Architettura degli Elaboratori

Ahmad Shatti
Angel Valeria Correa Caro

2020-2021

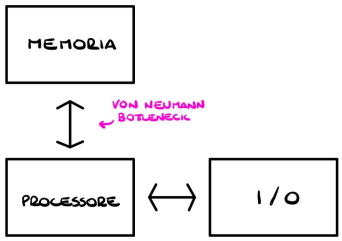
Indice

Chapter 1

Introduzione

1.1 Modello Von Neumann

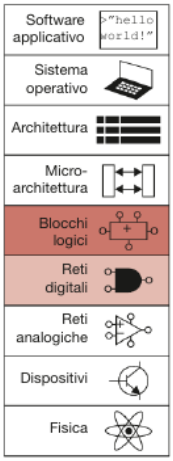
Il calcolatore viene schematizzato con il **modello Von Neumann** che consiste in una memoria che contiene programmi e dati, ed è collegata ad un processore tramite un canale chiamato **Von Neumann bottleneck**. Il processore ha accesso a un componente che è l'ingresso-uscita. Il processore esegue il ciclo while:




```
while(true){
   fetch
   decode
   execute + update(PC)
   writeback
   if(interrupt){
      interrupt_management
   }
}
```

dove per ogni istruzione nella fase di *fetch*, il processore trasmette l'indirizzo del program counter PC alla memoria che risponde con una istruzione, quest'ultima viene *decodificata*, *eseguita*, e si *aggiorna il PC* . Nella fase di *writeback* vengono consolidati i risultati nei registri interni del processore o nella memoria, e infine il processore nella fase *interrupt* controlla se qualche componente ingresso-uscita ha bisogno di attenzione. L'evoluzione tecnologica ha impattato fortemente la struttura del processore, ma non è cambiata altrettanto velocemente la struttura della memoria per cui al giorno d'oggi ci sono processori molto veloci e memoria che non sono in grado di restituire valori nella velocità richiesta. Nel capitolo della gestione della memoria si studieranno tutte le tecniche per far sì che la memoria riesca a stare dietro alle necessità del processore.

1.2 Astrazione



L'astrazione consente di gestire la complessità di un sistema nascondendo i dettagli quando essi non sono importanti. Un calcolatore può essere visto da molti diversi livelli di astrazione. Il primo livello è quello delle *applicazioni*, ed è quello che può vedere l'utente. Le applicazioni vedono un livello che è quello del *sistema operativo*. Scendendo più in basso vi è il livello dell'*architettura*, sotto ancora vi è il livello della *microarchitettura* e da qui in poi vi sono una serie di *componenti logici*. Agli ultimi livelli vi sono i *dispositivi (transistor)* e le *leggi della fisica*. Tra i diversi livelli di astrazioni è presente una interfaccia ben definita, in particolare tra applicazioni e sistema operativo c'è un insieme di chiamate del sistema operativo che permettono di fare delle operazioni particolari. Queste operazioni si distinguono da un sistema operativo all'altro. Il livello dell'architettura è quello che definisce l'insieme delle istruzioni del linguaggio assembler. La microarchitettura unisce il livello di astrazione della logica e quello dell'architettura, che descrive un calcolatore dal punto di vista del programmatore. I componenti logici sono fondamentalmente circuiti con porte booleane che permettono di costruire i moduli. Nei livelli di astrazione si utilizzano dei principi che aiutano a gestire l'astrazione:

- **Gerarchia**: dividere il sistema in moduli e successivamente suddividere ulteriormente ognuno di questi moduli finché i pezzi che li compongono non siano facili da comprendere;
- **Modularità**: avere interfacce ben definite così da connettere due pezzi in maniera semplice;
- **Regolarità**: cercare l'uniformità tra i moduli. I moduli comuni vengono riutilizzati più volte, riducendo il numero di moduli diversi che devono essere progettati.

Chapter 2

Sistemi numerici

2.1 Sistema di numerazione binaria

Per rappresentare informazioni all'interno del calcolatore si usa il **sistema di numerazione binaria** ed esso lavora su insieme di simboli che comprendono {0, 1}. Due soli simboli sono semplicemente rappresentabili come segnali elettrici, cosa non altrettanto semplice quando si hanno 10 segnali (numeri decimali). All'interno di un gruppo di bit, il bit che si trova nella colonna di peso 1 viene chiamato **bit meno significativo** e il bit che si trova all'estremità opposta viene chiamato **bit più significativo**. I numeri binari binari a N cifre senza segno hanno un intervallo pari a $[0, 2^N - 1]$.

2.1.1 Conversione da Binario a Decimale

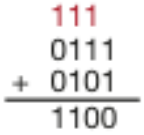
$10110_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 22_{10}$

2.1.2 Conversione da Decimale a Binario

17_{10} = 17 : 2 = 8 resto 1
8 : 2 = 4 resto 0
4 : 2 = 2 resto 0
2 : 2 = 1 resto 0
1 : 2 = 0 resto 1
Leggendo i resti in ordine inverso $17_{10} = 10001_2$

2.1.3 Somma binaria

Come per la somma decimale, se la somma di due cifre è maggiore di quanto si possa esprimere con una singola cifra, allora si ha un riporto di 1 nella posizione successiva.



I sistemi digitali operano sempre su un numero fisso di cifre. Se il risultato della somma è troppo grande per essere espresso con le cifre a disposizione, si dice che la somma da luogo a un **overflow**. Un numero binario a 4 bit copre l'intervallo [0, 15], quindi una somma binaria a 4 bit da un overflow se il risultato è maggiore di 15, in questo caso il bit più significativo viene scartato, producendo un risultato scorretto nei rimanenti 4 bit. La somma di due numeri da N bit può produrre un numero da N o N+1 bit.

2.1.4 Numeri binari relativi

Per rappresentare i numeri binari negativi esistono due sistemi:

- **Modulo e segno**: utilizza il bit più significativo per esprimere il segno e i rimanenti N-1 bit come rappresentazione naturale del modulo. Il bit di segno per un numero positivo vale 0 e per un numero negativo vale 1. $5_{10} = 0101_2$, $-5_{10} = 1101$. Il metodo per calcolare la somma binaria visto prima non funziona per i numeri espressi in modulo e segno: utilizzando tale metodo infatti $-5_{10} + 5_{10} = 10010_2$ invece di 0, quindi un risultato insensato e vi è un'altra complicanza perché esiste una codifica per +0 e una diversa -0 ma entrambe indicano lo stesso numero 0. Un numero in modulo e segno a N bit ha un intervallo pari a $[-2^{N-1} +1, 2^{N-1} -1]$.
- **Complemento a due**: lo zero ha una unica rappresentazione ovvero una sequenza di bit uguali a 0. Un numero in complemento a due ha un intervallo pari a $[-2^{N-1}, 2^{N-1} -1]$. I numeri positivi hanno 0 come bit più significativo, mentre i numeri negativi hanno 1. Il segno di un numero può essere invertito grazie a un'operazione detta **calcolo del complemento a due** che consiste nell'invertire tutti i bit all'interno del numero e poi aggiungere 1. Per trovare la rappresentazione di -2 a 4 bit: +2 = 0010 si invertono tutti i bit allora 1101 e si aggiunge 1, cioè 1101 + 1 = 1110, quindi -2 = 1110. Per trovare il valore decimale di 1001_2: il bit più significativo è 1, quindi si tratta di un numero negativo. Si inverte e si aggiunge 1, cioè $0110_2 + 1 = 0111_2 = 7_{10}$ allora $1001_2 = -7$. Si considerino i due numeri in complemento a 2 su 4 bit: 1110 e 1010, la loro somma è:

	1	1	1	0	+
	1	0	1	0	=
1	1	0	0	0	

  dato che si sta lavorando su 4 bit la somma è 1000. Se la somma fosse stata di due numeri positivi, ci sarebbe stato un overflow, ma siccome il valore risultato è negativo non vi è.

2.1.5 Moltiplicare e dividere per 2

Moltiplicare per 2 un numero binario significa shiftare a sinistra di una posizione, questo accade ogni volta che si moltiplica per un numero che è una potenza di 2, cioè moltiplicare per 8 significa shiftare a sinistra 3 volte. Dualmente dividere un numero per 2 significa shiftare a destra una volta. Questo vale anche per i numeri negativi, cioè, se si vuole effettuare la divisione di -4 per 2 utilizzando 8 bit:

1. Si converte 4 in binario, 00000100
2. Si svolge il calcolo del complemento a due, 11111011 + 1 = 11111100
3. Si shifta a destra di una posizione facendo attenzione che essendo il numero negativo, il bit più significativo che entra è 1: 11111110. Questa operazione di shift si chiama **shift aritmetico** perchè la cifra che entra è la ultima della configurazione precedente

2.1.6 Numeri binari in virgola mobile

Per rappresentare i numeri in virgola mobile si usa lo standard IEEE 754, dove sono stati definiti un certo numero di formati che usano 32 o 64 bit. Si suppone di avere 32 bit con 3 campi: 1 bit che definisce il *segno*, un certo numero di bit per l'*esponente* e il resto di bit per la *mantissa*, con questi significati:

- Se il segno è 0 allora il numero è un positivo, altrimenti se è 1 il numero è negativo;
- La mantissa è il valore del numero senza virgola;
- L'esponente è il numero a cui si deve elevare la base per moltiplicare la mantissa con il segno davanti per avere il valore. L'esponente è rappresentato in "eccesso a k" dove, nello standard IEEE 754, k è uguale a 127 su 32 bit

Per esempio: convertire -5.828125 in binario

1. *determinare il segno*, poichè il numero è negativo allora il segno è uguale a 1
2. *conversione parte intera*, $5 = 101_2$
3. *conversione parte frazionaria*, $0.828125 \cdot 2 = 1.65625$
   $0.65625 \cdot 2 = 1.3125$
   $0.3125 \cdot 2 = 0.625$
   $0.625 \cdot 2 = 1.25$
   $0.25 \cdot 2 = 0.5$
   $0.5 \cdot 2 = 1$

   $0.828125 = 0.110101_2$

4. *normalizzazione della mantissa*, il numero calcolato finora è 101.110101_2, lo standard IEEE 754 richiede che per essere rappresentato il numero sia nella forma 1.m dove m è la mantissa, quindi $1.01110101 \cdot 2^2$
5. *rappresentazione dell'esponente*, si ricava l'esponente sommando 2 al **bias** (127), cioè $2 + 127 = 129_{10}$ che in binario equivale a 10000001_2

-5.828125 corrisponde a 1 10000001 01110101

2.2 Numeri esadecimali

Scrivere lunghe file di cifre binarie aumenta le possibilità di commettere errori. La notazione esadecimale è più compatta e si utilizzano le cifre decimali da 0 a 9 e le prime sei lettere dell'alfabeto dalla A alla F.

2.2.1 Conversione da Esadecimale a Decimale

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

2.2.2 Conversione da Decimale a Esadecimale

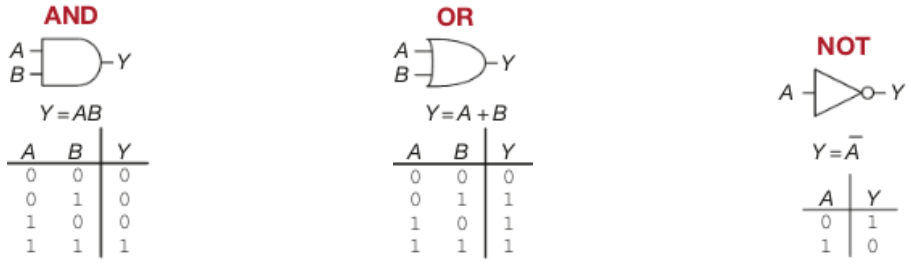
333_{10} = 333 : 16 = 20 resto 13
20 : 16 = 1 resto 4
1 : 16 = 0 resto 1
Leggendo i resti in ordine inverso $333_{10} = 14D_{16}$

Chapter 3

Reti e componenti combinatorie, e logica booleana

3.1 Porte logiche

Le porte logiche sono semplici circuiti digitali che utilizzano uno o più ingressi binari per produrre un'uscita binaria. Le porte logiche implementano 3 funzioni:



La relazione tra ingressi e uscita può essere descritta con una **tabella delle verità** o con una **espressione di Boole**. Una tabella delle verità riporta l'elenco degli ingressi a sinistra e l'uscita a destra, e presenta una riga per ogni possibile combinazione dei valori di ingresso. Una espressione booleana, invece, è un'espressione matematica che utilizza variabili binarie e operatori dell'**algebra di Boole**.

3.2 Reti combinatorie

Le reti combinatorie sono composizione di porte AND, OR e NOT, con queste caratteristiche:

- ogni elemento è di per sè combinatorio;
- ogni nodo della rete è un ingresso per la rete oppure è connesso solamente a un terminale di uscita di un elemento della rete;
- la rete non contiene percorsi ciclici: ogni percorso che la attraversa passa attraverso ogni nodo al massimo una volta.

se manca una di queste proprietà la rete non è combinatoria. Le uscite di una rete combinatoria dipendono dai valori presenti in quel momento agli ingressi; in altre parole, una rete combinatoria utilizza i valori presenti agli ingressi per calcolare i valori delle uscite. Tradizionalmente l'AND e l'OR vengono rappresentati rispettivamente con un $\cdot$ e con un $+$.

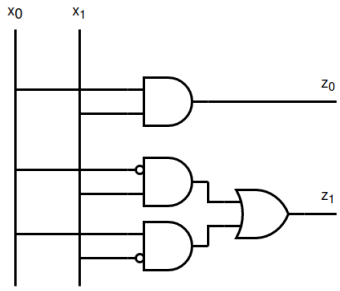
Esempio 1. Si suppone di voler calcolare una funzione g che dice il numero di bit a 1 su due ingressi, questa funzione la si può specificare con una tabella di verità:

x_0	x_1	z_0	z_1
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

ora si vuole trovare un circuito che calcoli esattamente questa tabella di verità in termini di AND, OR e NOT: per ogni colonna delle uscite si avrà un circuito e per ogni riga che ha un 1 nella colonna delle uscite si somma un prodotto:

$$z_0 = x_0 \cdot x_1$$
$$z_1 = \overline{x_0} \cdot x_1 + x_0 \cdot \overline{x_1}$$

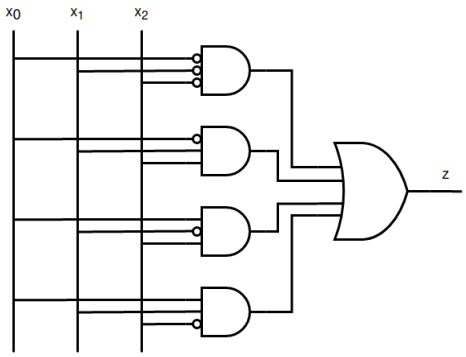
Queste due espressioni rappresentano una sequenza di porte logiche che calcolano la funzione g:



Esempio 2. Calcolare un'altra funzione che dice se i bit a 1 di 3 bit sono pari

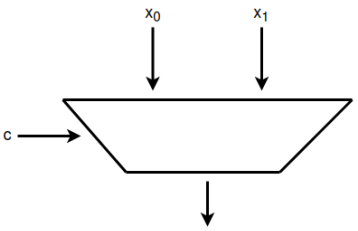
x_0	x_1	x_2	z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$$z = \overline{x_0}\overline{x_1}\overline{x_2} + \overline{x_0}x_1x_2 + x_0\overline{x_1}x_2 + x_0x_1\overline{x_2}$$



Esempio 3. Si studia ora una funzione che deve scegliere uno fra due ingressi possibili a seconda di un ingresso di controllo. Il blocco con questa caratteristica si chiama **multiplexer**

Se l'ingresso di controllo è 0 allora passa x_0
Se l'ingresso di controllo è 1 allora passa x_1



la sua tabella di verità sarà quindi:

x_0	x_1	c	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

che si può riassumere come:

x_0	x_1	c	z
0	-	0	0
1	-	0	1
-	0	1	0
-	1	1	1

perché se c è 0 non interessa il valore di x_1, e quando c è 1 allora non interessa il valore di x_0. Questo dal punto di vista del circuito semplifica molto il lavoro perché l'espressione booleana sarà solo $z = x_0\overline{c} + x_1\overline{c}$ che è una forma molto più semplice rispetto a quella che si avrebbe ottenuto con la prima tabella di verità. I valori con "-" si chiamano **non specificati**.

3.3 Logica booleana

Gli operatori AND, OR, NOT insieme all'alfabeto {0, 1} definiscono un algebra di Boole che possiede assiomi e teoremi:

Assioma del NOT:	se $A = 0$ allora $\overline{A} = 1$	se $A = 1$ allora $\overline{A} = 0$		
Assioma dell'AND:	$0 \cdot 0 = 0$	$0 \cdot 1 = 0$	$1 \cdot 0 = 0$	$1 \cdot 1 = 1$
Assioma dell'OR:	$0 + 0 = 0$	$0 + 1 = 1$	$1 + 0 = 1$	$1 + 1 = 1$

Unità:	$A \cdot 1 = A$	$A + 0 = A$
Zero:	$A \cdot 0 = 0$	$A + 1 = 1$
Idempotenza:	$A \cdot A = A$	$A + A = A$
Terzo escluso:	$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$
Commutativa:	$A \cdot B = B \cdot A$	$A + B = B + A$
Associativa:	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C = A + (B + C)$
Distributività:	$(A \cdot B) + (A \cdot C) = A \cdot (B + C)$	$(A + B) \cdot (A + C) = A + (B \cdot C)$
De Morgan:	$\overline{(A + B)} = \overline{A} \cdot \overline{B}$	$\overline{(A \cdot B)} = \overline{A} + \overline{B}$

Con questi assiomi e teoremi si possono semplificare le espressioni booleane e questo riduce il numero di porte per eseguire una funzione, cioè la rende più piccola.

Esempio 4. Espressione: $\overline{A}BC + A\overline{B}C + AB\overline{C}$ per idempotenza
$\overline{A}BC + A\overline{B}C + AB\overline{C} + A\overline{B}C$ per distributività
$(\overline{A} + A)B\overline{C} + A\overline{B}(\overline{C} + C)$ per terzo escluso
$B\overline{C} + A\overline{B}$

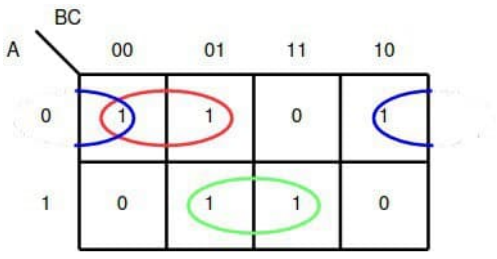
3.3.1 Mappe di Karnaugh

Le mappe di Karnaugh sono un metodo grafico di semplificazione di espressioni booleane con al massimo 4 variabili. Nelle mappe di Karnaugh viene utilizzato il **codice Gray** che è diverso dall'ordine binario perché ogni elemento adiacente differisce per una sola variabile. All'interno della mappa si raggruppano gli 1 con queste regole:

- i raggruppamenti devono essere di potenze di 2;
- raggruppare tutti gli 1, utilizzando il minor numero di cerchi.

Esempio 5. Funzione di 3 variabili:

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



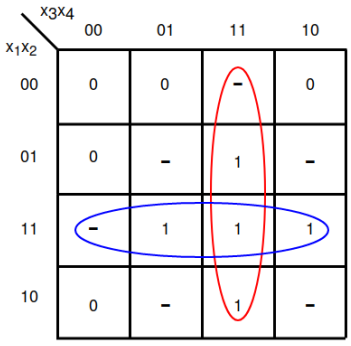
$\overline{A}BC + \overline{A}B\overline{C} = AB(\overline{C} + C) = AB$

$\overline{A}BC + \overline{A}\overline{B}C = \overline{A}C(\overline{B} + B) = \overline{A}C$

$\overline{A}BC + \overline{A}B\overline{C} = \overline{A}C(\overline{B} + B) = \overline{A}C$

Z = $AB + \overline{A}C + \overline{A}C$

In una mappa di Karnaugh vi è la rappresentazione della tabella di verità in forma compatta tale per cui caselle adiacenti corrispondono a valori degli ingressi che differiscono per una sola variabile. Nel caso di una tabella con non specificati si può supporre che essi siano sia 1 che 0:



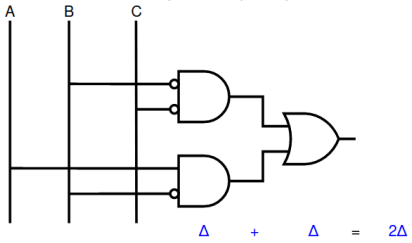
Senza racchiudere i non specificati occorrono 4 implicanti e quindi 4 porte AND e una porta OR. Racchiudendo i non specificati si dimezzano le porte AND:

$$z = x_1x_2 + x_3x_4$$

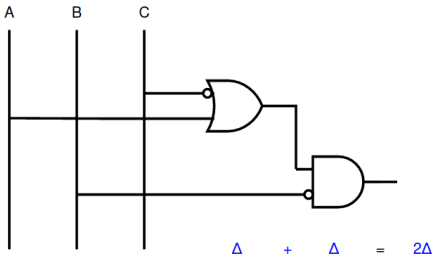
3.4 Ritardi

Ogni rete è caratterizzata da un ritardo Δt necessario affinché in seguito ad una variazione dello stato di ingresso, si produca la corrispondente variazione dello stato di uscita. Solo dopo questo tempo si dice che la rete si è *stabilizzata*. Si suppone che le eventuali variazioni delle variabili d'ingresso avvengono contemporaneamente e che nelle porte AND/OR il ritardo dipende dal numero di ingressi della porta, per N $\leq$ 8 il ritardo è Δt, mentre per le porte NOT si ha ritardo nullo, o meglio inglobato nel ritardo delle porte AND/OR connesse alle uscite delle porte NOT. Non conviene avere porte con tanti ingressi, ma conviene usare porte più piccole da usare in composizione.

$\overline{BC} + A\overline{B}$



$\overline{B}(\overline{C} + A)$



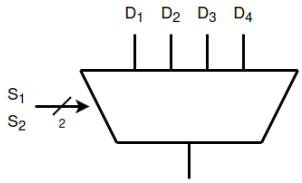
Il secondo circuito ha meno porte rispetto al primo, però può produrre degli effetti indesiderati. Nel primo circuito la porta OR riceve in ingresso i risultati delle porte AND dopo che quest'ultime si sono stabilizzate e dopo un'ulteriore Δt produce un risultato. Nel secondo circuito la porta AND riceve come ingresso B e deve aspettare un Δt della porta OR, questo ritardo può produrre un *glitch*.

3.5 Valutazione costi

Per valutare i "costi" di un circuito, si usano due modelli:

- *classico*: cioè si utilizza la tabella di verità per trovare un circuito di porte AND, OR, NOT;
- si utilizzano componenti "più piccoli".

Si considera il caso del multiplexer 4x1:

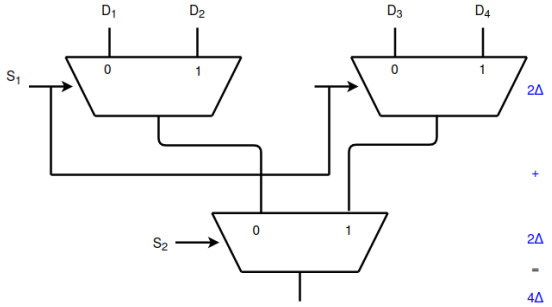


S_1	S_2	D_1	D_2	D_3	D_4	Y
0	0	1	-	-	-	1
0	1	-	1	-	-	1
1	0	-	-	1	-	1
1	1	-	-	-	1	1

$$Y = \overline{S_1}\overline{S_2}D_1 + \overline{S_1}S_2D_2 + S_1\overline{S_2}D_3 + S_1S_2D_4$$

Costo = 2Δ

che si può costruire anche con tre multiplexer 2x1 in cascata:



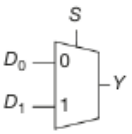
$D_1 \rightarrow S_1 = 0\ S_2 = 0$
$D_3 \rightarrow S_1 = 0\ S_2 = 1$

Il secondo multiplexer è ovviamente meno efficiente del primo ma in termini di *progettazione* è la scelta migliore perchè nel caso in cui i tre multiplexer 2x1 fossero già presenti in un circuito, il costo di collegarli e ottenere un multiplexer 4x1, è irrisorio. Invece per quanto riguarda l'*esecuzione*, cioè quanto tempo, sono nettamente a favore del primo multiplexer. Quindi in termini di tempo il primo multiplexer è la scelta migliore, in termini di velocità di progettazione si usa l'approccio modulare del secondo multiplexer. I livelli aggiuntivi ad albero servono quando si incrementano le alternative in ingresso, se si aumentano le dimensioni allora si mettono in parallelo.

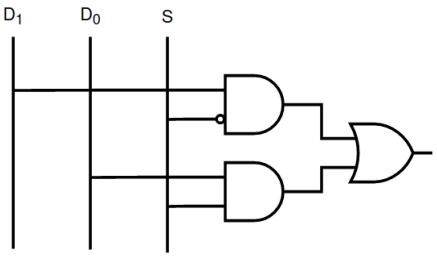
3.6 Componenti reti combinatorie

La logica combinatoria viene raggruppata in blocchi più ampi per costruire sistemi più complessi.

