

Università degli Studi di Catania

FACOLTÀ DI MATEMATICA E INFORMATICA

Corso di Laurea in Informatica

RIASSUNTI DI:

Architettura degli elaboratori e Laboratorio

Candidato:

Salvo Polizzi Salamone

Matricola 1000030092

Relatore:

Prof. Emiliano Alessio Tramontana

Indice

1 INTRODUZIONE AL CALCOLATORE	9
1.1 Famiglie di calcolatori	9
1.2 Componenti funzionali	10
1.2.1 La memoria	11
1.2.2 Il processore	12
1.3 Concetti operativi di base	13
1.3.1 Connessione fra processore e memoria	14
1.3.2 Passi dell'istruzione	14
1.3.3 Interruzioni	15
1.4 Prestazioni di un processore	15
2 ISTRUZIONI MACCHINA	17
2.1 Introduzione	17
2.2 Memoria del calcolatore	17
2.2.1 Indirizzamento	18
2.2.2 Bus di indirizzi	18
2.2.3 Indirizzamento a byte	19
2.3 Operazioni di memoria	19
2.4 Istruzioni macchina	19
2.4.1 Notazione RTN	20
2.4.2 Notazione simbolica	20
2.5 Architetture RISC e CISC	21
2.6 Istruzioni RISC	21
2.6.1 Formato di istruzioni	21
2.6.2 Operazione di addizione	22
2.6.3 Esecuzione delle istruzioni	23
2.6.4 Esecuzione di n somme e salto	23
2.7 Modi di indirizzamento	26
2.7.1 Programma per la somma di una lista di n numeri	27
2.7.2 Esempio di indirizzamento indiretto	28
2.7.3 Modo di indirizzamento con indice e spiazzamento	28
2.7.4 Modo con base e indice	30
2.8 Linguaggio assemblativo	30
2.8.1 Direttive di assemblatore	30

2.8.2	Assemblaggio ed esecuzione	31
2.9	Gestione pila	31
2.9.1	Operazione push e pop	32
2.10	Gestione di sottoprogrammi	33
2.10.1	Metodo di collegamento	33
2.10.2	Annidamento	33
2.10.3	Passaggio di parametri	34
2.10.4	Area di attivazione	38
2.11	Ulteriori istruzioni macchina	40
2.11.1	Scorrimento	41
2.11.2	Rotazione	42
2.11.3	Moltiplicazione e Divisione	43
2.12	Valori immediati a 32 bit	44
2.13	Insieme di istruzioni CISC	45
2.13.1	Istruzioni di autoincremento e autodecremento	46
2.13.2	Indirizzamento con modo relativo a PC	47
2.13.3	Bit di esito o condizione	47
2.14	Stili RISC e CISC	47
2.15	Codifica di istruzioni	48
3	OPERAZIONI DI INGRESSO/USCITA	51
3.1	Accesso a dispositivi di I/O	51
3.1.1	Interfaccia dei dispositivi	52
3.1.2	I/O controllati da programma	52
3.1.3	Lettura dati dalla tastiera	53
3.1.4	Scrittura caratteri sul video	53
3.1.5	Programma di lettura e scrittura di una linea di caratteri	54
3.1.6	Esempio in stile CISC	55
3.2	Interruzioni	56
3.2.1	Esempio di utilizzo delle interruzioni	57
3.2.2	Routine di servizio dell'interruzione	57
3.2.3	Servizio delle interruzioni	57
3.2.4	Controllo delle interruzioni	58
3.2.5	Dispositivi multipli	59
3.2.6	Annidamento delle interruzioni	59
3.2.7	Richieste di interruzioni simultanee	60
3.2.8	Controllo della richiesta	60
3.2.9	Registri di controllo del processore	60
3.2.10	Programmi con interruzioni	61
3.3	Eccezioni	63
4	STRUTTURA DI BASE DEL PROCESSORE	65
4.1	Concetti fondamentali	65
4.1.1	Componenti hardware	65
4.1.2	Hardware per l'elaborazione dati	66
4.1.3	Struttura hardware a più stadi	67

4.2	Esecuzione di istruzioni	68
4.2.1	Istruzioni di caricamento	68
4.2.2	Istruzioni aritmetico/logiche	69
4.2.3	Istruzioni di immagazzinamento	69
4.2.4	Schema di esecuzione delle istruzioni	70
4.3	Hardware e percorso dati	70
4.3.1	Banco di registri	70
4.3.2	ALU	71
4.3.3	Struttura a 5 stadi	72
4.3.4	Percorso dati (datapath)	73
4.3.5	Sezione di prelievo delle istruzioni	76
4.4	Passi di esecuzione	79
4.4.1	ADD	79
4.4.2	Load e Store	80
4.4.3	Salti	81
4.4.4	In attesa della memoria	82
4.5	Segnali di controllo	83
4.5.1	Segnali di controllo per l'interfaccia processore-memoria	84
4.5.2	Segnali di controllo al generatore di indirizzi	85
4.6	Controllo di tipo cablato	85
4.6.1	Generazione di segnali di controllo	86
4.6.2	Segnali di controllo nel percorso dati	87
4.6.3	Ritardi della memoria	88
4.7	Processori CISC	88
4.7.1	Organizzazione CISC	89
4.7.2	Controllo di accesso al bus	90
4.7.3	Interconnessione a tre bus	90
4.7.4	Controllo microprogrammato	93
5	INTRODUZIONE AL PIPELINING	95
5.1	Organizzazione in pipeline	96
5.1.1	Percorso dati nel pipelining	96
5.2	Problematiche nel pipelining	99
5.2.1	Dipendenza di dato	100
5.2.2	Ritardi della memoria	102
5.2.3	Ritardi nei salti	103
5.2.4	Limiti di risorse	107
5.3	Valutazione delle prestazioni	107
5.3.1	Effetti di stalli	108
5.3.2	Effetti di penalità di salto	109
5.3.3	Effetti di cache miss	109
5.3.4	Numero di stadi della pipeline	110
5.4	Processore superscalare	110
5.4.1	Organizzazione hardware superscalare	110
5.4.2	Esempio di esecuzione superscalare	111
5.4.3	Salti e dipendenze di dato	112

5.4.4	Esecuzione fuori ordine	113
5.4.5	Funzionamento dello smistamento	113
6	SISTEMA DI INGRESSO E USCITA	115
6.1	Struttura a bus	115
6.2	Funzionamento del bus	116
6.2.1	Bus sincrono	116
6.2.2	Bus asincrono	120
6.2.3	Confronto bus sincrono e bus asincrono	121
6.2.4	Pilotaggio del bus	122
6.3	Arbitraggio del bus	122
7	SISTEMA DI MEMORIA	125
7.1	Concetti di base	126
7.2	Memoria RAM a semiconduttori	126
7.2.1	Organizzazione interna di chip di memoria	127
7.2.2	Memoria statica	128
7.2.3	Memoria dinamica	129
7.2.4	Organizzazione dei collegamenti della RAM	130
7.2.5	Moduli di memoria	130
7.3	Gerarchia di memoria	131
7.4	Memoria cache e località	132
7.4.1	Uso della cache	132
7.4.2	Cache hit	132
7.4.3	Cache miss	133
7.4.4	Indirizzamento di cache	133
7.4.5	Dati scaduti	137
7.4.6	Algoritmi di sostituzione	137
I	APPENDICE	139
A	Sistemi di numerazione e rappresentazione binaria dei numeri	141
A.1	Sistemi di numerazione posizionali	141
A.1.1	Conversione di qualsiasi n in base B in base 10	141
A.1.2	Conversione di qualsiasi n in base 10 in un'altra base B .	142
A.1.3	Numeri di valori rappresentabili	142
A.1.4	Conversioni di binari in esadecimali e viceversa	143
A.2	Rappresentazione dell'informazione	143
A.2.1	Somma di due binari	144
A.2.2	Somma modulare	144
A.2.3	Rappresentazione dei numeri relativi	145
A.2.4	Trabocco	146
A.3	Rappresentazione numeri con la virgola in binario	146
A.3.1	Conversione da decimale a binario	147
A.3.2	Rappresentazione dei numeri in virgola mobile	147

A.3.3 Rappresentazione dei caratteri in binario	148
B Algebra booleana	149
B.1 Le operazioni fondamentali dell'algebra booleana	149
B.1.1 Somma logica (OR)	149
B.1.2 Prodotto logico (AND)	150
B.1.3 Negazione (NOT)	150
B.1.4 Proprietà duali di AND e OR	151
B.1.5 Teoremi di De morgan	151
B.2 Funzioni logiche ed espressioni logiche	151
B.2.1 Equivalenza tra espressioni logiche	151
B.2.2 Da una funzione a espressione logica	152
B.3 Forma minima	154
B.3.1 Passaggio da SOP a forma minima	154
B.3.2 Metodo di Karnaugh	154
B.4 Circuiti logici	155
B.4.1 Porte a più ingressi	156
B.4.2 XOR, NAND e NOR	156
B.4.3 Porte universali	158
C Circuiti integrati	159
C.1 Rappresentazione delle variabili binarie	159
C.2 Transistori	159
C.2.1 Transistori MOS	160
C.2.2 Circuiti NOT, NAND E NOR con transistor NMOS . . .	161
C.3 Circuiti CMOS	163
C.3.1 Da espressione logica a circuito CMOS	163
C.4 Ritardi di un circuito, fan-in e fan-out e porte tri-state	164
C.4.1 Fan-in e fan-out	164
C.4.2 Porte tri-state	164
C.5 Circuiti integrati	164
C.5.1 Decodificatore (decoder) e multiplatore (multiplexer) . .	165
D ALU	167
D.1 Addizionatore	167
D.1.1 Addizionatore a 1 bit	167
D.1.2 Addizionatore completo (full adder)	167
D.1.3 Addizionatore a propagazione di riporto	167
D.2 Addizionatori algebrici	168
D.2.1 Trabocco	168
D.2.2 Addizionatore algebrico a n bit	168
D.3 ALU	169
D.3.1 ALU a 1 bit	169
D.3.2 ALU a n bit	169

E Reti sequenziali	171
E.1 Bistabile asincrono	171
E.1.1 Diagramma temporale di un bistabile asincrono	172
E.2 Bistabile sincrono	173
E.2.1 Bistabile di tipo D	174
E.3 Flip-Flop	175
E.3.1 Flip-Flop master-slave	175
E.3.2 Flip-Flop di tipo T	176
E.3.3 Flip-Flop di tipo JK	176
E.4 Registri paralleli e seriali	176
E.4.1 Registri paralleli	176
E.4.2 Registri a scorrimento	177
E.4.3 Registri seriali-paralleli	178
E.5 Contatore	178

Capitolo 1

INTRODUZIONE AL CALCOLATORE

1.1 Famiglie di calcolatori

Esistono 4 tipi di calcolatori in grado di compiere azioni e algoritmi dettati dell'essere umano:

- Calcolatori **embedded**: in italiano possono essere chiamati anche come sistemi *incorporati* poichè sono delle macchine che sono state programmate per uno specifico scopo e per questo difficilmente possono essere riprogrammati, anche se ad oggi è difficile distinguere cosa sia embedded o no perchè esistono dei sistemi riprogrammabili come l'apple watch che è difficile classificare. Un esempio di sistema embedded potrebbe essere il display di una lavatrice che permettere di prendere un input e compiere azioni ben definite
- Calcolatori **personal**i: sono i normali computer che utilizziamo a scopo personale con prestazioni notevoli e potenziate rispetto a un sistema embedded(ad esempio il *desktop* o il *laptop*). Per avere delle prestazioni ancora maggiori rispetto a un calcolatore personale si utilizzano le cosiddette **workstation** che sono proprio stazioni di lavoro atte proprio a un utilizzo produttivo o lavorativo che riescono a erogare migliori prestazioni rispetto a un portatile
- **Server**: Sono delle macchine molto potenti che sono in grado di immagazzinare una enorme quantità di *richieste* e che attraverso una rete di computer riesce a comunicare questi dati offrendo servizi ad altri computer
- **Supercalcolatori**: sono dei computer che possiamo immaginare come se si trovassero dentro un "armadio" che contiene delle **lame** all'interno del quale vi sono server molto potenti con vari processori molto potenti.

Quindi attraverso questi tipi di calcolatori riusciamo ad avere il massimo delle prestazioni

1.2 Componenti funzionali

Tutti i calcolatori sono composti da:

- **Memoria:** E' la parte che utilizziamo per conservare o memorizzare i dati e i programmi da utilizzare; man mano che questi ultimi vengono processati vengono memorizzati nuovi dati
- Unità di **ingresso/uscita:** sono quei dispositivi che ci permettono di inserire (**input**) o di fare uscire dei dati (**output**). Dispositivi in input possono essere:
 - La *tastiera*, nel quale noi pigliando un tasto, memorizziamo il codice e lo trasformiamo in binario per poi inviarlo alla memoria.
 - Il *mouse*, nel quale riusciamo a vedere a video dove stiamo puntando attraverso lo scorrimento di questo.

Dispositivi output possono essere:

- Lo *schermo*, nel quale vediamo le informazioni che il processore sta elaborando.
- La *stampante*, nel quale stampiamo un foglio scannerizzato dal computer.

E infine abbiamo dispositivi ibridi, ovvero che fanno sia da input che da output, come il *touch screen*, nel quale vediamo a schermo le informazioni elaborate dal processore (output), ma possiamo contemporaneamente toccare lo schermo con le dita, il pennino... (input).

- **Rete di interconnessione:** sono collegamenti in bus, in pratica è l'unità centrale che grazie ai suoi *fili di interconnessione* riesce a collegare il processore sia alla memoria che alla periferia (dispositivi di I/O)
- **Processore:** Il processore, che è l'unità fondamentale del calcolatore poichè elabora e governa le istruzioni che avvengono al suo interno, è composto da due unità o sottosistemi fondamentali:
 - **Unità aritmetica logica:** L'unità aritmetica logica o **ALU**, è quella parte del processore che riesce a eseguire le istruzioni con operazioni logiche o matematiche, cioè è la parte "esecutiva" del processore;
 - **Unità di controllo:** L'unità di controllo ha il compito di governare ciò che avviene all'interno del processore e scandisce anche il tempo per eseguire una determinata istruzione, è in pratica la parte "governativa" del processore.

1.2.1 La memoria

Esistono diversi tipi di memoria:

- La **memoria principale** o **RAM dinamica**, che è una memoria creata attraverso diversi *transistor* ed è **volatile**, ovvero conserva e memorizza i dati fino a quando vi è alimentazione. E' una memoria abbastanza costosa ma allo stesso tempo molto veloce.
- La **memoria di massa**, che è una memoria **non volatile** o permanente poichè anche quando leviamo l'alimentazione essa non perde i dati memorizzati. E' una memoria poco costosa ma anche abbastanza lenta.

Differenze fra memoria principale e di massa

Le differenze principali tra le due memorie sono nei **costi** e nella **velocità** di memorizzazione dei dati: se infatti una memoria di massa può essere meno costosa di una memoria centrale, è sicuramente più lenta nel memorizzare i dati. Quindi ne consegue che se dobbiamo memorizzare temporaneamente dati ci converrà allocarli nella RAM dinamica piuttosto che nel disco, che è più utile nel caso in cui dobbiamo conservare dati permanentemente.

La parola di memoria

Sappiamo che il computer utilizza i **bit** per memorizzare i dati, per cui ogni numero o parola viene codificata in una serie di bit. Ad esempio se abbiamo bisogno di scrivere delle parole solitamente facciamo uso di 8 bit ma se volessimo codificare stringhe, numeri, o dati con lunghezze maggiori avremo sicuramente bisogno di un numero superiore di bit. I processori di oggi, infatti, sentiamo molto spesso dire che riescono a gestire dati fino a 32 o 64 bit. Ciò significa che riescono ad utilizzare questo numero di bit tutti contemporaneamente e, quest'unica informazione che riescono a gestire viene chiamata **parola di memoria**. Quando ad esempio dobbiamo memorizzare un indirizzo nella RAM, possiamo utilizzarne uno da 32 o 64 bit, dipende dal processore ovviamente, attraverso 32 o 64 "fili" di bus, ovvero quella famosa rete di interconnessione.

La memoria cache

Il processore è molto più veloce della memoria ma, abbiamo una memoria che sta in mezzo tra processore e RAM dinamica: la **memoria cache**. Essa ha una velocità maggiore rispetto alla RAM dinamica, infatti se il processore deve leggere un dato dalla RAM dinamica lo si deve trasportare alla memoria cache e poi lo può leggere poichè questo tipo di memoria si trova internamente nel chip del processore. Se invece deve leggere un dato dalla memoria cache, questo processo avviene molto più velocemente poichè riesce ad accedere a quest'ultimo direttamente senza "trasportarlo". Questo tipo di memoria viene anche chiamato **RAM statica**.

Cosa significa RAM dinamica o statica?

Ma cosa significa quindi effettivamente RAM dinamica e RAM statica? Potremmo pensare che c'entri il fatto che questo tipo di memoria sia volatile ma in realtà non è così:

- **RAM dinamica:** questo tipo di RAM nonostante ci sia l'alimentazione tenderà a perdere i dati memorizzati se non rinfrescati, poichè col passare del tempo la corrente all'interno della cella di memoria viene dissipata. Quando diciamo che dobbiamo rinfrescare il dato significa che dobbiamo riprendere il dato e rileggerlo con la massima corrente possibile ad ogni intervallo di tempo. In media il tempo di risposta per fornire un dato, di una RAM dinamica, è di 20 nanosecondi circa.
- **RAM statica:** a differenza della precedente questo tipo di memoria non ha bisogno di rinfrescare il dato, per cui il processo di lettura di quest'ultimo avviene molto più velocemente alla memoria RAM dinamica, però è ovviamente molto più costosa in termini di quantità di transistor (possiamo immaginare che se in media una RAM dinamica presenta 2 transistor, quella statica ne presenta 8), e in media fornisce un dato in 2 nanosecondi.

1.2.2 Il processore

Come avevamo accennato prima il processore è composto da due unità fondamentali:

- **Unità aritmetica-logica:** la parte esecutiva del processore che prende gli operandi e fornisce risultati attraverso varie operazioni. L'ALU utilizza i **registri** come memoria interna: ogni operazione infatti prima di essere eseguita deve essere letta nel registro apposito, come anche qualsiasi tipo di dato. In seguito quest'operazione passa all'ALU che la esegue. Questa è una memoria molto più veloce anche della cache (infatti può leggere un dato anche in 0,5 nanosecondi), ma è ovviamente presente in minor quantità per il suo costo elevato. Inoltre la dimensione dei registri va di pari passo con la parola di memoria, per cui se ad esempio abbiamo a disposizione 64 bit, i registri ne utilizzano 64.
- **Unità di controllo:** governa il coordinamento delle operazioni del processore e ne scandisce il tempo, per cui deve riconoscere se sono stati eseguiti tutti i passi di un'operazione e in caso allungare il tempo della stessa o accorciare in base a quanto tempo c'è bisogno per concluderla. Ad esempio il processore non sa né quanto dura il processo di lettura della memoria né da dove prendere i dati dalla memoria; sarà l'unità di controllo a verificare se un'informazione è stata letta correttamente e in caso passare allo step successivo.

Funzionamento del processore

In breve il processore esegue le operazioni seguendo queste fasi:

1. Prende un'istruzione in memoria
2. Codifica quest'istruzione
3. Governa le azioni successive da compiere
4. Memorizza l'informazione nei registri

Quando finisce un'istruzione il processore passa alla successiva seguendo gli stessi passi. Il flusso può interrompersi solo quando si verifica un *interrut*, cioè un fattore che ferma il ciclo di operazioni del processore

1.3 Concetti operativi di base

Ovviamente le operazioni che deve compiere il processore dipendono esclusivamente dal programma che stiamo eseguendo, quindi, dobbiamo dargli una serie di istruzioni in sequenza. Inoltre con questa serie di istruzioni mappiamo la codifica 0,1 in un linguaggio simbolico comprensibile all'umano (**linguaggio assembly**). Il processore riconosce 3 tipi di operazioni:

- **Trasferimento dalla memoria al processore:** è l'istruzione che permette di prendere un dato dalla memoria e trasportarlo al processore. Scriviamo l'istruzione in questo modo:

`Load R2,LOC`

Analizziamo questa scrittura:

- Load: **codice operativo**, è quello che ci dice cosa fare. In questo caso ”carica”, ovvero trasferisce dei dati dalla memoria al processore
- R2,LOC: **operandi**, il primo operando in qualsiasi operazione è sempre un **registro** con un numero che indica la posizione di quest'ultimo. Il secondo operando è una **variabile** che ci indica la locazione di memoria, viene chiamato anche *identificatore simbolico*

In questo caso stiamo dicendo: ”devi eseguire quest'istruzione prendendo la locazione di memoria(**LOC**), poi leggi il dato presente al suo interno e trasferiscilo (**LOAD**) al registro ¹(**R2**)”

- **Operazioni aritmetiche:**

`ADD R4,R3,R2`

¹Il processore non sa da dove prelevare i dati dalla memoria, ma da l'indirizzo al bus

Analizziamo:

- ADD: **codice operativo**, significa "aggiungi" o somma
- R4,R3,R2: **operandi**, il primo operando è il registro dove trasferiremo il risultato (questo registro viene chiamato *destinazione*), mentre degli altri due operandi dobbiamo prendere i valori all'interno dei rispettivi registri e sommarli (questi due registri vengono chiamate *sorgenti*)

In questa istruzione diciamo: "Prendi i valori di **R3** e **R2**, sommali (**ADD**), e memorizza il risultato in **R4**"

- **Trasferimento dal processore alla memoria:**

STORE R4,LOC

Analizziamo:

- STORE: **codice operativo**, "immagazina"
- R4,LOC: **operandi**, il primo operando ci dice di prendere il contenuto da quel registro (**R4**), il secondo ci dice di trasferirlo nella locazione di memoria che ha come variabile **LOC**

1.3.1 Connessione fra processore e memoria

Abbiamo accennato al fatto che i registri appunto salvino dei dati o delle istruzioni al loro interno. Nel processore vi sono alcuni registri specializzati per alcune funzioni: il registro **PC** o *program counter* che contiene al suo interno l'**indirizzo** della parola di memoria con l'istruzione de eseguire e il registro **IR** o *instruction register* che contiene l'istruzione corrente e permette alle varie componenti del processore di eseguire quest'istruzione correttamente, quindi la **decodifica**. Quindi ciò che avviene è che il PC "punta" all'indirizzo con la parola di memoria con l'istruzione da eseguire e la trasferisce all'IR che, conterrà l'istruzione in tutti i suoi passi e sovrascriverà la precedente istruzione. Quest'interfaccia permette insomma di dialogare tra processore e memoria, ciò avviene attraverso dei **segnali di controllo** che permettono di far capire al processore o alla memoria come comportarsi: vi è ad esempio il segnale di **lettura** inviato dal processore alla memoria che appunto permette di dire alla memoria di mandare un dato o un istruzione da un determinato indirizzo al processore. Un altro segnale, inviato dalla memoria al processore è quello di **scrittura** in cui la memoria dice al processore di scrivere quel determinato dato in un registro.

1.3.2 Passi dell'istruzione

Quindi, riassumendo ciò che abbiamo detto, possiamo dire che un'istruzione per essere eseguita esegue questi passi:

- Il programma con la lista di istruzioni viene caricato nella memoria principale
- L'istruzione viene **prelevata** dalla memoria principale dal PC che **punta** alla parola di memoria corrispondente con l'istruzione da eseguire
- L'istruzione viene **trasportata** dal PC all'IR. Il PC punta all'istruzione successiva
- Nell'IR l'istruzione viene **decodificata**
- L'istruzione viene **eseguita**

1.3.3 Interruzioni

Mentre eseguiamo un'istruzione può capitare che arrivi un **segnale di interruzione** che fa fermare temporaneamente il programma che il processore stava eseguendo in quel determinato momento. Questo segnale arriva dalla periferia, ovvero dai dispositivi di input (tastiera, mouse...), e dice al processore di eseguire la cosiddetta **routine di interruzione** che è un altro programmino che esegue l'istruzione mandata dalla periferica, ad esempio se pigiamo un tasto della tastiera mentre il processore sta eseguendo un programma, questo si fermerà per eseguire la routine che permette di decodificare l'istruzione inviata pigiando quel determinato tasto. Una volta eseguita la routine il processore riprende l'esecuzione del programma che aveva lasciato in pausa poichè aveva salvato i dati necessari alla ripresa di tale programma. E' molto importante comprendere questo tipo di segnali poichè fanno sì che l'esecuzione di istruzioni non sia **deterministica**, cioè non sappiamo sempre esattamente quello che sta accadendo all'interno del processore e quindi ogni programma può dipendere anche dai segnali che arrivano dall'esterno, ovvero dalla periferia.

1.4 Prestazioni di un processore

Per misurare le prestazioni di un processore ci si basa sulla **velocità** di esecuzione delle istruzioni da parte del processore e dalla **tecnologia** con cui vengono sviluppate le componenti funzionali.

- Per quanto riguarda la velocità di esecuzione i **transistor** sono fondamentali: quando leggiamo un dato in un registro, esso viene decodificato in 0 o 1 attraverso la corrente che passa in questi transistor; più velocemente questa corrente passa dal 0 all' 1, più velocemente l'istruzione viene eseguita. Questa velocità viene chiamata **velocità di commutazione** e dipende dalle **dimensioni** di un transistor: più è piccolo e più aumenta la velocità, poichè non solo aumenta la velocità di commutazione ma c'è la possibilità di inserire molti più transistor

- Per quanto riguarda invece l'**organizzazione dell'hardware** invece abbiamo il concetto fondamentale di **parallelismo** che viene sviluppato sia nelle istruzioni, che nei multicore che anche nei sistemi con piu processori:
 - A livello di **istruzioni** il concetto di parallelismo si basa nel eseguire piu istruzioni contemporaneamente attraverso il **pipelining**
 - A livello di **core**: un chip, che è il circuito integrato che contiene al suo interno il processore, può possedere piu core (parallelismo dei multicore) che sono le unità di elaborazione del processore. Ogni core ha una **cache** dedicata.
 - A livello di **sistema**: esistono calcolatori, come i supercalcolatori o i server, che possiedono piu processori che comunicano tutti fra loro. Essi possono avere una memoria **condivisa**, ovvero un'unica memoria centrale oppure possono lavorare con memorie **distinte**.

Capitolo 2

ISTRUZIONI MACCHINA

2.1 Introduzione

Quando viene progettato un processore, esso sarà caratterizzato da una serie di istruzioni mappate in linguaggio macchina, chiamate **ISA** o *instruction set architecture*, che ci indicano:

- Il nome delle istruzioni eseguibili mediante quel processore e le operazioni che riescono a fare
- I dati che possiamo utilizzare
- Le regole di combinazione di tali dati

L'ISA quindi non è altro che l'**interfaccia fra hardware e software**; se qualcuno volesse quindi programmare tale processore deve per forza riferirsi all'ISA. E ovviamente se noi scriviamo un programma con un linguaggio *ad alto livello* verrà tradotto da un **compilatore** in linguaggio macchina; mentre se scriviamo in un linguaggio *a basso livello* come l'**assembly** esso rappresenta la versione leggibile del linguaggio macchina

2.2 Memoria del calcolatore

Partendo dal fatto che la memoria è organizzata in tantissime **celle** che possono contenere solo un **bit** (0 o 1), possiamo supporre che queste singole celle prese a sole ci possono restituire informazioni insignificanti. Per questo queste celle sono state raggruppate in gruppi di celle contenenti n bit, solitamente pari a **potenze di 2** chiamate **parole di memoria**. Oggi le parole di memoria variano su lunghezze di 16-64 bit, in base al processore utilizzato. In questo senso possiamo dire che la memoria consiste nella successione di parole di memoria che sono numerate da 0 a $n - 1$, dove n rappresenta il **vettore di parole**, che sarebbe l'insieme di tutte le parole presenti in memoria. Ogni parola può contenere un numero, una parola o un dato della lunghezza della parola: ad esempio se

abbiamo una parola di memoria di 32 bit puo contenere un numero di 32 bit oppure se ad esempio volessimo codificare una parola di memoria che al suo interno presente dei caratteri ASCII, potremmo codificarne 4 poichè sappiamo che un carattere ASCII è lungo 8 bit = 1 **byte**.

2.2.1 Indirizzamento

Ora se volessimo prelevare una specifica parola di memoria contenente quindi una specifica istruzione dovremmo specificarne la posizione: questo viene fatto attraverso gli **indirizzi** che possono essere richiamati e presentano al loro interno la **locazione di memoria** che ha il suo interno una specifica parola; gli indirizzi con numero più in basso, se immaginassimo la memoria graficamente, si troverebbero in alto e viceversa quelli con numero più alto si troverebbero in basso nella memoria. Per essere rappresentati o specificati gli indirizzi occorre un numero di **m bit**, che è quasi sempre diverso dai bit che usiamo per rappresentare la parola di memoria. Lo **spazio di indirizzamento** o *addressing space*, invece, è la quantità di indirizzi che il processore può utilizzare per indirizzare le parole di memoria ed è uguale a 2^n . Se ad esempio avessimo un indirizzo di 24 bit, possiamo indirizzare 2^{24} parole di memoria che sono circa 16 milioni di parole o 16 Mega parole.

2.2.2 Bus di indirizzi

Il dialogo che il processore ha con la memoria e con le altre periferiche, ovvero la famosa **rete di interconnessione** formata da linee di bus (ovvero piccoli circuiti) di cui abbiamo accennato all'inizio non è formata solo da **bus di dati**, che sono le linee di collegamento tra il processore e la memoria nelle quali viaggiano dati, ma a questa si aggiungono il **bus di indirizzi** e il **bus di controllo**:

- Il primo quindi rappresenta quelle linee di collegamento tra processore e memoria nella quale viaggiano gli indirizzi delle locazioni di memoria
- Il secondo rappresenta invece quelle linee di collegamento dove passano i segnali che il processore manda alla memoria, ovvero il **read** in cui il processore chiede alla memoria qualcosa, e viceversa i segnali che la memoria manda al processore, ovvero il **write** in cui la memoria dice al processore che può scrivere un dato, istruzione.. dentro la memoria

Riflessione sull'indirizzamento Quando acquistiamo un carto processore, che mettiamo caso codifichi parole di 32 bit, dobbiamo assicurarci che possa indirizzare tutte la parole. Se per esempio avessimo bisogno di indirizzare 16 GIGA parole, noi in realtà con un processore da 32 bit potremmo indirizzarne solo $2^{32} = 4$ GIGA parole, ovvero solo un quarto. Con questo comprendiamo il perchè del fatto che il numero di bit per la rappresentazione di un indirizzo sia solitamente minore rispetto ai bit della parola di memoria: tutto ciò risulterebbe sconveniente sia in termini di hardware, bus di collegamenti ecc.

2.2.3 Indirizzamento a byte

Abbiamo constatato che $8 \text{ bit} = 1 \text{ byte}$, con un byte quindi l'informazione che può essere codificata può avere un senso e può essere utile: potrebbe essere una lettera o un carattere particolare. Per questo è nato l'**indirizzamento a byte**, proprio per l'esigenza di indirizzare non solo una parola di memoria ma anche uno specifico byte. Quindi possiamo immaginare come se suddividessimo l'intera parola di memoria, costituita da 32 bit, in sequenze da 8 bit (ovvero 1 byte), come risultato abbiamo che se la **prima** parola di memoria ha indirizzo 0 e la **seconda** per logica, avendo "diviso in 4" la parola, avrà indirizzo 4. Quindi se specificassimo indirizzo 1, 2, 3 diremmo la locazione di memoria contenente un byte della parola che ha numero 1,2,3. Ciò significa che abbiamo due comandi a disposizione: prelevare l'**intera** parola di memoria o il **singolo** byte.

Ordinamento di byte

Soltamente per conoscere precisamente l'indirizzo di un byte si dovrebbe sapere come sono numerati poichè abbiamo due tipi di "schemi" di numerazione:

- Schema **crescente**, ovvero ordiniamo gli indirizzi partendo da 0 da sinistra verso destra
- Schema **decrescente**, ovvero ordiniamo gli indirizzi partendo da 0 da destra verso sinistra

Lo schema più diffuso è quello decrescente poichè presenta le **cifre significative** a sinistra. Inoltre se il processore manda un indirizzo multiplo della lunghezza della parola in byte, solitamente i multipli di 4, si dirà **allineato**

2.3 Operazioni di memoria

Le operazioni di base che il processore può eseguire con la memoria o viceversa sono due:

- **Lettura** o *read* o LOAD: un dato viene prelevato da un indirizzo di memoria e viene scritto in un registro del processore
- **Scrittura** o *write* o STORE: un dato viene passato dal processore alla memoria e viene salvato in un indirizzo di memoria

2.4 Istruzioni macchina

Esistono 4 categorie di istruzioni macchina:

- **Trasferimento** dei dati (Load e store)
- Operazioni **aritmetiche** o **logiche** sui dati (somma, prodotto, and, or, not..)

- Controllo della **sequenza di esecuzione** delle istruzioni attraverso salti che fanno tornare indietro il flusso di esecuzione delle istruzioni oppure "saltano" avanti
- Trasferimento di dati dalle **periferiche di input** verso il processore o dal processore verso le **preiferiche di output**

2.4.1 Notazione RTN

La *Register transfer notation*, ovvero la **notazione RTN** usa delle convenzioni e dei simboli per rappresentare le istruzioni macchina in un linguaggio comprensibile all'umano ed è molto vicino al linguaggio assembly. Utilizza:

- **Costanti** numeriche o simboliche per indicare indirizzi di parole di memoria, es: IND
- **Nome dei registri:** R1, R2...
- **Parentesi quadre**, indicano di prelevare il contenuto presente nell'indirizzo. Es: [IND], prelevo il contenuto presente nell'indirizzo IND
- **Freccia verso sinistra**, indica il trasferimento o la copia di un valore. Es: $R2 \leftarrow [LOC]$, il valore a sinistra della freccia rappresenta la **destinazione**, che può essere quindi o un registro o una locazione di memoria, a destra indichiamo la **sorgente**; nel caso in esempio stiamo dicendo prendi il contenuto presente nella memoria con indirizzo LOC e trasferiscilo nel registro di indice R2

2.4.2 Notazione simbolica

Trattiamo adesso le istruzioni presenti nel **linguaggio assembly**: ogni istruzione specifica un'**operazione** da eseguire e gli **gli operandi** che utilizziamo per cui abbiamo:

- Un **codice operativo**, che specifica attraverso l'uso di parole inglese quali LOAD, STORE, ADD.. l'operazione che stiamo andando a fare
- Gli **operandi**, di cui il primo rappresenta la **destinazione** nella quale allocare l'operazione, ovvero un registro o una locazione di memoria, e gli altri rappresentano la **sorgente**

Esempio

```
LOAD R2,LOC
```

In questo caso il codice operativo ci sta dicendo di trasferire un dato che è presente nella memoria che ha come indirizzo la variabile LOC, inserendolo nel registro di denominazione R2

2.5 Architetture RISC e CISC

Quando viene progettato un processore, ci sono due tipologie di realizzazione:

- Architettura **RISC**, o *reduced instruction set computer*: ogni istruzione occupa esattamente **una parola di memoria**. Questo semplifica molto la realizzazione dell'hardware che, occuperà minor spazio a causa del minor numero di transistor. Nel corso degli anni, si è visto inoltre che questo tipo di architettura molto spesso riesce ad essere più veloce ed efficiente rispetto a un CISC. Ovviamente però potendo utilizzare la lunghezza della parola di memoria come numero di bit fissati per ogni istruzione, dobbiamo limitare ognuna di queste alla parola di memoria, e, di conseguenza, ciò comporta dover utilizzare più istruzioni RISC per eseguire ad esempio una semplice somma.
- Architettura **CISC**, o *complex instruction set computer*: in questo caso non abbiamo più una lunghezza predefinita, ma la lunghezza di bit di ogni istruzione è **variabile**; quindi rispetto a un'istruzione assembly RISC, utilizziamo una singola istruzione con una lunghezza di bit maggiore. La conseguenza di non avere una lunghezza di bit predefinita per ogni istruzione comporta innanzitutto una costruzione di hardware più complessi con un numero elevato di transistor; e, inoltre, ogni istruzione sarà molto complessa a causa del fatto che può occupare molto di più di una parola di memoria, e quindi, si deve vedere anche quanto occupa un'istruzione per eseguire la successiva. Fino a un paio di anni fa, inoltre, si pensava che un'architettura di tipo CISC fosse più veloce di una RISC, ad oggi invece, attraverso lo sviluppo tecnologico le architetture RISC sono le più veloci

2.6 Istruzioni RISC

Dalla descrizione di un'architettura RISC abbiamo compreso che ogni istruzione occupa una parola di memoria. Aggiungiamo inoltre che si basa su una architettura **load/store**: ogni valore deve essere preso dalla memoria e trasferita in un registro, si fanno le varie operazioni aritmetico-logiche e si scrive il nuovo valore di nuovo in memoria, nel CISC non avrei bisogno di scrivere tutte queste istruzioni. Quindi attraverso una **LOAD**, avviene l'accesso in memoria in modo tale da prendere un valore e attraverso la **STORE**, scrivo un valore in memoria¹.

2.6.1 Formato di istruzioni

Vediamo ora come scrivere i vari tipi di istruzioni RISC in **assembly**:

- **LOAD d,s**

¹**NB:** queste sono le uniche 2 istruzioni che mi permettono di accedere in memoria o di scrivere qualcosa in memoria, nelle operazioni aritmetico logiche infatti i valori vengono presi esclusivamente o dai registri o dall'istruzione stessa

Nella load la prima variabile è la destinazione, che è sempre un **registro**, mentre la seconda è la sorgente, ovvero la **locazione di memoria** che ha indirizzo indicato con la variabile s. Quindi prende il valore della sorgente e lo copia nella destinazione

- **STORE s,d**

Nella store, al contrario della load, la prima variabile è la sorgente, quindi il registro da dove preleviamo il valore; la seconda variabile rappresenta la destinazione, ovvero la locazione di memoria all'interno della quale vogliamo scrivere un valore

- **ADD d, S1,S2**

ADD è un'istruzione aritmetica che indica appunto la somma: in questo caso la prima variabile è la destinazione, che è sempre un registro, mentre le altre due rappresentano gli **operandi** da sommare, ovvero la sorgente. In alcuni casi il registro di destinazione coincide con il registro della sorgente: in tal caso il valore presente nel registro sorgente viene sovrascritto con il nuovo valore restituito dall'operazione in questione.

2.6.2 Operazione di addizione

Quindi quali sono la serie di istruzioni da eseguire per ottenere un'**operazione di somma** nel linguaggio assembly?

- In un linguaggio ad alto livello possiamo scrivere un'operazione di somma assegnando a una variabile C la somma dei valori contenuti nelle variabili A e B:

$$C = A + B$$

- In notazione RTN, più vicina al linguaggio assembly, avremmo scritto:

$$C \text{ (freccia verso sinistra)} [A] + [B]$$

Quindi prendi il contenuto della variabile dell'indirizzo della locazione di memoria con simbolo A, sommalo con il contenuto della variabile dell'indirizzo della locazione di memoria con simbolo B, e copia la somma in C

- LOAD R2, A
LOAD R3, B
ADD R4,R3,R2
STORE R4, C

In linguaggio assembly abbiamo:

- caricato dalla memoria il valore della variabile che ha indirizzo di memoria con simbolo A

- caricato dalla memoria il valore della variabile che ha indirizzo di memoria con simbolo B
- preso i valori contenuti nei registri R3 e R2 e sommati; infine copiato il risultato della somma in R4
- Copiato il valore presente nel registro R4 e traferito nella locazione di memoria che ha indirizzo con simbolo della variabile C

2.6.3 Esecuzione delle istruzioni

Abbiamo compreso che con un'architettura RISC ogni istruzione occupa esattamente una parola di memoria, che assumiamo in generale con lunghezza di 32 bit. Quindi vediamo come il PC, ovvero il registro che punta gli indirizzi successivi, agisce. Prendiamo come esempio l'addizione vista in precedenza:

Indirizzo	Contenuto
i	LOAD R2,A
i + 4	LOAD R3, B
i + 8	ADD R4,R3,R2
i + 12	STORE R4, C

Table 2.1: Indirizzi a cui punta il PC

Quindi notiamo che il registro PC, ogni istruzione successiva, punterà all'indirizzo che ha numero incrementato di 4 poichè facendo riferimento all'indirizzamento a byte, una parola di memoria da 32 bit è costituita da 4 byte. In seguito una volta che il PC ha puntato l'indirizzo contenente l'istruzione lunga una parola di memoria; avverrà il **prelievo dell'istruzione** da parte del registro IR, che preleverà il contenuto dell'indirizzo, utilizzato come locazione di memoria per prendere l'istruzione, decodificarla ed eseguirla.

2.6.4 Esecuzione di n somme e salto

Ora costruiamo il nostro primo programma in assembly: sommare n numeri interi, con n dato in input e $n > 1$. Ragioniamo sulle varie istruzioni successive da eseguire:

- dobbiamo **prelevare** i valori dalla memoria;
- dobbiamo **trasferirli** nei registri e farne la somma, che può contenere al massimo due operandi
- dobbiamo scrivere il risultato finale in memoria

Utilizziamo in questo programma il registro R2 come accumulatore parziale, mentre in R3 sovrascriviamo i successivi n numeri da sommare con il valore di R2:

```

LOAD R2, NUM 1
LOAD R3, NUM 2
ADD R2, R2, R3
LOAD R3, NUM 3
ADD R2, R2, R3
...
LOAD R3, NUMn
ADD R2, R2, R3
STORE R4, SOMMA

```

Quindi facciamo varie somme parziali con i valori contenuti nel registro R2, che funge da accumulatore parziale²; e sovrascriviamo gli n numeri in R3 poi da sommare con il contenuto di R2 attraverso l'operazione **ADD**. Una volta che abbiamo sovrascritto in R3 il numero n, dobbiamo fare l'ultima somma e copiarla in memoria nell'indirizzo che ha locazione di memoria con variabile somma. Notiamo che scrivere tutte queste istruzioni sarebbe dispendioso in termini di tempo e funzionalità, ed è per questo che esiste il **salto**.

Esecuzione con salto

In assembly esiste l'istruzione di **salto**, che ci permette effettivamente di creare un **ciclo**, ovvero un tipo di istruzione che ci permette di eseguire un blocco di istruzioni finché la condizione che imponiamo risulta falsa. Ora riscriviamo il programma attraverso l'istruzione di salto e utilizziamo:

- N, **CICLO**, **SOMMA** come costanti, ovvero le variabili che hanno indirizzo che punta alla parola di memoria corrispondente. Nel caso di N, conterrà il numero di cifre da sommare; SOMMA conterrà il risultato della somma e CICLO è un marcitore o segnaposto che indicherà dove eseguire il salto e quindi l'inizio del corpo del ciclo.
- Il registro R2 verrà utilizzato come **contatore**, ovvero riusciremo a vedere con questo registro il numero di passate³ che il ciclo dovrà compiere prima di uscire ed eseguire l'istruzione seguente
- Il registro R3 verrà utilizzato come **accumulatore parziale** delle varie somme
- Il registro R5 contiene il numero contenuto nella locazione di memoria con indirizzo che viene puntato dal PC nelle istruzioni successive. Cioè rappresenta il numero caricato dalla memoria
- Indichiamo un **valore immediato** all'interno dell'istruzione con il simbolo #, cioè diciamo al processore di leggere un valore numerico che ci viene dato all'interno dell'istruzione

²sovrascriviamo le varie somme parziali in R2

³Indica il numero di volte che devono essere effettuate le istruzioni dentro al ciclo

Quindi vediamo come scriviamo il programma:

```

LOAD R2, N
CLEAR R3
CICLO: Determina l'indirizzo del prossimo numero
      Carica il prossimo numero nel registro R5
      ADD R3, R3, R5
      SUBTRACT R2, R2, #1
      Brench_if_[R2]>0 CICLO
      STORE R3, SOMMA
  
```

Vediamo adesso la logica del programma:

- Come già accennato abbiamo caricato N nel registro R2 con una **LOAD**; e con una **CLEAR** abbiamo "pulito" il contenuto di R3, ovvero abbiamo caricato li dentro il valore 0
- Abbiamo posto l'**etichetta** CICLO, che fungerà da segnaposto per l'inizio del ciclo. Ciò che abbiamo scritto dopo i due punti verrà sostituito da istruzioni nel quale diciamo al registro PC di puntare all'indirizzo del prossimo numero e caricarlo nel registro R5
- Abbiamo eseguito la somma con una **ADD** dei valori contenuti in R3 e R5 e poi abbiamo sovrascritto questo nuovo valore nell'accumulatore parziale R3
- Abbiamo aggiornato il contatore con l'operazione **SUBTRACT**, ovvero "sottrazione" sottraendo al valore contenuto in R2 (inizialmente N) 1 in modo tale da dire che abbiamo eseguito un'operazione di somma. Questo valore sarà fondamentale per rendere poi falsa la condizione del salto
- Abbiamo scritto l'istruzione di salto con **Breach if** che è il codice operativo che indica questa istruzione; abbiamo imposto la condizione $[R2] > 0$ ⁴, ovvero, fino quando il valore del contatore è > 0 , esegui il ciclo; quando è uguale a 0 esci dal ciclo ed esegui l'istruzione successiva. Abbiamo delimitato l'inizio del corpo del ciclo con l'etichetta CICLO dicendo al PC di puntare all'indirizzo con parola di memoria che presenta l'istruzione contenuta nella variabile dell'indirizzo "CICLO". Abbiamo quindi detto di saltare all'etichetta ciclo e da lì eseguire le istruzioni che compongono il corpo del ciclo
- Infine, con una **STORE**, abbiamo copiato il valore del risultato delle somme parziali nella variabile che ha indirizzo contenente una locazione di memoria con simbolo SOMMA.

