Architettura degli Elaboratori

Alessio Delgadillo

Anno Accademico 2019/2020

Indice

Capitolo 1

Fondamenti di Strutturazione di Sistemi di Elaborazione

1.1 Strutturazione a livelli di interpretazione

1.1.1 Macchine virtuali e supporto a tempo di esecuzione

Le funzionalità di un sistema di elaborazione nel suo complesso possono essere viste come ripartite, secondo un certo numero di livelli, o **macchine virtuali** $\{\mathrm{MV}_0,\ \mathrm{MV}_1,\ \ldots\ ,\mathrm{MV}_n\}$. I livelli $\{\mathrm{MV}_0,\ \mathrm{MV}_1,\ \ldots\ ,\mathrm{MV}_n\}$ sono caratterizzati da un proprio **linguaggio** e sono ordinati secondo una **relazione gerarchica**. Ciò significa che ogni istruzione primitiva I_k^i, o ***meccanismo*** del linguaggio L_i $(i > 0)$ è implementata da un programma P_k, o ***politica***, scritta nel linguaggio L_j.

Il concetto che sta alla base della strutturazione a livelli è il seguente:

1. per ogni livello MV_i $(i > 0)$ l'insieme dei livelli $\{\mathrm{MV}_j\ mid 0 \leq j < i\}$ permette di implementare complessivamente il **supporto a tempo di esecuzione** di $\mathbf{L}_i$, $\mathbf{STE(L_i)}$, cioè una collezione di algoritmi e strutture dati che provvedono all'interpretazione dei meccanismi di L_i;

2. *tale supporto è a sua volta strutturato in modo gerarchico* riconoscendo un livello più alto MV_{i-1} (per $i > 1$) il cui supporto a tempo di esecuzione è implementato dai livelli $\{\mathrm{MV}_h \mid 0 < h < i-1\}$ e così via.

Questi principi sono alla base di una strutturazione dei sistemi con elevate potenzialità di *modularità*, *modificabilità*, *portabilità*, *manutenibilità* e *testabilità*.

Il livello MV_0 è tipicamente quello dei componenti fisici (elettronici) a partire dai quali viene costruita un'astrazione, o virtualizzazione, sempre maggiore delle funzionalità di interesse per il sistema. Il livello MV_n è quello che possiamo chiamare delle "applicazioni".

Notiamo che per rendere possibile l'implementazione di un meccanismo I_k^i, di MV_i occorre definire con chiarezza un'**interfaccia** nei confronti del livello MV_j che lo implementa. La natura di questa interfaccia sarà di volta in volta dipendente dallo specifico livello considerato.

1.1.2 Compilazione e interpretazione

La traduzione del programma *sorgente* scritto in linguaggio ad alto livello, nel programma oggetto o *eseguibile*, può essere effettuata secondo una delle due ben note tecniche, la *compilazione* e l'*interpretazione*, o loro combinazioni. In entrambi i casi, il punto di partenza è la sequenza di comandi, costituenti il programma scritto nel linguaggio ad alto livello, operanti su determinati insiemi di dati. Un compilatore o un interprete sono essi stessi dei programmi, già disponibili in forma eseguibile, che accettano come dato d'ingresso il programma sorgente e producono come dato di uscita il programma eseguibile.

In generale nella traduzione da parte di un interprete, poiché questa avviene passo per passo, non viene considerato il contesto in cui si trova il comando da interpretare a ogni passo, applicando sempre *la stessa regola di traduzione* per uno stesso comando. Invece, la traduzione da parte di un compilatore prende in considerazione una sequenza, più o meno ampia, in cui si trova inserito un comando e, per uno stesso comando, sceglie tra *regole di traduzione diverse* allo scopo di ottimizzare le prestazioni.

1.2 Strutturazione a moduli

In un sistema a qualunque livello, *una funzionalità complessa è ottenuta mediante composizione di funzionalità più semplici*: ognuna di queste, affidata a un modulo (autonomo e sequenziale), è specializzata verso determinati obiettivi, ma la cooperazione tra moduli rende possibile l'implementazione della funzionalità complessiva del sistema.

1.2.1 Modello di calcolatore general - purpose a programma memorizzato

Modello Von Neumann: il programma assembler da eseguire risiede in una memoria accessibile al Processore, detta *Memoria Principale* del calcolatore. Questo comporta che le *istruzioni del programma assembler*, rappresentate in binario, siano *memorizzate in locazioni di memoria*, che possono essere lette dal Processore nel giusto ordine: *è il Processore stesso (il suo microprogramma) che reperisce le istruzioni dalla memoria e provvede alla loro esecuzione (interpretazione).*

Capitolo 2

Rappresentazione Binaria e Strutture di Calcolo

2.1 Rappresentazione binaria delle informazioni

2.1.1 Conversione decimale-binario e binario-decimale

La conversione da binario a decimale si effettua, nel modo più semplice, applicando lo sviluppo polinomiale. La conversione da decimale in binario può essere fatta in due modi:

1. dal numero decimale viene sottratta la massima potenza di 2 minore del numero stesso e lo stesso processo è ripetuto sulla differenza risultante. Una volta decomposto il numero in potenze di due, si ottiene il numero binario ponendo 1 nelle posizioni di bit corrispondenti alla potenza di due usata nella decomposizione e 0 nelle restanti posizioni;

2. dividere ripetutamente il numero decimale per 2, considerando i resti, finché si ottiene come quoziente il numero 0. Il numero binario è dato dalla stringa dei resti in ordine inverso rispetto a come sono stati ottenuti.

2.1.2 Numeri binari con segno

Per la rappresentazione dei numeri con segno sono usati principalmente tre metodi. In tutti, il bit più significativo è (anche) il bit del segno: $0 = +$, $1 = -$.

- **Modulo e segno**: i bit restanti, tolto il bit del segno, rappresentano il valore assoluto (modulo) del numero.
- **Complemento a uno**: se il numero è positivo, il numero in complemento a uno è uguale a quello modulo e segno, mentre, se il numero è negativo, si ottiene complementando bit a bit la rappresentazione modulo e segno.
- **Complemento a due**: se il numero è positivo, il numero in complemento a due è uguale a quello del modulo e segno, mentre, se il numero è negativo, si ottiene sommando 1 alla rappresentazione in complemento a uno.

Sia il sistema modulo e segno che quello del complemento a uno hanno due rappresentazioni possibili per il valore zero, cioè $+0$ e -0. Questa situazione è indesiderata. *Il sistema del complemento a due non ha questo problema* perché il complemento a due di $+0$ è ancora $+$. Anche questo sistema presenta una singolarità: il numero costituito da un 1 seguito da tutti 0 è il complemento di sé stesso e questo fatto rende *l'intervallo dei numeri positivi asimmetrico rispetto a quelli dei negativi*; esiste cioè un numero negativo senza il corrispondente controvale positivo.

Nell'aritmetica in complemento a due il riporto generato dall'addizione sul bit più a sinistra è semplicemente scartato. Se due addenti sono di segno opposto, non può capitare un errore di traboccamento (overflow), mentre se sono di ugual segno e il risultato è di segno opposto significa che si è avuto un errore di overflow e il risultato non è corretto. L'errore di overflow può capitare se e solo se il riporto che arriva al bit di segno differisce dal riporto che esce dal bit del segno. Normalmente in uscita da un addizionatore si ha sia il riporto sul bit del segno che uno speciale bit di overflow.

2.2 Sull'utilizzazione della rappresentazione

2.2.1 Proprietà fondamentali

Una proprietà fondamentale dei numeri binari è la seguente:

- Con k bit si possono rappresentare tutti i numeri naturali che vanno da 0 a $2^k - 1$, estremi inclusi.

Altrettanto fondamentale è la seguente proprietà:

- Sia un insieme di m oggetti dello stesso tipo. Vogliamo denotare ogni oggetto con un numero naturale distinto, cioè un ***identificatore unico*** dell'oggetto, rappresentato in binario. Il minimo numero di bit necessario per rappresentare un qualunque identificatore è $n = \lceil lg_2 m \rceil$.

2.2.2 Alcune proprietà utili

a) Quoziente e resto della divisione per potenze di due

Siano A e B due numeri binari corrispondenti a numeri naturali, e sia $B = 2^h$. In questa ipotesi, è molto facile il calcolo del quoziente della divisione intera e del resto (modulo):

- il valore di A%B (A *mod* B) è dato dagli h bit meno significativi di A,
- il valore di A/B (A *div* B) è dato dai rimanenti bit più significativi di A.

Come caso particolare notevole: un numero binario *pari* (*dispari*) ha il bit meno significativo uguale a 0 (1).

b) Operazioni di traslazione(shift)

Dato un numero binario naturale A, la sua traslazione destra (*shift destro*) di 1 bit $sh^1_R(A)$ è il numero naturale che si ottiene spostando tutti i suoi bit a destra di una posizione e inserendo il valore 0 nel bit più significativo; il vecchio bit meno significativo viene perso.

Dato un numero binario naturale A, la sua traslazione sinistra (*shift sinistro*) di 1 bit $sh^1_L(A)$ è il numero naturale che si ottiene spostando tutti i suoi bit a sinistra di una posizione e inserendo il valore 0 nel bit meno significativo; il vecchio bit più significativo viene perso.

Valgono le seguenti proprietà, facilmente dimostrabili:

1. $sh^k_R(A) = A/2^k$
2. $sh^k_L(A) = A \cdot 2^k \bmod 2^n$

2.3 Strutture di calcolo, tipi di dato e memorie

2.3.1 Parole d'informazione

Le informazioni a livello di macchina firmware e di macchina assembler sono rappresentate come *stringhe di bit*, che possono avere il significato più vario a

seconda dell'uso che intendiamo farne. In generale, useremo il termine ***parola*** per denotare una stringa di bit significativa.

2.3.2 Registri

Fondamentale per la strutturazione a livello firmware e a livello assembler è il registro, che funge da *componente con memoria* (*con stato*).

Un registro di un bit ha il compito di memorizzare, per tutto il tempo che si ritiene necessario, l'informazione presente sull'ingresso *in* e di rendere disponibile tale informazione (*contenuto*) sull'uscita *out*. Per scrivere (memorizzare) il valore *in*, è necessario che sia presente un *segnale di controllo*: la scrittura avviene se $enable = 1$. Finché $enable = 0$, successivi valori che si presentino su *in* vengono perduti (non memorizzati), mentre appena $enable = 1$ il valore che in quel momento è presente sull'ingresso *in* viene memorizzato (diviene il *contenuto* del registro) e comparirà sull'uscita *out*. Il valore su *out* può sempre essere letto ed elaborato (un valore è *sempre* presente, non occorre abilitazione per utilizzarlo in lettura).

Nella trattazione delle reti logiche e delle unità di elaborazione, faremo uso di *registri impulsati* nei quali il segnale *enable* è messo in `AND` con un segnale, detto *clock*, impulsivo e periodico. Il periodo, cioè l'intervallo tra l'inizio di due impulsi di clock consecutivi, è detto ***ciclo di clock***. La scrittura nel registro avviene in corrispondenza di un impulso di clock a condizione che contemporaneamente $enable = 1$. Questa caratteristica permetterà un funzionamento sincrono delle reti (Parte Operativa e Parte Controllo) costituenti ogni unità di elaborazione.

2.3.3 Funzioni e reti combinatorie

A livello hardware implementeremo funzioni "pure" (computazioni senza stato: per ogni valore dell'ingresso sia ha sempre e un solo valore dell'uscita) mediante opportune strutture dette ***reti combinatorie***, ottenute a loro volta come composizioni di componenti più elementari. Ad esempio una funzione molto utilizzata è la **ALU**: si tratta di **reti di calcolo multifunzione**. A seconda del valore di alcune variabili d'ingresso secondarie (*variabili di controllo*), il valore dell'uscita è dato dall'applicazione agli ingressi primari di una ben specifica funzione: addizione, sottrazione, confronto, shift, `AND` bit a bit, `OR` bit a bit, ecc.

Capitolo 3

Il Livello Hardware: Reti Logiche

3.1 Reti combinatorie

Una rete combinatoria è una rete logica che ha n ingressi binari $x_1, \ldots, x_n$ ed m uscite binarie $z_1, \ldots, z_n$. A ciascuna combinazione dei valori degli ingressi corrisponde una e una sola combinazione dei valori delle uscite: la corrispondenza definisce la funzione implementata dalla rete combinatoria.

$x_1, \ldots, x_n$ e $z_1, \ldots, z_n$ sono dette ***variabili logiche***, di ingresso e di uscita rispettivamente. Tutte le combinazioni possibili delle variabili logiche sono dette ***stati***, di ingresso (in numero di 2_n) e di uscita (in numero di 2_m) rispettivamente.

3.1.1 Elementi di algebra booleana della commutazione

L'algebra della commutazione è un sistema algebrico in cui ogni variabile può assumere uno solo tra due valori, 0 e 1, nel quale sono applicate alle variabili le operazioni binarie di *moltiplicazione logica* e *somma logica* e l'operazione unaria di *complementazione* o *negazione*.

1.	*Complementazione:*	$A + \bar{A} = 1$	$A\bar{A} = 0$
2.	*Involuzione:*	$\bar{\bar{A}} = A$	
3.	*Potenza identica:*	$A + A = 1$	$AA = A$
4.	*Unione e intersezione:*	$\begin{cases} A + 0 = A \\ A + 1 = 1 \end{cases}$	$\begin{cases} A1 = A \\ A0 = 0 \end{cases}$
5.	*Proprietà commutativa:*	$A + B = B + A$	$A + B = B + A$
6.	*Proprietà associativa:*	$A + (B + C) = (A + B) + C$	$A(BC) = (AB)C$
7.	*Proprietà distributiva:*	$A(B + C) = AB + AC$	$A + (BC) = (A + B)(A + C)$
8.	*Teorema di De Morgan:*	$\overline{A + B} = \bar{A}\bar{B}$	$\overline{AB} = \bar{A} + \bar{B}$

Proprietà notevoli

Una funzione logica può essere rappresentata sia da una **tabella di verità**, sia un'**espressione algebrica**. Tuttavia, *mentre c'è una sola tabella per ogni funzione, vi sono molte espressioni che possono rappresentare una stessa funzione*. Due espressioni che rappresentano la stessa funzione sono chiamate *equivalenti*.

Chiamiamo *lettera* una variabile affermata o complementata e *termine* un prodotto di lettere (termine prodotto) una somma di lettere (termine somma) che compare in un'espressione. Chiamiamo *mintermine* un termine prodotto e *maxtermine* un termine somma che contengono un numero di lettere di variabili distinte uguale al numero n delle variabili della funzione rappresentata dall'espressione.

Indichiamo con SP e PS un'espressione in Somma di termini Prodotto (o Somma di Prodotti) e Prodotti di termini Somma (o Prodotti di Somma), rispettivamente. Chiamiamo infine *forma normale* la forma di un'espressione SP.

Chiameremo **forma canonica** di una funzione in n variabili l'espressione in SP o Ps in cui ogni termine è un mintermine o un maxtermine.

3.1.2 Definizione di reti combinatorie, loro formalizzazione e comportamento

Una volta ricavata l'espressione logica dalla tabella di verità, è immediato realizzare lo *schema logico* utilizzando *i componenti hardware elementari*, detti anche ***porte logiche***, `AND`, `OR`, `NOT`.