- **Multiplexer**
  I multiplexer sono tra le reti combinatorie più comunemente usate, ha 2^N ingressi da 1 bit, un segnale di controllo da $\log_2 N$ bit e un'uscita da 1 bit. L'uscita sceglie un ingresso basandosi sul segnale di controllo. I multiplexer sono anche chiamati, in gergo, **mux**. Un multiplexer 2x1 può essere costruito a partire dalla logica a somma di prodotti:

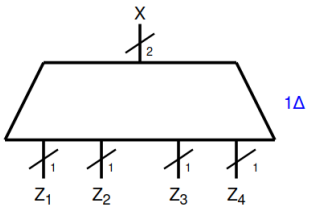


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

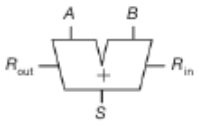


- **Demultiplexer** (o codificatore)
  Il duale del multiplexer è il demultiplexer che prende in ingresso un valore da N bit, un segnale di controllo da $\log_2 N$ bit e restituisce 2^N uscite da un bit. Il demultiplexer attiva una delle sue uscite a seconda della combinazione in ingresso. Per esempio nel caso prendesse un valore da 2 bit:

X_1	X_2	Z_1	Z_2	Z_3	Z_4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

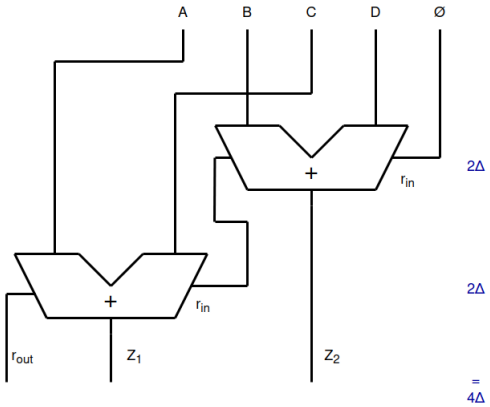


- **Full adder**
  L'addizione è una delle operazioni più comuni nei sistemi digitali. Un full adder a N bit somma due ingressi a N bit, A e B, e aggiunge R_{in} per produrre un risultato a N bit S e un riporto R_{out}. Questo full adder viene chiamato **a propagazione di riporto** perchè il riporto di un bit si propaga nel bit successivo. Il caso di un full adder a 1 bit:

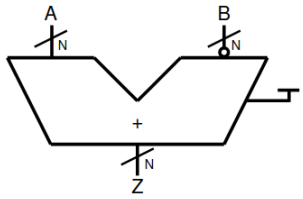


R_{in}	A	B	R_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

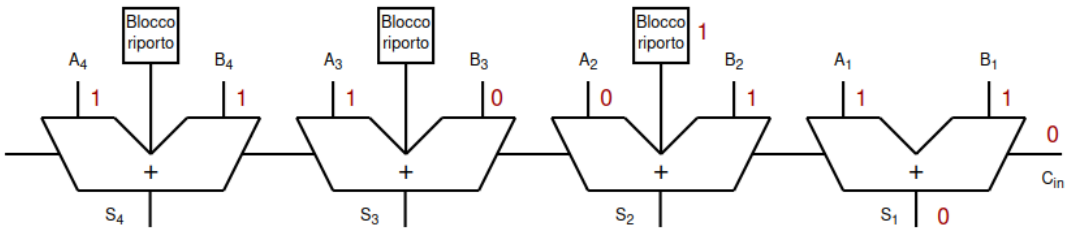
  Se si ha un full adder da un bit, e si vuole costruire un full adder da 2 bit (o da N bit in generale) si possono mettere in cascata i full adder tenendo conto che però i ritardi si sommano:



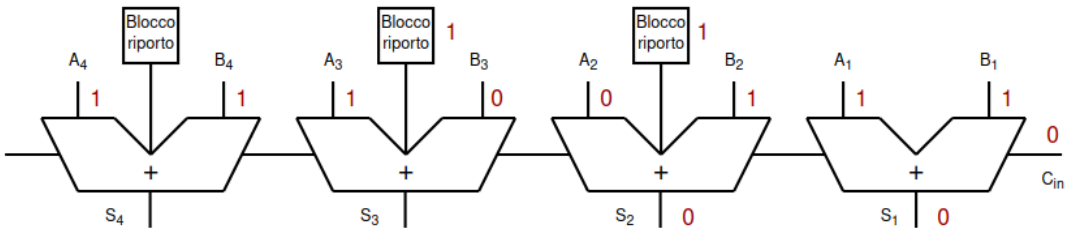
I full adder sono in grado di sommare numeri sia positivi che negativi utilizzando la rappresentazione dei numeri in complemento a due. La sottrazione è quindi facile quanto l'addizione, per calcolare Z = A - B: si crea il numero in complemento a due di B cioè si negano tutti i bit di B per ottenere $\overline{B}$ e si aggiunge 1 per ricavare -B = $\overline{B}$ + 1. Questo valore viene aggiunto ad A per ottenere Z = A + $\overline{B}$ + 1 = A - B. Questo risultato può essere ottenuto con l'utilizzo di un full adder a propagazione di riporto facendo la somma A + $\overline{B}$ con R_{in} = 1. I full adder a propagazione di riporto di grandi dimensioni sono lenti per il fatto che si deve propagare il riporto. Il **carry lookahead adder** risolve il problema della velocità dividendo il full adder in blocchi e aggiungendo un circuito per determinare velocemente il riporto di uscita da ciascun blocco appena è noto il riporto di ingresso. Due bit generano un riporto quando entrambi sono uguali ad 1 e propagano un riporto quando uno dei due è uguale a 1, quindi il riporto si può calcolare con questa formula $\mathbf{C_{i+1} = P_iC_i + G_i}$, dove P_i è la propagazione di un riporto $A_i + B_i$ e, G_i è la generazione di un riporto A_iB_i.



Esempio 6. Se si volesse effettuare la somma 1101 + 1011, in passaggi:

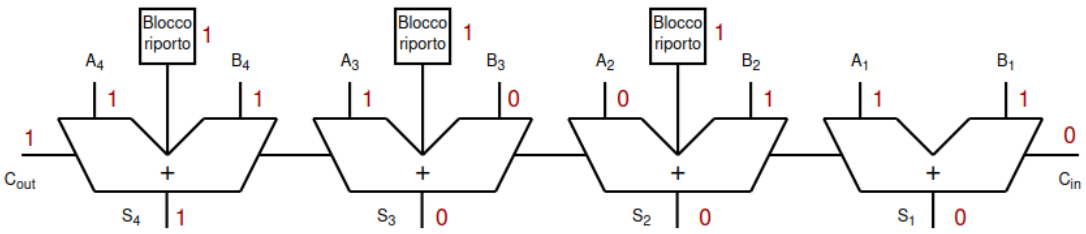


$$C_2 = P_1C_1 + G_1 = 0 \cdot 1 + 1 = 1$$



$$C_3 = P_2C_2 + G_2 = P_2(P_1C_1 + G_1) + G_2 = P_1P_2C_1 + P_2G_1 + G_2 = 0 \cdot 1 \cdot 1 + 1 \cdot 1 + 0 = 1$$

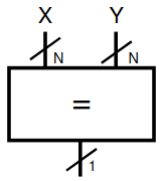
Stesso procedimento per gli altri due passaggi:



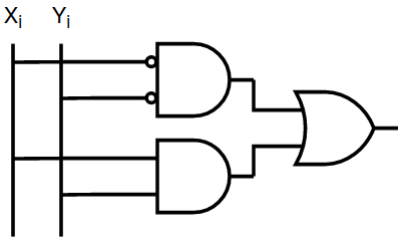
Sostanzialmente il riporto va a definire una serie di porte a cascata di AND e OR che ha una lunghezza proporzionale al numero di bit della somma.

- **Confrontatore**
  Un confrontatore determina se due numeri binari sono uguali o diversi. Un confrontatore riceve due numeri binari a N bit e produce una singola uscita che è 1 se i due valori sono uguali, 0 altrimenti.

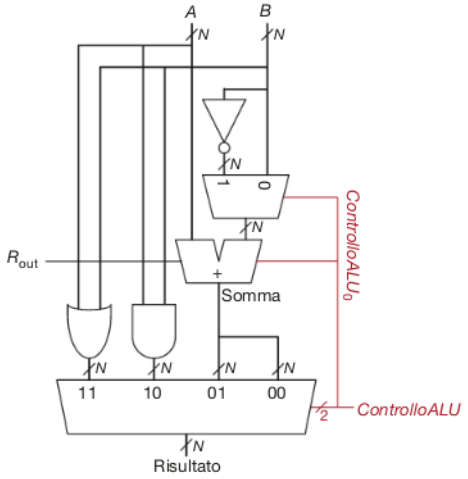
X_i	Y_i	Z
0	0	1
0	1	0
1	0	0
1	1	1



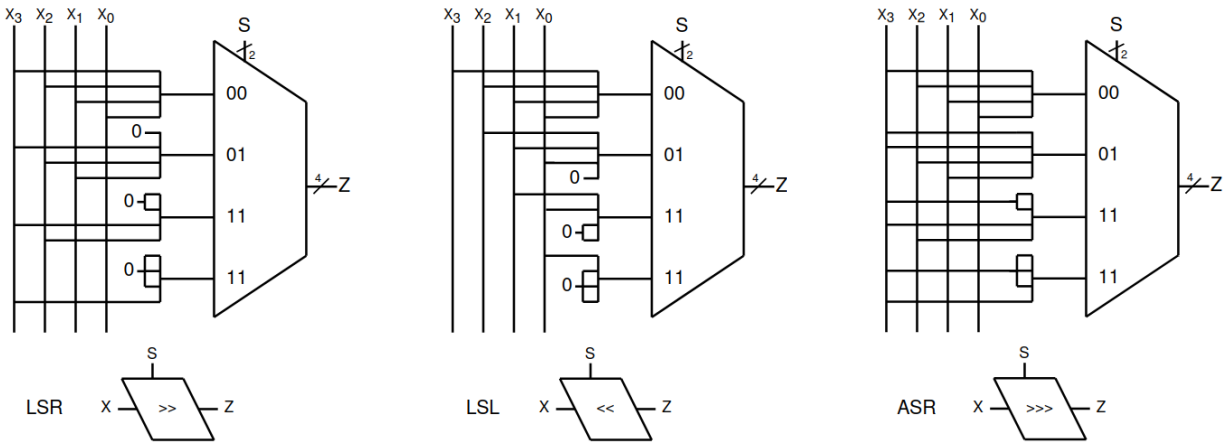
$$Z = \overline{X_i Y_i} + X_i Y_i$$



- **ALU**
  Una ALU è il cuore dei calcolatori e svolge una serie di operazioni logiche e matematiche come addizione, sottrazione, AND, OR di numeri a N bit. Siccome vi sono 4 operazioni, vi è un multiplexer che a seconda del segnale di controllo da 2 bit (*00 = addizione, 01 = sottrazione, 10 = AND, 11 = OR)* farà una delle operazioni. Le operazioni di OR e AND sono semplici basta avere una porta che effettua l'AND o OR bit a bit. Per la somma si utilizza un full adder a N bit dove nella seconda uscita si utilizza un altro multiplexer che nel caso in cui il segnale di controllo è 1 nega il valore di B bit a bit e riporta in ingresso al full adder un 1. Nel caso in cui il valore del multiplexer è 0, il valore di B non viene negato e il riporto in ingresso del full adder è 0. Alcune ALU producono uscite ulteriori chiamate **flag** che danno informazioni aggiuntive: se il risultato dell'ALU è negativo *Negative*, se è uguale a zero *Zero*, o che il full adder ha generato un riporto *Carry* o un overflow *oVerflow*.



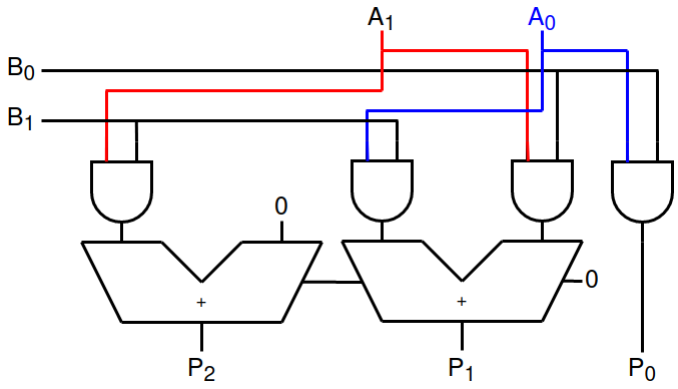
- **Traslatori**
  I traslatori shiftano un numero binario a destra o a sinistra di uno specifico numero di posizioni. Lo shift può essere logico o aritmetico. Un traslatore **logico** (LSR/LSL) trasla i numeri verso sinistra o destra e riempie gli spazi lasciati vuoti con 0. Un traslatore **aritmetico** (ASR) quando trasla i numeri verso destra riempie i bit più significativi con una copia del precedente bit più significativo, invece la traslazione aritmetica verso sinistra si comporta come la traslazione logica verso sinistra. Un traslatore a N bit può essere costruito con un multiplexer NxN. L'ingresso viene traslato da 0 a N-1 posizioni, a seconda del valore presente sull'ingresso di controllo da $\log_2 N$ bit. Per esempio dei traslatori a 4 bit:



- **Moltiplicatore**
  La moltiplicazione tra numeri binari senza segno è simile alla moltiplicazione decimale ma con solo uni e zeri. 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 NxN 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. Per esempio in un moltiplicatore 4x4, il prodotto parziale della prima riga sarà (A_3, A_2, A_1, A_0) AND B_0. Questo prodotto parziale viene sommato al secondo prodotto parziale traslato, (A_3, A_2, A_1, A_0) AND B_1. Le righe seguenti di porte AND e di sommatori

formano e sommano i prodotti parziali restanti. Il moltiplicatore è molto costoso perchè vi è una serie di ritardi per calcolare i prodotti e un'altra serie di ritardi per passare i riporti da un full adder all'altro. Un moltiplicatore 2x2:

	A_1	A_0	·
	B_1	B_0	=
-	A_1B_0	A_0B_0	+
A_1B_1	A_0B_1	-	=
P_2	P_1	P_0	



3.7 Verilog

Verilog fa parte di una serie di linguaggi RTL che sono linguaggi a basso livello che permettono di definire componenti che calcolano funzioni, con o senza stato, che possono essere utilizzati per la **simulazione** o per la **sintesi** di circuiti. Nel primo caso, il programma descrive un certo componente e viene utilizzato per simulare il comportamento e per controllare che i calcoli sono corretti. Nel secondo caso, il programma viene utilizzato per generare le specifiche da utilizzare per realizzare un circuito che implementi fisicamente quello descritto dal programma. Le specifiche possono consistere nel file di configurazione di una FPGA.

3.7.1 Sintassi

In verilog per rappresentare una tabella di verità si usano le **primitive** ma facendo attenzione che hanno un solo bit di output. I **module** servono per scrivere le espressioni dell'algebra booleana e all'interno si usano uno o più **assign** dove si possono avere più bit di uscita. Implementazione del full adder:

```
primitive fa_somma(output s, input r, input x1, input x2);
    table
      0 0 0 : 0 ;
      0 0 1 : 1 ;
      0 1 0 : 1 ;
      0 1 1 : 0 ;
      1 0 0 : 1 ;
      1 0 1 : 0 ;
      1 1 0 : 0 ;
      1 1 1 : 1 ;
   endtable
endprimitive // fa_somma

primitive fa_riporto(output s, input r, input x1, input x2);
    table
      0 0 0 : 0 ;
      0 0 1 : 0 ;
      0 1 0 : 0 ;
      0 1 1 : 1 ;
      1 0 0 : 0 ;
      1 0 1 : 1 ;
      1 1 0 : 1 ;
      1 1 1 : 1 ;
   endtable
endprimitive // fa_riporto

module fulladder(output riporto, output somma, input ripin, input x1, input x2);

   fa_riporto m1(riporto, ripin, x1, x2);
   fa_somma   m2(somma, ripin, x1, x2);

endmodule // fulladder
```

Per testare il circuito si scrive un modulo in cui si testano i valori:

```
module testFA();

   reg rin, a, b;              // un registro per ognuno degli ingressi
```

```
  wire rip, som;              // un filo per ognuno delle uscite

  fulladder adder(rip, som, rin, a, b);

  initial
    begin
 $dumpfile("testFA.vcd");   // istruzioni per vedere il risultato
 $dumpvars;

 a   = 0;                   // inizialmente a, b sono 0 e rin = 1
 b   = 0;
 rin = 1;
 #2;                        // dopo 2 unita' di tempo...
 a   = 1;
 b   = 0;
 rin = 1;
 #2;
 a   = 1;
 b   = 1;
 rin = 1;
 #2;
 a   = 1;
 b   = 1;
 rin = 0;
 #5 $finish;

    end
endmodule // testFA
```

Qualora volessimo utilizzare più bit si deve far procedere l'identificatore da una coppia di parentesi quadre che al loro interno contengono un indice che rappresenta il bit più significativo e un indice che rappresenta il bit meno significativo, separati dal simbolo :. Per esempio **reg [7:0] a** dichiara un registro da 8 bit dove il bit più significativo è a[7] e quello meno significativo è a[0]. I non specificati si indicano con il valore ?.

Chapter 4

Reti e componenti sequenziali

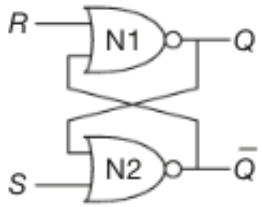
4.1 Reti sequenziali

Nelle reti sequenziali le uscite dipendono dai valori presenti in quel momento agli ingressi quanto dai valori precedenti, cioè hanno uno stato. Lo stato, si comporta come una memoria, ed è un insieme di bit che contiene tutte le informazioni sul passato necessarie a spiegare il comportamento futuro della rete. Le reti sequenziali vengono usate per calcolare automi.

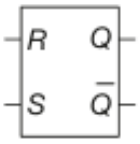
4.2 Latch e Flip Flop

I Latch e i Flip flop sono semplici reti sequenziali che ricordano un bit di stato, e sono i seguenti:

- **Latch SR** è composto da due porte OR negate (NOR) collegate a croce. Il suo stato può essere controllato mediante gli ingressi, S e R, che settano o resettano l'uscita Q. La figura seguente mostra: lo schema, la tabella di verità e il simbolo circuitale.

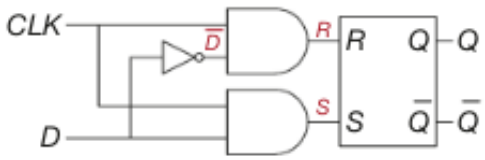


S	R	Q	$\bar{Q}$
0	0	Q_{prec}	$\bar{Q}_{prec}$
0	1	0	1
1	0	1	0
1	1	0	0

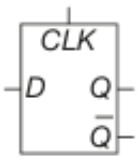


  Se S=1 allora vi è un set e quindi Q=1. Se R=1 allora si applica un reset e quindi Q=0. Se S e R sono entrambi uguali a 0, non si sa che valore può avere Q allora si suppone che Q assuma un valore precedente noto, denominato Q_{prec} che può essere 0 o 1. Q_{prec} rappresenta lo stato del sistema. Attivare simultaneamente entrambi gli ingressi non ha molto senso, in quanto il Latch SR dovrebbe settarsi e resettarsi allo stesso tempo, il che non è possibile, e in questo caso, dà come risposta 0 per entrambe le uscite.

- **Latch D** risolve il problema di S e R attivati simultaneamente. Il Latch D ha due ingressi: un ingresso dati che controlla il prossimo stato, e un ingresso **clock** che controlla il momento del cambio di stato. La lunghezza di un ciclo di clock si chiama τ.

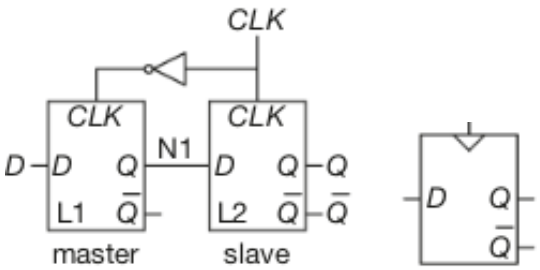


CLK	D	$\bar{D}$	S	R	Q	$\bar{Q}$
0	X	$\bar{X}$	0	0	Q_{prec}	$\bar{Q}_{prec}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0



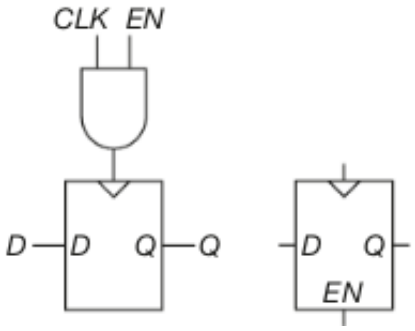
  Quando clock=0, S e R sono 0 indipendentemente dal valore assunto da D, e Q ricorda il valore precedente (*Latch opaco*). Se invece clock=1, allora una porta AND produce un 1 e l'altra 0, a seconda del valore di D, e Q = D (*Latch trasparente*). Tutte le scritture avvengono sempre quando il clock=1.

- **Flip flop D** può essere costruito a partire da due Latch D in cascata controllati da due segnali di clock. Il primo Latch viene detto *master* e riceve il segnale di clock negato, mentre il secondo viene detto *slave*. Il nodo che unisce i due Latch D si chiama N1.

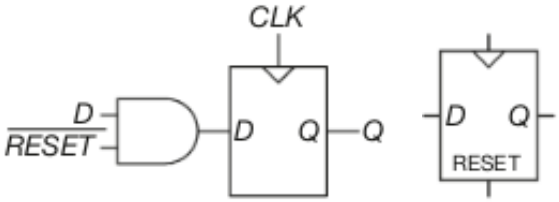


Quando il clock=0, il Latch master è trasparente, mentre il Latch slave è opaco quindi qualsiasi valore di D viene portato a N1. Quando il clock=1, il Latch master è opaco, è quello slave trasparente allora il valore di N1 viene trasmesso a Q. In altre parole qualunque sia il valore di D subito prima del fronte di clock (passaggio da 0 a 1) questo è il valore che viene trasferito a Q.

- **Flip flop con abilitazione** aggiunge un altro ingresso, chiamato **enable** (EN o WE) che indica quando scrivere sul fronte del clock (clock=1) e ignorare gli altri cicli di clock. D'ora in poi il Flip flop con abilitazione verrà chiamato **registro da 1 bit**.



- **Flip flop resettabile** aggiunge un ulteriore ingresso chiamato **reset**. Quando reset è 0, il Flip flop resettabile si comporta come un normale Flip flop D. Quando invece reset è 1, il Flip flop resettabile ignora D, e resetta l'uscita a 0.

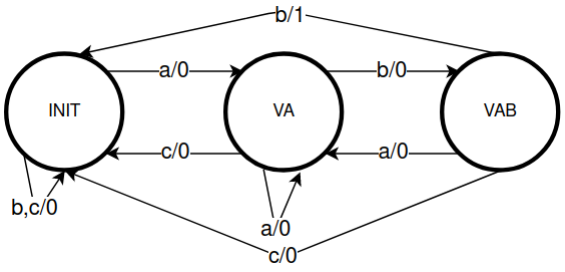


4.3 Macchine a stati finiti

Le reti sequenziali sincrone vengono utilizzate per creare delle **macchine a stati finiti** (FSM o automi). Una FSM possiede M ingressi, N uscite, k bit di stato e inoltre riceve un segnale di clock. Una FSM è composta da due blocchi di logica combinatoria: la **logica di stato prossimo** e la **logica d'uscita**, e da un registro che immagazzina lo stato. I bit all'interno del registro dipendono da quanti stati si devono rappresentare facendo $\log_2 S$ dove S è il numero degli stati. A ogni fronte di salita del clock, la FSM avanza allo stato prossimo, definito in base agli ingressi e allo stato presente. Esistono due classi generali di macchine a stati finiti:

- **Macchine alla Mealy** dove le uscite dipendono sia dallo stato presente della macchina sia dagli ingressi attuali;
- **Macchine alla Moore** dove le uscite dipendono esclusivamente dallo stato presente della macchina.

Si suppone di avere un alfabeto di lettere {a, b, c} e che si voglia riconoscere le sequenze "abb". L'automa di Mealy corrisponde a:



Per realizzarlo come circuito occorre:

1. Rappresentazione stringhe in ingresso (scelta qualunque);

a	00
b	01
c	11

2. Un modo per rappresentare lo stato (scelta qualunque);

INIT	00
VA	01
VAB	11

3. Funzione che dato lo stato e gli ingressi, descriva come sono fatte le uscite;

S_1S_0	In_1 In_0	Z
00	00	0
	01	0
	11	0
01	00	0
	01	0
	11	0
11	00	0
	01	1
	00	0

dove S_1S_0 sono i 2 bit dello stato, mentre In_1In_0 sono i 2 bit degli ingressi. Per esempio la prima riga: se si vede una "a" si arriva allo stato "VA" ma si rimane in 0 quindi z=0.

$$Z = S_1 S_0 \overline{In_1} In_0$$

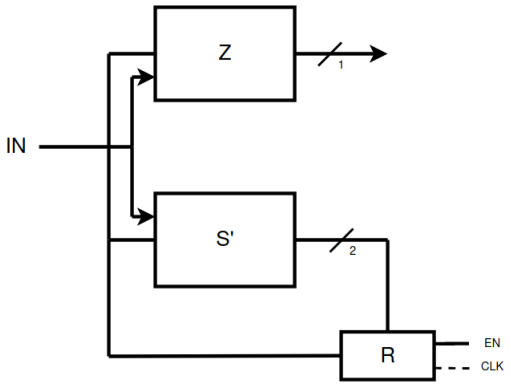
4. Funzione che dato lo stato e gli ingressi, descriva il prossimo stato.