⁴**NB:** la condizione può essere anche un confronto tra registri (*Breachif[R2] > [R4]*)

2.7 Modi di indirizzamento

Nelle varie istruzioni che scriviamo in linguaggio assembly abbiamo diverse possibilità per indicare l'indirizzo dei vari operandi (liste, array...). Per questo esistono diversi **modi di indirizzamento**, ovvero per accedere alla grande diversità di dati che dobbiamo utilizzare. Abbiamo diversi modi di indirizzamento:

- **Modo di registro:** nell'istruzione il dato o l'operazione è contenuta nel registro Esempio:

```
ADD R3,R3,R5
```

- **Modo assoluto o diretto:** nell'istruzione il dato o l'operazione si trova all'interno di una variabile che indica l'indirizzo associato alla locazione di memoria che contiene il contenuto di tale variabile. Esempio:

```
LOAD R2, NUM1
```

- **Modo immediato:** specifichiamo all'interno dell'istruzione l'indirizzo della locazione di memoria di una determinata costante⁵ e la indichiamo con il carattere #. Esempio:

```
ADD R3,R5,#200
```

- **Modo indiretto:** nell'istruzione specifichiamo un registro al cui interno vi è l'indirizzo che punta a una determinata parola di memoria. Il registro fa quindi da **puntatore**, verso una determinata parola di memoria, specificata dall'indirizzo, prende il contenuto dell'indirizzo e lo trasferisce in un registro di destinazione. Esempio:

```
LOAD R2, (R5)
```

In quest'ultimo caso stiamo dicendo di prendere l'indirizzo contenuto in R5 per puntare alla locazione di memoria associata al registro, prendere il contenuto e trasferirlo in R5. Possiamo vedere adesso il programma di somma di una lista di numeri utilizzando il modo indiretto, in modo tale da riuscire a scrivere delle istruzioni nell'etichetta CICLO.

⁵NB: possiamo inserire una sola costante

2.7.1 Programma per la somma di una lista di n numeri

Conoscendo il modo di indirizzamento indiretto scriviamo il programma per la somma di una lista di n numeri:

```

LOAD R2,N
CLEAR R3
MOVE R4, #NUM1
CICLO: LOAD R5, (R4)
       ADD R3, R3, R5
       ADD R4, R4, #4
       SUBTRACT R2, R2, #1
       BRANCH_IF_[R2] > 0
       STORE R3, SOMMA

```

Ragioniamo quindi su come abbiamo costruito il programma:

1. Abbiamo caricato la lista nel registro R2, che sarà quindi il **contatore**;
2. Abbiamo "azzerato" il contenuto di R3, che sarà quindi il nostro **accumulatore** di somme parziali
3. Abbiamo utilizzato il codice operativo **move** per indicare di spostare l'indirizzo di un generico **valore numerico** NUM1 in R4
4. Abbiamo inizializzato l'**etichetta** CICLO, ovvero il punto di partenza del corpo del ciclo, dicendo di puntare all'indirizzo contenuto in R4, e, di mettere il contenuto della locazione di memoria associato a quell'indirizzo nel registro R5;
5. Abbiamo **sommato** i contenuti delle variabili R3 e R5, sovrascrivendo R3 col nuovo valore trovato
6. Abbiamo aggiornato l'**indirizzo** a cui dovrà puntare R4 aggiungendogli 4, poiché ricordiamo che se dobbiamo puntare alla parola di memoria successiva contenente il prossimo numero con un indirizzamento a byte,abbiamo bisogno di aggiungere 4 byte per arrivare alla prossima parola di memoria (32 bit= 4 byte)
7. Abbiamo aggiornato il **contatore** R2 sottraendogli 1, ovvero stiamo dicendo che abbiamo fatto la prima operazione
8. Abbiamo imposto la **condizione del salto** dicendo che fino a che il valore contenuto in R2 è maggiore di 0, puoi fare il salto all'etichetta CICLO, quando è uguale a 0, vai alla prossima istruzione
9. Infine abbiamo **immagazzinato** il valore di R3, accumulatore contenente tutte le somme parziali, in una variabile di un indirizzo associato a una locazione di memoria che chiamiamo "SOMMA"

2.7.2 Esempio di indirizzamento indiretto

In un linguaggio di alto livello un’istruzione che dice di puntare all’indirizzo di una variabile e assegnare il valore contenuto in quell’indirizzo a un’altra variabile è molto semplice :

```
A = *B
```

Quest’operazione è chiamata **deferenziazione** e ha un corrispettivo in linguaggio assembly:

```
LOAD R2, B
LOAD R3, (R2)
STORE R3, A
```

Quindi abbiamo caricato dalla memoria il contenuto dell’ indirizzo associato alla variabile B; abbiamo puntato all’indirizzo contenuto in R2 e caricato il suo contenuto in R3; e infine abbiamo immagazzinato il contenuto di R3 nella locazione di memoria A.

2.7.3 Modo di indirizzamento con indice e spiazzamento

Se volessimo però utilizzare dei vettori o array all’interno del nostro programma avremmo bisogno di un **valore costante** sommato a un **registro** per arrivare all’indirizzo a cui voglio puntare. Per questo esiste il modo di indirizzamento con indice e costante:

```
LOAD R2, 20(R5)
```

Quindi stiamo dicendo punta all’indirizzo che ottieni sommando il valore costante (**spiazzamento**) con il contenuto del registro (**indice**) e trasferisci il contenuto dell’indirizzo in R5. Se quindi riprendessimo il programma di prima, in cui, anzichè aggiornare il registro puntatore aggiungendo 4 e traferire questo risultato in un ulteriore registro, potremmo scrivere questo:

```
LOAD R2, (R5)
...
...
LOAD R2, 4(R5)
```

Esempio indirizzamento con indice e spiazzamento Calcolare il voto totale degli studenti delle prove data una lista di record, formata da numero di matricola, voto della prima prova, voto della seconda prova e voto della terza prova, calcolare il voto di ogni studente sommando i voti ottenuti ad ogni prova; escludendo il numero di matricola dalla somma:

```
MOVE R2, #LISTA
CLEAR R3
CLEAR R4
```

```

CLEAR R5
LOAD R6, N
CICLO: LOAD R7, 4(R2)
        ADD R3,R3,R7
        LOAD R7, 8(R2)
        ADD R4,R4,R7
        LOAD R7, 12(R2)
        ADD R5,R5,R7
        ADD R2,R2,#16
        SUBTRACT R6, R6, #1
        BRANCH_IF_[R6]>0 CICLO
        STORE R3, SOMMA 1
        STORE R4, SOMMA 2
        STORE R5, SOMMA 3
    
```

Andiamo ad analizzare questo programma che fa uso dell'indirizzamento con indice e spiazzamento:

- Con l'istruzione **MOVE** portiamo il numero dell'indirizzo di partenza nel registro R2, che in seguito aggiorneremo e "puliamo", con un'istruzione **CLEAR**, gli **accumulatori** R3, R4 ed R5, delle somme parziali delle tre prove per i vari studenti;
- Carichiamo dalla memoria con l'istruzione **LOAD** il valore della variabile di indirizzo N che punta alla locazione di memoria contenente il numero di elementi della lista;
- Poniamo un'etichetta **CICLO**, che è una variabile che indicherà al PC dove puntare per l'istruzione di salto, e carichiamo nel registro R7 il contenuto dell'indirizzo di R2+4, ovvero contenente la parola di memoria che indica il voto della prima prova;
- Sommiamo il voto della prova 1 con l'accumulatore R3, e sovrascriviamo R3
- Ripetiamo la stessa azione per gli altri 2 voti;
- Dopo aver salvato le varie somme parziali degli accumulatori, aggiorniamo il valore del registro R2, che conteneva l'indirizzo di partenza, aggiungendogli 16, poichè abbiamo già salvato la prima lista (quindi **incrementiamo il puntatore** per puntare alla prossima lista); e sottraiamo al valore del registro R6, contenente il numero di elementi in lista, 1 (abbiamo calcolato le somme parziali dei voti del primo studente)
- Inizializziamo l'istruzione di **salto**, con la condizione $R6 > 0$, poichè dobbiamo calcolare le somme parziali di n elementi; e saltiamo all'istruzione corrispondente all'etichetta ciclo, per poi ripartire sequenzialmente con le istruzioni;

- Una volta che la condizione diventa falsa; immagazziniamo le somme delle prove 1,2,3 in delle variabili SOMMA 1, SOMMA 2, SOMMA 3

2.7.4 Modo con base e indice

Il modo con **base e indice** è una variante significativa del modo con indice e spiazzamento e il valore di memoria che si andrà a prelevare sarà dato dall'indirizzo ottenuto dalla **somma di due registri**, anzichè sommare una valore immediato. Di conseguenza lo spiazzamento non è costante ma può variare:

`LOAD (Ri, Rj)`

2.8 Linguaggio assemblativo

Abbiamo già accennato al fatto che il **linguaggio macchina** sia costituito da stringhe di 0 e 1 che rappresentano le varie istruzioni che deve svolgere il programma; inoltre le istruzione macchina tra calcolatori diversi possono **cambiare** molto. Per questo è stato creato un **linguaggio simbolico** che permette di codificare semplici istruzione gestite in linguaggio macchina in modo comprensibile per gli umani. L'insieme di regole che costituiscono il linguaggio macchina codificato in linguaggio simbolico, interpretate dal calcolatore costituiscono il **linguaggio assembly**. Inoltre esiste un **assemblatore** o *assembler*, che è un **programma** che permette di verificare la **correttezza delle istruzioni** e di **codificare in linguaggio macchina**, ovviamente solo dopo aver verificato che le istruzioni siano corrette; ad esempio si chiede se la **sintassi** è corretta e nel caso fosse errata ci specifica che le istruzioni sono errate

2.8.1 Direttive di assemblatore

Esistono delle **direttive di assemblatore**, che non vengono codificate come istruzioni macchina, ma sono solo delle informazioni che l'assemblatore usa per codificare il codice sorgente in linguaggio macchina e possono avere la funzione di:

- Codificare **simboli** con dei **valori costanti** (*equate*)
- Decidere dove allocare i dati in memoria
- Decidere dove allocare le istruzioni (risultati o operandi)

Per tutte queste funzioni abbiamo delle "parole chiave" che vanno utilizzate prima che l'assemblatore codifichi il vero programma con le istruzioni:

- `VENTI EQU 20`

In questa direttiva **EQU** dice all'assemblatore che ogni volta che incontrerà il simbolo "VENTI" all'interno del programma, dovrà sostituirgli il valore 20

- ORIGIN 100

In questa direttiva **ORIGIN** dice all'assemblatore di allocare le istruzioni dall'**indirizzo 100**

- SOMMA : RESERVE 4

In questa direttiva **RESERVE** dice all'assemblatore di riservare uno **spazio di 4 byte** per conservare l'etichetta somma. Le **etichette** vengono associate dall'assemblatore a un **indirizzo di memoria**

- N: DATAWORD 150

In questa direttiva **DATAWORD** dice all'assemblatore di riservare una **parola di memoria** per l'etichetta N e di inserire il valore 150 nella locazione che contiene quest'etichetta

2.8.2 Assemblaggio ed esecuzione

Una volta scritto l'intero programma con le varie direttive per l'assemblatore e le istruzioni vere e proprie che compongono il programma; l'assemblatore creerà un file contenente il programma oggetto che verrà caricato da una componente del **sistema operativo**, ovvero il **loader** o caricatore, in memoria quando verrà richiesta l'esecuzione. Il loader caricherà solamente il programma e seguirà solo le **direttive utili** per se stesso; ad esempio caricherà le istruzioni del programma in memoria a partire da dove richiesto con la direttiva origin. Possiamo inoltre eseguire il programma in **modalità debug**, ovvero con un istruzione alla volta e inoltre se dobbiamo specificare che un **valore numerico** utilizzato è scritto in base **binaria** o **esadecimale** lo specificheremo rispettivamente con i **prefissi** "%" e "0x"

2.9 Gestione pila

La **pila**, o *stack*, è una struttura dati fondamentale caratterizzata da una **lista di elementi** che posso **inserire** o **prelevare** solo dall'alto. L'indirizzo che sta più in fondo alla pila viene chiamato **base** della pila; quello che sta più in alto **cima** della pila, quindi la pila cresce verso l'alto e decresce verso il basso. Inoltre la pila è una struttura di tipo **LIFO** ("last-in-first-out"): quindi l'ultimo elemento a entrare nella pila è quello che posso estrarre. Quindi noi possiamo operare solo con la cima della pila o inserendo un nuovo valore (**push**) in cima, oppure prelevando un valore (**pop**) dalla cima; queste sono le uniche 2 operazioni che possibili con la pila.

2.9.1 Operazione push e pop

Il processore è pensato anche per gestire la pila, infatti, vi è un **registro che punta alla cima** della pila, ovvero il registro **SP**, o *stack pointer*. Attraverso l'uso di questo registro possiamo anche gestire le operazioni di **push** e **pop**:

- Istruzione **push**: in questo caso dobbiamo aggiornare la cima della pila aggiungendo un nuovo valore. Quindi dobbiamo decrementare di 4 SP, per poi aggiornare il valore a cui punta SP (la nuova cima) con una STORE:

```
....  
SUBTRACT SP,SP,#4  
STORE Rj, (SP)  
....
```

- Istruzione **pop**: in questo caso dobbiamo prelevare il contenuto presente in cima alla pila, ovvero quello puntato da SP, e incrementare il puntatore di 4, aggiornandolo con una nuova cima:

```
LOAD Rj, (SP)  
ADD SP,SP,#4
```

E' importante comprendere che con le operazioni di push e pop, non riassegnamo i valori della cima, quindi **non perdiamo i valori precedenti** ma, aggiorniamo la cima della pila **decrementando l'indirizzo a cui punta SP** e aggiungendogli un nuovo dato, oppure estraendo un valore e **incrementando l'indirizzo a cui punta SP**.

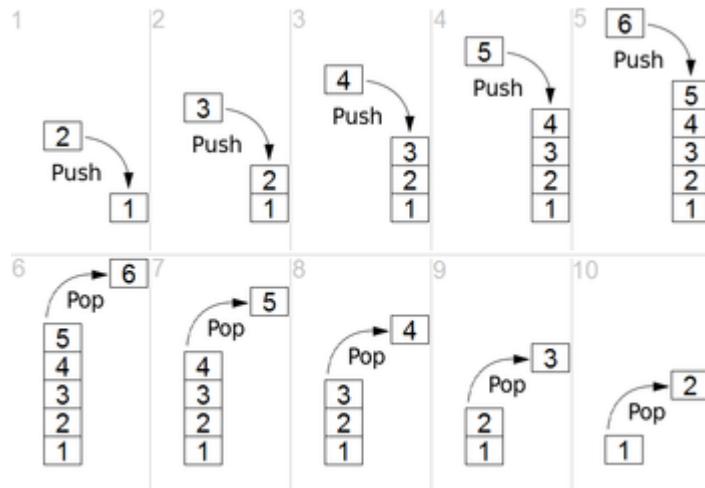


Figure 2.1: Operazioni push e pop

2.10 Gestione di sottoprogrammi

Può capitare che vogliamo eseguire istruzioni più volte o pronte all'utilizzo all'interno di un programma. In questo caso "isoliamo" queste istruzioni con una **funzione** detta **subroutine**, che può essere richiamata dal programma principale. Ciò che facciamo, quindi, è un'istruzione di salto verso la subroutine:

```
CALL SUB
```

In questo caso il codice operativo CALL indica la **chiamata della funzione** e SUB il **nome della subroutine**.

Alla fine della routine, dobbiamo inserire un'istruzione che ci permetta di tornare al programma principale esattamente all'indirizzo all'istruzione appena successiva alla call: questa istruzione viene chiamata **return instruction**. Il PC in questo caso, avrà come **punto di rientro**, il valore dato dal valore che aveva subito prima della CALL, e subito dopo aver eseguito quest'istruzione, quindi non coinciderà con la prossima istruzione del programma.

2.10.1 Metodo di collegamento

Abbiamo compreso che quando viene letta l'istruzione **call**, il PC dovrà puntare alla prima istruzione della subroutine; ma poi come dovremmo fare per tornare nel programma principale e far sì che il PC punti all'indirizzo dell'istruzione successiva alla call? Per questo esiste un **registro di collegamento**, chiamato **LR** ("link register"), che ci permette di salvare il valore di PC subito prima della chiamata in modo tale che con la **return** prendiamo l'indirizzo salvato in LR e lo copiamo in PC, che, di conseguenza, punterà all'indirizzo dell'istruzione successiva alla call. Noi in realtà non ci preoccupiamo di salvare l'indirizzo in LR ma gestisce tutto il processore.

2.10.2 Annidamento

Molto spesso può capitare di dover inserire delle **call all'interno delle subroutine**, quindi eseguire chiamate **annidate**; il procedimento di chiamata che utilizziamo è lo stesso con la call nel programma **chiamante** e il programma **chiamato** che termina con una **return**. Si pone però un problema, poiché se con il registro LR possiamo salvare l'indirizzo di PC; se facciamo diverse chiamate annidate sovrascriveremmo il valore di LR diverse volte a causa delle varie return che salvano l'indirizzo di PC dell'**ultima call**. Per evitare quindi di perdere i valori in LR utilizziamo la **pila** che è una struttura di tipo **LIFO**, quindi l'ultimo elemento salvato è quello con cui possiamo operare, in modo tale da salvare l'ultimo indirizzo di rientro in cima alla pila (SP) e man mano ritornare al programma principale con gli indirizzi corretti poiché ricordiamo che l'indirizzo di rientro è l'ultimo generato. In questo caso si dice che il meccanismo di chiamata si serve della **pila in modo implicito**. La presenza o meno del registro LR dipende quindi da ogni singolo processore; per cui per alcuni

processori, deve essere il programmatore a mettere le istruzioni per le chiamate annidate e quindi decidere se operare con la pila o meno.

2.10.3 Passaggio di parametri

La **subroutine** deve quindi poter eseguire le stesse istruzioni anche con **parametri** diversi che variano in base a dove è posta ogni chiamata alla subroutine. Esistono 2 modi per **passare dei parametri** alla subroutine:

1. Tramite **registri**: in questo caso il programma **chiamante** passerà valori a determinati registri che poi dovranno essere letti all'interno del programma **chiamato**. Questo metodo di passaggio di parametri è molto **veloce**, poichè ricordiamo che i registri sono la memoria più veloce di un elaboratore, ma i registri potrebbero essere in **numero limitato** per eseguire tutte le istruzioni della subroutine;
2. Tramite **pila**: in questo caso possiamo passare un **qualsiasi numero di parametri**. Il chiamante passerà nella pila i valori nelle giuste posizioni (push, utilizza la store) che il chiamato leggerà (pop, utilizza la load). Ovviamente però visto che il processore deve operare con la memoria, ovviamente sarà un processo molto più lento rispetto al passaggio di parametri tramite registri.

Una cosa importante da tenere in conto quando si passano i parametri nel programma chiamante e nel chiamato, è che il **passaggio di parametri** deve avvenire in **un unico modo**. Cioè se nel chiamante passiamo parametri con i registri, anche il chiamato dovrà utilizzare i registri per passare parametri; mentre se nel chiamante passiamo parametri con la pila, anche il chiamato dovrà utilizzare la pila per passare parametri.

Passaggio di parametri tramite registri Vediamo adesso un esempio di passaggio di parametri tramite registri nel quale il nostro sottoprogramma dovrà **sommare una lista di N elementi**:

- **Programma chiamante:**

```
LOAD R2,N
MOVE R4, #NUM1
CALL TOTALE
STORE R3,SOMMA
```

Nel programma chiamante:

- carichiamo la **lista N** nel registro R2;
- copiamo l'indirizzo espresso dal valore **#NUM1**, che individua l'**indirizzo iniziale** della lista, nel registro R4;
- chiamiamo il **sottoprogramma** che sarà individuato dall'etichetta "TOTALE";

- immagazziniamo la **somma totale** dell'accumulatore R3, in un indirizzo di memoria denominato con la variabile SOMMA.

In questo caso nel chiamante abbiamo passato i parametri che sono: **dimensione della lista** e **indirizzo iniziale della lista**.

- **Programma chiamato:**

```

TOTALE: SUBTRACT SP,SP,#4
        STORE R5, (SP)
        CLEAR R3
CICLO:  LOAD R5, (R4)
        ADD R3,R3,R5
        ADD R4,R4,#4
        SUBTRACT R2,R2,#1
        BRANCH_IF_[R2]>0 CICLO
        LOAD R5,(SP)
        ADD SP,SP,#4
        RETURN
    
```

Analizziamo il sottoprogramma:

- Con le prime 2 istruzioni salviamo in cima alla **pila** il valore di R5 in modo tale che alla fine del sottoprogramma possiamo ripristinare il suo valore per poterlo riutilizzare all'interno del programma chiamante;
- Azzeriamo l'**accumulatore** delle somme R3, proprio perchè dovrà accumulare le somme parziali e delimitiamo l'inizio del ciclo con l'**etichetta** "CICLO";
- Nel blocco del ciclo:
 - * **Carichiamo** il valore dell'indirizzo puntato da R4, che sarebbe l'**indirizzo di inizio della lista**, nel registro R5 (quindi carichiamo il primo valore della lista);
 - * **sommiamo** R5 ed R3 e copiamo il valore risultato in R3, che man mano conterrà la varie somme parziali;
 - * Aggiorniamo l'indirizzo di R4 alla prossima parola di memoria contenente il prossimo elemento;
 - * Aggiorniamo il contatore, ovvero il registro R2, sottraendo al suo valore 1, in modo tale che ogni operazione si sottragga 1
 - * La condizione imposta, ovviamente, fa ripetere questo salto fino a quando il contatore è maggiore di 0;
- Come accennato, una volta usciti del ciclo **ripristiniamo** il valore di R5, caricandolo dalla cima della pila (ovvero da SP), e ripristiniamo anche la cima della pila aggiungendo 4 a SP;
- Con l'istruzione **return** torniamo esattamente all'istruzione successiva alla call del programma chiamante.

Passaggio di parametri tramite pila Vediamo adesso come passare i parametri tramite pila, utilizzando come esempio di riferimento lo stesso programma utilizzato in precedenza:

- **Programma chiamante:** il nostro obiettivo nel chiamante, in questo caso, è di **creare spazio nella pila, passare i dati nella pila** immagazzinandoli con le **store**:

Assumere che la cima della pila sia a livello 1 in Figura 2.19

Move	R2, #NUM1	Impila i parametri
Subtract	SP, SP, #4	
Store	R2, (SP)	
Load	R2, N	
Subtract	SP, SP, #4	
Store	R2, (SP)	
Call	TOTALE	Chiama il sottoprogramma (cima della pila a livello 2)
Load	R2, 4(SP)	Spila il risultato e conservalo in SOMMA
Store	R2, SOMMA	
Add	SP, SP, #8	Ripristina la cima della pila (cima della pila a livello 1)
:		

Figure 2.2: Programma chiamante

Vediamo i passi di cui si compone il chiamante:

- Carichiamo l'**indirizzo della lista** in R2;
- Aggiorniamo la **cima della pila** sottraendo di 4 SP, e immagazziniamo l'indirizzo della lista in SP, ovvero in cima alla pila;
- Carichiamo la **dimensione della lista** in R2;
- Aggiorniamo la **cima della pila** sottraendo di 4 SP, e immagazziniamo la dimensione della lista (nella nuova cima della pila) in SP;
- Chiamiamo il **sottoprogramma**;
- Una volta che nel sottoprogramma ci sarà la return, carichiamo il **risultato** spilandolo dall'indirizzo puntato da SP e lo immagazziniamo in "SOMMA";
- **Ripristiniamo** la cima della pila aggiungendo 8 a SP;

- **Programma chiamato:**

TOTALE :	Subtract	SP, SP, #16	Salva in pila i registri
	Store	R2, 12(SP)	
	Store	R3, 8(SP)	
	Store	R4, 4(SP)	
	Store	R5, (SP)	(cima della pila a livello 3)
	Load	R2, 16(SP)	Inizializza il contatore a <i>n</i>
	Load	R4, 20(SP)	Inizializza il puntatore alla lista
	Clear	R3	Inizializza la somma a 0
CICLO:	Load	R5, (R4)	Preleva il prossimo numero
	Add	R3, R3, R5	Aggiungi questo numero alla somma
	Add	R4, R4, #4	Incrementa di 4 il puntatore
	Subtract	R2, R2, #1	Decrementa il contatore
	Branch_if_[R2]>0	CICLO	
	Store	R3, 20(SP)	Impila il risultato
	Load	R5, (SP)	Ripristina i registri
	Load	R4, 4(SP)	
	Load	R3, 8(SP)	
	Load	R2, 12(SP)	
	Add	SP, SP, #16	(cima della pila a livello 2)
	Return		Rientra al programma chiamante

Figure 2.3: Programma chiamato

Analizziamo il programma chiamato:

- Decidiamo di salvare uno spazio di 4 parole di memoria nella pila per **immagazzinare tutti i dati**, quindi aggiorniamo la cima della pila, ovvero l'indirizzo puntato da SP, incrementandolo di 16; in questo modo alla fine del sottoprogramma possiamo ripristinare i valori dei registri;
- **Carichiamo i valori dei registri** da R2 a R5, con R5 che coincide con la cima della pila;
- Inizializziamo il registro R2 con il **valore N**, che avevamo passato nel chiamante, e inoltre inizializziamo il registro R4 con l'**indirizzo della lista**, che anch'esso avevamo passato nel chiamante;
- Il **ciclo** che eseguiamo è pressocchè uguale a quello descritto in precedenza, infatti:
 - * **preleviamo il numero**;
 - * **Aggiungiamo questo numero alla somma parziale**;
 - * **Aggiorniamo il puntatore**;

* Decrementiamo il contatore;

- Infine **ripristiniamo** i valori nei registri, ripristiniamo la cima della pila e torniamo al programma principale;

2.10.4 Area di attivazione

Quando passiamo dal chiamante al chiamato dei parametri tramite **stack**, ed eseguiamo la chiamata a sottoprogramma, i dati verranno impilati nello stack seguendo un certo ordine che costituisce l'**area di attivazione**, o *activation record*. Gli strati di quest'area sono quindi composti da:

- Inizialmente dal chiamante impileremo la pila con i **parametri da passare**, se devono essere passati, quindi il registro SP inizialmente punterà alla cima della pila costituita dai parametri passati dal chiamante;
- Dopo aver passato i parametri viene inserito il **registro FP**, di cui spiegheremo a breve il funzionamento e lo scopo;
- Sopra il registro FP impiliamo eventuali **variabili locali** o **temporanee**;
- Infine impileremo i registri da salvare e che a fine routine dovranno essere ripristinati;

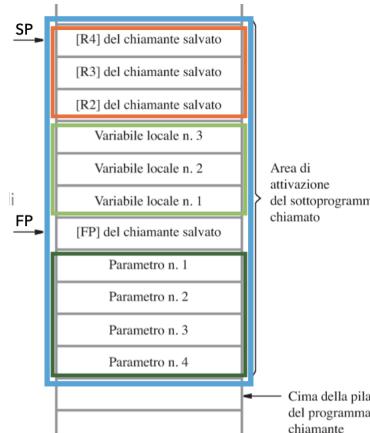


Figure 2.4: Area di attivazione

Registro FP Il **registro SP**, come vediamo dalla figura 2.4, punterà sempre alla cima della pila, nello specifico alla routine che si sta eseguendo, mentre, per prelevare i parametri, la routine dovrebbe contare quante parole di memoria si trovano al di sotto di SP. Per questa funzione si utilizza il **registro FP**: esso è il **puntatore all'area di attivazione** e punta nel punto dello stack subito dopo della chiamata a sottoprogramma dal chiamante così da avere un registro in

una posizione "strategica" che ci permette attraverso un incremento positivo di puntare ai parametri e con incremento negativo di puntare alle eventuali variabili locali. Il punto nel quale viene salvato FP, in uso da parte del chiamante, è dato dalla parola di memoria che punta al **vecchio valore di FP** del chiamante. Quindi succede che la routine impila FP, dopo che il chiamante ha passato i parametri, aggiornando la cima della pila con SP e poi impila le eventuali variabili locali e registri. Quando bisogna **ripristinare** i valori di **registri** e del registro FP sarà compito della routine ripristinarli; mentre i **parametri** verranno ripristinati dal chiamante.

Aree di attivazione con annidamento Si abbia un programma principale *PRO*, che chiama la routine *SUB1*, che a sua volta chiama la routine *SUB2* e passiamo i parametri tramite pila:

- **PRO:** Impiliamo i parametri da passare a SUB1
- **SUB1:**
 - Viene salvato il registro LR, che serve come **punto di ritorno**;
 - **Impiliamo e conserviamo FP**, in uso da parte del chiamante, la cui parola di memoria punterà al valore di FP di PRO, e aggiorniamo SP con la nuova cima della pila;
 - **Salviamo i registri**, che a fine programma dovranno essere ripristinati dalla routine;
 - **Passiamo un parametro** a SUB2;
- **SUB2:**
 - **Salviamo un nuovo valore di FP**, la cui parola di memoria punta al valore di FP in SUB1;
 - **Salviamo i registri** che saranno poi da ripristinare da parte della routine;

In conclusione abbiamo capito che le aree di attivazione dei sottoprogrammi si succedono in modo sequenziale l'una sopra l'altra e inoltre i vari FP sono **concatenati**, cioè puntano sempre a quello del chiamante⁶. Possiamo vedere il procedimento appena descritto in questo programma che fa uso di due sottoprogrammi:

⁶Il registro FP ricordiamo che è un **valore costante** e quindi a differenza di SP non deve essere cambiato. Solo SP man mano aggiorna la cima della pila

Programma principale				Primo sottoprogramma			
		:		2100	SUB1:	Subtract	SP, SP, #24
2000	PRO:	Load R2, PARAM2	Impila i parametri	2104		Store	LINK,_reg.20(SP)
2004		Subtract SP, SP, #4		2108		Store	FP, 16(SP)
2008		Store R2, (SP)		2112		Store	R2, 12(SP)
2012		Load R2, PARAM1		2116		Store	R3, 8(SP)
2016		Subtract SP, SP, #4		2120		Store	R4, 4(SP)
2020		Store R2, (SP)		2124		Store	R5, (SP)
2024		Call SUB1	Chiama il sottoprogramma	2128		Add	FP, SP, #16
2028		Load R2, (SP)	Immagazzina il risultato	2132		Load	R2, 8(SP)
2032		Store R2, RIS		2136		Load	R3, 12(SP)
2036		Add SP, SP, #8	Ripristina il livello della pila			Load	R4, (SP)
2040		prossima istruzione				Add	SP, SP, #4
Secondo sottoprogramma				2100		Load	R4, PARAM3
3000	SUB2:	Subtract SP, SP, #12	Salva in pila i registri	2104		Subtract	SP, SP, #4
3004		Store FP, 8(SP)		2108		Store	R5, (SP)
		Store R2, 4(SP)		2112		Load	R4, 4(SP)
		Store R3, (SP)		2116		Load	R3, 8(SP)
		Add FP, SP, #8	Inizializza il puntatore all'area di attivazione	2120		Load	R2, 12(SP)
		Load R2, 4(FP)	Preleva il parametro	2124		Load	FP, 16(SP)
		:		2128		Load	LINK,_reg.20(SP)
		Store R3, 4(FP)	Impila il risultato di SUB2	2132		Add	SP, SP, #24
		Load R3, (SP)	Ripristina i registri	2136		Return	Rientro al programma principale
		Load R2, 4(SP)					
		Load FP, 8(SP)					
		Add SP, SP, #12					
		Return	Rientro al sottoprogramma				

46

Figure 2.5: Aree di attivazione con annidamento

Quindi come possiamo vedere nel primo sottoprogramma salviamo uno spazio di 6 parole di memoria nella pila aggiornando SP, ovvero sottraendo 16; salviamo il Link Register in ultima posizione, in seguito il registro FP e poi salviamo i registri R2, R5 nelle varie parole di memoria in modo sequenziale.

2.11 Ulteriori istruzioni macchina

Molto spesso può essere utile fare uso di istruzioni che facciano operazioni logiche come **AND**, **OR**, **NOT**.

And R4, R2, R3

In questo caso il **codice operativo** è And, i **registri sorgente** sono R2 e R3 e il **registro di destinazione** è R4. In questo caso avremo un **prodotto bit a bit**, cioè facciamo il prodotto di ogni singolo bit di entrambi i registri delle rispettive posizioni. Un uso molto utile potrebbe essere, ad esempio, quello di voler confrontare un carattere azzerando i 3 byte più significativi a sinistra:

And R2, R2, #0xFF

In questo caso ciò che facciamo è fare l'and tra un **valore immediato esadecimale** (0x), che in questo caso pone a 1 gli ultimi 8 bit a destra e **estende a 0** automaticamente tutti i bit vuoti (24) a sinistra, e il valore presente in R2. In questo caso avremo che i 3 byte a sinistra l'and per qualsiasi valore di R2

sarà 0. Abbiamo compreso quindi che esiste una componente del processore che pone automaticamente a 0 tutti i bit non specificati nella parola di memoria. Questa istruzione potrebbe essere utile per magari effettuare un operazione di confronto tra 2 caratteri che occupano solo un byte come nel precedente caso del valore `#0xFF`

2.11.1 Scorrimento

Un'altra istruzione molto utile è quella di **scorrimento** o *shift*, che appunto "scorre" un di bit di un certo numero di posizioni o **verso destra** o **verso sinistra**. Vediamo come si codificano queste istruzioni in assembly:

- `LshiftL Ri, Rj, #contatore`

In questo caso il codice operativo "LshiftL" (*logic shift left*), indica uno **scorrimento verso sinistra** di R_j di contatore posizioni, il risultato verrà copiato in R_i . In questo caso succede che i bit **più significativi** a sinistra vengono eliminati e le posizioni vuote a destra vengono riempite con 0. In realtà, prima di essere eliminati, i bit significativi nel caso dello scorrimento verso sinistra, vengono posti uno a uno nel **bit di riporto** o *carry*, che quindi viene modificato in base allo scorrimento.

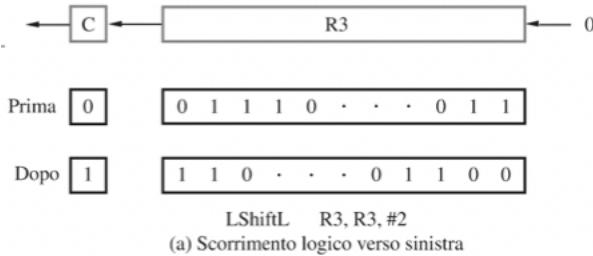


Figure 2.6: Scorrimento verso sinistra

- Lo **scorrimento verso destra** è molto simile:

`LshiftR Ri, Rj, contatore`

In questo caso il codice operativo è "LshiftR", perdiamo i bit **meno significativi** a destra e la posizioni vuote a sinistra verranno riempite con 0. Anche in questo caso il bit di riporto verrà aggiornato in base allo scorrimento.

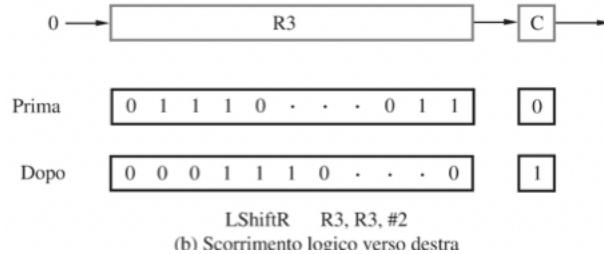


Figure 2.7: Scorrimento verso destra

- Se i bit rappresentano **numeri relativi** con rappresentazione in complemento a due, quando si scorre il **bit più significativo** deve essere replicato, a questo scopo si usa *AShiftR* (**scorrimento aritmetico**):

Utilizzi dello scorrimento Facciamo adesso una breve considerazione sull'utilizzo dello scorrimento:

- **Scorrimento verso destra:** se scorriamo a destra di una sola posizione, ciò che otteniamo è il numero precedente **dimezzato**, infatti scorrere verso destra di una posizione significa "andare" a posizionare i bit in una potenza di 2 in meno ($2^2 = \text{bit } 1 \rightarrow 2^1 = \text{bit } 1$, ovvero la metà). Possiamo verificare il **resto** inoltre con il **carry** in modo tale che se esso è 0, allora il numero sarà **pari**, se è 1, allora sarà **dispari**.
- **Scorrimento verso sinistra:** al contrario dello scorrimento verso destra, in questo caso se scorriamo di una sola posizione verso sinistra ciò che otteniamo è il numero **raddoppiato** poichè stiamo posizionando i bit in modo tale che si trovi con una potenza di 2 in più ($2^2 = \text{bit } 1 \rightarrow 2^3 = \text{bit } 1$, ovvero il doppio).
- **Impaccamento:** lo scorrimento è utile anche per l'impaccamento. Molto spesso un dato è contenuto in **pochi bit** di conseguenza ci conviene ad esempio utilizzare un unico byte per memorizzare tale dato. Ciò che facciamo è uno **scorrimento verso sinistra** di 4 bit e in seguito un **OR** con il dato che contiene a destra i bit meno significativi.

2.11.2 Rotazione

Vi è un istruzione che ci permette di scorrere i bit in modo tale da non eliminarne nessuno ma effettuare una **rotazione** che faccia scorrere i bit da un estremo e li faccia rientrare nell'altro estremo in base a se la rotazione avviene verso **destra** o **sinistra**. In questo caso la rotazione copierà il **bit uscente sul bit di riporto** prima di farlo rientrare nell'altro estremo. Vediamo alcuni esempi:

- **Rotazione verso sinistra:**

```
RotateL R3, R3, #2
```

In questo caso la rotazione scorrerà 2 bit da sinistra, li copierà sul bit di riporto, e li copierà come ultimi due bit a destra;

- **Rotazione verso sinistra:**

```
RotateR R3, R3, #2
```

In questo caso la rotazione scorrerà 2 bit da destra, li copierà sul bit di riporto, e li copierà come primi due bit a sinistra;

- Oppure, la rotazione può coinvolgere il bit di riporto che si aggiunge come **posizione ulteriore di rotazione**:

```
RotateLC R3, R3, #2
```

2.11.3 Moltiplicazione e Divisione

Abbiamo anche delle apposite istruzioni in RISC per effettuare **moltiplicazione** e **divisione**:

- **Moltiplicazione:**

```
Multiply Rk,Ri,Rj
```

In questo caso la **sintassi** è molto semplice in quanto già in parte l'abbiamo vista: il primo registro a sinistra è la **destinazione** del risultato del **prodotto**; mentre gli altri 2 registri a destra sono la **sorgente**, ovvero i fattori da moltiplicare. Bisogna fare molta attenzione, però, al fatto che molto spesso il prodotto non può essere inserito nello **spazio di n bit dei fattori**. Ad esempio: $15 = 1111$ (4 bit); il prodotto $15 \cdot 15 = 1111 \cdot 1111 = 1110001$ occuperà uno spazio di 8 bit⁷. Per ovviare a tale problema, riscontrabile anche ovviamente con un numero più elevato di bit che faccia sì che non basti lo spazio di una parola di memoria per contenerlo, il compilatore **suddivide il risultato** in tal modo:

- La parte di bit del risultato **meno significativa**, ovvero i bit più a destra, verranno trasferiti nel registro di destinazione generico "Rk";

⁷Ovviamente in questo caso, uno spazio di 8 bit non si configura come un problema dato che lo spazio di una parola di memoria è di 32 bit; ma se dovessimo operare con fattori più grandi ci sarebbe questo problema.

- La parte di bit del risultato **più significativa**, ovvero i bit più a sinistra, verranno trasferiti nel **registro successivo** a quello di destinazione "R_k + 1"

- **Divisione:**

Divide R_k,R_i,R_j

Come per il prodotto, nella **divisione** il primo registro a sinistra è la **destinazione** del risultato della **divisione**; mentre gli altri 2 registri a destra sono la **sorgente**. In tal caso, dobbiamo fare attenzione all'**ordine dei registri sorgente**, poichè il primo a sinistra (R_j) rappresenta il **numeratore**, mentre quello a destra rappresenta il **denominatore**. Inoltre nella divisione non dobbiamo preoccuparci, del numero di bit che ci occorrono per allocare il risultato; ma abbiamo un **resto**, che può essere salvato nel **registro successivo a quello di destinazione** (R_k + 1)

2.12 Valori immediati a 32 bit

In architettura RISC sappiamo che ogni **registro** è grande 1 parola di memoria, ovvero **32 bit**, tanto quanto la dimensione di un istruzione. Quando, però, passiamo **valori immediati** attraverso i modi di indirizzamento immediato e assoluto, per costruzione essi **occuperanno solo 16 bit**, poichè si vuole lasciare spazio per l'intera istruzione. Infatti, esiste un meccanismo del processore che **estenderà a 0** i 16 bit più significativi riempiendo solo quelli **meno significativi**. Il problema si pone quando abbiamo bisogno di codificare **valori immediati più grandi di 16 bit**, come ad esempio un esadecimale del tipo #0x20004FF0, per il quale ogni cifra occupa 4 bit, per un totale di 32 bit. In tal caso, utilizziamo le seguenti istruzioni:

```
OrHigh R2,R0,#0x2000
Or R2,R2,#0x4FF0
```

Analizziamo queste istruzioni:

- L'**istruzione OrHigh** è un operazione che si serve dell'OR per i bit più significativi. Di conseguenza ci è servita per porre a 0 i bit meno significativi, e allocare le **prime 4 cifre** del valore immediato, che **occupano 16 bit** dal passaggio da esadecimale a binario, nei **bit più significativi**. Infatti, per costruzione, il registro "R₀" contiene al suo interno tutti 0; per cui un OR logico tra 0 e 0, restituirà 0 nei bit meno significativi, e l'OR logico tra 0 e il valore immediato lascerà invariato il valore **allocandolo nei bit più significativi** del registro R₂;

- L'**istruzione Or** utilizza, a differenza del "OrHigh" **solo i bit meno significativi**: infatti, avendo già allocato le prime 4 cifre del valore, e, avendo occupato di conseguenza i primi 16 bit più significativi, adesso ci servirà fare l'OR logico tra il valore già allocato e la parte del valore ancora non allocato, in modo tale da allocare #0x4FF0 nei bit meno significativi, occupando gli ultimi 16 bit.
- In conclusione abbiamo visto quindi che per allocare un **valore immediato più grande di 16 bit** con architettura RISC, abbiamo bisogno di più istruzioni per allocarlo in un solo registro:

2.13 Insieme di istruzioni CISC

Abbiamo visto in questo momento istruzioni per un **architettura RISC**, che presupponeva che ogni istruzione veniva codificata da una parola di memoria e si basava su un'architettura **LOAD/STORE** che prevedeva che potevamo lavorare con la memoria solamente tramite caricamento e immagazzinamento. In un **architettura CISC** non abbiamo né il vincolo di poter utilizzare una parola di memoria, infatti possiamo utilizzare **diverse parole di memoria per singola istruzione**, e inoltre, non siamo vincolati da un'architettura LOAD/STORE, ovvero possiamo effettuare **operazioni aritmetico-logiche su operandi che si trovano nei registri**. Vediamo adesso un'insieme di **istruzioni di tipo CISC** e la loro **semantica**:

- Operazione di **addizione**:

```
ADD B,A
```

Analizzando, quindi, il formato dell'istruzione notiamo già delle differenze diverso a un istruzione di tipo RISC. Innanzitutto abbiamo solo **2 operandi e un codice operativo**, e, l'operando "B" rappresenta **sia la sorgente che la destinazione della somma**. Inoltre le due variabili, che costituiscono gli operandi della somma, **rappresentano locazioni di memoria**; quindi, in conclusione questa istruzione fa la somma tra il contenuto di "A" e "B" e lo alloca in B;

- Istruzione **move**:

```
MOVE C,B  
ADD C,A
```

L'istruzione move, come per l'addizione, si serve solo di 2 operandi, dove il primo rappresenta la **destinazione** e il secondo la **sorgente** e entrambe sono **locazioni di memoria**, e sostituisce la LOAD/STORE utilizzata

in RISC per lavorare con la memoria. Nell'esempio qui sopra stiamo codificando l'istruzione ad alto livello $C = A + B$; in realtà, a seconda del processore CISC a disposizione, gli operandi da utilizzare possono essere o solo variabili che indicano locazioni di memoria, oppure **un registro e una locazione di memoria**:

```
Move Ri, A
Add Ri, B
Move C, Ri
```

Come possiamo notare, quindi, le istruzioni di un processore CISC, sono molto **differenti a livello di semantica** da un RISC: infatti, nell'esempio precedente, stiamo sommando qualcosa presente in un registro con qualcosa presente in memoria; cosa che in un architettura RISC non possiamo fare.

2.13.1 Istruzioni di autoincremento e autodecremento

Quando utilizzavamo la **pila** in un processore con architettura RISC, riservavamo uno spazio di byte congruo a salvare tutti i valori che ci servivano e poi li inserivamo nella cima della pila con la **push** o li estraevamo con la **pop**. In CISC abbiamo delle istruzioni che ci permettono di impilare o spilare in modo leggermente differente:

- Istruzione di **autodecremento** (push):

```
MOVE -(SP), ELEMENTO
```

In questo caso stiamo dicendo all'assemblatore di decrementare l'indirizzo puntato da SP di un certo numero di byte, in base alla dimensione del dato da inserire, e inserire di conseguenza in cima alla pila (push) il valore "ELEMENTO"

- Istruzione di **autoincremento**:

```
MOVE ELEMENTO, (SP)+
```

In questo caso stiamo dicendo all'assemblatore di estrarre elemento dalla cima della pila (pop), e di incrementare l'indirizzo puntato da SP di un certo numero di byte, in base alla dimensione del dato che viene estratto.

2.13.2 Indirizzamento con modo relativo a PC

Nei processori con architettura RISC non potevamo indicare il registro PC, anche sapendo che esisteva con un ben determinato scopo. Nei processori CISC possiamo effettuare un **indirizzamento con indice e spiazzamento** utilizzando il registro PC come indice:

X(PC)

Quindi in tal caso indicheremo una **parola di memoria traslata di X byte rispetto a PC**. Non è molto consigliato utilizzare questo tipo di indirizzamento poichè potrebbe danneggiare seriamente il programma creato.