Partendo da un'espressione logica in forma SP, la rete combinatoria ottenuta è "***a due livelli di logica***", con questo intendendo che esiste un primo livello di porte AND operanti in parallelo, seguite da una porta OR finale.

3.1.3 Specifica di reti combinatorie

Le seguenti sono le specifiche di alcune reti combinatorie che supporremo standard, o primitive, cioè componenti che è possibile usare (al pari delle porte AND, OR, NOT) come blocchi basici nella progettazione di strutture più complesse:

- ***confrontatore*** ($\oplus$) a due ingressi x, y ed una uscita z (z è vero se x, y sono diversi): $z = x \oplus y =$ **not** $(x = y) \rightarrow$ *rete a due livelli di logica.*
- ***commutatore*** (K) a due ingressi primari x, y, un ingresso secondario di controllo α, ed una uscita z (z assume il valore di x o y a seconda che α sia falso o vero rispettivamente): $z =$ **if not** α **then** x **else** $y \rightarrow$ *rete a due livelli di logica*
- ***selezionatore***, o selettore (S), con un ingresso primario x, un ingresso secondario o di controllo α, e due uscite z_1, z_2 (se α è falso z_1 assume il valore di x e z_2 vale falso (0), se α è vero z_1 vale falso (0) e z_2 assume il valore di x): **if not** α **then** $(z_1 = x, z_2 = 0)$ **else** $(z_1 = 0, z_2 = x) \rightarrow$ *rete a un solo livello di logica.*

3.1.4 Procedimento di sintesi

Una volta data la specifica di una rete (di una funzione logica), nel *caso più generale* il procedimento di sintesi della rete stessa è il seguente:

1. traduzione della specifica nella tabella di verità, per enumerazione di tutti i casi (2_n, per n variabili di ingresso);
2. per ogni variabile di uscita, scrittura della espressione logica in forma canonica SP (i termini `AND` corrispondono agli stati d'ingresso per i quali la variabile di uscita assume valore vero);
3. eventuale riduzione delle espressioni logiche;
4. traduzione di ogni espressione logica in uno schema di rete a due livelli di logica.

Questo procedimento ha complessità $O(2^n)$, ed è quindi praticamente applicabile solo nei casi in cui il numero degli ingressi n assuma un valore "contenuto".

3.1.5 Comportamento temporale delle reti combinatorie

Ogni rete reale è caratterizzata da un *ritardo* t_r, necessario affinché, in seguito ad una variazione dello stato d'ingresso, si produca la corrispondente variazione dello stato di uscita. Solo dopo questo tempo si dice che la rete si è ***stabilizzata***: durante l'intervallo di durata t_r, detto appunto ***ritardo di stabilizzazione***, il comportamento della rete, in ogni suo punto, è impredicibile e, in particolare, il valore delle uscite non è significativo.

Nel seguito supporremo che

1. Le variazioni dello stato d'ingresso siano sincronizzate, cioè avvengano ad istanti discreti ed equidistanti, di una sequenza temporale;

2. gli istanti di tale sequenza siano distanziati di un periodo $\geq t_r$.

Per una porta logica, indichiamo con t_p il ritardo di stabilizzazione. Nel seguito supporremo che le porte `NOT` abbiano ritardo nullo, o meglio inglobato nel ritardo delle porte `AND`/`OR` connesse alle uscite delle porte `NOT`.

È importante ricordare che t_p è il massimo valore che può assumere il ritardo di stabilizzazione di una porta.

Dalle assunzioni 1), 2) deriva che il massimo ritardo di stabilizzazione di una rete a L *livelli di logica* è dato da $t_r = Lt_p$. *Il valore di* t_p *dipende dal numero d'ingressi* n *della porta*. La funzione $tp(n)$ ha un andamento monotono crescente; la crescita è relativamente lenta per $n \leq n_0$, tale da poter considerare t_p costante, mentre è molto più rapida per $n > n_0$. Valori tipici di n_0 allo stato attuale variano da 4 a 8. Il valore t_p che si associa a una porta logica è quello per $n \leq n_0$.

La conseguenza di questa caratteristica è che, ove l'espressione logica di una variabile di uscita contenga un termine `AND` con più di n_0 variabili di ingresso, la porta `AND` della rete deve essere decomposta in più porte secondo una struttura ad albero, ognuna con al più n_0 ingressi. Lo stesso vale per una porta `OR` finale.

3.1.6 Reti combinatorie operanti su parola

Capita spesso che la funzione che definisce la rete è applicata a parole di N bit (ad esempio, 32) invece che a semplici variabili booleane. In questi casi, la complessità del procedimento di sintesi delle reti combinatorie impedisce di ricorre al metodo generale basato sulla tabella di verità. Si cerca invece di *comporre reti a 1 bit, del tipo corrispondente, per ottenere la rete a N bit*; ad

esempio, realizzare un commutatore ad N bit utilizzando commutatori ad 1 bit.

- Per alcune reti la composizione è ***parallela***: la rete a N bit è composta da N reti a 1 bit tutte indipendenti, cioè nessuna utilizza in ingresso le uscite di altre. Il ritardo della rete a N bit è dunque uguale a quello della rete a 1 bit. È il caso del *commutatore*, *selezionatore*, *confrontatore*.

- Per altre reti, la composizione può essere ***in cascata***: la rete a N bit è composta da N reti ad 1 bit tali che le uscite dell'i-esima costituiscono ingressi della $(i+1)$-esima. Il ritardo della rete a N bit è ora N volte quello della rete a 1 bit. È il caso dell'*addizionatore*, a causa della propagazione del riporto.

- Un altro caso molto significativo è quello della composizione ***ad albero***: la rete a N bit è composta da $\log_k N$ reti a 1 bit disposte ad albero di arietà k.

3.2 Reti sequenziali

3.2.1 Definizione di reti sequenziali, loro formalizzazione e comportamento

Un ***automa a stati finiti*** è una macchina caratterizzata da

- n variabili logiche di ingresso, e corrispondentemente $h = 2^n$ ***stati d'ingresso*** $X_1, \ldots, X_h$;

- m variabili logiche di uscita, e corrispondentemente $k = 2^m$ ***stati d'uscita*** $Z_1, \ldots, Z_k$;

- r variabili logiche dello stato interno, e corrispondentemente $p = 2^r$ ***stati interni*** $S_1, \ldots, S_p$;

- una ***funzione di transizione dello stato interno***:

$$\sigma : \ X \times S \to S$$

  Tale funzione definisce la trasformazione dello stato interno dal valore ***presente*** al valore ***successivo*** in corrispondenza del valore dello stato d'ingresso;

- una ***funzione delle uscite***:

$$\omega : \ X \times S \to Z$$

  Tale funzione definisce la trasformazione dello stato di uscita in corrispondenza del valore dello stato d'ingresso e dello stato interno *presente*.

Una rete logica sequenziale implementa, a livello hardware, un automa a stati finiti.

In un *modello strutturale ideale* la rete sequenziale di tipo ***sincrono***, le variazione degli stati avvengono in corrispondenza degli istanti di una sequenza temporale discreta $t_0, t_1, \ldots, t_n, \ldots$ di intervallo (periodo) costante $\Delta = t_{i+1} - t_i$.

Si possono definire due distinti ***modelli matematici di automa***, e quindi di rete sequenziale: il modello di *Mealy* e il modello di *Moore*.

- In entrambi i modelli, considerando il comportamento dell'automa al tempo *t*, lo stato interno successivo *S(t+1)* dipende tanto dallo stato d'ingresso al tempo *t*, *X(t)*, quanto dallo stato interno presente, *S(t)*:

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

- Nel modello di Mealy, lo stato di uscita al tempo *t*, *Z(t)*, dipende tanto dallo stato d'ingresso al tempo *t*, *X(t)*, quando dallo stato interno presente, *S(t)*:

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

- Nel modello di Moore, *Z(t)* dipende solo da *S(t)*:

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

Nel modello di Moore la dipendenza tra stati di uscite e stati d'ingresso è quindi espressa da:

$$Z(t) = \omega(S(t)) = \omega(\sigma(X(t-1), S(t-1))) = \omega_1((X(t-1), S(t-1)))$$

Ne discende che, dati due automi di Mealy e di Moore equivalenti, con stati interni iniziali equivalenti e per la stessa sequenza d'ingresso si ha che la sequenza di uscite dell'automa di Moore è ritardata, rispetto a quella dell'automa di Mealy, di un intervallo Δ della sequenza temporale.

In generale, il numero di stati interni dell'automa di Moore è maggiore o uguale del numero di stati interni dell'automa di Mealy equivalente.

3.2.2 Reti sequenziali sincrone LLC

Il modello strutturale di rete sequenziale sincrona reale comprende le due reti combinatorie reali ω (funzione delle uscite) e σ (funzione di transizione dello stato interno), e le richiusure dello stato interno sono realizzate con k registri in parallelo impulsati dallo stesso segnale di clock (registro di k bit).

Il modello è detto LLC (**L**evel input, **L**evel output, **C**locked), a significare che i segnali su cui si applicano le funzioni ω e σ sono ancora livelli, mentre l'unico segnale impulsivo è quello per la sincronizzazione dei (per provocare la scrittura dei) registri.

Il periodo τ dell'impulso è detto **ciclo di clock** della rete sequenziale. Per tenere conto della durata δ dell'impulso, e quindi per evitare il fenomeno dello stato metastabile dei registri, si ha che il *ciclo di clock della rete va determinato come*

$$\tau \geq t_r + \delta$$

La durata δ dell'impulso di clock può essere assunta uguale a t_p.

3.2.3 Reti sequenziali realizzate con componenti standard

Nella parte dedicata al firmware saremo interessati a sintetizzare, per ogni unità di elaborazione, due specifiche reti sequenziali: la Parte Controllo e la Parte Operativa, per ognuna delle quali adotteremo due diverse metodologie.

La *Parte Controllo* ha spesso un numero relativamente basso di stati (interno, d'ingresso, di uscita); inoltre, dal microprogramma è possibile ricavare formalmente una sua descrizione tipo grafo di stato. Di conseguenza, la sua sintesi è effettuata con il metodo classico visto finora; le reti combinatorie ω e σ sono realizzate mediante porte `AND`, `OR`, `NOT`.

La *Parte Operativa* presenta, anche per le unità più semplici, un numero di stati (interni, d'ingresso, d'uscita) relativamente molto grande; basti pensare che i dati su cui si opera sono tipicamente a parola a N bit: per $N = 32$ il numero di stati è sull'ordine di molti miliardi. In questo caso l'approccio è completamente diverso e si basa sulla composizione di componenti standard a partire dalla specifica delle operazioni elementari delegate alla Parte Operativa, specifica ricavata formalmente dal microprogramma.

3.3 Componente logico memoria

Nella realizzazione di unità di elaborazione è spesso conveniente, o necessario, fare uso di un ulteriore componente logico "standard": il componente

memoria.

3.3.1 Realizzazione logica

In versione ***RAM*** (lettura-scrittura) un componente logico memoria è definito come un *array unidimensionale M di registri* su cui sono definite le seguenti operazioni fondamentali:

- lettura sull'uscita *out* del contenuto della cella d'indirizzo *i*: $out = \texttt{M[i]}$;
- scrittura del valore presente sull'ingresso *in* nella locazione di indirizzo i: $\texttt{M[i]} = in$.

L'operazione di lettura è realizzata dal *commutatore* K di uscita, i cui ingressi primari sono le uscite dei registri e i cui ingressi secondari sono i bit dell'indirizzo.

L'operazione di scrittura, invece, è realizzata dal *selezionatore* S d'ingresso, avente anch'esso come ingressi secondari i bit dell'indirizzo. Come ingresso primario ha il segnale di controllo β per l'abilitazione alla scrittura. Inoltre, l'ingresso *in* viene collegato a tutti gli ingressi dei registri; di conseguenza, se $\beta = 1$, il valore di *in* viene scritto solo nel registro indirizzato.

Nel caso di una memoria ***ROM***, è ovviamente presente solo il commutatore di uscita.

Si possono avere RAM *a più ingressi e più uscite* utilizzando il numero corrispondenti di selezionatori e commutatori.

3.3.2 Realizzazione fisica e ritardi

Oltre alle modalità in cui verranno realizzati il commutatore e il selezionatore per indirizzare le celle, il tempo di accesso è influenzato in modo significativo dal *tipo* di tecnologia elettronica utilizzato:

- le memorie RAM più economiche sono quelle così dette ***dinamiche*** (DRAM), in quanto, per ragioni di potenza del segnale relativo al contenuto delle celle, ogni cella necessita di un "rinfresco" periodico (la cella va letta e quindi riscritta con lo stesso valore letto);
- le memorie ***statiche*** (SRAM), che non necessitano di rinfresco, presentano tempi di accesso decisamente più bassi, ma a parità di generazione tecnologica, sono anche assai meno "dense" delle dinamiche e quindi caratterizzate da una minore capacità per chip.

3.3.3 Numero massimi d'ingressi per porta nelle reti combinatorie

Ricordiamo che, in generale, per una porta logica AND/OR è fissato un *numero massimo d'ingressi*, N_0, ad esempio $N_0 = 8$. Il valore del massimo ritardo di stabilizzazione della porta, t_p, vale per un numero di ingressi $N \leq N_0$, in quanto per $N > N_0$ il ritardo di stabilizzazione aumenterebbe linearmente con forte pendenza. In pratica, non esistono porte con $N > N_0$. Di questa caratteristica tecnologica occorre tenere conto nella realizzazione di una *qualunque rete combinatoria*, in particolare, nella realizzazione di commutatori (selezionatori) con un elevato numero d'ingressi, come si ha anche (ma non solo) nell'implementazione dei componenti logici memoria.

Dal punto di vista tecnologico, occorre tener conto del vincolo che, per ogni porta, deve essere $N \leq N_0$. Se questa condizione non si verifica, ***occorre sostituire la singola porta con una struttura ad albero di arietà*** N_0 che, di tutte le possibili realizzazioni, è quella che garantisce il minimo ritardo di stabilizzazione.

In generale, quindi, *il ritardo di stabilizzazione comportato dalla realizzazione di un termine* AND/OR *vari in modo logaritmico con il numero delle sue variabili d'ingresso.*

3.3.4 Memorie realizzate a partire da memorie preesistenti

Di regola, le memorie di grande capacità sono realizzate a partire da componenti logici base dati, che vanno combinati opportunamente in quella che prende il nome di **memoria modulare**.

3.3.5 Organizzazione di memoria modulare

La memoria modulare **sequenziale** è caratterizzata dalla distribuzione degli indirizzi sequenzialmente all'interno di un modulo: quando si esaurisce la capacità di un modulo, si passa a distribuire gli indirizzi sequenzialmente nel modulo successivo.

$$\textit{identificatore del modulo} = \textit{indirizzo}/C$$
$$\textit{indirizzo interno del modulo} = \textit{indirizzo}\%C$$

(dove C è la capacità del singolo modulo).
L'organizzazione alternativa di memoria modulare è detta organizzazione **interallacciata**, per la quale si ha

$$identificatore\ del\ modulo = indirizzo/m$$
$$indirizzo\ interno\ del\ modulo = indirizzo\%m$$

Dove m è il numero dei moduli. Ciò significa che, nella memoria modulare interallacciata, m parole aventi indirizzi consecutivi sono allocate in altrettanti moduli di memoria distinti e consecutivi.

