri di EU (il più aggiornati possibile) per alcune operazioni di decodifica.

### Istruzioni operative

Le **istruzioni operative compieranno sempre il giro IM → IU → EU.**

IM, usando la copia che ha di IC, manda una coppia **<istr, IC'>** – dove **istr** è l'istruzione da eseguire e **IC'** indica da dove è stata presa e che serve per trattare le interruzioni di salto.

IU invia alla EU una **istruzione decodificata**, cioè una **tupla di valori** – per esempio **ADD R<sub>1</sub>, R<sub>2</sub>, R<sub>3</sub> → <+, 1, 2, 3>** dove 1, 2, 3 corrispondono a indirizzi di registri.

EU esegue l'istruzione e manda il risultato alla IU "avvertendola" che una copia di registri deve essere aggiornata con il risultato appena calcolato.

### Istruzioni di salto

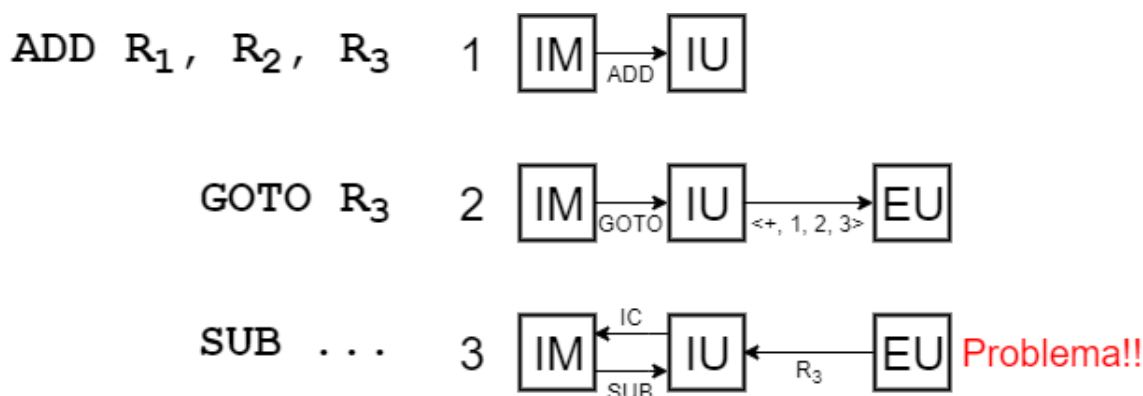
Le istruzioni di **salto incondizionato compieranno il giro IM → IU.**

IM manda **<istr, IC'>** – per esempio **GOTO R<sub>4</sub>**.

IU va nella propria copia dei registri, accede al contenuto del registro R<sub>4</sub> e invia alla IM un messaggio dicendo che il nuovo IC deve essere il contenuto del registro, quindi la IM aggiorna IC.

### Meccanismi per sincronizzazione

Partiamo da un esempio per capire il problema. Consideriamo il seguente set d'istruzioni:



Si può notare che al passo 3 è presente un problema di sincronizzazione poiché, benché nel codice abbiamo una istruzione di salto, la IM manda alla IU l'istruzione successiva "quasi" ignorando il salto, mentre la IU manda alla IM il valore nuovo di IC.

Per ogni registro affianchiamo un contatore C, il quale funge da indicatore della correttezza del contenuto del registro.

Inizialmente  $C = 0$ , quando dalla IU alla EU mando un'istruzione che scriverà in quel registro, incremento il contatore.

Quando dalla EU si riceverà il risultato aggiornato lo decremento.

La IU leggerà il registro  $\Leftrightarrow C == 0$ , altrimenti si blocca.

### Meccanismi di gestione della copia IC e IC'

IU aggiorna costantemente il suo Instruction Counter, invia IC a IM ogni volta che questo viene aggiornato e scarta le istruzioni che arrivano da IM e che non hanno IC' della tupla uguale al proprio valore IC.

### Istruzioni LOAD

Le istruzioni LOAD compiono il giro  $IM \rightarrow IU \rightarrow DM \rightarrow EU \rightarrow IU$ .