S_1S_0	In_1 In_0	$S_1'S_0'$
00	00	01
	01	00
	11	00
01	00	01
	01	11
	11	00
11	00	01
	01	00
	00	00

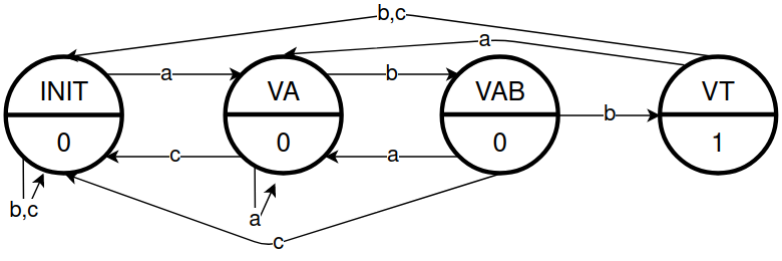
dove $S_1'S_0'$ sono i 2 bit dello stato successivo

$$S_1' = \overline{S_i} S_0 \overline{In_1} In_0$$
$$S_0' = \overline{In_1}(S_1 \overline{In_0} + S_0 \overline{In_0})$$

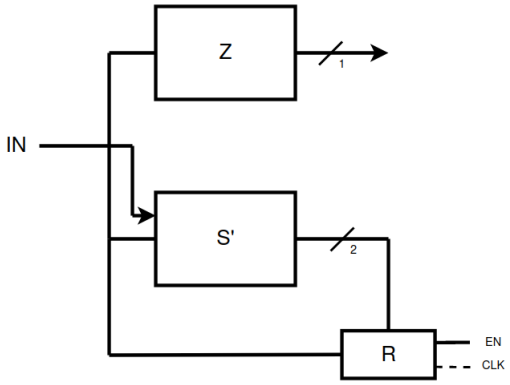
Si possono utilizzare queste funzioni per metterli all'interno di un circuito e realizzare una rete di Mealy:



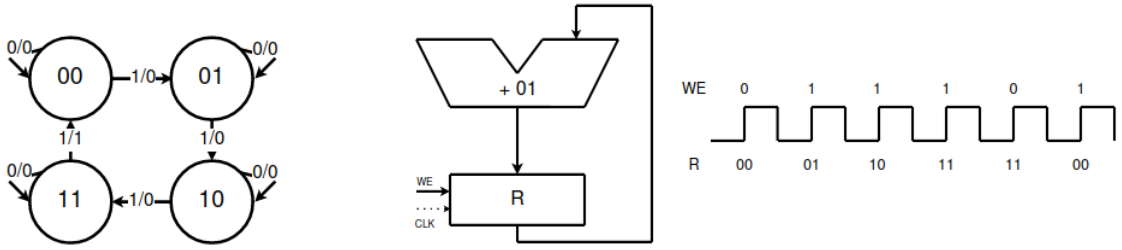
A ogni giro di clock prende il vecchio stato interno insieme al nuovo ingresso e, calcola l'uscita e il prossimo stato interno. L'automa di Moore che riconosce "abb" sull'alfabeto {a, b, c} corrisponde a:



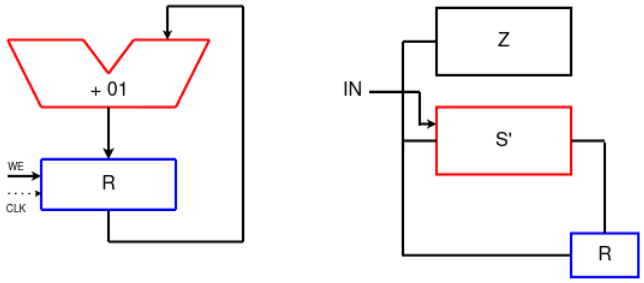
A livello di circuito adesso si hanno tabelle di verità diverse, e la rete di Moore equivale a:



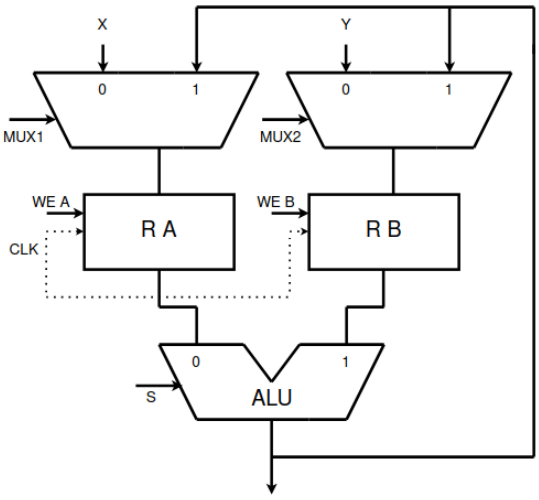
Riassumendo il procedimento standard per realizzare una rete sequenziale è derivare l'automa, calcolare la logica di stato prossimo, la logica d'uscita, capire quanti bit inserire nel registro e collegare il tutto realizzando una rete sequenziale. Si suppone di avere un contatore modulo 2^2, l'automa deve avere tanti stati quanti sono gli oggetti che si devono contare, in questo caso 4, e ogni volta che si ha un evento, si passa allo stato successivo.



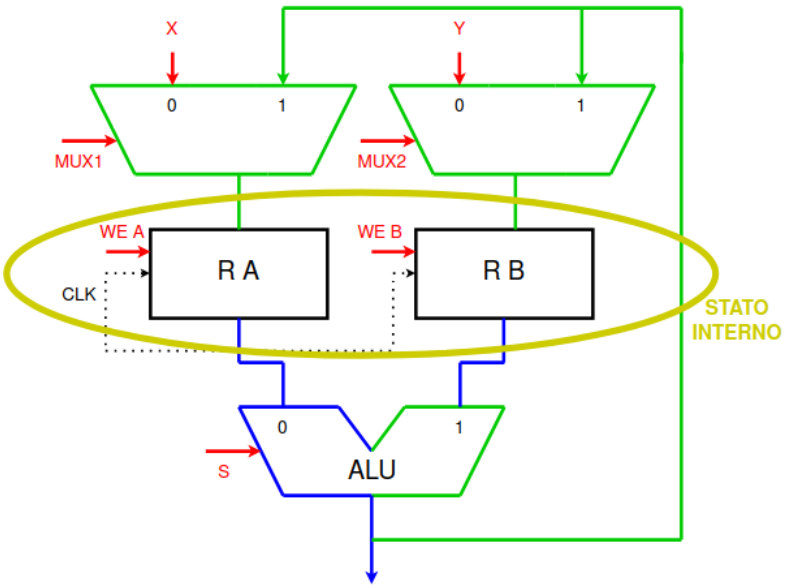
Se ora si volesse calcolare un contatore modulo 2^3, si dovrebbe fare molti cambiamenti sull'automa ma non sui componenti: la ALU sommerebbe 3 bit e il registro conterebbe 3 bit, cioè si aumentano semplicemente la quantità dei componenti standard. Se ora si volesse riconoscere nel contatore modulo 2^2 i vari componenti della rete sequenziale:

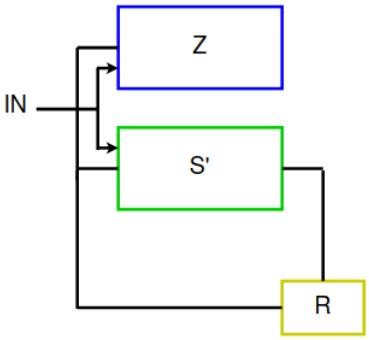


dove Z non ha nessun componente, quindi si può dire che Z in questo caso è la **funzione identità**, cioè l'uscita del prossimo stato interno passa direttamente. Si è riconosciuto una rete di Moore (perchè non è necessario mandare l'ingresso alla funzione che calcola le uscite) implementata con dei componenti standard anzichè con la logica delle porte AND/OR. Ultimo esempio, data questa rete sequenziale:



determinare la funzione che calcola le uscite, la funzione che calcola la logica di stato prossimo, lo stato interno e riconoscere se è di Mealy o di Moore. Si devono individuare lo stato interno, gli ingressi e le uscite. S





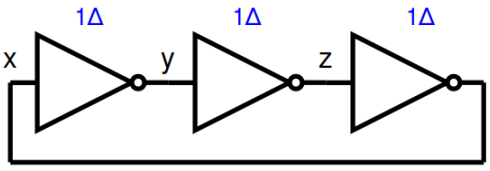
Si colorano di rosso gli ingressi e di blu l'uscita. Se è una rete di Moore l'uscita dovrebbe dipendere solo dallo stato interno: l'uscita di R A non dipende da WE A, analogamente per R B, quindi fino all'ingresso della ALU le uscite dipendono solo dallo stato interno. La ALU dipende da S che è un ingresso esterno perchè `Z = if(S == 0) then (A + B) else (A - B)`, quindi è una rete di Mealy. Siccome i multiplexer prendono anche l'uscita della ALU allora anch'essa fa parte, oltre della logica d'uscita, della logica del prossimo stato interno.

4.4 Reti sequenziali sincrone e asincrone

Le reti sequenziali con percorsi ciclici possono avere comportamenti instabili, per evitare questi problemi si predilige 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 è sequenziale sincrona se:

- ogni elemento della rete è un registro o una rete combinatoria;
- deve essere presente almeno un registro;
- tutti i registri ricevono lo stesso segnale di clock;
- ogni percorso ciclico contiene almeno un registro.

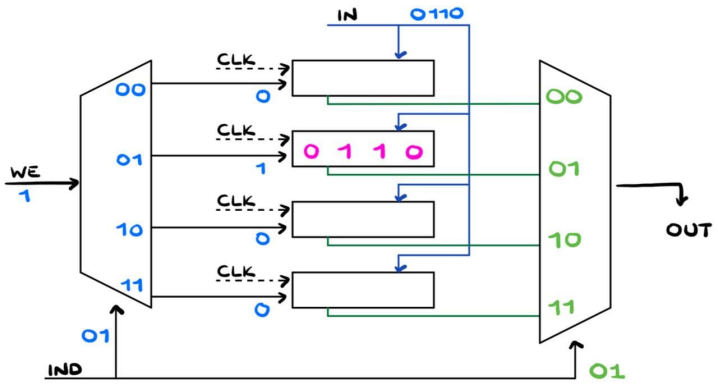
In teoria, la progettazione di reti **asincrone** è più generale rispetto a quella di reti sincrone perchè la temporizzazione del sistema non è limitata dal segnale di clock dei registri. Le reti asincrone talvolta sono necessarie quando la comunicazione avviene tra due sistemi con segnali di clock differenti. In una rete di Mealy asincrona il valore del prossimo stato interno andrà direttamente alla logica che calcola l'uscita. Per esempio si considera una rete asincrona formata da 3 porte NOT dove l'uscita della terza porta è collegata in retroazione all'ingresso della prima porta e si immagina che ogni porta abbia un ritardo di $1\Delta t$. Si supponga che il nodo X sia inizialmente a 0, questo significa che Y = 1 e Z = 0, allora X = 1, il che non corrisponde al valore iniziale. La rete non ha stati stabili e viene dunque detta **instabile**. Il valore di ogni singolo nodo è difficile da prevedere perchè dipende dal ritardo di propagazione e dal modo in cui la porta NOT è stata fabbricata.



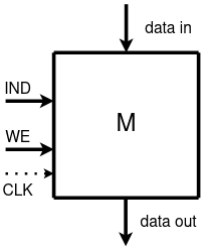
4.5 Componenti reti sequenziali

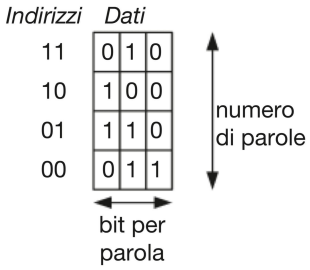
I componenti delle reti sequenziali sono quei blocchi che possono mantere uno stato, e sono:

- **Registro**: è un banco di N Flip flop con lo stesso segnale di clock in modo che tutti i bit vengono aggiornati allo stesso tempo e ricorda valori in ingresso precedenti;
- **Unità registri**: viene utilizzato per immagazzinare variabili temporanee. A volte l'implementazione è come segue: si considera il caso in cui si ha 4 registri: ogni registro ha un segnale di clock e un segnale di WE che arriva da un demultiplexer. Il demultiplexer ha come segnale di controllo un indirizzo. Un dato in ingresso viene propagato su tutti gli ingressi del banco di registri. Si suppone di voler scrivere la configurazione 0110 all'indirizzo 01. Il segnale al demultiplexer sarà 01, questo significa che il WE sui registri sono tutti 0 tranne nel registro che verrà scritto, e il dato in ingresso 0110 verrà scritto solo sul registro con il WE=1, quando il segnale di clock=1. In tutti gli altri registri rimarrà il vecchio valore. Se invece si suppone di voler leggere uno dei registri: le uscite dei registri vengono collegate a un multiplexer, sempre con segnale di controllo 01, e restituisce il risultato.

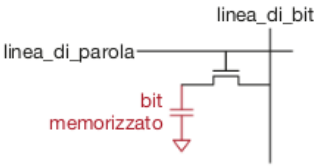


- **Memorie**, i sistemi digitali richiedono delle memorie per immagazzinare i dati utilizzati. I Flip flop sono un esempio di memoria in grado di immagazzinare una piccola quantità di dati. La memoria è costituita da un ingresso e un'uscita dati, e ha come segnale di controllo l'indirizzo per leggere o scrivere, e il segnale di WE che indica quando scrivere. A un livello più alto di astrazione, da un punto di vista logico, 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, chiamata **parola**, viene specificata da un indirizzo. Il valore letto o scritto nella memoria viene chiamato **dato**. Un componente con un numero N di bit di indirizzo e numero M di bit di dato possiede 2^N righe e M colonne. In figura vi è un esempio di un componente memoria con 2 bit di indirizzo e tre bit di dato. Le memorie vengono classificate in **RAM** e **ROM**. La RAM è una memoria volatile, cioè perde traccia dei suoi dati una volta spenta, invece la ROM è una memoria non volatile perché trattiene i suoi dati anche in assenza di alimentazione. I due tipi principali di memorie RAM sono:





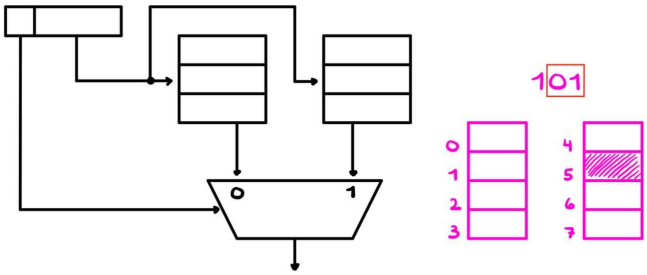
  - **RAM statica**: dove i bit immagazzinati nella memoria non hanno bisogno di essere ricaricati;
  - **RAM dinamica**: dove occorre ricaricare il contenuto di ogni cella di memoria, altrimenti il contenuto viene perso.



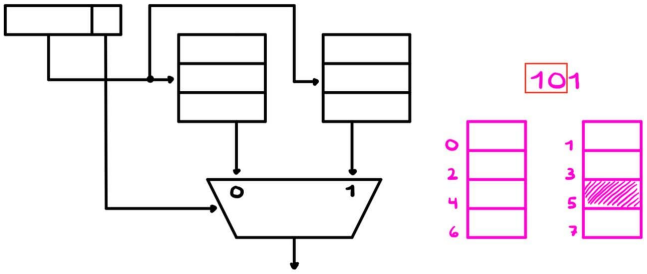
La RAM dinamica memorizza un bit come presenza o assenza di carica in un **condensatore**. Il *transistor* si comporta come un interruttore che connette o disconnette il condensatore dalla linea di bit. Quando la linea di parola è attiva, il transistor si accende e il valore del bit immagazzinato viene trasferito alla o dalla linea di bit. La lettura distrugge il valore del bit immagazzinato nel condensatore, quindi la parola di dato deve essere riscritta a ogni lettura. Anche quando la RAM dinamica non viene letta è necessario ricaricare i contenuti ogni millisecondi perché la carica sul condensatore si perde gradualmente. Per implementare una memoria esistono due schemi:

  - **Memoria modulare sequenziale**: è caratterizzata dalla distribuzione degli indirizzi sequenzialmente all'interno di un modulo. Quando si esaurisce la capacità di un modulo, si passa a distribuire gli indirizzi sequenzialmente nel modulo successivo. Per esempio se ci fossero 2 moduli di memoria: gli indirizzi con i bit meno

significativi vengono inviati ai moduli di memoria, per ottenere 2 parole, mentre il bit più significativo dell'indirizzo è il segnale di controllo di un multiplexer per selezionare una delle 2 parole. Si immagina di avere 101_2 come indirizzo, allora verrà selezionata la posizione 1 del secondo modulo perché il bit più significativo è 1;



- **Memoria modulare interallacciata**: m parole aventi indirizzi consecutivi sono allocate in m moduli di memoria distinti e consecutive. Per esempio con 2 moduli di memoria ma i bit più significativi vengono inviati ai moduli per ottenere 2 parole e il multiplexer è comandato dal bit meno significativo dell'indirizzo. Si immagina di avere 101_2 come indirizzo, quindi verrà selezionata la posizione 2 del secondo modulo perché il bit meno significativo è 1.



Nella memoria modulare interallacciata se si vuole accedere a due parole consecutive, per esempio nella posizione 4 e 5, basta non considerare il multiplexer e prendere entrambe le uscite dei due moduli. Questo procedimento sarà utile nelle *memorie cache*. Nella memoria modulare sequenziale, se si vuole accedere alla posizione 4 e 5, lo si deve fare in due cicli di clock.

Chapter 5

Forme di parallelismo

5.1 Parallelismo

Le forme di parallelismo permettono di eseguire i programmi più velocemente seguendo dei modelli:

- **parallelismo spaziale**: vengono usate più copie dell'hardware in modo che più lavori possano essere svolti contemporaneamente;
- **parallelismo temporale**: ogni compito viene diviso in fasi come in una catena di montaggio. Più compiti possono essere distribuiti tra le varie fasi. Nonostante ogni compito debba passare attraverso tutte le fasi, compiti diversi possono trovarsi in ogni fase in qualsiasi momento, cosicché i diversi compiti si sovrappongono.

Esempio: un pasticcere deve aumentare la sua produzione di biscotti. Può utilizzare il *parallelismo spaziale* dove chiede a un altro pasticcere di aiutarlo perché anche lui possiede una teglia da biscotti e un forno, o il *parallelismo temporale* dove il pasticcere compra una seconda teglia per biscotti. Così una volta inserita una teglia nel forno, può incominciare a impastare i biscotti sulla nuova teglia mentre aspetta che gli altri siano pronti.

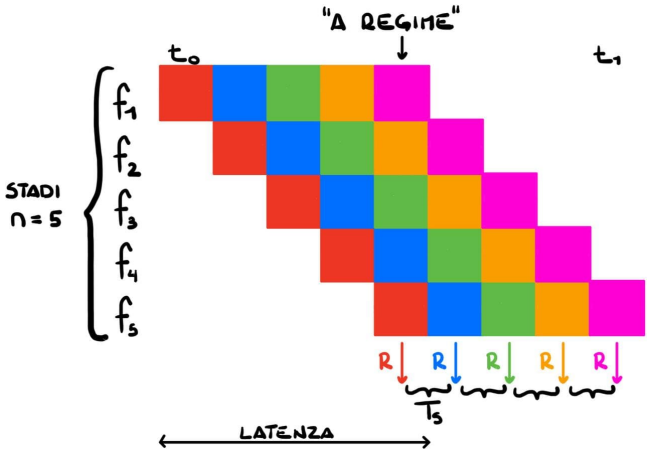
Si suppone di voler calcolare una certa *f(x)* impiegando un certo t_f:

- con il parallelismo spaziale con *m* task, si impiega sequenzialmente $\boldsymbol{T_{seq} = m \cdot t_f}$,
  invece parallelamente con *n* workers $\boldsymbol{T(n)_{par} = \frac{m}{n} \cdot t_f}$;
- con il parallelismo temporale con *m* task e *n* workers $\boldsymbol{T_{seq} = m \cdot (n \cdot t_f)}$
  invece $\boldsymbol{T(n)_{par} = n \cdot t_f + (m - 1) \cdot t_f}$.

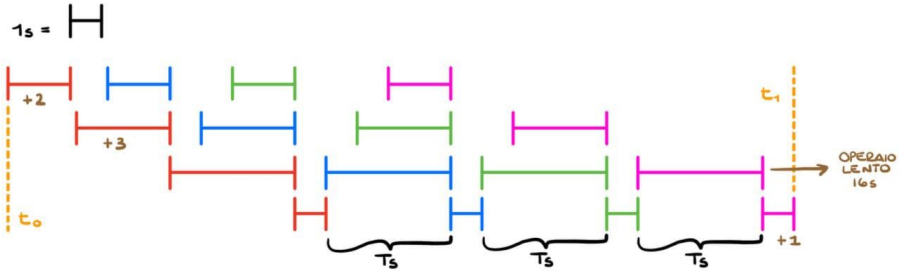
Il tempo sequenziale e parallelo servono per sapere quanto si ha guadagnato in tempo, cioè lo $\boldsymbol{speedup(n) = \frac{T_{seq}}{T(n)_{par}}}$ dove (n) è il grado di parallelismo. Quindi nel parallelismo spaziale si avrà uno speedup(n) = n, cioè si va n volte più veloci, mentre nel parallelismo temporale, supponendo che m è molto più grande di n, si avrà speedup(n) $\simeq$ n. La **scalabilità** è il rapporto fra il tempo impiegato a calcolare con grado di parallelismo 1 e quello impiegato con grado di parallelismo n: $\boldsymbol{scalab(n) = \frac{T_{par}(1)}{T_{par}(n)}}$. Il parallelismo temporale è chiamato anche *pipeline*. Il parallelismo spaziale è composto da due tipi:

- **Map**: riceve un dato in input, divide il dato in sottotask, ognuno dei workers calcola la stessa funzione f sul sottotask che riceve in input. La politica di schedulazione dei task in input può essere round robin o altro. Quando tutti i workers hanno finito, i risultati dei sottotask vengono utilizzati per ricostruire il risultato finale, normalmente avente la stessa struttura del dato in input.
- **Farm**: riceve uno stream di dati in input, calcola dei risultati e li manda sullo stream di output. Ognuno dei workers calcola la stessa funzione f sui dati che riceve in input. I task in ingresso arrivano in istanti diversi, cioè uno dopo l'altro. La politica di raccolta dati può mantenere o meno l'ordinamento di ingresso, mentre la politica di schedulazione dei task in input può essere round robin o altro.

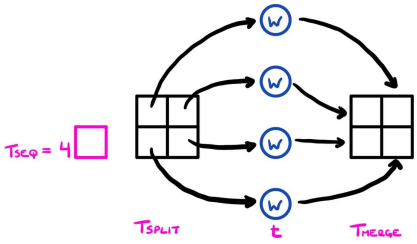
La velocità di un sistema è caratterizzato dalla **latenza *L***, che misura il tempo fra l'inizio di un task e la produzione del relativo risultato, e il **throughput** ovvero il numero di task completati per unità di tempo. Nel parallelismo spaziale si cerca in generale di ridurre la latenza, mentre nel parallelismo temporale si cerca di incrementare il throughput. Si definisce **tempo di servizio** $\boldsymbol{T_s}$ il tempo che intercorre fra la produzione di due risultati consecutivi. Nel caso sequenziale $\boldsymbol{L = T_s}$. L'inverso del tempo di servizio è il throughput. Lo stadio dove tutti i workers lavorano si chiama **a regime** (o steady state). Il **tempo di completamento** $\boldsymbol{T_c}$ rappresenta l'intervallo fra il tempo in cui inizia il primo calcolo e quello che termina l'ultimo calcolo $\boldsymbol{T_c = t_1 - t_0}$.



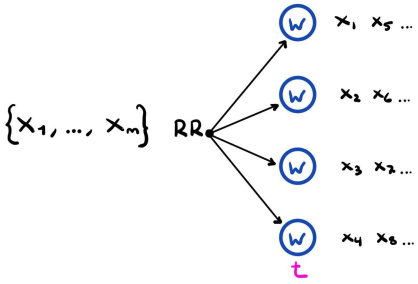
Nel parallelismo temporale $\boldsymbol{T_s = max\{t_i\}}$ mentre $\boldsymbol{T_c = \sum t_i + (m-1) \cdot max\{t_i\}}$, dove $\sum t_i$ è la latenza. Se m è molto più grande di n allora $T_c \simeq m \cdot T_s$. Si considera il caso con 4 task: $t_1 = 2s$, $t_2 = 3s$, $t_3 = 4s$ e $t_4 = 1s$, allora $T_s = 4s$ mentre il tempo di completamento è il tempo per l'operaio più lento 4s + 4s + 4s + 4s = 16s e in seguito si aggiunge il tempo per arrivare all'operaio più lento 2s + 3s = 5s e per uscire da esso + 1s, quindi T_c = 16s + 5s + 1s = 22s, altrimenti seguendo la formula 10s + (4s - 1s) · 4s = 22s.