Capitolo 4

Livello Firmware

4.1 Caratteristiche di un sistema a livello firmware

Un *sistema di elaborazione* a livello firmware è costituito da un certo numero di *unità di elaborazione* tra loro interagenti, $U_1, \ldots, U_n$.

A ogni unità di elaborazione è affidato un certo sottoinsieme delle funzionalità dell'intero sistema. Attraverso l'interazione tra unità è realizzato il compito che globalmente s'intende affidare all'intero sistema. Ogni unità è specializzata verso uno specifico compito, ma attraverso l'interazione tra esse si realizza un sistema avente la caratteristica di *essere generale nei confronti del supporto fornito ai livelli superiori del sistema.* In particolare, il funzionamento di ogni unità e l'interazione tra le varie unità realizzano l'*interpretazione* di qualunque programma espresso con il formalismo che definisce il livello assembler del sistema.

Essendo un modulo di elaborazione, un'unità svolge il proprio specifico compito in modo:

- autonomo: *l'unità è capace di controllare la propria elaborazione in modo del tutto indipendente, pur ricavando informazioni sulle funzionalità da eseguire e valori dei dati attraverso l'interazione con altre unità*;
- sequenziale: *il fondamento dell'unità è descritto da un programma sequenziale. Chiameremo microprogramma la descrizione del funzionamento dell'unità, e microlinguaggio il linguaggio di programmazione sequenziale con cui esprimere il microprogramma.*

4.1.1 Modello Parte Controllo - Parte Operativa

Il microprogramma di un'unità di elaborazione è a sua volta implementato dal *livello hardware* sottostante. Questa interpretazione è realizzata da due reti sequenziali LLC tra loro interagenti, dette **Parte Controllo** (PC) e **Parte Operativa** (PO).

PO provvede all'esecuzione delle operazioni dai comandi del microlinguaggio, o **microistruzioni**, disponendo delle tipiche risorse strutturali messe a disposizioni dal livello hardware: *commutatori*, *selezionatori*, *rete di calcolo* mono-funzione o multi-funzione (ALU), *registri*.

PC provvede:

- al *controllo della sequenzializzazione* delle microistruzioni, allo scopo utilizzando i valori delle **variabili di condizionamento** relative allo stato di PO;
- a *ordinare a PO l'esecuzione* delle operazioni di ogni microistruzione mediante i valori delle **variabili di controllo** α e β.

PC e PO sono *reti sequenziali LLC* impulsate dallo stesso segnale di clock, e quindi aventi lo stesso ciclo di clock:

- questo, detto ***ciclo di clock*** dell'unità è determinato in modo tale da permettere la stabilizzazione di entrambe le reti per l'esecuzione di una qualsiasi microistruzione.

Poiché la struttura PC-PO rappresenta l'interprete hardware del microlinguaggio del livello firmware, *il modello di programmazione del livello firmware è dunque* ***sincrono***:

- *ogni microistruzione è eseguita in tempo costante uguale al ciclo di clock.*

4.1.2 Il procedimento di progettazione delle unità

I passi essenziali nel procedimento formale di progettazione di un'unità sono i seguenti:

1. Specifica delle operazioni esterne affidate all'unità
2. Scrittura del microprogramma che interpreta le operazioni esterne
3. Progetto PO

4. Progetto PC
5. Valutazione del ciclo di clock dell'unità
6. Valutazione tempo medio di elaborazione

4.2 Formalizzazione del procedimento

A partire dalla specifica delle operazioni esterne, il procedimento di progettazione ha lo scopo di realizzare la struttura dell'unità di elaborazione secondo il modello PC-PO, e di valutarne le prestazioni.

4.2.1 Microlinguaggio

La formalizzazione del procedimento è basata sulla scrittura del *microprogramma eseguibile*, ogni passo del quale (*microistruzione*) è eseguito esattamente in un ciclo di clock.

4.2.2 Ottimizzazione delle micro operazioni

Esistono diversi modi per scrivere microprogrammi efficienti:

- Eliminazione delle *nop*: ogni volta che si esegue una nop, agli effetti del tempo di elaborazione viene perduto un ciclo di clock;
- Parallelismo delle micro operazioni: trasformare una computazione sequenziale nella computazione parallela equivalente.

Formalmente valgono le seguenti **condizioni di Bernstein**:

- $W(\mu - op_1) \cap R(\mu - op_2) = \emptyset$
- $R(\mu - op_1) \cap W(\mu - op_2) = \emptyset$
- $W(\mu - op_1) \cap W(\mu - op_2) = \emptyset$

La terza condizione vale solo in un modello asincrono di computazione, nel quale non si fanno ipotesi sulla durata delle operazioni e sui loro istanti di inizio.

4.2.3 Struttura di PO

La struttura di PO, vista come rete sequenziale di Moore, è ricavata formalmente dal microprogramma.

1. Si individuano tante classi di operazioni elementari per quanti sono i registri destinazione (che compaiono a destra dell'operatore $\rightarrow$). Ciò permette di individuare i possibili ingressi di ogni registro.

2. Si ricava una sottostruttura indipendente per ogni registro, con la propria variabile di controllo β; nel caso in cui gli ingressi possibili IN siano in numero $r > 1$, sull'ingresso del registro viene inserito un commutatore, comandato da $\lceil \log_2 r \rceil$ variabili di controllo α e i cui ingressi principali sono quelli di IN.

3. Si individuano tante classi di operazioni elementari per quante sono le reti logiche dagli operatori che compaiono nelle operazioni elementari.

4. Si ricava una struttura indipendente per ogni rete logica: se si tratta di una rete multifunzione, come una ALU, con r funzioni, occorrono $\lceil \log_2 r \rceil$ variabili di controllo α per comandare la scelta della funzione; per ogni ingresso principale, se è prevista più di una sorgente, si inserisce un commutatore pilotato da opportune variabili di controllo α.

5. Si implementano le variabili di condizionamento, come funzioni di uscite di registri o uscite di reti combinatorie

4.2.4 Struttura di PC

La struttura di PC, vista come rete sequenziale di Mealy, è ricavata formalmente dal microprogramma mediante il seguente procedimento.

1. Gli stati interni corrispondono biunivocamente alle etichette delle microistruzioni del microprogramma. Se m è il numero degli stati interni, il registro di stato del controllo è di s bit con $s = \lceil \log_2 m \rceil$.

2. Gli stati d'ingresso corrispondono biunivocamente alle possibili combinazioni di variabili di condizionamento per formare le condizioni logiche del microprogramma

3. Gli stati di uscita corrispondono biunivocamente alle possibili combinazioni di segnali di controllo α e β necessari a eseguire tutte le micro-operazioni del microprogramma.

4. La tabella di verità della funzione delle uscite di ω_{PC} e della funzione di transizione dello stato interno σ_{PC} viene ricavata in base alle tre corrispondenze suddette e alla struttura del microprogramma. Da questa tabella è possibile sintetizzare PC con parte combinatoria a due o più livelli di logica.

Si osservi che *tutti i segnali β devono sempre essere specificati per ogni possibile combinazione*, mentre i segnali α potranno eventualmente essere non specificati per qualche combinazione.

4.2.5 Ciclo di clock

A differenza di una rete sequenziale isolata, per un'unità di elaborazione PC-PO occorre ricavare la *condizione di stabilizzazione di entrambe le reti sequenziali PC e PO*. Il ciclo di clock non può dunque concludersi prima che siano stabili le funzioni di transizione di PC e di PO, cioè prima che siano stabili, rispettivamente, gli ingressi del registro di stato del controllo e di tutti i registri di PO.

Poiché PO è una rete di Moore, dopo un tempo T_ωPO, sono stabili le variabili di condizionamento. Solo a questo punto, poiché PC è una rete di Mealy, può iniziare, per l'ultima volta all'interno di questo ciclo di clock, la stabilizzazione delle funzioni ω_{PC} e σ_{PC} in parallelo. Una volta che ω_{PC} si è stabilizzata definitivamente, dopo un intervallo T_ωPC, può quindi iniziare, per l'ultima volta all'interno di questo ciclo di clock, la stabilizzazione della funzione σ_{PO}; il ciclo di clock si può concludere quando si sono stabilizzate definitivamente tanto la σ_{PC} quanto la σ_{PO}.

Di conseguenza, la lunghezza del ciclo di clock è data da:

$$\tau = T_{\omega_{PO}} + \max\{T_{\omega_{PC}} + T_{\sigma_{PO}}, T_{\sigma_{PC}}\} + \delta$$

Mentre il tempo medio di elaborazione di una unità è calcolato come

$$T = k\tau$$

Dove k è il *numero medio di cicli di clock necessari a eseguire la generica operazione esterna*.

In generale, in un microprogramma sono presenti più sottosequenze di microistruzioni eseguite con una certa probabilità p_i dove:

$$\sum_{i=0}^{n-1} p_i = 1$$

Di ogni sottosequenza si calcola il numero di cicli di clock k, necessario a eseguirla e quindi si ricava il valore di T come media pesata:

$$T = \tau \times \sum_{i=0}^{n-1} p_i k_i$$

Quando non siano note le p_i si assume che tutte le sequenze siano equiprobabili.

*La **banda di elaborazione** è definita come il numero medio di operazioni esterne che l'unità può eseguire nell'unità di tempo*: $B = \frac{1}{T}$

4.3 Comunicazioni a livello firmware

4.3.1 Comunicazioni tra unità

Nel modello a scambio di messaggi a livello firmware i canali di comunicazione sono implementati da supporti fisici (strutture d'interconnessione) costituiti da interfacce presenti nella Parte Operativa delle unità e da collegamenti tra le unità stesse.

Il canale di comunicazione tra due unità è implementato da tre componenti: interfaccia di uscita, collegamento fisico e interfaccia d'ingresso.

Le forme di comunicazioni possibili sono

- Simmetrica vs asimmetrica: nel caso simmetrico il canale di comunicazione ha un singolo mittente e un singolo destinatario, nel caso asimmetrico in ingresso il canale di comunicazione ha più mittenti e un singolo destinatario, nel caso asimmetrico in uscita il canale di comunicazione ha un singolo mittente un più destinatari.

- Asincrona vs sincrona: si dice che un canale (simmetrico) ha grado di asincronia k, con $k \geq 0$, se il mittente può inviare fino a k messaggi senza attendere che il destinatario ne abbia ricevuto uno o più. Nel caso che il mittente intenda inviare un $(k+1)$-esimo messaggio senza che il destinatario ne abbia ricevuto alcuno dei k precedenti, occorre che il mittente stesso attenda che il destinatario ne abbia ricevuto almeno uno. Il caso $k = 0$ è quello della comunicazione sincrona: per ogni messaggio, il mittente, prima di poter proseguire, deve attendere che il destinatario lo abbia ricevuto.

4.3.2 Protocollo mediante segnali di RDY e ACK

La trasmissione del messaggio (`MSG`) avviene attraverso un registro di uscita di U_1 (`OUT`) e un registro d'ingresso di U_2 (`IN`) connessi da un collegamento

fisico. Si tratta dei registri d'ingresso e di uscita con le caratteristiche tipiche di un'unità firmware. Una volta che U_1 ha scritto il valore di `MSG` in `OUT`, questo valore si propaga all'ingresso di `IN` di U_2, e sarà quindi scritto in tale registro al primo impulso di clock di U_2. Il valore di `MSG` è ora disponibile a U_2 come stato interno di PO.

La *sincronizzazione* è implementata facendo uso di due linee addizionali, di un bit ciascuna, detta ***linea di Ready*** (`RDY`) e ***linea di Acknowledgement*** (`ACK`):

- mediante la linea `RDY` U_1 fa sapere a U_2 che ha inviato un nuovo messaggio; essa ha dunque il significato di segnalare a U_2 la presenza di un nuovo messaggio;
- mediante la line `ACK` U_2 fa sapere U_1 che ha ricevuto il nuovo messaggio; essa ha dunque il significato di segnalare a U_1 la ricezione di un messaggio.

Poiché intendiamo implementare un protocollo per la comunicazione asincrona di grado $k = 1$, il significato della linea `ACK` è di far sapere a U_1 che è possibile inviare un nuovo messaggio.

4.3.3 Interfaccia a transizione di livello

I quattro elementi di memoria, `RDYOUT`, `ACKOUT`, `RDYIN`, `ACKIN`, sono detti ***indicatori d'interfaccia*** e sono così caratterizzati:

- sugli *indicatori d'interfaccia di uscita* (`RDYOUT` e `ACKIN`) è definita l'unica operazione chiamata ***set***: l'esecuzione di `set RDYOUT` da parte di U_1 ha come effetto finale il fatto che l'indicatore d'interfaccia d'ingresso di U_2 `RDYIN` assume l'uscita uguale a uno; analogamente, l'esecuzione di `set ACKIN` da parte di U_2 ha come effetto finale il fatto che l'uscita di `ACKOUT` in U_1 assume il valore uno;
- sugli *indicatori d'interfaccia d'ingresso* (`RDYIN` e `ACKOUT`) sono definite due operazioni. Una è il test del valore di uscita di tali indicatori. L'altra, chiamata ***reset***, ha come effetto di portare l'uscita degli indicatori stessi a zero: `reset RDYIN` porta a zero l'uscita di `RDYIN`, `reset ACKOUT` porta a zero l'uscita di `ACKOUT`.

Per quanto riguarda l'impatto del protocollo sulla struttura di PC e PO:

- tutti gli indicatori d'interfaccia fanno parte dello stato interno di PO;
- `RDYIN` e `ACKOUT` fanno parte delle variabili di condizionamento.

4.3.4 Altre forme di comunicazione

Comunicazione domanda e risposta: si consideri una unità U_1 che invia un messaggio a U_2 dopodiché attende un messaggio da U_2 stessa; tipicamente, si tratta di una situazione cliente-servente, in cui il cliente U_1 prima invia un messaggio al servente U_2 per chiedere un servizio, e quindi attende dal servente un messaggio di risposta. Anche se le interfacce tra U_1 e U_2 contengono gli indicatori di `ACK`, questi non servono nella comunicazione a domanda e risposta: U_1 non invierà un nuovo messaggio prima di aver ricevuto la risposta da U_2, quindi l'`ACK` è implicito nella risposta.

Comunicazione asincrona con grado di asincronia $k > 1$: in questo caso non esiste una soluzione basata su semplici interfacce nelle due unità comunicanti, ma è necessario interporre una terza unità (Unità Buffer) che implementi una coda FIFO a k posizioni.

Collegamenti a bus: il bus risulta composto da $n + 2$ linee, per dati di n bits. Tutti i bit di `RDY`, sia ingresso sia in uscita, di tutte le unità sono infatti connessi a una singola linea, così come tutti i bit di `ACK`, e tutti i bit facenti parte del messaggio. L'uscita del registro `OUT` è messa in `AND` con il segnale `RDYOUT` allo scopo di "ripulire" il bus dopo che il messaggio è stato ricevuto.

Capitolo 5

Il livello della Macchina Assembler

5.1 Modello a programma memorizzato

Come già introdotto, il modello di calcolatore adottato comunemente è quello di *Von Neumann* o a programma memorizzato, caratterizzato dal fatto che il linguaggio assembler è di tipo imperativo, quindi facente uso

1. del concetto di variabile, e della tecnica di assegnamento e riassegnamento di variabili,
2. del concetto di sequenzializzazione esplicita delle istruzioni, implementato da una particolare variabile detta contatore di istruzioni, IC (instruction counter).

