ARCHITETTURE E SISTEMI OPERATIVI

Questi sono i miei appunti di AESO

La parte di Architetture è stata presa completamente dal libro:

- Architettura degli elaboratori Harris, Harris ARM Edition 1.

La parte di Sistemi Operativi è stata invece fatta seguendo le slides del professor Torquati.

La parte di SO è stata scritta in un secondo momento rispetto a quella di architetture e presenta alcune diversità a livello grafico quali:

- sottolineature
- assenza dei capitoli (slides)
- assenza di link (tutti gli appunti sono stati scritti su Obsidian che permette di linkare le note fra di loro)

Spero vi possano essere utili come lo sono stati per me! :)

Marco Briglia.

1.2-L'arte di Gestire le complessità

L'Astrazione

L'Astrazione è la tecnica fondamentale per imparare a controllare la complessità e consiste nel nascondere dettagli quando questi non sono importanti

La Disciplina

è l'atto di restringere intenzionalmente le scelte di progetto in modo da poter lavorare in maniera più produttiva a un livello di astrazione più alto

Le 3 Y

- *Gerarchia*: implica dividere un sistema in moduli e successivamente suddividere ulteriormente ognuno di questi moduli finchè i pezzi che li compongono non sono facili da comprendere
- *Modularità*: implica che i moduli abbiano funzioni e interfacce ben definite così da connettersi tra di loro in maniera semplice
- *Regolarità*: cerca l'uniformità tra i moduli. I moduli più comuni vengono riutilizzati più volte riducendo il numeri di moduli da progettare

1.4-Sistemi Numerici

Numeri Decimali

È il sistema numerico che si impara fin dalle elementari e che usiamo nella matematica classica. I numeri vanno da 0,..,9 e la concatenazione dei simboli porta alla creazione di numeri più grandi secondo questa regola:

$$9742_{10} = 9 * 10^3 + 7 * 10^2 + 4 * 10^1 + 2 * 10^0$$

Numeri Binari

È il sistema dei numeri in base 2. I valori che possono assumere sono quelli dei bit ovvero (0,1).

- Se 0 $\rightarrow$ False
- Se 1 $\rightarrow$ True
  Sono rappresentati sotto questa forma:

$$10110_2 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 22_{10}$$

> *Per convertire un numero da decimale a binario basta cercare la potenza di 2 più vicina che non superi il mio numero e mettere 1 in quella posizione, poi guardare se il resto della sottrazione è $\geqslant$ alla potenza di 2 successivamente minore. Se non lo è la posizione verrà occupata da uno 0, altrimenti 1. Si procede così fino ad aver convertito totalmente il numero.*

Numeri Esadecimali

Numeri in base 16 che vanno dallo 0,..9,A,B,C,D,E,F comprendendo le prime 6 lettere dell'alfabeto. Si scrivono in questa forma:

$$2ED_{16} = 2 * 16^2 + E * 16^1 + D * 16^0 = 749_{10}$$

> *Per convertire in binario bisogna ricordarsi che ogni cifra esadecimale corrisponde a 4 bit binari, quindi:*

$$2ED_{16} = 2->0010 + E->1110 + D->1101 = 001011101101_2$$

Byte, Nibble e Word

Un insieme di 8 bit è chiamato *byte* e rappresenta una di $2^8 = 256$ possibilità. è la grandezza di misura utilizzata per misurare la grandezza degli oggetti immagazzinati nelle memorie dei calcolatori.

Un gruppo di 4 bit è invece chiamato *nibble* e rappresenta una di $2^4 = 16$ possibilità. Una cifra esadecimale rappresenta un *nibble* mentra 2 un *byte*. I microprocessori gestiscono dat in gruppi di bit chiamati *word* la cui grandezza dipende dall'architettura del processore. Ad oggi operano su *word* di 64 bit.

All'interno di un gruppo di bit, il bit che si trova nella colonna di peso 1 è chiamato *bit meno significativo* (lsb, less significant bit) e il bit che si trova dalla parte opposta viene chiamato *bit più significativo* (msb, most significant b). Allo stesso modo in una parola esiste il *byte più significativo* e il *byte meno significativo* composti da 8 bit.

Somma binaria

Per sommare 2 numeri binari basta metterli in colonna e sommare le cifre nella stessa colonna. 1+1=0 con riporto di 1, 1+0=1, 0+1=1, 0+0=0.

> *ex.*
> *10011+*
> *00111=*
> *======*
> *11010*

Numeri Binari Relativi

Esistono due modi per rappresentare i numeri binari relativi (ovvero con il segno) e sono:

- *modulo e segno*
- *complemento a 2*

Numeri in modulo e segno

Un numero espresso in modulo e segno a N bit utilizza il bit più significativo per esprimere il segno e i restanti N-1 bit per esprimere il modulo del numero. Il bit più significativo ha valore:

- 1 $\rightarrow$ *negativo*
- 0 $\rightarrow$ *positivo*

Ha un intervallo di variabilità pari a $[-2^{N-1}+1, 2^{N-1}-1]$, però per lo 0 esistono 2 rappresentazioni e diventa complicato. Inoltre non è possibile svolgere la somma con il classico metodo.

Numeri in complemento a 2

Qua lo 0 ha una sola rappresentazione ed è possibile effettuare la somma con il classico metodo. Lo 0 èscritto come una sequenza di tutti 0 $\rightarrow$ $00..000_2$, il massimo numero positivo ha il bit più significativo a 0 e il resto tutto a 1 $\rightarrow$ $011..111_2 = 2^{N-1}$ mentre il numero più negativo è composto da un 1 seguito da tutti 0 $\rightarrow$ $100..000_2 = -2^{N-1}$ e il numero -1 è scritto come una sequenza di tutti 1 $\rightarrow$ $11..111_2$.
Il bit più significativo è sempre il bit di segno e può essere invertito con l'operazione del *complemneto a due*.
Consiste nell'invertire tutti i bit del numero e poi aggiungere 1 al bit meno significativo.

> *La sottrazione è un operazione che avviene facendo il complemento a 2 del secondo numero e poi facendo la somma classica fra 2 numeri binari.*

La somma tra due numeri in complemento a 2 potrebbe risultare in un traboccamento se supera l'intervallo $[-2^{N-1}, 2^{N-1}]$ mentre una sottrazione non genererà mai questo problema.

1.5-Porte Logiche

Cosa sono le porte logiche?

Le Porte logiche (logic gates) sono semplici circuiti digitali che utilizzano uno o più ingressi binari per produrre un'uscita binaria. Solitamente gli ingressi vengono disegnati a sinistra o in alto mentre l'uscita a destra o in basso.
Per gli ingressi solitamente vengono utilizzate le prima lettere dell'alfabeto mentre per l'uscita la lettera Y.
Le relazioni tra ingressi e uscita possono essere descritti attraverso una *tabella di verità* o con un *espressione booleana*.

Ci sono diversi tipi di porte logiche:

La porta NOT

Una porta NOT ha un ingresso A, e un'uscita Y. L'uscita è esattamente il contrario rispoetto all'ingresso, quindi se entra TRUE → FALSE mentre se entra FALSE → TRUE.

A		Y
0		1
1		0

La porta AND

La porta AND genera TRUE all'uscita Y solo se entrambi A e B sono TRUE, altrimenti genera FALSE.

A	B		Y
0	0		0
0	1		0
1	0		0
1	1		1

La porta OR

La porta OR produce all'uscita Y TRUE se A, oppure B, oppure entrambi hanno valore TRUE, altrimenti FALSE.

A	B		Y
0	0		0
0	1		1
1	0		1
1	1		1

Buffer

Riproduce semplicemente il valore di ingresso.

A		Y
0		0
1		1

Altre porte logiche a 2 ingressi

XOR

Restituisce TRUE se A oppure B, ma non entrambi hanno valore TRUE. Una porta XOR a N ingressi restituisce TRUE se si ha un numero dispari di ingressi TRUE

A	B		Y
0	0		0
0	1		1
1	0		1
1	1		0

XNOR

Restituisce TRUE solo se si ha un numero pari (compreso 0) di ingressi TRUE

A	B		Y
0	0		1
0	1		0
1	0		0
1	1		1

NAND

Esegue AND e NOT. Restituisce sempre TRUE ad eccezione di quando A e B sono TRUE.

A	B		Y
0	0		1
0	1		1
1	0		1
1	1		0

NOR

Esegue OR e NOT. Restituisce TRUE se né A né B sono TRUE.

A	B		Y
0	0		1
0	1		0
1	0		0
1	1		0

Porte a ingressi multipli

Ne esistono molte ma le più comuni sono:

- AND
- OR
- XOR
- NAND
- NOR

- XNOR

AND produce TRUE solo se tutti i suoi ingressi hanno valore TRUE, OR invece se almeno uno dei suoi ingressi ha valore TRUE.

2.2-Espressioni Booleane

Cosa sono le espressioni booleane?

Le espressioni booleane si basano su variabili che possono assumere solo i valori TRUE e FALSE.

Terminologia

Il *complemento* di una variabile A è il suo negato $\bar{A}$.
L'AND è definito come *prodotto logico* o *implicante*.
Un minitermine è il prodotto di tutti gli ingressi di una funzione, quindi $A\bar{B}\bar{C}$ è un minitermine della funzione che prende come input le 3 variabili A,B,C; $\bar{A}B$ non lo è perchè non considera C.
L'OR invece è definito come *somma logica* o *implicato*.
Un maxitermine è la somma di tutti gli ingressi della funzione, quindi $\bar{A}+B+C$ è un maxitermine.

> *L'ordine delle operazioni è importante:*
>
> - *NOT ha la massima precedenza*
> - *AND segue NOT*
> - *OR segue AND*

Forma somma di prodotti

Una tabella della verità con N ingressi contiene 2^N righe e ognuna di esse è associata ad un minitermine che è TRUE per quella riga.

A	B	Y	minitermine	nome del minitermine
0	0	0	$\bar{A}\bar{B}$	m_0
0	*1*	*1*	$\bar{A}B$	m_1
1	0	0	$A\bar{B}$	m_2
1	1	0	AB	m_3

è possibile scrivere un'espressione booleana tramite la somma di tutti i minitermini in corrispondenza dei quale Y → TRUE.

> *ex.*

A	B	Y	minitermine	nome del minitermine
0	0	0	$\bar{A}\bar{B}$	m_0
0	*1*	*1*	$\bar{A}B$	m_1
1	0	0	$A\bar{B}$	m_2
1	*1*	*1*	AB	m_3

$Y = \bar{A}B + AB$
→ questa espressione è chiamata *forma canonica somma di prodotti* e può essere scritta così:

$$F(A,B) = \sum(m_1, m_2)$$

oppure:

$$F(A,B) = \sum(1,3)$$

> *(1,3) perchè nella riga 1 e nella riga 3 della tebella di verità abbiamo come risultato TRUE (partono da 0).*

Forma prodotto di somme

Un modo alternativo per scrivere le funzioni booleane è la *forma canonica prodotto di somme*. Ad ogni riga della tabella corrisponde un maxitermine che è FALSE per quella riga. La forma canonica prodotto di somme può essere scritta con la notazione *pi greco* utilizzando il simbolo di produttoria, $\prod$.

A	B	Y	maxitermine	nome del maxitermine
0	*0*	*0*	$A+B$	M_0
0	1	1	$A+\bar{B}$	M_1
1	*0*	*0*	$\bar{A}+B$	M_2
1	1	1	$\bar{A}+\bar{B}$	M_3

$Y = (A+B) + (\bar{A}+B)$

Può essere scritta con la produttoria così:

$$Y = \prod(M_0, M_2)$$

oppure:

$$Y = \prod(0, 2)$$

nb

La somma di prodotti produce un'espressione più corta quando poche uscite hanno TRUE, analogamente il prodotto di somme quando ci sono poche uscite FALSE.

2.3-Algebra Booleana

Cos'è l'algebra booleana?

L'algebra booleana è utilizzata per semplificare le espressioni booleane e le regole sono simili a quelle dell'algebra ordinaria. Si basa su un'insieme di postulati che per definizione sono considerati corretti e obbediscono al *principio di dualità*:

- Se i simboli 0 e 1 e gli operatori $\cdot$ (AND) e $+$ (OR) sono scambiati fra di loro, l'affermazione rimane corretta.

Postulati

	Postulato		Forma Duale	Nome
A1	B=0 se B≠1	A1'	B=1 se B≠0	Alegebra binaria
A2	$\bar{0}$=1	A2'	$\bar{1}$=0	NOT
A3	0 $\cdot$ 0 = 0	A3'	1 + 1 = 1	AND/OR
A4	1 $\cdot$ 1 = 1	A4'	0 + 0 = 0	AND/OR
A5	0 $\cdot$ 1 = 1 $\cdot$ 0 = 0	A5'	1 + 0 = 0 + 1 = 1	AND/OR

Teoremi a una variabile

	Postulato		Forma duale	Nome
T1	B $\cdot$ 1 = B	T1'	B + 0 = B	Identità
T2	B $\cdot$ 0 = 0	T2'	B + 1 = 1	Nullo
T3	B $\cdot$ B = B	T3'	B + B = B	Idempotenza
T4	$\bar{\bar{B}}$ = B			Involuzione
T5	B $\cdot$ $\bar{B}$ = 0	T5'	B + $\bar{B}$ = 1	Complementi

Teoremi a più variabili

	Postulato		Forma duale	Nome
T6	B $\cdot$ C = C $\cdot$ B	T6'	B + C = C + B	Commutatività

	Postulato		Forma duale	Nome
T7	(B · C) · D = B · (C · D)	T7'	(B + C) + D = B + (C + D)	Associatività
T8	(B · C) + (B · D) = B + (C · D)	T8'	(B + C) · (B + D) = B + (C · D)	Distributività
T9	B · (B + C) = B	T9'	B + (B · C) = B	Assorbimento
T10	(B · C) + (B · $\bar{C}$) = B	T10'	(B + C) · (B + $\bar{C}$) = B	Combinazione
T11	(B · C) + ($\bar{B}$ · D) + (C · D) = (B · C) + ($\bar{B}$ · D)	T11'	(B + C) · ($\bar{B}$ + D) · (C + D) = (B + C) · ($\bar{B}$ + D)	Consenso
T12	$\overline{B_0 \cdot B_1 \cdot B_2 \cdot \ldots}$ = $(\bar{B}_0 + \bar{B}_1 + \bar{B}_2 + \ldots)$	T12'	$\overline{B_0 + B_1 + B_2 + \ldots}$ = $(\bar{B}_0 \cdot \bar{B}_1 \cdot \bar{B}_2 \cdot \ldots)$	Teorema di De Morgan

Bolla

Il circoletto di negazione viene chiamato *bolla*. Le regole alla base dello spostamento della bolla sono:

- Spingere una bolla all'indietro (dall'uscita all'ingresso) o in avanti (viceversa) trasforma una porta AND in una porta OR e viceversa;
- se si spinge una bolla dall'uscita verso gli ingressi, questa viene trasferita a ognuno di essi;
- se si spingono tutte le bolle degli ingressi di una porta verso la sua uscita, quest'ultima avrà una bolla sola.

Semplificare le espressioni

I teoremi dell'algebra booleana sono utili per semplificare le espressioni booleane.

> *ex.*
> $Y = \bar{A}B + AB \quad per\ il\ T10 \quad \longrightarrow \quad Y = B$

Un'espressione è definita *minima* se utilizza il minor numero possibile di implicanti.
Un implicante è detto *implicante primo* se non può essere combinato con nessun altro elemento all'interno dell'espressione per formare

un nuovo implicante con un minor numero di letterali.
In un'espressione minima, gli implicanti devono essere tutti implicanti primi.

2.7-Mappe di Karnaugh

Cosa sono le Mappe di Karnaugh?

Le mappe di Karnaugh (K-map) sono un modello grafico di semplificazione delle espressioni booleane.
Sono uno strumento molto efficace per risolvere problemi con al massimo 4 variabili.
La riga in alto mostra le possibili 4 combinazioni di valori di ingresso A e B, mentre i 2 possibili valori di C sono riportati della colonna di sinistra. Ogni quadrato corrisponde a una riga della tabella di verità e contiene il valore dell'uscita Y corrispondente a quella riga.

> nb
>
> L'ordine 00,01,11,10 è chiamato *codice Gray* ed è utilizzato nelle mappe perchè preserva la proprietà la quale gli elementi adiacenti differiscono solo per una variabile.

Pensare in cerchi

C\AB	00	01	11	10
0	(1)	0	0	0
1	(1)	0	0	0
C\AB	00	01	11	10
0	$\overline{ABC}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	ABC	$A\bar{B}C$

è sempre possibile utilizzare l'algebra booleana per semplificare le e spressioni:

$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C = \bar{A}\bar{B}(\bar{C} + C) = \bar{A}\bar{B}$$

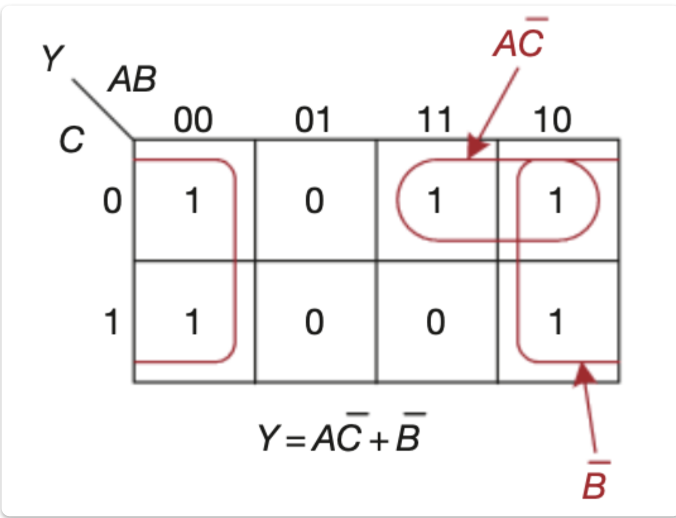
Ma usando la mappa di Karnaugh semplifichiamo graficamente

cerchiando tutti gli 1 nei riquadri adiacenti e bisogna scrivere gli implicanti corrispondenti al quadrato dove è presente 1. Il risultato è analogo ma è più veloce vederlo.

Minimizzazione logica con le mappe di Karnaugh

Le regole per torvare l'espressione minima a partire da una mappa di Karnaugh sono:

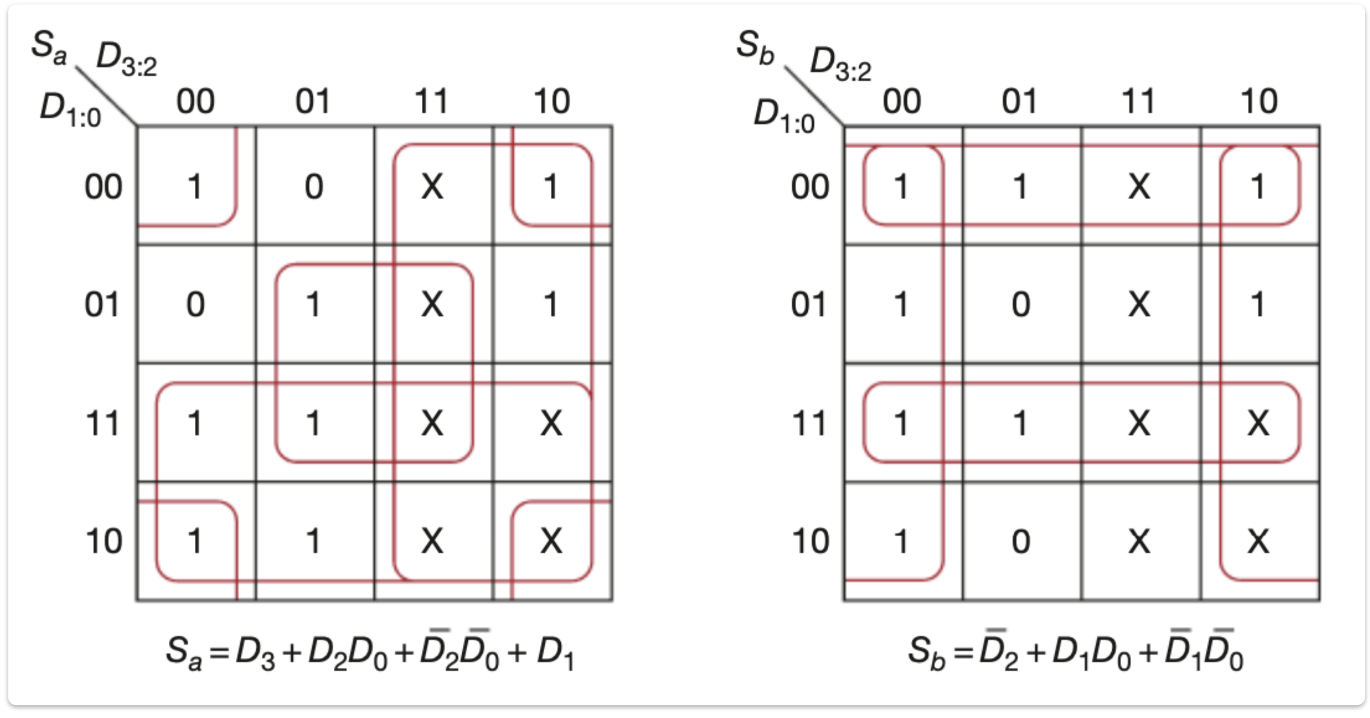
- Utilizzare il minor numero possibile di cerchi per includere tutti gli 1;
- Tutti i riquadri racchiusi in un cerchio devono contenere solamente 1;
- Ogni cerchio deve includere un numero di riquadri che sia una potenza di 2 (1,2,4,8...) in qualsiasi direzione;
- Ogni cerchio deve essere il più largo possibile;
- è possibile disegnare un cerchio che avvolga le estremità della mappa di Karnaugh;
- Un 1 in una mappa di Karnaugh può essere cerchiato più di una volta, se questa operazione permette di utilizzare un minor numero di cerchi.



Indifferenze

Vengono indicate con il simbolo X, che significa che il valore che può assumere è sia 0, sia 1.

In una mappa di Karnaugh, una X permette di aumentare ulteriormente la minimizzazione logica: infatti, queste possono essere incluse nei cerchi se l'operazione può essere utile a coprire più 1 con un numero minore di cerchi, mentre da ignorare se non sono d'aiuto.



2.8-Blocchi costitutivi combinatori

Multiplexer

I *multiplexer* (MUX) sono fra le reti più comunemente usate. Sono in grado di scegliere un'uscita a partire da un certo numero di ingressi basandosi sul valore del *segnale di selezione*.

Multiplexer 2:1

Un mux 2:1 ha 2 ingressi dati D_0, D_1, un *ingresso di selezione* S e un'uscita Y.
Il multiplexer sceglie tra i 2 ingressi di dato a seconda del valore S:

- Se $S = 0, Y = D_0$
- Se $S = 1, Y = D_1$

> *S viene chiamato anche segnale di controllo proprio perchè controlla l'uscita del multiplexer*

Multiplexer più grandi

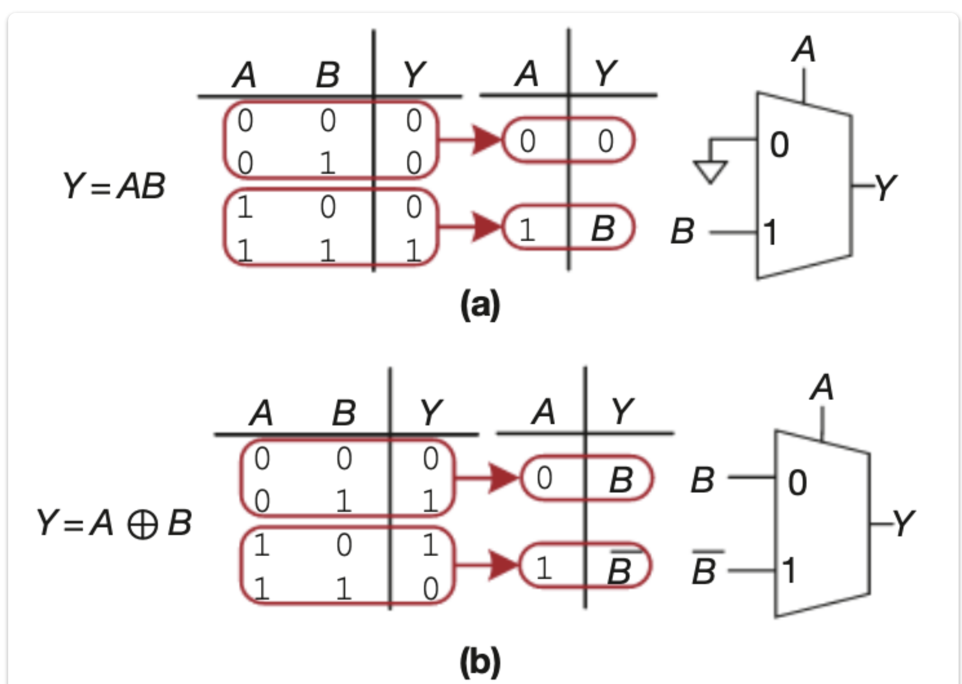
Un 4:1 possiede 4 entrate e un'uscita sola e di conseguenza tutti quelli più grande come i 8:1,16:1... sono composizioni di multiplexer più piccoli che si rifanno al 2:1.

> nb
>
> Un mux N:1 necessita di log_2N ingressi di selezione S

Logica a multiplexer

I multiplexer possono essere utilizzati come tabelle di ricerca (lookup table) per eseguire funzioni logiche.

Con un po' di astuzia è possibile dimezzare la taglia del multiplexer utilizzando solo un multiplexer a 2^{N-1} ingressi per eseguire una qualsiasi funzione logica a N ingressi.
La strategia di base è fornire uno dei letterali di ingresso della funzione, insieme agli 1 e 0, agli ingressi di dato del multiplexer come mostrato qua sotto:

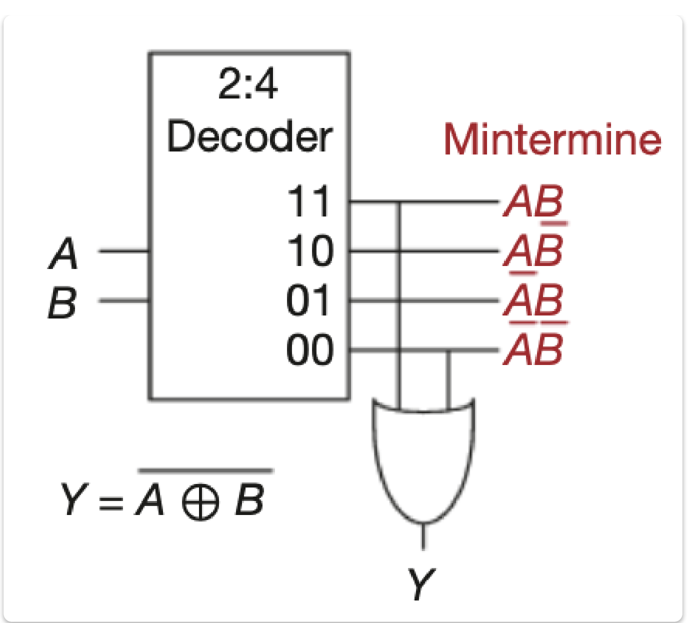


Decoder

Un decoder ha N ingressi e 2^N uscite e attiva una delle sue uscite a seconda della combinazione dei valori di ingresso.
Le uscite sono dette *one hot* perchè solo un'uscita è "calda" (ALTA) in ogni momento

Logica a decoder

Ogni uscita di un decoder rappresenta un minitermine.
Una funzione a N ingressi con un numero M di 1 in un'unica tabella della verità può essere costruita con un decoder $N:2^N$ e una porta OR a M ingressi connessa a tutti i minitermini associati ad un 1 in uscita nella tabella.
Qua sotto viene riportata la funzione XNOR a 2 ingressi:


2:4
Decoder
Mintermine
11
10
01
00
A
B
AB
$A\overline{B}$
$\overline{A}B$
$\overline{A}\,\overline{B}$
$Y = \overline{A \oplus B}$
Y

2.9-Temporizzazioni

Cosa sono le Temporizzazioni?

Uno dei problemi più importanti legato alle reti è la *temporizazione* (timing), in altre parole fare in modo che la rete funzioni velocemente.
Il *Ritardo* (delay) è il tempo tra un cambiamento in un ingresso e il conseguente adattamento dell'uscita di un buffer.

Ritardi di propagazione e di contaminazione

La logica combinatoria è caratterizzata da:

- *Ritardo di propagazione*: (t_{pd}) è il tempo massimo dal cambiamento in ingresso al momento in cui l'uscita raggiunge il suo valore finale
- *Ritardo di contaminazione*: (t_{cd}) è il tempo minimo in cui cambia un ingresso al momento in cui una qualsiasi uscita inizia in suo processo di adattamento del suo valore
- *Percorso critico*: è il percorso seguito dal segnale tra l'ingresso A o B fino all'uscita Y. Limita la velocità alla quale la rete opera
- *Percorso minimo*: è il percorso seguito dal segnale tra l'ingresso D è l'uscita Y. è il percorso più breve (e quindi anche il più veloce) perchè attraversa solo una porta dall'ingresso fino all'uscita.

Alee

Le *Alee* sono molteplici cambiamenti in uscita causati da un singolo cambiamento in ingresso.
Si presenta un'Alea quando una singola variazione in una singola

variabile di ingresso attraversa i confini di due implicanti primi in una mappa di Karnaugh.

2.10-Riassunto

Una rete digitale è un modulo con ingressi e uscite a valori discreti e una specifica che descrive la funzione e la temporizzazione del modulo.

La funzione di una rete combinatoria può essere data da una tabella delle verità o da un'espressione booleana. L'espressione booleana relativa a una qualsiasi tabella delle verità può essere ottenuta in maniera sistematica utiliz- zando la forma somma di prodotti o la forma prodotto di somme. Nella sua forma somma di prodotti, l'espressione è scritta come la somma (OR) di uno o più implicanti, cioè prodotti (AND) dei letterali. I letterali, a loro volta, sono la forma diritta o negata delle variabili d'ingresso della funzione.

Le espressioni booleane possono essere semplificate utilizzando le regole dell'algebra booleana. In particolare, esse possono essere semplificate nella loro forma minima somma di prodotti combinando tra loro gli implicanti che differiscono solo di un letterale nella forma diritta e negata: PA + PA = P. Le mappe di Karnaugh sono strumenti grafici utilizzati per minimizzare espressioni che hanno fino a quattro variabili.

Le porte logiche sono connesse tra loro per creare reti combinatorie che eseguono la funzione richiesta. Una qualsiasi funzione in forma somma di prodotti può essere costruita utilizzando la logica a due livelli: in particolare, le porte NOT formano i complementi degli ingressi, le porte AND i prodotti e le porte OR formano la somma.

Le porte logiche vengono inoltre combinate per creare reti più grandi, come i multiplexer, i decoder e le reti a priorità. Un multiplexer ha la capacità di scegliere un ingresso di dato basandosi su un ingresso di selezione, un decoder attiva una delle uscite a 1 in base alla configurazione presente agli ingressi, mentre una rete a priorità produce un'uscita che indica l'ingresso

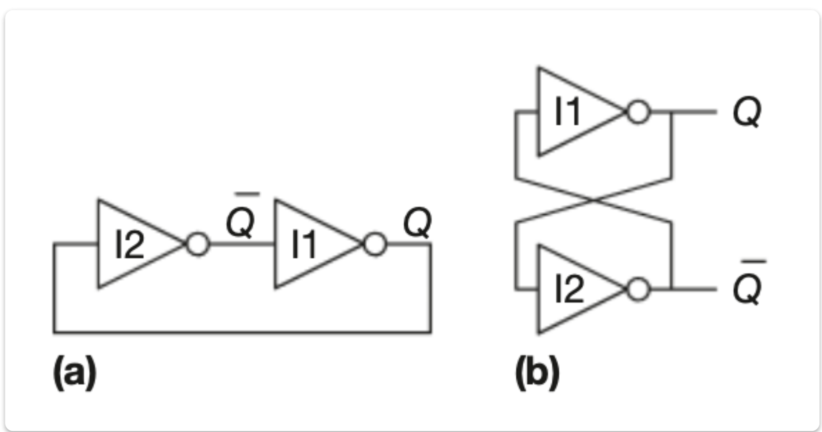
a priorità più alta. Queste reti rappresentano esempi di blocchi costitutivi combinatori.

La specifica di temporizzazione di una rete combinatoria consiste nei ritardi di propagazione e di contaminazione attraverso la rete, che indicano rispettivamente il tempo più lungo e quello più corto tra un cambiamento di un ingresso della rete e il cambiamento dell’uscita che ne consegue. Calcolare il ritardo di propagazione di una rete implica l’individuazione del percorso critico attraverso il circuito, e successivamente la somma dei vari ritardi di propagazione di ogni elemento lungo il percorso. Ci sono diversi modi per realizzare complesse reti combinatorie che offrono buoni compromessi tra la velocità della rete e il suo costo.

3.2-Latch e flip-flop

Introduzione agli elementi bistabili

Il blocco costitutivo della memoria è un elemento *bistabile*, cioè un elemento con 2 stati stabili.
La figura mostra un elemento bistabile composto da una coppia di negatori (inverter) connessi ad anello.



La rete non ha ingressi, ma possiede 2 uscite, Q e $\bar{Q}$ che dipendono una dall'altra:

1. $Q = 0$:
   I1 riceve un ingresso FALSE su Q e produce un'uscita TRUE su $\bar{Q}$. I2 riceve TRUE su Q e produce un'uscita FALSE su $\bar{Q}$.
2. $Q = 1$:
   I1 riceve un ingresso TRUE su Q e produce un'uscita FALSE su $\bar{Q}$. I2 riceve FALSE su Q e produce un'uscita TRUE su $\bar{Q}$.

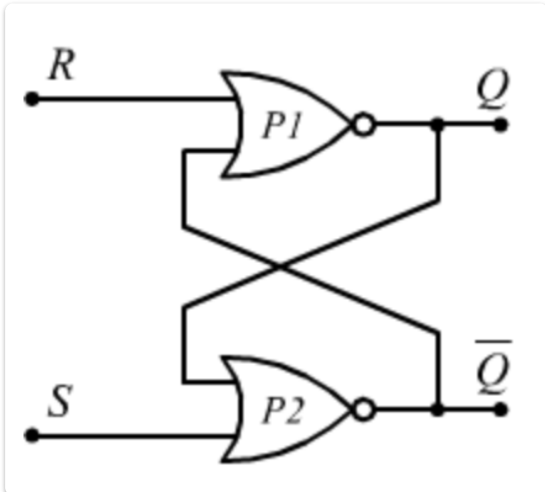
Un elemento con un numero N di stati stabili è caratterizzato da $log_2 N$ bit di informazione, quindi un elemento bistabile immagazzina un bit di infomazione.

L'utente non ha a disposizione un ingresso che gli permetta di controllare lo stato, ma con i *latch* e i *flip-flop* l'ingresso è controllato dall'utente con delle variabili di stato.

Latch

Latch SR

Il Latch SR è composto da 2 porte NOR collegate a croce. Il Latch ha 2 ingressi, S e R, e 2 uscite, Q e $\bar{Q}$. Può essere controllato mediante S e R che attivano (settano) o disattivano (resettano) l'uscita Q.



Qua sotto è mostrata la tabella di verità per comprendere il funzionamento nei vari casi:

S	R	Q	$\bar{Q}$
0	0	Q_{prec}	$\bar{Q}_{prec}$
0	1	0	1
1	0	1	0
1	1	0	0

nb

Q_{prec} e $\bar{Q}_{prec}$ sono i valori precedenti di Q e $\bar{Q}$ salvati nella memoria della rete

Settare un bit significa fargli assumere il valore TRUE, resettarlo fargli assumere il valore FALSE.
Quando viene attivato R, Q viene resettato a 0 e $\bar{Q}$ in modo opposto, quando viene attivato S, Q viene settato a 1 e $\bar{Q}$ in modo opposto.

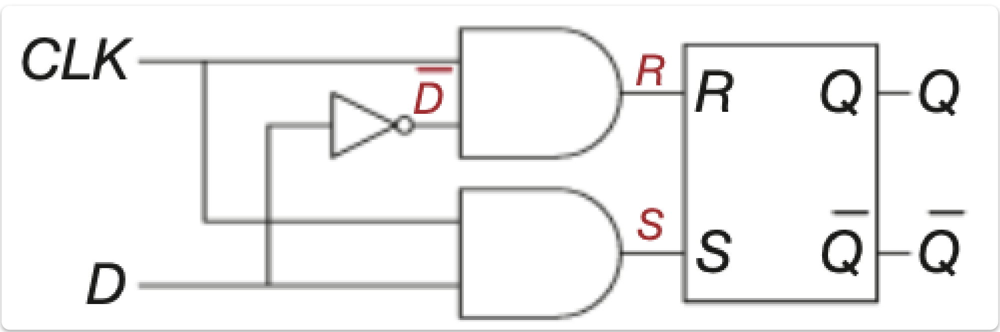
Latch D

Questo Latch ha 2 ingressi: un ingresso *dati*, D, che controlla il prossimo stato, e un ingresso *clock*, CLK, che controlla invece il momento del cambio di stato.
Qua sotto la tabella di verità che descrive il comportamento:

CLK	D		$\bar{D}$	S	R		Q	$\bar{Q}$
0	X		$\bar{X}$	0	0		Q_{prec}	$\bar{Q}_{prec}$
1	0		1	0	1		0	1
1	1		0	1	0		1	0

Quando CLK = 0, sia S sia R sono FALSI, indipendentemente dal valore assunto da D. Se invece CLK = 1, allora una porta AND produce un valore VERO e l'altra un valore FALSO, a seconda del valore di D. La Tabella di prima mostra come, dati i valori di S e di R, sia possibile determinare Q e $\bar{Q}$. Si osservi che, quando CLK = 0, Q ricorda il suo valore precedente, Q_{prec}, mentre quando CLK = 1, Q = D. In ogni caso, $\bar{Q}$ resta, come logico, il complemento di Q. Il latch D è in grado di evitare il caso anomalo in cui gli ingressi S e R vengano attivati simultaneamente.

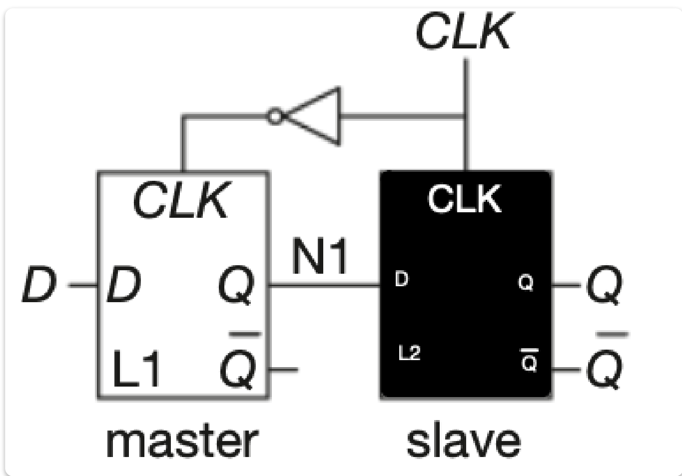
- Quando $CLK = 1$, il Latch è detto *trasparente*, cioè i dati scorrono da D a Q come se fosse un Buffer;
- Quando $CLK = 0$, il Latch è detto *opaco*, quindi viene bloccato il passaggio di dati verso Q che mantiene il valore precedente



Flip-Flop

Flip-flop D

Un Flip-flop D può essere costruito a partire da 2 Latch D in cascata controllati da 2 segnali di clock complementari come in figura:



- L1 viene detto *master*
- L2 viene detto *slave*

Quando $CLK = 0$, il *master* è trasparente e lo *slave* opaco. Di conseguenza il valore D viene portato a $N1$.
Quando $CLK = 1$, il *master* è opaco e lo *slave* trasparente. In questo caso $N1$ viene trasmesso a Q ma $N1$ resta isolato da D.

Quindi, qualunque sia il valore di D subito prima del fronte di salita (passaggio da 0 a 1) del CLK, quello di $N1$ è il valore che viene trasferito a Q al momento di tale fronte. In tutti gli altri casi, Q mantiene il suo valore precedente, dal momento che c'è sempre un latch opaco che blocca il passaggio di dati tra D e Q.

> *Viene anche chiamato flip-flop master-slave o flip-flop attivato sui fronti*

Registro

Un registro a N bit è un banco di N flip-flop che condividono un ingresso CLK comune in modo che tutti i bit vengano aggiornati nello stesso tempo.

Flip-flop con abilitazione

Un flip-flop con abilitazione aggiunge un altro ingresso, chiamato EN o $ENABLE$, per determinare se memorizzare o no il dato sul fronte del CLK. Quando EN è TRUE, il flip-flop con abilitazione reagisce come un normale flip-flop D; quando invece EN è FALSE, il flip-flop con abilitazione ignora il CLK e mantiene il proprio stato.

I flip-flop con abilitazione sono utili quando si desidera inserire un nuovo valore in un flip-flop esclusivamente in alcuni precisi momenti, piuttosto che a ogni cambio del clock.

Flip-flop resettabile

Un flip-flop resettabile aggiunge un altro ingresso, chiamato $RESET$. Quando l'ingresso $RESET$ è FALSE, il flip-flop resettabile si comporta come un flip-flop D. Quando invece $RESET$ è TRUE, il flip-flop resettabile ignora D e resetta l'uscita a 0. Questa tipologia di flip-flop è utile nel caso in cui si desideri forzare uno stato noto (cioè 0) in tutti i flip-flop della rete quando viene accesa.

Questi flip-flop possono essere resettabili in modo *sincrono* o *asincrono*:

- *Sincrono*: si resettano solo al fronte di salita di CLK;
- *Asincrono*: si resettano nel momento in cui $RESET$ diventa TRUE, indipendentemente dal valore da CLK.

3.3-Progetto di reti logiche sincrone

Reti sequenziali sincrone

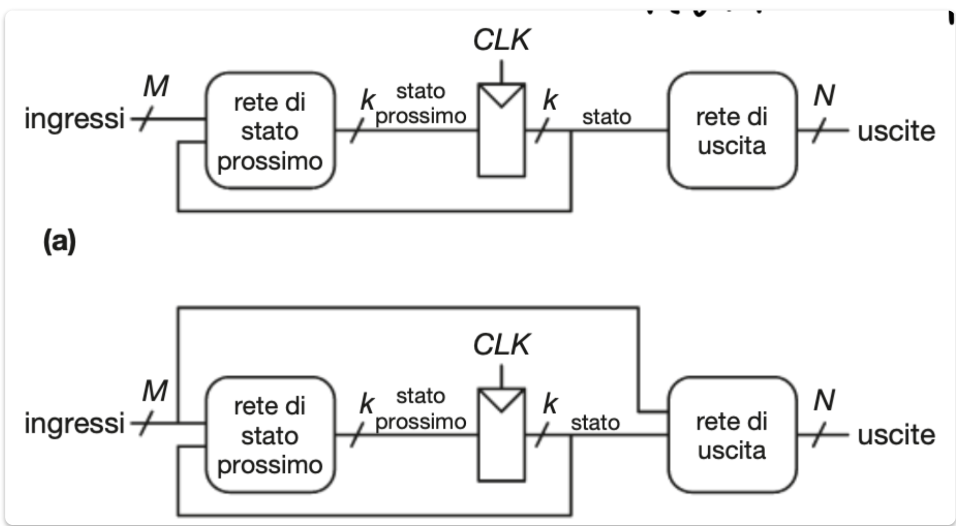
Se si applicano valori di ingresso alla logica combinatoria, le uscite si portano sempre al valore corretto dopo un tempo pari al ritardo di propagazione. Al contrario, le reti sequenziali con percorsi ciclici possono avere condizioni di corsa o comportamenti instabili.
Per evitare questi problemi, i progettisti preferiscono interrompere i percorsi ciclici inserendo in alcuni punti dei registri. Questa operazione trasforma la rete in un insieme di logica combinatoria e registri. I registri contengono lo stato del sistema, che cambia solo in corrispondenza dei fronti di clock, motivo per cui lo stato viene detto sincronizzato con il clock.

Una rete sequenziale ha una serie finita di stati discreti {S0, S1, ..., Sk1}. Una rete sequenziale sincrona ha un ingresso di clock i cui fronti di salita indicano una sequenza di istanti di tempo nei quali hanno luogo le transizioni di stato. Vengono spesso utilizzati i termini *stato presente* e *stato prossimo* per distinguere lo stato in cui il sistema si trova al momento attuale dallo stato in cui si porterà al prossimo fronte di clock.
La specifica temporale consiste in un *limite superiore* (tpcq) e in un *limite inferiore* (tccq) del tempo dal fronte di salita del clock fino ai cambiamenti delle uscite, oltre che dei tempi di setup (attivazione) e di hold (mantenimento), tsetup e thold, che indicano invece quando gli ingressi devono essere stabili rispetto al fronte di salita del clock.
Un flip-flop rappresenta l'esempio più semplice di rete sequenziale sincrona: possiede un ingresso D, un clock, CLK, un'uscita, Q, e due stati, {0, 1}. La specifica funzionale di un flip-flop consiste nell'affermazione che lo stato prossimo è D e che l'uscita, Q, è lo stato presente.

3.4-Macchine a stati finiti

Cosa sono le macchine a stati finiti?

Sono reti sequenziali sincrone che sono in questa forma:



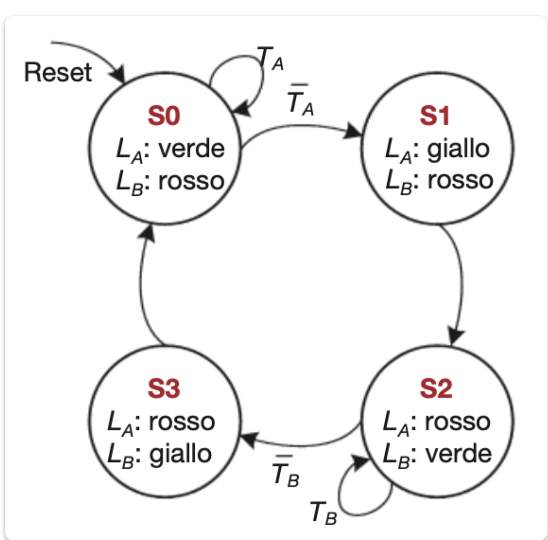
Il loro nome deriva dal fatto che una rete con k registri può trovarsi in 2^k stati diversi. Una FSM ha M ingressi, N uscite, k bit di stato, riceve un segnale clock (CLK) e a volte un segnale di $reset$. è composta da 2 blocchi di logica combinatoria, la *logica di stato prossimo* e la *logica di uscita*, e da un registro che immagazzina lo stato.

Esempio di progettazione di una FSM

Vogliamo far fare al nostro ingegnere Ben un *semaforo* per controllare il traffico a un incrocio universitario.
Ben decide di risolvere il problema utilizzando una *macchina a stati finiti*, e installa due sensori per il traffico, T_a e T_b, rispettivamente in via Accademia e in viale Ateneo. Ognuno di questi sensori indica TRUE se sono presenti degli studenti sulla strada, e FALSE se non sta passando nessuno. Installa anche due semafori stradali, L_a e L_b, per controllare il traffico. Ogni semaforo riceve ingressi digitali che specificano se la luce accesa deve essere verde, gialla o rossa. Quindi la macchina a stati finiti ha due ingressi, T_a e T_b, e due uscite, L_a e L_b

(ciascuna da due bit). Inserisce un clock con un periodo di 5 secondi: a ogni periodo del clock (fronte di salita) i semafori possono cambiare a seconda di quello che indicano i sensori del traffico. Decide di inserire anche un bottone di reset per far sì che si possa portare il controllore a uno stato iniziale noto al momento dell'accensione.

Il *diagramma degli stati* indica tutti i possibili stati del sistema e le transizioni tra di essi. I cerchi rappresentano gli stati e gli archi rappresentano le transizioni tra di essi. Le transizioni avvengono al fronte di salita del clock.
Se il sistema si trova nello stato S_0, rimane in quello stato se T_a è TRUE e passa invece allo stato S_1 se T_a è FALSE.
Se da uno stato parte un solo arco, ciò significa che quella particolare transizione ha luogo indipendentemente dai valori degli ingressi. Per esempio, una volta arrivato allo stato S_1, il sistema passa in ogni caso allo stato S_2. Il valore assunto dalle uscite in ogni stato è indicato nello stato stesso. Per esempio, quando il sistema si trova in stato S_2, L_a è rosso e L_b è verde.



La *tabella degli stati (3.1, 3.2, 3.3)* indica, per ogni stato presente e valori di ingresso, lo stato prossimo S' del sistema.
Per costruire una rete reale però bisogna codificare gli stati e le uscite per scrivere la *tabella delle transizioni (3.4)* ed utilizzare questi codici binari.

Tabella 3.1 Tabella degli stati.

Stato corrente S	Ingressi		Stato prossimo S'
	T_A	T_B	
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Tabella 3.2 Codifica degli stati.

Stato	Codifica $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Tabella 3.3 Codifica delle uscite.

Uscite	Codifica $L_{1:0}$
Verde	00
Giallo	01
Rosso	10

Tabella 3.4 Tabella delle transizioni con codifica binaria degli stati.

Stato corrente		Ingressi		Stato prossimo	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

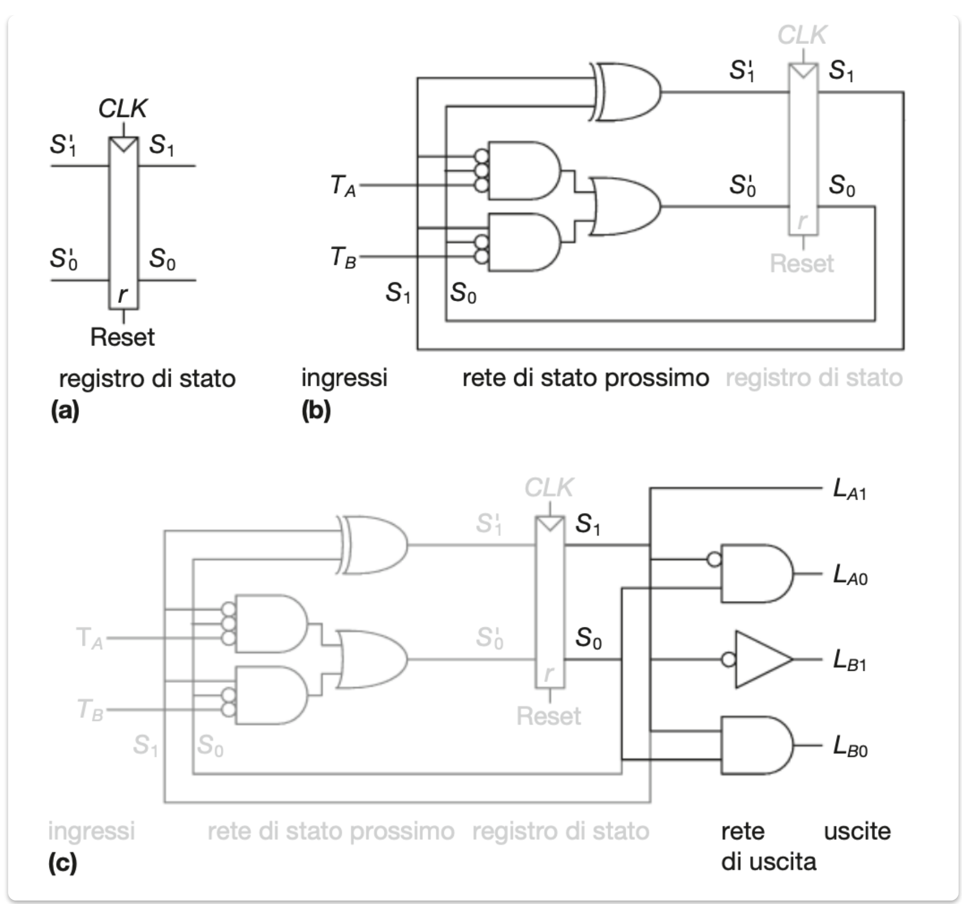
La *Tabella delle uscite* indica per ogni stato quale deve essere il valore assunto dall'uscita.

Tabella 3.5 Tabella delle uscite.

Stato corrente		Uscite			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	T_{B0}
0	0	0	0	0	X
0	1	0	1	0	X
1	0	1	0	0	X
1	1	1	0	1	0

- $L_{A_1} = S_1$
- $L_{A_0} = \bar{S}_1 S_0$
- $L_{B_1} = \bar{S}_1$
- $L_{B_0} = S_1 S_0$

> *Schema macchina alla Moore*



Codifica degli stati

Un problema comune è quello di determinare quale codifica produce la rete desiderata col minor numero di porte logiche o col minor ritardo di propagazione. Non c'è un modo semplice per individuare la codifica migliore.
Spesso è possibile scegliere una buona codifica tramite ispezione, facendo in modo che gli stati o le uscite con- nesse condividano dei bit.
La scelta ricade tra una *codifica binaria* o *codifica a singolo 1*:

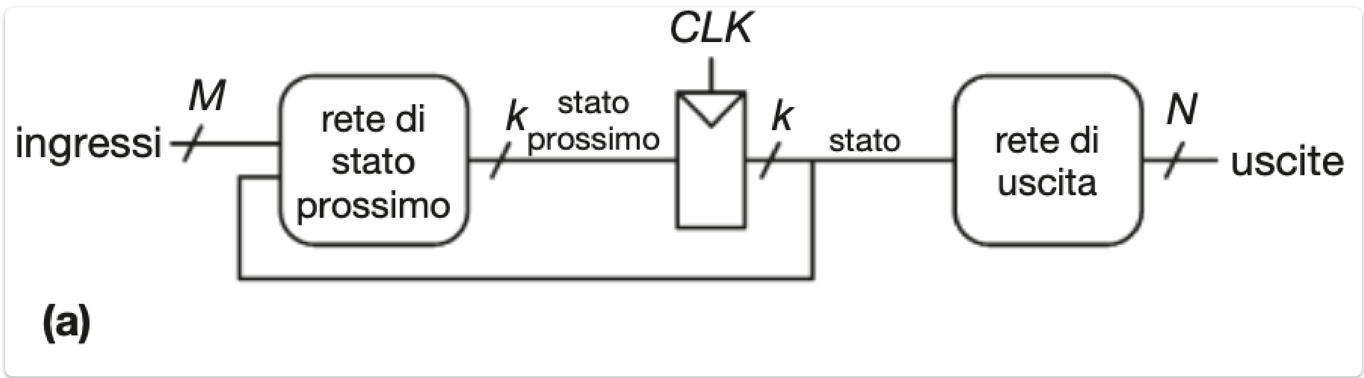
- *Codifica Binaria*: ogni stato viene rappresentato da un numero binario e dal momento che K numeri binari possono essere rappresentati con $log_2 K$ bit, un sistema con K stati avrà
- *Codifica a singolo 1*: viene utilizzato un bit di stato per ognuno degli stati

> *Una FSM con tre stati con codifica a singolo 1 avrà come codifiche di stato 001, 010 e 100. Ogni bit di stato viene immagazzinato in un flip-flop, quindi una codifica a singolo 1*

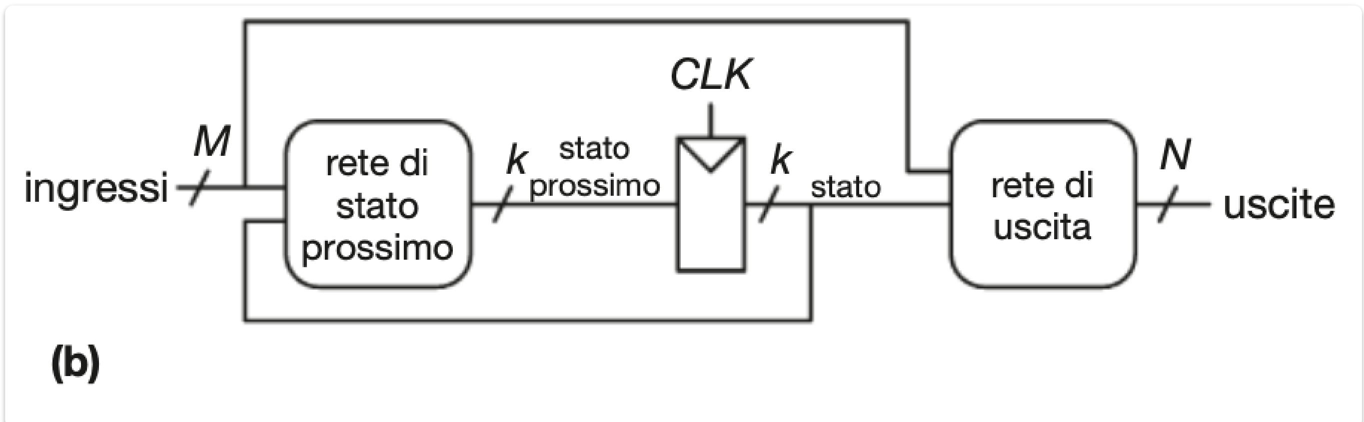
> *necessita di più flip-flop rispetto a una codifica binaria. Ciononostante, con la codifica a singolo 1, le logiche di stato prossimo e di uscita risultano spesso più semplici.*

Macchine alla Moore e macchine alla Mealy

- *Macchine alla Moore*: le uscite dipendono solo dallo stato presenta della macchina. Nel diagramma degli stati i valori delle uscite vengono indicati nei cerchi.