IM compie la fetch dell'istruzione di LOAD e la manda alla IU

IU manda un ordine di caricamento alla DM  $\langle \text{load}, \text{ind} \rangle$  – dove  $\text{ind}$  è calcolato dalla IU con  $R_{base} + R_{ind}$ . Contemporaneamente manda a EU un messaggio che indica di aver ordinato una LOAD sul registro  $R_x \langle \text{load}, R_x \rangle$ . Infine **aggiorna il contatore di  $R_x$** .

DM compie la LOAD e la manda alla EU.

EU prende il valore mandato da DM e lo mette nel registro  $R_x$ . Manda il nuovo valore  $R_x$  a IU e aggiorna il contatore di  $R_x$ .

LOAD	R1 , R2 , R3
ADD	R3 , R4 , R5

	1	2	3	4	5	
IM	LOAD	ADD				
IU		LOAD	ADD			
DM			LOAD			
EU				LOAD	ADD	

Posso eseguire subito l'istruzione perché il contenuto di  $R_3$  è preso dalla EU, quindi già aggiornato

### Istruzioni STORE

Le istruzioni di STORE compiono il giro  $IM \rightarrow IU \rightarrow DM$ .

IM compie la fetch dell'istruzione di STORE e la manda alla IU.

IU manda un messaggio  $\langle \text{STORE}, \text{ind}, \text{val} \rangle$  alla DM che si occupa di concludere l'istruzione.

STORE	R1 , R2 , R3
ADD	R3 , R4 , R5

	1	2	3	4	5	
IM	STORE	ADD				
IU		STORE	ADD			
DM			STORE			
EU				STORE	ADD	

## 9.2.2 Dipendenze Logiche

Una istruzione **I** induce una dipendenza logica su una istruzione **J** quando **I** produce **x** sull'unità  $U_a$  che è letta da **J** sull'unità  $U_b$ , quindi scrittura e lettura avvengono su unità diverse. Un po' come la prima condizione di Bernstein ma su unità diverse.

**Esempio** Decodifico una LOAD/STORE su IU usando un registro che sta per essere modificato nella EU.

INC	R <sub>i</sub>		1	2	3	4	5	6
LOAD	Rbase, R <sub>i</sub> , R1							
<b>IM</b>	INC	LOAD						
<b>IU</b>		INC*	LOAD**	LOAD				
<b>DM</b>						LOAD		
<b>EU</b>			INC					LOAD

\*: il contatore del registro R<sub>i</sub> va a 1

INC induce una dipendenza logica sulla LOAD: scrivo R<sub>i</sub> e lo leggo dopo, INC opera su EU e LOAD su IU.      \*\*: LOAD tenta di leggere R<sub>i</sub> con contatore = 1, quindi il sistema si blocca.

**Esempio** Decodifico un salto condizionato o incondizionato su un registro

INC	R <sub>i</sub>		1	2	3	4	5	6
IF<	R <sub>i</sub> , R <sub>n</sub> , loop							
<b>IM</b>	INV	IF<						
<b>IU</b>		INC*	IF<**	IF<				
<b>DM</b>								
<b>EU</b>			INC					

\*: il contatore del registro R<sub>i</sub> va a 1

\*\*: IF< tenta di leggere R<sub>i</sub> con contatore = 1 per la valutazione della guardia, quindi il sistema si blocca.

Per tali dipendenze teniamo conto anche della distanza k, cioè quanto distano due istruzioni con dipendenza logica – negli esempi precedenti la distanza era pari a 1, quindi k = 1. Solo le dipendenze logiche con  $k \leq 2$  possono avere un peso, e il peso può aumentare se c'è una LOAD nella sequenza di istruzioni che porta alla dipendenza.

### Effetti dei salti

Partiamo da un semplice esempio per capire che effetto fanno i salti condizionati.