Ora si analizza il caso del parallelismo spaziale di tipo map: vi è un dato che viene diviso in pezzi e ogni pezzo viene trattato da un worker diverso che impiegano un certo *t*, e dopo questo tempo il dato viene ricostruito. Il tempo di completamento è dato dal T_{split} per dividere il dato + il $\frac{T_{seq}}{n}$ dove ognuno dei operai fa il suo lavoro + il T_{merge} per ricostruire il dato. Se il T_{split} e T_{merge} sono trascurabili rispetto a *t* allora $\boldsymbol{T_c \simeq \frac{T_{seq}}{n}}$, mentre $\boldsymbol{T_s = max\{T_{split}, \frac{T_{seq}}{n}, T_{merge}\}}$.



Nel caso di parallelismo spaziale di tipo farm si immagina ci sono un certo numero di task $\{x_1,...,x_m\}$, ogni task viene mandato ai workers in *round-robin* e ogni lavoratore impiega un certo *t*. Il $\boldsymbol{T_c = \frac{m}{n} \cdot t = m \cdot T_s}$ e il tempo di servizio coincide a $\boldsymbol{T_s = max\{T_{sched}, T_{coll}, \frac{t}{n}\}}$, dove T_{sched} serve per assegnare un task in ingresso ad un worker e T_{coll} per raccogliere un valore calcolato da un worker e spedirlo in uscita al farm. Questo è il procedimento che viene usato nei processori superscalari.



Chapter 6

Microarchitetture

6.1 Introduzione

Si è trattato: *logica binaria* per rappresentare le informazioni, *reti combinatorie* e *reti sequenziali* per elaborare le informazioni e *assembler ARM* per programmare a basso livello. Ora mettendo tutto insieme in un disegno logico in cui si usa delle reti combinatorie e/o sequenziali che funzionano con la logica di boole e che sono in grado di fornire un interprete per l'assembler ARM, cioè saranno in grado di eseguire il ciclo:

```
while(true){
   fetch
   decode
   execute + update(PC)
   writeback
}
```

6.2 Assembler ARM: formato istruzioni

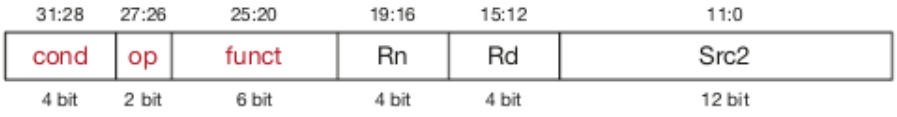
Il formato istruzioni rispetta i principi della progettazione *RISC* cioè:

- la regolarità favorisce la semplicità;
- rendere veloci le cose frequenti;
- più è piccolo è più veloce;
- un buon progetto richiede buoni compromessi.

ARM sceglie il compromesso di avere tre formati principali per le istruzioni: operative, accesso a memoria e salti. Questo numero limitato di formati consente di avere una certa regolarità tra le varie istruzioni, quindi di semplificare i circuiti.

6.2.1 Istruzioni operative

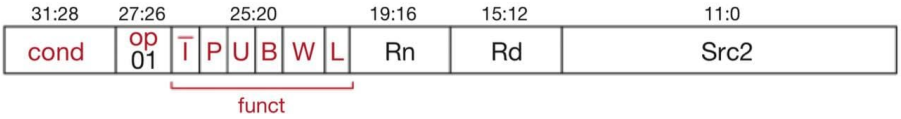
L'istruzione a 32 bit ha sei campi: *cond, op, funct, Rn, Rd e Src2*.



I primi 4 bit sono la condizione secondo cui l'istruzione deve essere eseguita, per esempio in una istruzione condizionale MOVEQ, EQ corrisponde a una configurazione di bit che verrà scritta all'interno di *cond*. Per le istruzioni non condizionate si avrà *cond* = 1110_2. *Op* indica il tipo di istruzione: operativa (00), accesso a memoria (01) o istruzione di salto (10). I seguenti 6 bit di *funct* indicano quale istruzione, e ha tre sottocampi: I, cmd e S.
Il bit I vale 1 quando *Src2* è un immediato, cmd indica la specifica operazione da svolgere, per esempio ADD = 0100_2 o SUB = 0010_2 e il bit S vale 1 quando l'istruzione imposta la flag di condizione, come SUBS. Gli operandi sono codificati in tre campi: *Rn* è il registro del primo operando sorgente, *Src2* è il secondo operando sorgente e *Rd* è il registro destinazione. Src2 consente di avere come secondo operando: un immediato o un registro eventualmente traslato.

6.2.2 Istruzioni accesso a memoria

Le istruzioni di accesso a memoria usano un formato simile a quello delle istruzioni di eleborazione dati, con gli stessi sei campi *cond, op, funct, Rn, Rd e Src2* ma hanno una codifica diversa.



I primi 4 bit di *cond* funzionano nello stesso modo come nelle istruzioni operative e *op* = 01. Il campo *funct* è costituito da 6 bit di controllo: i due bit Ī e U determinano se l'offset è un immediato o un registro, e se deve essere sommato o sottratto; i due bit P e W specificano il modo di gestione indice (pre/post indice); i due bit L e B specificano il tipo di accesso a memoria: STR, STRB, LDR, LDRB. Rn è il registro base, Rd è il registro destinazione e Src2 costituisce l'offset.

6.2.3 Istruzioni salto

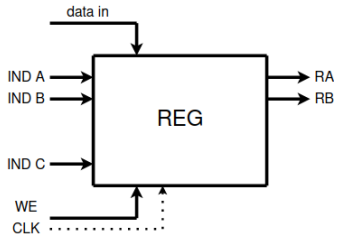
Come per le istruzioni operative e di accesso alla memoria, anche i salti cominciano con un campo a 4 bit per l'esecuzione condizionata e un campo *op* = 10.

Il campo *funct* è di soli 2 bit dove il più significativo è sempre 1 mentre il meno significativo indica il tipo di salto: 0 per B e 1 per BL. I rimanenti 24 bit di *imm24* serve a specificare l'indirizzo dell'istruzione alla quale saltare.

6.3 Progettazione

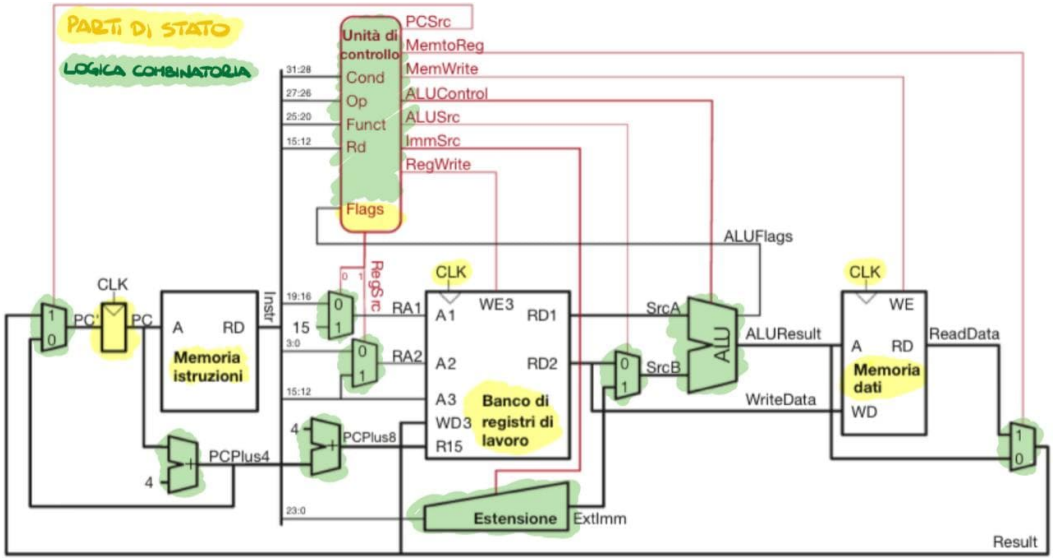
Lo **stato architetturale** del processore ARM è definito dal contenuto di 16 registri a 32 bit e di un registro di stato, quindi ogni microarchitettura ARM deve poter memorizzare per intero questo stato. Un buon modo per iniziare il progetto di un microprocessore è quello di partire dall'hardware necessario:

- **Unità registri**: è una memoria con 2 ingressi, IND A e IND B per leggere dei dati e un ingresso IND C per scrivere dei dati. Gli indirizzi a 4 bit sono in grado di accedere ai 16 registri, in questo modo 2 registri possono essere consultati e un registro può essere scritto contemporaneamente. Insieme a questi indirizzi vi è un segnale di WE e di clock. Per il fatto di avere più indirizzi in lettura e scrittura la si chiama memoria **multiporta**.



- **Program counter**: è un registro e possiede il segnale di clock e di WE. Serve per mantenere lo stato, la sua uscita punta all'istruzione corrente, mentre il suo ingresso è l'indirizzo della prossima istruzione da eseguire. PC è anche il registro R15 quindi ci dovrà essere una copia dentro l'unità registri;
- **Memoria**: si ha bisogno di una **memoria istruzioni** per il fetch dell'istruzioni e una **memoria dati** per le LDR e STR. La memoria istruzioni ha una sola porta di lettura (ROM). Riceve in ingresso un indirizzo di istruzione a 32 bit, ed emette sull'uscita di lettura il dato a 32 bit, cioè MI[PC]. La memoria dati ha una sola porta di lettura/scrittura, se viene attivato il suo segnale di WE, il dato in ingresso viene scritto;
- **Unità di controllo**: è in gran parte una rete combinatoria, ma la parte che gestisce i flag è stato interno. L'unità di controllo genera i segnali di controllo sulla base dei campi *cond, op e funct*, tutti i segnali per i multiplexer, abilitazione dei registri, segnali di lettura/scrittura in modo opportuno a seconda di quale operazione sta eseguendo.

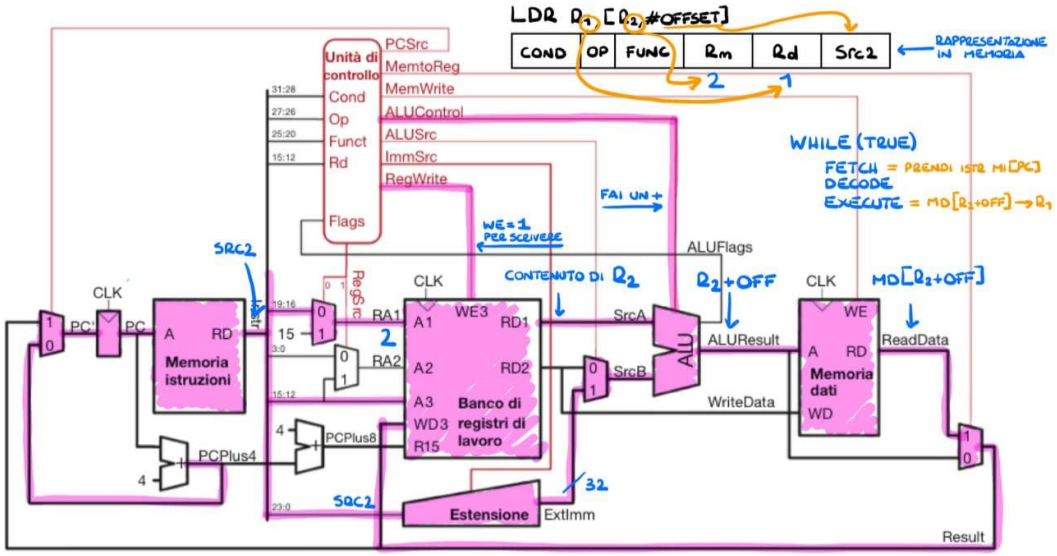
Il processore è costituito da un insieme di componenti hardware di **stato interno** (registri o memorie) e **logica combinatoria**. L'unità registri, la memoria dati e la memoria istruzioni sono stato interno e i primi due possiedono un segnale di clock (anche la memoria istruzioni lo ha, ma in questo contesto non serve), le altre parti sono logica combinatoria.



6.4 Processore single cycle

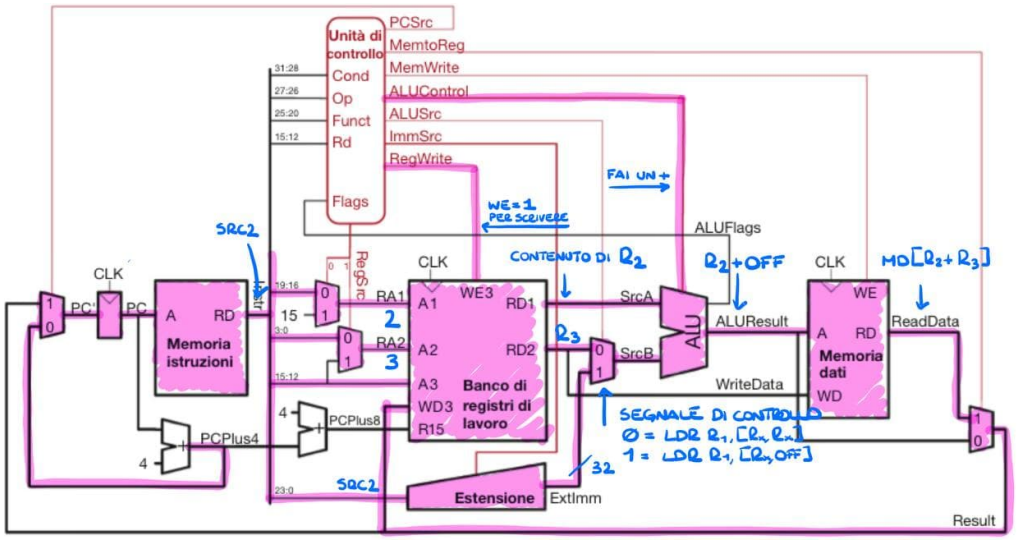
La microarchitettura single cycle esegue un'intera istruzione in un ciclo di clock. Il tempo del clock è imposto dall'istruzione più lenta (LDR) e il processore richiede memoria istruzioni e memoria dati separate. Si suppone di voler eseguire:

- LDR R1, [R2, #OFFSET]



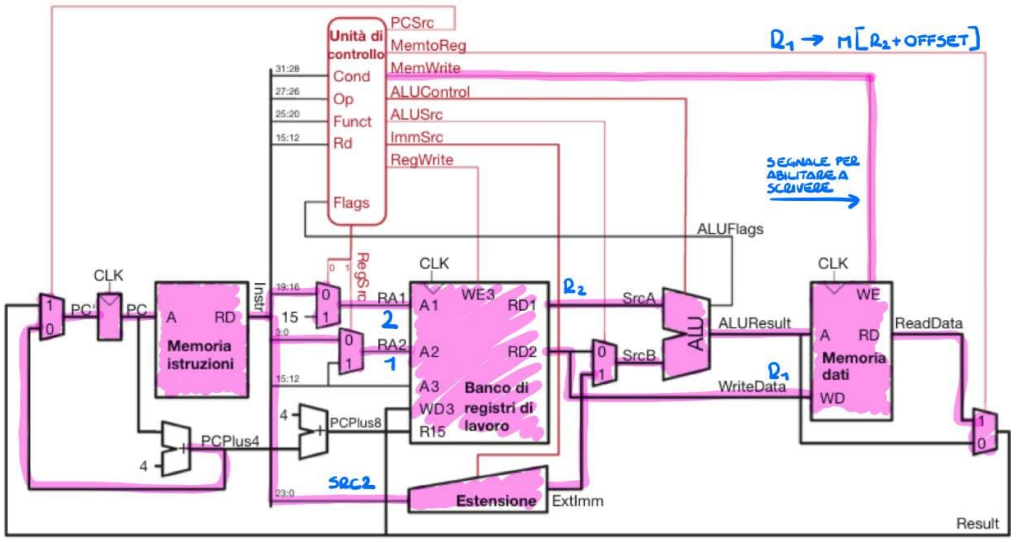
Il PC contiene l'indirizzo dell'istruzione da eseguire. Si legge l'istruzione dalla memoria istruzioni, cioè si esegue il fetch. Sul primo indirizzo dell'unità registri si ha 2, e quindi sulla prima uscita si ottiene il contenuto di R2. L'offset è una parte di *Src2* (al max 12 bit) e viene mandato in un circuito che estende i bit a 32. A questo punto la ALU controllata dall'ALUControl decide di fare la somma e calcola l'indirizzo R2 + offset. Il WE della memoria dati è 0 (quindi il dato non viene scritto) e all'uscita si ha MD[R2 + offset] che è il valore che verrà memorizzato nel registro R1 all'interno dell'unità registri abilitando il segnale di WE. Ora per ricominciare il ciclo del while, dato che non è una operazione di salto basta fare PC + 4.

- LDR R1, [R2, R3]



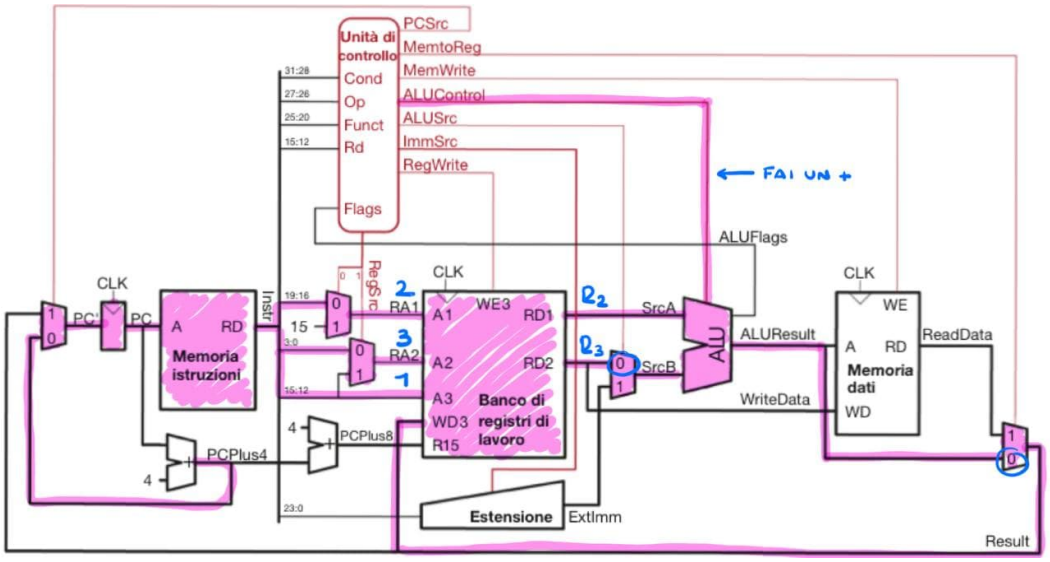
In questa variante occorre che sul secondo ingresso della ALU ci sia un multiplexer che sceglie tra R3 e l'estensore, con segnale di controllo che può essere 0 nel caso in cui il secondo sorgente sia un registro, o 1 nel caso in cui sia un immediato.

- STR R1, [R2, #OFFSET]



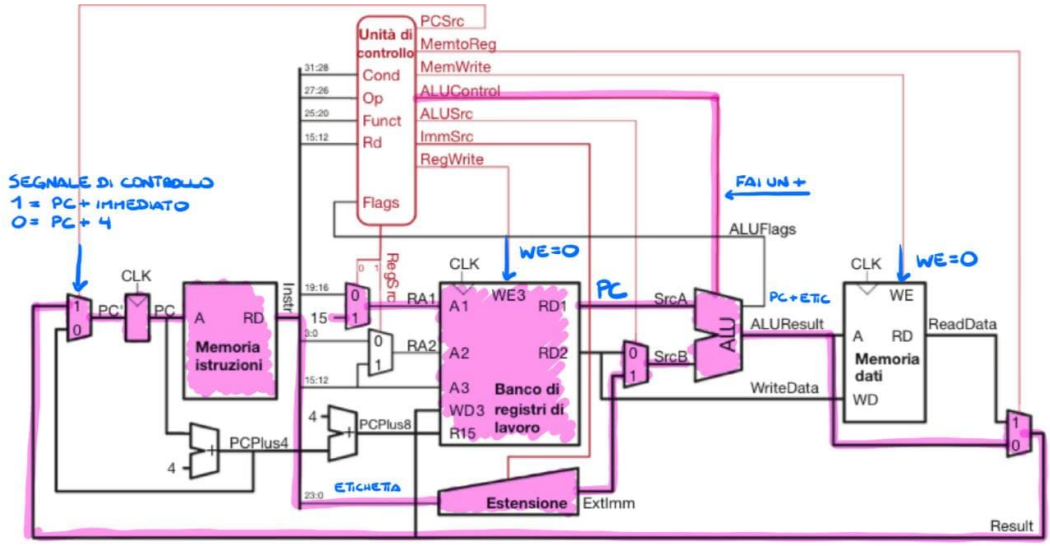
La STR è l'unico caso in cui il regsitro destinazione lo si utilizza come sorgente. Nella memoria dati è necessario indicare cosa deve scrivere, cioè *Rd*, e abilitare il segnale di WE, che è controllato dal segnale MemWrite. Allora sulla seconda uscita dell'unità registri vi è il valore che deve essere scritto dentro la memoria dati (il contenuto di R1).

- ADD R1, R2, R3



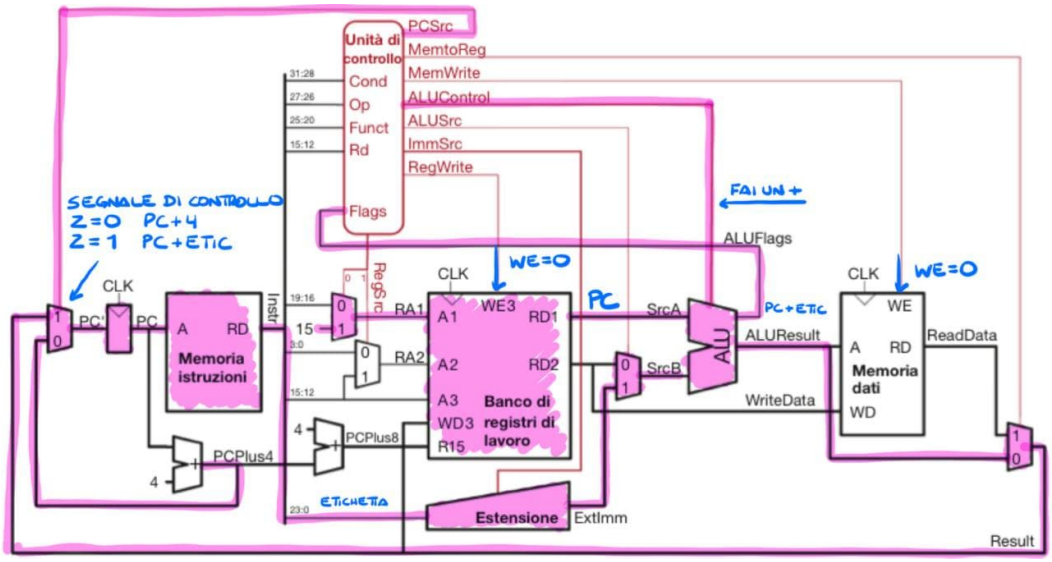
R2 e R3 sono i primi due ingressi dell'unità registri e R1 come ingresso di scrittura. La ALU fa la somma di R2 e R3, che non passa dalla memoria dati, arriva in un multiplexer con segnale di controllo *op* = 00 se esegue una operazione operativa, altrimenti se *op* = 01 se esegue una istruzione di accesso alla memoria (in questo caso solamente LDR), e infine scrive il risultato nell'unità registri.

- B ETICHETTA



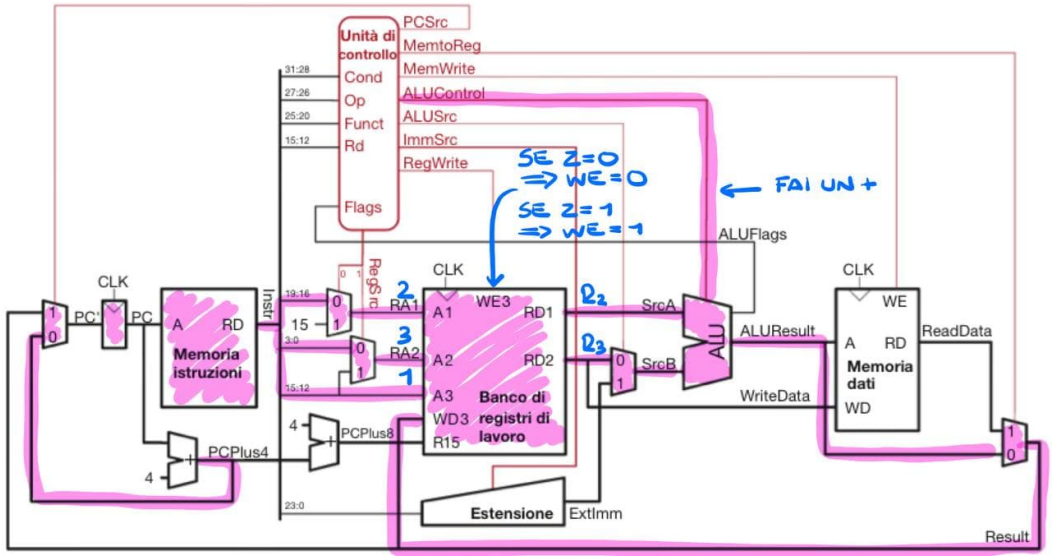
Nel caso di una istruzione di salto si deve fare PC + ETICHETTA. Il valore del PC è R15 che è all'interno anche dell'unità registri, quindi da quest'ultima esce il PC. L'immediato da 24 bit si estendono a 32 tramite l'extend. La ALU fa la somma di PC + ETICHETTA e il risultato arriva al multiplexer davanti al PC con segnale di controllo PC + 4 nel caso di una istruzione operativa o PC + IMMEDIATO nel caso di una istruzione di salto. Il risultato andrebbe anche come ingresso di scrittura dell'unità di registri, ma il WE=0 e quindi non viene scritto.