- *Macchine alla Mealy*: le uscite dipendono sia dallo stato presente della macchina sia dagli ingressi attuali. Nel diagramma degli stati le uscite sono indicate sugli archi.



Derivare un FSM da uno schema circuitale

Serve:

- Esaminare la rete, gli ingressi e le uscite, i bit di stato.
- Scrivere le espressioni di stato prossimo e di uscita.
- Creare le tabelle delle transizioni e delle uscite.

- Ridurre le tabelle eliminando gli stati che non possono essere raggiunti.
- Assegnare un nome simbolico a ogni valida combinazione di bit di stato.
- Riscrivere le tabelle delle transizioni e delle uscite con i nuovi nomi.
- Disegnare il diagramma degli stati.
- Esprimere a parole il funzionamento della FSM.

Riassunto

Per progettare una FSM si utilizza la procedura seguente:

- Identificare gli ingressi e le uscite.
- Disegnare un diagramma degli stati.
- Per una macchina alla Moore:
  - Scrivere la tabella degli stati.
  - Scrivere la tabella delle uscite.
- Per una macchina alla Mealy:
  - Scrivere una tabella unica di stati e uscite.
- Decidere la codifica degli stati, tenendo presente che la scelta influenza il progetto hardware.
- Scrivere le espressioni booleane per la logica di stato prossimo e la logica di uscita.
- Disegnare lo schema della rete.

3.5-Temporizzazione della logica sequenziale

Cicli di clock

Un flip-flop copia D sull'uscita Q a ogni fronte di salita del clock. Questo processo viene detto *campionamento* di D al fronte di salita del clock. Se D è stabile a 0 o 1 quando il clock ha fronte di salita, il comportamento del flip-flop è definito in modo chiaro.
Il *tempo di apertura* di un elemento sequenziale viene definito da un tempo di *setup* e da un tempo di *hold*.

> *è possibile interpretare il tempo con i cicli di clock*

Il periodo del clock deve essere abbastanza lungo da permettere a tutti i segnali di stabilizzarsi, il che costituisce un limite per la velocità del sistema. Nei sistemi reali, il clock solitamente non raggiunge tutti i flip-flop allo stesso tempo e questa differenza di tempo, detta sfasamento del clock, aumenta ulteriormente il periodo di clock necessario.

Temporizzazione del sistema

Il *periodo di clock* o *tempo di ciclo*, T_c, è il tempo fra i fronti di salita periodici del clock. $f_c = 1/T_c$ è la *frequenza di clock*; aumentare la frequenza aumenta la quantità di lavoro.

Vincolo sul tempo di setup

$$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$

Dove:

- T_c: periodo di clock
- t_{pcq}: ritardi di propagazione da clock a Q
- t_{setup}: tempo di setup
- t_{pd}: tempo di propagazione del segnale

Quindi:

$$t_{pd} \leq T_c - (t_{pcq} + t_{setup})$$

Il termine racchiuso tra parentesi si chiama *sovraccarico di sequenziamento*

Vincolo sul tempo di hold

$$t_{ccq} + t_{cd} \geq t_{hold}$$

Dove:

- t_{ccq}: tempo di commutazione del segnale (il tempo che impiega un segnale a cambiare di stato da un valore logico all'altro)
- t_{hold}: tempo in cui il dato D non deve cambiare dopo il fronte di salita del clock

Quindi:

$$t_{cd} \geq t_{hold} - t_{ccq}$$

L'espressione viene chiamata *vincolo sul tempo di hold* o *vincolo di ritardo minimo*

> nb
>
> Le reti sequenziali hanno vincoli sul tempo di setup e sul tempo di hold che impongono i ritardi massimi e minimi della logica combinatoria tra i diversi flip-flop

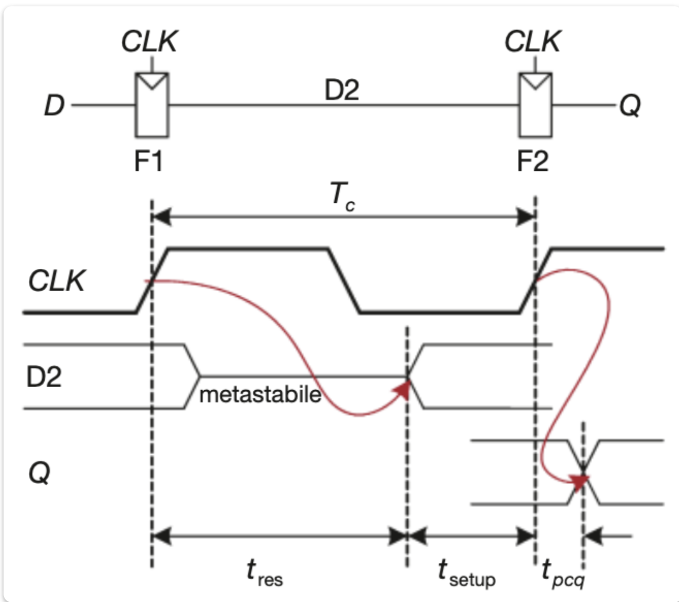
Sincronizzatori

Per garantire dei livelli logici corretti, tutti gli ingressi asincroni dovrebbero essere fatti passare attraverso dei *sincronizzatori*.
Il sincronizzatore, mostrato nella Figura, è un dispositivo che riceve un ingresso asincrono D e un clock CLK, per produrre un'uscita Q entro un certo periodo di tempo; la probabilità che l'uscita abbia un livello logico valido è estremamente alta. Se D è stabile durante il tempo di apertura, Q assume lo stesso valore

di D. Se invece D cambia durante il tempo di apertura, Q può assumere un valore TRUE o FALSE, ma non può assumere un valore *metastabile*.

Nb

Un valore metastabile è un valore intermedio e instabile che può essere assunto da un circuito digitale quando si verifica una transizione di stato. In questo stato, il valore di uscita del circuito oscilla tra due valori logici possibili, senza stabilizzarsi su nessuno dei due.



Img

La Figura mostra un modo semplice di costruire un sincronizzatore a partire da due flip-flop. $F1$ campiona D al fronte di salita di CLK: se D cambia in quel momento, l'uscita $D2$ potrebbe momentaneamente assumere un valore *metastabile*. Se il periodo del clock è abbastanza lungo, $D2$ con elevata probabilità si stabilizza su un livello logico valido prima della fine del periodo. Successivamente $F2$

campiona $D2$, che a questo punto è stabile, producendo un'uscita accettabile Q.

3.6-Parallelismo

Cos'è il parallelismo?

- *token*: un gruppo di ingressi che vengono elaborati per produrre un gruppo di uscite
- *latenza*: tempo richiesto a un token per attraversare il sistema dall'inizio alla fine
- *capacità produttiva*: numero di token che possono essere elaborati per unità di tempo

La capacità produttiva può essere aumentata se si elaborano più token allo stesso tempo. Questo modo di lavorare viene chiamato *parallelismo* e ne esistono di 2 tipi:

- *parallelismo spaziale*: vengono usate piuù copie dell'hardware in modo che più lavori possano essere svolti conteporaneamente
- *parallelismo temporale*: ogni compito viene diviso in fasi come una catena di montaggio (pipelining)

> nb
>
> Il problema principale del parallelismo è dato dalle dipendenze. Se un’azione presente dipende da un’azione precedente, invece di dipendere solo dai passi precedenti compiuti sulla stessa azione, tale azione non può iniziare finché quella prima non si è conclusa.

3.7-Riassunto

In questo capitolo sono stati descritti l'analisi e il progetto della logica sequenziale. Al contrario della logica combinatoria, le cui uscite dipendono solo dagli ingressi presenti in quel momento, le uscite della logica sequenziale dipendono sia dagli ingressi presenti in quel momento sia dagli ingressi precedenti.

Nelle reti sequenziali ci si limita ad utilizzare un numero ridotto di blocchi costitutivi progettati opportunamente. Per gli obiettivi di questo libro, l'elemento più importante è il flip-flop, che riceve un clock e un ingresso D e produce un'uscita Q. Il flip-flop copia D su Q al *fronte di salita* del clock e in tutte le altre situazioni ricorda lo stato precedente di Q. Un gruppo di flip-flop che condividono lo stesso clock viene detto registro. I flip-flop possono anche avere ingressi di reset e abilitazione.

Nonostante esistano molte forme di logica sequenziale, si è scelto di utilizzare le reti sequenziali sincrone perché più semplici da progettare. Le reti sequenziali sincrone consistono di blocchi di logica combinatoria divisi da registri con clock. Lo stato della rete viene immagazzinato nei registri e aggiornato solo ai fronti del clock.

Le macchine a stati finiti (FSM) costituiscono una tecnica molto efficace per progettare le reti sequenziali. Per progettare una FSM si seguono i passi descritti nel riassunto.

Le reti sequenziali sincrone hanno una *specifica temporale* che include i ritardi di propagazione e di contaminazione da clock a Q, rispettivamente t_{pcq} e t_{ccq}, e i tempi di setup e hold, t_{setup} e t_{hold}. Perché la rete operi correttamente, gli ingressi devono essere stabili per un tempo di apertura che inizia un tempo di setup prima del fronte di salita del clock e finisce un tempo di hold dopo il fronte di salita del clock. Il tempo di ciclo minimo T_c del sistema è uguale al ritardo di propagazione t_{pd} attraverso la logica combinatoria più t_{pcq} + t_{setup} del registro. Inoltre, il

ritardo di contaminazione attraverso il registro e la logica combinatoria deve essere maggiore di t_{hold}. Nonostante l'errata convinzione comune, il tempo di hold non ha effetti sul tempo di ciclo.

Le prestazioni globali del sistema vengono misurate in termini di latenza e di capacità produttiva. La latenza è il tempo necessario a un *token* per attraversare la rete dall'inizio alla fine. La capacità produttiva è il numero di token che il sistema può elaborare per unità di tempo. Il parallelismo migliora la capacità produttiva del sistema.

4.1-Introduzione

Linguaggi di descrizione dell'hardware

I progettisti si sono resi conto che è molto più produttivo fornire le specifiche delle funzioni logiche lasciando a una strumento di *CAD* (Computer Aided Design) il compito di produrre una rete di porte logiche ottimizzata. Tali specifiche sono fornite in genere tramite *linguaggi di descrizione dell'hardware* (HDL, Hardware Description Language) come *System Verilog* e *VHDL*.

Origini dei linguaggi

L'industria sembra indirizzarsi verso $SystemVerilog$ ma molte aziende usano ancora $VHDL$ nonostante sia più pesante e verboso di System Verilog.

$SystemVerilog$	$VHDL$
Verilog è stato sviluppato nel 1984 dalla ditta Gateway Design Automation come linguaggio proprietario per la simulazione logica.	VHDL è stato sviluppato nel 1981 dal Dipartimento della Difesa per descrivere struttura e funzionamento dell’hardware. Le sue radici derivano dal linguaggio di programmazione Ada. Inizialmente pensato come linguaggio di documentazione, è divenuto presto uno strumento di simulazione e sintesi circuitale.

Simulazione e sintesi

I due obbiettivi principali degli HDL sono la *simulazione* e la *sintesi*.
Nella simulazione si forniscono valori di ingresso al modulo e si controlla alle uscite se il modulo funziona correttamente.
Nella sintesi la descrizione testuale del modulo viene tradotta in rete di porte logiche.

Simulazione

La *simulazione* è fondamentale per consentire di collaudare un sistema prima di costruirlo fisicamente visto che correggere un errore quando il sistema digitale è già stato prodotto può essere enormemente costoso. L'operazione di correggere un errore (bug) viene detta "debuggare":

Sintesi

La sintesi logica trasforma il codice HDL in uno schema di porte logiche e fili (netlist) che descrivono la realizzazione circuitale del modulo. Il sintetizzatore logico può fare ottimizzazioni per ridurre la quantità di hardware necessaria.
Il codice HDL viene diviso tra *moduli sintetizzabili* e *"banco di prova"* (testbench), dove i moduli sono la parte circuitale e il banco contiene le istruzioni per applicare valori agli ingressi, verificare la correttezza dei valori in uscita ed evidenziare la discrepanze tra valori attesi e valori effettivi.
I vari HDL hanno modi specifici di descrivere le classi di componenti logiche, definiti *idiomi*.

4.2-Logica Combinatoria

Operatori a singolo bit

Gli operatori a singolo bit (bitwise) agiscono su segnali costituiti da *bit singoli* o su *bus multibit*.

> *ex.*
> *l'operatore* `neg` *descrive 4 negatori collegati a bus a 4 bit*

```language-sv
module neg (input logic [3:0] a,
                        output logic [3:0] y);
        assign y = ~a;
endmodule
```

> nb
>
> l'ordinamento dei bit di un bus è arbitrario e può essere:
>
> - *little-endian*: a[$N-1$:0]
> - *big-endian*: a[0:$N-1$]

Commenti a spazio vuoto

I commenti in SystemVerilog sono simili a quelli in C: iniziano con la coppia di caratteri `/*` e continuano, anche su più righe del file, fino alla coppia di caratteri `*/`. Commenti che iniziano con la coppia di caratteri `//` terminano invece alla fine della riga.

Operatori di riduzione

Gli operatori di riduzione sono costituiti da porte logiche a tanti ingressi che producono una sola uscita

```language-sv
module and8(input logic [7:0] a, output logic y);
        assign y = &a;
```

```
// assign y = &a è molto più facile da scrivere di
// assign y = a[7] & a[6] & a[5] & a[4] &
//            a[3] & a[2] & a[1] & a[0];

endmodule
```

Assegnamento condizionale

Un assegnamento condizionale seleziona l'uscita da generare fra varie alternative sulla base di un ingresso chiamato *condizione* (preticamente un if)

```language-sv
module mux2(input logic [3:0] d0, d1, input logic s, output logic [3:0] y);

assign y = s ? d1 : d0;

endmodule
```

> **Nb**
>
> L'operatore condizionale ?: seleziona, sulla base della prima espressione, la seconda o la terza espressione. La prima espressione è denominata condizione: se la condizione vale 1 l'operatore sceglie la seconda espressione, se vale 0 la terza

Variabili interne

`p` e `g` sono variabili interne perchè non sono nè ingressi nè uscite ma sono utilizzate solo all'interno del modulo. Sono simili alle variabili locali nei linguaggi di programmazione.

```language-sv
module sommatore (input logic a, b, rin, output logic s, rout);

logic p, g; //qua avviene la dichiarazione delle variabili interne

assign p =a ^ b;
assign g =a & b;
assign s = p ^ rin;
```

```
  assign rout = g | (p & rin);

endmodule
```

Precedenze

	Op	Significato
Alta	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PIÙ, MENO
	<<, >>	Traslazione logica a sinistra/destra
	<<<, >>>	Traslazione aritmetica a sinistra/destra
	<, <=, >, >=	Confronto relativo
	==, !=	Confronto di uguaglianza
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
Bassa	?:	Condizionale

Numeri

I numeri possono essere rappresentati in binario, ottale, decimale o esadecimale. La dimensione, ovvero il numero di bit, può essere specificata, e nelle posizioni più significative vengono inseriti 0 fino a riempire tale dimensione.

Numeri	Bit	Base	Val	Risultato
3'b101	3	2	5	101
'b11	?	2	3	000 ... 0011
8'b11	8	3	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	2	00 ... 0101010

Concatenazione di bit

Spesso è necessario operare su un sottoinsieme dei segnali di un bus, o concatenare segnali provenienti da sorgenti diverse in un bus. Queste operazioni sono denominate in inglese *bit swizzling* (cocktail di bit).

```language-sv
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};

// L'operatore {} è usato per concatenare bus. {3{d[0]}} indica tre copie di d[0].
```

Ritardi

Le istruzioni HDL possono essere associate a ritardi, specificati in unità arbitrarie. I ritardi sono molto utili durante la simulazione per predire quanto velocemente funzionerà un circuito (se vengono specificati ritardi significativi) e anche in fase di debugging per comprendere cause ed effetti dei vari comportamenti (capire quale sia la causa di un'uscita sbagliata è molto difficile se tutti i segnali cambiano simultaneamente durante la simulazione). I ritardi vengono ignorati durante la sintesi: il ritardo di una porta logica prodotta dallo strumento di sintesi dipende dalle specifiche in termini di t_{pd} e t_{cd} e non da valori numerici messi nel codice HDL.

```language-sv
'timescale 1ns/1ps  //è scritta qua!!!
module esempio(input logic a, b, c, output logic y);

logic ab, bb, cb, n1, n2, n3;

assign #1 {ab, bb, cb} = ~{a, b, c};
assign #2 n1 = ab & bb & cb;
assign #2 n2 = a & bb & cb;
assign #2 n3 = a & bb & c;
assign #4 y = n1 | n2 | n3;
endmodule
```

> Nb
>
> I file SystemVerilog possono includere una direttiva di scala temporale che indica il valore di ogni unità di tempo. La direttiva è nella forma 'timescale unit/precision'. In questo

esempio ogni unità è 1 ns, e la simulazione ha 1 ps di precisione. Se non si specifica la scala temporale, viene adottato, in genere, il valore di 1 ns sia per l'unità sia per la precisione. In SystemVerilog il simbolo # è usato per indicare il numero di unità di ritardo.

4.3-Modellazione strutturale

L’Esempio 1 mostra come costruire un multiplexer 4:1 con tre multiplexer 2:1. Ogni copia del multiplexer 2:1 è chiamata istanza. Più istanze di uno stesso modulo si distinguono grazie a nomi diversi, in questo caso `muxbasso`, `muxalto` e `muxuscita`.

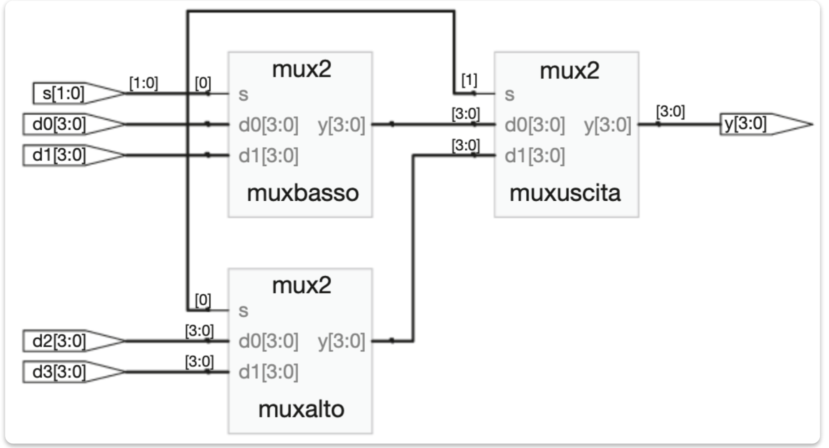
```language-sv
module mux4(input logic [3:0] d0, d1, d2, d3, input logic [1:0] s,
output logic [3:0] y);

logic [3:0] basso, alto;

mux2 muxbasso(d0, d1, s[0], basso);
mux2 muxalto(d2, d3, s[0], alto);
mux2 muxuscita(basso, alto, s[1], y);

endmodule
```

> *Le tre istanze di mux2 sono denominate muxbasso, muxalto e muxuscita. Il modulo mux2 deve essere definito da qualche altra parte nel codice SystemVerilog*



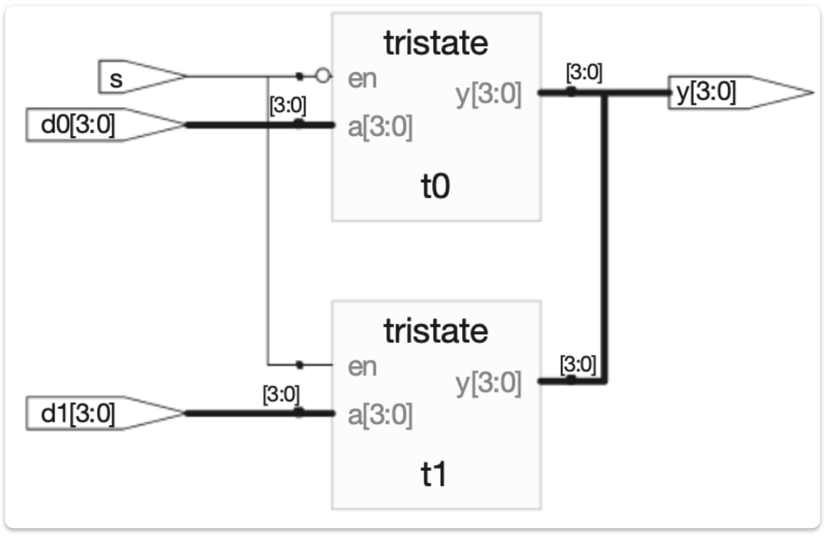
L’Esempio 2 usa la modellazione strutturale per costruire un multiplexer 2:1 partendo da una coppia di buffer tristate. Tuttavia non è consigliabile costruire circuiti logici facendo uso di buffer tristate.

```language-sv
module mux2(input logic [3:0] d0, d1, input logic s, output tri [3:0]
y);
```

```
tristate t0(d0, ~s, y);
tristate t1(d1, s, y);

endmodule
```

> *In SystemVerilog, espressioni come ~s sono permesse nell'elenco di ingressi e uscite di un'istanza. Espressioni arbitrariamente complesse sono comunque sconsigliate perché rendono il codice poco leggibile.*

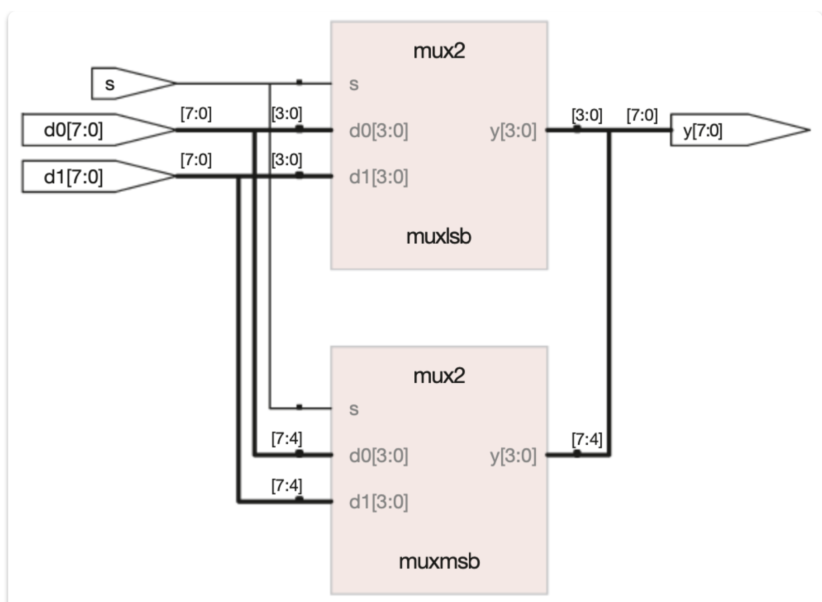


L'Esempio HDL 3 mostra come i moduli possano accedere a parti di un bus. Un multiplexer 2:1 a 8 bit è realizzato con due dei multiplexer 2:1 a 4 bit già definiti, che lavorano rispettivamente sul nibble alto e su quello basso degli 8 bit.

```language-sv
module mux2_8(input logic [7:0] d0, d1, input logic s, output logic [7:0] y);

mux2 muxlsb(d0[3:0], d1[3:0], s, y[3:0]);
mux2 muxmsb(d0[7:4], d1[7:4], s, y[7:4]);

endmodule
```



nb

In generale, i sistemi complessi vengono realizzati in modo gerarchico. Il sistema completo viene descritto strutturalmente istanziando le sue componenti principali, quindi ogni componente viene descritta strutturalmente in termini di blocchi che la costituiscono, e il processo viene ripetuto ricorsivamente finché i blocchi sono abbastanza semplici da poter essere descritti in modo comportamentale. È buona norma evitare (o comunque contenere al massimo) di mischiare descrizioni strutturali e comportamentali all’interno di un singolo modulo.

5.2-Circuiti aritmetici

Cosa fanno?

I circuiti aritmetici sono i blocchi costruttivi centrali dei calcolatori. I calcolatori e la logica digitale eseguono molte funzioni aritmetiche: addizioni, sottrazioni, confronti, traslazioni, moltiplicazioni e divisioni.

Addizione

Si inizia esaminando come sia possibile [sommare due numeri binari a 1 bit](). Dopodiché, l'operazione viene estesa ai numeri binari a N bit.

Semisommatore

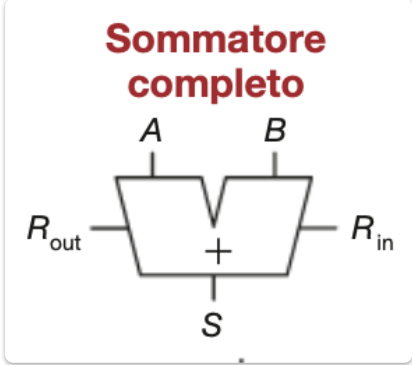
Il semisommatore ha due ingressi, A e B, e due uscite, S e R_{out}. S rappresenta la somma di A e B. Se sia A sia B hanno valore 1, S è uguale a 2, un valore che non può essere rappresentato con una sola cifra binaria. Di conseguenza, il valore 2 viene rappresentato con un *riporto* (carry) R_{out} nella colonna successiva. Il semisommatore può essere costruito con una porta XOR e una AND.
R_{out} viene sommato al bit più significativo successivo, però al semisommatore manca un ingresso R_{in} per accettare R_{out} della colonna precedente.

Semisommatore

A	B	R_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

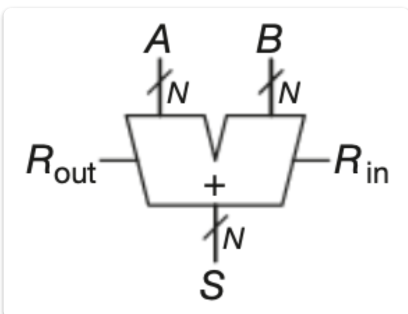
Sommatore completo

Un sommatore completo acceta l'ingresso R_{in}



Sommatore a propagazione di riporto

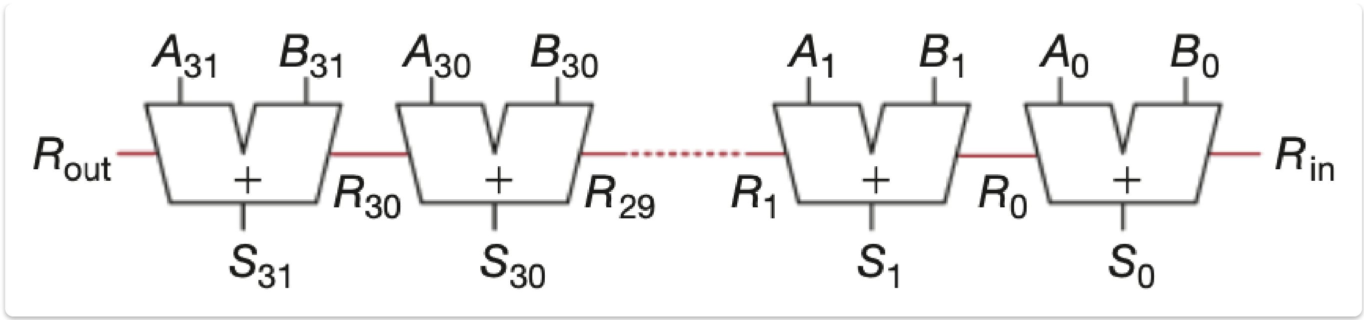
Un sommatore a N bit somma due ingressi a N bit, A e B, e aggiunge R_{in} per produrre un risultato a N bit S e un riporto R_{out}. Viene comunemente chiamato *sommatore a propagazione di riporto* (o CPA, Carry Propagate Adder) perché il riporto di un bit si propaga nel bit successivo.



Sommatore a propagazione di riporto a onda

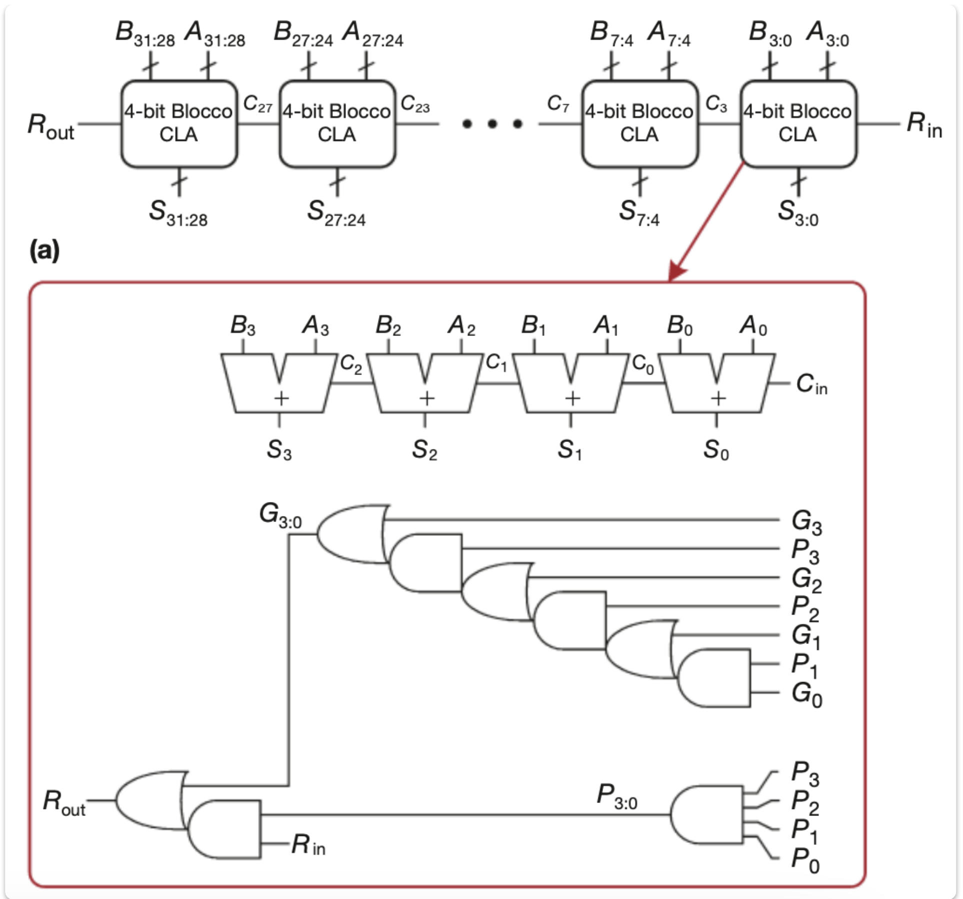
Il metodo per costruire un *sommatore a propagazione di riporto a onda* a N bit è collegare in cascata N full adder completi. In questo modo R_{out} di uno stadio costituisce R_{in} per lo stadio successivo. Il ritardo di un full adder è:

$$t_{propag} = Nt_{FA}$$



Sommatore ad anticipazione di riporto

Un *sommatore ad anticipazione di riporto* (CLA, Carry-Lookahead Adder) è un altro tipo di sommatore a propagazione di riporto che risolve il problema della velocità dividendo il sommatore stesso in blocchi e aggiungendo un circuito per determinare velocemente il riporto di uscita da ciascun blocco appena è noto il riporto di ingresso.
I sommatori ad anticipazione di riporto utilizzano segnali di *generazione* (G) e di *propagazione* (P) che descrivono come una colonna o un blocco determinano il proprio riporto. La colonna `i` di un sommatore genera sicuramente R_i se A_i e B_i sono entrambi uguali a 1. Di conseguenza G_i, cioè il riporto generato dalla colonna `i`, viene calcolato come $G_i = A_iB_i$.
Tutti i blocchi del CLA calcolano i segnali di generazione e di propagazione sia di colonna sia di blocco simultaneamente. Il percorso critico della rete inizia con il calcolo di G_0 e $G_{3:0}$ nel primo blocco del sommatore ad anticipa- zione di riporto.
Dopodiché, R_{in} avanza direttamente fino a R_{out} attraverso la porta AND/OR in ogni blocco fino all'ultimo

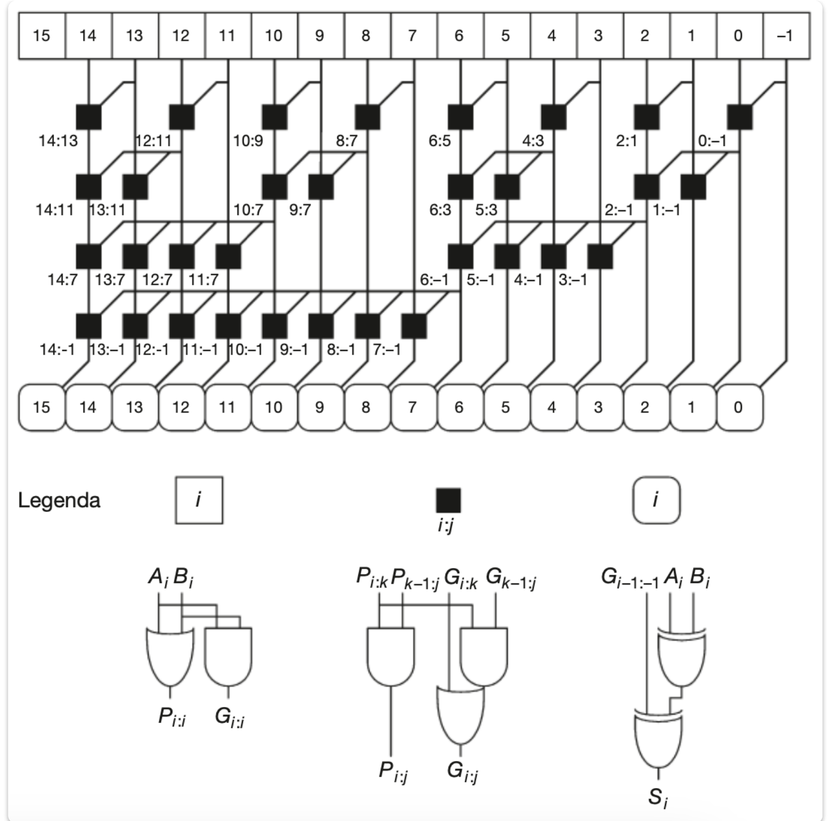


Sommatore a prefissi

Questi sommatori calcolano G e P per coppie di colonne, poi per gruppi di 4 colonne, poi di 8, 16 ecc., fino a che il segnale di generazione di ogni colonna non è noto. Le somme sono calcolate a partire da questi segnali di generazione.

La strategia di un sommatore a prefissi è di calcolare il più rapidamente possibile il segnale di ingresso R_{i1} per ogni colonna `i` per poi eseguire la somma, utilizzando l'espressione:

$$S_i = (A_i \oplus B_i) \oplus R_{i-1}$$



Sottrazione

I sommatori sono in grado di sommare numeri sia positivi sia negativi utilizzando la rappresentazione dei numeri in complemento a 2. La *sottrazione* è facile quanto l'addizione:
si inverte il segno del secondo numero e poi si esegue la somma.

Comparatori

Un comparatore determina se due numeri binari sono uguali o se uno dei due è maggiore o minore dell’altro. Un comparatore riceve due numeri binari a N bit, A e B. Esistono due tipi comuni di comparatori:

- *Comparatore di uguaglianza*: produce una singola uscita che indica se $A == B$ oppure no.
- *Comparatore di valore*: produce invece una o più uscite che indicano i valori relativi di A e di B.

Il comparatore determina se i bit corrispondenti a ogni colonna A e B sono uguali utilizzando delle porte XNOR.

La comparazione avviene effettuata calcolando $A - B$ e guardando il segno (bit più significativo).

- Se 1 $\rightarrow$ $A < B$
- Se 0 $\rightarrow$ $A \geq B$

ALU

Un'*unità logica/aritmetica* (ALU, Arithmetic/Logical Unit) unisce all'interno di una singola unità una serie di operazioni logiche e matematiche.
La Figura mostra il simbolo di una *ALU* a N bit con ingressi e uscite di N bit. La ALU riceve un segnale di controllo a 2 bit, chiamato $ControlloALU$, che specifica quale funzione debba eseguire.

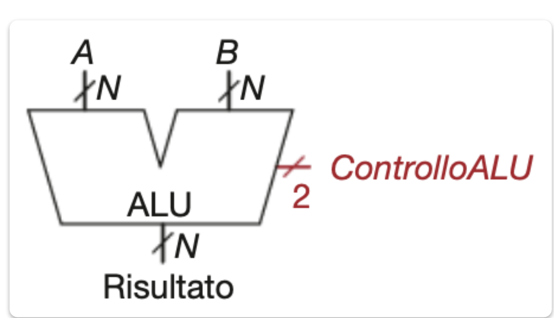


Tabella 5.1 **Operazioni dell'ALU.**

$ControlloALU_{1:0}$	Funzione
00	Addizione
01	Sottrazione
10	AND
11	OR

Alcune ALU producono uscite ulteriori, chiamate *flag*, che danno informazioni aggiuntive sul risultato dell'ALU.
L'uscita $FlagALU$ è composta dalle flag N, Z, C e V che indicano, rispettivamente, che il risultato dell'ALU è *negativo* (Negative) o uguale a *zero* (Zero) o che il sommatore ha generato un *riporto* (Carry) o un *traboccamento* (oVerflow).

Traslatori e rotatori

I *traslatori* (shifter) e i *rotatori* (rotator) traslano i bit ed eseguono la moltiplicazione o la divisione per potenze di 2. Come suggerisce il nome, i traslatori traslano un numero binario a destra o a sinistra di uno specifico numero di posizioni. Esistono diversi tipi di traslatori:

- *Traslatore logico*: trasla un numero verso sinistra (LSL, Logical Shift Left) o verso destra (LSR, Logical Shift Right)

e riempie gli spazi lasciati vuoti con 0.

> *Ex.*
> *11001 LSR 2 = 00110; 11001 LSL 2 = 00100*

- *Traslatore aritmetico*: stessa funzione di un traslatore logico ma quando trasla un numero verso destra (ARS, Arithmetic Shift Right), riempie i bit più significativi con una copia del precedente bit più significativo (msb, most significant bit). La traslazione aritmetica verso sinistra (ASL, Arithmetic Shift Left) si comporta come la traslazione logica verso sinistra.

> *Ex.*
> *11001 ASR 2 = 11110; 11001 ASL 2 = 00100*

- *Rotatore*: trasla un numero verso sinistra (ROL, Rotate Left) o verso destra (ROR, Rotate Right) circolarmente, in modo che gli spazi lasciati vuoti vengano riempiti dai bit all'estremità opposta del numero.

> *Ex.*
> *11001 ROR 2 = 01110; 11001 ROL 2 = 00111*

Un traslatore a N bit può essere costruito con un numero N di Multiplexer N:1.

Moltiplicazione

La moltiplicazione tra numeri binari senza segno è simile alla moltiplicazione decimale ma con solo uni e zeri.
La Figura confronta la moltiplicazione di numeri decimali e binari.

230	moltiplicando	0101
× 42	moltiplicatore	× 0111
460	prodotti	0101
+920	parziali	0101
9660		0101
		+0000
	risultato	0100011
230 × 42 = 9660		5 × 7 = 35

In entrambi i casi vengono formati dei prodotti parziali moltiplicando una singola cifra del moltiplicatore per tutte le cifre del moltiplicando. I prodotti parziali traslati vengono poi sommati per ottenere il risultato finale.

In generale un moltiplicatore $N \times N$ moltiplica due numeri a N bit e produce un risultato a $2N$ bit. I prodotti parziali nella moltiplicazione binaria corrispondono o al moltiplicando o a tutti 0. La moltiplicazione tra numeri binari a 1 bit è equivalente a un'operazione AND, quindi vengono utilizzate porte AND per formare i prodotti parziali.

Divisione

La divisione binaria può essere effettuata utilizzando l'algoritmo seguente per i numeri senza segno a N bit nell'intervallo $[0, 2^{N-1}]$:

```language-sv
R' = 0

for i=N-1 to 0
        R = {R' << 1, Ai}
        D = R - B
        if D < 0 then      Qi = 0, R' = R //R<B
        else               Qi = 1, R' = D //R⩾B

R = R'
```

5.3-Sistemi di numerazione

I calcolatori operano sia con numeri interi sia con numeri frazionari. Esistono i *numeri in virgola fissa* e i *numeri in virgola mobile* (guarderemo solo qeusti ultimi).

Numeri in virgola mobile

Come la notazione scientifica, anche i numeri in virgola mobile hanno un *segno*, una *mantissa* (M), una *base* (B) e un esponente (E). Il punto decimale è mobile perchè viene posizionato immediatamente a destra della cifra più significativa. I numeri in virgola mobile sono in base 2 con una mantissa binaria: vengono usati 32 bit per rappresenatare 1 bit di segno, 8 bit di esponente e 23 di mantissa.
Il primo bit della mantissa è sempre uguale a 1 e quindi non ha bisogno di essere rappresentato (*uno più significativo implicito*), infatti solo i bit frazionari vengono riportati così da liberare un bit di dato utile.

L'esponente deve essere in grado di rappresentare sia numeri positivi, sia numeri negativi. Per fare ciò si utilizza un esponente con *codifica ad eccesso*, costituito da l'esponente sommato ad un eccesso costante. La virgola mobile utlizza l'eccesso 127:

$E = 7 \ \rightarrow \ 7 + 127 = 134 = 10000110_2$
$E = -4 \ \rightarrow \ -4 + 127 = 123 = 01111011_2$

Casi speciali $0, \pm\infty$ e NaN

Numero	Segno	Esponente	Mantissa
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non nulla

Formati a precisione singola e doppia

- *singola precisione*: a 32 bit, `float`
- *doppia precisione*: a 64 bit, `double`

Formato	Bit totali	Bit di segno	Bit di esponente	Bit di mantissa
Singola precisione	32	1	8	23
Doppia precisione	64	1	11	52

Arrotondamento

I metodi di arrotondamento sono:

- arrotondamento per difetto
- arrotondamento per eccesso
- arrotondamento verso lo zero
- arrotondamento al numero più vicino

Si genera un *traboccamento per eccesso* (overflow) quando un numero è troppo grande per essere rappresentato.
Si genera un *traboccamento per difetto* (underflow) quando un numero è troppo piccolo per essere rappresentato.

Addizione in virgola mobile

I passaggi per sommare due numeri in virgola mobile con lo stesso segno sono:

1. Estrarre i bit dell'esponente e quelli della mantissa.
2. Aggiungere l'1 più significativo per ottenere la mantissa effettiva.
3. Confrontare gli esponenti.
4. Traslare se necessario la mantissa con esponente minore.
5. Sommare le due mantisse.
6. Normalizzare la mantissa risultante sistemando, se necessario, l'esponente.
7. Arrotondare il risultato.

8. Assemblare l’esponente e la mantissa nella notazione in virgola mobile.

5.5-Componenti di memoria

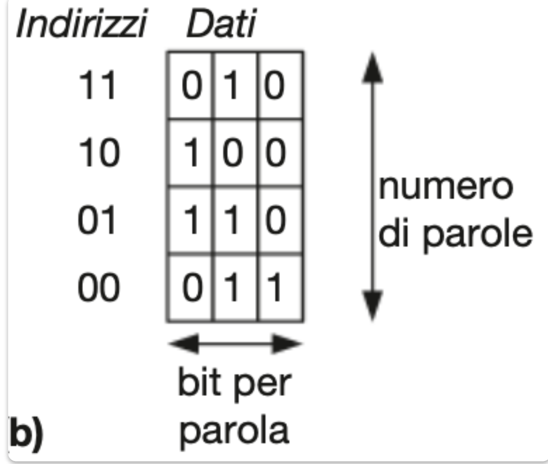
Tipi di memorie

I sistemi digitali hanno bisogno delle *memorie* per immagazzinare i dati utilizzati e generati da questi tipi di circuiti.
Ci sono 3 tipi di memorie:

- *Memoria ad accesso casuale dinamica*: (DRAM)
- *Memoria ad accesso casuale statica*: (SRAM)
- *Memoria a sola lettura*: (ROM)

Panoramica

La memoria è organizzata come una matrice bidimensionale di celle di memoria. A ogni accesso la memoria può leggere o scrivere il contenuto di una riga della matrice. Questa riga viene specificata da un indirizzo (address). Il valore letto o scritto nella memoria viene chiamato dato (data).
Un componente con un numero N di bit di indirizzo e un numero M di bit di dato possiede $2N$ righe e M colonne. Ogni riga di dati viene chiamata *parola*.



componente di 4 parole x 3 bit

Celle di bit

I componenti di memoria vengono realizzati come matrici di *celle di bit* e ogni cella di dato è connessa a una *linea di parola* e una

linea di bit. Per ogni configurazione dei bit di indirizzo, la memoria attiva una sola linea di parola che, a sua volta, attiva le celle di bit presenti nella riga corrispondente.

Porte di memoria

Tutte le memorie possiedono una o più *porte*. Ognuna di queste fornisce un accesso in lettura e/o scrittura a un indirizzo di memoria.
Le memorie *multi-porta* possono accedere a più indirizzi nello stesso momento.

Tipi di memorie

Le memorie sono classificate in base a come immagazzinano i bit nella cella di bit.

- *Memorie ad accesso casuale* (RAM)
- *Memorie a solo lettura* (ROM)
  La RAM è una memoria *volatile*, ovvero perde traccia dei suoi dati una volta spenta.
  La ROM è una memoria *non volatile*, cioè trattiene i suoi dati per un tempo indefinito.

Memoria ad accesso casuale dinamica

La *RAM dinamica* (DRAM) memorizza un bit come presenza o assenza di carica in un condensatore.
In caso di *lettura*, il valore dato viene traferito dal condensatore alla linea di bit.
In caso di scrittura, il valore del dato viene trasferito dalla linea di bit al condensatore.
La lettura distrugge il valore del bit, quindi la parola deve essere *refreshata* dopo ogni lettura.

Memoria ad accesso casuale statica

In una *RAM statica* (SRAM) i bit immagazzinati in memoria non hanno bisogno di essere ricaricati.
A differenza della DRAM, se il rumore deteriora il valore del bit

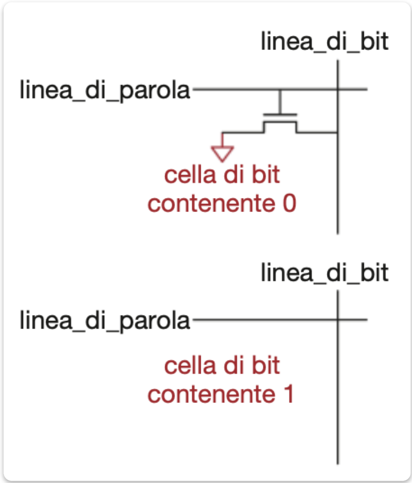
immagazzinato, i negatori collegati a croce lo riportano al valore di partenza.

Banchi di registri

I sistemi digitali utilizzano un certo numero di registri per immagazzinare variabili temporanee. Questo gruppo, chiamato *banco di registri*, viene solitamente costruito come una piccola componente SRAM multi-porta.

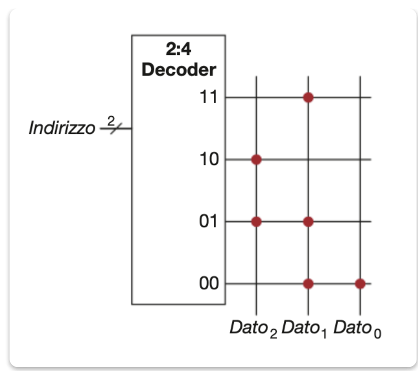
Memoria a sola lettura

Una memoria a sola lettura (ROM, Read Only Memory) memorizza un bit come presenza o assenza di un transistore. La Figura mostra una semplice cella di bit di una ROM.



Per leggere la cella, la linea di bit viene portata debolmente TRUE. Dopodiché, viene accesa la linea di parola. Se il transistore è presente, questo spinge la linea di bit a un valore FALSE; se invece è assente, la linea di bit resta TRUE.

Il contenuto di una ROM può essere indicato con la *notazione a punti*. La figura mostra una ROM da 4 parole x 3 bit:



Un punto posto all'intersezione tra una riga (linea di parola) e una colonna (linea di bit) indica che il bit di dato è uguale a 1.
Per esempio:

A	B	$Dato_0$	$Dato_1$	$Dato_2$		Y
0	0	1	1	0		011
0	1	0	1	1		110
1	0	0	0	1		100
1	1	0	1	0		010

Le ROM possono essere costruite utilizzando la logica a due livelli, con una serie di porte AND seguite da una serie di porte OR. Le porte AND producono tutti i mintermini possibili, quindi vanno a formare un decoder.

Esistono diversi tipi di ROM:

- *ROM programmabile*
- *ROM programmabile a fusibili*
- *ROM cancellabile*
- *ROM cancellabili elettricamente*

Reti logiche realizzate con componenti di memoria

Componenti di memoria utilizzati per svolgere funzioni logiche vengono chiamati *lookup table*.
Utilizzando una memoria per eseguire funzioni logiche, l'utente può consultare il valore di uscita corrispondente a una data combinazione di ingressi (indirizzo). Ogni indirizzo corrisponde a

una riga nella tabella delle verità e ogni bit di dato corrisponde a un valore di uscita.

Descrizione HDL delle memorie

Nelle *RAM*, le scritture avvengono sul fronte di salita del clock se l'abilitazione in scrittura è attivata (we, write enable). Le letture avvengono immediatamente invece.
Nelle *ROM* i contenuti vengono specificati nell'istruzione HDL_{case}.

```sv
module memoria_ram #(parameter N = 6, M = 32)
                                (input logic         clk,
                                 input logic         we,
                                 input logic [N-1:0] ind,
                                 input logic [M-1:0] ding,
                                 input logic [M-1:0] dusc);
        logic [M-1:0] mem [2**N-1:0];
        always_ff @(posedge clk)
                if (we) mem [ind] <= ding;
        assign dusc = mem[ind];
endmodule
```

```sv
module rom(input logic [1:0] ind,
                   input logic [2:0] dusc):
        always_comb
                case(ind)
                        2'b00: dusc = 3'b011
                        2'b01: dusc = 3'b110
                        2'b10: dusc = 3'b100
                        2'b11: dusc = 3'b010
                endcase
endmodule
```

5.6-Matrici logiche

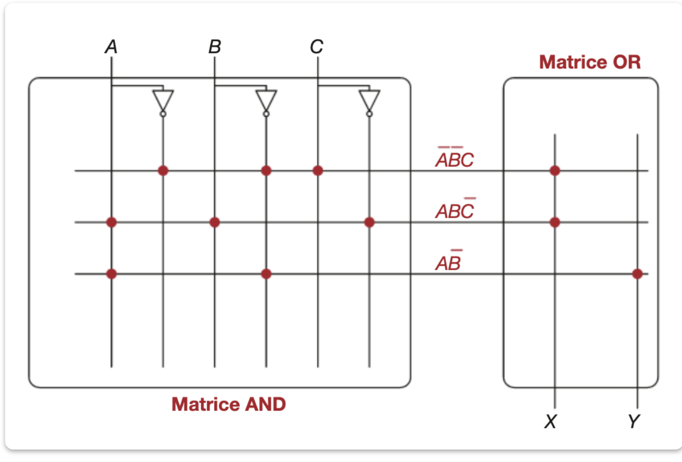
Cosa sono?

Come le memorie, anche le porte logiche possono essere organizzate in matrici regolari. Se le connessioni sono fatte in modo da essere programmabili, queste matrici logiche possono essere configurate in modo da eseguire qualsiasi funzione senza che l'utente debba connettere i fili in una maniera particolare. La struttura regolare semplifica il progetto. Le matrici logiche vengono prodotte in grandi quantità e sono quindi poco costose.

Matrici logiche programmabili

Le *matrici logiche programmabili* (PLA) realizzano funzioni combinatorie a due livelli nella forma somma di prodotti. Le PLA sono costituite da una matrice AND seguita da una matrice OR. Gli ingressi (in forma diritta e negata) pilotano la matrice AND, che produce degli implicanti, a loro volta combinati nella matrice OR per generare le uscite.

Ogni riga della matrice AND genera un implicante. I punti presenti in ogni riga della matrice AND indicano quali letterali sono inclusi nell'implicante. La matrice AND nella Figura forma tre implicanti: $\bar{A}\bar{B}C$, $AB\bar{C}$ e $A\bar{B}$. I punti nella matrice OR indicano invece quali implicanti fanno parte di ciascuna funzione di uscita.