2.13.3 Bit di esito o condizione

Quando eseguiamo operazioni logico-aritmetiche, il processore si basa e tiene conto dei **bit di esito** che sono:

- **N** (negativo): vale 1 se l'operazione ha restituito un numero negativo, viceversa, vale 0 se l'operazione ha restituito un valore positivo;
- **Z** (zero): vale 1 se il risultato dell'operazione ha restituito zero, altrimenti per il resto dei casi vale 0;
- **V** (trabocco): vale 1 se vi è stato trabocco nell'operazione, mentre vale 0 se non vi è stato trabocco;
- **C** (riporto): vale 1 se vi è stato riporto, mentre vale 0 se non vi è stato riporto.

Per comprendere la **funzione dei bit di esito** prendiamo come esempio un istruzione di **branch** :

```
SUBTRACT R2, #0
BRANCH > 0  CICLO
```

In questo caso la branch si baserà sui **bit di esito per valutare la condizione**: infatti la branch dice di saltare all'etichetta "CICLO", solo quando la condizione è maggiore di 0; di conseguenza i bit di esito **N,Z**, dovranno essere entrambi diversi da zero, ovvero dovranno essere uguali a 1. Quando Z risulterà 1, ovvero l'operazione ha restituito 0, la condizione non sarà più soddisfatta e non salteremo più a ciclo.

2.14 Stili RISC e CISC

Quindi abbiamo capito che le architetture RISC e CISC agiscono in modo diametralmente opposto:

- **RISC:**

- Abbiamo **molte istruzioni ma semplici**;
- Ogni **istruzione occupa una parola di memoria**;
- **Esecuzione rapida**;
- **Hardware semplice**;
- Architettura **LOAD/STORE**;
- possiamo effettuare **operazioni aritmetico-logiche** solamente tramite **registri e valori immediati**;

• **CISC:**

- Abbiamo **poche istruzioni ma complesse**;
- Ogni **istruzione può occupare più di una parola di memoria**;
- **Esecuzione lenta**;
- **Hardware complesso**;
- possiamo effettuare **operazioni aritmetico-logiche** possono usare **operandi in memoria e registri del processore**;

Ad oggi i **processori più utilizzati** hanno un architettura **RISC**, proprio perchè i vantaggi di questo tipo di architettura superano di molto quelli di un'architettura CISC. Fino a pochi anni fa, si tendeva a utilizzare processori CISC poichè permettevano di scrivere poche istruzioni, occupando di conseguenza poca memoria, ma molto più complesse; ad oggi non abbiamo problemi di memoria, di conseguenza, si tende a utilizzare processori RISC per scrivere istruzioni molto più semplici e di rapida esecuzione.

2.15 Codifica di istruzioni

Per comprendere la **codifica di istruzioni** (RISC) in binario, dobbiamo comprendere effettivamente in ogni istruzione quanto **spazio** occupano registri, codici operativi e valori immediati. In questo senso le istruzioni si suddividono in 3 categorie:

1. **Istruzioni con 3 registri e un codice operativo:** queste istruzioni si identificano con la **add, subtract...** In tal caso occorre fare tale ragionamento:
 - Se ho in memoria uno spazio di **32 registri**, ogni registro occuperà **5 bit** ($2^5 = 32$);
 - In un'istruzione come la ADD, che presenta 3 registri, quindi io dovrò avere uno **spazio di 15 bit per i registri**, e lo spazio rimanente sarà occupato dal **codice operativo**: $32 - 15 = 17$, di conseguenza in questa categoria di istruzioni il codice operativo occuperà **17 bit**;

- Inoltre, c'è un'altra importante considerazione da fare per quanto riguarda i **registri**: questi, infatti, non solo occupano uno spazio di 5 bit, ma la **posizione di questi registri è fissata dal processore**.
2. **Istruzioni con 2 registri, un valore immediato e un codice operativo:** queste istruzioni, tra le quali possono rientrare le add con valore immediato oppure le **load/store** che utilizzano indice e spiazzamento, vengono codificate nel seguente modo:
- 2 **registri** occuperanno uno spazio di **10 bit**, infatti $5 \cdot 2 = 10$;
 - Il **valore immediato** sappiamo che per i processori RISC, occupa **16 bit**;
 - Il **codice operativo** in tal caso occuperà i restanti bit, ovvero $32 - (10 + 16) = 6$, cioè occuperà **6 bit**;
 - Anche in questo caso la posizione dei registri è fissata dal processore;
- Anche le **istruzioni di salto** utilizzano tale codifica e, il valore immediato inserito nell'istruzione, è lo **spiazzamento tra l'etichetta e l'indirizzo del salto**. In tal caso anche lo spiazzamento, essendo un valore immediato, sarà codificato in **16 bit**.
3. **Istruzioni di chiamata a sottoprogramma:** le istruzioni di chiamata a sottoprogramma sono molto semplici poiché contengono:
- Un **codice operativo**, codificato in **6 bit**;
 - Un **valore immediato**, codificato nei restanti bit, ovvero **26 bit**;

Capitolo 3

OPERAZIONI DI INGRESSO/USCITA

3.1 Accesso a dispositivi di I/O

Abbiamo già visto nel capitolo 1, come tutte le componenti del calcolatore siano collegate tramite *fili di bus* a una **rete di interconnessione**, quindi tra i dispositivi collegati a questa rete vi sono anche i **dispositivi di I/O**, dette *periferiche*. Vediamo adesso come il **processore accede alle periferiche**, ricordandoci come accedeva alla memoria: quando appunto dovevamo effettuare un'operazione di **load/store** per accedere in memoria dal processore, cioè leggere o scrivere qualcosa in memoria, inviavamo attraverso queste istruzioni un **indirizzo**, con i vari modi di indirizzamento, che, passava dalla rete di interconnessione tramite fili di bus, e una volta arrivato in memoria faceva attivare la locazione di memoria corrispondente a quell'indirizzo, tornando indietro il dato al processore, che poi lo salvava. Quando **accediamo ai dispositivi I/O** tramite il processore, avviene un processo simile. In particolare, abbiamo due modi per accedere alle periferiche:

1. **Unificazione degli indirizzi di memoria:** in questo caso vi sono degli **indirizzi speciali** che vengono salvati nello **spazio di indirizzamento del processore** e che vengono **associati a una determinata periferica**. In tal caso potremmo comunicare con le periferiche in maniera analoga a come facevamo con la memoria con load/store.
2. Un altro modo per accedere tramite il processore alle periferiche, è quello di **non utilizzare lo spazio di indirizzamento del processore**, ma **creare istruzioni apposite** per indirizzare le periferiche. Le modalità di accesso dipendono dalla costruzione del calcolatore, ma, ad oggi, il metodo di unificazione degli indirizzi è il più utilizzato.

3.1.1 Interfaccia dei dispositivi

Facendo riferimento all'*unificazione degli indirizzi di memoria*, abbiamo quindi che lo **spazio di indirizzamento** di n bit non è più 2^n , poiché vi sono indirizzi riservati alle periferiche. Inoltre, un dispositivo di I/O è collegato alla rete di interconnessione tramite un circuito che sta all'*interno della periferica* chiamato **interfaccia del dispositivo**, che contiene al suo interno **registri**, che non sono quelli del processore, ma sono **locazioni di memoria** che possono contenere:

- **Dati;**
- **Stato;**
- **Controllo;**

Quindi quando vogliamo accedere alla periferica, invieremo col processore un particolare indirizzo tramite la **load** che ci permetterà di **leggere un registro dell'interfaccia**, quindi, ritornando al discorso fatto in precedenza, occupiamo degli **indirizzi per accedere ai registri delle periferiche**.

3.1.2 I/O controllati da programma

Vediamo adesso, come da titolo, come effettivamente avviene il **processo di input/output di un determinato dato**, utilizzando come riferimenti la *tastiera* e il *video* (display). Dobbiamo, però, fare un importante premessa: le periferiche sono **molto più lente** del processore per quanto riguarda il **fornire dati** se pensiamo che con la tastiera possiamo fornire un carattere ogni 0,3 secondi, mentre colo processore ogni nanosecondo. Bisogna quindi partire dall'idea di trovare una **sincronizzazione temporale tra il processore e la periferica** in questione. Vediamo degli esempi di ciò che abbiamo detto, utilizzando come esempio di periferica di input, la tastiera, e come periferica di output, il video(display):

- **Tastiera:**
 - Ogni volta che premiamo un tasto con la tastiera, esso è **collegato ad un circuito di controllo**, presente all'interno alla tastiera, in modo tale che **ogni carattere corrisponda a una codice binario**, utilizzando il sistema *ASCII* come riferimento.
 - Il codice binario corrispondente al tasto premuto, viene **salvato nel registro dati dell'interfaccia della tastiera**, che si comporta come *buffer*, ovvero contenitore temporaneo di dati;
 - Viene impostato dalla tastiera un **bit di stato** per dire al processore che ha ricevuto un dato;
- **Video:** Avviene esattamente lo stesso processo che avveniva con la tastiera; in questo caso quando un dato è stato stampato a video, viene impostato il **bit di stato** che dice al processore che può ricevere un altro dato.

3.1.3 Lettura dati dalla tastiera

I registri, di cui abbiamo parlato fino ad adesso, che sono **contenuti nell'interfaccia** della tastiera e che **corrispondono ognuno a un indirizzo**, sono questi:

- **Registro dati:**

KBD_DATA

dove "KBD" è la dicitura "keyboard", ovvero tastiera;

- **Registro di stato:**

KBD_STATUS

- **Registro di controllo:**

KBD_CONT

Inoltre sappiamo che:

- *KBD_DATA* è registro occupa una **spazio di 8 bit**, corrispondente a un carattere in codifica ASCII, cioè al dato che dovrà contenere;
- *KBD_STATUS* è un registro che presenta un'**etichetta**, chiamata **KIN**, che pone un determinato **bit a 1**, se **esiste un nuovo dato da leggere** da *KBD_DATA*, mentre pone questo **bit a 0**, se **ha letto un dato** da *KBD_DATA*;

3.1.4 Scrittura caratteri sul video

Anche in questo caso abbiamo i 3 registri che sono **contenuti nell'interfaccia** del video e che **corrispondono ognuno a un indirizzo**, e sono:

- **Registro dati:**

DISP_DATA

dove "DISP" è la dicitura "display", ovvero video;

- **Registro di stato:**

DISP_STATUS

- **Registro di controllo:**

DISP_CONT

Sappiamo inoltre che:

- *DISP_DATA* è registro occupa una **spazio di 8 bit**, corrispondente a un carattere in codifica ASCII, cioè al dato che dovrà essere inviato dal processore;
- *DISP_STATUS* è un registro che presenta un'**etichetta**, chiamata **DOUT**, che pone un determinato **bit a 1**, se **pronto a ricevere un altro carattere** da *DISP_DATA*, mentre pone questo **bit a 0**, se **ha scritto un dato** da *DISP_DATA*;

3.1.5 Programma di lettura e scrittura di una linea di caratteri

Vediamo adesso come effettivamente codifichiamo un programma che utilizzi i registri dell'interfaccia di una periferica di input (tastiera) e di una periferica di output (video):

Move	R2, #LOC	Inizializza il registro puntatore R2 per puntare all'indirizzo della prima locazione nella memoria principale dove immagazzinare i caratteri
MoveByte	R3, #CR	Carica in R3 il codice ASCII per il Ritorno Carrello
LEGGI: LoadByte And Branch_if_[R4]=0	R4, KBD_STATUS R4, R4, #2 LEGGI	Attendi l'immissione di un carattere Controlla la condizione di stato KIN Leggi il carattere da KBD_DATA (ciò azzerà KIN)
LoadByte	R5, KBD_DATA	Leggi il carattere da KBD_DATA (ciò azzerà KIN)
StoreByte	R5, (R2)	Scrivi il carattere nella memoria principale e incrementa il puntatore alla memoria principale
Add	R2, R2, #1	
ECO: LoadByte And Branch_if_[R4]=0	R4, DISP_STATUS R4, R4, #4 ECO	Attendi che lo schermo sia pronto Controlla la condizione di stato DOUT Trasferisci il carattere appena letto al registro buffer dello schermo (ciò azzerà DOUT)
StoreByte	R5, DISP_DATA	
Branch_if_[R5]≠[R3]	LEGGI	Controlla se il carattere appena letto sia il Ritorno carrello. Se non lo è, reitera la lettura di caratteri

Lettura di un carattere dalla tastiera

LoadByte indica che l'operando è un byte
And controlla il bit di stato

Visualizzazione di un carattere sullo schermo

8

Figure 3.1: Lettura e scrittura in RISC

Analizzando il programma, possiamo fare alcune considerazioni:

- Nel blocco delle prime due istruzioni **inizializziamo il registro puntatore all'indirizzo nel quale inserire i caratteri** e con una **Move-byte**¹, carichiamo in R3 il **ritorno carrello**, che corrisponde al tasto invio, codificato in ASCII, della tastiera;

¹Le istruzioni **load/store/move** possono essere utilizzate per accedere anche a singoli byte, anzichè a intere parole di memoria, come in questo caso per caricare un carattere, con la dicitura opcode **byte**

- Iniziamo il blocco di lettura, con l'**etichetta** "LEGGI", che contiene l'istruzione di **caricamento del registro di stato** ("KBD_STATUS") **in un registro del processore** per capire se c'è o meno un **dato da leggere in kin**, attraverso un and ;
- Facciamo un **and bit a bit**, tra il registro R4 e il valore immediato 2, che codificato in binario, ci permetterà di capire se l'**and restituisce 1** che kin, che **si trova in posizione 1** (2^1), è 1; mentre se l'**and restituisce 0**, kin si trova a 0;
- Quindi eseguiamo il salto con la branch, che ci serve per il discorso che il processore è molto più veloce della periferica. Infatti fino a che $R4 = 0$, deve eseguire il salto fino a che kin si trova a 0, ovvero **non ha nessun dato da leggere**. Capiamo quindi che questa metodologia di programma per la lettura fa sprecare molto tempo al processore poiché deve fare questo **ciclo di scansione** fino a quando il **bit di stato diventa 1**;
- Quando usciamo dal ciclo, scriviamo questo valore inserito nell'**indirizzo puntato da R2** e incrementiamo il puntatore di 1;
- Lo stesso procedimento vale per la periferica di output, solo che in questo caso, utilizzando il registro "DISP_STATUS", utilizzeremo il DOUT per vedere che valore restituisce con l'and. Il DOUT però, a differenza del kin, si trova in **posizione 2**, ovvero 2^2 , di conseguenza dovremo fare l'and con il **valore immediato 4**;

3.1.6 Esempio in stile CISC

In stile CISC è possibile effettuare alcune **operazioni aritmetiche e logiche direttamente su operandi in memoria**.

```
TestBit destinazione, #k
```

Controlla il bit b_k dell'operando destinazione e pone Z, *bit di stato zero*, a 1 se $b_k = 0$, altrimenti lo pone a 0.

```
Compare destinazione, sorgente
```

Effettua il controllo sottraendo e **aggiorna i bit di esito** in base al risultato, senza modificare il contenuto né di destinazione, né di sorgente:

	Move	R2, #BLOCCO	Inizializza il registro R2 per puntare all'indirizzo della prima locazione nella memoria principale dove immagazzinare i caratteri
LEGGI	TestBit Branch=0	KBD_STATUS, #1 LEGGI	Monitorando la condizione di stato KIN, attendi l'immissione di un carattere nel registro di I/O KBD_DATA
	MoveByte	(R2), KBD_DATA	Scrivi nel byte di memoria puntato da R2 il carattere contenuto nel registro di I/O KBD_DATA (ciò azzerà KIN)
ECO	TestBit Branch=0	DISP_STATUS, #2 ECO	Attendi che lo schermo sia pronto monitorandone la condizione di stato DOUT
	MoveByte	DISP_DATA, (R2)	Scrivi il carattere puntato da R2 nel registro di I/O DISP_DATA (ciò azzerà DOUT)
	CompareByte Branch≠0	(R2)+, #CR LEGGI	Verifica se il carattere appena letto da tastiera sia il Ritorno Carrello; se non lo è, reitera la lettura di caratteri; in ogni caso incrementa il registro puntatore R2

Figure 3.2: Programma di lettura/scrittura in CISC

Analizzando questo programma, possiamo fare le seguenti considerazioni, considerando le premesse iniziali:

- Nel **ciclo di scansione** monitoriamo il bit di kin, in posizione 1, con test bit, che agirà sul **bit di stato z**, ponendolo a 0, se *kin* = 1, e, viceversa, ponendolo a 1, se *kin* = 0. Il ciclo verrà fatto ogniqualvolta si verificherà che *kin* = 0, ovvero non ha nessun dato;
- Infine con la movebyte, che ricordiamo in CISC serve per accedere alla memoria, **immagazziniamo il dato contenuto nel registro dati "KBD DATA"** all'**indirizzo puntato da R2**.
- Stesso procedimento vale per la periferica output, e, come avevamo visto in precedenza, in tal caso il testbit agirà nel **bit in posizione 2**, che rappresenta "DOUT".

3.2 Interruzioni

Le **interruzioni** sono un meccanismo fondamentali dei calcolatori che permettono la comunicazione tra periferiche e processore. Abbiamo compreso, infatti, che i **cicli di attesa** farebbero sprecare troppo tempo al processore. Per questo esistono degli **ingressi del processore** appositi etichettati con le linee di controllo "**INT_REQ**" (*interrupt request*), che è un **segnale di controllo** che permette di dialogare tra la periferica in questione e il processore, e inoltre questa linea di segnale viene **pilotata dalla periferica**. Quando l'interrupt request viene pilotata dalla periferica a 1, cioè si dice che **il segnale viene posto in alto**, il processore **interrompe** l'esecuzione del **programma** che stava eseguendo e **salta all'inizio della routine di servizio dell'interruzione**, e, una volta finita la routine di servizio, torna all'esecuzione del programma principale. Grazie a tale meccanismo di "interruzione" il processore si dedica al programma, e, solo quando riceve l'interrupt request si dedica a ciò che chiede la periferica. In tal modo **non viene sprecato il tempo del processore**

3.2.1 Esempio di utilizzo delle interruzioni

Un programma deve fare dei calcoli e visualizzare i risultati ogni 10 secondi. Una soluzione è avere un cronometro che segnala una richiesta di interruzione ogni 10 secondi. Il programma è costituito da due routine: "CALCOLA" e "VISUALIZZA". Quando il processore riceve una richiesta di interruzione, esso sospende l'esecuzione della routine CALCOLA ed esegue la routine VISUALIZZA. Al termine della routine VISUALIZZA, il processore riprende l'esecuzione della routine CALCOLA che aveva sospeso. Il sottoprogramma attivato alla richiesta di interruzione è chiamato **routine di servizio di interruzione** (*Interrupt Service Routine, ISR*)

3.2.2 Routine di servizio dell'interruzione

Abbiamo introdotto la routine di servizio di interruzione che ha caratteristiche molto simili alla chiamata a sottoprogramma:

- Quando il processore riceve l'**interrupt request**, il **PC** deve puntare alla routine di servizio, e, in maniera analoga ai sottoprogrammi, deve essere **salvato o nella pila o su un registro** per far sì che riprendi la compilazione del programma dall'istruzione successiva a quella nella quale si era fermato²;
- Una volta terminata la routine di servizio, si ritorna al programma principale con l'istruzione **return-from-interrupt**, che riporta PC all'indirizzo successivo all'istruzione che stava eseguendo in quel momento;
- Inoltre il **processore informa la periferica dell'avvenuta ricezione dell'interrupt** tramite la linea **INT_ACK** nel bus di controllo.

3.2.3 Servizio delle interruzioni

Le routine di servizio delle interruzioni non vengono progettate dal programmatore, ma sono progettate da chi ha costruito il calcolatore. Inoltre non possiamo fare assunzioni su quando queste routine di servizio verranno eseguite poiché le **interruzioni sono incerte**. Di conseguenza è importante **salvare i dati più importanti** (PC, bit di stato, registri) in modo tale che siano ripristinati una volta che si riprende l'esecuzione del programma. Un altro registro fondamentale che deve essere salvato è il **PS (program status)**, che è un registro che **contiene bit di stato** e di cui vedremo l'utilizzo nel **controllo delle interruzioni**. Per salvare il registro PS, la routine di servizio utilizza un **registro ausiliario**, cioè una copia del registro PS oppure, molto raramente (poiché occupa molta memoria), un **banco registri ausiliario**. Inoltre tale meccanismo di interruzione è fondamentale per gestire eventi esterni, in molti sistemi di controllo, con routine di servizio in **real time**

²Quando il processore riceve l' interrupt request comunque deve completare l'ultima istruzione che stava eseguendo

3.2.4 Controllo delle interruzioni

Molto spesso potrebbe capitare che per certi programmi io non debba essere "interrotto", ad esempio quando sono in una routine di servizio e devo saltare in un'altra poichè ciò provocherebbe diversi **annidamenti di PS** e rallenterebbe il programma. Quindi devo prevedere all'interno del programma meccanismi per **abilitare/disabilitare le interruzioni**:

- All'interno del processore, specificatamente nel registro PS, si trova il **bit IE**, che quando vale 1 **abilita le routine di servizio**, viceversa, quando vale 0 le **disabilita**;
- Anche all'interno dell'interfaccia della periferica, nel registro di controllo, si trova un **IE** che, quando vale 1, indica che la periferica può imporre a 1 la linea di segnale per l'**interrupt request**, viceversa, quando vale 0, non può imporre la linea di segnale a 1 per l'interrupt request.

Dopo aver introdotto che esiste questo bit IE sia all'interno del processore, che della periferica, vediamo la **sequenza di eventi per una richiesta di interruzione**, così da capire come vengono abilitate/disabilitate le interruzioni. Prendiamo come periferica di esempio la *tastiera*:

1. Viene premuto un tasto dall'utente;
2. Attraverso il **circuito di controllo** della tastiera, viene codificato il tasto in binario seguendo la codifica ASCII;
3. Il bit **KIN**, del registro di stato, viene imposto a 1, dichiarando che esiste un dato nel registro dati;
4. La periferica attiva l'**INT_REQ** pilotando il segnale in alto, grazie all'utilizzo del **bit KIRQ**, contenuto nel registro di stato, che ha appunto questa funzione di alzare il segnale dell'interrupt request quando è posto a 1 e abbassarlo quando è posto a 0;
5. Il processore sospende il programma che stava eseguendo **salvando PC e PS**, che contiene il **bit IE**;
6. Prima di eseguire la routine di servizio dell'interruzione pone il bit IE a 0, così da non ricevere ulteriori interruzioni;
7. Viene eseguita dal processore la **routine di servizio** e viene informata la periferica dell'esecuzione o attraverso il segnale **INT_ACK** oppure non utilizzando questo segnale poichè i registri della periferica sono tale da **capire autonomamente che il processore ha letto il dato dalla periferica**.
8. Una volta terminata la routine di servizio, vengono **ripristinati PC e PS**, quindi anche il bit **IE ripristinato a 1**, e si ritorna all'esecuzione del programma interrotto.

3.2.5 Dispositivi multipli

Fino ad adesso abbiamo parlato di singole interruzioni e delle loro rispettive routine di servizio, ma attraverso il bus noi **colleghiamo diverse periferiche al processore**. Di conseguenza quando abbiamo una singola richiesta il processore deve identificare la periferica dalla quale proviene la richiesta in modo tale da abilitare la routine di servizio apposita per tale periferica. Esistono due modi con cui il processore può identificare la richiesta:

- **Polling o scansione:** in tal caso il processore scansionerà le varie periferiche collegate con il bus, in modo tale da vedere che quando il **bit IRQ**, del registro di stato delle periferiche, è posto a 1, quella determinata periferica ha fatto la richiesta. Inoltre il processore avrà un **ordine di precedenza** nello scansionare le varie periferiche. Capiamo, però, che questa tecnica può essere utile con poche periferiche, poiché, in caso contrario, abbiamo un tempo di interruzione che si allungherebbe troppo;
- **Interruzioni vettorizzate:** l'altro metodo ben più utilizzato è sicuramente quello delle **interruzioni vettorizzate**. Con tale metodo la periferica che mette alto il segnale di interrupt request, manda un **codice di pochi bit** al processore. Tale codice identifica la periferica attraverso una **tabella di vettori d'interruzione**, che rappresentano ognuno una periferica, e di cui il codice ne rappresenta l'**indice**, poiché ovviamente la periferica non riuscirebbe a gestire bit tali che abbiano dimensione di una parola di memoria. L'**indirizzo del vettore d'interruzione** dato dall'indice inviato dalla periferica viene letto dal processore che lo passa al PC, così da eseguire la specifica routine di servizio della periferica.

3.2.6 Annidamento delle interruzioni

Abbiamo visto come il processore riesce a comprendere come identificare la periferica ed eseguire la rispettiva routine di servizio. Ma cosa succede se mentre sta eseguendo una routine di servizio riceve una richiesta di interruzione? Piuttosto che abilitare/disabilitare le interruzioni, che è un processo molto dispendioso, si utilizzano **livelli di priorità**, codificati nel registro PS:

- Quando la periferica che sto eseguendo ha un **livello di priorità più alto** rispetto a un'altra, il processore la serve;
- Quando la periferica che ha richiesto l'interruzione ha **livello di priorità più basso** non la servo.

Succede, quindi, che quando arriva un'interruzione vengono **confrontati i bit di priorità** contenuti in PS e quando il processore inizia a servire la richiesta il **livello di priorità viene alzato**, cosicché non ci sia possibilità che un'altra richiesta di interruzione possa interrompere la routine di servizio. Quando usciamo dalla routine di servizio vengono **ripristinati i livelli di priorità**, quindi il registro PS, e le interruzioni non eseguite vengono salvate nel **registro IPENDING**.

3.2.7 Richieste di interruzioni simultanee

Abbiamo visto il caso nel quale il processore serve le richieste con livello di priorità più alto, ma cosa succederebbe se avvenissero **diverse richieste contemporaneamente**? Esistono del calcolatore dei **circuiti di arbitraggio** (hardware) che selezionano una periferica: quando viene alzato il segnale di interrupt request dalla periferica, essa non solo è collegata tramite il bus alla periferica, ma è collegata anche a questi circuiti che **disabilitano la possibilità che altre periferiche possano richiedere ulteriori interruzioni**. Di conseguenza una sola periferica è abilitata a inviare il segnale di interruzione tramite questi circuiti. Un altro modo per vedere quale richiesta viene servita è una scansione del processore dei registri di stato delle varie periferiche, controllando l'ordine di scansione.

3.2.8 Controllo della richiesta

Facendo un piccolo riassunto di ciò che abbiamo detto, sappiamo che per l'abilitazione/disabilitazione delle richiesta può essere fatta dal processore o dall'interfaccia della periferica: di conseguenza, se il processore vuole disabilitare l'interruzione pone il **bit IE a 0**. Quando invece, c'è una richiesta si vede se il bit di stato del dato e IE sono entrambi a 1, in modo tale che il bit **IRQ** possa alzare il segnale e abilitare la richiesta di interruzione.

3.2.9 Registri di controllo del processore

Abbiamo inoltre compreso che, per il controllo delle interruzioni, il processore ha dei registri appositi:

- **PS**: che include al suo interno il **bit IE** che abilita/disabilita le interruzioni;
- **IPS**: nel registro IPS è il **registro ausiliario** nel quale vengono salvati automaticamente i contenuti di PS, quando una richiesta di interruzione è accettata. Quando abbiamo un annidamento di interruzioni bisogna salvare in pila il contenuto di IPS;
- **IENABLE**: è il registro del processore che permette di abilitare le varie periferiche assegnando una bit a ogni periferica che fa una richiesta;
- **IPENDING**: è il registro che indica le interruzioni attive. Visto che i registri riescono a contenere fino a 32 bit; al massimo IPENDING potrà contenere 32 richieste

Per accedere a tali registri in assembly dovremo usare un istruzione speciale **”Movecontrol”**:

```
MoveControl    R2,PS
```

3.2.10 Programmi con interruzioni

Vediamo adesso, in assembly, come scrivere un programma che preveda delle interruzioni, in particolare leggere da tastiera una linea di caratteri e stamparli a schermo; e inoltre vedremo come gestire più interruzioni contemporaneamente (**interrupt handler**).

Programma principale Il programma principale carica l'indirizzo LINEA nella posizione di memoria PNTR, usata poi dalla routine di servizio delle interruzioni. Abilita le interruzioni nel registro d'interfaccia della tastiera e nel processore. Abilita il processore ad accettare le interruzioni da tastiera, e le interruzioni in generale:

```
INIZIO:      MOVE   R2,#LINEA
             STORE  R2,PNTR
             CLEAR  R2
             STORE  R2, EOL
             MOVE   R2, #2
             STOREBYTE R2,KBD_CONT
             MOVECONTROL R2,IENABLE
             OR     R2, R2, #2
             MOVECONTROL IENABLE,R2
             MOVECONTROL R2,PS
             OR     R2,R2,#1
             MOVECONTROL PS,R2
```

Possiamo dire, come detto nella premessa, che il programma principale abilita sia l'interruzione della tastiera, ponendo il bit IE nel registro di controllo dell'interfaccia della tastiera "KBD_CONT" a 1, e inoltre abilita in generale le interruzioni ponendo IENABLE a 1, cosicchè il processore accetti le richieste dalla periferica corrente, e anche il bit IE contenuto in PS, cosicchè il processore abiliti la routine di servizio.

Routine di servizio Vediamo adesso la routine di servizio per gestire la tastiera e scrivere sul video:

Routine di servizio delle interruzioni			
ILOC:	Subtract Store Store Load LoadByte StoreByte Add Store	SP, SP, #8 R2, 4(SP) R3, (SP) R2, PNTR R3, KBD_DATA R3, (R2) R2, R2, #1 R2, PNTR	Salva i registri Carica il puntatore all'indirizzo Leggi un carattere dalla tastiera Scrivi il carattere in memoria Incrementa il puntatore Aggiorna il puntatore in memoria
ECO:	LoadByte And Branch_if_[R2]=0 StoreByte Move Branch_if_[R3]≠[R2] Move Store Clear StoreByte	R2, DISP_STATUS R2, R2, #4 ECO R3, DISP_DATA R2, #CR RTRN R2, #1 R2, EOL R2 R2, KBD_CONT	Attendi che lo schermo sia pronto Visualizza il carattere appena letto Codice ASCII per il Ritorno Carrello Rientra se non è CR Indica la fine della linea Disabilita le interruzioni nell'interfaccia della tastiera
RTRN:	Load Load Add Return-from-interrupt	R3, (SP) R2, 4(SP) SP, SP, #8	Ripristina i registri

Figure 3.3: Routine di servizio

Analizzando le istruzioni del programma sequenzialmente:

1. Vediamo che innanzitutto salviamo i registri, ricavandoci nella pila uno spazio di 2 posizioni scrivendo i registri R2 ed R3;
2. Carichiamo il puntatore all'indirizzo che nel programma principale avevamo dato il nome simbolico di *LINEA* che identifica il punto in cui salveremo i caratteri che leggiamo dal registro dati "KBD_DATA". Infine carichiamo il valore del registro dati in R3, e lo immagaziniamo all'indirizzo puntato da R2 (PNTR);
3. Incrementiamo e aggiorniamo il valore del puntatore
4. Per scrivere il dato nel display, controlliamo se il registro di stato del display, appunto, è diverso da 0 attraverso un AND per controllare il bit in posizione 2 (DOUT).
5. Se non abbiamo un ritorno carrello visualizziamo il carattere letto e disabilitiamo le interruzioni da tastiera, azzerando il registro KBD_CONT.

Gestione delle interruzioni (interrupt handler) Quando più dispositivi inviano contemporaneamente una richiesta di interruzione, si utilizza il **registro IPENDING**, in modo tale che se il bit per quel determinato dispositivo è impostato a 1, significa che c'è una richiesta attiva da parte di quella periferica che ancora il processore deve servire. Altrimenti si passa alla prossima periferica facendo lo stesso controllo. In particolare Il programma Main abilita ciascuna

periferica (registri di interfaccia e registro PS). Alla richiesta di interruzione, si carica automaticamente l'indirizzo della routine di servizio rispettiva in PC e infine si identifica il dispositivo tramite il registro IPENDING.

3.3 Eccezioni

Abbiamo parlato in maniera specifica, in questo capitolo, di interruzioni provenienti da dispositivi di I/O. Più in generale possiamo dire che qualsiasi evento provochi un'interruzione è chiamato **eccezione**:

- **Eccezioni da memoria:** può essere ad esempio che ci siano dei dati inesistenti o alterati, provocando quindi una richiesta di interruzione al processore;
- **Altre Eccezioni:** ad esempio il tentativo di fare una divisione per zero, oppure tentativi di accesso a parti di memoria protette;

Tutto ciò provoca una richiesta di interruzione che sospende il programma principale ed esegue una routine di servizio, e, se si tratta di un errore il programma rinuncia all'esecuzione del programma principale

Capitolo 4

STRUTTURA DI BASE DEL PROCESSORE

4.1 Concetti fondamentali

Sappiamo, già dai capitoli precedenti, che l'esecuzione di un'istruzione si sviluppa in 3 passi, che avvengono sequenzialmente:

- L'istruzione deve essere **prelevata dalla memoria** puntata dal **registro PC**, e deve essere inserita nel **registro IR**;
- Viene **incrementato PC** alla prossima istruzione, ovvero alla prossima parola di memoria;
- Si **esegue l'istruzione**;

I primi 2 passi compongono la **fase di prelievo**, detto **fetch**, mentre l'**esecuzione dell'istruzione**, si sviluppa tramite le seguenti azioni:

- Caricamento di un **contenuto di memoria in un registro**;
- **Lettura di dati** da un registro del processore;
- **Esecuzione di istruzioni aritmetiche**;
- **Scrittura dei dati** da un registro alla memoria;

4.1.1 Componenti hardware

Le **componenti hardware** del processore che ci permettono quindi di eseguire un'istruzione sono:

- **Banco registri**: assiste il processore fornendo **registri** nelle **operazioni di caricamento dalla memoria e scrittura in memoria**;

- **ALU:** esegue tutte le operazioni aritmetico/logiche;
- **Circuiti di controllo:** unità che genera segnali di controllo;
- **Registro IR:** contiene l'istruzione da eseguire per tutta la durata dell'esecuzione;
- **Registro PC:** contiene l'indirizzo della prossima istruzione da eseguire;
- **Generatore di indirizzi:** che aggiorna PC dopo aver prelevato l'istruzione;
- **Interfaccia tra processore e memoria:** permette di trasferire dati dal processore alla memoria

Possiamo vedere bene tutte le componenti hardware del processore in questa figura:

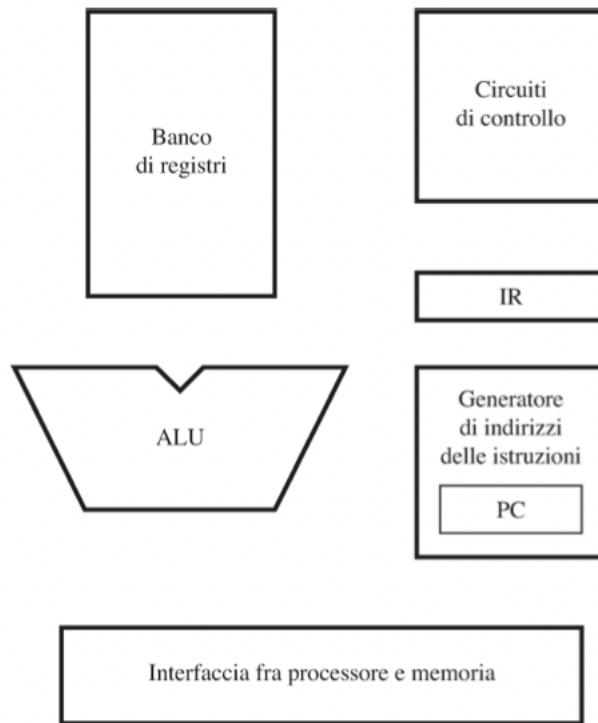


Figure 4.1: Componenti hardware del processore

4.1.2 Hardware per l'elaborazione dati

Con le componenti hardware, posso creare una **funzione logica** formata da varie **porte logiche**, che rappresentano le differenti operazioni logiche appunto,

attraverso un **circuito combinatorio**. Tale circuito ha delle **caratteristiche**, che sono:

- **Tempo di propagazione**: è il **tempo che passa dall'ingresso all'uscita di un valore** dal circuito. Questo tempo è dato dalla **somma dei tempi di propagazione delle porte logiche del circuito**;
- **Periodo di clock**: è il **tempo tra due successivi fronti di salita** (vedi sezione E.2 dell'appendice). Questo tempo deve essere **sufficientemente lungo per comprendere il tempo di propagazione** per produrre l'uscita
- **Flip-Flop**: i **registri**, appunto, sono flip-flop che sono **memorizzati sul fronte di salita del clock** (vedi sezione E.3).

Soltamente non si utilizza un unico circuito combinatorio, poichè molto dispendioso dal punto di vista della **velocità di esecuzione**. Per questo solitamente si **suddivide il circuito combinatorio** in vari "pezzi", che vengono chiamati **stadi** in cascata, cioè che agiscono in maniera sequenziale. In tal modo abbiamo sequenze di porte logiche suddivise in più parti aventi, quindi, un **periodo di clock minore** e una **frequenza di clock maggiore**; e, di conseguenza, un **tempo di esecuzione dell'istruzione minore**

4.1.3 Struttura hardware a più stadi

Abbiamo accennato nella sezione precedente alla **struttura hardware a più stadi**; vediamo adesso come è effettivamente composta tale struttura. Innanzitutto esistono dei registri, quelli di "inizio" e di "fine", che rispettivamente **prendono dei valori in input** nel primo stadio e **salvano i valori dall'uscita del circuito** nell'ultimo stadio. Ad ogni stadio abbiamo piccoli circuiti che **elaborano i dati**, e il dato che forniscono farà da **input per il prossimo stadio**. Questo "ciclo" si ripete per tutti gli stadi in cascata: la conseguenza quindi è, che, avendo un periodo di clock minore con una frequenza di clock maggiore ogni stadio deve essere regolato in modo tale che i vari **tempi di propagazione per ogni stadio siano simili fra loro**. Inoltre sappiamo che **n stadi equivalgono a n cicli di clock**. Possiamo vedere un esempio di struttura hardware a più stadi in tale figura:

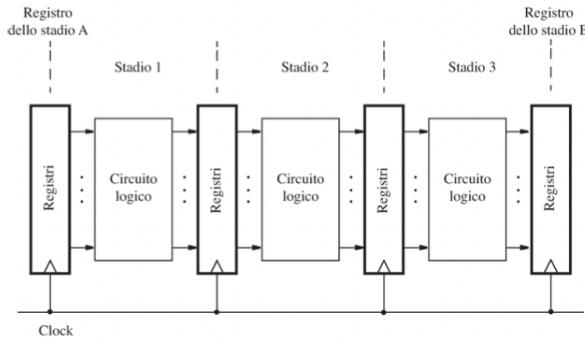


Figure 4.2: Struttura hardware a più stadi

4.2 Esecuzione di istruzioni

In un processore RISC abbiamo essenzialmente tre categorie di istruzioni:

- **Caricamento** (load);
- **Operazioni aritmetico/logiche** (add, or...);
- **Immagazzinamento** (store);

Sappiamo che, per ogni categoria di istruzione, si procede in *5 passi*, che corrispondono ognuno a uno **stadio hardware**. Vediamo adesso per ogni categoria questi passi.

4.2.1 Istruzioni di caricamento

Prendiamo come esempio l'istruzione:

Load R5, X(R7)

Vediamo i 5 passi dell'istruzione:

1. **Prelievo** dell'istruzione della memoria e **incremento PC**;
2. **Decodifica** dell'istruzione e **lettura** del contenuto di R7. Sapendo che nella decodifica dell'istruzione **i registri occupano una posizione ben precisa**, quindi, posso già leggere il contenuto di R7;
3. **Calcolo** dell'indirizzo effettivo $X + [R7]$. Quindi, sapendo il contenuto di R7, con l'**ALU** si calcola l'**indirizzo effettivo della locazione di memoria**;
4. **Lettura** dell'operando **dalla memoria** all'indirizzo effettivo $X + [R7]$;
5. **Scrittura** dell'operando nel registro R5;

In tal caso abbiamo utilizzato l'istruzione *load* con lo spiazzamento; però ovviamente nel caso non usassimo lo spiazzamento possiamo eseguire l'istruzione in questi 5 passi modificando leggermente il terzo passo:

- Se ci viene passato un indirizzo senza spiazzamento, **tramite registro** quindi, semplicemente l'ALU calcolerà l'indirizzo effettivo senza spiazzamento;
- Se ci viene passato un **valore immediato**, l'ALU calcolerà l'indirizzo effettivo con spiazzamento "0"

4.2.2 Istruzioni aritmetico/logiche

Vediamo, anche per questa categoria di istruzioni, i 5 passi che utilizziamo per l'esecuzione dell'istruzione. Prendiamo come esempio tale istruzione:

`ADD R3, R4, R5`

Abbiamo questi passi:

1. **Prelievo** dell'istruzione della memoria e **incremento PC**;
2. **Decodifica** dell'istruzione e **lettura** del contenuto dei registri R4 e R5;
3. **Calcolo** della somma;
4. **Nessuna azione**: infatti in tal caso **non leggiamo alcun operando dalla memoria**;
5. **Scrittura** del risultato nel registro R3;

Una **variante** di tale istruzione può essere quella nella quale specifichiamo un **registro sorgente** e **un valore immediato**: in tal caso nel secondo passo leggiamo solo il registro sorgente, e, nel passo 3 si calcola la somma col valore immediato.

4.2.3 Istruzioni di immagazzinamento

Prendiamo, per tale categoria di istruzioni, questo esempio:

`Store R6,X(R8)`

Vediamo i 5 passi:

1. **Prelievo** dell'istruzione della memoria e **incremento PC**;
2. **Decodifica** dell'istruzione e **lettura** del contenuto dei registri R6 e R8;
3. **Calcolo** l'indirizzo effettivo $X + [R8]$;
4. **Immagazzinamento** di R6 in memoria all'indirizzo effettivo;
5. **Nessuna azione**: infatti non scriviamo alcun valore nei registri.

4.2.4 Schema di esecuzione delle istruzioni

Notiamo quindi che lo **schema di esecuzione delle istruzioni** sia alquanto simile fra le varie categorie di istruzioni. Per cui possiamo riassumere questi 5 passi in uno **schema generale**:

1. **Prelievo** dell'istruzione della memoria e **incremento PC**;
2. **Decodifica** dell'istruzione e **lettura** di registri dal **banco registri**;
3. **Operazione con l'ALU**;
4. **Leggi o scrivi in memoria**, se l'istruzione coinvolge un operando in memoria;
5. **Scrittura nel registro destinazione**, se l'istruzione prevede di scrivere in un registro del processore.

4.3 Hardware e percorso dati

Vediamo in questa sezione come i dati vengono elaborati nelle varie componenti hardware (banco registri, ALU) durante l'esecuzione di un'istruzione, vedendo effettivamente la loro struttura.

4.3.1 Banco di registri

Abbiamo visto nei 5 passi di esecuzione di un'istruzione che possiamo avere nel secondo passo, 1 o 2 **registri sorgente** dai quali leggiamo dei dati, oppure, nel quinto passo abbiamo un **registro destinazione** nel quale scriviamo un dato elaborato durante l'istruzione. Il **banco registri**, essendo l'unità interna al processore che fornisce i registri, ci dovrà fornire a seconda dell'istruzione 1 o 2 registri sorgente e 1 registro destinazione. Per quanto riguarda la **codifica del registro** da leggere, questa si trova già nella "codifica" dell'istruzione stessa: infatti sappiamo che i registri nella codifica dell'istruzione occuperanno una **posizione ben precisa**. Il banco registri riceve quindi in ingresso un **gruppo di bit** dal registro IR, dove è contenuta l'istruzione, che codificano un **indirizzo A** e un **indirizzo B**, che corrispondono a **2 registri sorgente** del **banco registri**, e inoltre verrà inviato un gruppo di bit per un **indirizzo C**, che corrisponde a un **registro destinazione all'interno del banco registri**. Il problema si pone, quindi, poiché se noi inviamo in ingresso un gruppo di bit che identificheranno 2 indirizzi del banco registri, quest'ultimo dovrà fornirci **contemporaneamente 2 registri sorgente in uscita**. Sappiamo, però, che le memorie sono fatte per prendere un ingresso e fornire un'uscita; di conseguenza in questo caso si può fare una duplice scelta di costruzione, che sarebbe:

- **Duplicare la memoria**: in tal modo, avendo due banchi registri, mandiamo un gruppo di bit che codificano l'indirizzo A ad un banco di registri, che ci fornirà in uscita un registro sorgente; e, mandiamo un gruppo di bit

che codificano l'indirizzo B, contemporaneamente, a un altro banco registri che ci fornirà, sempre contemporaneamente, l'altro registro sorgente. Per quanto riguarda l'indirizzo C, che codifica il registro destinazione, in questo caso duplichiamo il dato in ingresso, in modo tale da averlo a disposizione in entrambe le uscite;

- **Duplicare i circuiti di accesso:** in tal modo utilizziamo un unico banco registri, che però presenta appunto più circuiti per ricevere in ingresso gli indirizzi A,B.

4.3.2 ALU

L'**ALU** ha una struttura per la quale presenta **due ingressi** collegati alle **due uscite del banco registri**, in particolare:

- L'uscita **A** del **banco registri** è collegata direttamente con l'ingresso **InA** dell'**ALU**;
- L'uscita **B** del **banco registri**, invece, è collegata a un **multiplatore**, precisamente nell'ingresso "0". Questo multiplatore, avrà il compito di scegliere l'operando sorgente da inviare all'ALU per l'elaborazione, in base alla codifica dell'istruzione: infatti, se abbiamo come **operando un valore immediato**, che è codificato nel multiplatore con la linea di ingresso "1", quest'ultimo sceglierà in uscita l'operando sorgente immediato. Se, invece, la codifica dell'istruzione prevede due registri come operandi sorgente, sceglierà come linea di ingresso "0", ovvero quella dove collegavamo l'uscita B del banco registri, selezionando quindi un **registro come operando sorgente**.