- BEQ ETICHETTA



I flag prodotti dalla ALU vanno a finire nell'unità controllo. L'istruzione è tutta come una B "normale", ma essendo una EQ significa che se Z=0 allora occorre fare PC + 4, se Z=1 invece PC + ETICHETTA, quindi il risultato della ALU che deve essere mandato al PC deve essere controllato da un segnale PCSrc che arriva dall'unità di controllo.

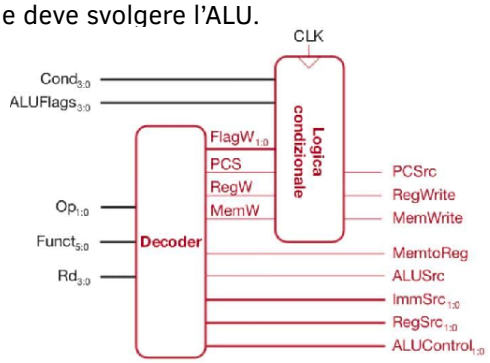
- ADDEQ R1, R2, R3



L'indirizzo dell'istruzione si prende dal PC. Sul primo indirizzo dell'unità registri si ha 2 e sul secondo indirizzo 3, quindi all'uscita avremmo il contenuto di R2 e R3. La ALU farà la somma R2+R3, non passa dalla memoria dati e al momento di scrittura nell'unità di registri, se il flag Z è uguale a 0 il segnale WE dell'unità registri sarà 0, se Z=1 allora significa che deve essere scritto e quindi WE = 1. L'istruzione viene sempre eseguita poi in base ai flag il risultato può essere scritto oppure no.

6.4.1 Unità di controllo

L'unità di controllo è suddiviso in due parti: la **logica condizionale** mantiene i flag e decide se l'istruzione condizionale va scritta nel PC (PCSrc), nell'unità registri (RegWrite) o nella memoria dati (MemWrite); il **decoder** genera i segnali di controllo in base al il tipo di istruzione (*op*), quale istruzione (*funct*), il registro di destinazione (*Rd*), e manda alcuni segnali alla logica condizionale in base ai flag e imposta i segnali di controllo per i multiplexer, l'estensore e decide quale operazione deve svolgere l'ALU.



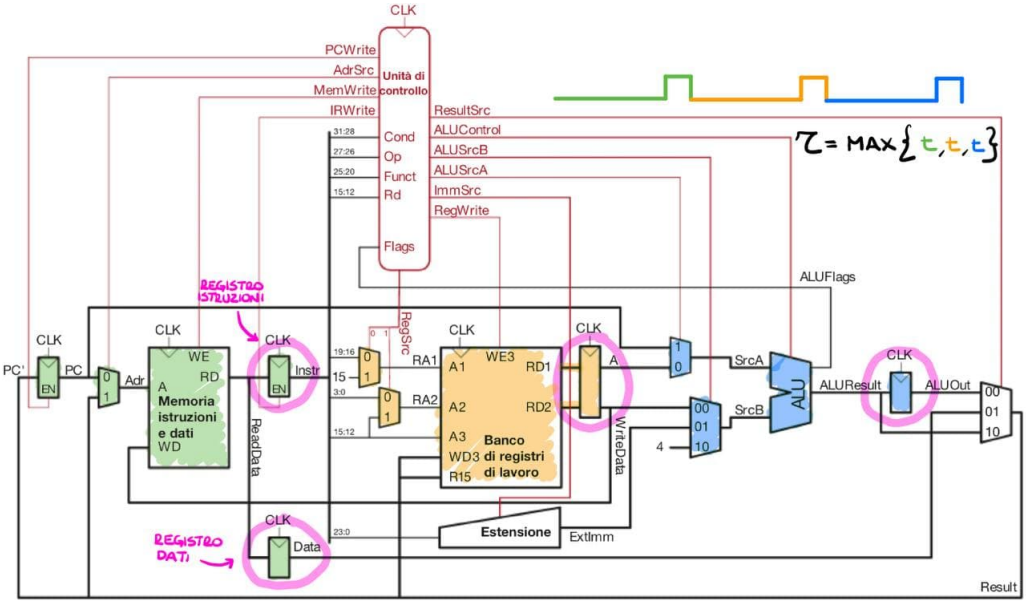
6.4.2 Analisi delle prestazioni single cycle

Ogni istruzione occupa un ciclo di clock, quindi la CPI (clock per istruzione) vale 1. Il percorso per l'istruzione più lunga, LDR, è di circa 840 ps che sono circa 1 ns. Altre istruzioni hanno percorsi più brevi, come le istruzioni operative che non hanno bisogno di accedere alla memoria dati, ma in ogni caso l'accesso alla memoria dati si deve contare perchè il periodo di clock deve essere costante e dimensionato sull'istruzione più lenta.

Elemento	Parametro	Ritardo (ps)
Registro: clock-uscita	t_{pcq}	40
Registro: *setup*	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Lettura da memoria	t_{mem}	200
Lettura dal banco di registri	t_{RFread}	100
Setup del banco di registri	$t_{RFsetup}$	60

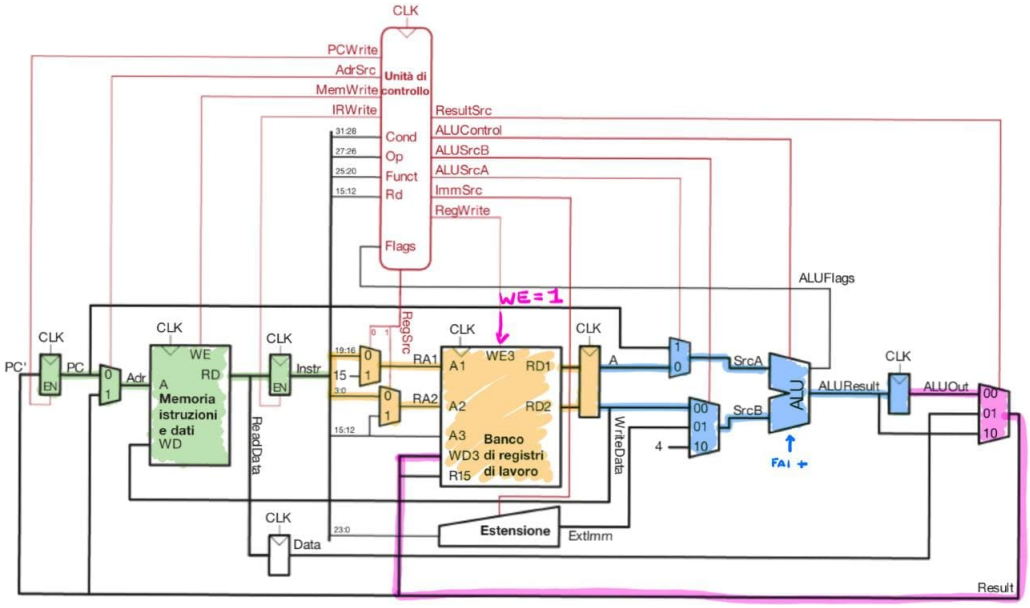
6.5 Processore multicycle

Il processore single cycle ha degli elementi di debolezza come il fatto che richiede memorie separate per istruzioni e dati, mentre la maggior parte dei processori ha una sola memoria esterna che contiene entrambi. Richiede un ciclo di clock abbastanza lungo da consentire l'esecuzione dell'istruzione più lenta anche se molte istruzioni potrebbero essere più veloci. Il processore multicycle elimina queste debolezze dividendo l'istruzione in una sequenza di passi più brevi.



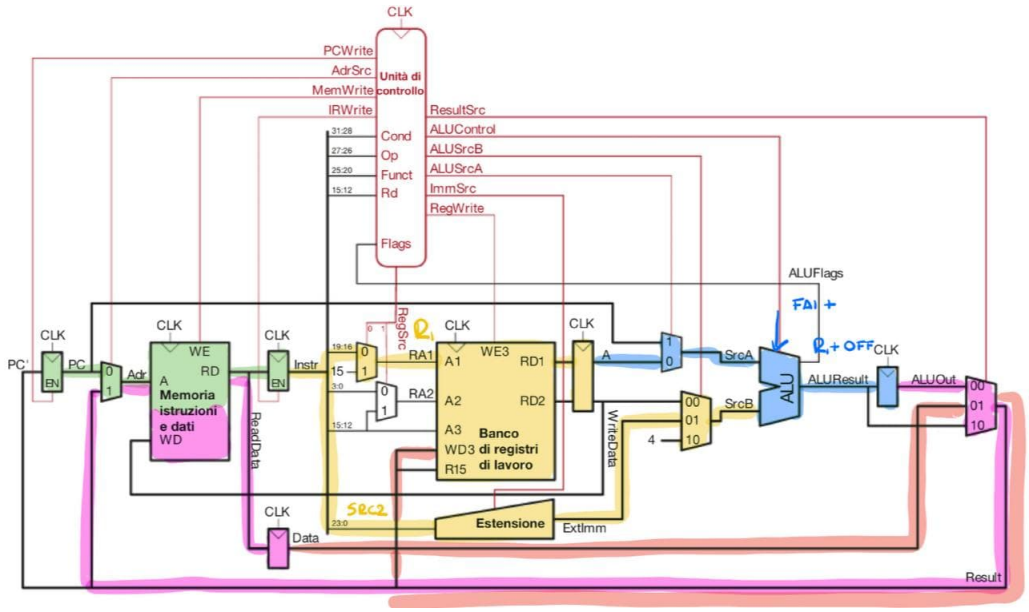
L'istruzione viene letta da memoria in un ciclo, e i dati possono essere letti o scritti in cicli successivi, quindi è possibile usare una sola memoria per contenere entrambi. Istruzioni diverse usano numeri diversi di cicli, quindi le istruzioni più semplici vengono portate a termine in meno tempo rispetto a quelle più complesse. L'istruzione viene letta e memorizzata nel registro istruzioni in modo da renderla disponibile nei cicli successivi. I contenuti dell'unità registri e il risultato della ALU vengono memorizzati in un registro. Per esempio nel primo ciclo di clock si svolge la lettura di un'istruzione dalla memoria, cioè il fetch e scrive l'istruzione all'interno del registro istruzioni. Nel secondo ciclo di clock l'istruzione viene letta, si entra nell'unità registri e si scrive il contenuto dei due registri nel registro accanto all'unità registri. Nel terzo ciclo, la ALU lavora e scrive il risultato nel registro. Tutte queste operazioni non vengono calcolate tutte insieme, ma il ciclo di clock sarà calcolato come il max tra i tre cicli. Nel multicycle vi sono tanti cicli di clock brevi invece che un singolo ciclo di clock lungo come nel caso del single cycle, e quindi mediamente il multicycle è più veloce.

- ADD R0, R1, R2



L'istruzione viene letta e memorizzata nel registro istruzioni (1 ciclo); vengono letti i registri R1 e R2 e memorizzati nel registro (2 ciclo); lavora la ALU, cioè calcola R1 + R2 e conserva il risultato in un registro (3 ciclo); e infine si scrive il risultato R0 = R1 + R2 nell'unità registri abilitando il WE (4 ciclo). In questi 4 cicli di clock, l'unità di controllo ha dovuto dare dei segnali diversi per ogni ciclo.

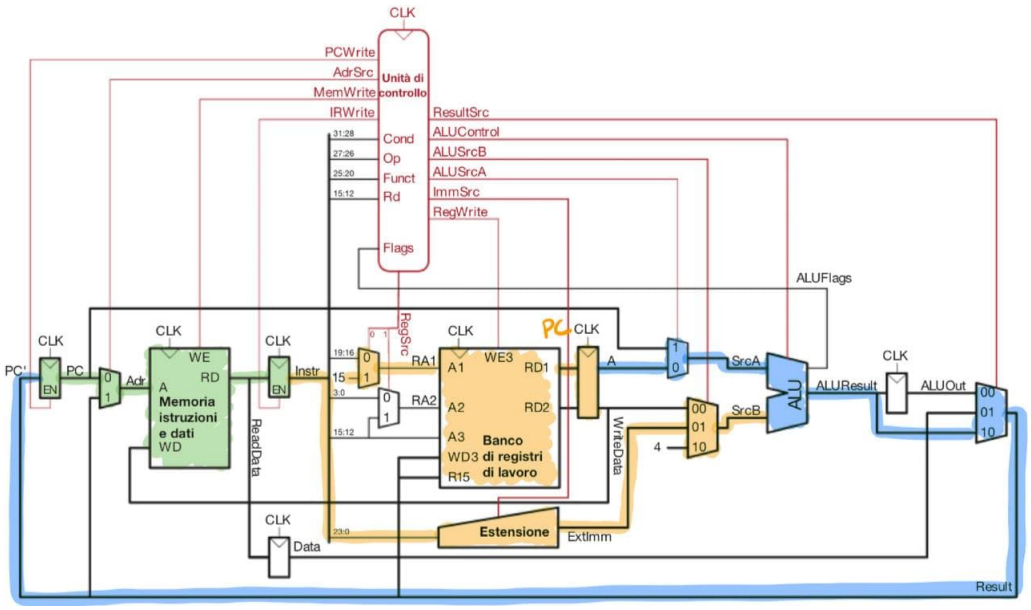
- LDR R0, [R1, #OFFSET]



L'istruzione viene letta e memorizzata nel registro istruzioni (1 ciclo); viene letto il registro R1, memorizzato nel registro e vengono estesi i 12 bit di *Src2*. L'extend è una funzione combinatoria, non varia durante l'esecuzione

dell'istruzione corrente, quindi non serve un registro dedicato (2 ciclo); lavora la ALU, cioè calcola R1 + OFFSET e conserva il risultato (3 ciclo); carica il risultato appena calcolato nella memoria, il multiplexer davanti alla memoria serve appunto per selezionare l'indirizzo dal PC o dal risultato della ALU (4 ciclo). Il dato letto viene memorizzato nel registro dati e infine il dato viene scritto nell'unità registri (5 ciclo).

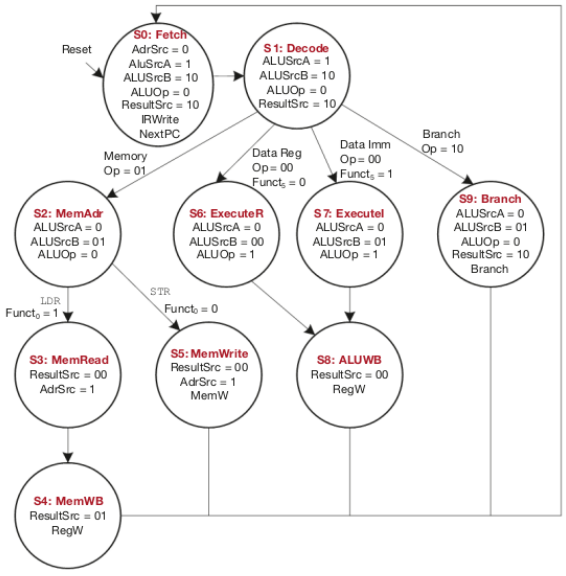
- B ETICHETTA



L'istruzione viene letta e memorizzata nel registro istruzioni (1 ciclo); L'immediato da 24 bit vengono estesi a 32 bit tramite l'extend, viene letto R15 e viene conservato in un registro. Quindi la ALU ha pronto un solo ingresso (2 ciclo). Il valore del PC arriva alla ALU che calcola PC + ETICHETTA e il risultato va scritto come ingresso del PC. (3 ciclo).

6.5.1 Unità controllo multicycle

L'unità di controllo, come per il single cycle, genera i segnali di selezione dei multiplexer, abilitazione dei registri, e flag di stato opportuni ma nel processore multicycle l'unità di controllo è un componente molto più complesso per inviare i segnali nei cicli opportuni e corrisponde ad un automa a stati finiti. Per esempio nel caso della ADD vi è il fetch dell'istruzione, quindi la selezione del primo multiplexer è 0 e il segnale di WE=1 nel registro istruzioni. In tutto il resto del processore non deve succedere nulla, cioè WE dell'unità registri e degli altri registri devono essere uguale a 0. Si entra nello stato di decode, si leggono i registri dall'unità registri e si scrive all'interno del registro con WE=1, mentre il WE del registro istruzioni e della memoria deve essere uguale a 0. In questa fase il processore realizza che è una istruzione operativa. Ora lavora la ALU facendo la somma e manda come segnale 10 al multiplexer per scrivere il risultato nell'unità registri, quindi con WE=1. In questo schema, che è una versione semplificata rispetto al precedente, si hanno bisogno di 3 cicli di clock per svolgere una ADD.

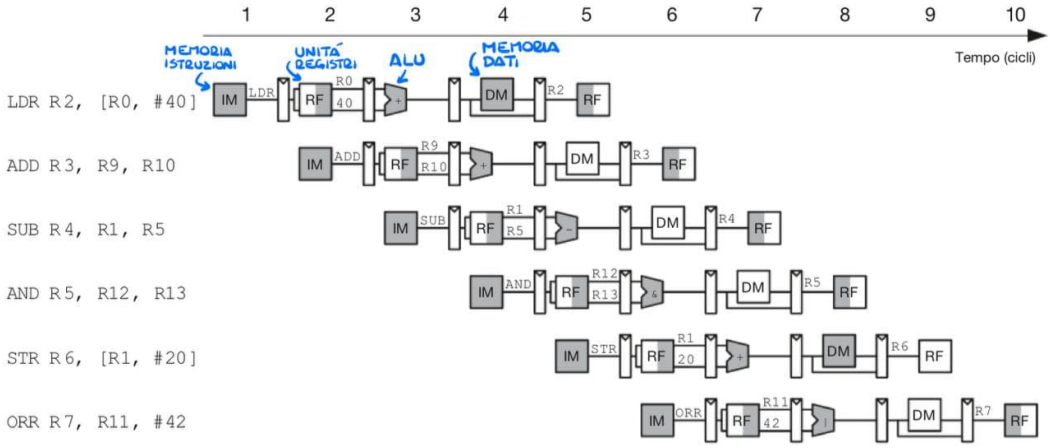


6.5.2 Analisi delle prestazioni multicycle

Il single cycle esegue tutte le istruzioni in un solo ciclo, il multicycle usa più cicli per le diverse istruzioni, però svolge meno attività, quindi ha un tempo inferiore $T_c = t_{REGclock} + 2t_{MUX} + max\{t_{ALU} + t_{MUX} + t_{MEM}\} + T_{REGsetup}$. Il CPI dipende dalla probabilità relativa di utilizzo di ciascuna tipologia di istruzione nel programma perchè le istruzioni operative impiegano 4τ, invece le istruzioni di salto 3τ e infine le istruzioni di accesso alla memoria 5τ. Il multicycle è meno costoso perché unisce la memoria istruzioni e dati, ed elimina due circuiti sommatori ma introduce cinque registri non architetturali e alcuni multiplexer aggiuntivi.

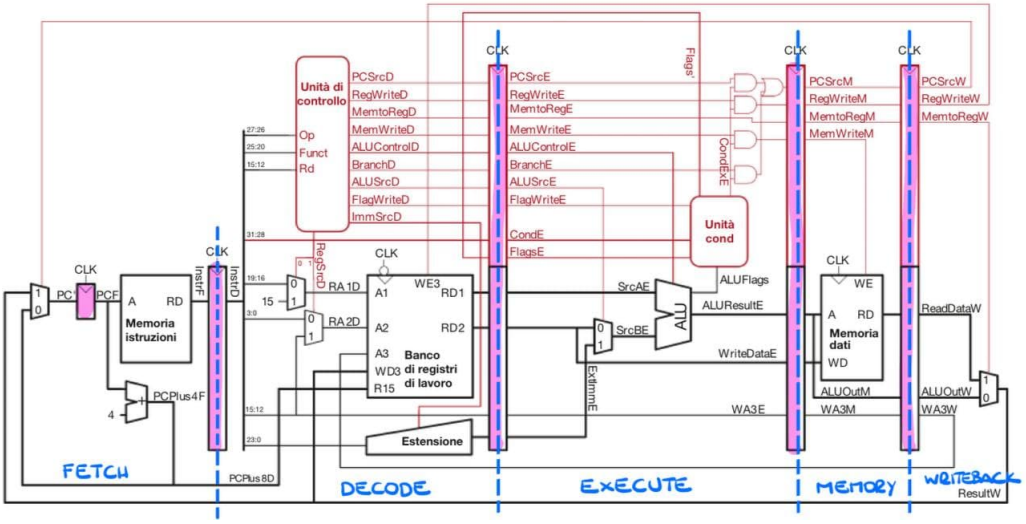
6.6 Processore pipeline

Si vuole cercare di adattare il processore single cycle dividendolo in cinque stadi. In questo modo, cinque istruzioni alla volta possono essere in esecuzione, una per ogni stadio. Ci sono 5 stadi denominati Fetch, Decode, Execute, Memory, Writeback. Sono simili ai cinque cicli di clock che il processore multicycle svolge per l'istruzione LDR. Nello stadio Fetch il processore legge l'istruzione dalla memoria; nello stadio Decode legge gli operandi dall'unità registri e decodifica l'istruzione; nello stadio Execute esegue i calcoli con l'ALU; nello stadio Memory legge o scrive i dati in memoria; infine nello stadio Writeback scrive il risultato nell'unità registri. L'unità di controllo usa gli stessi segnali di controllo del processore single cycle e tali segnali devono essere propagati nella pipeline insieme ai dati per rimanere sincronizzati con l'istruzione cui si riferiscono. Gli stadi sono colorati di grigio quando vengono effettivamente utilizzati:



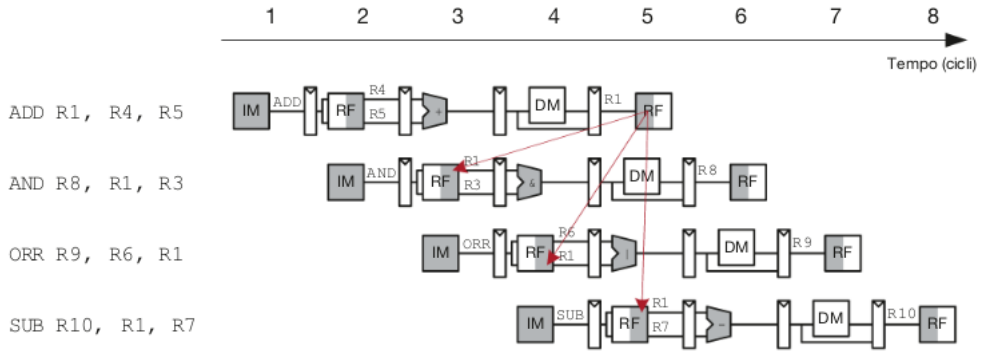
6.6.1 Percorso dati pipeline

Il processore pipeline è ottenuto dividendo il percorso dati del processore single cycle in cinque stadi separati da registri di pipeline, anche i segnali di controllo sono mediati da registri per salvare i valori. L'unità registri è un caso particolare, perché viene letto nello stadio Decode e scritto nello stadio Writeback. Questo collegamento all'indietro è la causa delle *dipendenze*. Uno degli aspetti critici nelle pipeline è il fatto che tutti i segnali associati a una particolare istruzione devono procedere all'unisono attraverso la pipeline. L'unità cond nello stadio execute confronta i vecchi valori dei flag. Per esempio in una istruzione ADDEQ, va eseguita in base a una CMP precedente.

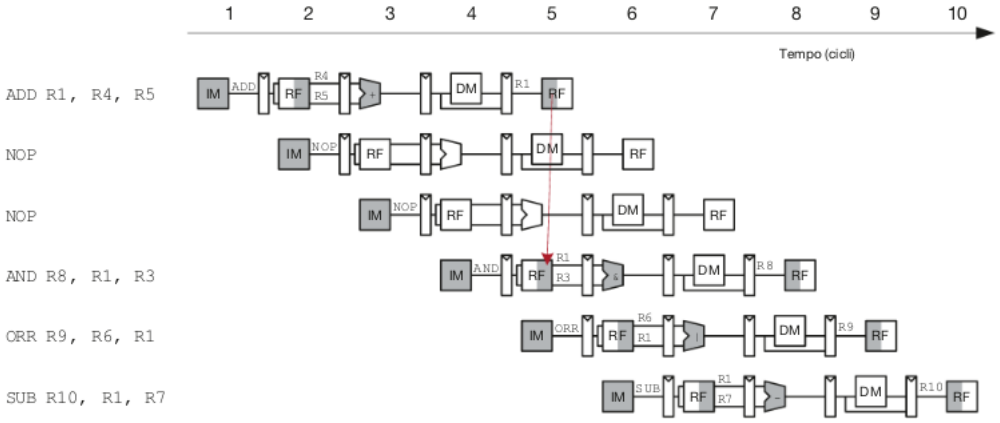


6.6.2 Dipendenze

Un grosso problema presente nei sistemi pipeline è la gestione delle dipendenze (hazards). Una **dipendenza** si ha quando un'unità deve leggere un dato che ancora non è stato prodotto da un'altra unità.