> *I dispositivi logici programmabili semplici (SPLD, Simple Programmable Logic Devices) sono delle PLA modificate che aggiungono dei registri e diverse altre caratteristiche alle matrici AND/OR. Tuttavia, SPLD e PLA sono stati rimpiazzati in larga misura dalle FPGA, che sono più flessibili ed efficienti per costruire grandi sistemi.*

Matrici di porte logiche programmabili sul campo

Una *FPGA* è una matrice di porte logiche riconfigurabile. Le FPGA sono più potenti e più flessibili rispetto alle PLA per diverse ragioni:

- Consentono di realizzare sia funzioni combinatorie sia funzioni sequenziali
- Possono realizzare funzioni logiche su più livelli
  Le FPGA vengono costruite come una matrice di *elementi logici* (LE) configurabili, chiamati anche *blocchi logici configurabili* (CLB). Ogni LE può essere configurato in modo da eseguire funzioni combinatorie o sequenziali.
  Gli LE sono circondati da *elementi di ingresso/uscita* (IOE) per interfacciarsi al mondo esterno.

5.7-Riassunto

In questo capitolo sono stati introdotti i blocchi costruttivi utilizzati in molti sistemi digitali. Questi blocchi includono circuiti aritmetici, come ad esempio i sommatori, i comparatori, i traslatori, i moltiplicatori e i divisori; includono inoltre le reti sequenziali come i contatori e i registri a scorrimento, nonché le matrici di memoria e le matrici logiche. In questo capitolo sono anche state analizzate le rappresentazioni in virgola fissa e mobile per i numeri frazionari.

I sommatori sono la base della maggior parte dei circuiti aritmetici. Un semisommatore (half adder) somma due ingressi a 1 bit, A e B, e produce una somma e un riporto. Un sommatore intero (full adder) estende il semisommatore in modo che questo possa accettare anche un riporto di ingresso. N full adder possono essere collegati tra loro a cascata per formare un sommatore con propagazione di riporto che somma due numeri a N bit. Questo tipo di sommatore viene chiamato sommatore a propagazione di riporto a onda perché il riporto si propaga attraverso ognuno dei sommatori. È possibile co- struire sommatori più veloci utilizzando le tecniche lookahead e a prefissi.

Un sottrattore trasforma il secondo ingresso in negativo per poi sommarlo al primo. Un comparatore di valori sottrae un numero da un altro e ne determina il valore relativo basandosi sul segno del risultato. Un moltiplicatore calcola prodotti parziali utilizzando delle porte AND, e successivamente somma questi bit utilizzando dei full adder. Un divisore sottrae ripetutamente il numero divisore dal resto parziale e controlla il segno della differenza per determinare i bit del quoziente. Infine, un contatore utilizza un sommatore e un registro per effettuare un conteggio.

I numeri frazionari vengono rappresentati utilizzando le notazioni in virgola fissa o mobile. I numeri in virgola fissa sono analoghi ai numeri decimali, mentre i numeri in virgola mobile sono analoghi ai numeri rappresentati in notazione scientifica. I

numeri in virgola fissa utilizzano circuiti aritmetici ordinari, mentre i numeri in virgola mobile necessitano di circuiti più complessi per estrarre ed elaborare il segno, l'esponente e la mantissa.

Le grandi memorie sono organizzate in matrici di parole. Le memorie hanno una o più porte in grado di leggere e/o scrivere le parole. Le memorie volatili, come le SRAM e le DRAM, perdono il loro stato nel momento in cui non sono più alimentate. Una SRAM è più rapida di una memoria DRAM, ma richiede un numero maggiore di transistori. Un banco di registri è un piccolo componente SRAM a più porte. Le memorie non volatili, chiamate ROM, mantengono il loro stato anche in assenza di alimentazione. Nonostante il loro nome, la maggior parte delle ROM moderne può anche essere scritta.

Le matrici di memoria costituiscono anche un metodo regolare per la realizzazione di reti logiche. I componenti di memoria possono essere utilizzati come lookup table per eseguire funzioni combinatorie. Le PLA sono costituite da connessioni dedicate tra una matrice AND e una matrice OR configurabili, e svolgono solo funzioni combinatorie. Le FPGA sono invece composte da tante piccole lookup table e da registri, e sono in grado di svolgere funzioni sia combinatorie sia sequenziali. I contenuti delle lookup table e le loro interconnessioni possono essere configurati in modo da eseguire qualsiasi funzione logica.

6.2-Il linguaggio assembly

Il linguaggio *assembly* è la forma leggibile dall'uomo del linguaggio nativo del calcolatore. Ogni istruzione in linguaggio assembly specifica sia l'operazione da svolgere sia gli operandi da elaborare.

Istruzioni

L'operazione più comune nei calcolatori è l'addizione

```assembly
ADD a,b,c
```

La prima parte dell'istruzione assembly `ADD` è chiamata *mnemonico* e indica l'operazione da eseguire. Tale operazione deve essere eseguita su `b` e `c`, gli *operandi sorgente*, e il risultato deve essere scritto in `a`, l'*operando destinazione*.

Gli esempi qua sotto mostrano la sottrazione e una sequenza di somma e sottrazione:

```assembly
SUB a,b,c
```

```assembly
ADD t,b,c    ;t=b+c
SUB a,t,d    ;a=t+(-d)
```

I commenti si fanno con `;` e sono su una sola riga.

ARM è un calcolatore con architettura RISC (Reduced Instruction Set Computer) che minimizza la complessità hardware e la codifica necessaria per le istruzioni mentenendo piccolo l'insieme di diverse istruzioni.

6.7-Evoluzione dell'architettura ARM

Set di istruzione thumb

Le istruzioni Thumb sono lunghe 16 bit per aumentare la densità del codice macchina; sono identiche alle normali istruzioni ARM ma con alcune limitazioni in quanto:

- Possono accedere solo ai primi otto registri;
- Utilizzano un registro sia come sorgente sia come destinazione;
- Gestiscono solo immediati più corti;
- Non dispongono dell’esecuzione condizionata;
- Modificano sempre le flag di stato.

Praticamente tutte le istruzioni ARM hanno un equivalente Thumb.
Le istruzioni Thumb sono utili non solo perchè riducono le dimensioni e il costo della memoria, ma anche perchè permettono di usare un più economico bus a 16 bit per accedere alla memoria e riducono la potenza consumata per il fetch delle istruzioni

7.3-Processore a ciclo singolo

Come iniziamo?

Si parte con il progetto di una microarchitettura che esegue le istruzioni in un singolo ciclo.
I segnali di controllo determinano quale specifica istruzione viene eseguita dal percorso dati a ogni istante. L'unità di controllo contiene la logica combinatoria che genera i segnali di controllo adeguati in base all'istruzione corrente.

Percorso dati a ciclo singolo

Il *program counter* contiene l'indirizzo dell'istruzione da eseguire. Per prima cosa bisogna leggere l'istruzione dalla memoria istruzioni. La figura mostra che il PC è semplicemente collegato all'ingresso di indirizzo della memoria istruzioni, che emette l'istruzione a 32 bit, denominata `Instr`, in modo che possa essere prelevata dal processore (*fetch*).

Figura 7.2 Fetch dell'istruzione da memoria.

LDR

Per eseguire l'istruzione `LDR`, il prossimo passo è leggere il registro sorgente contenente l'indirizzo base, cioè i bit $Instr_{19:16}$. Questi bit vengono collegati agli ingressi di indirizzo di una delle porte del *banco di registri di lavoro*(register file), la porta $A1$, come mostrato in figura. Il banco dei registri emette il valore del registro all'uscita $RD1$.

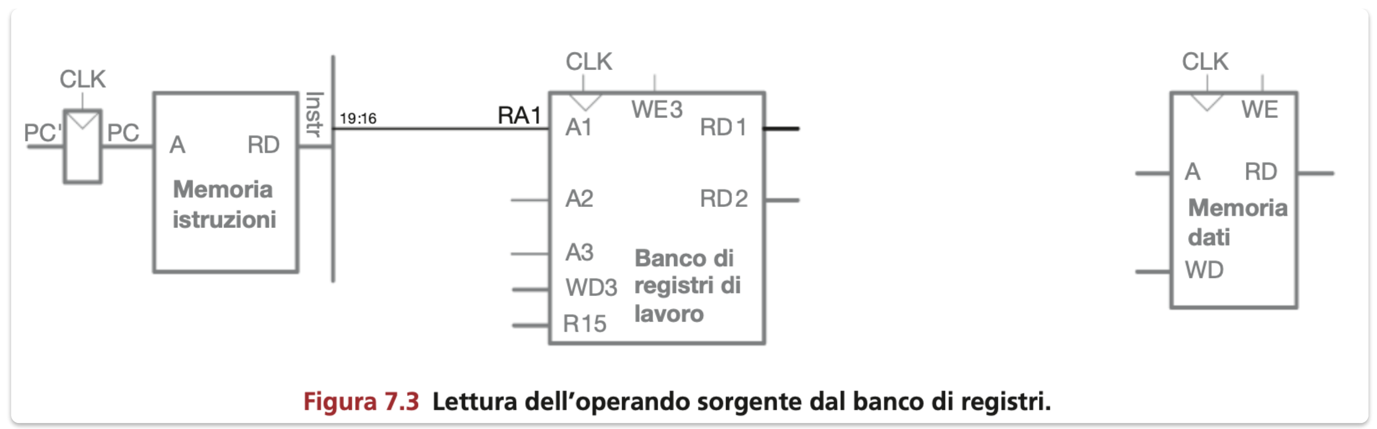


Figura 7.3 Lettura dell'operando sorgente dal banco di registri.

L'istruzione `LDR` richiede anche uno spiazzamento: è un valore senza segno esteso con zeri fino a 32 bit.

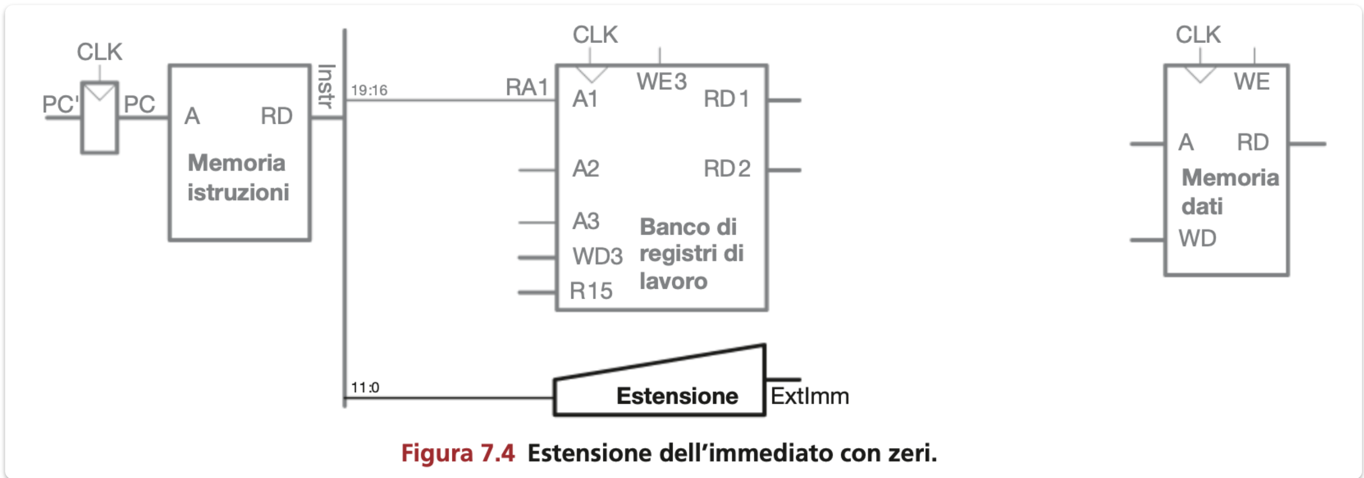


Figura 7.4 Estensione dell'immediato con zeri.

Il processore deve sommare lo spiazzamento al registro di base per ottenere l'indirizzo di memoria da cui leggere il dato. La figura introduce un'ALU per eseguire tale somma.

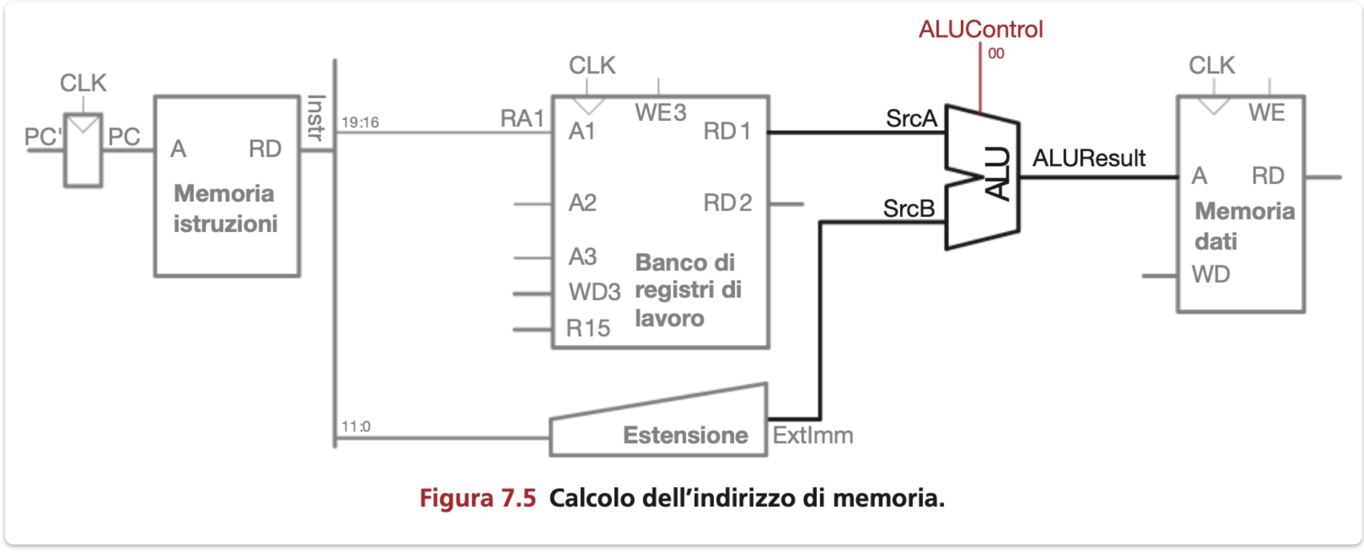


Figura 7.5 Calcolo dell'indirizzo di memoria.

L'ALU riceve due operandi: $SrcA$ e $SrcB$ che provengono dal banco di registri e dall'immediato esteso a 32 bit.
La porta $A3$ è la porta di scrittura. Il registro destinazione per `LDR` è specificato nei bit $Instr_{15:12}$.
Per l'istruzione `LDR`, $ALUcontrol$ deve avere valore 00 per indicare la somma, e $ALUresult$ viene inviato alla memoria dati come indirizzo della parola da leggere.

Il dato viene emesso dalla memoria dati sul bus $ReadData$ e scritto nel registro destinazione alla fine del ciclo, come mostrato in figura:

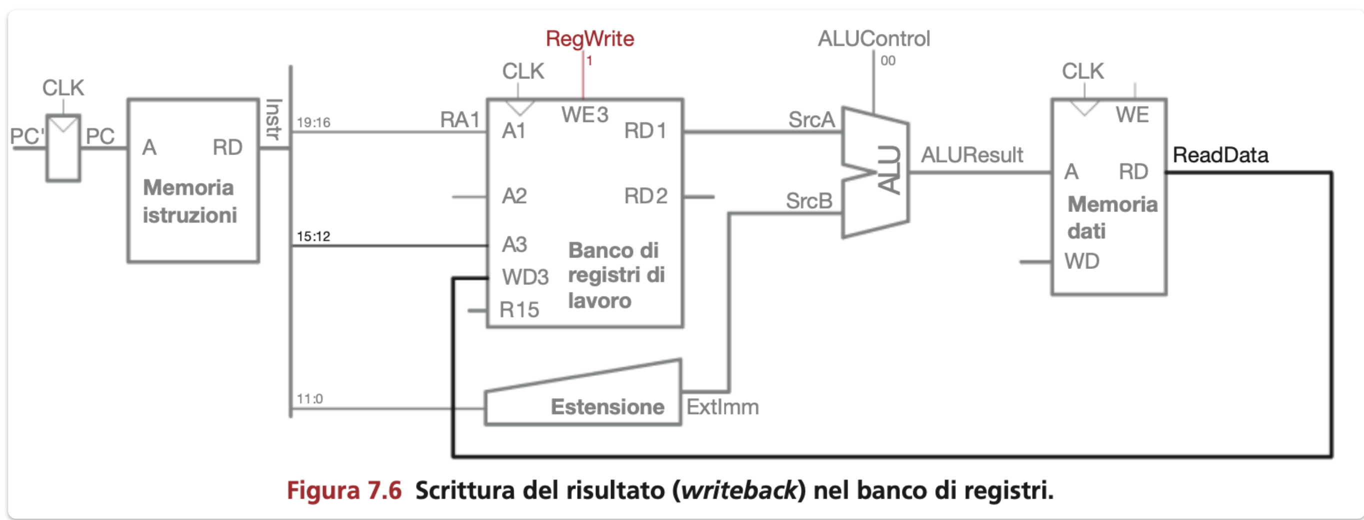


Figura 7.6 Scrittura del risultato (*writeback*) nel banco di registri.

Il bus $ReadData$ è collegato agli ingressi di dato della porta di scrittura $WD3$ del banco di registri. Un segnale di controllo denominato $RegWrite$ è collegato all'abilitazione alla scrittura della porta 3, $WE3$, e viene attivato durante l'istruzione `LDR` per scrivere il dato letto da memoria del banco di registri.
Mentre l'istruzione viene eseguita, il processore deve calcolare l'indirizzo dell'istruzione successiva, PC'. L'istruzione successiva si trova a $PC + 4$ e viene usato un sommatore per incrementare di 4 come in figura:

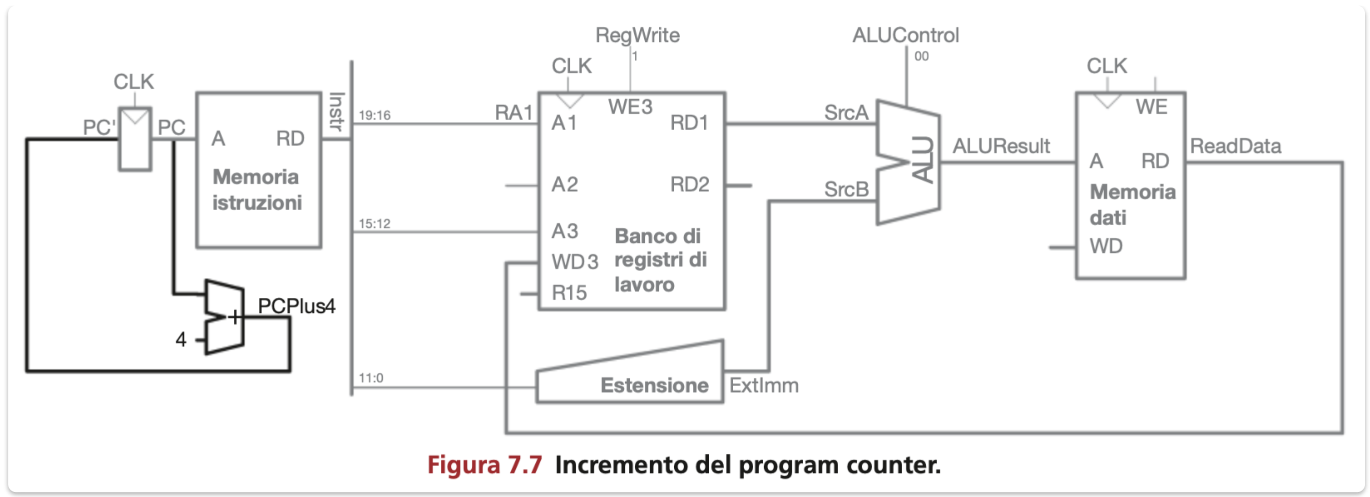


Figura 7.7 Incremento del program counter.

Il nuovo indirizzo viene scritto nel PC in corrispondenza del prossimo fronte di salita del clock. Questo completa il percorso dati per l'istruzione `LDR`, tranne nel caso particolare dove il registro base o il registro destinazione è $R15$. Nell'architettura ARM leggere dal registro $R15$ restituisce $PC + 8$. Serve quindi un altro sommatore per incrementare ulteriormente il PC e passare il risultato alla porta $R15$ del banco di registri. Scrivere nel $R15$ modifica anche il PC. Il valore di PC può venire da $ReadData$ invece che da $PCplus4$. Serve quindi un multiplexer per scegliere tra le due possibilità. $PCsrc$ è messo a 0 per selezionare $PCplus4$ e a 1 per selezionare $ReadData$.

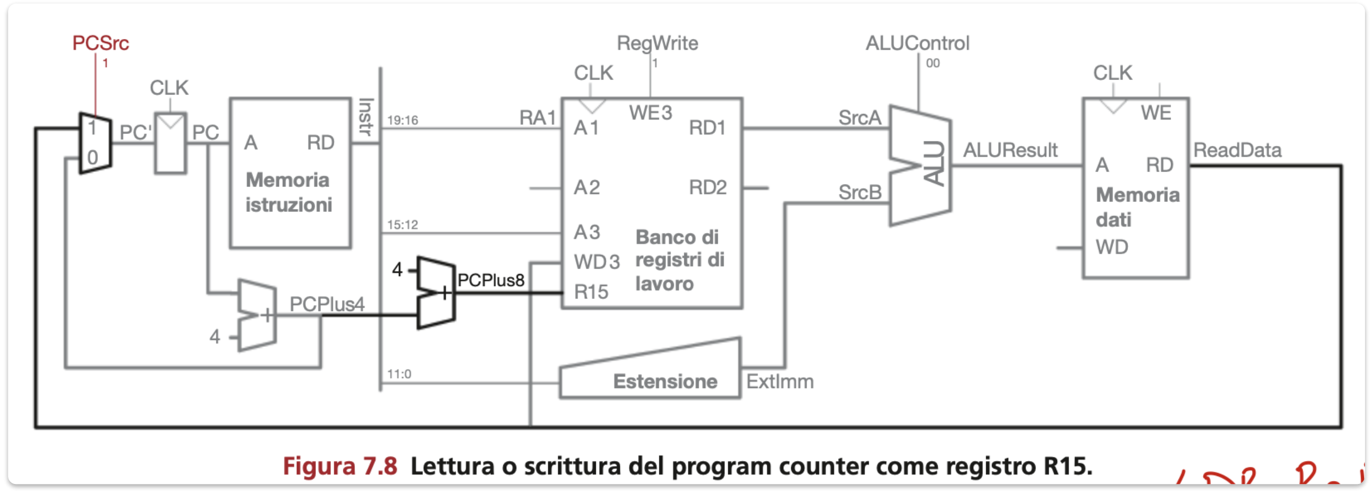


Figura 7.8 Lettura o scrittura del program counter come registro R15.

STR

Si può estendere il percorso dati per gestire anche l'istruzione `STR`. `STR` legge un indirizzo base dalla porta 1 del banco di registri ed estende con zeri l'immediato. L'ALU somma l'immediato esteso all'indirizzo base per trovare l'indirizzo di memoria.

Tutti questi passaggi sono già supportati dal percorso dati.
L'istruzione `STR` legge anche un secondo registro dal banco e lo scrive nella memoria dati. La figura mostra le nuove connessione necessarie per quest'operazione.

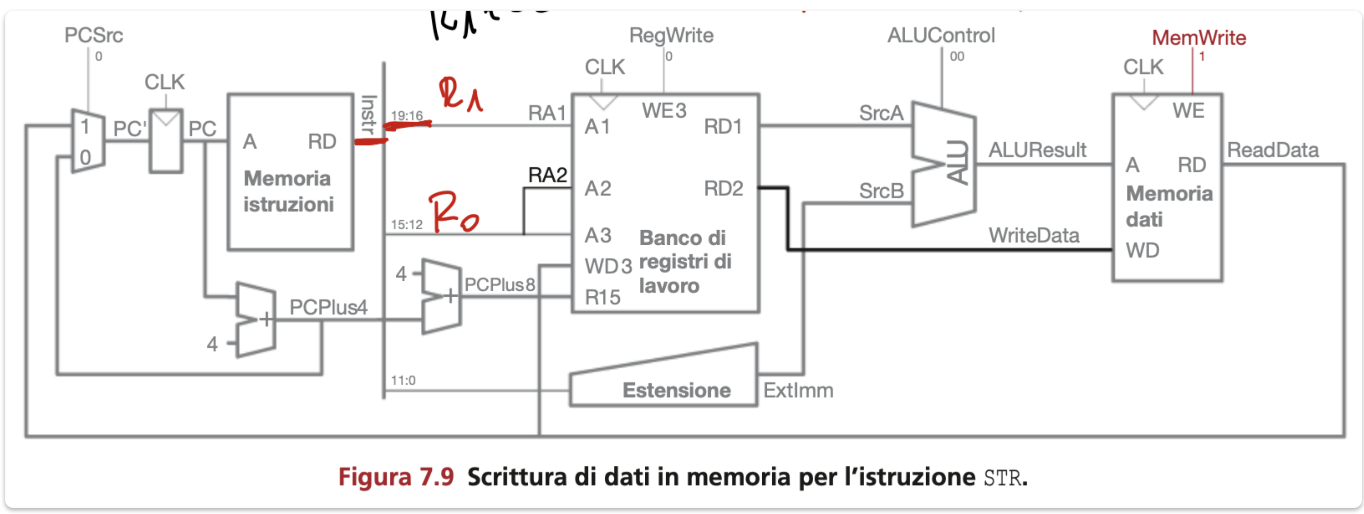


Figura 7.9 Scrittura di dati in memoria per l'istruzione STR.

Il registro è specificato nel campo Rd, cioè i bit $Instr_{15:12}$ collegati alla porta $A2$ del banco di registri. Il valore del registro viene emesso sulla porta $RD2$, collegata alla porta di scrittura WD della memoria dati. WE è controllata dal segnale $MemWrite$: per `STR` deve essere $MemWrite = 1$ per scrivere il dato; $ALUcontrol = 00$ per sommare l'indirizzo di base con lo spiazzamento; $RegWrite = 0$ perché non si deve scrivere nulla nel banco di registri.

Istruzioni di elaborazione dati con indirizzamento immediato

Si passa ora all'estensione del percorso dati per gestire le istruzioni di elaborazione dat: `ADD,SUB,AND,ORR`.
Tutte le istruzioni leggono un registro sorgente dal banco e un immediato dai bit meno significativi dell'istruzione, eseguono un'operazione dell'ALU sui valori così ricavati e scrivono il risultato in un registro. Possono quindi essere eseguite con lo stesso hardware semplicemente utilizzando differenti segnali $ALUcontrol$:

- 00: `ADD`
- 01: `SUB`

- `10`: `AND`
- `11`: `ORR`
  L'ALU genera 4 flag, $ALUflags_{3:0}$ (Zero, Negative, Carry, Verflow), che vengono inviate all'unità di controllo.
  La figura mostra il percorso dati esteso per gestire le istruzioni di elaborazione dati con indirizzo immediato.

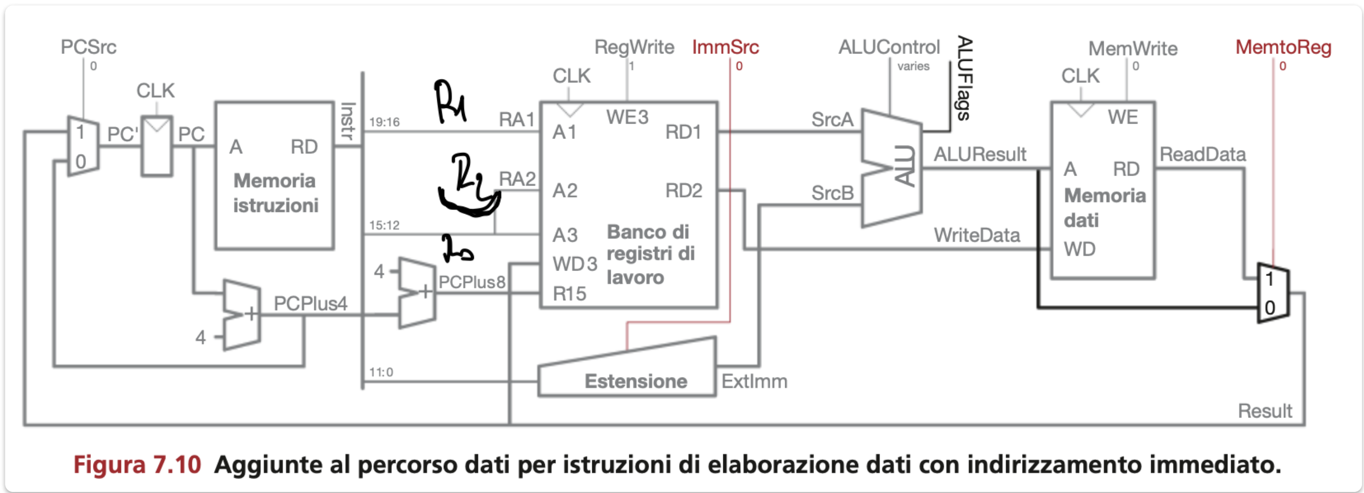


Figura 7.10 Aggiunte al percorso dati per istruzioni di elaborazione dati con indirizzamento immediato.

Queste istruzioni però usano un immediato a 8 bit invece che a 12, quindi serve il nuovo segnale $ImmSrc$ al blocco circuitale dell'estensione:

- 0: $ExtImm$ viene esteso con zeri da $Instr_{7:0}$ in avanti per le istruzioni di elaborazione dati;
- 1: $ExtImm$ viene esteso con zeri da $Instr_{11:0}$ in avanti per le istruzioni `LDR` e `STR`.

Le istruzioni di elaborazione dati devono scrivere nel banco di registri il risultato dell'ALU: $ALUresult$. Serve quindi un ulteriore multiplexer per selezionare tra $ReadData$ e $ALUresult$, la cui uscita è denominata $Result$. Il multiplexerè pilotato da un nuovo segnale di controllo, $MemtoReg$:

- 0: per le istruzioni di elaborazioe dati per selezionare come $Result$ il risultato dell'ALU $ALUresult$;
- 1: per `LDR` per selezionare $ReadData$.
  Per `STR` il valore di $MemtoReg$ non interessa perchè `STR` non scrive nel banco di registri.

Istruzioni di elaborazione dati con indirizzamento a registro

Queste istruzioni ricevono il secondo operando sorgente da Rm, indicato dai bit $Instr_{3:0}$ invece che dall'immediato. Servono quindi ulteriori multiplexer agli ingressi del banco di registri e dell'ALU per selezionare questo registro sorgente, come mostrato in figura:

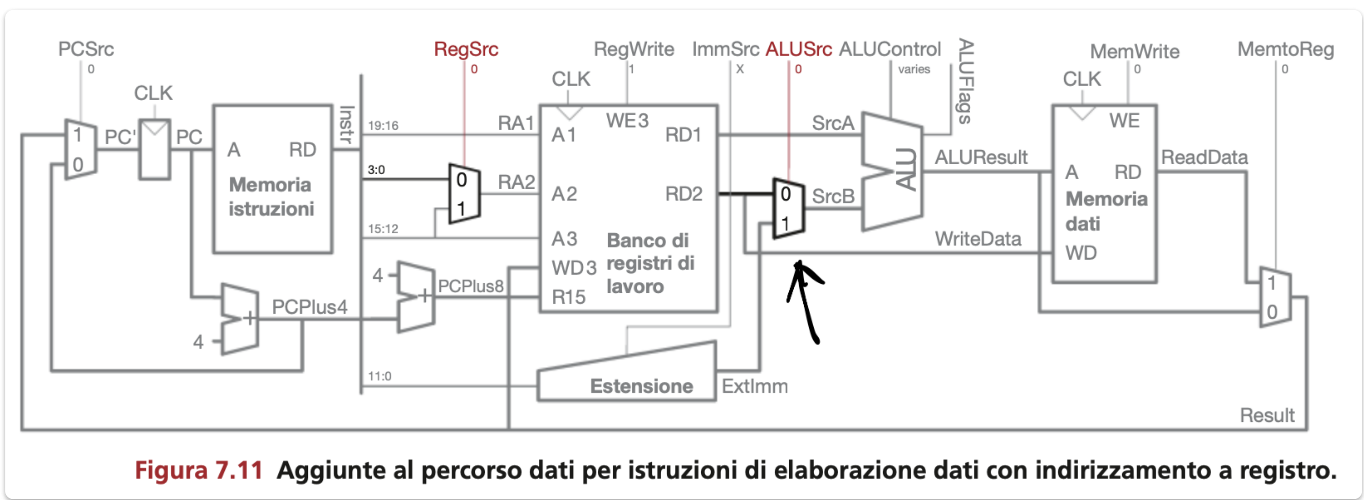


Figura 7.11 Aggiunte al percorso dati per istruzioni di elaborazione dati con indirizzamento a registro.

$RA2$ viene prelevato dal campo Rd (i bit $Instr_{15:12}$) per l'istruzione `STR` e dal campo Rm (i bit $Instr_{3:0}$) per le istruzioni di elaborazione dati con indirizzamento a registro in base al valore del segnale di controllo $RegSrc$.
$ALUsrc$ seleziona come secondo operando dell'ALU $ExtImm$ per le istruzioni che usano indirizzamento immediato e il banco di registri per le istruzioni che usando indirizzamento a registro.

B

Infine si estende il percorso dati per gestire l'istruzione di salto `B`, come mostrato in figura:

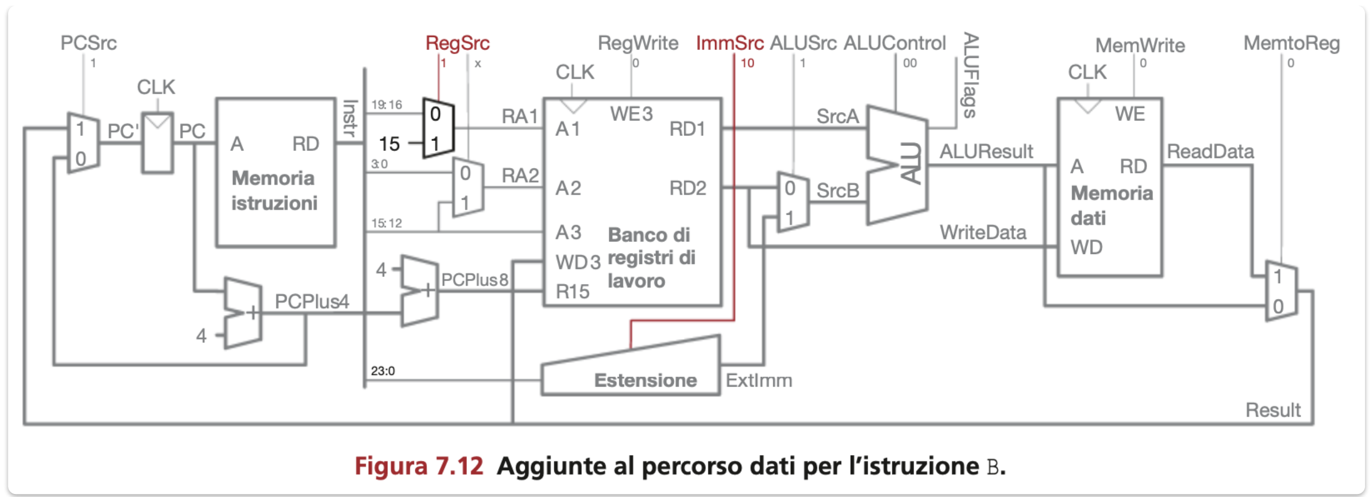


Figura 7.12 **Aggiunte al percorso dati per l'istruzione** B.

L'istruzione somma un immediato a 24 bit a $PC + 8$ e scrive il risultato in PC. L'immediato deve essere moltiplicato per 4 ed esteso con segno. Quindi la logica di estensione richiede un'ulteriore modalità di funzionamento: $ImmSrc$ deve essere aumentato a 2 bit, con le codifiche riportate nella tabella:

Tabella 7.1 **Codifica di ImmSrc.**

ImmSrc	ExtImm	Descrizione
00	{24 0s} $Instr_{7:0}$	Immediato senza segno a 8 bit per elaborazione dati
01	{20 0s} $Instr_{11:0}$	Immediato senza segno a 12 bit per le istruzioni LDR/STR
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	Immediato con segno a 24 bit moltiplicato per 4 per l'istruzione B

$PC + 8$ è letto dalla prima porta del banco di registri, quindi serve un multiplexer per selezionare $R15$ come ingresso $RA1$: questo multiplexer è pilotato da un altro bit di $RegSrc$ che seleziona $Instr_{19:16}$ per le altre istruzioni e 15 per l'istruzione `B`. $MemtoReg$ è portato a 0 e $PCsrc$ a 1 per selezionare il nuovo valore di PC da $ALUresult$.

> *In questo modo il progetto del percorso dati per il processore a ciclo singolo è completo*

Unità di controllo a ciclo singolo

L'unità di controllo genera i segnali di controllo sulla bse dei campi $cond, op\ e funct$ dell'istruzione (i bit $Instr_{31:28}$, $Instr_{27:26}$, $Instr_{25:20}$), come pure dalle flag della ALU e del fatto che il registro destinazione sia il PC. L'unità di controllo deve anche memorizzare e aggiornare opportunamente le flag di stato. La

figura mostra l'intera struttura del processore a ciclo singolo con l'unità di controllo collegata al percorso dati.

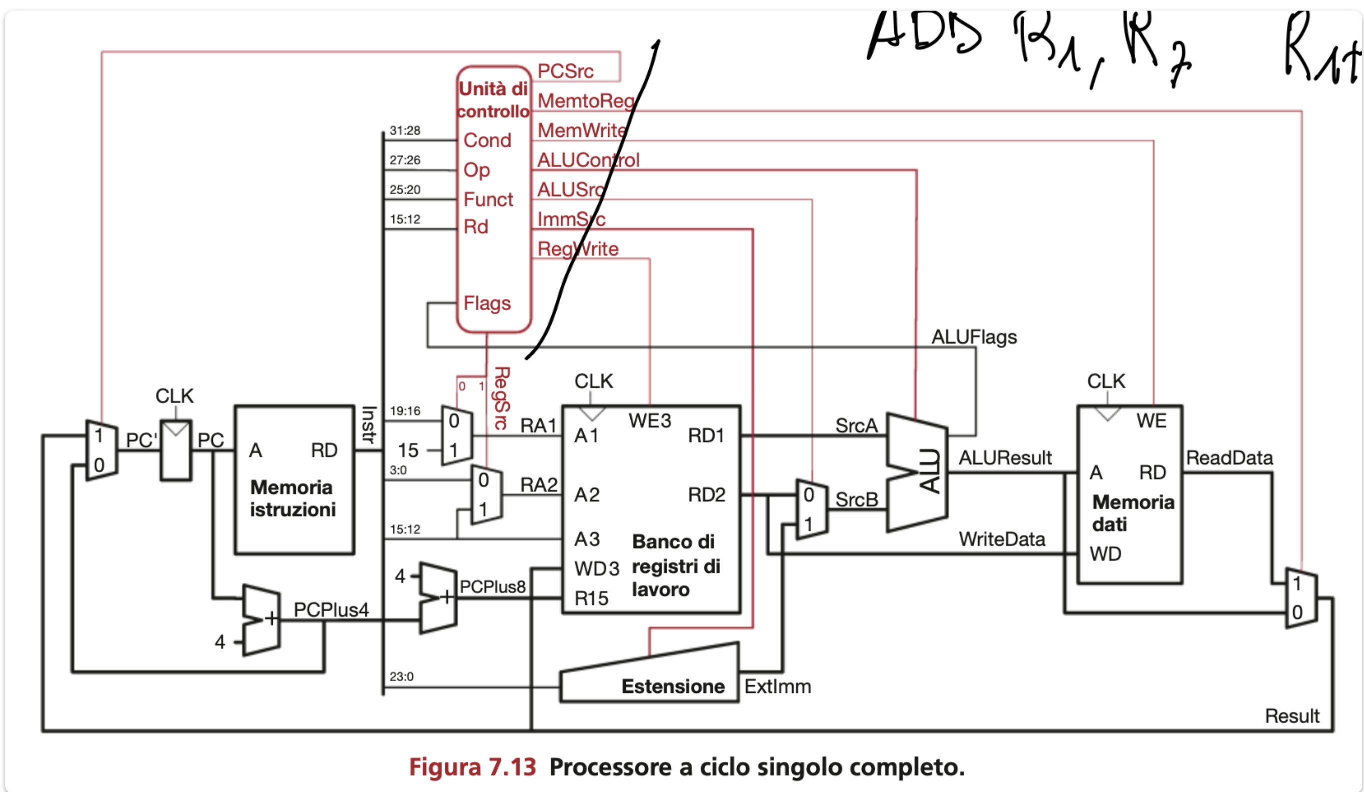


Figura 7.13 **Processore a ciclo singolo completo.**

La figura sotto mostra in dettaglio la struttura dell'unità di controllo:

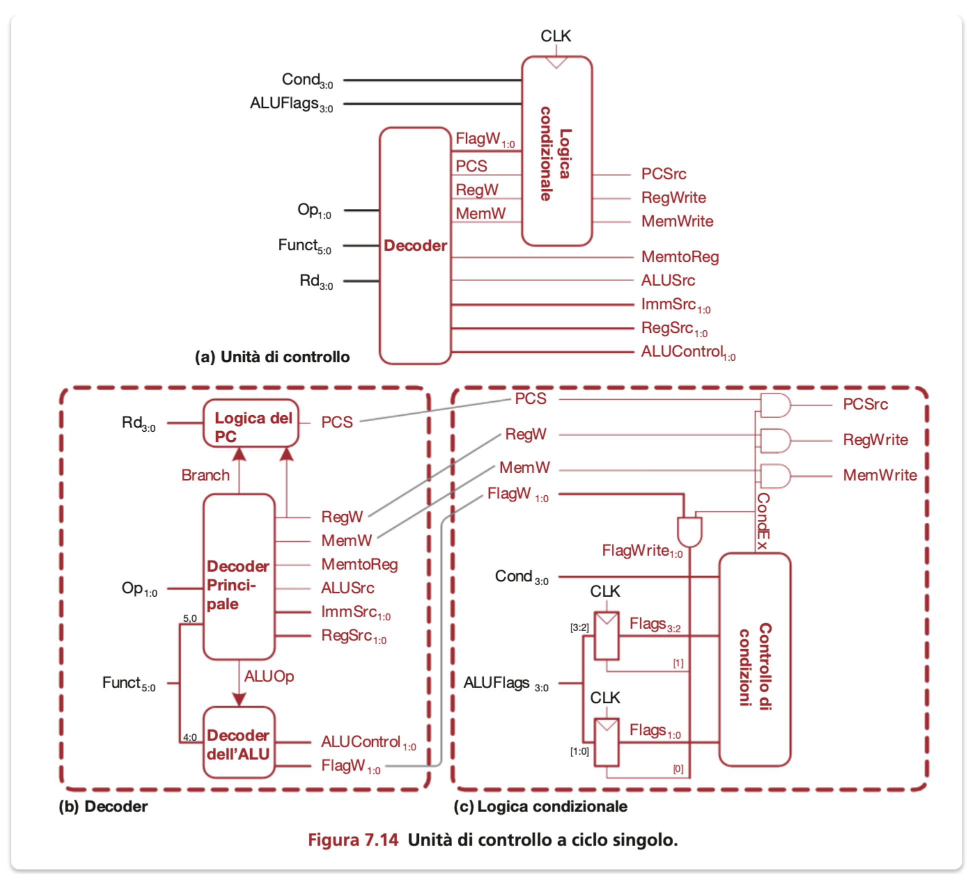


Figura 7.14 Unità di controllo a ciclo singolo.

è divisa in 2 parti:

- *Decoder*: genera i segnali di controllo in base a $Instr$;
- *Logica condizionale*: mantiene le flag di stato e abilita gli aggiornamenti dello stato architetturale quando l'istruzione deve essere eseguita in modo condizionato.

Il decoder è costituito da un *Decoder Principale* che genera la maggior parte dei segnali di controllo, da un *Decoder dell'ALU* che usa i campi $funct$ per determinare il tipo di operazione aritmetica, e da una *logica del PC* per decidere se il PC deve essere modificato per un'istruzione di salto oppure per una scrittura in $R15$.
Il comportamento del Decoder Principale è descritto dalla tabella delle verità riportata in figura:

Tabella 7.2 Tabella delle verità del Decoder Principale.

Op	$Funct_5$	$Funct_0$	Tipo	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

Il Decoder Principale determina il tipo di istruzione:

- *Elaborazione dati a registro*
- *Elaborazione dati a immediato*
- `STR`
- `LDR`
- `B`
  e genera opportunamente i segnali di controllo per il percorso dati.
  Invia $MemtoReg$, $ALUsrc$, $ImmSrc_{1:0}$, $RegSrc_{1:0}$ direttamente al percorso dati. Le abilitazioni di scrittura $MemW$ e $RegW$ devono invece passare nella *logica condizionale* prima di diventare i segnale di controllo $MemWrite$ e $RegWrite$ del percorso dati. Tali segnali possono essere forzati a 0 dalla logica condizionale se la condizione di scrittura non si è verificata. Il Decoder Principale genera anche i segnali $Branch$ e $ALUop$ usati all'interno dell'unità di controllo per indicare rispettivamente che l'istruzione è `B` oppure un'istruzone di elaborazione dati.

Il comportamento del *Decoder dell'ALU* è descritto dalla tabella delle verità riportata in tabella:

Tabella 7.3 **Tabella delle verità del Decoder dell'ALU.**

ALUOp	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Tipo	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

Per le istruzioni di elaborazioni dati il Decoder dell'ALU genera $ALUcontrol$ in base al tipo di istruzione (`ADD, SUB, AND, ORR`). Inoltre attiva $FlagW$ per aggiornare le flag di stato quando il bit S vale 1. `ADD e SUB` aggiornano tutte le flag, mentre `AND e ORR` aggiornano solo le flag N e Z: servono quindi due bit per $FlagW$:

- $FlagW_1$: per aggiornare N e Z ($Flags_{3:2}$)
- $FlagW_0$: per aggiornare C e V ($Flags_{1:0}$)
  $FlagW_{1:0}$ è forzato a 0 dalla logica condizionale quando la condizione non si è verificata ($CondEx = 0$).
  La Logica del PC controlla se l'istruzione è una scrittura di $R15$ o un salto, che implicano aggiornamento del PC.
  L'espressione logica corrispondente è:

$$PCS = ((Rd == 15) \& RegW) | Branch$$

  PCS può essere forzato a 0 dalla Logica Condizionale prima di essere inviato al percorso dati come $PCsrc$.
  La logica condizionale determina se l'istruzione deve essere eseguita ($CondEx$) in base al campo $cond$ dell'istruzione e ai valori attuali delle flag N, Z, C e V.
  Se l'istruzione non deve essere eseguita, le abilitazione alla scrittura e il segnale $PCsrc$ sono forzati a 0 in modo tale che lo stato architetturale non venga modificato. La logica

condizionale aggiorna anche alcune o tutte le flag ai valori $ALUflags$ quando $FlagW$ è attivato dal *Decoder dell'ALU* e la condizione di di esecuzione dell'istruzione si è verificata ($CondEx = 1$)

Istruzioni aggiuntive

Si possono aggiungoere i supporti per l'istruzione di confronto (`CMP`) e per i modi di indirizzamento nei quali il secondo operando sorgente è un registro traslato.
Per gestire alcune istruzioni aggiuntive basta estendere i decoder, mentre in altri casi serve aggiungere hardware al percorso dati.

Analisi delle prestazioni

Ogni istruzione occupa un ciclo di clock nel processore a ciclo singolo, quindi il CPI vale 1.

7.4-Processore multi ciclo

Come iniziamo?

Il processore a ciclo singolo ha tre elementi di debolezza. In primo luogo, richiede memorie separate per istruzioni e dati, mentre la maggior parte dei processori ha una sola memoria esterna che contiene sia istruzioni sia dati. In secondo luogo, richiede un ciclo di clock abbastanza lungo da consentire l'esecuzione dell'istruzione più lenta (`LDR`) anche se molte istruzioni potrebbero essere più veloci. Infine, richiede tre circuiti sommatori (uno nell'ALU, e due per la Logica del PC): circuiti relativamente costosi soprattutto se devono essere veloci.

Il processore multi ciclo si propone di eliminare queste tre debolezze dividendo l'istruzione in una sequenza di passi più brevi: in ciascun passo, il processore legge o scrive in memoria o nel banco di registri, oppure usa l'ALU. L'istruzione viene letta da memoria in un passo, e i dati possono essere letti o scritti in passi successivi, quindi è possibile usare una sola memoria per contenere entrambi. Istruzioni diverse usano numeri diversi di passi, quindi le istruzioni più semplici vengono portate a termine in meno tempo rispetto a quelle più complesse. E il processore richiede un solo circuito sommatore, usato a scopi differenti nei diversi passi.

Percorso dati multi ciclo

Si inizia ancora dal progetto dalla memoria e dallo stato architetturale, come mostrato nella Figura:

Figura 7.19 **Elementi di stato con memoria istruzioni e dati unificata.**

Qui si possono riunire istruzioni e dati in un'unica memoria, si può leggere l'istruzione in un ciclo e leggere/scrivere un dato in un ciclo diverso. PC e banco di registri rimangono gli stessi.
Il PC contiene l'indirizzo dell'istruzione da eseguire. Il primo passo è dunque la lettura (*fetch*) di tale istruzione dalla memoria. La Figura mostra che il PC viene semplicemente connesso all'ingresso di indirizzo della memoria.

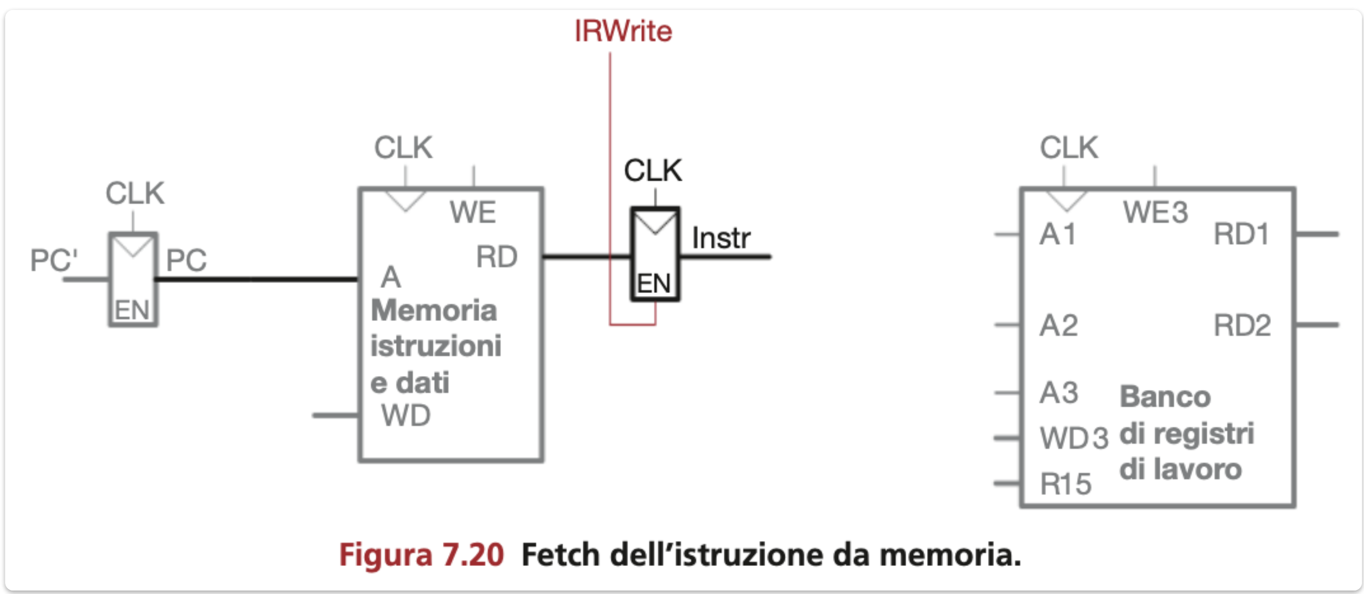


Figura 7.20 **Fetch dell'istruzione da memoria.**

L'istruzione viene letta e memorizzata in un registro non architetturale, il registro istruzioni (IR, Instruction Register), in modo da renderla disponibile nei passi successivi. IR riceve un segnale di abilitazione, denominato IR_{write}, che viene attivato quando IR deve essere caricato con la nuova istruzione.