Ogni istruzione assembler è rappresentata in binario secondo un certo formato dell'istruzione. Tipicamente, il formato dell'istruzione contiene elementi come:

- codice operativo dell'istruzione,
- informazioni per ricavare indirizzi di variabili o di altre istruzioni,
- eventuali costanti,
- eventuali opzioni legate all'ottimizzazione del programma e dell'architettura.

Al momento della loro utilizzazione durante l'esecuzione del programma, le variabili possono risiedere in:

- locazioni di memoria principale,
- registri generali.

Il linguaggio assembler deve prevedere un insieme di regole per indirizzare una locazione da parte di un'istruzione: tali regole saranno dette **modi di indirizzamento**.

I registri generali rappresentano, in un certo senso, un'estensione della memoria disponibile al programma. Essendo implementati, a livello firmware, nella Parte Operativa del processore, essi sono caratterizzati da un tempo di accesso molto basso, di diversi ordini di grandezza inferiore a quello necessario per leggere o scrivere parole residenti in memoria principale. Normalmente, i registri generali sono organizzati in una (piccola) memoria di registri; la sua capacità può variare da poche unità a qualche decina fino alle centinaia (noi supporremo che sia di 64 unità).

Il significato dell'aggettivo "*generali*" sta a indicare che l'utilizzo di tali registri non è specifico di una certa funzionalità. Un registro generale verrà usato per contenere il valore di una variabile riferita dal programma, ma anche un indirizzo di memoria o una quantità che viene utilizzata per calcolare un indirizzo di memoria, o altre eventuali informazioni manipolabili a livello assembler.

I registri generali sono, ovviamente, visibili anche a livello firmware secondo la specifica struttura che assumono nella Parte Operativa del processore.

In alcune macchine, il contatore istruzioni è, come i registri generali, *visibile a livello assembler* e su esso sono possibili tutte le operazioni consentite sui registri generali. In altre macchine, *IC è visibile solo a livello firmware*, nel senso che la sua manipolazione (modifica) è effettuata dal microprogramma del processore nell'interpretare ogni istruzione assembler. In questo secondo caso, esistono comunque istruzioni assembler (*istruzioni di salto*) il cui effetto finale è di modificare il contenuto di IC per fargli assumere un valore fuori sequenza, pur se IC non è riferito esplicitamente dall'istruzione; allo stesso modo, tutte le istruzioni non di salto modificano IC nel senso di incrementarlo per puntare all'istruzione immediatamente successiva in sequenza, ma, a maggior ragione, questo effetto è implicito e IC non è riferito esplicitamente dall'istruzione.

In ogni caso, *la semantica di un'istruzione* deve contenere sempre la manipolazione di IC; si ricorda infatti che la semantica di un comando deve fornire tutti gli elementi che permettano di definirne il supporto (interprete).

5.2 Compilazione ed esecuzione

5.2.1 Programmi e processi

I *moduli di elaborazione* a livello delle applicazioni e a livello del sistemo operativo sono i *processi*: programmi sequenziali con propria capacità di controllo, eseguibili con concorrentemente ad altri e cooperanti con essi.

Un qualunque programma *viene compilato in un processo* o, nel caso che il programma stesso sia parallelo, in una collezione di processi.

La fase di linking ha il compito di collegare a ogni processo applicativo i *servizi* di sistema necessari all'esecuzione del programma corrispondente e non visibili esplicitamente nel codice sorgente e di arricchire il processo con un insieme di *informazioni di utilità*, cioè informazioni che sono essenziali per l'esecuzione e la gestione del processo, che sono contenute in una *struttura dati* che chiameremo **descrittore di processo** (PCB, *Process Control Back*). Il PCB stesso fa parte delle informazioni di un processo.

5.2.2 Compilazione e caricamento

La compilazione di un programma è effettuata da un compilatore, il quale, prendendo in ingresso il programma sorgente, lo trasforma in un processo *rappresentato da un* ***file oggetto*** *binario*, o **codice eseguibile**, contenete tutte le informazioni necessarie per l'esecuzione del processo stesso.

Il procedimento di compilazione costa, operativamente, di più fasi:

- traduzione da linguaggio sergente in assembler simbolico;
- traduzione da assembler simbolico a codice binario;
- linking.

In base alle regole di compilazione legate alla struttura del linguaggio sorgente, il compilatore provvede a generare una lista di istruzioni assembler. Nel far questo, un importante compito del compilatore è quello:

- dell'allocazione statica della memoria virtuale e dei registri generali,
- dell'inizializzazione di una parte delle locazioni di memoria virtuale e registri generali.

Gli indirizzi generati dal processore, durante l'esecuzione di un processo, non sono direttamente indirizzi di memoria principale (*indirizzi fisici*), bensì

indirizzi logici, cioè indirizzi riferiti a un'*astrazione* della memoria del processo, detta **memoria virtuale**. Questa può essere vista come un array unidimensionale, con indici (indirizzi logici) a partire da zero fino al massimo necessario per rappresentare il programma o fino al massimo consentito dall'ampiezza dell'indirizzo logico in bit; per una macchina a 32 bit e con indirizzamento alla parola, la massima ampiezza della memoria virtuale di ogni programma è 4G parole.

L'insieme degli indirizzi logici di un processo è detto il suo **spazio logico di indirizzamento**.

Il codice eseguibile del processo, generato dal compilatore, è quindi riferito alla memoria virtuale. Il processore genera indirizzi logici sia per le istruzioni sia per i dati.

Quando un processo viene allocato (o riallocato) in memoria principale, viene stabilita una corrispondenza tra gli indirizzi logici della parte allocata e gli indirizzi fisici in cui viene allocata. Questa funzione, detta *funzione di rilocazione* o *di traduzione dell'indirizzo*, è di norma implementata come una tabella associata al processo (**Tabella di Rilocazione**).

La funzione di rilocazione viene aggiornata ogni volta che l'allocazione del processo viene modificata.

La traduzione dell'indirizzo deve essere effettuata in modo molto efficiente e quindi l'accesso alla Tabelle di Rilocazione viene effettuata con opportune soluzioni hardware-firmware delegate a un'unità interposta tra il Processore e la memoria principale (*Memory Management Unit*, o *MMU*).

5.3 D-RISC: un assembler didattico di tipo Risc

Le principali caratteristiche della macchina assembler, che chiameremo D-RISC, sono le seguenti:

- sono presenti 64 *registri generali* RG[0..63], mentre il registro *contatore istruzioni* (IC) è visibile solo a livello firmware; solo alcune istruzioni, tipicamente di salto o speciali, hanno effetto sul contenuto di IC, ma, a livello assembler, non esiste alcun altro modo di modificare tale registro;
- *la parola è di* 32 *bit*: questa caratteristica non è comunque vincolante per quanto riguarda la rappresentazione dei dati, che possono essere a 64 bit senza alterare le altre caratteristiche della macchina. Come detto sotto, vincolante è invece la lunghezza dell'istruzione e la lunghezza dell'indirizzo logico;
- la memoria, sia virtuale sia fisica, è *indirizzabile* alla parola;
- *lo spazio di indirizzamento logico di un processo è unidimensionale* e gli indirizzi logici sono di 32 bit. Lo spazio di indirizzamento logico di un processo è dunque più ampio al più 4G parole;
- le istruzioni sono tutte rappresentate su 32 bit. Supporremo che il *codice operativo* sia codificato negli 8 bit più significativi; sono dunque disponibili fino a 256 istruzioni, anche se di fatto ne potranno essere implementate meno.

5.4 Regole di compilazione e composizione

5.4.1 Comandi condizionali e iterativi

Indicheremo con C un predicato (guardia) e con B un comando o blocco (sequenza) di comandi. Con IF_C ETICHETTA e IF_N_C ETICHETTA indicheremo, in generale, le istruzioni assembler di salto condizionato C e not C rispettivamente.

La compilazione di un comando/blocco B nella corrispondente sequenza di istruzioni sarà indicata con *compile* (B).

Abbiamo allora le seguenti regole:

compile (**if** *C* **then** *B*) ≡		IF_N_*C* CONTINUA
		compile (B)
	CONTINUA: ...	
compile (**if** *C* **then** *B1* else *B2*) ≡		IF_*C* THEN
		compile (B2)
		GOTO CONTINUA
	THEN:	*compile* (B1)
	CONTINUA: ...	
compile (**while** *C* **do** *B*) ≡	LOOP:	IF_N_*C* CONTINUA
		compile (B)
		GOTO LOOP
	CONTINUA: ...	
compile (**do** *B* **while** *C*) ≡	LOOP:	*compile (B)*
		IF_*C* LOOP
	CONTINUA: ...	

5.4.2 Compilazioni di procedure e funzioni

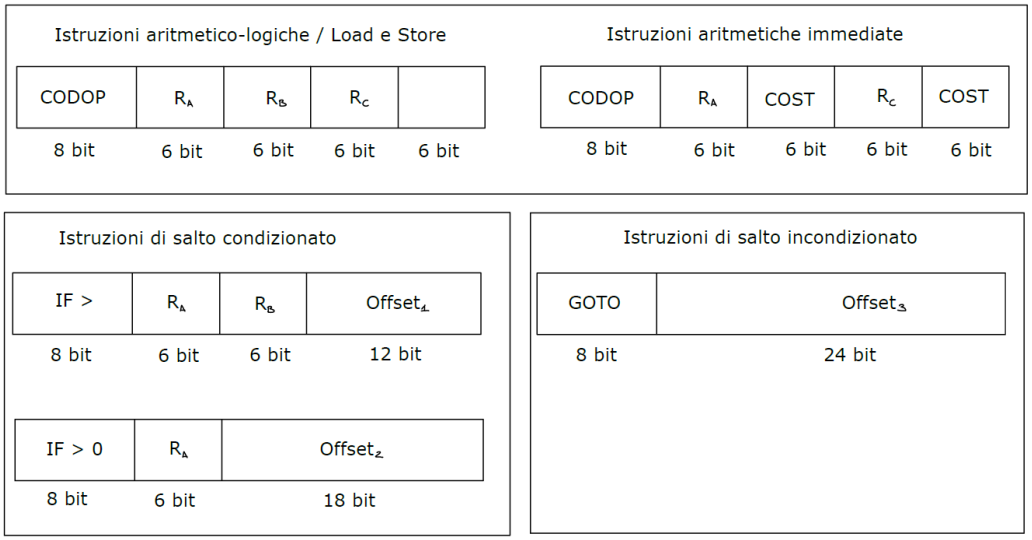
Si consideri una computazione che contenga una procedura o una funzione, nel seguito indicata con S.

I problemi da risolvere per la sua compilazione sono essenzialmente:

1. *meccanismo di chiamata* a S e *di ritorno* da S al programma chiamante;
2. *passaggio di parametri e risultati* tra programma chiamante e programma chiamato S.

1. All'atto della *chiamata*, nel programma chiamante occorre salvare l'indirizzo di ritorno ed effettuare un salto all'indirizzo di S: queste due azioni sono effettuate dall'unica istruzione `CALL` $\texttt{R}_{\texttt{proc}}, \texttt{R}_{\texttt{ret}}$. Il *ritorno* da S al programma chiamante avviene semplicemente con l'istruzione (già nota) `GOTO` $\texttt{R}_{\texttt{ret}}$ dove $\texttt{R}_{\texttt{ret}}$ è l'indirizzo del registro nel quale è stato salvato il valore di IC+1 al momento della chiamata.

2. **Passaggio dei parametri**: nel caso più semplice, ma piuttosto frequente, in cui i parametri di ingresso e di uscita di S siano relativamente pochi, il passaggio di tali parametri può essere effettuato attraverso registri generali. Altrimenti, e comunque qualora le procedure/funzioni siano annidate o ricorsive, occorre utilizzare una struttura dati in memoria di tipo pila gestita mediante istruzioni di `Load` e `Store`.



Assembler D-RISC

Istruzioni aritmetico logiche

Istruzione	Semantica	Registri letti	Registri scritti
ADD Ri, Rj, Rk	Reg[i]+Reg[j] → Reg[k], IC+1→ IC	{Ri, Rj}	{Rk}
ADDi Ri, #n, Rk	Reg[i]+n → Reg[k], IC+1→ IC	{Ri}	{Rk}
SUB Ri, Rj, Rk	Reg[i]-Reg[j] → Reg[k], IC+1→ IC	{Ri, Rj}	{Rk}
SUBiRi, #n, Rk	Reg[i]-n → Reg[k], IC+1→ IC	{Ri}	{Rk}
MUL Ri, Rj, Rk	Reg[i]*Reg[j] → Reg[k], IC+1→ IC	{Ri, Rj}	{Rk}
MULi Ri, #n, Rk	Reg[i]*n → Reg[k], IC+1→ IC	{Ri}	{Rk}
DIV Ri, Rj, Rk	Reg[i]/Reg[j] → Reg[k], IC+1→ IC	{Ri, Rj}	{Rk}
DIVi Ri, #n, Rk	Reg[i]/n → Reg[k], IC+1→ IC	{Ri}	{Rk}
INC Ri	Reg[i] + 1 → Reg[i], IC+1→ IC	{Ri}	{Ri}
DECR Ri	Reg[i] - 1 → Reg[i], IC+1→ IC	{Ri}	{Ri}
SHL Ri, Rj	Shift sn di Reg[j] posizioni di Reg[i] → Reg[i]	{Ri,Rj}	{Ri}
SHL Ri, #n	Shift sn di n posizioni di Reg[i] → Reg[i]	{Ri,Rj}	{Ri}
SHR Ri, Rj	Shift ds di Reg[j] posizioni di Reg[i] → Reg[i]	{Ri,Rj}	{Ri}
SHR Ri, #n	Shift ds di n posizioni di Reg[i] → Reg[i]	{Ri,Rj}	{Ri}

Istruzioni sulla memoria

Istruzione	Semantica	Registri letti	Registri scritti
LOAD Ri, Rj, Rk	Mem[Reg[i]+Reg[j]] → Reg[k], IC+1→ IC	{Ri, Rj}	{Rk}
LOADi Ri, off, Rk	Mem[Reg[i]+off] → Reg[k], IC+1→ IC	{Ri}	{Rk}
STORE Ri, Rj, Rk	Reg[k] → Mem[Reg[i]+Reg[j]], IC+1→ IC	{Ri, Rj}	{Rk}
STOREi Ri, off, Rk	Reg[k] → Mem[Reg[i]+off], IC+1→ IC	{Ri}	{Rk}
EXCH Ri, Rj, Rk	Scambia Reg[k] con Mem[Reg[i]+Reg[j]]	{Ri, Rj, Rk}	{Rk}

Istruzioni di salto incondizionato

Istruzione	Semantica	Registri letti	Registri scritti
GOTO Ri	Reg[i] → IC	{Ri}	
CALL Rf, Rret	IC + 1 → Reg[ret], Reg[f] → IC	{Rf}	{Rret}
GOTO etich	IC + etich → IC		