Una volta eseguita l'elaborazione dei dati da parte dell'ALU, l'**uscita** di quest'ultima è collegata con l'**indirizzo C**, ovvero quello che codifica il **registro destinazione** nel quale viene scritto un dato, che verrà salvato nel banco registri. Possiamo fare una piccola considerazione sui **valori immediati**: infatti noi sappiamo che ogni registro occupa 32 bit, ovvero una parola di memoria, mentre il valore immediato ne occupa solo 16. Di conseguenza, se prendiamo un valore immediato a 16 bit dovremo **estendere a 0 i bit più significativi**. Per questo esiste un **circuito di controllo**, che esegue questa estensione a 0 dei bit. Possiamo vedere, ciò che abbiamo fino ad ora descritto con parole in questa figura:

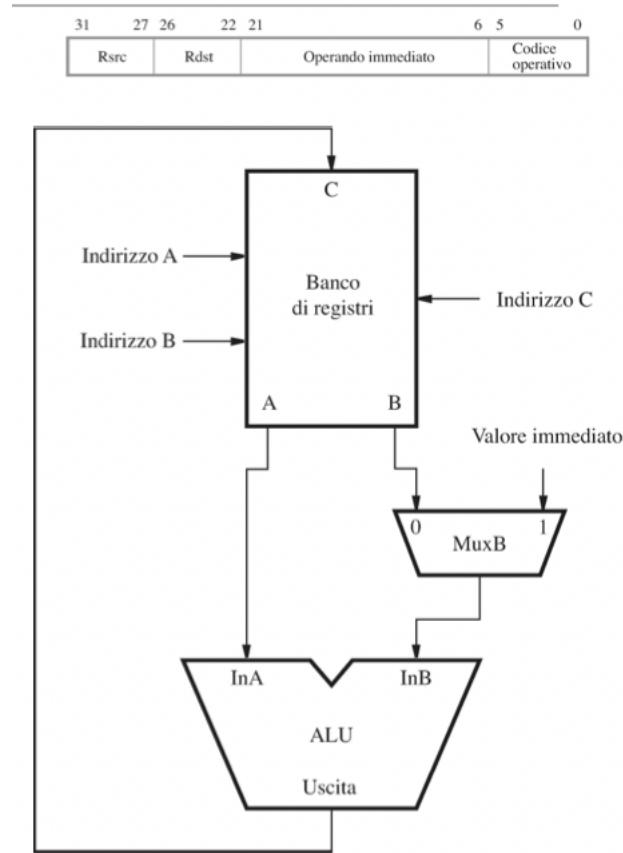


Figure 4.3: Collegamento ALU-banco registri

4.3.3 Struttura a 5 stadi

Abbiamo già detto che l'esecuzione di istruzione avviene in 5 passi, ognuno dei quali corrisponde a uno **stadio**. Vediamo adesso, conoscendo più nello specifico le componenti hardware la struttura dei 5 stadi:

1. **Stadio 1:** Prelievo istruzione che viene posta nel registro IR, **incremento PC**;
2. **Stadio 2:** Decodifica istruzione e lettura registri sorgente dal banco registri;
3. **Stadio 3:** ALU;
4. **Stadio 4:** Accesso alla memoria;
5. **Stadio 5:** Scrittura nel registro destinazione.

Sappiamo, inoltre, che **ogni stadio è completato in un periodo di clock**

4.3.4 Percorso dati (datapath)

Vediamo adesso, una volta compreso il compito di ogni stadio, il **percorso dei dati**, durante l'esecuzione dell'istruzione. Dalla struttura a 5 stadi vista nella sezione precedente, possiamo dire quindi che:

- Il passo 2 e 5 utilizzano il **banco registri**;
- Il passo 3 utilizza l'**ALU**;
- Il passo 4 legge o scrive in **memoria**;
- Il passo 5 scrive il risultato in un **registro destinazione**;

Tutti questi passi sono uniti in un unico **circuito**, che presenta dei *registri interstadi* che hanno una specifica funzione, ovvero quella di **tenere i risultati prodotti da uno stadio**, rimanendo stabili per il periodo di clock successivo, e, di conseguenza, facendo da **input per il prossimo stadio**. Vediamo ad esempio tale circuito:

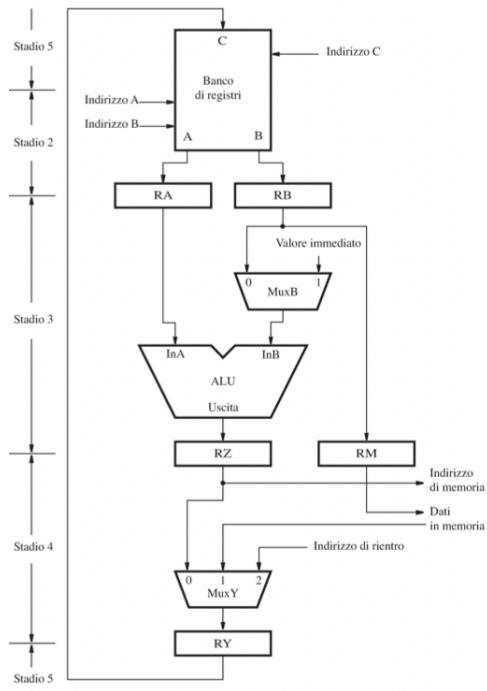


Figure 4.4: Percorso dati nei vari stadi

Analizzando i vari stadi dall'immagine possiamo dire:

- **Stadio 2:** Opera il **banco registri**, che fornisce i registri che ci serviranno per l'istruzione. L'uscita del banco registri verrà posta nei **registri interstadio RA e RB**, che per tutto il periodo di clock manterranno il valore stabile, che manderanno in input al prossimo stadio
- **Stadio 3:** Con l'inizio del terzo stadio, cioè con l'inizio del periodo di clock successivo al secondo, il registro RA e RB vengono mandati come ingresso per l'unità successiva, ovvero l'**ALU**. In particolare l'uscita RB viene collegata a un **multiplexer** (MuxB), in modo tale che in base alla codifica dell'istruzione, si sceglierà la *linea di ingresso del multiplexer adeguata* (1: valore immediato; 0: registro operando). L'uscita del multiplexer è poi collegata ad un ingresso dell'**ALU**, che, quando terminerà l'elaborazione dei dati (operazioni aritmetico-logiche), manderà l'output nel **registro interstadio RZ**. In RZ, precisamente, verrà scritto un **indirizzo effettivo** per le istruzioni come la *load/store*, mentre un valore **risultato**, per operazioni come la *add, subtract, or....*
- **Stadio 4:** Questo stadio prevede di **interagire con la memoria**. In tal caso bisogna fare un distinguo fra le varie categorie di operazioni:
 - *Load:* in tal caso mandiamo in memoria l'indirizzo contenuto in RZ attraverso il bus degli indirizzi, in particolare, nella **porta della memoria che contiene gli indirizzi**. In seguito la memoria ci restituisce il **dato contenuto nell'indirizzo**, che verrà collegato nella *linea di ingresso 1* del multiplexer "MuxY", così da scrivere il valore nel **registro interstadio RY**, e mandarlo come input in ingresso all'*indirizzo C del banco registri* nello **stadio 5**.
 - Operazioni *aritmetico-logiche*: in questo caso non serve interagire con la memoria perchè, ovviamente, lavoriamo con i registri del processore. Per cui, nello stadio 4, copio semplicemente il dato dal registro RZ, passando per la linea di ingresso "0" di MuxY, al registro RY. Infine, nello **stadio 5** tale dato viene inviato come ingresso del banco registri all'indirizzo C;
 - *Store:* nel caso di *immagazzinamento in memoria*, si attiva il **registro interstadio RM**: infatti, una volta decodificata l'istruzione, questo registro prende un dato proveniente dal banco registri, che poi passa in RB, e lo **invia in memoria**. Quindi il flusso che arriva in RM si attiva solamente quando eseguiamo un'istruzione **store**, per il quale stiamo fornendo un indirizzo di memoria nel quale scrivere un dato. Quindi abbiamo, nel caso di una store, che il dato che esce dall'indirizzo B del banco registri passa nel registro RB nello *stadio 2*, va nel registro RM nello *stadio 3*, e infine nello *stadio 4* viene inviato all' indirizzo di destinazione in memoria codificato dal dato. Nello **stadio 5** in questo caso non accade niente
- **Stadio 5:** Scrive risultato in un registro (indirizzo C del banco registri) se necessario: infatti abbiamo già visto che nel caso di una load oppure

un'operazione aritmetico-logica avviene la scrittura del risultato, nel caso di una store non accade nulla;

Esempi Vediamo adesso degli esempi, utilizzando istruzioni *assembly*, del percorso dei dati nelle varie unità hardware:

- Vediamo l'esempio di una **load**:

Load R5, (X)R7

Analizziamo adesso i vari *stadi*:

- **Stadio 1:** preleva l'istruzione che viene posta nel *registro IR*, e *incrementa PC*;
- **Stadio 2:** leggi R7 e metti il valore di R7 nel *registro interstadio RA*;
- **Stadio 3:** manda il contenuto di RA come input dell'*ALU*, e, attraverso il *multiplexer* (MuxB), con la codifica dell'istruzione, mandiamo un valore immediato nella linea di ingresso 1 di MuxB, collegata a un ingresso dell'*ALU*. Mandiamo infine il risultato, che sarebbe l'indirizzo effettivo, nel *registro interstadio RZ*.
- **Stadio 4:** faccio una lettura dell'indirizzo in RZ; il dato che corrisponde all'indirizzo contenuto in RZ, viene restituito dalla memoria, passando per la linea di ingresso 1 di MuxY, e, in seguito viene posto nel *registro RY*;
- **Stadio 5:** scrivi il contenuto di RY in R5

- Vediamo adesso l'esempio di una **store**:

Store R6, (X)R8

Analizziamo i vari **stadi**:

- **Stadio 1:** preleva l'istruzione che viene posta nel *registro IR*, e *incrementa PC*;
- **Stadio 2:** leggi dal banco registri R6 e R8, e poni R8 in RA e R6 in RB;
- **Stadio 3:** Esegui la somma, nell'*ALU*, tra il contenuto di RA (R8) e il valore immediato X, passato in MuxB per la linea di ingresso 1, e inviato come input all'*ALU*, e poni il valore della somma in RZ. Mentre poni RB, che contiene R6, in RM;
- **Stadio 4:** Scrivi nella locazione di memoria con indirizzo RZ il valore RM

4.3.5 Sezione di prelievo delle istruzioni

Abbiamo parlato nel percorso dati di come il flusso di dati si concretizzasse negli stadi da 2 a 5. Non abbiamo trattato in profondità, però, lo *stadio 1*, ovvero quello che si occupa del **prelievo dell'istruzione**. In tal caso **il generatore di indirizzi ci fornisce l'indirizzo di PC**, ovvero dove si trova la **prossima istruzione in memoria**: succede che mandiamo il valore di PC a un *multiplexer*, chiamato "MuxMa", che sceglie la sorgente da inviare in memoria. Infatti nello stadio 1, la linea di ingresso del multiplexer sarà sempre 1, e, in memoria, nell'indirizzo di PC, troveremo l'istruzione che salveremo in IR. Lo scopo del multiplexer in questo caso è scegliere, con la linea di ingresso 1, che, come abbiamo detto, sarà sempre nel primo stadio, di prelevare l'indirizzo della prossima istruzione; mentre, come abbiamo visto in precedenza, nello stadio 4 nel caso di una *load*, ci servirà anche in questo caso prelevare un dato contenuto in un indirizzo di memoria, che viene inviato tramite l'indirizzo di memoria contenuto in RZ. In tal caso la linea di ingresso sarà sempre la linea "0", che corrisponderà sempre al *quarto stadio*. Possiamo vedere ad esempio da quest'immagine ciò che abbiamo descritto:

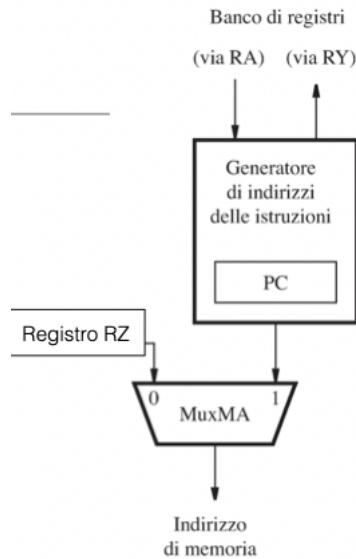


Figure 4.5: Prelievo dell'istruzione

Per quanto riguarda il **contenuto del registro IR**, quando l'istruzione viene posta in IR, grazie a dei **circuiti di controllo** vengono generati **segnali di controllo**, di cui parleremo in seguito, che cambiano in base all'istruzione che stiamo eseguendo. Nel caso in cui l'istruzione contenuta in IR, contenga un **valore immediato**, essendo l'istruzione composta da 32 bit, mentre il valore immediato ne contiene solo 16; succede che vengono **estesi a 0 i bit più**

significativi attraverso un *blocco hardware* chiamato **”immediato”**.

Generatore di indirizzi delle istruzioni

Abbiamo compreso che il **generatore di indirizzi delle istruzioni** ha il compito di **cambiare opportunamente l'indirizzo contenuto in PC**, infatti fino ad adesso abbiamo analizzato semplicemente il caso in cui l'istruzione successiva si trova nella parola di memoria successiva, quindi bastava incrementare PC di 4; ma cosa succede se dovessimo incrementare PC di un determinato spiazzamento per eseguire l'istruzione di salto o chiamata a sottoprogramma? Per dare una risposta basta analizzare l'unità del generatore di indirizzi e capire come funziona:

- Abbiamo un certo valore in PC dato ad esempio da una direttiva *ORIGIN* che ci indica l'indirizzo di inizio programma;
- In questo passaggio bisogna distinguere se dobbiamo fare un'operazione non di salto, oppure un'operazione di salto:
 - Nel primo caso, quello in cui l'operazione non è un salto, il contenuto di PC viene copiato in un "registro interstadio" *PC-Temp*, che è un registro che salva temporaneamente l'indirizzo contenuto in PC. Esiste all'interno del generatore di indirizzi una componente chiamata **sommatore**, indipendente dall'ALU, che ha il compito di prendere in input il valore di PC, e, attraverso la linea di ingresso di un multiplexer chiamato *MuxINC*, prendere un incremento da sommare a PC. Nel caso che l'istruzione non sia di salto, la linea di ingresso scelta è la "0", che contiene il valore 4, ovvero i 4 byte che indicano la parola di memoria successiva, e quindi la prossima istruzione. Dopo aver sommato, attraverso il sommatore *PC + 4*, questo lo manderà in output a un altro multiplexer chiamato "MuxPC", che sceglierà la linea di ingresso 1, per mandare l'indirizzo in PC. Alla fine di questo "ciclo" PC e PC-temp conterranno l'*indirizzo dell'istruzione successiva*.
 - Nel caso in cui l'operazione sia un **salto**, il passaggio che cambierà sarà che noi dobbiamo passare un **valore immediato**, anziché 4, da sommare a PC all'interno del sommatore. Questo valore immediato sarà dato dallo spiazzamento dell'istruzione corrente al valore dell'etichetta che indica l'indirizzo nel quale avviene il salto. Quindi, in *MuxINC*, la linea di ingresso scelta sarà "1", attraverso il quale passeremo questo valore immediato da sommare a PC. Per il resto delle azioni il generatore si comporta allo stesso modo.

Possiamo vedere come è composto il generatore di indirizzi all'interno, così da capire al meglio ciò che è stato descritto:

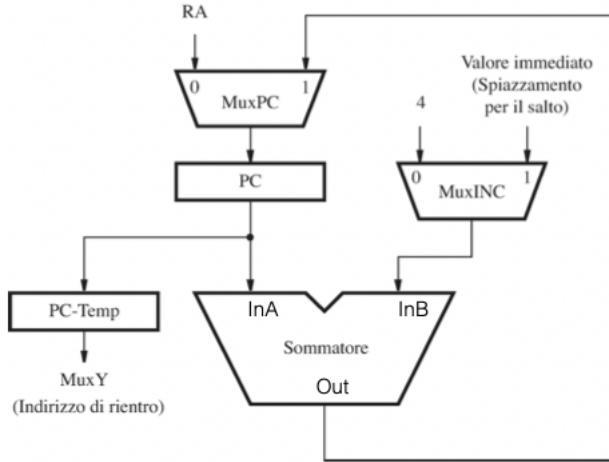


Figure 4.6: Generatore di indirizzi delle istruzioni

Non abbiamo ancora analizzato però l'istruzione di **chiamata a sottoprogramma**. Ricordiamo che l'istruzione tipo di chiamata a sottoprogramma era:

```
CALL    SUB1
```

Dove veniva decodificato il codice operativo della *call* e l'etichetta che indicava l'indirizzo dell'area di attivazione del sottoprogramma. Esiste però la forma del tipo:

```
CALL_REGISTER R9
```

In questo caso viene specificato il **registro sorgente che contiene l'indirizzo della subroutine**. Per quanto riguarda la decodifica di entrambe le istruzioni c'è una differenza: infatti se nel primo caso (CALL SUB1) utilizzavamo 8 bit per l'*op code*, abbiamo uno *spazio di indirizzamento* di 2^{24} parole di memoria; utilizzando quindi solo una parte di memoria. Nella chiamata a sottoprogramma attraverso un registro sorgente, noi abbiamo la possibilità di indirizzare **tutta la memoria**. Il programmatore, ovviamente, non si deve preoccupare di come il calcolatore è stato progettato per le chiamate a sottoprogramma, quindi è una cosa che dipende da ogni calcolatore. Ma quindi ci chiediamo quale sia il **percorso dati per la chiamata a sottoprogramma**: vediamo quindi cosa succede nei vari cicli di clock:

- Alla fine del **secondo ciclo**, una volta che è stata decodificata l'istruzione, il valore del salto, indicato dall'indirizzo contenuto nel registro sorgente, si troverà nel registro RA;
- Quindi, nel terzo passo, si deve **aggiornare PC**: infatti per questo si torna al circuito del generatore di indirizzi (figura 4.6), nel quale RA è specificamente collegato a *MuxPC* nella linea di ingresso 0. Infatti dopo che nel

secondo passo è stata decodificata l'istruzione di chiamata a subroutine; un **segnale di controllo** sceglie la linea di ingresso 0 in MuxPC;

- All'interno del generatore di indirizzi, quando passiamo RA in MuuxPC, *PC-temp* non viene stato aggiornato poiché in questo caso ci servirà come **indirizzo per l'istruzione di rientro**, visto che contiene l'indirizzo dell'istruzione precedente alla chiamata. Infatti come possiamo vedere dall'immagine 4.6, PC-temp è collegato a *MuxY*, e, come possiamo vedere dall'immagine 4.4, l'ingresso 2 di *MuxY* presenta l'indirizzo di rientro che verrà salvato in RY, e, nel passo 5, mandato al banco registri e scritto nel link register.

4.4 Passi di esecuzione

Vediamo adesso, per le varie categorie di istruzione, ciò che avviene nel *data-path* nei vari stadi attraverso la *notazione RTN*. Prima di comprendere con la notazione RTN ciò che avviene nei vari stadi, dobbiamo fare una premessa: per ogni istruzione ricordiamo che abbiamo una **posizione fissa dei registri** nella decodifica dell'istruzione e, che il banco registri avrà come ingresso sempre due indirizzi (A e B) poiché prima della fine del passo 2 l'istruzione non è stata ancora decodificata. Di conseguenza, i circuiti di controllo si attiveranno solo a partire dalla fine della decodifica, ovvero alla fine del passo 2. Questo è il motivo per cui abbiamo sempre 2 uscite dal banco registri, poiché non sappiamo sin da subito l'istruzione che dovrà essere eseguita, quindi, nel prelievo dell'istruzione verranno prelevati sempre 2 registri.

4.4.1 ADD

Vediamo l'esempio dell'istruzione ADD, che ricomprende la categoria di istruzioni **aritmetico-logiche**:

ADD R3,R4,R5

Analizziamo i vari passi:

1. Stadio 1:

- Indirizzo di memoria $\leftarrow [PC]$;
- Leggi memoria;
- IR \leftarrow dati da memoria;
- PC $\leftarrow [PC] + 4$

2. Stadio 2:

- Decodifica istruzione;
- $RA \leftarrow [R4]$, (banco registri: $RA \leftarrow$ Indirizzo A (R4));

- $RB \leftarrow [R5]$, (banco registri: $RB \leftarrow$ Indirizzo B (R5));
3. Stadio 3:
- $RZ \leftarrow [RA] + [RB]$, (ALU: $InA \leftarrow [RA]$; $InB \leftarrow MuxB = 0 \leftarrow [RB]$)
4. Stadio 4:
- $RY \leftarrow [RZ]$, ($MuxY = 0 \leftarrow RZ$)
5. Stadio 5:
- $R3 \leftarrow [RY]$, (banco registri: indirizzo C (R3) $\leftarrow [RY]$)

4.4.2 Load e Store

Vediamo l'esempio di istruzioni di **lettura** (*load*) o **scrittura** (*store*) in memoria. Iniziamo con la load:

Load R5, (X)R7

Analizziamo i vari passi:

1. Stadio 1:
- Indirizzo di memoria $\leftarrow [PC]$;
 - Leggi memoria;
 - IR \leftarrow dati da memoria;
 - PC $\leftarrow [PC] + 4$
2. Stadio 2:
- Decodifica istruzione;
 - $RA \leftarrow [R7]$, (banco registri: RA \leftarrow Indirizzo A (R7));
 - **NB:** In **RB** ci sarà il contenuto del registro che si trova nella codifica nei bit successivi a R7, ovvero R5, anche se poi **non verrà utilizzato**
3. Stadio 3:
- $RZ \leftarrow [RA] +$ valore immediato X, (ALU: $InA \leftarrow [RA]$; $InB \leftarrow MuxB = 1 \leftarrow X$)
4. Stadio 4:
- Indirizzo di memoria $\leftarrow [RZ]$;
 - Leggi memoria;
 - $RY \leftarrow$ dati da memoria ($MuxY = 1 \leftarrow$ memoria);
5. Stadio 5:

- $R5 \leftarrow [RY]$, (banco registri: indirizzo C ($R5 \leftarrow [RY]$))

Vediamo adesso l'esempio di una store:

Load R6,(X)R8

Analizziamo i vari passi:

1. Stadio 1:

- Indirizzo di memoria $\leftarrow [PC]$;
- Leggi memoria;
- IR \leftarrow dati da memoria;
- PC $\leftarrow [PC] + 4$

2. Stadio 2:

- Decodifica istruzione;
- $RA \leftarrow [R8]$, (banco registri: RA \leftarrow Indirizzo A (R8));
- $RB \leftarrow [R6]$, (banco registri: RB \leftarrow Indirizzo A (R6));

3. Stadio 3:

- $RZ \leftarrow [RA] +$ valore immediato X, (ALU: InA $\leftarrow [RA]$; InB $\leftarrow MuxB = 1 \leftarrow X$)
- $RM \leftarrow [RB]$;

4. Stadio 4:

- Indirizzo di memoria $\leftarrow [RZ]$;
- Dati da scrivere in memoria $\leftarrow [RM]$;
- Scrivi in memoria;

5. Stadio 5:

- Nessuna azione

4.4.3 Salti

In tal caso distinguiamo l'istruzione di salto in **salto incondizionato** e **salto condizionato**:

- **Salto incondizionato:**

1. Stadio 1:

- Indirizzo di memoria $\leftarrow [PC]$;
- Leggi memoria;
- IR \leftarrow dati da memoria;

- $PC \leftarrow [PC] + 4$
- 2. Stadio 2:
 - Decodifica istruzione
- 3. Stadio 3:
 - $PC \leftarrow PC + spiazzamento$ (valore immediato decodificato dall'istruzione)
- 4. Stadio 4:
 - Nessuna azione
- 5. Stadio 5:
 - Nessuna azione

Salto condizionato:

- 1. Stadio 1:
 - Indirizzo di memoria $\leftarrow [PC]$;
 - Leggi memoria;
 - IR \leftarrow dati da memoria;
 - $PC \leftarrow [PC] + 4$
- 2. Stadio 2:
 - Decodifica istruzione
 - $RA \leftarrow [R5]; RB \leftarrow [R6]$
- 3. Stadio 3:
 - Confronta RA e RB
 - $PC \leftarrow PC + spiazzamento$ (valore immediato decodificato dall'istruzione)
- 4. Stadio 4:
 - Nessuna azione
- 5. Stadio 5:
 - Nessuna azione

4.4.4 In attesa della memoria

Abbiamo visto che nel passo 1, quando *preleviamo l'istruzione dalla memoria*, e nel passo 4, dove *accediamo alla memoria* per leggere/scrivere dalla memoria, abbiamo assunto che anche questi 2 stadi vengono eseguiti in 1 ciclo di clock. Però, ci ricordiamo dal primo capitolo, che la **memoria è molto più lenta del processore**, e, inoltre, aggiungiamo che ogni **periodo di clock** è tarato e progettato per i passi eseguiti dal processore, ovvero i passi 2,3 e 5.

Infatti, anche se tutti gli stadi del processore durano un ciclo di clock, alcuni potrebbero finire prima di altri, però sappiamo che essendo tutti tarati per terminare nello stesso periodo ci si basa sullo stadio che potrebbe occupare un periodo più lungo, ovvero il passo 3 che utilizza l'ALU. Per quanto riguarda la memoria, se dobbiamo accedere alla memoria centrale per una lettura/scrittura, non riuscirà mai a completare l'istruzione di un ciclo di clock, ma ne impiegherà svariati: se ad esempio abbiamo un processore con una *frequenza* di 2GHz, e *periodo di clock* di 0,5ns, e consideriamo una *RAM*, che ha un tempo di accesso di 100 ns, capiamo che per completare l'istruzione potremmo impiegare più di 200 cicli di clock. Per questo esiste una memoria interna al processore, chiamata **memoria cache**, che ha un tempo di accesso che può coincidere con il ciclo di clock per la lettura. Esiste inoltre un segnale di controllo chiamato **MFC**, pilotato dalla memoria, che quando va a 1, significa che la memoria ha fornito il dato. In conclusione, abbiamo compreso che **il tempo di esecuzione per gli stadi che lavorano con la memoria è variabile**, ovvero ha un **numero di ciclo di clock variabile**

4.5 Segnali di controllo

Abbiamo accennato già al fatto che esistono dei *circuiti di controllo* che emettono **segnali di controllo**, che servono molto spesso in diverse istruzioni ad abilitare diversi registri attraverso i vari segnali di ingresso dei *multiplexer*. I **registri interstadio** saranno **sempre abilitati**, poichè non ci occorre essere più precisi per tali registri, visto che fanno da input tra uno stadio e l'altro. I registri PC e IR, sappiamo che devono essere abilitati nel generatore di indirizzi solamente quando vi è bisogno di una scrittura. Di conseguenza, nel **percorso dati**, i segnali di controllo forniscono l'ingresso di selezione di *MuxB*, *MuxY*, *MuxC*. Inoltre, nella **sezione di prelievo**, da l'input di selezione per *MuxMA*, che ricordiamo cambia tra lo stadio 1 e 4, e, nel generatore di indirizzi forniscono l'ingresso di selezione per *MuxINC* e *MuxPC*. Nello stadio 3, invece, i segnali di controllo hanno il compito di selezionare l'operazione da far eseguire all'**ALU**. Vediamo effettivamente come si sviluppa, all'interno del percorso dati, la gestione dei segnali di controllo:

- Inizialmente vengono inviati **3 indirizzi al banco registri**, di cui 2 sono codificati in 5 bit, ovvero gli indirizzi collegati in posizione fissa con l'*indirizzo A* e l'*indirizzo B*, e colleghiamo all'*indirizzo C* a un multiplexer a 3 ingressi, controllato da un segnale di controllo, chiamato **C_SELETTORE**, che ha il compito di decodificare nel registro IR la posizione del registro destinazione, negli ingressi "0" e "1", e nell'ingresso "2" scriviamo, nel caso di chiamata a sottoprogramma il *link register*. Inoltre **C_SELETTORE** ha "lunghezza" di 2 bit, poichè abbiamo 3 ingressi di selezione.
- Abbiamo un segnale di controllo, chiamato **B_SELETTORE**, che seleziona l'ingresso di *MuxB*, ovvero seleziona l'input da mandare all'**ALU**,

nel caso sia un registro sorgente (ingresso 0 di MuxB) o un valore immediato codificato nell'istruzione (ingresso 1 di MuxB). In questo caso B_SELETTORE ha "lunghezza" di 1 bit, poiché abbiamo solo due ingressi di selezione.

- Abbiamo il segnale di controllo **RF_scrittura**, che abilita la scrittura sul registro nello stadio 5;
- Abbiamo il segnale di controllo **ALU_op**, è un segnale di controllo che specifica l'operazione che deve eseguire l'ALU, in questo caso la lunghezza in bit sarà k , tale al numero di operazioni possibili. Inoltre una volta che l'ALU ha completato la sua operazione, essa emette **segnali di condizione** collegati ai bit di esito, che variano in base al risultato dell'operazione.

Quindi, vediamo come nel percorso dati, i segnali di controllo interagiscono nei vari stadi, attraverso quest'immagine:

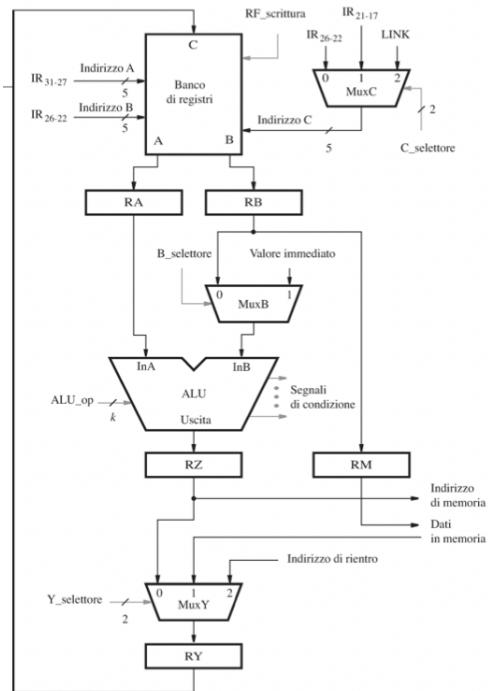


Figure 4.7: Segnali di controllo

4.5.1 Segnali di controllo per l'interfaccia processore-memoria

Abbiamo parlato di segnali di controllo concernenti il percorso dati, e approfondendo la parte di **interfaccia tra processore e memoria**, vi sono altri

segnali di controllo che regolano appunto lo scambio di dati tra processore e memoria:

- In ingresso alla memoria vi sono i segnali **MEM.LETTURA** e **MEM.SCRITTURA**, che, rispettivamente, abilitano operazioni di *load*, ovvero di lettura dalla memoria, e di *store*, ovvero di scrittura in memoria;
- Vi è il segnale **MFC** (*memory function completed*) che è il segnale, pilotato dalla memoria, che indica che l'operazione da parte della memoria è terminata;
- E' presente un **collegamento tra la memoria e il registro IR** attraverso il segnale di controllo **IR_abilita**, che abilita la scrittura dell'istruzione nel registro IR;
- Quando dobbiamo selezionare l'ingresso di *MuxMa*, ovvero nel caso in cui l'indirizzo che mandiamo da leggere in memoria, sia l'indirizzo della prossima istruzione (PC), quindi nel primo ciclo di clock; oppure, nel caso di un'istruzione di salto, l'indirizzo contenuto nel registro inerstadio RZ;
- Nel blocco *"immediato"* dobbiamo scegliere il tipo di estensione del valore immediato. Ciò avviene tramite il segnale di controllo **EXTEND**

4.5.2 Segnali di controllo al generatore di indirizzi

Anche nel generatore di indirizzi abbiamo di segnali di controllo:

- Il multiplexer *MuxPC*, ad esempio, ha un segnale di controllo **PC_SELETTORE**, che seleziona l'ingresso di controllo 0, nel caso quindi il prossimo indirizzo provenga da RA, quindi una *chiamata a sottoprogramma*, oppure l'ingresso 1, se dobbiamo semplicemente incrementare PC;
- Per sapere di quanto incrementare PC, per questo esiste il segnale **INC_SELETTORE**: infatti alla fine dello stadio 1, l'ingresso di controllo sarà sempre 0, poichè la prossima istruzione si trova sempre nella parola di memoria successiva (4 byte); nel caso in cui, nello stadio 4, ci sia un'istruzione di salto, il segnale di controllo **PCABILITA**, che fa sì che l'incremento sia dato dall'indirizzo fornito dell'alu tramite spiazzamento. Quindi in base all'istruzione i segnali **PC_SELETTORE** e **INC_SELETTORE** si comportano in maniera diversa ad ogni stadio.

4.6 Controllo di tipo cablato

I **circuiti di controllo** devono generare segnali di controllo una volta che l'istruzione è stata decodificata. Ad oggi esistono 2 tecnologie di realizzazione di tali circuiti:

- **Controllo cablato**;

- **Controllo micropogrammato.**

In questa sezione ci occuperemo del controllo cablato, utilizzato nei processori RISC, e, nella sezione successiva, parlando di tecnologie di segnali di controllo per i processori CISC, parleremo del controllo micropogrammato. Per quanto riguarda, quindi, il controllo cablato, si tratta di una **rete combinatoria** apposita che genera segnali di controllo **in base all'input** che abbiamo. Tali input sono:

- Dobbiamo sapere il **passo** che si sta eseguendo, ovvero lo stadio o ciclo di clock, così da generare i segnali utili per gli specifici stadi (*B_SELETTORE*, *CSELETTORE*...);
- Dobbiamo sapere l'**istruzione da eseguire** attraverso l'*op code*, per generare segnali utili all'esecuzione dell'istruzione stessa (*MEMLETTURA*, *MEMSCRITTURA*);
- Ci servono i **bit di esito** ad esempio per le istruzioni di salto, e, quindi, per eseguire un *confronto*;
- Ci serve mandare al circuito di controllo **segnali che provengono dall'esterno**, come *interruzioni* o *MFC*.

4.6.1 Generazione di segnali di controllo

Una volta compresi gli input che servono al circuito di controllo, vediamo *come è effettivamente composto tale circuito, e in che modo riesce a generare segnali di controllo*. Essenzialmente i circuiti di controllo sono composti da 2 parti:

1. **Decodificatore di istruzioni;**
2. **Generatore di segnali di controllo;**

Per comprendere al meglio come lavora il circuito, analizziamo tale immagine:

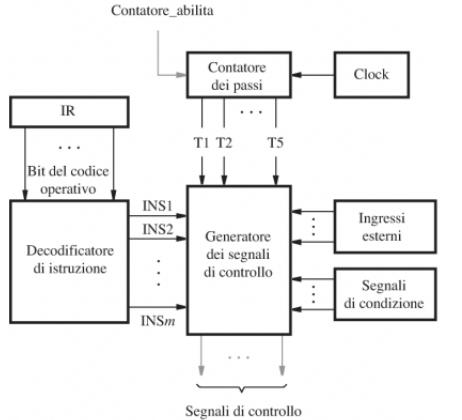


Figure 4.8: Circuito di controllo

Analizzando l'immagine 4.8 possiamo comprendere come avviene il processo di generazione di segnali di controllo:

- Il registro IR viene collegato al **decodificatore**, che si prenderà i bit necessari per prelevare l'*op code*. Una volta decodificata l'istruzione attraverso i bit dell'*op code*, in base all'istruzione decodificata manderà un segnale alto nelle linee INS_1, \dots, INS_n che indicano la **tipologia di istruzione**;
- Il clock manderà il suo segnale al **contatore di passi** che decodificherà, in base al segnale di clock inviato, ovvero in base al *fronte di salita* nel quale si trova, il passo che si sta eseguendo attualmente. Bisogna però fare un importante considerazione su questo passo: infatti nella sezione 4.4.4 abbiamo parlato del fatto che quando lavoriamo con la memoria nei passi 1 e 4, tali passi dureranno più di 1 ciclo di clock, per cui in tal caso si dovrà tener conto del segnale *MFC* che indica che l'operazione da parte della memoria è stata completata. Tale segnale, insieme al segnale di clock, vengono collegati al segnale di controllo **CONTATORE_ABILITA**, che dirà al contatore di passare al passo successivo se e solo se il passo precedente è stato completato (segnale *MFC* alto), ed è stato ricevuto il segnale di clock successivo;
- Il **generatore** ha bisogno di sapere semplicemente il passo in cui siamo, in modo tale da abilitare i segnali di controllo adatti attraverso la funzione logica degli appositi segnali creata dalla rete combinatoria.

4.6.2 Segnali di controllo nel percorso dati

Facciamo un esempio della generazione del segnale *RF_SCRITTURA*, così da capire come vengono generati i vari segnali. In tal caso *RF_SCRITTURA* ci serve quando dobbiamo **scrivere un dato nel registro destinazione**, di

conseguenza il segnale viene posto a 1 solo nel *passo 5*, e, bisogna elencare tutte le istruzioni nel quale ci serve che vada alto nel passo 5 RF_SCRITTURA. Scriviamo dunque questa funzione logica:

$$RF_SCRITTURA = T5 \cdot (ALU + LOAD + CALL) \quad (4.1)$$

Quindi il segnale *RF_SCRITTURA* è uguale all'*and logico* tra il passo 5 (*T5*) e l'*or logico* di tutte le operazioni che hanno bisogno del registro destinazione (ALU: operazioni aritmetico-logiche, Load: operazione di caricamento dalla memoria, Call: chiamata a sottoprogramma). Tale funzione verrà poi **sintetizzata in porte logiche** e, una volta generato il segnale, esso esce dal generatore di segnali. Facendo ulteriori esempi, possiamo dire che:

- *PC_ABILITA* serve nello stadio 3 per le istruzioni di salto e call;
- Vi sono dei segnali che **non dipendono dal passo di esecuzione**. Ad esempio *B_SELETTORE* si può scrivere come:

$$B_SELETTORE = immediato \quad (4.2)$$

infatti *B_SELETTORE* vale 1 per tutte le istruzioni che hanno un valore immediato in IR.

4.6.3 Ritardi della memoria

Abbiamo già accenato spesso al fatto che quando lavoriamo con la memoria dobbiamo capire se passare al passo successivo o meno. In tal caso il contatore di passi deve essere abilitato o meno in base a quanto detto prima, quindi scriveremo la funzione logica di *CONTATORE_ABILITA* come:

$$CONTATORE_ABILITA = NOT(WMFC) + MFC \quad (4.3)$$

nel quale *WMFC* è il segnale generato dal processore che indica di aspettare che *MFC* sia alto. Inoltre anche quando ad esempio utilizziamo *PC_ABILITA* lavoriamo con la memoria, quindi scriveremo la sua funzione come:

$$T1 \cdot MFC + T3 \cdot BR \quad (4.4)$$

nel quale *BR* indica un istruzione di *branch*, cioè un salto.

4.7 Processori CISC

I processori CISC hanno un'organizzazione hardware più complessa, poiché:

- Hanno **modi di indirizzamento** e **op code** sicuramente molto vari rispetto a un processore RISC;
- E' possibile fare **operazioni con operandi in memoria**;
- Le istruzioni hanno **lunghezza variabile**, che non sempre è una parola di memoria.

4.7.1 Organizzazione CISC

Parlando dell'*hardware* di un processore CISC, esso è sostanzialmente composto da queste componenti:

- **Banco di registri**, come nei processori RISC;
- **ALU**, come nei processori RISC;
- **Registri temporanei**, che sostituiscono i *registri interstadio* nei processori RISC;
- Blocco **interconnessione**, che è una componente fondamentale che **collega** tutte le componenti tra loro.

Vediamo ad esempio nella seguente immagine, come è composta l'architettura di un processore CISC:

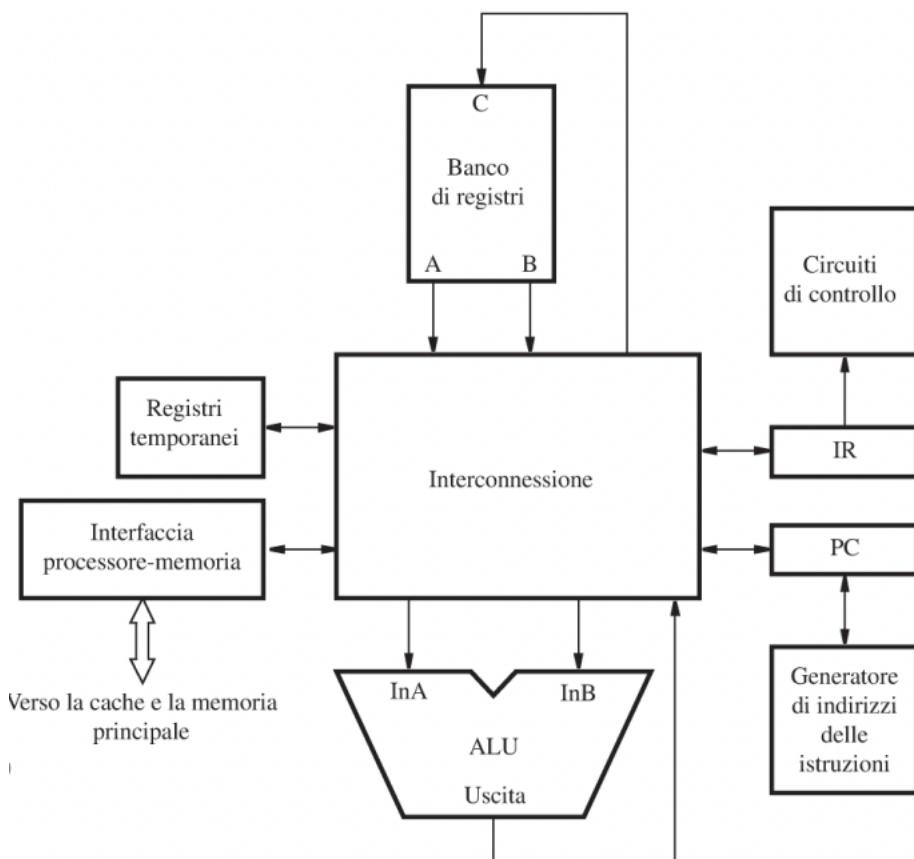


Figure 4.9: Architettura processore CISC

Il blocco **interconnessione**, quindi, permette di **trasferire dati in maniera bidirezionale** per quasi tutti i componenti del processore. Come abbiamo potuto comprendere, quindi, il blocco *interconnessione* serve per i collegamenti, ed è infatti formato da **bus**, cioè un insieme di **linee di collegamento** che riescono a collegare qualsiasi unità del processore.

4.7.2 Controllo di accesso al bus

Per quanto riguarda il *bus* sorge un problema: infatti quando ho più dispositivi, come nel caso dei CISC, collegati alla linea del bus, tutti potrebbero impostare un valore nel bus, creando confusione tra i vari segnali, e provocando un possibile cortocircuito. Per evitare tale problema si utilizza una **porta a 3 stati** che serve a pilotare opportunamente la linea del bus: tale porta è **collegata all'uscita di un flip-flop di tipo D**, in modo tale che quando l'uscita della porta vale "1", il segnale può essere impostato sul bus; altrimenti, quando vale "0", il segnale non viene imposto. Per quanto riguarda l'**ingresso del flip-flop**, viene utilizzato un **multiplexer**, per il quale quando il suo segnale in ingresso vale 1, si deve prendere qualcosa in ingresso, mentre se vale 0, il flip-flop mantiene il suo valore corrente. Se denotiamo il multiplexer come R_{in} e la porta a 3 stati come R_{out} , il controllo sul bus avviene come nella seguente immagine:

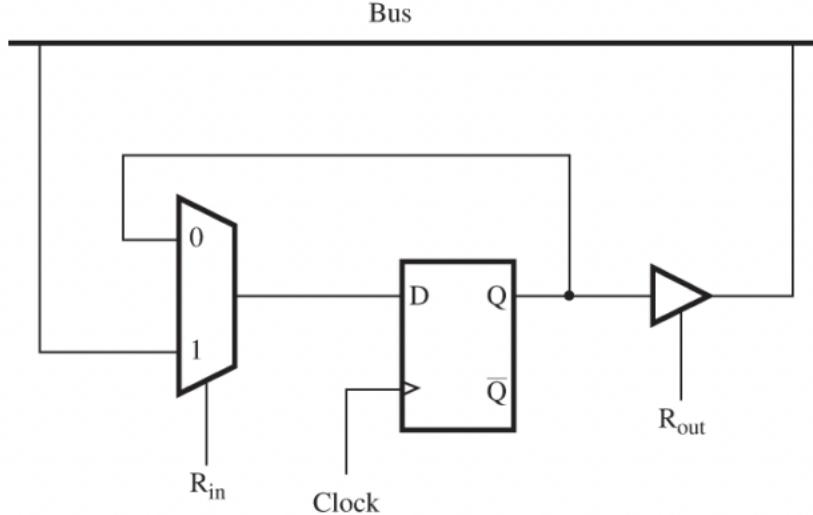


Figure 4.10: Controllo di accessi al bus

4.7.3 Interconnessione a tre bus

Internamente al processore, abbiamo compreso che esiste un *blocco interconnessione* che collega tutti i blocchi funzionali tra loro. Tale blocco è formato da **tre bus**, come possiamo vedere nella seguente immagine:

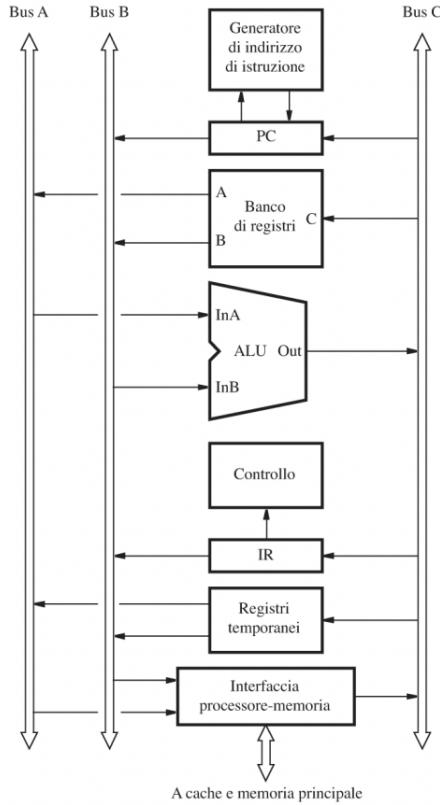


Figure 4.11: Interconnessione a 3 bus

Esempio:istruzione ADD Vediamo adesso di fare un esempio di come i passi vengono eseguiti in un processore CISC, considerando l'istruzione:

ADD R5, R6

il quale scopo è di sommare i contenuti di R5 e R6, e conservare la somma in R5. L'istruzione verrà eseguita in **3 passi**, ovvero **3 cicli di clock**:

1. **Prelievo dell'istruzione:** una volta che l'istruzione è contenuta in PC, viene prelevato l'indirizzo contenuto in PC e viene immesso nel **bus B**, che invia tale dato all'*interfaccia processore-memoria*. Tale interfaccia restituirà l'istruzione al **bus C**, che la da in ingresso al *registro IR*. In tal caso stiamo considerando che il processo avvenga in 1 ciclo di clock, utilizzando quindi una **memoria veloce**; ma con una memoria più lenta tale passo potrebbe durare più di un ciclo di clock.
2. **Decodifica istruzione:** prima che finisca il *secondo ciclo di clock*, e, quindi, una volta che sono stati generati i giusti **segnali di controllo**, usciranno dal banco registri i due registri R5 ed R6, che saranno immessi

rispettivamente nel bus A e nel bus B, e infine da tali bus arriveranno come *ingressi dell'ALU*.