LDR

Come già fatto per il processore a ciclo singolo, si inizia a definire le interconnessioni del percorso dati per eseguire l'istruzione `LDR`.
Dopo la fase di fetch di `LDR`, il passo successivo è la lettura del registro sorgente contenente l'indirizzo base. Tale registro è specificato nel campo Rn dell'istruzione, cioè i bit $Instr_{19:16}$: tali bit vengono collegati all'ingresso di indirizzo $A1$ del banco di registri, come mostrato nella Figura:

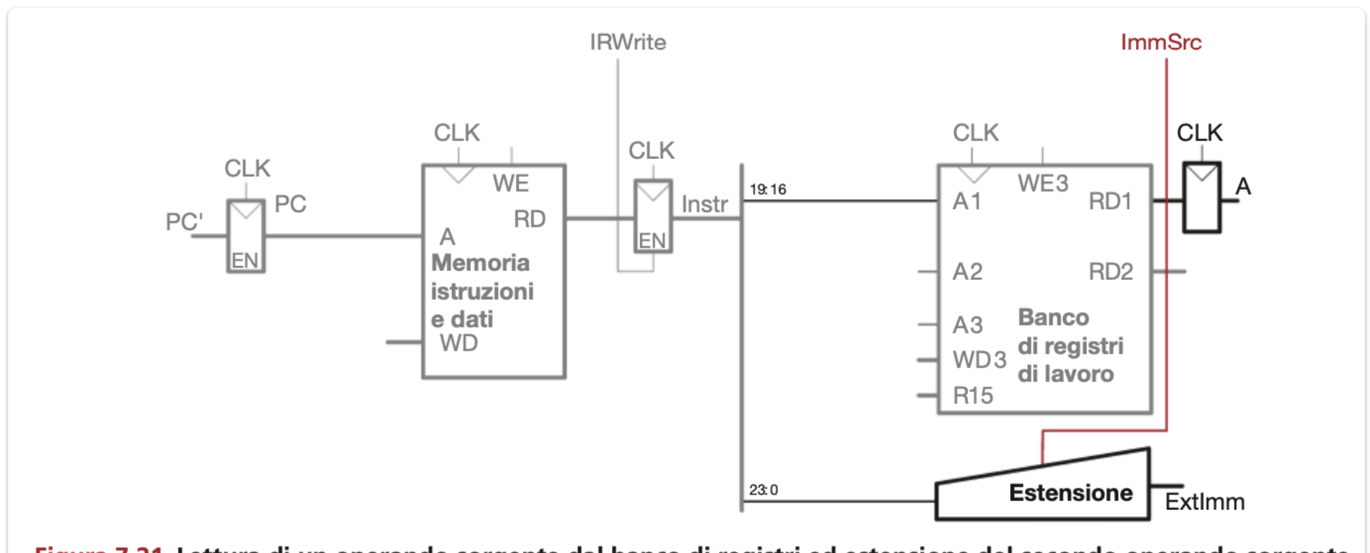


Figura 7.21 **Lettura di un operando sorgente dal banco di registri ed estensione del secondo operando sorgente dal campo immediato.**

Il banco emette il contenuto del registro indirizzato sull'uscita di dato $RD1$, e il valore viene memorizzato in un altro registro non architetturale: A.
L'istruzione `LDR` richiede anche uno spiazzamento a 12 bit, che deve essere esteso con zeri a 32 bit.
Il blocco di estensione riceve un segnale di controllo $ImmSrc$ che specifica se si deve estendere un immediato a 8, 12 o 24 bit per i vari tipi di istruzioni. L'immediato esteso a 32 bit è denominato $ExtImm$.
L'indirizzo del dato da caricare è la somma dell'indirizzo base e dello spiazzamento. Si può usare l'ALU per fare questa somma, come mostrato nella Figura:

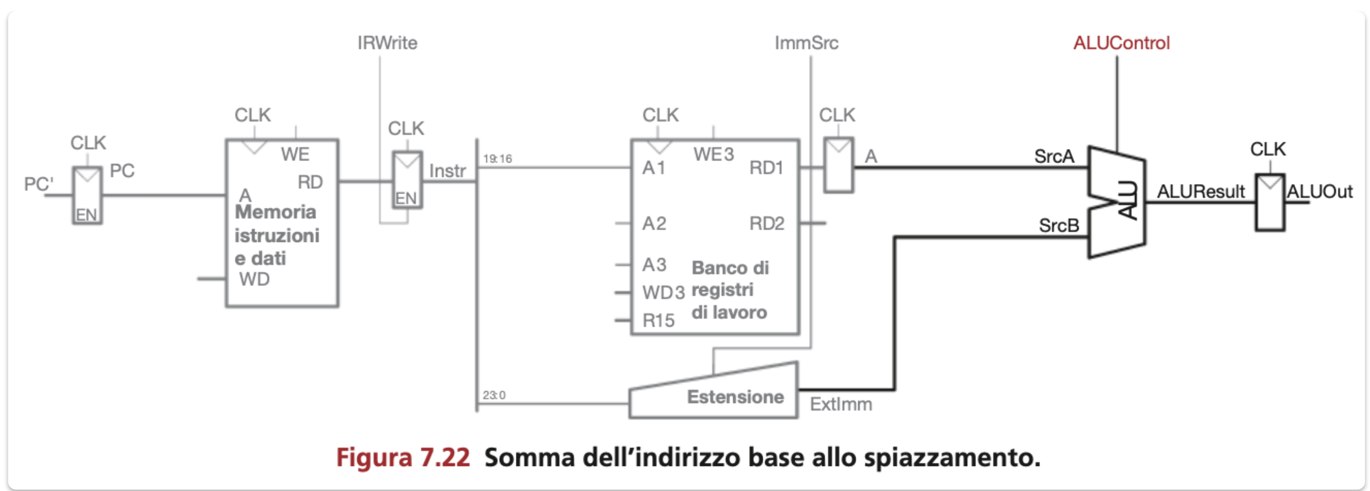


Figura 7.22 **Somma dell'indirizzo base allo spiazzamento.**

$ALUControl$ deve valere 00 per eseguire la somma, e il risultato $ALUResult$ viene memorizzato in un altro registro non architetturale, denominato $ALUOut$.

Il passo successivo è il caricamento del dato dalla parola di memoria il cui indirizzo è appena stato calcolato. Si aggiunge un multiplexer davanti alla memoria per selezionare l'indirizzo, Adr, dal PC oppure da $ALUOut$, in base al segnale di selezione $AdrSrc$, come mostrato nella Figura:

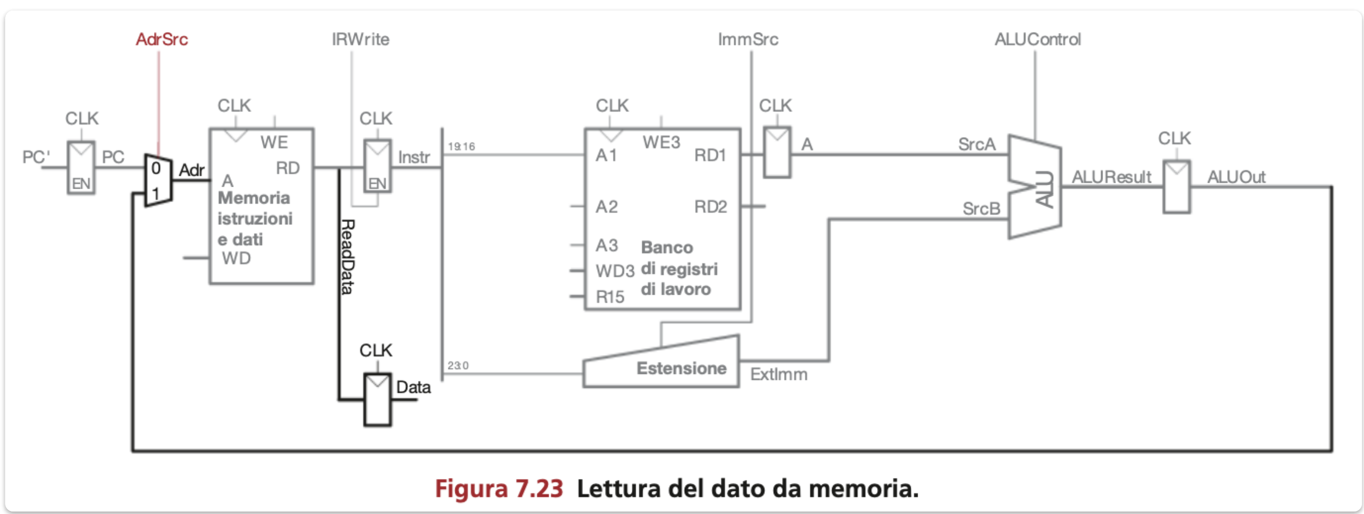


Figura 7.23 **Lettura del dato da memoria.**

Il dato letto dalla memoria viene memorizzato in un altro registro non architetturale: $Data$. Si noti che il multiplexer sull'indirizzo consente di riutilizzare la stessa memoria durante l'esecuzione dell'istruzione `LDR`: al primo passo l'indirizzo di memoria proviene dal PC per fare il *fetch* dell'istruzione, in un passo successivo l'indirizzo proviene da $ALUOut$ per caricare un dato. Quindi $AdrSrc$ deve avere valori diversi in passi diversi.

Infine, il dato deve essere scritto nel banco di registri, come mostrato dalla Figura:

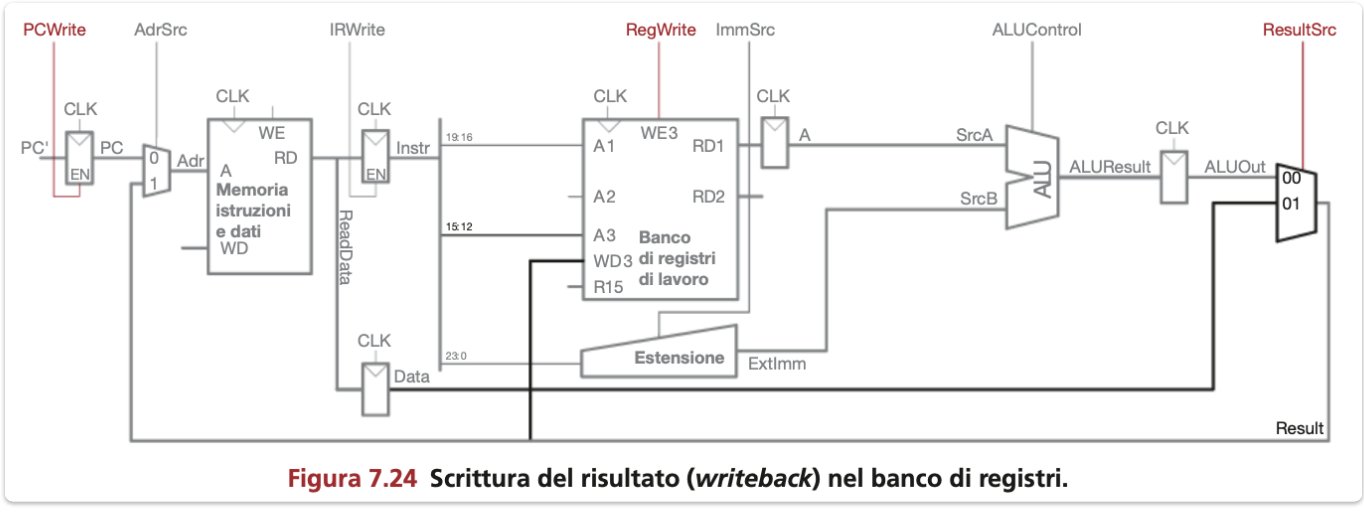


Figura 7.24 Scrittura del risultato (*writeback*) nel banco di registri.

Il registro destinazione è indicato dal campo Rd dell'istruzione: $Instr_{15:12}$. Il valore da scrivere proviene dal registro $Data$. Invece di collegare direttamente il registro $Data$ alla porta di scrittura $WD3$ del banco di registri, conviene aggiungere un multiplexer sul bus $Result$, in modo da poter scegliere $ALUOut$ oppure $Data$ prima di inoltrare $Result$ alla porta di scrittura del banco di registri. Questo serve perché altre istruzioni dovranno scrivere in un registro il risultato dell'ALU. Il segnale $RegWrite$ vale 1 per indicare che il banco di registri deve essere modificato.

Mente si svolgono tutti questi passi, il processore deve anche aggiornare il program counter sommando 4 al vecchio valore PC. Nel processore multi ciclo si può usare l'ALU che non è utilizzata durante la fase di *fetch*. Per fare questo, si devono inserire dei multiplexer per selezionare PC e il valore costante 4 come ingressi all'ALU, come mostrato nella Figura:

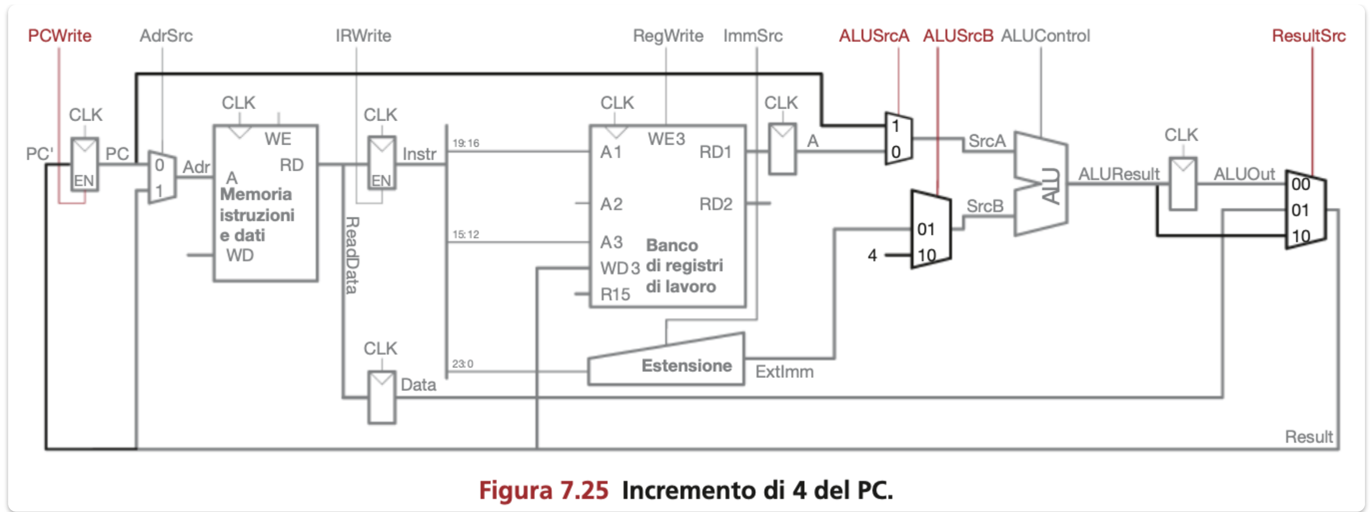
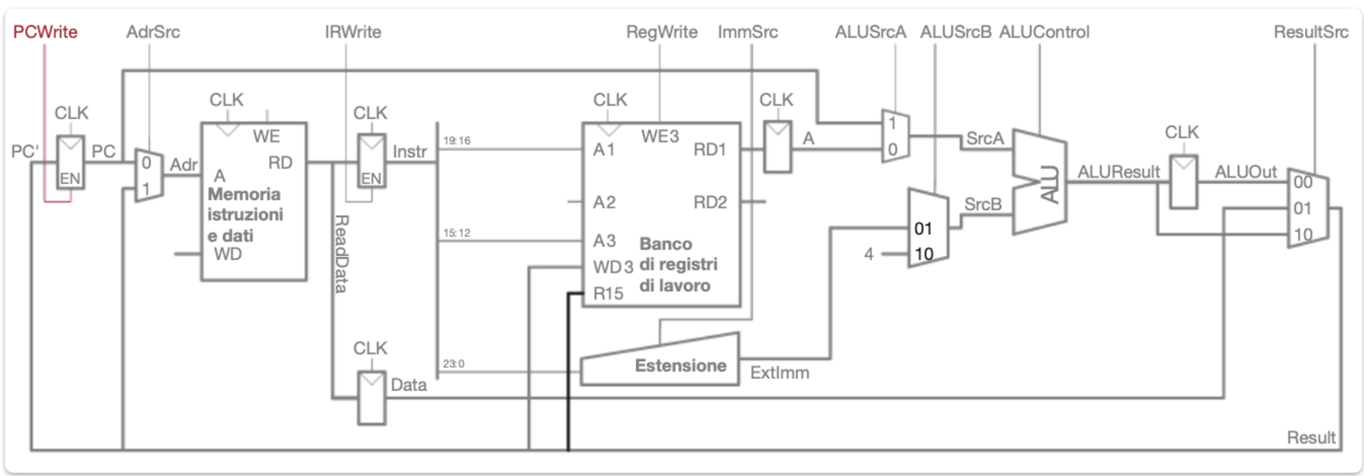


Figura 7.25 Incremento di 4 del PC.

Per aggiornare il PC, l'ALU somma $SrcA$ (cioè PC) e $SrcB$ (cioè 4) e il risultato deve essere scritto nel program counter. Il multiplexer controllato da $ResultSrc$ deve poter selezionare questa somma da $ALUResult$ invece che da $ALUOut$: serve quindi un terzo ingresso al multiplexer. Il segnale di controllo $PCWrite$ abilita la scrittura nel PC solo nei cicli opportuni.

Per $R15$ si può ottenere $PC+8$ sommando con l'ALU ancora il valore 4 al PC già aggiornato. $ALUResult$ è selezionato come $Result$ e inviato alla porta di ingresso $R15$ del banco di registri. Le scritture in $R15$ richiedono di scrivere nel PC invece che nel banco di registri. Il percorso dati è già in grado di fare ciò.

La figura mostra il percorso dati completo per l'istruzione `LDR`:



STR

Come LDR, STR legge un indirizzo base dalla porta 1 del banco di registri ed estende un immediato. L'ALU somma l'indirizzo base all'immediato per trovare l'indirizzo di memoria.

L'unica nuova funzione di `STR` è la necessità di leggere un secondo registro dal banco di registri e scrivere in memoria il valore letto, come mostrato nella Figura:

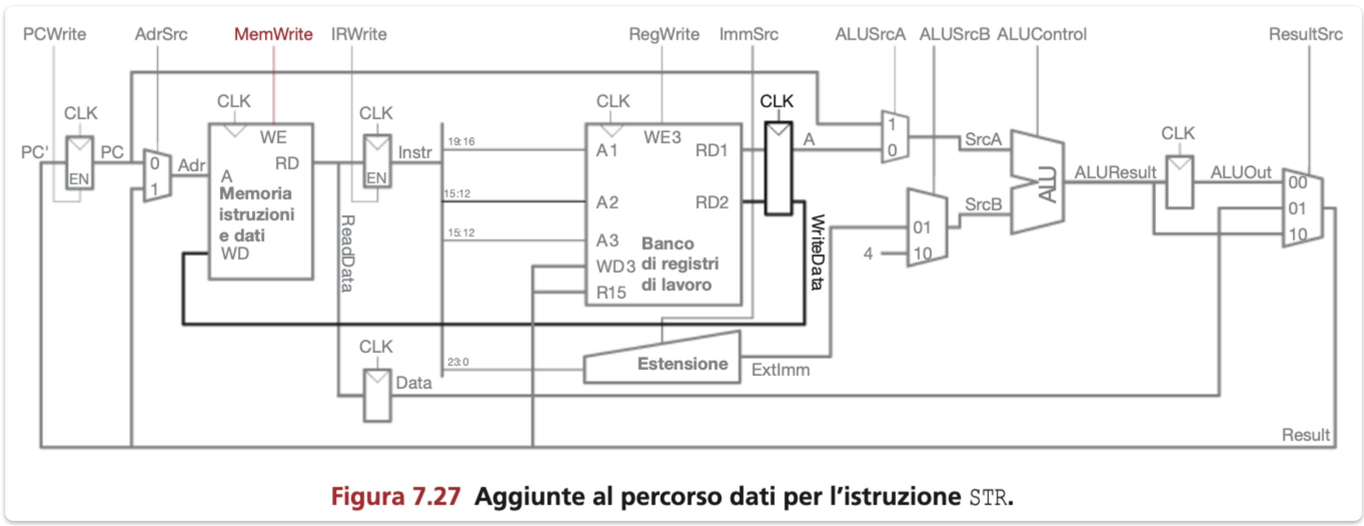


Figura 7.27 Aggiunte al percorso dati per l'istruzione STR.

Il registro è indicato dal campo Rd dell'istruzione, $Instr_{15:12}$, che viene collegato alla seconda porta del banco di registri. Quando il registro viene letto, il valore viene memorizzato in un altro registro non architetturale: $WriteData$. Al passo successivo, tale valore viene inviato alla porta di scrittura WD della memoria, la quale riceve il segnale di controllo $MemWrite$ che attiva appunto l'operazione di scrittura in memoria.

Istruzioni di elaborazione dati con indirizzamento immediato

Le istruzioni di elaborazione dati con indirizzamento immediato leggono il primo operando sorgente da Rn ed estendono il secondo operando sorgente da un immediato a 8 bit. L'ALU usa il segnale di controllo $ALUControl$ per determinare il tipo di operazione richiesta; le $ALUFlags$ sono inviate all'unità di controllo per aggiornare il registro di stato.

Istruzioni di elaborazione dati con indirizzamento a registro

Le istruzioni di elaborazione dati con indirizzamento a registro selezionano il secondo operando dal banco di registri. Il registro è specificato nel campo Rm, ovvero $Instr_{3:0}$, quindi serve un

multiplexer per selezionare questo campo come $RA2$ del banco di registri. Serve inoltre estendere il multiplexer $SrcB$ per ricevere il valore letto dal banco di registri, come mostrato nella Figura:

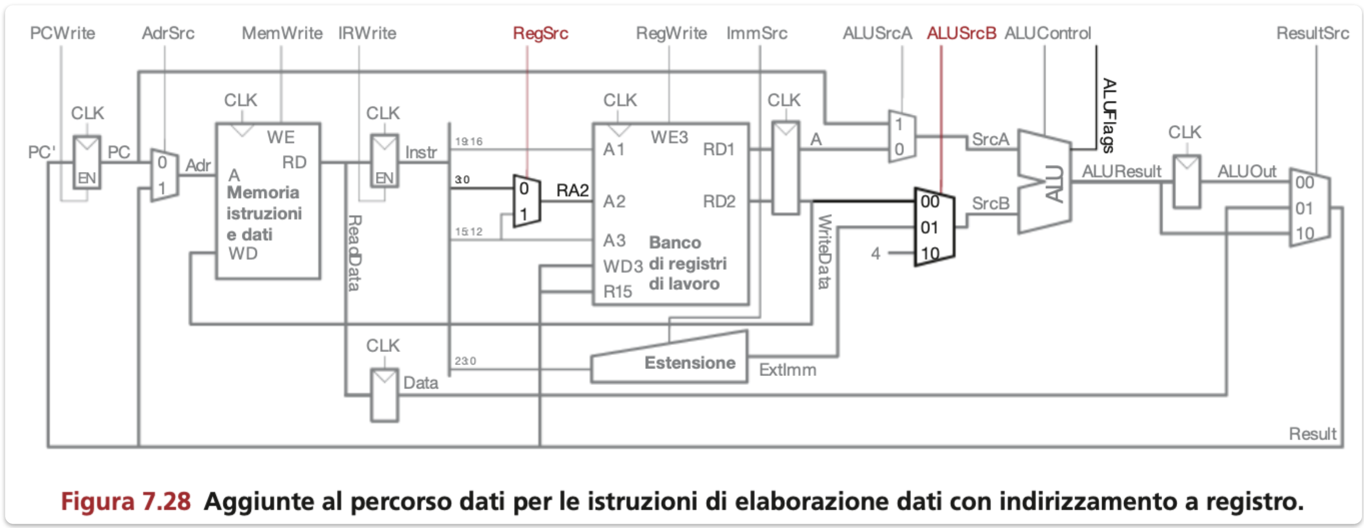


Figura 7.28 Aggiunte al percorso dati per le istruzioni di elaborazione dati con indirizzamento a registro.

B

L'istruzione di salto `B` legge $PC + 8$ e un immediato a 24 bit, li somma e somma il risultato al contenuto del PC. Una lettura di $R15$ restituisce $PC + 8$, quindi serve un ulteriore multiplexer per selezionare $R15$ come $RA1$ del banco di registri, come mostrato nella Figura:

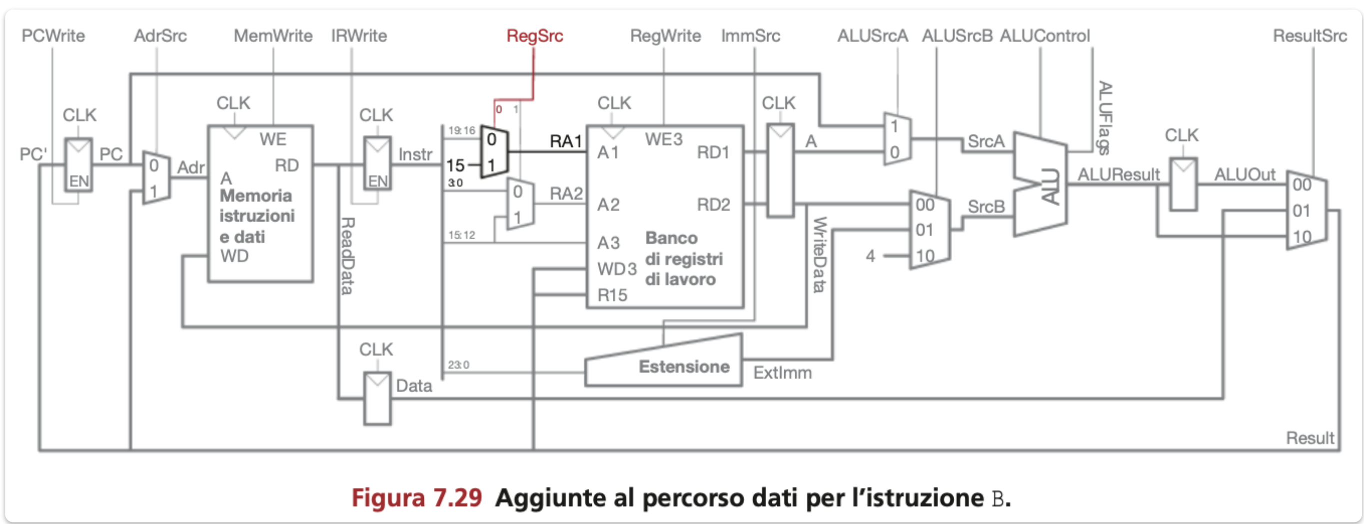


Figura 7.29 Aggiunte al percorso dati per l'istruzione B.

> **nb**
>
> Sono stati aggiunti registri non architetturali per memorizzare i risultati intermedi di ogni passo. In questo modo la memoria può essere condivisa da istruzioni e dati, e

l'ALU può essere riutilizzata diverse volte riducendo i costi hardware.

Unità di controllo multi ciclo

Come per il processore a ciclo singolo, l'unità di controllo genera i segnali di controllo sulla bse dei campi $cond, op, funct$ dell'istruzione (i bit $Instr_{31:28}$, $Instr_{27:26}$, $Instr_{25:20}$) e deve memorizzare e aggiornare opportunamento le flag di stato. LA figura mostra l'intera struttura del processore multi ciclo con l'unità di controllo integrata:

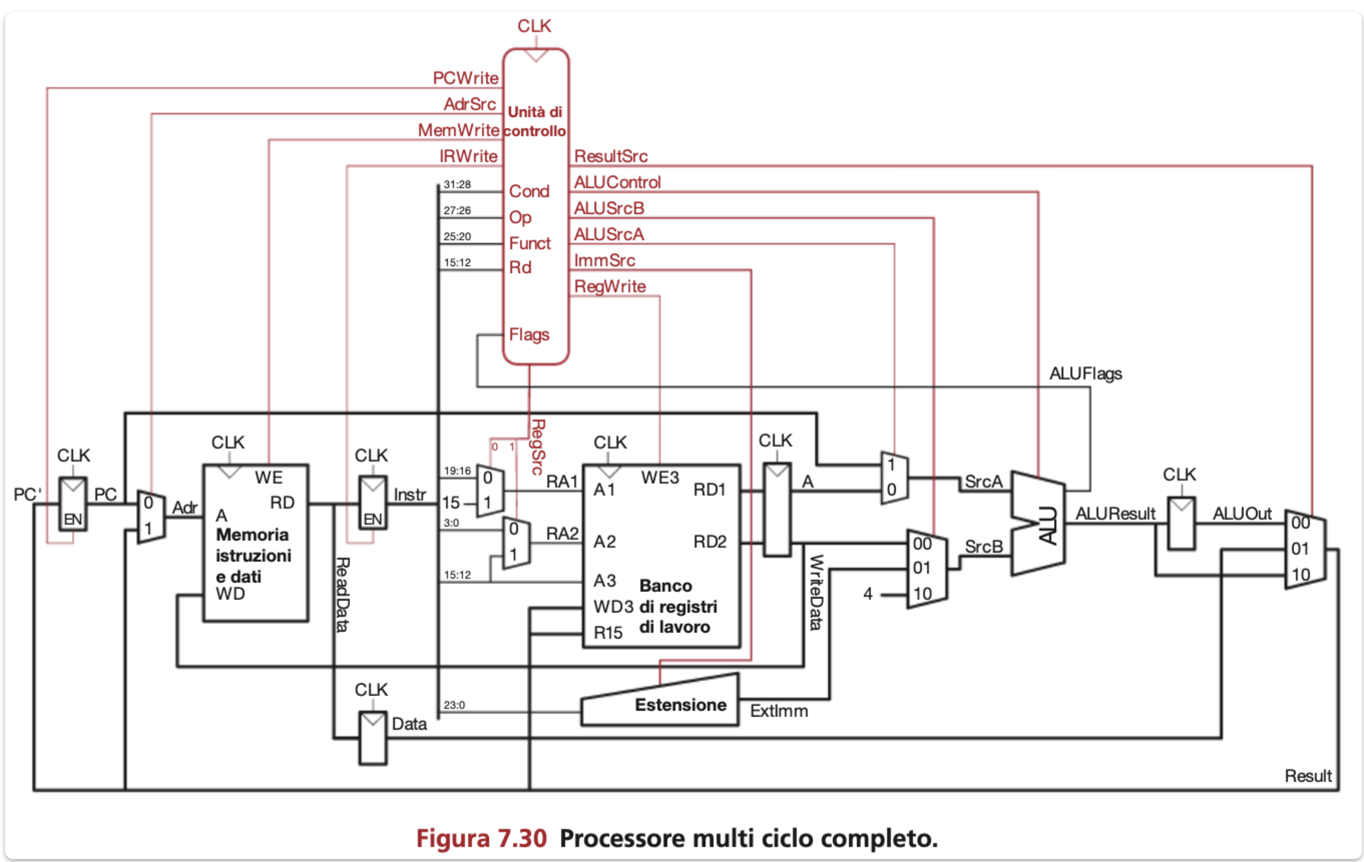


Figura 7.30 Processore multi ciclo completo.

L'*unità di controllo* è divisa in 2 blocchi:

- *Decoder*
- *Logica condizionale*
  Il decoder è suddiviso in parti a suo volta come mostrato in figura:

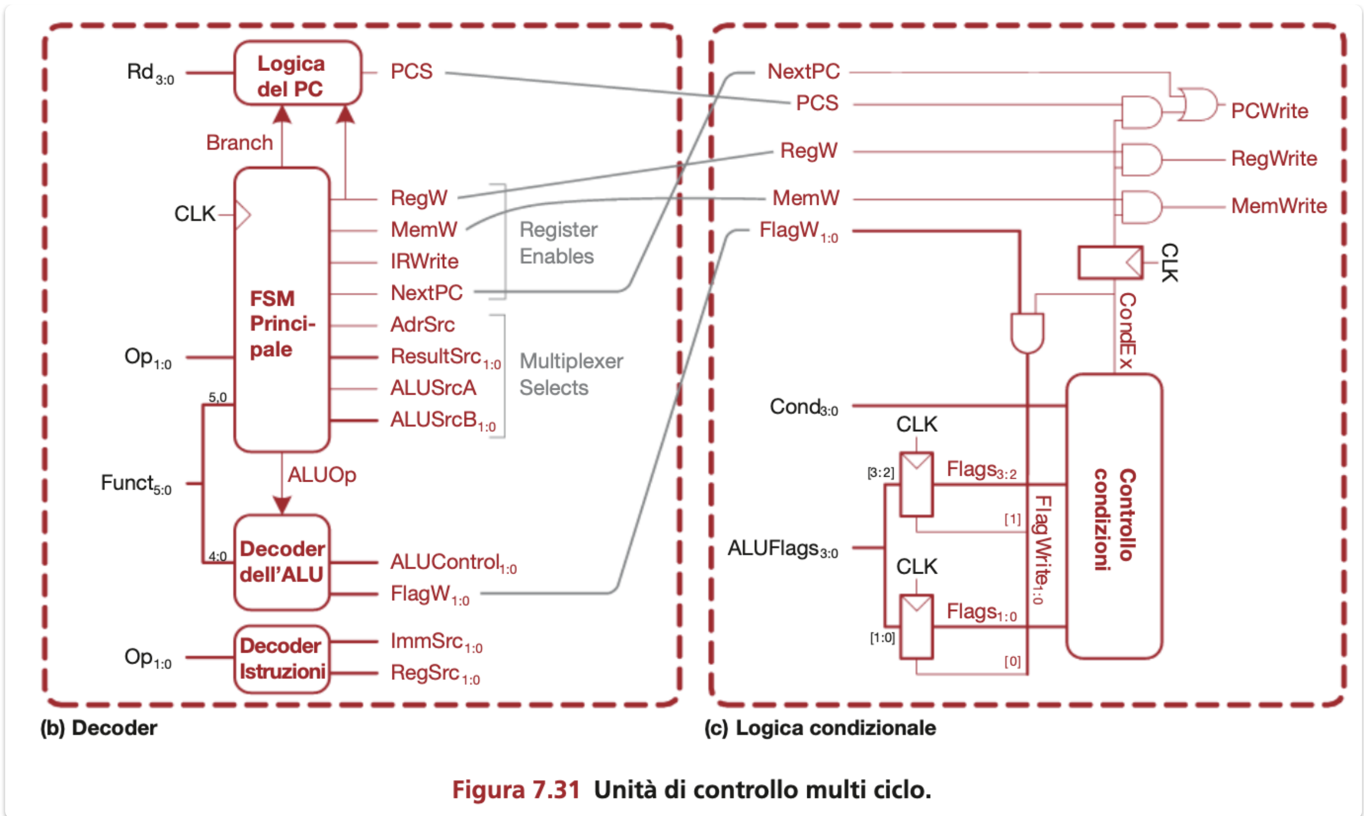


Figura 7.31 Unità di controllo multi ciclo.

Il Decoder principale viene sostituito nel processore multi ciclo da una FSM Principale che deve produrre la sequenza di segnali di controllo nei passi opportuni.
Si progetta questa FSM come macchina di Moore, in modo che le sue uscite siano funzione del solo stato presente. Si vedrà però durante il progetto della FSM che $ImmSrc$ e $RegSrc$ sono funzione di op invece che dello stato presente, quindi si userà un piccolo Decoder Istruzioni per generare tali segnali, come descritto dalla Tabella:

Tabella 7.6 Logica del *Decoder* istruzioni per *RegSrc* e *ImmSrc*.

Istruzione	*Op*	*Funct*$_5$	*Funct*$_0$	*RegSrc*$_1$	*RegSrc*$_0$	*ImmSrc*$_{1:0}$
LDR	01	X	1	X	0	01
STR	01	X	0	1	0	01
ED immediato	00	1	X	X	0	00
ED a registro	00	0	X	0	0	00
B	10	X	X	X	1	10

Il *Decoder dell'ALU* e la Logica del PC sono identici a quelli del processore a ciclo singolo. La Logica Condizionale è quasi uguale

a quella del processore a ciclo singolo: serve solo il segnale aggiuntivo $NextPC$ per forzare una scrittura nel PC quando si calcola $PC + 4$. Serve anche ritardare di un ciclo $CondEx$ prima di inviare tale segnale a $PCWrite$, $RegWrite$ e $MemWrite$ in modo che le flag di condizione aggiornate non siano visibili fino al termine dell'istruzione corrente.
La FSM Principale deve generare i segnali di selezione dei multiplexer, le abilitazioni dei registri e i segnali di scrittura in memoria del percorso dati.
I segnali di abilitazione ($RegW$, $MemW$, $IRWrite$ e $NextPC$) sono elencati solo quando devono essere attivati, cioè portati a 1: se non sono elencati si assume che valgano 0.
Il primo passo di ogni istruzione è il *fetch*: La FSM si porta in questo stato al $reset$.
Per leggere dalla memoria, $AdrSrc = 0$, in modo che l'indirizzo sia preso dal PC. $IRWrite$ è attivato per salvare l'istruzione del registro istruzioni IR. Nel mentre, il PC deve essere incrementato di 4 per puntare all'istruzione successiva. Dal momento che l'ALU non è utilizzata per altri scopi, il processore può adoperarla per calcolare $PC + 4$ in parallelo alla fase di *fetch*: $ALUSrcA = 1$, in modo che $SrcA$ provenga dal PC; $ALUSrcB = 10$, quindi $SrcB$ è la costante 4. $ALUOp = 0$, quindi l'unità di controllo genera $ALUControl = 00$ per far effettuare all'ALU la somma. Per aggiornare il PC con il valore $PC + 4$, $ResultSrc = 10$ per selezionare $ALUResult$ e $NextPC = 1$ per abilitare $PCWrite$.

Il secondo passo è la lettura del banco di registri e/o dell'immediato e la decodifica delle istruzioni. I registri e l'immediato sono selezionati da $RegSrc$ e $ImmSrc$, generati dal Decoder Istruzioni sulla base dei bit $Instr$ dell'istruzione.
La logica del Decoder Istruzioni può essere semplificata così:

$$RegSrc_1 = (Op == 01)$$
$$RegSrc_0 = (Op == 10)$$
$$ImmSrc_{1:0} = Op$$

Nel mentre, l'ALU viene usata per calcolare $PC + 8$ sommando ancora 4 al PC già incrementato nello stato *Fetch*. I segnali di controllo sono generati in modo tale da selezionare il valore PC

come primo ingresso dell'ALU ($ALUSrcA = 1$) e 4 come secondo ingresso dell'ALU ($ALUSrcB = 10$) e da attivare l'operazione di somma ($ALUOp = 0$). La somma viene selezionata come $Result$ ($ResultSrc = 10$) e inviata all'ingresso $R15$ del banco di registri in modo tale che una lettura di $R15$ restituisca $PC + 8$.
Ora la FSM procede a uno di diversi possibili stati, in base ai campi op e $funct$ dell'istruzione esaminati durante il passo $Decode$.
Se l'istruzione è un accesso a memoria (`LDR` oppure `STR`, quindi $op = 01$) il processore multi ciclo deve calcolare l'indirizzo sommando all'indirizzo base lo spiazzamento esteso con zeri.
L'indirizzo calcolato viene memorizzato nel registro $ALUOut$ per i passi successivi.

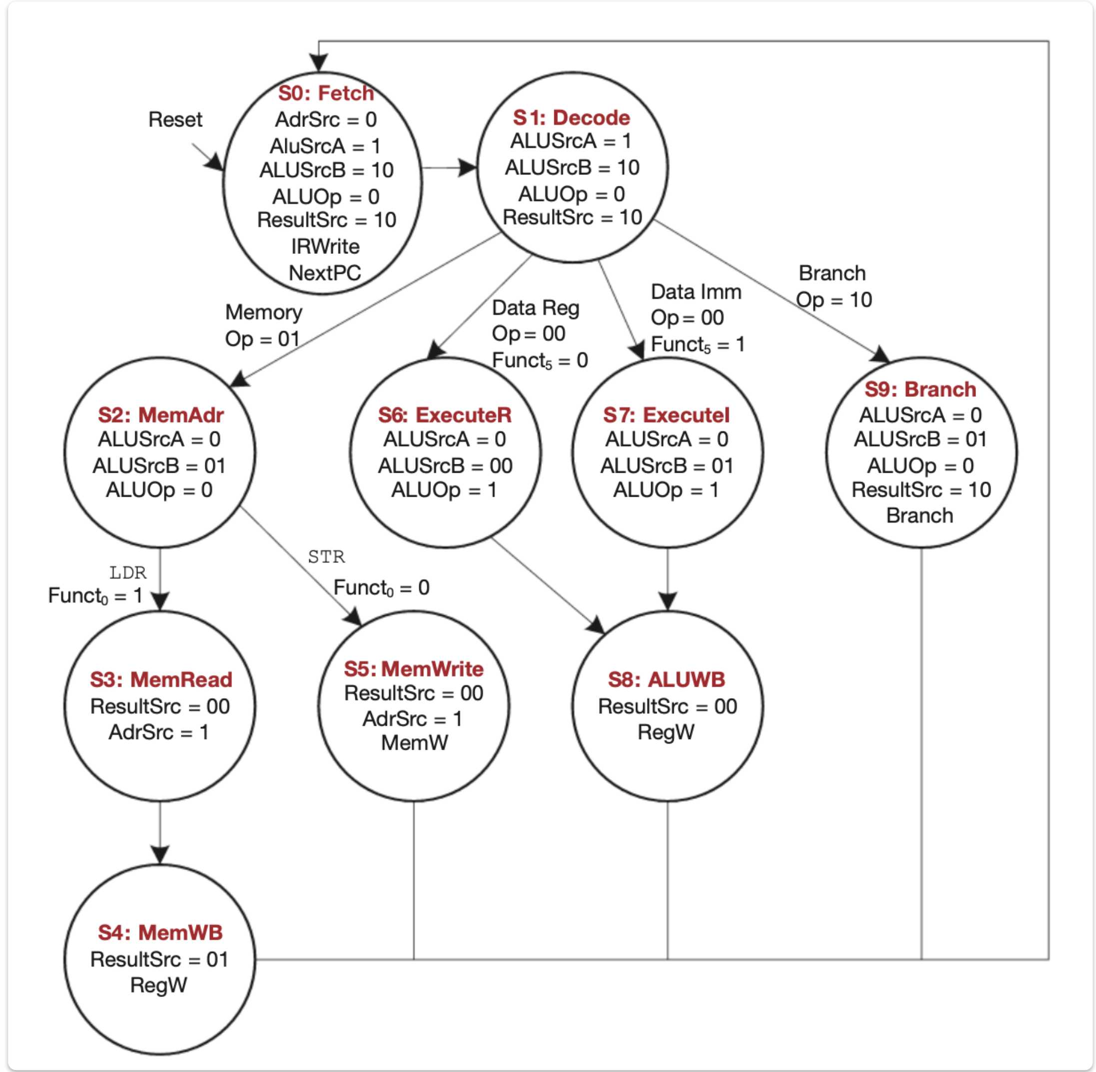
- Se l'istruzione è `LDR` ($funct_0 = 1$) il processore multi ciclo deve leggere un dato dalla memoria e scriverlo nel banco di registri:
  Il contenuto della parola indirizzata viene letto dalla memoria e salvato nel registro $Data$ durante il passo $MemRead$.
  Quindi, nel passo di scrittura finale $MemWB$ (Write Back) il contenuto di Data viene scritto nel banco di registri. Infine la FSM torna allo stato *Fetch* per iniziare l'istruzione successiva.
- Se l'isruzione è `STR` ($funct_0 = 0$) il dato letto dalla seconda porta del banco di registri deve essere semplicemente scritto in memoria.
  $MemW$ viene attivato per scrivere in memoria, e la FSM torna di nuovo nello stato *Fetch*.

Per le istruzioni di elaborazione dati ($Op = 00$) il processore multi ciclo deve calcolare il risultato usando l'ALU e scriverlo del banco di registri. Il primo operando sorgente proviene sempre da un registro, il secondo operando, invece, proviene dal banco di registri per istruzioni con modo di indirizzamento a registro ($ALUSrcB = 00$) oppure da $ExtImm$ per istruzioni con modo di indirizzamento immediato ($ALUSrcB = 01$). Quindi la FSM ha bisogno dei due stati $ExecuteR$ ed $ExecuteI$ per gestire le due diverse situazioni.
In entrambi i casi, l'istruzione di elaborazione dati avanza poi allo stato $ALUWB$ di scrittura del risultato dell'ALU, nel quale

il risultato del calcolo viene selezionato da $ALUOut$ ($ResultSrc = 00$) e scritto nel banco di registri.

Per l'istruzione di salto ($branch$), il multi ciclo deve calcolare l'indirizzo di destinazione ($PC + 8 + spiazzamento$) e scriverlo nel PC.

Mettendo tutto insieme si ottiene il diagramma degli stati completo per la FSM Principale per il processore multi ciclo riportato in figura:



Analisi delle prestazioni

Il tempo di esecuzione di un'istruzione dipende dal tempo di ciclo e dal numero di cicli necessari all'istruzione stessa. Mentre il [processore a ciclo singolo] eseguiva tutte le istruzioni in un solo ciclo, il processore multi ciclo usa numeri variabili di cicli per

le diverse istruzioni, però questo processore svolge meno attività in ogni ciclo, quindi ha senz'altro un tempo di ciclo inferiore. Il processore multi ciclo richiede tre cicli per i salti, quattro per le istruzioni di elaborazione dati e di scrittura in memoria e cinque per le letture da memoria. Il CPI dipende quindi dalla probabilità relativa di utilizzo di ciascuna tipologia di istruzioni nel programma.

7.5-Processore pipeline

Come funziona?

Si progetta qui un processore pipeline suddividendo il processore a ciclo singolo in cinque stadi di pipeline. In questo modo, cinque istruzioni alla volta possono essere in esecuzione, una per ogni stadio.
I cinque stadi sono denominati $Fetch$, $Decode$, $Execute$, $Memory$ e $Writeback$:

- *Fetch*: il processore legge l’istruzione dalla memoria;
- *Decode*: legge gli operandi dal banco di registri e decodifica l’istruzione per generare i segnali di controllo appropriati;
- *Execute*: esegue i calcoli con l’ALU;
- *Memory*: legge o scrive dati in memoria;
- *Writeback*: scrive il risultato nel banco di registri, quando previsto dall’istruzione.
  Un grosso problema presente nei sistemi pipeline è la gestione delle dipendenze, che si verificano se i risultati di un’istruzione servono a un’istruzione successiva quando ancora l’istruzione che li deve produrre non è terminata.
  In figura un esempio di un processore pipeline:

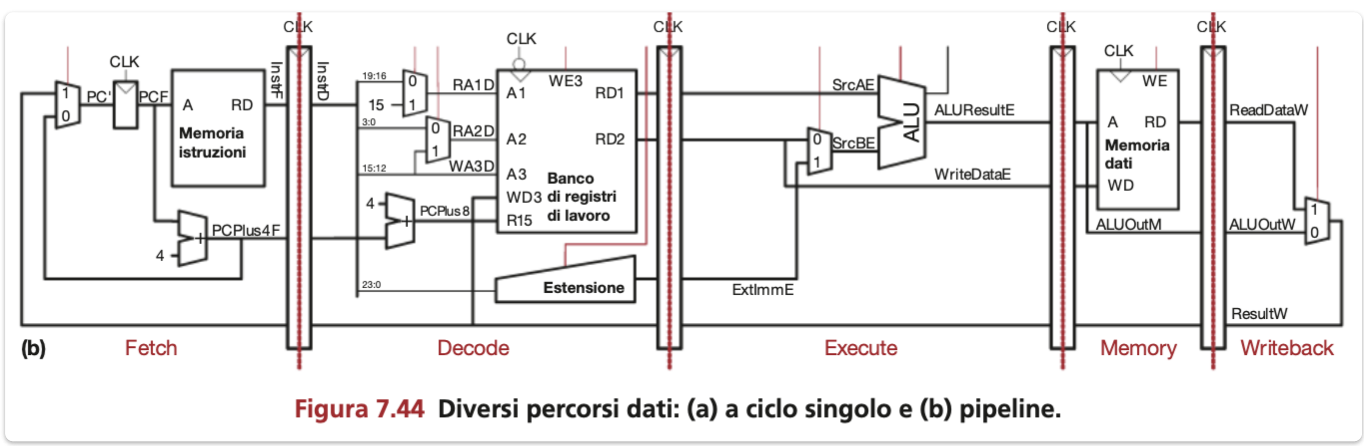


Figura 7.44 **Diversi percorsi dati: (a) a ciclo singolo e (b) pipeline.**

Unità di controllo della pipeline

Il processore pipeline usa gli stessi segnali di controllo del processore a ciclo singolo, quindi ha la [stessa unità di controllo](), che prende in considerazione i campi op e $funct$ dell'istruzione nello stadio $Decode$ per generare i segnali di controllo. Tali segnali devono essere propagati nella pipeline insieme ai dati per rimanere sincronizzati con l'istruzione cui si riferiscono. L'unità di controllo esamina anche il campo Rd per gestire le scritture nel registro $R15$ (PC).
L'intero processore pipeline con l'unità di controllo è mostrato nella Figura:

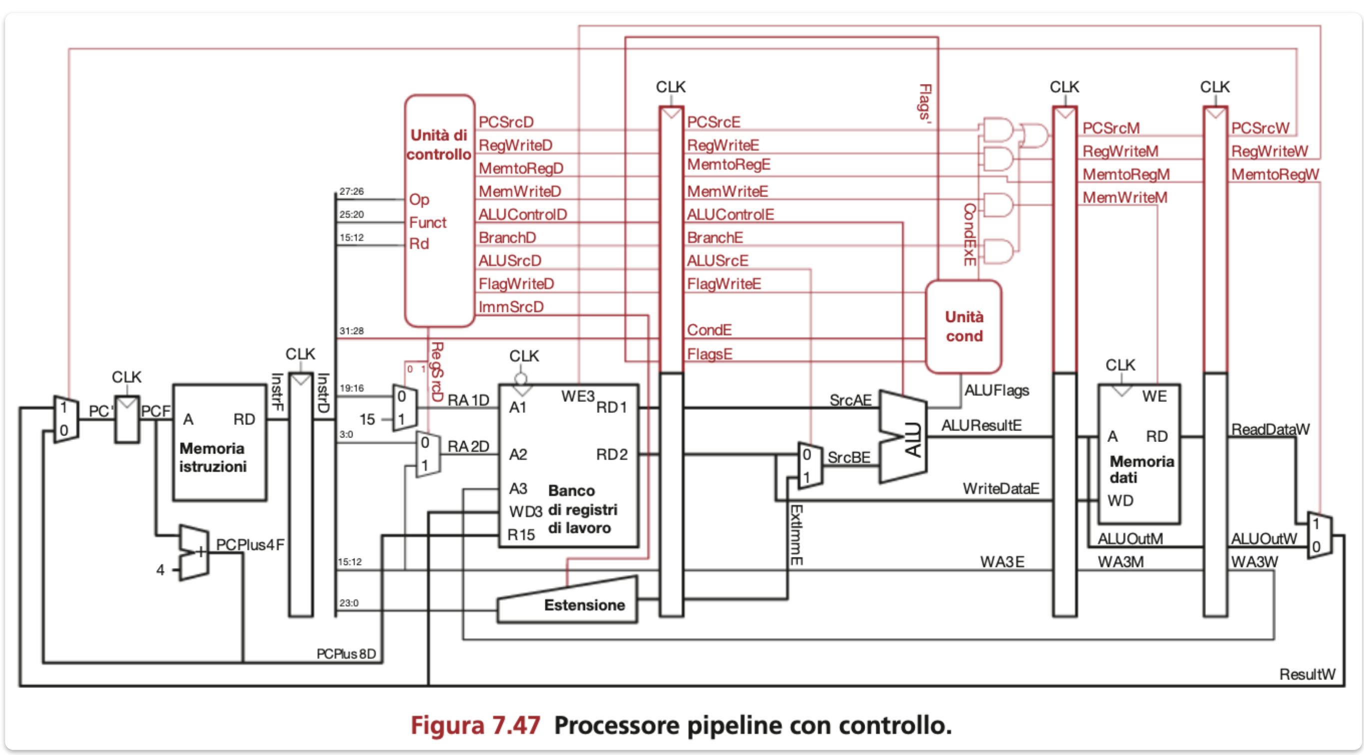


Figura 7.47 Processore pipeline con controllo.

$RegWrite$ deve essere propagato nella pipeline fino allo stadio $Writeback$ prima di essere inviato al banco di registri.

Dipendenze

In una struttura pipeline, più istruzioni sono eseguite in modo concorrente. Si verifica *dipendenza* (hazard) quando un'istruzione dipende dai risultati di un'istruzione che la precede e che non è ancora conclusa.
Il banco di registri può essere modificato e letto nel medesimo ciclo senza indurre in dipendendze.
La Figura mostra le dipendenze che si verificano quando

un'istruzione scrive in un registro ($R1$) che deve essere letto da istruzioni successive:

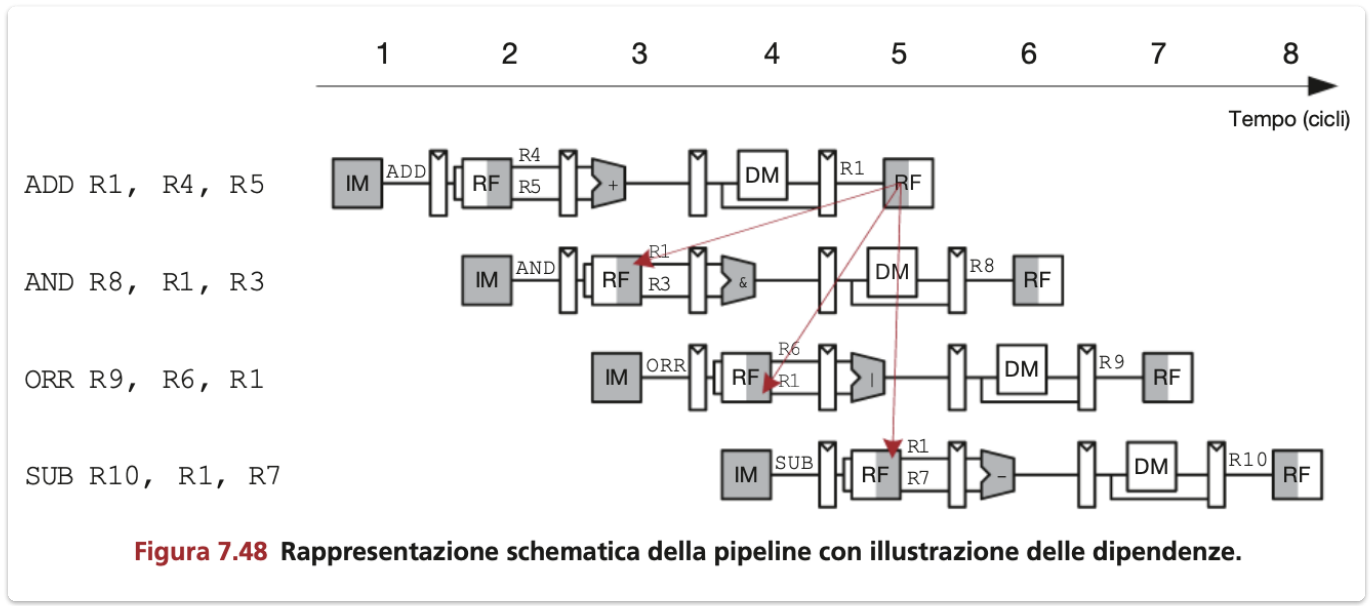


Figura 7.48 **Rappresentazione schematica della pipeline con illustrazione delle dipendenze.**

> *Questo tipo di dipendenza viene denominato RAW (Read After Write, lettura dopo la scrittura).*

Si può notare che la somma dell'istruzione `ADD` viene eseguita dall'ALU nel ciclo 3, e non serve all'istruzione `AND` finché l'ALU non userà il risultato di tale somma nel ciclo 4. In linea di principio si è quindi in grado di inoltrare il risultato di un'istruzione alla successiva per risolvere le dipendenze di tipo *RAW* senza aspettare che tale risultato venga scritto nel banco di registri.

Le dipendenze sono classificate in *dipendenze di dato* e *dipendenze di controllo*:

- Una *dipendenza di dato* si verifica quando un'istruzione vuole leggere un registro che non è ancora stato aggiornato da un'istruzione precedente
- Una *dipendenza di controllo* si verifica quando la decisione di quale istruzione debba essere prelevata da memoria nella fase di fetch non è ancora stata presa al momento del fetch.