	ADD	R1, R2, R3		1	2	3	4	5	6
	GOTO	eti							
	SUB	R4, R5, R6							
	INC	...							
	...								
eti:	LOAD	Rp, Ri, R7							
	MUL	...							
<b>IM</b>	ADD	GOTO	SUB	LOAD	MUL				
<b>IU</b>		ADD	GOTO ↗	XSUBX	LOAD				
<b>DM</b>									
<b>EU</b>			ADD						

Cerchiamo di capire il meccanismo firmware che permette questo tipo di comportamento:

IM manda a IU <istr, IC'>

IU va nella copia dei registri e accede al contenuto del registro dove è indicato il nuovo valore di IC e invia alla IM un messaggio dicendo che il nuovo IC deve essere il contenuto di quel registro. Dopodiché aggiorna il proprio IC.

IU scarnerà tutte le istruzioni che non hanno il valore di IC' uguale al proprio IC.

## 9.3 Ottimizzazione del codice D-RISC

Nel caso del processore monolitico abbiamo visto in sostanza due tipi di ottimizzazioni, entrambe nella LOAD: il prefetch e la non-deallocazione. Questo perché il degrado delle prestazioni avveniva esclusivamente a causa dell'accesso alla memoria.

Nel processore pipeline possiamo operare in due modi: sui **salти** o sulle **dipendenze logiche** – **principale causa** del degrado delle prestazioni.

### 9.3.1 Inlining

Questa tecnica cerca di **eliminare i salti relativi alle chiamate di procedura** o funzione. Ad esempio

<pre>int inc(int x) {return x+1;} int main() {... y[i] = inc(z); ...}</pre>	<pre>inc:      INC      Rx           GOTO     Rret            ADD      Rz, R0, Rx           CALL     Rinc, Rret           STORE    Ry, Ri, Rx</pre>
---	---

	1	2	3	4	5	6	7	8	9	10
IM	CALL	STORE	INC	GOTO	xxx	STORE				
IU	ADD	CALL ↗	XSTOREX	INC	GOTO ↗	XxxxX	STORE			
DM								STORE		
EU		ADD			INC					

5 istruzioni ⇒ 7τ

Usando la parola chiave **inline** davanti ad una procedura o funzione in fase di programmazione, il compilatore va a **sostituire la chiamata di funzione con il codice stesso della funzione**.

<pre>inline int inc(int x) {return x+1;} int main() {... y[i] = inc(z); ...}</pre>	<pre>INC      Rz STORE    Ry, Ri, Rz</pre>
--	--

Si scarta così il salto di chiamata a procedura ed il salto di ritorno, evitando due bolle sicure.

### 9.3.2 Out Of Order

La compilazione di un programma è un **processo statico** durante il quale **possiamo fare molte ottimizzazioni** **ma ci è impossibile prevedere il comportamento durante l'esecuzione** – se i salti vengono fatti o meno...

Molti processori adottano uno schema di implementazione chiamato **Out Of Order Execution/Decode**. Vediamo un esempio:

<pre>INC      Ri LOAD     Rp, Ri, R1 SUB      Ri, R2, Ri</pre>		1	2	3	4	5	6	7
IM	INC	LOAD		SUB				
IU		INC	LOAD*	LOAD	SUB			
DM					LOAD	↘		
EU			INC ↗				LOAD	SUB

\* = bolla, attesa

Se avessi una IU che lavora **out of order** potrei **"accantonare"** momentaneamente l'istruzione che mi provoca il blocco/bolla e **tentare di eseguire l'istruzione successiva**.

<pre>INC      Ri LOAD     Rp, Ri, R1 SUB      Ri, R2, Ri</pre>		1	2	3	4	5	6	7
IM	INC	LOAD	SUB					
IU		INC	LOAD	SUB	LOAD			
DM							LOAD	
EU			INC ↗		SUB			LOAD