Istruzioni di salto condizionato

Istruzione	Semantica	Registri letti	Registri scritti
IF< Ri, Rj, etich	(Reg[i]<Reg[j] ? IC + etich : IC+1) → IC	{Ri, Rj}	
IF= Ri, Rj, etich	(Reg[i]=Reg[j] ? IC + etich : IC+1) → IC	{Ri, Rj}	
IF> Ri, Rj, etich	(Reg[i]>Reg[j] ? IC + etich : IC+1) → IC	{Ri, Rj}	
IF<= Ri, Rj, etich	(Reg[i]<=Reg[j] ? IC + etich : IC+1) → IC	{Ri, Rj}	
IF>= Ri, Rj, etich	(Reg[i]>=Reg[j] ? IC + etich : IC+1) → IC	{Ri, Rj}	
IF<0 Ri, Rj, etich	(Reg[i]<0 ? IC + etich : IC+1) → IC	{Ri}	
IF=0 Ri, Rj, etich	(Reg[i]=0 ? IC + etich : IC+1) → IC	{Ri}	
IF>0 Ri, Rj, etich	(Reg[i]>0 ? IC + etich : IC+1) → IC	{Ri}	
IF<=0 Ri, Rj, etich	(Reg[i]<=0 ? IC + etich : IC+1) → IC	{Ri}	
IF>=0 Ri, Rj, etich	(Reg[i]>=0 ? IC + etich : IC+1) → IC	{Ri}	

“Pseudo” istruzioni

Istruzione	Compilata come	Registri letti	Registri scritti
CLEAR Ri	ADD R0, R0, Ri		{Ri}
MOV Ri, Rj	ADD Ri, R0, Rj	{Ri}	{Rj}

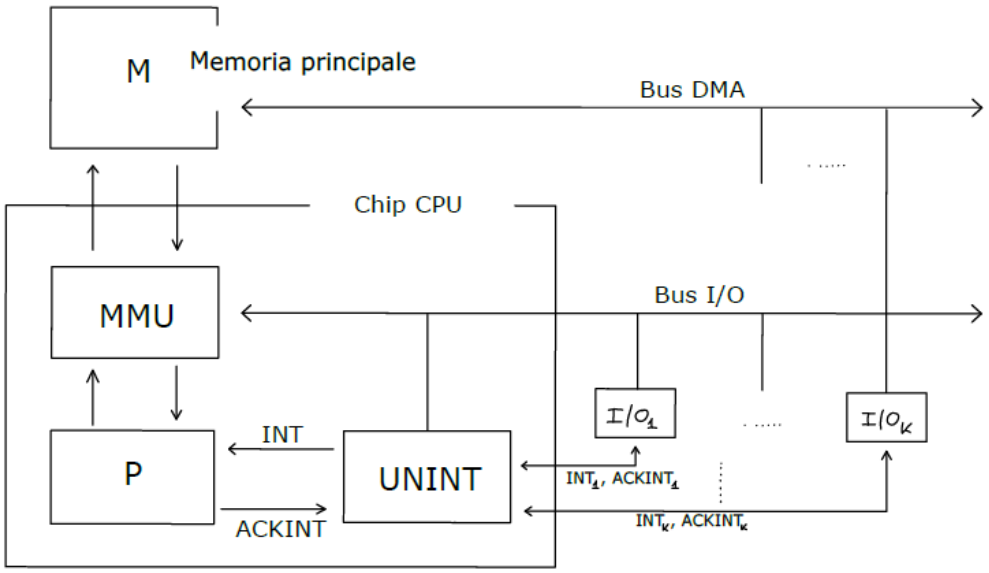
Istruzioni speciali

Istruzione	Compilata come	Registri letti	Registri scritti
START_PROC Ra, Rb	Ra inviato a MMU, Reg[b] → IC	{Ra, Rb}	
NOP	Non fa nulla		
MASKINT Ra	Utilizza Ra come registro maschera delle interruzioni	{Ra}	
DI	Disabilita interruzioni		
EI	Abilita interruzioni		
COPY_IC Ra	IC → Reg[a]		{Ra}
END	Fine programma		

Capitolo 6

Architettura Base della CPU

6.1 Architettura complessiva



L'unità MMU, oltre che tradurre in maniera molto efficiente gli indirizzi logici, ha anche il compito più generale di interfacciare tutte le unità, svincolando il progetto del processore da molti aspetti legati alla realizzazione fisica della memoria nonché del sottosistema di I/O.

Tutte le unità di I/O usano il Bus di I/O per scambiare informazioni direttamente con il processore centrale; tali informazioni potranno essere *dati* e/o *comandi* che interessano la cooperazione tra processi centrali e unità di I/O.

Un certo sottoinsieme di unità di I/O può anche trasferire dati nei confronti dell'unità centrale accedendo *direttamente in memoria* (*DMA*: Direct Memory Access): è soprattutto il caso di unità (come dischi, stampanti veloci, unità di rete) che effettuano i trasferimenti a *blocchi*, alleggerendo quindi notevolmente il processore dal compito di gestire il trasferimento, e quindi permettendo al processore stesso di eseguire parallelamente altri processi. Il *Bus DMA* provvede a realizzare un canale diretto tra le unità di I/O e la memoria principale.

Le *interruzioni* hanno un doppio significato:

- *a livello firmware*, sono segnali per il meccanismo di arbitraggio del Bus di I/O. Ogni volta che un'unità di I/O ha bisogno di segnalare eventi direttamente al processore, deve usare il Bus di I/O in scrittura e, quindi, deve prima interagire con l'arbitro di tale Bus attraverso dei segnali, detti appunto *segnali d'interruzione*;
- *a livello dei processi*, partecipano a supportare la cooperazione e sincronizzazione tra i processi eseguibili dalla CPU ed elaborazioni concorrenti effetturare dall'unità di I/O.

All'interno della CPU, è l'unità indicata con UNINT che esegue le funzioni di arbitraggio vere e proprie. Anche in questo caso, la presenza di UNINT svincola il progetto del processore da tutta una serie di scelte implementative legate all'arbitraggio delle interruzioni; inoltre, tale arbitraggio avviene in parallelo all'elaborazioni del processore.

Il segnale d'interruzione INT, testato dal processore alla fine dell'interpretazione di ogni istruzione assembler è il risultato della funzione OR, realizzato in UNINT, di tutti i segnali d'interruzione. Nel caso accetti l'interruzione, il processore invia il segnale `ACKINT` a `UNINT`, la quale provvede a smistarlo all'unità prescelta dal meccanismo di arbitraggio.

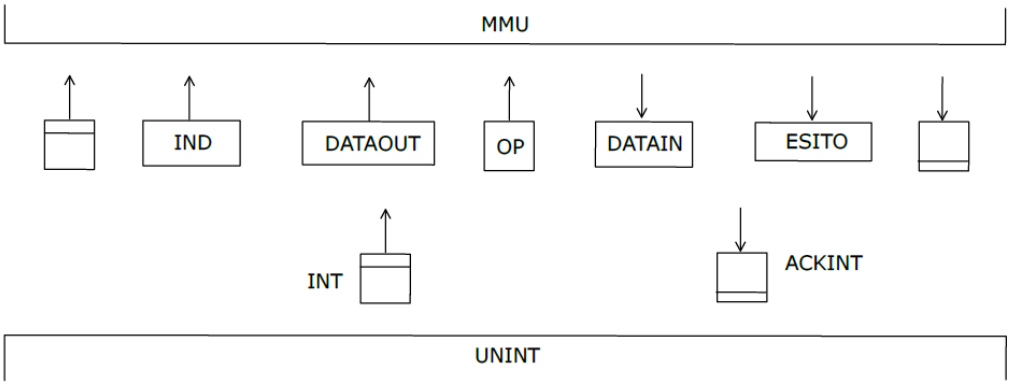
6.2 Interprete del set d'istruzioni

6.2.1 Specifiche della macchina firmware

Oltre agli stessi registri assembler RG[0..63] e al registro IC, la macchina firmware contiene ovviamente (nella PO del processore) altri registri, *visibili solo a questo livello*, come:

- IR: registro istruzione corrente;
- registri temporanei e interfacce.

La memorietta per implementare l'array RG consente, in uno stesso ciclo di clock, *la lettura di due registri* e *la scrittura in un singolo registro*. RG deve essere indirizzabile, oltre che dai campi di IR, anche dai campi di altri registri e costanti generati dalla PC.



Interfaccia a transizione di livello del processore verso la MMU e verso la UNINT

```
while (true) {
    fetch istruzione corrente;
    decode istruzione corrente;
    execute istruzione corrente;
    controlla le interruzioni;
}
```

Ch0. "read" ⟶ OP, IC ⟶ IND, set RDYOUT, Ch1.

Ch1. (RDYIN, or (ESITO) = 0 -) nop, Ch1.
(= 11) nop., tratt. ecc.
(= 10) reset RDYIN, DATAIN ⟶ IR, Ch2.

Ch2. (IR.COP, INT = "add", 0) REG (IR.R_A) + REG (IR.R_B) ⟶ REG (IR.R_C), IC+1 ⟶ IC, Ch0.
(= "add", 1) REG (IR.R_A) + REG (IR.R_B) ⟶ REG (IR.R_C), IC+1 ⟶ IC, tratt. int.
(= "sub", 0) REG (IR.R_A) - REG (IR.R_B) ⟶ REG (IR.R_C), IC+1 ⟶ IC, Ch0.
(= "sub", 1) REG (IR.R_A) - REG (IR.R_B) ⟶ REG (IR.R_C), IC+1 ⟶ IC, tratt. int.

(= "load", -) REG(IR.R_{BASE}) + REG(IR.R_{INDICE}) ⟶ IND, "read" ⟶ OP, set RDYOUT, load0.

(= "store", -) REG(IR.R_{BASE}) + REG(IR.R_{INDICE}) ⟶ IND, "write" ⟶ OP, set RDYOUT,
REG(IR.R_C) ⟶ DATAOUT, store0.

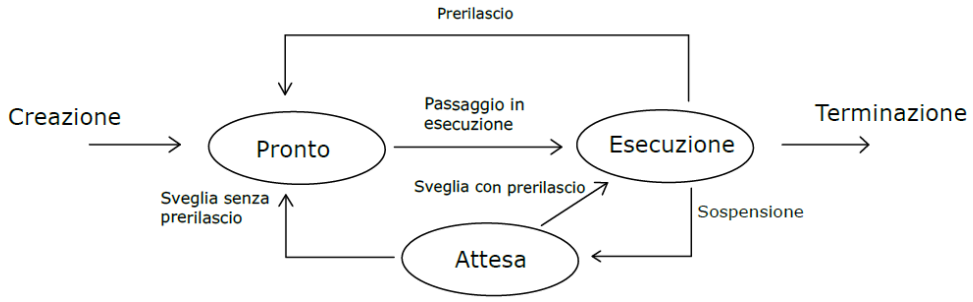
(= "GOTOR", 0) REG(IR.R_A) ⟶ IC, Ch0.
(= "GOTOR", 1) REG(IR.R_A) ⟶ IC, tratt. int.
(= "GOTOL", 0) IC + IR.OFFSET ⟶ IC, Ch0.
(= "GOTOL", 1) IC + IR.OFFSET ⟶ IC, tratt. int.

Capitolo 7

Il livello dei processi

7.1 Scheduling a basso livello

Lo *scheduling a basso livello* è il complesso di funzionalità per la gestione degli *stati di avanzamento dei processi* e quindi la gestione della risorsa processore.



Nello stato di *esecuzione* ("running") un processo Q utilizza effettivamente la risorsa processore per eseguire le proprie istruzioni. Nello stato di *pronto* Q ha tutte le caratteristiche per essere eseguibile, eccetto la disponibilità della risorsa processore; nel frattempo, in generale, esistono uno o più processi che precedono Q in un ordinamento di processi pronti e quindi destinati a utilizzare il processore (Lista Pronti). Nello stato di *attesa* Q è sospeso nei confronti del verificarsi di un certo evento logico, senza dunque avere possibilità di essere eseguito, nemmeno se una risorsa processore fosse disponibile, fintanto che tale evento non si sarà verificato.

Il passaggio da *esecuzione* ad *attesa* provoca la conseguente *commutazione di contesto* (del processore) per permettere il passaggio di un altro processo da *pronto* a *esecuzione*.

Quando si verifica l'evento sul quale un processo si è sospeso, il processo viene risvegliato e passa in stato di *pronto*.

Se è prevista una priorità dei processi e il processo svegliato ha priorità più alta del processo in *esecuzione*, si può avere la transizione da *attesa* direttamente a *esecuzione* con conseguente passaggio del processo "running" da *esecuzione* a *pronto* (*prerilascio* del processore).

In generale, il *prerilascio di un processore* si può verificare in due occasioni distinte:

1. un processo A in *attesa* viene svegliato da un processo B e A ha priorità maggiore di B: A passa direttamente in *esecuzione* sul processore utilizzato da B, e B passa in stato di *pronto*;

2. un processo A in *esecuzione* passa in stato di *pronto* per permettere al primo processo *pronto* di passare in *esecuzione*: questo meccanismo viene utilizzato nella gestione del processore *a quanti di tempo*, in modo da bilanciare l'utilizzazione del processore stesso da parte di processi "lunghi", e/o con scarse occasioni di cooperazione con altri processi, e di processi "corti", e/o con frequenti interazioni con altri processi.

Lo scheduling a quanti di tempo utilizza un'apposita unità di I/O che funge da *Timer*; a distanza di un quanto di tempo (scandito da un certo numero di cicli di clock dell'unità), tale unità invia un'*interruzione* di "fine quanto di tempo": il suo trattamento consiste nella *sequenza di azioni per effettuare il prerilascio del processore*.

Una volta *creato*, un processo passa in stato di *pronto*.

Eseguendo l'istruzione `END` il processo *termina*.

Capitolo 8

Gerarchie di Memoria e Architettura con Cache

8.1 Gerarchie di Memoria

La grande quantità di dati, normalmente presente in un sistema di elaborazioni, viene memorizzata su sopporti con caratteristiche molto diverse tra loro in termini di tempo di accesso, capacità e costo. Diremo che esiste una *gerarchia di memoria*, nella quale al *livello più alto* stanno i dispositivi di memoria più capaci, più lenti e meno costosi. Man mano che si scende di livello nella gerarchia, i supporti hanno capacità più piccola, tempo di accesso sempre più basso e costo per bit sempre più grande.

Salendo di livello nella gerarchia di memoria, il tempo di accesso aumenta non solo per le *caratteristiche intrinseche dei supporti di memorizzazione*, ma anche per i ritardi crescenti nelle *comunicazioni* necessarie per richiedere le informazioni e per farle pervenire ove devono essere elaborate.

Di tutta l'enorme massa d'informazioni disponibili in un sistema, istante per istante solo una piccolissima parte viene utilizzata dall'elaborazione in corso; d'altronde, questa parte varia *dinamicamente* durante l'elaborazione stessa. L'obiettivo di una gerarchia di memoria, che possa dirsi "efficiente", è quello di riuscire, istante per istante, a mantenere nei livelli più bassi tutte e sole le informazioni strettamente indispensabili all'elaborazione corrente. Ciò comporta un *continuo spostamento* d'informazioni attraverso i livelli della gerarchia di memoria: quelle che, man mano, si rendono sempre più utili scendono di livello, andando a sostituirne altre, ritenute momentaneamente meno utili, le quali vengono riportate nei livelli più alti.