Gestione delle dipendenze di dato tramite inoltro

Alcune dipendenze di dato possono essere risolte mediante la tecnica dell'*inoltro* (forwarding) di un risultato disponibile negli stadi *Memory* o *Writeback* all'istruzione che ha dipendenza e che si trova nello stadio *Execute*. Questo richiede l'aggiunta di multiplexer davanti all'*ALU* per selezionare l'operando dal banco di registri oppure dagli stadi *Memory* o *Writeback*. La Figura mostra questa tecnica:

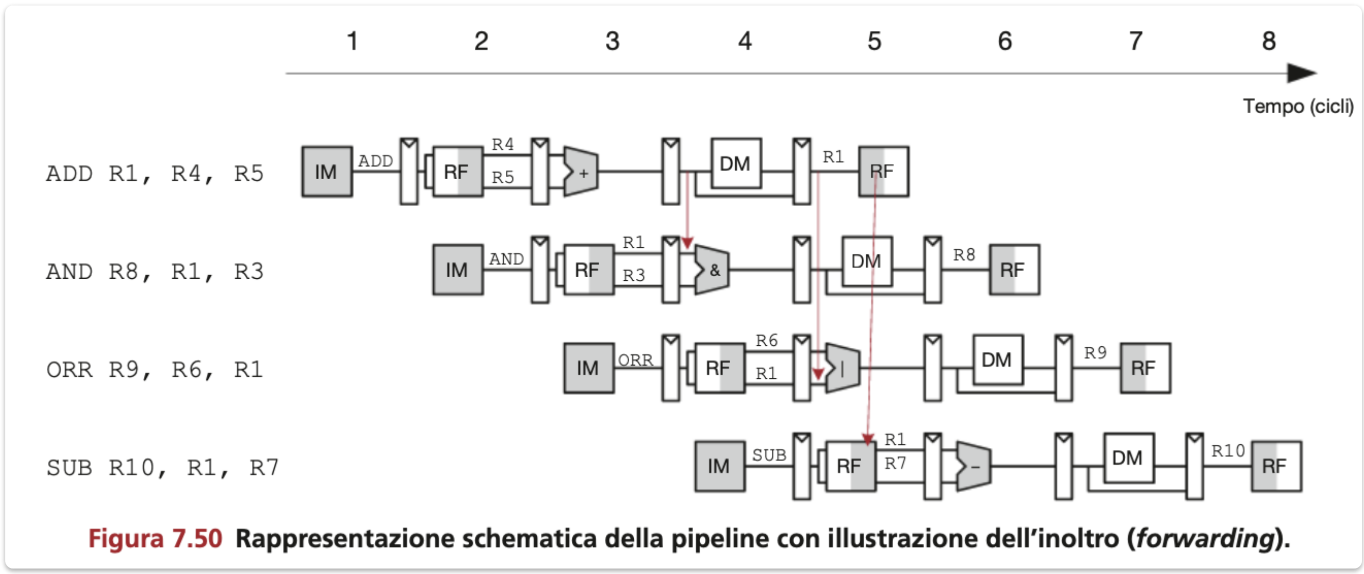


Figura 7.50 **Rappresentazione schematica della pipeline con illustrazione dell'inoltro (*forwarding*).**

L'inoltro è necessario quando un'istruzione nello stadio *Execute* ha un registro sorgente coincidente con il registro destinazione delle istruzioni nello stadio *Memory* oppure *Writeback*. Nella Figura è mostrata la modifica del processore pipeline per gestire l'inoltro:

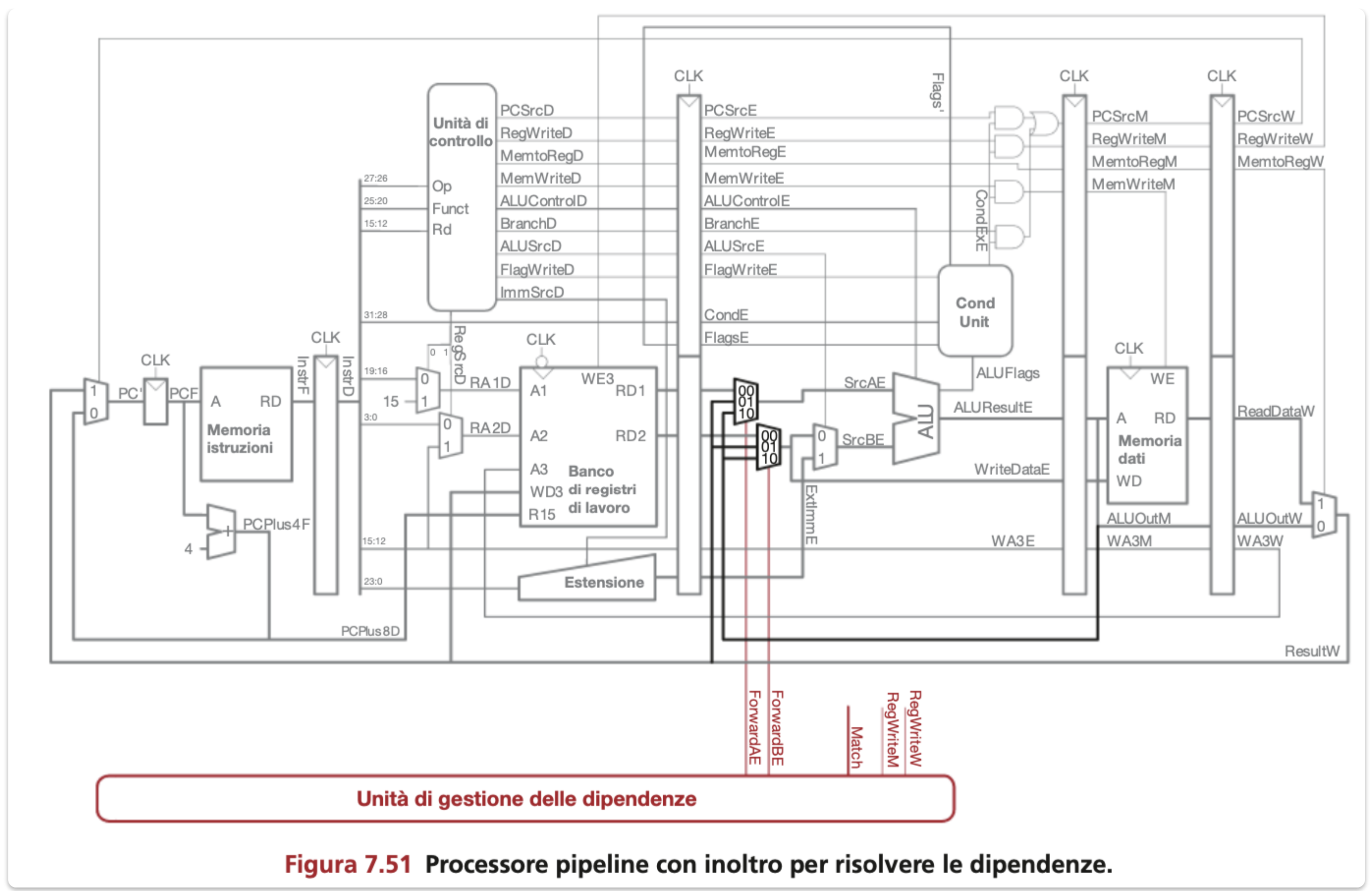


Figura 7.51 Processore pipeline con inoltro per risolvere le dipendenze.

Si aggiungono un'unità di gestione delle dipendenze e due multiplexer di inoltro. L'unità di gestione delle dipendenze riceve quattro segnali di uguaglianza dal percorso dati (abbreviati in $Match$) che indicano se un registro sorgente nello stadio Execute è uguale al registro destinazione negli stadi $Memory$ o $Writeback$:

```sv
Match_1E_M = (RA1E == WA3M)
Match_1E_W = (RA1E == WA3W)
Match_2E_M = (RA2E == WA3M)
Match_2E_W = (RA2E == WA3W)
```

L'unità di gestione delle dipendenze riceve anche i segnali $RegWrite$ dagli stadi $Memory$ e $Writeback$ per sapere se il registro destinazione verrà davvero modificato.
I collegamenti sono indicati da un pezzetto di filo etichettato con il nome del segnale al quale è connesso.

L'unità di gestione delle dipendenze genera i segnali per i multiplexer di inoltro per selezionare gli operandi dal banco dei registri oppure dai risultati negli stadi $Memory$ o $Writeback$ (

ALUOutM o *ResultW*). L'inoltro deve essere attivato da uno dei due stadi se quello stadio scrive in un registro che è sorgente nello stadio *Execute*. Se entrambi gli stadi *Memory* e *Writeback* contengono registri destinazione uguali allo stesso registro sorgente di *Execute*, si deve dare la precedenza allo stadio *Memory* che contiene il valore più recente.
Quindi, la funzione di inoltro per *SrcAE* è quella riportata sotto. La funzione di inoltro per *SrcBE* (*ForwardBE*) è identica, ad eccezione dell'utilizzo del segnale di controllo $Match_2E$.

```language-sv
if              (Match_1E_M • RegWriteM) ForwardAE = 10;
                //SrcAE = ALUOutM

else if         (Match_1E_W • RegWriteW) ForwardAE = 01;
                // SrcAE = ALUOutW

else            ForwardAE = 00;
                // SrcAE dal banco di registri
```

Gestione delle dipendenze di dato tramite stalli

L'inoltro è sufficiente a risolvere le dipendenze di tipo *RAW* .
La figura mostra questo problema in caso di un'istruzione `LDR`:

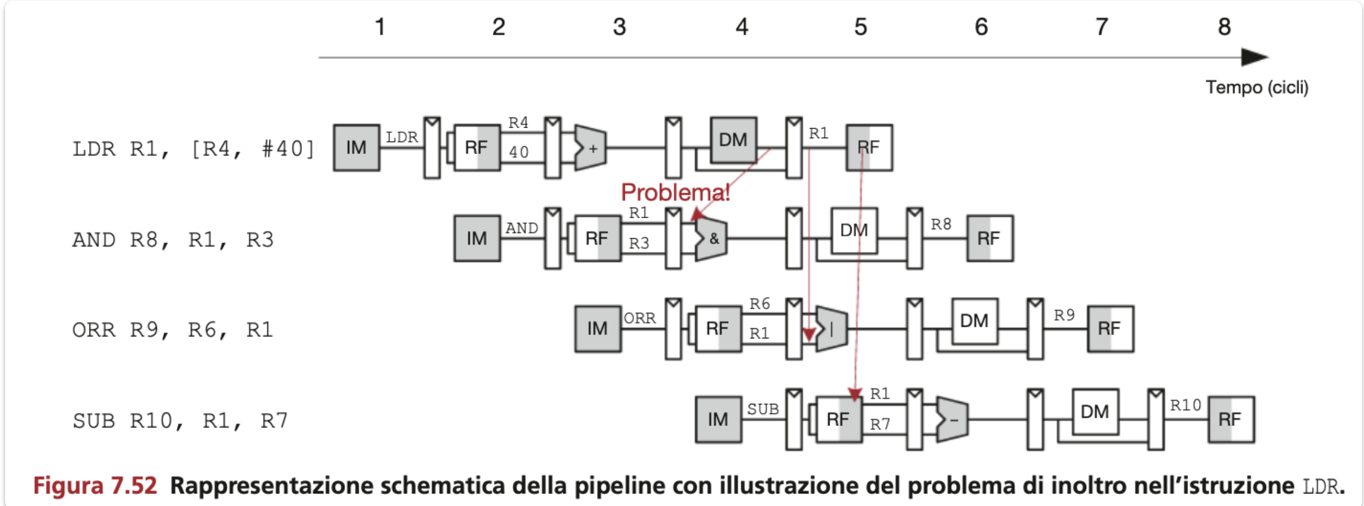


Figura 7.52 **Rappresentazione schematica della pipeline con illustrazione del problema di inoltro nell'istruzione** `LDR`.

l'istruzione `LDR` riceve il dato da memoria alla fine del ciclo 4, ma l'istruzione `AND` ha bisogno di avere tale dato come operando all'inizio del ciclo 4, quindi non c'è modo di risolvere questa dipendenza tramite inoltro.

La soluzione alternativa è quella di forzare in *stallo* la pipeline, sospendendo le operazioni fino all'arrivo del dato. La Figura mostra lo stallo dell'istruzione dipendente (AND) nello stadio Decode:

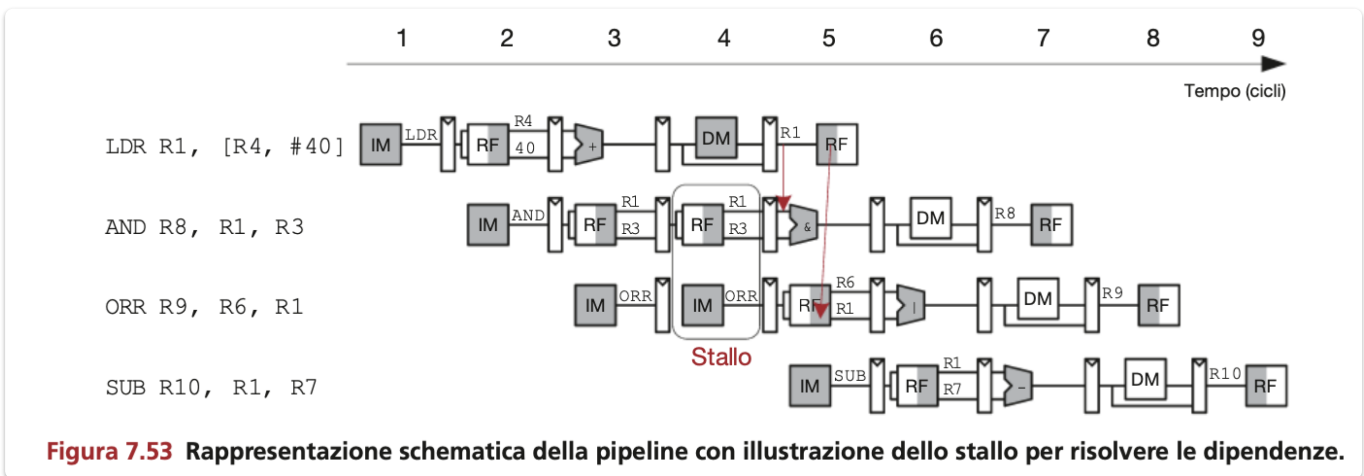


Figura 7.53 **Rappresentazione schematica della pipeline con illustrazione dello stallo per risolvere le dipendenze.**

`AND` entra nello stadio *Decode* nel ciclo 3 e viene mantenuta in stallo nel ciclo 4. L'istruzione successiva (`ORR`) deve a sua volta rimanere per due cicli nello stadio *Fetch* perché lo stadio *Decode* è occupato.

Nel ciclo 5 il risultato può essere inoltrato dallo stadio *Writeback* di `LDR` allo stadio *Execute* di `AND`. Sempre nel ciclo 5, il registro sorgente $R1$ dell'istruzione `ORR` può essere letto direttamente dal banco di registri senza bisogno di inoltro.

Si noti che lo stadio *Execute* non è usato nel ciclo 4, così come non lo sono lo stadio *Memory* nel ciclo 5 e lo stadio *Writeback* nel ciclo 6. Questo mancato utilizzo di uno stadio che si propaga lungo la pipeline è denominato bolla (bubble) e si comporta come un'istruzione `NOP` (NoOperation). La bolla viene generata azzerando i segnali di controllo dello stadio *Execute* durante lo stallo dello stadio *Decode*, in modo tale che non vengano eseguite dalla bolla azioni che modifichino lo stato architetturale del processore.

La Figura modifica il processore pipeline per aggiungere gli stalli per le dipendenze di dato dall'istruzione LDR:

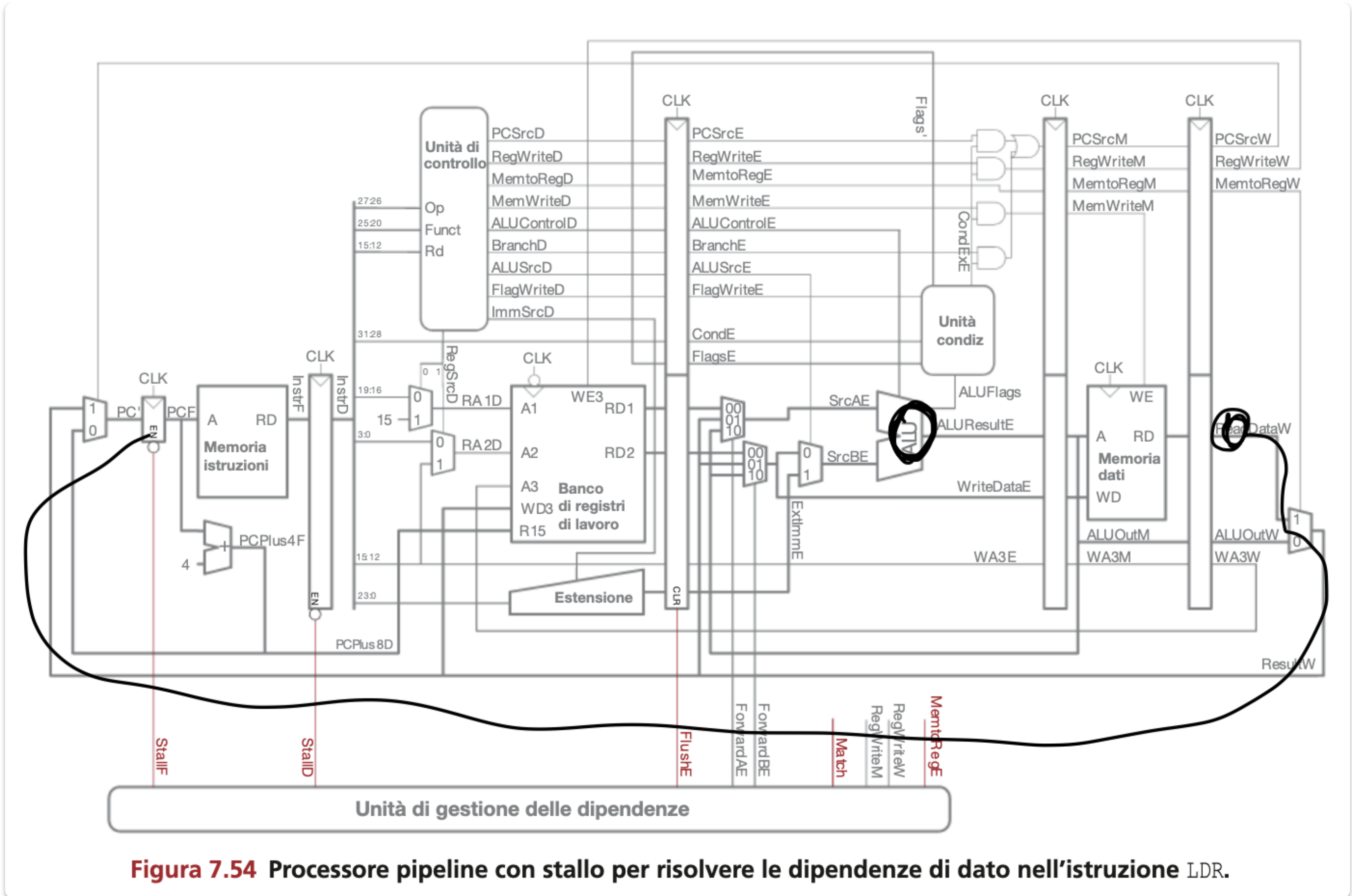


Figura 7.54 **Processore pipeline con stallo per risolvere le dipendenze di dato nell'istruzione** `LDR`.

L'unità di gestione delle dipendenze esamina l'istruzione presente nello stadio *Execute*: se è un'istruzione `LDR` e il suo registro destinazione ($WA3E$) coincide con uno dei due registri sorgente nello stadio *Decode* ($RA1D$ oppure $RA2D$), allora si deve mettere in stallo l'istruzione nello stadio *Decode* finché il dato sorgente non sia stato reso disponibile.

Gli stalli sono realizzati aggiungendo:

- Ingressi di abilitazione ai registri di pipeline degli stadi *Fetch* e *Decode*
- Un *ingresso di reset sincrono* (CLR) al registro di pipeline dello stadio Execute.

La logica per generare stalli e svuotamenti risulta essere:

```sv
Match_12D_E = (RA1D == WA3E) + (RA2D == WA3E) LDRstall = Match_12D_E • MemtoRegE
StallF = StallD = FlushE = LDRstall
```

Gestione delle dipendenze di controllo

L'istruzione `B` presenta una dipendenza di controllo: il processore pipeline non sa di quale istruzione fare il $fetch$ come istruzione successiva, perché la decisione circa il fatto di saltare o meno non è ancora stata presa al momento di tale $fetch$. Le scritture nel registro $R15$ (cioè nel PC) presentano lo stesso tipo di dipendenza.

La soluzione è prevedere se il salto dovrà essere fatto oppure no, e cominciare l'esecuzione delle istruzioni sulla base di tale previsione: una volta presa la decisione circa il salto, il processore dovrà eliminare tali istruzioni se la previsione si fosse rivelata sbagliata. Nella pipeline vista sino a qui il processore prevede che i salti non vengano effettuati ma se il salto andava fatto, le quattro istruzioni successive al salto devono essere scartate, cioè la pipeline deve essere svuotata cancellando i registri di pipeline per tali istruzioni.

Si può ridurre la penalizzazione per salto mal previsto se si riesce ad anticipare la decisione se saltare o meno. In effetti tale decisione viene presa nello stadio $Execute$ quando la destinazione di salto è già stata calcolata e si conosce il valore di $CondEx$.
La Figura modifica il processore pipeline per anticipare la decisione di salto e gestire le dipendenze di controllo:

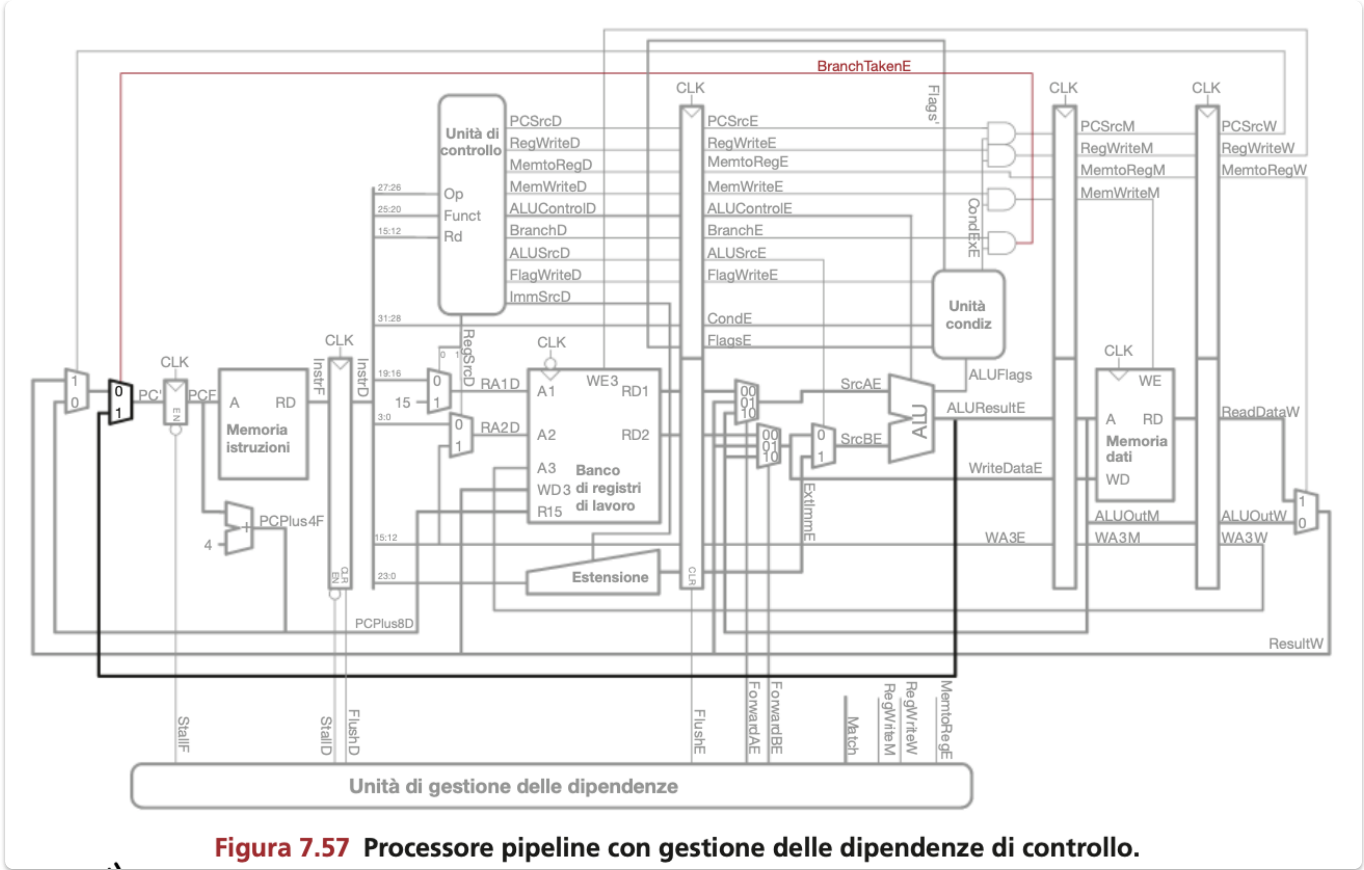


Figura 7.57 **Processore pipeline con gestione delle dipendenze di controllo.**

Si aggiunge un multiplexer di salto prima del registro PC per selezionare la destinazione di salto da $ALUResultE$, e il segnale $BranchTakenE$ che controlla tale multiplexer è attivato per le istruzioni di salto le cui condizioni sono soddisfatte. $PCSrcW$ viene attivato solo per scritture nel PC, che ancora avvengono nello stadio $Writeback$.

Infine, serve generare i segnali di controllo per stallo e svuotamento per gestire salti e scritture nel PC, che rischiano di generare errori. Quando nella pipeline è presente una scrittura nel PC bisogna mettere in stallo la pipeline fino al termine della scrittura: questo è ottenuto mettendo in stallo lo stadio di $Fetch$. La logica per gestire queste situazioni è riportata di seguito:

```sv
PCWrPendingF = PCSrcD + PCSrcE + PCSrcM; StallD = LDRstall;
StallF = LDRstall + PCWrPendingF;
FlushE = LDRstall + BranchTakenE;
FlushD = PCWrPendingF + PCSrcW + BranchTakenE;
```

7.7-Microarchitetture avanzate

Prestazioni

Il tempo richiesto per eseguire un programma è proporzionale al periodo del clock e al numero di cicli di clock per istruzione (CPI): per migliorare le prestazioni serve quindi velocizzare il clock e/o diminuire il CPI.

Micro operazioni

Gli architetti di calcolatori rendono veloci i casi frequenti definendo un insieme di semplici micro operazioni (spesso citate in inglese come $micro-ops$ o ops) che possono essere eseguite su percorsi dati semplici. Le istruzioni vere e proprie vengono scomposte in una o più micro operazioni.
Anche se la maggior parte delle istruzioni ARM è costituita da istruzioni semplici, alcune possono comunque essere scomposte in micro operazioni. Per esempio:

```sv
//OPERAZIONE COMPLESSA

LDR R1, [R2], #4

ORR R3, R4, R5, LSL R6
```

```sv
//SEQUENZA DI MICRO OPERAZIONI

LDR R1, [R2]
ADD R2, R2, #4
LSL T1, R5, R6
ORR R3, R4, T1
```

Previsione dei salti

Un processore pipeline dovrebbe avere idealmente un CPI pari a 1. La causa principale di aumento del CPI è la penalizzazione per salti mal previsti.

Per affrontare questo problema, la maggior parte dei processori pipeline adotta un *predittore di salto* (branch predictor) per cercare di prevedere se il salto andrà eseguito o meno.

Alcuni salti sono posizionati alla fine di un ciclo, e saltano indietro all'inizio del ciclo per ripeterlo (per es. nei cicli `for` e `while`). I cicli sono di solito eseguiti molte volte, quindi i salti all'indietro molto spesso sono da fare. Una forma molto semplice di previsione dei salti è quindi quella che verifica la direzione del salto e assume che i salti all'indietro debbano essere fatti. Viene chiamata *previsione statica dei salti*.
I salti in avanti sono più difficile da prevedere, quindi molti processori usano una previsione dinamica dei salti che si basa sulla storia del programma per cercare di indovinare se il salto vada o meno eseguito. I predittori dinamici memorizzano una tabella denominata *buffer delle destinazioni di salto* (branch target buffer) o anche *tabella di previsione dei salti* (branch prediction table), include la destinazione di ciascun salto e la storia del salto, ovvero se sia stato o meno eseguito in passato

> *Un predittore dinamico a un bit ricorda se il salto è stato eseguito oppure no l'ultima volta che è stato incontrato, quindi sbaglia la previsione nella prima e nell'ultima iterazione di un ciclo.*

> *Un predittore dinamico dei salti a due bit risolve parzialmente il problema memorizzando quattro stati relativi a ogni salto, denominati* $stronglytaken$, $weaklytaken$, $weaklynottaken$, $stronglynottaken$ *e sbaglia solo la previsione relativa all'ultima iterazione del ciclo.*

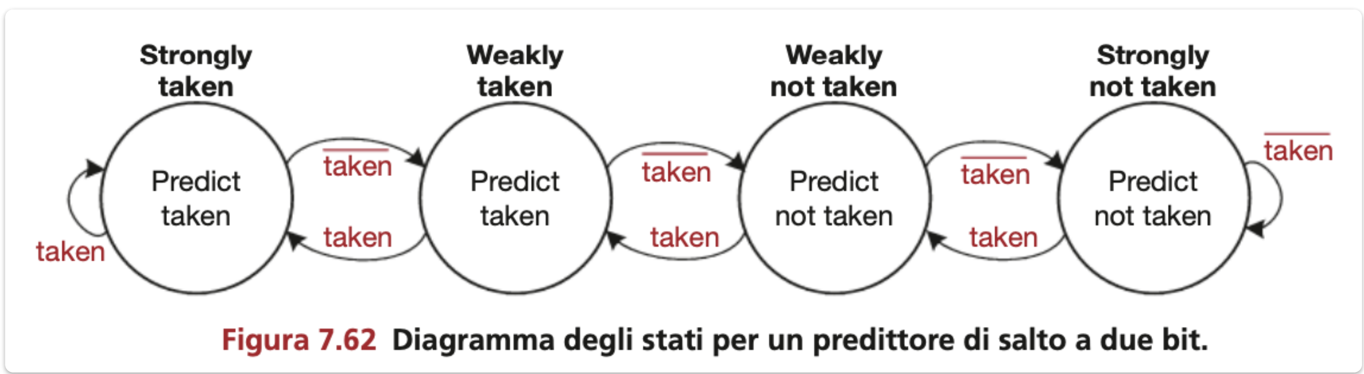


Figura 7.62 **Diagramma degli stati per un predittore di salto a due bit.**

Processori superscalari

Un processore superscalare contiene più copie dell'hardware del percorso dati, per poter eseguire più istruzioni contemporaneamente. La Figura mostra lo schema a blocchi di un processore superscalare a due vie, che attiva due istruzioni in ogni ciclo.

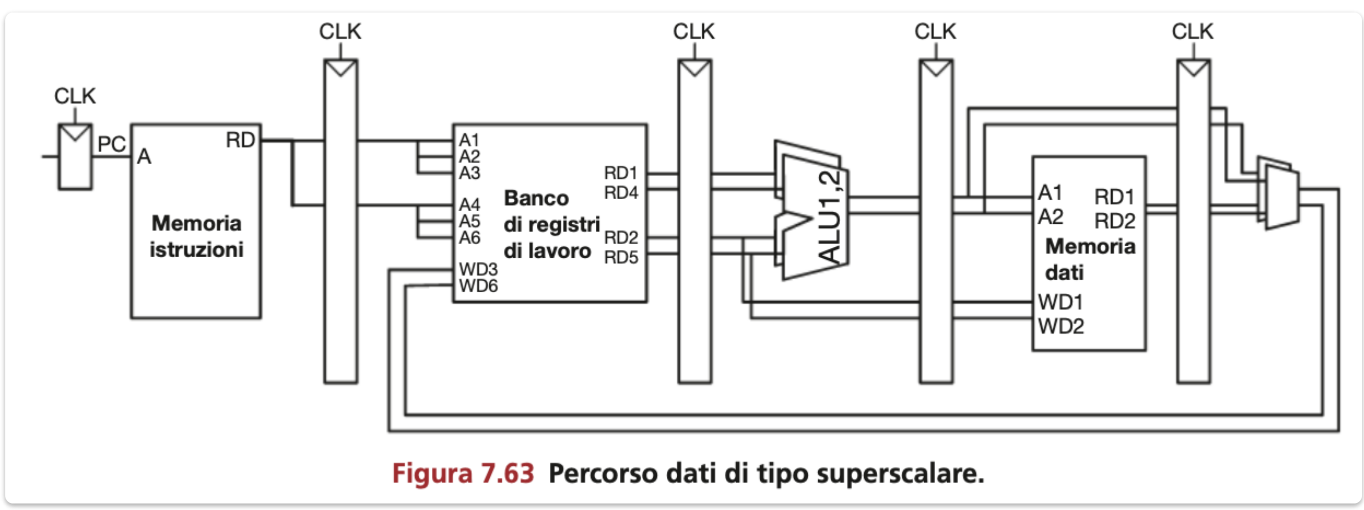


Figura 7.63 Percorso dati di tipo superscalare.

> *L'esecuzione simultanea di più istruzioni crea problemi per le dipendenze.*

Quando si parla di parallelismo di esecuzione si fa riferimento a due forme di parallelismo: temporale e spaziale. La pipeline è un caso di parallelismo temporale, mentre unità di esecuzione multiple sono un caso di parallelismo spaziale.
I processori superscalari sfruttano entrambe le forme di parallelismo per "spremere" dalla microarchitettura tutte le prestazioni possibili ben oltre i limiti dei processori a ciclo singolo o multi ciclo.

7.9-Riassunto

In questo capitolo si sono confrontate le microarchitetture a ciclo singolo, multi ciclo e pipeline per il processore ARM: tutte le tre microarchitetture realizzano lo stesso sottoinsieme del set di istruzioni ARM e hanno il medesimo stato architetturale. La più semplice è quella del processore a ciclo singolo con un CPI pari a 1.

Il processore multi ciclo usa un numero variabile di passi più brevi per eseguire le istruzioni, quindi può riutilizzare l'ALU invece di richiedere la presenza di vari sommatori. Ha però bisogno di diversi registri non architetturali per memorizzare i risultati intermedi tra un passo e l'altro. In linea di principio potrebbe essere più veloce del precedente, perché non tutte le istruzioni hanno la stessa durata, ma in pratica si rivela spesso più lento perché il tempo di ciclo è vincolato dal passo più lungo e per il sovraccarico di sequenziamento in ogni passo.

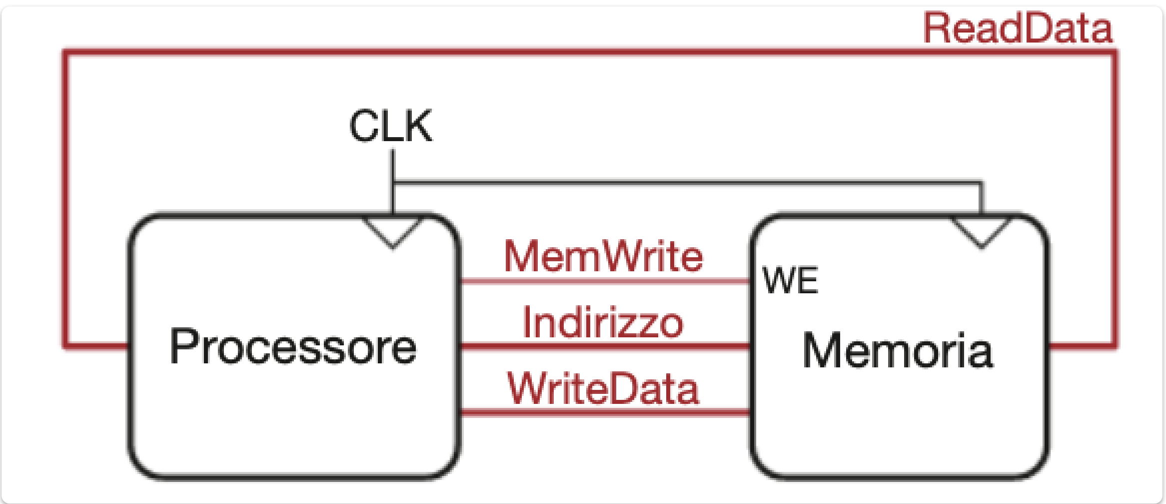
Il processore pipeline divide il processore a ciclo singolo in cinque stadi di pipeline relativamente veloci, e aggiunge registri di pipeline tra gli stadi per separare le cinque istruzioni simultaneamente in esecuzione. Teoricamente ha un CPI pari a 1, ma le dipendenze costringono a stalli e svuotamenti che aumentano un po' il CPI. La gestione delle dipendenze costa anche in termini di hardware aggiuntivo e di complessità progettuale. Il periodo di clock potrebbe in teoria essere un quinto di quello del processore a ciclo singolo, ma in pratica è vincolato dallo stadio più lento e dal sovraccarico di sequenziamento di ogni stadio. In ogni caso la struttura pipeline assicura un sostanziale incremento di prestazioni, tanto che tutti i moderni microprocessori ad alte prestazioni la adottano.

Il limite principale di questo capitolo è l'ipotesi di avere una memoria veloce e grande abbastanza per contenere l'intero programma e i suoi dati: in realtà, memorie veloci e grandi hanno costi proibitivi. Quindi nel prossimo capitolo si illustra come

poter disporre di quasi tutte le caratteristiche di una memoria veloce e grande usando una piccola memoria veloce che contiene le informazioni usate più di frequente e memorie molto più grandi ma più lente che contengono il resto delle informazioni.

8.1-Introduzione

Le prestazioni dei calcolatori dipendono dal sistema di memoria tanto quanto dalla microarchitettura del processore.
Il processore comunica con la memoria tramite un'interfaccia a memoria. La Figura mostra la semplice interfaccia a memoria utilizzata dal processore ARM multi ciclo:



Il processore invia un indirizzo al sistema di memoria tramite il *bus indirizzi*. In caso di lettura, il segnale $MemWrite$ (scrittura in memoria) vale 0 e la memoria restituisce il dato sul *bus di lettura dati*. In caso di scrittura, il segnale $MemWrite$ vale 1 e il processore invia il dato alla memoria sul *bus di scrittura dati*.

I calcolatori memorizzano istruzioni e dati usati più di frequente in una memoria più veloce ma più piccola, denominata *cache* e costituita generalmente da $SRAM$ situata a bordo dello stesso chip del processore.
Le cache possono memorizzare sia istruzioni sia dati, anche se spesso si parla genericamente di "dati".
Se il processore richiede un dato che è presente nella cache, tale dato viene reso disponibile rapidamente: questo evento di "dato trovato" viene denominato in inglese *hit* ("colpito"). In caso contrario, il processore recupera il dato dalla memoria principale

(DRAM); l’evento di “dato non trovato” viene denominato in inglese *miss* (“mancato”).

8.3-Memoria cache

La memoria cache contiene i dati usati più di frequente. Il numero di parole che la cache può contenere è definito capacità C della cache.
Quando il processore deve accedere a un dato, per prima cosa verifica se tale dato è presente in cache. In caso di hit, il dato è immediatamente disponibile. In caso di $miss$, il processore preleva il dato da memoria principale e lo copia in cache per futuro utilizzo. Naturalmente, per fare spazio al nuovo dato, la cache deve *sostituire* (replace) un dato vecchio.
Le cache sono dimensionate in termini di *capacità* (C), *numero di set* (S), *dimensione del blocco* (b), *numero di blocchi* (B) e *grado di associatività* (N).

Quali dati devono essere memorizzati nelle cache?

Una cache ideale dovrebbe prevedere in anticipo tutti i dati necessari al processore e prelevarli dalla memoria principale con anticipo sufficiente ad avere un tasso di $miss$ pari a zero. Dal momento che è ovviamente impossibile predire il futuro con completa accuratezza, la cache deve indovinare quali dati saranno necessari in base alle sequenze di accessi a memoria verificatesi nel passato.

- *Località temporale*: il processore ha un'elevata probabilità di accedere nuovamente nel prossimo futuro a un dato se lo ha utilizzato da poco
- *Località spaziale*: quando il processore accede a un certo dato, ha un'elevata probabilità di accedere nel prossimo futuro ad altri dati in locazioni di memoria vicine al dato in questione.

Quindi quando la cache preleva una parola da memoria principale, preleva anche alcune altre parole adiacenti: questo gruppo di parole è denominato *blocco di cache* o *linea di cache*. Il numero b

di parole in un blocco di cache è definito *dimensione di blocco*.
Una cache di capacità C contiene quindi $B = C/b$ blocchi.

Come si verifica se un dato è in cache?

Ogni cache è organizzata in S insiemi o *set*, ciascuno dei quali contiene uno o più blocchi di dati. La relazione tra l'indirizzo di un dato in memoria principale e la locazione di tale dato in cache è definita *mappatura* (mapping).

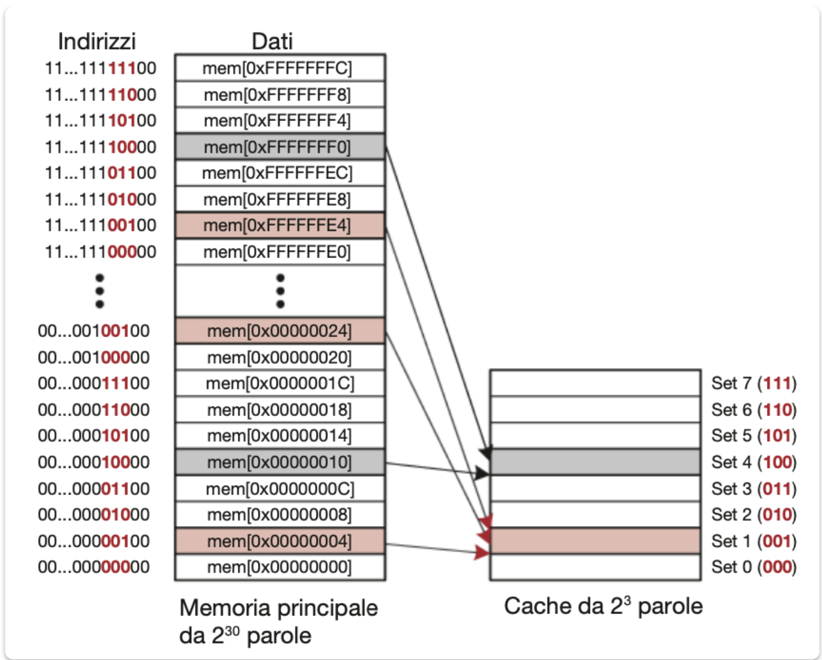
Le cache sono categorizzate in base al numero di blocchi presenti in un set.

- In una *cache a mappatura diretta* (direct mapped) ogni set contiene un solo blocco, quindi la cache ha $S = B$ set.
- In una *cache parzialmente associativa* (set associative) a N vie ogni set contiene N blocchi. L'indirizzo di memoria principale è mappato in un solo set, con $S = B/N$ set.
- Una *cache completamente associativa* (fully associative) ha solo $S = 1$ set. Un dato può andare in uno qualsiasi dei B blocchi del set, quindi una cache completamente associativa può essere definita come una cache parzialmente associativa a B vie.

Cache a mappatura diretta

Una cache a mappatura diretta ha un solo blocco in ogni set, quindi è organizzata in $S = B$ set.
Un indirizzo nel blocco 0 della memoria principale viene mappato nel set 0 della cache, un indirizzo nel blocco 1 della memoria principale viene mappato nel set 1 della cache, e così via fino a un indirizzo nel blocco B - 1 della memoria principale che viene mappato nel set B - 1 della cache. Non ci sono altri blocchi nella cache, quindi la mappatura si ripete circolarmente con il blocco B della memoria principale che viene mappato nel set 0 della cache, e così via.

Questa mappatura è illustrata nella Figura per una cache a mappatura diretta con capacità di otto parole e dimensione di blocco di una parola.



Dal momento che molti indirizzi sono mappati nel medesimo set, la cache deve tenere traccia dell'indirizzo del dato effettivamente presente in ogni set. I bit meno significativi dell'indirizzo indicano in quale set è mappato il dato, i restanti bit più significativi sono denominati *tag* (etichetta) e indicano quale dei tanti possibili indirizzi è effettivamente presente in quel particolare set.

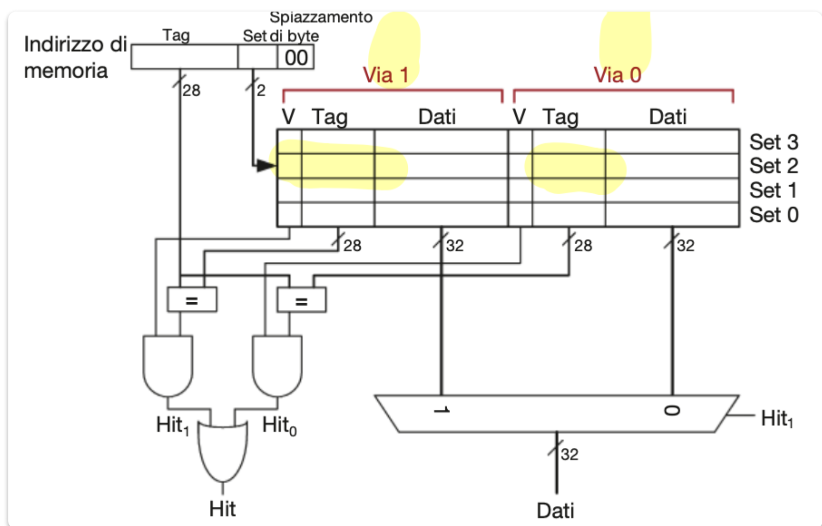
I due bit meno significativi sono denominati *spiazzamento di byte* perché indicano un byte all'interno della parola. I successivi tre bit sono denominati *bit di set* perché indicano in quale set viene mappato l'indirizzo (in generale, il numero di bit di set è $log_2 S$). I rimanenti 27 *bit di tag* indicano l'indirizzo del dato effettivamente contenuto in un certo set della cache.
La Figura mostra i campi di cache per l'indirizzo `0xFFFFFFE4`: tale indirizzo viene mappato nel set 1 e il suo tag è costituito da tutti 1.

La cache usa un *bit di validità* per ogni set che indica se il set contiene dati significativi. Se il bit di validità è 0, il contenuto del set non è significativo.
Quando due indirizzi generati di recente dal processore si mappano nel medesimo blocco di cache, si verifica un *conflitto*, e il dato cui si accede per ultimo *espelle* il precedente dal blocco. Le cache a mappatura diretta hanno un solo blocco in ogni set, quindi due indirizzi che si mappano nel medesimo set causano sempre un conflitto.

Cache parzialmente associative a molte vie

Una *cache parzialmente associativa* a N vie riduce i conflitti prevedendo N blocchi in ciascun set dove il dato che viene mappato in tale set può trovarsi: ogni indirizzo di memoria viene dunque mappato in uno specifico set, ma può essere copiato in uno qualsiasi degli N blocchi di tale set. Si può dunque dire che la cache a mappatura diretta equivale a una cache parzialmente associativa a una via. N viene definito come il grado di associatività della cache.

La Figura mostra la struttura circuitale di una cache parzialmente associativa a $N = 2$ vie di capacità $C = 8$ parole.



Cache completamente associativa

Una *cache completamente associativa* è costituita da un unico set a B vie, dove B è il numero di blocchi, quindi un indirizzo di

memoria può essere mappato in una qualsiasi di queste vie. La cache completamente associativa è dunque una cache parzialmente associativa a B vie.

La Figura mostra la SRAM di una cache completamente associativa con otto blocchi.

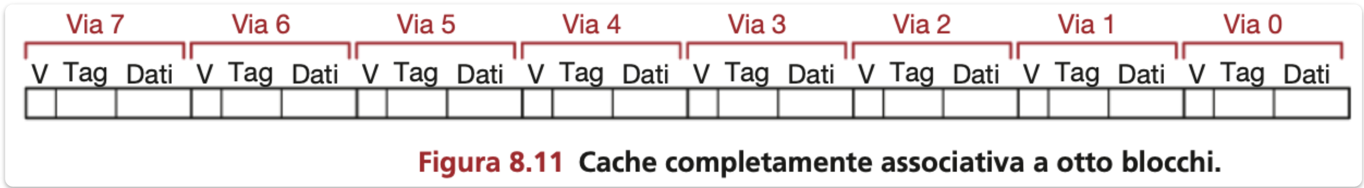


Figura 8.11 Cache completamente associativa a otto blocchi.

> *Alla richiesta di un dato, otto confronti di tag devono essere fatti, dal momento che il dato potrebbe trovarsi in un blocco qualsiasi.*

Dimensione del blocco

Negli esempi fatti finora si è sfruttata solo la *località temporale*, perché la dimensione del blocco era di una sola parola. Per sfruttare anche la *località spaziale*, una cache utilizza blocchi più grandi per memorizzare più parole di memoria consecutive.

Il vantaggio di una dimensione di blocco maggiore di 1 è il fatto che in caso di $miss$ vengono copiate in cache la parola non trovata e anche le parole adiacenti nel blocco di memoria principale. Gli accessi successivi hanno dunque maggiore probabilità di dare luogo a hit grazie alla *località spaziale*. Tuttavia, una dimensione di blocco grande significa, a parità di capacità della cache, che questa ha meno blocchi, quindi può dare luogo a un maggior numero di *conflitti* aumentando il tasso di $miss$. Inoltre, serve più tempo in caso di $miss$ per prelevare da memoria principale tutte le parole del blocco e trasferirle in cache: il tempo necessario per caricare in cache il blocco mancante viene detto *penalizzazione di miss*.

La Figura mostra la struttura circuitale di una cache a mappatura diretta con una dimensione di blocco b = 4 parole:

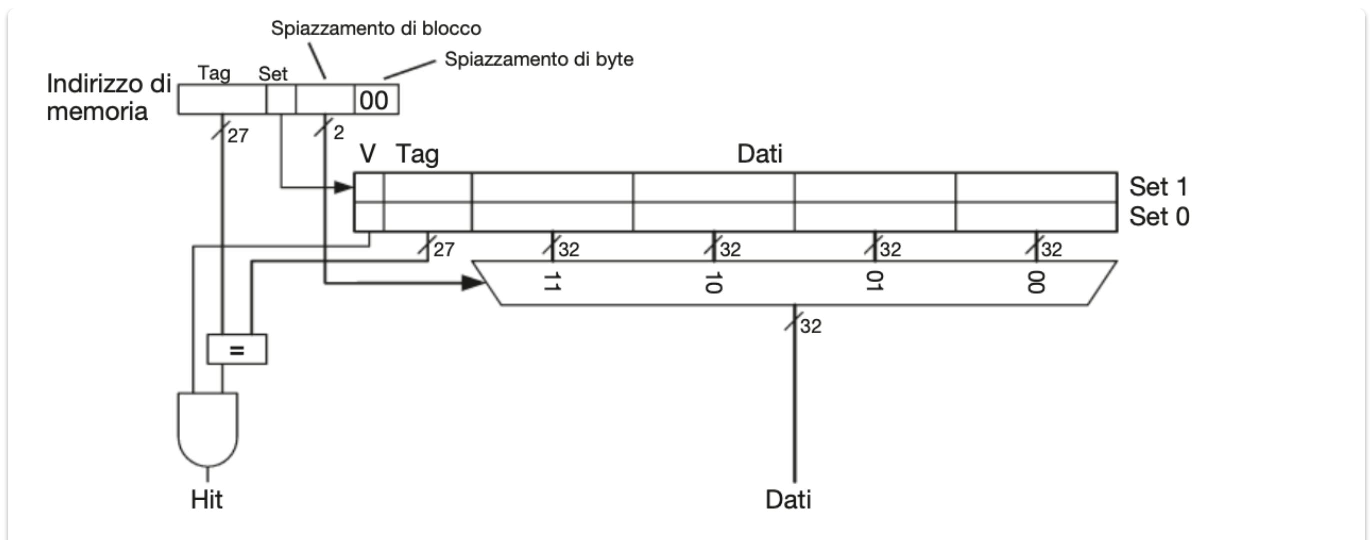


Figura 8.12 **Cache a mappatura diretta con due blocchi e una dimensione di blocco di quattro parole.**

La cache ha in questo caso solo $B = C/b = 2$ blocchi. Una cache a mappatura diretta ha solo un blocco in ciascun set, quindi questa cache è organizzata in due set: serve dunque solo $log_2 2 = 1$ bit per selezionare il set. Serve poi un multiplexer per selezionare la parola all'interno del blocco: tale multiplexer è controllato da $log_2 4 = 2$ bit di spiazzamento di blocco nell'indirizzo. I 27 bit più significativi dell'indirizzo costituiscono il tag: serve un solo tag per l'intero blocco, perché le parole del blocco si trovano a indirizzi consecutivi.

Quale dato viene sostituito?

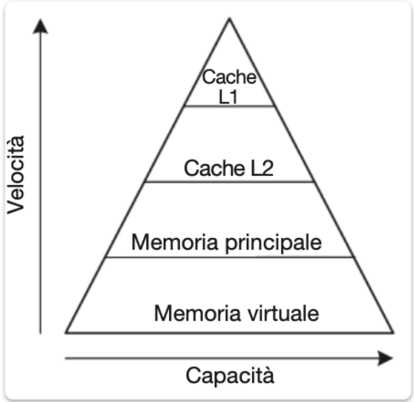
Il principio di località temporale suggerisce che la scelta migliore sia quella di espellere il blocco utilizzato meno recentemente, perché è quello con la minore probabilità di essere riutilizzato a breve. Questa politica è chiamata *LRU* (Least Recently Used).

In una cache parzialmente associativa a due vie, un *bit di utilizzo U*, indica quale via nel set è stata utilizzata meno recentemente. Per cache parzialmente associative a più di due vie le vie sono spesso divise in 2 gruppi e U indica quale gruppo di vie è quello usato meno recentemente. Al momento della sostituzione, il nuovo blocco sostituisce un blocco a caso del gruppo usato meno recentemente. Questa politica prende il nome di *pseudo-LRU*.

Progetto di cache avanzate

Cache multi livello

Cache di grandi dimensioni sono utili perché hanno maggiore probabilità di contenere i dati necessari e quindi di ridurre il tasso di $miss$, ma tendono a essere più lente delle cache piccole.
I moderni calcolatori adottano spesso almeno 2 livelli di cache, come mostrato nella Figura:



La *cache di primo livello* ($L1$) è abbastanza piccola da garantire un tempo di accesso di 1 o 2 cicli di clock;
La *cache di secondo livello* ($L2$) è pure realizzata con SRAM ma è più grande, quindi più lenta, della cache $L1$.

Come ridurre il tasso di miss

Le situazioni di miss possono essere ridotte modificando la *capacità*, la *dimensione di blocco* e/o l'*associatività della cache*.
La prima richiesta di un blocco di cache è definita *miss inevitabile* perché il blocco deve essere per forza letto dalla memoria principale indipendentemente dall'organizzazione della cache.
I *miss di capacità* si verificano quando la cache è troppo piccola per contenere tutti i dati utilizzati in modo concorrente.
I *miss di conflitto* si verificano, infine, quando più indirizzi vengono mappati nello stesso set ed espellono blocchi ancora necessari.