$R(\text{LOAD}) = \{R_p, R_i\}$

$W(\text{LOAD}) = \{R_i\}$

$R(\text{SUB}) = \{R_i, R_2\}$

$W(\text{SUB}) = \{R_i\}$

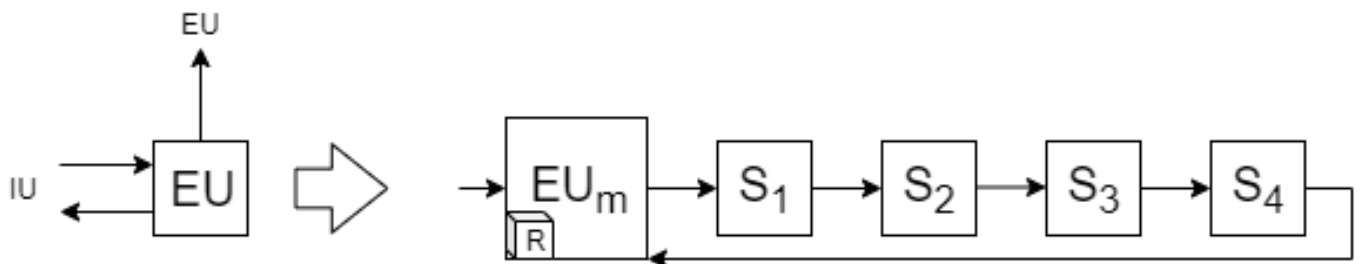
In questo esempio, per implementare out of order, dovrei **avere un registro che mantenga la copia del valore vecchio di  $R_i$**  poiché l'intersezione tra l'insieme dei registri scritti da SUB e l'insieme dei registri letti da LOAD **non è vuota**.

Se, in questo caso, la SUB fosse andata a scrivere in un altro registro che non fosse  $R_i$ , allora avrei potuto implementare tale schema out of order senza problemi e senza copie momentanee di registri.

Questo modo permette di evitare bolle o blocchi e quindi rendere più performante il processore.

### 9.3.3 Artimetiche lunghe con EU Master e Slave

Per questo tipo di istruzioni, nel processore pipeline abbiamo una soluzione molto **semplice**: **espandere le EU con delle ALU specializzate che calcolano in pipeline i risultati delle aritmetico-logiche lunghe**



Quando la  $EU_m$  riceve una istruzione aritmetico-logica lunga, **delega tale compito alla  $EU_{*/}$**  (EU slave che fa moltiplicazioni e divisioni intere), mandando all'unità una istruzione del tipo  $\langle *, R[a], R[b] \rangle$ .

Vediamo un esempio:

ADD	R1 , R2 , R3
MUL	R4 , R5 , R6
MUL	R7 , R8 , R9
ADD	R6 , R9 , R10

	1	2	3	4	5	6	7	8	9	10
IM	A	M <sub>1</sub>	M <sub>2</sub>	A <sub>2</sub>						
IU		A	M <sub>1</sub>	M <sub>2</sub>	A <sub>2</sub>					
DM										
EU <sub>m</sub>			A	M <sub>1</sub>	M <sub>2</sub>	A <sub>2</sub> *	*	*	*	A <sub>2</sub>
EU <sub>*/</sub>					M <sub>1</sub>	M <sub>1</sub> <sup>a</sup> M <sub>2</sub>	M <sub>1</sub> M <sub>2</sub>	M <sub>1</sub> ↗ M <sub>2</sub>	M <sub>2</sub> ↗	

<sup>a</sup>: lo stadio S<sub>1</sub> ha finito di calcolare M<sub>1</sub>, quindi inizia M<sub>2</sub>

Considereremo una  $EU_m$  e 3 EU slave: una per  $*/$  intere, una per  $+-$  in virgola mobile e una per  $*/$  in virgola mobile

### 9.3.4 Loop Unrolling

Questa tecnica cerca di **eliminare i salti relativi dovuti ai cicli** – cioè ai loop.

<pre>for (i = 0; i &lt; n; i++)     x[i] = a[i] + b[i];</pre>	<pre>loop:  LOAD         LOAD         ADD         STORE         IF&lt;      loop         ...</pre>
---	--

	1	2	3	4	5
IM	IF<	...	LOAD		
IU		IF< ↗	X...X	LOAD	

Ad ogni iterazione del ciclo ho sicuramente una bolla dovuta ad un salto. **Sapendo il numero di iterazioni compiute** – essendo un ciclo determinato – **possiamo cercare di operare in questo modo: sapendo che n è pari**

```
for (i = 0; i < n; i += 2) {
    x[i] = a[i] + b[i];
    x[i + 1] = a[i + 1] + b[i + 1];
}
```

In questa maniera ho **dimezzato il numero di salti dovuti al ciclo** e quindi ho ottimizzato le prestazioni. **Molti compilatori utilizzano questa tecnica.**

### 9.3.5 Delayed Branch

Un'interessante **tecnica a tempo di compilazione** è quella chiamata **delayed branch**, che può essere considerata come un **caso particolare di spostamento del codice** cioè **basata sul concetto di sfruttare i tempi morti** introdotti dalle bolle **per effettuare del lavoro utile**.