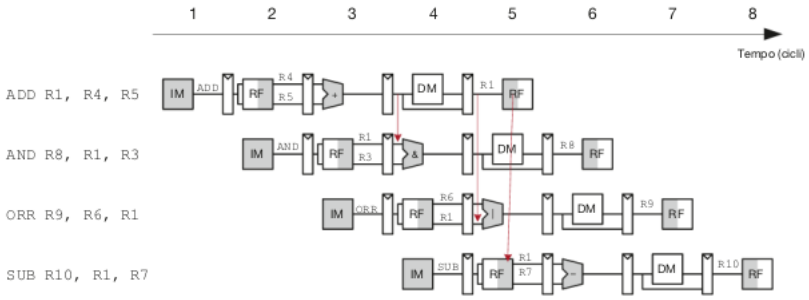


Quando una istruzione richiede un dato che non è stato ancora calcolato, questo tipo di dipendenza viene denominato **RAW** (read after write). L'istruzione ADD scrive il proprio risultato in R1 nella prima metà del ciclo 5, ma l'istruzione AND legge R1 nel ciclo 3, ottenendo quindi un valore scorretto; l'istruzione ORR legge R1 nel ciclo 4, ottenendo anche lei un valore scorretto; invece l'istruzione SUB legge R1 nella seconda metà del ciclo 5, ottenendo il valore corretto che era stato scritto in R1 nella prima metà dello stesso ciclo, e come lei tutte le istruzioni successive. Le dipendenze sono classificate in **dipendenze di dato** cioè si verifica quando un'istruzione vuole leggere un registro che non è stato ancora aggiornato da un'istruzione precedente, e **dipendenze di controllo** che si verificano quando la decisione di quale istruzione debba essere prelevata da memoria nella fase di fetch non è ancora stata presa al momento del fetch. Una soluzione a livello software sarebbe quello di inserire istruzioni **NOP** (istruzione che non fa nulla) tra ADD e AND in modo che l'istruzione che ha dipendenza non vada a leggere R1 finché non sia stato reso disponibile nell'unità registri. Questo approccio complica la programmazione e degrada le prestazioni.



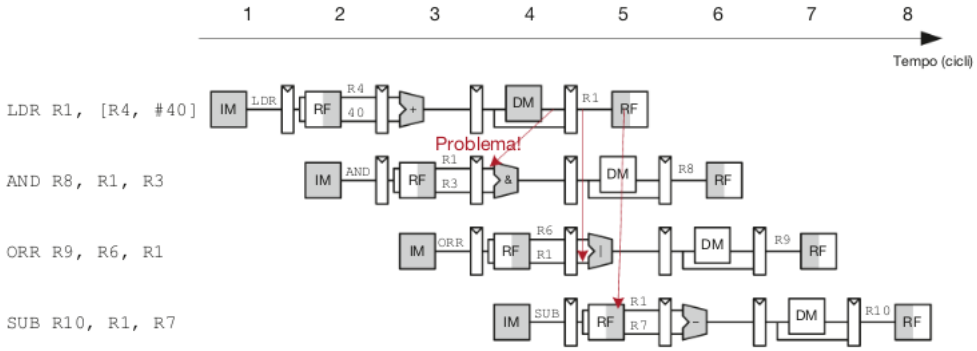
Gestione dipendenze di dato
Ci sono due modi:

- **Forwarding**: le istruzioni operative possono essere risolte mediante la tecnica dell'inoltro di un risultato disponibile negli stadi Memory o Writeback all'istruzione che ha dipendenza che si trova nello stadio Execute. In questo modo non è necessario salvare un valore in un registro e quindi leggerlo al ciclo dopo, ma si cerca di fare un *bypass* dei segnali rendendoli disponibili un po' prima.

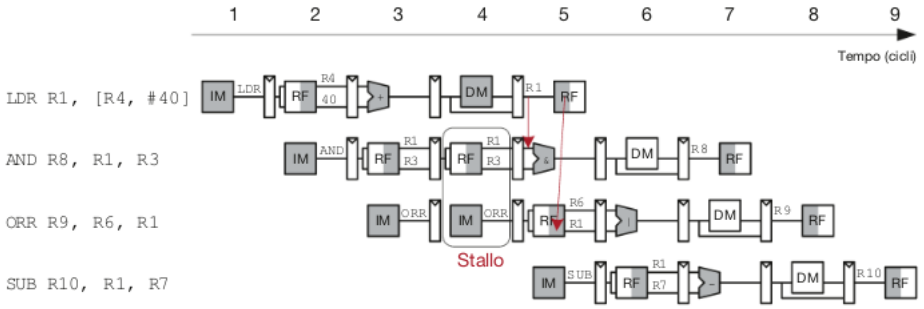


Nel ciclo 4, R1 è inoltrato dallo stadio Memory dell'istruzione ADD allo stadio Execute dell'istruzione dipendente AND; nel ciclo 5, R1 è inoltrato dallo stadio Writeback dell'istruzione ADD allo stadio Execute dell'istruzione dipendente ORR. L'inoltro è necessario quando un'istruzione nello stadio Execute ha un registro sorgente coincidente con il registro destinazione delle istruzioni nello stadio Memory oppure Writeback. Questo richiede l'aggiunta di due multiplexer di inoltro e di un'**unità di gestione delle dipendenze** che genera i segnali per i multiplexer di inoltro per selezionare gli operandi dall'unità registri oppure dai risultati negli stadi Memory o Writeback. L'impatto sul clock di aggiungere un multiplexer dipende da dove lo si inserisce perchè il clock si basa sulla fase più lunga che solitamente sono gli accessi in memoria.

- **Stallo**: il forwarding è sufficiente a risolvere le dipendenze di quando il risultato viene calcolato nello stadio Execute di un'istruzione, perché può essere inoltrato allo stadio Execute dell'istruzione successiva. Sfortunatamente l'istruzione LDR termina di leggere il dato alla fine dello stadio Memory, per cui il risultato non può essere inoltrato allo stadio Execute dell'istruzione successiva. Si può dire che l'istruzione LDR ha una latenza di due cicli, perché un'istruzione che dipenda da lei può avere i suoi risultati solo due cicli dopo.



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.

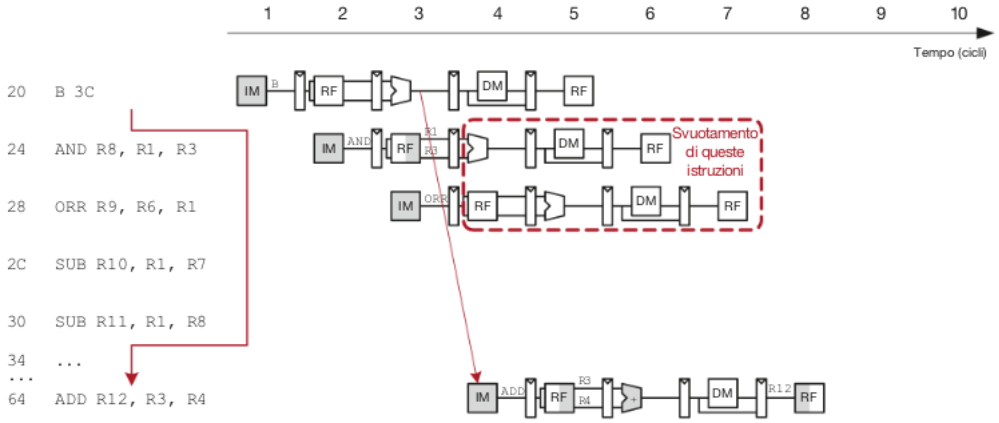


AND entra nello stadio Decode nel ciclo 3 e viene mantenuta in stallo nel ciclo 4. L'istruzione successiva ORR deve a sua volta rimanere per due cicli nello stadio Fetch perché lo stadio Decode è occupato. Nel ciclo 5 il risultato può essere inoltrato dallo stadio Writeback di LDR allo stadio Execute 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 e si comporta come un'istruzione NOP. 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. Il registro di pipeline subito dopo lo stato portato in stallo deve essere svuotato per evitare che informazioni fasulle si propaghino nella pipeline. Lo stallo degrada le prestazioni, quindi va usato solo quando strettamente necessario.

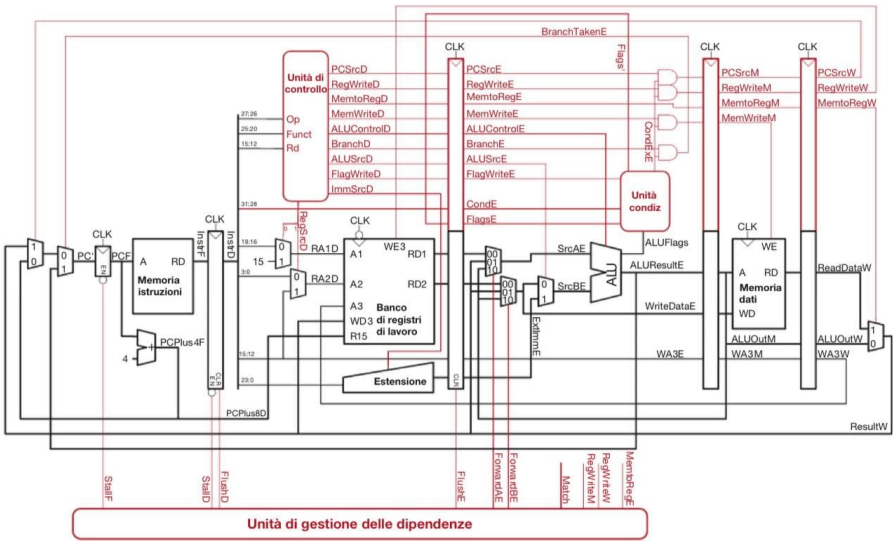
Gestione dipendenze di controllo

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 stata ancora presa al momento di tale fetch. Le scritture nel PC presentano lo stesso tipo di dipendenza. Un modo per gestire le dipendenze di controllo è mandare in stallo la pipeline fino a quando la decisione circa il salto sarà stata presa. Dal momento che tale decisione viene presa nello stadio Writeback, la pipeline deve essere messa in stallo per quattro cicli a

ogni istruzione di salto, con conseguente notevole degrado di prestazioni. L'alternativa è prevedere se il salto dovrà essere fatto oppure no, e cominciare l'esecuzione delle istruzioni sulla base di tale previsione: se il salto andava fatto, le quattro istruzioni successive al salto devono essere scartate, queste istruzioni da eliminare sono la cosidetta penalizzazione per salto mal previsto.



Si può ridurre la penalizzazione per salto mal previsto se si riesce ad anticipare la decisione se saltare o meno, perchè tale decisione viene presa nello stadio Execute quando la destinazione di salto è già stata calcolata.



Il multiplexer davanti al PC seleziona il risultato dalla ALU con un segnale di bypass (risparmiando due cicli di clock) o il valore del multiplexer precedente. Un istruzione di salto quindi ha una latenza di 3 cicli di clock. L'unità di gestione delle dipendenze gestisce l'enable della scrittura del PC, registro della memoria istruzioni e dell'unità registri in modo che possa effettuare uno stallo in base a una decisione dello stadio Execute, fino a che alcuni risultati non sono arrivati allo stadio Writeback.

Conclusione sulle dipendenze

Un altro modo per evitare le dipendenze consiste nel modificare il codice per fare in modo che ci siano meno dipendenze, ed è un procedimento che può essere eseguito sia dal programmatore che dal compilatore. Per esempio:

```
MOV R0, #0
ADD R1, R2, R3     ; dipendenza
SUB R4, R1, R5     ; dipendenza
```

è equivalente ad

```
ADD R1, R2, R3
MOV R0, #0
SUB R4, R1, R5
```

Questo procedimento non cambia la semantica del programma. In generale vi sono le **3 condizioni di Berneistain** cioè se vi è una istruzione A e una istruzione B, si considera:

- R1 gli insiemi letti dall'istruzione A e W1 gli insiemi scritti dall'istruzione A;
- R2 gli insiemi letti dall'istruzione B e W2 gli insiemi scritti dall'istruzione B;

allora istruzione A seguito dall'istruzione B è equivalente a istruzione B seguito dall'istruzione A se e solo se:

- $R1 \cap W2 = 0$ AND
- $W1 \cap R2 = 0$ AND
- $W1 \cap W2 = 0$

cioè se la prima istruzione non scrive qualcosa che viene letta dalla seconda istruzione, se la prima istruzione non legge qualcosa che viene scritto dalla seconda istruzione e se entrambe le istruzioni non scrivono nello stesso registro. Si analizza il costo di un loop in un processore pipeline:

```
FOR:  CMP R8, R6
      BEQ FINE            ; preso solo alla fine del ciclo
      LDRB R0, [R4, R8]
      MOV R1, R5
      BL CIFRA            ; preso sempre
      LDRB R2, [R4, R8]
      CMP R0, R2
      BEQ CONT          ;    un if
      STRB R0, [R4, R8]   ; 50% viene preso
      ADD R7, R7, #1      ; 50% non viene preso
CONT: ADD R8, R8, #1
      B FOR                ; preso sempre
```

La CMP costa 1τ, la BEQ FINE si suppone non viene presa quindi le due istruzioni successive non vengono eliminate, e quindi costa 1τ. LDRB e MOV costano entrambi 1τ. BL costa $3\tau + \tau$(CIFRA) cioè il costo del sottoprogramma CIFRA. La LDRB scrive R2 che viene letto dalla CMP seguente, questo provoca una dipendenza, e quindi la LDRB costerà 2τ e la CMP 1τ. La BEQ CONT è un if, si può ipotizzare che 50% viene preso e 50% non viene preso. Nel caso in cui il salto viene preso la BEQ costerà 3τ e poi si andrà ad eseguire CONT. Nel caso in cui non viene preso la BEQ CONT, STRB e ADD costano ognuno 1τ. L'if costa 3τ indipendentemente se la BEQ CONT viene presa o no. Infine la ADD costa 1τ e la B finale 3τ. In totale il loop impiega $17\tau + \tau$(CIFRA). Se questo loop fosse stato eseguito in un processore single cycle, quindi senza dipendenze: nel caso in cui BEQ CONT viene preso avrei $10\tau + \tau$(CIFRA), altrimenti $12\tau + \tau$(CIFRA).

6.6.3 Analisi delle prestazioni pipeline

Il processore pipeline dovrebbe avere un CPI pari a 1, perchè a ogni ciclo viene attivata una nuova istruzione. Tuttavia, gli stalli e gli svuotamenti sprecano cicli, quindi il CPI è un po' maggiore di 1 e dipende dal programma in esecuzione. Il pipeline è molto più veloce degli altri due con un tempo pari a $T_c = (\#stadi\text{-}1)\tau + \#istruzioni \cdot \tau$, ma in termini di requisiti hardware, il pipeline è simile al processore a ciclo singolo, ma richiede 8 registri, qualche multiplexer e un po' di logica di controllo per la gestione delle dipendenze.

6.7 Microarchitetture avanzate

I microprocessori ad alte prestazioni usano una miriade di tecniche per eseguire più velocemente i programmi. Il tempo richiesto per eseguire un programma è proporzionale al periodo del clock e al numero di istruzioni di cicli di clock: per migliorare le prestazioni serve quindi velocizzare il clock e/o diminuire il CPI.

6.7.1 Lunghezza pipeline

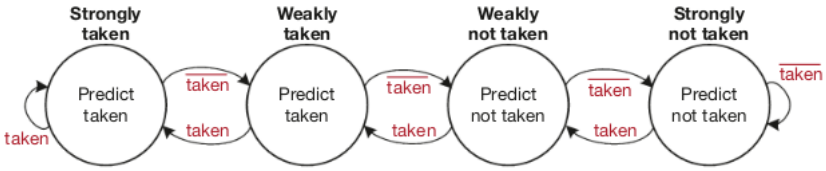
Il modo più semplice per velocizzare il clock è quello di dividere la pipeline in più stadi: ogni stadio contiene meno logica, quindi può essere più veloce ma pipeline più lunghe introducono più dipendenze e aggiungere stadi comporta costi maggiori per i registri e per le componenti hardware necessarie per gestire le dipendenze.

6.7.2 Micro operazioni

Si vuole rendere veloci i casi frequenti definendo un insieme di semplici micro operazioni che possono essere eseguite su percorso dati semplici. Le istruzioni vengono scomposte in una o più micro operazioni, per esempio, l'istruzione LDR R1, [R2], #4, può essere scomposto dal percorso dati in LDR R1, [R2] e ADD R2, R2, #4. Il programmatore avrebbe potuto scrivere direttamente le sequenze di istruzioni più semplici nel programma ma l'istruzione complessa occupa meno spazio in memoria.

6.7.3 Previsione dei salti

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** 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. 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. I salti in avanti sono più difficili da prevedere senza conoscere meglio la struttura del programma in esecuzione, 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 che contiene la destinazione di ciascun salto e la storia del salto, ovvero se sia stato eseguito oppure no l'ultima volta che è stato incontrato. Un predittore dinamico dei salti a *due bit* risolve parzialmente il problema usando un automa a 4 stadi denominati come salto "decisamente da fare", "forse da fare", "forse da non fare", "decisamente da non fare".



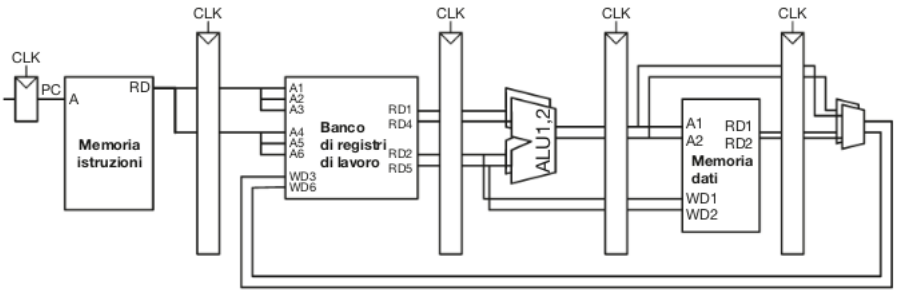
Al primo ciclo se il salto viene preso il predittore entra nello stadio *forse da fare*, se il salto viene preso nuovamente entra nello stadio *decisamente da fare*. Se al primo ciclo il salto non viene preso un volta, il predittore passa a *forse da non fare*, se non viene preso due volte allora *decisamente da non fare*. Il predittore opera nello stadio di Fetch della pipeline per decidere di quale istruzione fare il fetch nel prossimo ciclo. Il processore deve essere pronto ad annullare istruzioni nel caso in cui la previsione sia sbagliata e quindi salvare il risultato in un registro temporaneo prima di consolidare il valore nell'unità registri, questo perchè nello stadio di Writeback vi deve essere la conoscenza esatta se il salto è stato preso oppure no.

6.7.4 Processore out of order

Un processore out of order esamina un certo numero di prossime istruzioni per iniziare a eseguire istruzioni indipendenti il più rapidamente possibile: le istruzioni possono venire attivate in ordine diverso da quello scritto dal programmatore, a patto che le dipendenze vengano rispettate e che il programma produca i risultati previsti. I processori out of order usano una tabella **scoreboarding**, riguardanti le dipendenze, per tenere traccia delle istruzioni che attendono di essere attivate. Questa operazione è molto importante perché permette di trattare a runtime le dipendenze che altrimenti indurrebbero alla creazione di stalli nel pipeline, ma ha un costo elevato.

6.7.5 Processori superscalari

Nel processore pipeline viene utilizzato il parallelismo temporale, invece in un processore superscalare si utilizza il parallelismo spaziale, cioè contiene più copie hardware del percorso dati, per poter eseguire più istruzioni contemporaneamente. Il percorso dati esegue il fetch di due istruzioni alla volta dalla memoria istruzioni; ha un banco di registri a sei porte per consentire di leggere quattro operandi sorgente e scrivere due risultati in ogni ciclo; contiene inoltre due ALU e una memoria dati a due porte per eseguire contemporaneamente le due istruzioni.



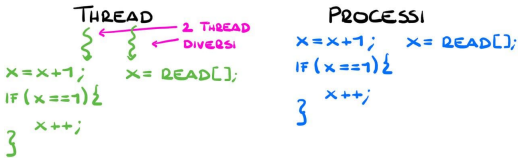
Questo processore è un superscalare a due vie (ci potrebbero essere a quattro vie, a sei vie, ecc...).
L'esecuzione simultanea di più istruzioni aggrava il trattamento delle dipendenze:

```
1 LDR R1, [R2, R3]
2 ADD R4, R5, R6
3 SUB R1, R1, #3
4 ADD R4, R4, #1
```

In processore pipeline non vi sarebbero problemi di dipendenze perchè le 3 condizioni di Berneistain sono soddisfatte, ma in un processore superscalare a due vie, dato che l'istruzione vengono eseguite a coppie, vi è la dipendenza tra LDR e la SUB. Il processore superscalare ha un CPI pari a 0.5 perchè esegue due istruzioni in ogni ciclo.

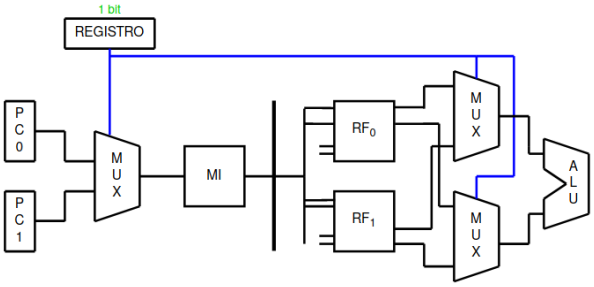
6.7.6 Multithreading e Multiprocessing

Un **thread** e un **processo** sono entrambi flussi di controllo indipendenti, cioè frammenti di programmi che possono eseguire con la loro logica interna. I thread condividono lo spazio di indirizzamento mentre i processi hanno spazi di indirizzamento diversi.



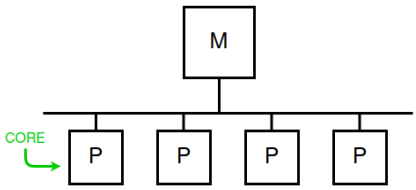
Nel caso dei thread, la x, in entrambi i codici, è la stessa variabile perchè lo spazio di indirizzamento è condiviso. Nel caso dei processi, la x non è la stessa e sono salvate in due memorie separate.

Thread
Il supporto multithread replica lo stato architetturale (PC e registri) per un certo numero di volte, per esempio 2 volte, per fare in modo che il processore esegua il programma dello stato architetturale 0 o il programma definito dallo stato architetturale 1. Vi sono 2 PC: il PC del primo thread e quello del secondo thread. All'ingresso del multiplexer vi è PC0 e PC1 a seconda di cosa ci sia scritto nel registro da 1 bit che indica in quale thread è in esecuzione; il risultato arriva alla memoria istruzioni. Vi è la linea con tutti i campi di istruzioni, e il valore che deve essere letto arriva sia all'unità registri 0 che all'unità registri 1. Alla ALU arriva il valore dei due registri utilizzando dei multiplexer comandati dal registro da 1 bit. Al costo di qualche multiplexer in più si può organizzare un processore che può eseguire un istruzione del thread 0 o del thread 1.



Si possono fare per ipotesi che per un giro si lavora con PC0 e nel giro successivo con PC1 e poi si ricomincia da capo, cioè si alterna prendendo un'istruzione da un thread e un'altra istruzione da un altro thread. In questo modo si possono eseguire 2 thread alla velocità di $\frac{1}{2}$ processore pipeline. Il fatto di lavorare in questo modo permette di allontanare le istruzioni che hanno dipendenze fra di loro. Questo procedimento si chiama **hyperthreading**. Se vi è una dipendenza RAW con una LDR seguita da una istruzione operativa causa una bolla nel pipeline, mentre se vi è un salto preso B, questo significa 2 bolle nel pipeline. Allora con l'hyperthreading significa che le bolle da un ciclo spariscono, mentre le bolle da due cicli vengono dimezzate. In alternativa all'hyperthreading, si può eseguire il thread 0 finchè non vi è una dipendenza, poi si passa al thread 1 e così via.

Processi
Un processo è circa uguale a un programma in esecuzione, che potrebbe voler avere una sua macchina processo e memoria dedicata. Dal punto di vista architetturale si chiama **multiprocessore**, vi è un certo numero di processori (normalmente a potenza di due), i quali hanno accesso a una unica memoria, e ognuno di questi processori (core) è in grado di fare il singolo tipo di processore trattato in questo capitolo. In particolare ognuno può eseguire un programma diverso accedendo a una parte di memoria che è disgiunta da quella che accedono gli altri programmi. Questo tipo di multiprocessore si chiama **memoria condivisa**.