3. **ALU**: nel passo 3 l'ALU effettuerà la somma tramite il segnale di controllo apposito, e il risultato della somma verrà inviato nel bus C, che manderà il dato in ingresso al banco registri all'indirizzo C per poi essere scritto in R5.

Possiamo dire che, per i processori CISC, questi **primi 3 passi sono uguali per ogni istruzione**.

Esempio: istruzione con operando in memoria Consideriamo adesso la seguente istruzione CISC:

AND X(R7), R9

Il nostro scopo in questo caso è quello di fare l'and logico tra il dato presente in memoria in $X + [R7]$ e R9, e mettere il risultato in $X + [R7]$. Stiamo assumendo, inoltre, che il *valore immediato* X sia contenuto in una **seconda parola di memoria** da 32 bit. Tale istruzione viene eseguita in **7 passi**¹:

1. **Prelievo istruzione**: come abbiamo visto in precedenza, viene trasferito il contenuto di PC nel bus B, che viene inviato all'interfaccia processore-memoria. L'interfaccia fornirà poi l'istruzione al bus C che la inserirà nel registro IR. Inoltre PC viene incrementato di 4 per leggere la prossima parola di memoria (valore immediato X). Ovviamente tutto ciò avviene se la memoria completa in un ciclo di clock.
2. **Decodifica istruzione**: vengono generati i segnali di controllo, vengono letti i registri R7 ed R9, e viene letta la prima parola di memoria
3. Ricordiamo dalla premessa che l'istruzione è contenuta in 2 parole di memoria poiché abbiamo il valore immediato X. Per cui nel passo 3 **leggiamo il valore immediato** e lo mettiamo in un **registro temporaneo** che chiamiamo **TEMP 1**. Alla fine del passo 3 incrementiamo di nuovo PC di 4.
4. Calcolo **indirizzo effettivo** X(R7). In tal caso viene mandato un segnale di controllo per leggere R7 e leggere X, inviare i dati al bus B, e tale bus invierà tali dati come ingressi dell'ALU. L'ALU fa la somma $[R7 + X]$ e invia il contenuto (**indirizzo**) in un altro registro temporaneo chiamato **TEMP 2**:

$[TEMP\ 2] \leftarrow [TEMP\ 1] + [R7]$

¹Notiamo che nel processore CISC il numero di passi per ogni istruzione è **variabile**, a differenza del processore RISC dove ogni istruzione viene eseguita in 5 passi

5. Leggiamo il **dato presente nella locazione di memoria $R7 + X$** . Quindi esce l'indirizzo contenuto in TEMP 2 nel bus B che invia il dato all'interfaccia processore-memoria. L'interfaccia restituirà il dato presente in quella locazione che verrà salvato in TEMP 1.
6. Usciamo dal banco registri R9, che viene inviato come ingresso all'ALU e viene **calcolato l'AND**. Il risultato viene salvato in TEMP 1
7. Infine, all'ultimo passo, viene preso il risultato in TEMP 1, l'indirizzo in TEMP 2, e vengono inviati all'interfaccia processore-memoria, che **scriverrà il risultato in memoria** all'indirizzo effettivo.

4.7.4 Controllo microprogrammato

Per generare segnali di controllo all'interno di un processore CISC si utilizza il **controllo microprogrammato** che ha il vantaggio di essere *più flessibile* rispetto al controllo cablato e, inoltre, occorrono meno circuiti hardware per costruirlo. Come premessa possiamo dire che all'interno di un processore CISC ci serve attivare una **sequenza di segnali di controllo** per ogni passo e disattivare quelli che non ci servono: a questo scopo ci aiuta la **microprogrammazione**. In microprogrammazione vi sono questi *concetti di base*:

- Un *insieme di segnali di controllo* viene chiamata **parola di controllo**;
- La parola di controllo impartita ad ogni passo dell'istruzione viene chiamata **microistruzione**;
- La sequenza di microistruzioni utili per l'esecuzione dell'istruzione viene chiamata **microroutine**

Quindi sappiamo che, nei CISC, le prime 2 microistruzioni sono le stesse per ogni istruzione (prelievo e decodifica); alla fine della decodifica sappiamo quali altre microistruzioni ci serviranno per l'esecuzione dell'istruzione. Tale tipo di controllo, quindi, è molto flessibile poiché permette di salvare tutte le microistruzioni per ogni microroutine.

Unità di controllo microprogrammato

Vediamo come è fatta all'interno l'unità di controllo microprogrammato, e come gestiamo i segnali di controllo per l'esecuzione di ogni istruzione²:

- La **memoria di controllo** contiene le micro-routine;
- Le micro-istruzioni sono indirizzate dal PC, aggiornato dal **generatore di micro-indirizzi**;

²simile al meccanismo di ricorsione

- Viene decodificato il codice operativo e la modalità di indirizzamento in [IR], quindi trova il micro-indirizzo della micro-istruzione iniziale della micro-routine che governa l'esecuzione dell'istruzione
- In ciascun passo la micro-routine, attraverso ciascuna micro-istruzione indica i **segnali di controllo attivi**

Controllo cablato vs microprogrammato

Concludendo, vediamo di confrontare i due tipi di controllo:

- Il controllo micro-programmato è **semplice da realizzare** rispetto al cablato ma ha anche il difetto di essere **più lento** del cablato poichè dobbiamo fare diverse "operazioni" all'interno del ciclo di clock;
- In tempi più recenti il controllo cablato è diventata la scelta preferita poichè il costo di realizzazione è diventato superfluo in confronto alla velocità che può offrire tale sistema di controllo

Capitolo 5

INTRODUZIONE AL PIPELINING

Pipelining significa eseguire istruzioni diverse in **parallelo**. Ora, se dovessimo pensare di eseguire diverse istruzioni, senza pipeline, avremmo che:

- La prima istruzione prende 5 cicli di clock;
- La seconda istruzione prende 5 cicli di clock;
- ecc.

Eseguendo, invece, un'istruzione in pipeline abbiamo che a ogni ciclo vengono eseguite **contemporaneamente diverse istruzioni** che si trovano in **stadi diversi di esecuzione**, come possiamo vedere nella seguente immagine:

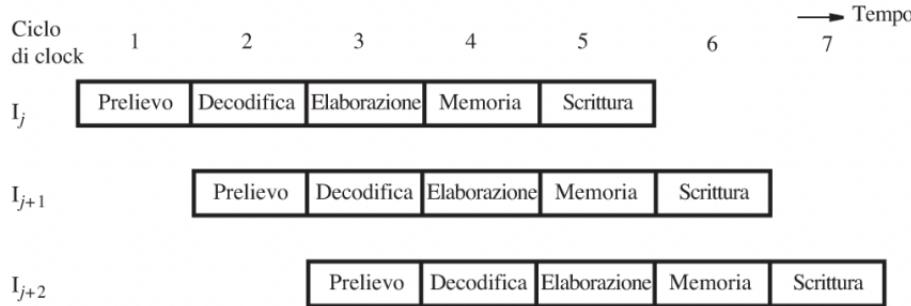


Figure 5.1: Pipeline di 3 istruzioni

Quindi se io avessi eseguito sequenzialmente per 3 istruzioni diverse avremmo dovuto aspettare 15 cicli di clock; mentre in parallelo solo 7. Per l'esecuzione in pipeline ci si ispira alla **catena di montaggio**: infatti all'interno di una catena di montaggio abbiamo prodotti che si trovano in diversi stadi e man mano che

avanzo nella catena essi saranno sempre più completi. Si dice che aumenta il **throughput**, ovvero la quantità di prodotti che escono nell'unità di tempo. Ciò porta a un **tempo di esecuzione costante**, cioè ogni stadio ha un tempo di esecuzione fissato per far scorrere la catena.

5.1 Organizzazione in pipeline

Andando più nello specifico su come viene *organizzato* il *pipelining*. Il **datapath** viene leggermente **modificato**: al posto dei registri interstadio, abbiamo dei **buffer interstadio**, che sono **set di registri**, che contengono non solo i registri interstadio, ma anche una *copia del registro IR e del registro PC* e dei *segnali di controllo* per lo stadio successivo. Si dice infatti che i buffer interstadio **alimentano lo stadio successivo** poichè ciascun **buffer trasferisce informazioni allo stadio successivo**. In particolare sono 4 i buffer interstadio:

- **Buffer B1:** *alimenta* lo stadio di **decodifica**;
- **Buffer B2:** *alimenta* lo stadio di **elaborazione**;
- **Buffer B3:** *alimenta* lo stadio di **memoria**;
- **Buffer B4:** *alimenta* lo stadio di **scrittura**.

5.1.1 Percorso dati nel pipelining

Supponiamo di dover eseguire in *pipeline*, quindi più istruzioni diverse in stadi diversi, la seguente istruzione:

100	ADD R2,R3,#100
101	OR R4,R5,R6
102	SUBTRACT R9,R8,#30

Abbiamo nella colonna di sinistra gli *indirizzi delle istruzioni*, mentre nella colonna di destra le *istruzioni stesse*.

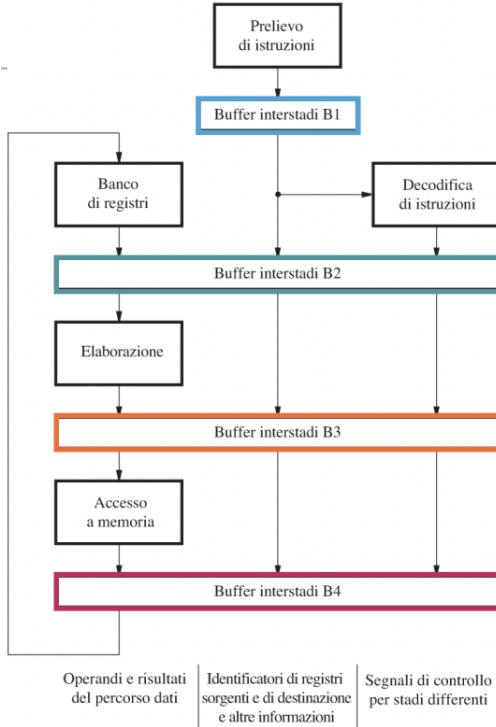


Figure 5.2: Datapath in pipeline

Seguendo l'immagine qui sopra, vediamo ora per i **5 cicli di clock** come avviene l'esecuzione in pipeline delle istruzioni viste prima¹:

- **Primo ciclo di clock:** in questo primo ciclo viene semplicemente **prelevata la prima istruzione** (Add), e in questo caso abbiamo, quindi, che per quanto riguarda i **buffer interstadio**:
 - **B1** contiene l'*indirizzo dell'istruzione successiva in PC* ($PC \leftarrow [101]$), mentre *in IR è presente l'istruzione Add*.
- **Secondo ciclo di clock:** in questo stadio viene **prelevata la seconda istruzione** (Or), e viene **decodificata la prima istruzione**. Per quanto riguarda i **buffer interstadio**, abbiamo che B1, che ricordiamo alimentava la decodifica, *trasferisce i suoi dati a B2*, di conseguenza:
 - **B1** contiene l'*indirizzo dell'istruzione successiva in PC* ($PC \leftarrow [102]$), mentre *in IR è presente l'istruzione Or*;
 - **B2** contiene l'*indirizzo presente in PC nel primo ciclo di clock* ($PC \leftarrow [101]$), *in IR è presente l'istruzione Add*, ovvero quella

¹Tale esempio vale comunque per tutti i tipi di istruzioni

presente nel primo ciclo di clock, e *il registro interstadio RA, uscita del banco registri, contiene il valore del registro R3.*

Quindi notiamo che il buffer B1 del primo ciclo di clock, ovvero della prima istruzione, ha alimentato lo stadio di decodifica della prima istruzione nel secondo ciclo di clock. Ovviamente B2 conterrà anche il valore nei registri interstadio.

- **Terzo ciclo di clock:** in questo stadio viene prelevata la terza istruzione (Subtract), viene decodificata la seconda istruzione (Or) e viene elaborata (ALU) la terza istruzione (Add). Quindi, per i ragionamenti fatti precedentemente, i *buffer interstadio* saranno composti nel seguente modo:
 - **B1** contiene *in PC l'indirizzo 103* ($PC \leftarrow [103]$), *in IR è presente la Subtract*;
 - **B2** contiene i dati trasferiti da B1 dal precedente ciclo di clock, ovvero *in PC vi è l'indirizzo 102*, ($PC \leftarrow [102]$), *in IR vi è l'istruzione Or*, e abbiamo che *i registri interstadio RA ed RB, uscite del banco registri, contengono rispettivamente i contenuti di R5 ed R6*, ovvero i registri sorgente dell'istruzione Or.
 - **B3** contiene i dati trasferiti da B2 del precedente ciclo di clock, ovvero *in PC vi è l'indirizzo 101*, ($PC \leftarrow [101]$), *in IR vi è l'istruzione Add*, e *il registro interstadio RZ, uscita dell'ALU, contiene il valore della somma* $[R3] + 100$
- **Quarto ciclo di clock:** in questo stadio viene prelevata la quarta istruzione (non prevista), viene decodificata la terza istruzione (Subtract), viene elaborata (ALU) la seconda istruzione (Or) e avviene l'accesso in memoria per la prima istruzione (Add). Quindi i *buffer interstadio* saranno composti in questo modo:
 - **B1** contiene *in PC l'indirizzo 104* ($PC \leftarrow [104]$), *in IR è presente la prossima istruzione*;
 - **B2** contiene i dati trasferiti da B1 dal precedente ciclo di clock, ovvero *in PC vi è l'indirizzo 103*, ($PC \leftarrow [103]$), *in IR vi è l'istruzione Subtract*, e abbiamo che *il registro interstadio RA, uscita del banco registri, contiene il contenuto di R2*, ovvero i registro destinazione dell'istruzione Add.
 - **B3** contiene i dati trasferiti da B2 del precedente ciclo di clock, ovvero *in PC vi è l'indirizzo 102*, ($PC \leftarrow [102]$), *in IR vi è l'istruzione Or*, e *il registro interstadio RZ, uscita dell'ALU, contiene il valore dell'or* $[R5] OR [R6]$;
 - **B4** contiene i dati trasferiti da B3 del precedente ciclo di clock, ovvero *in PC vi è l'indirizzo 101*, ($PC \leftarrow [101]$), *in IR vi*

è l'istruzione Add, e nel registro interstadio RY, uscita di MuxY, vi è la somma R3 + [100];

- Quinto ciclo di clock: prelievo istruzione 5, decodifica istruzione 4, elaborazione istruzione 3, memoria istruzione 2, scrittura istruzione 1. Di conseguenza i *buffer interstadio* saranno composti in questo modo:
 - **B1** contiene *in PC l'indirizzo 105* ($PC \leftarrow [105]$), *in IR è presente la prossima istruzione*;
 - **B2** contiene i dati trasferiti da B1 dal precedente ciclo di clock, ovvero *in PC vi è l'indirizzo 104*, ($PC \leftarrow [104]$), *in IR vi è l'istruzione prossima alla Subtract*, e abbiamo che *il registro interstadio RA, uscita del banco registri, contiene il contenuto dell'uscita dei registri della prossima istruzione*.
 - **B3** contiene i dati trasferiti da B2 del precedente ciclo di clock, ovvero *in PC vi è l'indirizzo 103*, ($PC \leftarrow [103]$), *in IR vi è l'istruzione Subtract*, e *il registro interstadio RZ, uscita dell'ALU, contiene il valore della Subtract* [$R8] - 30$;
 - **B4** contiene i dati trasferiti da B3 del precedente ciclo di clock, ovvero *in PC vi è l'indirizzo 102*, ($PC \leftarrow [102]$), *in IR vi è l'istruzione Or*, e nel *registro interstadio RY, uscita di MuxY, vi è l'or R5 OR R6*;
 - In [R2] infine come stadio di **scrittura**, vengono passati i valori di B4 dal precedente ciclo di clock, e quindi viene scritto il valore della somma in R2: $[R2] \leftarrow [R3] + 100$.

5.2 Problematiche nel pipelining

Gli esempi di pipelining visti in precedenza sono stati concepiti per *situazioni ideali* nella quale vi è *una perfetta sovrapposizione delle istruzioni*; in pratica non sempre è così, ma anzi possono venire a crearsi delle problematiche. Supponiamo infatti di avere un'istruzione i_{j+1} , successiva alla i_j , tale che i_{j+1} abbia come *registro sorgente* il *registro destinazione* di i_j , del tipo:

```
Add R2, R3, #100 // R2 è il risultato della prima istruzione
Subtract R9, R2, #30 // R2 è un operando della seconda istruzione
```

In tal caso succede che fino a quando la prima istruzione non completa la **scrittura di R2**, la seconda istruzione rimarrà in **stallo**. In tal caso si dice che vi è un **conflitto**, o *hazard*, che è *una qualsiasi situazione che causa un rallentamento di esecuzione*, ovvero uno **stallo**. Nell'esempio delle due istruzioni precedenti il conflitto è causato da una **dipendenza di dato**, ma vi sono altri conflitti causati da **ritardi in memoria**, **istruzioni di salto** o **limiti di risorse**. Andremo in questa sezione ad analizzare tutte queste problematiche e come vengono risolti per far eseguire correttamente più istruzioni in pipeline.

5.2.1 Dipendenza di dato

Riprendiamo l'esempio visto in precedenza:

```
Add R2, R3, #100 // R2 è il risultato della prima istruzione
Subtract R9, R2, #30 // R2 è un operando della seconda istruzione
```

In tal caso notiamo un **conflitto di dato** per il quale la decodifica della *Subtract*, seconda istruzione, può avvenire solamente una volta che è stato scritto il risultato della *Add* nel registro R2, poiché questo è registro destinazione della *Add* e registro sorgente della *Subtract*. Per queste motivazioni il conflitto è causato da una **dipendenza di dato**. I modi per risolverla sono i seguenti:

- Possiamo, attraverso dei **segnali di controllo**, far sì che si generino *cicli di attesa*. Esattamente 3, come possiamo vedere dalla seguente immagine:

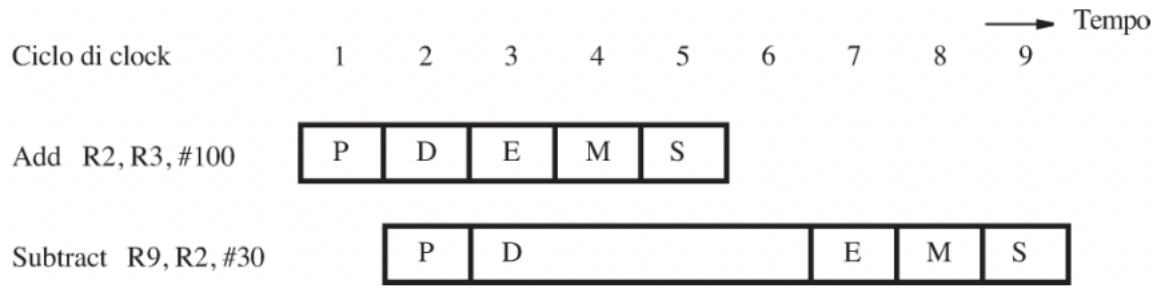


Figure 5.3: Cicli di attesa

- Si possono produrre questi 3 cicli di attesa attraverso **NOP**, ovvero *no operation*, in modo tale da rallentare l'esecuzione fino a quando l'istruzione *Add* ha completato lo stadio di scrittura. Tale metodo viene chiamato **bolle**, o *bubble*, poiché è come se facessimo uscire *bolle virtuali* per visualizzare che in quei cicli di clock non è stato fatto niente per la *Subtract*.
- L'ultimo metodo, che è sicuramente quello più efficiente, è quello dell'**inoltro degli operandi**, che spieghiamo più nel dettaglio nel paragrafo a seguire.

Inoltro degli operandi Come abbiamo già accennato, il metodo più efficace ed efficiente per risolvere il conflitto di dato è quello dell'**inoltro degli operandi**. Partiamo da una piccola considerazione: quando abbiamo completato il *terzo ciclo di clock*, avremo il valore della somma nel registro interstadio RZ. Se avessimo quindi la possibilità di inoltrare l'uscita dell'ALU della prima istruzione come ingresso dell'ALU della prossima istruzione, anziché prelevarlo da R2 aspettando per 3 cicli di clock, riusciamo a procedere nel pipelining senza rallentamenti. Per ottenere questa situazione, però, è necessario **modificare**

il percorso dati in funzione dell'inoltro operandi, come vediamo nella seguente immagine:

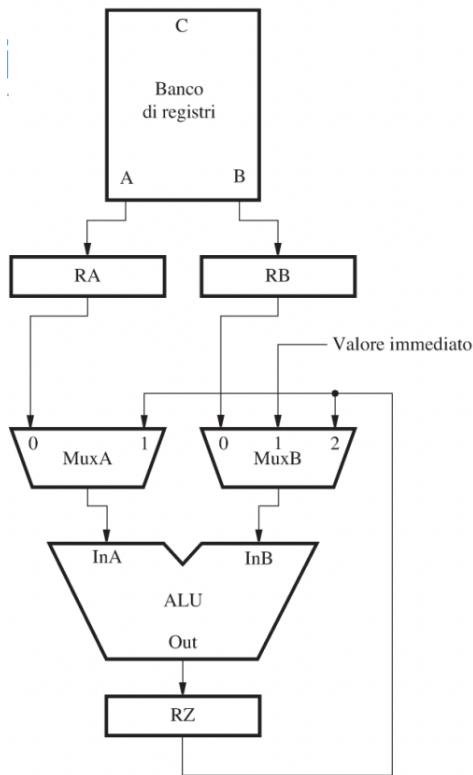


Figure 5.4: Percorso dati per inoltro operandi

Notiamo quindi che il percorso dati viene modificato in modo tale che l'uscita dell'ALU, ovvero RZ, viene collegata agli ingressi dell'ALU. Inoltre è necessario inserire **MuxA** che seleziona l'input dell'ALU tra il registro RA ed RZ. L'inoltro su InA può essere esteso a RY per risolvere dipendenze come nella sequenza:

```

Add R2, R3, #100
Or R4, R5, R6
Subtract R9, R2, #30
  
```

In tal caso verrà aggiunto sia in MuxA che in MuxB un ingresso al fine di farvi arrivare RY. Ciò che quindi avviene *semanticamente* per le istruzioni viste in precedenza è la seguente:

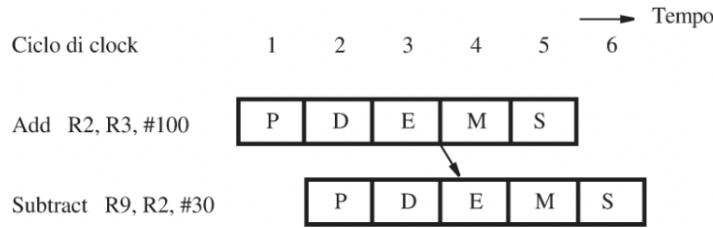


Figure 5.5: Semantica inoltro operandi

5.2.2 Ritardi della memoria

Quando l'istruzione si trova nello **stadio memoria**, questo, come abbiamo già detto, per terminare la sua esecuzione sarà mediamente molto più lento rispetto agli altri stadi di esecuzione. Abbiamo infatti due possibilità:

- Se il dato *si trova in cache*, allora lo stadio può durare 1 ciclo di clock come gli altri;
- Se il dato *non si trova in cache*, allora ciò provoca una **cache miss**, che è uno **stallo**, provocando quindi un *conflitto*.

Supponiamo di dover eseguire l'istruzione:

Load R2, (R3)

Supponiamo inoltre che ci sia una **cache miss** la locazione di memoria (R3). In questo caso verranno introdotti per completare lo stadio memoria due cicli di clock aggiuntivi, fino a quando ovviamente la memoria completa la sua istruzione e ciò provoca un rallentamento di tutte le istruzioni successive a questa:

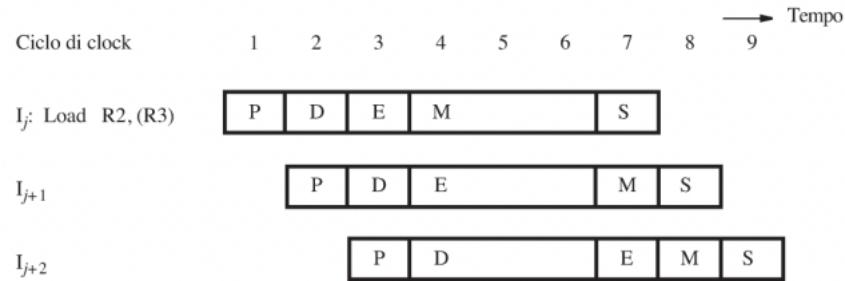


Figure 5.6: Ritardo della memoria

Supponiamo di avere la sequenza di istruzioni:

Load R2, (R3)
Subtract R9, R2, #30

La **dipendenza da un dato letto dalla memoria**, come in questo caso, anche se trovato in cache (locazione (R3) in cache), provoca uno stallo (bolla) se l'istruzione successiva usa il dato appena letto, poiché la scrittura di RY in B4 avviene al ciclo 4, e la scrittura in R2 al ciclo 5. La Subtract deve restare in stallo per un ciclo per permettere di inoltrare RY all'ingresso dell'ALU nel ciclo 5:

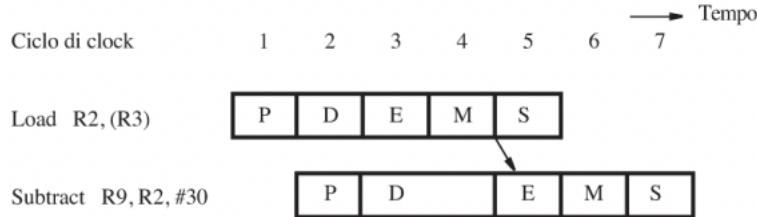


Figure 5.7: Caption

5.2.3 Ritardi nei salti

Un'altra problematica che è possibile riscontrare con il pipelining è quella legata alle **istruzioni di salto**. In un'istruzione di salto, infatti, la *sequenza delle istruzioni viene alterata*, cioè "saltiamo" da un'altra parte del programma.

Salti incondizionati

Supponiamo adesso di avere un'istruzione i_j che deve *saltare* in un'altra istruzione i_k . Inizialmente non sappiamo che i_j sia un'istruzione di salto, ma lo sappiamo dopo che l'istruzione è stata **decodificata** ed è stato **incrementato PC**. Con la pipeline nel frattempo, però, sono state già caricate altre 2 istruzioni i_{j+1}, i_{j+2} che dovranno essere **scartate**. Ciò significa che *abbiamo sprecato 2 cicli di clock* che vengono detti **penalità di salto**. Se pensiamo che ogni volta che incontriamo un salto abbiamo una *penalità* di 2 cicli di clock, e se pensiamo che con il **conteggio dinamico**² delle istruzioni, statisticamente il salto è utilizzato circa il 20% delle volte all'interno dei nostri programmi; capiamo che il tempo di esecuzione potrebbe aumentare del 40% e ciò rallenterebbe troppo i nostri programmi. La soluzione che si attua è quella che, attraverso *apposito hardware*, si **anticipa il calcolo della destinazione di salto** al ciclo 2 anziché al ciclo 3, ovvero:

- Al *primo ciclo di clock* si carica l'istruzione i_j ;
- Al *secondo ciclo di clock* si carica l'istruzione i_{j+1} e si calcola la destinazione di salto di i_j ;
- Al *terzo ciclo di clock* avviene il **salto**

²Si tratta di un conteggio che si basa sulle istruzioni che vengono scritte all'interno di un programma, per poi avere una *statistica* basata su migliaia di conteggi.

In tal modo riusciamo a guadagnare una *penalità di salto* di solamente **un ciclo di clock**

Salti condizionati

Per quanto riguarda le istruzioni di **salto condizionato**, come ad esempio:

```
Branch_if_[R5]=[R6]    CICLO
```

possiamo sapere se saltare o meno soltanto alla fine del *terzo ciclo di clock* che ci dice ***se la condizione è soddisfatta***. Ciò che facciamo in questo caso è, sempre attraverso *apposito hardware*, **anticipare il calcolo della destinazione di salto al secondo ciclo di clock**. Possono essere fatti dei miglioramenti aggiuntivi, però, per le istruzioni di salto. Supponiamo infatti di avere tali istruzioni:

```
ADD R7,R8,R9
Branch_if_[R3]=0 DESTINAZIONE
i_j+1
...
...
DESTINAZIONE i_k
```

Sapendo che in i_{j+1} , che è il **posto del ritardo del salto**, ci sarà una penalità di salto di un ciclo di clock, quale istruzione utile potremmo **sostituire** per fare in modo di non avere penalità di salto? Dovrò scegliere un'istruzione che si trovi ***prima del salto*** e che ***non abbia dipendenza di dato***. Nell'istruzione precedente la ADD viene eseguita indipendentemente dal salto poichè veniva prima di quest'ultimo e inoltre non ha dipendenza di dato; per cui il *compilatore riordinerà le istruzioni* in modo tale che non ci sia nessuna penalità di salto, quindi avremo:

```
Branch_if_[R3]=0 DESTINAZIONE
ADD R7,R8,R9
i_j+1
...
...
DESTINAZIONE i_k
```

Quando il compilatore funziona in questo modo sa che deve eseguire sempre l'istruzione dopo il salto e, nel caso in cui nessuna istruzione può essere inserita verrà inserita una NOP, in modo tale da evitare la penalità.

Predizione di salti

La decisione di salto, per le istruzioni di salto condizionato, abbiamo capito che viene presa al secondo ciclo di clock, e potrebbe capitare magari che l'istruzione

successiva a quella di salto venga scartata. Per evitare questa situazione, si effettua una **predizione statica o dinamica** al *primo ciclo di clock*, in modo tale da ridurre ancora la penalità di salto. Andiamo adesso a descrivere come funzionano entrambi i tipi di predizione.

Predizione statica Banalmente possiamo dire che *staticamente* la probabilità di indovinare se il salto venga effettuato o meno è del 50%. Se andiamo però più nello specifico nel salto condizionato, allora possiamo avere maggiore accuratezza nella predizione:

- Nel caso in cui la **condizione** è alla **fine del ciclo** (ES: do-while), allora sicuramente vorremo saltare all'indietro più di una volta, quindi vogliamo fare più di una passata. In tal caso la predizione si basa sullo **spiaz-zamento negativo**, attraverso apposito hardware, è ci dice che è più probabile che si salti.
- Nel caso in cui la **condizione** è all'**inizio del ciclo** (ES:while), allora non dobbiamo saltare per i salti in avanti, ma spesso entriamo per fare più di una passata. In questo caso la predizione con **spiazzamento positivo** ci dice che è più probabile che non si salti.

Predizione dinamica (2 stati) Per migliorare ancora di più la predizione, si utilizza la **predizione dinamica di salto**, che si basa su un **automa a 2 stati** che sono:

- **PS**: che significa "probabilmente salta";
- **PNS**: che significa "probabilmente non salta".

Questo tipo di predizione si basa sul fatto che il processore decide se saltare o meno in base alla predizione che è stata fatta l'ultima volta, quindi la predizione dinamica è molto utile quando abbiamo un ciclo in cui dobbiamo fare più passate. Ora, possiamo immaginare l'automa a 2 stati in questo modo:

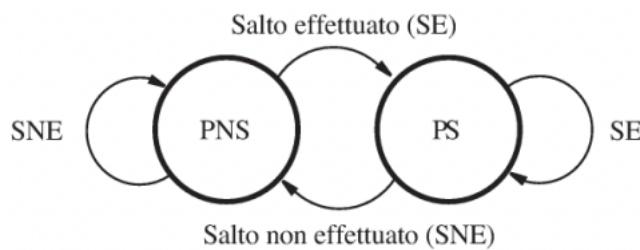


Figure 5.8: Automa a 2 stati

Supponiamo quindi di trovarci nello stato PS, che è la **predizione di salto**: come possiamo notare dall'immagine, accade che se il *salto viene effettuato*, che

possiamo vedere nell'immagine con la notazione "**SE**" (*salto effettuato*), allora la predizione rimane in PS; altrimenti se il salto non viene effettuato, che possiamo vedere nell'immagine con la notazione "**SNE**" (*salto non effettuato*), e quindi la predizione era sbagliata, allora la predizione diventa PNS, e così via in base al risultato dell'ultima passata del ciclo. In tal caso per rappresentare la **storia dell'istruzione** ci basta **1 bit** soltanto.

Predizione dinamica (4 stati) Per migliorare il più possibile l'accuratezza della predizione è stata "creata" un'evoluzione dell'automa a 2 stati, che è appunto un **automa a 4 stati**, rappresentati in **2 bit**, che sono:

- **PS**: che significa "*probabilmente salta*";
- **MPS**: che significa "*molto probabilmente salta*";
- **PNS**: che significa "*probabilmente non salta*";
- **MPNS**: che significa "*molto probabilmente non salta*";

Immaginiamo l'automa a 4 stati come un'evoluzione di quello a 2 stati:

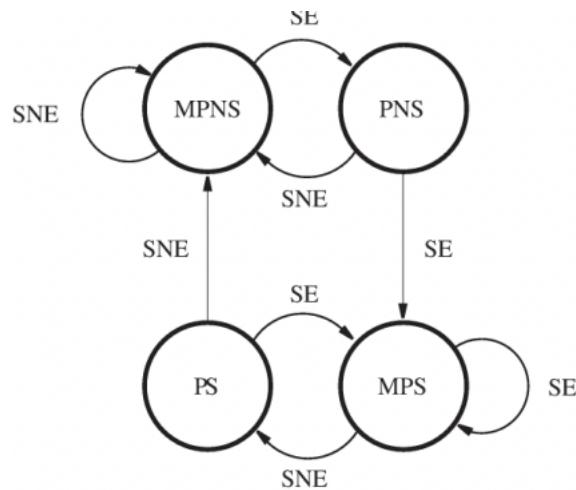


Figure 5.9: Automa a 4 stati

Guardando l'immagine sopra, ci rendiamo già conto che la predizione risulta già molto più veritiera. Supponiamo infatti di trovarci nello stato di predizione **PNS**, come all'*inizio di un ciclo do-while* che ha la condizione alla fine; quindi abbiamo che se il salto viene effettuato (**SE**), abbiamo che la predizione cambia in **MPS**, che fino a quando il salto viene effettuato rimane uguale. Quando il salto non viene più effettuato (**SNE**), perché la condizione non viene più soddisfatta, allora lo stato di predizione passa a **PS**. In questo modo quando si incontrerà lo stesso ciclo, trovandosi lo stato di predizione in **PS** si farà in

maniera corretta almeno la prima passata, nell'esempio di un do-while. Quindi se nella predizione a 2 stati le predizioni rispettivamente della prima e l'ultima passata di ogni ciclo erano sempre sbagliate, in quella a 4 stati riusciamo a essere ancora più precisi.

Buffer di destinazione di salto

Abbiamo visto che riusciamo ad anticipare al *secondo ciclo di clock* la **valutazione della condizione** e il **calcolo dello spiazzamento per il salto**; inoltre, riconosciamo al *primo ciclo di clock* che sia un'istruzione di salto e facciamo la predizione su quest'ultimo. Ora ci possiamo chiedere: *come facciamo nel ciclo di prelievo dell'istruzione a capire se sia un salto e predire se si salta?* Si utilizza per questo un **buffer di destinazione del salto** che è una **tabella** che contiene in ogni record gli **indirizzi delle istruzioni di salto, gli indirizzi della destinazione del salto e i bit di predizione del salto**. Accade quindi che quando *preleviamo l'istruzione* al primo ciclo di clock viene confrontato l'indirizzo contenuto in PC, con quelli contenuti nel *buffer* per capire se sia un'istruzione di salto e, nel ciclo 2, se si tratta di un salto, si utilizzano i bit di predizione del salto per capire la prossima istruzione da prelevare. In questo modo abbiamo eliminato la penalità di salto con un'istruzione utile. Inoltre tali tabelle sono molto piccole e quindi permettono *ricerche veloci* e quindi possiamo dire che sono **memorie veloci**.

5.2.4 Limiti di risorse

Il parallelismo in pipeline si ottiene quando istruzioni diverse accedono a **risorse diverse**. Potrebbero però succedere dei conflitti nell'accesso delle risorse in memoria: ad esempio lo **stadio di prelievo e memoria accedono entrambi alla memoria**. Inoltre contando il fatto che lo stadio di prelievo è presente a ogni ciclo di clock della pipeline e contando che statisticamente il 25% delle istruzioni accedono alla memoria (**load e store**), possiamo avere un conflitto che provoca un rallentamento del tempo di esecuzione del 25%. Per evitare questo problema si utilizza una **cache separata per istruzioni e dati**, in modo tale che nello stadio di prelievo di un'esecuzione si utilizza la cache per le istruzioni, per l'accesso in memoria si utilizza la cache di dati.

5.3 Valutazione delle prestazioni

Andiamo ad analizzare, in questa sezione, quanto effettivamente sia il **tempo di esecuzione** (numericamente) e quanto i rallentamenti nella pipeline incidono. Sappiamo che l'**equazione di base delle prestazioni** è:

$$T = \frac{N \cdot S}{R} \quad (5.1)$$

dove:

- **T** è il **tempo di esecuzione**;
- **N** è il **conteggio dinamico** delle istruzioni;
- **S** è il **numero medio di cicli di clock** per istruzione;
- **R** è la **frequenza di clock**, che è l'*inverso* del periodo di clock.

Vi è un'altra variabile molto importante, che denotiamo con **P**, che indica il **throughput**, ovvero il *numero medio di istruzioni che vengono eseguite nell'unità di tempo*, che vale:

- Senza pipeline:

$$P_{np} = \frac{R}{S} \quad (5.2)$$

- Con pipeline ideale:

$$P_p = R \quad (5.3)$$

Inoltre nel caso *reale*, ovvero dove non può esistere pipeline ideale a causa di **conflitti** dovuti da *dipendenze di dato, salti..* è attenuato il **guadagno ideale S**

5.3.1 Effetti di stalli

Abbiamo visto il caso generale in cui avevamo una *pipeline ideale*. Vediamo adesso gli effetti dei conflitti dovuti da **stalli** quantitativamente tramite la seguente formula:

$$P_p = \frac{R}{1 + \delta} \quad (5.4)$$

dove δ rappresenta l'*incremento del tempo di esecuzione*, ovviamente con $\delta = 0$ abbiamo la pipeline ideale. Se abbiamo un *processore con inoltro operandi* abbiamo uno stall di 1 ciclo di clock per dipendenze di dato per le istruzioni **load**. Se per esempio contiamo che le istruzioni load sono pari al 25% nel conteggio dinamico, con 40% di queste che sono seguite da istruzioni dipendenti, abbiamo che il δ viene calcolato come:

$$\delta_{stall} = istrl_{load} \cdot istrd_{dipend} \cdot cicl_{extra} = 0,25 \cdot 0,40 \cdot 1 = 0,1 \quad (5.5)$$

Quindi abbiamo che P_p sarà:

$$P_p = \frac{R}{1 + \delta_{stall}} = \frac{R}{1,1} \quad (5.6)$$

Quindi possiamo dire che il *throughput* finale P_p darà di circa $0,91R$, con una perdita di prestazioni di circa il 10%.

5.3.2 Effetti di penalità di salto

Consideriamo adesso un processore che abbia:

- **Calcolo della destinazione di salto** al secondo stadio;
- **Predizione dinamica** di salto;
- **Buffer di predizione** di salto.

Nonostante l'architettura del processore sia completa dal punto di vista del calcolo del salto, abbiamo visto che, anche con la predizione, dinamica capita di sbagliare la predizione di salto. Facciamo un esempio con dati rilevati su analisi reali fatte su diversi programmi, che dicono che le istruzioni di salto occupano il **20%** del **conteggio dinamico** e che per il **10%** delle volte si **sbaglia la predizione di salto**. Ci calcoliamo il $\delta_{penalita_salto}$, come abbiamo fatto in precedenza, moltiplicando **numero di istruzioni, percentuale di istruzioni di salto e percentuale di errore di predizione**. Abbiamo quindi:

$$\delta_{penalita_salto} = 0,20 \cdot 0,10 \cdot 1 = 0,02 \quad (5.7)$$

Notiamo quindi che in questo caso il δ incide in maniera minore rispetto alla dipendenza di dato, ma essendo dipendenze di dato ed errori di predizione indipendenti abbiamo che gli **effetti** di entrambi si **sommano**.

5.3.3 Effetti di cache miss

Supponiamo di avere un parametro **p_m**, che ci indica il numero di cicli di attesa per avere una risposta dalla memoria RAM, e supponiamo di avere questi altri parametri:

- **m_i**: indica la *frazione di istruzioni prelevate con cache miss*;
- **d**: indica la *frazione di istruzioni load/store con conteggio dinamico*;
- **m_d**: indica la frazione di istruzioni con **accesso in memoria** (load/store) con **cache miss**.

Abbiamo che, dati questi parametri appena elencati, il δ_{miss} , cioè l'incremento del tempo di esecuzione per cache miss, è dato dalla seguente equazione:

$$\delta_{miss} = (m_i + d \cdot m_d) \cdot p_m \quad (5.8)$$

Quindi se abbiamo per esempio $m_i = 0,05; d = 0,3; m_d = 0,1; p_m = 0,8$, abbiamo che il δ_{miss} è:

$$\delta_{miss} = (0,05 + 0,3 \cdot 0,1) \cdot 10 = 0,8 \quad (5.9)$$

Quindi se sommassimo gli effetti di tutti gli effetti degli stalli, avremmo una perdita di prestazioni molto grande di circa il 92%:

$$\delta = \delta_{stallo} + \delta_{salto} + \delta_{miss} = 0,1 + 0,02 + 0,8 = 0,92 \quad (5.10)$$

e notiamo anche che gli effetti della **cache miss** sono quelli che provocano una maggiore perdita di prestazioni. Diciamo che quindi per evitare l'incidenza della cache miss sulle prestazioni si possono adottare principalmente due soluzioni:

- **Ingrandire la cache con più strati** in modo tale che si eviti il fatto che il dato non si trovi in cache ed è la soluzione più usata tra le due;
- **Diminuire p_m** , ovvero il numero di cicli di attesa per avere una risposta dalla memoria RAM. Ciò ovviamente comporterebbe avere una memoria ancora più veloce, il chè è un processo abbastanza complesso considerando che le memorie di oggi occupano diveri gigabyte.

5.3.4 Numero di stadi della pipeline

Potremmo pensare, banalmente, che *aumentare il numero di stadi in pipeline* farebbe **aumentare il throughput ideale**. In realtà vi sono diverse controindicazioni e problematiche che potrebbero insorgere, come ad esempio:

- La probabilità di **stallo**, con un numero di stadi maggiore, sarebbe molto più probabile e gli effetti di tali stalli inciderebbero in maniera ancora maggiore poichè vi potrebbero essere più dipendenze di dato o conflitti dati da istruzioni distanti nella pipeline;
- Ogni volta che viene inserito uno stadio deve avvenire un *frazionamento* di quest'ultimo che provoca un costo di realizzazione sempre maggiore;
- Infine, considerando che l'**ALU** è l'unità funzionale più lenta limiterà la frequenza di clock e quindi il frazionamento degli altri stadi.

5.4 Processore superscalare

Per ovviare alle problematiche legate alla pipeline e aumentare ancora di più il *throughput* delle istruzioni sono stati creati dei **processori superscalari** che hanno la caratteristica di avere più unità funzionali che lavorano in *parallelo* ognuno con la sua pipeline. Si ha quindi la caratteristica per il quale vengono prelevate più istruzioni (2 o 4) per il quale si ha un throughput ideale incrementato per un ciclo di clock.