Politiche di scrittura

Le cache sono classificate nelle due categorie *write-through* e *write-back*.
In una cache *write-through*, il dato viene scritto simultaneamente sia nella memoria cache sia nella memoria principale.
In una cache *write-back*, un *bit di modifica* M (dirty bit) è associato a ogni blocco di cache: tale bit vale 1 se il blocco è stato modificato da almeno una scrittura, altrimenti vale 0. I blocchi di cache modificati vengono riscritti nella memoria principale solo al momento di essere espulsi dalla cache.

Sistemi Operativi

Gerarchie di memoria

Topics

- Tecnologie di memoria
- Gerarchia della memoria
- Memorie cache
- Misurazione e miglioramento delle prestazioni della cache

Tecnologie di memoria

Le memorie volatile e non volatile si distinguono principalmente per la persistenza dei dati memorizzati quando l'alimentazione viene interrotta.

- *Memorie volatili*:
  - Latch, flip-flop, registri, SRAM, DRAM.
- *Memorie non volatili*:
  - ROM, NVRAM, Memoria flash, Dischi magnetici, Dischi ottici, Nastro.

Obiettivo della gerarchia della memoria

Illusione di una memoria che è grande quanto il livello più grande della gerarchia e (quasi) veloce quanto il livello più vicino.

Illusione di una memoria grande e veloce:

- All'inizio, i dati risiedono nell'ultimo livello di memoria.
- I dati richiamati si spostano dal livello n al livello $n-1$ utilizzando diverse granularità di trasferimento dati (ad esempio, pagine, blocchi).

Terminologia

Se i dati richiesti dal processore compaiono in qualche blocco nel livello di memoria più vicino, questo viene chiamato *hit*. Altrimenti, la richiesta è chiamata *miss* e viene quindi acceduto il livello di memoria successivo per recuperare il blocco contenente i dati richiesti.

• *Hit-rate* è la frazione di accessi alla memoria trovati nel livello superiore.
• *Miss-rate* è la frazione di accessi alla memoria non trovati nel livello superiore.
• *Miss-penalty* è il tempo necessario per sostituire un blocco nel livello n con il blocco corrispondente dal livello n-1.
• *Miss-time* è il tempo necessario per ottenere l'elemento in caso di miss: miss-time = miss-penalty + hit-time

AMAT = hit-time + (1 - hit-rate) * miss-penalty

Principio di località

Il principio della località si riferisce al fenomeno in cui un programma tende ad accedere allo stesso insieme di posizioni di memoria per un dato periodo di tempo.

È stato osservato che, se il programma si riferisce a una posizione di memoria, allora:

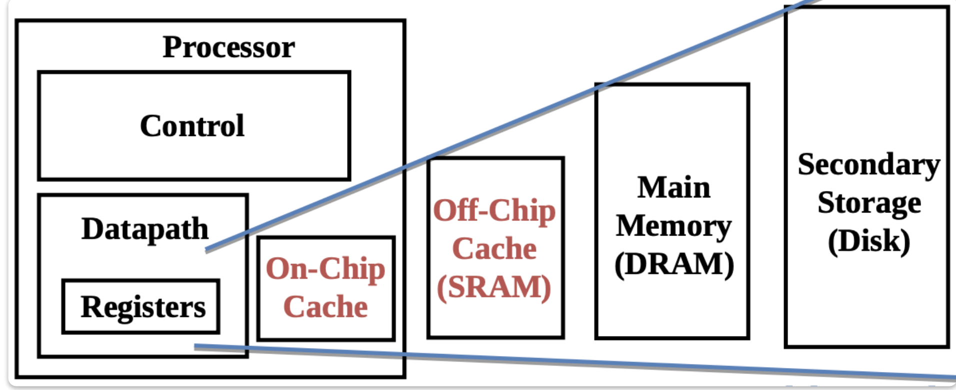
- La stessa posizione di memoria verrà utilizzata nuovamente presto con una probabilità alta (*Località Temporale*).
- Gli elementi "vicini" alla memoria appena acceduta saranno referenziati presto con probabilità alta (*Località Spaziale*).

Trasferimento dati

- I dati vengono trasferiti solo tra due livelli di memoria adiacenti alla volta
- I trasferimenti avvengono sempre con granularità a blocco per sfruttare la località spaziale.

Memorie cache

La memoria cache è il livello della gerarchia della memoria più vicino alla CPU.



Lo spostamento tra i vari livelli di memoria è così gestito:

- Tra la cache di primo livello e i registri è gestito dal *compilatore*.
- Tra i livelli di cache e tra cache e RAM è gestito dall'*hardware* (microarchitettura).
- Tra i dispositivi di archiviazione e la RAM è gestito dal *SO e dalle applicazioni*.

Utilizzo delle cache

- Le cache sono organizzate in *linee*, ogni linea contiene un *blocco* di parole di memoria (ad esempio, 8 parole di memoria).
- La prima volta che il processore richiede una parola di memoria si verifica un *miss* di cache.
  - Il blocco contenente la parola di memoria viene trasferito nella cache.
- Richieste successive:
  - Se i dati sono presenti nel blocco → *hit di cache*.
  - Se i dati non sono presenti → *miss di cache*.
    - Il blocco contenente i dati viene trasferito in una linea di cache.

Come vengono trovati i dati?

- Una cache di *Capacità* C è organizzata in S *Set*, ognuno contenente B *Blocchi* (o linee). b è il *numero di parole* per

blocco.

- La funzione di mapping mappa un indirizzo di memoria esattamente su un insieme.
- Le cache possono essere categorizzate in base al numero di blocchi in un insieme:
  - *Mappatura diretta*: $S = B$
  - *Associativa a N-vie*: N blocchi. $S = B/N$
  - *Completamente associativa*: $S = 1$

Mappatura diretta

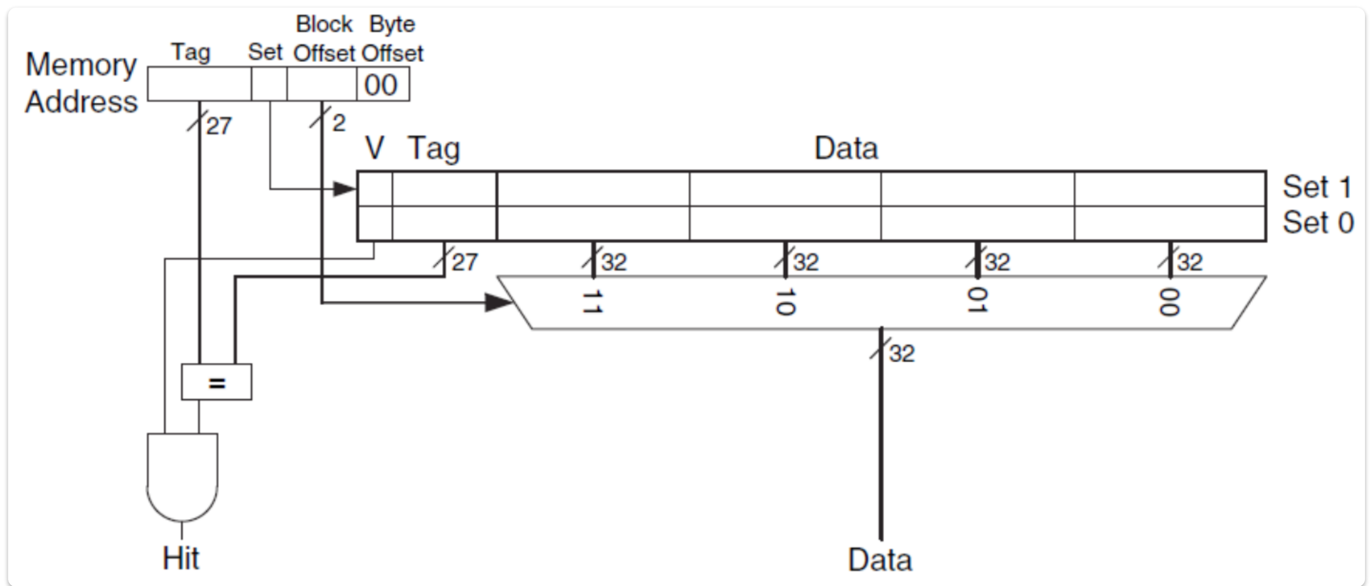
Come facciamo a sapere quale blocco è in una posizione di cache?

- Abbiamo bisogno di un insieme di *tag* per ogni posizione di cache che identifichi l'indirizzo del blocco nella cache.

Come facciamo a sapere se i dati in una posizione di cache sono validi?

- Abbiamo bisogno di un *bit di validità* per ogni posizione di cache (valido = 1; non valido = 0).

Indirizzo a 32 bit + 1 bit nel set per indicare la validità.
4 Words per set per sfruttare la località spaziale.



Vantaggi:

- Semplice da realizzare

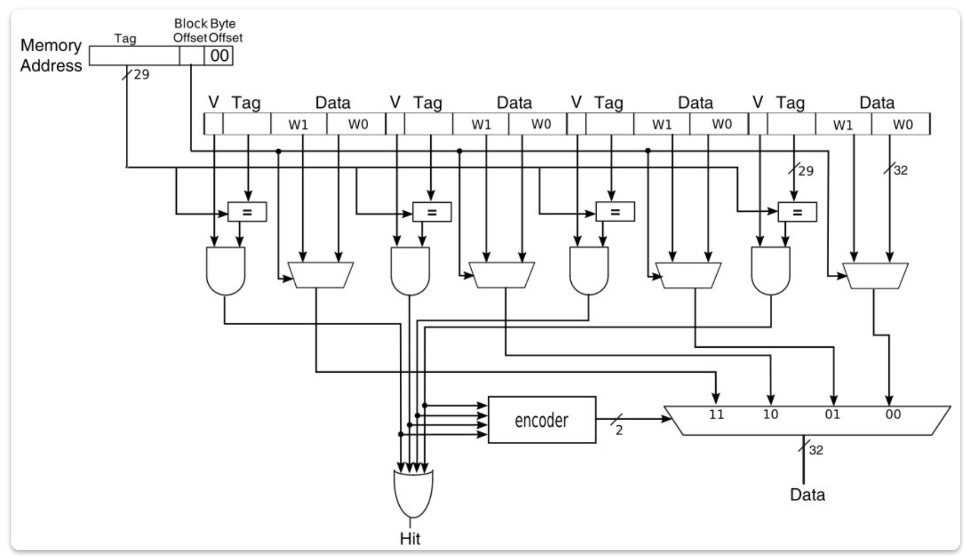
- Molto veloce in caso di hit di cache

Svantaggi:

- Troppo rigida, una data word può essere collocata in una sola entry della cache
- Questa rigidità può avere un grande impatto sul numero di conflitti di cache a seconda del posizionamento della memoria e dell'uso delle strutture dati

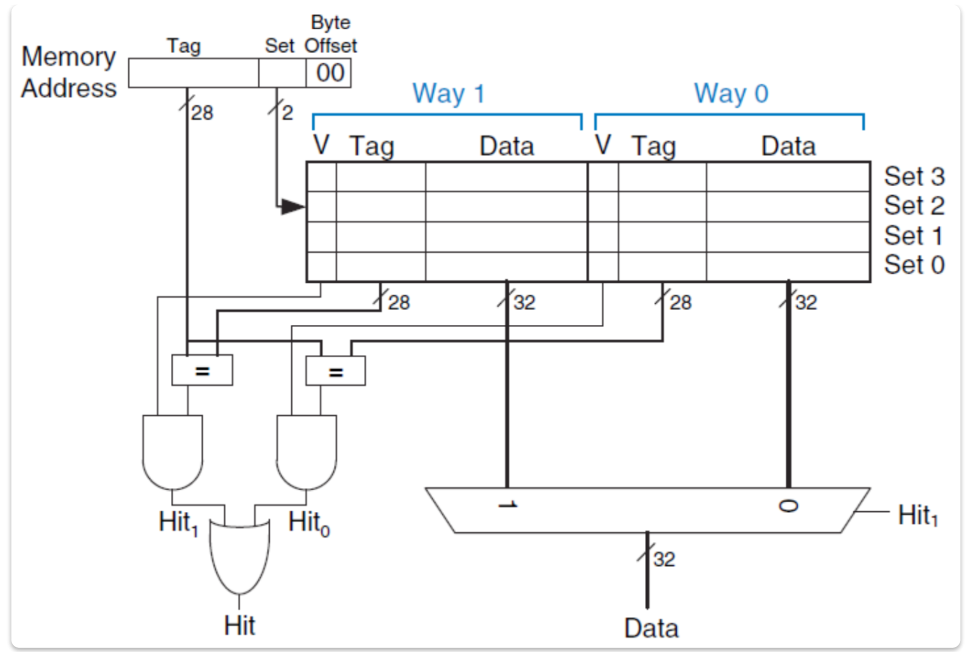
Cache associative

Cache completamente associativa



- *Blocchi* ⟶ qualsiasi entry della cache
- Per trovare un dato blocco è necessario cercare in tutte le entry in modo parallelo.
- Ogni entry ha un comparatore

Cache set-associativa a N-vie



- I Blocchi possono essere collocati in un numero fisso di N entry
- Ogni indirizzo di Blocco è mappato esattamente in un insieme che contiene N entry
- Un Blocco può essere collocato in qualsiasi entry dell'insieme
- Per trovare un dato Blocco, la ricerca viene effettuata in parallelo nelle N entry dell'insieme (utilizzando N comparatori distinti)

Cache coherence problem

Nei multiprocessori, la *Cache coherence* è un problema in cui le modifiche ai dati da parte di un processore potrebbero non essere riflesse correttamente nelle cache degli altri processori, causando dati obsoleti o inconsistenti.

Tipi di miss di cache

Miss obbligatori:

- Questi sono miss di cache causati dal primo accesso a un blocco che non è mai stato nella cache.

Miss di capacità:

- Sono causati quando la cache non può contenere tutti i blocchi necessari. Alcuni blocchi vengono eliminati e successivamente recuperati.

Miss di conflitto:

- Solo per cache a mappatura diretta e cache set-associative

Gestione dei miss di cache

Un miss di cache blocca l'intero processore, congelando i contenuti di tutti i registri in attesa della memoria.

I passaggi intrapresi per un miss di cache sono:

1. Informare il livello di memoria successivo di leggere il valore mancante.
2. Attendere che la memoria risponda (questo può richiedere più cicli).
3. Aggiornare la linea di cache corrispondente con i dati ricevuti.
4. Riprendere l'esecuzione dell'istruzione (ora è un hit di cache).

Gestione delle scritture

Write misses:

- *Write-allocate*: Il Blocco viene caricato nella cache seguito da un hit di scrittura (usato da write-back).
- *No-write-allocate*: Il Blocco non viene caricato nella cache (usato da write-through).

Write hits:

- Se l'istruzione di scrittura scrive i dati solo nella cache, allora cache e memoria sono inconsistenti

Write-Through

Le scritture che colpiscono aggiornano sempre sia la cache che il livello di memoria inferiore

Vantaggi:

- Soluzione semplice, facile da implementare
- I dati sono sempre consistenti tra i due livelli di memoria.

Svantaggi:

- La velocità delle scritture dipende dal livello di memoria inferiore.
- Traffico di memoria più elevato.

Write-Back

Le scritture che colpiscono aggiornano solo la cache, quindi il blocco modificato viene scritto quando viene sostituito

Vantaggi:

- La velocità delle scritture è quella della cache.
- Traffico di memoria inferiore.

Svantaggi:

- È necessario tenere traccia dei blocchi modificati (dirty bit).
- Più complesso da implementare.
- La sostituzione della linea di cache è più costosa.

Politica di Sostituzione della Cache

Least Recently Used

LRU sostituisce il blocco meno recentemente usato, massimizzando la località temporale. Per cache a 2-vie, usa 1 bit di utilizzo U; a 4-vie, 2 bit; oltre diventa complesso.
In cache ad alta associatività, il comportamento casuale approssima LRU.

Input Output

Topics

- Dispositivi di I/O
- Controller di dispositivi
- Bus
- Gestione dell'I/O
- Driver dei dispositivi

Dispositivi di I/O

Legge di Amdahl

Si consideri un programma in cui solo la frazione f può essere ottimizzata (ad esempio, parallelizzata utilizzando N processori), mentre la frazione $(1 - f)$ rimane inalterata.
La velocità di accelerazione è limitata dalla frazione sequenziale $(1 - f)$.

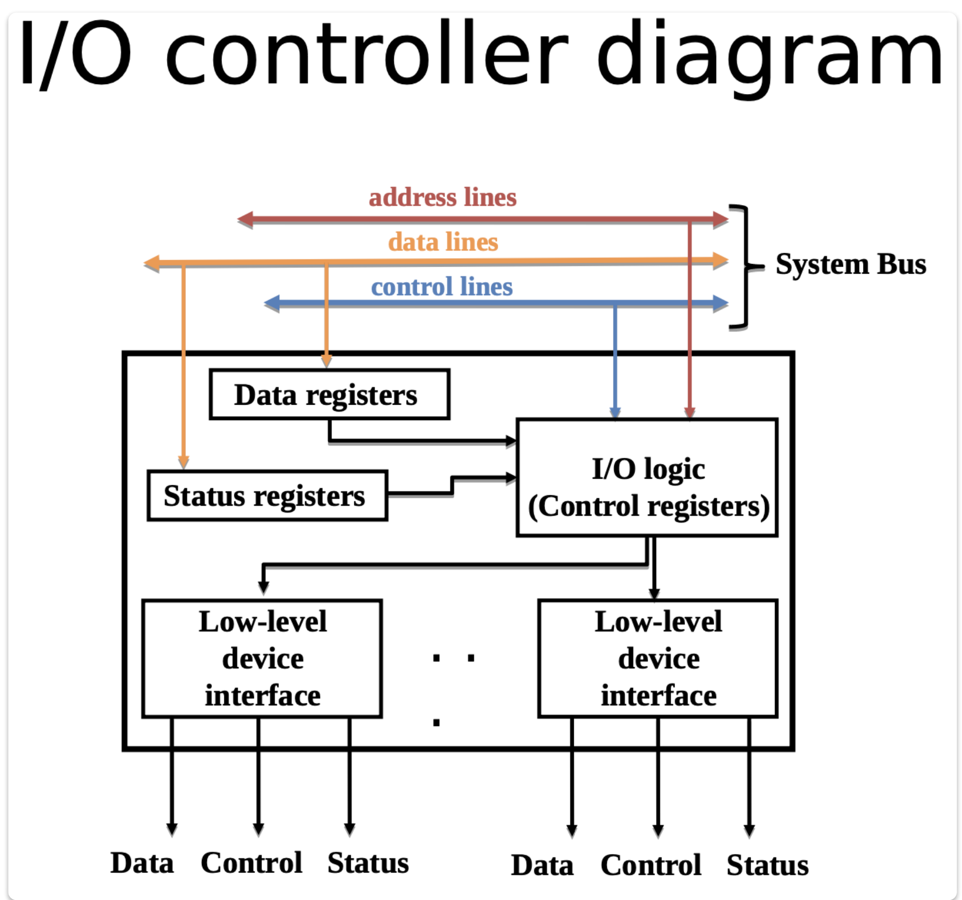
I/O: controllo + dati

I dispositivi I/O hanno 2 porte:

- *Controllo*: sia comandi che segnalazioni di stato.
  - Come diciamo al dispositivo cosa fare.
  - Come il dispositivo ci informa sulle sue funzionalità e sullo stato operativo.
- *Dati*:
  - Da/a memoria del dispositivo.

Funzioni I/O Controller

- Controllo e sincronizzazione
- Comunicazione con il processore
  - Decodifica dei comandi
  - Scambio di dati
  - Segnalazione dello stato
  - Riconoscimento dell'indirizzo
- Comunicazione con il dispositivo
- Buffering dei dati
- Rilevamento degli errori



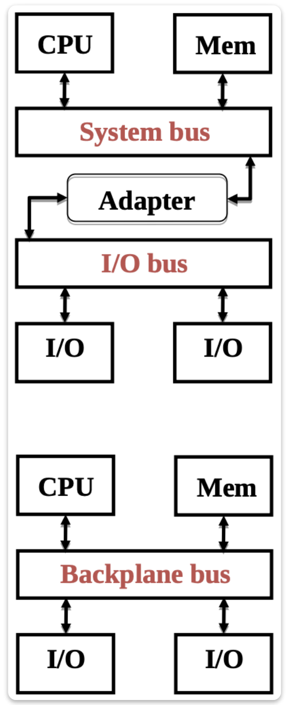
Bus

Raccolta di linee dati trattate come un unico segnale logico.

Esempi di Bus

- *Bus di sistema*: Collega processore, memoria e I/O tramite adattatori. Veloce e ad alta banda.

- *Bus I/O*: Collega dispositivi I/O. Più lento e banda più bassa rispetto al bus di sistema.
- *Bus posteriore*: Collega CPU, memoria e dispositivi I/O. Velocità media/bassa. Standard industriale.



Progettazione del bus

Obiettivi: alta performance, standardizzazione, basso costo.

Problemi di progettazione:

- Condivisione o separazione dei fili?
- Acquisizione e rilascio del controllo del bus.
- Sincronizzazione del bus.
- Arbitraggio del bus (quale dispositivo ha il diritto di trasmettere dati sul bus).

Sincronizzazione del bus

Sincrono:

- Dispositivi condividono lo stesso clock di bus.
- Limitato a bus corti.

Asincrono:

- Il bus non ha un clock.
- Può essere più lungo e richiede protocolli di handshake.

Arbitraggio del bus

Gestito da un protocollo di comunicazione.
Bus master: unità che può iniziare una richiesta di bus.

- Configurazione centralizzata: un solo master.
- I bus hanno solitamente più master.
  Arbitrato: scegliere un master tra le richieste per implementare priorità ed equità.
- Un solo master alla volta.
- Ruolo dell'*arbitro*: gestire le richieste e assegnare i permessi.
  Modelli di implementazione: Daisy-chain, Centralizzato, Distribuito.

Gestione dell'I/O

1. Come la CPU invia comandi ai dispositivi I/O?
2. Come la CPU sa quando i dispositivi I/O hanno completato le operazioni?
3. Come eseguono i dispositivi I/O i trasferimenti di dati?

Invio comandi ai dispositivi I/O

Solo il sistema operativo può inviare comandi.
I programmi nello spazio utente usano le System Call.

Due opzioni:

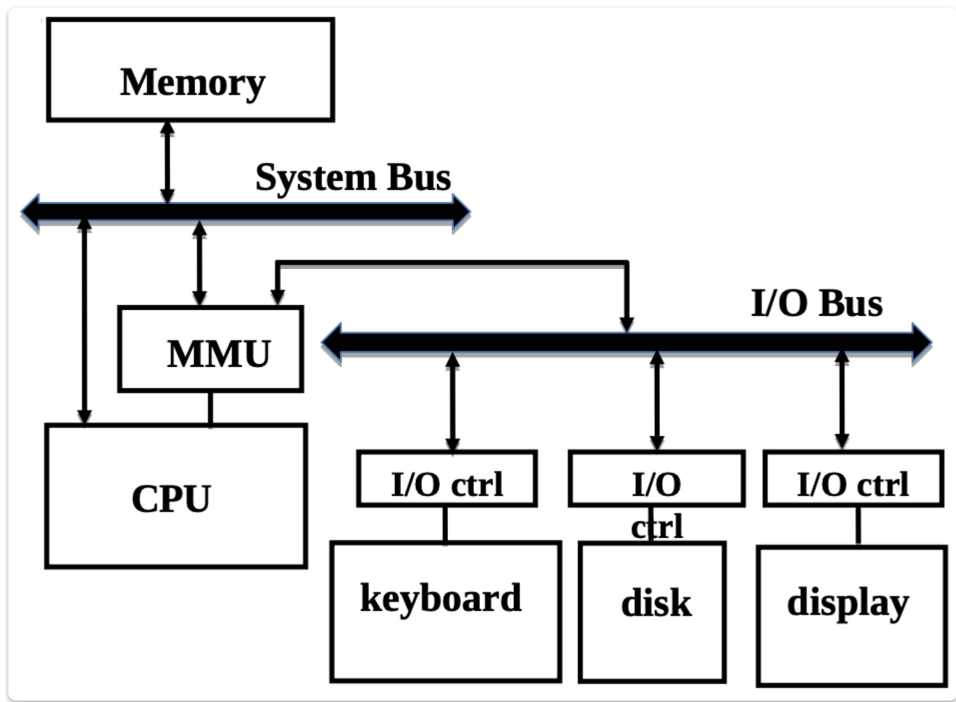
- *Istruzioni I/O*: privilegiate e indirizzano registri.
- *Memory Mapped I/O*: parte dello spazio fisico è dedicato all' I/O.

Memory Mapped I/O

I registri interni dei dispositivi sono mappati a locazioni della memoria principale (a *indirizzi fisici riservati*).

I comandi I/O sono standard letture/scritture di memoria.

Le operazioni a tali locazioni sono reindirizzate ai controller dei dispositivi dalla MMU.
Gli accessi in modalità utente alle aree mappate in memoria generano eccezioni di violazione di memoria.



Interrogare lo stato I/O

Come la CPU sa se la richiesta di operazione è stata completata?

I/O programmato

La CPU controlla direttamente l'operazione e lo stato I/O attraverso il *Polling*.

- *Vantaggi*: Semplice da implementare.
- *Svantaggi*: Spreco di cicli del processore.
    - La CPU è molto più veloce di qualsiasi dispositivo I/O.
    - La CPU potrebbe leggere il registro di stato molte volte solo per scoprire che il dispositivo non ha completato l'operazione.

I/O basato su interrupt

Le *interrupt* sono un'alternativa al *polling*:

- Risolvono il problema dello spreco di tempo della CPU.
- La CPU continua altre attività mentre il dispositivo I/O lavora.
- L'*interrupt* viene generato quando i dati sono pronti.
- Priorità degli interrupt:
  - dispositivi ad alta larghezza di banda > dispositivi a bassa larghezza di banda.

Dalla prospettiva della CPU:

- Emette comandi I/O (inviare istruzioni ai dispositivi di input/output per eseguire operazioni specifiche).
- Continua altre attività.
- Controlla l'interrupt alla fine del ciclo fetch-execute.
- Se riceve un *interrupt*:
  - Salva e passa al livello privilegiato.
  - Esegue il gestore dell'interrupt.
  - Ripristina il contesto per l'istruzione successiva.
- Queste sono considerate *operazioni atomiche* dal punto di vista del sistema operativo.

Data Transfers (DMA)

Come eseguono i dispositivi I/O i *trasferimenti dati*?

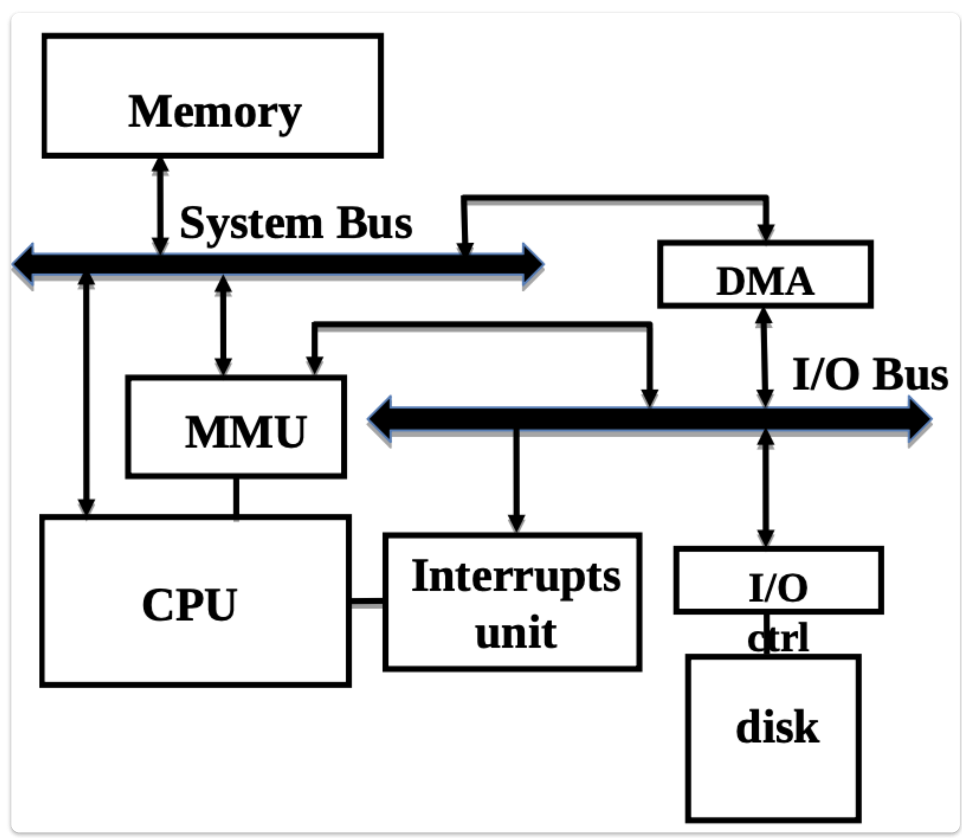
- L'I/O basato su *interrupt* risolve i problemi del *polling*, ma:
  - Il sistema operativo deve ancora trasferire i dati uno per uno.
  - Il costo della gestione degli interrupt è maggiore rispetto al polling.
- *Accesso diretto alla memoria* (DMA):
  - Un meccanismo che consente ai controller di I/O di trasferire dati direttamente alla e dalla memoria principale senza coinvolgere la CPU.
  - Gli *interrupt* vengono utilizzati solo al completamento del trasferimento I/O o in caso di errore.

Controller DMA

- La DMA è implementata con un *controller specializzato*
- Un dispositivo I/O è collegato a un controller DMA per eseguire l'operazione DMA.
    - Più dispositivi possono essere collegati allo stesso controller.
- Il controller è visto come un *memory-mapped I/O*.
- Si occupa dell'arbitrato del bus e dei trasferimenti dati.
- Diventa il master del bus.
- Condivide il bus di memoria con la CPU.

Trasferimento DMA:

1. La CPU configura il DMA fornendo identità, operazione, indirizzo di memoria e numero di byte da trasferire.
2. Il DMA avvia l'operazione sul dispositivo e arbitra la connessione. Trasferisce i dati dal dispositivo o dalla memoria.
3. Una volta completato il trasferimento DMA, il controller invia un'interruzione alla CPU.



DMA e Gerarchia della Memoria:

- *Senza DMA*: il processore gestisce tutti i trasferimenti dati.
    - Attraversa la *traduzione degli indirizzi* (MMU).
    - I trasferimenti possono superare i limiti delle pagine virtuali.
    - Nessun impatto sulla gerarchia della cache.
- *Con DMA*: il controller DMA gestisce i trasferimenti dati.
    - Aggiunge un altro percorso verso la memoria.
    - Possibili problemi:
        1. Utilizzo di indirizzi virtuali o fisici?
        2. Scrittura di dati in posizioni cache?

Virtual DMA: il controller DMA esegue la traduzione degli indirizzi internamente utilizzando una cache (TLB) piccola. Complesso ma flessibile.

> *La cache riduce la latenza di accesso della CPU e libera più banda per i trasferimenti DMA, ma introduce problemi di coerenza. Soluzione: invalidare selettivamente le linee cache coinvolte nei trasferimenti DMA.*

Fisico DMA: il controller DMA lavora con indirizzi fisici, trasferendo dati a livello di pagina. Più semplice ma meno flessibile.

Driver dei dispositivi

Il *file system* definisce lo spazio dei nomi e le politiche di caching, è indipendente dall'hardware.

Il *driver del dispositivo* lavora nello spazio del kernel, gestisce l'interfaccia con il controller del dispositivo, la sincronizzazione e i trasferimenti di dati.

....
....

....

Operating Systems: Principles and Practice

Topics

- Definizione di sistema operativo
  - Software per gestire le risorse di un computer per gli utenti e le applicazioni
- Sfide dei sistemi operativi
  - Affidabilità, sicurezza, reattività, portabilità, …

Cosa è un Sistema Operativo

Funzionalità Principali:

- Pianificazione delle risorse
- Memoria Virtuale
- Comunicazione e sincronizzazione tra processi (IPC)

Ruoli del SO

Arbitro: Gestisce l'allocazione delle risorse e garantisce l'isolamento tra utenti e applicazioni.

Illusionista: Creare l'illusione di risorse illimitate per ogni singola applicazione.

Collante: Fornisce servizi comuni e standard per semplificare lo sviluppo delle applicazioni.

Sfide dei SO

Affidabilità:

- Il sistema fa ciò per cui è stato progettato?

- Disponibilità
  Sicurezza:
- Il sistema può essere compromesso da un attaccante?
- Privacy
  Portabilità:
- Per i programmi:
  - Interfaccia di programmazione delle applicazioni (API) e Interfaccia astratta della macchina
- Per il sistema operativo:
  - Livello di astrazione dell'hardware
  Prestazioni:
- Latenza
  - Quanto tempo impiega un'operazione per completarsi?
- Throughput
  - Quante operazioni possono essere eseguite per unità di tempo?
- Overhead
  - Quanto lavoro aggiuntivo viene svolto dal sistema operativo?
- Equità
  - Quanto è equa la performance ricevuta da utenti diversi?
- Prevedibilità
  - Quanto è consistente la performance nel tempo?

Struttura del SO:

Trade-off nella progettazione del SO:

- *Kernel monolitico*:
  - La maggior parte delle funzionalità del SO viene eseguita all'interno del kernel.
- *Microkernel*:
  - Nel kernel vi sono solo meccanismi centrali, la maggior parte delle funzionalità del SO viene eseguita come server a livello utente.
  - Più affidabile grazie all'isolamento maggiore; più facile da debuggare, mantenere ed estendere.

- *Modello ibrido*:
  - Combina le migliori pratiche dei due approcci precedenti.

....

....

....

The Kernel Abstraction

Topics

Concetto di *processo*:

- Un processo è un'astrazione del SO per l'esecuzione di un programma con privilegi limitati
  Operazione in *dual-mode*: user mode vs kernel mode
- *Kernel mode*: eseguire con privilegi completi
- *Utente mode*: eseguire con meno privilegi
  Safe control transfer:
- Come passiamo da una modalità all'altra?

Concetto di processo

Processo: una sequenza di attività avviata da un programma, eseguita con privilegi limitati.

- Il *Process Control Block* (PCB) tiene traccia delle informazioni del processo.
- La *tabella dei processi* contiene tutti i PCB.
- Include *thread* (esecutori di istruzioni) e *spazio degli indirizzi* (memoria accessibile).

Programma: sequenza statica di istruzioni.
Processo: una sequenza di attività descritta da un programma.

Più processi possono essere attivati sullo stesso programma.

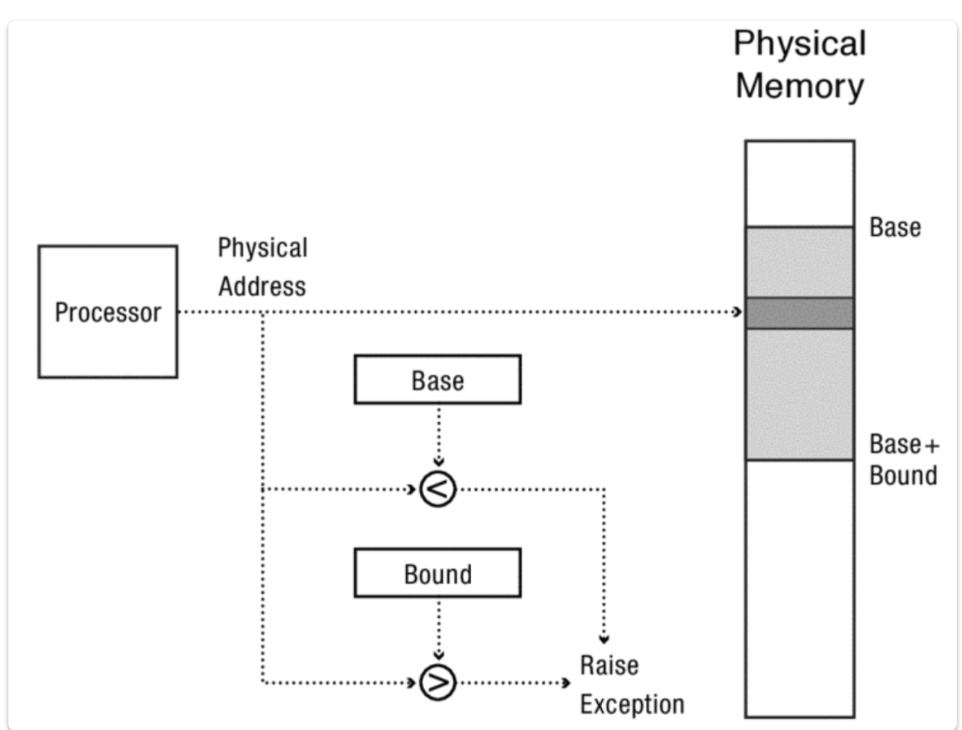
Il PCB è una struttura dati che contiene informazioni su un processo, come lo stato corrente, i registri della CPU, i puntatori ai thread e le risorse assegnate come memoria e file aperti.

Dual-Mode Operation

Nel *dual-mode operation*, il sistema può operare in due modalità:

- *kernel mode*: accesso completo alle risorse hardware e ai privilegi.
- *user mode*: privilegi limitati solo a quelli concessi dal kernel del sistema operativo.

Protezione della memoria con indirizzi fisici basata su Base-Bound



Il metodo *base-bound* viene utilizzato per fornire protezione alla memoria, limitando l'accesso alla memoria utilizzata da un processo

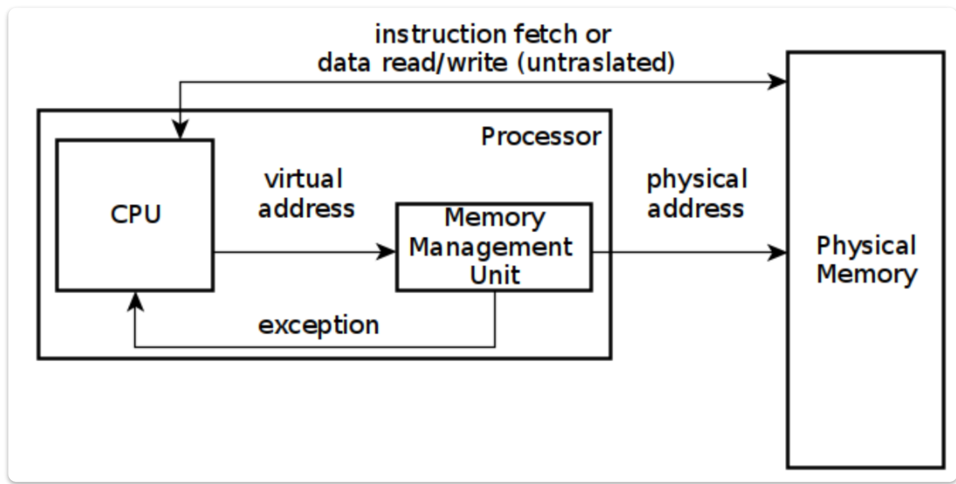
Utilizza due registri, *base* e *bound*, per definire l'intervallo di memoria accessibile da un processo.

Problemi con base-bound?

- Nessuna o limitata possibilità di espansione di heap e stack
- Assenza di condivisione della memoria
- Necessità di lavorare con indirizzi fisici → difficile spostare la memoria
- Frammentazione della memoria

Indirizzi Virtuali

- Traduzione eseguita dall'hardware, utilizzando una tabella impostata dal kernel
- Di solito, l'MMU si occupa della traduzione degli indirizzi



Mode Switch

Da user-mode a kernel-mode:

- *Interruzioni*:
  - Scatenate da timer e dispositivi di I/O
- *Eccezioni*:
  - Scatenate da comportamenti imprevisti del programma o comportamenti maligni
- *Chiamate di sistema*:
  - Richiesta del programma al kernel di eseguire un'operazione per suo conto

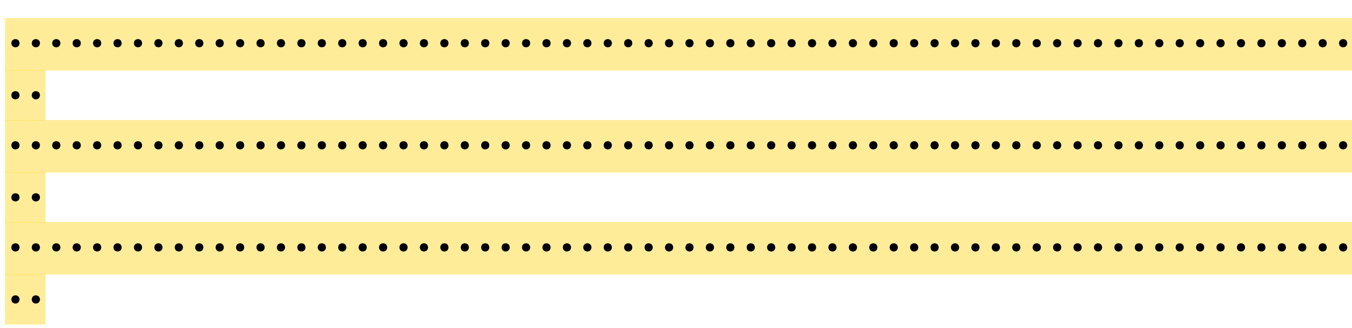
    - Solo un numero limitato di punti di ingresso molto attentamente codificati
    - Chiamate anche Interruzioni Software (SWI)

Da `kernel-mode` a `user-mode`:

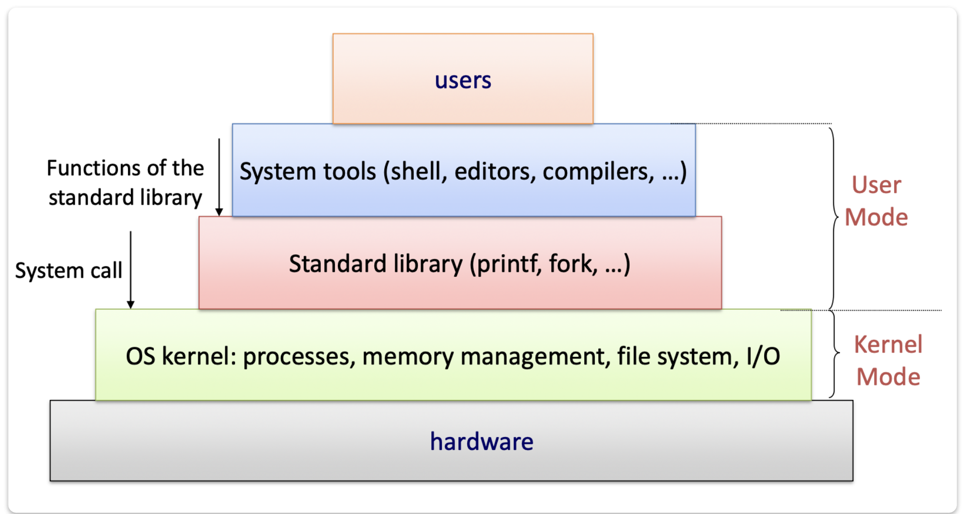
- *Ritorno da interruzioni, eccezioni, chiamate di sistema*
    - Riprende l'esecuzione sospesa
- *Avvio di un nuovo processo/nuovo thread*
    - Salto alla prima istruzione nel programma/thread
- *Cambio di contesto del processo/thread*
    - Ripresa di un altro processo
- *User-level upcall*
    - Notifica asincrona al programma utente

Come gestire le interruzioni in modo sicuro?

- *Vettore e stack delle interruzioni*:
    - Situato nella memoria del kernel ed è specifico per ogni processore.
- *Mascheramento delle interruzioni*:
    - L'handler delle interruzioni viene eseguito con le interruzioni disabilitate e vengono riattivate quando l'handler completa l'operazione.
- *Trasferimento atomico del controllo*:
    - Salva il puntatore dello stack corrente
    - Salva il contatore del programma corrente
    - Salva la parola di stato del processore corrente (condizioni)
    - Passa allo stack del kernel; mette SP, PC, PSW sullo stack
    - Passa alla modalità kernel
    - Indirizza attraverso la tabella delle interruzioni
    - Passa all'handler, PC e PSW del kernel
    - L'handler dell'interruzione salva i registri che potrebbero essere sovrascritti
    - Gestione delle interruzioni ...
- *Esecuzione trasparente e riavviabile*



Processi



Topic

- Creazione e Gestione dei processi
    - fork(), exec(), wait(), exit(), ...

Shell

Una *shell* è un sistema di controllo dei processi che consente agli utenti di creare e gestire un insieme di programmi per eseguire determinati compiti.

Creazione

La chiamata di sistema per creare un nuovo processo coinvolge diverse fasi:

- Creare e inizializzare il PCB nel kernel.

- Creare e inizializzare un nuovo spazio degli indirizzi.
- Caricare il programma nello spazio degli indirizzi.
- Copiare gli argomenti nella memoria dello spazio degli indirizzi.
- Inizializzare il contesto hardware per avviare l'esecuzione dal punto di inizio specificato.
- Informare lo scheduler che il nuovo processo è pronto per essere eseguito.

Gestione UNIX

La gestione dei processi in UNIX comprende diverse chiamate di sistema:

- `fork`: crea una copia del processo corrente e lo avvia in esecuzione, senza argomenti.
- `exec`: cambia il programma in esecuzione nel processo corrente.
- `wait`: attende il completamento di un processo.
- `signal`: invia notifiche tra processi.

Terminazione

La terminazione di un processo in UNIX può avvenire per diverse ragioni:

- A causa di un'eccezione.
- Invocando `exit`.

Quando un processo termina, restituisce un valore di uscita al suo processo padre:

- Se il padre non ha già chiamato `wait`, il processo terminato passa allo stato zombie.
- Se il processo padre è già terminato, il processo *init* (primo processo avviato dal kernel durante l'avvio del SO → PID = 1) si occupa dell'attesa della terminazione dei figli.

....

....

....

Concurrency

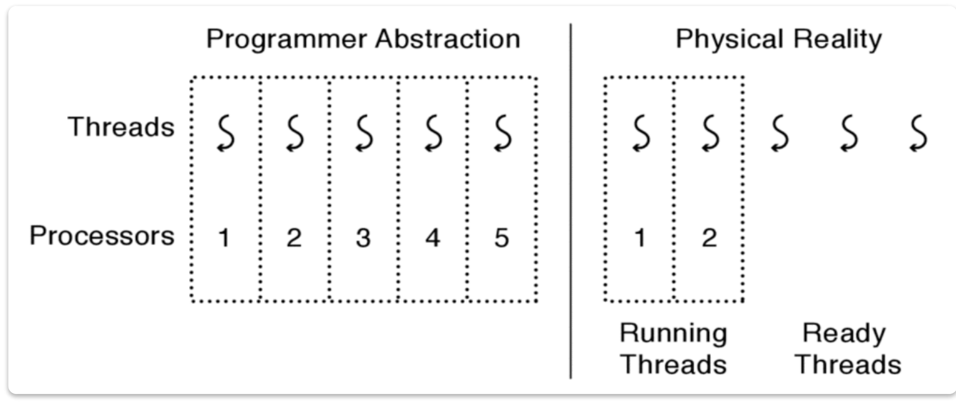
Thread

Un *thread* è una singola sequenza di esecuzione separata e schedulabile, utilizzata per strutturare programmi, garantire responsività, sfruttare i multi-core e interagire con dispositivi di I/O lenti.

Un processo può avere uno o più thread:

- Programma utente single-threaded
- Programma utente multi-threaded
- Kernel multi-threaded: più thread che condividono le strutture dati del kernel, in grado di utilizzare istruzioni privilegiate.

Il concetto di *astrazione* dei thread prevede un numero illimitato di processori, con un processore dedicato a ciascun thread. Questa disposizione consente ai thread di eseguire con velocità variabile, il che richiede che i programmi siano progettati in modo da funzionare con qualsiasi programma.



One Execution

Thread 1

Thread 2

Thread 3

Another Execution

Thread 1

Thread 2

Thread 3

Another Execution

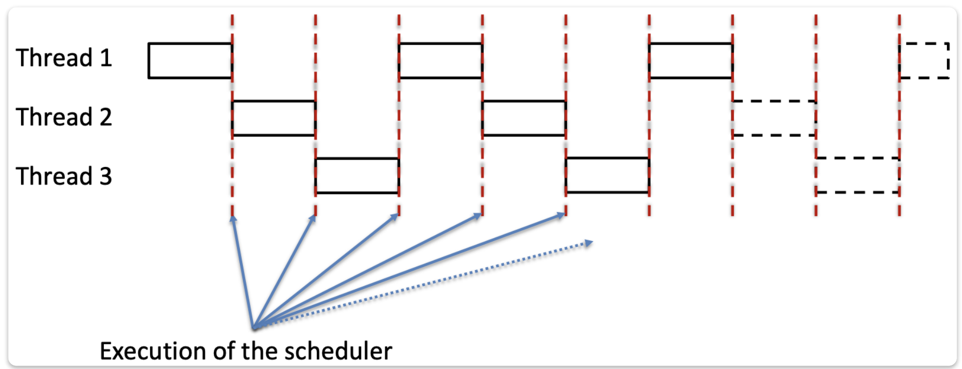
Thread 1

Thread 2

Thread 3

L'implementazione dei *thread* richiede diversi elementi, tra cui il *Thread Control Block* (TCB), una struttura dati che memorizza informazioni sul thread. È inoltre necessario definire un insieme di operazioni sui thread e un *scheduler*, una funzione del sistema operativo che assegna i processori ai thread.

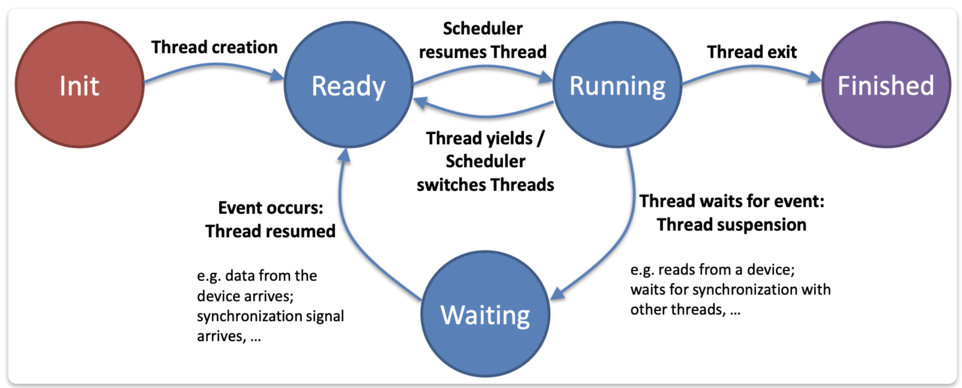
- PCB (Process Control Block): Struttura dati associata ad ogni processo.
- *Tabella dei processi*: Contiene tutti i PCB, uno per l'intero sistema nel kernel.
- TCB (Thread Control Block): Una per ogni thread.
- *Tabella dei thread*: Una per ogni processo (thread di livello utente), una per l'intero sistema nel kernel (thread di livello kernel).



Funzioni per thread

- `thread_create(thread, func, args)`

- `thread_yield()`
- `thread_join(thread)`
- `thread_exit(exit_status)`



User-level threads

Gli *User-level thread* sono implementati attraverso una libreria a livello utente, senza coinvolgere il sistema operativo. Ogni processo contiene una tabella dei thread nello spazio utente, e lo scheduling dei thread è gestito dal *RunTime Support* (RTS) del processo.

I thread possono utilizzare la funzione `thread_yield()` per rilasciare il processore, e un'attivazione dello scheduler consente lo *scheduling preemptive* (Scheduler interrompe thread quando vuole).

Vantaggi:

- Creazione, terminazione e cambio di contesto molto efficienti: non richiedono chiamate di sistema, solo chiamate alla libreria dei thread.
- Possono essere implementati su qualsiasi SO che non supporti il multi-threading.

Limitazioni:

- Le sys-call bloccanti bloccano tutti i thread di livello utente di un processo.
- Non sfruttano le architetture multiprocessore: tutti i thread di un processo vengono pianificati sullo stesso processore.

Kernel-level threads

Sono gestiti direttamente dal *kernel*, consentendo loro di sfruttare appieno le architetture multiprocessore.

- Operazioni sui thread e interazioni tra i thread tramite sys-call
- Maggiore overhead rispetto ai thread di livello utente.
- Lo scheduling dei thread è implementato dal sistema operativo.
- I thread possono invocare chiamate di sistema bloccanti, ma solo l'invocatore viene bloccato.

Thread Switch

Due cause:

- *Volontaria*
- Dovuta a un'*interruzione/eccezione*.

Modo *volontario* sia per i thread user-level sia kernel-level:

- Si salvano i registri
- Si passa al nuovo thread
- Si ripristinano i registri dal TCB del nuovo thread.

Per *interruzione*:

- L'handler dell'interruzione salva i registri nello stack del kernel e chiama una funzione per lo switch dei thread.
- Altra versione: l'handler salva direttamente i registri nel TCB e riprende lo stato salvato dal TCB.

....

....

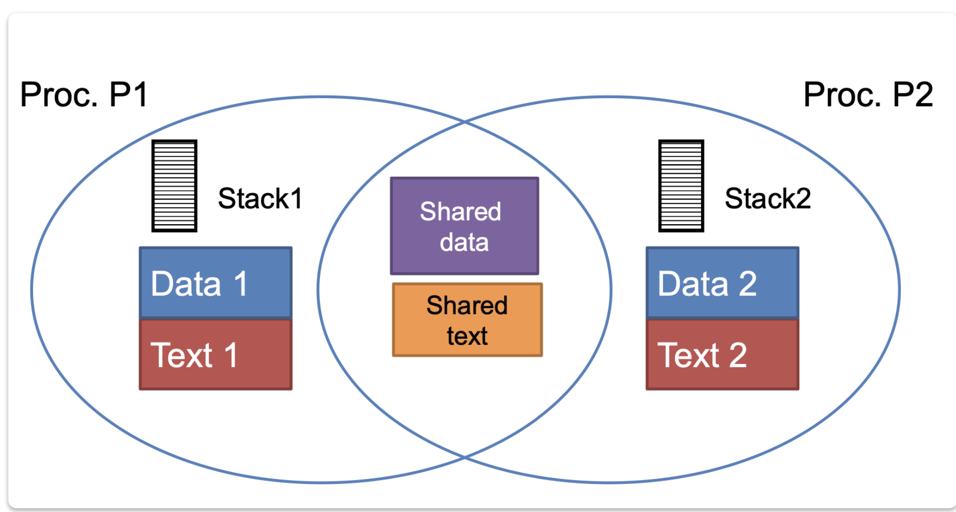
....