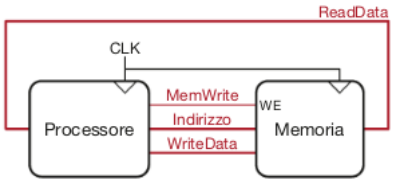


Chapter 7

Sistemi di memoria

7.1 Introduzione

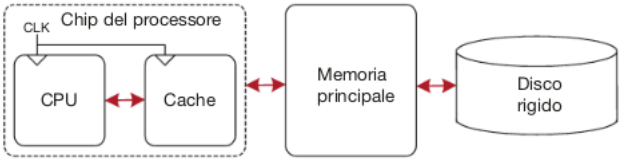
I processori di un tempo erano relativamente lenti, e la memoria era in grado di reggere il passo, ma la velocità dei processori è cresciuta a un ritmo maggiore di quella della memoria, e questo crescente divario di prestazioni fra processore e memoria richiede sistemi sempre più sofisticati per cercare di approssimare una memoria veloce quanto il processore. Il processore può eseguire istruzioni in maniera sempre più veloce, ma alla fine quello che impatta è il tempo della memoria.



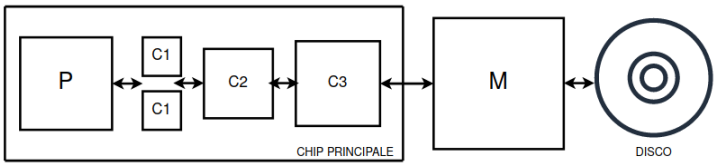
Come nel modello di Von Neumann: il processore invia un indirizzo al sistema di memoria, e può inviare e ricevere dei dati. Questa iterazione avviene durante il fetch di un istruzione o durante un load/store in memoria. Le memorie dei calcolatori sono costituite da:

1. Latch;
2. Flip flop;
3. Registri;
4. RAM dinamiche DRAM (array di transistor) e RAM statiche SRAM (array di registri);
5. Dischi: HDD e SSD.

I Latch, Flip flop e Registri sono componenti logici che possono contenere una quantità minima di informazioni.
I calcolatori usano i dischi per contenere i dati che non trovano spazio nella memoria principale. Andando dal primo verso l'ultimo in questa gerarchia, rallentano i tempi di accesso, aumentano le dimensioni e diminuiscono i costi. Si può approssimare la memoria ideale combinando una memoria veloce, piccola ed economica con una memoria lenta, grande ed economica. La memoria veloce, denominata **cache**, memorizza le istruzioni e i dati usati più di frequente. La memoria grande memorizza il resto delle istruzioni e dei dati. La combinazione delle due memorie economiche è molto meno costosa di un'unica memoria grande e veloce.



Il processore cerca il dato nella memoria cache: se il processore trova il dato, questo evento di "dato trovato" si chiama **hit**, altrimenti guarda nella memoria principale e questo evento di "dato non trovato" si chiama **miss** (o **fault**). Se non lo trova neanche lì, lo preleva dal disco rigido, capiente ma lento. Vi sono due principi: **località spaziale** cioè si intende il fatto che se si usa un dato particolare è molto probabile avere bisogno di altri dati *vicini* al primo dato, e **località temporale** cioè si intende il fatto che se si è usato recentemente un dato è molto probabile doverlo usare ancora a breve. La località spaziale e temporale definiscono un concetto di **working set**, cioè un insieme di istruzioni e dati che permettono di eseguire ad un certo istante un programma.



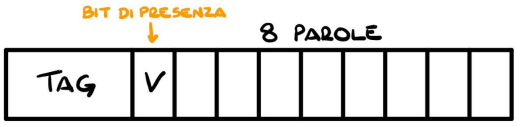
Con questi due principi si può organizzare sistemi di memoria con uno schema differente: il processore comunica con le due cache di primo livello (una per i dati e una per le istruzioni), in seguito c'è una cache più grande di secondo livello e infine una cache ancora più grande di terzo livello. Le cache di primo livello sono più veloci ma meno capienti rispetto alla cache di secondo livello, stessa cosa per la cache di secondo livello rispetto alla cache di terzo livello. Tutto questo fa parte del chip principale (CPU) ma sono componenti separati. La cache di terzo livello comunica con la memoria principale che a sua volta comunica con il disco.

7.2 Memoria 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. Per fare spazio al nuovo dato, la cache deve sostituire un dato vecchio e vi sono 2 problemi:

1. Come muovere i dati e istruzioni tra livelli?
2. Quale dato viene sostituito dal nuovo dato quando la cache è piena?

Il motivo trainante nel rispondere a queste domande è la località spaziale e temporale: le cache usano i due principi per prevedere quali dati saranno necessari nel prossimo futuro. Località temporale, il processore ha un'elevata probabilità di accedere nuovamente a un dato se lo ha utilizzato da poco, quindi, quando il processore legge o scrive un dato non presente in cache, tale dato viene copiato in cache in modo che successivi accessi allo stesso dato diano luogo a hit. Località spaziale, quando il processore accede a un certo dato, ha un'elevata probabilità di accedere ad altri dati in locazioni di memoria vicine al dato in questione, allora quando la cache preleva una parola da memoria principale, preleva anche alcune altre parole adiacenti: questo gruppo di parole è denominato **b blocco** o **linea di cache**. La linea di cache contiene alcuni bit che servono per memorizzare **p parole** nella memoria e possiede un numero arbitrario di parole, solitamente 2^k per un certo k. Il resto dei bit della linea di cache sono il **tag** e servono per memorizzare le informazioni di quale blocco di indirizzi corrispondono alle parole e infine vi è un **bit di presenza** (o validità) che se vale 1 significa che la linea è piena, mentre, se vale 0 significa che il contenuto non è significativo come per esempio all'accensione del calcolatore. Ogni cache è organizzata in **set**, ciascuno dei quali contiene uno o più linee di cache.



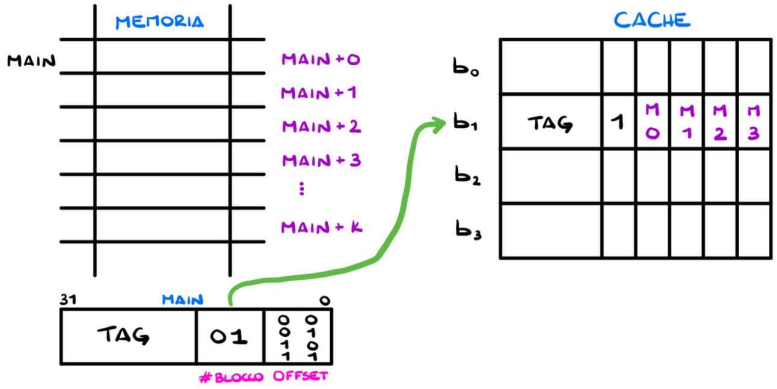
7.2.1 Come organizzare i dati nella memoria cache

Vi sono 3 metodi per accedere ai dati nella cache:

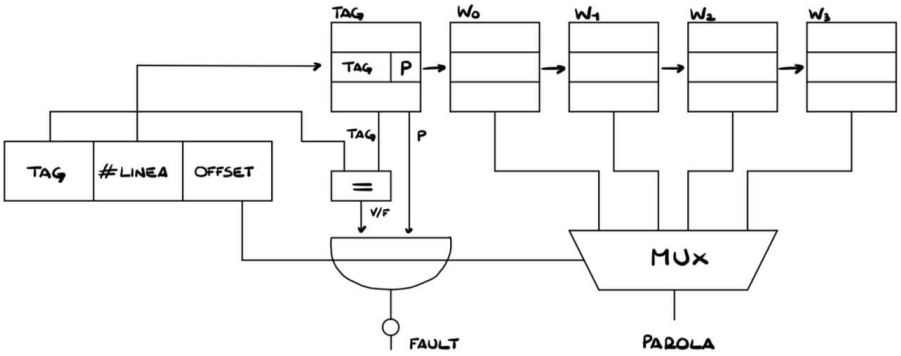
- **Indirizzamento diretto**: semplice ma presenta qualche problema;
- **Indirizzamento associativo**: più complicato da implementare ma non ha complicanze del metodo diretto;
- **Indirizzamento set-associativo**: ha i vantaggi del metodo diretto e del metodo associativo senza i problemi del diretto e senza i costi dell'associativo.

Indirizzamento diretto

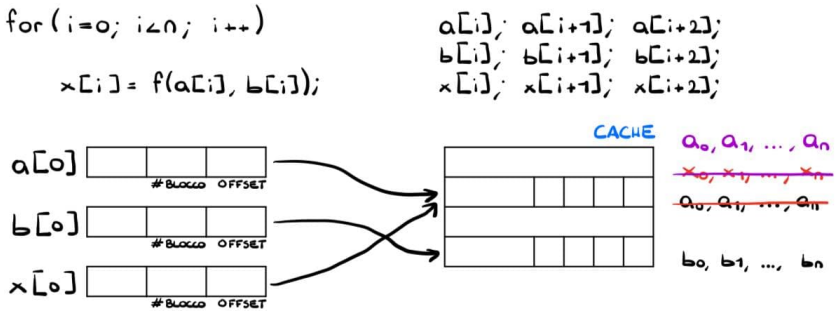
L'indirizzamento diretto ha una sola linea di cache per ogni set. Si suppone che un programma sia stato memorizzato in memoria a partire da un indirizzo main e si suppone di avere una cache con 4 blocchi, ognuno dei quali può contenere 4 parole. Durante il fetch dell'indirizzo main vengono considerate 4 parole: main+0, ..., main+3 perchè nella memoria cache si hanno 4 parole. L'indirizzo del main da 32 bit possiede $\log_2 p = 2$ bit che indicano l'**offset**, altri $\log_2 b = 2$ bit per il **numero di blocco**, e il resto dei bit più significativi sono il tag. Per esempio se i bit di blocco sono 01 allora le 4 parole con offset 00, 01, 10, 11 vengono salvate nella cache di blocco 1. Il tag viene copiato nella linea e il bit di presenza viene settato a 1 perchè la linea è piena. Ora si considera il caso di cercare una informazione nella cache: main+1. Il numero di blocco dell'indirizzo indica in quale linea cercare, si confronta il tag dell'indirizzo con il tag della linea e se sono uguali allora viene usato l'offset per prendere la parola.



A livello di componenti la cache contiene 5 moduli di memoria di cui il primo è il tag e gli altri W_0, ..., W_3 parole. Il numero di linea viene usato come indirizzo per le parole dei moduli di memoria. Il primo modulo, all'interno della parola ha un tag e un bit di presenza. La parte del tag del modulo viene confrontato con il tag dell'indirizzo e verifica se sono uguali. Il risultato del confrontatore viene messo in AND con il bit di presenza. Il valore dell'AND viene negato, diventando un segnale di **fault**, cioè se il fault è vero allora qualsiasi operazione fatta sulla linea di cache non è valida perché la linea non è piena o i tag non sono uguali. Se il fault è falso, allora le 4 parole della cache vengono inviate in un multiplexer comandato dall'offset dell'indirizzo che restituirà la parola.



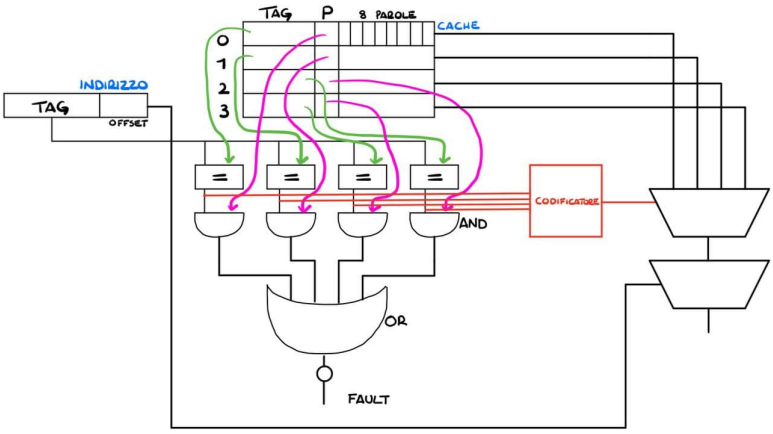
L'accesso a memoria costa un tempo di accesso per leggere il tag e le parole nei moduli, e $\max\{t_{CONF} + t_{AND}, t_{MUX}\}$. Nel caso di scrittura il procedimento è lo stesso con la differenza che l'offset dell'indirizzo comanda un demultiplexer e vi è un segnale di WE che viene usato per le 4 parole. L'offset sceglie dove mandare il WE, e quindi solo uno dei moduli viene scritto, se e solo se WE && !fault. La procedura per l'indirizzamento diretto è fare il fetch di un indirizzo, allora fare l'accesso alla memoria cache, se non vi è un segnale di fault allora la parola viene trovata, altrimenti in memoria vengono lette le parole che vanno sulla linea e vengono aggiornate nella memoria cache insieme al tag e il bit di presenza. Questo evento si chiama **fault di cache**. Una volta sostituita la linea, se si accede nuovamente alla cache non si avrà un segnale di fault. Il vantaggio del metodo diretto è che serve pochissimo hardware per implementarla ma il lato negativo è che se due indirizzi si mappano nello stesso set, si verifica un **conflitto**, e il dato cui si accede per ultimo espelle il precedente dal blocco. Le cache a mappatura diretta ha un solo blocco in ogni set, quindi due indirizzi che si mappano nel medesimo set causano sempre un conflitto.



Si suppone di avere un loop come in figura. Per località temporale si ha bisogno di a[i], e degli elementi vicini, stessa cosa per b[i] e x[i]. Si immagina che a[0] viene mappato in una posizione e quindi vengono trasferiti tutte le parole da a[0] fino a a[n]. In seguito si deve andare a caricare b[0] che si immagina viene mappato in un'altra posizione. Infine si suppone che x[0] viene mappato nella stessa posizione di a[0], in questo caso vi è un conflitto, si deve scaricare gli a[0],...,a[n] e inserire x[0],...,x[n]. Il ciclo ricomincia, quindi si deve inserire a[1], scaricare x[0],...,x[n] e caricare nuovamente gli a[0],...,a[n] e così via. Questo evento si chiama **trashing** ed è una situazione molto sfavorevole perché per fare un singolo accesso alla memoria, si deve scaricare una linea di cache, caricare una nuova linea, utilizzare una singola parola e scaricare nuovamente la linea.

Indirizzamento associativo

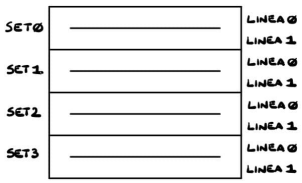
Una cache associativa è costituita da un unico set composto da b blocchi. Si suppone di avere una memoria cache con 4 linee ognuno con un tag, un bit di presenza e 8 parole. In questo caso l'indirizzo è formato solo da due sezioni: il tag e l'offset. I diversi tag della memoria cache vengono mandati in dei confrontatori insieme al tag dell'indirizzo. Se viene cercato un indirizzo, il confronto dei tag viene fatto istantaneamente su tutti i blocchi della cache. Il risultato di ogni confrontatore sarà 0 nel caso in cui il confronto sia falso, o 1 nel caso in cui sia vero. In seguito il risultato viene messo in AND con il bit di presenza di ogni blocco e il valore risultante viene messo in una porta OR negata per ottenere il segnale di fault. Se il segnale di fault è falso, significa che almeno una porta AND era vera e quindi bisogna capire quale tag coincideva con il tag dell'indirizzo e con l'offset prendere la parola cercata. Le uscite dei confrontatori vengono mandati in un codificatore il cui risultato è il segnale di controllo per un multiplexer che prende le linee della memoria cache. Il risultato del multiplexer viene mandato in un altro multiplexer comandato dall'offset dell'indirizzo per ottenere la parola cercata.

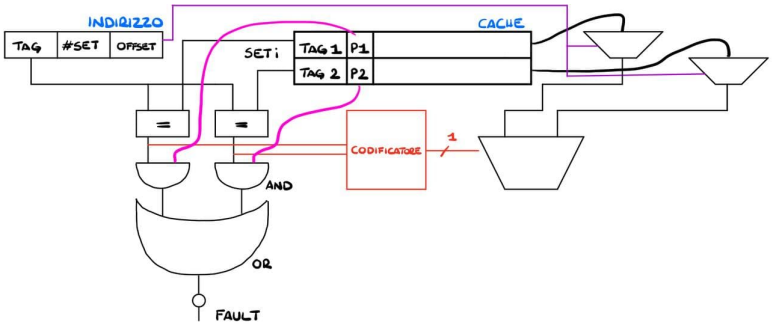


Un'altra versione è avere 4 multiplexer nelle linee di cache, tutti comandati dallo stesso offset, e dal risultato di questo multiplexer escono 4 parole che vanno in un altro multiplexer comandato dal codificatore. Le due soluzioni sono equivalenti dal punto di vista funzionale ma non dal punto di vista del tempo. Le cache a indirizzamento associativo sebbene siano molto efficienti sono molto costosi dal punto di vista hardware.

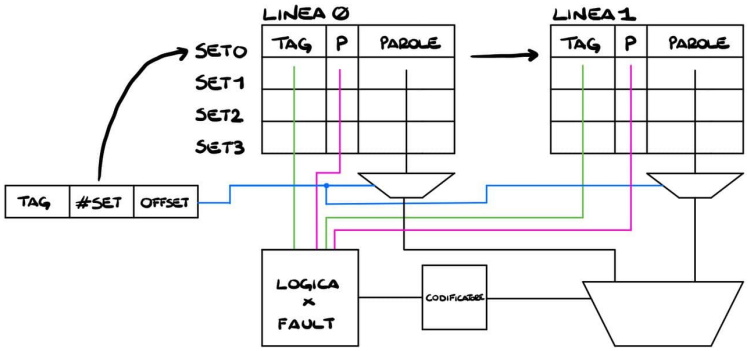
Indirizzamento set-associativo

La cache a indirizzamento set-associativo è il giusto compromesso: la cache viene divisa in un certo numero di set, per esempio 4, dove all'interno vi sono un certo numero di linee, per esempio 2.Si sceglie l'indirizzamento diretto per decidere quale set e il metodo a indirizzamento associativo per decidere quale linea dell'insieme. Si studia un singolo set composto da 2 linee e 8 parole per ciascuna linea. Occorrono 2 confrontatori per i tag, i cui risultati vengono messi in due AND con il bit di presenza. Il risultato delle due porte logiche viene messo in un OR negato per ricavare il segnale di fault. Le parole vengono messi in due multiplexer comandati dall'offset, il risultato del multiplexer viene mandato a un secondo multiplexer. I risultati dei confrontatori vengono messi in un codificatore per ottenere il segnale di controllo per il secondo multiplexer che sceglierà tra la prima o la seconda linea.



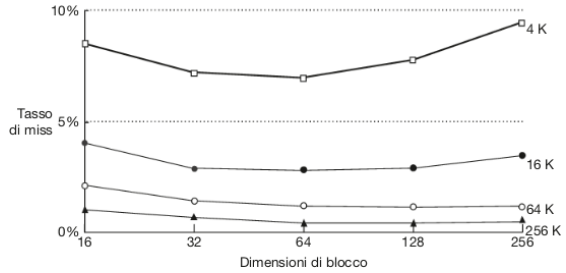


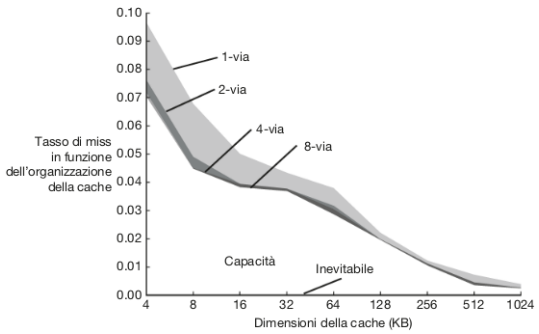
Invece di fare una memoria apposita solamente per i tag, si può immaginare di avere 2 memorie per ogni linea. Il set dell'indirizzo indica il set e i due tag insieme ai bit di presenza vengono mandati alla logica del fault. Le parole vanno in dei multiplexer comandati dall'offset e il risultato va in un secondo multiplexer con segnale di controllo il codificatore per ottenere la parola.



7.2.2 Analisi delle prestazioni del sistema memoria

Viene considerata **hit rate** la percentuale di tentativi riusciti per trovare un dato nella cache, e **miss rate** è la percentuale di miss. Le situazioni di miss possono essere classificate come **miss inevitabile** perché il primo blocco deve essere per forza letto dalla memoria principale indipendentemente dall'organizzazione della cache; **miss di capacità** si verificano quando la cache è troppo piccola per contenere tutti i dati e **miss di conflitto** quando più indirizzi vengono mappati nello stesso set ed espellono blocchi ancora necessari.





La modifica dei parametri della cache può influenzare uno o più tipi di miss:

- incrementare la capacità della cache può ridurre i miss di conflitto e di capacità, ma non ha effetto sui miss inevitabili. Anche aumentare i set nelle cache associative fa calare il miss rate;
- incrementare la dimensione del blocco può ridurre i miss inevitabili (per località spaziale) ma aumentare i miss di conflitto perchè più indirizzi mappano sullo stesso set creando conflitti. Nella zona iniziale del primo grafico ci sono poche parole per linea e per il principio di località spaziale aumenta il tasso di miss. Avere tante parole per linea però aumenta il tempo necessario a prelevare dalla memoria principale il blocco mancante.

7.2.3 Quale dato viene sostituito?

Il principio di località temporale suggerisce che la scelta migliore sia quella di espellere il blocco più "vecchio", perché per probabilità è quella che per più tempo non viene utilizzato. La politica di sostituzione dei blocchi adottata dalla maggior parte delle cache è dunque la **politica LRU** (Least Recently Used). In ogni linea di cache insieme al tag, al bit di presenza e alle parole viene aggiunto un **bit di utilizzo U** che viene messo uguale ad 1 a ogni accesso ad una parola della linea. Periodicamente U viene rimesso a 0. In questo caso vi è l'implementazione di una politica **pseudo-LRU** e si sceglie di rimpiazzare la linea con U uguale a 0.

7.2.4 Politiche di scrittura

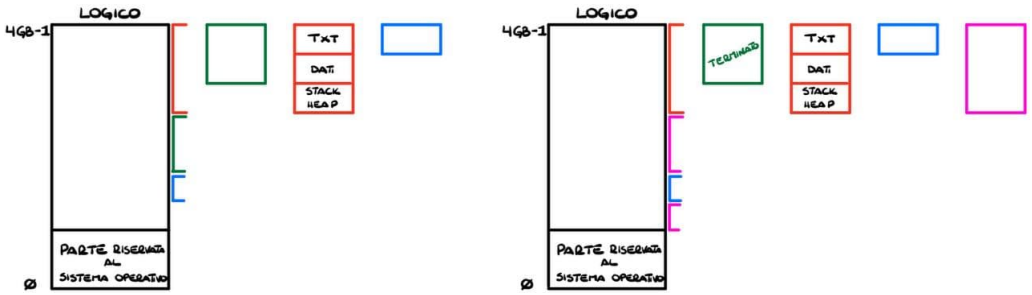
Le cache sono classificate in due categorie:

- **Write-through**: il dato viene scritto nella memoria cache e in maniera asincrona riporta l'aggiornamento ai livelli superiori fino alla memoria principale;
- **Write-back**: vi è un **bit di modifica M** 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.

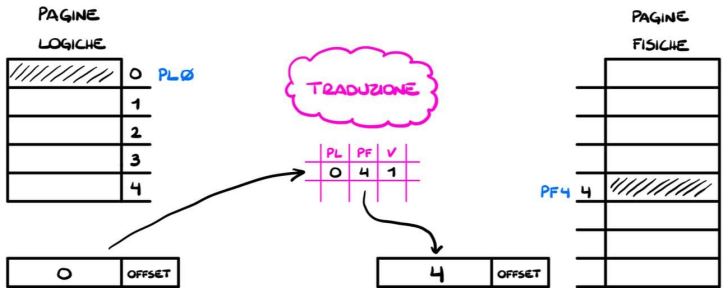
Una cache write-through non ha bisogno del bit di modifica ma richiede più accessi alla memoria principale di una cache write-back. Le moderne cache sono quasi sempre write-back per l'eccessiva lentezza della memoria principale.

7.3 Memoria

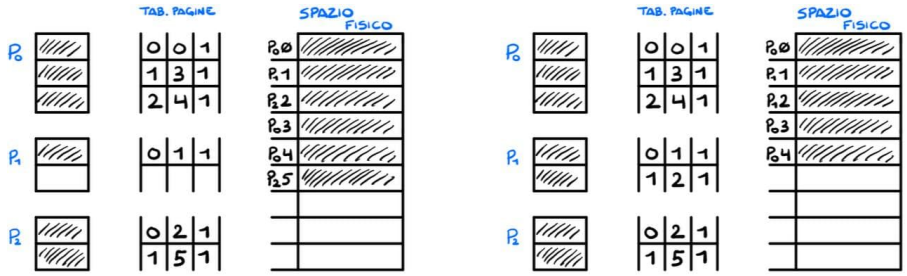
Dal punto di vista logico la memoria è un vettore, compreso tra 0 fino a 4GB-1 (dal punto di vista fisico qualunque calcolatore non ha tutta la memoria disponibile nel suo spazio di indirizzamento). Il problema è come utilizzare la memoria in maniera efficiente per far si che si possa eseguire una serie di programmi che hanno ampiezze diverse all'interno della memoria, ma tenendo conto che una parte della memoria è occupata dal sistema operativo. Ogni programma ha una sezione testo, dati, stack, heap,...