8.1.1 Gerarchie di memoria con paginazione

Consideriamo due livelli contigui, M_1 e M_2, nella gerarchia di memoria, con M_2 livello più alto. Il sistema genera indirizzi appartenenti a M_2; questi vengono tradotti in indirizzi di M_1 se l'informazione riferita in M_2 è attualmente allocata in M_1, altrimenti viene generata un'eccezione che provoca la riallocazione di M_1.

Il metodo che sta alla base della maggior parte delle applicazioni del concetto di gerarchia di memoria è quello della **paginazione**. Lo spazio di M_1 e quello di M_2 *vengono pensati* come suddivisi in blocchi, o pagine, di ampiezza fissa composti di informazioni a indirizzi contigui.

Indicheremo con

- γ la capacità complessiva di uno spazio di memoria;
- σ l'ampiezza di una pagina nella gerarchia di memoria considerata;
- $v = \gamma/\sigma$ il numero di pagine componenti quello spazio di memoria nella gerarchia considerata.

8.1.2 Paginazione su domanda

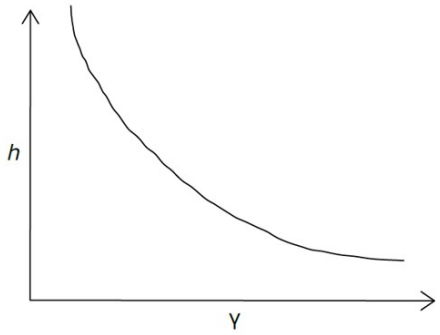
Nel metodo detto della *paginazione su domanda*, quando, nel tentativo di tradurre un indirizzo n di M_2 viene generato un fault, allora l'*intera pagina* che contiene inf[n] viene trasferita da M_2 a M_1 provocando il *rimpiazzamento* di una pagina già presente in M_1. L'efficacia di questo metodo si basa su due proprietà dei programmi che si verificano di frequente:

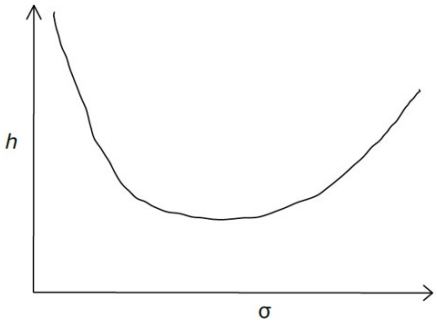
- la *località*, o località temporale, secondo la quale i riferimenti generati da un processo tendono ad accentrarsi, in ogni istante, in gruppi relativamente piccoli di indirizzi tra loro vicini, e tali gruppi tendono a cambiare in modo relativamente lento e intermittente;
- il *riuso*, o località spaziale, secondo la quale, nel corso dell'esecuzione di un processo, queste tende a riferirsi più volte a certe locazioni (cicli iterativi, procedure, riassegnamenti di una variabile).

In tal modo, una volta che una pagina è stata trasferita in M_1, questa tende a essere riferita più volte. Se in M_1, è presente un numero sufficientemente alto di pagine di M_2, la probabilità che si generi un fault (**probabilità di fault**) può essere mantenuta a un valore basso a piacere.

La probabilità di fault, h, è in genere una funzione della capacità del livello più basso della gerarchia, dell'ampiezza del blocco, e della politica di rimpiazzamento delle pagine.

Le seguenti figure mostrano qualitativamente l'andamento di h al variare di γ, per σ costante, e di h al variare di σ, γ con costante:





8.1.3 Insieme di lavoro

In una gerarchia di memoria M_2-M_1, l'insieme di lavoro di un programma (processo) è definito come l'*insieme di pagine* (*blocchi*) che, *se presenti contemporaneamente in* M_1*, rendono minima la probabilità di fault*. L'insieme di lavoro va considerato sia in termini di quante pagine che di quali pagine devono essere possibilmente presenti in M_1 contemporaneamente.

L'obiettivo di una gestione efficiente di una gerarchia di memoria è quello di individuare l'insieme di lavoro e cercare di mantenerlo in M_1. A questo scopo, occorre cercare di sfruttare al meglio le proprietà di *località* e, in particolare, di *riuso* di ogni programma.

8.2 Gerarchia Memoria Virtuale - Memoria Principale

8.2.1 Spazi di indirizzamento

Durante la fase di traduzione, il compilatore provvede a scegliere gli indirizzi di memoria per le istruzioni e per i dati di Q. Viene così definito lo *spazio di indirizzamento logico del processo*, N, cioè l'insieme di tutti gli indirizzi, noti a tempo di compilazione e generati dal processore a tempo di esecuzione.

Chiamiamo *spazio di indirizzamento fisico*, M, l'insieme degli indirizzi di memoria principale (*indirizzi fisici*) assegnati a un processo quando viene allocato in memoria principale. M può coincidere o essere un sottoinsieme dei possibili indirizzi della memoria principale.

8.2.2 Allocazione della memoria principale

Nel caso di allocazione statica della memoria principale, N e M coincidono. Per essere eseguito, tutto il processo è interamente caricato da memoria secondaria a memoria principale in un'area nota.

Nel caso di allocazione dinamica della memoria principale, lo spazio M assegnato al processo varia a tempo di esecuzione sia in posizione sia in quantità. In generale, non tutto N è allocato contemporaneamente in memoria principale, ma solo una sua parte, possibilmente quella che, istante per istante, permette al processo di reperire la maggior parte delle informazioni (istruzioni e dati) in memoria principale. La parte di N allocata in memoria principale varia a tempo di esecuzioni, così come la zona di memoria principale in cui allocare N. Inoltre, in generale tale zona non è contigua, anche nel caso che N lo sia.

Per un processo Q deve essere definita una funzione di rilocazione, o funzione di traduzione degli indirizzi:

$$\mu_Q : N \rightarrow M$$

La funzione di traduzione degli indirizzi μ_Q, implementata con la Tabella di Rilocazione, viene aggiornata e valutata a tempo di esecuzione per ogni indirizzo logico generato dal processo Q in esecuzione sul processore. Ciò avviene con l'ausilio di supporti hardware-firmware particolarmente efficienti: l'unità *MMU* che, facente parte della CPU, fa da tramite tra processore e memoria principale, provvedendo a tradurre gli indirizzi da logici a fisici in un solo ciclo di clock; nel caso che μ_Q non sia definita, *MMU* restituisce l'eccezione al processore.

8.2.3 Creazione di processi, caricamento, e commutazione di contesto

Quando un processo Q esegue una primitiva di *creazione* di un altro processo R, questa provoca un *caricamento* di R, cioè il trasferimento di informazioni di R da memoria secondaria a memoria principale. Una volta creato, R viene posto nello *stato di pronto*.

Il file, che rappresenta il prodotto della traduzione dal programma al processo R, contiene diverse informazioni, sotto forma di opportune strutture dati, che verranno utilizzate da funzionalità del supporto durante l'evoluzione del processo stesso: in particolare, la struttura dati *descrittori di processo* (*PCB*).

Il PCB viene inizializzato a tempo di compilazione, in particolare assegnando i valori costanti con cui inizializzare i registri generali e il contatore istruzioni.

Quando il processo R viene creato, in memoria principale viene compilato anche il suo PCB con il valore che è stato inizializzato a tempo di compilazione.

Quando verrà il turno di R a passare in esecuzione sul processore, ha luogo *la commutazione di contesto*:

- l'immagine di RG e di IC viene copiata dall'area facente parte del $\mathrm{PCB_R}$ nei registri RG e nel registro IC. In tal modo:
  - i registri sono inizializzati come stabilito a tempo di compilazione,
  - la prima istruzione da eseguire sarà quella all'indirizzo logico di R, stabilito a tempo di compilazione, che è stato scritto in IC;
- il processo (T) che esce dallo stato di esecuzione provvede a salvare nel proprio PCB_T i valori correnti di RG e IC. In tal modo, quando successivamente T tornerà in esecuzione, RG e IC verranno ripristinati con le rispettive immagini presenti nel PCB_T, esattamente secondo lo schema descritto precedentemente, in modo che T possa riprendere l'esecuzione a partire dall'ultima eseguita prima di sospendersi e con i contenuti dei registri generali presenti prima di sospendersi.

8.2.4 Allocazione dinamica della memoria con paginazione

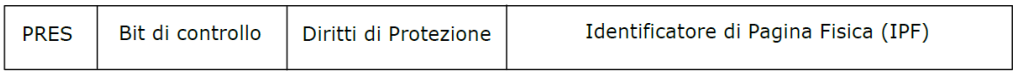
Il processo, una volta creato, in ogni istante risulta allocato in uno spazio fisico di memoria principale, che in generale non è né contiguo né completo: non tutto il processo risiede contemporaneamente in memoria, e la parte che vi risiede è distribuita in *blocchi*, o *pagine*, di ampiezza fissa e al loro interno contigue, ma sparse in zone diverse della memoria.

In un caso tipico, una pagina è ampia 1K parole con indirizzi logici di 32 bit. L'indirizzo logico può dunque essere visto come composto dalla concatenazione di due campi: quello dei bit più significativi fornisce l'*identificatore di pagina logica*, IPL, e quello dei bit meno significativi l'*indirizzo all'interno della pagina* o *displacement*.

La funzione di rilocazione del processo Q, μ_Q, viene implementata mediante una tabella, detta Tabella di Rilocazione del processo, TABRIL_Q, il cui contenuto è inizializzato a tempo di caricamento/creazione e varia durante l'esecuzione del processo in seguito all'allocazione dinamica di pagine.

Attraverso la Tabella di Rilocazione la funzione di rilocazione viene applicata alle pagine. L'entrata i-esima della Tabella di Rilocazione contiene, tra le altre informazioni:

- un *bit di presenza*, `PRES`, che, se a uno, indica che la pagina logica i-esima è presente in memoria principale;
- un *identificatore di pagina fisica*, `IPF`, cioè l'identificatore della pagina di memoria principale dove risiede la pagina logica i-esima nel caso che `PRES` $= 1$.



Indichiamo con $x^{\circ}y$ il concatenamento dei valori x, y in una stessa unità di informazione. Dato l'indirizzo logico IPL$^{\circ}$*displ* generato dal processo Q, se TABRIL$_Q$[IPL].`PRES` $= 1$, allora l'indirizzo fisico corrispondente è dato da TABRIL$_Q$[IPL].`IPF`$^{\circ}$*displ*.

Se TABRIL$_Q$[IPL].`PRES` $= 0$, allora viene generata l'eccezione di fault di pagina.

Nello schema di TABRIL è anche indicato il campo "Bit di controllo" contenente un bit per ricordare se la pagina è stata modificata e informazioni per implementare la politica di rimpiazzamento. Quest'ultima politica, *eseguita dalle funzionalità di gestione della memoria in seguito all'eccezione di fault di pagina*, ha come scopo la scelta della pagina da sostituire; *la scelta che mediamente fornisce il risultato migliore, agli effetti della riduzione della frequenza dei fault di pagina* è la **LRU** (Least Recently Used).

8.2.5 MMU

La traduzione dell'indirizzo logico e l'eventuale generazione di eccezioni di fault di pagina sono implementate a hardware-firmware nell'unità *MMU* facente parte della CPU.

Al processore P non è visibile come avviene la traduzione degli indirizzi, ma deve comunque essere noto l'*esito* di ogni accesso in memoria.

Per effettuare efficientemente la traduzione dell'indirizzo, *MMU* deve disporre di risorse hardware opportune. Occorre allora realizzare in hardware una *tabella accedibile per contenuto*: questo componente è chiamato **Memoria Associativa** (MA) ed è costituita da C registri, ognuno contenente una *chiave*, e altrettanti contenenti i corrispondenti *valori*.

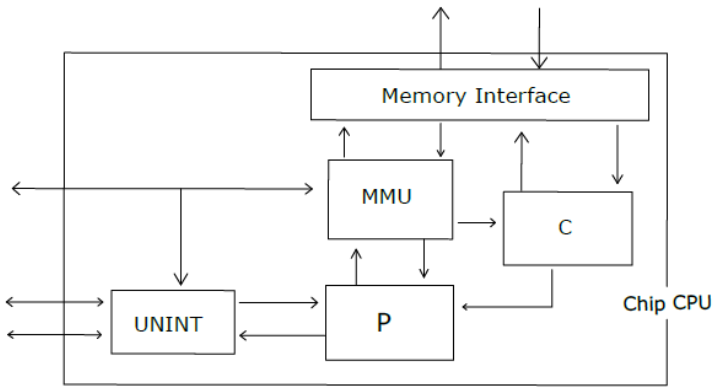
Le uscite dei registri-chiave sono ingressi di C confrontatori, il cui secondo ingresso è costituito dalla *chiave Key* con la quale si vuole effettuare la ricerca. Quindi viene effettuato il confronto *simultaneo* di *Key* con tutte le Chiavi in tabella. Uno dei confronti è positivo, allora l'indice del registro-chiave relativo fornisce univocamente l'indirizzo della RAM dalla quale leggere il Valore corrispondente a *Key*; in caso di tutti i confronti negativi, l'evento negativo viene segnalato attraverso il valore di un bit di Fault.

Una parte di Tabella di Rilocazione TABRIL è mantenuta, dinamicamente, nella Memoria Associativa MA nella PO della *MMU. Le chiavi sono gli identificatori di pagina logica IPL*, mentre *i valori sono i corrispondenti contenuti TABRIL[IPL] della Tabella di Rilocazione.*

Normalmente, si mantengono in MA le entrate di TABRIL *usate più di recente*: questo permette di rendere trascurabile la probabilità che l'entrata di interesse non risieda in MA. In caso che venga generato un fault di MA, *MMU* provvede a copiare in MA l'entrata di TABRIL corrispondente alla pagina logica riferita, sostituendo l'entrata usata meno di recente, per questa ragione all'atto della commutazione di contesto, quando viene posto in esecuzione il processo Q, *l'indirizzo di* TABRIL_Q deve essere comunicato a *MMU*, che lo mantiene in un suo registro interno.

8.3 Architettura con cache

8.3.1 Struttura della CPU con cache



Nel caso di successo nella rilocazione dell'indirizzo logico (di memoria virtuale), *MMU* passa la richiesta con l'indirizzo fisico (di memoria principale) all'unità cache C. Questa provvede *all'ulteriore traduzione da indirizzo di memoria principale a indirizzo di memoria cache*, nel caso che la funzione

di traduzione sia definita; in caso di insuccesso (cache *fault*) ha luogo il *trasferimento* del blocco di memoria principale richiesto in uno dei blocchi di C, in generale sostituendone uno già presente scelto mediante un opportuno algoritmo di rimpiazzamento. Il trasferimento del blocco è, a differenza del fault di memoria virtuale, del tutto *invisibile* a P.

Occorre prevedere un metodo per il trattamento delle **scritture nella memoria cache**:

1. nel metodo *Write Back*, il blocco modificato viene ricopiato nel livello superiore di memoria solo all'atto della sostituzione del blocco stesso nel livello superiore o alla terminazione del processo;

2. nel metodo *Write Through*, ogni modifica della cache è effettuata immediatamente e contemporaneamente anche in memoria principale.

8.3.2 Metodi di indirizzamento della cache

I principali metodi sono chiamati **diretto**, **completamente associativo** e **associativo su insiemi**.