Sicronizzazione

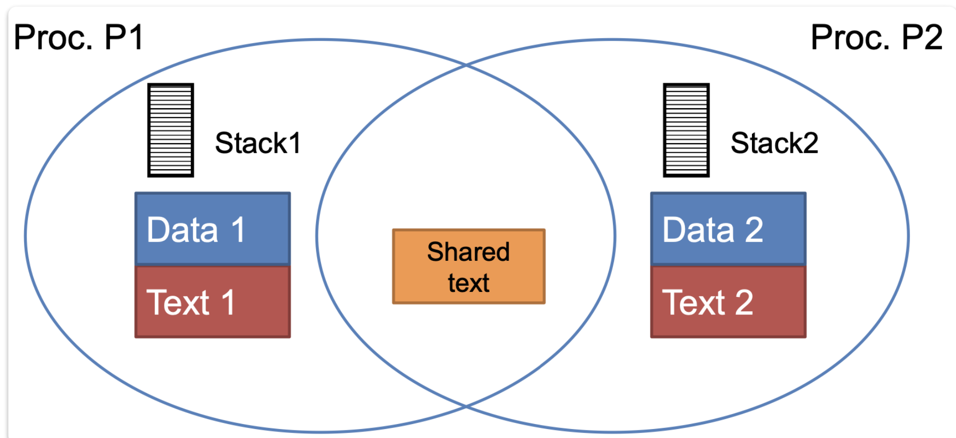
Modello Globale vs Locale

Nel modello *globale*, i processi/thread possono condividere dati attraverso variabili di memoria condivisa.
Nel modello *locale*, invece, non vi è condivisione di dati e la cooperazione avviene attraverso lo scambio esplicito di messaggi.

> *Global*



> *Local*



Sincronizzazione thread

Quando più thread operano sulla stessa memoria condivisa, possono sorgere problemi di inconsistenza e concorrenza. Questa incertezza

può causare comportamenti imprevisti nel programma.

Definizioni

- *Race condition*: il risultato dipende dall'ordine delle operazioni tra i thread
- *Mutua esclusione*: solo un thread può eseguire una particolare operazione alla volta
- *Sezione critica*: blocco di codice eseguito da un singolo thread alla volta
- *Lock*: variabile di sincronizzazione che garantisce la mutua esclusione
  - Acquisizione del lock prima dell'accesso alla sezione critica e rilascio dopo aver completato l'accesso ai dati condivisi
  - Attesa dei thread se il lock è già acquisito

Too much milk problem

Il problema del "too much milk" è un esempio classico di *race condition*. Immagina due coinquilini che condividono un frigorifero e decidono di fare la spesa separatamente per comprare il latte. Entrambi i coinquilini controllano il frigorifero per vedere se c'è abbastanza latte prima di andare a comprarlo.
Il problema sorge quando entrambi vedono che il latte sta finendo, quindi entrambi decidono di comprarne di più. Alla fine, quando entrambi tornano con il latte, scoprono che ora hanno troppo latte perché non si sono coordinati. Questo è il risultato di una mancanza di sincronizzazione tra i due coinquilini, che porta a una situazione indesiderata. La soluzione a questo problema sarebbe l'utilizzo di un meccanismo di mutua esclusione, come ad esempio una *lock* sul frigorifero. In questo modo, un coinquilino acquisirebbe il blocco prima di controllare il latte nel frigorifero e rilascerebbe il blocco solo dopo aver completato l'operazione di spesa. In questo modo, l'altro coinquilino sarebbe bloccato dall'accedere al frigorifero fino a quando il primo non avesse finito, evitando così il problema del "too much milk".

Regole per Lock

- La lock deve essere libera inizialmente.
- *Acquisire* sempre la lock prima di accedere a dati condivisi.
- *Rilasciare* sempre la lock dopo aver finito con i dati condivisi.
- Mai far rilasciare la lock da qualcun altro.
- Mai accedere ai dati condivisi senza aver prima acquisito la lock.

Semantica Mesa VS Hoare

- Semantica Mesa:
  - `Signal` mette il thread in attesa nella lista dei pronti.
  - Chi emette il segnale mantiene il lock e il processore.
- Semantica Hoare:
  - `Signal` assegna il processore e il lock al thread in attesa.
  - Quando il thread in attesa termina, il processore/lock viene restituito a chi ha emesso il segnale.
  - Possibilità di segnali nidificati!

Condition Variables

Le *Condition Variables* consentono di sincronizzare l'accesso a dati condivisi senza dover rimanere in attesa attiva.
Le operazioni principali sono:

- `wait`: rilascia atomicamente la lock e mette in attesa il thread fino a quando non viene segnalato.
- `signal`: sveglia uno dei thread in attesa, se presente.
- `broadcast`: sveglia tutti i thread in attesa, se presenti.

Altre regole:

- Si deve SEMPRE mantenere il lock quando si chiama `wait`, `signal`, `broadcast`.
- Le *condition variables* non mantengono memoria: se si invia un segnale quando nessuno sta aspettando, non accade nulla.
- La funzione `wait` rilascia atomicamente il lock.

- Se si chiama prima `wait` e poi si rilascia il lock, e viceversa, il comportamento è diverso.
- Quando un thread viene svegliato da `wait`, potrebbe non essere eseguito immediatamente.
- `Signal`/`broadcast` mettono il thread nella lista dei pronti.
- Quando il lock viene rilasciato, chiunque potrebbe acquisirlo.
- La funzione `wait` DEVE essere inclusa in un ciclo `while`.
  - Ciò semplifica l'implementazione sia delle variabili di condizione che dei lock.
- Semplifica il codice che utilizza variabili di condizione e lock.

Spinlock

- Una *Spinlock* è un tipo di Lock in cui il processore aspetta in un loop che il lock diventi libero (attesa attiva!).
- Si presume che il lock venga tenuto per un breve periodo.
- Viene utilizzato per proteggere l'elenco dei processi pronti e per implementare i lock.

SpinlockAcquire() {	SpinlockRelease() {
while (TestAndSet(&spinLockValue) == BUSY)	spinLockValue = FREE;
;	memory_barrier();
}	}

Semafori

- Un *semaforo* ha un valore intero non negativo.
- `P()` attende in modo atomico che il valore diventi > 0, quindi lo decrementa. (*Prende*)
- `V()` incrementa in modo atomico il valore. (*Lascia*)
- Le *strutture dati* di un semaforo sono un intero e una coda.
- Le uniche operazioni sono `P` e `V`, e sono atomiche.

Riassunto

- Utilizza una struttura coerente per la sincronizzazione.
- Utilizza sempre blocchi e variabili di condizione.

- Acquisisci sempre il blocco all'inizio della procedura e rilascialo alla fine.
- Tieni sempre il blocco quando utilizzi una variabile di condizione.
- Utilizza sempre un ciclo `while` durante l'attesa.
- Evita sempre l'attesa attiva nella funzione `sleep()`.

....
..
....
..
....
..

Multi-Object Synchronization

Topics

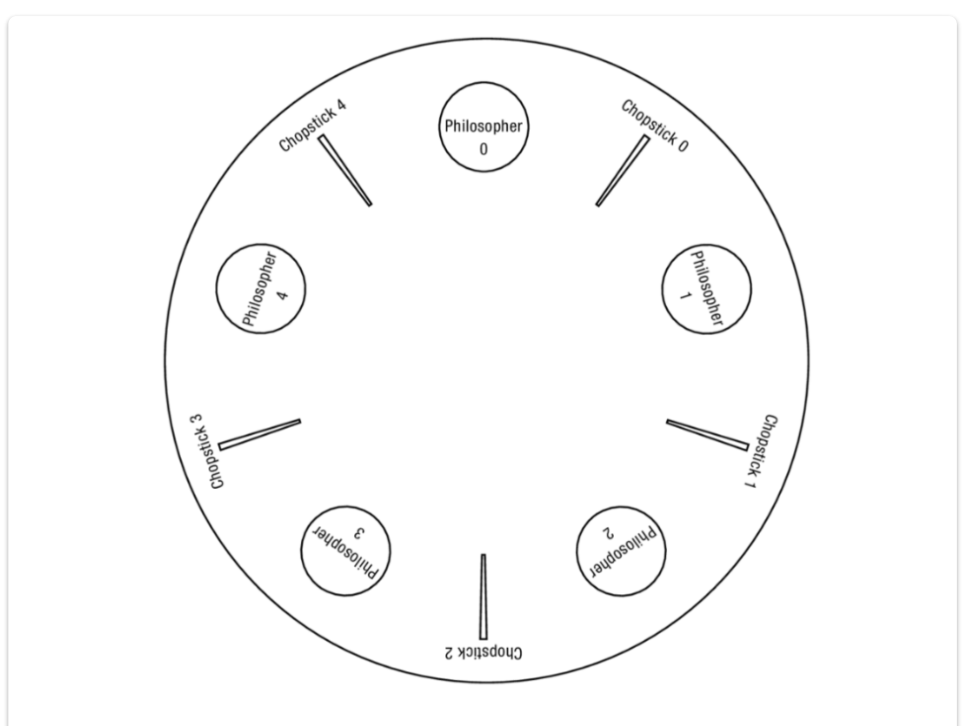
- Definizione di Deadlock.
- Condizioni per la sua comparsa.
- Evitare e risolvere il Deadlock (Algoritmo del Banchiere)

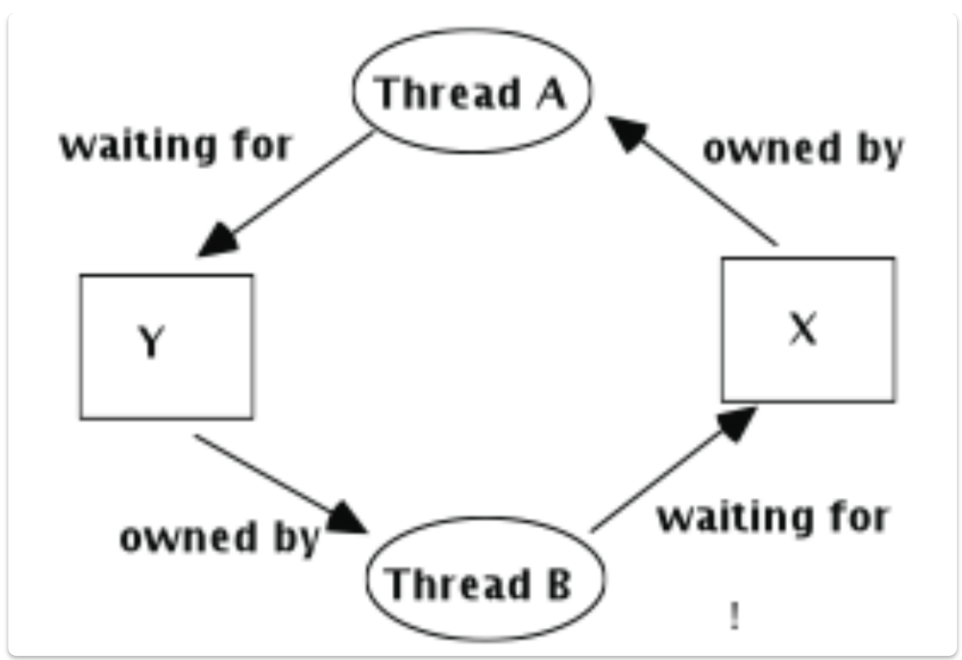
Deadlock

- *Risorsa*: qualsiasi elemento necessario a un thread per svolgere il proprio lavoro (CPU, spazio disco, memoria, blocco).
  - *Preemptable*: può essere tolto dall'OS (e successivamente restituito).
  - *Non-preemptable*: deve essere rilasciato dal thread.
- *Starvation*: il thread attende indefinitamente.
- *Deadlock*: attesa circolare delle risorse.
  - Deadlock ⇒ starvation, ma non viceversa.

Condizioni per Deadlock

- *Accesso limitato alle risorse*:
    - Un numero limitato di thread può utilizzare contemporaneamente una risorsa.
- *Nessuna preemption*
- *Attesa con risorse in mano*:
    - Un thread detiene le risorse assegnate mentre attende un'altra.
- *Catena circolare di richieste*.





Algoritmo del banchiere

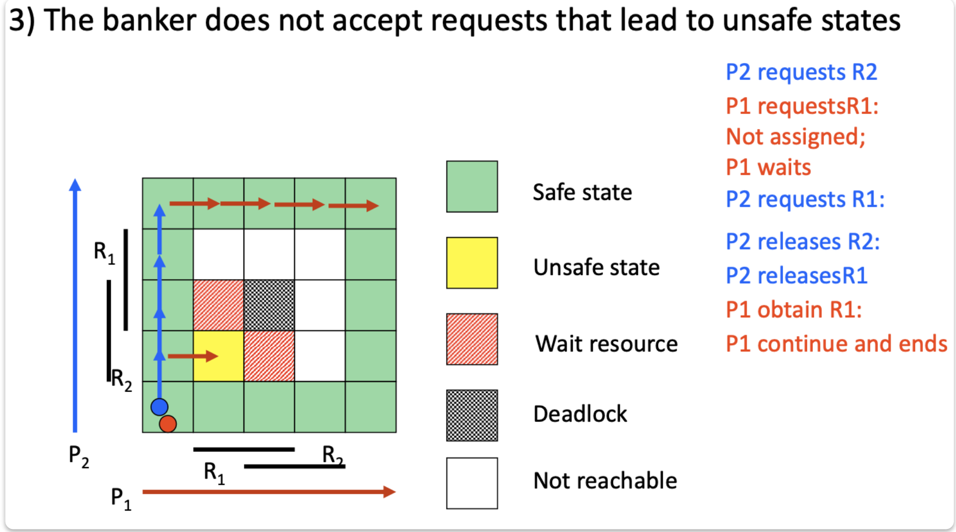
- Stabilisce in anticipo i bisogni massimi di risorse.
- Assegna dinamicamente le risorse quando sono necessarie
  - Attende se il soddisfacimento della richiesta porterebbe al deadlock.
- La richiesta può essere concessa se esiste un ordinamento sequenziale dei thread privo di deadlock.

Definizioni

Stato sicuro:

- Per qualsiasi sequenza possibile di future richieste di risorse, è possibile alla fine concedere tutte le richieste e quindi far terminare correttamente tutti i processi
- Potrebbe richiedere attesa anche quando le risorse sono disponibili!
  Stato non sicuro:
- Alcune sequenze di richieste di risorse possono risultare in un deadlock
  Stato condannato:
- Tutti i possibili calcoli portano al deadlock

Esempio



- Concedi la richiesta solo se il risultato è uno stato sicuro
- La somma delle massime necessità di risorse dei thread correnti può essere maggiore delle risorse totali
  - A condizione che ci sia un modo per far terminare tutti i thread senza finire in un deadlock
- Esempio: procedi solo se
  - numero di risorse libere ⩾ massimo rimanente che potrebbe essere necessario a questo thread per terminare
  - Garantisce che questo thread possa terminare
- Ogni processo dichiara il numero di risorse di cui ha bisogno.
- Alla richiesta di un processo P, il banchiere verifica se l'assegnazione delle risorse mantiene uno stato sicuro. A questo scopo:
  - Considera lo stato S raggiunto se la richiesta viene concessa.
  - Per ogni processo calcola il requisito residuo R (il numero di risorse di cui ha ancora bisogno).
  - Ordina i processi in base al valore crescente di R.
  - Esegue l'algoritmo.
  - Se alla fine tutti i processi sono contrassegnati, lo stato è sicuro.
- Se S è sicuro, la richiesta può essere concessa.
- Altrimenti, il processo P attende fino a quando non ci sono abbastanza risorse per consentirgli di procedere.

••
••
••

Scheduling

Topics

• Politica di pianificazione: cosa fare dopo, quando ci sono più thread pronti a essere eseguiti
• Politiche per un singolo processore
• Politiche per multiprocessori

Definizioni

- *Task/Job*
  - Richiesta dell'utente: es. clic del mouse, richiesta web, comando shell, …
- *Latenza/tempo di risposta*
  - Quanto tempo impiega un compito per essere completato?
- *Throughput*
  - Quanti compiti possono essere eseguiti per unità di tempo?
- *Overhead*
  - Quanto lavoro extra viene svolto dallo scheduler?
- *Fairness*
  - Quanto è uguale la performance ricevuta da diversi utenti?
- *Predictability*
  - Quanto è consistente la performance nel tempo?
- *Workload*
  - Insieme di compiti da eseguire per il sistema
- *Scheduler preemptive*
  - Se possiamo togliere risorse da un compito in esecuzione

- *Work-conserving*
  - La risorsa è utilizzata ogni volta che c'è un compito da eseguire
- *Algoritmo di scheduling*
  - Prende un carico di lavoro in input e decide quali compiti fare prima
  - Metrica di performance (throughput, latenza) in output
  - Considerare solo gli scheduler work-conserving

Politiche per un singolo processore

FIFO

Programma i compiti nell'ordine in cui arrivano

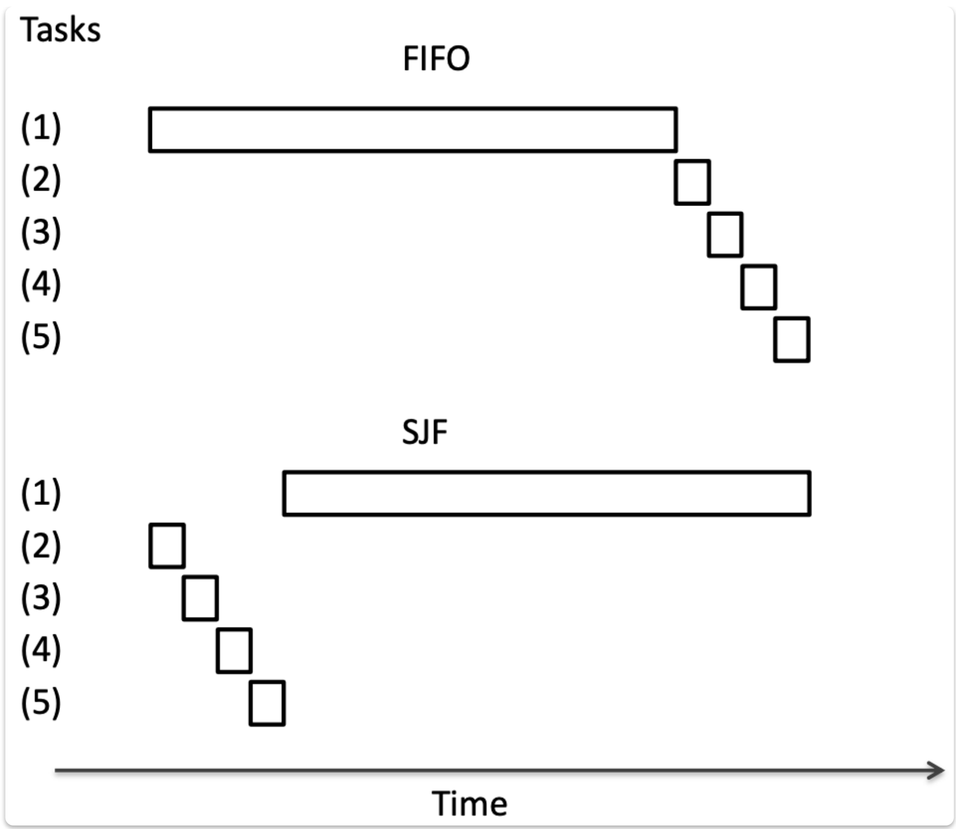
- Li continua ad eseguire finché non sono completati o rilasciano il processore

Shortest Job First

SJF (Shortest Job First) è un'algoritmo di scheduling che può essere non pre-emptive o pre-emptive.

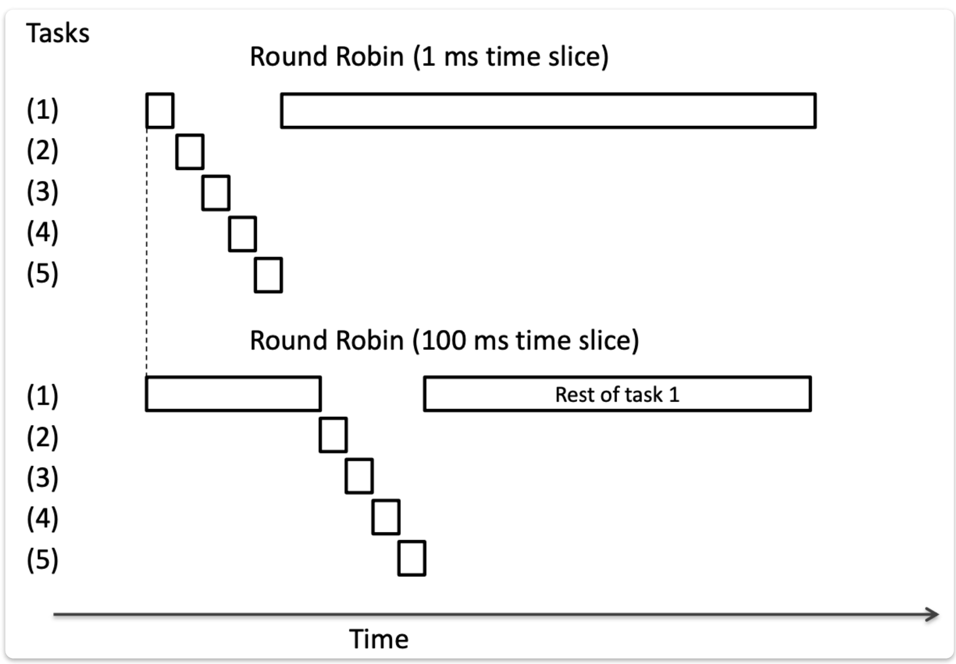
- *Non pre-emptive*: viene eseguito il compito con il minor lavoro rimanente fino al termine o al rilascio del processore.
- *Pre-emptive*: viene eseguito il compito con il minor lavoro rimanente e, se si presenta un compito più breve, viene effettuato uno switch di contesto.

SJF è considerato ottimale per il tempo medio di risposta, ma può causare problemi come la *Starvation* e la *varianza* nel tempo di risposta.



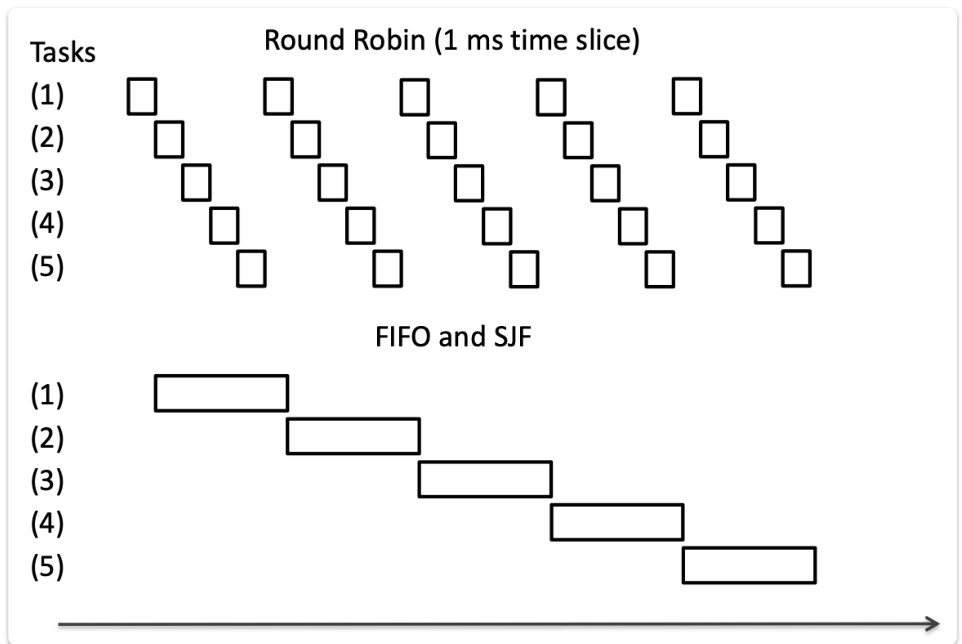
Round Robin

Round Robin è un algoritmo di scheduling in cui ogni compito ottiene una risorsa per un periodo di tempo fisso, chiamato "time quantum". Se il compito non si completa, ritorna in coda per essere eseguito di nuovo.



Garantisce una proporzionalità del tempo trascorso nel sistema rispetto alla lunghezza del compito.

- La fine del tempo è segnalata dal timer.
    - L'interruzione attiva lo scheduler.
    - Lo scheduler riavvia il timer.
- Lo scheduler interviene anche in caso di sospensione del processo in esecuzione.
    - Riassegna la CPU e riavvia il timer.



Max-Min Fairness

Max-Min Fairness è un principio di scheduling che massimizza l'allocazione minima garantita a ciascun compito, bilanciando le risorse tra processi CPU-bound e I/O-bound. Si da priorità ai compiti più piccoli e si distribuiscono equamente le risorse rimanenti per garantire un utilizzo ottimale del sistema.

Multi-level Feedback Queue

Obiettivi:

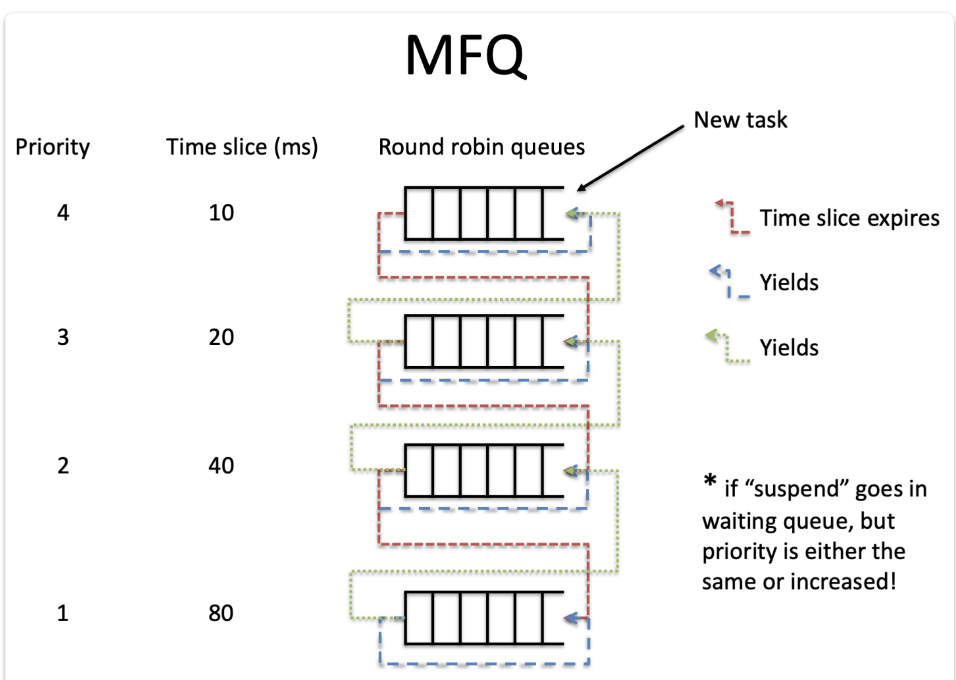
- Reattività
- Basso overhead
- Starvation free

- Alcuni compiti hanno priorità alta/bassa
- Equità (tra compiti della stessa priorità)
  Non perfetto su nessuno di questi fronti!

Struttura

Insieme di code Round Robin

- Ogni coda ha una priorità diversa
- Le code ad alta priorità hanno fette di tempo brevi
- Le code a bassa priorità hanno fette di tempo lunghe
  Lo *scheduler* seleziona il primo thread nella coda di priorità più alta



- I compiti iniziano nella coda di priorità più alta.
- Se la fetta di tempo scade, il compito scende di un livello.
- Se il compito viene sospeso o cede il processore (`yield()`), rimane nel suo livello o viene aumentato se è possibile.

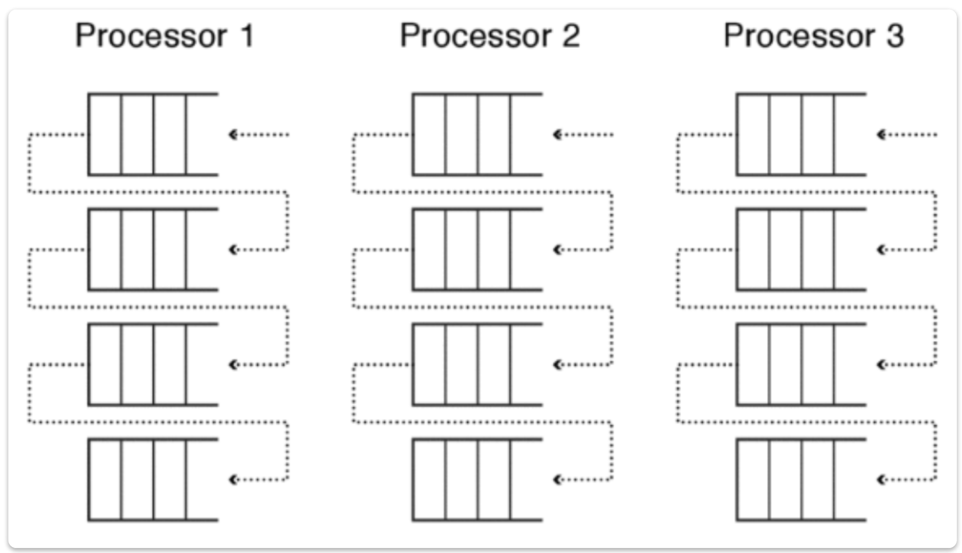
Priorità dinamica: i compiti possono cambiare "abitudini": da legati alla CPU a legati all'I/O e viceversa.

Se le code superiori sono sempre piene di processi legati all'I/O si può verificare *starvation*. Si ha la necessità di politiche per

aumentare la priorità dei processi legati alla CPU che sono in starvation.

Politiche per Multiprocessore

Per-Processor Multi-level Feedback with Affinity Scheduling



- Ogni processore ha la propria *ready-list* (a più livelli)
- I thread vengono rimessi nella *ready-list* dove hanno recentemente eseguito
- I processori inattivi possono rubare lavoro da altri processori
    - Quanti lavori è conveniente rubare? Da chi?
        - Il numero di lavori da rubare dipende dalla disponibilità di risorse sul processore inattivo e dalla priorità dei lavori sul processore attivo.
    - Quando è conveniente riequilibrare il carico di lavoro?
        - Il riequilibrio del carico di lavoro è conveniente quando si verifica un'ineguaglianza significativa tra i carichi di lavoro dei vari processori. Ad esempio, quando un processore è sovraccarico mentre un altro è inattivo, oppure quando ci sono cambiamenti nella natura del carico di lavoro (ad esempio, passaggio da computazioni intensive a operazioni di I/O).
- *Oblivious Scheduling:*

    - Lo scheduler assegna i compiti senza considerare le caratteristiche specifiche dei processori o dei compiti stessi.
    - Ogni compito viene trattato allo stesso modo.
- *Gang Scheduling:*
    - Un insieme di processi correlati o interdipendenti viene eseguito insieme su un gruppo di processori.
    - Adatto per applicazioni parallele o distribuite in cui i processi devono lavorare in modo coordinato o comunicare tra loro durante l'esecuzione.
    - Obiettivo: utilizzo delle risorse e riduzione del tempo di comunicazione tra i processi del gruppo.

....

....

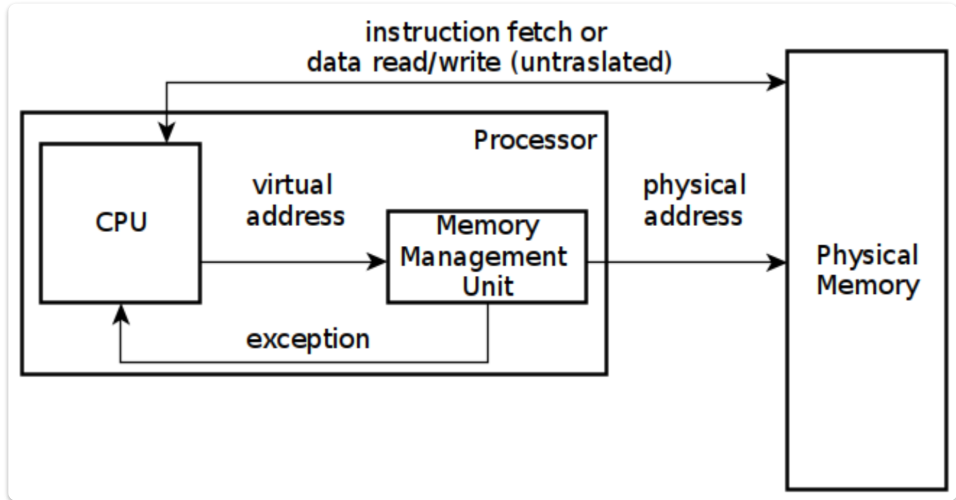
....

Traduzione degli Indirizzi

Topic

- Concetto di Traduzione degli Indirizzi.
- Traduzione degli Indirizzi Flessibile:
    - Base e bound
    - Segmentazione
    - Paginazione
    - Traduzione multilivello
- Traduzione degli Indirizzi Efficiente:
    - Buffer di Traduzione Lookaside (TLB)
    - Cache con indirizzamento virtuale e fisico

Concetto di Traduzione degli Indirizzi



Obiettivi

- Protezione della memoria
- Condivisione della memoria
- Posizionamento flessibile della memoria
- Indirizzi sparsi
- Efficienza nella ricerca durante l'esecuzione
- Tabelle di traduzione compatte
- Portabilità

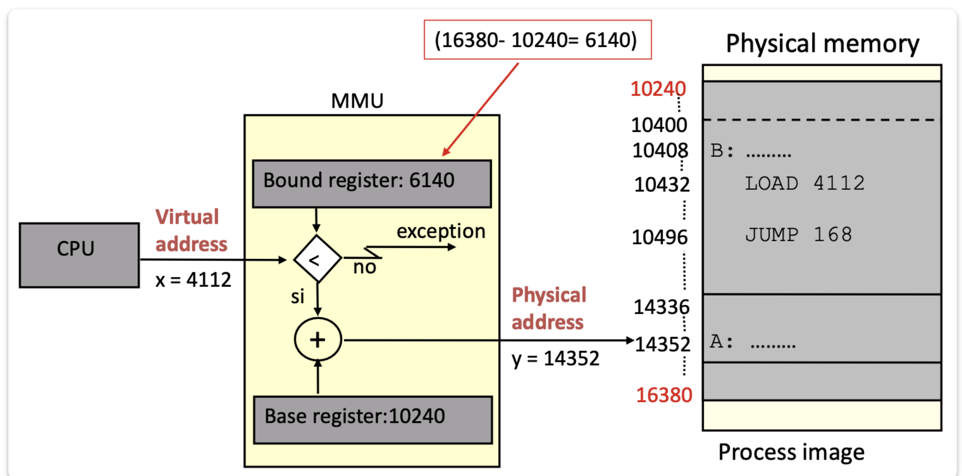
Traduzione degli Indirizzi Flessibile

Caratteristiche

- La traduzione degli indirizzi serve per isolare i processi, consentire la comunicazione efficiente tra di essi e condividere segmenti di codice. Inoltre, facilita l'inizializzazione dei programmi e l'allocazione dinamica della memoria.
- La gestione della cache migliora le prestazioni del sistema, mentre l'I/O senza copia e i file mappati in memoria consentono un accesso diretto ai dati del dispositivo e ai file.
- La traduzione degli indirizzi supporta il checkpointing/riavvio, le strutture dati persistenti, la

migrazione dei processi, il controllo del flusso di informazioni e la memoria condivisa distribuita.

Base and Bounds



Base and Bounds utilizza due registri, il registro *base* e il registro *bound*. Quando un processo tenta di accedere alla memoria, il sistema confronta l'indirizzo virtuale richiesto con il registro base per determinare l'offset all'interno della memoria fisica. Se l'indirizzo virtuale è compreso tra il registro base e la somma del registro base e del registro bound, l'accesso è consentito; altrimenti, viene generata un'eccezione di segmentazione.

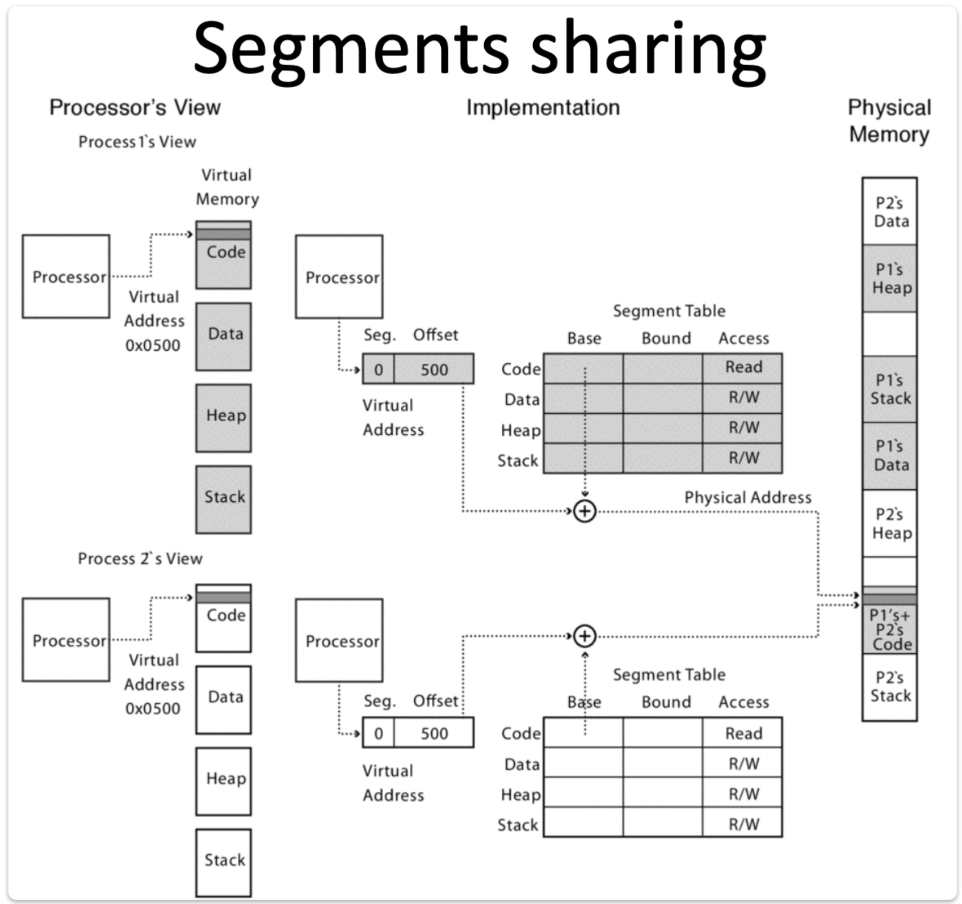
Questo metodo consente di rilocare il processo nella memoria fisica senza modificarlo, garantendo una buona efficienza. Tuttavia, presenta alcune *limitazioni* significative:

- non è in grado di impedire a un programma di sovrascrivere il proprio codice
- non supporta la condivisione di codice o dati tra processi
- non consente di espandere dinamicamente lo stack o l'heap

Segmentazione

- Un *segmento* è una regione contigua di memoria, che può essere virtuale o fisica.
- Ogni processo ha una *tabella dei segmenti*, che consiste in un array di entry di segmento.

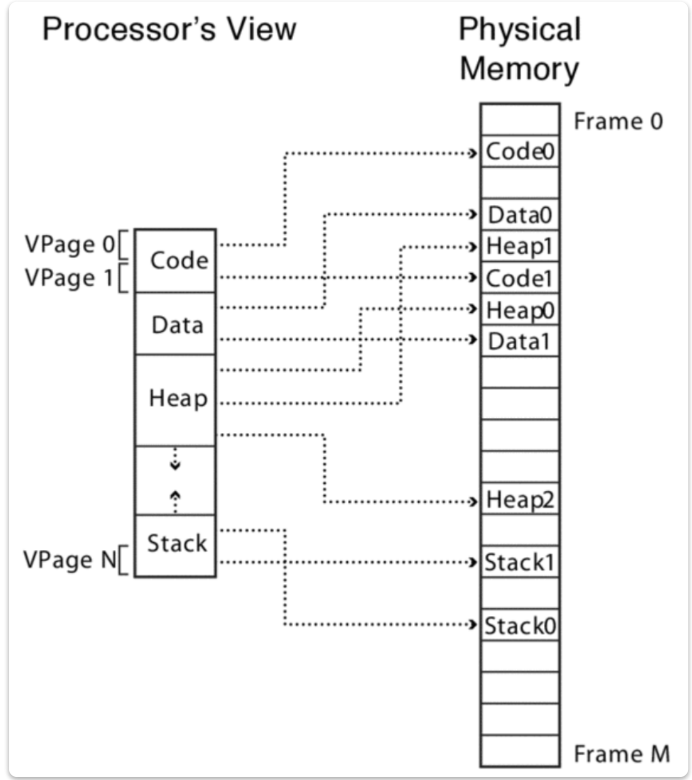
- Una *entry* nella tabella corrisponde a un segmento e contiene una coppia di base e bound.
- Un segmento può essere posizionato in qualsiasi punto della memoria fisica, specificando il punto di inizio, la lunghezza e le autorizzazioni di accesso.
- I processi possono condividere segmenti.

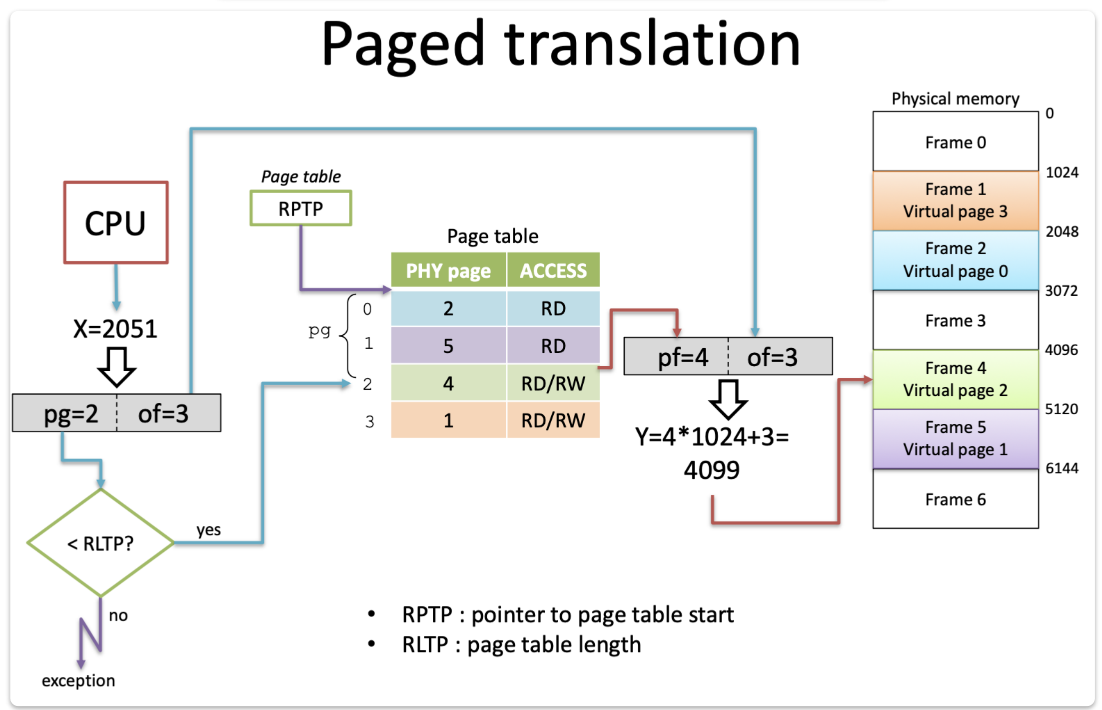


Paginazione

- Gestione della memoria in unità di dimensione fissa.
- Trovare una pagina libera è semplice.
- Ogni processo ha la propria tabella delle pagine.
  - La tabella delle pagine è memorizzata nella memoria fisica. Perché?
    - È utilizzata dal sistema operativo per effettuare la traduzione degli indirizzi virtuali dei processi in indirizzi fisici nella memoria principale.
  - Abbiamo bisogno di due registri:
    - Puntatore all'inizio della tabella delle pagine

- Lunghezza della tabella delle pagine





> *pg: indice nella tabella delle pagine*
> *of: offset*
> *PHY: pagina fisica corrispondente*

Cosa succede se la dimensione della pagina è molto piccola?

- Tabelle delle pagine enormi.

Cosa succede se la dimensione della pagina è molto grande?

- *Frammentazione interna*: se non abbiamo bisogno di tutto lo spazio all'interno di un chunk di dimensione fissa.

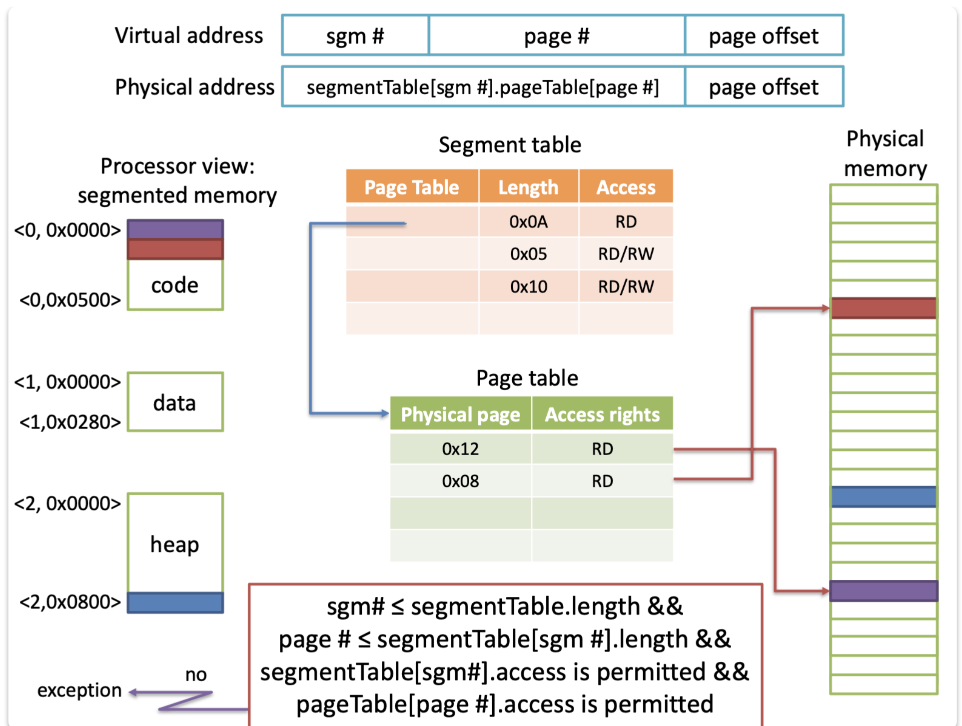
Possiamo condividere memoria tra processi impostando voci nelle loro tabelle delle pagine per puntare agli stessi frame di pagina.

Traduzione multilivello

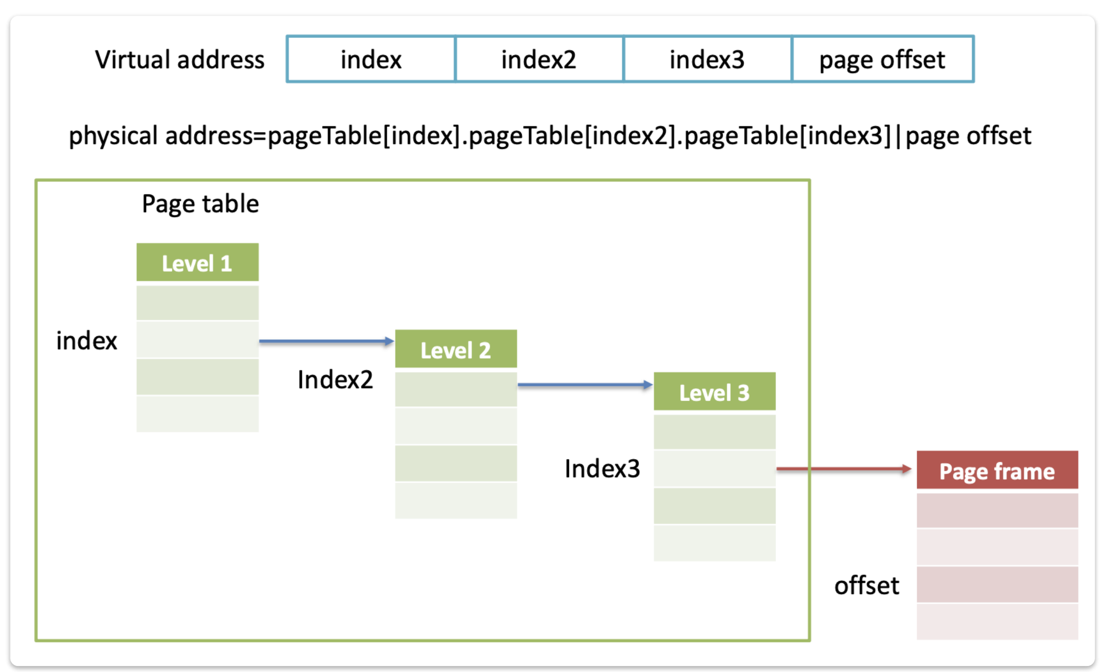
1. Segmentazione paginata
2. Pagine multilivello
3. Segmentazione paginata multilivello

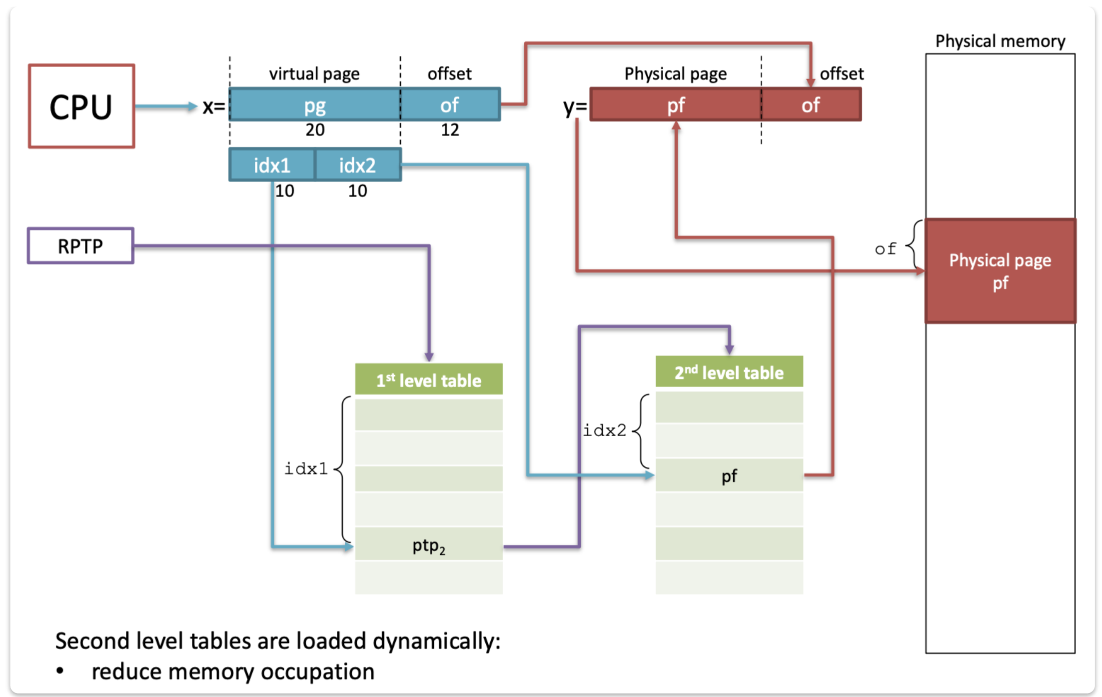
Segmentazione Paginata

- La memoria del processo è segmentata.
- L'ingresso della tabella dei segmenti è:
  - Puntatore alla tabella delle pagine
  - Lunghezza della tabella delle pagine
  - Autorizzazioni di accesso
- L'ingresso della tabella delle pagine è:
  - Frame di pagina
  - Autorizzazioni di accesso
- Condivisione/protezione a livello di pagina o segmento.



Pagine multilivello



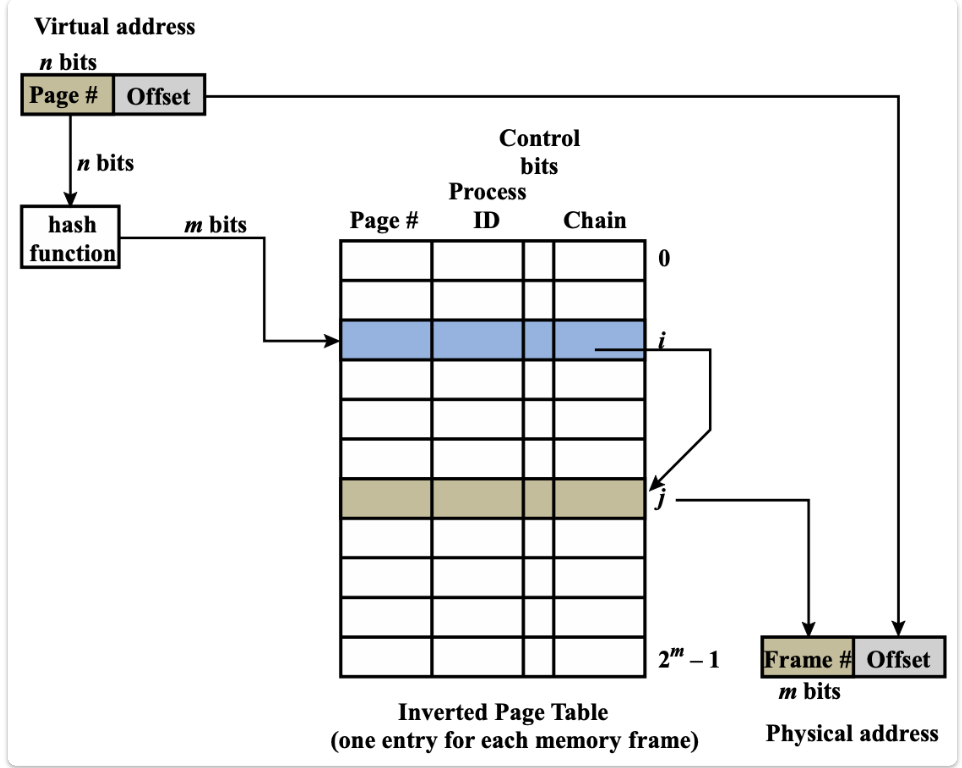


Segmentazione paginata multilivello

La *segmentazione paginata multilivello* è una tecnica di gestione della memoria che combina i concetti di segmentazione e paginazione multilivello
In questa tecnica, la memoria virtuale di un processo è divisa in segmenti, ognuno dei quali è composto da una serie di pagine. Ogni segmento ha la sua tabella dei segmenti, che contiene i descrittori di ogni pagina all'interno del segmento. Ogni descrittore di pagina può puntare a un altro descrittore di pagina e così via per ogni livello che abbiamo fino a puntare alla locazione fisica della pagina in memoria.

Inverted Page Table

- Utilizza una *funzione hash* che mappa dalle pagine virtuali alle pagine fisiche.