Se il primo programma termina, lascia un buco in mezzo alla memoria e se ora ci fosse un altro programma, ci sarebbe lo spazio per inserirlo ma si dovrebbe dividere il programma. Questo problema non è facile da gestire e si chiama **frammentazione della memoria** cioè a forza di far girare programmi e fare tanti piccoli buchi, di avere un sacco di memoria libera ma non riuscire a sfruttarla. Un altro problema è che in un programma, per esempio assembler, ci sono delle istruzioni che fanno riferimento alla memoria, quindi se viene caricato il programma in una posizione A o caricato in una posizione B, gli indirizzi di memoria cambiano. Per risolvere il problema viene separata il concetto di visione logica e visione fisica della memoria, cioè si ha una **memoria fisica** e una **memoria logica** dove rispettivamente **indirizzi logici** e **indirizzi fisici**. Si cerca di fare in modo ci sia un meccanismo di **traduzione** fra gli indirizzi della memoria logica e fisica: questa operazione si chiama **paginazione**, cioè la memoria logica e fisica vengono divisi in porzioni. Ogni porzione della memoria logica si chiama **pagina logica**, mentre ogni porzione della memoria fisica si chiama **pagina fisica**. Pagine logiche e fisiche hanno la stessa dimensione, tipicamente potenze di 2 (di solito 4K). L'indirizzo logico è composto dal numero di pagina logica e un offset, quando si effettua la traduzione il sistema operativo decide dove allocare la pagina logica in una pagina fisica. Per esempio decide di allocare la pagina logica 0 nella pagina fisica 4. Il sistema operativo si mantiene una **tabella delle pagine** che alla posizione 0 corrisponde a 4, e un bit di presenza che indica se quella pagina logica è mappata nella pagina fisica, altrimenti la pagina logica va caricata dal disco. L'indirizzo fisico è composto dalla posizione della pagina fisica e dallo stesso offset dell'indirizzo logico. La tabella delle pagine risiede nella memoria principale.



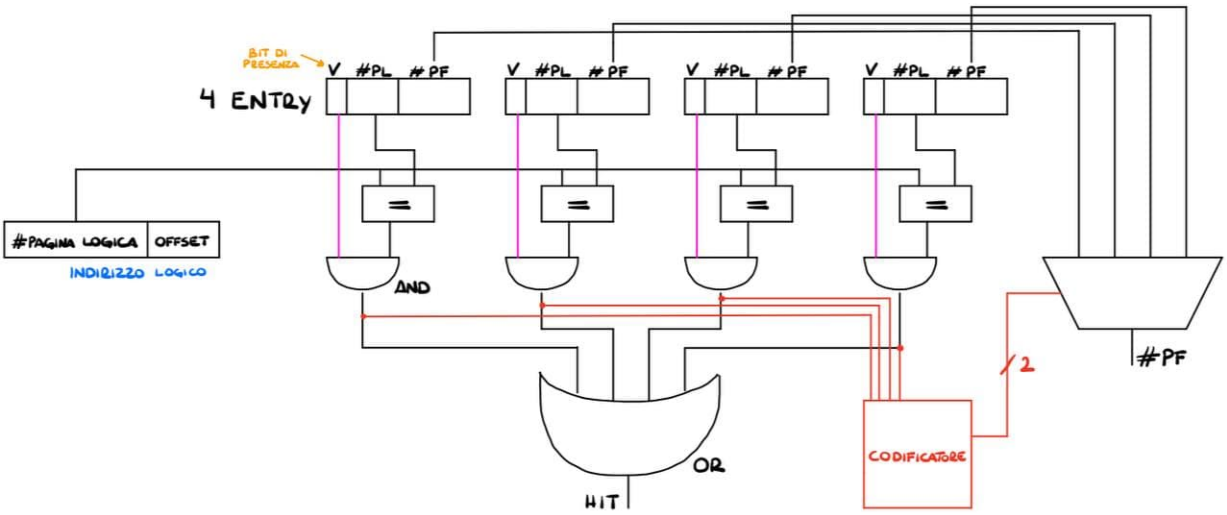
Con questa semplice operazione, si può tradurre un indirizzo valido nello spazio di indirizzamento logico in uno spazio di indirizzamento fisico. Si studia un altro esempio con 3 programmi dove il primo richiede 3 pagine e gli altri due richiedono 2 pagine. Ogni programma ha una tabella delle pagine. Si immagina che il primo programma va in esecuzione e quindi viene allocata la prima pagina fisica, stesso procedimento per gli altri due programmi. Il primo programma continua mandando in esecuzione le altre due pagine, e anche il terzo programma manda in esecuzione la seconda pagina e termina. Le pagine del terzo programma possono essere liberate, quindi se ora il secondo programma prosegue può occupare le pagine che erano del terzo programma.



Se nello spazio fisico si arriva a un punto in cui tutte le pagine sono piene e si deve allocare una ulteriore pagina allora si applica la politica LRU, cioè si sceglie la pagina più vecchia, la si inserisce sul disco e al suo posto si aggiunge la pagina nuova. Con il procedimento di traduzione si è risolto il problema del miglior utilizzo della memoria, ma sono peggiorate le prestazioni perchè ad ogni fecth istruzione occorre fare 2 accessi alla memoria.

7.3.1 Translation lookaside buffer (TLB)

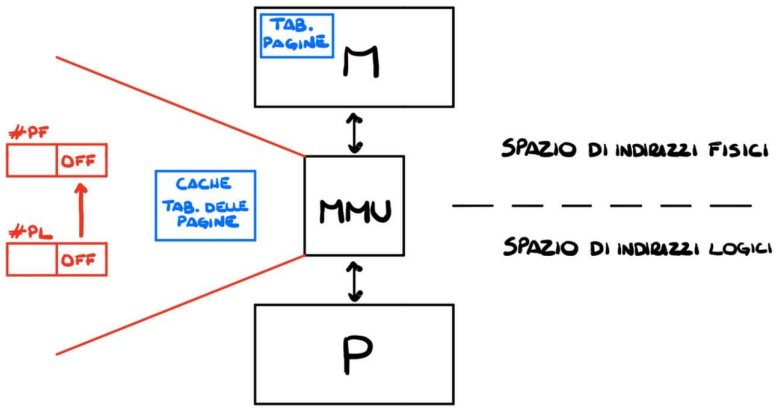
I due principi di località spaziale e temporale si possono trasferire anche a livello di traduzione di indirizzi, cioè invece di accedere alla intera tabella delle pagine, si vorrà accedere a solo una porzione. Si immagina di avere una cache di primo livello che contiene le associazioni #PL/#PF che servono in quel momento. Le cache che compiono queste operazioni si chiamano **TLB** e sono cache a indirizzamento associativo che contengono **entry**, cioè due elementi #PL, #PF e il bit di presenza. Le entry sono in piccolo numero (da 16 a 512). La TLB permette di effettuare la ricerca nella tabella delle pagine in modo molto efficiente.



Si suppone ci siano 4 entry. Il numero di pagina logica lo si confronta con il numero di pagina logica degli entry. L'uscita del confrontatore lo si mette in AND con il bit di presenza, e le uscite arrivano alla porta OR e al codificatore. Se uno degli AND è vero, allora si ha un hit, altrimenti abbiamo un miss. Le pagine fisiche delle entry vengono mandati in un multiplexer comandato dal codificatore. Il codificatore restituisce i 2 bit che indicano qual è il numero di pagina fisica giusto per tradurre quel numero di pagina logica. Nel caso in cui il risultato dell'OR è 0 ci sono due motivi: la pagina logica non è presente nelle 4 entry (lo si può assumere come se fosse un fault di cache, cioè si sostituisce dalla memoria principale con politica LRU e in questo caso viene gestito dalla TBL) oppure il bit di presenza è 0, in questo caso quello che avviene è un **fault di pagina** e questo richiede l'intervento del sistema operativo. Il sistema operativo carica la pagina mancante in memoria principale.

7.3.2 Memory management unit (MMU)

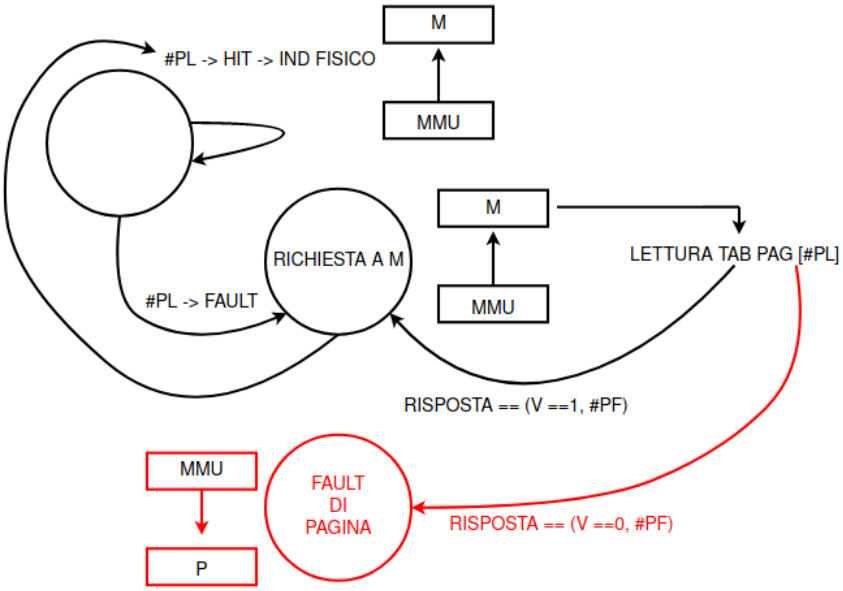
La MMU dialoga tra il processore e la memoria: da processore fino a MMU riguarda spazi di indirizzi logici, da MMU fino alla memoria riguarda spazi di indirizzi fisici.



All'interno di MMU vi è la traduzione degli indirizzi, e serve per risolvere tutta una serie di problemi di gestione della memoria come il problema della frammentazione, la maggiore disponibilità di memoria, e il problema della

protezione. La MMU contiene la TLB e funziona come un automa, cioè risiede sempre in uno stato in cui riceve una richiesta di traduzione dell'indirizzo da parte del processore:

- Se il #PL richiesto determina un hit nella cache interna allora si genera un indirizzo di pagina fisica, e si torna nello stesso stato per considerare la prossima traduzione degli indirizzi. La MMU manda l'indirizzo fisico al sottosistema di memoria. Questo è il caso più comune, e quindi deve essere trattato velocemente;
- Se il #PL richiesto determina un fault nella cache interna, allora occorre andare in un altro stato per fare richiesta alla memoria della entry della tabella delle pagine, relativa alla pagina logica che ha provocato il fault. La MMU manda la richiesta di lettura della tabella delle pagine con indirizzo della pagina logica alla memoria. La memoria risponde, e se la risposta è per cui il bit di presenza è uguale a 1, allora la risposta è come se fosse un hit del punto precedente. Se la risposta è per cui il bit di presenza è uguale a 0, allora si entra in un altro stato in cui vi è un fault di pagina che significa che la MMU interagisce con il processore perché ha cercato di tradurre un indirizzo che non è stata ancora caricata nella memoria fisica e quindi richiede l'intervento del sistema operativo.



Riassumendo

Si studia un ultimo esempio in cui vengono raccolti, tutti i concetti visti finora: si considera un programma che prende due aree di memoria v e w, e svolge un for(i=0; i<n; i++) x[i] = y[i]; Si cerca di immaginare dalla versione assembler la problematica di traduzione degli indirizzi logici a fisici.

```
        .data
v:      .word 1, 2, 3, 4, 5, 6, 7, 8
w:      .word 8, 7, 6, 5, 4, 3, 2, 1

        .text
        .global main

main:   mov r0, #8                  @primo parametro numero degli elementi nel vettore
        ldr r1, =v                  @secondo parametro indirizzo del vettore
        ldr r2, =w                  @terzo parametro indirizzo del secondo vettore
        push {lr}                   @salva indirizzo di ritorno del main
        bl copia                    @chiama la funzione
        pop {pc}                    @return

copia:  push {r4}
        mov r3, #0                  @azzera ri
loop:   ldr r4, [r1, r3, lsl #2]    @carica elemento
        str r4, [r2, r3, lsl #2]    @e copialo
        add r3, r3, #1              @i++
        cmp r3, r0                  @vedi se sei arrivato in fondo
        bne loop                    @se non sei in fondo ... cicla
fine:   mov r0, #1                  @altrimenti restituisci il risultato
        pop {r4}
        mov pc, lr
```

Come indirizzi lascia:

```
Disassembly of section .text:

00000000 <main>:
   0: e3a00008    mov   r0, #8
   4: e59f1034    ldr   r1, [pc, #52]  ; 40 <fine+0xc>
   8: e59f2034    ldr   r2, [pc, #52]  ; 44 <fine+0x10>
   c: e52de004    push  {lr}     ; (str lr, [sp, #-4]!)
  10: eb000000    bl 18 <copia>
  14: e49df004    pop   {pc}     ; (ldr pc, [sp], #4)

00000018 <copia>:
  18: e52d4004    push  {r4}     ; (str r4, [sp, #-4]!)
  1c: e3a03000    mov   r3, #0

00000020 <loop>:
  20: e7914103    ldr   r4, [r1, r3, lsl #2]
  24: e7824103    str   r4, [r2, r3, lsl #2]
  28: e2833001    add   r3, r3, #1
  2c: e1530000    cmp   r3, r0
  30: 1affffa     bne   20 <loop>

00000034 <fine>:
  34: e3a00001    mov   r0, #1
  38: e49d4004    pop   {r4}     ; (ldr r4, [sp], #4)
  3c: e1a0f00e    mov   pc, lr
  40: 00000000    .word 0x00000000
  44: 00000020    .word 0x00000020
```

Gli indirizzi sono in esadecimale: il primo indirizzo logico è la MOV del main che è andata all'indirizzo 0 e occupa la parola e3a00008. La LDR è all'indirizzo 4 perchè l'indirizzamento è al byte. Siccome viene eseguito =v, è stata tradotta come un indirizzamento relativo al PC che somma ai 4 dell'indirizzo i #52 e va all'indirizzo 40. STesso procedimento per la LDR successiva. All'indirizzo c la PUSH viene tradotta come una STR usando link register e stack point in pre-incremento. Lo SP viene decrementato di 4 perchè lo stack cresce verso il basso nell'ARM, e così via per le altre istruzioni.

In decimale, gli indirizzi generati sono:

- **0** MOV;
- **4** fetch LDR;
- **64** accede alla memoria per caricare l'indirizzo r1;
- **8** fetch seconda LDR;
- **68** accede alla memoria per caricare l'indirizzo r2;
- **12** PUSH;
- **16380** il valore dello SP;
- **16** BL;
- **24** la BL salta all'indirizzo 24;
- **16376** il valore dello SP;
- **28** MOV;
- **32** LDR,
- **1024** la precedente LDR prende l'indirizzo base di v;
- **36** STR;
- **2048** la STR prende l'indirizzo di base w;
- **40** ADD;
- **44** CMP;
- **48** BNE e da qui ricomincia il loop e quindi torna all'indirizzo **24**.

Chapter 8

Ingresso uscita I/O

8.1 Memory mapped I/O

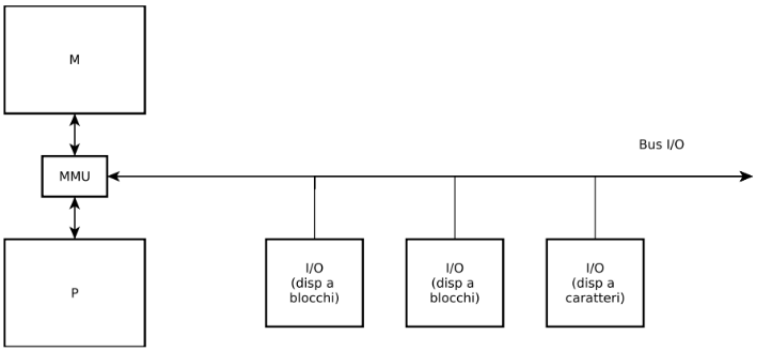
Un dispositivo è un oggetto hardware che ha una sua parte controllo e una sua parte operativa, e come tutti gli oggetti hardware ha un loop:

```
dispositivoI/O:
  while(true) {
    read(comando, parametri);
    results = exec(comando, parametri);
    writeback(results);
  }
```

Per esempio la tastiera restituisce il codice dei tasti premuti. Questi dispositivi hanno una piccola memoria che permette di memorizzare comandi e risultati delle operazioni che sono in grado di eseguire. Il ciclo di funzionamento della tastiera potrebbe essere:

```
while(true) {
  if(ordinedilettura == 1) {
     ... attendi pressione tasto ...;
     codice = codice(tastoPremuto);
     0 -> ordinedilettura;
  }
}
```

La tecnica del memory mapped I/O permette di utilizzare le normali istruzioni di load/store per andare a scrivere e leggere la memoria del dispositivo. Il processore riconosce alcuni indirizzi come appartenenti allo spazio di I/O (è questo il vero e proprio concetto di memory mapped I/O) e, tramite la MMU, redirige le richieste relative a questi indirizzi sul *bus I/O* che collega processore e dispositivi di I/O. Tutti i dispositivi di I/O vedono passare le operazioni, ma ognuno intercetta solo quelle che fanno riferimento agli indirizzi che gli sono stati assegnati.



8.2 Interruzioni

Un'interruzione è un evento asincrono rispetto all'esecuzione del programma sul processore. Quando il processore fa il suo solito ciclo, prima di ricominciare, testa le interruzioni.

```
while(true){
   fetch
   decode
   execute + update(PC)
   writeback
   if(interrupt){
      interrupt_management
   }
}
```

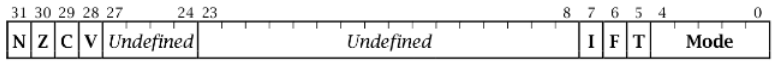
Le interruzioni sono generate dai dispositivi esterni al processore come i dispositivi I/O o dal sottosistema di memoria. Nel caso generale, le interruzioni vengono utilizzate per:

- gestire le operazioni di ingresso-uscita, come segnalare il completamento di una operazione di I/O;
- per gestire situazioni di errore, come un fault di pagina o una divisione per 0;
- per eseguire chiamate di sistema, ovvero per chiamare parti di codice con diritti diversi da quelli normalmente assegnati agli utenti.

Quando si verifica un'interruzione, il suo trattamento prevede una sequenza di passi standard:

1. l'istruzione in esecuzione viene completata, lo stato del processore viene aggiornato di conseguenza (per esempio viene calcolato il PC corrispondente alla prossima istruzione);
2. si salva parte dello stato del processore, tipicamente PC, LR e SP, e viene eseguito un **handler** cioè codice assembler dell'interruzione che avviene in uno stato particolare, a seconda del tipo di interruzione;
3. al termine, si ripristina lo stato del processore, e al prossimo ciclo while(true) si procede ad eseguire la prossima istruzione del programma quella corrispondente al PC calcolato all'inizio.

ARM riconosce un certo numero di stati diversi: **User** che è lo stadio normale, tutti i programmi quando vengono eseguiti sono in user mode; **System** che è lo stato del sistema operativo. **Fast interrupt** e **Interrupt** che vengono utilizzati per il trattamento delle interruzioni generate dai dispositivi di ingresso-uscita; **Abort** e **Undefined** che sono utilizzati per trattare le interruzioni generati da accessi illegali in memoria, e lo stato **Supervisor** che viene utilizzato per l'esecuzione delle chiamate di sistema (istruzioni SVC).



Questi stati del processore sono rappresentati nella parola di stato (CPSR): all'inizio ci sono gli stati dei flag, poi ci sono dei bit indefiniti, in seguito ci sono 3 bit che sono **I** (bit IRQ) e **F** (bit FIQ) che sono dedicate alle interruzioni. Se entrambi vengono messi a 1, fanno in modo che non vengono trattate le interruzioni normali e quelle "fast": le interruzioni di tipo fast vengono utilizzati per segnalare eventi che possono essere trattati molto rapidamente, quelle di tipo normale richiedono tempi di trattamento più lunghi. **T** è il Thumb mode, cioè se è 0 esegue istruzioni a 32 bit, se è 1 esegue istruzioni a 16 bit. Alla fine ci sono 4 bit che rappresentano gli stati, ciascuno di questi stati dispone di alcuni registri duplicati che vengono messi a disposizione per il trattamento delle interruzioni.

User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	**R8_fiq**	R8	R8	R8	R8
R9	R9	**R9_fiq**	R9	R9	R9	R9
R10	R10	**R10_fiq**	R10	R10	R10	R10
R11	R11	**R11_fiq**	R11	R11	R11	R11
R12	R12	**R12_fiq**	R12	R12	R12	R12
R13 (SP)	R13 (SP)	**R13_fiq**	**R13_irq**	**R13_svc**	**R13_abt**	**R13_und**
R14 (LR)	R14 (LR)	**R14_fiq**	**R14_irq**	**R14_svc**	**R14_abt**	**R14_und**
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
Program Status Registers						
CPSR	CPSR	CPSR **SPSR_fiq**	CPSR **SPSR_irq**	CPSR **SPSR_svc**	CPSR **SPSR_abt**	CPSR **SPSR_und**

Negli stati User e System ci sono gli stessi classici registri. Nel modo Fast Interrupt, i registri da R8 fino a R14 sono una copia. Questo significa che non occorre salvare quando si passa ad uno di questi stati operativi. Questi registri vengono ripristinati quando si ritorna dal trattamento dell'interruzione che provoca il cambiamento di stato. Tutti gli stati hanno a disposizione un registro SPSR che mantiene lo stato del CPSR al momento del passaggio di stato in modo che tale registro possa essere ripristinato al rientro dal trattamento delle interruzioni.

Quando si verifica un'interruzione il processore, una volta completata l'esecuzione dell'istruzione corrente ed aggiornato lo stato del processore esegue una sequenza di passi:

1. salva il PC corrente nella copia del registro LR;
2. salva la CPSR nella SPSR;
3. setta lo stato del processore nella CPSR (Fast Interrupt, Interrupt, ...);
4. si può disabilitare ulteriori istruzioni, andando a scrivere i bit nella CPSR (I e F);
5. salta ad eseguire del codice che si trova all'indirizzo 0x0000000 + tipo dell'interruzione. (vedi tabella seguente).

Al termine del trattamento dell'interruzione:

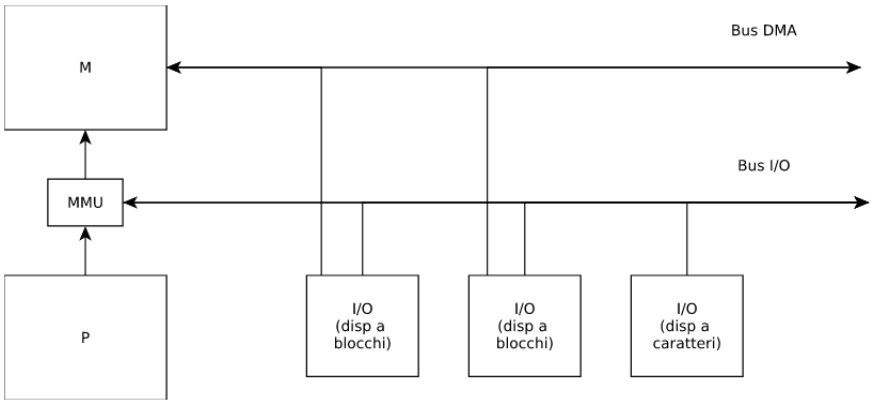
1. sposta il contenuto del SPSR in CPSR;
2. si azzerano eventuali 1 nei bit I o F;
3. sposta il contenuto del LR in PC.

0	reset
4	undef instruction
8	sw interrupt (SVC)
12	abort (fecth da indirizzo illegale)
16	abort (ldr/str indirizzo illegale)
20	reserved
24	IRQ
28	FIQ

8.3 Direct memory access (DMA)

DMA consente ad un dispositivo di ingresso-uscita di avere un accesso controllato in memoria centrale. Il dispositivo è collegato alla memoria centrale mediante un bus DMA condiviso fra tutte le periferiche in grado di lavorare in DMA. La memoria diventa un dispositivo che deve soddisfare le richieste di lettura e scrittura che arrivano da più unità (processore, periferiche, bus DMA) e deve pertanto essere dotato di un'unità controllo più sofisticata. Un'operazione di lettura dal disco su un dispositivo che supporta sia memory mapped I/O che DMA avviene tramite i seguenti passi:

- il sistema operativo, invia alla periferica utilizzando il memory mapped I/O l'ordine di lettura che contiene: la specifica che si tratta di un operazione di lettura, l'indirizzo del blocco interessato, l'indirizzo in memoria del buffer da utilizzare per i dati letti da disco;
- il dispositivo effettua la lettura in un proprio buffer interno alla memoria tramite il bus DMA;
- infine il dispositivo acquisisce il controllo del bus DMA e avvia il trasferimento del blocco letto nella posizione di memoria dato dal sistema operativo.



I dispositivi che normalmente sono dotati di DMA sono quelli cosiddetti "a blocchi" come i dispositivi di memorizzazione di massa(dischi a stato solido e non). Quelli che non sono dotati di DMA sono i cosiddetti "a caratteri" come tastiere, mouse, ecc...