Metodo diretto

In questo caso ogni blocco di memoria principale può essere trasferito solo in un bene determinato blocco della cache: esiste dunque uno o un solo blocco della cache in cui una certa informazione può risiedere.

Si può scegliere di adottare una semplice funzione di corrispondenza tra blocchi, tipicamente la funzione *modulo*, secondo la quale l'identificatore BC del blocco di cache è dato da: $BC = BM \ mod \ NC$, dove BM indica l'identificatore del blocco di M e NC il numero di blocchi in cache.

Se, come di regola, NC è una potenza di 2, il campo BM dell'indirizzo di M contiene l'informazione BC direttamente nei $\log_2 NC$ bit meno significativi, mentre i restanti bit di BM, indicati con TAG, $TAG = BM \ div \ NC$ identificano il blocco di M all'interno dell'insieme di tutti quelli che corrispondono a quel particolare blocco di C identificato da BC.

In questo metodo dunque la funzione di traduzione è immediata, essendo l'eventuale BC corrispondente a BM già presente nello stesso indirizzo fisico. L'accesso alla cache è dunque effettuato direttamente. Il sistema dovrà mantenere una *Tabella di Corrispondenza*, avente NC posizioni: ogni posizione contiene il valore del TAG corrispondente al blocco di M effettivamente presente nel blocco di cache individuato da BC.

Con il metodo diretto tenendo conto anche della presenza della *MMU*, il tempo di accesso alla memoria cache, in assenza di fault è $t_c = 2\tau$ che rappresenta il minimo valore possibile con il protocollo a domanda e risposta.

I vantaggi di questo metodo sono la velocità e la semplicità. Per contro esso presenta uno svantaggio: quando occorre accedere ripetutamente a coppie di informazioni che stanno in blocchi di M corrispondenti allo stesso blocco di C, il numero di fault risulta molto elevato.

Rispetto ai valori per la probabilità di fault h calcolati per gli altri metodi si può verificare una degradazione fino al 50%, a parità di ampiezza dei blocchi e capacità dalla cache.

Un caso in cui l'inconveniente descritto è poco rilevante è quello delle istruzioni. Suddividendo, come di regola, la cache in una **cache istruzioni** o una **cache dati**, in diversi sistemi la prima adotta il metodo diretto.

Metodo completamente associativo

Questo metodo, a differenza del precedente, offre la massima flessibilità circa la corrispondenza tra blocchi di M e blocchi di C: ogni blocco di M può risiedere in *qualsiasi* blocco di C.

La parte BM dell'indirizzo fisico generato è usata come chiave per una ricerca tabellare mediante la quale implementare la funzione di traduzione dell'indirizzo. Per ragioni di efficienza, tale *Tabella di Corrispondenza* non può che essere realizzata con una memoria associativa: la sua uscita fornisce, se non si verifica fault, l'identificatore del blocco di C.

Tenendo conto anche della presenza della *MMU*, il tempo di accesso alla memoria cache, in assenza di fault, è: $t_c = 3\tau$.

Assumendo che in ciclo di clock sia possibile stabilizzazione di un solo componente logico di memoria.

Questo metodo fornisce la massima flessibilità al prezzo di un maggiore tempo di accesso e di un aumento di costo dovuto alla memoria associativa.

Metodo associativo su insiemi

Questo metodo approssima soddisfacentemente sia la flessibilità del metodo completamente associativo che la semplicità del metodo diretto.

Ogni blocco di memoria principale è fatto corrispondere a un ben determinato *insieme* di blocchi della cache, potendo essere però allocato in uno qualsiasi dei blocchi di tale insieme.

Similmente a quanto fatto nel Metodo Diretto, adottiamo la funzione *modulo* per esprimere la corrispondenza tra blocchi di M e insiemi di cache; l'identificatore SET dell'insieme dei blocchi di cache è dato da: $SET =$

BM mod NS dove *BM* indica l'identificatore del blocco M e *NS* il numero di insiemi di blocchi della memoria di cache. Se, come di regola NS è una potenza di 2, il campo *BM* dell'indirizzo di M contiene l'informazione *SET direttamente* nei $\log_2 NS$ bit meno significativi, mentre i restanti bit di *MB*, indicati con *TAG*, *TAG = BM div NS* identificano il blocco di M all'interno dell'insieme di tutti quelli che corrispondono a quel particolare insieme di blocchi di C identificato da *SET*.

Tenendo conto anche della presenza della *MMU*, il tempo di accesso alla memoria cache, in assenza di fault, è $t_c = 2\tau$.

È stato verificato mediante simulazioni ed esperimenti che insiemi di 8 blocchi permettono di ottenere una *probabilità di fault* molto vicina a quella del metodo completamente associativo. D'altra parte, la logica necessaria è paragonabile a quella del metodo diretto.

8.3.3 Modello dei costi

Si vuole valutare il tempo di completamento del programma e il tempo di servizio per istruzione. L'*efficienza relativa* dell'architettura con cache, ϵ, è valutata come il rapporto tra tempo di completamento T_{c-0} *in assenza di fault di cache* e il tempo di completamento *effettivo* T_c considerando la penalità dovuta ai fault T_{fault}.

Il tempo di completamento è valutato come:

$$T_c = T_{c-0} + T_{\texttt{fault}}$$

T_{fault} si valuta come:

$$T_{fault} = N_{fault} + T_{transf}$$

Dove

- N_{fault} è il numero medio di fault di cache che hanno luogo durante tutta l'esecuzione dello specifico programma,
- T_{transf} è il tempo necessario per effettuare il trasferimento di un blocco di cache dal livello superiore alla cache primaria.

Da T_c si ricavano il tempo di servizio T:

$$T = \frac{T_c}{N_{istr}}$$

Dove N_{istr} è il numero medio di istruzioni eseguite per completare il programma, e l'efficienza relativa:

$$\epsilon = \frac{T_{c-0}}{T_c}$$

Trasferimento di blocchi dalla memoria principale

Nel caso di trasferimento del blocco da M a C, poiché dobbiamo leggere σ parole a indirizzi consecutivi, una soluzione adatta è dunque l'organizzazione di memoria modulare interallacciata ottenendo così:

$$T_{transf} = 2T_{tr} + \frac{\sigma}{m}\tau_M + m\tau$$

8.3.4 Cache a più livelli

La soluzione della memoria interallacciata riesce solo in parte a soddisfare il requisito di minimizzare la latenza del trasferimento dei blocchi. Occorre considerare che sono ancora significative le latenze dei singoli moduli di memoria e dei collegamenti. Si deve perciò ricorrere a *tecnologie di memoria che riducano la latenza di per sé*, senza necessariamente introdurre il parallelismo. In pratica, questo principio conduce all'introduzione di ulteriori livelli di cache, in particolare la **cache secondaria** integrata sullo stesso chip CPU con la stessa tecnologia della primaria.

La capacità di C_2 è un ordine di grandezza superiore a C_1, con blocchi più ampi. C_1 è utilizzata da un solo processo alla volta. In C_2, invece, sono normalmente presenti blocchi appartenenti a più processi in stato di pronto.

Il principio è estendibile alla **cache terziaria**, anch'essa on-chip quando prevista.

Combinando le due soluzioni discusse (*M esterna interallacciata per avere parallelismo), C_2 ed eventualmente C_3 on-chip per abbattere le latenze*, si giunge alla soluzione architetturale più frequente, nella quale la cache primaria funziona on demand, ma i livelli superiori di cache funzionano con prefetching, e le memorie esterne al chip CPU hanno organizzazione interallacciata. Vale la pena notare che a parità di capacità complessiva γ, mantenere la distinzione tra cache primaria e secondaria on-chip è conveniente rispetto ad avere soltanto un'unica cache primaria di capacità γ. Infatti:

- la cache primaria contiene solo informazioni del processo in esecuzione, mentre la cache secondaria può mantenere informazioni di più processi, e quindi essere pronta a trasferire blocchi in/da cache primaria in caso di commutazione di contesto;
- blocchi della cache secondaria hanno dimensione adatta al trasferimento con la memoria principale (o cache terziaria);
- la cache secondaria può trasferire blocchi dalla memoria principale (o cache terziaria) in parallelo all'esecuzione del processo e all'uso della ca-

che primaria, ad esempio anticipando blocchi del processo in esecuzione o di altri processi con la tecnica del *prefetching*.

Queste caratteristiche fanno sì che, in molti programmi, si possa assumere *trascurabile la probabilità di fault in* C_2 in seguito ad una richiesta di C_1.

Capitolo 9

Ingresso - Uscita

9.1 Compiti delle unità di I/O e cooperazione con i processi della CPU

Ogni unità di I/O svolge il compito di interfacciare, nei confronti dell'unità centrale, un certo tipo di dispositivo periferico, come dischi fissi e altre memorie di massa, stampanti, scanner, dispositivi audio, dispositivi video, interfacce di rete ...

9.1.1 Unità di I/O e driver

Oltre che compiti d'interfacciamento, una unità di I/O svolge, in generale, anche compiti di *pre-elaborazione*, o *post-elaborazione*, nei confronti delle funzionalità di trasferimento dati tra unità centrale e dispositivi periferici e in molti casi svolge anche compiti più sofisticati di *gestione* e *controllo del dispositivo*.

In effetti, il confine tra funzionalità delegate al processore e funzionalità delegate all'unità di I/O per la gestione di un dispositivo è molto sfumato. Normalmente, a livello di sistema sono previsti degli appositi programmi, o processi, detti *driver* dei dispositivi periferici, aventi appunto il compito di eseguire alcune delle funzionalità necessarie per gestire al meglio tali dispositivi nei confronti dei processi utente. Una parte di tali funzionalità, quelle più critiche in termini di prestazioni, viene realizzata "a firmware" nelle unità di I/O, mentre quello meno critiche vengono realizzate "a software" nei driver.

9.1.2 Cooperazione processi - unità di I/O

Le elaborazioni svolte dalle unità di I/O possono essere considerate dei veri e propri processi, detti **processi esterni**, eseguibili solo sui "processori" costituiti dalle unità di stesse. Tali funzionalità possono essere, di fatto, implementate:

- a livello firmware, in unità specializzate,
- a livello assembler, "embedded" in processori general-purpose con propria memoria e MMU.

In ogni caso, occorre implementare anche le primitive per la cooperazione con i processi eseguibili dalla CPU (*processi interni*). Nel caso più semplice, un'unità di I/O comunica solo con il proprio processo Driver, ma questa non è assolutamente una regola. In particolare, occorre che il processore sia avvertito, *in modo asincrono rispetto all'esecuzione del processo interno corrente*, di eventi comunicati da un'unità di I/O. Il meccanismo delle interruzioni serve proprio a questo scopo.

9.1.3 Interruzioni ed eccezioni

La distinzione fondamentale tra questi due tipi di eventi è la seguente:

- le *eccezioni* sono eventi *sincroni* rispetto all'esecuzione del processo corrente: infatti, esse si verificano in quanto provocati dalla stessa computazione in corso sul processore, la quale è espressa in modo da testare esplicitamente la presenza di ben determinate eccezioni a istanti ben determinati;
- le *interruzioni* sono eventi *asincroni* rispetto all'esecuzione del processo corrente: infatti, essi non sono provocati dalla computazione in corso sul processore, bensì da computazioni che evolvono parallelamente a essa su unità di I/O (o altri processori); non avrebbe senso che il processo eseguito dal processore testasse la presenza di ben determinate interruzioni a istanti scelti arbitrariamente; convenzionalmente si sceglie allora la fine del microprogramma di ogni istruzioni per testare la presenza di una qualsiasi interruzione, proseguendo subito alla chiamata dell'istruzione successiva se non sono attualmente segnale interruzioni presenti.

9.1.4 Trasferimento dati

Distinguiamo due aspetti riguardanti l'architettura di tale sottosistema:

1. il *trasferimento di dati* tra Unità Centrale (CPU, M) e unità di I/O,

2. comunicazioni di *eventi asincroni* da unità di I/O a Processore mediante il meccanismo delle interruzioni.

Il trasferimento dati tra Unità Centrale e unità di I/O avviene quando un processo, PROC, in esecuzione sulla CPU vuole inviare dati a, o ricevere dati da, un'unità di I/O. PROC può essere il Driver dell'unità di I/O oppure un qualsiasi altro processo.

I dati trasferiti possono essere di tipo elementare, implementati come parole o byte, o, più frequentemente, *blocchi* di dati.

Distinguiamo due modelli:

- **Direct Memory Access** (*DMA*): il trasferimento di dati tra I/O e memoria principale, e viceversa, avviene indipendentemente attraverso un canale distinto, utilizzando uno o più *Bus DMA*. Questi trasferimenti, per definizione, avvengono *in parallelo all'elaborazione in corso sul processore*.

- **Memory mapped I/O** (*MMI/O*): poiché a ogni unità di I/O è associata a una certa capacità di memoria locale, questa è *indirizzabile direttamente anche dal processore P*; il trasferimento dati tra P e I/O avviene dunque attraverso letture e scritture da parte di P nelle memorie locali di I/O. In altre parole, un programma in esecuzione su P *vede le unità di I/O come memoria*. Anche in questo caso gli accessi alla memoria locale di I/O da parte di una delle due unità - P o I/O - *avvengono in parallelo* all'elaborazione dell'altra unità.

Nel modello DMA, M è condivisa tra P e le unità di I/O operanti in DMA.

Nel modello MMI/O, ogni memoria locale di unità di I/O è condivisa tra P e tale unità.

9.2 Trattamento delle interruzioni

Come detto più volte, a livello firmware le interruzioni hanno il significato di segnali di richiesta all'arbitro del Bus di I/O di inviare un messaggio di I/O su tale bus. Ai livelli superiori, il significato di un'interruzione è di segnalare, attraverso il messaggio di I/O, un **evento**. Una procedura del processo in

esecuzione, che chiameremo **Handler** dell'interruzione, provvede a compere le azioni necessarie per trattare l'evento stesso.

A ogni evento che le unità di I/O possono comunicare corrisponde un proprio Handler.

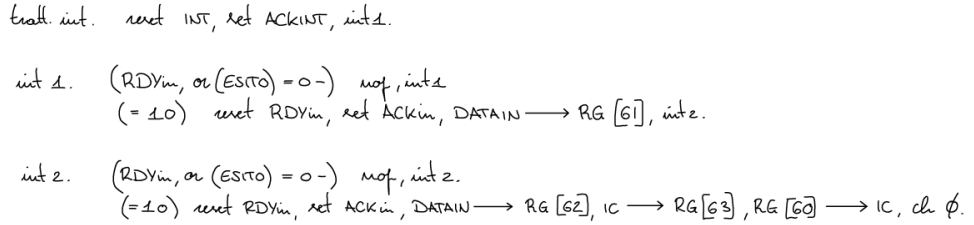
Il trattamento dell'interruzione deve prevede due fasi:

- una prima **fase firmware**, nella quale il processore accetta l'interruzione, attende il messaggio di I/O e usando tale messaggio come parametro, *chiama* una procedura indicata con *Routine di interfacciamento Interruzioni*. Il messaggio di I/O conterrà l'identificatore dell'evento e un dato elementare;

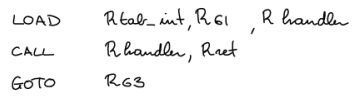
- una seconda **fase assembler**, nella quale viene eseguita la Routine di Interfacciamento Interruzioni e la procedura Handler dell'evento comunicato con l'interruzione.