Per capire come funziona vediamo un esempio:

IF<      Ri, Rn, loop, delayed ADD      R1, R2, R3	<i>; se il salto viene preso esegui comunque la ADD                  ; e poi salta</i>
---	--

Ovviamente posso fare questa operazione **solo se lo spostamento non cambia la semantica del programma: se scrive qualcosa nessuna delle istruzioni successive deve leggerlo e se legge qualcosa nessuna delle istruzioni successive deve scriverlo**. Valido soprattutto in casi di salti con i cicli.

loop:    LOAD      Ra, Ri, R1 LOAD      Rb, Ri, R2 ADD        R1, R2, R3 STORE     Rc, Ri, R3 INC        Ri IF<        Ri, Rn, loop	loop:    LOAD      Ra, Ri, R1 LOAD      Rb, Ri, R2 ADD        R1, R2, R3 STORE     Rc, Ri, R3 INC        Ri IF<        Ri, Rn, loop, delayed LOAD      Ra, Ri, R1
--	---

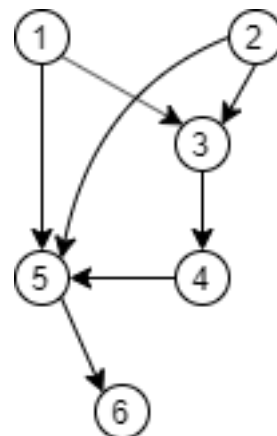
Diventa ⇒

**Questa tecnica richiede una modifica firmware, in particolare sulla IU.**

### 9.3.6 Dipendenze Logiche con Data-Flow

Partiamo da un semplice esempio:

```
1. loop: LOAD Ra, Ri, R1
2.        LOAD Rb, Ri, R2
3.        ADD R1, R2, R3
4.        STORE Rx, Ri, R3
5.        INC Ri
6.        IF< Ri, Rn, loop
```



**Grafo Data-Flow**

Fornisce un ordinamento sulle istruzioni

Il **grafo data-flow** mette in evidenza il **flusso di dati**, quindi **fornisce il minimo ordinamento dei dati per ottenere un risultato corretto nel programma**. Se si viola una qualunque delle frecce del grafo otterremo un risultato errato.

Questo grafo **risulta molto utile per cercare di allontanare le dipendenze logiche tra istruzioni mantenendo un corretto ordinamento**.



### 9.3.7 Utilizzo del registro modificato

Nell'esempio precedente possiamo notare, **anche grazie al grafo**, che **non possiamo spostare le istruzioni per aumentare l'efficienza**. Possiamo anche notare che c'è una **dipendenza logica con  $k = 1$**  tra la **ADD** e la **STORE** che comporta una bolla nell'esecuzione del programma.

Possiamo pensare di invertire l'istruzione **INC** e **STORE** in modo da allontanare la dipendenza logica pensando  $R_x$  come un registro modificato che punta sempre a  $i - 1$ .

```

loop:  LOAD Ra, Ri, R1
        LOAD Rb, Ri, R2
        ADD R1, R2, R3
        STORE Rx, Ri, R3
        INC Ri
        IF< Ri, Rn, loop

```

6 istruzioni  $\Rightarrow 10\tau - \varepsilon = \frac{6}{10}$

	1	2	3	4	5	6	7	8	9	10	11	12
IM	L	L	A	S			I	IF<		x	L	
IU		L	L	A	S*	S*	S	I	IF< *	IF< ↗	x	L
DM			L	L				S				
EU				L	L	A ↗			I ↗			

```

loop:  LOAD Ra, Ri, R1
        LOAD Rb, Ri, R2
        ADD R1, R2, R3
        INC Ri
        IF< Ri, Rn, loop, delayed
        STORE Rx, Ri, R3