5.4.1 Organizzazione hardware superscalare

Per quanto riguarda l'*hardware* che compone un processore superscalare, abbiamo:

- L' **unità di prelievo** che preleva più istruzioni che vanno a finire in una *coda di istruzioni*;

- L'**unità di smistamento** che preleva le istruzioni in *testa* alla coda controllando cosa deve effettivamente fare quella determinata istruzione inoltrandola all'unità adatta, ovvero all'**unità di elaborazione** (ALU) oppure all'**unità load/store** (accesso alla memoria)

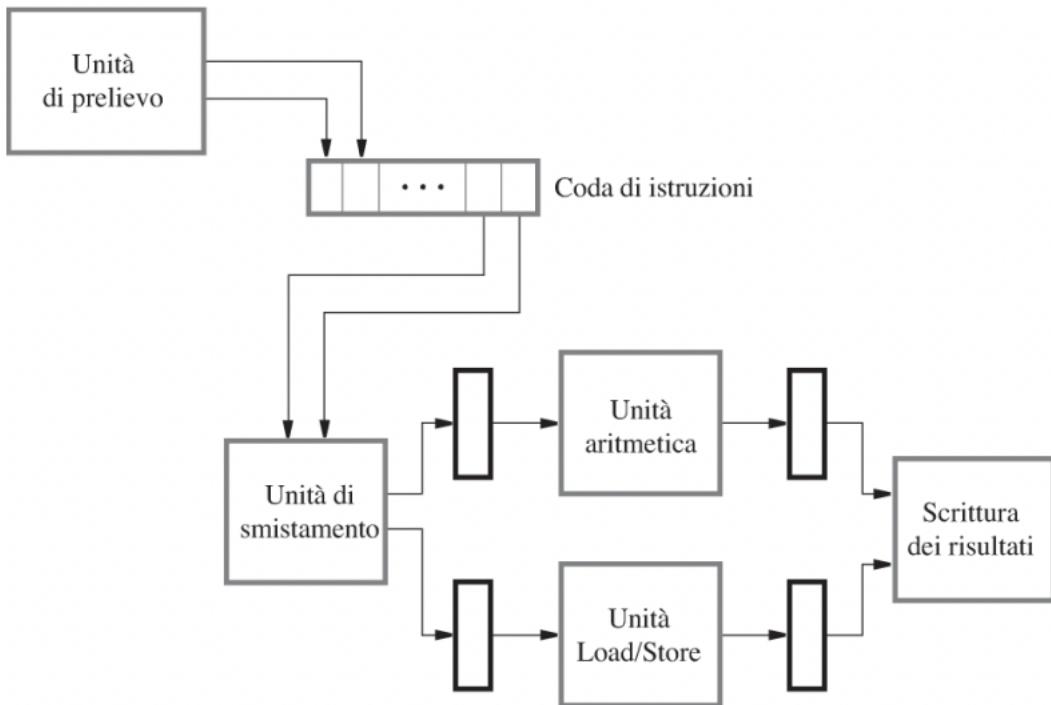


Figure 5.10: Organizzazione hardware superscalare

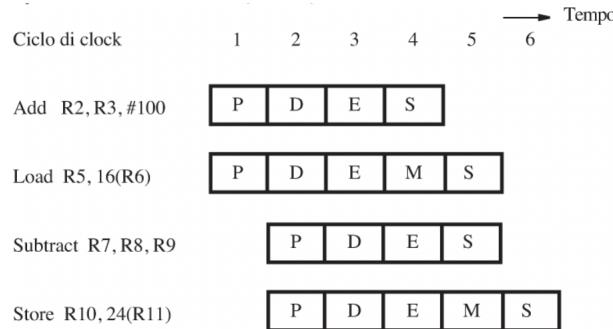
5.4.2 Esempio di esecuzione superscalare

Facciamo un esempio di esecuzione superscalare con delle istruzioni nella quale non vi sono né dipendenze di dato né salti:

```

ADD R2,R3,#100
LOAD R5,16(R6)
SUBTRACT R7,R8,R9
STORE R10,24(R11)
  
```

Possiamo vedere attraverso la seguente immagine come i vari stadi si sviluppano nell'esecuzione superscalare:



Ovviamente in questo caso l'istruzione ADD verrà eseguita nel terzo ciclo di clock tramite l'**unità di elaborazione (ALU)**, mentre per quanto riguarda l'istruzione LOAD verrà eseguita tramite l'**unità load/store**. Come abbiamo detto in precedenza, sarà l'**unità di smistamento** che, una volta prelevate le istruzioni dalla coda, le smisterà in base all'operazione da eseguire all'unità di elaborazione (ADD) o all'unità load/store. Possiamo notare, inoltre, che è possibile che due istruzioni vengano scritte nei registri contemporaneamente allo stadio S (scrittura), come in questo caso per la load e la subtract e, poichè sappiamo che il banco registri presenta solamente due uscite e un ingresso per ogni istruzione, servirà dell'hardware aggiuntivo per permettere la **lettura e scrittura multipla di registri**.

5.4.3 Salti e dipendenze di dato

Abbiamo capito quindi che l'ottimizzazione del throughput per un processore superscalare si ha con l'**alternanza di unità funzionali** per istruzioni successive (unità di elaborazione e unità load/store). In effetti molti compilatori moderni fanno in modo di riordinare le istruzioni in modo da avere questa alternanza. Tuttavia possono insorgere delle problematiche come:

- **Salti** che intervengono sulla **sequenza di esecuzione di istruzioni**;
- **Dipendenze di dato** che possono impedire il riordino delle istruzioni;
- **Cache miss** che impone diversi stalli.

Per quanto riguarda i primi 2 punti si adottano le seguenti soluzioni:

- Per il salto, disponendo di maggiore hardware nel processore, eseguo l'istruzione sia in caso di salto che in caso di non salto, facendo la cosiddetta **esecuzione speculativa**. Infine una volta che so il risultato della predizione utilizzerò il risultato che mi interessa e scarterò l'altro.
- Per le dipendenze di dato l'istruzione con tale dipendenze non può essere fornita dall'unità di smistamento a valle fino a quando gli operandi saranno pronti: ovvero o quando il dato è stato scritto nel registro destinazione oppure quando è stato passato lo stadio di elaborazione.

5.4.4 Esecuzione fuori ordine

Potrebbe capitare, per come è fatta l’architettura di un processore superscalare, che alcune istruzioni vengano completate in ordine diverso rispetto a come erano nel programma. Per quanto riguarda la dipendenza di dato il processore si comporterà correttamente; ma il problema si pone quando nascono **eccezioni imprecise** per il quale vengono eseguite le istruzioni successive a quella in cui è nata tale eccezione, che in realtà non dovevano essere eseguite. La soluzione che si utilizza è quella di un **esecuzione fuori ordine** per il risultato dell’esecuzione delle istruzioni viene messo in dei *buffer temporanei*, e in uno step successivo avverrà il **completamento** mettendo i risultati delle varie esecuzioni nell’ordine corretto del programma

Completamento dell’esecuzione

Una volta eseguita l’esecuzione fuori ordine delle istruzioni, c’è bisogno di un **completamento dell’esecuzione in ordine**. Ciò avviene tramite un’unità chiamata **unità di commitment** (o *impegno*) che gestisce tale completamento: infatti una volta eseguita l’istruzione con il commitment, i suoi effetti saranno irreversibili. Si usa un buffer di riordino delle istruzioni. Una coda garantisce che l’ordine dei commitment delle istruzioni coincida con quello di smistamento

5.4.5 Funzionamento dello smistamento

L’**unità di smistamento** deve assicurarsi che tutte le risorse necessarie all’esecuzione di un’istruzione siano disponibili. Per esempio se l’istruzione ha necessità di scrivere il risultato in un buffer temporaneo deve essercene uno libero, e viene **prenotato** dall’unità di smistamento. Deve pure essere disponibile un’unità di esecuzione e una locazione nel buffer di riordino per il commitment dei risultati. Quando tutte le risorse sono state prenotate l’istruzione viene inviata. Inoltre per quanto detto riguardo all’*esecuzione fuori ordine* si potrebbero smistare fuori ordine le istruzioni: per via della mancanza di una risorsa per un’istruzione si potrebbe mandare in esecuzione la successiva che ha tutte le risorse, assicurandosi che siano ritirate in ordine corretto e che non si verifichi un blocco, che viene detto **deadlock** e consiste nel fatto che due unità usano una risorsa condivisa, e ciascuna unità aspetta di completare l’operazione finché l’altra non abbia completato. Ad esempio se la Subtract viene inviata prima della Load e il registro temporaneo viene prenotato per la Subtract; la Load, quindi non viene inviata poiché aspetta lo stesso registro temporaneo. Il registro non diventerà disponibile finché la Subtract non sia stata ritirata, ma si aspetta che prima sia ritirata la Load. Si ha un blocco. Emettere le istruzioni in ordine evita questo problema

Capitolo 6

SISTEMA DI INGRESSO E USCITA

In questo capitolo parleremo di come *comunica* il processore con le varie periferiche, facendo riferimento alla **struttura a bus**

6.1 Struttura a bus

La **struttura a bus** è formata da **linee elettriche** di interconnessione che fanno comunicare il processore con la memoria o le varie periferiche per una **coppia di dispositivi** alla volta. In queste linee di bus facciamo viaggiare tipologie diverse di segnali:

- **Bus di dati:** il processore o la periferica imporranno in queste linee dei *dati*;
- **Bus di controllo:** verranno imposti sulle linee *segnali di controllo*;
- **Bus di indirizzo:** i registri presenti nell’interfaccia della periferica o del processore corrisponderanno a certi indirizzi selezionati tramite le linee che figurano il bus di indirizzi.

Quindi, quando rappresentiamo una linea di bus, dobbiamo immaginarcì che ce ne siano diverse **parallele**, ognuna in base alla tipologia di segnali che passa al suo interno. Inoltre ricordiamo che, grazie all'**unificazione degli indirizzi di memoria**, o *memory-mapped I/O* le **interfacce** di ogni periferica decodificano in base all’indirizzo passato dal bus di indirizzi, se questo appartiene all’insieme proprio dell’interfaccia, se riguarda i circuiti di controllo oppure se riguarda dei dati presenti nel registro di dati.

6.2 Funzionamento del bus

Per quanto riguarda il funzionamento del bus, il processore e le periferiche si attengono a un certo **protocollo** che è un *insieme di regole* per tutte le unità legate al bus. Tali regole vengono specificate attraverso specifici **segnali di controllo**: ad esempio se prendiamo il segnale di controllo **R/W**, esso specifica l'operazione di **lettura** o **scrittura** nel quale ovviamente il soggetto è sempre il processore. Abbiamo che secondo il protocollo quando tale segnale sarà posto *alto*, ovvero a 1, vogliamo fare una lettura; mentre se sarà posto *basso*, ovvero a 0, vogliamo fare una scrittura. Inoltre per il protocollo abbiamo bisogno di una certa **temporizzazione** per imporre o togliere segnali dal bus in base al tipo di protocollo che sceglieremo. Esistono, infatti, due grandi famiglie di protocolli del bus:

- **Bus sincrono:** la **temporizzazione** in questo caso viene dettata da un **segnale di clock**;
- **Bus asincrono:** la **temporizzazione** non è dettata dal segnale di clock, poiché non esiste in questo caso, ma avremo dei **segnali di controllo appositi** che determineranno la temporizzazione.

Infine nei **dispositivi coinvolti** nel trasferimento di dati, avremo dei ruoli per entrambi i dispositivi, per il quale abbiamo:

- Un **master** che è il *dispositivo che da inizio* all'operazione;
- Uno **slave** che è l'altro dispositivo che riceve il dato dell'operazione

Andiamo adesso a vedere più nello specifico le due famiglie di protocolli, ovvero il bus sincrono e asincrono.

6.2.1 Bus sincrono

Per capire bene come sia fatto il **bus sincrono** prendiamo in considerazione la seguente figura, che raffigura appunto *l'andamento temporale del bus sincrono*:

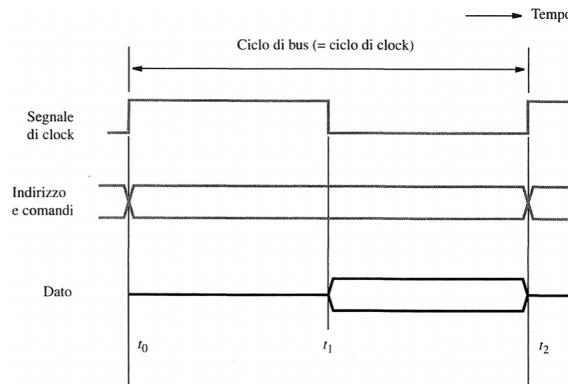


Figure 6.1: Andamento temporale del bus sincrono

Come già detto in precedenza, nel caso di bus sincrono la **temporizzazione** è gestita dal **segnale di clock**, che come possiamo vedere in figura 6.1, a una durata $t_2 - t_0$ che è più lunga rispetto al periodo di clock del processore. Analizzando più in dettaglio la figura possiamo vedere che nel bus si distinguono **linee parallele** che, per quanto detto nella sezione 6.1, rappresentano la realtà fisica di come è effettivamente costruito il bus. Inoltre possiamo notare che nel bus vi sono dei momenti nel quale *le linee si incrociano* e, in tal caso, significa che qualcosa è cambiato nel bus; mentre quando vediamo solo una *linea intermedia* significa che il processore è *scollegato* dalle periferiche e, di conseguenza, nel bus *non vi è un valore significativo*, cioè vi è il cosiddetto **valore di alta impedenza**.

Uso del bus Una volta spiegate come vengono rappresentate le linee nel bus della figura 6.1, adesso passiamo a vedere come effettivamente avvenga il **funzionamento del bus sincrono**. Prendiamo come esempio l'*operazione di lettura*:

- Abbiamo compreso che il segnale di clock impone la *tempistica* per fare le operazioni. Come possiamo notare in figura 6.1, al **fronte di salita del clock**, il processore impone il valore dell'indirizzo dell'istruzione o del dato che vuole leggere, infatti vediamo in figura che le linee del bus di indirizzi si incrociano, ovvero è stato imposto un indirizzo nel bus, e inoltre imporrà il segnale di controllo R/ \bar{W} alto.
- La *periferica* si accorgerà che nel bus di indirizzi è stato imposto un valore e se la riguarderà andrà a leggerlo;
- Una volta letto, la periferica dovrà emettere il dato in un tempo $t_1 - t_0$, ovvero la **metà del periodo di clock**. Se la periferica non dovesse riuscire a emettere il dato in questo periodo, quindi se lo emettesse troppo tardi, il processore non capirebbe che il dato sia della periferica poiché si tratta di **dati instabili**;

- Se è avvenuto correttamente l'emissione del dato dalla periferica, nell'intervallo di tempo $t_2 - t_1$ il processore leggerà il dato e lo metterà nei suoi registri interni

Per quanto riguarda la **scrittura** il processo è molto simile, l'unica cosa che cambia è che ovviamente il segnale R/\bar{W} sarà impostato dalla periferica e sarà impostato basso, poiché appunto si vuole fare una scrittura.

Ritardi di propagazione

Abbiamo analizzato come funziona effettivamente il bus, senza però tener conto del fatto che la **velocità di propagazione** dei segnali attraverso il bus è un'onda finita. Ciò comporta il fatto che esiste un **ritardo di propagazione** dovuto allo sfasamento dato dal fatto che il segnale deve arrivare da un punto all'altro del bus, che non sempre è vicino. Analizziamo quindi la seguente figura:

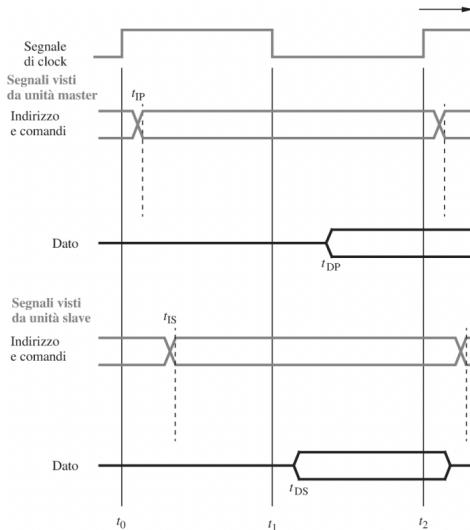


Figure 6.2: Ritardi di propagazione del bus sincrono

Supponiamo quindi che le unità legate al bus riescano a vedere lo stesso segnale di clock del bus. Notiamo dalla figura 6.2 che abbiamo uno sfasamento dato dall'intervallo di tempo $t_{ip} - t_0$, dove t_{ip} rappresenta l'istante di tempo nel quale il processore impone gli indirizzi. Inoltre ciò porta al fatto che l'unità *slave* non avrà più un intervallo $t_1 - t_0$ per leggere l'indirizzo, ma avrà un tempo ridotto $t_1 - t_{is}$, dove t_{is} rappresenta l'istante di tempo nel quale la periferica riesce a vedere l'indirizzo imposto dal *master*. Allo stesso modo quando la periferica imporrà il dato che il processore dovrà leggere, nell'istante t_{DS} , il processore leggerà tale dato in un istante successivo t_{DP} prima di t_2 . Possiamo vedere inoltre che il dato rimane stabile un po' oltre il tempo t_2 , ovvero l'istante di tempo che sancisce la fine del periodo di clock del bus: questo avviene perché

si aspetta che *il dato si stabilizzi*, ovvero far sì che non vari per essere scritto correttamente nei registri interni dell'unità in considerazione, per poi diventare in alta impedenza.

Trasferimento di dati in più cicli

Possiamo, col bus sincrono, apportare dei miglioramenti alla temporizzazione che ci possa aiutare in 2 punti sfavorevoli, che sono:

- Dobbiamo progettare il periodo di clock in modo tale che la periferica riesca a completare le sue operazioni. Infatti, potrebbero esserci periferiche più veloci, rispetto a quella più lenta, che vengono limitate dal fatto che esse potrebbero imporre il dato in maniera più veloce ma per la temporizzazione del periodo di clock si ha una cera *perdita di tempo*.
- L'altro punto riguarda il fatto che non sempre la periferica riesce a capire che l'indirizzo di selezione riguarda se stessa e quindi capita che non legge tale indirizzo.

Analizziamo tale figura:

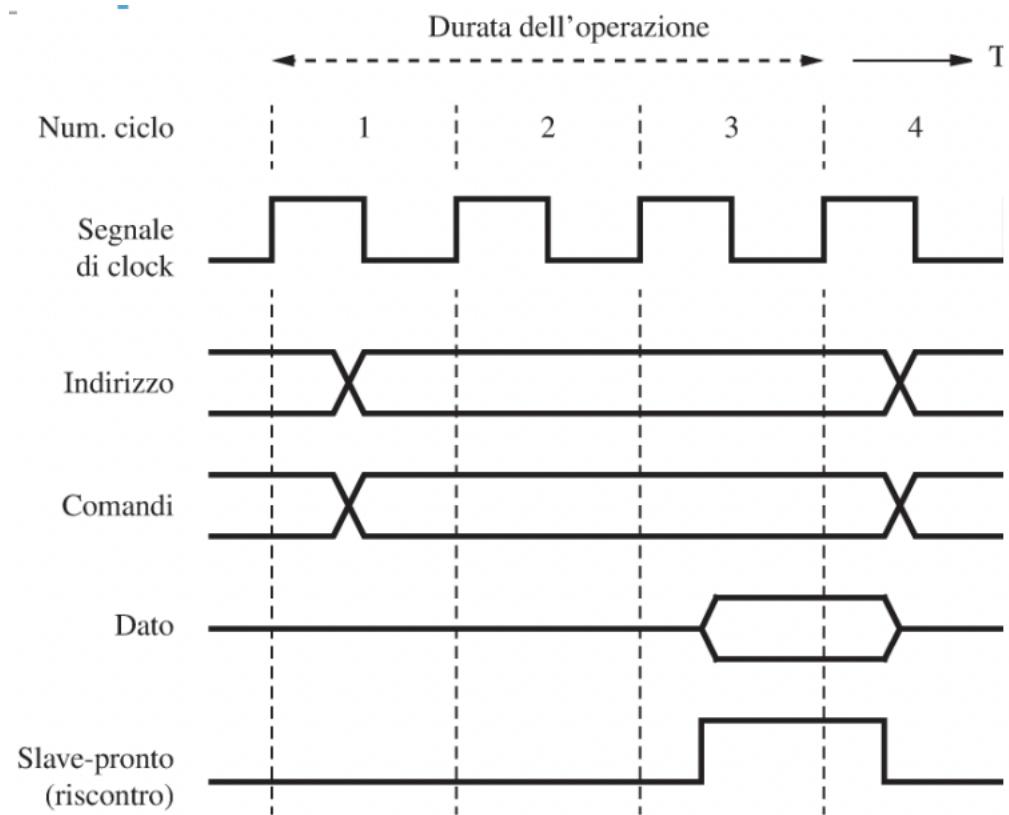


Figure 6.3: Bus sincrono multiciclo

Per ovviare ai problemi descritti nei 2 punti in precedenza, innanzitutto, come possiamo vedere in figura 6.3 progettiamo un **segnale di clock più piccolo** e aggiungiamo un **segnale di conferma della periferica slave-pronto**. Ciò porta ad avere più cicli di clock per l'attesa del segnale di conferma, che, una volta che viene imposto alto dalla periferica, significa che il dato potrà essere letto contestualmente dal processore. In tal modo se abbiamo periferiche più veloci, esse emetteranno il segnale slave-pronto prima così da completare prima l'operazione. Se, invece, il segnale slave-pronto non è imposto alto dalla periferica, il processore aspetterà per un numero variabile di cicli di clock, ma tal numero è comunque limitato: se il segnale non verrà imposto alto, il processore non aspetterà più.

6.2.2 Bus asincrono

Come abbiamo già accennato in precedenza, parlando del **bus asincrono** la **temporizzazione avviene tramite segnali di controllo**. Tali segnali servono a collegare tra loro le tempistiche delle varie unità sincronizzando quindi

master e slave:

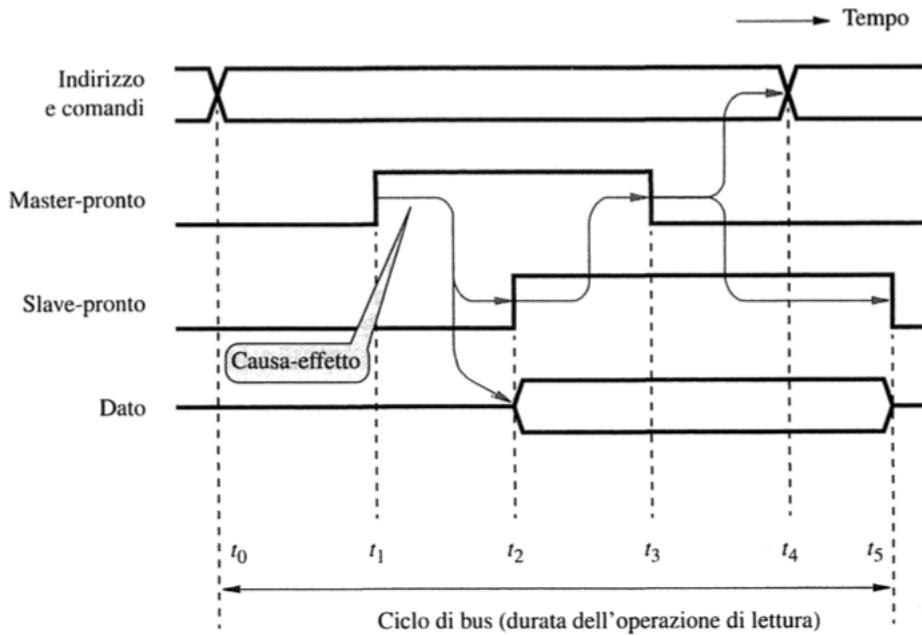


Figure 6.4: Andamento temporale dell'operazione di lettura tramite bus asincrono

Prendiamo in considerazione quindi sempre il fatto che stiamo eseguendo un'operazione di lettura: seguendo l'immagine sopra notiamo che il *master* immette un dato nel bus di indirizzi e mette il **segnale master-pronto** alto. Tale segnale verrà visto da ogni periferica, e, in special modo, dalla periferica interessata, a differenza del segnale di clock che individuava il fatto che il dato fosse presente un dato. Una volta che lo *slave* riconosce che il dato lo riguarda essa emetterà il dato e alzerà il segnale **slave-pronto** che indicherà il fatto che è stato messo il dato dalla periferica. Qualche istante dopo verrà abbassato *master-pronto*, in modo tale che venga tolto l'indirizzo dal bus di indirizzi, e come conseguenza anche *slave-pronto* andrà basso. Dall'immagine possiamo notare che il tempo $t_1 - t_0$ rappresenta lo **sfasamento temporale** con cui il segnale viene visto dalla periferica, dovuto a *ritardi variabili* dovuti al segnale che si propaga dal master agli slave. Notiamo quindi che tale tipo di connessione è molto tollerante riguardo ai ritardi, a differenza del bus sincrono che si basa su un certo periodo di clock.

6.2.3 Confronto bus sincrono e bus asincrono

Dopo aver trattato nel dettaglio le due tipologie di bus, ovvero **sincrono** e **asincrono**, vediamo di fare un confronto tra i 2 utilizzando come criteri:

- **Protocolli** di comunicazione utilizzati
- **Velocità** di propagazione

Per quanto riguarda il primo punto, ovvero i **protocolli utilizzati**, abbiamo compreso che il **bus asincrono** fa a meno del segnale di clock e inoltre sono tollerati gli eventuali ritardi di propagazione. Da questo punto di vista quindi potremmo pensare che sia meglio il bus asincrono. Però, se analizziamo il secondo punto, noteremo che la **velocità di propagazione** del bus sincrono è sicuramente maggiore rispetto a quella del bus asincrono: infatti quando il segnale master-pronto verrà messo alto, esso dovrà viaggiare tra il processore e la periferica e la stessa cosa dicasì per il segnale slave-pronto. Di conseguenza se facciamo una stima, dovrà aspettare 4 volte di più in termini di tempo col bus asincrono rispetto al bus sincrono. Per questo molto spesso nei moderni calcolatori è presente il bus sincrono.

6.2.4 Pilotaggio del bus

In ogni momento un solo dispositivo può essere abilitato all'invio del segnale, tutti gli altri devono avere i loro bus driver disabilitati, ovvero le porte logica che sono abilitate a inviare l'output su una linea del bus, e per questo si usano **porte a tre stati** (tri-state)

6.3 Arbitraggio del bus

Abbiamo trattato il dispositivo **master** generalizzando il fatto che si trattasse del processore; ma in realtà *non sempre il master è il processore*. Ci sono infatti delle unità di I/O che vengono pilotate dal processore per eseguire operazioni che farebbero perdere troppo tempo al processore stesso, come ad esempio *trasferire dati nel disco*. La domanda che ci possiamo porre quindi è: *visto che possono esserci 2 o più master, chi prenderà il controllo del bus?* In questo caso utilizziamo un **circuito di arbitraggio**:

- Abbiamo 2 o più master collegati al circuito di arbitraggio e ognuno manda una **richiesta d'uso**;
- Il circuito di arbitraggio, che farà appunto da *arbitro*, deciderà in base alla **priorità** a chi concedere il controllo del bus: la priorità, molto spesso, può derivare dalla necessità di accedere al bus senza ritardi per evitare errori.

Possiamo vedere quanto scritto sopra "riassuntato" nella seguente immagine:

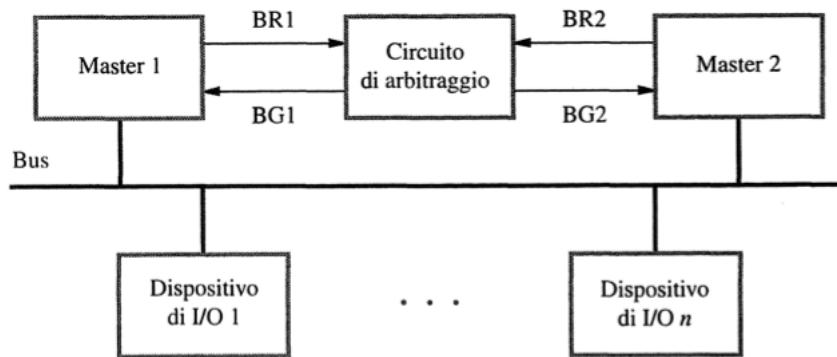


Figure 6.5: Circuito di arbitraggio del bus

Vediamo appunto che BR1 e BR2 sono le *linee di richiesta* da parte dei master, mentre BG1 e BG2 sono le *linee di grant*, ovvero di "concessione del bus" da parte del circuito di arbitraggio.

Capitolo 7

SISTEMA DI MEMORIA

Con **sistema di memoria** si intende tutto ciò che serve per *memorizzare i dati*, e i dispositivi principali utilizzati a questo scopo sono:

- La **memoria cache**, ovvero la memoria più "vicina" in termini di velocità al processore
- La **memoria RAM** o *random access memory*, ovvero qualsiasi locazione di memoria in cui vi è un *tempo di accesso costante*
- La **memoria secondaria**, ovvero dischi fissi, periferiche di I/O o memoria flash (SSD)

Tali dispositivi di memoria appena elencati presentano alcune **caratteristiche**, che sono i *criteri* per distinguere le varie tipologie di memorie tra loro, e sono:

- **Velocità**: misura il **tempo di accesso in memoria**, ovvero quanto *velocemente* accediamo ad essa
- **Latenza**: misura quanto **tempo** passa dall'**emissione di un indirizzo** all'**emissione del successivo**
- **Larghezza di banda**: misura **quanti dati** riesce la memoria a fornire **contemporaneamente**
- **Capacità**: misura **quanti byte** possiamo **memorizzare all'interno della memoria**
- **Costo**: in base alla tipologia di memoria, e in base ai vari criteri appena elencati ogni memoria ha un costo differente

Riguardo alla *capacità di memoria* può essere utile tale considerazione: poichè sappiamo che lo **spazio di indirizzamento del processore** è limitato a 2^n bit, sarebbe inutile creare una ram di capacità di 1 TB poichè il processore non riuscirebbe a indirizzare fino a 1 TB. Per questo tutto ciò che verrà messo oltre ai 2^n bit di indirizzamento sarà inutilizzato.

7.1 Concetti di base

Andando ulteriormente ad approfondire quanto detto nell'introduzione del capitolo, possiamo elencare i seguenti **concetti di base**:

- La **velocità di memoria** è misurata come il **reciproco del tempo di accesso alla memoria stessa**, ovvero il tempo necessario fra l'inizio dell'operazione e il completamento
- Il **tempo di ciclo di memoria**, solitamente leggermente maggiore rispetto al tempo di accesso, è quell'unità di misura che indica il **tempo minimo** che si deve aspettare per l'inizio di due operazioni di memoria successive
- Se stiamo accedendo a un **tempo costante** a qualsiasi parola di memoria, allora la memoria a cui stiamo accedendo viene detta ad **accesso casuale** o **RAM** (*random access memory*)
- La memoria **cache** è una memoria *veloce*, in mezzo tra processore e memoria centrale RAM, che è sufficientemente veloce per permettere al processore di perdere meno tempo ma è anche più piccola rispetto alla memoria centrale. La motivazione per cui è relativamente difficile costruire un unico blocco grande di cache è il suo **costo**, che è molto più alto di una cache.
- Poichè molto spesso il processore ha bisogno molto spesso di dati, per migliorare le prestazioni si trasferiscono **blocchi contigui di dati**, ovvero memorizzati vicino. Questa caratteristica è molto importante: se ad esempio immaginiamo di utilizzare una memoria lenta, come ad esempio il disco, e voglio sfruttare tutta la sua *larghezza di banda*, trasferendo blocchi di dati, avremo sicuramente un *tempo di latenza* minore.

7.2 Memoria RAM a semiconduttori

Per quanto riguarda la **memoria RAM**, ovvero la memoria centrale, i suoi *tempi di accesso* variano da 100 ns (nanosecondi) a 1 ns. Ovviamente più le tecnologie sono avanzate e costose, più la velocità della RAM aumenterà. La *caratteristica che accomuna tutte le RAM* è la **volatilità**, ovvero il dato è mantenuto fin quando c'è *alimentazione*; una volta che l'alimentazione viene staccata tali dati presenti in memoria verranno persi. Fatte queste premesse, possiamo dire che le RAM si suddividono in 2 tipi (entrambe volatili):

- **Statica:** il dato viene mantenuto fin quando c'è corrente
- **Dinamica:** il dato viene perso anche se c'è alimentazione poichè deve essere periodicamente **rinfrescato**, ovvero riscritto periodicamente.

Potremmo pensare, una volta descritti i 2 tipi di RAM, che quella statica sia la migliore. In effetti è così: considerando il fatto che per la RAM dinamica

bisogna **aggiungere circuiti hardware** per permettere il rinfresco del dato, e considerando il fatto che nonostante il rinfresco il dato potrebbe essere perso poichè **la corrente si dissipa** nella RAM dinamica; effettivamente la scelta migliore è la RAM statica. In questo capitolo però parleremo anche di RAM dinamica poichè dobbiamo considerare anche il fattore del **costo**, che è sicuramente molto più elevato per una RAM statica, e, di conseguenza, dobbiamo tener conto anche dell'utilizzo di una RAM dinamica meno efficiente ma più economica.

7.2.1 Organizzazione interna di chip di memoria

Analizziamo la seguente immagine, che descrive l'**organizzazione interna di un chip di memoria**:

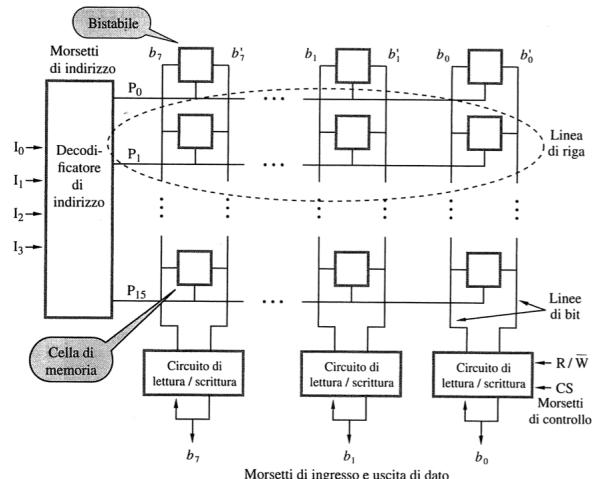


Figure 7.1: Chip di memoria 16x8 come matrice di celle da 1 bit

Come possiamo vedere, la memoria è disposta a **celle** che al loro interno memorizzano 1 bit. Tali celle vengono disposte poi come una **matrice di celle** che è composta, come possiamo vedere dall'immagine sopra, da 8 bit per riga, e il numero di righe può variare. Nel nostro esempio in figura abbiamo 16 righe e 8 colonne, quindi abbiamo una capienza di $16 \times 8 = 128$ bit. Ogni linea di riga, quindi, conterrà una parola di memoria e, inoltre, tali linee sono pilotate dal decodificatore di indirizzo, che è un'unità al quale passiamo il numero di collegamenti adatti per selezionare una delle 16 righe. In questo caso $2^4 = 16$, quindi mi serviranno esattamente 4 linee. Le linee verticali, invece, sono collegati al **circuito di lettura e scrittura** tramite 2 linee. Quindi nel caso di **lettura**, viene messo alto il segnale R/\bar{W} , così il bit uscirà dal circuito di controllo verso il circuito di lettura/scrittura attraverso il bus di dati. Per la **scrittura**, invece, manderò un indirizzo al decodificatore e impongo il segnale R/\bar{W} basso, in

modo tale che il circuito di lettura/scrittura imporrà il bit dal basso nelle linee verticali alla riga corrispondente. Facendo quindi un conto del **numero totale di collegamenti**, abbiamo:

- 4 linee di indirizzo
- 8 linee di dati
- 2 linee di segnali di controllo
- 2 linee di alimentazione (**morsetti**)

Un totale quindi di 17 collegamenti. Se ad esempio però volessimo memorizzare 1 kbit, avremo una matrice 128×8 . In tal caso il numero di collegamenti sarebbe maggiore: infatti per selezionare una delle 128 righe non abbiamo bisogno di 4 ma bensì di 7 linee ($2^7 = 128$). Il conto totale di collegamenti sarebbe quindi $7 + 8 + 2 + 2 = 19$ collegamenti.

7.2.2 Memoria statica

Andiamo adesso a vedere l'organizzazione delle singole celle in una **memoria statica**. Ricordiamo che la peculiarità della memoria statica è che non abbiamo bisogno di rinfrescare il dato: vediamo adesso di spiegare il perchè.

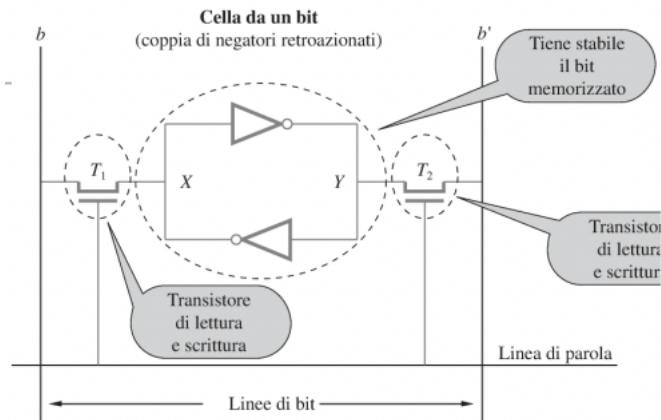


Figure 7.2: Singola cella di memoria statica

Come possiamo vedere dalla figura ogni cella è con le linee di riga e di colonna. La linea di riga viene detta **linea di parola**, in verticale invece abbiamo il valore del bit che esce o entra. All'interno della cella di memoria sono presenti *due transistor*, etichettati come nell'immagine con T_1, T_2 , e tali transistor sono pilotati dalla linea di parola. Inoltre tali transistor sono collegati a un circuito con **porte NOT** nel quale l'uscita di una porta è collegata all'ingresso dell'altra. Se nella linea di parola quindi ho il valore 0, quindi, farò sì che i

transistor T_1, T_2 lavoreranno in **interdizione**, ovvero il *circuito* è come se fosse *aperto*, e il valore presente all'interno della cella rimarrà sempre uguale. Tale caratteristica quindi permetterà di mantenere lo stato senza dover rinfrescare il dato. Quando dovremo, invece fare delle operazioni con la memoria si procede nel seguente modo:

- Per la **lettura** il circuito dei transistor lavorerà in **saturazione** poichè abbiamo fornito alla linea di parola il valore 1. In tal caso nella linea verticale verrà fornito il bit memorizzato in b e in b' il suo *complemento*.
- Per la **scrittura** il circuito imporrà il valore in b e il suo complemento in b' , che sarà trasmesso a X attraverso l'attivazione della linea di parola. Tale valore una volta che la linea di parola sarà impostata a 0, verrà mantenuto, per i motivi elencati in precedenza, all'interno della cella.

Ovviamente, come avevamo detto nell'introduzione parlando di memoria statica, abbiamo un costo abbastanza elevato, infatti per ogni singola cella utilizziamo **6 transistor**. Ovviamente il fatto che il dato non ha bisogno di essere rinfrescato ci da un enorme vantaggio in termini di utilizzo di *memoria cache*

7.2.3 Memoria dinamica

Abbiamo parlato della memoria statica. Vediamo adesso quali sono le caratteristiche principali di una **memoria dinamica**.

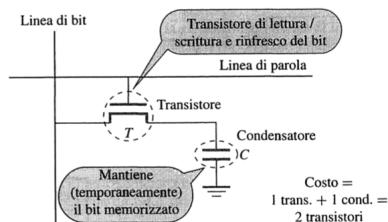


Figure 7.3: Memoria dinamica

Come possiamo vedere dall'immagine, le celle di memoria presentano al loro interno un *transistor* T collegato a un **condensatore** che va a terra. Il valore del bit è contenuto nel condensatore e , quando memorizziamo un valore, il transistor sarà in **saturazione** e il circuito sarà quindi chiuso: in tal caso riuscirà a **caricare il condensatore**, quindi a fare il cosiddetto **rinfresco** di cui abbiamo parlato quando abbiamo introdotto la memoria dinamica. Quando, invece, la linea di parola viene messa a 0, e il circuito diventa quindi in **interdizione**, allora viene scollegato il condensatore dal transistor. In tal caso il valore di carica del condensatore nel tempo scenderà e , se non rinfreschiamo il dato, esso andrà perso. Una caratteristica della memoria dinamica è inoltre che più siamo vicini ai **valori soglia** 0 o 1, meno si riesce a distinguere il dato presente in

memoria. In conclusione, quindi, quando facciamo una lettura dalla memoria la linea di parola viene messa ad 1 e viene rilevato lo stato del condensatore che ci passerà il dato. Per quanto riguarda il *costo in termini hardware*, abbiamo che qui abbiamo il costo di 1 transistor e 1 condensatore: sicuramente molto meno rispetto ad avere 6 transistor come nella memoria statica, ma con prestazioni meno efficienti.

7.2.4 Organizzazione dei collegamenti della RAM

Abbiamo visto che per selezionare una riga e una colonna di una matrice di memoria abbiamo bisogno di numerosi collegamenti. Ciò porta ad avere costi maggiori e inoltre dal punto di vista fisico non è molto utile aggiungere collegamenti. Per questo possiamo cercare di **ridurre il numero di collegamenti**. Supponiamo di avere un chip da 256 Mbit con 8 colonne, ovvero una matrice $32M \times 8$, dove 32M sono le linee di riga. Tale memoria è organizzata in una matrice $16K \times 16K$ ovvero 2048 gruppi da 8 bit. Abbiamo quindi che dovremmo utilizzare 11 collegamenti per selezionare 1 dei possibili gruppi ($2^{11} = 2048$) e avremmo bisogno di 14 collegamenti per selezionare una delle possibili 16k righe ($2^{14} = 16k$): il totale di collegamenti sarebbe $11 + 14 = 25$ collegamenti. Per ridurre tale numero splittiamo gli indirizzi di riga e di colonna attraverso specifici segnali di controllo RAS (decodificatore di riga) e CAS (decodificatore di colonna), rispettivamente che prendono gli indirizzi per la riga e gli indirizzi per la colonna. In tal modo si manda prima l'indirizzo di riga (14 collegamenti) al RAS e poi l'indirizzo di colonna (11 collegamenti).

7.2.5 Moduli di memoria

Per ottenere una capacità di memoria molto più ampia si utilizzano una serie di chip disposti in una *matrice di chip*. Tale matrice andrà poi a comporre il **modulo di memoria**.

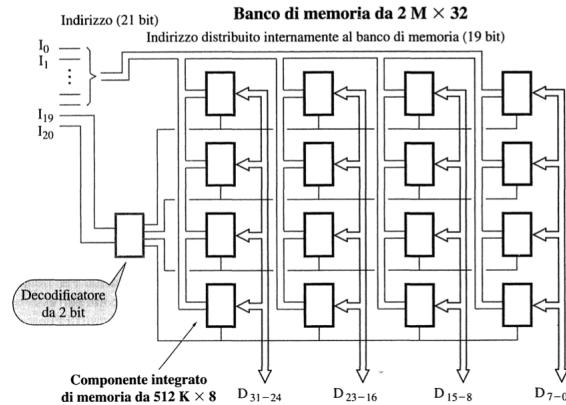


Figure 7.4: Banco di memoria 2M x 32

Supponiamo di avere, come in figura, l'organizzazione di un modulo $2M \times 32$ bit con chip da $512k \times 8$ bit: per indirizzare $2M$ parole di memoria (1 parola= 32 bit), ci serviranno esattamente 21 bit di cui:

- 19 bit selezionano 1 parola all'interno dei 4 chip ($2^{19} = 512$, indirizziamo quindi 1 riga di 512);
- 2 bit vengono mandati a un segnali di controllo chiamato **chip select**, che seleziona la riga attraverso il decodificatore di riga.

Con tale organizzazione quindi riusciamo ad avere una capacità di memoria molto più ampia.

7.3 Gerarchia di memoria

Abbiamo compreso fino ad adesso che in base a *tipologia, costo, velocità, capacità...* esistono *diversi tipi di memoria* che sono organizzati nel calcolatore secondo una **gerarchia di memoria**:

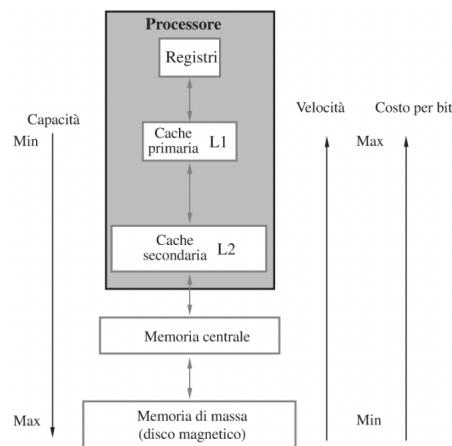


Figure 7.5: Gerarchia di memoria

Come possiamo notare dall'immagine sopra, nella scala gerarchica delle memorie, la capacità di memoria man mano che ci avviciniamo al processore diminuisce e in maniera *inversamente proporzionale* aumenta la velocità e il costo: quindi il fatto che la capacità diminuisca man mano che aggiungiamo memorie più veloci è data proprio dal fattore costo. Ad esempio, prendendo in esame la **cache**, che è una memoria più lenta del processore ma sicuramente più veloce della memoria centrale, essa presenta più **livelli** proprio per questa gerarchia per il quale avere un'unica cache di livello L1, quindi la più veloce, comporterebbe un enorme costo. Infatti esistono per questo più livelli di cache, che fanno sì che il costo sia comunque non troppo elevato, ed evitare che il processore vada a cercare il dato in memoria centrale perdendo troppo tempo.

7.4 Memoria cache e località

Il funzionamento e lo scopo della cache di basano sul principio fondamentale di **località**. In particolare abbiamo 2 principi di località:

- **Località temporale**: si basa su una statistica fatta su moltissimi programmi che indica che al loro interno sono presenti *molti cicli*. La conseguenza è proprio il principio di **località temporale** che dice che se accediamo a una certa locazione di memoria è molto probabile (cicli) che accederemo in tempo breve nuovamente alla stessa locazione.
- **Località spaziale**: si basa anche questo su statistiche fatte in larga scala, che indicano che molto spesso nei programmi vengono utilizzati *vettori*, quindi leggiamo molto spesso locazioni di memoria contigue. L'uso dei vettori ha come conseguenza il principio di località spaziale che dice che se accediamo a una certa locazione di memoria, molto probabilmente accederemo a una locazione di memoria adiacente.

Inoltre tali principi si verificano molto spesso contemporaneamente: se scorriamo un array, avrò sicuramente bisogno di un ciclo che mi permetta di prendere le locazioni di memoria contigue. Riformulando, quindi, i 2 principi possiamo dire che con probabilità elevata che entro breve tempo saranno usate più parole di memoria o parole a breve distanza, ovvero, piccoli gruppi di parole di memoria sono usate più volte entro breve tempo.