La Routine di Interfacciamento Interruzioni serve a permettere alla fase firmware di collegare *qualunque* istruzione di *qualunque* processo agli Handler, *svincolando la progettazione firmware del processore della conoscenza a priori degli Handler stessi.*

Fase Firmware



Fase Assembler



Fondamenti di Elaborazione in Parallelo

Per ottimizzare la velocità del nostro modulo di elaborazione si introduce il concetto di **parallelismo**: quando parliamo di parallelismo ci riferiamo alla possibilità di eseguire più core contemporaneamente.

Il parallelismo è diverso dalla concorrenza → esempio: Scheduling a basso livello: "Fa sembrare" all'utente che le operazioni siano eseguite contemporaneamente

↓

c'è il supporto per eseguire le operazioni insieme

MISURE

LATENZA: tempo dall'inizio alla fine di un'attività

TEMPO DI SERVIZIO: tempo fra il momento in cui accetto un'attività e il tempo in cui accetto la prossima.

Nel caso di un elaborazione sequenziale le due misure coincidono

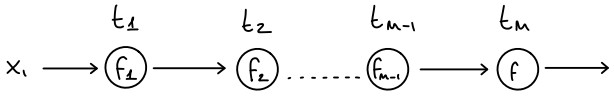
THROUGHPUT (o banda di elaborazione): attività completate × unità di tempo → $B = \frac{1}{T_S}$

TEMPO DI COMPLETAMENTO (relativo a una serie di m valori): tempo dal momento in cui hai avuto x_i al momento in cui ha consegnato $f(x_m)$

L e T_C misurano il tempo totale (di un'attività o di un gruppo di attività)
T_S e B misurano la rapidità nel trattare un gruppo di task

Per un sottoinsieme parallelo con grado di parallelismo n l'EFFICIENZA è definita come $\mathcal{E} = \frac{T_{id}^{(n)}}{T^{(n)}}$

PIPELINE DI n STADI

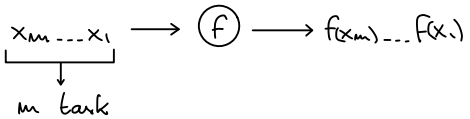


$$f_n(f_{n-1}(\dots(f_2(f_1(x_1)))\dots))$$

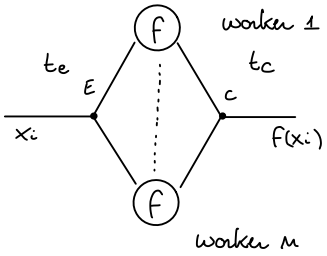
$$L_{PIPE} = \sum_{i=1}^{n} t_i \qquad T_{S\,PIPE} = \max\{T_{Si}\} = \max\{t_i\}$$

$$T_{C\,PIPE} = m \cdot \max\{t_i\} + (L_{PIPE} - \max\{t_i\}) = m \cdot T_S + (L - T_S)$$

FARM



→ Se sequenziale: $T_S = L = t_w$ e $T_C = m \cdot t_w$



Ipotesi:

$t_e \approx t_c << \frac{t_w}{n}$

$(\exists k \mid n = kn)$

$$L_{FARM} = t_e + t_w + t_c \qquad T_{S\,FARM} = t_e \qquad T_{C\,FARM}(n) = (t_e + n t_c) + \frac{n}{n} t_w \approx \frac{n}{n} t_w$$

Capitolo 10

Architetture con Parallelismo a Livello di Istruzioni

10.1 Architettura della CPU pipeline

Il concetto alla base di quest'architettura è la parallelizzazione della CPU mediante la *parallelizzazione dell'interprete firmware* delle istruzioni, eseguite dal processore con la collaborazione delle altre unità della CPU stessa. È noto che tale interprete costa di più fasi, alcune delle quali eventualmente effettuate in uno stesso ciclo di clock da parte di un processore elementare:

1. chiamata istruzione, contenente la lettura dell'istruzione dalla memoria con la cooperazione di MMU e memoria,
2. decodifica dell'istruzione, preparazione degli indirizzi di eventuali operandi in memoria, preparazione degli eventuali operandi presenti in registri generali,
3. eventuale lettura degli operandi in memoria, ancora con la cooperazione di MMU e memoria,
4. fase di esecuzione vera e propria e aggiornamento del contatore istruzioni,
5. eventuale scrittura dei risultati nei registri generali o in memoria, con la cooperazione di MMU e memoria,
6. trattamento delle interruzioni.

Queste fasi hanno un ordinamento totale e operano concettualmente su uno *stream* d'istruzioni: di conseguenza l'interprete si presta a una parallelizzazione in *pipeline*.

10.1.1 Memoria e MMU

La fase 1 e la fase 3 (5) necessitano, per essere eseguite in parallelo su istruzioni diversa, di due unità di memoria distinte, una dedicata a memorizzare solo istruzioni (*Memoria Istruzioni*, **IM**) e l'altra a memorizzare solo dati (*Memoria Dati*, **DM**). Entrambe sono dotate di propria MMU.

IM e DM sono partizioni disgiunte della *cache primaria*.

L'unità MINF (**Memory Interface**) mette in comunicazione, e arbitra, le richieste per il trasferimento di blocchi nei confronti del livello superiore della memoria esterna.

Il Bus di I/O è connesso alla MMU_D, principalmente allo scopo di implementare la tecnica del Memory Mapped I/O.

Per non introdurre comunicazioni a domanda-risposta con il processore nella chiamata dell'istruzione, deleghiamo a IM il compito di leggere, senza soluzione di continuità, istruzioni a indirizzi logici consecutivi, generando di fatto lo ***stream d'ingresso*** della pipeline. A tale scopo, MMUI contiene una copia (IC1) del contatore istruzioni (IC) del processore.

I contenuti di IC_1 e IC sono identici e consistenti *finché* non è eseguita un'istruzione di salto che fa effettivamente saltare, dopodiché verrà ripristinato il nuovo stato consistente.

10.1.2 Processore

Lo stream d'istruzione viene raccolto dallo stadio che chiameremo *Unità Istruzioni* (**IU**), delegato alle fasi 2, parte della 4 e 6. IU *decodifica* ogni istruzione e delega ad altri stadi della pipeline il suo proseguimento.

IU contiene la copia "affidabile" del *contatore istruzioni IC*. Di conseguenza, è naturale pensare che tutte le ***istruzioni di salto*** siano eseguite interamente da IU. In caso di salto, è assolutamente necessario introdurre una prima *retroazione nella pipeline*, che quindi "non è una pipeline pura": IU invia l'indirizzo logico di salto a IM che *aggiorna IC1 e lo stream di istruzioni riprende a partire dal nuovo indirizzo*. È inevitabile che, in caso di salto, IM possa aver già inviato a IU una o più istruzioni, che quindi dovranno essere scartate finché IU non riceve l'istruzione che ha richiesto esplicitamente. Questo fenomeno introduce una prima *degradazione* delle prestazioni.

Inoltre, IU provvede a *calcolare gli indirizzi di memoria* delle `Load` e `Store` e a chiedere a DM l'esecuzione delle rispettive operazioni di lettura e scrittura. Nel caso di `Load`, il valore letto verrà inviato da DM a uno stadio successivo della pipeline (EU), nel caso di Store IU provvederà anche a inviare a DM il dato da scrivere. Di conseguenza, la `Store` *termina nella DM*.

Poiché il contatore istruzioni è contenuto in IU, quest'ultima è collegata a UNINT e provvede ad effettuare il *trattamento delle interruzioni*.

Tutte le *istruzioni aritmetico-logiche* sono delegate allo stadio chiamato Unità Esecutiva (EU), che provvede a scrivere il risultato nel registro generale destinazione (fasi 4, 5). La `Load` è vista come un'istruzione "identità" con un operando in memoria, ragione per cui è EU a occuparsi di ricevere il dato in memoria e a scriverlo nel registro destinazione.

Tutte le comunicazioni tra unità della pipeline avvengono su collegamenti dedicati con protocollo asincrono.

Occupiamoci ora dei Registri Generali. Una soluzione efficiente consiste nel tenerne una doppia copia: una in IU (RG_1) e l'altra in EU (RG). IU necessità solo di *leggere* i registri generali di RG_1. Le modifiche dei registri generali, in istruzioni aritmetico-logiche e di `Load`, sono effettuate da EU su RG e sono, da quest'unità, comunicate a IU che provvede a scrivere i valori ricevuti in RG_1. Di conseguenza, la copia sempre aggiornata è quella in EU (RG), mentre ci saranno dei momenti in cui la copia in IU (RG_1) non sarà aggiornata: un meccanismo di sincronizzazione provvede al corretto funzionamento di IU, in particolare a impedire a IU la lettura di registri non ancora aggiornati.

Oltre alla *retroazione dovuta alla presenzadi istruzioni di salto*, un'altra seria causa di retroazione è data dalle ***dipendenze logiche sui registri generali***: IU può voler leggere in RG_1 il contenuto di un registro che deve ancora essere aggiornato dall'esecuzione in EU di un'istruzione precedente dello stream. Entrambe le retroazioni

- richiedono che, per ragioni di correttezza, vengano introdotti degli opportuni meccanismi di sincronizzazione che ordinino gli eventi in modo consistente con la semantica del programma,
- rappresentano cause di degradazione delle prestazioni.

10.2 Architettura pipeline astratta

Adotteremo una visione semplificata dell'architettura, che rappresenta l'***architettura astratta***: questa è costituita da soli quattro stadi, **IM**, **IU**, **DM**, **EU**.

- Sappiamo sia IM sia DM possono essere realizzati come pipeline di 2 o 3 stadi; si tratta di strutture pipeline "pure", che non introducono alcun problema di retroazione o dipendenza locica. Considerare sia IM sia DM come un unico sottosistema, avente il tempo di servizio

effettivo uguale a quello della scrittura interna, semplifica la valutazione delle prestazioni senza nascondere eventi significativi o introdurre approssimazioni;

- come in precedenza, supporremo che il registro IC sia presente in doppia copia in IU e IM e che i registri generali siano presenti in doppia copia in IU ed EU. In entrambi i casi, opportuni meccanismi di sincronizzazione assicurano la correttezza di funzionamento quando si verifichino momentanee inconsistenze.

L'architettura astratta ha inoltre le seguenti caratteristiche:

- *ogni stadio (sottosistema) ha* ***tempo di servizio ideale per istruzione***:
$$t = T_{id} = 2\tau$$
- *tutti i canali di comunicazione sono asincroni con grado di asincronia uguale a uno.*

10.3 Implementazione delle unità della CPU pipeline

Risolviamo anzitutto il problema della *sincronizzazione tra IU e IM in seguito a istruzioni di salto*. La richiesta da IU a IM è accompagnata da un *identificatore unico*, che IM associa all'istruzione inviata a IU, mantenendo lo stesso valore dell'identificatore in tutte le successive richieste finché IU non effettua una nuova richiesta. A ogni ricezione di istruzione, IU confronta il proprio valore dell'identificatore con quello ricevuto: in caso di concordanza l'istruzione è valida, altrimenti IU scarta l'istruzione ricevuta e attende altre istruzioni finché non trova concordanza.

10.3.1 Registri generali

La sincronizzazione per il corretto uso dei registri generali RG_1 può essere così implementata all'interno dell'unità IU:

- a ogni registro RG_1 è associato un *semaforo interno*, non negativo, inizializzato a zero;
- per ogni istruzione aritmetico-logica o di `Load`, IU incrementa di uno il semaforo associato al registro destinazione in RG_1;

- ogni volta che scrive in un registro di RG_1 su richiesta di EU, IU decrementa di uno il semaforo associato;
- quando IU intende leggere un registro di RG_1 la lettura può avere luogo solo se il semaforo associato al registro stesso è uguale a zero; in caso contrario, IU:

  1. si mette in attesa degli aggiornamenti del valore del registro, finché il semaforo non ritorna a zero

     oppure

  2. ricorda la situazione dell'istruzione corrente e continua a servire nuove istruzioni in arrivo.

10.3.2 Funzionamento in-order e out-of-order

Consideriamo le due situazioni 1), 2) descritte a proposito dell'attesa che un registro generale divenga aggiornato.

Il funzionamento 1) non introduce modifiche all'ordinamento delle istruzioni elaborate in IU rispetto allo stream d'ingresso. Le architetture che adottano questo funzionamento sono dette "**in-order**".

Il funzionamento 2) introduce modifiche all'ordinamento delle istruzioni elaborate dalla IU rispetto allo stream d'ingresso: alcune istruzioni, purché abilitate (tutti i registri utili a IU sono aggiornati), "scavalcano" altre istruzioni in attesa. Le architetture che adottano questo funzionamento sono dette "**out-of-order**".

10.4 Ottimizzazioni

Avendo individuato le cause di degradazione delle performance, le ottimizzazioni tendenti a ridurne l'effetto devono occuparsi di:

- *per le degradazioni dovute ai salti*:
  - ridurre la probabilità di salto;
  - sfruttare i tempi morti, introdotti dalle bolle, per effettuare lavoro utile;
  - "predire" l'esito di salti condizionati in macchine "out-of-order";
- *per le degradazioni dovute alle dipendenze logiche*:
  - ridurre la probabilità di dipendenza logica;
  - aumentare la distanza delle dipendenze logiche.

10.4.1 Minimizzazioni delle degradazioni dovute ai salti

Ciò significa ridurre la frequenza stessa con cui vengono utilizzate istruzioni di salto. Si tratta di tipiche ottimizzazioni *a tempo di compilazione*, come:

- espansione di *macro*, al posto di subroutine o procedure,
- *loop unfolding*,

che comportano un aumento della memoria *logica* del processo. In termini di memoria *fisica*, queste tecniche possono comportare un aumento del working set del processo.

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

10.4.2 Minimizzazione delle degradazioni dovute alle dipendenze logiche

A tempo di compilazione possono essere applicate tecniche dirette ad "allontanare" il più possibile le istruzioni che sono legate da dipendenze logiche. Questo comporta, principalmente, l'adozione di tecniche di *spostamento di codice*.

10.4.3 Unità Esecutiva parallela

L'Unità Esecutiva può essere definita secondo lo schema seguente, contenente:

- una Unità Funzionale pipeline *moltiplicatore/divisore* in virgola fissa,
- una Unità Funzionale pipeline *addizionatore* in virgola mobile,
- una Unità Funzionale pipeline *moltiplicatore/divisore in virgola mobile*, che può anche includere altre operazioni in virgola mobile, come la radice quadrata,
- l'unità EU_{MASTER}.

L'unità indicata con EU_{MASTER} interfaccia IU e DM e *contiene la copia principale dei registri generali* (RG) *e dei registri in virgola mobile* RF. Ricevendo una istruzione da IU, il funzionamento è il seguente:

- se l'operazione richiesta da IU è una aritmetica corta o una `LOAD`, la *esegue direttamente* (per la `LOAD` attende il dato da DM) e invia a IU il valore del registro generale destinazione e il suo indirizzo;
- se l'operazione è lunga in virgola fissa, oppure è in virgola mobile, la distribuisce all'Unità Funzionale corrispondente *insieme ai valori degli operandi* letti dai registri RG o RF rispettivamente.