```

6 istruzioni  $\Rightarrow 8\tau - \varepsilon = \frac{3}{4}$   
Miglioro l'efficienza dal 60% al 75%

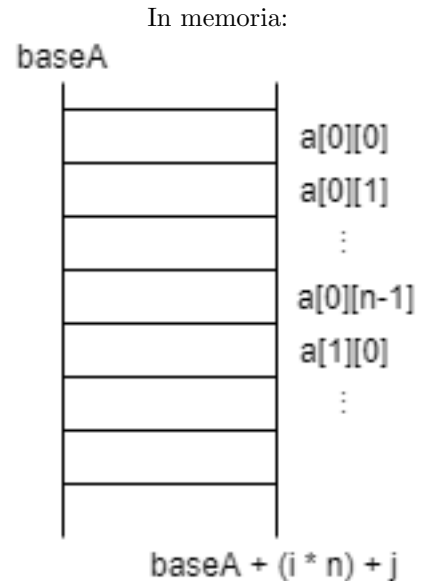
	1	2	3	4	5	6	7	8	9	10	11	12
IM	L	L	A	I	IF<			S	L			
IU		L	L	A	I	IF< *	IF< *	IF< * ↗	S	L		
DM			L	L	↘	↘				S	L	
EU				L	L	A	I ↗					L

### 9.3.8 Rimozione Invarianti

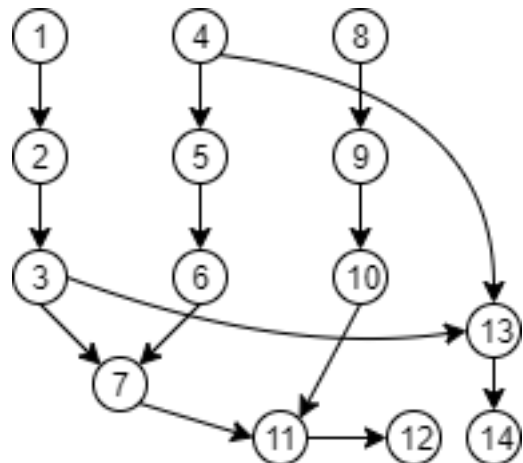
Se in un ciclo ho una istruzione che calcola qualcosa che viene consumato nelle istruzioni successive **senza che nessuno vada a modificare quel registro** e se tale istruzione non dipende dalla variabile di iterazione, allora tale istruzione la posso anticipare prima di entrare nel ciclo.

Vediamo un esempio: moltiplicazione fra matrici

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    c[i][j] = 0;
    for (k = 0; k < n; k++) {
      c[i][j] = a[i][k] * b[k][j];
    }
  }
}
```



```
1. loop: MUL R1, Rn, Rai ; INVARIANTE
2.      ADD Rai, Rbasea, Rai ; INVARIANTE
3.      LOAD Rai, Rk, R1
4.      MUL Rk, Rn, Rbk
5.      ADD Rbk, Rbaseb, Rbk
6.      LOAD Rbk, Rj, R2
7.      MUL R1, R2, R3
8.      MUL Ri, Rn, Rc ; INVARIANTE
9.      ADD Rci, Rbasec, Rci ; INVARIANTE
10.     LOAD Rci, Rj, R4
11.     ADD R4, R3, R4
12.     STORE Rci, R5, R4
13.     INC Rk
14.     IF< Rk, Rn, loop
```



Oltre a portare fuori le invarianti, possiamo anche spostare istruzioni per evitare grandi bolle causate dalle MUL e LOAD seguendo sempre lo schema data-flow.

```
4.      MUL Rk, Rn, Rbk
3.      LOAD Rai, Rk, R1
10.     LOAD Rci, Rj, R4
5.      ADD Rbk, Rbaseb, Rbk
6.      LOAD Rbk, Rj, R2
7.      MUL R1, R2, R3
11.     ADD R4, R3, R4
12.     STORE Rci, R5, R4
13.     INC Rk
14.     IF< Rk, Rn, loop
```

Se simulo questo codice noterò che riduce di molto le bolle.