7.4.1 Uso della cache

Vediamo adesso come i principi di *località* si riflettono effettivamente sull'uso della cache. Quando il processore emette un indirizzo esso non sa dell'esistenza della cache. Succede, quindi, che quando mi servirà una parola di memoria, trasferirò dalla memoria centrale un **blocco** di parole di memoria contenenti anche l'indirizzo inviato dal processore e, per la conseguenza del principio di località, in tal modo un indirizzo che si trova in locazioni di memoria adiacenti, che molto probabilmente il processore richiederà a breve, si troverà in cache e possa essere fornito velocemente. Ogni blocco in cache, inoltre, occupa una **posizione di cache** e quando il processore usa una qualsiasi delle parole del blocco la trova già in cache. Ovviamente però lo spazio in cache è limitato: per questo se una posizione è occupata e ha bisogno di essere liberata per un trasferimento di un blocco dalla memoria centrale, verrà liberata tramite appositi **algoritmi di sostituzione** (LRU).

7.4.2 Cache hit

Quando il processore emette un indirizzo vi saranno determinati circuiti di controllo che esaminano se la parola di memoria richiesta si trova in cache. Se tale parola si trova in cache, si verifica una **cache hit**, ovvero la cache contiene il dato ricercato. In particolare abbiamo il caso in cui il processore voglia fare una lettura o una scrittura:

- In caso di lettura, avremo una **cache read hit**, ovvero la cache contiene il dato e la memoria centrale non viene coinvolta, e, quindi il dato verrà fornito dalla cache;
- In caso di scrittura, avremo una **cache write hit**, e si procede in 2 modi:
 - **Scrittura immediata**: se la cache contiene l'indirizzo viene scritto in memoria centrale.
 - **Scrittura differita**: in questo caso non viene scritto subito il dato in memoria, ma si aggiorna un *bit di modifica*, che quando la posizione in cache dovrà essere liberata, aggiornerà la memoria centrale col dato nuovo

Per quanto riguarda la scrittura viene utilizzata molto spesso la **scrittura differita** poichè, dato il fatto che molto spesso viene aggiornata la cache (incremento indice ciclo for), è più efficiente scrivere successivamente il dato in memoria centrale.

7.4.3 Cache miss

Abbiamo visto il caso in cui la parole di memoria richieste dal processore si trovino in cache (cache hit). Può capitare però che gli indirizzi mandati dal processore non si trovino in cache, con una conseguenza di **cache miss**. In tal caso il processore dovrà attendere che il blocco venga caricato in cache dalla memoria centrale e inserito in una posizione di cache:

- In caso di lettura, si può avere una **read back** (lettura differita), ovvero il blocco prima viene caricato interamente in una posizione di cache e poi viene letta la parola di memoria richiesta dal processore. Oppure si può procedere con una **load through**, ovvero non si attende per l'intero caricamento del blocco, ma viene letta la parola appena viene caricata in cache.
- Se l'operazione è di scrittura e la parola non si trova in cache si ha una miss di scrittura di cache, ovvero **cache write miss** (o semplicemente miss). Se si adotta la scrittura immediata (**write through**), la parola viene scritta subito in memoria centrale, oppure con la scrittura differita (**write back**) il blocco viene caricato in cache e quindi subito dopo viene aggiornato.

7.4.4 Indirizzamento di cache

Vediamo adesso come avviene il trasferimento di dati dalla memoria centrale alla cache. In particolare esistono tre tipi di indirizzamento:

- **Indirizzamento diretto**
- **Indirizzamento associativo**
- **Indirizzamento associativo a gruppi**

Indirizzamento diretto

Analizziamo l'immagine a seguire che raffigura un esempio di **indirizzamento diretto**:

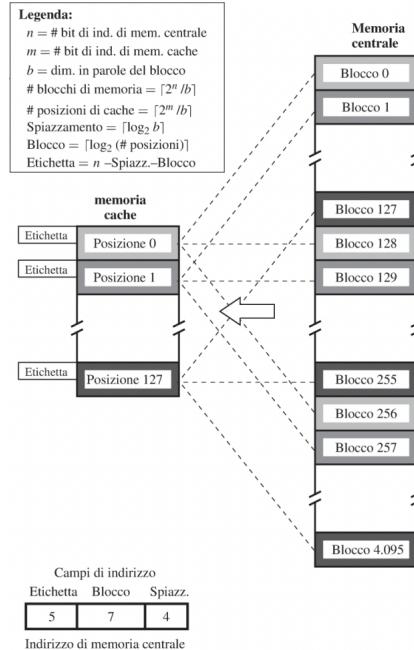


Figure 7.6: Indirizzamento diretto

Notiamo che la memoria centrale è suddivisa in **blocchi**, nell'esempio 4096 blocchi, in cui sono raggruppate le varie parole di memoria. Nell'esempio dell'immagine assumiamo che ogni blocco contenga 16 parole. Quindi abbiamo un totale di $4096 \times 16 = 64K$ parole di memoria e $4K$ blocchi nella memoria centrale. Suddividiamo invece la cache in **128 posizioni**, che potranno contenere ciascuna un blocco di memoria, con un totale di $128 \times 16 = 2K$ blocchi. Nel caso dell'**indirizzamento diretto** ogni blocco di memoria può essere caricato in una sola posizione in cache, che è data dal resto della divisione per 128 ($i \bmod 128$). Per cui ogni blocco a distanza di 128 posizioni in memoria centrale occuperà in *momenti diversi* la stessa posizione in cache. Per effettuare una decisione di hit/miss di cache, l'indirizzo della parola di memoria, che occupa 16 bit, viene suddiviso nei cosiddetti **campi di indirizzo** e sono 3:

- **Campo blocco:** occupa esattamente 7 bit e individua 1 di 128 posizioni in cache
- **Campo spiazzamento:** occupa esattamente 4 bit e individua la parola di memoria all'interno del blocco

- **Campo etichetta:** occupa esattamente 5 bit e individua l'etichetta del blocco in cache.

Per effettuare quindi una decisione di hit/miss viene presa esattamente la posizione del blocco in cache presente nel campo blocco dell'indirizzo di memoria centrale, e si confronta il campo etichetta dell'indirizzo con l'etichetta presente in cache: se sono uguali allora il dato è presente in cache (hit), altrimenti non è presente in cache (miss).

Indirizzamento associativo

Con l'indirizzamento diretto abbiamo il problema che il processore emette indirizzi che possono riferirsi a blocchi in memoria centrale a distanza di 128 posizioni, ma i 2 blocchi dovranno occupare la stessa posizione in cache, e questo può rappresentare un conflitto non banale. Si è pensato per questo a un **indirizzamento associativo** nel quale ogni blocco in memoria centrale può occupare una qualsiasi delle 128 posizioni in cache.

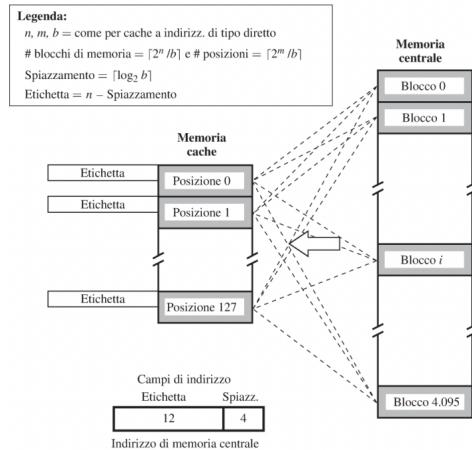


Figure 7.7: Indirizzamento associativo

Se mi servono 128 blocchi, potrò metterli tutti contemporaneamente in cache, o altrimenti, dovrò liberare qualche posizione in cache attraverso appositi **algoritmi di sostituzione**. L'indirizzo di memoria, nel caso di indirizzamento associativo, viene suddiviso solamente in 2 campi:

- **Campo spiazzamento:** occupa 4 bit e indica dove si trova la parola all'interno del blocco
- **Campo etichetta:** occupa 12 bit e indica quale etichetta della cache corrisponde al campo etichetta.

In tal caso, quindi, dovremo confrontare ogni etichetta della cache, poiché non sappiamo la posizione che ogni blocco può occupare in cache. Si fa quindi

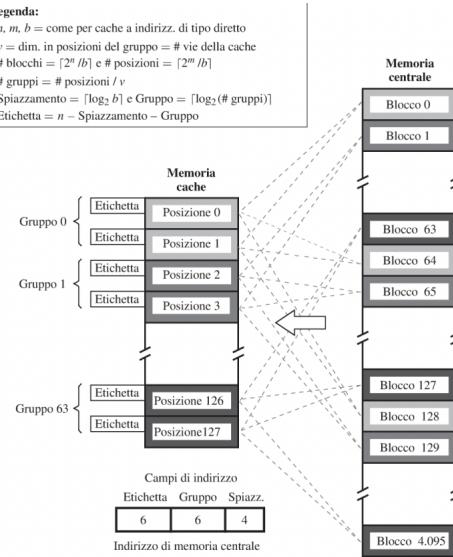


Figure 7.8: Indirizzamento associativo a gruppi

una **ricerca associativa**, ovvero vengono confrontate in *parallelo* tutte le etichette della cache. Ciò, se da un lato ci porta il vantaggio di non avere delle posizioni ben specifiche per ogni blocco, ci porta lo svantaggio di dover avere molti circuiti hardware in più, con un conseguente costo maggiore.

Indirizzamento associativo a gruppi

Abbiamo visto che dal punto di vista dell'hardware è molto complicato avere un indirizzamento associativo che sia efficiente. Per questo esiste l'**indirizzamento associativo a gruppi**. essenzialmente la cache è suddivisa in gruppi, e ogni gruppo ha un certo numero di posizioni (n vie). Ogni gruppo può essere occupato, come nell'indirizzamento diretto, da specifici blocchi in memoria. Ad esempio il blocco 0 può essere messo solo nel gruppo 0, in posizione 0 o 1 (se il gruppo è a 2 vie), lo stesso vale per il blocco 128. In tal modo riduco il conflitto dell'indirizzamento diretto per la ricerca di un dato, e effettuerò un numero molto minore di confronti delle etichette. In tal caso l'indirizzo della memoria centrale viene suddiviso nei seguenti 3 campi:

- **Campo gruppo:** occupa 6 bit individua il gruppo corrispondente in cache
- **Campo spiazzamento:** occupa 4 bit individua la parola all'interno del blocco
- **Campo etichetta:** individua l'etichetta all'interno del gruppo.

Notiamo quindi che avendo a disposizione i gruppi, dovremo fare un numero di confronti in parallelo molto minore e avremo un costo hardware sicuramente

minore rispetto ad avere un indirizzamento associativo. Inoltre in base a quante posizioni può occupare il gruppo, la cache si dice a **n vie**, ovvero ci sono n posizioni all'interno del gruppo.

7.4.5 Dati scaduti

Per la memoria cache si pone un problema di **coerenza di dati**. Oltre al processore che può leggere dati dalla memoria centrale vi è un dispositivo, detto **DMA**, che si occupa di trasferire dati dalla memoria centrale alla periferica o disco e viceversa e "alleggerisce", quindi, il lavoro del processore. Il problema, quindi, si pone perché ovviamente abbiamo più dispositivi che lavorano la memoria e potrebbe capitare che il processore aggiorni la cache e il DMA aggiorni la memoria centrale. Per questo a ogni posizione della cache è associato un **bit di validità**. Quindi nella **scrittura di blocchi della memoria centrale** da parte del DMA, i corrispondenti dati in cache devono essere marcati *non validi*, poiché contengono appunto dati scaduti. Nel caso invece di **lettura di blocchi dalla memoria** da parte del DMA, in questo caso la **cache deve essere svuotata**, in modo che i cambiamenti più recenti siano copiati in memoria centrale con un'operazione del sistema operativo.

7.4.6 Algoritmi di sostituzione

Abbiamo visto che nel caso di indirizzamento diretto la posizione che i blocchi di memoria possono occupare la cache sono ben delineati. Nel caso, invece, dell'indirizzamento associativo o associativo a gruppi, può capitare di dover sostituire blocchi di memoria per sostituire un blocco dalla memoria. Per questo si utilizza un algoritmo **LRU**, che si basa essenzialmente sulla sostituzione del blocco usato meno recentemente. Per una memoria cache associativa a gruppi, con 4 posizioni per gruppo (a 4 vie), per ogni posizione (del gruppo) vi sarà un **contatore da due bit** (modulo 4), inizialmente vale 0. Se si ha una *cache hit*, il contatore della posizione interessata va messo a 0, i contatori del gruppo con valore minore (prima dell'azzeramento) sono incrementati di 1. Se si ha una *cache miss* e si ha una posizione libera nel gruppo, si carica il blocco, il contatore della sua posizione vale zero e gli altri contatori sono incrementati di 1. Se si ha una cache miss e il gruppo è pieno, la posizione con contatore massimo (valore 3) va liberata e riempita col nuovo blocco, il contatore corrispondente vale 0, e gli altri contatori vanno incrementati di 1. Si può dimostrare che in ciascun gruppo i quattro contatori sono sempre diversi fra loro.

Part I

APPENDICE

Appendice A

Sistemi di numerazione e rappresentazione binaria dei numeri

Ogni calcolatore lavora con dati in codice binario ma nella quotidianità noi siamo abituati ad utilizzare un sistema di numerazione **decimale**: questo è un sistema **posizionale**, ovvero, ogni simbolo deve occupare una posizione ben precisa, infatti la cifra più a destra viene chiamata *unità*, quella a sinistra dell'unità *decina* ecc. Noi con 10 simboli riusciamo a rappresentare ogni tipo di valore numerico. Ciò non vale ad esempio nel sistema di numerazione *romano*, che è **additivo** ovvero abbiamo bisogno di diversi simboli che appunto si sommano tra loro per dare il numero finale. Nessuno ci vieta però di usare sistemi di numerazione con più o meno simboli.

Esempio dell'abaco Se prendessimo un abaco dentro il quale possiamo mettere 10 palline, che consideriamo quindi come le cifre che vanno da 0 a 9, e dovessimo rappresentare il numero 12; poichè non possiamo mettere 12 palline tutte in una stessa asta dovremmo spostarne una a sinistra, che rappresenterebbe la *decina*, e due a destra che rappresenterebbero le due *unità* rimanenti da aggiungere alla decina. Ovviamente esiste anche l'abaco-2, contenente al più una pallina, l'abaco-8, l'abaco-16: ciò significa che noi possiamo rappresentare un numero con diversi sistemi numerazione con base diversa.

A.1 Sistemi di numerazione posizionali

A.1.1 Conversione di qualsiasi n in base B in base 10

Ogni sistema di numerazione posizionale è composto quindi da:

- Una **base** detta **B**

- Un insieme di **B simboli**

Un numero a n cifre con qualsiasi base può essere rappresentato in decimale secondo tale formula:

$$\sum_{i=0}^{n-1} (p_i * B^i) = P_{n-1} * B^{n-1} + P_{n-2} * B^{n-2} + \dots + P_0 * B^0 \quad (\text{A.1})$$

Esempio

$$(1100)_2 = P_3 * B^3 + P_2 * B^2 + P_1 * B^1 + P_0 * B^0 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = (12)_{10} \quad (\text{A.2})$$

A.1.2 Conversione di qualsiasi n in base 10 in un'altra base B

Per convertire qualsiasi decimale in un n si eseguono i seguenti passaggi:

- Si prende il numero in questione e si eseguono **divisioni successive** tra i quozienti utilizzando come divisore la base B a cui vogliamo arrivare e conserviamo i resti
 - **Ordiniamo** il resto da sotto a sopra

Esempio Convertiamo 13 in base 2:

13	
6	1
3	0
1	1
0	1

Table A.1: Conversione di 13 in base 2

In questo caso abbiamo i **quozienti** delle divisioni successive a sinistra e a destra il **resto** delle varie divisioni¹, che vanno ordinati da sotto verso sopra per dare il numero convertito. Quindi $(13)_{10} = (1101)_2$

A.1.3 Numero di valori rappresentabili

Il numero di possibili **combinazioni** con **n** cifre e base **B** è uguale: $[0, B^n) = 0x < B^n$.

Esempio Possibili combinazioni di un binario con tre cifre: abbiamo quindi $n = 3$ e $B = 2$, di conseguenza $B^n = 8$. Abbiamo 8 possibili combinazioni di numeri binari con 3 cifre.

¹**NB:** ricorda che i resti possono variare da 0 a B-1, quindi nel caso del binario possono essere solo 0 e 1

1	000
2	001
3	010
4	011
5	100
6	101
7	110
8	111

Table A.2: Esempio di possibili combinazioni

A.1.4 Conversioni di binari in esadecimale e viceversa

Molto spesso conviene a noi informatici convertire un binario in esadecimale e viceversa poichè sono le basi che più utilizziamo. Esiste infatti un metodo per convertire direttamente numeri che hanno queste basi. In realtà la condizione per attuare tale metodo è quella di avere un numero con **base** che è pari a una **potenza di 2**, ad esempio $2^2, 2^3, 2^4\dots$

Se dobbiamo convertire in questo caso un esadecimale in binario ogni cifra dell'esadecimale va codificata in **4 bit** di binario, poichè $16 = 2^4$. Al contrario se dobbiamo convertire un binario in esadecimale, dovremmo prendere 4 bit del binario e convertirli in una cifra dell'esadecimale; in questo caso è importante prendere le cifre sempre a blocchi di 4, ma ad esempio se dovessimo convertire un ottale, $8 = 2^3$, in questo caso dovremmo considerare un blocco di 3 bit, ecc.

Esempio Convertiamo $(F13C)_{16}$ ²:

1. $C = 12 = 8 + 4 = 2^3 + 2^2 + 0 * 2^1 + 0 * 2^0 = 1100$;
2. $3 = 2 + 1 = 0 * 2^3 + 0 * 2^2 + 2^1 + 2^0 = 0011$;
3. $1 = 2^0 = 0001$
4. $F = 8 + 4 + 2 + 1 = 2^3 + 2^2 + 2^1 + 2^0 = 1111$;

Unendo i vari gruppi da 4 bit avremo che: $(F13C)_{16} = (1111000100111100)_2$.

A.2 Rappresentazione dell'informazione

Il calcolatore è composto da una serie di **circuiti** che quindi presentano diverse **tensioni**, che equivalgono alla codifica di valori binari 0 (bassa tensione) e 1 (alta tensione). Ogni tipo di dato, valore numerico, immagine, istruzione viene impartito dal calcolatore attraverso una **sequenza** ben precisa di 0,1. È fondamentale comprendere quindi che ogni istruzione, ad esempio la somma, ha la sua sequenza di codice binario che è diversa da istruzione a istruzione.

²Nel sistema esadecimale le cifre che vanno da 10 a 15, visto che non abbiamo dei simboli appropriati, vengono indicati con le lettere che vanno da **A(10)** a **F(15)**

A.2.1 Somma di due binari

Dopo aver fatto questa premessa capiamo effettivamente come avvengono le operazioni tra binari partendo dalla **somma**. Abbiamo 4 casi:

- Somma di $0 + 0$:

$$\begin{array}{r} 0 \\ 0 \\ \hline 0 \end{array}$$

Table A.3: Somma di $0+0$

- Somma di $1 + 0$:

$$\begin{array}{r} 1 \\ 0 \\ \hline 1 \end{array}$$

Table A.4: Somma di $0+0$

- Somma di $0 + 1$:

$$\begin{array}{r} 0 \\ 1 \\ \hline 1 \end{array}$$

Table A.5: Somma di $0+0$

- Somma di $1 + 1$:

$$\begin{array}{r} 1 \\ 1 \\ \hline 1 \ 0 \end{array}$$

Table A.6: Somma di $0+0$

In quest'ultimo caso $1 + 1 = 2$ utilizziamo il riporto 1 che poi sommato con 0 ci da a destra 0 e a sinistra 1.

A.2.2 Somma modulare

Abbiamo considerato fino a questo momento di poter fare una somma senza avere un limite nell'utilizzo dei bit, ma nella maggiorparte dei casi noi abbiamo a che fare con un numero limitato di bit (ad esempio se la parola di memoria ha una lunghezza di 32 bit dobbiamo utilizzare al massimo 32 bit per rappresentare

la somma e il risultato). Proprio per questo è nata la **somma modulare**, ovvero quella funzione che, presi due valori $mod(x, y)$ o $x \text{ mod } y$ ci restituisce il **resto** della divisione tra i due. Se dovessimo rappresentare questa somma modulare in una circonferenza vedremmo che ciclerà sempre gli stessi valori, ovvero da 0 a $y-1$. Esempio: $5 \text{ mod } 2 = 1$. Se dovessimo utilizzare quindi 3 bit, ovvero 2^3 valori, potremmo rappresentare coi bit i valori da 0 a 7.

A.2.3 Rappresentazione dei numeri relativi

Rappresentazione dei bit con i relativi

Si pone adesso un problema con i numeri relativi poichè dobbiamo porre anche un segno, in questo caso si procede in questo modo:

- Il primo bit a sinistra rappresenta il **segno**, 0 = bit di segno “-” e 1 = bit di segno “+”
- I restanti bit si rappresentano come numero naturale

Esempio Se quindi dovessimo rappresentare i valori da 3 bit, sappiamo che i valori rappresentati saranno 8, ovvero da -3 a 3, considerando il bit di segno. Immaginando i bit rappresentati su una circonferenza possiamo immaginare che fissato il bit di segno 0, scorriamo i numeri da 0 a 3; fissato il bit di segno 1, scorriamo i valori da -0 a -3. Vediamo adesso un esempio con 2^4 valori rappresentabili:

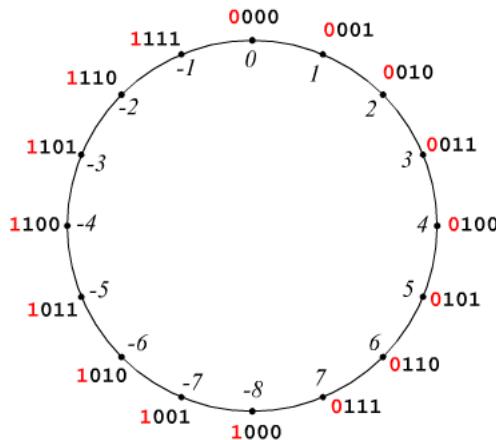


Figure A.1: complemento a 2

Complemento a 1

La rappresentazione vista appena prima è però molto scomoda per eseguire operazioni in binario, per cui si è deciso per rappresentare i numeri **negativi** invertendo il bit segno e le cifre a seguire che rappresentano il numero naturale, in modo tale da facilitare i calcoli. Se lo vedessimo all'interno della circonferenza è come se si invertissero le posizioni dei numeri con segno meno.

Esempio Un esempio di complemento a 1, sempre utilizzando 3 bit, può essere la trasformazione da 1 a -1: $1 = 001$; $-1 = 110$, abbiamo invertito tutte le cifre di bit

Complemento a 2

Il complemento a 2 è il più semplice poichè basta aggiungere 1 al risultato del complemento a 1. Esempio: $-1 = 110$ (complemento a 1); $-1 = 110 + 1 = 111$ (complemento a 2). La somma o la differenza, in questo caso col complemento a 2, avviene quindi normalmente trascurando però il **riporto in uscita**

A.2.4 Trabocco

Abbiamo quindi compreso che i **numeri naturali rappresentabili** dati n bit è da 0 a 2^{n-1} , mentre i numeri interi rappresentabili è da -2^{n-1} a $2^{n-1} - 1$. Nella somma e nella differenza inoltre se abbiamo ovviamente un numero fissato di bit si trascura il **riporto in uscita**. Si pone però un problema quando però con una somma o una differenza si ha un risultato che non è compreso nell'intervallo degli interi rappresentabili (da -2^{n-1} a $2^{n-1} - 1$); in tal caso si dice che è avvenuto il **trabocco** che può avvenire solamente quando si verificano due condizioni:

- Gli **addendi** sono **concordi** di segno, ovvero quando il bit di segno è uguale
- Il **bit di segno** è **discorde** rispetto agli addendi, ovvero a un bit di segno diverso

A.3 Rappresentazione numeri con la virgola in binario

Per la rappresentazione dei numeri **frazionari** utilizziamo sempre il bit di segno, rappresentiamo la parte intera e quella decimale. Questa rappresentazione viene chiamata a **virgola fissa** o *fixed point*, ovvero noi sappiamo esattamente le cifre della parte intera e di quella decimale. Si procede in questo modo: Dati 5 bit abbiamo il seguente numero " $b_4b_3b_2.b_1b_0$ ", rappresentiamo la parte intera partendo da 2^0 e la parte decimale da 2^{-1} : $b_4 \cdot 2^2 + b_3 \cdot 2^1 + b_2 \cdot 2^0 + b_1 \cdot 2^{-1} + b_0 \cdot 2^{-2}$. Inoltre più è grande la parte decimale più l'intervallo di valori rappresentabili sarà grande e viceversa più la parte decimale è grande più l'intervallo di valori

sarà piccolo. Esempio: $b^4b^3b^2b^1b^0$. → possiamo rappresentare i valori da 0 a 2^{n-1} ; $b^4b^3b^2b^1.b^0$ → possiamo rappresentare i valori da 0 a $2^{n-1} - 2^1$ (togliamo la potenza della parte decimale).

A.3.1 Conversione da decimale a binario

Per rappresentare un decimale in binario si suddivide sempre la parte intera con quella decimale e si procede nel seguente modo:

- Per la parte **intera** si fanno le divisioni successive della base prendendo il resto
- Per la parte **decimale** si moltiplica quest'ultima per la base prendendo la parte intera risultante fino a quando arriva a risultare 0.0

A.3.2 Rappresentazione dei numeri in virgola mobile

La rappresentazione a virgola fissa vista in precedenza molto spesso non è sufficiente per i calcoli scientifici; per questo viene utilizzata la rappresentazione in **virgola mobile** o *floating point*. Questo tipo di rappresentazione è caratterizzata dallo spostamento della virgola, appunto virgola mobile, per aumentare l'intervallo di calcoli possibili. Questo "spostamento della virgola" si ottiene **normalizzando** ogni valore numerico di qualsiasi base. Esempio: $6.25345 \times 10^2 = (625.345)_{10}$; $1.0011 \times 2^2 = (100.11)_2$, quindi portiamo la virgola ad avere una sola cifra a sinistra e l'intera parte decimale a destra.

Segno, mantissa e esponente Per rappresentare un numero in virgola mobile in bit abbiamo bisogno essenzialmente di tre componenti:

- **Segno**: che può essere 0 se il numero è positivo; 1 se il numero è negativo;
- **Mantissa**: che rappresenta le cifre significative dopo la virgola, escludendo sempre l'1 prima della virgola. In una rappresentazione a 32 bit la mantissa avrà lunghezza di 23 bit, mentre in una rappresentazione a 64 bit la mantissa avrà lunghezza di 52 bit;
- **Esponente**: che viene codificato in 8 bit con uno spazio di rappresentazione di 32 bit e in 11 bit con uno spazio di rappresentazione di 64 bit. I valori che l'esponente può assumere vengono shiftati di 127 in una rappresentazione a 32 bit: per cui se pensiamo che $-126 \leq e \leq 127$ (esponente effettivo), l'esponente codificato in bit sarà uguale: $e = e' - 127$ (esponente rappresentato in bit) con $1 \leq e' \leq 254$ poiché escludiamo 0 e 255 che sono **valori speciali**. La stessa cosa vale per la rappresentazione dell'esponente a 64 bit; ovviamente però avendo più bit a disposizione shiftiamo l'esponente di 1023 per cui si avrà $e = e' - 1023$

Valori speciali dell'esponente Abbiamo alcuni valori dell'esponente chiamati **valori speciali** che sono:

- $e'=0; m=0$, quando abbiamo l'esponente e la mantissa uguali a 0 il numero rappresenta lo 0 esatto
- $e'=255; m=0$, quando abbiamo l'esponente uguale a 255 e la mantissa uguale a 0 il numero rappresenta ∞
- $e'=0; m \neq 0$, rappresenta la forma non normale ± 0
- $e'=255, m \neq 0$, rappresenta le forme indeterminate (**NaN**)

A.3.3 Rappresentazione dei caratteri in binario

Quando abbiamo un certo numero di bit a disposizione assegniamo a ogni bit un **carattere** diverso, per cui con n bit a disposizione possiamo rappresentare 2^n caratteri. Una delle codifiche più utilizzate per i caratteri è la codifica **ASCII**, che si basa su una rappresentazione con 7 bit, quindi 128 valori rappresentati. La peculiarità di tale codifica è che i codici di lettere e numeri sono impostati in modo tale che vadano in ordine crescente

Caratt.	Byte	Caratt.	Byte	Caratt.	Byte	Caratt.	Byte
NUL	00000000	SPACE	00100000	@	01000000	'	01100000
SOH	00000001	!	00100001	B	01000001	a	01100001
STH	00000010	-	00100010	C	01000011	b	01100010
EXT	00000011	#	00100011	D	01000100	c	01100011
EQT	00000100	\$	00100100	E	01000101	d	01100100
EQN	00000101	%	00100101	F	01000110	e	01100101
ACK	00000110	&	00100110	G	01000111	f	01100110
BEL	00000111	.	00100111	H	01001000	g	01100111
BS	00001000	(00101000	I	01001001	h	01101000
HT	00001001)	00101001	J	01001010	i	01101001
LF	00001010	*	00101010	K	01001011	j	01101010
VT	00001011	+	00101011	L	01001100	k	01101011
FF	00001100	,	00101100	M	01001101	l	01101100
CR	00001101	-	00101101	N	01001110	m	01101101
SO	00001110	.	00101110	O	01001111	n	01101110
SI	00001111	/	00101111	P	01010000	o	01101111
DLE	00010000	0	00110000	Q	01010001	p	01110000
DC1	00010001	1	00110001	R	010100010	q	01110001
DC2	00010010	2	00110010	S	01010011	r	01110010
DC3	00010011	3	00110011	T	01010100	s	01110011
DC4	00010100	4	00110100	U	01010101	t	01110100
NAK	00010101	5	00110101	V	01010110	u	01110101
SYN	00010110	6	00110110	W	01010111	v	01110110
ETB	00010111	7	00110111	X	01011000	w	01110111
CAN	00011000	8	00111000	Y	01011001	x	01111000
EM	00011001	9	00111001	Z	01011010	y	01111001
SUB	00011010	:	00111010	^	01011011	z	01111010
ESC	00011011	:	00111011	°	01011100	ò	01111011
FS	00011100	<	00111100	\	01011101	ó	01111100
GS	00011101	=	00111101	^	01011110	ú	01111101
RS	00011110	>	00111110	_	01011111	ñ	01111110
US	00011111	?	00111111	DEL			01111111

Figure A.2: CODIFICA ASCII

Appendice B

Algebra booleana

L'**algebra booleana** è la parte algebrica che i calcolatori utilizzano ad esempio nei **circuiti logici**, nel quale ogni operazione viene fatta con **variabili binarie** che possono assumere valori 0 o 1.

B.1 Le operazioni fondamentali dell'algebra booleana

Nell'**algebra booleana** ci sono 3 operazioni fondamentali:

- **Somma logica** o **OR**
- **Prodotto logico** o **AND**
- **Complementazione** o **NOT**

Possiamo dire che nell'algebra booleana attraverso queste 3 operazioni possiamo scrivere qualsiasi **funzione logica**. Andiamo a vedere adesso queste 3 operazioni cosa effettivamente producono.

B.1.1 Somma logica (OR)

La **somma logica** prende in ingresso due variabili e si denota tramite gli operatori "+" o " \vee "; avrà quindi forma algebrica: $x_1 + x_2 = x_1 \vee x_2$. Il risultato sarà sempre 1, tranne nel caso in cui entrambe le variabili in ingresso valgono 0, infatti la tabella di verità sarà:

x_1	x_2	$x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

Table B.1: SOMMA LOGICA

Proprietà della somma logica Come per la somma algebrica, molte proprietà valgono per la somma logica:

- **Proprietà commutativa:** $x_1 + x_2 = x_2 + x_1$
- **Proprietà associativa:** $(x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$
- **Estensione** a più variabili: $x_1 + x_2 + \dots + x_n$
- **Elemento neutro** (0): $0 + x_1 = x_1$

B.1.2 Prodotto logico (AND)

Il **prodotto logico** prende in ingresso due variabili e si denota tramite gli operatori “.” o “ \wedge ”; avrà quindi forma algebrica: $x_1 \cdot x_2 = x_1 \wedge x_2$. A differenza della somma logica, il prodotto logico da sempre risultato 0, tranne nel caso in cui entrambe le variabili in ingresso valgono 1, e quindi ha tabella di verità:

x_1	x_2	$x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Table B.2: PRODOTTO LOGICO

Proprietà del prodotto logico Valgono le stesse proprietà della somma logica:

- **Proprietà commutativa:** $x_1 \cdot x_2 = x_2 \cdot x_1$
- **Proprietà associativa:** $(x_1 \cdot x_2) \cdot x_3 = x_1 \cdot (x_2 \cdot x_3)$
- **Estensione** a più variabili: $x_1 \cdot x_2 \cdot \dots \cdot x_n$
- **Elemento neutro** (1): $1 \cdot x_1 = x_1$

B.1.3 Negazione (NOT)

La **negazione** prende in ingresso una variabile e restituisce il valore opposto. Si denota con \bar{x} oppure $\neg x$ e ha tabella di verità:

x	\bar{x}
0	1
1	0

Table B.3: NEGAZIONE

B.1.4 Proprietà duali di AND e OR

Oltre alle proprietà precedentemente elencate la somma e il prodotto logico possiedono ulteriori proprietà:

- **Proprietà distributiva:** a differenza dell'algebra di base vale sia per la somma che per il prodotto logico: $x + y \cdot z = (x + z) \cdot (x + y)$; $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
- **Idempotenza:** $x + x = x$; $x \cdot x = x$
- **Complemento:** $x + \bar{x} = 1$; $x \cdot \bar{x} = 0$
- **Proprietà dell'1 e 0:** $1 + x = 1$; $0 \cdot x = 0$

B.1.5 Teoremi di De morgan

Per quanto riguarda la negazione abbiamo i teoremi di De morgan che affermano che **la negazione di una somma è il prodotto delle negazioni** e viceversa **la negazione di un prodotto è la somma delle negazioni**

B.2 Funzioni logiche ed espressioni logiche

Una **funzione logica** è una funzione con una o più variabili in entrata e una sola variabile in uscita epuò essere espressa da una e una sola **tabella di verità**; un'**espressione logica** è una delle possibili realizzazioni della funzione logica, infatti un'espressione logica rappresenta una e una sola funzione logica mentre una funzione logica è rappresentata da infinite espressioni logiche. Si rivela molto utile ricavare da una funzione logica, un'espressione logica e viceversa da un'espressione logica una funzione logica.

B.2.1 Equivalenza tra espressioni logiche

Per **confrontare** due espressioni logiche, e di conseguenza se sono uguali basta dimostrarlo con la tabella di verità. Esempio: dimostriamo che $x_1 x_2 = (x_1 + x_2)(\bar{x}_1 + x_2)(\bar{x}_1 + x_2)$

x_1	x_2	$x_1 x_2$	$(x_1 + x_2)(\bar{x}_1 + x_2)(\bar{x}_1 + x_2)$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Table B.4: Equivalenza tra espressioni logiche

In questo caso vediamo come ad esempio, le tabelle di verità delle 2 espressioni sono uguali, quindi anche le espressioni stesse. Quindi per sapere quale **funzione** è espressa da un **espressione logica** basta calcolarne la tabella di verità.

B.2.2 Da una funzione a espressione logica

Abbiamo compreso che se volessimo trovare una funzione logica equivalente all'espressione data basterebbe calcolare la tabella di verità di quest'ultima; il problema si pone quando dobbiamo passare da una funzione a un'espressione logica poichè esistono **infinite espressioni** che rappresentano una funzione. Per questo esistono dei "metodi" che ci permettono di trovare un'unica espressione logica data una funzione logica:

Prima forma canonica o SOP

Il primo metodo per passare da una funzione logica a un'espressione è chiamato **prima forma canonica**. Prima di comprendere come si sviluppa; dobbiamo comprendere cosa sia un **mintermine**. I mintermini sono funzioni che valgono 1 **solo** per gli ingressi che danno come risultato finale della funzione 1. La **somma logica dei mintermini** quindi da come risultato la funzione di partenza:

Somma di Mintermini

- $F_1 = m_1 + m_4 + m_7$

- $F_1 = \overline{x} \overline{y} z + x \overline{y} \overline{z} + x y z$

x y z	index	$m_1 + m_4 + m_7 = F_1$
0 0 0	0	0 + 0 + 0 = 0
0 0 1	1	1 + 0 + 0 = 1
0 1 0	2	0 + 0 + 0 = 0
0 1 1	3	0 + 0 + 0 = 0
1 0 0	4	0 + 1 + 0 = 1
1 0 1	5	0 + 0 + 0 = 0
1 1 0	6	0 + 0 + 0 = 0
1 1 1	7	0 + 0 + 1 = 1

Figure B.1: Somma logica di mintermini

Un mintermine quindi è rappresentato dal prodotto delle variabili in entrata in **forma diretta** se valgono 1; in **forma negata** se valgono 0; poichè ricordiamo che il prodotto logico restituisce 1 solamente se tutte le variabili moltiplicate valgono 1. Quindi una funzione logica viene rappresentata dalla **somma**

di **prodotti** o **SOP**, ed è quella che viene chiamata **prima forma canonica**. Facciamo un esempio in cui abbiamo tre variabili in entrata x_1, x_2, x_3 e la funzione f_1

x_1	x_2	x_3	f_1
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1

Table B.5: SOP

Vediamo in questa tabella di verità come si procede per trovare un'espressione logica data la funzione con la SOP:

- Vediamo le righe nelle quali la funzione vale 1, ovvero dove esiste il mintermine;
- Una volta trovate queste righe sappiamo che la funzione è uguale alla somma dei **prodotti** dei mintermini;
- Dobbiamo quindi prima trovare i prodotti dei mintermini, ricordandoci di utilizzare la forma diretta se la variabile vale 1, la forma negata se vale 0. Per la prima riga: $\bar{x}_1\bar{x}_2\bar{x}_3$; per la seconda riga: $\bar{x}_1\bar{x}_2x_3$; per la quarta riga: $x_1\bar{x}_2\bar{x}_3$;
- Una volta trovati i prodotti che danno come risultati i mintermini; li **sommiamo** così da trovare l'espressione logica equivalente alla funzione: $\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3$

Seconda forma canonica o POS

Il secondo metodo per passare da una funzione logica a un'espressione logica viene chiamato **seconda forma canonica** o **POS**, ed si sviluppa in modo opposto alla **SOP**. In questo caso dobbiamo trovare i **maxtermini**, che sono funzioni che valgono 0 solo per gli ingressi che danno come risultato finale della funzione 0. In questo caso il **prodotto logico dei maxtermini**, a differenza della SOP che è la somma logica dei mintermini, è uguale alla funzione di partenza. Un maxtermine è rappresentato, a differenza di un mintermine, dalla somma logica delle variabili in ingresso dalla somma della variabili in **forma diretta** se valgono 0; in **forma negata** se valgono 1. Riprendiamo l'esempio di prima e vediamo come svolgere la POS:

- Vediamo le righe nelle quali la funzione vale 0, ovvero dove esiste il maxtermine;
- Una volta trovate queste righe sappiamo che la funzione è uguale al prodotto della **somma** dei maxtermini;

x_1	x_2	x_3	f_1
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1

Table B.6: POS

•

- Dobbiamo quindi prima trovare la somma dei maxtermini, ricordandoci di utilizzare la forma diretta se la variabile vale 0, la forma negata se vale 1. In questo caso l'unico maxtermine si trova nella terza riga: $x_1 + \bar{x}_2 + x_3$
- Se ci fossero stati altri maxtermini si faceva il prodotto tra tutti i maxtermini

B.3 Forma minima

Molto spesso per rappresentare circuiti logici, conviene rappresentare sempre le espressioni logiche in **forma minima**, ovvero con il minor **costo** possibile. Per costo di una espressione possiamo intendere il numero di variabili che compone quest'ultima: $(x_1 + x_2) + (x_1 + \bar{x}_2) + (\bar{x}_1 + x_2)$ ha un costo 6.

B.3.1 Passaggio da SOP a forma minima

Per passare da una SOP a forma minima dobbiamo eseguire 3 passaggi utilizzando le proprietà delle operazioni logiche:

- **Proprietà distributiva:** raggruppiamo la variabile di differenza tra 2 prodotti utilizzando la distributiva: $\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 = \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3)$
- **Legge del complemento:** $(\bar{x}_3 + x_3) = 1 \rightarrow \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3) = \bar{x}_1\bar{x}_2$
- **Legge di idempotenza:** serve per duplicare i mintermini, qualora nella somma di prodotti ci fosse un termine isolato: $x_1x_2x_3 + x_1x_2x_3 = x_1x_2x_3$

B.3.2 Metodo di Karnaugh

Molto spesso ci capita di avere a che fare con espressioni molto complesse e minimizzarle può diventare un processo molto lungo. Per questo è stato inventato il **metodo di Karnaugh**, che è un metodo **geometrico** che ci permette di trovare le forme minime molto più semplicemente. Costruiamo una **mappa** in cui se abbiamo 3 variabili; sarebbe una matrice 2×4 ; mentre con 4 variabili una 4×4 .

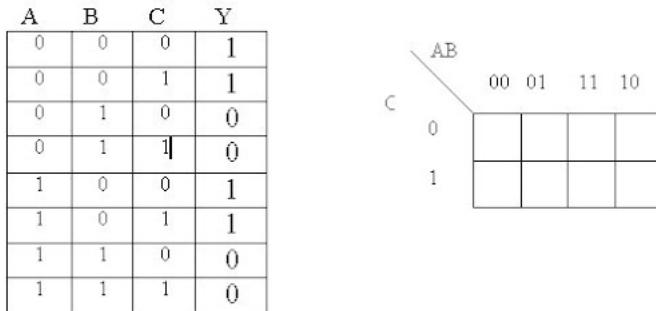


Figure B.2: MAPPA DI KARNAUGH A 3 VARIABILI

Procedimento di minimizzazione con le mappe Come vediamo gli indici delle colonne sono ordinati in modo che le variabili AB varino di **un solo valore**. Ogni casella verrà poi riempita col valore che la funzione restituisce per quei determinati ingressi. Se dobbiamo passare da una **SOP** a forma minima, si prenderanno i **gruppi di 1** che formano una **linea retta**, o un **quadrato** oppure che sono **adiacenti** dai lati con la **grandezza massima possibile** e **numero di gruppi minore possibile**; infine si vedrà per ogni valore che la funzione restituisce quali valori degli indici restano uguali: se l'indice è 1 (per la SOP) si scriverà la variabile in **forma diretta**, se l'indice è 0 (per la SOP) la variabile si scriverà in **forma negata**. Una volta scritte le variabili di un gruppo si farà il prodotto fra loro e si sommerà per gli eventuali valori degli altri gruppi. Per la **POS** il procedimento è **opposto**, come abbiamo mostrato nella costruzione del prodotto di somme, quindi: si prendono i gruppi di 0; si prendono gli indici uguali e se hanno 0 la variabile verrà scritta in forma diretta, 1 in forma negata; si sommano le variabili fra loro; e infine si fa il prodotto fra i vari gruppi.

Condizione di indifferenza Spesso può capitare che, per alcune entrate di una funzione, essa non restituisca un'uscita 0 o 1. In tal caso nella mappa di Karnaugh porremo delle "x". Questa **variabile di indifferenza** può assumere 0 o 1. Sceglieremo, ovviamente, il valore che ci permetterà di **minimizzare** il più possibile la funzione; vale a dire sceglieremo in base a SOP o POS, se scegliere rispettivamente 1 o 0 in modo tale da formare gruppi di dimensione maggiore possibile e il minor numero possibile.

B.4 Circuiti logici

Le operazioni logiche di base (AND, OR, NOT) sono realizzate tramite **circuiti elettronici**. Questi circuiti vengono chiamati **porte** che collegate tra loro formano una **rete combinatoria**. Ogni porta logica ha la sua **rappresentazione**

grafica nel quale appunto sono rappresentate le variabili in entrata e la funzione in uscita:

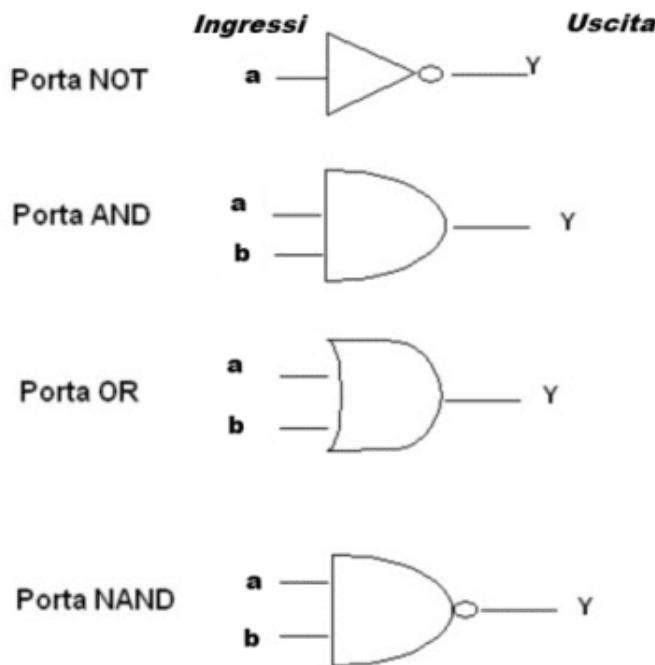


Figure B.3: Rappresentazione porte logiche

B.4.1 Porte a più ingressi

Le porte AND o OR possono estendere i propri ingressi a più di 2 e, grazie alla proprietà **associativa**, ritornare a 2 ingressi. Inoltre ogni circuito può essere suddiviso in **livelli**, in cui, i livelli più a sinistra sono quelli con la **priorità** più **alta**, ovvero quelli eseguiti prima, quelli a destra quelli con priorità più **bassa**. Una SOP, ad esempio, è un'espressione a **due livelli** nel quale, il primo livello è costituito dai prodotti fra variabili (mintermine) e il secondo è la somma tra i vari prodotti (somma di mintermini).

B.4.2 XOR, NAND e NOR

Esistono altre porte logiche che sono molto utili che utilizziamo per molte funzioni e sono: **XOR**, **NAND** e **NOR**.