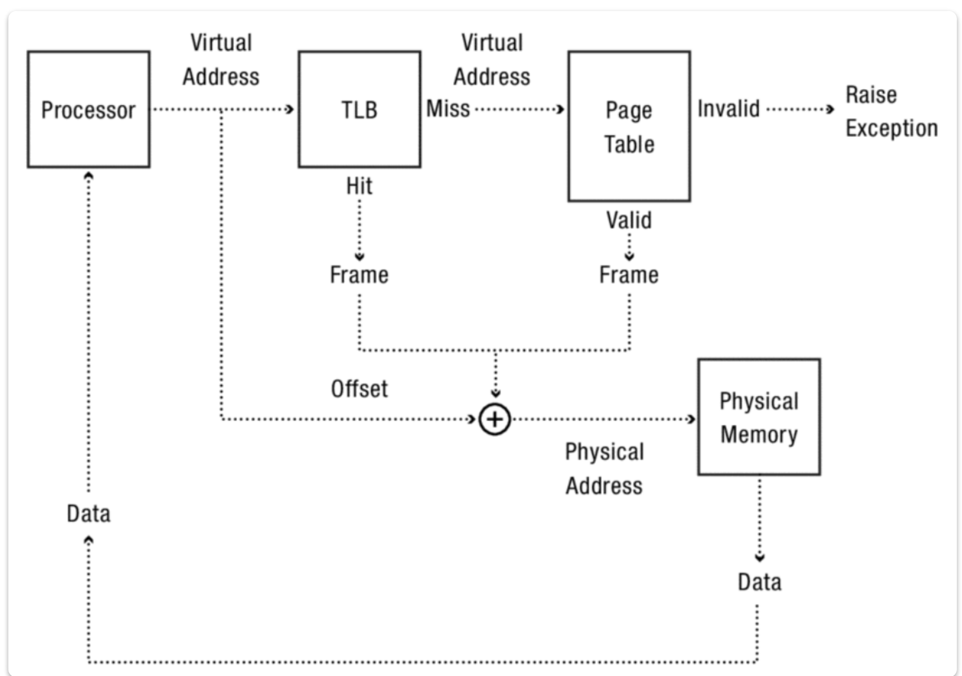


1. Hashing: Quando un processo cerca di accedere a una pagina virtuale, il sistema utilizza una funzione hash per mappare l'indirizzo virtuale della pagina alla sua corrispondente pagina fisica nella memoria principale. Questo processo di hashing produce un indice nella tabella delle pagine invertita.
2. Ricerca: Una volta ottenuto l'indice tramite la funzione hash, il sistema cerca nella tabella delle pagine invertita utilizzando questo indice. La tabella delle pagine invertita è strutturata in modo tale che ogni indice corrisponda direttamente alla pagina fisica associata alla pagina virtuale.
3. Accesso alla memoria fisica: Una volta trovata la corrispondenza nella tabella delle pagine invertita, il sistema può ottenere l'indirizzo fisico della pagina desiderata nella memoria principale. Questo indirizzo viene quindi utilizzato per l'accesso effettivo alla memoria fisica.

Traduzione degli Indirizzi Efficiente

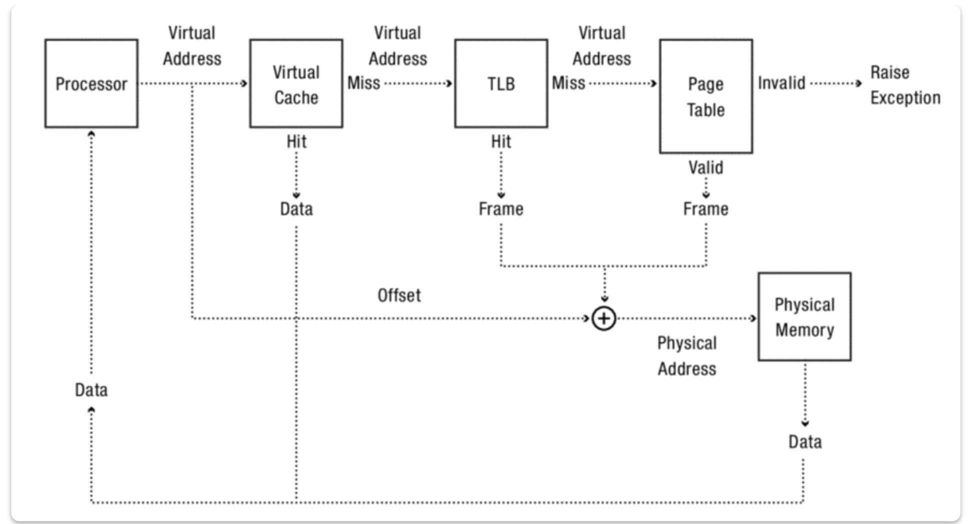
- *Translation Lookaside Buffer* (TLB)

    - È una cache delle traduzioni recenti da pagina virtuale a pagina fisica.
    - Se *cache-hit*, utilizza la traduzione disponibile nell'ingresso del TLB.
    - Se *cache-miss*, si procede con la ricerca nella tabella delle pagine multilivello.
- La *cache TLB* è posizionata nel MMU
- Possiamo avere più livelli di TLB.

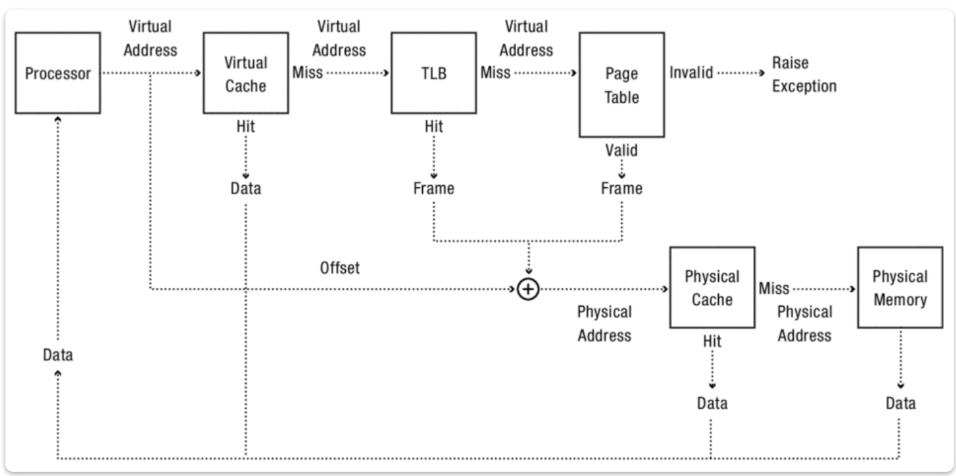


Cache Virtuali VS Fisiche

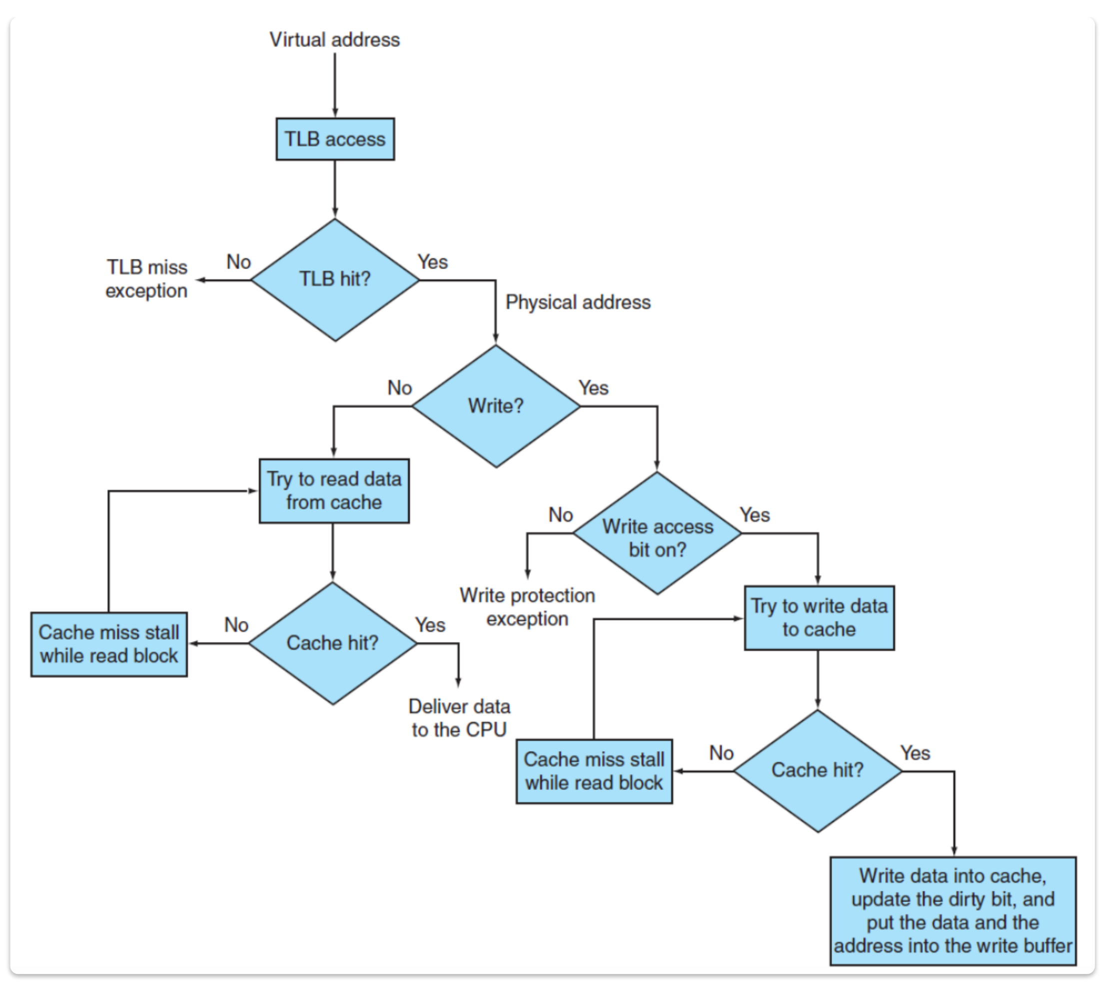
Una *cache virtuale* funziona memorizzando i dati associati agli indirizzi virtuali e fornendo un accesso rapido ai dati senza dover attendere la traduzione degli indirizzi in indirizzi fisici.

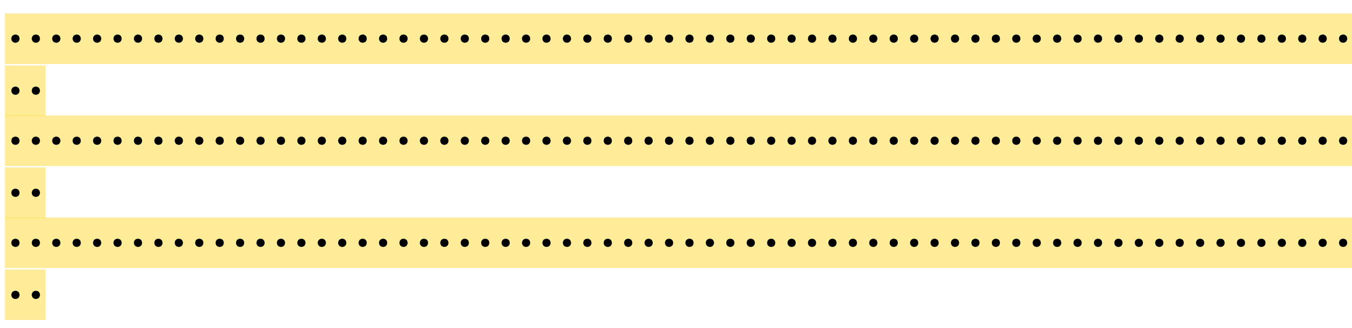


Una *cache fisica* memorizza i dati relativi agli indirizzi fisici dovendo aspettare la traduzione da parte del MMU (TLB) dell'indirizzo prima di poter accedere al dato in caso di hit



Schema riassuntivo interazioni





Page Caching

Topic

- *Memoria virtuale richiesta*: carica in memoria solo le pagine di dati necessarie per l'esecuzione dei processi.
- *Politiche di sostituzione della cache*: determinano quale blocco di dati sostituire quando la cache è piena e viene

richiesto un nuovo blocco di dati. Le politiche comuni includono FIFO, MIN, LRU, LFU e Clock.
- *File mappati in memoria*: Consentono ai processi di accedere ai dati di un file come se fossero memorizzati in memoria.

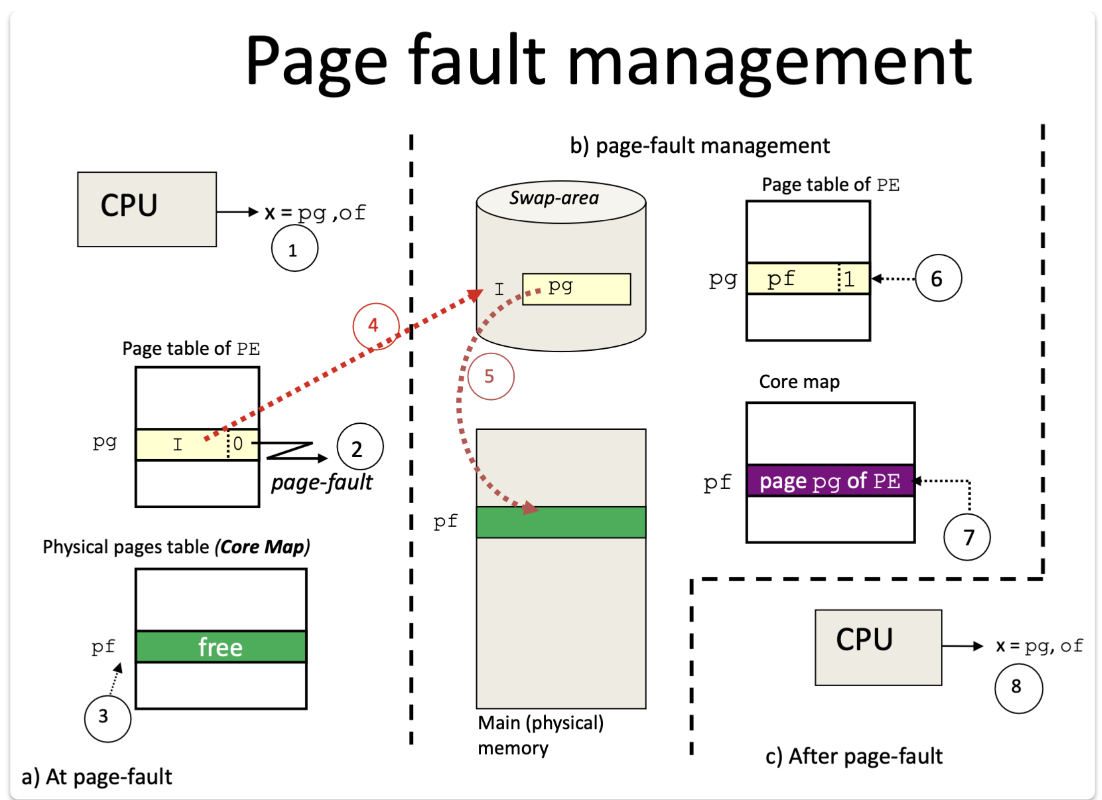
Memoria virtuale richiesta

- La memoria virtuale fornisce alle applicazioni un'astrazione di più memoria di quella fisicamente disponibile.
- La memoria principale è trattata come una cache per i dati memorizzati su disco.
- Le pagine non presenti in memoria vengono caricate su richiesta durante l'accesso, tramite il *demand paging*.

La tabella delle pagine contiene un descrittore per ciascuna pagina. Oltre alle informazioni necessarie per la traduzione degli indirizzi, il descrittore contiene alcune flag:

- `R`, `W`: diritti di accesso in read/write
- `M`, `U`: bit di modified/use
- `P`: bit di presenza (la pagina è invalida se non è presente nella memoria principale)

> ➢ *P = 1: page in main memory*
> ➢ *P = 0: page not in main memory ⟶ Page-fault*



Spiegazione Immagine

1. Richiesta di una pagina da parte del processo
2. Il processo tenta di accedere a una pagina di memoria che non è presente nella memoria fisica.
3. La CPU genera un errore di pagina e interrompe il processo.
4. Il sistema operativo salva l'indirizzo di memoria virtuale a cui il processo stava tentando di accedere (x = pg,of) nella tabella delle pagine del processo.
5. Il sistema operativo seleziona una pagina di memoria libera nella memoria principale.
6. Il sistema operativo carica la pagina di memoria dalla swap-area alla pagina di memoria libera.
7. Il sistema operativo aggiorna la tabella delle pagine del processo per indicare che la pagina di memoria a cui il processo stava tentando di accedere è ora presente nella memoria fisica.
8. Il sistema operativo riavvia il processo e gli consente di continuare a eseguire il codice.

Allocazione di un frame di pagina

- Selezionare la vecchia pagina da eliminare.
- Trovare tutte le voci della tabella delle pagine che si riferiscono alla vecchia pagina.
  - Se il frame di pagina è condiviso:
    - Impostare ogni voce della tabella delle pagine su invalida.
    - Rimuovere eventuali voci del TLB che copiano l'entry della tabella delle pagine ora non valida.
- Scrivere le modifiche alla pagina su disco se la pagina è stata modificata.

Problema:

- Se il sistema operativo sposta una pagina su disco (swap) proprio prima che sia pronta per essere utilizzata, sarà necessario recuperare quella pagina quasi immediatamente.

Thrashing:

- Il sistema passa la maggior parte del tempo a spostare pagine su disco anziché eseguire le istruzioni dell'applicazione.

Working Set:

- Insieme di posizioni di memoria che devono essere memorizzate nella cache per ottenere un cache-hit ragionevole.
- Il *thrashing* si verifica quando il sistema dispone di una cache troppo piccola.

Politiche di sostituzione della cache

Obbiettivo: ridurre cache-miss
Le politiche comuni includono FIFO, MIN, LRU, LFU e Clock.

FIFO

Rimpiazza la pagina che sta in cache da più tempo

reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

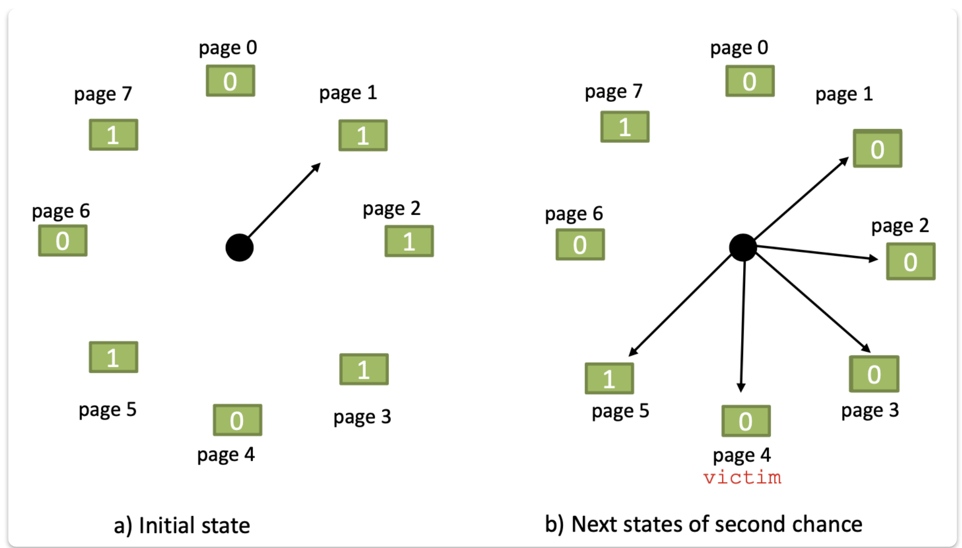
MIN, LRU, NRU, LFU

- *MIN* (Ideal, Optimal) ESERCIZIO
  - Sostituisci la pagina che non sarà utilizzata per il tempo più lungo nel futuro.
- *Least Recently Used* (LRU) ESERCIZIO
  - Sostituisci la pagina che non è stata utilizzata per il tempo più lungo nel passato.
- *Not Recently Used* (NRU)
  - Sostituisci una delle pagine che non sono state utilizzate di recente.
- *Least Frequently Used* (LFU) ESERCIZIO
  - Sostituisci l'entrata della cache utilizzata meno frequentemente.
  - Esempi: *second chance*, *working set algorithm*.

Clock

Algoritmo dell'Orologio (Clock Algorithm):

- Periodicamente, scansiona tutte le pagine.
- Se una pagina non è stata utilizzata, viene sostituita.
- Se una pagina è stata utilizzata, viene contrassegnata come non utilizzata.



Spiegazione Immagine

1. L'algoritmo mantiene una struttura dati a forma di anello che contiene le pagine presenti nella cache.
2. Ogni pagina nella struttura dati ha associato un bit aggiuntivo chiamato "bit di riferimento" (R bit).
3. Durante la scansione periodica delle pagine, l'algoritmo esamina ogni pagina nella struttura dati:
   - Se il bit di riferimento di una pagina è impostato a 0, significa che la pagina non è stata utilizzata dalla scansione precedente e può essere candidata per la sostituzione.
   - Se il bit di riferimento di una pagina è impostato a 1, significa che la pagina è stata utilizzata dalla scansione precedente. In questo caso, l'algoritmo imposta il bit di riferimento a 0 e passa alla pagina successiva.
4. Se durante la scansione non viene trovata alcuna pagina con bit di riferimento a 0, l'algoritmo esegue un secondo ciclo di scansione.
5. Questo processo continua fino a quando viene individuata una pagina con bit di riferimento a 0, che viene quindi selezionata per la sostituzione.

Local and global page replacement

La sostituzione delle pagine locale e globale sono due approcci utilizzati per selezionare quale pagina rimuovere dalla memoria

quando si verifica un page-fault:

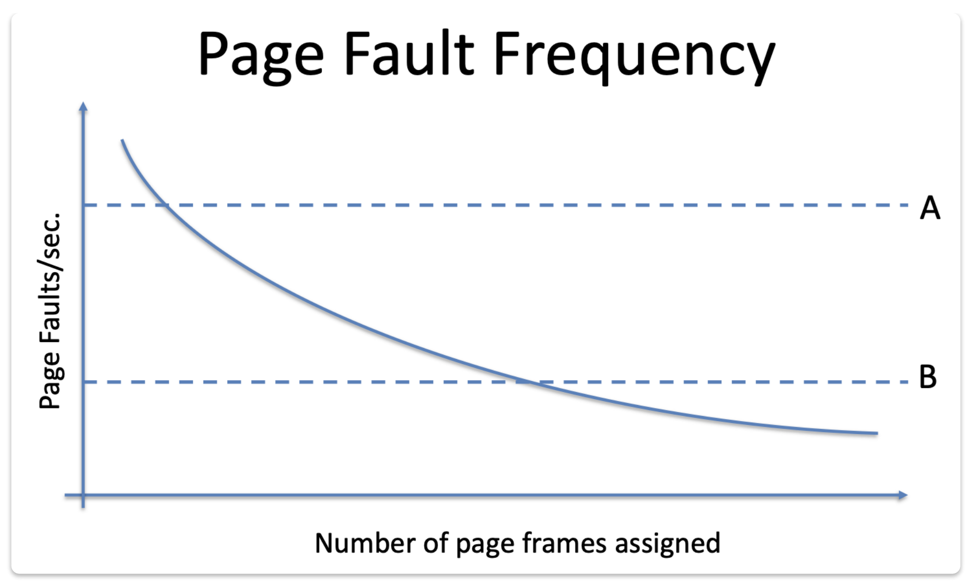
- *Sostituzione globale delle pagine:*
    - La pagina selezionata per la rimozione è scelta tra tutte le pagine nella memoria principale.
    - Può portare a situazioni in cui i processi più lenti sono influenzati negativamente, potenzialmente causando il thrashing.
- Sostituzione locale delle pagine:
    - La pagina selezionata per la rimozione appartiene al processo che ha causato il page fault.
    - Garantisce equità tra i processi, beneficiando in particolare i processi più lenti riducendo la probabilità di thrashing.

Working set algorithm

Working set algorithm definisce il set di pagine referenziate nell'ultimo periodo T, dove T è un parametro dell'algoritmo.
Per ogni pagina:

- *bit R*: indica se la pagina è stata referenziata nell'ultimo intervallo di tempo.
- *TLR*: stima del tempo dall'ultima referenza alla pagina.
    - Alla fine di ogni intervallo di tempo, il bit R per ogni pagina viene azzerato e si aggiorna l'approssimazione del tempo dell'ultima referenza.
- *Età*: definita come la differenza tra il tempo corrente e il tempo dell'ultima referenza della pagina.

ESERCIZIO



Paging on demand:

- Inizialmente nessuna pagina del processo è caricata in memoria
- Le pagine vengono caricate dal processo generando page fault
  - Inizialmente il numero di page fault è alto
- Quando il working set è stato caricato, il numero di page fault diminuisce
  Prepaging:
- Un nuovo processo diventa pronto quando tutte le sue pagine nel working set sono caricate in memoria principale
- Bisogna conoscere (o prevedere) quali pagine saranno nel working set inizialmente
- Non è facile, può essere fatto per alcune pagine

File mappati in memoria

Vantaggi:

- Semplicità di programmazione, specialmente per file di grandi dimensioni
  - Operare direttamente sul file, anziché copiarlo dentro/fuori
- I/O senza copia
  - Dati portati dal disco direttamente nel frame della pagina
- Pipelining

    - Il processo può iniziare a lavorare prima che tutte le pagine siano popolate
- Comunicazione tra processi
    - Segmento di memoria condiviso rispetto a file temporaneo

---

File Systems

Topic

- Astrazioni utili sopra i dispositivi fisici
- Pattern di utilizzo
- Layout del file
- Layout della directory

Astrazione del File System

Persistenza dei dati:

- I crash del sistema operativo lasciano il sistema di file in uno stato valido
  Nominazione:
- Dati nominati invece dei numeri dei blocchi del disco
- Directory invece di storage piatto
  Prestazioni:
- Dati in cache
- Organizzazione del posizionamento dei dati e della struttura dei dati

Accesso controllato ai dati condivisi

L'astrazione del file system include concetti come:

- *Directory*: Gruppi di file o subdirectory, con una mappatura dal nome del file alla sua posizione nei metadati.
- *Path*: Stringhe che identificano univocamente un file o una directory.
- *Link*: Possono essere hard o soft.
  - *Hard link*: voci di directory che collegano due nomi diversi allo stesso file tramite lo stesso *inode*.
  - *Soft link*: collegano nomi ad altri nomi di file.
- *Mount*: associa un nome in un file system alla radice di un altro.

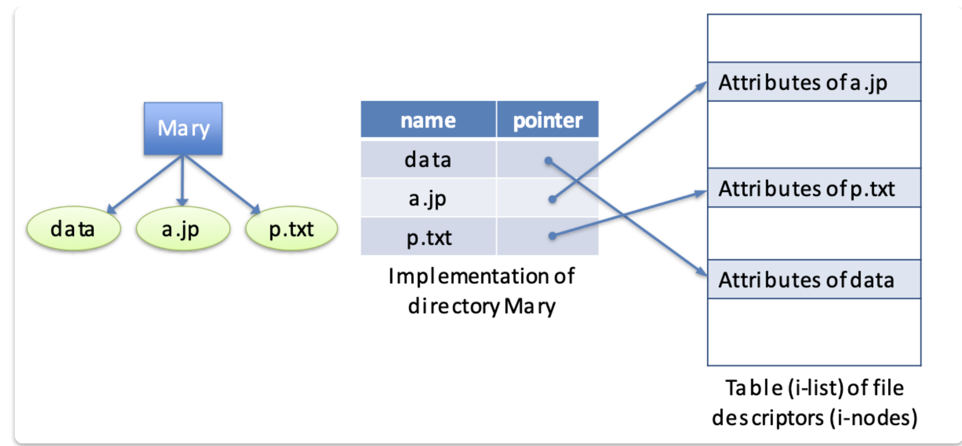
INode

Un *inode* (index node) è una struttura dati in un Fyle System che contiene informazioni dettagliate su un file o una directory, ad eccezione del nome del file stesso e del contenuto effettivo dei dati.
Tra i metadati memorizzati in un inode ci sono:

- permessi di accesso
- il tipo di file
- la dimensione del file
- i timestamp di creazione
- accesso e modifica
- puntatori ai blocchi di dati che compongono il file o la directory.

Directory

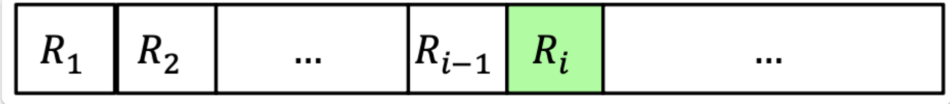
Una *directory* è una struttura dati che collega i nomi dei file agli attributi dei file stessi. Una directory può essere realizzata come una tabella che associa ciascun nome di file al suo descrittore di file, contenente tutti gli attributi del file.



Accesso al file

I metodi di accesso ai file possono essere di due tipi:

- *Sequenziale*: I dati vengono letti/scritti in ordine, uno dopo l'altro.
  - In un file, una sequenza di record logici [R1, R2,.. RN], per accedere a ciascun record logico Ri è necessario prima accedere a tutti i record precedenti (i - 1):



- *Diretto*: Possibile accedere direttamente a una posizione specifica del file.
  - Il file è un insieme [R1, R2,.. RN] di record logici. L'utente può accedere direttamente a ciascun record logico. Le operazioni di accesso includono `read(f, i, &V)`, che legge il record logico `i` del file `f` e memorizza i dati letti nel buffer `V`, e `write(f, i, &V)`, che scrive il contenuto del buffer `V` sul record logico `i` del file `f`.

File System Workload

- La maggior parte dei file nel sistema sono di piccole dimensioni ma i file più grandi possono occupare la maggior parte dello spazio di archiviazione.
- Gli accessi si concentrano principalmente sui file più piccoli, mentre i file più grandi tendono ad avere meno

accessi ma comportano una maggiore quantità di I/O in termini di byte.

- La maggior parte dei file viene acceduta in *modo sequenziale*, ma ci sono anche file che vengono letti e scritti in *modo casuale*, come i file di database.
- Alcuni file hanno una dimensione predefinita alla creazione, mentre altri crescono nel tempo. Per ottimizzare l'efficienza di memorizzazione e l'accesso ai file, è consigliabile utilizzare blocchi di dimensioni appropriate in base alle dimensioni dei file e ai pattern di accesso.

file name —directory→ *file number offset* —Index structure→ *storage block*

FAT

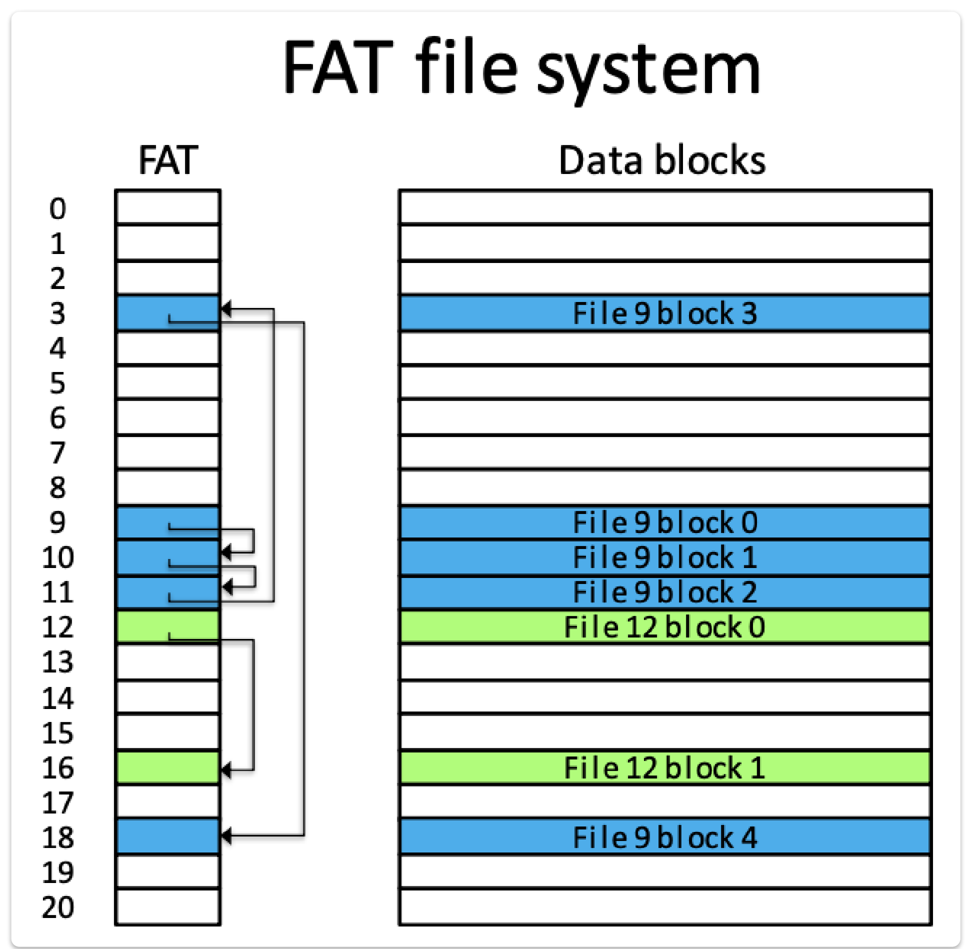
Il *File Allocation Table* (FAT) è una struttura di dati utilizzata nei file-system, in particolare nei sistemi di file FAT e FAT32 (che supporta 2^28 blocchi e file di dimensione pari a 2^32-1), per tenere traccia dell'allocazione dello spazio su disco. La *FAT* è essenzialmente una tabella che mappa i numeri dei blocchi del disco con i file e le directory che li utilizzano.

La FAT è memorizzata all'inizio del disco e consiste in una serie di entry che rappresentano uno specifico cluster sul disco. Ogni entry nella tabella contiene informazioni sull'uso dello spazio disco associato al cluster corrispondente.

Le principali *funzioni* della FAT includono:

1. *Allocazione dello spazio su disco*: La FAT tiene traccia dei cluster liberi e assegnati. Quando un file viene scritto sul disco, la FAT viene aggiornata per segnalare quali cluster sono utilizzati dal file.
2. *Indicizzazione dei file*: La FAT fornisce un meccanismo per seguire i cluster di un file. Ogni file inizia con un numero di cluster iniziale, e la FAT viene consultata per trovare i successivi cluster del file.

3. *Rimozione dei file*: Quando un file viene eliminato, la FAT viene aggiornata per segnalare che i cluster precedentemente utilizzati dal file sono nuovamente disponibili per l'allocazione.



> *Disco di 20 blocchi.*
> *Abbiamo 2 file: File 9 e File12 (nomi interni). Il File9 allocato in 9,10,11,3,18 File12 allocato nei blocchi 12,16.*
> *Nella FAT, nella posizione 9 c'e' il puntatore al blocco successivo (10). In 18 c'e' un valore speciale per dire EOF.*
> *Blocchi liberi sono marcati con 0 nella FAT*

Il file system FAT ha limiti sulla dimensione massima dei file e delle partizioni.
Se la *lunghezza degli elementi* della FAT è L bit e la *dimensione dei blocchi* del disco è B byte:

- Numero massimo di blocchi indirizzabili è 2^L (la capacità del disco o della partizione).

- Massima estensione del file system è quindi 2^L blocchi, corrispondente a $B * 2^L$ byte.
- Se ogni elemento occupa N byte, la FAT occupa $N * 2^L$ byte.

Limitazioni dei file systems FAT

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

- Massima dimensione del File System per diverse ampiezze dei blocchi

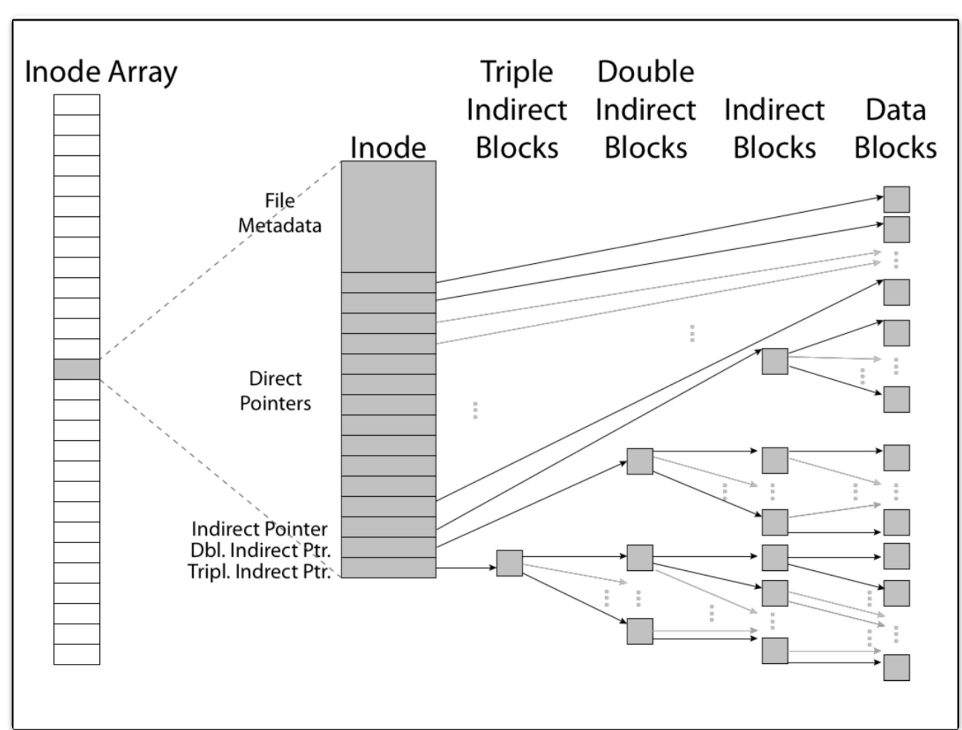
FFS

Il Fast File System utilizza una indicizzazione multi-livello ed un i-node per ogni file
I-list formato da un record chiamato i-node. Mi permette di caricare in memoria solo gli i-node dei file che sono aperti e non come nella FAT che la devo caricare tutta.

1. Struttura a blocchi: I dati sono organizzati in blocchi di dimensioni fisse (4KB). Questo consente un accesso efficiente ai dati, specialmente per file di grandi dimensioni.
2. Indicizzazione migliorata: Algoritmo di indicizzazione migliorato. Tempi di accesso più rapidi.
3. Riduzione della frammentazione: Implementa tecniche per ridurre la frammentazione dei file, consentendo una migliore utilizzazione dello spazio su disco.
4. Journaling: Registra le modifiche in un "journal" prima di applicarle effettivamente al file-system. Questo aiuta a garantire l'integrità del file system anche in caso di crash improvviso o spegnimento non corretto del sistema.

I metadati di un file includono informazioni come il proprietario, i permessi di accesso e i tempi di accesso.
Il file system utilizza un insieme di puntatori per indicizzare i blocchi di dati del file, compresi puntatori diretti, indiretti, doppi e tripli, consentendo di gestire file di varie dimensioni in modo efficiente.

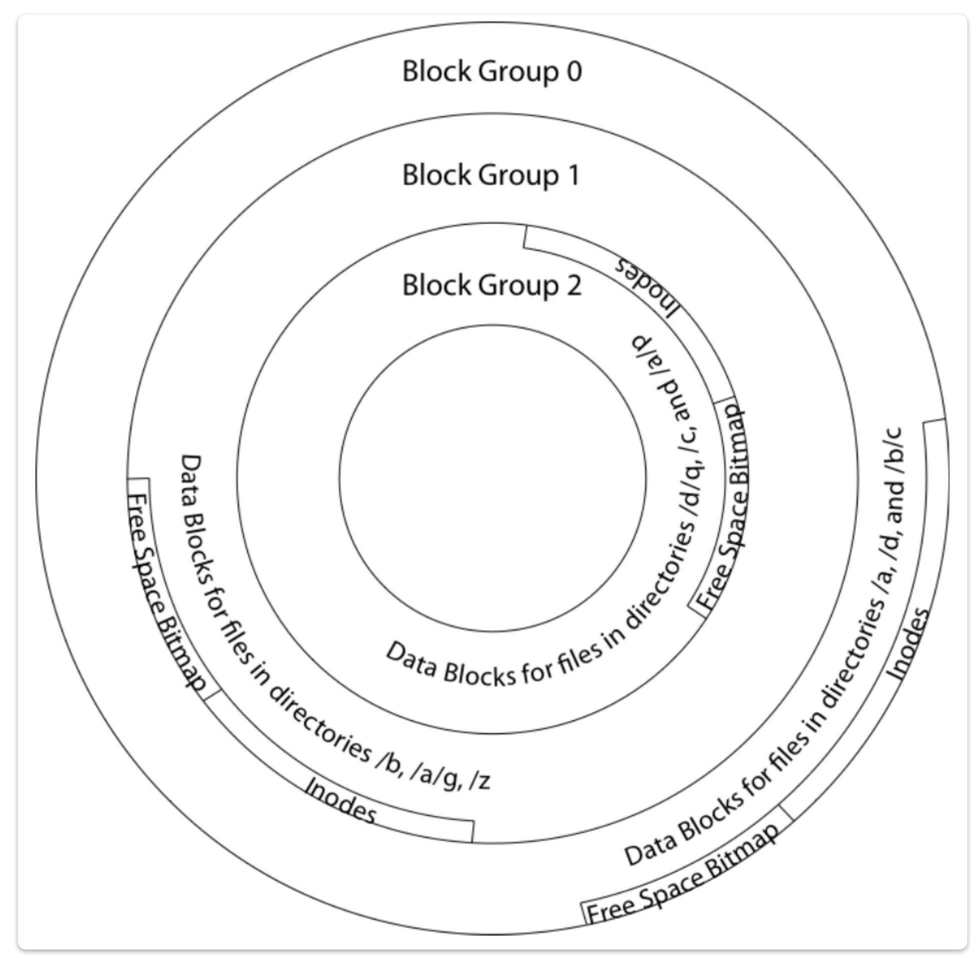
- 12 *puntatori diretti* ai dati (32 bit), *blocchi* da 4 KB permettono di indicizzare fino a 48 KB di dati.
- Un *puntatore indiretto*, che punta a blocchi di puntatori ai dati. Con blocchi da 4 KB, questo consente di indicizzare fino a un massimo di 4 MB di dati.
- Un *puntatore indiretto doppio*, che punta a blocchi indiretti. Consente di indicizzare fino a un massimo di 4 GB di dati, +4 MB del blocco indiretto singolo +48 KB dei puntatori diretti.
- Un *puntatore indiretto triplo*, che punta a blocchi indiretti doppi. Consente di indicizzare fino a un massimo di 4 TB di dati +4 GB del blocco indiretto doppio +4 MB del blocco indiretto singolo +48 KB dei puntatori diretti.



I blocchi del disco sono suddivisi in gruppi.
Un gruppo di blocchi rappresenta un insieme di settori fisicamente contigui sul disco. Quando un file viene memorizzato, viene

allocato in un gruppo di blocchi specifico. I presenti nella stessa directory sono generalmente memorizzati nello stesso gruppo di blocchi.

FFS mantiene uno spazio riservato sul disco per migliorare ulteriormente la località. Questo spazio riservato viene utilizzato per allocare nuovi file in modo che possano crescere in modo contiguo, riducendo così la frammentazione del disco.



> *Ogni gruppo raggruppa cilindri del disco vicini tra loro. FFS cerca di allocare i file di una directory nello stesso gruppo. Subdirectory un po' sparpagliate per rendere più omogenea la distribuzione dell'uso dei blocchi.*

NTFS

NTFS fonde il concetto di i-node e di blocco per file molto piccoli.
Master File Table equivalente delle i-list.
MFT e' essa stessa un file che può essere allocata dovunque,

questo permette di mettere la struttura in zone del disco che non sono danneggiate.

Journaling log delle operazioni per fare ripristini veloci.

In NTFS, ogni file è rappresentato da un albero a profondità variabile. Gli alberi più profondi sono necessari solo se il file diventa fortemente frammentato. Le radici di questi alberi sono memorizzate in una MFT simile all'array di i-node di FFS.
MFT memorizza un array di record MFT da 1 KB, ognuno dei quali contiene una sequenza di record attributo di dimensione variabile. I primi 16 blocchi sono riservati. Il descrittore dell'MFT si trova nel record 0 (che è duplicato nel record 1). L'entry 5 è la root directory del file system. Per trovare l'MFT, viene consultato il superblocco del disco (blocco 0 o 1 che contiene il puntatore al file che contiene l'MFT). Il blocco 16 contiene il descrittore del primo file.

....
....
....

Storage Systems

Topic

- Magnetic disks
- Flash memory
- RAID storage

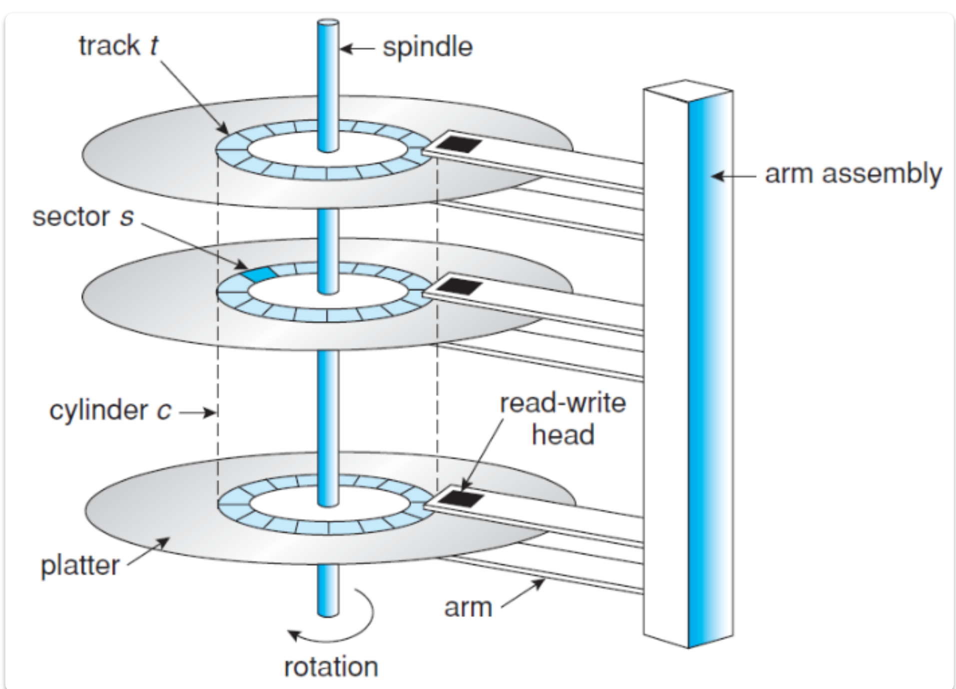
Magnetic disk

Dischi magnetici:

- Archiviazione che raramente si corrompe
- Ampia capacità a basso costo
- Accesso casuale a livello di blocco
- Prestazioni lente per accessi casuali

  Memoria flash:
- Archiviazione che raramente si corrompe
- Buona capacità a costo intermedio (2 volte quella dei dischi)
- Accesso casuale a livello di blocco
- Buone prestazioni per le letture; peggiori per le scritture casuali

Composizione



- 2 facce per disco
- *Track*: corona circolare.
  - Separate da regioni di protezione non utilizzate
    - Riduce la probabilità che le tracce adiacenti siano corrotte durante le scritture (ancora una piccola possibilità non nulla)
  - La lunghezza delle tracce varia lungo il disco
    - All'esterno: più settori per traccia, maggiore larghezza di banda

        - Il disco è organizzato in regioni di tracce con lo stesso numero di settori per traccia
        - Viene utilizzata solo la metà esterna del raggio
- La maggior parte dell'area del disco è nelle regioni esterne del disco.
- *Cilindri*: Insieme delle tracce alla stessa distanza.
- Ogni *track* e' divisa in *Settori*.
    - Risparmio di settori
        - Riassegna settori danneggiati in modo trasparente a settori di riserva sulla stessa superficie
    - Risparmio di scorrimento
        - Riassegna tutti i settori (quando c'è un settore danneggiato) per preservare il comportamento sequenziale
    - Sfalsamento della traccia
        - I numeri dei settori sono sfalsati da una traccia all'altra, per consentire lo spostamento della testina del disco per operazioni sequenziali
- Con una rotazione completa leggo N tracce.

Settori e blocchi

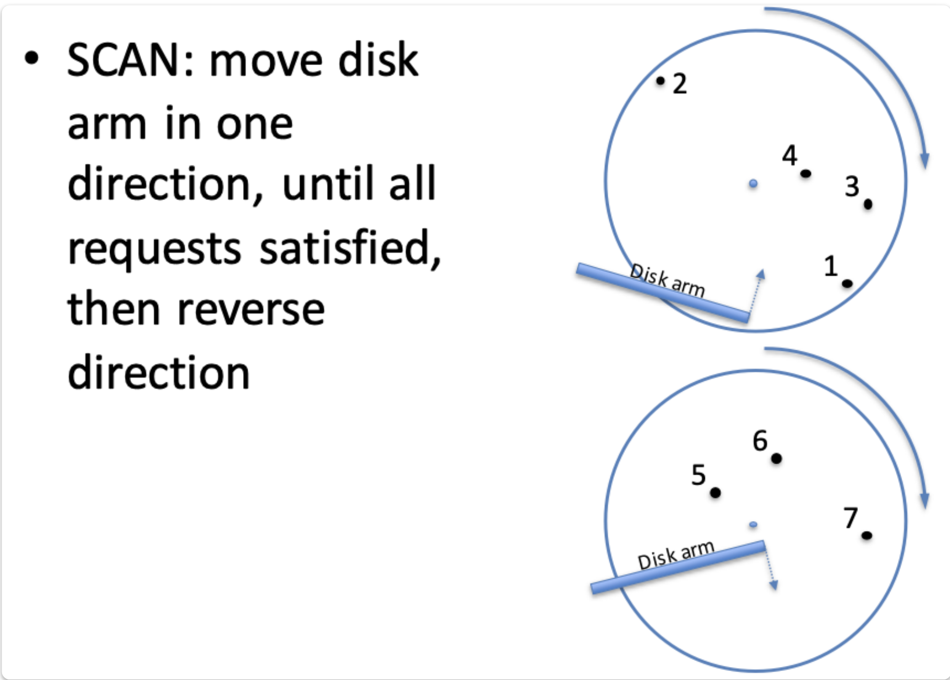
- A basso livello il controller accede ai singoli settori
    - La dimensione tipica del settore è di $256/512\ byte$ identificati da una tripla: $<cilindro, superficie, settore>$
- A livello superiore il driver del disco raggruppa un insieme di settori contigui in un blocco
    - La dimensione tipica del blocco è di $2/4/8\ KB$ identificato da un puntatore su uno spazio di indirizzamento contiguo (da 0 a max-blocchi)

Accesso sequenziale molto più performance dell'accesso random.

Disk scheduling

- *FIFO*

- *Shortest Seek Time First*: seleziona la richiesta più vicina al braccio del disco in movimento
- *SCAN*: muove il braccio del disco in una direzione fino a soddisfare tutte le richieste, quindi inverte direzione.

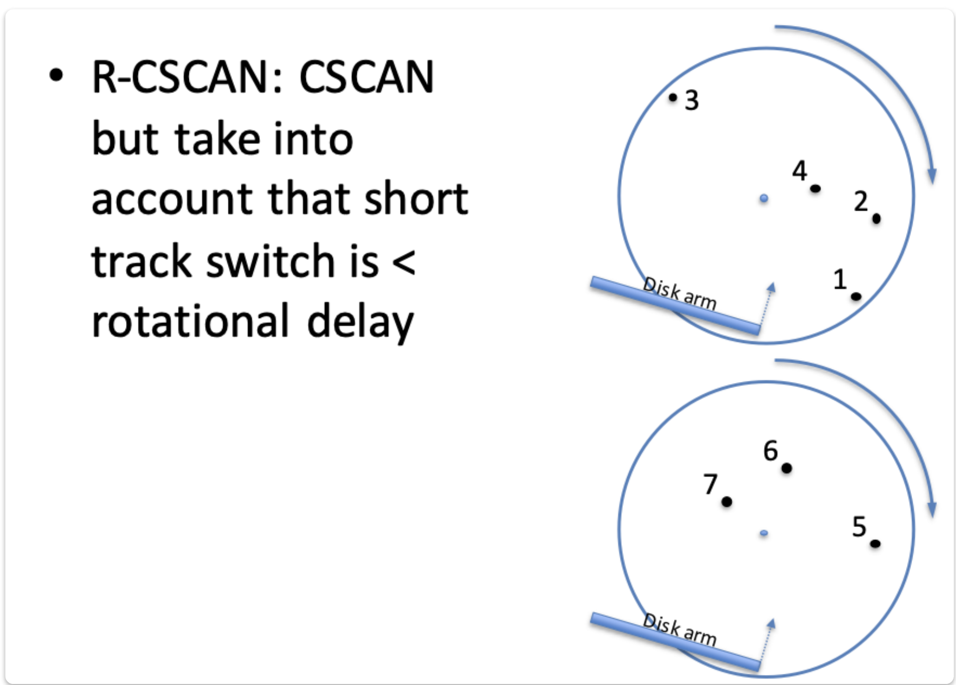


- *CSCAN*: simile all'algoritmo SCAN, ma invece di tornare indietro al primo elemento dopo aver raggiunto l'ultimo, ritorna all'estremità opposta dell'area di ricerca.

- CSCAN: move disk arm in one direction, until all requests satisfied, then start again from farthest request

2
4
3
1
Disk arm
6
7
5
Disk arm

- *R-CSCAN*: variante del CSCAN che tiene conto del fatto che il tempo necessario per passare da una traccia all'altra è minore del ritardo di rotazione del disco.



SSD

I dischi a stato solido (SSD) sono dispositivi di archiviazione non volatili basati sulla tecnologia della memoria flash. Sono più affidabili e veloci dei dischi rigidi (HD) perché non hanno parti in movimento e non hanno tempo di ricerca. I SSD consumano meno energia e sono più costosi per MB di dati. Tuttavia, la capacità dei dischi rigidi è solitamente maggiore. In alcuni sistemi, gli SSD vengono utilizzati come livello di cache aggiuntivo nella gerarchia della memoria.

RAID

Redundant Array of Independent Disks realizza un disco virtuale di capacità superiore a quella dei singoli dischi

- Interfaccia è quella di un unico disco
  Sfrutta il parallelismo per un accesso più veloce
- Blocchi consecutivi di uno stesso file sono distribuiti sui dischi dell'array in modo da permettere operazioni

contemporanee
Sfrutta la ridondanza per accrescere l'affidabilità

- la ridondanza permette di correggere gli errori di certe classi

Diversi livelli di architettura RAID, con diversi livelli di ridondanza:

- *RAID 0*: Dischi asincroni, nessuna ridondanza (striping)
    - Si possono effettuare contemporaneamente operazioni indipendenti
- *RAID 1*: Dischi asincroni, disco con copie ridondanti (mirror)
    - Si possono effettuare contemporaneamente operazioni indipendenti e correggere errori