XOR Lo "XOR", chiamato "*or esclusivo*" e denotato con l'operatore " \oplus ", restituisce nella funzione 1, se il numero di 1 in ingresso è dispari; restituisce 0, se il numero di 1 in ingresso è pari. Infatti dati x_1 e x_2 avrà tabella di verità:

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Table B.7: XOR

La rappresentazione **grafica** è la seguente:

$$X = A \oplus B$$

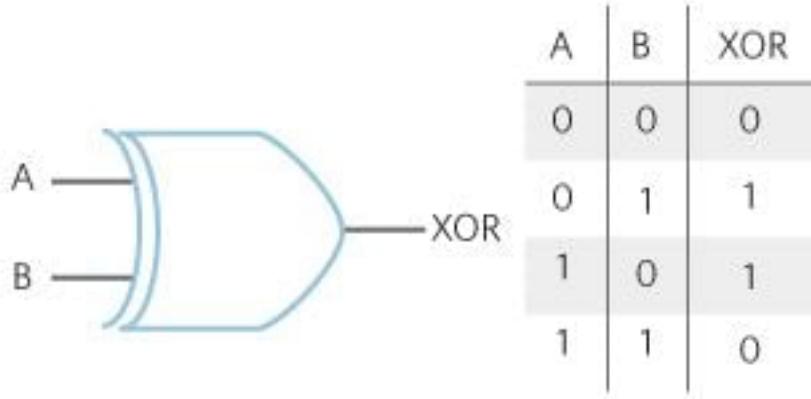


Figure B.4: Rappresentazione XOR

NAND e NOR In precedenza nella figura B.3, abbiamo notato che appariva nella rappresentazione la **porta NAND** che insieme alla **NOR**, viene molto utilizzata nei circuiti logici. "NAND" e "NOR" sono rispettivamente l'equivalente dell'**and negato** e dell'**or negato**, e si denotano con la simbologia " \uparrow " (NAND) e " \downarrow " (NOR). Inoltre la porta NAND restituisce il valore 0 nella funzione solo quando entrambe le variabili in ingresso sono 1; mentre la porta NOR restituisce 1 solo quando entrambe le variabili sono 0. Le tabelle di verità sono quindi:

x_1	x_2	$\neg(x_1 \cdot x_2)$	$\neg(x_1 + x_2)$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Table B.8: NAND e NOR

La loro rappresentazione del NOR in porta logica è come quella vista in figura B.3 per la NAND. Cambia semplicemente che la parte antestante la "palina" della negazione è come l'OR.

B.4.3 Porte universali

Abbiamo detto in precedenza che la NAND e la NOR sono porte logiche molto utilizzate. Questo poiché sono porte molto convenienti dal punto di vista della componentistica e della costruzione di conseguenza. Inoltre noi abbiamo la possibilità di rappresentare **qualsiasi funzione logica** solo con porte NAND e NOR e per questo vengono chiamate **porte universali**. Vediamo ad esempio come passare da una **SOP** a un'espressione di **soli NAND**:

$$\begin{aligned}
 & x_1 x_2 + x_3 x_4 = \\
 &= x_1 \bar{x}_2 + x_3 \bar{x}_4 = \\
 &= x_1 \uparrow x_2 + x_3 \bar{\uparrow} x_4 = \\
 &= (x_1 \uparrow x_2) \cdot (x_3 \uparrow x_4) = \\
 &= (x_1 \uparrow x_2) \uparrow (x_3 \uparrow x_4) =
 \end{aligned}$$

(B.1)

Analizziamo adesso i vari passaggi per comprendere come siamo passati da una SOP a una espressione di soli NAND:

- Nel primo passaggio abbiamo effettuato una **doppia negazione** di entrambi i fattori dei prodotti, poiché sappiamo che la doppia negazione è uguale al valore di partenza;
- Nel secondo passaggio abbiamo trasformato le negazioni dei prodotti, quindi la negazione dell'AND, in NAND;
- Per trasformare la somma in prodotto abbiamo utilizzato **De Morgan** che diceva che la somma di negazioni è uguale alla negazione del prodotto;
- Infine abbiamo trasformato l'ultimo prodotto negato anch'esso in NAND.

Se invece avessimo avuto una **POS**, avremmo trasformato allo stesso modo l'espressione in un'espressione di **soli NOR**

Appendice C

Circuiti integrati

Abbiamo ribadito diverse volte come ogni istruzione che il calcolatore fa è una **sequenza binaria** ben specifica. A livello fisico questo avviene in dei **circuiti elettronici** nel quale utilizziamo dei **livelli di tensione** per individuare i valori binari.

C.1 Rappresentazione delle variabili binarie

I circuiti accennati nell'introduzione, quindi, sono caratterizzati da una **tensione di voltaggio**:

- Se questa tensione è **alta**, allora il valore binario corrispondente sarà 1;
- Se questa tensione è **bassa**, allora il valore binario corrispondente sarà 0;

Ed inoltre, andando nello specifico su questo concetto, vi è una **soglia di tensione** per il quale tutti i valori di tensione più alti alla soglia rappresentano 1; mentre quei valori di tensione al di sotto della soglia rappresentano lo 0. I valori più prossimi alla soglia, chiamati **banda vietata**, però, vengono scartati per evitare l'incertezza data dal rumore del circuito.

C.2 Transistori

Oltre alla rappresentazioni delle variabili binarie, a noi serve **rappresentare le porte logiche**. Per questo ci serviamo di **interruttori**, ovvero i **transistori**, che possono essere in **stato di conduzione** se l'interruttore è chiuso e quindi il valore binario corrispondente sarà 1; oppure possono essere in **stato di interdizione** se l'interruttore è aperto e quindi il valore binario corrispondente sarà 0. Inoltre la tecnologia di costruzione più utilizzata per i transistori è la **tecnologia MOS**

C.2.1 Transistori MOS

I transistori MOS sono formati da **3 collegamenti**:

- Base;

- Sorgente;

- Pozzo

A seconda della tensione in ingresso della base il transistor collegherà o meno la sorgente al pozzo; inoltre se il transistor è in stato di conduzione la tensione del pozzo sarà uguale alla tensione della sorgente.

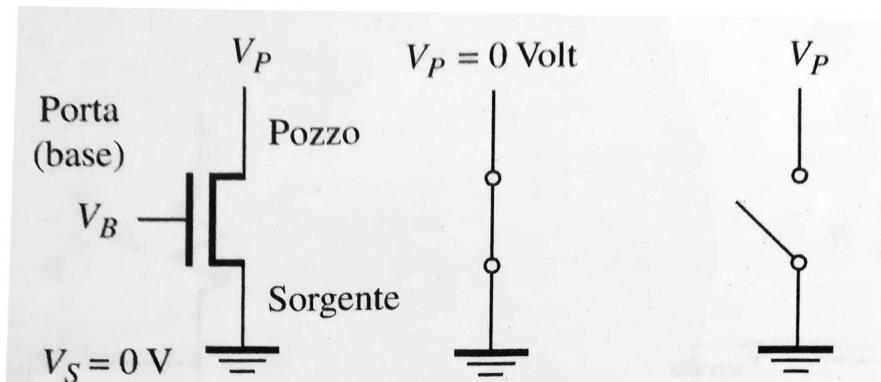


Figure C.1: Transistori MOS

Esistono 2 tipi di transistori MOS:

- Transistori **NMOS**: in questo tipo di transistor se la tensione di base è alta allora il transistor sarà in conduzione; se la tensione di base è bassa allora il transistor sarà in interdizione. La sorgente, invece, è collegata alla massa.

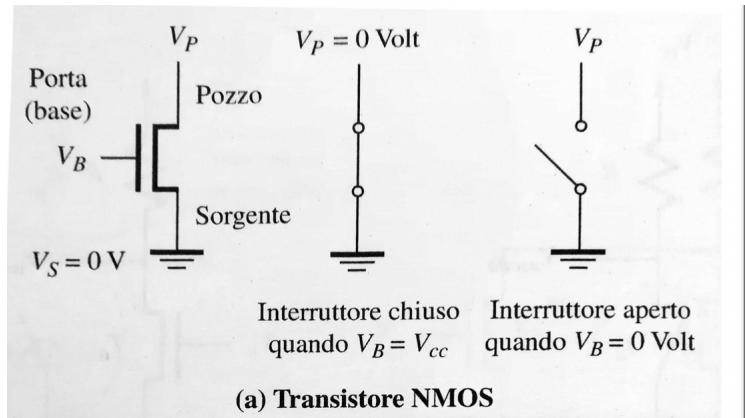


Figure C.2: Circuito NMOS

- **Transistori PMOS:** in questo tipo di transistor se la tensione di base è alta allora il transistor sarà in interdizione; se la tensione di base è bassa allora il transistor sarà in conduzione. La sorgente, invece, è collegata all'alimentazione.

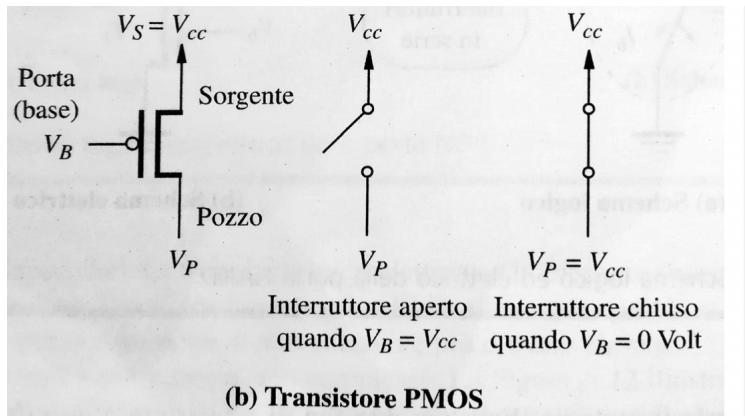


Figure C.3: Circuito PMOS

C.2.2 Circuiti NOT, NAND E NOR con transistor NMOS

Ovviamente tramite questi circuiti è possibile rappresentare, come avevamo detto, le varie porte logiche. Solitamente quelle utilizzate sono le **porte universali** NOT, NAND e NOR:

- **Circuito NOT:** Per ottenere un circuito NOT attraverso un transistor NMOS basta collegare la sorgente alla massa e l'alimentazione al pozzo. Se nella base abbiamo come **tensione di ingresso "1"**, quindi il **transistore**

è in **conduzione**, come **tensione in uscita** avremo "0" e viceversa, quando la tensione in ingresso è "0", il **transistore** è in **interdizione** e restituirà come tensione di uscita "1";

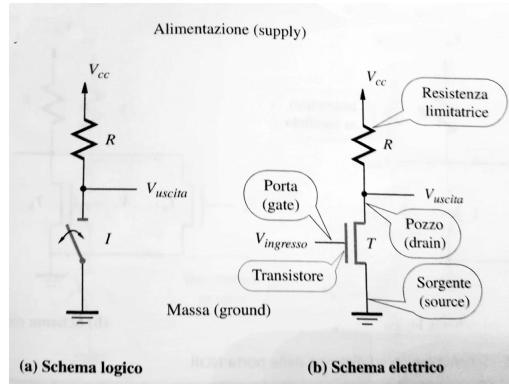


Figure C.4: Circuito NOT

- Circuito **NAND**: Per ottenere un circuito NAND attraverso un transistor NMOS basta collegare in serie 2 transistor. Se lo stato di entrambi i transistor è in **conduzione** (1), allora la tensione in uscita sarà "0"; per il resto dei casi la tensione in uscita sarà "1";

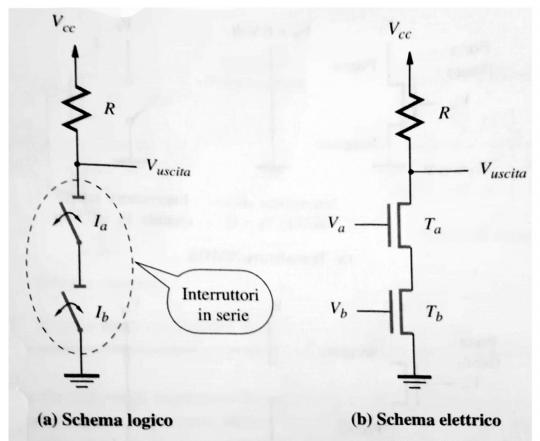


Figure C.5: Circuito NAND

- Circuito **NOR**: Per ottenere un circuito NOR attraverso un transistor NMOS basta collegare in parallelo 2 transistor. Se lo stato di entrambi i transistor è in **interdizione** (0), allora la tensione in uscita sarà "1"; per il resto dei casi la tensione in uscita sarà "0";

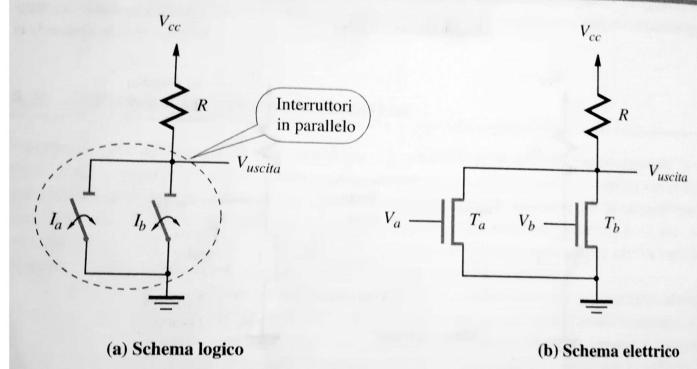


Figure C.6: Circuito NOR

C.3 Circuiti CMOS

I transistori NMOS hanno il problema che **consumano molta energia** in stato di conduzione a causa della **resistenza**. Per questo motivo solitamente si costruiscono transistori **CMOS** formati da una rete di **NMOS** o *rete di pull-down* (la cui sorgente è legata alla massa) collegata a una **PMOS** o *rete di pull-up* (la cui sorgente è legata all'alimentazione). In tal modo il comportamento dei due rami è **complementare** e in stato stabile non c'è mai continuità tra massa e alimentazione.

C.3.1 Da espressione logica a circuito CMOS

Per capire come rappresentare da un'espressione logica la rete di **pull-down** e **pull-up**, dobbiamo eseguire le seguenti operazioni:

- **RETE DI NMOS** (pull-down): bisogna fare il negato di ciò che vogliamo rappresentare;
- **RETE DI PMOS** (pull-up): bisogna negare solo le variabili in ingresso, ma non gli operatori

Esempio Vediamo un esempio pratico data l'espressione logica $(x_1 \cdot \neg x_2 \cdot x_3) + (x_2 \cdot \neg x_3)$:

- PULL DOWN:

$$\begin{aligned}
 (x_1 \cdot \neg x_2 \cdot x_3) + (x_2 \cdot \neg x_3) &= \overline{(x_1 \cdot \neg x_2 \cdot x_3)} + \overline{(x_2 \cdot \neg x_3)} = \\
 &= \overline{(x_1 \cdot \neg x_2 \cdot x_3)} \cdot \overline{(x_2 \cdot \neg x_3)} = \quad (\text{C.0}) \\
 &= (\neg x_1 + x_2 + \neg x_3) \cdot (\neg x_2 + x_3)
 \end{aligned}$$

- PULL UP:

$$\begin{aligned} & (x_1 \cdot \neg x_2 \cdot x_3) + (x_2 \cdot \neg x_3) = \\ & = (\neg x_1 \cdot x_2 \cdot \neg x_3) + (\neg x_2 \cdot x_3) \end{aligned} \quad (C.0)$$

- Per rappresentare il circuito nella rete di pull-down le variabili dovranno essere "legate" in **parallelo**, mentre la rete di pull-up le variabili dovranno essere "legate" in **serie**.

C.4 Ritardi di un circuito, fan-in e fan-out e porte tri-state

Quando costruiamo un qualsiasi tipo di circuito, andremo incontro a un **ritardo di propagazione** di quest'ultimo dovuto al fatto che da quando riceve dei dati in input avrà bisogno di un certo per **aggiornarsi** e poter ricevere nuovi dati. Quindi, il **tempo di transizione totale** di un segnale è dato dal cosiddetto **percorso critico**, ovvero dove il segnale impiega più tempo per uscire. Inoltre i circuiti hanno una certa **frequenza di lavoro** data dalle volte in cui il circuito commuta.

C.4.1 Fan-in e fan-out

Essenzialmente il concetto di **fan-in** e **fan-out** è molto semplice: infatti il **numero massimo di ingressi per porta logica** è chiamato fan-in, mentre il **numero massimo di uscite** è chiamato fan-out. Fan-in e fan-out incidono direttamente sul *ritardo di un circuito*, poiché ne aumentano il *rumore*, di conseguenza molto spesso le troviamo in numero limitato

C.4.2 Porte tri-state

Se provassimo a collegare in una porta **più uscite a uno stesso ingresso**, ci accorgeremmo di un corto circuito poiché si è impossibilitati a distinguere i valori in ingresso. Per questo esistono le **porte tri-state** che permettono, attraverso un **ingresso di controllo** e un **ingresso di segnale**, di avere 3 uscite (0,1,Z(alta impedenza)). In tal caso **quando l'ingresso di controllo è posto a 0** l'uscita sarà Z; **quando invece il l'ingresso di controllo è posto a 1**, l'uscita sarà quella dell'ingresso di segnale. In tal modo possiamo "scegliere" l'uscita della porta senza avere corti circuiti. Possiamo vedere anche con la seguente tabella di verità:

C.5 Circuiti integrati

Tutti i circuiti con porte logiche che abbiamo rappresentato graficamente, vengono raggruppati in **circuiti integrati**, che sono piastrine in silicio incapsulate

ingresso di controllo	ingresso di segnale	uscita di segnale
0	0	Z
0	1	Z
1	0	0
1	1	0

Table C.1: Porta tri-state

in un **involturco protettivo con dei morsetti esterni**, detti *pin*. Inoltre esistono diversi tipi di circuiti integrati in base alla **scala di integrazione**:

- Scala **piccola**: circuiti formati da **pochi porte logiche**;
- Scala **media**: addizzionatore, sottrattore, multiplatore...;
- Scala **Grande**: ALU, banco registri, piccoli processori..;
- Scala **molto grande**: memorie molto capaci, processori potenti...

C.5.1 Decodificatore (decoder) e multiplatore (multiplexer)

Tra i circuiti di *media scala* troviamo il **decodificatore** e il **multiplatore**:

- Il **decodificatore** prende **n ingressi** e **2^n uscite**. In base al numero binario codificato in ingresso (ES: $x_1 = 1, x_2 = 1$; *linea di uscita*=3), viene codificata appunto la **linea di uscita corrispondente**.
- Il **multiplatore** sceglie il segnale in ingresso da convogliare nell'uscita: infatti prende **n ingressi di controllo**, **2^n ingressi di segnale** e **un'uscita**. L'ingresso dato inoltre viene selezionato dalla configurazione degli ingressi di controllo ed è realizzabile come somma di prodotti.

Appendice D

ALU

Ricordiamo dal capitolo A dell'appendice, che quando sommavamo $1 + 1$, avevamo un **riporto in uscita**, che diventava **riporto in entrata** della cifra successiva. Adesso vediamo come costruire un **addizionatore completo** utilizzando sia riporto in uscita che in entrata. In seguito grazie a una "serie di addizionatori" costruiremo l'**ALU**, o *unità aritmetico-logica*

D.1 Addizionatore

D.1.1 Addizionatore a 1 bit

Un semplice **addizionatore a 1 bit**, che quindi sommi 2 singoli bit, è costituito da 2 funzioni logiche con 3 ingressi, che sono i **2 bit di ingresso e il riporto in entrata**, e le funzioni restituiscono:

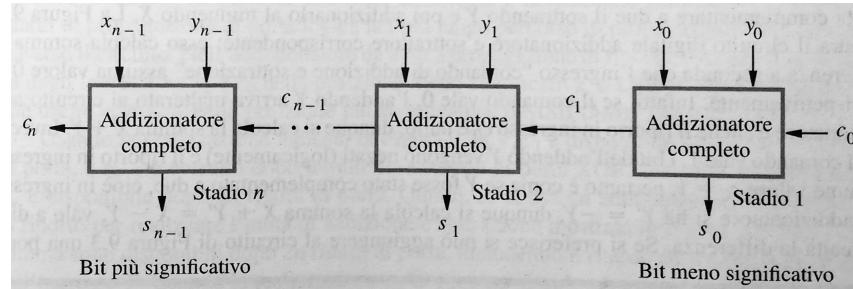
1. La prima uscita, che riguarda la prima funzione, restituisce la **somma dei bit e del riporto in entrata**;
2. La seconda uscita, che riguarda la seconda funzione, restituisce il **riporto in uscita della somma**

D.1.2 Addizionatore completo (full adder)

Adesso abbiamo fatto l'esempio di addizionatore a 1 bit; ma, collegando le reti logiche per le **funzioni somma e riporto in uscita** otteniamo un **addizionatore completo** che prende in ingresso i due bit da sommare e il riporto in entrata, e restituisce in uscita la somma e il riporto in uscita.

D.1.3 Addizionatore a propagazione di riporto

Qualora collegassimo una serie di **n addizionatori completi**, possiamo **propagare il riporto**, per **sommare numeri di n bit**. Tale addizionatore viene denominato **addizionatore a propagazione di riporto**:

Figure D.1: Addizionatore a propagazione di riporto c_1/c_n

D.2 Addizionatori algebrici

Ci siamo preoccupati fino ad ora di **somme di numeri interi**, però bisogna far sì che sia possibile anche l'operazione di **sottrazione**, per questo dobbiamo creare **addizionatori algebrici**

D.2.1 Trabocco

Possiamo anche avere la possibilità di fare una **somma con complemento a 2**, poichè ad esempio quando dobbiamo fare una **sottrazione**, dobbiamo **complementare a 2 il sottraendo**. La somma con complemento a 2, come ricordiamo dal capitolo A, avviene **trascurando il riporto in uscita**, e inoltre, bisogna garantire che non avvenga **trabocco**. Il trabocco in realtà viene calcolato come **xor** di due riporti consecutivi:

$$\text{trabocco} = c_n \oplus c_{n-1} \quad (\text{D.0})$$

D.2.2 Addizionatore algebrico a n bit

Avendo introdotto come **gestire il trabocco** dal punto di vista logico, possiamo considerare come **unità logica di addizione e sottrazione** attraverso un *addizionatore a propagazione* caratterizzato dal Bit **OpType₀**, che serve a **complementare a 2 il secondo addendo** qualora ci sia una **sottrazione**. Infatti quando viene posto a 1, questo bit, aggiunge un **riporto in ingresso** nel bit meno significativo, e, attraverso una serie di **xor paralleli** completerà a 2 il secondo addendo ($y_n \oplus \text{OpType}_0$). Il trabocco, come detto precedentemente viene calcolato come $c_n \oplus c_{n-1}$. Possiamo vedere la rappresentazione di un addizionatore algebrico, come descritto sopra, nella seguente immagine:

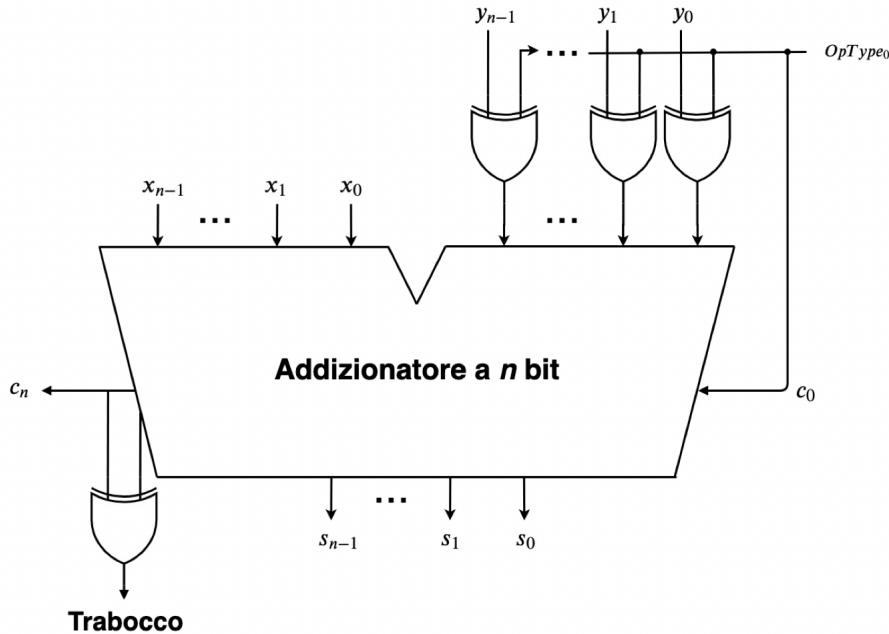


Figure D.2: Addizionatore algebrico a n bit

D.3 ALU

D.3.1 ALU a 1 bit

Abbiamo parlato fino ad adesso di somma e sottrazione. Se aggiungiamo le **operazioni logiche** AND, OR e NOT otteniamo un **ALU**. Per estendere la possibilità di scegliere tra un **sommatore** o una **porta logica** aggiungiamo un **multiplatore**, che attraverso gli ingressi di controllo **OpType₁**, **OpType₂** consente di mandare in output una delle seguenti linee di uscita:

- **Sommatore:** $x_i + y_i$;
- **Porta AND:** $x_i \text{ AND } y_i$;
- **Porta OR:** $x_i \text{ OR } y_i$;
- **Porta NOT:** $x_i \text{ NOT } y_i$

D.3.2 ALU a n bit

Collegando in serie n ALU a 1 bit in un'unità logica simile all'addizionatore visto in precedenza otterremo una **ALU a n bit** dove i **bit di controllo**

OpType₁, **OpType₂** servono a selezionare l'operazione da eseguire (addizione, AND, OR o NOT) e **OpType₀** serve a selezionare la sottrazione e complementare a 2 il sottraendo, come possiamo vedere nella seguente figura:

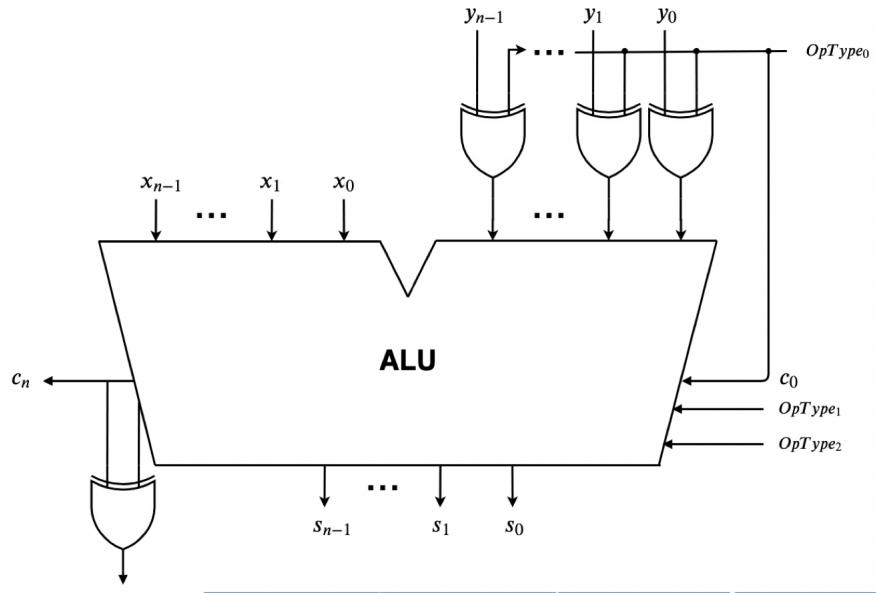


Figure D.3: ALU a n bit

Appendice E

Reti sequenziali

Nei capitoli precedenti abbiamo visto come le varie unità come ALU, multiplatore, decodificatore... sono costruite tramite **reti combinatorie**. Si pone un problema però quando dobbiamo conservare una quantità di dati tale per cui non basta la rete combinatoria, e, soprattutto abbiamo bisogno di un circuito che ci permetta di fare dei **cicli** per cui l'uscita attuale dipenda da "stati" precedenti. Tali reti vengono chiamate **reti sequenziali** e sono appunto *reti logiche* che presentano dei cicli

E.1 Bistabile asincrono

Le reti sequenziali, da quanto abbiamo capito, sono reti molto complesse e, partendo dal grado più in basso presentano dei **bistabili** che sono circuiti nei quali le *uscite sono ingressi di altre porte*, e, sono capaci di **memorizzare un solo bit**, che denotiamo in **forma diretta** come Q_a , e in **forma negata** come Q_b . Per rappresentare un bistabile, quindi, non è adatta una tabella di verità, ma in tal caso parleremo di **tabella di transizione**. In tal caso, analizziamo il bistabile in figura, che presenta prete logiche NOR:

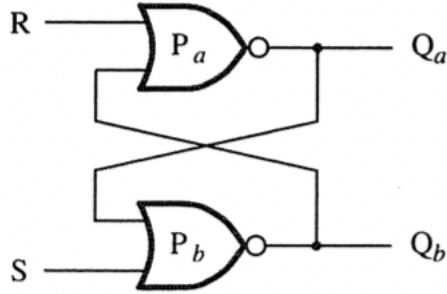


Figure E.1: Schema logico di un bistabile asincrono

Tale bistabile avrà **tabella di transizione**:

R	S	Q_a	Q_b
0	0	Q_a	Q_b
0	1	0	1
1	0	1	0
1	1	0	0

Table E.1: Tabella di transizione bistabile

Ragioniamo su come costruiamo tale tabella. Prendiamo il caso in cui $R = 1$ ed $S = 0$, per la porta P_a varrà 0, poichè essendo una porta NOR sappiamo con un solo valore 1 che l'uscita sarà 0, in tal caso l'uscita $Q_a = 0$. Vediamo che la porta P_b è **retroazionata in ingresso** da Q_a , di conseguenza, avendo $S = 0$ e $Q_a = 0$, avremo come uscita NOR 1, quindi $Q_b = 1$. Inoltre Q_b è un'uscita **retroazionata in ingresso** su P_a , di conseguenza scopriamo che gli ingressi di P_a sono $R = 1$ e $Q_b = 0$. Continuando con tale ragionamento arriveremmo a vedere a fine ciclo che Q_a potrà assumere se $R = 0, S = 0$ due valori 0/1, di conseguenza nella tabella di transizione scriveremo che Q_a ha uscita Q_a , poichè ricordiamo che i bistabili, alla base delle reti sequenziali, dipendono dal comportamento passato dei circuiti e ad esempio degli ingressi/uscite transitati in precedenza. Per il resto dei casi, dove si esclude il caso $R = 0$ ed $S = 0$, le uscite sono univocamente determinate in maniera esplicita. Inoltre notiamo che $Q_a = \bar{Q}_b$, cioè l'uscita Q_b è il complemento di Q_a , tranne nel caso $R = S = 1$, e, di conseguenza non aggiunge nulla di nuovo.

E.1.1 Diagramma temporale di un bistabile asincrono

Siccome la rete sequenziale ha comportamento in parte dipendente dal passato, ovvero dal tempo, spesso se ne rappresenta il funzionamento tramite **diagramma temporale** come fatto in Figura:

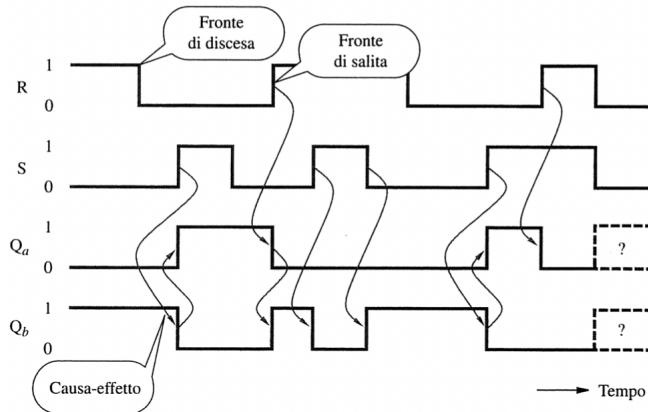


Figure E.2: Diagramma temporale

In tal caso i vari **fronti di transizione**, che possono essere *"in salita o in discesa"*, gli istanti di tempo nei quali le forme d'onda degli ingressi 0 e 1 transitano di valore (da 1 a 0: discesa, da 0 a 1: salita). I nomi S e R stanno per set e reset, ovvero ingresso di impostazione e di ripristino ; è inoltre convenzione comune chiamare Q (ovvero l'uscita Q_a) l'uscita del bistabile intendendone la forma diretta (benché se serve si possa usare anche la forma negata $Q - b$ che è inevitabilmente presente). Riassumendo il funzionamento del bistabile SR (bistabile set-reset), si vede quanto segue: attivando S a 1 si forza Q a 1 (si scrive 1 nel bistabile); attivando R a 1 si forza Q a 0 (si scrive 0 nel bistabile); lasciando sia S sia R disattivi a 0 l'uscita Q mantiene il valore logico che aveva assunto in precedenza, qualunque esso fosse. Ciò giustifica il nome bistabile, cioè circuito logico che a parità di ingressi ha **due stati di funzionamento** $Q = 0$ oppure $Q = 1$, vale a dire **un bit**¹, ed è capace di tenere memoria del bit corrente a tempo indefinito finché, manovrando gli ingressi, non si decida di cambiarlo attivamente.

E.2 Bistabile sincrono

Abbiamo parlato del bistabile asincrono, e, possiamo denotare, che ingressi e uscite non sono sincronizzati, poiché quando transita un ingresso si modifica l'uscita cambiando lo stato del bistabile. Se aggiungiamo un **segnale di clock** che scandisce il tempo di un ciclo costruiremo un **bistabile sincrono**. Se il segnale di clock vale 1, allora il bistabile si comporterà come bistabile asincrono, se il segnale di clock vale 0, è come se bloccassimo lo stato del bistabile al valore di bit corrente nonostante qualsiasi modifica facciamo a S o R. Tale bistabile avrà tabella di transizione molto simile a quella del bistabile asincrono, solamente con l'aggiunta del segnale di clock, dove Q_t rappresenta lo **stato attuale** e Q_{t+1} rappresenta lo **stato futuro**. Inoltre dal *diagramma temporale*, per come

¹Il bistabile mantiene in uscita **un solo bit**

abbiamo spiegato, quando il segnale di clock attraversa il fronte di transizione ovviamente cambia lo stato dello stabile. Lo possiamo vedere appunto in questa figura:

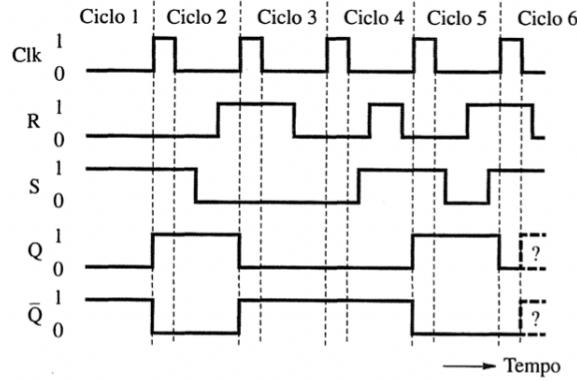


Figure E.3: Diagramma temporale bistabile sincrono

Inoltre, il bistabile sincrono, ha come *tabella di transizione*:

CLK	R	S	Q_{t+1}
0	x	x	Q_t
1	0	0	Q_t
1	1	0	1
1	0	1	0
1	1	1	x

Table E.2: Tabella di transizione bistabile sincrono

E.2.1 Bistabile di tipo D

I bistabili visti finora hanno sempre gli stessi ingressi di controllo S e R. Abbiamo compreso che $R = \bar{S}$, cioè R e S sono **complementari**. Per questo è stato creato un bistabile che **unifica gli ingressi** in un unico ingresso D , l'ingresso D viene inviato in forma diretta e negata sui due segnali interni che assolvono la funzione originaria di S e R, rispettivamente. Quando $CLK = 1$ l'uscita Q assume valore 1 o 0 se $D = 1$ o $D = 0$, rispettivamente, giacché è come dire $S = 1$ o $R = 1$. Un bistabile sincrono può transitare varie volte quando l'ingresso di clock è alto, cioè $CLK = 1$ tale effetto viene chiamato **trasparenza**, e, spesso tale effetto è *indesiderabile* poiché quando il segnale di clock è alto è come se non esercitasse funzioni in memoria, e, di conseguenza *mutamenti in ingresso si mutano immediatamente in mutamenti in uscita*. Ciò non dovrebbe avvenire ad esempio quando costruiamo registri a scorrimento, poiché non ci deve essere

propagazione di bit tra più bistabili durante un ciclo di clock. Possiamo vedere questo comportamento in tale figura:

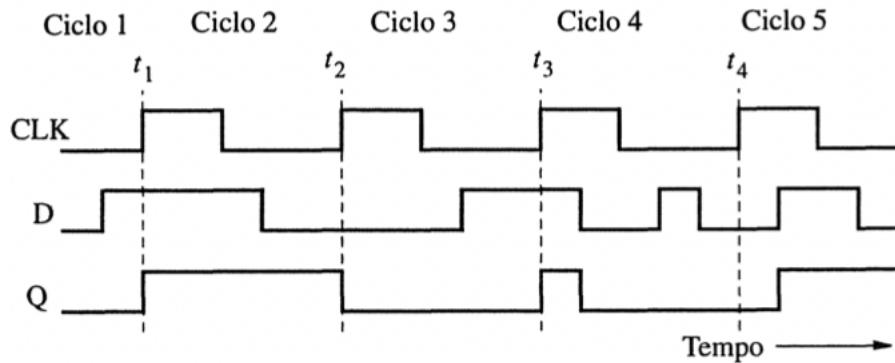


Figure E.4: Diagramma temporale bistabile tipo D

E.3 Flip-Flop

Esistono dei tipi di bistabili sincroni, che risolvono il problema del bistabile tipo D, che riescono a commutare solo in corrispondenza della transizione tra ciclo di clock corrente e consecutivo (transizione delimitata per esempio dal fronte di discesa dell'ingresso di clock), e sono comunemente chiamati **flip-flop**

E.3.1 Flip-Flop master-slave

Un *bistabile sincrono* che risolve il problema della trasparenza, come visto in figura E.4, infatti è formato da due bistabili di tipo D in serie, chiamati **principale (master)**, e **ausiliario (slave)**, che ricevono rispettivamente il **segnale di clock in forma diretta e negata**. In tal modo la transizione dei valori, avviene solo quando il segnale di clock è basso, cioè soltanto durante il fronte di discesa:

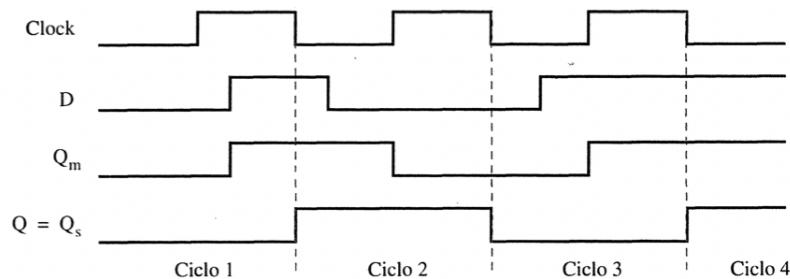


Figure E.5: Flip-flop di tipo D master-slave

E.3.2 Flip-Flop di tipo T

Il **flip-flop di tipo T**, presentano un'ingresso T, per il quale il flip-flop fa una transizione di stato ogni ciclo di clock se il suo **input T** è a 1, altrimenti viene riconfermato. A seconda del valore di T viene rimandata in ingresso ad un flip-flop di tipo D la sua uscita diretta o negata ed è molto utile per realizzare i contatori

E.3.3 Flip-Flop di tipo JK

Il flip-flop di tipo JK **unisce le funzionalità dei flip-flop di tipo D e T**. In esso sono presenti **due ingressi J,K** per il quale:

- Se lo stesso bit viene duplicato negli ingressi J e K si comporta come un **flip-flop T** (1 commuta lo stato e 0 lo riconferma);
- Se J e K sono **complementari** si comporta come un **flip-flop di tipo D**, dove l'input J corrisponde a D.

E.4 Registri paralleli e seriali

Un **registro** è formato da un insieme di *flip-flop*, i quali possono immagazzinare soltanto 1 bit, che in base alla loro costruzione si suddividono in:

- **Registri paralleli;**
- **Registri a scorrimento;**
- **Registri seriali-paralleli;**

E.4.1 Registri paralleli

I **registri paralleli** presentano n flip-flop collegati in *parallelo* ad uno stesso segnale di clock, e grazie ai quali possiamo accedere contemporaneamente a tutti gli ingressi e le uscite dei flip-flop. Possiamo vedere in figura un esempio:

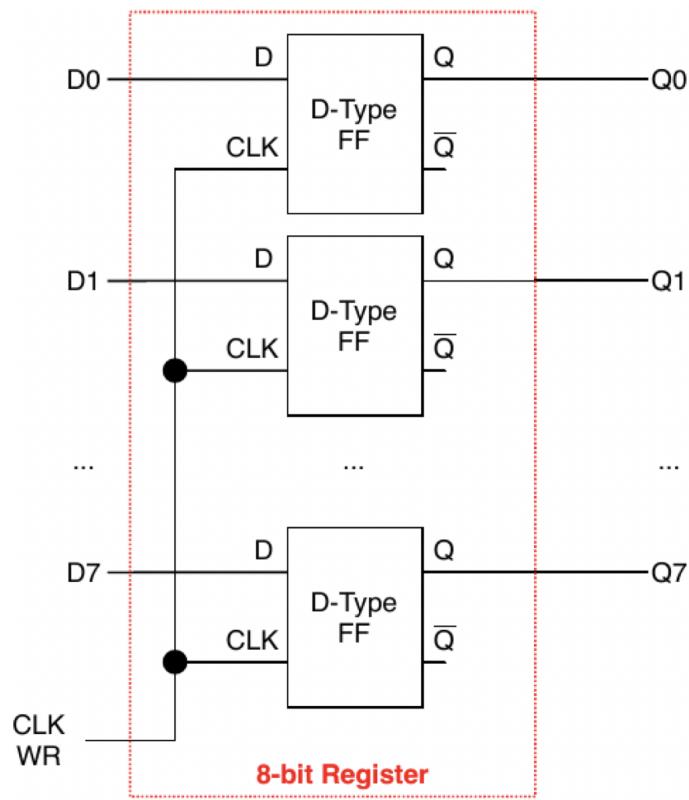


Figure E.6: Registro parallelo a 8 bit

E.4.2 Registri a scorrimento

I **registri a scorrimento** sono formati da n flip-flop collegati in serie. In tal caso il primo flip-flop riceve 1 bit in ingresso, e il contenuto di tale flip-flop si sposta a ogni ciclo di clock di una posizione. L'uscita del registro dipende dal bit di uscita dell'ultimo flip-flop:

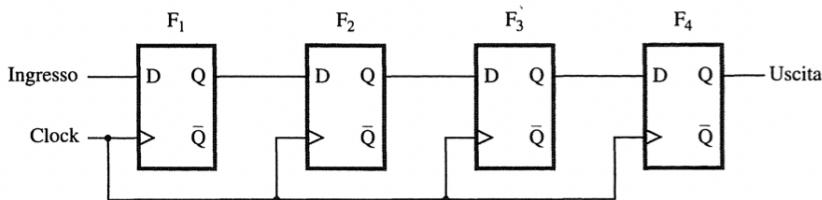


Figure E.7: Registri a scorrimento a 4 bit

E.4.3 Registri seriali-paralleli

Il registro **seriale-parallelo** unisce le funzionalità dei due registri visti finora. Infatti presenta un comando *shift/load* che permette di cambiare la modalità in ingresso:

- shift/load = 0: shift (registro a scorrimento);
- shift/load = 1: load (registro parallelo);

La modalità viene selezionata attraverso un *multipli* a un *ingresso di selezione* (shift/load):

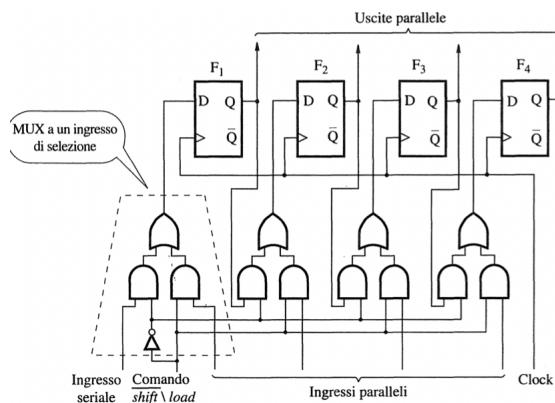


Figure E.8: Registro seriale-parallelo a 4 bit

E.5 Contatore

Un **contatore** a n bit è in grado di scorrere la sequenza di numeri da 1 a 2^{n-1} , aumentando, ad ogni ciclo di clock, di 1 in binario i suoi valori. È realizzato attraverso una serie di flip-flop T collegando in serie l'uscita $\neg Q$ di ogni flip-flop all'ingresso di clock del flip-flop successivo. Il numero attuale del conteggio è dato dalla stringa delle uscite Q (dal bit meno significativo al bit più significativo):

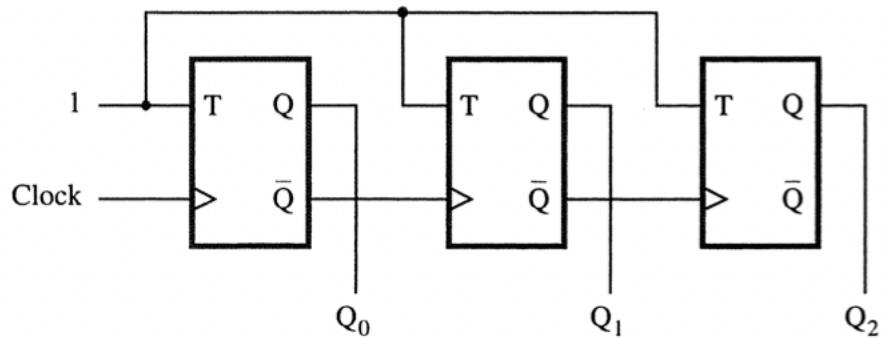


Figure E.9: Contatore